Estructuras de Almacenamiento de la Información

Libreta de prácticas
Curso 2009/2010

Manuel Barrena García

El laboratorio de E.A.I.

1. Introducción

Las prácticas de la asignatura EAI consisten en una serie de sesiones tutorizadas en el laboratorio *Novell* y un conjunto de ejercicios a desarrollar individualmente por cada alumno y alumna de la asignatura. De modo general, las prácticas versan sobre la programación en lenguaje C de una variedad de estructuras para el almacenamiento de la información que se estudian durante la fase teórica de la asignatura.

Para cubrir la totalidad de objetivos planteados en este curso práctico, es necesario que el alumno conozca en profundidad las bases de la metodología de programación impartidas en el transcurso de las asignaturas de ***Elementos de Programación*** *y* ***Estructuras de Datos y Algoritmos****, así como sus correspondientes asignaturas prácticas* ***Laboratorio de Programación I y II****. Aquellos alumnos que no hayan superado con éxito estas asignaturas encontrarán serias dificultades para cursar con aprovechamiento las prácticas que aquí presentamos.*

Antes de entrar a detallar los aspectos más relevantes de este curso práctico, conviene puntualizar que a lo largo del mismo no se impartirán recursos básicos de programación, tales como el uso adecuado de las estructuras del lenguaje (construcción de bucles, modularidad, funciones, recursión, etc.), la selección correcta y uso adecuado de las estructuras de datos (vectores, registros, listas, pilas, colas, árboles, etc.), la manipulación apropiada de punteros, etc. Esta puntualización no implica que en estas prácticas no se preste atención al uso correcto de todos y cada uno de los aspectos relacionados con la metodología de programación. Muy al contrario, en el espíritu de estas sesiones prácticas, la aplicación de los elementos que constituyen la metodología de programación aprendida en los cursos anteriores, juega un papel primordial. Así pues, no sólo nos centraremos en encontrar soluciones que “funcionen”, sino que tales soluciones deberán primeramente adecuarse a unos patrones de calidad acordes con el título para el que optan los alumnos matriculados en estas asignaturas.

Con el ánimo de orientar convenientemente al alumnado matriculado de esta materia, el presente documento pretende servir de guía básica para todos aquellos que hayan tomado la relevante decisión de cursar estas prácticas con objeto de aprender lo que en ellas se propone.

2. Organización

Las prácticas de EAI se organizan por grupos de 20 a 30 alumnos, los cuales compartirán laboratorio durante las sesiones planificadas. Estos grupos se constituirán en las primeras dos semanas de curso, durante las cuales los alumnos tendrán posibilidad de preparar y ensayar con el material que se relaciona en el presente documento.

A cada grupo se le asigna un periodo de dos horas fijas por semana en un día concreto, de modo que cada alumno podrá elegir el grupo cuyo horario más le convenga. Una vez constituidos los grupos y en función del número de alumnos del mismo y las disponibilidades materiales del laboratorio en ese momento, cada puesto de trabajo será compartido como máximo por dos alumnos. Por su parte, cada grupo estará tutorizado por un profesor de la asignatura, que llevará durante todo el curso el control de dicho grupo.

La duración real de cada sesión será de una hora y cuarenta y cinco minutos, dando así un colchón de quince minutos para el intercambio de usuarios del laboratorio.

Cada práctica cuenta con un material consistente en la documentación de la misma y algunos ficheros. Todo este material y cualquier otro relevante a la asignatura se encontrará a disposición permanente de los alumnos en la red local.

El curso consiste en un total de 5 prácticas, cada una con diferente duración. La documentación relativa al laboratorio de EAI está disponible (en formato pdf) al alumnado de la asignatura en la plataforma virtual. Se recomienda su impresión y encuadernación con objeto de consultarla en cualquier momento, tanto durante las sesiones de laboratorio, como en el ejercicio del trabajo personal.

En el transcurso de las sesiones de laboratorio, los alumnos materializarán parte del contenido de las prácticas, sin embargo el grueso del trabajo experimental está previsto para ser desarrollado personalmente en horario no presencial.

La inscripción a las prácticas supone un compromiso explícito por parte del alumno de aceptación de las condiciones en las que se desarrolla el curso, entre las que se establecen:

- Asistencia continuada a todas y cada una de las sesiones.
- Realización de los ejercicios propuestos tanto en el laboratorio como fuera de él.
- Aportación de soluciones originales y en ningún caso copias procedentes de compañeros, academias, etc.
- Participación activa en el transcurso de las sesiones de laboratorio, así como en las actividades que se propongan desde la plataforma virtual.

3. Materialización de ejercicios

Las prácticas de la asignatura EAI se materializan mediante la utilización del lenguaje de programación C. Aunque el número de primitivas que se utilizarán en el desarrollo de estas prácticas es particularmente pequeño, se requiere un conocimiento básico del lenguaje C en su totalidad con el fin de obtener máximo aprovechamiento de este curso.

El laboratorio “Novell” dispone del entorno de programación Eclipse, mediante el cual se plantearán las soluciones a los ejercicios propuestos. Este entorno integra un editor de textos, compilador, enlazador (*linker*), depurador y ejecutor de programas. Es preciso que el alumno tenga un conocimiento previo del uso de tal entorno de programación con antelación al comienzo de las prácticas.

Respecto al propio lenguaje de programación, el alumno de estas prácticas debe tener un conocimiento profundo sobre la construcción de las estructuras básicas de programación en lenguaje C, tales como:

- Constantes y variables.
- Datos y tipos simples de datos (char, int, long, unsigned, …)
- Entrada y salida de datos (scanf, printf)
- Composición alternativa (if, else, switch, case) e iterativa (for, while).
- Funciones y parámetros (prototipos, llamadas, parámetros por valor y por variable, recursión)
- Tipos estructurados (vectores, cadenas de caracteres, estructuras).
- Gestión y uso de la memoria dinámica (punteros).

Así pues antes de comenzar las prácticas, el alumno debe tener soltura en el uso de y cada uno de los puntos anteriormente destacados.

Para llevar un control sobre las habilidades adquiridas por el alumno en el desarrollo de estas prácticas, cada alumno deberá disponer de un cuaderno dedicado a las prácticas de esta asignatura donde se irán consignando los resultados de los ejercicios, comentarios personales, dudas particulares sobre aspectos específicos de las prácticas, etc. Este cuaderno podrá ser solicitado en cualquier momento por los profesores de la asignatura para evaluar con mayor rigor la evolución del alumno.

4. Aspectos específicos sobre programación en lenguaje C

En este apartado comentamos algunos de los problemas que con más frecuencia han aparecido a lo largo de los últimos cursos en el desarrollo de las prácticas de EAI.

Edición, compilación y ejecución.

Resulta muy frecuente la confusión que se produce entre las tareas de edición, compilación y ejecución de programas. Aunque el alumno disponga de un entorno integrado a partir del cual pueden llevarse a cabo todas estas tareas sin salir de él, es preciso que el alumno tenga una concepción clara sobre las diferencias entre ellas.

Cuando el usuario se dispone a realizar un ejercicio, la primera tarea es utilizar un editor de textos para llevar a cabo la escritura del programa. Algunos usuarios, antes de finalizar la escritura completa del programa, se disponen a compilarlo en un intento de descubrir si lo que llevan escrito hasta ahora contiene o no errores de sintaxis. Esta forma de proceder no es en modo alguno correcta y más que ayudar al programador, puede ocasionarle una total desorientación sobre su trabajo. El programa no se debe compilar hasta que el programador no haya finalizado completamente la edición del mismo y esté definitivamente seguro de que el programa que acaba de escribir es correcto y va a funcionar conforme a sus expectativas. La depuración del diseño del algoritmo es una tarea que sólo puede realizar el programador mediante la lectura detenida de su programa y no existen herramientas “mágicas” en las que el programador pueda delegar esta importantísima tarea de confección de un código correcto (desde luego el compilador no es una de ellas).

Una vez que el programador da el visto bueno a su programa y está completamente convencido de que lo que ha escrito (mediante el editor de textos) se ajusta a sus intereses y es exactamente lo que quiere escribir, entonces y sólo entonces, puede proceder a la fase de compilación.

Con bastante probabilidad, la compilación del programa le retornará al programador una serie de errores de sintaxis cometidos por despiste en la escritura del código. Es entonces cuando de nuevo se debe repetir el proceso desde el principio, es decir, editando de nuevo el código, modificando su contenido para eliminar los errores, repasando finalmente todo el código para asegurarnos de nuevo que tal código se ajusta a nuestras expectativas y volviendo a compilar el programa. Este proceso se repite hasta “limpiar” el código de errores y poder confeccionar un módulo que pueda ser ejecutado sin más.

Resulta frecuente ver a un usuario que, tras editar por ejemplo uno de los módulos de utilidades aportados para el desarrollo de estas prácticas (como “utilp1.txt”), la primera tarea que realiza es ¡compilar dicho módulo!, antes siquiera de conocer su contenido o investigar su funcionalidad. ¿Cuál es el objeto de tal compilación? En el desarrollo de estas prácticas huiremos de este tipo de actividades alejadas de la metodología.

Uso de punteros.

Uno de los problemas más frecuentes en las prácticas estriba en la falta de experiencia del alumno respecto al manejo de punteros en C o mejor dicho variables específicas para almacenar direcciones de memoria. Los punteros en el lenguaje C resultan fundamentales para la implementación de programas eficientes, dinámicos y flexibles. Es por ello que el alumno debe tener un conocimiento previo suficientemente amplio sobre el modo de utilizar variables de tipo puntero en C. Una idea simple que ahorra bastantes despistes es considerar un puntero (cualquiera que sea la forma en que se ha declarado) como una variable que contiene una dirección de memoria. Con esta perspectiva, resulta fácil abordar el estudio de los punteros en C. La diferencia respecto al tipo en que se ha declarado una variable de tipo puntero estriba básicamente en la interpretación que el compilador hace de la zona de memoria a donde apunta el contenido de dicha variable, así como la forma de interpretar la aritmética de punteros.

Imaginemos por ejemplo esta declaración:

```
char *cp;
int *ip;
void *vp;
int v[5];
```

La variables cp, ip y vp no se diferencian en nada. Todas son del mismo tipo y ocupan el mismo número de bytes dentro de la memoria. Para todas estas variables el contenido de esos bytes se interpreta exactamente del mismo modo, es decir, el valor de cada una de estas variables se interpreta como una dirección de memoria RAM. El programador es el único responsable de asignarle un valor correcto a cualquiera de estas tres variables. Por el contrario v difiere de las demás y escrito sin más, el lenguaje lo interpreta como una constante cuyo valor contiene la dirección de comienzo del vector v. Supongamos que cp contiene el valor 215, ip = 438 y vp = 123. Si el programador hace referencia a *cp, el sistema se dirige a la dirección 215, toma de allí un byte (lo que ocupa el tipo char) y lo interpreta como un entero en el rango [-127, 127]. A diferencia de esto, si se escribe *ip, el sistema acude a la dirección 438, toma de allí 2 o 4 bytes (el tamaño del tipo int) e interpreta este contenido como un número entero. Por último, una referencia *vp, hace que el sistema acuda a la dirección 123 y no haga interpretación alguna sobre lo que allí se almacene. El sistema trabajará entonces con el byte allí almacenado sin hacer interpretación alguna sobre el mismo.

Respecto a la aritmética de punteros, (cp + 1) apunta a la dirección 216, (ip + 1) apunta a la dirección 442 (considerando que el tipo int ocupa 4 bytes) y (vp + 1) genera un error de compilación. Para poder aplicar aritmética de punteros a vp, es preciso hacer una reasignación de tipos (*casting*) de la variable.

Concordancia de los programas con la metodología.

Uno de los problemas más acuciantes de los asistentes a las prácticas de EAI es la inexplicable falta de concordancia de los programas escritos en C, respecto de la metodología aprendida en cursos anteriores. La escritura de programas en C, así como en cualquier otro lenguaje de programación, se debe asentar sobre unas bases metodológicas que son introducidas en asignaturas previas de la carrera. No es admisible que la inexperiencia en el uso de un determinado lenguaje de programación resulte finalmente en un diseño erróneo del algoritmo. La destreza en el uso de la herramienta concreta sólo puede conseguirse mediante la aplicación escrupulosa del método, el cual es independiente de aquella. Por ello, a lo largo de estas prácticas de EAI, no sólo se presta atención al hecho de que los programas escritos en C resuelvan de uno u otro modo los problemas planteados. En estas prácticas incidiremos en la

característica fundamental de que las soluciones planteadas por los alumnos se ajusten a los niveles de calidad metodológica que se debe exigir a un futuro ingeniero, partiendo del planteamiento de un algoritmo correcto y finalizando en la materialización de dicho algoritmo mediante el uso del lenguaje C. En este sentido, el lenguaje es el medio, lo cual no excluye que dicho medio se utilice con criterio para llegar al fin deseado, que es el planteamiento de una solución apropiada.

5. Normas de estilo

Todos los programas que construyamos a lo largo de este curso deben seguir siempre unas sencillas normas de estilo, que ayudarán al programador a mantener un orden y una cierta disciplina para la escritura homogénea de todos los programas.

Todos los programas llevarán un archivo de cabecera de nombre PN.h (dependiendo de la práctica que se esté realizando N=1, 2, 3...) y extensión h que contendrá los siguientes elementos:

Líneas de comienzo y fin del archivo

Comenzará siempre con las líneas

```
#if !defined (_PN_H)
#define _PN_H
```

y finalizará con la línea

```
#endif
```

Bloque de inclusiones

El siguiente bloque de este archivo estará compuesto por una serie de directivas que nos permitan incluir los archivos de cabecera de manejo más usual, por ejemplo:

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

Definición de constantes

Este bloque incluye todas las constantes que vayan a ser utilizadas en las funciones que se utilicen a lo largo del programa. Los nombres de las constantes siempre se escribirán en MAYÚSCULAS, por ejemplo:

```
#define UNO             1
#define BLOQUESZ        512
#define NOMBREFICH      "mifichero.dat"
```

Definición de tipos

Este bloque incluye la definición de todos los tipos definidos por el usuario y necesarios para aportar claridad y sencillez a los programas. Los tipos definidos por el usuario se escribirán siempre en MAYÚSCULAS y finalizan con la letra “T”. Por ejemplo:

```
typedef struct {
  unsigned blid;
  char resto[BLOQUESZ - sizeof (unsigned)];
} BLOQUET;
```

Definición de variables globales

En este bloque se declararán todas las variables globales que se precisen para la construcción del programa. Las variables globales se nombrarán con el primer carácter en Mayúsculas, y el resto en minúsculas. Por ejemplo:

```
char Global;
```

Declaración de prototipos

Incluiremos aquí la declaración de los prototipos de todas las funciones escritas por el programador y que vayan a ser utilizadas en el programa principal. Los prototipos de las funciones podrían aparecer en orden alfabético para facilitar su lectura. Es conveniente que los nombres de las funciones escritas por el programador se distingan de las funciones proporcionadas por el compilador. Para ello, en estas prácticas daremos nombres a las funciones cuya característica diferencial es que comenzarán por una letra mayúscula. Si la función se compone de varias palabras, cada una de ellas comenzará por una letra mayúscula. Por ejemplo:

```
void CrearFicheroDatos (char *nombreficherodatos, void *bloque, long blsz);
```

Aparte de los mencionados bloques, algunas normas de estilo incluyen:

- Escribir las constantes siempre con mayúsculas (BLSZ)
- Definir las variables y nombres de campos con letras minúsculas (blid)
- Definir los nombres de tipos siempre en mayúsculas e incluir en su nombre una referencia que los identifique como tales (BLT)

Construido el archivo de cabecera, el cual no incluye detalle alguno de implementación, precisamos ahora el archivo de extensión *c* conteniendo en exclusiva la implementación de todas las funciones necesarias para la resolución del problema propuesto y cuyos prototipos se han declarado en su correspondiente archivo de cabecera con extensión h. Este archivo de implementación comenzará siempre con las siguientes tres líneas:

```
#ifndef _PN_H
#include "pn.h"
#endif
```

Y a continuación de estas tres líneas, aparecerán en orden alfabético la implementación de todas y cada una de las funciones cuyos prototipos aparecen en el archivo de cabecera, más la función principal main(), que aparecerá la última de todas.

Con estas sencillas normas de estilo conseguiremos escribir programas legibles y uniformes, que nos facilitarán las tareas de depuración y mejora del código.

6. Relación alumno-asignatura

Para tratar de impulsar una relación permanente y directa entre alumnado y profesorado de la asignatura, se pone a disposición de todos una página *web* en el portal institucional del centro (http://epcc.unex.es/) con información académica sobre la asignatura EAI (objetivos, temario, horario, tutorías, apuntes, notas, material para prácticas, etc.). Además de esta dirección, la asignatura EAI se encuentra disponible en el Campus Virtual de la UEX (http://campusvirtual.unex.es/), donde se colocará material de utilidad y herramientas complementarias de apoyo para un mejor aprendizaje.

7. Planificación EAI Lab para el curso 2009/2010

Lunes	Martes	Miércoles
Octubre		
05/10 16:00 y 18:00 h. Eclipse	06/10 16:00 y 18:00 h. Eclipse	07/10 16:00 P1S1
~~12/10 NO LECTIVO~~	13/10 16:00 y 18:00 h. P1S1	14/10 16:00 h. P1S1
19/10 16:00 y 18:00 h. P2S1	20/10 16:00 y 18:00 h. P2S1	21/10 16:00 h. P2S1
26/10 16:00 y 18:00 h. P2S2	27/10 16:00 y 18:00 h. P2S2	28/10 16:00 h. P2S2
Noviembre		
~~02/11 NO LECTIVO~~	03/11 16:00 y 18:00 h. P3S1	04/11 16:00 h. P3S1
09/11 16:00 y 18:00 h. P3S2	10/11 16:00 y 18:00 h. P3S2	11/11 16:00 h. P3S2
16/11 16:00 y 18:00 h. P4S1	18/11 16:00 y 18:00 h. P4S1	19/11 16:00 h. P4S1
23/11 16:00 y 18:00 h. P4S2	24/11 16:00 y 18:00 h. P4S2	25/11 16:00 h. P4S2
30/11 16:00 y 18:00 h. P4S3		
Diciembre		
	01/12 16:00 y 18:00 h. P4S3	02/12 16:00 h. P4S3
07/12 LIBRE ACCESO	~~08/12 NO LECTIVO~~	09/12 LIBRE ACCESO
14/12 16:00 y 18:00 h. P4S4	15/12 16:00 y 18:00 h. P4S4	16/12 16:00 h. P4S4
Enero		
11/01 16:00 y 18:00 h. P5S1	12/01 16:00 y 18:00 h. P5S1	13/01 16:00 h. P5S1
18/01 16:00 y 18:00 h. P5S2	19/01 16:00 y 18:00 h. P5S2	20/01 16:00 h. P5S2

Práctica 1

La caja de herramientas básica.

Entrada y salida en C

1. Introducción

La entrada y salida en C se realiza por medio de un conjunto de funciones de librería, por lo que no existen en el lenguaje instrucciones o sentencias reservadas específicamente para el tratamiento de ficheros. El lenguaje C pone a disposición del programador dos familias de funciones para el tratamiento de la E/S en ficheros:

- E/S de alto nivel (ANSI)
- E/S de bajo nivel (UNIX)

La diferencia fundamental entre ambas familias estriba en la forma en que el compilador lleva a cabo la gestión de la E/S. En el estándar UNIX, las funciones de E/S invocan directamente al sistema operativo para llevar a cabo las operaciones de E/S a un dispositivo

Por el contrario, el estándar ANSI proporciona una capa de software intermedio entre las funciones que manipulan directamente el dispositivo físico y el programa de aplicación. Es por ello que el programador no debe preocuparse acerca de consideraciones físicas directamente relacionadas con el dispositivo sobre el que se ejecuta la operación de E/S. A lo largo de este curso, las operaciones de E/S sobre ficheros de disco se realizarán utilizando la familia de funciones proporcionada por el estándar ANSI, ya que se espera que el sistema tipo UNIX vaya decayendo en un futuro.

Antes de comenzar con la descripción del conjunto básico de rutinas para el manejo de ficheros en C, conviene definir el concepto de *canal* o *flujo* en conexión con el concepto de *fichero* por parte del lenguaje de programación C.

En el estándar ANSI, el lenguaje C proporciona al programador una interfaz independiente del dispositivo al que se está accediendo. En este sentido el programador trabaja en un nivel de abstracción entre el programa de aplicación y el dispositivo que se está usando para llevar a cabo la E/S. A esta interfaz se le denomina *canal*, mientras que el término *fichero* está directamente asociado con el conjunto de datos que se sitúa en un dispositivo específico.

El sistema de ficheros de C está diseñado para trabajar con una amplia variedad de dispositivos, incluyendo terminales, controladores de disco o modems. Aunque cada dispositivo es diferente (con su propia arquitectura y modo de trabajo), el sistema de ficheros ANSI transforma cada uno de ellos en un dispositivo lógico llamado canal, de modo que todos los canales se comportan de la misma forma. Existen básicamente dos tipos de canales: canales de texto y canales binarios.

Un *canal de texto* es una simple secuencia de caracteres. En un canal de este tipo pueden ocurrir ciertas conversiones de caracteres si el entorno del sistema lo requiere. Así por ejemplo un carácter de salto de línea puede convertirse en un par de caracteres retorno de carro y salto de línea, si el dispositivo de destino es una impresora. Es por ello que puede no existir una relación uno a uno entre los caracteres del canal y los correspondientes caracteres que se leen o escriben en el dispositivo físico.

Por su parte un *canal binario* es una secuencia de bytes con una correspondencia uno a uno con los del dispositivo destino del canal. En este caso no se realiza ningún tipo de conversión entre los caracteres almacenados en el canal y los que finalmente se dirigen al dispositivo.

En lo que concierne al programador, toda la E/S tiene lugar a través de un canal. Es el sistema de E/S ANSI de C el que se encarga de enlazar un canal a un archivo, que no es otra cosa que cualquier dispositivo capaz de realizar operaciones de E/S de datos. A lo largo de este curso, como es natural nos centraremos en los dispositivos de disco como los destinatarios de nuestras operaciones de E/S, por lo que cada vez que usemos el término fichero o archivo, lo asociaremos con un conjunto estructurado de datos almacenados sobre un disco. El uso de canales permite al programador aislarse en gran medida de los detalles físicos del dispositivo al que se destina la E/S, y concentrarse así en los aspectos lógicos de las operaciones de entrada y salida sobre el canal.

Cada vez que el programador precisa utilizar un fichero sobre disco, necesitará pues asociar un canal a dicho fichero. Para llevar a cabo esta asociación, el lenguaje C pone a disposición del programador una estructura de control cuyo tipo se denomina FILE. El siguiente apartado detalla el modo de asociar un canal a un fichero de datos por medio de esta estructura de control.

2. Apertura y cierre de ficheros

Las funciones de E/S en C utilizan la librería <stdio.h>. Para llevar el control de las operaciones de E/S sobre un canal y la conversión de dichas operaciones en acciones de lectura y escritura sobre el archivo de datos, el lenguaje C utiliza una zona de memoria interna para almacenar lo que nosotros denominamos la *información de control del canal*. Esta información de control se estructura en un tipo predefinido denominado FILE[1]. Por tanto, cada fichero que utilicemos en C, llevará asociada en cualquier momento una información de tipo FILE cuyo contenido es controlado en todo momento por el lenguaje C. Una vez que el sistema toma el control de esta información, éste retorna al usuario la dirección de memoria donde se encuentra dicha información para que el programador pueda realizar operaciones sobre dicho canal. Para llevar a cabo operaciones de E/S sobre un canal concreto, el programador sólo debe referenciar la dirección de memoria principal donde se encuentra la información de control de dicho canal.

Para llevar a cabo la asociación entre canal y objeto físico (en nuestro caso fichero de datos en disco), es preciso realizar una operación de *apertura*. La función fopen() lleva a cabo esta tarea. El prototipo es el siguiente:

```
FILE *fopen (char *nombre_fichero, char *modo)
```

El parámetro *modo* es una cadena de caracteres que indica la forma en que va a utilizarse el fichero. Según dijimos, un fichero (canal) puede abrirse en dos posibles modos, a saber, modo texto y modo binario. Los valores típicos para el parámetro modo son:

[1] "FILE" puede traducirse como fichero en español, sin embargo, es importante no confundir el contenido de un fichero con este nombre de estructura. Es un error común pensar que una variable de tipo FILE es un fichero en disco.

Valor	Tipo	Operación
“rt”	Texto	Abre un fichero existente para sólo lectura. Error si no existe.
“wt”	Texto	Crea un fichero para escritura. Si ya existe lo destruye.
“rb”	Binario	Abre un fichero existente para sólo lectura. Error si no existe.
“wb”	Binario	Crea un fichero para escritura. Si ya existe lo destruye.
“r+t”	Texto	Abre un fichero existente para lectura y escritura. Error si no existe.
“w+t”	Texto	Crea un fichero nuevo para lectura y escritura. Si ya existe lo destruye.
“r+b”	Binario	Abre un fichero existente para lectura y escritura. Error si no existe.
“w+b”	Binario	Crea un fichero nuevo para lectura y escritura. Si ya existe lo destruye.

La función fopen() retorna una dirección de memoria cuyo contenido se interpreta de tipo FILE. La dirección de memoria retornada por esta función **debe ser capturada por el programador**, pues esta dirección resulta ser la única referencia para realizar nuestras operaciones sobre un determinado fichero de datos.

La función fopen() devuelve un valor nulo si la operación no termina con éxito, lo cual nos permite interrumpir ordenadamente la ejecución de una función en caso de que fracase una operación de apertura. Es conveniente entonces consultar el valor de retorno de esta función antes de proseguir con el resto de operaciones a realizar sobre un fichero. Una buena práctica consiste en invocar la apertura del fichero del modo siguiente:

```
fp = fopen(“miarchivo.dat”,”rb”);
if (fp == NULL) return ERROR;
<< Continúa el código>>
```

Cualquier fichero que haya sido abierto por medio de la función fopen() debe cerrarse una vez utilizado. Esto permite liberar todos los recursos que destina el sistema a controlar las operaciones de E/S sobre el canal. Para cerrar un fichero se utiliza la función fclose(), cuyo prototipo es:

```
int fclose (FILE *fp)
```

Una vez abierto un fichero, necesitamos disponer de operaciones básicas para leer y escribir bloques de información (páginas) en él. En lo que sigue tratamos separadamente ambos aspectos del trabajo con ficheros.

3. Lectura

Para poder trabajar con los datos almacenados en un fichero es preciso realizar un trasvase de datos desde memoria secundaria hacia memoria principal. Esta operación se denomina *lectura* y la función que utilizaremos para llevar a efecto esta operación tiene el prototipo siguiente:

```
int fread (void *dir_bloque, int num_bytes, int contador, FILE *fp)
```

En esta función el parámetro dir_bloque contiene la dirección de memoria destino de los datos que se leen desde el fichero en disco asociado a fp. Es responsabilidad del usuario que los bytes trasvasados a la dirección indicada (dir_bloque), información que se almacena en el parámetro num_bytes, no destruyan información crítica o sensible a otras partes del programa o del propio sistema. Esto significa que el programador se debe asegurar de proporcionar una dirección de memoria interna que sea segura. Como

ya hemos adelantado, la cantidad de bytes que van a leerse viene indicado por dicho parámetro num_bytes. El parámetro contador indica el número de veces que se va a realizar esta operación de lectura, sin embargo, nosotros siempre utilizaremos el valor 1 en este parámetro.

Finalmente, el parámetro fp será la dirección de memoria de la información de control del canal asociado al fichero desde donde deseamos leer.

4. Escritura

De modo similar a la operación de lectura, una operación de escritura sobre un fichero en disco conlleva un traslado de información desde memoria principal hacia memoria secundaria, por tanto, para poder escribir datos en un fichero, es preciso disponer antes los datos a escribir sobre alguna zona de memoria principal.

La realización de una escritura sobre un fichero se lleva a cabo por medio de la función fwrite(), cuyo prototipo es:

```
int fwrite (void * dir_bloque, int num_bytes, int contador, FILE *fp)
```

El primer parámetro es la dirección de memoria principal donde se encuentra el contenido del bloque que deseamos escribir en el fichero. El segundo parámetro indica la longitud del bloque donde se encuentran los datos y el parámetro contador almacena el número de veces que deseamos realizar esta operación (para nosotros contador = 1). El último parámetro...bueno no es necesario repetir el significado de ese parámetro, ¿no?.

5. Posicionamiento

Las operaciones anteriores nos permiten almacenar y recuperar datos en ficheros de disco. Sin embargo, como ya hemos dicho previamente, las funciones de lectura y escritura son operaciones de trasvase de información entre disco y memoria interna y los prototipos de las funciones nos permiten indicar en qué punto de la memoria interna tiene lugar el destino (lectura) u origen (escritura) de la operación, pero ¿qué sucede con el origen (lectura) y destino (escritura) de la operación en el otro extremo del trasvase, es decir en el origen y destino en el archivo? ¿cómo le indica el programador al sistema a partir de qué punto en el archivo deseamos leer una cierta información, o hacia qué punto del archivo van a parar los datos que deseamos escribir? La indicación del destino u origen en el extremo del archivo la mantiene el lenguaje C en algún lugar dentro de la información de control del canal que nosotros denominamos *variable de posicionamiento actual*, la cual apunta permanentemente a una posición concreta del fichero de datos, dictando el punto de destino u origen de la operación en el archivo de datos.

Cuando el archivo se abre, la variable de posicionamiento actual se inicializa al byte 0 del archivo y posteriormente, cada vez que se realiza una operación de lectura/escritura, la variable de posicionamiento actual dentro del fichero se desplaza automáticamente tantos bytes como se hayan leído o escrito en el fichero. El programador puede desplazar “manualmente” la variable de posicionamiento actual del archivo según sus necesidades. Para ello, el compilador C, dispone de la función fseek(), cuyo prototipo es el siguiente:

```
int fseek (FILE *fp, long num_bytes, int origen)
```

Donde el parámetro fp referencia la información de control del canal asociado al fichero sobre el que se desea realizar la operación de desplazamiento, num_bytes indica el desplazamiento en bytes que deseamos realizar dentro del fichero a partir del origen

que se le indique en el tercer parámetro. El valor de origen es habitualmente el comienzo del fichero, sin embargo es posible especificar otros puntos de origen a partir del cual realizar el desplazamiento. Son estos:

CONSTANTE	VALOR	SIGNIFICADO
SEEK_SET	0	Desde el principio del fichero
SEEK_CUR	1	Desde la posición actual (almacenada en curp)
SEEK_END	2	Desde el final del fichero

6. Toma de contacto

Estos sencillos ejercicios tienen por objeto experimentar con las operaciones básicas sobre ficheros de datos. Antes de proceder a la ejecución del código que se genere conviene invocar la función TestPrincipal(), la cual realiza un chequeo sobre los tamaños de las estructuras utilizadas en las prácticas. Para evitar que el compilador realice una optimización de los tamaños de las estructuras y modifique los tamaños de las mismas haciendo fracasar nuestro código, incluya la siguiente opción en la línea de compilación del programa:

```
-fpack-struct
```

Teniendo esto en cuenta, podemos ya abordar los primeros ejercicios.

Ejercicio P1.1. Escriba una función:

```
char EjercicioP1_1(void)
```

para crear en el disco un archivo de nombre “uno.dat”. Este archivo debe contener un único byte con todos sus bits a 0. La función retornará EXITO en caso de terminar correctamente y FRACASO en caso de no terminar bien (por ejemplo porque el fichero no se hubiera podido abrir correctamente).

Recuerde los siguientes puntos:

- Incluir el archivo de cabecera stdio.h
- Abrir el archivo en el modo correcto (binario para escritura)
- Que la operación de escritura es un trasvase de información desde RAM a disco, donde el programador debe indicar la dirección de memoria origen del trasvase. (Disponga en memoria RAM una variable que ocupe un único byte y asígnele un valor que haga que todos los bits estén a cero. Luego mande escribir este byte al archivo)
- Cerrar el archivo.
- Comprobar que el archivo está en el disco. Seleccione el archivo y examine sus propiedades (botón derecho del ratón).

Ejercicio P1.2. Escriba una función:

```
char EjercicioP1_2(void)
```

que cree un archivo “dos.dat” consistente en una pequeña cabecera[2] que tenga por objeto indicar el tamaño en bytes del archivo “dos.dat” (incluyendo los bytes dedicados a la propia cabecera). Una vez creado cierre el archivo. Abra de nuevo el fichero “dos.dat” en el modo adecuado que permita escribir nuevos datos sin destruir los datos existentes. Posiciónese al final del fichero y escriba *con una única sentencia fwrite* un bloque de 1024 bytes, cada uno de ellos con valor 0. Una vez escrito el bloque de datos,

[2] Una cabecera no es más que un bloque de bytes que va al comienzo del archivo.

actualice el valor de la cabecera y cierre el archivo. Retorne EXITO en caso de terminar correctamente y FRACASO en caso contrario.

Ejercicio P1.3. Escriba una función:

```
char EjercicioP1_3(char *filename, unsigned numbytes, char valorbyte)
```

que cree un archivo de nombre filename dotado con una cabecera que informe sobre el tamaño en bytes del archivo. Escriba luego en este fichero tantos bytes como indique el segundo parámetro, todos los bytes deben tener el valor indicado en el tercer parámetro. Cierre el archivo y devuelva ÉXITO o FRACASO según el resultado de la operación.

Ejercicio P1.4. Escriba una función:

```
char CreaFichero(char *filename, void *bloque, unsigned blsz)
```

que tenga por efecto crear un fichero nuevo en disco de nombre "filename". Dicho fichero estará dotado de una cabecera de tamaño blsz cuyos datos se encuentran en la dirección de memoria indicada en el segundo parámetro por bloque. El llamador se encargará de colocar en la memoria el bloque de bytes con la información que desea contenga el fichero. La función retornará ÉXITO o FRACASO según el resultado de la operación.

Ejercicio P1.5. Escriba una función:

```
char EjercicioP1_5(char *filename)
```

que tenga como objetivo mostrar por pantalla información relativa a un archivo creado mediante la función EjercicioP1_3. Si el fichero indicado no existe indíquelo por pantalla y retorne FRACASO. Si existe el archivo, muestre por pantalla la información de su tamaño almacenada en su cabecera. A continuación lea y muestre cada byte del fichero. Retorne ÉXITO o FRACASO según el resultado de la operación. Compruebe mirando en las propiedades del archivo que el tamaño del mismo coincide con lo que mostró por pantalla. Edite el archivo con un editor de texto. ¿Qué observa? ¿Coincide lo que observa con lo que leyó por pantalla?. Haga una interpretación de todo ello.

7. Acceso secuencial y acceso directo

El acceso secuencial a un fichero consiste en la ejecución de operaciones de lectura o escritura de los bloques (páginas) del fichero de forma masiva y contigua. Conceptualmente hablando, un acceso se define como secuencial cuando *el orden lógico en que se procesan los elementos del fichero coincide con el orden físico en el que se disponen dentro del fichero.* Este tipo de acceso es el más rápido posible dentro de un fichero de datos, considerando que el espacio en disco que el sistema reserva para el fichero es *efectivamente* contiguo.

Para llevar a efecto un acceso secuencial al fichero, nos posicionaremos en el byte inicial del fichero y a partir de aquí comenzaremos a ejecutar iterativamente instrucciones de lectura o escritura *sin que intervengan operaciones de movimiento de las cabezas de L/E del disco generadas expresamente por el programador (por medio de la función fseek() que estudiaremos en el apartado 8).*

Con objeto de ensayar el acceso secuencial a un fichero y antes de escribir una función que lleve a cabo este tipo de acceso, lleve a cabo las siguientes declaraciones

```
#define BLSZ 1024  //Tamaño del bloque de bytes
#define UINTSZ 4 //Depende del compilador
```

```
typedef struct {
  unsigned in blid; //Identificador del bloque
  char datosbl[BLSZ - UINTSZ]; //Contenido
} BLT; // Tipo para el bloque
typedef struct {
  unsigned int blid; //Identificador del bloque
  unsigned int ocupacion; // Ocupación del fichero (en bloques)
  char extrainfo[BLSZ – 2* UINTSZ];
} CABFT; //Tipo para la cabecera del fichero
```

Una variable de tipo BLT será un bloque de 1024 bytes. El campo blid es un identificador de bloque y el campo datosbl es la zona del bloque de bytes que contiene los datos (en este ejemplo, no prestaremos atención al valor de este bloque de bytes). Por su parte CABFT es la declaración del tipo que va a constituir la cabecera del fichero. En este caso, la cabecera es también un bloque de bytes del mismo tamaño que los bloques de tipo BLT. El campo blid es el identificador del bloque de cabecera y el campo ocupacion almacena el número de bloques que contiene en cada momento el fichero (incluyendo el bloque de cabecera).

Ejercicio P1.6. Escriba una función

```
char EscrituraSecuencial(unsigned numbl)
```

que lleve a cabo las siguientes acciones:

- Crear un fichero de nombre "bloques.dat" que incluya como cabecera un bloque de tipo CABFT, cuyo valor blid sea 0 y ocupacion tenga valor 1. Esta creación deberá hacerla mediante la llamada a la función CreaFichero() del ejercicio P1.4..
- Abrir el fichero "bloques.dat" en el modo conveniente y posicionarse al final del mismo (ejecute la sentencia fseek(…) con los valores que corresponda).
- Escribir secuencialmente un total de numbl bloques de tipo BLT, donde los identificadores de estos bloques coinciden con la posición que ocupan en el fichero. Vaya actualizando el valor ocupación del bloque cabecera.
- Volcar la cabecera y cerrar el fichero "bloques.dat".

Compruebe que el tamaño del fichero "bloques.dat" almacenado en el disco es acorde con el número y tamaño de los bloques que hemos escrito en él. Realizar las modificaciones pertinentes hasta conseguir los resultados esperados.

Ejercicio P1.7. Escriba una función

```
char LecturaSecuencial(void)
```

que lleve a cabo las siguientes acciones:

- Abrir el fichero "bloques.dat" en el modo pertinente para realizar lecturas sobre el mismo.
- Leer secuencialmente todos los bloques de este fichero y mostrar por pantalla los valores de los identificadores de cada uno de ellos.
- Cerrar el fichero.

Aunque el acceso secuencial es la forma más rápida de trabajar con los ficheros de datos, a veces resulta necesario poder acceder aisladamente a un bloque situado en una posición no necesariamente consecutiva al último acceso realizado. Esta forma de acceso a bloques individuales situados en distintas localizaciones del fichero, se conoce como acceso directo o aleatorio. En este caso, para leer o escribir un bloque, es preciso indicar llevar a cabo una operación de posicionamiento, variando el valor de la variable de posicionamiento actual del fichero.

Dado que, en general, los accesos que haremos durante el desarrollo de estas prácticas se llevarán a cabo en términos de bloques o páginas del fichero, será preciso realizar una pequeña conversión capaz de traducir el número del bloque o página a la que se desea acceder (para leer o escribir allí) en el *desplazamiento en bytes* donde comienza dicho bloque o página. Así por ejemplo, si estamos trabajando con bloques de tamaño BLSZ bytes, para acceder al bloque con identificador 4 (quinto bloque del fichero, ya que contabilizamos el bloque de cabecera que siempre tiene identificador 0) habremos de hacer una llamada a la función fseek() con los siguientes parámetros:

```
fseek (fp, 4*(long)BLSZ, SEEK_SET)
```

Para experimentar con el acceso directo utilizaremos el fichero "bloques.dat" creado en el apartado anterior.

Ejercicio P1.8 Escriba una función

```
char AccesoDirecto(void)
```

encargada de abrir el fichero "bloques.dat" y leer un 80% de los bloques de modo aleatorio. Use para ello la función Permutación(), la cual le sirve para obtener una permutación aleatoria de los identificadores de bloque del fichero. Muestre por pantalla después de cada lectura el valor del identificador del bloque accedido y finalice cerrando el fichero.

8. Lectura y escritura de bloques individuales

Una vez que hemos adquirido cierto grado de destreza en la programación de operaciones básicas de acceso a ficheros de datos, podemos abordar la programación de operaciones específicas de lectura y escritura que podrán ser de utilidad en las siguientes prácticas.

Ejercicio P1.9 Escriba una función

```
char LeeBloque(void *bl, unsigned blsz, unsigned long blid, FILE *fp)
```

que lea un bloque de un archivo de datos a partir del identificador del mismo. Cuando el archivo de datos se compone de bloques todos del mismo tamaño, cada bloque tiene un identificador que habitualmente coincide con la posición física del bloque dentro del archivo. Esta función posibilita la operación de lectura en este tipo de archivos. La función debe retornar FRACASO en caso de intentar leer un bloque más allá del final del archivo.

Ejercicio P1.10 Escriba una función

```
char EscribeBloque(void *bl, unsigned blsz, unsigned long blid, FILE *fp)
```

similar a la anterior capaz de escribir un bloque en la posición indicada por el blid. Esta función debe permitir añadir un bloque a partir del final del fichero, pero debe retornar un valor de FRACASO en caso de intentar escribir bloques más allá de la posición final del fichero. Por ejemplo si un archivo se compone de 32 bloques (con blid's desde 0 a 31), la función debe permitir por supuesto escribir cualquier bloque con blid desde 0 a 31, también debe permitir añadir un bloque nuevo con blid 32, pero debería retornar FRACASO si se intenta escribir un bloque con blid mayor que 32.

Práctica 2

La paginación de registros

1. Introducción

Cuando trabajamos con aplicaciones que manejan grandes cantidades de datos, en la mayoría de los casos es preciso utilizar almacenamiento externo. Una parte muy importante del tiempo de ejecución de este tipo de aplicaciones se concentra en las operaciones de entrada y salida a discos. Es por ello que resulta conveniente conocer el modo de funcionamiento de este tipo de operaciones con el fin de minimizar el tiempo de ejecución de estas aplicaciones.

En esta segunda práctica, trabajaremos con el concepto de bloque o página al objeto de entender en toda su profundidad los beneficios que se obtienen de una correcta planificación del trabajo a realizar con los ficheros de datos almacenados en disco.

Hay que decir que las operaciones de acceso a los datos del disco vienen impuestas por distintos elementos que inciden directamente en la forma en que se procesan tales operaciones. Estos elementos incluyen básicamente el tipo de controladora y el sistema operativo utilizado. Así pues, si deseamos hacer un análisis profundo sobre los costes del acceso a datos almacenados en memoria secundaria, será preciso indagar en las características de ambos elementos.

Lo primero a tener en cuenta es que las operaciones de E/S son ejecutadas en último lugar por la controladora del disco duro donde tengamos almacenados los datos. Normalmente, los lenguajes de alto nivel, como C, no realizan directamente llamadas a las funciones de la controladora de disco, sino que es el S.O. el responsable de mediar entre las sentencias de E/S emitidas en un lenguaje y las operaciones de la controladora de disco.

Así pues antes de investigar las implicaciones en la eficiencia durante el uso de sentencias de E/S en C, haremos un repaso sobre el modo en que se ejecutan realmente este tipo de operaciones a bajo nivel.

2. El acceso a disco bajo Windows

Recordemos en primer lugar que los discos instalados en nuestro PC, se organizan según una arquitectura tridimensional consistente en <*Cilindros, Pistas, Sectores*>. Como sabemos este tipo de arquitectura optimiza la velocidad en el acceso a los datos.

El formato habitual de un disco duro bajo Windows incluye sectores de tamaño 512 bytes. Sin embargo, aparte de la componente *sector* en la arquitectura tridimensional, el SO define un elemento intermedio denominado *cluster* (agrupamiento) consistente en un número determinado de sectores consecutivos (adyacentes) en el disco. Por ejemplo, en el ordenador que hay sobre mi mesa, el cluster tiene un tamaño de 8K, equivalente por tanto a un total de 16 sectores de 512 bytes.

El sentido de reunir varios sectores en un cluster resulta de la lógica con la que el SO escribe datos en un disco. Como es sabido, Windows fragmenta los archivos durante su creación entre el conjunto de sectores libres que se vayan encontrando (producto de borrados previos). Así pues para evitar que esta fragmentación ralentice en exceso el acceso al disco, posicionando constantemente la cabeza de lectura/escritura por cada sector al que se desee acceder, un grupo de sectores contiguos se reúnen en un cluster, garantizando así que los sectores de un cluster contienen un trozo de un archivo,

cuyos bytes se pueden leer en secuencia. Esto supone que la asignación de espacio a ficheros se hace en términos de *clusters*. Evidentemente hay un precio que pagar (en fragmentación externa) por el uso de clusters. ¿Descubre cuál es?

El uso de cluster viene a clarificarnos un poco el significado de acceso secuencial, cuando se trabaja bajo Windows: leer un fichero desde el primer al último byte puede significar (si el disco está muy fragmentado, y por ende el fichero en sí) un montón de operaciones de desplazamiento de cabezas hacia delante y hacia atrás en el disco. Esto supone que en realidad tal acceso secuencial puede convertirse en un acceso *"pseudosecuencial"*. No obstante el tamaño del cluster puede darnos una idea de la longitud máxima que debe tener una página, como luego veremos.

Además de lo anteriormente expuesto, es preciso indicar que, siendo el SO el elemento encargado de enviar las órdenes a la controladora del disco duro, conviene saber que en Windows, el número máximo de sectores que es posible leer con una llamada a la interrupción correspondiente es de 128 sectores (64 Kb). Este dato aporta también una idea sobre la limitación física acerca del tamaño máximo que puede tener una página del fichero.

Hay que añadir, por último que las propias controladoras y el mismo sistema operativo incluye sus propias memorias cachés con objeto de acelerar las operaciones de E/S, por tanto los tiempos invertidos en realizar este tipo de operaciones dependerán fuertemente de las características de estas memorias intermedias o cachés. Esto se traduce en una dificultad añadida a la hora de investigar los efectos de la paginación mediante la ejecución de sentencias de L/E invocadas a través del lenguaje C.

3. Registros lógicos y páginas

Un fichero se puede definir como un almacén de datos donde se guarda información sobre un conjunto de instancias de una entidad concreta. Por ejemplo, un fichero de alumnos guarda información sobre un conjunto de alumnos, cada uno con características diferentes. En este caso, la entidad sobre la que almacenamos información es el Alumno. Cada alumno concreto es una instancia diferente de dicha entidad.

Pues bien, a partir de esta definición podemos decir que nuestro fichero está lógicamente compuesto por un conjunto de registros, cada uno representando a una instancia concreta de la entidad alumno, a un alumno concreto. Nuestra unidad básica de información semántica es el registro individual, la unidad mínima de información con sentido es lo que se define como el *registro lógico*.

El párrafo anterior aporta una definición de fichero desde el punto de vista semántico, que tiene que ver con el significado de los datos que se almacenan en él. Ahora podríamos aportar una visión estrictamente física y en cierto modo diferente de fichero. En este sentido podríamos pensar en nuestro anterior fichero como un conjunto de bytes almacenados en un dispositivo de disco. ¿Cómo se realiza el proceso de escribir y leer datos a y desde este fichero? La pregunta a esta respuesta es sencilla. Para escribir datos en un fichero disponemos un “paquete” de bytes en memoria principal y emitimos una orden que nos traslade este paquete desde la memoria interna hacia el disco. Inversamente, el proceso de lectura se lleva a cabo mediante una orden que toma un “paquete” de bytes desde el disco y nos lo traslada a la memoria interna, desde donde podrá ser procesada como queramos por el procesador.

Con el propósito de homogeneizar las operaciones de lectura y escritura sobre un mismo fichero, simplificando así nuestro programa de tratamiento de los datos, habitualmente estos “paquetes” son todos del mismo tamaño. El programador tiene entonces capacidad de definir el “paquete” de información como la unidad de entrada y

salida de información en el fichero de datos. A partir de aquí resulta sencillo definir físicamente nuestro fichero como un conjunto de *páginas de datos*, todas del mismo tamaño, que constituyen la unidad de entrada y salida de datos desde y hacia tal fichero.

¿Cómo enganchamos los conceptos semántico y físico de un fichero?, o con otras palabras ¿cómo relacionamos los conceptos de *registro lógico* y *página* (también llamado *bloque*) dentro de un mismo fichero?

La respuesta a esta pregunta debe ser respondida por el programador mismo, bajo un estricto criterio de eficiencia. En general, nuestro interés será traernos de disco una cantidad de bytes suficientemente grande como para **amortizar** el tiempo que se invierte en la operación. En este sentido, aunque existen excepciones, la solución más habitual es definir una página de un tamaño suficiente como para que quepa un considerable conjunto de registros lógicos.

Teniendo en cuenta lo anterior, las operaciones de lectura y escritura en nuestros ficheros de datos se encargarán de leer y escribir *páginas* de datos. Para llevar a cabo el procesamiento de los registros lógicos (que es finalmente la unidad de información que *entiende* el programador, esto es, la unidad *semántica*), será entonces necesario realizar un proceso de *codificación-decodificación* de los registros lógicos en páginas. Por tanto, un registro lógico nunca se escribe o lee individualmente de un fichero de datos. Antes de escribir un registro lógico tendremos que *colocarlo* dentro de una página que será la que finalmente escribamos al fichero de datos. De modo similar, leer un registro supone trasladar toda una página desde el fichero de datos a memoria para, posteriormente, *extraer* de esta página el registro lógico deseado.

4. Lectura y escritura de páginas

De acuerdo al apartado anterior, todas las operaciones de lectura y escritura de datos sobre un fichero en disco las llevaremos a cabo usando como unidad de comunicación programa-fichero la página. Una página puede considerarse por tanto como un contenedor de tamaño predefinido, con dos partes diferenciadas: una pequeña cabecera con información administrativa de utilidad para el procesamiento de la misma (usualmente conteniendo el identificador de página) y la porción principal de la página que almacena los datos propios de la aplicación.

Un tipo base para las páginas podría definirse conforme a las siguientes declaraciones:

```
#define PAGSZ n // n es el número de bytes que ocupa la página
typedef struct {
  unsigned pagid;
  char resto[PAGSZ - UINTSZ];
} PAGT;
```

El tipo base para la página cabecera de todos nuestros ficheros será el siguiente:

```
typedef struct {
  unsigned pagid;
  unsigned ocupacion;
  char nombre_fichero[50];
  unsigned pagsz; // Tamaño de las páginas
  unsigned regsz; // Tamaño de los registros (caso de longitud fija)
  char resto[PAGSZ – 4* UINTSZ -50];
} CABFT;
```

De este modo cualquier acceso a datos del disco, se deberá realizar por medio de la lectura/escritura de variables de tipo PAGT (páginas), las cuales deben anterior/posteriormente sufrir un proceso de codificación/decodificación de registros lógicos dentro de las mismas. Por su parte, la página cabecera deberá mantenerse en memoria convenientemente actualizada mientras el fichero permanezca abierto.

Dado que la comunicación entre programa de aplicación y fichero de datos se hará mediante unidades de página, se hace necesario construir un capa de software que lleve a cabo la conexión entre los registros lógicos y su almacenamiento en disco en forma de páginas. En este apartado abundaremos en los procesos de codificación/decodificación de registros lógicos en páginas de tamaño predefinido. Este temática hay que abordarla desde dos escenarios distintos. El primero de ellos, supone la existencia de registros de longitud fija, el segundo –bastante más complejo– considera la materialización de registros de longitud variable en páginas de tamaño predefinido.

5. Registros de longitud fija (Práctica P21)

Cuando los registros tienen una longitud fija, el mecanismo para direccionar un registro, tanto en el ámbito de una página como en el marco del fichero global, es sencillo. Teniendo en cuenta que los tamaños de registro y página son fijos, el número de registros B que caben en una página es constante. Para acceder entonces al k-ésimo registro del fichero de datos, bastará con acceder a la página ($\lfloor k/B \rfloor + 1$) y en el ámbito de esta página, recuperar el registro (k % B).

Fijémonos en el hecho de que este mecanismo permite acceder a un registro según su posición ordinal en el fichero de datos. Si la posición ordinal de un registro lógico no tiene relación alguna con el lugar físico que tal registro ocupa en el fichero, algún otro mecanismo de traducción nos diría qué página del fichero debería ser accedida para localizar el registro. Una vez dentro de la página nos sería sencillo localizar el registro buscado, bien haciendo un recorrido por toda la página o bien empleando alguna otra fórmula prevista para localizar de forma directa el registro.

Un aspecto importante sobre el empaquetamiento (codificación) de registros dentro de la página es el tipo con el que debemos definir la página. En este sentido y de acuerdo al apartado 4, supongamos que tenemos definidas sendas constantes CABPAGSZ y DATPAGSZ que representan los tamaños de la parte de cabecera y datos de una página.

Frecuentemente, los registros de longitud fija tienen un tipo predefinido y cada instancia de registro se ajusta al tipo predefinido. Considerando que los registros lógicos son del tipo REGT y tamaño REGSZ, y teniendo en cuenta que la página tiene un tamaño fijo PAGSZ, podemos declarar una constante NUMREGPAG que nos indique el número de registros que caben en la página. Esta constante se define en función del valor de DATPAGSZ como:

$$\lfloor \text{DATPAGSZ} / \text{REGSZ} \rfloor$$

Teniendo en cuenta que en C, la división entera trunca el resultado y que numerador y denominador son positivos, podemos definir:

```
#define NUMREGPAG (unsigned) (DATPAGSZ / REGSZ)
```

Como en general el tamaño de la zona de datos de la página no será múltiplo del tamaño del registro es posible que la división DATPAGSZ / REGSZ no tenga resto cero, por lo que tendremos que “pagar” un coste en fragmentación interna equivalente a DATPAGSZ % REGSZ. Así pues, podemos definir una constante FRAGPAG que represente el espacio desocupado de la página.

Con las constantes definidas, podemos ya declarar el tipo de la página. La forma más sencilla de hacer esta declaración consiste en ajustar en tiempo de compilación la estructura de la página al tipo de datos que va a contener. De acuerdo a esta fórmula, la página dispondrá de la zona de cabecera, una zona de datos (un vector de registros de tipo predefinido) y *posiblemente* una zona desocupada (fragmentación interna), de modo que el tipo de página podría definirse como:

```
typedef struct{
        unsigned pagid; //Identificador de página
        unsigned ocupación; //Ocupación actual de la página
        } CABPT;
typedef struct {
        CABPT cab; //Información de cabecera de página
        REGT reg[NUMREGPAG]; // Zona de datos
        char frag[FRAGPAG]; //Zona desocupada
        } PAGT;
```

El problema con la anterior declaración radica en la circunstancia de que el valor de la constante FRAGPAG sea cero. En este caso el compilador nos generaría un error, ya que no es posible definir vectores con cero elementos. Por tanto esta declaración debe ser revisada para que la zona desocupada se represente dentro de la estructura de la página sólo en el caso de que el valor FRAGPAG sea mayor que cero. Esto se haría así:

```
typedef struct {
        CABPT cab; //Información de cabecera de página
        REGT reg[NUMREGPAG]; // Zona de datos
#if (FRAGPAG)
        char frag [FRAGPAG]; //Zona desocupada
#endif
        } PAGT
```

Sin embargo, a la hora de escribir el código resulta más cómodo declarar la cabecera como campos independientes de la página, aunque la declaración anterior resulte más ajustada al concepto que deseamos transmitir. La página finalmente vendrá definida como:

```
typedef struct {
        unsigned pagid;
        unsigned ocupacion;
        REGT reg[NUMREGPAG]; // Zona de datos
#if (FRAGPAG)
        char frag [FRAGPAG]; //Zona desocupada
#endif
        } PAGT
```

La figura 1 muestra ilustra la estructura de página que utilizaremos para codificar registros de longitud fija.

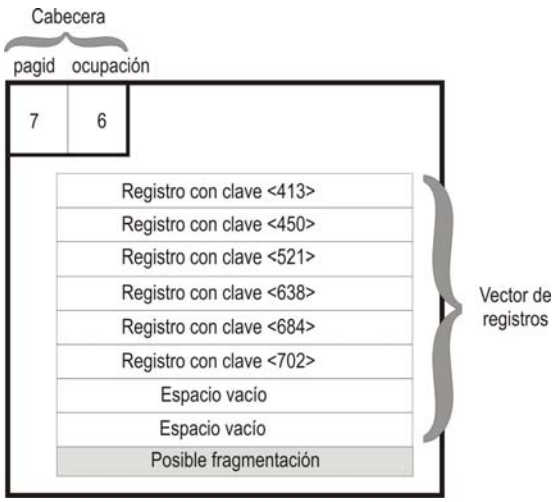

Figura 1. Estructura de una página para codificar registros de longitud fija

Basados en la información previa vamos a plantear algunos ejercicios para experimentar con el proceso de codificación/decodificación de registros lógicos de longitud fija en páginas. Disponemos de un fichero en formato texto denominado "Pedidos.txt", con información sobre pedidos que ejecutan diversas empresas suministradoras. Cada línea de este fichero contiene información relativa a un pedido particular. El tipo de registro se corresponde con la estructura siguiente:

```
typedef struct {
        char codcli[5]; //Código de cliente
        unsigned  idpedido; //Identificador de pedido (clave primaria)
        char nombre[40];
        char ciudad[15];
        FECHAT fecha;
        float importe;
} REGT;
```

Donde el tipo para las fechas es:

```
typedef struct {
 char dia, mes, anio;
 } FECHAT;
```

Disponemos además de una función ObtenSgteReg (FILE *fd, REGT *reg) que nos facilita la extracción de registros del fichero "pedidos.txt", convirtiendo cada línea de este fichero en un registro del tipo indicado. Cada vez que se invoca, esta función accede a la siguiente línea del fichero, devolviendo un valor de 0 en caso de que no existan más líneas a procesar en el fichero de texto.

Ejercicio P2.1. Escriba una función

```
char ColocaRegEnPag(PAGT *pag, REGT reg, unsigned pos)
```

que coloque el registro que le llega por parámetro (reg) en la posición indicada (pos) dentro de la página dada en el primer parámetro. Esta función debe comprobar que la posición indicada (pos) se corresponde con una posición válida dentro de la página. La función devolverá EXITO si la acción tiene éxito y FRACASO en caso contrario.

Ejercicio P2.2. Escriba una función

```
char ObtenRegDePag(PAGT pag, REGT *reg, unsigned pos)
```

que devuelva en el parámetro reg el registro obtenido de la página en la posición indicada. La función debe realizar los mismos controles que la anterior.

Ejercicio P2.3. Codifique la función:

```
char CreaFichero(char *filename,void *cabfile, unsigned int cabsz)
```

para crear el fichero vacío con la página cabecera pasada como parámetro, de tamaño indicado en el tercer parámetro. Devuelve éxito o fracaso dependiendo del resultado.

Ejercicio P2.4. Escriba la función:

```
char ConstruyePedidosDat (void)
```

para construir un nuevo fichero paginado "pedidos.dat", consistente en un conjunto de páginas conteniendo los registros que se almacenan en el fichero "pedidos.txt". La función comenzará creando el fichero paginado vacío, es decir, conteniendo exclusivamente una página cabecera con la información necesaria. Una vez que el fichero ya existe, se abre y se lleva la página cabecera a memoria principal, donde residirá mientras el fichero permanezca abierto recibiendo las actualizaciones necesarias (tales como el número de páginas que contiene en cada momento el fichero). Al fichero "pedidos.dat" se accede secuencialmente mediante sentencias fwrite() y sin operaciones de posicionamiento. Finalmente antes de cerrar el fichero, la cabecera con los datos actualizados, se reescribe en su posición. La función retorna EXITO o FRACASO según su resultado.

Ejercicio P2.5. Codifique la función:

```
char VolcadoPedidosDatARespedTxt(void)
```

cuya misión sea la de acceder secuencialmente al fichero paginado "pedidos.dat" v volcar su resultado a un fichero texto. El contenido de este fichero debe coincidir con el contenido del fichero pedidos.txt. Use si lo desea la función

```
char *RegistroAString(REGT reg)
```

que convierte el contenido de un registro en una cadena de caracteres que luego se puede volcar directamente sobre una línea de un fichero texto.

6. Registros de longitud variable (práctica P22)

En el apartado anterior hemos tratado el modo de codificar/descodificar registros de longitud fija en páginas de un fichero. La condición de invariabilidad en el tamaño de los registros nos ha proporcionado un método sencillo para acomodarlos dentro de las páginas del fichero. Sin embargo, cuando esta condición de invariabilidad no se mantiene, la situación se torna algo más complicada. En este caso, la pérdida de esa constancia, nos impide averiguar --como hacíamos antes-- la página que ocupa el registro *k*-ésimo.

Centrándonos en el marco de una página, la codificación/descodificación de registros genera también algunos nuevos problemas. En principio no es posible conocer cuántos registros podemos colocar en una determinada página, por lo que una posible fórmula para incluir registros en una página sería añadir un nuevo registro justo en la posición siguiente al último registro añadido. Esta fórmula a primera vista tan sencilla puede acarrear complicaciones a la hora de descodificar el registro, ya que al ser de longitud variable, ¿cómo sabemos dónde empieza y dónde termina un determinado registro? Por tanto, antes de decidir un método para colocar los registros en la página, es preciso determinar una forma de poder identificar el comienzo y extensión de un registro dentro de la página.

Aunque existen varios métodos para representar registros de longitud variable dentro de una página, en esta práctica vamos a utilizar uno que aporta muy buenos resultados y no resulta complicado de implementar. La idea básica es utilizar un directorio dentro de la página, con una entrada por registro, desde donde se apunta al final de las tuplas. La página se compone así de tres partes, una cabecera, la zona de datos y el directorio. Mientras la cabecera es una zona con tamaño fijo, las otras dos zonas son variables, de modo que la zona de datos crece en la dirección del fin de la página, el directorio crece desde el final de la página hacia el principio de la misma. La figura 2 ilustra la idea de página con la que vamos a trabajar.

Con objeto de experimentar con este mecanismo de codificación/descodificación de registros de longitud variable, vamos a detallar la estructura de nuestra página. En primer lugar definimos la cabecera de nuestra página:

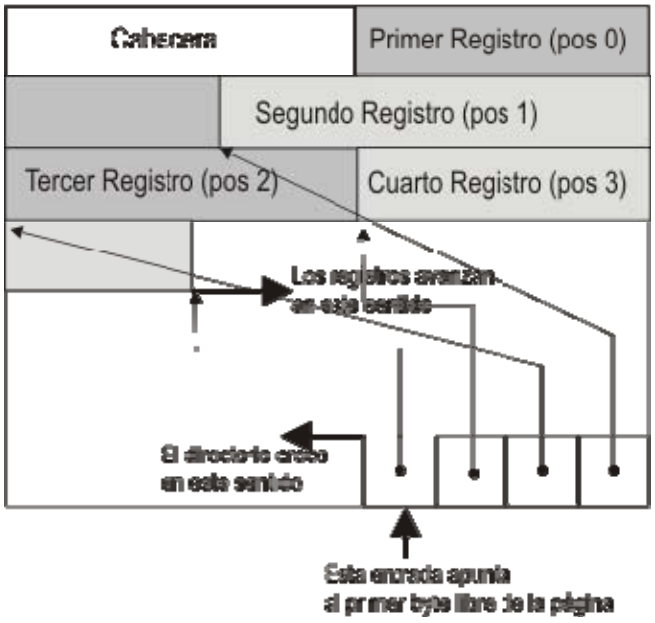

Figura 2: Diseño de una página para almacenar registros de longitud variable.

```
typedef struct {
        unsigned pagid;
        unsigned ocupacion; // Número de registros que contiene
} CABPT;
```

El tamaño de la cabecera es:

```
#define CABPSZ (2*UINTSZ)
```

Y la estructura de nuestras páginas viene definida por:

```
typedef struct {
        CABPT cabp; // La información de la cabecera de página
        char datos[PAGSZ –CABPSZ]; // Resto de la página
        } PAGT;
```

Como puede comprobarse, las zonas de datos y directorio comparten un mismo espacio dentro de la página. Sin embargo para tratar el directorio de la página vamos a utilizar un pequeño truco. Definimos la siguiente estructura:

```
typedef struct {
        char no_necesario[PAGSZ]; // Ocupa toda la página
        unsigned entrada[]; // Permite acceder a cada entrada del
directorio
        } DIRPT;
```

Para acceder al directorio de una página, declaramos un apuntador DIRPT *dir y asignamos a dir la dirección de la página de datos. Esto nos permitirá acceder a los elementos del directorio de la página en cuestión con sólo hacer referencia al valor dir->entrada[-k]. De este modo, la dirección de comienzo del registro situado en la posición cero, será el byte cero de la zona de datos de la página (pag.datos), mientras que la dirección de comienzo del registro en la posición j se almacena en dir->entrada[-j].

Para codificar un registro de longitud regsz dentro de una página, comprobamos primero si hay sitio en la página para colocar el nuevo registro (más una entrada de directorio). Utilizamos para ello una función (que luego implementaremos) denominada BytesLibresPag(pag) que nos devuelve el número de bytes libres que tiene la página. Si se verifica regsz + sizeof(unsigned) $\leq$ BytesLibresPag(pag) entonces procedemos a colocar el registro en el primer byte libre de la página (cuya posición será devuelta por la función PrimerByteLibre(pag), la cual será objeto del ejercicio P2.6), añadiendo posteriormente una nueva entrada en el directorio que apunte al byte inmediatamente posterior al final del registro.

```
dir->entrada[-pag.cabp.ocupacion] = PrimerByteLibre(pag)+regsz;
```

La longitud del registro (regsz) se añade debido a que la computación del primer byte libre se hace sobre la página antes de ser modificada (naturalmente). Para experimentar con la codificación/descodificación de registros de longitud variable en páginas del fichero, vamos a contar con la existencia de un fichero de texto “lvreg.txt”. Este fichero almacena datos sobre registros de longitud variable, que se componen de dos campos, uno numérico (unsigned int) y otro consistente en una cadena de caracteres de longitud variable. Cada línea de este fichero representa los datos de un registro diferente y contiene tres valores (separados por comas). El primero es un número que indica la longitud de la cadena de caracteres, el segundo valor representa la clave del registro y el tercero es la cadena de caracteres de la longitud indicada en el primer valor. La función ObtenSgteRegLongVar(FILE *fd, char *reg, unsigned *regsz) procede leyendo la siguiente línea del fichero de texto y convirtiendo esa línea en un registro, cuyo contenido se devuelve en la dirección apuntada por reg, ocupando un total de regsz bytes. Con la información previamente introducida, estamos ya en disposición de abordar ejercicios para experimentar con la paginación de registros de longitud variable.

Ejercicio P2.6. Escriba las funciones

```
unsigned BytesLibresPag (PAGT *pag)
```

que retorne los bytes disponibles en una página de datos pasada como parámetro. Y

```
unsigned PrimerByteLibre (PAGT *pag)
```

que retorne el primer byte disponible dentro de la página a partir del cual podemos añadir nuevos registros.

Ejercicio P2.7. Escriba una función

```
char AddRegAPag(PAGT *pag, char *reg, unsigned regsz)
```

que añada (siempre que sea posible) el contenido del registro de tamaño regsz situado en la dirección reg a la página pasada como primer parámetro. La función actualizará la cabecera de página, añadirá el nuevo registro a la zona de datos y añadirá una nueva entrada en el directorio de la página. Si la adición tiene éxito, la función retornará EXITO, en caso contrario retornará FRACASO.

Ejercicio P2.8. Escriba una función

```
char ObtenRegDePag(PAGT *pag, char *reg, unsigned *regsz, unsigned pos)
```

para obtener de la página pasada como parámetro el registro situado en la posición indicada en el parámetro pos (¡OJO! se trata de un contador que comienza en 0) almacenando el contenido a partir de la dirección reg pasada como segundo parámetro y su tamaño en el tercer parámetro regsz.

Ejercicio P2.9. Codifique la función:

```
char ConstruyeLvregDat (void)
```

para construir un nuevo fichero paginado “lvreg.dat”, consistente en un conjunto de páginas conteniendo los registros de longitud variable que se almacenan en el fichero “lvreg.txt”. El archivo “lvreg.dat”, se deberá ajustar al modelo de fichero paginado descrito previamente y la forma de construirlo será similar a lo indicado en el ejercicio P2.4.

Ejercicio P2.10. Implemente la función:

```
char VolcadoLvregDatAReslvregTxt(void)
```

que vuelque a un fichero texto “reslvreg.txt” el resultado de procesar secuencialmente el fichero “lvreg.dat” construido en el ejercicio anterior. El fichero de salida “reslvreg.txt” debe coincidir por tanto con el archivo original “lvreg.txt” que sirvió como fuente de datos del ejercicio anterior. Utiliza para ello la función ImprimeReg() que escribe el registro en forma de línea de texto en el archivo de salida con el mismo formato que tienen las líneas en el archivo fuente “lvreg.txt”.

Práctica 3

La indexación simple

1. Introducción

Si bien almacenar registros de datos en un fichero es cosa inicialmente sencilla, no resulta tanto si nuestro objetivo es conseguir un acceso rápido a los distintos registros allí almacenados. En esta práctica vamos a comenzar trabajando con el concepto de indexación simple por clave primaria para conseguir el objetivo mencionado. El punto de partida de esta práctica es el fichero "pedidos.txt", cuya estructura y significado ya ha sido tratado con anterioridad.

El objeto de esta práctica es organizar el fichero de datos de modo que podamos acceder rápidamente a los registros de datos por el valor de su clave primaria (idpedido). Para ello construiremos un **índice simple** capaz de direccionar de forma única cada registro del fichero de datos. El ejercicio para la construcción y utilización de este índice lo llevaremos a cabo implementando cada una de las operaciones a realizar con los datos. Veamos cada una de ellas paso a paso.

2. Diseño de las páginas y estructuras para el índice simple

El primer paso para la construcción y uso de nuestro archivo indexado consiste en diseñar las páginas para el fichero de datos que denominaremos "pedidos.dat". Dado que los registros son de longitud fija, utilizaremos lo aprendido en la práctica anterior sobre paginación de registros de longitud fija. Las páginas del archivo de datos se componen de dos partes: una cabecera con información administrativa sobre la página y un vector con tantas entradas como registros puedan ser almacenados en la página. Eventualmente, añadiremos a la página una zona de fragmentación que nos permita definir la página del tamaño apropiado PAGSZ.

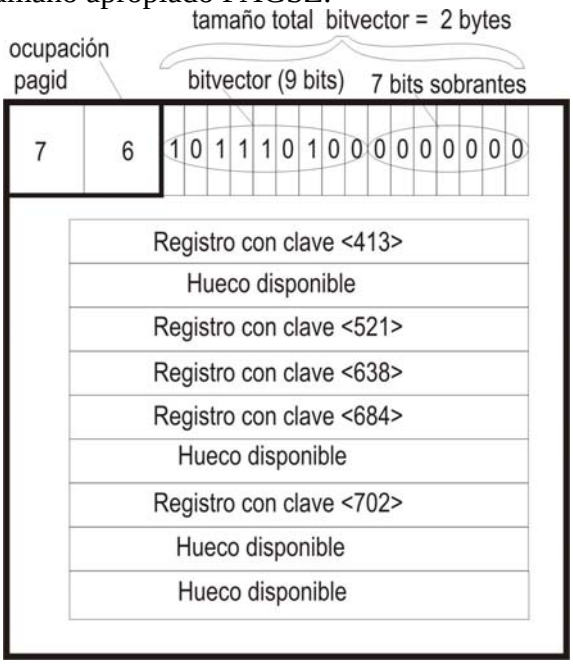

Figura 3. Ejemplo de diseño de una página del archivo de datos "pedidos.dat" con 9 registros de tamaño fijo

Como puede verse en la figura 3, la cabecera de la página se compone de tres campos: el identificador de página, la ocupación de la página (número de registros que actualmente contiene) y un *bitvector*, con tantos bits como entradas contenga el vector de registros de la página. Cada bit de este bitvector informará sobre el estado actual de su correspondiente entrada del vector de registros. Concretamente, un bit con valor 0 en la posición *i* del bitvector indica que la entrada *i* del vector de registros se considera un hueco disponible, mientras que un bit con valor 1 en la posición *j* del bitvector indica que la entrada *j* del vector de registros está ocupada por un registro activo. Dado que la unidad mínima de asignación de espacio es el byte, el tamaño del bitvector vendrá dado por $\lceil numero\_de\_bits/8 \rceil$. Así la figura 3 muestra la necesidad de asignar 2 bytes a un bitvector de 9 bits ($\lceil 9/8 \rceil = 2$). Dado que necesitamos un bit por cada registro, el tamaño del bitvector dependerá del número de registros que quepan en la página. Sin embargo el número de registros que caben en la página depende del espacio disponible para ello, que no es otro que el tamaño de la página (PAGSZ) menos lo que ocupe la cabecera, uno de cuyos campos es el bitvector. Tenemos una recurrrencia que será preciso deshacer.

Con objeto de resolver la recurrencia mencionada, vamos a considerar que el bitvector está fuera de la cabecera, de modo que el tamaño de la cabecera de página es ahora fijo e igual al tamaño de dos enteros (CABPSZ=2*UINTSZ). Como necesitamos un bit por cada registro que pueda almacenarse en la página, vamos a considerar que cada registro lleva asociado un bit, o lo que es lo mismo un octavo de byte (REGSZ + 1/8). De este modo para calcular el número de registros que podemos almacenar en la página (NUMREGPAG), atenderemos a la siguiente inecuación:

$$NUMREGPAG*(REGSZ + 1/8) \leq PAGSZ - CABPSZ$$

despejando el valor en el que estamos interesados, nos queda:

$$NUMREGPAG \leq (PAGSZ - CABPSZ)/ (REGSZ + 1/8)$$

y para evitar la fracción 1/8 (en C la división entera nos resultaría igual a cero), multiplicamos por 8 numerador y denominador:

$$NUMREGPAG \leq 8*(PAGSZ - CABPSZ)/ (8*REGSZ + 1)$$

La anterior inecuación nos invita a declarar NUMREGPAG como el número entero más próximo por defecto al valor indicado por el segundo término de la inecuación anterior, es decir:

$$NUMREGPAG = \lfloor 8*(PAGSZ - CABPSZ)/(8*REGSZ + 1) \rfloor$$

Una vez calculado el número de registros de la página, el tamaño del bitvector (BITVPSZ) no será otro que:

$$BITVPSZ = \lceil NUMREGPAG/8 \rceil$$

Dado que en tiempo de compilación no podemos usar la función techo (ceil) de la biblioteca <math.h>, definimos este valor así:

```
#if (NUMREGPAG % 8)
        #define BITVPSZ (NUMREGPAG / 8) + 1
#else
        #define BITVPSZ (NUMREGPAG / 8)
#endif
```

La única información que nos falta para poder definir el tipo de las páginas de datos es conocer el tamaño de la zona de fragmentación, pero esto ya es tarea sencilla, pues la fragmentación es todo lo que resta hasta el tamaño de página, lo cual viene dado por:

$$PAGSZ - CABPSZ - BITVPSZ - NUMREGPAG*REGSZ$$

Con todo ello, estamos ya en disposición de declarar el tipo de página para nuestro archivo “pedidos.dat”:

```
typedef struct {
        unsigned pagid;
        unsigned ocupacion;
        unsigned char bitvp[BITVPSZ];
        REGT reg[NUMREGPAG];
#if (FRAGPAGDATSZ)
        char frag [FRAGPAGDATSZ];
#endif
        } PAGDT
```

La página cabecera del archivo “pedidos.dat” contendrá la información habitual que se introdujo en la práctica anterior con una diferencia: utilizaremos un bitvector, en donde cada bit nos informe del estado de la correspondiente página del archivo “pedidos.dat”.. Ya que en esta práctica implementaremos la operación de borrado, es muy posible que esta operación genere huecos dispersos entre las páginas del fichero. Con el fin de aprovechar estos huecos para añadir nuevos registros, sería muy conveniente conocer rápidamente si una página del fichero contiene algún hueco disponible para almacenar un nuevo registro. Cada bit del bitvector de la página cabecera nos proporcionará esta información. Para recordar el significado del valor del bit utilizaremos sendas constantes significativas CONHUECOS y SINHUECOS, a las cuales asignaremos los valores 0 y 1, respectivamente. El tamaño de este bitvector dependerá del número de páginas que queramos controlar. Definiremos una constante MAXNUMPAGDATOS que almacene el número máximo de páginas que pueda contener el fichero “pedidos.dat” y utilizaremos el valor de esta constante para declarar el tamaño (en bytes, por supuesto) del bitvector de la página cabecera y finalmente el tipo para la página cabecera del fichero:

```
#if (MAXNUMPAGDATOS % 8)
        #define BITVECTORFILESZ (MAXNUMPAGDATOS /8) + 1
#else
        #define BITVECTORFILESZ (MAXNUMPAGDATOS /8)
#endif
typedef unsigned char BITVECTORFILE[BITVECTORFILESZ];
#define FRAGCABDATOS (PAGSZ - 4*UINTSZ - BITVECTORFILESZ - 50)
typedef struct {
        unsigned pagid;
        unsigned ocupacion;
        BITVECTORFILE bitvector;
        char nombre_fichero[50];
        unsigned pag_sz;
        unsigned reg_sz;
 #if (FRAGCABDATOS)
        char frag[FRAGCABDATOS];
#endif
} CABFILEDT; //Tipo página cabecera archivo “pedidos.dat”
```

Una vez definidas las estructuras para las páginas del archivo “pedidos.dat”, vamos a abordar la declaración de los tipos necesarios para almacenar el índice simple. Antes de embarcarnos en estas definiciones, conviene recordar que el uso de un índice simple tiene sentido en tanto y en cuenta el índice resida enteramente en memoria principal. Con este objeto, vamos a utilizar una tabla (un vector), que declararemos de tamaño suficiente para poder almacenar todas las entradas necesarias.

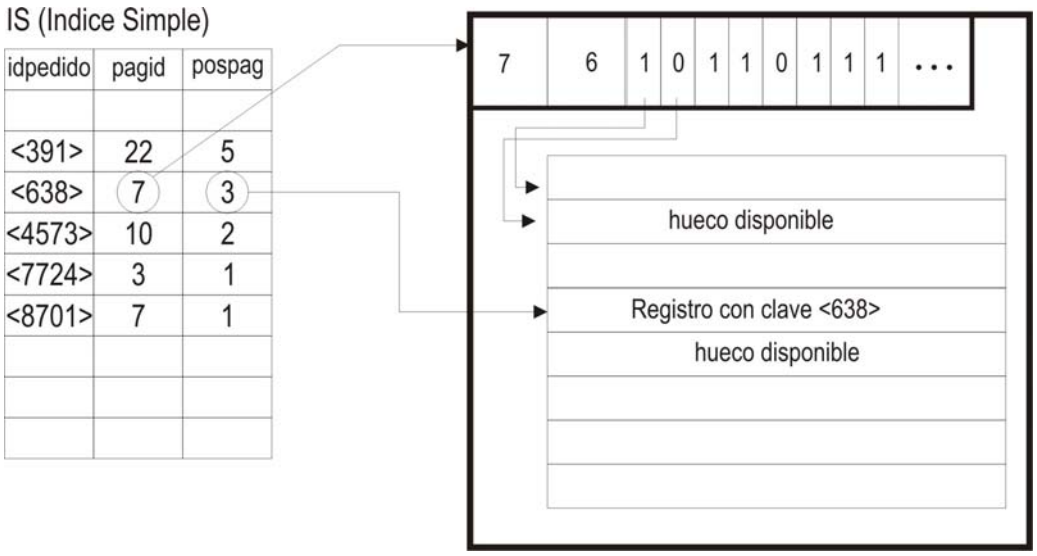

Figura 4. Vista parcial del contenido del índice simple y página de "pedidos.dat".

Dado que nuestro índice será denso, necesitamos disponer en memoria principal una tabla con tantas entradas como registros pueda contener en un momento dado el archivo "pedidos.dat". Así pues, el número máximo de registros que pueda contener el fichero de datos, vendrá dado por la constante NUMMAXREG, la cual será el límite máximo para el número de entradas del índice simple.

Por su parte, cada entrada del índice simple almacenará el valor clave del registro (idpedido) seguido de una referencia a la posición donde se encuentra el registro con dicho valor clave en el archivo de datos. Esta posición la indicaremos mediante el identificador de la página y la posición dentro de dicha página que ocupa el registro. La figura 4 ilustra la forma que tendrá nuestro índice simple.

De acuerdo a lo anterior podemos declarar los tipos para el índice simple como:

```
typedef struct ENTRADAINDICET{
 unsigned idpedido;
 unsigned pagid;
 unsigned pospag;
 } ENTRADAINDICET;
#define ENTRADAINDICETSZ (3*UINTSZ)
typedef ENTRADAINDICET IST[NUMMAXREG];
```

Cuando la aplicación esté en funcionamiento, el índice simple se irá actualizando con las operaciones sobre el archivo de datos, sin embargo cuando la aplicación se cierre, si no contemplamos algún mecanismo de respaldo, la información del índice simple se perderá (está en memoria principal), de modo que la próxima vez que ejecutemos la aplicación tendremos de nuevo que reconstruir el índice. Para evitar este inconveniente y acelerar así el proceso de carga del índice, utilizaremos un archivo —que denominaremos "pedidos.ndx"— donde guardaremos temporalmente (durante el periodo de inactividad de la aplicación) una copia de la información actualizada del índice en el momento en que el usuario decida salir de la aplicación. Tenemos entonces que planificar el diseño de las páginas del fichero "pedidos.ndx" que almacenará la copia del índice. Las páginas de este archivo tendrán como única misión "empaquetar" tantas entradas del índice como sea posible, para ponerlas a salvo en disco mientras la aplicación está inactiva. El proceso que seguiremos para el tratamiento de este archivo "pedidos.ndx" será enteramente secuencial y dado que las entradas del índice simple son de tamaño fijo, aplicaremos todo lo que aprendimos en el desarrollo de la práctica

P2 sobre paginación de registros de longitud fija. Con tales premisas, el tipo para las páginas del archivo “pedidos.ndx” será el siguiente:

```
#define NUMENTRADASPAG (PAGSZ -
2*UINTSZ)/ENTRADAINDICETSZ
#define FRAGPAGIND ((PAGSZ - 2*UINTSZ)%ENTRADAINDICETSZ)
typedef struct {
 unsigned pagid;
 unsigned ocupacion;
 ENTRADAINDICET entrada[NUMENTRADASPAG];
 #if (FRAGPAGINDICE)
        char frag[FRAGPAGINDICE];
 #endif
 } PAGIT;
```

Finalmente la página cabecera del archivo “pedidos.ndx” contiene la información habitual más un campo adicional, denominado numentradas que utilizaremos durante el desarrollo de las operaciones para conocer el número de entradas que actualmente contiene el índice (el índice simple se ha definido con un tamaño máximo, pero en general habrá menos entradas que ocuparán una porción inicial del vector, el número de entradas nos permitirá procesar sólo la porción utilizada del vector). La estructura de la página cabecera del archivo “pedidos.ndx” será entonces:

```
#define FRAGCABINDICE      (PAGSZ - 5*UINTSZ - 50)
typedef struct CABFILEIT {
        unsigned pagid;
        unsigned ocupacion;
        unsigned numentradas;
        char nombre_fichero[50];
        unsigned pagsz;
        unsigned entsz;
#if (FRAGCABINDICE)
        char frag[FRAGCABINDICE];
#endif
} CABFILEIT;
```

3. Creación del fichero de datos

En la práctica anterior aprendimos que antes de trabajar con un archivo de datos, la primera operación a ejecutar es la creación del mismo. Antes pues de comenzar a operar con un archivo de datos es preciso ejecutar una operación de creación del archivo, consistente en añadir al archivo su página cabecera con la información pertinente. Siguiendo este esquema, la primera vez que se ejecute la aplicación, invocaremos una función para la creación del archivo de datos con el que vamos a trabajar en esta práctica. Esta función se encargará de crear el archivo “pedidos.dat” vacío, es decir conteniendo únicamente su página cabecera con la información adecuada. En este aspecto habrá que prestar atención al modo de inicializar el valor del bitvector de la cabecera de página del archivo “pedidos.dat”. Dado que los bits de este bitvector informan sobre la disponibilidad o no de huecos en la correspondiente página de datos del archivo, el bit correspondiente a la página con identificador pagid=0 (la página cabecera) debería inicializarse de modo que no haga elegible a la página cabecera como posible página con huecos, de este modo evitamos seleccionar esta página como destino para un posible nuevo registro. Con tales premisas el bitvector de la página cabecera del

archivo “pedidos.dat” debería inicializarse de modo que el primer bit (el correspondiente a la página con pagid=0) tenga un valor que indique que dicha página no contiene huecos (constante significativa SINHUECOS = 1). Como en la práctica anterior ya hemos codificado funciones para la creación de un archivo de datos, esta operación de creación no debe representar dificultad alguna.

Ejercicio P3.1. Escriba una función

```
void CreaFicheroSistema (void)
```

para crear el fichero “pedidos.dat” vacío.

4. Apertura del sistema

La función anterior nos posibilita abordar una función de apertura completa consistente en disponer el sistema para dar soporte a las operaciones básicas de búsqueda, adición, borrado y modificación de registros. Esta función intentará abrir el fichero de datos para realizar lecturas y escrituras sobre él. Si esta operación fracasa invocamos la función de creación anterior para crear el archivo vacío y poder comenzar sin problemas.

Ya sin problemas abriremos el archivo de datos “pedidos.dat” en el modo conveniente para realizar lecturas y escrituras sobre él, llevándonos inmediatamente la página cabecera a memoria, de modo que sea accesible durante todo el proceso con objeto de mantener actualizada en todo momento su información. Para ello dispondremos una variable global Cabdat, así como la variable global Fd donde mantendremos la información de control del canal correspondiente al archivo de datos. Una vez listo el archivo de datos para recibir operaciones de lectura y escritura, es momento de disponer la información del índice simple.

Durante el modo de operación normal del sistema, el índice simple se va modificando a medida que los registros se insertan y borran del archivo de datos. Cuando el usuario decide finalizar su sesión de trabajo, el índice simple se vuelca sobre un archivo “pedidos.ndx” con objeto de salvaguardar los datos y hacerlos disponibles en cuanto el usuario decida comenzar una nueva sesión de trabajo. Por tanto, cada vez que el usuario inicia la aplicación, la apertura del sistema debe llevar a cabo la carga del índice simple a partir de los datos almacenados en el archivo “pedidos.ndx”. Así pues, después de realizar la apertura del archivo de datos, llega el momento de abrir el archivo de respaldo del índice “pedidos.ndx”. Este fichero podría no existir en disco (por haber abortado inesperadamente la aplicación y no haberse producido la salvaguarda del índice simple), lo que significaría la necesidad de reconstruir el índice simple a partir de los datos del archivo “pedidos.dat”. Si fuera así, procederíamos a inicializar la variable global Cabind (pues al no existir archivo, no tenemos página cabecera) tras lo cual invocaremos la función ReconstruyeIS() para reconstruir el índice simple realizando un *scan* secuencial del archivo de datos. Aquí vienen los primeros ejercicios de esta práctica.

Ejercicio P3.2. Escriba una función

```
char InsertaEntradaISOrden (ENTRADAINDICET ent)
```

que inserte en el orden que le corresponda según el valor de clave (ent.idpedido) la entrada pasada como parámetro en el índice simple (almacenado en la variable global Is). Si se alcanzara el límite máximo del tamaño del índice, la función retornará el valor ERRINDICEEXHAUSTO. En otro caso, retornará EXITO.

Ejercicio P3.3. Escriba una función

```
void ReconstruyeIS(void)
```

que tenga como misión la reconstrucción del índice simple a partir de los datos del archivo "pedidos.dat" (considere abierto el archivo y use como información de control del canal la variable global Fd). Esta función procesará secuencialmente el archivo. Para cada página accederemos a los registros activos que almacene (¡ojo!, los registros en la página no tienen por qué encontrarse en las posiciones iniciales del vector de registros) y construiremos su correspondiente entrada que almacenaremos en la posición que le corresponda dentro del índice simple. Esta función irá actualizando el número de entradas del índice simple en el campo Cabind.numentradas.

Con el índice ya reconstruido, finaliza la operación de apertura dejando el archivo de datos abierto y listo para ser utilizado (variable global Fd).

Si el archivo de respaldo hubiera existido, entonces tendremos que leer la cabecera del mismo (Cabind) y proceder a cargar el índice simple a partir del archivo de respaldo. Esta carga es bien sencilla, ya que los datos en el archivo se guardaron ordenados de acuerdo al valor de clave. Ahora basta leer cada página del archivo y trasladar su contenido al índice simple, teniendo cuidado, como es natural, de no sobrescribir datos ya trasladados de páginas anteriores. Esto lo haremos invocando una nueva función:

Ejercicio P3.4. Escriba una función

```
void CargaIS(FILE *fi)
```

encargada de procesar secuencialmente el archivo de respaldo (cuya variable de control le pasamos como parámetro) con objeto de trasladar su información al índice simple. Estando el índice en memoria listo para su uso, podemos cerrar el fichero de respaldo y destruir su información (función de biblioteca remove en <stdio.h>), evitando así que los datos del archivo de respaldo quedarán inconsistentes ante un aborto inesperado de la aplicación. Si hemos cubierto con éxito todo lo anterior, no nos costará nada abordar finalmente la operación de apertura del sistema:

Ejercicio P3.5. Escriba una función

```
char AperturaSistema (void)
```

encargada de realizar las acciones necesarias para comenzar a operar con los datos del archivo "pedidos.dat" indexado a través de un índice simple. Retornará FRACASO si algo ha fallado y EXITO en caso contrario.

5. Búsqueda de un registro

La operación de búsqueda de un registro se realiza a partir del valor de su clave. Dado un valor de clave, nuestro sistema debe ser capaz de localizar (si es que existe) eficientemente dentro del fichero de datos el registro que tiene dicho valor de clave. Para ello realizaremos una búsqueda binaria en el índice simple hasta encontrar una entrada coincidente con la clave buscada. Si existe, una vez localizada, retornamos la posición del índice simple donde hemos encontrado la entrada. Si no existe, la búsqueda binaria retornará la posición donde debería estar la entrada con dicho valor de clave. Ello nos facilitará –como veremos con posterioridad—la escritura de la función de adición de nuevos registros al archivo de datos. Así pues, antes de escribir la función de búsqueda de registro, codificaremos una función para llevar a cabo la búsqueda binaria en el índice.

Ejercicio P3.6. Escriba una función

```
char BuscaEnIndice (unsigned clave, unsigned *pos)
```

que lleve a cabo una búsqueda binaria en el índice simple. Si existe una entrada con el valor de clave indicado, la función retornará EXITO y almacenará en el parámetro de salida pos la posición donde se encuentra dentro del índice. Si no existe, retorna FRACASO y guarda en pos la posición donde debería estar la entrada con dicho valor de clave dentro del índice.

La función anterior nos facilita enormemente la codificación de la operación de búsqueda de un registro con un valor de clave dada.

Ejercicio P3.7. Escriba una función

```
char BuscaRegistro (unsigned clave, REGT *reg)
```

que implemente la operación de búsqueda de un registro en el archivo de datos. Si el registro existe, accede al fichero y devuelve su valor en el parámetro reg, retornando un valor de EXITO. En caso contrario, retorna FRACASO.

6. Adición de nuevos registros

A lo largo de estas prácticas aprenderemos que todas las operaciones básicas sobre los registros de un archivo comienzan por una operación de búsqueda. Esta singularidad tiene un doble propósito, por una parte una búsqueda previa evita posibles inconsistencias (como una clave primaria duplicada) y por otra parte nos dirige hacia el objeto de la operación. La adición de registros comienza pues con una búsqueda sobre el índice simple. Si la búsqueda tiene éxito, hacemos fracasar la operación, evitando así un problema de clave duplicada. Si la búsqueda fracasa, podemos proceder con la inserción del nuevo registro en el archivo de datos. Como la búsqueda en el índice la implementamos de modo que si el registro no existe, nos permite conocer la posición donde debemos colocar finalmente la entrada en el índice para el nuevo registro, rentabilizamos la operación de búsqueda para que, una vez hayamos compuesto la entrada para el nuevo registro, vayamos directamente a dicha posición y la insertemos allí directamente.

Para poder añadir el nuevo registro, necesitamos buscarle un sitio en el archivo de datos y con tal objeto codificaremos una función para tal fin. Si no quisiéramos reutilizar posibles huecos producto de anteriores borrados, bastaría acceder a la última página del archivo y colocar allí el nuevo registro, o si no hubiera allí espacio, añadir una nueva página y colocar en ella el nuevo registro. Sin embargo, como ya comentamos con anterioridad, nuestro deseo es reutilizar los huecos que han quedado disponibles producto de posibles borrados. Estos huecos se encuentran en posiciones dispersas a lo largo del archivo de datos, de modo que la función encargada de localizar un destino en el archivo para el nuevo registro, deberá identificar las posiciones de tales huecos para determinar el destino final del mismo. Pero toda esta información se encuentra en los vectores de bits que hemos dispuesto en el archivo (tanto en la página cabecera, como en las páginas de datos). El bitvector de la página cabecera nos informa sobre las páginas que tienen huecos y el bitvector de la página informa sobre la posición de tales huecos. Aquí viene el siguiente ejercicio.

Ejercicio P3.8. Escriba la función

```
char LocalizaDestinoRegistro (PAGDT *pag, unsigned *posreg)
```

que localice el destino para un nuevo registro. Si resultara imposible encontrar hueco para el nuevo registro (porque el archivo ya no admita más páginas), retornará un valor de FRACASO. En caso contrario, en el primer parámetro almacenará la página destino y en el segundo la posición donde se debe insertar el nuevo registro. Puede darse el caso de que todas las páginas del archivo estén llenas y no contengan posibles huecos. En este caso, la función se encargará de añadir una nueva página (recuerde que estamos en el caso de que el fichero admite una nueva página) y retornar valores adecuados para los parámetros de salida. Si ha sido posible localizar destino, se retorna EXITO.

Construida la función anterior, la operación de adición de un registro resulta ya sencilla de implementar. Localizado el destino del registro, la función puede proceder a colocar el registro en esa posición (sin olvidar invertir el bit correspondiente para indicar que tal posición antes libre está ya ocupada). Si el registro ocupa el único hueco libre que quedaba en la página, la página se llena y debe reflejarse en el bitvector de la cabecera del archivo. Después de esto sólo resta construir la entrada correspondiente al registro e insertarla en la posición del índice que nos retornó la función de búsqueda. La tarea de insertar una entrada en una posición determinada del índice la podemos implementar mediante la función InsertaEntradaISPos que codificamos en el ejercicio P3.1:

Ejercicio P3.9. Escriba una función

```
char InsertaEntradaISPos (ENTRADAINDICET ent, unsigned pos)
```

que inserte la entrada pasada como parámetro en el índice simple en la posición indicada por el parámetro pos. Si la posición indicada se sale de rango, retorna FRACASO. En otro caso, retornará EXITO.

A partir de este punto resulta trivial abordar la codificación de la operación de adición de un nuevo registro:

Ejercicio P3.10. Escriba una función

```
char InsertaRegistro (REGT registro)
```

que implemente la operación de inserción de un registro sobre el fichero “pedidos.dat” indexado mediante el índice simple. La función retorna EXITO o diferentes constantes de error en función de las diversas circunstancias que puedan surgir (clave duplicada, archivo exhausto,…)

7. Borrado de un registro

La operación de borrado se inicia con una búsqueda por el índice simple a partir de la clave del registro que se desea borrar. Si la búsqueda fracasa, el borrado también. Si la búsqueda tiene éxito, la entrada del índice simple nos proporciona la información necesaria para acceder al registro y poder “borrarlo”. Pero ¿qué significa borrar el registro? Con el esquema que hemos diseñado, borrar un registro significa sencillamente informar de que el hueco que ocupaba dicho registro puede considerarse disponible, ya no está activo. Al borrar entonces un registro, la página se convierte en elegible como destino de un posible nuevo registro, lo cual se refleja modificando el bit correspondiente en la página cabecera del archivo de datos. Después de realizar el borrado “físico” del registro, eliminamos el rastro de este registro en el índice simple. Es decir, eliminamos su entrada del índice simple. Para ello, codificamos una función:

Ejercicio P3.11. Escriba una función

char EliminaEntradaIS (unsigned pos)

para eliminar la entrada del índice situada en la posición indicada por el parámetro. La función desplazará las entradas restantes en posiciones posteriores a la indicada para "tapar" el hueco dejado por la entrada que se elimina.

Estamos ya en disposición de codificar la función que nos implemente la operación de borrado de un registro en nuestro archivo indexado

Ejercicio P3.12. Escriba una función

char BorraRegistro (unsigned clave)

para eliminar el registro cuya clave se le pasa como parámetro. Esta función realizará los ajustes oportunos tanto en el archivo de datos como en el índice simple para hacer desaparecer cualquier rastro acerca del registro que se borró. Retorna posibles valores de éxito o fracaso en función del desarrollo de la operación.

8. Modificación de registros

Cuando tratamos con registros de longitud fija, la operación de modificación de registros no plantea mayores problemas. La única consideración a tener en cuenta es que esta operación debe ser tratada de modo diferente dependiendo de si la modificación afecta o no a la clave primaria, pues ello influye de modo directo en el contenido del índice. Si la clave primaria no se ve afectada por la modificación, el índice tampoco y, por tanto, la operación consiste básicamente en localizar el registro a modificar y realizar la actualización sobre la página en que se encuentre el registro. Si la clave primaria se viera afectada por la modificación, la entrada correspondiente en el índice simple podría quedar fuera de sitio (recuerde que el índice está ordenado por el valor de la clave), lo cual sería desastroso para la funcionalidad de la estructura en su conjunto. Es por ello que una modificación de registro que afecte a la clave primaria se resuelve de manera inmediata convirtiendo la operación en una secuencia de dos operaciones: un borrado del registro con la antigua clave, seguido de una adición del registro con la nueva clave.

Debido a la ausencia de complejidad de esta operación y teniendo en cuenta que su implementación plantea más problemas de forma que de fondo (¿Cómo representamos una transacción de modificación? ¿Informamos del campo que deseamos modificar seguido del valor con el que deseamos modificarlo? ¿Componemos la transacción de modificación como un registro con todos sus campos vacíos salvo aquellos que deseamos modificar?), en esta práctica no abordamos la implementación de esta operación.

9. Cierre del sistema

Cuando el usuario decide abandonar la sesión de trabajo, antes de salir de la aplicación debemos realizar ciertas operaciones que nos faciliten el trabajo en las próximas sesiones. Por una parte, es necesario reescribir la página cabecera del archivo de datos, la cual hemos ido actualizando durante las operaciones. Por otra parte, con el fin de facilitar la carga del índice durante la apertura del sistema, debemos ahora realizar el volcado del índice simple sobre el archivo "pedidos.ndx". Esta operación de salvaguarda la efectuaremos mediante una función específica que implemente el algoritmo que ya contemplamos en la práctica anterior para la creación de los archivos

de datos (ejercicios P2.4 y P2.9), con la única diferencia de que en este caso, los datos de partida se encuentran en un vector y no en un archivo de texto.

Ejercicio P3.13. Escriba una función

```
char SalvaguardaIS (char *filename)
```

que tenga como misión la salvaguarda del índice simple en el archivo indicado (en nuestro caso "pedidos.ndx"). La función creará un archivo nuevo y tras almacenar su página cabecera, procederá a "empaquetar" las entradas de índice en las páginas del fichero, las cuales se irán enviando a disco a medida que se vayan llenando. La función retorna el acostumbrado valor booleano en función del resultado de la operación.

El volcado del índice sobre disco es la operación más compleja del proceso de cierre del sistema y como ya la hemos implementado, podemos abordar sin problemas el siguiente ejercicio:

Ejercicio P3.14. Escriba una función

```
char CierreSistema (void)
```

para realizar las operaciones necesarias antes de dar por finalizada la aplicación.

10. La aplicación

En las secciones anteriores hemos construido una biblioteca de funciones para realizar las operaciones básicas sobre nuestro archivo indexado mediante un índice simple. Ahora llega el momento de construir una pequeña aplicación que nos muestre la funcionalidad de nuestra biblioteca. Con tal fin, lo primero que haremos será construir una función que nos genere el fichero "pedidos.dat" a partir del archivo de texto "pedidos.txt" que contiene la información de los registros que vamos a almacenar. Para ello, utilizaremos la función ObtenSgteReg() que conocemos por la práctica P2. La generación de nuestro archivo comenzará examinando si existe el archivo de datos (se abre y se chequea el resultado de fopen). Si el archivo no existe se crea vacío, ejecutando posteriormente la función de apertura del sistema e invocando la operación de adición de nuevos registros para cada registro extraído del archivo fuente ("pedidos.txt"). Cualquier incidencia que se produzca durante las operaciones, provocará la inserción de una línea explicativa de la incidencia en un fichero de texto "log.txt" (en lugar de mostrarla por pantalla). Así pues la función que se encargue de generar el archivo "pedidos.txt" tendrá que atender el uso del fichero de log. Una vez se hayan cargado todos los registros del archivo fuente, se invocará la función de cierre del sistema y se finalizará la aplicación.

Ejercicio P3.15. Escriba una función

```
char GeneraPedidosDat (void)
```

encargada de generar, a partir de los datos del fichero fuente de texto (NOMFICH_FTE), el archivo "pedidos.dat".

Después de generar el archivo "pedidos.dat" a partir del archivo fuente de texto, finalizaremos esta práctica chequeando el correcto funcionamiento de las operaciones que hemos implementado sobre nuestro archivo "pedidos.dat". Para ello, se facilita la función ProcesaTransacciones ( ) que lee transacciones de inserción, borrado y búsqueda de del archivo "trans.txt" y las procesa de acuerdo a la operación solicitada.

Como ejercicio adicional, pruebe a escribir una función que implemente la operación de modificación de un registro. Modifique las funciones que se incluyen para

interpretar las transacciones con objeto de que la aplicación permita ejecutar transacciones de modificación de registros.

Práctica 4

El árbol B^+

1. Introducción

El árbol B^+ es una de las estructuras que presenta mejor rendimiento si se desea obtener un acceso eficiente a los datos almacenados en un fichero, tanto en accesos directos por la clave como en procesos secuenciales ordenados. En esta práctica nos centraremos en la implementación de las principales operaciones de manejo de esta estructura. A lo largo de esta práctica trabajaremos con los mismos registros de longitud fija con los que hemos experimentado en prácticas anteriores. Al igual que en prácticas anteriores, la información sobre estos registros se almacena en el archivo "pedidos.txt".

El objeto de esta práctica es implementar, mediante una estructura de árbol B^+, las operaciones básicas sobre una colección de registros de longitud fija, cuya información se almacena en el archivo "pedidos.txt". Para simplificar la construcción y utilización de la estructura vamos a centrarnos en operaciones de inserción y búsqueda de registros, obviando operaciones de modificación y borrado de registros.

2. Diseño de las páginas

Vamos a concentrarnos en primer lugar en el diseño de las páginas que formarán parte de nuestra estructura de árbol B^+, tanto de las páginas que conformarán el conjunto secuencia, como de las páginas que constituirán el conjunto índice. Aunque existen diferentes alternativas para implementar una estructura de árbol B^+, en esta práctica nos hemos inclinado por utilizar dos archivos que componen finalmente la estructura en su conjunto. El archivo "pedidos.dat" va a contener las páginas del conjunto secuencia, mientras que el archivo "pedidos.ndx" almacenará las páginas del conjunto índice.

Siguiendo el esquema aprendido a lo largo de estas prácticas todas las páginas de nuestros archivos tendrán un tamaño fijo, representado por la constante PAGSZ. Comenzamos nuestro análisis por las páginas del conjunto secuencia.

Cada página del conjunto secuencia incluirá una cabecera donde almacenaremos el identificador de la página y el número de registros que contiene. Además, dado que las páginas de nuestro conjunto secuencia deben estar doblemente enlazadas, necesitaremos añadir a la cabecera dos campos con las direcciones de las páginas anterior y siguiente. Estas direcciones se registrarán en forma de identificadores de página. El resto de la página estará constituida por un vector de registros con tantos huecos como sea posible, más posiblemente (de modo condicional) una zona destinada a fragmentación con objeto de ocupar el espacio establecido para la página (PAGSZ). La figura 5 ilustra el contenido de las páginas del conjunto secuencia.

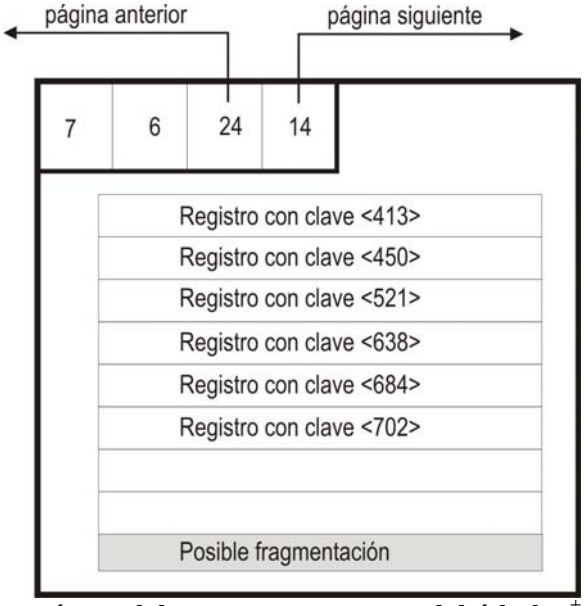

Figura 5. Página del conjunto secuencia del árbol B^+.

Teniendo en cuenta que no vamos a abordar la operación de borrado, no precisamos llevar a cabo operaciones de gestión de espacio libre dentro de nuestros archivos. Esto significa que la página cabecera del archivo "pedidos.dat" no precisa incluir mucha más información que la necesaria para conocer el estado de carga de nuestro archivo. Además de la información que habitualmente incorporamos en esta página cabecera, añadiremos un campo adicional que nos permita conocer la dirección de la página que se sitúa al final de la secuencia en orden lógico del conjunto secuencia que representa el archivo "pedidos.dat" (ya que el principio de la secuencia siempre es la primera página del archivo, es decir aquella con identificador pagid = 1).

En lo que respecta a las páginas del conjunto índice, no debemos olvidar que la información básica que se almacena en las mismas son separadores (usaremos la clave primaria idpedido) y apuntadores a páginas del conjunto secuencia o bien a páginas del siguiente nivel del conjunto índice. En cada página usaremos una información de cabecera compuesta por un identificador de página, un campo ocupación, que se refiere al número de separadores almacenados actualmente en la página y añadiremos un campo que nos informe acerca de la posición de dicha página respecto de la jerarquía en el conjunto índice, ya que las páginas al final de la jerarquía contienen apuntadores a páginas de otro tipo (páginas del conjunto secuencia, las cuales se encuentran en otro archivo distinto), mientras que las páginas en los niveles interiores de la jerarquía (páginas no hoja del conjunto índice) apuntan a páginas del mismo conjunto, lo cual es necesario conocer cuando accedamos a estas páginas con objeto de continuar la búsqueda en el árbol B^+. Así pues este campo adicional (de tipo char) nos dirá si se trata de una página interna (para lo cual usaremos la constante significativa INTERNA) o si es una página hoja del conjunto secuencia (constante significativa HOJA).

Tras la cabecera debemos almacenar el resto de la información. La parte izquierda de la figura 6 muestra de modo conceptual el contenido de una página índice de orden 6. Nuestro objetivo ahora es conformar una estructura apropiada para almacenar en una página de tamaño fijo dicha información. Para ello, aunque existen diferentes alternativas, nosotros optamos por almacenar la información sobre separadores y apuntadores en dos vectores diferentes: uno para los separadores y el otro para los apuntadores, tal y como muestra la figura 6. El orden del árbol B que constituye el conjunto índice nos indica el número de apuntadores que contiene, mientras que el vector de separadores debe contener un elemento menos.

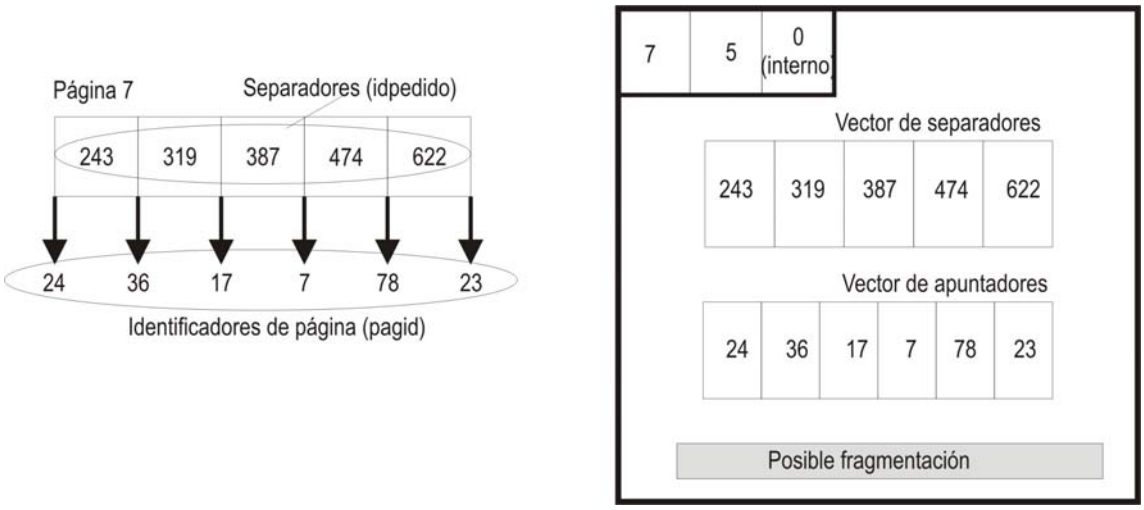

Figura 6. Perspectiva conceptual y estructural de una página del conjunto índice.

Así pues si el vector de apuntadores lo declaramos con un total de ORDEN entradas, el vector de separadores debemos declararlo con un total de ORDEN - 1 elementos. Ahora bien, ¿Cuál debe ser el orden de nuestro árbol B^+?

Es conocido que mientras mayor sea el orden del árbol, menos altura necesitará para indexar un número determinado de registros, lo que nos indica que el orden de nuestro árbol debemos definirlo lo más grande posible, sin embargo el número de apuntadores y separadores que podemos almacenar en nuestras páginas está limitado por el tamaño fijo de éstas (PAGSZ). Por tanto, la respuesta a la anterior pregunta es el número máximo de apuntadores que podamos almacenar en una página de tamaño PAGSZ . Veamos cómo se calcula el orden del árbol B^+.

Llamando SEPSZ al tamaño (en bytes) de un separador y APSZ al tamaño de un apuntador, el tamaño del vector de apuntadores será ORDEN * APSZ, mientras que el tamaño del vector de separadores será (ORDEN – 1) * SEPSZ. De modo que ambos vectores ocuparán en nuestra página un tamaño de ORDEN*APSZ+(ORDEN – 1)*SEPSZ. Ahora bien, el espacio disponible en la página para almacenar ambos vectores es PAGSZ – CABSZ , siendo este último el tamaño de la información de cabecera de página. Así pues para conocer el valor de ORDEN, podemos utilizar la inecuación:

$$ORDEN*APSZ+(ORDEN – 1)*SEPSZ \leq PAGSZ – CABSZ$$

De donde podemos despejar el valor que queremos conocer, dando como resultado:

$$ORDEN \leq (PAGSZ – CABSZ + SEPSZ) / (APSZ + SEPSZ)$$

Por tanto podemos declarar el valor de ORDEN como el entero mayor de todos los que son menores o iguales al segundo miembro de la inecuación, o lo que es lo mismo:

$$ORDEN = \lfloor (PAGSZ – CABSZ + SEPSZ) / (APSZ + SEPSZ) \rfloor$$

Finalmente, como es posible que nos reste alguna fragmentación en la página, declaramos un valor FRAGPAGINDSZ que nos permita ocupar todo el espacio de la página:

$$FRAGPAGINDSZ = PAGSZ – CABSZ - ORDEN*APSZ- (ORDEN – 1)*SEPSZ$$

Con todo lo anterior estamos en disposición de declara el tipo para nuestras páginas del conjunto índice:

```
typedef struct PAGIT{
 unsigned pagid;
 unsigned ocupacion;
 unsigned altura; //¿Interno u hoja?
 unsigned sep[ORDEN-1];
 unsigned ap[ORDEN];
```

```
#if (FRAGPAGINDICE)
        char frag[FRAGPAGINDICE];
#endif
} PAGIT;
```

Finalmente la página cabecera de nuestro fichero índice “pedidos.ndx”, además de la información habitual contendrá dos campos adicionales, uno de ellos nos informará sobre dónde se encuentra la raíz de nuestro árbol B^+, ya que esta página es el punto de entrada de todas las operaciones sobre la estructura, el otro registrará el orden de nuestro árbol B^+.

```
typedef struct CABFILEIT{
 unsigned pagid;
 unsigned ocupacion;
 unsigned raiz; // pagid de la página raíz del árbol
 unsigned orden; // Orden del árbol
 char nombrefichero[50];
 unsigned pagsz; // Tamaño de las páginas
 #if (FRAGCABINDICE)
  char frag[FRAGCABINDICE];
 #endif
} CABFILEIT;//Tipo página cabecera fichero índice
```

3. Creación de la estructura

Igual que hicimos en prácticas anteriores, si la estructura de árbol B+ sobre la que deseamos trabajar no existe, es preceptivo invocar una función para crearla, permitiendo así trabajar con una estructura que aún estando “vacía” existe y por lo tanto se puede manipular.

La creación de la estructura pasará por crear inicialmente los archivos que contendrán el conjunto índice y secuencia del árbol, incorporando sus páginas cabeceras convenientemente inicializadas. Pero además de las páginas cabecera, la estructura de árbol B+ la crearemos colocando una primera página raíz en el conjunto índice y una primera página en el conjunto secuencia, siguiendo el esquema mostrado en la figura 7. De este modo, una operación de búsqueda sobre el árbol B+ (sin datos) finaliza con éxito en la correspondiente página del conjunto secuencia. Procediendo así, evitamos generar código para proceder de un modo extraordinario en caso de que la estructura aún no exista por no haber incluido suficiente número de registros en ella.

Es el momento de comenzar con los ejercicios. El primero que planteamos hace referencia a la creación de la estructura.

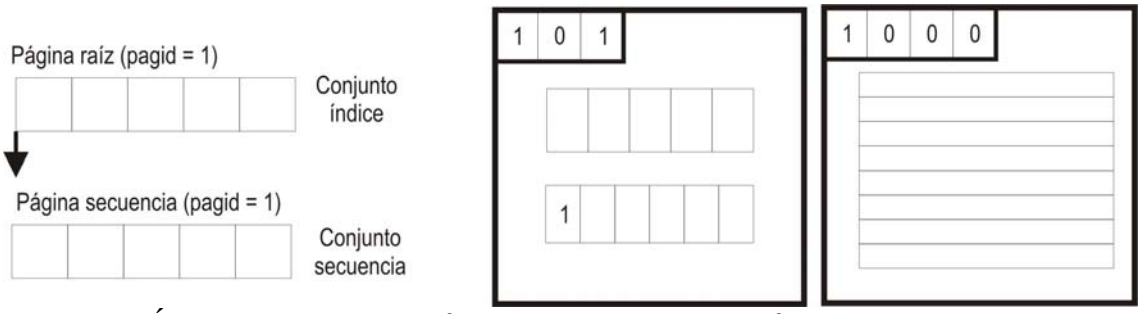

Figura 7. Árbol B+ tras la creación de la estructura. Una búsqueda por la estructura nos lleva directamente a la única página del conjunto secuencia.

Ejercicio P4.1. Escriba una función

void CreaFicherosSistema (void)

encargada de crear la estructura vacía de árbol B^+.

4. Apertura del sistema

Para llevar a cabo la operación de apertura del sistema, nos aseguraremos en primer lugar de que la estructura de árbol B^+ sobre la que deseamos trabajar existe. Para ello una operación de apertura (con un modo diferente al de creación "w") del fichero de datos será suficientemente reveladora. Si operación de apertura fracasa, la función de creación del sistema codificada en el apartado anterior nos posibilita crear la estructura vacía, de modo que una subsiguiente operación de apertura será necesariamente exitosa. Sea cual fuere la circunstancia, podemos entonces considerar que siempre comenzaremos con una situación regular en la que la estructura y los archivos existen y están listos para ser convenientemente abiertos. Con tales premisas, la apertura consiste básicamente en abrir los archivos en modo lectura/escritura y llevar a memoria las páginas cabecera, tanto del conjunto índice como del conjunto secuencia. Además, ya que la página raíz es una página característica del conjunto índice, a la cual se accede para casi todas las operaciones a efectuar sobre el árbol B^+, esta página la llevaremos igualmente a memoria (sobre una variable global) para evitar accesos reiterados al archivo de índice. Recuerde que la dirección de la página raíz se encuentra almacenada en la página cabecera del archivo índice.

Ejercicio P4.2. Escriba una función

void AperturaSistema (void)

encargada de abrir los archivos y dejar la estructura lista para comenzar a realizar operaciones sobre ella.

5. Búsqueda de un registro

La búsqueda exacta de un registro en un árbol B+ a partir del valor de su clave es una operación sencilla. El algoritmo comienza por la raíz del árbol y desciende hasta localizar la página de datos que contiene (o debería contener) el registro con el valor clave indicado. En cada nivel del conjunto índice, la clave buscada se enfrenta con los separadores de la página actual, obteniendo como resultado la dirección de la página por la que debemos proseguir la búsqueda. Para ello se compara el valor de la clave con cada uno de los separadores de la página, cuando la clave buscada sea menor que el separador actual (recuerde que los valores iguales al separador decidimos colocarlo en el hijo derecho), el apuntador situado en esa posición nos indicará la página por la que debemos descender. Por ejemplo, en el caso de la figura 6, si enfrentamos la clave con valor idpedido = 340 con la página actual pagid = 7, la búsqueda por el vector de separadores finaliza en la posición 2. El apuntador almacenado en la posición 2 del vector de apuntadores contiene la dirección de la página por donde deberíamos continuar buscando (pagid = 17). Codificaremos una función para llevar a cabo esta tarea de localizar la dirección de descenso en la búsqueda de un registro.

Ejercicio P4.3. Escriba una función

unsigned ObtenDirDescenso (unsigned idpedido, PAGIT *pag)

que retorne el identificador de la página por la que debemos descender en la búsqueda de un registro, confrontando el valor de la clave que buscamos (idpedido) con el

contenido de la página índice actual cuya dirección de memoria pasamos como parámetro pag.

La función anterior nos facilita el recorrido por las páginas del conjunto índice. Una vez procesada la página hoja, la dirección de descenso que se obtiene hace referencia a una página del conjunto secuencia. Será entonces necesario acceder al archivo que guarda el conjunto secuencia, leer la página indicada y localizar allí dentro el registro deseado. Si el registro no existe, nos quedaremos con la posición que debería ocupar dicho registro en caso de existir (esta información nos será de utilidad durante la operación de inserción de registros). La búsqueda interna en la página del conjunto secuencia la implementamos en una función específica para tal fin.

Ejercicio P4.4. Escriba una función

char ObtenRegDePag (unsigned idpedido, PAGDT *pag, REGT*reg, unsigned *pos)

que localice el registro con clave idpedido en la página del conjunto secuencia cuya dirección de memoria le traspasamos en el parámetro pag. Si lo encuentra, almacena el resultado en la dirección indicada por el tercer parámetro y retorna EXITO. Si no está allí, almacena la posición que ocuparía (si estuviera) dentro de la página en el parámetro pos y retornaría FRACASO.

La escritura de la operación de búsqueda exacta de un registro en el árbol B^+ se simplifica bastante si utilizamos las funciones anteriores. Como la página raíz está visible para toda la aplicación en la variable global Raiz, no tendremos problemas para iniciar la operación. Recorremos el conjunto índice, localizando la dirección de descenso y accediendo a los siguientes niveles, hasta llegar a una hoja. Desde una hoja, accedemos directamente al conjunto secuencia para localizar allí el registro deseado.

Ejercicio P4.5. Escriba una función

char BuscaReg (unsigned idpedido, REGT*reg)

para efectuar la operación de búsqueda exacta del registro con clave idpedido en el árbol B^+. Si lo encuentra, almacena el contenido del registro en la dirección indicada por reg y retorna EXITO. En otro caso, devuelve FRACASO.

6. Inserción de registros

La operación de inserción vamos a codificarla mediante el uso de dos funciones componentes: una función recursiva principal y una función invocante que implementa la operación global de inserción. La función recursiva principal se encarga de efectuar las operaciones necesarias dentro de la estructura. La función invocante llama a la función principal y procede a construir, si es preciso, una nueva raíz para el árbol B^+ en caso de que la estructura precise crecer en altura, o a retornar el resultado de la operación a la función que invocó la operación de inserción.

La inserción de un nuevo registro comienza (como ya sabemos) por un proceso de búsqueda que tiene dos objetivos. El primero es evitar un posible problema de clave duplicada. El segundo es localizar la posición que debe ocupar el nuevo registro una vez comprobado que no existen problemas de clave duplicada. Nótese que el camino descendente que se recorre durante la búsqueda del registro puede tener que ser revisitado (parcial o totalmente) en caso de desborde y división de páginas. Por ello, a diferencia de como hemos procedido en prácticas anteriores, en que durante la

operación de inserción se invocaba la función que servía para realizar la búsqueda, en el caso del árbol B^+, la búsqueda no es una operación aislada dentro de la inserción sino que forma parte intrínseca de aquella. Por tal razón, el algoritmo de inserción no hará una llamada a la función implementada en el ejercicio P4.5, sino que la fase de búsqueda se codifica como parte de la propia operación de inserción.

El algoritmo de inserción de nuevos registros que implementaremos sigue un paradigma de "vuelta atrás" o "*backtracking*". Se trata de un típico algoritmo con tres partes bien definidas: una primera fase que lleva a cabo el procesamiento del árbol durante el descenso (localizando el destino del nuevo registro), una segunda parte consistente en la llamada recursiva y una tercera fase (vuelta atrás) que se encarga de realizar el procesamiento durante la fase de ascenso (inserción de nuevos separadores y apuntadores productos de previos desbordamientos, y posible división de páginas índice). Dado que el árbol B+ se compone de dos conjuntos diferentes de páginas, la recursión procede exclusivamente sobre el conjunto índice. Esto significa que la fase de descenso finaliza cuando se alcanza una hoja del conjunto índice, de modo que sobre el conjunto secuencia se opera "desde un nivel superior". Cuando finaliza la recursión, la página actual que se sitúa en la cima de la pila de recursión es una página hoja del conjunto índice. La operación sobre el conjunto secuencia (inserción del nuevo registro) se realiza fuera ya de la recursión. Si esta operación provoca desborde y división, un nuevo separador y apuntador deben ser insertados en la página índice actual, operación que se repite si es necesario en la fase de vuelta atrás, hasta posiblemente alcanzar la raíz.

Para expresarlo gráficamente, considere un edificio de viviendas con varias plantas (conjunto índice), todas construidas siguiendo el mismo plano salvo la planta baja, que contiene locales comerciales (conjunto secuencia). Nuestro algoritmo, comenzaría en la planta más alta (raíz) e iría descendiendo hasta una vivienda situada en la planta principal (hoja). Como todas las viviendas en esas plantas son similares, se baja siempre de la misma forma (recursión). Ahí finalizaría el descenso, que no el proceso, pues tenemos todavía trabajo que hacer en la planta baja. Desde una ventana de la vivienda en la planta principal tendríamos que operar sobre alguno de los locales comerciales (página del conjunto secuencia) situados abajo. Es posible que el trabajo sobre el local comercial nos obligue (desborde de una página secuencia) a realizar algunos ajustes en la vivienda en que nos encontramos (inserción de un nuevo separador y su apuntador de la derecha), si así fuera, actuamos en consecuencia y si estos ajustes provocan desajuste en la vivienda (desborde y división), la fase de vuelta atrás repite el proceso en las plantas superiores. Esto puede llegar a provocar división en la planta más alta (raíz), en cuyo caso, la función invocante procederá a construir una nueva planta en el edificio (nueva raíz).

Para abordar con mayor comodidad la codificación de la función recursiva principal, vamos a ir escribiendo funciones que nos resuelvan tareas independientes que debemos afrontar durante la inserción.

Cuando finaliza la fase de descenso, leída la página del conjunto secuencia y comprobado que no hay problemas de clave duplicada, el algoritmo debe proceder a insertar un registro en el lugar que le corresponda según el valor de su clave. Este es el objeto del siguiente ejercicio.

Ejercicio P4.6. Escriba una función

```
char InsertaEnPagSec (REGT reg, PAGDT *pag, unsigned pos)
```

que inserte el registro dado en la posición indicada por el tercer parámetro correspondiente a la página almacenada en la dirección dada por el segundo parámetro.

Si la página está llena, se retorna DESBORDE. Si el registro se inserta sin problemas retorna EXITO.

Si el intento de insertar un registro en una página del conjunto secuencia provoca el desborde de la página, se hace necesario dividir la página. La división de página en el conjunto secuencia se puede realizar utilizando como estructura auxiliar un vector de registros con capacidad para un registro más que el factor de bloqueo de la página (es decir, los registros que llenan la página desbordada más el registro que provoca el desborde). Todos los registros implicados en la división se trasladan en el orden correspondiente del valor de clave a la estructura auxiliar, para proceder posteriormente al reparto de los registros entre la página desbordada y una nueva página que debemos añadir al conjunto secuencia. La antigua y la nueva página se "limpian" de datos de modo que queden listas para recibir los nuevos registros y sus apuntadores a las páginas anterior y siguiente en la secuencia se actualizan convenientemente. Tenga presente que al tratarse de una lista doblemente encadenada, será preciso acceder a una página ajena a la división (la página posterior a la página desbordada, si es que existe, o bien la página cabecera, desde donde se apunta al final de la secuencia) para modificarle su apuntador a la página anterior (que tras la división pasa a ser la nueva página). Una vez realizado el reparto equitativo del vector auxiliar de registros entre las páginas de la división, es necesario obtener el nuevo separador de ambas páginas. Este separador, así como el apuntador a la nueva página (situado conceptualmente a la derecha del separador en la página índice padre de la página desbordada, lo que en términos coloquiales denominamos su "apuntador de la derecha"), tendrán que ser capturados y devueltos en parámetros de salida para que la función de inserción pueda posteriormente actuar sobre la página padre. Con esta descripción detallada resulta sencillo resolver el siguiente ejercicio.

Ejercicio P4.7. Escriba una función

char DividePagSec (REGT reg, PAGDT *pagd, unsigned pos, unsigned *sep, unsigned *ap)

para realizar la división de la página desbordada situada en la dirección pagd y correspondiente al conjunto secuencia. El primer parámetro almacena el registro que ha provocado el desborde, mientras que los dos últimos parámetros se utilizan para devolver en sendas direcciones el valor del nuevo separador entre las páginas resultantes de la división, y el identificador de página correspondiente a la nueva página (el apuntador de la derecha). Por su parte el parámetro pos indica la posición que le corresponde (según el valor de clave) al registro dentro de la página desbordada. La función retorna EXITO o FRACASO dependiendo del resultado de la operación.

Los métodos anteriores nos proporcionan toda la funcionalidad necesaria para el trabajo con las páginas del conjunto secuencia. Aún nos quedan por escribir los métodos que nos proporcionen similares funcionalidades dentro de las páginas del conjunto índice. La primera que abordamos es la inserción de una entrada en una página del conjunto índice. En el árbol B+, las entradas dentro del conjunto índice se insertan en todo caso producto de una previa división de páginas en el nivel inferior. Cuando una página se divide, un separador y un apuntador deben ser posteriormente insertados en su página padre. Necesitamos entonces un método que efectúe la inserción de un separador y el apuntador de su derecha en una página del conjunto índice. Es evidente que separador y apuntador se deben insertar en orden del valor del separador en el lugar que le corresponda dentro de la página.

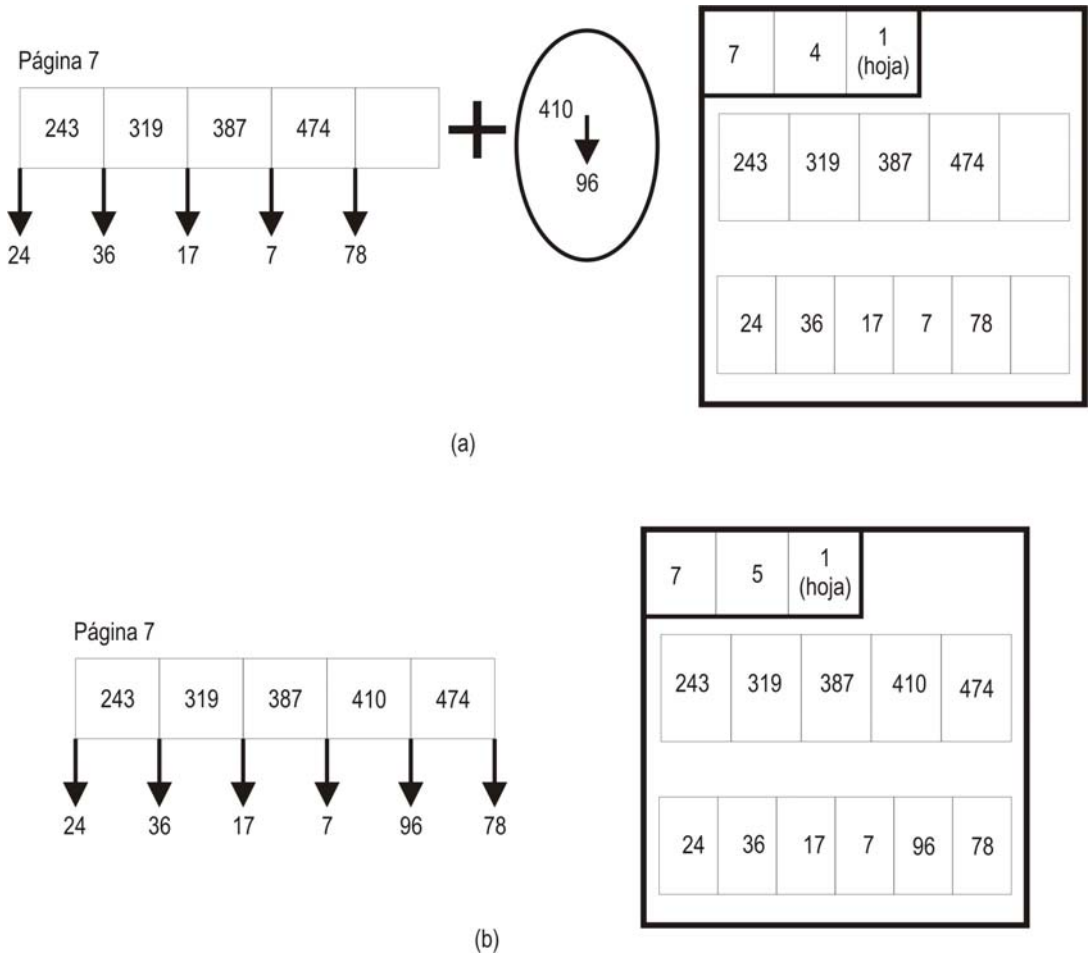

Figura 8. Inserción de un separador y apuntador en una página índice. (a) Situación antes de la inserción. (b) Despúes de insertar separador y apuntador

Esta inserción en orden se puede hacer de forma sencilla a través de un bucle de atrás hacia delante, desplazando elementos (separadores y apuntadores) hacia la derecha mientras el elemento a insertar no encuentre su posición. La figura 8 ilustra la inserción de una nueva entrada en una página índice.

Ejercicio P4.8. Escriba una función

```
char InsertaEnPagInd (unsigned sep, unsigned ap, PAGIT *pagi)
```

que efectúe la inserción del separador y apuntador indicados en los primeros dos parámetros dentro de la página índice cuya dirección en memoria principal viene indicada por el tercer parámetro. Si el intento de inserción provoca una situación de desborde de página, la función retorna DESBORDE. En caso de que se haya insertado sin problemas, la función retorna EXITO.

Si la función de insertar en página índice provoca un desborde, se hará necesario disparar un proceso de división. Necesitamos pues un método que lleve a cabo la tarea de dividir una página índice. Similar a la función que codificamos en el ejercicio P4.7, la división en la página índice se resuelve con la utilización de dos estructuras auxiliares (un vector de separadores y uno de apuntadores) donde colocar todos los elementos que intervienen en la división. A modo de ejemplo, la figura 9 muestra una página del conjunto índice en situación de desborde y la figura 10 ilustra el modo de dividirla desde la dos perspectivas posibles: (a) la conceptual y (b) la de implementación.

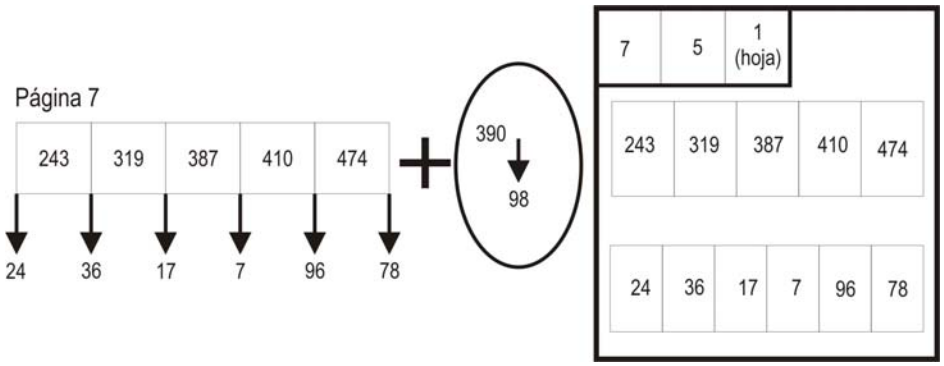

Figura 9: Página del conjunto índice en situación de desborde.

Como se aprecia en la figura 10, una vez que los elementos intervinientes en la división han sido convenientemente colocados en los vectores auxiliares y el separador situado en la mitad del vector ha sido identificado, la información a la izquierda de éste (en la figura los separadores en las posiciones 0 y 1 del vector y los apuntadores en las posiciones 0, 1 y 2), se trasladan a la antigua página, mientras que los elementos a la derecha del separador mitad (separadores en posiciones 3,4 y 5, así como apuntadores en posiciones 3 a 6) se trasladan a la nueva página. Finalmente, después de escribir las páginas en el archivo de índice y actualizar su cabecera, la información de ascenso (en la figura separador 387 y apuntador 20) se retornan al llamador en sendos parámetros de salida.

Ejercicio P4.9. Escriba una función

char DividePagInd (unsigned sep, unsigned ap, PAGIT *pagd, unsigned *sepa, unsigned *apa)

que realice la división de la página índice pagd desbordada por el intento de inserción de los dos primeros parámetros. Los dos últimos parámetros de salida registran el separador y apuntador que deben ascender a la página padre durante el proceso de inserción. La función retonará EXITO si todo va bien y FRACASO en caso de producirse alguna situación anómala.

La anterior batería de funciones que implementan las tareas de bajo nivel que intervienen en el proceso de inserción de registros en un árbol B^+, nos coloca en una situación inmejorable para codificar la función recursiva principal. Recuerde que una función recursiva se escribe de forma mucho más clara si la primera instrucción de la función es la condición de fin de la recursión. Utilizando seudocódigo, la función debe parecerse a esto:

```
Si (Pagind es hoja) //Fin de la recursión
        Leer PagSec //Página del conjunto secuencia
        si (existe reg en PagSec) retorna CLAVE_DUP
        resultado = Inserta reg en PagSec
        si resultado = EXITO retorna HECHO
        datos_ascenso = Divide PagSec
        resultado = Inserta datos_ascenso en PagInd
        si resultado = ÉXITO retorna HECHO
        datos_ascenso = Divide PagInd
        retorna ASCENSO
Obtener dirección descenso
Leer Pagina índice descendiente
retorno = Llamada recursiva con la página índice leída
Dependiendo de retorno
        CLAVE DUP: retorna CLAVE DUP
        HECHO: retorna HECHO
        ASCENSO: Inserta en página actual (PagInd) y actúa en consecuencia
```

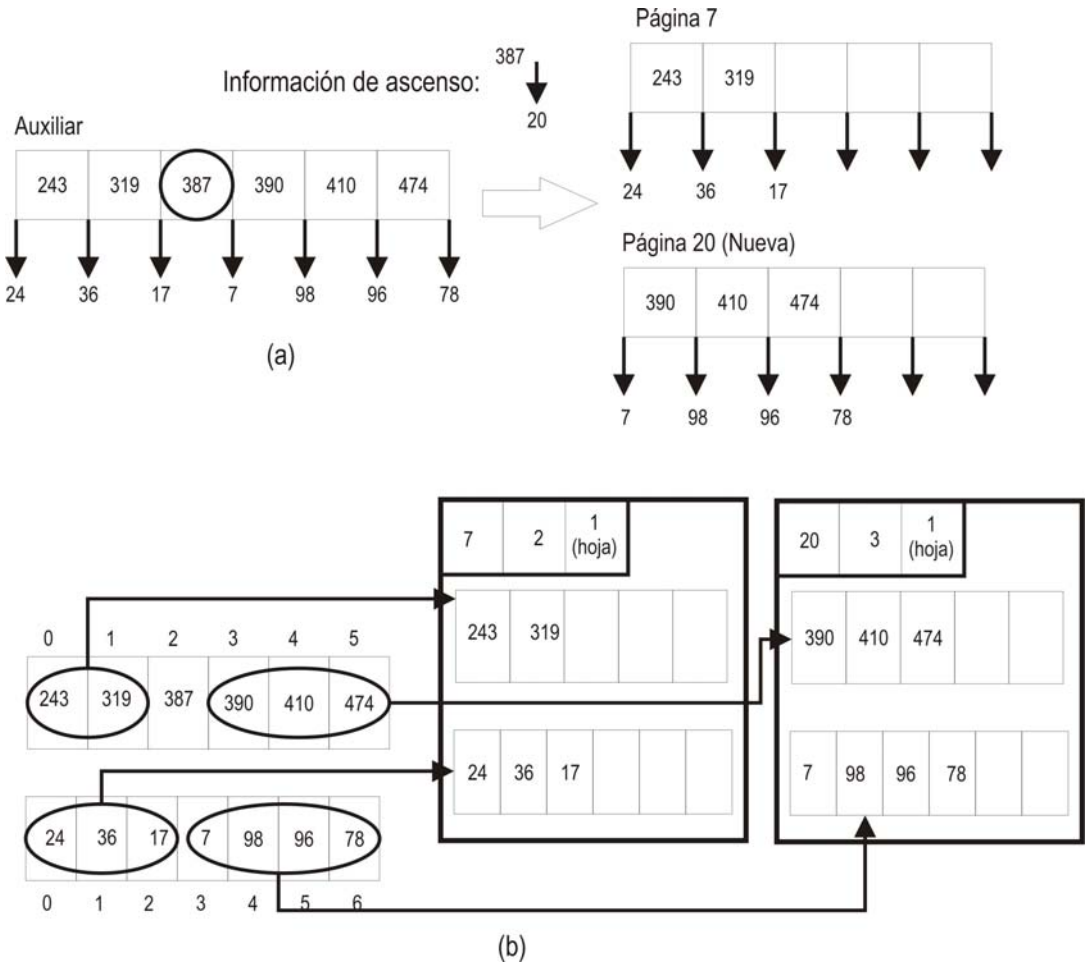

Figura 10. Dos perspectivas del proceso de división de una página del conjunto índice mediante el uso de un vector auxiliar. (a) Vista conceptual. (b) Perspectiva de implementación.

Ejercicio P4.10. Escriba una función

char InsertaRegR (REGT reg, PAGIT *pagact, unsigned *sepa, unsigned *apa)

que lleve a cabo la inserción del registro reg en el árbol B+ enraizado en la página cuya dirección se le pasa en el segundo parámetro. Los dos últimos parámetros representan el separador y apuntador que deben ser trasladados al siguiente elemento de la pila de recursión (o en último caso a la función invocante) en caso de desborde de la página actual. La función puede retornar tres posibles valores: CLAVEDUPLICADA en caso de encontrar un problema tal, HECHO en el caso de que la inserción se haya realizado sin desborde, o ASCENSO en el caso de desborde y división de la página actual. Estos valores dan a conocer al siguiente elemento de la pila de recursión o bien a la función invocante el resultado de la actuación de la función en cada nivel del conjunto índice del árbol B^+. La figura 11 ilustra la forma de proceder de la función anterior mediante un ejemplo.

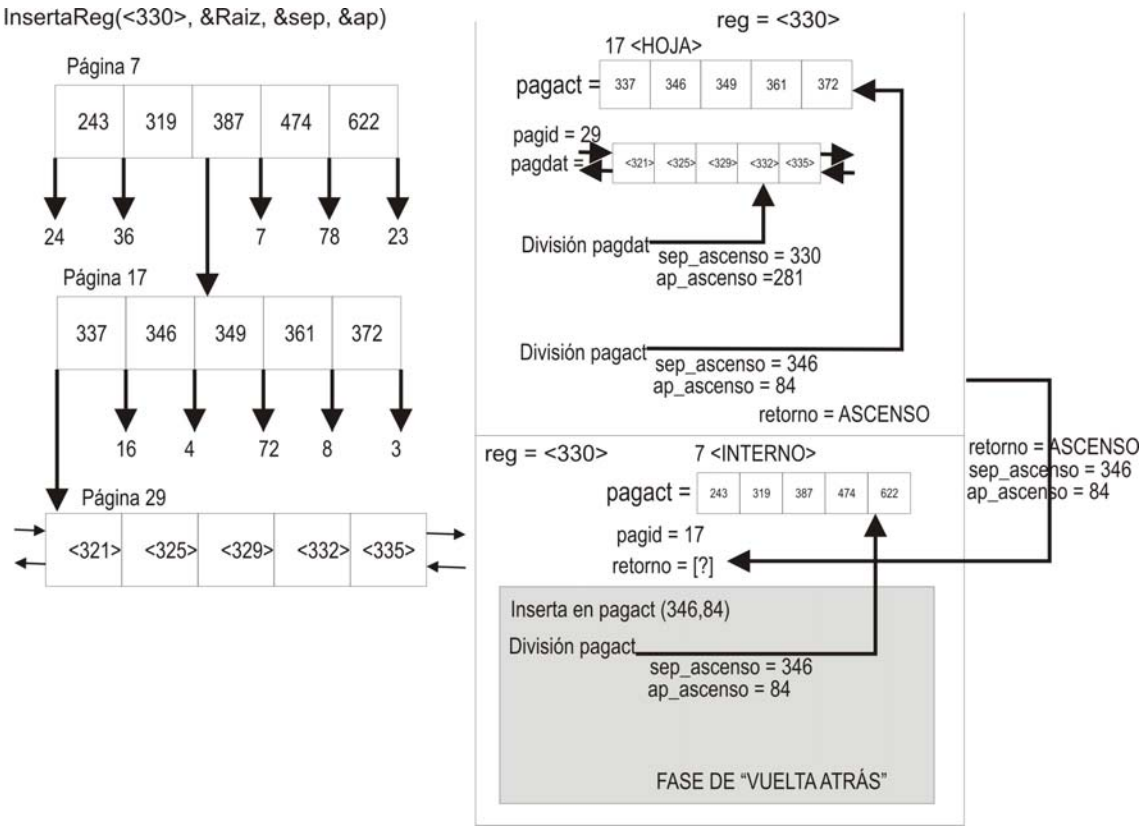

Figura 11. Ejemplo de ejecución del algoritmo recursivo principal InsertaReg. La parte superior de la figura muestra el árbol B+ y la parte inferior la pila de recursión creada durante la ejecución de la función.

Como es posible que el resultado final de la función anterior sea ASCENSO, necesitamos una función invocante capaz de actuar ante tal situación. La principal tarea de esta función será la creación de una nueva raíz y la actualización del árbol para que refleje su crecimiento en altura. Previo a la codificación de la función invocante, conviene escribir una función para realizar la creación de la nueva raíz. Este es el objeto del siguiente ejercicio.

Ejercicio P4.11. Escriba una función

char CreaRaiz (unsigned sep, unsigned api, unsigned apd)

que se encargue de construir una nueva página raíz para el árbol B+ con la información que se le pasa por parámetro, donde el primer parámetro representa el separador y los dos últimos parámetros representan los apuntadores izquierdo y derecho respectivamente del separador anterior. La función debe añadir la página al archivo que almacena el conjunto índice y actualizar la cabecera y la página raíz que se almacena en la variable global. Retorna ÉXITO o FRACASO en función del resultado de la operación.
Después de esto para terminar de escribir la operación de inserción de un registro ya sólo nos resta codificar la función invocante que realiza la operación en su conjunto.

Ejercicio P4.12. Escriba una función

char InsertaReg (REGT reg)

que implementa la operación de inserción de un nuevo registro al árbol B+. Básicamente su misión es llamar a la rutina recursiva principal, capturando su resultado y procediendo en consecuencia. Retorna ÉXITO o FRACASO en función del resultado de la operación.

7. Cierre del sistema

Cuando el usuario decide poner fin a su operación con el árbol B+, la aplicación que gestiona la estructura debe asegurar que todo queda convenientemente listo para una futura utilización de la misma. Tal operación conlleva salvaguardar las cabeceras de ambos archivos así como la página raíz que se mantiene en memoria para un ágil acceso al árbol. Una vez realizadas estas operaciones ya se pueden cerrar los dos ficheros que almacenan la estructura en su conjunto.

Ejercicio P4.13. Escriba una función

void CierreSistema (void)

que realice las acciones pertinentes antes de abandonar la operación con el árbol B+.

8. La aplicación

La batería de funciones que hemos construido a lo largo de las secciones anteriores nos sitúan en una estupenda posición para enfrentarnos al problema de construir una aplicación capaz de generar una estructura de árbol B^+ a partir de los datos del fichero fuente "pedidos.txt" y realizar transacciones sobre el árbol. La generación del árbol la realizaremos mediante la invocación de la función GeneraArbolBMas( ), la cual se incluye en el archivo de utilidades "utilp4.txt".

Las transacciones que procesaremos sobre el árbol generado serán de tres tipos: búsqueda exacta, búsqueda por rango e inserción. Las transacciones se almacenan en un archivo de texto "trans.txt", el cual registra una línea por transacción, que incluye el código de la transacción y la información asociada (dependiendo de la operación). Para agilizar el desarrollo de esta práctica, se facilita la función ProcesaTrans() que procesa este archivo de transacciones sobre la estructura de árbol B generada previamente, de modo que el lector o lectora no tendrá que preocuparse por los detalles de esta función. Lo único que nos queda por resolver es la codificación de la operación de búsqueda en rango en el árbol B. La consulta en rango admite como entrada un rango de valores de clave [kini, kfin] y obtiene como resultado todos los registros del árbol B^+ cuyos valores de clave se encuentren en dicho rango. El modo más eficiente de procesar este tipo de búsquedas es realizar una búsqueda exacta por el valor inferior del rango hasta alcanzar una página del conjunto secuencia. A partir de ahí, basta con seguir los enlaces del conjunto secuencia a las páginas siguientes para localizar aquellos registros que cumplan el predicado. El descenso hasta la hoja puede hacerse iterativo, ya que no precisamos registrar el camino seguido.

Ejercicio P4.14. Escriba una función

char BuscaRango (unsigned min, unsigned max)

que efectúa una búsqueda por el árbol B^+ de todos los registros cuyas claves se encuentren en el intervalo cerrado [min,max]. Al diseñar el algoritmo tenga presente que no es necesario que existan registros con los valores claves min y/o max. Vuelque los registros encontrados que satisfagan el predicado sobre el archivo NOMFICH_SAL, usando para ello la función RegistroAString(). Cree el fichero NOMFICH_SAL cada vez que ejecute esta operación. Dado que esta función formará parte del procesamiento general de transacciones y será invocada por ProcesaTrans(), la función BuscaRango no debe realizar apertura ni cierre alguno de la estructura. Si el rango no es correcto Oo ningún registro satisface el predicado retorna FRACASO. En caso contrario retorna EXITO.

Cuando haya finalizado el ejercicio anterior, invoque la función ProcesaTrans() para procesar las transacciones almacenadas en el archivo “trans.txt” y consulte el resultado de dichas transacciones editando el fichero de log “log.txt” el cual informará del resultado de las operaciones realizadas sobre el árbol B^+.

Como ejercicio adicional a esta práctica P4, pruebe a codificar una función que implemente la operación de borrado en el árbol B^+.

Práctica 5

Hashing

1. Introducción

El objetivo central de una organización de ficheros mediante una técnica hashing es acceder a un registro por valor de clave en un único acceso. A diferencia de las organizaciones de archivo que utilizan algún método de indexación, la organización hash está orientada a la búsqueda exacta, de modo que si la aplicación precisa búsquedas eficientes por rango (localizar registros cuyo valor de clave se encuentra en un rango dado), la organización hash no proporciona respuestas adecuadas. En esta práctica nos fijamos como objetivo implementar, mediante una estructura de hashing, las operaciones básicas de inserción, borrado y búsqueda exacta sobre la ya conocida colección de registros de longitud fija almacenada en el archivo "pedidos.txt".

Antes de abordar el diseño de las estructuras necesarias para conseguir nuestro objetivo, debemos despejar los diferentes parámetros que intervienen en la organización, principalmente la capacidad de los cubos que hospedan los registros y el método de resolución de desbordes. Respecto al primero de los parámetros y con objeto de seguir el modelo utilizado a lo largo de las presentes prácticas, el concepto de cubo lo asimilaremos al concepto de página, de modo que la capacidad de cubo se corresponderá con el número de registros que podamos almacenar en la página (dependiente pues de la constante PAGSZ). Como mecanismo de desborde y con objeto de no complicar en exceso esta práctica, utilizaremos un método sencillo de desborde progresivo. Con esta información podemos ya comenzar a plantear los detalles de nuestra organización de archivo.

2. Diseño de las páginas

Teniendo en cuenta que el método de resolución de desbordes no precisa un área de desborde separada, un único fichero ("pedidos.dat") será suficiente para almacenar todos los registros de nuestra organización. Comentamos a continuación la estructura de las páginas de datos y terminaremos el apartado describiendo la estructura de la página cabecera del archivo.

La necesidad de implementar la operación de borrado, nos obliga a plantear un mecanismo para reutilizar los espacios que quedan disponibles producto de una operación de borrado, durante la inserción de nuevos registros al archivo "pedidos.dat". Siguiendo el mismo esquema que aplicamos en la práctica P3, utilizaremos un bitvector que nos permita distinguir las posibles lápidas que contiene una determinada página. Con ello, la estructura de la página se reduce a:

```
typedef struct PAGT{
 unsigned pagid;
 unsigned ocupacion;
 BITVP          bitv;
 REGT           reg[NTP];
 #if (FRAGPAG)
     char frag[FRAGPAG];
 #endif
 } PAGT;
```

Por su parte la cabecera de página contendrá la información habitual más un campo adicional que utilizaremos para informar sobre el número de registros activos (registros existentes que no están borrados) que se almacenan en el archivo. De acuerdo a ello, la estructura de la página cabecera del archivo se define como:

```
typedef struct CABFILET{
        unsigned pagid;
        unsigned ocupacion;
        unsigned long nregactivos; //Número de registros activos del fichero
        char     filename[50];
        unsigned pagsz; // Tamaño de las páginas
        unsigned regsz; // Tamaño de los registros (caso de longitud fija)
#if (FRAGCABDATOS)
        char resto[FRAGCABDATOS];
#endif
} CABFILET;
```

Como puede apreciarse, la descripción de las estructuras no presenta dificultad añadida alguna sobre lo ya conocido. Proseguimos con las operaciones básicas sobre el archivo hash.

3. Creación y apertura

Como sabemos la función hash tiene la propiedad de transformar un espacio de claves de tamaño considerable en un espacio de direcciones de dimensiones reducidas (adaptadas al número esperado de registros que esperamos tener en el archivo). Por tanto, una de las particularidades de este tipo de organización es la necesidad de conocer a priori el número de direcciones que generará la función hash a partir del valor de la clave. Así pues, para que el resultado de la función hash que apliquemos represente una dirección en el archivo, será preciso disponer de un archivo con tantas direcciones válidas como valores posibles pueda retornar la función hash. Este hecho obliga a que previo a la realización de cualquier operación con los registros de datos, dispongamos de un archivo de datos “pedidos.dat” con tantas direcciones (en nuestro caso páginas, ya que el hashing que implementamos usa cubos) como pueda retornar la función hash.

Por otra parte, sabemos que un parámetro que afecta directamente al rendimiento de una organización hashing es el factor o densidad de carga. El factor de carga hace referencia al grado de ocupación que va a tener nuestro archivo una vez se encuentre en una fase regular de explotación. Para decidir el factor de carga, debemos pues hacer una estimación del número máximo de registros activos que va a contener el archivo en un momento cualquiera de su existencia. Es importante ajustar este valor, ya que si la previsión no es correcta y el número de registros supera la estimación realizada, el archivo ya no podrá asumir más registros, ya que no es posible asignar dinámicamente nuevas páginas al archivo “pedidos.dat” (la función hash no puede direccionar esas nuevas páginas). Por otra parte, una estimación incorrecta puede llevarnos a un deterioro del rendimiento del archivo, bien porque el factor de carga real esté muy por debajo del factor de carga estimado desperdiciando demasiado espacio, o bien por lo contrario, lo cual puede conducirnos a largas cadenas de desborde, provocando un mayor coste en el acceso a los datos.

Dicho esto, el número de direcciones que vamos a asignar a nuestro archivo, lo calcularemos en función de dos parámetros: el número máximo de registros activos que esperamos tener en un instante dado en el archivo (MAXNUMREG), y un factor de carga

que establecemos como constante significativa en nuestra aplicación (FACTORCARGA). Este valor lo establecemos como proporción normalizada en el rango [0,1], de modo que por ejemplo para establecer un factor de carga del 80% (de todos los posibles huecos del archivo sólo se ocupa un 80% con registros válidos), asignamos a la constante anterior un valor de 0.8.

La primera operación a considerar es por tanto la creación del archivo hash. A diferencia del resto de las prácticas, en que la creación consistía en escribir al archivo de datos exclusivamente su página cabecera convenientemente inicializada, en el caso que nos ocupa esta operación consistirá en crear el archivo de datos escribiendo no sólo la página cabecera, sino también las páginas de datos que contendrán nuestros registros (convenientemente inicializadas) en un número acorde a la estimación MAXNUMREG y al valor FACTORCARGA.

Los datos de la página cabecera relativos al nombre del archivo, tamaño de las páginas de datos y tamaño de los registros, son conocidos en tiempo de compilación, y pueden ser directamente asignados al comienzo de la función de creación. Igualmente, dado que aún no hemos insertado registro alguno, ni lo haremos durante el proceso de inicialización, la información sobre el número de registros activos se puede inicializar a cero. Sin embargo el dato relativo a la ocupación, es decir al número de páginas que va a contener el fichero, tendremos que calcularlo como parte de la función. Para calcular este dato, computamos el número de huecos (entendiendo por hueco el espacio para almacenar un registro) que deberíamos disponer en el archivo “pedidos.dat”. Este número de huecos debe ser tal que cuando el fichero alcance la previsión del número esperado de registros activos, se consiga un factor de carga igual al deseado. En otros términos, el número de huecos se corresponderá con el valor MAXNUMREG/FACTORCARGA.

Calculado el número de huecos que queremos hacer disponible en el archivo, estaremos ya en disposición de decidir el número de páginas que precisamos para conseguir esos huecos. El valor resultante obtenido, no será sin embargo el valor definitivo. Como sabemos, la función hash que apliquemos a los registros debe retornar una de estas direcciones antes obtenidas, pero es conocido que si prevemos utilizar la función residuo como función hash, el método reparte más uniformemente el espacio de claves en el espacio de direcciones, si el divisor es un número primo. Con tal premisa, resultaría más efectivo crear el fichero con un número de páginas que fuera número primo. Es por ello, que al valor obtenido anteriormente le calcularemos el número primo más cercano y tomamos dicho valor como número de páginas de datos que contendrá nuestro archivo. El resto de la operación de creación consiste en realizar la escritura de las páginas al archivo, primero la página cabecera y tras ella el resto de páginas de datos convenientemente inicializadas. Podemos plantear ya el primer ejercicio de esta práctica.

Ejercicio P5.1. Escriba una función

void CreaFicheroHash (void)

que lleve a efecto la creación e inicialización del fichero hash“pedidos.dat” vacío.

Con una función capaz de crear el archivo en caso de que éste no exista, la escritura de la función de apertura del sistema es trivial.

Ejercicio P5.2. Escriba una función

void AperturaSistema (void)

que lleve a efecto la apertura del archivo hash dejando el sistema listo para comenzar a realizar las operaciones básicas de inserción, búsqueda y borrado de registros. Esta función se encargará de crear el archivo hash vacío en caso de que aún no exista en el disco.

4. Búsqueda de un registro

La búsqueda de un registro en un fichero hash comienza por obtener la dirección base del registro, es decir aquella que se obtiene como resultado de aplicar la función hash a la clave de dicho registro. La dirección obtenida es un identificador de página a la que habrá que acceder para localizar el registro deseado. Si el registro existe, no fue previamente borrado y no se desbordó de su página base, una búsqueda por el vector de registros de la página nos proporcionará el resultado deseado. Ya que tenemos que examinar el contenido de una página a la búsqueda de un registro concreto, vamos a plantear como primer paso para la codificación de la operación de búsqueda en el fichero hash, la implementación de una función que realice una búsqueda local dentro de una página.

Antes de adentrarnos en la codificación de la búsqueda local en una página, debemos detallar el modo en que los registros se van añadiendo y eliminando localmente en una página de datos. Cuando la página está vacía, los registros nuevos van ocupando posiciones consecutivas dentro de la página, mientras que el campo ocupación de su cabecera se va incrementando para mostrar el número de registros que contiene la página. Por su parte, cuando se elimina un registro de la página no podemos sencillamente hacer un borrado físico, compactando el espacio que ocupaba el registro y dejando el espacio libre al final de la página, pues ello nos impediría localizar posibles registros desbordados de dicha página. En lugar de ello, necesitamos utilizar un mecanismo de lápidas en la página para indicar que determinado registro ha sido eliminado. Las lápidas se implementan mediante un bitvector, utilizando un bit por cada entrada de la página. Un bit activo representa una lápida para la correspondiente entrada del vector de registros, de modo que un bit no activo lo único que nos dice es que la correspondiente entrada no es una lápida, pero igual puede ser un registro activo como un hueco que nunca antes ha sido utilizado. Para averiguar esta información debemos examinar la entrada para ver si ésta tiene el valor con el que inicializamos (todos sus bytes a cero) o se ha visto modificado.

Cuando borramos entonces un registro de la página, activamos su correspondiente lápida en el bitvector de página. En este punto debemos decidir si tras la activación de la lápida conviene decrementar el campo ocupación en la cabecera de página. Si decrementamos su valor, el campo ocupación nos indicará el número de registros “vivos” que contiene la página, pero perderemos a cambio la oportunidad de inferir a partir de este valor las entradas que son huecos vacíos nunca antes utilizados. Ya que esta última característica nos resulta interesante optaremos por no decrementar el campo ocupación tras el borrado de registros. Recuerde que en la página cabecera del archivo disponemos de un campo para mantener el número de registros activos del archivo en su conjunto.

La figura 12 nos muestra a modo de ejemplo una página del archivo hash. Esta página tiene un valor ocupación igual a 5, lo que indica que los registros en las posiciones 0 a 4 del vector de registros no son huecos vacíos, sino que contiene registros que han sido asignados a esa página, aunque algunos de ellos fueron posteriormente borrados. La información sobre los borrados la registra el bitvector, el cual muestra dos lápidas, una para la posición 0 y otra para la posición 4.

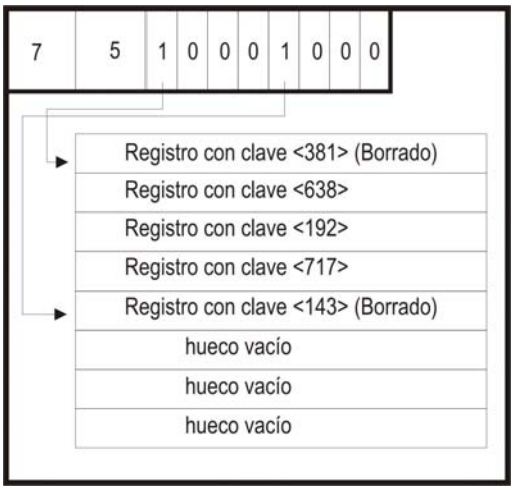

Figura 12. Página de datos del archivo hash "pedidos.dat". La página muestra una ocupación de 5 registros, dos de ellos borrados y tres huecos vacíos que nunca antes han sido ocupados.

De este modo una búsqueda local en dicha página procedería solamente sobre las entradas ocupadas, es decir sobre las entradas en las posiciones 0 a pagina.ocupacion − 1.

Teniendo en cuenta finalmente que la operación de búsqueda la utilizaremos posteriormente para implementar la operación de inserción, implementaremos la búsqueda local de manera que además de localizarnos el registro (o informarnos de que allí no se encuentra el registro buscado), nos indique si ha encontrado en dicha página una lápida. Esta última información nos servirá para reutilizar durante la inserción espacios producto de borrados previos. Podemos ya encarar la escritura de la función de búsqueda local.

Ejercicio P5.3. Escriba una función

char BuscaRegEnPag (unsigned idpedido, PAGT *pag, unsigned *slot, char *lapida)

que realice una búsqueda local del registro con clave idpedido en la página pag. Los dos últimos parámetros son de salida, mientras que slot debe registrar la posición donde se ha localizado el registro (si es que existe), lapida es un valor booleano (con posibles valores EXITO o FRACASO) que debe informar si durante la búsqueda hemos localizado una lápida. Si se localiza el registro, la información sobre la existencia de lápidas en la página es irrelevante. La función retorna EXITO o FRACASO dependiendo de que se haya encontrado o no el registro buscado.

Después de implementar la función anterior, sólo nos resta codificar la función de búsqueda exacta de un registro en el archivo hash. Como adelantamos al inicio del presente apartado, la búsqueda de un registro se inicia con el acceso a su página base. Una búsqueda local nos dirá si la tupla se encuentra allí. Si es así, la operación termina registrando el valor del registro y retornando EXITO. En caso contrario, pueden suceder diferentes circunstancias. Si la página accedida contiene huecos vacíos (nunca antes utilizados) podemos asegurar que el registro buscado no se desbordó de esta página y por tanto no existe en el archivo. Sin embargo, si la página accedida no contiene huecos, aún a pesar de encontrar registros borrados en ella, no podemos asegurar que el registro no se desbordó, por lo que la búsqueda debe continuar en direcciones alternativas, de acuerdo al método de resolución de desbordes que utilicemos.

En el apartado introductorio avanzamos que el método de resolución de colisiones a utilizar será el de desborde progresivo, lo que significa que los registros que

se desbordan de su página base, se intentan alojar en la primera dirección que se localice en páginas siguientes (si la página base fuera la última del archivo, se intenta como página alternativa la primera del mismo). Consecuentemente, si la página accedida durante la búsqueda no tiene huecos vacíos, debemos ir a la siguiente dirección y repetir el proceso con la nueva página accedida. Durante este proceso iterativo, hemos de controlar el caso especial de un archivo que no contenga ningún hueco vacío y tampoco el registro que estamos buscando. Si no tomamos un testigo al inicio del proceso de búsqueda, perderemos la referencia de dónde hemos iniciado la búsqueda y nos quedaríamos dando vueltas indefinidamente por el archivo.

En el caso en que la búsqueda finalice sin éxito, para que nos sea útil durante la inserción, la función nos debe informar de un posible lugar alternativo para almacenar dicho registro. Si durante el recorrido hemos encontrado alguna lápida, la función nos debe informar de ello e indicarnos en qué página la encontramos. En su recorrido una búsqueda infructuosa puede encontrar diferentes lápidas en las páginas accedidas, pero la que nos interesa es la que se encuentre más próxima a la página base del registro buscado. En sendos parámetros de salida, devolveremos la información relativa a la existencia de lápidas durante la búsqueda infructuosa y a la primera página donde localizamos dicha lapida durante el recorrido. Si en el recorrido no se han encontrado lápidas, la función nos debe indicar en qué página exite un hueco vacío.

La figura 13 nos ayuda a comprender mejor la funcionalidad que pedimos a la función de búsqueda. Esta figura muestra un archivo con 6 páginas y un número de registros insertados en él. Vamos a detallar cómo procede nuestra función sobre la búsqueda de diferentes registros. Supongamos que la dirección base de la clave 552 es la página 2. La búsqueda de dicho registro accedería a la página 2 y procedería a una búsqueda local. La búsqueda local fracasa y como la página no tiene huecos, el proceso debe seguir en la siguiente dirección, es decir en la página 3. Se accede a dicha página y se realiza una búsqueda local, la cual es exitosa. Finalmente retornamos éxito y devolvemos el registro <552>, situado en la posición 0 de la página 3. Siguiendo con la figura 13, veamos qué sucedería si buscásemos el registro inexistente con clave 890. Supongamos que la función hash nos retorna como dirección base la página 4. Una búsqueda local en dicha página fracasa. La página no tiene huecos vacíos por lo que hemos de proseguir la búsqueda en la página siguiente (es posible que el registro buscado esté desbordado, aún no lo sabemos), pero resulta que en la página 4 encontramos una lápida. Antes de acceder a la siguiente página, capturamos la dirección de la página 4 e informamos de que ahí tenemos una lápida. Hecho esto, accedemos a la página 5 y realizamos una búsqueda local. En esa página finaliza la búsqueda, pues la página tiene huecos vacíos (lo que nos dice que no hay más desbordes). Sin embargo la página 5 también tiene una lápida en la posición 1. Pero como ya hemos descubierto una lápida anteriormente, ya no registramos la existencia de esta lápida. Podemos entonces retornar un valor de fracaso.

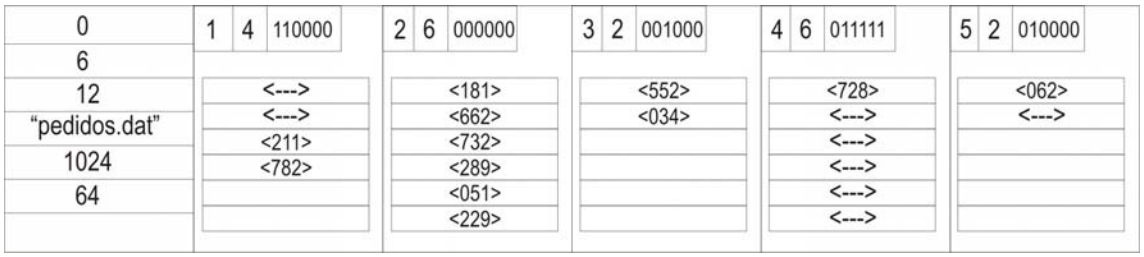

Figura 13. Ejemplo de archivo hash con su cabecera y 5 páginas de datos.

Si esta búsqueda se hubiera realizado como parte del proceso de inserción del registro con clave 890, además de asegurarnos que no tendremos un problema de clave duplicada, estaremos en disposición de reutilizar la lápida encontrada en su página base (página 4) y colocar allí dicho registro. Nótese que si no hubiéramos capturado la primera lápida encontrada, y nos conformáramos con la última lápida encontrada, el registro 890 lo colocaríamos en la página 5, desplazado de su dirección base.

Consideremos como último ejemplo la búsqueda del registro con clave 442 cuya dirección base corresponde a la página 2. La búsqueda de este registros recorre la página 2 y accede a la página 3, donde finaliza por contener huecos vacíos. Antes de retornar un valor de fracaso, además de informar de que no ha encontrado lápidas en su recorrido, debe devolver el identificador de la página 3 como página destino del registro en caso de una operación de inserción.

Es momento de plantearse la codificación de la función que implementa la operación de búsqueda en el archivo hash.

Ejercicio P5.4. Escriba una función

```
char BuscaRegistro (unsigned idpedido, REGT *reg, unsigned *pagid, char *lapida, unsigned
                                                                    *intentos)
```

que efectúe la búsqueda del registro con clave idpedido en el archivo de datos. Si la búsqueda es exitosa, devuelve el contenido del registro en el segundo parámetro y retorna un valor de EXITO. En este caso, los parámetros pagid y lapida resultan irrelevantes. Si por el contrario la búsqueda fracasa, el parámetro lapida nos informa si durante el recorrido hemos encontrado (lapida = EXITO) algún registro borrado. Si es así, el parámetro pagid contiene el identificador de la página donde encontramos la primera lápida disponible. Si no hemos encontrado lápidas durante el recorrido (lapida = FRACASO) el valor de pagid registra el identificador de página donde existe un hueco vacío (ahí habremos finalizado el recorrido). Como es lógico si la búsqueda fracasa se retorna el correspondiente valor de FRACASO. El último parámetro nos dice cuántos accesos hemos realizado durante el recorrido. Este valor nos será de utilidad para calcular posteriormente la longitud media de búsqueda en el archivo.
Los dos últimos parámetros son de salida, mientras que slot debe registrar la posición donde se ha localizado el registro (si es que existe), lapida es un valor booleano (con posibles valores EXITO o FRACASO) que debe informar si durante la búsqueda hemos localizado una lápida. Si se localiza el registro, la información sobre la existencia de lápidas en la página es irrelevante. La función retorna EXITO o FRACASO dependiendo de que se haya encontrado o no el registro buscado.

5. Inserción de un registro

La implementación tan completa que hemos llevado a cabo de la operación de búsqueda de un registro nos facilita enormemente la escritura del algoritmo de inserción de un nuevo registro al archivo de datos. A estas alturas ya sabemos que la operación de inserción de un registro comienza por una búsqueda. Esta búsqueda nos debe proporcionar una doble información. Por una parte, nos evita un potencial problema de clave duplicada, asegurándonos que el fichero sólo contiene un único registro con el valor de clave indicado. Por otra parte nos informa de una posible localización para el nuevo registro. Si la búsqueda del registro a partir del valor de clave del registro a insertar tiene éxito, la inserción debe fracasar evitando así una inconsistencia por duplicidad de clave primaria. En caso de que la búsqueda previa fracase, podemos proceder a insertar el registro en la posición correcta, pero en lugar de iniciar el proceso

de localización de un hueco para el nuevo registro, amortizaremos todo el trabajo invertido en la búsqueda previa. La implementación que hemos realizado de la función de búsqueda nos permite localizar con rapidez la posición que debe ocupar el nuevo registro dentro de nuestro fichero hash "pedidos.dat".

Así por ejemplo, utilizando como referencia la figura 13, si deseamos insertar el registro con clave <117>, tal que h(117) = 2, la búsqueda finalizaría retornando un valor de FRACASO, y aportando como valores de salida los siguientes:

*pagid = 3; *lapida = FRACASO; intentos = 2.

lo que nos permite deducir que la página 3 posee algún hueco para añadir allí el nuevo registro. Por otra parte si deseáramos insertar el registro con clave <533> con h(533)=4, la función de búsqueda fracas reportando como valores de salida:

*pagid = 4; *lapida = EXITO; intentos = 2.

lo que nos facilita la rápida colocación del nuevo registro en alguna de las lápidas que contiene la página 4.

Con los datos aportados, podemos no retrasar más la codificación de la función de inserción.

Ejercicio P5.5. Escriba una función

char InsertaRegistro (REGT reg)

capaz de llevar a cabo la operación de inserción de un registro en el archivo hash. La función retorna tres posibles valores: FRACASO ante un problema de clave duplicada, FICHEROLLENO en caso de que el archivo no pueda almacenar ni un solo registro más y EXITO en caso de haber insertado correctamente el archivo. Recuerde que si el nuevo registro ocupa la posición de una antigua lápida, no debe incrementar la ocupación de página.

Como se puede comprobar, después de haber codificado una operación de búsqueda tan completa, la operación de inserción ha resultado sumamente sencilla. La misma tónica se sigue en la operación de borrado.

6. Borrado de un registro

Si bien en el desarrollo de todas las prácticas anteriores hemos podido comprobar la estrecha conexión que existe entre todas las operaciones básicas de un archivo, en la organización hashing sobre la que trabajamos en esta práctica se ha puesto de manifiesto con enorme claridad. Hasta tal punto ha sido así que para describir la realización de la operación de búsqueda hemos tenido que detallar el modo en que se abordan tanto inserciones como borrados. Es por esto que, llegados a este punto, la operación de borrado está ya suficientemente descrita, por lo que la lectora o el lector de este documento no precisa explicaciones adicionales para implementar con éxito esta operación. Encaramos pues el siguiente ejercicio.

Ejercicio P5.6. Escriba una función

char BorraRegistro (unsigned idpedido)

que implemente la operación de borrado de un registro en el archivo hash. Tras localizar el registro a borrar, la función debe hacer los ajustes oportunos para reflejar el borrado en el archivo. Devuelve éxito o fracaso dependiendo del resultado de la operación.

6. Cierre del sistema

Sin estructuras auxiliares en memoria que debamos salvaguardar antes del cierre de la aplicación, la operación de cierre se convierte básicamente en una sencilla actividad de cierre del archivo, previo volcado de la página cabecera. Aunque es claro que la usaremos, no proponemos ejercicio alguno para implementar esta sencilla operación.

7. La aplicación

Siguiendo el mismo esquema que en capítulos anteriores, vamos a diseñar una pequeña aplicación que nos permita gestionar los datos procedentes del archivo fuente "pedidos.txt" mediante un fichero con organización hashing. El modo de proceder de nuestra aplicación seguirá el modelo conocido de cargar inicialmente el archivo "pedidos.dat" a partir de los datos del fichero texto y realizar posteriormente una serie de transacciones almacenadas en el archivo "trans.txt". Abordamos primeramente la carga del archivo hash a partir de los datos guardados en el archivo fuente.

Ejercicio P5.7. Escriba una función

```
char CargaFicheroHash (void)
```

que realice la carga del archivo "pedidos.dat" a partir de los datos del archivo "pedidos.txt". Utilice el fichero "log.txt" para registrar las posibles incidencias con la operación de inserción. Devuelve éxito o fracaso dependiendo del resultado de la operación.

Disponiendo ya del archivo "pedidos.dat" organizado en hashing, sólo nos resta ejecutar transacciones de búsqueda, inserción y borrado sobre los registros de dicho fichero. Para ello, reutilice algunas de las funciones que se aportaron en los archivos utilp3.txt ó utilp4.txt con objeto de procesar las transacciones de búsqueda (código 's'), inserción (código 'a') y borrado (código 'b') que se almacenan en el archivo "trasn.txt".

Ejercicio P5.8. Escriba una función

```
void ProcesaTrans (void)
```

encargada de procesar las transacciones que se almacenan en el archivo de texto "trans.txt". Las incidencias sobre las transacciones se anotarán en el archivo "log.txt", siguiendo el esquema que se utilizó en las prácticas P3 y P4.

Finalmente, vamos a escribir una función que nos retorne la longitud media de búsqueda en el archivo hash "pedidos.dat". Esta función procede por el archivo hash, accediendo a una página cada vez y realizando una búsqueda por cada registro activo encontrado. Dado que la búsqueda del registro informa sobre el número de accesos realizados durante el recorrido, calculamos el número de accesos totales. Por último retornamos la longitud media de búsqueda.

Ejercicio P5.9. Escriba una función

```
float LongitudMediaBusqueda()
```

que calcule y retorne la longitud media de búsqueda en el archivo "pedidos.dat". Realice ensayos para diferentes archivos hash variando los tamaños de página y la densidad de carga y descubra cómo afectan estos parámetros en el rendimiento del archivo hash.