PROGRAMACION ESTRUCTURADA BASICA

UNIDAD 5. MANEJO AVANZADO DE ARCHIVOS

INDICE

UNIDAD 5 - MANEJO AVANZADO DE ARCHIVOS BINARIOS

OBJETIVOS: *Realizar un manejo avanzado de archivos permitiendo realizar programas que trabajen íntegramente con archivos sin pasar por memoria.*

1. Acceso discontinuo a los archivos.

Con las funciones y los modos de apertura vistos y en la unidad 3 solo era posible acceder a los archivos de forma secuencial, es decir que de forma automática ya sea al leer o al escribir en un archivo el indicador de posición en el archivo siempre se mueve hacia adelante y no se puede volver a un registro que ya fue leído. Tampoco era posible abrir un archivo en modo mixto de lectura/escritura obligando a tener que cerrar el archivo para abrirlo en otro modo.

En esta unidad se utilizarán modos de apertura y funciones para poder acceder de forma no secuencial a los archivos.

1.1 Modos de Apertura

Como se trabajará con archivos binarios los modos de aperturas que se utilizarán serán los de la tabla 1.

Tabla 1: Modos de apertura de archivos binario para acceso no secuencial

Modo	Función	Ubicación inicial en el archivo
r+b	Lectura y escritura, el archivo debe existir	Al comienzo
w+b	Escritura y lectura, crea el archivo y si ya existe lo sobreescribe	Al comienzo
a+b	Agregar y lectura, crea el archivo y si existe no lo sobreescribe ya se posiciona al final para agregar registros. IMPORTANTE: al abrir un archivo en este modo siempre la escritura se hará al final del archivo sin importar si se cambia el indiciador de posición en el archivo utilizando fseek que se explica a continuación	Al comienzo, pero siempre al escribir escribe al final

1.2 Función para reposicionarse en el archivo (fseek)

Las funciones fread y fwrite utilizadas para leer y escribir del archivo hacen que el indicador de posición en el archivo se mueva hacia al final del archivo luego de leer o escribir avanzando de forma automática. Esto es útil ya que una vez que leemos la siguiente lectura nos dará el siguiente registro sin necesidad de realizar ninguna operación adicional. De igual forma al escribir el indicador de posición quedará luego del regitro que se escribe haciendo que si se vuelve a escribir no se pierda la información. Estos modos secuenciales son útiles para realizar una sola operación en simultaneo sobre un archivo, es decir se abre para lectura y siempre se lee o se abre para escritura y siempre se escribe.

Pero ahora con los modos mixtos es posible ir leyendo registros y en algún momento querer escribir por ejemplo para modificar un dato del registro leído y por lo tanto tendremos que poder

regresar el indicador de posición en archivo al registro que acabamos de leer recordando que automáticamente este indicador se movió al registro siguiente.

La función para posicionarse en el archivo es fseek disponible al igual que todas las funciones de archivos en la biblioteca stdio.h. El formato de la función es el siguiente:

fseek (FILE *fp, long int despBytes, int referencia)

donde

fp es el puntero al archivo sobre el cual se desea desplazarse

despBytes es la cantidad de bytes que se desea desplazarse en el archivo

referencia es desde que ubicación se va a realizar dicho desplazamiento pudiendo tomar los valores mostrados en la tabla 2.

Tabla2: valores posibles para la referencia desde donde se realizará el desplazamiento con el fseek

valor	Función	Constante que puede utilizarse en lugar del número
0	Se desplaza desde el comienzo del archivo. Siempre el desplazamiento deberá ser un valor positivo en este caso	SEEK_SET
1	Se desplaza desde la posición actual. El desplazamiento podrá ser tanto positivo como negativo según la ubilicación actual y hacia donde se necesita desplazarse	SEEK_CUR
2	Se desplaza desde el final del archivo. Siempre el desplazamiento deberá ser un valor negativo en este caso	SEEK_END

De esta forma es posible realizar tanto lecturas como escrituras sobre un archivo abierto pero antes de cambiar de modo es necesario reposicionarse en el archivo utilizando fseek o liberar el buffer del flujo con fflush. Es decir inmediatamente luego de un fread no es posible tener un fwrite sin antes realizar un fseek de reposicionamiento (incluso si el desplazamiento es de 0 bytes es necesario realizarlo ya que se limpian flags internos de la estructura FILE). De igual forma luego de un fwrite no será posible realizar un fread sin antes realizar un fseek o realizar un fflush que envíe los datos a escribir del buffer al archivo físico.

1.3 Función para conocer la posición actual en el archivo (ftell)

En ocasiones es necesario conocer la posición actual dentro del archivo y para ello es posible utilizar la función ftell que tiene el siguiente formato

```
long ftell (FILE *)
```

La función ftell recibe un puntero del archivo y retorna un entero indicando la cantidad de bytes desde el origen del archivo (long indica un entero largo pero se recuerda que en el entorno actual el entero largo y el entero son ambos de 4 bytes).

Con el dato de la cantidad de bytes de la posición actual en el archivo es posible calcular el número de registro en el cual se encuentra posicionado dividiendo por el tamaño de cada registro.

Registro actual = ftell(FILE *) / sizeof(estructura que define el tamaño del registro)

1.4 Función para regresar al comienzo del archivo (rewind)

void rewind (FILE * FP);

Esta función hace que el indicador de posición en el archivo vuelva al inicio y resetee todos los flags de la estructura FILE. Por el ejemplo el flag de fin de archivo, el de error, de modo de acceso, etc. También puede realizase esta misma acción utilizando el fseek desplanzandose 0 bytes desde el inicio.

2. Otras funciones para el manejo de archivos

Dentro de la biblioteca stdio.h existen otras funciones que serán de utilidad para el manejo de archivos. Las que se utilizaran están resumidas en a tabla 3

Tabla 3: Funciones adicionales para el manejo de archivos

Función	Utilidad	Uso
int remove (char [])	Eliminar un archivo	Esta función recibe un string correspondiente al nombre físico del archivo que se desea eliminar. Retorna 0 si se eliminó correctamente.
int rename (char nombreViejo[], char nombreNuevo[])	Cambiar de nombre un archivo	Esta función recibe dos strings el primero con el nombre actual de un archivo y el segundo el nombre por el cual se desea cambiarlo. Retorna 0 si se pudo realizar el cambio de nombre correctamente.

3. Actualización de datos sobre un mismo archivo

En unidades anteriores las actualizaciones se realizaban siempre en memoria y luego se sobrescribía el archivo original o se generaba un nuevo archivo con otro nombre. Para actualizar el mismo archivo se debía descargar todo el contenido en un vector en memoria y luego sobreescribir el archivo original. El problema de tener un vector de estructuras en memoria es que se debe conocer la cantidad máxima posible de registros del archivo lo que le quita flexibilidad. Con las funciones incorporadas en esta unidad es posible realizar la actualización sobre le mismo archivo de uno o más registros puntuales.

Para ver la actualización se plantea el siguiente caso de aplicación:

Se dispone de un archivo con precios de productos con el siguiente formato de registro:

- Código de Producto (entero)
- Descripción (texto de 30 caracteres máximo
- Precio Unitario (float)
- Tipo de Producto (char P de fabricación propia, I importado, N nacional)

Debido a un cambio en la normativa impositiva vigente a todos los productos importados se les agrega un costo adicional del 20%.

El procedimiento para resolver este requerimiento será el siguiente:

- Se abre el archivo en modo r+b, es decir en modo lectura para que comience del principio pero que también deje escribir en cualquier posición del archivo.
- Por cada registro que se lee se chequea si es importado.
- Si es importado entonces se incrementa el precio en la variable del tipo estructura en memoria donde se leyó el registro y luego se debe actualizar en el archivo. Para ello se debe mover el indicador de posición del archivo hacia atrás un registro utilizando fseek. Luego se debe escribir el registro
- Se continua con el proceso de lectura chequeando todos los registros hasta el fin del archivo y repitiendo el procedimiento pero se debe recordar que entre cambios de modo lectura-escritura o escritura-lectura se debe realizar un reposicionamiento en el archivo con fseek o un fflush en el caso del paso de escritura a lectura.

A continuación, se muestra el código fuente del proceso descripto:

```c
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef struct
{
     int codigo;
     char descripcion[31];
     float precioUnitario;
     char tipo;
}SProducto;

void MostrarProductos();
void IncrementarPreciosPorTipo(char, int);
int main()
{
     MostrarProductos();
     IncrementarPreciosPorTipo('I',20);

     printf("\n\n---Precios Actualizados---\n");
     MostrarProductos();

     return 0;
}
```

```c
void IncrementarPreciosPorTipo(char tipo, int porInc)
{
    FILE * ap;
    SProducto producto;
    ap = fopen("productos.dat","r+b");
    if (ap==NULL)
    {
        printf("Error al abrir el archivo de productos.");
        getch();
        exit(1);
    }
    fread(&producto,sizeof(producto),1, ap);
    while (!feof(ap))
    {
        if (toupper(producto.tipo)==toupper(tipo))
        {
            producto.precioUnitario += (producto.precioUnitario * porInc)/100;
            fseek(ap,sizeof(SProducto)*-1,SEEK_CUR);
            fwrite(&producto,sizeof(producto),1, ap);
            fflush(ap);
            //fseek(ap,0,SEEK_CUR); //para volver al modo neutro
        }
        fread(&producto,sizeof(producto),1, ap);
    }
    fclose(ap);
}
```

```
void MostrarProductos()
{
    FILE * ap;
    SProducto producto;
    ap = fopen("productos.dat","rb");
    if (ap==NULL)
    {
        printf("Error al abrir el archivo de productos.");
        getch();
        exit(1);
    }
    printf("%6s %-30s %8s %4s","CODIGO","DESCRIPCION","PRECIO","TIPO");
    fread(&producto,sizeof(producto),1, ap);
    while (!feof(ap))
    {
        printf("\n%6d %-30s %8.2f %4c",producto.codigo, producto.descripcion,\
                producto.precioUnitario, producto.tipo);
        fread(&producto,sizeof(producto),1, ap);
    }
    fclose(ap);
}
```

4. Conocer la cantidad de registros de un archivo

Para conocer la cantidad de registros de un archivo es posible utilizar la función ftell vista anteriormente que retorna la posición actual dentro del archivo. El procedimiento es el siguiente:

1. Se abre el archivo en modo lectura
2. Se mueve el indicador de posición de archivo al final utilizando fseek
3. Se invoca a la función ftell para obtener el desplazamiento en bytes desde el inicio de archivo y se divide esa cantidad de bytes por el tamaño de la estructura del formato de registro del archivo,

El siguiente código de ejemplo muestra una función de como obtener el tamaño en bytes de un archivo y la cantidad de regitros considerando que el archivo contiene registros grabados con una estructura llamada SProducto

```
void ContarRegistros()
{
    FILE * ap;
    int bytes,registros;
    ap = fopen("productos.dat","r+b");
    if (ap==NULL)
    {
        printf("Error al abrir el archivo de productos.");
        getch();
        exit(1);
    }
    fseek(ap,0,SEEK_END);
    bytes = ftell(ap);
    registros = bytes /sizeof(SProducto);
    printf("\nCantidad de bytes del archivo: %d", bytes);
    printf("\nCantidad de registros del archivo: %d", registros);
    fclose(ap);
}
```

5. Borrado de registros

Debido a que un archivo es una sucesión de bytes no es posible simplemente eliminar un registro de un archivo. Por lo tanto, para poder borrar registros hay dos formas de trabajo el borrado físico pasando por un archivo intermedio o el borrado lógico.

5.1 Borrado Lógico

El borrado lógico implica agregar a cada registro un campo que actue como indicador de si el registro está vigente o no. Por ejemplo dada la siguiente estructura de productos:

```
typedef struct
{
    int codigo;
    char descripcion[31];
    float precioUnitario;
    char tipo;
}SProducto;
```

Para poder realizar un borrado lógico se debe agregar un campo adicional que indique si ese registro fue eliminado o no. Ese campo puede ser un entero S o N o un número 0 o 1 por ejemplo. Es una definición que debe tomar el programador a la hora de diseñar sus registros. Por ejemplo la estructura podría definirse de la siguiente forma:

```
typedef struct
{
    char eliminado;
    int codigo;
    char descripcion[31];
    float precioUnitario;
    char tipo;
}SProducto;
```

Donde se agrega una variable del tipo char para indicar si el registro fue eliminado o no. En este caso se usa una variable char ya que es la variable que ocupa la menor cantidad de espacio (1 byte) entonces podría guardarse por defecto siempre la letra N indicando que no fue eliminado y cuando se desea eliminar cambiar esa N por una S. También puede guardarse un número 0 o 1 recordando que internamente en realidad las variables char guardan números que luego si se muestran con el formato %c son interpretados de acuerdo con el código ASCII.

En cualquier caso, esta forma de trabajo trae varios problemas:

- Se agrega un campo a cada registro lo que ocupa lugar haciendo que el archivo sea más pesado
- Se deben modificar todas las funciones de lectura y trabajo sobre el archivo para que no tome en cuenta aquellos archivos marcados como eliminados. Lo que hace más lento el proceso ya que se recorren registros que no se utilizan.
- Si la eliminación es frecuente el archivo va a seguir creciendo en tamaño total en bytes con datos que no se utilizan y por lo tanto es recomendable realizar algún proceso que pueda ejecutarse periódicamente que borre físicamente los registros marcados como eliminados para achicar el archivo y dejarlo solo con datos útiles.

5.2 Borrado Físico

Para borrar físicamente un registro de un archivo la única opción es crear un archivo temporal donde simplemente no se copien los registros que se desean eliminar y luego con ese archivo temporal sobre escribir el archivo original.

A continuación, se muestra un ejemplo donde dado un archivo se desea eliminar el registro de un producto dado su código.

El formato de registro del archivo es el siguiente:

```
typedef struct
{
    int codigo;
    char descripcion[31];
    float precioUnitario;
    char tipo;
}SProducto;
```

Y el archivo que contiene los productos se llama productos.dat. A continuación, se muestra el código de la función que realiza el borrado físico copiando todos los registros de un archivo a otro excepto el del código buscado y luego sobrescribe el archivo original.

```c
void EliminarProducto(int codigoABorrar)
{
    FILE * archOrig, *archTemp;
    SProducto reg;

    archOrig = fopen("productos.dat","rb");
    archTemp = fopen("productos.tmp","wb");

    if (archOrig==NULL || archTemp==NULL)
    {
        printf("Error al abrir el archivo de productos.");
        getch();
        exit(1);
    }
    fread(&reg,sizeof(SProducto),1,archOrig);
    while (!feof(archOrig))
    {
        if (reg.codigo != codigoABorrar)
            fwrite(&reg,sizeof(SProducto),1,archTemp);
        fread(&reg,sizeof(SProducto),1,archOrig);
    }

    fclose(archOrig);
    fclose(archTemp);

    //borra el archivo original donde estaba el producto a eliminar
    remove("productos.dat");

    //hace que el archivo temporal pase a ser ahora el arhivo de productos
    rename("productos.tmp","productos.dat");
}
```

Una mejora sobre el código anterior podría ser agregar una bandera indicando si realmente se encontró o no el código a borrar y en caso de que no se haya encontrado entonces no realizar la parte de sobre escritura del archivo original.

5.3 Archivo Histórico

En algunos casos es de utilidad mantener un archivo histórico con los datos borrados por lo tanto antes de borrarlo definitivamente es posible agregar el o los registros a borrar en otro archivo para en caso de contingencia poder acceder a un dato borrado previamente este archivo se abrirá en modo append para ir siempre agregando al final los registros antes de eliminarlos definitivamente del archivo original.

6. Ordenar un archivo

Sobre un archivo pueden aplicarse los mismos métodos de ordenamiento que en memoria, pero considerando que el proceso va a ser mucho más lento ya que se debe recorrer varias veces el archivo completo. A continuación, se explica la aplicación del algoritmo de burbujeo para ordenar el archivo por código.

El algoritmo de burbujeo sobre vectores trabaja mirando la posición actual y la siguiente, pero al trabajar sobre un archivo no es posible acceder directamente al siguiente registro, sino que se deben realizar dos lecturas guardando en dos variables los registros leídos. En el caso de que se

deba realizar el intercambio se debe retroceder 2 registros en el archivo y grabar primero el segundo registro leído y luego el segundo.

Todo algoritmo de orden necesita de ciclos anidados y por lo tanto con un solo recorrido del archivo se logrará asegurarse que un solo registro está ordenado y por lo tanto debe volver a recorrerse para ordenar el resto.

A fin de optimizar el proceso el algoritmo planteado esta optimizado para detectar cuando el archivo queda ordenado (al no realizar intercambios) y además solo recorrer la sección del archivo que le falte ordenar (guardando la posición del último intercambio)

```
void OrdenarArchivo()
{
    FILE * ap;
    SProducto actual, anterior;
    int ordenado, cont;
    int primerRecorrido=1,ultimoIntercambio,intercambio;
    ap = fopen("productos.dat","r+b");
    if (ap==NULL)
    {
        printf("Error al abrir el archivo de productos.");
        getch();
        exit(1);
    }
    do
    {
        ordenado=1;
        fread(&anterior,sizeof(SProducto),1,ap);
        cont =1;
        while ((primerRecorrido && !feof(ap)) || \
            (!primerRecorrido && cont<=ultimoIntercambio))
        {
            fread(&actual,sizeof(SProducto),1,ap);
            cont ++;
            if (!feof(ap))
            {
                if (anterior.codigo>actual.codigo)
                {
                    fseek(ap,sizeof(SProducto)*(-2),SEEK_CUR);
                    fwrite(&actual,sizeof(SProducto),1,ap);
                    fwrite(&anterior,sizeof(SProducto),1,ap);
                    fflush(ap);
                    intercambio = cont;
                    ordenado =0;
                }
                else
                    anterior = actual;
            }
        }
        ultimoIntercambio = intercambio;
        primerRecorrido=0;
        rewind(ap);
    }while(!ordenado);
    fclose(ap);
}
```

La primera vez que recorre el archivo se hace hasta el fin de archivo, si ya estaba ordenado no volverá a recorrerse por el uso de la bandera que cambiar solo si hay algun intercambio.

En los siguientes recorridos ya no se recorrer hasta el fin del archivo sino hasta la posición del último intercambio.

7. Búsqueda en archivos

Sobre el archivo pueden aplicarse los mismos algoritmos de búsqueda que sobre los vectores, pero siempre se debe tener en cuenta que el costo de recorrer un archivo completo es mayor al realizarlo en memoria.

Para separar la lógica es habitual separar la búsqueda en una función, pero en dicho caso según para que se hace la búsqueda habrá que evaluar que retornará la función de búsqueda teniendo las siguientes opciones:

- Retornar el número de registro donde se encuentra el dato buscado o un número de registro inválido sino se encuentra, por ejemplo -1. El problema de este método es que si se quiere acceder a algún dato particular del registro deberá reposicionarse en el archivo y desplazarse hasta el número de registro indicado utilizando fseek desde el inicio del archivo

- Retornar el registro completo que contiene el dato buscado. Esta forma tiene dos posibles inconvenientes, la primera es que no se sabría el número de registro donde se encontró el dato, pero puede ser que no se lo requiera. La segunda es como detectar que no se encontró el registro buscado ya que es necesario retornar igualmente un registro (variable del tipo estructura). Una posible solución sería crear un registro con un valor especial en alguno de los campos (que no pueda existir en ningún registro) para detectar el caso de que no se encuentra.

- Retornar directamente el dato que esta estamos necesitando, generando nuevamente una convención de un dato incorrecto para detectar el caso donde no se encontró el registro. Por ejemplo, si se hace la búsqueda de un producto por su código para obtener el precio del producto entonces la función retorna directamente el precio o -1 sino lo encuentra ya que -1 no es un precio posible.

A continuación, se muestran dos algoritmos de búsqueda diferentes ambos basados en el mismo ejemplo de una búsqueda de producto por su código. El main es el mismo para ambos ejemplos y puede verse a continuación. Luego en las secciones 7.1 y 7.2 se desarrolla el contenido de cada algoritmo de búsqueda. En el ejemplo desarrollado la función de búsqueda retorna una variable del tipo estructura donde sino se encuentra se guarda un -1 en el código de producto.

```c
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef struct
{
    int codigo;
    char descripcion[31];
    float precioUnitario;
    char tipo;
}SProducto;

void MostrarProductos();
SProducto Busqueda(int);

int main()
{
    SProducto prod;
    printf("----Contenido del archivo----\n");
    MostrarProductos();
    int aBuscar;
    printf("\n\nIngrese el codigo a buscar (0 fin):");
    scanf("%d",&aBuscar);
    while (aBuscar !=0)
    {
        prod = Busqueda(aBuscar);
        if (prod.codigo!=-1)
            printf("Producto %s precio es %.2f",\
                    prod.descripcion,prod.precioUnitario);
        else
            printf("No existe un producto con ese codigo.");

        printf("\n\nIngrese el codigo a buscar (0 fin):");
        scanf("%d",&aBuscar);
    }
    return 0;
}
```

7.1 *Búsqueda secuencial*

El algoritmo de búsqueda secuencial sobre archivos es similar al utilizado en vectores, simplemente se debe ir leyendo el archivo en forma secuencial hasta encontrar el registro que coincida por el campo de búsqueda utilizado.

```
SProducto Busqueda(int codABuscar)
{
    FILE * ap;
    SProducto regProducto;
    int encontrado =0;
    ap = fopen("productos.dat","rb");
    if (ap==NULL)
    {
        printf("Error al abrir el archivo de productos.");
        getch();
        exit(1);
    }
    fread(&regProducto,sizeof(SProducto),1,ap);
    while(!feof(ap) && !encontrado)
    {
        if (regProducto.codigo == codABuscar)
            encontrado=1;
        else
            fread(&regProducto,sizeof(SProducto),1,ap);
    }
    if (!encontrado)
        regProducto.codigo=-1;
    fclose(ap);
    return regProducto;
}
```

7.2 Búsqueda binaria

La búsqueda secuencial es lenta ya que recorre uno a uno todos los registros del archivo. Existe un tipo de búsqueda mucho más rápida que es la búsqueda binaria pero que trabajar buscando sobre datos ordenados. Es decir que para poder realizar una búsqueda binaria por un campo de un registro de un archivo es necesario que el archivo esté previamente ordenado por dicho campo.

La búsqueda binaria trabaja reduciendo el conjunto de búsqueda a la mitad con cada lectura. Los pasos para desarrollar el algoritmo son los siguientes.

1 Se cuentan la cantidad de registros del archivo

2 Se lee el registro central del archivo

3 Se compara el dato buscado con el campo del registro leído y según si es mayor o menor ya se descarta la mitad del archivo. Por ejemplo, si se busca el registro con código 500 y el registro central del archivo tienen código 300 se sabe que en la primer mitad del archivo no se va a encontrar el registro con código 500 ya que el archivo está ordenado por dicho código de menor a mayor. Por lo que ahora se reduce la búsqueda a la mitad del archivo.

4 Sino se encuentra se vuelve a calcular cual sería el registro central de los registros que quedan ahora en el archivo y se repite el procedimiento hasta que se encuentre o

hasta que se detecte que no se encontró porque el registro inicial y final de la búsqueda se cruzan indicando que no se encontró.

A continuación, se muestra el código de la función de búsqueda binaria sobre el archivo de productos considerando que previamente se encuentra ordenado de menor a mayor.

```
SProducto Busqueda(int codABuscar)
{
    FILE * ap;
    SProducto regProducto;
    int inicio, fin,mitad;
    int encontrado =0;
    ap = fopen("productos.dat","r+b");
    if (ap==NULL)
    {
        printf("Error al abrir el archivo de productos.");
        getch();
        exit(1);
    }

    inicio = 0;
    fseek(ap,0,SEEK_END);
    fin = ftell(ap) /sizeof(SProducto);
    mitad = (fin+inicio)/2;

    fseek(ap,mitad*sizeof(SProducto),SEEK_SET);
    fread(&regProducto,sizeof(SProducto),1,ap);

    while(inicio <= fin && regProducto.codigo!=codABuscar)
    {
        if (codABuscar > regProducto.codigo)
            inicio = mitad+1;
        else
            fin = mitad-1;

        mitad = (fin+inicio)/2;
        if (inicio <= fin && regProducto.codigo!=codABuscar)
        {
            fseek(ap,mitad*sizeof(SProducto),SEEK_SET);
            fread(&regProducto,sizeof(SProducto),1,ap);
        }
    }

    if (regProducto.codigo!=codABuscar)
        regProducto.codigo=-1;
    fclose(ap);
    return regProducto;
}
```

7.3 Archivos Indexados

Muchas veces es necesario realizar búsquedas sobre distintos campos de los registros de un archivo. Por ejemplo, si se dispone de un archivo de productos sería de utilidad que un usuario pueda buscar un producto por su código o por su descripción. Si se hacen búsquedas secuenciales no habría problema de trabajar sobre el mismo archivo, pero si los datos son muchos los tiempos de búsqueda pueden ser considerables. La búsqueda binaria es mucho más rápida, pero requiere que el archivo esté ordenado por ese campo y se perdería más tiempo reordenando el archivo por un campo u otro en lugar de realizar la búsqueda secuencial.

Una solución a este problema es crear distintos archivos de índices sobre el archivo original y ordenar cada archivo para poder aplicar la búsqueda secuencial.

Por ejemplo, tomando la estructura de productos:

```
typedef struct
{
    int codigo;
    char descripcion[31];
    float precioUnitario;
    char tipo;
}SProducto;
```

Consideremos un archivo con los datos de la tabla 4, donde para facilitar la lectura se agregó de costado el número de cada registro según la posición el archivo original.

Nótese que este archivo no se encuentra ordenado por ningún campo.

Tabla 4: Archivo de productos de ejemplo donde se muestra el número de registro correspondiente a cada registro

	CODIGO	DESCRIPCION	P.UNITARIO	TIPO
1	100	PINZA CURVA	3250.38	I
2	111	PINZA RECTA	1250.50	N
3	200	CABLE X100	5000.00	N
4	210	CABLE COBRE X50	9023.00	I
5	915	TUERCA 24	123.00	N
6	215	CABLE ACERO X10	6234.00	I
7	300	LAMPARA ALOGENA	1234.33	N
8	815	TORNILLO LARGO	114.00	N
9	310	LAMPARA LED	2000.00	N
10	350	LAMPARA RGB	4020.76	I
11	995	MASILLA EPOXI	2320.56	I
12	999	SELLADOR	1020.76	I

Para poder realizar búsquedas binarias (más rápidas) sobre este archivo tanto por código como por descripción se pueden crear dos archivos de índices uno para cada campo. Por ejemplo, el formato de registro para el archivo de indexación por código sería:

```
typedef struct
{
    int codigo;
    int numeroRegistro;
}SIndiceCodigo;
```

El primero paso para crear dicho índice sería recorrer el archivo de productos copiando en un nuevo archivo el código de cada producto y el número de registro correspondiente a cada código. El contenido de dicho archivo quedaría con los registros mostrados en la tabla 5

Tabla 5: Primer paso para la creación de archivo de índices guardar el campo de búsqueda junto con el número de registro

codigo	numeroRegistro
100	1
111	2
200	3
210	4
915	5
215	6
300	7
815	8
310	9
350	10
995	11
999	12

Una vez que se tienen dicho archivo se lo debe ordenar por el campo de búsqueda en este casó por el campo de código quedando ahora el contenido de dicho archivo como los registros de la tabla 6.

Tabla 6: Contenido del archivo de índice ordenado por código de producto.

codigo	numeroRegistro
100	1
111	2
200	3
210	4
215	6
300	7
310	9
350	10
815	8
915	5
995	11
999	12

Al tener este archivo ahora ordenado es posible aplicar búsquedas binarias sobre el archivo reduciendo la cantidad de accesos y recuperando el número de registro del archivo original donde se encuentra el dato que estamos buscando. Luego sobre el archivo original utilizando un fseek desde el comienzo del archivo se puede acceder directamente a dicho registro.

De forma análoga se puede crear un segundo archivo de índices, pero ahora por el campo descripción, para ello se debe crear un segundo archivo con una estructura diferente:

```
typedef struct
{
    char descripcion[31];
    int numeroRegistro;
}SIndiceDescripcion;
```

Y repitiendo el mismo procedimiento se obtendría un archivo ordenado como muestra la tabla 7

Tabla 7. Contenido de ejemplo de un archivo de índices sobre el campo descripción.

descripcion	numeroRegistro
CABLE ACERO X10	6
CABLE COBRE X50	4
CABLE X100	3
LAMPARA ALOGENA	7
LAMPARA LED	9
LAMPARA RGB	10
MASILLA EPOXI	11
PINZA CURVA	1
PINZA RECTA	2
SELLADOR	12
TORNILLO LARGO	8
TUERCA 24	5

Esta forma de trabajo agilizará de forma considerable los tiempos de búsqueda, pero tiene el problema de que ante cualquier actualización del archivo original que incluya algún cambio en los datos indexados o se agreguen nuevos registros, deberán volver a generarse todos los archivos de índices. Otra ventaja es que al tener solo dos campos el proceso de ordenamiento no necesita mover grandes volúmenes de datos como si se trabajara sobre el archivo original.