Universidad Autónoma de Madrid

Departamento de Informática

Fundamentos de Bases de Datos

Práctica - 3: Tablas e Índices

Jorge de las Peñas Plana

Registro de Cambios

Versión[1]	**Fecha**	**Autor**	**Descripción**
1.0	01-09.2025	JPP	Primera versión.

[1]La asignación de versiones se realizan mediante 2 números $X.Y$. Cambios en Y indican aclaraciones, descripciones más detalladas de algún punto o traducciones. Cambios en X indican modificaciones más profundas que o bien varían el material suministrado o el contenido de la práctica.

Índice

1. Objetivo

En esta práctica implementaremos una base de datos muy simplificada. En ella los datos se almacenarán como registros binarios, se mantendrá un índice en memoria e implementaremos las funciones de: búsqueda, borrado e inserción siguiendo diversas estrategias (*first fit*, *last fit* y *best fit*).

Aunque en esta práctica se mantiene el índice en memoria notad que esta aproximación no es muy realista puesto que es complicado mantener un índice en memoria cuando la base puede ser accedida concurrentemente por varios usuarios.

Como en la práctica anterior los ficheros auxiliares se encuentran en Moodle.

2. Estructura de los ficheros de datos e índices

2.1. Fichero de datos

Vamos a modelar la base de datos de una biblioteca, en particular nos concentraremos en la información relacionada con los libros. Cada libro se almacenará en un registro diferente el cual contendrá los cuatro campos siguientes: identificador del libro (*bookID*), ISBN (*isbn*), título (*title*), editorial (*printedBy*). Por ejemplo, `12345, 978-2-12345680-3, El Quijote, cátedra`. *bookID* es un entero (integer) que debe almacenarse en formato binario, *isbn* es una cadena de 16 caracteres y finalmente *title* y *printedBy* son cadenas de longitud variable. Cada registro viene precedido por un entero de tipo *size_t* donde se almacena el tamaño total del registro sin incluir los 8 bytes que ocupa el mismo y finalmente todos los campos de longitud variable acaban con el separador | excepto el último.

El registro debe almacenarse como datos binarios, por lo tanto no deben almacenarse las variables que contienen el tamaño del registro o el identificador del libro como ASCII. De esta forma, y asumiendo que el tamaño de un entero son 4 bytes y el de un *size_t* 8, nuestro registro se almacenaría en disco como:

0	26 00 00 00 00 00 00 00	38
8	3A 30 00 00	12346
12	39 37 38 2D 32 2D 31 32 33 34 35 36 38 30 2D 33	978-2-12345680-3
28	45 6C 20 75 69 6A 6F 74 65 7C	El Quijote|
39	43 61 74 65 64 72 61	Catedra
46	24 00 00 00 00 00 00 00	36
54	39 30 00 00	12345
58	39 37 38 2D 32 2D 31 32 33 34 35 30 38 36 2D 33	978-2-12345086-3
74	4C 61 20 62 75 73 63 61 7C	La busca|
83	43 61 74 65 64 72 61	Catedra

Tabla 2: Descripción gráfica de la estructura del fichero de datos mostrando dos registros. Para cada campo se muestra su offset en el fichero de datos (primera columna) así como su contenido tal y como lo visualizaría un editor hexadecimal (segunda columna). En la tercera columna se muestra la representación decimal o ASCII del contenido binario. Nótese el símbolo separador | tras la primera cadena de longitud variable pero no después de la última.

En la implementación podéis asumir que el tamaño máximo del campo "título" y del campo "editorial" es de 128 caracteres, pero, a la hora de acceder al disco duro solo se escribe (o lee) los caracteres relevantes. Esto es, si el título es *El Quijote* no se escriben 128 caracteres sino 11: 'E', 'l', ' ', 'Q', 'u', 'i', 'j', 'o', 't', 'e' y '|' sin el '\0' de final de cadena.

2.2. Índice

Para mejorar la velocidad de acceso al código vamos a implementar un índice que mantendremos en memoria. Cada registro del índice tendrá una estructura similar a:

```
typedef struct {
    int key;      /* book isbn */
    long int offset; /* book is stored in disk at this position */
    size_t size; /* book record size. This is a redundant
                    field that helps in the implementation */
} indexbook;
```

Las entradas del índice deben estar ordenadas de forma ascendente, con la clave más pequeña al principio del índice y la más grande al final. Esto

permitirá utilizar una búsqueda binaria para encontrar las claves en el índice. Para mantener el índice ordenado tanto a la hora de insertar un registro como de borrarlo se deberá:

- Buscar el registro a borrar o el lugar de inserción mediante una búsqueda binaria.
- Desplazar los registros para cubrir el hueco abierto (borrado) o para hacer espacio para el registro a introducir (inserción) mediante la orden `memmove`.
- Insertar el registro (en el caso de inserción).

El índice debe almacenarse en memoria de una forma que permita el acceso directo y la expansión dinámica del mismo. Una posible implementación se basaría en un array expansible dinámicamente que esté ordenado desde la clave primaria menor al principio del índice a la mayor al final. En el Apéndice A os mostramos algunas líneas de código que os pueden ayudar a la hora de implementar esta sección.

El índice no debe reconstruirse cada vez que se cargue la base de datos. Por el contrario, al finalizar el programa se escribirá el índice en el disco, guardando su contenido en un archivo de índice. Cuando se vuelva a cargar el fichero con los libros, se cargará el archivo de índice correspondiente en memoria quedando el índice exactamente en el mismo estado que tenía cuando fue volcado a disco y quedará listo para ser utilizado para acceder a los registros del fichero de libros.

Puesto que el índice se realiza sobre una clave primaria no pueden existir dos entradas en el índice con la misma clave.

Advertencia: tened cuidado a la hora de manejar estructuras en C. Dado que el tamaño de un `int` y un `size_t` en los ordenadores de los laboratorios es 4 y 8 bytes respectivamente se podría concluir que el tamaño de la estructura `indexbook` sería 12 bytes pero esto no es cierto porque el tamaño de una estructura es un multiplo del tamaño de su miembro más grande (en este caso 8 bytes) y por tanto la estructura `indexbook` ocupa 16 bytes en memoria.

0	39 30 00 00	12345
4	2E 00 00 00 00 00 00 00	46
12	24 00 00 00 00 00 00 00	36
20	3A 30 00 00	12346
24	00 00 00 00 00 00 00 00	0
32	32 00 00 00 00 00 00 00	50
40	3B 30 00 00	12347
44	5A 00 00 00 00 00 00 00	90
52	52 00 00 00 00 00 00 00	38
60	3C 30 00 00	12348
64	88 00 00 00 00 00 00 00	136
72	24 00 00 00 00 00 00 00	36

Tabla 3: Descripción gráfica de la estructura del fichero de índices generado por el test `add_index_text.sh`. Para cada campo se muestra su offset en el fichero de índice (primera columna) así como su contenido tal y como lo visualizaría un editor hexadecimal (segunda columna). En la tercera columna se muestra la representación decimal del número.

2.3. Listado de registros borrados

En el apartado anterior se describe cómo actualizar el índice que se encuentra almacenado en memoria. Podríamos seguir un sistema similar con el fichero de datos pero mover registros en disco es una operación muy costosa por lo que vamos a seguir una aproximación diferente. Se registrará el tamaño y la posición (offset) del registro de datos borrado en una lista de registros borrados que mantendremos en memoria. Cada entrada en la lista de registros borrados tendrá una estructura similar a:

```
typedef struct {
    size_t register_size; /* size of the deleted register */
    size_t offset; /* Record's offset in file */
    } indexdeletedbook;
```

A diferencia con el comportamiento del índice, el listado de registros borrados puede necesitar de una reconstrucción cada vez que se cargue la base de datos. Como en el caso anterior, al salir del programa se escribirá el listado en el disco, guardando su contenido en un archivo. Cuando se vuelva a

cargar el fichero con el índice en memoria se cargará el listado de registros borrados atendiendo a las indicaciones pasadas en la línea de comandos la cual permite elegir entre las estrategias *best fit*, *first fit* y *worst fit*.

A la hora de guardar el listado de registros borrados os hará falta almacenar la estrategia de búsqueda usada para crearlo. Almacena un entero (en binario) al principio del fichero siguiendo la convención:

```
#define BESTFIT 0
#define WORSTFIT 1
#define FIRSTFIT 2
```

3. Interfaz de usuario

3.1. Descripción

El programa a implementar debe llamarse `library` y debe ser ejecutable desde la línea de comandos. A la hora de ejecutarlo se le pasarán dos parámetros. El primero seleccionará la estrategia de ordenamiento a seguir en el listado de ficheros borrados, los posibles valores de este argumento son: `best_fit`, `first_fit` o `worst_fit`. El segundo argumento contendrá la raíz del nombre usado por los ficheros que necesita crear el programa. De esta forma si el valor del segundo argumento es `test`, el programa creará tres ficheros con los nombres `test.db` (datos), `test.ind` (índice), `test.lst` (listado registros borrados).

Una vez inicializado el programa, este se comunicará con el usuario a través del teclado. El programa debe entender los comandos: *add*, *find*, *del*, *exit*, *printRec*, *printInd*, *printLst*. A continuación se describen los comandos en detalle.

3.2. Implementación

Add

Añade un libro a la base y actualiza el índice.
Ejemplo: `add 12345|978-2-12345680-3|El Quijote|catedra`
Para añadir un libro a la base se debe:

- Realizar una búsqueda binaria en el índice. Sí la clave proporcionada ya existe se debe escribir por pantalla: `Record with BookID=XXXX exists`.
- Mirar en el listado de registros borrados si existe un hueco capaz de albergar el nuevo libro. Tened en cuenta la estrategia de búsqueda seleccionada por el usuario.
- Si existe un hueco reúsalo. Si no se rellena todo el hueco no te olvides de añadir el fragmento resultante a la lista de registros borrados.
- Si no podemos reaprovechar un hueco añadid el registro al final del fichero de datos.
- Actualizar el índice y, si fuera necesario, el registro de ficheros borrados.
- Finalmente imprime por pantalla un mensaje confirmando la inserción del libro: `Record with BookID=12345 has been added to the database`

Find

Dada una clave busca el registro y lo imprime por pantalla.
Ejemplo: `find 12345`

- Realizar una búsqueda binaria en el índice.
- Si la clave se encentra imprimir por pantalla el registro como se muestra a continuación: `12345|978-2-12345680-3|El Quijote|catedra`
- Si la clave no existe imprimid el mensaje de error, `Record with bookId=12345 does not exist`

Del

Borra el registro del fichero de datos y actualiza tanto el índice como la lista de registros borrados.
Ejemplo: `del 12345`.

- Realiza una búsqueda binaria del registro.

- Si la clave existe añade una entrada en el listado de registros borrados y actualiza el índice. Finalmente imprime por pantalla el mensaje: `Record with bookId=12345 has been deleted`
- Si la entrada no existe imprime por pantalla: `Record with bookId=12345 does not exist`

Exit

Finaliza el programa, debes:

- Cerrar el fichero que contiene los registros de los libros
- Guardar el índice y el listado de registros borrados al disco duro.
- Liberar todos los recursos solicitados por el programa antes de terminar la ejecución.

3.2.1. printInd

Imprime el contenido del índice tal y como está ordenado en memoria. Imprime en cada línea un registro siguiendo el formato:

```
Entry #0
    key: #12345
    offset: #46
    size: #36
Entry #1
    key: #12346
    offset: #0
    size: #38
```

3.2.2. printLst

Imprime el contenido del listado de registros borrados tal y como está ordenado en memoria. Imprime en cada línea un registro siguiendo el formato:

```
Entry #0
    offset: #90
```

```
    size: #41
Entry #1
    offset: #46
    size: #36
```

3.2.3. printRec

Imprime el contenido del fichero que contiene los libros siguiendo el orden marcado por el índice e ignorando los registros borrados. Imprime en cada línea un registro siguiendo el formato:

```
12345|978-2-12345680-3|El Quijote|Catedra
12346|978-2-12345681-3|La busca|Catedra
```

4. Tests

En Moodle encontraréis una colección de ficheros de test con extensión `sh`. Estos ficheros lanzan el programa llamado `library` con diferentes opciones e interactúan con él simulando ser un usuario.

Estos ficheros de test se usarán para corregir vuestras prácticas, así que es importante que comprobéis que funcionan. En particular, aseguraros de que las cadenas buscadas sean impresas por pantalla por el programa y que el formato de salida de las consultas de vuestro programa coincida con el que espera el fichero.

Nota: para ejecutar los ficheros en vuestra casa necesitaréis instalar la utilidad "expect".

A continuación se enumeran el nombre de los tests y su función:

cli_tests.sh Comprueba la línea de comandos.

add_data_test.sh Comprueba que la función "add" está implementada. El test sólo añade entradas, nunca las borra. No se comprueba la existencia del índice o de la lista de libros borrados.

add_index_test.sh Similar al test anterior pero comprueba la creación de índice en memoria y su escritura a disco al finalizar el programa.

reload_index.sh Similar al test anterior pero comprueba que en caso de existir el fichero de índices el mismo se lee al lanzar el programa

many_entries.sh Similar al test *add_data_test.sh* pero insserta 30000 books. Sirve para comprobar la calidad de la implementación.

add_delete_test_01.sh Similar al test *add_data_test.sh* pero comprueba que tras borrar un libro este ha desaparecido del fichero de índices.

add_delete_test_02.sh Similar al anterior pero comprueba que tras borrar un libro la lista de de libros borrados se actualiza. La estrategia utilizada es "first fit"

add_delete_test_03.sh Similar al anterior pero la estrategia utilizada es "best fit"

add_delete_test_04.sh Similar al anterior pero la estrategia utilizada es "worst fit"

No se proporciona ningún test que compruebe que tras borrar un libro el espacio liberado en el fichero de datos es reaprovechado por el siguiente comando *add*. Se deja este test como ejercicio a los alumnos. Guardad el mismo en un fichero llamadao *test_use_deleted_books.sh*.

5. Compilación y Ejecución del Programa

Proporcionad un fichero llamado `makefile` con el que se pueda compilar vuestro código usando el comando `make`. Los comandos `make bestfit`, `make worstfit` y `make firstfit` debe lanzar la aplicación con la estrategia de búsqueda correspondiente.

6. Resumen del material a entregar al final de la práctica

Subid a *Moodle* un fichero único en formato zip que contenga:

1. Todos los ficheros de código necesarios para compilar vuestro programa y los ficheros de test disponibles en *Moodle*.
2. Un fichero `makefile` que permita compilar el programa ejecutando el comando `make` y ejecutarlo con el comando `make estrategia`, donde `estrategia` puede tener los valores `best_fit`, `first_fit` o `worst_fit`. Tras ejecutar `make estrategia` se mostrará por pantalla el interfaz de usuario.
3. El fichero de test llamado *test_use_deleted_books.sh* solicitado en la sección de test.
4. Memoria en formato pdf que incluya: (1) descripción del objetivo de esta práctica así como de los algoritmos a implementar; utiliza tus propias palabras, no copies el enunciado de esta practica; (2) explicación de cada algoritmo, ilustrado con un ejemplo.

El fichero a subir a *Moodle* será practica3.zip.

7. Criterios de corrección

Para aprobar es necesario:

- Implementar las operaciones: *add*, *printRec* y *printInd*. No hace falta tener en cuenta el caso en el que se borran registros. No hace falta implementar el listado de libros borrados. Ni la escritura/lectura del índice cuando el programa se lanza/termina.
- Subir a *Moodle* el código que implementa los algoritmos requeridos junto al fichero `makefile` que permite compilar el código y ejecutar el interfaz de usuario. Igualmente deben incluirse los ficheros de test.

- El código subido a *Moodle* debe poder compilarse con el fichero makefile suministrado en los ordenadores de los laboratorios de prácticas.
- Los tests *cli_tests.sh*, *add_data_test.sh* y *many_entries.sh* deben funcionar incluso si se modifica el valor (o el número) de los datos a introducir.
- Los registros con los libros se almacenarán exclusivamente en un fichero de datos.

Para obtener una nota en el rango 5-5.9 es necesario:

- Satisfacer todos los requerimientos del apartado anterior.
- Implementar la función $find$.
- Implementar la escritura/lectura del fichero conteniendo los índices.
- El programa detectará si se intenta introducir una clave repetida.
- El test *reload_index.sh* debe funcionar incluso si se modifica el valor (o el número) de los datos a introducir.
- Las búsquedas se realizan utilizando un algoritmo de búsquedas en arboles binarios.

Para obtener una nota en el rango 6-7.9 es necesario:

- Satisfacer todos los requerimientos del apartado anterior.
- Implementar la función `del` de forma que: borre el libro seleccionado del fichero de datos, actualice el índice y actualice la lista de libros borrados. No se solicita en este apartado que la orden *add* sea capaz de reaprovechar el espacio dejado en el fichero de datos tras borrar algún libro.
- Los tests *add_delete_test_*01.*sh*, *add_delete_test_*02.*sh*, *add_delete_test_*03.*sh* y *add_delete_test_*04.*sh* deben funcionar incluso si se modifica el valor (o el número) de los datos a introducir.
- Presentar una memoria describiendo el objetivo de la práctica y los algoritmos a implementar. Se espera que la memoria esté correctamente redactada y demuestre vuestra comprensión de la práctica

- Crear un código estructurado y comentado en el que se identifique claramente dónde se implementa cada paso de cada algoritmo. Cada función debe tener el nombre del autor, una descripción de la tarea que implementa, y de los parámetros de entrada y de salida.
- El programa se comporta correctamente para bases con miles de libros. Correctamente significa que el programa es capaz de ejecutar test similares a *many_entries.sh* en un tiempo razonable. Tiempo razonable se define como el tiempo que tarde la mejor implementación de esta práctica multiplicada por 20.
- Manejar los errores correctamente. Por ejemplo si se proporciona un comando o una clave inválidos, el programa debe gestionar el problema adecuadamente. En ningún caso el programa debe abortar o dar un coredump.
- La utilidad `valgrind` no debe detectar ningún "memory leak" atribuible a las líneas de código que habéis implementado.

Para obtener una nota en el rango 8-8.9 es necesario:

- Satisfacer todos los requerimientos del apartado anterior.
- Implementar toda la funcionalidad requerida en la práctica excepto los tests. para `addTableEntry` (la función sí debe estar implementada).

Para obtener una nota en el rango 9-10 es necesario:

- Satisfacer todos los requerimientos del apartado anterior.
- Reusar el espacio dejado por los registros borrados a la hora de añadir nuevos registros.
- Implementar un test similar a los que os hemos proporcionado que compruebe esta funcionalidad.

NOTA: Cada día (o fracción) de retraso en la entrega de la práctica se penalizará con un punto.

NOTA: Si no se puede compilar en la imagen de la partición linux de los ordenadores de los laboratorios con el comando `make` el programa solicitado en la práctica la calificación de la práctica será cero.

NOTA: El código a corregir será el subido a *Moodle*.

A. Ejemplo de contenedor expansible.

Ejemplo de contenedor expansible. En este ejemplo se almacenan números enteros, vosotros tendréis que almacenar estructuras de tipo indexbook.

```c
typedef struct {
  int *array;
  size_t used;
  size_t size;
} Array;

void initArray(Array *a, size_t initialSize) {
  /* create initial empty array of size initialSize*/
  a->array = malloc(initialSize * sizeof(int));
  a->used = 0;
  a->size = initialSize;
}

void insertArray(Array *a, int element) {
  /* insert item "element" in array
     a->used is the number of used entries,
     a->size is the number of entries */

  if (a->used == a->size) {
    a->size *= 2;
    a->array = realloc(a->array, a->size * sizeof(int));
  }
  a->array[a->used++] = element;
}

void freeArray(Array *a) {
  /* free memory allocated for array */
  free(a->array);
  a->array = NULL;
  a->used = a->size = 0;
}

/* Example of use */
```

```c
Array a;
int i;

initArray(&a, 5);  // initially 5 elements
for (i = 0; i < 100; i++)
  insertArray(&a, i);  // automatically resizes as necessary
printf("%d\n", a.array[i]);  // print i-th element
printf("%d\n", a.used);  // print number of elements
freeArray(&a); // free array
```

(Nota: este código está sacada de `https://stackoverflow.com/questions/3536153/c-dynamically-growing-array`)