Universidad Autónoma de Madrid

Departamento de Informática

Estructura de Datos
Práctica - 3: Tablas e Índices

Roberto Marabini Ruiz

Índice

1. Objetivo

En esta práctica aprenderemos cómo se manejan las tablas y los índices en una base de datos simplificada. En la implementación que os solicitamos, tanto los índices como los datos se almacenan en sendos ficheros a los que se accede constantemente para insertar y buscar información. La información no se almacena en memoria sino que se lee cuando se necesita y se escribe tras una nueva inserción o una modificación. De esta forma es posible compartir los datos entre varios programas independientes.

2. Estructura de los ficheros de datos e índices

2.1. Estructura de los ficheros de datos

En las bases de datos relacionales los datos se guardan en tablas. Cada tabla consta de (1) una cabecera que, en nuestra versión simplificada, contiene únicamente un puntero que apunta a una lista de registros borrados y (2) una colección de registros de longitud variable. En general, el número y tipo de campos que hay en cada registro es muy variado pero, en nuestra práctica, asumiremos que únicamente se va a almacenar en cada registro una clave primaria (una cadena alfanumérica de 4 caracteres que NO acaba en '\0') y una cadena de caracteres de longitud variable (que tampoco acaba en '\0'). De esta forma, cada registro constará de tres campos: una cadena de caracteres de longitud fija, un número entero y una cadena de caracteres de longitud variable. El primer campo se usará para almacenar la clave primaria, el segundo campo almacenará el tamaño de la cadena de longitud variable que se desea guardar y, finalmente, almacenaremos los valores de la cadena. La tabla 1 describe gráficamente cómo está organizado el fichero de datos mientras que la tabla 2 muestra el contenido de estos ficheros byte a byte.

0	-1		
4	BAR1	22	Zalacain el aventurero
34	GAR1	21	Poema del cante jondo
63	BAR2	8	La busca

Tabla 1: Descripción gráfica de la estructura del fichero de datos. En la cabecera tenemos el offset al primer registro borrado o -1 si no existe ningún registro borrado. A continuación se muestran los registros. La primera columna muestra el offset de cada registro.

```
00000 | FF FF FF FF 42 41 52 31 16 00 00 00 5A 61 6C 61 | ....BAR1....Zala
00016 | 63 61 69 6E 20 65 6C 20 61 76 65 6E 74 75 72 65 | cain el aventure
00032 | 72 6F 47 41 52 31 15 00 00 00 50 6F 65 6D 61 20 | roGAR1....Poema
00048 | 64 65 6C 20 63 61 6E 74 65 20 6A 6F 6E 64 6F 42 | del cante jondoB
00064 | 41 52 23 08 00 00 00 4C 61 20 62 75 73 63 61 00 | AR2....La busca.
```

Tabla 2: El fichero de datos descrito en la tabla 1 abierto con un editor binario. La zona de la izquierda muestra el offset en el fichero (en hexadecimal), la segunda el contenido de cada byte como un número hexadecimal y la tercera el contenido usando caracteres ASCII. Aquéllos bytes que no codifican caracteres visibles se han mostrado usando un punto.

Para terminar está subsección quisiéramos hacer hincapié una vez más en que estamos simplificando el problema para facilitar la implementación. En un caso más realista habría que tener en cuenta que los ficheros de datos suelen guardar los registros ordenados por su clave primaria usando un sistema de punteros, las claves primarias pueden estar formadas por varios atributos, el número y tipo de campos es muy variado, la cabecera suele guardar una descripción de los campos contenidos, etc.

2.2. Estructura del fichero de índices

En la práctica usaremos un árbol binario de búsqueda para crear un índice. La estructura del fichero de índices consiste en una cabecera seguida de los registros que contienen los nodos de un árbol binario de búsqueda.

La cabecera almacena la información siguiente:

root (integer), puntero al registro raíz. El primer nodo es el nodo 0, el segundo el nodo 1 y así sucesivamente. *root* almacena el nodo donde se encuentra la raíz del árbol.

deleted (integer), puntero al primer nodo de la cadena de nodos borrados.

A continuación viene una colección de registros de tamaño fijo donde se almacena cada nodo del árbol. Su estructura es la siguiente:

- Espacio para almacenar la clave primaria (4 bytes).
- Espacio para almacenar tres punteros a los nodos hijo-izquierdo, hijo-derecho y padre (3×integer). El puntero al nodo padre no es imprescindible pero simplifica los cálculos.
- Puntero a los datos (integer). Esto es, posición en el fichero de datos donde comienza el registro que se indexa.

En los registros borrados se almacenará un puntero (integer). Este puntero apuntará al siguiente registro de la cadena de registros borrados. Nota: no es extraño almacenar también el tamaño del registro borrado pero por simplicidad no lo lo haremos en nuestra implementación.

La figura 1 muestra un árbol binario de búsqueda que ordena los datos introducidos en la tabla 1 alfabéticamente (sobre la clave primaria). Las tablas 3 y 4 ilustran la estructura de los ficheros de índices para este ejemplo.

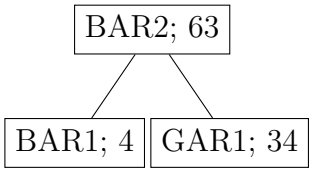

Figura 1: Árbol binario de búsqueda usado como índice. Cada nodo muestra la clave primaria de los registros indexados así como el offset del registro en el fichero de datos

	2	-1			
0	BAR1	-1	-1	2	4
1	GAR1	-1	-1	2	34
2	BAR2	0	1	-1	63

Tabla 3: Descripción gráfica de un fichero de índice realizado sobre la clave primaria del fichero de datos mostrado en la subsección anterior. La primera fila muestra el puntero (offset) al nodo raíz seguido del puntero a la lista de registros borrados. Para el resto de las filas la convención seguida es: la primera columna muestra la numeración de los nodos, la segunda, la clave primaria de la tabla indexada, y la tercera, cuarta y quinta son punteros a los nodos hijo-izquierdo e hijo-derecho así como padre, la última columna es el offset del registro indexado en el fichero de datos.

```
00000 | 02 00 00 00 FF FF FF FF 42 41 52 31 FF FF FF FF | ........BAR1....
00016 | FF FF FF FF 02 00 00 00 04 00 00 00 47 41 52 31 | ............GAR1
00032 | FF FF FF FF FF FF FF FF 02 00 00 00 22 00 00 00 | ............``...
00048 | 42 41 52 32 00 00 00 00 01 00 00 00 FF FF FF FF | BAR2............
00064 | 3F 00 00 00                                     | ?...
```

Tabla 4: El fichero de índice del ejemplo anterior abierto con un editor binario. La primera columna representa el offset en el fichero, la segunda el contenido de cada byte como un número hexadecimal y la tercera muestra el contenido usando caracteres ASCII. Aquellos bytes que no codifican caracteres visibles se muestran usando un punto.

3. Operaciones a implementar

Las operaciones que se realizarán serán las de inserción y búsqueda de registros. Así como la posibilidad de imprimir los datos contenidos en el fichero de registros por pantalla. Los prototipos de las funciones a implementar serán:

```c
#include <stdio.h>
#include <stdbool.h>

#define NO_DELETED_REGISTERS -1

#define INDEX_HEADER_SIZE 8
```

```c
#define DATA_HEADER_SIZE  4
/* primary key size */
#define PK_SIZE 4

/* Our table is going to contain a string (title) and
   an alphanumeric primary key (book_id)
*/
typedef struct book {
    char book_id[PK_SIZE ]; /* primary key */
    size_t title_len; /* title length */
    char *title; /* string to be saved in the database */
} Book;

/* Note that in general a struct cannot be used to
   handle tables in databases since the table structure
   is unknown at compilation time.
*/

typedef struct node {
    char book_id[PK_SIZE];
    int left, right, parent, offset;
}Node;


/* Function prototypes.
   see function descriptions in the following sections

   All function return true if success or false if failed
   except findKey which return true if register found
   and false otherwise.
   */

bool createTable(const char * tableName);
bool createIndex(const char * indexName);
bool findKey(const char * book_id, const char * indexName,
             int * nodeIDOrDataOffset);
bool addTableEntry(Book * book, const char * tableName,
                   const char * indexName);
bool addIndexEntry(char * book_id, int bookOffset,
                   const char * indexName);

void printTree(size_t level, const char * indexName);
```

Se asume que los ficheros conteniendo datos tienen extensión `dat` y los ficheros conteniendo índices tienen el mismo nombre y la extensión `idx`.

NOTA IMPORTANTE: aunque no se solicita que implementéis la función de borrado vuestro código debe funcionar aunque los ficheros de datos contengan registros que han sido borrados.

3.1. Descripción de las funciones

3.1.1. Funciones `createTable` y `createIndex`

`createTable` intentará abrir el fichero cuyo nombre esté almacenado en la variable `tableName`. Si falla (esto es, el fichero no existe) la rutina debe crear el fichero e inicializar la cabecera con el número entero -1, lo que significa que el listado de registros borrados está vacío (recordad que hay que escribir siempre en binario).

Tras crear (si fuera necesario) el fichero con los datos, se llamará a la función `createIndex` quien creará (si es necesario) el fichero de índices. El nombre de este fichero será el almacenado en la variable `tableName` reemplazando la extensión `dat` por `idx`. En caso de que se cree el fichero debe inicializarse el cabecero con dos punteros iguales a -1. Estos punteros apuntan al nodo raíz y al primer nodo borrado.

NOTA: puesto que estamos creando un sistema que debe funcionar como una base de datos, nuestras funciones deben abrir el fichero al que necesiten acceder, realizar la operación sobre el mismo y proceder a cerrarlo. En caso contrario, una segunda instancia de nuestro programa, no podría acceder a los datos modificados a medida que se producen. En esta práctica ignoraremos los problemas derivados del acceso concurrente, esto es, dos instancias del programa modificando exactamente a la vez el mismo fichero. Lo que si debe satisfacerse es un acceso secuencial. Esto es, si se ejecutan dos instancias del mismo programa llamadas `i1` e `i2` y una vez arrancadas ambas instancias realizamos una operación usando `i1` si esperamos a que esta operación concluya, desde `i2` debemos ver el resultado de la operación efectuada en `i1`.

3.1.2. Función `printTree`

La función `printTree` debe mostrar por pantalla el árbol binario de búsqueda almacenado en el fichero de índice siguiendo el modelo mostrado en la figura 3 donde se imprime el árbol mostrado en la figura 2.

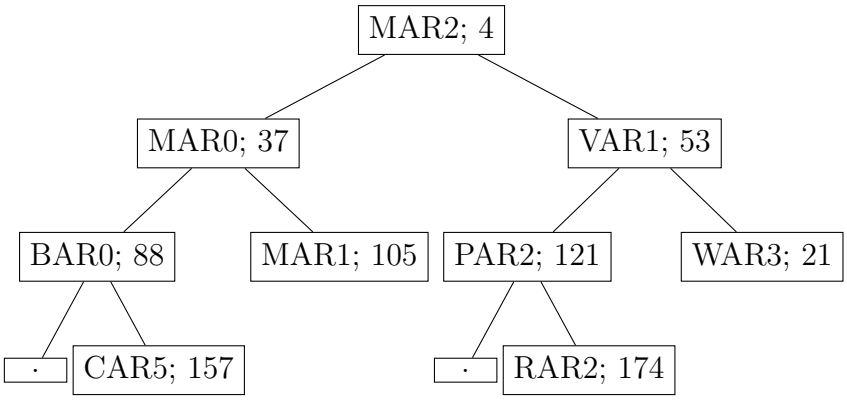

Figura 2: Árbol binario de búsqueda usado como índice. Cada nodo muestra la clave primaria de los registros indexados así como el offset del registro en el fichero de datos

```
MAR2 (0): 4
 l MAR0 (2): 37
     l BAR0 (5): 88
        r CAR5 (9): 157
     r MAR1 (6): 105
 r VAR1 (3): 53
     l PAR2 (7): 121
         r RAR2 (10): 174
      r WAR3 (1): 21
```

Figura 3: Ejemplo de salida de la función `printTree` para el árbol descrito por la figura 2. Para cada nodo se muestra: (1) carácter **l** o **r** para diferenciar la rama izquierda de la rama derecha, (2) clave, (3) identificador de cada nodo en el archivo de índices y, (4) offset a los datos en el fichero de datos. La indentación está relacionada con la profundidad a la que se encuentra el nodo.

El parámetro `level` controla hasta que profundidad se va a imprimir el árbol binario. Si `level==0` entonces se mostrará sólo el nodo raíz.

Recordad que NO debéis almacenar la información en memoria sino que hay que acceder al disco para leerla o modificarla.

3.1.3. Función `findKey`

Dada una clave `book_id`, `findKey` devolverá `true` si se ha encontrado la clave en el índice y `false` en caso contrario. Adicionalmente, almacenará en el parámetro `nodeIDOrDataOffset` el identificador del último nodo visitado si no se ha encontrado la clave o, el offset del registro de datos si se ha encontrado la clave. Nótese que en el primer caso, cuando la clave no ha sido encontrada, el nodo devuelto sería el punto de inserción de la clave buscada si quisiéramos insertarla en el árbol.

3.1.4. Funciones `addIndexEntry` y `addTableEntry`

`addTableEntry` recibe como argumentos una estructura de tipo `Book` y el nombre del fichero de datos. Debe añadir la información contenida en la estructura como se describe a continuación y debe concluir llamando a `addTableIndex` quien actualizará el fichero índice.

- Comprueba, usando `findKey`, si la clave existe. En caso afirmativo devuelve `false` y el programa debe mostrar un mensaje de error.
- Comprueba si hay algún registro borrado en el fichero de datos accediendo a la cabecera del mismo y comprobando si el valor almacenado en la misma es -1
- Si no hay ningún registro borrado, la nueva estructura se añadirá al final del fichero usando el formato descrito. Esto es, guardando una cadena de 4 caracteres con el identificador, un número entero con el tamaño de la cadena variable a almacenar y finalmente la cadena. (En esta práctica asumiremos que la cadena a guardar está formada por caracteres ASCII y que por lo tanto cada carácter ocupa un byte y que NO se añade un '\0' al final de la cadena guardada). Hay que anotar el offset del registro añadido puesto que nos hará falta a la hora de actualizar el índice.
- Si existe algún registro borrado, se mirará en la cadena de registros borrados si alguno es suficientemente grande como para almacenar el nuevo registro. En caso afirmativo, se usará el primer espacio disponible y se actualizará tanto la cadena de registros borrados como el tamaño del registro borrado utilizado (si este no se ha ocupado en su totalidad).

- A continuación, se llamará a `addTableIndex` quien actualizará el índice añadiendo un nuevo nodo. Al final de esta memoria hay un apéndice describiendo cómo se añaden y se quitan nodos en un árbol binario. Al igual que en el caso anteriormente descrito, se reusarán los registros borrados en caso de que sea posible. Nótese que en el fichero índice todos los registros tienen el mismo tamaño.
- Una diferencia importante entre el fichero de datos y el de índices es que en el primero el puntero a la lista de nodos borrados guarda offsets (posiciones en el fichero) mientras que en el segundo guarda el número de registro (el cual se puede convertir a offset mediante la fórmula numero_registro * INDEX_REGISTER_SIZE + INDEX_HEADER_SIZE)

IMPORTANTE, a la hora de transcribir a disco una variable de tipo `Book` no basta con el comando

```
fwrite( &book, sizeof( Book ), 1, fp );
```

puesto que solo guarda en el fichero la parte no dinámica de la estructura. La forma correcta de almacenar los datos es:

```
/* defined in include file */
#define PK_SIZE 4

int lenTitle = strlen(book.title);
fwrite( book.book_id, PRIMARYKEYLENGTH, 1, fp );
fwrite( &lenTitle, sizeof(int), 1, fp );
fwrite( book.title, lenTitle, 1, fp );
```

4. Interfaz de Usuario

El usuario podrá comunicarse con las rutinas desarrolladas a través de un menú similar al implementado en la práctica anterior y donde se han de implementar los comandos:

use: El sistema preguntará por el nombre de la tabla a utilizar y llamará a la función `createTable`.

insert: El sistema preguntará por la clave y el título a almacenar y llamará a `addTableEntry`. Se debe mostrar un mensaje de error si no se ha seleccionado una tabla antes usando `use`. Recuerda: (1) `addTableEntry` debe llamar a `addIndexEntry` y (2) tras añadir un libro al sistema este debe estar accesible por cualquier otra instancia del mismo programa, por lo tanto, no se pueden almacenar los datos en memoria si no que deben almacenarse en disco.

print: Se muestra el árbol binario de búsqueda por pantalla. Se debe mostrar un mensage de error si no se ha seleccionado una tabla antes usando `use`.

exit: Se deben cerrar el fichero de datos y de índice y salir del programa.

5. Tests

En *Moodle* tenéis accesible el fichero `test.zip` que contiene una colección de tests unitarios y funcionales (no necesariamente completa) que os puede ser útil para crear vuestra implementación de la práctica. Para compilar los tests se suministra un fichero `makefile`, tras ejecutarlo se produce un fichero binario llamado `tester` que al ser ejecutado llama a los distintos tests. Una condición necesaria, pero no suficiente, para que demos vuestro código por bueno es que satisfaga el test correspondiente.

El fichero `makefile` suministrado asume que todas las funciones implementadas se encuentran en un único fichero llamado `utils.c`. En general es más práctico distribuir las funciones a implementar en varios ficheros. Si os decidís a usar varios ficheros modificad el `makefile` para que el código producido sea compilable tecleando `make`.

No os suministramos tests para la función `addTableEntry`. Dejamos como ejercicio la creación de un conjunto no trivial de tests para probar esta función.

NOTA IMPORTANTE: A la hora de realizar la implementación, se recomienda seguir una estrategia TDD (test-driven development) tratando de satisfacer uno a uno y siguiendo el orden establecido cada uno de los tests. Esto es: (1) compilad los tests (`make`), (2) ejecutadlos (`./tester`) e (3) implementad el código necesario para corregir los fallos.

6. Compilación y Ejecución del Programa

Proporcionad un fichero llamado `makefile.cod` para que podamos compilar vuestro código usando el comando `make -f makefile.cod`. El comando `make -f makefile.cod run` debe lanzar la aplicación de forma que se muestre el interfaz de usuario por pantalla.

7. Material a entregar tras la segunda semana de prácticas

Tras la segunda semana de la practica deberéis entregar un código compilable que satisfaga los tests: `checkCreateIndex(indexName)`, `checkCreateTable(tableName)` y `checkPrint(indexName)`.
Subid a *Moodle* un fichero único en formato zip que contenga:

1. Todos los ficheros de código necesarios para compilar el programa de test con la funcionalidad requerida al principio de esta sección.
2. Un fichero `makefile` que permita crear el programa `tester`.

8. Resumen del material a entregar al final de la practica

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 prueba disponibles en *Moodle*.
2. Un fichero `makefile.cod` que permita compilar el programa ejecutando el comando `make -f makefile.cod` y ejecutarlo con el comando `make -f makefile.cod run`. Tras ejecutar `make -f makefile.cod run` se mostrará por pantalla el interfaz de usuario.
3. 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.

9. Criterios de corrección

Para aprobar es necesario:

- Subir a *Moodle* el código que implementa los algoritmos requeridos en el punto siguiente junto a los ficheros `makefile` y `makefile.cod` que permite compilar el código y ejecutar el interfaz de usuario y cada uno de los tests propuestos.
- Que la implementación proporcione el resultado correcto para las operaciones: *use* y *print*.
- Que los test relacionados con las opciones *use* y *print* se satisfagan.
- Que tras cada operación **NO** se guarde el árbol binario de búsqueda o los datos en memoria sino que se haga en el disco duro. Igualmente los datos deben leerse del disco duro a medida que haga falta y nuca se

deben almacenar en memoria nada más que el registro al que se está accediendo en ese momento.

Para obtener una nota en el rango 5-6 es necesario:

- Satisfacer todos los requerimientos del apartado anterior.
- Implementar la función findKey. La implementación debe ser valida para cualquier instancia de la base de datos y satisfacer el test `checkFindKey`
- El programa detectará si se intenta introducir una clave repetida.

Para obtener una nota en el rango 6-7 es necesario:

- Satisfacer todos los requerimientos del apartado anterior.
- Implementar la función `addIndexEntry`. La implementación debe ser valida para cualquier instancia de la base de datos y satisfacer el test `checkAddIndexEntry`.
- 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 y una descripción.
- El programa se comporta correctamente para árboles con miles de nodos. Correctamente significa que el programa es capaz de crear el árbol 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, en caso de que algún fichero de entrada no tenga una sintaxis correcta, el programa debe dar un mensaje de error comprensible y descriptivo. Igualmente si se proporciona un comando o una clave invalidos, 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 7-9 es necesario:

- Satisfacer todos los requerimientos del apartado anterior.
- Implementar toda la funcionalidad requerida en la practica 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.
- Implementar una colección de tests no triviales para la función `addTableEntry`.

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

NOTA: La ausencia de entrega intermedia (o su entrega tardía) 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 de test y con el comando `make -f makefile.cod` el programa solicitado en la práctica la calificación de la práctica será cero.

NOTA: La falta de uso de git como sistema de control de versiones se penalizará con -0.5 puntos.

NOTA: Si se ha utilizado un repositorio público para guardar el código la calificación de la práctica será cero.

NOTA: El código a corregir será el subido a *Moodle* en ningún caso se usará el existente en un repositorio.

A. Árboles binarios de búsqueda

(Nota: esta descripción esta sacada de `https://jesgargardon.com/blog/operaciones-con-arboles-binarios-de-busqueda/`)

Los arboles binarios de búsqueda se componen de nodos que pueden tener un máximo de dos hijos. Y cumplen las condiciones siguientes:

- Si el nodo tiene un hijo izquierdo, este tiene que ser menor que él.
- Si el nodo tiene un hijo derecho, este tiene que ser mayor que él.

A.1. Búsqueda

La operación `búsqueda` comienza comprobando el nodo raíz y, si este es el nodo buscado, la operación acaba. Si no lo es, nos moveríamos a su hijo izquierdo o al derecho, dependiendo de si el dato que buscamos es menor o mayor del que contiene el nodo padre. Y así proseguiríamos hasta encontrar el dato o terminar de recorrer el árbol sin encontrarlo.

A.2. Inserción

Para insertar un nodo en un árbol binario de búsqueda, recorremos éste de forma similar a como lo hacíamos en el proceso de búsqueda y, cuando lleguemos a un nodo sin hijos, procederemos a insertar un nuevo nodo bien a la izquierda o a la derecha dependiendo de si la nueva clave a almacenar es mayor o menor que la clave del nodo padre. Nota, puesto que nuestra clave es una clave primaria no se admite introducir la misma clave dos veces.

A.3. Recorrido

El recorrido de un árbol binario es una operación que consiste en visitar todos sus nodos, de tal manera que cada nodo se visite exactamente una sola vez. Existen varias posibilidades, para implementar la función `printTree` se recomienda usar el recorrido `preorden`. En este recorrido primero se visita la raíz; segundo se recorre el subárbol izquierdo y por último se va al subárbol derecho.

```
preorden(nodo)
 si nodo == nulo entonces retorna
 imprime nodo.valor
 preorden(nodo.izquierda)
 preorden(nodo.derecha)
```