Universidad Autónoma de Madrid

Departamento de Informática

Data Structures
Assignment 3: Tables and Indices

Roberto Marabini Ruiz

Contents

1 Goal

In this lab we will learn how tables and indexes are handled in a simplified database. In the requested implementation, both the indexes and the data are stored in separate files that are constantly accessed to insert and search for information. The information is not stored in memory but is read when needed and written after a new insertion or modification. In this way it is possible to share the data between several independent programs.

2 Structure of data and index files

2.1 Structure of data files

In relational databases the data is stored in tables. Each table consists of (1) a header which, in our simplified version, contains only a "pointer" (really, a file offset) pointing to a list of deleted records and (2) a collection of variable length records. In general, the number and type of fields in each record can vary a lot, but in this assignment we will assume that only one primary key is going to be stored in each record (a 4-character alphanumeric string that DOES NOT end in '\0 ') and a variable-length character string (which also does not end in '\0 '). In this way, each record will consist of three fields: a fixed-length character string, an integer, and a variable-length character string. The first field will be used to store the primary key, the second field will store the size of the variable length string to be saved, and the third field the string itself. Table 1 graphically displays how the data file is organized, and Table 2 shows the contents of these files byte by byte in hexadecimal notation.[1]

[1] You can use `xxd` orhexdump (or its more convenient form `hd`) to view binary files in a Linux terminal. A number of binary/hexadecimal editors also exist, see https://www.tecmint.com/best-hex-editors-for-linux/, as well as extensions for popular text editors, e.g. for Visual Studio Code.

0	-1		
4	BAR1	22	Zalacain el Aventurero
34	GAR1	21	Poema del Cante Jondo
63	BAR2	8	La busca

Table 1: Graphic description of the data file structure. The header contains the offset to the first deleted record or -1 if there is no deleted record. Next come the records, which are shown one per line. The first column shows the offset of each record in the data file.

```
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 .... Poem
00048 64 65 6C 20 63 61 6E 74 65 20 6A 6F 6E 64 6F 42 del cante jondoB
00064 41 52 23 20 08 00 00 4C 61 20 62 75 73 63 61 00 AR2 .... You are looking for it.
```

Table 2: The data file described in Table 1 opened with a binary editor. The left area shows the position (offset) in the file in hexadecimal, the second the content of each byte as a hexadecimal number and the third the content using ASCII characters. Those bytes that do not encode visible characters are shown with dots.

In closing this subsection we would like to emphasize once again that we are simplifying the problem to facilitate implementation. In a more realistic case, it should be taken into account that the records are usually stored in the data files ordered by their primary key using a pointer system, the primary keys can be made up of several attributes, the number and type of fields is very varied, the header usually stores a description of the table fields, etc.

2.2 Structure of index files

We will use a binary search tree to create an index. The structure of the index file consists of a header followed by the records that contain the nodes of a binary search tree.

The header stores the following information:

root (integer) pointer to root record. The first node is node 0, the second node is node 1, and so on. *root* stores the node where the root of the tree is located.

deleted (integer) pointer to the first node in the list of deleted nodes.

This is followed by a collection of fixed-size records storing the tree nodes. The structure of each record is as follows:

- Primary key (4 bytes).
- Three pointers to the left-child, right-child and parent nodes (4 bytes each). The pointer to the parent node is not essential but it simplifies the calculations.
- Pointer to record. That is, position in the *data file* where the record to be indexed begins.

A pointer (integer) will be stored in deleted records. This pointer will point to the next record in the string of deleted records. Note: it is common to also store the size of the deleted record, but for simplicity we will not do it in our implementation.

Figure 1 shows a binary search tree that sorts the data entered in Table 1 alphabetically by primary key. Tables 3 and 4 illustrate the structure of the index file for this example.

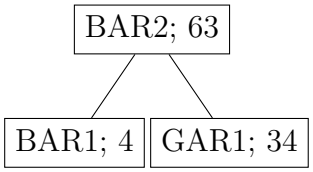

Figure 1: Binary search tree used as an index. Each node shows the primary key of the indexed record and the offset of the record in the data file

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

Table 3: Graphical description of an index file for the primary key of the data file shown in the previous subsection. The first row shows the pointer (offset) to the root node followed by the pointer to the list of deleted records. The following rows are the nodes. For each node, the first column shows the numbering of the nodes; the second one contains the primary key of the indexed record; the third, fourth and fifth columns are pointers to the left and right child nodes and the parent node, respectively; and the last column is the offset of the indexed record in the data file.

00000	00 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	?...

Table 4: The index file from the previous example opened with a binary editor. The first column represents the offset in the file, the second the content of each byte as a hexadecimal number, and the third shows the content using ASCII characters. As before, bytes that do not encode visible characters are displayed with dots.

3 Required functions

Your main task in this assignment is to implement functionality to insert and search for records, as well as printing the data from the data and index files. Specifically, you must implement the functions with the following prototypes:

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

#define NO_DELETED_REGISTERS -1

#define INDEX_HEADER_SIZE 8
#define DATA_HEADER_SIZE  4
```

```c
/* 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 functions return true if success or false if failed, except
       findKey, which returns true if the register is 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);
```

NOTE: It is assumed that the files containing data have the extension `dat` and the files containing the indexes have the same name and the extension `idx`.

IMPORTANT NOTE: though you are not required to implement the delete function, your code should work even if the data files contain records that have been deleted.

3.1 Function descriptions

3.1.1 Functions `createTable` and `createIndex`

The `createTable` function will attempt to open the file whose name is stored in the variable `tableName`. If it fails (that is, the file does not exist) the routine must create the file and initialize the header with the integer number -1, which means that the list of deleted records is empty (remember that you must always write in binary).

After creating (if needed) the data file, the `createIndex` function must be called to create (if necessary) the index file. The name of this file will be the one stored in the variable `tableName` replacing the extension `dat` with `idx`. When the index file is created, the header is initialized with two pointers equal to -1. These pointers point to the root node and the first deleted node (neither of which exist at the time of creation of the index file).

NOTE: since we are creating a system that must function as a database, our functions must open the file they need to access, perform the operation on it and proceed to close it. Otherwise, a second instance of our program would not be able to access the modified data as it occurs. In this assignment we will ignore the problems derived from concurrent access, that is, two instances of the program modifying exactly the same file at the same time. But proper sequential access is required. That is, if two instances of the same program called `i1` and `i2` are run, once both instances are started, if we perform an operation using `i1` and wait for this operation to finish, then we should be able to see from `i2` the result of the operation performed by `i1`.

3.1.2 Function `printTree`

The `printTree` function should display on the screen the binary search tree stored in the index file following the model shown in Figure 3 to print the tree of Figure 2.

The `level` parameter controls how deep the binary tree is to be printed. If `level == 0` then only the root node will be shown.

Remember that you should NOT store the information in memory but you have to access the disk to read or modify it.

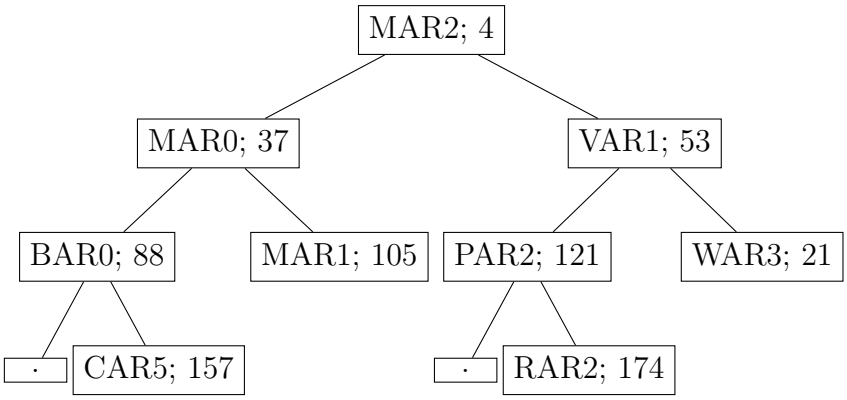

Figure 2: Binary search tree used as an index. Each node shows the primary key of the indexed records as well as the offset of the record in the data file

```
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
```

Figure 3: Example output of the `printTree` function for the tree described by figure 2. For each node, it shows: (1) the character **l** or **r** to differentiate the left branch from the right branch; (2) key; (3) identifier of each node in the index file; and (4) offset to the data in the data file. Indentation is proportional to the node's depht in the tree.

3.1.3 Function `findKey`

Given a key `book_id`, `findKey` will return `true` if the key is found in the index and `false` otherwise. Additionally, it will store in the `nodeIDOrDataOffset` parameter the identifier of the last node visited if the key was not found, or

the offset of the data record if the key was found. Note that in the first case, when the key is not found, the returned node would be the insertion point of the searched key if we wanted to insert it into the tree.

3.1.4 Functions `addIndexEntry` and `addTableEntry`

Function `addTableEntry` receives as arguments a structure of type `Book` and the name of the data file. You must add the information contained in the structure as described below and must conclude by calling `addTableIndex`, which will update the index file.

The function must do the following:

- Check, using `findKey`, if the key exists. If so, it returns `false` and the program should display an error message.
- Check if there are any deleted records in the data file by accessing its header and checking if the value stored in it is -1
- If there is no deleted record, the new structure will be added to the end of the file using the format described above. That is, saving a 4-character string with the identifier, an integer with the size of the variable string to store, and finally the string. (In this assignment we assume that the string to be saved is made up of ASCII characters and that therefore each character occupies one byte, and that a `'\0'` is NOT added to the end of the saved string). Save the offset of the added record, as you will need it when updating the index.
- If there are any deleted records, it will look in the list of deleted records looking for one which is large enough to store the new record. If so, it will use the first available space, and update both the list of deleted records and the size of the deleted record used (if it has not been fully occupied).
- Next, it calls `addTableIndex` to update the index by adding a new node. At the end of this report there is an appendix describing some operations on binary search trees. As in the case described above, deleted records will be reused if possible. Note that in the index file all the records have the same size.

- An important difference between the data file and the index file is that in the former the pointer to the list of deleted nodes saves offsets (positions in the file) while in the latter it saves the record number (which can be convert to offset using the formula number _registration * INDEX _REGISTER _SIZE + INDEX _HEADER _SIZE)

IMPORTANT, when transcribing a variable of type `Book` to disk, this command is not enough:

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

since it only saves the non-dynamic part of the structure in the file. The correct way to store the data is by writing it one field at a time:

```
/* 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 User Interface

The user will be able to communicate with the routines developed through a menu similar to the one implemented in Assignment 2. You must implement the following commands:

use: The system asks for the name of the table to use and call the `createTable` function.

insert: The system asks for the key and title to store and call `addTableEntry`. It should display an error message if a table has not been selected before using `use`. Remember: (1) `addTableEntry` must call `addIndexEntry`, and (2) after adding a book to the system, it must be accessible by any other instance of the same program. So the data should not just be stored in memory, but on disk.

print: Shows the binary search tree on screen. Displays an error message if no table was selected in advance with `use`.

exit: Closes the data and index files, and exits the program.

5 Tests

The file `test.zip`, available in *Moodle*, contains a collection of unit and functional tests (not necessarily complete) that can be useful while developing your code. We also provide a `makefile` that creates a program `tester` to run the various tests. A necessary, but not sufficient, condition for us to consider your code as good is that it satisfies the corresponding tests.

The supplied `makefile` file assumes that all implemented functions are in a single file called `utils.c`. In general, it is more practical to distribute the functions to be implemented in several files. If you decide to use several files, modify the `makefile` so that the produced code is compilable by typing `make`.

We do not supply you with tests for the `addTableEntry` function. We leave as an exercise the creation of a non-trivial set of tests to test this function.

IMPORTANT NOTE: We recommend using a TDD (test-driven development) strategy to develop your implementation. According to this strategy, you should try to satisfy the tests one by one, following the given order. That is: (1) compile the test program (`make`), (2) run it (`./ tester`) and (3) implement the necessary code to correct the bugs.

6 Compiling and Running the Program

Include a file named `makefile.cod` so that we can compile your code using the command `make -f makefile.cod`. The command `make -f makefile.cod run` should launch the application so that the user interface is displayed on the screen.

7 Partial submission for second week

At the end of the second week of the assignment you must submit compilable code that satisfies the tests for the following functions: `checkCreateIndex (indexName)`, `checkCreateTable (tableName)` and `checkPrint (indexName)`.
Upload to Moodle a single file in zip format containing a single folder, named **EDAT_p3_gX_pY**. Here, X is your group, and Y is your team number. This folder should contain:

1. All the source files necessary to compile the test program with the functionality required above.
2. A `makefile` to create the `tester` program.

8 Final submission

Upload to *Moodle* a single zip file in the format described above, containing:

1. All the code files necessary to compile your program and the test files available in *Moodle*.
2. A file `makefile.cod` that allows compiling the program by running the command `make -f makefile.cod` and run the program with the command `make -f makefile.cod run`. After running `make -f makefile.cod run` the user interface will be displayed.
3. Report in pdf format that includes: (1) description of the goal of this practice as well as the algorithms to be implemented; use your own words, do not copy our description; (2) explanation of each algorithm, illustrated with an example.

9 Grading criteria

To obtain a grade of 5 points, your submission must satisfy:

- Your *Moodle* submission includes the code that implements the algorithms required in the following point, and the `makefile` and `makefile.cod` files that allow compiling the code and run the user interface and the tests for these algorithms.
- The implementation provides the correct result for the operations: *use* and *print*, and passes the corresponding tests.
- After each operation the binary search tree or the data is **NOT**saved in memory, but rather on the hard disk. Likewise, the data must be read from the hard disk as needed and nothing else should be stored in memory other than the record that is being accessed at the time.

To obtain a grade in the range 5-6 it is necessary:

- Satisfy all the requirements of the previous section.
- Implement the `findKey` function. The implementation must be valid for any instance of the database and satisfy the test `checkFindKey`
- The program detects if the user tries to enter a repeated key.

To obtain a grade in the range 6-7 it is necessary:

- Satisfy all the requirements of the previous section.
- Implement the `addIndexEntry` function. The implementation must be valid for any instance of the database and satisfy the `checkAddIndexEntry` test.
- Present a report describing the goal of the assignment and the algorithms to be implemented. The report is expected to be well-written and demonstrate your understanding of the assignment.
- The code is well structured and commented code, and clearly identifies where each step of each algorithm is implemented. Every function should include the author's name and a description.

- The program behaves correctly for trees with thousands of nodes. Correctly means that the program is able to create the tree in a reasonable time. Reasonable time is defined as the time taken for the best implementation of this assignment multiplied by 20.
- The program handles errors correctly. For example, if some input file does not have a correct syntax, the program should give an understandable and descriptive error message. Likewise, if an invalid command or key is provided, the program must handle the problem properly. In no case should the program abort or give a coredump.
- The `valgrind` utility should not detect any "memory leak" caused by your code.

To obtain a grade in the range 7-9 it is necessary:

- Satisfy all the requirements of the previous section.
- Implement all the functionality required in the assignment except the tests for `addTableEntry` (but the function itself should be implemented).

To obtain a grade in the range 9-10 it is necessary:

- Satisfy all the requirements of the previous section.
- Implement a collection of non-trivial tests for the `addTableEntry` function.

NOTE: Each day (or fraction) of delay in the partial and/or final submissions will be penalized with one point.

NOTE: Your code should be compilable and run in the image of the linux partition of the laboratory computers provided in *Moodle*. Otherwise the grade will be 0.

NOTE: Failure to use git as a version control system will be penalized with -0.5 points.

NOTE: If a public repository has been used to store the code, the grade will be zero.

NOTE: The code to be graded is the one uploaded to *Moodle*, not code from a repository.

A Binary Search Trees

(Note: this description is taken from https://jesgargardon.com/blog/operaciones-con-arboles-binario-de-busqueda/)

Binary search trees are made up of nodes with a maximum of two children, meeting the following conditions:

- If the node has a left child, the left child's key must be smaller than the node's key.
- If the node has a right child, the right child's key must be greater than the node's key.

A.1 Search

The `search` operation begins by checking the root node, if its key is the one we are searching for, the operation ends with success. If it is not, we move to its left or right child, depending on whether the data we are looking for is less than or greater than the one contained in the parent node. And so we would continue until we find the data or finish going through the tree without finding it.

A.2 Insert

To insert a node in a binary search tree, we go through it in a similar way as we did in the search process and, when we reach a node without children, we will proceed to insert a new node either to the left or to the right depending on whether the new key to store is greater or smaller than the key of the parent node. Note, since our key is a primary key, entering the same key twice is not allowed.

A.3 Tree traversal

The traversal of a binary tree is an operation that consists of visiting all its nodes, in such a way that each node is visited exactly once. There are several ways to implement the `printTree` function, but it is recommended to use the `preorder` traversal. In this traversal, the root is visited (printed) first; then the left subtree is traversed, and finally the right subtree is traversed.

```
preorder (node)
 if node == null then return
 print node.value
 preorder(node.left)
 preorder(node.right)
```