DATA STRUCTURES 2017/18

LAB ASSIGNMENT 3

Fun with binary files.

In this assignment, we shall implement and manage our own table so that the users of the bookstore data base can manage some data locally. We shall write some C program that use ODBC and the functions that we shall write in order to integrate local data and data base data.

This assignment is divided in two parts: in the first part, we shall write a collection of functions that manage a table, storing it in a binary file (the header of these function is given in the file table.h). In the second part, we shall write programs that use ODBC and these function to integrate local and remote data.

1 File Management

You should have seen in theory how a collection of records is stored in a file but, just in case, here is a crash course.

A data base table has a header that, apart from the column names that here we don't need, is characterized by two things: the number of its column, and the data type of each column. In our simple design, there are only two data types: INT and STR (integers and variable-length strings). In the file types.h you have the definitions relative to these types, and in types.c you have some utiility functions that you can use to write your own code.

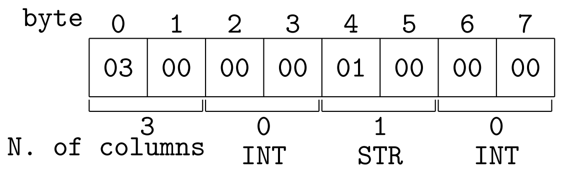
A table is stored in a file, composed of a header and a collection of records. The header of the file is quite simple: it contains the number of columns of the file (an integer) and an array of integer telling us, for each column, what type it is. When the file is opened, we shall create in memory a table structure and store this information in the structure together with any additional information we might need to work (for example: the FILE * for the file where the table is stored). The table structure is passed to the functions that manage the table.

In the type.h file, codes are assigned to the data types. In particular, we define

```
#define INT   0
```

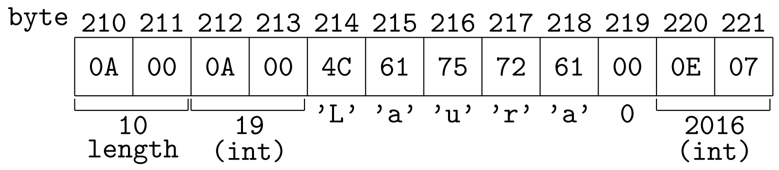
```
#define STR   1
```

So, if we are defining a table with three columns, the first and the third being integers and the second being a string, the beginnig of the binary file would look like this (the figures illustrate the bytes as hexadecimal numbers; for the sake of simplicity, we assume that the integers have a length of two bytes instead of the four bytes used by gcc as a default):



Following the header, we store (in binary format) all the records, one by one. The presence of strings complicates things a bit, since each record will have in general a differnt length (integers all have the same length: 4 bytes in the gcc default). There are a number of solutions to manage records of variable length, here we shall present one of the simplest ones (but you are, of course, welcome to experiment and look for your own solution). The method consists simply in storing the length of the record before the record itself.

Suppose that our three columns store the age of a person, her name, and the year she entered the university. A tuple of this table could look something like: (19, Laura, 2016). The two numbers occupy two bytes each (we are assuming short integers, in your assignment they will probably occupy 4 each) and the name, including the final 0 that marks the end of the string, occupies 6 bytes. Let us say that the record is stored in the file beginning with location 210 (the actual location depends, of course, on how many records are stored before this one and what their length is). The record would be stored as follows (remember that we are expressing number in hexadecimal notation, so the 13 that appears in the first byte is the decimal 19):



With this solution, reading a record in memory is quite easy. Suppose that we have a file pointer, `fp`, already opened, and a long integer, `pt`, which tells us where the record begins. The fragment of code that reads the record in memory is the following.

```
int len;
char *buf;
```

```
fseek(fp, pt, SEEK_SET);
fread(&len, sizeof(int), 1, fp);

buf = (char *) malloc(len);
fread(buf, sizeof(char), len, fp);
```

And it's done! Now, of course, we have to interpret the buffer and transform the data back into numerical variables inside the program. If we had a fixed schema, that is, if we knew, a priori that the first and the third column are integers, and that the second is a string, things would be easy, we could simply do:

```
int age, year;
char *name;
char *tmp = buf;

age = *( (int *) tmp);
tmp += sizeof(int);
name = strdup(tmp);
tmp += strlen(tmp)+1;
year = *( (int *) tmp);

free(buf);
```

Of course, things are not as simple, since we have to read what types the columns are from the table structure that we have stored in memory, but that is something we shall talk about in class.

With this general solution, you are required to implement the interface defined in the file `table.h`. These functions allow a sequential scanning of a table.

Note that, in order to simplify things, we are so nice as not to ask you to implement record deletion: there is no delete function in the table. This means that new records are always inserted at the end of the file and that you are spared the headache of having to manage the erased records list. You are welcome.

What you should do

i) Implement the functions in the file table.c. It is important that the functions work exactly as specified: in this case you have no freedom to decide how things should work, this has been decided for you: your requirements have been expressed in the file `table.h`, and you have to implement them.

ii) Write a program that tests the table: the program will create a simple table, insert a few records, close the table, open it again, read the records and print them. The functions in the file `types.c` have been provided for your convenience, you can use them or not, depending on your preference.

iii) Add the data type LLNG (long long integers) and DBL (double precision numbers) to the table. This will probably be easier if you make judicious use of the type functions that have been provided, but it won't be too hard anyway. Write a program to test your new data types.

2 Using local data

Now you have implemented (and tested) your table functions, it is time to use them. We shall add a "personal" table to the bookstore data base. The table is personal in the sense that the user keept it on his computer and doesn't share it with the rest of teh world.

The table contains personal evaluations on books that the person has read. Its schema is the following:

```
(ISBN : STR, title : STR, score : INT, book_id : INT)
```

where score is an integer from 1 (the book sucks) to 100 (wow! This is the best book ever!).

This information is to be stored in a local table managed using the functions implemented in the first part. Of course, the table has to work together with the bookstore database, and this is where ODBC comes in... again!

You must implement two C programs that work as follows:

```
score <title> <score>
```

Gives a score to a book. For example:

```
# ./score "La vie, mode d'emploi" 100
# ./score "L'etranger" 100
# ./score "The Da Vinci Code" 2
```

gives a 100 to *La vie, mode d'emploi* (I happen to love that book) and to *L'étranger* (that too) and a 2 to *The Da Vinci Code* (you can imagine what I think of it).

The second program asks for suggestions: you give it a score, and it reports a list of all books from authors that have obtained that score.

```
./suggest <score>
```

For example, if the three executions above are the only two scores I have given, I can do the following call with the given results:

```
# ./suggest 100
Georges Perec    W ou le souvenir d'enfance
Georges Perec    Un homme qui dort
Georges Perec    La vie mode d'emploi
Albert Camus     La peste
Albert Camus     L'etranger
Albert Camus     La chute
```

Note that, in order to simplify things, the results are not filtered: the books that we have scored appear in the list.

i) Implement the two programs and test them. Turn in the code.

ii) Show an execution of the two: call the first program to score a book and then the second one to get back suggestions from the author of the book you just scored (try to choose an author without too many books, so as not to clutter your putput). Copy the terminal output in a file and add it to your report.

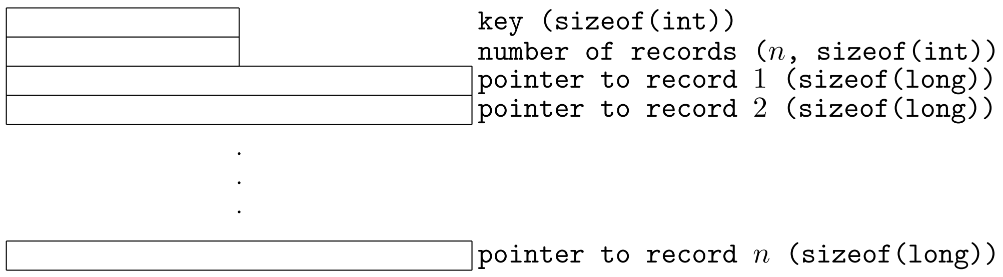
3 More work

The last part of the endeavor is to create an index. We create simple indices (indices that are managed in memory) and, to simplify things, only on integer fields (but check the optional part). For this last part, you are required to implement the functions in index.c and use them to create an index on the score field and one on the book_id of the table.

The indices that we use are a hybrid between secondary indices and primary ones. We do use them as secondary indices (which means that key can be repeated: the same secondary key may be associated to a plethora of records, from one to a gazillon). On the other hand, we do not associate the primary key to the secondary one to make searches indirectly. Rather, we associate to each key the pointers to the records, that is, the position of the corresponding record in the table file. This simplifies things, and we can get away with it in this assignment since in our table records are never deleted, so a record never changes position once it has been created.

Indices are stored in a binary file but, unlike tables, when we open the file we read the whole index in memory. All the information necessary to use the index is stored in an index_t structure that you will define.

How is the index stored in the file? Well, there are a number of ways. You can simply store it as pairs (key, pointer) (an integer and a long), keeping in mind that the keys can be repeated, something which complicates the search a bit. One alternative solution (I think it is easier, but opinions on the matter vary) is to store each key only once followed by all the positions corresponding to that key. That is, the file image of an entry in the index would be something like this:


key (sizeof(int))
number of records (n, sizeof(int))
pointer to record 1 (sizeof(long))
pointer to record 2 (sizeof(long))
pointer to record n (sizeof(long))

(Remember that in your memory management you have to make sure that the list of pointers associated to a key is kept ordered.)

The header of the file will contain only two integers: a code from types.h telling us what type is the index (in this version this will always be INT—that is, 0), and the number of records in the file.

What you should do

i) Implement the functions in the file index.c. It is important that the functions work exactly as specified: in this case you have no freedom to decide how things should work, this has been decided for you: your requirements have been expressed in the file `index.h`, and you have to implement them;

ii) write a program `score` that works as explained in the previous section;

iii) write a program `suggest` that works as explained in the previous section;

iv) turn in all the code that you have written

Finally:

Write a report explaining your work, the problems that you have found and your solution. Have you implemented the indices as mentioned here, or have you found your own solution? Have you implemented the table the way I did, or have you found your own way (there are other ways... we'll talk about it in class).

4 Optional

For those of you who didn't have enough and want to do some more programming (always a good idea): change the file index.c so that it creates indices both on INT and on STR. Note that now the secondary key is no longer of fixed length (but your implementation of table should give you some idea on how to fix the problem).

Test your code by writing an index for the title field of your score table. To test it, write a program that, given a title, returns all the books in your personal archive that have the same score:

```
./scorematch "la vie, mode d'emploi"
"la vie, mode d'emploi"
"l'etranger"
```

(Note that this program is completely self-contained, you don't need to use ODBC to connect to the data base.)