DATA STRUCTURES 2019/20

LAB ASSIGNMENT 3

Fun with binary files.

In this assignment, we shall learn how tables and indices are implemented by implementing a simple table and an in-memory index.

Note that the interface of the table management and the index library is given to you in the files ”table.h” and ”index.h” and partially implemented in ”table.c” and ’dqtindex.c: your objetive is to implement these functions, which must work exactly as requested. A test program will be given to you do check whether your functions work correctly.

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.

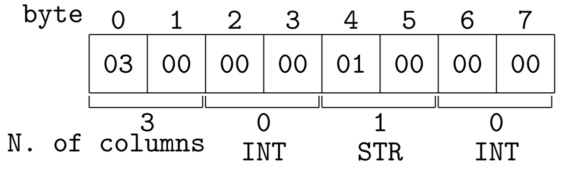
A data base table has a header that, apart from the column names that here we don't need, is characterized by two things: the number of its columns, and the data type of each column. In our simple design, there are only two data types: INT and STR (integers and variable-length strings). In the file type.c you have a few utility functions to work with these types, in particular, you have a function that, given a type and a value of that type, gives you the occupation in bytes of that value. We'll see how that function can help implement the table.

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. That way, the same functions can work on different tables at the same time simply by receiving different structures.

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

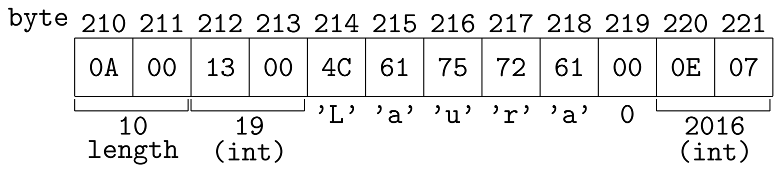
```
#define INT   0
#define STR   1
```

So, if we are defining a table with three columns, the first and the third being integers and the second being a string, the beginning 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 differnet length (integers all have the same length: 4 bytes in the gcc default, 2 bytes in our drawings). 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 occupy 4 each) and the name, including the final 0 (never forget the final 0!), 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) Use the program provided to test the table (see below)

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.

iv) Test the new data types: the program provided should work, once you have done the required modifications to "type.c" and "type.h".

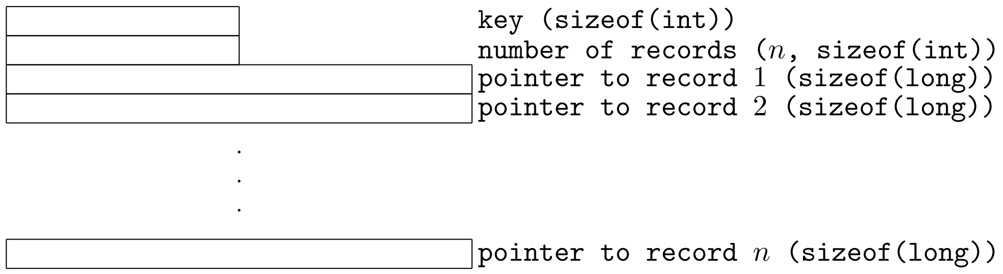
2 Indices

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 part, you are required to implement the functions in index.c and use them to create an index on the score field.

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))

Note that, while the keys have to be kept ordered in order to use binary search, it is not necesary that the list of keys associated to a key be kept ordered.

Remember also that **all the searches must be done using binary search**, no linear search is allowed. You can use the binary function in C, write your own, or use one that you have.

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.

Your goal:

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) Use the test program to check your work.

2.1 A trick on binary search

When you insert a key and search it in the file, you may have two possibilities: either the key is already in the file (in this case you simply have to add a new position value to the end of its list, since this is not kept in order), or it is not. In this case you must insert the key in the proper position so tha the array of lists is kept ordered. Binary search has a nice property: if you don't find in the array the element you were looking for, at the end the "middle" index tells you where this element should go. This is useful information, and it would be a pity to let it go to waste. In my program, I let my binary search function return a non-negative value, middle$\geq$0if I found the key in the position "middle", and a negative value, indicating where the key should go if I didn't find it. Now, I can't simply return "-middle", that is, the following fragment doesn't work:

```
int binary(array, key, n) {  // THIS DOESN'T WORK!!
    int low = 0;
    int high = n-1;
    int middle = (low+high) >> 1;
    while(low <= high) {
       .
       .
       if (array[middle] == key)
          return middle;   // key found
       .
       .
    }
    return -middle  // Key not found.
}
```

The problem is that $-0 = 0$, so if the function returns 0, I don't know whether the key has been found in the position 0 or it hasn't been found and it should go in the position 0. To solve this solution, rather than returning "-middle" I return "-(middle+1)" so that I am sure that, when the key is not found, I always have a negative number

```
int binary(array, key, n) {  // THIS WORKS!!
    int low = 0;
    int high = n-1;
    int middle = (low+high) >> 1;
    while(low <= high) {
       .
       .
       if (array[middle] == key)
          return middle;   // key found
       .
       .
    }
    return -(middle+1)  // Key not found.
}
```

When I call the function, I do something like:

```
int m = binary(array, key, n);
if (m>=0) {
   .
   .  // Found: key was found in position m
   .
}
else {
  m = -(m+1);
  .
  .   // Not found: key must go in position m
  .
}
```

3 How to test your stuff

In the directory in which you have to work (the directory development/src) there are a number of files that implement a test program for your code. Among them are the files "table.c" and "index.c" that you have to work on. The test program is created with the command "make all" and creates a program, which includes your files, in the directory install. In the same directory, you will have a couple of text files containing data that will be stored in your table. It would be a good idea to create your own test files, on the model of those provided, to check your work. Start with small files and build from there.

The test program accepts the following commands:

read <file> reads a text file and keeps its contents in memory.

tmake <file> stores the contents of the text file currently in memory in a table using the table.c functions, the table is stored in the given file. The table is kept open in memory.

tclose closes the table that is currently kept open.

topen <file> opens the given table file and keeps it open in memory.

check checks the contents of the current text file with those of the table currently open in memory.

tshow <n> shows the contents of the nth record of the table and of the nth record of the text file currently in memory (if they are loaded in memory).

verify <file> complete verification step: loads the text <file>, stores it as a table in <file>.dat, closes the table, opens it again, and checks the correspondence of all the records.

mkindex <file> creates an empty index in the given file.

iinsert <key> <pos> unserts the pair <key> <pos> in the index currently in memory.

isave saves the index in the current index file.

ifind <key> finds the key <key> and displays the position(s) associated.

tindex <file> <col> creates an index in the file <file> for the column <col> of the table currently in memory, and stores all the record in the index

retrieve <key> retrieves and prints the contents of the key <key>.

ishow shows the whole contents of the index currently in memory.

4 Finally:

Write a report explaining your work, the problems that you have found and your solution. Any special problems implementing the tables? How have you done it, which solution have you used to store the length, the same one I used or a different one?

Have you implemented the indices as mentioned here, or have you found your own solution? Have you used the C program binary search? If not, have you written your own, looked for one on the internet, used an old one you had lying around? Motivate your decision.

Turn in a *good* report and all your code.

Do not include your code in the report. You might want to include some snippets if you need them to explain some fine point (the way I have done it to explain the return value of the binary search, for example). If you need to explain some algorithm that doesn't depend too much on the C functions, pseudo-code is also a good option: that way you can explain your algorithm without being bogged in the details of the actual implementation. Use, as always, your own good judgment and common sense.

5 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).

Write your own test program, or modify the one given (understanding and modifying a progrm written by somebody else is aleays a good exercise) to check your work.