Javier Delgado del Cerro
Javier López Cano

REPORT - ASSIGNMENT 3

Once the database “books” is created on the localhost server, with *alumnodb* as postgres' username and password using the provided dump, you can just execute the provided execute.sh (after give it permissions with *chmod 775 ./execute.sh*) to see a working example. The text in blue and purple are just our explanation, while the others are the program values.

Table

Regarding the implementation of the table struct we had to make two important decisions.

First, we realised that in order to use the table we have to access almost constantly to the file were the table is stored so, as this was highly inefficient, we decided to keep the file constantly open (until the end of the program where we close it) and store in the struct table a pointer to the file so that we can write and read from the it easily.

The second mayor decision we had to take was that when we read a record with the function read-record, the memory in which the record is temporarily stored, is freed in the same function the next time is called, or when the table is closed, so the user should not free the memory of the void** that this function returns.

ODBC

To implement the program score we have decided to implement function called *fill_scores* wich receives as an argument the path to the binary file where the table is stored and a double pointer to void with the data that has to be stored in the table and prints this information in the document.
In the main of this program we use the functions of ODBC and some SQL queries to get the book_id and the ISBN from the book given as an argument to the program and after we get these values, we create a void** that stores these two as well as the title and the score given to the book (both given as arguments to the program). After this we call the function *fill_scores* with the path to the document and this void** created so that it prints these values to the table.

To implement the second program we were asked for (suggest), we created another function called *get_books* which receives a score as an argument and returns an *int** containing all the book_ids from the books with the given score stored in the file of the table. To get these book_ids, as we still don’t have any kind of index, we read the table records linearly.
In the main of this second program, we first call the *get_books* function with the score passed as argument of the program, and the we use the ODBC functions and some SQL queries to get the title of all the books of the same author or authors as the books in the int* returned by the *get_books* function. In the end, we print the author and the title of all these books.

For the executables we use a book written by Cervantes and another one from Hugo Albert Rennert because they don’t have many books and it is easier to see the output of the program.

Javier Delgado del Cerro
Javier López Cano

Index

We have defined a structure called *irecord* which stores a key and an array with the positions of all the records that contain that key. In addition to this, we use two *ints* to identify the number of positions of the array and the maximum capacity of it (its size).
To implement the structure index we use an array of pointers to the structure irecord, a string which contains the path to the file where the index is stored so that we can open it and close it at will (e.g. when we call the function *index_save* we have to erase everything in the file and rewrite it so that it stays in the correct order), two *ints* that tell us the number of *irecords* that the index has and the maximum number of *irecords* the index can store, and lastly, a variable that tells us the type of the element we have stored as the key.

For the index programs we store at the beginning of the table the type of key and the total number of records. After this, we store for each key, the key, the number of records of the given key and all the records one after another.
When we insert, we save memory for bigger arrays tan the ones we need so that we can insert several without having to make re-allocs constantly, as this would be highly inefficient. This "extra memory" is defined in a macro called NUM_ADD.
In order to insert we use the algorithm insertSort so that the table remains in the correct order. Lastly, we use binary search in the table records inside of the index in order to find all the records that have the same key.

Finally, to test the index we use *score_index* and *create_index*, which works exactly as the previously mentioned *score* and *suggest*, but using the index. We create the index in score_*index* and we insert the index data at the same time as the table data, later, in *suggest_index* we use this index to get the records' positions.

Optional

In order to do the optional assignment, we store, for each key, the length this key has (to know if it is a string), the key, the number of records the key has and the position of each of these records.
We use void pointers to store the key so that we can use them as a char* if the key is a string, or as an *int** if it is an integer.
Furthermore, we had to implement a function *cmp* which allow us to compare the keys (regardless if they are integers or strings) so that we can use binary search and insertSort.

To implement the *scorematch* program we have decided to do it separately, and, as a result, it is in that specific program where we create the index by reading the full table.

In this optional exercise we only include the programs to test the index as the table is the same as in the previous ones.