DATA STRUCTURES

ASSIGNMENT 3:

**FUN WITH BINARY FILES**

S. Xiaofan Fernández Marín and Samai García González

Group 1292

Pair 1

**INTRODUCTION**

The objective of this assignment is to learn to implement tables and indices by implementing a table and an in-memory index.

**1.FILE MANAGEMENT**

In this part of the practice we were asked to implement the functions defined in table.c.

First of all we created the table structure we needed, and then we completed the prototype functions.

The first function we were asked to implement was **int table\_create(char\* path, int ncols, type\_t\* types)**. Here, we first open a file with the route defined in path argument, and we write in it the number of columns and the type for each of them.

The second function we created was **table\_t\* table\_open(char\* path)**, where we really created the table. First we reserved memory for the table and we open the pointer to the file of the table. Then we read the number of columns from the file and we reserved memory for as many types of data as columns are. After that, we read from the file the type of data for each column and we wrote it on the table. Finally we filled the field of the structure where the first position of the table is stored.

Next we filled the function **void table\_close(table\_t\* table)**, which closes the file of the table and frees all the resources of the table.

Then, we developed **int table\_ncols(table\_t\* table)**, **type\_t \*table\_types(table\_t\* table), long table\_first\_pos(table\_t\* table) and long table\_last\_pos(table\_t\* table).** In the three fisrt ones, we just had to return their corresponding field in the structure. In the last\_pos function, we just had to situate on the end of the table with fseek and return the position with ftell.

Next function was **long table\_read\_record(table\_t\* t, long pos)**. This function reads a record that begins at a given position in the table file. First we access to that position through fseek. Then we freed the old buffer if it exists. Then we reserved memory for as many values as the record has, known by the number of columns.

Then we coded function **void \*table\_get\_col(table\_t\* table, int col)**. This function just returns the pointer to the value contained in each column of the record currently in memory.

The last function to implement was **int table\_insert\_record(table\_t\* t, void\*\* values)**. The purpose of this function is to insert a record at the end of the file. We did this with a loop that executes as many times as columns are.

The way we have used to store the length was different from the one of the professor. Instead of storing first the total size of the record, and then the value of each column, we first store the length in bytes of a single value or column, and then the content of it. And this progressively until we reach the last column.

Once we did that, we had to check that they work using the test program provided.

Finally, we had to add the data types LLNG (long long integers) and DBL(double precision numbers). This was not very difficult, we just had to add them to the enumeration and make some little changes in the functions in table.c.

**2.INDICES**

In this part of the practice, we were asked to create an index on integer fields.

We have created the structure index as we have needed while coding.

The first function we were asked to create is, obviously, **int index\_create(char \*path, type\_t type)**. As in table, this function creates a file given the path in the argument of the function, and writes the value 0 in it.

Then, we developed **index\_t\* index\_open(char\* path)**. The first thing we did was creating a data of type index and reserving memory for it. Then we open the file with the path given as argument, and we read the data inside it to know what we need to initialize and the memory we should reserve for the different fields of the structure. First we read the number of keys, and we reserve the corresponding memory.

Next function was **int index\_save(index\_t\* idx)**. This function is in charge of opening the file linked to index, and write in that file the data corresponding to the index: the number of keys, the keys, and the position for each key.

Then, we did **int index\_put(index\_t \*idx, int key, long pos)**. The goal of this function is to introduce in an index a particular key in one position. Here we reached two cases: the first one, is when the index is empty, when we just have to reserve memory and introduce the key in the corresponding position and finally we save that position for later functions.

The second case is when the index is not empty. In this case, we first have to see where we need to introduce the key, done with binary search. Then we created an auxiliary table to introduce it in index and finally we reordered everything.

Next function was **long \*index\_get(index\_t \*idx, int key, int\* nposs)**. This function returns a key from the index as well as its position. We again used binary search to see if the key is in the index. Once we did that, we reserved memory for the position and returned the position.

Then we developed **void index\_close(index\_t \*idx)**, as it names says, it is in charge of closing the file attached to index and freeing the resources of the index structure.

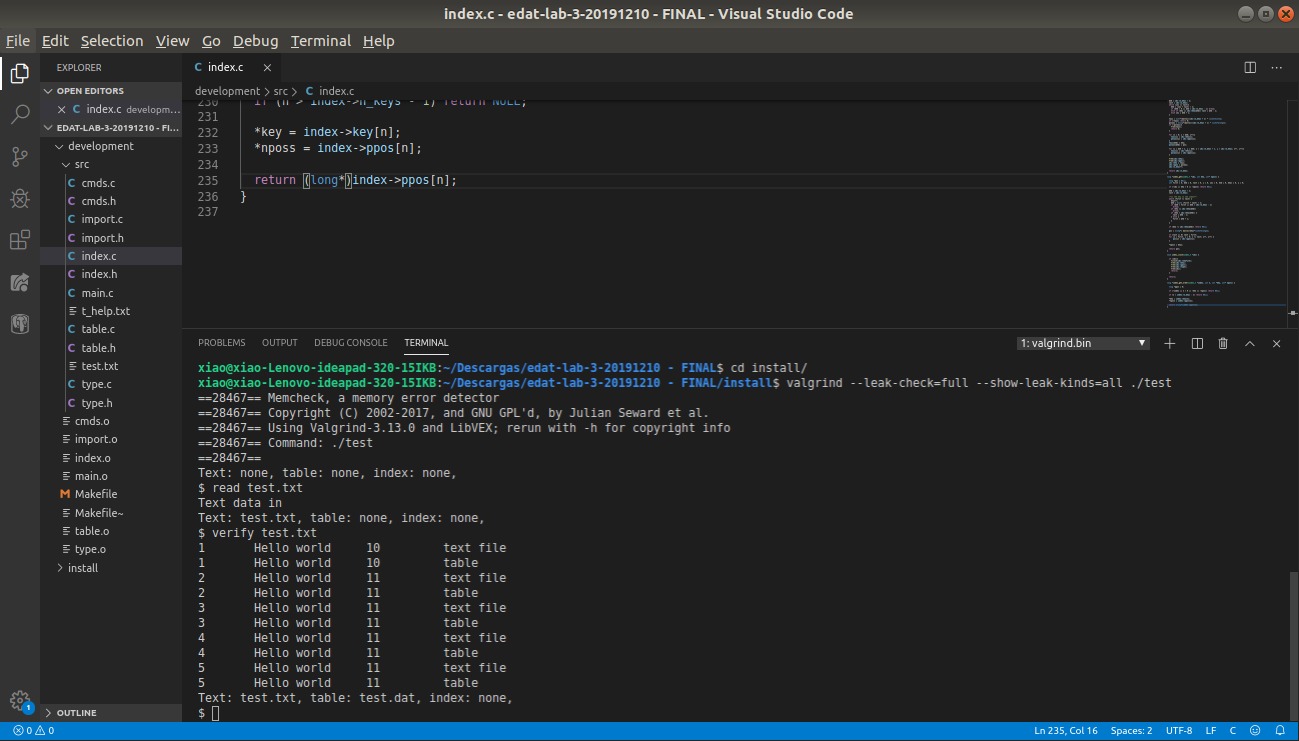
Last function was **long \*index\_get\_order(index\_t \*index, int n, int \*key, int\* nposs)**. This function is used for debugging and returns the nth record in the index.

To do this part of the assignment, we have used the C program binary search given in the enunciate.

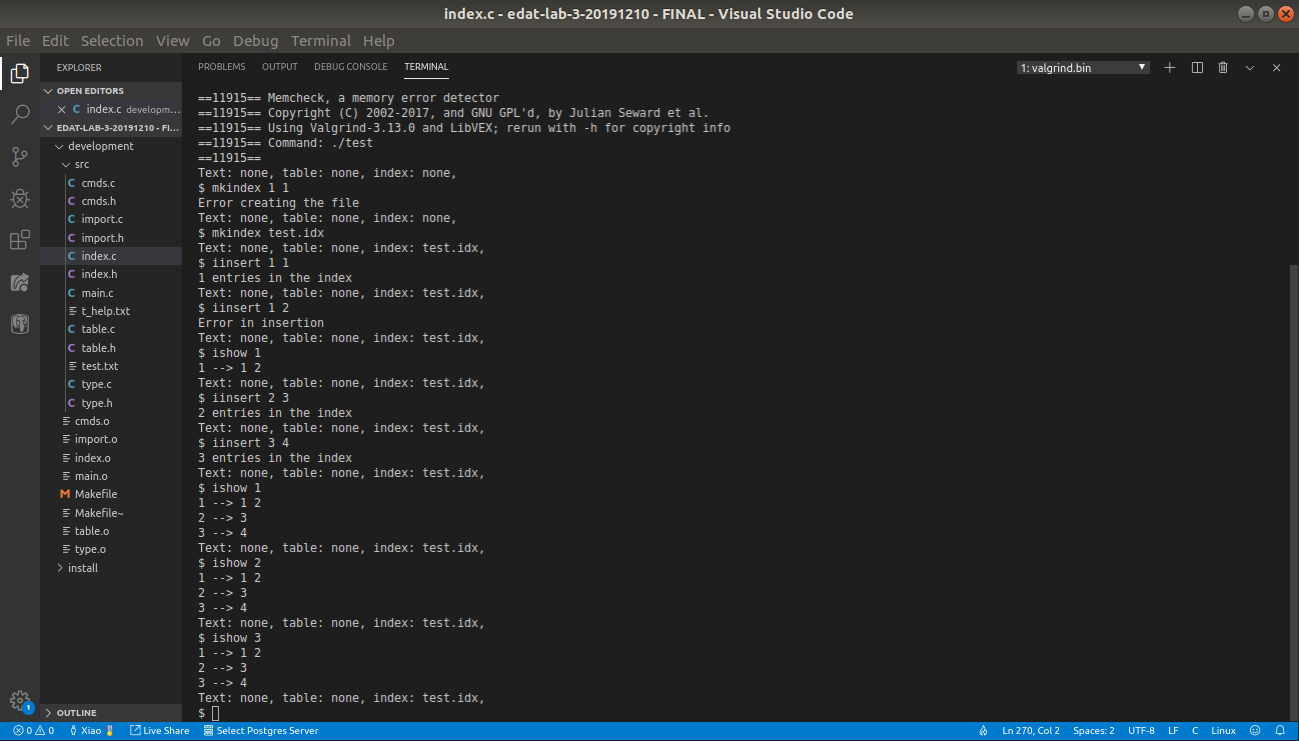
**3. HOW TO TEST YOUR STUFF**

In this last part, we were asked to test the files table.c and index.c to see if they work how they should. They gave us a test program to do this, and several commands to test the files.

Here is the result of testing “table.c” with the command verify <table>, which 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:



For testing index.c, we used command ishow, which shows the whole contents of the index currently in memory. Here is the result of doing it:



**CONCLUSION**

In this practice, we have learned how tables and indices work, and how a collection of records is stored in a file. It was nice to implement this by ourselves, although there were some functions that costed us a little bit more, such as table\_read\_record or index\_put.

Despite that, we finally obtained a more or less satisfactory result and we learned how binary files worked.