**replaceExtensionByIdx.**

This function is created to change the extension in a file. First, we copy in the target char indexName the same letters in fileName until we arrived at a dot. Then, we add the end of the array ‘\0’ at the end of this name and at the end +4. Then we add the extension “.idx” and return.

**createTable.**

This function tries to open a file called as the indexTable. If the file does not exist, the program creates it to use it in binary language and then writes -1 in the line of first element in deleted registers.

If it exists, it opens it to read it in binary language.

After that, we allocate memory for the index name and call the function replaceExtensionByIdx to change the name’s extension. We call the function crerateIndex to create the index with the name, we free the memory allocated for the index name and we return the result of the last call.

**createIndex**

This function tries to open a file called as the indexName. If the file does not exist, the program creates it to use it in binary language and then writes -1 in the root and in the deleted registers.

If it exists, it opens it to read it in binary language.

Then, it returns true.

**printNode**

The stop condition of the recursion is if the level we are printing is bigger than the level up which we want to print. If it is not, we add one to the level counter.

Then, we jump in the file until the first node to print, we read its information, we print the tabulators needed, then the side if needed, and then the information of the node.

After the print, we call recursively to the function printnode, first with the left son and “l” as side, and then with the right son and “r” as side.

**printTree**

This function opens the index file, reads the root node, and calls the function printnode.

Then we close the file and return.

**findKey**

To implement this function, we need to follow some steps.

First, we will open a file called as the param indexName.

Once it is open, we will read the first number in the file, which is the root of the tree. Then, while the next node exists, we have to do some things.

Our comparation is going to start in the root\_node, so we have to send the pointer to the line in where it is and read the value. This value is the primary key, which we are going to compare with the book\_id we are trying to find.

If the primary key is the same as book\_id, this means we have found the key, so we are going to jump to where the offset is written in the line, read the value and store it in the pointer nodeIDOrDataOffset. Then we are going to close the file and return true.

If the primary key is bigger than the book\_id, this means that the key we are trying to find, if exists, is going to be in the left subtree of this primary key with which we are comparing, so we need to move on to the left son of it and starts the process again.

Then, if the primary key is smaller than the book\_id, this means that the key we are trying to find, if exists, is going to be in the right subtree of this primary key with which we are comparing, so we need to move on to the right son of it and starts the process again.

If the key is not found, we should close the file, equal the pointer nodeIDOrDataOffset to the last position in where we have searched and return false.

**addIndexEntry.**

First, we have to check if the key we want to add is already in the index. If so, there is nothing to add and the function finishes. If not, we have to open the file in order to write in it. There are two possibilities, if there are not deleted records, the key is going to be insert at the end of the file. If there are deleted records, we can add the key in the position of the deleted record, we have to change the delete record in the header. Then, for both cases, we have to decide if the key inserted is the left son or the right son of its father.

If there are not deleted records, we send the pointer to the end of the file with fseek and we add the information with fwrite. First the id, then the left son, the right son, the father and the bookOffset. Then we assign the new position in where the node has been inserted. Finally, we check if the root is -1, and if it is, we update it with the position in where the new key has been inserted.

If there are deleted records, we send the pointer to the deleted entry, we save the next node deleted in a variable and change the information with the new one. Then we send the pointer to the first line and update the deleted record with the one saved.

After this, we compare the primary key of the father and of the key inserted, so if the son is bigger it will be the right son of the father and if not, the left son.

**addTableEntry.**

This function adds a new entry in a table in a .dat file. There are some cases that we have to consider. If there are not deleted records, it is going to add the element at the end of the table.

If there are deleted records, the function is going to try to add it in any of the spaces of the deleted records if the title of the new entry fits. If not, it will add it at the end of the file.

First, we need to know if the new record already exists in the file by calling the function findKey(). If so, we say that the book already exists and return false. If not, we open the file and read the offset of the deleted register.

If there are not deleted registers, we send the pointer to the end of the file and write the information there.

If there are deleted registers, we sent the pointer to the first deleted register and in a do-while loop we read the offset of the next node deleted and the size of this one.

Now, we should compare the size of this node with the size of the new entry. If they are equal or the new size is smaller that the original, we can write in this position all the information, add a new entry in the Index, close the file, and return true.

If the sizes don’t fit, we send the pointer to the next deleted node by updating the variable with the offset.

When the loop finishes, we add a new entry in the Index, close the file, and return true.

**checkAddTableEntry.**

This is a function created in order to verify if addTableEntry works properly. Therefore, we add some elements with different cases and print the new tables with them inserted.

The cases we have add are:

* Adding an existed book: MAR2.
* Adding a book in a table with deleted records and a title of the same size that one existed.
* Adding a book in a table with deleted records and a title of a smaller size that one existed.
* Adding a book in a table with no deleted records.

To do these cases, we have allocated memory for a book, then write on it an Id, a length and a title and have called to the function addTableEntry() with the parameters. Finally, we print the new table, and we can see that the entries are made.

Also, we needed to change tester.c file and checkAddIndexEntry.c file. In the second one, we only need to erase the final exit(EXIT\_SUCCESS); to make sure that the next test is going to be run. And in the first one we have added the declaration of the checkAddTableEntry() function in the beginning of the file and then call it in the main function.

**menu.c file.**

**main ().**

The main function calls inside a while-loop the ShowMainMenu() function and checks if the input is correct and is one of the options we have. If it is, it calls the function that corresponds to each of the options.

For the Use interface, we will call to the function ShowUseMenu().

For the Insert interface, we will call to the function ShowInsertMenu().

For the Print interface, we will call to the function ShowPrintMenu().

**ShowMainMenu().**

In this function with a while-loop we print the interface with the menu, and we scan the option the user writes by keyboard.

We assign the value nSelected from 1 to 4 depending on the user’s input and return it.

**ShowUseMenu().**

This function shows and processes the use menu. We ask for the name of the file we want to create, scan it, call the function createTable() and the function replaceExtensionByIdx() using this name and a variable to store the index name.

**ShowInsertMenu().**

This function shows and processes the insert menu. We ask for the name of the file in where we want to write and scan it. Then, we verify that the file exists and allocs memory for a book structure.

We asked for the book’s key and for the book’s title, scan it; we allocate memory for the book title in the structure, we save the information of the book in the structure and then call the function replaceExtensionByIdx() to get the index name and then the function addTableEntry(). Then, we close the file and the function ends.

**ShowPrintMenu().**

This function shows and processes the print menu. We ask for the name of the index name, the level until which we want to print the index and the table name the user wants to print, scan them. Then, we verify that the file exists and we call the printTree() function and the printTable() function.