**TOOLS AND METHOLOGY**

Our environment has been Windows, in where we have used the Visual Studio Code with the terminal associated, gcc and Valgrind to compile and check all the memory errors. Then, to create the different histograms we have used GNUPlot.

**Exercise 1. Dictionary ADT**🡪 First part of the practice consists of creating all the function to use an ADT dictionary.

* PDICT init\_dictionary (int size, char order) 🡪 in this function we receive two arguments, the size of the dictionary we are above to create and the order (SORTED or NOT\_SORTED). We need to allocate memory for the dictionary and for the table of elements. We also need to initialize the elements of the structure.
* void free\_dictionary (PDICT pdict) 🡪 function that receive the dictionary to erase. It is in where we free the table, we have allocated in init and the dictionary structure.
* int insert\_dictionary(PDICT pdict, int key) 🡪 this function receives the dictionary in where we want to introduce the key and the key to introduce. It introduces the key element in the right position. If the array is not sorted, it will introduce the key in the last position and add one in the number of elements in the structure.

If the array is sorted, we add the key in the last position and add one in the number of elements of the structure and then we compare our key with all the other elements in order until we get the right position.

* int massive\_insertion\_dictionary (PDICT pdict,int \*keys, int n\_keys) 🡪 this function receives a dictionary in where we want to introduce the elements, an array of keys to introduce and the number of keys to introduce. It uses a for loop from 0 to the number of keys in where it calls the insert\_dictionary function.
* int search\_dictionary(PDICT pdict, int key, int \*ppos, pfunc\_search method) 🡪 it receives the dictionary in where we want to search, the key to search, a pointer to store the position of the key in the dictionary and the function we want to use to search. It calls the function we want to use to search with the table of the dictionary, 0 as initial position, the number of elements in the dictionary minus one as last position, the key to search and the position’s pointer as arguments.
* int bin\_search(int \*table,int F,int L,int key, int \*ppos) 🡪 routine to implement the binary search. This function receives the table in where we need to search, the first position to search, the last position to search, the key to find and the pointer to store the position in where the key is found. It needs a sorted array because it divides the array by the middle and searches in the middle that correspond calling recursively to the function until the key to find is equal to the element in the array.
* int lin\_search(int \*table,int F,int L,int key, int \*ppos) 🡪 routine to implement the linear search. This function receives the table in where we need to search, the first position to search, the last position to search, the key to find and the pointer to store the position in where the key is found. This function does not need a sorted array and searches by comparing the key with all the elements in the array in a linear way.
* int lin\_auto\_search(int \*table,int F,int L,int key, int \*ppos) 🡪 routine to implement the linear auto search. This function receives the table in where we need to search, the first position to search, the last position to search, the key to find and the pointer to store the position in where the key is found. In this function, the array is sorted in a way that the elements that are more likely to appear are in earlier positions than the others. The system to search is the same as in linear search, but the time is reduced because of the order in which the elements are in the array.

**Exercise 2. Comparison of the search efficiency**🡪 In this part of the practice we need to create some functions in order to measure the efficiency of the searching functions we have already created in exercise 1. Specifically, we have to create two new functions similar to the ones created in times.c in the last practices.

* short average\_search\_time (pfunc\_search method, pfunc\_key\_generator generator, char order, int N, int n\_times, PTIME\_AA ptime) 🡪 this function receives as arguments the method we want to use to search, the method to generate the keys, the order of the structure, the number of elements in the dictionary (the size), the number of times to search each key and a structure ptime to store the information measured. To create this function we have followed the steps in the pdf assignment and we have combined with the ideas in function average\_sorting\_time() we have created in last practice.
* short generate\_search\_times(pfunc\_search method, pfunc\_key\_generator generator, int order, char\* file, int num\_min, int num\_max, int incr, int n\_times) 🡪 this function receives as arguments the method we want to use to search, the method to generate the keys, the order of the structure, the file we will use to store the information, the minimum number to start the searching, the maximum number where the searching must stop, the incrementation between the elements and the number of times we want to search each of the keys. This function allocates memory for a structure to store all the values we want to measure and then calls average\_search\_time() function and save\_time\_table(), this last is the same as in the last practice and it stores in a file the information.

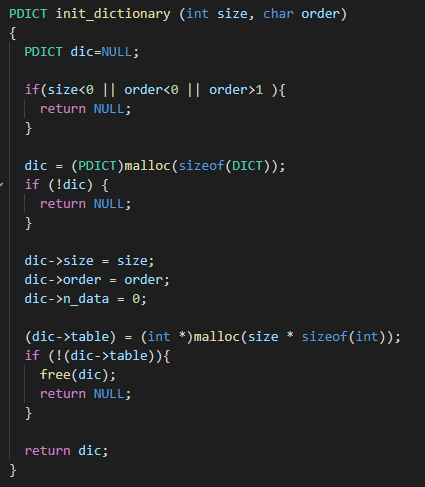
In the next part of exercise 2 we have to different subexercises.

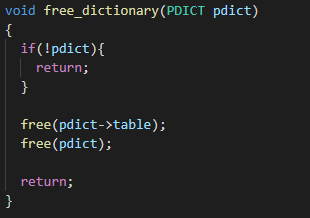
In the first one we have to compare the average clock time and average number of basic operations of the linear and binary search. To do this we need to take into account that binary search needs a sorted array, but linear doesn’t. We will add the changes explained in the section printed code.

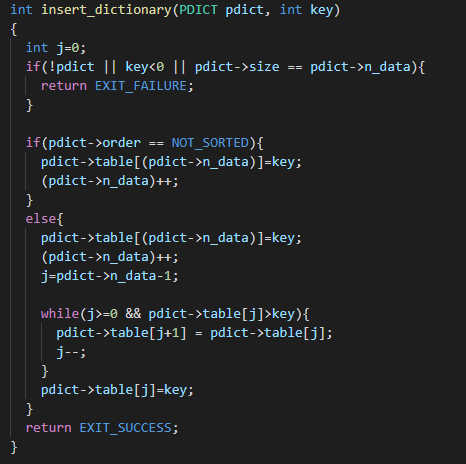
In the second, we need to compare the average clock time and the average number of basic operations for two algorithms: first, the binary search for sorted dictionaries, and, second, the auto–organized search (lin auto search) in unsorted dictionaries, changing the value of the n times parameter to 1, 100 and 10000. As in the first exercise, we will add the changes explained in the section printed code.

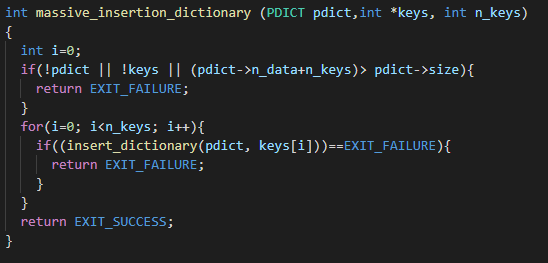
**PRINTED CODE**

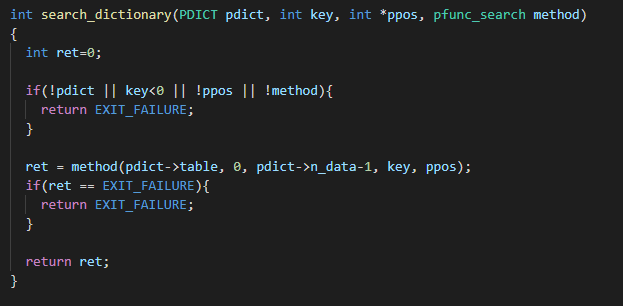
**Exercise 1:**

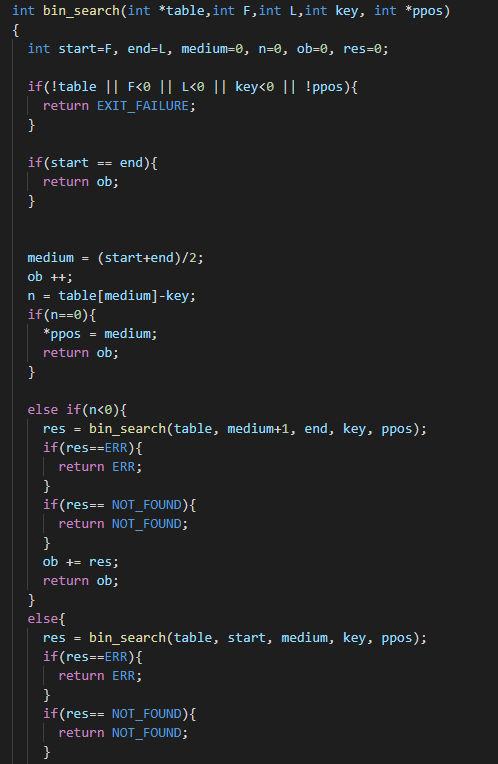


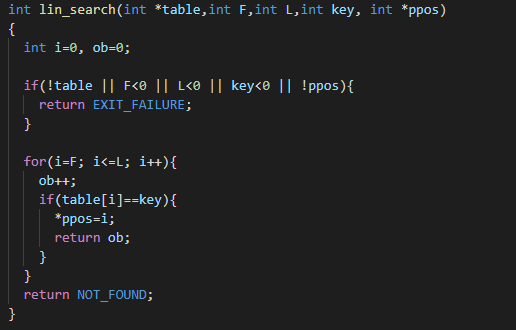


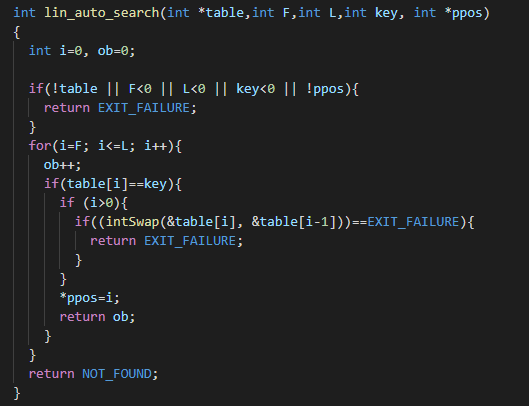




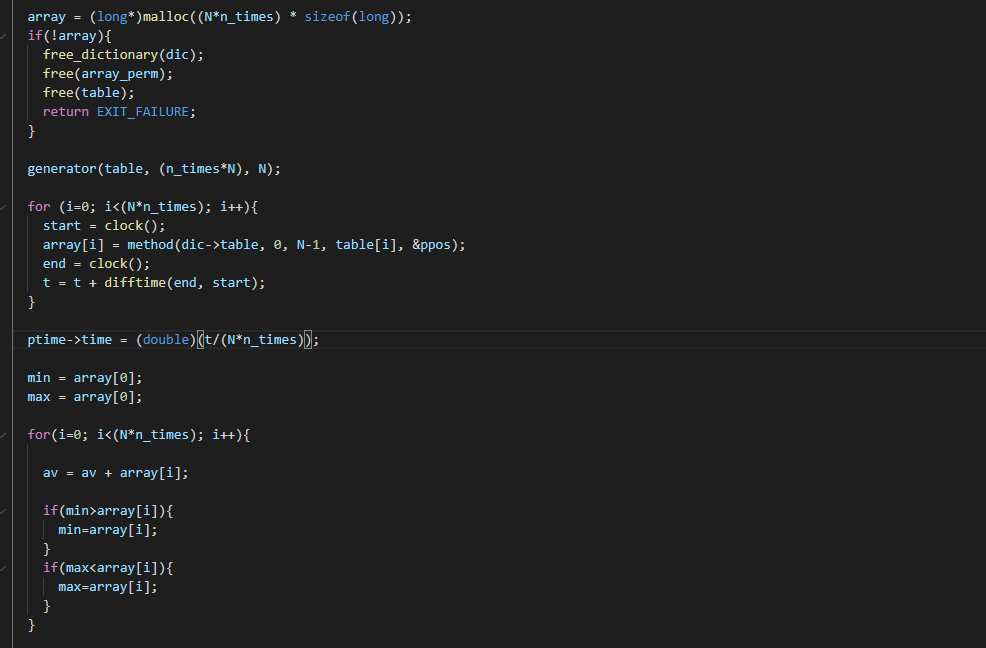
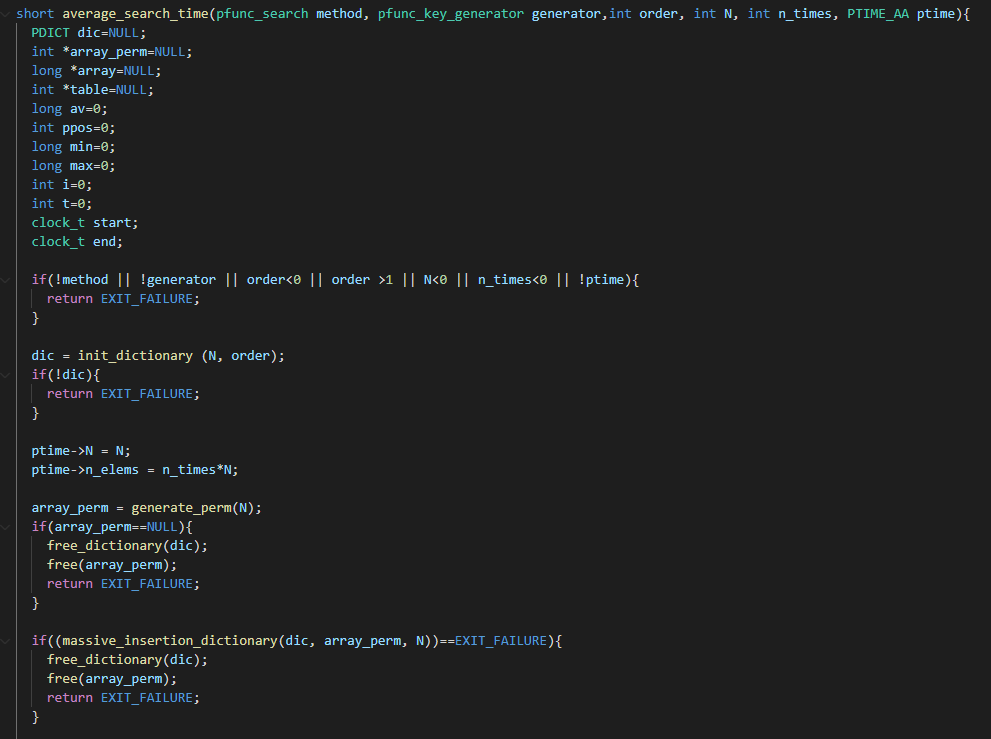


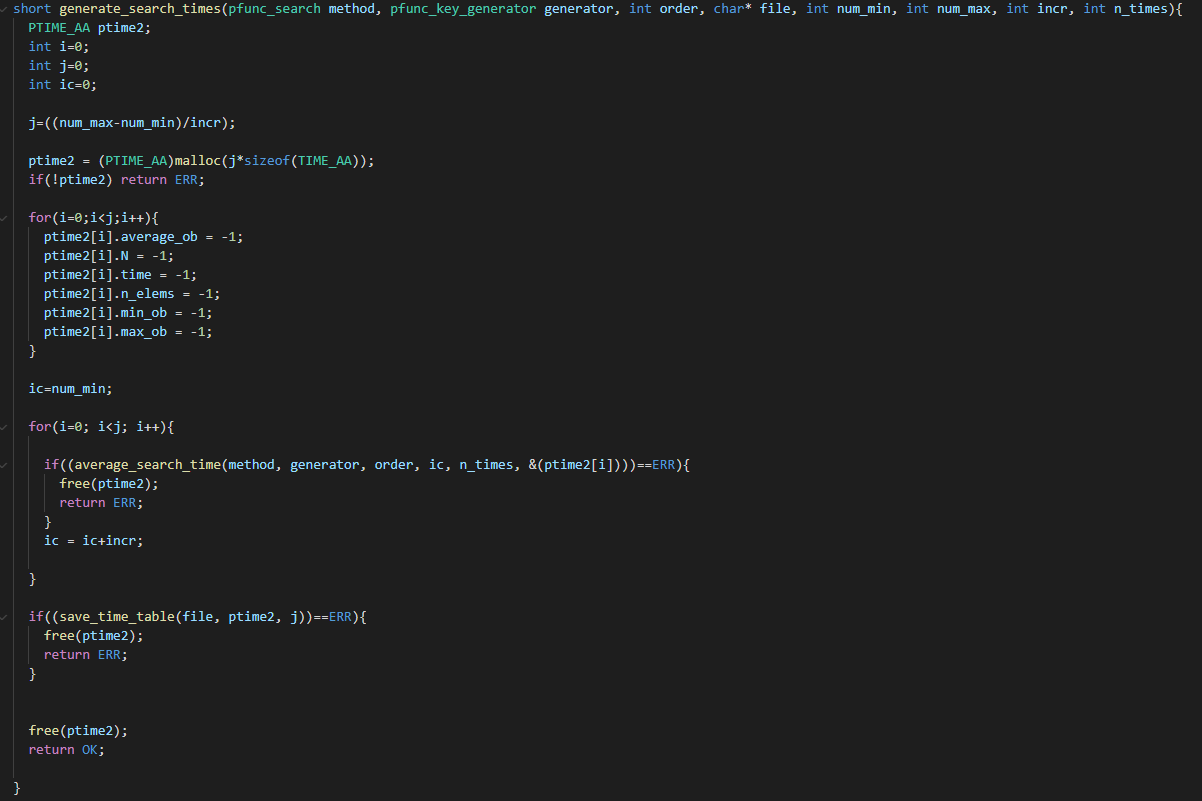






**Exercise 2**

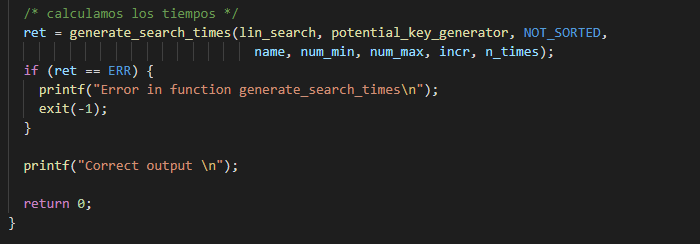


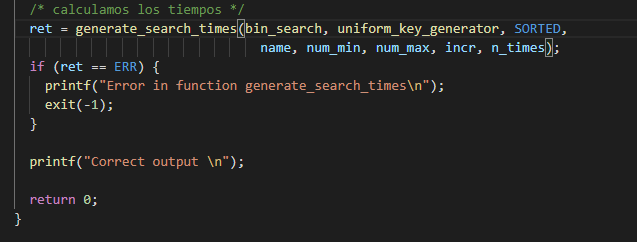


In the next two sections we have only changed line 72 in file exercise2.c

**Exercise 2.1.**

To compare both searching methods, we have to send the following arguments to the function generate\_search\_times():





And we have to use these two commands in the terminal (first to generate the logfile for binary search and the last for linear search):

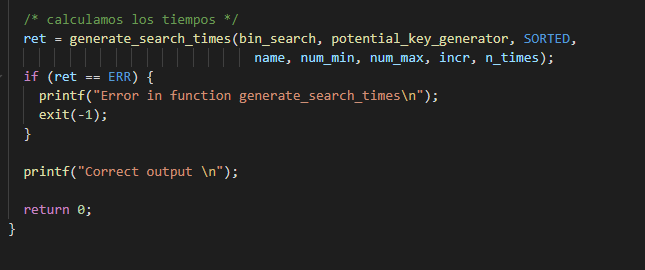


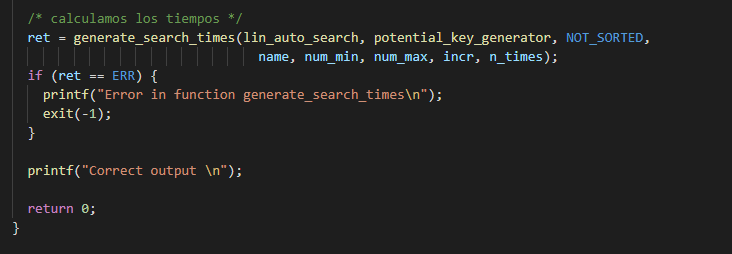


This way we are generating two logs files, one for the linear search and another one for the binary search in where we store the information measured.

**Exercise 2.2.**

To compare both searching methods, we have to send the following arguments to the function generate\_search\_times():





To get all the log files with the different n times parameters and for both functions, we need to use the following commands (first three to generate the logfiles for binary search and last three for linear auto search):









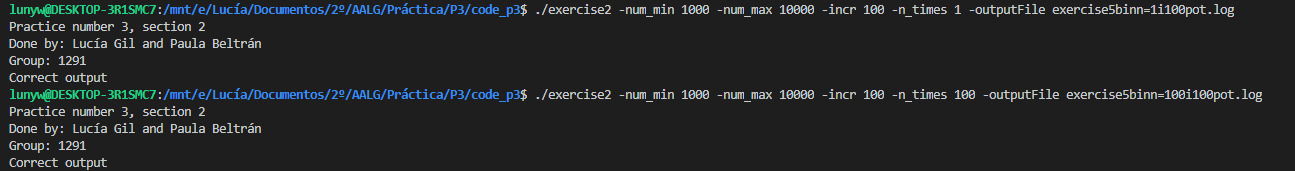


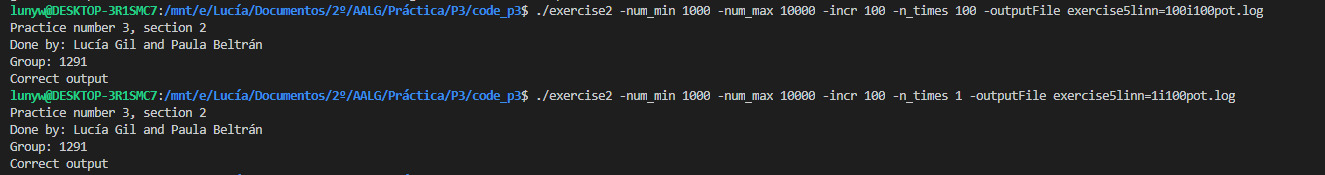


**RESULTS**

**Exercise 2:**

When we execute the first command in exercise 2.1. we get, in the terminal the following outputs and logfiles (in the logfiles, 1st column corresponds with size N, 2nd with Average clock time, 3rd with Average ob, 4th with Maximum ob and 5th with minimum ob):





When we execute the first command in exercise 2.2. we get, in the terminal the following outputs and logfiles (in the logfiles, 1st column corresponds with size N, 2nd with Average clock time, 3rd with Average ob, 4th with Maximum ob and 5th with minimum ob):

First command is the same as the first command of exercise 2.1. so, it is not necessary to include it again.

