Analysis of Algorithms
Escuela Politécnica Superior, UAM, 2019–2020

Set of practices n. 3

Date of delivery

- Thursday groups: December 12th.
- Friday groups: December 13th.

In this practice, the development and analysis of search algorithms over dictionaries is studied. Specifically, the average time to search elements over a dictionary which uses as data type a table will be experimentally obtained.

1 Dictionary ADT

Creation of a dictionary abstract data type (ADT) defined in **search.h** using the following structures:

```
#define NOT_FOUND -2

#define SORTED 1
#define NOT_SORTED 0

typedef struct dictionary {
  int size;  /* size of the table */
  int n_data;  /* number of data in the table */
  char order;   /* sorted table (SORTED) or unsorted (NOT_SORTED) */
  int *table;   /* tabla de datos
} DICT, *PDICT
```

The **DICT** structure will be used to implement a dictionary which uses a (sorted or unsorted) table as data structure. Using that structure, implement the following routines in the **search.c** file:

```
PDICT init_dictionary (int size,char order);
void free_dictionary(PDICT pdict);
int insert_dictionary(PDICT pdict,int key);
int massive_insertion_dictionary (PDICT pdict,int *keys,int n_keys);
int search_dictionary(PDICT pdict,int key,int *ppos,pfunc_search method);

int bin_search(int *table,int F,int L,int key,int *ppos);
int lin_search(int *table,int F,int L,int key,int *ppos);
int lin_auto_search(int *table,int F,int L, int key,int *ppos);
```

where **pfunc_search** is a pointer to a search function, defined as:

```
  typedef int (* pfunc_search)(int*, int,int,int,int*);
```

a prototype followed by the search functions **bin_search**, **lin_search** and **lin_auto_search** declared in **search.h** file.

- The **init_dictionary** function creates an empty dictionary (without any data) of the type indicated by its parameters, **size** indicates the initial size we want the dictionary to have and **order** indicates, using the **SORTED** and **NOT_SORTED** constants, if the data structure used to define the dictionary must be a sorted or an unsorted list. The method must allocate memory for a table with, at least, **size** entries.

- The **insert_dictionary** function introduces the **key** element in the adequate position of the dictionary defined by **pdict**, depending on the value of the dictionary's **order** field. If that field takes **NOT_SORTED** as value, then, the routine will simply place the element at the end of the table.

  However, if the field **order** takes the **SORTED** value, the routine inserts the element in the position which maintains the table sorted in one single iteration of the insertion method, i.e. it adds the element at the end of the table, and, then, changes its position to its proper place using the following piece of code:

```
A=T[L];
j=L-1;
while (j >= F && T[j]>A);
   T[j+1]=T[j];
   j--;
T[j+1]=A;
```

  Note: it is not necessary to call any sorting algorithm; just, add the previous code to your **insert_dictionary** routine to manage the case where the table has to be sorted.

- The **massive_insertion_dictionary** function introduces the **n_keys** keys in the **keys** table in the dictionary, via doing **n_keys** successive calls (one per key) to the **insert_dictionary** function.

- The **search_dictionary** looks for a key in the dictionary defined by **pdict** using the routine indicated by the **method** parameter. The function will return the position of the key in the table storing it in the memory position pointed by **ppos**.

- The **free_dictionary** frees all the memory which has been allocated by the calls to the different dictionary ADT routines.

- The **bin_search**, **lin_search** y **lin_auto_search** routines implement, respectively, the binary search, linear search and auto-organized linear search algorithms. For the auto-organized linear search, when a key is found, its position in the table is swapped with the previous one (except for the case where the found key was already in the first position of the table). By storing it in the memory position pointed by **ppos**, the functions will return the position of the key in the table or the **NOT_FOUND** constant in case the key is not in the table.

All the previous functions shall return the number of Basic Operations which have needed or **ERR**, depending on whether their execution has proceeded as desired or not, except for **init_dictionary**, which will just return **NULL** if an error has happened or the built dictionary if everything went smoothly.
The C program for testing this methods is exercise1.c, which is included in the **code_p3.zip** file.
Modify the program **exercise1** to allow checking the behaviour of the binary and linear search algorithms over a sorted table.

2 Comparison of the search efficiency

In this section, several functions oriented to measure the efficiency of the different search functions developed in the previous one must be implemented. For that, you have to declare in the **times.h** and add to the **times.c** file the function:

```
short average_search_time(pfunc_busqueda method, pfunc_key_generator generator,
                 char order,
                 int N,
                 int n_times,
                 PTIME_AA ptime);
```

where:

- **order** indicates if the dictionary ADT uses sorted tables.

- **N** indicates the size of the dictionary, i.e. the number of elements it contains.

- **n_times** represents how many times each one of the N keys in the dictionary is searched.

- **ptime** is a pointer to a structure of type **TIME_AA** that, after executing the function, contains:

- The size of the dictionary (i.e the **N** argument) in the **N** field.
- The number of searched keys (i.e **N*n_times**) in the **n_elems** field.
- The average execution time (in seconds) in the **time** field.
- The average number of times that the BO was executed in the **avg_ob** field.
- The minimum number of times that the BO was executed **min_ob**.
- The maximum number of times that the BO was executed **max_ob**.

In addition, the routine receives a pointer to the search function (**method** parameter), and another pointer to the function which must be used to generate the keys to look for in the dictionaries (**generator** parameter), which is already implemented in the given **search.c** and **search.h** files. The routine **average_search_time** returns **ERR** if an error occurs and **OK** in case the searches are done correctly.

In order to help, the steps that the **average_search_time** function are described next:

1. Create a dictionary of size **N**.
2. Create a permutation of size **N** using the **generate_perm** routine.
3. Using the **massive_insertion_dictionary**, insert the elements in the permutation into the dictionary.
4. Allocate memory for the table containing the **n_times*N** keys to search in the 1 to N range.
5. Fill the previous table with the **n_times*N** keys to search using the key generator.(**Note: the key generators create numbers from 1 to N, so it is important that permutations also contain numbers from 1 to N**
6. Measure the time (clock y BOs) that the search method takes to search the **n_times*N** keys stored in the previously mentioned table.
7. Properly fill the fields of the **ptime** structure.
8. Free the allocated memory and exit the function.

In addition, the function:

```
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);
```

which automatizes the time measurement must be implemented. This function calls the previous one with a dictionary size which comes from **num_min** to **num_max** (both included), using increments of size **incr**. The routine returns **ERR** if an error occurs, and **OK** otherwise. It must also store the results in the **file** file using the **save_time_table(char* file, PTIME_AA time, int N)** function previously implemented. The rest of parameters are equivalent to the previous routine.
With this time measurement routines, the following tests must be done:

1. Compare the average clock time and average number of basic operations of the linear and binary search. Linear search approach must be applied over unsorted dictionaries, whereas binary search must be applied over sorted dictionaries. Every key in the dictionary must be searched once (i.e. n_times=1) and the key generation function must be the **uniform_key_generator** function included in search.h and search.c **The comparison must be done, at least, for dictionary sizes from 1000 to 10000 elements**.

2. However, the keys to search do not always follow a uniform distribution. Some times, some of the keys are generated more frequently than others. Under these conditions, the average extraction time can be improved if we facilitate finding the most common keys in a time close to O(1). A way to do this is using auto-organized linear search, which uses the keys to search to move the most common ones to the beginning of the list. To ensure that the auto-organized search is effective, lots of keys must be searched to ensure that the keys are optimally ordered.

   Auto-organized lists are particularly useful when the data queries follow rules like the 80/20 one: the 80% of the queries ask for the 20% of the keys, or when the frequency of the data follow potential distributions as the Zipf one, as it happens with the distribution of words in a text: there are a few words with large frequencies, and lots of words with small frequencies. This could also happen with the links of a web page.

   To test this method, the key generation function **potential_key_generator** must be used. With this key generator, you must 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.

The C program to test this routines is the exercise2.c one included in the **code_p3.zip** file.

Theoretical questions

1. Which is the basic operation of **lin_search**, **bin_search** and **lin_auto_search**?
2. Give the execution times, in terms of the input size n for the worst $W_{SS}(n)$ and best $B_{SS}(n)$ cases of **bin_search** and **lin_search**. Use the asymptotic notation (O, Θ, o, Ω,etc) as long as you can.
3. When **lin_auto_search** and the given not-uniform key distribution are used, how does it vary the position of the elements in the list of keys as long as the number of searches increases?
4. Which is the average execution time of **lin_auto_search** as a function of the number of elements in the dictionary n for the given not uniform key distribution? Consider that a large number of searches have been conducted and the list is in a quite stable state.
5. Justify as formally as you can the correction (in other words, why it searches well) of the **bin_search** algorithm.

Material to be delivered in each of the sections

Documentation: The documentation will consist of the following sections:

1. **Introduction:** it consists of a technical description of the work to be carried out, what are the objectives theyintend to reach, what input data is required and what data is obtained from the output, as well as any kind of comment about the practice.
2. **Printed code:** the routine code according to the corresponding section. The code also includes the header ofthe routine.
3. **Results:** description of the results obtained, comparative graphics, of the result obtained with the theoretical ones and comments on them.
4. **Questions:** answers to theoretical questions.

All the files necessary to compile the practice and documentation have to be stored in a single compressed file, in zip format, or tgz (tgz represents a tar file compressed with gzip).
Additionally, the practices should be stored in some storage medium (pendrive, hard disk, remote virtual disk, etc)by the student for the day of the practice exam in December.
Warning: The importance of carrying a pendrive is stressedin addition to other storage media such as usb, remote virtual disk, email to own address, etc, since it is not guaranteed that they can be mounted and accessed each and every one of them during the exam, which will mean the grade not pass in practices.
A copy of the source code and the documentation must be delivered using the electronic practice delivery system, as it is indicated next:

Instructions for the delivery of the source code of the practice

The delivery of the source codes corresponding to the practices of the subject AA will be carried out through the Webpage **https://moodle.uam.es/**.