Analysis of Algorithms 2020/2021

Practice 3

S. Xiao Fernández Marín, 1292.

Code	Plots	Documentation	Total

1. Introduction.

In this practice I have created an ADT called Dictionary, based in tables, and then I have found the average time to search elements over this ADT.

2. Objectives

2.1 Section 1

The objective of this section is to create the Dictionary ADT and check its correct functioning.

2.2 Section 2

The objective of this section is the analysis of the efficiency of the different search functions developed in Section 1.

3. Tools and methodology

Principally, the tool I have been using during this practice is Visual Studio Code. This is because I have gotten used to it. I have also executed all the exercises with the Valgrind tool to make sure that there were no memory errors. To create the plots, I have used Gnuplot tool.

3.1 Section 1

For this section, I have guided myself following the instructions they gave us in the script and reading the pseudocode we have in the whiteboards of the subject. I created the functions in the order we were told, and that facilitates the development of the practice. At last, I tested them by using the program they gave us included in exercise1.c. I also executed it with Valgrind to avoid memory errors.

3.2 Section 2

For this other section, I have also followed the order and steps given in the task. We have tested the routines implemented using exercise2.c, and then I have also checked it with Valgrind. In addition, I have used Gnuplot to plot the results and check if they adapt to what was expected.

4. Source code

4.1 Section 1

```
/*************************************************/
/* Function: init_dictionary    Date: 28/11/2020   */
/* Authors: S. Xiao Fdez                           */
/*                                                 */
/* Function that creates an empty dictionary of the*/
/* type indicated by its parameters.               */
/*                                                 */
/* Input:                                          */
/* int size: initial size we want the dictionary   */
/* to have                                         */
/* char order: indicates if the list is SORTED or  */
/* NOT_SORTED                                      */
/*                                                 */
/* Output:                                         */
/* PDICT: the new initialised dictionary           */
/*************************************************/
PDICT init_dictionary (int size, char order){

  PDICT new_dictionary = NULL;

  if (size < 0 || (order != NOT_SORTED && order != SORTED)) return NULL;

  new_dictionary = (PDICT) malloc (sizeof(new_dictionary[0]));
  if (new_dictionary == NULL) return NULL;

  new_dictionary->table = NULL;
  new_dictionary->table = (int*)calloc(size, sizeof(new_dictionary->table[0]));
  if (new_dictionary->table == NULL) {
    free_dictionary(new_dictionary);
    return NULL;
  }

  new_dictionary->size = size;
  new_dictionary->n_data = 0;
  new_dictionary->order = order;

  return new_dictionary;
}


/*************************************************/
/* Function: free_dictionary     Date: 28/11/2020  */
/* Authors: S. Xiao Fdez                           */
/*                                                 */
/* Function that frees all the memory allocated of */
/* the dictionary                                  */
/*                                                 */
/* Input:                                          */
/* PDICT pdict: pointer to the struct dictionary   */
/*************************************************/
void free_dictionary(PDICT pdict){
  if (pdict == NULL) return;
  if (pdict->table){
    free(pdict->table);
    pdict->table = NULL;
  }
  free(pdict);
  pdict = NULL;
}
```

```
/**************************************************/
/* Function: insert_dictionary   Date: 28/11/2020  */
/* Authors: S. Xiao Fdez                          */
/*                                                */
/* Function that introduces the key element in the */
/* adequate position of the dictionary            */
/*                                                */
/* Input:                                         */
/* PDICT pdict: pointer to the struct of the      */
/* dictionary                                     */
/* int key: the element we want to search for in  */
/* the dictionary                                 */
/*                                                */
/* Output:                                        */
/* int: Returns the number of basic operations    */
/* if everything work correctly and if not,       */
/* it returns ERROR                               */
/**************************************************/
int insert_dictionary(PDICT pdict, int key){

  int i = 0, ob = 0, A = 0;

  if (pdict == NULL || key < 0 || pdict->n_data < 0|| pdict->n_data >= pdict->size)
return ERR;

  if(pdict->order == NOT_SORTED){
    pdict->table[pdict->n_data] = key;
    pdict->n_data++;

  }else if (pdict->order == SORTED){
    pdict->table[pdict->n_data] = key;

    A = pdict->table[pdict->n_data];
    i = pdict->n_data - 1;

    while (i >= 0 && pdict->table[i] > A){
      pdict->table[i + 1] = pdict->table[i];
      ob++;
      i--;
    }
    if (i >= 0) ob++;
    pdict->table[i + 1] = A;
    pdict->n_data++;
  }
  return ob;
}
```

```
/**************************************************************/
/* Function: massive_insertion_dictionary  Date: 28/11/2020 */
/* Authors: S. Xiao Fdez                                      */
/*                                                            */
/* Function that introduces the keys in the keys table in     */
/* the dictionary                                             */
/*                                                            */
/* Input:                                                     */
/* PDICT pdict: pointer to the struct of the dictionary       */
/* int *keys: keys to be inserted                             */
/* int n_keys: number of keys to be inserted                  */
/*                                                            */
/* Output:                                                    */
/* int: Returns the number of BO's if everything worked       */
/* correctly or in the contrary, ERROR                        */
/**************************************************************/
int massive_insertion_dictionary (PDICT pdict,int *keys, int n_keys){

  int i = 0, ob = 0, aux = 0;

  if (pdict == NULL || keys == NULL || n_keys < 0) return ERR;

  for (i = 0; i < n_keys; i++){
    aux = insert_dictionary(pdict, keys[i]);
    if (aux == ERR){
      return ERR;
    }
    ob += aux;
  }
  return ob;
}


/**************************************************/
/* Function: search_dictionary    Date: 28/11/2020 */
/* Authors: S. Xiao Fdez                          */
/*                                                */
/* Function that looks for a key in the dictionary */
/*                                                */
/* Input:                                         */
/* PDICT pdict: pointer to the struct of the      */
/* dictionary                                     */
/* int key: key we want to look for               */
/* int *ppos: memory position where the key is    */
/* stored                                         */
/* pfunc_search method: pointer to a search       */
/* function                                       */
/*                                                */
/* Output:                                        */
/* int: Returns the number of BO's if everything  */
/* worked correct, or in the contrary, ERROR      */
/**************************************************/
int search_dictionary(PDICT pdict, int key, int *ppos, pfunc_search method){

  int ob = 0;

  if (pdict == NULL || key < 0 || ppos == NULL || method == NULL) return ERR;

  ob = method(pdict->table, 0, (pdict->n_data) - 1, key, ppos);
  if (ob == ERR) return ERR;

  return ob;
}
```

```
/* Search functions of the Dictionary ADT */

/*************************************************/
/* Function: bin_search           Date: 28/11/2020 */
/* Authors: S. Xiao Fdez                          */
/*                                                */
/* Function that looks for a key in the dictionary */
/* using the binary search algorithm              */
/*                                                */
/* Input:                                         */
/* int *table: pointer to the data table          */
/* dictionary                                     */
/* int F: first element of the table              */
/* int L: last element  of the table              */
/* int key: key we want to look for               */
/* int *ppos: memory position where the key is    */
/* stored                                         */
/*                                                */
/* Output:                                        */
/* int: Returns the number of BO's if everything  */
/* worked correct, NOT_FOUND if the key is not in */
/* the table                                      */
/*************************************************/
int bin_search(int *table,int F,int L,int key, int *ppos){

  int ob = 0, mid=0, start = F, end = L, result = 0;

  if (table == NULL || F > L || F < 0 || L < 0 || key < 0 || ppos == NULL) return
ERR;

  while (start <= end){
    mid = (end + start) / 2;
    ob++;

    result = table[mid] - key;

    if(result == 0){
      *ppos = mid;
      return ob;
    }

    if (result < 0){
      start = mid + 1;
    }else{
      end = mid - 1;
    }
  }
  *ppos=NOT_FOUND;
  return NOT_FOUND;
}
```

```
/*************************************************/
/* Function: lin_search   Date: 28/11/2020       */
/* Authors: S. Xiao Fdez                         */
/*                                               */
/* Function that looks for a key in the dictionary */
/* using the linear search algorithm             */
/*                                               */
/* Input:                                        */
/* int *table: pointer to the data table         */
/* dictionary                                    */
/* int F: first element of the table             */
/* int L: last element  of the table             */
/* int key: key we want to look for              */
/* int *ppos: memory position where the key is   */
/* stored                                        */
/*                                               */
/* Output:                                       */
/* int: Returns the number of BO's if everything */
/* worked correct, NOT_FOUND if the key is not in */
/* the table                                     */
/*************************************************/
int lin_search(int *table,int F,int L,int key, int *ppos){

  int i = 0, ob = 0;

  if (table == NULL || F > L || F < 0 || L < 0 || key < 0 || ppos == NULL) return
ERR;

  for (i = F; i <= L; i++){
    ob++;
    if (table[i] == key){
      *ppos = i;
      return ob;
    }
  }

  *ppos=NOT_FOUND;
  return NOT_FOUND;
}
```

```
/************************************************/
/* Function: lin_auto_search   Date: 28/11/2020    */
/* Authors: S. Xiao Fdez                        */
/*                                              */
/* Function that looks for a key in the dictionary */
/* using the auto-organized linear search algorithm*/
/*                                              */
/* Input:                                       */
/* int *table: pointer to the data table        */
/* dictionary                                   */
/* int F: first element of the table            */
/* int L: last element  of the table            */
/* int key: key we want to look for             */
/* int *ppos: memory position where the key is  */
/* stored                                       */
/*                                              */
/* Output:                                      */
/* int: Returns the number of BO's if everything   */
/* worked correct, NOT_FOUND if the key is not in  */
/* the table                                    */
/************************************************/
   int lin_auto_search(int *table,int F,int L,int key, int *ppos){

     int i = 0, ob = 0;

     if (table == NULL || F > L || F < 0 || L < 0 || key < 0 || ppos == NULL)
   return ERR;

     ob++;

     if (table[F] == key){
       *ppos = F;
       return ob;
     }

     for (i = F+1; i <= L; i++){
       ob++;
       if (table[i] == key){
         if(i > 0){
           *ppos = i - 1;
           if(Swap(&table[i], &table[i-1]) == ERR) return ERR;
           return ob;
         }
       }
     }

     *ppos=NOT_FOUND;
     return NOT_FOUND;
   }
```

4.2 Section 2

```
/*****************************************************************/
/* Function: generate_search_times        Date: 9/12/2020         */
/* Authors: S. Xiao Fdez                                          */
/*                                                                */
/* Function that generates search times from min to max using     */
/* increments of size incr                                        */
/*                                                                */
/* Input:                                                         */
/* pfunc_search method: pointer to a search function              */
/* pfunc_key_generator: pointer to a function that generates keys */
/* char order: indicates if the dictionary uses sorted tables     */
/* char* file: name of the file where the results are stored      */
/* int num_min: minimum size of permutations                      */
/* int num_max: maximum size of permutations                      */
/* int incr: increment in size                                    */
/* int n_times: number of times each key in the dictionary is     */
/* searched                                                       */
/*                                                                */
/* Output:                                                        */
/* returns ERR in case of error and OK in case the tables are     */
/* ordered correctly                                              */
/*****************************************************************/
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){

  int perm = 0, i = 0, min = 0;
  short  control = 0;
  PTIME_AA ptime = NULL;

  if (method == NULL || generator == NULL || (order != SORTED && order!=
NOT_SORTED) || file == NULL || num_max < num_min || incr <= 0 || n_times < 0)
return ERR;

  perm = (((num_max - num_min) / incr) + 1);

  ptime = (PTIME_AA) malloc (perm * sizeof(ptime[0]));
  if (ptime == NULL) return ERR;

  min = num_min;

  for (i = 0; min <= num_max; i++, min +=incr) {
   control = average_search_time(method, generator, order, min, n_times,
&ptime[i]);
    if (control == ERR) {
      free(ptime);
      ptime = NULL;
      return ERR;
    }
  }

  if (save_time_table (file, ptime, perm) == ERR) {
    free(ptime);
    ptime = NULL;
    return ERR;
  }

  free(ptime);
  ptime = NULL;

  return OK;
}
```

```
/**************************************************************/
/* Function: average_search_time        Date: 9/12/2020       */
/* Authors: S. Xiao Fdez                                      */
/*                                                            */
/* Function that calculates the average time the search algorithm */
/* takes                                                      */
/*                                                            */
/* Input:                                                     */
/* pfunc_search method: pointer to a search function          */
/* pfunc_key_generator: pointer to a function that generates keys */
/* char order: indicates if the dictionary uses sorted tables */
/* int N: indicates the size of the dictionary                */
/* int n_times: number of times each key in the dictionary is */
/* searched                                                   */
/* PTIME_AA ptime: pointer to the structure TIME_AA           */
/*                                                            */
/* Output:                                                    */
/* returns ERR in case of error and OK in case the tables are */
/* ordered correctly                                          */
/**************************************************************/
short average_search_time(pfunc_search metodo, pfunc_key_generator generator, int
order, int N, int n_times, PTIME_AA ptime){

  PDICT dic = NULL;
  int ob = 0, insert = 0, time_ob = 0, i = 0, ppos = 0, aux;
  clock_t time1 = 0, time2 = 0;
  double times = 0;
  int *ele = NULL, *key = NULL;

  if (metodo == NULL || generator == NULL || (order != SORTED && order!=
NOT_SORTED) || N < 1 || n_times < 1 || ptime == NULL) return ERR;

  ptime->average_ob = 0;
  ptime->max_ob = 0;
  ptime->min_ob = 0;
  ptime->N = 0;
  ptime->n_elems = 0;
  ptime->time = 0;

  dic = init_dictionary(N, order);
  if (dic == NULL) return ERR;

  ele = generate_perm(N);
  if (ele == NULL) {
    free_dictionary(dic);
    dic = NULL;
    return ERR;
  }

  insert = massive_insertion_dictionary(dic, ele, N);
  if (insert == ERR) {
    free(ele);
    ele = NULL;
    free_dictionary(dic);
    dic = NULL;
    return ERR;
  }

  aux = N*n_times;

  key = (int*) calloc(aux, (sizeof(key[0])));
  if (key == NULL){
    free(ele);
    free_dictionary(dic);
    return ERR;
  }
```

```
  generator(key, aux, N); /*void, so it does not return anything*/

  time1 = clock();
    if(time1 == (clock_t) -1){
    free(ele);
    free_dictionary(dic);;
    return ERR;
  }

  for (i = 0; i < aux; i++){
    time_ob = search_dictionary(dic, key[i], &ppos, metodo);
    if (time_ob == ERR){
      free(key);
      free(ele);
      free_dictionary(dic);
      return ERR;
    }

    if (ptime->min_ob == 0 || ptime->min_ob > time_ob) ptime->min_ob = time_ob;
    if (ptime->max_ob == 0 || ptime->max_ob < time_ob) ptime->max_ob = time_ob;
    ob+=time_ob;
  }

  time2 = clock();
  if(time2 == (clock_t) -1){
    free(key);
    free(ele);
    free_dictionary(dic);
    return ERR;
  }
  times += ((time2-time1)/(double) CLOCKS_PER_SEC);
  times = times/aux;

  ptime->average_ob = (double) ob/aux;

  ptime->N = N;
  ptime->n_elems = n_times*N;
  ptime->time = times;

  free_dictionary(dic);
  free(key);
  free(ele);

  return OK;
}
```

5. Results, plots

5.1 Section 1

I have executed the different functions of this section with program exercise1.c. I have modified it to check the different routines: lin_search, lin_auto_search and bin_search. For all of them I have also proved that there are no memory errors using Valgrind.

I have also printed the numbers in the screen as they were inserted in the dictionary to be sure they were being inserted okay.

We have used a table of size 100 to check the three of them.

- lin_search:

The first algorithm I have checked is the linear search one. First it looked for key 38 in a table of size 100, and the result is correct, it also tells us the position of the key and the number of basic operations used. We also can see that there are 0 memory errors.

FOUND:

```
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$ valgrind ./exercise1 -size 100 -key 38
==13103== Memcheck, a memory error detector
==13103== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13103== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13103== Command: ./exercise1 -size 100 -key 38
==13103==
Pratice number 3, section 1
Done by: Your names
Group: Your group
50 88 45 14 49 31 29 2 26 35 92 99 25 63 43 73 83 1 36 97 61 90 59 68 72 100 17 79 65 74 69 32 66 9 42 98 53 60 22 33 78 85 67 8 62 87 71 52 5
6 27 37 84 41 51 7 3 28 34 94 38 10 77 4 15 57 70 6 89 48 20 16 75 39 46 19 12 18 21 55 44 81 5 64 93 23 47 96 24 11 95 80 58 91 86 54 40 13 3
0 82 76
Key 38 found in position 37 in 38 basic op.
==13103==
==13103== HEAP SUMMARY:
==13103==     in use at exit: 0 bytes in 0 blocks
==13103==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==13103==
==13103== All heap blocks were freed -- no leaks are possible
==13103==
==13103== For counts of detected and suppressed errors, rerun with: -v
==13103== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$
```

Then, I have entered a key out of range, in this case 111. So, the result is that it is not found in the table of size 100, as expected. Also, we can observe there are 0 memory errors.

NOT FOUND:

```
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$ valgrind ./exercise1 -size 100 -key 111
==13129== Memcheck, a memory error detector
==13129== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13129== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13129== Command: ./exercise1 -size 100 -key 111
==13129==
Pratice number 3, section 1
Done by: Your names
Group: Your group
24 88 90 98 95 86 11 22 74 75 8 9 45 36 70 79 4 17 32 34 12 3 33 71 93 27 23 42 38 89 46 58 81 28 19 78 18 63 30 26 48 97 64 72 56 47 59 57 10
0 15 68 84 65 76 67 29 16 73 21 6 51 52 43 20 7 41 60 53 25 40 50 37 82 69 85 39 94 92 77 5 1 83 61 2 31 80 10 62 54 44 35 87 96 99 55 14 13 4
9 91 66
Key 111 not found in table
==13129==
==13129== HEAP SUMMARY:
==13129==     in use at exit: 0 bytes in 0 blocks
==13129==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==13129==
==13129== All heap blocks were freed -- no leaks are possible
==13129==
==13129== For counts of detected and suppressed errors, rerun with: -v
==13129== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$
```

- lin_auto_search:

The second algorithm to check is the auto-organized linear search. In this case, I have done the same as in the previous one. First, it looked for key 38 in a table of size 100 and the result was satisfactory. It also gives as the position and number of basic operations. And again, we have 0 memory errors.

FOUND:

```
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$ valgrind ./exercise1 -size 100 -key 38
==13377== Memcheck, a memory error detector
==13377== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13377== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13377== Command: ./exercise1 -size 100 -key 38
==13377==
Pratice number 3, section 1
Done by: Your names
Group: Your group
67 71 32 93 26 22 33 24 96 1 31 69 41 13 28 17 34 91 82 29 2 98 51 5 83 21 36 70 63 12 87 14 50 3 39 10 55 60 79 54 99 94 42 61 56 47 64 15 9
40 77 35 37 49 81 4 52 88 76 100 8 44 68 19 23 72 38 58 86 89 97 11 66 43 7 74 75 53 46 59 85 80 73 48 57 6 20 84 95 65 18 16 92 25 90 78 30 6
2 27 45
Key 38 found in position 36 in 38 basic op.
==13377==
==13377== HEAP SUMMARY:
==13377==     in use at exit: 0 bytes in 0 blocks
==13377==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==13377==
==13377== All heap blocks were freed -- no leaks are possible
==13377==
==13377== For counts of detected and suppressed errors, rerun with: -v
==13377== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$
```

Then, I have tried to look for key 111. As the table is of size 100, this key is not found. Again, we have 0 memory errors.

NOT FOUND:

```
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$ valgrind ./exercise1 -size 100 -key 111
==13398== Memcheck, a memory error detector
==13398== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13398== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13398== Command: ./exercise1 -size 100 -key 111
==13398==
Pratice number 3, section 1
Done by: Your names
Group: Your group
20 88 18 66 37 83 41 42 43 92 50 60 79 49 56 73 19 54 46 62 74 11 33 14 23 77 67 9 95 76 24 40 69 1 85 93 52 17 3 26 28 82 5 89 31 8 72 39 48
90 38 44 61 22 59 68 30 35 13 75 47 91 81 94 71 21 34 51 63 99 53 84 45 55 80 64 97 36 32 100 16 27 96 57 7 98 70 4 65 58 15 86 25 2 10 78 12
29 6 87
Key 111 not found in table
==13398==
==13398== HEAP SUMMARY:
==13398==     in use at exit: 0 bytes in 0 blocks
==13398==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==13398==
==13398== All heap blocks were freed -- no leaks are possible
==13398==
==13398== For counts of detected and suppressed errors, rerun with: -v
==13398== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$
```

- bin_search:

The last algorithm to check is the binary search algorithm. I have also done the same as in the previous ones. First, it looked for key 38 in a table of size 100, and the result is that it is found, we also obtained the position and the number of basic operations. We can see we had 0 memory errors.

FOUND

```
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$ valgrind ./exercise1 -size 100 -key 38
==13488== Memcheck, a memory error detector
==13488== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13488== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13488== Command: ./exercise1 -size 100 -key 38
==13488==
Pratice number 3, section 1
Done by: Your names
Group: Your group
74 34 76 86 55 57 7 25 14 89 79 30 38 32 31 37 49 1 26 75 3 62 11 48 35 59 54 67 21 18 88 19 98 72 50 68 44 53 81 8 47 36 65 66 82 77 45 40 22
 15 87 33 43 20 28 12 96 85 52 41 71 92 99 2 46 64 91 83 60 69 90 51 6 56 63 24 5 13 78 9 29 16 97 94 73 70 4 58 42 100 17 61 80 84 39 95 93 2
3 10 27
Key 38 found in position 37 in 6 basic op.
==13488==
==13488== HEAP SUMMARY:
==13488==     in use at exit: 0 bytes in 0 blocks
==13488==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==13488==
==13488== All heap blocks were freed -- no leaks are possible
==13488==
==13488== For counts of detected and suppressed errors, rerun with: -v
==13488== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$
```

In the second case, I also used a table of size 100, but this time I have looked for key 111. As it is out of range, the result is that the key is not found in the table. Again, we have 0 memory errors.

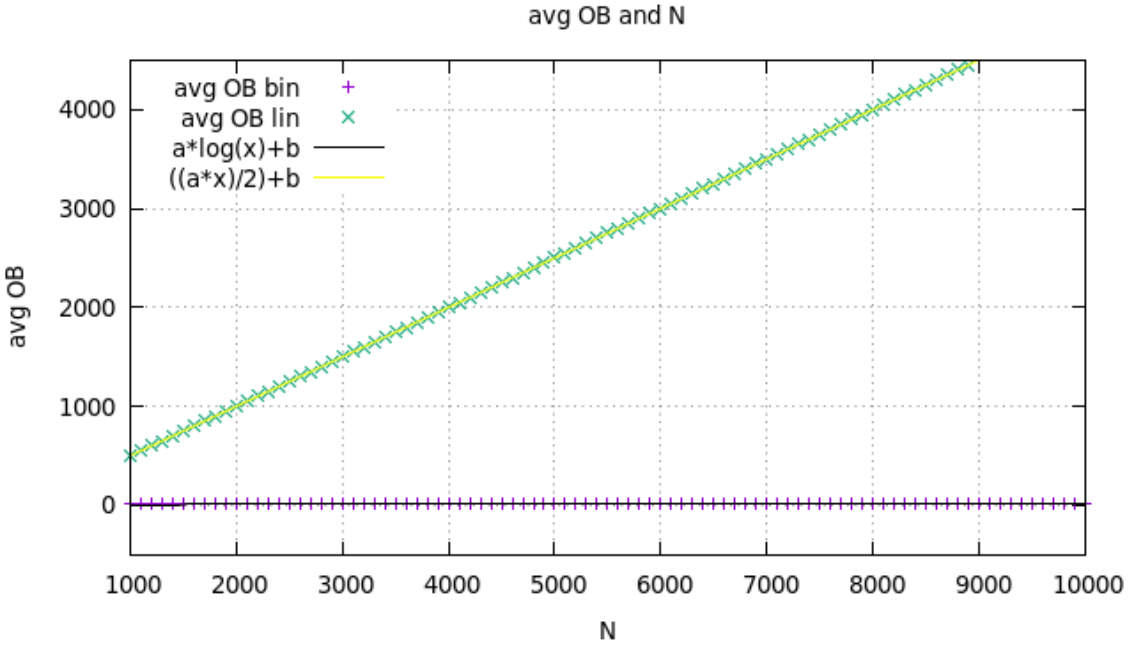
NOT FOUND:

```
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$ valgrind ./exercise1 -size 100 -key 111
==13503== Memcheck, a memory error detector
==13503== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13503== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==13503== Command: ./exercise1 -size 100 -key 111
==13503==
Pratice number 3, section 1
Done by: Your names
Group: Your group
89 82 58 78 65 88 69 50 90 44 39 27 74 59 26 17 98 11 80 32 30 48 96 64 24 87 36 79 10 21 66 38 6 18 92 25 97 71 95 9 34 55 43 4 86 53 19 85 7
 68 94 77 45 91 57 5 84 56 16 51 67 28 14 100 83 60 81 3 62 47 49 29 33 2 37 8 15 73 20 75 70 63 1 93 41 35 61 46 22 52 72 13 31 42 99 76 12 5
4 23 40
Key 111 not found in table
==13503==
==13503== HEAP SUMMARY:
==13503==     in use at exit: 0 bytes in 0 blocks
==13503==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==13503==
==13503== All heap blocks were freed -- no leaks are possible
==13503==
==13503== For counts of detected and suppressed errors, rerun with: -v
==13503== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica3eng$
```

5.2 Section 2

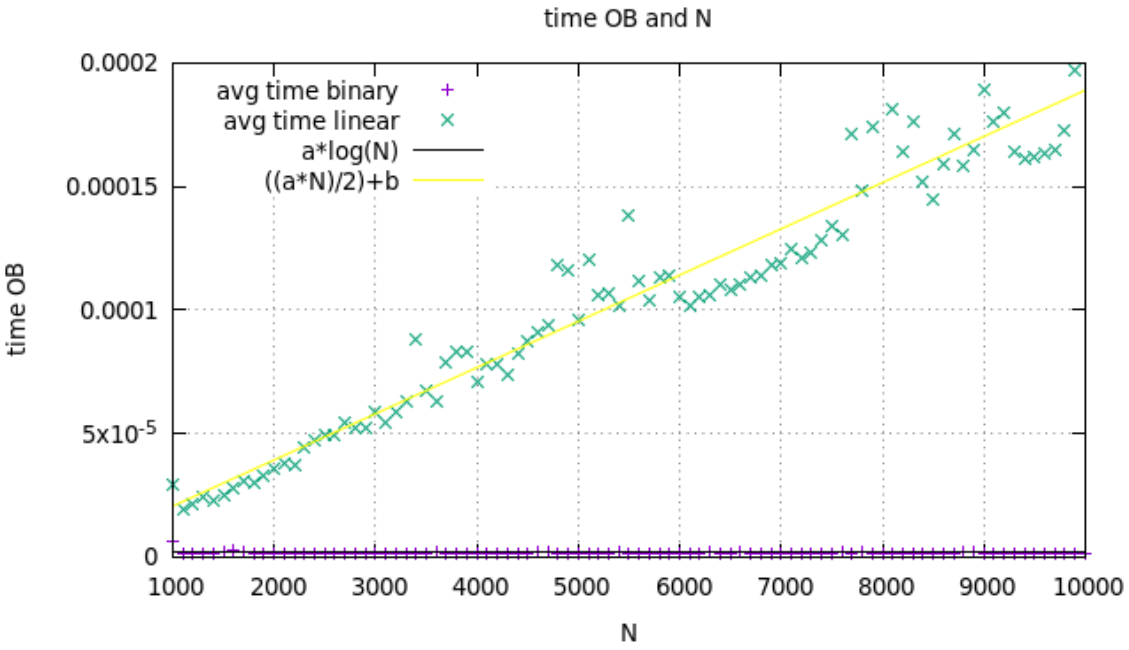
In this section we have used the functions we created to measure the efficiency of the different search algorithms. To compare them, we have created six different graphs with tables of sizes from 1000 to 10000, having in each plot different algorithms to see the difference.

Plot comparing the average number of BOs of linear and binary search approaches.



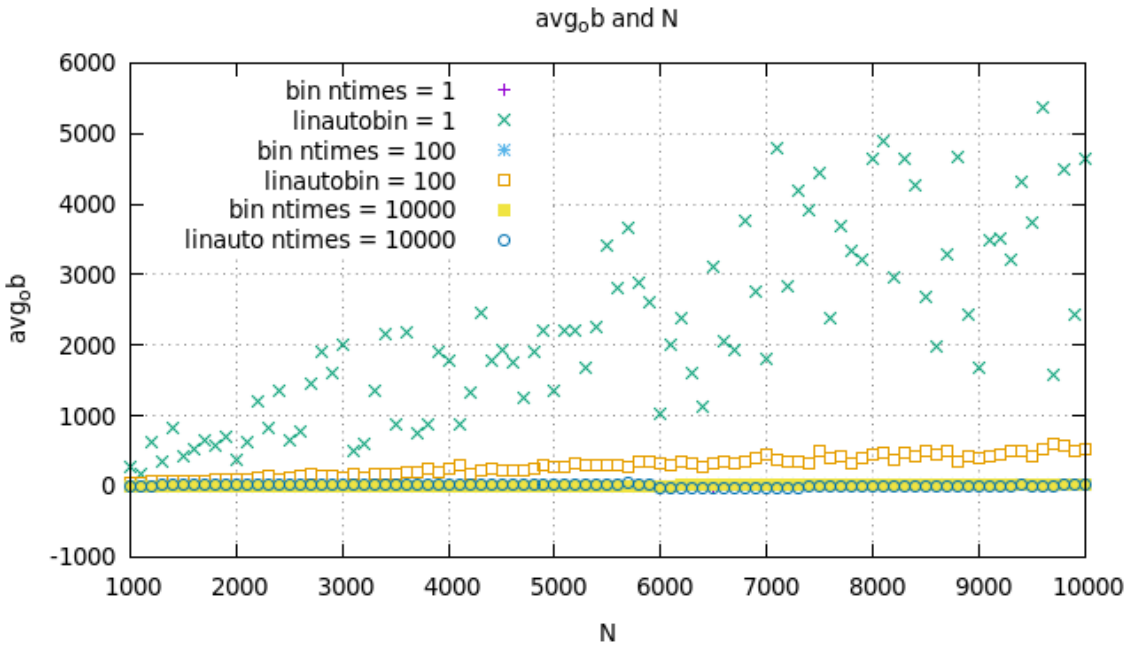
We can see a big difference between the average of basic operations from the binary_search algorithm and the linear search algorithm. This is because the binary_search is a logarithmic equation and the linear_search is a linear equation.

Plot comparing the average clock time for the linear and binary search approaches, comments to the plot.



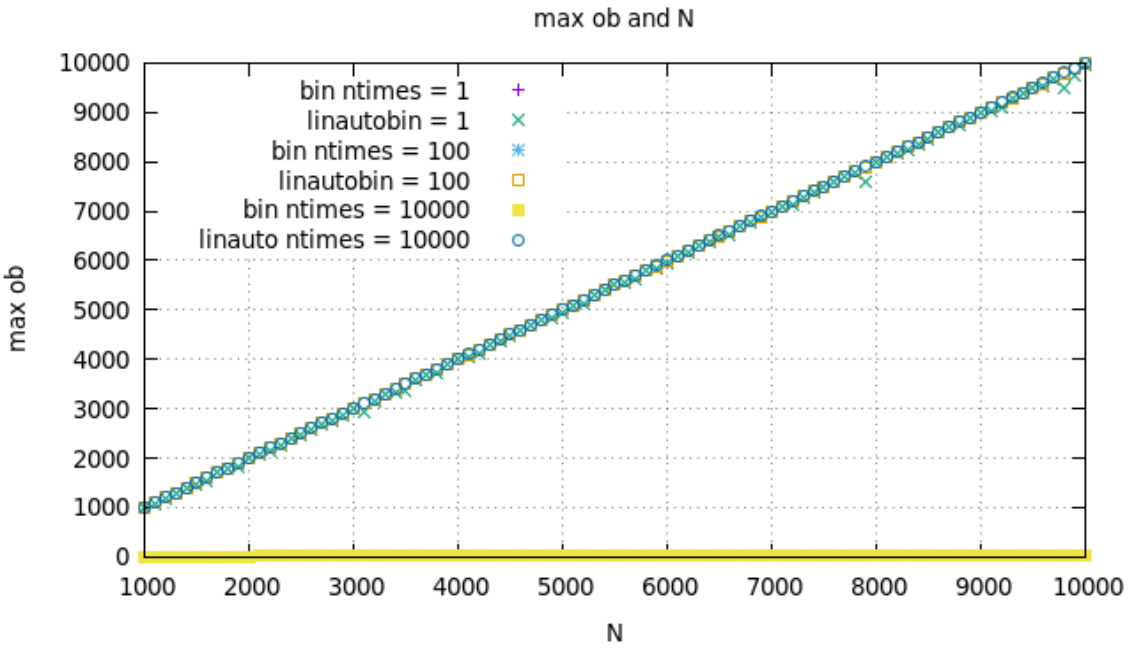
We again can see a big difference between the average time of binary_search algorithm and linear search algorithm. This is because the binary_search is a logarithmic equation and the linear_search is a linear equation.

Plot comparing the average number of BOs of the binary and auto-organized linear search (for n_times=1, 100 y 10000), comments to the plot.



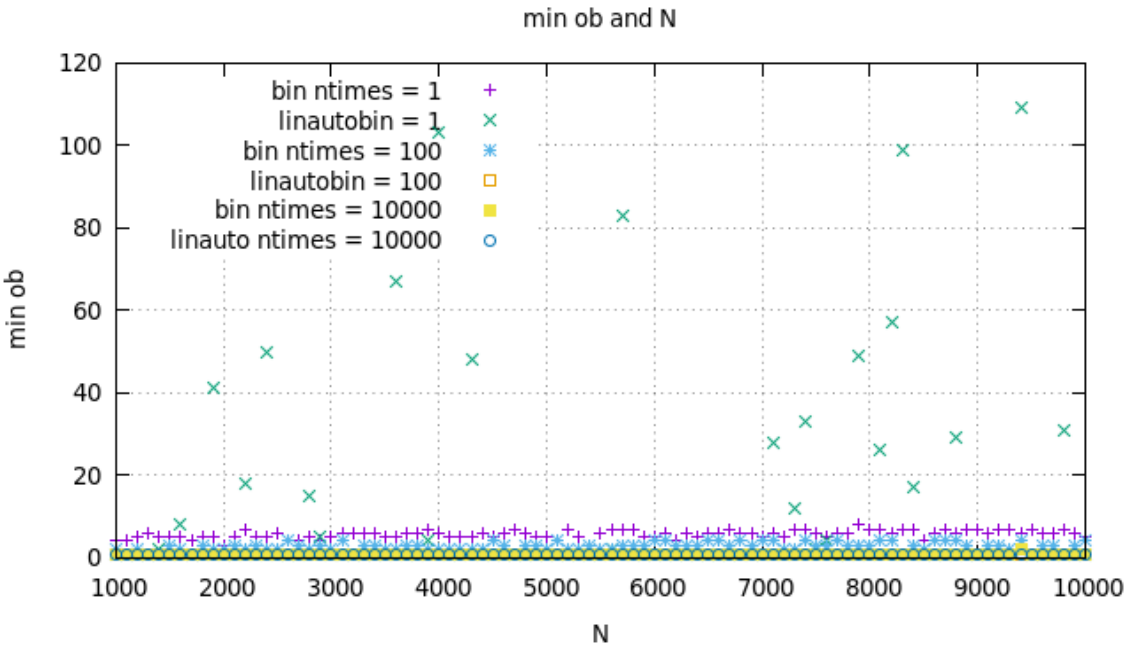
All the basic operations are quite linear, except for the linear-auto search algorithm. That is because this algorithm not only looks for the key, but it also makes a swap when it finds it.

Plot comparing the maximum number of BOs of the binary and auto-organized linear search (for n_times=1, 100 y 10000), comments to the plot.



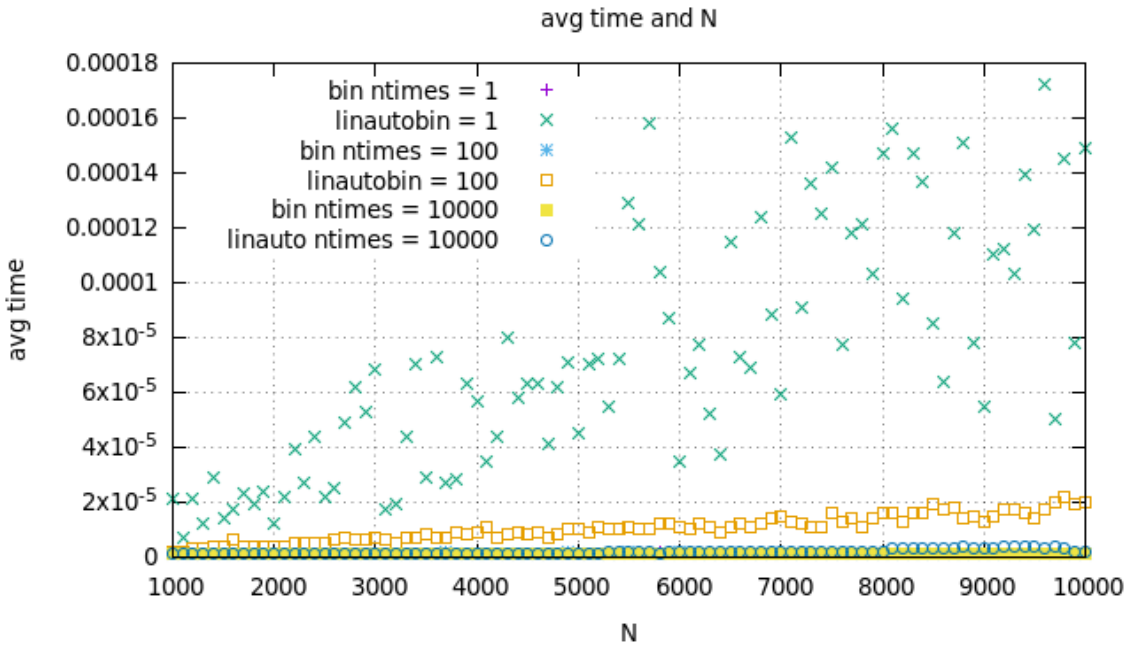
We can see that both algorithms are very similar and linear.

Plot comparing the minimum number of BOs of the binary and auto-organized linear search (for n_times=1, 100 y 10000), comments to the plot.



We can see that the minimum number of BOs for the binary search is almost a straight line with a value of 1. This is the case when the key is in the first position of the table. In the case of auto-organized linear search is not the same, this is because this algorithm, in addition to searching for the key, it also makes a swap with the previous one when it finds it.

Plot comparing the average clock time for the binary and auto-organized linear search (for n_times=1, 100 y 10000), comments to the plot.



We can see that the clock time of all the functions are linear except the linear auto search when n_times = 1, where we can observe that it gets really scattered.

5. Response to the theoretical questions.

5.1. Question 1: **Which is the basic operation of lin search, bin search and lin auto search?**

The basic operation of linear search is the comparison between the key and all the elements of the table, from the first to the last.

The basic operation of binary search is the comparison between the key and the middle position of the table.

The basic operation of linear auto search is the same as in linear search.

5.2. Question 2: **Give the execution times, in terms of the input size n for the worst WSS(n) and best BSS(n) cases of bin search and lin search. Use the asymptotic notation (O, Θ, o, Ω,etc) as long as you can.**

Binary search:

- WSS(n) = O(lg(n))
- BSS(n) = O(1)

Linear search:

- WSS(n) = Θ(n)
- BSS(n) = O(1)

5.3. Question3: **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?**

If we run the program with linear_auto_search and potential_key_generator, the position of the elements in the list of keys will have first the elements with a smaller value. This is because the linear auto search function swaps the found key with the element that has at its left side.

Potential_key_generator generates keys where the smallest values are much more likely to appear than the bigger ones.

Having this both things into account, the smaller elements will appear at the front as long as the number of searches increases.

5.4. Question 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.**

The average execution time of lin_auto_search as a function of the number of elements in the dictionary n for potential_key_generator is approximately (n+1)/2.

This is because after a large number of searches, the list would be sorted and it would be the same as a lin_search with a sorted array.

5.5. Question 5: **Justify as formally as you can the correction (in other words, why it searches well) of the bin search algorithm.**

The bin search algorithm searches a sorted table by repeatedly dividing the search interval in half.

It first begin with an interval covering the whole table. If the value of the key is less than the element in the middle of the interval, it narrows the interval to the lower half. In the opposite case, narrows it to the upper half. It repeatedly checks until the value is found or the interval is empty.

It basically keeps reducing the possibilities to find the key, that is why it searches well.

6. Conclusions.

In this practice we have learn how search algorithms over dictionaries work. We also have obtained the average time to search elements over a dictionary with a table data type. Then we have compared the results of the execution time for the different implemented search algorithms, to see which one of them is more efficient. As a result, we have obtained that the binary search algorithm is the most efficient one.