Analysis of Algorithms 2020/2021

Practice 3

Pablo Almarza and Miguel Arnaiz

G1292

1. Introduction.

In this practice we will create the correspondant functions to run the dictionary data structure. Afterwards we will create three different functions to search for a key and two functions to check the efficiency of the search algorithms we just created.

2. Objectives

2.1 Section 1

In this practice we will create the correspondant functions to run the dictionary data structure. Afterwards we will create three different functions to search for a key and two functions to check the efficiency of the search algorithms we just created.

2.2 Section 2

In this section we implement the two functions in times.c to check the efficiency of the search algorithms implemented in section 1. This check will be done by using a dictionary to search for keys.

3. Tools and methodology

We have used Visual Studio on Linux. Using C/C++ and LiveShare extensions for VS and Valgrind when executing the programs to check errors and memory leaks.

3.1 Section 1

For the dictionary data structure we implemented it following the instructions in the comments on the functions which we had to create. For the search algorithms we followed the explanation of the teacher given in class.

3.2 Section 2

For the two new functions, we based on the functions created on times.c on the first practice as it is the same idea. We just changed it so that it could be implemented with the key generator and the dictionary with the code given.

4. Source code

Here, include the source code only **for the routines you have developed** for each exercise.

4.1 Section 1

```
PDICT init_dictionary (int size, char order)
{

 PDICT pdict=NULL;
 if(size<1 || (order!=SORTED && order!=NOT_SORTED)){return NULL;}
  pdict=(PDICT)malloc(sizeof(DICT));
 if(!pdict){return NULL;}
  pdict->table = NULL;
  pdict->table = (int*)malloc(sizeof(int)*size);
 if(!pdict->table){
   free_dictionary(pdict);
   return NULL;
 }

 pdict->size = size;
 pdict->n_data = 0;
 pdict->order = order;
  return pdict;
}
```

```
void free_dictionary(PDICT pdict)
{
 if(!pdict){return;}
 if(pdict->table!=NULL){
   free(pdict->table);
 }
 free(pdict);
}

int insert_dictionary(PDICT pdict, int key)
{
 int A, j=0, count=0;

 if(!pdict || pdict->size==pdict->n_data) return ERR;

 pdict->table[pdict->n_data]=key;
  if(pdict->order==SORTED && pdict->n_data>0){
   A=pdict->table[pdict->n_data];
   j=pdict->n_data-1;
   while (j >= 0 && pdict->table[j]>A){
     pdict->table[j+1]=pdict->table[j];
     j--;
     count++;
   }
   if(j>=0){
    count++;
  }
  pdict->table[j+1]=A;
 }
 pdict->n_data++;
 return count;
}

int massive_insertion_dictionary (PDICT pdict,int *keys, int n_keys)
{
 int countOB=0, i, aux;

 if(!pdict || !keys || n_keys<1 || (pdict->size - pdict->n_data) < n_keys){return ERR;}
  for(i=0 ; i<n_keys ; i++){
   if((aux = insert_dictionary(pdict, keys[i]))==ERR){return ERR;}
   countOB += aux;
 }
 return countOB;
}

int search_dictionary(PDICT pdict, int key, int *ppos, pfunc_search method)
{
 if(!pdict || !method || !ppos || key<0){return ERR;}

 return method(pdict->table, 0, pdict->n_data-1, key, ppos);
 }

/* Search functions of the Dictionary ADT */
int bin_search(int *table,int F,int L,int key, int *ppos)
{
 int M, count=0;
 if(!table || F>L || F<0) return ERR;


 while(F<=L){
   M=(L+F)/2;
   count++;
   if(table[M]==key){
     *ppos=M;
     return count;
   }
   else if(table[M]>key){
     L=M-1;
   }
   else{
     F=M+1;
   }
```

```c
 }
 *ppos=NOT_FOUND;

 return count;
}


int lin_search(int *table,int F,int L,int key, int *ppos)
{
 int countOB=0, i;
  if(!table || F>L || F<0 || !ppos){return ERR;}
  for(i=F ; i<=L ; i++){
   countOB++;
   if(table[i]==key){
     *ppos=i;
     return countOB;
   }
 }
 *ppos=NOT_FOUND;
 return countOB;
}

int lin_auto_search(int *table,int F,int L,int key, int *ppos)
{
 int countOB=0, i;

 if(!table || F>L || F<0 || !ppos){return ERR;}

 for(i=F ; i<=L ; i++){
   countOB++;
   if(table[i]==key){
     if(i!=F){
       swap(&table[i], &table[i-1]);
       i--;
     }

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

4.2 Section 2

```c
/***************************************************/
/* Function: generate_search_times                 */
/* Date: 10/12/2020                          */
/* Authors: Pablo Almarza and Miguel Arnaiz        */
/***************************************************/
short generate_search_times(pfunc_search method, pfunc_key_generator generator,
                            char order, char* file,
                            int num_min, int num_max,
                            int incr, int n_times){

  PTIME_AA ptime=NULL;
  int n=0, i=0, j=0;
  if(!method || !generator || !file || incr<=0 || num_min>num_max || num_min<0 ||
(order!=NOT_SORTED && order!=SORTED) || n_times<=0){
    return ERR;
  }
  n=((num_max-num_min)/incr)+1;
  ptime=malloc(sizeof(TIME_AA)*n);
  if(!ptime){return ERR;}
  for(i=num_min, j=0; i<=num_max ; i+=incr, j++){
    if(average_search_time(method, generator, order, i, n_times, &ptime[j])==ERR){
      free(ptime);
      return ERR;
    }
  }
  if(save_time_table(file, ptime, n)==ERR){
    free(ptime);
    return ERR;
  }
  free(ptime);
  return OK;
}

/***************************************************/
/* Function: average_search_time                   */
/* Date: 10/12/2020                          */
/* Authors: Pablo Almarza and Miguel Arnaiz        */
/***************************************************/
short average_search_time(pfunc_search method, pfunc_key_generator generator,
                          char order,
                          int N,
                          int n_times,
                          PTIME_AA ptime){

  int i=0, *perm=NULL, *keys=NULL, ppos=0, aux=0, n_ob=0;
```

```
 clock_t begin, end;
 double time;
 PDICT pdict=NULL;

 if(!method|| !generator || !ptime || N<=0 || n_times<0 || (order!=NOT_SORTED &&
order!=SORTED)) return ERR;
 ptime->average_ob=0;
 ptime->max_ob=0;
 ptime->min_ob=0;
 ptime->time=0;
 pdict=init_dictionary(N, order);
 if(!pdict) {return ERR;}
 perm=generate_perm(N);
 if(!perm){
   free_dictionary(pdict);
   return ERR;
 }
 if(massive_insertion_dictionary(pdict, perm, N)==ERR){
   free_dictionary(pdict);
   free(perm);
   return ERR;
 }
 keys=(int*)malloc(sizeof(int)*N*n_times);
 if(!keys){
   free_dictionary(pdict);
   free(perm);
   return ERR;
 }
 generator(keys, n_times*N, N);
 begin=clock();
 for(i=0;i<N*n_times;i++){
   aux=search_dictionary(pdict, keys[i], &ppos, method);
   if(aux==ERR || ppos==NOT_FOUND){
     free_dictionary(pdict);
     free(perm);
     free(keys);
     return ERR;
   }
   if(aux<ptime->min_ob || ptime->min_ob == 0){ptime->min_ob = aux;}
   if(aux>ptime->max_ob || ptime->max_ob == 0){ptime->max_ob = aux;}
   n_ob+=aux;
 }
 end=clock();
 time=(double)(end-begin)/CLOCKS_PER_SEC;
 time=time/(N*n_times);
 ptime->time=time;
 ptime->average_ob=(double)(n_ob)/(N*n_times);
```

```
  ptime->N=N;
  ptime->n_elems=N*n_times;
  free_dictionary(pdict);
  free(perm);
  free(keys);
  return OK;
}
```

5. Results, plots

5.1 Section 1

linear search and not sorted dictionary:

```
./exercise1 -size 20 -key 5

Pratice number 3, section 1

Done by: Pablo Almarza and Miguel Arnaiz

Group: 1292

Key 5 found in position 11 in 12 basic op.
```

linear auto search and not sorted dictionary:

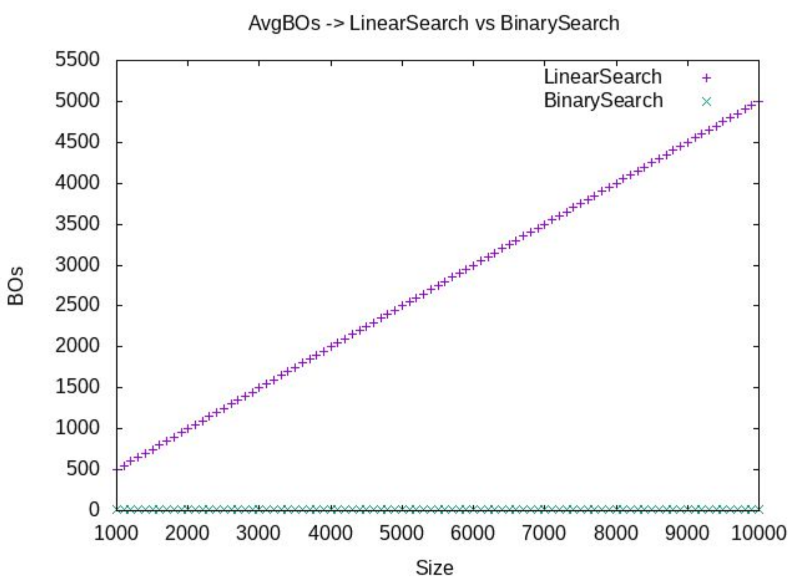
```
./exercise1 -size 20 -key 5

Pratice number 3, section 1

Done by: Pablo Almarza and Miguel Arnaiz

Group: 1292

Key 5 found in position 1 in 3 basic op.
```

binary search and sorted dictionary:

```
./exercise1 -size 20 -key 5

Pratice number 3, section 1

Done by: Pablo Almarza and Miguel Arnaiz

Group: 1292

Key 5 found in position 4 in 2 basic op.
```

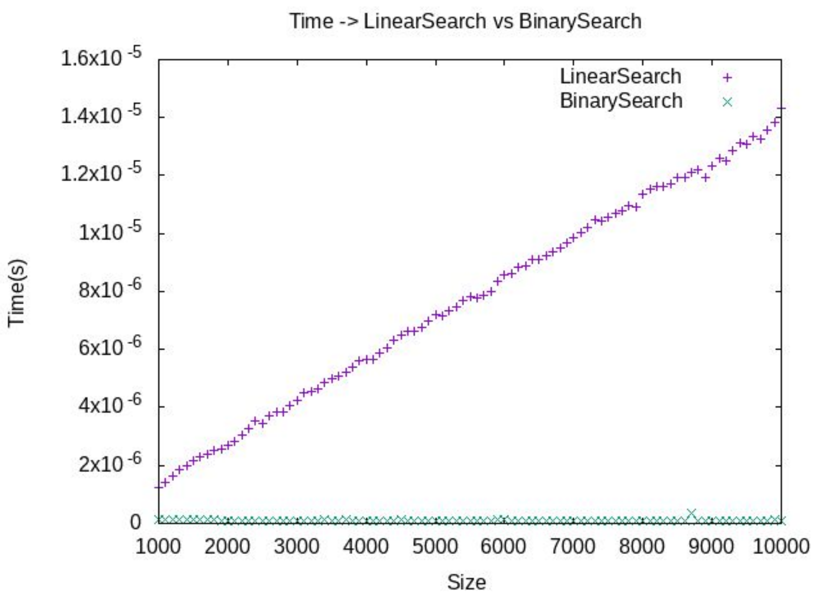
5.2 Section 2

Plot comparing the average number of BOs of linear and binary search approaches, comments to the plot.



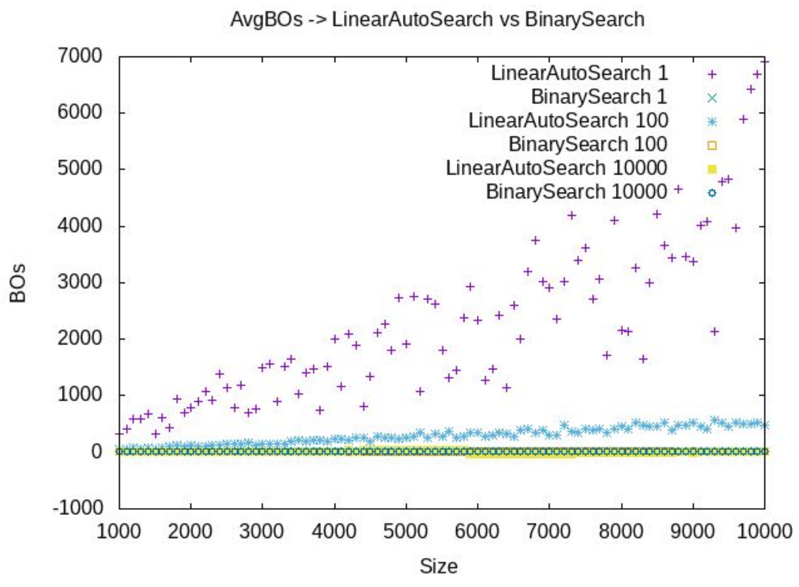
In this plot we can observe how the results for the average number of basic operations are as expected, following the theoretical curves (log(N) for binary search, and N/2 for linear search).

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



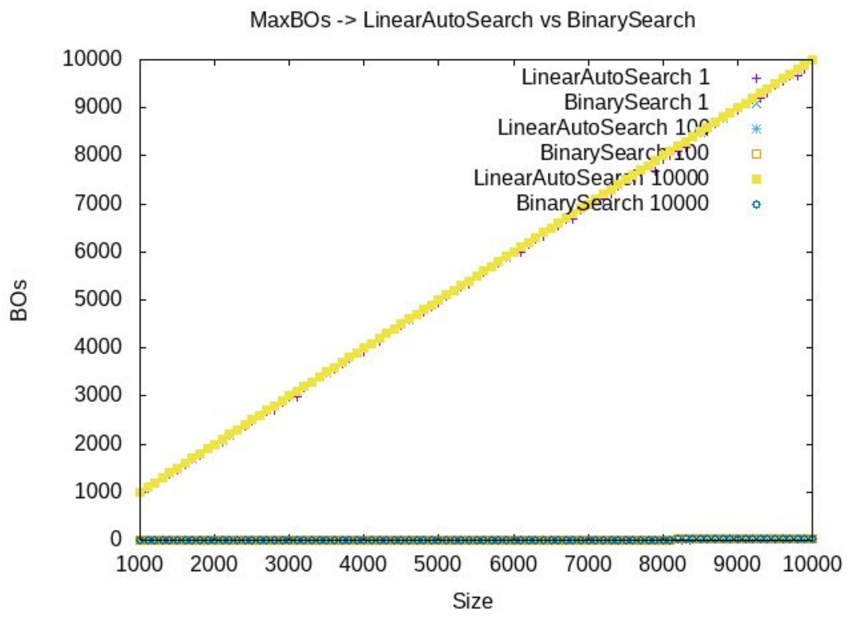
In this plot, similar to the previous one, we observe the average time for the binary and linear searches, and how the results follow the theoretical curves (it looks like a stair because there are no more decimal available so the results are approximated).

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.



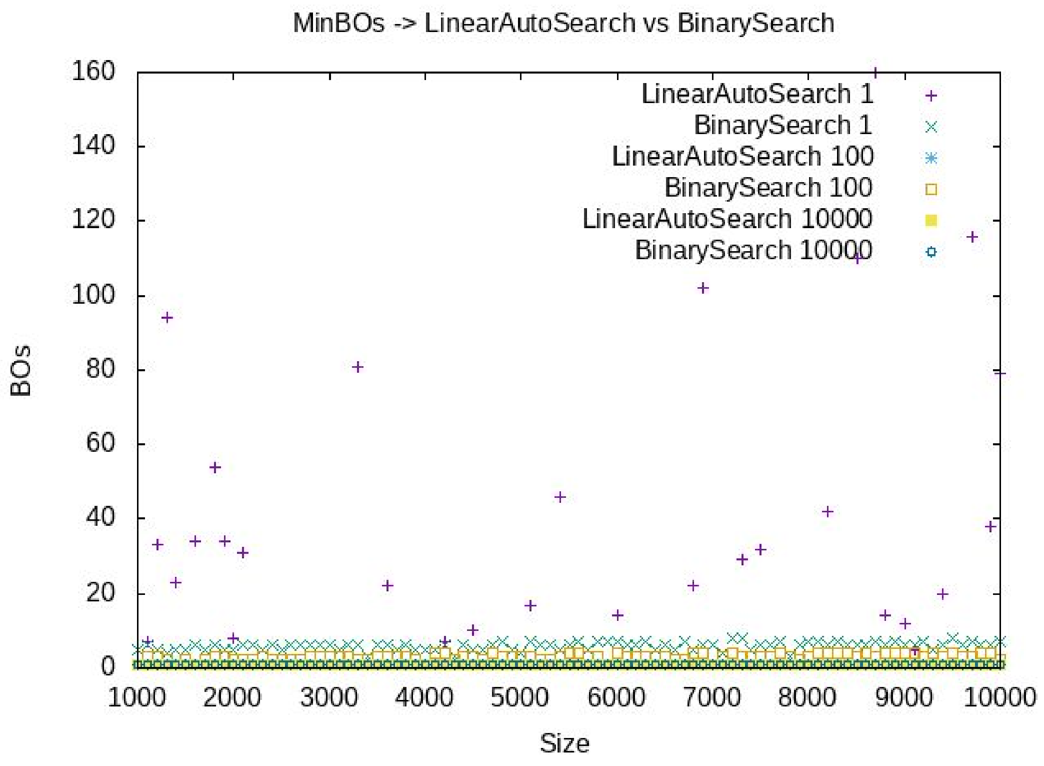
In this plot we see 6 different things, the data for binary and linear auto given different n_times. We observe how maxing n_times makes the avg basic operations reduce, as expected.

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.



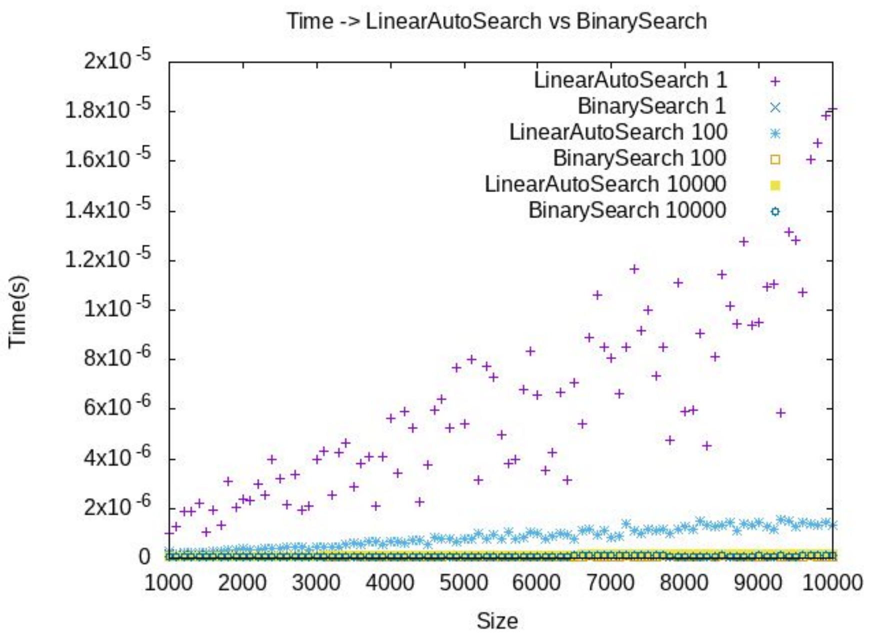
Now we observe the maximum number of BOs in a similar way as before, with a key difference, the maximum number of BOs isn't affected by the n_times variable, also as expected.

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.



In an opposite way we can see how the minimum number of BOs are stabilised with the increase of n_times, decreasing its values.

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.



In this plot, similar to the avg BOs, the time is decreased with the increase of n_times as expected. The plot looks like a stair due to the lack of decimals.

5. Response to the theoretical questions.

5.1 Question 1

In the linear and linear auto search the key comparison is to compare each element of the table from the first to the last element.

In binary search the basic operation is to see if the middle element is equal to the key.

5.2 Question 2

For binary search WSS(n) = $O(\log_2 n)$ and BSS(n) = O(1).
For linear search WSS(n) = n and BSS(n) = O(1).

5.3 Question 3

Using the potential key generator and searching them with linear auto-search, the table will be more or less ordered as low values are much more likely to be generated.

5.4 Question 4

When multiple searches have been done with linear auto search and the potential key generator, the table will be more or less sorted. Since the keys you are going to search are generated by the potential key generated, it is more probable to get the keys that are at the beginning of the table.

As n/2 is the average time for normal linear search n/4 will be the average time when using the potential key generator.

5.5 Question 5

Binary search is a very efficient algorithm. Always dividing the table or array by half every time we make a key comparison can make that a very large table can be done by making few comparisons. For example, with a table of 1024 elements binary search could make 10 comparisons at max, while linear search in the worst case could do 1024 comparisons.

6. Conclusions.

As a conclusion we have learned to implement three new search algorithms pointing out the great efficiency of binary search over linear search algorithms.