Written Report | AED

Teachers:
Tomás Oliveira e Silva
João Manuel Rodrigues

2nd Pratical Work

Hash Table

Hugo Paiva, 93195 - 50%
João Laranjo, 91153 - 50%

DETI
Universidade de Aveiro
29-01-2020

Contents

1 Introduction

The main goal of this second practical work is to implement a Hash Table where it will be stored all the different words present in a text file. In each entry of that Table should be a structure capable of storing:

- The number of occurrences of each distinct word
- The location of the first and last occurrences of each distinct word
- The smallest, largest, and average distances between consecutive occurrences of the same distinct word

As a mean of comparison, the Hash Table will be developed using two different structures for each table entry: a Linked List and a Ordered Binary Tree. By this way, it will be possible to compare times of search and execution between them.

Lastly, it was also asked that the Table should grow dynamically. This means that when the Table is nearly full, it should be resized to a larger size.

1.1 Getting Started

These instructions will help to compile and run developed programs. All the code can be found in a GitHub repository. A clone can be made using the next command, if you have permissions[1]:

```
git clone https://github.com/hugofpaiva/aed_p2
```

1.2 Prerequisites

To compile programs, it is necessary to have a C compiler like cc installed on your local machine.

1.3 Compiling

The following command compiles the program (main.c) where `<executable_filename>` will be the executable filename:

```
cc -Wall -O2 main.c -o <executable_filename> -lm
```

1.4 Running

Options:

```
-l ..... Initialize program using Hash Table with Linked Lists

-b ..... Initialize program using Hash Table with Ordered Binary Trees

-t ..... Initialize program and runs some tests
```

[1] For confidentiality reasons, the repository may be private.

2 Implementation

The following explanations focus on the main components that allow the running of the program. All code can be found in the report appendix.

2.1 Structures

In order to implement both the Linked Lists and the Ordered Binary Trees, two different types of structures capable of storing all the information were needed.

The structures *link_ele* and *tree_node* were used, respectively, as an element of a Linked List and as an element of a Ordered Binary Tree. These structures are able to store:

- The word on this entry
- The number of occurrences of this word
- Total sum distances between this consecutive word (related to the general word counter of the text file)
- Total sum distances between this consecutive word (related to the index position on the text file)
- Minimum distance between this consecutive word (related to the general word counter of the text file)
- Maximum distance between this consecutive word (related to the general word counter of the text file)
- Minimum distance between this consecutive word (related to the index position on the text file)
- Maximum distance between this consecutive word (related to the index position on the text file)
- The last position of this word (related to the general word counter of the text file)
- The first position of this word (related to the general word counter of the text file)
- The last position of this word (related to the index position on the text file)
- The first position of this word (related to the index position on the text file)
- On a *link_ele*:
  - A pointer to next element of the Linked List
- On a *tree_node*:
  - A pointer to left child of this node
  - A pointer to right child of this node
  - A pointer to parent of this node

The structure *file_data* was also used and adapted from a similar one presented in the slides of the theoretical classes.

```
1
2  typedef struct file_data
3  {                  // public data
4      long word_pos; // zero-based
5      long word_num; // zero-based
6      char word[64];
7      // private data
8      FILE *fp;
9      long current_pos; // zero-based
10 } file_data_t;
11
12 typedef struct link_ele
13 {
14     char word[64];
15     long count;            // word counter
16     long tdist;            // total sum of distances (in relation to the general word counter)
17     long tdistp;           // total sum of distances (in relation to the index position)
18     long dmin;             // min distance (in relation to the general word counter)
19     long dmax;             // max distance (in relation to the general word counter)
20     long dminp;            // min distance (in relation to the index position)
21     long dmaxp;            // max distance (in relation to the index position)
22     long last;             // last position (in relation to the general word counter)
23     long first;            // first position (in relation to the general word counter)
24     long lastp;            // last position (in relation to the index position)
25     long firstp;           // first position (in relation to the index position)
26     struct link_ele *next; // next word pointer
27 } link_ele;
28
29 typedef struct tree_node
30 {
31     struct tree_node *left;   // pointer to the left branch (a sub-tree)
32     struct tree_node *right;  // pointer to the right branch (a sub-tree)
33     struct tree_node *parent; // optional
34     char word[64];
35     long count;  // word counter
36     long tdist;  // total sum of distances (in relation to the general word counter)
37     long tdistp; // total sum of distances (in relation to the index position)
38     long dmin;   // min distance (in relation to the general word counter)
39     long dmax;   // max distance (in relation to the general word counter)
40     long dminp;  // min distance (in relation to the index position)
41     long dmaxp;  // max distance (in relation to the index position)
42     long last;   // last position (in relation to the general word counter)
43     long first;  // first position (in relation to the general word counter)
44     long lastp;  // last position (in relation to the index position)
45     long firstp; // first position (in relation to the index position)
46     long data;   // the data item (we use an int here, but it can be anything)
47 } tree_node;
```

2.2 Hash Function

The index of the Hash Table where a given word will be stored is obtained using a Hash Function. The idea is that, using a word and the size of the Hash Table, the function will always return the same index, trying to avoid returning this index when using other words.

This code was adapted from a similar code present in the slides of the theoretical classes.

```
1
2 unsigned int hash_function(const char *str, unsigned int s)
3 { // for 32-bit unsigned integers, s should be smaller that 16777216u
4     unsigned int h;
5     for (h = 0u; *str != '\0'; str++)
6         h = (256u * h + (0xFFu & (unsigned int)*str)) % s;
7     return h;
8 }
```

2.3 Text and Word Processing

To process different text files several functions were developed, such as *open_text_file*, *close_txt_file* and *read_word*.

Like the Hash Function code used before, these functions were developed based on similar functions present in the slides of the theoretical classes.

```
1
2 int open_text_file(char *file_name, file_data_t *fd)
3 {
4     fd->fp = fopen(file_name, "rb");
5
6     if (fd->fp == NULL)
7         return -1;
8     fd->word_pos = 0;
9     fd->word_num = 0;
10    fd->word[0] = '\0';
11    fd->current_pos = -1;
12    return 0;
13 }
14
15 void close_text_file(file_data_t *fd)
16 {
17    fclose(fd->fp);
18    fd->fp = NULL;
19 }
20
21 int read_word(file_data_t *fd)
22 {
23    int i, c;
24    // skip white spaces
25    do
26    {
27        c = fgetc(fd->fp);
28        if (c == EOF)
29            return -1;
30        fd->current_pos++;
31    } while (c <= 32);
32    // record word
33    fd->word_pos = fd->current_pos;
```

```
    fd->word_num++;
    fd->word[0] = (char)c;
    for (i = 1; i < (int)sizeof(fd->word) - 1; i++)
    {
        c = fgetc(fd->fp);
        if (c == EOF)
            break; // end of file
        fd->current_pos++;
        if (c <= 32)
            break; // terminate word
        fd->word[i] = (char)c;
    }
    fd->word[i] = '\0';
    return 0;
}
```

2.4 Add Node to Ordered Binary Tree

This function was developed so that it was possible to add a new node to a Hash Table index, using the Ordered Binary Tree struct, storing all the information needed.

First of all, a hash-code is generated to the word being read from the *file_data_t* struct introduced into the function, in order to know where to store the word information, as it was previously explained on the function *hash_function*.

After that, it is verified if that position contains an element or if it's doesn't (*NULL*). If it doesn't contains, a new node will be created, storing all the information and being the *root* of that index. If it contains, the Ordered Binary Tree is traveled until it finds the node of that word, using the *strcmp* function and updating the information stored, if found. When not found, a new node of that word will be created and added to the proper position of the Tree.

```

void add_node(tree_node **words, file_data_t *f, int size)
{
    int index = hash_function(f->word, size);
    tree_node *actual = words[index];
    if (actual != NULL) // if there is already an element in the ordered binary tree
    {
        if (strcmp(actual->word, f->word) == 0)
        { // if that element is the same
            long tempdist = f->word_num - actual->last;
            long tempdistp = f->current_pos - actual->lastp;
            actual->tdist = actual->tdist + tempdist;
            actual->tdistp = actual->tdistp + tempdistp;
            if (tempdist < actual->dmin)
                actual->dmin = tempdist;
            if (tempdist > actual->dmax)
                actual->dmax = tempdist;
            if (tempdistp < actual->dminp)
                actual->dminp = tempdistp;
            if (tempdistp > actual->dmaxp)
                actual->dmaxp = tempdistp;
            actual->count++;
            actual->last = f->word_num;
            actual->lastp = f->current_pos;
        }
        else
        { // if the element is not the same we travel through the next elements to check if there is any equal
            bool found = false;
```

```
            while (actual != NULL) // While word not found and children not null
            {
                if (strcmp(f->word, actual->word) < 0 && actual->left != NULL) // actual word is smaller
                    actual = actual->left;

                else if (strcmp(f->word, actual->word) > 0 && actual->right != NULL) // actual word is bigger
                    actual = actual->right;

                else if (strcmp(f->word, actual->word) == 0)
                { // if equal
                    long tempdist = f->word_num - actual->last;
                    long tempdistp = f->current_pos - actual->lastp;
                    actual->tdist = actual->tdist + tempdist;
                    actual->tdistp = actual->tdistp + tempdistp;
                    if (tempdist < actual->dmin)
                        actual->dmin = tempdist;
                    if (tempdist > actual->dmax)
                        actual->dmax = tempdist;
                    if (tempdistp < actual->dminp)
                        actual->dminp = tempdistp;
                    if (tempdistp > actual->dmaxp)
                        actual->dmaxp = tempdistp;
                    actual->count++;
                    actual->last = f->word_num;
                    actual->lastp = f->current_pos;
                    found = true;
                    break;
                }
                else
                    break;
            }

            if (!found) // check that no elem was found
            {
                tree_node *temp = malloc(sizeof(tree_node));
                strcpy(temp->word, f->word);
                temp->first = f->word_num;
                temp->count = 1;
                temp->last = f->word_num;
                temp->lastp = f->current_pos;
                temp->firstp = f->word_pos;
                temp->parent = actual;
                temp->dmin = plus_inf;   // dist not altered
                temp->dmax = minus_inf;  // dist not altered
                temp->dminp = plus_inf;  // dist not altered
                temp->dmaxp = minus_inf; // dist not altered
                if (strcmp(f->word, actual->word) < 0)
                { // current word is the smallest in the node
                    actual->left = temp;
                }
                else if (strcmp(f->word, actual->word) > 0)
                { // current word is the biggest in the node
                    actual->right = temp;
                }
            }
        }
    }
    else
    { // New tree root
        tree_node *new = malloc(sizeof(tree_node));
        strcpy(new->word, f->word);
        new->parent = NULL;
```

```
91          new->left = NULL;
92          new->right = NULL;
93          new->count = 0;
94          new->dmin = plus_inf;   // dist not altered
95          new->dmax = minus_inf;  // dist not altered
96          new->dminp = plus_inf;  // dist not altered
97          new->dmaxp = minus_inf; // dist not altered
98          new->first = f->word_num;
99          new->count++;
100         new->last = f->word_num;
101         new->lastp = f->current_pos;
102         new->firstp = f->word_pos;
103         words[index] = new;
104     }
105 }
```

2.5 Add Element to List

Like the previous function ("Add Node to Ordered Binary Tree"), this function was needed to add a new element with all the required information to a Linked List struct.

A hash-code is generated to the word being read from the *file_data_t* struct introduced into the function, in order to know where to store the word information, as it was previously explained on the function *hash_function*.

Thereafter, it is verified if that position contains an element or if it's doesn't (*NULL*). If it doesn't contains, a new element will be created, storing all the information and being the first one on the Linked List of that index. If it contains, the Linked List is traveled until it finds the element of that word, using the *strcmp* function and updating the information stored, if found. When not found, a new node of that word will be created and added to the next position of the last element on the Linked List.

```
1
2  void add_ele(link_ele **words, file_data_t *f, int size)
3  {
4      int index = hash_function(f->word, size);
5      link_ele *actual = words[index];
6      if (actual != NULL) // if an element in the list already exists in that index
7      {
8          if (strcmp(actual->word, f->word) == 0)
9          { // if equal
10             long tempdist = f->word_num - actual->last;
11             long tempdistp = f->current_pos - actual->lastp;
12             actual->tdist = actual->tdist + tempdist;
13             actual->tdistp = actual->tdistp + tempdistp;
14             if (tempdist < actual->dmin)
15                 actual->dmin = tempdist;
16             if (tempdist > actual->dmax)
17                 actual->dmax = tempdist;
18             if (tempdistp < actual->dminp)
19                 actual->dminp = tempdistp;
20             if (tempdistp > actual->dmaxp)
21                 actual->dmaxp = tempdistp;
22             actual->count++;
23             actual->last = f->word_num;
24             actual->lastp = f->current_pos;
25         }
26         else
27         { // if not equal it is needed to run over all the elements
28             bool found = false;
```

```
            while (actual->next != NULL)
            {
                actual = actual->next;
                if (strcmp(actual->word, f->word) == 0)
                { // if equal
                    long tempdist = f->word_num - actual->last;
                    long tempdistp = f->current_pos - actual->lastp;
                    actual->tdist = actual->tdist + tempdist;
                    actual->tdistp = actual->tdistp + tempdistp;
                    if (tempdist < actual->dmin)
                        actual->dmin = tempdist;
                    if (tempdist > actual->dmax)
                        actual->dmax = tempdist;
                    if (tempdistp < actual->dminp)
                        actual->dminp = tempdistp;
                    if (tempdistp > actual->dmaxp)
                        actual->dmaxp = tempdistp;
                    actual->count++;
                    actual->last = f->word_num;
                    actual->lastp = f->current_pos;
                    found = true;
                    break;
                }
            }
            if (!found) // not found verification
            {
                link_ele *temp = malloc(sizeof(link_ele));
                strcpy(temp->word, f->word);
                temp->first = f->word_num;
                temp->count = 1;
                temp->last = f->word_num;
                temp->lastp = f->current_pos;
                temp->firstp = f->word_pos;
                temp->next = NULL;
                temp->dmin = plus_inf;   // dist not altered
                temp->dmax = minus_inf;  // dist not altered
                temp->dminp = plus_inf;  // dist not altered
                temp->dmaxp = minus_inf; // dist not altered
                actual->next = temp;
            }
        }
    }
    else
    {   // New Start of a linked list
        count_array++;
        link_ele *new = malloc(sizeof(link_ele));
        strcpy(new->word, f->word);
        new->next = NULL;
        new->count = 0;
        new->dmin = plus_inf;   // dist not altered
        new->dmax = minus_inf;  // dist not altered
        new->dminp = plus_inf;  // dist not altered
        new->dmaxp = minus_inf; // dist not altered
        new->first = f->word_num;
        new->count++;
        new->last = f->word_num;
        new->lastp = f->current_pos;
        new->firstp = f->word_pos;
        words[index] = new;
    }
}
```

2.6 Add Element to list using resize

This function was needed to add a word with all the required information from the old Table to the new one, when doing the resize.

A hash-code is generated to the word being read from the old array introduced into the function, in order to know where to store the word information.

Thereafter, it is verified if that position contains an element or if it's doesn't (*NULL*). If it doesn't contains, a new element will be created, storing all the information and being the first one on the Linked List of that index. If it contains, the Linked List is traveled until the first empty position is founded and a new node is created with the information of the word.

```
1
2  void add_ele_resize(link_ele **words, link_ele *f, int size)
3  {
4      int index = hash_function(f->word, size);
5      link_ele *actual = words[index];
6      if (actual != NULL)
7      {
8          while (actual->next != NULL)
9          {
10             actual = actual->next;
11         }
12         link_ele *temp = malloc(sizeof(link_ele));
13         strcpy(temp->word, f->word);
14         temp->next = NULL;
15         temp->count = f->count;
16         temp->dmin = f->dmin;
17         temp->dmax = f->dmax;
18         temp->dminp = f->dminp;
19         temp->dmaxp = f->dmaxp;
20         temp->first = f->first;
21         temp->count = f->count;
22         temp->last = f->last;
23         temp->lastp = f->lastp;
24         temp->firstp = f->firstp;
25         actual->next = temp;
26
27     }
28     else
29     {
30         link_ele *new = malloc(sizeof(link_ele));
31         strcpy(new->word, f->word);
32         new->next = NULL;
33         new->count = f->count;
34         new->dmin = f->dmin;
35         new->dmax = f->dmax;
36         new->dminp = f->dminp;
37         new->dmaxp = f->dmaxp;
38         new->first = f->first;
39         new->count = f->count;
40         new->last = f->last;
41         new->lastp = f->lastp;
42         new->firstp = f->firstp;
43         words[index] = new;
44     }
45 }
```

2.7 Resize

To avoid collisions (same hash-codes for different words) the next function was developed. Note that it only makes sense to develop a resize function for Linked Lists. Binary Trees are supposed to enhance our search efficiency, resizing them would not make any difference.

Every time the Table was almost full (it was decided to resize at 80% of occupation) the the Table was resized to a bigger size. All the information is stored using *words_temp* and then returned.

In a more detailed description, the size of the new array was decided to be twice the size of the older one. Then a temporary array was created and the words were stored on that new array, using a the function "Add Element to list using resize". After that, the new array is simply returned.

```
1
2  ink_ele **resize_link(link_ele **words, int *size)
3  {
4      int newsize = 2 * (*size);
5      link_ele **words_temp = (link_ele *)calloc(newsize, sizeof(link_ele *));
6      for (int i = 0; i < (*size); i++)
7      {
8          if (words[i] != NULL)
9          {
10             link_ele *actual = words[i];
11             add_ele_resize(words_temp, actual, newsize);
12             while (actual->next != NULL)
13             {
14                 actual = actual->next;
15                 add_ele_resize(words_temp, actual, newsize);
16
17             }
18         }
19     }
20     *size = 2 * (*size);
21     return words_temp;
22 }
```

2.8 Get Info Link

To retrieve information about a single word in the Hash Table with Linked Lists the following code was developed.

It is asked for a word to search in the correspondent table. The hash-code for that word is calculated and after that, the Linked List of the index relative to that hash-code is accessed, traveling trough all the elements looking for the respective word and printing the data stored, if found.

If no element of the Linked List is correspondent to the word provided by the user, a warning message is returned, informing the user that the inserted word is not present in the Table.

```
1
2  void get_info_link(link_ele **words, int size)
3  {
4      char name[64];
5      printf("Insert word for info: ");
6      scanf("%[^\n]", name);
7      fflush(stdin);
8      //get info about a word
```

```
    int index = hash_function(name, size);
    link_ele *actual = words[index];
    bool found = false;
    if (actual != NULL)
    {
        while (actual != NULL)
        {
            if (strcmp(actual->word, name) == 0)
            {
                printf("\nInformation about word '%s'\n", actual->word);
                printf("\nCount: %ld\n", actual->count);
                printf("\nPosition (related to the index position of all the text):\n");
                printf("First: %ld\n", actual->first);
                printf("Last: %ld\n", actual->last);
                printf("\nPosition (related to the distinct word counter):\n");
                printf("First: %ld\n", actual->firstp);
                printf("Last: %ld\n", actual->lastp);
                if (actual->count > 1)
                {
                    printf("\nDistances beetween consecutive occurrences (related to the index position of all
    the text):\n");
                    printf("Smallest: %ld\n", actual->dminp);
                    printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 number of
    dist and not words
                    printf("Largest: %ld\n", actual->dmaxp);
                    printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\
    n");
                    printf("Smallest: %ld\n", actual->dmin);
                    printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
                    printf("Largest: %ld\n\n", actual->dmax);
                }
                else
                {
                    printf("\n No distances stats available.\n\n");
                }
                found = true;
                break;
            }
            actual = actual->next;
        }
    }

    if (!found)
    {
        printf("Word %s not found!\n", name);
        exit(0);
    }
}
```

2.9 Get Info Link All

To retrieve information about all the words or to search for a word inserted into the Hash Table with Linked Lists, the following code was developed.

First of all, it is asked if it is wanted a word to search, inserting the star of it, or to get information about all words. This is accomplished using the *gets* function that, although being unsafe, was the easiest and best way to implement. Then, if a word is being find, the function will go through all the Hash Table and all the Linked Lists, printing the information of the stored words that have the first characters equals to the ones inserted by the user. Otherwise, the function will go through all the Hash Table and all the Linked Lists printing the information of all the stored words.

When no words were previously saved, a warning message is returned.

In conjunction with these functionalities, the function also counts the number of words and different words stored, using the *bool all*, in order to jump to printing and counting all words when testing.

```
1
2  int get_info_link_all(link_ele **words, int size, bool all)
3  {
4      bool found = false;
5      int c_st = 0;
6      count_diff = 0;
7      if (all == true)
8          goto all;
9      char name[64];
10     printf("Insert word, or start of it, for info (empty for all): ");
11     if (gets(name) != NULL)
12     {
13         int s_name = strlen(name);
14         for (int i = 0; i < size; i++)
15         {
16             link_ele *actual = words[i];
17             while (actual != NULL)
18             {
19                 if (strncmp(name, actual->word, s_name) == 0)
20                 {
21                     found = true;
22                     printf("\nInformation about word '%s'\n", actual->word);
23                     printf("\nCount: %ld\n", actual->count);
24                     printf("\nPosition (related to the index position of all the text):\n");
25                     printf("First: %ld\n", actual->first);
26                     printf("Last: %ld\n", actual->last);
27                     printf("\nPosition (related to the distinct word counter):\n");
28                     printf("First: %ld\n", actual->firstp);
29                     printf("Last: %ld\n", actual->lastp);
30                     if (actual->count > 1)
31                     {
32                         printf("\nDistances beetween consecutive occurrences (related to the index position of
       all the text):\n");
33                         printf("Smallest: %ld\n", actual->dminp);
34                         printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because
       number of distances and not words
35                         printf("Largest: %ld\n", actual->dmaxp);
36                         printf("\nDistances beetween consecutive occurrences (related to the distinct word
       counter):\n");
37                         printf("Smallest: %ld\n", actual->dmin);
38                         printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
39                         printf("Largest: %ld\n\n", actual->dmax);
40                     }
41                     else
42                     {
43                         printf("\nNo distances stats available.\n\n");
44                     }
45                 }
46                 actual = actual->next;
47             }
48         }
49     }
50     else
51     {
52     all:
53         for (int i = 0; i < size; i++)
```

```
        {
            link_ele *actual = words[i];
            while (actual != NULL)
            {
                found = true;
                c_st += actual->count;
                count_diff++;
                printf("\nInformation about word '%s'\n", actual->word);
                printf("\nCount: %ld\n", actual->count);
                printf("\nPosition (related to the index position of all the text):\n");
                printf("First: %ld\n", actual->first);
                printf("Last: %ld\n", actual->last);
                printf("\nPosition (related to the distinct word counter):\n");
                printf("First: %ld\n", actual->firstp);
                printf("Last: %ld\n", actual->lastp);
                if (actual->count > 1)
                {
                    printf("\nDistances beetween consecutive occurrences (related to the index position of all
    the text):\n");
                    printf("Smallest: %ld\n", actual->dminp);
                    printf("Average: %2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because number
     of distances and not words
                    printf("Largest: %ld\n", actual->dmaxp);
                    printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\
    n");
                    printf("Smallest: %ld\n", actual->dmin);
                    printf("Average: %2f\n", (float)(actual->tdist) / (actual->count - 1));
                    printf("Largest: %ld\n\n", actual->dmax);
                }
                else
                {
                    printf("\nNo distances stats available.\n\n");
                }

                actual = actual->next;
            }
        }
    }
    if (!found)
    {
        printf("No words found!\n");
        exit(0);
    }
    fflush(stdin);
    return c_st;
}
```

2.10 Get Info Node

This function is used to get information from a specific node in a Ordered Binary Tree. It was developed in a very similar way to the "Get Info Link".

It is asked for a word to search in the correspondent Tree. The hash-code for that word is calculated and after that, the Ordered Binary Tree of the index relative to that hash-code is accessed, traveling trough the elements, using the *strcmp* function for improving search speed in this type of struct, looking for the respective word and printing the data stored, if found.

If no element of the List is correspondent to the word provided by the user, a warning message is returned, informing the user that the inserted word is not present in the Table.

```
1
2  void get_info_node(tree_node **words, int size)
3  {
4      char name[64];
5      printf("Insert word for info: ");
6      scanf("%[^\n]", name);
7      fflush(stdin);
8      //get info about a word
9      int index = hash_function(name, size);
10     tree_node *actual = words[index];
11     bool found = false;
12     if (actual != NULL)
13     {
14         while (actual != NULL)
15         {
16             if (strcmp(name, actual->word) < 0 && actual->left != NULL) // word smaller than the node
17                 actual = actual->left;
18
19             else if (strcmp(name, actual->word) > 0 && actual->right != NULL) // word bigger than the node
20                 actual = actual->right;
21
22             else
23             { // if equal
24                 printf("\nInformation about word '%s'\n", actual->word);
25                 printf("\nCount: %ld\n", actual->count);
26                 printf("\nPosition (related to the index position of all the text):\n");
27                 printf("First: %ld\n", actual->first);
28                 printf("Last: %ld\n", actual->last);
29                 printf("\nPosition (related to the distinct word counter):\n");
30                 printf("First: %ld\n", actual->firstp);
31                 printf("Last: %ld\n", actual->lastp);
32                 if (actual->count > 1)
33                 {
34                     printf("\nDistances beetween consecutive occurrences (related to the index position of all
       the text):\n");
35                     printf("Smallest: %ld\n", actual->dminp);
36                     printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because
       number of distances and not words
37                     printf("Largest: %ld\n", actual->dmaxp);
38                     printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\
       n");
39                     printf("Smallest: %ld\n", actual->dmin);
40                     printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
41                     printf("Largest: %ld\n\n", actual->dmax);
42                 }
43                 else
44                 {
45                     printf("\n No distances stats available.\n\n");
46                 }
47                 found = true;
48                 break;
49             }
50         }
51     }
52     if (!found)
53     {
54         printf("Word %s not found!\n", name);
55         exit(0);
56     }
57 }
```

2.11 Get Info Node All

To travel across all words and information stored in the Table with Ordered Binary Trees the code that follows was developed.

Like the Linked List approach, in this function the entire Table is traveled. For each index of the Table there is a Tree storing information about the words processed. Each of these trees are also traveled and printed. By this way, it is possible to show the user all the information stored inside the Hash Table for all words.

This function was developed based on a similar one found on the computer science portal "Geeks for Geeks".

```
1
2  int get_info_node_all(tree_node **words, int size)
3  {
4      int c_stored = 0;
5      count_diff = 0;
6      bool found = false;
7      for (int i = 0; i < size; i++)
8      {
9          tree_node *actual = words[i];
10         tree_node *pre;
11         if (actual != NULL)
12         {
13             while (actual != NULL)
14             {
15                 if (actual->left == NULL)
16                 {
17                     c_stored += actual->count;
18                     count_diff++;
19                     printf("\nInformation about word '%s'\n", actual->word);
20                     printf("\nCount: %ld\n", actual->count);
21                     printf("\nPosition (related to the index position of all the text):\n");
22                     printf("First: %ld\n", actual->first);
23                     printf("Last: %ld\n", actual->last);
24                     printf("\nPosition (related to the distinct word counter):\n");
25                     printf("First: %ld\n", actual->firstp);
26                     printf("Last: %ld\n", actual->lastp);
27                     if (actual->count > 1)
28                     {
29                         printf("\nDistances beetween consecutive occurrences (related to the index position of
       all the text):\n");
30                         printf("Smallest: %ld\n", actual->dminp);
31                         printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because
       number of distances and not words
32                         printf("Largest: %ld\n", actual->dmaxp);
33                         printf("\nDistances beetween consecutive occurrences (related to the distinct word
       counter):\n");
34                         printf("Smallest: %ld\n", actual->dmin);
35                         printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
36                         printf("Largest: %ld\n\n", actual->dmax);
37                     }
38                     else
39                     {
40                         printf("\n No distances stats available.\n\n");
41                     }
42                     found = true;
43                     actual = actual->right;
44                 }
45                 else
46                 {
47                     /* Find the inorder predecessor of current */
48                     pre = actual->left;
49                     while (pre->right != NULL && pre->right != actual)
```

```
50                          pre = pre->right;
51
52                      /* Make current as the right child of its inorder
53                 predecessor */
54                      if (pre->right == NULL)
55                      {
56                          pre->right = actual;
57                          actual = actual->left;
58                      }
59
60                      /* Revert the changes made in the 'if' part to restore
61                 the original tree i.e., fix the right child
62                 of predecessor */
63                      else
64                      {
65                          pre->right = NULL;
66                          c_stored += actual->count;
67                          count_diff++;
68                          printf("\nInformation about word '%s'\n", actual->word);
69                          printf("\nCount: %ld\n", actual->count);
70                          printf("\nPosition (related to the index position of all the text):\n");
71                          printf("First: %ld\n", actual->first);
72                          printf("Last: %ld\n", actual->last);
73                          printf("\nPosition (related to the distinct word counter):\n");
74                          printf("First: %ld\n", actual->firstp);
75                          printf("Last: %ld\n", actual->lastp);
76                          if (actual->count > 1)
77                          {
78                              printf("\nDistances beetween consecutive occurrences (related to the index position
    of all the text):\n");
79                              printf("Smallest: %ld\n", actual->dminp);
80                              printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1
    because number of distances and not words
81                              printf("Largest: %ld\n", actual->dmaxp);
82                              printf("\nDistances beetween consecutive occurrences (related to the distinct word
    counter):\n");
83                              printf("Smallest: %ld\n", actual->dmin);
84                              printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
85                              printf("Largest: %ld\n\n", actual->dmax);
86                          }
87                          else
88                          {
89                              printf("\n No distances stats available.\n\n");
90                          }
91                          found = true;
92                          actual = actual->right;
93                      }
94                  }
95              }
96          }
97      }
98      if (!found)
99      {
100         printf("No words found!\n");
101         exit(0);
102     }
103     return c_stored;
104 }
```

2.12 Main

In the main function, the user has three different options:

- Using the **-l** option will start the program creating a Hash Table with Linked Lists. The program will ask the user for a filename to be processed. Finally, the user may search and get information for a specific word, for words that start with the one inserted or simply list all the words.
- If the user specifies the **-b** option, the program will create a Hash Table using a Ordered Binary Tree rather than a Linked List. As the previous option, -l, after the reading of the specified text file, the user will have the some options, search for a single word or show the entire table content.
- For test purposes, a **-t** option was also developed. With this option the program will read a text file specified by the user, storing the information using HashTable with Linked Lists and, right after, using HashTable with Ordered Binary Trees. A clock is placed for the reading and processing and then, the file *results.txt* is created with times of execution, processing, words read and words processed. This file will serve as a mean of comparison between different implementations and different files of text.

```
1
2  int main(int argc, char *argv[])
3  {
4      if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'l')
5      {
6          count_array = 0;
7          printf("Initializing HashTable with Linked List\n");
8          int s_hash = 500;
9          link_ele **words = (link_ele *)calloc(s_hash, sizeof(link_ele *)); // creates and announce them as zero (
       null)
10         file_data_t *f = malloc(sizeof(file_data_t));
11         char file[64];
12         printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
13         scanf("%[^\n]", file);
14         fflush(stdin);
15         if (!open_text_file(file, f))
16         {
17             while (!read_word(f))
18             {
19                 if ((double)count_array / s_hash >= 0.8)
20                 {
21                     words = resize_link(words, &s_hash);
22                 }
23                 add_ele(words, f, s_hash);
24             }
25             printf("File read successfully!\n");
26             close_text_file(f);
27         }
28         else
29         {
30             printf("--------------------\n");
31             printf("Error opening file!\n");
32             printf("--------------------\n");
33             exit(0);
34         }
35         printf("\n1 - Search for a certain word stats\n2 - Search with a piece of a word or list all words stats\
       n");
36         char option[5];
37         printf("\nOption: ");
```

```
        scanf("%[^\n]", option);
        fflush(stdin);
        if (strcmp(option, "1") == 0)
            get_info_link(words, s_hash);
        else if (strcmp(option, "2") == 0)
            get_info_link_all(words, s_hash, false);
        else
        {
            printf("Invalid option");
            exit(0);
        }
    }
    else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'b')
    {
        printf("Initializing HashTable with Ordered Binary Tree\n");
        int s_hash = 500;
        tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and announce them as
    zero (null)
        file_data_t *f = malloc(sizeof(file_data_t));
        char file[64];
        printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
        scanf("%[^\n]", file);
        fflush(stdin);
        if (!open_text_file(file, f))
        {
            while (!read_word(f))
            {
                add_node(words, f, s_hash);
            }
            printf("File read successfully!\n");
            close_text_file(f);
        }
        else
        {
            printf("-----------------\n");
            printf("Error opening file!\n");
            printf("-----------------\n");
            exit(0);
        }
        printf("\n1 - Search for a certain word stats\n2 - Show all words stats\n");
        char option[5];
        printf("\nOption: ");
        scanf("%[^\n]", option);
        fflush(stdin);
        if (strcmp(option, "1") == 0)
            get_info_node(words, s_hash);
        else if (strcmp(option, "2") == 0)
            get_info_node_all(words, s_hash);
        else
        {
            printf("Invalid option");
            exit(0);
        }
    }
    else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 't')
    {
        char file[64];
        printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
        scanf("%[^\n]", file);
        fflush(stdin);

        printf("Initializing HashTable with Ordered Binary Tree\n");
```

```
99          reset_time();
100         int s_hash = 500;
101         int count_stored = 0;
102         (void)elapsed_time();
103         tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and anounce them as zero
        (null)
104         file_data_t *f = malloc(sizeof(file_data_t));
105
106         if (!open_text_file(file, f))
107         {
108             while (!read_word(f))
109             {
110                 add_node(words, f, s_hash);
111             }
112             printf("File read successfully!\n");
113             close_text_file(f);
114         }
115         else
116         {
117             printf("--------------------\n");
118             printf("Error opening file!\n");
119             printf("--------------------\n");
120             exit(0);
121         }
122
123         cpu_time = elapsed_time();
124         printf("%s %.6f s \n", "File read! Elapsed Time! - Reading", cpu_time);
125
126         FILE *fw = fopen("results.txt", "a+");
127
128         if (fw == NULL)
129         {
130             printf("Erro a abrir o ficheiro escrita!\n");
131             exit(1);
132         }
133         else
134         {
135             printf("%s\n", "Aberto ficheiro results.txt");
136             fprintf(fw, "Filename \t %s \n", file);
137             fprintf(fw, "HashTable OBT Reading Time \t %.6f \n", cpu_time);
138         }
139
140         reset_time();
141
142         printf("\nPrinting all words stored...\n");
143         (void)elapsed_time();
144         usleep(5000000);
145         count_stored = get_info_node_all(words, s_hash);
146         printf("\n ---------------------------------------------------------------------- \n");
147         printf("\n Words read - %ld\n", f->word_num);
148         printf(" Words stored - %d\n", count_stored);
149         printf("%s %d \n", "Number of different word", count_diff);
150         cpu_time = elapsed_time();
151         printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);
152
153         if (fw == NULL)
154         {
155             printf("Erro a abrir o ficheiro results!\n");
156             exit(1);
157         }
158         else
159         {
```

```
            fprintf(fw, "HashTable OBT Words Read \t %ld \n", f->word_num);
            fprintf(fw, "HashTable OBT Words Stored \t %d \n", count_stored);
            fprintf(fw, "%s %d \n", "Number of different word", count_diff);
            fprintf(fw, "HashTable OBT Time Travel Print \t %.6f \n", cpu_time);
        }

        free(words);
        free(f);
        //-------------------------------------------------------------------//

        printf("\n.....\n");

        printf("\nInitializing HashTable with Linked List\n");
        s_hash = 500;
        count_array = 0;
        count_stored = 0;
        reset_time();
        (void)elapsed_time();
        link_ele **words1 = (link_ele *)calloc(s_hash, sizeof(link_ele *)); // creates and anounce them as zero (
    null)
        file_data_t *f1 = malloc(sizeof(file_data_t));

        if (!open_text_file(file, f1))
        {
            while (!read_word(f1))
            {
                if ((double)count_array / s_hash >= 0.8)
                {
                    words1 = resize_link(words1, &s_hash);
                }
                add_ele(words1, f1, s_hash);
            }
            printf("File read successfully!\n");
            close_text_file(f);
        }
        else
        {
            printf("-----------------\n");
            printf("Error opening file!\n");
            printf("-----------------\n");
            exit(0);
        }

        cpu_time = elapsed_time();
        printf("%s %.6f s \n", "File read! Elapsed Time! - Reading", cpu_time);

        if (fw == NULL)
        {
            printf("Erro a abrir o ficheiro escrita!\n");
            exit(1);
        }
        else
        {
            fprintf(fw, "Filename \t %s \n", file);
            fprintf(fw, "HashTable LL Reading Time \t %.6f \n", cpu_time);
        }

        reset_time();

        printf("\nPrinting all words stored...\n");
        (void)elapsed_time();
        usleep(5000000);
```

```
221         count_stored = get_info_link_all(words1, s_hash, true);
222         printf("\n ---------------------------------------------------------------------- \n");
223         printf("\n Words read - %ld\n", f1->word_num);
224         printf(" Words stored - %d\n", count_stored);
225         printf("%s %d \n", "Number of different word", count_diff);
226         cpu_time = elapsed_time();
227         printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);
228 
229         if (fw == NULL)
230         {
231             printf("Erro a abrir o ficheiro results!\n");
232             exit(1);
233         }
234         else
235         {
236             fprintf(fw, "HashTable LL Words Read \t %ld \n", f->word_num);
237             fprintf(fw, "HashTable LL Words Stored \t %d \n", count_stored);
238             fprintf(fw, "%s %d \n", "Number of different word", count_diff);
239             fprintf(fw, "HashTable LL Time Travel Print \t %.6f \n", cpu_time);
240         }
241 
242         fclose(fw);
243         free(words1);
244         free(f1);
245     }
246     else
247     {
248         usage(argv);
249     }
250 }
```

3 Results

After running the program using the -t option, explained previously, without printing the words, the following results[2] were obtained:

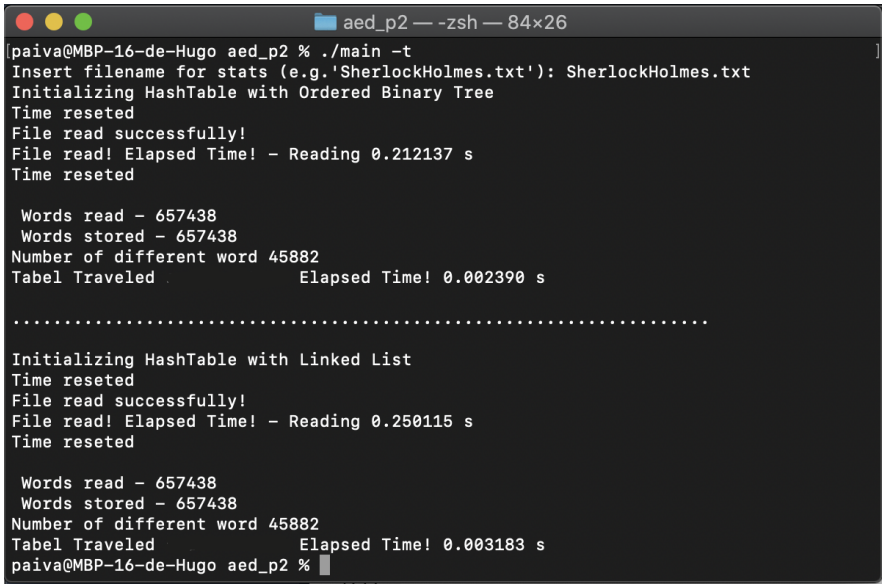

Figure 1: Results after running the program using the -t option and reading from the file *SherlockHolmes.txt*

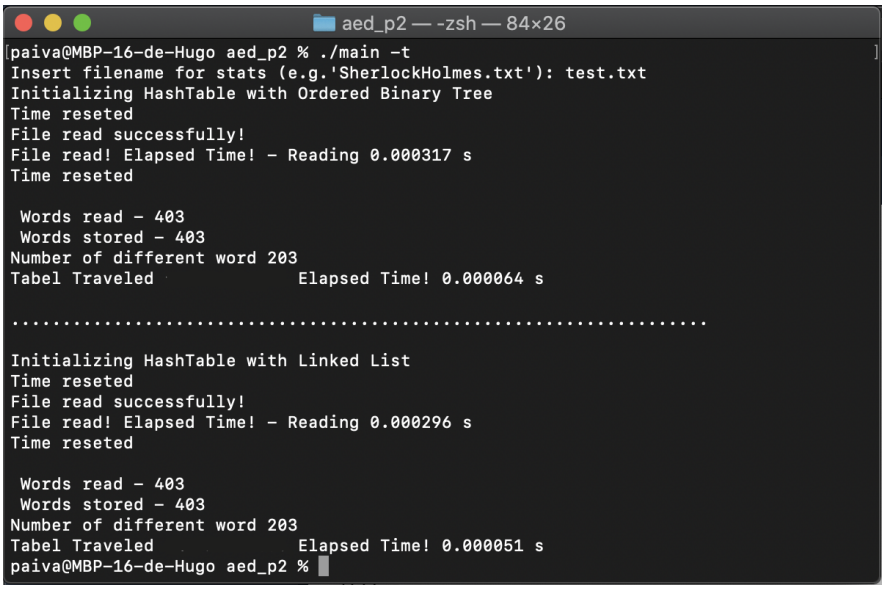

Figure 2: Results after running the program using the -t option and reading from the file *test.txt*

[2]The following results were obtained using the files *SherlockHolmes.txt* and *test.txt*

The other two options of the program were also tested:

```
paiva@MBP-16-de-Hugo aed_p2 % ./main -b
Initializing HashTable with Ordered Binary Tree
Insert filename for stats (e.g.'SherlockHolmes.txt'): SherlockHolmes.txt
File read successfully!

1 - Search for a certain word stats
2 - Show all words stats

Option: 1
Insert word for info: Sherlock

Information about word 'Sherlock'

Count: 391

Position (related to the index position of all the text):
First: 23
Last: 657397

Position (related to the distinct word counter):
First: 268
Last: 3867591

Distances beetween consecutive occurrences (related to the index position of all the text):
Smallest: 27
Average: 9916.19
Largest: 178060

Distances beetween consecutive occurrences (related to the distinct word counter):
Smallest: 3
Average: 1685.57
Largest: 30523

paiva@MBP-16-de-Hugo aed_p2 % ./main -l
Initializing HashTable with Linked List
Insert filename for stats (e.g.'SherlockHolmes.txt'): SherlockHolmes.txt
File read successfully!

1 - Search for a certain word stats
2 - Search with a piece of a word or list all words stats

Option: 1
Insert word for info: Sherlock

Information about word 'Sherlock'

Count: 391

Position (related to the index position of all the text):
First: 23
Last: 657397

Position (related to the distinct word counter):
First: 268
Last: 3867591

Distances beetween consecutive occurrences (related to the index position of all the text):
Smallest: 27
Average: 9916.19
Largest: 178060

Distances beetween consecutive occurrences (related to the distinct word counter):
Smallest: 3
Average: 1685.57
Largest: 30523
```

Figure 3: Running the program with both the options -b and -l and searching for the word *Sherlock*

```
paiva@MBP-16-de-Hugo aed_p2 % ./main -l
Initializing HashTable with Linked List
Insert filename for stats (e.g.'SherlockHolmes.txt'): SherlockHolmes.txt
File read successfully!

1 - Search for a certain word stats
2 - Search with a piece of a word or list all words stats

Option: 2
warning: this program uses gets(), which is unsafe.
Insert word, or start of it, for info (empty for all): Beeche

Information about word 'Beecher's'

Count: 1

Position (related to the index position of all the text):
First: 519584
Last: 519584

Position (related to the distinct word counter):
First: 3054180
Last: 3054189

No distances stats available.


Information about word 'Beecher,'

Count: 1

Position (related to the index position of all the text):
First: 519545
Last: 519545

Position (related to the distinct word counter):
First: 3053941
Last: 3053949

No distances stats available.


Information about word 'Beecher'

Count: 2

Position (related to the index position of all the text):
First: 519478
Last: 519653

Position (related to the distinct word counter):
First: 3053530
Last: 3054575

Distances beetween consecutive occurrences (related to the index position of all the text):
Smallest: 1038
Average: 1038.00
Largest: 1038

Distances beetween consecutive occurrences (related to the distinct word counter):
Smallest: 175
Average: 175.00
Largest: 175


Information about word 'Beeches'

Count: 3

Position (related to the index position of all the text):
```

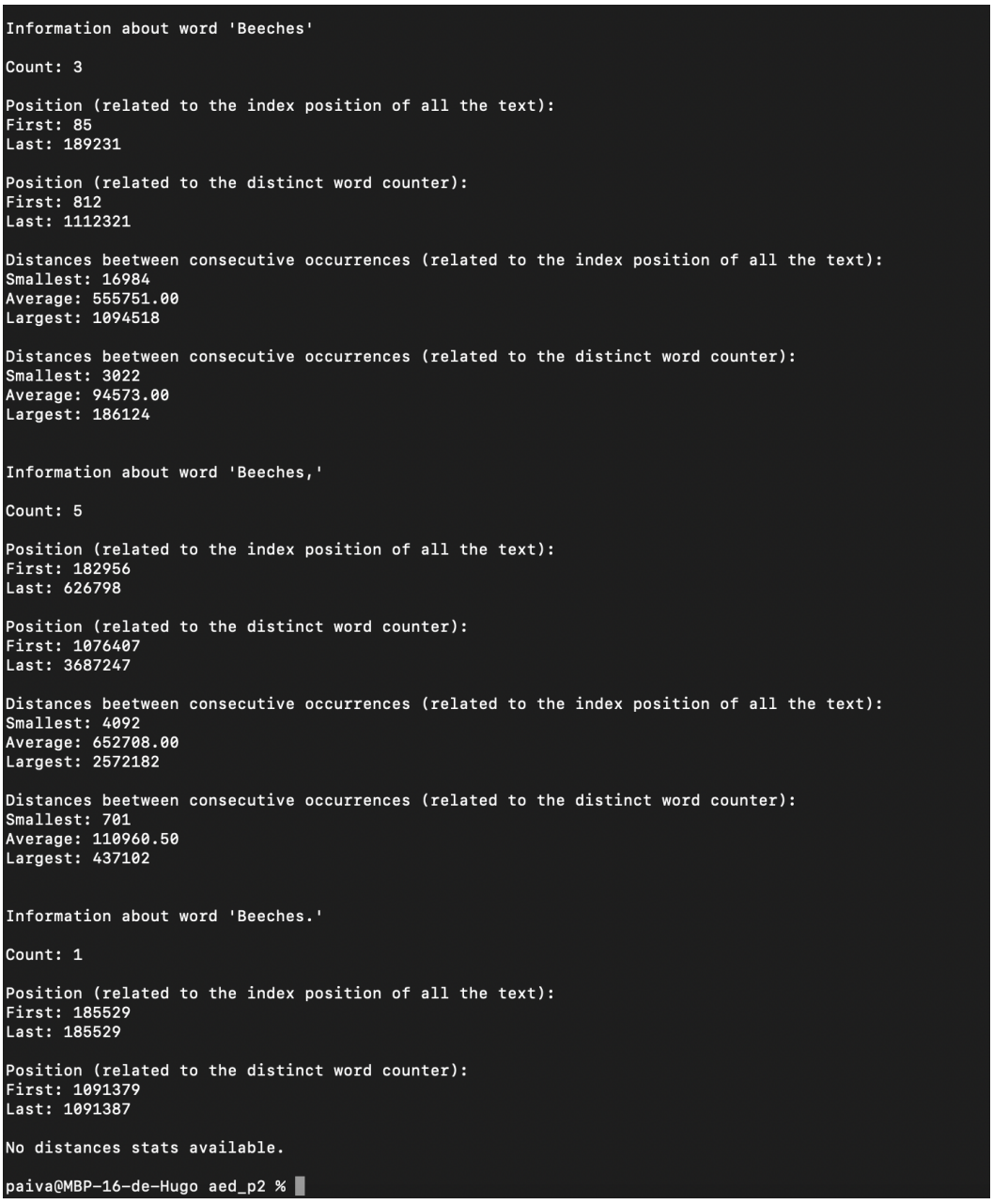

Figure 4: Running the program with the option -l and search by the start of a word

There are no figures of the options for printing all the words because they are not relevant for this example.

Using the data acquired from the reading the files using the program:

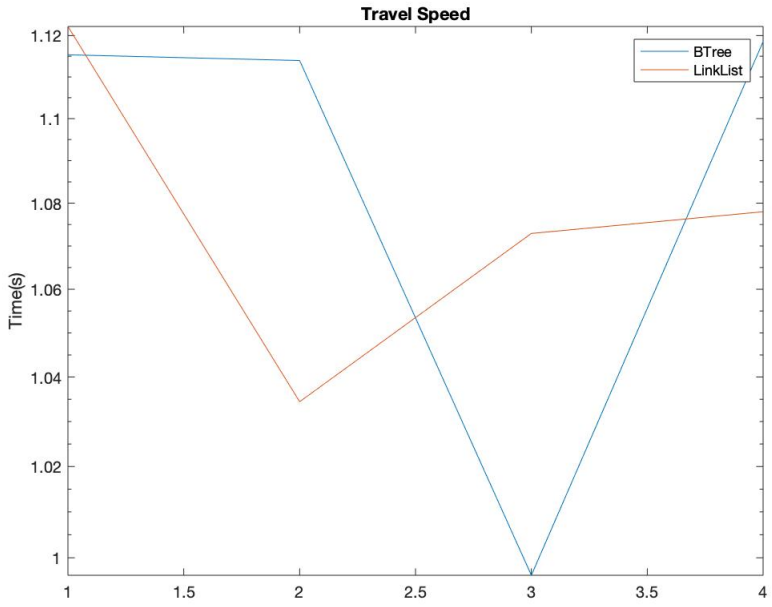

Figure 5: Time spent while travelling through all the words from the *SherlockHolmes.txt* file, according to the number of times the program has ran and for both, Linked List and Ordered Binary Tree structs

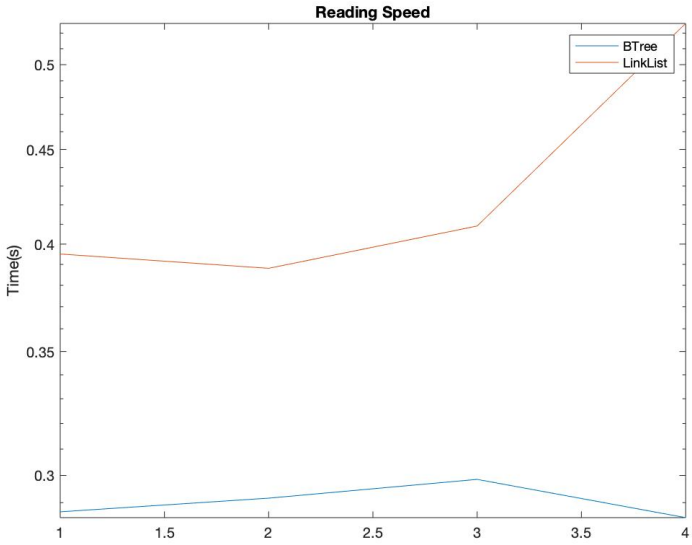

Figure 6: Time spent while reading all the words from the *SherlockHolmes.txt* file, according to the number of times the program has ran and for both, Linked List and Ordered Binary Tree structs

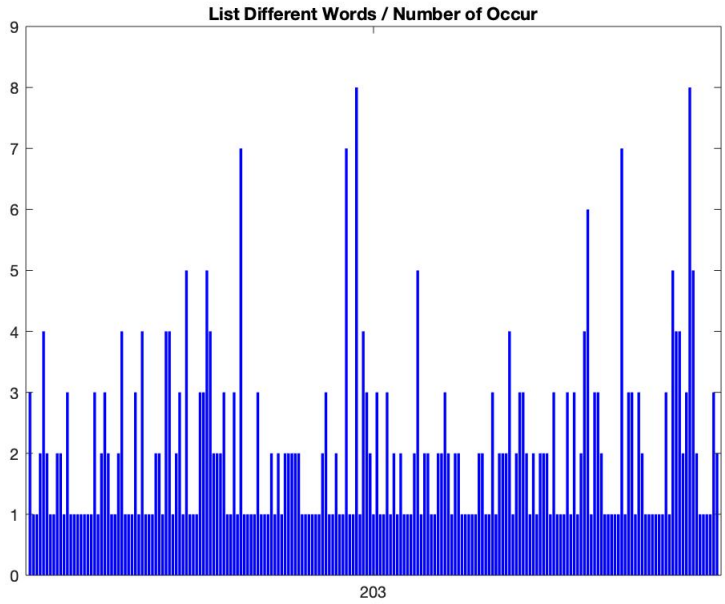

Figure 7: Number of occurrences of different words on the *test.txt* file, using Linked List struct

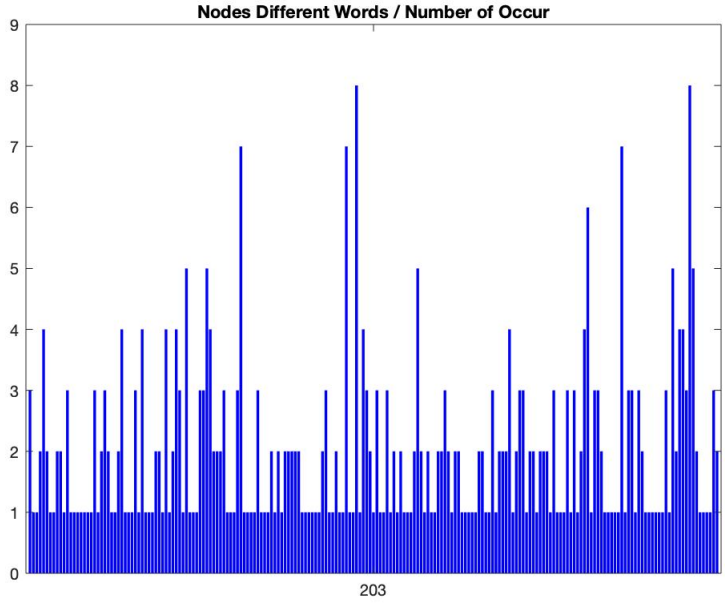

Figure 8: Number of occurrences of different words on the *test.txt* file, using Ordered Binary Tree struct

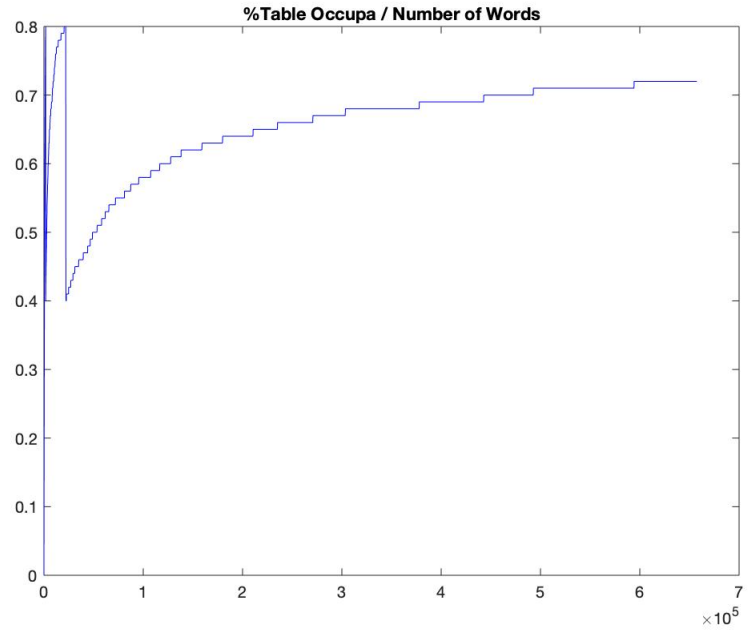

Figure 9: Table occupancy rate relative to the number of words read, on the *SherlockHolmes.txt* file, using Linked List struct

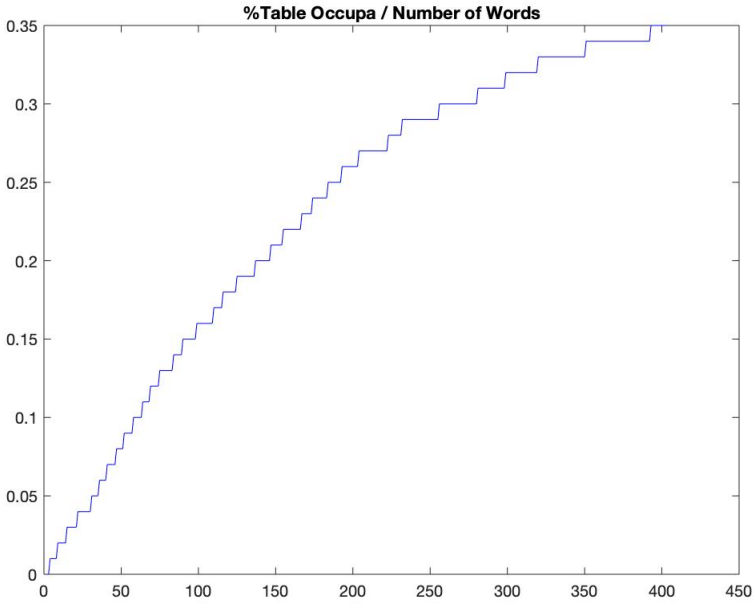

Figure 10: Table occupancy rate relative to the number of words read, on the *test.txt* file, using Linked List struct

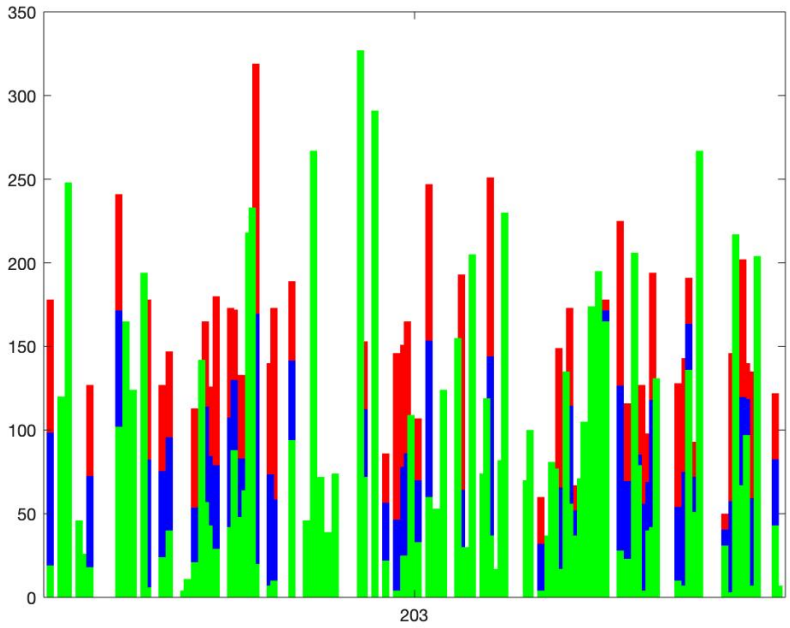

Figure 11: Maximum (red), average (blue) and minimum (green) distances between consecutive words, on the *test.txt* file

From the previous results it can be stated that the reading time for both, Linked List and Ordered Binary Tree structs are very similar however, the number of words read and stored in the Table are the same. This proves that every word read was processed and stored.

It can also be seen that, no matter if it is used Ordered Binary Tree or Linked List structs, the number of occurrences of different words are the same, proving a correct reading.

Furthermore, as expected, the resize of the Table worked as it was designed, lowering its occupancy rate to half when it was at eighty percent of its capacity.

Analyzing Figure 11, sometimes there is only a distance recorded so, the minimum and maximum are the same. In these cases, the minimum is visible instead of the maximum.

Travel and reading time across the Tables with Linked Lists and Ordered Binary Trees are also very similar.

4 Conclusion

During the development of this practical work it was expected that the search time of the Ordered Binary Tree was less than the time of the Linked List implementation. This was expected because of the fact that the Tree is ordered while the Lists are not. From the results previously obtained, although the time of travel through the Ordered Binary Tree is, sometimes less, both implementations behave in a very similar way. They have a similar reading and search time, which was not accord what was expected.

Initially, there was also some confusion on how to resize the Table but, after many errors, the group came to a conclusion that every word should be hashed and added to new Table, instead of an entire Linked List of an index.

Other than this, according to the goals set by the teachers, the work was a success.

5 Bibliography

[1] https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/

6 Appendix

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <unistd.h>
#include <ctype.h>

#define minus_inf -1000000000 // a very small integer
#define plus_inf +1000000000  // a very large integer
int count_array;              // array size being used

int count_diff; // different word counter

static double cpu_time; // time counter

////////////////////////////////////////////////////////////////////////////////////////////
//
// code to measure the elapsed time used by a program fragment (an almost copy of elapsed_time.h)
//
// use as follows:
//
//   (void)elapsed_time();
//   // put your code to be time measured here
//   dt = elapsed_time();
//   // put morecode to be time measured here
//   dt = elapsed_time();
//
// elapsed_time() measures the CPU time between consecutive calls
//

#if defined(__linux__) || defined(__APPLE__)

//
// GNU/Linux and MacOS code to measure elapsed time
//

#include <time.h>

static double elapsed_time(void)
{
    static struct timespec last_time, current_time;

    last_time = current_time;
    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &current_time) != 0)
        return -1.0; // clock_gettime() failed !!!
    return ((double)current_time.tv_sec - (double)last_time.tv_sec) + 1.0e-9 * ((double)current_time.tv_nsec - (
     double)last_time.tv_nsec);
}

#endif

#if defined(_MSC_VER) || defined(_WIN32) || defined(_WIN64)

//
// Microsoft Windows code to measure elapsed time
//

#include <windows.h>
```

```
58
59 static double elapsed_time(void)
60 {
61     static LARGE_INTEGER frequency, last_time, current_time;
62     static int first_time = 1;
63
64     if (first_time != 0)
65     {
66         QueryPerformanceFrequency(&frequency);
67         first_time = 0;
68     }
69     last_time = current_time;
70     QueryPerformanceCounter(&current_time);
71     return (double)(current_time.QuadPart - last_time.QuadPart) / (double)frequency.QuadPart;
72 }
73
74 #endif
75
76 static void reset_time(void)
77 {
78     printf("%s\n", "Time reseted");
79     cpu_time = 0.0;
80 }
81
82 typedef struct file_data
83 {                   // public data
84     long word_pos; // zero-based
85     long word_num; // zero-based
86     char word[64];
87     // private data
88     FILE *fp;
89     long current_pos; // zero-based
90 } file_data_t;
91
92 typedef struct link_ele
93 {
94     char word[64];
95     long count;            // word counter
96     long tdist;            // total sum of distances (in relation to the general word counter)
97     long tdistp;           // total sum of distances (in relation to the index position)
98     long dmin;             // min distance (in relation to the general word counter)
99     long dmax;             // max distance (in relation to the general word counter)
100    long dminp;            // min distance (in relation to the index position)
101    long dmaxp;            // max distance (in relation to the index position)
102    long last;             // last position (in relation to the general word counter)
103    long first;            // first position (in relation to the general word counter)
104    long lastp;            // last position (in relation to the index position)
105    long firstp;           // first position (in relation to the index position)
106    struct link_ele *next; // next word pointer
107 } link_ele;
108
109 typedef struct tree_node
110 {
111     struct tree_node *left;   // pointer to the left branch (a sub-tree)
112     struct tree_node *right;  // pointer to the right branch (a sub-tree)
113     struct tree_node *parent; // optional
114     char word[64];
115     long count;  // word counter
116     long tdist;  // total sum of distances (in relation to the general word counter)
117     long tdistp; // total sum of distances (in relation to the index position)
118     long dmin;   // min distance (in relation to the general word counter)
119     long dmax;   // max distance (in relation to the general word counter)
```

```
120     long dminp;  // min distance (in relation to the index position)
121     long dmaxp;  // max distance (in relation to the index position)
122     long last;   // last position (in relation to the general word counter)
123     long first;  // first position (in relation to the general word counter)
124     long lastp;  // last position (in relation to the index position)
125     long firstp; // first position (in relation to the index position)
126     long data;   // the data item (we use an int here, but it can be anything)
127 } tree_node;
128 
129 unsigned int hash_function(const char *str, unsigned int s)
130 { // for 32-bit unsigned integers, s should be smaller that 16777216u
131     unsigned int h;
132     for (h = 0u; *str != '\0'; str++)
133         h = (256u * h + (0xFFu & (unsigned int)*str)) % s;
134     return h;
135 }
136 
137 void add_node(tree_node **words, file_data_t *f, int size)
138 {
139     int index = hash_function(f->word, size);
140     tree_node *actual = words[index];
141     if (actual != NULL) // if there is already an element in the ordered binary tree
142     {
143         if (strcmp(actual->word, f->word) == 0)
144         { // if that element is the same
145             long tempdist = f->word_num - actual->last;
146             long tempdistp = f->current_pos - actual->lastp;
147             actual->tdist = actual->tdist + tempdist;
148             actual->tdistp = actual->tdistp + tempdistp;
149             if (tempdist < actual->dmin)
150                 actual->dmin = tempdist;
151             if (tempdist > actual->dmax)
152                 actual->dmax = tempdist;
153             if (tempdistp < actual->dminp)
154                 actual->dminp = tempdistp;
155             if (tempdistp > actual->dmaxp)
156                 actual->dmaxp = tempdistp;
157             actual->count++;
158             actual->last = f->word_num;
159             actual->lastp = f->current_pos;
160         }
161         else
162         { // if the element is not the same we travel through the next elements to check if there is any equal
163             bool found = false;
164             while (actual != NULL) // While word not found and children not null
165             {
166                 if (strcmp(f->word, actual->word) < 0 && actual->left != NULL) // actual word is smaller
167                     actual = actual->left;
168 
169                 else if (strcmp(f->word, actual->word) > 0 && actual->right != NULL) // actual word is bigger
170                     actual = actual->right;
171 
172                 else if (strcmp(f->word, actual->word) == 0)
173                 { // if equal
174                     long tempdist = f->word_num - actual->last;
175                     long tempdistp = f->current_pos - actual->lastp;
176                     actual->tdist = actual->tdist + tempdist;
177                     actual->tdistp = actual->tdistp + tempdistp;
178                     if (tempdist < actual->dmin)
179                         actual->dmin = tempdist;
180                     if (tempdist > actual->dmax)
181                         actual->dmax = tempdist;
```

```
182                     if (tempdistp < actual->dminp)
183                         actual->dminp = tempdistp;
184                     if (tempdistp > actual->dmaxp)
185                         actual->dmaxp = tempdistp;
186                     actual->count++;
187                     actual->last = f->word_num;
188                     actual->lastp = f->current_pos;
189                     found = true;
190                     break;
191                 }
192                 else
193                     break;
194             }
195 
196             if (!found) // check that no elem was found
197             {
198                 tree_node *temp = malloc(sizeof(tree_node));
199                 strcpy(temp->word, f->word);
200                 temp->first = f->word_num;
201                 temp->count = 1;
202                 temp->last = f->word_num;
203                 temp->lastp = f->current_pos;
204                 temp->firstp = f->word_pos;
205                 temp->parent = actual;
206                 temp->dmin = plus_inf;   // dist not altered
207                 temp->dmax = minus_inf;  // dist not altered
208                 temp->dminp = plus_inf;  // dist not altered
209                 temp->dmaxp = minus_inf; // dist not altered
210                 if (strcmp(f->word, actual->word) < 0)
211                 { // current word is the smallest in the node
212                     actual->left = temp;
213                 }
214                 else if (strcmp(f->word, actual->word) > 0)
215                 { // current word is the biggest in the node
216                     actual->right = temp;
217                 }
218             }
219         }
220     }
221     else
222     { // New tree root
223         tree_node *new = malloc(sizeof(tree_node));
224         strcpy(new->word, f->word);
225         new->parent = NULL;
226         new->left = NULL;
227         new->right = NULL;
228         new->count = 0;
229         new->dmin = plus_inf;   // dist not altered
230         new->dmax = minus_inf;  // dist not altered
231         new->dminp = plus_inf;  // dist not altered
232         new->dmaxp = minus_inf; // dist not altered
233         new->first = f->word_num;
234         new->count++;
235         new->last = f->word_num;
236         new->lastp = f->current_pos;
237         new->firstp = f->word_pos;
238         words[index] = new;
239     }
240 }
241 
242 void add_ele(link_ele **words, file_data_t *f, int size)
243 {
```

```
244     int index = hash_function(f->word, size);
245     link_ele *actual = words[index];
246     if (actual != NULL) // if an element in the list already exists in that index
247     {
248         if (strcmp(actual->word, f->word) == 0)
249         { // if equal
250             long tempdist = f->word_num - actual->last;
251             long tempdistp = f->current_pos - actual->lastp;
252             actual->tdist = actual->tdist + tempdist;
253             actual->tdistp = actual->tdistp + tempdistp;
254             if (tempdist < actual->dmin)
255                 actual->dmin = tempdist;
256             if (tempdist > actual->dmax)
257                 actual->dmax = tempdist;
258             if (tempdistp < actual->dminp)
259                 actual->dminp = tempdistp;
260             if (tempdistp > actual->dmaxp)
261                 actual->dmaxp = tempdistp;
262             actual->count++;
263             actual->last = f->word_num;
264             actual->lastp = f->current_pos;
265         }
266         else
267         { // if not equal it is needed to run over all the elements
268             bool found = false;
269             while (actual->next != NULL)
270             {
271                 actual = actual->next;
272                 if (strcmp(actual->word, f->word) == 0)
273                 { // if equal
274                     long tempdist = f->word_num - actual->last;
275                     long tempdistp = f->current_pos - actual->lastp;
276                     actual->tdist = actual->tdist + tempdist;
277                     actual->tdistp = actual->tdistp + tempdistp;
278                     if (tempdist < actual->dmin)
279                         actual->dmin = tempdist;
280                     if (tempdist > actual->dmax)
281                         actual->dmax = tempdist;
282                     if (tempdistp < actual->dminp)
283                         actual->dminp = tempdistp;
284                     if (tempdistp > actual->dmaxp)
285                         actual->dmaxp = tempdistp;
286                     actual->count++;
287                     actual->last = f->word_num;
288                     actual->lastp = f->current_pos;
289                     found = true;
290                     break;
291                 }
292             }
293             if (!found) // not found verification
294             {
295                 link_ele *temp = malloc(sizeof(link_ele));
296                 strcpy(temp->word, f->word);
297                 temp->first = f->word_num;
298                 temp->count = 1;
299                 temp->last = f->word_num;
300                 temp->lastp = f->current_pos;
301                 temp->firstp = f->word_pos;
302                 temp->next = NULL;
303                 temp->dmin = plus_inf;   // dist not altered
304                 temp->dmax = minus_inf;  // dist not altered
305                 temp->dminp = plus_inf;  // dist not altered
```

```
306                 temp->dmaxp = minus_inf; // dist not altered
307                 actual->next = temp;
308             }
309         }
310     }
311     else
312     { // New Start of a linked list
313         count_array++;
314         link_ele *new = malloc(sizeof(link_ele));
315         strcpy(new->word, f->word);
316         new->next = NULL;
317         new->count = 0;
318         new->dmin = plus_inf;   // dist not altered
319         new->dmax = minus_inf;  // dist not altered
320         new->dminp = plus_inf;  // dist not altered
321         new->dmaxp = minus_inf; // dist not altered
322         new->first = f->word_num;
323         new->count++;
324         new->last = f->word_num;
325         new->lastp = f->current_pos;
326         new->firstp = f->word_pos;
327         words[index] = new;
328     }
329 }
330
331 void add_ele_resize(link_ele **words, link_ele *f, int size)
332 {
333     int index = hash_function(f->word, size);
334     link_ele *actual = words[index];
335     if (actual != NULL)
336     {
337         while (actual->next != NULL)
338         {
339             actual = actual->next;
340         }
341         link_ele *temp = malloc(sizeof(link_ele));
342         strcpy(temp->word, f->word);
343         temp->next = NULL;
344         temp->count = f->count;
345         temp->dmin = f->dmin;
346         temp->dmax = f->dmax;
347         temp->dminp = f->dminp;
348         temp->dmaxp = f->dmaxp;
349         temp->first = f->first;
350         temp->count = f->count;
351         temp->last = f->last;
352         temp->lastp = f->lastp;
353         temp->firstp = f->firstp;
354         actual->next = temp;
355
356     }
357     else
358     {
359         link_ele *new = malloc(sizeof(link_ele));
360         strcpy(new->word, f->word);
361         new->next = NULL;
362         new->count = f->count;
363         new->dmin = f->dmin;
364         new->dmax = f->dmax;
365         new->dminp = f->dminp;
366         new->dmaxp = f->dmaxp;
367         new->first = f->first;
```

```
368         new->count = f->count;
369         new->last = f->last;
370         new->lastp = f->lastp;
371         new->firstp = f->firstp;
372         words[index] = new;
373     }
374 }
375 
376 link_ele **resize_link(link_ele **words, int *size)
377 {
378     int newsize = 2 * (*size);
379     link_ele **words_temp = (link_ele *)calloc(newsize, sizeof(link_ele *));
380     for (int i = 0; i < (*size); i++)
381     {
382         if (words[i] != NULL)
383         {
384             link_ele *actual = words[i];
385             add_ele_resize(words_temp, actual, newsize);
386             while (actual->next != NULL)
387             {
388                 actual = actual->next;
389                 add_ele_resize(words_temp, actual, newsize);
390 
391             }
392         }
393     }
394     *size = 2 * (*size);
395     return words_temp;
396 }
397 
398 void get_info_link(link_ele **words, int size)
399 {
400     char name[64];
401     printf("Insert word for info: ");
402     scanf("%[^\n]", name);
403     fflush(stdin);
404     //get info about a word
405     int index = hash_function(name, size);
406     link_ele *actual = words[index];
407     bool found = false;
408     if (actual != NULL)
409     {
410         while (actual != NULL)
411         {
412             if (strcmp(actual->word, name) == 0)
413             {
414                 printf("\nInformation about word '%s'\n", actual->word);
415                 printf("\nCount: %ld\n", actual->count);
416                 printf("\nPosition (related to the index position of all the text):\n");
417                 printf("First: %ld\n", actual->first);
418                 printf("Last: %ld\n", actual->last);
419                 printf("\nPosition (related to the distinct word counter):\n");
420                 printf("First: %ld\n", actual->firstp);
421                 printf("Last: %ld\n", actual->lastp);
422                 if (actual->count > 1)
423                 {
424                     printf("\nDistances beetween consecutive occurrences (related to the index position of all
    the text):\n");
425                     printf("Smallest: %ld\n", actual->dminp);
426                     printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 number of
    dist and not words
427                     printf("Largest: %ld\n", actual->dmaxp);
```

```
428                     printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\
    n");
429                     printf("Smallest: %ld\n", actual->dmin);
430                     printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
431                     printf("Largest: %ld\n\n", actual->dmax);
432                 }
433                 else
434                 {
435                     printf("\n No distances stats available.\n\n");
436                 }
437                 found = true;
438                 break;
439             }
440             actual = actual->next;
441         }
442     }
443 
444     if (!found)
445     {
446         printf("Word %s not found!\n", name);
447         exit(0);
448     }
449 }
450 
451 void get_info_node(tree_node **words, int size)
452 {
453     char name[64];
454     printf("Insert word for info: ");
455     scanf("%[^\n]", name);
456     fflush(stdin);
457     //get info about a word
458     int index = hash_function(name, size);
459     tree_node *actual = words[index];
460     bool found = false;
461     if (actual != NULL)
462     {
463         while (actual != NULL)
464         {
465             if (strcmp(name, actual->word) < 0 && actual->left != NULL) // word smaller than the node
466                 actual = actual->left;
467 
468             else if (strcmp(name, actual->word) > 0 && actual->right != NULL) // word bigger than the node
469                 actual = actual->right;
470 
471             else
472             { // if equal
473                 printf("\nInformation about word '%s'\n", actual->word);
474                 printf("\nCount: %ld\n", actual->count);
475                 printf("\nPosition (related to the index position of all the text):\n");
476                 printf("First: %ld\n", actual->first);
477                 printf("Last: %ld\n", actual->last);
478                 printf("\nPosition (related to the distinct word counter):\n");
479                 printf("First: %ld\n", actual->firstp);
480                 printf("Last: %ld\n", actual->lastp);
481                 if (actual->count > 1)
482                 {
483                     printf("\nDistances beetween consecutive occurrences (related to the index position of all
    the text):\n");
484                     printf("Smallest: %ld\n", actual->dminp);
485                     printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because
    number of distances and not words
486                     printf("Largest: %ld\n", actual->dmaxp);
```

```
487                     printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\
    n");
488                     printf("Smallest: %ld\n", actual->dmin);
489                     printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
490                     printf("Largest: %ld\n\n", actual->dmax);
491                 }
492                 else
493                 {
494                     printf("\n No distances stats available.\n\n");
495                 }
496                 found = true;
497                 break;
498             }
499         }
500     }
501     if (!found)
502     {
503         printf("Word %s not found!\n", name);
504         exit(0);
505     }
506 }
507
508 int get_info_node_all(tree_node **words, int size)
509 {
510     int c_stored = 0;
511     count_diff = 0;
512     FILE *f_node_all = fopen("results_fna.txt", "a+");
513     if (f_node_all == NULL)
514     {
515         printf("Erro a abrir o ficheiro escrita!\n");
516         exit(1);
517     }
518     else
519     {
520         printf("%s\n", "Aberto ficheiro results_fna.txt");
521     }
522
523     fprintf(f_node_all, "Word \t Count \t FPos \t LPos \t Dmin \t AvgD \t Dmax \n");
524     bool found = false;
525     for (int i = 0; i < size; i++)
526     {
527         tree_node *actual = words[i];
528         tree_node *pre;
529         if (actual != NULL)
530         {
531             while (actual != NULL)
532             {
533                 if (actual->left == NULL)
534                 {
535                     c_stored += actual->count;
536                     count_diff++;
537                     printf("\nInformation about word '%s'\n", actual->word);
538                     printf("\nCount: %ld\n", actual->count);
539                     printf("\nPosition (related to the index position of all the text):\n");
540                     printf("First: %ld\n", actual->first);
541                     printf("Last: %ld\n", actual->last);
542                     printf("\nPosition (related to the distinct word counter):\n");
543                     printf("First: %ld\n", actual->firstp);
544                     printf("Last: %ld\n", actual->lastp);
545                     fprintf(f_node_all, "%s \t %ld \t %ld \t %ld \t ", actual->word, actual->count, actual->first
    , actual->last);
546                     if (actual->count > 1)
```

```
547                     {
548                         printf("\nDistances beetween consecutive occurrences (related to the index position of
    all the text):\n");
549                         printf("Smallest: %ld\n", actual->dminp);
550                         printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because
    number of distances and not words
551                         printf("Largest: %ld\n", actual->dmaxp);
552                         printf("\nDistances beetween consecutive occurrences (related to the distinct word
    counter):\n");
553                         printf("Smallest: %ld\n", actual->dmin);
554                         printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
555                         printf("Largest: %ld\n\n", actual->dmax);
556                         fprintf(f_node_all, "%ld \t %.2f \t %ld \n ", actual->dmin, (float)(actual->tdist) / (
    actual->count - 1), actual->dmax);
557                     }
558                     else
559                     {
560                         printf("\n No distances stats available.\n\n");
561                         fprintf(f_node_all, "%d \t %.2f \t %d \n ", 0, 0.00, 0);
562                     }
563                     found = true;
564                     actual = actual->right;
565                 }
566                 else
567                 {
568                     /* Find the inorder predecessor of current */
569                     pre = actual->left;
570                     while (pre->right != NULL && pre->right != actual)
571                         pre = pre->right;
572 
573                     /* Make current as the right child of its inorder
574                 predecessor */
575                     if (pre->right == NULL)
576                     {
577                         pre->right = actual;
578                         actual = actual->left;
579                     }
580 
581                     /* Revert the changes made in the 'if' part to restore
582                 the original tree i.e., fix the right child
583                 of predecessor */
584                     else
585                     {
586                         pre->right = NULL;
587                         c_stored += actual->count;
588                         count_diff++;
589                         printf("\nInformation about word '%s'\n", actual->word);
590                         printf("\nCount: %ld\n", actual->count);
591                         printf("\nPosition (related to the index position of all the text):\n");
592                         printf("First: %ld\n", actual->first);
593                         printf("Last: %ld\n", actual->last);
594                         printf("\nPosition (related to the distinct word counter):\n");
595                         printf("First: %ld\n", actual->firstp);
596                         printf("Last: %ld\n", actual->lastp);
597                         fprintf(f_node_all, "%s \t %ld \t %ld \t %ld \t ", actual->word, actual->count, actual->
    first, actual->last);
598 
599                         if (actual->count > 1)
600                         {
601                             printf("\nDistances beetween consecutive occurrences (related to the index position
    of all the text):\n");
602                         printf("Smallest: %ld\n", actual->dminp);
```

```
603                             printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1
    because number of distances and not words
604                             printf("Largest: %ld\n", actual->dmaxp);
605                             printf("\nDistances beetween consecutive occurrences (related to the distinct word
    counter):\n");
606                             printf("Smallest: %ld\n", actual->dmin);
607                             printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
608                             printf("Largest: %ld\n\n", actual->dmax);
609                             fprintf(f_node_all, "%ld \t %.2f \t %ld \n ", actual->dmin, (float)(actual->tdist) /
    (actual->count - 1), actual->dmax);
610
611                         }
612                         else
613                         {
614                             printf("\n No distances stats available.\n\n");
615                             fprintf(f_node_all, "%d \t %.2f \t %d \n ", 0, 0.00, 0);
616                         }
617                         found = true;
618                         actual = actual->right;
619                     }
620                 }
621             }
622         }
623     }
624     if (!found)
625     {
626         printf("No words found!\n");
627         exit(0);
628     }
629     return c_stored;
630 }
631
632 int get_info_link_all(link_ele **words, int size, bool all)
633 {
634     bool found = false;
635     int c_st = 0;
636     count_diff = 0;
637     FILE *f_link_all = fopen("results_fla.txt", "a+");
638     if (all == true)
639         goto all;
640     char name[64];
641     printf("Insert word, or start of it, for info (empty for all): ");
642     if (gets(name) != NULL)
643     {
644         int s_name = strlen(name);
645         for (int i = 0; i < size; i++)
646         {
647             link_ele *actual = words[i];
648             while (actual != NULL)
649             {
650                 if (strncmp(name, actual->word, s_name) == 0)
651                 {
652                     found = true;
653                     printf("\nInformation about word '%s'\n", actual->word);
654                     printf("\nCount: %ld\n", actual->count);
655                     printf("\nPosition (related to the index position of all the text):\n");
656                     printf("First: %ld\n", actual->first);
657                     printf("Last: %ld\n", actual->last);
658                     printf("\nPosition (related to the distinct word counter):\n");
659                     printf("First: %ld\n", actual->firstp);
660                     printf("Last: %ld\n", actual->lastp);
661                     if (actual->count > 1)
```

```
                    {
                        printf("\nDistances beetween consecutive occurrences (related to the index position of all the text):\n");
                        printf("Smallest: %ld\n", actual->dminp);
                        printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because number of distances and not words
                        printf("Largest: %ld\n", actual->dmaxp);
                        printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\n");
                        printf("Smallest: %ld\n", actual->dmin);
                        printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
                        printf("Largest: %ld\n\n", actual->dmax);
                    }
                    else
                    {
                        printf("\nNo distances stats available.\n\n");
                    }
                }
                actual = actual->next;
            }
        }
    }
    else
    {
    all:

        if (f_link_all == NULL)
        {
            printf("Erro a abrir o ficheiro escrita!\n");
            exit(1);
        }
        else
        {
            printf("%s\n", "Aberto ficheiro results_fla.txt");
        }

        fprintf(f_link_all, "Word \t Count \t FPos \t LPos \t Dmin \t AvgD \t Dmax \n");

        for (int i = 0; i < size; i++)
        {
            link_ele *actual = words[i];
            while (actual != NULL)
            {
                found = true;
                c_st += actual->count;
                count_diff++;
                fprintf(f_link_all, "%s \t %ld \t %ld \t %ld \t ", actual->word, actual->count, actual->first, actual->last);
                printf("\nInformation about word '%s'\n", actual->word);
                printf("\nCount: %ld\n", actual->count);
                printf("\nPosition (related to the index position of all the text):\n");
                printf("First: %ld\n", actual->first);
                printf("Last: %ld\n", actual->last);
                printf("\nPosition (related to the distinct word counter):\n");
                printf("First: %ld\n", actual->firstp);
                printf("Last: %ld\n", actual->lastp);
                if (actual->count > 1)
                {
                    printf("\nDistances beetween consecutive occurrences (related to the index position of all the text):\n");
                    printf("Smallest: %ld\n", actual->dminp);
                    printf("Average: %2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because number
```

```
      of distances and not words
                    printf("Largest: %ld\n", actual->dmaxp);
                    printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\
    n");
                    printf("Smallest: %ld\n", actual->dmin);
                    printf("Average: %2f\n", (float)(actual->tdist) / (actual->count - 1));
                    printf("Largest: %ld\n\n", actual->dmax);

                    fprintf(f_link_all, "%ld \t %.2f \t %ld \n ", actual->dmin, (float)(actual->tdist) / (actual
    ->count - 1), actual->dmax);
                }
                else
                {
                    fprintf(f_link_all, "%d \t %.2f \t %d \n ", 0, 0.00, 0);
                    printf("\nNo distances stats available.\n\n");
                }

                actual = actual->next;
            }
        }
        fclose(f_link_all);
    }
    if (!found)
    {
        printf("No words found!\n");
        exit(0);
    }
    fflush(stdin);
    return c_st;
}

int open_text_file(char *file_name, file_data_t *fd)
{
    fd->fp = fopen(file_name, "rb");

    if (fd->fp == NULL)
        return -1;
    fd->word_pos = 0;
    fd->word_num = 0;
    fd->word[0] = '\0';
    fd->current_pos = -1;
    return 0;
}

void close_text_file(file_data_t *fd)
{
    fclose(fd->fp);
    fd->fp = NULL;
}

int read_word(file_data_t *fd)
{
    int i, c;
    // skip white spaces
    do
    {
        c = fgetc(fd->fp);
        if (c == EOF)
            return -1;
        fd->current_pos++;
    } while (c <= 32);
    // record word
```

```
778     fd->word_pos = fd->current_pos;
779     fd->word_num++;
780     fd->word[0] = (char)c;
781     for (i = 1; i < (int)sizeof(fd->word) - 1; i++)
782     {
783         c = fgetc(fd->fp);
784         if (c == EOF)
785             break; // end of file
786         fd->current_pos++;
787         if (c <= 32)
788             break; // terminate word
789         fd->word[i] = (char)c;
790     }
791     fd->word[i] = '\0';
792     return 0;
793 }
794 
795 void usage(char *argv[])
796 {
797     printf("Unknown option\n");
798     printf("\nUsage: %s -l -b -t\n\n", argv[0]);
799     printf("-l Initialize program using HashTable with Linked List\n");
800     printf("-b Initialize program using HashTable with Ordered Binary Tree\n");
801     printf("-t Initialize program for Tests\n");
802 
803     exit(0);
804 }
805 
806 int main(int argc, char *argv[])
807 {
808     if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'l')
809     {
810         count_array = 0;
811         printf("Initializing HashTable with Linked List\n");
812         int s_hash = 500;
813         link_ele **words = (link_ele *)calloc(s_hash, sizeof(link_ele *)); // creates and announce them as zero (
     null)
814         file_data_t *f = malloc(sizeof(file_data_t));
815         char file[64];
816         printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
817         scanf("%[^\n]", file);
818         fflush(stdin);
819         if (!open_text_file(file, f))
820         {
821             while (!read_word(f))
822             {
823                 if ((double)count_array / s_hash >= 0.8)
824                 {
825                     words = resize_link(words, &s_hash);
826                 }
827                 add_ele(words, f, s_hash);
828             }
829             printf("File read successfully!\n");
830             close_text_file(f);
831         }
832         else
833         {
834             printf("-------------------\n");
835             printf("Error opening file!\n");
836             printf("-------------------\n");
837             exit(0);
838         }
```

```
839         printf("\n1 - Search for a certain word stats\n2 - Search with a piece of a word or list all words stats\
    n");
840         char option[5];
841         printf("\nOption: ");
842         scanf("%[^\n]", option);
843         fflush(stdin);
844         if (strcmp(option, "1") == 0)
845             get_info_link(words, s_hash);
846         else if (strcmp(option, "2") == 0)
847             get_info_link_all(words, s_hash, false);
848         else
849         {
850             printf("Invalid option");
851             exit(0);
852         }
853     }
854     else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'b')
855     {
856         printf("Initializing HashTable with Ordered Binary Tree\n");
857         int s_hash = 500;
858         tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and announce them as
    zero (null)
859         file_data_t *f = malloc(sizeof(file_data_t));
860         char file[64];
861         printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
862         scanf("%[^\n]", file);
863         fflush(stdin);
864         if (!open_text_file(file, f))
865         {
866             while (!read_word(f))
867             {
868                 add_node(words, f, s_hash);
869             }
870             printf("File read successfully!\n");
871             close_text_file(f);
872         }
873         else
874         {
875             printf("--------------------\n");
876             printf("Error opening file!\n");
877             printf("--------------------\n");
878             exit(0);
879         }
880         printf("\n1 - Search for a certain word stats\n2 - Show all words stats\n");
881         char option[5];
882         printf("\nOption: ");
883         scanf("%[^\n]", option);
884         fflush(stdin);
885         if (strcmp(option, "1") == 0)
886             get_info_node(words, s_hash);
887         else if (strcmp(option, "2") == 0)
888             get_info_node_all(words, s_hash);
889         else
890         {
891             printf("Invalid option");
892             exit(0);
893         }
894     }
895     else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 't')
896     {
897         char file[64];
898         printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
```

```
899         scanf("%[^\n]", file);
900         fflush(stdin);
901 
902         printf("Initializing HashTable with Ordered Binary Tree\n");
903         reset_time();
904         int s_hash = 500;
905         int count_stored = 0;
906         (void)elapsed_time();
907         tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and anounce them as zero
    (null)
908         file_data_t *f = malloc(sizeof(file_data_t));
909 
910         if (!open_text_file(file, f))
911         {
912             while (!read_word(f))
913             {
914                 add_node(words, f, s_hash);
915             }
916             printf("File read successfully!\n");
917             close_text_file(f);
918         }
919         else
920         {
921             printf("-----------------\n");
922             printf("Error opening file!\n");
923             printf("-----------------\n");
924             exit(0);
925         }
926 
927         cpu_time = elapsed_time();
928         printf("%s %.6f s \n", "File read! Elapsed Time! - Reading", cpu_time);
929 
930         FILE *fw = fopen("results.txt", "a+");
931 
932         if (fw == NULL)
933         {
934             printf("Erro a abrir o ficheiro escrita!\n");
935             exit(1);
936         }
937         else
938         {
939             printf("%s\n", "Aberto ficheiro results.txt");
940             fprintf(fw, "Filename \t %s \n", file);
941             fprintf(fw, "HashTable OBT Reading Time \t %.6f \n", cpu_time);
942         }
943 
944         reset_time();
945 
946         printf("\nPrinting all words stored...\n");
947         (void)elapsed_time();
948         usleep(5000000);
949         count_stored = get_info_node_all(words, s_hash);
950         printf("\n ---------------------------------------------------------------- \n");
951         printf("\n Words read - %ld\n", f->word_num);
952         printf(" Words stored - %d\n", count_stored);
953         printf("%s %d \n", "Number of different word", count_diff);
954         cpu_time = elapsed_time();
955         printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);
956 
957         if (fw == NULL)
958         {
959             printf("Erro a abrir o ficheiro results!\n");
```

```
960             exit(1);
961         }
962         else
963         {
964             fprintf(fw, "HashTable OBT Words Read \t %ld \n", f->word_num);
965             fprintf(fw, "HashTable OBT Words Stored \t %d \n", count_stored);
966             fprintf(fw, "%s %d \n", "Number of different word", count_diff);
967             fprintf(fw, "HashTable OBT Time Travel Print \t %.6f \n", cpu_time);
968         }
969 
970         free(words);
971         free(f);
972         //-------------------------------------------------------------//
973 
974         printf("\n.....\n");
975 
976         printf("\nInitializing HashTable with Linked List\n");
977         s_hash = 500;
978         count_array = 0;
979         count_stored = 0;
980         reset_time();
981         (void)elapsed_time();
982         link_ele **words1 = (link_ele *)calloc(s_hash, sizeof(link_ele *)); // creates and anounce them as zero (
    null)
983         file_data_t *f1 = malloc(sizeof(file_data_t));
984 
985         if (!open_text_file(file, f1))
986         {
987             while (!read_word(f1))
988             {
989                 if ((double)count_array / s_hash >= 0.8)
990                 {
991                     words1 = resize_link(words1, &s_hash);
992                 }
993                 add_ele(words1, f1, s_hash);
994             }
995             printf("File read successfully!\n");
996             close_text_file(f);
997         }
998         else
999         {
1000            printf("-----------------\n");
1001            printf("Error opening file!\n");
1002            printf("-----------------\n");
1003            exit(0);
1004        }
1005 
1006        cpu_time = elapsed_time();
1007        printf("%s %.6f s \n", "File read! Elapsed Time! - Reading", cpu_time);
1008 
1009        if (fw == NULL)
1010        {
1011            printf("Erro a abrir o ficheiro escrita!\n");
1012            exit(1);
1013        }
1014        else
1015        {
1016            fprintf(fw, "Filename \t %s \n", file);
1017            fprintf(fw, "HashTable LL Reading Time \t %.6f \n", cpu_time);
1018        }
1019 
1020        reset_time();
```

```
1021
1022            printf("\nPrinting all words stored...\n");
1023            (void)elapsed_time();
1024            usleep(5000000);
1025            count_stored = get_info_link_all(words1, s_hash, true);
1026            printf("\n ---------------------------------------------------------------- \n");
1027            printf("\n Words read - %ld\n", f1->word_num);
1028            printf(" Words stored - %d\n", count_stored);
1029            printf("%s %d \n", "Number of different word", count_diff);
1030            cpu_time = elapsed_time();
1031            printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);
1032
1033            if (fw == NULL)
1034            {
1035                printf("Erro a abrir o ficheiro results!\n");
1036                exit(1);
1037            }
1038            else
1039            {
1040                fprintf(fw, "HashTable LL Words Read \t %ld \n", f->word_num);
1041                fprintf(fw, "HashTable LL Words Stored \t %d \n", count_stored);
1042                fprintf(fw, "%s %d \n", "Number of different word", count_diff);
1043                fprintf(fw, "HashTable LL Time Travel Print \t %.6f \n", cpu_time);
1044            }
1045
1046            fclose(fw);
1047            free(words1);
1048            free(f1);
1049        }
1050        else
1051        {
1052            usage(argv);
1053        }
1054    }
```