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
31-12-2019

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 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, all hash-codes were generated and words stored on that new array. After that, the new array is simply returned.

```
1
2  link_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             int index = hash_function(words[i]->word, newsize);
11             words_temp[index] = words[i];
12         }
13     }
14     *size = 2 * (*size);
15     return words_temp;
16 }
```

2.7 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
9      int index = hash_function(name, size);
10     link_ele *actual = words[index];
11     bool found = false;
12     if (actual != NULL)
13     {
14         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);
    }
}
```

2.8 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++)
54         {
55             link_ele *actual = words[i];
56             while (actual != NULL)
57             {
58                 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;
}
```

2.9 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.

```

void get_info_node(tree_node **words, int size)
{
```

```
    char name[64];
    printf("Insert word for info: ");
    scanf("%[^\n]", name);
    fflush(stdin);
    //get info about a word
    int index = hash_function(name, size);
    tree_node *actual = words[index];
    bool found = false;
    if (actual != NULL)
    {
        while (actual != NULL)
        {
            if (strcmp(name, actual->word) < 0 && actual->left != NULL) // word smaller than the node
                actual = actual->left;

            else if (strcmp(name, actual->word) > 0 && actual->right != NULL) // word bigger than the node
                actual = actual->right;

            else
            { // if equal
                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("\n No distances stats available.\n\n");
                }
                found = true;
                break;
            }
        }
    }
    if (!found)
    {
        printf("Word %s not found!\n", name);
        exit(0);
    }
}
```

2.10 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)
```

```
                        pre = pre->right;

                    /* Make current as the right child of its inorder
               predecessor */
                    if (pre->right == NULL)
                    {
                        pre->right = actual;
                        actual = actual->left;
                    }

                    /* Revert the changes made in the 'if' part to restore
               the original tree i.e., fix the right child
               of predecessor */
                    else
                    {
                        pre->right = NULL;
                        c_stored += 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("\n No distances stats available.\n\n");
                        }
                        found = true;
                        actual = actual->right;
                    }
                }
            }
        }
    }
    if (!found)
    {
        printf("No words found!\n");
        exit(0);
    }
    return c_stored;
}
```

2.11 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                     count_array = 0;
23                 }
24                 add_ele(words, f, s_hash);
25             }
26             printf("File read successfully!\n");
27             close_text_file(f);
28         }
29         else
30         {
31             printf("-----------------\n");
32             printf("Error opening file!\n");
33             printf("-----------------\n");
34             exit(0);
35         }
36         printf("\n1 - Search for a certain word stats\n2 - Search with a piece of a word or list all words stats\
       n");
37         char option[5];
```

```
38          printf("\nOption: ");
39          scanf("%[^\n]", option);
40          fflush(stdin);
41          if (strcmp(option, "1") == 0)
42              get_info_link(words, s_hash);
43          else if (strcmp(option, "2") == 0)
44              get_info_link_all(words, s_hash, false);
45          else
46          {
47              printf("Invalid option");
48              exit(0);
49          }
50      }
51      else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'b')
52      {
53          printf("Initializing HashTable with Ordered Binary Tree\n");
54          int s_hash = 500;
55          tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and announce them as
    zero (null)
56          file_data_t *f = malloc(sizeof(file_data_t));
57          char file[64];
58          printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
59          scanf("%[^\n]", file);
60          fflush(stdin);
61          if (!open_text_file(file, f))
62          {
63              while (!read_word(f))
64              {
65                  add_node(words, f, s_hash);
66              }
67              printf("File read successfully!\n");
68              close_text_file(f);
69          }
70          else
71          {
72              printf("-----------------\n");
73              printf("Error opening file!\n");
74              printf("-----------------\n");
75              exit(0);
76          }
77          printf("\n1 - Search for a certain word stats\n2 - Show all words stats\n");
78          char option[5];
79          printf("\nOption: ");
80          scanf("%[^\n]", option);
81          fflush(stdin);
82          if (strcmp(option, "1") == 0)
83              get_info_node(words, s_hash);
84          else if (strcmp(option, "2") == 0)
85              get_info_node_all(words, s_hash);
86          else
87          {
88              printf("Invalid option");
89              exit(0);
90          }
91      }
92      else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 't')
93      {
94          char file[64];
95          printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
96          scanf("%[^\n]", file);
97          fflush(stdin);
98
```

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

```
160         else
161         {
162             fprintf(fw, "HashTable OBT Words Read \t %ld \n", f->word_num);
163             fprintf(fw, "HashTable OBT Words Stored \t %d \n", count_stored);
164             fprintf(fw, "%s %d \n", "Number of different word", count_diff);
165             fprintf(fw, "HashTable OBT Time Travel Print \t %.6f \n", cpu_time);
166 
167         }
168 
169         free(words);
170         free(f);
171         //-------------------------------------------------------------//
172 
173         printf("\n.....\n");
174 
175         printf("\nInitializing HashTable with Linked List\n");
176         s_hash = 500;
177         count_array = 0;
178         count_stored = 0;
179         reset_time();
180         (void)elapsed_time();
181         link_ele **words1 = (link_ele *)calloc(s_hash, sizeof(link_ele *)); // creates and anounce them as zero (
    null)
182         file_data_t *f1 = malloc(sizeof(file_data_t));
183 
184         if (!open_text_file(file, f1))
185         {
186             while (!read_word(f1))
187             {
188                 if ((double)count_array / s_hash >= 0.8)
189                 {
190                     words1 = resize_link(words1, &s_hash);
191                     count_array = 0;
192                 }
193                 add_ele(words1, f1, s_hash);
194             }
195             printf("File read successfully!\n");
196             close_text_file(f);
197         }
198         else
199         {
200             printf("-----------------\n");
201             printf("Error opening file!\n");
202             printf("-----------------\n");
203             exit(0);
204         }
205 
206         cpu_time = elapsed_time();
207         printf("%s %.6f s \n", "File read! Elapsed Time! - Reading", cpu_time);
208 
209         if (fw == NULL)
210         {
211             printf("Erro a abrir o ficheiro escrita!\n");
212             exit(1);
213         }
214         else
215         {
216             fprintf(fw, "Filename \t %s \n", file);
217             fprintf(fw, "HashTable LL Reading Time \t %.6f \n", cpu_time);
218         }
219 
220         reset_time();
```

```
221
222         printf("\nPrinting all words stored...\n");
223         (void)elapsed_time();
224         usleep(5000000);
225         count_stored = get_info_link_all(words1, s_hash, true);
226         printf("\n ---------------------------------------------------------------- \n");
227         printf("\n Words read - %ld\n", f1->word_num);
228         printf(" Words stored - %d\n", count_stored);
229         printf("%s %d \n", "Number of different word", count_diff);
230         cpu_time = elapsed_time();
231         printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);
232
233         if (fw == NULL)
234         {
235             printf("Erro a abrir o ficheiro results!\n");
236             exit(1);
237         }
238         else
239         {
240             fprintf(fw, "HashTable LL Words Read \t %ld \n", f->word_num);
241             fprintf(fw, "HashTable LL Words Stored \t %d \n", count_stored);
242             fprintf(fw, "%s %d \n", "Number of different word", count_diff);
243             fprintf(fw, "HashTable LL Time Travel Print \t %.6f \n", cpu_time);
244
245
246         }
247
248
249         fclose(fw);
250         free(words1);
251         free(f1);
252
253
254     }
255     else
256     {
257         usage(argv);
258     }
259 }
```

3 Results

After running the program using the -t option, explained previously, the following results were obtained:

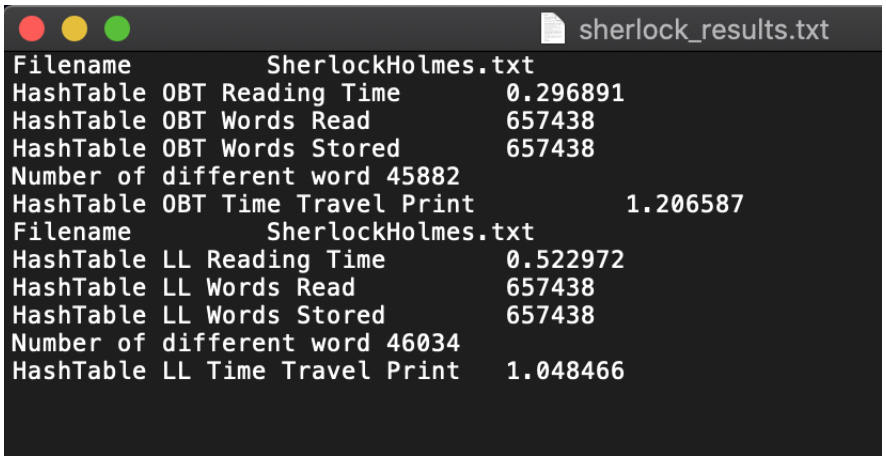

Figure 1: Results after running the program using the -t option and reading from the file *SherlockHolmes.txt*

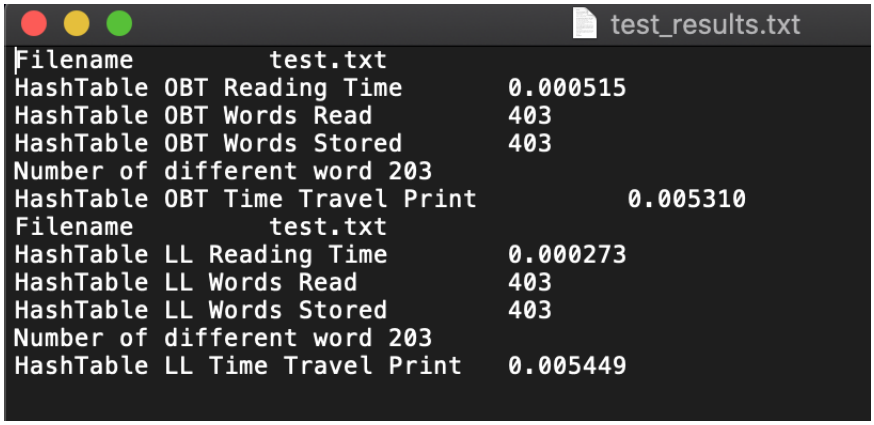

Figure 2: Results after running the program using the -t option and reading from the file *test.txt*

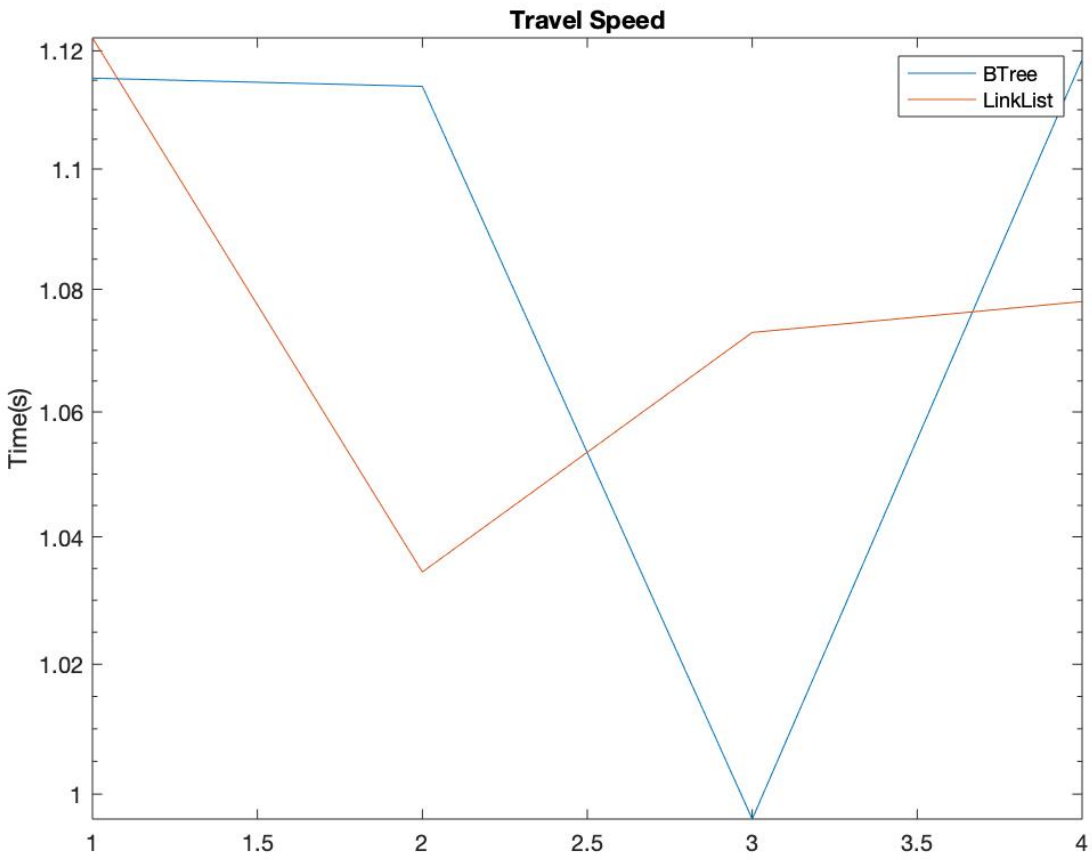

Figure 3: Time spent while travelling through all the words from the *SherlockHolmes.txt* file, according to the number of times the program has ran and for both, Liked List and Ordered Binary Tree structs

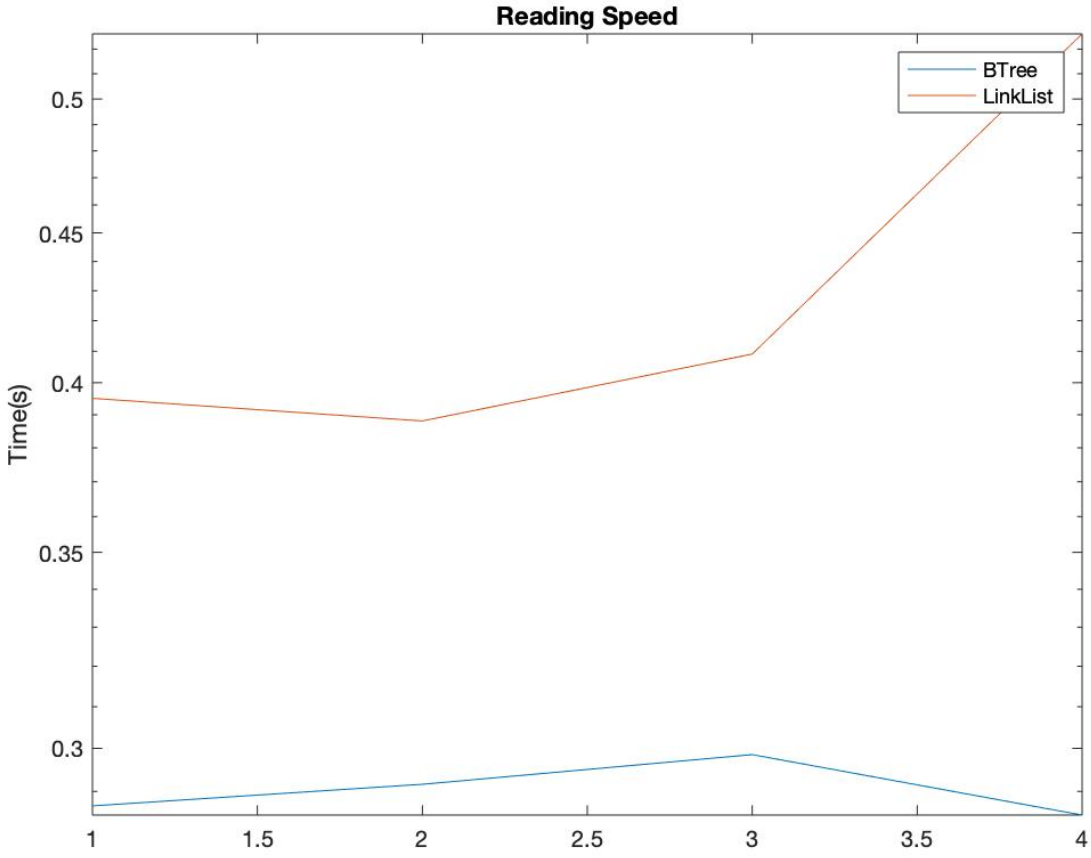

Figure 4: Time spent while reading all the words from the *SherlockHolmes.txt* file, according to the number of times the program has ran and for both, Liked List and Ordered Binary Tree structs

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

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

Note that, somehow, the number of different words using Ordered Binary Trees and Linked Lists are not the same. This only happens to the *SherlockHolmes.txt* file. Different files were used, like *test.txt* and the number was the same.

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 5: 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):
```

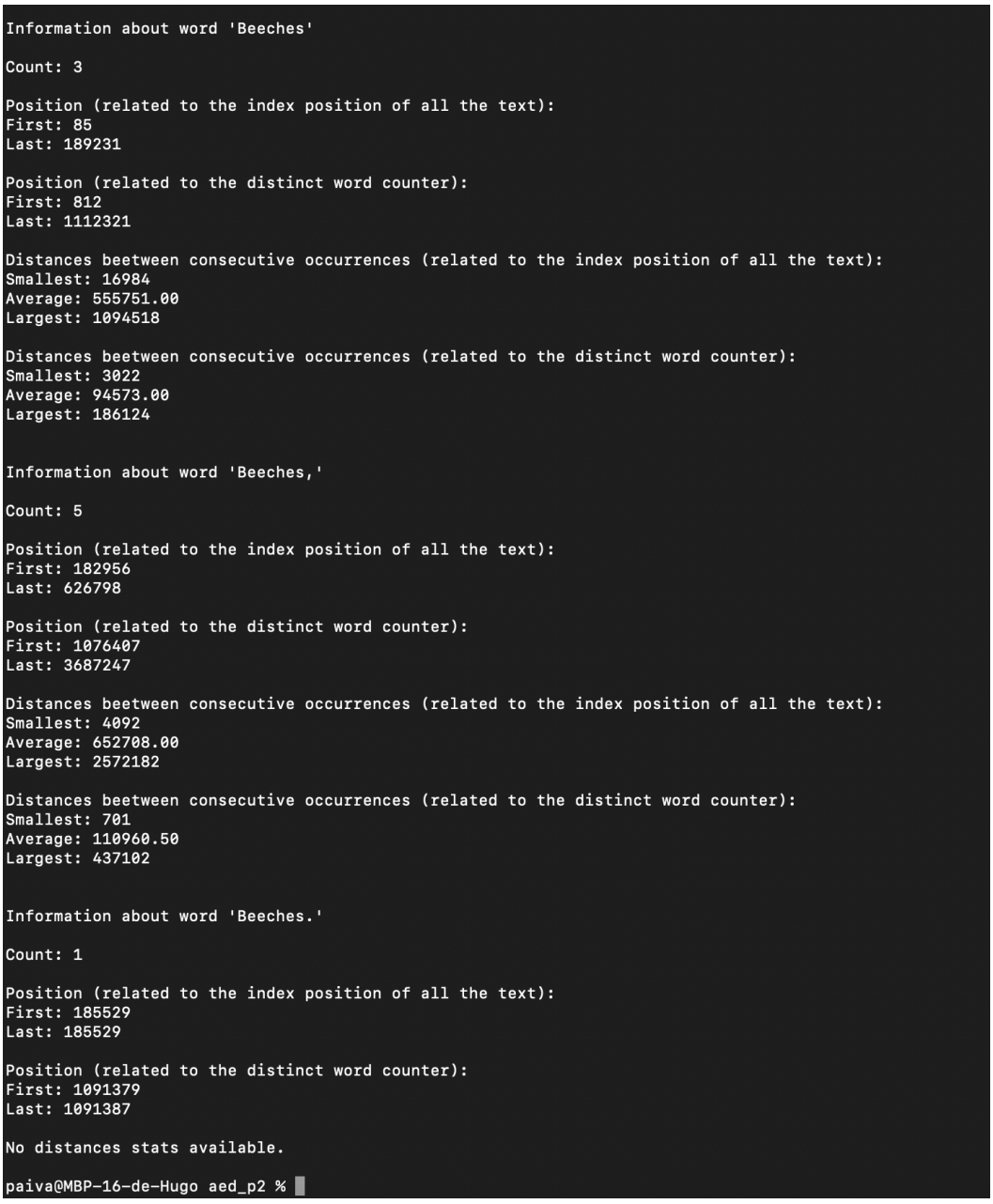

Figure 6: Running the program with the option -l and search by the start of a word

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

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.

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>

static double elapsed_time(void)
{
  static LARGE_INTEGER frequency, last_time, current_time;
  static int first_time = 1;

  if (first_time != 0)
  {
    QueryPerformanceFrequency(&frequency);
    first_time = 0;
  }
  last_time = current_time;
  QueryPerformanceCounter(&current_time);
  return (double)(current_time.QuadPart - last_time.QuadPart) / (double)frequency.QuadPart;
}

#endif

static void reset_time(void)
{
    printf("%s\n", "Time reseted");
    cpu_time = 0.0;
}




typedef struct file_data
{                 // public data
    long word_pos; // zero-based
    long word_num; // zero-based
    char word[64];
    // private data
    FILE *fp;
    long current_pos; // zero-based
} file_data_t;

typedef struct link_ele
{
    char word[64];
    long count;           // word counter
    long tdist;           // total sum of distances (in relation to the general word counter)
    long tdistp;          // total sum of distances (in relation to the index position)
    long dmin;            // min distance (in relation to the general word counter)
    long dmax;            // max distance (in relation to the general word counter)
    long dminp;           // min distance (in relation to the index position)
    long dmaxp;           // max distance (in relation to the index position)
    long last;            // last position (in relation to the general word counter)
    long first;           // first position (in relation to the general word counter)
    long lastp;           // last position (in relation to the index position)
    long firstp;          // first position (in relation to the index position)
    struct link_ele *next; // next word pointer
} link_ele;

typedef struct tree_node
{
    struct tree_node *left;   // pointer to the left branch (a sub-tree)
    struct tree_node *right;  // pointer to the right branch (a sub-tree)
    struct tree_node *parent; // optional
    char word[64];
    long count;  // word counter
```

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

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

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

```
306                 temp->next = NULL;
307                 temp->dmin = plus_inf;   // dist not altered
308                 temp->dmax = minus_inf;  // dist not altered
309                 temp->dminp = plus_inf;  // dist not altered
310                 temp->dmaxp = minus_inf; // dist not altered
311                 actual->next = temp;
312             }
313         }
314     }
315     else
316     {   // New Start of a linked list
317         count_array++;
318         link_ele *new = malloc(sizeof(link_ele));
319         strcpy(new->word, f->word);
320         new->next = NULL;
321         new->count = 0;
322         new->dmin = plus_inf;   // dist not altered
323         new->dmax = minus_inf;  // dist not altered
324         new->dminp = plus_inf;  // dist not altered
325         new->dmaxp = minus_inf; // dist not altered
326         new->first = f->word_num;
327         new->count++;
328         new->last = f->word_num;
329         new->lastp = f->current_pos;
330         new->firstp = f->word_pos;
331         words[index] = new;
332     }
333 }
334
335 link_ele **resize_link(link_ele **words, int *size)
336 {
337     int newsize = 2 * (*size);
338     link_ele **words_temp = (link_ele *)calloc(newsize, sizeof(link_ele *));
339     for (int i = 0; i < (*size); i++)
340     {
341         if (words[i] != NULL)
342         {
343             int index = hash_function(words[i]->word, newsize);
344             words_temp[index] = words[i];
345         }
346     }
347     *size = 2 * (*size);
348     return words_temp;
349 }
350
351 void get_info_link(link_ele **words, int size)
352 {
353     char name[64];
354     printf("Insert word for info: ");
355     scanf("%[^\n]", name);
356     fflush(stdin);
357     //get info about a word
358     int index = hash_function(name, size);
359     link_ele *actual = words[index];
360     bool found = false;
361     if (actual != NULL)
362     {
363         while (actual != NULL)
364         {
365             if (strcmp(actual->word, name) == 0)
366             {
367                 printf("\nInformation about word '%s'\n", actual->word);
```

```
368                 printf("\nCount: %ld\n", actual->count);
369                 printf("\nPosition (related to the index position of all the text):\n");
370                 printf("First: %ld\n", actual->first);
371                 printf("Last: %ld\n", actual->last);
372                 printf("\nPosition (related to the distinct word counter):\n");
373                 printf("First: %ld\n", actual->firstp);
374                 printf("Last: %ld\n", actual->lastp);
375                 if (actual->count > 1)
376                 {
377                     printf("\nDistances beetween consecutive occurrences (related to the index position of all
    the text):\n");
378                     printf("Smallest: %ld\n", actual->dminp);
379                     printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 number of
    dist and not words
380                     printf("Largest: %ld\n", actual->dmaxp);
381                     printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\
    n");
382                     printf("Smallest: %ld\n", actual->dmin);
383                     printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
384                     printf("Largest: %ld\n\n", actual->dmax);
385                 }
386                 else
387                 {
388                     printf("\n No distances stats available.\n\n");
389                 }
390                 found = true;
391                 break;
392             }
393             actual = actual->next;
394         }
395     }
396 
397     if (!found)
398     {
399         printf("Word %s not found!\n", name);
400         exit(0);
401     }
402 }
403 
404 void get_info_node(tree_node **words, int size)
405 {
406     char name[64];
407     printf("Insert word for info: ");
408     scanf("%[^\n]", name);
409     fflush(stdin);
410     //get info about a word
411     int index = hash_function(name, size);
412     tree_node *actual = words[index];
413     bool found = false;
414     if (actual != NULL)
415     {
416         while (actual != NULL)
417         {
418             if (strcmp(name, actual->word) < 0 && actual->left != NULL) // word smaller than the node
419                 actual = actual->left;
420 
421             else if (strcmp(name, actual->word) > 0 && actual->right != NULL) // word bigger than the node
422                 actual = actual->right;
423 
424             else
425             { // if equal
426                 printf("\nInformation about word '%s'\n", actual->word);
```

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

```
486                     if (actual->count > 1)
487                     {
488                         printf("\nDistances beetween consecutive occurrences (related to the index position of
    all the text):\n");
489                         printf("Smallest: %ld\n", actual->dminp);
490                         printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because
    number of distances and not words
491                         printf("Largest: %ld\n", actual->dmaxp);
492                         printf("\nDistances beetween consecutive occurrences (related to the distinct word
    counter):\n");
493                         printf("Smallest: %ld\n", actual->dmin);
494                         printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
495                         printf("Largest: %ld\n\n", actual->dmax);
496                     }
497                     else
498                     {
499                         printf("\n No distances stats available.\n\n");
500                     }
501                     found = true;
502                     actual = actual->right;
503                 }
504                 else
505                 {
506                     /* Find the inorder predecessor of current */
507                     pre = actual->left;
508                     while (pre->right != NULL && pre->right != actual)
509                         pre = pre->right;
510 
511                     /* Make current as the right child of its inorder
512                 predecessor */
513                     if (pre->right == NULL)
514                     {
515                         pre->right = actual;
516                         actual = actual->left;
517                     }
518 
519                     /* Revert the changes made in the 'if' part to restore
520                 the original tree i.e., fix the right child
521                 of predecessor */
522                     else
523                     {
524                         pre->right = NULL;
525                         c_stored += actual->count;
526                         count_diff++;
527                         printf("\nInformation about word '%s'\n", actual->word);
528                         printf("\nCount: %ld\n", actual->count);
529                         printf("\nPosition (related to the index position of all the text):\n");
530                         printf("First: %ld\n", actual->first);
531                         printf("Last: %ld\n", actual->last);
532                         printf("\nPosition (related to the distinct word counter):\n");
533                         printf("First: %ld\n", actual->firstp);
534                         printf("Last: %ld\n", actual->lastp);
535                         if (actual->count > 1)
536                         {
537                             printf("\nDistances beetween consecutive occurrences (related to the index position
    of all the text):\n");
538                             printf("Smallest: %ld\n", actual->dminp);
539                             printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1
    because number of distances and not words
540                             printf("Largest: %ld\n", actual->dmaxp);
541                             printf("\nDistances beetween consecutive occurrences (related to the distinct word
    counter):\n");
```

```
542                             printf("Smallest: %ld\n", actual->dmin);
543                             printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
544                             printf("Largest: %ld\n\n", actual->dmax);
545                         }
546                         else
547                         {
548                             printf("\n No distances stats available.\n\n");
549                         }
550                         found = true;
551                         actual = actual->right;
552                     }
553                 }
554             }
555         }
556     }
557     if (!found)
558     {
559         printf("No words found!\n");
560         exit(0);
561     }
562     return c_stored;
563 }
564 
565 int get_info_link_all(link_ele **words, int size, bool all)
566 {
567     bool found = false;
568     int c_st = 0;
569     count_diff = 0;
570     if (all == true)
571         goto all;
572     char name[64];
573     printf("Insert word, or start of it, for info (empty for all): ");
574     if (gets(name) != NULL)
575     {
576         int s_name = strlen(name);
577         for (int i = 0; i < size; i++)
578         {
579             link_ele *actual = words[i];
580             while (actual != NULL)
581             {
582                 if (strncmp(name, actual->word, s_name) == 0)
583                 {
584                     found = true;
585                     printf("\nInformation about word '%s'\n", actual->word);
586                     printf("\nCount: %ld\n", actual->count);
587                     printf("\nPosition (related to the index position of all the text):\n");
588                     printf("First: %ld\n", actual->first);
589                     printf("Last: %ld\n", actual->last);
590                     printf("\nPosition (related to the distinct word counter):\n");
591                     printf("First: %ld\n", actual->firstp);
592                     printf("Last: %ld\n", actual->lastp);
593                     if (actual->count > 1)
594                     {
595                         printf("\nDistances beetween consecutive occurrences (related to the index position of
    all the text):\n");
596                         printf("Smallest: %ld\n", actual->dminp);
597                         printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because
    number of distances and not words
598                         printf("Largest: %ld\n", actual->dmaxp);
599                         printf("\nDistances beetween consecutive occurrences (related to the distinct word
    counter):\n");
600                         printf("Smallest: %ld\n", actual->dmin);
```

```
601                         printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
602                         printf("Largest: %ld\n\n", actual->dmax);
603                     }
604                     else
605                     {
606                         printf("\nNo distances stats available.\n\n");
607                     }
608                 }
609                 actual = actual->next;
610             }
611         }
612     }
613     else
614     {
615     all:
616         for (int i = 0; i < size; i++)
617         {
618             link_ele *actual = words[i];
619             while (actual != NULL)
620             {
621                 found = true;
622                 c_st += actual->count;
623                 count_diff++;
624                 printf("\nInformation about word '%s'\n", actual->word);
625                 printf("\nCount: %ld\n", actual->count);
626                 printf("\nPosition (related to the index position of all the text):\n");
627                 printf("First: %ld\n", actual->first);
628                 printf("Last: %ld\n", actual->last);
629                 printf("\nPosition (related to the distinct word counter):\n");
630                 printf("First: %ld\n", actual->firstp);
631                 printf("Last: %ld\n", actual->lastp);
632                 if (actual->count > 1)
633                 {
634                     printf("\nDistances beetween consecutive occurrences (related to the index position of all
    the text):\n");
635                     printf("Smallest: %ld\n", actual->dminp);
636                     printf("Average: %2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because number
     of distances and not words
637                     printf("Largest: %ld\n", actual->dmaxp);
638                     printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\
    n");
639                     printf("Smallest: %ld\n", actual->dmin);
640                     printf("Average: %2f\n", (float)(actual->tdist) / (actual->count - 1));
641                     printf("Largest: %ld\n\n", actual->dmax);
642                 }
643                 else
644                 {
645                     printf("\nNo distances stats available.\n\n");
646                 }
647
648                 actual = actual->next;
649             }
650         }
651     }
652     if (!found)
653     {
654         printf("No words found!\n");
655         exit(0);
656     }
657     fflush(stdin);
658     return c_st;
659 }
```

```
660
661 int open_text_file(char *file_name, file_data_t *fd)
662 {
663     fd->fp = fopen(file_name, "rb");
664
665     if (fd->fp == NULL)
666         return -1;
667     fd->word_pos = 0;
668     fd->word_num = 0;
669     fd->word[0] = '\0';
670     fd->current_pos = -1;
671     return 0;
672 }
673
674 void close_text_file(file_data_t *fd)
675 {
676     fclose(fd->fp);
677     fd->fp = NULL;
678 }
679
680 int read_word(file_data_t *fd)
681 {
682     int i, c;
683     // skip white spaces
684     do
685     {
686         c = fgetc(fd->fp);
687         if (c == EOF)
688             return -1;
689         fd->current_pos++;
690     } while (c <= 32);
691     // record word
692     fd->word_pos = fd->current_pos;
693     fd->word_num++;
694     fd->word[0] = (char)c;
695     for (i = 1; i < (int)sizeof(fd->word) - 1; i++)
696     {
697         c = fgetc(fd->fp);
698         if (c == EOF)
699             break; // end of file
700         fd->current_pos++;
701         if (c <= 32)
702             break; // terminate word
703         fd->word[i] = (char)c;
704     }
705     fd->word[i] = '\0';
706     return 0;
707 }
708
709 void usage(char *argv[])
710 {
711     printf("Unknown option\n");
712     printf("\nUsage: %s -l -b -t\n\n", argv[0]);
713     printf("-l Initialize program using HashTable with Linked List\n");
714     printf("-b Initialize program using HashTable with Ordered Binary Tree\n");
715     printf("-t Initialize program for Tests\n");
716
717     exit(0);
718 }
719
720 int main(int argc, char *argv[])
721 {
```

```
722     if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'l')
723     {
724         count_array = 0;
725         printf("Initializing HashTable with Linked List\n");
726         int s_hash = 500;
727         link_ele **words = (link_ele *)calloc(s_hash, sizeof(link_ele *)); // creates and announce them as zero (
    null)
728         file_data_t *f = malloc(sizeof(file_data_t));
729         char file[64];
730         printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
731         scanf("%[^\n]", file);
732         fflush(stdin);
733         if (!open_text_file(file, f))
734         {
735             while (!read_word(f))
736             {
737                 if ((double)count_array / s_hash >= 0.8)
738                 {
739                     words = resize_link(words, &s_hash);
740                     count_array = 0;
741                 }
742                 add_ele(words, f, s_hash);
743             }
744             printf("File read successfully!\n");
745             close_text_file(f);
746         }
747         else
748         {
749             printf("--------------------\n");
750             printf("Error opening file!\n");
751             printf("--------------------\n");
752             exit(0);
753         }
754         printf("\n1 - Search for a certain word stats\n2 - Search with a piece of a word or list all words stats\
    n");
755         char option[5];
756         printf("\nOption: ");
757         scanf("%[^\n]", option);
758         fflush(stdin);
759         if (strcmp(option, "1") == 0)
760             get_info_link(words, s_hash);
761         else if (strcmp(option, "2") == 0)
762             get_info_link_all(words, s_hash, false);
763         else
764         {
765             printf("Invalid option");
766             exit(0);
767         }
768     }
769     else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'b')
770     {
771         printf("Initializing HashTable with Ordered Binary Tree\n");
772         int s_hash = 500;
773         tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and announce them as
    zero (null)
774         file_data_t *f = malloc(sizeof(file_data_t));
775         char file[64];
776         printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
777         scanf("%[^\n]", file);
778         fflush(stdin);
779         if (!open_text_file(file, f))
780         {
```

```
781             while (!read_word(f))
782             {
783                 add_node(words, f, s_hash);
784             }
785             printf("File read successfully!\n");
786             close_text_file(f);
787         }
788         else
789         {
790             printf("-------------------\n");
791             printf("Error opening file!\n");
792             printf("-------------------\n");
793             exit(0);
794         }
795         printf("\n1 - Search for a certain word stats\n2 - Show all words stats\n");
796         char option[5];
797         printf("\nOption: ");
798         scanf("%[^\n]", option);
799         fflush(stdin);
800         if (strcmp(option, "1") == 0)
801             get_info_node(words, s_hash);
802         else if (strcmp(option, "2") == 0)
803             get_info_node_all(words, s_hash);
804         else
805         {
806             printf("Invalid option");
807             exit(0);
808         }
809     }
810     else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 't')
811     {
812         char file[64];
813         printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
814         scanf("%[^\n]", file);
815         fflush(stdin);
816 
817         printf("Initializing HashTable with Ordered Binary Tree\n");
818         reset_time();
819         int s_hash = 500;
820         int count_stored = 0;
821         (void)elapsed_time();
822         tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and anounce them as zero
      (null)
823         file_data_t *f = malloc(sizeof(file_data_t));
824 
825         if (!open_text_file(file, f))
826         {
827             while (!read_word(f))
828             {
829                 add_node(words, f, s_hash);
830             }
831             printf("File read successfully!\n");
832             close_text_file(f);
833         }
834         else
835         {
836             printf("-------------------\n");
837             printf("Error opening file!\n");
838             printf("-------------------\n");
839             exit(0);
840         }
841 
```

```
842         cpu_time = elapsed_time();
843         printf("%s %.6f s \n", "File read! Elapsed Time! - Reading", cpu_time);
844 
845         FILE *fw = fopen("results.txt", "a+");
846 
847         if (fw == NULL)
848         {
849             printf("Erro a abrir o ficheiro escrita!\n");
850             exit(1);
851         }
852         else
853         {
854             printf("%s\n", "Aberto ficheiro results.txt");
855             fprintf(fw, "Filename \t %s \n", file);
856             fprintf(fw, "HashTable OBT Reading Time \t %.6f \n", cpu_time);
857         }
858 
859         reset_time();
860 
861 
862         printf("\nPrinting all words stored...\n");
863         (void)elapsed_time();
864         usleep(5000000);
865         count_stored = get_info_node_all(words, s_hash);
866         printf("\n ---------------------------------------------------------------------- \n");
867         printf("\n Words read - %ld\n", f->word_num);
868         printf(" Words stored - %d\n", count_stored);
869         printf("%s %d \n", "Number of different word", count_diff);
870         cpu_time = elapsed_time();
871         printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);
872 
873         if (fw == NULL)
874         {
875             printf("Erro a abrir o ficheiro results!\n");
876             exit(1);
877         }
878         else
879         {
880             fprintf(fw, "HashTable OBT Words Read \t %ld \n", f->word_num);
881             fprintf(fw, "HashTable OBT Words Stored \t %d \n", count_stored);
882             fprintf(fw, "%s %d \n", "Number of different word", count_diff);
883             fprintf(fw, "HashTable OBT Time Travel Print \t %.6f \n", cpu_time);
884 
885         }
886 
887         free(words);
888         free(f);
889         //-------------------------------------------------------------//
890 
891         printf("\n.....................................................................\n");
892 
893         printf("\nInitializing HashTable with Linked List\n");
894         s_hash = 500;
895         count_array = 0;
896         count_stored = 0;
897         reset_time();
898         (void)elapsed_time();
899         link_ele **words1 = (link_ele *)calloc(s_hash, sizeof(link_ele *)); // creates and anounce them as zero (
    null)
900         file_data_t *f1 = malloc(sizeof(file_data_t));
901 
902         if (!open_text_file(file, f1))
```

```
903         {
904             while (!read_word(f1))
905             {
906                 if ((double)count_array / s_hash >= 0.8)
907                 {
908                     words1 = resize_link(words1, &s_hash);
909                     count_array = 0;
910                 }
911                 add_ele(words1, f1, s_hash);
912             }
913             printf("File read successfully!\n");
914             close_text_file(f);
915         }
916         else
917         {
918             printf("-----------------\n");
919             printf("Error opening file!\n");
920             printf("-----------------\n");
921             exit(0);
922         }
923 
924         cpu_time = elapsed_time();
925         printf("%s %.6f s \n", "File read! Elapsed Time! - Reading", cpu_time);
926 
927         if (fw == NULL)
928         {
929             printf("Erro a abrir o ficheiro escrita!\n");
930             exit(1);
931         }
932         else
933         {
934             fprintf(fw, "Filename \t %s \n", file);
935             fprintf(fw, "HashTable LL Reading Time \t %.6f \n", cpu_time);
936         }
937 
938         reset_time();
939 
940         printf("\nPrinting all words stored...\n");
941         (void)elapsed_time();
942         usleep(5000000);
943         count_stored = get_info_link_all(words1, s_hash, true);
944         printf("\n ---------------------------------------------------------------- \n");
945         printf("\n Words read - %ld\n", f1->word_num);
946         printf(" Words stored - %d\n", count_stored);
947         printf("%s %d \n", "Number of different word", count_diff);
948         cpu_time = elapsed_time();
949         printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);
950 
951         if (fw == NULL)
952         {
953             printf("Erro a abrir o ficheiro results!\n");
954             exit(1);
955         }
956         else
957         {
958             fprintf(fw, "HashTable LL Words Read \t %ld \n", f->word_num);
959             fprintf(fw, "HashTable LL Words Stored \t %d \n", count_stored);
960             fprintf(fw, "%s %d \n", "Number of different word", count_diff);
961             fprintf(fw, "HashTable LL Time Travel Print \t %.6f \n", cpu_time);
962 
963 
964         }
```

```
965
966
967             fclose(fw);
968             free(words1);
969             free(f1);
970
971
972         }
973         else
974         {
975             usage(argv);
976         }
977 }
```