Written Report | AED

Teachers:
Tomás Oliveira e Silva
João Manuel Rodrigues

2nd Pratical Work

Hash Table

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

DETI
Universidade de Aveiro

29-01-2020

Contents

1 Introduction

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

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

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

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

1.1 Getting Started

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

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

1.2 Prerequisites

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

1.3 Compiling

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

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

1.4 Running

Options:

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

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

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

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

2 Implementation

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

2.1 Structures

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

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

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

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

```c

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
    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)
    long data;   // the data item (we use an int here, but it can be anything)
} 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.

```

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

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.

```

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

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

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

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

```c
    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.

```c

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;
        new->left = NULL;
        new->right = 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;
    }
}
```

2.5 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.

```

void add_ele(link_ele **words, file_data_t *f, int size)
{
    int index = hash_function(f->word, size);
    link_ele *actual = words[index];
    if (actual != NULL) // if an element in the list already exists in that index
    {
        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;
        }
        else
        { // if not equal it is needed to run over all the elements
            bool found = false;
            while (actual->next != NULL)
            {
                actual = actual->next;
                if (strcmp(actual->word, f->word) == 0)
                { // if equal
                    long tempdist = f->word_num - actual->last;
                    long tempdistp = f->current_pos - actual->lastp;
                    actual->tdist = actual->tdist + tempdist;
                    actual->tdistp = actual->tdistp + tempdistp;
                    if (tempdist < actual->dmin)
                        actual->dmin = tempdist;
                    if (tempdist > actual->dmax)
                        actual->dmax = tempdist;
                    if (tempdistp < actual->dminp)
                        actual->dminp = tempdistp;
                    if (tempdistp > actual->dmaxp)
                        actual->dmaxp = tempdistp;
                    actual->count++;
                    actual->last = f->word_num;
                    actual->lastp = f->current_pos;
                    found = true;
                    break;
                }
            }
            if (!found) // not found verification
            {
                link_ele *temp = malloc(sizeof(link_ele));
                strcpy(temp->word, f->word);
                temp->first = f->word_num;
                temp->count = 1;
                temp->last = f->word_num;
                temp->lastp = f->current_pos;
                temp->firstp = f->word_pos;
                temp->next = NULL;
                temp->dmin = plus_inf;   // dist not altered
                temp->dmax = minus_inf;  // dist not altered
                temp->dminp = plus_inf;  // dist not altered
                temp->dmaxp = minus_inf; // dist not altered
                actual->next = temp;
            }
        }
    }
    else
    {   // New Start of a linked list
        count_array++;
        link_ele *new = malloc(sizeof(link_ele));
        strcpy(new->word, f->word);
        new->next = NULL;
        new->count = 0;
        new->dmin = plus_inf;   // dist not altered
        new->dmax = minus_inf;  // dist not altered
        new->dminp = plus_inf;  // dist not altered
        new->dmaxp = minus_inf; // dist not altered
        new->first = f->word_num;
        new->count++;
        new->last = f->word_num;
        new->lastp = f->current_pos;
        new->firstp = f->word_pos;
        words[index] = new;
    }
}
```

2.6 Add Element to list using resize

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

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

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

```

void add_ele_resize(link_ele **words, link_ele *f, int size)
{
    int index = hash_function(f->word, size);
    link_ele *actual = words[index];
    if (actual != NULL)
    {
        while (actual->next != NULL)
        {
            actual = actual->next;
        }
        link_ele *temp = malloc(sizeof(link_ele));
        strcpy(temp->word, f->word);
        temp->next = NULL;
        temp->count = f->count;
        temp->dmin = f->dmin;
        temp->dmax = f->dmax;
        temp->dminp = f->dminp;
        temp->dmaxp = f->dmaxp;
        temp->first = f->first;
        temp->count = f->count;
        temp->last = f->last;
        temp->lastp = f->lastp;
        temp->firstp = f->firstp;
        actual->next = temp;

    }
    else
    {
        link_ele *new = malloc(sizeof(link_ele));
        strcpy(new->word, f->word);
        new->next = NULL;
        new->count = f->count;
        new->dmin = f->dmin;
        new->dmax = f->dmax;
        new->dminp = f->dminp;
        new->dmaxp = f->dmaxp;
        new->first = f->first;
        new->count = f->count;
        new->last = f->last;
        new->lastp = f->lastp;
        new->firstp = f->firstp;
        words[index] = new;
    }
}
```

2.7 Resize

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

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

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

```

ink_ele **resize_link(link_ele **words, int *size)
{
    int newsize = 2 * (*size);
    link_ele **words_temp = (link_ele *)calloc(newsize, sizeof(link_ele *));
    for (int i = 0; i < (*size); i++)
    {
        if (words[i] != NULL)
        {
            link_ele *actual = words[i];
            add_ele_resize(words_temp, actual, newsize);
            while (actual->next != NULL)
            {
                actual = actual->next;
                add_ele_resize(words_temp, actual, newsize);

            }
        }
    }
    *size = 2 * (*size);
    return words_temp;
}
```

2.8 Get Info Link

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

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

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

```

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

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

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

2.9 Get Info Link All

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

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

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

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

```

int get_info_link_all(link_ele **words, int size, bool all)
{
    bool found = false;
    int c_st = 0;
    count_diff = 0;
    if (all == true)
        goto all;
    char name[64];
    printf("Insert word, or start of it, for info (empty for all): ");
    if (gets(name) != NULL)
    {
        int s_name = strlen(name);
        for (int i = 0; i < size; i++)
        {
            link_ele *actual = words[i];
            while (actual != NULL)
            {
                if (strncmp(name, actual->word, s_name) == 0)
                {
                    found = true;
                    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;
            }
        }
    }
    else
    {
    all:
        for (int i = 0; i < size; i++)
        {
            link_ele *actual = words[i];
            while (actual != NULL)
            {
                found = true;
                c_st += actual->count;
                count_diff++;
                printf("\nInformation about word '%s'\n", actual->word);
                printf("\nCount: %ld\n", actual->count);
                printf("\nPosition (related to the index position of all the text):\n");
                printf("First: %ld\n", actual->first);
                printf("Last: %ld\n", actual->last);
                printf("\nPosition (related to the distinct word counter):\n");
                printf("First: %ld\n", actual->firstp);
                printf("Last: %ld\n", actual->lastp);
                if (actual->count > 1)
                {
                    printf("\nDistances beetween consecutive occurrences (related to the index position of all the text):\n");
                    printf("Smallest: %ld\n", actual->dminp);
                    printf("Average: %2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because number of distances and not words
                    printf("Largest: %ld\n", actual->dmaxp);
                    printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\n");
                    printf("Smallest: %ld\n", actual->dmin);
                    printf("Average: %2f\n", (float)(actual->tdist) / (actual->count - 1));
                    printf("Largest: %ld\n\n", actual->dmax);
                }
                else
                {
                    printf("\nNo distances stats available.\n\n");
                }

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

2.10 Get Info Node

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

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

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

```c

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);
    }
}
```

2.11 Get Info Node All

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

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

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

```

int get_info_node_all(tree_node **words, int size)
{
    int c_stored = 0;
    count_diff = 0;
    bool found = false;
    for (int i = 0; i < size; i++)
    {
        tree_node *actual = words[i];
        tree_node *pre;
        if (actual != NULL)
        {
            while (actual != NULL)
            {
                if (actual->left == 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;
                }
                else
                {
                    /* Find the inorder predecessor of current */
                    pre = actual->left;
                    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;
}
```

2.12 Main

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

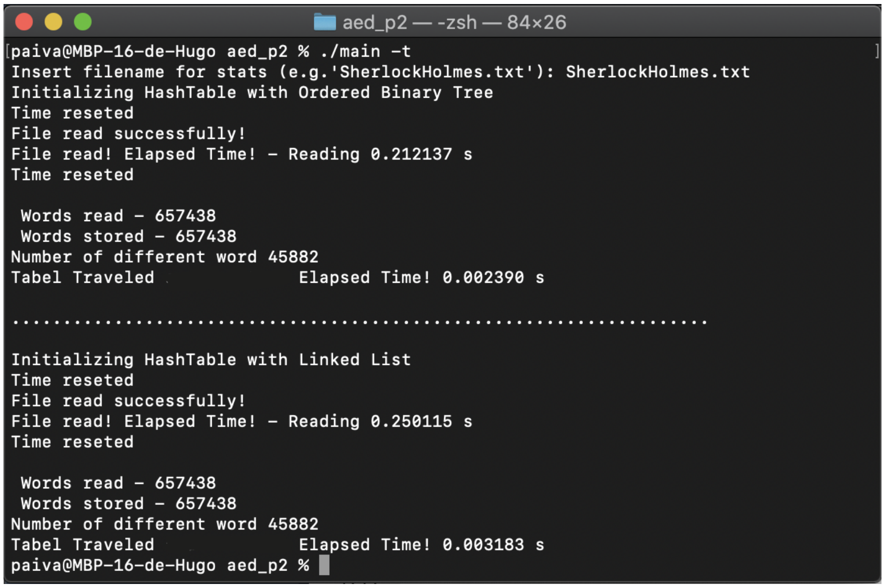
- Using the **-l** option will start the program creating a Hash Table with Linked Lists. The program will ask the user for a filename to be processed. Finally, the user may search and get information for a specific word, for words that start with the one inserted or simply list all the words.
- If the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

```c

int main(int argc, char *argv[])
{
    if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'l')
    {
        count_array = 0;
        printf("Initializing HashTable with Linked List\n");
        int s_hash = 500;
        link_ele **words = (link_ele *)calloc(s_hash, sizeof(link_ele *)); // creates and announce them as zero (null)
        file_data_t *f = malloc(sizeof(file_data_t));
        char file[64];
        printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
        scanf("%[^\n]", file);
        fflush(stdin);
        if (!open_text_file(file, f))
        {
            while (!read_word(f))
            {
                if ((double)count_array / s_hash >= 0.8)
                {
                    words = resize_link(words, &s_hash);
                }
                add_ele(words, f, s_hash);
            }
            printf("File read successfully!\n");
            close_text_file(f);
        }
        else
        {
            printf("-----------------\n");
            printf("Error opening file!\n");
            printf("-----------------\n");
            exit(0);
        }
        printf("\n1 - Search for a certain word stats\n2 - Search with a piece of a word or list all words stats\n");
        char option[5];
        printf("\nOption: ");
        scanf("%[^\n]", option);
        fflush(stdin);
        if (strcmp(option, "1") == 0)
            get_info_link(words, s_hash);
        else if (strcmp(option, "2") == 0)
            get_info_link_all(words, s_hash, false);
        else
        {
            printf("Invalid option");
            exit(0);
        }
    }
    else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'b')
    {
        printf("Initializing HashTable with Ordered Binary Tree\n");
        int s_hash = 500;
        tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and announce them as zero (null)
        file_data_t *f = malloc(sizeof(file_data_t));
        char file[64];
        printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
        scanf("%[^\n]", file);
        fflush(stdin);
        if (!open_text_file(file, f))
        {
            while (!read_word(f))
            {
                add_node(words, f, s_hash);
            }
            printf("File read successfully!\n");
            close_text_file(f);
        }
        else
        {
            printf("-----------------\n");
            printf("Error opening file!\n");
            printf("-----------------\n");
            exit(0);
        }
        printf("\n1 - Search for a certain word stats\n2 - Show all words stats\n");
        char option[5];
        printf("\nOption: ");
        scanf("%[^\n]", option);
        fflush(stdin);
        if (strcmp(option, "1") == 0)
            get_info_node(words, s_hash);
        else if (strcmp(option, "2") == 0)
            get_info_node_all(words, s_hash);
        else
        {
            printf("Invalid option");
            exit(0);
        }
    }
    else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 't')
    {
        char file[64];
        printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
        scanf("%[^\n]", file);
        fflush(stdin);

        printf("Initializing HashTable with Ordered Binary Tree\n");
        reset_time();
        int s_hash = 500;
        int count_stored = 0;
        (void)elapsed_time();
        tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and anounce them as zero (null)
        file_data_t *f = malloc(sizeof(file_data_t));

        if (!open_text_file(file, f))
        {
            while (!read_word(f))
            {
                add_node(words, f, s_hash);
            }
            printf("File read successfully!\n");
            close_text_file(f);
        }
        else
        {
            printf("--------------------\n");
            printf("Error opening file!\n");
            printf("--------------------\n");
            exit(0);
        }

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

        FILE *fw = fopen("results.txt", "a+");

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

        reset_time();

        printf("\nPrinting all words stored...\n");
        (void)elapsed_time();
        usleep(5000000);
        count_stored = get_info_node_all(words, s_hash);
        printf("\n ---------------------------------------------------------------------- \n");
        printf("\n Words read - %ld\n", f->word_num);
        printf(" Words stored - %d\n", count_stored);
        printf("%s %d \n", "Number of different word", count_diff);
        cpu_time = elapsed_time();
        printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);

        if (fw == NULL)
        {
            printf("Erro a abrir o ficheiro results!\n");
            exit(1);
        }
        else
        {
            fprintf(fw, "HashTable OBT Words Read \t %ld \n", f->word_num);
            fprintf(fw, "HashTable OBT Words Stored \t %d \n", count_stored);
            fprintf(fw, "%s %d \n", "Number of different word", count_diff);
            fprintf(fw, "HashTable OBT Time Travel Print \t %.6f \n", cpu_time);
        }

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

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

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

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

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

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

        reset_time();

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

```c
        count_stored = get_info_link_all(words1, s_hash, true);
        printf("\n ------------------------------------------------------------------ \n");
        printf("\n Words read - %ld\n", f1->word_num);
        printf(" Words stored - %d\n", count_stored);
        printf("%s %d \n", "Number of different word", count_diff);
        cpu_time = elapsed_time();
        printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);

        if (fw == NULL)
        {
            printf("Erro a abrir o ficheiro results!\n");
            exit(1);
        }
        else
        {
            fprintf(fw, "HashTable LL Words Read \t %ld \n", f->word_num);
            fprintf(fw, "HashTable LL Words Stored \t %d \n", count_stored);
            fprintf(fw, "%s %d \n", "Number of different word", count_diff);
            fprintf(fw, "HashTable LL Time Travel Print \t %.6f \n", cpu_time);
        }

        fclose(fw);
        free(words1);
        free(f1);
    }
    else
    {
        usage(argv);
    }
}
```

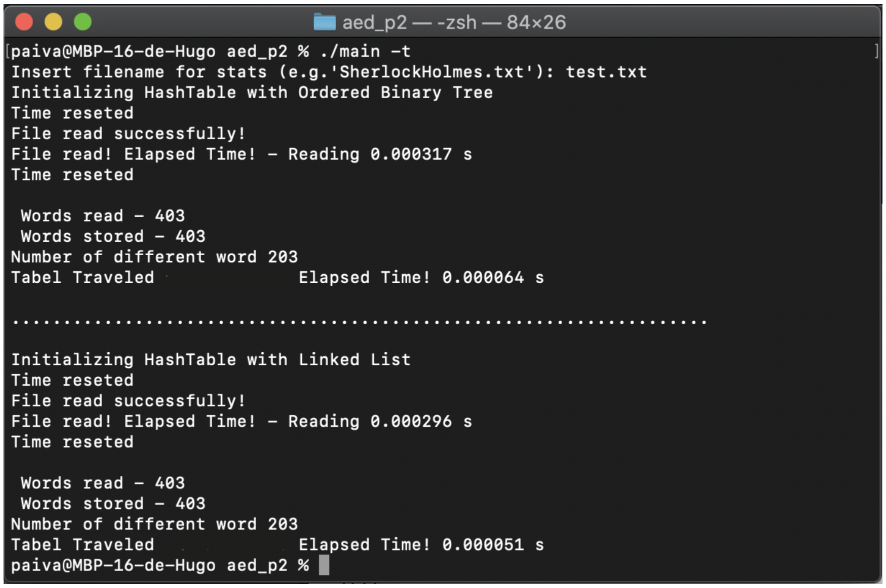
3 Results

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



```
[paiva@MBP-16-de-Hugo aed_p2 % ./main -t                                                    ]
Insert filename for stats (e.g.'SherlockHolmes.txt'): SherlockHolmes.txt
Initializing HashTable with Ordered Binary Tree
Time reseted
File read successfully!
File read! Elapsed Time! - Reading 0.212137 s
Time reseted

 Words read - 657438
 Words stored - 657438
Number of different word 45882
Tabel Traveled              Elapsed Time! 0.002390 s

....................................................................

Initializing HashTable with Linked List
Time reseted
File read successfully!
File read! Elapsed Time! - Reading 0.250115 s
Time reseted

 Words read - 657438
 Words stored - 657438
Number of different word 45882
Tabel Traveled              Elapsed Time! 0.003183 s
paiva@MBP-16-de-Hugo aed_p2 % 
```

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

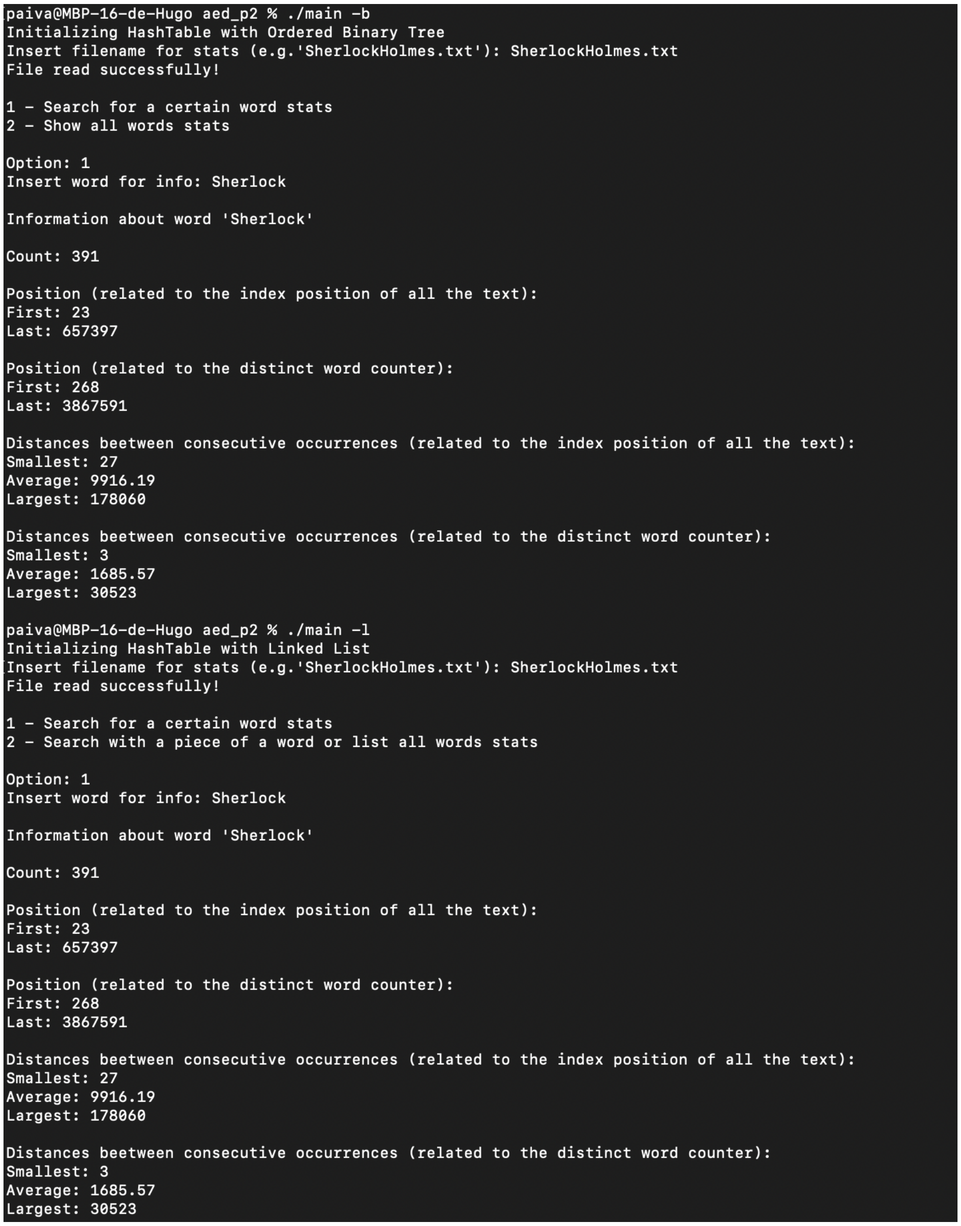


```
[paiva@MBP-16-de-Hugo aed_p2 % ./main -t                                                    ]
Insert filename for stats (e.g.'SherlockHolmes.txt'): test.txt
Initializing HashTable with Ordered Binary Tree
Time reseted
File read successfully!
File read! Elapsed Time! - Reading 0.000317 s
Time reseted

 Words read - 403
 Words stored - 403
Number of different word 203
Tabel Traveled              Elapsed Time! 0.000064 s

....................................................................

Initializing HashTable with Linked List
Time reseted
File read successfully!
File read! Elapsed Time! - Reading 0.000296 s
Time reseted

 Words read - 403
 Words stored - 403
Number of different word 203
Tabel Traveled              Elapsed Time! 0.000051 s
paiva@MBP-16-de-Hugo aed_p2 % 
```

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

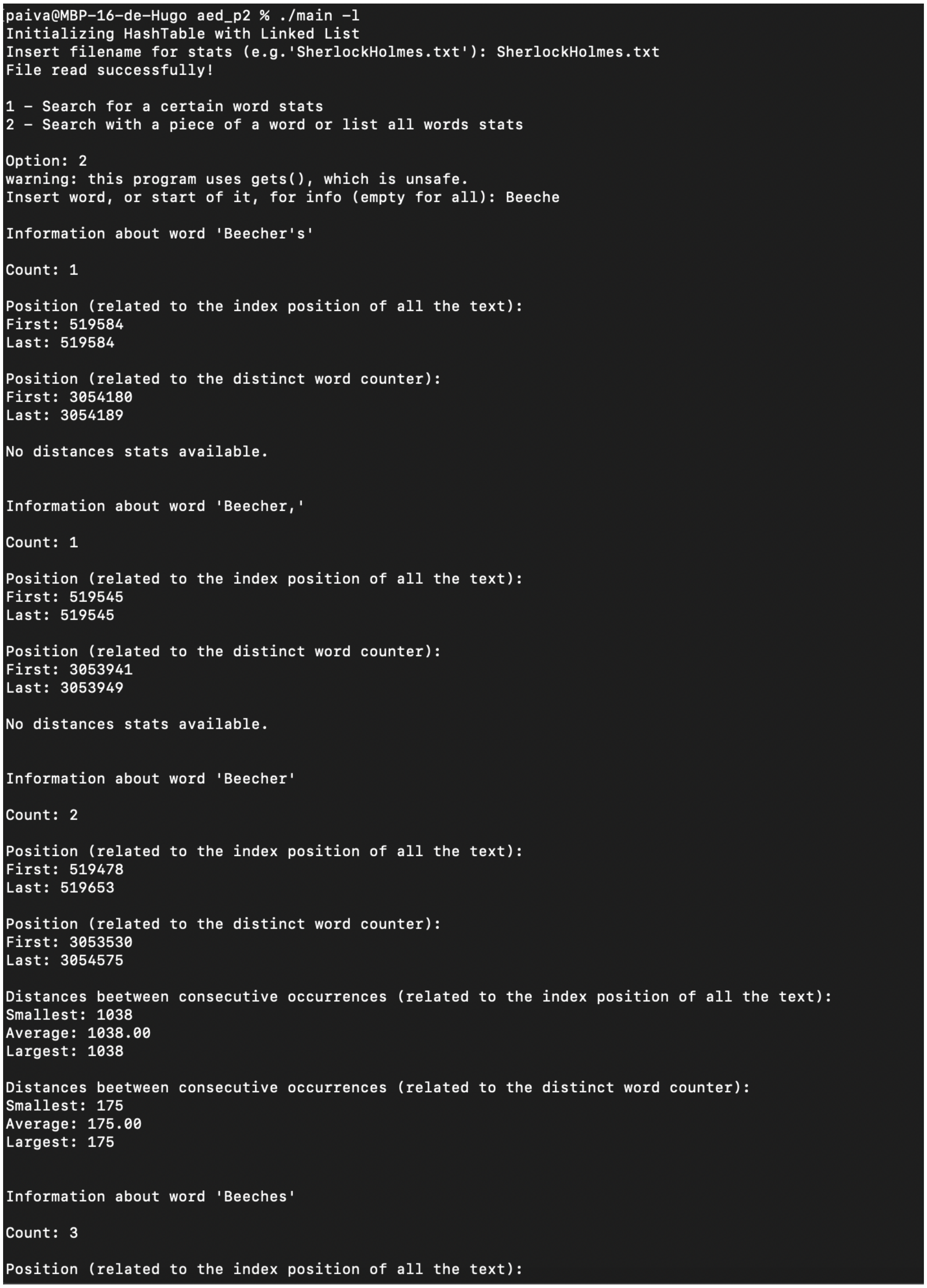
[2] The following results were obtained using the files *SherlockHolmes.txt* and *test.txt*

The other two options of the program were also tested:

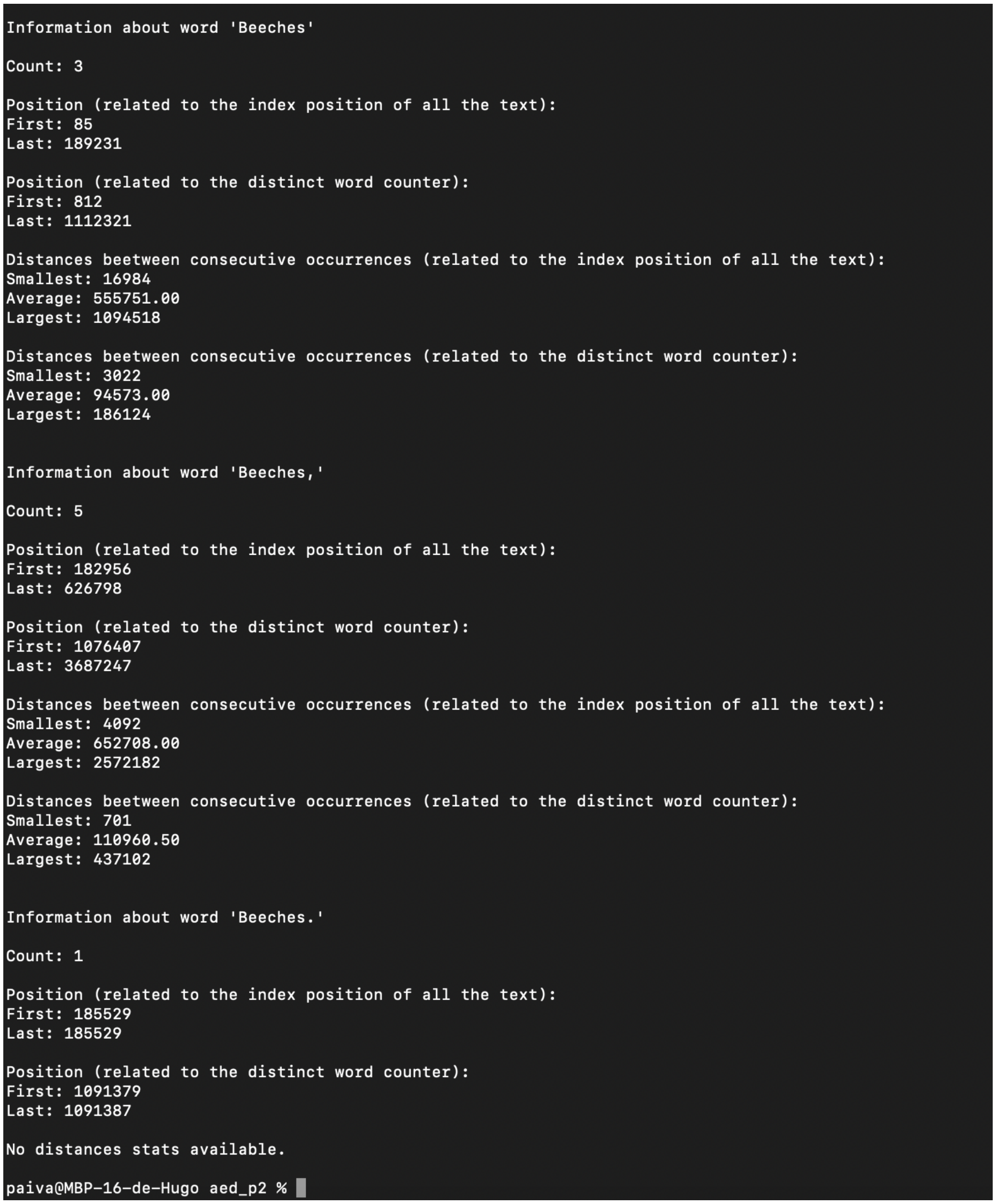


```
paiva@MBP-16-de-Hugo aed_p2 % ./main -b
Initializing HashTable with Ordered Binary Tree
Insert filename for stats (e.g.'SherlockHolmes.txt'): SherlockHolmes.txt
File read successfully!

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

Option: 1
Insert word for info: Sherlock

Information about word 'Sherlock'

Count: 391

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

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

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

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

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

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

Option: 1
Insert word for info: Sherlock

Information about word 'Sherlock'

Count: 391

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

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

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

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

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



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

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

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

Information about word 'Beecher's'

Count: 1

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

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

No distances stats available.


Information about word 'Beecher,'

Count: 1

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

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

No distances stats available.


Information about word 'Beecher'

Count: 2

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

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

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

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


Information about word 'Beeches'

Count: 3

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



```
Information about word 'Beeches'

Count: 3

Position (related to the index position of all the text):
First: 85
Last: 189231

Position (related to the distinct word counter):
First: 812
Last: 1112321

Distances beetween consecutive occurrences (related to the index position of all the text):
Smallest: 16984
Average: 555751.00
Largest: 1094518

Distances beetween consecutive occurrences (related to the distinct word counter):
Smallest: 3022
Average: 94573.00
Largest: 186124


Information about word 'Beeches,'

Count: 5

Position (related to the index position of all the text):
First: 182956
Last: 626798

Position (related to the distinct word counter):
First: 1076407
Last: 3687247

Distances beetween consecutive occurrences (related to the index position of all the text):
Smallest: 4092
Average: 652708.00
Largest: 2572182

Distances beetween consecutive occurrences (related to the distinct word counter):
Smallest: 701
Average: 110960.50
Largest: 437102


Information about word 'Beeches.'

Count: 1

Position (related to the index position of all the text):
First: 185529
Last: 185529

Position (related to the distinct word counter):
First: 1091379
Last: 1091387

No distances stats available.

paiva@MBP-16-de-Hugo aed_p2 %
```

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

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

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

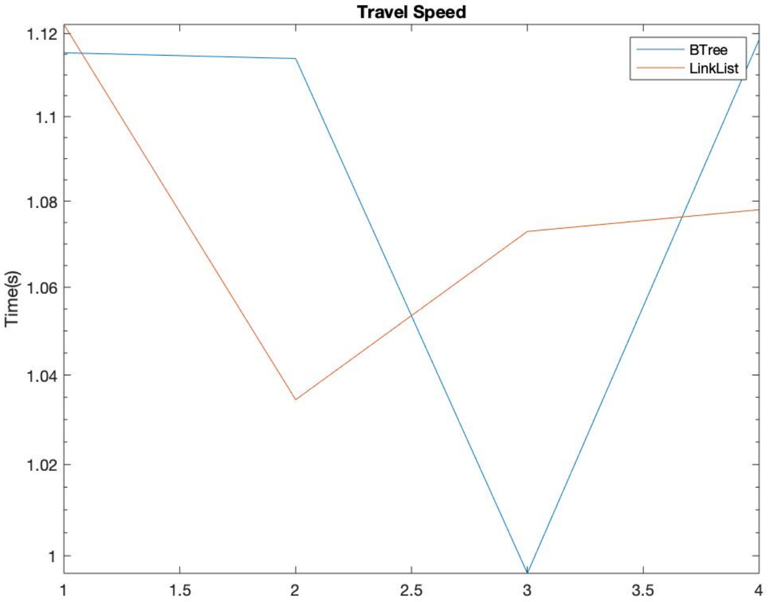


Figure 5: Time spent while travelling through all the words from the *SherlockHolmes.txt* file, according to the number of times the program has ran and for both, Linked List and Ordered Binary Tree structs

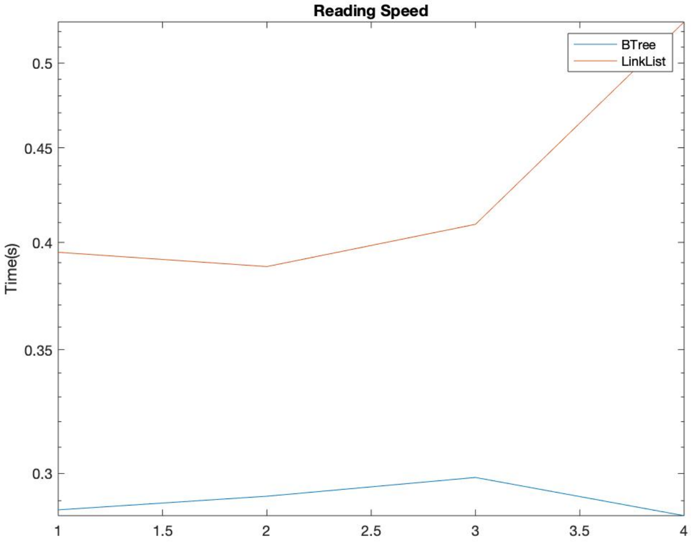


Figure 6: Time spent while reading all the words from the *SherlockHolmes.txt* file, according to the number of times the program has ran and for both, Linked List and Ordered Binary Tree structs

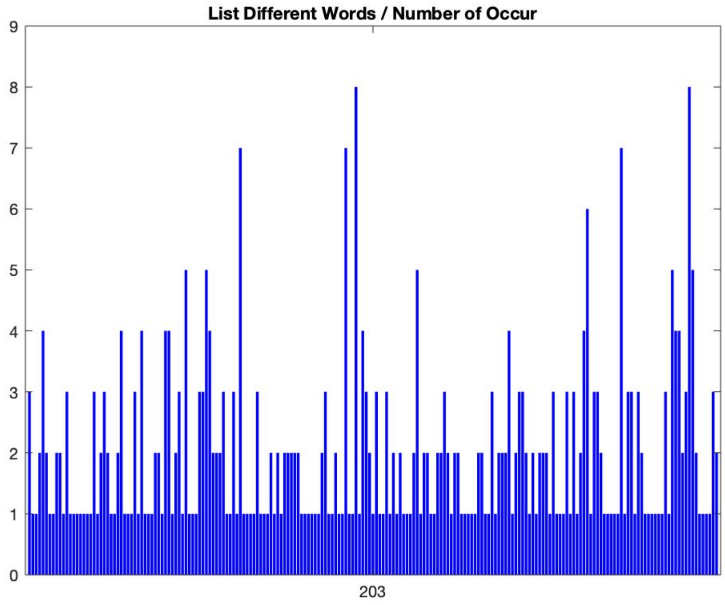


Figure 7: Number of occurrences of different words on the *test.txt* file, using Linked List struct

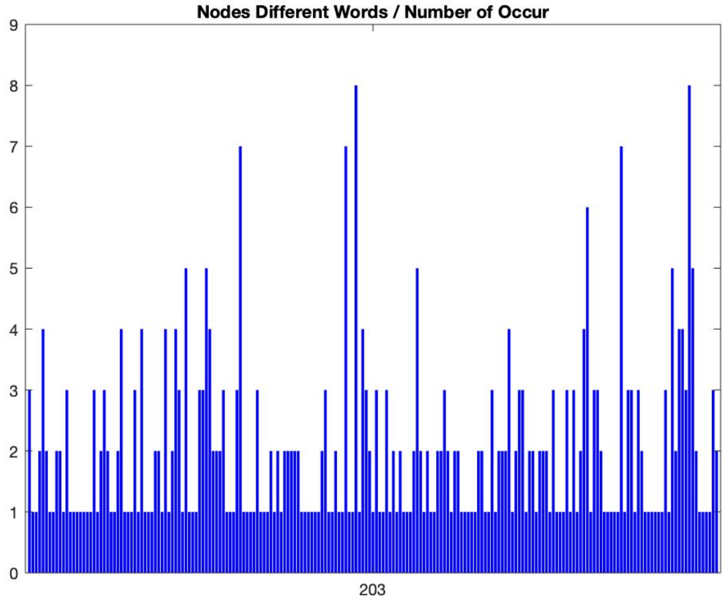


Figure 8: Number of occurrences of different words on the *test.txt* file, using Ordered Binary Tree struct

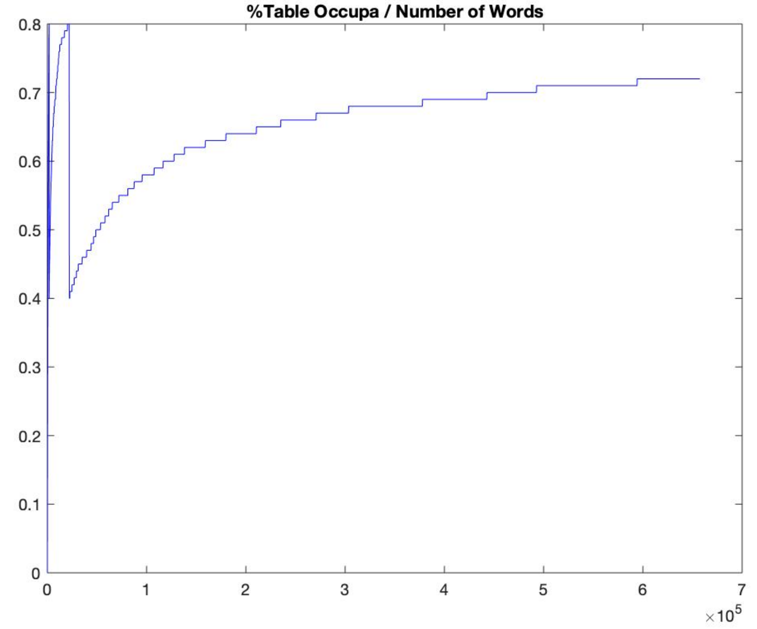


Figure 9: Table occupancy rate relative to the number of words read, on the *SherlockHolmes.txt* file, using Linked List struct

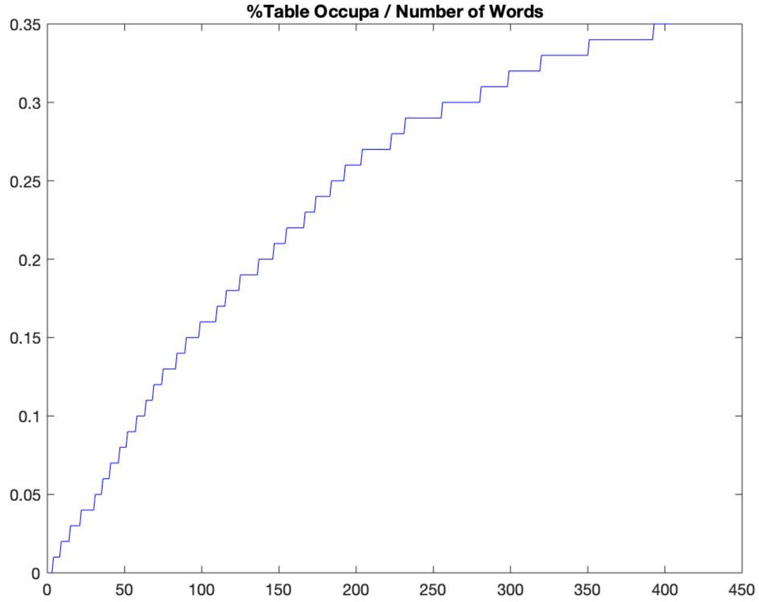


Figure 10: Table occupancy rate relative to the number of words read, on the *test.txt* file, using Linked List struct

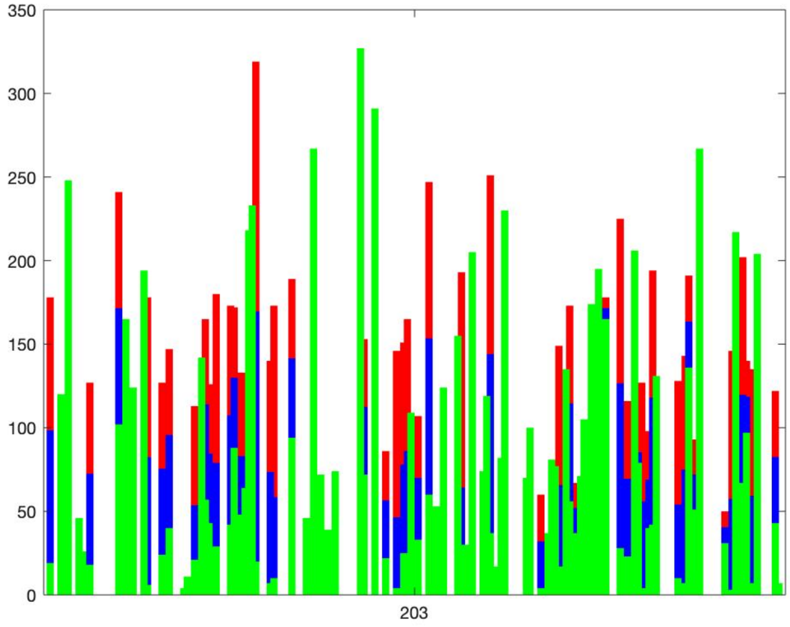


Figure 11: Maximum (red), average (blue) and minimum (green) distances between consecutive words, on the *test.txt* file

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

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

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

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

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

4 Conclusion

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

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

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

5 Bibliography

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

6 Appendix

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

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

int count_diff; // different word counter

static double cpu_time; // time counter

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

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

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

#include <time.h>

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

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

#endif

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

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

#include <windows.h>
```

```c

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
    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)
    long data;   // the data item (we use an int here, but it can be anything)
} tree_node;

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

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;
        new->left = NULL;
        new->right = 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;
    }
}

void add_ele(link_ele **words, file_data_t *f, int size)
{
```

```c
    int index = hash_function(f->word, size);
    link_ele *actual = words[index];
    if (actual != NULL) // if an element in the list already exists in that index
    {
        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;
        }
        else
        { // if not equal it is needed to run over all the elements
            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;
    }
}

void add_ele_resize(link_ele **words, link_ele *f, int size)
{
    int index = hash_function(f->word, size);
    link_ele *actual = words[index];
    if (actual != NULL)
    {
        while (actual->next != NULL)
        {
            actual = actual->next;
        }
        link_ele *temp = malloc(sizeof(link_ele));
        strcpy(temp->word, f->word);
        temp->next = NULL;
        temp->count = f->count;
        temp->dmin = f->dmin;
        temp->dmax = f->dmax;
        temp->dminp = f->dminp;
        temp->dmaxp = f->dmaxp;
        temp->first = f->first;
        temp->count = f->count;
        temp->last = f->last;
        temp->lastp = f->lastp;
        temp->firstp = f->firstp;
        actual->next = temp;

    }
    else
    {
        link_ele *new = malloc(sizeof(link_ele));
        strcpy(new->word, f->word);
        new->next = NULL;
        new->count = f->count;
        new->dmin = f->dmin;
        new->dmax = f->dmax;
        new->dminp = f->dminp;
        new->dmaxp = f->dmaxp;
        new->first = f->first;
        new->count = f->count;
        new->last = f->last;
        new->lastp = f->lastp;
        new->firstp = f->firstp;
        words[index] = new;
    }
}

link_ele **resize_link(link_ele **words, int *size)
{
    int newsize = 2 * (*size);
    link_ele **words_temp = (link_ele *)calloc(newsize, sizeof(link_ele *));
    for (int i = 0; i < (*size); i++)
    {
        if (words[i] != NULL)
        {
            link_ele *actual = words[i];
            add_ele_resize(words_temp, actual, newsize);
            while (actual->next != NULL)
            {
                actual = actual->next;
                add_ele_resize(words_temp, actual, newsize);

            }
        }
    }
    *size = 2 * (*size);
    return words_temp;
}

void get_info_link(link_ele **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);
    link_ele *actual = words[index];
    bool found = false;
    if (actual != NULL)
    {
        while (actual != NULL)
        {
            if (strcmp(actual->word, name) == 0)
            {
                printf("\nInformation about word '%s'\n", actual->word);
                printf("\nCount: %ld\n", actual->count);
                printf("\nPosition (related to the index position of all the text):\n");
                printf("First: %ld\n", actual->first);
                printf("Last: %ld\n", actual->last);
                printf("\nPosition (related to the distinct word counter):\n");
                printf("First: %ld\n", actual->firstp);
                printf("Last: %ld\n", actual->lastp);
                if (actual->count > 1)
                {
                    printf("\nDistances beetween consecutive occurrences (related to the index position of all the text):\n");
                    printf("Smallest: %ld\n", actual->dminp);
                    printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 number of dist and not words
                    printf("Largest: %ld\n", actual->dmaxp);
```

```c
                    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);
    }
}

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);
    }
}

int get_info_node_all(tree_node **words, int size)
{
    int c_stored = 0;
    count_diff = 0;
    FILE *f_node_all = fopen("results_fna.txt", "a+");
    if (f_node_all == NULL)
    {
        printf("Erro a abrir o ficheiro escrita!\n");
        exit(1);
    }
    else
    {
        printf("%s\n", "Aberto ficheiro results_fna.txt");
    }

    fprintf(f_node_all, "Word \t Count \t FPos \t LPos \t Dmin \t AvgD \t Dmax \n");
    bool found = false;
    for (int i = 0; i < size; i++)
    {
        tree_node *actual = words[i];
        tree_node *pre;
        if (actual != NULL)
        {
            while (actual != NULL)
            {
                if (actual->left == 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);
                    fprintf(f_node_all, "%s \t %ld \t %ld \t %ld \t ", actual->word, actual->count, actual->first, actual->last);
                    if (actual->count > 1)
                {
                    printf("\nDistances beetween consecutive occurrences (related to the index position of all the text):\n");
                    printf("Smallest: %ld\n", actual->dminp);
                    printf("Average: %.2f\n", (float)(actual->tdistp) / (actual->count - 1)); // -1 because number of distances and not words
                    printf("Largest: %ld\n", actual->dmaxp);
                    printf("\nDistances beetween consecutive occurrences (related to the distinct word counter):\n");
                    printf("Smallest: %ld\n", actual->dmin);
                    printf("Average: %.2f\n", (float)(actual->tdist) / (actual->count - 1));
                    printf("Largest: %ld\n\n", actual->dmax);
                    fprintf(f_node_all, "%ld \t %.2f \t %ld \n ", actual->dmin, (float)(actual->tdist) / (actual->count - 1), actual->dmax);
                }
                else
                {
                    printf("\n No distances stats available.\n\n");
                    fprintf(f_node_all, "%d \t %.2f \t %d \n ", 0, 0.00, 0);
                }
                found = true;
                actual = actual->right;
            }
            else
            {
                /* Find the inorder predecessor of current */
                pre = actual->left;
                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);
                    fprintf(f_node_all, "%s \t %ld \t %ld \t %ld \t ", actual->word, actual->count, actual->first, actual->last);

                    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);
```

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

                        }
                        else
                        {
                            printf("\n No distances stats available.\n\n");
                            fprintf(f_node_all, "%d \t %.2f \t %d \n ", 0, 0.00, 0);
                        }
                        found = true;
                        actual = actual->right;
                    }
                }
            }
        }
    }
    if (!found)
    {
        printf("No words found!\n");
        exit(0);
    }
    return c_stored;
}

int get_info_link_all(link_ele **words, int size, bool all)
{
    bool found = false;
    int c_st = 0;
    count_diff = 0;
    FILE *f_link_all = fopen("results_fla.txt", "a+");
    if (all == true)
        goto all;
    char name[64];
    printf("Insert word, or start of it, for info (empty for all): ");
    if (gets(name) != NULL)
    {
        int s_name = strlen(name);
        for (int i = 0; i < size; i++)
        {
            link_ele *actual = words[i];
            while (actual != NULL)
            {
                if (strncmp(name, actual->word, s_name) == 0)
                {
                    found = true;
                    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;
            }
        }
    }
    else
    {
    all:

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

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

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

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

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

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

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

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

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

```c
    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;
}

void usage(char *argv[])
{
    printf("Unknown option\n");
    printf("\nUsage: %s -l -b -t\n\n", argv[0]);
    printf("-l Initialize program using HashTable with Linked List\n");
    printf("-b Initialize program using HashTable with Ordered Binary Tree\n");
    printf("-t Initialize program for Tests\n");

    exit(0);
}

int main(int argc, char *argv[])
{
    if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'l')
    {
        count_array = 0;
        printf("Initializing HashTable with Linked List\n");
        int s_hash = 500;
        link_ele **words = (link_ele *)calloc(s_hash, sizeof(link_ele *)); // creates and announce them as zero (null)
        file_data_t *f = malloc(sizeof(file_data_t));
        char file[64];
        printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
        scanf("%[^\n]", file);
        fflush(stdin);
        if (!open_text_file(file, f))
        {
            while (!read_word(f))
            {
                if ((double)count_array / s_hash >= 0.8)
                {
                    words = resize_link(words, &s_hash);
                }
                add_ele(words, f, s_hash);
            }
            printf("File read successfully!\n");
            close_text_file(f);
        }
        else
        {
            printf("-------------------\n");
            printf("Error opening file!\n");
            printf("-------------------\n");
            exit(0);
        }
        printf("\n1 - Search for a certain word stats\n2 - Search with a piece of a word or list all words stats\n");
        char option[5];
        printf("\nOption: ");
        scanf("%[^\n]", option);
        fflush(stdin);
        if (strcmp(option, "1") == 0)
            get_info_link(words, s_hash);
        else if (strcmp(option, "2") == 0)
            get_info_link_all(words, s_hash, false);
        else
        {
            printf("Invalid option");
            exit(0);
        }
    }
    else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'b')
    {
        printf("Initializing HashTable with Ordered Binary Tree\n");
        int s_hash = 500;
        tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and announce them as zero (null)
        file_data_t *f = malloc(sizeof(file_data_t));
        char file[64];
        printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
        scanf("%[^\n]", file);
        fflush(stdin);
        if (!open_text_file(file, f))
        {
            while (!read_word(f))
            {
                add_node(words, f, s_hash);
            }
            printf("File read successfully!\n");
            close_text_file(f);
        }
        else
        {
            printf("--------------------\n");
            printf("Error opening file!\n");
            printf("--------------------\n");
            exit(0);
        }
        printf("\n1 - Search for a certain word stats\n2 - Show all words stats\n");
        char option[5];
        printf("\nOption: ");
        scanf("%[^\n]", option);
        fflush(stdin);
        if (strcmp(option, "1") == 0)
            get_info_node(words, s_hash);
        else if (strcmp(option, "2") == 0)
            get_info_node_all(words, s_hash);
        else
        {
            printf("Invalid option");
            exit(0);
        }
    }
    else if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 't')
    {
        char file[64];
        printf("Insert filename for stats (e.g.'SherlockHolmes.txt'): ");
        scanf("%[^\n]", file);
        fflush(stdin);

        printf("Initializing HashTable with Ordered Binary Tree\n");
        reset_time();
        int s_hash = 500;
        int count_stored = 0;
        (void)elapsed_time();
        tree_node **words = (tree_node *)calloc(s_hash, sizeof(tree_node *)); // creates and anounce them as zero (null)
        file_data_t *f = malloc(sizeof(file_data_t));

        if (!open_text_file(file, f))
        {
            while (!read_word(f))
            {
                add_node(words, f, s_hash);
            }
            printf("File read successfully!\n");
            close_text_file(f);
        }
        else
        {
            printf("--------------------\n");
            printf("Error opening file!\n");
            printf("--------------------\n");
            exit(0);
        }

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

        FILE *fw = fopen("results.txt", "a+");

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

        reset_time();

        printf("\nPrinting all words stored...\n");
        (void)elapsed_time();
        usleep(5000000);
        count_stored = get_info_node_all(words, s_hash);
        printf("\n ---------------------------------------------------------------------- \n");
        printf("\n Words read - %ld\n", f->word_num);
        printf(" Words stored - %d\n", count_stored);
        printf("%s %d \n", "Number of different word", count_diff);
        cpu_time = elapsed_time();
        printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);

        if (fw == NULL)
        {
            printf("Erro a abrir o ficheiro results!\n");
```

```c
        exit(1);
    }
    else
    {
        fprintf(fw, "HashTable OBT Words Read \t %ld \n", f->word_num);
        fprintf(fw, "HashTable OBT Words Stored \t %d \n", count_stored);
        fprintf(fw, "%s %d \n", "Number of different word", count_diff);
        fprintf(fw, "HashTable OBT Time Travel Print \t %.6f \n", cpu_time);
    }

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

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

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

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

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

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

    reset_time();
```

```
        printf("\nPrinting all words stored...\n");
        (void)elapsed_time();
        usleep(5000000);
        count_stored = get_info_link_all(words1, s_hash, true);
        printf("\n --------------------------------------------------------------------------- \n");
        printf("\n Words read - %ld\n", f1->word_num);
        printf(" Words stored - %d\n", count_stored);
        printf("%s %d \n", "Number of different word", count_diff);
        cpu_time = elapsed_time();
        printf("%s %.6f s \n", "Tabel Traveled and Printed! Elapsed Time!", cpu_time);

        if (fw == NULL)
        {
            printf("Erro a abrir o ficheiro results!\n");
            exit(1);
        }
        else
        {
            fprintf(fw, "HashTable LL Words Read \t %ld \n", f->word_num);
            fprintf(fw, "HashTable LL Words Stored \t %d \n", count_stored);
            fprintf(fw, "%s %d \n", "Number of different word", count_diff);
            fprintf(fw, "HashTable LL Time Travel Print \t %.6f \n", cpu_time);
        }

        fclose(fw);
        free(words1);
        free(f1);
    }
    else
    {
        usage(argv);
    }
}
```