Algorithms and Data Structures

Occurrences Counter

Students:

Isadora Loredo – 91322-50%
Margarida Martins – 93169-50%

31th of December 2019

universidade de aveiro deti departamento de eletrónica, telecomunicações e informática

Index

The Assignment

Count the number of occurrences of each word of a text file (book Sherlock Holmes – A Study In Scarlet), records the location of the first and last occurrences of each distinct word and records the smallest, largest, and average distances between consecutive occurrences of the same distinct word.

The program must use a **hash table** (separate chaining), the hash table size should grow dynamically, and each hash table entry should point to either a **linked list** or an ordered **binary tree**.

Implementation

manipulate_file.h

The header file 'manipulate_file.h' contains functions declaration and data structures that are used by both 'c' programs, with **linked list** and with **binary tree**, through the directive **#include**.

hash_size

Global variable used to define the size of the **hash table**.

```
unsigned int hash_size= 2000u;
```

file_data_t

The new data type struct 'file_data_t' defines the stats of the text file read and stores the current word.

- long word_pos: position of the current word being read in the file;
- long word_num: number of words read in the file;
- char word[]: array of characters meaning the current word being read in the file;
- FILE *fp: file pointer pointing to the file being accessed;
- long current_pos: current position in the file for the read word;

```
// FILE STRUCT
typedef struct file_data_t {
    // 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;
```

word_stats

The new data type struct 'word_stats' defines the stats of each word read by the text file.

- int number_occurrences: number of occurrences of the word in the file;
- int first_appearance: first appearance of the word in the file;
- int last_appearance: last appearance of the word in the file;
- int s_distance: smallest distance between consecutive occurrences of the same distinct word;
- int l_distance: largest distance between consecutive occurrences of the same distinct word;
- int m_distance: medium distance between consecutive occurrences of the same distinct word, so the average distance can be calculated;

- int t_distance: total distance between consecutive occurrences of the same distinct word;

```
// STRUCT THAT DEFINES THE STATS FOR THE WORDS IN THE HASH-TABLE
typedef struct word_stats {
    int number_occurrences;
    int first_appearence;
    int last_appearence;
    int s_distance; // smallest distance
    int l_distance; // largest distance
    int m_distance; //. medium distance
    int t_distance; // total distance
} word_stats;
```

open_text_file()

The function 'open_text_file' is used to open the file to be read and initialize the variables of the data type 'file_data_t'.

Arguments:

- char *file_name: file pointer pointing to the text file to be accessed;
- file_data_t *fd: file stats pointer pointing to the struct to be initialized;

Return:

- int *value*: returns a integer value, '0' if success or '-1' if file stats pointer is null;

```
// OPEN FILE
int open_text_file(char *file_name,file_data_t *fd) {
    fd->fp = fopen(file_name,"r");
    if(fd->fp == NULL)
        return -1;
    fd->word_pos = -1;
    fd->word_num = -1;
    fd->word[0] = '\0';
    fd->current_pos = -1;
    return 0;
}
```

close_text_file()

The function 'close_text_file' is used to close the file read.

Arguments:

- file_data_t *fd: file stats pointer pointing to the struct initialized;

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

read_word()

The function 'read_word' is called to read the next word of the accessed file and save update its stats. It reads each character of the file using white spaces, special characters and punctuations to identify the beginning and end of a word. The characters of the read words are passed to lower case so same words are not detected as different.

Arguments:

- file_data_t *fd: file stats pointer pointing to the struct of the file being accessed;

Return:

- int *value*: returns a integer value, '-1' if end of file reached or 0' if end of file still not reached;

```
// READ WORD
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 >= 48 && c<58) || (c>=65 && c<=90) || (c>=97 && c<=122) || (c>=192)));
    // record word
    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 >= 48 && c<58) || (c>=65 && c<=90) || (c>=97 && c<=122) || (c>=192)))
            break; // terminate word
        fd->word[i] = (char)c;
    }
    fd->word[i] = '\0';
    for(int j = 0; fd->word[j]; j++){
        fd->word[j] = tolower(fd->word[j]);
    }
    return 0;
}
```

hash_function()

The function 'hash_function' maps each possible key (word of the file) to an integer. The integer will then be used to access the **hash table** array, as the index to the array.

Arguments:

- const char *str: the key (word) to me mapped;
- unsigned int s: size of the hash table;

Return:

- unsigned int *value*: calculated value as the index to the hash table;

```c
// HASH-FUNCTION
unsigned int hash_function(const char *str, unsigned int s) {
    unsigned int h;
    for(h = 0u;*str != '\0';str++)
        h = 157u * h + (0xFFu & (unsigned int)*str); // arithmetic overflow may occur here (just ignore it!)
    return h % s; // due to the unsigned int data type, it is guaranteed that 0 <= h % s < s
}
```

hash_data

The new data type struct ‘hash_data’ defines the nodes of data to be inserted in the **hash table** to the **linked list** implementation.

- struct hash_data *next: pointer to the next node in the linked list;
- char key[]: word read in the file;
- struct word_stats word: stats of the word (key);

```c
// HASH-TABLE NODE LINKED LIST
typedef struct hash_data {
    struct hash_data *next;
    char key[64];
    struct word_stats word;
} hash_data;
```

new_hash_data()

The function ‘new_hash_data’ initializes a new hash_data type struct, new node, allocating the necessary memory size.

Return:

- hash_data *hd: pointer to the start location to the initialized node;

```c
// ALLOCATE NEW HASH-DATA
hash_data *new_hash_data(void) {
    hash_data *hd = (hash_data *)malloc(sizeof(hash_data));
    if(hd == NULL) {
        fprintf(stderr,"Out of memory\n");
        exit(1);
    }
    return hd;
}
```

hash_resize()

The function ‘hash_resize’ is called when the **hash table** reaches a certain number of nodes to be inserted and needs to grow in order to avoid an unwanted too large size of the **linked lists**. It allocates the necessary memory to the hash table with its new size then initializes its null nodes. Finally goes through the old hash table and its linked lists and recalculates the index of each node’s key to relocate them in the new resized hash table.

Arguments:

- hash_data **hash_table: pointer pointing to the address that points to the beginning of the **hash table;**
- unsigned int inc: value of the increment to be added to the hash size;

Return:

- hash_data **hash_data: pointer pointing to the address that points to the beginning of the **hash table** with the increased size;

```
//RESIZE LINKED LIST
 struct hash_data ** hash_resize(struct hash_data **hash_table, unsigned int inc){
    //printf("Resizing..........................................................................................
    hash_data *next;
    hash_size+=inc;
    struct hash_data **hash_table_new=  malloc((hash_size)*sizeof(struct hash_data));
    int new_idx;
    for(int m = 0;m < hash_size;m++) hash_table_new[m] = NULL;
    for(int l=0;l<hash_size-inc;l++){
        while(hash_table[l]!=NULL){
            new_idx= hash_function(hash_table[l]->key, hash_size);
            next=hash_table[l]->next;
            hash_table[l]->next=hash_table_new[new_idx];
            hash_table_new[new_idx]=hash_table[l];
            hash_table[l]=next;
        }

    }
    free(hash_table);
    return hash_table_new;
}
```

hash_data_bt

The new data type struct 'hash_data_bt' defines the nodes of data to be inserted in the **hash table** to the **binary tree** implementation.

- struct hash_data *left: pointer to the node on the left side (the left branch);
- struct hash_data *right: poin pointer to the node on the right side (the right branch);
- char key[]: word read in the file;
- struct word_stats word: stats of the word (key);

```
// HASH-TABLE NODE BINARY TREE
typedef struct hash_data_bt {
    struct hash_data_bt *left;
    struct hash_data_bt *right;
    char key[64];
    struct word_stats word;
} hash_data_bt;
```

new_hash_data_bt()

The function 'new_hash_data_bt' initializes a new hash_data_bt type struct, new node, allocating the necessary memory size.

Return:

- hash_data *hd: pointer to the start location to the initialized node;

```c
// ALLOCATE NEW HASH-DATA
hash_data_bt *new_hash_data_bt(void) {
    hash_data_bt *hd = (hash_data_bt *)malloc(sizeof(hash_data_bt));
    if(hd == NULL) {
        fprintf(stderr,"Out of memory\n");
        exit(1);
    }
    return hd;
}
```

hash_resize_bt()

The function 'hash_resize_bt' is called when the **hash table** reaches a certain number of nodes to be inserted and needs to grow in order to avoid an unwanted too large size of the **binary trees**. It allocates the necessary memory to the **hash table** with its new size then initializes its null nodes. Finally goes through the old **hash table** and its **binary trees** and recalculates the index of each node's key to relocate them in the new resized hash table.

Arguments:

- hash_data_bt **hash_table: pointer pointing to the address that points to the beginning of the **hash table;**
- unsigned int inc: value of the increment to be added to the hash size;

Return:

- hash_data_bt **hash_data: pointer pointing to the address that points to the beginning of the **hash table** with the increased size;

```c
//RESIZE BINARY TREE
 struct hash_data_bt ** hash_resize_bt(struct hash_data_bt **hash_table, unsigned int inc){
    //printf("Resizing.....................................................................................
    hash_data_bt *next;
    struct hash_data_bt **hash_table_new=  malloc((hash_size+inc)*sizeof(struct hash_data_bt));
    int new_idx;
    hash_size+=inc;
    for(int m = 0;m < hash_size;m++) hash_table_new[m] = NULL;
    for(int l=0;l<hash_size-inc;l++){
        traverse_tree(hash_table[l],hash_table_new);
    }
    free(hash_table);
    return hash_table_new;
}
// CLOSE FILE
void close_text_file(file_data_t *fd) {
    fclose(fd->fp);
    fd->fp = NULL;
}
```

traverse_tree()

The function 'traverse_tree' is called when is necessary to perform visits in the **binary trees**. The visits are made using depth-first search recursively. It performs the traverse search, then it calculates the new index of the given node in the resized hash table and if the calculated index has still no allocated node it allocates it as the root of a new tree, otherwise it calls the function to insert the node in the tree.

Arguments:

- hash_data_bt *hb: pointer pointing to the node to be inserted in the resized **hash table;**
- hash_data_bt **hash_table: pointer pointing to the address that points to the beginning of the resized **hash table;**

```
//TRAVERSE_TREE
void traverse_tree(struct hash_data_bt *hb, struct hash_data_bt **hash_table){
    if(hb!=NULL){
        int new_idx= hash_function(hb->key, hash_size);
        if(hash_table[new_idx]==NULL){
            hash_table[new_idx]=new_hash_data_bt();
            strcpy(hash_table[new_idx]->key,hb->key);
            hash_table[new_idx]->left=NULL;
            hash_table[new_idx]->right=NULL;
            hash_table[new_idx]->word=hb->word;
        }
        else{
            hash_data_bt *temp =new_hash_data_bt();
            temp->left=NULL;
            temp->right=NULL;
            temp->word=hb->word;
            strcpy(temp->key,hb->key);
            insert_bt(hash_table[new_idx],temp);
        }
        traverse_tree(hb->left, hash_table);
        traverse_tree(hb->right, hash_table);
    }
}
```

insert_bt()

The function ‘insert_bt’ is called when is necessary to insert a new node in the **binary tree** at the appropriate location, and it is done recursively. It receives the root node and the node to be inserted as arguments, it compares the keys (word) of the arguments so it can insert the node or keep going forward until it finds the proper leaf to insert it as it’s child.

Arguments:

- hash_data_bt *root: pointer pointing to the node to be inserted in the resized **hash table;**
- hash_data_bt *hd: pointer pointing to the address that points to the beginning of the resized **hash table;**

```
// INSERT
void insert_bt(hash_data_bt *root, hash_data_bt *hd ){
    if(strcmp(hd->key,root->key)<0){
        if(root->left!=NULL)
            insert_bt(root->left,hd);
        else
            root->left=hd;
    }
    if(strcmp(hd->key,root->key)>0){
        if(root->right!=NULL)
            insert_bt(root->right,hd);
        else
            root->right=hd;
    }
}
```

Linked List

The first solution implemented uses a hash table that grows dynamically and each hash table entry points to a **linked list** when a collision occurs.

The structure *[*hd](#)* is the node to be stored to the hash table, before it does it's index is calculated using a [hash function](#). The word's stats are stored, location of the first and last occurrences and the smallest, largest and average distances between consecutive occurrences. It also counts the number of occurrences of each distinct word in the text file and the number of times the hash table is resized.

```
int main(int argc, char **argv){
    file_data_t fd;
    int idx;
    int j = 0;
    int count = 0;
    int i = 0;
    int r_count = 0;
    struct hash_data **hash_table = malloc(hash_size * sizeof(struct hash_data));
    for (int k = 0; k < hash_size; k++)
        hash_table[k] = NULL;
    open_text_file("SherlockHolmes.txt", &fd);
    while (read_word(&fd) != -1){
        idx = hash_function(fd.word, hash_size);
        hash_data *hd;
        for (hd = hash_table[idx]; hd != NULL && strcmp(fd.word, hd->key) != 0; hd = hd->next)
            i++;
        if (hd == NULL){
            if (i == 0){
                hd = new_hash_data();
                strcpy(hd->key, fd.word);
                hd->word.number_occurrences = 1;
                hd->word.first_appearence = count;
                hd->word.last_appearence = count;
                hd->next = NULL;
                hash_table[idx] = hd;
            } else{
                hd = new_hash_data();
                hd->next = hash_table[idx];
                strcpy(hd->key, fd.word);
                hd->word.number_occurrences = 1;
                hd->word.first_appearence = count;
                hd->word.last_appearence = count;
                hash_table[idx] = hd;
            }
            if (j > hash_size * 5){
                hash_table = hash_resize(hash_table, 1000u);
                r_count++;
            }
            j++;
        } else{
            hd->word.number_occurrences += 1;
            if (hd->word.number_occurrences == 2){
                hd->word.s_distance = count - hd->word.first_appearence;
                hd->word.l_distance = count - hd->word.first_appearence;
                hd->word.t_distance += hd->word.s_distance;
            } else{
                if (hd->word.s_distance > count - hd->word.last_appearence)
                    hd->word.s_distance = count - hd->word.last_appearence;
                if (hd->word.l_distance < count - hd->word.last_appearence)
                    hd->word.l_distance = count - hd->word.last_appearence;
                hd->word.t_distance += count - hd->word.last_appearence;
            }
            hd->word.m_distance = hd->word.t_distance / hd->word.number_occurrences;
            hd->word.last_appearence = count;
        }
        i = 0;
        count++;
    }
    close_text_file(&fd);
    free(hash_table);
    return 1;
}
```

Binary Tree

The second solution implemented uses a **hash table** that grows dynamically and each **hash table** entry points to an ordered **binary tree** when a collision occurs.

The structure **hd* is the node to be stored to the hash table, before it does it's index is calculated using a hash function. The word's stats are stored, location of the first and last occurrences and the smallest, largest and average distances between consecutive occurrences. It also counts the number of occurrences of each distinct word in the text file and the number of times the hash table is resized.

```c
int main(int argc, char **argv){
    file_data_t fd;
    int idx, j = 0, i = 0, count = 0, r_count = 0;
    struct hash_data_bt **hash_table= malloc(hash_size * sizeof(struct hash_data));
    for (int k = 0; k < hash_size; k++) hash_table[k] = NULL;
    open_text_file("SherlockHolmes.txt", &fd);
    while (read_word(&fd) != -1){
        idx = hash_function(fd.word, hash_size);
        hash_data_bt *hd = hash_table[idx];
        while (hd != NULL && strcmp(fd.word, hd->key) != 0){
            if (strcmp(fd.word, hd->key) < 0){
                hd = hd->left;
            } else{
                hd = hd->right;
            }
            i++;
        }
        if (hd == NULL){
            if (i == 0){
                hd = new_hash_data_bt();
                strcpy(hd->key, fd.word);
                hd->word.number_occurrences = 1;
                hd->word.first_appearence = count;
                hd->word.last_appearence = count;
                hd->right = NULL;
                hd->left = NULL;
                hash_table[idx] = hd;
            } else{
                hd = new_hash_data_bt();
                hd->word.number_occurrences = 1;
                hd->word.first_appearence = count;
                hd->word.last_appearence = count;
                hd->right = NULL;
                hd->left = NULL;
                strcpy(hd->key, fd.word);
                insert_bt(hash_table[idx], hd);
            }
            if (j > hash_size * 5){
                hash_table = hash_resize_bt(hash_table, 1000U);
                r_count++;
            }
            j++;
        } else{
            hd->word.number_occurrences += 1;
            if (hd->word.number_occurrences == 2){
                hd->word.s_distance = count - hd->word.first_appearence;
                hd->word.l_distance = count - hd->word.first_appearence;
                hd->word.t_distance += hd->word.s_distance;
            } else{
                if (hd->word.s_distance > count - hd->word.last_appearence)
                    hd->word.s_distance = count - hd->word.last_appearence;
                if (hd->word.l_distance < count - hd->word.last_appearence)
                    hd->word.l_distance = count - hd->word.last_appearence;
                hd->word.t_distance += count - hd->word.last_appearence;
            }
            hd->word.m_distance = hd->word.t_distance / hd->word.number_occurrences;
            hd->word.last_appearence = count;
        }
        i = 0;
        count++;
    }
    close_text_file(&fd);
    free(hash_table);
    return 1;
}
```

Word statistics of text file

In order analyze the statistics of the words present in the book “Sherlock Holmes – A Study In Scarlet” the words stats present in the **hash table** (outcome of running the “.c” files) were stored in multiples “.txt” files.

Number of occurrences

According to the results obtained the book has 19192 different words in a total of 668874.

The word with the biggest number of appearances is the word “the” showing up 36233 times throughout the text (5,4% of all word occurrences).

On the graph below (figure 1) we can observe that the number of words decreases very fast with the number of occurrences. There are 6729 words who only appear once 2799 who appear twice and 1655 thrice, together they make up more than half of the words present in the text (58,3%).

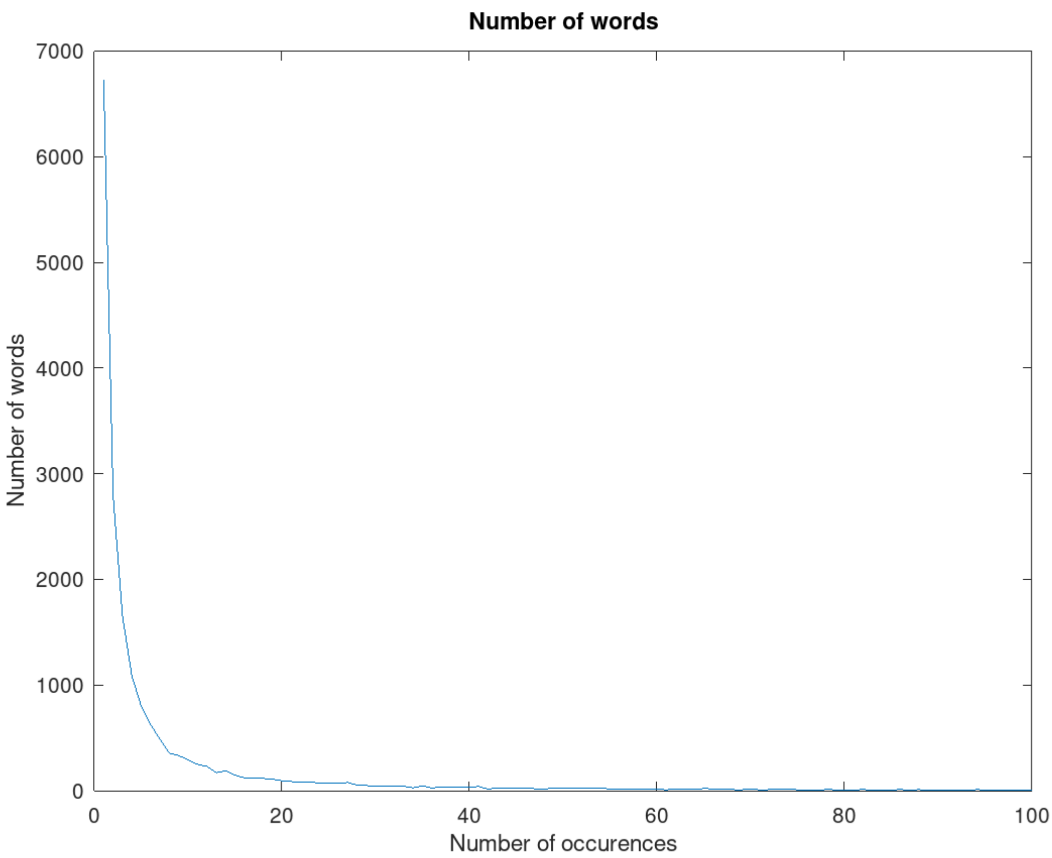

Figure 1- Number of words who occur between 1 and 100 times throughout the text.

First and last appearances

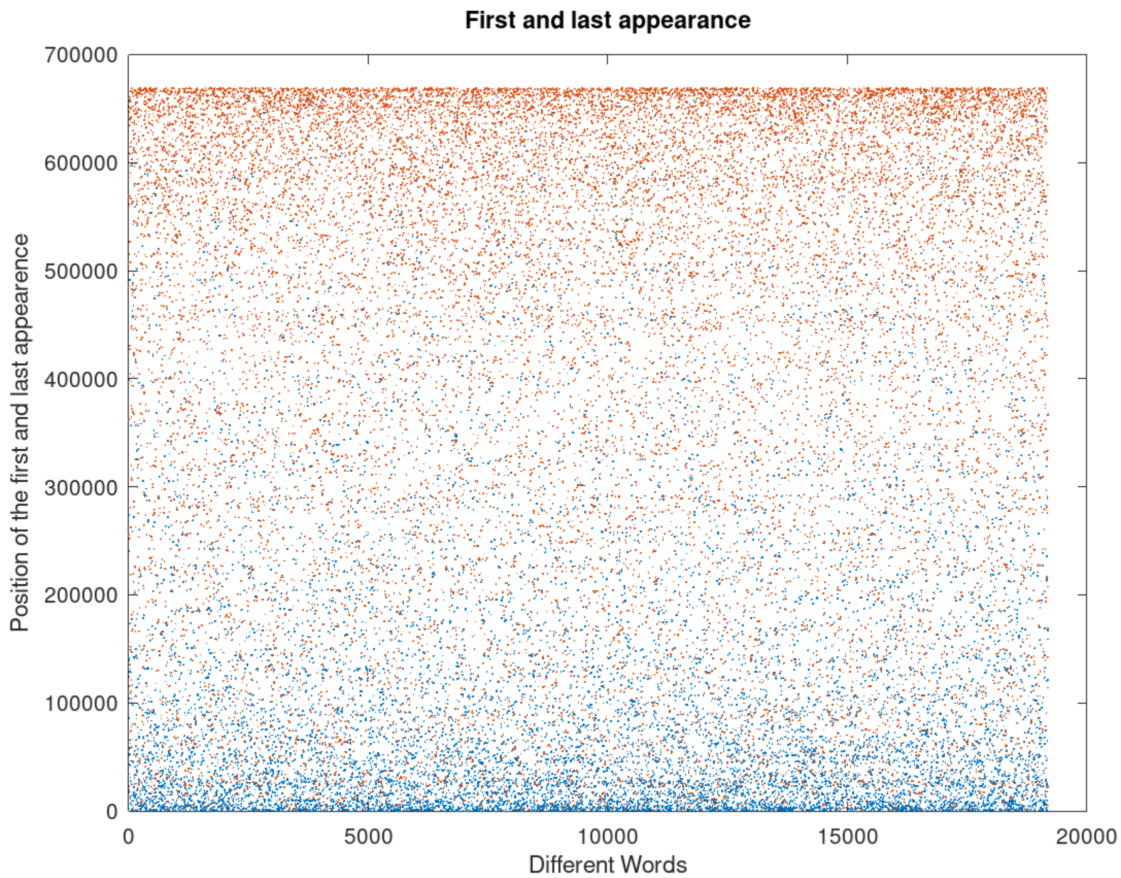

Figure 2- First (blue) and last (orange) appearances of different words in the text.

As expected, the position of the first appearance of a word is concentrated on the first 50000 words while the last appearance is concentrated on the last 50000 too. Still there are a considerable big number of words which appear, for the first or the last time, in the middle of the text, this is justified given the high amount of words who occur only once or twice int the book.

Smallest, medium and largest distance

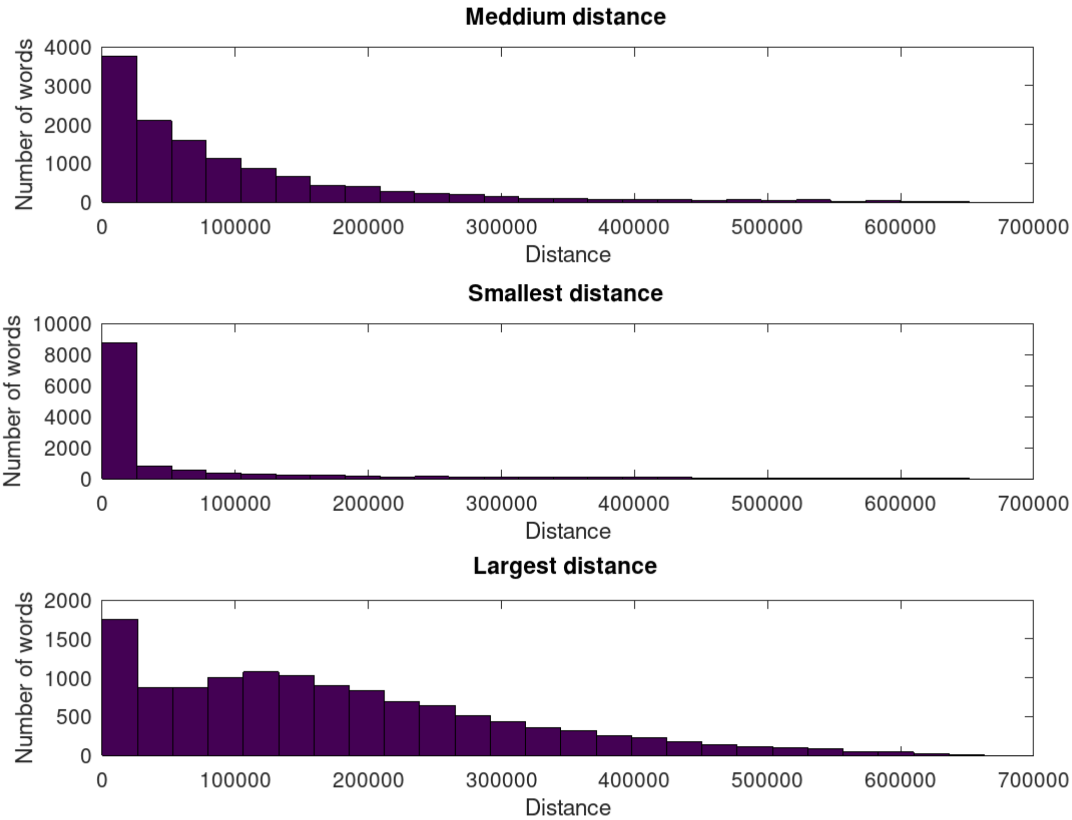

Figure 3- Medium, smallest and largest distance between equal words.

Almost every word has less than 100000 words as the smallest distance, however the values for the largest distance are more scattered and interestingly a considerably big number of words (~1750) has the largest distance below 25000 words. Almost 4000 words have a medium distance of less than 25000.

Time of execution and resize

For both the **linked list** and the **binary tree** approach it was established that the **hash tables** would resize when the number of different words was bigger than 5 times the size of the **hash table** (this value could be bigger for the **binary tree** approach given the properties of the structure).

Setting the increment of the **hash table** size after each resize at 1000, the time of execution of the programs was measured changing the initial size of the **hash table**. The results are shown below (figure 4).

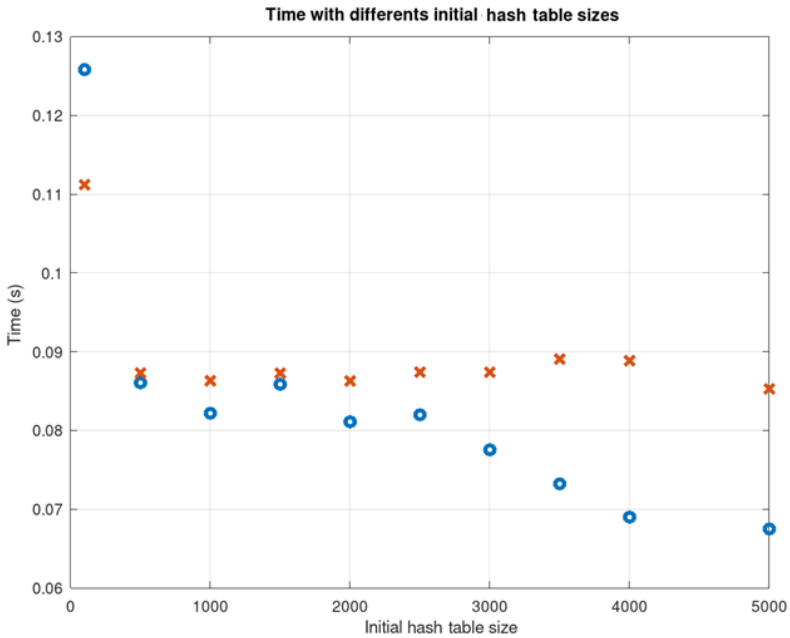

Figure 4 – Time of execution with different hash table sizes, for the binary tree approach (blue) and for the linked list approach (orange).

Hash table size	Resizes
100	4
500	4
1000	3
1500	3
2000	2
2500	2
3000	1
3500	1
4000	0
5000	0

Table 1 – Number of resizes done for each initial hash table size.

Apart from the **hash table** size 100, in the **linked list** approach the time of execution is quite constant as the **hash table** size increases. This doesn´t happen with the **binary tree** approach where we can observe that the lowest times of execution occur when there is no resize at all (4000 and 5000).

Setting the initial size of the **hash table** at 2000, the time of execution of the programs was measured changing the increment on the **hash table** size when resizing. The results are shown below (figure 5).

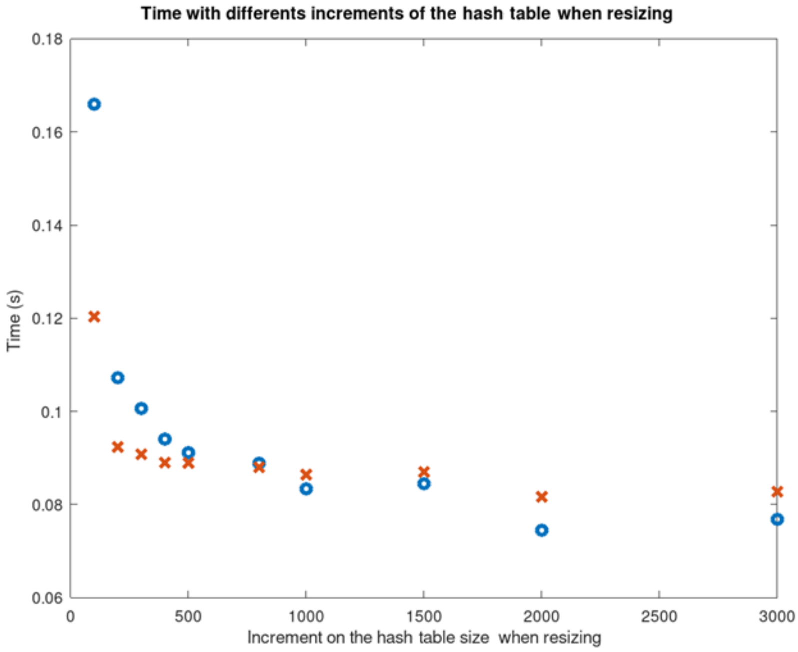

Figure 5- Time of execution with different hash table increments, for the binary tree approach (blue) and for the linked list approach (orange).

Hash table increment	Resizes
100	19
200	10
300	7
400	5
500	4
800	3
1000	2
1500	2
2000	1
3000	1

Table 2 – Number of resizes done for each hash table size increment.

Both approaches show a decrease of the time of execution when the increment is bigger (less resizes), however the **binary tree** approach decreases much faster having a time of execution lower than the **linked list** approach when the increment is bigger than 800.

Considering the results of the two graphs above we can conclude that if the number of elements to insert on a **hash table** is known (or belongs to a narrow interval), it is more efficient to use a **binary tree** approach, because searching an element will usually be faster, given that the tree is ordered. Meanwhile if the number of elements is unknown and it is necessary to resize the **hash table** multiple times the **linked list** approach is better because the cost of inserting a member should be taken account and insertion at the head is an $O(1)$ operation, way faster than inserting an element on an **binary tree**.

Appendix

Occurrences_tree.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "manipulate_file.h"
#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);
}


int main(int argc,char **argv){
    FILE *out= fopen("outBinaryTree_size.txt", "w");
    for(int x=1;x<argc;x++){
        file_data_t fd;
        int idx;
        int j=0;
        int i=0;
        int count=0;
        int r_count=0;
        (void)elapsed_time();
        hash_size=(unsigned int)atoi(argv[x]);
        struct hash_data_bt **hash_table=  malloc(hash_size*sizeof(struct hash_data));
        for(int k = 0;k < hash_size;k++) hash_table[k] = NULL;
        open_text_file("SherlockHolmes.txt", &fd);
        while(read_word(&fd) != -1) {
            idx = hash_function(fd.word, hash_size);
            hash_data_bt *hd=hash_table[idx];
            while (hd != NULL && strcmp(fd.word,hd->key) != 0)
             {
                 if(strcmp(fd.word,hd->key)<0 ){
                     hd=hd->left;
                 }
                 else
                 {
                     hd=hd->right;
                 }

                 i++;
             }

             if(hd == NULL ){
                 if(i==0){
                     hd=new_hash_data_bt();
                     strcpy(hd->key, fd.word);
                     hd->word.number_occurrences=1;
                     hd->word.first_appearence=count;
                     hd->word.last_appearence=count;
                     hd->right=NULL;
                     hd->left=NULL;
                     hash_table[idx]=hd;

                 }
```

```
                else{
                    hd=new_hash_data_bt();
                    hd->word.number_occurrences=1;
                    hd->word.first_appearence=count;
                    hd->word.last_appearence=count;
                    hd->right=NULL;
                    hd->left=NULL;
                    strcpy(hd->key, fd.word);
                    insert_bt(hash_table[idx],hd);
                }
                if (j>hash_size*5){

                    hash_table=hash_resize_bt(hash_table,1000u);
                    r_count++;
                }
                j++;
            }
            else {
                hd->word.number_occurrences+=1;
                if(hd->word.number_occurrences==2){
                    hd->word.s_distance=count - hd->word.first_appearence;
                    hd->word.l_distance=count - hd->word.first_appearence;
                    hd->word.t_distance=hd->word.s_distance;
                } else{
                    if(hd->word.s_distance > count-hd->word.last_appearence)
                        hd->word.s_distance=count-hd->word.last_appearence;
                    if(hd->word.l_distance < count-hd->word.last_appearence)
                        hd->word.l_distance=count-hd->word.last_appearence;
                    hd->word.t_distance+=count-hd->word.last_appearence;
                }
                hd->word.m_distance=hd->word.t_distance/(hd->word.number_occurrences -1);
                hd->word.last_appearence=count;
            }
            i=0;
            count++;
            /*
            printf("hash: %s %d \n", hash_table[idx]->key,hash_table[idx]->word.number_occurrences);
            if(hash_table[idx]->left!=NULL){
                printf("%s\n" , hash_table[idx]->left->key);
            }
            */
        }
        double cpu_time = elapsed_time();
        close_text_file(&fd);
        fprintf(out, "%d %.10f %d\n",(unsigned int)atoi(argv[x]), cpu_time,r_count);
        free(hash_table);

    }
    fclose(out);
    return 1;
}
```

Occurrences.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "manipulate_file.h"
#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);
}


int main(int argc,char **argv){
    //FILE *out= fopen("outLinked_inc.txt", "w");
    FILE *out= fopen("large_distance.txt", "w");
    for(int x=1;x<argc;x++){
        //printf("%d\n",(unsigned int)atoi(argv[x]));
        file_data_t fd;
        int idx;
        int j=0;
        int count=0;
        int i=0;
        int r_count=0;
        (void)elapsed_time();
        struct hash_data **hash_table=  malloc(hash_size*sizeof(struct hash_data));
        for(int k = 0;k < hash_size;k++) hash_table[k] = NULL;
        open_text_file("SherlockHolmes.txt", &fd);
        while(read_word(&fd) != -1) {
            idx = hash_function(fd.word, hash_size);
            hash_data *hd;
            for(hd = hash_table[idx];hd != NULL && strcmp(fd.word,hd->key) != 0;hd = hd->next)
                i++;
            if(hd == NULL ){
                if(i==0){
                    hd=new_hash_data();
                    strcpy(hd->key, fd.word);
                    hd->word.number_occurrences=1;
                    hd->word.first_appearence=count;
                    hd->word.last_appearence=count;
                    hd->next=NULL;
                    hash_table[idx]=hd;

                }
                else{
                    hd=new_hash_data();
                    hd->next=hash_table[idx];
                    strcpy(hd->key, fd.word);
                    hd->word.number_occurrences=1;
                    hd->word.first_appearence=count;
                    hd->word.last_appearence=count;
                    hash_table[idx]=hd;
                }
```

```
                if (j>hash_size*5){

                    hash_table=hash_resize(hash_table,(unsigned int)atoi(argv[x]));
                    r_count++;
                }
                j++;
            }
            else {
                hd->word.number_occurrences+=1;
                if(hd->word.number_occurrences==2){
                    hd->word.s_distance=count - hd->word.first_appearence;
                    hd->word.l_distance=count - hd->word.first_appearence;
                    hd->word.t_distance=hd->word.s_distance;
                } else{
                    if(hd->word.s_distance > count-hd->word.last_appearence)
                        hd->word.s_distance=count-hd->word.last_appearence;
                    if(hd->word.l_distance < count-hd->word.last_appearence)
                        hd->word.l_distance=count-hd->word.last_appearence;
                    hd->word.t_distance+=count-hd->word.last_appearence;
                }
                hd->word.m_distance=hd->word.t_distance/(hd->word.number_occurrences -1);
                hd->word.last_appearence=count;
            }
            i=0;
            count++;
            //printf("hash: %s %d \n", hash_table[idx]->key,hash_table[idx]->word.number_occurrences);
            //if(hash_table[idx]->next!=NULL){
            //    printf("%s\n" , hash_table[idx]->next->key);
            //}
        }
        double cpu_time = elapsed_time();
        close_text_file(&fd);
        //fprintf(out, "%d %.10f %d\n",(unsigned int)atoi(argv[x]), cpu_time,r_count);
        for(int l=0;l<hash_size;l++){
            while(hash_table[l]!=NULL){
                fprintf(out,"%d\n",hash_table[l]->word.l_distance);
                hash_table[l]=hash_table[l]->next;
            }

        }

        free(hash_table);
        hash_size=2000u;
    }
    fclose(out);
    return 1;
}
```

manipulate_file.h

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

// FILE STRUCT
typedef struct file_data_t {
    // 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;

// HASH-FUNCTION
unsigned int hash_function(const char *str, unsigned int s) {
    unsigned int h;
    for(h = 0u;*str != '\0';str++)
        h = 157u * h + (0xFFu & (unsigned int)*str); // arithmetic overflow may occur here (just ignore it!)
    return h % s; // due to the unsigned int data type, it is guaranteed that 0 <= h % s < s
}

// STRUCT THAT DEFINES THE STATS FOR THE WORDS IN THE HASH-TABLE
typedef struct word_stats {
    int number_occurrences;
    int first_appearence;
    int last_appearence;
    int s_distance; // smallest distance
    int l_distance; // largest distance
    int m_distance; //. medium distance
    int t_distance; // total distance
} word_stats;

// HASH-TABLE NODE LINKED LIST
typedef struct hash_data {
    struct hash_data *next;
    char key[64];
    struct word_stats word;
} hash_data;

// HASH-TABLE NODE BINARY TREE
typedef struct hash_data_bt {
    struct hash_data_bt *left;
    struct hash_data_bt *right;
    char key[64];
    struct word_stats word;
} hash_data_bt;
```

```c
// ALLOCATE NEW HASH-DATA
hash_data *new_hash_data(void) {
    hash_data *hd = (hash_data *)malloc(sizeof(hash_data));
    if(hd == NULL) {
        fprintf(stderr,"Out of memory\n");
        exit(1);
    }
    return hd;
}
// ALLOCATE NEW HASH-DATA
hash_data_bt *new_hash_data_bt(void) {
    hash_data_bt *hd = (hash_data_bt *)malloc(sizeof(hash_data_bt));
    if(hd == NULL) {
        fprintf(stderr,"Out of memory\n");
        exit(1);
    }
    return hd;
}

unsigned int hash_size= 2000u;

// INSERT
void insert_bt(hash_data_bt *root, hash_data_bt *hd ){
    if(strcmp(hd->key,root->key)<0)
        {
                if(root->left!=NULL)
                        insert_bt(root->left,hd);
                else
                        root->left=hd;
        }

        if(strcmp(hd->key,root->key)>0)
        {
                if(root->right!=NULL)
                        insert_bt(root->right,hd);
                else
                        root->right=hd;
        }
}

// OPEN FILE
int open_text_file(char *file_name,file_data_t *fd) {
    fd->fp = fopen(file_name,"r");
    if(fd->fp == NULL)
        return -1;
    fd->word_pos = -1;
    fd->word_num = -1;
    fd->word[0] = '\0';
    fd->current_pos = -1;
    return 0;
}


//RESIZE LINKED LIST
 struct hash_data ** hash_resize(struct hash_data **hash_table, unsigned int inc){
    //printf("Resizing..............................................................................................\n");
    hash_data *next;
    hash_size+=inc;
    struct hash_data **hash_table_new=  malloc((hash_size)*sizeof(struct hash_data));
    int new_idx;
```

```c
    for(int m = 0;m < hash_size;m++) hash_table_new[m] = NULL;
    for(int l=0;l<hash_size-inc;l++){
        while(hash_table[l]!=NULL){
            new_idx= hash_function(hash_table[l]->key, hash_size);
            next=hash_table[l]->next;
            hash_table[l]->next=hash_table_new[new_idx];
            hash_table_new[new_idx]=hash_table[l];
            hash_table[l]=next;
        }

    }
    free(hash_table);
    return hash_table_new;
}
//TRAVERSE_TREE
void traverse_tree(struct hash_data_bt *hb, struct hash_data_bt **hash_table){
    if(hb!=NULL){
        int new_idx= hash_function(hb->key, hash_size);
        if(hash_table[new_idx]==NULL){
            hash_table[new_idx]=new_hash_data_bt();
            strcpy(hash_table[new_idx]->key,hb->key);
            hash_table[new_idx]->left=NULL;
            hash_table[new_idx]->right=NULL;
            hash_table[new_idx]->word=hb->word;
        }
        else{
            hash_data_bt *temp =new_hash_data_bt();
            temp->left=NULL;
            temp->right=NULL;
            temp->word=hb->word;
            strcpy(temp->key,hb->key);
            insert_bt(hash_table[new_idx],temp);
        }
        traverse_tree(hb->left, hash_table);
        traverse_tree(hb->right, hash_table);
    }
}
//RESIZE BINARY TREE
 struct hash_data_bt ** hash_resize_bt(struct hash_data_bt **hash_table, unsigned int inc){
    //printf("Resizing...........................................................................................\n");
    hash_data_bt *next;
    struct hash_data_bt **hash_table_new=  malloc((hash_size+inc)*sizeof(struct hash_data_bt));
    int new_idx;
    hash_size+=inc;
    for(int m = 0;m < hash_size;m++) hash_table_new[m] = NULL;
    for(int l=0;l<hash_size-inc;l++){
        traverse_tree(hash_table[l],hash_table_new);
    }
    free(hash_table);
    return hash_table_new;
}
// CLOSE FILE
void close_text_file(file_data_t *fd) {
    fclose(fd->fp);
    fd->fp = NULL;
}
```

```c

// READ WORD
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 >= 48 && c<58) || (c>=65 && c<=90) || (c>=97 && c<=122) || (c>=192)));
    // record word
    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 >= 48 && c<58) || (c>=65 && c<=90) || (c>=97 && c<=122) || (c>=192)))
            break; // terminate word
        fd->word[i] = (char)c;
    }
    fd->word[i] = '\0';
    for(int j = 0; fd->word[j]; j++){
        fd->word[j] = tolower(fd->word[j]);
    }
    return 0;
}

```