AED - Algoritmos e Estruturas de Dados
Hash Table implementation

Hash Tables implementation using singly linked lists and binary trees

Dinis Cruz	Duarte Mortágua	Tiago Oliveira
92080	92963	93456
33,3%	33,3%	33,3%

Teacher: Tomás Oliveira e SIlva

Departamento de Eletrónica,

Telecomunicações e Informática

Universidade de Aveiro

december 27, 2019

Contents

List of Tables

List of Figures

1. Introduction

1.1 A brief description of the problem

To start of, the main goal of this practical work was the implementation of an hash-table. The purpose of an hash-table implementation is to store data, so we can see an hash-table as a data structure. With the hast-table implementation the main goal was to store all the different words in a determined text and how many times every single word occurred in that text in each linked list entry corresponding each hash-table entry. Besides, we were told to store in each linked-list entry the first and last locations of the word and the maximum, minimum and medium distances between words. As we had to store a lot of information about each distended word, we thought that was clever to use structures as the representation of each word and the linked lists entries would be this structures. The language we used to implement this was C, although we are not very familiar with it we think we have done a good job using it. As soon as we started thinking about the implementation we decided that the data that we were going to store would pass through an hash function first, however as we all know hashing methods always have collisions and as we want to have a efficient implementation we decided to use, in every hash-table entry, a linked list. Liked lists are basically separated elements that have a pointer to next element and so on, in this way every time we got an hashing collision we would put the next element in the linked list and the element that was already there with a pointer to it. As soon as we finish implementing the hash-table with linked lists we thought we could implement the hash-table but now with binary trees instead of linked lists to compare which one is the most efficient way.

2. Implementation methods

2.1 Singly linked lists implementation

As we described earlier, the linked list implementation is based in each element pointing to the next. Every time we have an hashing collision the linked list in that hashing position will grow. Nevertheless, every time we hash something to the hash-table we need to check if that position is still empty or not and if the word we are hashing is already in the hash-table or not. First of all, as soon as we have the hashing position that the word we are hashing will occupy, we go through the linked list in that position and in each element we check if it matches the word we are storing. If we find a match then we update all the data of that structure(which are what we used to represent each word) such as the number of times it occurred and the last, maximum, minimum and medium distances. On the other hand, while we are going trough the linked list if we reach a pointer that points do NULL then we know for sure that we are checking the last element of the linked list and as we have not got any match this means that the word we are hashing still does not exists in the hash-table so we need to add him. As we already described the adding in the linked list is basically changing the pointer of the last element of the linked list to point to the new element so the new element becomes the last element of the that linked list.

2.2 Binary trees implementation

The binary trees implementation is based in having more efficiency in the search. Different than the linked list implementation, every structure in the hash table has a pointer to a binary tree. We thought that it would not make sense if we used unordered binary trees because that's what makes the search efficient. Of course every binary tree node has the same information as the linked list structure, such as the number of times the word occurred and the last, minimum, maximum and medium distances and a few more details we found imperative. On one hand, the collisions problem is solved exactly the same way the linked list implementation is, every time we have a collision, if the hashing position has already a pointer to the head of the binary tree, we go through all the binary tree and once we arrive the end of the tree if we found no match to the word then we add a new node to the binary tree.

On the other hand, the adding of a new node is different. As the binary tree is ordered, every time we have to add a new node to a tree, we compare the word we are adding with the head, if it is lower we go left, if it is greater we go right, and we go through the binary tree until we find the exact spot that fits the word.

About the resize, we did not implement it a long with the binary tree implementation because the efficiency is in the capacity of the search being shorter, and so we think that a resize would not be a must need functionality (see 3.1.2).

3. Dynamic resizing

In order to reduce the number of collisions as the hash table gets filled, we implemented a dynamic resize function that doubles the size of the hash table and reallocates the words in the table. This action is triggered whenever the load factor is reached. The load factor is the number of linked lists (or binary trees) stored in the hash table divided by it's capacity, and in this case, we use a load factor of 0.5.

3.1 Approach description

Our hash table structure is not only characterized by the table itself, but by the elements "count" and "size" too. Whenever a word is read and occupies a new entry of the hash table, the count element is increased, so the count keeps track of the hash table slots that are not null. The size initially represents the number of slots of the hash table, equaling 2000. When $count/size >= 0.5$, the resize action is triggered.

3.1.1 Singly linked lists

We start the resize process by creating a new empty table of pointers to words (our new hash table). This new table will have the size equal to two times the old size. Then, we iterate thought the hash table entries. In each one, we detach the singly link list, keeping track of the head pointer in a new variable, "next". Then, as the size will be changed and so the hash-code too, we cant just move the entire linked list to a new entry of the new table, we have to detach the entire linked list and map each one of it's words to a new linked list in the new hash table. In order to do that, and having the "next" pointer, we iterate through all the words of each linked list, and since each word has an attribute "hash" (djb2 direct hash), we get the index of the word's new entry in the new hash table by getting the rest of the division of the hash by the size of the new hash table. After getting the index, instead of appending the word to the end of the new linked list, we attach it to the beginning. We do this by causing the new word to point to the first word in the list, and then replace the first word in the list with the new word.

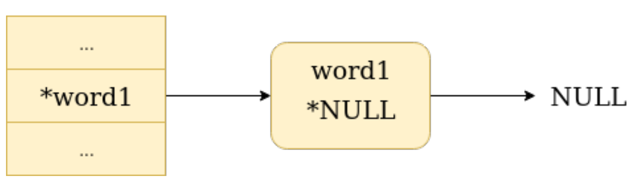

Figure 3.1: Adding word1 to empty linked list

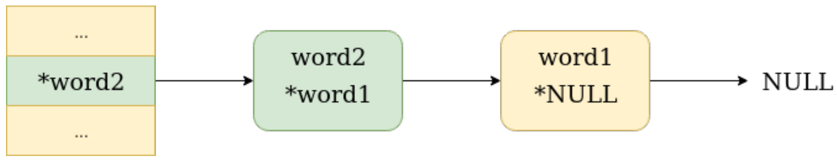

Figure 3.2: Adding word2 to linked list containing word1

After doing this to all the linked lists in the old hash table, we free the memory of the old hash table, replace it by the new one and update the hash table size with the new size.

3.1.2 Binary trees

We decided not to implement dynamic resize when implementing the hash table with binary trees because:

- When talking about adding elements, resizing the table or not, the time complexity is $O(1)$ for both implementations, being that the dynamic resize would only have impact in time execution when searching for a word in the table.
- Each time we do the dynamic resize, in linked lists, we go through all the words in all the old lists one by one ($O(n)$) and add them to the new linked lists in the new table ($O(1)$), having a total complexity of ($O(n)$). On the other hand, in binary trees, we go through all the words in all the old trees one by one ($O(n)$) and then insert them on the correct position in the new binary trees ($O(n)$), having a total complexity of ($O(n^2)$). So, resizing when the

objective is adding words to the hash table isn't such a great idea when talking about binary trees implementation.

- Ignoring dynamic resize in binary trees doesn't affect the complexity that much when searching, because the fact of using ordered binary trees, although huge, decreases the search time of O(n) of the linked lists search to $O(logn)$, so it becomes fast enough to keep up with the search when dealing with linked lists (resized).

In sum, since the proposed problem was to add words to the hash table, we thought it would be less time expensive to implement an hash table with singly linked lists with dynamic resizing and an hash table with binary trees without dynamic resizing.

4. Tests

4.1 Singly linked lists tests

4.1.1 Output

To conclude, as we finished the linked lists implementation we started testing it so we could analyze if our implementation was doing what it was supposed to and also to prove that it was actually working. To help us doing the testing we used C asserts. First of all we made the program print the whole hash-table along with each words characteristics. We have also printed the number of words read, the number of words inserted in the hash-table, the size of the hash-table and the number hash-table slots that were occupied just for us to have an idea during the testing time, if we run the program with the same text file these numbers are supposed to be the same.

Next test was to verify if every single word read was in the correct hash-table position, so after the hash-table was completed we read the text again and hashed every single word again and compare it to the word that was already in that position so we could assure the words were hashed correctly.

Finally the last test we have made was the top 10 words with more occurrences in the text file and the next step was to compare it with the binary tree (that will be approached in another topic). We also made a test only in linked lists that is commented because it takes to long to run (about 20min), which is to check if every single word occurs in the hash table once and only once. As we have so much words to read and search, this test has to check the whole hash-table the number of times as the number of words it would read so this explains the 20 min time running.

This tests were written along with the hash-table count in the end of the program so they would be showed as we run the program in the output.

We have also calculated some execution times, and concluded that the insertion of all the Sherlock Holmes book words (657438 words) takes about 0.11s (almost the same as the binary trees).

4.2 Binary trees solution

4.2.1 Output

The tests we have made about the binary tree implementation were exactly the same as in the linked list implementation, the only difference was in the way we go trough the hash-table as with binary trees it works different and also in the hash-table print, here we printed it in a graphical way so the words would actually be displayed in a tree format.

4.3 Solution comparison

To sum up, when the whole testing session was done we compared the execution times, concluding that the hash table implementation with binary trees (not resizable) takes near 0.14s and that the hash tables implementation with singly linked lists (resizable) takes near 0.11s to run.

A. Code appendix

Below is the code of our ht_sll.c and ht_bt.c files respectively. All the comments made by us are in capital letter. Some comments made by the teacher were omitted in order to keep this appendix concise. These comments were no longer relevant since they were supposed to guide our code work in an early stage.

A.1 Hash Tables with Singly Linked Lists (ht_sll.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>
#include <time.h>

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;

//Representa cada palavra distinta num determinado ficheiro
typedef struct word {
    struct word* next;
    char word[64];
    unsigned long hash;
    int first_location;
    int last_location;
    int max_dist;
    int min_dist;
    int medium_dist;
    int count;
} word_t;

typedef struct hash_table {
    unsigned int size;
    unsigned int count;
    word_t** table;
} hash_table_t;

int open_text_file(char* file_name, file_data_t* fd)
```

```
{
    fd->fp = fopen(file_name, "r");
    if (fd->fp == NULL){
        printf("File does not exist.\n");
        return -1;
    }
    fd->word_pos = -1;
    fd->word_num = -1;
    ;
    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;
    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;
}

unsigned long hash(unsigned char* str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    return abs(hash);
}

int main(int argc, char* argv[])
{
    file_data_t* fl;
    fl = (file_data_t*)malloc(sizeof(file_data_t));

    if (open_text_file(argv[1], fl) == -1) {
        return EXIT_FAILURE;
    }
    double time_spent_total = 0;

    clock_t begin_total = clock();

    // HASHTABLE INITIALIZATION
    hash_table_t* hash_table = NULL;
    hash_table = malloc(sizeof(hash_table_t));
    hash_table->table = malloc(2000 * sizeof(word_t*));
    hash_table->size = 2000;
```

```c
    hash_table->count = 0;
    for (int i = 0; i < hash_table->size; i++) {
        hash_table->table[i] = NULL;
    }

    // POINTERS DECLARATION TO USE INSIDE WHILE CYCLE
    word_t* head;
    word_t* prev;
    head = (word_t*)malloc(sizeof(word_t));
    prev = (word_t*)malloc(sizeof(word_t));

    int hashcode=0;
    int word_counter=0;
    int resize_counter=0;
    double time_spent_resize = 0;

    while (read_word(fl) != -1) {

        //DYNAMIC RESIZE
        if (hash_table->count >= hash_table->size / 2) {
            clock_t begin_resize = clock();
            word_t **table, *curr, *next;
            size_t i, k;
            next = malloc(sizeof(word_t));
            curr = malloc(sizeof(word_t));
            int new_size = hash_table->size * 2;
            table = malloc(new_size * sizeof(word_t*));
            if (!table)
                return -1; // OUT OF MEMORY

            // INITIALIZE NEW TABLE TO EMPTY
            for (i = 0; i < new_size; i++) {
                table[i] = NULL;
            }

            for (i = 0; i < hash_table->size; i++) {
                // DETACH THE SINGLY LINKED LIST
                next = hash_table->table[i];
                hash_table->table[i] = NULL;
                while (next) {
                    // DETACH THE NEXT ELEMENT AS CURRENT
                    curr = next;
                    next = next->next;

                    // K IS THE INDEX OF CURR IN THE NEW TABLE
                    k = curr->hash % new_size; // o curr->hash    o resultado da word em curr ao passar pela f u n   o hash()

                    // PREPEND TO THE LINKED LIST IN TABLE[K]
                    if (curr != table[k]) {
                        curr->next = table[k];
                        table[k] = curr;
                    }
                }
            }
            // NO LONGER NEED NEXT AND CURR
            free(next);
            free(curr);

            // NO LONGER NEED THE OLD HASH TABLE
            free(hash_table->table);

            // REPLACE THE OLD HASH TABLE WITH THE NEW ONE
            hash_table->table = table;
            hash_table->size = new_size;
            clock_t end_resize = clock();
            time_spent_resize = (double)(end_resize - begin_resize) / CLOCKS_PER_SEC;
            resize_counter++;
        }

        word_counter++;
        int flag = 0; //IF A WORD IS FOUND, THEN WE DONT NEED TO CREATE IT
        hashcode = hash(fl->word) % hash_table->size;
        head = hash_table->table[hashcode];
        if (head == NULL) { // IF THERES NOTHING IN TABLE[HASHCODE], CREATE A NEW WORD THERE
            word_t* new;
            new = (word_t*)malloc(sizeof(word_t));
```

```
            new->next = NULL;
            new->hash = hash(fl->word);
            new->first_location = fl->current_pos;
            new->last_location = fl->current_pos;
            new->max_dist = NULL;
            new->min_dist = NULL;
            new->medium_dist = 0;
            new->count = 1;
            strcpy(new->word, fl->word);
            hash_table->table[hashcode] = new;
            hash_table->count += 1;
        }
        else {
            while (head != NULL) {
                if (strcmp(head->word, fl->word) == 0) { // IF MATCH IS FOUND
                    flag = 1; // MATCH FOUND
                    int temp = head->last_location;
                    int dist = fl->current_pos - temp;
                    head->last_location = fl->current_pos;
                    if (dist > head->max_dist || head->max_dist == NULL) {
                        head->max_dist = dist;
                    }
                    if (dist < head->min_dist || head->min_dist == NULL) {
                        head->min_dist = dist;
                    }
                    head->medium_dist = head->medium_dist + (dist - head->medium_dist) / head->count;
                    head->count++;
                    break;
                }
                prev = head;
                head = head->next;
            }
            if (flag == 0) { // MATCH WAS FOUND? IF NOT, CREATE NEW WORD AND ATTACH IT TO THE LAST WORD (PREV)
                word_t* new;
                new = (word_t*)malloc(sizeof(word_t));
                new->next = NULL;
                new->hash = hash(fl->word);
                new->first_location = fl->current_pos;
                new->last_location = fl->current_pos;
                new->max_dist = NULL;
                new->min_dist = NULL;
                new->medium_dist = 0;
                new->count = 1;
                strcpy(new->word, fl->word);
                prev->next = new;
            }
        }
    }
    clock_t end_total = clock();
    time_spent_total = (double)(end_total - begin_total) / CLOCKS_PER_SEC;

    // HASHING DONE //

    // TESTING //
    int new_words = 0;
    word_t* word;

    // PRINT HASH TABLE
    for (int k = 0; k < hash_table->size; k++) {
        printf("%d: ", k);
        if (hash_table->table[k] == NULL) {
            printf("NULL\n");
        }
        else {
            word = hash_table->table[k];
            while (word->next != NULL) {
                printf("%s (FL: %d, LL: %d, MAXD: %d, MIND: %d, MEDD: %d, WC: %d) --> ", word->word, word->first_location, word->last_location, word->max_dist, word->min_dist, word->medium_dist, word->count);
                word = word->next;
                new_words++;
            }
            printf("%s (FL: %d, LL: %d, MAXD: %d, MIND: %d, MEDD: %d, WC: %d) --> NULL\n", word->word, word->first_location, word->last_location, word->max_dist, word->min_dist, word->medium_dist, word->count);
            new_words++;
        }
    }

```

```
    printf("===================================\n");
    printf("TABLE STATS\n");
    printf("Words read: %d\n", word_counter);
    printf("Hash table count: %d\n", hash_table->count);
    printf("Hash table size: %d (resized %d times)\n", hash_table->size, resize_counter);
    printf("Total duration: %5.4fs\n", time_spent_total);
    printf("Duration of last resize: %5.4fs\n", time_spent_resize);
    printf("Words inside table: %d\n", new_words);



    // SEARCH TEST // --> ASSERT THAT WORD IS IN HASHTABLE

    file_data_t* ft;
    ft = (file_data_t*)malloc(sizeof(file_data_t));
    word_t* head_test;
    head_test = (word_t*)malloc(sizeof(word_t));
    if (open_text_file("Teste.txt", ft) == -1) {
        return EXIT_FAILURE;
    }

    int flag_search = 0;
    while (read_word(ft) != -1) {
        int hashcode_test=0;
        hashcode_test = hash(fl->word) % hash_table->size;
        head_test = hash_table->table[hashcode];
        while (head_test) {
           if (strcmp(head_test->word, fl->word) == 0){
               flag_search++;
           }
           head_test = head_test->next;
        }

    }

    // // SEARCH TEST // --> ASSERT THAT A WORD APPEARS ONCE AND ONLY ONCE IN THE HASH TABLE (COMMENTED BACAUSE TAKES TOO MUCH TIME)

    // file_data_t* ft;
    // ft = (file_data_t*)malloc(sizeof(file_data_t));
    // word_t* head_test;
    // head_test = (word_t*)malloc(sizeof(word_t));
    // if (open_text_file("Teste.txt", ft) == -1) {
    //     return EXIT_FAILURE;
    // }


    // while (read_word(ft) != -1) {
    //     int hashcode_test=0;
    //     int flag_search = 0;

    //     for (int k = 0; k < hash_table->size; k++){
    //         head_test = hash_table->table[k];
    //         while (head_test) {
    //             if (strcmp(head_test->word, fl->word) == 0){
    //                 flag_search++;
    //             }
    //         head_test = head_test->next;
    //         }
    //     }
    //     assert(flag_search==1);
    // }


    // WORDS INSIDE TABLE TEST //

    int test_counter=0;
    word_t* head_test_2;
    head_test_2 = (word_t*)malloc(sizeof(word_t));

    for (int k = 0; k < hash_table->size; k++){
        head_test_2 = hash_table->table[k];
        while (head_test_2) {
           test_counter++;
           head_test_2 = head_test_2->next;
        }
    }
```

```c

    assert(new_words == test_counter);

    // TOP 10 MORE FREQUENT WORDS //

    char a[10][64];
    int max_test = 0;
    word_t* head_test_3;
    char word_test[64];
    int maxs[10];
    head_test_3 = (word_t*)malloc(sizeof(word_t));
    for (int j = 0; j < 10; j++){
        max_test = 0;
        for (int k = 0; k < hash_table->size; k++){
            head_test_3 = hash_table->table[k];
            while (head_test_3) {
                if (head_test_3->count > max_test){
                    int flagg = 1;
                    for (int l = 0; l < j; l++){
                        if (strcmp(a[l],head_test_3->word) == 0){
                            flagg = 0;
                            break;
                        }
                    }
                    if (flagg){
                        max_test = head_test_3->count;
                        memset(word_test, 0, 64);
                        strcpy(word_test, head_test_3->word);
                    }
                }
                head_test_3 = head_test_3->next;
            }
        }

        strcpy(a[j], word_test);
        maxs[j] = max_test;

    }
    printf("===================================\n");
    printf("TOP 10 MOST FREQUENT WORDS\n");
    for (int k = 0; k < 10; k++){
        printf("%-5s (%d)\n", a[k], maxs[k]);
    }
    return 0;

}
```

A.2 Hash Tables with Binary Trees (ht_bt.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>
#include <time.h>

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;

//Representa cada palavra distinta num determinado ficheiro
typedef struct word {
    struct word* left;
    struct word* right;
    char word[64];
    int hash;
    int first_location;
```

```
    int last_location;
    int max_dist;
    int min_dist;
    int medium_dist;
    int count;
} word_t;

typedef struct hash_table {
    unsigned int size;
    unsigned int count;
    word_t** table;
} hash_table_t;

int open_text_file(char* file_name, file_data_t* fd)
{
    fd->fp = fopen(file_name, "r");
    if (fd->fp == NULL){
        printf("File does not exist.\n");
        return -1;
    }
    fd->word_pos = -1;
    fd->word_num = -1;
    ;
    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;
    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;
}

unsigned long
hash(unsigned char* str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return abs(hash);
}
void print2DUtil(word_t* root, int space)
{
```

```
    // Base case
    if (root == NULL) {
        printf("\n");
        for (int i = 10; i < space + 10; i++)
            printf(" ");
        printf("%15s\n", NULL);
        return;
    }
    // Increase distance between levels
    space += 10;

    // Process right child first
    print2DUtil(root->right, space);

    // Print current node after space
    // count
    printf("\n");
    for (int i = 10; i < space; i++)
        printf(" ");
    printf("%15s\n", root->word);

    // Process left child
    print2DUtil(root->left, space);
}

void visit(word_t* word)
{
    printf("%s (FL: %d, LL: %d, MAXD: %d, MIND: %d, MEDD: %d, WC: %d)\n", word->word, word->first_location, word->last_location, word->max_dist, word->min_dist, word->medium_dist, word->count);
}

word_t* search_recursive(word_t* link, char* data)
{
    if (link == NULL || strcmp(link->word, data) == 0)
        return link;
    if (strcmp(link->word, data) > 0)
        return search_recursive(link->left, data);
    else
        return search_recursive(link->right, data);
}

void insert_non_recursive(word_t** link, word_t** insert, char* data)
{
    word_t* parent = NULL;
    while (*link != NULL) {
        parent = *link;
        link = (strcmp((*link)->word, data) > 0) ? &((*link)->left) : &((*link)->right);
    }
    *link = *insert;
}

void traverse_in_order_recursive(word_t* link)
{
    if (link != NULL) {
        traverse_in_order_recursive(link->left);
        //printf("--------------------------\n");
        visit(link);
        //printf("--------------------------\n");
        traverse_in_order_recursive(link->right);
    }
}
void most_used_words(hash_table_t* hash_table)
{
    word_t* head_test;
    word_t* checker;
    head_test = (word_t*)malloc(sizeof(word_t));
    checker = (word_t*)malloc(sizeof(word_t));
    int max_prev = __INT_MAX__;
    int max_in_cicle = 0;
    int occ;
    char palavra[64] = "word";
    for (int c = 0; c < 10; c++) {
        max_in_cicle = 0;
        for (int f = 0; f < hash_table->size; f++) {

            checker = hash_table->table[f];
            if (checker->count < max_prev) {
```

```c
                if (checker->count > max_in_cicle) {
                    strcpy(palavra, checker->word);
                    max_in_cicle = checker->count;
                    occ = checker->count;
                }
            }
        }
        printf("%-5s (%d)\n", palavra, occ);
        max_prev = max_in_cicle;
    }
}

int main(int argc, char* argv[])
{
    file_data_t* fl;
    fl = (file_data_t*)malloc(sizeof(file_data_t));

    if (open_text_file(argv[1], fl) == -1) {
        return EXIT_FAILURE;
    }

    double time_spent_total = 0;

    clock_t begin_total = clock();

    // HASHTABLE INITIALIZATION
    hash_table_t* hash_table = NULL;
    hash_table = malloc(sizeof(hash_table_t));
    hash_table->table = malloc(2000 * sizeof(word_t*));
    hash_table->size = 2000;
    hash_table->count = 0;
    for (int i = 0; i < 2000; i++) {
        hash_table->table[i] = NULL;
    }

    // HEAD DECLARATION TO USE INSIDE WHILE CYCLE
    word_t* head;
    head = (word_t*)malloc(sizeof(word_t));
    int counterrr = 0;
    int hashcode = 0;
    int word_counter = 0;
    while (read_word(fl) != -1) {
        word_counter++;
        hashcode = hash(fl->word) % hash_table->size;
        head = hash_table->table[hashcode];
        if (head == NULL) {
            word_t* new;
            new = (word_t*)malloc(sizeof(word_t));
            new->left = NULL;
            new->right = NULL;
            new->hash = hash(fl->word);
            new->first_location = fl->current_pos;
            new->last_location = fl->current_pos;
            new->max_dist = NULL;
            new->min_dist = NULL;
            new->medium_dist = 0;
            new->count = 1;
            strcpy(new->word, fl->word);
            hash_table->table[hashcode] = new;
            hash_table->count += 1;
            counterrr++;
        }
        else {
            word_t* this_one;
            this_one = search_recursive(head, fl->word);
            if (this_one != NULL) {
                int temp = this_one->last_location;
                int dist = fl->current_pos - temp;
                this_one->last_location = fl->current_pos;
                if (dist > this_one->max_dist || this_one->max_dist == NULL) {
                    this_one->max_dist = dist;
                }
                if (dist < this_one->min_dist || this_one->min_dist == NULL) {
                    this_one->min_dist = dist;
                }
                this_one->medium_dist = this_one->medium_dist + (dist - this_one->medium_dist) / this_one->count;
                this_one->count++;
```

```
            }
            else {
                word_t* new1;
                new1 = (word_t*)malloc(sizeof(word_t));
                new1->left = NULL;
                new1->right = NULL;
                new1->hash = hash(fl->word);
                new1->first_location = fl->current_pos;
                new1->last_location = fl->current_pos;
                new1->max_dist = NULL;
                new1->min_dist = NULL;
                new1->medium_dist = 0;
                new1->count = 1;
                strcpy(new1->word, fl->word);
                insert_non_recursive(&head, &new1, new1->word);
                counterrr++;
            }
        }
    }

    clock_t end_total = clock();
    time_spent_total = (double)(end_total - begin_total) / CLOCKS_PER_SEC;

    word_t* word;
    for (int k = 0; k < hash_table->size; k++) {
        printf("==================\n");
        if (hash_table->table[k] == NULL) {
            printf("NULL\n");
        }
        else {
            word = hash_table->table[k];
            traverse_in_order_recursive(word);
            //print2DUtil(word,0);
        }
        printf("==================\n");
        printf("LINE: %d ABOVE\n", k);
    }
    printf("==================\n");
    printf("TABLE STATS\n");
    printf("Words read: %d\n", word_counter);
    printf("Hash elements count: %d\n", hash_table->count);
    printf("Hash elements size: %d\n", hash_table->size);
    printf("Total duration: %5.4fs\n", time_spent_total);
    printf("Number of words inside hash table: %d\n", counterrr);

    file_data_t* ft;
    ft = (file_data_t*)malloc(sizeof(file_data_t));
    word_t* head_test;
    word_t* checker;
    head_test = (word_t*)malloc(sizeof(word_t));
    checker = (word_t*)malloc(sizeof(word_t));
    if (open_text_file("Teste.txt", ft) == -1) {
        return EXIT_FAILURE;
    }
    int count_words = 0;
    int hashcode_test = 0;
    while (read_word(ft) != -1) {
        hashcode_test = 0;
        int flag_test = 0;

        hashcode_test = hash(fl->word) % hash_table->size;
        head_test = hash_table->table[hashcode_test];
        checker = search_recursive(head_test, fl->word);
        if (checker != NULL) {
            flag_test = 1;
            count_words = count_words + checker->count;
        }
        assert(flag_test);
    }
    assert(count_words == word_counter);
    printf("==================\n");
    printf("TOP 10 MOST FREQUENT WORDS\n");
    most_used_words(hash_table);
    return 0;
}
```