universidade de aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Words Statistics

Autores:

Nome: Pedro Santos	**Nº93221**	**-**	**33,3%**
Nome: Pedro Amaral	**Nº93118**	**-**	**33,3%**
Nome: Diogo Cunha	**Nº95278**	**-**	**33,3%**

Curso: Engenharia Informática

Aveiro
2019

Índice

1 Introdução

Para este trabalho prático foi-nos proposto o processamento de palavras presentes num documento de texto, retirando diversas informações e estatísticas como:

- Número de ocorrências de cada palavra distinta;
- Localização das primeira e última ocorrências de cada palavra distinta;
- Distância menor, média e maior entre ocorrências consecutivas da mesma palavra.

A implementação deverá:

- Usar uma Hash Table com Separate Chaining;
- O tamanho da Hash Table deve crescer dinamicamente;
- Cada entrada da Hash Table deve apontar para uma Linked List ou Ordered Binary Tree.

2 Estruturas Utilizadas

2.1 Linked List

Uma Linked List é uma estrutura de dados dinâmica, composta por 1 ou mais nodes. Cada node contem:

- a informação em si;
- um ponteiro para o próximo node de informação.

Na nossa implementação a linked list e singly-linked list isto e não usamos ponteiros aceder ao node anterior ao atual.

Para a criação de uma node temos a função *new_Node()* que cria novos nodes apartir de uma palavra e da posição que ela ocupa, depois para inserir informação numa linked list temos a função *insert()* que percorre todos os nodes e decide se já existe um node para essa palavra ou não se não usa a função *new_Node()*, caso já exista um node para essa palavra apenas vai aceder a essa informação e atualiza-la.

2.2 Binary Tree

Uma Binary Tree ou Árvore Binária é uma estrutura de dados dinâmica em forma de árvore invertida composta por nodes. Cada node contém:

- a informação em si;
- um ponteiro para o lado esquerdo, *left*, que no caso de ser ordenada aponta para um node com informação menor;
- um ponteiro para o lado direito, *right*, que no caso de ser ordenada aponta para um node com informação maior;
- opcionalmente, um ponteiro para o node pai, *parent*.

Na nossa implementação não achámos necessário o ponteiro para o node pai e portanto os nodes da nossa árvore binária, os quais correspondem à estrutura *tree_node* contêm apenas a informação e os ponteiros *left* e *right*. Além disso decidimos usar uma árvore binário ordenada.

Relacionada com esta estrutura temos a função *new_tree_node()* que devolve um *tree_node* recebendo como argumentos uma palavra e a posição que ocupa no ficheiro. Esta função aloca então memória para a estrutura verificando depois se foi alocada. Caso não seja alocada o programa fecha, caso contrário prossegue o rumo normal inicializando

as várias variáveis contidas na tree_node. Para inserir informação na árvore binário temos a função *insertT()* que tenta encontrar a palavra e caso encontre atualiza as informações, caso contrário usa a função *new_tree_node()* para acrescentar um novo node à árvore binária.

2.3 Hash Table

Uma Hash Table é uma estrutura de dados dinâmica que armazena dados de uma maneira associativa, cada valor tem um índice único (key) que não é obrigatoriamente um inteiro, pode ser, por exemplo, uma string.

Existem 2 métodos de guardar dados numa Hash Table, Open Addressing ou Separate Chaining, na nossa implementação utilizámos o método de Separate Chaining, no qual distribuímos os dados o mais uniformemente possível pelo número de elementos ("*int size*") da tabela, estes ficam armazenados numa estrutura que tanto pode ser, por exemplo, uma Linked List ou uma Binary Tree.

Para o desempenho ser otimizado, é importante, como referido acima, que os elementos sejam distribuídos o mais uniformemente possível, para isto utilizamos uma Hash Function muito eficiente, que se encontra nos slides da disciplina.

Na nossa implementação utilizámos ambas uma Hash Table com Linked List ("*hash_table*") e uma Hash Table com Binary Tree ("*hash_table_tree*"), cada uma com a sua respetiva estrutura na forma da variável "*table*", um inteiro "*used*" o qual indica o número de espaços usados e um inteiro "*size*" que indica o tamanho da tabela.

Para diminuir a quantidade de espaços inutilizados, ambas as implementações da Hash Table são dinâmicas, aumentando o tamanho total de espaços sempre que o número de espaços utilizados (variável *used*) ultrapassa 80% do número de espaços disponíveis (variável *size*).

2.3.1 Hash Table com Linked List

A função *new_hash_table()* devolve uma Hash Table que utiliza Linked List, tendo como parâmetro o tamanho pretendido.

A função *resize_HashTable()* é utilizada para aumentar o tamanho da tabela, criando uma nova tabela com mais 50% do tamanho atual e movendo para esta todos os elementos da antiga. Por fim, liberta-se o espaço alocado para a antiga e guardamos na variável

hashTable um ponteiro para a nova estrutura.

2.3.2 Hash Table com Binary Tree

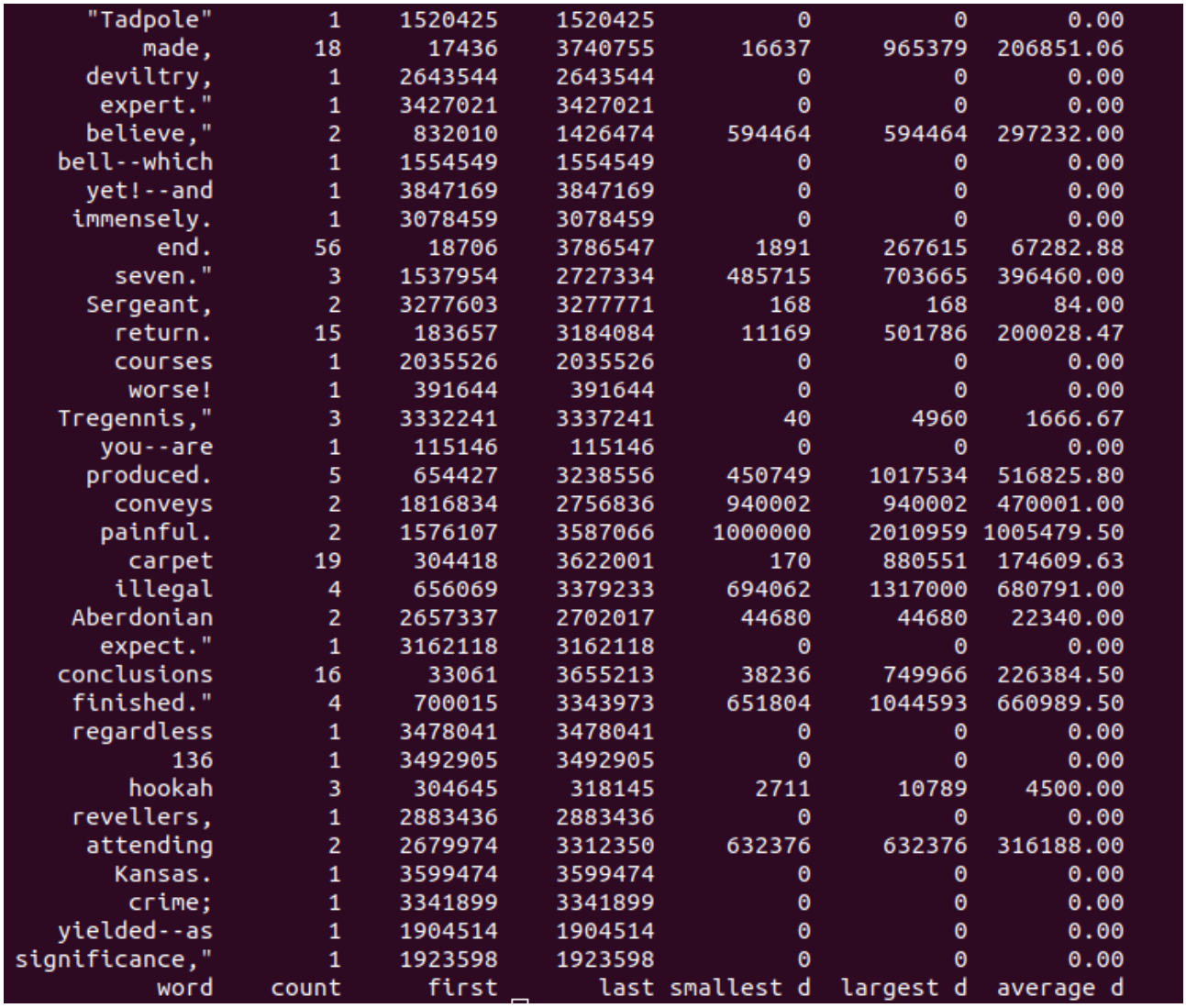
A função *new_hash_table_tree()* devolve uma Hash Table que utiliza Binary Tree, tendo como parâmetro o tamanho pretendido.

A função *resize_HashTableT()* é utilizada para aumentar o tamanho da tabela, criando uma nova tabela com mais 50% do tamanho atual. Para mover os elementos da tabela anterior para a nova tabela utilizamos a função *move_recursive()* que recursivamente irá encontrar a posição certa para cada elemento e irá inseri-lo nessa posição. Por fim, de forma semelhante à *resize_HashTable()* liberta-se o espaço alocado para a antiga tabela e guarda-se em hashTableT um ponteiro para a nova estrutura.

3 A Implementação

O nosso programa começa por abrir o ficheiro e pedir ao utilizador se quer utilizar uma Hash Table com Linked Lists ou Binary Trees. Após a escolha todas as palavras são lidas para a HashTable e por fim é impressa uma tabela com as várias palavras, o número de vezes que a palavra aparece, a primeira e última ocorrência e a maior, menor e média distância entre ocorrências consecutivas.

Para validar a nossa solução testamos o nosso programa com ficheiros de menor tamanho e também contando o número de palavras lidas verificando se é igual à soma das contagens dos elementos da Hash Table.



word	count	first	last	smallest d	largest d	average d
"Tadpole"	1	1520425	1520425	0	0	0.00
made,	18	17436	3740755	16637	965379	206851.06
deviltry,	1	2643544	2643544	0	0	0.00
expert."	1	3427021	3427021	0	0	0.00
believe,"	2	832010	1426474	594464	594464	297232.00
bell--which	1	1554549	1554549	0	0	0.00
yet!--and	1	3847169	3847169	0	0	0.00
immensely.	1	3078459	3078459	0	0	0.00
end.	56	18706	3786547	1891	267615	67282.88
seven."	3	1537954	2727334	485715	703665	396460.00
Sergeant,	2	3277603	3277771	168	168	84.00
return.	15	183657	3184084	11169	501786	200028.47
courses	1	2035526	2035526	0	0	0.00
worse!	1	391644	391644	0	0	0.00
Tregennis,"	3	3332241	3337241	40	4960	1666.67
you--are	1	115146	115146	0	0	0.00
produced.	5	654427	3238556	450749	1017534	516825.80
conveys	2	1816834	2756836	940002	940002	470001.00
painful.	2	1576107	3587066	1000000	2010959	1005479.50
carpet	19	304418	3622001	170	880551	174609.63
illegal	4	656069	3379233	694062	1317000	680791.00
Aberdonian	2	2657337	2702017	44680	44680	22340.00
expect."	1	3162118	3162118	0	0	0.00
conclusions	16	33061	3655213	38236	749966	226384.50
finished."	4	700015	3343973	651804	1044593	660989.50
regardless	1	3478041	3478041	0	0	0.00
136	1	3492905	3492905	0	0	0.00
hookah	3	304645	318145	2711	10789	4500.00
revellers,	1	2883436	2883436	0	0	0.00
attending	2	2679974	3312350	632376	632376	316188.00
Kansas.	1	3599474	3599474	0	0	0.00
crime;	1	3341899	3341899	0	0	0.00
yielded--as	1	1904514	1904514	0	0	0.00
significance,"	1	1923598	1923598	0	0	0.00

Fig. 1 Parte do output da opção 2 (Binary Tree) utilizando o ficheiro "SherlockHolmes.txt"

4 Anexo (Código Utilizado)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define n_big 1000000;

// Função de Dispersão
unsigned int hash_function(const char *str,unsigned int s)
{
	static unsigned int table[256];
	unsigned int crc,i,j;
	if(table[1] == 0u) // do we need to initialize the table[] array?
	{
		for(i = 0u;i < 256u;i++)
			for(table[i] = i,j = 0u;j < 8u;j++)
				if(table[i] & 1u)
					table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant
				else
					table[i] >>= 1;
	}
	crc = 0xAED02019u; // initial value (chosen arbitrarily)
	while(*str !='\0')
		crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((unsigned int)*str++ << 24);
	return crc % s;}
//---------------Hash Table com Linked List---------------
// Global Variable
struct hash_table *hashTable;

// Linked List
typedef struct Node {
	char word[64];
	int word_count;
   int first_occurence;
   int last_ocurrence;
   int smallest_distance;
   int largest_distance;
   double average_distance;
   struct Node* next;
} Node;

Node *new_Node(char *data, int pos) {
   Node* nNode = malloc(sizeof(Node));
   if(nNode == NULL) { //verificar se foi alocada memória
		fprintf(stderr,"Out of memory\n");
		exit(0);
   }
	nNode->next = NULL;
	strcpy(nNode->word, data);
	nNode->word_count = 1;
   nNode->first_occurence = pos;
   nNode->last_ocurrence = pos;
   nNode->smallest_distance = n_big;
   nNode->largest_distance = 0;
   nNode->average_distance = 0.0;
   return nNode;
}

// Hash Table
typedef struct hash_table {
   struct Node **table;
   int used; //numero de nodes da table usados
   int size; //numero total de nodes na table
} hash_table;

hash_table *new_hash_table(int size) { // cria uma nova hashtable com size elementos
   hash_table *ht = malloc(sizeof(hash_table));
   if(ht == NULL) { //verificar se foi alocada memória
		fprintf(stderr,"Out of memory\n");
		exit(0);
   }
   ht->table = malloc(size * sizeof(Node));
	if(ht->table == NULL) { //verificar se foi alocada memória
		fprintf(stderr,"Out of memory\n");
		exit(0);
   }
   ht->used = 0;
   ht->size = size;
	return ht;
}

void resize_HashTable() {
   hash_table *newTable = new_hash_table(1.5*hashTable->size); //nova hash_table

   for (int i=0; i<hashTable->size; i++) {
      Node *l1 = hashTable->table[i]; //nodes a transferir

      if (l1!=NULL) {
         do {
            int ind = hash_function(l1->word, newTable->size); //indice na nova hashtable
            //se nesse indice já houver nodes
            if (newTable->table[ind]!=NULL) {
               Node *l2 = newTable->table[ind]; //node que vai receber
               while (l2->next!=NULL) l2 = l2->next; //encontrar a node com next=NULL
               l2->next = l1; //passar de NULL para a node do antiga table
               l1 = l1->next;
               l2->next->next=NULL; //a node transferida deixa de apontar para outras nodes
            //se não houver nodes
            } else {
               newTable->table[ind]=l1;//a hash_table passa a apontar para esta node
               l1 = l1->next;
               newTable->table[ind]->next=NULL;
               newTable->used++;
            }
         } while (l1!=NULL);
      }
   }

   free(hashTable->table);
   free(hashTable);
   hashTable = newTable;
}

void insert(int key, char *data, int pos) { // insere novo valor na hashtable
   // se a lista não estiver vazia
```

```
    if (hashTable->table[key] != NULL) {
        // percorrer todos os elementos da lista e ver se são iguais
        Node* l = hashTable->table[key];
        while (1) {
            // Se forem iguais
            if (strcmp(l->word, data) == 0) {
                l->word_count+=1;

                int distance = pos - l->last_ocurrence;
                if (distance<l->smallest_distance) {
                    l->smallest_distance = distance;
                }
                if (distance>l->largest_distance) {
                    l->largest_distance = distance;
                }

                l->average_distance=(double)(pos - l->first_occurence)/l->word_count;
                l->last_ocurrence=pos;
                return;
            }
            // se não forem iguais
            if (l->next != NULL) {
                l = l->next;
            } else { //Se não há nenhuma igual
                l->next = new_Node(data, pos);
                return;
            }
        }

    } else { // se a lista estiver vazia
        hashTable->table[key] = new_Node(data, pos);
        hashTable->used++; //mais um node da hashtable está a ser usado
        if (hashTable->used>0.8*hashTable->size) {
            //se a hashtable estiver muito cheia aumentamos a hashtable de tamanho
            resize_HashTable();
        }
    }
}

//---------------Hash Table com Binary Tree---------------
//Global Variable
struct hash_table_tree *hashTableT;

// Binary Tree
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)
    char word[64];
                int word_count;
    int first_occurence;
    int last_ocurrence;
    int smallest_distance;
    int largest_distance;
    double average_distance;
} tree_node;

tree_node *new_tree_node(char *data, int pos) {
    tree_node *tn = malloc(sizeof(tree_node));
    if(tn == NULL) { //verificar se foi alocada memória
                                fprintf(stderr,"Out of memory\n");
                                exit(0);
    }
    tn->left = NULL;
    tn->right = NULL;
                strcpy(tn->word, data);
                tn->word_count = 1;
    tn->first_occurence = pos;
    tn->last_ocurrence = pos;
    tn->smallest_distance = n_big;
    tn->largest_distance = 0;
    tn->average_distance = 0.0;
    return tn;
}

// Hash Table
typedef struct hash_table_tree {
    struct tree_node **table;
    int used; //numero de nodes da table usados
    int size; //numero total de nodes na table
} hash_table_tree;

hash_table_tree *new_hash_table_tree(int size) { // cria uma nova hashtable com size elementos
    hash_table_tree *htt = malloc(sizeof(hash_table_tree));
    if(htt == NULL) { //verificar se foi alocada memória
                                fprintf(stderr,"Out of memory\n");
                                exit(0);
    }
    htt->table = malloc(size * sizeof(tree_node));
                if(htt->table == NULL) { //verificar se foi alocada memória
                                fprintf(stderr,"Out of memory\n");
                                exit(0);
    }
    htt->used = 0;
    htt->size = size;
                return htt;
}

//função para mover todos os tree_nodes apontados pelos sucessivos left and right
//de tnode inclusive tnode para a nova hashtable recursivamente
void move_recursive(tree_node *tnode, hash_table_tree *newTable) {
    if (tnode->left != NULL) move_recursive(tnode->left, newTable);
    if (tnode->right != NULL) move_recursive(tnode->right, newTable);

    int ind = hash_function(tnode->word, newTable->size); //indice na nova hashtable

    //se nesse indice já houver nodes
    if (newTable->table[ind]!=NULL) {
        tree_node *tnode2 = newTable->table[ind]; //node que vai receber

        while(1) {
            if (strcmp(tnode2->word, tnode->word) < 0) {
                if (tnode2->left != NULL) {
                    tnode2 = tnode2->left;
                } else {
                    tnode->left=NULL; //a node transferida deixa de apontar para outras
                    tnode->right=NULL;
                    tnode2->left = tnode;
                    return;
                }
            } else {
                if (tnode2->right != NULL) {
                    tnode2 = tnode2->right;
                } else {
```

```
                tnode->left=NULL;
                tnode->right=NULL;
                tnode2->right = tnode;
                return;
            }
        }
    }
    //se não houver nodes
    } else {
        tnode->left=NULL;
        tnode->right=NULL;
        newTable->table[ind] = tnode; //a hash_table passa a pontar para esta node
        newTable->used++; //mais um node da hashtable está a ser usado
    }
}

void resize_HashTableT() {
    hash_table_tree *newTable = new_hash_table_tree(1.5*hashTableT->size); //nova hash_table

    for (int i=0; i<hashTableT->size; i++) {
        if (hashTableT->table[i]!=NULL) move_recursive(hashTableT->table[i], newTable);
    }

    free(hashTableT->table);
    free(hashTableT);
    hashTableT = newTable;
}

void insertT(int key, char *data, int pos) { // insere novo valor na hashtable
    // se a tree não estiver vazia
    if (hashTableT->table[key] != NULL) {
        // percorrer a tree
        tree_node* l = hashTableT->table[key];
        while (1) {
            // Se forem iguais
            if (strcmp(l->word, data) == 0) {
                l->word_count+=1;

                int distance = pos - l->last_ocurrence;
                if (distance<l->smallest_distance) {
                    l->smallest_distance = distance;
                }
                if (distance>l->largest_distance) {
                    l->largest_distance = distance;
                }

                l->average_distance=(double)(pos - l->first_occurence)/l->word_count;
                l->last_ocurrence=pos;
                return;
            }
            //Se não forem iguais temos de continuar procurar a palavra ou o local certo a criar novo node
            if (strcmp(l->word, data) < 0) {
                if (l->left != NULL) {
                    l = l->left;
                } else {
                    l->left = new_tree_node(data, pos);
                    return;
                }
            } else {
                if (l->right != NULL) {
                    l = l->right;
                } else {
                    l->right = new_tree_node(data, pos);
                    return;
                }
            }
        }

    } else { // se a tree estiver vazia
        hashTableT->table[key] = new_tree_node(data, pos);
        hashTableT->used++; //mais um node da hashtable está a ser usado
        if (hashTableT->used>0.8*hashTableT->size) {
            resize_HashTableT();
        }
    }
}

// Funções e Estrutura para trabalhar um ficheiro----------
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;

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 = -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;
}
//

//print recursivo para binary tree
void print_recursive(tree_node *n)
{
    if (n->word_count == 1) n->smallest_distance = 0;
    printf("%15s %8d %10d %10d %10d %10d %10.2f\n", n->word, n->word_count, n->first_occurence, n->last_ocurrence, n->smallest_distance, n->largest_distance, n->average_distance);
    if (n->left != NULL) print_recursive(n->left);
    if (n->right != NULL) print_recursive(n->right);
}

//Main
int main(int argc,char **argv)
{
    //verificar se obtivemos o número certo de argumentos
    if ( argc != 2 ) {
                                        printf("%s\n", "Erro, falta passar o ficheiro como argumento");
                                        return -1;
                        }

    file_data_t fd;
                        open_text_file(argv[1], &fd); //abrir ficheiro

    int option;
    printf("1-usar hashtable com linked lists\n");
    printf("2-usar hashtable com binary trees\n");
    scanf("%d", &option);

    //Linked Lists
    if (option == 1) {
         hashTable = new_hash_table(1000); //inicializar hashTable com linked lists

        //armazena as palavras na hashtable
        int ind;
        while(read_word(&fd)!=-1) {
            ind = hash_function(fd.word, hashTable->size);
            insert(ind, fd.word, fd.word_pos);
        }

        for (int i=0; i<hashTable->size; i++) {
            Node *n = hashTable->table[i];
            while (n!=NULL) {
                if (n->word_count == 1) n->smallest_distance = 0;
                printf("%15s %8d %10d %10d %10d %10d %10.2f\n", n->word, n->word_count, n->first_occurence, n->last_ocurrence, n->smallest_distance, n->largest_distance, n->average_distance);
                n=n->next;
            }
        }
        printf("          word    count      first       last smallest d  largest d  average d \n");
    }

    //Binary Trees
    if (option == 2) {
        hashTableT = new_hash_table_tree(1000); //inicializar hashtable com binary trees

        //armazena as palavras na hashtable
        int ind;
        while(read_word(&fd)!=-1) {
            ind = hash_function(fd.word, hashTableT->size);
            insertT(ind, fd.word, fd.word_pos);
        }

        for (int i = 0; i < hashTableT->size; i++){
            if (hashTableT->table[i]!=NULL) print_recursive(hashTableT->table[i]);
        }
        printf("          word    count      first       last smallest d  largest d  average d \n");
    }
}
```