Algoritmos e Estruturas de Dados

Segundo Trabalho Prático

20 de Dezembro de 2019

Trabalho Realizado por:

- Daniel Gomes, Nmec 93015, Contributo:50%
- Rui Fernandes, Nmec 92952, Contributo: 50%

Índice

Introdução

Como objetivo deste segundo trabalho prático pretendia-se criar um programa que fosse capaz de contar o número de ocorrências de cada palavra diferente de um ficheiro de texto. Além disso, teria de apresentar as seguintes funcionalidades:

- Guardar a localização da primeira e última ocorrência de cada palavra distinta no ficheiro de texto;
- Guardar a menor, maior e média distância entre ocorrências consecutivas da mesma palavra.

Para armazenar a informação necessária, foi pedido que se utilizasse, como estrutura de dados, uma hash table implementada com o método de *separate chaining*, ou seja, cada entrada desta tabela deveria apontar para uma lista ligada(*linked list)* ou então para uma árvore binária ordenada,onde cada posição da respectiva sub estrutura iria conter toda a informação para uma dada palavra.

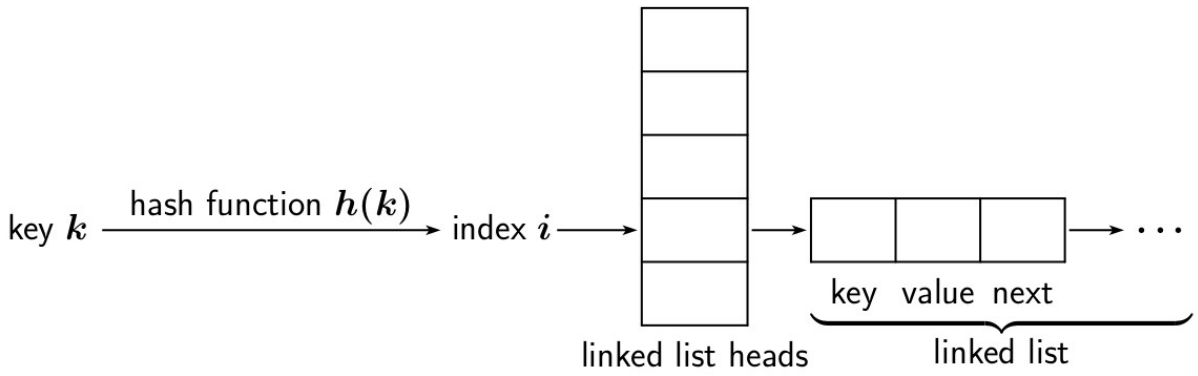
Implementação do Problema

Por opção, decidiu-se, além da implementação de *Separate Chaining* com Listas Ligadas, uma implementação com Árvores Binárias. A primeira, encontra-se no programa **TP2_LL.c** enquanto que a segunda no programa **TP2_BT.c**. Pretendemos, com estas duas implementações, encontrar as respetivas vantagens e desvantagens de cada uma. Além disto, em ambos programas, era pretendido que a Hash Table fosse redimensionada de forma dinâmica.

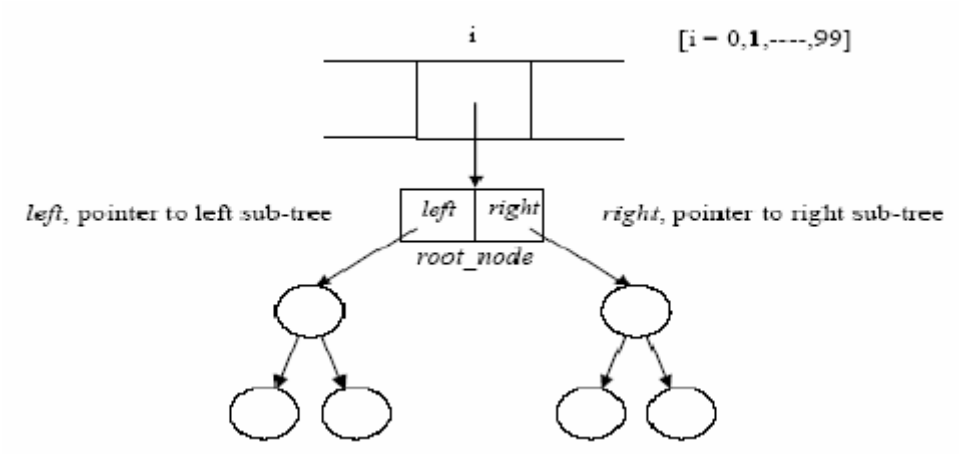
Porquê Hash Tables?

Hash Tables, ou Tabelas de Dispersão, são uma estrutura de dados que associa chaves de pesquisa a valores. Estas chaves são um array de caracteres no nosso caso (a palavra a guardar) e ao serem passadas por uma Função de Dispersão é gerado o índice correspondente no array representativo da tabela, onde se vai guardar a informação pretendida. Assim através da hashtable e da função de dispersão, a pesquisa de informação torna-se muito mais rápida. Contudo, como cada índice gerado pela função de dispersão usada pode ser o mesmo para algumas palavras distintas (devido ao código ASCII associado a cada caractere) dão-se as chamadas colisões, ou seja neste caso, duas keys que apontam para a mesma posição da tabela. Assim, o uso de *Separate Chaining*, torna-se fundamental para que possamos lidar com estes casos, podendo ter várias chaves a apontar para o mesmo *index* sem perder informação.

Nas imagens,que se seguem encontram-se esquemas visuais de como se encontram os dados das formas que até agora temos abordado



Separate Chaining com Linked Lists



Separate Chaining com Binary Trees

Resultados obtidos

TP2_LL.c

Para podermos concluir se o nosso programa estaria de acordo com o expectável, decidiu-se criar um “menu”, idêntico ao que foi utilizado na Aula Prática 08 no programa **countwords.c**, onde o utilizador pode escolher se pretende:

- visualizar o número de total de palavras;
- visualizar o número de palavras distintas;
- listar palavras distintas e o número respetivo de ocorrências;
- listar palavras distintas e as respetivas posições;

- listar palavras distintas e as respetivas distâncias;
- listar palavras distintas e informação completa acerca de cada uma.

Com vista à utilização correta deste programa, sempre que é executado introduzindo uma opção ou um ficheiro de texto inválidos, ou sem se introduzir algum dos dois, é impressa a seguinte mensagem.

```
danielgomes@danielgomes-Lenovo-ideapad-530S-14IKB:~/Desktop/Projeto2AED$ ./TP2_LL
usage: ./TP2_LL -a text_file  # count how many words are in the file
       ./TP2_LL -u text_file  # count how many unique words are in the file
       ./TP2_LL -c text_file  # list words and their count
       ./TP2_LL -p text_file  # list words and their positions
       ./TP2_LL -d text_file  # list words and their distances
       ./TP2_LL -i text_file  # list words and all info
danielgomes@danielgomes-Lenovo-ideapad-530S-14IKB:~/Desktop/Projeto2AED$
```

Menu Utilizado em **TP2_LL.c**

Quanto às primeiras duas opções obtiveram-se os seguintes resultados,onde iremos comparar com o programa desenvolvido pelo Professor na Aula Prática já referida.

```
danielgomes@danielgomes-Lenovo-ideapad-530S-14IKB:~/Desktop/Projeto2AED$ ./TP2_LL -a SherlockHolmes.txt
The file SherlockHolmes.txt contains 657439 words
danielgomes@danielgomes-Lenovo-ideapad-530S-14IKB:~/Desktop/Projeto2AED$ ./TP2_LL -u SherlockHolmes.txt
The file SherlockHolmes.txt contains 45882 unique words
danielgomes@danielgomes-Lenovo-ideapad-530S-14IKB:~/Desktop/Projeto2AED$
```

Resultados Obtidos em **TP2_LL.c**

Assim, utilizando o mesmo ficheiro de texto conseguimos concluir que as duas primeiras opções têm um bom funcionamento, visto que os resultados obtidos são praticamente iguais aos do programa desenvolvido pelo professor (Aula P08):

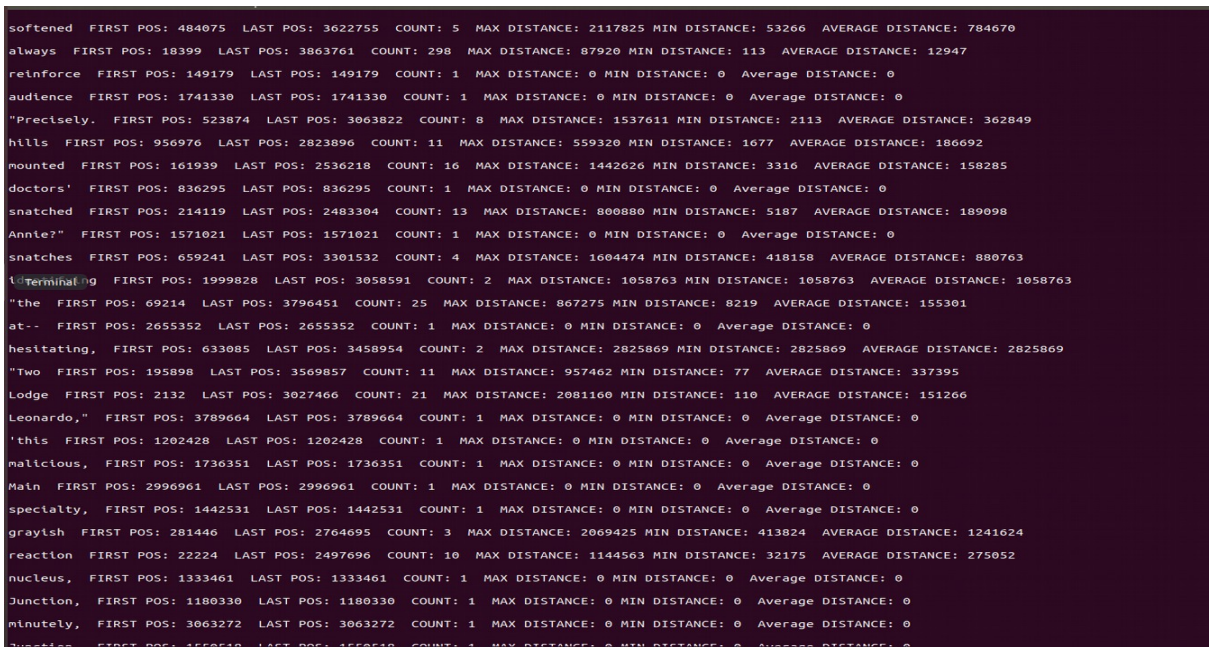
```
danielgomes@danielgomes-Lenovo-ideapad-530S-14IKB:~/Desktop/AED/P08$ ./count_words -d SherlockHolmes.txt
The file SherlockHolmes.txt contains 45882 distinct words
```

```
danielgomes@danielgomes-Lenovo-ideapad-530S-14IKB:~/Desktop/AED/P08$ ./count_words -a
usage: ./count_words -a text_file ...  # count the number of words
       ./count_words -d text_file ...  # count the number of different words
       ./count_words -l text_file ...  # list all words
danielgomes@danielgomes-Lenovo-ideapad-530S-14IKB:~/Desktop/AED/P08$ ./count_words -a SherlockHolmes.txt
The file SherlockHolmes.txt contains 657438 words
```

Resultados obtidos na Aula Prática

Ocorreu de facto uma diferença no número total de palavras de 1, o que se torna relativamente insignificante, e é provavelmente de fácil correção, apesar de não termos encontrado o motivo da diferença.

Passando para a opção mais importante, ***-i***, devido ao facto de conter toda a informação sobre as palavras do ficheiro de texto, tal como requerido no enunciado do projeto.
Foi utilizado o ficheiro de texto ***SherlockHolmes.txt*** (ficheiro várias vezes utilizado para realização de testes ao longo trabalho), e obteve-se os seguintes resultados:



Excerto do resultado obtido da opção -i do programa **TP2_LL.c**

Neste excerto de informação de algumas palavras do ficheiro mostra-se em cada linha a informação associada a uma dada palavra, tal que:

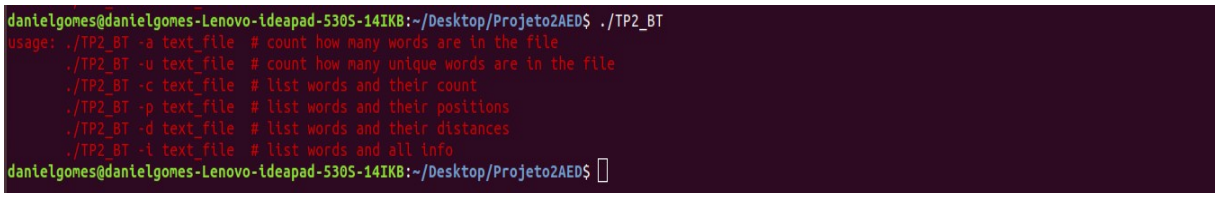
- Primeira coluna corresponde à palavra;
- Segunda e terceira coluna correspondem à primeira e última posição onde ocorreu a palavra no ficheiro (em número caracteres face ao início do ficheiro);
- Quarta coluna contém o número de aparições da palavra;
- As restantes colunas dizem respeito à máxima,mínima e média distâncias (em número caracteres).

Todas as palavras cujo número de ocorrências não fosse superior a uma, apresentaram distâncias máximas mínimas e médias de 0, enquanto que a distância média mostrou ser sempre um valor entre a distância mínima e máxima pelo que podemos aceitar os valores obtidos.

TP2_BT.c

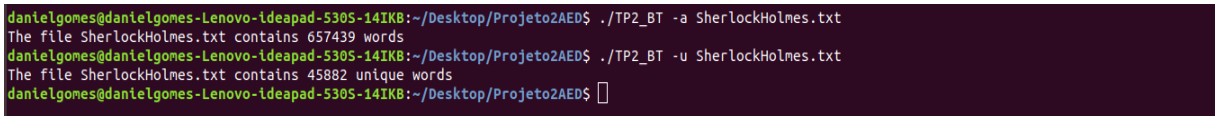
A estrutura deste programa é a mesma do que foi referido no ponto anterior,ou seja, apresenta-se, da mesma forma, um menu com as mesmas 5 opções:

- visualizar o número de total de palavras;
- visualizar o número de palavras distintas;
- listar palavras e o número respetivo de ocorrências;
- listar palavras distintas e as respetivas posições;
- listar palavras distintas e as respetivas distâncias;
- listar palavras distintas e informação completa acerca de cada uma.



Menu de Utilização do programa **TP2_BT.c**

Comparando da mesma forma as primeira duas opções da implementação desenvolvida- contagem de palavras totais do ficheiro e a contagem de palavras únicas deste- às duas primeiras do ficheiro **countwords.c** já desenvolvido pelo Professor na Aula Prática, obtiveram-se os seguintes resultados:



Opção -a e -u do programa **TP2_BT.c**

Tendo em conta que os valores para estas opções por parte do ficheiro **countwords.c** já foram enunciados, podemos concluir que mais uma vez o número de palavras analisadas foi um sucesso pois detetou-se 45882 palavras diferentes, ou seja, o mesmo valor, e 657439 palavras totais no ficheiro (diferença ligeira de uma palavra para o que foi determinado em **countwords.c**).

Da mesma forma que foi efetuado para TP2_LL.c, iremos agora apresentar um excerto da execução, (visto que é extremamente extensa para ser apresentada por completo) da mesma opção, -i, para este programa,TP2_BT.c:

```
Brunton,  FIRST POS: 1322491  LAST POS: 1343616  COUNT: 2  MAX DISTANCE: 21125 MIN DISTANCE: 21125  AVERAGE DISTANCE: 21125
rolls  FIRST POS: 1624452  LAST POS: 3624871  COUNT: 4  MAX DISTANCE: 849377 MIN DISTANCE: 432199  AVERAGE DISTANCE: 666806
necktie.  FIRST POS: 775478  LAST POS: 1814524  COUNT: 3  MAX DISTANCE: 666142 MIN DISTANCE: 372904  AVERAGE DISTANCE: 519523
Brunton.  FIRST POS: 1320021  LAST POS: 1346394  COUNT: 4  MAX DISTANCE: 16894 MIN DISTANCE: 2912  AVERAGE DISTANCE: 8791
gutter,  FIRST POS: 1238168  LAST POS: 3408230  COUNT: 2  MAX DISTANCE: 2170062 MIN DISTANCE: 2170062  AVERAGE DISTANCE: 2170062
K.';  FIRST POS: 735017  LAST POS: 735017  COUNT: 1  MAX DISTANCE: 0 MIN DISTANCE: 0  Average DISTANCE: 0
Congratulate  FIRST POS: 2475771  LAST POS: 2475771  COUNT: 1  MAX DISTANCE: 0 MIN DISTANCE: 0  Average DISTANCE: 0
forgery  FIRST POS: 36613  LAST POS: 2045628  COUNT: 3  MAX DISTANCE: 1253432 MIN DISTANCE: 755583  AVERAGE DISTANCE: 1004507
done."  FIRST POS: 61251  LAST POS: 3846556  COUNT: 16  MAX DISTANCE: 860443 MIN DISTANCE: 26457  AVERAGE DISTANCE: 252353
'teens  FIRST POS: 2398855  LAST POS: 2398855  COUNT: 1  MAX DISTANCE: 0 MIN DISTANCE: 0  Average DISTANCE: 0
multitudinous  FIRST POS: 3768140  LAST POS: 3768140  COUNT: 1  MAX DISTANCE: 0 MIN DISTANCE: 0  Average DISTANCE: 0
organized  FIRST POS: 259577  LAST POS: 2932833  COUNT: 4  MAX DISTANCE: 1340763 MIN DISTANCE: 76190  AVERAGE DISTANCE: 891085
sedentary  FIRST POS: 818848  LAST POS: 1824913  COUNT: 2  MAX DISTANCE: 1006065 MIN DISTANCE: 1006065  AVERAGE DISTANCE: 1006065
vacancy,  FIRST POS: 576613  LAST POS: 3600323  COUNT: 3  MAX DISTANCE: 2344118 MIN DISTANCE: 679592  AVERAGE DISTANCE: 1511855
cove.  FIRST POS: 77077  LAST POS: 77077  COUNT: 1  MAX DISTANCE: 0 MIN DISTANCE: 0  Average DISTANCE: 0
done.'  FIRST POS: 474980  LAST POS: 3416076  COUNT: 2  MAX DISTANCE: 2941096 MIN DISTANCE: 2941096  AVERAGE DISTANCE: 2941096
Put  FIRST POS: 141084  LAST POS: 3859784  COUNT: 12  MAX DISTANCE: 1413315 MIN DISTANCE: 15626  AVERAGE DISTANCE: 338063
cracked,  FIRST POS: 348201  LAST POS: 1831183  COUNT: 3  MAX DISTANCE: 1013827 MIN DISTANCE: 469155  AVERAGE DISTANCE: 741491
were,"  FIRST POS: 833416  LAST POS: 3001359  COUNT: 3  MAX DISTANCE: 1628858 MIN DISTANCE: 539085  AVERAGE DISTANCE: 1083971
mankind,  FIRST POS: 1307805  LAST POS: 1307805  COUNT: 1  MAX DISTANCE: 0 MIN DISTANCE: 0  Average DISTANCE: 0
cracked.  FIRST POS: 1569717  LAST POS: 1569717  COUNT: 1  MAX DISTANCE: 0 MIN DISTANCE: 0  Average DISTANCE: 0
threads,  FIRST POS: 2387709  LAST POS: 3122544  COUNT: 2  MAX DISTANCE: 734835 MIN DISTANCE: 734835  AVERAGE DISTANCE: 734835
organizer  FIRST POS: 1599710  LAST POS: 2643520  COUNT: 2  MAX DISTANCE: 1043810 MIN DISTANCE: 1043810  AVERAGE DISTANCE: 1043810
afraid,  FIRST POS: 78374  LAST POS: 2299008  COUNT: 9  MAX DISTANCE: 608662 MIN DISTANCE: 3805  AVERAGE DISTANCE: 277579
saucer  FIRST POS: 124136  LAST POS: 3438866  COUNT: 9  MAX DISTANCE: 2511975 MIN DISTANCE: 197  AVERAGE DISTANCE: 414341
comers  FIRST POS: 151257  LAST POS: 151257  COUNT: 1  MAX DISTANCE: 0 MIN DISTANCE: 0  Average DISTANCE: 0
afraid.  FIRST POS: 3540936  LAST POS: 3806917  COUNT: 2  MAX DISTANCE: 265981 MIN DISTANCE: 265981  AVERAGE DISTANCE: 265981
cursing  FIRST POS: 439878  LAST POS: 439878  COUNT: 1  MAX DISTANCE: 0 MIN DISTANCE: 0  Average DISTANCE: 0
```

Excerto da execução da opção -i do programa TP2_BT.c

Tal como anteriormente, o resultado apresentado contém a mesma estrutura, ou seja, em cada linha do terminal está presente uma palavra diferente, seguida da primeira e última posição onde ocorreram no ficheiro de texto, assim como o número de aparições e distâncias máximas,mínimas e médias.

Também importante referir que, mais uma vez, as distâncias aparentam estar bem determinadas tendo em conta que se a palavra não passar de uma aparição singular, não apresentará qualquer valor paras as distâncias, mantendo-se em zero, além de que, em caso contrário, o valor médio de distância nunca encontra-se superior ou inferior à distâncias máxima e mínima, respetivamente.

Testes Efetuados

Como forma de teste de eficiência para ambos os programas decidiu-se determinar o tempo de execução entre ambos ficheiros e em diversas condições (o tamanho inicial da tabela é de 8192 para todos estes testes). Assim, inicialmente comparou-se os programas nas condições já pré definidas, onde se obteve os seguintes tempos de execução:

- **TP2_LL.c** demorou 0.089 segundos, utilizando a função resize() sempre que **80 por cento** da tabela estivesse ocupada;
- **TP2_BT.c** demorou 0.063 segundos,utilizando a função de resize() sempre que **80 por cento** da tabela estivesse ocupada.

Tempo efetuado em cada Programa

Assim, revela-se que a execução utilizando separate chaining com árvores binárias torna-se ligeiramente melhor em termos de eficiência do que utilizando listas ligadas.

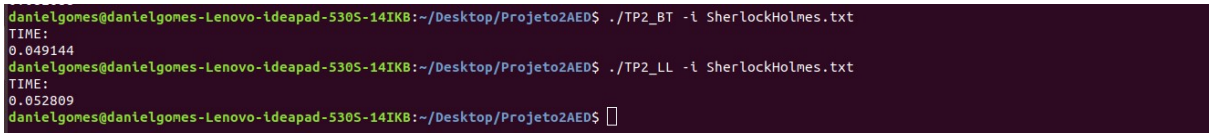
Alterando em cada um destes programas, a percentagem de preenchimento de tabela necessária para que fosse chamada a função de redimensionamento para **50 por cento** conseguiu-se obter os seguintes resultados:

- **TP2_LL.c** demorou 0.103 segundos
- **TP2_BT.c** demorou 0.098 segundos.

Os valores são expectáveis tendo em conta que, ao diminuir a percentagem referida anteriormente, a função resize() é chamada mais vezes, e, por sua vez, aumenta o tempo necessário para correr o código.

Seguidamente, efetuou-se a realização de testes às duas implementações mas **sem efetuar qualquer redimensionamento à tabela de dispersão**, pelo que o *runtime* de cada uma foi o seguinte:

- **TP2_LL.c** demorou 0.052 segundos
- **TP2_BT.c** demorou 0.049 segundos.



Concluindo, sem ter de efetuar redimensionamento, que envolve copiar toda a informação de uma tabela para uma nova com maior tamanho, melhora-se o *runtime*. Contudo, não será de estranhar algumas vezes demorar um tempo semelhante àquele que apresentavam ambos programas nas condições iniciais visto que a tabela pode gerar mais colisões, o que por consequência envolve ter de percorrer a lista ligada, à qual aponta cada índice da tabela, até ao fim, para introduzir novas palavras.

Conclusão

Este trabalho prático permitiu consolidar conhecimentos acerca de vários assuntos lecionados ao longo da disciplina de Algoritmos e Estruturas de Dados, nomeadamente *Hash Tables, Linked Lists e Ordered Binary Trees*. Além disso, este trabalho permitiu nos utilizar vários conceitos que estão por base na Linguagem de Programação usada ,***C*** , como alocação de memória. Aliado a tudo isto, concluiu-se que a utilização por listas ligadas torna-se mais lenta, mesmo que por pouco, quando comparada com árvores binárias além de que a utilização de Separate Chaining permite de forma relativamente fácil estruturar os nossos dados,evitando colisões.

Por fim, como base deste trabalho, é importante referir que de uma maneira eficiente e de fácil implementação, aprendemos uma nova forma de obter informação variada de um ficheiro de texto.

Código Utilizado

TP2_LL.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>
#define initial_size 8192

//LINKED LISTS IMPLEMENTATION

int unique_word_count = 0;
int total_words = 0;

typedef struct data_cell
{
    struct data_cell *next; //aponta pra proxima palavra
    char word[64];  //palavra associada
    int firstPos;
    int prevWordPos;  // index da ultima vez que esta palavra ocorreu (no ficheiro)
    int totaldistance;
    int mindistance; //distancia minima entre ocorrencias
    int maxdistance; //distancia maxima entre ocorrencias
    int counter; // contar o numero de ocorrencias
} data_cell;

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
    int is_end;
}
file_data_t;


int open_text_file(char *file_name,file_data_t *fd)
{
    fd->fp = fopen(file_name,"rb");
    if(fd->fp == NULL){
        fprintf(stderr,"unable to open file %s\n",file_name);
        return -1;
    }
    fd->word_pos = -1;
    fd->word_num = -1;
    fd->word[0] ='\0';
    fd->current_pos = -1;
    fd->is_end = 0;
```

```
    fd->is_end = 0;
    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){
            fd->is_end = 1;
            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){
            fd->is_end = 1;
            break; // end of file
        }
        fd->current_pos++;
        if(c <= 32)
            break; // terminate word
        fd->word[i] = (char)c;
    }
    fd->word[i] ='\0';
    return 0;
}


// função de hash....
unsigned int hash_function(const char *str,unsigned int s)
{
    unsigned int h;
    for(h = 0u;*str != '\0' ;str++)
```

```
// função de hash....
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
}
//...inicializar cada posiçao da linked list
static data_cell *initNewCell(int pos,char *word){
    data_cell *newcell = (data_cell *)malloc(sizeof(data_cell));
    if(newcell == NULL){
        fprintf(stderr,"Out of
        exit(1);
    }
    strcpy(newcell->word,word);
    newcell->firstPos=pos;
    newcell->prevWordPos=pos;
    newcell->counter=1;
    newcell->mindistance=INT_MAX;
    newcell->maxdistance=0;
    newcell->totaldistance=0;
    newcell->next=NULL;
    return newcell;
}

//procurar informação
data_cell *find_data(const char *key,data_cell*hashtable[],int hash_size){
    unsigned int idx; //index-> hash code posteriormente calculado
    data_cell *dc;//head da linked list
    idx = hash_function(key,hash_size);
    dc=hashtable[idx];
    while(dc!=NULL && strcmp(key,dc->word)!=0){
        dc=dc->next;
    }
    return dc;
```

Definition:
TP2_LL.c:21

```
    return dc;
}
//
void updateCell(data_cell * cell,int pos){

    int tempdist=pos-cell->prevWordPos;
    cell->counter+=1;
    cell->totaldistance+=tempdist;
    if(tempdist>cell->maxdistance)
        cell->maxdistance=tempdist;
    if(tempdist<cell->mindistance)
        cell->mindistance=tempdist;
    cell->prevWordPos=pos;
}

data_cell** resizeTable(int old_size, data_cell**hashtable, int new_size){
    //initialize new Hash Table
    data_cell** new_hashtable = calloc(new_size, sizeof(data_cell));
    //fill new Hash Table with old values
    int hashcode=0;
    for(int i=0;i<old_size;i++){
        for(data_cell* h = hashtable[i]; h!=NULL; h=h->next){
            hashcode=hash_function(h->word,new_size);

            data_cell *head = new_hashtable[hashcode], *previous = NULL;
            data_cell *tmp_h = (data_cell *)malloc(sizeof(data_cell));

            tmp_h->next = NULL;
            strcpy(tmp_h->word, h->word);
            tmp_h->firstPos = h->firstPos;
            tmp_h->prevWordPos = h->prevWordPos;
            tmp_h->totaldistance = h->totaldistance;
            tmp_h->mindistance = h->mindistance;
            tmp_h->maxdistance = h->maxdistance;
            tmp_h->counter = h->counter;

            while (head!=NULL) {
                previous = head;
                head = head->next;
            }
            if (previous!=NULL){
                previous->next = tmp_h;
            }else{
```

```
            }else{
                new_hashtable[hashcode] = tmp_h;
            }
        }
    }
    free(hashtable);
    return new_hashtable;
}


void printCount(data_cell * cell){

    printf("%s  ", cell->word);
    printf("COUNT: %d  \n\n", cell->counter);
    if(cell->next!=NULL){
        printCount(cell->next);
    }
}

void printPositions(data_cell * cell){

    printf("%s  ", cell->word);
    printf("FIRST POS: %d   ", cell->firstPos);
    printf("LAST POS: %d\n\n", cell->prevWordPos);
    if(cell->next!=NULL){
        printPositions(cell->next);
    }
}

void printDistances(data_cell * cell){

    printf("%s  ", cell->word);
    printf("MAX DISTANCE: %d  ", cell->maxdistance);
    if(cell->mindistance==INT_MAX){
        printf("MIN DISTANCE: 0   ");
        printf("Average DISTANCE: 0\n\n");
    }else{
        printf("MIN DISTANCE: %d   ", cell->mindistance);
        printf("AVERAGE DISTANCE: %d\n\n", (int)( (cell->totaldistance)/(cell->counter-1) ));
    }
    if(cell->next!=NULL){
        printDistances(cell->next);
    }
}
```

```
}

void printInfo(data_cell * cell){

    printf("%s  ", cell->word);
    printf("FIRST POS: %d  ", cell->firstPos);
    printf("LAST POS: %d  ", cell->prevWordPos);
    printf("COUNT: %d  ", cell->counter);
    printf("MAX DISTANCE: %d ", cell->maxdistance);
    if(cell->mindistance==INT_MAX){
        printf("MIN DISTANCE: 0  ");
        printf("Average DISTANCE: 0\n\n");
    }else{
        printf("MIN DISTANCE: %d  ", cell->mindistance);
        printf("AVERAGE DISTANCE: %d\n\n", (int)( (cell->totaldistance)/(cell->counter-1) ));
    }
    if(cell->next!=NULL){
        printInfo(cell->next);
    }
}

int main(int argc, char * argv[])
{

    //OPTIONS
    int opt = -1;
      if(argc >= 2 && strcmp(argv[1],"-a") == 0) opt = 'a'; // all words
      if(argc >= 2 && strcmp(argv[1],"-u") == 0) opt = 'u'; // unique words
      if(argc >= 2 && strcmp(argv[1],"-c") == 0) opt = 'c'; // list words
      if(argc >= 2 && strcmp(argv[1],"-p") == 0) opt = 'p';
      if(argc >= 2 && strcmp(argv[1],"-d") == 0) opt = 'd';
      if(argc >= 2 && strcmp(argv[1],"-i") == 0) opt = 'i';
      if(opt < 0)
      {
        fprintf(stderr,"\e[5;31m"); // blink on (may not work in some text terminals), text in red
        fprintf(stderr,"usage: %s -a text_file  # count how many words are in the file\n",argv[0]);
        fprintf(stderr,"       %s -u text_file  # count how many unique words are in the file\n",argv[0]);
        fprintf(stderr,"       %s -c text_file  # list words and their count\n",argv[0]);
        fprintf(stderr,"       %s -p text_file  # list words and their positions\n",argv[0]);
        fprintf(stderr,"       %s -d text_file  # list words and their distances\n",argv[0]);
        fprintf(stderr,"       %s -i text_file  # list words and all info\n",argv[0]);
        fprintf(stderr,"\e[0m"); // normal output
        return 1;
```

```
        return 1;
      }

    //MAIN PROGRAM
    data_cell** hashtable = calloc(initial_size, sizeof(data_cell));
    int hash_size = initial_size;

    // read all words and put them on the hash table
    data_cell *tmp=NULL;
    int hashcode=0;
    file_data_t *fd = (file_data_t *)malloc(sizeof(file_data_t));

    if(open_text_file(argv[2],fd)==-1){
        exit(1);
    }

    while(fd->is_end==0){
        if (unique_word_count>((int)hash_size*0.8))
        {
            hashtable = resizeTable(hash_size,hashtable,(int)(hash_size*1.5));
            hash_size = (int)(hash_size*1.5);
        }
        read_word(fd);
        total_words++;
        tmp = find_data(fd->word,hashtable,hash_size);
        if(tmp==NULL)
        {
            unique_word_count++;
            hashcode=hash_function(fd->word,hash_size);

            data_cell *head = hashtable[hashcode], *previous = NULL;
            while (head!=NULL) {
                previous = head;
                head = head->next;
            }

            data_cell *nd = initNewCell(fd->word_pos,fd->word);

            if (previous!=NULL) previous->next = nd;
            else hashtable[hashcode] = nd;

        }
        else updateCell(tmp,fd->word_pos);
```

```
      }
      else updateCell(tmp,fd->word_pos);
  }
  clock_t end= clock();
  double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
  //OPTION TREATMENT

    switch(opt)
    {
    case 'a': printf("The file %s contains %d words\n",argv[2],total_words);; break;
    case 'u': printf("The file %s contains %d unique words\n",argv[2],unique_word_count); break;
    case 'c': printf("Contents of the file %s:\n",argv[2]); for(int i=0;i<hash_size;i++) if(hashtable[i]!=NULL) printCount(hashtable[i]); break;
    case 'p': printf("Contents of the file %s:\n",argv[2]); for(int i=0;i<hash_size;i++) if(hashtable[i]!=NULL) printPositions(hashtable[i]); break;
    case 'd': printf("Contents of the file %s:\n",argv[2]); for(int i=0;i<hash_size;i++) if(hashtable[i]!=NULL) printDistances(hashtable[i]); break;
    case 'i': printf("Contents of the file %s:\n",argv[2]); for(int i=0;i<hash_size;i++) if(hashtable[i]!=NULL) printInfo(hashtable[i]); break;
    }

 // printf("%s\n", "TIME:");
  //printf("%f\n",time_spent );
  close_text_file(fd);
  free(fd);

}
```

TP2_BT.c

```c
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){
            fd->is_end = 1;
            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){
            fd->is_end = 1;
            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 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
}

static node *initNewNode(int pos,char *word){
    node *newnode = (node *)malloc(sizeof(node));
    if(newnode == NULL){
        fprintf(stderr,"Out of memory\n");
        exit(1);
    }
```

```
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    strcpy(newnode->word,word);
    newnode->firstPos=pos;
    newnode->prevWordPos=pos;
    newnode->counter=1;
    newnode->mindistance=INT_MAX;
    newnode->maxdistance=0;
    newnode->totaldistance=0;
    newnode->left=NULL;
    newnode->right=NULL;
    return newnode;
}
void updateNode(node * n,int pos){

    int tempdist=pos-n->prevWordPos;
    n->counter+=1;
    n->totaldistance+=tempdist;
    if(tempdist>n->maxdistance)
        n->maxdistance=tempdist;
    if(tempdist<n->mindistance)
        n->mindistance=tempdist;
    n->prevWordPos=pos;
}

 node** add_word(char *key ,node**hashtable, int hash_size, int pos){
        unsigned int idx;
        node *root;//root da binary tree
        idx = hash_function(key,hash_size);
        root=hashtable[idx];

        if(root == NULL){
            hashtable[idx] = initNewNode(pos,key);
            unique_word_count++;
            return hashtable;
        }
         while(1){
            int c = strcmp(key,root->word);
            if     (c <  0) { if(root->left  == NULL) { root->left  = initNewNode(pos,key); unique_word_count++; return hashtable; } else root = root->left;  }
            else if(c == 0) { updateNode(root,pos); return hashtable;}
            else            { if(root->right == NULL) { root->right = initNewNode(pos,key); unique_word_count++; return hashtable; } else root = root->right; }
        }
 }

 node** add_resize(node* old_root, node**hashtable, int hash_size){
    unsigned int idx;
    idx=hash_function(old_root->word,hash_size);

    node *tmp = (node *)malloc(sizeof(node));
    tmp->left = NULL;
    tmp->right = NULL;
    strcpy(tmp->word, old_root->word);
```

```
  strcpy(tmp->word, old_root->word);
  tmp->firstPos = old_root->firstPos;
  tmp->prevWordPos = old_root->prevWordPos;
  tmp->totaldistance = old_root->totaldistance;
  tmp->mindistance = old_root->mindistance;
  tmp->maxdistance = old_root->maxdistance;
  tmp->counter = old_root->counter;

  node *new_root = hashtable[idx];

  if(new_root == NULL){
          hashtable[idx] = tmp;
  }else{
      while(1){
          int c = strcmp(old_root->word,new_root->word);
          if     (c <  0) { if(new_root->left  == NULL) { new_root->left  = tmp; break; } else new_root = new_root->left;  }
          else if(c == 0) { break;}
          else            { if(new_root->right == NULL) { new_root->right = tmp; break; } else new_root = new_root->right; }
      }
  }

  if(old_root->left!=NULL){
      hashtable = add_resize(old_root->left, hashtable, hash_size);
  }
  if(old_root->right!=NULL){
      hashtable = add_resize(old_root->right, hashtable, hash_size);
  }
  return hashtable;


ode** resizeTable(int old_size, node**hashtable, int new_size){
  //initialize new Hash Table
  node** new_hashtable = calloc(new_size, sizeof(node));
  //fill new Hash Table with old values
  int hashcode=0;
  for(int i=0;i<old_size;i++){
      if (hashtable[i]!=NULL){
          new_hashtable = add_resize(hashtable[i], new_hashtable, new_size);
      }
  }
  free(hashtable);
  return new_hashtable;


id printCount(node * node){

  printf("%s  ", node->word);
  printf("COUNT: %d  \n\n", node->counter);
  if(node->left!=NULL){
      printCount(node->left);
  }
  if(node->right!=NULL){
```

```
    if(node->right!=NULL){
        printCount(node->right);
    }
}

void printPositions(node * node){

    printf("%s  ", node->word);
    printf("FIRST POS: %d   ", node->firstPos);
    printf("LAST POS: %d\n\n", node->prevWordPos);
    if(node->left!=NULL){
        printPositions(node->left);
    }
    if(node->right!=NULL){
        printPositions(node->right);
    }
}

void printDistances(node * node){

    printf("%s  ", node->word);
    printf("MAX DISTANCE: %d  ", node->maxdistance);
    if(node->mindistance==INT_MAX){
        printf("MIN DISTANCE: 0   ");
        printf("Average DISTANCE: 0\n\n");
    }else{
        printf("MIN DISTANCE: %d   ", node->mindistance);
        printf("AVERAGE DISTANCE: %d\n\n", (int)( (node->totaldistance)/(node->counter-1) ));
    }
    if(node->left!=NULL){
        printDistances(node->left);
    }
    if(node->right!=NULL){
        printDistances(node->right);
    }
}

 void printInfo(node * node){

    printf("%s  ", node->word);
    printf("FIRST POS: %d  ", node->firstPos);
    printf("LAST POS: %d  ", node->prevWordPos);
    printf("COUNT: %d  ", node->counter);
    printf("MAX DISTANCE: %d ", node->maxdistance);
    if(node->mindistance==INT_MAX){
        printf("MIN DISTANCE: 0  ");
        printf("Average DISTANCE: 0\n\n");
    }else{
        printf("MIN DISTANCE: %d  ", node->mindistance);
        printf("AVERAGE DISTANCE: %d\n\n", (int)( (node->totaldistance)/(node->counter-1) ));
    }
    if(node->left!=NULL){
```

```c
    if(node->left!=NULL){
        printInfo(node->left);
    }
    if(node->right!=NULL){
        printInfo(node->right);
    }
}


int main(int argc,char *argv[]){

    //OPTIONS
    int opt = -1;
      if(argc >= 2 && strcmp(argv[1],"-a") == 0) opt = 'a';
      if(argc >= 2 && strcmp(argv[1],"-u") == 0) opt = 'u';
      if(argc >= 2 && strcmp(argv[1],"-c") == 0) opt = 'c';
      if(argc >= 2 && strcmp(argv[1],"-p") == 0) opt = 'p';
      if(argc >= 2 && strcmp(argv[1],"-d") == 0) opt = 'd';
      if(argc >= 2 && strcmp(argv[1],"-i") == 0) opt = 'i';
      if(opt < 0)
      {
        fprintf(stderr,"\e[5;31m"); // blink on (may not work in some text terminals), text in red
        fprintf(stderr,"usage: %s -a text_file  # count how many words are in the file\n",argv[0]);
        fprintf(stderr,"       %s -u text_file  # count how many unique words are in the file\n",argv[0]);
        fprintf(stderr,"       %s -c text_file  # list words and their count\n",argv[0]);
        fprintf(stderr,"       %s -p text_file  # list words and their positions\n",argv[0]);
        fprintf(stderr,"       %s -d text_file  # list words and their distances\n",argv[0]);
        fprintf(stderr,"       %s -i text_file  # list words and all info\n",argv[0]);
        fprintf(stderr,"\e[0m"); // normal output
        return 1;
      }

    //MAIN PROGRAM
    clock_t begin= clock();
    node ** hashtable=calloc(initial_size,sizeof(node));
    int hash_size = initial_size;

    file_data_t *fd = (file_data_t *)malloc(sizeof(file_data_t));

    if(open_text_file(argv[2],fd)==-1){
        exit(1);
    }

    while(fd->is_end == 0){
        read_word(fd);
        total_words++;
        if(unique_word_count>((int)hash_size*0.8))
        {
            hashtable = resizeTable(hash_size,hashtable,(int)hash_size*1.5);
            hash_size = (int)hash_size*1.5;
        }
        hashtable=add_word(fd->word,hashtable,hash_size,fd->word_pos);
```

```c
    while(fd->is_end == 0){
        read_word(fd);
        total_words++;
        if(unique_word_count>((int)hash_size*0.8))
        {
            hashtable = resizeTable(hash_size,hashtable,(int)hash_size*1.5);
            hash_size = (int)hash_size*1.5;
        }
        hashtable=add_word(fd->word,hashtable,hash_size,fd->word_pos);
    }
    clock_t end= clock();
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    //OPTION TREATMENT

      switch(opt)
      {
      case 'a': printf("The file %s contains %d words\n",argv[2],total_words); break;
      case 'u': printf("The file %s contains %d unique words\n",argv[2],unique_word_count); break;
      case 'c': printf("Contents of the file %s:\n",argv[2]); for(int i=0;i<hash_size;i++) if(hashtable[i]!=NULL) printCount(hashtable[i]); break;
      case 'p': printf("Contents of the file %s:\n",argv[2]); for(int i=0;i<hash_size;i++) if(hashtable[i]!=NULL) printPositions(hashtable[i]); break
      case 'd': printf("Contents of the file %s:\n",argv[2]); for(int i=0;i<hash_size;i++) if(hashtable[i]!=NULL) printDistances(hashtable[i]); break
      case 'i': printf("Contents of the file %s:\n",argv[2]); for(int i=0;i<hash_size;i++) if(hashtable[i]!=NULL) printInfo(hashtable[i]); break;
      }
    close_text_file(fd);
    free(fd);

    //printf("%s\n", "TIME:");
    //printf("%f\n",time_spent );
}
```