ALGORITMOS E ESTRUTURAS DE DADOS

Second written report

Licenciatura em Engenharia Informática

Pedro Miguel Bastos Almeida – 93150 – 33,3%

Eduardo Henrique Ferreira Santos – 93107 – 33,3%

José Vaz – 88903 – 33,3%

Índice

Introdução

Com este projeto nós pretendemos aprofundar os conhecimentos na linguagem C, aperfeiçoando também a prática em programar utilizando novos métodos necessários para a resolução deste problema.

Ajudará também a um melhor entendimento das hash tables e linked lists, assim como o método separate chaining para a sua implementação.

Hash Tables

A hash table é uma estrutura de dados que suporta a sua criação e destruição, inserção de um par key-value, a procura e a remoção de dados. A informação é procurada através da chave (key). Com um tamanho apropriado, estas operações têm uma complexidade computacional de O(1).

A implementação da hash table pode ser feita de duas maneiras. Com ***Open Addressing,*** é usado um array de chaves e respetivos valores. Já com ***Separate chaining,*** é usado um array com ponteiros para as heads de ***linked lists*** ou ***binary trees***. Neste trabalho, iremos usar o método de separate chaining, com ponteiros para linked lists.

Separate Chaining

Quando se usa este método, é então usado um array com ponteiros para as linked lists. Cada chave (key), passa pela ***hash function*** que dita qual das linked lists iremos operar (procura, inserção ou remoção). O principal objetivo com o chaining é distribuir as Keys o melhor possível pelas posições do array. Quanto melhor for a distribuição, menos colisões vão existir, e mais rápida será a procura de elementos.

Código implementado

Em termos de estruturas, temos a 'file_data_t' e a 'hash_data'. A primeira servirá para trabalhar com o ficheiro que vamos trabalhar, e a segunda será a estrutura que define um nó (node) das linked lists usadas na hash table.

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

//#define hash_size 1009u
#define MAXVALUE INT_MAX

//-------------------------------------------------------------
typedef struct file_data {

// public data
long word_pos; // zero-based
long word_num; // zero-based
char word[64];

// private data
FILE *fp;
long current_pos; // zero-based
}
file_data_t;


typedef struct hash_data {

    struct hash_data *next;
    char word[64];
    int firstP;
    int IndexLastWord;
    int contador;
    int minD;
    int maxD;
    int totalD;

}hash_data;
```

Usámos também o código já fornecido pelo docente para trabalhar com o ficheiro, visto que facilita a abertura (open_text_file()) e o fecho do ficheiro (close_text_file()). A função read_word() irá ler cada palavra do ficheiro, o que nos será bastante útil.

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

    fd->fp = fopen(file_name,"rb");

    if(fd->fp == NULL)
        printf("Error opening file!\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;
}
```

```c
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;
}
```

Já na parte da hash table, temos a função 'hash_function()', retirada dos slides das Lecture notes, que calcula um hash value dada uma string.

```c
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
return h % s; // due to the unsigned int data type, it is guarantee
}
```

Seguidamente, temos a função 'find_data()', também adaptada da contida nas Lecture notes, que passando como argumentos a key, a hash table e o seu tamanho retorna o node correspondente, se já estiver contido na hash table. Basicamente, calcula o index do array através da hash function e, depois de encontrar a linked list, percorre-a até ao último elemento.

```c
hash_data *find_data(const char *key, hash_data *hashT[], int hashSize) {

        unsigned int idx;
        hash_data *hd;
        idx = hash_function(key,hashSize);
        hd = hashT[idx];

        while(hd != NULL && strcmp(key,hd->word) != 0)
            hd = hd->next;

        return hd;
}
```

Temos também a função new_hash_data() que cria um novo nó com os parâmetros necessários. Há uma pre-alocação da memória necessária para o nó. Depois, basta inicializar os valores.

```
static hash_data *new_hash_data(int index, char * word) {

    hash_data *hd = (hash_data *)malloc(sizeof(hash_data));
    if(hd == NULL) {
        fprintf(stderr,"Out of memory\n");
        exit(1);
    }
    strcpy(hd->word, word);
    hd->firstP = index;
    hd->IndexLastWord = index;
    hd->contador = 1;
    hd->minD = MAXVALUE;
    hd->maxD = 0;
    hd->totalD = 0;
    hd->next = NULL;


    return hd;
}
```

Para nós já existentes, temos a função update() que atualiza os valores de um nó já existente. Para isso, é passado como argumento o nó existente e a posição da palavra repetida. Depois basta calcular a distância entre a posição dada e a última vez que a palavra apareceu e verificar se é maior que 'maxD' ou menor que ´minD´ e, se for, substituir. Incrementa-se um no contador, soma-se à distância total (totalD) e atualiza-se a posição da última palavra para a passada como argumento.

```
void update(hash_data * node,int posicao){

    if((posicao - node->IndexLastWord) > node->maxD)
        node->maxD = (posicao - node->IndexLastWord);

    if((posicao - node->IndexLastWord) < node->minD)
        node->minD = (posicao - node->IndexLastWord);

    node->contador += 1;
    node->totalD += (posicao - node->IndexLastWord);
    node->IndexLastWord = posicao;
}
```

Por fim, temos a função que aumenta o tamanho da hash table: 'resize()'. Esta recebe como argumentos o tamanho atual, a hash table e o novo tamanho pretendido. Aloca-se a memória equivalente ao novo tamanho da hash table e coloca-se tudo a NULL. Em seguida, para cada posição do array de ponteiros (primeiro for loop), percorre-se a linked list correspondente (segundo for loop), sendo o primeiro nó da linked list 'hashtable[i]'. Para cada nó, calcula-se o novo hash value (muda pois o tamanho da hash table também muda). Depois basta manipular os nós de forma a que se coloque na nova hash table. Retorna a nova hash table já com o tamanho pretendido.

```
hash_data** resize(int lastSize, hash_data**hashtable, int newSize){

    hash_data** newHashTable = malloc(newSize * sizeof(hash_data));
    for(int i=0;i<newSize;i++){
        newHashTable[i]=NULL;
    }

    int hc = 0;
    for(int i=0;i<lastSize;i++){
        for(hash_data* node = hashtable[i]; node!=NULL; node = node->next){

            hc = hash_function(node->word, newSize);
            hash_data *tempNode = node;
            tempNode->next=NULL;

            hash_data *head = newHashTable[hc];
            int count = 0;

            while (head!=NULL) {
                head = head->next;
                count++;
            }
            if(count == 0)
                newHashTable[hc] = tempNode;
            else
                head = tempNode;
        }
    }

    return newHashTable;
}
```

Passando agora para a função 'main()', aloca-se memória e abre-se o ficheiro pretendido. Definimos um tamanho inicial para a hash table e aloca-se memória para a mesma. Em seguida, coloca-se todos os seus valores a NULL. Lemos o ficheiro palavra a palavra usando a função read_word (que retorna -1 quando acabam as palavras). Verificamos se o número de palavras excede os 60% do tamanho da hash table e, se sim, redimensionamos a hash table para o dobro do tamanho.

Agora temos duas situações possíveis. Se a palavra não existir na hash table (o find_data vai ser NULL), temos de criar um nó novo. Ou seja, calcular o hash value para achar o index pretendido, depois percorrer essa linked list até ao último valor e adicionar um nó com a palavra pretendida. Por outro lado, se já existir na hash table, temos de chamar a função 'update()', já mencionada anteriormente, para atualizar os dados. Está assim então preenchida a hash table.

Por fim, apenas precisamos de imprimir no terminal toda a informação de cada palavra. Percorre-se a hash table. Para cada index verifica-se se contém algum nó e, se sim, é porque tem uma linked list nesse index. Assim, percorre-se essa linked list e, para cada nó, imprime-se no terminal as informações da palavra: primeira e última posição, número de vezes que aparece e a menor, maior e média distância de ocorrências consecutivas da palavra.

```c
int main(int argc, char const *argv[])
{
    file_data_t *ficheiro = (file_data_t *)malloc(sizeof(file_data_t));
    open_text_file("SherlockHolmes.txt", ficheiro);

    int hash_size = 1009u;
    hash_data ** hash_table = malloc(hash_size * sizeof(hash_data));
    int wordContador = 0;
    int hash_code = 0;
    hash_data *temp = NULL;

    for(int i=0;i<hash_size;i++){
        hash_table[i]=NULL;
    }

    while(read_word(ficheiro) != -1){
        wordContador++;
        if(wordContador > (int)hash_size * 0.6){
            hash_table = resize((int)hash_size, hash_table, (int)hash_size*2);
            hash_size = (int)hash_size * 2;
        }

        if(find_data(ficheiro->word, hash_table, hash_size) == NULL){
            hash_code = hash_function(ficheiro->word, hash_size);
            hash_data *t = hash_table[hash_code];
            int count = 0;
            while(t != NULL){
                t = t->next;
                count++;
            }

            hash_data *newNode = new_hash_data(ficheiro->word_pos, ficheiro->word);
            if(count == 0)
                hash_table[hash_code] = newNode;
            else
                t = newNode;

        }else{
                update(find_data(ficheiro->word, hash_table, hash_size), ficheiro->word_pos);
            }

    }
```

```c
    for(int i=0;i<hash_size;i++){
        if(hash_table[i]!=NULL){
            hash_data *node = hash_table[i];
            while(node != NULL){
                printf("%s  ", node->word);
                printf("First position: %d  ||  ", node->firstP);
                printf("Last position: %d  ||  ", node->IndexLastWord);
                printf("Counter: %d  ||  ", node->contador);
                printf("Largest distance: %d  ||  ", node->maxD);
                if(node->minD == INT_MAX){
                    printf("Smallest distance: 0  ||  ");
                    printf("Average distance: 0\n\n");
                }else{
                    printf("Smallest distance: %d  ||  ", node->minD);
                    printf("Average distance: %d\n\n", (int)( (node->totalD)/(node->contador-1) ));
                }
                node = node->next;

            }
        }
    }

    close_text_file(ficheiro);
    free(ficheiro);

    return 0;
}
```

Conclusão

Em suma, este trabalho ajudou-nos a uma melhor percepção de métodos de criação de código e estruturas de dados, bem como a implementação das hash tables com separate chaining. Enriquecemos também os nossos conhecimentos da linguagem de programação C e na gestão de tempo e de trabalho entre grupo.