Instituto Federal Catarinense - Campus Videira
Aluno: Rafael Bortolozo
Turma: Ciência da computação 2019
Matéria: Estrutura de dados I

Trabalho Final - Tabela Hash

Nesse trabalho, foi realizado uma tabela hash com o objetivo de espalhar, de maneira equilibrada, uma lista contendo 100788 nomes de brasileiros, contidos em um arquivo de texto. Inicialmente, foi feito a inclusão das bibliotecas e declaração de valores constantes:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define CHAVES 59
#define QTDCARACTERES 16
```

Declaração de 2 TADs: a Lista e Elemento:

```c
typedef struct sElemento{
    struct sElemento *next;
    struct sElemento *prev;
    char nome[QTDCARACTERES];
    int id;
}Elemento;

typedef struct sLista{
    struct sElemento *head;
    struct sElemento *tail;
    int size;
}Lista;
```

O tratamento de colisão é resolvido usando encadeamento com listas duplamente encadeadas. A quantidade de Listas vai depender do valor da constante CHAVES (também pode ser representado pela letra M), para ter acesso a cada uma dessas listas, foi declarado um vetor dinâmico do tipo Lista de tamanho M, cada posição do vetor aponta para uma lista diferente. O vetor foi declarado dentro da main(), quando uma função exige os dados, terá que receber como parâmetro o ponteiro, que dá acesso às listas e consequentemente aos dados contidos nelas. A imagem abaixo mostra todas as funções e a main(), em seguida, será comentado as principais funções:

```c
Lista *criaListas();
void readArquivoTXT(Lista *chave);
void imprimeListas(Lista *chave);
void imprimePesquisa(Elemento *pesquisa);
void imprimeRemove(Elemento *remove);
int hash(char *nome);
int ascii(char letra);
void insereElemento(Lista *chave, char *nome);
Elemento *alocaMemoriaNovo_elemento(Lista *chave,char *nome, int nHash);
Elemento *pesquisaElemento(Lista *chave, char *nome);
void removerElemento(Lista *chave, Elemento *nome);
void ajustarID(Elemento *aux);
void freeElementos(Lista *chave);

void inicializa_quicksort(Lista *chave);
int particiona(Lista *chave, int inicio, int fim, int numChave);
int ordenar_pivo_quicksort(int esquerda,int direita,char *vetPivo,Elemento *elementoDireita,Elemento *elementoEsquerda);
Elemento *troca_pivo_direita(int *esq,int *dir,char *vetPivo, Elemento *elementoDireita, Elemento *elementoEsquerda);
Elemento *troca_pivo_esquerda(int *esq,int *dir,char *vetPivo, Elemento *elementoDireita, Elemento *elementoEsquerda);
void quicksort(Lista *chave, int inicio, int fim, int numChave);

main(){
    Lista *chave= criaListas();
    readArquivoTXT(chave);

    //Pesquisa e remocao de nomes
        char pesquisar[]="ADALVINO";
        //imprimePesquisa(pesquisaElemento(chave, pesquisar));
        //imprimeRemove(pesquisaElemento(chave, pesquisar));
        //removerElemento(chave, pesquisaElemento(chave, pesquisar));
    //freeElementos(chave);
    //imprimeListas();

    //Ordenacao de elementos
    inicializa_quicksort(chave);
    imprimeListas(chave);
}
```

A função criaListas() vai criar uma lista para cada posição do vetor:

```
Lista* criaListas(){
    Lista *vetListas= (Lista*)malloc(CHAVES * sizeof(Lista));
    for(int i=0; i<CHAVES; i++){
        vetListas[i].head=NULL;
        vetListas[i].tail=NULL;
        vetListas[i].size=0;
    }
    return vetListas;
}
```

A função readArquivoTXT() lê cada linha do arquivo nomes.txt, guardando os nomes em um vetor de char e chamando a função de inserir elemento, mas antes, deve-se remover o caractere "\n" que o fgets() captura em cada nome:

```
void readArquivoTXT(Lista *chave){
    FILE *file = fopen("nomes.txt", "r");
    char nome[QTDCARACTERES];
    while(fgets(nome, QTDCARACTERES, file) != NULL){
        nome[strcspn(nome, "\n")]='\0';
        insereElemento(chave, nome);
    }
    fclose(file);
}
```

As funções hash() e ascii() trabalham juntas, elas servem, respectivamente, para calcular o valor da chave e converter a letra para decimal. A função hash() é utilizada principalmente na hora da inserção de elementos, informando em qual chave o elemento deve entrar, também é usada na pesquisa e remoção de nomes, para saber em qual chave o nome informado deve ser procurado. Foi usado uma função modular, onde é feito a soma das letras, convertidas pra decimal, e calcula-se o resto da divisão por M. Nesse trabalho foi usado M=59 por nenhum motivo especifico, mas é bom ter um valor M alto para deixar os elementos bem espalhados, assim melhorando a performance para pesquisar, remover, ordenar elementos, etc....

```
int ascii(char letra){
    return letra;
}

int hash(char *nome){
    int x=0;
    for(int i=0; i<strlen(nome) ;i++){
        x= x+ascii(nome[i]);
    }
    x= x % CHAVES;
    return x;
}
```

A função insereElemento() faz a inserção dos nomes em suas respectivas chaves, que é calculado usando a função hash explicado anteriormente. O código é praticamente igual ao que foi apresentado na aula de lista encadeada dupla:

```
void insereElemento(Lista *chave, char *nome){
    int nHash= hash(nome);
    Elemento *pivo= chave[nHash].tail;
    Elemento *novo_elemento= alocaMemoriaNovo_elemento(chave, nome, nHash);

    if(chave[nHash].size==0){
        chave[nHash].head=novo_elemento;
        chave[nHash].head->prev=NULL;
        chave[nHash].head->next=NULL;
        chave[nHash].tail=novo_elemento;
    }else{
        novo_elemento->next=pivo->next;
        novo_elemento->prev=pivo;

        if(pivo->next==NULL){
            chave[nHash].tail= novo_elemento;
        }else{
            pivo->next->prev=novo_elemento;
        }
        pivo->next=novo_elemento;
    }
    chave[nHash].size++;
}
```

A função pesquisaElemento() e removerElemento() trabalham juntas, a função removerElemento() remove o elemento que for recebido como parâmetro, para isso, entra em ação o pesquisaElemento(), que realiza uma pesquisa retornando o elemento que possui o mesmo nome informado na main(). Antes de excluir o elemento, é realizado a correção dos IDs, decrementando -1 em todos os IDs posteriores. A parte de impressão dos resultados fica responsável pelas funções de imprimir.

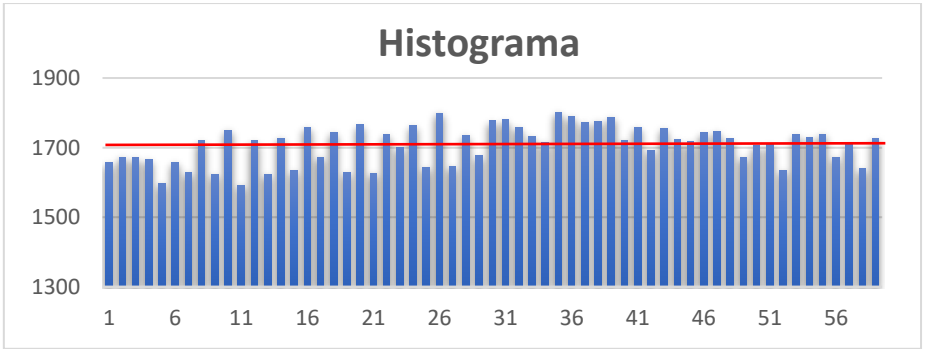
```c
Elemento *pesquisaElemento(Lista *chave, char *nome){
    int nHash= hash(nome);
    Elemento *aux= chave[nHash].head;
    for(int x=0; x<chave[nHash].size; x++){
        if(strcmp(aux->nome, nome)==0){
            return aux;
        }
        aux=aux->next;
    }
    return NULL;
}
```

```c
void removerElemento(Lista *chave, Elemento *aux){
    if(aux!=NULL){
        int nHash= hash(aux->nome);
        if(aux==chave[nHash].head){
            chave[nHash].head = aux->next;
            if(chave[nHash].head==NULL){
                chave[nHash].tail=NULL;
            }else{
                aux->next->prev=NULL;
            }
        }else{
            aux->prev->next = aux->next;
            if(aux->next==NULL){
                chave[nHash].tail= aux->prev;
            }else{
                aux->next->prev = aux->prev;
            }
        }
        ajustarID(aux);
        free(aux);
        chave[nHash].size--;
    }
}
```

A função destroiElementos() desaloca toda a memória utilizada pelos elementos e listas:

```c
void freeElementos(Lista *chave){
    Elemento *aux;
    for(int i=0; i<CHAVES; i++){
        while(chave[i].size>0){
            aux= chave[i].head;
            chave[i].head = aux->next;
            if(chave[i].head==NULL){
                chave[i].tail=NULL;
            }else{
                aux->next->prev=NULL;
            }
            free(aux);
            chave[i].size--;
        }
    }
    free(chave);
}
```

Antes de mostrar a metodologia utilizada no método de ordenação quicksort, temos que analisar o histograma da tabela hash. Usando M=59 e a função modular explicado anteriormente, obtém-se esse gráfico:



O eixo Y indica a quantidade de nomes e o eixo X indica as chaves. A linha vermelha mostra a média do gráfico, que é aproximadamente **1708**. O desvio padrão é **54,84**. Pode-se considerar que a tabela ficou bem equilibrada, mas não 100% uniforme, pois é praticamente impossível criar uma função que satisfaça a hipótese do hashing uniforme.

Quicksort

Primeiramente, o código foi desenvolvido com base em uma vídeo-aula, disponível no link: https://www.youtube.com/watch?v=spywQ2ix_Co&t=168s. Nesse vídeo é mostrado a implementação do quicksort utilizando vetor, então esse código foi adaptado para funcionar com listas, por isso que cada elemento armazena um ID, ele serve para simular um vetor, olhe o exemplo a seguir onde deseja-se acessar os dados na posição 10:

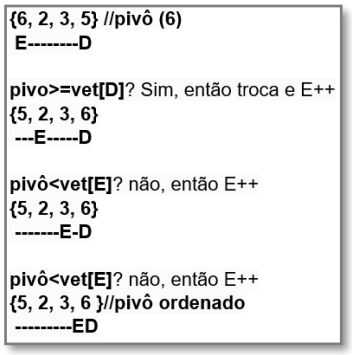
Com vetor: vet[10];

Com lista:

```
Elemento *aux= lista->head;
While(aux->id != posicao){
        Aux=aux->next;
}
```

Esta adaptação infelizmente não deu certo, conforme pode ser visualizado em alguns commits no Github, então, em vez de ficar corrigindo um algoritmo que nem era de minha autoria, optei por readaptar esse mesmo código para funcionar que nem os passo-a-passos feitos na prova teórica, para isso, foi estudado o funcionamento do quicksort em um vídeo do youtube: https://www.youtube.com/watch?v=oYmHH1-f_L0&t=170s. Analisando o vídeo, pode-se imaginar um outro algoritmo.

Primeiro é definido um pivô, normalmente é escolhido o primeiro elemento da lista. Também é declarado dois inteiros, chamados Esquerda (início da lista) e Direita (final da lista), que indicam quais elementos devem ser comparados. Primeiro é feito um while, onde o pivô vai ser comparado com o elemento da direita até encontrar alguém menor que o pivô, ao encontrar, realiza-se a troca dos valores. Depois é feito mais um while, porém, comparando o pivô com o elemento da esquerda até encontrar alguém maior que o pivô, ao encontrar, realiza-se a troca dos valores. Esse ciclo se repete até que os inteiros Esquerda e Direita tenham o mesmo valor, indicando que o pivô está ordenado. Depois de ordenado, vai gerar 2 partições, então é aplicado o quicksort para ambas recursivamente. Exemplo:



A implementação do quicksort pode ser visualizada no arquivo hashing.cpp, disponível no link: https://github.com/RafaelBortolozo/Trabalho_final-Estrutura_de_Dados