Instituto Federal Catarinense - Campus Videira

Turma - Ciência da computação – 3º semestre 2021

Aluno: Pedro Henrique Fagundes

Trabalho Final-Estrutura de Dados.

Implementação de uma Tabela de Espalhamento (Tabela Hash)

Este trabalho procura implementar uma tabela de espalhamento, mais conhecido como tabela hash, feita especificamente com listas duplamente encadeadas em linguagem C.

Oque é uma tabela hash? Resumidamente é uma estrutura de dados que associa várias “chaves” de pesquisa a valores, seu objetivo é a partir de uma chave, fazer uma busca rápida e obter um dado/valor desejado. As tabelas hash são tipicamente utilizadas para grandes volumes de dados.

O Algoritmo

Como dito o algoritmo foi feito em C, usando suas respectivas bibliotecas que foram necessárias para o funcionamento, bibliotecas como:

- <stdio.h>, <stdlib.h>, -<string.h>.

Após isso foi declarado os as estruturas necessárias, “Elemento”, “ElementoChave”, “Hash”:

```c
typedef struct sElemento{

struct sElemento* previous;
struct sElemento* next;
char* nome;

}Elemento;

typedef struct sElementoChave{

struct sElementoChave* previous;
struct sElementoChave* next;
int chave;
//
struct Elemento* head;
struct Elemento* tail;
int size;

}ElementoChave;

typedef struct sHash{

struct sElementoChave* head;
struct sElementoChave* tail;
int size;

}Hash;
```

Com as principais estruturas já declaradas, o próximo passo foi declaras as funções que serão usadas:

```
Elemento* criaElemento(char*);
void insereElemento(ElementoChave*,Elemento*,char*);
void removeElemento(ElementoChave*,Elemento*);
void mostrarLista(ElementoChave*);
void mostrarListaTail(Hash*);
Elemento* encontrarElemento(Hash*,char*);
//=============================================
        //Tabela Hash
Hash* criaHash();
void mostrarListaChaves(Hash*);
ElementoChave* encontrarChave(Hash*,int);
void encontrarNome(Hash*,char*);
int buscarNome(ElementoChave*, char*);
void inicializarTabelaChaves(Hash*,int);
void insereElementoChave(Hash*,ElementoChave*,int);
int funcaoHash(char*);
void lerArquivo(Hash*);
void escreveArquivo(Hash*);
void mostrarListas(Hash*);
void mostrarNomes(Hash*);
void inserirNomeHash(Hash*,char*);
FILE* abreArquivo(char*,char*);
void removeNome(Hash*,char*);
void quicksort(Elemento*, Elemento*);
void swap(Elemento*,Elemento*);
```

As funções “criaElemento”, “insereElemento”, “removeElemento”, “mostrarLista”, montam as listas encadeadas duplas, elas são usadas nas próximas funções

Funções hash

```
Hash* criaHash();
void inicializarTabelaChaves(Hash*,int);
void insereElementoChave(Hash*,ElementoChave*,int);
ElementoChave* criaElementoChave(int);
```

As quatro primeiras funções são usadas para criar a lista hash, onde as chaves que apontaram para os dados, vão ficar:

```
//Inicializa a tabela hash com as chaves
void inicializarTabelaChaves(Hash* listaChave,int tamanho){

   for(int i = 0;i<tamanho;i++){
    insereElementoChave(listaChave,listaChave->tail,i);
   }

}
//aloca a lista hash
Hash* criaHash(){

    Hash* lista=(Hash*)malloc(sizeof(Hash));

    lista->head = NULL;
    lista->tail = NULL;
    lista->size = 0;



    return lista;
}
//aloca uma chave
ElementoChave* criaElementoChave(int chave){

    ElementoChave* novo = (ElementoChave*)malloc(sizeof(ElementoChave));

    novo->previous = NULL;
    novo->next = NULL;
    novo->head = NULL;
    novo->tail = NULL;
    novo->chave = chave;
    novo->size = 0;

    return novo;
}
```

```
//insere um a chave na lista das chaves
void insereElementoChave(Hash* lista, ElementoChave* pivo,int chave){

    ElementoChave* novo_elemento = criaElementoChave(chave);



        if(lista->size == 0){
            lista->head = novo_elemento;
            lista->tail = novo_elemento;
        }else{
            novo_elemento->next = pivo->next;
            novo_elemento->previous = pivo;

                if(pivo->next==NULL){
                    lista->tail=novo_elemento;
                }else{
                    pivo->next->previous = novo_elemento;
                }

            pivo->next = novo_elemento;
        }

    lista->size++;
}
```

Além de criar as chaves, também é alocado uma lista para respectiva chave, e nessa lista ficará os dados;

Em seguida as funções que fazem o cálculo do hash e adicionam os dados em suas respectivas chaves:

```
FILE* abreArquivo(char*,char*);
void lerArquivo(Hash*);
int funcaoHash(char*);
void inserirNomeHash(Hash*,char*);
```

Primeiramente a função “abreArquivo”, abre um aquivo .txt com várias nomes, e a função “lerArquivo” por meio de parâmetros passados lê o arquivo linha por linha, lendo cada nome e jogando direto para a “funcaoHash”, nessa função um estrutura de repetição le cada letra do nome e a transforma em ASCII, para assim então fazer a equação que retornará em qual chave o nome será colocado, a equação é:

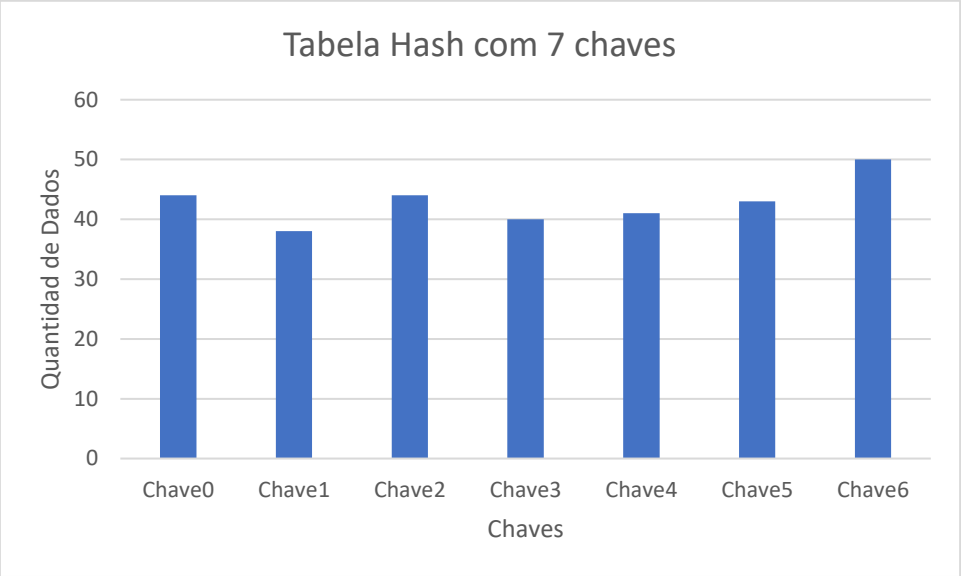
soma = (31 * soma + (int)nome[j]) % TAM;

```
//função hash que retorna a chave que o nome deve ser inserido
int funcaoHash(char* nome){
int soma = 0;
    for (int j = 0; j < strlen(nome); j++){
        soma = (31 * soma + (int)nome[j]) % TAM;
    }

    return soma;
}
```

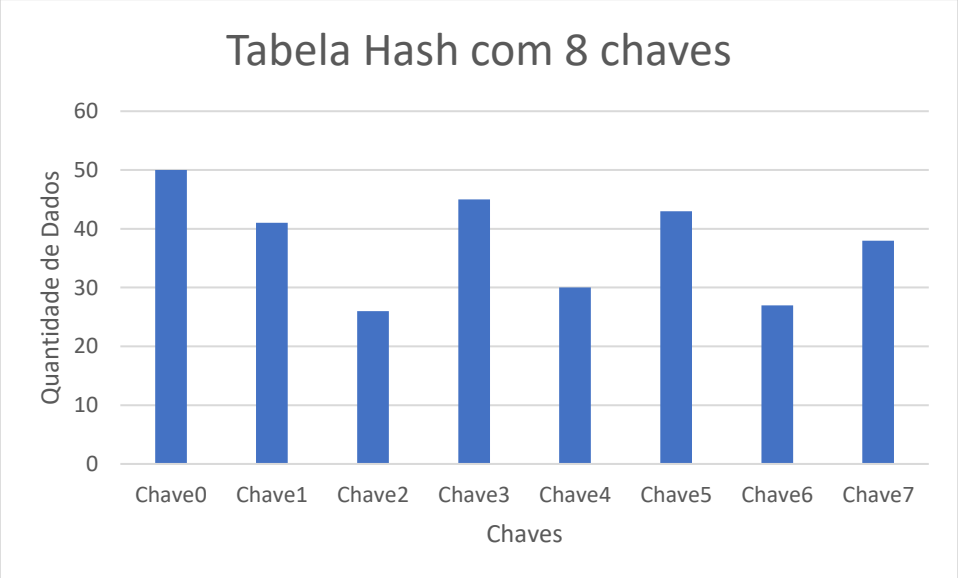
A tabela hash sofre de um problema inevitável chamado colisão, onde dois dados vão para a mesma chave, oque se pode fazer é tratar esse problema, e um jeito é usando lista encadeadas ou duplamente encadeadas, onde se houver colisão a lista automaticamente aumenta e adiciona o novo dado.

Um ponto importante é a quantidade de dados em cada lista, por isso é necessário analisar a quantidade de dados que vão ser inseridos e a quantidade de chaves que vai ter. A comunidade cientifica diz que por um motivo desconhecido um número primo de quantidade de chaves tem melhor desempenho em deixar a tabela hash uniforme, segue um exemplo de uma tabela com 7 chave e a outra com 8 chaves.

-7 chaves:



-8 chaves:



Segue as funções de inserção, “abreArquivo”, “lerArquivo”, “inserirNomeHash”.

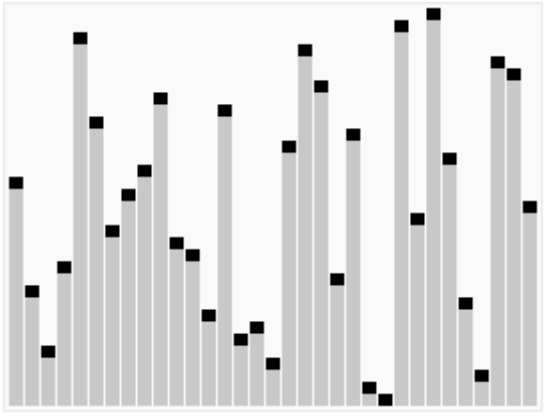
```c
//Abre o arquivo de texto
FILE* abreArquivo(char* mode,char* nomeArquivo){
    FILE* arquivo = fopen(nomeArquivo, mode);
    if (arquivo == NULL) {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE);
    }
    return arquivo;
}

//Le o arquivo e insere na tabela
void lerArquivo(Hash* hash){
FILE *arquivo;
char* nome[15];
char *result;
arquivo = abreArquivo("rt", "nomesteste.txt");
int i = 0;
        while (!feof(arquivo)){
            result = fgets(nome, 99, arquivo);
            if (result != NULL){
                nome[15] = result;
                nome[15][strcspn(nome[15], "\n")] = 0;
                inserirNomeHash(hash,nome);
            }
            i++;
        }
    fclose(arquivo);
}

//insere o na tabela hash
void inserirNomeHash(Hash* hash,char* nome){
int chave = funcaoHash(nome);
ElementoChave* lista = encontrarChave(hash,chave);
insereElemento(lista, lista->head, nome);
quicksort(lista->head,lista->tail);

    //printf("\n\nChave [%i] -> Adicionou o nome: %s\n",chave,nome);
    return;
}
```

Algoritmo de ordenação

Logo após a inserção, cada lista de nomes é ordenada com a estrutura de ordenação quicksort. Um algoritmo de ordenação como o próprio nome já diz, lê uma quantidade x de dados e os ordena de uma respectiva forma, nesse caso que os dados são nomes, será ordenado em ordem alfabética, existe varios algoritimos de ordenação, como “bubble sort”, “Tim sort”, “Insertion sort”, sendo um melhor que o outro importando a quantidade e o tipo de dado, segue o algoritimo de ordenação quicksort:

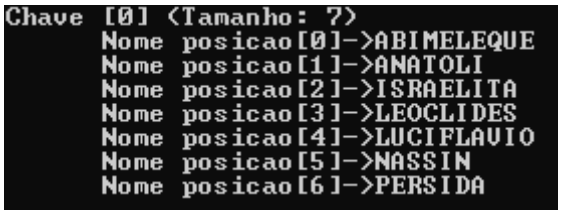
```c
// Ordenacao Quicksort
void quicksort(Elemento* start, Elemento* end){
  if (end != NULL && start != end && start != end->next){
    Elemento* i = start->previous;
    Elemento* pivo = end;

    // Percorre a lista do início ao fim
    for (Elemento *j = start; j != end; j = j->next){
      // Se o valor for menor que o pivô, trocar com o i
      if (strcmp(j->nome, pivo->nome) < 0){
        i = (i == NULL ? start : i->next);
        swap(i, j);
      }
    }
    // Coloca o pivô no lugar certo
    i = (i == NULL ? start : i->next);
    swap(i, pivo);

    // Ordena as duas partes da lista
    quicksort(start, i->previous);
    quicksort(i->next, end);
  }
}

void swap(Elemento* a, Elemento* b){
  char* aux = (char*) malloc(sizeof(char) * 15);
  strcpy(aux, a->nome);
  strcpy(a->nome, b->nome);
  strcpy(b->nome, aux);
  free(aux);
}
```

Primeiramente o programador deve escolher um dado da lista para ser o pivô, dele o algoritmo reorganiza os dados em duas listas, a lista a esquerda do pivô fica com os dados menores que o pivô, e a lista da direita fica com os dados maiores que o pivô. Com a tabela divida e parcialmente ordenada, fica mais facil de ordenar as duas pares separadamente, e com isso a lista inteira fica ordenada, segue um exemplo gráfico:



Exemplo de uma lista não ordenada e um ordenada:

```
Chave [0] (Tamanho: 7)
      Nome posicao[0]->ABIMELEQUE
      Nome posicao[1]->NASSIN
      Nome posicao[2]->ANATOLI
      Nome posicao[3]->LEOCLIDES
      Nome posicao[4]->ISRAELITA
      Nome posicao[5]->LUCIFLAVIO
      Nome posicao[6]->PERSIDA
```



Com uma lista ordenada, a busca por um dado fica muito mais rápida.

Referências

https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-hash.html

https://pt.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort

https://pt.wikipedia.org/wiki/Ficheiro:Sorting_quicksort_anim.gif

https://pt.wikipedia.org/wiki/Tabela_de_dispers%C3%A3o