Instituto Federal Catarinense - Campus Videira
Aluna: Suellen Oliveira 3º semestre

Projeto Final de Estrutura de Dados I - Tabela de espalhamento

A proposta para o trabalho final da disciplina, foi implementar em uma tabela hash uma lista contendo 100.788 nomes de brasileiros e ordená-los utilizando quicksort de Hoare ou um método de ordenação de complexidade algorítmica equivalente.

Para a implementação utilizei 3 bibliotecas e a constante igual a 53 que é usada para definir a quantidade de chaves da tabela hash e esse número foi fornecido pelo professor.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TAM 53
```

E visando tratar as colisões na tabela hash, utilizei lista encadeada dupla, assim a alocação de memória será dinâmica,o que permitirá que não ocupe espaço de memória desnecessário.

Declaração da lista e nó:

```c
typedef struct no {
    char *info;
    struct no *prox;
    struct no *ant;
} No;

typedef struct lista {
    struct no *head;
    struct no *tail;
    int size;
} Lista;
```

A lista encadeada usada foi a mesma usada para os outros trabalhos da disciplina e tem três funções principais: inserirNaLista, RemoverDaLista e buscaNo. A inserção na lista acontece pela tail e a remoção pela head. A função buscaNo percorre a lista da head para a tail comparando as strings.

```c
void inserirNaLista(Lista *lista, char info[]) {
    No *novo = criaNo(info);
    No *pivo = lista->tail;

    if(lista->size == 0){
        lista->head = novo;
        lista->tail = novo;
    }else{
        novo->prox = pivo->prox;
            novo->ant = pivo;
        if (pivo->prox == NULL) {
                lista->tail = novo;
        }else{
            pivo->prox->ant = novo;
        }
        pivo->prox = novo;
    }
        lista->size++;
}
```

```c
int RemoverDaLista(Lista *lista, No *no){
     if((no!= NULL) && (lista->size!=0)){

        if(no == lista->head){
          lista->head = no->prox;

          if(lista->head == NULL){
             lista->tail = NULL;
          }else{
              no->prox->ant = NULL;
              }
       }else{
            no->ant->prox = no ->prox;
            if(no->prox == NULL){
                lista->tail = no->ant;
            }else{
                no->prox->ant = no->ant;
            }
     }
       free(no);
       lista->size--;
   }
   return 0;
}
```

```c
No* buscaNo(Lista *lista, char info[]){

  No *busca = lista->head;
  while(busca!=NULL){
    if(strcmp(busca->info, info)==0){
      return busca;
    }
    busca=busca->prox;
  }
  return NULL;
}
```

Para o Quicksort usei como referência o código de Hoare fornecido pelo professor e modifiquei-o para usar lista encadeada dupla.

Passos do algoritmo de Hoare (Completo)

```
procedimento QuickSort(X[], IniVet, FimVet)
var
   i, j, pivo
início
   i <- IniVet
   j <- FimVet
   pivo <- X[(IniVet + FimVet) div 2]
   enquanto(i <= j)
    |      enquanto (X[i] < pivo) faça
    |       |   i <- i + 1
    |      fimEnquanto
    |      enquanto (X[j] > pivo) faça
    |       |   j <- j - 1
    |      fimEnquanto
    |      se (i <= j) então
    |       |   troca(X[i], X[j])
    |       |   i <- i + 1
    |       |   j <- j - 1
    |      fimSe
   fimEnquanto
   se (IniVet < j) então
    |  QuickSort(X, IniVet, j)
   fimSe
   se (i < FimVet) então
    |  QuickSort(X, i, FimVet)
   fimSe
fimProcedimento
```

12

Código criado com lista encadeada dupla:

```c
void Quicksort(Lista *lista, No *inicio, No *fim){
    No *i = inicio;
    No *f = fim;
    No *pivo = Meio(lista, inicio, fim);
    char *InfoDoPivo = pivo ->info;

    while((i != NULL) && (f != NULL) && (Igual(i,f))){

        while((i != NULL) && (pivo != NULL) && (strcmp(i->info, InfoDoPivo) < 0)){
            i = i->prox;
        }

        while((f != NULL) && (pivo != NULL) && (strcmp(f->info, InfoDoPivo) > 0)){
            f = f->ant;
        }

        if((i != NULL) && (f != NULL) && (Igual(i,f))){
            Trocar(i,f);
            i = i->prox;
            f = f->ant;
        }
    }

    if((inicio != NULL) && (f != NULL) && (Menor(inicio,f))) {
        Quicksort(lista, inicio, f);
    }
     if((i != NULL) && (fim != NULL) && (Menor(i,fim))){
        Quicksort(lista, i, fim);
    }
}
```

O Quicksort de Hoare usa, em sua forma mais eficiente, o meio da lista como pivô, e para definir o meio da lista eu criei uma função que percorre a lista e vai incrementando uma posição até chegar ao fim. O resultado desse contador é guardado na variável pivo e depois divido por 2, e assim temos o meio da lista.

```c
No* Meio(Lista *lista, No *inicio, No *fim){
     if(lista->size != 0){
        int pivo = 0;
        int i;
        No *busca = inicio;
        while (busca != fim){
            pivo++;
            busca = busca -> prox;
        }
        pivo = pivo/2;
        busca = inicio;
        for(i = 0; i < pivo; i++) {
            busca = busca->prox;
        }
        return busca;
    }
    return NULL;
}
```

Para realizar a troca dos elementos foi criada a função trocar, que realiza a troca das informações por meio da variável aux.

```c
void Trocar(No *i, No *f){
    char *aux = i->info;
    i->info = f->info;
    f->info = aux;
}
```

O Quicksort compara as duas informações e se a da esquerda for maior que a da direita ele troca, e os ponteiros continuam percorrendo até eles se encontrarem, porém como estamos utilizando lista não conseguimos usar o < > para comparar a posição dos ponteiros e para resolver esse problema foram criadas duas funções, sendo elas:

```c
int Igual(No *i,No *f){
    No *busca = i;
    while (busca != NULL){
        if(busca == f){
            return 1;
        }else{
            busca = busca ->prox;
        }
    }
    return 0;
}
```

```c
int Menor(No *i,No *f){
    No *busca = i;
    while (busca != NULL){
        if(busca == f->ant){
            return 1;
        }else{
            busca = busca ->prox;
        }
    }
    return 0;
}
```

Ambas tem um ponteiro que eu chamei de i, que vai verificar se ele está em uma posição antes do f.

Para ler o arquivo .txt usei a função abaixo e quando o arquivo é lido ele já é inserido na tabela hash.

```c
void ler(char f[], Lista *t) {
    FILE *file = fopen(f, "r");
    char text[20];

    if (file) {
        printf("\nTexto lido\n");
        while (fgets(text, 20, file)) {
            text[strcspn(text, "\n")] = '\0';
            inserirTabela(t, text);
            printf("%s\n", text);
        }
        fclose(file);
    } else {
        printf("\nErro ao abrir o arquivo.\n");
    }
}
```

Para a tabela hash foi definido um número de índices, 53, e cada um desses índices recebe uma lista encadeada onde serão armazenados os nomes.

A função a seguir, inicializa a tabela chamando a função criaLista, que está presente no código da lista encadeada e que aloca memória para a lista e define head e tail como NULL e o tamanho da lista para 0, para cada um dos 53 índices.

```c
void inicializarTabela(Lista t[]){
    int i;
    for(i = 0; i < TAM; i++){
        t[i] = *criaLista();
    }
}
```

A função Hash é a principal função da tabela hash, e define para qual índice da tabela cada nome vai. A função recebe uma string e retorna o valor inteiro que é referente ao índice. A variável “i” serve como um contador para o for e “tam” recebe o comprimento da string pela função strlen da biblioteca string.h. O código ASCII das strings é multiplicado por i + 1 e somado todas, para atribuir um valor diferente a cada caractere, e é adicionado na variável hash. E por fim, o valor de hash é retornado e dividido pelo tamanho da tabela, o resto dessa divisão é o índice para o qual o nome vai.

```c
int funcaoHash(char info[]){
    int i, tam = strlen(info);
    int hash = 0;

    for ( i = 0; i < tam; i++){
        hash += info[i] * (i + 1);
    }
     return hash % TAM;
}
```

A função inserir na tabela chama a função hash, para saber em qual índice vai inserir o nome, e insere-o na lista presente dentro do índice usando a função inserirNaLista.

```c
void inserirTabela(Lista t[], char info[]){
    int i = funcaoHash(info);
    inserirNaLista(&t[i], info);
}
```

A função buscaTabela usa a função buscaNo da lista encadeada, que percorre a lista e compara os nomes, e a função hash para informar o índice em que o nome está presente.

```c
No *buscaTabela(Lista t[], char info[]){
    int i = funcaoHash(info);
    printf("\nIndice: %d\n", i);
    return buscaNo(&t[i], info);
}
```

A função imprimirTabela usa a função ImprimeLista e mostra para o usuário a tabela completa, com os nomes e índices.

```c
void imprimirTabela(Lista t[]){
    int i;
    for(i = 0; i < TAM; i++){
        printf("\tIndice %d: ", i);
        ImprimeLista(&t[i]);
        printf("\n");
    }
}
```

A função removerTabela usa a função buscaTabela para encontrar o nome fornecido pelo usuário, se encontrar chama a função RemoverDaLista e se não encontrar aparece “Nome não encontrado”.

```c
void removerTabela(Lista t[], char info[]){
    No *no = buscaTabela(t, info);

    if (no != NULL){
        int i = funcaoHash(info);
        RemoverDaLista(&t[i], no);
        printf("Nome removido: %s\n", info);
    }else{
        printf("Nome não encontrado\n");
    }

}
```

A função a seguir recebe um índice da tabela e ordena os nomes que estão nele em ordem alfabética. Ela chama outras duas funções, a ImprimeLista para mostrar a lista para o usuário e a Quicksort que é a função que vai ordenar os nomes.

```c
void imprimirEOrdenarIndice(Lista t[], int i){
    if(i >= 0 && i < TAM){
        printf("\tIndice %d: ", i);
        Quicksort(&t[i], t[i].head, t[i].tail);
        ImprimeLista(&t[i]);
        printf("\n");
    }else{
        printf("Não foi possivel encontrar esse indice\n");
    }
}
```

Respondendo perguntas

1.Quando deveria ser implementado tratamento de colisão?
2.Como poderia ser esta implementação?
3.Qual sua avaliação da tabela hash gerado em relação a hipótese do hashing uniforme?
4. Qual sua análise em relação ao histograma de frequência de cada uma das chaves da tabela hash?

O tratamento de colisão usado foi o de lista encadeada dupla e ele foi implementado na inserção dos dados na tabela hash, o que faz com que a quantidade de dados alocados seja dinâmico e não ocupe memória extra. O desvio padrão das chaves do hash foi em torno de 38, o que é um desvio baixo, então eu acredito que o hashing está uniforme. O histograma mostra a uniformidade com

mais clareza, já que nele podemos observar que não existe uma diferença tão grande de dados de uma chave para a outra.