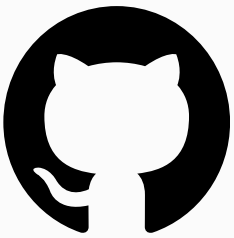
TRABALHO T2

Pesquisa e Ordenação de Dados

João Luis Almeida Santos - **Matrícula: 202400082508**

SUMÁRIO



Clique aqui para acessar o repositório no GITHUB

I. Introdução

Este relatório descreve o funcionamento de um programa em linguagem C cujo propósito é ler dados numéricos a partir de um arquivo e construir uma **tabela hash** com feito com nódulos (nodes) para o armazenamento dos números em um array de ponteiros. O código também inclui funções auxiliares para visualização da tabela e busca de elementos.

Aqui estão as bibliotecas utilizadas e as variáveis globais usadas no código:

```
#ifdef _WIN32
// Configurado para o meu computador, alterar se for usar no Windows
#define INPUT "C:\\Users\\neoka\\CLionProjects\\Trabalho---POD\\dados.txt"
#define EMPTY_VALUE -1
#else
// Mudei o sistema de arquivos, caso não consiga ler o arquivo, por favor mude o caminho aqui
#define INPUT "Trabalho T2/dados.txt"
#define EMPTY_VALUE -1
#endif
#include <stdio.h>
#include <stdlib.h>
```

II. Desafios e Evolução do Código

A implementação do código foi realizada de maneira relativamente rápida. Os maiores desafios enfrentados concentraram-se na etapa de leitura e processamento dos arquivos de entrada. A manipulação de strings e a extração correta dos valores numéricos exigiram atenção. Felizmente, pude simplesmente reutilizar a função que eu já havia criado anteriormente para o último trabalho.

A implementação feita a partir dessa etapa, com as structs, no entanto, pareceu mais simples para mim. Lidar com structs é algo que foi bastante trabalhado na disciplina de estrutura de dados, e a lógica se desenvolveu de forma fluída a partir dessa etapa.

Inicialmente eu havia esquecido de inserir os dois tipos de tabela hash. No entanto, isso foi implementado na versão final do trabalho, onde se utiliza uma switch-case para alternar entre a tabela de sondagem linear e a tabela feita a partir de listas encadeadas, com base na segunda linha do arquivo.

III. Arquitetura da Implementação

Leitura dos dados

Conforme pedido pela demanda do trabalho, a função makeNumList lê um arquivo de texto contendo números organizados. Ela primeiro abre o arquivo, verifica se está acessível e, em seguida, lê a primeira linha para obter um valor que será usado para definir o tamanho da Hash Table. Depois, ela lê a linha seguinte, que contém vários números separados por ponto e vírgula, conta quantos números existem, aloca memória para armazená-los e converte esses números de texto para valores inteiros. Ao final, a função retorna esses números e o tamanho definido para uso posterior no programa. Caso ocorra algum erro durante a leitura ou alocação, ela informa o problema e interrompe o processo.

```
int main () {

 FILE *f = fopen(INPUT, "r");

    // A função vai adicionando os caracteres dos números a uma string
    // até encontrar um caracter vazio, ponto e vírgula, ou o fim do arquivo
    // Ela ignora o primeiro número e guarda ele, e no fim ela retorna a quantidade de dados e os numeros

    if (f == NULL) {
        printf("[ERRO] Não foi possível abrir o arquivo, saindo.\n");
        return -1;
    }


    // Pega primeira linha e aloca lista do tamanho dos numeros
```

```c
char data_count_string[1000];
fscanf(f, "%[^\n]", data_count_string);
int hash_list_size = strtol(data_count_string, NULL, 10);

fgetc(f);
// Limpar o buffer

char hash_type_string[1000];
if (fgets(hash_type_string, sizeof(hash_type_string), f) == NULL) {
    printf("[ERRO] Erro lendo o arquivo dados.txt\n");
    fclose(f);
    return 0;
}
int hash_type = strtol(hash_type_string, NULL, 10);
fgetc(f);

char number_line[1000];
if (fgets(number_line, sizeof(number_line), f) == NULL) {
    printf("[ERRO] Erro lendo o arquivo dados.txt\n");
    fclose(f);
    return 0;
}

// Count = 1 para caso o último número não tenha ;
int count = 1;
for (int i = 0; number_line[i]; ++i) {
    if (number_line[i] == ';') count++;
}

// Aloca lista baseada na quantidade de numeros
int* num_list = malloc(count * sizeof(int));
if (num_list == NULL)
{
    printf("[ERRO] Erro lendo o arquivo dados.txt\n");
    fclose(f);
    return -1;
}

int i = 0;
int j = 0;
char number[16];
```

```
    // Adiciona os números em uma string até achar ";", depois converte pra int
    for (int k = 0; number_line[k] && j < count; ++k) {
        char c = number_line[k];
        if (c == ';' || c == '\n' || c == '\0') {
            if (i > 0) {
                number[i] = '\0';
                num_list[j++] = strtol(number, NULL, 10);
                i = 0;
            }
        } else if (i < 15) {
            number[i++] = c;
        }

    }

    // Retorna o endereço pro tamanho da hash_table e a lista de numeros pro input da função
    *hash_size_ptr = hash_list_size;
    *num_count_ptr = count;
    *num_list_ptr = num_list;
    *hash_table_type = hash_type;

    return 0;
}

}
}
```

Node

Para lidar com números repetidos na Hash Table feita a partir de listas encadeadas, foi criada a seguinte estrutura, cujo propósito é armazenar uma lista de valores lineares que pode-se percorrer caso hajam valores repetidos. A tabela feita a partir de nós é acessível quando se coloca o número 0 na segunda linha do arquivo dados.txt.

Faz-se a construção da Hash Table como a seguir:

```
// Para utilizar-se na hash-table
typedef struct Node
{
    int value;
    struct Node *next;
} Node;

// Forma supostamente mais segura de alocar um array de pointers pra uma struct, inicializa todos os pointers p
Node** hash_table = (Node**)calloc(hash_table_size, sizeof(Node*));
for (int i = 1; i < data_count; i++)
{
    int target_index = number_list[i] % hash_table_size;
    if (hash_table[target_index] == NULL)
    {
        hash_table[target_index] = (Node*)malloc(sizeof(Node));
        hash_table[target_index]->value = number_list[i];
        hash_table[target_index]->next = NULL;
    }
    else
    {
        Node *current = hash_table[target_index];
        while (current->next != NULL)
        {
            current = current->next;
        }
        Node* next_hash = (Node*)malloc(sizeof(Node));
        next_hash->value = number_list[i];
        next_hash->next = NULL;
        current->next = next_hash;
    }
```

Para a leitura da Hash Table com nós, temos duas funções:

```c
void printHashTable(Node** table, int size) {
    for (int i = 0; i < size; i++) {
        printf("Index %d:", i);
        Node* current = table[i];
        while (current != NULL) {
            printf(" %d ->", current->value);
            current = current->next;
        }
        printf(" NULL\n");
    }
}

int findInHashTable(int num, Node** hash_table, int hash_size)
{
    int target_index = num % hash_size;
    if (hash_table[target_index] == NULL)
    {
        return -1;
    }
    return target_index;
}
```

Sondagem linear

Para o segundo tipo de Hash Table, acessível quando se coloca o número 1 na segunda linha do arquivo **dados.txt**.

```c
void makeLinearHashTable(int hash_table_size, int data_count, int* number_list)
{
    int* hash_table = (int*)malloc(hash_table_size * sizeof(int));
    // Valores vazios são declarados como -1, supondo que nenhum dos valores venha a ser negativo
    // Caso isso não seja o caso, define-se no topo da função a global EMPTY_VALUE
    for (int i = 0; i < hash_table_size; i++) {
        hash_table[i] = EMPTY_VALUE;
    }

    for (int i = 0; i < data_count; i++) {
        int value = number_list[i];
```

```
        int index = value % hash_table_size;

        int attempts = 0;

        while (hash_table[index] != EMPTY_VALUE) {
            index = (index + 1) % hash_table_size;
            attempts++;

            if (attempts == hash_table_size) {
                printf("[ERRO] Tabela hash cheia. Não é possível inserir o valor %d\n", value);
                break;
            }

            hash_table[index] = value;
        }
        free(hash_table);
        printLinearHashTable(hash_table, hash_table_size);

    }
}
```

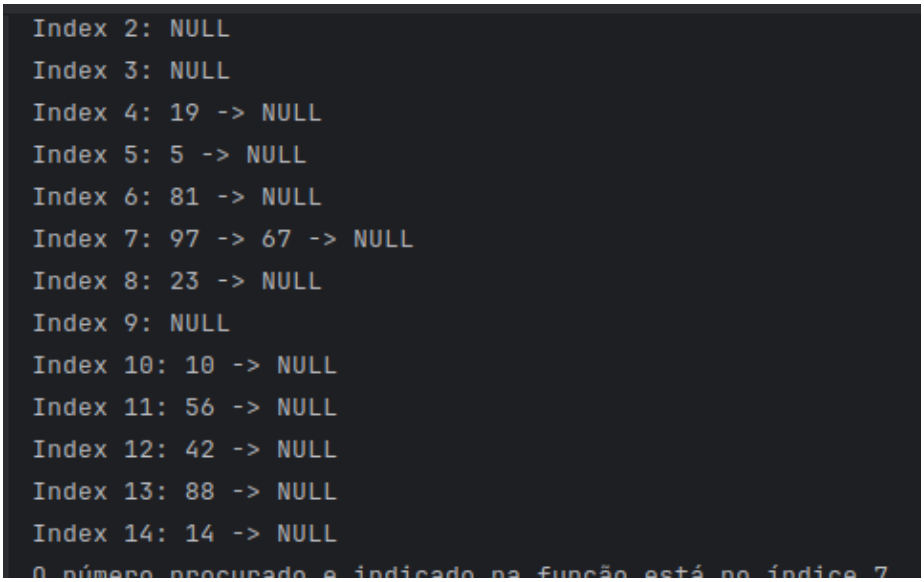
```
// Função utilizada para printar a hash table final linear
 for (int i = 0; i < size; i++) {
        if (hash_table[i] == EMPTY_VALUE) {
            printf("Index %d: NULL\n", i);
        } else {
            printf("Index %d: %d\n", i, hash_table[i]);
        }
    }
}
```

IV. Implementação da Função Main e Complexidade

Primeiro, ela chama a função makeNumList para buscar as informações necessárias, como o tipo de tabela que o usuário quer usar, o tamanho dela e a lista de números que serão guardados. Depois de pegar tudo

isso, ela usa a escolha do usuário para decidir qual função chamar. Se o tipo escolhido for 0, ela chama a função makeNodeHashTable, mas se a escolha for 1, ela chama a makeLinearHashTable. Caso o usuário digite um número que não seja nem 0 nem 1, o programa simplesmente avisa que a opção é inválida. Resumindo, a main organiza as informações e passa a tarefa de criar a tabela para a função correta.

Caso o usuário coloque o número 0, aloco a tabela hash como um array de ponteiros para Node, utilizando calloc para garantir que todos os ponteiros sejam inicializados como NULL.



I. O programa, quando executado, fica assim

Itero sobre cada número da lista (começando da posição 1, pois suponho que o índice 0 seja algum dado especial ou tamanho) e calculo o índice alvo na tabela usando o operador módulo com o tamanho da tabela hash.

Depois de popular a tabela hash, uso a função printHashTable para exibir seu conteúdo.

Por fim, realizo uma busca na tabela com o valor 97 (como exemplo, mas pode ser alterado), usando a função findInHashTable. Caso o número não seja encontrado, imprimo uma mensagem informando que ele não existe na tabela; caso contrário, informo o índice onde foi localizado.

Finalizo liberando a memória alocada para a tabela hash e retorno 0 para indicar que o programa terminou corretamente.

No caso contrário, aloca-se uma lista de números inteiros normal. A inserção ocorre a partir dos módulos, e caso o valor já exista em algum lugar, ele vai para a próxima. No caso da lista estiver cheia e não houver mais espaço para colocar números, em vez dele entrar em um loop infinito ele simplesmente printa que não há mais espaço, e dá um break.

```
int main(){

    int* number_list;
    int data_count = 0;
    int hash_table_size = 0;
    int hash_table_type;
    // A função manda os valores direto pro endereço das variáveis declaradas acima
    makeNumList(&hash_table_type, &hash_table_size, &number_list, &data_count);
    switch (hash_table_type)
    {
        case 0:
            makeNodeHashTable(hash_table_size, data_count, number_list);
        break;
        case 1:
            makeLinearHashTable(hash_table_size, data_count, number_list);
        break;
        default:
            printf("Por favor insira um tipo de tabela hash válido (entre 1 e 2).\n");

    }


    return 0;
}
```

Na criação da tabela com lista encadeada: T(n, m) = O(n + m). Para ambos os casos, a lista é O(1) na busca, com a exceção de quando houver elementos repetidos.

A partir desse ponto, torna-se O(n) No pior caso, se todas as colisões caírem na mesma posição, a complexidade de inserção seria O(n^2).

O pior caso para a criação de uma tabela de sondagem linear é O(m), sendo m o tamanho da tabela.

V. CONCLUSÃO

O programa desenvolvido demonstrou a aplicação prática de uma tabela hash para armazenamento e busca eficiente de números. O uso de listas encadeadas (nodes) para tratamento de colisões na tabela hash mostrou-se uma solução eficaz para gerenciar números repetidos. O algoritmo foi implementado conforme as instruções do trabalho.