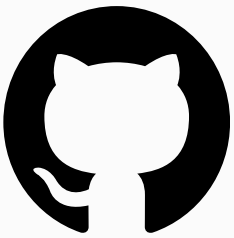
TRABALHO T2

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 PROCESS_N "C:\\Users\\neoka\\CLionProjects\\Trabalho---POD\\processo_%d.txt"
#define OUTPUT "C:\\Users\\neoka\\CLionProjects\\Trabalho---POD\\saida.txt"
#else
// Mudei o sistema de arquivos, caso não consiga ler o arquivo, por favor mude o caminho aqui
#define INPUT "Trabalho T2/dados.txt"
#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.

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 () {

// Function for reading data
int makeNumList(int *hash_size_ptr, int** num_list_ptr, int *num_count_ptr) {
    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
    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
```

```c
char number_line[1000];
if (fgets(number_line, sizeof(number_line), f) == NULL) {
    printf("[ERRO] Erro lendo o arquivo dados.txt");
    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");
    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;

    return 0;
}

}
}
```

Node

Para lidar com números repetidos na Hash Table, foi criada a seguinte estrutura, cujo propósito é armazenar uma lista de valores lineares que pode-se percorrer caso hajam valores repetidos.

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, temos duas funções:

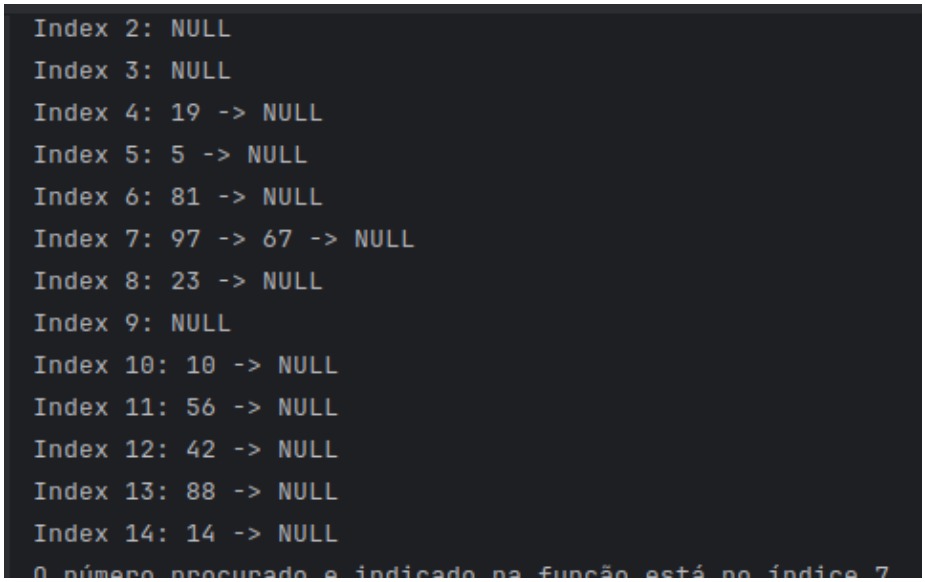
```
void printHashTable(Node** table, int size) {
    for (int i = 0; i < size; i++) {
        printf("Index %d:", i);
        Node* current = table[i];
        while (current != NULL) {
            printf(" %d ->", current->value);
            current = current->next;
        }
        printf(" NULL\n");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

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;
}
```

IV. Implementação da Função Main e Complexidade

Na função main, eu começo declarando os ponteiros e variáveis que vão armazenar a lista de números, a quantidade total de dados e o tamanho da tabela hash. Chamo a função makeNumList para ler os dados do arquivo, que retorna esses valores diretamente pelos ponteiros que forneci.

Em seguida, 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.

```
int main()
{
            // A função manda os valores direto
            // pro endereço das variáveis declaradas acima

    makeNumList(&hash_table_size, &number_list, &data_count); // O(n)
        // Forma supostamente mais segura de
        //alocar um array de pointers pra uma struct,
        //inicializa todos os pointers pra null

    Node** hash_table = (Node**)calloc(hash_table_size, sizeof(Node*)); // O(m)

    for (int i = 1; i < data_count; i++) // O(n²) no pior caso
    {
        int target_index = number_list[i] % hash_table_size; // O(1)

        if (hash_table[target_index] == NULL) // O(1)
        {
            hash_table[target_index] = (Node*)malloc(sizeof(Node)); // O(1)
            hash_table[target_index]->value = number_list[i]; // O(1)
            hash_table[target_index]->next = NULL; // O(1)
        }
        else
        {
            Node *current = hash_table[target_index]; // O(1)
            while (current->next != NULL) // O(k), k = comprimento da lista
            {
                current = current->next; // O(1)
            }
            Node* next_hash = (Node*)malloc(sizeof(Node)); // O(1)
            next_hash->value = number_list[i]; // O(1)
            next_hash->next = NULL; // O(1)
            current->next = next_hash; // O(1)
        }
    }
```

```
    printHashTable(hash_table, hash_table_size); // O(m + n)
        // Função usada pra achar números

    int number_index_if_exists = findInHashTable(97, hash_table, hash_table_size); // O(1)

    free(hash_table); // O(m)

    return 0;
}
```

T(n, m) = O(n + m)
No pior caso, se todas as colisões caírem na mesma posição,
a complexidade de inserção seria $O(n^2)$

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.