Relatório - Implementação de Tabela de Dispersão

Identificação:

Gabriel Emile Resende Abi-Abib (2220498)

Tomás Pegado Lenzi (2220711)

Introdução:

Este relatório descreve a implementação de uma tabela de dispersão (hash table) para armazenar placas de automóveis no formato "CCCNNNN". Foram realizados testes para avaliar a eficácia do sistema de hashing e tratamento de colisões, e os resultados são apresentados em detalhes.

Estrutura do programa:

O programa é composto por três arquivos principais:

1. **interface.h**: Define a estrutura da tabela de dispersão e as funções principais para a manipulação da mesma.
2. **hashtable.c**: Contém a implementação das funções definidas em interface.h.
3. **main.c**: Implementa o fluxo principal do programa, lendo placas de um arquivo, inserindo-as na hash table, buscando e, por fim, removendo-as. Além disso, são calculados os tempos de execução para essas operações.

Solução:

1. Estrutura da Hashtable

Primeiro, definimos a estrutura da hashtable. Para cada item na tabela, armazenamos a placa e um flag indicando se a posição está ocupada.

```c
void initHashTable(HashTable* table) {
    table->num_items = 0;
    for (int i = 0; i < M; i++) {
        table->items[i].isOccupied = 0;
        strcpy(table->items[i].plate, "");
    }
}
```

2. Função de Hash

A função de hash é baseada na soma dos valores ASCII dos caracteres da placa. A tentativa é adicionada para lidar com colisões.

```c
int hashFunction(const char* plate, int attempt) {
    unsigned int hashValue = 0;
    int i = 0;
    while (plate[i] != '\0') {
        hashValue = (hashValue * 31) + plate[i]; //
        i++;
    }
    return (hashValue + attempt) % M;
}
```

3. Inserção na Hashtable

A inserção utiliza a função de hash e verifica se a posição indicada está ocupada. Se estiver, fazemos uma nova tentativa até encontrar uma posição livre ou até que a tabela esteja cheia.

```c
int insertPlate(HashTable* table, const char* plate) {
    int attempt = 0;
    while (attempt < M) {
        int index = hashFunction(plate, attempt);
        if (!table->items[index].isOccupied) {
            strcpy(table->items[index].plate, plate);
            table->items[index].isOccupied = 1;
            table->num_items++;
            return attempt; // Retorna o n◆mero de ten
        }
        attempt++;
    }
    return attempt; // Hashtable cheia.
}
```

4. Busca na Hashtable

A busca verifica a posição indicada pela função de hash. Se a posição estiver ocupada e a placa corresponder à buscada, a função retorna sucesso. Caso contrário, realiza novas tentativas até encontrar a placa ou concluir que ela não está na tabela.

```c
int searchPlate(const HashTable* table, const char* plate) {
    int attempt = 0;
    while (attempt < M) {
        int index = hashFunction(plate, attempt);
        if (table->items[index].isOccupied && strcmp(table->i
            return 1; // Placa encontrada.
        }
        if (!table->items[index].isOccupied) {
            return 0; // Placa n�o est� na hashtable.
        }
        attempt++;
    }
    return 0; // Placa n�o encontrada ap�s todas tentativas.
}
```

5. Remoção na Hashtable

A remoção segue uma lógica semelhante à busca. Ela procura pela placa e, quando encontrada, a remove da tabela.

```c
int removePlate(HashTable* table, const char* plate) {
    int attempt = 0;
    while (attempt < M) {
        int index = hashFunction(plate, attempt);
        if (table->items[index].isOccupied && strcmp(ta
            table->items[index].isOccupied = 0;
            strcpy(table->items[index].plate, "");
            table->num_items--;
            return 1; // Placa removida.
        }
        attempt++;
    }
    return 0; // Placa n�o encontrada.
}
```

6. Testes e Métricas

No programa principal, realizamos testes de inclusão, busca e remoção de placas. Medimos o tempo gasto em cada operação e o número de colisões durante a inserção. Por exemplo, aqui está um exemplo dos testes:

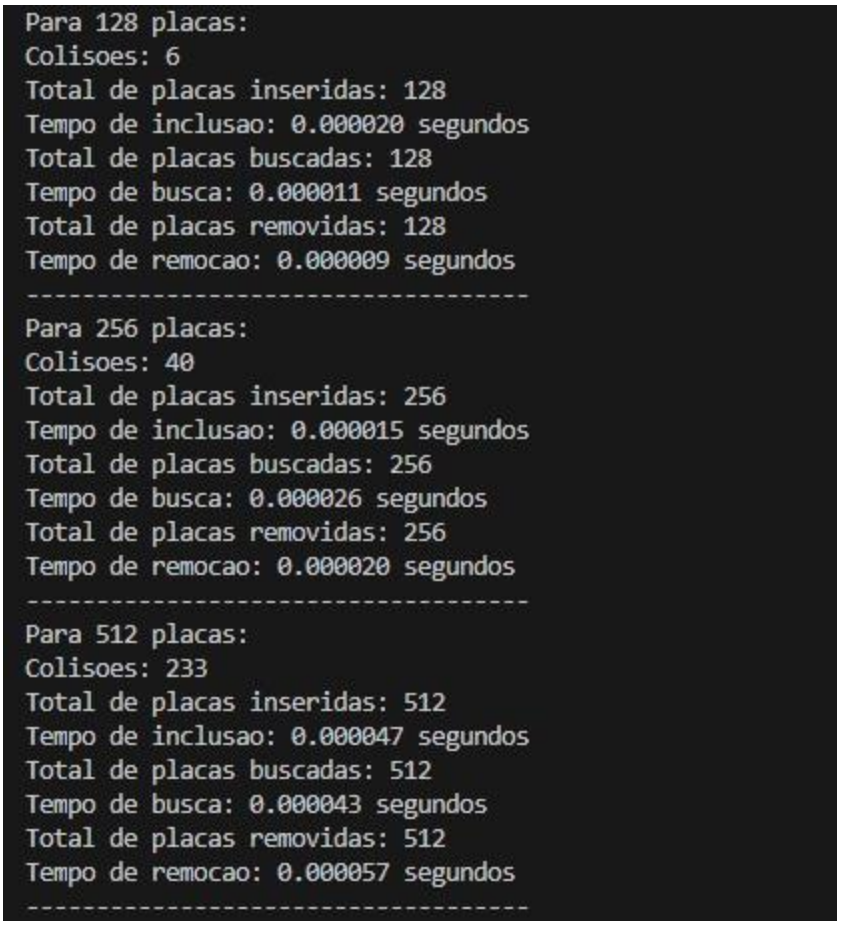
```c
int plateCounts[] = { 128, 256, 512 };
for (int k = 0; k < 3; k++) {
    int collisions = 0;
    int count = 0;

    // Reinicializa a tabela para cada novo conjunto de placas
    initHashTable(&table);

    // Medi��o do tempo de inclus�o
    start = clock();
    while (fscanf(file, "%7s< >\n", plate) == 1 && count < plateCounts[k]) {
        collisions += insertPlate(&table, plate);
        count++;
    }
    end = clock();
    totalPlatesInserted = count;
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Para %d placas: \n", plateCounts[k]);
    printf("Colis�es: %d\n", collisions);
    printf("Total de placas inseridas: %d\n", totalPlatesInserted);
    printf("Tempo de inclus�o: %f segundos\n", cpu_time_used);
```

Resultados:

Os tempos de execução e o número de colisões para diferentes números de placas inseridas foram:



Observações e conclusões:

Como observado nos resultados, à medida que o número de placas inseridas aumenta, o número de colisões também aumenta. Isso era esperado, já que com mais itens na tabela, a probabilidade de colisões também aumenta.

O tempo de execução para as operações também aumenta com o número de placas, mas não de forma proporcional, indicando que a tabela de dispersão é eficiente mesmo com um número maior de itens.

A função hash escolhida e o método de tratamento de colisões mostraram-se eficazes para o problema em questão, com tempos de execução aceitáveis e um número controlado de colisões.