Relatório do Trabalho Prático 3
AEDS II (CSI104)

Iago C. Nuvem[2], Igor Marques Passos[2],

[1]Instituto de Ciências Exatas e Aplicadas(ICEA) – UFOP
91.501-970 – João Monlevade – MG – Brazil

[2]Departamento de Computação e Sistemas
Universidade Federal de Ouro Preto – João Monlevade, MG – Brazil

18.2.8003

22.2.8118

1. Hash Simples (Hash Division)

Conforme solicitado, escolhemos o método de gerenciamento de arquivos utilizando tabelas Hash simples, com as colisões sendo tratadas por listas encadeadas, aplicados na entidade Consultas.

Para implementar as tabelas hash com encadeamento exterior criamos a estrutura Node para representar nós de uma lista encadeada, e a estrutura Bucket, que recebe nós de uma lista encadeada, representando a tabela base vista em sala, conforme abaixo:

```
// Estrutura para um no da lista encadeada
typedef struct Node {
   TConsulta consulta;
   struct Node *next;
} Node;

// Estrutura para um bucket (lista encadeada)
typedef struct {
   Node *head;
} Bucket;
```

Na implementação, criamos um Bucket com tamanho pré-definido, para facilitar a implementação da função Hash e otimizar a quantidade de colisões, utilizando o método de divisão visto em sala, a nossa função hash foi implementa conforme abaixo:

```
// Funcao de hash simples (divisao)
int funcaoHash(int id) {
   return id % TAM_TABELA;
}
```

Para otimizar as operações de Insert, criamos uma função que percorre a lista encadeada e verifica se a consulta já se encontra na lista:

```
// Verifica se a consulta ja existe na lista encadeada
int consultaExiste(int indice, int id) {
```

```
    Node *current = tabelaHash[indice].head;
    while (current != NULL) {
        if (current->consulta.id == id) {
            return 1; // Encontrado
        }
        current = current->next;
    }
    return 0; // Nao encontrado
}
```

As funções de Insert e Remoção estão descritas com comentários abaixo:

```
// Funcao para inserir uma consulta na tabela hash
void inserirConsulta(TConsulta consulta) {
    int indice = funcaoHash(consulta.id);

    // Verificar se a consulta ja existe
    if (consultaExiste(indice, consulta.id)) {
        printf("Consulta com id %d ja existe.\n", consulta.id);
        return; // No insere se j existir
    }

    Node *novoNode = (Node *)malloc(sizeof(Node));
    if (novoNode == NULL) {
        perror("Erro de alocacao de memoria");
        exit(1);
    }

    // Adiciona dados da consulta
    novoNode->consulta = consulta;

    // Referencia o cabecalho da lista ao no criado (No criado
        aponta pra cabeca da lista)
    novoNode->next = tabelaHash[indice].head;

    // Atualiza o cabecalho da lista com o no criado (Insere no
        inicio)
    tabelaHash[indice].head = novoNode;
}

// Funcao para remover uma consulta da tabela hash
void removerConsulta(int id) {
    int indice = funcaoHash(id);

    // Inicializa o ponteiro 'atual' para percorrer a lista e
        'anterior' como NULL
    Node *atual = tabelaHash[indice].head;
    Node *anterior = NULL;
```

```
    // Percorre a lista ate encontrar a consulta com o id
       fornecido ou chegar ao final da lista
    while (atual != NULL && atual->consulta.id != id) {
       anterior = atual; // Mantem o no anterior
       atual = atual->next; // Vai para o proximo no
    }

    // Verifica se a consulta nao foi encontrada na lista
    if (atual == NULL) {
       // Consulta nao encontrada
       return;
    }

    // Se 'anterior' ainda e NULL, significa que a consulta a
       ser removida e a primeira da lista
    if (anterior == NULL) {
       // Atualiza o cabecalho da lista para o proximo elemento,
          removendo o primeiro
       tabelaHash[indice].head = atual->next;
    } else {
       // Remove a consulta da lista ligando o no anterior ao
          proximo do no 'atual'
       anterior->next = atual->next;
    }

    // Libera memoria alocada para atual
    free(atual);
}
```

2. Testes Realizados

Realizamos testes inserindo 5000, 10000 e 10000 registros na tabela Hash, e os resultados em termos de tempo de execução foram registrados em um arquivo de log descrito abaixo:

```
Tabela Hash - Insercao
Tempo de execucao: 0.001328 segundos

Tamanho da base: 5000
==========================================

Tabela Hash - Busca
Tempo de execucao: 0.000002 segundos

Tamanho da base: 5000
==========================================

Tabela Hash - Remocao
Tempo de execucao: 0.000002 segundos
```

```
Tamanho da base: 5000
==========================================

Tabela Hash - Insercao
Tempo de execucao: 0.002694 segundos

Tamanho da base: 10000
==========================================

Tabela Hash - Busca
Tempo de execucao: 0.000004 segundos

Tamanho da base: 10000
==========================================

Tabela Hash - Remocao
Tempo de execucao: 0.000002 segundos

Tamanho da base: 10000
==========================================

Tabela Hash - Insercao
Tempo de execucao: 0.023023 segundos

Tamanho da base: 100000
==========================================

Tabela Hash - Busca
Tempo de execucao: 0.000832 segundos

Tamanho da base: 100000
==========================================

Tabela Hash - Remocao
Tempo de execucao: 0.000020 segundos

Tamanho da base: 100000
==========================================
```

---

3. Analise de Resultados

Conforme visto nos conceitos em sala de aula, e comprovado pelos testes realizado, a utilização de tabelas hash permite manipular grandes quantidades de dados sem grandes aumentos em tempo de processamento, apesar de ainda haver um aumento linear, conforme figura X, ao observarmos o vetor tempo, podemos perceber que a diferença não é significante quando comparada à quantidade de dados. Podemos perceber também ao analisarmos o vetor tempo, que a operação mais demorada é a de remoção.

Figure 1. Análise de tempo de processamento de Operações sobre tabela Hash