Relatório HashTable

O projeto a seguir, desenvolvido na linguagem Java, possui como objetivo principal desenvolver uma estrutura de dados do tipo tabela hash, implementando diferentes tratamentos de colisões entre os índices, seguindo os padrões de Programação Orientada a Objetos.

Objetivos específicos:

- Implementar uma tabela hash simples em Java.
- Utilizar duas funções de hash diferentes.
- Inserir elementos em uma tabela de tamanho fixo utilizando sondagem linear para tratar colisões.
- Realizar buscas nas tabelas implementadas.
- Compreender o comportamento e as limitações das tabelas hash.

A arquitetura do código se divide em classes, sendo uma delas a “HashTable” como a abstrata, que possui todos os métodos necessários para a execução do código. Ela possui 2 classes estendidas, a “HashEncadeado1” e a “HashEncadeado2”, que possuem cada uma um tratamento de colisões diferentes.

Hash Table:

Atributos:

- capacidade: tamanho do vetor.
- tamanho: quantidade atual de elementos.
- colisões: número de colisões ocorridas.

Métodos abstratos:

- inserir(String chave)
- buscar(String chave)

Métodos utilitários:

- getCapacidade(), getTamanho(), getColisoes()
- imprimirTabela()
- calcularIndice(String chave)

Hash Encadeado:

Implementa a tabela hash utilizando listas encadeadas para resolver colisões:

- Cada posição do vetor armazena uma lista ligada (Node) com as chaves que colidem.
- Quando ocorre uma colisão, a nova chave é inserida no início da lista.
- Permite armazenar múltiplas chaves na mesma posição.
- Realiza a sobrescrita da função hash, podendo variar a forma de gerar índices.

Cálculo Hash Encadeado 1:

```java
@Override
protected int calcularIndice(String chave) {
    long hash = 5381;
    for (int i = 0; i < chave.length(); i++) {
        hash = ((hash << 5) + hash) + chave.charAt(i);
    }
    return (int) Math.abs(hash % capacidade);
}
```

Resultados função HashEncadeado1:

Colisões: **4969**

Tempo de inserção e busca: **4.9413 ms**

Colisões por indice:

- [0] tem 147

- [1] tem 160
- [2] tem 160
- [3] tem 177
- [4] tem 148
- [5] tem 160
- [6] tem 171
- [7] tem 159
- [8] tem 146
- [9] tem 150
- [10] tem 149
- [11] tem 167
- [12] tem 157
- [13] tem 165
- [14] tem 147
- [15] tem 157
- [16] tem 149
- [17] tem 174
- [18] tem 149
- [19] tem 181
- [20] tem 150
- [21] tem 139
- [22] tem 146
- [23] tem 162
- [24] tem 160
- [25] tem 150
- [26] tem 152
- [27] tem 158
- [28] tem 159
- [29] tem 147
- [30] tem 148
- [31] tem 157

Cálculo Hash Encadeado 2:

```java
@Override
protected int calcularIndice(String chave) {
        int hash = 7;
        for (char c : chave.toCharArray()) {
            hash = hash +- c;
        }
        return Math.abs(hash) % capacidade;
}
```

Resultados função HashEncadeado2:

Colisões: **4969**

Tempo de inserção e busca: **4.5174 ms**

Colisões por indice:

- [0] tem 166
- [1] tem 137
- [2] tem 151
- [3] tem 161
- [4] tem 158
- [5] tem 164
- [6] tem 140
- [7] tem 164
- [8] tem 134
- [9] tem 159
- [10] tem 157
- [11] tem 166
- [12] tem 153
- [13] tem 122
- [14] tem 132
- [15] tem 193
- [16] tem 158
- [17] tem 173
- [18] tem 161

- [19] tem 169
- [20] tem 166
- [21] tem 164
- [22] tem 152
- [23] tem 162
- [24] tem 140
- [25] tem 158
- [26] tem 162
- [27] tem 167
- [28] tem 160
- [29] tem 133
- [30] tem 148
- [31] tem 171

Conclusão:

A atividade proporcionou uma compreensão prática do funcionamento das tabelas hash, destacando especialmente o impacto das funções de dispersão e das estratégias de tratamento de colisões. Por meio da implementação em Java, foi possível perceber que, embora a estrutura seja conceitualmente simples, detalhes como a escolha da função hash e o método de resolução de colisões são fundamentais para garantir a eficiência da tabela. As abordagens implementadas, como o encadeamento separado e o endereçamento aberto com áreas distintas para inserção e colisão, mostraram-se eficazes ao lidar com colisões, permitindo uma comparação clara entre diferentes formas de organização e desempenho da estrutura.