# Algoritmos e Estruturas de Dados II

Capítulo da Aula 4: Tabelas Hash e Funções de Dispersão

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Fevereiro de 2026

## 1. O Problema da Busca

Imagine que você quer verificar se um CPF está em uma base de dados.

- Em um vetor desordenado: $O(N)$.
- Em um vetor ordenado (Busca Binária): $O(\log N)$.
- **Em uma Tabela Hash:** $O(1)$ (no caso médio).

As Tabelas Hash (ou de Dispersão) buscam o "Santo Graal" das estruturas de dados: operações de busca, inserção e remoção em tempo constante.

## 2. Conceito Fundamental

A ideia é usar uma **Função de Hash** ($h(x)$) que mapeia uma chave de busca (string, int grande, objeto) para um índice de vetor pequeno $[0, M-1]$.

$$h(\text{chave}) \rightarrow \text{índice}$$

### Propriedades de uma Boa Função Hash

1. **Determinística:** Se $x = y$, então $h(x)$ deve ser igual a $h(y)$. 2. **Rápida:** Deve ser calculada em $O(1)$ (ou proporcional ao tamanho da chave). 3. **Distribuição Uniforme:** Deve espalhar as chaves uniformemente pelo vetor para minimizar colisões.

**Exemplo de Hash Polinomial para Strings:** Para uma string $S = c_0 c_1 ... c_k$, uma função comum é:

$$h(S) = (c_0 \cdot P^0 + c_1 \cdot P^1 + \cdots + c_k \cdot P^k) \mod M$$

Onde $P$ é um primo (ex: 31 ou 53) e $M$ é o tamanho da tabela.

## 3. Tratamento de Colisões

Pelo **Princípio da Casa dos Pombos**, se temos mais chaves possíveis do que índices na tabela, colisões são inevitáveis. Ou seja, $h(k1) == h(k2)$ para $k1 \neq k2$.

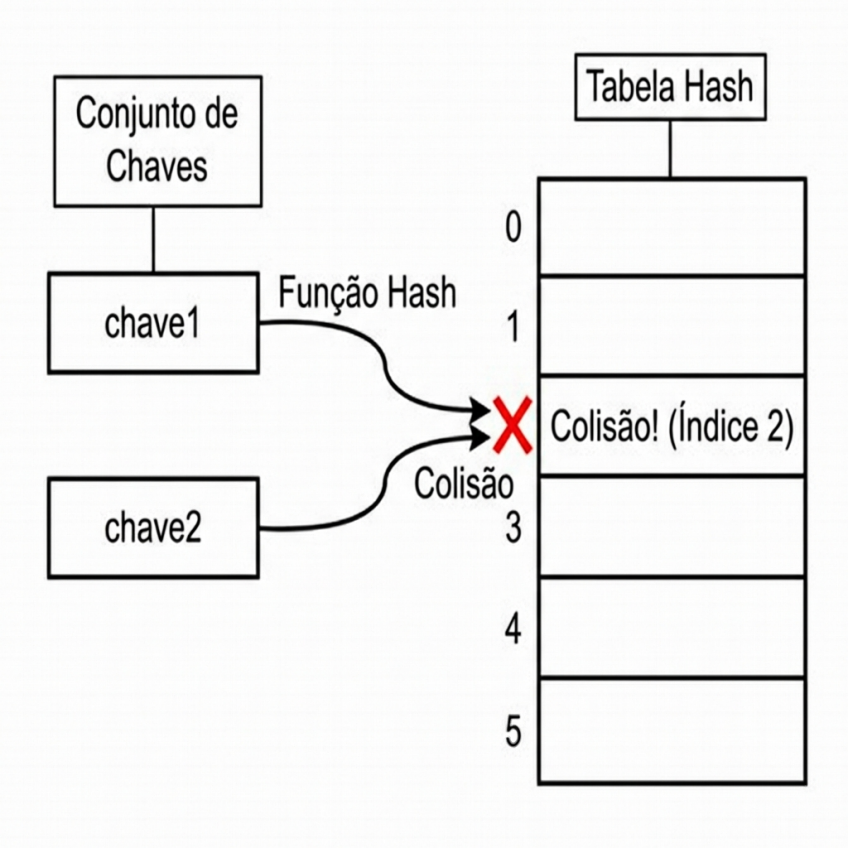


Figure 1: Função de hash mapeia chave (ex.: CPF, string) para um índice do vetor.

### 3.1. Encadeamento Exterior (Separate Chaining)

Cada posição da tabela aponta para uma lista encadeada (ou vetor, ou árvore) contendo todos os elementos que colidiram ali.

- **Vantagem:** Simples de implementar; a tabela "nunca enche" (apenas as listas ficam longas).
- **Desvantagem:** Uso extra de memória (ponteiros); perde localidade de cache.
- **Java:** O `HashMap` do Java usa essa técnica. Se a lista ficar muito grande (limit $> 8$), ele converte a lista em uma Árvore Rubro-Negra para garantir busca $O(\log N)$ no pior caso.

### 3.2. Endereçamento Aberto (Open Addressing)

Se ocorrer colisão, procuramos outro lugar *livre* dentro da própria tabela seguindo uma regra de sondagem.

- **Sondagem Linear:** Tenta $index + 1, index + 2...$
- **Vantagem:** Economia de memória (sem ponteiros extra); excelente para cache.
- **Desvantagem:** Sofre com **Clusterização Primária** (aglomerados de ocupados); performance degrada drasticamente se a tabela estiver quase cheia.

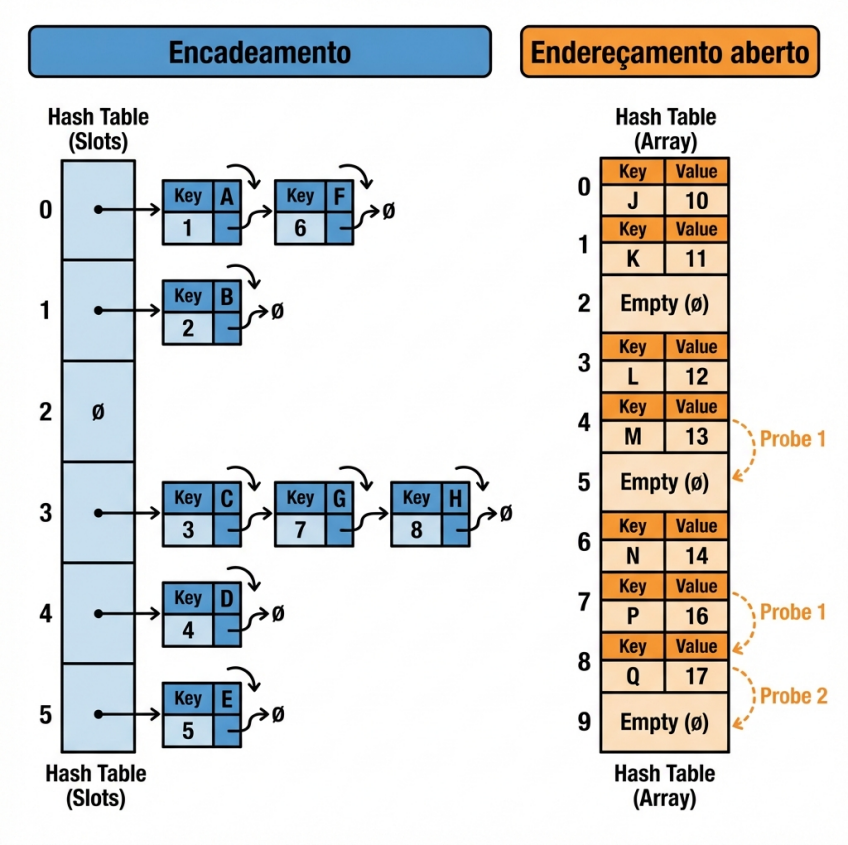


Figure 2: Tratamento de colisões: encadeamento (lista em cada célula) vs endereçamento aberto (sondagem).

## 4. Complexidade e Fator de Carga

O desempenho depende do **Fator de Carga** ($\alpha$):

$$\alpha = \frac{N}{M}$$

Onde $N$ é o número de elementos inseridos e $M$ é o tamanho da tabela.

| Operação | Caso Médio (Boa Hash) | Pior Caso (Tudo colide) | | :— | :— | :— | | Busca / Inserção | $O(1)$ | $O(N)$ (lista longa) |

Para garantir $O(1)$, as implementações dobram o tamanho da tabela ($M$) e remapeiam tudo (**Rehash**) quando $\alpha$ passa de um limiar (no Java, 0.75).

## 5. Prática em Java (‘HashSet‘ vs ‘HashMap‘)

### Interfaces Importantes

- `Set` **(Conjunto):** Guarda apenas chaves únicas. Não permite duplicatas.
- `Map` **(Dicionário/Mapa):** Guarda pares **Chave** $\rightarrow$ **Valor**.

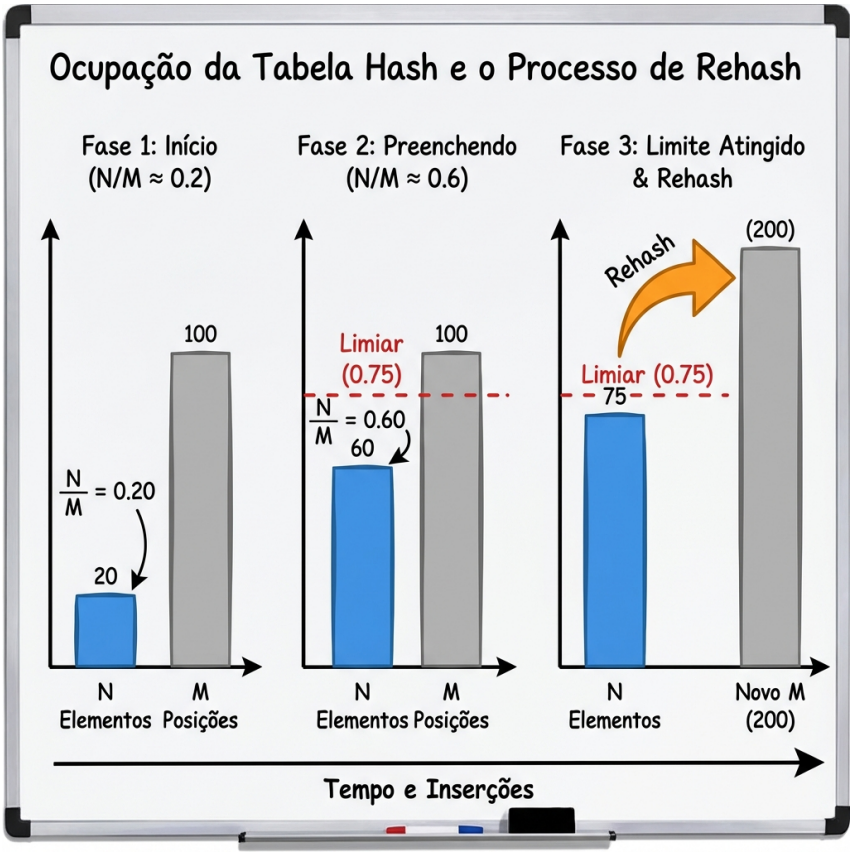


Figure 3: Fator de carga: razão entre número de elementos e tamanho da tabela; rehash quando passa do limiar.

### ‘HashMap‘ / ‘HashSet‘

- **Ordem:** Não garante nenhuma ordem dos elementos.
- **Complexidade:** $O(1)$ para `put`, `get`, `contains`.
- **Requisito:** As chaves devem implementar `equals()` e `hashCode()` corretamente.

### ‘TreeMap‘ / ‘TreeSet‘

- **Ordem:** Mantém os elementos ordenados (naturalmente ou via `Comparator`).
- **Implementação:** Árvore Rubro-Negra (Red-Black Tree).
- **Complexidade:** $O(\log N)$.
- **Uso:** Quando você precisa processar dados em ordem ou buscar faixas de valores (`subSet`, `ceiling`, `floor`).

### O Contrato ‘hashCode‘ e ‘equals‘

Se você colocar objetos próprios (ex: classe `Pessoa`) no HashMap: 1. Se dois objetos são iguais (`equals` retorna `true`), eles **DEVEM** ter o mesmo `hashCode`. 2. O inverso não é

obrigatório (colisão).

```java
import java.util.HashMap;
import java.util.Map;

public class ExemploMap {
    public static void main(String[] args) {
        // Mapa de CPF (String) para Nome (String)
        Map<String, String> cadastro = new HashMap<>();

        cadastro.put("111.222.333-44", "João Silva");
        cadastro.put("555.666.777-88", "Maria Souza");

        // Busca O(1)
        if (cadastro.containsKey("111.222.333-44")) {
            System.out.println("Encontrado: " + cadastro.get("
               111.222.333-44"));
        }

        // Iterar sobre Chaves
        for (String cpf : cadastro.keySet()) {
            System.out.println(cpf + " -> " + cadastro.get(cpf)
               );
        }
    }
}
```

## 6. Exercício Resolvido: Two Sum

**Problema:** Dado um array de inteiros e um alvo `target`, retorne os índices de dois números que somam `target`.

**Solução Ingênua** $O(N^2)$**:** Dois loops for. **Solução com Hash** $O(N)$**:**

```java
public int[] twoSum(int[] nums, int target) {
    // Mapa: Valor -> Índice
    Map<Integer, Integer> map = new HashMap<>();

    for (int i = 0; i < nums.length; i++) {
        int complemento = target - nums[i];

        if (map.containsKey(complemento)) {
            return new int[] { map.get(complemento), i };
        }

        map.put(nums[i], i);
    }
    return new int[]{}; // Não encontrou
}
```

**Análise:** Percorremos o array uma vez ($N$). Cada operação de mapa é $O(1)$. Total: $O(N)$.

# 7. Exercícios

## 7.1. Exercícios Conceituais

1 Explique a diferença entre encadeamento exterior (separate chaining) e endereçamento aberto (open addressing) no tratamento de colisões. Discuta vantagens e desvantagens de cada abordagem em termos de uso de memória, localidade de cache e pior caso.

2 Defina fator de carga $\alpha$ de uma tabela hash. Explique por que, na prática, mantemos $\alpha$ limitado (por exemplo, $\alpha \leq 0{,}75$) e qual o impacto disso em tempo de redimensionamento (rehash).

3 Descreva o contrato entre `hashCode()` e `equals()` em Java. Dê um exemplo de classe que viola esse contrato e explique os problemas que isso causa em um `HashMap`.

## 7.2. Exercícios Analíticos

1 Considere uma tabela hash com encadeamento e fator de carga $\alpha = N/M$. Mostre, de forma qualitativa, que o tamanho médio das listas encadeadas tende a $\alpha$ e discuta o impacto disso na busca e inserção.

2 Analise o fenômeno de clusterização primária em sondagem linear. Para uma tabela quase cheia, descreva como o tamanho médio de um cluster afeta o tempo de busca e inserção.

3 Suponha uma função de hash polinomial para strings com base $P = 31$ e módulo $M = 10^9 + 7$. Discuta as vantagens de usar módulo primo grande em relação a um módulo pequeno ou não primo.

## 7.3. Exercícios de Programação

1 Implemente sua própria tabela hash com encadeamento exterior em C/C++/Java, suportando operações básicas de `insert`, `find` e `erase`. Meça o tempo dessas operações para diferentes fatores de carga (*load factors*) e compare com a implementação padrão da linguagem (`unordered_map`, `HashMap`, etc.).

2 Implemente uma tabela hash com endereçamento aberto e sondagem linear. Experimente também sondagem quadrática e hashing duplo. Compare o tempo de busca/inserção/remoção em diferentes cenários de carga.

3 Resolva problemas de juiz online clássicos envolvendo hash, como:

- **LeetCode 1:** Two Sum.
- **LeetCode 49:** Group Anagrams (use hash de contagem de letras ou string ordenada como chave).
- **Beecrowd 1256:** Tabelas Hash (implementação manual do encadeamento).

Para pelo menos um dos problemas, escreva um parágrafo justificando formalmente a complexidade $O(N)$ esperada.