

# Estruturas de Dados

## Aula 13 – Tabelas Hash

---

Prof. Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação

Instituto Federal da Bahia – Campus Feira de Santana

2026

#

Motivação

Definição

Funções de Hash

Colisões

Implementação

Exercício

# Motivação

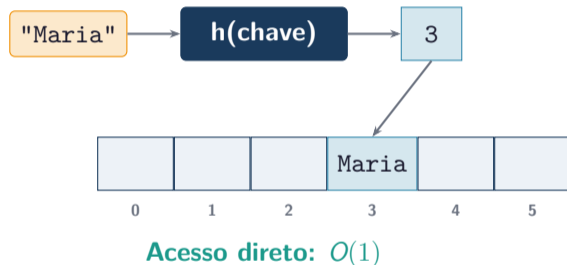
Por que precisamos de Tabelas Hash?

## ▪ Limitações das estruturas anteriores

Em **listas** e **vetores** não ordenados a busca é  $O(N)$ . Em **ABBs** balanceadas (AVL) a busca é  $O(\log N)$ . Mas e se quiséssemos tempo **constante**  $O(1)$ ?

## ▪ Solução: Tabelas Hash

Estrutura que associa uma **chave** a uma **posição** no vetor por meio de uma **função de hash**, permitindo busca, inserção e remoção em  $O(1)$  no caso médio.



| Estrutura                  | Busca                    | Inserção                 | Remoção                  |
|----------------------------|--------------------------|--------------------------|--------------------------|
| Vetor não ordenado         | $O(N)$                   | $O(1)$                   | $O(N)$                   |
| Vetor ordenado             | $O(\log N)$              | $O(N)$                   | $O(N)$                   |
| Lista encadeada            | $O(N)$                   | $O(1)$                   | $O(N)$                   |
| ABB (pior caso)            | $O(N)$                   | $O(N)$                   | $O(N)$                   |
| AVL / Rubro-negra          | $O(\log N)$              | $O(\log N)$              | $O(\log N)$              |
| <b>Tabela Hash (médio)</b> | <b><math>O(1)</math></b> | <b><math>O(1)</math></b> | <b><math>O(1)</math></b> |
| Tabela Hash (pior)         | $O(N)$                   | $O(N)$                   | $O(N)$                   |

## ▪ Atenção

No **pior caso** (muitas colisões), a tabela hash degrada para  $O(N)$ . Por isso a escolha da **função de hash** e do **tratamento de colisões** é crucial.

# Tabelas Hash

Conceitos fundamentais

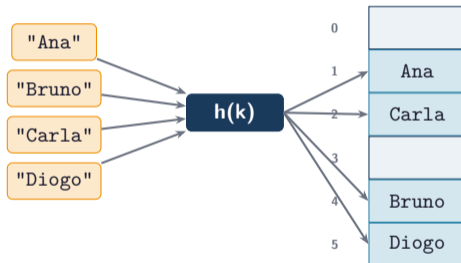
# O que é uma Tabela Hash?

## ▪ Definição

Estrutura de dados que armazena pares (**chave, valor**) em um vetor de tamanho  $M$ , usando uma **função de hash**  $h(k)$  que mapeia cada chave  $k$  a um índice  $i \in \{0, 1, \dots, M-1\}$ .

## ▪ Componentes principais

- **Universo**  $U$ : conjunto de todas as chaves possíveis
- **Tabela**  $T$ : vetor com  $M$  posições (*buckets*)
- **Função de hash**  $h : U \rightarrow \{0, \dots, M-1\}$
- **Tratamento de colisões**: o que fazer quando  $h(k_1) = h(k_2)$



## ▪ Função de hash ideal

- **Determinística:** mesma chave  $\rightarrow$  mesmo índice
- **Rápida** de calcular:  $O(1)$
- **Uniforme:** distribui as chaves uniformemente sobre  $\{0, \dots, M - 1\}$
- **Minimiza colisões**

## ▪ Observação importante

Como  $|U| \gg M$  (universo bem maior que a tabela), **colisões são inevitáveis** pelo *Princípio da Casa dos Pombos*. O segredo está em **tratá-las** bem.

## ▪ Fator de carga

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

onde  $n$  é o número de elementos e  $M$  é o tamanho da tabela.

- $\alpha$  pequeno  $\Rightarrow$  poucas colisões, espaço desperdiçado
- $\alpha$  grande  $\Rightarrow$  muitas colisões, busca degrada
- Recomendado:  $\alpha \leq 0,75$

# Funções de Hash

Como mapear chaves em índices

## ▪ Definição

A função mais simples: o índice é o **resto da divisão** da chave por  $M$ .

$$h(k) = k \bmod M$$

## ▪ Recomendações

- Escolher  $M$  **primo** e distante de potências de 2
- Evitar  $M = 2^p$  (usa só os  $p$  últimos bits)
- Cálculo extremamente rápido

## ▪ Exemplo: $M = 7$

| $k$ | $h(k) = k \bmod 7$ | posição         |
|-----|--------------------|-----------------|
| 15  | 1                  | T[1]            |
| 22  | 1                  | T[1] (colisão!) |
| 31  | 3                  | T[3]            |
| 44  | 2                  | T[2]            |
| 50  | 1                  | T[1] (colisão!) |

## ▪ Definição

Multiplica a chave por uma constante  $A$  ( $0 < A < 1$ ) e usa a parte fracionária multiplicada por  $M$ :

$$h(k) = \lfloor M \cdot (k \cdot A \bmod 1) \rfloor$$

## ▪ Vantagens

- $M$  pode ser qualquer valor (ex.: potência de 2)
- $A = (\sqrt{5} - 1)/2 \approx 0,618$  (Knuth)
- Distribui melhor que o método da divisão para certas chaves

## ▪ Outras estratégias

**Dobra (folding):** divide a chave em partes e soma

**Meio do quadrado:** eleva  $k$  ao quadrado e usa os dígitos centrais

**Hash de strings:** combina os códigos ASCII dos caracteres

## ▪ Hash para strings (Java-like)

```
h = 0;  
para cada char c em s:  
    h = 31 * h + c;  
retorne h % M;
```

# Tratamento de Colisões

Encadeamento e Endereçamento Aberto

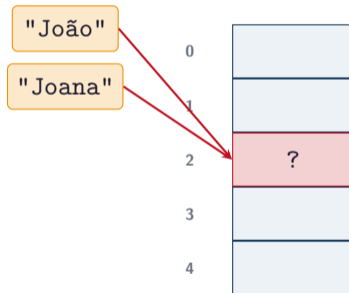
## ▪ Definição

Ocorre uma **colisão** quando duas chaves diferentes  $k_1 \neq k_2$  produzem o mesmo índice:  $h(k_1) = h(k_2)$ .

## ▪ Estratégias principais

- 1 **Encadeamento separado** (*separate chaining*): cada posição armazena uma lista
- 2 **Endereçamento aberto** (*open addressing*): procura outra posição vazia na própria tabela
  - Sondagem linear
  - Sondagem quadrática
  - Hash duplo

**Colisão:**  $h(k_1) = h(k_2) = 2$



# Encadeamento Separado

## ▪ Como funciona

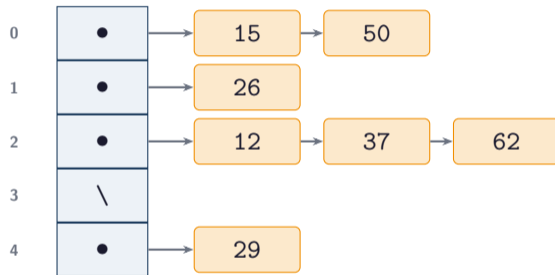
Cada posição  $T[i]$  guarda o ponteiro para uma **lista encadeada** com todas as chaves cujo hash é  $i$ .

## ▪ Operações

- **Inserir:** insere no início da lista de  $T[h(k)]$
- **Buscar:** percorre a lista de  $T[h(k)]$
- **Remover:** encontra e remove da lista

## ▪ Custo

Médio:  $O(1 + \alpha)$ . Tabela nunca “enche”.



$M = 5$ , função:  $h(k) = k \bmod 5$

## ▪ Como funciona

Se  $T[h(k)]$  está ocupada, tenta a **próxima posição**, e a próxima, e assim por diante (de forma circular).

$$h_i(k) = (h(k) + i) \bmod M, \quad i = 0, 1, 2, \dots$$

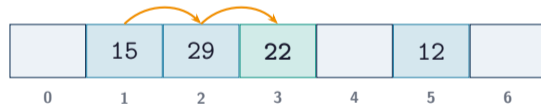
## ▪ Problema: agrupamento primário

Tende a formar **grandes blocos contíguos** de posições ocupadas, aumentando o tempo de busca.

## ▪ Importante

Para remover, é preciso usar uma **marca de removido** (*tombstone*) ou rehash dos elementos do cluster.

**Inserindo 22 com  $h(k) = k \bmod 7$**   
 $h(22) = 1$ , mas  $T[1]$  e  $T[2]$  ocupados



Inserido em  $T[3]$  após 2 sondagens

## ▪ Sondagem quadrática

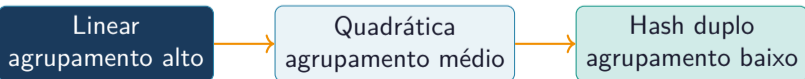
$$h_i(k) = (h(k) + c_1 i + c_2 i^2) \bmod M$$

- Reduz o agrupamento primário
- Pode ainda ter **agrupamento secundário**: chaves com mesmo  $h(k)$  seguem a mesma sequência
- Requer  $M$  primo e  $\alpha < 0,5$  para garantir varredura completa

## ▪ Hash duplo

$$h_i(k) = (h_1(k) + i \cdot h_2(k)) \bmod M$$

- Usa **duas funções** de hash
- $h_2(k)$  deve ser coprimo com  $M$
- Distribui melhor: cada chave segue uma sequência diferente
- Considerada uma das melhores estratégias



| Critério                | Encadeamento             | End. Aberto            |
|-------------------------|--------------------------|------------------------|
| Estrutura externa       | lista encadeada          | nenhuma (in-place)     |
| Memória                 | ponteiros extras         | somente o vetor        |
| Fator de carga $\alpha$ | pode ser $> 1$           | deve ser $< 1$         |
| Remoção                 | simples                  | exige <i>tombstone</i> |
| Cache                   | pior (saltos de memória) | melhor (localidade)    |
| Tabela cheia            | não acontece             | precisa redimensionar  |

## ▪ Rehashing

Quando  $\alpha$  ultrapassa o limite (tipicamente 0,75), cria-se uma **nova tabela** com tamanho  $\approx 2M$  (próximo primo) e **reinsere-se** todos os elementos. O custo amortizado de inserção permanece em  $O(1)$ .

# Implementação

Estrutura em C

## ▪ Definição da tabela

```
#define M 101 // primo
typedef struct No {
    int chave;
    int valor;
    struct No *prox;
} No;

typedef struct {
    No *tabela[M];
    int n;
} Hash;
```

## ▪ Função de hash

```
int h(int chave) {
    return chave % M;
}
```

## ▪ Inserção

```
void inserir(Hash *H, int k, int v)
{
    int i = h(k);
    No *novo = malloc(sizeof(No));
    novo->chave = k;
    novo->valor = v;
    novo->prox = H->tabela[i];
    H->tabela[i] = novo;
    H->n++;
}
```

# Exercício

Implementação da Tabela Hash

## ▪ Implemente uma Tabela Hash com o seguinte menu

- 1 **Inserir** um par (chave, valor) na tabela
- 2 **Buscar** um valor a partir da chave
- 3 **Remover** um par a partir da chave
- 4 **Listar** todos os pares armazenados
- 5 **Mostrar** o estado de cada *bucket* (índice  $\rightarrow$  lista)
- 6 Exibir **estatísticas**:  $n$ ,  $M$ ,  $\alpha$ , maior lista, número de colisões
- 7 **Redimensionar** (rehash) quando  $\alpha > 0,75$
- 8 Fechar

## ▪ Atenção

Implemente **duas versões**: uma com **encadeamento separado** e outra com **endereçamento aberto** (sondagem linear). Compare o número de colisões para o mesmo conjunto de chaves.