

# Tabelas Hash

Prof. Denio Duarte

[duarte@uffs.edu.br](mailto:duarte@uffs.edu.br)

Prof. Claunir Pavan

[claunir.pavan@uffs.edu.br](mailto:claunir.pavan@uffs.edu.br)

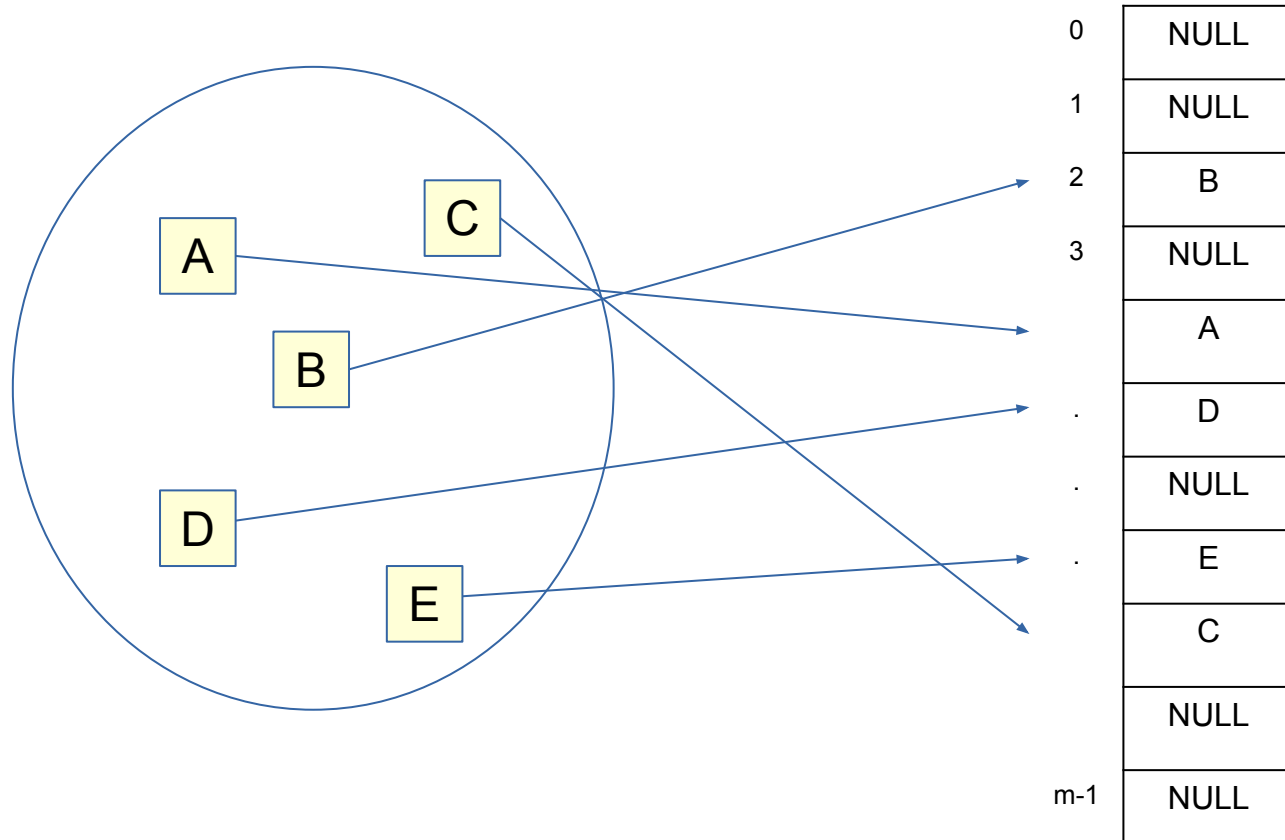
# Tabelas Hash

- Diversos métodos de busca vistos funcionam através de comparações de chaves.
  - Busca binária: requer que os dados estejam ordenados
    - Depende da ordenação dos elementos, e isso tem um custo caso seja necessário fazer a ordenação
    - Custo da busca, no pior caso, é de  $\log_2 n$  operações.
- Tabelas Hash: permitem acesso direto ao elemento procurado, sem comparações de chaves e sem necessidade de ordenação

# Tabelas Hash

- Tabela de **Dispersão** ou Tabela de **Espalhamento**
  - Estrutura de dados capaz de armazenar pares chave-valor (**key, value**)
    - **Chave**: parte da informação que compõe o elemento a ser armazenado
    - **Valor**: posição ou índice onde o elemento se encontra no array que representa a tabela
- Suporta as mesmas operações que as listas sequenciais (inserção, remoção, busca), porém, de forma mais eficiente.
- Utiliza uma **função** para espalhar os dados na tabela
  - Função será utilizada em todas as operações
- Elementos ficam dispostos de forma **não ordenada**

# Tabelas Hash



# Tabelas Hash

- A implementação de uma tabela hash considera o mapeamento do conjunto de  $N$  chaves em um **vetor** de tamanho  $M > N$ .
  - Cada posição do vetor é também chamada de *bucket* ou *slot*.
- Função de hash  $\rightarrow$  a partir da chave a ser inserida, transforma este valor em um inteiro equivalente a um dos índices do vetor.
- Usamos então este índice para armazenar a chave e o valor no vetor.

# Função de hash

- A função de hash executa a transformação do valor de uma chave em um índice de vetor, por meio da aplicação de operações aritméticas e/ou lógicas.
- Os valores das chaves podem ser numéricos, alfabéticos ou alfanuméricos (a função irá converter o que não é número).
- Portanto, cada chave deve ser mapeada para um inteiro entre 0 e  $M-1$  (para uso como índice do vetor de  $M$  posições).



# Função de hash

- Exemplo
  - Armazenar um conjunto de números em um vetor com 10 posições (0 a 9)
    - Qual função podemos utilizar como função de hash?

# Função de hash

- Exemplo
  - Armazenar um conjunto de números em um vetor de 10 posições (0 a 9)
    - Vamos aplicar a função: a posição de armazenamento é o valor resultante do resto do valor chave dividido por 10, ou seja,  $\text{key \% 10}$

```
int hashFunction(int k) {  
    return k%10;  
}
```



# Função de hash

- Exemplo
  - Armazenar um conjunto de números em um vetor de 10 posições
  - Inserir **4**

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

# Função de hash

- Exemplo

- Armazenar um conjunto de números em um vetor de 10 posições

- Inserir 4

- **HashFunction(4)**



0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

# Função de hash


- Exemplo

- Armazenar um conjunto de número em um vetor de 10 posições

- Inserir 4

- **HashFunction(4)**

- `vetor[HashFunction(4)] = 4;`



0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

# Função de hash

- Exemplo

- Armazenar um conjunto de número em um vetor de 10 posições
- Inserir 4
  - **HashFunction(4)**
  - `vetor[HashFunction(4)] = 4;`



0	
1	
2	
3	
4	4
5	
6	
7	
8	
9	

# Função de hash

- Exemplo
  - Armazenar um conjunto de número em um vetor de 10 posições
  - Inserir **17**

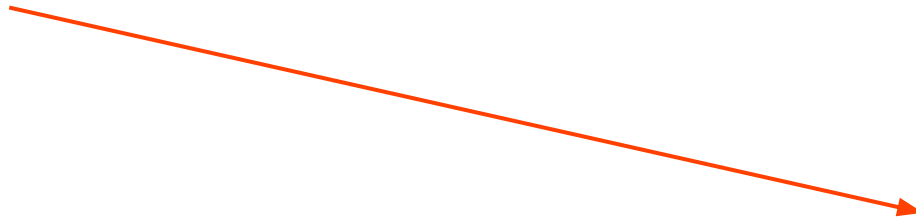
0	
1	
2	
3	
4	4
5	
6	
7	
8	
9	

# Função de hash

- Exemplo

- Armazenar um conjunto de número em um vetor de 11 posições
- Inserir **17**
  - **HashFunction(17)**

0	
1	
2	
3	
4	4
5	
6	
7	
8	
9	



# Função de hash

- Exemplo

- Armazenar um conjunto de número em um vetor de 11 posições
- Inserir **17**
  - **HashFunction(17)**
  - `vetor[HashFunction(17)] = 17;`



0	
1	
2	
3	
4	4
5	
6	
7	17
8	
9	

# Função de hash

- Complexidade?
  - $O(1)$ 
    - Melhor Impossível :)
  - Não existe almoço grátis (free lunch)



# Função de hash

- Exemplo
  - Armazenar um conjunto de número em um vetor de 10 posições
  - Inserir **104**

0	
1	
2	
3	
4	4
5	
6	
7	17
8	
9	

# Função de hash

- Exemplo

- Armazenar um conjunto de número em um vetor de 10 posições

- Inserir **104**

- **HashFunction(104)**



0	
1	
2	
3	
4	4
5	
6	
7	17
8	
9	

# Função de hash

- Exemplo

- Armazenar um conjunto de número em um vetor de 10 posições

- Inserir **104**

- HashFunction(104)



0	
1	
2	
3	
4	4

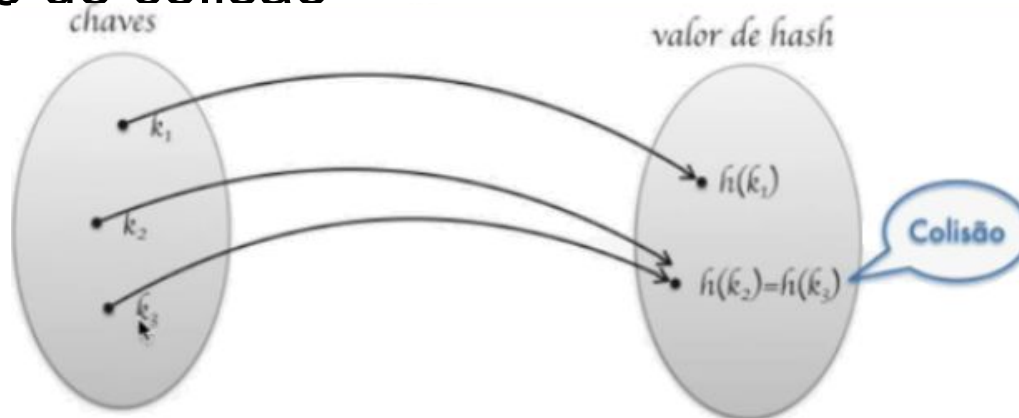


# Colisões

- Uma **colisão** ocorre quando a função de hash gera o mesmo valor para 2 ou mais chaves diferentes.
- Possíveis causas:
  - o número de chaves a armazenar é maior do que o tamanho da tabela;
  - a função de hash utilizada não produz uma boa distribuição (espalhamento).

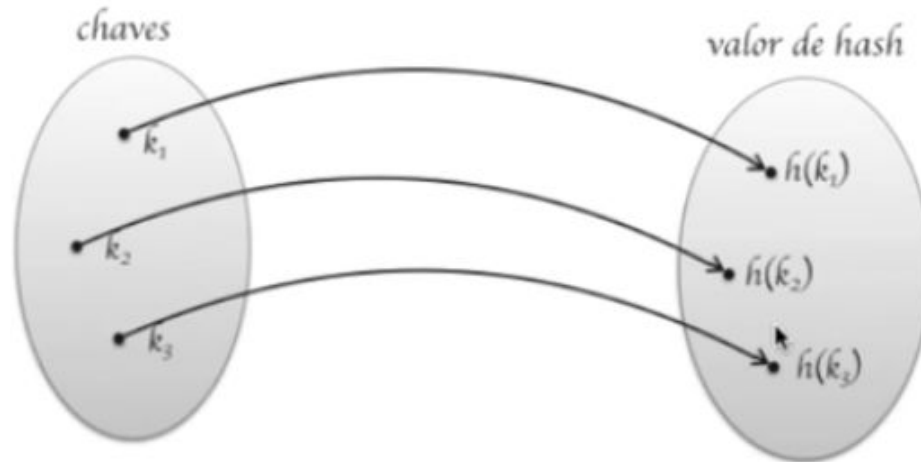
# Colisões

- Não é proibido, mas é sempre preferível evitar, pois degrada o desempenho
  - Se todas as chaves colidem, o desempenho da busca pode cair para  $O(n)$
- Quanto melhor a função, melhor a dispersão e menor a probabilidade de colisão



# Colisões

- Hashing perfeito
  - para cada chave diferente, é obtido um valor de hash diferente.
    - situação muito específica, como quando todas as chaves são previamente conhecidas



# Tratamento de Colisões

- Existem várias opções para tratar colisão
  - Vamos ver a mais simples
    - Endereçamento aberto (*open addressing*)
      - Utiliza-se a chamada “Área de Overflow”

# Endereçamento Aberto

- Todas as chaves são adicionadas à própria tabela, sem nenhuma estrutura de dados auxiliar.
- Em caso de colisão, é necessário procurar uma nova posição para a chave a ser inserida.
- Vantagem: recuperação mais rápida (dados estão no próprio vetor) - sem ponteiros
- Desvantagem: custo extra de calcular a posição. Busca pode se tornar  $O(n)$  quando todas as chaves colidem



# Endereçamento Aberto - Versão Simples

- $chaves = \{7, 17, 36, 100, 106, 205\}$
- $h(k) = k \bmod 10$
- Temos que:
  - $h(7) = 7$
  - $h(17) = 7$  (colisão)
  - $h(36) = 6$
  - $h(100) = 0$
  - $h(106) = 6$  (colisão)
  - $h(205) = 5$

Área de overflow

0	
1	100
2	
3	
4	
5	205
6	36
7	7
8	
9	
10	17
11	106
12	