

Tabelas Hash

Prof. Andrei Braga
Prof. Geomar Schreiner

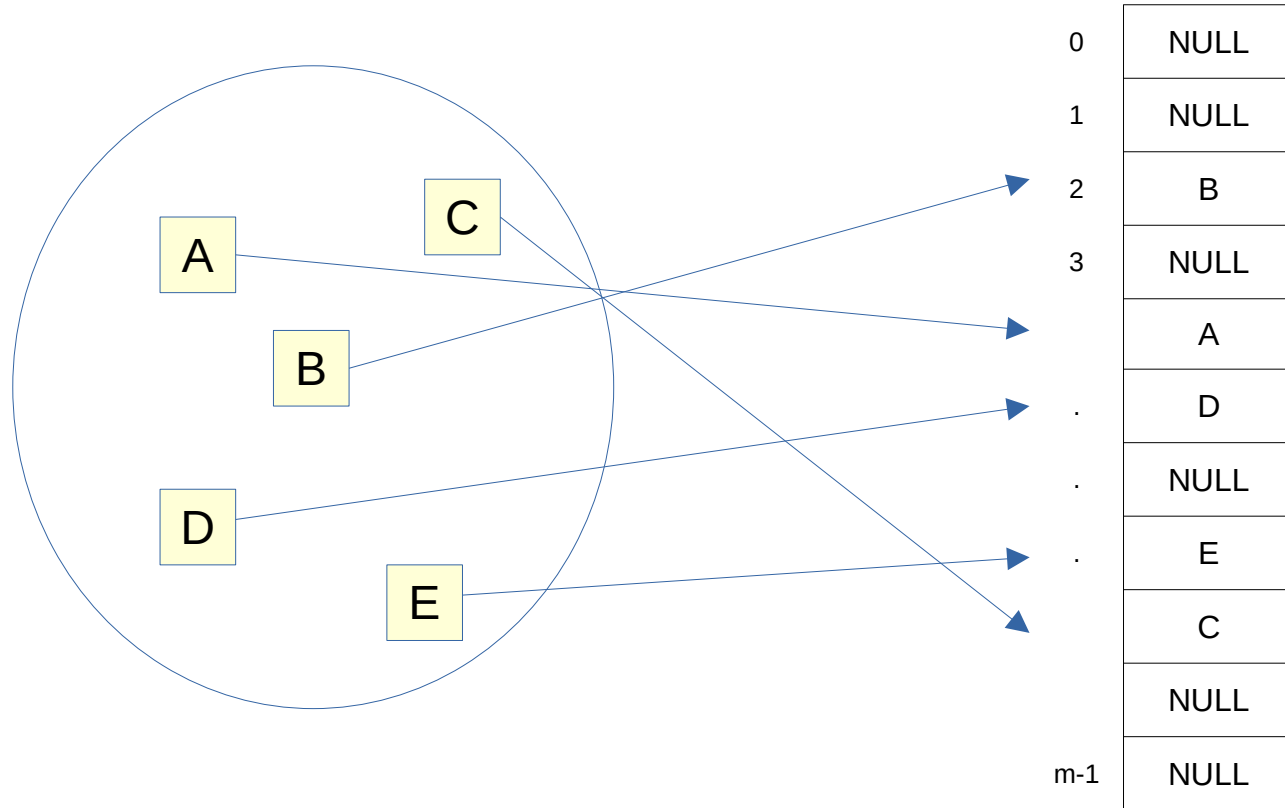
Tabelas Hash

- 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 é na casa de $\log n$ 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

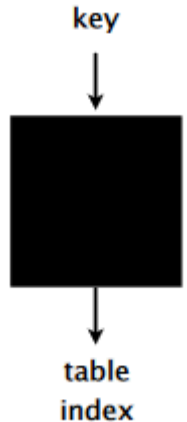


Tabelas Hash

- 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 → 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

- 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 alfabéticos ou alfanuméricos (que não é número).
- Portanto, cada chave deve ser transformada em um inteiro entre 0 e $M-1$ (para M posições).



key
↓

Função de hash

- Exemplo
 - Armazenar um conjunto de número em um vetor de 10 posições
 - Qual função podemos utilizar como função de hash?

Função de hash

- Exemplo
 - Armazenar um conjunto de número em um vetor de 11 posições
 - Vamos aplicar sobre o valor de entrada 'k' mod 10 e o resultado vai ser a posição que queremos utilizar

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


Função de hash

- Exemplo
 - Armazenar um conjunto de número em um vetor de 11 posições
 - Inserir **4**

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Função de hash

- Exemplo
 - Armazenar um conjunto de número em um vetor de 11 posições
 - Inserir 4
 - **HashFunction(4)**



0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	


Função de hash

- Exemplo

- Armazenar um conjunto de número em um vetor de 11 posições

- Inserir 4

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



0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Função de hash

- Exemplo

- Armazenar um conjunto de número em um vetor de 11 posições

- Inserir 4

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



0	
1	
2	
3	
4	4
5	
6	
7	
8	
9	
10	

Função de hash

- Exemplo
 - Armazenar um conjunto de número em um vetor de 11 posições
 - Inserir **17**

0	
1	
2	
3	
4	4
5	
6	
7	
8	
9	
10	

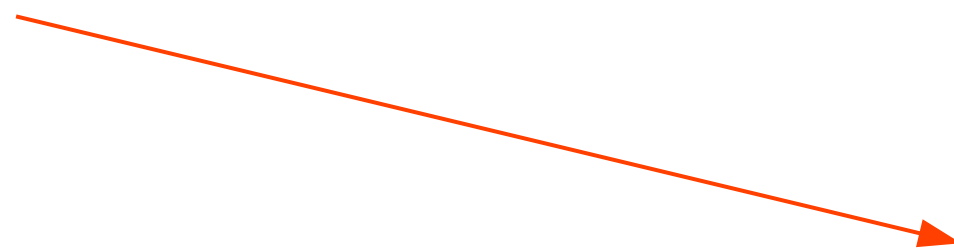
Função de hash

- 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	
10	

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	
10	

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	
8	
9	
10	

Função de hash

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

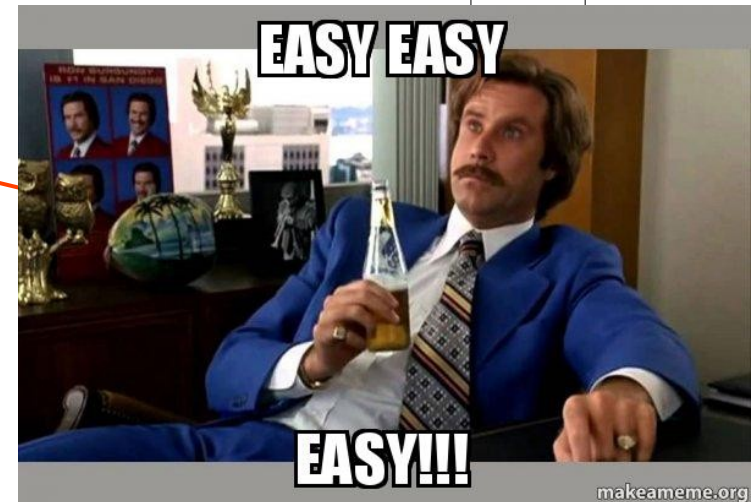


0	
1	
2	
3	
4	4
5	
6	
7	17
8	
9	
10	

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	



Função de hash

- Exemplo
 - Armazenar um conjunto de número em um vetor de 11 posições
 - Inserir **104**

0	
1	
2	
3	
4	4
5	
6	
7	17
8	
9	
10	

Função de hash

- Exemplo
 - Armazenar um conjunto de número em um vetor de 11 posições
 - Inserir **104**
 - **HashFunction(104)**

0	
1	
2	
3	
4	4
5	
6	
7	17
8	
9	
10	

Função de hash

- Exemplo

- Armazenar um conjunto de número em um vetor de 11 posições

- Inserir **104**

- **HashFunction(104)**



0	
1	
2	
3	
4	4
5	
6	
7	17
8	
9	
10	

Função de hash

- Exemplo
 - Armazenar um conjunto de número em um vetor de 11 posições
 - Inserir **104**
 - **HashFunction(104)**

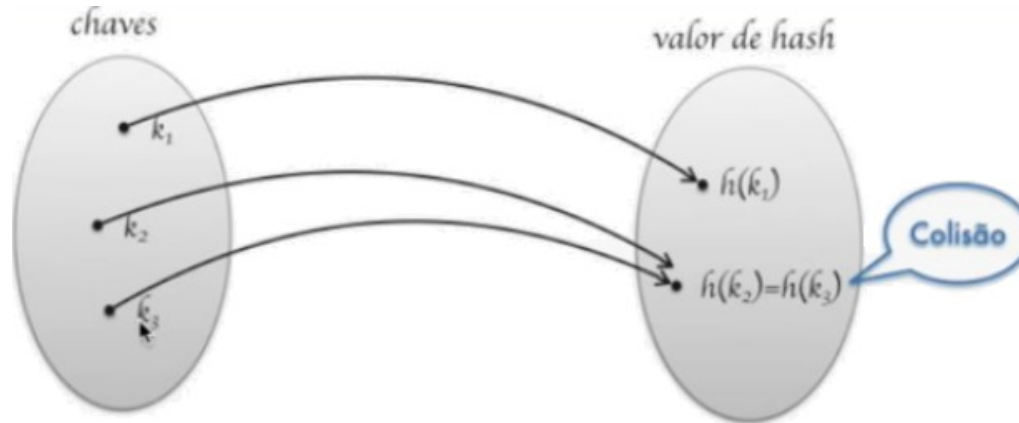


Colisões

- Uma **colisão** ocorre quando a função de hash gera o mesmo valor para 2 ou mais chaves.
- 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).

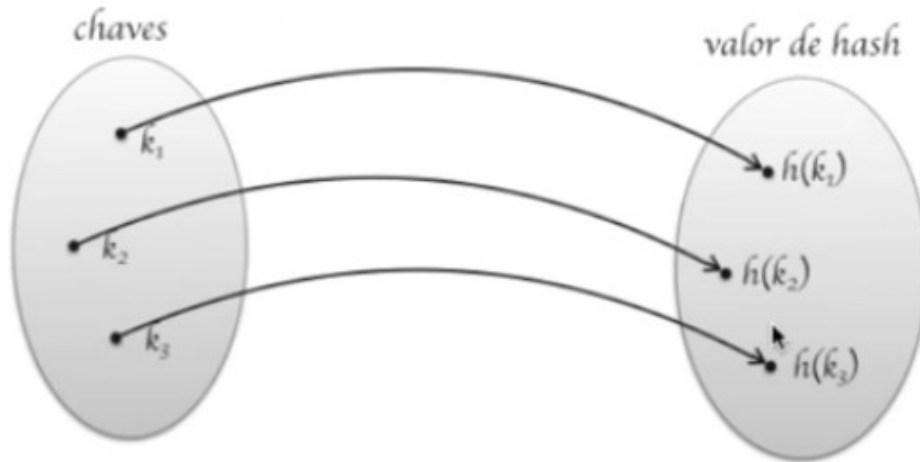
Colisões

- 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

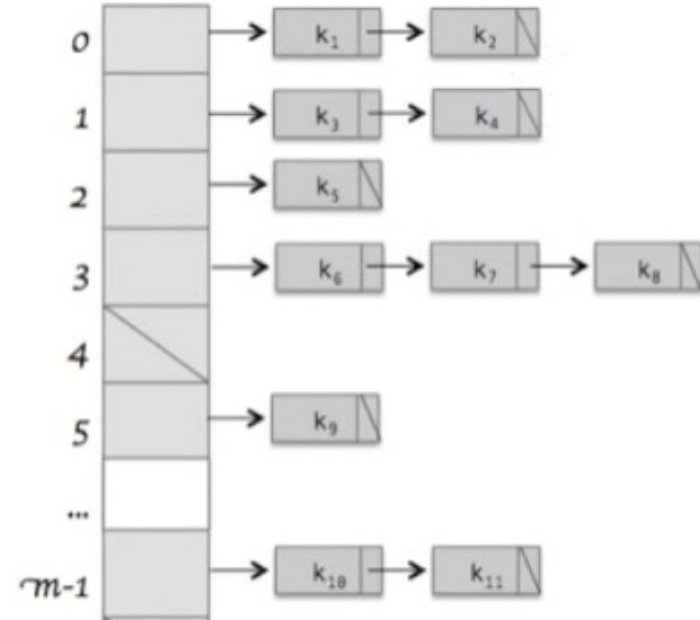
- Endereçamento aberto (*open addressing*)
- Encadeamento separado (*separate chaining*)

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.
 - sondagem linear
 - sondagem quadrática
 - hashing duplo
- Vantagem: recuperação mais rápida (dados estão no próprio vetor). Não utiliza ponteiros.
- Desvantagem: custo extra de calcular a posição. Busca pode se tornar $O(n)$ quando todas as chaves colidem

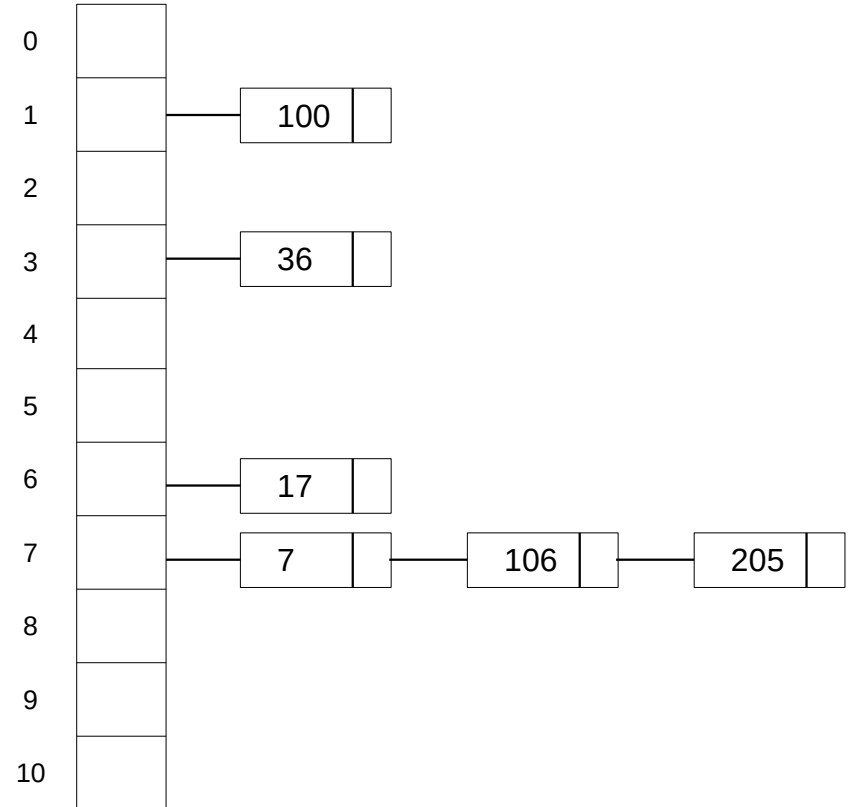
Encadeamento Separado

- Cada posição da tabela aponta para o início lista encadeada.
- Todas as chaves colidem são armazenadas na respectiva lista encadeada (no início ou ao final).
 - Cada valor armazenado deve, portanto, possuir um ponteiro para o próximo elemento da lista.
- Necessita de memória adicional (o que não ocorre no endereçamento aberto).



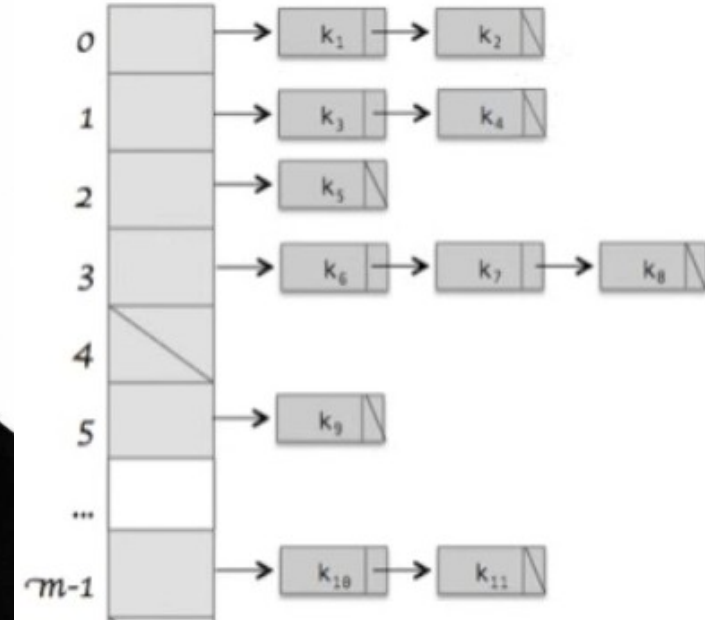
Encadeamento Separado

- chaves = {7, 17, 36, 100, 106, 205}
- $h(k) = k \bmod M$
- Temos que:
 - $h(7) = 7$
 - $h(17) = 6$
 - $h(36) = 3$
 - $h(100) = 1$
 - $h(106) = 7$ (colisão)
 - $h(205) = 7$ (colisão)



Encadeamento Separado

- Cada posição i no array é o início de uma lista encadeada.
- Todas as chaves armazenadas no array são encadeadas (no array).
- Cada valor armazenado no array possui um ponteiro para o próximo da lista.
- Necessita de memória extra, o que não ocorre no encadeamento comum.



Exercício

- Adapte o código construído em aula para suportar uma estrutura complexa (struct).

- A struct a ser definida é a seguinte:

```
typedef struct {  
    int id;  
    char nome[41];  
    double salario;  
    int idade;  
} Funcionario;
```

- Com base nessa estrutura crie um hash que permita a busca pela idade das pessoas. O tamanho máximo da sua hash table é de 20 elementos.
 - Seu sistema deve ser capaz de imprimir na tela (quando solicitado) a hashTable gerada até o momento.