INSTITUTO FEDERAL
Fluminense
Campus Campos Centro

Estrutura de Dados Avançada

- Encontro 14 -

Engenharia da Computação
Prof.º Philippe Leal
philippeleal@yahoo.com.br

Agenda

- Tabelas Hash
  - ✓ Tamanho da Tabela
  - ✓ Função de Hashing
  - ✓ Hashing Universal
  - ✓ Hashing Imperfeito
  - ✓ Hashing Perfeito

Problema

- Princípio de funcionamento dos métodos de busca:
  - Procurar a informação desejada com base na comparação de suas chaves. Isto é, com base em algum valor que a compõe.
- Problema:
  - Algoritmos eficientes necessitam que os elementos estejam armazenados de forma ordenada;
  - Custo da **ordenação**: $\boldsymbol{O(n\log_2 n)}$;
  - Custo da **busca** (valores ordenados): $\boldsymbol{O(\log_2 n)}$.

Problema

- Custo da comparação de chaves é alto;
- O que seria uma operação de **busca ideal**?
  - Seria aquela que permitisse o acesso direto ao elemento procurado, sem nenhuma etapa de comparação de chaves;
  - Nesse caso, teríamos um custo ***O*(1)**
    - Tempo sempre constante de acesso.

Problema

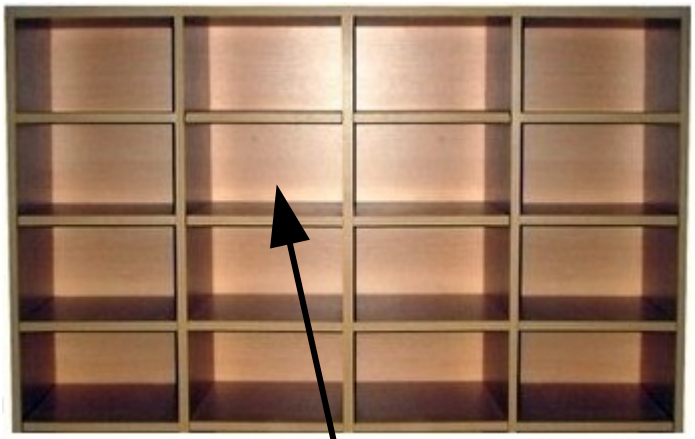
- Uma saída é usar vetores:
  - São estruturas que utilizam índices para armazenar informações;
  - Permite acessar um determinada posição com custo ***O*(1)**;
- Problema:
  - vetores não possuem nenhum mecanismo que permita calcular a posição onde uma informação está armazenada;
  - A operação de busca não é ***O*(1)**.

Problema

- Precisamos do tempo de acesso do vetor juntamente com a capacidade de buscar um elemento em tempo constante;
- Solução: usar uma **Tabela Hash**.

Tabelas Hashing

- Ilustração:



Função

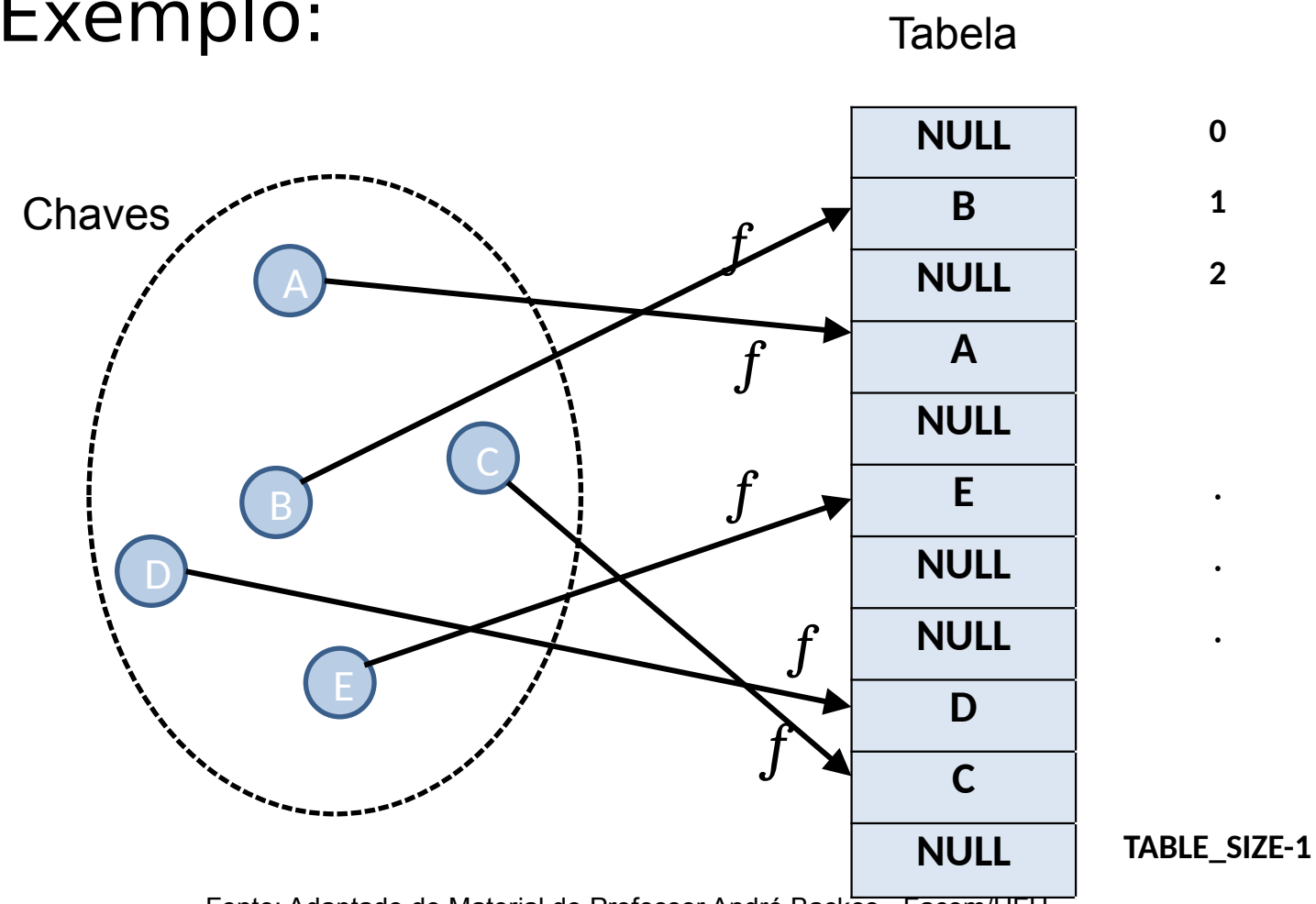
Chave
(nº da gaveta)

Tabelas Hashing

- Também conhecidas como **Tabelas de Indexação** ou **de Espalhamento** ou **de Dispersão**, é uma generalização da ideia de vetor;
- Ideia central:
  - Utilizar uma função f, chamada **Função de Hashing**, para espalhar os elementos que queremos armazenar na tabela;
  - Esse espalhamento faz com que os elementos fiquem dispersos de forma não ordenada dentro do vetor que define a tabela.

Tabelas Hashing

- Exemplo:



Fonte: Adaptado do Material do Professor André Backes - Facom/UFU.

Tabelas Hashing

- Por que espalhar os elementos melhora a busca?
  - A tabela permite associar valores à chaves:
    - **chave**: parte da informação que compõe o elemento a ser inserido ou buscado na tabela;
    - **valor**: é a posição (índice) onde o elemento se encontra no vetor que define a tabela.
  - Assim, a partir de uma **chave** podemos acessar de forma rápida uma determinada **posição** do vetor.
    - Na média, essa operação tem custo ***O*(1)**.

Tabelas Hashing

- Vantagens:
  - Alta eficiência na operação de busca
    - Caso médio é ***O*(1)** enquanto o da busca linear é ***O*(n)** e a da busca binária é ***O*($\log_2 n$)**;
  - Tempo de busca é praticamente independente do número de chaves armazenadas na tabela;
  - Implementação simples.

Tabelas Hashing

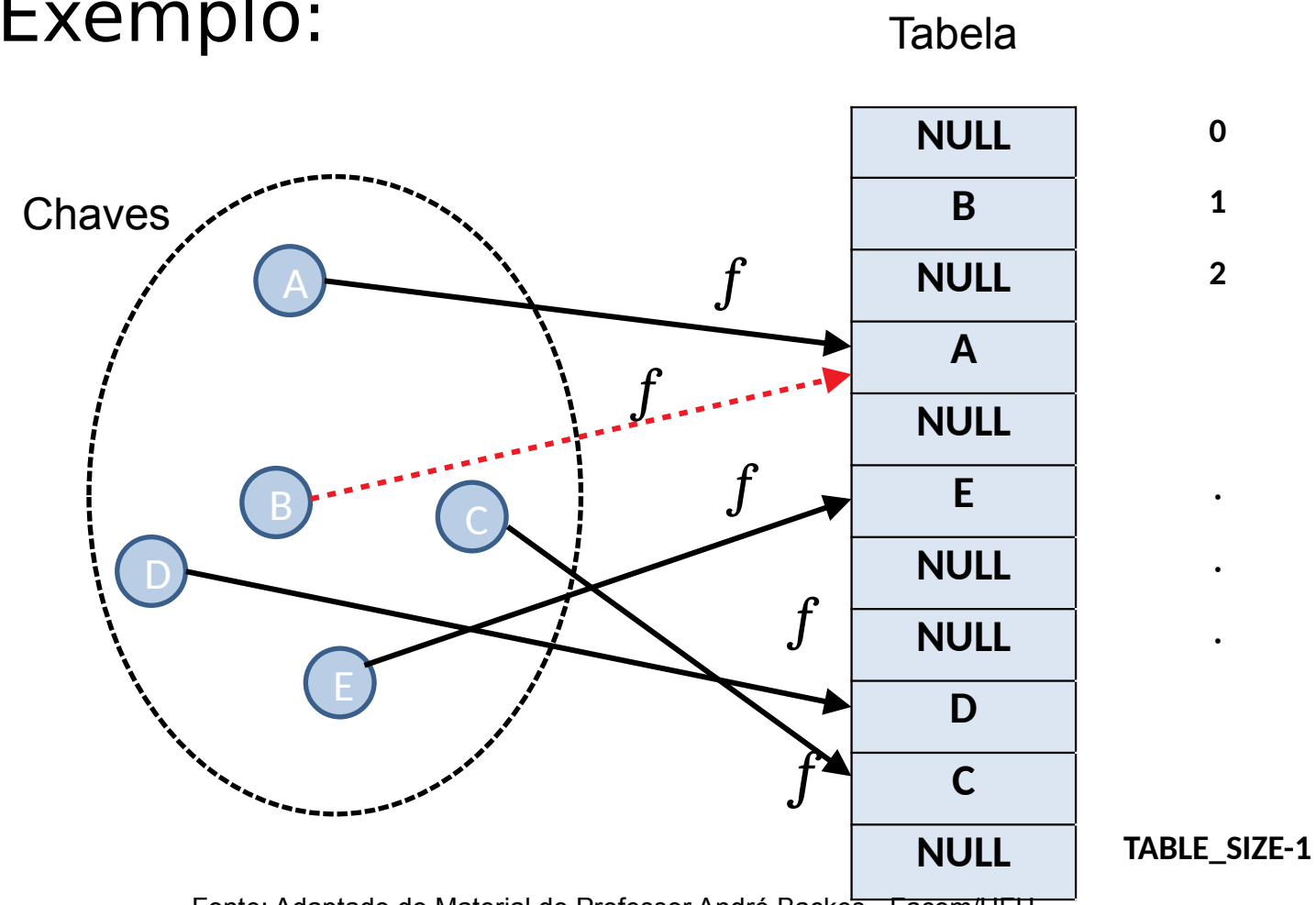
- Infelizmente, esse tipo de implementação também tem suas desvantagens:
  - Alto custo para recuperar os elementos da tabela ordenados pela chave.
    - Nesse caso, é preciso ordenar a tabela.
  - O pior caso é ***O*(n)**, sendo **n** o tamanho da tabela
    - Alto número de **colisões**.

Tabelas Hashing

- O que é uma **colisão**?
  - Uma colisão ocorre quando duas (ou mais) chaves diferentes geram a mesma posição na Tabela Hash.
    - A colisão de chaves não é algo exatamente ruim, é apenas algo indesejável, pois diminui o desempenho do sistema.

Tabelas Hashing

- Exemplo:



Fonte: Adaptado do Material do Professor André Backes - Facom/UFU.

Tabelas Hashing

- Importante:
  - Por questões de desempenho, a tabela irá armazenar apenas o **endereço para a estrutura que contém os dados** e não os dados em si;
  - Isso evita o gasto excessivo de memória.

Aplicações

- A Tabela Hash pode ser utilizada para, por exemplo:
  - **Busca de elementos em base de dados**:
    - Estruturas de dados em memória, bancos de dados e mecanismos de busca na Internet;
  - **Verificação de integridade de dados e autenticação de mensagens**:
    - Os dados são enviados juntamente com o resultado da Função de Hashing;
    - Quem receber os dados recalcula a Função de Hashing usando os dados recebidos e compara o resultado obtido com o que ele recebeu;
    - Resultados diferentes: erro de transmissão.

Aplicações

- A Tabela Hash pode ser utilizada para, por exemplo:
  - **Armazenamento de senhas**:
    - A senha não é armazenada no servidor, mas sim o resultado da Função de Hashing.

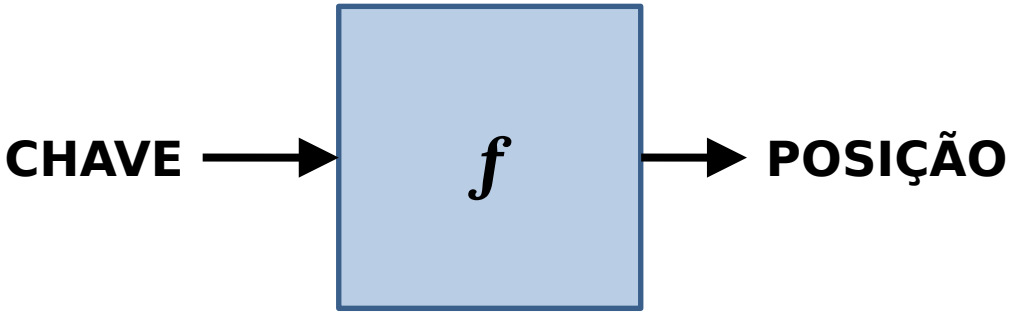
Tamanho da Tabela Hashing

- O ideal é escolher um **número primo**:
  - Reduz a probabilidade de colisões, mesmo que a Função de Hashing utilizada não seja muito eficaz.

Função de Hashing

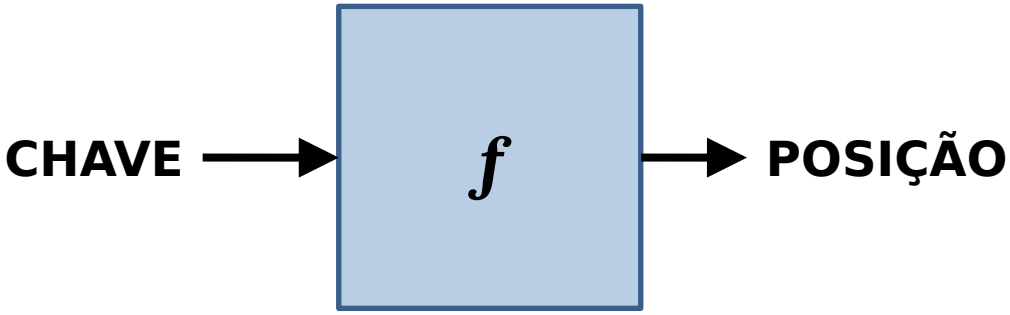
- Inserção e busca: é necessário calcular a posição dos dados dentro da tabela.

- **Função de Hashing**:
  - Calcula a posição a partir de uma chave escolhida dos dados manipulados.



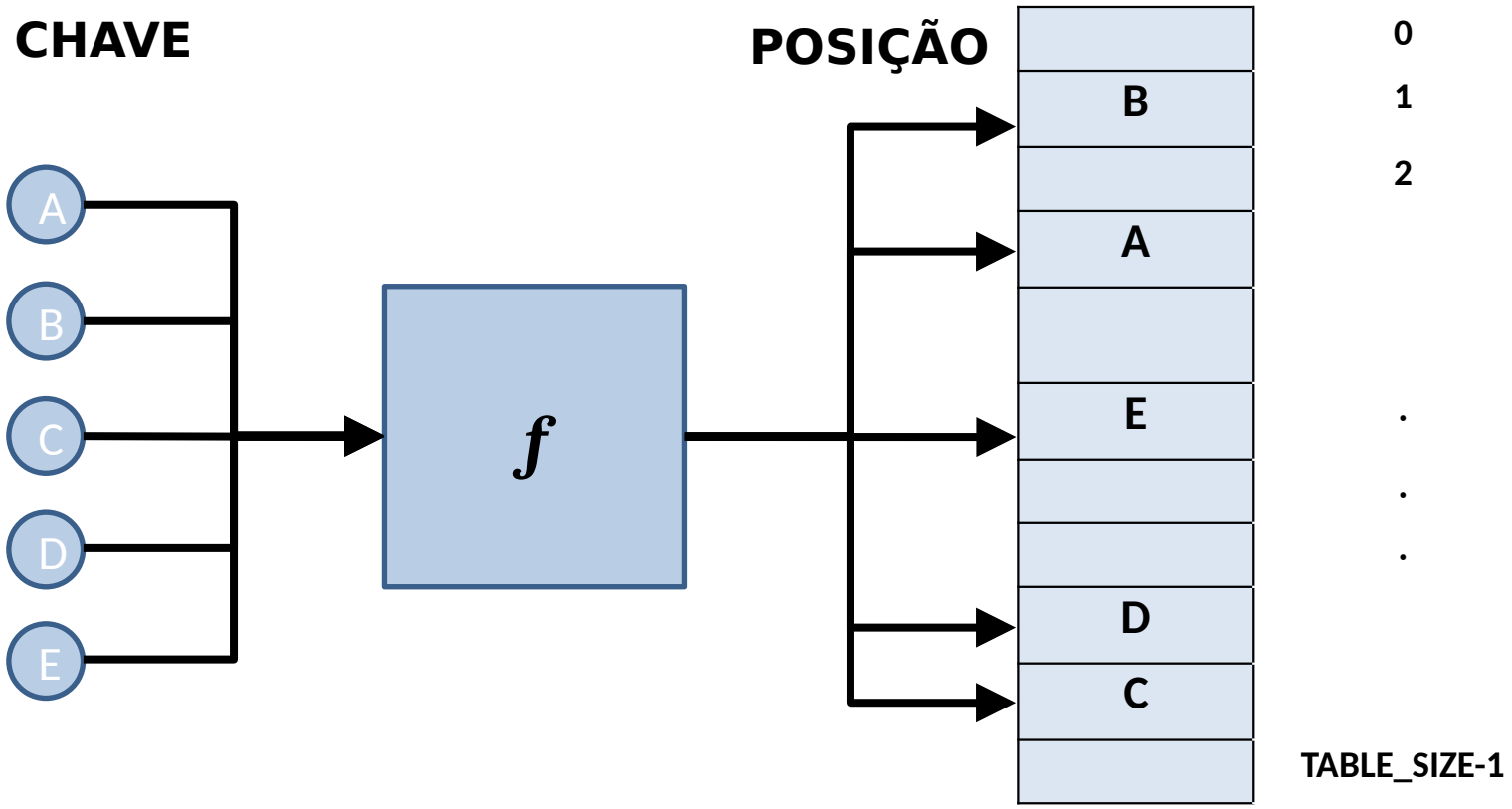
Função de Hashing

- **Função de Hashing**:
  - É extremamente importante para o bom desempenho da Tabela;
  - Ela é responsável por distribuir as informações de forma equilibrada pela Tabela Hash.



Função de Hashing

- Exemplo de funcionamento:



Função de Hashing

- Para um bom funcionamento, a Função de Hashing deve tentar satisfazer as seguintes condições:
  - Ser simples e de baixo custo computacional;
  - Reduzir o número de colisões;
  - Gerar uma distribuição equilibrada dos dados na tabela:
    - Cada posição da tabela tem a mesma chance de receber uma chave (máximo espalhamento).

Função de Hashing

- Sua implementação depende do conhecimento prévio da natureza e domínio da chave a ser utilizada:
  - Exemplo: utilizar apenas três dígitos do número de telefone de uma pessoa para armazená-lo na tabela.
    - Neste caso, seria melhor usar os três últimos dígitos do que os três primeiros, pois os primeiros costumam se repetir com maior frequência e iriam gerar posições iguais na tabela.

Função de Hashing

- Alguns exemplos de Função de Hashing comumente utilizadas:
  - **Método da Divisão**;
  - **Método da Multiplicação**;
  - **Método da Dobra**.

Função de Hashing

- **Método da Divisão**
  - Também chamado **Método da Congruência Linear**, consiste em calcular de maneira simples e direta o **resto da divisão** do valor inteiro que representa o elemento pelo tamanho da tabela (TABLE_SIZE):

$$f(x) = x \textbf{ mod } TABLE\_SIZE$$

  - Assim, $f(x)$ tem valores de [0, TABLE_SIZE–1];
  - Exemplo:

$$f(44) = 44 \textbf{ mod } 23 = 21$$

Função de Hashing

- **Método da Divisão**
  - Apesar de simples, apresenta problema.
    - Resto da divisão: valores diferentes podem resultar na mesma posição.
  - Exemplo:
    - 11 **mod** 10 e 21 **mod** 10 resultam no mesmo valor de posição: 1;
    - Uma maneira de **reduzir** esse tipo de problema é utilizar como tamanho da tabela (TABLE_SIZE):
      - um número primo;
      - um número que não possua divisores primos menores do que 20.

Função de Hashing

- **Método da Multiplicação**
  - Também chamado de **Método da Congruência Linear Multiplicativo**:
    - Utiliza uma constante fracionária **A**, **0 < A < 1**, para multiplicar o valor da chave que representa o elemento;
    - Em seguida, a parte fracionária resultante é multiplicada pelo tamanho da tabela para calcular a posição do elemento.

Função de Hashing

- **Método da Multiplicação**
  - Exemplo: calcular a posição da chave **123456**, usando a constante fracionária **A = 0,724** e que o tamanho da tabela seja **1024**:

f(chave) = ParteInteira(TABLE_SIZE x ParteFracionária(chave x A))

f(chave) = ParteInteira(1024 x ParteFracionária(123456 x 0,724))

f(chave) = ParteInteira(1024 x ParteFracionária(89382,144))

f(chave) = ParteInteira(1024 x 0,144)

f(chave) = ParteInteira(147,456)

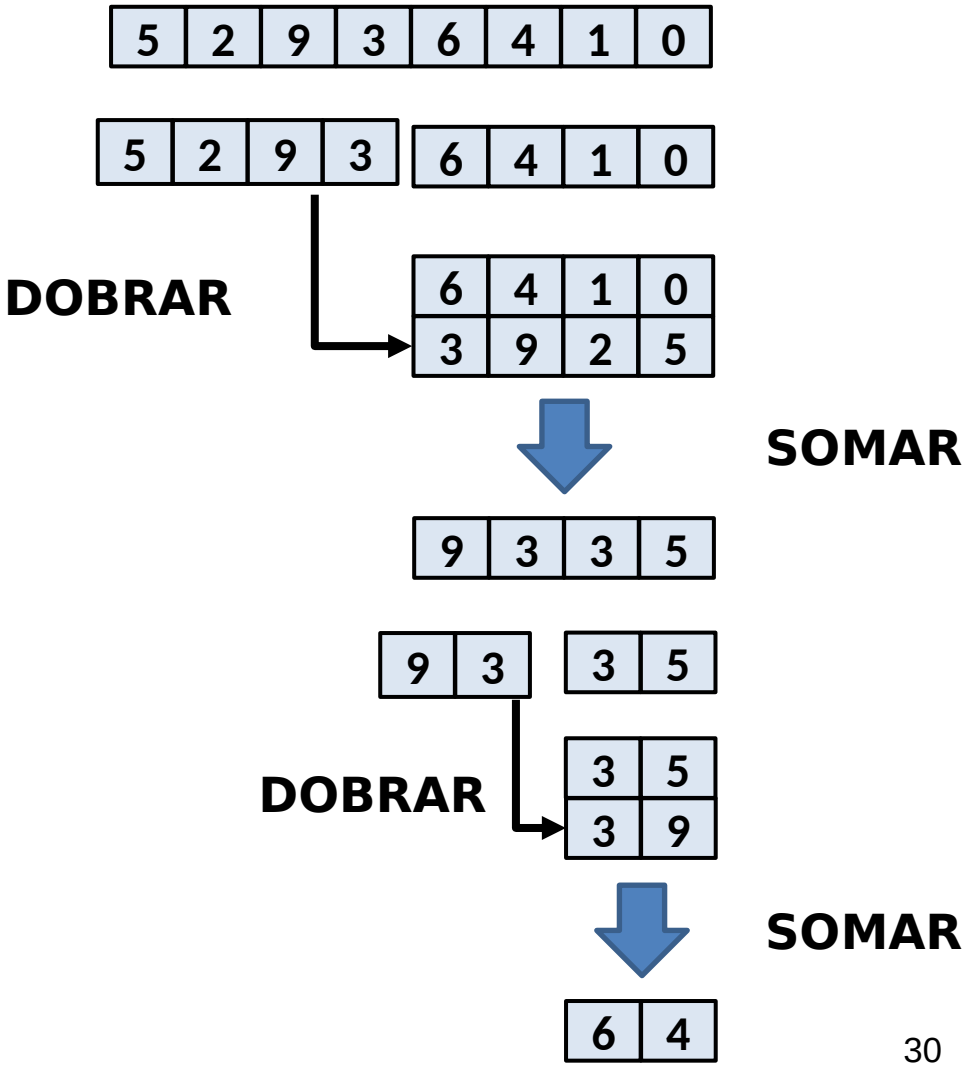
f(chave) = 147

Função de Hashing

- **Método da Dobra**
  - Utiliza um esquema de “dobrar” e somar os dígitos da chave para calcular a sua posição:
    - Considera o valor inteiro que representa o elemento como uma sequência de dígitos escritos num pedaço de papel;
    - Enquanto esse valor for maior que o tamanho da tabela, o papel é dobrado e os dígitos sobrepostos são somados, desconsiderando-se o “vai-um”;
    - Este processo deve ser repetido enquanto os dígitos formarem um número maior que o tamanho da tabela.

Função de Hashing

- **Método da Dobra**
  - Exemplo:



Hashing Universal

- A Função de Hashing está sujeita ao problema de gerar posições iguais para chaves diferentes:
  - Por se tratar de uma função **determinística**, ela pode ser manipulada de forma indesejada;
  - Conhecendo a Função de Hashing, pode-se escolher as chaves de entrada de modo que todas colidam, diminuindo o desempenho da tabela na busca.

Hashing Universal

- **Hashing Universal** é uma estratégia que busca minimizar esse problema de colisões:
  - Basicamente, devemos escolher **aleatoriamente** (em tempo de execução) a Função de Hashing que será utilizada;
  - Para tanto, construímos um conjunto (ou família) de Funções de Hashing.

Hashing Universal

- Uma família de funções pode ser obtida, por exemplo, da seguinte forma:
  - Escolha um número primo **p**. Ele deve ser maior do que qualquer chave **k** (**k ≥** 0) a ser inserida;
  - **p** também deve ser maior do que o tamanho da tabela (TABLE_SIZE);
  - Escolha, aleatoriamente, dois números inteiros, **a** e **b**, de tal modo que **0 < a ≤ p** e **0 ≤ b ≤ p**;

Hashing Universal

- Dados os valores **p**, **a**, e **b**, definimos a Função de Hashing Universal como sendo
  - $\mathbf{h(k)_{a,b} = ((a \times k + b) \% p) \% TABLE\_SIZE}$
    - Este tipo de Função de Hashing Universal permite que o tamanho da tabela não seja necessariamente primo;
    - Além disto, como existem **p** valores diferentes para o valor de **a** e **p+1** valores possíveis para **b**, é possível gerar $\mathbf{p \times (p+1)}$ Funções de Hashing diferentes.

Hashing Imperfeito e Perfeito

- A depender do tamanho da tabela e dos valores inseridos, uma Função de Hashing pode ser definida como:
  - **Hashing Imperfeito**;
  - **Hashing Perfeito**.

Hashing Imperfeito e Perfeito

- **Hashing Imperfeito**:
  - Para duas chaves diferentes, a saída da Função de Hashing é a mesma posição na tabela.

- **Hashing Perfeito**:
  - Chaves diferentes sempre produzirão posições diferentes;
  - Trata-se de um tipo de aplicação muito específica, quando uma colisão não é tolerável. Por exemplo, o conjunto de palavras reservadas de uma linguagem de programação. Neste caso, é conhecido previamente o conteúdo a ser armazenado na tabela.

Tratamento de Colisões

- **Mundo Ideal**
  - **Hashing Perfeito**
    - Função de Hashing fornecerá posições diferentes para cada uma das chaves inseridas.

- **Mundo Real**
  - Independente da Função de Hashing utilizada, ela vai retornar a mesma **posição** para duas **chaves** diferentes: **colisão**!

Tratamento de Colisões

- A criação de uma Tabela Hash necessita basicamente de dois elementos:
  - uma **Função de Hashing**; e
  - uma **abordagem para o tratamento de colisões**.

Tratamento de Colisões

- Uma escolha adequada do **tamanho da tabela** pode minimizar as colisões:
  - Colisões ocorrem porque temos mais chaves para armazenar do que o tamanho da tabela suporta;
  - Não há espaço suficiente para todas as chaves;
  - Uma forma de tentar reduzir as colisões é diminuir o **fator de carga** (α) da tabela:

$$\alpha = \frac{n}{\text{TABLE\_SIZE}}$$

    onde n é o número de registros armazenados na tabela.
  - O número de colisões cresce rapidamente quando o fator de carga aumenta.

Tratamento de Colisões

- Uma escolha adequada da **Função de Hashing** pode minimizar as colisões
  - Escolher uma função que produza um espalhamento uniforme das chaves reduz o número de colisões
    - Infelizmente, não se pode garantir que as Funções de Hashing possuam um bom potencial de espalhamento, porque as colisões também são uniformemente distribuídas;
    - Colisões são teoricamente **inevitáveis**.

Tratamento de Colisões

- Colisões são teoricamente inevitáveis. Por isso, devemos sempre ter uma abordagem para tratá-las.
  - Duas técnicas muito comuns:
    - **Endereçamento Aberto** (*Open Addressing* ou *Rehash*);
    - **Encadeamento Separado** (*Separate Chaining*).

Referências

- BACKES, A. R. **Estrutura de Dados Descomplicada: em Linguagem C**. 1ª Ed. Rio de Janeiro: Elsevier, 2016.

- Material do Professor André Backes (FACOM – UFU).

- SZWARCFITER, J. L. e MARKENZON, L. **Estrutura de Dados e Seus Algoritmos**. 3ª Ed. LTC, 2010.