UNIVERSIDADE CATÓLICA DE PERNAMBUCO

Hashing

Prof Márcio Bueno

ed2tarde@marciobueno.com / ed2noite@marciobueno.com

Hashing

- Fundamentos
- Funções de Hashing
- Tabelas de Hashing
- Operações Básicas

Objetivo

- Agilizar o processo de pesquisa de informações através do armazenamento e recuperação de informações sem utilizar técnicas de ordenação

Definição

- O uso de funções para localizar elementos em uma tabela a partir da conversão de uma chave em um número (o seu endereço) é chamado de "***hashing***".
- "***Hashing***" ⇨ Técnica de conversão de chaves.

Definição

- Seja **P** o conjunto que contém todos os elementos de um determinado universo, possivelmente infinito.
- Chamamos de ***hashing*** ao particionamento de **P** em um número finito de classes $P_1, P_2, \ldots, P_n$, **n > 1**, tal que:

$$\bigcup_{i=1}^{n} P_I = P$$

  - Para todo elemento $k \in P$ deve existir uma classe P_i, $1 \leq i \leq n$, tal que $k \in P_i$.

Definição

- Não existe **k∈P** e **1 ≤ i < j ≤ n**, tal que **k∈P$_i$** e **k∈P$_j$**, ou seja, um mesmo elemento não pode pertencer a mais de uma classe ao mesmo tempo.

$$\bigcap_{i=1}^{n} P_i = \varnothing$$

Definição

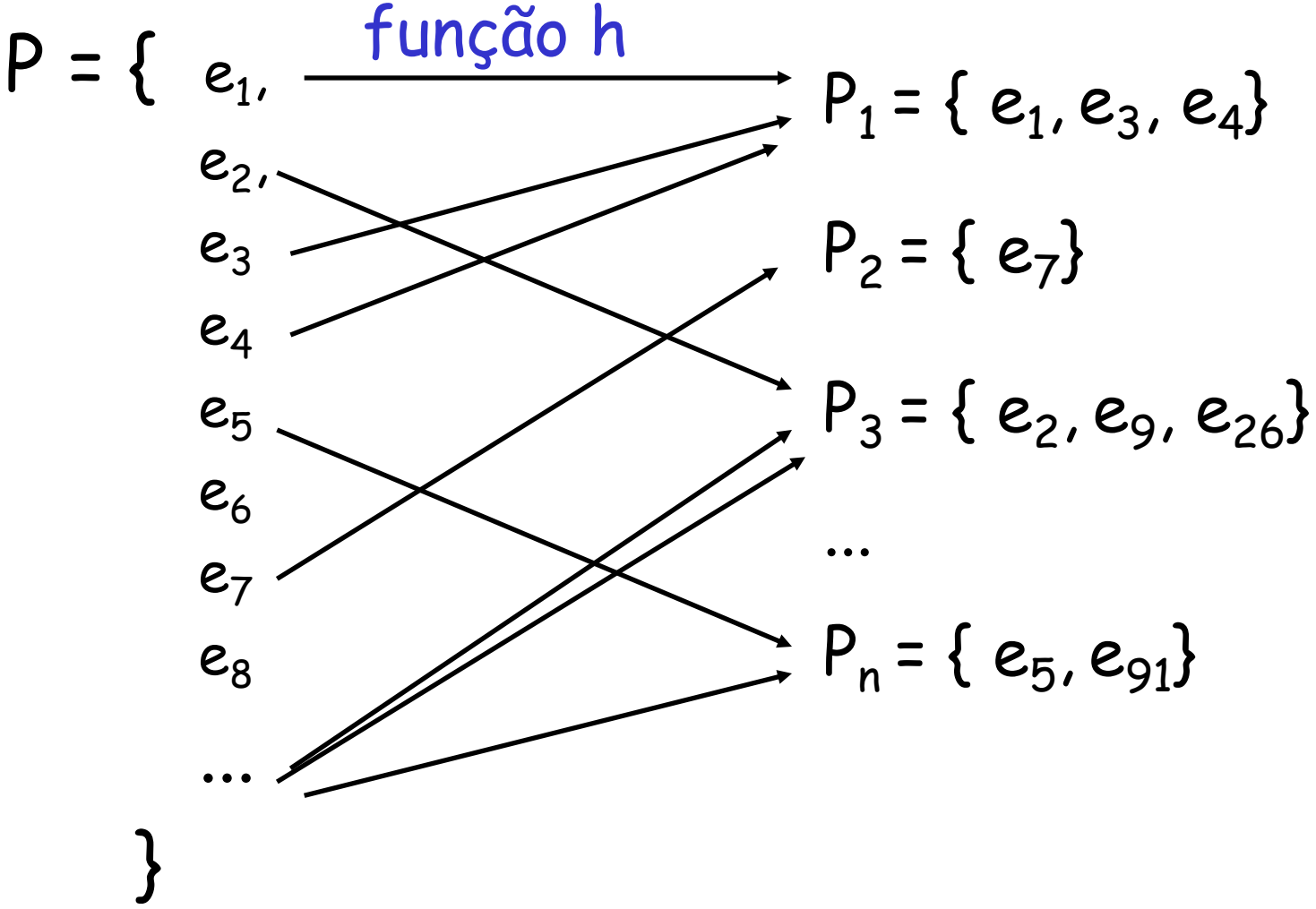
- A correspondência unívoca entre os elementos do conjunto **P** e as **n** classes sugere a existência de uma função **h**, através da qual é feito o particionamento.
- A função **h:P→[1..n]**, que leva cada elemento de **P** à sua respectiva classe, é chamada de ***função de hashing***.

Definição

- Sendo **k** um elemento qualquer do conjunto **P**, o seu valor de *hashing*, **h(k)**, determina a que classe ele pertence.

$$k \in P_{h(k)}$$

Definição


P = {
e1,
e2,
e3
e4
e5
e6
e7
e8
...
}
função h
P1 = { e1, e3, e4}
P2 = { e7}
P3 = { e2, e9, e26}
...
Pn = { e5, e91}

Definição

- Definimos uma função de hashing como sendo um mapeamento

$$H : K \rightarrow E$$

onde "**K**" é a chave que identifica um elemento em uma tabela, "**E**" é o endereço onde esse elemento está (ou deveria estar) e "**H**" é uma função finita e não-recursiva que, dado um valor para **K**, retorna um e apenas um valor para **E**.

Função de Hashing

- Função Ótima - Definição
  - Seja |C| a cardinalidade de um conjunto C.
  - Se, em um *hashing*, a diferença absoluta entre as cardinalidades de quaisquer duas classes for no máximo 1, dizemos que a função de *hashing* utilizada é ótima.

$$\text{Função Ótima} \Leftrightarrow Abs(|Pi| - |Pj|) \leq 1, \quad 1 \leq i < j \leq n$$

Função de Hashing

- Função Ótima - Exemplo
  - Sejam o conjunto **P={a,b,c,d,e,f,g,h}** e a função **h:P→[1..3]**, que "espalha" os elementos de **P** entre três classes distintas e que fornece os seguintes valores de *hashing*:

    `h(a) = 2  h(b) = 1  h(c) = 3  h(d) = 3`
    `h(e) = 1  h(f) = 2  h(g) = 3  h(h) = 1`

  - Temos:

    $P_1 = \{b,e,h\}$ $P_2 = \{a,f\}$ $P_3 = \{c,d,g\}$
    $|P_1| = 3$ $|P_2| = 2$ $|P_3| = 3$

Hashing Uniforme

- É aquele em que todas as classes têm aproximadamente a mesma quantidade de elementos.
- Utiliza uma função de *hashing* ótima.
- A menor classe tem no mínimo **|P| div n** elementos e a maior classe tem no máximo **(|P| div n) + 1** elementos.
- No exemplo anterior:
  - |P| = 8 e n = 3
    - Menor classe: $|P_2| = (8 \text{ div } 3) = 2$
    - Maior classe: $|P_1| = |P_3| = (8 \text{ div } 3) + 1 = 3$

Hashing Perfeito

- Suponha um *hashing* no qual uma função de *hashing* ótima é utilizada para particionar um conjunto **P** em **|P|** classes distintas.
- Como o *hashing* é uniforme (realizado por uma função ótima), cada classe terá um único elemento de **P**.
- Dizemos, neste caso, que o ***hashing*** é **perfeito**.

Hashing Perfeito

- Sinônimos
  - Se um *hashing* não é perfeito, existe pelo menos uma classe com mais de um elemento, isto é, existem $x \in P$ e $y \in P$, tal que $x \neq y$ mas $h(x) = h(y)$;
  - Neste caso, dizemos que **x** e **y** são **sinônimos**.

Considerações

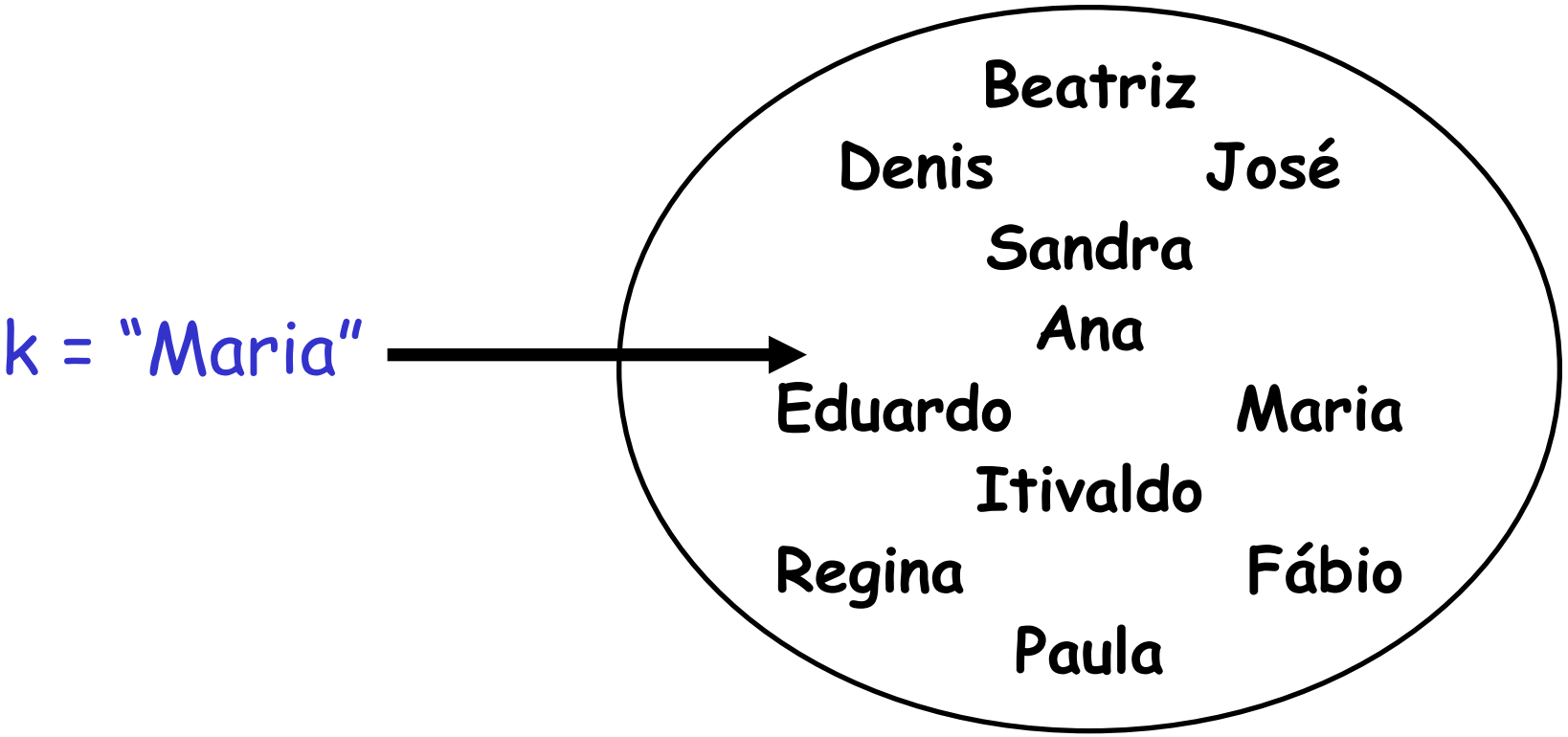
- A grande vantagem do *hashing* está no fato de que, dado um elemento **k** de um conjunto **P**, o valor de *hashing* de **h(k)** pode ser calculado em tempo constante, fornecendo imediatamente a classe de partição em que o elemento se encontra.

Considerações

- Se considerarmos P uma coleção de elementos a ser pesquisada, percebe-se que o processo será mais eficiente se a pesquisa for restrita a uma pequena parte do conjunto P (uma única classe).

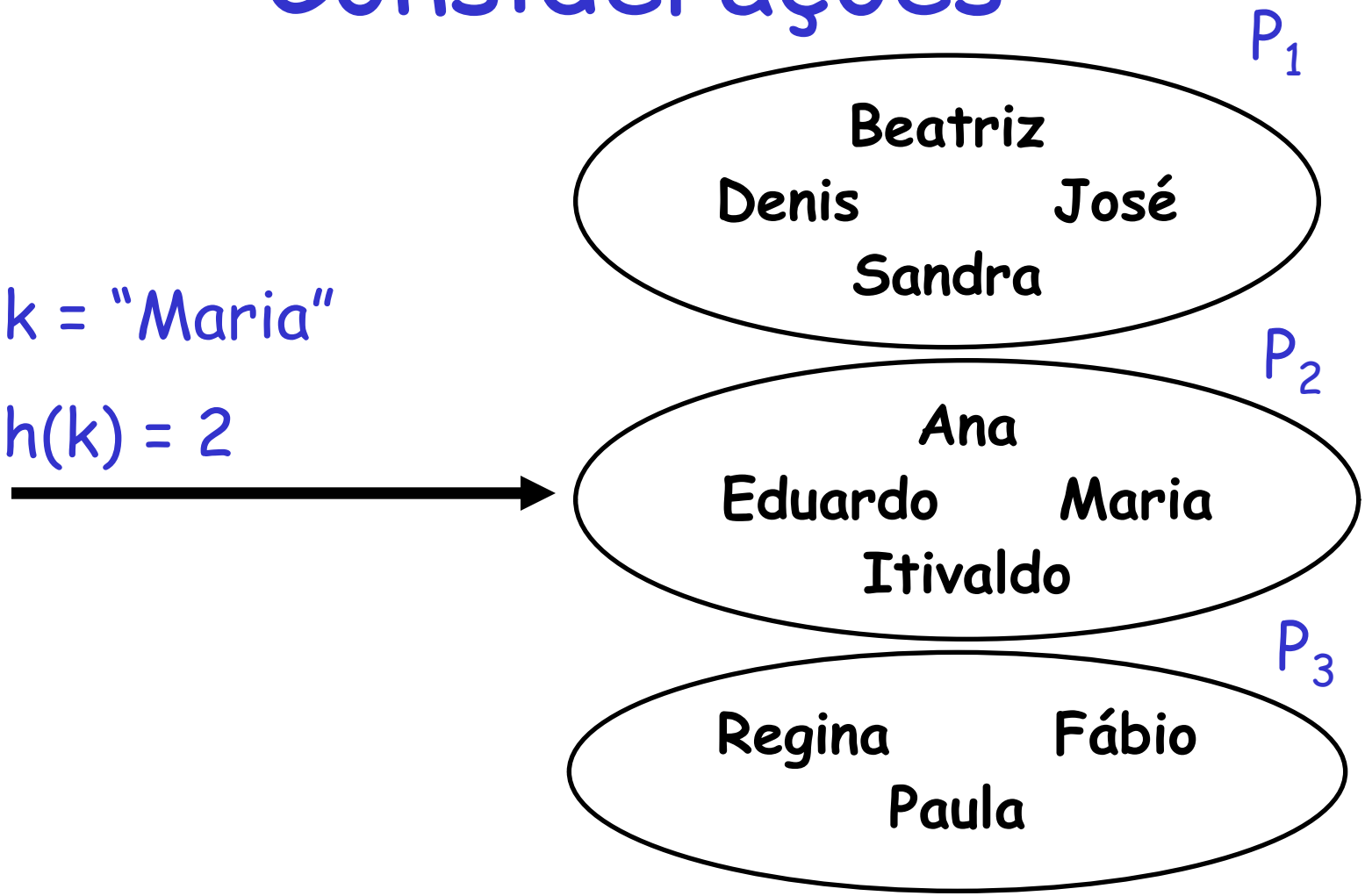
- Exemplo: Pesquisar o elemento "Maria" em um conjunto de nomes próprios.

Considerações



Pesquisa sem Hashing

Considerações



Pesquisa com Hashing

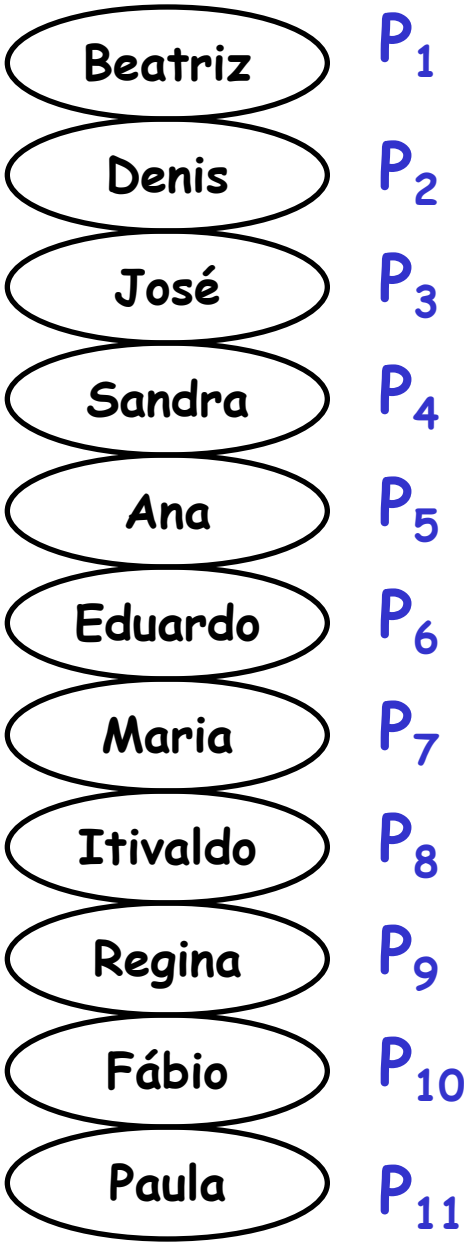
Considerações

- O *hashing* pode ser utilizado como uma técnica de redução do espaço de busca.
- O processo de pesquisa será tão mais eficiente quanto menores forem as partições.
- Se o *hashing* for perfeito, o valor de *hashing* calculado dará diretamente a localização do elemento procurado. Neste caso, temos um **acesso direto** ou **randômico**.

Considerações

k = "Maria"

h(k) = 7



Pesquisa com Hashing – Acesso Direto ou randômico

Funções de Hashing

- Um *hashing* pode ser visto como um método de busca que permite acessar dados diretamente, através de uma função que transforma uma chave **k** em um endereço físico, relativo ou absoluto, **h(k)**.

Funções de Hashing

- Uma função de *hashing* ideal seria aquela capaz de mapear **n** chaves em exatamente **n** endereços, sem ocorrência de colisões.
- Existem $n!$ formas de se obter mapeamento ideal sem colisões
- Entretanto, existem n^n formas possíveis de atribuir n chaves a n endereços
  - Probabilidade mínima: $n!/n^n$

Funções de Hashing

- Na prática, devemos nos contentar com funções capazes de obter um *hashing* mais ou menos uniforme entre **m** endereços, onde **m < n** (sempre haverá colisões)

Funções de Hashing

- Exemplos:
  - Método da Divisão Inteira
  - Método da Divisão Inteira para Chaves Alfanuméricas
  - Método da Permutação para Chaves Alfanuméricas
  - Método da Dobra
  - Método da Multiplicação
  - Método da Análise dos Dígitos

Método da Divisão Inteira

- O método da divisão consiste em realizar uma divisão inteira e tomar o seu resto.

```
função dh (chv:inteiro):inteiro;
  início
     retorne (chv mod N) + 1;
  fim;
```

Método da Divisão Inteira

- Exemplo:

  P = {54,21,15,46,7,33,78,9,14,62,95,87}

  Tabela contendo N=5 encaixes

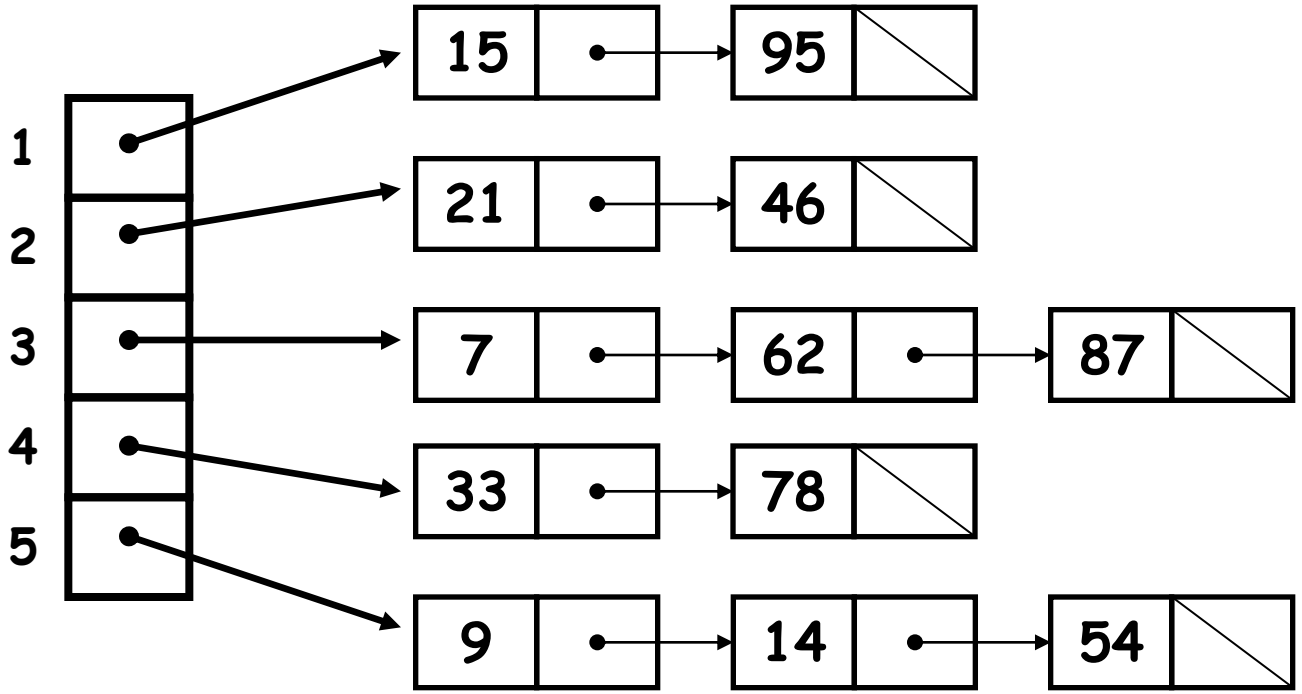
  - dh(54) = (54 mod 5) + 1 = 5
  - dh(21) = (21 mod 5) + 1 = 2
  - dh(15) = (15 mod 5) + 1 = 1
  - dh(46) = (46 mod 5) + 1 = 2
  - dh(7) = (7 mod 5) + 1 = 3

Método da Divisão Inteira

- Exemplo (cont.):
  - dh(33) = (33 mod 5) + 1 = 4
  - dh(78) = (78 mod 5) + 1 = 4
  - dh(9) = (9 mod 5) + 1 = 5
  - dh(14) = (14 mod 5) + 1 = 5
  - dh(62) = (62 mod 5) + 1 = 3
  - dh(95) = (95 mod 5) + 1 = 1
  - dh(87) = (87 mod 5) + 1 = 3

Método da Divisão Inteira

- Exemplo:



Método da Divisão Inteira para Chaves Alfanuméricas

- Transformação das chaves alfanuméricas em valores numéricos para a posterior divisão por N

```
função adh (chv:string):inteiro;
  var i,soma : inteiro
  início
     soma := 0;
     para i de 1 até length(chv) faça
        soma := soma + ord(chv[i]);
     retorne (soma mod N) + 1;
  fim;
```

Método da Divisão Inteira para Chaves Alfanuméricas

- Exemplo 1:

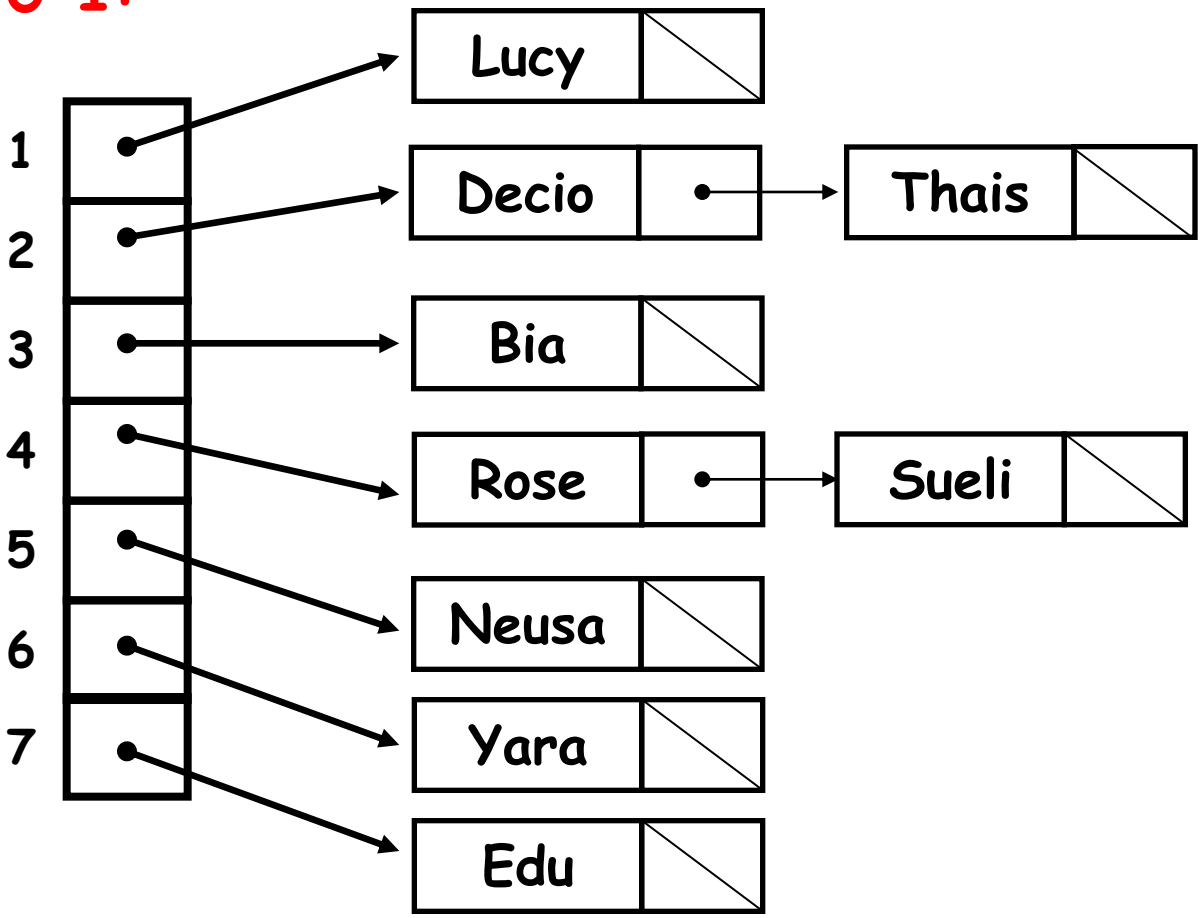
  P = {"Thaís", "Edu", "Bia", "Neusa", "Lucy", "Rose", "Yara", "Decio", "Sueli"}

  Tabela contendo N=7 encaixes

  - adh("Thaís") = 2
  - adh("Edu") = 7
  - adh("Bia") = 3
  - adh("Neusa") = 5
  - adh("Lucy") = 1
  - adh("Rose") = 4
  - adh("Yara") = 6
  - adh("Decio") = 2
  - adh("Sueli") = 4

Método da Divisão Inteira para Chaves Alfanuméricas

- Exemplo 1:



Método da Divisão Inteira para Chaves Alfanuméricas

- Exemplo 2:

  P = {"ABC", "ACB", "BAC", "BCA", "CAB", "CBA"}

  Tabela contendo N=7 encaixes

- Qual o problema?

Método da Divisão Inteira para Chaves Alfanuméricas

- Exemplo 2:

  P = {"ABC", "ACB", "BAC", "BCA", "CAB", "CBA"}

  Tabela contendo N=7 encaixes

  - adh("ABC") = 3
  - adh("ACB") = 3
  - adh("BAC") = 3
  - adh("BCA") = 3
  - adh("CAB") = 3
  - adh("CBA") = 3

- Não há espalhamento!

Método da Divisão Inteira para Chaves Alfanuméricas

- Solução: Somatório com Deslocamentos
  - Associa a cada caractere da chave uma quantidade de bits que deverá ser deslocada à esquerda no seu código ASCII antes de ser adicionado à soma total
  - As quantidades a serem deslocadas variam de 0 a 7, de forma cíclica

Método da Divisão Inteira para Chaves Alfanuméricas

- Somatório com Deslocamentos
  - Posição 1 2 3 4 5 6 7 8 9 ...
  - Chave
  - Desloc. 0 1 2 3 4 5 6 7 0 ...

  - **ord(X) deslocado à esquerda de [(posição-1) mod 8] bits**

Método da Divisão Inteira para Chaves Alfanuméricas

- Somatório com Deslocamentos
  - Exemplo: *A*(65) = 01000001
    - `1ª pos: 01000001 shl 0 = 01000001 = 65`
    - `2ª pos: 01000001 shl 1 = 10000010 = -126`
    - `3ª pos: 01000001 shl 2 = 00000100 = 4`
    - `4ª pos: 01000001 shl 3 = 00001000 = 8`
    - `5ª pos: 01000001 shl 4 = 00010000 = 16`
    - `6ª pos: 01000001 shl 5 = 00100000 = 32`
    - `7ª pos: 01000001 shl 6 = 01000000 = 64`
    - `8ª pos: 01000001 shl 7 = 10000000 = -128`
    - `9ª pos: 01000001 shl 0 = 01000001 = 65`
  - 8º bit da direita para à esquerda é bit de sinal, e se está *setado*, o byte representa um valor negativo em complemento de 2.

Método da Permutação para Chaves Alfanuméricas

- Utiliza somatório com deslocamentos e divisão inteira, em conjunto.

```
função sdh (chv:string):inteiro;
 var i,soma : inteiro
 início
  soma := 0;
  para i de 1 até length(chv) faça
   soma := soma + ord(chv[i])shl((i-1) mod 8);
  retorne ( abs(soma) mod N ) + 1;
 fim;
```

Método da Permutação para Chaves Alfanuméricas

- Exemplo:

  P = {"ABC", "ACB", "BAC", "BCA", "CAB", "CBA"}

  Tabela contendo N=7 encaixes

  - sdh("ABC") = 4
  - sdh("ACB") = 2
  - sdh("BAC") = 3
  - sdh("BCA") = 6
  - sdh("CAB") = 7
  - sdh("CBA") = 5

Tabelas de Hashing

- Uma tabela de *hashing* é uma estrutura de dados que implementa o *hashing* em aplicações computacionais.
- Representada por um vetor onde cada posição, denominada **encaixe**, mantém uma classe da partição.
- O número de encaixes da tabela deve coincidir com o número de classes criadas pela função de *hashing*.

Tabelas de Hashing

- Colisão
  - Quando a função de *hashing* não é perfeita, poderão existir elementos sinônimos;
  - Neste caso, é possível que tenhamos que armazenar um elemento em uma posição da tabela que já se encontre ocupada por outro valor.

Tabelas de Hashing

- Métodos de Tratamento de Colisão
  - Encadeamento
    - Externo
    - Interno
  - Endereçamento Aberto
    - Tentativas Lineares
    - Hashing Duplo

Tabelas de Hashing

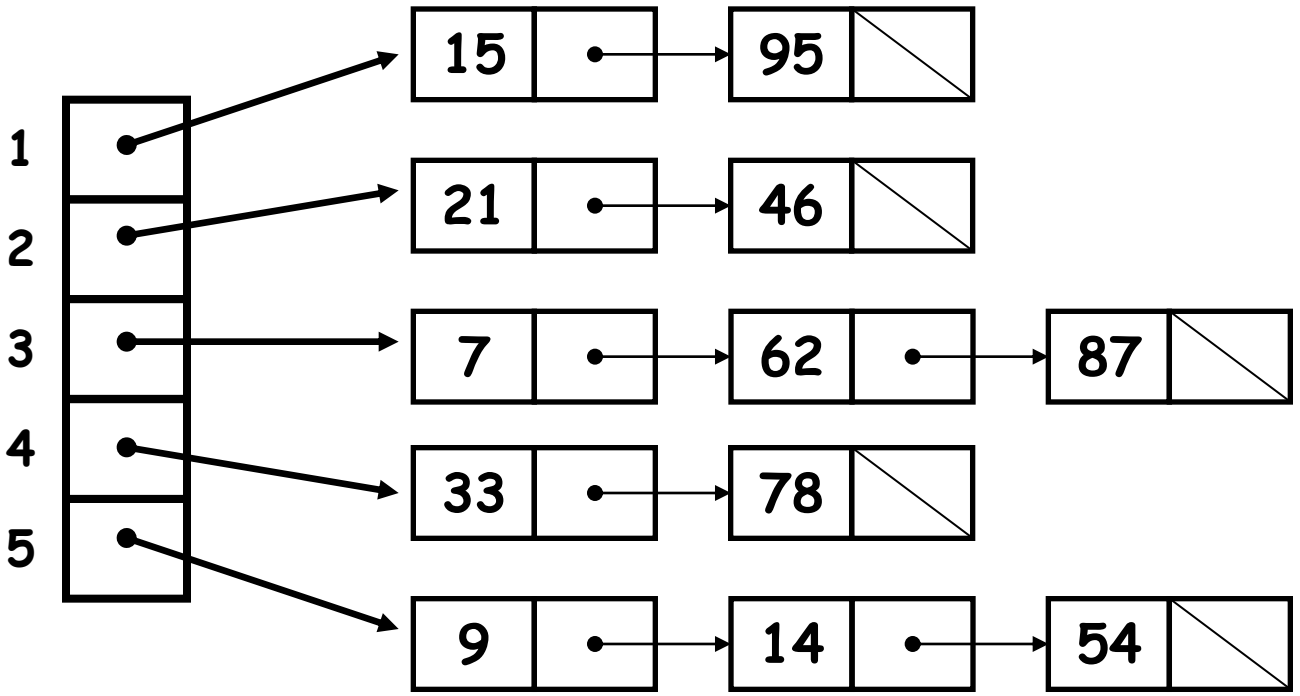
- Métodos de Tratamento de Colisão
  - Encadeamento
    - Quando houver colisão, todas as chaves mapeadas para a mesma posição ficarão juntas em uma lista encadeada.
    - Tipos de Encadeamento
      - Encadeamento Externo
      - Encadeamento Interno

Encadeamento Externo

- Parte do princípio de que existirão muitas colisões durante o carregamento das chaves na tabela e que, na verdade, cada encaixe armazenará não um único elemento, mas uma coleção de elementos sinônimos
- Como a quantidade de elementos sinônimos em cada classe pode variar bastante, a tabela de *hashing* será um vetor cujos elementos são ponteiros para listas encadeadas que representam as classes do *hashing*.

Encadeamento Externo

- Exemplo:



Encadeamento Externo

- Definição da Tabela

```
const TAM = <Tamanho_Tabela>;
type  Clas = ^No;
type  No = record
           chave:<Tipo_Chave>;
           endereco:<Tipo_Endereco>;
           prox:Clas;
         end;
type tabHash = array[1..TAM] of Clas;
var T:tabHash;
```

Encadeamento Interno

- Nesse método, as colisões são resolvidas com ponteiros para outras localizações da tabela de *hashing*.
- O encadeamento interno prevê a divisão da tabela de *hashing* em duas zonas: uma de endereços-base e outra reservada aos sinônimos.

Encadeamento Interno

- Exemplo:

Encadeamento Interno

- Definição da Tabela

```
const TAM = <Tamanho_Tabela>
type No = record
            chave:<Tipo_Chave>;
            endereco:<Tipo_Endereco>;
            prox: inteiro;
          end;
type tabHash = array[1..TAM] of No;
var T:tabHash;
```

Tabelas Hashing

- Métodos de Tratamento de Colisão
  - Endereçamento Aberto
    - Quando houver colisão, é feita uma busca para a localização de um endereço livre, sendo nele armazenado o registro.
    - Utiliza uma tabela circular.
    - Sem ponteiros.
    - Tipos:
      - Tentativas Lineares
      - Hashing Duplo

Hashing com Tentativas Lineares

- Resolve colisões tentando sempre a próxima posição na tabela simplesmente aplicando outra função hashing
- Assim, se há uma colisão causada por h(k) usa-se uma outra função rh(h(K)) que determina outra entrada onde a chave deve ser inserida. Se houver outra colisão, usa-se rh(rh(h(K))) e assim por diante.

Hashing com Tentativas Lineares

- Se não é encontrada nenhuma posição vazia é porque a tabela já está cheia e não podem ser incluídos novos elementos.
- Posição na tabela é dada por:
  `rh(i) = ( i + 1 ) mod N`
- A busca da chave segue a mesma estratégia:
  - Busca com sucesso é indicada quando a chave é encontrada
  - Busca sem sucesso é indicada quando uma posição livre é encontrada ou a exaustão da tabela

Hashing com Tentativas Lineares

- Exemplo:
  - N = 7:

chave	K = ord(chave)	$i_1 = h(K)$	$i_2 = rh(i_1)$	$i_3 = rh(i_2)$	$i_4 = rh(i_3)$
C	67	4	-	-	-
H	72	2	-	-	-
A	65	2	3	-	-
V	86	2	3	4	5
E	69	6	-	-	-
S	83	6	0	-	-

Hashing com Tentativas Lineares

- Exemplo:
  - N = 7:

Tabela

0	S
1	
2	H
3	A
4	C
5	V
6	E

Distribuição das Chaves

Hashing Duplo

- Nesse método de endereçamento livre, o algoritmo resolve o problema das colisões através do uso de duas funções de hashing, **h2 e rh**.
- **h**, chamada de hashing primário é utilizada na primeira vez para determinar a posição na qual o registro deve ficar.
- Se a posição estiver ocupada, a função de re-hashing **rh** será usada sucessivas vezes até que uma posição vazia seja encontrada.

Hashing Duplo

- Para o primeiro cálculo:
  - `h(k) = k mod N`
- Caso haja colisão, inicialmente calculamos h2(K), que pode ser definida como:
  - `h2(k) = 1 + ( k mod (N-1) )`
- Em seguida calculamos a função re-hashing como sendo:
  - `rh(i,k) = ( i + h2(k) ) mod N`

Hashing Duplo

- Exemplo:
  - Suponha uma tabela com 10 entradas (N = 10), todas inicialmente vazias.
  - A inserção do valor 35 será feita na posição 5
    - `h(35) = 35 mod 10 = 5`
  - A inserção, em seguida, do valor 65 será calculada da seguinte maneira:
    - `h(65) = 65 mod 10 = 5`
    - **Colisão: Aplicar hashing duplo!**

Hashing Duplo

- Exemplo:
  - **h(65) = 65 mod 10 = 5 (colisão)**
  - **h2(65) = 1 + (65 mod (10-1)) = 3**
  - **rh(5,65) = (5 + 3) mod 10 = 8**

  - Portanto, 65 será inserido na posição 8 da tabela.

Operações Básicas

- Considerando uma tabela hashing T com espalhamento externo tem-se as seguintes operações.

- Hinit(T) inicia a tabela no estado vazio.

```
procedure Hinit (var T : TabHash);
  var i : integer;
  begin
     for i:=1 to N do
     T[i] := nil;
  end;
```

Operações Básicas

- Hins(T, k) insere a chave k no h(k)-ésimo encaixe da tabela. Ins(L, k) insere numa lista ordenada.

```
procedure Hins (var T:TabHash; k:elem);
  begin
     Ins( T[h(k)], k);
  end;
```

Onde: T[h(k)] retorna um ponteiro para a lista ordenada.

Operações Básicas

- Hrem(T, k) remove a chave k no h(k)-ésimo encaixe da tabela. Rem(L, k) remove numa lista ordenada.

```
procedure Hrem (var T:TabHash; k:elem);
  begin
     Rem( T[h(k)], k);
  end;
```

Onde: T[h(k)] retorna um ponteiro para a lista ordenada.

Operações Básicas

- Hfnd(T, k) procura a chave k no h(k)-ésimo encaixe da tabela. Fnd(L, k) pesquisa numa lista ordenada.

```
procedure Hfnd (var T:TabHash;
  k:elem):boolean;
  begin
     Hfnd:=(Fnd(T[h(k)], k) <> nil);
  end;
```

Onde: T[h(k)] retorna um ponteiro para a lista ordenada.

Bibliografia

- PEREIRA, S. do L. **Estrutura de Dados Fundamentais: conceitos e aplicações.** São Paulo: Érica, 2001.
- SZWARCFITER, J. L.; MARKENZON, L.. **Estruturas de Dados e seus Algoritmos**. 2. ed. rev. Rio de Janeiro: LTC, 1994.
- VELOSO, P. et al. **Estrutura de Dados.** Rio de Janeiro: Campus, 1992.
- WIRTH, N. **Algoritmos e Estruturas de Dados**. Rio de Janeiro: Prentice Hall do Brasil, 1989.
- ZIVIANI, N. **Projetos de Algoritmos com Implementação em Pascal e C**. São Paulo: Pioneira, 1993.