Pesquisa em Memória Primária

Estrutura de Dados II

Prof. João Dallyson

Email: Joao.dallyson@ufma.br

Sumário

- **Introdução**
- **Pesquisa Sequencial**
- **Pesquisa Binária**

Conceitos Básicos

- **Pesquisa**
  - É o processo de procurar por um item com propriedades especificadas dentre uma coleção de itens.
  - A informação é dividida em registros e cada registro contém uma chave.
- **Objetivo:**
  - Encontrar itens com chaves iguais a chave dada na pesquisa
- **Aplicações:**
  - Contas em um banco; Reservas de uma companhia aérea

Conceitos Básicos

- **Escolha do método de pesquisa mais adequada a uma determinada aplicação**
  - Depende principalmente:
    - Quantidade dos dados envolvidos
    - Arquivo está sujeito a inserções e retiradas frequentes

  Se o conteúdo do arquivo é estável é importante minimizar o tempo de pesquisa, sem preocupação com o tempo necessário para estruturar o arquivo

Tipos Abstratos de Dados

- **Considerar os algoritmos de pesquisa e tipos abstratos de dados (TADs)**
  - Há independência de implementação para operações
- **Operações:**
  - Inicializar a estrutura de dados
  - Pesquisar um ou mais registros com uma dada chave
  - Inserir um novo registro
  - Remover um registro específico
  - Ordenar os registros

Tabelas de símbolos

- **Estrutura de dados contendo itens com chaves que suportam três operações**
  - Inserção de um novo item com uma determinada chave
  - Remover um item com uma determinada chave
  - Recuperar um item com uma determinada chave
- **Tabelas são também conhecidas como dicionários**
- **Mapeamento de chaves para valores**
  - **matrícula** – nome, CR, curso,
  - **palavra** – significado, pronúncia, separação silábica
  - **carro** – construtora, potência, comprimento, cilindradas

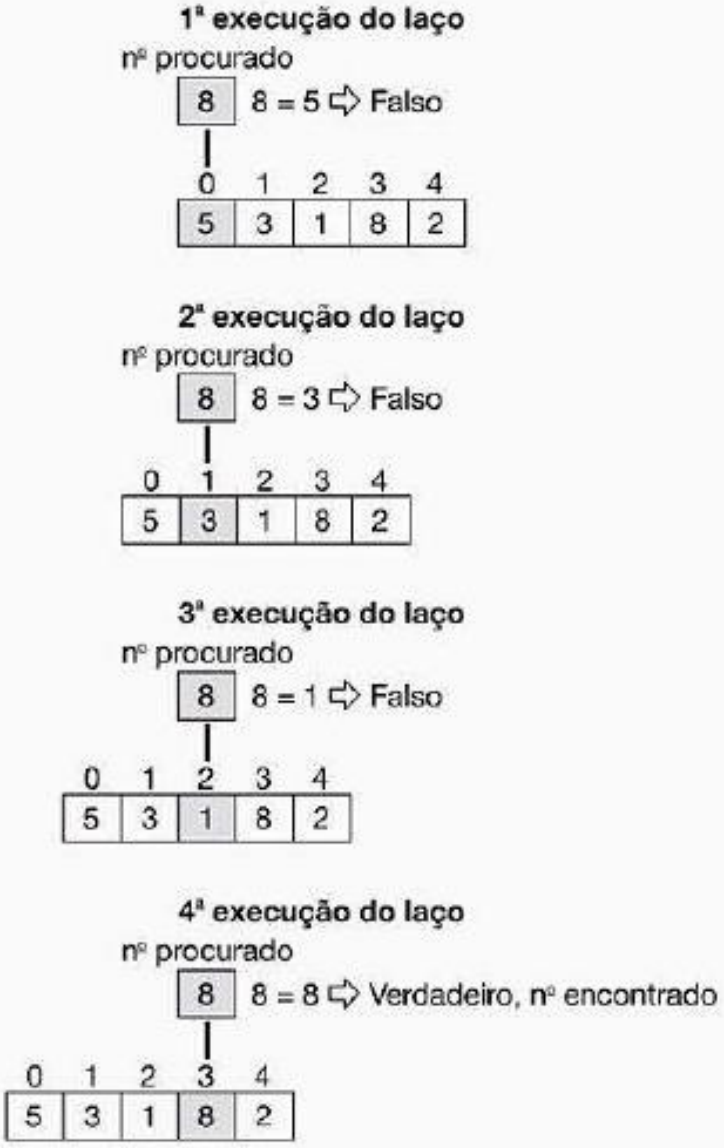
Dicionário

- **Nome comumente utilizado para descrever uma estrutura de dados para pesquisa.**
- **Dicionário é um tipo abstrato de dados com as operações:**
  - 1. Inicializa
  - 2. Pesquisa
  - 3. Insere
  - 4. Retira
- **Analogia com um dicionário da língua portuguesa:**
  - Chaves () ←→ palavras
  - Registros ()←→ entradas associadas com cada palavra:
    - pronúncia
    - Definição
    - Sinônimos
    - outras informações

Pesquisa Sequencial

- **Método de pesquisa mais simples**
  - A partir do primeiro registro, pesquisa sequencialmente até encontrar a chave procurada
  - Registros ficam armazenados em um vetor
  - Pode ser executado em um vetor ordenado ou não ordenado
  - Inserção de um novo item
    - Adiciona no final do vetor
  - Remoção de um item com chave específica
    - Localiza o elemento, remove-o e coloca o último item do vetor em seu lugar.

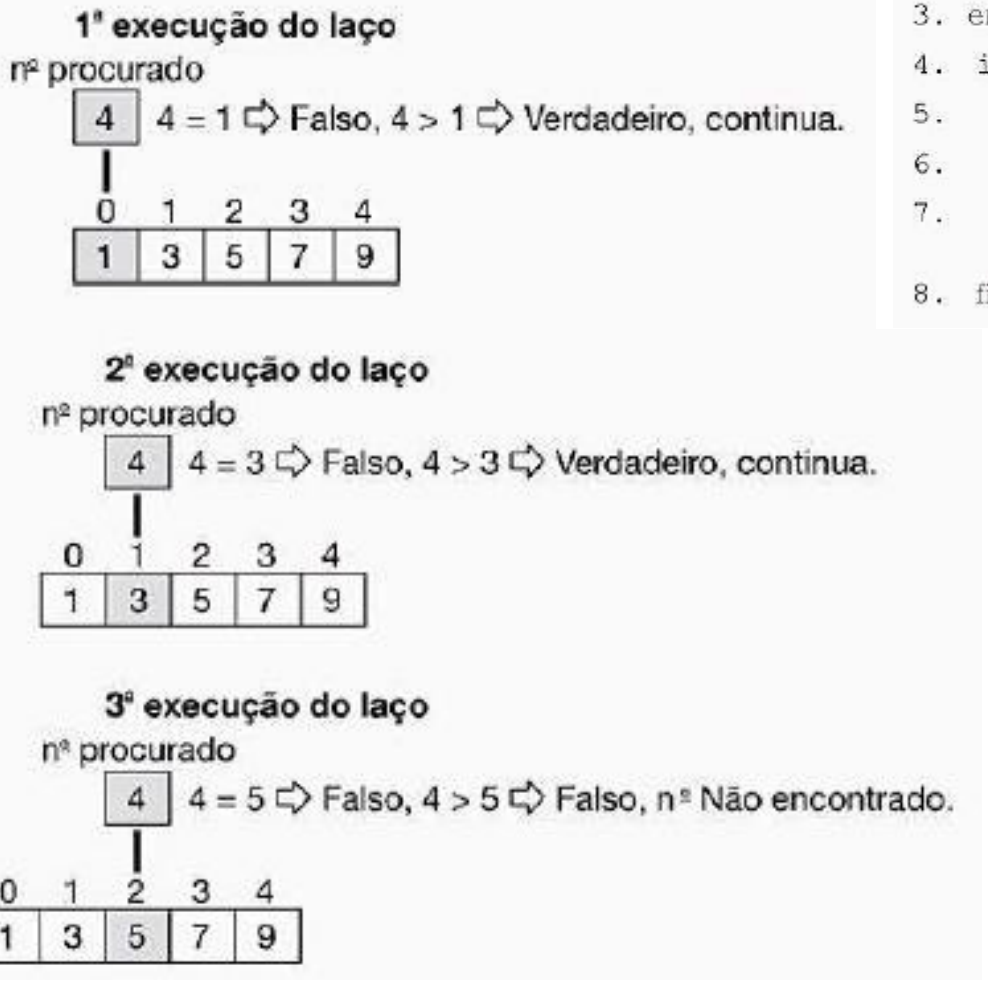
Pesquisa Sequencial – itens não ordenado




```
1. achou ← 0
2. i ← 0
3. enquanto (i <= 9 e achou = 0) faça
4.  início
5.      se (X[i] = n)
6.      então achou ← 1
7.      senão i ← i + 1
8. fim
```

FONTE: [ASCENCIO, 2010]

Pesquisa Sequencial – itens ordenados




```
1. achou ← 0
2. i ← 0
3. enquanto (i <= 9 e achou = 0 e n >= X[i]) faça
4.  início
5.      se (X[i] = n)
6.      então achou ← 1
7.      senão i ← i + 1

8.  fim
```

FONTE: [ASCENCIO, 2010]

Pesquisa Sequencial

```java
package deinf.ufma.br.pesquisa;

public interface Item {

    public int compara ( Item it ) ;
    public void alteraChave (Object chave) ;
    public Object recuperaChave ( ) ;

}
```

```java
package deinf.ufma.br.pesquisa;

public class MeuItem implements Item {
    private int chave;

    public MeuItem ( int chave) { this.chave = chave; }

    @Override
    public int compara(Item it) {
        MeuItem item = (MeuItem) it;
        if ( this.chave < item.chave) return -1;
        else if ( this.chave > item.chave) return 1;
        return 0;
    }

    @Override
    public void alteraChave(Object chave) {
        Integer ch = ( Integer ) chave;
        this.chave = ch. intValue ( ) ;

    }

    @Override
    public Object recuperaChave() {
        return new Integer ( this.chave) ;
    }

}
```

Pesquisa sequencial

```
package deinf.ufma.br.pesquisa;

public class Tabela {
    private Item registros [ ] ;

    private int n;

    public Tabela ( int maxN) {
        this.registros = new Item[maxN+1];
        this.n = 0;
    }

    public int pesquisa ( Item reg) {
        this.registros [0] = reg ; // sentinela
        int i = this.n;
        while ( this.registros[i].compara ( reg) != 0) i--;
        return i ;
    }

    public void insere ( Item reg) throws Exception {
        if ( this.n == ( this.registros.length - 1))
            throw new Exception ( "Erro : A tabela esta cheia" ) ;
        this.registros[++this.n] = reg;
    }

}
```

Pesquisa Sequencial

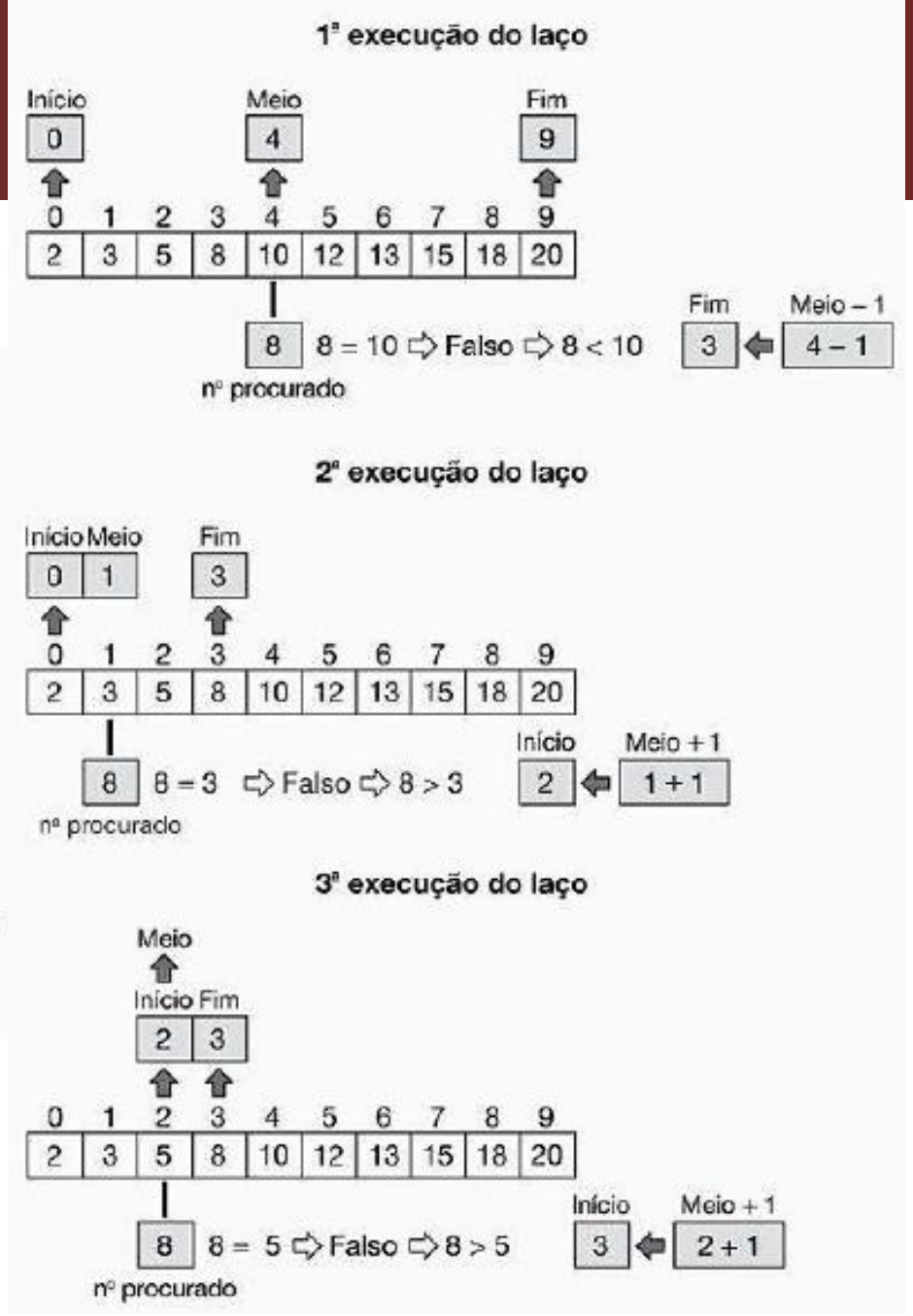
- **Análise de complexidade:**
  - Pesquisa com sucesso
    - Melhor caso: T(n) = 1
    - Pior caso: T(n) = n
    - Caso médio: T(n) = (n+1)/2
  - Pesquisa sem sucesso
    - T(n) = n+1

Pesquisa Binária

- **Pesquisa mais eficiente:**
  - Se os registros forem mantidos em ordem
- **Localizar chave na tabela:**
  1. **Compare** a chave com o registro que está na posição do **meio** da tabela.
  2. Se a chave é **menor** então o registro procurado está na **primeira metade** da tabela
  3. Se a chave é **maior** então o registro procurado está na **segunda metade** da tabela.
  4. Repita o processo até que a chave seja encontrada, ou fique apenas um registro cuja chave é diferente da procurada, significando uma pesquisa sem sucesso

Pesquisa Binária

```
1. achou ← 0
2. início ← 0
3. fim ← 9
4. meio ← parteinteira((início+fim)/2)
5. enquanto (início <= fim e achou = 0) faça
6.  início
7.       se (X[meio] = np)
8.       então achou ← 1
9.       senão início
10.            se (np < X[meio])
11.              então fim ← meio-1
12.              senão início ← meio+1
13.              meio ← parteinteira((início+fim)/2)
14.            fim
15. fim
```



FONTE: [ASCENCIO, 2010]

Pesquisa Binária

```
// Busca binária... a tabela deve estar ordenada
public int buscaBinaria ( Item chave) {
    if ( this.n == 0) return 0;

    int esq = 1 , dir = this.n, i ;

    do {
        i = (esq + dir ) / 2;
        if (chave.compara(this.registros[i]) > 0) esq = i + 1;
        else dir = i - 1;
    } while ((chave.compara(this.registros[i]) != 0) && (esq <= dir));

    if (chave.compara(this.registros[i]) == 0) return i ;
    else return 0;

}
```

Pesquisa Binária – Análise de complexidade

- A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio.
- **Logo**: o número de vezes que o tamanho da tabela é dividido ao meio é cerca de **logn**.
- **Ressalva**: o custo para manter a tabela Ordenada é alto: a cada inserção na posição p da tabela implica no deslocamento dos registros a partir da posição p para as posições seguintes.
- Consequentemente, a pesquisa binária não deve ser usada em aplicações muito dinâmicas.

Pesquisa em Memória Primária

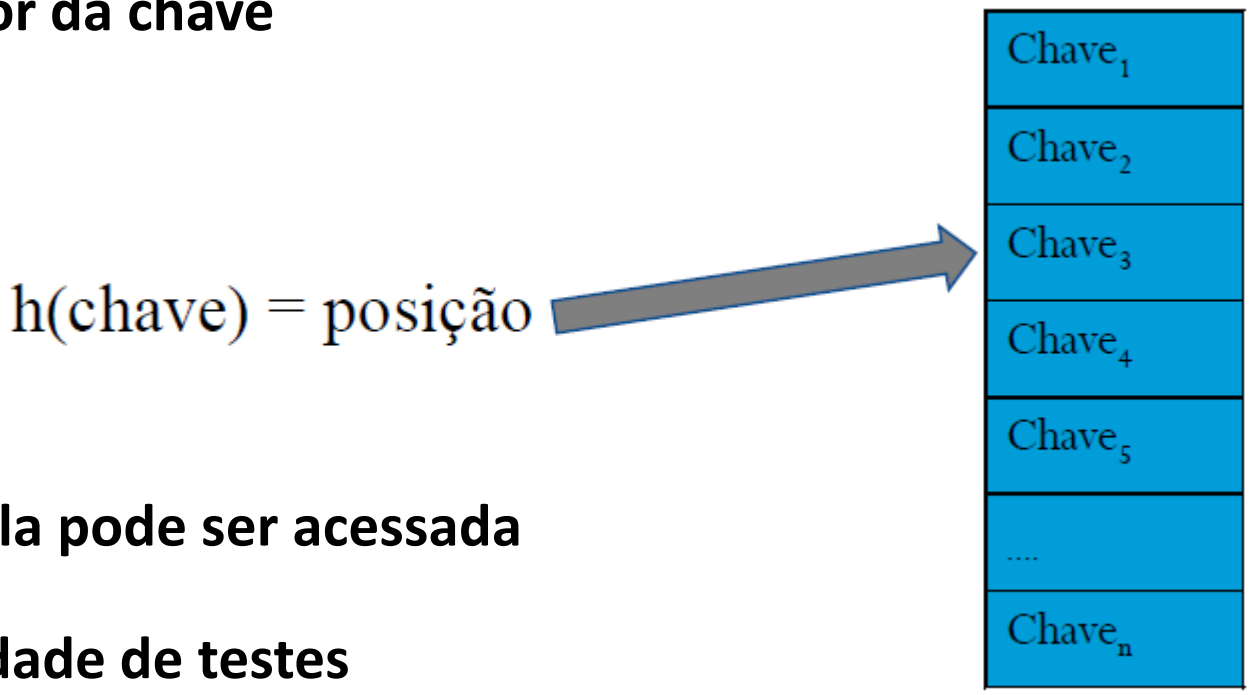
- **Outras estruturas de dados:**
  - Árvores de pesquisa binária
  - Árvores B (estrutura para memória secundária)
  - Árvores SBB (2 tipos de referências: vertical e horizontal)
  - **Hashing**.

Tabela Hash

- **Diversas aplicações exigem de forma dinâmica apenas as operações de dicionário:**
  - INSERIR
  - BUSCAR
  - DELETAR
- **Uma tabela Hash é uma estrutura de dados eficiente para implementar dicionários!**
- **Hash significa:**
  - Fazer picadinho de carne e vegetais para cozinhar.
  - Fazer uma bagunça. (Webster's New World Dictionary)

Tabela Hash

- **Calcular a posição da chave na tabela baseado no valor da chave**

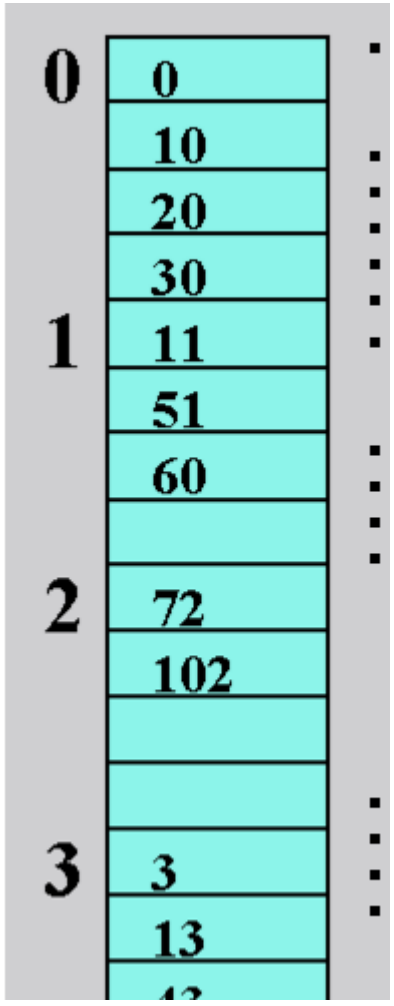


- **Posição na tabela pode ser acessada diretamente**
- **Não há necessidade de testes**
- **Complexidade é reduzida para O(1)**
  - Busca sequencial: complexidade é O(n)
  - Busca binária: complexidade é O(lg n)

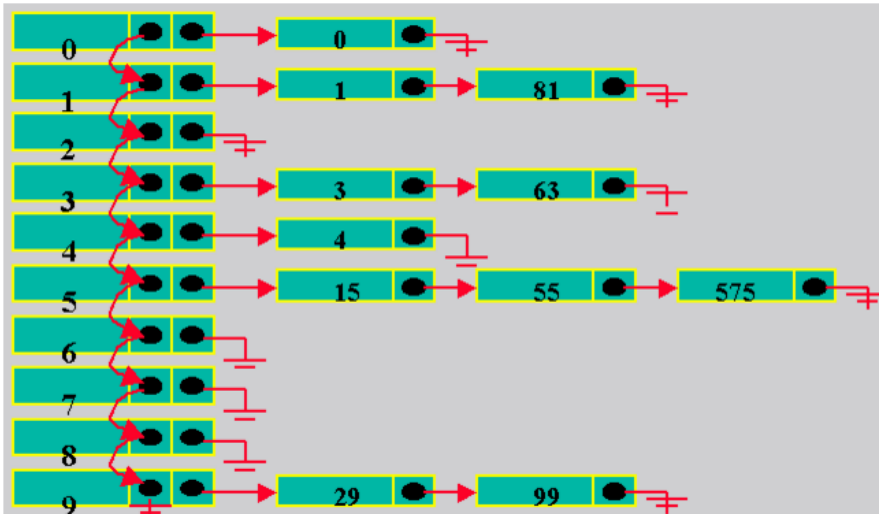
Hash

- **Filosofias (2 tipos):**
  - (1) **hashing fechado** ou de ***endereçamento aberto***
  - **(2) hashing aberto** ou ***encadeado***.

1



2

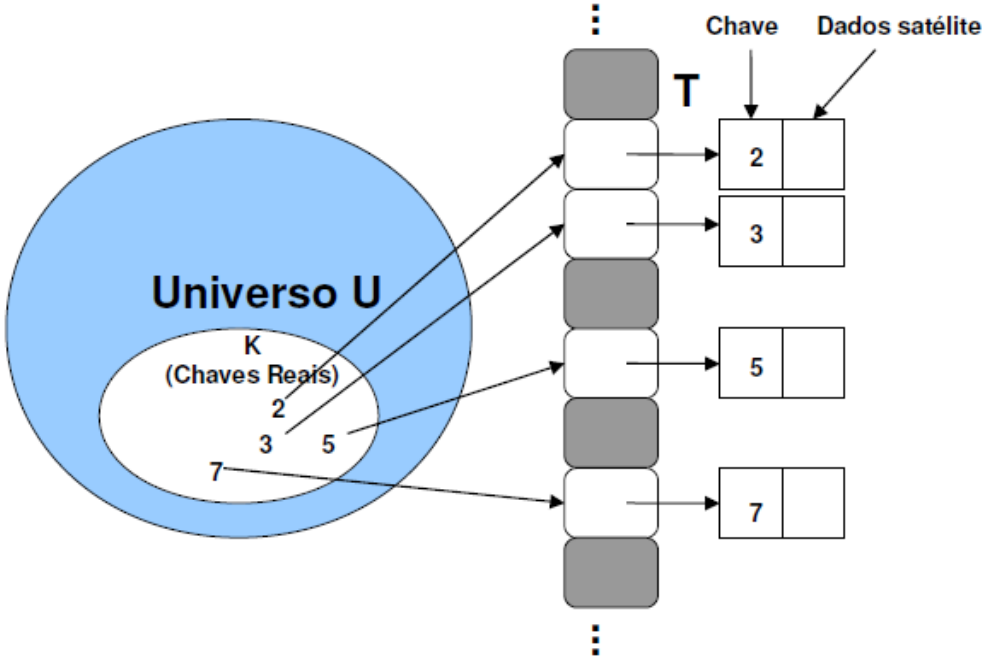


Hash de endereçamento direto (vetor)

- **Dado um universo de chaves U**
  - Se a cardinalidade de U é pequena, o endereçamento direto é uma técnica simples que funciona bem!
- **Suponha que uma dada aplicação necessite de um conjunto dinâmico onde cada elemento tenha uma chave definida a partir do universo**
  - U = {0, 1, ..., m1}
  - Onde m não é muito grande, e não há dois elementos com a mesma chave
  - Para tanto vamos utilizar uma **tabela de endereçamento direto T[0..m1]**

Hash de endereçamento direto

- **Cada slot, ou posição, de T corresponde a uma chave no universo U**
  - A posição k aponta para um elemento no conjunto com chave k.
    - Se o conjunto não contém nenhum elemento com chave k, então *T[k] = NULL*



Hash de endereçamento direto

- **Para estas tabelas hash também é possível armazenar o dado satélite na própria tabela T**
  - Isso irá economizar memória, visto que a chave pode ser encarada como o índice da tabela T.
  - Contudo, se feito isto, algo deve ser realizado para saber se a posição está vazia

Operações em uma Tabela Hash

- **DIRECT ADDRESS SEARCH(T,k)**
  - return T[k]
- **DIRECT ADDRESS INSERT(T,k)**
  - T[chave[k]] ⟵ x
- **DIRECT-ADDRESS-DELETE(T,k)**
  - T[chave[k]] ⟵ NULL

- **Observe que todas estas operações são rápidas: é necessário apenas tempo O(1)**

Tabela Hash

- **Qual a dificuldade do endereçamento direto?**
  - Se o universo U é muito grande, o armazenamento de uma tabela T de tamanho |U| pode ser impraticável!
  - Além do que a quantidade de chaves reais utilizadas pode ser muito menor que |U| implicando que a maior parte de T seria desperdiçada.
    - Para este caso, uma tabela hash exige muito menos espaço de endereçamento do que uma tabela de endereçamento direto!
    - Os requisitos de armazenamento são Θ(|k|)

Exemplo Hash Direto

- **Exemplo: Armazenamento dos empregados de uma empresa em um array**
  - Quantas posições o array terá que ter?
  - Considerando que o número máximo de funcionários possíveis é igual à quantidade de CPFs diferentes que podem existir.
  - Quantas posições o array deverá ter?

Exemplo Hash Direto

- **Exemplo: Empregados identificados por CPF**

```
Class Empregado{
   int cpf;
   char nome[80];
   char end[120];
   ...
}
```

- **CPF como índice: 1 bilhão de entradas 000 000 000 a 999 999 999**
- **emp vet[1000000000]**
- **Reservar espaço para o total de possíveis empregados**
- **Mas, se empresa possui apenas 500 empregados, temos desperdício enorme de memória.**

Tabela Hash

- **Em uma tabela hash, um elemento com chave k é armazenado na posição *h(k)***
  - Ou seja, *função **hash h**, que é utilizada para calcular* a posição do elemento a partir da chave k.

- **Função Hash**
  - Mapeamento do Universo U de chaves nas posições da tabela hash T[0..m1]

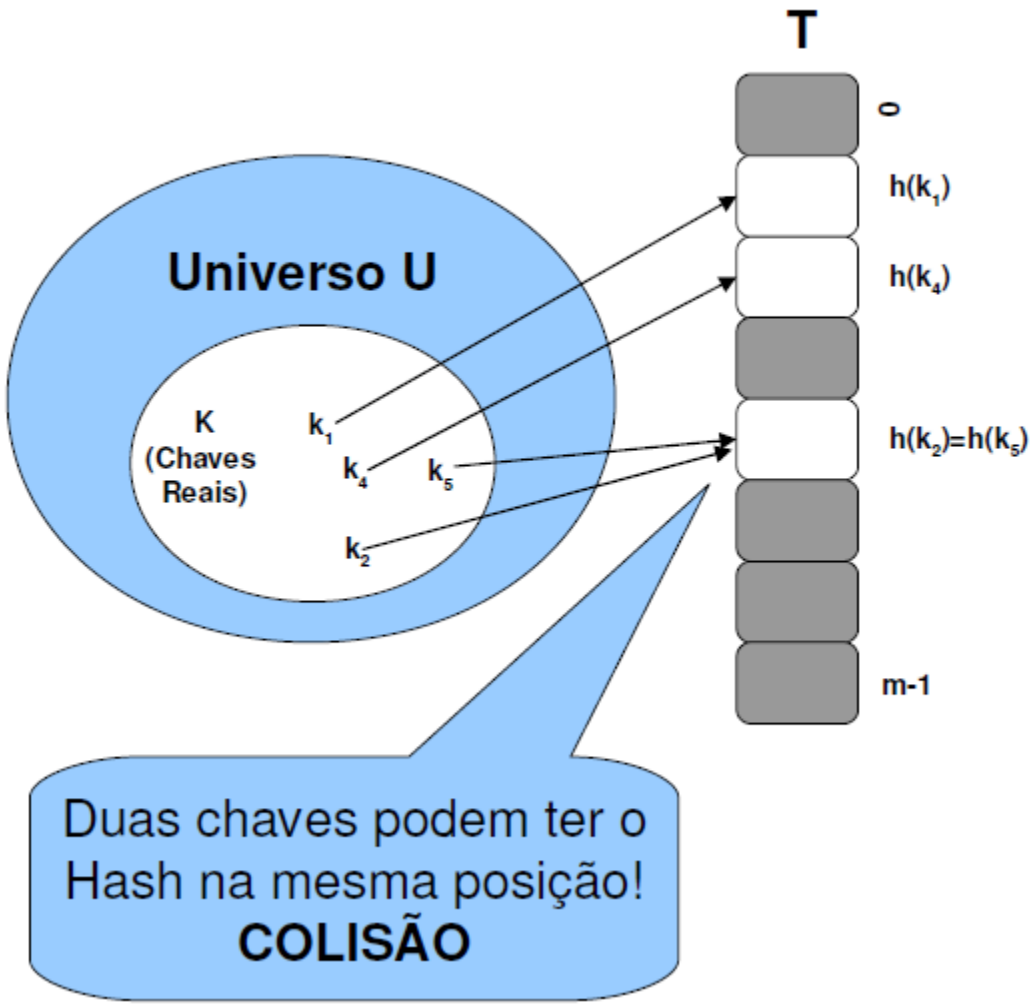
    - $h : U \rightarrow \{0,1, \ldots , m-1\}$

Tabela Hash

- É dito que um elemento com chave **k** *efetua o hash para* a posição *h(k)*
- É dito também que *h(k) é o* ***valor hash*** da chave *k.*
- *Função mais usada:*
  - *h(k) = k mod m*

  k = chave

  m = endereço da tabela



Hash implementada com endereçamento aberto

- Método aplicado quando o número de chaves a serem armazenadas é reduzido e as posições vazias são utilizadas para o tratamento de colisões.
- Quando uma chave k é endereçada na posição h(k) e esta já está ocupada, outras posições vazias são procuradas.
- A busca de uma posição livre pode ser feita por **tentativa linear**

Tentativa Linear

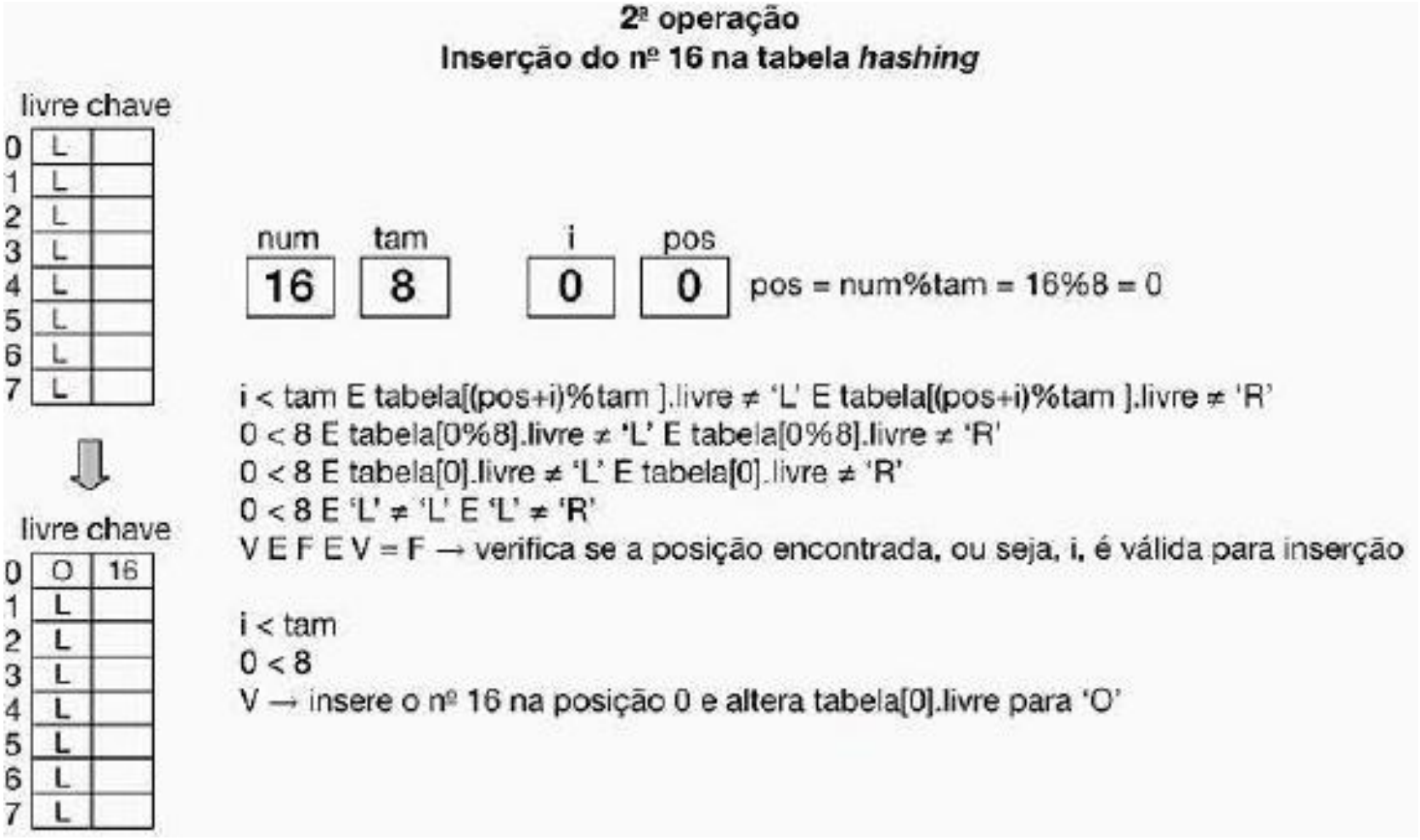
- **Quando uma chave x deve ser inserida e há uma colisão, usa-se a função**
  - $h'(x) = (h(x) + j) \bmod m$
  - para $1 \leq j \leq m-1$,
  - sendo que $h(x) = x \bmod m$.

- **O objetivo é armazenar a chave no endereço consecutivo h(x)+1, h(x)+2, …, até encontrar uma posição vazia.**

Exemplo

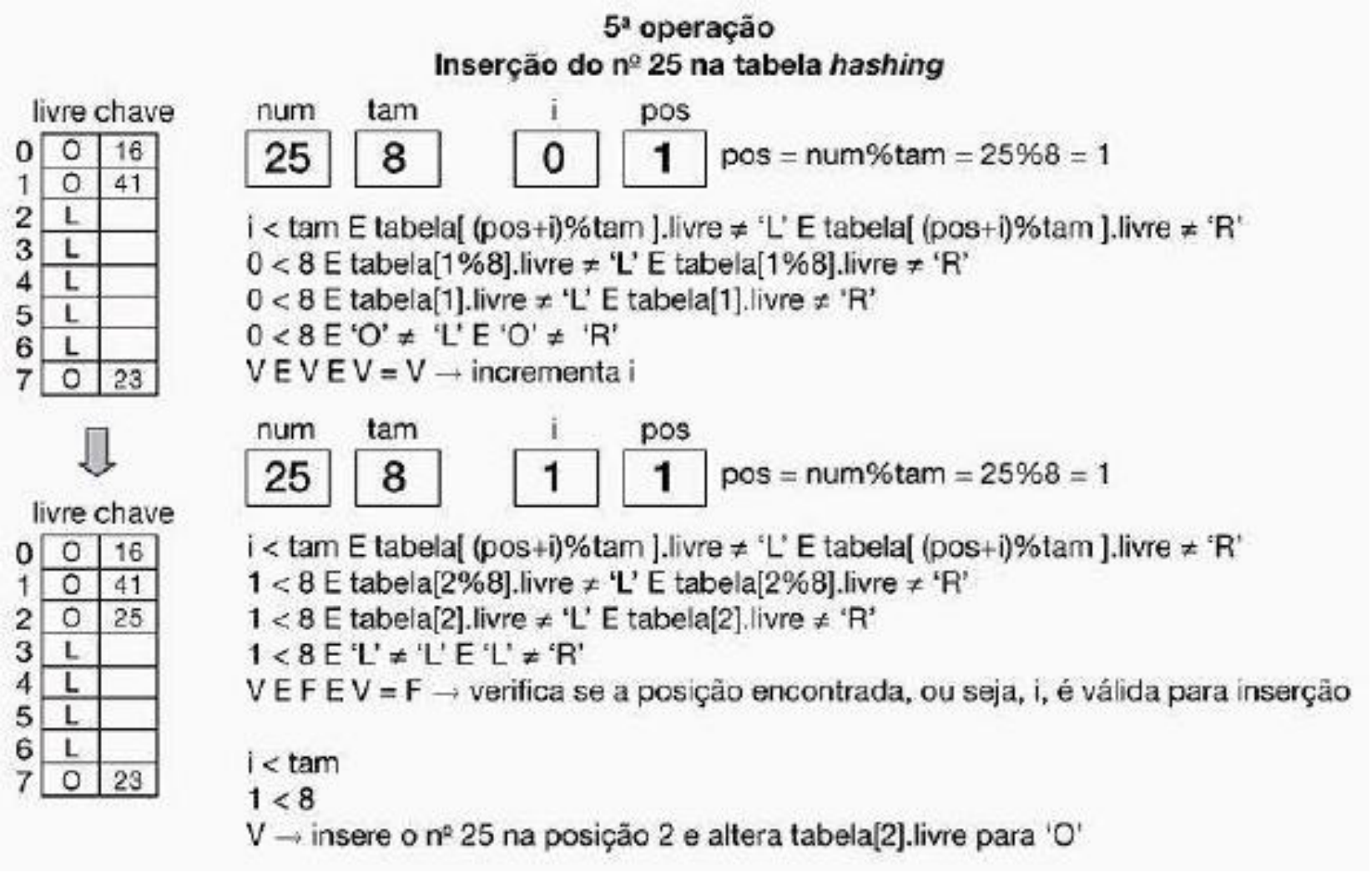
FONTE: [ASCENCIO, 2010]

Exemplo



FONTE: [ASCENCIO, 2010]

Exemplo



FONTE: [ASCENCIO, 2010]

Exemplo

7ª operação
Remoção do nº 41 da tabela *hashing*

	livre	chave
0	O	16
1	O	41
2	O	25
3	O	39
4	L	
5	L	
6	L	
7	O	23

num	tam	i	pos
41	8	0	1

pos = num%tam = 41%8 = 1

i < tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].chave ≠ num
0 < 8 E tabela[1%8].livre ≠ 'L' E tabela[1%8].chave ≠ 41
0 < 8 E tabela[1].livre ≠ 'L' E tabela[1].chave ≠ 41
0 < 8 E 'O' ≠ 'L' E 41 ≠ 41
V E V E F = F → verifica se na posição (pos+i)%tam está a chave e se esta já foi removida

	livre	chave
0	O	16
1	R	41
2	O	25
3	O	39
4	L	
5	L	
6	L	
7	O	23

tabela[(pos+i)%tam].chave = num E tabela[(pos+i)%tam].livre ≠ 'R'
tabela[1%8].chave = num E tabela[1%8].livre ≠ 'R'
tabela[1].chave = num E tabela[1].livre ≠ 'R'
41 = 41 E 'O' ≠ 'R'
V E V = V → remove o nº 41 da posição 1 alterando tabela[1]. livre para 'R'

FONTE: [ASCENCIO, 2010]

Exem

9ª operação
Remoção do nº 25 da tabela *hashing*

	livre	chave
0	O	16
1	R	41
2	O	25
3	O	39
4	L	
5	L	
6	L	
7	R	23

num	tam	i	pos
25	8	0	1

pos = num%tam = 25%8 = 1

i < tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].chave ≠ num
0 < 8 E tabela[1%8].livre ≠ 'L' E tabela[1%8].chave ≠ 25
0 < 8 E tabela[1].livre ≠ 'L' E tabela[1].chave ≠ 25
0 < 8 E 'R' ≠ 'L' E 41 ≠ 25
V E V E V = V → incrementa i

	livre	chave
0	O	16
1	R	41
2	R	25
3	O	39
4	L	
5	L	
6	L	
7	R	23

num	tam	i	pos
25	8	1	1

pos = num%tam = 25%8 = 1

i < tam E tabela[(pos+i)%tam].livre ≠ 'L' E tabela[(pos+i)%tam].chave ≠ num
1 < 8 E tabela[2%8].livre ≠ 'L' E tabela[2%8].chave ≠ 25
1 < 8 E tabela[2].livre ≠ 'L' E tabela[2].chave ≠ 25
1 < 8 E 'O' ≠ 'L' E 25 ≠ 25
V E V E F = F → verifica se na posição (pos+i)%tam está a chave e se esta já foi removida

tabela[(pos+i)%tam].chave = num E tabela[(pos+i)%tam].livre ≠ 'R'
tabela[2%8].chave = num E tabela[2%8].livre ≠ 'R'
tabela[2].chave = num E tabela[2].livre ≠ 'R'
25 = 25 E 'O' ≠ 'R'
V E V = V → remove o nº 25 da posição 2 alterando tabela[2].livre para 'R'

FONTE: [ASCENCIO, 2010]

Referências

Básica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. Editora Campus, 2002
- Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Cengage Learning, 2004.

Complementar

- TENENBAUM, Aaron; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C. São Paulo: Makron Books, 1995. ISBN: 9788534603485*
- ASCENCIO, Ana Fernanda Gomes; ARAUJO, Graziela Santos. Estruturas de Dados: Algoritmos, análise da complexidade e implementações em Java e C/C++. Pearson Prentice Hall, 2010
- DROZDEK, Adam. Adam Drozdek. Data Structures and Algorithms in Java. 2. Cengage Learning. 2004. 2. Cengage Learning. 2004
- GOODRICH, Michael T. Estruturas de dados e algoritmos em java. 4 ED. Porto Alegre: Bookman, 2007. 600.
- SKIENA, Steven S.. **The Algorithm Design Manual**. 2. Springer-Verlag. 2008

Perguntas....