Tabelas *Hash*

Algoritmos e Estruturas de Dados
Prof. Dr. Luciano Demétrio Santos Pacífico
{uciano.pacifico@ufrpe.br}

UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

Conteúdo

- **Introdução**
- **Tratamento de Colisões**
- **Busca, Inserção e Remoção**

Introdução

UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

Tabelas *Hash*

- Suponha que você pudesse criar um vetor de dados onde qualquer item pudesse ser localizado através de acesso direto.

- Isso seria ideal em aplicações do tipo *Dicionário*, onde gostaríamos de fazer consultas aos elementos da tabela em tempo constante, ou seja O(1).

Tabelas *Hash*

- Exemplo: A linguagem de programação C faz uso da Tabela ASCII.

- Como essa tabela é composta por 256 símbolos, apenas 256 bytes seriam necessários para o armazenamento de todos os seus símbolos na memória.

- Cada símbolo da tabela ASCII pode ser acessado diretamente em C.

Tabelas *Hash*

- Um problema ocorre quando o número de possíveis chaves é muito grande, o que faria com que este vetor tivesse um tamanho enorme.

  - Ex.: Em uma tabela de nomes com 32 caracteres por nome, teríamos $26^{32} > 16^{32} = (2^4)^{32} = 2^{128}$ possíveis elementos.

- Haveria também o *desperdício de espaço*, pois a cada execução somente uma pequena fração das chaves estarão de fato presentes.

Tabelas *Hash*

- O objetivo de uma ***Tabela Hash*** é mapear um número enorme de chaves em um espaço de inteiros relativamente pequeno.

- Isso é feito através de uma função chamada ***hash function***.

- O inteiro gerado pela *hash function* é chamado ***hash code*** e é usado para encontrar a localização do item.

Exemplo

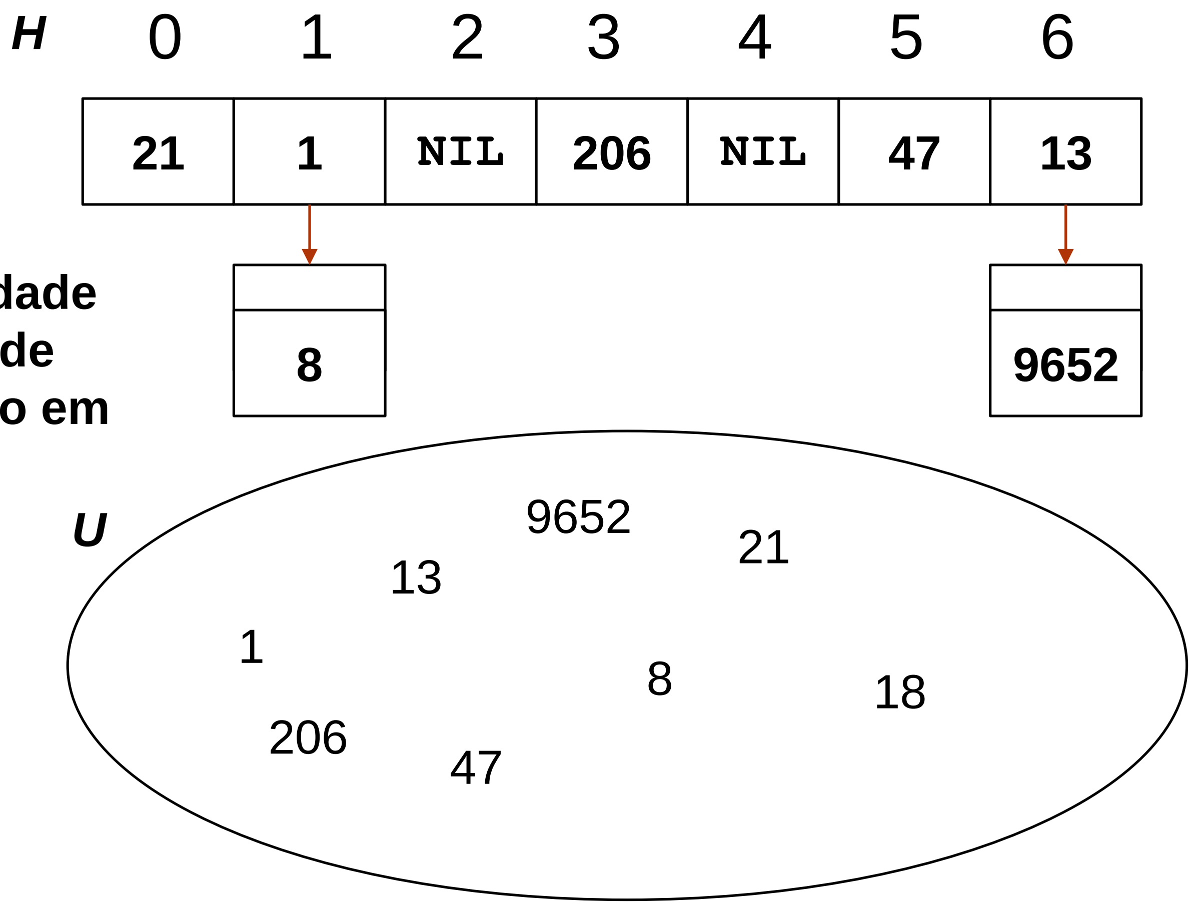
- Suponha que o espaço de chaves seja o conjunto de inteiros de quatro dígitos.

- Deseja-se traduzi-los no conjunto de chaves {0, 1, ..., 6}.

- A *Hash Function* adotada é:

$$h(k) = k \; mod \; 7$$

Exemplo



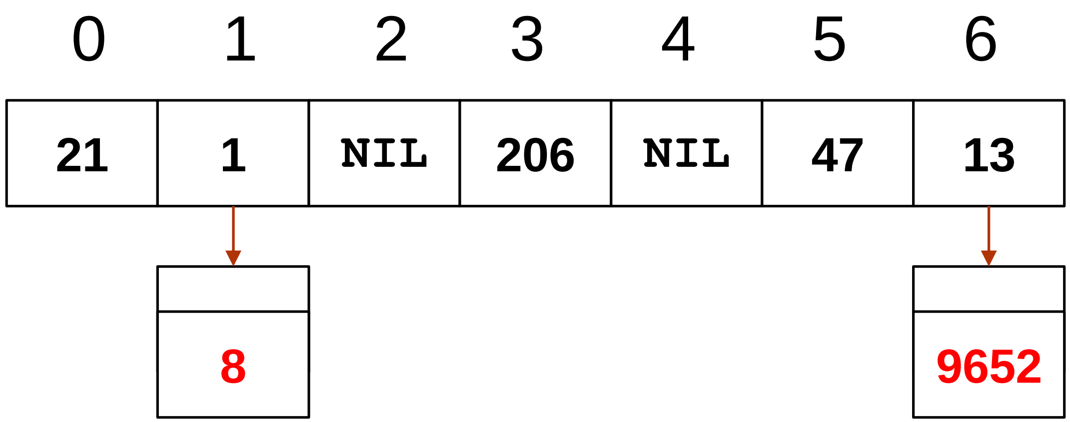
Complexidade na ordem de $\theta(H)$, e não em $\theta(U)$.

Tratamento de Colisões

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Colisões

- No exemplo anterior dizemos que entre as chaves 13 e 9652 (assim como entre 1 e 8) ocorreu uma *colisão*, isto é estas duas chaves geraram o mesmo *hash code*, ou seja, foram mapeadas no mesmo índice.



Colisões

- Problema: se o número de colisões for muito grande, a complexidade das operações em uma Tabela *Hash* passa a ser O(*n*).

- O número de colisões depende da *hash function* escolhida.

- Seria desejável à escolha de uma *hash function* **injetiva**, de forma a evitar colisões, mas como isso é muito difícil, há vários esquemas para trabalhar a ocorrência de colisões.

Colisões

- Há duas grandes classes de abordagens:

1. ***Closed Address Hashing*** (endereçamento fechado);

2. ***Open Address Hashing*** (endereçamento aberto).

Closed Address Hashing

- *Closed Address Hashing* ou ***hashing* encadeado** é a forma mais simples de tratamento de colisão.

- Cada entrada $H[i]$ da Tabela *Hash* é uma **Lista Ligada** (**Lista Encadeada**), cujos elementos têm *hash code* $h(k)$.

- Para inserir um elemento na tabela:
  1. Compute o seu *hash code* $h(k)$;
  2. Insira o elemento na lista ligada $H[h(k)]$.

Closed Address Hashing

- Problema: Embora uma *hash function* bem escolhida promova um bom balanceamento, não se pode garantir que as listas terão tamanhos próximos.

- Seria possível substituir a lista ligada por estruturas mais eficientes de busca, como **árvores balanceadas** (vide aula sobre **AVLs**), mas isso não se faz na prática.

Open Address Hashing

- No *open address hashing*, ou **endereçamento aberto**, a estratégia adotada procura **guardar todas as chaves na tabela propriamente dita**, mesmo quando ocorre colisão.

- Cada posição *H*[*i*] contém uma **chave**, ao invés de um link.

- Tem a vantagem de não usar espaço extra.

- Em caso de colisão, **um novo endereço é computado**. Esse processo é chamado ***rehashing***.

Open Address Hashing

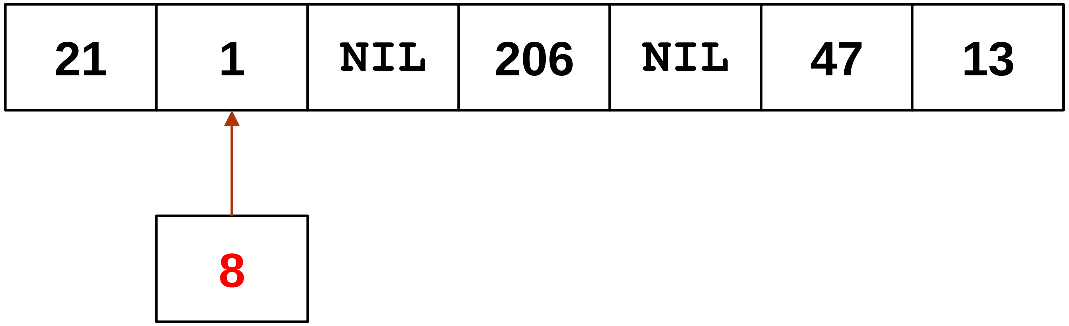
- A forma mais simples de *rehashing* é ***linear probing***.

- Seja o *hash code* da chave k dado por $h(k) = i$, então a posição na tabela H ocupada pela chave k é dada por:

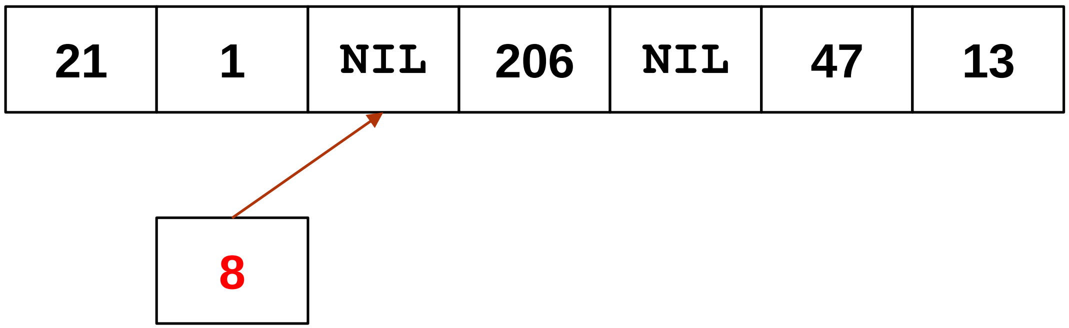
$$rehash(i, j) = (i+j) \text{ mod } m$$

onde j é o número de colisões.

Exemplo



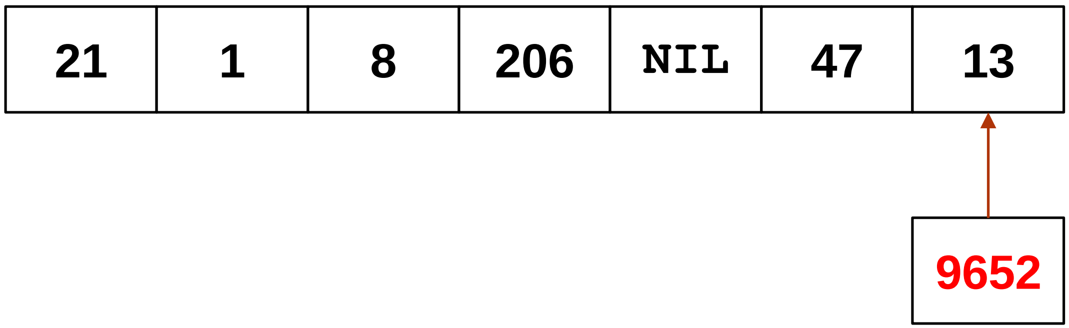
Exemplo



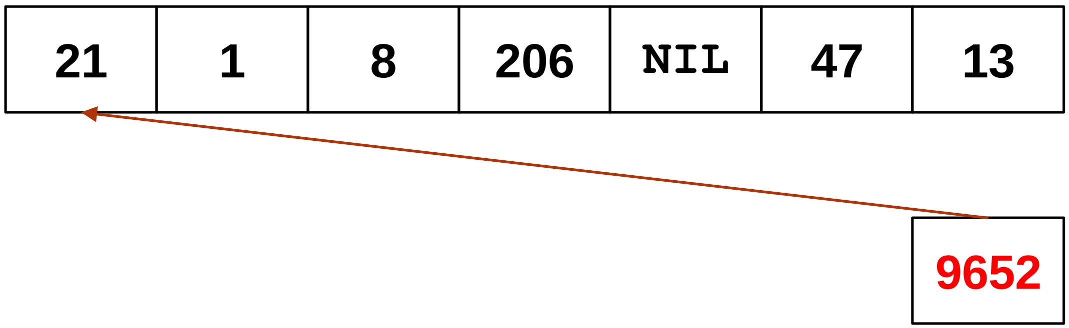
Exemplo

21	1	8	206	NIL	47	13

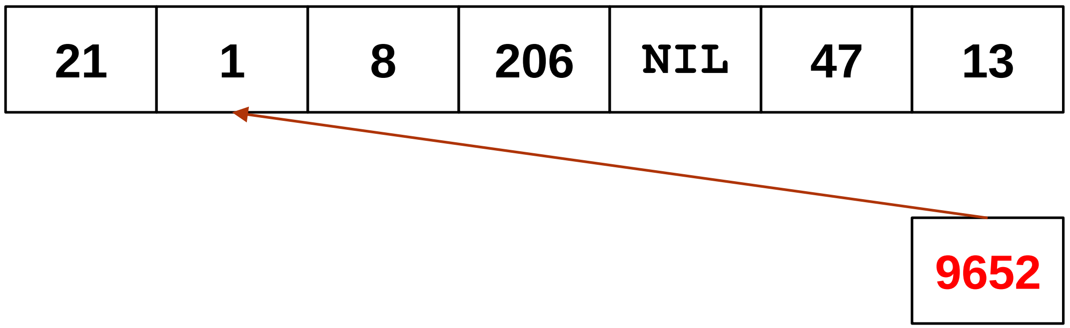
Exemplo



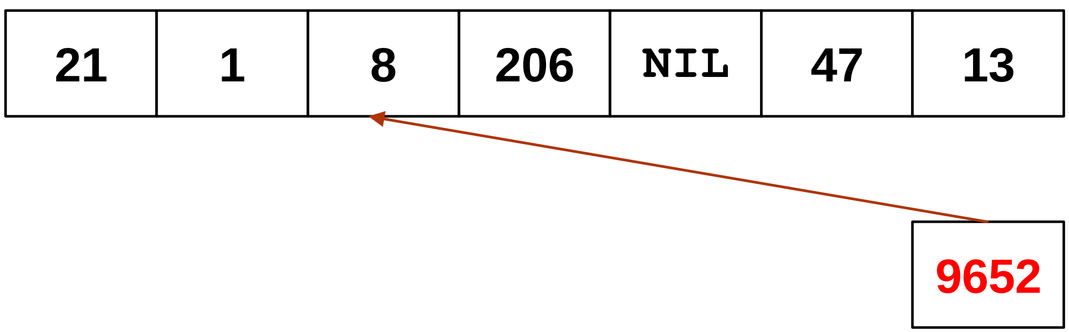
Exemplo



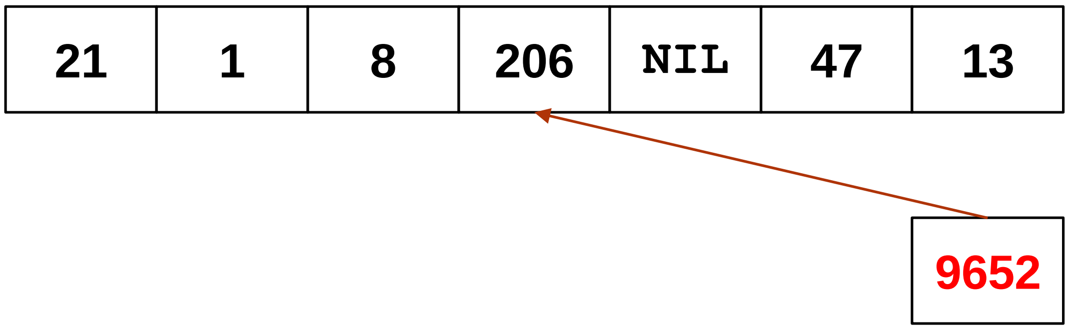
Exemplo



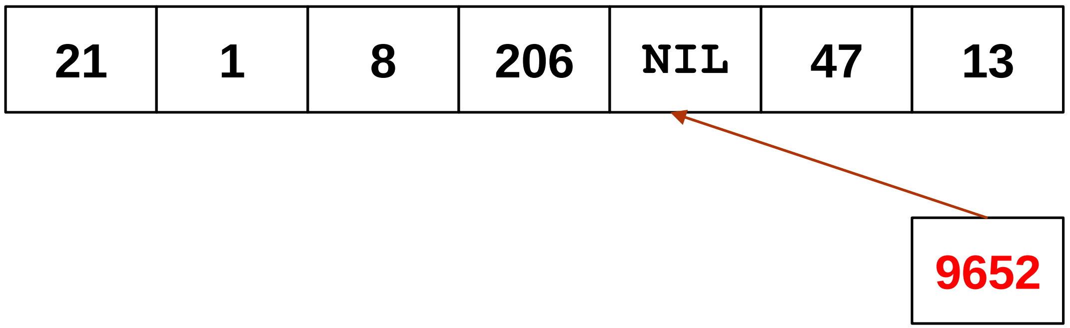
Exemplo



Exemplo



Exemplo



Exemplo

21	1	8	206	9652	47	13

Open Address Hashing

- *Rehashing* por *Linear Probing* pode trazer sérios problemas de colisão se houver uma alta taxa de ocupação na Tabela *Hash* (**problema do *agrupamento primário***).

- Para um bom desempenho é importante manter a taxa de ocupação da tabela próxima a 0,5 (50% do espaço).

Open Address Hashing

- Podemos observar que:

1. É possível que uma posição *i* da tabela *hash* já esteja ocupada com alguma chave cujo *hash code* é diferente de *h*(*k*).

2. *Rehashing* por *linear probing* não depende do valor da chave *k*.

Quadratic Probing

- No ***quadratic probing***, uma **função quadrática** em relação ao número de colisões *j* é usada para o cálculo da posição que será ocupada pela chave *k* na tabela *H*.

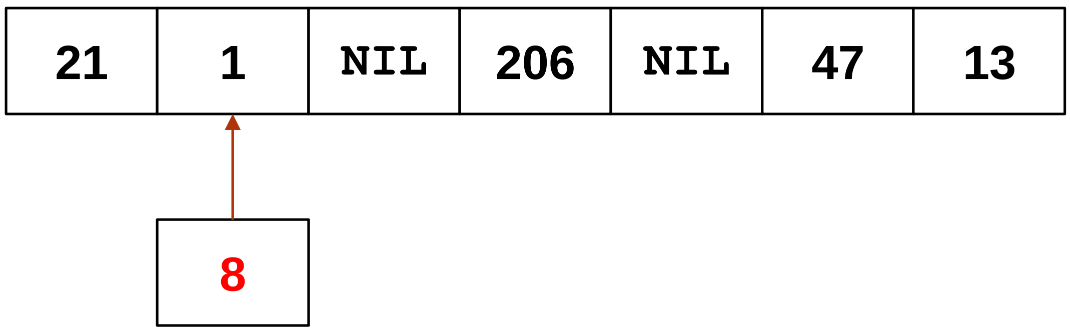
$$rehash(i, j) = (i + c_1 j + c_2 j^2) \bmod m$$

onde c_1 e c_2 são constantes não nulas.

- Apresenta como desvantagem o fato de que se duas chaves *k* e *l* forem tais que $h(k) = h(l)$, então a função $rehash(i, j) = rehash(l, j)$ (**problema do *agrupamento secundário***).

Exemplo

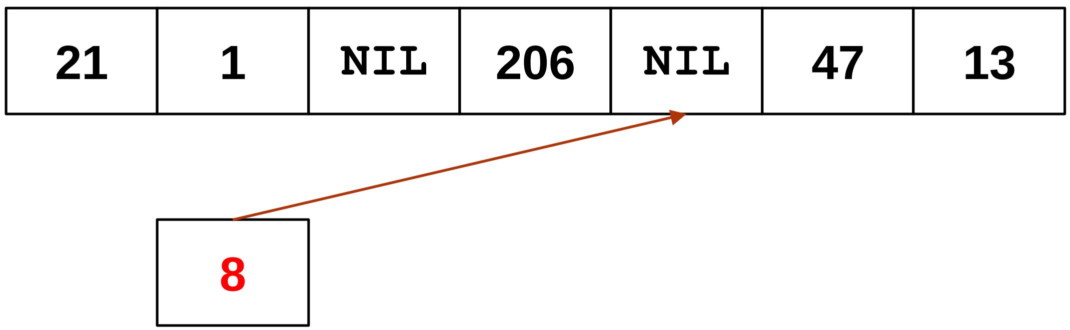
$$rerash(i, j) = (i + 2j + j^2) \bmod 7$$



$$i = 8 \bmod 7 = 1$$
$$rerash(1, 0) = 1$$

Exemplo

$$rerash(i, j) = (i+2j+j^2) \bmod 7$$



$i = 8 \bmod 7 = 1$

$rerash(1, 1) = 4$

Exemplo

$$\textit{rerash}(i, j) = (i+2j+j^2) \bmod 7$$

21	1	NIL	206	8	47	13

Exemplo

$$rerash(i, j) = (i + 2j + j^2) \bmod 7$$

21	1	`NIL`	206	8	47	13

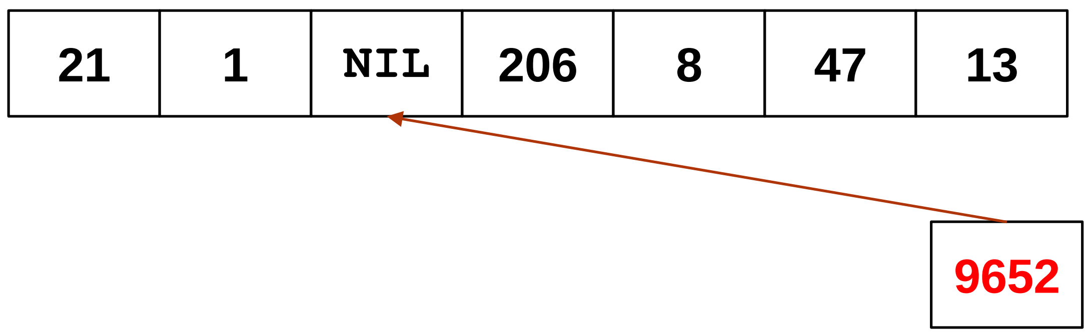
9652

i = 9652 mod 7 = 6

$rerash(6, 0) = 6$

Exemplo

$$rerash(i, j) = (i + 2j + j^2) \bmod 7$$



$$i = 9652 \bmod 7 = 6$$

$$rerash(6, 1) = 2$$

Exemplo

$$rerash(i, j) = (i+2j+j^2) \bmod 7$$

21	1	9652	206	8	47	13

Double Hashing

- ***Double Hashing*** é uma forma eficiente de fazer o *rehashing*.

- Neste método, uma função que leva a chave k em consideração é usada.

$$rehash(k, j) = (h(k)+jg(k)) \bmod m$$

- Para que a função auxiliar $g(k)$ seja efetiva, é preciso que $g(k)$ e m sejam relativamente primos.

Busca, Inserção e Remoção

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Tabelas Hash – Estruturas Básicas

- Por questões de facilidade de entendimento, definiremos duas Estruturas de Dados diferentes, de acordo com o tipo de tratamento de colisões adotado.

```
registro NoTabelaHashOpen
    chave:inteiro
```

```
registro TabelaHashClosed
    tamanho:inteiro,
    tabela:Array<ListaDuplaEnc>
```

```
registro TabelaHashOpen
    tamanho:inteiro,
    tabela:Array<NoTabelaHashOpen>
```

- Optamos por uma Lista Duplamente Encadeada, mas uma Lista Simplesmente Encadeada teria o mesmo efeito, neste caso.

Busca, Inserção e Remoção

- Como estamos indexando vetores de 1 até *n*, consideraremos que **o resultado da *hash function* adotada já estará acrescido de 1**.

- Para o caso de tabelas *hash* com *closed address hashing*, as operações de busca, inserção e remoção dependem unicamente do código *h*(*k*) da chave.

- Após encontrar o *hash code* do elemento, a operação ocorrerá como em uma **Lista Linear Encadeada** (vide aula sobre **Listas Encadeadas**).

- Como a Tabela *Hash* pode degenerar, o custo de pior caso destas operações pode ser da ordem de O(*n*).

Busca, Inserção e Remoção – *Closed Address Hashing*

```
//x -> chave do nó buscado
//H -> TabelaHashClosed contendo os dados
procedimento buscarTabelaHashClosed(x, H)
    k = hashFunction(x, H.tamanho) //hashFunction escolhida
    L = H.tabela[k]
    retorne buscarListaDuplaEnc(x, L)//vide Aula Listas Encadeadas
```

```
//X -> nó a ser inserido
//H -> TabelaHashClosed contendo os dados
procedimento inserirTabelaHashClosed(X, H)
    k = hashFunction(X.chave, H.tamanho) //hashFunction escolhida
    L = H.tabela[k]
    inserirListaDuplaEnc(X, L)//vide Aula Listas Encadeadas
```

```
//x -> chave do nó a ser removido
//H -> TabelaHashClosed contendo os dados
procedimento removerTabelaHashClosed(x, H)
    k = hashFunction(x, H.tamanho) //hashFunction escolhida
    L = H.tabela[k]
    retorne removerListaDuplaEnc(x, L)//vide Aula Listas Encadeadas
```

Open Address Hashing com *Linear Probing* – Busca

- Embora usemos o hash code gerado para a chave como início da busca, não há garantias de que se o objeto estiver na tabela, o encontraremos nessa posição.
- Teremos de vasculhar toda a tabela, tornando o custo dessa operação na ordem de O(*n*).

```
//x -> chave do nó buscado
//H -> TabelaHashOpen contendo os dados
procedimento buscarTabelaHashOpen(x, H)
    pt = NIL
    k = hashFunction(x, H.tamanho)
    se (H.tabela[k] != NIL) e (H.tabela[k].chave == x)
        pt = H.tabela[k]
    senão
        i = mod(k, H.tamanho) + 1 //mod é a operação de resto inteiro
        enquanto i != k //se i == k, toda a tabela já foi vasculhada
            se (H.tabela[i] != NIL) e (H.tabela[i].chave == x)
                pt = H.tabela[i]
                i = k    //força saída do laço
            senão
               i = mod(i, H.tamanho) + 1
    retorne pt
```

Open Address Hashing com *Linear Probing* – Inserção

- Quando a Tabela *Hash* faz uso de open *address hashing* com *linear probing*, caso a posição correspondente à chave *x* esteja ocupada, usa-se o *rehashing*.

```
//X -> nó a ser inserido
//H -> TabelaHashOpen contendo os dados
procedimento inserirTabelaHashOpen(X, H)
    indice = 0
    k = hashFunction(X.chave, H.tamanho)
    se H.tabela[k] == NIL
        H.tabela[k] = X
        indice = k
    senão
        i = mod(k, H.tamanho) + 1 //mod é a operação de resto inteiro
        enquanto i != k //se i == k, toda a tabela já foi vasculhada
            se H.tabela[i] == NIL
                H.tabela[i] = X
                indice = i
                i = k    //força saída do laço
            senão
                i = mod(i, H.tamanho) + 1
        se indice == 0
            imprimir("Overflow")
    retorne indice
```

Open Address Hashing com *Linear Probing* – Remoção

```
//x -> chave do nó a ser removido
//H -> TabelaHashOpen contendo os dados
procedimento removerTabelaHashOpen(x, H)
    pt = NIL
    k = hashFunction(x, H.tamanho)
    se (H.tabela[k] != NIL) e (H.tabela[k].chave == x)
        pt = H.tabela[k]
        H.tabela[k] = NIL
    senão
        i = mod(k, H.tamanho) + 1 //mod é a operação de resto inteiro
        enquanto i != k //se i == k, toda a tabela já foi vasculhada
            se (H.tabela[i] != NIL) e (H.tabela[i].chave == x)
                pt = H.tabela[i]
                H.tabela[i] = NIL
                i = k    //força saída do laço
            senão
               i = mod(i, H.tamanho) + 1
        se pt == NIL
            imprimir("Nó " + x + " não existe!")
    retorne pt
```

Referências

- SZWARCFITER, J.; MARKENZON, L. Estruturas de Dados e seus Algoritmos, 3ª ed. Rio de Janeiro: LTC, 2010.

- CORMEN, H. T.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms, 3rd ed., *Boston: MIT Press*, 2009.

- FEOFILOFF, Paulo. Algoritmos em Linguagem C. Editora Campus/Elsevier, 2009.

Tabelas *Hash*

Algoritmos e Estruturas de Dados

Prof. Dr. Luciano Demétrio Santos Pacífico

{uciano.pacifico@ufrpe.br}

UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO