



ESTRUTURA DE DADOS

AULA 6



Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

O objetivo desta aula é apresentar os conceitos que envolvem a estrutura de dados do tipo *hash*. Ao longo deste documento essa estrutura de dados será conceituada, e o seu funcionamento e a sua aplicabilidade serão apresentados. Investigaremos também o que são funções *hashing* e veremos algumas das mais comuns. Veremos ainda possibilidades de implementação da tabela *hash* com tratamento para colisões. Também estudaremos:

- Implementação com endereçamento aberto (linear e quadrática);
- Implementação com endereçamento em cadeia.

Conceitos já conhecidos, como análise assintótica, vetores, recursividade, listas encadeadas, bem como conceitos básicos de programação estruturada e manipulação de ponteiros, aparecerão de forma recorrente ao longo de toda esta aula.

Todos os códigos apresentados ao longo do documento estarão na forma de pseudocódigos. Para a manipulação de ponteiros e endereços em pseudocódigo, será adotada a seguinte nomenclatura:

- Para indicar o endereço da variável, será adotado o símbolo **(->]** antes do nome da variável. Por exemplo: $px = (->]x$. Isso significa que a variável px recebe o endereço da variável x .
- Para indicar um ponteiro, será adotado o símbolo **[->)** após o nome da variável. Por exemplo: $x(->]: \text{inteiro}$. Isso significa que a variável x é uma variável do tipo ponteiro de inteiros.

TEMA 1 – HASHS: DEFINIÇÕES

Para entendermos o que são *hashs* e sua aplicabilidade, vamos primeiro compreender qual é a sua utilidade. Para isso, vamos imaginar que você trabalha para o Instituto Brasileiro de Geografia e Estatística (IBGE) e está implementando um código para catálogo de dados de diferentes estados brasileiros. Dentre esses dados, você precisará armazenar muitas informações, como a capital do estado, população total, lista de cidades, PIB do estado, índice de criminalidade, nome do governador, dentre inúmeros outros dados.

Você começa a armazenar essas informações numa estrutura de dados unidimensional (vetor) e percebe, ao realizar testes, que necessita de uma maneira fácil de localizar cada estado dentro da estrutura de dados.

Para isso, resolve adotar a sigla de cada estado como uma palavra-chave para as buscas. Desse modo, cada posição do vetor conterá a sigla do estado e todos os diversos dados coletados e que você cadastrou.

Para a inserção e manipulação dos dados nesse vetor, você adota uma implementação denominada *endereçamento direto*. Nela, cada estado inserido é colocado na primeira posição livre do vetor. Como iniciamos o vetor na posição zero, o primeiro estado será colocado nela. O segundo estado irá na posição um, o terceiro na posição dois, e assim sucessivamente até preencher o vetor. Todos os dados são inseridos na sequência em que foram cadastrados. Na Figura 1, temos um exemplo de um vetor de dimensão 10 com uma sigla inserida em cada posição sua, servindo como referência para os dados cadastrados.

Figura 1 – Vetor com dados de estados brasileiros inseridos com endereçamento direto

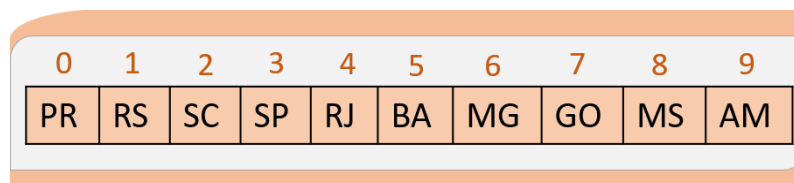


Diagrama de um vetor de 10 posições. Cada posição é representada por um índice numérico (0 a 9) e uma sigla de estado brasileiro. O vetor é visualizado como uma barra horizontal com uma borda arredondada e uma sombra.

0	1	2	3	4	5	6	7	8	9
PR	RS	SC	SP	RJ	BA	MG	GO	MS	AM

Em um vetor, o tempo de acesso a qualquer dado é sempre constante, e independe do tamanho do conjunto de dados. Temos a complexidade para acesso a qualquer posição do vetor como sendo $O(1)$.

Diferente do tempo de acesso, o tempo de busca de um dado no vetor não será constante quando utilizamos o endereçamento direto. Como já se sabe, os algoritmos de busca apresentam uma complexidade que depende do tamanho do conjunto de dados.

Relembrando, um algoritmo de busca sequencial apresenta complexidade $O(n)$ e a busca binária $O(\log n)$. Isso significa que, se quiséssemos encontrar os dados referentes ao estado da Bahia no vetor da Figura 1, precisaríamos busca pela sigla BA, aplicando um algoritmo de busca para reaver todos esses dados. Quanto maior o conjunto de dados de entrada, mais tempo levará para nossos dados serem localizados.

Outra possível implementação para nosso exemplo de cadastro de estados seriam as listas encadeadas, as quais, embora trabalhem com um conceito diferente de vetores e usufruam do uso de ponteiros para criar um encadeamento entre os dados, apresentam tempo de busca de um dado que é dependente do tamanho do seu conjunto de dados, uma vez que cada elemento da lista só conhece seu sucessor (lista simples) e em alguns casos seu antecessor também (lista dupla). A varredura pelos elementos da lista é necessária para a localização de um dado.

E se pudéssemos tornar esse tempo de busca aos dados sempre constante, com complexidade $O(1)$, e independente do tamanho do conjunto de entrada de dados, existiria uma solução para este problema? A resposta é sim. E, para isso, devemos utilizar uma estrutura de dados denominada de *hash*.

Voltemos ao exemplo de cadastro de dados de estados brasileiros em um vetor de dimensão 10. Na Figura 1, a inserção dos dados no vetor deu-se de forma incremental, iniciando-se na posição zero. No exemplo a seguir, vamos inserir os dados nesse mesmo vetor, mas utilizando uma outra abordagem.

Novamente, vamos adotar a sigla de cada estado como sendo um *valor-chave*, ou *palavra-chave*. A diferença agora é que, com base nessa chave, aplicaremos uma equação lógica e/ou matemática para definir uma posição de inserção no vetor. A essa função denominamos de *função hash*, ou *algoritmo de hash*.

Como sabemos de antemão que a sigla de todos os estados brasileiros é sempre constituída de dois caracteres alfanuméricos, vamos converter cada caractere para seu valor em decimal seguindo o padrão da tabela ASCII. Então vamos definir uma função *hash* como sendo a soma em decimal de ambas as letras, e dividir o resultado pelo tamanho de nosso vetor (dimensão 10). A posição de inserção no vetor será o resto dessa divisão.

Saiba mais

Para consultar a tabela ASCII, acesse:
ASCII Table and Description. Disponível em: <<http://www.asciitable.com/>>.
Acesso em: 6 mar. 2019.

Consideraremos ambos caracteres em letras maiúsculas. Por exemplo, para o estado do Paraná, sigla PR, o caractere ASCII da letra P é 80, e o da letra R é 82. A soma desses valores resulta em 162. Dividindo esse valor pelo

tamanho do vetor (dimensão 10), e obtendo somente o resto desta divisão, temos o valor 2, que será, portanto, a posição de inserção dos dados referentes ao estado do Paraná. Assim, mesmo que PR seja o primeiro dado a ser cadastrado no vetor, ele não será posicionado na posição zero do vetor, mas sim na posição dois.

De forma análoga, podemos calcular a posição no vetor do estado do Rio Grande do Sul, sigla RS. Temos R = 82 e S = 83. A soma de ambos valores será 165, e o resto da divisão por 10 resultará no valor 5, valor este que corresponde a posição de inserção deste dado no vetor.

Na Tabela 1, temos o cálculo da posição de inserção para alguns estados brasileiros utilizando a função *hash*.

Tabela 1 – Cálculo da posição de inserção para alguns estados brasileiros utilizando a função *hash* (1)

Sigla	ASCII (Dec)	ASCII (Dec)	Soma	Posição (Soma MOD 10)
PR	P (80)	R (82)	162	2
SC	S (83)	C (67)	150	0
RS	R (82)	S (83)	165	5
SP	S (83)	P (80)	163	3
RR	R (82)	R (82)	164	4
RJ	R (82)	J (74)	156	6
AL	A (65)	L (76)	141	1
DF	D (68)	F (70)	138	8
PE	P (80)	E (69)	149	9

A equação que generaliza os cálculos da Tabela 1 é apresentada na Equação 1, em que o termo *MOD* representa o resto da divisão entre ambos os valores.

$$Posição = (Char1_{ASCII} + Char2_{ASCII}) MOD Tamanho_Vetor \quad (1)$$

Na Figura 2, temos os dados calculados usando a função *hash* e agora inseridos nas suas respectivas posições.

Figura 2 – Vetor com dados de estados brasileiros inseridos utilizando uma função *hash*.

0	1	2	3	4	5	6	7	8	9
SC	AL	PR	SP	RR	RS	RJ	-	DF	PE

É interessante analisar que, alterando o tamanho de nosso vetor, podemos acabar alterando também a posição de inserção de cada valor. Por exemplo, se o vetor fosse de dimensão 12, a sigla PR seria posicionada no índice 6, e não no índice 2, conforme mostrado para um tamanho 10.

Na Figura 2, observe que, caso desejarmos encontrar os dados referentes ao estado do Rio de Janeiro (posição 6), se aplicarmos um algoritmo de busca, estamos sujeitos a uma complexidade atrelada ao tamanho do nosso vetor. Porém, podemos utilizar como recurso de busca a mesma função *hash* usada para inserir o dado e recalcular a posição de RJ no vetor aplicando a Equação 1. Assim, encontraremos a posição 6, bastando realizar o acesso em *Vetor*[6] com tempo constante. Desse modo, o tempo de busca passa a se tornar *independente do tamanho do vetor*, dependendo somente do tempo para realizar o cálculo matemático e lógico da função de *hash* sempre que necessário encontrar algum dado. Todo e qualquer valor desse vetor poderá ser encontrado com uma complexidade $O(1)$.

O vetor contendo os valores-chave é denominado de *tabela hashing*. Para cada palavra-chave, podemos ter a quantidade que necessitarmos de dados cadastrados referentes aquela chave, dados estes chamados de *dados satélites*.

Ao longo desta aula, iremos investigar um pouco mais sobre *hashs*. Veremos alguns tipos de funções *hash* bastante comuns, bem como tipos distintos de endereçamentos e implementações.

Acerca da aplicabilidade da estrutura de dados do tipo *hash*, a gama de aplicações é bastante grande. Citamos:

- Podemos manter um rastreamento de jogadas efetuadas por jogadores em jogos como xadrez, damas, ou diversos outros jogos com alta quantidade de possibilidades;
- Compiladores necessitam manter uma tabela com variáveis mapeadas na memória do programa. O uso de *hashs* é muito empregado para tal fim;
- Aplicações voltadas para segurança, como autenticação de mensagens e assinatura digital empregam *hashs*;
- A estrutura de dados base que permite as populares criptomoedas, como *bitcoin*, operarem são *hashs*. Elas trabalham com cadeias de *hashs* altamente complexas para manipular transações, oferecem segurança e descentralizar as operações.

TEMA 2 – FUNÇÕES HASH

Um exemplo de função *hash* pode ser aquele que leva em consideração o resto de uma divisão para definir a posição na estrutura de dados. Porém uma função *hash* não apresenta uma fórmula definida, e deve ser projetada levando-se em consideração o tamanho do conjunto de dados, seu comportamento e os tipos de dados-chave utilizados.

As funções *hash* são o cerne na construção das tabelas *hashing*, e o desenvolvimento de uma boa função de *hash* é essencial para que o armazenamento dos dados, a busca e o tratamento de colisões (assunto abordado no próximo tema) ocorram de forma mais eficiente possível. Uma boa função *hash*, em suma, deve ser:

- Fácil de ser calculada. De nada valeria termos uma função com cálculos tão complexos e lentos que todo o tempo que seria ganho no acesso a informação com complexidade $O(1)$, seria perdido calculando uma custosa função de *hash*;
- Capaz de distribuir palavras-chave o mais uniforme possível;
- Capaz de minimizar colisões. Os dados devem ser inseridos de uma forma que as colisões sejam as mínimas possíveis, reduzindo o tempo gasto resolvendo colisões e também reavendo os dados;
- Capaz de resolver qualquer colisão que ocorrer;

2.1 O método da divisão

Um tipo de função *hash* muito adotada é o método da divisão, em que dividimos dois valores inteiros e usamos o resto dessa divisão como a posição desejada.

Quando nossas palavras-chave são valores inteiros, dividimos o número pelo tamanho do vetor e usamos o resto dessa divisão como a posição a ser manipulada na tabela *hashing*. Caso uma palavra-chave adotada seja um conjunto grande de valores inteiros (como um número telefônico ou CPF por exemplo), poderíamos somar esses valores agrupados (em pares ou trios) e também dividir pelo tamanho do vetor, obtendo o resto da divisão.

Exemplificando, um número telefônico com 8 valores (como 9988-2233) pode ser quebrado em pares e somado para gerar um só valor:

$99 + 88 + 22 + 33 = 242$. Este valor é usado no método da divisão.

De modo geral, quando precisamos mapear um número de chaves em m espaços (como os de um vetor) pegando o resto da divisão entre ambos, chamamos isso de *método da divisão*, conforme apresentado na Equação 2. Esse método é bastante rápido, uma vez que requer unicamente uma divisão.

$$h(k) = k \text{ MOD } m \quad (2)$$

Exemplificando, no caso de um vetor de tamanho 12 ($m = 12$) e uma chave de valor 100 ($k = 100$), o resultado da função será $h(k) = 4$, ou seja, adotaríamos o quarto espaço para manipular esta chave na tabela.

Existem alguns valores de m que devem ser minuciosamente escolhidos para não gerar povoamentos de tabelas *hash* ruins. Por exemplo, utilizar 2 ou múltiplos de 2 para o valor de m tende a não ser uma escolha. Isso porque o resto da divisão por dois, ou qualquer múltiplo seu, sempre resultará em um dos *bits* menos significativos, gerando um número bastante elevado de colisões de chaves.

Em *hash* precisamos trabalhar com números naturais para definir as posições na tabela, uma vez que linguagens de programação indexam as estruturas de dados (como vetores e matrizes) usando esse tipo de dado numérico. Desse modo, precisamos que nossas chaves sejam também valores naturais. Caso não sejam, precisamos encontrar uma forma de transformá-las para que sejam.

Para chaves com caracteres alfanuméricos, podemos adotar a mesma Equação 2, fazendo pequenas adaptações. Convertamos os caracteres para números decimais seguindo uma codificação (tabela ASCII, por exemplo), somamos os valores, dividimos pelo tamanho do vetor e obtemos o resto da divisão como posição de inserção da palavra-chave na tabela *hashing* (Equação 3). Esse processo também foi visto no exemplo do Tema 1.

$$h(k) = \left(\sum k_{ASCII_Dec} \right) \text{ MOD } m \quad (3)$$

Adotar números primos, especialmente aqueles não muito próximos aos valores de potência de 2, tende a ser uma boa escolha para o tamanho do vetor

com palavras-chaves alfanuméricas. Por exemplo, suponhamos que temos 2000 conjuntos de caracteres para serem colocadas em uma tabela *hashing*. Por projeto, definiu-se que realizar uma busca, em média, em até três posições antes de encontrar um espaço vazio é considerado aceitável. Se fizermos $\frac{2000}{3} \cong 666$. Para o valor de m podemos adotar um número primo não muito próximo de um múltiplo de 2. Podemos usar o valor 701. Nossa função *hash* resultante seria $h(k) = k \text{ MOD } 701$.

Por fim, quando trabalhamos com valores-chave sendo conjuntos de caracteres, devemos tomar cuidado com palavras que contenham as mesmas letras, mas em ordens diferentes (anagrama). Por exemplo, uma palavra-chave com quatro caracteres chamada *ROMA* poderá gerar o mesmo resultado que uma palavra-chave chamada *AMOR*, pois os caracteres são os mesmos rearranjados de outra maneira. Uma função de *hash* deve ser cuidadosamente definida para tratar este tipo de problema caso ele venha a ser recorrente; caso contrário, teremos excessivas colisões.

2.2 O método da multiplicação

Esse método de construção de funções *hash* funciona da maneira que, primeiro, multiplicamos a chave k por uma constante A . Essa constante deve estar em um intervalo $0 < A < 1$ e extrair a fração de kA . Em seguida, multiplicamos esse valor por m e arredondamos o resultado para baixo (Equação 4).

$$h(k) = \lfloor m(kA \text{ MOD } 1) \rfloor \quad (4)$$

A Equação 4 é equivalente à escrita da Equação 5, em que $kA \text{ MOD } 1$ pode ser escrito como $kA - \lfloor kA \rfloor$, e representa a parte fracionária de kA .

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor \quad (5)$$

Esse método apresenta como desvantagem o fato de ser mais lento para execução em relação ao método da divisão, pois temos mais cálculos envolvidos no processo, porém tem a vantagem de que o valor de m não é crítico, não importando o valor escolhido.

Em contraponto ao método de divisão, normalmente adotamos um múltiplo de 2 para seu valor, devido à facilidade de implementação. A constante

A , embora possa ser qualquer valor dentro do intervalo $0 < A < 1$, é melhor com alguns determinados valores. Segundo Knuth (1998), um ótimo valor para essa constante é $A = \frac{\sqrt{5}-1}{2} \cong 0,618$. Exemplificando, se $k = 123456$, e $m = 16384$, e a sugestão para o valor de A dada por Knuth for seguida, teremos $h(k) = 67$.

2.3 Hashing universal

Considerando que uma chave k qualquer tem a igual probabilidade de ser inserida em qualquer uma das posições de um vetor, em um pior cenário seria possível que um conjunto de chaves a serem inseridas caiam sempre na mesma posição do vetor, caso utilizem a mesma função *hash* $h(k)$. Portanto a complexidade para a inserção nessa *hash* será $O(n)$. Podemos evitar esse tipo de problema escolhendo uma função *hashing* aleatoriamente dentro de um universo H de funções. A esta solução chamamos de *hashing universal*.

Na *hashing* universal, no início da execução de um algoritmo de inserção, sorteamos aleatoriamente uma função de *hash* dentro de uma classe de funções cuidadosamente desenvolvida para a aplicação desejada. A aleatoriedade evita que qualquer entrada de dado resulte no pior caso. É bem verdade que a aleatoriedade poderá nunca resultar em um caso perfeito, em que nenhuma colisão ocorre, mas teremos sempre uma boa situação média.

Uma classe H de funções de *hash* é considerada universal se o número de funções $h \in H$ for igual a $\frac{|H|}{m}$. A probabilidade de que $h(k_1) = h(k_2)$ ocorra será $\frac{1}{m}$ com a seleção aleatória. A prova matemática da *hashing* universal pode ser encontrada no livro do Cormen (2011).

Implementação:

Imaginemos um número primo p e um conjunto de valores $Z_p = \{0, 1, 2 \dots p-1\}$ e definimos que $Z_p^* = Z_p - \{0\}$, ou seja, é o conjunto Z_p excluindo o valor zero.

Podemos adotar uma classe de funções H que seja dependente deste número primo p e do seu conjunto Z_p (Equação 6):

$$h_{a,b}(k) = ((ak + b) \text{ MOD } p) \text{ MOD } m \quad (6)$$

em que $a \in Z_p^*$ e $b \in Z_p$. A variação das constantes a e b constituem diversas possibilidades para esta classe de funções dependente de um valor primo p .

Se assumirmos o número primo $p = 17$, um vetor de dimensão $m = 6$, e também $a = 3$ e $b = 4$, obtemos pela Equação 6 que $h_{3,4}(8) = ((3 \cdot 8 + 4) \text{MOD } 17) \text{MOD } 6 = 5$.

Existe outro método bastante conhecido de implementação de *hash* universal que emprega matrizes aleatórias. Suponha que você tem um conjunto de dados de entrada de tamanho b_1 bits e você deseja produzir palavras-chave de tamanho b_2 bits. Criamos então uma matriz binária aleatória de dimensão $M = b_1 \times b_2$. A função *hash* desta classe será, portanto, $h(k) = M \cdot k$.

Você deve considerar seus dados como sendo valores binários. Esses valores binários podem ser valores inteiros ou mesmo caracteres convertidos para valores binários. Por exemplo, suponha que você tem dados de 4 bits e precisa gerar chaves com 3 bits. Faremos uma matriz $M = 3 \times 4$. Uma possível matriz binária aleatória seria:

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix} \quad (7)$$

Assumindo um dado de entrada como sendo 1011, multiplicamos pela matriz aleatória e obtemos um resultado de 3 bits (Equação 8). A geração aleatória dessa matriz binária para cada nova chave caracteriza um conjunto de funções H que pode ser considerada uma *hash* universal, pois a possibilidade de uma matriz gerada ser igual a outra é bastante pequena.

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad (8)$$

TEMA 3 – TABELA *HASHING* DE ENDEREÇAMENTO ABERTO E TENTATIVA LINEAR

A tabela *hash* pode ser implementada de diferentes formas. A implementação impacta diretamente em como os dados são inseridos e no tratamento das colisões de *hashs*, assunto que será explorado neste tema.

Podemos desenvolver uma tabela *hashing* por meio de uma implementação utilizando endereçamento aberto. Nesse tipo de implementação, a tabela é um vetor de dimensão m e todas suas chaves são armazenadas no próprio vetor, da mesma forma que vimos no exemplo do Tema 1. O endereçamento aberto é bastante empregado quando o número de *hashs* a

serem armazenadas é pequeno se comparado com o tamanho do vetor (Cormen, 2012, p. 270).

Vamos, em um primeiro momento, entender como podemos criar a estrutura da *hash* em pseudocódigo. É possível implementá-la utilizando uma estrutura heterogênea do tipo registro, em que temos como campos um valor para ser usado como palavra-chave, podendo ser um inteiro ou alfanumérico, por exemplo. E temos também um campo que mantém o *status* daquela posição, representando se ele está livre, ocupado por uma chave ou se a chave daquela posição já foi removida.

A Figura 3 mostra o pseudocódigo exemplo para a implementação da *hash* e de um menu principal com inserção e remoção nesta *hash*. Vejamos:

- *Linhas 1-5*: registro com implementação da estrutura de dados *hash*;
- *Linha 9*: declaração do vetor que usará o registro como tipo de dado, representando o endereçamento aberto;
- *Linhas 11-14*: povoamento do vetor com status livre em todas suas posições;
- *Linhas 16-25*: menu com seleção de inserção ou remoção na *hash*. As respectivas funções serão investigadas posteriormente nesta seção;
- *Linhas 28-32*: função de *hash* adotada neste exemplo. Optou-se por trabalhar com o método de divisão (Tema 2). É válido observar que a função *hashing* poderia ser qualquer outra desejada pelo desenvolvedor.

Figura 3 – Pseudocódigo de implementação da *hash* com menu para inserção e remoção de chave.

```
1  registro Hash
2    chave: inteiro
3    status: caractere
4    //L = Livre, O = Ocupado, R = Removido
5  fimregistro
6
7  algoritmo "HashMenu"
8  var
9    Tabela: Hash[TAMANHO_VETOR]
10   op, pos, num, i: inteiro
11  inicio
12    para i de 0 até (TAMANHO_VETOR - 1) faça
13      Tabela[i].status = 'L' //Coloca todos como Livre
14    fimpara
15
16    leia(op) //Escolhe o que deseja fazer
17    escolha (op)
18      caso 1:
19        leia(num)
20        pos = FuncaoHashing(num)
21        InserirNaHash(Tabela, pos, num)
22      caso 2:
23        leia(num)
24        RemoverDaHash(Tabela, num)
25    fimsecolha
26  fimalgoritmo
27
28  função FuncaoHashing (num: inteiro)
29  var
30  inicio
31    retorne (num MOD TAMANHO_VETOR)
32  fimfunção
```

No exemplo do Tema 1, vimos que cada estado brasileiro acabou sendo posicionado, convenientemente, em uma posição diferente sem a coincidência das posições. E se duas palavras-chave precisam ser posicionadas exatamente na mesma posição do vetor, como tratar esse problema?

Para resolver isso, iremos investigar duas possíveis soluções para o tratamento das chamadas *colisões*, ou seja, quando uma palavra-chave deve ser posicionada em um espaço do vetor em que já está ocupado: a tentativa linear e a tentativa quadrática.

Para resolver essas colisões, temos diferentes algoritmos que tentam reaproveitar os espaços vazios do vetor realocando a chave colidida para outro lugar. Iremos investigar dois testes algoritmos bem como apresentar a implementação deles em pseudocódigo. Veremos o algoritmo linear no Tema 3 e o quadrático no Tema 4.

Quando uma chave k é endereçada em uma posição $h(k)$, e esta já está ocupada, outras posições vazias na tabela são procuradas para armazenar k . Caso nenhuma seja encontrada, isso significa que a tabela está totalmente preenchida e k não pode ser armazenado.

Na *tentativa linear*, sempre que uma colisão ocorre, tenta-se posicionar a nova chave no próximo espaço imediatamente livre do vetor. Vamos compreender o funcionamento por meio de um exemplo.

Queremos preencher um vetor de dimensão 10 com palavras-chave como a sigla de cada estado brasileiro (2 caracteres). O cálculo de cada posição é feito utilizando o método de divisão para caracteres alfanuméricos (Equação 3).

Agora, imaginemos uma situação inicial em que temos o vetor preenchido com 3 estados, conforme a Tabela 2, as siglas PR, RS e SC, em que cada sigla está em uma posição distinta do vetor.

Tabela 2 – Cálculo da posição de inserção para alguns estados brasileiros utilizando a função *hash* (2)

Sigla	ASCII (Dec)	ASCII (Dec)	Soma	Posição (Soma MOD 10)
PR	P (80)	R (82)	162	2
SC	S (83)	C (67)	150	0
RS	R (82)	S (83)	165	5

Temos na Figura 4 os dados inseridos nas posições 0, 2 e 5. Isso significa que essas três posições estão com o *status* de ocupado, enquanto que todas as outras estão com *status* livre.

Figura 4 – Vetor com dados de estados brasileiros inseridos da Tabela 2

0	1	2	3	4	5	6	7	8	9
SC		PR			RS				

Agora queremos inserir o estado do Amazonas (AM) nesse vetor. Utilizando a Equação 3, o somatório de seus caracteres em ASCII resulta em 142 ($A_{DEC} + M_{DEC}$). O resto da divisão pelo tamanho do vetor (10) resultará na posição 2.

Essa posição já está ocupada pelo estado do Paraná (PR), *resultando em uma colisão*. Sendo assim, é necessário tratar a colisão e resolvê-la de alguma maneira. No algoritmo da tentativa linear, quando ocorre essa colisão, segue-se

para a posição subsequentemente livre. Após a posição 2, seguimos para a posição 3. Esta posição está vazia e podemos inserir o estado do AM nela. O resultado é visto na Figura 5.

Figura 5 – Vetor com inserção e colisão por tratativa linear

0	1	2	3	4	5	6	7	8	9
SC		PR	AM		RS				

Continuando, vamos inserir mais um estado neste vetor, o estado do Acre (AC). O somatório de seus caracteres em ASCII resulta em 132 ($A_{DEC} + C_{DEC}$), e, portanto, o resto da divisão pelo tamanho do vetor (10) resultará, mais uma vez, na posição 2.

Conforme visto anteriormente, a posição 2 está ocupada pelo PR. Seguimos para a próxima posição pela tentativa linear, porém a posição 3 também está ocupada pelo estado do AM. Assim, incrementamos novamente nossa posição e atingimos a posição 4, vazia, e podemos fazer a inserção nela. A Figura 6 ilustra nosso exemplo.

Figura 6 – Vetor com inserção e colisão por tratativa linear

0	1	2	3	4	5	6	7	8	9
SC		PR	AM	AC	RS				

Caso o algoritmo de tentativa linear fique buscando uma posição vazia indefinidamente até chegar ao final do vetor e não a encontre, ele retorna ao início e busca até voltar à posição inicialmente testada. Caso nenhum local livre seja localizado, a palavra-chave não pode ser inserida.

3.1 Tentativa linear: pseudocódigo

Vamos investigar os pseudocódigos de inserção, remoção e busca por tentativa linear utilizando endereçamento aberto. Cada manipulação de dado está representada de forma separada por uma função, a qual pode ser chamada pelo algoritmo principal da Figura 3. Na Figura 7, temos o algoritmo para a função de inserção. Vejamos alguns detalhes:

- *Linha 1:* cabeçalho da função que recebe como parâmetro a tabela *hash* usada, a posição inicial a ser testada (isso não garante a inserção, pois depende do espaço estar livre), já calculada pela função *hashing*, e o valor que será inserido;
- *Linhas 5 a 9:* laço de repetição que localiza uma posição para inserir no vetor, iniciando os testes pela posição recebida como parâmetro. As condições impostas no laço *enquanto* dizem que a posição é selecionada quando o espaço está livre (L) ou ele apresenta uma chave já removida (R);
- *Linhas 11 a 16:* quando a posição é selecionada, definimos ela como ocupada (O) e inserimos a chave no local. Caso o tamanho do vetor tenha sido atingido, a inserção não é possível, e uma mensagem é informada ao usuário.

Figura 7 – Pseudocódigo de inserção por tentativa linear

```

1  função InserirNaHash(Tabela: Hash, pos: inteiro, n: inteiro)
2  var
3      i: inteiro
4  inicio
5      enquanto ((i < TAMANHO_VETOR)
6          E (Tabela[(pos+i) MOD TAMANHO_VETOR].status <> 'L')
7          E (Tabela[(pos+i) MOD TAMANHO_VETOR].status <> 'R'))
8          i = i + 1
9      fimenquanto
10
11     se (i < TAMANHO_VETOR) então
12         Tabela[(pos+i) MOD TAMANHO_VETOR].chave = n
13         Tabela[(pos+i) MOD TAMANHO_VETOR].status = 'O'
14     senão
15         escreva("Tabela Cheia!")
16     fimse
17 fimfunção

```

É válido notar que em todas as linhas em que acontece o acesso ao vetor (linha 6, 7, 12 e 13), temos a posição do vetor sendo dada por: $(pos + i) \text{ MOD } TAMANHO_VETOR$. É assim feito pois quando o contador *i* chega ao final do vetor, ou seja, teremos $10 \text{ MOD } 10 = 0$. Assim, retornamos ao início do vetor sem nem precisarmos zerar o contador.

Para a remoção de uma chave da *hash*, podemos informar unicamente qual palavra-chave precisamos remover, sem sabermos sua posição na tabela.

Assim, com base na chave calculamos a posição pela função *hash* e localizamos a posição. Em seguida, marcamos aquele índice como removido (R). Vejamos:

- *Linha 19*: cabeçalho da função que recebe como parâmetro somente a tabela e o valor-chave para remoção;
- *Linha 23*: realiza a busca na *hash*. Nesse algoritmo, a busca está implementada em uma outra função, apresentada na Figura 8;
- *Linhas 25 a 29*: marca a posição encontrada como removida (R). O valor atualmente colocado nesta posição não precisa ser removido/limpado. Manter o valor no vetor é uma estratégia de *backup* caso seja necessário reaver o valor-chave em algum momento.

Figura 8 – Pseudocódigo de remoção por tentativa linear

```
19  função RemoverDaHash(Tabela: Hash, n: inteiro)
20  var
21      posicao: inteiro
22  inicio
23      posicao = BuscarNaHash(Tabela, n)
24
25      se (posicao < TAMANHO_VETOR) então
26          Tabela[posicao].status = 'R'
27      senão
28          escreva("Elemento não existente na tabela!")
29      fimse
30  fimfunção
```

Por fim, temos a busca em uma tabela *hash* com tentativa linear. Em nossos pseudocódigos, a função está separada da remoção, uma vez que podemos somente buscar sem remover simultaneamente.

Iniciamos com o cálculo da função de *hash*. Nela, reavemos a posição calculada, porém o simples cálculo não representa que a chave buscada estará naquela posição, pois ela pode ter sido inserida em uma posição subsequente devido a colisões. A Figura 9 mostra os passos do algoritmo para realizar este processo:

- *Linha 32*: cabeçalho da função que recebe a tabela *hashing* como parâmetro e o valor a ser buscado nela;
- *Linha 37*: calcula a posição inicial usando a função *hash* definida, neste caso, o método da divisão;
- *Linhas 39 a 43*: realiza a varredura a partir da posição recebida como parâmetro;

- *Linhas 45 a 59:* quando uma posição é encontrada verifica-se se a posição não contém um valor removido;

Figura 9 – Pseudocódigo de busca por tentativa linear

```
32 função BuscarNaHash(Tabela: Hash, n: inteiro)
33   var
34     i, pos: inteiro
35   início
36     i = 0
37     pos = FuncaoHashing(n)
38
39   enquanto ((i < TAMANHO_VETOR)
40     E (Tabela[(pos+i) MOD TAMANHO_VETOR].status <> 'L')
41     E (Tabela[(pos+i) MOD TAMANHO_VETOR].chave <> n))
42     i = i + 1
43   fimenquanto
44
45   se ((Tabela[(pos+i) MOD TAMANHO_VETOR].chave <> n)
46     E (Tabela[(pos+i) MOD TAMANHO_VETOR].status <> 'R')) então
47     retorne (pos+i) MOD TAMANHO_VETOR
48   senão
49     retorne TAMANHO_VETOR //Não encontrado
50   fimse
51 fimfunção
```

3.2 Tentativa linear: complexidade

Para a tentativa linear, todas as funções de manipulação da tabela de *hash* (inserção, remoção e busca de um elemento) apresentam risco de colisão para cada tentativa de manipulação.

Embora essas funções apresentem no melhor caso complexidade constante, algo que estávamos buscando desde o começo desta aula, não podemos garantir que elas terão sempre esse tempo, uma vez que o risco de colisão é sempre iminente, independentemente da operação.

Além disso, a tentativa linear funciona com agrupamentos primários de dados, ou seja, podem ocorrer longos trechos de endereços ocupados em sequência. Como a tentativa linear testa um elemento por vez, sequencialmente, a complexidade para o pior caso sempre será $O(n)$ para todas as funções apresentadas (inserção, busca e remoção).

Segundo Cormen (2012), devemos considerar uma outra medida ao analisarmos *hashs*, que é o fator de carga α . Essa medida representa o número de tentativas realizadas para inserção, busca ou remoção e nada mais é do que a razão entre o número de elementos n serem inseridos e o tamanho do vetor m

($\alpha = n/m$). Em endereçamento aberto, temos sempre $n < m$ e portanto $\alpha < 1$, pois temos sempre no máximo um elemento em cada posição.

Supondo uma função de *hash* uniforme aplicada em uma tabela com fator de carga $\alpha < 1$, temos:

- Número máximo de tentativas de uma busca sem sucesso: $1/(1 - \alpha)$
- Número média esperado de tentativas de uma busca: $1/(1 - \alpha)$
- Número máximo de tentativas de uma busca com sucesso: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

TEMA 4 – TABELA *HASHING* DE ENDEREÇAMENTO ABERTO E TENTATIVA QUADRÁTICA

Quando uma chave k é endereçada em uma posição $h(k)$, e esta já está ocupada, outras posições vazias na tabela são procuradas para armazenar k . Caso nenhuma seja encontrada, a tabela está totalmente preenchida e k não pode ser armazenada.

Na tentativa quadrática, sempre que uma colisão ocorre, tenta-se posicionar a nova chave no próximo espaço que está d posições de distância do primeiro testado, em que $1 \leq d \leq TAMANHO_VETOR$.

A função *hashing* adotada como exemplo será novamente o método de divisão para caracteres alfanuméricos (Equação 3). Agora, imaginemos um estado inicial em que temos o vetor preenchido com 3 siglas, conforme a Tabela 3.

Tabela 3 – Cálculo da posição de inserção para alguns estados brasileiros utilizando a função *hash* (3)

Sigla	ASCII (Dec)	ASCII (Dec)	Soma	Posição (Soma MOD 10)
PR	P (80)	R (82)	162	2
SC	S (83)	C (67)	150	0
RS	R (82)	S (83)	165	5

Nenhuma das três siglas inseridas na tabela apresentaram colisão e todas as posições calculadas pela função *hashing* foram diferentes. Vemos na Figura 10 esta representação.

Figura 10 – Vetor com dados de estados brasileiros inseridos da Tabela 3

0	1	2	3	4	5	6	7	8	9
SC		PR			RS				

Vamos inserir o estado do Pará (PA) nesse vetor. O cálculo pelo método da divisão para caracteres alfanuméricos resulta na posição 5 para a sigla PA, posição em que já temos o estado RS. Vejamos as etapas para a inserção:

1. *Status inicial*. Calcula-se a posição inicial e inicializa-se a variável i com o valor um.
 - $d = P(80) + A(65) = 145 \text{ MOD } 10 = 5$.
 - A variável i é inicializada: $i = 1$.
2. *Etapas 1*. Verifica-se a posição 5. Como ela está ocupada, incrementa-se o valor de i nela, e realiza o cálculo da função *hashing* novamente, resultando na posição 6. Incrementa-se o contador.
 - Posição 5 já está ocupada.
 - $d = (d + i) \text{ MOD } 10 = (5 + 1) \text{ MOD } 10 = 6$
 - $i = i + 1 = 2$
3. *Etapas 2*.
 - Posição 6 está livre. Inserção realizada.

Figura 11 – Vetor com dados de estados brasileiros inseridos da Tabela 3

0	1	2	3	4	5	6	7	8	9
SC		PR			RS	PA			

Vamos continuar inserindo neste vetor com o estado do Amapá (AP). O cálculo pelo método da divisão para caracteres alfanuméricos resulta na posição 5, mais um vez, para a sigla PA.

Vejamos as etapas de inserção:

1. *Status inicial*. Calcula-se a posição inicial e inicializa-se a variável i com o valor um.
 - $d = A(65) + P(80) = 145 \text{ MOD } 10 = 5$.
 - A variável i é inicializada: $i = 1$.

2. *Etapa 1.* Verifica-se a posição 5. Como ela está ocupada, incrementa-se o valor de i nela, e realiza o cálculo da função *hashing* novamente, resultando na posição 6. Incrementa-se o contador.

- Posição 5 já está ocupada.
- $d = (d + i) \text{ MOD } 10 = (5 + 1) \text{ MOD } 10 = 6$
- $i = i + 1 = 2$

3. *Etapa 2.* Verifica-se a posição 6. Como ela está ocupada, incrementa-se o valor de i nela, e realiza o cálculo da função *hashing* novamente, resultando na posição 8. Incrementa-se o contador.

- Posição 6 já está ocupada.
- $d = (d + k) \text{ MOD } 10 = (6 + 2) \text{ MOD } 10 = 8$
- $i = i + 1 = 3$

4. *Etapa 3.*

- Posição 8 está livre. Inserção realizada.

Figura 12 – Vetor com dados de estados brasileiros inseridos da Tabela 3

0	1	2	3	4	5	6	7	8	9
SC		PR			RS	PA		AP	

4.1 Tentativa quadrática: pseudocódigo

O pseudocódigo da Figura 3, usado para a criação da tabela *hashing*, a função de *hash* e o menu com as chamadas de funções adotado na tentativa linear pode ser reprisado aqui na tentativa quadrática. O que irá diferir são as funções de inserção, busca e remoção.

Iniciando com a inserção, a Figura 13 mostra o pseudocódigo desta função. Vejamos em detalhes:

- *Linha 1:* cabeçalho da função que recebe como parâmetro a tabela *hash* usada, a posição inicial a ser testada (isso não garante a inserção, pois depende do espaço estar livre) e o valor que será inserido;
- *Linha 5:* cálculo da posição usando a função *hashing*;

- *Linha 6:* inicialização da variável incremental i ;
- *Linhas 8 a 13:* laço de repetição que localiza uma posição para inserir no vetor, iniciando os testes pela posição recebida como parâmetro. As condições impostas no laço enquanto dizem que a posição é selecionada quando o espaço está livre (L) ou ele apresenta uma chave já removida (R). O cálculo da distância é feito da mesma maneira que no exemplo apresentado no TEMA 4;
- *Linhas 15 a 20:* quando a posição é selecionada, nós a definimos como ocupada (O) e é inserida a chave no local. Caso o tamanho do vetor tenha sido atingido, a inserção não é possível, e uma mensagem é informada ao usuário.

Figura 13 – Pseudocódigo de inserção por tentativa quadrática

```

1  função InserirNaHash(Tabela: Hash, n: inteiro)
2  var
3      d, i: inteiro
4  inicio
5      d = FuncaoHashing(n)
6      i = 1
7
8  enquanto ((i <= TAMANHO_VETOR)
9      E (Tabela[d].status <> 'L')
10     E (Tabela[d].status <> 'R'))
11      d = (d + i) MOD TAMANHO_VETOR
12      i = i + 1
13  fimenquanto
14
15  se (i <= TAMANHO_VETOR) então
16      Tabela[d].chave = n
17      Tabela[d].status = 'O'
18  senão
19      escreva("Tabela cheia!")
20  fimse
21  fimfunção

```

Para a remoção de uma chave da *hash* podemos informar unicamente qual palavra-chave precisamos remover, sem sabermos sua posição na tabela. Assim, a partir da chave calculamos a posição pela função *hash* e localizamos a posição. Em seguida, marcamos aquele índice como removido (R). Vejamos:

- *Linha 23:* cabeçalho da função que recebe como parâmetro somente a tabela e o valor-chave para remoção;

- *Linha 27*: realiza a busca na *hash*. Neste algoritmo a busca está implementada em uma outra função, apresentada na Figura 14;
- *Linhas 28 a 32*: marca a posição encontrada como removida (R). O valor atualmente colocado nesta posição não precisa ser removido/limpado. Manter o valor no vetor é uma estratégia de *backup* caso seja necessário reaver o valor-chave em algum momento.

Figura 14 – Pseudocódigo de remoção por tentativa quadrática

```

23  função RemoverDaHash(Tabela: Hash, n: inteiro)
24  var
25      posicao: inteiro
26  inicio
27      posicao = BuscarNaHash(Tabela, n)
28      se (posicao < TAMANHO_VETOR) então
29          Tabela[posicao].status = 'R'
30      senão
31          escreva("Elemento não existente na tabela!")
32      fimse
33  fimfunção

```

Por fim, temos a busca em uma tabela *hash* com tentativa quadrática. Em nossos pseudocódigos a função está separada da remoção, uma vez que podemos somente buscar sem a necessidade de remover, simultaneamente.

Iniciamos com o cálculo da função de *hash*. Nela, reavemos a posição calculada, porém o simples cálculo não representa que a chave buscada estará naquela posição, pois ela pode ter sido inserida em uma posição subsequente devido a colisões. A Figura 15 mostra os passos do algoritmo para realizar este processo:

- *Linha 35*: cabeçalho da função que recebe a tabela *hashing* como parâmetro e o valor a ser buscado nela;
- *Linha 39*: calcula a posição inicial usando a função *hash* definida, neste caso, o método da divisão;
- *Linha 40*: inicializa a variável incremental *i*;
- *Linhas 42 a 47*: realiza a varredura a partir da posição recebida como parâmetro. Lembrando que os cálculos das distâncias são dados de forma quadrática, conforme o exemplo deste tema;
- *Linhas 49 a 54*: quando uma posição é encontrada verifica-se se a posição não contém um valor removido;

Figura 15 – Pseudocódigo de busca por tentativa quadrática

```
35 função BuscarNaHash(Tabela: Hash, n: inteiro)
36   var
37     d, i: inteiro
38   inicio
39     d = FuncaoHashing(n)
40     i = 1
41
42   - enquanto ((i <= TAMANHO_VETOR)
43     E (Tabela[d].status <> 'L')
44     E (Tabela[d].chave <> n))
45     | d = (d + i) MOD TAMANHO_VETOR
46     | i = i + 1
47   fimenquanto
48
49   - se ((Tabela[d].chave == n)
50     E ((Tabela[d].status <> 'R')) então
51     | retorne d
52   senão
53     | retorne TAMANHO_VETOR
54   fimse
55 fimfunção
```

4.2 Tentativa quadrática: complexidade

A análise para o pior case da tentativa quadrática não difere da tentativa linear. A inserção, busca e remoção apresenta complexidade BigO $O(n)$. Este método acaba por ser mais eficiente em tempo de execução somente para situações de casos médios. A mesma análise de fator de carga e tentativas apresentada anteriormente aplica-se na quadrática também.

TEMA 5 – TABELA *HASHING* COM ENDEREÇAMENTO EM CADEIA

Podemos implementar uma tabela *hashing* empregando conceitos de listas encadeadas. Nesse cenário, temos também um vetor de dimensão m para representar a tabela, porém cada posição do vetor armazenará um endereço para o início de uma lista encadeada, de forma semelhante ao que trabalhamos na construção de uma lista de adjacências em grafos.

A função *hashing* a ser empregada aqui continua sendo qualquer uma das já apresentadas no decorrer desta aula, portanto continuaremos adotando o método da divisão em nossos exemplos. A Tabela 4 apresenta três estados e suas respectivas posições calculadas por esse método.

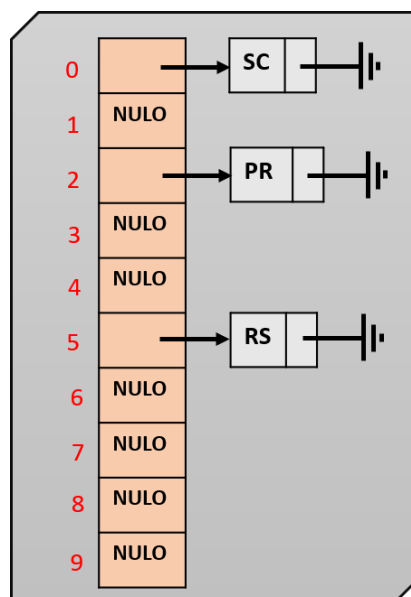
Tabela 4 – Cálculo da posição de inserção para alguns estados brasileiros utilizando a função *hash* (4)

Sigla	ASCII (Dec)	ASCII (Dec)	Soma	Posição (Soma MOD 10)
PR	P (80)	R (82)	162	2
SC	S (83)	C (67)	150	0
RS	R (82)	S (83)	165	5

A inserção dos dados agora é tratada da seguinte maneira: cada chave a ser inserida é alocada como um elemento na memória e, em seguida, seu endereço é colocado em uma lista encadeada referente à posição calculada. Por exemplo, a sigla PR, que tem a posição 2 pela Tabela 4, terá seu endereço posicionado nesta posição do vetor.

Todas as siglas calculadas na Tabela 4, como estão sozinhas em cada posição, representam o *Head* de uma lista encadeada. Assim, as posições 0, 2 e 5 contêm uma lista encadeada com um só elemento (Figura 16).

Figura 16 – Endereçamento em cadeia com dados de estados brasileiros inseridos da Tabela 4



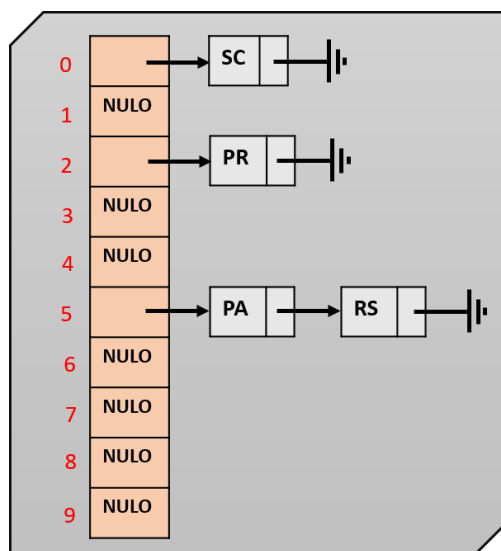
Vamos inserir a sigla PA neste vetor da Figura 16 usando o endereçamento em cadeia. A sigla PA também corresponderá a posição 5 pelo método da divisão. Nesta posição já temos a sigla RS, resultando em uma colisão (Figura 17).

A forma como as colisões são tratadas por esse tipo de endereçamento é diferente. Nele, não é necessário encontrarmos uma nova posição no vetor para

alocarmos o elemento colidido. Basta inseri-lo na mesma posição 5 calculada, mas como mais um elemento da lista encadeada simples.

A inserção é dada sempre antes do *Head*. Assim, a sigla PA virará o novo *Head* da lista da posição 5, apontando para a sigla RS que está na segunda posição da lista. Por se tratar de uma lista não circular, o último elemento conterá um ponteiro nulo para próximo elemento.

Figura 17 – Endereçamento em cadeia. Adicionando PA na posição 5

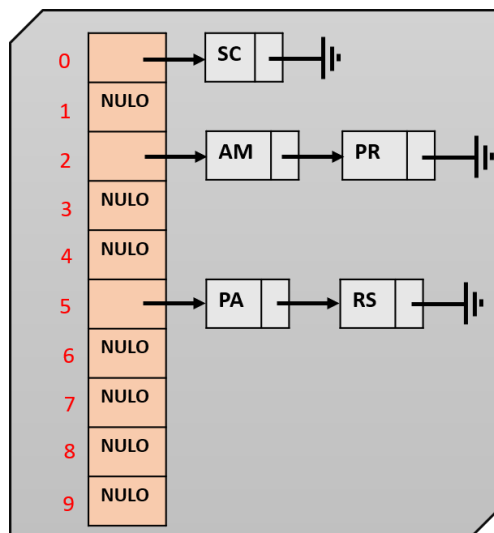


De forma análoga, na Figura 18, acrescentamos a sigla AM. A posição calculada para ela pelo método da divisão resulta na posição 2 do vetor. O elemento será de fato inserido nessa posição, sem a necessidade de encontrar outra.

Como já existe a sigla PR na posição 2, insere AM na lista encadeada dessa posição. A sigla AM será o *Head* e apontará para PR, que apontará para nulo.

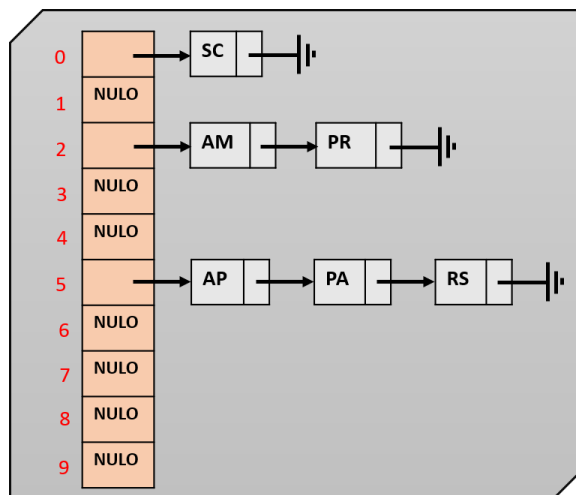
Note que, sempre que necessário buscar uma chave desse vetor, basta recalcular a posição usando a função *hashing* e, em seguida, varrer a lista encadeada simples daquela posição até localizar a palavra-chave correspondente.

Figura 18 – Endereçamento em cadeia. Adicionando AM na posição 2



Podemos continuar inserindo elementos indefinidamente em cada lista encadeada. Por exemplo, se um terceiro elemento colidir na posição 5, conforme apresentado na Figura 19, acontecerá o mesmo processo. A sigla AP precisa entrar na lista encadeada. Assim, ela será colocada no lugar do *Head* da lista encadeada da posição 5, deslocando todos os outros elementos desta lista.

Figura 19 – Endereçamento em cadeia. Adicionando AP na posição 5



5.1 Pseudocódigo

Veremos, a partir de agora, os algoritmos referentes ao endereçamento de cadeia. Note que, por estarmos tratando com listas encadeadas, todos os conceitos estudados serão levados em consideração em relação a esses algoritmos.

O algoritmo apresentado na Figura 20 corresponde ao menu pseudocódigo principal referente à criação da estrutura da *hash*, do vetor e o menu de seleção para inserção e remoção da tabela. Vejamos alguns pontos relevantes:

- *Linhas 1 a 4*: criação do registro cara um elemento da lista encadeada. Temos um valor inteiro que será a chave e o ponteiro para o próximo elemento da lista encadeada simples;
- *Linha 8*: declaração da tabela *hashing*, que agora deve ser um ponteiro para conter os endereços dos *Heads* das listas;
- *Linhas 11 a 13*: inicializa o vetor com endereços nulos em todas as posições;
- *Linhas 15 a 25*: menu com as funções de inserção e remoção da tabela. Estas funções serão detalhas posteriormente;
- *Linhas 27 a 31*: função *hashing* escolhida. Método da divisão.

Figura 20 – Pseudocódigo de implementação da *hash* com endereçamento em cadeia e menu para inserção e remoção de chave

```
1  registro HashLista
2      chave: inteiro
3      prox: HashLista[->)
4  fimregistro
5
6  algoritmo "HashMenuLista"
7  var
8      Tabela: HashLista[TAMANHO_VETOR][->)
9      op, pos, num, i: inteiro
10 inicio
11     para i de 0 até (TAMANHO_VETOR - 1) faça
12         Tabela[i] = NULO
13     fimpara
14
15     leia(op) //Escolhe o que deseja fazer
16     escolha (op)
17         caso 1:
18             leia(num)
19             pos = FuncaoHashing(num)
20             InserirNaHash(Tabela, pos, num)
21         caso 2:
22             leia(num)
23             RemoverDaHash(Tabela, num)
24     fimsecolha
25 fimalgoritmo
26
27 função FuncaoHashing (num: inteiro)
28 var
29 inicio
30     retorne (num MOD TAMANHO_VETOR)
31 fimfunção
```

Na Figura 21, temos a função de inserção no vetor com endereçamento de cadeia. A inserção se dá da mesma maneira que uma inserção no início de uma lista simplesmente encadeada. Vejamos:

- *Linha 1:* cabeçalho da função de inserção que recebe como parâmetro a tabela *hashing*, a posição de inserção, já calculada pela função de *hash* e o valor da chave;
- *Linha 3:* declaração de um novo elemento (nó) que será inserido em uma lista encadeada;
- *Linha 5:* o novo elemento, ainda não colocado na lista, recebe o valor da palavra-chave;
- *Linha 6:* o ponteiro para o próximo elemento deste novo elemento aponta para o *Head* da lista encadeada que está na posição de inserção;
- *Linha 7:* o novo elemento transforma-se no *Head* daquela posição da tabela.

Figura 21 – Pseudocódigo de inserção no endereçamento de cadeia

```
1  função InserirNaHash(Tabela: Hash[->), pos: inteiro, n: inteiro)
2  var
3      Novo: Hash[->)
4  inicio
5      Novo->chave = n
6      Novo->prox = Tabela[pos]
7      Tabela[pos] = Novo
8  fimfunção
```

Na Figura 22, temos a função de remoção no vetor com endereçamento de cadeia. A remoção se dá de maneira semelhante que uma remoção de uma lista simplesmente encadeada. Vejamos:

- *Linha 10:* cabeçalho da função que recebe como parâmetro a tabela *hashing* e o valor da chave para remoção;
- *Linhas 12 e 13:* declaração de dois elementos que auxiliarão da busca e remoção de um elemento da lista encadeada;
- *Linha 18:* verifica a existência de uma lista encadeada na posição desejada para remoção;
- *Linha 19:* verifica se o *Head* da lista encadeada é a palavra-chave buscada;

- *Linhas 20 a 22*: caso a condição da linha 19 for verdadeira, isso significa que o *Head* da lista é o elemento buscado. Portanto, deleta-o e transforma o próximo elemento no novo *Head*;
- *Linhas 24 a 30*: caso a condição da linha 19 for falsa, significa que precisamos varrer a lista encadeada buscando o elemento para remover. Essa parte do código executa esta varredura na lista até localizar o valor correspondente;
- *Linha 32*: verifica se a variável auxiliar de varredura está vazia;
- *Linhas 33 e 34*: se a condição da linha 32 for verdadeira, significa que a variável não está vazia, e portanto deletamos o elemento que está nela e reorganizamos os ponteiros da lista encadeada;
- *Linhas 35 e 36*: se a condição da linha 32 for falsa, significa que a variável auxiliar está vazia, e portanto o valor não foi localizado.

Figura 22 – Pseudocódigo de remoção no endereçamento de cadeia

```

10  função RemoverDaHash(Tabela: Hash[->], n: inteiro)
11  var
12      aux: Hash[->)
13      ant: Hash[->)
14      pos: inteiro
15  inicio
16      pos = FuncaoHashing(n)
17
18      se (Tabela[pos] <> NULO) então
19          se (Tabela[pos]->chave == n) então
20              aux = Tabela[pos]
21              Tabela[pos] = Tabela[pos]->prox
22              delete aux
23          senão
24              aux = Tabela[pos]->prox
25              ant = Tabela[pos]
26
27              enquanto ((aux <> NULO) E (aux->chave <> n))
28                  ant = aux
29                  aux = aux->prox
30              fimenquanto
31
32              se (aux <> NULO) então
33                  ant->prox = aux->prox
34                  delete aux
35              senão
36                  escreva("Valor não encontrado!")
37              fimse
38          fimse
39      senão
40          escreva("Valor não encontrado!")
41      fimse
42  fimfunção

```

5.2 Complexidade

A complexidade para o endereçamento de cadeia apresenta tempo constante para o pior caso $O(1)$ para a inserção na tabela. Esse valor é constante, pois a inserção consiste somente em calcular a função *hashing* e inserir no *Head* da lista da posição correspondente, sem a necessidade de varredura da lista nem de tratamento de colisões.

Para a remoção da lista, o pior cenário seria a necessidade de varrer uma lista encadeada de uma posição buscando um elemento que está na última posição dessa lista simplesmente encadeada. Portanto, a complexidade está atrelada ao número de palavras-chave (números de elementos) de cada lista encadeada, resultando em $O(n)$.

FINALIZANDO

Nesta aula aprendemos sobre a estrutura de dados do tipo *hash*. O objetivo da *hash* é construir uma estrutura de dados capaz de obter tempo de acesso constante às informações contidas nela, independentemente do tamanho do conjunto de dados.

Vimos que tabelas *hashing* armazenam palavras-chave que servem para acessar dados satélite. Essas palavras-chave são armazenadas em posições de um vetor calculadas com base em funções *hashing*. Vimos que essas funções são expressões matemáticas e/ou lógicas e aprendemos duas das mais conhecidas: o método da divisão e o método da multiplicação. Vimos também como melhorar o desempenho dessas funções usando *hashing* universal.

Analizamos diferentes algoritmos para resolver os problemas das colisões, ou seja, quando duas chaves precisam ser posicionadas em uma mesma posição. Aprendemos a tentativa linear e a tentativa quadrática para resolver este problema usando endereçamento aberto.

Vimos ainda uma forma que emprega endereçamento de cadeia, ou seja, listas encadeadas de palavras-chaves em cada posição do vetor, evitando a necessidade do tratamento de colisões.

REFERÊNCIAS

ASCENCIO, A. F. G. **Estrutura de dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson, 2011.

_____. **Fundamentos da programação de computadores**: algoritmos, Pascal, C/C++ (padrão ANSI) JAVA. 3. ed. São Paulo: Pearson, 2012.

CORMEN, T. H. **Algoritmos**: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

KNUTH, D. E. **The art of computer programming**: Sorting and searching (v. 3). 2. ed. Boston/USA: Addison-Wesley, 1998.

LAUREANO, M. **Estrutura de dados com algoritmos E C**. São Paulo: Brasport, 2008.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson, 2008.

PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.