PROGRAMAÇÃO III

AULA 5

CONVERSA INICIAL

O objetivo aqui é apresentar os conceitos que envolvem a estrutura de dados do tipo hash. Ao longo da etapa, essa estrutura de dados será conceituada e o seu funcionamento e aplicabilidade apresentados. Investigaremos também o que são funções hash e veremos algumas das mais comuns. Ainda, as possibilidades de implementação da tabela hash com tratamento para colisões serão estudadas. Por fim, veremos a implementação com endereçamento aberto (linear e quadrática) e a implementação com endereçamento em cadeia.

TEMA 1 – TABELAS HASH

Tabelas hash, ou somente hash, são estruturas de dados que usufruem de características de arrays, mas que melhoram – e muito – o tempo de inserção e busca dos dados. Para uma melhor compreensão, vejamos o caso a seguir.

1.1 O problema da quitanda do Seu Zé

Imagine que você trabalha em uma quitanda que vende frutas, verduras, legumes, laticínios, carnes e diversos outros produtos alimentícios. A quitanda do Seu Zé, infelizmente, não acompanhou o avanço tecnológico e até os dias atuais anota todos os preços de produtos vendidos em um caderninho. Para cada produto vendido, é necessário olhar o preço dele no caderno.

Portanto, qual a melhor maneira de se buscar um preço de produto nesse caderno? Bom, caso os produtos estejam fora de ordem, a única maneira seria olhar linha por linha do caderno, equivalente a fazer uma busca sequencial no conjunto de dados. Caso, por sorte, os produtos tenham sido anotados em ordem alfabética, podemos aplicar uma pesquisa binária, aprimorando o desempenho de O(n) para O(logn).

Apesar de conhecermos o bom desempenho de uma busca binária, não queremos deixar o cliente esperando, certo? Imagine ter que aplicar uma busca, seja ela qual for, em um caderno com dezenas ou centenas de linhas. É difícil e inconveniente. Pensando nesse problema é que Seu Zé resolveu contratar a Joana. A moça é conhecida no bairro como “a menina que tudo lembra”.

Basicamente, Joana é muito boa em decorar números e palavras e foi capaz de decorar toda a lista de preços da quitanda.

Sendo assim, sempre que você precisa do preço de um produto, em vez de buscar no caderninho, você simplesmente pergunta a Joana, que instantaneamente responde.

O uso de Joana para informar os preços faz com que não tenhamos dependência alguma do tamanho do conjunto de dados. Não importa quantos dados de produtos temos no caderno, ela sempre vai responder instantaneamente. Sendo assim, podemos fazer um cálculo comparativo entre as três abordagens, assumindo que cada execução levaria 100 ms (0,1 s) para ocorrer.

Tabela 1 – Cálculo comparativo

	Busca sequencial	Busca binária	Busca pela Joana
Itens no caderno	O(n)	O(logn)	O(1)
100	10 s	1 s	Instantâneo (0,1 s)
1000	1.6 min	1 s	Instantâneo (0,1 s)
10000	16.6 min	2 s	Instantâneo (0,1 s)

Bom, a ideia é fantástica, certo? Mas e agora, como criamos e implementamos uma “Joana” em uma estrutura de dados no programa? Podemos juntar as características e vantagens dos arrays, que envolvem acesso instantâneo ao dado em qualquer posição, fazendo também uma busca instantânea? É aqui que entra a construção de um array com base em uma função hash.

1.2 Função hash

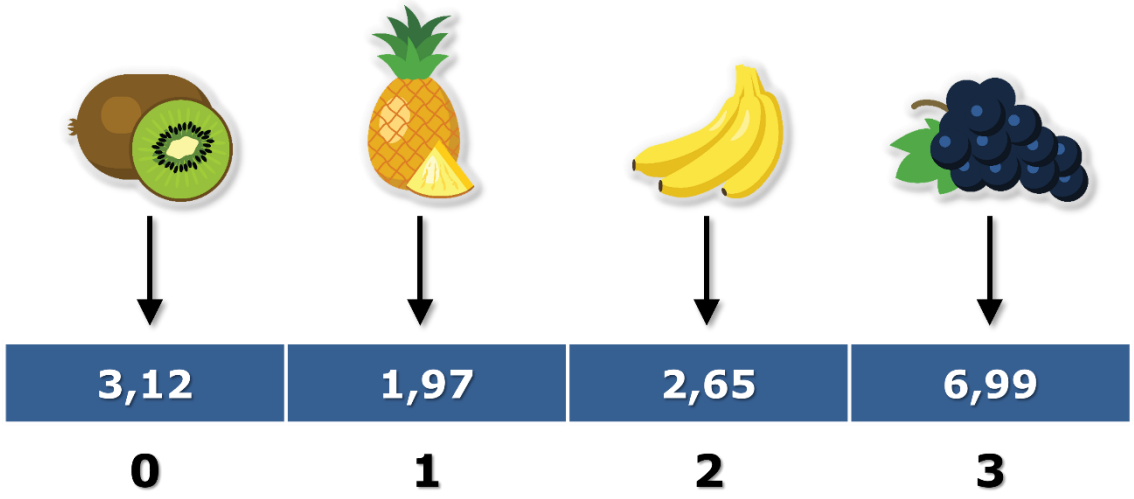
Como você povoaria um array? A maneira mais simples certamente é a de localizar a primeira posição livre e inserir nela um dado. Você sempre inseriu em arrays dessa maneira. Mas a partir de agora, vai aprender a inserir em uma posição qualquer empregando uma função hash.

Uma função hash é uma expressão matemática e/ou lógica que é capaz de receber como entrada um dado, seja ele um número, um caractere, uma string etc., e retornar como resultado um número. Chamamos esse processo de

mapeamento. Em nosso exemplo da quitanda, podemos assumir que queremos mapear nomes de produtos (strings) em números.

Ou seja, sempre que uma palavra específica aparecer, como uva ou banana, uma função hash precisará convertê-la para um número. Esse número corresponderá a um índice (posição) em uma array. Dentro daquela posição do array calculada, teremos a informação que de fato nos interessa, como o preço daquele produto. A figura 1 mostra um exemplo com um array de quatro elementos, no qual cada índice corresponde a um produto.

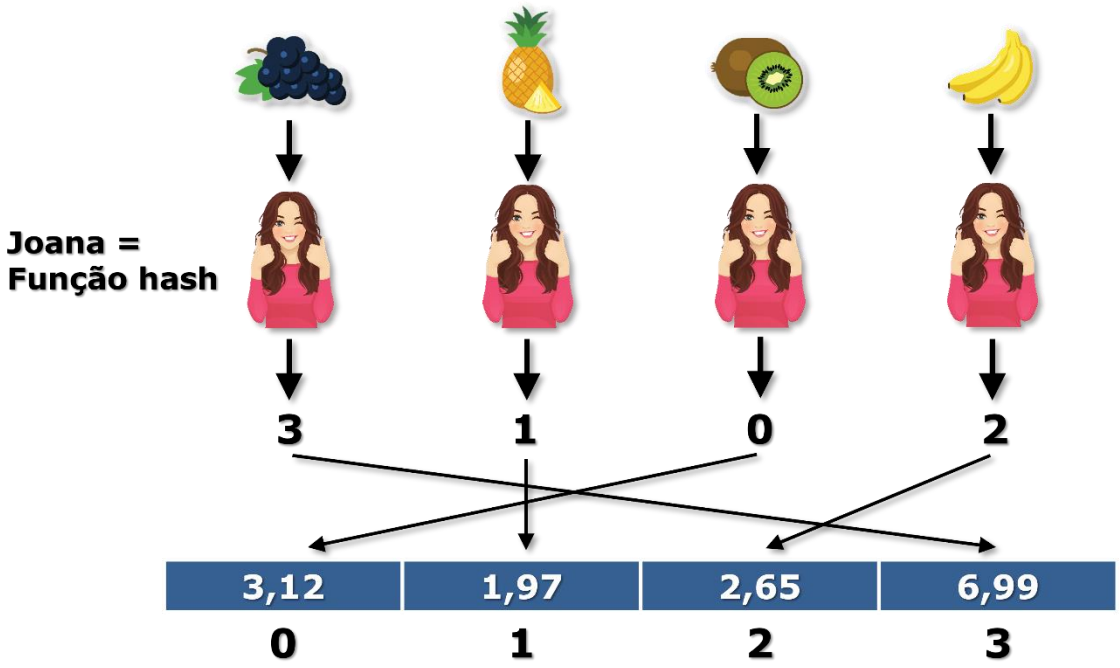
Figura 1 – Exemplo com nomes de frutas mapeando números em um array



Crédito: Inspiring/Shutterstock.

Em nosso exemplo da quitanda, a Joana seria nossa função hash. De alguma maneira, ela é capaz de calcular e saber de cabeça qual o valor de cada produto instantaneamente. A figura 2 mostra que uma fruta é mapeada sempre em um número. O número resultante para aquele nome de fruta (string) será sempre o mesmo. Assim, a banana sempre vai retornar 2, por exemplo. A partir do número 2, acessamos o índice do array que contém o preço da banana.

Figura 2 – Mapeamento de produto para um valor numérico



Crédito: Inspiring/Shutterstock; Volha Hlinskaya/Shutterstock.

Pela figura 2, é possível notar que a ordem com que os dados são mapeados no array não é sequencial. Dependendo de quais dados existirem e também da função hash escolhida, o mapeamento pode mudar.

1.3 A vantagem do emprego de uma função hash

Quando trabalhamos com uma estrutura de dados do tipo array, seja ele estático ou dinâmico, sabemos que, devido a sua **alocação sequencial de dados, somos capazes de acessar qualquer dado dentro dele com o mesmo tempo. Porém, o tempo de busca de um dado em um array é sempre dependente de algum algoritmo**. Portanto:

- Acesso ao dado na memória do array – O(1);
- Busca de um dado no array – depende do algoritmo de busca, pesquisa sequencial O(n) e pesquisa binária O(logn).

Todavia, toda essa análise é feita considerando que estamos cadastrando os dados no array de uma maneira convencional, ou seja, inserimos o dado sempre na próxima posição livre, ordenada ou desordenadamente.

A função hash tem como objetivo calcular uma posição para inserirmos os dados por meio de um cálculo aritmético e/ou lógico. Portanto, sempre que a palavra “banana” aparecer, será representada pelo número 2. Sendo assim, sempre que precisamos reaver os dados contidos em “banana”, podemos novamente aplicar o cálculo e, instantaneamente, acessar aquele índice. Dessa maneira, **deixamos de precisar fazer uma varredura no array e tornamos o acesso ao dado independente do tamanho do conjunto de dados**. Por isso, o uso de uma tabela hash nos proporciona:

- Acesso ao dado na memória na tabela – O(1), pois é implementada como um array sequencial;
- Busca de um dado na tabela hash – O(1), pois acessa o índice por meio de uma função hash.

1.4 A tabela hash e suas implementações

Um array, quando manipulado com inserção e acesso aos dados por meio do emprego de funções hash, é chamado **tabela hash**.

Os dados que utilizamos como entrada na tabela são chamados **palavras-chave**, ou somente **chaves** (do inglês, ***keys***). Em nosso exemplo, as chaves são os nomes das frutas, que são do tipo string.

É válido ressaltar aqui que algumas linguagens de programação já apresentam estruturas de hash implementadas e prontas para uso. Você provavelmente já trabalhou com estruturas de hash existentes em linguagens de programação. A linguagem C++, por exemplo, trabalha com um contêiner associativo chamado *map* (https://cplusplus.com/reference/map/map/). Já na linguagem Java temos a classe *HashMap*, que é bastante similar a nossa tabela *hash*. Por fim, talvez a estrutura hash mais conhecida por você seja a de **dicionário (*dict*) em linguagem Python.** Podemos criar um dicionário em Python e manipulá-lo conforme segue.

Figura 3 – Dicionário

```
#Cria o dicionário
caderno = dict()

#atribui dados ao dicionário
caderno['kiwi'] = 3.12
caderno['abacaxi'] = 1.97
caderno['banana'] = 2.65
caderno['uva'] = 6.99

#imprime o dicionário mapeado: chaves:valores
print(caderno)
#imprime somente o valor da chave banana
print(caderno['banana'])

#imprime todas a chaves
for fruta in caderno.keys():
    print(fruta)
```

Note que, para acessar as chaves, utilizamos o método *keys*. Para um estudo aprofundado, recomendamos a leitura do material a seguir, sobre a função hash implementada no Python: <https://peps.python.org/pep-0456/#conclusion> (acessado em: 19 set. 2022). Posto isso, é importante que você observe que, ao longo desta etapa, **não vamos trabalhar com estruturas de hash prontas na linguagem de programação, mas sim aprender a construir uma do zero**.

1.5 Aplicações de hash

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

- 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 criptografia e autenticação de mensagens e assinatura digital, empregam hashs.
- A estrutura de dados base que permite que as populares criptomoedas – como *bitcoin* – operem são hashs. Elas trabalham com cadeias de hashs altamente complexas para manipular transações, oferecendo segurança e descentralizando as operações.

TEMA 2 – FUNÇÕES HASH

A função hash é parte fundamental do processo de criação de uma tabela hash. Uma função hash, também chamada **algoritmo de hash**, **é uma expressão aritmética e/ou lógica específica para resolver uma determinada aplicação**. A 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 hash, 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 ainda nesta etapa) ocorram de forma mais eficiente possível.

Na figura 4, vemos, de maneira genérica, o processo de hash.

Figura 4 – Processo de *hash*

O processo de *hash* é o ato de definirmos um conjunto de chaves como dados de entrada e aplicarmos a eles uma função hash, gerando, na saída, uma posição que será utilizada para acessar um array.

Antes de entrarmos em exemplos de funções hash, é importante ressaltar que, como a possibilidade de funções hash é muito grande, precisamos entender quando elas são consideradas boas funções. Uma boa função hash, em suma, deve ser o que se segue.

- Fácil de ser calculada. De nada valeria termos uma função com cálculos tão complexos e lentos que fizesse com que todo o tempo ganho no acesso à informação com complexidade O(1) fosse perdido calculando uma custosa função de hash.
- Capaz de distribuir palavras-chave o mais uniformemente possível dentro da estrutura do array.
- 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

Talvez a função hash mais comum aplicada é o método da divisão. Nela, a função consiste em convertermos nossas chaves em um valor numérico e dividirmos esse valor por outro, e o resto dessa divisão será o índice do array a ser acessado.

Na equação 1, temos a equação que descreve esse método da divisão. Tal método é bastante rápido em termos computacionais, uma vez que requer unicamente uma divisão.

$$h(k) = k\ MOD\ n \quad (1)$$

Em que k é uma chave qualquer, n é o tamanho do array, e MOD representa o resto de uma divisão**. Quando nossas palavras-chave são valores inteiros, dividimos o número pelo tamanho do array e usamos o resto dessa divisão como sendo a posição a ser manipulada na tabela hash**. Caso uma palavra-chave adotada seja um conjunto grande de valores inteiros (como um número telefônico ou um CPF, por exemplo), poderíamos somar esses valores agrupados (em pares ou trios) e também dividi-los 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: $99 + 88 + 22 + 33 = 242$. Esse valor é usado no método da divisão.

Para um array de tamanho 12 (n$= 12$) e uma chave de valor 242 ($k = 242$), o resultado da função será $h(k) = 2$, ou seja, adotaríamos o índice 2 para manipular essa chave na tabela. Existem alguns arrays 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 *n* tende a não ser uma boa 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 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 tal.

Para chaves com caracteres alfanuméricos, podemos adotar a mesma equação 1, fazendo pequenas adaptações. Convertemos 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 da palavra-chave na tabela hash (equação 2).

$$h(k) = \left(\sum k_{ASCII\_decimal}\right) MOD\ m \quad (2)$$

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-chave alfanuméricas.

Por exemplo, suponha que temos 2000 conjuntos de caracteres para colocar em uma tabela hash. Definiu-se que realizar uma busca, em média, em até três posições antes de encontrar um espaço vazio é considerado aceitável para essa aplicação. Assim, fazendo $\frac{2000}{3} \cong 666$ para o valor de *n*, podemos adotar um número primo próximo de 666, mas não muito próximo de um múltiplo de 2. Podemos usar o valor 701 para *n*. Nossa função *hash* resultante seria $h(k) = k\ MOD\ 701$.

Por fim, quando trabalhamos com strings como palavras-chave, devemos tomar cuidado com palavras que contenham as mesmas letras, mas em ordens diferentes (anagrama). Por exemplo, uma palavra-chave com quatro caracteres como *ROMA* poderá gerar o mesmo resultado que uma palavra-chave chamada *AMOR*, pois os caracteres são os mesmos rearranjados de outra maneira. Convertendo ambas as palavras para ASCII, decimal, e somando os valores, obtemos 303. Suponha que *n = 13.* Teríamos:

$$h(k) = k\ MOD\ 13$$
$$h(ROMA) = 303\ MOD\ 13 = 4$$
$$h(AMOR) = 303\ MOD\ 13 = 4$$

Uma função de hash deve ser cuidadosamente definida para tratar esse tipo de problema, caso ele venha a ser recorrente. Do contrário, teremos excessivas colisões.

2.2 A hash universal

Considerando que uma chave k qualquer tem 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 manipular nessa hash será $O(n)$. Podemos minimizar esse tipo de problema escolhendo uma função hash aleatoriamente dentro de um universo H de funções. A essa solução chamamos **hash universal**.

Na hash 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, no qual nenhuma colisão ocorre, mas teremos sempre uma situação em média aceitável.

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

2.2.1 Implementação matemática de hash universal

Imaginemos um número primo p e um conjunto de valores $Z_p = \{0, 1, 2 \dots p-1\}$. 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 desse número primo p e do seu conjunto Z_p (equação 3):

$$h_{a,b}(k) = \big((ak + b)\ MOD\ p\big)\ MOD\ n \qquad (3)$$

Onde $a \in Z_p^*$ e $b \in Z_p$. A variação das constantes a e b constituem diversas possibilidades para essa classe de funções dependente de um valor primo p. Se assumirmos o número primo $p = 17$, um vetor de dimensão $n = 6$ e também $a = 3$ e $b = 4$, obtemos pela equação 3 que $h_{3,4}(8) = \big((3.8 + 4)\ MOD\ 17\big)\ MOD\ 6) = 5$.

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. Você deseja produzir palavras-chave de tamanho b_2 bits. Criamos, então, uma matriz binária aleatória de dimensão $M = b_1 x b_2$. A função hash dessa classe será, portanto, $h(k) = M.k$.

Você deve considerar seus dados como sendo valores binários. Esses valores 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 $M = 3x4$. Uma possível matriz binária aleatória seria:

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix} \quad (4)$$

Assumindo um dado de entrada como sendo 1011, multiplicamos pela matriz aleatória e obtemos um resultado de 3 bits (equação 5). 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.

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad (5)$$

TEMA 3 – IMPLEMENTANDO UMA TABELA HASH

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. Entre esses dados, você precisará armazenar muitas informações, como a capital do estado, a população total, a lista de cidades, o PIB do estado, o índice de criminalidade, o nome do governador, entre inúmeros outros dados.

Você vai implementar esse cadastro usando uma tabela hash. Vamos adotar a sigla de cada estado como sendo uma palavra-chave. 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 (para consulta, veja <http://www.asciitable.com/>, acessado em: 19 set. 2022). Então, vamos definir uma função hash como sendo a soma em decimal de ambas as letras e dividir o resultado pelo tamanho de nosso array (dimensão 10). A posição de inserção no vetor será o resto dessa divisão.

Consideraremos ambos os 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 dessa divisão, temos o valor 2. Esse 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 os valores será 165, e o resto da divisão por 10 resultará no valor 5, valor esse que corresponde à posição de inserção desse dado no vetor.

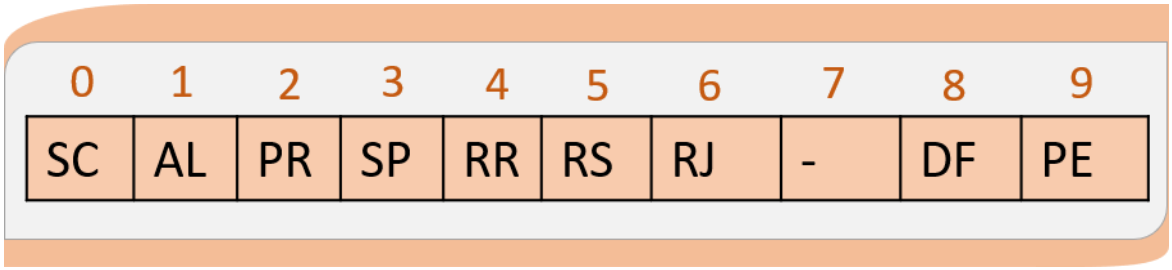
Na tabela a seguir, temos o cálculo da posição de inserção para alguns estados brasileiros utilizando a função hash.

Tabela 2 – Cálculo da posição de inserção

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**

Na figura a seguir, temos os dados calculados usando a função hash e agora inseridos nas suas respectivas posições. Cada estado resultou em uma posição diferente na tabela hash. Veja a inserção dos dados na tabela.

Figura 5 – Array com dados de estados brasileiros inseridos utilizando uma função hash



É interessante analisar que, alterando o tamanho de nosso vetor, podemos alterar 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.

3.1 Implementação em Python: função hash

A seguir, temos uma implementação de uma função hash usando o método da divisão e empregando valores numéricos. A função criada é bastante simples. Ela recebe a chave e o tamanho de uma lista em Python e retorna o resto dessa divisão (símbolo de %).

Figura 6 – Implementação em Python

```
def hashFunc (k, n):
    return k % n
```

Uma variação dessa mesma função do método da divisão pode ser vista a seguir. O código funciona para dois caracteres, podendo ser aplicado para as siglas de estados brasileiros.

Figura 7 – Implementação em Python – variação

```
def hashFuncSigla (k, n):
    k = list(k)
    return (ord(k[0]) + ord(k[1])) % n
```

Note que primeiro convertemos a chave recebida, que é uma string, em uma lista. Em seguida, usando uma função do Python chamada ord, convertemos cada caractere em seu respectivo valor decimal da tabela ASCII. Por fim, fazemos a soma e pegamos o resto da divisão.

3.2 Implementação em Python completa

O código a seguir implementa um array com inserção e remoção empregando uma função hash. Foi criado um menu de seleção, contendo as opções de inserir, remover e listar a estrutura de dados. Note que, **se houver uma mesma chave que caia em uma posição em que já existe algum dado, o código não permite a inserção do dado. Quando essa situação ocorre, na verdade temos uma colisão**. Vamos aprender a tratar colisões no próximo tópico. Quando aprendermos esse assunto, vamos reformular esse código para atender às colisões também.

Figura 8 – Array com inserção e remoção de função hash

```
#Programa principal
n = 10
tabelaHash = [None] * n

while True:
  print('1 - Inserir na tabela hash')
  print('2 - Remover na tabela hash')
  print('3 - Listar a tabela hash')
  print('4 - Sair')

  op = int(input("Escolha uma opção:"))
  if op == 1:
    chave = input('Digite a sigla de um estado: ')
    pos = hashFuncSigla(chave, n)
    if tabelaHash[pos] == None:
        tabelaHash.insert(pos, chave)
    else:
        print('Já existe um dado neste lugar!')
  elif op == 2:
    chave = int(input('Digite o que deseja remover: '))
    pos = hashFunc(chave, n)
    if tabelaHash[pos] == chave:
        tabelaHash.pop(pos)
    else:
        print('Valor não localizado para a remoção!')
  elif op == 3:
     print(tabelaHash)
  elif op == 4:
    print('Encerrando...')
    break
  else:
    print("Selecione outra opção!\n")
```

TEMA 4 – COLISÕES EM TABELAS HASH

E se duas palavras-chave resultarem em uma mesma posição dentro do array, como procedemos? Quando isso ocorre, dizemos que uma colisão ocorreu. Mas como tratamos essa colisão? Podemos ainda inserir um dado colidido ou precisamos descartá-lo?

É basicamente **impossível escrevermos uma função hash que seja livre de colisões**. Não importa a função hash nem a aplicação, colisões sempre ocorrerão. Podemos sim tratar essas colisões e inserir um dado colidido em outra posição da tabela hash, mas, para isso, vamos precisar investigar e estudar maneiras de tratar esse tipo de problema.

4.1 Colisões com endereçamento aberto

A maneira como vamos tratar as colisões depende muito do tipo de endereçamento usado para construir a tabela hash. O primeiro tipo é o **endereçamento aberto**. Nele, **cada posição da estrutura de dados só pode conter uma única palavra-chave**. Isso significa que, caso uma segunda chave seja calculada para entrar em uma posição já preenchida, precisaremos aplicar um algoritmo para inserir em outra posição livre.

4.1.1 Tentativa linear

Na tentativa linear, sempre que uma colisão ocorre, tenta-se posicionar a nova chave no próximo espaço imediatamente livre do array. Vamos compreender o funcionamento por meio de um exemplo. Queremos preencher um vetor de dimensão 10 com palavras-chave iguais à sigla de cada estado brasileiro (dois caracteres). O cálculo de cada posição é feito utilizando o método de divisão para caracteres alfanuméricos (equação 2).

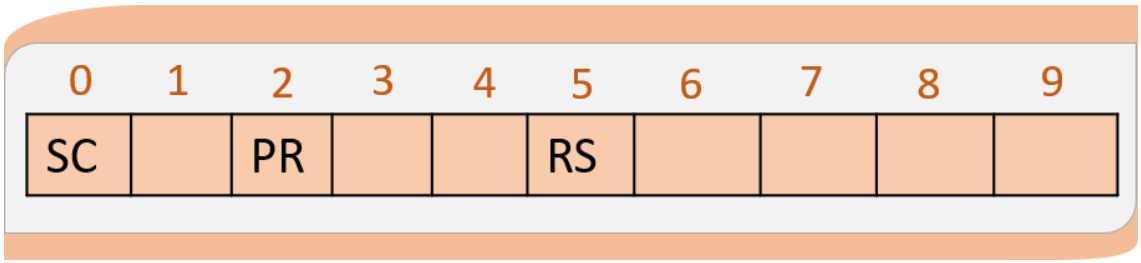
Agora, imaginemos uma situação inicial em que temos o vetor preenchido com três estados, conforme a tabela com as siglas PR, RS e SC, em que cada sigla está em uma posição distinta do vetor.

Tabela 3 – Siglas em posições distintas

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 a seguir, os dados inseridos nas posições 0, 2 e 5. Isso significa que essas três posições estão com o status de ocupado, enquanto todas as outras estão com status livre.

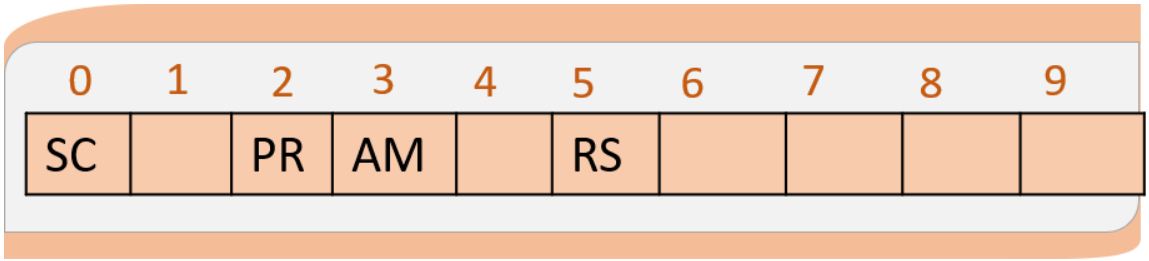
Figura 9 – Array com dados de estados brasileiros inseridos



Agora, queremos inserir o estado do Amazonas (AM) nesse vetor. Utilizando a equação 2, 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. Essa posição está vazia, de modo que podemos inserir o estado do AM nela. O resultado é visto na figura a seguir.

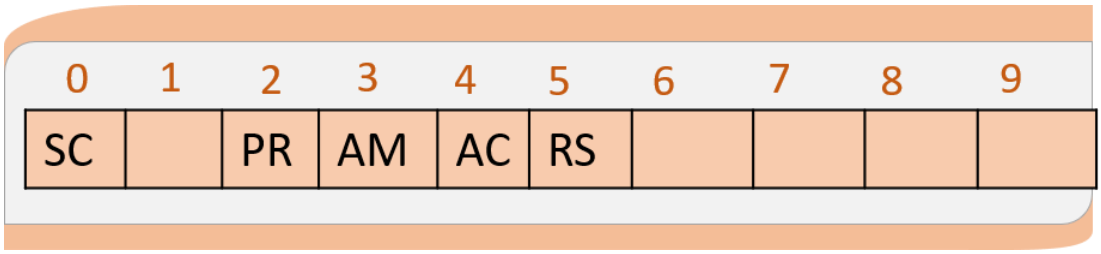
Figura 10 – Vetor com inserção e colisão por tratativa linear



Continuando, vamos inserir mais um estado nesse array, 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 a seguir ilustra nosso exemplo.

Figura 11 – Vetor com inserção e colisão por tratativa linear



O algoritmo de tentativa linear pode ficar buscando uma posição vazia indefinidamente até chegar ao final do vetor. Caso não encontre nenhuma posição vazia, 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.

4.1.2 Tentativa quadrática

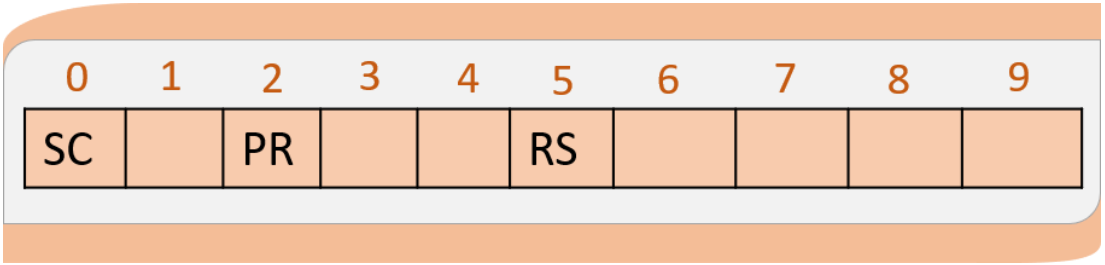
Na tentativa quadrática, sempre que uma colisão ocorre, tenta-se posicionar a nova chave no próximo espaço que está a *d* posições de distância do primeiro espaço testado, onde 1≤ d ≤ n, em que *n* é o tamanho do vetor. A função hash adotada como exemplo será novamente o método de divisão para caracteres alfanuméricos (equação 2). Agora, imaginemos um estado inicial em que temos o vetor preenchido com três siglas.

Tabela 4 – Estado com três siglas

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 hash foram diferentes. Vemos essa representação na figura a seguir.

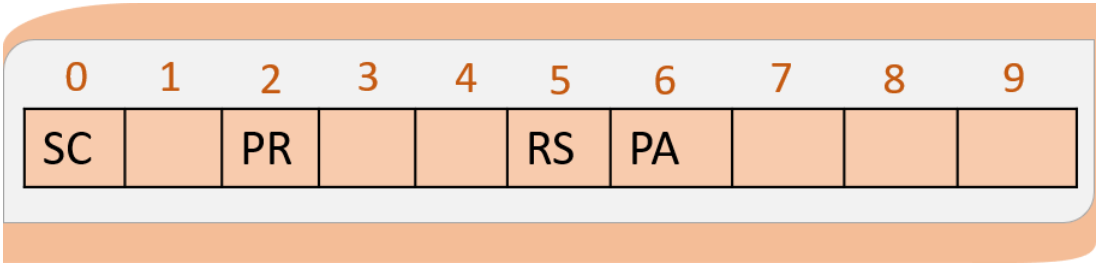
Figura 12 – Vetor com dados de estados brasileiros inseridos



Vamos inserir o estado do Pará (PA) nesse array. O cálculo pelo método da divisão para caracteres alfanuméricos resulta na posição 5 para a sigla PA, posição na qual 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 1.
   - $d = P\,(80) + A\,(65) = 145\,MOD\,10 = 5$
   - 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 e realiza-se o cálculo da função hash novamente, resultando na posição 6. Incrementa-se o contador.
   - Posição 5 já está ocupada;
   - $d = (d + i)\,MOD\,10 = (5 + 1)\,MOD\,10 = 6$
   - $i = i + 1 = 2$
3. *Etapa 2.*
   - Posição 6 está livre. Inserção realizada.

Figura 13 – Vetor com dados de estados brasileiros inseridos

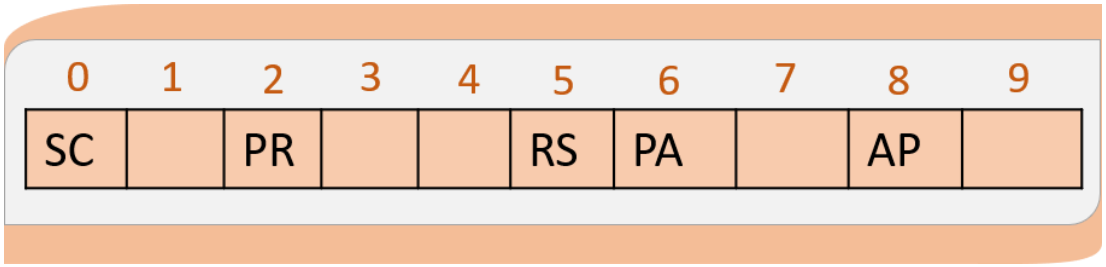


Vamos continuar inserindo nesse vetor com o estado do Amapá (AP). O cálculo pelo método da divisão para caracteres alfanuméricos resulta na posição 5, mais uma vez, para a sigla PA.

Vejamos as etapas de inserção.

1. *Status inicial.* Calcula-se a posição inicial e inicializa-se a variável *i* com o valor um.
   - $d = A\,(65) + P\,(80) = 145\ MOD\ 10 = 5$
   - 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-se o cálculo da função hash novamente, resultando na posição 6. Incrementa-se o contador.
   - Posição 5 já está ocupada;
   - $d = (d + i)\ MOD\ 10 = (5 + 1)\ MOD\ 10 = 6$
   - $i = i + 1 = 2$
3. *Etapa 2*. Verifica-se a posição 6. Como ela está ocupada, incrementa-se o valor de i nela e realiza-se o cálculo da função hash novamente, resultando na posição 8. Incrementa-se o contador.
   - *Posição 6 já está ocupada.;*
   - $d = (d + k)\ MOD\ 10 = (6 + 2)\ MOD\ 10 = 8$
   - $i = i + 1 = 3$
4. *Etapa 3.*
   - *Posição 8 está livre. Inserção realizada.*

Figura 14 – Vetor com dados de estados brasileiros inseridos



4.2 Endereçamento em cadeia

O segundo tipo é o **endereçamento em cadeia**. Nele, **cada posição da estrutura de dados pode contar diversas palavras-chave.** Na prática, para implementá-lo, vamos precisar trabalhar com listas encadeadas.

Nesse cenário, temos também um array de dimensão *n* para representar a tabela. Porém, cada posição do vetor armazenará um endereço para o início de uma lista encadeada. A função hash a ser empregada aqui continua sendo

qualquer uma das já apresentadas no decorrer desta caminhada, portanto, continuaremos adotando o método da divisão em nossos exemplos. Vamos ao nosso exemplo dos estados.

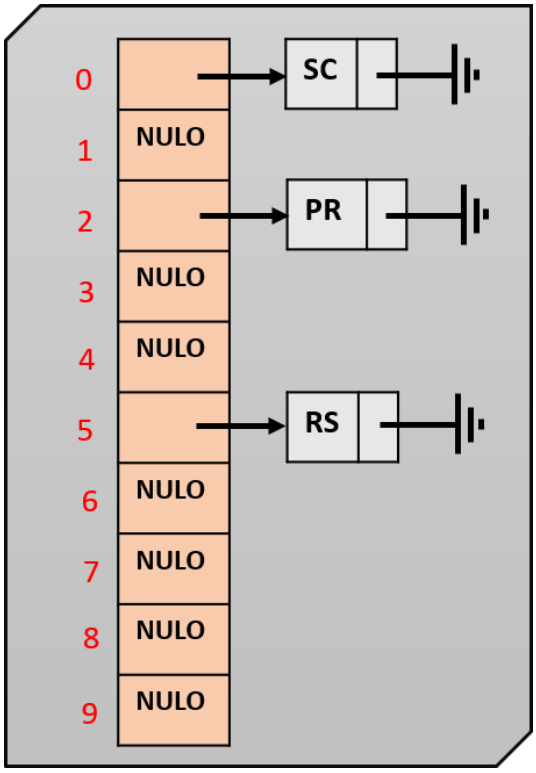
Tabela 5 – Siglas dos estados

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, terá seu endereço posicionado nessa posição do vetor.

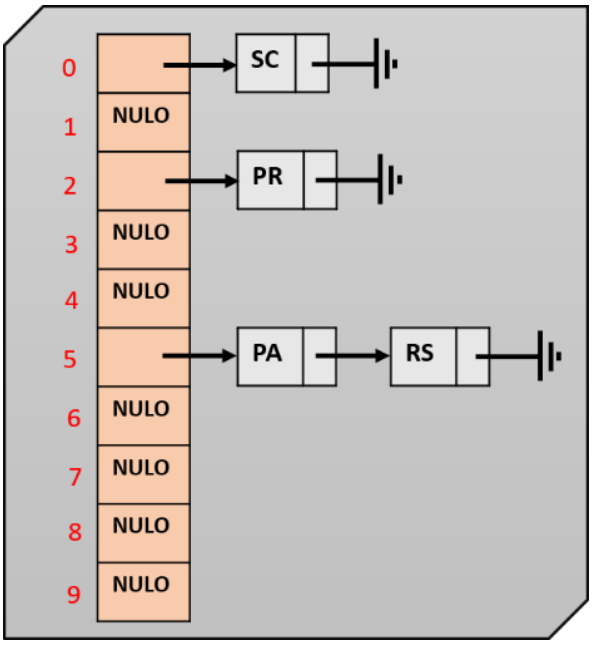
Todas as siglas calculadas na tabela, como estão sozinhas em cada posição, representam o *Head* de uma lista encadeada. Assim, as posições 0, 2 e 5 contêm uma lista encadeada com um só elemento.

Figura 15 – Endereçamento em cadeia com dados de estados brasileiros inseridos da tabela



Vamos inserir a sigla PA nesse vetor da figura usando o endereçamento em cadeia. A sigla PA também corresponderá à posição 5 pelo método da divisão. Nessa posição já temos a sigla RS, resultando em uma colisão.

Figura 16 – Endereçamento em cadeia: adicionando PA à posição 5



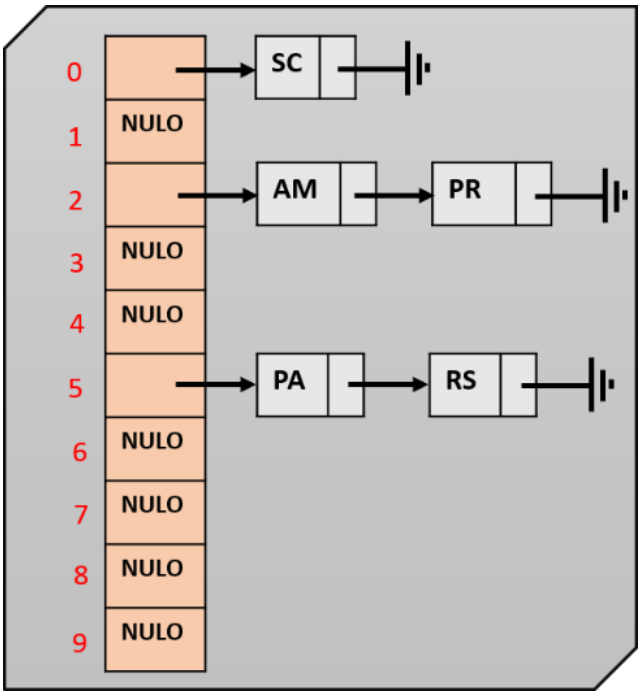
A forma como as colisões são tratadas por esse tipo de endereçamento é diferente. Nele, não é necessário encontrarmos uma nova posição no vetor para alocarmos o elemento colidido. Basta inseri-lo na mesma posição 5 calculada, mas como mais um elemento da lista encadeada simples.

A inserção é dada sempre antes do *Head* (início da lista encadeada). 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 o próximo elemento.

De forma análoga, na figura 17, 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 posição.

Como já existe a sigla PR na posição 2, insere-se AM na lista encadeada dessa posição. A sigla AM será o *Head* e apontará para PR, que apontará para nulo. Note que sempre que for necessário buscar uma chave desse vetor, basta recalcular a posição usando a função hash e, em seguida, varrer a lista encadeada simples daquela posição até localizar a palavra-chave correspondente.

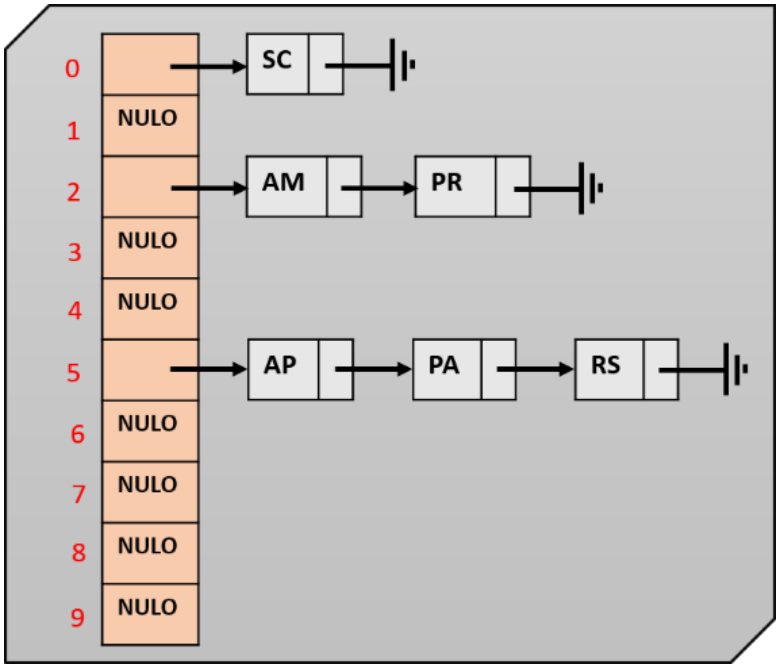
Figura 17 – Endereçamento em cadeia: adicionando AM à posição 2



Podemos continuar inserindo elementos indefinidamente em cada lista encadeada. Por exemplo, se um terceiro elemento colidir na posição 5,

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 dessa lista.

Figura 18 – Endereçamento em cadeia: adicionando AP à posição 5



TEMA 5 – IMPLEMENTANDO COLISÕES E DESEMPENHO HASH

Vamos implementar uma tabela hash tratando as colisões em linguagem Python.

5.1 Endereçamento aberto e tentativa linear

Para o endereçamento aberto, implementamos em linguagem Python com listas, que são arrays dinâmicos. A seguir, podemos ver a implementação para a tentativa linear.

Figura 19 – Tentativa linear

```
#Tentativa linear
def tentativaLinear(k, n, pos, tabelaHash):
  tentativa = pos
  while (tabelaHash[tentativa] != None):
    tentativa += 1
    if tentativa == n:
      tentativa = 0
    if tentativa == pos:
      tentativa = -1
      break
  return tentativa
```

A função implementada recebe como parâmetro a tabela, a posição a ser inserida e também a chave e o tamanho do vetor. Criamos uma variável adicional local chamada *tentativa* que irá receber como valor inicial a posição inicial a ser testada para a inserção. Caso a posição não esteja vazia, precisamos incrementar a posição de tentativa até aparecer uma posição vazia.

Todavia, note que a posição a ser inserida ocasionalmente pode ser a última, ou uma das últimas, do array. Mas ainda temos posições livres no início do array. Portanto, temos uma condicional que verifica se atingimos o final do array e voltamos para o início. Temos também outra condição que identifica se todas as posições já foram testadas, retornando um código de erro.

A seguir, vemos também o código para remover um dado com colisões sendo tratadas com tentativa linear.

Figura 20 – Tratamento com tentativa linear

```
#Tentativa linear Remover
def tentativaLinearDel(k, n, pos, tabelaHash):
  tentativa = pos
  while (tabelaHash[tentativa] != k):
    tentativa += 1
    if tentativa == n:
      tentativa = 0
    if tentativa == pos:
      tentativa = -1
      break
  return tentativa
```

O código completo de manipulação da hash é visto a seguir.

Figura 21 – Código completo

```
#Programa principal
n = 10
tabelaHash = [None] * n

while True:
  print('1 - Inserir na tabela hash')
  print('2 - Remover na tabela hash')
  print('3 - Listar a tabela hash')
  print('4 - Sair')

  op = int(input("Escolha uma opção:"))
  if op == 1:
    chave = input('Digite a sigla de um
estado: ')
    pos = hashFuncSigla(chave, n)
    if tabelaHash[pos] == None:
        tabelaHash[pos] = chave
    else: #Colisão!
        pos = tentativaLinear(chave, n,
pos, tabelaHash)
        if pos != -1:
          tabelaHash[pos] = chave
        else:
          print('Tabela has cheia.
Impossível inserir!')
  elif op == 2:
    chave = input('Digite o que deseja
remover: ')
    pos = hashFuncSigla(chave, n)
    if tabelaHash[pos] == chave:
        tabelaHash[pos] = None
    else: #Colisão
        pos = tentativaLinearDel(chave, n,
pos, tabelaHash)
        if pos != -1:
          tabelaHash[pos] = None
        else:
            print('Valor não localizado
para a remoção!')
  elif op == 3:
      print(tabelaHash)
  elif op == 4:
    print('Encerrando...')
    break
  else:
    print("Selecione outra opção!\n")
```

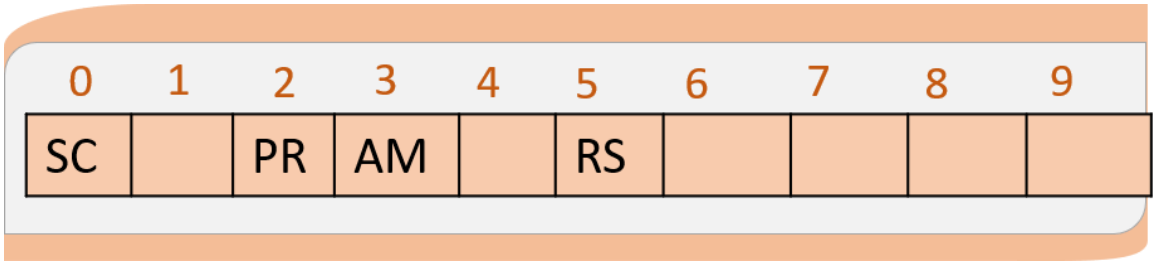
5.2 Fator de carga

Suponha que você precisa armazenar o preço de 100 produtos da quitanda do Seu Zé na tabela hash. Se sua tabela hash tiver 100 espaços no array, na melhor hipótese, sem colisões, cada chave terá seu espaço próprio. Isso significa que o fator de carga desse exemplo será 1. O fator de carga é sempre calculado como sendo:

$$\frac{Total\ de\ chaves}{Total\ de\ espaços}$$

No exemplo, 100 produtos (chaves) para 100 espaço dá 100 dividido por 100, resultando em 1. Observe a figura a seguir. Temos array de dez espaços. Se retomarmos o exemplo dos estados brasileiros, temos 26 estados no Brasil, mais o Distrito Federal.

Figura 22 – Exemplo para o fator de carga



Para calcular o fator de carga, fazemos: 27/10 = 2,7. O fator de carga está bastante alto! Idealmente, precisamos deixar o fator de carga abaixo de 1, pois assim teremos certeza de que existirão espaços livres para todas as chaves. Sempre que o fator de carga começar a aumentar para uma determinada aplicação, precisaremos ficar atentos e redimensionar o tamanho do conjunto de dados.

5.3 Desempenho de hash

No pior caso, uma tabela hash tem tempo de execução O(n). Isso ocorre porque o pior caso contempla o fato de sempre existirem colisões e sempre precisarmos tratá-las. Para fins de comparação, estamos apresentando dessa vez não só o pior caso, como também o caso médio das manipulações da tabela hash.

Tabela 6 – Busca, inserção e remoção

	Tabela hash (caso médio)	Tabela hash (pior caso)	Array	Listas encadeadas
Busca	O(1)	O(n)	O(1)	O(n)
Inserção	O(1)	O(n)	O(n)	O(1)
Remoção	O(1)	O(n)	O(n)	O(1)

Preste atenção ao caso médio para tabelas hash. As tabelas hash são tão velozes quanto arrays para busca. Elas são tão velozes quanto as listas para inserção e remoção. Ou seja, ela é o melhor entre dois mundos! É claro que, para o pior caso, ela acaba sendo lenta. Portanto, precisamos evitar esse pior caso, evitando ao máximo as colisões.

FINALIZANDO

Nesta etapa, aprendemos sobre a estrutura de dados do tipo hash. O objetivo da hash foi 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 hash armazenam palavras-chave que servem para acessar dados. Essas palavras-chave são armazenadas em posições de um vetor calculadas por meio de funções hash. Vimos também que essas funções são expressões matemáticas e/ou lógicas e aprendemos duas das mais conhecidas.

Analisamos 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 esse problema usando endereçamento aberto.

REFERÊNCIAS

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