

# Estrutura de Dados

## PROFESSORES

Esp. Edson Orivaldo Lessa Junior

Esp. Rafael Orivaldo Lessa

ACESSE AQUI O SEU  
LIVRO NA VERSÃO  
**DIGITAL!**

# EXPEDIENTE

## DIREÇÃO UNICESUMAR

**Reitor** Wilson de Matos Silva **Vice-Reitor** Wilson de Matos Silva Filho **Pró-Reitor de Administração** Wilson de Matos Silva Filho **Pró-Reitor Executivo de EAD** William Victor Kendrick de Matos Silva **Pró-Reitor de Ensino de EAD** Janes Fidélis Tomelin **Presidente da Mantenedora** Cláudio Ferdinandi

## NEAD - NÚCLEO DE EDUCAÇÃO A DISTÂNCIA

**Diretoria Executiva** Chrystiano Mincoff, James Prestes, Tiago Stachon **Diretoria de Graduação e Pós-graduação** Kátia Coelho **Diretoria de Cursos Híbridos** Fabricio Ricardo Lazilha **Diretoria de Permanência** Leonardo Spaine **Diretoria de Design Educacional** Paula Renata dos Santos Ferreira **Head de Graduação** Marcia de Souza **Head de Metodologias Ativas** Thuinie Medeiros Vilela Daros **Head de Tecnologia e Planejamento Educacional** Tania C. Yoshie Fukushima **Gerência de Planejamento e Design Educacional** Jislaine Cristina da Silva **Gerência de Tecnologia Educacional** Marcio Alexandre Wecker **Gerência de Produção Digital** Diogo Ribeiro Garcia **Gerência de Projetos Especiais** Edison Rodrigo Valim **Supervisora de Produção Digital** Daniele Correia

## FICHA CATALOGRÁFICA

### Coordenador(a) de Conteúdo

Flavia Lumi Matuzawa

### Projeto Gráfico e Capa

André Morais, Arthur Cantareli e  
Matheus Silva

### Editoração

Sabrina Novaes e Juliana Duenha

### Design Educacional

Ivana Cunha Martins

### Curadoria

Elziane Vieira Alencar

### Revisão Textual

Cintia Prezoto Ferreira

### Ilustração

André Azevedo e Eduardo Aparecido  
Alves

### Fotos

Shutterstock

### C397 CENTRO UNIVERSITÁRIO DE MARINGÁ.

Núcleo de Educação a Distância. **LESSA JUNIOR**, Edson Orivaldo; **LESSA**, Rafael Orivaldo.

**Estrutura de Dados.** Edson Orivaldo Lessa Junior; Rafael Orivaldo Lessa. Maringá - PR: Unicesumar, 2021.

**176 p.**

"Graduação - EaD".

1. Estrutura 2. Dados 3. informação. 4. EaD. I. Título.

CDD - 22 ed. 005.1

Impresso por:

Bibliotecário: João Vivaldo de Souza CRB-9-1679



NEAD - Núcleo de Educação a Distância

Av. Guedner, 1610, Bloco 4 Jd. Aclimação - Cep 87050-900 | Maringá - Paraná

www.unicesumar.edu.br | 0800 600 6360

# BOAS-VINDAS

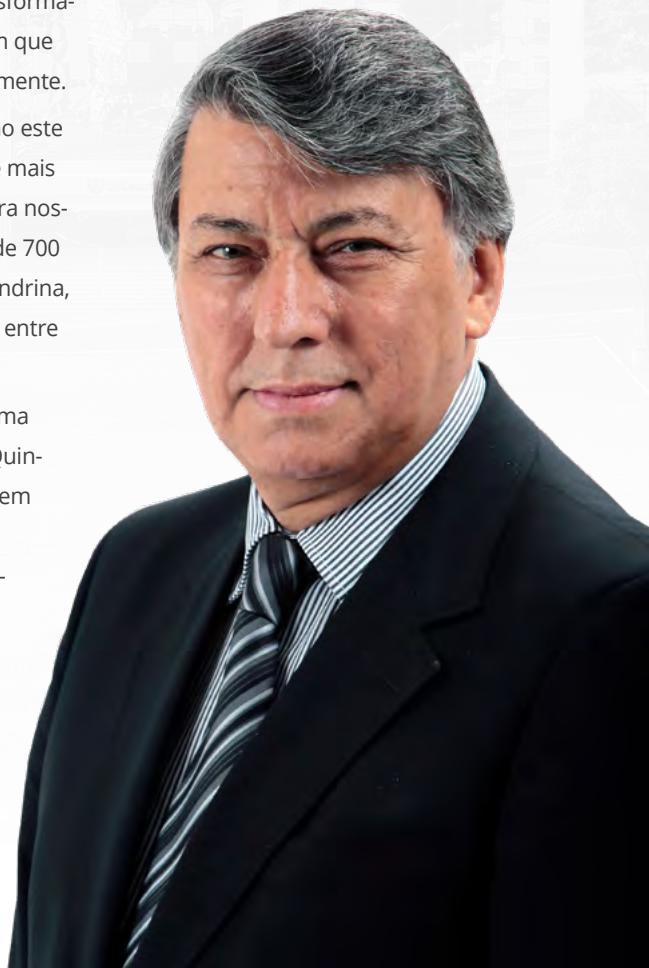
A UniCesumar celebra os seus 30 anos de história avançando a cada dia. Agora, enquanto Universidade, ampliamos a nossa autonomia e trabalhamos diariamente para que nossa educação à distância continue como uma das melhores do Brasil. Atuamos sobre quatro pilares que consolidam a visão abrangente do que é o conhecimento para nós: o intelectual, o profissional, o emocional e o espiritual.

A nossa missão é a de "Promover a educação de qualidade nas diferentes áreas do conhecimento, formando profissionais cidadãos que contribuam para o desenvolvimento de uma sociedade justa e solidária". Neste sentido, a UniCesumar tem um gênio importante para o cumprimento integral desta missão: o coletivo. São os nossos professores e equipe que produzem a cada dia uma inovação, uma transformação na forma de pensar e de aprender. É assim que fazemos juntos um novo conhecimento diariamente. São mais de 800 títulos de livros didáticos como este produzidos anualmente, com a distribuição de mais de 2 milhões de exemplares gratuitamente para nossos acadêmicos. Estamos presentes em mais de 700 polos EAD e cinco campi: Maringá, Curitiba, Londrina, Ponta Grossa e Corumbá, o que nos posiciona entre os 10 maiores grupos educacionais do país.

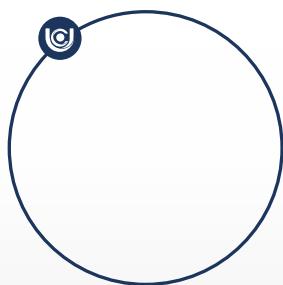
Aprendemos e escrevemos juntos esta belíssima história da jornada do conhecimento. Mário Quintana diz que "Livros não mudam o mundo, quem muda o mundo são as pessoas. Os livros só mudam as pessoas". Seja bem-vindo à oportunidade de fazer a sua mudança!

*Tudo isso para honrarmos a  
nossa missão, que é promover  
a educação de qualidade nas  
diferentes áreas do conhecimento,  
formando profissionais  
cidadãos que contribuam para  
o desenvolvimento de uma  
sociedade justa e solidária.*

**Reitor**  
Wilson de Matos Silva



# MINHA HISTÓRIA MEU CURRÍCULO



Aqui você pode conhecer um pouco mais sobre mim, além das informações do meu currículo.

## Edson Orivaldo Lessa Junior

Olá, sou Edson Orivaldo Lessa Junior e atuo diretamente no ciclo de desenvolvimento de software e no Discovery de novas oportunidades. Já trabalhei como desenvolvedor, analista de requisitos, analista de processo e, hoje, com foco na melhoria dos processos e transformação digital das organizações. Atualmente, tenho focado na análise de negócio, no alinhamento de expectativas, negociação de prazos, funcionalidade e metas de tal forma que contribua com a melhoria contínua e sempre com objetivo de entrega de valor aos clientes.

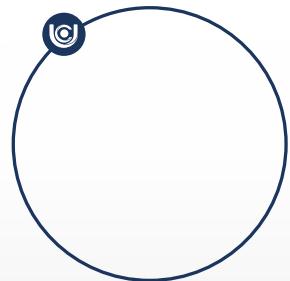
Sou entusiasta Agile e fomentador de transformação digital em organizações por meio Business Process Manager, gerenciando escopo, cronograma, custo, comunicação e aplicando metodologias e ferramentas de análise e melhoria de processo com ferramentas e técnicas de BPM Ágil, Simulação, PDCA, Análise SWOT, Cadeia de Valor, LEAN, Six Sigma, GAP Analysis, Kaizen, BPMN, SPEM-OMG e Kanban. Também atuo como professor universitário, palestrante e coordenador de trilhas.

# **MINHA HISTÓRIA**

# **MEU CURRÍCULO**

## **Rafael Orivaldo Lessa**

Olá, sou Rafael Orivaldo Lessa, sou VP de Engenharia na Neoway e com solução de Big Data Analytics, atuo na organização dos times de desenvolvimento para criar um ambiente inovador com times high performance e com uma cultura de agilidade. Tenho extensa experiência em gestão de equipes e desenvolvimento, inovação, transformação digital, ágil e lean, para transformar necessidades em soluções inovadoras que agregam valor para o negócio. Sou altamente orientado a resultados, gosto de formar, gerir e mentorar times para entregar soluções que agregam valor ao cliente e utilizando a melhor tecnologia para resolver a necessidade do cliente. Também atuo como Professor, em cursos de Sistemas de Informação, nas áreas de Modelagem de Processos, Automação de Processos, Engenharia de Software, Interface Humano Computador, Qualidade de Software e para a Pós-Graduação de Engenharia de Software nas áreas de Fundamentos em Business Process Management e Engenharia de Requisitos. Gosto de jogar futebol, andar de roller e ir na praia com a família.



**Aqui você pode  
conhecer um  
pouco mais sobre  
mim, além das  
informações do  
meu currículo.**

# **PROVOCACÕES INICIAIS**

## **ESTRUTURA DE DADOS**

A necessidade da informação é uma realidade que todos buscamos com maior velocidade e objetividade. Todos precisamos, de alguma forma, gerenciar e organizar as diversas informações que estão disponíveis, perceba o quanto de informação se perde como simples dados, ou seja, soltos sem um sentido. Como as soluções computacionais podem auxiliar na estruturação e organização dos dados para proporcionar uma melhor informação aos usuários?

De acordo com o Instituto Gartner, até 2020, é possível que haja um total de 40 trilhões de gigabytes de dados no mundo, com o advento do uso massivo de mídias sociais, e no cenário de home office esse número tende a aumentar significativamente. Sem uma efetiva gestão computacional e de estrutura de dados, podemos entrar em uma era de caos informacional e sem direcionamento de conteúdo.

Você mesmo já deve ter vivenciado este problema com suas redes sociais, como facebook, twitter e sem falar nos grupos digitais no qual a grande quantidade de informação que chega até você acaba por ser, em muitos casos, problemática no gerenciamento e na qualidade da informação para suprir suas necessidades.

Você consegue organizar todas as informações que recebe de forma simples e rápida? Quantas destas informações você ignora?

Você já deve ter percebido que a organização e estruturação dos dados é essencial nos próximos anos que virão: a forma que os profissionais devem estruturar os dados para que seja possível a manipulação e recuperação da informação de forma cada vez mais eficiente e eficaz.

Agora que você já compreendeu o quanto que a informação e os dados fazem parte da sua vida, e mais, que o grande volume que se apresenta se faz indispensável à estruturação e organização dos dados para que possa compor a informação necessária para as pessoas, procure criar uma cadeia de dados compostos pelos seus contatos de telefone. Gere esta cadeia ou lista em uma ordem aleatória e tente encontrar uma pessoa apenas utilizando o recurso da leitura. Após isso, organize em ordem alfabética e utilizando somente o recurso da leitura para encontrar o mesmo nome.

Cada vez que realiza a ação de procurar um contato em uma lista desorganizada e outra em uma estrutura organizada vai depender de alguns fatores.

Procure analisar o resultado de tempo e condições que levaram a uma apuração de tempo em um conjunto de contatos que você elegeu na sua busca. Procure identificar a média de tempo e qual forma se tornou melhor. Qual fator levou um menor tempo em uma busca na lista desorganizada e qual fator influenciou na busca do contato na lista organizada em ordem alfabética.

# **RECURSOS DE IMERSÃO**



## **REALIDADE AUMENTADA**

Sempre que encontrar esse ícone, esteja conectado à internet e inicie o aplicativo Unicesumar Experience. Aproxime seu dispositivo móvel da página indicada e veja os recursos em Realidade Aumentada. Explore as ferramentas do App para saber das possibilidades de interação de cada objeto.



## **RODA DE CONVERSA**

Professores especialistas e convidados, ampliando as discussões sobre os temas.



## **PÍLULA DE APRENDIZAGEM**

Uma dose extra de conhecimento é sempre bem-vinda. Posicionando seu leitor de QRCode sobre o código, você terá acesso aos vídeos que complementam o assunto discutido.



## **PENSANDO JUNTOS**

Ao longo do livro, você será convidado(a) a refletir, questionar e transformar. Aproveite este momento.



## **EXPLORANDO IDEIAS**

Com este elemento, você terá a oportunidade de explorar termos e palavras-chave do assunto discutido, de forma mais objetiva.



## **NOVAS DESCOBERTAS**

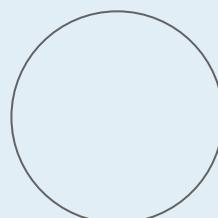
Enquanto estuda, você pode acessar conteúdos online que ampliam a discussão sobre os assuntos de maneira interativa usando a tecnologia a seu favor.



## **OLHAR CONCEITUAL**

Neste elemento, você encontrará diversas informações que serão apresentadas na forma de infográficos, esquemas e fluxogramas os quais te ajudarão no entendimento do conteúdo de forma rápida e clara

Quando identificar o ícone de QR-CODE, utilize o aplicativo **Unicesumar Experience** para ter acesso aos conteúdos on-line. O download do aplicativo está disponível nas plataformas:



# CAMINHOS DE APRENDIZAGEM

1

11

ESTRUTURA DE  
DADOS (TIPOS  
DE DADOS)

2

41

PILHAS E FILAS

3

71

LISTAS  
DINÂMICAS

4

109

TEORIA DE  
GRAFOS  
APLICADO A  
PYTHON

5

139

BUSCA EM  
GRAFOS



# Estrutura de dados (Tipos de dados)

Esp. Edson Orivaldo Lessa Junior

Esp. Rafael Orivaldo Lessa

## OPORTUNIDADES DE APRENDIZAGEM

Entre as áreas do conhecimento e as atividades humanas, talvez nenhuma supere a Computação quanto à convergência entre simplicidade e complexidade. A estrutura de dados é, fundamentalmente, o início de um bom desenvolvimento de algoritmos que podem gerar um aumento do desempenho e na mitigação dos problemas. As experiências acumuladas demonstram que o entendimento dos fluxos de processamento da informação é utilizado para uma melhoria constante na técnica que melhor representa as estruturas dos dados. Nesta etapa, iremos aprofundar no dado, em formas de armazenar, escalabilidade e como utilizar a linguagem de programação Python. Navegar pelas próximas páginas, certamente, vai propiciar a você um engajamento estimulante resultando em mais conhecimento sobre a estrutura de dados, que se torna mais interessante e intrigante quanto mais sabemos e atuamos nele.



Atualmente, as pessoas estão mais conectadas à internet e fazendo com que se produza um grande volume de informações. As soluções computacionais têm o desafio de automatizar as tarefas em um processo de desenvolvimento, que envolve hardwares e softwares. Nesta grande evolução de automação e conexão da internet e do mundo computacional, precisamos aprimorar os nossos métodos de desenvolvimento e melhorar a manipulação dos nossos dados, para facilitar a guarda e busca dessas informações. Uma disciplina de desenvolvimento, que auxilia neste desafio é a Estrutura de Dados e, com esta disciplina, começamos a estudar os diversos mecanismos de organização de dados para atender aos diferentes requisitos de processamento de forma eficiente. A estrutura de dados vem para auxiliar no desenvolvimento de soluções computacionais visando escolher a estrutura mais adequada. Nesse sentido, precisamos entender o objetivo a ser atendido, analisando a melhor forma de como organizar os dados, identificar os métodos de armazenamento eficiente dos dados e quais códigos e algoritmos são mais propícios para realizar as operações de manipulação dos dados da estrutura. Assim, pode-se basear nos tipos de armazenamento vistos dia a dia, ou seja, são transformações de uma forma de armazenamento já conhecida e utilizada no mundo real modificada para o mundo computacional. Por isso, cada tipo de estrutura de dados possui suas características e a sua correta utilização dependendo do problema que deseja ser resolvido.

Com os tipos de dados definidos, o desenvolvedor tem outro desafio que é escolher os processos de armazenamento e organização de estruturas como listas, pilhas, árvores, entre outros, umas destas técnicas realiza um processo, seja ele de organizar, armazenar ou de acessar os dados que precisam ser utilizados.

É importante entendermos os termos tipos de dados, estrutura de dados e tipos abstratos de dados que, embora sejam parecidos, possuem significados diferentes. Os tipos de dados são utilizados pelas linguagens de programação para definir o conjunto de valores que uma variável pode assumir.

Por exemplo, se precisamos armazenar todos os dados de uma pessoa tais como Nome, CPF, Data de Nascimento e Endereço ou uma lista de compras do supermercado. Qual seria a melhor estrutura de dados a ser utilizada? Esse é um dos desafios que o desenvolvedor ou cientista de dados enfrenta no dia a dia, precisa entender o escopo e o problema para escolher a melhor estrutura de dados.

De acordo com o IDC (International Data Corporation), em 2020, seria possível ter um total de 40 trilhões de gigabytes de dados no mundo, ou seja, 2,2 milhões de terabytes. Com o advento do uso massivo de mídias sociais e no cenário de *home office*, esse número tende a aumentar significativamente. Este grande volume de dados nos faz refletir que temos que atender os pilares de velocidade, volume e variedade, ou seja, poder ter qualquer dado (texto, vídeo, imagens, áudio, *Internet of Things - IOT*) em tempo real e em grande volume.

Importante entender o significado de dados e informação, o DADO são os registros sem quaisquer análises, é a informação não tratada que ainda não apresenta relevância. Um dado não consegue transmitir uma mensagem. Já a INFORMAÇÃO é a estruturação e organização desses dados, ela é um registro que transmite uma mensagem, portanto, a informação é base para a produção de conhecimento e experiência de cada indivíduo no contexto de seu negócio.

Em alguns negócios, 1 segundo pode ser muito tempo para detectar uma fraude, liberar um pagamento, fazer um diagnóstico médico ou qualquer informação sensível a tempo que necessite desses dados o mais próximo do tempo real. Um exemplo disso foi divulgado, em 2016, pelo Instituto de Pesquisa Econômica Apli-

cada (IPEA), no qual soluções computacionais foram desenvolvidas para intervir instantaneamente diante de situações inesperadas no mercado de ações, esses sistemas são responsáveis por mais de 70% das decisões de compra e venda de ações nos Estados Unidos.

Um estudo conduzido pela DOMO, empresa especializada em Cloud, traz dados surpreendentes da quantidade de dados produzidos por minuto por várias empresas, e em 2020, viu uma crescente em algumas soluções devido ao cenário *home office*.

Alguns desses dados são (DOMO, 2020, on-line)<sup>1</sup>:

- Zoom tem 208 mil pessoas em conferências;
- TikTok é instalado 2.704 vezes em novos dispositivos;
- Spotify inclui 28 novas músicas;
- YouTube ganha 500 novas horas de vídeo enviados pelos usuários;
- Twitter recebe 319 novos usuários;
- 69.444 pessoas se candidatam à vaga no LinkedIn;
- Amazon envia 6.659 encomendas;
- 4,5 bilhões de pessoas com acesso à Internet;
- \$ 1 milhão por minuto em compras pela Internet;
- Whatsapp 41.666.667 milhões de mensagens.

Você pode perceber que a estrutura de dados é fundamental para que possamos construir soluções computacionais, que utilizem da melhor forma os recursos de hardware e software. Temos um grande potencial para fazer muita coisa com estes 40 trilhões de *terabytes* ou podemos tornar eles inúteis.

Tudo dependerá dos nossos esforços que trabalhamos com soluções computacionais, então podemos começar a entender melhor as estruturas classificando-as em duas categorias: homogêneas e heterogêneas. Ambas possuem o objetivo de organizar os dados e possibilitar funcionalidades que permitem explorar melhor os seus conteúdos.

Você deve estar se perguntando: como estas estruturas podem me ajudar a resolver problemas do cotidiano? Procure refletir a seguinte situação: se você tiver um carregamento com 100 produtos entre frutas e legumes, como você os classificaria da forma mais simples?

Agora, vamos organizá-los um pouco diferente: imagine que você necessita organizar estes produtos em diversas visões diferentes como: tipo (tomate, banana, brócolis etc), cor, acidez, docura, peso e região de origem. Como seria a sua estrutura para listar os produtos por meio de qualquer uma destas características?

Você já deve ter percebido que, atualmente, a organização e estruturação dos dados é essencial e está ligada direta e indiretamente em todos os segmentos da sociedade para tomada de decisões importantes - até mesmo em ações do dia a dia, como dirigir com um GPS monitorando o trânsito e definindo a melhor rota.

Os profissionais de tecnologia devem estruturar os dados para que seja possível organizar, manipular e recuperar a informação de forma adequada a velocidade, variedade e volume para atender às necessidades de maneira eficiente e eficaz.

Comente no seu diário de bordo o seu entendimento de como os profissionais de tecnologia podem ajudar a introduzir essa disciplina na estrutura de dados cotidiano das pessoas. E ainda conte um pouco de como os dados influenciam nas suas decisões atuais e quais estruturas poderiam ser utilizadas.

## DIÁRIO DE BORDO



Como vocês já viram, a utilização dos dados de forma estruturada auxilia muito na resolução dos problemas que iremos enfrentar no dia a dia de trabalho. Quando você entende a diferença entre os tipos de armazenamentos de dados possibilita uma melhor utilização dos recursos disponíveis. Mas, então, o que são estruturas homogêneas e heterogêneas?

**Estruturas Homogêneas:** armazenam dados únicos. Portanto, quando você necessitar armazenar uma informação por uma única característica, você poderá utilizar uma estrutura homogênea dentro das estruturas existem as variáveis, vetores e matrizes.

**Estruturas Heterogêneas:** armazenam dados que possuem múltiplas características, ou seja, uma variável que armazena diferentes tipos de dados (características) de informações representando um único elemento.

Veja o quadro comparativo de cada tipo de dados e a sua respectiva estrutura:

Tipos de dados	Homogêneas	Heterogêneas
Vetor	X	
Matriz	X	
Set		X
Tuplas		X
Dictionary		X

Quadro 1 – Comparativo de tipo de dados por estruturas / Fonte: o Autor.

**Variáveis:** são estruturas simples que armazenam em um espaço de memória um único dado. Esta estrutura é fundamental na criação de algoritmos e códigos de software, porém ela não representa uma escala. Por exemplo, se você tiver que armazenar na memória um nome de pessoa, uma variável resolve. Agora, vamos ver na prática como é possível resolver um problema usando variáveis. A seguir, veremos exemplos de estruturas homogêneas, abra o programa no seu editor, teste e acompanhe a explicação deste programa que calcula uma equação do segundo grau como o emprego das variáveis.

```

1
2 def delta(a, b, c):
3     return (b ** 2 - 4 * a * c)
4
5
6 def equacao(a, b, c):
7
8     resultadoDelta = delta(a, b, c)
9     if resultadoDelta < 0:
10         return "Não possui resultados reais"
11     else:
12         print("Delta: {}".format(resultadoDelta))
13
14     x1 = (((-1) * b) + (resultadoDelta ** (1 / 2))) / (2 * a)
15
16     x2 = (((-1) * b) - (resultadoDelta ** (1 / 2))) / (2 * a)
17
18     return f'Os resultados possíveis são \nX1: {x1:.0f}\nX2: {x2:.0f}'
19
20 def main():
21
22     print("Calculo de equação do 2º grau\n ax²+bx+c=0")
23     print("\n")
24
25     try:
26
27         valorA = int(input("Informe o valor de 'a': "))
28
29         valorB = int(input("Informe o valor de 'b': "))
30
31         valorC = int(input("Informe o valor de 'c': "))
32
33         print(equacao(valorA, valorB, valorC))
34
35     except ValueError as erro:
36
37         print("O valor informado não é inteiro")
38
39
40 main()
41
42
43

```

Vamos entender o código anterior:

- **Linhas 2 e 3:** é uma função responsável por calcular o delta da equação do segundo grau. Veja que logo após o nome da função espera-se 3 parâmetros, que são variáveis de entradas das funções. Estas variáveis serão utilizadas na fórmula que está na linha 3.
- **Linha 6:** novamente uma função que onde esperamos 3 parâmetros para que possam ser utilizadas no decorrer do código.
- **Linha 8:** realizamos a chamada da função delta, que é responsável por realizar o cálculo desta parte da equação. Perceba que estamos atribuindo uma variável “resultadoDelta”. Será responsável por receber o resultado do cálculo.
- **Linhas 9 a 12:** nesta parte do código, realizamos verificação se é possível calcular a equação do segundo grau e emitir a mensagem apropriada.
- **Linhas 14 a 16:** realizamos o cálculo com base nas variáveis e atribuímos o resultado em duas variáveis “x1” e “x2”.
- **Linha 18:** é formatado o texto de retorno para que os resultados sejam possíveis de ser apresentado de forma legível.
- **Linhas 20 a 23:** é definido a função principal (sem parâmetros de entrada) e informativos para o usuário.
- **Linha 25:** é aberto um tratamento de erro, ou seja, se será tentado executar o bloco contido no try.
- **Linhas 27 a 31:** captura dos valores digitados pelo usuário. Cada valor será armazenado em uma variável.
- **Linha 33:** é executado a impressão do cálculo contido na função equação no qual é passado as variáveis coletadas para a função. Perceba que nessa tratativa, espera-se que a função retorne a mensagem correta para que possa ser exibida ao usuário. A mensagem de retorno está na linha 18.
- **Linhas 35 a 37:** é coletado o erro quando ocorrer, ou seja, quando uma equação não for possível calcular, vamos esperar que ocorra um erro que irá parar a execução e exibir a mensagem da linha 37.

Veja que o programa resolve o cálculo da equação apresentada, mas se quiser armazenar os resultados, como faria? Ainda, se você necessitasse armazenar n cálculos, como você poderia manter estruturas que fosse possível manter os registros? A tarefa parece impossível. Felizmente, uma grande parte das linguagens possuem estruturas que nos auxiliam em promover a escala.

## Estrutura de dados Homogêneas

**Vetores** ou *Arrays* são estruturas em lista que possibilitam armazenamento de estruturas homogêneas e permitem a escala dos dados. Pense em vetores como uma lista de dados, por exemplo, suponha que queira organizar em lista os nomes dos alunos poderíamos fazer:

1. Maria;
2. Gui;
3. Jô;
4. Edu;
5. João;
6. Carlos;
7. Joaquina.

Nesta relação, se você quiser saber qual o nome da posição 3, terá o resultado Jô. Neste exemplo, perceba que os dados são todos do mesmo tipo, isto é, todos os dados são do tipo de caractere, e você consegue recuperar o dado por meio de um índice. Vejamos em execução, abra o programa no seu editor e acompanhe como o Python pode definir os vetores.

```
1 print("Definição de Vetores\n")
2
3
4
5 def main():
6     vetorVazio = []
7     vetorVazio2: list
8
9     # imprime o tamanho do vetor
10    print("Tamanho do vetor vazio é: ", len(vetorVazio))
11    for i in range(9):
12        # acrescenta um valor no final do vetor
13        vetorVazio.append(i)
14
15    print("Novo Tamanho do vetor vazio é: ", len(vetorVazio))
16    print("Valores agora no vetor vazio são: ", vetorVazio)
17
18    vetorInteiros = [1, 2, 3, 4, 5, 6, 7]
19
20    # imprime [1, 2, 3, 4, 5, 6, 7]
21    print("Valores do vetor inteiros são: ", vetorInteiros)
22
23    for i in range(0, len(vetorInteiros)):
24        # imprime a posição e o valor da posição
25        print("O Valor na posição ", i, " é: ", vetorInteiros[i])
26
27    vetorCaracteres = ["Maria", "Gui", "Jô", "Edu", "João", "Carlos", "
28 Joaquina"]
29    for i in range(len(vetorCaracteres)):
30        print("O Valor na posição ", i, " é: ", vetorCaracteres[i])
31
32    vetoresReais = [1.2, 3.5, 12.3, 4.2]
33    print("Valores no vetorReais são: ", vetoresReais)
34
35
36 main()
37
38
39
```

Vamos falar um pouco deste código:

- **Linha 2:** exibe um texto na tela;
- **Linha 5:** função principal;
- **Linhas 6 a 7:** define dois vetores, perceba que das duas formas é válido criar um vetor vazio;
- **Linha 10:** imprime o tamanho do vetor, neste caso, o resultado esperado zero assim como a mensagem para o usuário;

- **Linhas 11 a 13:** inclui o valor da variável “i” no vetor “vetorVazio”. Veja que o método “append” adiciona um valor vetor;
- **Linha 15:** exibe a quantidade do vetor “vetorVazio”;
- **Linha 16:** exibe o conteúdo do vetor “vetorVazio”;
- **Linha 18:** define um vetor com valores;
- **Linha 21:** exibe o conteúdo do vetor;
- **Linhas 23 a 25:** exibe a posição e o valor da respectiva posição no vetor, perceba que podemos inferir a posição que desejamos recuperar do vetor para encontrar mais facilmente os valores;
- **Linha 27:** cria um vetor com valores de caracteres;
- **Linhas 29: a 30:** exibe a posição e o valor da respectiva posição no vetor;
- **Linhas 32 a 33:** define um valor de decimais e exibe os seus valores;
- **Linha 36:** chama a função principal.

Fácil, não acha? Vetores são uma parte fundamental no desenvolvimento, pois permite a escala de elementos. A escala permite trabalhar dados de diversas origens e objetivos e não são finais, existem outras estruturas que também permitem utilizar variáveis em escala de outras formas.



**Matrizes:** os vetores tratam dos dados em lista chamadas de uma dimensão, ou seja, apenas um índice é responsável por referenciar os dados armazenados. Porém existem situações que precisamos de mais índices para referenciar os dados em memória, quando esta necessidade é evidente utiliza-se a técnica de Matriz para estruturar os dados em memória, no Python existe a possibilidade de criar vetores multidimensionais, as matrizes são divididas em linhas e colunas e para identificar um valor, precisamos saber o número da linha e da coluna para recuperá-lo. Mais uma vez, vamos abrir o código no editor e acompanhar o código de matrizes.

```
1 def bidimensional():
2     bidimensional = [[1, 2, 3, 4], [9, 8, 7, 6]]
3
4
5     for i in range(0, len(bidimensional)):
6         # neste for é acessado o primeiro nível da matriz
7         for j in range(0, len(bidimensional[i])):
8             # neste for é acessado o segundo nível, onde contém os dados
9             print("Bidimensional Valor [", i, ", ", j, "]", bidimensional[i][j])
10
11
12 def tridimensional():
13     tridimensional = [[[1, 2, 3, 4], [4, 3, 2, 1]], [[9, 8, 7, 5], [4, 5, 6, 7]]]
14
15     for i in range(0, len(tridimensional)):
16         # neste for é acessado o primeiro nível da matriz
17         for j in range(0, len(tridimensional[i])):
18             # neste for é acessado o segundo nível da matriz
19             for k in range(0, len(tridimensional[i][j])):
20                 # neste for é acessado o segundo nível, onde contém os dados
21                 print("Tridimensional Valor [", i, ", ", j, ", ", k, "]: ", tridimensional[i][j][k])
22
23
24 def executarAplicacao():
25     print("Exemplo de Matriz (vetor bidimensional)")
26     bidimensional()
27     print("-----\n\n")
28
29     print("Exemplo de Matriz de três dimensões (vetor tridimensional)")
30     tridimensional()
31     print("-----")
32
33
34 executarAplicacao()
35
36
37
```

Vamos passar os principais pontos do programa para entendermos melhor o código

- **Linha 3:** definimos uma matriz, veja que uma matriz é em essência um vetor que armazena vetores. Neste caso, temos na posição 0 o vetor com dados de 1 a 4 e na posição 1 o vetor com dados de 9 a 6;
- **Linha 5:** laço de repetição que controla a variável “*i*” que é utilizada para definir a posição do vetor mais externo;
- **Linha 7:** laço de repetição que controla a variável “*j*” que é utilizada para definir a posição dos vetores internos;
- **Linha 9:** imprime o valor da matriz “bidimensional”;
- **Linha 13:** define uma matriz de 3 dimensões, veja que agora teremos um vetor que contém um outro vetor que por sua vez, contém um novo vetor;
- **Linhas 15 a 19:** são os laços de repetição que controlam as variáveis usadas para definir a posição da matriz 3 dimensões (ou vetor tridimensional);
- **Linhas 24 a 34:** são formatadas as exibições para o usuário.

Algumas linguagens permitem utilizar vetores de n dimensões, possibilitando uma estrutura complexa e adaptativa para diversos cenários.

## Estrutura de dados Heterogêneas

**Tuplas** uma tupla é uma estrutura bastante similar a um vetor, mas o seu funcionamento difere, porque uma vez inseridos os elementos em uma tupla não podem ser alterados, diferente do vetor no qual podem ser alterados livremente. Sendo assim, em um primeiro momento, podemos pensar em uma tupla como um vetor que não pode ser alterada. As tuplas possuem diversas características dos vetores, porém os usos são distintos. Os vetores são destinados a serem sequências homogêneas, enquanto que as tuplas são estruturas de dados heterogêneas. Portanto, a tupla é utilizada para armazenar dados de tipos diferentes, enquanto que os vetores são para dados do mesmo tipo. (PERKOVIC, 2016). Vejamos o exemplo:

```

1
2 def tuplaInicio():
3     primeiraTupla = (1, 3, "Teste", 0.4) # define uma tupla
4     print("Primeira tupla: ", primeiraTupla)
5
6     segundaTupla = 3, 6, 9 # define uma nova tupla
7     print("Segunda tupla: ", segundaTupla)
8
9     print("Tamanho da tupla: ", len(primeiraTupla)) # informa o tamanho da
tupla
10
11     primeiraTupla += "Novo elemento",
12     print("Acréscima um elemento na tupla", primeiraTupla)
13
14     print("Tipo de dados: ", type(primeiraTupla))
15
16     print("Referência do elemento 3: ", primeiraTupla[2])
17
18     primeiraTupla += (4.8, 9) # concatenação entre tuplas
19     print(primeiraTupla)
20
21     primeiraTupla = primeiraTupla * 6 # multiplica a quantidade de elementos
pelo número informado
22     print(primeiraTupla)
23
24
25 tuplaInicio()
26

```

Vamos entender o código passando pelas principais linhas:

- **Linha 3:** define uma tupla, perceba que a definição da tupla é feita pelos valores;
- **Linha 6:** define uma tupla, neste caso, não utilizamos o parênteses para agrupar;
- **Linha 11:** operação de adicionar um novo elemento;
- **Linha 14:** exibe o tipo do dado armazenado na variável.

As tuplas são estruturas versáteis, que podem ser utilizadas no Python, porém não são todas as linguagens que possuem esta estrutura.

**Set :** Python também inclui um tipo de dados para conjuntos, chamado set. Uma diferença essencial é que o set é uma coleção desordenada de elementos que não possui elementos em duplicidade e índices para referência. Desta forma, poderemos utilizar o set, quando há a necessidade de manter um dado sem duplicidade. O set pode ainda manter as operações matemáticas como união, interseção, diferença e diferença simétrica (PERKOVIC, 2016). Veja como ocorre:

```

1 def conjunto():
2     primeiroSet = {6, "Dez", 4} # criação
3     print("Conjunto: ", primeiroSet)
4
5     primeiroSet.add("Um")
6     print("Adicionando um elemento ao conjunto: ", primeiroSet)
7
8     primeiroSet.remove("Dez") # caso não exista lança um erro
9     print("Exemplo de set com remove: ", primeiroSet)
10
11    primeiroSet.discard("Dez") # caso não exista nenhum erro é lançados
12    print("Exemplo de set com discard: ", primeiroSet)
13
14    print("Tamanho do conjunto: ", len(primeiroSet))
15
16    segundoSet = {3.2, 4, True, "Novo", 20, 5, "Um"}
17
18    uniao = primeiroSet.union(segundoSet)
19    print("União: ", uniao)
20
21    intersecao = primeiroSet.intersection(segundoSet)
22    print("Intersecção: ", intersecao)
23
24
25
26 conjunto()
27

```

Agora que você testou, vejamos as principais linhas:

- **Linha 3:** define um *set*;
- **Linha 6:** adiciona um elemento no *set*;
- **Linha 9:** remove um elemento, veja que os elementos são removidos por igualdade. Com o “remove”, obrigatoriamente, deve existir o elemento para que não ocorra erro. Deve ser utilizado, quando você deseja controlar o cenário que não exista;
- **Linha 12:** remove um elemento caso exista, mas desconsidera se não encontra;
- **Linha 19:** une dois *set* e atribui a um terceiro;
- **Linha 22:** cria um *set* com a intersecção de outros dois.

Em **Python** existem dois tipos de conjunto: o *set* e o *frozenset*. Em praticamente tudo, eles são iguais, a única e fundamental diferença entre ambos é que o *frozenset* é imutável e, uma vez criado, não pode ter seus membros alterados, incluídos ou removidos. Vejamos o exemplo:

```

1
2 def conjuntoolmutavel():
3     listaUm = ["maçã", "banana", "morango"]
4     print("Exemplo lista um:", listaUm)
5
6     conjuntoCongeladoUm = frozenset(listaUm)
7     print("Exemplo conjunto um:", conjuntoCongeladoUm)
8
9     conjuntoDois = {"Gui", 34, True, 40, "masculino"}
10    print("Conjunto dois: ", conjuntoDois)
11
12    conjuntoCongeladoDois = frozenset(conjuntoDois)
13    print("Conjunto dois congelado: ", conjuntoCongeladoDois)
14
15    contador = 1
16    for extracao in conjuntoCongeladoDois:
17        print(f'Extraindo elemento {contador}: ', extracao)
18        contador += 1
19
20
21 conjuntoolmutavel()
22

```

Vamos ver os principais elementos:

- **Linha 3:** define uma lista;
- **Linha 6 e linha 21:** define um conjunto congelado, a função *frozenset* possui esta propriedade de congelar um vetor ou *set*.

Esta característica de congelar o conjunto impossibilita que IDEs (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado) realizem o autocomplete de funções para alteração do conjunto. Porém se você escreve o código como remove, as próprias IDEs não o criticam como erro. Veja o código:



The screenshot shows a Python script running in a code editor. The code defines a frozen set and attempts to modify it using the `remove` method. The code is as follows:

```

mirror_mod.use_x = False
mirror_mod.use_y = True
mirror_mod.use_z = False
elif operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

#selection at the end -add back the deselected mirror modifier object
mirror_obj.select= 1
modifier_obj.select=1
bpy.context.scene.objects.active = modifier_obj
print("Selected " + str(modifier_obj)) # modifier obj is the active ob
    mirror_obj.select = 0
bpy.context.scene.objects.active = mirror_obj

```

The code editor highlights the line `mirror_mod.remove('banana')` in red, indicating a syntax error. This demonstrates that while the IDE prevents modification of a frozen set, it does not catch common mistakes like attempting to remove elements from it.

```

1 def definindoConjunto():
2     conjuntoBase = {"maçã", "banana", "morango"}
3
4     conjuntoSet = set(conjuntoBase) # criando um cópia do conjunto
5     print("Tipo conjuntoSet: ", type(conjuntoSet))
6     print("Conjunto:", conjuntoSet)
7
8     conjuntoSet.remove("morango")
9     print("Resultado da remoção do conjuntoSet", conjuntoSet)
10
11    conjuntoCongelado = frozenset(conjuntoBase)
12    print("Tipo conjuntoCongelado: ", type(conjuntoCongelado))
13    print("Conjunto congelado: ", conjuntoCongelado)
14    conjuntoCongelado.remove("banana")
15    print("Resultado da remoção do conjuntoCongelado: ",
conjuntoCongelado)
16
17
18 definindoConjunto()
19

```

Veja que neste código, estamos forçando o erro, isto é, se executar o código anterior irá resultar em um erro na execução do programa.

```

2 Traceback (most recent call last):
3   File "D:\Users\ledson\PycharmProjects\Unidade 1\impressao\exemplo7\
conjuntoImutavelComErro.py", line 19, in <module>
4     definindoConjunto()
5   File "D:\Users\ledson\PycharmProjects\Unidade 1\impressao\exemplo7\
conjuntoImutavelComErro.py", line 15, in definindoConjunto
6     conjuntoCongelado.remove("banana")
7 AttributeError: 'frozenset' object has no attribute 'remove'
8 Tipo conjuntoSet: <class 'set'>
9 Conjunto: {'banana', 'morango', 'maçã'}
10 Resultado da remoção do conjuntoSet {'banana', 'maçã'}
11 Tipo conjuntoCongelado: <class 'frozenset'>
12 Conjunto congelado: frozenset({'banana', 'morango', 'maçã'})
13
14 Process finished with exit code 1
15

```

**Dictionary:** os dicionários são estruturas de dados que são usados basicamente para mapear chaves e valores. Essa estrutura é utilizada tanto para manipular, quanto para armazenar os dados. Os índices(chaves) podem ser praticamente

qualquer valor e pode estar associada a somente um valor. A criação de dicionários pode ser feita por meio de diversos modos, usando listas ou conjuntos, de pares que associam um primeiro componente que é o valor da chave a seu valor correspondente (PERKOVIC, 2016). Vejamos o exemplo a seguir.

```
1 import datetime
2
3
4 def dionario():
5     dionario = {"nome": "Edu", "idade": 23, "maiordade": True, "salario":
4567.22,
6         "filhos": [{"nome": "Gu", "idade": 3}, {"nome": "Jô", "idade": 10}],
7         "numerosPrimos": [1, 2, 3, 5, 7]}
8
9     print("Dicionário: ", dionario, "\n")
10
11    print("Exibir a informação da idade: ", dionario["idade"], "\n")
12
13    print("Teste do get: ", dionario.get("caso", 0), "\n")
14
15    print("Retorna uma tupla (chave, valor): ", dionario.items(), "\n")
16
17    print("Retorna somente chaves: ", dionario.keys(), "\n")
18
19    print("Retorna somente os valores: ", dionario.values(), "\n")
20
21    dionario.update({"dataNascimento": datetime.datetime(1980, 2, 20)})
22    print("Imprimindo os dados do dicionário")
23    for chave, valor in dionario.items():
24        if chave == "dataNascimento":
25            print("Exibindo os dados da chave ", chave, " com o valor ", valor.
strftime("%d/%m/%Y"))
26
27        elif chave == "filhos":
28            for i in range(len(valor)):
29                for chaveFilhos, valorFilhos in valor[i].items():
30                    print("Exibindo os dados da chave dos filhos ",
31                        chaveFilhos, " com o valor dos filhos ",
32                        valorFilhos)
33
34        elif chave == "numerosPrimos":
35            for i in range(len(valor)):
36                print("Valor primos: ", valor[i])
37
38    else:
39        print("Exibindo os dados da chave ", chave, " com o valor ", valor)
40
41
42 dionario()
43
```

Vejamos alguns detalhes dos códigos:

- **Linha 5:** define um dictionary;
- **Linha 11:** exibe o valor da chave “idade”;
- **Linha 15:** permite retornar uma tupla que conterá chave e valor;
- **Linha 17:** irá retornar somente as chaves do dictionary;
- **Linha 19:** irá retornar somente os valores do dictionary;
- **Linha 21:** adiciona um novo valor;
- **Linhas 23 a 39:** exibe o conteúdo do dictionary formatado para o usuário final.

A operação básica sobre dicionários é a indexação. Um dicionário indexado por uma chave fornece o valor correspondente à chave indexada. Se usado como uma expressão (para obter um valor), ocorre uma exceção (KeyError), se o valor fornecido não for uma chave do dicionário, mas se usado como uma variável, do lado esquerdo de um comando de atribuição, a indexação permite adicionar nova chave ao dicionário.

Utilizando o método `get` permite recuperar um valor de forma controlada, isto é, ao invés de lançar um erro de referência errada, o método `get` permite recuperar o elemento e caso não exista retorna `None`. Porém o método `get` permite que você lance um valor default para a ação, caso não seja identificado a chave de referência. No exemplo da linha 13, quando não encontrar a chave `caso`, irá retornar o valor zero.

```
12  
13     print("Teste do get: ", dicionario.get("caso", 0), "\n")  
14
```

Outro método útil é o `pop` que possibilita a remoção da chave de referência, assim como o seu valor. Mas é importante que se tenha o cuidado de garantir que a chave existe, pois a não existência irá lançar uma exceção.

**Ponteiros:** são muito úteis, quando uma variável tem que ser acessada em diferentes partes de um programa. Um ponteiro é uma variável que armazena o endereço da memória, que aponta para outra variável, ou seja, eles fazem a referência a um dado armazenado em outro local de memória.

Você pode ver que há uma diferença entre realizar uma cópia do dado e fazer uma referência do dado. Quando há uma cópia, qualquer alteração da cópia não reflete no original, o que não ocorre quando há um ponteiro. Quando realizado um ponteiro para um determinado dado, qualquer alteração será refletida no dado original.

Esta característica pode parecer inconveniente a um primeiro olhar, pois é necessário ter um cuidado adicional.

Porém existem muitos benefícios em usar ponteiros como:

Substituir a manipulação de vetores/matrizes com eficiência;

Passar valores e mudar valores dentro de funções;

Manipular arquivos;

Aumento de eficiência para algumas rotinas;

Possibilitar a alocação dinâmica de memória.

Como podemos observar o uso de ponteiros nas listas encadeadas, no qual um ponteiro armazena o endereço do próximo elemento da lista, assim para percorrer a lista, basta seguir os ponteiros ao longo dela.

Importante destacar que os ponteiros em linguagens de baixo nível como o C necessitam declarar na variável se ela é um ponteiro. Outras linguagens como o Python tratam a definição do ponteiro de forma implícita. Passagens de parâmetros em Python, naturalmente, são ponteiros, o que significa que até mesmo variáveis que contém somente um inteiro serão tratadas com referências. Isso torna a linguagem mais dinâmica e de fácil leitura, porém mais custosa em termos de alocação de memória e também não tão segura.

Vamos entender melhor o uso dos ponteiros no Python utilizando uma referência com lista utilizando em uma passagem de parâmetro.

```
1 def referencia(novaLista):
2     novaLista[1] = "novo valor"
3
4
5 def ponteiro():
6     lista = [] # criar lista vazia
7     print("Tamanho da lista", len(lista))
8
9     lista = ['O carro', 'peixe', 123, 111] # criando lista com os elementos
10
11    for elemento in lista:
12        print("Original :", elemento)
13
14    referencia(lista) # chamando a referência
15    print("-----")
16
17    for elemento in lista:
18        print("Teste :", elemento)
19
20
21 ponteiro()
22
```

No exemplo, perceba um vetor com determinados dados são passados por parâmetro a uma função no qual altera um dado. Pensando de forma rápida, podemos acreditar que a alteração realizada não irá refletir no vetor original, mas perceba como fica o resultado:

```
Tamanho da lista 0
Original : O carro
Original : peixe
Original : 123
Original : 111
-----
Teste : O carro
Teste : novo valor
Teste : 123
Teste : 111

Process finished with exit code 0
```

O valor na posição 1 da lista foi alterado mesmo que tenha sido em outros escopos do programa. Esta dinâmica oferece o benefício de que você não precisa retornar nenhum valor para atualizar a lista original, a referência para o dado original torna mais simples. Contudo, perceba que quando há a necessidade de realizar a cópia do valor, você deve fazer de forma explícita.

Lembre-se que um ponteiro armazena um endereço na memória, que aponta para o próximo elemento, seja ele de uma variável pré-definida ou alocada dinamicamente. Tome cuidado para não passar um endereço diretamente para um ponteiro, deixe para as classes e os métodos fazerem o controle. Caso passe um endereço, você pode acabar acessando uma porção da memória ocupada por outro programa ou pelo próprio sistema operacional e ocasionar um erro.

Até este momento, conseguimos ver como armazenar esse grande volume de informações que estamos vivendo no nosso mundo. Você aprendeu sobre variáveis, vetores, matrizes, tuplas, *dictionary* e ponteiros. Todos esses conceitos são bases para resolver os problemas relatados e criar sistemas com alta performance e escalabilidade.

**Gestão de Memória:** após entender melhor como funcionam as estruturas de armazenamento e identificar as suas diferenças é importante ter uma visão geral do gerenciamento de memória que o Python realiza. Claro que o objetivo é entender a estrutura geral da memória e como ela é gerenciada. O gerenciamento de memória em Python trabalha com uma estrutura de armazenamento privada contendo todos os objetos e estruturas de dados Python. O gerenciamento desta estrutura é assegurado internamente pelo gerenciador de memória do Python. Este gerenciador possui diferentes componentes que lidam com vários aspectos de gerenciamento de armazenamento dinâmico, como: compartilhamento, segmentação, pré-alocação ou cache.



Quando olhamos no nível mais baixo, um alocador de memória bruta garante que haja espaço suficiente na estrutura para armazenar todos os dados relacionados ao Python, interagindo com o gerenciador de memória do sistema operacional. Além do alocador de memória bruta, vários alocadores específicos de objeto operam na mesma estrutura e implementam políticas de gerenciamento de memória distintas adaptadas às peculiaridades de cada tipo de objeto. Por exemplo, objetos inteiros são gerenciados de forma diferente dentro da estrutura do que strings, tuplas e dicionários, porque os inteiros implicam em requisitos de armazenamento diferentes e compensações de velocidade/espaço. O gerenciador de memória Python, portanto, delega parte do trabalho aos alocadores específicos do objeto, mas garante que os últimos operem dentro dos limites estipulados.

Entenda que o gerenciamento de memória pelo Python é executado pelo próprio interpretador e que o usuário não tem controle sobre ele, mesmo que manipule ponteiros de objeto regularmente para blocos de memória. Na maioria das situações, entretanto, a memória será gerenciada pelo próprio Python.

Mas existem situações que é necessário alterar esta alocação de forma que o interpretador é estendido com novos tipos de objetos escritos em C. Consequentemente, sob certas circunstâncias, o gerenciador de memória Python pode ou não acionar ações apropriadas, como coleta de lixo, compactação de memória ou outros procedimentos preventivos. Observe que, ao usar o alocador de biblioteca C, a memória alocada para o *buffer* de E/S escapa completamente do gerenciador de memória Python.



### PENSANDO JUNTOS

Nesta unidade, abordamos as principais implementações para começarmos a organizar e armazenar os dados com os principais conceitos, como criar funções ou sub-rotinas, a manipulação de matrizes, a implementação de programas utilizando ponteiros em listas. Você acha que com esse conteúdo, você consegue armazenar os dados do dia a dia para tomada de decisões?



### EXPLORANDO IDEIAS

Estruturas Homogêneas e Heterogêneas são classificações quanto aos tipos de dados que são armazenados:

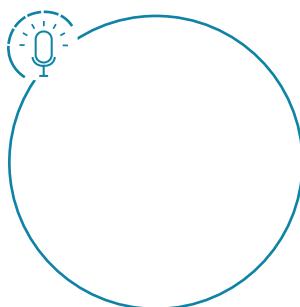
- *Estruturas homogêneas* armazenam dados únicos;
- *Estruturas heterogêneas* armazenam em uma variável diferentes tipos de dados.

Variáveis são estruturas simples, que armazenam em um espaço de memória um único dado. Vetores são estruturas que possibilitam armazenamento de dados com escalabilidade. Matrizes tratam a forma de alocação dos Vetores e aspecto multidimensional.

Uma tupla é uma estrutura bastante similar a um vetor, mas o seu funcionamento difere, porque uma vez inseridos os elementos em uma tupla não podem ser alterados.

Python também inclui um tipo de dados para conjuntos, chamado set. Uma diferença essencial é que o set é uma coleção desordenada de elementos, que não possui elementos em duplicidade e índices para referência.

Os dicionários são estruturas de dados que são usados basicamente para listas mapeadas por meio de chaves e seus respectivos valores.



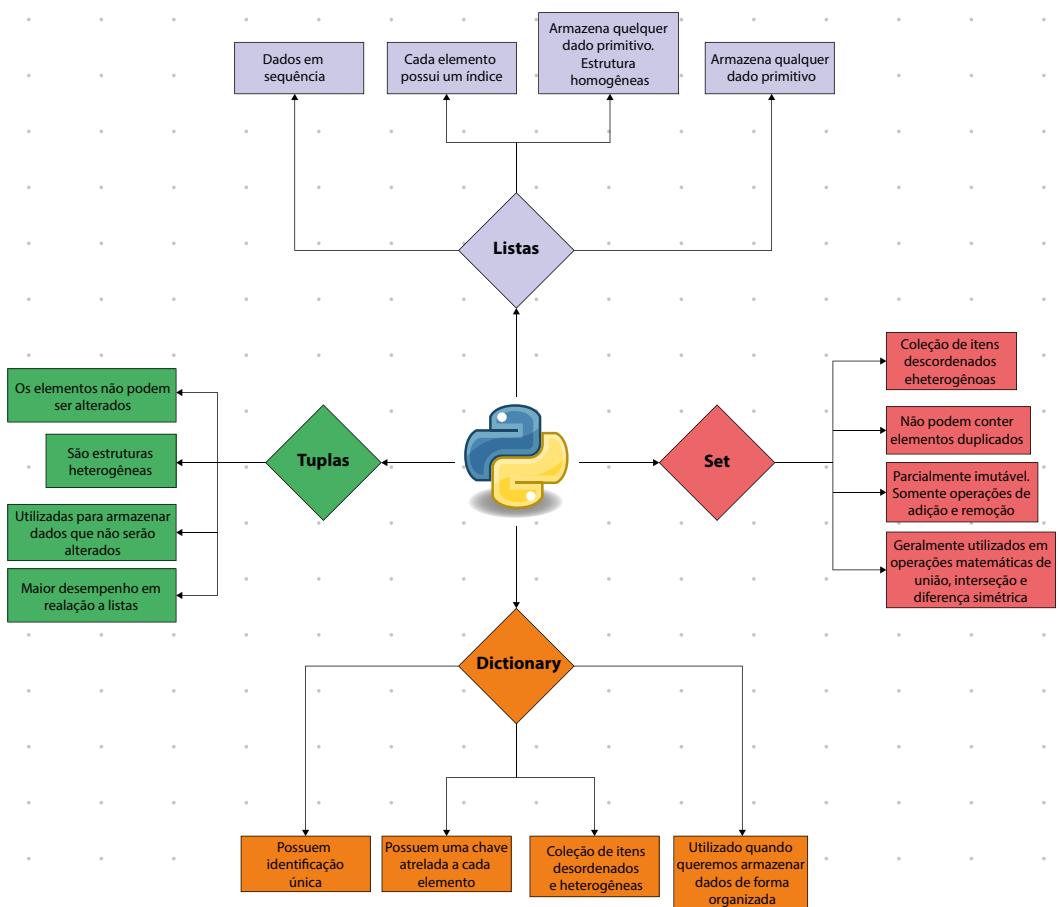
### Você precisa saber estrutura de dados para trabalhar como desenvolvedor ou cientista de dados?

Em um momento de mudanças e inovações na área de tecnologia, os profissionais de tecnologia tem que estar cada vez mais capacitados para um mercado tão exigente, novas profissões e oportunidades vão surgindo e outras vêm sendo transformadas. Uma das profissões que vem chamando a atenção é a de cientista de dados ou data science. Neste Podcast, vamos conversar sobre o assunto e verificar se, para ingressar na carreira de desenvolvedor ou cientista de dados, precisamos saber de estrutura de dados. Acesse o QR code para conferir essa conversa e conhecer melhor o que a disciplina ajuda no dia a dia dessas duas profissões.



## OLHAR CONCEITUAL

Durante esta unidade, estudamos as principais estruturas de dados de Python que são: Listas, Sets, Dicionários e Tuplas. Essas estruturas podem resolver problemas específicos e ser úteis em diversas situações. Você deve escolher a melhor estrutura de acordo com a sua característica e o problema que deve ser resolvido, como vimos cada uma tem sua particularidade. Confira o infográfico para ver de forma estruturada e compilada o conteúdo apresentado.

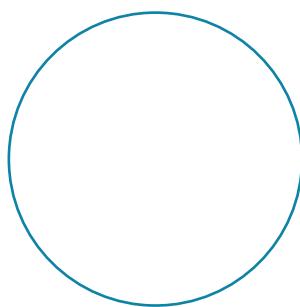


Infográfico com as principais estruturas de dados / Fonte: o autor.

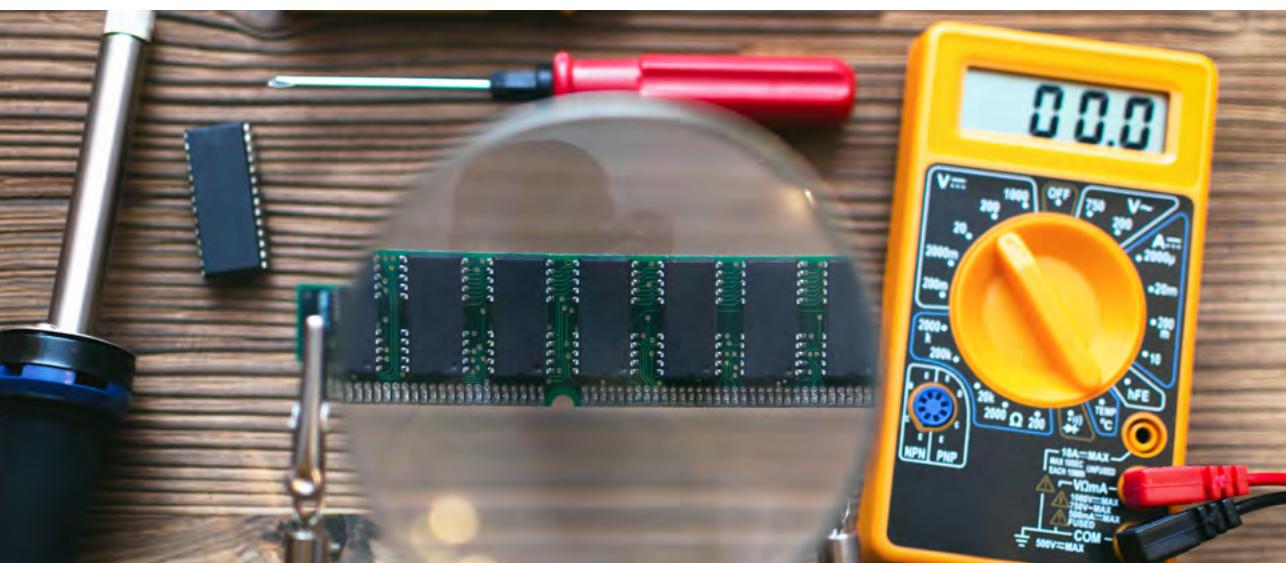
**NOVAS DESCOBERTAS**

Você sabia que a estrutura de dados pode ser aplicada para construir jogos digitais? Veja neste artigo, MARCACINI, R. G.; COUTO, H. A. JOGOS DIGITAIS: Aplicação da Estrutura de Dados com Lista Circular. Veja como a estrutura de lista circular contribuiu para controlar a alocação de espaço de memória e o acesso a estes dados, desta forma, é possível utilizar estas estruturas dinâmicas ao invés de estruturas estáticas devido a performance necessária para jogos digitais.

Acesse o link para baixar o PDF.

**Acesso aos Códigos Editáveis**

De acordo como vimos nesta unidade é importante entender as estruturas homogêneas e heterogêneas e a sua aplicabilidade no dia a dia. Com base nos seus estudos, converse com pelo menos 2 profissionais da área, para averiguar as dificuldades de desenvolver soluções computacionais utilizando as estruturas de dados homogêneas e heterogêneas, compreenda quais os benefícios e dificuldades geradas nesta abordagem.



# AGORA É COM VOCÊ



1. Quando uma estrutura de dados envolve a utilização de mais de um tipo básico de dado, denomina-se:
  - a) Multivariável.
  - b) Recursiva.
  - c) Aleatória.
  - d) Dinâmica.
  - e) Heterogênea.
  
2. Estudamos várias estruturas de dados, assinale a alternativa que não apresenta um tipo básico de estrutura de dados.
  - a) Tuplas.
  - b) Vetor.
  - c) Set.
  - d) Artefatos.
  - e) Matrizes.
  
3. Durante a execução do sistema, os dados são armazenados na memória do computador. As localizações de memória em que esses dados são armazenados se denominam de:
  - a) Banco de dados.
  - b) Algoritmos.
  - c) Estrutura de dados.
  - d) Tipos de dados.
  - e) Estrutura de arquivos.
  
4. Qual o tipo de estrutura de dados que é dividida em linhas e colunas. Assim pode-se armazenar diversos valores dentro dela. Trata-se de:
  - a) Matrizes.
  - b) Vetor.
  - c) Set.
  - d) Artefatos.
  - e) Tuplas.



5. Em estrutura de dados podemos trabalhar com coleção de dados. Qual o tipo de coleção que mantém os dados de forma desordenada e que não possuem elementos em duplicidade:
  - a) Matrizes.
  - b) Vetor.
  - c) Fila.
  - d) Set.
  - e) Array.
6. Assinale a alternativa correta relacionada a Dicionários.
  - a) São estruturas de dados que são usados para mapear chaves e valores. Essa estrutura é utilizada tanto para manipular quanto para armazenar os dados.
  - b) São estruturas de dados que são usados para mapear somente as chaves. Essa estrutura é utilizada tanto para manipular quanto para armazenar os dados.
  - c) São estruturas de dados que são usados para os valores. Essa estrutura é utilizada para armazenar os dados.
  - d) São estruturas de dados que são usados para mapear somente as chaves. Essa estrutura é utilizada para manipular os dados.
  - e) São estruturas de dados que são usados para mapear chaves e valores. Essa estrutura é utilizada somente para manipular os dados.



7. Um desenvolvedor implementou a seguinte estrutura de dados:

```
1 vetorA = [];
2 vetorB: list;
3 vetorA = ['A', 'B', 'C'];
4 vetorB = (vetorA);
5 print(vetorA);
6 print(vetorB);
7 print(vetorA==vetorB);
8 print(vetorA is vetorB);
```

Assinale a alternativa que apresenta o resultado correto:

a) O resultado é:

[A, B, C]

[]

False

False

b) O resultado é:

[A, B, C]

[A, B, C]

True

True

c) O resultado é:

[A, B, C]

[A, B, C]

True

False.

d) O resultado é:

[A, B, C]

[A, B, C]

True

Null

e) O programa apresenta um erro na linha 04, por que são atributos diferentes.

MEU ESPAÇO





# Pilhas e Filas

Esp. Edson Orivaldo Lessa Junior

Esp. Rafael Orivaldo Lessa

## OPORTUNIDADES DE APRENDIZAGEM

Nós estudamos na unidade anterior as estruturas de armazenamento, como classificá-las entre as estruturas homogêneas e heterogêneas, comportamentos diversos na linguagem Python e como podemos melhor utilizá-las. Nesta unidade, vamos nos aprofundar em estruturas mais complexas que são muito utilizadas em diversas soluções de softwares. Vamos falar de pilhas e filas! Por incrível que possa parecer, o conceito é da década de cinquenta e, até hoje, boa parte dos programas que você usa no dia a dia utilizam internamente os tipos abstratos de dados chamados filas e pilhas. Nomenclatura está marcad a pelas características associadas que definem seu comportamento.

Você deve estar se perguntando: “Para que preciso entender de pilhas e filas?”, uma pergunta muito simples e extremamente pertinente. Talvez para explicar melhor, vamos tentar abordar de outro ângulo. Antes de falarmos sobre as ferramentas (pilhas e filas), vamos falar do problema. Se pensarmos no setor de logística é muito importante para o sucesso da empresa, mas, para que funcione de maneira adequada, a gestão do estoque precisa estar alinhada com fornecedores, clientes e setores internos (compras, vendas e marketing). Para isso, é fundamental tomar conhecimento dos principais métodos de estocagem. Visto desta forma, como você poderia utilizar uma estrutura de dados tal que atendesse a demanda em que o primeiro que entra no estoque deve ser o primeiro a sair?

Ou ainda, se a empresa que contratar você não tenha problemas com a validade dos produtos ou estoque, mas uma necessidade de repassar um aumento das mercadorias ao consumidor, evitando prejuízo para a empresa, o que podemos exemplificar como um estoque de eletrônicos no geral. Como você poderia criar uma estrutura de dados para atender à demanda de que o último produto que entra no estoque é o primeiro a sair do mesmo?

No mundo em que vivemos, no qual a informação está cada vez mais presente, é necessário que se faça uma organização da nossa estrutura de dados respeitando algumas regras de acordo com o problema a ser resolvido. Um exemplo a ser resolvido é o comércio eletrônico que vem crescendo tão rápido, que o tamanho do mercado global de compras on-line está previsto para aumentar de acordo com o relatório do Emarketer (2020, on-line)<sup>1</sup>, em 2021, vai atingir 4,89 trilhões de dólares.

Podemos observar uma crescente no uso massivos de dados e transações pela internet, de acordo com a Invesp (2020)<sup>2</sup>. A seguir, podemos observar a movimentação financeira média por comprador no e-commerce:

- EUA - \$1.804;
- Reino Unido - \$1.629;
- Suécia - \$1.446;
- França - \$1.228;
- Alemanha - \$1.064;
- Japão - \$968;
- Espanha - \$849;
- China - \$626;
- Rússia - \$396;
- Brasil - \$350.

De acordo com o Olhar Digital (2020), no Brasil, em 2021 foram geradas 78,5 milhões de compras on-line no primeiro trimestre, agora, imaginem todos esses dados de produtos, clientes e pedidos que devem ser armazenados e tratados com uma boa estrutura de dados. Considerando este contexto, o uso de fila para os pedidos de compra vai ser o mais indicado tendo em vista a sua característica que a saída de uma fila deve obedecer à ordem de chegada, isto é, será atendido o primeiro pedido que chegou, e os novos pedidos que chegam devem ficar no final da fila.

Analizando outra necessidade de conseguir levar notícias atualizadas para o maior número de usuário, que atualmente de acordo com We Are Social e Hootsuite (2021)<sup>3</sup>, estima-se que há 4,66 bilhões de usuários na internet um aumento de 316 milhões (7,3%) relativo ao ano de 2020, este mesmo relatório mostra a existência de 5,22 bilhões de usuários com dispositivos móveis. Se, atualmente, temos uma estimativa de 7,8 bilhões de pessoas no nosso planeta, neste caso, teríamos metade das pessoas do planeta conectadas à internet. No Brasil, temos o seguinte cenário de acesso à informação de acordo com o PoderData (2020) que 41% dos brasileiros usam a internet para ler matérias jornalísticas, como principais meios para se manter informados, para resolver esse tipo de problema utilizamos o conceito de Pilha, porque na batalha por audiência sempre a última notícia é a que sai primeiro nas buscas e mídias sociais para engajar o leitor.

Veja o quanto é importante saber que para cada tipo de problema real, temos uma estrutura de dados que melhor vai se adaptar e resolver a necessidade dos clientes, leitores e usuários do nosso sistema.

Com certeza, você pode imaginar que um sistema em que o primeiro produto que entra é o primeiro que sai é sempre uma boa escolha lógica. Porém esta verdade não é absoluta! Vamos experimentar o seguinte, procure na internet um jogo on-line que se chama Torre de Hanoi. A ideia básica é mover os discos da primeira torre para a última, porém o conceito básico é o primeiro que entra sempre e precisa ser o primeiro a sair. Veja o quanto isso pode ser trabalhoso quanto maior o número de discos.

Você deve ter observado que na estrutura FIFO, nem sempre é a melhor abordagem para performance. Para qualquer escolha de estrutura é necessário ponderar sobre as necessidades que precisamos atender (necessidades do cliente), custo computacional (performance) e limitações.

Utilizando a experiência da Torre de Hanói como exemplo: se tivermos 9 discos, o mínimo de movimentos necessários para passar da primeira torre para

a última é de 511 movimentos. E se aumentarmos somente 1 disco, passando para 10, o mínimo de movimentos passa para 1023 movimentos. Apesar de continuar a entregar o resultado da forma que o cliente espera, o custo computacional fica cada vez mais alto; a cada disco inserido dobra o custo computacional (aproximadamente esforço) e teremos um limite de execução em algum momento.

Comente no seu diário de bordo o seu entendimento e como poderíamos utilizar as diferentes técnicas de ordenação e organização.

## DIÁRIO DE BORDO



Vimos algumas ideias de como o conceito de pilhas e filas são abordadas no mundo em que vivemos. Devemos então aprofundar nos estudos, nos entendimentos dos conceitos, estruturas e técnicas que definem a abordagem relacionada. Vamos entender um pouco mais do que está por trás dos conceitos: o último que entra, é o primeiro que sai e o primeiro que entra é o primeiro que sai.

Figura 1 - FIFO: First in First out

**Descrição da Imagem:** imagem de cubos empilhados com as palavras na primeira lista FIFO e na segunda first in, first out.



Figura 2 - FIFO vs LIFO

**Descrição da Imagem:**  
imagem de uma prancheta vermelha, segurada por duas mãos e nela está uma folha de papel em branco escrito em azul com as palavras LIFO vs FIFO.

**FIFO** (*First-in, First-out*): também é conhecido no Brasil como PEPS (Primeiro a entrar, primeiro a sair), trata-se

de uma estratégia na qual os ativos (no nosso caso dados), que estão armazenados há mais tempo, são os primeiros a serem encontrados.

**LIFO** (*Last-in, First-out*): também é conhecido no Brasil como UEPS (Último a entrar, primeiro a sair), esta é uma estratégia de armazenamento, que trabalha de maneira oposta ao FIFO, ou seja, o último produto a entrar no armazenamento deve ser o primeiro a sair dele.

Figura 3 - Último a entrar, primeiro a sair.

**Descrição da Imagem:**  
uma pilha de livros representando a ideia do LIFO.

Vimos os conceitos básicos por trás das estruturas de pilhas e filas, agora vamos entender como trabalhar estas estruturas.



**Tipos abstratos de dados** - os tipos de dados, que vimos na Unidade 1, são do tipo primitivos, porque são implementados conforme a especificação. Já um tipo abstrato de dado (TAD), especifica os métodos e a características da operação, mas não tem a especificação da implementação, o que faz ele se tornar abstrato. Nesse contexto, abstrato significa “esquecida a forma de implementação”, em outras palavras, um tipo de

dado abstrato especifica o tipo de dado (domínio e operações) sem referência aos detalhes da implementação.

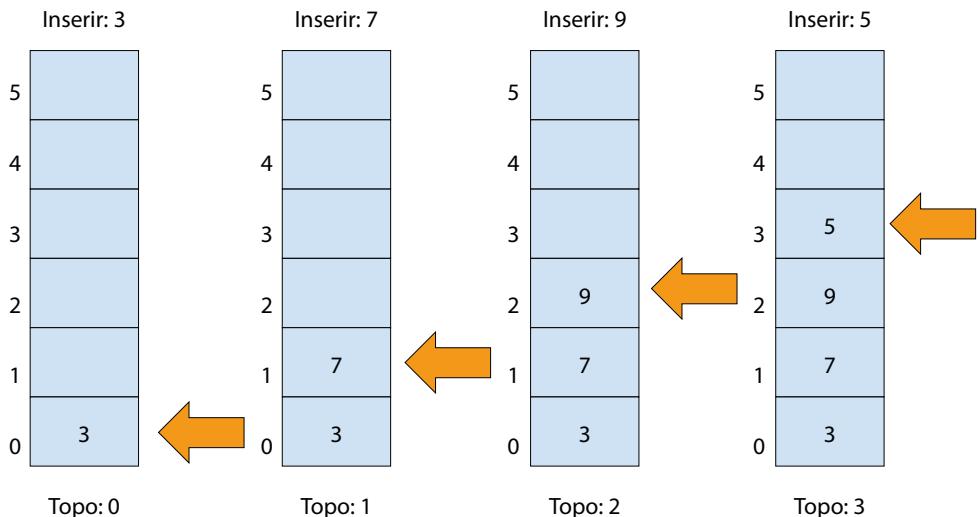
Para que se possa entender esta dinâmica, vamos interpretar o conceito de tipo de dado sob uma perspectiva diferente: o que o programador deseja fazer (realizar a adição de dois números, multiplicar um determinado número por dois etc.). Nesta perspectiva, o programador que necessitar definir uma variável do tipo inteiro, não se interessa com a representação no hardware relacionado à funcionalidade, mas sim com o conceito matemático de inteiro (quais operações um tipo inteiro suporta). O programador pode, em muitas ocasiões, pensar nas estruturas de dados em termos das operações que elas suportam e não da forma como elas são implementadas, portanto sendo chamada de TAD.

Um tipo abstrato de dado estabelece o conceito de tipo de dado separado da sua representação para que possamos reduzir códigos do programa, uma vez que omite os detalhes de implementação.

Assim, a tarefa de especificar um algoritmo se torna mais simples, e você pode descrever as operações que precisa sem ter que pensar na execução. Como o TAD é um desenvolvimento abstrato, temos que escrever um código para que seja possível de ser utilizado em qualquer implementação. As operações em TADs provêm de uma linguagem de alto nível comum para especificar e falar sobre algoritmos.

## Pilhas

A pilha é uma das estruturas ou coleções de dados mais versáteis e simples. Sua ideia fundamental é que todo acesso aos seus elementos seja feito a partir do topo. Assim, quando um elemento é inserido na pilha, ele passa a ser o elemento do topo e só temos acesso a ele. Daí a ideia de empilhamento de dados. Os elementos da pilha só podem ser retirados do último para o primeiro: último que entra e o primeiro que sai (LIFO). Uma pilha pode ser implementada utilizando um vetor, se tivermos a informação do número máximo de elementos que iremos armazenar. Para entendermos bem o conceito vamos ver como fica em uma estrutura.

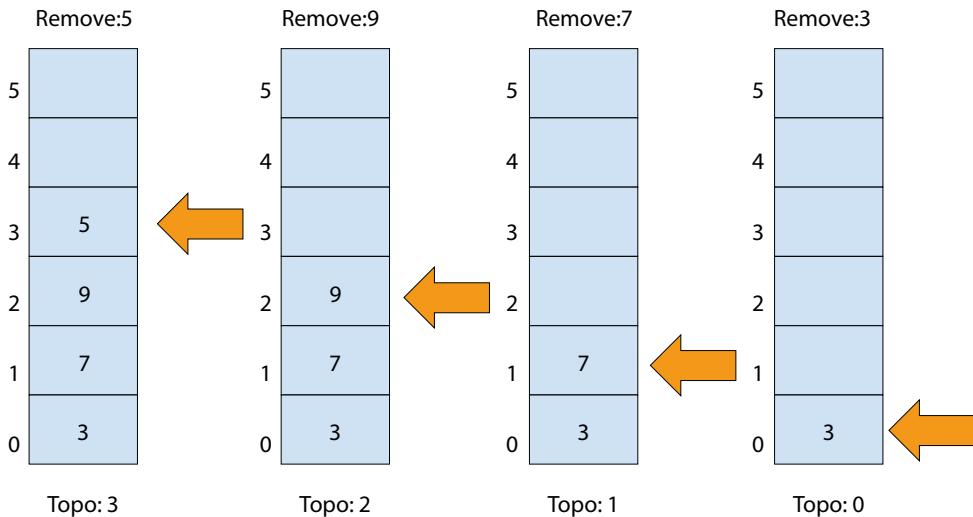


**Figura 4 – Empilhar / Fonte: o Autor.**

**Descrição da Imagem:** com quatro colunas representando quatro momentos de uma pilha de números. A cada momento, um elemento é empilhado e, com isso, sobe o ponteiro “topo” para o último elemento incluído na pilha.

Podemos observar na imagem anterior, que estão sendo inseridos elementos aleatórios na estrutura. Contudo, no armazenamento há uma organização lógica de inclusão, esta ação de inclusão é chamada de **empilhamento**. O primeiro elemento “3” foi empilhado na posição 0 e com isso o ponteiro “Topo” está em 0. Quando empilhamos o segundo elemento “7”, ele fica armazenado na posição 1 e o ponteiro “Topo” muda para a posição 1 também. Veja que o empilhamento ocorre em função que o ponteiro “Topo” muda conforme novos elementos são empilhados.

Vejamos como fica o comportamento, quando precisamos retirar os elementos da pilha.

**Figura 5 - Desempilhar / Fonte: o Autor.**

**Descrição da Imagem:** com quatro colunas representando quatro momentos de uma pilha de números. A cada momento, um elemento é desempenhado e, com isso, desce o ponteiro “topo” para o penúltimo elemento na pilha.

Veja que agora a situação está invertida, ou seja, cada elemento que está sendo removido está na posição em que o ponteiro “Topo” está referenciado. Esta ação é chamada de **desempilhar**. Portanto com os elementos empilhados [5, 9, 7, 3], se necessitarmos acessar o elemento 7 precisaremos desempilhar os elementos [5, 9] antes de alcançarmos o destino. O ponteiro “Topo” movimenta diminuindo o valor de referência de onde se encontra.

Uma analogia muito interessante para entendermos bem a estrutura de pilhas é do amontoado de pratos em restaurante. Cada item na pilha conhece tanto o de cima quanto o abaixo imediatamente. Assim, poderemos imaginar um algoritmo que resolva o cenário de empilhar e desempilhar os pratos.

No restaurante, quando formos organizar o buffet, teremos de distribuir os pratos para que as pessoas possam se servir, e os pratos consumam menos espaços possíveis. Além de permitir que os garçons possam repor os pratos em um único local. Uma solução lógica seria empilhar os pratos no início do buffet e, para isso, precisaremos de algumas informações:

- Onde está o prato anterior;
- Atingimos a quantidade limite da pilha;
- A pilha está vazia.

A segunda informação, vamos deixar no momento de lado, pois se pensarmos no computacional seria o limite de memória, e no nosso exemplo seria a altura adequada para uma pessoa pegar um prato. A preocupação principal que devemos ter está na primeira informação, onde está o prato anterior, temos 2 cenários possíveis:

- Não existe pratos na pilha;
  - Já existem pratos na pilha.

Vamos usar a seguinte implementação de uma pilha em Python:

```

1 class Pilha(object):
2     # Inicializa a pilha
3     def __init__(self):
4         self.elementos = []
5
6     # Método para fazer o empilhamento dos elementos
7     def empilha(self, elemento):
8         self.elementos.append(elemento)
9
10    # Método para fazer o desempilhamento dos elementos
11    def desempilha(self):
12        return self.elementos.pop()
13
14    # Método para verificar se a pilha está vazia
15    def vazio(self):
16        return len(self.elementos) == 0
17
18    # Método sobreescrito para converter o objeto em caracter
19    def __str__(self):
20        retorno: str = "\n{um} {dois} {tres}\n"
21        .format(um="Índice Lógico".ljust(15), dois="Índice Real".ljust(15),
22            tres="Elemento".ljust(20))
23
24        contador = 0
25        # Exibindo a pilha de forma inversa
26        for elemento in self.elementos[::-1]:
27            indice = self.elementos.index(elemento)
28            retorno += "{um} {dois} {tres}"\
29                .format(um=str(contador).ljust(15), dois=str(indice).ljust(15), tres=str(
30                    elemento).ljust(20))
31            retorno += "\n"
32            contador += 1
33
34    return retorno

```

Veja que neste exemplo, estamos utilizando quatro métodos na classe, o primeiro (self) que inicializa a estrutura de armazenamento (pilha). O último verifica se a estrutura (pilha) está vazia. Lembre-se que o comando len(self.elementos)==0 é uma expressão lógica que retorna true ou false. Os dois métodos restantes, resolvem o problema essencial da pilha: empilhar e desempilhar.

```
1 import random
2
3 from Pilha import Pilha
4
5 class inicializaPilha:
6     def __init__(self):
7         self.pilha = Pilha()
8         self.menu()
9
10    def imprimirPilha(self):
11        print("\n Imprimindo conteúdo da pilha\n")
12        print(self.pilha)
13
14    def removerElementos(self):
15        self.pilha.desempilha()
16        print("Elemento removido")
17
18    def incluirElementos(self):
19        valor = input("Informe o valor a ser incluído\n")
20        self.pilha.empilha(valor)
21        print("Elemento incluído\n\n")
22
23    def incluirDezMilElementos(self):
24        for i in range(10):
25            self.pilha.empilha(random.randint(1, 10000))
26
27    def verificarPilhaEstaVazia(self):
28        if self.pilha.vazio():
29            print("Estrutura está vazia ")
30        else:
31            print("Estrutura não está vazia ")
32
33    def menu(self):
34        while True:
```

```

35     opcao = input("1 - Incluir elemento na pilha\n"
36             "2 - Retirar elemento da pilha\n"
37             "3 - Exibir os elementos da pilha\n"
38             "4 - Incluir vários elementos da pilha\n"
39             "0 - Deseja sair\n")
40     if opcao == "1":
41         self.incluirElementos()
42     elif opcao == "2":
43         self.removerElementos()
44     elif opcao == "3":
45         self.imprimirPilha()
46     elif opcao == "4":
47         self.incluirDezMilElementos()
48     elif (opcao == "0"):
49         break
50     else:
51         print("Opação inválida\n\n")
52
53
54 inicializaPilha = inicializaPilha()
55

```

Neste trecho, realizamos o teste da pilha, no qual utilizamos os métodos para empilhar e desempilhar elementos e exibir o seu conteúdo. Note que, a cada execução do método pop, o item mais novo será removido da pilha. Em algum momento, sua pilha poderá estar vazia e esse método levantará a exceção “*IndexError: pop from empty list*”, ou “Erro de índice: pop de uma lista vazia”. Este erro irá ocorrer, pois não há mais valores a serem removidos e recuperados na pilha. Por isso, é importante verificar se a pilha está vazia antes de realizar uma operação.

Vejamos quando executamos a pilha, incluindo os elementos 10, 21 e 15. A pilha fica como:

#### Imprimindo conteúdo da pilha

Índice Lógico	Índice Real	Elemento
0	2	15
1	1	21
2	0	10

Se removermos um elemento, o resultado seria:

### Imprimindo conteúdo da pilha

Índice Lógico	Índice Real	Elemento
0	1	21
1	0	10

Perceba que no resultado, está imprimindo duas colunas “Índice Lógico” e “Índice Real”, esta opção de exibição é para que você perceba que a funcionalidade padrão de uma pilha no Python é que o último elemento fica no final, o que para efeitos de raciocínio comparativo de uma pilha é invertido. Ou seja, quando pensamos em pilha, o último elemento que colocamos fica no topo e não no final dela. Claro que poderíamos ter escolhido implementar a pilha usando uma lista em que o topo estaria no início e não no final. Neste caso, os métodos *pop()* e *append()* não seriam apropriados pois deveríamos indexar a posição 0 (o primeiro item da lista) explicitamente usando *pop(0)* e *insert*. Veja como ficaria o código com a pilha zero.

```

1 class Pilha(object):
2     # Inicializa a pilha
3     def __init__(self):
4         self.elementos = []
5
6     # Método para fazer o empilhamento dos elementos
7     def empilha(self, elemento):
8         self.elementos.insert(0, elemento)
9
10    # Método para fazer o desempilhamento dos elementos
11    def desempilha(self):
12        return self.elementos.pop(0)
13
14    # Método para verificar se a pilha está vazia
15    def vazio(self):
16        return len(self.elementos) == 0
17
18    # Retorna o topo da fila
19    def elementoTop(self):
20        return self.elementos[0]
21

```

```

22  # Método sobreescrito para converter o objeto em caracter
23  def __str__(self):
24      retorno: str = "\n{um} {dois} {tres}\n"
25      .format(um="Índice Lógico".ljust(15), dois="Índice Real".ljust(15),
26      tres="Elemento".ljust(20))
27
28      contador = 0
29      # Não é necessário exibir a pilha de forma invertida
30      for elemento in self.elementos:
31          indice = self.elementos.index(elemento)
32          retorno += "{um} {dois} {tres}\n"
33          .format(um=str(contador).ljust(15), dois=str(indice).ljust(15), tres=str(
34          (elemento).ljust(20)))
35          retorno += "\n"
36          contador += 1
37
38      return retorno

```

Alterando na inclusão da pilha, não é mais necessário realizar a exibição inversa para entendimento da pilha. Agora incluindo os mesmos elementos 10, 21 e 15, teríamos:

### Imprimindo conteúdo da pilha

Índice Lógico	Índice Real	Elemento
0	0	15
1	1	21
2	2	10

Veja que agora as colunas “Índice Lógico” e “Índice Real” estão com o mesmo valor, pois a inversão de inclusão foi feita no método *insert* e *pop*, que garante uma estrutura da mesma lógica que entendemos por pilha. Vale lembrar que essa característica é apenas para efeitos visuais, ou seja, apenas para que você veja como o Python trabalha. Quando utilizar em ambiente profissional, não é necessário se preocupar em que lado está o topo da pilha (início ou final), o Python irá tratar a pilha da maneira correta.

As pilhas trabalham apenas na extremidade da lista, por isso têm comportamento LIFO. O último elemento adicionado é considerado o elemento do topo da pilha; o primeiro elemento adicionado é o da base da pilha. Podemos passar por todos os elementos da pilha utilizando for ou while.



Figura 6 - Pilha de notebooks.

**Descrição da Imagem:** pilha de notebooks, representando o sistema de pilhas, nas cores prata e preto.



Figura 7 - Título Fila

**Descrição da Imagem:** imagem em vetor de uma fila de pessoas, em zig zag.

Posso arriscar em afirmar que você sabe o que é uma fila. Talvez não uma fila em estrutura de dados, mas fila comum onde você espera a sua vez para algum atendimento. Vamos refletir, se você está em uma fila no caixa de uma loja para realizar o pagamento, é necessário que todos sejam atendidos ou saiam da fila para que você possa realizar o pagamento. Por isso, podemos dizer que a ideia da fila é que o último a entrar na fila seja o último a sair. Uma fila em estrutura de dados possui a mesma ideia básica: o primeiro que entra é o primeiro a sair, ou como vimos anteriormente uma estrutura FIFO. Esta estrutura de dados abstrata é a fila que se comporta exatamente como uma fila na vida real. Elementos são adicionados no final e removidos da frente. (PERKOVIC, 2016).

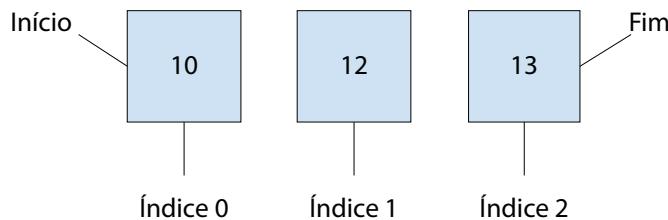
Entretanto, remover elementos do início pode ter problemas de performance do sistema. Para realizar essa operação, teríamos que usar as listas para criar as

filas, só que para que isso ocorra é movido todos os índices dos elementos posteriores ao removido, ocasionando um problema de desempenho que depende do tamanho da lista. Por este motivo, mesmo que exista uma forma de usarmos listas como se fossem filas em Python, não é recomendado por causa da forma como Python implementa listas. Felizmente, existe uma classe que implementa as operações de filas em Python que é a *deque*.

Segundo a documentação oficial do Python, a *deque* suporta incrementos e decrementos seguros para *thread* e com eficiência de memória de ambos os lados do *deque* com aproximadamente o mesmo desempenho satisfatório. Em resumo, *deque* é mais rápido para inserir e remover elementos da frente da fila (no índice 0).

Imagine a seguinte situação, você queria incluir em uma fila, senhas geradas aleatoriamente. Vamos supor incluiremos um elemento, portanto graficamente ficaria assim:

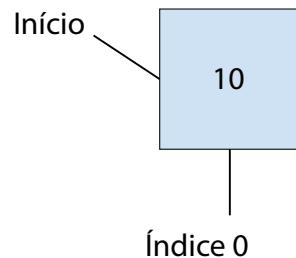
Vamos incluir mais 3 elementos



**Figura 9 - Inclusão de três elementos / Fonte: o Autor.**

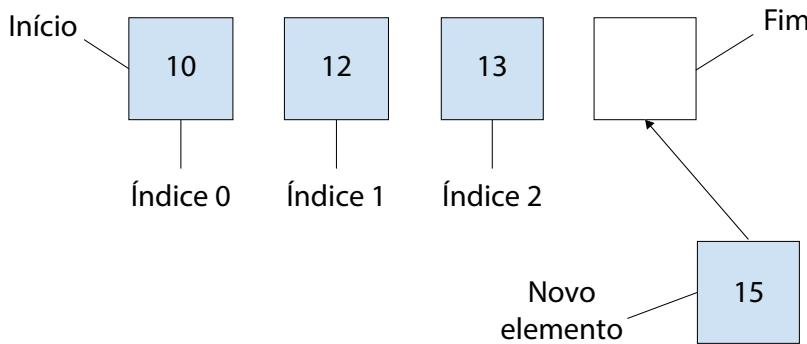
**Descrição da Imagem:** representação de uma fila com três elementos com os valores 10, 12 e 13.

Sempre que iremos incluir um elemento, perceba que é feito ao fim da fila.



**Figura 8 - Primeiro elemento / Fonte: o Autor.**

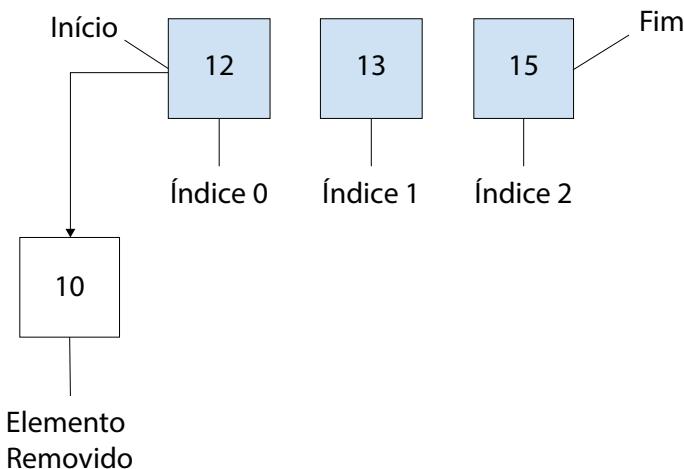
**Descrição da Imagem:** primeiro elemento na fila com valor 10.



**Figura 10 - Inclusão do novo elemento no fim da fila / Fonte: o Autor.**

**Descrição da Imagem:** representa a inclusão de um quarto elemento, no qual destaca onde será incluído (fim da fila).

Naturalmente, cada elemento inserido deverá ser incluído ao fim da estrutura. Diferentemente, quando for excluído.



**Figura 11 - Remove o primeiro elemento / Fonte: o Autor.**

**Descrição da Imagem:** demonstra a remoção do elemento da fila. O primeiro em conformidade com a instrução FIFO.

Além disso, perceba que agora, além da fila não ter mais o primeiro elemento (10), os índices foram reorganizados, 12 passou a ter o índice 0, 13, índice 1 e 15, índice 2.

Seguindo esta ideia, poderíamos ir contra a recomendação do Python e criar um código que faça uma fila utilizando lista, teríamos algo assim.

```
1 class FilacomLista(object):
2     # Inicializa os atributos
3     def __init__(self):
4         self.elementos = []
5
6     # Inclui os elementos na fila
7     def incluirNaFila(self, elemento):
8         posicao = self.elementos.__len__()
9         self.elementos.insert(posicao, elemento)
10
11    # Retira o elemento da fila
12    def retirarDaFila(self):
13        listaAuxiliar = self.elementos.copy()
14        self.elementos = []
15        for i in range(listaAuxiliar.__len__()):
16            if i != 0:
17                self.elementos.append(listaAuxiliar[i])
18
19    # Reescrevendo o método para recuperar a quantidade de elementos
20    def __len__(self):
21        return len(self.elementos)
22
23    # Reescrevendo o método para retorna o objeto como caracter
24    def __str__(self):
25        retorno: str = "\n"
26        for elemento in self.elementos:
27            indice = self.elementos.index(elemento)
28            retorno += str(indice) + " - " + str(elemento)
29            retorno += "\n"
30
31        return retorno
32
33
34 filaComLista = FilacomLista()
35 filaComLista.incluirNaFila(10)
36 filaComLista.incluirNaFila(23)
37 filaComLista.incluirNaFila(143)
38 print(filaComLista.__str__())
39 filaComLista.retirarDaFila()
40 print(filaComLista.__str__())
41
```

Veja que ao criarmos o código que possibilite uma lista ter o comportamento de uma fila, precisamos controlar a entrada de cada elemento e a sua remoção manualmente. Porém a remoção do elemento conforme está no método da linha 12 é bem mais complexa. Perceba que é necessário copiar todos os elementos para uma

lista auxiliar e para que possamos remover o elemento do início. Naturalmente, esta ação para poucos elementos é imperceptível, mas ao incluirmos uma massa considerável de elementos a performance naturalmente será ruim. Por este motivo, a documentação do Python NÃO recomenda o uso de filas por meio de listas.

Vejamos um código em Python utilizando a classe *deque*, que é recomendado pelo Python para resolver o problema das listas.

```
1 from collections import deque
2 from typing import Deque
3
4
5 class Fila(object):
6     # Inicializa os atributos
7     def __init__(self):
8         # Deque define o tipo de coção da fila, está notado que irá receber
9         # apenas elemento do tipo int
10        self.elementos: Deque[int]
11        # Inicializa o objeto da coleção de fila
12        self.elementos = deque()
13
14    def incluirNaFila(self, elemento):
15        # Inclui os elementos na fila
16        self.elementos.append(elemento)
17
18    # Método criado para facilitar a inclusão de vários elementos
19    def incluirMuitosNaFila(self, variosElementos: list):
20        for elemento in variosElementos:
21            self.elementos.append(elemento)
22
23    def retirarDaFila(self):
24        # Retira um elemento da fila
25        return self.elementos.popleft()
26
27    # Reescrevendo o método para recuperar a quantidade de elementos
28    def __len__(self):
29        return len(self.elementos)
30
31    # Reescrevendo o método para retorna o objeto como caracter
32    def __str__(self):
33        retorno: str = "\n"
34        for elemento in self.elementos:
35            indice = self.elementos.index(elemento)
36            retorno += str(indice) + " - " + str(elemento)
37            retorno += "\n"
38
39        return retorno
```

Veja que os métodos definidos são similares à lista, porém a otimização realizada pela classe faz com que não precisemos implementar a troca de índices. Vamos analisar as principais linhas do programa:

- linha 15: o método *append* é responsável em incluir o elemento na fila.
- linha 24: o método *popleft* remove um elemento à esquerda da fila.

Veja que com dois métodos, resolvemos o problema da fila, bem mais simples que utilizando lista, você não acha?

Vale ressaltar que a classe *deque*, resolve a inclusão de elementos nas extremidades e a forma que usaremos define a estrutura de dado que será utilizada. Antes de verificar o funcionamento, vejamos os métodos que a classe *deque* possui:

Os objetos *Deque* suportam os seguintes métodos:

- **append(x)**: adicione x ao lado direito do *deque*.
- **appendleft(x)**: adicione x ao lado esquerdo do *deque*.
- **clear()**: remove todos os elementos do *deque*, deixando-o com comprimento 0.
- **copy()**: crie uma cópia superficial do *deque*.
- **count(x)**: conte o número de elementos *deque* igual a x .
- **extend(iterable)**: estenda o lado direito do *deque* anexando elementos do argumento iterável.
- **extendleft(iterable)**: estenda o lado esquerdo do *deque* anexando elementos de iterável. Observe que a série de acréscimos à esquerda resulta na reversão da ordem dos elementos no argumento iterável.
- **index(x[, start[, stop]])**: retorne a posição de x no *deque* (no início ou depois do início do índice e antes da parada do índice). Retorna a primeira correspondência ou aumenta *ValueError* se não for encontrado.
- **insert(i, x )**: insere x no *deque* na posição i .Se a inserção fazer com que um *deque* limitado cresça além de maxlen , um *IndexError* é gerado.
- **pop()**: remova e devolva um elemento do lado direito do *deque*. Se nenhum elemento estiver presente, gera um *IndexError*.
- **popleft()**: remova e devolva um elemento do lado esquerdo do *deque*. Se nenhum elemento estiver presente, gera um *IndexError*.
- **remove(value)**: remova a primeira ocorrência de valor. Se não for encontrado, aumenta a *ValueError*.
- **reverse()**: inverte os elementos do *deque* no local e depois retorne *None*.

- **rotate(*n*=1)**: gire as etapas *deque n* para a direita. Se *n* for negativo, gire para a esquerda. Quando o *deque* não está vazio, girar um passo para a direita é equivalente a *d.appendleft(d.pop())* e girar um passo para a esquerda é equivalente a *d.append(d.popleft())*.
- **maxlen**: (atributo somente leitura) tamanho máximo de um *deque* ou *None* se ilimitado.

Agora, você já percebeu como a classe *deque* é construída e quais recursos possui que podemos usar de forma a solucionar os problemas. Vejamos como a fila é resolvida no exemplo e para inicializar utilizamos uma classe que inicialize e organize as informações de entrada.

```

1 from fila import Fila
2
3
4 class InicializaFila:
5
6     # Inicializa o atributo fila e executa o método de menu
7     def __init__(self):
8         self.fila = Fila()
9         self.menu()
10
11    # Controla o menu de opções
12    def menu(self):
13        while True:
14            opcao = input("1 - Incluir elemento na fila\n"
15                         "2 - Retirar elemento da fila\n"
16                         "3 - Exibir os elementos da fila\n"
17                         "4 - Incluir vários elementos da fila\n"
18                         "0 - Deseja sair\n")
19            if opcao == "1":
20                self.incluir()
21            elif opcao == "2":
22                self.remover()
23            elif opcao == "3":
24                self.imprimir()
25            elif opcao == "4":
26                self.incluirVarios()
27            elif (opcao == "0"):
28                break
29            else:
30                print("Opação inválida\n\n")
31
32    # Inclui um único elemento

```

```

33 def incluir(self):
34     valor = int(input("Informe o valor inteiro a ser incluído\n"))
35     self.fila.incluirNaFila(valor)
36     print("Elemento incluído\n\n")
37
38 # Remove um único elemento
39 def remover(self):
40     self.fila.retirarDaFila()
41     print("Elemento removido")
42
43 # Imprime todos os elementos
44 def imprimir(self):
45     if self.fila.__len__() > 0:
46         print("Os elementos atuais na fila são:\n", self.fila.__str__())
47     else:
48         print("Não existem elementos na fila\n")
49
50 # Inclui vários elementos
51 def incluirVarios(self):
52     quantidade = int(input("Informe a quantidade de elementos a serem
53 incluídos\n"))
54     valores: list = []
55     for i in range(quantidade):
56         valor = int(input(f"Digite o valor {i + 1}\n"))
57         valores.append(valor)
58         self.fila.incluirMuitosNaFila(valores)
59
60 InicializaFila()
61

```

Neste exemplo então é possível você verificar o funcionamento das filas de uma forma inicial, que permite você incluir e remover elementos da fila. Supondo que incluirmos os elementos 12, 15, 16, 18, 19, 20 ,23; teríamos a fila na seguinte ordem:

#### **Os elementos atuais na fila são:**

```

0 - 12
1 - 15
2 - 16
3 - 18
4 - 19
5 - 20
6 - 23

```

Vejamos como fica excluindo um elemento

**Os elementos atuais na fila são:**

- 0 - 15
- 1 - 16
- 2 - 18
- 3 - 19
- 4 - 20
- 5 - 23

O primeiro elemento que entrou foi retirado da estrutura. Portanto o comportamento FIFO é mantido o que garante uma solução performática.

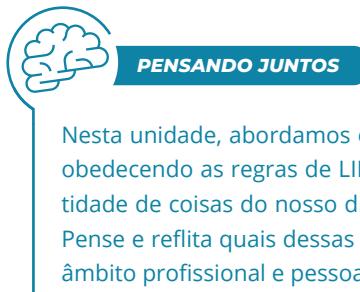


**Figura 12 - Título Pilhas**

**Descrição da Imagem:** Pilhas de vários tamanhos e formatos, criando a analogia para estrutura de pilhas.

Podemos então resumir estas estruturas da seguinte forma:

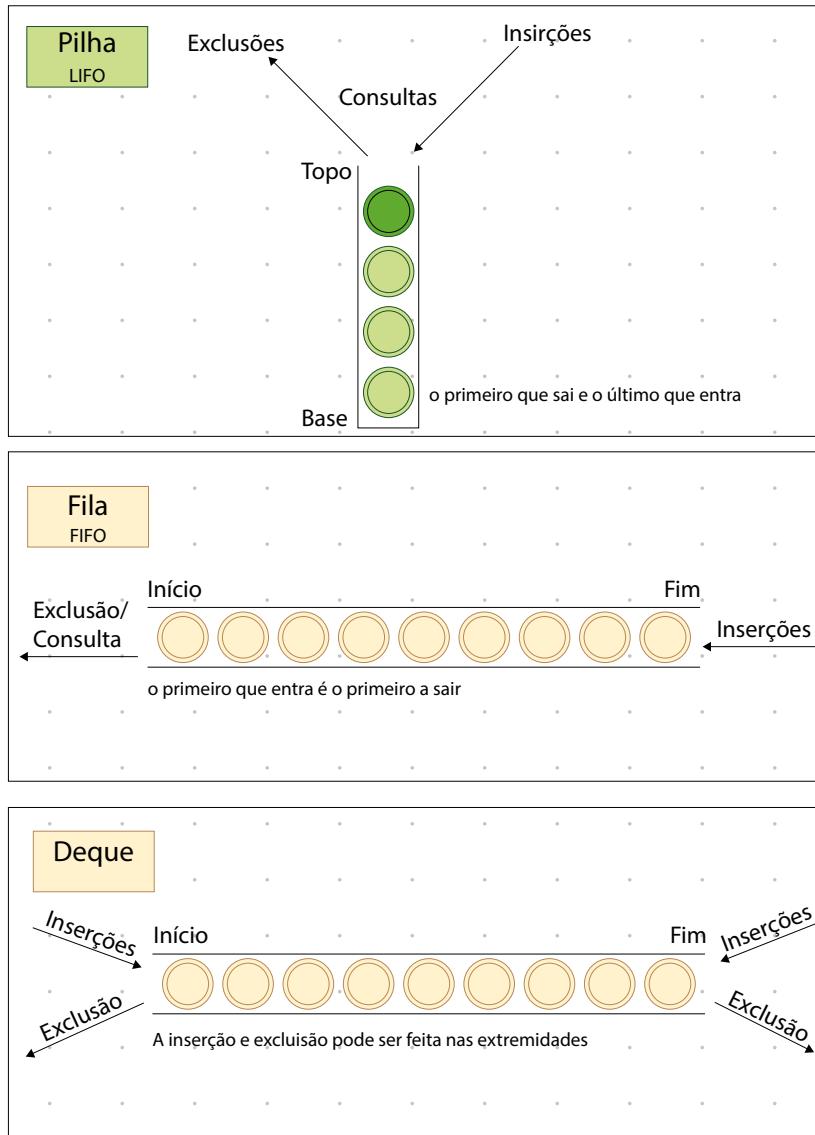
- **Pilha:** é uma estrutura de dados linear cujas informações podem ser acessadas somente por uma de duas extremidades, seja para armazenar ou recuperar informações. Podemos exemplificar como uma pilha de pratos de um restaurante, no qual o último prato colocado na pilha é o primeiro a ser retirado. Assim, a estrutura de dados de Pilha utiliza o conceito “LIFO (do inglês, *last in/first out*)”.
- **Fila:** é uma estrutura de dados linear cujas informações podem ser acessadas por ambas as extremidades, no entanto, uma para armazenar novos elementos e a outra para remover as informações. Podemos exemplificar como uma fila de supermercado, no qual a primeira pessoa que entra na fila é a primeira pessoa a ser atendida. Assim, a estrutura de dados de Fila utiliza o conceito “FIFO (do inglês, *first in/first out*)”.





## OLHAR CONCEITUAL

Durante esta unidade, estudamos pilhas, filas e deque em estruturas de dados de Python. Essas estruturas podem resolver problemas específicos e ser úteis em diversas situações. Você deve escolher a melhor estrutura de acordo com a sua característica e o problema que deve ser resolvido, como vimos cada uma tem sua particularidade.



Título: Infográfico com pilhas e filas / Fonte: os Autores.

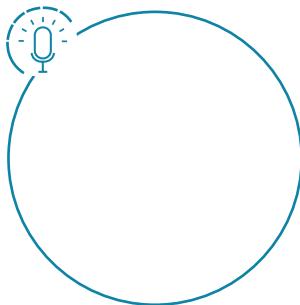


## EXPLORANDO IDEIAS

Filas e pilhas são estruturas de dados usualmente implementadas por meio de listas, restringindo a manipulação dos elementos da lista. Como estudamos, a fila utiliza a política FIFO -- first in, first out -- para manipulação dos dados, resumindo no momento de remover um dado "o mais antigo é o primeiro a sair", obedecendo a ordem de inserção.

Já a estrutura de pilha utiliza a política LIFO -- last in, first out. Resumindo, no momento de remover um dado "o mais novo é o primeiro sair", fazendo a ordem inversa da inserção.

A fila e a pilha são estruturas importantes e são muito utilizadas no desenvolvimento de soluções computacionais, porque resolvem problemas reais com um bom nível de performance e facilidade de programação.



## Você sabe escolher qual a melhor estrutura de dados para usar para seus problemas, pilha ou fila?

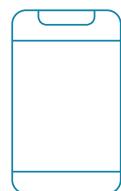
As estruturas de dados lineares (fila, pilha) são muito utilizadas para resolver problemas computacionais. Cada uma dessas estruturas pode ser implementada com diferentes características e atender aos diferentes tipos de problemas. A forma como as operações de acesso, inserção e exclusão da estrutura é executada é que vai determinar o seu tipo. É exatamente nesse ponto que nosso jogo começa. Neste Podcast, vamos conversar sobre cada um dos tipos de estrutura e dados e quais problemas do nosso dia a dia elas podem resolver, acesse o QR code para conferir essa conversa.



## NOVAS DESCOBERTAS

Você conhece o jogo Genius? Você pode jogar ele de forma on-line. Considerando que as luzes acendem em uma ordem e você tem que repetir a mesma ordem, conseguimos utilizar o conhecimento FIFO, a primeira cor que aparece é a primeira que tem que ser repetida e assim por diante. Jogue e veja que funciona exatamente igual a uma fila

Você pode jogar ele de forma online acesse o QR.



Utilizar uma estrutura de pilhas e filas é indispensável no dia a dia do desenvolvimento. Estas estruturas são utilizadas na mineração de dados, consultas da base de dados proporcionando escala e performance adequada. Realize uma pesquisa e encontre dois produtos que você utiliza no dia a dia e verifique se ele atende um conceito de Fila e Pilha. Faça um questionário para verificar quais das estruturas é mais utilizada, tente conseguir 10 respostas.

# AGORA É COM VOCÊ



1. Qual estrutura de dados segue a regra "o último a entrar é o primeiro a sair"?
  - a) pilha.
  - b) fila.
  - c) lista encadeada.
  - d) lista duplamente encadeada.
  - e) matriz.
2. Observe, a seguir, a estrutura de dados, em forma de tabela.

1	2	3	4	5

- Verifique as operações realizadas e responda qual a estrutura utilizada:
- Inserir 10,20,30,40

10	20	30	40	
1	2	3	4	5

Retirar 40

10	20	30		
1	2	3	4	5

Inserir 50 e 60

10	20	30	50	60
1	2	3	4	5

Retirar 30, 50 e 60

10	20			
1	2	3	4	5

# AGORA É COM VOCÊ



Inserir 70

10	20	70		
1	2	3	4	5

Pode-se deduzir, pelas operações realizadas, que tal estrutura é uma:

- a) lista indexada.
- b) árvore.
- c) fila.
- d) fila duplamente encadeada.
- e) pilha.

3. Observe, seguir, a estrutura de dados, em forma de tabela.

1	2	3	4	5

Verifique as operações realizadas e responda qual a estrutura utilizada:

Inserir 10,20,30,40

10	20	30	40	
1	2	3	4	5

Retirar 10

10	20	30	40	
1	2	3	4	5

Inserir 50 e 60.

20	30	40	50	60
1	2	3	4	5

# AGORA É COM VOCÊ



- Retirar 20, 30 e 40

50	60			
1	2	3	4	5

Inserir 70

50	60	70		
1	2	3	4	5

Pode-se deduzir, pelas operações realizadas, que tal estrutura é uma:

- a) lista indexada.
- b) árvore.
- c) fila.
- d) fila duplamente encadeada.
- e) pilha.

4. Verifique a estrutura de fila, a seguir, e informe o time que ficou em terceiro lugar.

Elemento	Anterior	Valor	Próximo
1	4	Flamengo	2
2	1	Figueirense	3
3	2	São Paulo	5
4	-	Avaí	1
5	3	Fluminense	

- a) Flamengo.
- b) Figueirense.
- c) São Paulo.
- d) Avaí.
- e) Fluminense.

# AGORA É COM VOCÊ



5. A estrutura de dados, que faz a inserção de um nó no meio da estrutura de:

- a) lista.
- b) árvore.
- c) fila.
- d) fila duplamente encadeada.
- e) pilha.

6. Analise o código, a seguir, e responda.

```
1  from typing import List
2
3  livros: List[str] = [] # {1}
4
5  livros.append('Livro 1')
6  livros.append('Livro 2')
7  livros.append('Livro 3')
8  livros.append('Livro 4')
9  livros.pop()
10 livros.append('Livro 5')
11 livros.append('Livro 6')
12 livros.pop()
13 livros.append('Livro 7')
14
15 for livro in livros[::-1]:
16     print(livro)
```

Qual a ordem que será exibido os valores da pilha:

- a) Livro 7, Livro 5, Livro 3, Livro 2, Livro 1.
- b) Livro 1, Livro 2, Livro 3, Livro 5, Livro 7.
- c) Livro 1, Livro 2, Livro 3, Livro 7; Livro 5.
- d) Livro 7, Livro 3, Livro 5, Livro 2, Livro 1.
- e) Livro 7, Livro 3, Livro 5, Livro 1, Livro 2.

MEU ESPAÇO



# 3

# Listas dinâmicas

Esp. Edson Orivaldo Lessa Junior

Esp. Rafael Orivaldo Lessa

## OPORTUNIDADES DE APRENDIZAGEM

Nós estudamos na unidade anterior, as estruturas de filas, pilhas e deques e seus comportamentos na linguagem Python e como podemos melhor utilizá-las. Nesta unidade, vamos nos aprofundar em estruturas mais dinâmicas para atender casos que necessitam de uma melhor performance de inserção e exclusão de dados. Essas estruturas são chamadas de listas ligadas e duplamente ligadas, possuem este nome porque são compostas por nós que estão ligados por meio de ponteiros, eles apontam para o endereço da memória do próximo nó, assim os dados não precisam estar contínuos na memória e o desenvolvedor consegue ter uma liberdade de inserir o dado na posição desejada.

No processo de desenvolvimento, utilizamos estruturas que trazem uma facilidade e praticidade de desenvolvimento, porém esse tipo de dado possui diversas limitações no seu uso, devido as suas regras de entrada e saída dos dados como a **FIFO** e a **LIFO**. Você deve estar pensando: “então, como faço para resolver problemas complexos sem ter esse tipo de limitação?”

Antes de falar dessas listas, vamos pensar que temos que resolver alguns problemas de uma FinTech, para abertura de conta corrente, no qual existem várias variáveis para análise financeira e não há uma regra definida de FIFO ou LIFO, porque depende do processamento de cada análise, neste caso, precisamos de mais robustez e versatilidade na manipulação dessa estrutura de dados. Para resolver esse tipo de problema, utilizamos as listas dinâmicas assim conseguimos realizar todo o processamento sem a necessidade de obedecer uma ordem.

Como temos visto, o grande volume de dados produzidos, atualmente, e o uso massivo dos dados para tomadas de decisão vêm trazendo uma rápida evolução nas tecnologias de Big Data e na forma como manipulamos esses dados. O Big Data traz vantagens reais, quando associado ao processo de tomada de decisão (GANDOMI; HAIDER, 2015). Atualmente, são várias as empresas que têm utilizado e demonstrado interesse na utilização de ferramentas de Big Data. Esse tipo de tecnologia traz oportunidades únicas de personalizar os produtos e serviços fornecidos pelas organizações, e as empresas que não utilizam tecnologias de Big Data não serão competitivas.

O **Big Data** trabalha com dois tipos de dados estruturados e não estruturados, essa tecnologia traz vantagens como: eficiência operacional, atendimento personalizado, capacidade de desenvolvimento de novos produtos devido às variações do mercado, identificar potenciais *prospects* etc. Com todo esse processo de cruzar os dados para chegar a uma conclusão para a tomada de decisão, podemos dizer que conseguimos estruturar um Big Data Analytics.

Porém esse volume de dados e a utilização de tecnologias de **Big Data Analytics** vêm com desafios de conceber sistemas que suportem com eficácia esse grande volume de dados e com uma capacidade de gerar valor para os negócios. Para algumas empresas, este tema é tão importante que elas já investiram US \$187 bilhões em Big Data Analytics, em 2019, de acordo com a IDC (*International Data Corporation*).

Agora, imagine esse grande volume de informações e trabalhar em uma estrutura de dados que não são dinâmicas e com pré-alocagem de memória: teríamos

um desafio de armazenar e processar de forma eficiente exabytes de dados, o sistema teria que dispor de um grande volume de memória e um longo tempo de processamento dos dados. Lembre-se que para cada problema, temos uma estrutura de dados mais adequada.

Você deve estar pensando: como um sistema pode aumentar o número de informações sempre que um novo dado é adicionado? A solução para esses problemas é utilizar estruturas de dados que cresçam à medida que precisarmos armazenar novos elementos e diminuam à medida que precisarmos retirar elementos armazenados anteriormente. Vamos experimentar! Imagine que você tem uma fila de atendimento que obedece a ordem de chegada (FIFO), porém você também tem uma fila de atendimento prioritário que a ordem é a idade, ou seja, 1º critério é a idade do indivíduo que está na fila, em anos; 2º é a ordem de chegada. Se houver mais de um indivíduo com a mesma idade, será respeitado o segundo critério. Faça um desenho de como essa fila estaria ligada.

Você deve ter observado que as estruturas de dados dinâmicas trazem benefícios do seu uso, que são a inserção ou remoção de dados na lista sem implicar na mudança de lugar de outros dados e ainda que não é necessário definir, ao criar uma lista, o número máximo de dados que será armazenado. Em outras palavras, é possível alocar memória “dinamicamente”, apenas para o número de nós que serão utilizados. Porém esse tipo de estrutura também traz algumas desvantagens, que é o uso incorreto da ligação de nós, caso a ligação seja feita errada, toda a lista pode ser perdida. Comente no seu diário outras vantagens e desvantagens do uso de listas ligadas.

## DIÁRIO DE BORDO



Figura 1 - Dados binários

**Descrição da Imagem:** imagem com números binários eletrônicos listados na vertical em três dimensões.

**Lista Ligada:** existem estruturas de dados que são mais especializadas que os vetores, algumas delas já vimos como as pilhas e filas, no qual possuem conceitos de regras para entrada e saída como as **FIFO** e a **LIFO**. Ainda existem outros tipos de estruturas, que possuem características próprias que proporcionam uma grande flexibilidade, uma destas estruturas é a **lista** que pode ser trabalhada para assumir características diferentes.

Podemos definir as listas como:

“Relação ordenada de nomes de pessoas ou de coisas; cadastramento, listagem, relação”. (MICHAELIS, 2021).

Porém existem diversos tipos de listas, nos quais são classificados de acordo com seu ordenamento de dados e seus elementos. Claro que você deve estar pensando em quantas formas de ordenação poderiam existir para que possamos classificar as listas. E sim, existem inúmeras formas de ordenar uma lista e para de fato organizar. Podemos considerar que um algoritmo, ou programa é uma lista de instruções que serão executadas numa ordem determinada pelo desenvolvedor. Por isso, a lista é um dos conceitos mais utilizados na computação, até porque, a principal tarefa do computador é processar algum tipo de lista.

Vamos compreender que, em linhas gerais, uma lista possui funcionalidade de **CRUD** (*create, read, update, delete*), mas existem listas que não permitem incluir, alterar ou excluir elementos - elas permitem somente a consulta. Mas vamos deixar claro que a consulta dos elementos é obrigatória em todas as listas, pelo motivo mais básico: se você não puder consultar, então poderá somente inserir elementos na lista. Vamos entender o funcionamento de algumas destas listas especializadas, sendo a primeira as Listas ligadas.

**Listas ligadas** são constituídas de nós, no qual cada um deles possui o endereçamento para o próximo nó na lista. Isso implica que cada nó seja uma estrutura heterogênea, pois também contém dados que serão utilizados.

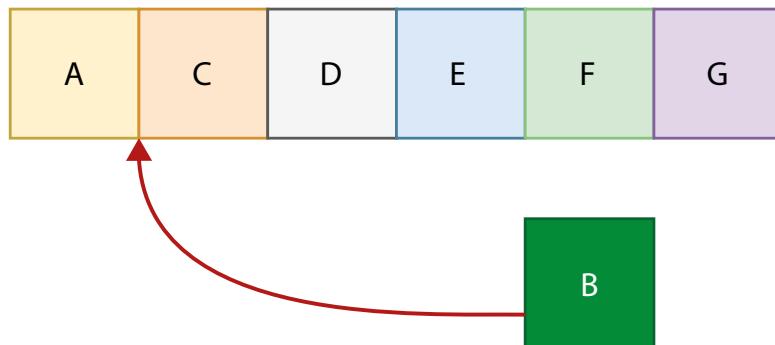
Mas, afinal, por que não podemos resolver o problema com um simples vetor (array)?

Para responder esta pergunta, vamos considerar uma situação do cotidiano, no qual uma distribuidora de frutas faz o abastecimento de diversas lojas de varejo. Por ser uma grande distribuidora, ela conta com um sistema que controla a rota das entregas, facilitando o gerenciamento. Esta distribuidora possui vários motoristas que saem com sua lista de pontos que devem ser entregues no dia. Durante o dia, a distribuidora recebe pedidos que devem ser realizados pelos motoristas, aproveitando as rotas já estabelecidas, assim possibilita uma economia considerável de entrega.

Apesar de funcionar, esta ação está ocorrendo de forma muito lenta no sistema, sem falar da remoção das lojas que não estão mais em funcionamento para que a distribuidora trabalhe apenas com lojas ativas e em pleno funcionamento. Após análise no sistema, descobrimos que este sistema utiliza apenas vetores simples para controlar a quantidade de dados de memória.

Mas afinal, o que acontece com o vetor? Ele não oferece um desempenho adequado? O problema não é o desempenho em si na busca, mas quando é necessário realizar inserção e/ou remoção de elementos em uma determinada ordenação, especialmente no início da lista. Isso ocorre porque com o vetor, é necessário movimentar todos os elementos para incluir ou remover um elemento em uma posição. Seria quase que se tivéssemos uma “fila india” com 100 pessoas e precisássemos incluir alguém no início da fila, logicamente, os 100 teriam de ir para trás e abrir o espaço no início, isso é bastante lento até todos se organizarem novamente.

Veja a simulação, a seguir, quando queremos incluir um elemento no início do vetor:



**Figura 2 - Incluir elemento no início da fila / Fonte: o Autor.**

**Descrição da Imagem:** uma fila de letras está sendo representado a inclusão do de uma nova letra entre a posição 1 e 2.

Operação simples, concorda? Porém vamos ver no detalhe da operação, para incluir o elemento no vetor, vamos considerar que primeiro inserimos ele na última posição.



**Figura 3 - Passo 1 para posicionar o novo elemento no início da fila / Fonte: o Autor.**

**Descrição da Imagem:** primeiro passo para posicionar o elemento na fila é incluir o elemento na última posição da fila.

Agora, precisamos movimentar este elemento para a posição que o antecede.



**Figura 4 - Passo 2 para posicionar o elemento na fila / Fonte: o Autor.**

**Descrição da Imagem:** segundo passo para posicionar o elemento na fila é deslocar para a penúltima posição.

Esta sequência se repete até encontrar a posição adequada:



Figura: 5 - Passos 3 ao 6 para reorganizar o elemento na segunda posição / Fonte: o Autor.

**Descrição da Imagem:** os passos restantes deslocando sempre o elemento uma posição à frente até ele alcançar o local de destino.

Não é tão simples assim, concorda? Cada posição é um processamento que o computador precisa realizar de alocação de memória. Para computadores atuais pode parecer pouco perceptível fazer esta alteração no cotidiano, mas pense que a cada dia temos mais dados a serem processados, e a escala começa a ficar realmente pesada para fazer este processamento.

Lembre-se que esta sistemática ocorre na exclusão, esse espaço vazio tem que ser ocupado pelo valor da posição seguinte e, assim, os valores são movidos até que os espaços abertos fiquem fechados. Uma das formas de resolver este problema é o uso da lista ligada, pois ela não vai depender da ação física da memória e sim pode ser usada a lógica para definir a sequência. Uma analogia interessante são as filas por senha, comuns em bancos. Você pega uma senha e espera sentado, eventualmente, alguém com atendimento preferencial é chamado na sua frente, mas não há um estresse de reposicionamento, você continua aguardando sentado ser chamado.

Vejamos o diagrama, a seguir, para entender como esta lógica funciona, utilizando o mesmo esquema anterior.

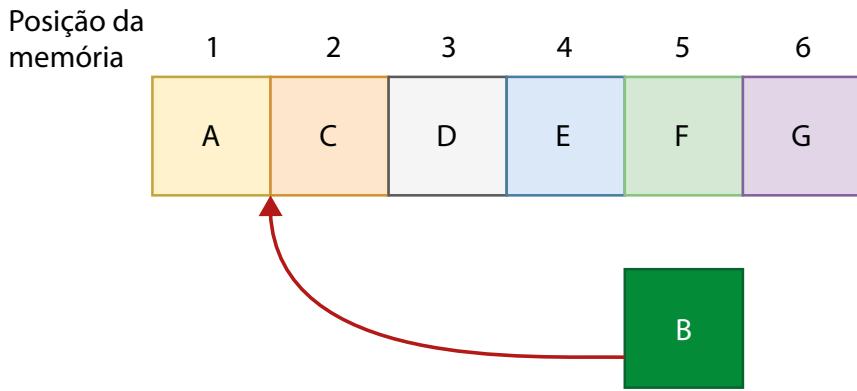


Figura 6 - Incluir um novo elemento na segunda posição em uma lista ligada / Fonte: o Autor.

**Descrição da Imagem:** incluir um novo elemento na lista ligada, oferecendo uma alternativa sem relocação de memória

Porém agora ao invés de reposicionar os elementos na memória, apenas vamos mudar um atributo do elemento, a posição lógica dele.

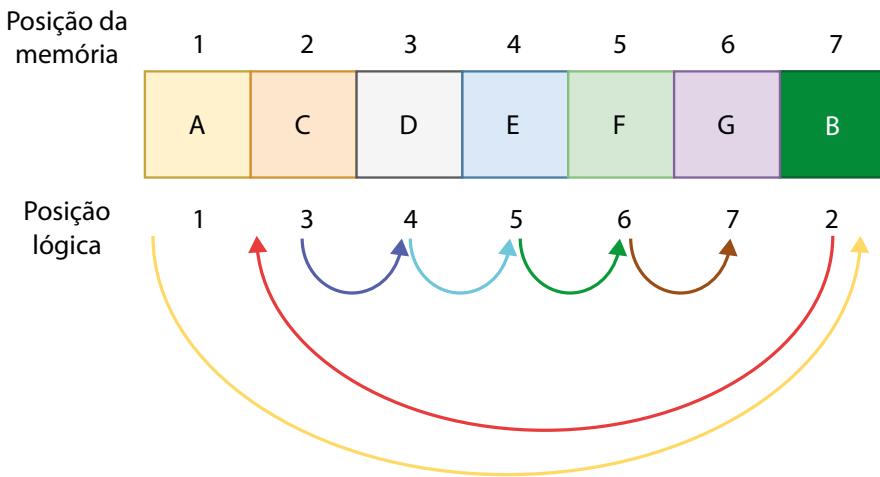


Figura 7 - Alterando a posição lógica. / Fonte: o Autor.

**Descrição da Imagem:** figura que representa os cenários de posição da memória e a posição lógica, na lista ligada à posição de memória não se altera. A posição lógica, que seria o equivalente ao número de ordem, é reorganizada para o novo elemento estar na segunda posição lógica.

Agora, não existe o mesmo custo de processamento, porque não reposicionamos nenhum elemento na memória, apenas é alterado um atributo dos elementos: posição lógica. Assim como na remoção se torna mais simples, assim como no exemplo:

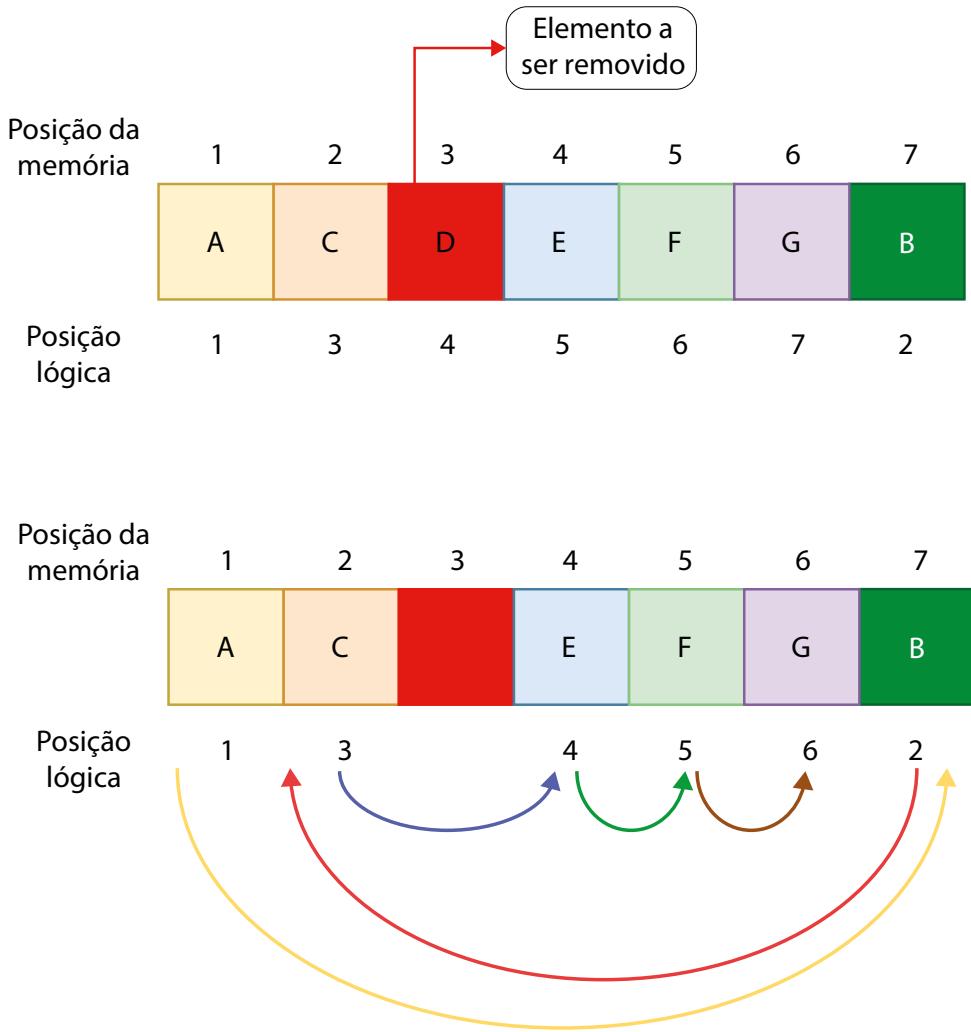


Figura 8 - Remove elemento da lista ligada. / Fonte: o Autor.

**Descrição da Imagem:** remover um elemento da lista ligada, implica em reorganizar a sequência lógica da fila. Não há reposicionamento de memória dos elementos.

Ajustar um valor é mais simples que reposicionar a memória, não acha? Assim como na vida, com o exemplo da fila india e da fila por senha do banco, nos sistemas computacionais também adotamos medidas similares para que possamos reduzir custos e melhorar o desempenho de uma operação.

Em um diagrama ou exemplificando com problemas do dia a dia parece ser simples, mas para fazer isso no código **Python** é preciso entender alguns elementos, o primeiro são as **classes**:

- Classe ElementoUnicoDaLista: responsável por manter o elemento, esta classe irá manter o conteúdo em si e o próximo elemento.
- Classe ListaLigada: responsável pelos métodos para operar a lista, nesta classe, que iremos fazer as operações de incluir, alterar e localizar. Esta classe é responsável por manter a lista.
- Classe Loja: responsável sobre o objeto a ser inserido na lista.

Vamos também dividir em duas partes, uma com a lista e o outro para iniciar a lista e manter os elementos a serem cadastrados. Partindo do exemplo anterior da distribuidora de frutas, vamos construir a classe ElementoUnicoDaLista:

```

1  class ElementoUnicoDaLista:
2      def __init__(self, elemento):
3          self.elementoDaLista = elemento
4          self.proximoElemento = None
5
6

```

Nesta classe, definimos como será mantido cada elemento da lista, iremos armazenar tanto o elemento atual quanto o próximo. Destaco aqui, na **linha 4**, que irá armazenar o elemento atual, e a **linha 5** que irá armazenar o próximo. Perceba que no momento da criação não há um próximo elemento, isso ocorre porque quando criamos os elementos é uma operação que antecede a inclusão na fila com posicionamento correto. Permitindo assim que cada elemento da lista saiba quem será o próximo na sequência.

A segunda classe é a da *ListaLigada* que irá manter a lista, vamos abordar ela por partes:

```

1 from elemento_lista import ElementoUnicoDaLista
2
3
4 class ListaLigada:
5     def __init__(self):
6         self._inicio = None
7         self._quantidade = 0
8
9     @property
10    def inicio(self):
11        return self._inicio
12
13    @property
14    def quantidade(self):
15        return self._quantidade
16

```

Vamos entender o código por partes, assim você terá maior clareza sobre o funcionamento desta lista.

- Linha 5: construtor da classe.
- Linha 6: atributo que armazena quem é o primeiro elemento, a partir do primeiro saberemos os outros.
- Linha 7: mantém a quantidade da lista, permitindo que a qualquer momento possamos saber quantos elementos estão na lista.
- Linha 9 e 13: é um **Python decotator**, isto é, define que um atributo terá um método para recuperar a informação.
- Linha 10 a 11: método criado para retornar o valor do atributo *início*.
- Linha 14 a 15: método criado para retornar o valor do atributo *quantidade*.

Nesta parte do código já conseguimos manter a lista, com os elementos e sequenciamento lógico. Vamos passar para os inserts:

```

17  def inserirNoInicioLista(self, conteudo):
18      elemento = ElementoUnicoDaLista(conteudo)
19      elemento.proximoElemento = self._inicio
20      self._inicio = elemento
21      self._quantidade += 1
22

```

Nesta etapa do código, iremos controlar as inversões, permitindo que os elementos começem a ser referenciados em lista, uma vez que cada elemento conhece o próximo.

- Linha 17: método que irá incluir um elemento no início da lista. Este método foi incluído por não precisar se preocupar qual elemento que antecede o que será inserido. E por que isso é importante? Quando inserir um elemento que seja diferente do início, precisaremos informar para o elemento anterior quem o elemento seguinte mudou.
- Linha 18: iremos armazenar o elemento da lista, neste momento, somente será criado o elemento com seu conteúdo, não há informação de quem será o próximo da lista.
- Linha 19: iremos posicionar quem é o próximo da lista, quando inserir o primeiro elemento não há um próximo. Porém quando vamos inserir um elemento na lista que já contém basta informar para o novo elemento que o seu próximo era o **antigo** primeiro elemento.

- Linha 20: agora, iremos informar para a lista que o primeiro elemento é o que estamos inserindo.
- Linha 21: aumentamos a quantidade do elemento.

O ponto fundamental desta estratégia é que não há reposicionamento de memória, apenas informamos para os atributos de controle quem é o início no elemento, quem é o próximo e aumentamos a quantidade. A lista como um todo será ajustada. Veja o esquema a seguir:

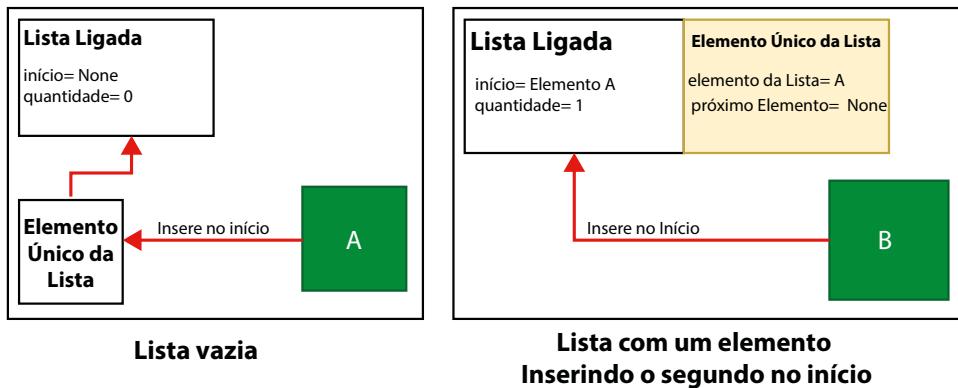


Figura 9 - Esquema do algoritmo da lista ligada / Fonte: o Autor.

**Descrição da Imagem:** a representação da imagem, no qual é incluído o elemento e os atributos de quem é o elemento no início, próximo elemento e quantidade são atualizados. Cada elemento incluído possui a informação de qual o elemento anterior e a quantidade na fila.

Perceba que a ideia é atualizar o atributo da lista e não movimentar dados na memória. Vejamos agora como inserir um elemento em qualquer posição:

```
23     def inserir(self, posicao, conteudo):
24         if posicao == 0:
25             self.inserirNoInicioLista(conteudo)
26             return
27         elemento = ElementoUnicoDaLista(conteudo)
28         esquerda = self._elemento(posicao - 1)
29         elemento.proximoElemento = esquerda.proximoElemento
30         esquerda.proximoElemento = elemento
31         self._quantidade += 1
32
33     def _elemento(self, posicao):
34         self._validarPosicao(posicao)
35         atual = self.inicio
36         for i in range(0, posicao):
37             atual = atual.proximoElemento
38         return atual
39
40     def _validarPosicao(self, posicao):
41         if 0 <= posicao < self.quantidade:
42             return True
43         raise IndexError("Posição inválida {posicao}")
44
```

Apesar de mudar um pouco o código de quando inserimos no início, a lógica é a mesma. Vejamos linha a linha:

- Linha 23: método que irá inserir um elemento em qualquer posição da lista. Veja que agora precisamos informar a posição que queremos incluir.
- Linha 24 a 26: condição para que se é o início do código, utilizamos o método descrito anteriormente.
- Linha 27: iremos armazenar o elemento da lista, neste momento, somente será criado o elemento com seu conteúdo, não há informação de quem será o próximo da lista.
- Linha 28: como já garantimos que não é o elemento no início da lista, e imaginando uma lista que observamos o crescimento da esquerda para direita, recuperamos o elemento da esquerda, ou seja, o elemento que antecede a posição que vamos inserir.
- Linha 29: agora que queremos o elemento à esquerda da posição que vamos inserir, este elemento possui a informação de quem é o próximo, no atributo *proximoElemento*. Portanto, o elemento que vamos inserir irá receber quem é o elemento na sua direção.



- Linha 20: o elemento na posição à esquerda do elemento atual, receberá a nova informação de quem está a sua direita.
- Linha 31: adiciona um número a quantidade de elementos.
- Linha 33: método para retornar o elemento na posição informada.
- Linha 34: método para verificar se a posição é válida.
- Linha 35: definimos uma variável com o elemento do início da lista.
- Linha 36 a 38: passa em todos os elementos afim de retornar o elemento na posição buscada. Perceba que não há um índice associado à posição, por isso, a busca é feita passando de um elemento para outro até encontrar o elemento na posição relacionada.
- Linha 40 a 43: método responsável em verificar se foi passado uma posição que está fora da quantidade de elementos que existem na lista.

Veja como ocorre as atualizações dos atributos em cada elemento da lista. Tudo isso sem necessitar alterar a posição de memória.

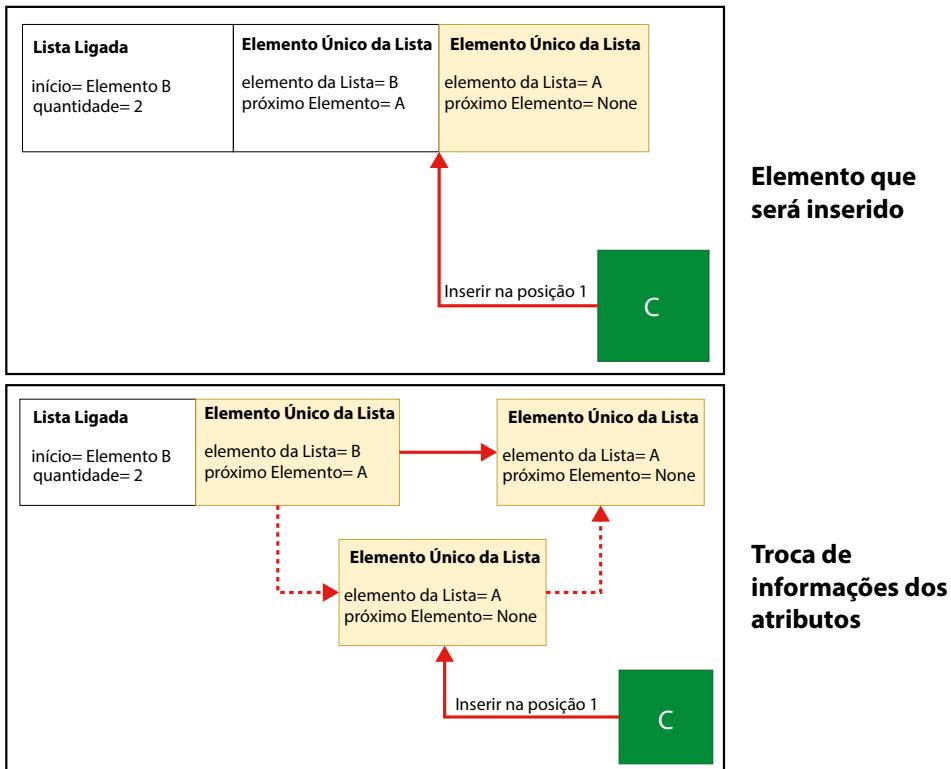
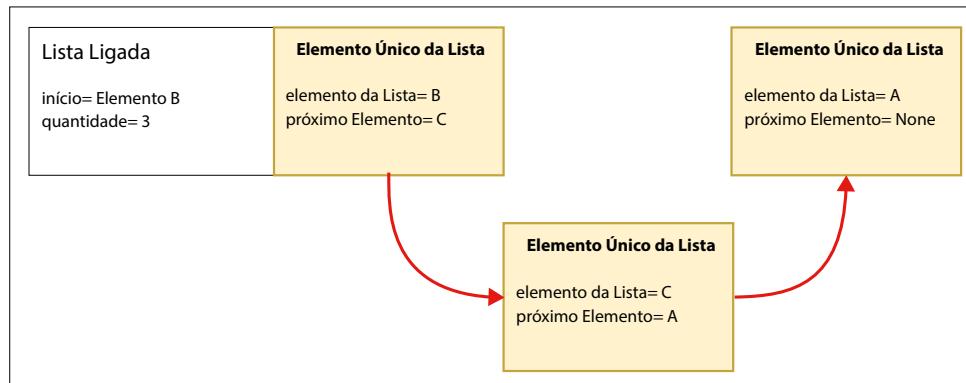


Figura 10 - Esquema do algoritmo para incluir na segunda posição. / Fonte: o Autor.

**Descrição da Imagem:** a lista ligada possui dois elementos e deseja-se inserir o terceiro na segunda posição. Para isso, é preciso atualizar os atributos de quantidade e próximo elemento.

O resultado final é uma lista atualizada com o primeiro elemento com a informação atualizada de quem é o próximo, assim como o novo elemento possui a informação de quem é o próximo. Veja que esta alteração impacta sempre em dois elementos, o anterior e o novo que irá ser incluído. Não importa a quantidade de elementos, sempre iremos trabalhar com dois elementos, porque não há um repositionamento físico.



### **Elementos com as informações atualizadas**

Figura 11 - Atualização dos atributos da lista / Fonte: o Autor.

**Descrição da Imagem:** atualizado os atributos de próximo elemento do primeiro, no qual aponta para o novo elemento, e o novo elemento que recebe as informações.

A inserção em qualquer posição está pronta, o fato de não precisarmos remanejar vários elementos faz com que ganhemos desempenho toda a vez que iremos alterar os dados da lista. Agora, vamos analisar o código no caso de remover elementos:

```

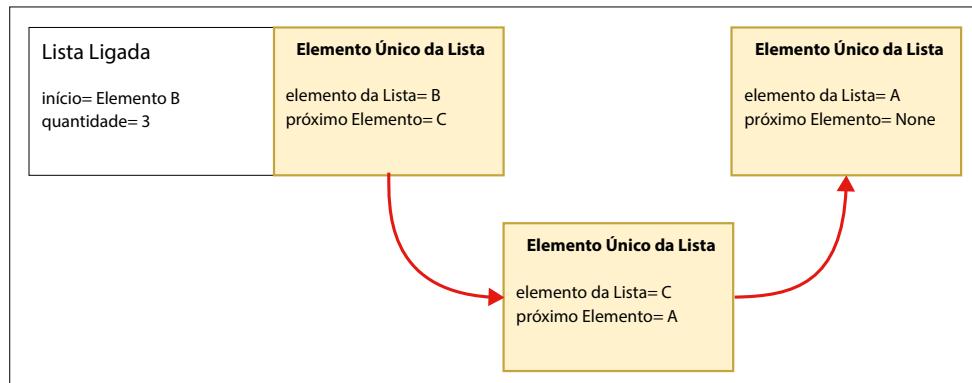
45  def removerDoInicio(self):
46      removido = self.inicio
47      self._inicio = removido.proximoElemento
48      removido.proximoElemento = None
49      self._quantidade -= 1
50      return removido.elementoDaLista
51
52  def remover(self, posicao):
53      esquerda = self._elemento(posicao - 1)
54      removido = esquerda.proximoElemento
55      esquerda.proximoElemento = removido.proximoElemento
56      removido.proximoElemento = None
57      self._quantidade -= 1
58      return removido.elementoDaLista

```

Sabendo como funciona a inserção de elementos, a remoção é a mesma operação com o detalhe de desfazer o que foi feito ao inserir.

- Linha 45: método para remover elementos no início da fila. É necessário um comportamento diferenciado, pois trabalharemos também o elemento anterior e no caso de ser o primeiro, o elemento anterior não existe.
- Linha 46: armazenamos o elemento que será removido, no caso é o primeiro.
- Linha 47: informamos que o elemento de início mudou, o elemento de início é o próximo do que estamos removendo.
- Linha 48: limpamos a informação de quem é o próximo do elemento que estamos removendo.
- Linha 49: diminuímos de elementos na lista.
- Linha 50: retornamos o elemento que foi removido.
- Linha 52: método para remover um elemento em qualquer posição, neste caso, passando a posição somente.
- Linha 53: armazenamos a informação do elemento à esquerda do que estamos removendo.
- Linha 54: coletamos a informação do elemento que iremos remover, como identificamos o elemento à esquerda da posição atual, descobrimos qual é o elemento na posição a ser excluída.
- Linha 55: atualizamos a informação do elemento anterior com a informação de quem é o próximo elemento que está sendo excluído.
- Linha 56: limpamos a informação do elemento que estamos removendo.
- Linha 57: diminui a quantidade de elementos na lista.
- Linha 58: retorna o elemento que foi removido.

A remoção acontece simplesmente atualizando os dados de quem são os próximos elementos. Reforçando: sem precisar reposicionar nenhum elemento em memória. Veja no diagrama, a seguir, a representação da exclusão de um elemento da lista ligada:

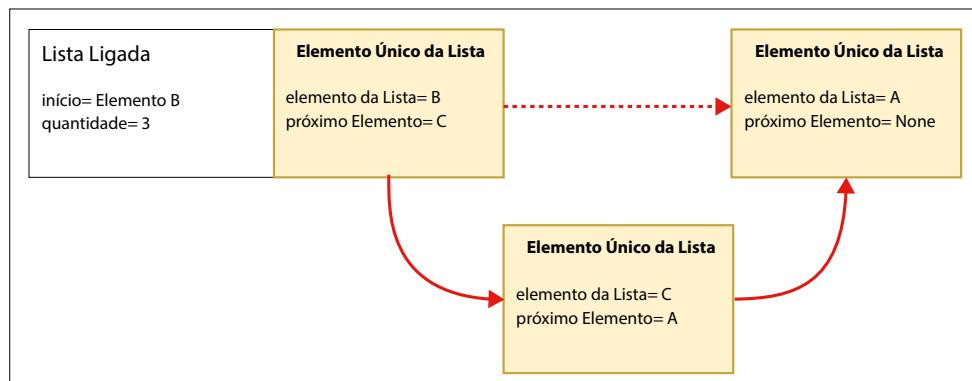


### Excluindo o Elemento C

Figura 12 - Remover o segundo elemento. / Fonte: o Autor.

**Descrição da Imagem:** na imagem representa a lista ligada no momento atual, contendo o elemento B, C, A.

Nesta apresentação, iremos remover o elemento C e, para isso, precisamos realizar um remapear as ligações entre os elementos. Porém diferente da ação de inserir, iremos agora remover, ou seja, remapear para os elementos que permanecem na lista.

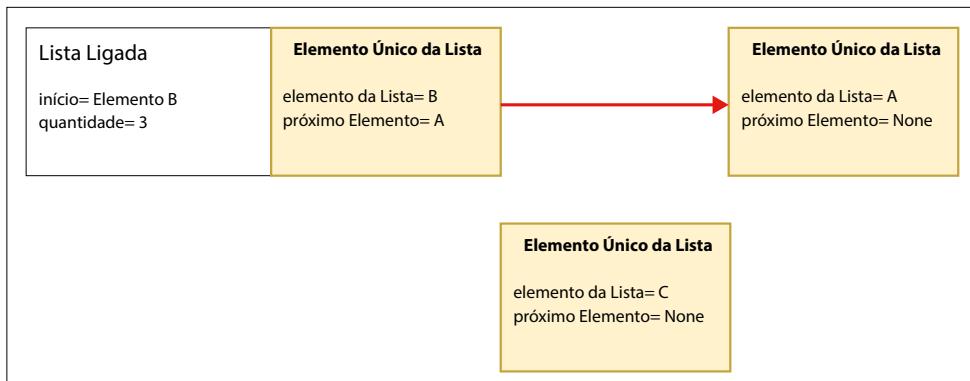


### Nova ligação Proposta

Figura 13 - Nova ligação sem o segundo elemento. / Fonte: o Autor.

**Descrição da Imagem:** a imagem mostra como será a nova ligação entre os elementos, neste contexto, a ligação será feita entre o elemento B e A.

Resultando então na seguinte ligação entre os elementos da lista:



### Ligaçāo sem o elemento C

Figura 14 - Lista ligada sem o elemento C / Fonte: o Autor.

**Descrição da Imagem:** a lista ligada sem o elemento C, no qual a primeira posição (elemento B) possui a informação do próximo (elemento A). A quantidade de elementos na fila é atualizada também para 2.

A busca de informações pela posição é parte essencial para qualquer lista e, para isso, vamos construir dois métodos, um para buscar um elemento pela posição, e outro para imprimir todos os elementos da lista.

```

59
60     def buscaElemento(self, posicao):
61         self._validarPosicao(posicao)
62         elemento = self._elemento(posicao)
63         return elemento.elementoDaLista
64
65     def imprimir(self):
66         atual = self.inicio
67         for i in range(0, self.quantidade):
68             print(atual.elementoDaLista)
69             atual = atual.proximoElemento
70
  
```

A busca de elementos fica mais simples agora, porque já criamos alguns métodos que varre a lista para encontrar um elemento na posição solicitada. A impressão é similar, mas a diferença mais marcante é que irá percorrer sempre a lista toda.

- Linha 60: método de busca de um elemento pela posição.
- Linha 61: verifica-se é uma posição válida.
- Linha 62: recupera o elemento na posição informada.
- Linha 63: retorna o valor contido no elemento da lista.
- Linha 65: método para imprimir todos os elementos da lista.
- Linha 66: armazena a informação do elemento que está no início da lista.
- Linha 67 a 69: varre todas as posições da lista imprimindo o valor contido no elemento e, por fim, atualiza a variável *atual* com o próximo elemento.

Nestes dois últimos métodos, fica evidente o ponto fraco da lista ligada: a consulta de elementos. Perceba que a lista não é um sequenciamento físico (em memória) e, com isso, precisamos fazer uma busca completa sobre todos os elementos até que chegue ao elemento na posição desejada. Porém conforme vimos no exemplo, a inclusão e movimentação dos elementos é muito mais produtiva. Para finalizar, vamos ver o projeto da distribuidora que precisamos manter as lojas.

```

1 from lista_ligada import ListaLigada, ElementoUnicoDaLista
2
3
4 class Loja:
5     def __init__(self, nome, endereco):
6         self._nome = nome
7         self._endereco = endereco
8
9     def __repr__(self):
10        return "{}\n {}".format(self._nome, self._endereco)
11
12
13 def main():
14     loja1 = Loja("Mercadinho do Zé", "Rua das frutas frescas, 1234")
15     loja2 = Loja("Direto do colono", "Rua do colono, 10000")
16     loja3 = Loja("Quitanda do Bairro", "Rua cds Cebolas, 9999")
17     loja4 = Loja("Boa Fruta", "Rua Eureka, 13254")
18     loja5 = Loja("Horti Agora", "Rua da Praia, 5464")
19     loja6 = Loja("Fruti-Fruti", "Av. dos Verdes, 5")
20     lista = ListaLigada()
21     lista.inserirNoInicioLista(loja1)
22     lista.inserirNoInicioLista(loja2)
23     lista.inserirNoInicioLista(loja3)
24     lista.inserir(1, loja4)
25     lista.inserir(0, loja5)
26     lista.inserir(lista.quantidade, loja6)
27     print(lista.quantidade)

```

```
28 lista.imprimir()
29 removido = lista.removerDoInicio()
30 print(lista.quantidade)
31 print("Removido: {}").format(removido)
32 print("\n")
33 print(lista.quantidade)
34 lista.imprimir()
35 print("-----")
36 print("elemento removido {}".format(lista.remover(2)))
37 print("\n")
38 print(lista.quantidade)
39 lista.imprimir()
40 print("\n\nEncontrando o elemento 2: {}".format(lista.buscaElemento(1)))
41
42
43 main()
```

O programa é simples, sem complexidades adicionais, mas é importante entendermos o funcionamento sem dúvidas e, para tal, vamos passar as linhas mais importantes.

- Linha 1: importando a classe da Lista Ligada.
- Linha 4 a 10: classe no qual definimos as informações sobre as lojas que serão entregues as mercadorias. Neste exemplo é o nome e endereço, além do método para imprimir os objetos da classe.
- Linha 13: classe principal main.
- Linha 14 a 19: instância dos objetos de exemplo para executar o programa.
- Linha 20: Instância da Lista Ligada.
- Linha 21 a 23: adiciona objetos do tipo loja no início da Lista Ligada.
- Linha 24 a 26: adiciona objetos do tipo loja em várias posições da Lista Ligada.
- Linha 27: imprime o número representando a quantidade de elementos da lista.
- Linha 28: imprime o conteúdo de todos os elementos da lista.
- Linha 29: remove o elemento o início da lista.
- Linha 30: imprime o número representando a quantidade de elementos da lista.
- Linha 31: imprime o elemento removido.
- Linha 33 a 34: imprime a quantidade de elementos na lista e o conteúdo de todos os elementos da lista.
- Linha 36: imprime o elemento removido na posição 3 (índice 2 iniciando em zero).

- Linha 38 a 39: imprime a quantidade de elementos na lista, e o conteúdo de todos os elementos da lista.
- Linha 40: realiza a busca pelo elemento com índice 1 e imprime o conteúdo do elemento encontrado.

A Lista Ligada construída funciona como um vetor, isto é, possui índices para recuperar elementos. Existem outras aplicações, além dos índices utilizando a lista ligada, que podem ser uma lista ordenada ou a busca por conteúdo, é possível ampliar as funcionalidades da lista para atender às necessidades que forem apresentadas. Apenas devemos criar novas implementações, de métodos e atributos para que seja possível resolver qualquer necessidade em que a estratégia de lista ligada se aplica melhor.

**Lista Duplamente Ligada** - a lista ligada aumenta consideravelmente a ação de incluir novos elementos em qualquer posição da lista. Sem sombra de dúvida, esta estratégia é muito importante, quando queremos uma melhor performance na inclusão e remoção de dados da lista. Porém há um problema, quando precisamos consultar ou incluir um dado no fim da lista, a ação fica lenta, pois não sabemos qual o último elemento da lista, precisamos passar por todos os elementos para chegar ao último elemento.

Existe uma estrutura que é derivada ou um melhoramento da lista ligada, que possui um melhor desempenho neste caso: a lista duplamente ligada. A estratégia é bem similar da lista ligada, com a principal diferença de que na lista ligada conhecemos somente o primeiro elemento. Já na lista duplamente ligada, conhecemos o elemento de início e o último, assim é possível alcançar mais rapidamente os últimos elementos. Mas sabendo o último elemento, em que ajuda no desempenho?

Veja que agora que conhecemos as extremidades de uma lista, pode ser necessário retornar um elemento em uma posição qualquer. O primeiro passo é saber qual é o índice do meio, assim poderemos decidir se vamos percorrer a lista do início ou do final. Ou seja, ao invés de percorrer a lista sempre em uma direção, como na lista ligada, agora podemos percorrer em duas direções. No pior cenário, teremos de percorrer metade da lista apenas para encontrar o elemento procurado. Diferente da lista ligada que teremos de percorrer toda a lista no pior cenário.

Parece complexo? Vamos fazer assim, a figura, a seguir, exemplifica uma lista duplamente ligada:

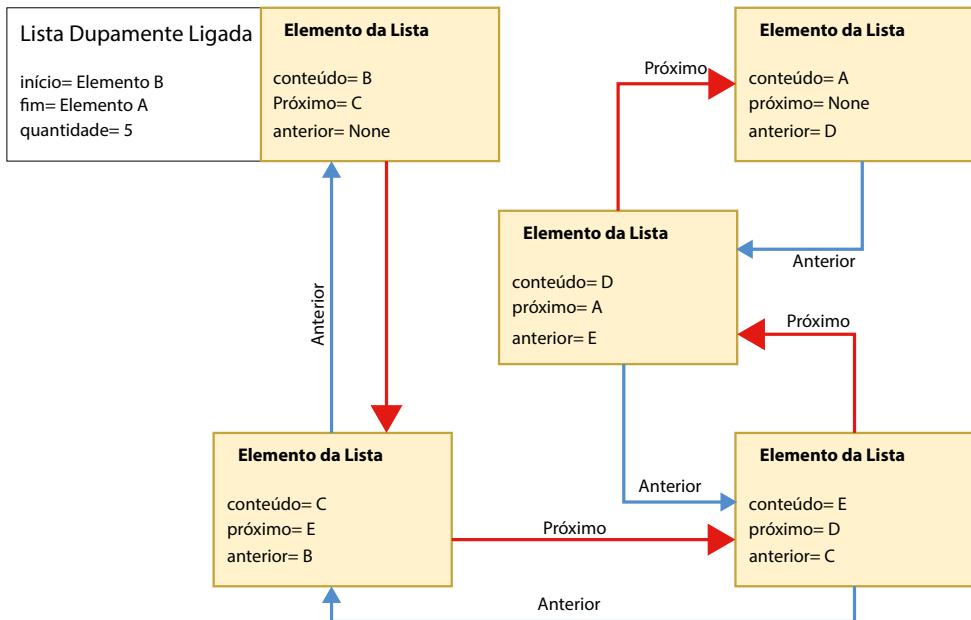


Figura 15 - Esquema de como é uma lista duplamente ligada. / Fonte: o Autor.

**Descrição da Imagem:** a representação de uma lista duplamente ligada com os elementos em ordem: B, C, E, D, A. Nesta representação, cada elemento possui a informação de qual é o próximo elemento e qual o elemento anterior.

Nesta representação, a lista conhece dois elementos: o primeiro e o último. Além disso, cada elemento da lista sabe quem é o próximo elemento e quem é o elemento anterior. Assim, podemos ter dois ponteiros em execução para percorrer a lista. Vamos ver na prática como essa estratégia funciona no código, considerando o exemplo da distribuidora.

```

1 class Elemento:
2     def __init__(self, conteudo):
3         self.conteudoElemento = conteudo
4         self.proximo: Elemento = None
5         self.anterior: Elemento = None
6
7

```

Nesta primeira classe, já temos uma diferença em comparação com a lista ligada. Além de possibilitar o armazenamento do conteúdo e do próximo elemento da lista, também poderemos armazenar o elemento anterior, como está descrito na linha 5. Vamos ver agora, a Classe da lista duplamente ligada por partes:

```
1 from elemento_da_lista import Elemento
2
3
4 class ListaDuplamenteLigada:
5     def __init__(self):
6         self._primeiro: Elemento = None
7         self._ultimo: Elemento = None
8         self._quantidade = 0
9
10    @property
11    def primeiro(self):
12        return self._primeiro
13
14    @property
15    def ultimo(self):
16        return self._ultimo
17
18    @property
19    def quantidade(self):
20        return self._quantidade
21
22    def _adicionaQuantidade(self):
23        self._quantidade += 1
24
25    def _diminuiQuantidade(self):
26        self._quantidade -= 1
27
```

Esta classe é responsável por controlar o funcionamento da lista duplamente ligada. Na classe, veremos todos os métodos para seu funcionamento:

- Linha 1: importação da classe Elemento, visto anteriormente.
- Linha 4: definição da classe *ListaDuplamenteLigada*.
- Linha 5 a 8: construtor com a definição dos atributos definidos da instância que irão controlar respectivamente:
  - o elemento na primeira posição;
  - o elemento na última posição;
  - a quantidade de elementos na lista.
- Linha 10 a 20: métodos responsáveis por recuperar os dados armazenados nos atributos.
- Linha 22 a 26: métodos criados para adicionar e diminuir valor 1 da quantidade de elementos da lista. Tem caráter exclusivamente de organização.

Já vimos os aspectos estruturais da classe da lista duplamente ligada, neste trecho, apenas definimos atributos e controladores básicos, a seguir, vamos analisar os métodos de inserir nas extremidades da lista.

```

28     def _inserirListaVazia(self, elemento):
29         self._primeiro = elemento
30         self._ultimo = elemento
31         self._adicionaQuantidade()
32
33     def _inserirNoInicio(self, conteudo):
34         elementoAtual = Elemento(conteudo)
35         if self._quantidade == 0:
36             return self._inserirListaVazia(elementoAtual)
37         elementoAtual.proximo = self._primeiro
38         self._primeiro.anterior = elementoAtual
39         self._primeiro = elementoAtual
40         self._adicionaQuantidade()
41
42     def _inserirNoFim(self, conteudo):
43         elementoAtual = Elemento(conteudo)
44         if self._quantidade == 0:
45             return self._inserirListaVazia(elementoAtual)
46         elementoAtual.anterior = self._ultimo
47         self._ultimo.proximo = elementoAtual
48         self._ultimo = elementoAtual
49         self._adicionaQuantidade()
50

```

Agora, já começamos a analisar a lógica básica da lista duplamente ligada, seguindo os passos:

- Linha 28: define o método para inserir na lista vazia, neste caso, abordamos um cenário em que o método é privado, ou seja, somente está acessível dentro da classe lista.
- Linha 29 a 30: nos atributos da lista estão sendo armazenados o elemento passado no parâmetro. Perceba, aqui, que estamos inserindo o mesmo elemento como primeiro e como último, pois como no momento somente teremos 1 elemento na lista, ele é o primeiro e último ao mesmo tempo.
- Linha 31: adiciona o valor 1 na quantidade.
- Linha 33 a 34: definição do método de inserir no início da lista, assim como instanciar o elemento a ser incluído.

- Linha 35 a 36: é verificado se a lista está vazia, se estiver será resolvido pelo método de inserir um elemento na lista vazia.
- Linha 37: no elemento que estamos incluindo, iremos armazenar a referência de quem será o próximo. Como estamos incluindo no início da lista, o próximo será o elemento que até então estava posicionado como primeiro.
- Linha 38: é definido quem é o anterior do primeiro elemento, sendo o elemento que estamos inserindo.
- Linha 39: é substituído o primeiro elemento pelo elemento que está sendo inserido.
- Linha 40: adiciona o valor 1 na quantidade.
- Linha 42 a 45: definição do método de inserir no fim da lista, assim como instanciar o elemento a ser incluído, aqui também verificamos se a lista está vazia, esta ação é preventiva, caso seja este método a ser chamado, será realizado a mesma ação da lista vazia.
- Linha 46: no elemento que será inserido, é atribuído o valor do elemento anterior. Como iremos inserir na última posição, o elemento anterior é o que estava armazenado no atributo de último elemento da lista.
- Linha 47: no último elemento da lista armazenado é atualizado o valor do próximo, como irá passar a ser o penúltimo recebe o elemento que está sendo inserido.
- Linha 48: é substituída a informação de quem é o último elemento da lista.
- Linha 49: adiciona o valor 1 na quantidade.

Com a definição de como será o comportamento, para inserir nas extremidades da lista, e quando a lista estiver vazia for definida. Estes métodos são criados para diminuir a complexidade de concentrar todos os cenários em um único método, facilitando a manutenção e entendimento da lista. Agora, vamos analisar a inclusão de um elemento em qualquer posição da lista.

```

51  def inserir(self, posicao, conteudo):
52      if posicao == 0:
53          return self._inserirNoInicio(conteudo)
54      if posicao >= (self.quantidade - 1):
55          return self._inserirNoFim(conteudo)
56
57      elementoAtual = Elemento(conteudo)
58      elementoDireita = self._elemento(posicao)
59      elementoEsquerda = elementoDireita.anterior
60
61      elementoAtual.proximo = elementoDireita
62      elementoAtual.anterior = elementoEsquerda
63
64      elementoEsquerda.proximo = elementoAtual
65      elementoDireita.anterior = elementoAtual
66      self._adicionaQuantidade()
67

```

No método de inserir, podemos incluir um elemento em qualquer posição da lista controlando quem é o elemento anterior e posterior de quem iremos incluir.

- Linha 51 a 55: definição do método de inserir, também é verificado se a posição a ser inserida é a primeira ou a última. Quando for uma das duas, será redirecionado para os métodos especialistas vistos anteriormente.
- Linha 57: instanciado o novo elemento, neste momento apenas é um elemento sem posicionamento.
- Linha 58: a ideia é simular um deslocamento de uma lista para a direita, pois naturalmente a lista cresce de forma positiva quanto a quantidade, por isso, a variável *elementoDireita* irá receber o elemento que, atualmente, ocupa a posição que será inserida. O método *self.\_elemento(posicao)* será explicado mais à frente.
- Linha 59: como foi recuperado o elemento que ficará à direita do elemento a ser inserido, ele contém a informação de que está à esquerda, assim é atribuído a variável *elementoEsquerda* o seu elemento.
- Linha 61 a 62: com os elementos da esquerda e da direita da posição a ser inserida identificados, é informado para o elemento a ser inserido quem são estes elementos armazenados nas propriedades de anterior e próximo formando a ligação da lista.
- Linha 64 a 66: neste ponto é fechado o ciclo de ligação da lista e, para isso, apenas precisamos informar para os dois elementos (da esquerda e

da direita da posição) quem é o elemento que está sendo inserido. Após isso, é incrementado a quantidade de elementos na lista.

Veja que o funcionamento e controle são similares aos da lista ligada, controles de quais elementos são os “vizinhos” de quem estamos inserindo. Mas o grande ganho, será possível observar na consulta por posição, veja:

```

68  def _validarPosicao(self, posicao):
69      if 0 <= posicao < self.quantidade:
70          return True
71      raise IndexError("Posição inválida {posicao}")
72
73  def _elemento(self, posicao):
74      self._validarPosicao(posicao)
75      metade = self.quantidade // 2
76      if posicao < metade:
77          print("**** busca na metade inferior - metade: {}, posição: {}****".
format(metade, posicao))
78          atual = self._buscaElementoMenorMetade(posicao)
79      else:
80          print("**** busca na metade superior - metade: {}, posição: {}****".
format(metade, posicao))
81          atual = self._buscaElementoMaiorMetade(posicao)
82      return atual
83
84  def _buscaElementoMaiorMetade(self, posicao):
85      atual = self.ultimo
86      for i in range(posicao + 1, self.quantidade)[::-1]:
87          atual = atual.anterior
88      return atual
89
90  def _buscaElementoMenorMetade(self, posicao):
91      atual = self._primeiro
92      for i in range(0, posicao):
93          atual = atual.proximo
94      return atual
95

```

Nesta parte do código, poderemos entender melhor o grande ganho desta lista, pois nela podemos observar os ponteiros sendo utilizados mais claramente, passando o código passo a passo:

- Linha 68 a 71: assim como na lista ligada, é verificado se o valor da posição informado não extrapola os limites da lista, caso sim irá lançar um erro.
- Linha 73 a 74: método criado para buscar um elemento pela posição. Veja que aqui é validado se é uma posição válida antes de prosseguir.

- Linha 75: nesta linha, coletamos a informação de qual é o índice que está no meio da lista. A divisão que está sendo executada somente retorna o valor inteiro, mesmo que a divisão não seja exata será truncado o valor (sem decimais).
- Linha 76 a 82: verificação se a posição que está sendo buscada está em que parte da lista, na menor que a metade ou na maior que a metade. Essa condição direciona para a lógica apropriada a fim de encontrar o elemento mais rapidamente. Após encontrar, retornar o elemento.
- Linha 84 a 88: busca o elemento que está “na parte de final da lista”, ou seja, está na posição maior que a metade da lista. A busca inicia pela última posição e vai diminuindo até encontrar o elemento.
- Linha 90 a 94: busca o elemento na primeira metade da lista, esta busca é igual da lista ligada passando por todos os elementos até chegar na posição.

Naturalmente, a complexidade aumenta um pouco quando se desenvolve o programa, mas o ganho é perceptível por se tratar de metade das buscas realizadas em comparação a lista ligada. Vejamos a remoção dos elementos:

```

96  def _removerUltimo(self):
97      elementoRemovido = self.ultimo
98      novoUltimo = elementoRemovido.anterior
99      novoUltimo.proximo = novoUltimo
100     self._ultimo = novoUltimo
101     elementoRemovido._anterior = elementoRemovido._proxima = None
102     self._diminuiQuantidade()
103     return elementoRemovido.conteudoElemento
104
105    def _removerPrimeiro(self):
106        if self.quantidade == 1:
107            return self._esvaziarLista()
108        elementoRemovido = self.primeiro
109        novoPrimeiro = elementoRemovido.proximo
110        novoPrimeiro.anterior = None
111        self._primeiro = novoPrimeiro
112        elementoRemovido.anterior = elementoRemovido.proximo = None
113        self._diminuiQuantidade()
114        return elementoRemovido.conteudoElemento
115
116    def _esvaziarLista(self):
117        if self._quantidade == 1:
118            elementoRemovido = self.primeiro

```

```
119     self._primeiro = None
120     self._ultimo = None
121     elementoRemovido.anterior = elementoRemovido.proximo = None
122     self._diminuiQuantidade()
123     return elementoRemovido.conteudoElemento
124
```

Remover um elemento da lista segue a mesma lógica de reconfigurar os atributos de anterior e próximo, vejamos no detalhe:

- Linha 96 a 97: definição do método para remover o último elemento da lista, veja que aqui o primeiro passo é recuperar qual é o último elemento.
- Linha 98 a 100: reconfigurado o elemento que passará a ser o último.
- Linha 101 a 103: as informações de anterior e próximas são limpas do elemento removido. Diminui a quantidade da lista e retorna o valor do elemento removido.
- Linha 105 a 107: definição do método responsável por remover o primeiro elemento da lista, além de verificar se o primeiro elemento é o último para direcionar os métodos que irá tratar a lista vazia.
- Linha 108: recuperamos o elemento a ser removido, como é o primeiro da lista, ele está na primeira posição.
- Linha 109 a 111: a configuração do elemento que passa a ser o primeiro é realizada.
- Linha 112 a 114: o elemento que foi removido tem os atributos de anterior e próximos limpos, após esta ação é diminuído a quantidade de elementos e retorna o conteúdo do elemento excluído.
- Linha 116 a 123: método responsável para tratar quando uma lista está vazia. Se faz necessário para garantir que os atributos primeiro e último estejam vazios.

Uma vez entendido a inserção, remover é a mesma operação só com substituições de referências realizadas de forma inversa, veja o restante do código:

```

125  def remover(self, posicao):
126      if posicao == 0:
127          return self._removerPrimeiro()
128      if posicao >= (self.quantidade - 1):
129          return self._removerUltimo()
130      elementoRemovido = self._elemento(posicao)
131      elementoEsquerda = elementoRemovido.anterior
132      elementoDireita = elementoRemovido.proximo
133      elementoEsquerda.proximo = elementoDireita
134      elementoDireita.anterior = elementoEsquerda
135      elementoRemovido.proximo = None
136      elementoRemovido.anterior = None
137      self._diminuiQuantidade()
138      return elementoRemovido.conteudoElemento
139
140  def buscaElemento(self, posicao):
141      elemento = self._elemento(posicao)
142      return elemento.conteudoElemento
143
144  def imprimir(self):
145      atual = self.primeiro
146      for i in range(0, self.quantidade):
147          print(atual.conteudoElemento)
148          atual = atual.proximo
149

```

Nesta última parte da classe da lista duplamente ligada está a remoção por posição, busca do elemento pelo índice e a impressão de todos os elementos da lista, vejamos no detalhe:

- Linha 125 a 129: método que permite a remoção por posição. Como a estratégia foi proteger os métodos, este é que está disponível para programas externos a classe executarem. Neste ponto é verificado se o elemento a ser excluído está nas extremidades da lista, para caso estiverem será chamado o método apropriado.
- Linha 130 a 132: identificado o elemento que será excluído, conforme a posição solicitada, são encontrados os elementos da esquerda e da direita em relação ao elemento excluído.
- Linha 133 a 134: reconfigurado os elementos da esquerda e da direita para ele “se conhecerem”, isto é, nos atributos próximo e anterior é substituído o valor do elemento removido por novos valores de referência refazendo a ligação da lista.

- Linha 135 a 138: os atributos do elemento removido são limpos, é atualizado a quantidade de elementos e retorna o valor do elemento excluído.
- Linha 140 a 142: método aberto para realizar a busca por posição, como já foi criado o método que faz isso, somente é feita a chamada e retorna o valor.
- Linha 144 a 148: método que passa por todos os elementos com o intuito de imprimir o conteúdo de todos os elementos.

A lista duplamente ligada, claramente, possui um desempenho melhor que a lista ligada, é possível comparar o funcionamento das duas e, para isso, teríamos de inserir algumas centenas de milhares de dados. Centenas de milhares, porque os processadores modernos iriam comparar este teste exploratório de percepção. Por sorte a ideia é simples, se eu sei qual é a metade e posso iniciar tanto do início quanto do final, não preciso mais percorrer a lista toda para encontrar um elemento, apenas a metade dela.

Na sequência, o código para executar o teste da lista duplamente ligada, similar ao da lista ligada. Neste código serão realizados os testes de inserir, consultar, remover e imprimir. Inclusive os dados são fixos para efeito de teste e validação, contudo podem ser alterados a fim de representar uma gama maior de dados. Importante mencionar que a lista ligada funciona com qualquer elemento, somente necessita ser enviado os dados para a lista, e ele irá controlar a dinâmica da inclusão e a remoção dos dados.

```

1 from lista_duplamente_ligada import ListaDuplamenteLigada
2
3
4 class Loja:
5     def __init__(self, nome, endereco):
6         self._nome = nome
7         self._endereco = endereco
8
9     def __repr__(self):
10        return "{}\n {}".format(self._nome, self._endereco)
11
12
13 def statusLista(lista):
14     print(lista.quantidade)
15     lista.imprimir()

```

```
16
17
18 def main():
19     loja1 = Loja("Mercadinho do Zé", "Rua das frutas frescas, 1234")
20     loja2 = Loja("Direto do colono", "Rua do colono, 10000")
21     loja3 = Loja("Quitanda do Bairro", "Rua cds Cebolas, 9999")
22     loja4 = Loja("Boa Fruta", "Rua Eureka, 13254")
23     loja5 = Loja("Horti Agora", "Rua da Praia, 5464")
24     loja6 = Loja("Fruti-Fruti", "Av. dos Verdes, 5")
25     lista = ListaDuplamenteLigada()
26     lista.inserir(0, loja1)
27     lista.inserir(0, loja2)
28     statusLista(lista)
29     lista.inserir(1, loja3)
30     statusLista(lista)
31     lista.inserir(1, loja4)
32     lista.inserir(1, loja5)
33     lista.inserir(3, loja6)
34     statusLista(lista)
35     print("\n")
36     print("Consultando o índice 3 {}".format(lista.buscaElemento(2)))
37     print("Consultando o índice 4 {}".format(lista.buscaElemento(4)))
38     print("\n-----REMOVER-----")
39     lista.remover(0)
40     statusLista(lista)
41     lista.remover(lista.quantidade - 1)
42     statusLista(lista)
43     lista.remover(1)
44     statusLista(lista)
45     lista.remover(0)
46     lista.remover(0)
47     lista.remover(0)
48     statusLista(lista)
49
50
51 main()
52
```

Por este código ser muito similar à lista ligada, apenas vamos passar os pontos importantes para seu funcionamento.

- Linha 1 a 10: importação da lista duplamente ligada e a criação da classe Loja que receberá os objetos a serem incluídos na lista.
- Linha 13 a 15: função para organização do teste, assim sempre que for impresso os dados da lista, também será exibido a quantidade.

- Linha 18 a 24: função principal é a criação de cada objeto do tipo Loja.
- Linha 25 a 33: criado a instância da lista e incluído os objetos na lista e determinadas posições.
- Linha 36 a 37: realiza consulta por índice, imprimindo os valores.
- Linha 39 a 48: remove os elementos da lista em diversas posições, do início até a último (linha 41 remove o elemento na última posição). Após a remoção é impresso o status da lista na console até ela estar vazia.

Existe uma estrutura de dados que faz parte da biblioteca padrão do Python e corresponde à lista duplamente ligada, pois também é otimizado para inserção e remoção dos itens nas extremidades. Chama-se de *deque* e para importar sómente precisa importar “*from collections import deque*”.

Segundo o site oficial do Python seus atributos são:

*append* - adiciona a direita da lista.

*appendleft* - adiciona do lado esquerdo da lista.

*clear* - remove todos os elementos do *deque* deixando-o com comprimento 0.

*copy* - cria um cópia de uma lista.

*count* - corresponde à quantidade.

*extend* - estende o lado direito da lista.

*extendleft* - estende o lado esquerdo da lista.

*index* - retorna a posição de um elemento.

*insert* - corresponde a inserir em uma dada posição.

*maxlen* - tamanho máximo de uma lista se for limitado.

*pop* - remover do fim.

*popleft* - remover do início.

*remove* - remover de qualquer posição.

*reverse* - inverte os elementos da lista.



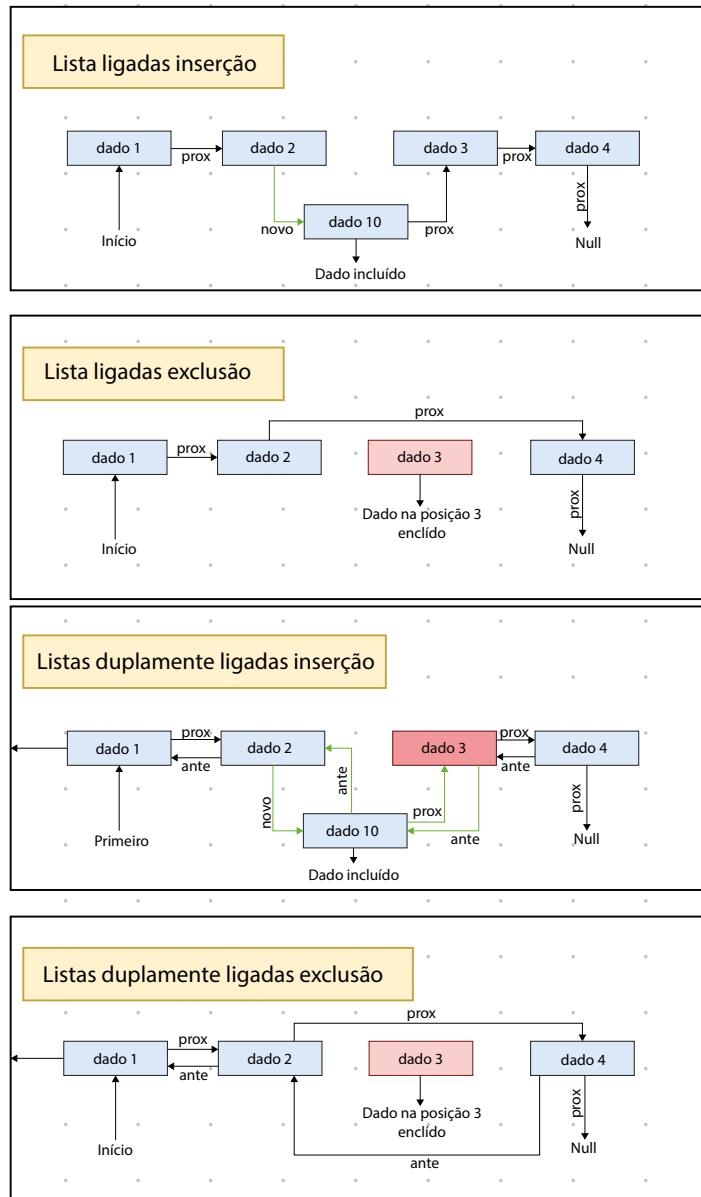
#### PENSANDO JUNTOS

Nesta unidade, abordamos as listas ligadas e duplamente ligadas, ambas permitem uma maior flexibilidade na ordem de inserção e exclusão dos elementos e sem realocar espaços em memória sem a necessidade. Pense e reflita, quando devemos usar uma estrutura de fila e quando devemos usar listas ligadas?



## OLHAR CONCEITUAL

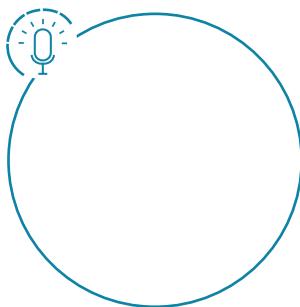
Durante esta unidade, estudamos listas dinâmicas (ligadas e duplamente ligadas) em Python, observe que na inserção e exclusão de um elemento só mexemos nos ponteiros e não na ordenação dos elementos. Olhem o infográfico, a seguir, para entender como funcionam essas listas dinâmicas.



**EXPLORANDO IDEIAS**

Uma das principais características de uma lista é possuir na estrutura do seu nó uma ligação que armazena o endereço do próximo elemento. Esta regra vai existir para qualquer tipo de implementação de uma lista, seja ela dinâmica ou estática.

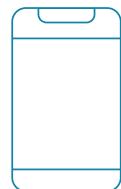
Lembrando que as listas dinâmicas armazenam seus elementos em posições que não necessitam ser contíguas na memória do computador, porém devemos cuidar com as ligações dos nós que também ocupam espaços na memória. As listas ligadas facilitam operações de remoção e inserção de elementos, uma vez que não necessitam de deslocamentos dos seus nós para realizar tais operações.



Agora, já vimos as estruturas de dados lineares (fila e pilha) e listas ligadas. Sendo que cada estrutura deve ser utilizada com consciência e sabendo das suas vantagens e desvantagens, podemos escolher a melhor estrutura que vai nos auxiliar na implementação. Neste Podcast, vamos conversar e falar mais das estruturas de dados dinâmicas e quais as suas vantagens e desvantagens no uso do dia a dia do desenvolvimento e como é a manutenção dessas listas, acesse o QR code para conferir essa conversa.

**NOVAS DESCOBERTAS**

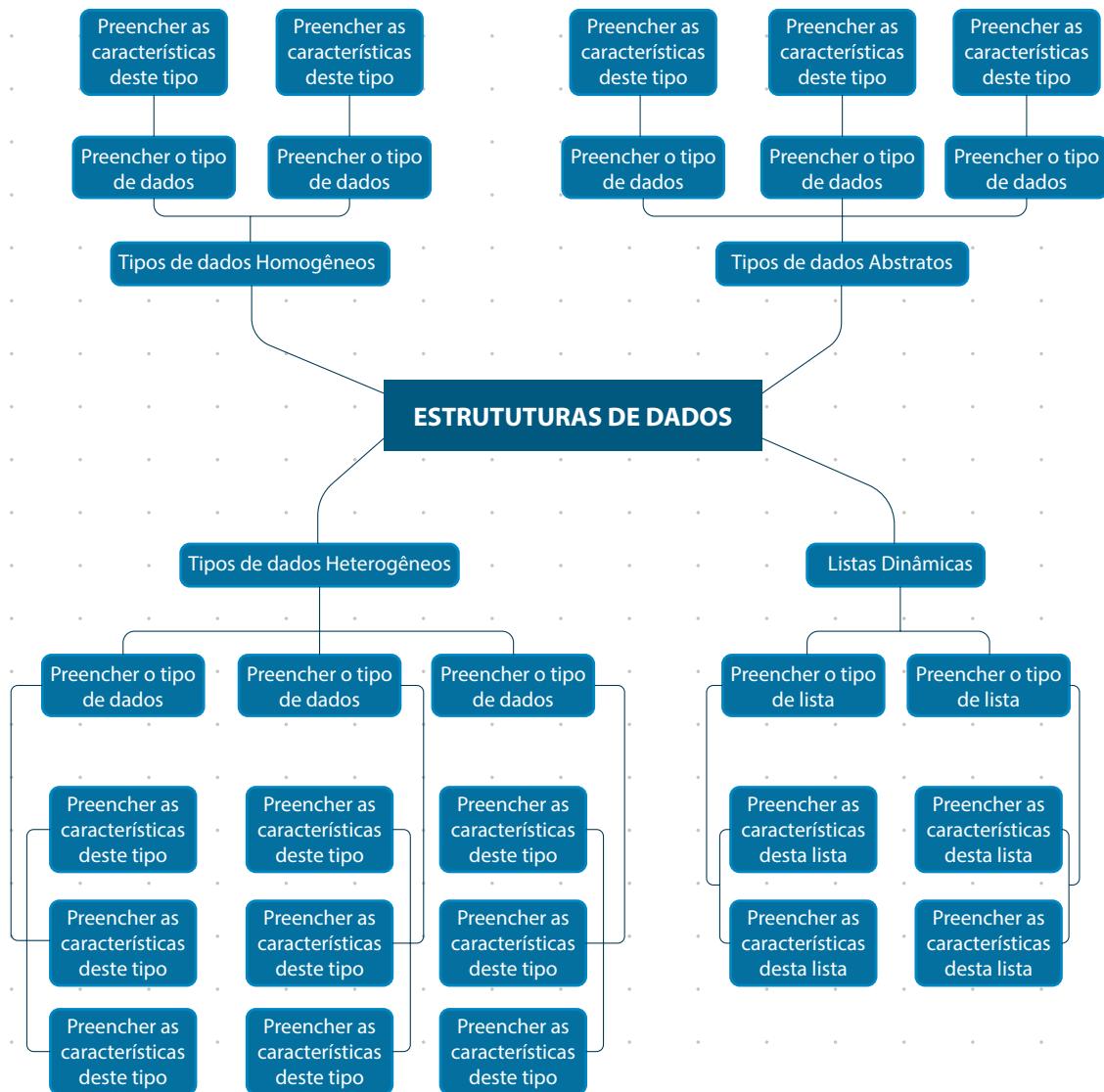
Você conhece o jogo Trilha? Considerando que você mexe as peças de acordo com as ligações, conseguimos utilizar os conceitos de listas ligadas. Jogue e veja que funciona exatamente as listas ligadas. Você pode jogar ele de forma online no QR.



De acordo como vimos nesta unidade, é importante entender quais situações as listas ligadas e duplamente ligadas atendem. Com base nos seus estudos, converse com profissionais da área e crie um quadro das vantagens e desvantagens das duas abordagens na criação e manutenção de sistemas.



Estruture um mapa mental com cada estrutura de dados vista até agora, lembrando que temos “tipo de dados”, “tipos de dados abstratos” e de “listas”. Lembre-se que os tipos de dados estão distribuídos em homogêneas e heterogêneas, os abstratos são relacionados às pilhas, filas e *deques* e as listas ligadas podem ser simples ou duplamente encadeadas. Você pode fazer seu mapa mental usando a ferramenta gratuita [www.goconqr.com](http://www.goconqr.com) ou mindmeister.



# Teoria de grafos aplicado a Python

Esp. Edson Orivaldo Lessa Junior

Esp. Rafael Orivaldo Lessa

## OPORTUNIDADES DE APRENDIZAGEM

Nesta unidade, vamos nos aprofundar em estruturas de dados em grafos, para resolver problemas matemáticos relacionados a ligações entre esses objetos de dados. A teoria de grafos é largamente utilizada para resolver contextos específicos, por exemplo, a ligação de amizade de rede social ou para criar uma gestão de conhecimento de uma empresa. Sendo assim, os grafos se tornam uma das estruturas de dados mais interessantes, ele é uma ferramenta poderosa e versátil, que nos permite representar facilmente a relação entre vários tipos de dados e recuperar esses dados de forma estruturada.

Um dos assuntos mais interessantes e pesquisados é a teoria de Grafos. Os Grafos, na estrutura de dados, permitem modelar, resolver problemas reais de custo, eficiência, logística e caminho mais curto. Ele é composto por dois conjuntos, um de vértices e outro de ligações (que é a relação entre os vértices).

Um problema muito interessante que é resolvido com Grafo é o do GPS. Quando buscamos um endereço do Waze, ou num GPS, as informações das ruas estão armazenadas em uma estrutura de dados em forma de grafo e, após a busca, é rodado um algoritmo para apresentar, na interface, possíveis caminhos do seu ponto atual até o destino e, ainda, pode exibir alternativas de caminhos, dependendo da configuração, como: evitar estrada de chão, caminho mais curto e caminho mais rápido. Agora, pense nesse mesmo problema do Waze para a aviação, como o grafo pode auxiliar para resolver diversos problemas, desde a visualização dos aviões, no radar, ao planejamento da malha área? Reflita como essa ferramenta é poderosa no nosso dia a dia.

A teoria de grafos fornece uma abstração simples e elegante para modelar os dados e seus relacionamentos. Formalmente, um grafo ou rede consiste em um conjunto de vértices, os quais podem representar as ligações, e um conjunto de arestas, que servem para ligar dois vértices, representando um relacionamento entre dados. Por meio desta abstração, é possível representar diversos fenômenos do mundo real.

Nos últimos anos, o tamanho dos grafos tem crescido exponencialmente, alavancado, principalmente, pelo fenômeno Big Data. Por exemplo, a rede social do Facebook, a qual pode ser representada por um grande grafo social, atingiu um tamanho de 3 bilhões de usuários, em 2021. É fácil imaginar que este grafo possui uma quantidade de arestas com, pelo menos, duas ordens de grandeza maiores que o número de vértices. Outro exemplo de grande grafo é de páginas webs conectadas, de acordo com Meusel *et al.* (2015), que mostra o estudo da webdatacommons, extraído em 2012, que a Web possui 3,53 bilhões de vértices, representando páginas da Web, e 128,736 bilhões de arestas, representando os links entre essas páginas. Outro dado interessante dessa mesma pesquisa é que são 101 milhões de vértices de host's e 2,043 milhões de arestas de links entre os hosts.

Podemos observar que a internet é um grande grafo todo interconectado, e que as páginas de buscas e as redes sociais têm contribuído muito para essa evolução da rede de computadores e da teoria de grafos.

Em estrutura de dados, a teoria dos grafos estuda as relações entre os objetos de um determinado conjunto. Esse tipo de armazenamento permite fazer represen-

tações visuais e resolver problemas matemáticos. Podemos aplicar isso em outros problemas: pense como uma empresa de aviação, assim cada aeroporto é considerado um vértice no gráfico, e a rota do voo entre eles é uma aresta. Se um passageiro decidir viajar de Florianópolis no até Orlando e não há um voo direto entre essas duas capitais, o sistema pode buscar e oferecer várias opções de conexões entre diferentes rotas. Agora, faça um grafo representando essa viagem no mapa.

Com as estruturas de dados em grafos, elas trazem benefícios do seu uso, entre eles, a crescente quantidade de informação que precisamos lidar para tomar decisões, a falta de padronização destes mesmos dados, impactando na qualidade e, em última análise, nas ideias retiradas dessa informação. Para conseguir trabalhar com esse universo de dados não estruturados, o grafo é a única maneira implementável e sustentável para resolver os problemas de interconexões. Contudo, esse tipo de estrutura também traz algumas dificuldades. Como ela depende de uma estrutura criada pelas arestas e vértices/nó, é necessário que o desenvolvedor crie essa estrutura de forma correta. Anote, no seu Diário de Bordo, como a análise de grafo pode auxiliar as pessoas no dia a dia do trânsito.

## DIÁRIO DE BORDO

## História da Teoria dos Grafos

O estudo e evolução do campo das ciências iniciam em um ponto em comum: um problema do cotidiano em que se pretende resolver por meio de teoremas e aplicabilidade em escala. A Teoria dos Grafos não é diferente, a sua história remonta um problema em uma cidade russa (na época Prússia), chamada Königsberg. Especula-se que a, aproximadamente, 300 anos, os cidadãos da cidade de Königsberg mantinham o hábito de longas caminhadas em torno de sua cidade, que possuía duas ilhas cortadas pelo Rio Pregel e, na época, sete pontes as ligavam às margens e às duas ilhas entre si, conforme demonstra a imagem a seguir.

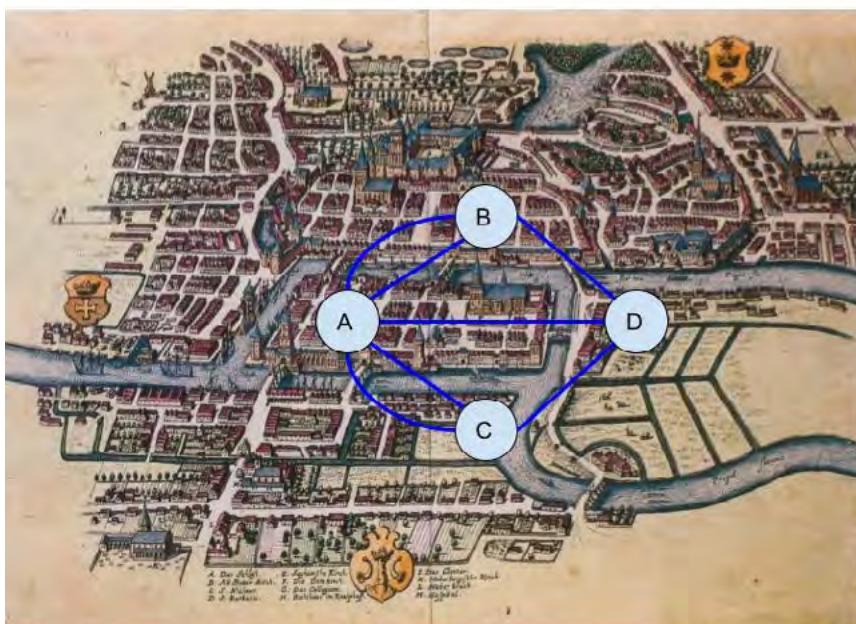


Figura 1 - Cidade Königsberg / Fonte: Google ([2021], on-line).

**Descrição da Imagem:** desenho ilustrativo do mapa com visão aérea da cidade de Königsberg do século XVII. Há construções de casas e poucas edificações, como igrejas e castelos em tamanhos maiores, que formam ruas e quarteirões. Na figura, é possível observar que a cidade é separada em quatro partes por um rio que perpassa em volta de uma das partes localizada ao centro e forma um quadrado ao centro; em volta estão as outras três partes da cidade, todas ligadas por pontes que permite a passagem de um ponto para o outro.

Nesta época, os cidadãos se desafiaram em um problema: como seria possível caminhar ao redor da cidade a pé, cruzando cada uma das sete pontes apenas uma vez e retornar ao seu ponto de partida. Este problema pode ser simples de

resolver, certo? Os moradores poderiam tentar resolver com a experimentação, ou seja, tentativa e erro; mas não foi bem assim, foi o matemático Leonhard Euler quem resolveu o problema, revelando ser impossível passar por todas as pontes uma única vez e retornar ao seu ponto de partida. Não adiantava somente ele afirmar ser impossível realizar a proeza, ele precisava comprovar a afirmação. Como matemático, Euler simplificou o problema eliminando os detalhes geométricos, por exemplo o cumprimento das pontes, formatos, distâncias, tamanhos das ilhas, obstáculos e tudo que não fazia parte do problema.



Cada vértice representa as ligações de terras e as linhas as pontes

Figura 2 - Solução de Euler / Fonte: adaptada de Google ([2021], on-line).

**Descrição da Imagem:** imagem da cidade de Königsberg com o grafo de Euler desenhado sobre ela. Os pontos A, B, C e D representando as terras, e as ligações (linhas) entre elas, as pontes, formando uma estrutura que lembra um losango. O ponto está "A" à esquerda; na parte superior está o ponto "B"; na parte inferior está o ponto "C"; e à direita está o ponto "D". As ligações são: A e B, B e A, A e C, C e A, D e A, D e B, C e D.

Lembre-se que o problema era atravessar uma única vez as pontes, não havia nenhuma relação com tempo ou distância. Veja a representação de Euler para o problema em um grafo.

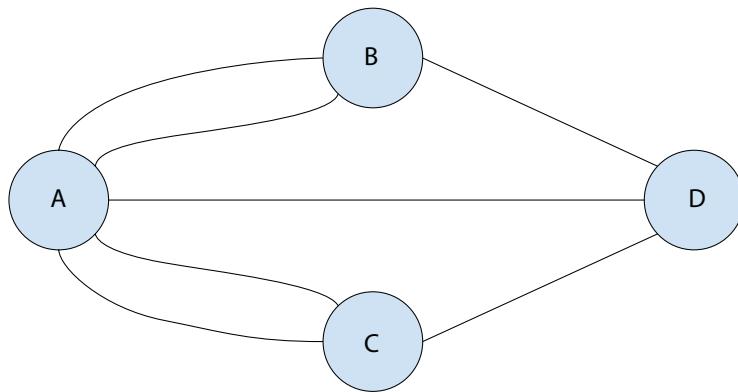


Figura 3 - Grafo de Euler / Fonte: o autor.

**Descrição da Imagem:** Grafo de Euler para o problema da cidade de Königsberg.

Um grafo é um desenho contendo pares de vértices ligados por linhas, assim, Euler idealizou que cada vértice representa um terreno (margem ou ilhas); como existiam duas margens e duas ilhas, faltava ligar os vértices. O raciocínio, então, passa a ser que a cada vértice são utilizadas duas linhas (pontos na representação), uma de entrada e uma para sair. A conclusão fica mais simples: para percorrer uma única vez as linhas, cada vértice deveria ter número par de linhas, mas o grafo das pontes de Königsberg tem, em seus vértices, ligações ímpares. Como poderíamos provar isso? Uma experimentação simples.

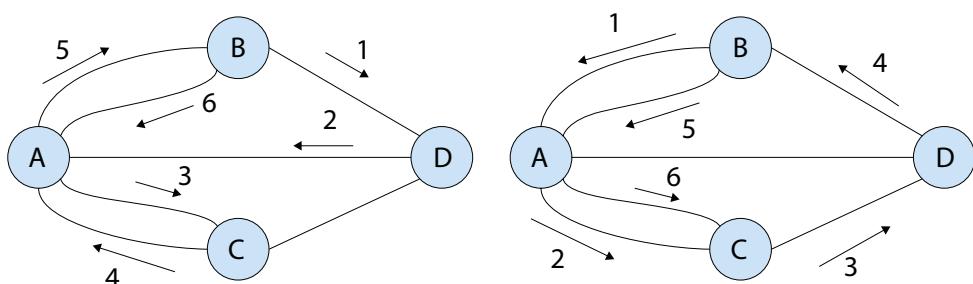


Figura 4 - Experimentação do Grafo de Euler / Fonte: os autores.

**Descrição da Imagem:** Grafo de Euler para o problema da cidade de Königsberg, no qual temos os pontos A, B, C e D representando as terras e as ligações (linhas) entre elas, as pontes, formando uma estrutura que lembra um losango. À esquerda está o ponto "A", na parte superior está o ponto "B", na parte inferior está o ponto "C" e à direita está o ponto "D". As ligações são: A e B, B e A, A e C, C e A, D e A, D e B, C e D.

Perceba que não é possível passar por todas as linhas uma única vez. Todas as tentativas possíveis de utilizar uma única vez acabam por deixar uma linha inacessível, por já ser consumidas as linhas dos vértices. Vamos extrapolar um pouco, se tivermos os mesmos números de vértices, mas com todas as ligações pares, seria possível utilizar uma ligação uma única vez?

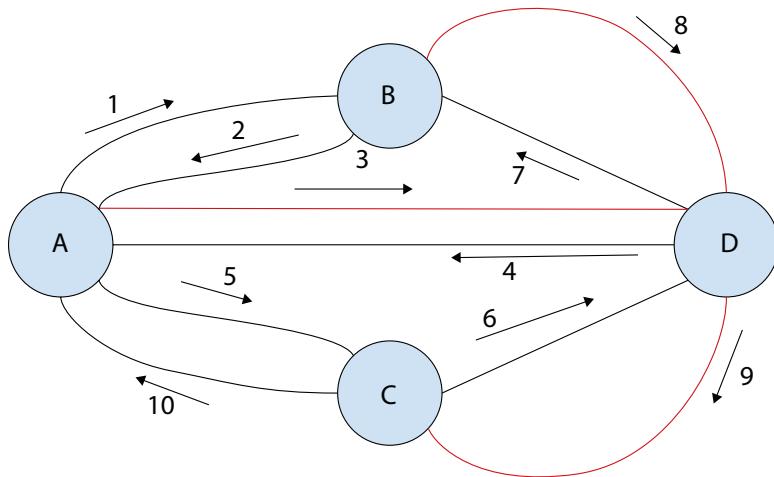


Figura 5 - Resolução proposta por Euler / Fonte: os autores.

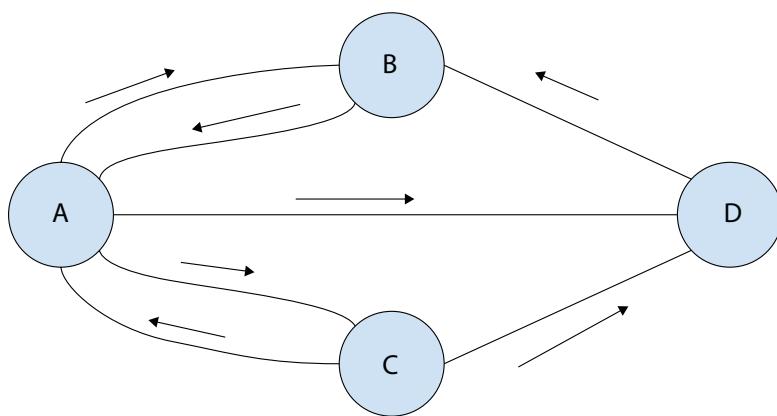
**Descrição da Imagem:** Grafo de Euler extrapolando a resolução do problema da cidade de Königsberg, no qual temos os pontos A, B, C e D representando as terras e as ligações (linhas) entre elas, as pontes, formando uma estrutura que lembra um losango. O ponto «A» está à esquerda, na parte superior está o ponto «B», na parte inferior está o ponto «C» e à direita está o ponto «D». As ligações são: A e B, B e A, A e C, C e A, D e A, D e B, C e D. A extração sugere três ligações adicionais que possibilitam a solução com êxito. As ligações adicionais foram: D e C, A e D, B e D.

A solução para o problema implica que a cidade de Königsberg tivesse mais três pontes dispostas de tal forma que as ligações entre as terras fossem pares em seu número de pontes. Somente assim seria possível atravessar todas as pontes uma única vez, retornando ao ponto de origem. Esta solução de Euler se aplica a qualquer número de vértices e se tornou a base para a Teoria de Grafos.

**Teoria de Grafos** - A teoria de grafo consiste em estudar a relação entre os objetos de um determinado conjunto empregando estruturas chamadas como grafos. A aplicabilidade da Teoria dos Grafos é ampla, alcançando áreas, tais como física, química, engenharias, psicologia, curso de ciências e engenharia da computação.

Conceituação formal: um grafo é um par  $(V, A)$ , em que  $V$  é um conjunto arbitrário e  $A$  é um subconjunto de  $V^{(2)}$ . Os elementos de  $V$  são chamados vértices

e os de A são chamados arestas. Neste texto, vamos nos restringir a grafos, em que o conjunto de vértices é finito. (FEOFILOFF; KOHAYAKAWA; WAKABAYASHI, 2005). Esta definição se aplica a solução de Euler, que possui uma representação do seu teorema contendo os elementos vértices e arestas. Vértices são os pontos representados por Euler como terra, e as arestas são as ligações entre os pontos, ou seja, as linhas, representadas como pontos. Podemos dizer que, para montar um grafo, devemos ter dois conjuntos: vértices ( $V$ ) e arestas ( $E$  do inglês edge).



$$V = \{A, B, C, D\}$$

$$E = \{(A, B), (B, A), (A, D), (A, C), (C, A), (C, D), (D, B), (D, C)\}$$

**Figura 6 - Grafos com Vértices e Arestas / Fonte: os autores.**

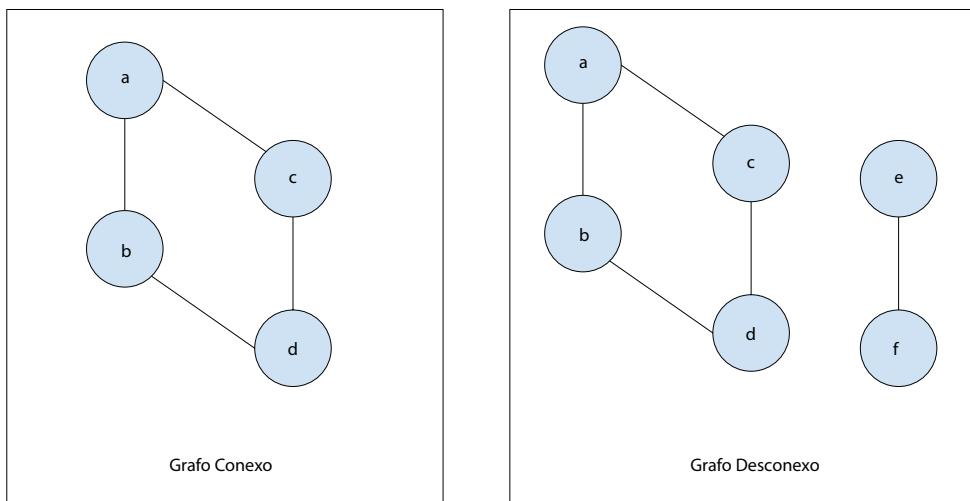
**Descrição da Imagem:** Grafo de Euler para o problema da cidade de Königsberg. Os pontos A, B, C e D representando as terras e as ligações (linhas) entre elas, as pontes, formando uma estrutura que lembra um losango. O ponto «A» está à esquerda, na parte superior está o ponto «B», na parte inferior está o ponto «C» e à direita está o ponto «D». As Arestas: A e B, B e A, A e C, C e A, D e A, D e B, C e D.

A representação do conjunto de aresta é feita em pares e pode ser orientada ou não. Orientado quando o sentido é definido, portanto, a aresta  $(A, B)$  é diferente de  $(B, A)$ . Por este motivo, na imagem, as setas representam a direção de cada aresta definida no conjunto.

Graus de um vértice são definidos pelo número de arestas que se liga a um vértice. No caso do exemplo anterior, o grau do vértice B é três porque tem três arestas ligadas a ele. Quando o grafo é orientado, pode receber uma classificação adicional como:

- Grau de emissão: arestas que saem do vértice, ex.: vértice D possui um grau de emissão (segundo o gráfico anterior).
- Grau de recepção: arestas que chegam no vértice, ex.: vértice D possui dois graus de recepção (segundo o gráfico anterior).

Existem alguns cenários em que o grafo possui vértices que estão desconectados de outros vértices e, por isso, é possível classificar o grafo que contém todos os vértices conectados entre vértices como **conexo**; e o que não possui um ou mais vértices conectados como **desconexo**.



**Figura 7 - Grafos Conexos e Desconexos / Fonte:** os autores.

**Descrição da Imagem:** a imagem à esquerda é um grafo conexo, em que todos os vértices estão conectados, no qual temos uma figura quadrilátera. Temos: o primeiro vértice superior A; abaixo B; em diagonal de B, temos o vértice D; e acima do vértice D, temos o vértice C. As arestas são formadas por A e B, B e D, D e C, C e A. A imagem à direita representa um grafo desconexo, no qual existem 2 grupos de grafos distintos que não se conectam. Primeiro, temos uma figura quadrilátera com vértice superior A; abaixo B; em diagonal de B, temos o vértice D; e acima do vértice D, temos o vértice C. As arestas são formadas por A e B, B e D, D e C, C e A. A segunda à direita do grafo quadrilátero, temos uma única ligação formada por dois vértices alinhados na vertical. O vértice superior é o E; abaixo o vértice F; e a aresta é formada por E e F.

## Aplicabilidade da Teoria de Grafos

Como já foi mencionado, a aplicabilidade dos grafos na resolução de problemas atinge diversas áreas de conhecimento. Entrando no detalhe da computação, podemos utilizar grafos para criar sistemas complexos de localização como soluções mais simples, como, por exemplo, a naveabilidade de um portal web.

Muitas empresas investem massivamente em usabilidade para as suas aplicações web de forma que fiquem mais atrativas e menos cansativas para os usuários. Uma ferramenta que é utilizada para melhorar a naveabilidade destas soluções é o sitemap. Que consiste em mapear todas as páginas e suas ligações, a fim de encontrar o uso excessivo de cliques.

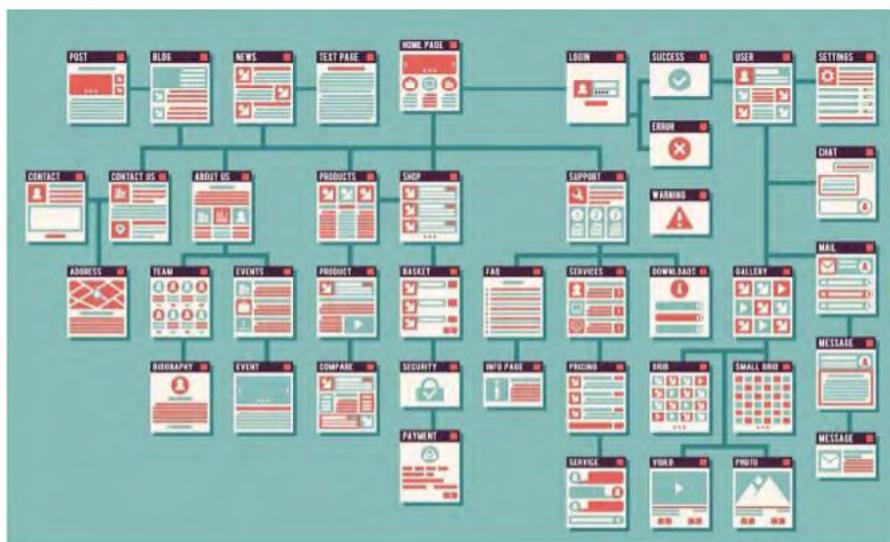


Figura 8 - Título da Imagem: Sitemap / Fonte: SEJ (2021, on-line).

**Descrição da Imagem:** desenho de um sitemap que representa as páginas de um site e suas ligações. A aplicabilidade de grafo consiste em encontrar o menor caminho para um usuário que está na home e alcançar uma determinada página.

Neste exemplo, podemos considerar que os vértices serão as páginas e as arestas o link entre elas. Por meio de grafos, podemos encontrar se existem páginas desconexas, páginas que precisam navegar por várias páginas antes de chegar ao destino. Parece muito simples? Pense o quanto um grande e-commerce perde de vendas se a ação de compras necessitar de vários cliques.

Outro cenário que possui aplicabilidade muito relevante é em análise e simulações de workflows e fluxos em bpmns.

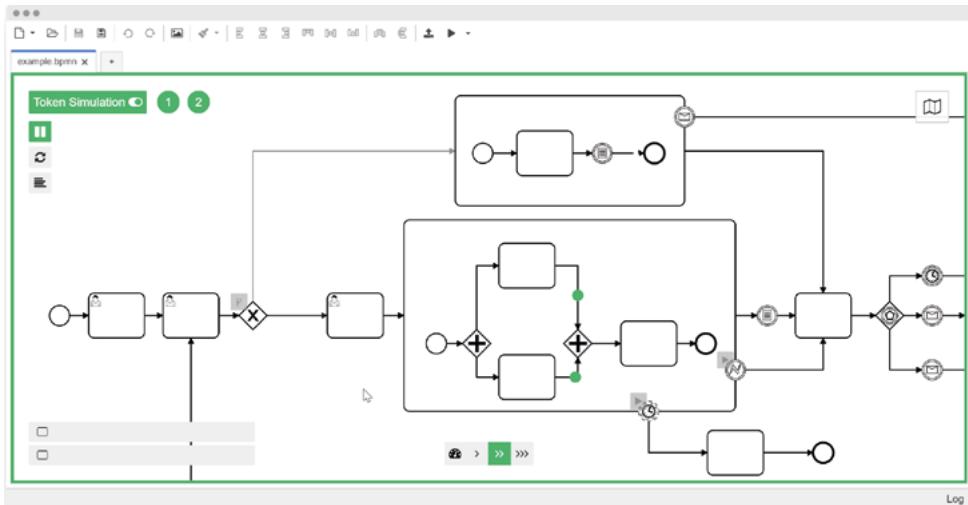


Figura 9 - Modelagem de um processo / Fonte: Github ([2021], on-line).

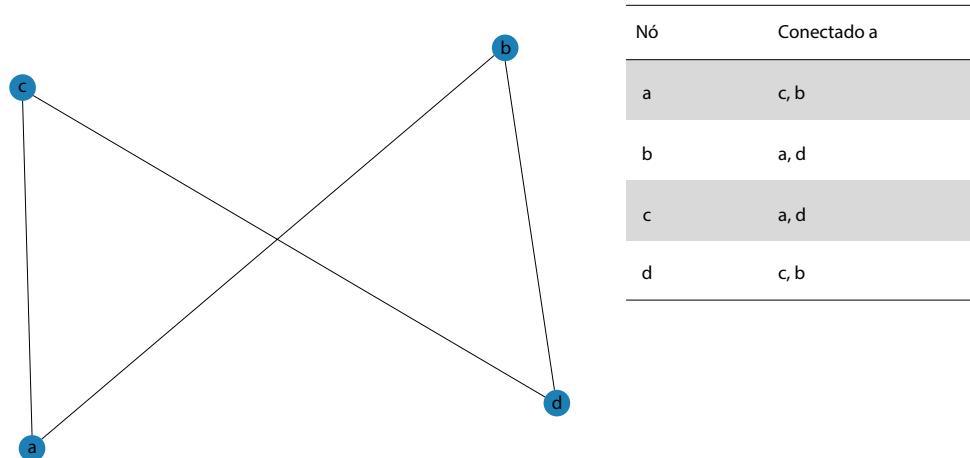
**Descrição da Imagem:** Com a aplicação do grafo, é possível aplicar o caminho mais curto para finalizar o processo. A imagem tem referência sobre a automação de processo.

Simular um fluxo de trabalho resulta em análise mais precisa para aumentar a produtividade de uma empresa – tarefas que, normalmente, são realizadas por meio da tentativa e erro ou, ainda, cálculos por meio de planilhas eletrônicas. A cada cenário é construído os cálculos e valores; uma vez que os caminhos se alternam, podem gerar erros na medida que existam mais cenários. Estes erros podem ser minimizados utilizando grafos como solução.

Outras soluções mais clássicas, como no ramo de logística de entrega, um diferencial que muitas empresas têm adotado é uma entrega cada vez mais rápida. Para isso, investem muito em soluções que possam melhorar o tempo com menor custo de deslocamento. Aplicativos de GPS que, além de montar a rota, levam em consideração a distância e o tempo para sair da origem e chegar ao destino planejado.

**Representação de Grafos em Sistemas** - A forma de representar um grafo em sistemas computacionais pode ocorrer da forma que melhor você entender. Parece bem genérico, mas lembre-se que tudo que criamos existem infinitas for-

mas de ser representadas e isso não é tão importante, mas sim o conceito. Uma casa não é, necessariamente, quatro paredes com um teto, uma casa pode ser criada com outros formatos, mantendo o conceito de uma casa. Com um grafo não é diferente, podemos representá-lo de inúmeras formas, mas precisamos manter o conceito dos vértices e arestas para que se caracterize como grafo. O comum é um conjunto de nós ligados.



**Figura 10 - Grafo não direcionado / Fonte:** os autores.

**Descrição da Imagem:** Representa do grafo não direcionado em diagrama de nós e em lista, em que o nó A está conectado em C e B; nó B conectado em A e D; nó C conectado em A e D; nó D conectado em C e B. À direita do grafo, há uma tabela definindo o vértice e suas ligações. A primeira coluna é o Vértice descrito com o cabeçalho “Nó”; a segunda coluna são as Arestas com o cabeçalho “Conectado a”. São elas: A - C, B; B - A, D; C - A, D; D - C, D.

Neste exemplo, há uma representação tanto em diagrama de nós quanto em uma lista, ambos são compreensíveis e mantêm o conceito inicial de grafo. Portanto, o ponto chave é manter o conceito em que um vértice precisa estar conectado a outro por meio de arestas.

Uma outra forma clássica de representar um grafo é por meio de matrizes, no qual podemos representar uma matriz  $n \times n$ . Entende-se uma matriz adjacente na seguinte representação.

	a	b	c	d	e
a	0	1	1	1	0
b	1	0	1	1	1
c	1	1	0	0	0
d	1	1	0	0	1
e	0	1	0	1	0

Figura 11 - Matriz Adjacente / Fonte: os autores.

**Descrição da Imagem:** Representação de uma matriz no qual os vértices de saída estão nas linhas e os vértices de destino nas colunas. As arestas são os valores 1 na matriz. As ligações são: A e B, A e C, A e D, B e A, B e C, B e D, B e E, C e A, C e B, D e A, D e B, D e E, E e B, E e D.

Na matriz adjacente que representa um grafo, o valor da célula que possui zero significa que não possui aresta. O valor que contém o valor 1 representa uma reta de ligação entre os vértices. A representação gráfica da matriz em um desenho com nós seria assim.

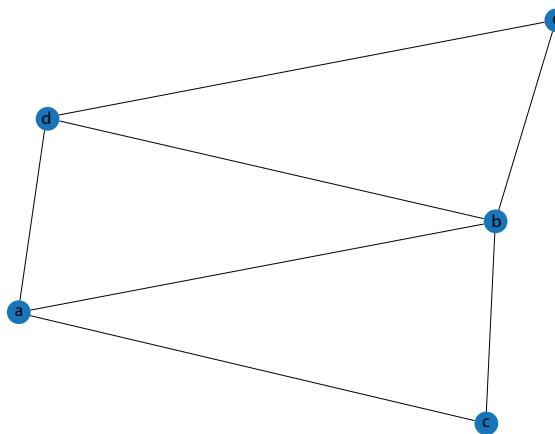


Figura 12 - Representação em nós de grafo em relação à matriz adjacente / Fonte: os autores.

**Descrição da Imagem:** um grafo que mostra a relação de nós e a matriz adjacente, em que o vértice à esquerda é o A, e está ligado em B, C e D; à direita está o vértice B e está ligado em A, C, D, E; abaixo está o vértice C, e está ligado em A, B; acima do vértice A está o vértice D, ligado em A, B, E; e à direita está o vértice E, ligado em B, D.

Um adjacente apenas simbolizando se existe (um) ou não existe (zero) pode ser chamada uma matriz de byte, ou matriz de dados binários. Outra representação é se há a necessidade de valores absolutos, entenda que nesta representação os valores podem influenciar no problema, como por exemplo a distância. Nestes casos devemos informar realmente os valores que ocorrem na matriz.

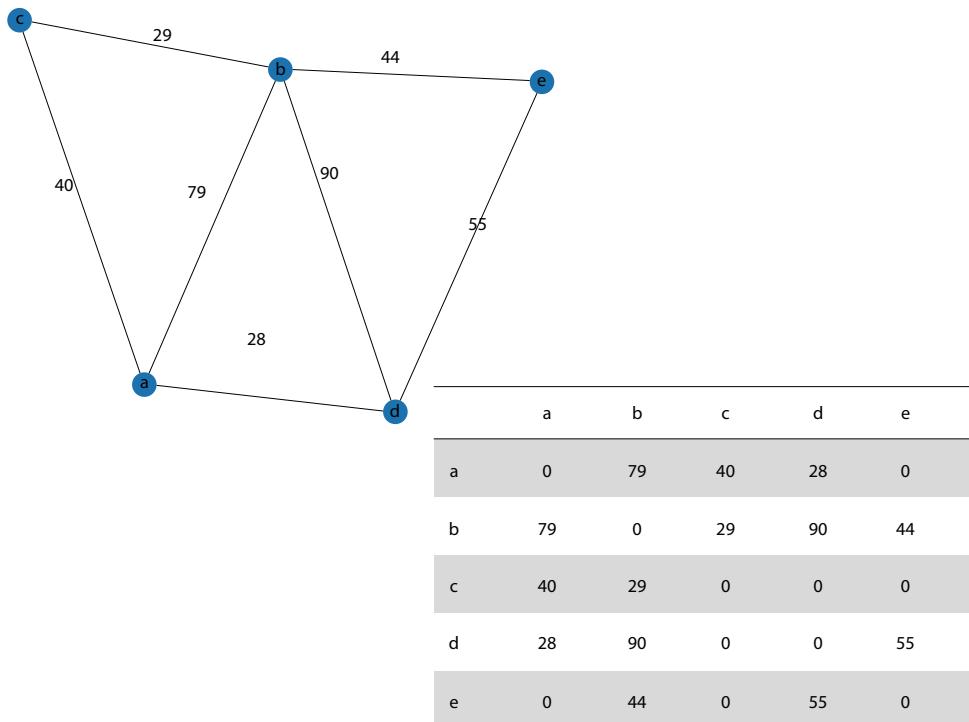


Figura 13 - Matriz adjacente com valores absolutos / Fonte: os autores.

**Descrição da Imagem:** Representação do grafo sendo criado com a matriz adjacente com valores absolutos. A figura lembra um pentágono invertido, o primeiro vértice superior à esquerda é o C; à direita e no centro o vértice B; à direita o vértice E; abaixo o vértice D; e à direita o vértice A. As arestas com seus valores são (C, B, 29), (B, C, 79), (B, D, 90), (B, E, 44), (E, D, 55), (D, A, 28) e (A, C, 40).

A matriz adjacente pode representar adequadamente um grafo, mas quando se trata de valores absolutos, é importante ter o cuidado de informar se o valor zero representa um valor ou a falta da aresta. Quando tratamos de valores absolutos, a referência faz a diferença de como interpretar os dados que estão sendo coletados. Zero pode representar um valor válido dependendo da referência do problema a ser resolvido.

## Desenvolvendo Grafos em Python

Agora, vamos aprender como implementar uma estrutura de dados em grafo em Python, vamos ver duas forma de representar a estrutura, que são: representação explícita, no qual nomeamos os vértices no grafo, e a outra é a representação implícita, nesta o identificador é um número inteiro associado automaticamente, e faz a referência ao vértice.

Primeiramente, vamos aprender como montar o grafo, vamos assumir que será fornecida a lista de vértices com as suas devidas ligações, a partir disso, iremos conectar os vértices e as arestas. Vamos trabalhar com a seguinte lista de arestas:

- A B
- B D
- B C
- C E
- C B
- D A
- E B

Para criar esse grafo em Python é bem simples, basta criar as arestas para os vértices, por exemplo, temos que dizer que possui uma aresta que vai de A para B; neste exemplo, vamos usar um dicionário de lista em uma representação explícita, em que a chave é o vértice e os valores são a lista. Veja o código abaixo que mostra como montamos esse grafo e, em seguida, vamos detalhá-lo:

```
1 grafoexplicito = { "A" : ["B"],  
2                 "B" : ["C", "D"],  
3                 "C" : ["B", "E"],  
4                 "D" : ["A"],  
5                 "E" : ["B"]}
```

- Linha 1: Cria a aresta do vértice A para o vértice B.
- Linha 2: Cria a aresta do vértice B para o vértice C e D.
- Linha 3: Cria a aresta do vértice C para o vértice B e E.
- Linha 4: Cria a aresta do vértice D para o vértice A.
- Linha 5: Cria a aresta do vértice E para o vértice B.

Veja a imagem de como esse grafo foi criado em uma representação visual:

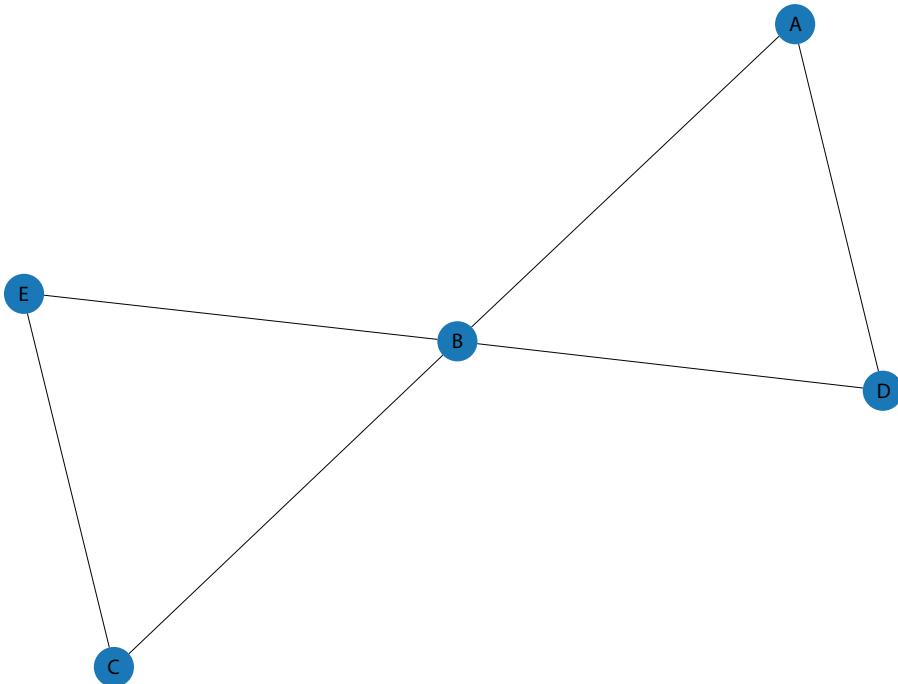


Figura 14 - Representação do grafo explícito criado em Python / Fonte: os autores.

**Descrição da Imagem:** representa o grafo em diagrama de vértice, em que, à direita superior está o vértice A; abaixo o vértice C; no centro o vértice B; à direita superior o vértice A; e abaixo o vértice D. As arestas são formadas por (E, C), (E, B), (C, B), (B, A), (B, D), (A, D).

Uma vez montado o grafo, é necessário executar operações básicas que serão utilizadas durante a manipulação desse grafo, essas operações são:

- Listar os vértices.
- Recuperar a lista de arestas.
- Verificar a existência de uma aresta.
- Adicionar uma aresta.
- Obter tamanho do grafo.
- Adicionar aresta entre dois vértices.

Veja o código de como criar cada uma dessas operações. Vamos usar a collections defaultdict para manipular o grafo e, logo em seguida, iremos detalhar essas linhas de códigos:

```
1 from collections import defaultdict
2
3 class grafo(object):
4     def __init__(self, aresta, direcionado=False):
5         self.adj = defaultdict(set)
6         self.direcionado = direcionado
7         self.adiciona_arestas(aresta)
8
9     def get_vertice(self):
10        return list(self.adj.keys())
11
12    def get_aresta(self):
13        return [(k, v) for k in self.adj.keys() for v
14               in self.adj[k]]
15
16    def adicionar_aresta(self, aresta):
17        for u, v in aresta:
18            self.adiciona_ligacao(u, v)
19
20    def adiciona_ligacao(self, u, v):
21        self.adj[u].add(v)
22        if not self.direcionado:
23            self.adj[v].add(u)
24
25    def existe_aresta(self, u, v):
26        return u in self.adj and v in self.adj[u]
27
28    def __len__(self):
29        return len(self.adj)
30
31    def __str__(self):
32        return '{}({})'.format(self.__class__.
33                               __name__, dict(self.adj))
34
35    def __getitem__(self, v):
36        return self.adj[v]
```

- Linha 3 a 7: inicializa a estrutura do grafo, passando com parâmetro se será direcionado ou não direcionado.
- Linha 9 e 10: recupera a lista de vértices.
- Linha 12 e 13: recupera a lista de arestas.
- Linha 15 a 17: adiciona arestas de acordo com parâmetro.

- Linha 19 a 22: adiciona uma ligação entre os vértices recebidos com parâmetro (u e v).
- Linha 24 e 25: verifica se possui uma aresta entre os parâmetros u e v.
- Linha 27 e 28: obtém o tamanho do grafo.
- Linha 30 e 31: formata a impressão do grafo de forma legível.
- Linha 33 e 34: obter a lista adjacências do vértice.

Agora, com esses métodos criados, podemos criar um grafo e utilizar as operações criadas nesta classe, veja o exemplo a seguir, seguido da explicação desse conteúdo:

```

1 from Grafo import grafo
2
3 aresta = [('A', 'B'), ('B', 'C'), ('C', 'B'), ('D', 'A'),
4           ('E', 'B')]
5 criagrafo = grafo(aresta, direcionado=True)
6 print(criagrafo.adj)
7
8 print(criagrafo.get_vertice())
9
10 print(criagrafo.get_aresta())
11
12 print(criagrafo.existe_aresta('A', 'B'))
13
14 print(criagrafo.existe_aresta('E', 'C'))

```

- Linha 1: importa a classe criada anteriormente.
- Linha 3: cria as arestas.
- Linha 5: inicializa o grafo direcionado com as arestas definidas na linha 3.
- Linha 6: imprime o grafo em forma de texto: defaultdict(<class 'set'>, {'A': {'B'}, 'B': {'C'}, 'C': {'B'}, 'D': {'A'}, 'E': {'B'}}).

- Linha 8: imprime os vértices do grafo em texto: ['A','B','C','D','E'].
- Linha 10: imprime as arestas do grafo em texto: [(A,B),(B,C),(C,B),(D,A),(E,B)].
- Linha 12: imprime em verdadeiro ou falso se existe uma aresta entre A e B, resultado true.
- Linha 14: imprime em verdadeiro ou falso se existe uma aresta entre E e C, resultado falso.

Dessa forma, podemos utilizar a classe grafo de forma genérica para criação e manipulação de qualquer grafo que deseja ser criado. Também podemos criar os grafos com representação implícitas, como mencionado anteriormente; neste caso, os vértices vão receber, automaticamente, um identificador inteiro; no caso anterior, adicionamos os nomes de forma manual, veja o exemplo a seguir, seguindo da explicação dessas linhas:

```
1 grafoimplicito = [ [1],
2                 [2, 3],
3                 [1, 4],
4                 [0],
5                 [1]]
```

- Linha 1: vértice 1 ligado ao vértice 0.
- Linha 2: vértice 2 e 3 ligados ao vértice 1.
- Linha 3: vértice 1 e 4 ligados ao vértice 2.
- Linha 4: vértice 0 ligado ao vértice 3.
- Linha 5: vértice 1 ligado ao vértice 4.

Veja a representação visual de como ficou esse grafo:

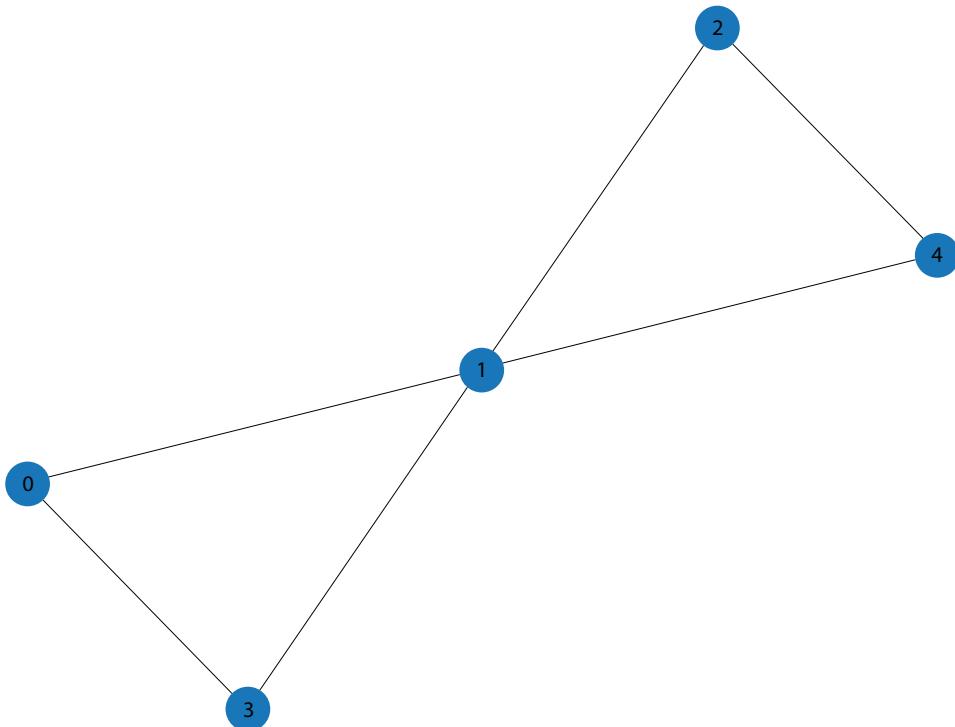


Figura 15 - Representação do grafo implícito criado em Python – vértices de valores absolutos  
Fonte: os autores.

**Descrição da Imagem:** representa o grafo em diagrama de nós, em que o nó 0 está ligado em 1,3; nó 1 ligado em 0,2,3,4; nó 2 ligado em 4,1; nó 3 está ligado em 0,1; e nó 4 está ligado em 2,1.

Neste exemplo, o vértice 0 representa o A; 1 o B; 2 o C; 3 o D; e 4 o E. Eles foram adicionados e referenciados de forma implícita, ou seja, não precisamos nomeá-los.

## Utilizando Networkx:

Quando trabalhamos com linguagens de alto nível, normalmente, possuem bibliotecas que resolvem problemas clássicos, como listas, buscas e gráficos; e também existem bibliotecas que resolvem grafos. O Python possui biblioteca que auxilia na criação, representação e solução dos grafos para auxiliar e facilitar o entendimento da construção de soluções. Uma das bibliotecas é a Networkx, que cria e mantém um grafo, além de possibilitar a criação de um gráfico de nós para o entendimento visual.

```
1 import networkx as bibliotecaNetworkx
2
3 grafo = bibliotecaNetworkx.Graph()
4
5 grafo.add_node("a")
6 grafo.add_node("b")
7 grafo.add_node("c")
8 grafo.add_node("d")
9
10
11 grafo.add_edge("a", "b")
12 grafo.add_edge("a", "c")
13 grafo.add_edge("b", "d")
14 grafo.add_edge("c", "d")
15 grafo.add_edge("c", "b")
16
17 numeroVertices = grafo.number_of_nodes()
18 numeroArestas = grafo.number_of_edges()
19 print("Visualizando as ligações de um vértice", grafo.adj["a"])
20 print("Quantidade de arestas por vértice", grafo.degree)
21 print("Quantidade de vértices:", numeroVertices)
22 print("Quantidade de arestas:", numeroArestas)
23 print("Vértices do grafo:", grafo.nodes())
24 print("Arestas do", grafo.edges)
25
```

Perceba que, no código, importamos a biblioteca para conseguirmos utilizar as funcionalidades de gerenciamento do grafo. Vamos detalhar o código:

- Linha 1: importa a biblioteca network para a definição bibliotecaNetWorkx.
- Linha 3: instala o objeto como um grafo para utilizar os seus métodos.
- Linha 5 a 8: adiciona os vértices ‘a’ até ‘d’. Veja que, aqui, utiliza-se “node” (nó) para representar os vértices.
- Linha 11 a 15: adiciona as arestas para que cada ligação seja formada.
- Linha 17 a 18: recupera a quantidade de vértices e arestas, respectivamente.
- Linha 19: imprime os vértices ligados ao vértice informado, caso não seja informado (grafo.adj), será impresso os vértices com seus respectivos dados.
- Linha 20: imprime a quantidade de arestas por vértices - [(‘a’, 2), (‘b’, 3), (‘c’, 3), (‘d’, 2)].
- Linha 21 a 22: imprime a quantidade de vértices e arestas existentes no grafo.
- Linha 23 a 24: imprime os vértices e as arestas do grafo.

A biblioteca networkx possui funcionalidades completas para criar grafos, vértices, arestas e, ainda, podemos remover cada um dos elementos, acessar diretamente e exibir um elemento gráfico. Veja o exemplo:

```
1 import matplotlib.pyplot as grafico
2 import networkx as bibliotecaNetworkx
3
4 grafo = bibliotecaNetworkx.Graph()
5
6
7 grafo.add_node("a")
8 grafo.add_node("b")
9 grafo.add_node("c")
10 grafo.add_node("d")
11 grafo.add_node("e")
12 grafo.add_node("f")
13
14 grafo.add_edge("a", "b")
15 grafo.add_edge("b", "c")
16 grafo.add_edge("c", "d")
17 grafo.add_edge("d", "e")
18 grafo.add_edge("e", "f")
19 grafo.add_edge("f", "a")
20
21 posicionamento = bibliotecaNetworkx.circular_layout(grafo)
22 bibliotecaNetworkx.draw_networkx_edge_labels(grafo, posicionamento,
23     edge_labels={('a', 'b'): 'aresta a-b',
24                 ('b', 'c'): 'aresta b-c',
25                 ('c', 'd'): 'aresta c-d',
26                 ('d', 'e'): 'aresta d-e',
27                 ('e', 'f'): 'aresta e-f',
28                 ('f', 'a'): 'aresta f-a'},
29     ),
30     font_color='green')
31 bibliotecaNetworkx.draw(grafo, with_labels=True,
32     pos=posicionamento,
33     node_color='red',
34     edge_color="blue")
35 grafico.show()
36
```

Neste exemplo, além de acrescentar os vértices e arestas, são utilizadas propriedades gráficas, tais como cor, layout e descrição das ligações. Todas estas propriedades podem ser utilizadas para melhor customizar ou, caso não sejam informadas, serão criadas aleatoriamente pela biblioteca (com a exceção da descrição).

- Linha 1: importação da biblioteca que irá gerar o desenho.
- Linha 2 a 19: idêntico ao exemplo anterior, que realiza a importação da biblioteca, cria o objeto grafo, adiciona os vértices e as arestas.
- Linha 21: definido o layout que o grafo terá, é importante que esta definição seja após a inclusão dos vértices e arestas para que o layout possa ser calculado.
- Linha 22 a 30: definido as informações que serão exibidas em cada linha do grafo, assim como a cor do texto. Esta configuração é opcional para quando existir a necessidade de exibir algum dado.
- Linha 31 a 34: desenho do grafo para inclusão do layout, exibição dos valores em cada label e cores dos vértices e arestas.
- Linha 35: exibe o grafo como imagem, quando usado em uma IDE, como Pycharm ou Júpiter, a imagem é exibida automaticamente.

The screenshot shows a PyCharm interface with a code editor and a results viewer. The code in the editor is:

```

import matplotlib.pyplot as grafico
import networkx as bibliotecaNetworkx

grafo = bibliotecaNetworkx.Graph()

grafo.add_node("a")
grafo.add_node("b")
grafo.add_node("c")
grafo.add_node("d")
grafo.add_node("e")
grafo.add_node("f")

grafo.add_edge("a", "b")
grafo.add_edge("b", "c")
grafo.add_edge("c", "d")
grafo.add_edge("d", "e")
grafo.add_edge("e", "f")
grafo.add_edge("f", "a")

posicionamento = bibliotecaNetworkx.circular_layout(grafo)
bibliotecaNetworkx.draw_networkx_edge_labels(grafo, pos=posicionamento)

```

The results viewer displays a circular graph with six nodes labeled a through f. The edges are labeled with their respective names: aresta a-f, aresta b-c, aresta c-d, aresta d-e, aresta e-f, and aresta f-a. The nodes are red circles, and the edges are blue lines.

**Figura 16 - Print da PyCharm / Fonte: os autores.**

**Descrição da Imagem:** a imagem apresenta a tela do PyCharm, e apresenta o grafo criado segundo o código descrito anteriormente. A imagem mostra a figura 17, que foi gerada no Pycharm, e está à direita do código (anteriormente já descrito). Essa formatação fica a cargo da sua configuração na IDE.

Veja que o gráfico é construído conforme as propriedades que você define do Python, possibilitando um melhor entendimento, geração de relatório, entre outros benefícios. Outro benefício que é essencial quando se cria / desenha um grafo é a possibilidade de criar grafos direcionados. Existem cenários em que

esta característica é crucial para a resolução correta, e para criar o grafo com esta característica, precisa-se instanciar o objeto *DiGraph*, que, utilizando o exemplo anterior, substituirá a linha 4: `grafo = bibliotecaNetworkx.DiGraph()`. O resultado do grafo fica assim:

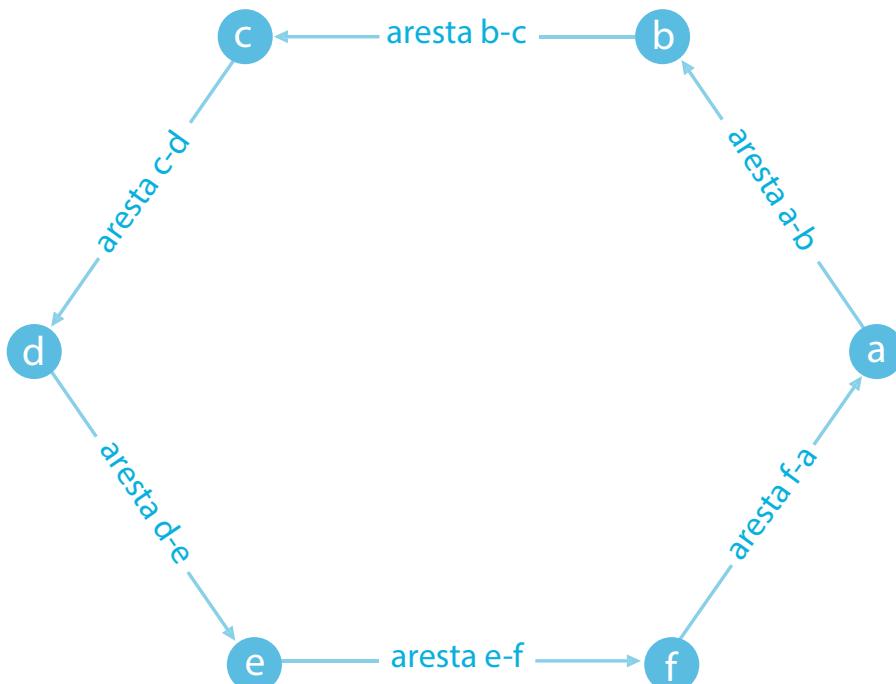


Figura 17 - Grafo com aresta direcionado no PyCharm / Fonte: os autores.

**Descrição da Imagem:** representação do grafo com arestas direcionadas quando instanciando o DiGraph e utilizando o PyCharm, no qual as arestas fazem as ligações em uma única direção, representadas pelas setas. A figura representa um hexágono com os vértices iniciando no centro à esquerda D; abaixo E; à direita F; centro superior A; superior B; esquerda C. As arestas com seus respectivos labels são: (D, E, aresta d-e), (E, F, aresta e-f), (F, A, aresta f-a), (A, B, aresta a-b), (B, C, aresta b-c), (C, D, aresta c-d).

Veja que cada aresta possui uma seta informando a direção da ligação, porque, no exemplo, apenas é realizada uma única direção. Caso necessite informar que há uma aresta com duas direções, deverá informar as duas direções, exemplo:

- grafo.add\_edge("a","b")
- grafo.add\_edge("b","a")

Neste exemplo, a ligação entre 'a' e 'b' terá uma ligação de duas direções. A biblioteca networkx possui outras funcionalidades que podem auxiliar na criação e análise de problemas relacionados a grafos. A documentação é bem detalhada e pode ser encontrada em <https://networkx.org/>.



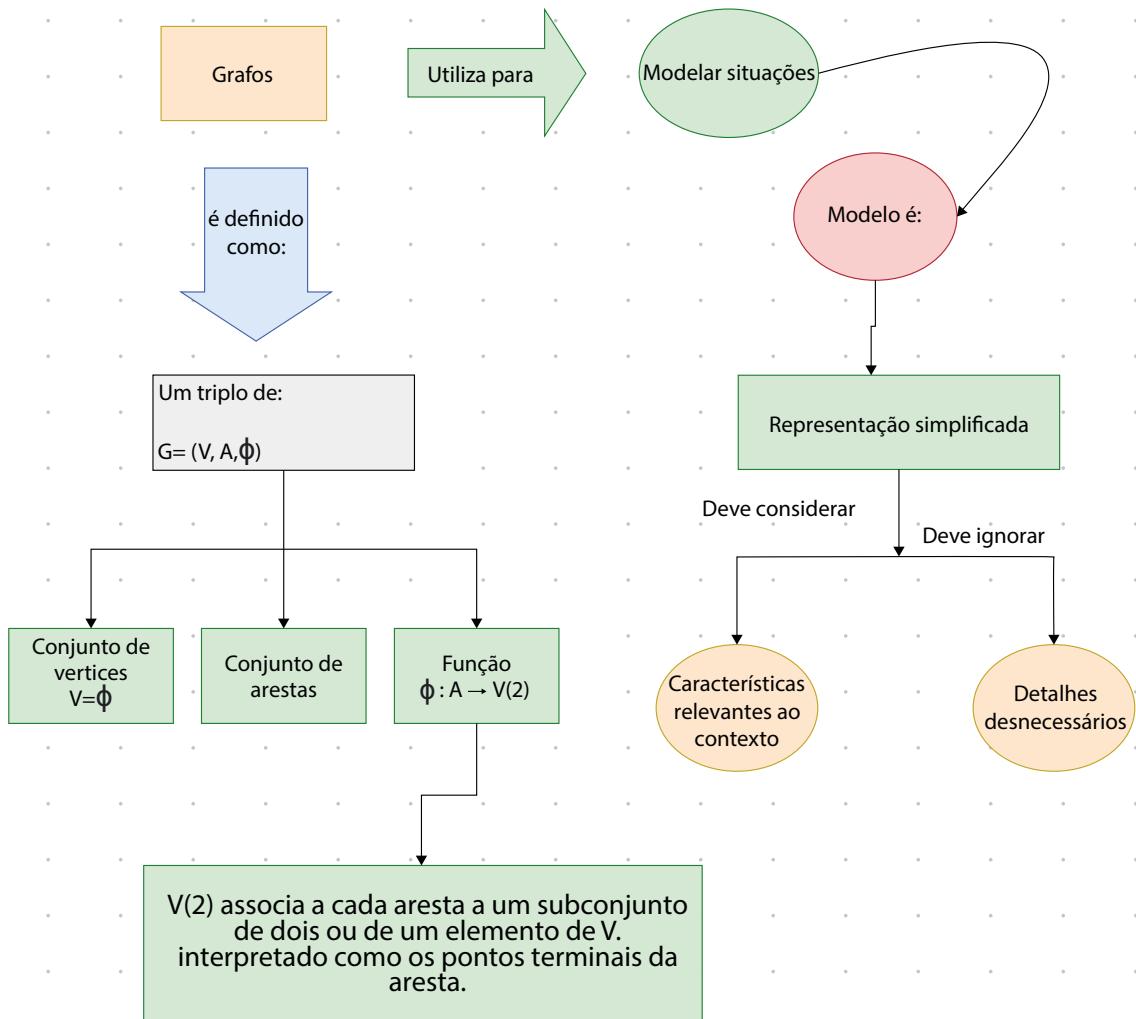
Nesta unidade, abordamos a teoria de grafos, no qual permite uma maior flexibilidade para ligar os dados de acordo com o seu contexto. Lembre-se que podemos fazer a ligação dos vértices de acordo com a necessidade do problema. Pense e reflita quando devemos utilizar a estrutura de grafos em vez de uma estrutura de lista dinâmica.





## OLHAR CONCEITUAL

Durante esta unidade, estudamos a teoria de grafos, observe que eles modelam situações reais e devem sempre considerar as características relevantes ao contexto e se basear nos vértices, arestas e função. Olhe o infográfico a seguir para entender o principal contexto do uso de grafos.

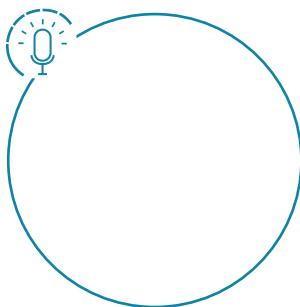




### EXPLORANDO IDEIAS

Se você já percebeu, os grafos quebram todas as regras das demais estruturas de dados que estudamos até agora. Contudo, existe uma característica que todo grafo possui: ele precisa sempre ter, pelo menos, um nó para ser considerado um "grafo". Assim como a estrutura de árvores, porém, não necessita que esse nó seja considerado raiz como as árvores.

Então, como estudamos, não existem regras em relação à forma que um nó se conecta a outro nó em um grafo. As arestas se conectam a nós de qualquer forma possível para atender a necessidade do problema, e lembre-se que a navegação dos grafos é bidirecional quando não dirigidos, e direcional quando dirigidos. Preste atenção! Os grafos estão ao nosso redor, são as pessoas que não conseguem enxergar como eles são. Quando você navega na internet, você está literalmente em um grafo. Assim, quando clicamos nos link's ou clicamos no botão voltar, navegamos entre as URLs, estamos navegando por um grafo.

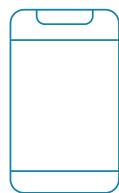


Vimos a teoria de grafos e o quanto ela é importante para resolver problemas do nosso dia a dia. Essa estrutura de dados é muito utilizada para tomada de decisão por sistemas de forma automatizada ou mostrar para o usuário uma ligação entre vários pontos. Neste Podcast, vamos falar mais das estruturas de dados em grafo, a sua aplicabilidade no uso do dia a dia do desenvolvimento e nas tomadas de decisão pelos sistemas. Acesse o QR Code para conferir essa conversa.



### NOVAS DESCOBERTAS

Você conhece o jogo Sudoku? Você sabia que pode resolver o jogo usando a teoria de grafos? Enumerarmos as células com 81 números e imaginarmos que cada célula é um vértice, dois vértices estão conectados se: eles estão na mesma coluna, fileira ou caixa (quadrado). Assim, temos o Sudoku como um grafo. Você pode jogar ele de forma online no QR Code e ler sobre como resolver o sudoku com a teoria de grafos, respectivamente.



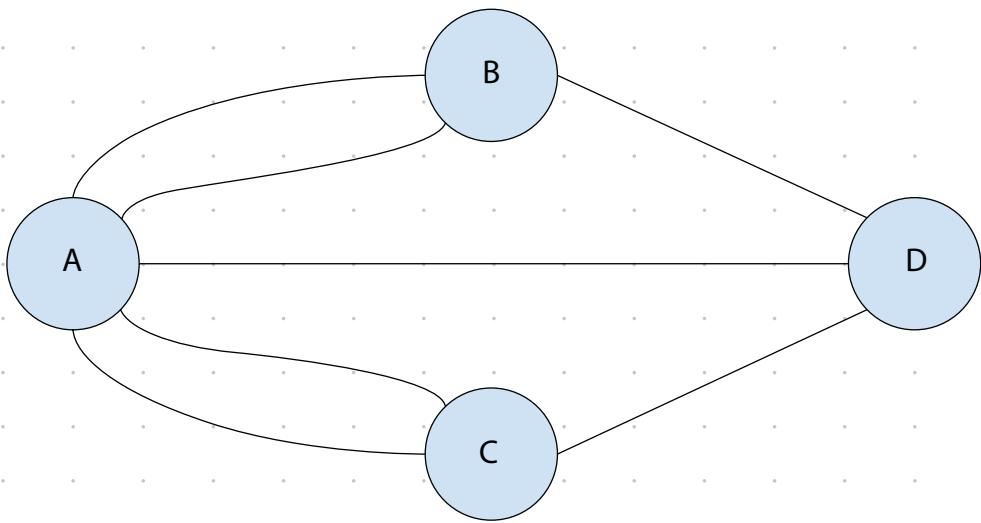
Como vimos nesta unidade, é importante entender como a teoria de grafos está inserida nos sistemas que utilizamos no dia a dia. Com base nos seus estudos, acesse o waze, trace uma rota e, depois, adicione um ponto de parada, tente entender o algoritmo que o waze utilizou para traçar a sua rota de acordo com as configurações.



1. Qual estrutura de dados abaixo utiliza vértices e arestas interligadas?

- a) Lista indexada.
- b) Grafo.
- c) Fila.
- d) Fila duplamente encadeada.
- e) Pilha:

2. Qual o tipo de grafo a imagem representa?

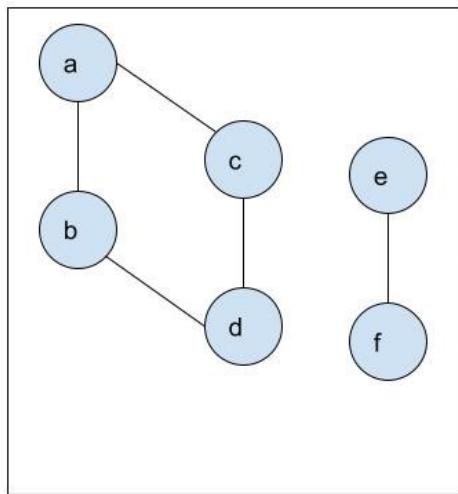


**Descrição da Imagem:** representação do grafo no qual temos os pontos A, B, C e D representando as terras, e as ligações (linhas) entre elas as pontes. À esquerda está o ponto "A"; na parte superior está o ponto "B"; na parte inferior está o ponto "C"; e à direita está o ponto "D". As ligações são: A e B, B e A, A e C, C e A, D e A, D e B, C e D.

- a) Grafo bipartido.
- b) Grafo euleriano.
- c) Grafo hamiltoniano.
- d) Grafo planar.
- e) Grafo direcionado



3. Os Grafos são formados por?
  - a) Vértices, arestas e função.
  - b) Somente vértices.
  - c) Somente arestas.
  - d) Somente função.
  - e) Somente nós.
  
4. Qual o tipo de grafo a imagem representa?

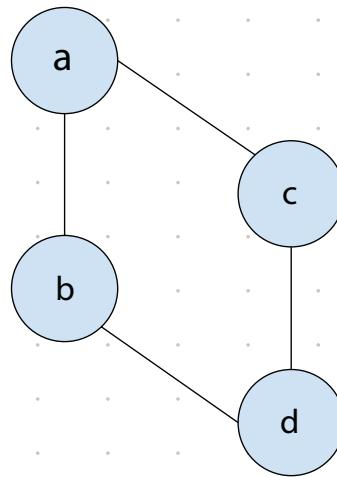


**Descrição da Imagem:** a imagem representa um grafo, no qual existem 2 grupos de grafos distintos que não se conectam. Primeiro, temos uma figura quadrilátera com vértice superior A; abaixo B; em diagonal de B, temos o vértice D; e acima do vértice D, temos o vértice C. As arestas são formadas por A e B, B e D, D e C, C e A. À direita do grafo quadrilátero, temos uma única ligação formada por dois vértices alinhados na vertical. O vértice superior é o E, abaixo o vértice F; a aresta é formada por E e F.

- a) Grafo bipartido.
- b) Grafo euleriano.
- c) Grafo desconexo.
- d) Grafo planar.
- e) Grafo direcionado



5. Qual o tipo de grafo a imagem representa?



**Descrição da Imagem:** a imagem representa um grafo em que todos os vértices estão conectados, no qual temos uma figura quadrilátera. O primeiro vértice superior A; abaixo B; em diagonal de B, temos o vértice D; e acima do vértice D, temos o vértice C. As arestas são formadas por A e B, B e D, D e C, C e A.

- a) Grafo bipartido.
- b) Grafo euleriano.
- c) Grafo desconexo.
- d) Grafo conexo.
- e) Grafo direcionado

# 5

# Busca em grafos

Esp. Edson Orivaldo Lessa Junior

Esp. Rafael Orivaldo Lessa

## OPORTUNIDADES DE APRENDIZAGEM

Nesta unidade, vamos nos aprofundar ainda mais nas estruturas de dados em grafos estudando como realizar buscas. Muitos problemas podem ser resolvidos por meio de grafos, nos quais a solução para o problema requer que realizemos uma busca pelo grafo. De uma forma simplificada, a busca em grafo sempre começa por um vértice qualquer, e navegamos no grafo de formas diferentes, dependendo do tipo de busca. Por exemplo, para alguns tipos de problemas, a solução reside no caminho percorrido e não em um nó alvo específico.

O primeiro passo é armazenar os dados em formato de grafo e seus relacionamentos, porém, após os dados estarem estruturados, é necessário realizar uma busca nos relacionamentos. Existem diversos algoritmos que podem ser utilizados ou adaptados para solucionar um problema específico, como o caminho mais curto entre o ponto A e B ou, ainda, o relacionamento entre duas pessoas, a fim de detectar fraudes em uma transação financeira.

Atualmente, com o uso massivo da internet, as fraudes tem aumentado significativamente nas transações eletrônicas; as instituições financeiras têm apostado na teoria de grafos para auxiliar nesse combate às fraudes relacionadas ao sistema financeiro e transações on-line, estruturando um relacionamento entre pessoas e empresas, como relações societárias, parentescos, endereços, dentre outros e utilizando os algoritmos específicos para resolver o problema, como, por exemplo, o caminho com menor custo. Você sabe escolher o algoritmo de busca correto para auxiliar na correta resolução do problema?

O uso de grafos e os tipos de buscas são comuns no uso de inteligência artificial e machine learning, principalmente para análises preditivas e prescritivas. O uso desse tipo de tecnologia e algoritmos tem muitos reflexos no nosso dia a dia, modifica a forma que compramos produtos e interagimos com mundo, algumas das atividades que machine learning e os diferentes tipos de busca em grafos podem resolver são: detecção de fraudes, pesquisa na web, mídia programática, análise de sentimento baseada em imagem, score de crédito, previsão de falhas, prescrição de modelos de especificação, detecção de invasão, reconhecimento de padrões e imagens, e filtragem de mensagens.

Com o advento do uso massivo da internet, também aumentou, significativamente, o número de fraudes em pagamentos on-line e transações financeiras, com isso as empresas têm investido em sistema que auxiliam no combate à fraude e prevenção de ataques cibernéticos para evitar perdas financeiras, esse investimento é para ajudar na prevenção de fraudes e gerar margens mais altas na venda e captação de leads para as empresas que trabalham com vendas on-line ou instituições bancárias.

Os algoritmos de busca de profundidade da teoria de grafo estão diretamente relacionados às essas necessidades e são poderosos para auxiliar na resolução do problema e no aprendizado das máquinas.

A busca em estrutura de grafos mostra como podemos percorrer o grafo para resolver determinados problemas. Esse tipo de solução de busca nos ajuda a resol-

ver vários problemas do dia a dia. Você já usou o LinkedIn? Faça uma busca pelo nome de uma pessoa que não tem na sua rede; as pessoas são os nós e as arestas são relações das pessoas, considerando que você é a raiz da busca. A pesquisa vai retornar todas as pessoas encontradas e qual a distância de conexão da raiz. Faça um grafo que representa essa busca e mostre as suas conexões.

Nesta unidade, estudaremos a busca em grafos que são a busca em profundidade e a busca em largura. Essas técnicas são similares, mudando apenas a forma como os vetores são armazenados. Os algoritmos são a base para quase todos os demais métodos de busca em grafos. O algoritmo de Dijkstra é uma busca em largura modificada para uma solução específica, que encontra o menor caminho entre dois vértices para grafos.

O conhecimento dessas técnicas é essencial e importante porque, se um problema for modelado em grafo, há diversos algoritmos de buscas modificados que podem resolver o problema, porém eles usam de base os dois principais. Convido você, neste momento, a registrar, em seu diário de bordo, a vantagem de usar o algoritmo de Dijkstra e quais problemas ele resolve.

## DIÁRIO DE BORDO

**Busca em profundidade** - Durante a última unidade, entendemos que um grafo é uma estrutura formada por vértices e arestas criadas para melhor representar os caminhos que devemos seguir. O grafo também é utilizado em alguns tipos de busca, visando obter uma melhor eficiência na resolução do problema.

Em geral, uma busca inicia em um vértice com o objetivo de alcançar o vértice alvo e, neste processo, precisamos percorrer uma sequência de arestas e vértices de forma ordenada. Seria similar a você utilizar um transporte terrestre coletivo (ônibus ou trem), no qual você entra no transporte no ponto A e, no caminho para o seu destino, acaba fazendo algumas paradas. As paradas seriam os vértices sequenciados que precisamos atravessar para alcançar o alvo, e a rota (caminho que o transporte sempre faz) são as arestas. Em outros momentos, o problema não está no alvo que queremos alcançar, mas na rota, o caminho percorrido é a solução do problema.

A primeira busca a ser estudada é a Busca em Profundidade (Depth-first search ou DFS), que consiste em seguir todas as arestas e, assim, irá alcançar todos os vértices. A ideia é que, a partir de um vértice alcançado, será “explorado” todos os vértices a ele ligado. Neste sentido, quando se encontra o vértice A, será percorrido a primeira aresta e encontra-se um novo vértice A1. Uma vez descoberto, busca-se o primeiro vértice de A1. Portanto, a busca de profundidade irá se aprofundar até o último vértice (quando não há arestas novas) e passa para a próxima aresta do nível anterior.

Segundo Donadelli (2010), a busca em profundidade consiste em encontrar todos os vértices alcançáveis a partir do vértice inicial. Nem sempre serão alcançáveis quando o grafo foi dirigível. Uma analogia similar é um labirinto, no qual temos uma entrada e precisamos alcançar um local. Intuitivamente, quando tentamos resolver um labirinto, percorremos um caminho até que não exista mais saídas. Quando isso acontece, retornamos para seguir outro caminho até encontrar a saída, um beco sem saída ou um caminho já visitado.

Para entender melhor esta mecânica, vamos supor que tenhamos um grafo com sete vértices, de ‘a’ até ‘g’. Estes vértices possuem arestas ligando-os de forma direcional e precisaremos fazer uma busca em profundidade. Neste momento, não iremos nos preocupar com o que estamos buscando, mas sim realizar a busca completa.

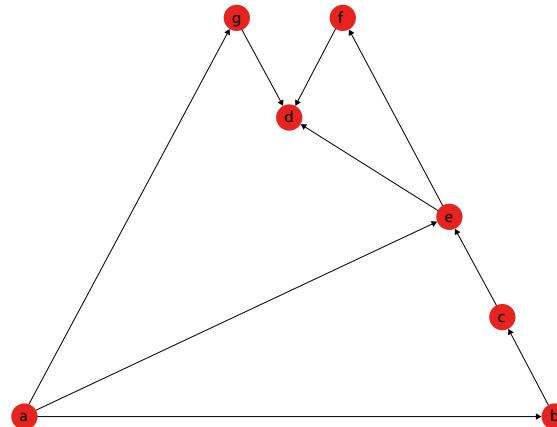


Figura 1 - Grafo com 7 vértices / Fonte: os autores.

**Descrição da Imagem:** Grafo com 7 vértices que contém as seguintes ligações: ('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd').

Ilustrando o exemplo, vamos tomar como início o vértice 'a', assim todos os vértices serão possíveis de atingir, uma vez que o grafo é direcional. Saindo do vértice 'a', teremos de escolher entre três opções de arestas para alcançar o próximo vértice. Inicialmente, vamos optar pelo vértice (a, b), que resultaria no seguinte grafo.

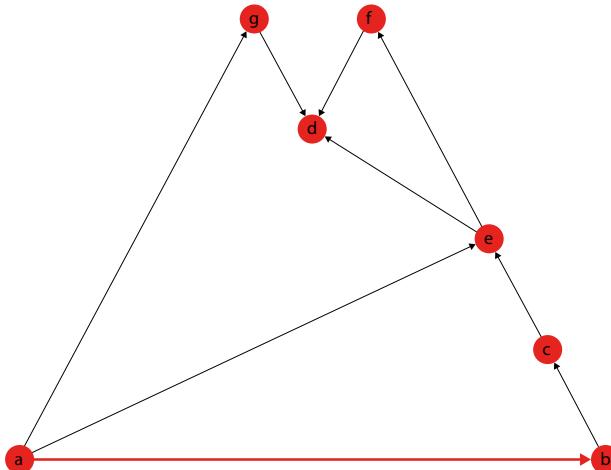


Figura 2 - Busca em profundidade, etapa 1 / Fonte: os autores.

**Descrição da Imagem:** Grafo com 7 vértices, identificadas de "a" a "g", com as seguintes ligações: ('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd'). Destaque na aresta (a, b) grifada em vermelho.

Uma vez que foi alcançado o vértice ‘b’, a busca entra no nível abaixo e continua para próxima aresta até encontrar um vértice que não possui sequência.

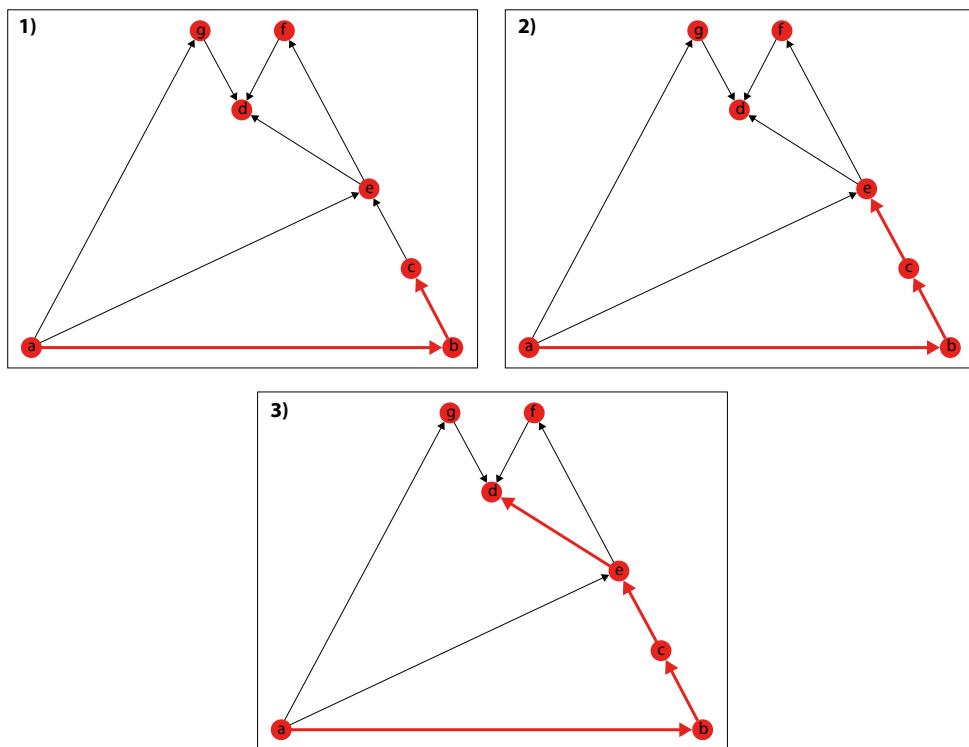


Figura 3 - Busca em profundidade, etapas 2, 3 e 4 / Fonte: os autores.

**Descrição da Imagem:** três imagens com grafo com 7 vértices que contém as seguintes ligações: ('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd'). Cada uma representando uma etapa. A primeira imagem destaca as arestas (a, b) e (b, c). A segunda imagem destaca as arestas (a, b), (b, c) e (c, e). A terceira imagem destaca as arestas (a, b), (b, c), (c, e) e (e, d).

Neste exemplo, o vértice mais profundo nesta primeira interação é o ‘d’, porém, não paramos neste vértice. Uma vez que alcançado um vértice que não possui saída, devemos voltar até o vértice que possui outra aresta para “caminhar”. Neste exemplo, vamos retornar até o vértice ‘e’ para que possamos testar outro caminho.

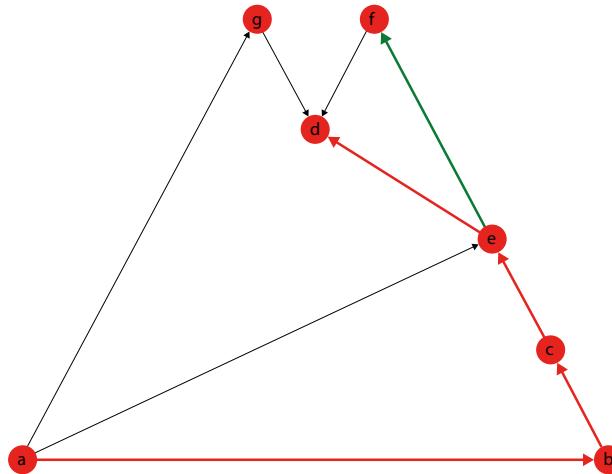


Figura 4 - Busca em profundidade, etapa 5 / Fonte: os autores.

**Descrição da Imagem:** Grafo com 7 vértices que contém as seguintes ligações: ('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd'). Destaque para as arestas (a, b), (b, c), (c, e), (e, d) e (e, f).

Nesta iteração, alcançamos o vértice 'f', e poderíamos continuar, pois ele possui uma aresta de saída (f, d). Contudo, o vértice 'd' já é conhecido e por este motivo, não precisa ser alcançado. Voltamos, então, ao cenário anterior, no qual não temos mais arestas e, com isso, devemos retornar ao vértice que possui arestas com vértices não conhecidos.

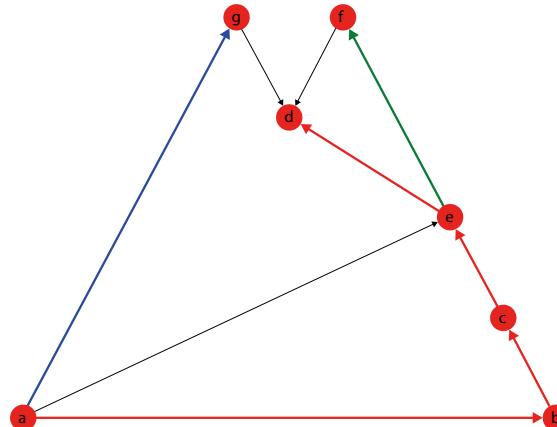


Figura 5 - Busca em profundidade, etapa 6 / Fonte: os autores.

**Descrição da Imagem:** Grafo com 7 vértices que contém as seguintes ligações: ('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd'). Destaque para as arestas (a, b), (b, c), (c, e), (e, d), (e, f), (a, g).

Veja que, nesta última etapa, alcançamos todos os vértices, pois retornamos ao vértice 'a' para descobrir o vértice 'g'. Algumas arestas não foram consideradas: (a, e), (f, d) e (g, d), pois os vértices de destino já eram conhecidos. Veja que, para cada vértice de inicial, poderíamos gerar outros resultados.

Como seria a construção deste programa usando Python? A busca em profundidade possui algumas alternativas diferentes que respeitam o algoritmo base. Em resumo, a busca é feita de vértice a vértice, utilizando métodos recursivos para melhor aproveitar os recursos. Para exemplificar, optamos por construir o algoritmo associado com a biblioteca networkx para que você possa entender visualmente como está ocorrendo o caminho no grafo. Dividimos o programa em partes para melhor entender a dinâmica.

```
1 import matplotlib.pyplot as plt
2 import networkx as nx
3 from matplotlib.backends.backend_pdf import PdfPages
4
5
6 def desenharGrafo(colors, weights):
7     global contadorFigura
8     nx.draw(G, with_labels=True,
9             pos=posicionamento,
10            edge_color=colors,
11            width=list(weights),
12            node_color='red')
13     pdf.savefig()
14     nomeImagem = "grafo" + str(contadorFigura) + ".jpg"
15     contadorFigura += 1
16     plt.savefig(nomeImagem)
17     plt.show()
18
19
20 def desenhoInicialGrafo():
21     global weights, posicionamento
22     for n1, n2 in arestas:
23         G.add_edge(n1, n2, label=(n1 + ">" + n2), color="black", weight=1)
24
25     colors, weights = definirAtributos()
26     posicionamento = nx.planar_layout(G)
27     desenharGrafo(colors, weights)
28
29
30 def definirAtributos():
31     colors = nx.get_edge_attributes(G, 'color').values()
```

```
32     weights = nx.get_edge_attributes(G, 'weight').values()
33     return colors, weights
34
35
36 def atualizarDesenho(v1, v2):
37     global corRedesenho, contadorCor, controleCor
38     controleCor = True
39     G.get_edge_data(v1, v2)["color"] = corRedesenho[contadorCor]
40     G.get_edge_data(v1, v2)["weight"] = 4
41     colors, weights = definirAtributos()
42     desenharGrafo(colors, weights)
43
44
```

Nesta primeira parte, estão os métodos criados para desenhar o grafo. É importante destacar que estas linhas apenas irão desenhar o grafo e salvar as imagens em um PDF que irá incluir todas as imagens geradas. Contudo, vamos entender linha a linha cada instrução:

- Linhas 1 a 3: importação das bibliotecas que irão montar o grafo, desenhar e gerar um PDF com os desenhos do grafo por fases.
- Linhas 6 a 17: definição do método que irá desenhar o grafo com cor e espessura parametrizada. Uma vez realizada a montagem, será salvo uma imagem por página em um arquivo PDF e um arquivo de imagem para o grafo.
- Linhas 20 a 27: composição inicial do grafo de exemplo, criado com informações de ligação entre os vértices.
- Linhas 30 a 33: define os atributos de cor e espessura para as arestas.
- Linha 36: método responsável por atualizar as arestas para gerar o grafo.
- Linhas 37 e 38: variáveis globais que controlam a cor do desenho.
- Linha 39: atualiza a propriedade da aresta no desenho em conformidade com o vetor de cor.
- Linha 40: atualiza a propriedade da aresta no desenho em conformidade com a espessura.
- Linhas 41 e 42: coletam os valores dos atributos de cor e espessura para serem enviados ao método de *desenhar Grafo*, permitindo que cada aresta seja atualizada em conformidade com a busca em profundidade.

A parte do código a seguir é responsável por inicializar as variáveis de controle, parametrização do grafo e também definir o método da busca em profundidade, que é responsável em adequar o grafo no formato da biblioteca networkx para o

formato do algoritmo que irá realizar a busca. Esta opção foi escolhida para não limitar busca à biblioteca networkx, permitindo um desacoplamento maior caso não necessite de uma representação visual.

Nesta parte, também podemos destacar a inicialização de variáveis. A inicialização está em método apenas como uma estratégia para dividir o código, organizando em partes menores para simplificar.

```

45 def inicializaVariaveis():
46     global valorProfundidadeEntrada, valorProfundidadeSaida,
        profundidadesEntradaSaida, verticesPai, niveis,
47         controleVertices, demarcadores, verticeComSaidasValidas,
        corRedesenho, contadorCor, controleCor
48     valorProfundidadeEntrada = 0
49     valorProfundidadeSaida = 0
50     profundidadesEntradaSaida = {}
51     verticesPai = {}
52     niveis = {}
53     controleVertices = {}
54     demarcadores = set()
55     verticeComSaidasValidas = set()
56     corRedesenho = ["red", "green", "blue", "Gray"]
57     contadorCor = 0;
58     controleCor = True
59
60
61 def buscaEmProfundidade(vertices, arestas, verticeGrafo):
62     inicializaVariaveis()
63     grafo = {}
64     for v in vertices:
65         grafo[v] = []
66         for a in arestas:
67             if v == a[0]:
68                 grafo[v].append(a[1])
69     for vertice in grafo:
70         controleVertices[vertice] = vertice
71
72     verticesPai[verticeGrafo] = None
73     quantidadeFilhosRaiz = procuraProximoVertice(grafo, verticeGrafo, 1)
74     if quantidadeFilhosRaiz <= 1:
75         verticeComSaidasValidas.remove(verticeGrafo)
76
77

```

Com este código já teríamos as condições básicas para desenhar o grafo, porém a busca apenas está sendo preparada para que se possa inicializar. Vejamos as linhas:

- Linhas 45 a 58: método de inicialização de variáveis que é responsável por definir e atribuir os valores iniciais das variáveis que serão utilizadas no código.
- Linhas 61 a 63: método para iniciar a busca, no qual inicializa as variáveis realizando a chamada apropriada e define a variável grafo para receber os valores.
- Linhas 64 a 68: composição dos valores para a variável grafo. Os valores estão separados em vértices (vetor com os vértices) e as arestas (ligações possíveis entre os vetores). O conjunto de ‘for’ realiza a estruturação do dictionary do Python em vértice com a lista das suas respectivas ligações (arestas).
- Linhas 69 a 72: definem o vetor de vértices para buscas e referências no restante do código. Também define o vetor dos vértices que são “pai” de outros.
- Linhas 73: atribuição da quantidade de vértices que estão na posição de filho em referência ao vértice pai. É realizada a chamada no método *procuraPróximoVértice*, no qual realiza a busca para o vértice mais profundo (ou seja, vértice filho).
- Linhas 74 e 75: controle para verificar se um vértice não possui mais saídas, ou seja, não há um caminho mais profundo.

Nesta parte, iremos ver em detalhe o método recursivo que irá navegar entre as arestas para descobrir todos os vértices.

```

78 def procuraProximoVertice(grafo, verticeGrafo, nivel):
79     global valorProfundidadeEntrada, valorProfundidadeSaida, contadorCor,
80         controleCor
81     valorProfundidadeEntrada += 1
82     profundidadesEntradaSaida[verticeGrafo] = [valorProfundidadeEntrada,
83         None]
84     niveis[verticeGrafo] = nivel
85
86     contadorFilhos = 0
87
88     for proximoVertice in grafo.get(verticeGrafo):
89         if not profundidadesEntradaSaida.get(proximoVertice):
90             verticesPai[proximoVertice] = verticeGrafo
91             contadorFilhos += 1
92             atualizarDesenho(verticeGrafo, proximoVertice)

```

```

91     procuraProximoVertice(grafo, proximoVertice, nivel + 1)
92     if niveis[controleVertices[proximoVertice]] < niveis[controleVertices[verticeGrafo]]:
93         controleVertices[verticeGrafo] = controleVertices[proximoVertice]
94
95     if proximoVertice in demarcadores:
96         verticeComSaidasValidas.add(verticeGrafo)
97     else:
98         if not profundidadesEntradaSaida[proximoVertice][1]:
99             if verticesPai[verticeGrafo] != proximoVertice:
100                 if niveis[proximoVertice] < niveis[controleVertices[verticeGrafo]]:
101                     controleVertices[verticeGrafo] = proximoVertice
102
103     valorProfundidadeSaida += 1
104     profundidadesEntradaSaida[verticeGrafo][1] = valorProfundidadeSaida
105     if controleVertices[verticeGrafo] in (verticeGrafo, verticesPai[verticeGrafo]):
106         demarcadores.add(verticeGrafo)
107         if controleCor:
108             contadorCor += 1
109             controleCor = False
110
111     return contadorFilhos
112
113

```

Vejamos o código em detalhes:

- Linhas 78 e 79: método que irá buscar o vértice “filho”, vértice com uma profundidade maior. Também define as variáveis globais que serão usadas.
- Linhas 80 a 84: incrementa a profundidade que se encontra o grafo e atribui a variável de controle.
- Linha 86: laço de repetição responsável por verificar as arestas a partir do vértice atual.
- Linha 87: realiza o teste se há mais vértices abaixo (se pode executar uma busca ainda mais profunda)
- Linha 88: define o vértice “pai” do vértice encontrado e registra na estrutura *vértices Pai*
- Linha 89: incrementa a quantidade de filho do vértice.
- Linha 90: chamada do método *atualizar Desenho* para gravar uma imagem informando a aresta que se passou.
- Linha 91: chamada recursiva para o próprio método, porém o vértice enviado é do próximo nível para ser uma nova referência.
- Linhas 92 e 93: condição para atualizar o nível do grafo em relação ao vértice.

- Linhas 95 e 96: condição para informar o controle se o vértice atual tem uma saída válida (possui uma ligação com algum vértice que ainda não foi acessado).
- Linhas 97 a 101: quando a condição da linha 87 for falsa, será atualizado a variável de *controle Vértice* para que ser registre o vértice alcançado.
- Linhas 103 e 104: atribui variáveis de controle para o nível de profundidade da saída.
- Linhas 105 a 111: verifica se o vértice já foi alcançado e adiciona no vetor de marcadores, além de definir o controle de cor para atualizar a imagem criada com cores diferentes em cada ramificação.

A última parte do código é responsável por inicializar o programa, são definidas as informações principais e as chamadas para os métodos.

```

114 G = nx.DiGraph()
115 contadorFigura = 1
116 pdf = PdfPages('GrafoPassoAPasso.pdf')
117 vertices = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
118 arestas = [('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd')]
119
120 desenhoInicialGrafo()
121
122 buscaEmProfundidade(vertices, arestas, "a")
123
124 pdf.close()
125

```

Vejamos linha a linha:

- Linha 114: inicialização da biblioteca networkx para a criação de um grafo direcionado.
- Linhas 115 e 116: atribuição das variáveis que controlam a imagem e o nome padrão do arquivo PDF.
- Linhas 117 e 118: definição dos vértices e arestas do grafo.
- Linha 120: desenho inicial do grafo, no estado que ainda não foi realizada nenhuma busca.
- Linha 122: inicializa a busca em profundidade informando os vértices e arestas do grafo e qual o vértice que irá iniciar a busca.
- Linha 124: finalizar a criação do arquivo PDF, disponibilizando para uso.

A busca em profundidade, aqui apresentada, faz a simulação para atingir todas as possibilidades, em uma aplicabilidade prática, deverá incluir um parâmetro adicional que informa o valor da busca (vértice a ser encontrado); quando encontrar o vértice (destino), para-se a busca e exibe o “caminho” percorrido.

**Busca em Largura** - A busca em largura é outra forma de realizar uma busca em grafo. Enquanto a busca em profundidade vai até o último nível para trocar um vértice inicial, a busca em largura procura em todos os vértices de um mesmo nível antes de ir para um nível mais profundo.

Donadelli (2010) explica que se deve percorrer um vértice não visitado em mesma profundidade ou nível. Após ter encontrado todos os vértices, repete-se o processo em um nível mais profundo.

Na representação da busca em largura, vamos utilizar o mesmo exemplo anterior com sete vértices (a, b, c, d, e, f, g) com as arestas de forma direcional, assim como está representado na imagem a seguir.

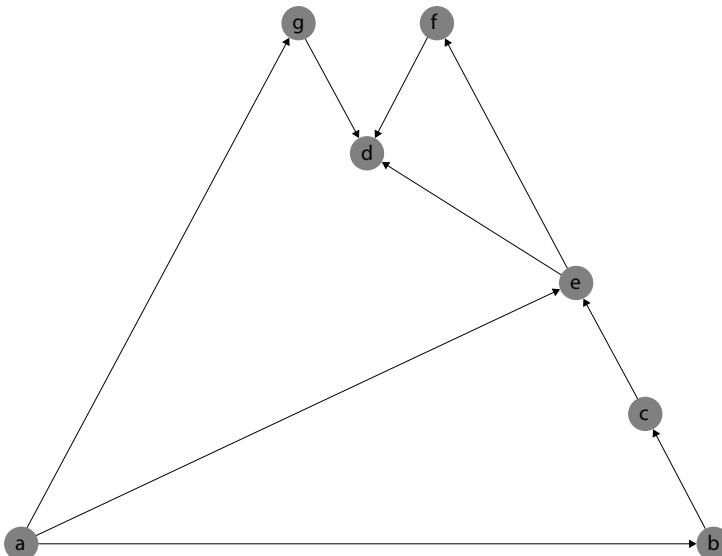


Figura 6 - Grafo com 7 vértices / Fonte: os autores.

**Descrição da Imagem:** Grafo com 7 vértices que contém as seguintes ligações: ('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('e', 'd'), ('a', 'g'), ('g', 'd').

Vamos supor que o vértice inicial seja o vértice com valor ‘a’ e estamos buscando o vértice com valor ‘f’, assim, a primeira busca poderia ser iniciada no vértice ‘b’.

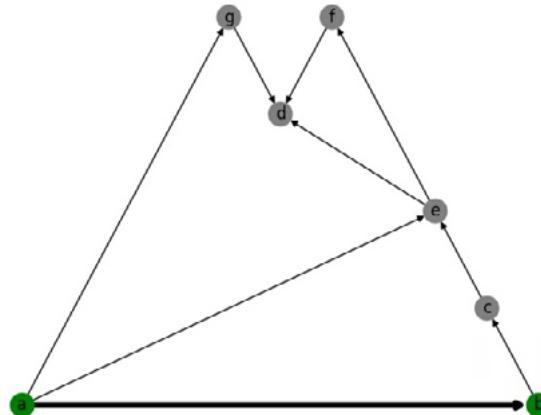


Figura 7 - Busca em largura, etapa 1 / Fonte: os autores.

**Descrição da Imagem:** Grafo com 7 vértices que contém as seguintes ligações: ('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd'). Destaque na aresta (a, b) grifada em preto e os vértices 'a' e 'b' em verde, representando um vértice visitado.

Perceba que, nesta etapa, a busca em largura e a busca em profundidade possui o mesmo resultado, pois um vértice testado não representa a diferença. Contudo, uma vez que não encontra, deverá testar o próximo vértice. Na busca em largura, o próximo vértice deve possuir uma aresta com o vértice 'a'.

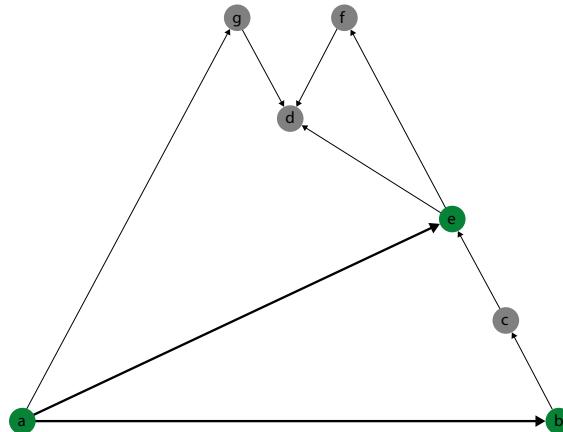


Figura 8 - Busca em largura, etapa 2 / Fonte: os autores.

**Descrição da Imagem:** Grafo com 7 vértices que contém as seguintes ligações: ('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd'). Destaque na aresta (a, b) e (a, e) grafadas em preto e os vértices 'a', 'b' e 'c' em verde representando um vértice visitado.

Veja que, agora, o vértice 'c' não foi buscado ainda, pois ele não possui ligação com o vértice 'a'. A busca em largura irá buscar todos os vértices que possuírem arestas com o vértice até passar para o vértice de nível mais baixo, por exemplo, o vértice 'b'. Essa estratégia se repetirá até o final ou quando encontrar o valor procurado em um vértice do grafo.

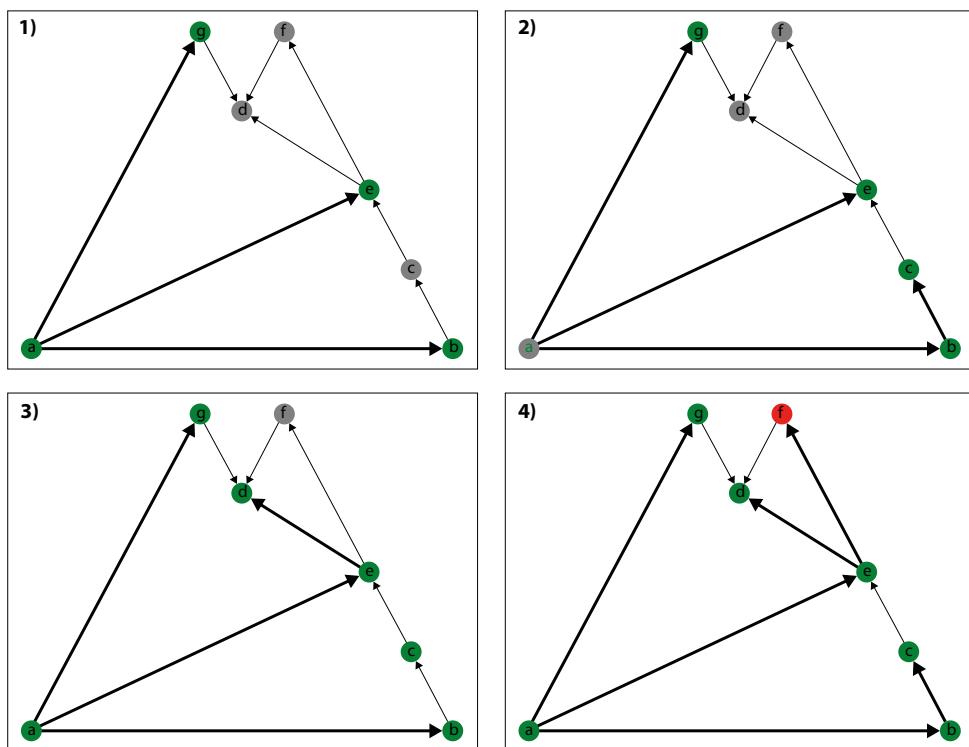


Figura 9 - Busca em largura, etapas 3, 4, 5 e 6. / Fonte: os autores.

**Descrição da Imagem:** quatro imagens representando as etapas finais da busca em largura no grafo com 7 vértices que contém as seguintes ligações: ('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd'). A imagem 1 representa a etapa 3, no qual passa por todos os vértices com arestas em 'a'. A imagem 2 (etapa 4) possui uma busca em um vértice de nível mais baixo, aleatoriamente foi escolhido o vértice 'b' como início e buscar todos os vértices com ligação com ele. Na imagem 3 (etapa 5), traça-se o vértice de início, pois o vértice 'b' apenas possui uma ligação. Na imagem 4 (etapa 6) é encontrado o vértice e destacado em vermelho o vértice 'f'.

Veja que esta busca em largura é similar à busca em profundidade com a diferença de buscar todos os vértices de um nível para passar ao próximo nível. Podemos criar o algoritmo em diversas linguagens e, neste exemplo, vamos montar, no

Python, utilizando a biblioteca networkx para criar o grafo e auxiliar nas buscas. O grafo é o mesmo da representação gráfica, no qual iniciaremos a busca no vértice 'a' e iremos procurar o valor 'f'. Dividimos o programa em partes para melhor explicar e representar.

```
1 import networkx as nx
2 from matplotlib import pyplot as plt
3 from matplotlib.backends.backend_pdf import PdfPages
4
5
6 def desenharGrafo(corAresta, espessuraAresta, corVertice):
7     global contadorFigura
8     nx.draw(G, with_labels=True,
9             pos=positionamento,
10            edge_color=corAresta,
11            width=list(espessuraAresta),
12            node_color=corVertice)
13 pdf.savefig()
14 nomeImagem = "grafo" + str(contadorFigura) + ".jpg"
15 contadorFigura += 1
16 plt.savefig(nomeImagem)
17 plt.show()
18
19
20 def desenhoInicialGrafo():
21     global positionamento
22     for n1, n2 in arestas:
23         G.add_edge(n1, n2, label=(n1 + "->" + n2), color="black", weight=1)
24         G.add_nodes_from(vertices, color="gray")
25         corAresta, espessuraAresta = definirAtributos()
26         positionamento = nx.planar_layout(G)
27         desenharGrafo(corAresta, espessuraAresta, nx.get_node_attributes(G, 'color').values())
28
29
30 def definirAtributos():
31     corAresta = nx.get_edge_attributes(G, 'color').values()
32     espessuraAresta = nx.get_edge_attributes(G, 'weight').values()
33     return corAresta, espessuraAresta
34
35
36 def atualizarDesenho(v1, v2, corVertice):
37     G.get_edge_data(v1, v2)["color"] = "black"
38     G.get_edge_data(v1, v2)["weight"] = 4
39     corAresta, espessuraAresta = definirAtributos()
40     desenharGrafo(corAresta, espessuraAresta, corVertice)
41
42
```

O programa utiliza alguns métodos que são similares ao da busca em profundidade, com algumas diferenças sutis que vamos passar no detalhe.

- Linhas 1 a 3: importação das bibliotecas que irão montar o grafo; desenhar e gerar um PDF com os desenhos do grafo por fases.
- Linhas 6 a 17: método para desenhar o grafo, são passados 3 parâmetros: cor da aresta, espessura e cor do vértice. Cada alteração do grafo será chamada para a atualizar as propriedades do grafo. Este método, que além de desenhar, deverá salvar a imagem na pasta e incluir no arquivo PDF resultante.
- Linhas 20 a 27: método que irá montar o grafo inicialmente com as arestas e vértices. Atenção: a linha 27 no qual será chamado o método de desenhar o grafo, o último parâmetro está recuperando as cores definidas para os vértices e convertendo para lista, assim poderá ser desenhada a imagem com a cor cinza para todos os vértices.
- Linhas 30 a 33: método que irá coletar as cores atuais das arestas e suas respectivas espessuras.
- Linhas 36 a 40: método que irá atualizar o desenho do grafo em cada etapa; ao final, irá chamar o método de desenhar o grafo para criar uma nova imagem do grafo atualizado.

O programa, até este momento, está realizando o controle visual das imagens resultantes do grafo para efeitos de exemplificação. O método responsável pela busca está realizando o controle por cor dos vértices. Veja o código.

```

43 def buscaEmLargura(G, verticeInicial, valorProcurado):
44     vetorNiveis = {}
45     vetorPredecessoras = {}
46     controleVisitadosPorCor = nx.get_node_attributes(G, 'color')
47     controleVisitadosPorCor[verticeInicial] = 'green'
48     vetorNiveis[verticeInicial] = 0
49     filaControle = [verticeInicial]
50     while filaControle:
51         verticePai = filaControle.pop(0)
52         for verticeEncontrado in G.neighbors(verticePai):
53             if controleVisitadosPorCor[verticeEncontrado] == 'gray':
54                 vetorNiveis[verticeEncontrado] = vetorNiveis[verticePai] + 1
55                 vetorPredecessoras[verticeEncontrado] = verticePai
56                 filaControle.append(verticeEncontrado)

```

O programa se resume a um método que realizará a busca, vamos entender linha a linha:

- Linha 43: definição do método no qual é passado o objeto grafo, vértice que irá iniciar a pesquisa e o valor a ser procurado.
  - Linhas 44 a 49: atribuição das variáveis de controle como o vetor dos níveis de profundidade por vértice e o vetor predecessores que terá o vértice de cada predecessor (pai), o controle de vértices visitados pelo atributo cor e, por fim, a fila de controle que irá manter em fila os vértices.
  - Linhas 50 e 51: laço de repetição que irá passar pela fila dos vértices que são pais de outros. A remoção do vértice a ser procurado da fila.
  - Linha 52: laço de repetição que irá passar por todos os vértices conectados ao vértice pai. O método *neighbors* retorna exatamente isso, todos os vértices conectados ao parâmetro passado.
  - Linha 53: teste para verificar se o vértice não foi visitado. O controle está sendo realizado pelo atributo color; quando for cinza, significa que não foi visitado.
  - Linhas 54 a 56: incluído o nível para o vértice encontrado para exemplificar em que nível se encontra o vértice. Assim como o vetor de predecessor e incluído na fila.
  - Linhas 57 a linha 63: condição para verificar se o vértice que foi encontrado é igual ao vértice procurado. Quando é verdadeiro, o vértice terá cor vermelha, irá desenhar o grafo no método *atualizar Desenho*. Irá exibir, no console, que foi encontrado o vértice e exibirá em qual nível ele se encontra.

- Linhas 64 a 66: como não encontrou o valor, irá informar que o vértice foi visitado com a cor verde e desenhar o grafo atualizado.

Veja que, neste exemplo, o controle para verificar se um vértice foi visitado foi realizado por meio de cores. Contudo, poderia ser uma lista com os valores dos vértices já visitados. O controle pode ser alterado conforme a necessidade e objetivo, porém a mecânica do algoritmo é a mesma em qualquer cenário. Por fim, a última parte do programa contém os dados de inicialização.

```

69 G = nx.DiGraph()
70 posicionamento = nx.planar_layout(G)
71 contadorFigura = 1
72 vertices = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
73 arestas = [('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd')]
74 pdf = PdfPages('GrafoPassoAPasso.pdf')
75 desenhoInicialGrafo()
76
77 buscaEmLargura(G, "a", "f")
78 pdf.close()
79

```

Vejamos o código:

- Linha 69: instanciado o objeto do grafo.
- Linhas 70 e 71: definição inicial do layout do grafo e o contador das imagens que serão salvas.
- Linhas 72 a 74: definição dos vértices e arestas do grafo e o nome do arquivo em PDF que será salvo.
- Linhas 75 a 78: chamada do método de desenho inicial para criar o grafo sem a busca e inicialização da busca em largura com o vértice de início e o valor buscado. Por fim, fecha-se o arquivo PDF.

Este grafo de exemplo representa uma estrutura simples, mas que pode ser adaptada a vários cenários que podem auxiliar e encontrar objetos de busca sempre na transversal. O que nos dá a ideia que ele é mais efetivo quando a probabilidade do objeto da busca não está em um nível profundo.

**Dijkstra** - O algoritmo é um dos vários utilizados para calcular o custo mínimo utilizando o grafo. Parte do princípio simples que um viajante terá um ponto de partida e precisa saber o custo mínimo para chegar ao destino. Barros,

Pamboukian e Zamboni (2007) explicam que o algoritmo foi desenvolvido, inicialmente, por Edsger Dijkstra, que estudou física teórica na Universidade de Leiden, mas acabou trocando sua área de atuação pela Ciências da Computação devido ao seu grande interesse no assunto.

O algoritmo consiste em identificar os vértices e os respectivos valores de suas arestas para calcular o menor custo do vértice de início e o vértice de destino. A premissa inicial é que não existem valores negativos para que se possa identificar o custo mínimo. Desta forma, iremos identificar todos os trajetos possíveis até o destino e calcular o que tem o menor custo. Tomaremos como exemplo o mesmo grafo utilizado anteriormente, porém, agora, com valores nas arestas.

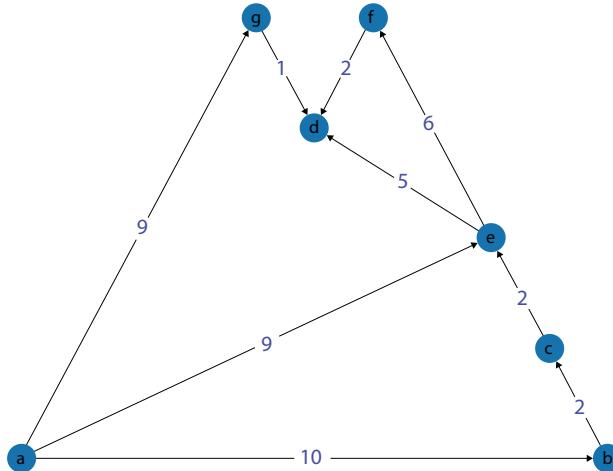


Figura 10 - Grafo de distância / Fonte: os autores.

**Descrição da Imagem:** Grafo com 7 vértices que contém as seguintes ligações e tamanhos: ('a', 'b'): 10, ('a', 'e'): 9, ('b', 'c'): 2, ('e', 'd'): 5, ('e', 'f'): 6, ('c', 'e'): 2, ('f', 'd'): 2, ('a', 'g'): 9, ('g', 'd'): 1.

Uma vez que tenhamos este grafo, é necessário que identifiquemos os percursos possíveis do vértice de início até o vértice de destino. O exemplo que vamos trabalhar, iremos utilizar o vértice 'a' como início e o vértice 'd' como destino. Desta forma, precisaremos testar todos os percursos possíveis até o destino, calculando o custo, conforme os valores que estão dispostos em cada aresta. No programa em Python, vamos utilizar duas abordagens simultâneas para calcular o custo mínimo. A primeira, um algoritmo criado em Python para exemplificar, e a segunda utilizando a biblioteca *networkx*.

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4
5 def caminhosPossiveis(grafo, proximo, alvo):
6     global caminhos, possibilidades
7     for aresta in list(filter(lambda filtro: (filtro[0] == proximo), grafo.edges)):
8         if aresta[0] == proximo:
9             caminhos.append(aresta)
10            if aresta[1] == alvo:
11                possibilidades.append(caminhos.copy())
12                caminhos.pop()
13            else:
14                caminhosPossiveis(grafo, aresta[1], alvo)
15    if len(caminhos) > 0:
16        caminhos.pop()
17
18
19 def montarRastreioVertices(trajeto, vertice):
20     if trajeto not in vertice:
21         vertice.append(trajeto)
22
23
24 def dijkstra(grafo, origem, alvo):
25     caminhosPossiveis(grafo, origem, alvo)
26     menor = -1
27     for percusto in possibilidades:
28         soma = 0
29         vertice = []
30         for trajeto in percusto:
31             soma += nx.get_edge_attributes(G, "length")[trajeto]
32             montarRastreioVertices(trajeto[0], vertice)
33             montarRastreioVertices(trajeto[1], vertice)
34         if menor == -1 or menor > soma:
35             menor = soma
36             encontrado["rastro"] = vertice
37             encontrado["percusro"] = percusto
38             encontrado["distancia"] = menor
39
40

```

Nesta primeira parte do programa, vamos tratar, exclusivamente, do algoritmo para encontrar o menor custo:

- Linhas 1 e 2: importação das bibliotecas networkx para trabalhar com o grafo e a matplotlib para desenhar o grafo.

- Linhas 5 e 6: definição da função para encontrar os caminhos possíveis e as variáveis globais que serão trabalhadas.
- Linha 7: laço *for* que irá extrair os vértices de início de cada aresta, neste exemplo foi adotada uma função lambda que permite criar uma função em um parâmetro; esta instrução realizará um filtro que irá considerar somente aos vértices de partida de cada aresta que forem igual a variável próximo. Isso garante, no nosso exemplo, se a variável ‘próximo’ conter o valor ‘a’ somente irá listar os vértices ('a', 'b'), ('a', 'e') e ('a', 'g').
- Linhas 8 a 9: é incluído a aresta e encontrada a lista de caminho realizado.
- Linhas 10 a 12: quando encontrar o destino na aresta, é incluído todo o caminho realizado na lista de possibilidades, e retira do caminho realizar a última aresta. Esta remoção é para garantir a verificação se há outro caminho que poderia ser seguido.
- Linhas 13 a 14: como não é o alvo, descemos o nível e será chamada a mesma função que fará a nova busca, porém será atualizado o valor do próximo que é o destino da aresta atual. Por exemplo: da aresta ('a', 'b') será extraído o vértice 'b' e enviado para a função como próximo para testar o próximo nível, no caso ('b', 'c').
- Linhas 15 a 16: após montar a lista de caminhos possíveis, é retirado removido a aresta da lista de caminho para voltar ao nível anterior (nível do pai).
- Linhas 19 a 21: extração dos vértices do trajeto para a lista de vértice.
- Linhas 24 a 25: definição da função dijkstra e a chamada para a função *caminhosPossiveis*.
- Linha 26: atribuição da variável ‘menor’.
- Linhas 27 a 29: laço *for* que irá verificar cada percurso identificado e incluído na lista de possibilidades. Neste trecho, inclusive, são definidas as variáveis auxiliares de soma e lista dos vértices.
- Linha 30: laço *for* que irá extrair o trajeto que seria a aresta entre cada vértice.
- Linha 31: a variável soma totaliza os valores de cada aresta do grafo.
- Linhas 32 e 33: extrai os vetores envolvidos no trajeto.
- Linha 34: após somar o valor de trajeto, verifica se o valor encontrado é o menor que o valor anterior ou se é o primeiro percurso.
- Linha 33 a 38: o percurso menor é encontrado, com isso, é possível atualizar a variável ‘menor’ com o valor de ‘soma’. O *dictionary* é atribuído com a lista de vértice, percurso realizado e o custo total do percurso.

O programa realiza a busca e armazena, em uma variável, as principais informações que respondem as perguntas de: quais referências? Por onde passar? Qual o custo? A segunda parte do programa é responsável por montar o grafo e apresentar os resultados.

```

41 def criarGrafo():
42     global arestas, labels, vertices
43     arestas = [('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'),
44             ('g', 'd')]
45     labels = {('a', 'b'): 10, ('a', 'e'): 9, ('b', 'c'): 2, ('e', 'd'): 5, ('e', 'f'): 6,
46             ('c', 'e'): 2, ('f', 'd'): 2, ('a', 'g'): 9,
47             ('g', 'd'): 1}
48     vertices = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
49     for n1, n2 in arestas:
50         G.add_edge(n1, n2, length=labels[(n1, n2)],
51                     color="black",
52                     weight=labels[(n1, n2)])
53     G.add_nodes_from(vertices, color="gray")
54     posicionamento = nx.planar_layout(G)
55     nx.draw_networkx_edge_labels(G, posicionamento,
56                                 edge_labels=nx.get_edge_attributes(G, "length"),
57                                 font_color="blue")
58     nx.draw(G, with_labels=True, pos=posicionamento)
59     plt.show()
60
61
62 labels = {}
63 arestas = []
64 vertices = []
65 encontrado = []
66 caminhos = []
67 possibilidades = []
68 G = nx.DiGraph()
69 criarGrafo()
70
71 verticePartida = "a"
72 verticeDestino = "d"
73
74 dijkstra(G, verticePartida, verticeDestino)
75 print("Rastreio: ", encontrado["rastro"])
76 print("Percuso: ", encontrado["percuso"])
77 print("Distância percorrida: ", encontrado["distancia"])
78 print("Biblioteca networkx")
79 print("Rastreio na biblioteca: ", nx.shortest_path(G, verticePartida, verticeDestino, 'length'))
80 print("Distância percorrida: ", nx.shortest_path_length(G, verticePartida, verticeDestino, 'length'))
81

```

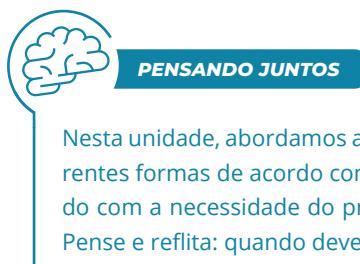
Vejamos linha a linha:

- Linhas 41 a 42: define a função para criar o grafo e as variáveis globais.
- Linhas 43 a 48: definem as arestas, valores e vértices para o grafo.
- Linhas 49 a 52: atribui ao grafo G as arestas com seus respectivos valores de custo, utilizando o atributo *length* para representar a distância.
- Linhas 53 a 59: é atribuído os vértices, layout do grafo, assim como exibir os valores dos tamanhos nas arestas.

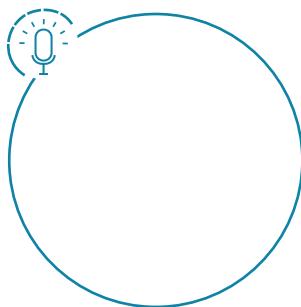
- Linhas 62 a 67: inicializar as variáveis globais.
- Linhas 68 e 69: cria o grafo usando a biblioteca networkx e chama a função *criarGrafo* para atribuir os valores.
- Linhas 71 a 74: define os vértices de partida e destino que serão utilizados no programa e chama a função *dijkstra* com seus parâmetros para encontrar o menor percurso.
- Linha 75 a 77: exibe os resultados encontrados.
- Linhas 78 a 80: realiza a mesma ação de encontrar o percurso com o menor custo, porém é calculado pela biblioteca *networkx*. São exibidos os vértices e o custo total.

Veja que, no exemplo, também é utilizando a biblioteca *networkx* para calcular o menor custo, ambas as formas irão exibir os mesmos resultados. Uma dica importante para entender o que está ocorrendo no programa é realizar um 'debug' na ferramenta, assim ficará mais claro cada ação.

A aplicabilidade desta estrutura de busca pode ser realizada para cálculos de rotas em GPS, percursos de aeronaves, naveabilidade marítima, ou seja, em qualquer problema que necessite minimizar os custos de distância baseada em números.



Nesta unidade, abordamos as buscas em grafos, no qual permite percorrer os nós de diferentes formas de acordo com o seu contexto. Lembre-se que o grafo está ligado de acordo com a necessidade do problema, então, temos que pensar no melhor tipo de busca. Pense e reflita: quando devemos utilizar a busca em profundidade ou busca em largura?



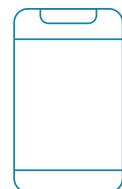
Agora que estudamos a busca por profundidade e o quanto ela é importante para resolver problemas como fraude, sendo que esse tipo de busca é muito utilizado em Machine Learning para modelos de prescrição e predição, vamos à nossa roda de conversa. Neste Podcast, vamos falar mais sobre as estruturas de busca em profundidade e como ela tem auxiliado nas automações residenciais e no combate à fraude. Não perca e acesse o QR Code para conferir essa conversa.


**EXPLORANDO IDEIAS**

**A busca em profundidade (Depth-first search ou DFS)** consiste em seguir todas as arestas e, assim, irá alcançar todos os vértices. Fazendo uma analogia, a busca se assemelha a um labirinto que procuramos um caminho entre a entrada e saída, escolhemos um caminho que pode conduzir à saída e quando encontramos um caminho sem saída temos que retornar e escolher outro caminho. A busca por profundidade é utilizada quando o problema possui um grande número de soluções, ou se a maioria dos caminhos pode levar a solução do problema, assim esse tipo de busca é muito mais eficiente.

**A busca por largura (Breadth-First Search ou BFS)** é utilizada para explorar o grafo visitando os vértices por níveis. Enquanto a busca em profundidade vai até o último nível para trocar um vértice inicial, a busca em largura procura em todos os vértices de um mesmo nível antes de ir para um nível mais profundo.

**A busca pelo algoritmo de Dijkstra ou busca por caminho mínimo** é utilizado quando existir, no grafo, vários caminhos entre um par de nós, como origem e destino. Esse algoritmo calcula os caminhos e atribui um peso para os vários caminhos. Aquele que possui o menor “peso” é chamado de caminho mínimo.

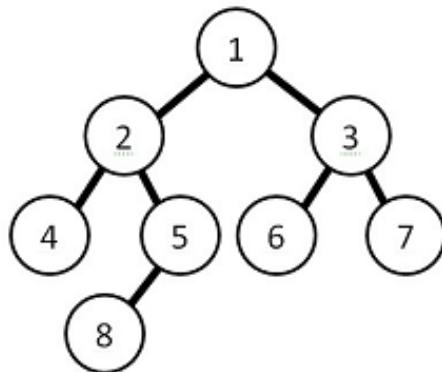

**NOVAS DESCOBERTAS**


Você conhece o jogo Pacman? Agora que você estudou busca por profundidades, saberia resolver o jogo usando esse tipo de busca? O Pacman é um labirinto e você pode escolher diferentes caminhos para pegar as bolinhas e poder prender os fantasmas, neste caso, a busca por profundidade poderá auxiliar para nos dizer todas as rotas possíveis. Você pode jogar ele de forma on-line acessando nosso QR Code.

Como vimos nesta unidade, é importante entender os algoritmos de busca por profundidade e como ela está inserida nos sistemas que utilizamos no dia a dia. Com base nos seus estudos, tente achar dois jogos que podem utilizar a busca por profundidade para auxiliar o jogador a ganhar, escreva uma explicação sobre a sua teoria de como esse algoritmo pode auxiliar.



1. De acordo com os problemas reportados a seguir, qual será melhor resolvido pela busca em profundidade?
  - a) Trajeto entre duas cidades mais rápido.
  - b) Buscar um amigo na rede social e mostrar todas ligações até chegar neste amigo.
  - c) Buscar o amigo mais próximo da sua localidade.
  - d) Jogo sudoku.
  - e) Achar o caminho mais curto entre duas pessoas na rede social.
  
2. Considerando a busca de profundidade no grafo a seguir, informe qual a ordem que seria a pesquisa.

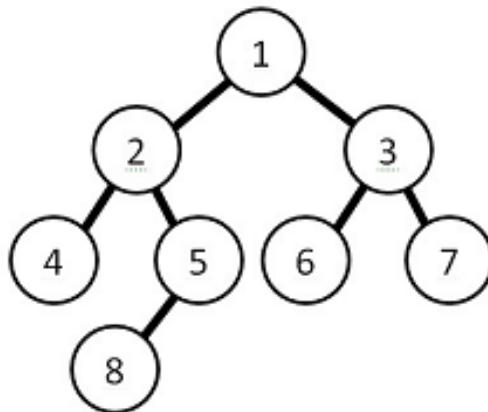


**Descrição da Imagem:** representação do grafo no qual temos os pontos 1, 2, 3, 4, 5, 6, 7, 8 e ligações (linhas) entre os pontos. No topo está no ponto "1", abaiixo e ligado ao ponto 1 está o ponto 2 e 3, abaiixo e ligado ao ponto 2 está o ponto 4 e 5, abaiixo e ligado ao ponto 5 está o ponto 8, abaiixo e ligado ao ponto 3 está o ponto 6 e 7..

- a) 4, 8, 5, 2, 6, 7, 3, 1;
- b) 1, 2, 4, 5, 8, 3, 6, 7;
- c) 1, 2, 3, 4, 5, 6, 7, 8.
- d) 4, 2, 8, 5, 1, 6, 3, 7.
- e) 4, 8, 2, 5, 7, 6, 3, 1;



3. Considerado a busca de largura no grafo a seguir, informe qual a ordem que seria a pesquisa.

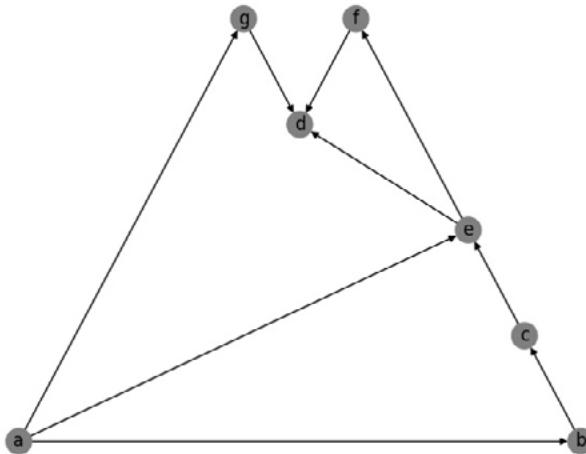


**Descrição da Imagem:** representação do grafo no qual temos os pontos 1, 2, 3, 4, 5, 6, 7, 8 e ligações (linhas) entre os pontos. No topo está no ponto "1", abaixo e ligado ao ponto 1 está o ponto 2 e 3, abaixo e ligado ao ponto 2 está o ponto 4 e 5, abaixo e ligado ao ponto 5 está o ponto 8, abaixo e ligado ao ponto 3 está o ponto 6 e 7.

- a) 4, 8, 5, 2, 6, 7, 3, 1.
- b) 1, 2, 4, 5, 8, 3, 6, 7.
- c) 1, 2, 3, 4, 5, 6, 7, 8.
- d) 4, 2, 8, 5, 1, 6, 3, 7.
- e) 4, 8, 2, 5, 7, 6, 3, 1.



4. Considerando a busca de profundidade no grafo a seguir, informe qual a ordem que seria a pesquisa iniciada no nó a.

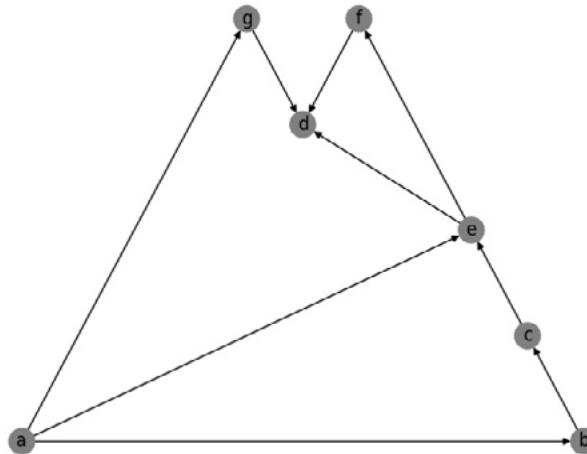


**Descrição da Imagem:** quatro imagens representando as etapas finais da busca em largura no grafo com 7 vértices que contém as seguintes ligações: ('a', 'b'), ('a', 'c'), ('b', 'c'), ('d', 'e'), ('f', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd')

- a) a, b, c, e, d, f, g.
- b) a, d, f, g, b, c, e.
- c) a, e, d, f, b, c, g.
- d) a, c, e, b, d, f, g.
- e) a, b, c, d, e, f, g.



5. Considerando o algoritmo de Dijkstra, qual o caminho percorrido para chegar do ponto A para o ponto F na imagem a seguir:



**Descrição da Imagem:** quatro imagens representando as etapas finais da busca em largura no grafo com 7 vértices que contém as seguintes ligações: ('a', 'b'), ('a', 'e'), ('b', 'c'), ('e', 'd'), ('e', 'f'), ('c', 'e'), ('f', 'd'), ('a', 'g'), ('g', 'd')

- a) a, b, c, e, d, f.
- b) a, b, c, e, f.
- c) a, e, d, f.
- d) a, e, f,
- e) a, g, d, f.

# REFERÊNCIAS



## UNIDADE 1

GANTZ, J.; REINSEL, D. **The digital universe in 2020: Big Data, bigger digital shadows, and biggest growth in the Far East.** Framingham: IDC, 2012. Disponível em: <https://www.cs.princeton.edu/courses/archive/spring13/cos598C/idc-the-digital-universe-in-2020.pdf>. Acesso em: 18 ago. 2021.

PARANÁ, E. **A digitalização do mercado de capitais no Brasil: tendências recentes.** Instituto de Pesquisa Econômica Aplicada. Brasília: Rio de Janeiro, 2018. Disponível em: [http://repositorioipea.gov.br/bitstream/11058/8280/1/TD\\_2370.PDF](http://repositorioipea.gov.br/bitstream/11058/8280/1/TD_2370.PDF), Acesso em: 18 ago. 2021.

PERKOVIC, L. **Introdução à computação usando Python:** um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

### Referências on-line

<sup>1</sup> Em: <https://www.domo.com/learn/data-never-sleeps-8>. Acesso em: 17 ago. 2021.

## UNIDADE 2

PERKOVIC, L. **Introdução à computação usando Python:** um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PODER360. 41% usam veículos jornalísticos na internet para ler notícias. Poder360, 16 out. 2020. Disponível em: <https://www.poder360.com.br/poderdata/41-usam-veiculos-jornalisticos-na-internet-para-ler-noticias/>. Acesso em: 19 ago. 2021.

HU, Y. Fácil Aprendizagem Estruturas De Dados e Algoritmos Python 3: Aprenda graficamente estruturas de dados e algoritmos Python. [S.l.s.n]: 2020.

### Referências on-line

<sup>1</sup>Em: <https://www.emarketer.com/chart/242908/retail-e-commerce-sales-worldwide-2019-2024-trillions-change-of-total-retail-sales%20>. Acesso em: 19 ago. 2021.

<sup>2</sup>Em: <https://www.invespcro.com/blog/images/blog-images/global-online-retail-spending.jpg>. Acesso em: 19 ago. 2021.

<sup>3</sup>Em: <https://wearesocial.com/digital-2021>. Acesso em: 19 ago. 2021.

# REFERÊNCIAS



## UNIDADE 3

GANDOMI, A.; HAIDER, M. **Beyond the hype : Big data concepts , methods , and analytics.** [S.I]: International Journal of Information Management, 2015. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0268401214001066?via%3Dihub>. Acesso em: 03 set. 2021.

MICHAELIS. **Dicionário Online de Português.** [S.I]: Editora Melhoramentos Ltda, 2021. Disponível em: <https://michaelis.uol.com.br/busca?r=0&f=0&t=0&palavra=listar>. Acesso em: 03 set. 2021.

### Referências on-line

<sup>1</sup> Em: <https://docs.python.org/3/>. Acesso em: 07 set. 2021.

## UNIDADE 4

FEOFIOFF, P.; KOHAYAKAWA, Y.; WAKABAYASHI, Y. **Uma Introdução Sucinta à Teoria dos Grafos.** Instituto de Matemática e Estatística da Universidade de São Paulo, 2005.

GITHUB. Token Simulation. **GitHub**, [2021]. Disponível em: <https://raw.githubusercontent.com/bpmn-io/bpmn-js-token-simulation-plugin/HEAD/docs/screenshot.png>. Acesso em: 21 out. 2021.

GOOGLE. **Cidade Königsberg.** [2021]. Disponível em: <https://lh3.googleusercontent.com/-j-3FgeieqV40/TtoeyD0k0sl/AAAAAAAHH9A/R3a27ZkGnzo/s1600/1641.jpg>. Acesso em: 21 out. 2021.

MEUSEL, R. *et al.* The Graph Structure in the Web – Analyzed on Different Aggregation Levels. **The Journal of Web Science**, v. 1, 2015. Disponível em: <https://webscience-journal.net/webscience/article/view/11>. Acesso em: 22 out. 2021.

NETWORKX. **NetworkX Analysis in Python.** Networkx, [2021]. Disponível em: <https://networkx.org/>. Acesso em: 22 out. 2021..

SEJ. Search Engine Journal. **Sitemaps** 2021. Disponível em: <https://cdn.searchenginejournal.com/wp-content/uploads/2021/04/sitemaps-image-and-xml-60745e8a86970-760x400.jpg>. Acesso em: 21 out. 2021.

# REFERÊNCIAS



## UNIDADE 5

BARROS, E. A. R.; PAMBOUKIAN, S. V. D.; ZAMBONI, L. C. **Algoritmo de Dijkstra:** Apoio Didático e Multidisciplinar na Implementação, Simulação e Utilização Computacional. International Conference on Engineering and Computer Education - ICECE, 2007. São Paulo Brasil. Disponível em: [http://meusite.mackenzie.com.br/edsonbarros/publicacoes/ICECE2007\\_212.pdf](http://meusite.mackenzie.com.br/edsonbarros/publicacoes/ICECE2007_212.pdf). Acesso em 26 out. 2021.

DONADELLI, J. **Teoria de Grafos:** Uma breve introdução com algoritmos. Universidade Federal do Paraná, 2010. Disponível em: <http://aleph0.info/cursos/tg/2011-q3/grafos-jair-donadelli.pdf>. Acesso em: 26 out. 2021.



## UNIDADE 1

1. E. As estruturas heterogêneas são responsáveis por armazenar variáveis de diferentes tipos de dados.
2. D. São consideradas estruturas de dados: tuplas, vetor, Set e Matrizes. Os Artefatos não são estruturas de dados.
3. C. A estrutura de dados é responsável pelas localizações de memória em que esses dados são armazenados durante a execução do sistema.
4. A. As matrizes são divididas em linhas e colunas de forma multidimensionais e para identificar um valor precisamos saber o número da linha e da coluna.
5. D. O Set é uma coleção desordenada de elementos que não possui elementos em duplicidade e índices para referência. Desta forma, poderemos utilizar o set quando a necessidade de manter um dado sem duplicidade.
6. A. Os dicionários são estruturas de dados que são usados basicamente para mapear chaves e valores. Essa estrutura é utilizada tanto para manipular quanto para armazenar os dados. Os índices(chaves) podem ser praticamente qualquer valor e pode estar associada a somente um valor.
7. B. Como na linha 4 o atribuiu o Vetor A em Vetor B os dois se tornaram o mesmo vetor.



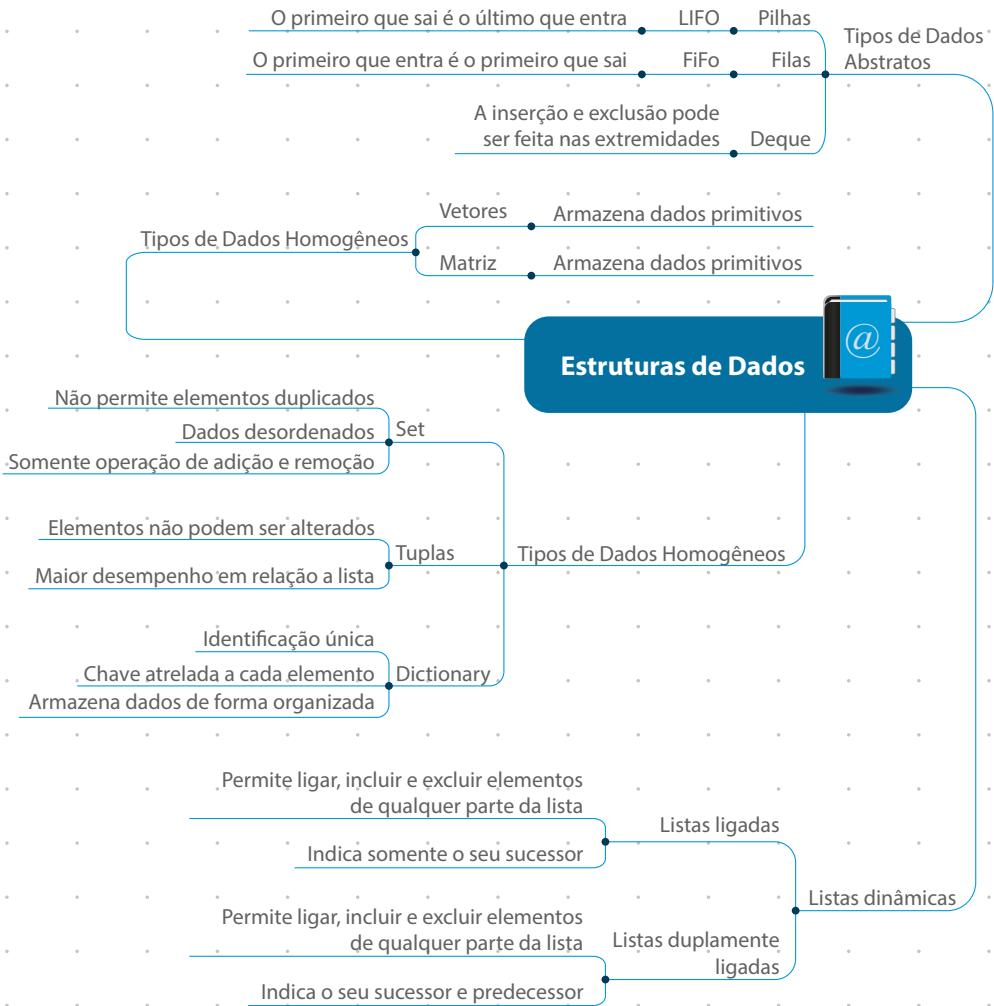
## UNIDADE 2

1. A. A Pilha utiliza o conceito de LIFO (Last In, First Out, ou seja, último a entrar, primeiro a sair).
2. E. De acordo com a ordem de inserção e remoção, observe que está utilizando o conceito de Pilha, os últimos que entraram foram o primeiro a sair.
3. C. De acordo com a ordem de inserção e remoção, observe que está utilizando o conceito de FILA, os primeiros que entraram foram os primeiros a sair.
4. B. De acordo com o conceito de fila, observe os ponteiros próximos, o Avaí aponta para o Flamengo que aponta para o Figueirense, sendo assim o terceiro registro.
5. A. A Lista permite a inserção no meio do nó.
6. A. Devido ao comando `livros.pop()` que removeu valores Livro 4 e Livro 6 a ordem ficou: Livro 7, Livro 5, Livro 3, Livro 2, Livro 1

# CONFIRA SUAS RESPOSTAS



## UNIDADE 3



# CONFIRA SUAS RESPOSTAS



## UNIDADE 4

1. B - Grafo é a estrutura de dados que trabalha com vértices e arestas interligadas.
2. B - Grafo Euleriano contém pares de vértices ligados por linhas, e cada vértice representa um terreno. Como existiam duas margens e duas ilhas, faltava ligar os vértices.
3. A - Os grafos são formados por Vértices, arestas e função (Subconjuntos).
4. C - O grafo que não possui um ou mais vértices conectados é chamado de desconexo.
5. D - O grafo que possui todas os vértices conectadas é chamado de conexo.

## UNIDADE 5

1. B. A busca por profundidade vai mostrar todas as ligações vinculadas a todos os níveis.
2. D. A busca vai começar da esquerda, no menor nível, e depois vai percorrendo os níveis para a direita.
3. C. A busca vai começar do primeiro nível e depois vai navegando lateralmente.
4. A. Considerando as ligações, é necessário percorrer na ordem das ligações e vá para o G quando não encontra um caminho direto, então tem que retornar.
5. D. O algoritmo de Dijkstra considera o caminho mais curto.

MEU ESPAÇO

