

Unidade 1

FUNDAMENTOS DE ESTRUTURAS DE DADOS

Aula 1

Introdução a Estrutura de Dados em Python

Introdução a Estrutura de Dados em Python

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá! Nesta aula iremos explorar as estruturas de dados *built-in* e definidas pelo usuário em Python. Primeiro, veremos as sequências *built-in*: listas são coleções mutáveis e ordenadas; tuplas são como listas, mas imutáveis; dicionários armazenam pares chave-valor; e conjuntos são coleções não ordenadas e sem duplicatas. Cada estrutura tem métodos específicos, otimizando diferentes tarefas. Além disso, Python oferece vários tipos de dados como inteiros, flutuantes, *strings* e booleanos, fundamentais para manipular essas estruturas. Ao compreender essas ferramentas, você aprimora significativamente suas habilidades em Python!

Ponto de Partida

Seja bem-vindo à disciplina de Estrutura de Dados! Os computadores são máquinas de calcular capazes de receber dados, processá-los e retornar informações úteis. Por meio das linguagens de programação, desenvolvedores podem manipular o fluxo dos dados determinando, através de algoritmos e *scripts* o que se deseja. Entretanto, apenas compreender a sintaxe de uma linguagem de programação não é o suficiente, é necessário também compreender como esses dispositivos armazenam e manipulam informações. Estruturas de dados estão diretamente

ESTRUTURA DE DADOS

relacionadas à forma como dados são armazenados na memória e como o sistema pode acessá-los.

Compreender como dados são organizados e como estruturas abstratas são elaboradas pode ser essencial para solucionar problemas por métodos computacionais. Por isso, conhecer os fundamentos da estrutura de dados é de vital importância para a formação de profissionais de tecnologia da informação altamente capacitados para diversas tarefas no mercado de trabalho.

Para assimilarmos de maneira prática o conteúdo, suponhamos que você é um programador trabalhando em um sistema de gestão de bibliotecas e precisa desenvolver uma funcionalidade que armazene e gerencie informações sobre os livros e os empréstimos. Os dados dos livros devem incluir o título, autor e ano de publicação, enquanto os empréstimos precisam registrar o livro emprestado, a data de empréstimo e a data de devolução. Você deve decidir entre usar estruturas de dados *built-in* disponíveis em Python ou definir suas próprias estruturas de dados personalizadas (definidas pelo usuário).

Vamos descobrir juntos como resolver essa situação?

Bons estudos!

Vamos Começar!

Quando criamos sistemas de computador, é como montar um grande quebra-cabeça. Os desenvolvedores, como você, usam algumas técnicas especiais para tornar esse processo mais fácil. Imagine que você tem um problema complicado para resolver. Às vezes, você precisa de uma solução muito rápida, que faça as coisas bem depressa. Para isso, usamos algo chamado "algoritmos de alto desempenho". É como uma receita especial que faz seu programa rodar mais rápido. Mas nem sempre precisamos ser extremamente rápidos. Algumas vezes, o mais importante é apenas resolver o problema de forma simples e direta, sem se preocupar tanto com a velocidade.

Para que tudo isso funcione bem, os desenvolvedores precisam pensar com cuidado sobre como vão construir o sistema. É como planejar uma cidade antes de construí-la. Você precisa pensar em como as informações vão viajar pelo sistema, como serão guardadas e como podem ser encontradas facilmente quando precisarmos delas. A base para tudo isso se chama "estruturas de dados": elas são como diferentes maneiras de organizar e lidar com informações no seu programa.

Nesta aula, iremos explorar os tipos de dados que já vêm prontos no Python, uma linguagem de programação. Vamos aprender a criar estruturas chamadas "pilhas" e "filas" usando recursos que o Python já oferece. Ademais, iremos aprender a fazer nossas próprias versões dessas estruturas usando "listas encadeadas". Com isso, você vai estar pronto para criar programas mais organizados e eficientes.

Built-in e estruturas definidas pelo usuário

Pense nas estruturas de dados como diferentes caixas de ferramentas. Cada caixa tem ferramentas (ou formas) especiais para guardar e usar informações em computadores. Assim como você escolhe a ferramenta certa para um trabalho, você também precisa escolher a caixa de ferramentas certa – ou seja, a estrutura de dados certa – para cada tipo de problema no mundo da computação.

Vamos imaginar um exemplo para entender melhor. Suponha que você tenha que criar um sistema para um programa assistencial do governo, que lida com muitos cadastros de pessoas. Há tantos cadastros que é impossível analisar tudo manualmente. Além disso, algumas dessas informações precisam estar disponíveis para o público, para que as pessoas possam ajudar a identificar fraudes. Nesse caso, você tem que pensar em duas coisas importantes: quantas pessoas vão usar seu sistema e como você vai guardar e encontrar as informações de cada pessoa. Se você não escolher a caixa de ferramentas certa, ou se seu computador não for potente o suficiente, procurar por fraudes nessa montanha de dados vai ser como procurar uma agulha no palheiro.

As estruturas de dados são como mapas que mostram a melhor maneira de organizar e acessar essas informações. Escolher a estrutura de dados certa é como escolher o melhor caminho em um mapa, para que você possa guardar e encontrar as informações de forma rápida e fácil.

Sequências: listas, tuplas, dicionários e conjuntos

Vamos falar sobre como as estruturas de dados funcionam, usando um exemplo bem simples: uma lista de compras. Imagine que você está escrevendo o que precisa comprar no supermercado. Você coloca os itens na lista um após o outro. À medida que vai comprando, você vai riscando cada item. Isso é parecido com o que fazemos em programação (Lambert, 2022).

Nós vamos usar a linguagem Python para aprender sobre isso. Muitos livros usam linguagens como C, C++ ou Java, mas Python é ótimo para iniciantes, pois torna os conceitos mais fáceis de entender.

Agora, pense na sua lista de compras. Se você pegar os itens na ordem que escreveu, isso é como um tipo de lista chamado FIFO (*First-In, First-Out*), que significa "o primeiro que entra é o primeiro que sai". É como uma fila em um banco: a primeira pessoa na fila é a primeira atendida.

Mas e se você decidir começar pelas últimas coisas que escreveu? Isso é chamado LIFO (*Last-In, First-Out*), ou seja, "o último que entra é o primeiro que sai". É como uma pilha de pratos: você sempre pega o prato de cima (Goodrich; Tamassia, 2013).

Na programação, esses tipos de listas são usados para diferentes coisas. O interessante é que, mesmo que a lista seja a mesma, a maneira como usamos ela pode mudar totalmente seu tipo e sua função.

ESTRUTURA DE DADOS

Na vida real, claro, você não segue a lista à risca. Você vai andando pelo mercado, pegando o que vê pela frente. Isso é como fazer uma "busca" na sua lista, procurando visualmente cada item. Em programação, fazemos algo parecido quando precisamos encontrar algo em uma lista de dados.

Se construirmos um algoritmo para isso, teríamos algo como:

```
# olhar cada item do mercado
PARA CADA ITEM-DO-MERCADO:
    # olhar cada item da lista
    PARA CADA ITEM-DA-LISTA:
        # conferir se item do mercado está na lista
        SE ITEM-DO-MERCADO É IGUAL AO ITEM-DA-LISTA:
            # colocar item no carrinho e remover item da lista
            COLOCAR ITEM-DO-MERCADO NO CARRINHO
            REMOVER ITEM-DA-LISTA
```

Se você considerar que vai ao supermercado com uma lista pequena de compras, essa tarefa pode ser fácil e rápida. Agora, pense que você trabalha em um restaurante e precisa comprar uma lista enorme de itens em um grande hipermercado. Isso demoraria muito mais, certo? Se você ficar olhando a lista o tempo todo para cada coisa que encontra, vai acabar se confundindo mais do que se seguir a lista na ordem que escreveu.

Isso é parecido com o que acontece com um computador. Se o computador tem que trabalhar com uma lista muito grande de uma maneira complicada, ele vai gastar muito mais tempo e energia para resolver o problema. Às vezes, pode demorar dias para encontrar a solução.

Por isso, é muito importante organizar bem essa lista. Em programação, isso significa escolher a melhor maneira de estruturar os dados. Vamos ver diferentes tipos de estruturas de dados para você saber como começar a programar de maneira mais eficiente e organizada.

Armazenamento de dados

Na ciência da computação, os fundamentos operacionais envolvem três processos essenciais: entrada, processamento e saída de dados. Esses processos são realizados por meio da integração sinérgica entre componentes de hardware e software. A metodologia de armazenamento de dados desempenha um papel essencial nesse contexto. O bit, acrônimo de '*binary digit*' (dígito binário), representa a unidade primordial de armazenamento na computação.

Conceptualmente, um bit pode ser considerado como a representação mais elementar de um estado em um dispositivo eletrônico, funcionando análogo a uma chave que se encontra em um dos dois estados mutuamente exclusivos: ligado ou desligado, aberto ou fechado. Esse estado binário é representado pelos valores 0 ou 1, estabelecendo a base do sistema binário, que é fundamental na arquitetura e no funcionamento dos sistemas computacionais contemporâneos.

ESTRUTURA DE DADOS

Dentro da arquitetura de sistemas computacionais, o sistema de numeração binário opera com bytes como uma unidade fundamental. Um byte é composto por uma sequência de oito bits, possibilitando a representação de 256 valores distintos, conforme ilustrado pela expressão:

$$2^8 \text{ (isto é, } 2 \times 2)$$

$$2^8 \text{ (isto é, } 2 \times 2)$$

Esse conceito de byte é fundamental para entender como os computadores armazenam uma variedade de dados, transcendendo a mera representação numérica.

Além de números, é necessário que os computadores sejam capazes de armazenar caracteres alfanuméricos e símbolos. Consequentemente, foram estabelecidos padrões para a representação binária de *strings* de caracteres. Cada caractere é mapeado para uma sequência específica de bits. Por exemplo, nos padrões de codificação adotados por alguns modelos de computadores antigos, o caractere 'A' poderia ser representado pela sequência binária '11000000', enquanto o 'B' seria representado por '11000001'. Portanto, a palavra "AB" seria representada na forma binária como '1100000011000001'. Esses padrões de codificação são essenciais para a representação e o processamento de informações textuais em sistemas computacionais (Tangon; Santos, 2016).

DECIMAL	BINÁRIO	OCTAL	HEXADECIMAL
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Tabela 1 | Tabela de conversão entre bases: 10, 2, 8 e 16. Fonte: adaptada de Tangon e Santos (2016, p. 156).

ESTRUTURA DE DADOS

A problemática central reside na distinção entre a representação de caracteres alfabéticos, como a letra 'A', e a representação numérica em um sistema binário, como o número 192, ambos podendo ser representados pela mesma sequência binária '11000000'. A solução para esse dilema encontra-se na especificação dos tipos de dados no contexto de linguagens de programação. Linguagens de alto nível facilitam a definição do tipo de dado a ser armazenado, seja ele um número inteiro, um decimal, um conjunto de caracteres (como palavras), uma sequência de elementos, ou até mesmo um valor booleano que assume estados binários (verdadeiro ou falso, 1 ou 0, etc.).

Em linguagens de programação de alto nível de gerações anteriores, como a linguagem C, a tipificação das variáveis é uma exigência explícita na sua declaração. Essa abordagem assegura que a interpretação dos dados armazenados seja realizada de forma coerente com o tipo de dado especificado, permitindo assim que a sequência binária seja corretamente interpretada conforme o contexto de sua aplicação, seja como um caractere ou como um valor numérico.

Parte superior do formulário:

```
int x, y;  
float a, b;
```

O exemplo citado aloca espaço na memória do computador para duas variáveis inteiros (x e y) e duas variáveis de ponto flutuante (a e b). Para armazenar uma palavra, usa-se o tipo 'char', especificando-se o número máximo de caracteres permitidos para essa variável.

Para uma gestão eficiente da informação, são necessárias estruturas de dados robustas. Essas estruturas, fundamentais na ciência da computação permitem a organização, manipulação e utilização eficaz das informações. Em Python, as estruturas de dados se dividem em duas categorias principais: estruturas embutidas (*built-in* ou nativas) e estruturas definidas pelo usuário, que exigem implementação específica pelo programador (Alves, 2021).

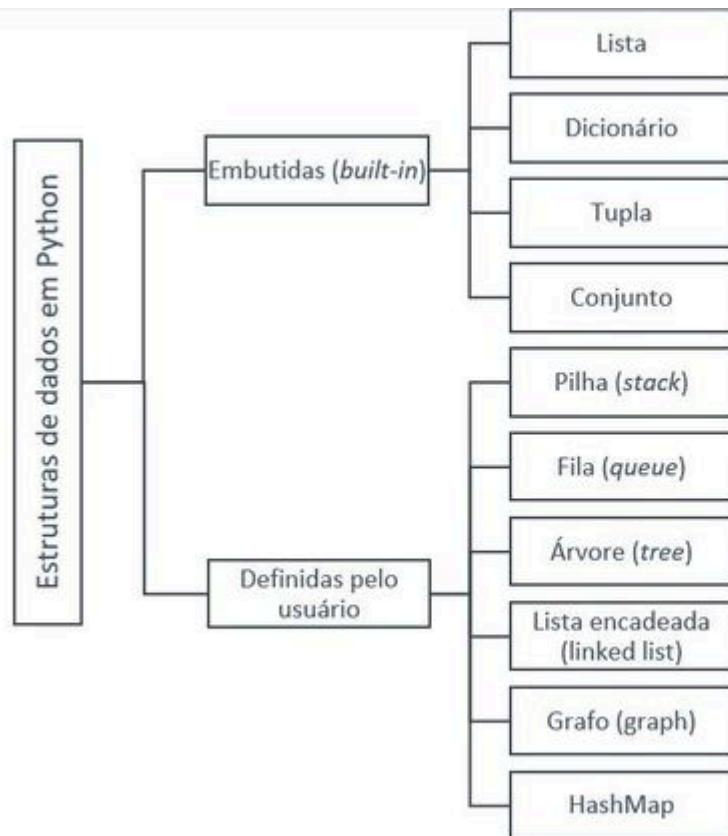


Figura 1 | Estruturas de dados em Python. Fonte: adaptada de Mariano (2021).

Python possui, de forma nativa, diversas estruturas de dados do tipo embutido (*built-in*), frequentemente chamadas de estruturas de dados sequenciais ou simplesmente sequências.

Sequências

No Python, bem como em outras linguagens de programação, as sequências são utilizadas para armazenar coleções de dados em um único objeto. Esses dados podem ser homogêneos (do mesmo tipo) ou heterogêneos (de tipos variados). Entre os tipos mais comuns de sequências estão listas, dicionários, tuplas e conjuntos (Alves, 2021).

Tipo de variável	Característica	Exemplo de declaração (variável v)
<code>list</code>	Lista de elementos. Pode receber elementos de tipos distintos, como strings,	<code>v = ["abacate", "banana", "cenoura"]</code>

	números inteiros, booleanos ou até mesmo outras listas, tuplas e dicionários.	
<i>tuple</i>	Uma tupla pode ser considerada uma lista imutável de elementos. Uma vez criada, não pode ser alterada.	v = (1, 2, 3)
<i>dictionary</i>	Um dicionário pode ser interpretado como lista composta por conjuntos de pares chave-valor.	v = { "A": "adenina", "C": "citosina", "T": "timina", "G": "guanina" }
<i>set</i>	Conjuntos de elementos únicos.	v = {1, 2, 3, 4, 5, 5, 5}

Tabela 2 | Estruturas de dados *built-in* de sequência. Fonte: adaptada de Mariano (2021).

No Python, *strings* são consideradas sequências, pois permitem o acesso a caracteres individuais. As sequências podem ser mutáveis, como listas e *arrays*, permitindo alterações, ou imutáveis, como *strings* e tuplas, que não mudam após a criação. Existem também sequências simples, que contêm itens de um tipo, e sequências contêiner, que abrigam itens de vários tipos. Sequências são úteis em programação para várias finalidades, desde listar itens similares até associar termos a definições em dicionários. Em campos específicos, como bioinformática e finanças, são usadas para mapear informações genéticas ou armazenar cotações, respectivamente. Sequências imutáveis são empregadas para representar valores constantes, como conversões de unidades (Alves, 2021).

A seguir, apresentamos como implementar os principais tipos de dados *built-in* Python.

Listas

Em Python, as listas representam uma estrutura de dados fundamental caracterizada por sua natureza ordenada e mutável. Essas estruturas são definidas pela capacidade de armazenar uma

coleção de elementos, que podem ser de tipos variados, incluindo números, *strings* e até outras listas, em uma única variável. A ordenação das listas implica que cada elemento é associado a um índice, começando do zero, permitindo assim o acesso e a manipulação eficiente dos dados (Alves, 2021).

Exemplos:

1. **Criação de uma Lista:** a lista a seguir armazena uma sequência de números inteiros.

```
lista_numerica = [1, 2, 3, 4, 5]
```

2. **Heterogeneidade dos elementos:** esta lista contém uma combinação de *strings*, números de ponto flutuante, inteiros e valores booleanos.

```
lista_mista = ["Python", 3.7, 2023, False]
```

3. **Acesso a elementos:** acessar o elemento na primeira posição, que retorna o valor 1.

```
primeiro_elemento = lista_numerica[0]
```

4. **Alteração de elementos:** modificar um item específico, resultando na alteração do segundo elemento para 10.

```
lista_numerica[1] = 10
```

5. **Inserção de elementos:** utilizando o método a seguir, adiciona o número 6 ao final da lista.

```
lista_numerica.append(6)
```

6. **Remoção de elementos:** o comando a seguir exclui o número 3 da lista.

```
lista_numerica.remove(3)
```

7. **Determinação do tamanho da lista:** fornece o número total de elementos na lista.

```
len(lista_numerica)
```

8. **Listas aninhadas:** criação de listas dentro de listas.

```
lista_aninhada = [[1, 2], [3, 4, 5]]
```

Tuplas

Uma tupla em Python, conhecida como tuple, é uma coleção ordenada de elementos que é imutável, ou seja, não pode ser alterada após sua criação. Os elementos de uma tupla são identificados e acessados através de suas posições. As tuplas podem ser criadas colocando os elementos entre parênteses ou simplesmente separando-os com vírgulas.

Exemplos de tuplas:

1. **Criação com parênteses:** aqui, `minha_tupla` é uma tupla contendo três números inteiros.

```
minha_tupla = (1, 2, 3)
```

2. **Criação sem parênteses:** esta também é uma tupla válida, contendo três números.

```
outra_tupla = 4, 5, 6
```

Dicionários

Dicionários em Python, conhecidos como *dict*, são estruturas de dados que armazenam pares de chave-valor. Eles são coleções não ordenadas, mas são mutáveis, permitindo que os dados sejam modificados após a criação. Cada chave no dicionário é única e é usada para acessar o valor correspondente. Dicionários são ideais para casos em que as relações entre os dados são essenciais (Alves, 2021).

Considere o seguinte exemplo acadêmico:

```
dicionario_curso = {"nome": "Ciência da Computação", "créditos": 240}
```

Aqui, `dicionario_curso` é um dicionário que armazena informações sobre um curso. A chave "nome" está associada ao valor "Ciência da Computação", e a chave "créditos" ao valor 240.

Podemos ainda usar os métodos `keys()` e `values()` para coletar chave e valor, respectivamente:

```
# Criando um dicionário
info_estudante = {
```

```
}
```

```
# Acessando as chaves do dicionário
```

```
chaves = info_estudante.keys()
```

```
print("Chaves:", chaves) # Saída: Chaves: dict_keys(['nome', 'idade', 'curso'])
```

```
# Acessando os valores do dicionário
```

```
valores = info_estudante.values()
```

```
print("Valores:", valores) # Saída: Valores: dict_values(['Ana Silva', 22, 'Engenharia de Software'])
```

Conjuntos

Conjuntos em Python, conhecidos como set, são coleções que armazenam elementos únicos e não ordenados. Diferente das listas e tuplas, os conjuntos não permitem valores duplicados, e a ordem dos elementos não é garantida. Eles são particularmente úteis para operações como união, interseção e diferença, comuns em lógica de conjuntos matemáticos.

1. Criação de um conjunto:

```
meu_conjunto = {1, 2, 3, 4, 5}
```

Aqui, **meu_conjunto** é um conjunto com números inteiros de 1 a 5

2. Adicionando elementos:

```
meu_conjunto.add(6)
```

Isso adiciona o número 6 ao conjunto, caso ele ainda não esteja presente.

3. Removendo elementos:

```
meu_conjunto.remove(2)
```

Remove o número 2 do conjunto. Se o número 2 não estiver no conjunto, isso causará um erro.

4. Operações de conjunto:

assim como as operações matemáticas, podemos realizar operações como a união, interseção, etc.

- União:

```
conjunto_a = {1, 2, 3}  
conjunto_b = {3, 4, 5}  
uniao = conjunto_a.union(conjunto_b)  
print(uniao) # Saída: {1, 2, 3, 4, 5}
```

Siga em Frente...

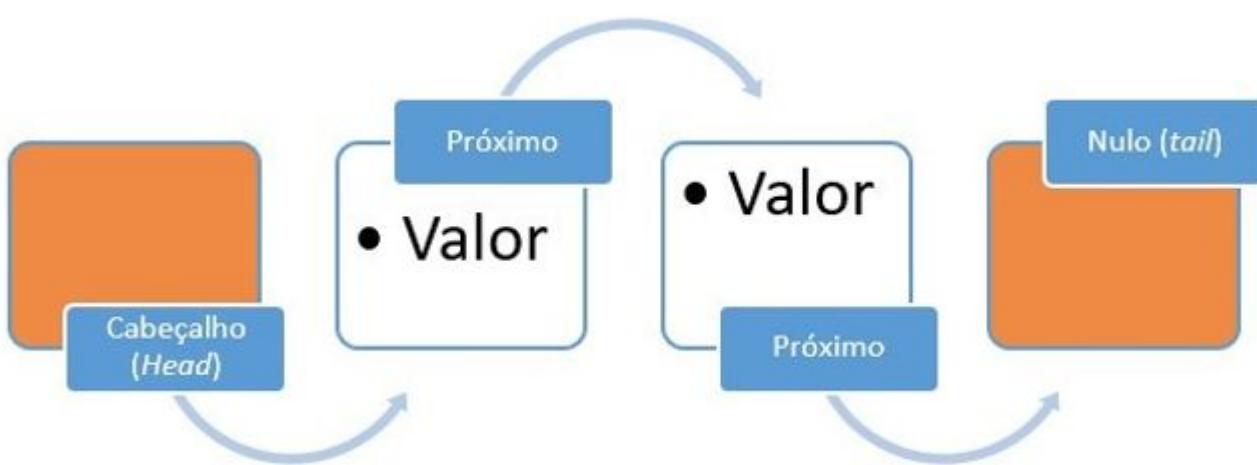
Tipos de dados em Python

Python distingue-se de outras linguagens de programação por sua sintaxe intuitiva e uma ampla gama de funcionalidades pré-construídas, incluindo várias estruturas de dados. No entanto, a linguagem também permite que os usuários criem suas próprias estruturas de dados personalizadas.

Conforme apontado por Alves (2021), existem seis tipos principais de estruturas de dados definidas pelo usuário que podem ser implementadas em Python. Essas estruturas incluem listas encadeadas, pilhas, filas, árvores, grafos e *hashmap*. Nas próximas aulas, esses conceitos serão detalhadamente introduzidos e discutidos, fornecendo uma base sólida para o entendimento e a aplicação dessas estruturas ao longo do livro.

Listas encadeadas, pilhas, filas, árvores e grafos são estruturas de dados fundamentais na computação, cada uma com suas características e usos específicos.

1. **Listas encadeadas:** são coleções de elementos chamados nós, onde cada nó está conectado sequencialmente ao próximo por meio de "ponteiros". Eles oferecem flexibilidade na inserção e remoção de elementos, já que não exigem realocação de outros elementos, ao contrário dos *arrays*.



ESTRUTURA DE DADOS

Figura 2 | Ilustração de uma lista encadeada . Fonte: adaptada de Mariano (2021).

2. **Pilhas:** funcionam com o princípio de "último a entrar, primeiro a sair" (LIFO). São usadas em diversas aplicações, como na execução de algoritmos de busca em profundidade e na gestão de tarefas em sistemas operacionais.

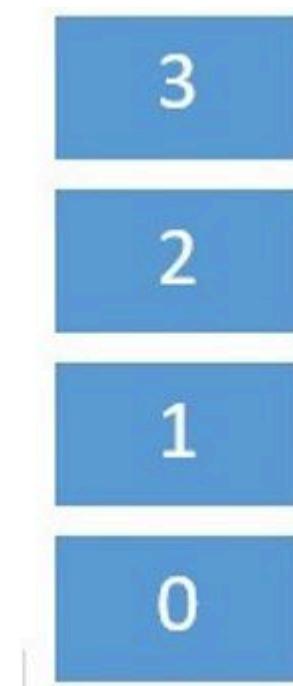


Figura 3 | Exemplo de uma pilha composta por quatro valores (de 0 a 3). Fonte: adaptada de Mariano (2021).

3. **Filas:** operam no princípio de "primeiro a entrar, primeiro a sair" (FIFO). São úteis em situações que exigem processamento sequencial, como na gestão de filas de impressão ou em algoritmos de busca em largura.

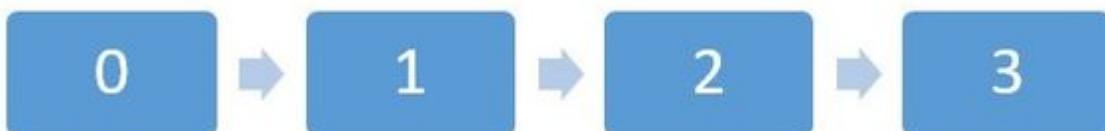


Figura 4 | Estruturas de dados definidas pelo usuário: fila. Fonte: adaptada de Mariano (2021).

4. **Árvores:** estruturas hierárquicas que começam com um nó raiz, de onde se ramificam nós filhos. São cruciais em aplicações como a organização de sistemas de arquivos, implementação

de bases de dados e algoritmos de busca.

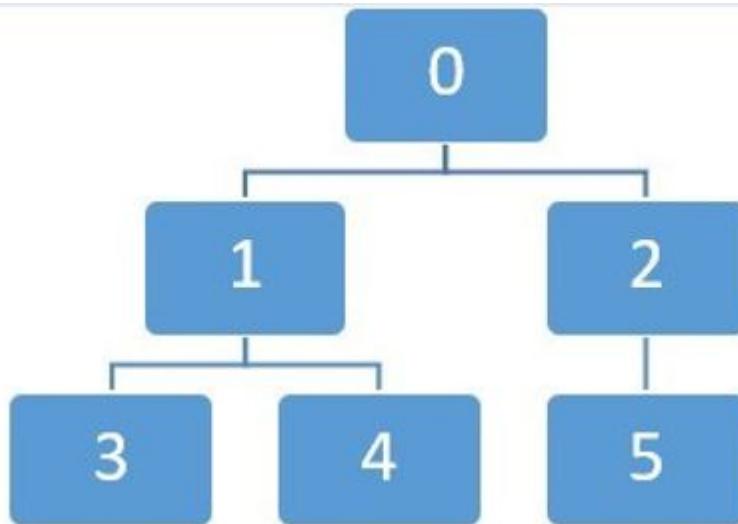


Figura 5 | Estruturas de dados definidas pelo usuário: árvore. Fonte: adaptada de Mariano (2021).

5. **Grafos:** consistem em nós (ou vértices) que podem ser conectados por arestas. Essenciais para representar relações complexas, os grafos são amplamente utilizados em redes sociais, sistemas de mapas e algoritmos de roteamento.

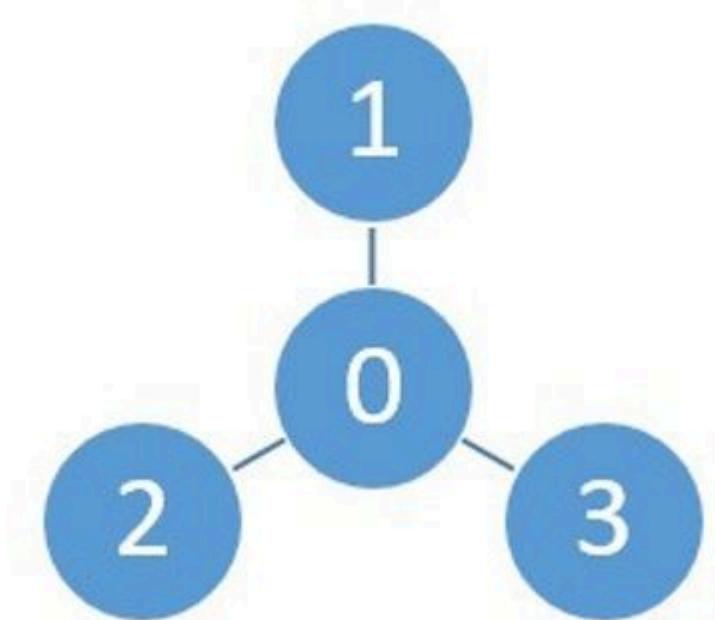


Figura 6 | Estruturas de dados definidas pelo usuário: grafo. Fonte: adaptada de Mariano (2021).

6. **Hashmap:** também conhecido como tabela de dispersão, o *hashmap* é uma estrutura de dados que armazena pares de chave-valor. Ele oferece acesso extremamente rápido aos

elementos, pois utiliza uma função de *hash* para converter a chave em um índice, onde o valor correspondente é armazenado. *Hashmaps* são ideais para situações onde é necessário um acesso rápido aos dados, como em caches, bases de dados e na implementação de conjuntos. Eles são eficientes tanto para inserção quanto para busca e remoção de elementos, tornando-os uma ferramenta poderosa para manipulação de dados em larga escala (Szwarcfiter; Markenzon, 2020).

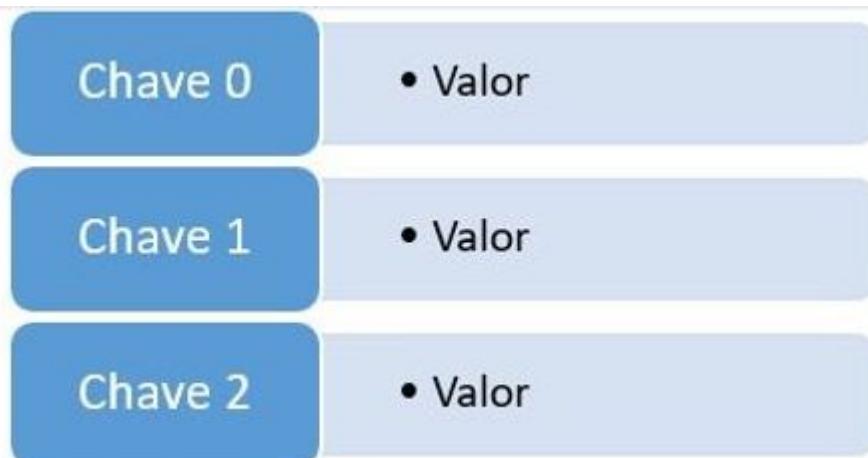


Figura 7 | Estruturas de dados definidas pelo usuário: hashmap. Fonte: adaptada de Mariano (2021).

Cada uma dessas estruturas tem características únicas que as tornam adequadas para resolver diferentes tipos de problemas no campo da computação.

Vamos Exercitar?

Você foi apresentado à seguinte problemática:

Você é um programador trabalhando em um sistema de gestão de bibliotecas e precisa desenvolver uma funcionalidade que armazene e gerencie informações sobre os livros e os empréstimos. Como você poderia construir uma solução em Python para tal?

Para armazenar informações sobre os livros, você pode utilizar um dicionário *built-in* em Python, pois ele permite armazenar pares de chave-valor, facilitando a associação dos dados do livro com valores específicos.

```
# Usando um dicionário built-in para armazenar informações do livro
livros = {
    'ISBN1': {'título': 'A Arte da Guerra', 'autor': 'Sun Tzu', 'ano': 500},
    'ISBN2': {'título': '1984', 'autor': 'George Orwell', 'ano': 1949}
}
```

ESTRUTURA DE DADOS

Para gerenciar os empréstimos, você poderia definir uma classe personalizada, pois os empréstimos podem envolver operações mais complexas que beneficiariam de métodos específicos, como a atualização das datas de empréstimo e devolução ou o cálculo de multas por atraso.

```
# Definindo uma classe personalizada para empréstimos
class Emprestimo:
    def __init__(self, livro, data_emprestimo, data_devolucao):
        self.livro = livro
        self.data_emprestimo = data_emprestimo
        self.data_devolucao = data_devolucao
# Instanciando objetos da classe Emprestimo
emprestimo1 = Emprestimo(livros['ISBN1'], '2023-01-01', '2023-01-15')
emprestimo2 = Emprestimo(livros['ISBN2'], '2023-02-01', '2023-02-15')
```

A escolha entre estruturas de dados *built-in* e personalizadas depende das necessidades específicas do sistema. No exemplo da biblioteca:

- Os **dicionários *built-in*** são adequados para armazenar informações dos livros, pois oferecem uma maneira simples e eficiente de acessar e manipular dados associativos sem a necessidade de funcionalidades adicionais.
- As **estruturas de dados definidas pelo usuário**, como a classe **Emprestimo**, fornecem maior flexibilidade para incorporar lógicas de negócios complexas, como métodos para manipular as datas dos empréstimos e calcular atrasos, o que seria mais complicado de implementar com estruturas de dados *built-in*.

Portanto, a decisão foi baseada na complexidade dos dados e operações necessárias: o uso pragmático de dicionários *built-in* para informações simples de livros e a criação de uma classe personalizada para gerenciar a lógica específica de empréstimos.

Saiba mais

- [Builtin e estruturas definidas pelo usuário](#);
- [Sequências: listas, tuplas, dicionários e conjuntos](#);

Tipos de dados em Python: você pode jogar o jogo CodeBo para aprender sobre estruturas como Pilhas.

- [Medium Educação & Comunicação](#);
- [CodeBo](#).

Referências

ESTRUTURA DE DADOS

ALVES, W. P. **Programação Python:** aprenda de forma rápida. São Paulo: Expressa, 2021.

GOODRICH, M. T.; TAMASSIA, R. **Estruturas de dados e algoritmos em java.** 5. ed. Porto Alegre: Bookman, 2013.

LAMBERT, K. A. **Fundamentos de Python:** estruturas de dados. São Paulo: Cengage Learning, 2022.

MARIANO, D. C. B. **Introdução à estrutura de dados em Python.** Estrutura de Dados, 2021. Disponível em: <https://tinyurl.com/y8uhm5uz>. Acesso em: 6 dez. 2023.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos.** 3. ed. Rio de Janeiro: LTC, 2020.

TANGON, L.; DOS SANTOS, R. C. **Arquitetura e organização de computadores.** 1. ed. Londrina: Editora e Distribuidora Educacional S.A., 2016.

Aula 2

Listas Encadeadas

Listas Encadeadas

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá! Bem-vindo à aula sobre listas encadeadas em Python! Listas encadeadas são coleções de elementos onde cada um aponta para o próximo, proporcionando flexibilidade na inserção e deleção de itens. Nesta aula, iremos implementar uma lista encadeada básica, entender como percorrê-la para buscar elementos e aprender a inserir e deletar itens eficientemente. Essas operações são essenciais para otimizar o desempenho em situações onde listas comuns seriam menos eficazes. Vamos mergulhar no código e explorar essas estruturas dinâmicas!

Ponto de Partida

As listas constituem uma estrutura fundamental não apenas em atividades cotidianas, mas também no contexto da ciência da computação. Frequentemente, indivíduos interagem com diversos arranjos de listas, como registros de endereços de clientes, listagens de passageiros de voos, ou até mesmo lista de compras. No domínio computacional, as listas adquirem uma dimensão crítica para a modelagem e solução de problemas algorítmicos.

No âmbito das estruturas de dados, uma lista é uma entidade que permite o armazenamento e a organização de dados de maneira sequencial. Em Python, por exemplo, a implementação intrínseca de uma lista pode ser realizada através da função `list()` ou mediante a inclusão de elementos entre colchetes. Essa linguagem de programação oferece uma série de métodos nativos para a manipulação de listas que facilitam operações como a inserção, atualização e remoção de elementos (Lambert, 2022).

Apesar da simplicidade proporcionada pelas estruturas de dados nativas de Python, existem variantes mais complexas de listas que não são intrínsecas à linguagem, como as listas encadeadas, também conhecidas como listas ligadas: imagine uma fila de pessoas onde cada pessoa segura um papel com seu próprio nome e uma seta apontando para a próxima pessoa na fila. Em computação, temos algo parecido chamado lista encadeada. Cada pessoa é como um "nó" que sabe quem vem depois dela na fila. Em uma lista encadeada, cada nó tem uma informação e um ponteiro que indica quem é o próximo nó. Se precisarmos tirar alguém do meio da fila, como a pessoa número 56 de uma fila de 100, não precisamos reorganizar toda a fila. Em vez disso, apenas dizemos para a pessoa número 55 que agora a pessoa número 57 está depois dela. É uma mudança simples que economiza muito tempo, pois não precisamos atualizar a posição de cada pessoa na fila.

Estruturas definidas dessa forma exigem uma implementação manual para sua utilização, dada a necessidade de funcionalidades específicas que transcendem as capacidades das listas padrão (Szwarcfiter; Markenzon, 2020). Parte superior do formulário.

Nesta aula, você aprenderá a implementar as listas encadeadas usando conceitos de orientação a objetos com Python. Para assimilar seu aprendizado, suponhamos que você é um desenvolvedor de software em uma clínica médica que está migrando os registros de saúde dos pacientes de um sistema baseado em papel para um sistema eletrônico.

A clínica deseja um método eficiente para acessar e atualizar os históricos de saúde dos pacientes, que incluem diagnósticos, tratamentos e consultas anteriores. Para atender a essa necessidade, você decide utilizar listas encadeadas em Python para manter os registros médicos. Os prontuários devem ser rapidamente acessíveis e facilmente atualizáveis cada vez que um paciente visita a clínica.

Sua tarefa é:

1. Implementar uma lista encadeada para armazenar os prontuários eletrônicos.

2. Permitir inserções eficientes de novos registros de saúde no histórico do paciente.
3. Facilitar a busca e atualização de prontuários existentes.

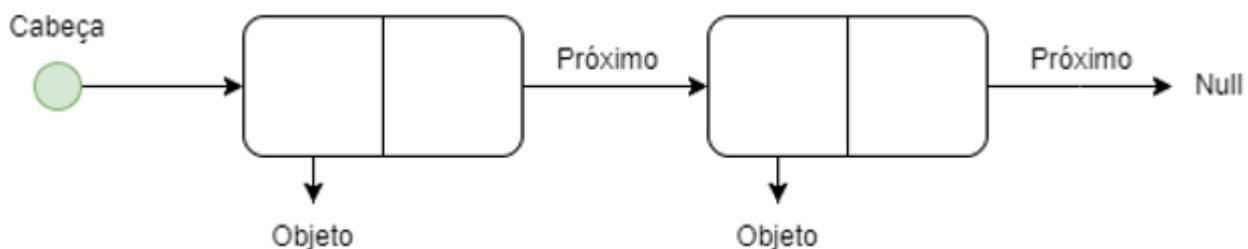


Figura 1 | Lista encadeada. Fonte: elaborada pela autora.

Bons estudos!

Vamos Começar!

Listas são uma parte integral de nossas vidas cotidianas e do mundo da computação. Por exemplo, muitos de nós temos listas de tarefas domésticas pendentes ou uma pilha de exercícios acumulados (Lambert, 2022). Da mesma forma, no campo da computação, as listas desempenham um papel essencial. Quando você inicia seu computador, o sistema operacional carrega uma lista de aplicativos, organizando-os com base em suas prioridades para acessar recursos específicos. Seu aplicativo de música favorito, por exemplo, mantém uma lista de músicas que reflete suas preferências pessoais, permitindo que você navegue, repita suas músicas preferidas e pule as que não deseja mais ouvir. Além disso, alguns players de música modernos empregam inteligência artificial para criar listas de reprodução personalizadas, baseando-se nos gostos de usuários com preferências semelhantes às suas. Esses exemplos ilustram claramente a importância das listas na modelagem e solução de problemas no âmbito da tecnologia da informação (Cury *et al.*, 2018).

Em programação, a estrutura de dados conhecida como lista é caracterizada por uma agregação sequencial de itens, cada um dos quais pode ser recuperado, atualizado ou removido. A identificação individual de cada item em uma lista é comumente realizada por meio de identificadores únicos (Lambert, 2022). Consideremos, por exemplo, a adição de duas versões de uma mesma música em uma lista de reprodução, uma ao vivo e outra de estúdio. Embora ambas compartilhem o mesmo nome, suas características diferem significativamente, como a presença de interações com o público na versão ao vivo. O emprego de identificadores únicos facilita a distinção e remoção seletiva de versões específicas da música.

Em termos práticos, os identificadores mais comuns são índices numéricos. Nas linguagens de programação, esses índices geralmente começam a partir do zero, designando a posição sequencial dos elementos dentro da lista (por exemplo, o primeiro elemento está na posição 0, o segundo na posição 1 e assim por diante).

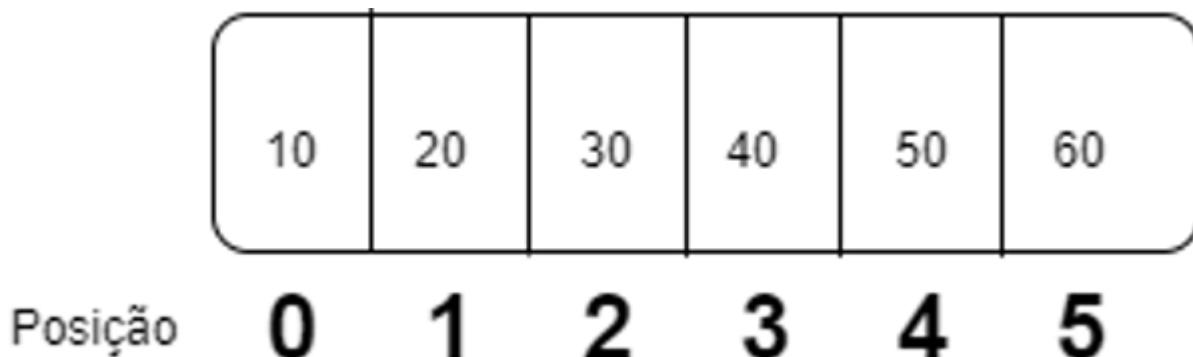


Figura 2 | Sequência lista. Fonte: elaborada pela autora.

No contexto da linguagem de programação Python, as listas são implementadas de maneira particularmente clara e acessível. Os elementos dentro de uma lista podem ser facilmente acessados utilizando seus índices específicos. Contudo, operações como a remoção de elementos da lista podem exigir ajustes nos índices dos elementos restantes, apresentando um desafio em termos de gerenciamento eficiente dessa estrutura de dados.

Vamos supor que desejemos remover o número 30 da lista:



Figura 3 | Número removido da lista. Fonte: elaborada pela autora.

Note que na nossa lista, o número 30 está na posição 2. Quando removemos o número 30, todos os índices subsequentes precisam ser atualizados: o número 40, que antes estava na posição 3, agora ocupa a posição 2, o número 50 move-se para a posição 3, e assim por diante, resultando na eliminação da posição 5 anterior.

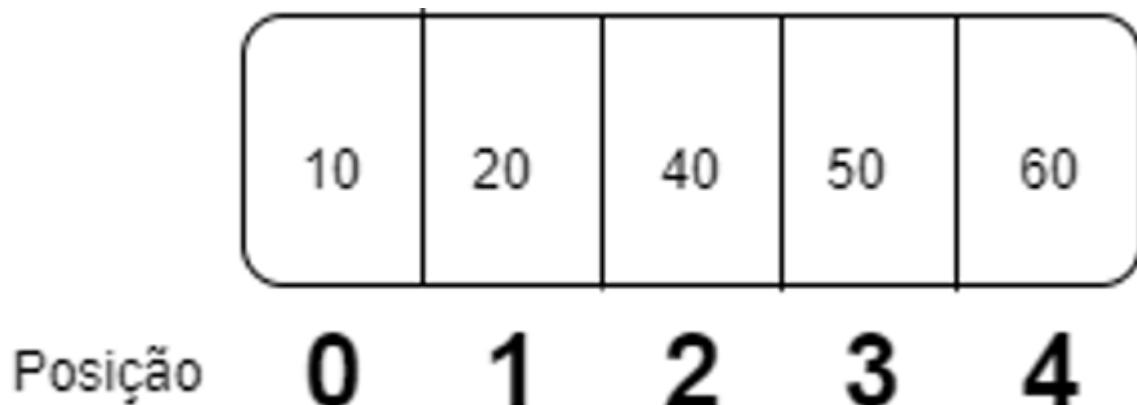


Figura 4 | Novas posições dos valores presentes na lista. Fonte: elaborada pela autora.

A remoção de um único elemento em uma lista no Python requer a reorganização de todos os índices subsequentes, o que evidencia uma limitação dessa estrutura de dados. Uma alternativa para superar essa desvantagem é o uso de **listas encadeadas**.

Nas listas encadeadas, também conhecidas como listas ligadas, os dados são armazenados em nós. Cada nó contém um valor e um ponteiro para o próximo nó na lista. Dessa forma, a posição de um item não depende de toda a lista, mas sim do item que o segue. Para navegar pela lista encadeada, começa-se pelo primeiro elemento, chamado de cabeça (*head*), seguindo os ponteiros de um nó para o outro até alcançar um valor nulo, que indica o fim da lista, conhecido como cauda (*tail*).

A Figura 5 ilustra uma lista ligada tradicional, isto é, uma lista linear. Nesse caso, cada nó da lista encadeada é composto por um valor de informação (no desenho ilustrado pelo bloco *info*) e por um bloco que indica o próximo item da lista (ilustrado por *next*).

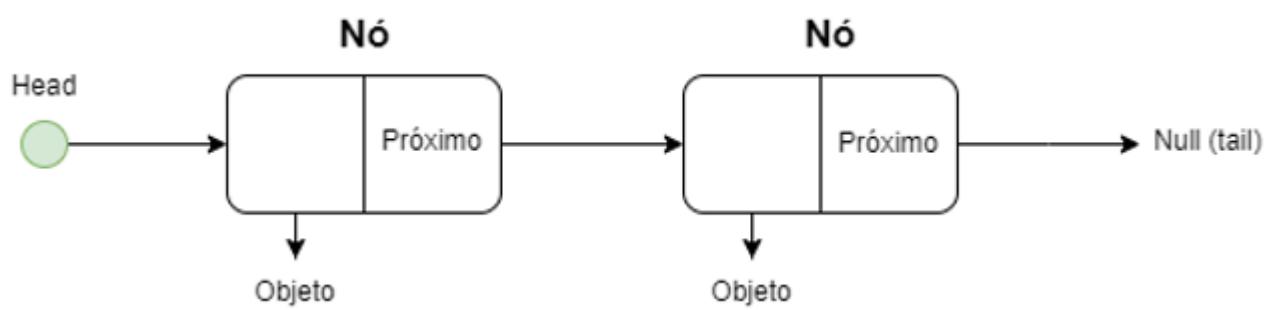


Figura 5 | Ilustração de uma lista ligada linear. Fonte: elaborada pela autora.

Adicionalmente, as listas encadeadas podem ser configuradas de modo que não tenham um "último item" definido. Nessa variação, o nó que seria o último na lista aponta de volta para o primeiro nó, criando um ciclo contínuo. Esse tipo de estrutura é conhecido como **lista circular** (Alves, 2021).

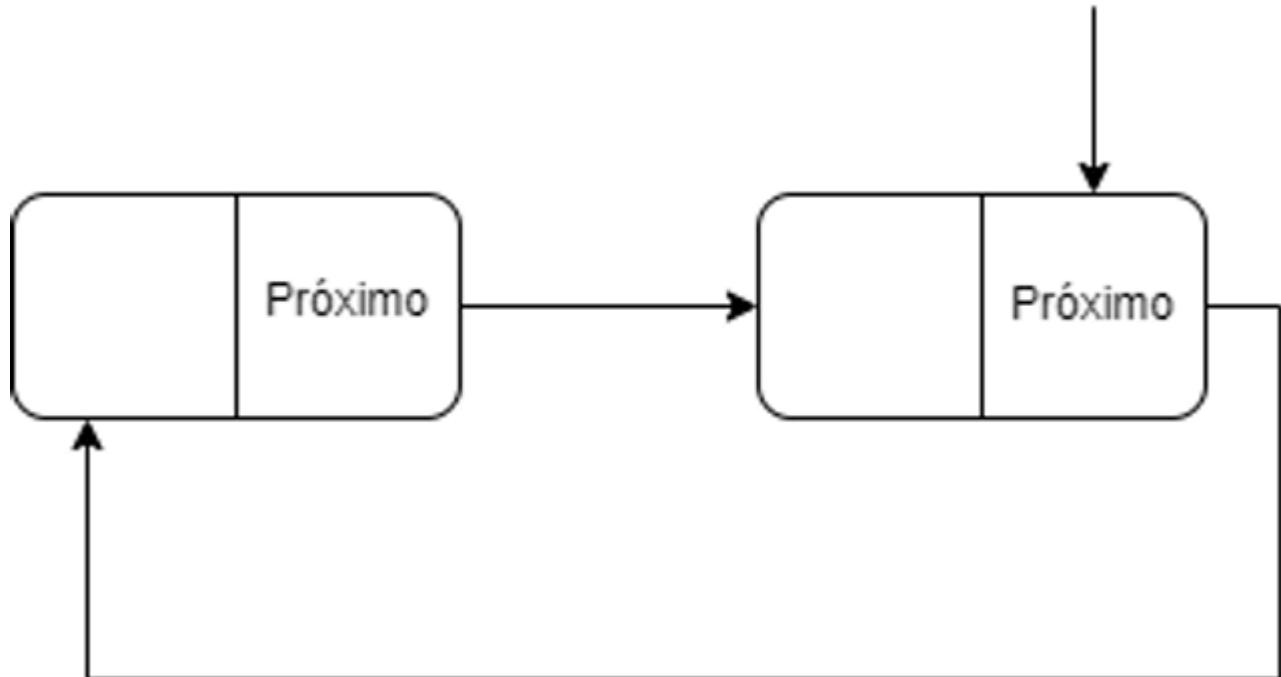


Figura 6 | Esquema de uma lista circular. Fonte: elaborada pela autora.

Siga em Frente...

Implementação e busca em listas encadeadas em Python

Agora, está na hora de implementar uma lista encadeada utilizando Python. Para isso, crie um arquivo chamado `ListaEncadeada.py` (é importante manter esse nome, já que planejamos importá-lo em outro arquivo posteriormente). Dentro desse arquivo, iremos adicionar as classes necessárias para representar nossa lista encadeada. Comece desenvolvendo uma classe específica para representar cada item que será incluído na lista encadeada (Mariano, 2021).

```
class ItemLista:  
    def __init__(self, data=0,  
                 nextItem=None):  
        self.data = data  
        self.next =  
            nextItem  
  
    def __repr__(self):  
        return '%s => %s' %  
            (self.data,  
             self.next)
```

A classe contém dois métodos essenciais:

- `__init__`: comumente referido como "*dunder init*" ou "*double underscore data*", que armazena o valor a ser incluído na lista, e `nextItem`, que guarda a referência ao próximo item na lista encadeada.
- `__repr__`: fornece uma representação em *string* do objeto, exibindo o item adicionado e o próximo item na sequência.

Vamos agora desenvolver a classe que representará a lista encadeada. Essa classe é essencial para criar uma nova instância de lista encadeada.

```
class ListaEncadeada:  
    def __init__(self):  
        self.head = None  
  
    def __repr__(self):  
        return "%s" %  
            (self.head)
```

Quando essa classe é instanciada, um único objeto, chamado de '*head*', é inicializado. Esse objeto '*head*' serve como indicador do primeiro item na lista encadeada. Ademais, na classe '*ListaEncadeada*', implementaremos métodos adicionais que facilitarão a inserção, remoção e localização de elementos na lista.

Busca de itens em uma lista encadeada

Imagine uma lista encadeada como uma linha de pessoas segurando as mãos, onde cada pessoa representa um item da lista. A primeira pessoa é o objeto "*head*".

Esse processo é a busca em uma lista encadeada. Você começa no início e segue os links ("mãos") de um item para outro até encontrar o que procura.

Para entendermos melhor, utilizando nosso exemplo prático, precisamos primeiro entender a adição e a remoção de itens na lista. Voltaremos com o código da busca posteriormente.

Inserção e deleção de itens

Inserindo itens na lista encadeada (`append`)

Pode-se adicionar um novo item na lista por meio de um método inserido na classe *ListaEncadeada*. Vamos denominar esse método como `insere()`, que receberá dois parâmetros: `lista` (que irá receber o objeto da lista encadeada) e `data` (que receberá o dado que será inserido).

```
def insere(self, lista, data):
    # cria um objeto para
    # armazenar um novo item
    # da lista
    item = ItemLista(data)
    # o head é apontado como
    # próximo item
    item.next = lista.head
    # o item atual se torna o
    # head
    lista.head = item
```

Esse método aciona a classe '*ItemLista*', passando o dado como argumento e armazenando o resultado na variável '*item*'. Em seguida, o atributo '*next*/*item*' do novo item é configurado para apontar para o atual primeiro item da lista (o '*head*'), e então o '*head*' da lista é atualizado para ser o novo item. Isso efetivamente coloca o novo item no início da lista.

Para exemplificar, vamos aplicar esse código em uma situação do dia a dia. Considere que você está usando a estrutura de lista encadeada para organizar uma lista de compras no supermercado. Você adiciona itens à sua lista que deseja comprar, como shampoo, biscoito, detergente, abobrinha e banana. Cada item é adicionado individualmente e em uma ordem específica, embora possam ser removidos em qualquer ordem durante as compras (Mariano, 2021).

Agora, vamos à implementação prática dessa lista. Antes de prosseguir, é importante verificar se o script está funcionando corretamente. Para isso, criaremos um novo arquivo Python chamado *processa.py*. É essencial que este arquivo esteja no mesmo diretório que o arquivo da lista encadeada para garantir que funcionem juntos corretamente.

O código a seguir mostra como importar o módulo Python criado anteriormente. Em seguida, uma nova lista é instanciada. Veja:

```
import ListaEncadeada as le
# Cria o objeto
lista = le.ListaEncadeada()
print("Conteúdo da lista:",
      lista) # lista está vazia
```

Note que importamos o arquivo *ListaEncadeada* (não é necessário incluir a extensão ".py") e atribuímos a ele um apelido (*le*) para facilitar a utilização dos métodos e das classes desse módulo. Ao imprimir o conteúdo da variável criada, a saída será "Conteúdo da lista: *NoneNone*".

Agora, vamos proceder com a adição de alguns itens na lista:

```
# Exemplo de uso
le.ListaEncadeada.insere(l
ista, "shampoo")
le.ListaEncadeada.insere(l
ista, "biscoito")
le.ListaEncadeada.insere(l
ista, "detergente")
le.ListaEncadeada.insere(l
ista, "abobrinha ")
le.ListaEncadeada.insere(l
ista, "banana")
print(lista)
```

Saída:

Saída do programa:

banana => abobrinha => detergente => biscoito => shampoo => None

Figura 7 | Saída do programa. Fonte: elaborada pela autora.

Vamos agora entender o passo a passo do que ocorre quando executamos o código apresentado de criação e adição à nossa lista encadeada:

Quando chamamos:

```
# Cria o objeto
lista =
le.ListaEncadea
da()
```

A lista é criada vazia. De fato, ao chamar o construtor da classe `ListaEncadeada`, o nosso `Head` recebe o valor `None`:

```
def
__init__(self):
    self.he
    ad =
    None
```

Portanto, temos uma representação conforme a Figura a seguir:

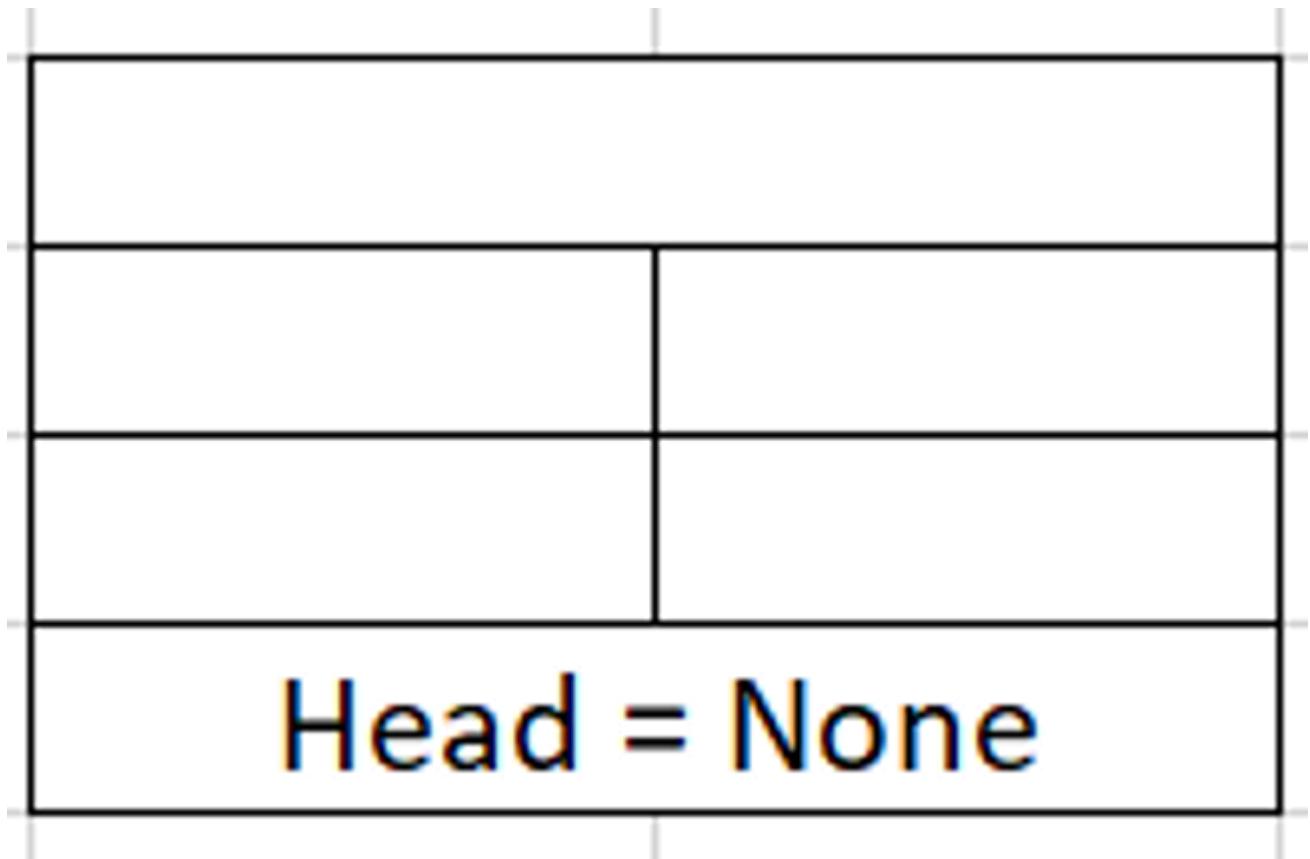


Figura 8 | Representação da lista ao iniciar o código. Fonte: elaborada pela autora.

Note que, atualmente, a representação da lista só mostra o valor do *head*, sem outros dados relevantes. Vejamos o que ocorre ao adicionar o primeiro item à lista (conforme ilustrado a seguir). É necessário utilizar o "apelido" previamente definido para o módulo ListaEncadeada.py (le), seguido pelo nome da classe e do método a ser acessado:

```
# Exemplo de uso  
le.ListaEncadeada.insere  
(lista, "shampoo")
```

Objeto "shampoo"	
Valor (data)	Próximo (next)
shampoo	None
Head = Objeto "shampoo"	

Figura 9 | Primeiro item inserido. Fonte: elaborada pela autora.

Observe que, na representação da lista, agora temos um item: o objeto "shampoo". Esse objeto tem dois atributos: o valor (*data*) e o próximo item (*next*). O valor é a *string* fornecida, neste caso, "shampoo". O próximo item, nesse momento, é o que era anteriormente o valor de *head*, isto é, *None* (já que temos apenas um item na lista). Além disso, o cabeçalho da lista é atualizado para armazenar o objeto "shampoo".

Agora veja o que acontece quando inserimos um novo item:

```
le.ListaEncadeada.insere
(lista, "biscoito")
```

Objeto "biscoito"		Objeto "shampoo"	
Valor (data)	Próximo (next)	Valor (data)	Próximo (next)
biscoito	Objeto "shampoo"	shampoo	None
Head = Objeto "biscoito"			

Figura 10 | Segundo item inserido. Fonte: elaborada pela autora.

O mesmo procedimento é realizado para os outros comandos. Por exemplo, após a inserção do demais itens da lista.

Note a distinção entre referir-se a um objeto e a um valor específico dentro desse objeto. Tomando o objeto "shampoo" como exemplo, ele é formado por um atributo de valor (a variável *data*), que armazena a *string* "shampoo", e um atributo para o próximo item (a variável *next*). No entanto, como "shampoo" está na cauda da lista, *next* recebe o valor *None*, indicando que não existe um próximo item na sequência.

Além disso, é importante observar que as inserções na lista encadeada são sempre realizadas no início. Embora seja possível implementar uma função para inserir itens no final da lista, esse processo requer a verificação da existência de itens já inseridos na lista (Backes, 2023).

Deleção de itens

A seguir, vamos desenvolver um método responsável pela remoção de itens da lista. Vejamos como deve ser a implementação desse método de remoção, que será incluído na classe 'ListaEncadeada':

```
def remove(lista, valor):
    # Verifica se o item a ser
    # removido é o head
    if lista.head and
        lista.head.data == valor:
```

```
lista.head =  
lista.head.next  
else:  
    # Detecta a posição do  
    # elemento  
    before = None  
    navegar = lista.head  
    # Navega pela lista para  
    encontrar o elemento  
  
    while navegar and  
        navegar.data != valor:  
        before = navegar  
        navegar = navegar.next  
    #print(navegar.data)  
    # Remove o item se ele for  
    encontrado  
    if navegar:  
        before.next = navegar.next
```

O método é designado para remover um valor específico da lista e, para isso, percorre toda a lista em busca desse valor. Primeiramente, verifica-se se o valor a ser removido é o que está no “Head” da lista, o que simplifica o processo de remoção. Caso contrário, o método procura pelos elementos adjacentes ao item desejado, iniciando a busca a partir do “Head”. É importante notar que apenas o ponteiro do item anterior ao valor removido é modificado para apontar para o item subsequente, eliminando a necessidade de reorganizar toda a lista e afetando apenas os elementos diretamente relacionados à remoção.

Em seguida, apresentamos um exemplo prático de como realizar a deleção de um item na lista:

```
Saida do programa:  
  
banana => abobrinha => detergente => biscoito => shampoo => None  
  
-----  
Deletar 'abobrinha':  
  
banana => detergente => biscoito => shampoo => None
```

Figura 11 | Exemplo de remoção. Fonte: elaborada pela autora.

Busca de itens em uma lista encadeada

Agora que já abordamos como criar uma lista encadeada e inserir itens nela, o próximo passo é aprender a realizar buscas nessa lista. É importante destacar que, em listas encadeadas, as operações de busca podem ser relativamente custosas, pois exigem a navegação pela estrutura completa da lista para encontrar o elemento desejado.

Na classe **ListaEncadeada** do arquivo **ListaEncadeada.py** deve ser adicionado o código:

```
def busca(lista, valor):
    navegar = lista.head
    while navegar and
        navegar.data != valor:
        navegar =
            navegar.next
    return navegar
```

Nessa função, a lista e um valor específico são recebidos como parâmetros para realizar a busca. Primeiramente, o cabeçalho atual da lista é capturado e armazenado na variável **navegar**. Essa variável será percorrida até que se encontre o valor **None** ou até que o valor armazenado em *data* seja igual ao valor buscado. Caso uma dessas condições seja satisfeita, o objeto correspondente é retornado. Observe um exemplo de como executar a chamada ao método “busca”:

```
print(" ")
query = "biscoito"
item_buscado =
le.ListaEncadeada.busca(lista,
query)
if item_buscado:
    print("Elemento
        encontrado")
else:
    print("Elemento nao
        encontrado")
```

Saida do programa:

```
banana => abobrinha => detergente => biscoito => shampoo => None
```

```
-----
```

Elemento encontrado

ESTRUTURA DE DADOS

Figura 12 | Execução do código de busca. Fonte: elaborada pela autora.

Nesta aula, exploramos as estruturas de dados disponíveis em Python e aprendemos a implementar listas encadeadas com essa linguagem. Além de serem fundamentais por si só, as listas encadeadas também são importantes para a implementação de outras estruturas de dados, como pilhas e filas, que serão abordadas nas aulas subsequentes.

Vamos Exercitar?

A lista encadeada oferece uma solução eficaz para o gerenciamento de prontuários médicos, pois sua estrutura permite adicionar e remover registros de forma ágil, sem a necessidade de reallocar todos os dados como em um *array*. A função **buscar_prontuario** fornece acesso rápido aos registros médicos, essencial para um atendimento eficiente.

Observe um exemplo de possível solução para o caso apresentado:

```
#resolucao situacao problema
class Prontuario:
    def __init__(self, paciente, diagnostico,
                 tratamento, proximo=None):
        self.paciente = paciente
        self.diagnostico = diagnostico
        self.tratamento = tratamento
        self.proximo = proximo

class ListaEncadeadaProntuarios:
    def __init__(self):
        self.cabeca = None

    def adicionar_prontuario(self, paciente,
                            diagnostico, tratamento):
        novo_prontuario = Prontuario(paciente,
                                      diagnostico, tratamento, self.cabeca)
        self.cabeca = novo_prontuario

    def buscar_prontuario(self, nome_paciente):
        atual = self.cabeca
        while atual:
            if atual.paciente == nome_paciente:
                return atual
            atual = atual.proximo
        return None
```

```
# Uso da lista encadeada para gerenciar prontuários
sistema_prontuarios = ListaEncadeadaProntuarios()
sistema_prontuarios.adicionar_prontuario("Alice
Santos", "Diabetes Tipo 2", "Metformina")
sistema_prontuarios.adicionar_prontuario("João
Silva", "Hipertensão", "Losartana")
# Adicionar mais prontuários conforme necessário

# Buscando um prontuário
prontuario_alice =
sistema_prontuarios.buscar_prontuario("Alice
Santos")
```

Saiba mais

Para complementar seus estudos, acesse os artigos abaixo:

1. [Introdução a listas encadeadas](#);
2. [Implementação e busca em listas encadeadas em Python](#);
3. [Inserção e deleção de itens](#).

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

BACKES, A. R. **Algoritmos e estruturas de dados em Linguagem C**. Rio de Janeiro: LTC, 2023.

CURY, T. E. *et al.* **Estrutura de dados**. Porto Alegre: SAGAH, 2018.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

MARIANO, D. C. B. **Introdução à estrutura de dados em Python**. Estrutura de Dados, 2021. Disponível em: <https://tinyurl.com/y8uhm5uz>. Acesso em: 6 dez. 2023.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

Aula 3

Pilhas e Filas

Pilhas e Filas



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá! Nesta aula, exploraremos duas estruturas de dados fundamentais: filas e pilhas. Filas operam no princípio FIFO (*First In, First Out*), essenciais para controle de processos e tarefas sequenciais. Pilhas, por outro lado, seguem o princípio LIFO (*Last In, First Out*), utilizadas em tarefas como desfazer ações e navegação de páginas. Abordaremos também como pilhas podem ser implementadas eficientemente usando listas encadeadas, permitindo operações dinâmicas e flexíveis. Entender essas estruturas é essencial para resolver problemas complexos em programação.

Vamos lá?

Ponto de Partida

Filas constituem uma estrutura amplamente observada em contextos variados, desde ambientes cotidianos, como em bancos e pontos de ônibus, até em aplicações de software. Caracterizam-se pelo princípio de que o primeiro elemento a chegar será também o primeiro a ser atendido e, consequentemente, o primeiro a deixar a fila. Em contrapartida, os elementos que chegam posteriormente ocupam as posições finais e são atendidos por último (Lambert, 2022).

No âmbito da ciência da computação, as filas desempenham um papel significativo em diversas operações, como na gestão de processos a serem executados por um sistema operacional ou na ordem de arquivos a serem impressos. A característica fundamental das filas é a política de "primeiro a entrar, primeiro a sair" (FIFO - *First In, First Out*), um conceito facilmente compreendido e aplicável a diversas estruturas de dados.

Considere, por exemplo, o uso de filas em um aplicativo de reprodução de músicas. A organização de listas de reprodução para diferentes contextos, como uma *playlist* animada para o trajeto matinal ao trabalho ou uma seleção mais serena para o jantar, ilustra o conceito FIFO. As músicas são selecionadas de um vasto catálogo e reproduzidas na ordem em que foram adicionadas, assumindo que a reprodução aleatória não está ativada. Assim, o aplicativo

ESTRUTURA DE DADOS

funciona como uma típica estrutura de dados FIFO, onde a primeira música adicionada é também a primeira a ser tocada (Szwarcfiter; Markenzon, 2020).

Outro tipo de estrutura de dados relacionada é a pilha, que difere da fila principalmente na ordem de inserção e remoção dos elementos. Enquanto nas filas o primeiro elemento adicionado é o primeiro a ser removido, nas pilhas, conhecidas pela política LIFO (*Last In, First Out*), o primeiro elemento adicionado é o último a ser removido, pois os elementos subsequentes são "empilhados" sobre ele.

Nesta aula, vamos abordar como implementar dois tipos de estruturas de dados baseadas em listas, ambas podendo ser facilmente desenvolvidas em Python. Para garantir seu entendimento, suponhamos que você é um desenvolvedor em uma empresa de seguros que recentemente expandiu suas operações devido a uma fusão com outra grande seguradora, abrangendo todos os estados do país.

O sistema que você desenvolveu inicialmente para investigar acidentes e detectar fraudes agora precisa ser adaptado para lidar com o aumento significativo no volume de dados, causado pela expansão da empresa. A infraestrutura de hardware atual não está equipada para gerenciar essa carga de trabalho ampliada, levando a falhas no sistema. Sua tarefa é criar uma solução temporária via software, implementando um sistema de processamento de demandas baseado na estrutura de uma fila usando listas encadeadas. Esse sistema deve incluir métodos para adicionar e remover itens da fila, e cada demanda será representada por um identificador numérico.

Vamos entender juntos como podemos resolver este problema?

Bons estudos!

Vamos Começar!

Introdução à estrutura de dados pilhas

Na ciência da computação, a estrutura conhecida como pilha (ou *stack* em inglês) é uma forma de armazenar dados que segue a política de "último a entrar, primeiro a sair" (LIFO - *Last In, First Out*). Essa estrutura permite que os itens sejam adicionados em sequência, mas a remoção ocorre a partir do item mais recentemente adicionado, isto é, do topo da pilha. Conforme descrito por Lambert (2022), a pilha é uma implementação de lista onde o último elemento inserido é o primeiro a ser removido. Um exemplo prático e ilustrativo dessa estrutura pode ser observado em uma pilha física de pratos ou livros, onde o item no topo é sempre o primeiro a ser removido, quando se deseja transferir itens de uma pilha para outra.

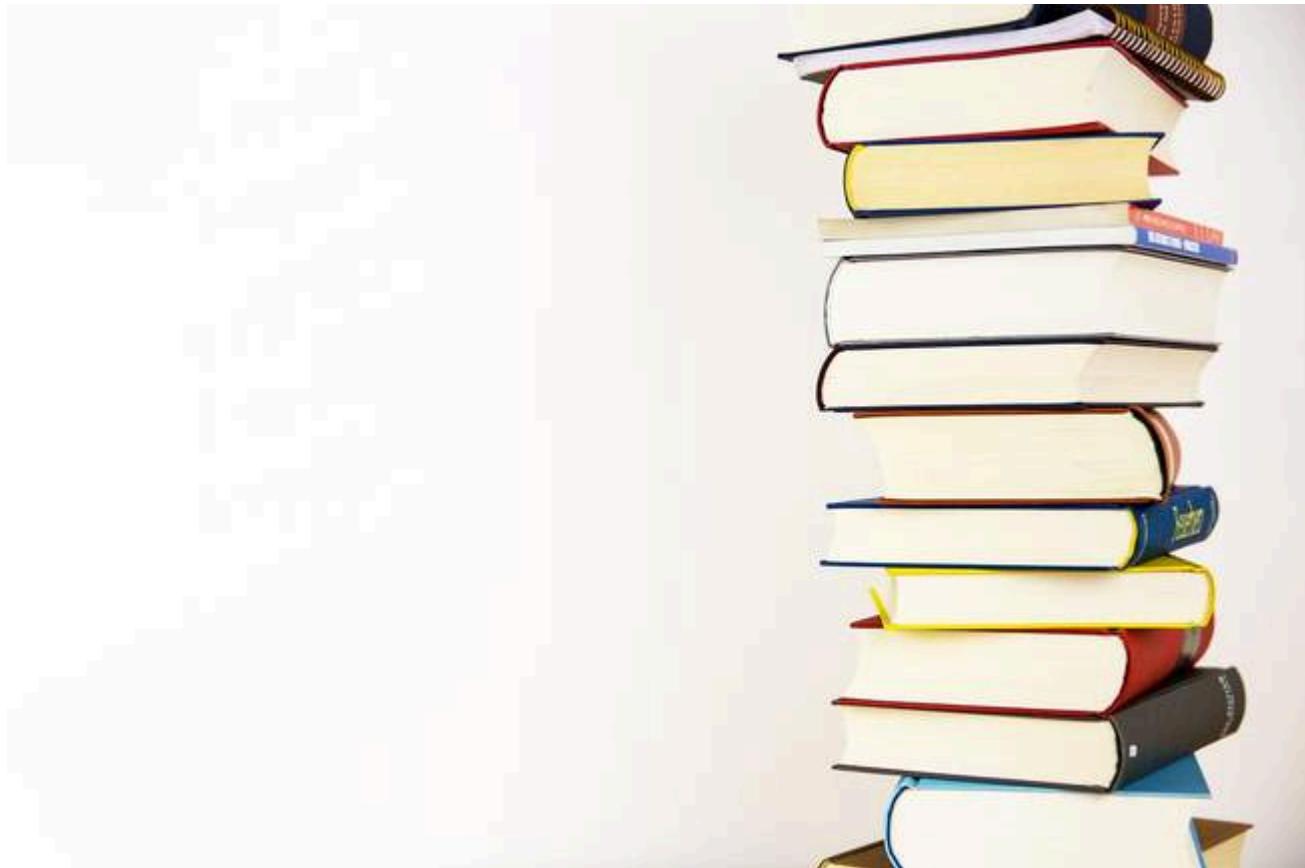


Figura 1 | Pilha de livros. Fonte: Pixabay.

Dentro do contexto acadêmico da ciência da computação, as pilhas são estruturas de dados que suportam duas operações fundamentais: o **empilhamento**, conhecido também como operação '*push*', que consiste na adição de um item à pilha, e o **desempilhamento**, referido como operação '*pop*', que envolve a remoção de um item da pilha. Essa descrição é corroborada pelos trabalhos de Lambert (2022). A natureza dessas operações implica que, ao adicionar elementos a uma pilha, estes são dispostos uns sobre os outros, seguindo a sequência de inserção.

Observe a Figura 2, nela temos quatro pilhas: cada uma delas representa um momento de uma única pilha, ou seja, cada uma delas representa um novo item sendo adicionado à pilha; um novo item é sempre inserido acima do item anterior (assim como em uma pilha de livros).



Figura 2 | Empilhamento. Fonte: elaborada pela autora.

Na operação de desempilhamento, se dá o processo inverso. A remoção dos itens é feita a partir do topo. Imagine o que aconteceria se você removesse o livro que está na base de uma pilha. Portanto, assim como usualmente removemos o livro do topo no processo de desempilhar uma pilha de livros, também o fazemos para as estruturas de dados do tipo pilha, conforme demonstrado na figura a seguir. Por esse motivo, dizemos que pilhas são estruturas do tipo LIFO.

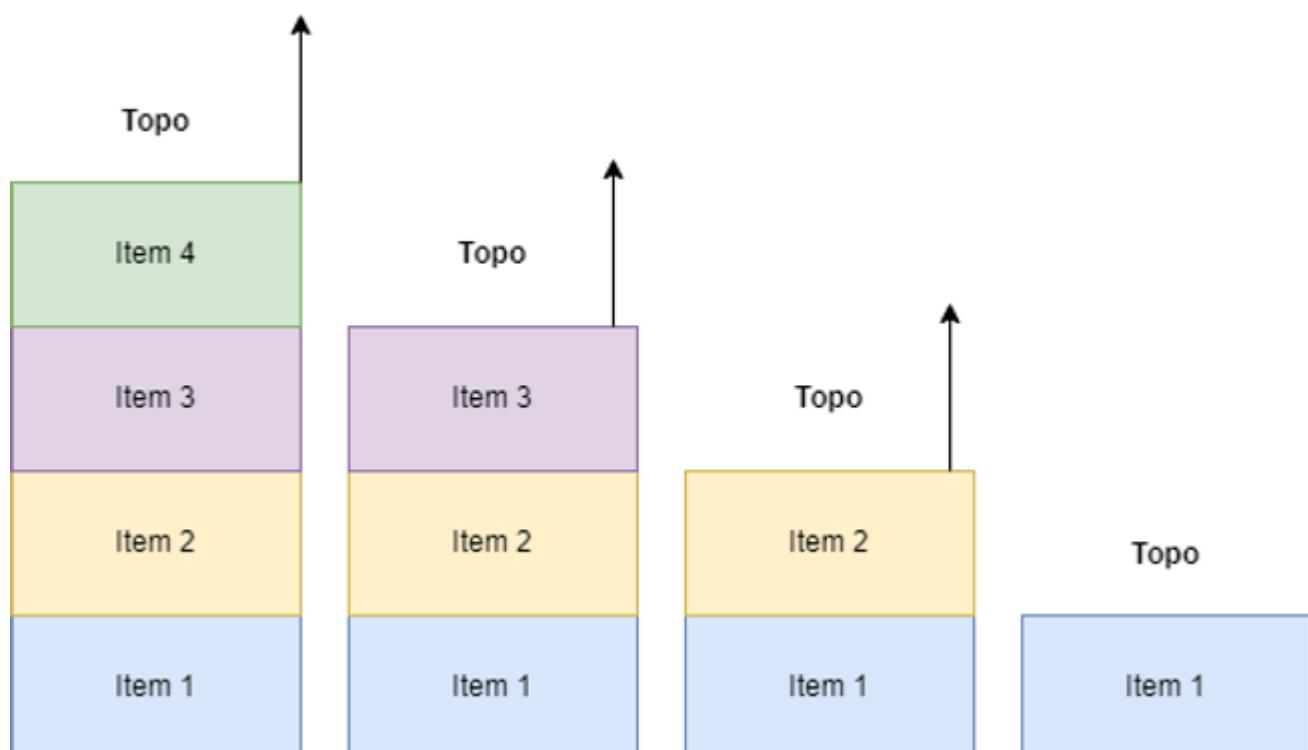


Figura 3 | Desempilhamento. Fonte: elaborada pela autora.

Operações de *pop* e *push*

Pilhas podem ser implementadas usando as estruturas nativas de listas Python (*list*). Os métodos embutidos de objetos do tipo lista (ou seja, listas simples criadas enumerando valores separados por vírgula entre colchetes) agem como se uma lista fosse uma pilha. Por exemplo, os métodos de inserção e remoção de itens em listas, os métodos *append()* e *pop()*, respectivamente (Lambert, 2022).

O método *append()* adiciona um elemento ao final de uma lista. Você pode interpretar isso como se fosse o topo de uma pilha. Se usarmos o método *pop()* nessa mesma lista, o valor removido, quando nenhum argumento é passado na chamada, será o último valor inserido. Logo, podemos concluir que o método *pop()* remove itens do topo da pilha (Backes, 2023).

Na programação Python, pilhas podem ser eficientemente implementadas utilizando as listas nativas da linguagem, conhecidas como *list*. A natureza inerente dos métodos associados aos objetos do tipo lista permite que sejam tratadas como pilhas. Especificamente, os métodos *append()* e *pop()* de listas podem ser empregados para simular as operações de empilhamento e desempilhamento, respectivamente.

O método *append()* é utilizado para adicionar um elemento ao final de uma lista, o que, em termos de uma pilha, corresponderia ao ato de colocar um item no topo. Por outro lado, o método *pop()*, quando invocado sem argumentos, remove e retorna o último elemento adicionado à lista. Essa funcionalidade é análoga à remoção de um item do topo de uma pilha. Portanto, é possível concluir que, através desses métodos, a estrutura de dados do tipo lista em Python pode ser utilizada para simular o comportamento de uma pilha (Mariano, 2021).

Observe a seguir um exemplo de uma pilha que armazena valores numéricos.

```
pilha = [4,1,5,1,6]
# operação de push com o método append()
pilha.append(10)
#print(pilha) # [4, 1, 5, 1, 6, 10]
#operação de pop com o método pop()
pilha.pop()
#print(pilha) # [4, 1, 5, 1, 6]
```

Também é viável empregar conceitos de orientação a objetos para desenvolver uma estrutura mais sofisticada. Veja o exemplo a seguir:

```
class Pilha:
    def __init__(self):
        self.items =
    []
```

```
def empilhar(self, item):
    self.items.append(item)

def desempilhar(self):
    if self.items:
        return self.items.pop()
    return None
```

Nesse caso, é necessário instanciar um objeto dessa classe para adicionar e remover itens:

```
# Exemplo de uso da pilha
my_stack = Pilha()
my_stack.empilhar(1)
my_stack.empilhar(2)
my_stack.empilhar(3)

# Removendo elementos
print("Elemento removido:", my_stack.desempilhar()) # Remove e retorna o elemento no topo
da pilha
```

Siga em Frente...

Pilhas baseadas em listas encadeadas

Em Ciência da Computação, a utilização de pilhas baseadas em listas encadeadas representa uma abordagem eficiente e sofisticada para o gerenciamento de dados seguindo o princípio *Last-In, First-Out* (LIFO). As pilhas são estruturas de dados fundamentais, amplamente empregadas em diversas aplicações, como na avaliação de expressões, navegação em históricos de navegadores e algoritmos de *backtracking* (Mariano, 2021).

Uma pilha baseada em lista encadeada difere das implementações convencionais de pilhas, que geralmente utilizam *arrays*. Enquanto as pilhas baseadas em *arrays* estão sujeitas a limitações de tamanho e requerem redimensionamento dinâmico para acomodar elementos adicionais, as pilhas baseadas em listas encadeadas oferecem flexibilidade considerável, pois sua capacidade pode expandir ou contrair dinamicamente, sem a necessidade de realocação de memória.

Em uma pilha implementada através de uma lista encadeada, cada elemento é um nó que contém não apenas o dado, mas também uma referência (ou ponteiro) para o próximo item na pilha. Essa estrutura permite que cada novo elemento seja inserido no início da lista (o topo da pilha), enquanto a remoção de elementos também ocorre a partir do início, assegurando o comportamento LIFO (Mariano, 2021).

Vejamos um exemplo de implementação de pilha utilizando listas encadeadas:

```
class Item:  
  
    def __init__(self, valor=None,  
                 anterior=None):  
        self.valor = valor  
        self.anterior = anterior  
    def __repr__(self):  
        return "%s\n%s%"  
        (self.valor, self.anterior)
```

Agora vamos implementar a classe **Pilha**:

```
class Pilha:  
  
    def __init__(self):  
        self.topo = None  
    def __repr__(self):  
        return "TOPO\n%s\nRODAPÉ" % (self.topo)
```

A classe **Pilha** será inicialmente definida com um único atributo chamado **topo**, que indica o elemento no topo da pilha. Em seguida, implementaremos um método para adicionar novos valores à pilha. Esse método, denominado **push()**, será integrado à classe Pilha:

```
def push(self, valor):  
    # Cria um novo  
    # objeto Item  
    item_novo =  
    Item(valor)  
    # o anterior passa  
    # a ser o antigo topo  
    item_novo.anterior  
    = self.topo  
    # o topo da pilha  
    # passa a ser o item  
    # novo
```

```
self.topo =  
item_novo
```

Por último iremos implementar o método que desempilha um objeto da pilha. Vamos chamar essa função de *pop()*:

```
def pop (self):  
    assert self.topo, "Erro:  
    pilha vazia."  
    # modifica o valor do  
    topo  
    self.topo =  
    self.topo.anterior
```

Esse método utiliza o comando *assert* para verificar se o topo da pilha está vazio. Se estiver, ele gerará um erro. Se a pilha tiver itens, a função altera o valor do topo, atribuindo o valor que está no atributo anterior. Vejamos agora um exemplo de utilização:

```
def main():  
    # cria um novo  
    objeto do tipo  
    Pilha  
    pilha = Pilha()  
    # Vamos inserir  
    alguns valores  
    pilha.push('a')  
    pilha.push('b')  
    pilha.push('c')  
    pilha.push('d')  
    print(pilha)
```

Topo

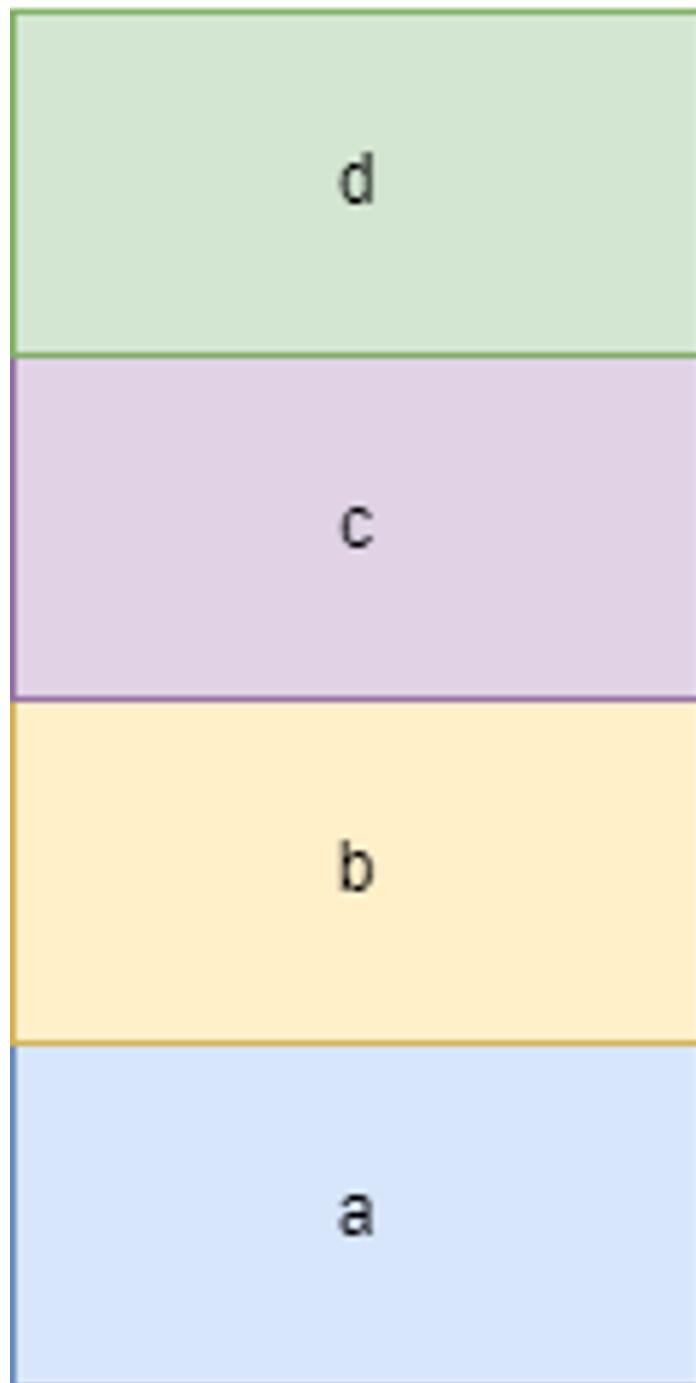


Figura 4 | Representação do comportamento do código. Fonte: elaborada pela autora.

```
# removendo os dois
últimos itens
pilha.pop() #
remove d
pilha.pop() #
remove c
print(pilha)
```

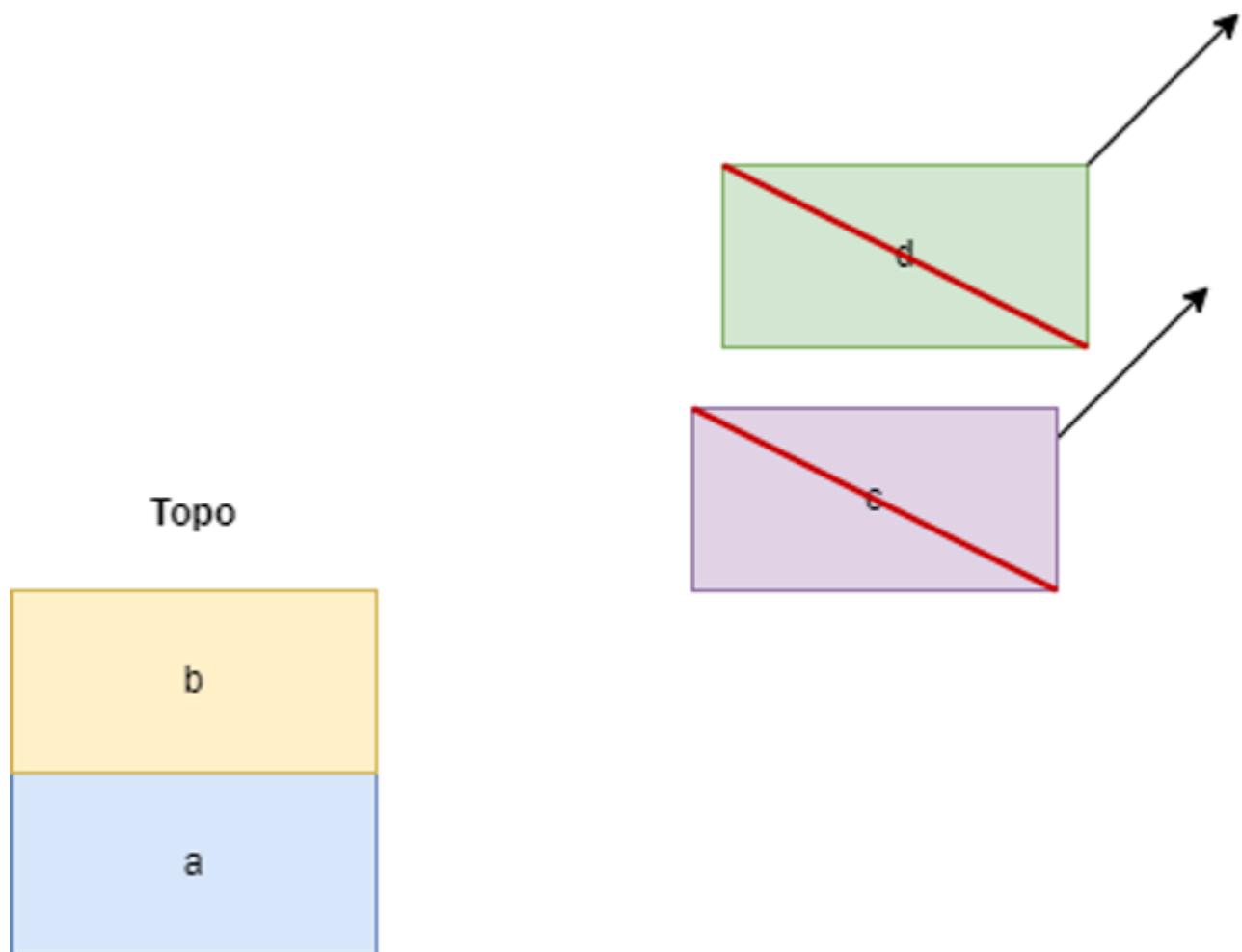


Figura 5 | Representação do comportamento POP do código. Fonte: elaborada pela autora.

Para uma compreensão acadêmica aprofundada do funcionamento deste código, referimo-nos à figura ilustrativa apresentada. Na subseção (A), observa-se que a pilha inicia-se em um estado vazio. Avançando para (B), o primeiro valor, denominado como "a", é introduzido na pilha. Nesse estágio, a variável 'pilha' sofre uma modificação única: o atributo 'topo' é atualizado para um novo objeto do tipo 'Item', com o atributo 'valor' definido como "a" e o atributo 'anterior' configurado como *None*. Prosseguindo para (C), um segundo elemento, "b", é acrescentado. Aqui, o atributo

'topo' da variável 'pilha' é atualizado para conter o objeto "b", que, por sua vez, tem seu atributo 'anterior' apontando para o objeto "a". A representação gráfica da pilha (localizada no painel superior direito) ilustra claramente que "b" é posicionado acima de "a". Por fim, em (D-E), observa-se que a inserção de novos itens segue a mesma lógica (MARIANO, 2021).

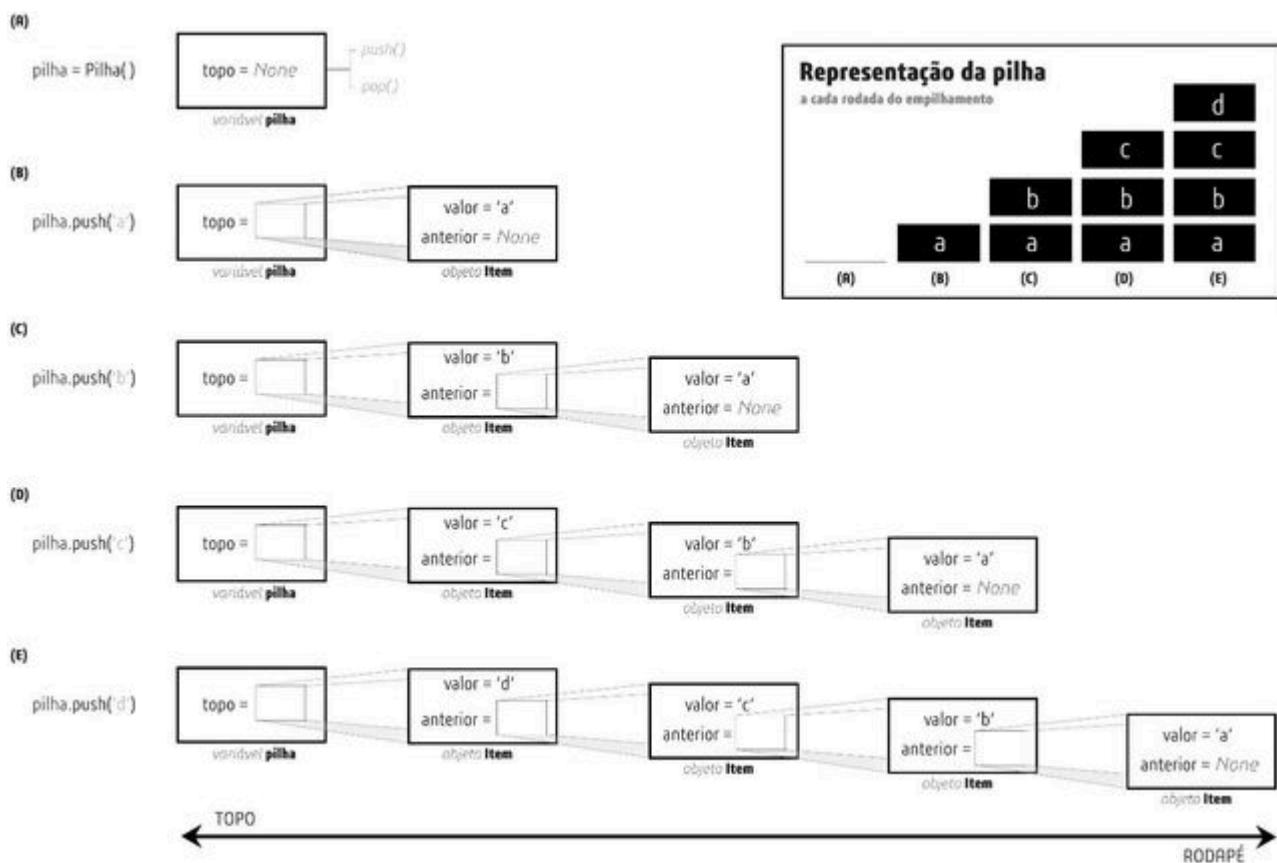


Figura 6 | Representação do funcionamento das inserções em uma pilha. Fonte: adaptada de Mariano (2021).

Após isso, a cada nova inserção, um novo objeto do tipo 'Item' será criado. Além disso, a variável 'topo' irá receber sempre o último item inserido. Note que cada item se conecta ao próximo item, conforme esperado de uma lista encadeada.

Por fim, ao executar o método `pop()`, o código simplesmente altera o valor do atributo `topo` da variável `pilha` para o conteúdo do objeto armazenado no atributo anterior do primeiro objeto do `topo`. Observe um exemplo ilustrativo do uso do método `pop()`.

A cada nova inserção na pilha, um novo objeto do tipo 'Item' é criado. Adicionalmente, a variável 'topo' é atualizada para refletir o último item inserido. É importante ressaltar que cada item na pilha está conectado ao item subsequente, alinhando-se com a característica fundamental de uma lista encadeada.

Ao utilizar o método `pop()`, o procedimento consiste simplesmente em modificar o valor do atributo 'topo' da variável 'pilha', atualizando-o para apontar para o objeto que está referenciado

no atributo 'anterior' do objeto atualmente no topo. Para uma compreensão mais clara desse processo, considere um exemplo ilustrativo que demonstre a aplicação do método `pop()`:

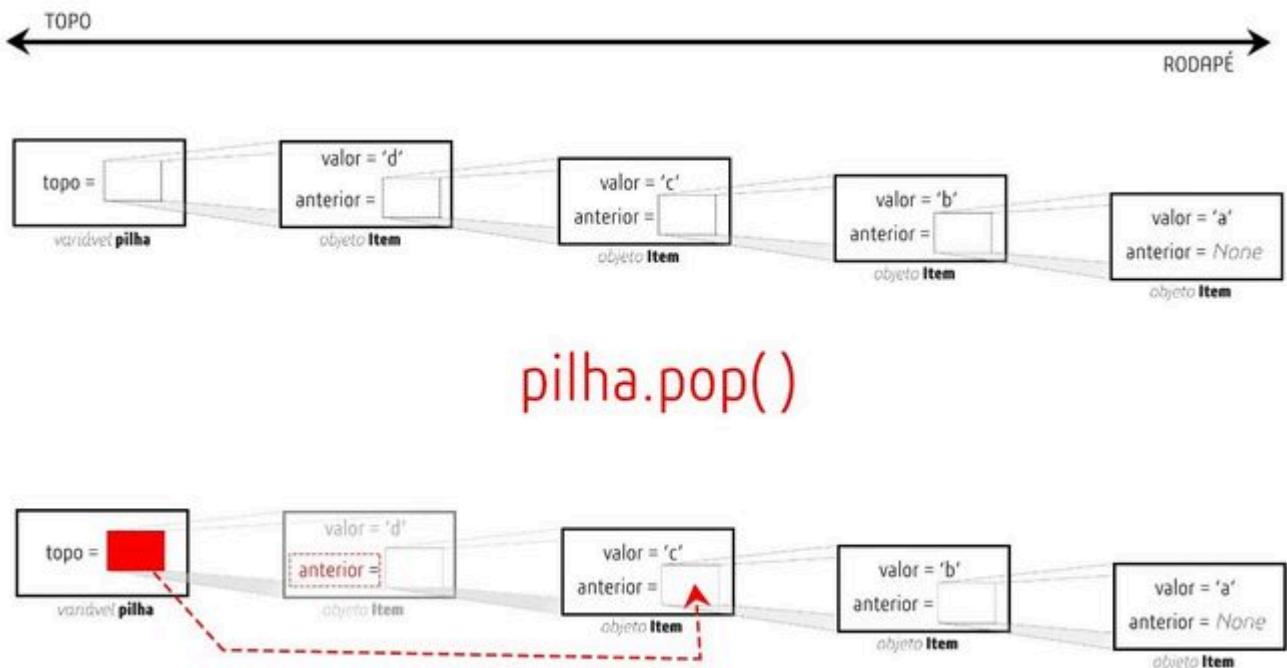


Figura 7 | Exemplo de uso do método `pop()`. Fonte: adaptada de Mariano (2021).

Introdução a filas e suas operações

Nos estudos de estruturas de dados, uma fila é frequentemente descrita como uma estrutura do tipo FIFO (*First-In, First-Out*), que se traduz como "primeiro a entrar, primeiro a sair" (Alves, 2021). Nessa estrutura, o elemento que é adicionado primeiro na fila será também o primeiro a ser removido e, consequentemente, o elemento adicionado por último será o último a sair. A figura ilustrativa a seguir exibe uma fila composta por três elementos: A, B e C. Nesse caso, o elemento A encontra-se na frente da fila, enquanto o C está na posição final. Durante uma operação de remoção, o elemento na frente da fila (A) é o primeiro a ser removido. Inversamente, quando novos elementos (como D e E) são adicionados à fila, eles são posicionados sequencialmente após o último elemento existente na fila.

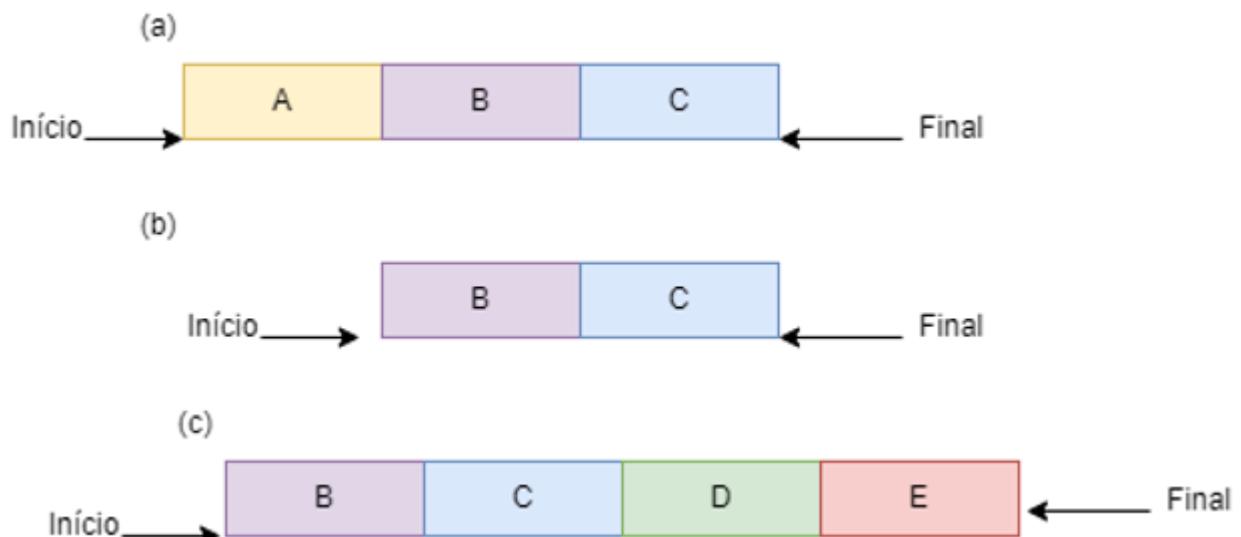


Figura 8 | Representação de uma fila. Fonte: elaborada pela autora.

É possível implementar filas em Python utilizando a biblioteca `collections.deque`. No exemplo a seguir, vamos construir uma fila contendo três itens: 1) banana, 2) chocolate e 3) morango.

```
from collections import deque
# usa deque para criar uma fila
fila = deque(["banana", "chocolate",
              "morango"])
print(fila) # deque(['banana',
              'chocolate', 'morango'])
```

Pode-se adicionar itens a essa fila usando o método `append()`:

```
# adicionando um novo elemento
fila.append("abacaxi")
print(fila) # deque(['banana',
              'chocolate', 'morango', 'abacaxi'])
```

Observe que o elemento foi inserido ao final da fila, após o item “morango”, que era o terceiro e último elemento da fila.

Pode-se remover itens usando o método `popleft()`:

```
# Remove o primeiro elemento
# adicionado à fila.
fila.popleft()
print(fila) # deque(['chocolate',
'morango', 'abacaxi'])
```

Nesse caso, o primeiro item da lista foi removido, ou seja, a *string* “banana”.

Nesta aula, foram abordados os conceitos básicos de duas estruturas de dados essenciais: filas e pilhas. Exploramos como implementar essas estruturas utilizando listas encadeadas. Com o entendimento adquirido, você está agora equipado para aplicar pilhas e filas em uma variedade de tarefas, incluindo o desenvolvimento de algoritmos e modelos para a recuperação de informações em bases de dados com diferentes modelagens.

Vamos Exercitar?

Para abordar este desafio, você pode implementar uma fila baseada em listas encadeadas em Python. Esta estrutura é ideal para gerenciar demandas em um ambiente de processamento intenso, pois permite inserções e remoções eficientes sem a necessidade de reallocar todos os elementos da fila. Aqui está um exemplo de como você poderia implementar essa fila:

```
class Node:
    def __init__(self, id):
        self.id = id
        self.next = None

class Queue:
    def __init__(self):
        self.head = None
        self.tail = None

    def enqueue(self, id):
        new_node = Node(id)
        if self.tail:
            self.tail.next =
                new_node
            self.tail = new_node
        if not self.head:
            self.head =
                new_node

    def dequeue(self):
        if not self.head:
```

```
        return None
removed_id =
self.head.id
self.head =
self.head.next
if not self.head:
    self.tail = None
return removed_id
```

```
# Exemplo de uso
process_queue = Queue()
process_queue.enqueue(1)
process_queue.enqueue(2)
process_id =
process_queue.dequeue()
```

Nesse código, a classe **Node** representa cada demanda na fila, enquanto a classe **Queue** gerencia a fila de processos. O método **enqueue** adiciona novas demandas ao fim da fila, e o método **dequeue** remove a demanda mais antiga do início da fila. Essa implementação garante que a ordem das demandas seja mantida e que a manipulação da fila seja feita de maneira eficiente, adequando-se ao aumento do volume de trabalho após a fusão da empresa.

Saiba mais

Introdução a filas e suas operações

- [Algoritmos em Python](#)

Introdução à estrutura de dados pilhas

- [Algoritmos em Python](#)

Pilhas baseadas em listas encadeadas

- [Estruturas de dados: listas, filas, pilhas, conjuntos, árvores e hash tables.](#)

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

BACKES, A. R. **Algoritmos e estruturas de dados em Linguagem C**. Rio de Janeiro: LTC, 2023.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

MARIANO, D. C. B. **Introdução à estrutura de dados em Python**. Estrutura de Dados, 2021. Disponível em: <https://tinyurl.com/y8uhm5uz>. Acesso em: 6 dez. 2023.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

Aula 4

Aplicação de Estruturas de Dados

Aplicação de Estruturas de Dados

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá! Nesta aula, enfatizaremos a importância de escolher a estrutura de dados adequada para seu sistema. Utilizar a estrutura correta é essencial para otimizar a performance e eficiência de nossos programas. Estruturas bem escolhidas melhoram o tempo de processamento e a gestão de memória, impactando diretamente a execução do software. Além disso, a gestão de dados complexos e relacionais se torna mais eficaz, permitindo a manipulação e análise de grandes volumes de dados com maior facilidade.

Vamos juntos entender quais e como aplicar essas estruturas?

Ponto de Partida

Na era digital contemporânea, a eficiência no processamento de dados é um pilar central para o sucesso de sistemas computacionais. Nesta aula, exploraremos a importância fundamental das estruturas de dados na otimização da performance e na eficiência, particularmente no que tange à gestão de dados complexos e relacionais.

ESTRUTURA DE DADOS

Começaremos abordando a importância das estruturas de dados, enfatizando como a escolha apropriada de uma estrutura pode impactar significativamente tanto o desempenho quanto a funcionalidade de um programa. Estruturas de dados não são apenas recipientes para armazenamento de dados, mas também são instrumentos essenciais para a organização lógica, acesso eficiente e manipulação de grandes conjuntos de dados.

Em seguida, focaremos na otimização de performance e eficiência, discutindo como diferentes estruturas de dados, como *arrays*, listas encadeadas, árvores, pilhas e filas são otimizadas para cenários específicos de uso. Analisaremos casos de uso reais onde a escolha correta da estrutura de dados resultou em melhorias notáveis na performance, ilustrando conceitos como complexidade de tempo e espaço.

Por fim, a aula avançará para a gestão de dados complexos e relacionais, onde as estruturas de dados desempenham um papel relevante na modelagem e na manipulação eficiente de dados em sistemas como bancos de dados relacionais. Abordaremos estruturas especializadas como árvores B+ e grafos, essenciais para representar e gerenciar relações complexas em grandes volumes de dados.

Para assimilar de forma prática o conteúdo, suponha que você está desenvolvendo um sistema para um centro de atendimento ao cliente que deve gerenciar eficientemente as solicitações de serviço. O sistema deve ser capaz de processar dois tipos de solicitações: urgente e padrão. As solicitações urgentes têm prioridade e devem ser atendidas antes das solicitações padrão. Além disso, o sistema deve garantir que as solicitações padrão sejam atendidas na ordem em que foram recebidas.

O seu desafio consiste em determinar qual estrutura de dados é a mais adequada para implementar o sistema de gerenciamento de solicitações do centro de atendimento ao cliente, justificando sua escolha com base na otimização de performance.

Vamos juntos descobrir como solucionar este problema?

Bons estudos!

Vamos Começar!

A importância da escolha de estruturas de dados

A aplicação de estruturas de dados é um aspecto fundamental em diversos âmbitos, em especial em Ciência da Computação, pois elas são a base da organização e o armazenamento eficiente de dados em programas e sistemas. A escolha apropriada de uma estrutura de dados pode influenciar significativamente o desempenho de um algoritmo ou aplicativo, afetando diretamente a eficiência, a rapidez e até a viabilidade de operações computacionais específicas (Lambert, 2022).

Por exemplo, as listas encadeadas são essenciais quando se necessita de inserções e exclusões eficientes em qualquer posição da lista, sem a necessidade de realocar a memória, como é o caso em *arrays* (Backes, 2023). As árvores, por outro lado, são extremamente úteis para operações de busca e ordenação de dados, como demonstrado nas árvores binárias de busca, que permitem inserções, exclusões e buscas em tempo proporcional à altura da árvore.

Além disso, as pilhas e filas são amplamente utilizadas para controlar o fluxo de execução em programas, com as pilhas apoiando a implementação de chamadas de função e as filas auxiliando na gestão de tarefas em sistemas operacionais e aplicativos de servidor (Backes, 2023). De forma similar, as tabelas de *hash* são empregadas para buscas rápidas e eficientes, sendo um componente crítico em muitos algoritmos de indexação e banco de dados (Lambert, 2022).

A escolha de uma estrutura de dados apropriada não é apenas uma questão de desempenho, mas também de conveniência e clareza. Estruturas de dados adequadas podem simplificar significativamente a implementação de um algoritmo e tornar o código mais comprehensível e manutenível.

Otimização de performance e eficiência

A aplicação de estruturas de dados na otimização de performance e eficiência é um tópico crítico na ciência da computação, envolvendo a seleção e utilização estratégica de estruturas adequadas para maximizar o desempenho e a eficácia das operações computacionais. A escolha correta da estrutura de dados pode resultar em melhorias significativas no tempo de execução, no consumo de memória e na complexidade geral dos algoritmos (Lambert, 2022).

Por exemplo, ao lidar com grandes volumes de dados, a utilização de árvores de busca balanceadas, como AVL ou Red-Black Trees, oferece uma vantagem significativa sobre listas não ordenadas, principalmente em operações de busca, inserção e deleção, mantendo o tempo de execução proporcional ao logaritmo do número de elementos (Szwarcfiter; Markenzon, 2020). Em contextos onde as buscas frequentes e a inserção rápida são essenciais, as tabelas de *hash* demonstram ser extremamente eficientes, fornecendo, na maioria dos casos, uma busca em tempo constante ($O(1)$) (Backes, 2023).

Outro exemplo notável é a aplicação de pilhas e filas. Pilhas são fundamentais na implementação de algoritmos de *backtracking* e na realização de chamadas recursivas, enquanto filas são essenciais em algoritmos de busca em largura e na gestão de *buffers* em sistemas operacionais (Alves, 2021). A seleção entre uma pilha ou uma fila pode impactar diretamente a maneira como os dados são acessados e processados, influenciando a eficiência do algoritmo. Filas, em particular, desempenham um papel significativo na otimização de processos em ambientes de computação em nuvem, facilitando o gerenciamento eficiente de carga de trabalho e processamento de dados.

Siga em Frente...

ESTRUTURA DE DADOS

Um exemplo bastante interessante é o uso de Filas na AWS (Amazon Web Services), especialmente implementadas através do Amazon Simple Queue Service (SQS). O SQS oferece uma solução de fila de mensagens escalável, que permite aos desenvolvedores desacoplar e escalar microsserviços, sistemas distribuídos e aplicações *serverless* (AWS Amazon, 2023).

Por exemplo, em um cenário de processamento de pedidos de *e-commerce*, uma fila SQS pode ser usada para gerenciar as solicitações de pedidos. Quando um pedido é feito, ele é enviado para a fila SQS, aguardando o processamento. Esse modelo permite que o serviço de processamento de pedidos opere independentemente do volume de pedidos, processando cada pedido sequencialmente, conforme ele chega na fila (Pereira, 2016). Isso reduz a carga no sistema de processamento, evita gargalos de desempenho e melhora a escalabilidade geral do sistema.

A otimização alcançada pelo uso de filas na AWS não se limita ao processamento de pedidos ou à arquitetura de microsserviços. Ela também se estende a cenários como a ingestão de dados, onde filas podem ser utilizadas para gerenciar o fluxo de entrada de grandes volumes de dados; e sistemas de notificação, onde filas garantem a entrega confiável e ordenada de mensagens.

A melhoria da performance de sistemas através da aplicação de estruturas de dados é uma prática essencial na Engenharia de Software e na Ciência da Computação. Estruturas de dados bem escolhidas podem significativamente melhorar a eficiência de um sistema, tornando-o mais rápido e capaz de lidar com grandes volumes de dados ou solicitações de maneira eficaz.

Para ilustrar esse conceito, considere o exemplo de um sistema de reserva de bilhetes on-line. Suponha que você esteja desenvolvendo um sistema de reserva de bilhetes para um cinema e que esse sistema precisa lidar com um grande número de consultas de clientes e processar reservas de assentos em tempo real.

Qual tipo de estrutura de dados você poderia utilizar na construção de uma possível solução?

1. **Uso de arrays ou matrizes:** para gerenciar os assentos do cinema, você pode usar uma matriz bidimensional, onde cada elemento representa um assento específico no cinema (por exemplo, assentos[fila][coluna]). Essa estrutura de dados permite acessar rapidamente o estado de qualquer assento, verificando se ele está disponível ou reservado.
2. **Filas para gerenciar reservas:** uma fila pode ser usada para gerenciar as solicitações de reserva. Como as filas seguem a política FIFO (*First-In, First-Out*), elas garantem que as reservas sejam processadas na ordem em que são recebidas, mantendo a justiça e a eficiência no processamento.
3. **Árvores para pesquisas rápidas:** caso o sistema permita aos usuários escolher assentos com base em preferências específicas (por exemplo, proximidade da tela, acesso para cadeirantes), uma árvore de busca, como uma árvore binária de busca balanceada, pode ser usada para organizar os assentos. Isso permite buscas rápidas e eficientes para encontrar assentos que atendam aos critérios dos usuários.

Impactos na performance:

ESTRUTURA DE DADOS

1. **Eficiência na consulta de assentos:** a matriz permite verificar rapidamente a disponibilidade de assentos, otimizando o tempo de resposta para os usuários.
2. **Justiça no processamento de reservas:** a fila assegura que as reservas sejam tratadas de forma justa e ordenada, evitando conflitos e atrasos.
3. **Pesquisas rápidas para preferências de assentos:** as árvores de busca permitem que os usuários encontrem rapidamente assentos que correspondam às suas necessidades, melhorando a experiência do usuário.

ESTRUTURA DE DADOS



The screenshot shows a mobile browser window for [Checkout - Ingresso.com](https://carrinho.ingresso.com). At the top, there's a header with a close button (X), a lock icon, the URL, and a bookmark icon. Below the header is a navigation bar with a back arrow, a search icon, and the text "ESCOLHA DE ASSENTOS". On the left, there's a "Legenda" (Legend) button and a "Ver números" (View numbers) button with a checkbox. The main area displays a seating chart for a movie screen. The chart consists of rows labeled from O at the top to A at the bottom. Seats are represented by blue dots. A legend on the left shows blue dots for occupied seats and grey dots for available seats. To the right of the seating chart, there's a vertical column of letters from O to A. Below the seating chart is a "TELA" (Screen) button. At the bottom, there's a summary bar with a left arrow, a price of "R\$ 0,00" in a central orange box, and a right arrow. Below this bar are three black navigation icons: a triangle, a circle, and a square.

Checkout - Ingresso.com
https://carrinho.ingresso.com

ESCOLHA DE ASSENTOS

Legenda Ver números

TEL

Use gesto pinça para dar zoom

R\$ 0,00

Reserva Cinema. Fonte: Meu Positivo.

Em conclusão, a escolha e aplicação apropriadas de estruturas de dados podem ter um impacto significativo na otimização da performance de sistemas. No exemplo do sistema de reserva de bilhetes, a combinação de matrizes, filas e árvores de busca contribui para um sistema mais eficiente, rápido e responsivo, resultando em uma melhor experiência para o usuário e maior eficiência operacional.

Gestão de dados complexos e relacionais

A gestão de dados complexos e relacionais é um aspecto importante da Ciência da Computação moderna, especialmente no campo de banco de dados e sistemas de informação. Estructuras de dados adequadas são fundamentais para armazenar, manipular e acessar grandes conjuntos de dados que são frequentemente inter-relacionados. A escolha correta da estrutura de dados pode influenciar significativamente a eficiência e a eficácia de operações de banco de dados, especialmente em ambientes onde a complexidade e o volume de dados são consideráveis (Pereira, 2016).

Por exemplo, árvores B e árvores B+ são amplamente utilizadas em sistemas de gerenciamento de banco de dados para indexação. Essas estruturas permitem inserções, buscas e exclusões eficientes, mesmo em grandes conjuntos de dados, o que é importante para o desempenho de operações de consulta e atualização em bancos de dados relacionais. As árvores B+ são particularmente eficazes para buscas intervalares e varreduras sequenciais, pois todos os valores estão nos nós folha e há um encadeamento entre eles, facilitando operações sequenciais (Pereira, 2016).

Além disso, as estruturas de grafos são amplamente utilizadas na modelagem e gestão de dados complexos e relacionais, particularmente em aplicações como redes sociais, sistemas de recomendação e análise de redes. Grafos permitem representar relações complexas entre entidades de maneira intuitiva e eficaz, e algoritmos baseados em grafos, como busca em profundidade e largura, são fundamentais para navegar e analisar essas relações (Lambert, 2022).

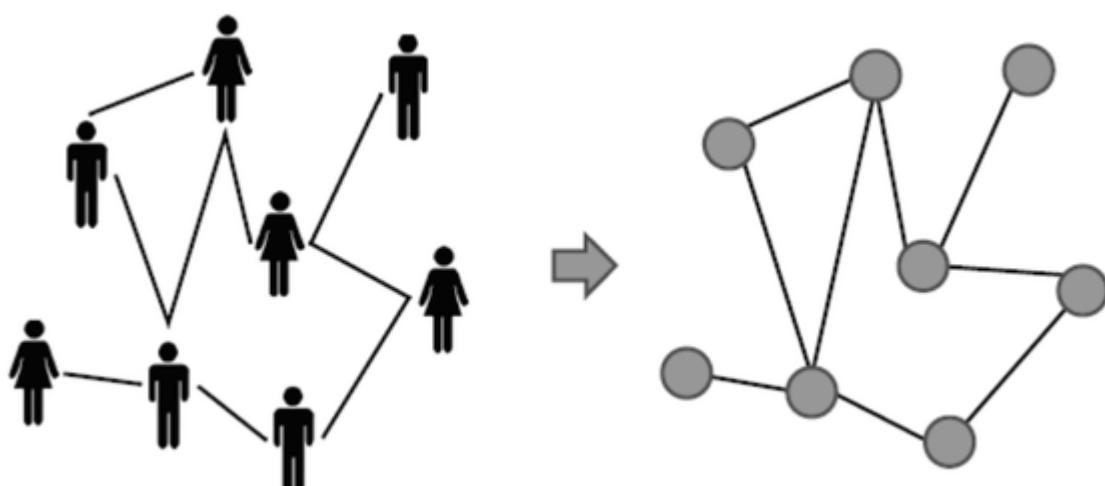


Figura 2 | Grafos. Fonte: adaptada de Backes (2023).

Tabelas *hash*, por outro lado, são usadas para garantir acessos rápidos a dados em operações de busca, sendo essenciais em cenários onde o tempo de resposta é um fator crítico. A estrutura de tabelas *hash*, com sua capacidade de fornecer acesso de tempo quase constante, é ideal para armazenamento e recuperação rápida de dados, especialmente em aplicações de alta frequência, como caches de sistemas web (Pereira, 2016).

Em resumo, a aplicação de estruturas de dados na gestão de dados complexos e relacionais é um campo extenso e importante para a eficiência dos sistemas de informação modernos. Estruturas como árvores B+, grafos e tabelas *hash* demonstram sua utilidade em ambientes que demandam armazenamento eficiente, relações complexas entre dados e recuperação rápida. A compreensão dessas estruturas e a habilidade de aplicá-las adequadamente são, portanto, competências indispensáveis para profissionais da área de Tecnologia da Informação e Ciência da Computação.

Vamos Exercitar?

No início desta aula você foi desafiado com o seguinte problema: fizemos a suposição de que você estaria desenvolvendo um sistema para um centro de atendimento ao cliente que deveria gerenciar eficientemente as solicitações de serviço. O sistema deveria ser capaz de processar dois tipos de solicitações: urgente e padrão.

Com base no que foi estudado nesta aula, a melhor escolha para a implementação desse sistema de atendimento seria uma **Fila de Prioridade (Heap)**, pois ela oferece as seguintes vantagens para o cenário em questão:

1. **Gerenciamento de prioridades:** filas de prioridade são ideais para situações onde diferentes elementos têm diferentes níveis de prioridade. Elas permitem que elementos com alta prioridade, como as solicitações urgentes, sejam atendidos antes dos elementos com prioridade mais baixa.
2. **Ordem de atendimento:** além de gerenciar prioridades, as filas de prioridade mantêm a ordem de chegada para elementos com a mesma prioridade, assegurando que todas as solicitações padrão sejam atendidas na sequência correta.
3. **Eficiência em inserção e remoção:** filas permitem inserção e remoção de elementos em tempo logarítmico ($O(\log n)$), o que é mais eficiente do que listas encadeadas para um grande volume de solicitações.
4. **Flexibilidade:** filas podem ser facilmente atualizadas para acomodar mudanças nas políticas de atendimento ou na introdução de novos níveis de prioridade.

As pilhas não seriam a escolha mais adequada, pois operam em um formato LIFO (*Last In, First Out*), o que contraria a necessidade de atender solicitações padrão na ordem de chegada. Filas padrão seriam eficazes para manter a ordem das solicitações padrão, mas não tratariam adequadamente as solicitações urgentes. Listas encadeadas oferecem flexibilidade, mas não

fornecem uma maneira eficiente de gerenciar prioridades sem varrer toda a lista. Árvores poderiam ser usadas, mas são mais complexas e não oferecem vantagens significativas sobre as filas de prioridade neste cenário.

Em resumo, a fila de prioridade é a estrutura de dados mais eficiente para esse sistema de atendimento, proporcionando uma solução otimizada tanto em termos de performance quanto de funcionalidade.

Saiba mais

A importância de se utilizar a estrutura de dados:

- [Introdução completa à estrutura de dados: conceitos e aplicações.](#)

Otimização de performance e eficiência:

- [A Importância das estruturas de dados! Juanrivillas, 2019.](#)

Gestão de dados complexos e relacionais:

- [ÁRVORE B.](#)

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

AWS AMAZON. Conceitos básicos do Amazon SQS. **AWS Amazon**, 2023. Disponível em: <https://aws.amazon.com/pt/sqs/getting-started/>. Acesso em: 6 dez. 2023.

BACKES, A. R. **Algoritmos e estruturas de dados em Linguagem C**. Rio de Janeiro: LTC, 2023.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

PEREIRA, S. do L. **Estruturas de dados em C - Uma abordagem didática**. São Paulo: Érica, 2016.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

Aula 5

FUNDAMENTOS DE ESTRUTURAS DE DADOS

Videoaula de Encerramento



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta aula iremos repassar pelos tópicos vistos nesta unidade: começamos explorando as estruturas de dados *built-in* em Python, como listas, tuplas, dicionários e conjuntos; aprofundamos em listas encadeadas, destacando sua importância para inserções e deleções eficientes; discutimos pilhas e filas, fundamentais para o controle de dados sequenciais e baseados em prioridade; finalmente, examinamos a aplicação dessas estruturas em cenários reais, ressaltando como escolhas estratégicas podem otimizar algoritmos e resolver problemas complexos.

Prepare-se para essa jornada de conhecimento! Vamos lá!

Ponto de Chegada

Olá, estudante! Para desenvolver a competência desta Unidade, que é "Conhecer os tipos de estruturas de dados e como se aplicam as estruturas do tipo *built-in* em Python", você primeiramente conheceu os conceitos fundamentais que formam a base da programação em Python. Começamos com uma introdução às estruturas de dados *built-in*, como listas, tuplas, dicionários e conjuntos, explorando como cada uma pode ser usada para armazenar e manipular dados de forma eficiente.

À medida que avançamos para as listas encadeadas, você aprendeu sobre essa estrutura de dados dinâmica, que permite inserção e remoção de elementos sem a necessidade de realocação de memória, uma habilidade importantíssima para o processamento de dados e desenvolvimento de algoritmos eficientes (Alves, 2021).

Seguimos apresentando pilhas e filas: você descobriu como essas estruturas de dados podem ser usadas para controlar o fluxo de dados em situações que requerem processamento sequencial (filas) ou baseado em prioridade (pilhas), compreendendo seus métodos de inserção e remoção de elementos (Mariano, 2023).

Finalmente, na aplicação de estruturas de dados, você aplicou seu conhecimento em cenários práticos, selecionando a estrutura de dados apropriada para otimizar operações e resolver problemas complexos, solidificando seu entendimento sobre a importância dessas estruturas no desenvolvimento de software (Lambert, 2022). Parte superior do formulário.

É Hora de Praticar!



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Na área de computação, modelar computacionalmente um problema é um passo essencial para identificar soluções viáveis. Tomemos como exemplo o desafio do caixeiro viajante, um problema clássico da computação, que busca o menor caminho que um vendedor deve percorrer entre diversas cidades, retornando ao ponto de partida. Esse conceito pode ser aplicado a sistemas modernos de navegação, como o GPS, onde um algoritmo deve analisar rotas disponíveis e determinar a melhor trajetória até o destino desejado. A estruturação dos dados, como a representação das ruas em um grafo, é essencial para a eficiência desse processo. Simulando um contexto prático, como o de um sistema de detecção de fraudes em seguros, pense em uma estrutura de dados que permita investigar relações entre indivíduos para identificar atividades suspeitas.

Por exemplo, ao verificar ligações familiares que possam indicar fraudes em seguros de veículos ou acidentes, é necessário um sistema que interaja com bancos de dados genealógicos.

Utilizando Python, uma linguagem versátil e com vastas bibliotecas, pode-se construir uma estrutura de dados que siga o princípio de pilhas, onde o último elemento inserido é o primeiro a ser removido, facilitando a busca por conexões entre indivíduos.

Para refletir sobre a importância dessas escolhas na resolução de tais problemas computacionais, considere as seguintes questões:

1. De que maneira a representação dos dados em um grafo pode afetar a busca pela melhor rota em um sistema de navegação?
2. Como a estrutura de dados escolhida pode influenciar a detecção de fraudes em um sistema de seguros?

3. Qual é o papel da linguagem de programação, especificamente Python, na implementação de estruturas de dados para tais aplicações?

Parte superior do formulário.

Agora, para refletir sobre o que você aprendeu, considere as seguintes perguntas:

1. Como a escolha de uma estrutura de dados pode influenciar a eficiência de um algoritmo?
2. Quais são os benefícios e as desvantagens de usar listas encadeadas em comparação com listas Python “padrão”?
3. Em que situações você usaria uma pilha em vez de uma fila e vice-versa?

Essas perguntas visam ajudá-lo a consolidar seu entendimento e a aplicar o conhecimento adquirido em situações do mundo real. Continue praticando e explorando essas estruturas para se tornar um programador mais competente e eficaz.

Respondendo às perguntas propostas anteriormente:

1. A representação dos dados em um grafo impacta diretamente a busca pela melhor rota em um sistema de navegação, pois os grafos são estruturas que permitem modelar relacionamentos entre pontos – neste caso, **ruas e cruzamentos**. Eles facilitam a aplicação de algoritmos de busca de caminho, como Dijkstra ou A* (A estrela), que podem calcular o caminho mais curto ou o mais rápido com base em diferentes critérios como distância, tempo, tráfego, entre outros (veremos mais adiante). Portanto, a eficácia do sistema de navegação em encontrar a rota ótima depende substancialmente de como as informações são estruturadas e interligadas dentro do grafo.
2. A estrutura de dados pilha, que opera seguindo o princípio LIFO (*Last In, First Out*), pode ser bastante útil na detecção de fraudes em um sistema de seguros de várias maneiras:
 - **Rastreamento de eventos:** ao investigar um caso de fraude, pode ser necessário rastrear eventos em uma ordem específica, como transações financeiras ou alterações de status de reivindicações de seguro. Uma pilha pode ser usada para armazenar esses eventos cronologicamente, de modo que o mais recente (último evento ocorrido) seja examinado primeiro.
 - **Análise de dados temporais:** em algumas investigações, é importante analisar a sequência de ações para detectar padrões suspeitos. Pilhas podem ajudar a armazenar e reverter sequências de ações para verificar se há alguma irregularidade ou inconsistência temporal que possa indicar fraude.
 - **Undo operations:** sistemas de detecção de fraudes podem ter funcionalidades de “desfazer” para reverter ações e analisar o estado anterior de uma reivindicação. As pilhas são ideais para esse tipo de operação, pois permitem que os investigadores voltem atrás nas suas ações de auditoria.
3. Python desempenha um papel de grande importância na implementação de estruturas de dados para tais aplicações devido à sua sintaxe clara, “tipagem” dinâmica e uma vasta coleção de bibliotecas padrão e de terceiros. Isso permite que desenvolvedores criem, manipulem e analisem estruturas de dados complexas de forma mais intuitiva e eficiente. Além disso, Python oferece suporte a paradigmas de programação orientada a objetos, o

ESTRUTURA DE DADOS

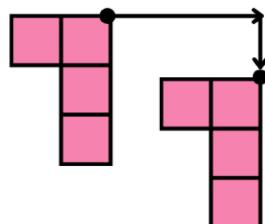
que facilita a criação de classes e objetos para representar e gerenciar dados. Isso torna Python uma escolha popular para o desenvolvimento de algoritmos de processamento de dados e sistemas de análise, como os utilizados em sistemas de navegação e detecção de fraudes.

Este infográfico é uma ferramenta didática que pode ajudar a visualizar e compreender melhor os conceitos abordados na unidade sobre estruturas de dados em Python:

FUNDAMENTOS DE ESTRUTURAS DE DADOS

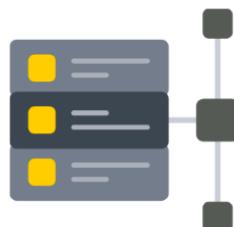
INTRODUÇÃO A ESTRUTURA DE DADOS EM PYTHON

Visão geral das estruturas de dados fundamentais disponíveis em Python



LISTAS ENCADEADAS

Uma forma de estrutura de dados onde cada elemento aponta para o próximo, facilitando a inserção e remoção de elementos sem reorganização da estrutura de dados inteira.



PILHAS E FILAS

Pilhas (LIFO - Last In, First Out) e **Filas** (FIFO - First In, First Out), estruturas de dados especializadas que gerenciam elementos em ordem específica, adequadas para certos algoritmos e aplicações.



LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

MARIANO, D. C. B. **Introdução à estrutura de dados em Python**. Estrutura de Dados, 2021. Disponível em: <https://tinyurl.com/y8uhm5uz>. Acesso em: 6 dez. 2023.

Unidade 2

ESTRUTURA DE DADOS ÁRVORES

Aula 1

Fundamentos de Árvores e Algoritmos

Fundamentos de Árvores e Algoritmos



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Bem-vindo a esta videoaula introdutória sobre árvores em estruturas de dados utilizando Python. Exploraremos o conceito de vértices e arestas, fundamentais para entender como as árvores organizam dados de forma hierárquica e eficiente. Abordaremos desde a definição básica, passando pela importância de árvores na resolução de problemas computacionais até a implementação prática em Python. Prepare-se para desbloquear uma ferramenta poderosa para seu arsenal de programação, ideal para quem busca aprofundar conhecimentos em estruturas de dados!

Ponto de Partida

Olá, estudante!

ESTRUTURA DE DADOS

Diariamente, interagimos com diversas aplicações que utilizam estruturas de dados baseadas em árvores, um conceito fundamental na computação. Por exemplo, ao navegar em uma página da Web e inspecionar seu código-fonte, você se depara com uma estrutura de árvore que organiza os elementos HTML. Essa estrutura hierárquica é visível nas etiquetas HTML, onde a etiqueta principal '`html`' contém '`head`' e '`body`', e '`body`', por sua vez, contém outras etiquetas como '`div`' e '`p`'.

Árvores de decisão são outro exemplo prático, utilizadas em sistemas como *chatbots*. Elas guiam o processo de resposta a partir de uma série de perguntas, levando a diferentes caminhos e conclusões com base nas respostas do usuário (Alves, 2021).

Existem diversos tipos de estruturas de árvores na ciência da computação, cada uma projetada para atender necessidades específicas de determinados problemas. Essa diversidade sublinha a importância de compreender os princípios fundamentais das árvores, o que pode inspirar o desenvolvimento de novas estruturas para resolver desafios inéditos ou aperfeiçoar soluções existentes.

Nesta aula, nosso foco será no estudo das árvores como estruturas de dados hierárquicas, explorando suas aplicações, operações e como representar dados categorizáveis. Aprenderemos a implementar árvores em Python, desenvolvendo habilidades essenciais para solucionar problemas reais no contexto profissional. Ao final, você terá a capacidade de identificar dados que podem ser representados por árvores e realizar implementações eficazes utilizando Python.

Para consolidar nosso estudo, suponhamos que você está desenvolvendo uma aplicação de recomendação de filmes, que leva em consideração múltiplos critérios, como palavras-chave, preferências dos usuários, gêneros e classificação etária. O projeto está sendo guiado pela abordagem do Produto Mínimo Viável (MVP), que visa criar uma versão inicial simples, porém funcional, do produto. Essa abordagem permite um lançamento rápido, minimizando esforços e recursos, facilitando a obtenção de feedback do mercado para melhorias futuras.

Para esta versão inicial, a aplicação deve ser capaz de recomendar ao menos três filmes para cada faixa etária: livre, a partir de 10 anos, a partir de 12 anos, a partir de 14 anos, a partir de 16 anos e a partir de 18 anos. Por exemplo, um usuário de 20 anos terá acesso a recomendações de todos os filmes, enquanto um usuário de 13 anos receberá recomendações de filmes adequados para menores de 14 anos.

Vamos juntos descobrir como resolver esse problema?

Bons estudos!

Vamos Começar!

Introdução a árvores

Na ciência da computação, uma árvore é uma estrutura de dados hierárquica e não linear, caracterizada por sua forma ramificada, análoga à de uma árvore biológica, da qual deriva sua nomenclatura. Essa estrutura é composta por entidades denominadas vértices ou nós, que representam os elementos de dados, e por conexões conhecidas como arestas, que estabelecem as relações entre os vértices. Um exemplo dessa estrutura pode ser observado na figura a seguir, onde se ilustra uma árvore que contém oito vértices interconectados por sete arestas (Szwarcfiter; Markenzon, 2021).

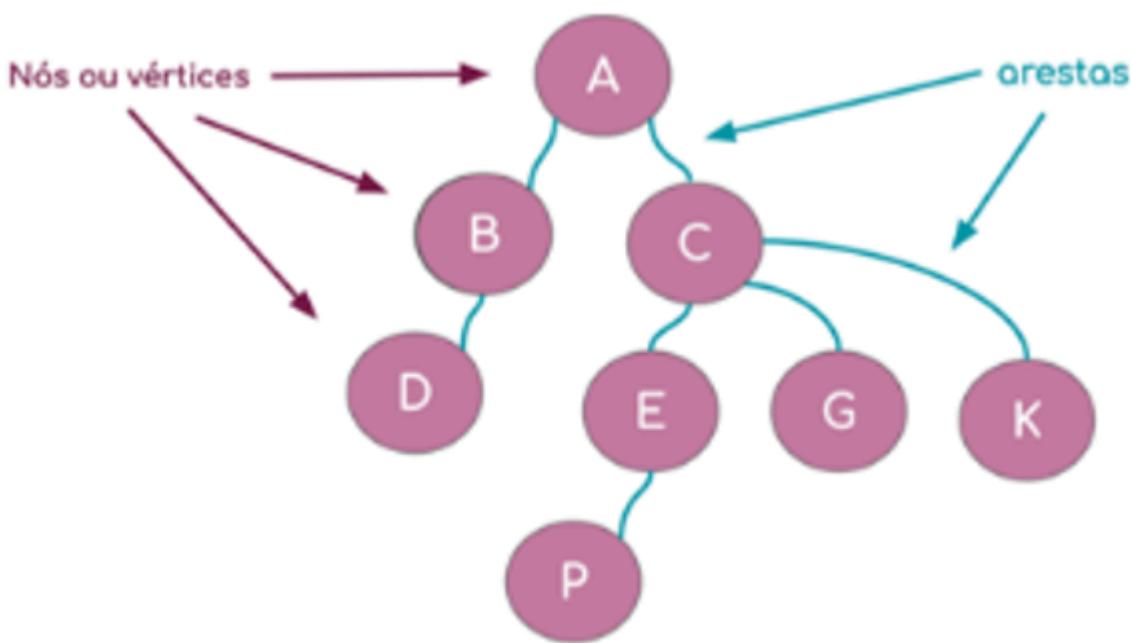


Figura 1 | Árvore com oito vértices e sete arestas. Fonte: adaptada de Takenaka (2021).

Vértices e arestas em estrutura de dados

Em uma estrutura de dados em árvore, as conexões entre os vértices, conhecidas como arestas, seguem regras específicas para garantir a formação adequada da estrutura. Primeiramente, cada vértice na árvore pode ter um máximo de uma aresta que chega até ele, garantindo que não existam múltiplas entradas para o mesmo ponto. Em segundo lugar, o número de arestas que partem de um vértice específico pode variar entre zero e um valor máximo **N**, onde **N** representa o número máximo de arestas saindo desse vértice. Essas regras asseguram que, ao percorrer os vértices da árvore, não se formará um circuito fechado, mantendo a estrutura acíclica e hierárquica característica das árvores (Alves, 2021).

Em uma estrutura de dados em forma de árvore, cada elemento é chamado de vértice ou nó. O nó do qual emergem outras conexões é referido como "pai", enquanto os nós que recebem essas conexões são denominados "filhos". Nós que compartilham o mesmo pai são conhecidos como "irmãos". Por exemplo, se o nó **A** é o pai dos nós **B** e **C**, **B** é pai de **D**, e **C** é pai de **E**, **G** e **K**, esses relacionamentos estabelecem uma hierarquia clara dentro da estrutura da árvore (Takenaka, 2021).

A posição de um vértice na árvore determina sua nomenclatura específica. A "raiz" é o vértice inicial da árvore e é único, pois não possui um nó pai. Já os "nós folha" ou "terminais" são aqueles que não possuem filhos. Entre a raiz e os nós folha estão os nós "internos" ou "não terminais", que desempenham o papel de conectar diferentes partes da árvore.

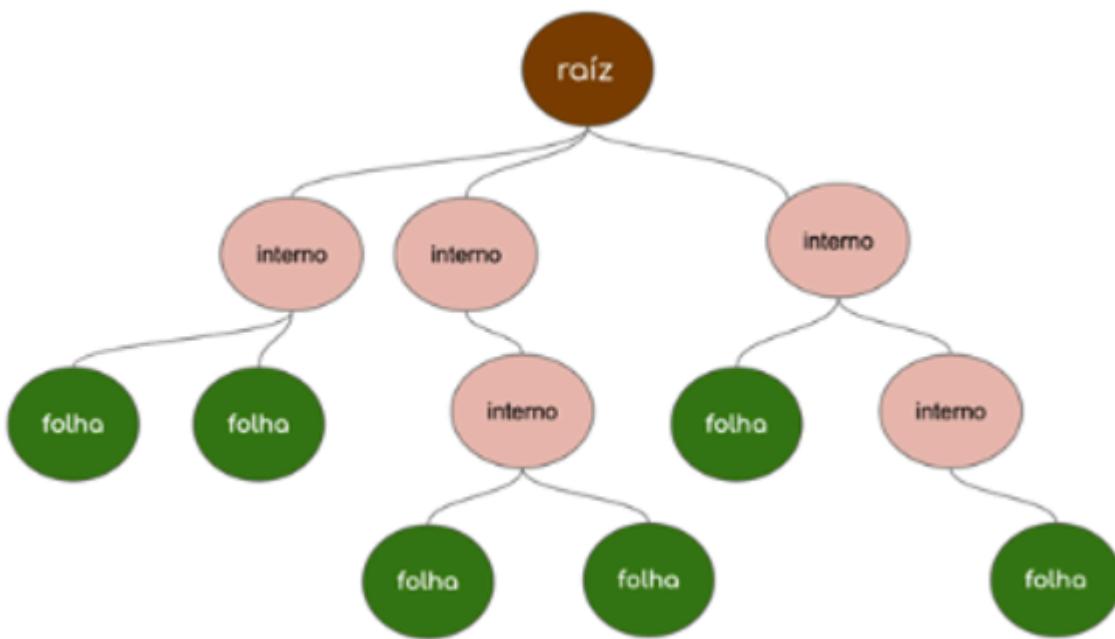


Figura 2 | Raiz e nós. Fonte: adaptada de Takenaka (2021).

A altura de um vértice em uma árvore é determinada pela contagem do número de arestas desde esse vértice até o mais distante em sua subárvore. Ou seja, a altura é medida de cima para baixo. Já a altura total da árvore é calculada a partir da raiz até o nó mais distante. Por outro lado, a profundidade de um vértice é calculada pelo número de arestas que se percorre desde esse vértice até a raiz, portanto, de baixo para cima. Importante notar que, embora os nós folha tenham a mesma altura, eles podem ter profundidades diferentes dependendo de sua localização na árvore (Takenaka, 2021).

ESTRUTURA DE DADOS

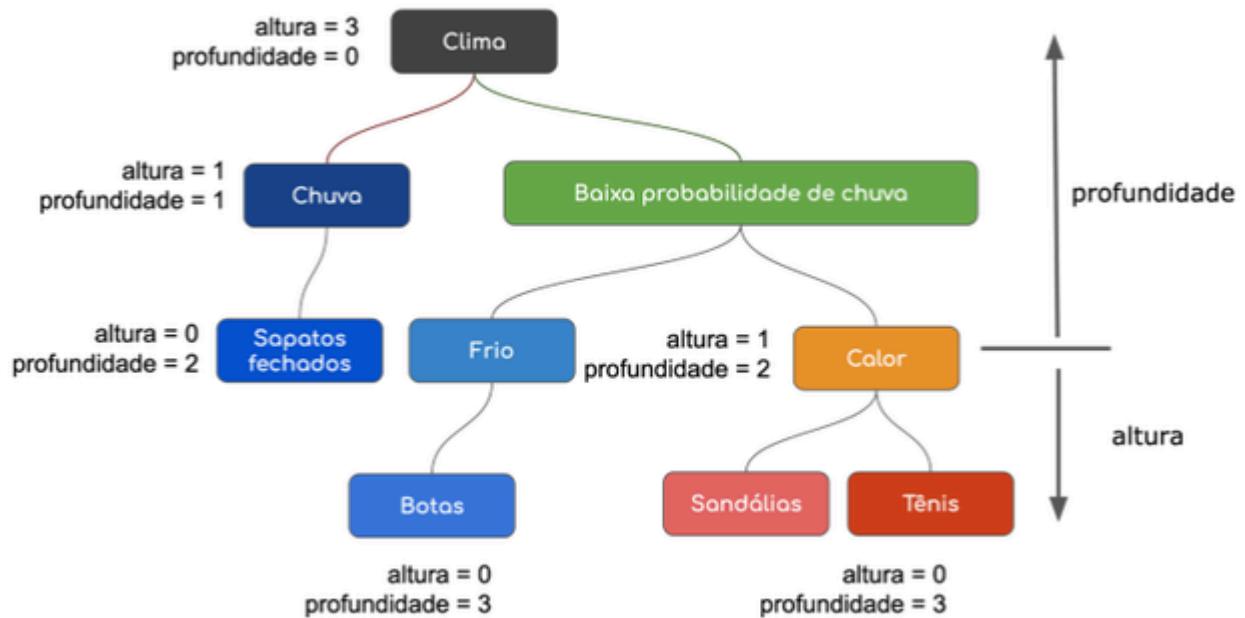


Figura 3 | Alturas e profundidades de vértices em uma árvore . Fonte: adaptada de Takenaka (2021).

Em árvores binárias, cada nó pode ter no máximo dois filhos, que são referenciados como filho esquerdo e filho direito.

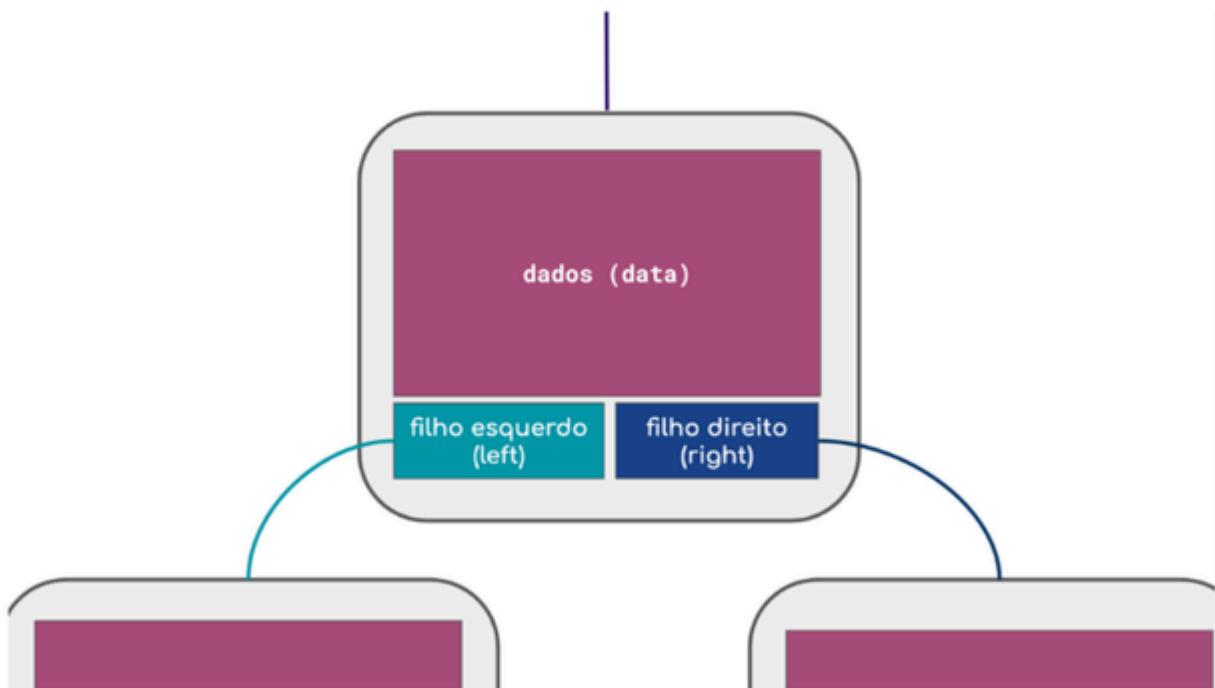


Figura 4 | Filhos de cada nó em árvores binárias. Fonte: adaptada de Takenaka (2021).

Siga em Frente...

Árvores em Python

Na linguagem de programação Python, pode-se representar uma árvore binária criando a classe **Vértice**, que inclui atributos para armazenar o valor do nó (dado) e referências para seus filhos esquerdo e direito.

```
class Vertice:  
  
    Vertice de Arvore Binária  
    """  
  
    def __init__(self, dado):  
        # dado propriamente dito, conteúdo do vértice  
        self.dado = dado  
        # filho da esquerda e da direita  
        self.esquerda = None  
        self.direita = None  
  
    def __str__(self):  
        return str(self.dado)  
  
    def representacao_com_parentheses(self):  
  
        Retorna a representação da árvore com aninhamento por parênteses  
        :return: (str)  
        """  
  
        if self.esquerda:  
            # recursividade  
            esq = self.esquerda.representacao_com_parentheses()  
        else:  
            esq = ""  
        if self.direita:  
            # recursividade  
            dir = self.direita.representacao_com_parentheses()  
        else:  
            dir = ""  
        return .format(str(self), esq, dir)  
  
    def representacao_com_reculo(self, numero_de_espacos=0):  
  
        Retorna a representação da árvore com recuo  
        :return: (str)  
        """
```

ESTRUTURA DE DADOS

```
if self.esquerda:  
    esq = self.esquerda.representacao_com_recuo(numero_de_espacos + 4)  
else:  
    esq =  
if self.direita:  
    dir = self.direita.representacao_com_recuo(numero_de_espacos + 4)  
else:  
    dir =  
return .format(  
    espacos=' '*numero_de_espacos,  
    self=str(self),  
    esq=esq,  
    dir=dir,  
)  
  
def imprimir_percурсo_em_ordem(self):  
  
    Percorre a árvore em ordem simétrica (esquerda, vértice, direita)  
    e imprime o dado do vértice  
    :return: None  
    """  
    if self.esquerda:  
        # recursividade: executa o mesmo atributo para seu filho esquerdo  
        self.esquerda.imprimir_percурсо_em_ordem()  
    # imprime o dado do vértice  
    print(self)  
    if self.direita:  
        # recursividade: executa o mesmo atributo para seu filho direito  
        self.direita.imprimir_percурсо_em_ordem()  
  
def imprimir_percурсо_pre_ordem(self):  
  
    Percorre a árvore em pré ordem (vértice, esquerda, direita)  
    e imprime o dado do vértice  
    :return: None  
    """  
    # imprime o dado do vértice  
    print(self)  
    if self.esquerda:  
        # recursividade: executa o mesmo atributo para seu filho esquerdo  
        self.esquerda.imprimir_percурсо_pre_ordem()  
    if self.direita:  
        # recursividade: executa o mesmo atributo para seu filho direito  
        self.direita.imprimir_percурсо_pre_ordem()  
  
def imprimir_percурсо_pos_ordem(self):
```

ESTRUTURA DE DADOS

Percorre a árvore em pré ordem (esquerda, direita, vértice)

e imprime o dado do vértice

:return: None

"""

```
if self.esquerda:  
    # recursividade: executa o mesmo atributo para seu filho esquerdo  
    self.esquerda.imprimir_percурсo_pos_ordem()  
if self.direita:  
    # recursividade: executa o mesmo atributo para seu filho direito  
    self.direita.imprimir_percурсо_pos_ordem()  
# imprime o dado do vértice  
print(self)
```

```
print(  
  
        Passeio  
        /   \  
    Diurno     Noturno  
    /   \   /   \  
  Frio    Calor Restaurante Cinema  
  / \   / \  
Planetario Museu Parque Praia
```

```
)  
  
# Criar os vértices  
passeio = Vertice()  
  
diurno = Vertice()  
frio = Vertice()  
planetario = Vertice()  
museu = Vertice()  
calor = Vertice()  
parque = Vertice()  
praia = Vertice()  
  
noturno = Vertice()  
restaurante = Vertice()  
ciname_noturno = Vertice()  
  
# Vincula os filhos de passeio  
passeio.esquerda = diurno  
passeio.direita = noturno
```

```
# Vincula os filhos de diurno
```

```
diurno.esquerda = frio
```

```
diurno.direita = calor
```

```
# Vincula os filhos de frio
```

```
frio.esquerda = planetario
```

```
frio.direita = museu
```

```
# Vincula os filhos de calor
```

```
calor.esquerda = parque
```

```
calor.direita = praia
```

```
# Vincula os filhos de noturno
```

```
noturno.esquerda = restaurante
```

```
noturno.direita = ciname_noturno
```

```
# Imprime representacao com parenteses
```

```
print()
```

```
print()
```

```
print(passeio.representacao_com_parentese  
s())
```

```
# Imprime representacao com recuo
```

```
print()
```

```
print()
```

```
print(passeio.representacao_com_recuo())
```

```
# Imprime os dados dos vértices
```

```
print()
```

```
print()
```

```
passeio.imprimir_percурсo_em_ordem()
```

```
print()
```

```
print()
```

```
passeio.imprimir_percурсo_pre_ordem()
```

```
print()
```

```
print()
```

```
passeio.imprimir_percурсo_pos_ordem()
```

Quando se criam objetos do tipo **Vértice**, inicialmente, eles existem como nós individuais, formando uma coleção de árvores únicas, também conhecida como floresta. Cada um desses objetos representa um nó da árvore com seu respectivo valor.

Para estruturar uma árvore binária, é necessário conectar esses nós de acordo com as regras de árvores binárias, estabelecendo relações entre nós pai e filhos. Por exemplo, em um contexto hipotético, pode-se ter nós com valores como "Passeio", "Diurno", "Frio", "Planetário", "Museu",

"Calor", "Parque", "Praia", "Noturno", "Restaurante" e "Cinema", cada um representando um vértice na árvore (Takenaka, 2021).

Ao conectar esses vértices, forma-se uma estrutura de árvore binária única, com um nó raiz e outros nós organizados hierarquicamente como filhos à esquerda ou à direita, conforme as regras específicas de árvores binárias. A configuração final da árvore reflete a maneira como os nós são interconectados. No código apresentado, temos 11 vértices soltos ou uma floresta com 11 árvores, com um vértice cada árvore. O programa criaria uma árvore com a configuração semelhante à apresentada na figura a seguir.

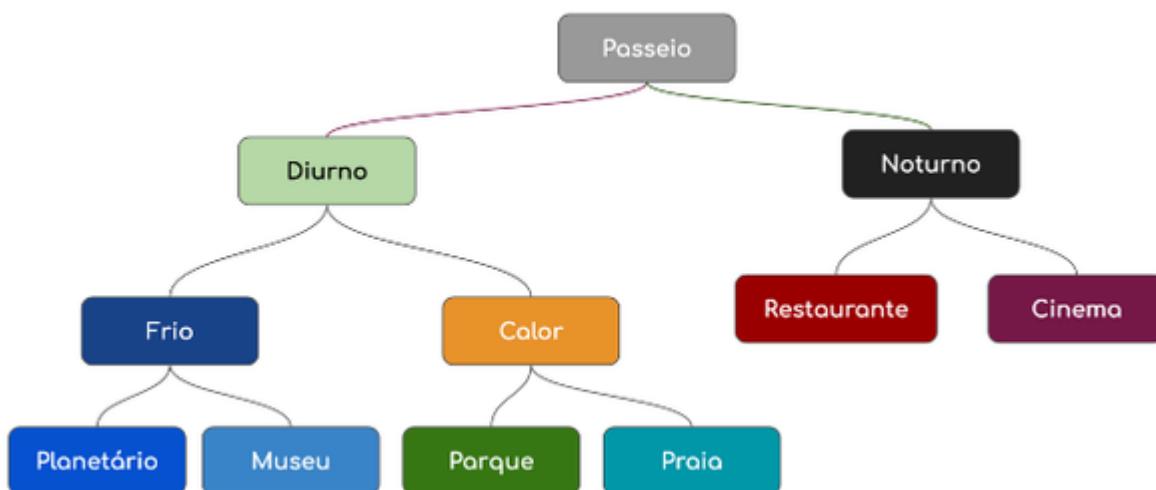


Figura 5 | Representação da árvore do programa. Fonte: adaptada de Takenaka (2021).

Para construir uma árvore que inclua todos os vértices, é necessário interligá-los estabelecendo conexões entre os filhos esquerdo e direito. Essa parte do código é responsável por estruturar a árvore ao conectar os nós de acordo com a relação pai-filho.

```
# Criar os vértices
passeio = Vertice()
```

```
diurno = Vertice()
frio = Vertice()
planetario = Vertice()
museu = Vertice()
calor = Vertice()
parque = Vertice()
praia = Vertice()
```

```
noturno = Vertice()
restaurante = Vertice()
```

```
ciname_noturno = Vertice()
```

A árvore resultante pode ser visualizada de duas maneiras: através de uma representação aninhada com parênteses ou por meio de uma representação por recuo. Ambas as representações podem ser geradas e exibidas utilizando funções definidas na classe da árvore.

No pedaço de código a seguir, a representação aninhada com parênteses é executada a partir do nó raiz, que neste caso é o vértice "passeio". Essa funcionalidade permite visualizar a estrutura da árvore ou de qualquer subárvore a partir de um vértice específico, proporcionando uma compreensão clara da hierarquia e das relações entre os nós da árvore (Lambert, 2022).

```
print()  
print(passeio.representacao_com_parentese  
s())
```

A função "*representacao_com_parenteses()*" é implementada como uma função recursiva, que é utilizada para processar tanto o filho esquerdo quanto o filho direito de um nó. Durante a execução dessa função, ela inicia com a abertura de um parêntese, seguido pelo dado contido no vértice atual. Em seguida, a função é chamada recursivamente para o filho esquerdo do vértice, e depois para o filho direito. Após as chamadas recursivas, a função finaliza com o fechamento do parêntese. Esse processo é ilustrado na figura associada, demonstrando como a estrutura da árvore é representada visualmente através de parênteses aninhados (Takenaka, 2021).

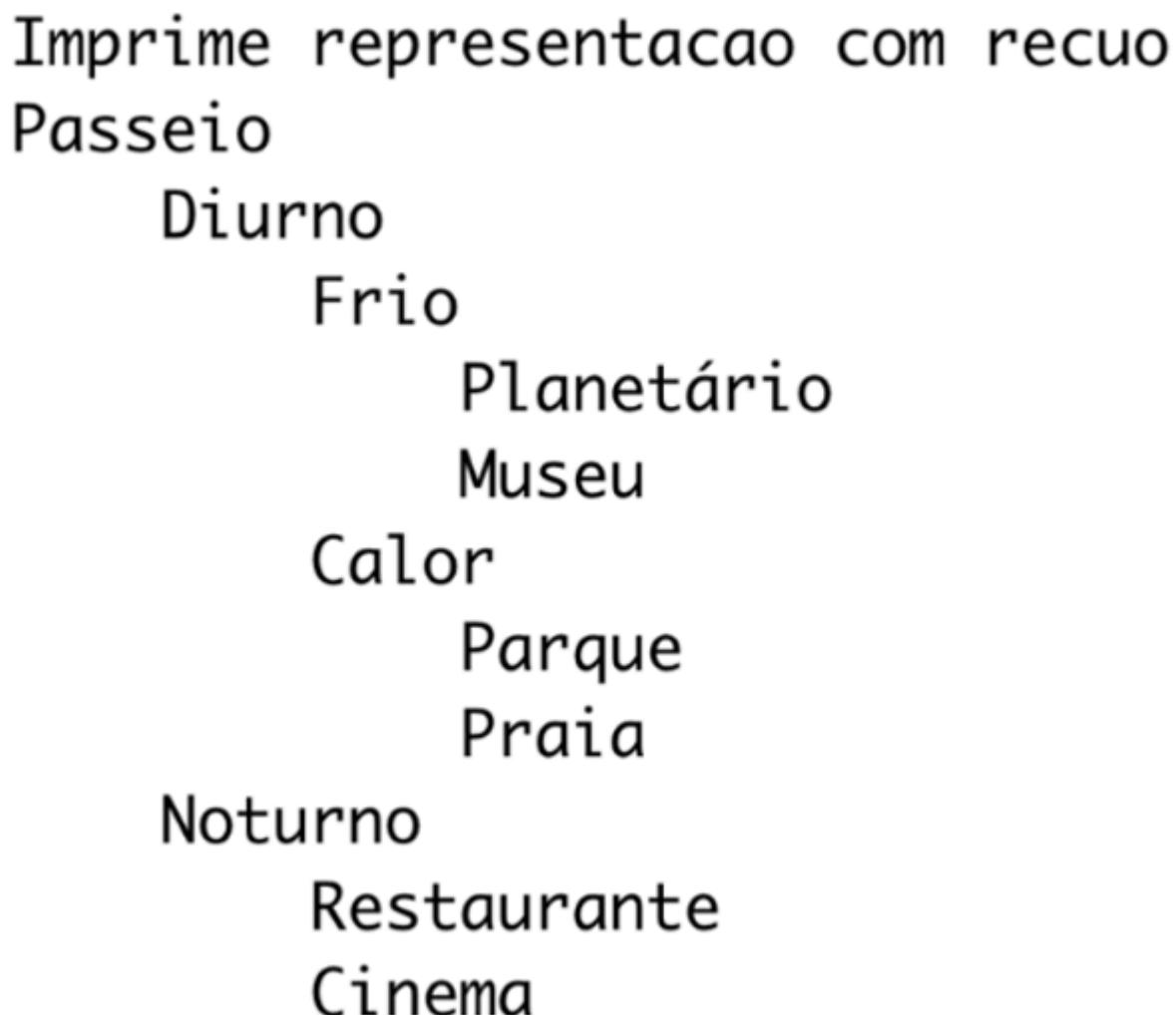


Figura 6 | Representação da árvore com recuo. Fonte: adaptada de Takenaka (2021).

Contudo, como podemos abordar a representação de estruturas mais complexas? Qual seria a forma de representar uma árvore que possui mais do que dois filhos em sua estrutura? Observe o código a seguir:

```
class Filhos:  
    def __init__(self):  
        # cria uma lista Python vazia  
        self._vertices = list()  
  
    def representacao_com_parenteses(self):  
        representacoes = []  
        for vertice in self._vertices:
```

ESTRUTURA DE DADOS

```
representacoes.append(  
  
vertice.representacao_com_parentheses()  
)  
return .join(representacoes)  
  
def representacao_com_recuo(self,  
numero_de_espacos=0):  
    representacoes = []  
    for vertice in self._vertices:  
        representacoes.append(  
  
vertice.representacao_com_recuo(numero_d  
e_espacos + 2)  
)  
    return .join(representacoes)  
  
def inserir(self, dado):  
    vertice_novo = Vertice(dado)  
    self._vertices.append(vertice_novo)  
    return vertice_novo  
  
def imprimir_percurso_pre_ordem(self):  
  
    Percorre a árvore em pré ordem (vértice,  
esquerda, direita)  
    e imprime o dado do vértice  
:return: None  
    ....  
  
    # imprime o dado do vértice  
    for vertice in self._vertices:  
        vertice.imprimir_percurso_pre_ordem()  
  
def imprimir_percurso_pos_ordem(self):  
  
    Percorre a árvore em pré ordem  
(esquerda, direita, vértice)  
    e imprime o dado do vértice  
:return: None  
    ....  
  
    for vertice in self._vertices:  
        vertice.imprimir_percurso_pos_ordem()
```

```
class Vertice:
```

ESTRUTURA DE DADOS

Vertice de Arvore N-aria

```
def __init__(self, dado):
    # dado propriamente dito, conteúdo do
    # vértice
    self.dado = dado
    # classe para instanciar filhos do vértice
    self.filhos = None

def __str__(self):
    return str(self.dado)

def representacao_com_parentheses(self):
    filhos =
    if self.filhos:
        filhos =
    self.filhos.representacao_com_parentheses()
    return .format(str(self), filhos)

def representacao_com_recuo(self,
                            numero_de_espacos=0):
    filhos =
    if self.filhos:
        filhos =
    self.filhos.representacao_com_recuo(numero
                                         _de_espacos + 2)
    return (

).format(
    espacos=' ' * numero_de_espacos,
    self=str(self),
    filhos=filhos,
)

def inserir_filho(self, dado):
    Inserir um filho no vértice atual

    if self.filhos is None:
        self.filhos = Filhos()
    return self.filhos.inserir(dado)

def imprimir_percurso_pre_ordem(self):
```

ESTRUTURA DE DADOS

Percorre a árvore em pré ordem (vértice, filhos)

e imprime o dado do vértice

:return: None

.....

imprime o dado do vértice

print(self)

if self.filhos:

recursividade: executa o mesmo atributo para seus filhos

```
self.filhos.imprimir_percorso_pre_ordem()
```

```
def imprimir_percorso_pos_ordem(self):
```

Percorre a árvore em pré ordem (filhos, vértice)

e imprime o dado do vértice

:return: None

.....

if self.filhos:

recursividade: executa o mesmo atributo para seus filhos

```
self.filhos.imprimir_percorso_pos_ordem()
```

imprime o dado do vértice

print(self)

```
print(
```

Pacotes Turisticos

/ | \

Tranquilidade Aventura Luxo

/ | \

Rafting Escalada Tirolesa

.....

```
)
```

```
# criacao de objetos da classe Vertice
```

```
raiz = Vertice()
```

```
# criação de filhos de raiz
```

```
tranquilidade = raiz.inserir_filho()
```

```
aventura = raiz.inserir_filho()
```

```
luxo = raiz.inserir_filho()
```

```
# criação de filhos de aventura
rafting = aventura.inserir_filho()
escalada = aventura.inserir_filho()
tirolesa = aventura.inserir_filho()

print()
print()
print(raiz.representacao_com_recuo())
print(raiz.representacao_com_parenteses())

print()
print()
raiz.imprimir_percuso_pre_ordem()
print()
print()
raiz.imprimir_percuso_pos_ordem()

print()
print()
aventura.imprimir_percuso_pre_ordem()
print()
print()
aventura.imprimir_percuso_pos_ordem()
```



Figura 7 | Árvore nária de classificação de pacotes turístico. Fonte: adaptada de Takenaka (2021).

No código, implementamos uma árvore n-ária, onde cada vértice pode ter um número variável de filhos. Inicialmente, criamos a raiz da árvore, que é um objeto da classe Vértice. Diferentemente de uma árvore binária, que possui filhos esquerdo e direito, na árvore n-ária, os vértices possuem uma lista de filhos, representada pela estrutura de dados “*list*” do Python.

Para gerenciar essa lista de filhos, utilizamos a classe Filhos, que contém uma lista vazia. Essa classe é responsável por inserir novos vértices na árvore. O método Filhos.inserir() cria um novo objeto do tipo Vértice e o adiciona à lista usando list.append(). Os filhos são adicionados aos vértices por meio do método Vertice.inserir_filho(), que cria e retorna o vértice filho recém-criado (Takenaka, 2021).

A classe “Vertice”, por sua vez, é adaptada para acomodar a estrutura da árvore n-ária, substituindo os atributos esquerda e direita pelo atributo filhos. O método Vertice.inserir_filho() verifica se self.filhos está instanciado e, se não estiver, instancia um objeto da classe “Filhos”. Em seguida, delega a criação e inserção do novo vértice a self.filhos.inserir(). O método retorna o vértice filho criado, mantendo o processo de inserção transparente ao usuário da classe “Vertice”.

Essa abordagem permite a criação de uma árvore n-ária flexível e eficiente, onde cada vértice pode ter um número indefinido de filhos, facilitando a representação de estruturas de dados hierárquicas complexas.

Vamos Exercitar?

O exercício proposto no ponto de partida lhe pedia para desenvolver uma aplicação de recomendação de filmes baseada em classificação etária. Com os assuntos abordados nesta aula, poderíamos propor uma solução que utiliza a estrutura de dados de árvore em Python:

- Utilizaremos uma árvore para representar as faixas etárias e os filmes recomendados para cada faixa. Cada nó da árvore representará uma faixa etária e seus filhos representarão os filmes recomendados.
- Criaremos uma classe **Node** para representar cada nó da árvore. Cada **Node** terá um atributo para armazenar a faixa etária ou o nome do filme e uma lista para armazenar os nós filhos.

```
class Node:  
    def __init__(self, value):  
        self.value = value  
        self.children = []  
  
    def add_child(self, child_node):  
        self.children.append(child_node)  
  
    def get_children(self):
```

```
return self.children
```

- Construiremos a árvore adicionando nós para cada faixa etária e, em seguida, adicionaremos nós filhos com os nomes dos filmes recomendados.

```
root = Node()  
  
livre = Node()  
dez_anos = Node()  
doze_anos = Node()  
# Adicionar outros nós para as faixas  
de 14, 16 e 18 anos  
  
root.add_child(livre)  
root.add_child(dez_anos)  
root.add_child(doze_anos)  
# Adicionar os demais nós de faixa  
etária à raiz  
  
# Adicionar filmes a cada nó de faixa  
etária  
livre.add_child(Node())  
livre.add_child(Node())  
livre.add_child(Node())  
  
dez_anos.add_child(Node())  
dez_anos.add_child(Node())  
dez_anos.add_child(Node())  
  
doze_anos.add_child(Node())  
doze_anos.add_child(Node())  
doze_anos.add_child(Node())  
# Continuar adicionando filmes para  
as demais faixas etárias
```

- Para recomendar filmes, a aplicação percorrerá a árvore e coletará os nomes dos filmes das faixas etárias que são adequadas para a idade do usuário.

```
def recomendar_filmes(raiz,  
idade_usuario):  
    recomendacoes = []  
    for faixa in raiz.get_children():
```

```
if idade_usuario >= obter_idade_minima(faixa.value):
    for filme in faixa.get_children():

        recomendacoes.append(filme.value)
    return recomendacoes

def obter_idade_minima(faixa_etaria):
    # Implementar a lógica para extrair a
    # idade mínima da string da faixa etária
    return idade_minima
```

- Testamos a aplicação chamando a função **recomendar_filmes** com diferentes idades de usuários para verificar se as recomendações estão corretas.

```
print(recomendar_filmes(root, 20)) # Deverá
# retornar filmes de todas as faixas
print(recomendar_filmes(root, 13)) # Deverá
# retornar filmes até a faixa de 12+
```

Saiba mais

Introdução a árvores:

- [Árvores: estrutura de dados](#). **AlgolDev**.

Vértices e arestas em estrutura de dados:

- [ÁRVORES. Algoritmos em Python](#).

Árvores em Python:

- DOWNEY, A. B. [Pense em Python](#).

Referências

ALVES, W. W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

LAMBERT, K. K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

SZWARCFITER, J. J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

TAKENAKA, R. R. M. **Fundamentos de árvores e algoritmos**. Estrutura de Dados, 2021. Disponível em: <http://tinyurl.com/bdfmd7h3>. Acesso em: 28 jan. 2024.

Aula 2

Árvores de Busca Binária

Árvores de Busca Binária

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Bem-vindo a esta videoaula sobre "Árvores de Busca"!

Nesta aula vamos explorar o fascinante mundo das árvores de busca binária e como elas são usadas para ordenar dados de forma eficiente. Você aprenderá a estrutura e o funcionamento dessas árvores e como implementá-las em Python. Descobriremos também suas diversas aplicações práticas, desde organização de informações até algoritmos de busca e remoção. Prepare-se para aprimorar suas habilidades de programação e compreender melhor essas poderosas ferramentas da Ciência da Computação!

Ponto de Partida

Olá, estudante!

Imagine que você está em uma biblioteca procurando um livro específico. Se a biblioteca estiver bem organizada, com todos os livros categorizados e ordenados, você saberá exatamente em qual seção e prateleira procurar. Por outro lado, se os livros estiverem desordenados, você terá que procurar em cada prateleira, o que pode ser muito demorado. Essa comparação é

ESTRUTURA DE DADOS

semelhante ao funcionamento das Árvores Binárias de Busca (ABBs). Elas organizam os dados de maneira eficiente, permitindo que você siga um caminho específico na busca.

Se a informação não estiver no caminho escolhido, você saberá que ela não existe na árvore. Assim como uma biblioteca bem organizada, as ABBs exigem que as informações sejam inseridas e removidas seguindo regras específicas para manter sua estrutura eficiente. ABBs são úteis em diversas aplicações computacionais, como codificação, bancos de dados, armazenamento de rotas e otimização de desempenho em buscas e ordenações. Elas armazenam dados (chaves) de forma que os vértices à esquerda tenham valores menores e os à direita, maiores, facilitando a busca por uma chave específica (Lambert, 2022).

Suponha que você foi encarregado de desenvolver um aplicativo de lista de tarefas para estudantes, onde é possível adicionar, remover e consultar atividades. Cada atividade vem com uma prioridade numérica, com os itens de maior importância tendo valores menores. As tarefas não podem ter a mesma prioridade, pois o usuário não pode realizar duas atividades simultaneamente.

Para essa aplicação, uma ABB seria ideal. Você poderia desenvolver o app usando uma ABB para organizar as tarefas por prioridade, garantindo eficiência na adição, remoção e busca de atividades. A seguir, você encontrará trechos de código que ilustram como implementar essa estrutura em Python para gerenciar as tarefas.

Boa sorte nos seus estudos!

Vamos Começar!

Árvores de busca

Ao analisar a estrutura e o funcionamento das árvores binárias de busca, pode-se estabelecer uma analogia com a experiência de visitar um museu com uma disposição particular de cômodos e portas. Nesse museu imaginário, cada sala possui uma porta de entrada e duas portas de saída, conduzindo a outros cômodos configurados de forma idêntica. Cada porta é acompanhada por uma placa indicativa, fornecendo informações sobre os objetos contidos no respectivo cômodo, o que auxilia o visitante na escolha do caminho a seguir (Takenaka, 2021).

Esse cenário é semelhante à estrutura de uma árvore binária de busca, onde os cômodos representam os vértices (nós) da árvore e as placas são as chaves associadas a cada vértice. Tal estrutura de dados é projetada para otimizar a recuperação de informações. Nas árvores binárias, os dados são organizados com o objetivo de classificação, seja de maneira ordenada (como em listas alfabéticas ou numéricas) ou não ordenada (como em mapas mentais), mas com ênfase na eficiência da recuperação.

De forma mais técnica, uma árvore binária de busca (ABB), também conhecida como BST (*Binary Search Tree*) em inglês, é um subtipo de árvore binária. Nessa estrutura, todas as chaves

(conteúdos dos nós) presentes na subárvore esquerda de um nó são menores do que as chaves do nó raiz. Por sua vez, na subárvore direita, todas as chaves são maiores (Alves, 2021). Esse padrão de organização é estabelecido durante o processo de inserção de dados na árvore, garantindo que a busca e recuperação de informações sejam realizadas de maneira eficiente.

Uma árvore de busca binária é uma estrutura de dados baseada em nós, onde cada nó possui no máximo dois filhos. Os elementos são organizados de forma que, para cada nó, todos os elementos na subárvore à esquerda são menores que o nó e todos os elementos na subárvore à direita são maiores. Vamos exemplificar como criar e inserir elementos nessa estrutura em Python, usando a seguinte sequência: 14, 4, 18, 0, 21, 17, 1, 8 e 13 (Takenaka, 2021).

Aplicações de árvores de busca binária em Python

Definindo a classe nó

Primeiramente, definimos uma classe para representar cada nó da árvore. Cada nó terá um valor (*val*) e referências para seus filhos da esquerda (*left*) e da direita (*right*).

```
class No:  
    def __init__(self, key):  
        self.left = None  
        self.right = None  
        self.val = key
```

Inserindo elementos na árvore

Para inserir um novo elemento, comparamos o valor do nó com o valor a ser inserido. Se for menor, movemos para a subárvore esquerda; se for maior, para a direita. Se o local onde o nó deve ser inserido estiver vazio, criamos um novo nó com o valor.

```
def inserir(root, key):  
    if root is None:  
        return No(key)  
    else:  
        if root.val < key:  
            root.right = inserir(root.right, key)  
        else:  
            root.left = inserir(root.left, key)  
    return root
```

Criando a árvore e inserindo valores

Começamos com uma árvore vazia e inserimos os elementos na ordem dada.

```
raiz = No(14) # Raiz da árvore com o primeiro valor  
sequence = [4, 18, 0, 21, 17, 1, 8, 13]  
  
for key in sequence:  
    inserir(raiz, key)
```

Para inicializar, criamos a raiz da árvore com o primeiro valor: 14.



Figura 1 | Ilustração da raiz de uma árvore. Fonte: adaptada de Takenaka (2021).

A cada inserção, a função “*inserir*” posiciona o elemento no local apropriado na árvore. Por exemplo, ao inserir 4, a função verifica que $4 < 14$ e o coloca à esquerda do nó raiz. Quando inserimos 18, ele é colocado à direita do nó raiz, pois $18 > 14$, e assim por diante.

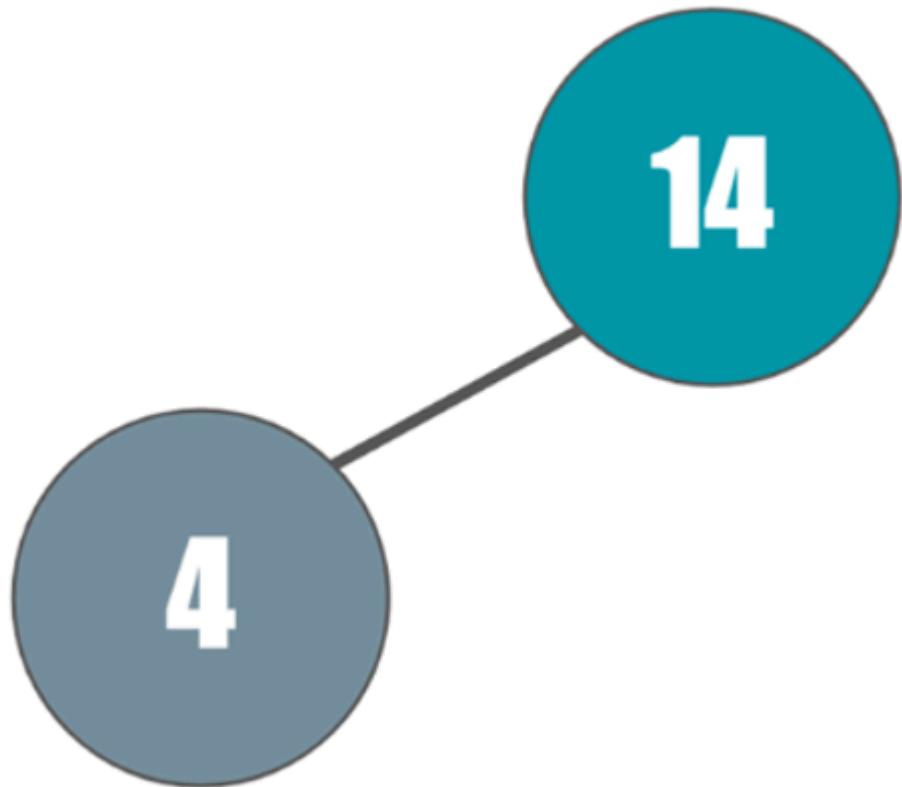


Figura 2 | Ilustração do número 4 inserido como filho de 14. Fonte: adaptada de Takenaka (2021).

Depois de inserir 14 e 4, vamos inserir o número 18, que é comparado com 14. Como 18 é maior que 14, então a execução segue, verificando se 14 tem filho maior.

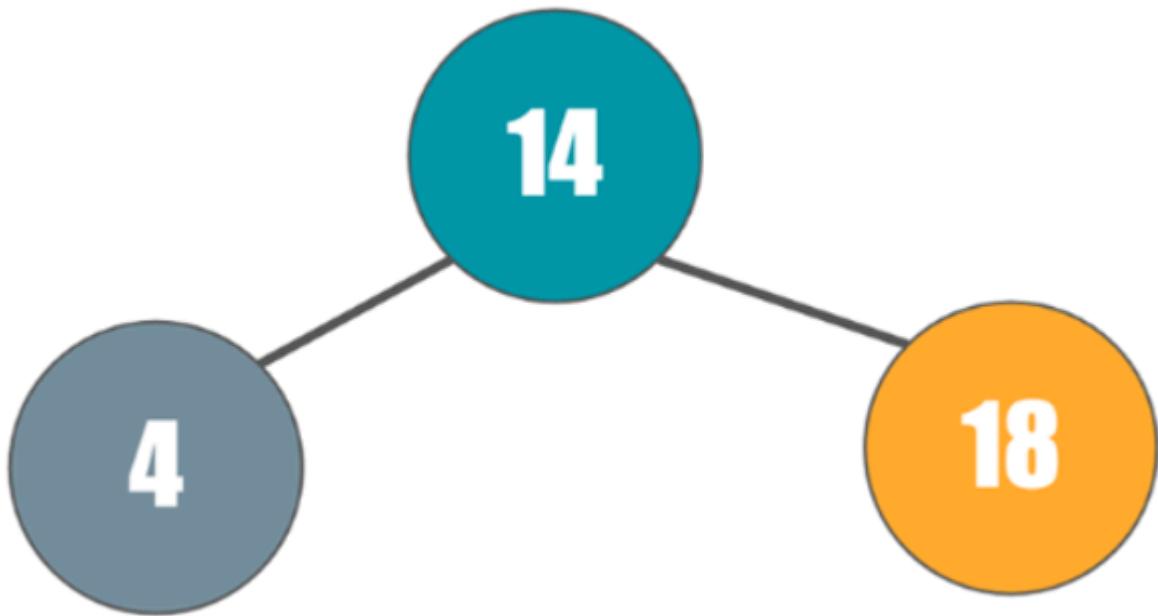


Figura 3 | Árvore binária de busca com vértices inseridos nesta ordem 14, 4 e 18. Fonte: adaptada de Takenaka (2021).

E assim ocorre sucessivamente até que a árvore tenha a configuração mostrada na figura a seguir:

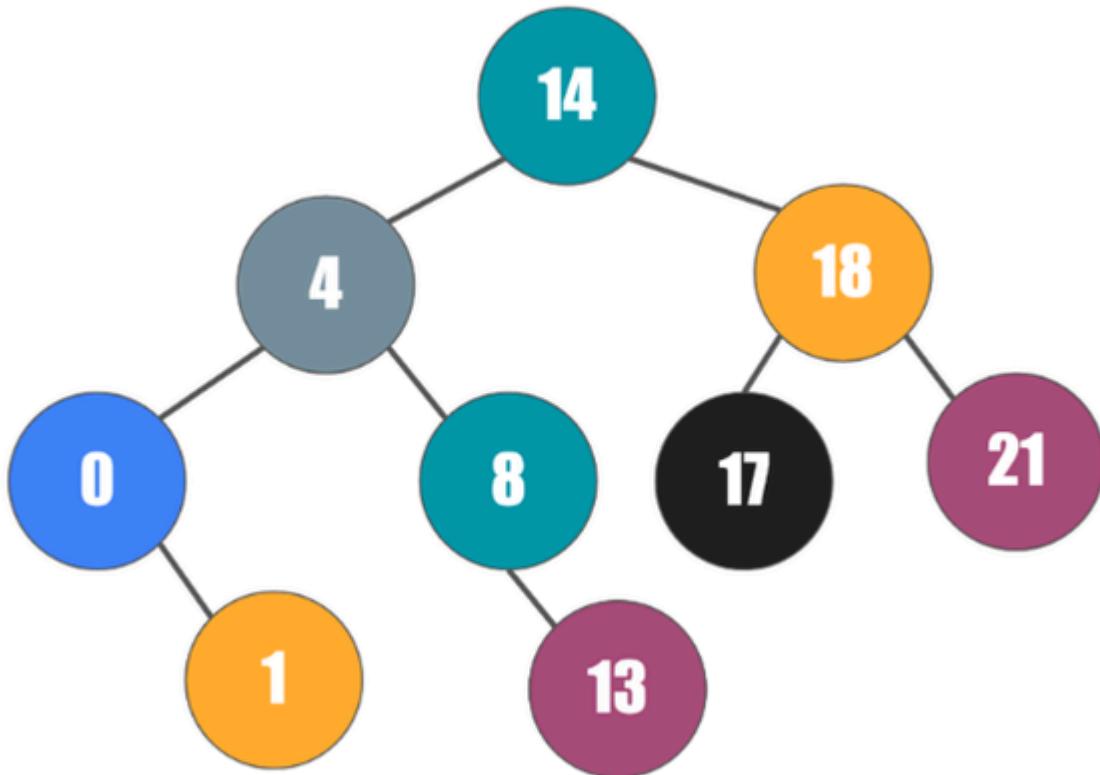


Figura 4 | Árvore binária de busca com vértices inseridos na sequência 14, 4, 18, 0, 21, 17, 1, 8 e 13. Fonte: adaptada de Takenaka (2021).

Siga em Frente...

Ordenação em árvores

E como funciona a remoção de vértices?

A remoção de vértices em uma árvore binária de busca envolve três cenários distintos que requerem abordagens específicas:

Remoção de vértice-folha

Este é o caso mais simples. Um vértice-folha é um nó que não possui filhos. Para removê-lo, basta desconectar esse nó da árvore, eliminando a referência a ele no nó pai.

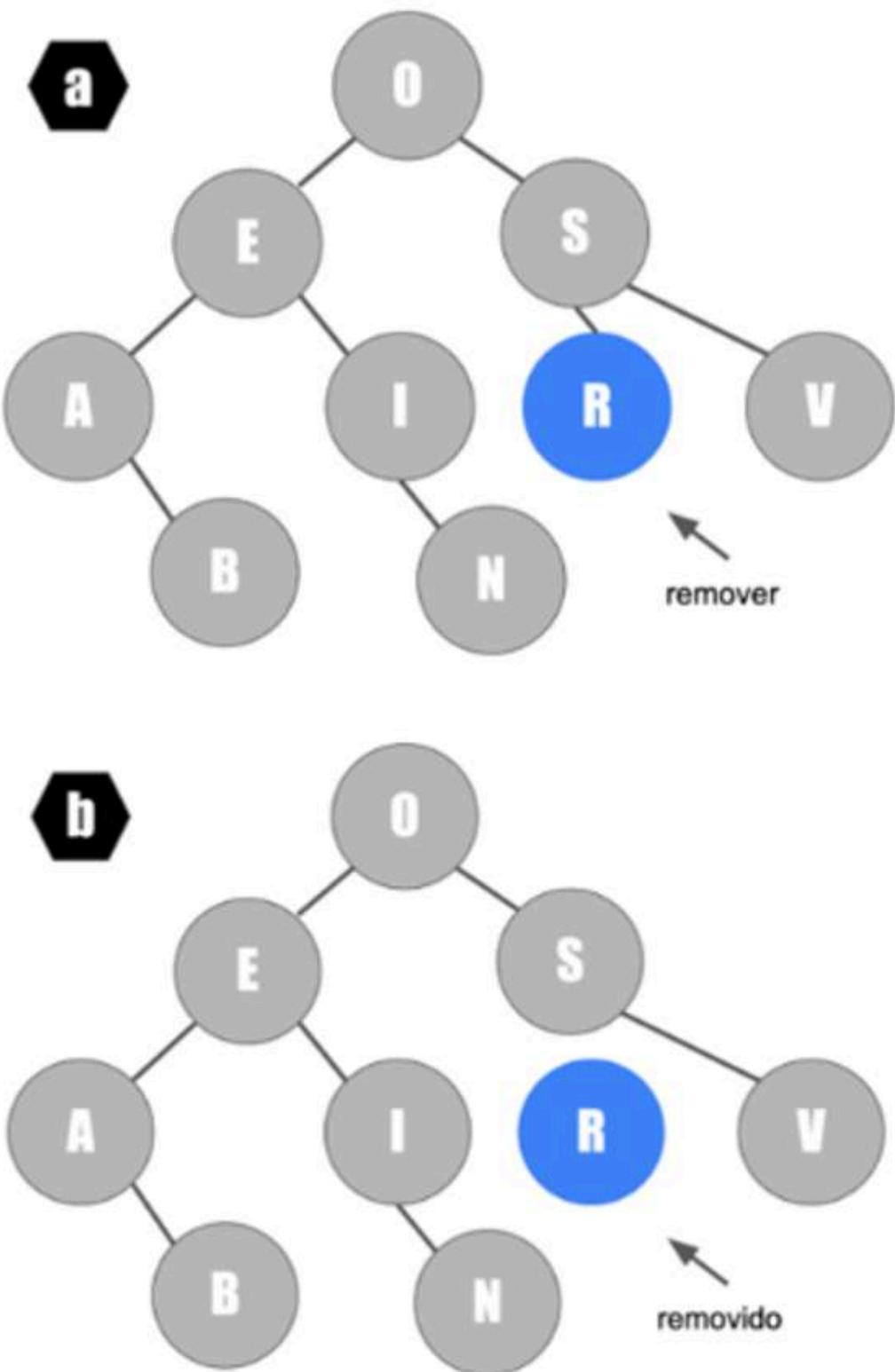


Figura 5 | Passos da remoção de um vértice-folha. Fonte: adaptada de Takenaka (2021).

Remoção de vértice com um filho

Neste cenário, o vértice a ser removido tem apenas um filho (esquerdo ou direito). A remoção envolve substituir o vértice pelo seu filho único no nó pai. Isso significa que o nó pai passa a apontar diretamente para o filho do vértice removido, efetivamente removendo o vértice original da árvore.

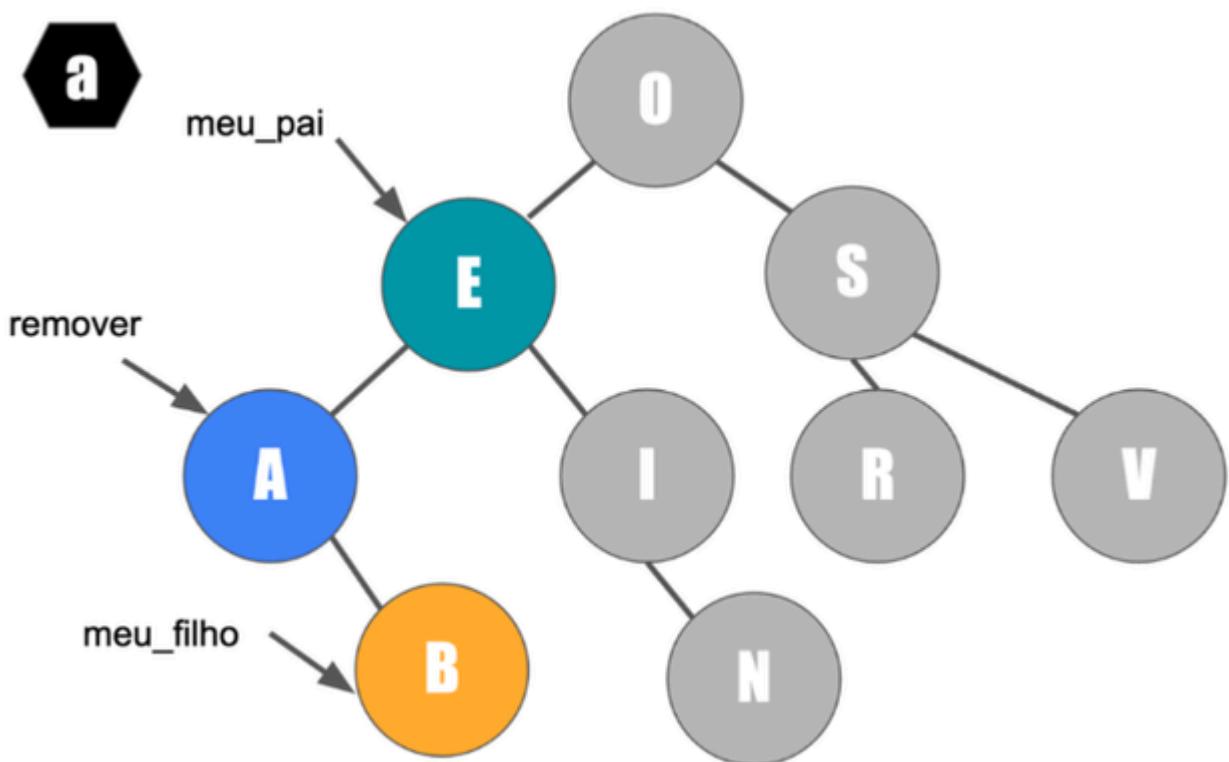


Figura 6 | Remover vértice pai de um filho – identificação de avô e neto. Fonte: adaptada de Takenaka (2021).

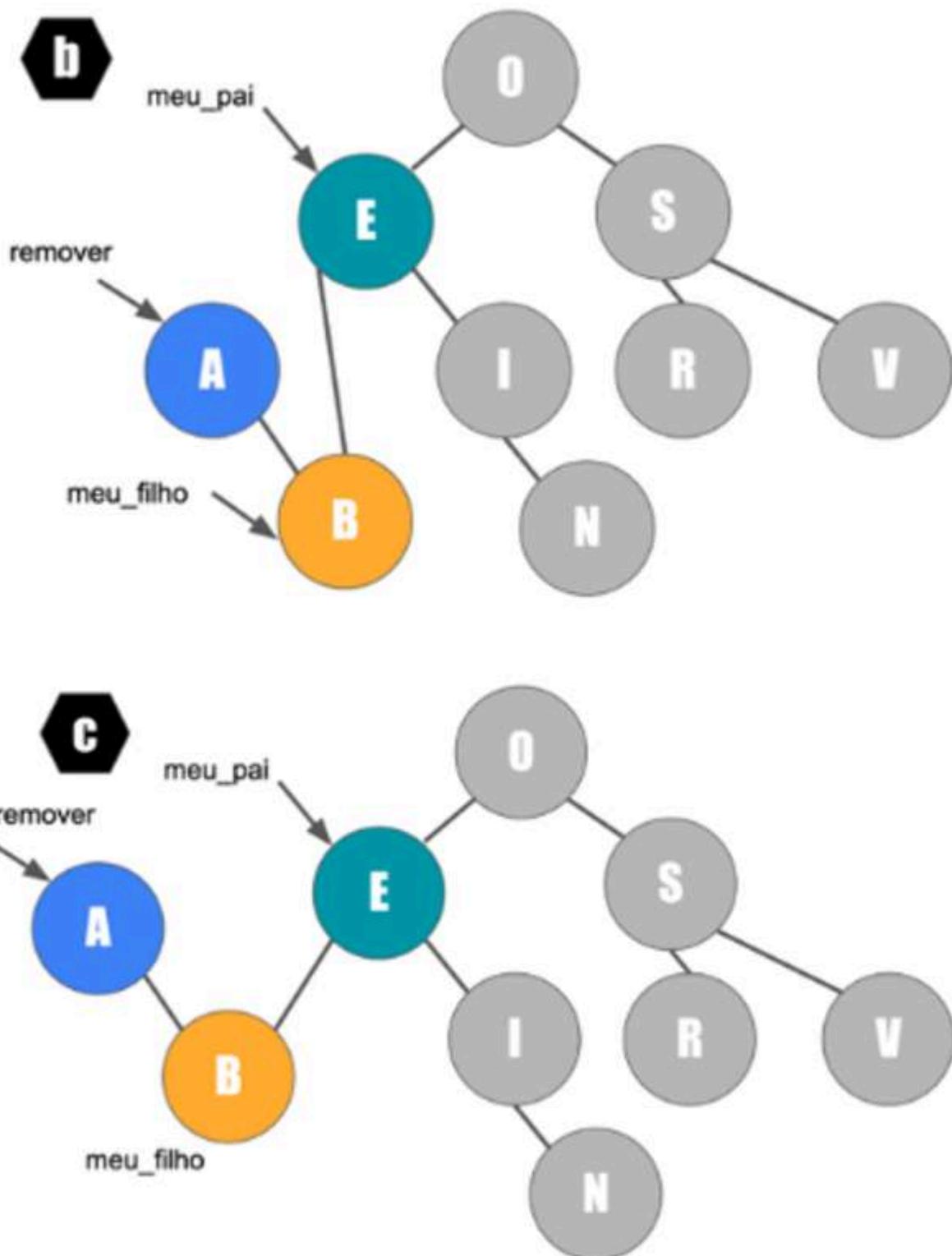


Figura 7 | Remover vértice pai de 1 filho – avô é pai e neto é filho. Fonte: adaptada de Takenaka (2021).

Remoção de vértice com dois filhos

Este é o caso mais complexo. Quando um vértice com dois filhos precisa ser removido, é necessário encontrar um substituto adequado que mantenha as propriedades da árvore binária de busca. Geralmente, escolhe-se o menor nó da subárvore direita do vértice a ser removido (conhecido como sucessor) ou o maior nó da subárvore esquerda (conhecido como antecessor). Após encontrar o substituto, este é movido para o local do vértice removido, e o processo de remoção é então aplicado ao substituto em sua posição original na árvore (Backes, 2023).

Observe o exemplo a seguir:

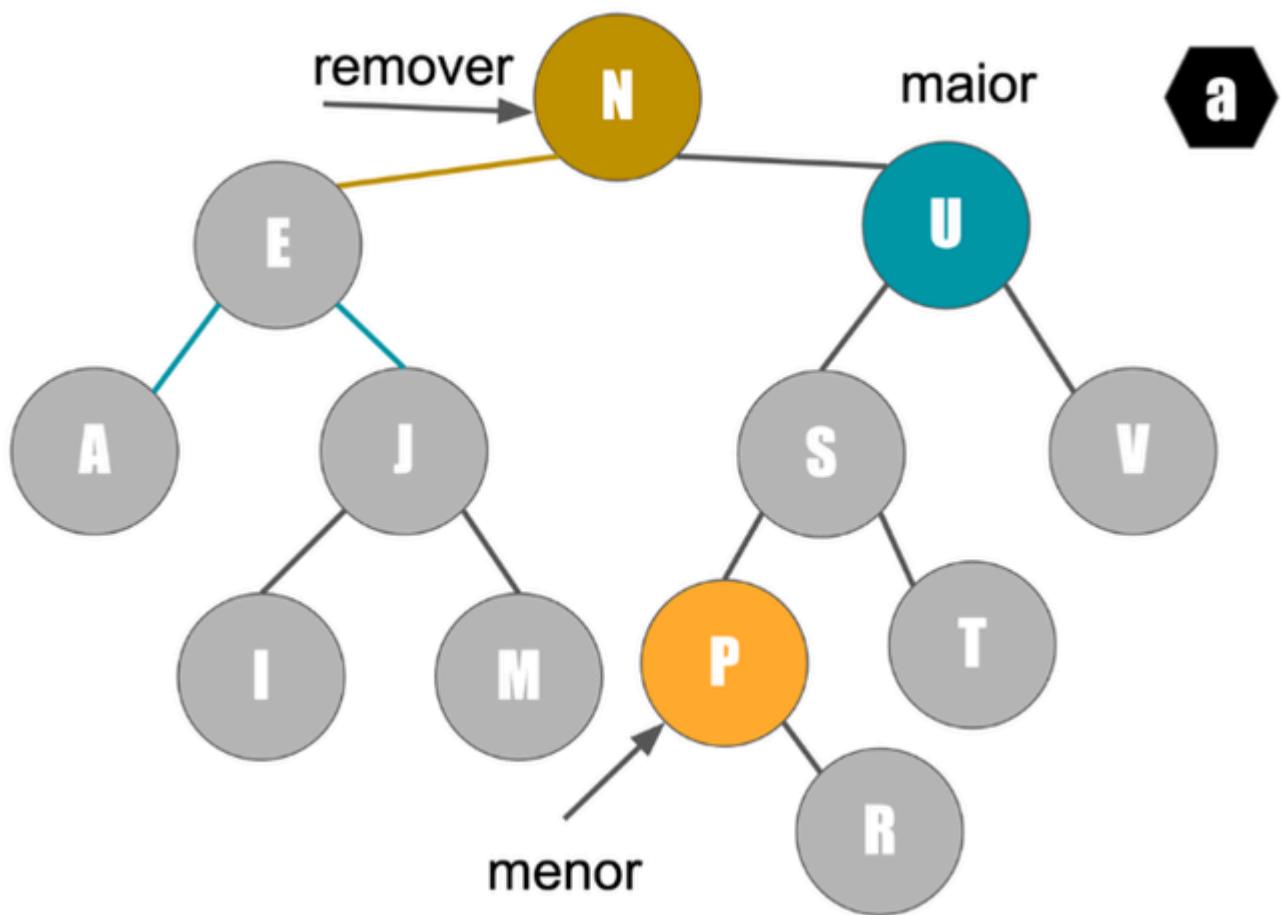


Figura 8 | Passo um da remoção de um vértice pai de dois filhos – identificação da menor chave do lado direito. Fonte: adaptada de Takenaka (2021).

Ocorre duas atribuições simultâneas, trocando os valores entre os dois.

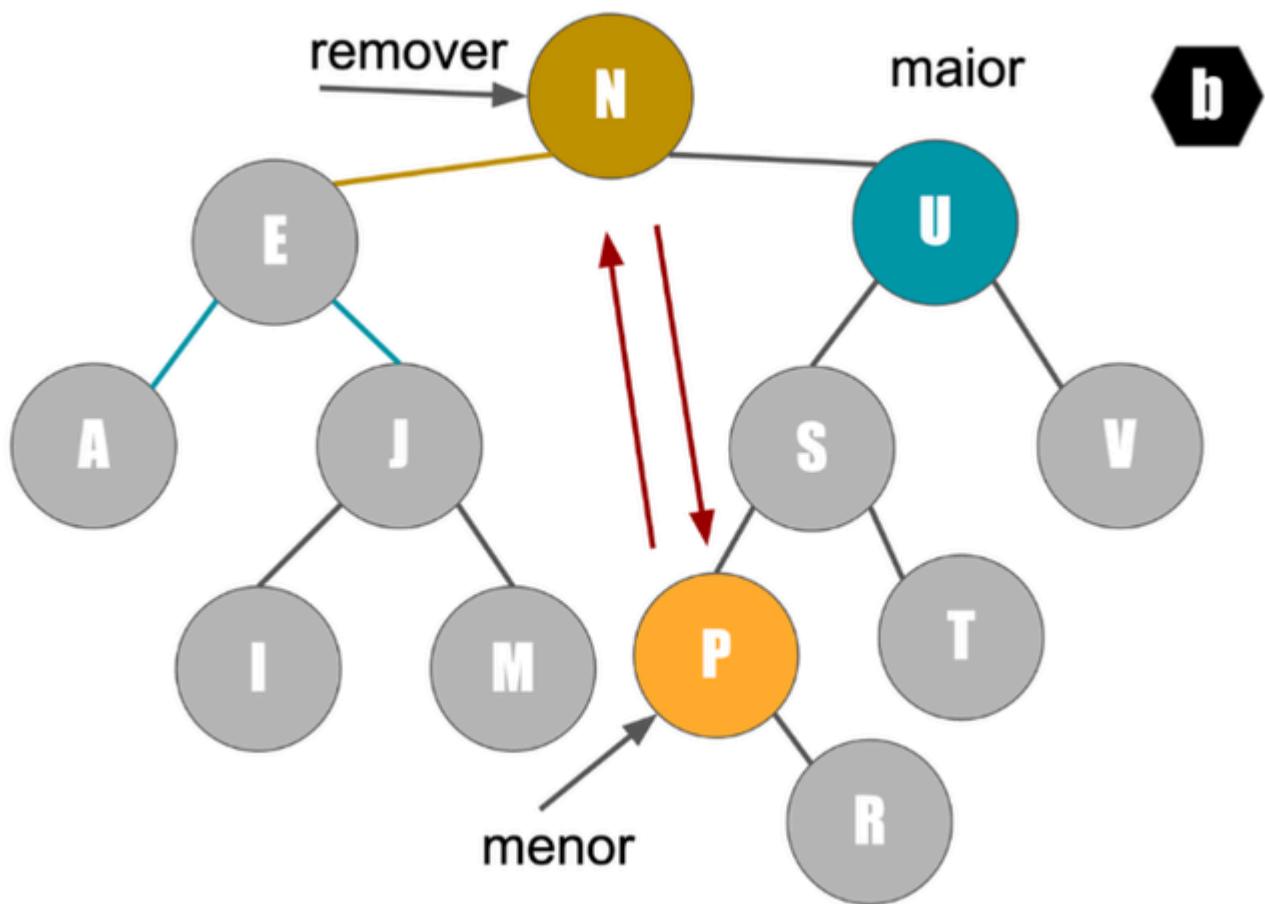


Figura 9 | Passo 2 da remoção de um vértice pai de dois filhos – troca das chaves. Fonte: adaptada de Takenaka (2021).

Note que como ele tomou o lugar do menor do lado direito, ou ele é um vértice-folha ou um vértice pai de um filho, que são casos já tratados anteriormente, então, deixamos a recursividade cuidar da remoção do vértice.

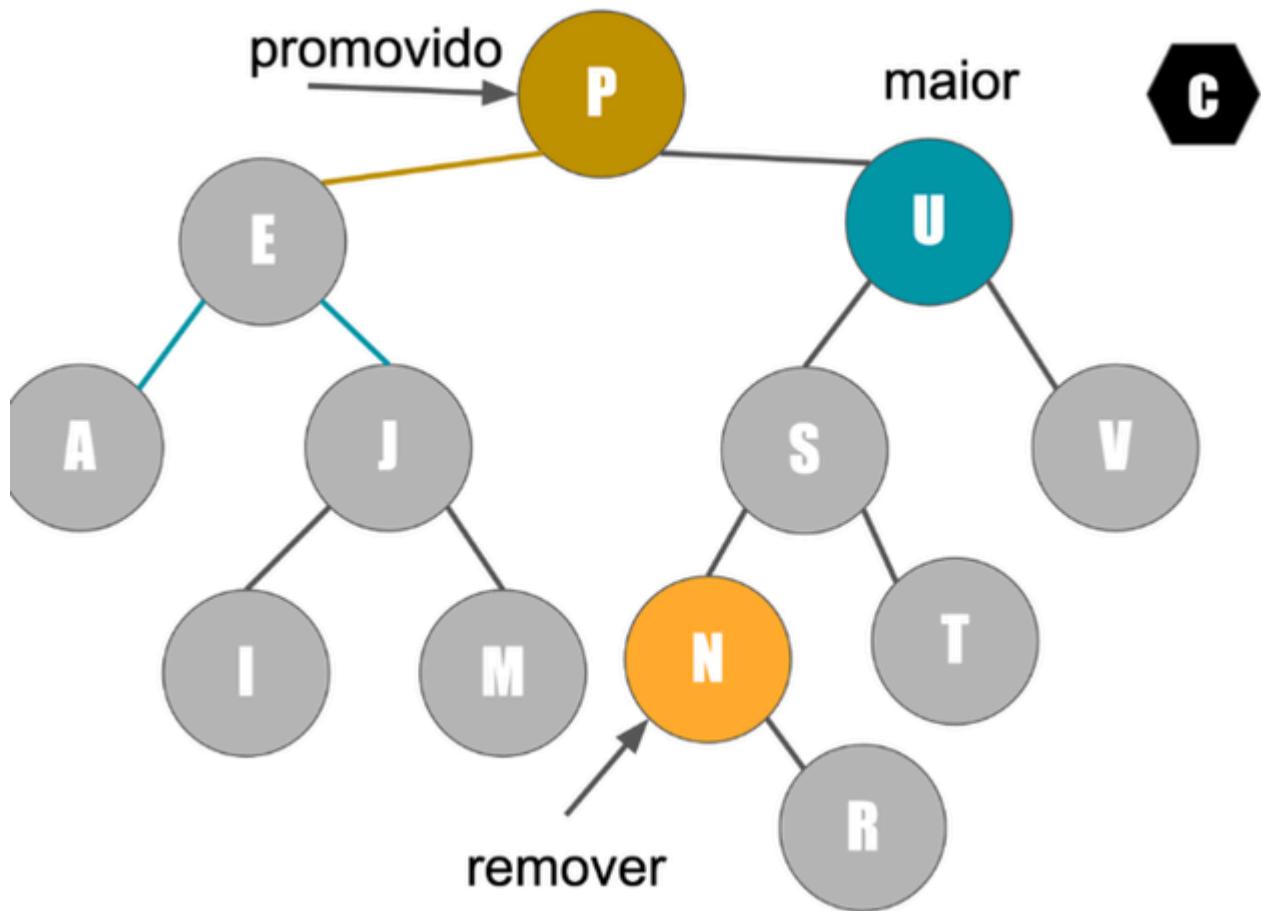


Figura 10. Fonte: adaptada de Takenaka (2021).

Vamos Exercitar?

O exercício proposto no ponto de partida te desafiava a desenvolver um aplicativo de lista de tarefas usando uma Árvore Binária de Busca (ABB). O foco dessa atividade estará em três operações principais: inserção, remoção e busca de tarefas. Vamos representar cada tarefa como um nó na ABB, onde a chave é a prioridade da tarefa.

```
class No:  
    def __init__(self, key):  
        self.left = None  
        self.right = None  
        self.key = key  
  
class ArvoreBB:  
    def __init__(self):  
        self.root = None
```

ESTRUTURA DE DADOS

```
def inserir(self, key):
    if self.root is None:
        self.root = No(key)
    else:
        self._inserir_recursivo(self.root, key)

def _inserir_recursivo(self, current_No, key):
    if key < current_No.key:
        if current_No.left is None:
            current_No.left = No(key)
        else:
            self._inserir_recursivo(current_No.left, key)
    else:
        if current_No.right is None:
            current_No.right = No(key)
        else:
            self._inserir_recursivo(current_No.right, key)

def remover(self, key):
    self.root = self._remover_recursivo(self.root, key)

def _remover_recursivo(self, current_No, key):
    if current_No is None:
        return current_No

    if key < current_No.key:
        current_No.left = self._remover_recursivo(current_No.left, key)
    elif key > current_No.key:
        current_No.right = self._remover_recursivo(current_No.right, key)
    else:
        if current_No.left is None:
            return current_No.right
        elif current_No.right is None:
            return current_No.left
        else:
            current_No.key = self._busca_valor_min(current_No.right)
            current_No.right = self._remover_recursivo(current_No.right, current_No.key)
    return current_No

def _busca_valor_min(self, No):
    while No.left is not None:
        No = No.left
    return No.key

def buscar(self, key):
    return self._busca_recursivo(self.root, key)
```

```
def _busca_recursivo(self, current_No, key):
    if current_No is None or current_No.key == key:
        return current_No
    if key < current_No.key:
        return self._busca_recursivo(current_No.left, key)
    return self._busca_recursivo(current_No.right, key)

# Exemplo de uso
bst = ArvoreBB()
bst.inserir(3) # Adiciona tarefa com prioridade 3
bst.inserir(1) # Adiciona tarefa com prioridade 1
bst.inserir(4) # Adiciona tarefa com prioridade 4

# Verifica se a tarefa com prioridade 1 existe
print( bst.buscar(1) is not None)

# remover a tarefa com prioridade 3
bst.remover(3)
print( bst.buscar(3) is not None)
```

Inserção:

- Cada tarefa é adicionada à árvore com base em sua prioridade. Se a prioridade for menor do que a do nó atual, ela é inserida à esquerda, caso contrário, à direita.

Remoção:

Ao remover um nó, consideramos três casos:

- Se o nó é uma folha, é removido diretamente.
- Se o nó tem apenas um filho, ele é substituído por esse filho.
- Se o nó tem dois filhos, substituímos o nó pelo menor nó da sua subárvore direita.

Busca:

- Para encontrar uma tarefa, comparamos sua prioridade com as chaves dos nós e seguimos pela esquerda ou direita conforme necessário.

Essa implementação fornece uma base para um aplicativo de lista de tarefas eficiente, onde as tarefas podem ser facilmente adicionadas, removidas e buscadas com base em suas prioridades.

Saiba mais

Árvores de busca:

- TONIN, N. [Árvore Binária de Busca](#). BeeCrowd.

Aplicações de árvores de busca binária em Python:

- ALBUQUERQUE, B. M. [O que são árvores binárias?](#) StatPlace.

Ordenação em árvores:

- DANIEL, M. [Brincando de arvore binária com Python](#). Me deu Branco!

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

BACKES, A. R. **Algoritmos e estruturas de dados em Linguagem C**. Rio de Janeiro: LTC, 2023.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

TAKENAKA, R. M. **Fundamentos de árvores e algoritmos**. Estrutura de Dados, 2021. Disponível em: <http://tinyurl.com/bdfmd7h3>. Acesso em: 1 fev. 2024.

Aula 3

Árvores B e Quadtrees

Árvores B e Quadtrees

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

ESTRUTURA DE DADOS

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Nesta videoaula, descobriremos estruturas de dados avançadas, começando com as árvores B, uma estrutura fundamental para gerenciamento eficiente de banco de dados e sistemas de arquivos. Exploraremos suas aplicações práticas, destacando como elas otimizam operações de busca, inserção e remoção. Em seguida, avançaremos para as quadtrees, uma estrutura especializada para representação espacial e otimização de buscas em áreas como processamento de imagens e simulações físicas.

Prepare-se para uma jornada enriquecedora pelo universo das estruturas de dados!

Ponto de Partida

Olá, estudante!

As árvores são estruturas de dados versáteis, capazes de facilitar uma ampla gama de soluções para problemas complexos. Considere, por exemplo, a organização de espaço em discos rígidos ou a gestão de bancos de dados, ambos os casos em que as árvores oferecem uma estruturação eficaz. Elas também são aplicáveis na modelagem de espaços bidimensionais para o processamento de imagens, entre outras utilidades.

Especificamente, as árvores B desempenham um papel significativo no gerenciamento de banco de dados e espaço em disco, manuseando eficientemente grandes volumes de dados. Essas estruturas são fundamentais no desenvolvimento de sistemas de gerenciamento de banco de dados renomados como MongoDB e MySQL, e também são empregadas nos sistemas operacionais para a administração de arquivos (Lambert, 2022).

Em um cenário prático, imagine que você é responsável por criar um leitor eletrônico de livros com funcionalidades para adicionar e remover anotações diretamente no conteúdo. Esse recurso visa enriquecer a experiência do usuário, permitindo marcações e comentários durante a leitura, potencialmente elevando a satisfação do cliente e atraindo novos usuários. Para implementar essa funcionalidade, é necessário desenvolver um sistema *back-end* que gerencie os dados das anotações, permitindo ao usuário adicionar, remover, consultar e listar todas as anotações feitas. A interface gráfica, ou *front-end*, não será foco deste desenvolvimento.

Esse projeto exigirá a implementação de operações específicas para gerenciar as anotações de forma eficaz. Ao concluir esta unidade, você estará equipado com o conhecimento necessário para aplicar as árvores B e quadtrees em situações reais, resolvendo problemas como este de maneira estruturada e eficiente.

Bons estudos!

Vamos Começar!

Introdução a árvores B

Árvores B são estruturas de dados hierárquicas e não lineares, destinadas ao armazenamento organizado de informações, facilitando operações como busca, inserção, remoção e navegação de forma eficaz. Essa estrutura é particularmente adequada para aplicações que manipulam extensos volumes de dados e exigem operações de leitura e escrita de grandes blocos de dados, tais como sistemas de gerenciamento de banco de dados e sistemas de arquivos. Em contraste com as árvores binárias de busca, que limitam cada nó a dois descendentes, os nós em uma árvore B podem ter um número substancialmente maior de filhos, definido pelo fator de ramificação da árvore (Takenaka, 2021).

Os dados em uma árvore B são mantidos平衡ados, com cada nó contendo várias chaves ordenadas e ponteiros para outros nós. Essa organização permite que cada nó abrigue um número variável de chaves, respeitando um limite preestabelecido, facilitando que as operações mencionadas sejam executadas em tempo logarítmico. Isso é particularmente eficiente para grandes conjuntos de dados, garantindo acesso rápido e organizado.

A manutenção do equilíbrio na estrutura das árvores B é assegurada através da divisão e fusão de nós quando necessário, particularmente durante inserções e remoções. Essa capacidade de ajuste garante o balanceamento contínuo da árvore, otimizando o acesso aos dados armazenados.

Aplicações de árvores B

Desenvolvidas por Rudolf Bayer e Edward McCreight em 1970, as árvores B são amplamente empregadas em sistemas de banco de dados e sistemas de arquivos devido à sua eficiência e capacidade de gestão de grandes volumes de dados (Takenaka, 2021). A estrutura de uma árvore B, portanto, facilita a organização de dados em páginas ou nós, com cada página podendo conter um número específico de chaves e ponteiros, organizando as chaves de maneira que facilita a busca e o acesso aos dados relacionados (vide Figura 1).

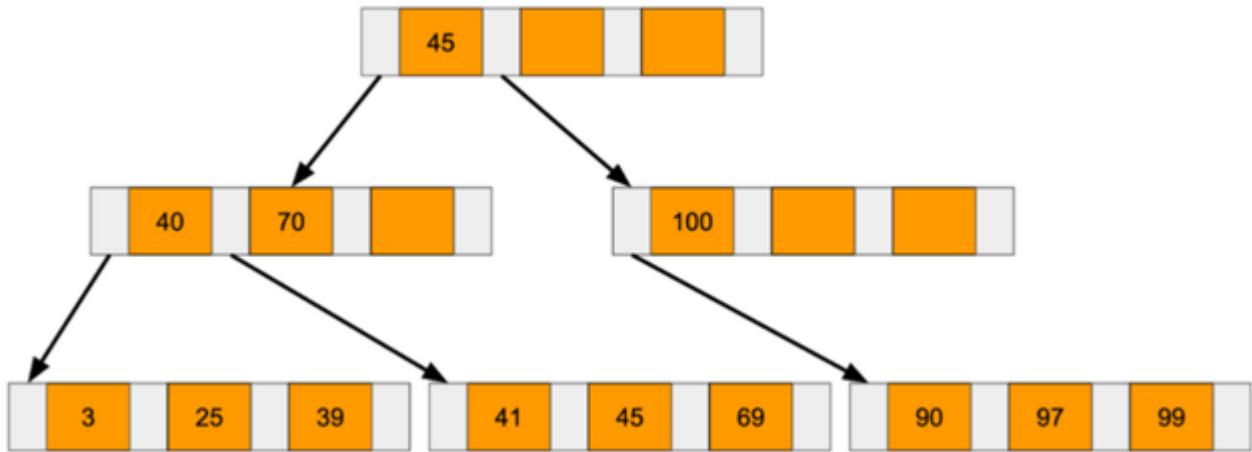


Figura 1 | Representação de árvore B. Fonte: adaptada de Takenaka (2021).

A ordem de uma árvore B, definida como o número máximo de ponteiros que uma página pode ter, é estabelecida com base na capacidade máxima de chaves que uma página pode conter, garantindo que a ocupação de chaves em cada página nunca seja inferior à metade da sua capacidade total. Essa característica permite que as árvores B lidem com uma vasta gama de aplicações, oferecendo uma solução eficiente para a organização e recuperação de informações em sistemas complexos (Szwarcfiter; Markenzon, 2020).

As árvores B representam uma categoria de estruturas de dados平衡adas que empregam procedimentos específicos de balanceamento durante as operações de **inserção** e **remoção** para preservar suas propriedades de organização. Diferentemente da árvore AVL (que iremos estudar posteriormente), onde o foco está em manter um balanceamento estrito através de rotações, as árvores B lidam com a complexidade adicional decorrente da presença de múltiplas chaves em cada vértice. Uma das características distintivas das árvores B é que suas folhas, ou páginas-folha, estão posicionadas no mesmo nível hierárquico, e o crescimento da estrutura ocorre de baixo para cima, das folhas em direção à raiz.

O processo de inserção em uma árvore B começa pela localização da página adequada para a nova chave, verificando-se a existência de espaço disponível. Se houver espaço, a chave é inserida de forma ordenada. Na eventualidade de a página estar cheia, procede-se à sua divisão, ou "split", criando uma nova página que alojará metade dos elementos da página original. Essa divisão exige que uma das chaves ascenda ao nível superior para manter a integridade estrutural da árvore, garantindo a correta referência às páginas divididas.

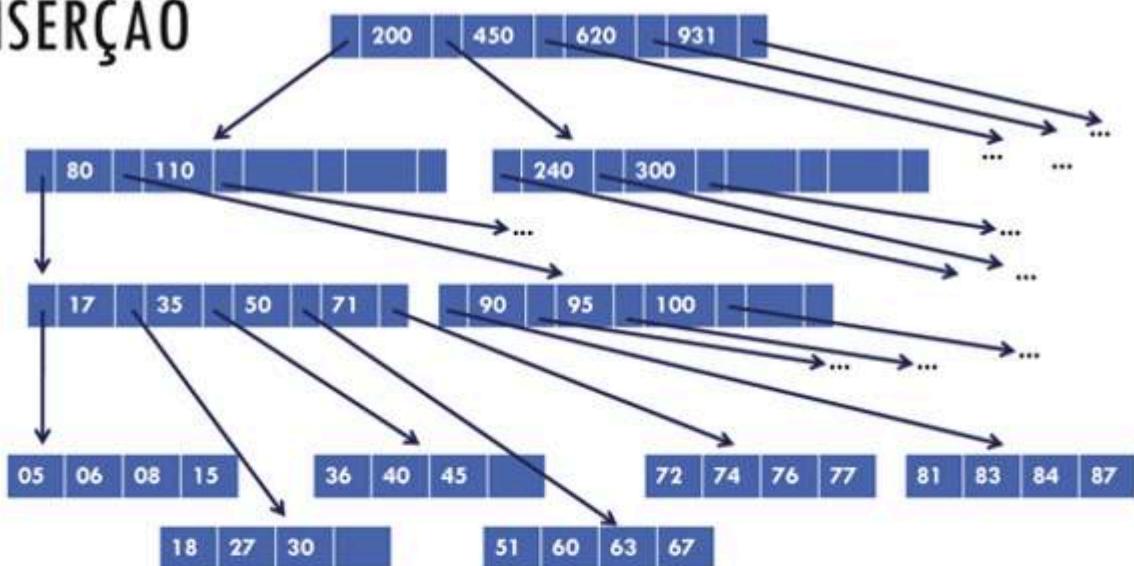
A inserção em uma árvore B segue uma série de passos para localizar o local apropriado para a nova chave e garantir que a árvore permaneça balanceada.

Localização da página apropriada: inicialmente, percorre-se a árvore a partir da raiz, seguindo os ponteiros adequados, para encontrar a página (nó) onde a nova chave deve ser inserida.

Inserção na página: se a página encontrada tem espaço para a nova chave, a chave é inserida mantendo a ordem das chaves na página.

Divisão da página (*split*): se a página estiver cheia, ela é dividida ao meio, criando uma nova página. Uma chave da página original é movida para o nó pai para servir como ponto de separação entre as duas novas páginas.

INSCRIÇÃO



Inserir chave 32

INSCRIÇÃO

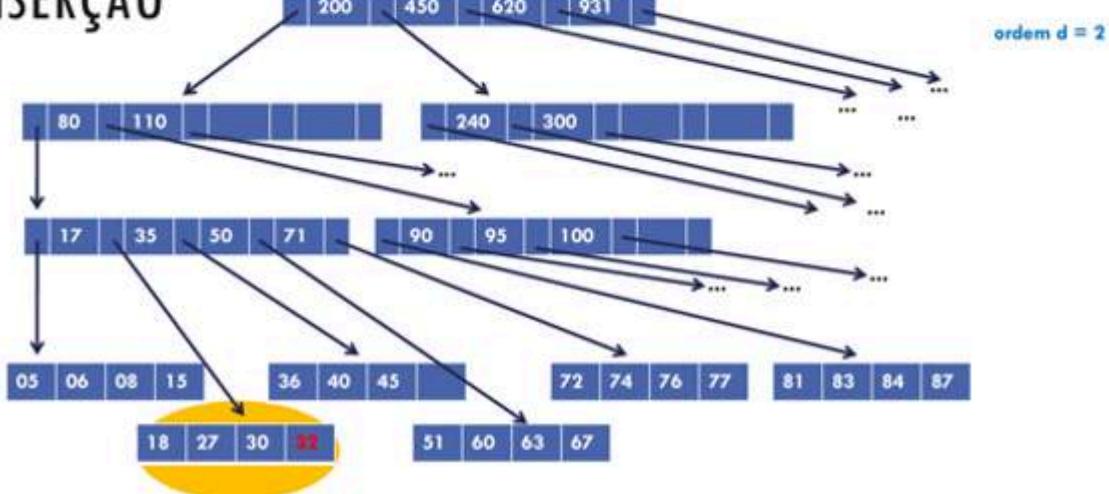


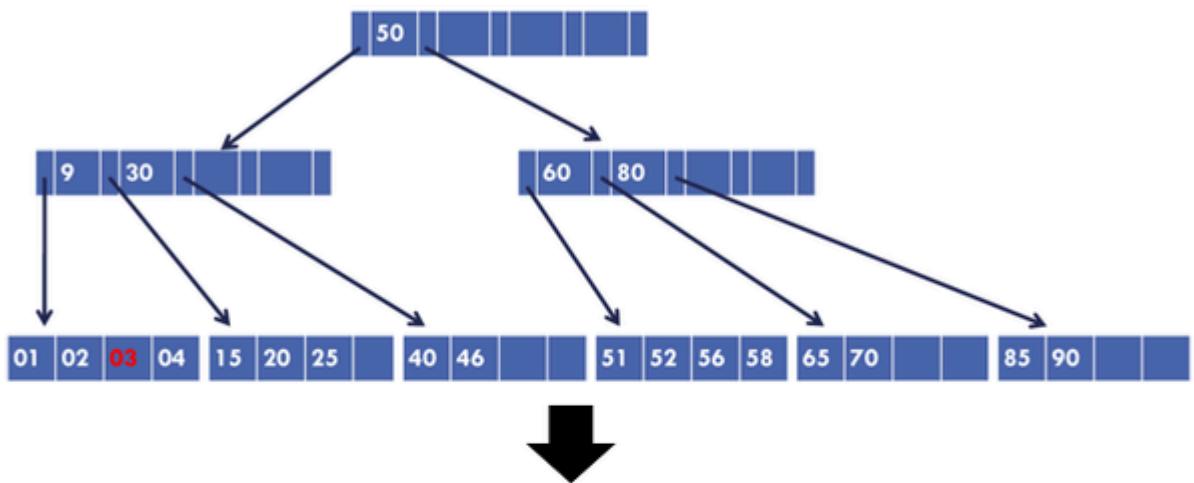
Figura 2 | Exemplo de inserção do número 32 em uma B-Tree. Fonte: adaptada de Braganholo (2020).

A remoção em árvores B é mais complexa devido à necessidade de manter a árvore balanceada após a remoção de uma chave.

Remoção de uma chave da folha: se a chave a ser removida está em uma folha e a folha tem chaves suficientes, simplesmente remove-se a chave.

Remoção de uma chave de um nó interno: se o nó não é uma folha, há várias estratégias para lidar com a remoção, como substituir a chave por seu predecessor ou sucessor imediato, ou, se necessário, fundir nós.

EXCLUSÃO DA CHAVE 03



EXCLUSÃO DA CHAVE 03

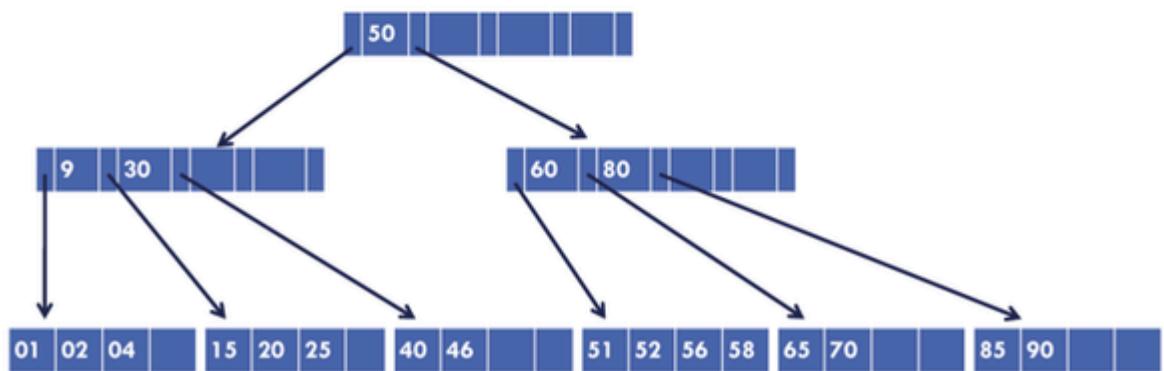
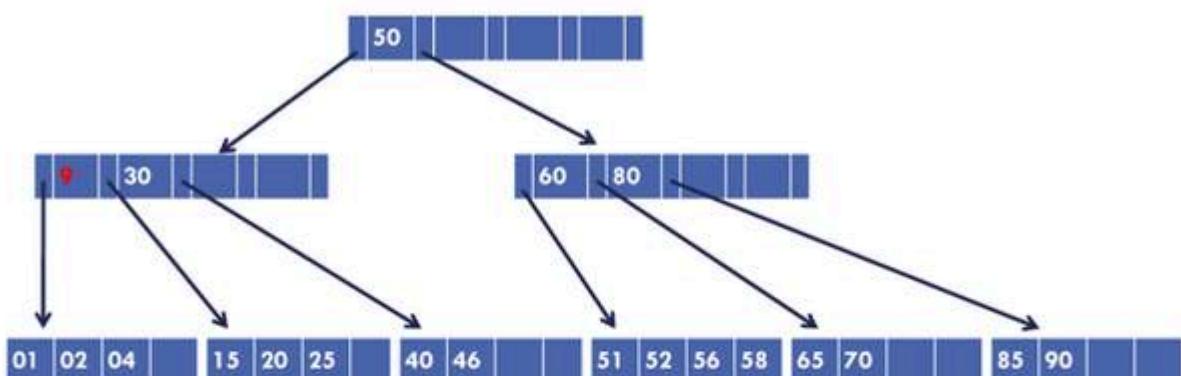


Figura 3 | Remoção de um nó folha (03). Fonte: adaptada de Braganholo (2020).

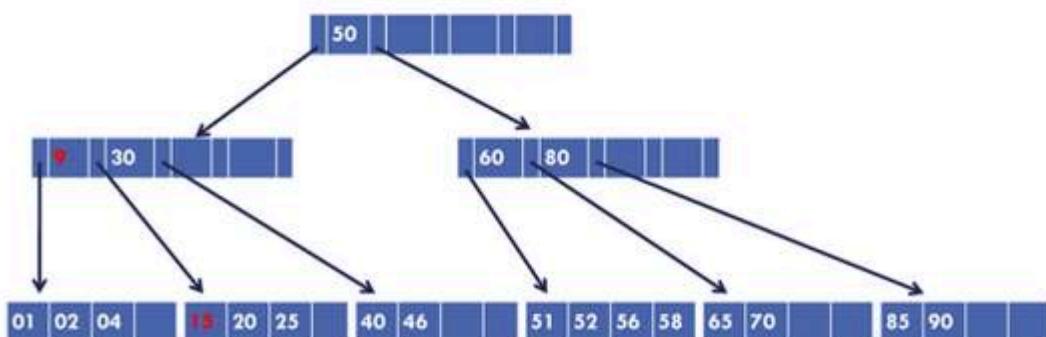
EXCLUSÃO DA CHAVE 9



Substituir pela chave imediatamente maior



EXCLUSÃO DA CHAVE 9



Substituir pela chave imediatamente maior

Figura 4 | Remoção de um nó interno (9). Fonte: adaptada de Braganholo (2020).

Em síntese, as árvores B são utilizadas extensivamente em sistemas de gerenciamento de banco de dados e sistemas de arquivos, elas otimizam o acesso a registros e arquivos, promovendo uma recuperação de dados rápida e eficiente (Alves, 2021). Embora exemplifiquemos com um número limitado de chaves por página, na prática, uma página em uma árvore B pode conter centenas de chaves, evidenciando a capacidade dessa estrutura de manipular grandes volumes de dados de maneira eficaz. Essa abordagem permite o

gerenciamento eficiente de dados complexos, destacando-se como uma solução robusta para a organização e o acesso rápido a grandes conjuntos de informações.

Siga em Frente...

Quadtree

A quadtree é uma estrutura de dados em árvore em que cada nó interno possui exatamente quatro filhos, representando uma maneira eficaz de partitionar um espaço bidimensional. É amplamente aplicada no processamento de imagens, em aplicações de jogos digitais e na compactação de imagens, possibilitando a representação de espaços e objetos de maneira hierárquica e eficiente.

Considerando uma imagem quadrada composta por diversos pixels, a quadtree permite sua decomposição em quatro quadrantes de dimensões iguais, cada um representado por um nó na estrutura da árvore, como ilustrado na Figura 5. Esse processo de decomposição é repetido recursivamente para cada quadrante até que se atinja um nível onde a divisão adicional não é mais necessária ou possível, tipicamente quando um quadrante se torna homogêneo em termos de cor ou quando atinge o tamanho de um pixel.

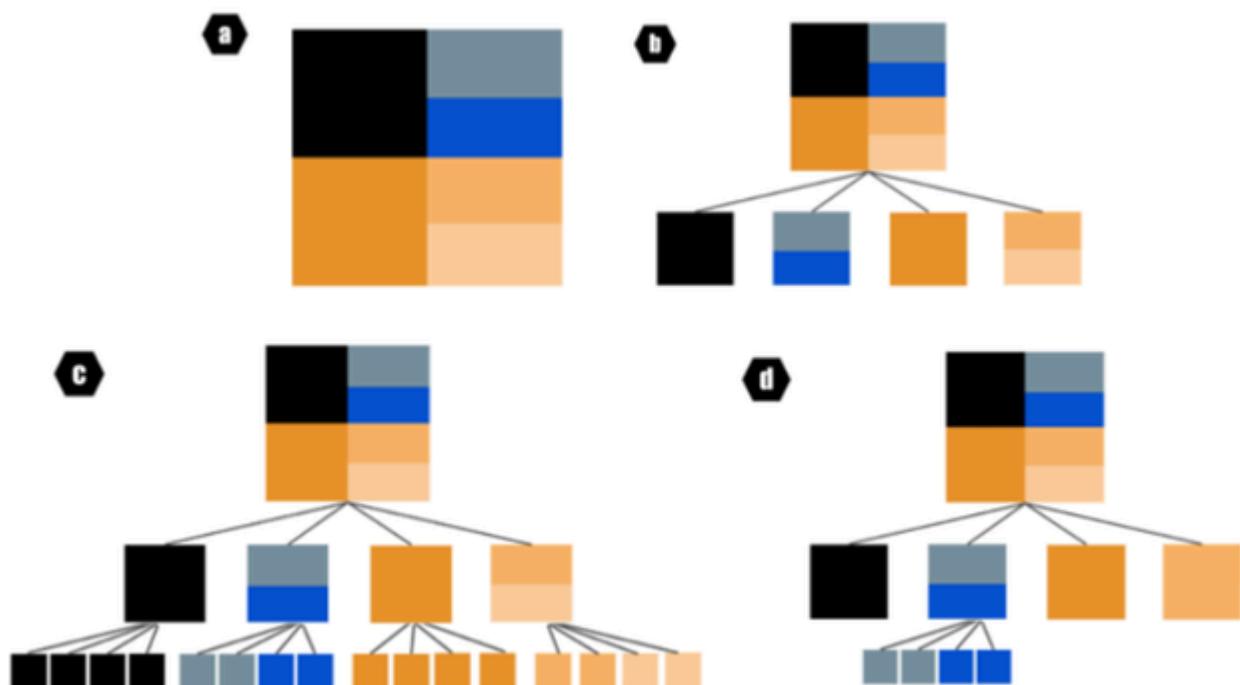


Figura 5 | Representação de quadtree. Fonte: adaptada de Takenaka (2021).

Nesse contexto, uma aplicação significativa das Quadtrees está na compactação de imagens. Quando um nó possui filhos que são suficientemente homogêneos em termos de cor ou padrão,

estes podem ser substituídos por um único nó, reduzindo a complexidade da representação da imagem sem comprometer significativamente sua integridade visual. Por exemplo, se um nó tem quatro filhos todos de cor preta, este nó pode ser simplificado para um nó folha preto, reduzindo o número total de nós necessários para representar a imagem.

A implementação de uma quadtree em Python começa com a definição da classe **Quadtree**, que inicializa os quatro quadrantes (Nordeste, Noroeste, Sudeste, Sudoeste) e possivelmente outros atributos relevantes como a altura da árvore. Métodos como **inserir_vertice** e **obter_vertice** são fundamentais para manipular a árvore, permitindo a inserção de novos nós com base nas características dos quadrantes da imagem e a recuperação da imagem ou de partes dela em diferentes níveis de detalhe.

```

1  class QuadTree:
...
15
16      def inserir_vertice(self, img):
17          self.imagem_com_cores_mescladas = obter_imagem_com_cores_mescladas(img)
18
19          imagem_dividida = dividir_em_4(img)
20
21          if imagens_diferentes(imagem_dividida):
22              h = self._altura + 1
23
24              self.NO = QuadTree(h)
25              self.NE = QuadTree(h)
26              self.SO = QuadTree(h)
27              self.SE = QuadTree(h)
28
29              self.NO.inserir_vertice(imagem_dividida[0])
30              self.NE.inserir_vertice(imagem_dividida[1])
31              self.SO.inserir_vertice(imagem_dividida[2])
32              self.SE.inserir_vertice(imagem_dividida[3])
33
34      return self

```



```

1  class QuadTree:
...
36      def obter_vertice(self, altura):
37          if self.folha or self._altura == altura:
38              return self.imagem_com_cores_mescladas
39
40          return concatenar(
41              self.NO.obter_vertice(altura),
42              self.NE.obter_vertice(altura),
43              self.SO.obter_vertice(altura),
44              self.SE.obter_vertice(altura))

```

Figura 6 | Exemplo de código em Python de uma quadtree. Fonte: adaptada de Takenaka (2021).

Essa estrutura oferece uma abordagem eficiente para a gestão de dados espaciais e imagéticos, permitindo operações de busca e manipulação rápidas e minimizando o espaço de armazenamento necessário, o que é de grande importância em contextos como o desenvolvimento de jogos digitais, processamento de imagens e sistemas de informações geográficas.

Vamos Exercitar?

O exercício proposto no ponto de partida lhe pedia para desenvolver um leitor eletrônico de livros com funcionalidades para adicionar e remover anotações diretamente no conteúdo. A estratégia para implementar essa funcionalidade é criar uma árvore B, onde cada nó (ou "página", na terminologia das árvores B) contém informações sobre as anotações, como a localização no livro (por exemplo, número da página e parágrafo) e o texto da anotação. A chave de ordenação poderia ser a localização da anotação no livro, permitindo que as anotações sejam rapidamente encontradas, listadas em ordem ou modificadas.

ESTRUTURA DE DADOS

Vamos fornecer uma versão muito simplificada e conceitual da implementação, focando na estrutura da árvore B e nas operações básicas de inserção e remoção de anotações. Por simplicidade, não incluiremos todos os detalhes de balanceamento e divisão de nós, que são partes complexas da implementação de árvores B, mas focaremos na lógica básica:

```
class Anotacao:  
    def __init__(self, pagina, paragrafo, texto):  
        self.pagina = pagina  
        self.paragrafo = paragrafo  
        self.texto = texto  
        self.chave = (pagina, paragrafo) # Uma tupla como chave para ordenação  
  
class NodoB:  
    def __init__(self, t):  
        self.chaves = [] # Lista de anotações  
        self.filhos = [] # Lista de nodos filhos  
        self.t = t # Grau mínimo da árvore B (determina o número máximo de chaves)  
  
class ArvoreB:  
    def __init__(self, t):  
        self.raiz = NodoB(t)  
        self.t = t  
  
    def inserir(self, anotacao):  
        # Implementação simplificada de inserção  
        # Esta função deve localizar o nodo correto e inserir a anotação, dividindo o nodo se  
necessário  
        ToDo  
  
    def remover(self, pagina, paragrafo):  
        # Implementação simplificada de remoção  
        # Esta função deve localizar a anotação e removê-la, realizando a fusão de nodos se  
necessário  
        ToDo  
  
    def consultar(self, pagina, paragrafo):  
        # Retorna a anotação na localização especificada, se existir  
        ToDo  
  
    def listar_anotacoes(self):  
        # Retorna uma lista de todas as anotações, em ordem  
        ToDo
```

Essa é apenas uma sugestão de como você poderia estruturar sua solução, ficando a seu critério escolher a melhor forma para tal.

ESTRUTURA DE DADOS

Bons estudos!

Saiba mais

Introdução a árvores B:

- BRAGANHOLO, V. [Árvore B. Estruturas de Dados e Seus Algoritmos.](#)

Aplicações de árvores B:

- MOURA, C. [Árvore B: O que é e para que serve? Medium.](#)

Quadtrees:

- QUADTREE. [Geocities](#), [s. d.].

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

BRAGANHOLO, V. Á. B. **Árvore B**. Estruturas de dados e seus algoritmos, 2020. Disponível em: <https://braganholo.github.io/material/ed/11-ArvoreB.pdf>. Acesso em: 30 jan. 2024.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

TAKENAKA, R. M. **Fundamentos de árvores e algoritmos**. Estrutura de Dados, 2021. Disponível em: <http://tinyurl.com/bdfmd7h3>. Acesso em: 28 jan. 2024.

Aula 4

Árvores AVL

Árvores AVL



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Nesta videoaula exploraremos uma estrutura de dados capaz de se autobalancear. Nos aprofundaremos na teoria, implementação e aplicações práticas das árvores AVL. Ao longo desta aula, você aprenderá a implementar árvores AVL em Python, compreendendo os mecanismos de rotação que mantêm a árvore balanceada. Além disso, discutiremos diversas aplicações práticas das árvores AVL em sistemas de computação, destacando seu papel fundamental na otimização de buscas e na manutenção de dados ordenados. Prepare-se para mergulhar nesse tópico fascinante!

Ponto de Partida

Olá, estudante!

Sejam bem-vindos à aula sobre árvores AVL, um conceito fundamental na ciência da computação que aprimora significativamente a eficiência de estruturas de dados.

As árvores que serão estudadas nesta aula são as árvores平衡adas, ou seja, árvores que possuem uma distribuição mais homogênea entre as subárvores e em vários níveis. No decorrer dos seus estudos, você entenderá que outros tipos de árvores podem ser tão desbalanceadas que se tornam uma lista, perdendo a característica de ser binária, consequentemente tendo uma queda no desempenho.

A característica distintiva dessas árvores é a garantia de que a diferença de altura entre as subárvores esquerda e direita de qualquer nó não exceda uma unidade. Esse equilíbrio estrito melhora consideravelmente as operações de busca, inserção e remoção, mantendo o tempo de execução dentro de limites logarítmicos (Lambert, 2022).

Ao longo desta aula, exploraremos como as árvores AVL alcançam esse balanceamento, as técnicas de rotação utilizadas para manter a árvore equilibrada após cada inserção ou remoção e como essas estratégias contribuem para a eficácia geral do algoritmo. Visando garantir sua compreensão do conteúdo, imagine que você está desenvolvendo um sistema de gestão de biblioteca digital que precisa armazenar e recuperar informações sobre livros rapidamente, mesmo à medida que o número de títulos cresce exponencialmente.

A eficiência na busca, inserção e remoção de registros do catálogo é essencial para manter um bom desempenho do sistema e a satisfação do usuário. Para atender a esses requisitos, você decide implementar uma estrutura de dados baseada em Árvore AVL, que garantirá um balanceamento automático e manterá as operações em tempo logarítmico (Takenaka, 2021).

Prepare-se para mergulhar na teoria e prática das árvores AVL, entendendo seu impacto e suas aplicações no mundo da computação.

Bons estudos!

Vamos Começar!

Introdução a árvores AVL

Ao lidar com árvores binárias de busca, observa-se que determinadas situações podem resultar em estruturas “subótimas”. Isso ocorre quando a disposição dos nós conduz a uma configuração onde a busca, idealmente binária e de tempo logarítmico, degrada para uma busca linear, conforme exemplificado na Figura 1.

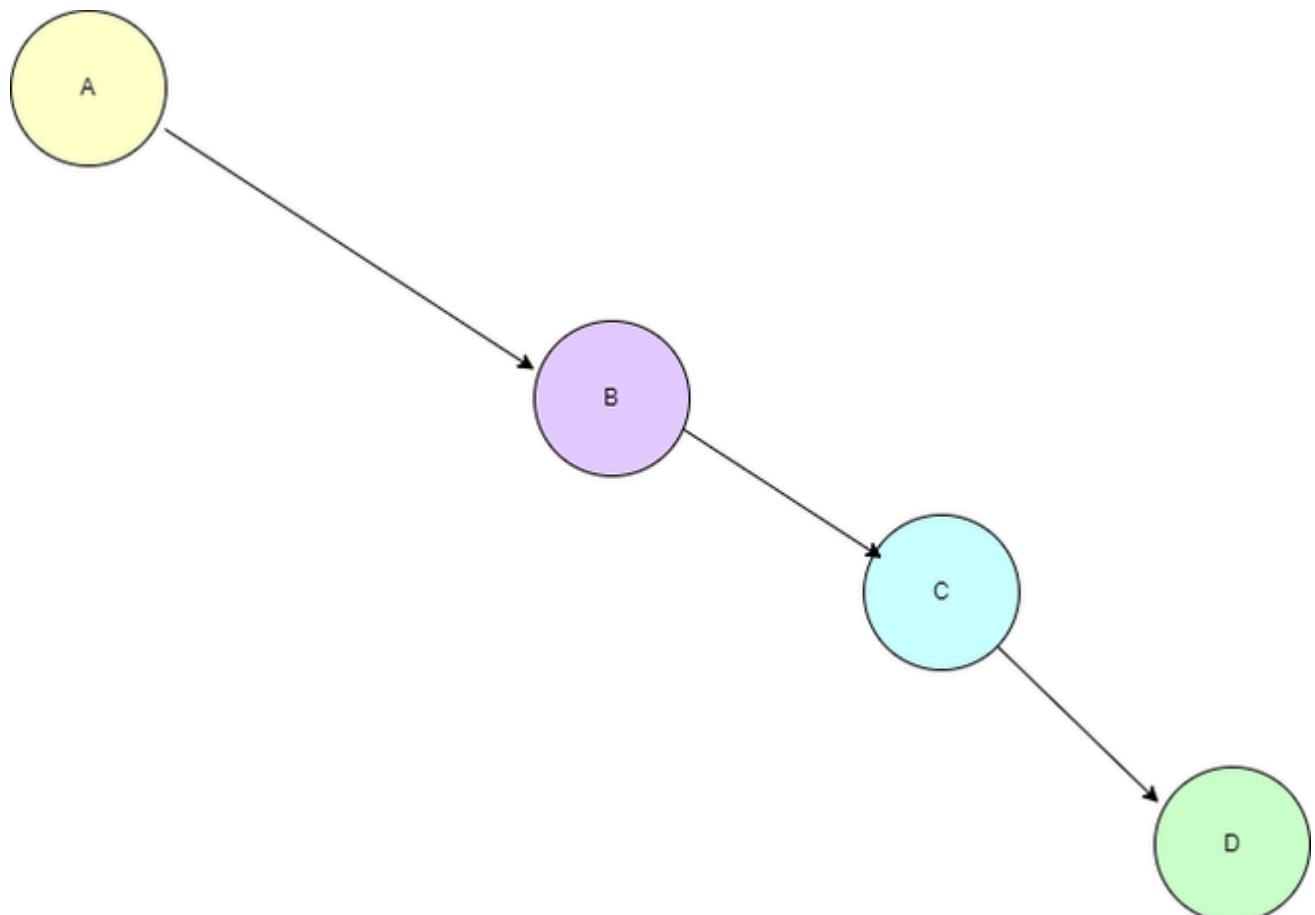


Figura 1 | Exemplo de árvore binária de busca degenerada. Fonte: elaborada pela autora.

As árvores degeneradas representam um tipo específico de estrutura de dados em que cada nó tem apenas um filho, à exceção do último nó, conhecido como nó folha. Nessa configuração, o número total de nós na árvore é igual à sua altura (a distância máxima de qualquer nó à folha mais distante) mais um. Tal estrutura é comparável a uma lista encadeada, devido à sua forma linear, o que pode levar a uma eficiência reduzida na busca, dado que se assemelha a uma operação de busca linear em vez de uma busca binária mais rápida.

Para evitar a perda de desempenho associado às árvores degeneradas (pois acabam por se tornarem listas), é essencial o uso de árvores平衡adas, que distribuem de maneira mais uniforme os nós entre as subárvores esquerda e direita, mantendo as operações de busca, inserção e remoção eficientes (Lambert, 2022).

Uma solução para o problema das árvores degeneradas é a árvore AVL, uma árvore binária de busca平衡ada nomeada em homenagem aos seus inventores, Georgy Adelson-Velsky e Evgenii Landis. As árvores AVL garantem o balanceamento automático após cada operação de inserção ou remoção, ajustando-se conforme necessário para manter a diferença de altura entre as subárvores esquerda e direita de qualquer nó a no máximo um (Takenaka, 2021).

Para compreender e implementar o algoritmo de balanceamento AVL, é essencial familiarizar-se com conceitos fundamentais como subárvores (esquerda e direita), raiz de subárvore, filhos (esquerdo e direito) e a altura de um nó, pois esses conceitos são a base do mecanismo de balanceamento que diferencia as árvores AVL.

Principais conceitos de árvores AVL

Altura

A altura de uma árvore é determinada pela distância mais longa da raiz até um nó folha, contabilizando o número de arestas nesse caminho. Uma árvore é considerada平衡ada se a sua altura é proporcional ao logaritmo do número de nós, o que indica eficiência nas operações de busca, inserção e remoção (Takenaka, 2021).

Diferentes tipos de árvores possuem distintas metodologias para alcançar e manter esse equilíbrio. As árvores AVL, por exemplo, utilizam a altura de cada nó como critério para o balanceamento, realizando ajustes conforme necessário para manter a diferença de altura entre as subárvores esquerda e direita de qualquer nó a no máximo um. Por outro lado, as árvores rubro-negras empregam uma abordagem baseada na coloração dos nós, seguindo regras específicas para preservar a estrutura平衡ada (Szwarcfiter; Markenzon, 2020).

Independentemente da técnica aplicada, a estratégia comum em árvores平衡adas envolve realizar rotações em nós que violam as regras de balanceamento. Isso significa que, para qualquer nó, a diferença entre as alturas de suas subárvores esquerda (E) e direita (D) não deve exceder uma unidade. Esse critério assegura que todas as operações essenciais na árvore, como

busca, inserção e remoção, possam ser realizadas em tempo logarítmico, otimizando a eficiência do processo.

Durante as operações de inserção ou remoção em árvores binárias de busca (BST), as subárvores podem perder seu equilíbrio. O algoritmo das árvores AVL é projetado para assegurar que a árvore permaneça equilibrada, adotando medidas corretivas quando o fator de balanceamento de qualquer nó atinge 2 ou -2, indicando desequilíbrio.

E como podemos manter uma árvore balanceada?

Para realinhar a estrutura e preservar a eficiência das buscas, utilizamos rotações. Essas operações ajustam as ligações entre os nós sem comprometer a ordenação da árvore. Existem **quatro** situações específicas nas árvores AVL que necessitam de rotações para restaurar o balanceamento. A exploração desses cenários requer a compreensão dos conceitos de altura de um nó e do balanceamento.

Altura de um nó

Diferentemente da altura total da árvore, que é a altura do nó raiz, a altura individual de cada nó também é relevante para manter o equilíbrio da árvore como um todo. Portanto, a estrutura de um nó é adaptada para incluir a informação sobre sua própria altura, facilitando o cálculo do balanceamento e a aplicação das rotações necessárias para manter a árvore AVL equilibrada após inserções ou remoções (Takenaka, 2021).

Cálculo do balanceamento de um nó

Com a capacidade de determinar a altura tanto da subárvore esquerda quanto da subárvore direita de um nó, é possível calcular o seu balanceamento. O balanceamento de um nó é obtido pela diferença entre a altura de sua subárvore esquerda e a altura de sua subárvore direita.

$$fb = h_d(u) - h_e(u)$$

Figura 2 | Equação do Fator de Balanceamento. Fonte: adaptada de Takenaka (2021).

Inserção em Árvores AVL

Na inserção de dados em árvores AVL, o procedimento segue o mesmo método das árvores de busca binária (BST). Contudo, após a adição de um novo elemento, é necessário atualizar as alturas de todos os nós afetados e verificar se a árvore mantém as propriedades de uma AVL. Se

a estrutura não estiver conforme os critérios de uma AVL, torna-se essencial realizar rotações específicas para restaurar o balanceamento (Takenaka, 2021).

Vamos detalhar o processo de inserção utilizando a seguinte terminologia:

z: o nó recém-inserido.

y: o nó pai do elemento inserido.

x: o avô do nó inserido, ou seja, o pai de **y**.

Caso 1. Rotação à direita (RR): esta situação ocorre quando o nó é inserido à esquerda de seu pai e este, por sua vez, também se encontra à esquerda de seu próprio pai, configurando uma estrutura que tende para a esquerda.

Na rotação à direita, quando temos uma configuração em que o nó e seu pai estão alinhados à esquerda, a rotação é aplicada para equilibrar a altura da subárvore. Por exemplo, numa configuração onde um nó **v1** possui um fator de平衡amento de +2 e seu filho **v2**, à direita, tem um fator de平衡amento de +1, realiza-se uma rotação simples à esquerda em **v1**, equilibrando assim a altura da subárvore, conforme ilustrado na figura referenciada (Takenaka, 2021).

Esse método assegura que as operações de inserção não comprometem a eficiência das buscas, inserções e remoções subsequentes, mantendo a estrutura de dados eficaz e balanceada.

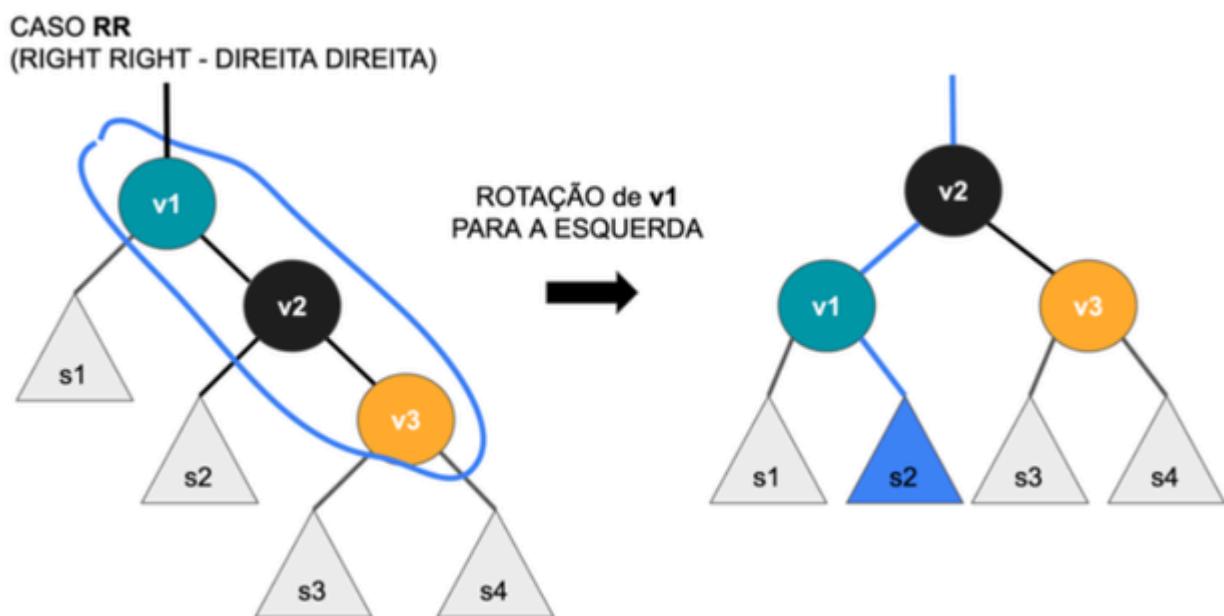


Figura 3 | Caso RR e rotação simples de v1 para a esquerda. Fonte: adaptada de Takenaka (2021).

Caso 2. Rotação à esquerda (LL): quando um novo nó é adicionado à direita de seu pai e este último é posicionado à direita de seu próprio pai, indicando uma tendência à direita da árvore, é necessária uma rotação à esquerda para manter o equilíbrio.

No cenário de rotação LL, onde ambos os filhos de uma subárvore estão situados à esquerda, procede-se com uma rotação simples à direita do vértice superior dessa cadeia, ou seja, **v3**, com o objetivo de diminuir a discrepância de altura nessa subárvore. Esse processo é exemplificado na figura mencionada, onde o fator de balanceamento (fb) de **v3** é de -2 e o de **v2** é de -1 (Takenaka, 2021).

Esse método de rotação à esquerda é empregado para assegurar que a árvore mantenha sua estrutura balanceada e eficiente, facilitando operações futuras de busca, inserção e remoção ao minimizar as diferenças de altura entre as subárvores.

CASO LL
(LEFT LEFT - ESQUERDA ESQUERDA)

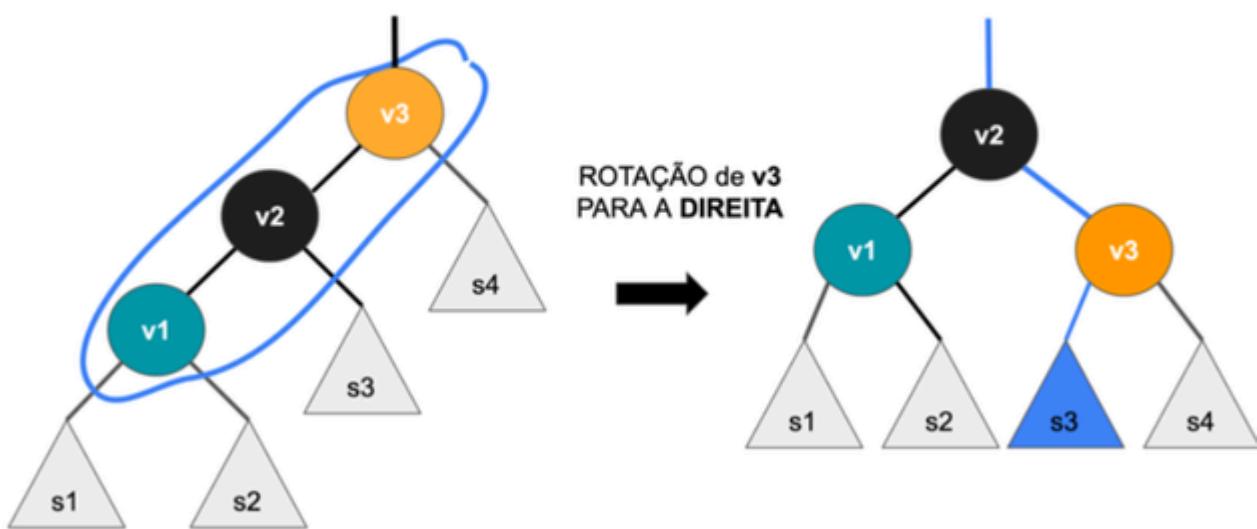


Figura 4 | Caso LL e rotação simples de v3 para a direita. Fonte: adaptada de Takenaka (2021).

Caso 3. Rotação dupla à esquerda e à direita (LR): quando um nó é adicionado à direita do seu pai, que por sua vez é filho à esquerda de seu avô e a árvore inclina-se para um lado, é necessário realizar uma rotação dupla, começando à esquerda e seguindo à direita.

No cenário LR (Esquerda Direita), o filho esquerdo (**v1**) da raiz da subárvore (**v3**) possui um filho à direita (**v2**). Para reequilibrar, inicialmente realizamos uma rotação à esquerda em **v1**, reconfigurando a subárvore para um caso LL. Após essa etapa, procede-se com uma rotação à direita em **v3**, realinhando a subárvore para uma estrutura balanceada. O fator de balanceamento

(fb) de v_3 é -2 e de v_1 é 1, indicando a necessidade dessa rotação composta para restabelecer o equilíbrio (Takenaka, 2021).

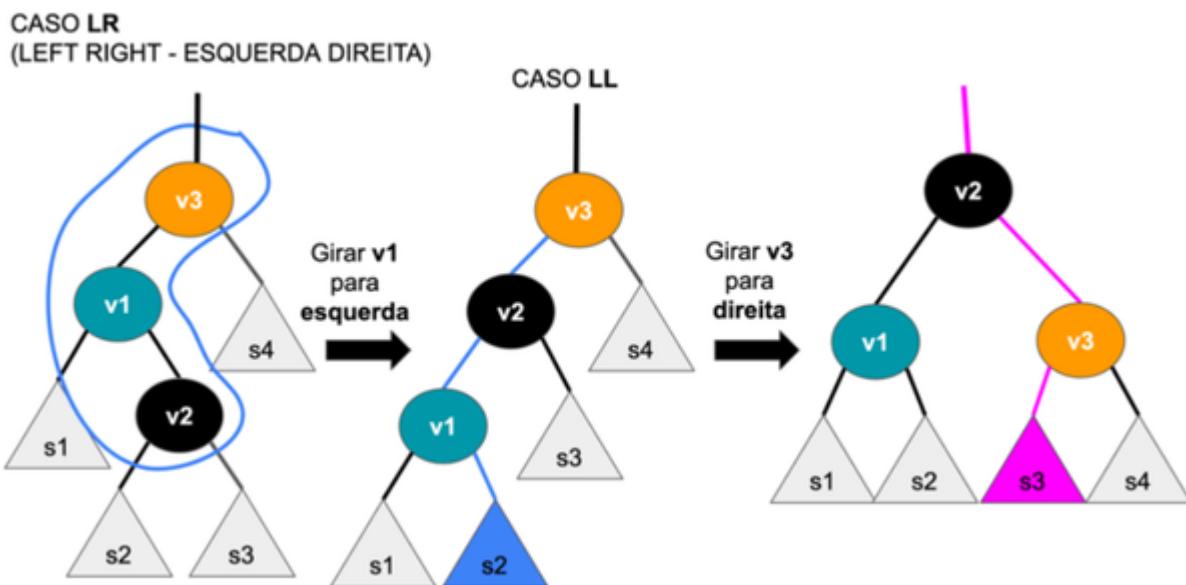


Figura 5 | Caso LR (Left Right) e rotação dupla: de v_1 para esquerda e de v_3 para a direita. Fonte: adaptada de Takenaka (2021).

Caso 4. Rotação dupla à direita e à esquerda (RL): na situação inversa, quando o nó é inserido à esquerda do seu pai, que é filho à direita do seu avô, resultando numa inclinação desbalanceada da árvore, procedemos com uma rotação dupla, iniciando à direita e finalizando à esquerda.

No contexto RL (Direita Esquerda), o filho direito (v_3) da raiz da subárvore (v_1) tem um filho à esquerda (v_2). Para corrigir o desequilíbrio, primeiramente realizamos uma rotação à direita em v_3 , ajustando a subárvore para um caso RR. Em seguida, aplica-se uma rotação à esquerda em v_1 , completando o reequilíbrio da estrutura. O fator de balanceamento de v_1 é 2 e de v_3 é -1, demonstrando a importância dessas rotações sequenciais para manter a estrutura da árvore otimizada para operações eficientes (Takenaka, 2021).

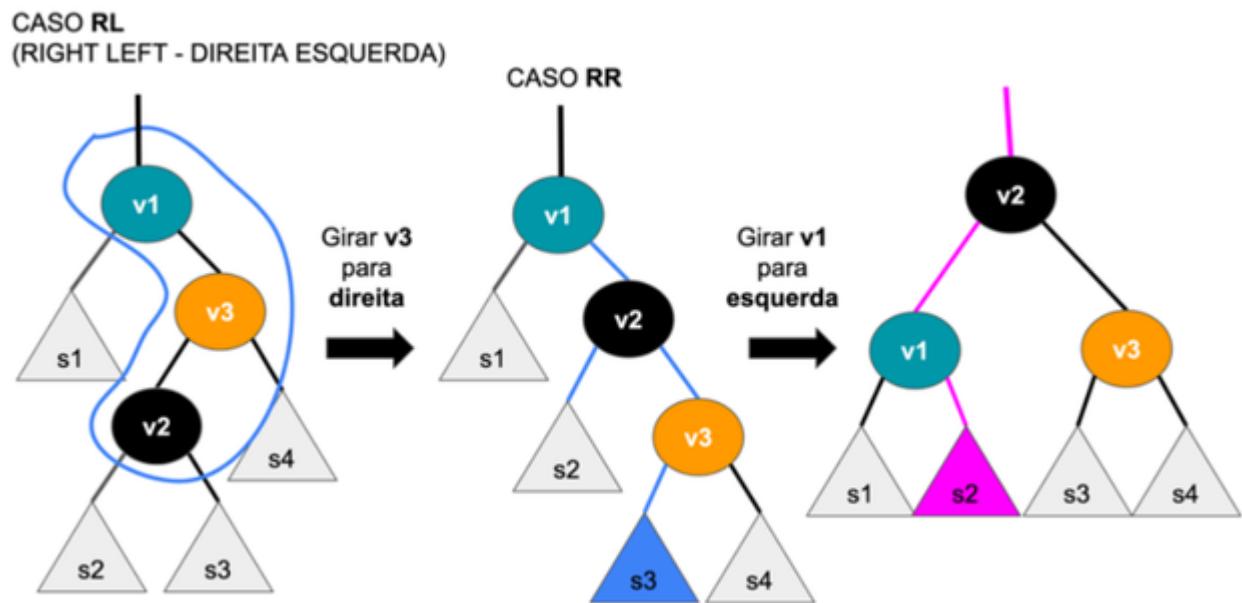


Figura 6 | Caso RL (Right Left) e rotação dupla: de v3 para direita e de v1 para a esquerda. Fonte: adaptada de Takenaka (2021).

Siga em Frente...

Aplicações de árvores AVL

Vamos agora observar o código em Python que traduz o que foi dito até então, seguido da sua explicação passo a passo:

```
class VerticeAVL:
    # Construtor: inicializa um vértice com uma chave, pai e subárvores esquerda e direita vazias.
    def __init__(self, chave, pai=None):
        self.chave = chave # Valor do nó
        self.pai = pai # Referência ao pai
        self.esquerdo = None # Subárvore esquerda
        self.direito = None # Subárvore direita
        self._altura = 0 # Altura do nó, inicialmente 0

    # Métodos __str__ e __repr__ para representar o vértice com sua chave.
    def __str__(self):
        return str(self.chave)

    def __repr__(self):
        return str(self.chave)
```

Método para imprimir a subárvore a partir deste vértice, com recuo visual para indicar a hierarquia.

```
def imprimir_subarvore(self, recuo=0, sentido=):
    if self.direito:
        self.direito.imprimir_subarvore(recuo+10, sentido=)
    print(
        "{}----> [{}](h={},fb={})".format(
            *recuo,
            sentido,
            self.chave,
            self.altura,
            self.fator_de_balanceamento,
        )
    )
    if self.esquerdo:
        self.esquerdo.imprimir_subarvore(recuo+10, sentido=\)
```

Método para inserir uma nova chave na árvore, respeitando as propriedades de árvore AVL.

```
def inserir(self, chave_nova):
    print("{} (atual={})".format(chave_nova, self.chave))
    if chave_nova == self.chave:
        return self # Se a chave já existe, não faz nada.

    raiz_da_subarvore = self
    # Insere na subárvore esquerda ou direita, conforme a comparação das chaves.
    if chave_nova < self.chave:
        if self.esquerdo:
            raiz_da_subarvore = self.esquerdo.inserir(chave_nova)
        else:
            self.esquerdo = VerticeAVL(chave_nova, self)
    elif chave_nova > self.chave:
        if self.direito:
            raiz_da_subarvore = self.direito.inserir(chave_nova)
        else:
            self.direito = VerticeAVL(chave_nova, self)

    # Atualiza a altura e balanceia a árvore após a inserção.
    raiz_da_subarvore.atualizar_altura()
    raiz_da_subarvore = raiz_da_subarvore.balancear()
    return raiz_da_subarvore.pai or raiz_da_subarvore # Retorna a nova raiz da subárvore.
```

Esse trecho define a classe **VerticeAVL**, que representa um nó da árvore AVL, com métodos para inserção, remoção, impressão e balanceamento da árvore. A inserção e remoção são seguidas

de operações de balanceamento para manter as propriedades da árvore AVL, garantindo que todas as operações sejam executadas em tempo logarítmico em relação ao número de nós na árvore (Alves, 2021).

O método **inserir** localiza o ponto de inserção para a nova chave e, se necessário, realiza rotações para manter a árvore balanceada. O método **balancear** verifica o fator de balanceamento de cada nó após a inserção ou remoção e aplica rotações à esquerda ou à direita conforme necessário.

A classe **ArvoreAVL** gerencia a árvore como um todo, mantendo uma referência à raiz e fornecendo métodos para inserção, remoção e impressão da árvore.

Já a remoção e o balanceamento em uma árvore AVL são realizados para manter a propriedade de balanceamento da árvore, onde para cada nó a diferença de altura entre suas subárvores esquerda e direita é no máximo de uma unidade (Alves, 2021). Isso garante que as operações de busca, inserção e remoção sejam eficientes, mantendo a complexidade em $O(\log n)$.

Remoção em Árvores AVL

A remoção de um nó em uma árvore AVL pode ser um dos três casos:

- Remoção de um vértice folha (`_remover_folha`):** se o nó a ser removido não tem filhos, ele é simplesmente removido, e a referência do pai a este nó é atualizada para **None**. O nó pai tem seu balanceamento e altura atualizados posteriormente.
- Remoção de um vértice com um filho (`_remover_pai_de_um_filho`):** se o nó a ser removido tem apenas um filho, o filho substitui a posição do nó removido na árvore, mantendo a estrutura de árvore. O pai do nó removido aponta agora para o filho do nó removido.
- Remoção de um vértice com dois filhos (`_remover_pai_de_dois_filhos`):** quando o nó a ser removido tem dois filhos, a estratégia é encontrar o sucessor do nó, que é o menor valor na subárvore direita. O valor desse sucessor é copiado para o nó a ser removido e, então, o sucessor é removido, o que reduz o problema a um dos dois casos anteriores, pois o sucessor sempre terá, no máximo, um filho.

```
def _remover_folha(self):  
  
    Remove o vértice folha  
    :return: nova raiz  
    ....  
    print('{} vértice folha, filho de {}'.format(  
        self.chave, self.pai.chave))  
    pai = self.pai  
    if self.pai:  
        # tem pai, então não sou a raiz  
        if self.pai.esquerdo is self:  
            # sou filho da esquerda, me desvincula da esquerda
```

ESTRUTURA DE DADOS

```
    self.pai.esquerdo = None
else:
    # sou filho da direita, me desvincula da direita
    self.pai.direito = None
# me desvinculo do meu pai
self.pai = None
return pai

def _remover_pai_de_um_filho(self):
    Remove o vértice que tem um filho direito ou esquerdo
    :return: nova raiz
    """
    print({}, pai de {}.format(
        self.chave, (self.esquerdo or self.direito).chave))
    # identifico meu pai
    meu_pai = self.pai
    # tenho só 1 filho, identifico meu filho (esquerdo ou direito)
    meu_filho = self.esquerdo or self.direito

    if meu_pai is None:
        # sou raiz, a árvore está apontando para mim,
        # não posso ser removido
        # então, vou trocar de lugar com meu filho
        meu_filho.chave, self.chave = self.chave, meu_filho.chave

        # agora estou no lugar do meu filho e posso ser removido
        # a recursividade tratará a forma como serei removido
        return meu_filho.remover(meu_filho.chave)

    # meu pai, é pai do meu filho
    meu_filho.pai = meu_pai
    # meu filho, passa a ser filho do meu pai
    if meu_pai.esquerdo is self:
        # sou filho da direita,
        # meu filho passa a ser seu filho da direita
        meu_pai.esquerdo = meu_filho
    else:
        # sou filho da esquerda,
        # meu filho passa a ser seu filho da esquerda
        meu_pai.direito = meu_filho

    # me desvinculo do meu pai e do meu filhho
    self.pai = None
    self.esquerdo = None
    self.direito = None
```

ESTRUTURA DE DADOS

```
return meu_pai

def _remover_pai_de_dois_filhos(self):
    Remove o vértice que tem 2 filhos
    :return: nova raiz
    """
    print("{} pai de {} e {}".format(
        self.chave, self.esquerdo.chave, self.direito.chave))
    # obter o vértice que tem a chave com menor valor (mais à esquerda)
    # na subárvore do subárvore direita
    esq = self.direito.buscar_vertice_menor_chave_na_subarvore()
    print("{}{}".format(str(esq)))

    # troca valor da chave entre o nó atual e o esq
    print("e {} trocam de vértice".format(self.chave, esq.chave))
    self.chave, esq.chave = esq.chave, self.chave

    # remover o esquerdo / recursividade
    return esq.remover(esq.chave)

def remover(self, chave):
    print("chave atual: {}".format(chave, self.chave))
    if self.chave == chave:
        print("para remover".format(chave))
        # encontrou a chave a ser removida

        if self.esquerdo and self.direito:
            # tem ambos filhos
            raiz_da_subarvore = self._remover_pai_de_dois_filhos()
        elif self.esquerdo or self.direito:
            # tem ou filho esquerdo ou filho direito
            raiz_da_subarvore = self._remover_pai_de_um_filho()
        else:
            # nao tem filhos
            raiz_da_subarvore = self._remover_folha()
        # retorna o pai
        return raiz_da_subarvore

    raiz_da_subarvore = self
    if chave < self.chave:
        # se esquerdo existe, continua a busca pelo esquerdo
        # senão a busca encerra e None é retornado
        raiz_da_subarvore = self.esquerdo and self.esquerdo.remover(chave)
    elif chave > self.chave:
        # se direito existe, continua a busca pelo direito
```

```

# senão a busca encerra e None é retornado
raiz_da_subarvore = self.direito and self.direito.remover(chave)

if raiz_da_subarvore:
    raiz_da_subarvore.atualizar_altura()
    raiz_da_subarvore = raiz_da_subarvore.balancear()
# retorna o pai do raiz_da_subarvore se existir, senão retorna raiz_da_subarvore
return raiz_da_subarvore.pai or raiz_da_subarvore

def buscar_vertice_menor_chave_na_subarvore(self):
    Procura o vértice que tem a menor chave na subárvore,
    ou seja, o vértice que esteja à extrema esquerda na subárvore
    """
    print(
        {}.format(str(self)))
    if self.esquerdo:
        # recursividade
        return self.esquerdo.buscar_vertice_menor_chave_na_subarvore()
    return self

```

Balanceamento

Após a inserção ou remoção de um nó, a árvore pode ficar desbalanceada. A AVL utiliza rotações para rebalancear a árvore:

- **Rotação para a esquerda (_rotacao_para_esquerda):** usada quando um nó tem um desbalanceamento à direita (RR). O nó filho direito do nó desbalanceado torna-se a nova raiz da subárvore, enquanto o nó desbalanceado se torna o filho esquerdo da nova raiz.
- **Rotação para a direita (_rotacao_para_direita):** usada quando um nó tem um desbalanceamento à esquerda (LL). O nó filho esquerdo do nó desbalanceado torna-se a nova raiz da subárvore, enquanto o nó desbalanceado se torna o filho direito da nova raiz.

As rotações duplas são combinações das rotações simples e são usadas nos casos de desbalanceamento esquerda-direita (LR) e direita-esquerda (RL).

```

@property
def altura(self):
    return self._altura

def atualizar_altura(self):
    # pega a maior altura entre as duas subárvores e soma 1
    self._altura = 1 + max([self.altura_esquerda, self.altura_direita])

```

ESTRUTURA DE DADOS

```
@property
def altura_esquerda(self):
    if self.esquerdo:
        return self.esquerdo.altura
    return -1

@property
def altura_direita(self):
    if self.direito:
        return self.direito.altura
    return -1

@property
def fator_de_balanceamento(self):
    return self.altura_direita - self.altura_esquerda

def balancear(self):
    fb = self.fator_de_balanceamento
    print("{}={}.format(self.chave, fb))
    if fb == 2:
        return self._balancear_subarvore_direita()
    if fb == -2:
        return self._balancear_subarvore_esquerda()
    return self

def _balancear_subarvore_direita(self):

    print("{} format(self.chave))
    if self.direito.fator_de_balanceamento == -1:
        # Caso RL: aplicar rotacao do filho direito para a direita
        # para ficar com a configuração do Caso RR
        print("{} à direita + Rotação de {} à esqueda".format(
            str(self.direito), self.chave)
        )
        self.direito._rotacao_para_direita()

    # Caso RR: aplicar rotacao a esquerda
    nova_raiz = self._rotacao_para_esquerda()
    return nova_raiz

def _balancear_subarvore_esquerda(self):

    print("{} format(self.chave))
    if self.esquerdo.fator_de_balanceamento == 1:
        # Caso LR: aplicar rotacao do filho esquerdo para a esquerda
        # para ficar com a configuração do Caso LL
```

ESTRUTURA DE DADOS

```

print("{} à esquerda + Rotação de {} à direita".format(
    str(self.esquerdo), self.chave)
)
self.esquerdo._rotacao_para_esquerda()

# Caso LL: aplicar rotacao direita
nova_raiz = self._rotacao_para_direita()
return nova_raiz

def _rotacao_para_esquerda(self):

    Caso RR (Right Right Case) - rotação única
    Caso LR (Left Right Case) - primeira rotação
        v1          v2
        / \        / \
    s1  v2      v1   v3
    / \  ->  / \  / \
    s2  v3      s1  s2  s3  s4
        / \
    s3  s4
    .....
    self._rotacao_info()

# identifica os vértices envolvidos:
# pai da subarvore, v1 (raiz da subarvore), v2 (nova raiz da subarvore) e s2
raiz_da_subarvore = self
pai_da_subarvore = raiz_da_subarvore.pai
v1 = raiz_da_subarvore
v2 = v1.direito
s2 = v2.esquerdo

# executa a rotação / atualiza os relacionamentos entre os vértices
if pai_da_subarvore:
    if raiz_da_subarvore is pai_da_subarvore.direito:
        # raiz da subarvore é filho direito
        pai_da_subarvore.direito = v2
    else:
        pai_da_subarvore.esquerdo = v2
    v1.pai = v2
    v1.direito = s2
    v2.pai = pai_da_subarvore
    v2.esquerdo = v1
    if s2:
        s2.pai = v1

# atualizar alturas
v1.atualizar_altura()

```

ESTRUTURA DE DADOS

```

v2.atualizar_altura()

self._rotacao_info(v2)

# retorna a nova raiz
return v2

def _rotacao_para_direita(self):

    Caso LL (Left Left Case) - rotação única
    Caso RL (RightLeft Case) - primeira rotação
        |
        |           |
    v3 (self)       v2
    / \           / \
v2 s4      ->  v1   v3
/ \           / \ / \
v1 s3      s1 s2 s3 s4
/ \
s1 s2
"""

self._rotacao_info(sentido=)
# identifica os vértices envolvidos
raiz_da_subarvore = self
v3 = raiz_da_subarvore
pai_da_subarvore = raiz_da_subarvore.pai
v2 = v3.esquerdo
s3 = v2.direito

# Faz a rotação / atualiza os relacionamentos entre os vértices
if pai_da_subarvore:
    if raiz_da_subarvore is pai_da_subarvore.direito:
        pai_da_subarvore.direito = v2
    else:
        pai_da_subarvore.esquerdo = v2
    v2.pai = pai_da_subarvore
    v2.direito = v3
    v3.pai = v2
    v3.esquerdo = s3
    if s3:
        s3.pai = v3

# atualizar alturas
v3.atualizar_altura()
v2.atualizar_altura()

self._rotacao_info(v2)

```

```
# retorna a nova raiz
return v2

def _rotacao_info(self, sentido=, nova_raiz_da_subarv=None):
    print()
    if nova_raiz_da_subarv is None:
        caso = if sentido == else
        print("{}: Rotação de {} à {}".format(caso, sentido, self.chave))
        print()
    else:
        print()

    atual = nova_raiz_da_subarv or self
    (atual.pai or atual).imprimir_subarvore()
    print()
    if nova_raiz_da_subarv:
        print("{} à {}: nova_raiz={} (pai={})".format(
            self.chave, sentido, str(nova_raiz_da_subarv), str(nova_raiz_da_subarv.pai)))
```

Após qualquer modificação na árvore, a altura de cada nó é atualizada e a árvore é percorrida de baixo para cima, aplicando as rotações necessárias para “rebalanceá-la”. Isso garante que a altura da árvore se mantenha logarítmica em relação ao número de nós, otimizando as operações de busca, inserção e remoção.

Vamos Exercitar?

O exercício proposto no ponto de partida lhe pedia para desenvolver um sistema de gestão de biblioteca digital que precisa armazenar e recuperar informações sobre livros rapidamente. A estratégia para implementar essa funcionalidade é implementar as funções básicas de uma Árvore AVL: inserção, remoção e busca de nós, além de garantir o balanceamento da árvore após cada inserção ou remoção. Para simplificar, considere que cada livro no catálogo da biblioteca é identificado por um número único (ID).

A implementação de uma Árvore AVL envolve a compreensão de rotações simples e duplas (à direita e à esquerda) que são aplicadas para manter a árvore balanceada após cada inserção ou remoção.

- Inserção:

A implementação de uma Árvore AVL envolve a compreensão de rotações simples e duplas (à direita e à esquerda) que são aplicadas para manter a árvore balanceada após cada inserção ou remoção.

- **Remoção:**

A remoção de um nó pode ser mais complexa, dependendo se o nó tem filhos. Nós folha podem ser removidos diretamente. Nós com um filho podem ser substituídos por seu filho. Nós com dois filhos são geralmente substituídos pelo seu sucessor (o menor nó da subárvore direita) ou pelo seu predecessor (o maior nó da subárvore esquerda), que é então removido. Após a remoção, assim como na inserção, verifica-se o balanceamento da árvore e aplicam-se rotações se necessário.

- **Busca:**

A busca em uma Árvore AVL segue o mesmo princípio de uma árvore binária de busca, comparando o ID buscado com o dos nós percorridos, movendo-se para a subárvore esquerda ou direita conforme necessário. Como a árvore é balanceada, a busca é eficiente.

```
# Exemplo de código para inserção em uma Árvore AVL (simplificado)
```

```
class No:  
    def __init__(self, key):  
        self.key = key  
        self.height = 1  
        self.left = None  
        self.right = None  
  
class AVLTree:  
    def inserir(self, root, key):  
        # Insere o nó e atualiza a altura  
        # Verifica o balanceamento e realiza rotações se necessário  
  
        def remover(self, root, key):  
            # Remove o nó e realiza rotações se necessário para manter o balanceamento  
  
        def buscar(self, root, key):  
            # Busca um nó na árvore
```

Esta é apenas uma sugestão de como você poderia estruturar sua solução, ficando ao seu critério escolher a melhor forma para tal.

Bons estudos!

Saiba mais

ESTRUTURA DE DADOS

Introdução a árvores AVL:

- BRUNET, J. A. [Árvores Balanceadas \(AVL\)](#). JoaoArthurBm.

Principais conceitos de árvores AVL:

- ÁRVORES AVL. [Desenvolvendo Software](#).

Aplicações de árvores AVL:

- SOUZA, L. Árvores AVL: [Entenda como funcionam essas estruturas de dados](#). Meu Verde Jardim.

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

TAKENAKA, R. M. **Fundamentos de árvores e algoritmos**. Estrutura de Dados, 2021. Disponível em: <http://tinyurl.com/bdfmd7h3>. Acesso em: 28 jan. 2024.

Aula 5

ESTRUTURA DE DADOS ÁRVORES

Videoaula de Encerramento

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

ESTRUTURA DE DADOS

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Nesta videoaula, iremos rever os fundamentos das estruturas de dados em árvores e algoritmos, explorando desde árvores de busca binária até árvores AVL, B e Quadtrees. Iremos recapitular como essas estruturas otimizam buscas, inserções e remoções, tornando-as essenciais para a eficiência em sistemas de gerenciamento de banco de dados e processamento de imagens. Compreender esses conceitos é importante para a sua prática profissional, abrindo portas para soluções mais eficazes e inovadoras no desenvolvimento de software.

Ponto de Chegada

Olá, estudante! Para desenvolver a competência desta Unidade, que é conhecer e ser capaz de implementar estruturas de dados do tipo árvores, você primeiramente conheceu os conceitos fundamentais que formam a base das estruturas de dados em árvores e compreendeu os algoritmos associados.

Logo após, você aprendeu sobre as árvores de busca binária e entendeu como organizar e acessar dados de maneira eficiente (Alves, 2021).

Com as árvores B e Quadtrees, você expandiu essa compreensão para sistemas que lidam com grandes volumes de dados e necessidades de indexação espacial, respectivamente (Takenaka, 2021).

Por fim, as árvores AVL introduziram o conceito de balanceamento, fundamental para manter a eficiência das operações à medida que os dados são inseridos ou removidos (Lambert, 2022).

Ao relacionar esses conteúdos, você desenvolveu a habilidade de escolher a estrutura de árvore mais adequada para cada situação, garantindo a otimização das operações de busca, inserção e remoção. Além disso, você aprendeu a implementar essas estruturas de maneira eficaz, preparando-se para enfrentar desafios reais no desenvolvimento de software.

É Hora de Praticar!



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

É hora de praticar! A seguir, iremos propor um estudo de caso para que você assimile todo o conteúdo aprendido nesta unidade.

Gerenciamento de contatos com árvore AVL

Seu objetivo é criar um sistema em Python que utilize uma árvore AVL para organizar uma lista de contatos, permitindo operações eficientes de adição, remoção e busca de contatos por nome. Especificações:

1. Estrutura do contato: cada contato deve ser um objeto contendo **id** (íntero único), **nome** (*string*) e **telefone** (*string*). O **id** servirá como chave para a organização na árvore AVL.
2. Adição de contatos: o sistema deve suportar a adição de novos contatos, mantendo a árvore AVL balanceada após cada inserção.
3. Remoção de contatos: deve ser possível remover contatos usando o **id** como referência, assegurando o balanceamento da árvore após cada remoção.
4. Busca de contatos: o sistema deve permitir a busca de contatos por nome, retornando todas as correspondências encontradas.

Boa prática!

Para assimilar todo o conteúdo visto, deixamos algumas perguntas para reflexão:

1. Como a escolha entre uma árvore de busca binária e uma árvore AVL pode impactar o desempenho de um sistema de gerenciamento de banco de dados?
2. De que maneira as árvores B e Quadtrees são especialmente úteis para aplicações que envolvem o processamento de grandes volumes de dados e dados espaciais?
3. Qual a importância do balanceamento em árvores AVL e como isso influencia a eficiência das operações de busca, inserção e remoção?

Bons estudos!

Este exercício prático foca na competência de implementação de estruturas de dados tipo árvores, utilizando uma árvore AVL para gerenciar uma lista de contatos.

Observe a seguir uma sugestão de estruturação da solução:

```
class Contato:  
    def __init__(self, id, nome, telefone):
```

ESTRUTURA DE DADOS

```
self.id = id
self.nome = nome
self.telefone = telefone

def __str__(self):
    return f'{self.id}, Nome: {self.nome}, Telefone: {self.telefone}'

class NodoAVL:
    def __init__(self, contato):
        self.contato = contato
        self.altura = 1
        self.esquerdo = None
        self.direito = None

# Métodos para inserção, remoção, busca, cálculo de altura e balanceamento aqui.

class AgendaAVL:
    def __init__(self):
        self.raiz = None

    # Método para adicionar contatos
    def adicionar(self, contato):
        self.raiz = self._adicionar(self.raiz, contato)

    def _adicionar(self, nodo, contato):
        if not nodo:
            return NodoAVL(contato)
        elif contato.id < nodo.contato.id:
            nodo.esquerdo = self._adicionar(nodo.esquerdo, contato)
        else:
            nodo.direito = self._adicionar(nodo.direito, contato)

        # Atualizar altura, verificar平衡amento e aplicar rotações se necessário.
        return nodo

    # Métodos para remoção e busca seriam implementados de forma similar, focando no
    # balanceamento da árvore.
```

A classe **Contato** define a estrutura básica de cada contato, enquanto a classe **NodoAVL** representa cada nó da árvore AVL, contendo um contato e informações de balanceamento. A classe **AgendaAVL** encapsula a árvore AVL, fornecendo métodos para adicionar, remover e buscar contatos.

A adição e remoção de contatos devem manter a árvore balanceada, conforme as regras das árvores AVL, assegurando eficiência nas operações de busca. O balanceamento é mantido por meio de rotações, necessárias para garantir que a diferença de altura entre subárvores de qualquer nó não excede 1.

ESTRUTURA DE DADOS

Este exercício destaca a aplicabilidade das árvores AVL em sistemas que demandam operações rápidas de inserção, remoção e busca em conjuntos de dados que variam dinamicamente, como uma agenda de contatos.

Explore nosso infográfico sobre árvores, um guia visual para entender as diferenças, os usos e as vantagens das árvores binárias, AVL, B e Quadtrees no gerenciamento eficaz de dados.

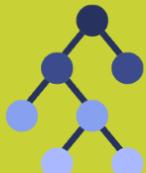
ESTRUTURA DE DADOS

Árvores



FUNDAMENTOS DE ÁRVORES E ALGORITMOS

Estruturas hierárquicas fundamentais para organizar dados de maneira eficiente e otimizar algoritmos de busca, inserção e remoção.



ÁRVORES DE BUSCA BINÁRIA

Estrutura de dados que organiza elementos de forma hierárquica, facilitando operações de busca binária rápida.



ÁRVORES B, QUADTREES E AVL

Árvores B gerenciam grandes blocos de dados com eficiência, enquanto Quadtrees modelam espaços bidimensionais para otimização espacial.

AVL: Árvores auto-balanceadas que garantem operações eficientes ajustando alturas para manter o equilíbrio após cada inserção ou remoção.

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.
LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

TAKENAKA, R. M. **Fundamentos de árvores e algoritmos**. Estrutura de Dados, 2021. Disponível em: <http://tinyurl.com/bdfmd7h3>. Acesso em: 28 jan. 2024.

Unidade 3

GRAFOS E SUAS OPERAÇÕES

Aula 1

Introdução a Grafos

Introdução a Grafos

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Descubra os segredos dos grafos nesta aula: aprenda sobre caminhos e ciclos, fundamentos essenciais de representação e explore grafos com custos, topológicos, direcionados e não direcionados. Entenda como esses conceitos são essenciais para a sua prática profissional, seja na área de Ciência da Computação, Engenharia, Logística ou Análise de Redes. Não perca essa oportunidade de aprimorar seus conhecimentos!

Ponto de Partida

Olá, estudante!

No decorrer de nossa vida cotidiana, frequentemente nos deparamos com a aplicação prática dos grafos sem ao menos notarmos. Essa interação ocorre em rotinas simples, como o trajeto matinal de ir ao trabalho ou à escola, ou mesmo dentro do ambiente doméstico. Considere, por exemplo, a sequência de atividades matinais: despertar, transitar entre o quarto, banheiro,

cozinha e retornar ao quarto. Essa rotina esboça um grafo, onde cada cômodo representa um vértice e os trajetos percorridos entre eles, as arestas. Diferentemente das árvores, que são uma categoria específica de grafo sem ciclos e com arestas que se ramificam, os grafos podem incluir ciclos.

Além disso, no contexto atual, é raro encontrar alguém que não utilize um telefone celular ou mantenha uma lista de contatos. Essas conexões, estendidas pelas redes sociais, criam uma teia de interações que podem ser mapeadas utilizando-se a teoria dos grafos, onde postagens, curtidas e imagens interligadas são representadas através dessa estrutura.

Embora possa parecer que os grafos são uma invenção moderna, sua origem remonta a 1736, muito antes da era digital. Grafos têm sido aplicados na resolução de problemas variados ao longo da história, inclusive no desenho das redes elétricas que conectam nossas residências ao fornecimento de energia, utilizando torres de transmissão, transformadores e subestações como vértices, e os cabos de energia como arestas. Mesmo antes da eletrificação, a disposição das residências, locais públicos, estradas e pontes já podiam ser modeladas por grafos, evidenciando sua utilidade no planejamento urbano e estrutural desde tempos antigos. Com a expansão urbana, a teoria dos grafos tornou-se ainda mais relevante, especialmente em desafios de planejamento como o direcionamento do tráfego urbano.

Para contextualizar ainda mais a aplicabilidade dos grafos, imagine que você atua em uma empresa desenvolvedora de jogos, buscando criar uma experiência prolongada e cativante para os usuários. Um projeto de jogo baseado em labirintos, por exemplo, ilustra perfeitamente a utilidade dos grafos, permitindo múltiplas trajetórias e escolhas ao jogador, sem limitações de repetição de visitas ou direções de percurso.

Portanto, está preparado para explorar a resolução de problemas complexos através da teoria dos grafos?

Bons estudos!

Vamos Começar!

Fundamentos de grafos e sua representação

Um grafo consiste em uma estrutura de dados compreendida por vértices (também chamados de nós) e arestas (conhecidas como arcos). As arestas estabelecem conexões entre pares de vértices, representando relações, ligações ou interações entre eles, ou até mesmo a inexistência dessas relações (Borin, 2020).

Conforme suas propriedades, os grafos podem ser classificados em diferentes tipos. Entre eles, destacam-se os grafos não orientados, caracterizados pela ausência de direção nas conexões entre os vértices. Isso significa que as relações são bidirecionais e simétricas. Para ilustrar, considere um torneio de xadrez com seis competidores, onde cada um enfrentará os demais

participantes. A forma de representar essa situação em um grafo é por meio de um grafo não orientado. Nesse caso, a relação entre os competidores A e B é recíproca, ou seja, não importa se dizemos que A enfrenta B ou que B enfrenta A. Em resumo, os grafos não orientados também são conhecidos por outros termos, como não direcionados ou não dirigidos, e são úteis para modelar situações em que a direção das relações entre os elementos não é relevante ou especificada (Takenaka, 2021).

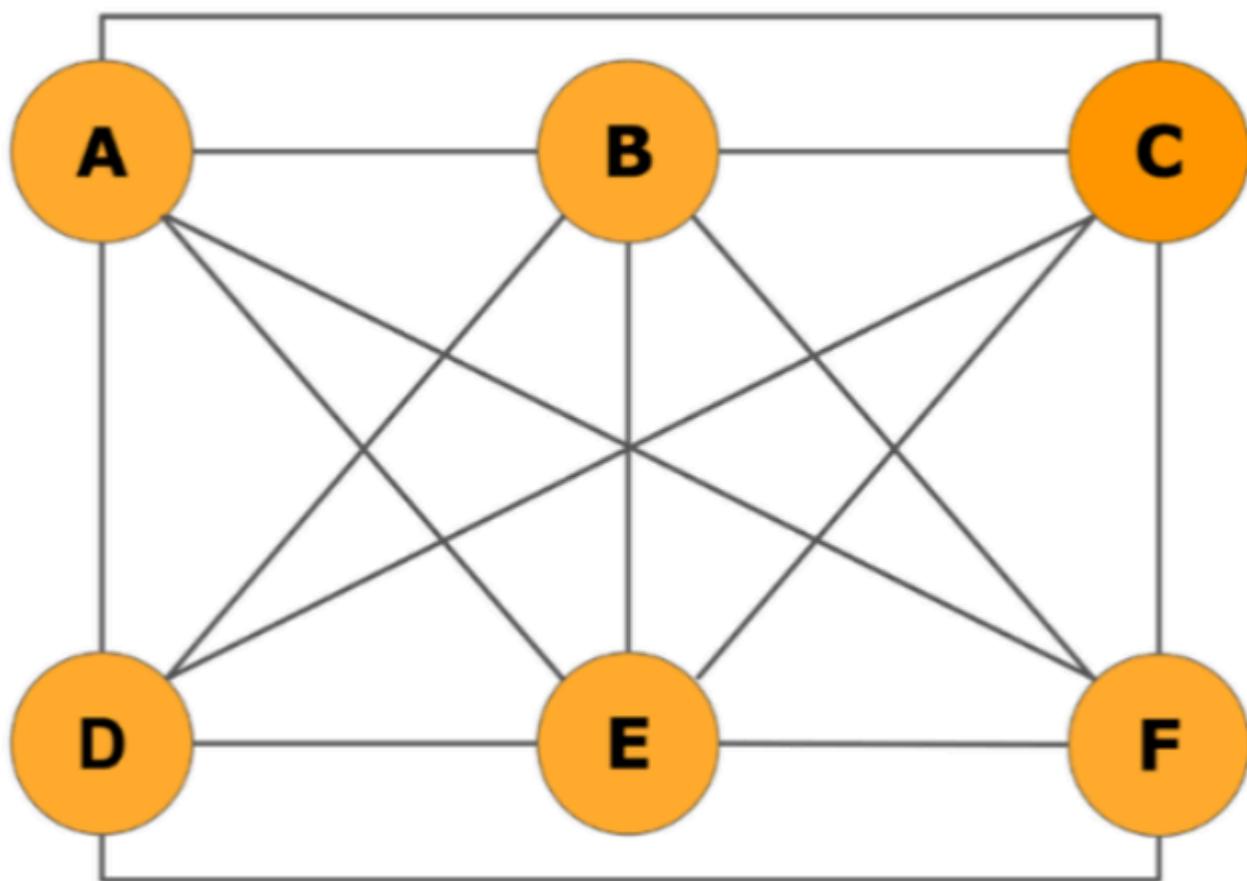


Figura 1 | Grafo não direcionado (ou não dirigido, não orientado). Fonte: adaptada de Takenaka (2021).

Um grafo não direcionado desconexo é utilizado para descrever cenários em que alguns elementos do grafo não estão interligados entre si. Isso ocorre, por exemplo, em redes sociais onde existem usuários que ainda não interagiram uns com os outros, estando apenas presentes na rede como entidades isoladas. Nessa situação, esses usuários são representados por vértices que não compartilham arestas, como é o caso dos vértices A, X e Z ilustrados. Um conjunto de usuários que se conhecem, simbolizado por {G, H}, pode ter ligações entre si, mas não possuem conexões com outros vértices do grafo (Takenaka, 2021).

Esse modelo de grafo é particularmente útil na análise de redes sociais para identificar indivíduos que possam ter maior influência dentro da rede. Usuários que estão conectados a um maior número de pessoas tendem a ter uma influência maior, pois suas ações e interações têm o

potencial de alcançar uma audiência mais ampla. Assim, ao examinar a estrutura desconexa do grafo, é possível identificar tanto os usuários isolados quanto aqueles que, pela sua conectividade, podem desempenhar um papel mais significativo na disseminação de informações ou na formação de comunidades dentro da rede.

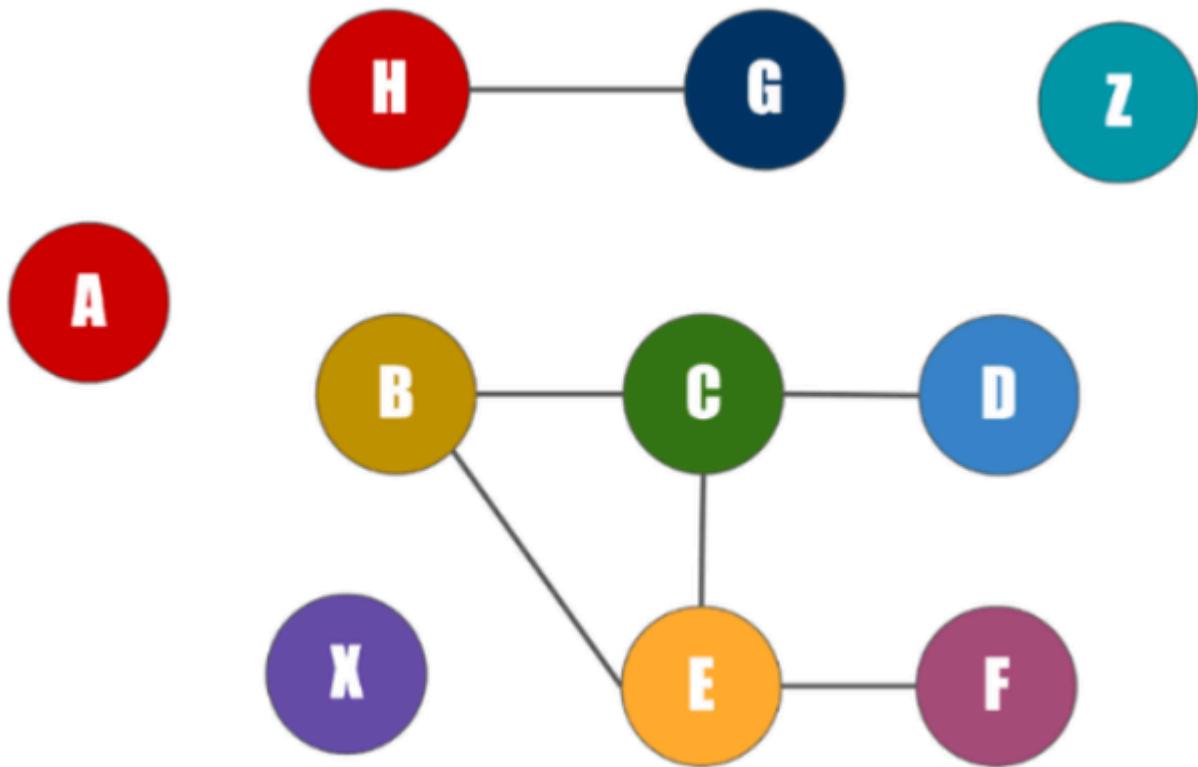


Figura 2 . Fonte: adaptada de Takenaka (2021).

Um grafo orientado, também conhecido como grafo dirigido, direcionado ou dígrafo, caracteriza-se por apresentar relações assimétricas entre seus vértices. Isso significa que a direção das conexões entre os vértices é especificada, tornando a relação entre eles não simétrica. Um exemplo prático dessa aplicação é o mapeamento da direção do tráfego em vias públicas.

Para ilustrar, considere o seguinte cenário: você está indo de carro para uma palestra e se comprometeu a dar carona a alguns colegas. Para organizar a viagem, você anota os endereços e planeja a rota, levando em consideração o sentido das ruas. Saindo do ponto A, você passa pelos pontos B, C e D para buscar cada pessoa, chegando finalmente ao evento no ponto E. Após o evento, decidem ir a uma festa no ponto D, buscando outra pessoa no ponto F no caminho. Após a festa, você deixa todos no ponto D e retorna para casa. Nesse exemplo, as conexões entre os pontos são representadas por setas, indicando a direção da viagem (Takenaka, 2021).

Em um contexto de logística empresarial, essas arestas direcionadas podem incluir custos associados a cada trecho do percurso. A soma desses custos pode fornecer o custo total da rota planejada, permitindo uma análise mais detalhada dos gastos envolvidos no transporte. Essa abordagem, que considera os custos associados às arestas de um grafo orientado, será explorada mais adiante no curso (Szwarcfiter; Markenzon, 2020).

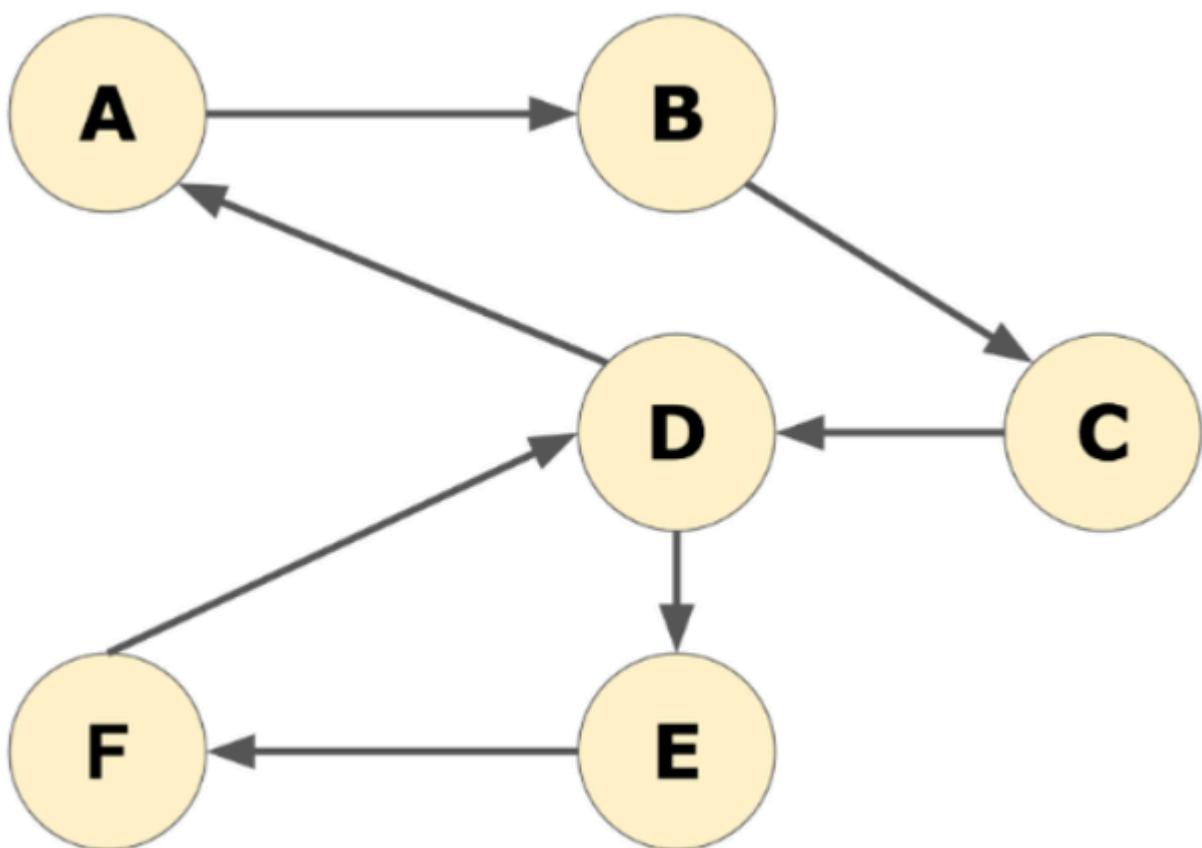


Figura 3 | Dígrafo ou grafo orientado ou grafo dirigido ou grafo direcionado. Fonte: adaptada de Takenaka (2021).

Outra forma de representar um grafo é a representação algébrica. Formalmente, grafo é um conjunto de vértices e um conjunto de arestas. Sendo V (vertices) o conjunto de vértices e $edges$ o conjunto de pares de arestas.

A representação algébrica para o grafo da Figura 3 é:

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

$$E = \{\{A, B\}, \{B, C\}, \{C, D\}, \{D, E\}, \{E, F\}, \{F, D\}, \{D, A\}\}$$

$$G = (V(G), E(G))$$

Uma alternativa para descrever um grafo é por meio de uma abordagem algébrica. De maneira formal, um grafo consiste em dois conjuntos: um conjunto de vértices e um conjunto de arestas. Denominamos V para representar o conjunto de vértices e E para designar o conjunto que contém os pares de arestas.

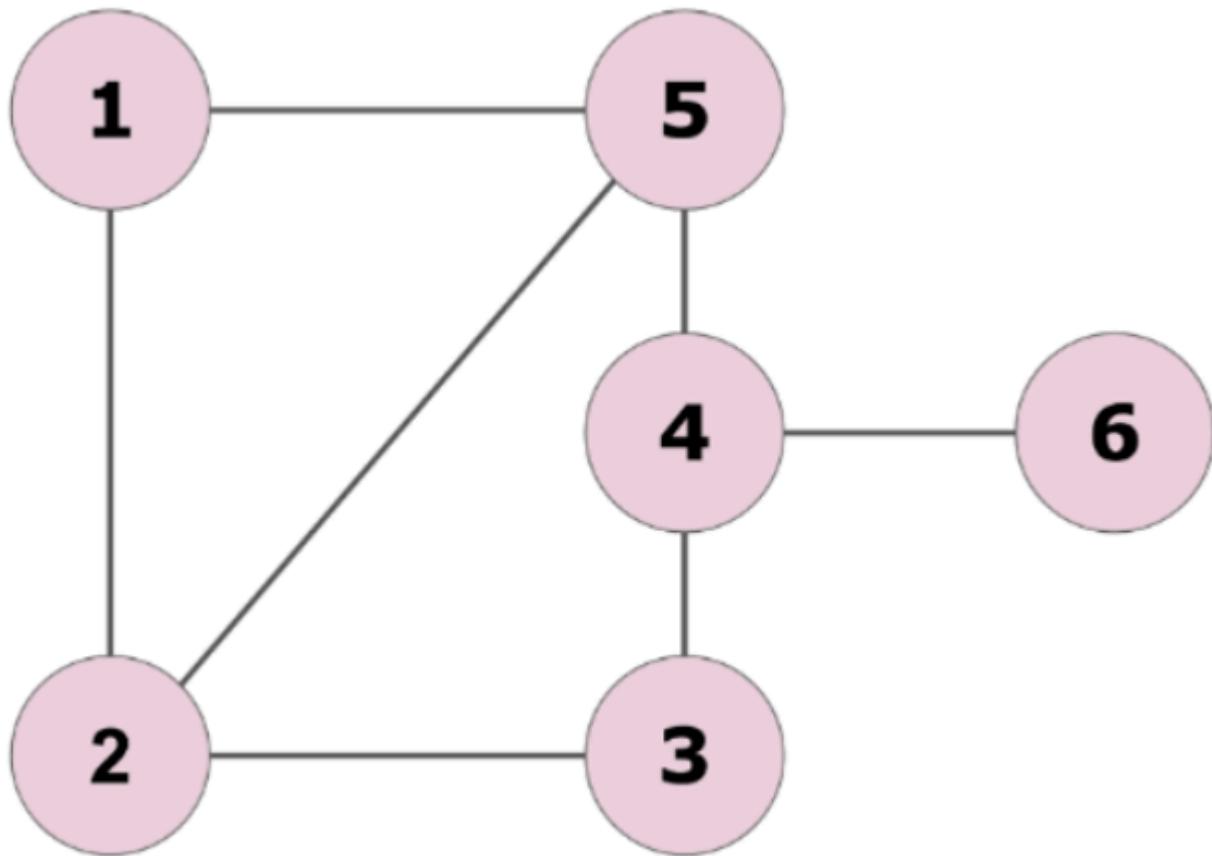


Figura 4 | Grafo não direcionado. Fonte: adaptada de Takenaka (2021).

Siga em Frente...

Caminhos e ciclos em grafos

Até agora, aprendemos sobre duas maneiras de representar grafos: visualmente e numericamente. Uma das representações numéricas é a matriz de adjacência, que é uma representação computacional compreensível pelas linguagens de programação. Na matriz, os vértices são listados nas linhas e nas colunas. Cada célula da matriz indica quantas arestas conectam os vértices correspondentes à linha e à coluna. Os valores na diagonal principal da

matriz representam os *loops*, ou seja, arestas que conectam um vértice a si mesmo. Como os grafos não direcionados têm simetria, podemos economizar memória considerando apenas metade da matriz. Além disso, podemos economizar ainda mais ao ignorar as células que representam *loops*, pois elas sempre serão zeros (Alves, 2021).

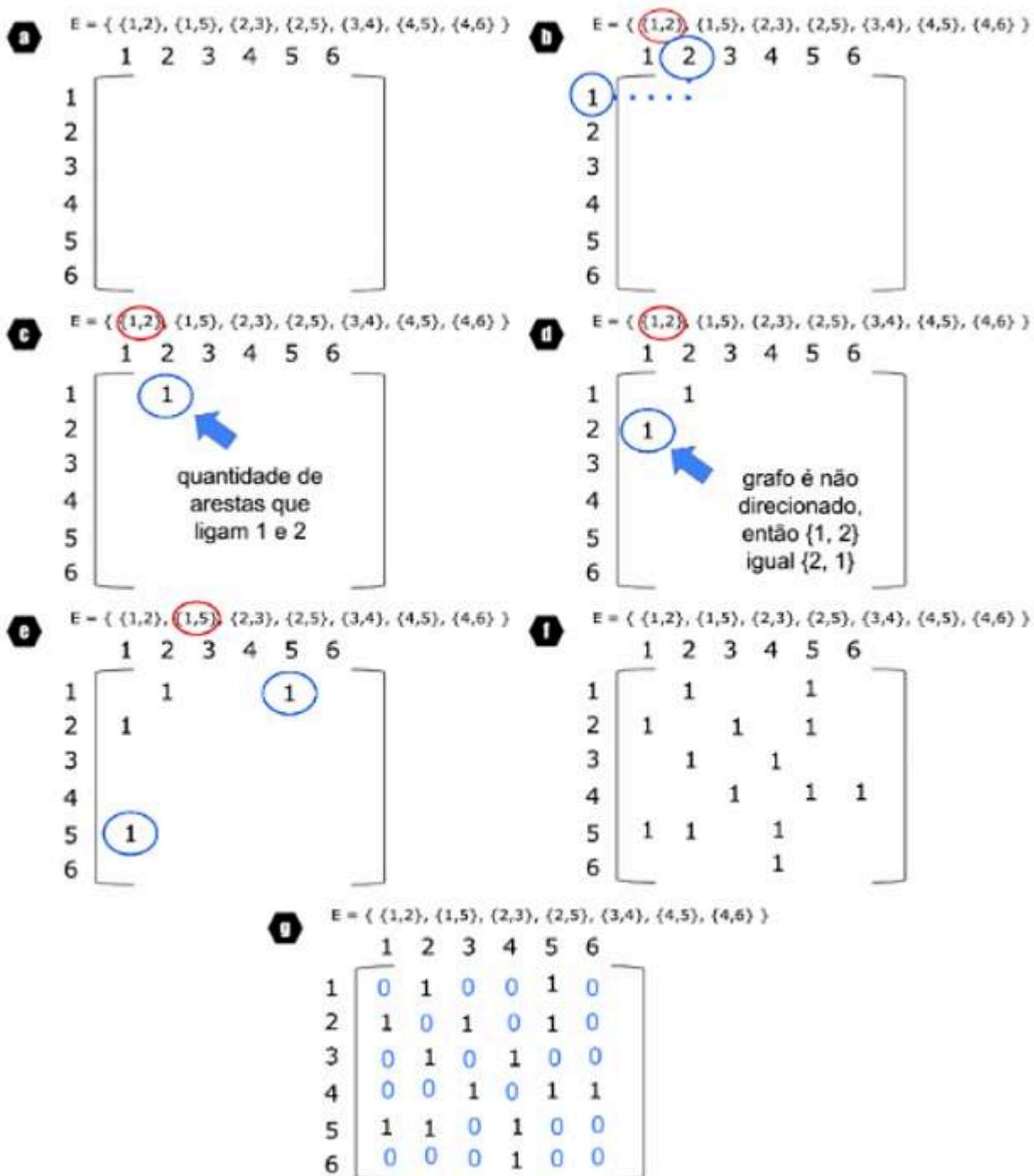


Figura 5 | Preenchimento de matriz de adjacência de grafo não direcionado. Fonte: adaptada de Takenaka (2021).

Na matriz de adjacência, os vértices são listados nas colunas e nas linhas e o número em cada célula representa a quantidade de arestas que conectam os vértices correspondentes à linha e à coluna. A representação gráfica, a algébrica e a matriz de adjacência são formas de representar grafos, sendo a matriz de adjacência uma representação computacional. Outra forma de representação computacional é a lista de adjacência. Ambas podem visualmente destacar características do grafo, como densidade e presença de *loops*. Na lista de adjacência, para cada vértice, criamos uma lista encadeada dos vértices aos quais está conectado (Alves, 2021).

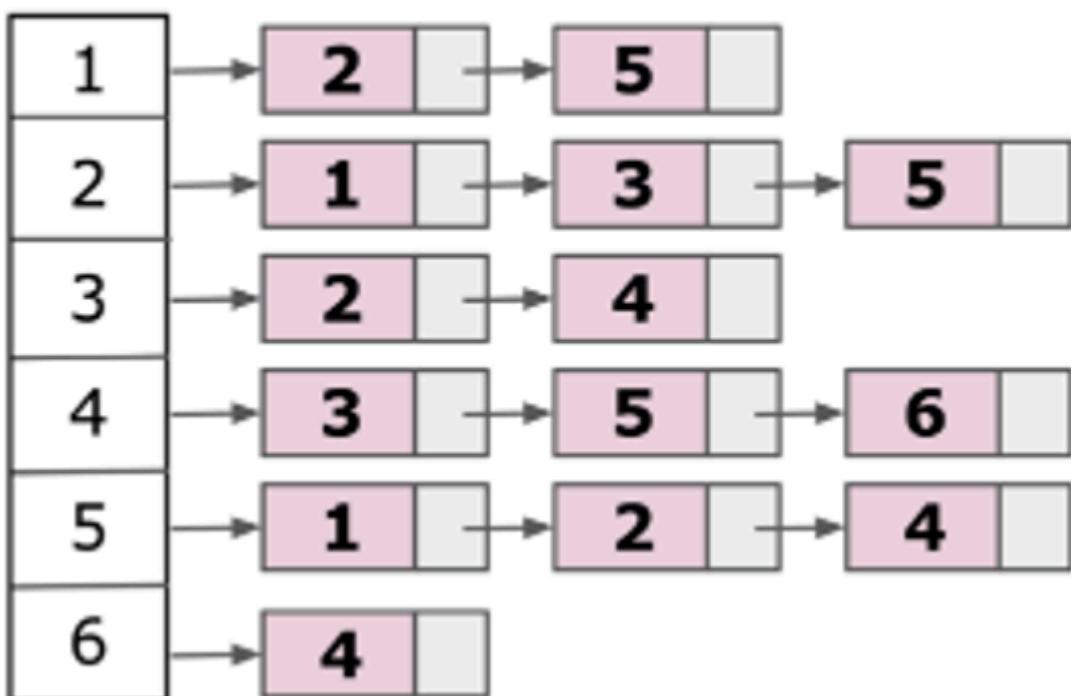


Figura 6 | Lista de adjacência do grafo. Fonte: adaptada de Takenaka (2021).

A teoria dos grafos surgiu com Leonhard Euler, que demonstrou a impossibilidade de atravessar as sete pontes de Königsberg apenas uma vez. Problemas semelhantes são resolvidos utilizando os conceitos de grafos, como encontrar a melhor rota em termos de tempo, quilometragem ou custo. Por exemplo, podemos questionar se é possível desenhar uma casinha, como na figura a seguir, sem tirar o lápis do papel e percorrendo cada aresta apenas uma vez (Takenaka, 2021).

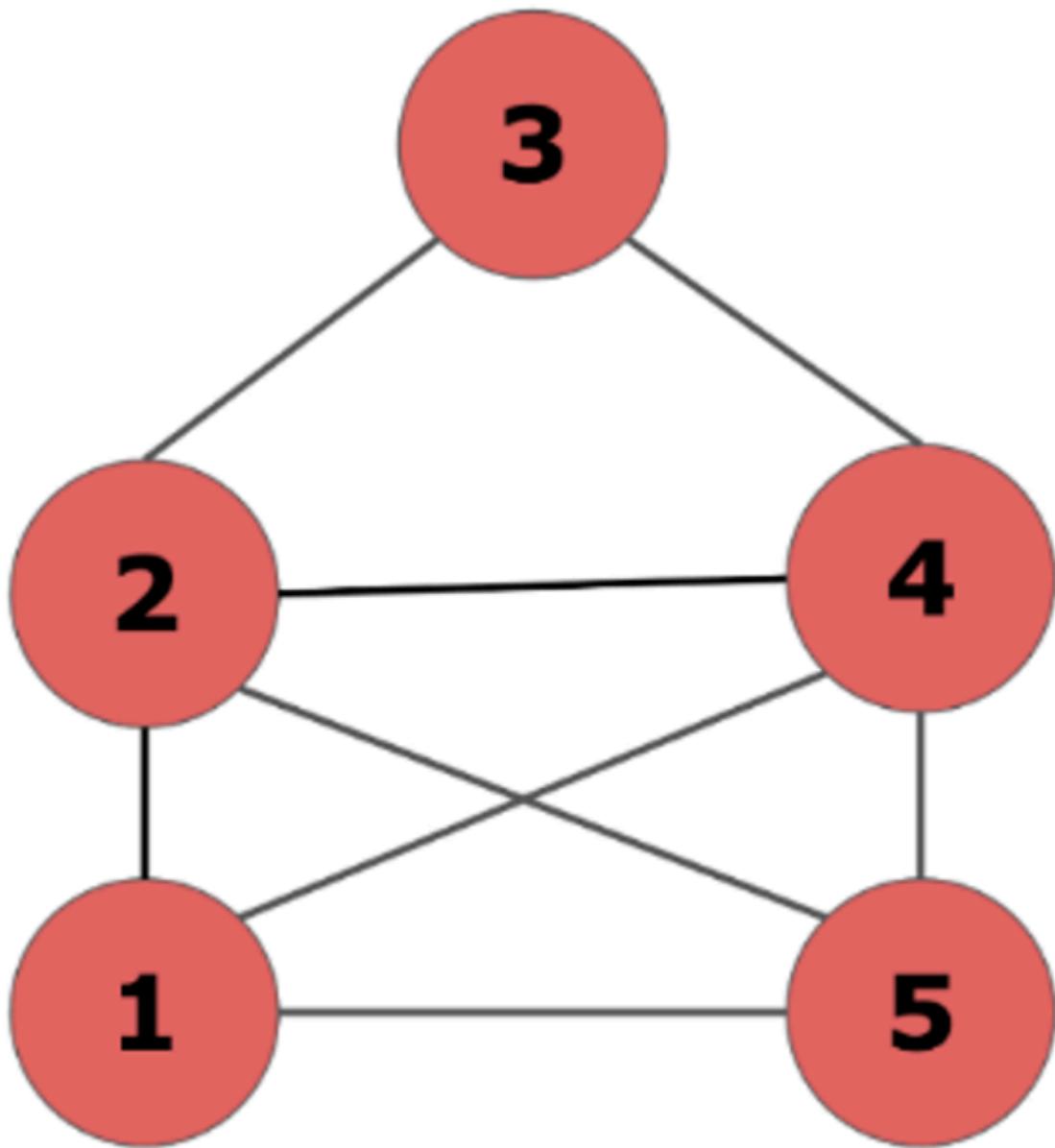


Figura 7 | Grafo em formato de casinha. Fonte: adaptada de Takenaka (2021).

Quando desenhamos a casinha, nosso objetivo é criar uma trilha fechada, percorrendo cada aresta apenas uma vez e retornando ao ponto inicial. Podemos determinar antecipadamente se isso é possível em um grafo através da verificação se ele é euleriano ou semieuleriano. Um grafo é euleriano se todos os vértices têm grau par, enquanto um grafo é semieuleriano se possui exatamente dois vértices de grau ímpar. No caso de grafos não direcionados, o grau de um vértice é o número de arestas que se conectam a ele. Já em grafos direcionados, o grau é a soma do número de arestas que chegam e saem do vértice.

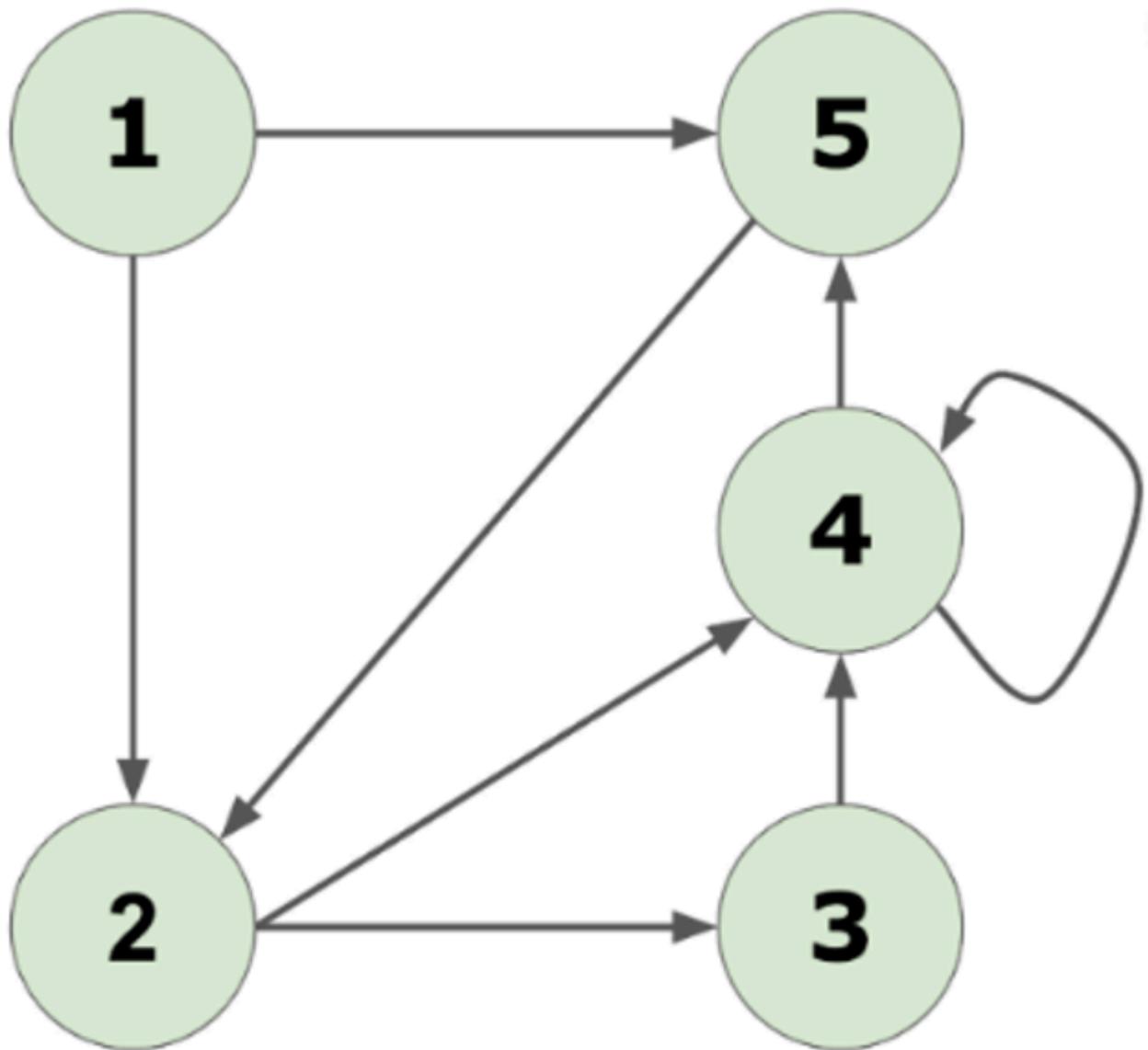


Figura 8 | Grafo orientado com laço (self-loop). Fonte: adaptada de Takenaka (2021).

Para atravessar um vértice sem repetir arestas, é necessário entrar por uma aresta e sair por outra, o que significa que o vértice deve ter um número par de arestas. No caso da casinha, é um grafo semieuleriano, com os vértices 1 e 5 tendo graus ímpares, e os demais vértices com grau par. A trilha da casinha deve começar ou terminar nos vértices 1 ou 5 para evitar repetir arestas. Um ciclo hamiltoniano é uma trilha que não repete os vértices, exceto o início e o fim, passando por todos os vértices, enquanto um grafo hamiltoniano é aquele que permite um ciclo hamiltoniano. Quando a preocupação é com o custo mínimo para percorrer um caminho, as arestas devem ter informações de custo ou peso.

Grafos com custos, topológicos, direcionados e não direcionados

Grafos que possuem informações de custo nas arestas são chamados de grafos ponderados, grafos com custo ou grafos valorados, podendo ser direcionados ou não. Na ilustração de um grafo direcionado ponderado (Figura 9), os pesos podem ser positivos ou negativos, representando ganhos ou gastos. Por exemplo, em uma viagem do ponto A ao ponto E, passando pelos pontos B, C e D, o viajante pode economizar ou gastar dinheiro ao oferecer ou pagar carona aos passageiros (Lambert, 2022).

A representação do grafo direcionado e ponderado como matriz de adjacência e como lista de adjacência substitui os "1s" da relação entre vértices pelos valores de custo ou peso associados às arestas.

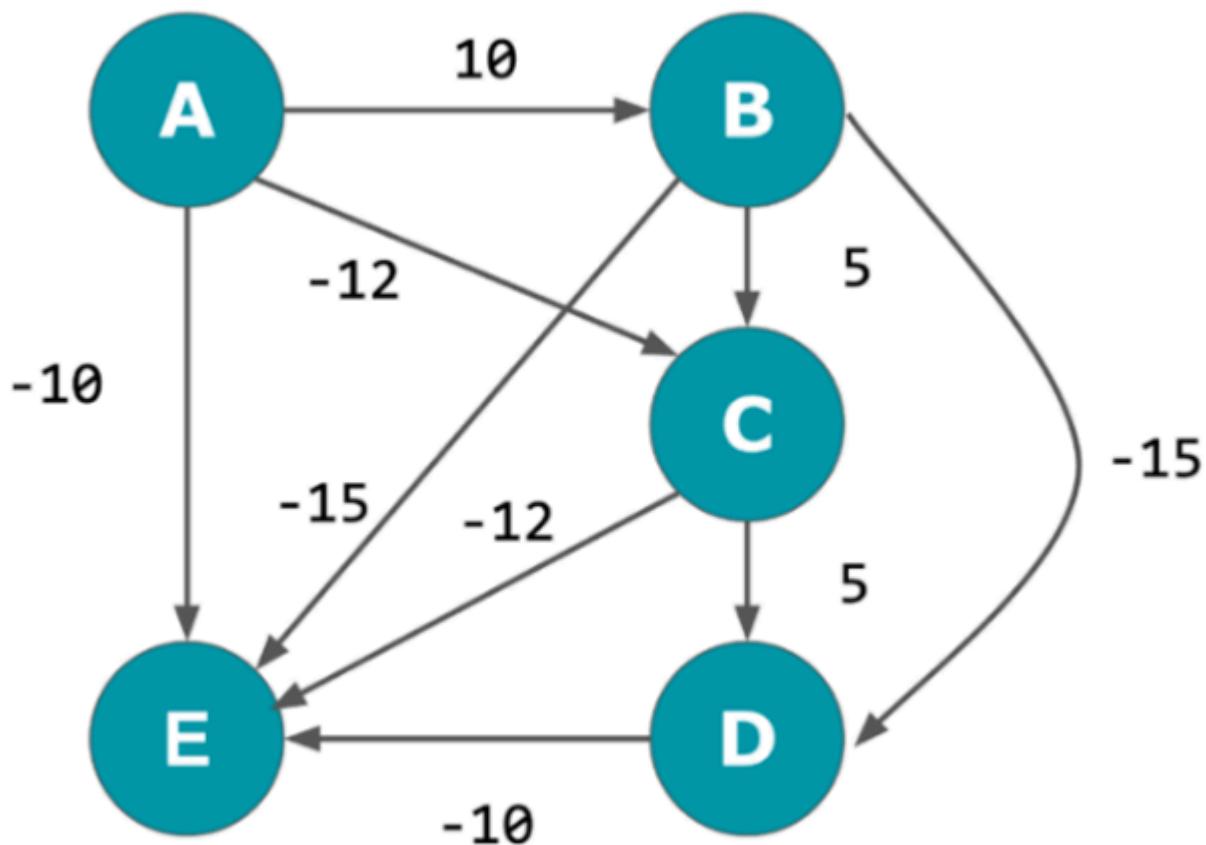


Figura 9 | Grafo direcionado ponderado. Fonte: adaptada de Takenaka (2021).

	A	B	C	D	E
A	0	10	-12	0	10
B	0	0	5	-15	-15
C	0	0	0	5	-12
D	0	0	0	0	-10
E	0	0	0	0	0

Figura 10 | Matriz de adjacência de grafo orientado ponderado. Fonte: adaptada de Takenaka (2021).

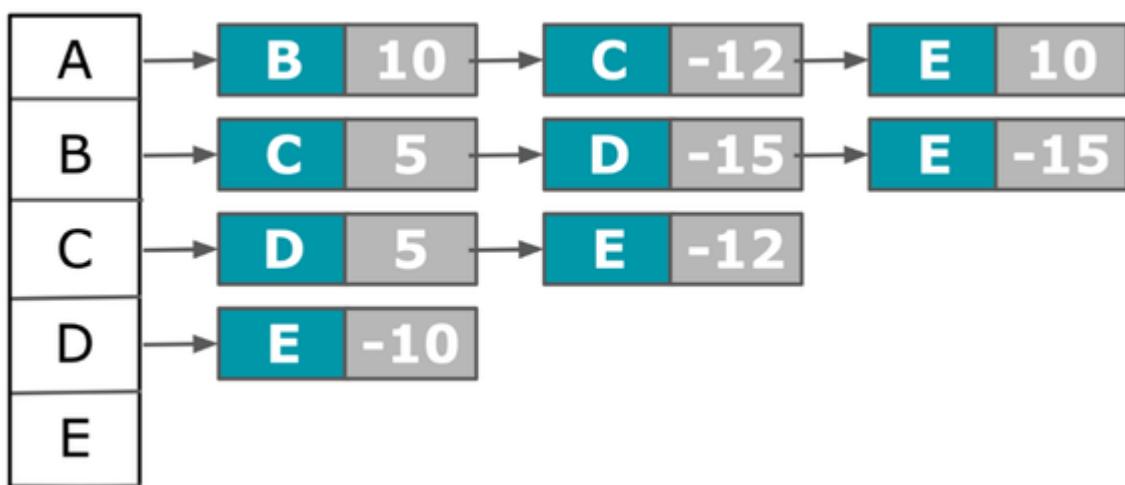


Figura 11 | Lista de adjacência de grafo orientado e ponderado. Fonte: adaptada de Takenaka (2021).

O grafo a seguir é um grafo topológico, que é um grafo acíclico usado para modelar sequências de tarefas e suas dependências. Na ilustração, a tarefa A deve ser concluída antes da B, enquanto D é a última. As tarefas A e E podem ser iniciadas independentemente, mas X e F dependem do término de E.

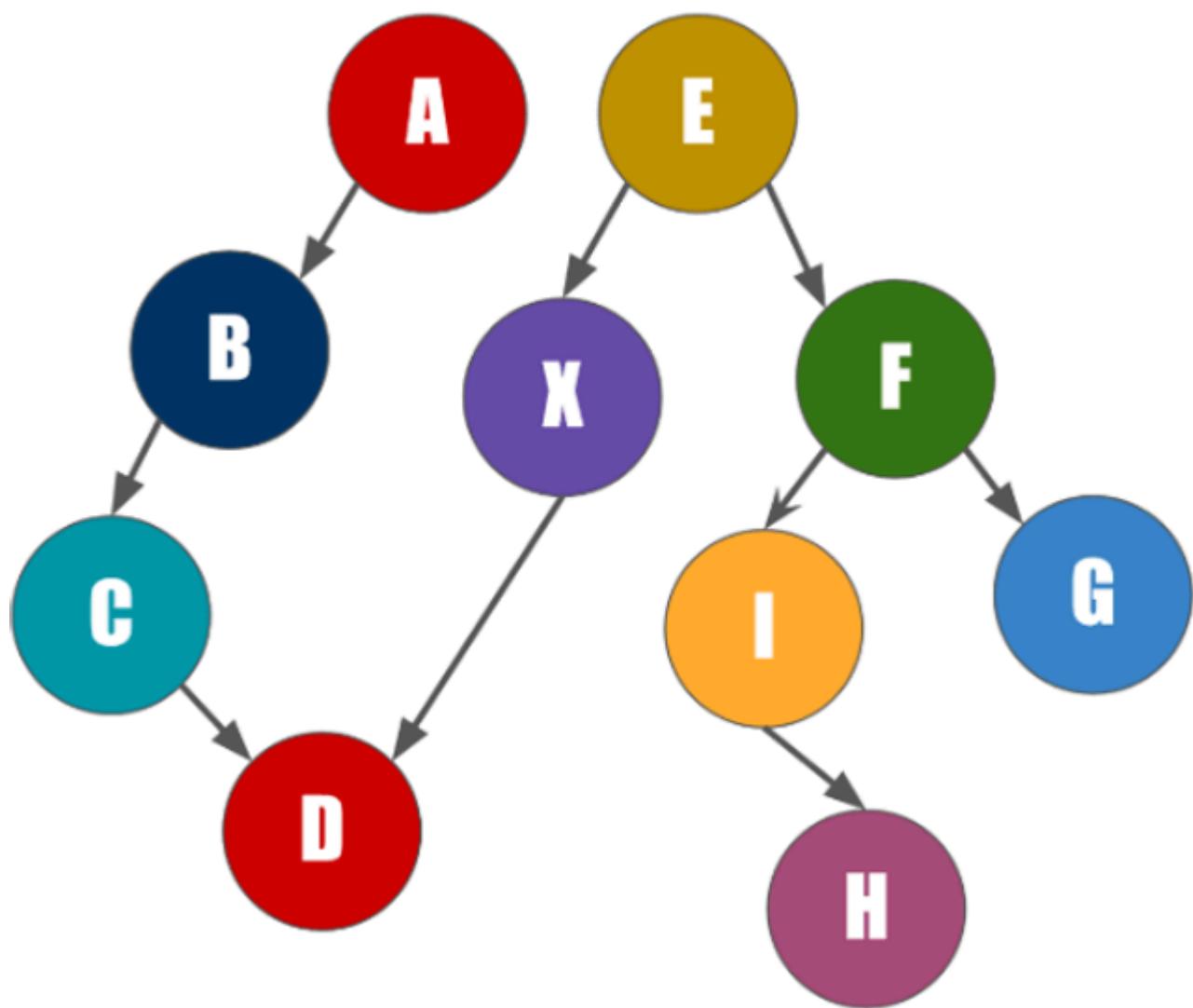


Figura 12 | Grafo topológico. Fonte: adaptada de Takenaka (2021).

Agora vamos ver como podemos implementar grafos em Python.

```
class Grafo:
```

ESTRUTURA DE DADOS

```
def __init__(self):
    self.vertices = {} # Dicionário para armazenar os vértices e suas adjacências

def adicionar_vertice(self, vertice):
    if vertice not in self.vertices:
        self.vertices[vertice] = [] # Inicializa a lista de adjacências vazia para o vértice

def adicionar_aresta(self, origem, destino):
    if origem in self.vertices and destino in self.vertices:
        self.vertices[origem].append(destino) # Adiciona destino à lista de adjacências de origem

def mostrar_vertices(self):
    print()
    for vertice in self.vertices:
        print(vertice)

def mostrar_arestas(self):
    print()
    for origem in self.vertices:
        for destino in self.vertices[origem]:
            print(f'{origem} -> {destino}')

# Cria um objeto grafo
grafo = Grafo()

# Adiciona vértices ao grafo
grafo.adicionar_vertice('A')
grafo.adicionar_vertice('B')
grafo.adicionar_vertice('C')
grafo.adicionar_vertice('D')

# Adiciona arestas ao grafo
grafo.adicionar_aresta('A', 'B')
grafo.adicionar_aresta('A', 'C')
grafo.adicionar_aresta('B', 'C')
grafo.adicionar_aresta('C', 'D')

# Mostra os vértices e as arestas do grafo
grafo.mostrar_vertices()
grafo.mostrar_arestas()
```

Nesse código, a classe Grafo é usada para representar um grafo. Os métodos da classe permitem adicionar vértices e arestas ao grafo, bem como mostrar os vértices e as arestas presentes no grafo.

Vamos Exercitar?

Inicialmente, fomos confrontados com a seguinte questão a ser resolvida: você faz parte de uma equipe em uma empresa dedicada à criação de jogos e está incumbido da tarefa de desenvolver um jogo baseado em labirintos. Nesse jogo, o jogador possui a liberdade de navegar por diversos caminhos, inclusive optando por retornar por onde veio. Importante destacar que não existem limitações quanto ao número de vezes que uma mesma sala pode ser visitada ou à escolha do trajeto a ser percorrido.

Para estruturar o design inicial do labirinto, sua equipe considerou um esquema específico, ilustrado na figura subsequente. Nessa representação gráfica, as áreas coloridas de cinza correspondem aos ambientes ou salas do labirinto, enquanto as seções em laranja representam as passagens que interligam essas salas. Adicionalmente, cada sala recebe uma identificação única, tal como v1, v2, v3, v4, etc., para facilitar a referência e orientação dentro do labirinto.

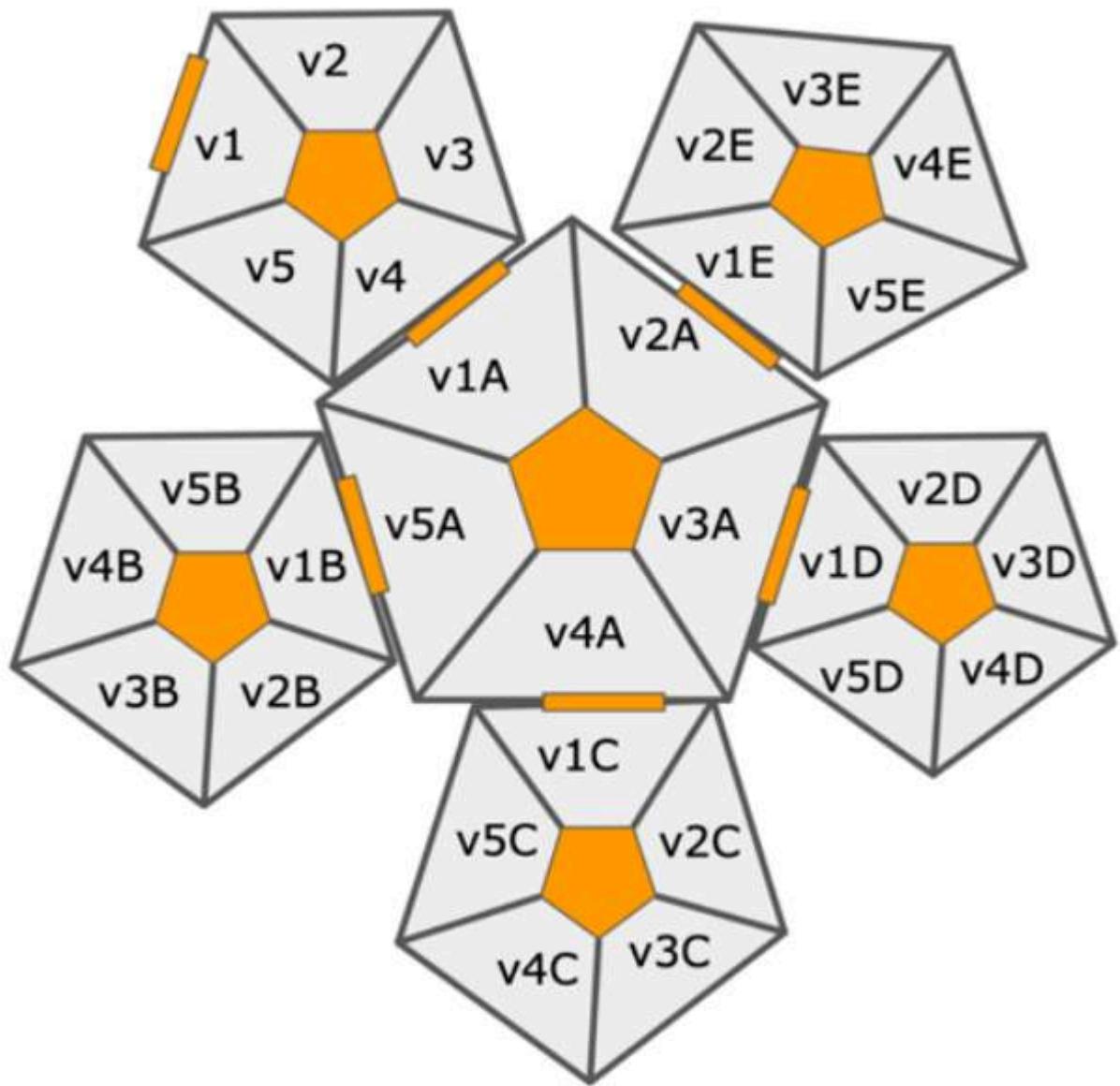


Figura 13 | Exemplo do labirinto. Fonte: adaptada de Takenaka (2021).

As salas v1, v2, v3, v4 e v5 estão interligadas através de um pentágono situado no centro. Existe uma conexão direta entre a sala v4 e a sala v1A, permitindo o trânsito em ambas as direções. Similarmente, a sala v1E é acessível a partir da sala v2A, e vice-versa. Durante essa etapa de desenvolvimento, a função principal da aplicação consiste em apresentar as salas adjacentes à localização atual do jogador. Após a escolha do jogador por uma das salas adjacentes, ele se desloca para a nova sala escolhida e o processo se repete, indicando novamente as salas vizinhas disponíveis para exploração.

Até o momento, o jogo não possui regras ou objetivos específicos estabelecidos. No entanto, há potencial para aprimorá-lo posteriormente, incorporando diversas modificações, tais como a alteração dos nomes das salas, a seleção aleatória do ponto de partida, a atribuição de pontos ao jogador conforme ele acessa certas salas, além da introdução de metas como alcançar uma sala específica com o mínimo de movimentos possíveis ou visitar todas as salas uma única vez sem repetições. Inicialmente, o jogador pode iniciar sua jornada a partir da sala v1. Com o avanço do desenvolvimento, a aplicação continuará a informar as salas adjacentes a cada nova posição do jogador, incentivando a tomada de decisão sobre a próxima sala a ser explorada.

Embora o jogo ainda não tenha regras ou objetivos claramente definidos, existe a possibilidade de torná-lo mais complexo e envolvente no futuro, por meio da implementação das sugestões mencionadas, como a modificação dos nomes das salas, a definição aleatória do ponto inicial, a implementação de um sistema de pontuação baseado na entrada em salas específicas e a introdução de desafios, como alcançar determinadas salas em um número limitado de movimentos ou visitar todas as salas sem repetições. O ponto de partida para os jogadores pode ser estabelecido na sala v1.

Saiba mais

Fundamentos de grafos e sua representação:

- ARAÚJO, M. [GRAFOS – 1ª PARTE. Revista Programar](#), 2007.

Caminhos e ciclos em grafos:

- SANTOS, M. de. [Ciclos Hamiltonianos em Grafos. Ciência e Natura](#).

Grafos com custos, topológicos, direcionados e não direcionados

- SAAD, D. [Tópicos Especiais em Algoritmos. Daniel Saad](#).

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

BORIN, V. P. **Estrutura de dados**. 1. ed. São Paulo: Contentus, 2020.

LAMBERT, K. A. **Fundamentos de Python: estruturas de dados**. São Paulo: Cengage Learning, 2022.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

TAKENAKA, R. R. M. **Introdução a grafos.** Estrutura de Dados, 2021. Disponível em:
<http://bit.ly/497p4MU>. Acesso em: 6 fev. 2024.

Aula 2

Operações em Grafos

Operações em Grafos

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante! Nesta videoaula, vamos mergulhar no mundo das operações em grafos e explorar algoritmos de busca essenciais para sua prática profissional. Aprenderemos sobre busca em largura e busca em profundidade em Python, habilidades essenciais para analisar redes complexas e resolver uma variedade de problemas do mundo real. Não perca essa oportunidade de aprimorar seus conhecimentos em grafos e algoritmos!

Ponto de Partida

Olá, estudante!

Nesta aula, exploraremos uma diversidade de operações e algoritmos aplicáveis a grafos, uma ferramenta versátil para representar numerosos cenários reais. A habilidade de abordar problemas com criatividade é fundamental, visto que muitas questões teóricas sobre grafos demandam avanços computacionais para sua comprovação. No entanto, os progressos tecnológicos têm ampliado nossas capacidades de ultrapassar tais barreiras.

O campo da Ciência de Dados, tem ganhado destaque devido às tecnologias emergentes que facilitam a análise de vastos volumes de dados. Grafos são frequentemente empregados em algoritmos de inteligência artificial, onde podem não ser visíveis diretamente, mas estão incorporados em estruturas algorítmicas maiores. Um exemplo disso é o algoritmo K-NN (K

vizinhos mais próximos), que é utilizado para classificar objetos com base na semelhança de suas características com as de grupos previamente definidos. Tal processo exemplifica a aplicação de busca em largura em grafos, similarmente utilizada em sistemas de GPS, redes *peer-to-peer*, comunicações entre dispositivos móveis e satélites, entre outras, visando a identificação de elementos adjacentes e a expansão dessa busca aos vizinhos desses elementos (Borin, 2020).

Além da busca em largura, discutiremos o algoritmo de busca em profundidade, útil para descobrir trajetos, realizar ordenações topológicas, identificar componentes fortemente conectados e solucionar labirintos. Demonstraremos como essas buscas, tanto em profundidade quanto em largura, podem ser implementadas em Python, ampliando seu arsenal de ferramentas para enfrentar desafios de programação.

Para assimilarmos de forma prática tudo o que aprenderemos, considere que você está envolvido em um projeto de manutenção da infraestrutura de transporte urbano. Um aspecto importante desse sistema é a capacidade de identificar rotas alternativas para evitar a interrupção do fluxo de tráfego. Isso é essencial para o planejamento eficaz de desvios ou outras medidas que reduzam os transtornos causados pelas obras, como a falta de comunicação sobre desvios ou congestionamentos que não são aliviados pelos desvios propostos.

Aplique os conhecimentos adquiridos sobre grafos para desenvolver uma funcionalidade que permita verificar se um segmento da rede viária está bloqueado e se existem rotas alternativas disponíveis. Questione como representar o bloqueio de uma via e a ocorrência de múltiplos bloqueios simultâneos. Pense em estratégias para identificar a existência de rotas alternativas. Considere também a importância da direção do tráfego nas vias.

Que tal explorar os grafos a seguir para solucionar essa problemática?

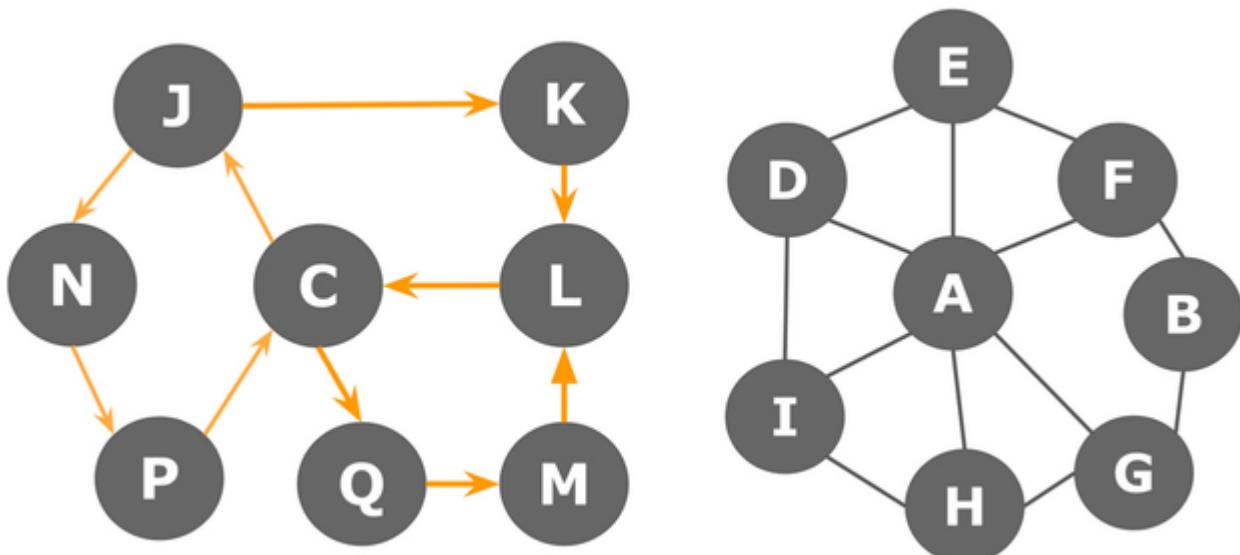


Figura 1 | Grafos do exercício. Fonte: adaptada de Takenaka (2021).

ESTRUTURA DE DADOS

Prepare-se para mergulhar nesses conceitos, expandindo seu entendimento sobre grafos e suas poderosas aplicações.

Bons estudos!

Vamos Começar!

Introdução a operações em grafos e algoritmos de busca para grafos

Como já vimos anteriormente, grafos são estruturas de dados fundamentais, usadas para modelar uma ampla variedade de sistemas no mundo real. Eles consistem em vértices (ou nós) e arestas (ou arcos) que conectam pares de vértices. Os grafos são poderosas ferramentas para representar redes sociais, mapas, estruturas de dados e muito mais, permitindo a análise de relações complexas e interconexões entre elementos.

Quando falamos de operações em grafos, estamos nos referindo a operações fundamentais que incluem a adição e remoção de vértices e arestas, a verificação da existência de uma aresta entre dois vértices e a busca de caminhos e ciclos dentro do grafo. Além disso, muitas vezes é necessário percorrer (ou "visitar") todos os vértices de um grafo de forma sistemática, para o qual algoritmos de busca são essenciais.

Os algoritmos de busca em grafos são essenciais para explorar grafos em diversas aplicações. Esses algoritmos, como a busca em largura e a busca em profundidade, são fundamentais para visitar todos os vértices de um grafo, cada um seguindo uma ordem de visita distinta. A busca em largura expande-se pelos vértices mais próximo primeiro, similarmente à propagação de uma notícia, e é utilizada em algoritmos como o de Prim e Dijkstra.

Já a busca em profundidade explora tão profundamente quanto possível ao longo de cada ramo antes de retroceder, assemelhando-se à navegação em um labirinto. Ambos os métodos são ilustrados na Figura 2, com a busca em profundidade representada na parte (a) e a busca em largura na parte (b), demonstrando suas diferenças nos percursos (Takenaka, 2021).

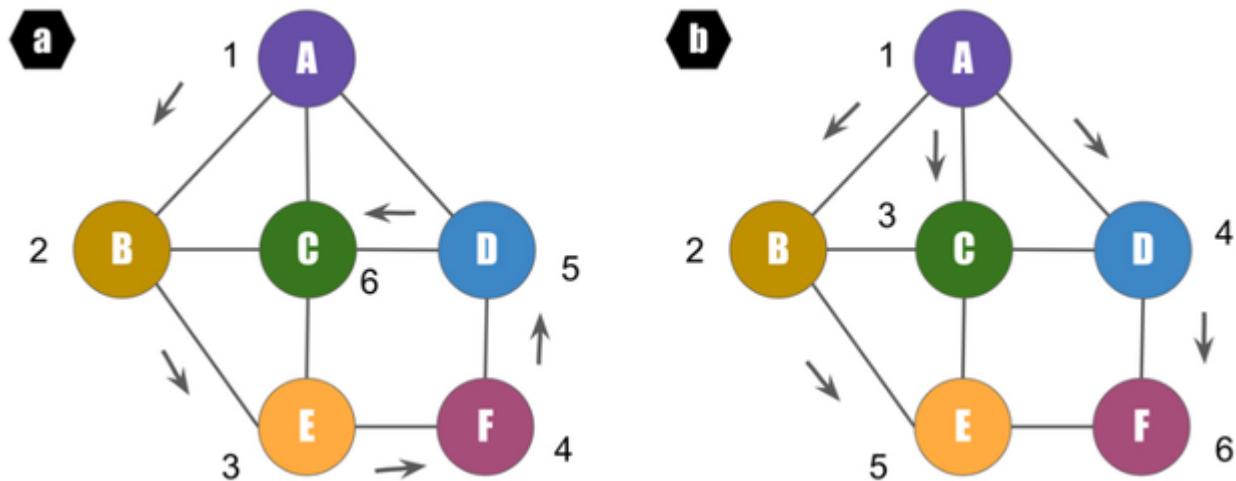


Figura 2 | Comparação de busca em profundidade (a) e busca em largura (b). Fonte: adaptada de Takenaka (2021).

Para entender as diferenças entre busca em profundidade e busca em largura, considere o exemplo de uma viagem ao redor do mundo. Utilizando a busca em profundidade, você exploraria todos os países de um continente, como a América do Sul, antes de passar para o próximo. Já com a busca em largura, você visitaria um país de cada continente por vez, alternando entre os continentes a cada nova rodada. Ambos os métodos podem ser aplicáveis a diversos problemas, mas cada um tem situações onde é mais eficaz. Enquanto a busca em profundidade pode ser mais rápida em certos cenários, a busca em largura é preferida para encontrar o caminho mais curto, devido à sua eficiência em explorar opções mais próximas primeiro (Lambert, 2022).

Busca em profundidade em Python

Na técnica de Busca em Profundidade, também conhecida como DFS (do inglês, *Depth-First Search*), adotamos a estratégia de avançar o máximo possível a partir do vértice de origem, registrando os vértices por onde passamos, continuando esse processo até que não restem mais vértices inexplorados ao longo dessa rota (Szwarcfiter; Markenzon, 2020).

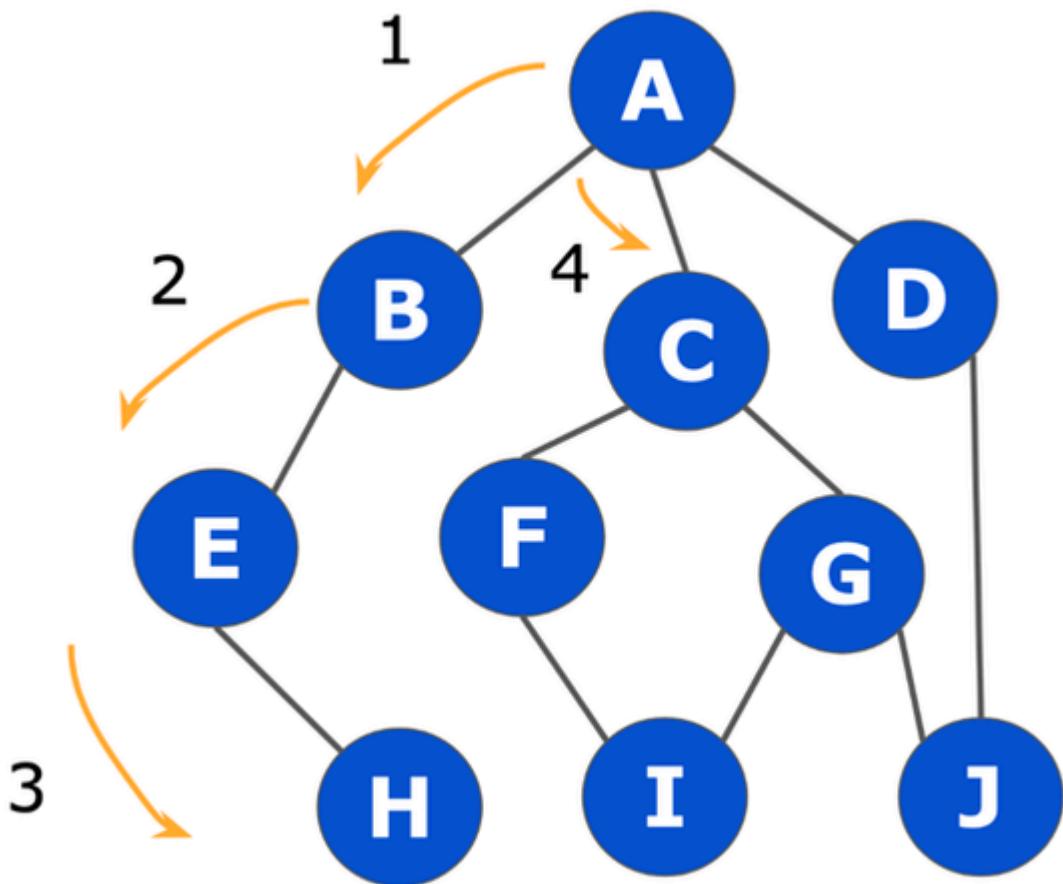


Figura 3 | Caminho inicial de busca em profundidade. Fonte: adaptada de Takenaka (2021).

Nessa busca, adotamos a técnica de retroceder (*backtracking*), explorando novas direções a partir de cada vértice visitado, sempre que viável, prosseguindo até retornarmos ao vértice de partida. Esse processo é semelhante ao método utilizado para sair de um labirinto, onde você mantém contato constante com uma das paredes – nesse caso, a direita – para guiar seu caminho, retrocedendo nos becos sem saída e seguindo adiante até encontrar a saída, garantindo que não se perca.

No entanto, na busca em profundidade, a gestão dos vértices explorados é feita por meio de uma pilha, em vez da analogia da mão na parede. Nesse método, adicionamos à pilha os vértices a serem explorados e removemos de lá para determinar qual será o próximo vértice a examinar. Ao visitar um vértice, seus vizinhos são empilhados. Quando uma visita é concluída, para descobrir qual é o próximo vértice a ser explorado, desempilhamos um elemento, que indica a nova direção (Lambert, 2022).

Apresentamos a seguir seu algoritmo: útil para diversas finalidades, incluindo a identificação de caminhos entre dois vértices específicos, a realização de ordenações topológicas, a localização

ESTRUTURA DE DADOS

de pontes e componentes fortemente conectados, a detecção de ciclos, a verificação da conectividade de grafos e a resolução de labirintos, além de outras aplicações.

```
class Grafo:  
    def __init__(self):  
        self.grafo = {}  
  
    def adicionar_vertice(self, vertice):  
        if vertice not in self.grafo:  
            self.grafo[vertice] = []  
  
    def adicionar_aresta(self, vertice_origem, vertice_destino):  
        if vertice_origem in self.grafo:  
            self.grafo[vertice_origem].append(vertice_destino)  
        else:  
            print()  
  
    def dfs(self, vertice, visitados=None):  
        if visitados is None:  
            visitados = set()  
        visitados.add(vertice)  
        print(vertice, end=' ')  
  
        for vizinho in self.grafo[vertice]:  
            if vizinho not in visitados:  
                self.dfs(vizinho, visitados)  
  
# Exemplo de uso  
g = Grafo()  
g.adicionar_vertice('A')  
g.adicionar_vertice('B')  
g.adicionar_vertice('C')  
g.adicionar_vertice('D')  
g.adicionar_vertice('E')  
  
g.adicionar_aresta('A', 'B')  
g.adicionar_aresta('A', 'C')  
g.adicionar_aresta('B', 'D')  
g.adicionar_aresta('C', 'E')  
  
print(end=' ')  
g.dfs('A')
```

No código, criamos uma classe Grafo para representar o grafo. Implementamos métodos para adicionar vértices e arestas ao grafo: o método “**dfs**” realiza a busca em profundidade a partir de

um vértice dado. Ele utiliza um conjunto visitado para controlar quais vértices já foram visitados.

Na busca em profundidade, visitamos um vértice e, em seguida, recursivamente visitamos todos os seus vizinhos não visitados. Finalmente, realizamos um exemplo de uso do algoritmo com um grafo de exemplo e imprimimos os vértices visitados durante a busca em profundidade (Alves, 2021).

Siga em Frente...

Busca em largura em Python

Na busca em largura (BFS), começamos do vértice inicial e visitamos todos os vértices adjacentes. Em seguida, partimos desses vértices, repetindo o procedimento. É como disseminar uma notícia para os amigos mais próximos, que por sua vez a repassam aos amigos mais próximos deles, e assim por diante. Eventualmente, a notícia pode chegar a alguém que já a conhecia (vértice já visitado).

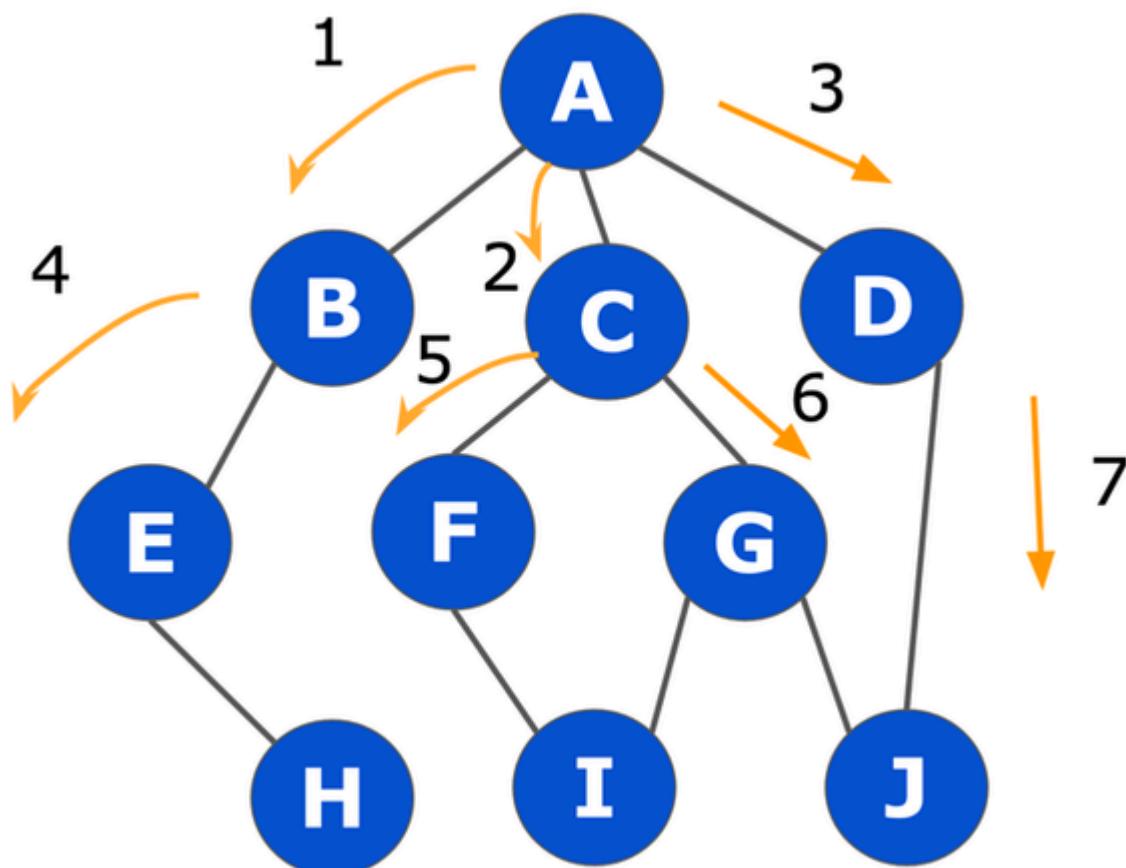


Figura 4 | Caminho inicial de busca em largura. Fonte: adaptada de Takenaka (2021).

ESTRUTURA DE DADOS

No algoritmo de busca em largura, controlamos os vértices visitados usando uma fila. Adicionamos os vértices a serem visitados na fila e os removemos para determinar o próximo a ser visitado. Ao visitar um vértice, adicionamos seus vizinhos à fila. Após cada visita, retiramos o próximo vértice da fila para continuar a exploração. Essa abordagem é útil em várias aplicações, como identificar redes *peer-to-peer*, calcular o grau de separação em redes sociais e sistemas de navegação por GPS.

Observe o código a seguir:

```
from collections import deque

def bfs(graph, start):
    visited = set() # Conjunto para armazenar os nós já visitados
    queue = deque([start]) # Uma fila para armazenar os nós que serão explorados, começando pelo nó inicial

    while queue: # Enquanto houver elementos na fila
        vertex = queue.popleft() # Remove o elemento mais à esquerda da fila e o armazena em 'vertex'
        if vertex not in visited: # Se o vértice não foi visitado
            visited.add(vertex) # Marca o vértice como visitado
            print(vertex, end=' ') # Imprime o vértice visitado

        # Adiciona todos os vizinhos não visitados do vértice atual à fila
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                queue.append(neighbor)

# Exemplo de uso
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

bfs(graph, 'A') # Inicia a busca em largura a partir do vértice 'A'
```

Olhando para o código, temos os seguintes pontos:

- Importação: O módulo deque da biblioteca *collections* é importado para utilizar uma fila que permite adições e remoções eficientes de elementos.

- Função bfs: A função bfs recebe um grafo e um nó de partida start como argumentos. O grafo é representado como um dicionário, onde cada chave é um vértice e os valores são listas dos vértices adjacentes.
- Estruturas de dados: Dois tipos de estruturas de dados são usados: um conjunto “*visited*” para armazenar os nós já visitados e evitar ciclos e uma fila *queue* para armazenar os nós que serão explorados, utilizando a estratégia FIFO (*First-In First-Out*).
- Loop principal: O *loop* continua enquanto houver elementos na fila. A cada iteração, um vértice é removido da fila, verificado se já foi visitado, e se não, é marcado como visitado e seus vizinhos não visitados são adicionados à fila.
- Execução: No exemplo de uso, um grafo é definido e a função “**bfs**” é chamada com esse grafo e um vértice inicial ('A'). O algoritmo então explora o grafo usando a busca em largura e imprime os vértices na ordem em que são visitados.

Em síntese, os grafos são essenciais para modelar e resolver problemas que envolvem relações e conexões complexas. As operações básicas em grafos, juntamente com os algoritmos de busca BFS e DFS, fornecem a base para uma ampla gama de aplicações computacionais, desde a otimização de redes até a análise de mídias sociais. Compreender esses conceitos e algoritmos é fundamental para qualquer pessoa que deseja aplicar técnicas de Ciência da computação para resolver problemas do mundo real.

Vamos Exercitar?

No ponto de partida desta aula, lhe apresentamos a seguinte problemática: você estava envolvido em um projeto de manutenção da infraestrutura de transporte urbano e sua tarefa era verificar se um segmento da rede viária está bloqueado e se existem rotas alternativas disponíveis. Você foi questionado sobre:

Como representar o bloqueio de uma via e a ocorrência de múltiplos bloqueios simultâneos?

Quais as possíveis estratégias para identificar a existência de rotas alternativas?

Qual a importância da direção do tráfego nas vias?

Para abordar o desafio proposto, a utilização de **grafos direcionados** (dígrafos) é fundamental. Os vértices do grafo representam interseções ou pontos críticos na rede viária, enquanto as arestas representam os segmentos de estrada que conectam esses pontos. O bloqueio de uma via pode ser representado pela remoção da aresta correspondente no grafo ou pela atribuição de um peso infinito ou um marcador específico a essa aresta, indicando que o caminho está intransitável.

Quando mais de uma via pública está interrompida, múltiplas arestas seriam removidas ou marcadas como bloqueadas. Para detectar a existência de caminhos alternativos, algoritmos de busca em grafos, como o de busca em largura (para encontrar o caminho mais curto) ou o de Dijkstra (para encontrar o caminho com menor custo, considerando pesos nas arestas), podem

ESTRUTURA DE DADOS

ser aplicados. A direção do tráfego é relevante e justifica o uso de grafos direcionados, pois a rota de desvio deve respeitar o sentido permitido nas vias.

Ademais, a atividade solicita a aplicação prática dos conceitos de grafos na resolução de um problema real de planejamento urbano, especificamente na gestão de vias públicas durante obras ou bloqueios. A representação da rede viária como um grafo direcionado permite modelar com precisão as direções do tráfego e os bloqueios nas vias. A remoção de arestas ou a alteração de seus pesos reflete os bloqueios.

A utilização de algoritmos de busca em grafos habilita a identificação eficiente de rotas alternativas, garantindo que as medidas de desvio propostas sejam viáveis e minimizem o impacto das obras no fluxo de trânsito. Essa abordagem não só exemplifica a aplicabilidade dos grafos em contextos práticos, mas também reforça a importância da computação gráfica na solução de problemas complexos de engenharia e planejamento urbano.

ESTRUTURA DE DADOS

```

141 def verificar_caminhos_alternativos(arestas, orientado=True):
142     # para cada aresta, verifica se ela for removida,
143     # há um caminho alternativo?
144
145     for aresta in arestas:
146         print()
147         u, v = aresta
148
149         # cria um grafo
150         grafo = Grafo(orientado)
151         # insere arestas no grafo
152         grafo.inserir_arestas(arestas)
153
154         # imprimir grafo
155         grafo.imprimir()
156
157         print("Remover {}, {}".format(u, v))
158         # remove uma das arestas
159         grafo.remover_aresta("aresta")
160
161         print("Busca em largura")
162         visitados = busca_em_largura(grafo, u)
163         if v in visitados:
164             print("Sim, há caminho alternativo")
165         else:
166             print("Nenhum caminho alternativo")
167
168
169     arestas1 = [
170         ('J', 'K'),
171         ('J', 'N'),
172         ('N', 'P'),
173         ('P', 'C'),
174         ('C', 'Q'),
175         ('C', 'J'),
176         ('Q', 'M'),
177         ('M', 'L'),
178         ('L', 'C'),
179         ('K', 'L'),
180     ]
181     verificar_caminhos_alternativos(arestas1, orientado=True)
182     arestas2 = [
183         ('E', 'D'),
184         ('E', 'F'),
185         ('E', 'A'),
186         ('D', 'A'),
187         ('D', 'I'),
188         ('A', 'H'),
189         ('I', 'A'),
190         ('I', 'H'),
191         ('H', 'G'),
192         ('G', 'A'),
193         ('G', 'B'),
194         ('B', 'F'),
195         ('F', 'A'),
196     ]
197     verificar_caminhos_alternativos(arestas2, orientado=False)

```

Figura 5 | Código solução. Fonte: adaptada de Takenaka (2021).

Saiba mais

Introdução a operações em grafos e Algoritmos de busca para grafos:

- CHAGAS, F. [Grafos e algoritmos](#). Programadores ajudando Programadores - Medium.

Busca em largura em Python:

- BUSCA em largura. [Algoritmos em Python](#).

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

BORIN, V. P. **Estrutura de dados**. 1. ed. São Paulo: Contentus, 2020.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

TAKENAKA, R. M. **Introdução a grafos**. Estrutura de Dados, 2021. Disponível em:
<http://bit.ly/497p4MU>. Acesso em: 6 fev. 2024.

Aula 3

Caminhos e Ciclos

Caminhos e Ciclos



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Nesta videoaula, exploraremos algoritmos fundamentais de teoria dos grafos aplicados em redes de comunicação e sociais. Aprenderemos sobre algoritmos de caminhos mais curtos e detecção de ciclos, essenciais para otimizar rotas e identificar padrões em redes complexas. Esses conhecimentos são cruciais para profissionais que lidam com análise de dados, redes sociais e logística. Venha aprimorar suas habilidades e expandir seu conhecimento nessa área em constante crescimento! Não perca!

Ponto de Partida

Olá, estudante!

Nesta aula, exploraremos alguns conceitos fundamentais em teoria dos grafos, focando em algoritmos de caminhos mais curtos, detecção de ciclos e sua aplicação em redes de comunicação e sociais. Esses conteúdos são de extrema importância, pois nos ajudam a entender de forma prática como as informações são organizadas e fluem em sistemas complexos, como redes de computadores, redes sociais e sistemas de transporte.

Durante a aula, vamos nos deparar com a problemática de encontrar o caminho mais curto entre dois pontos em um grafo, uma tarefa essencial em muitas aplicações práticas, como navegação GPS e roteamento de pacotes em redes de computadores. Além disso, discutiremos a detecção de ciclos em grafos, para identificar *loops* ou feedbacks em sistemas complexos, como redes sociais, onde a presença de ciclos pode indicar padrões de interação entre os usuários.

Ao longo da aula, fique atento aos algoritmos e às técnicas apresentadas, pois eles serão fundamentais para resolver problemas práticos relacionados à organização e análise de dados em redes de comunicação e sociais. Para assimilarmos de maneira prática, você foi contratado por uma empresa de marketing para identificar os influenciadores em uma determinada rede social. Um influenciador é definido como um usuário que possui uma grande influência sobre outros usuários na rede, ou seja, suas postagens têm um grande impacto e alcançam muitas pessoas. Sua tarefa é desenvolver um algoritmo em Python que identifique os influenciadores com base nas interações entre os usuários na rede.

Aproveite e esteja preparado para explorar como a teoria dos grafos pode ser aplicada no cotidiano profissional, seja na otimização de rotas de transporte público, na análise de redes sociais para identificar influenciadores ou na detecção de padrões de comunicação em sistemas de telecomunicações.

Vamos começar!

Vamos Começar!

Algoritmos de caminhos mais curtos

No universo dos grafos, um dos problemas mais fundamentais e estudados é o de encontrar o caminho mais curto entre dois vértices. Esse problema não apenas possui uma aplicabilidade teórica significativa na ciência da computação e matemática, mas também é essencial em várias aplicações práticas, como sistemas de navegação GPS, redes de telecomunicações, planejamento de rotas de transporte e até em redes sociais para encontrar o "caminho" mais curto entre pessoas. Nesta aula, exploraremos os algoritmos clássicos para encontrar caminhos mais curtos em grafos, suas aplicações e alguns exemplos que ilustram seu funcionamento (Takenaka, 2021).

Definição

O problema do caminho mais curto pode ser formulado da seguinte maneira: dado um grafo $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas, e uma função de peso $w: E \rightarrow \mathbb{R}$, que associa um peso (ou custo) a cada aresta do grafo, queremos encontrar o caminho de menor peso entre dois vértices específicos u e v do grafo (Borin, 2020).

Principais Algoritmos para encontrar o caminho mais curto em um grafo

Existem vários algoritmos projetados para resolver o problema do caminho mais curto, cada um com suas próprias características e casos de uso. Os mais conhecidos incluem o algoritmo de **Dijkstra**, o algoritmo de **Bellman-Ford**, o algoritmo de **Floyd-Warshall** e o algoritmo A*.

Algoritmo de Dijkstra

O algoritmo de Dijkstra é talvez o mais conhecido e utilizado para encontrar o caminho mais curto em grafos ponderados sem arestas de peso negativo. A ideia central é simples: a partir de um vértice de origem, o algoritmo explora todos os vizinhos, atualizando o custo do caminho mais curto para cada vértice. O processo se repete, escolhendo sempre o vértice com o menor custo de caminho acumulado ainda não explorado, até que todos os vértices tenham sido visitados.

Imagine que você quer encontrar o caminho mais rápido de sua casa ao trabalho. Cada rua é uma aresta do grafo, e o tempo para percorrê-la é o peso da aresta. Utilizando o algoritmo de Dijkstra, começamos na sua casa, explorando as ruas com menor tempo de percurso primeiro, até chegar ao trabalho. Iremos nos aprofundar nele com mais detalhes futuramente.

Algoritmo de Bellman-Ford

O algoritmo de Bellman-Ford estende a capacidade do algoritmo de Dijkstra ao permitir arestas de peso negativo, oferecendo também a detecção de ciclos negativos no grafo. Ele opera relaxando repetidamente todas as arestas, atualizando o custo do caminho para cada vértice se um caminho mais curto for encontrado. Apesar de mais lento que o Dijkstra para grafos sem pesos negativos, o Bellman-Ford é útil em situações onde pesos negativos estão presentes. Em uma rede de trocas comerciais entre cidades, onde os custos de transporte podem variar, incluindo valores negativos (subvenções, por exemplo), o algoritmo de Bellman-Ford pode determinar o custo mínimo para enviar produtos de uma cidade a outra (Lambert, 2022).

Algoritmo de Floyd-Warshall

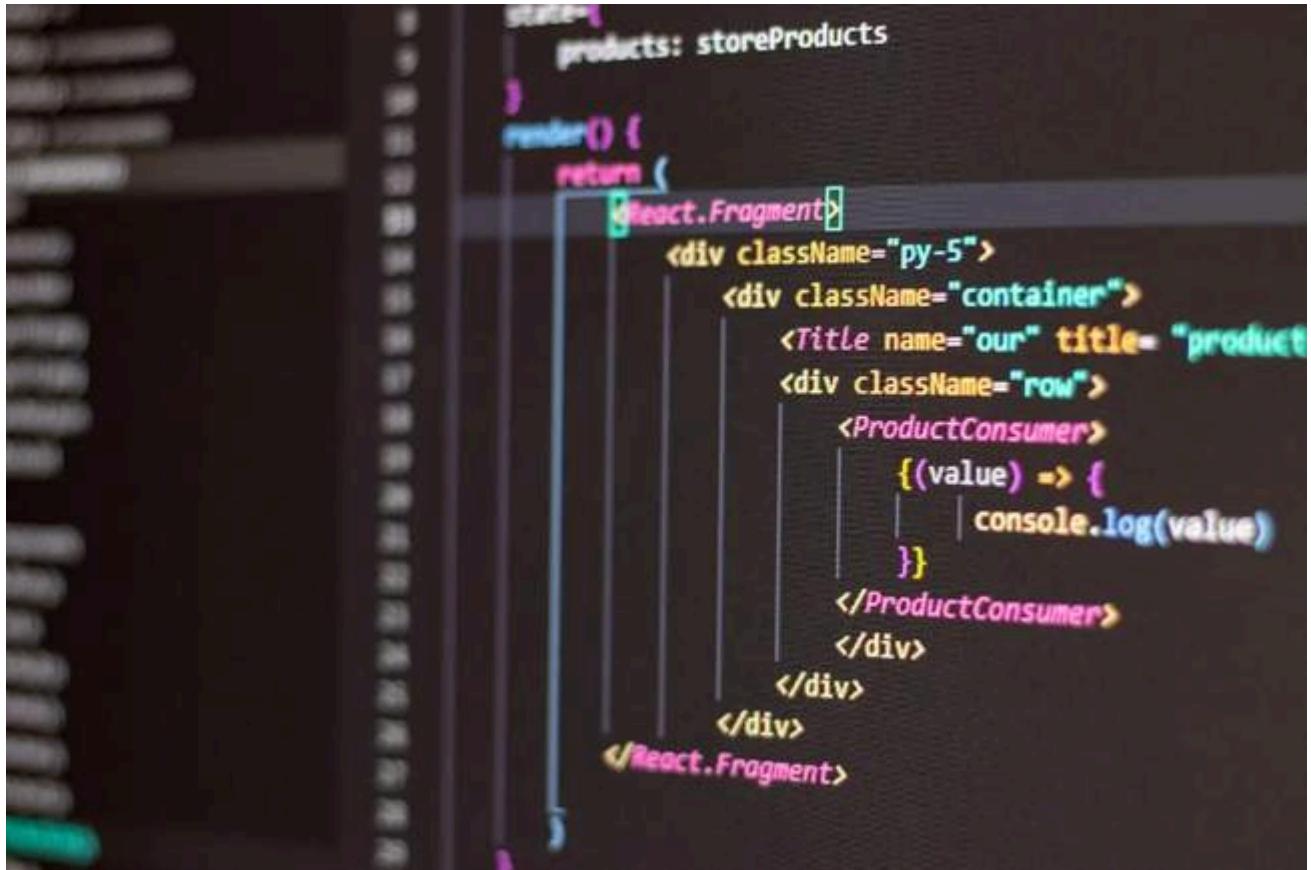
Diferente dos algoritmos anteriores, o Floyd-Warshall é usado para encontrar o caminho mais curto entre todos os pares de vértices em um grafo. Ele é ideal para aplicações que necessitam de uma visão completa das distâncias entre todos os pontos, como em análises de rede. O algoritmo opera por meio de uma abordagem de programação dinâmica, calculando iterativamente as distâncias mais curtas considerando cada vértice como um ponto intermediário possível nos caminhos. Em uma empresa com várias filiais, onde cada conexão entre filiais tem um custo associado (tempo, dinheiro), o algoritmo de Floyd-Warshall pode determinar o custo mínimo para comunicação ou transporte entre todas as filiais (Lambert, 2022).

Algoritmo A*

O algoritmo A* (lê-se “A star”, ou “A estrela”) é uma melhoria do Dijkstra que incorpora heurísticas no processo de busca para encontrar o caminho mais curto de maneira mais eficiente. Ele é particularmente útil em buscas em espaços de estados grandes, como em mapas ou jogos, onde uma heurística bem escolhida pode significativamente acelerar a busca sem comprometer a garantia de encontrar o caminho mais curto (Lambert, 2022).

Escolha do algoritmo

A escolha de qual algoritmo utilizar depende de vários fatores, incluindo se o grafo tem arestas de peso negativo, se a busca é por um caminho mais curto entre um único par de vértices ou entre todos os pares e se a eficiência é uma preocupação primordial. Os algoritmos de caminhos mais curtos têm uma ampla gama de aplicações. Na logística, são utilizados para otimizar as rotas de entrega. Nas telecomunicações, ajudam a encontrar a rota mais eficiente para a transmissão de dados. Em redes sociais, podem ser usados para encontrar a “distância” ou o número mínimo de conexões entre dois usuários.



```
products: storeProducts

render() {
  return (
    <React.Fragment>
      <div className="py-5">
        <div className="container">
          <Title name="our" title="products" />
          <div className="row">
            <ProductConsumer>
              {(value) => {
                console.log(value)
              }}
            </ProductConsumer>
          </div>
        </div>
      </React.Fragment>
    )
}
```

Figura 1 | Algoritmo. Fonte: Pexels

Siga em Frente...

Detecção de ciclos

A detecção de ciclos em grafos é um tema fundamental na teoria dos grafos e tem aplicações significativas em várias áreas da Ciência da Computação, como análise de redes, sistemas operacionais e otimização de processos. Como já vimos, um ciclo em um grafo é uma sequência de vértices em que o primeiro e o último vértices são os mesmos, sem repetir os demais vértices na sequência. Anteriormente, falamos sobre alguns algoritmos que nos auxiliam na solução desse problema, como a Busca em Profundidade. Agora, exploraremos os conceitos básicos da detecção de ciclos em grafos e apresentaremos alguns métodos e exemplos para ilustrar como esses ciclos podem ser identificados.

Como já sabemos, um grafo pode ser direcionado ou não direcionado. A presença de ciclos em um grafo pode indicar redundâncias, dependências ou inconsistências dentro de um sistema ou uma rede modelada pelo grafo. Por exemplo, em um sistema de gerenciamento de projetos, um ciclo em um grafo de dependências de tarefas pode indicar uma impossibilidade de completar o projeto devido a uma dependência circular entre as tarefas.

Métodos de detecção de ciclos

Existem vários algoritmos para detectar ciclos em grafos, cada um adequado para tipos específicos de grafos (direcionados ou não direcionados).

Busca em profundidade (*Depth-First Search - DFS*)

A busca em profundidade (DFS) é uma técnica comum para detectar ciclos em grafos. Ela explora tão profundamente quanto possível ao longo de cada ramo antes de retroceder. Para grafos não direcionados, um ciclo é detectado se uma aresta leva a um vértice que já foi visitado. Para grafos direcionados, além de verificar vértices visitados, também precisamos considerar os ancestrais do vértice atual na árvore de busca DFS para detectar ciclos.

Algoritmo de *Union-Find*

Para grafos não direcionados, o algoritmo de Union-Find é eficaz na detecção de ciclos. Esse algoritmo gerencia um conjunto de elementos particionados em subconjuntos disjuntos e pode rapidamente determinar se a adição de uma nova aresta criaria um ciclo, verificando se os dois vértices da aresta pertencem ao mesmo subconjunto.

Em uma rede de amizades, ao adicionar uma nova conexão entre duas pessoas, podemos usar o algoritmo de *Union-Find* para verificar se essa conexão criaria um "ciclo de amizade", onde um grupo de pessoas está interconectado.

Detecção de ciclos em grafos direcionados

Para grafos direcionados, além do DFS, podemos usar algoritmos específicos como o de Tarjan, que identifica componentes fortemente conectados (subconjuntos de vértices onde cada par de vértices é mutuamente alcançável) e, consequentemente, ciclos.

Em sistemas de controle de versão, onde diferentes versões de um arquivo dependem umas das outras, a detecção de ciclos pode ajudar a identificar dependências circulares que impedem a construção de uma versão estável do sistema (Lambert, 2022).

A detecção de ciclos tem várias aplicações práticas. Em redes de computadores, é usada para evitar rotas de *loop* que podem causar congestionamento de rede. Em análise de sistemas, ajuda a identificar *deadlocks* em sistemas operacionais. Em teoria dos jogos e economia, pode ser usada para detectar ciclos de feedback positivo que levam a instabilidades em modelos econômicos. A capacidade de detectar ciclos em grafos é uma ferramenta valiosa para a modelagem e análise de sistemas complexos. Os métodos e exemplos apresentados destacam a importância dessa capacidade em várias aplicações, desde a gestão de projetos até a otimização de redes. Vamos entender ainda mais como isso nos auxilia em problemas mais próximos do nosso cotidiano?

Teoria dos grafos em redes de comunicação e sociais

A teoria dos grafos é um ramo da Matemática e Ciência da Computação que estuda as relações entre objetos. Essa teoria é fundamental para entender as complexidades das redes de comunicação e sociais, onde os objetos (nós) representam entidades como pessoas, dispositivos ou páginas da web, e as conexões (arestas) representam relações ou interações entre essas entidades. Abordaremos a seguir a aplicação da teoria dos grafos em redes de comunicação e sociais, destacando conceitos chave e fornecendo exemplos práticos (Lambert, 2022).

Grafos em redes de comunicação

Nas redes de comunicação, os grafos modelam a estrutura e a dinâmica da transmissão de dados entre dispositivos conectados. Nesses grafos, os vértices representam dispositivos de rede (como roteadores, switches e terminais de usuário) e as arestas representam as conexões físicas ou lógicas (como cabos Ethernet, links Wi-Fi, ou conexões VPN).

Considere uma pequena empresa com uma rede interna. Os computadores dos funcionários, o servidor de arquivos e o roteador de internet podem ser representados como vértices em um grafo. As conexões físicas por cabos ou conexões Wi-Fi entre esses dispositivos são as arestas do grafo. A análise desse grafo pode revelar a eficiência da rede, pontos de falha potenciais e caminhos ótimos para a transmissão de dados (Alves, 2021).

Grafos em redes sociais

Em redes sociais, os grafos são usados para modelar as relações entre indivíduos ou organizações. Aqui, os vértices representam usuários ou entidades, e as arestas representam as relações entre eles, como amizade, seguimento ou interações.

Em uma rede social como o Facebook, um usuário pode ser representado por um vértice e uma aresta pode representar uma relação de amizade entre dois usuários. A análise desse grafo social pode identificar padrões de interação, comunidades de usuários com interesses comuns e indivíduos influentes dentro da rede.

Análise de grafos em redes

A análise de grafos em redes permite a identificação de características importantes, como:

- Componentes conectados

Em uma rede social, componentes conectados podem representar grupos de usuários que estão todos interconectados de alguma forma, indicando comunidades com interesses compartilhados.

- Caminhos mais curtos:

Em redes de comunicação, encontrar o caminho mais curto entre dois dispositivos pode otimizar a transmissão de dados, reduzindo a latência e aumentando a eficiência da rede.

- Centralidade:

A centralidade de um vértice em um grafo de rede social pode indicar a influência de um usuário. Usuários com alta centralidade podem ser influenciadores importantes dentro da rede.

Aplicações da teoria dos grafos

- Otimização de redes: a teoria dos grafos ajuda a otimizar o layout e o funcionamento de redes de comunicação, minimizando custos e maximizando a eficiência da transmissão de dados.
- Detecção de comunidades: em redes sociais, a detecção de comunidades pode ajudar a identificar grupos de usuários com interesses semelhantes, facilitando a segmentação de mercado para campanhas publicitárias.
- Análise de fluxo de informação: a teoria dos grafos permite analisar como as informações se propagam através de redes sociais, ajudando a entender a disseminação de notícias, rumores, ou campanhas virais.



Figura 2 | Redes. Fonte: Pexels.

Vamos Exercitar?

Sua tarefa era desenvolver um algoritmo em Python que identifique os influenciadores com base nas interações entre os usuários na rede.

- Entrada: Um grafo representando a rede social, onde os nós são os usuários e as arestas representam conexões ou interações entre eles.
- Saída: Uma lista dos usuários identificados como influenciadores na rede.

Algoritmo para identificação de influenciadores:

```
class Grafo:  
    def __init__(self):  
        self.adjacencias = {}  
  
    def adicionar_vertice(self, vertice):  
        if vertice not in self.adjacencias:  
            self.adjacencias[vertice] = []  
  
    def adicionar_aresta(self, vertice_origem, vertice_destino):  
        self.adicionar_vertice(vertice_origem)  
        self.adicionar_vertice(vertice_destino)  
        self.adjacencias[vertice_origem].append(vertice_destino)  
  
    def grau_entrada(self, vertice):  
        grau = 0  
        for vizinhos in self.adjacencias.values():  
            if vertice in vizinhos:  
                grau += 1  
        return grau  
  
    def identificar_influenciadores(self):  
        influenciadores = []  
        for usuario in self.adjacencias:  
            grau_entrada_usuario = self.grau_entrada(usuario)  
            if all(grau_entrada_usuario > self.grau_entrada(vizinho) for vizinho in  
self.adjacencias[usuario]):  
                influenciadores.append(usuario)  
        return influenciadores
```

```
# Exemplo de uso do algoritmo  
rede_social = Grafo()  
rede_social.adicionar_aresta('Alice', 'Bob')  
rede_social.adicionar_aresta('Alice', 'Carol')
```

```
rede_social.adicionar_aresta('Bob', 'Alice')
rede_social.adicionar_aresta('Bob', 'Dave')
rede_social.adicionar_aresta('Carol', 'Alice')
rede_social.adicionar_aresta('Carol', 'Dave')
rede_social.adicionar_aresta('Dave', 'Bob')
rede_social.adicionar_aresta('Dave', 'Carol')
rede_social.adicionar_aresta('Dave', 'Eve')
rede_social.adicionar_aresta('Eve', 'Dave')

influenciadores = rede_social.identificar_influenciadores()
print(influenciadores)
```

Saiba mais

Algoritmos de caminhos mais curtos:

- LIMA, E. [Projeto e análise de algoritmos](#). Edirlei Lima – Research Website.

Detecção de ciclos:

- Leitura do capítulo: [CORMEN, T. Algoritmos de Grafos](#). In: CORMEN, T. [Algoritmos - Teoria e Prática](#).

Teoria dos grafos em redes de comunicação e sociais:

- JÚNIOR, I. [Grafos e redes sociais: entenda como esses dois assuntos se relacionam](#). ICMC Júnior.

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

BORIN, V. P. **Estrutura de dados**. 1. ed. São Paulo: Contentus, 2020.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

TAKENAKA, R. M. **Introdução a grafos**. Estrutura de Dados, 2021. Disponível em: <http://bit.ly/497p4MU>. Acesso em: 6 fev. 2024.

Aula 4

Algoritmos para Grafos em Python

Algoritmos para Grafos em Python

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Nesta videoaula, exploraremos os fundamentos e as aplicações dos algoritmos em grafos, focando no Algoritmo de Kruskal e Prim para árvores geradoras mínimas e no Algoritmo de Dijkstra para encontrar o caminho mais curto. Esses conceitos são importantes para a resolução de problemas complexos em diversas áreas da computação e análise de redes, enriquecendo sua base teórica e prática. Junte-se a nós para desvendar essas poderosas ferramentas de análise.

Ponto de Partida

Caro estudante, ao longo do nosso percurso pelo fascinante mundo da teoria dos grafos, descobrimos que esta área da matemática está repleta de oportunidades para investigação e inovação, especialmente no contexto da crescente acessibilidade e capacidade computacional. Esse avanço tecnológico tem sido um catalisador para a validação prática de uma vasta gama de algoritmos e conceitos teóricos.

Nesta aula, iremos mergulhar no estudo de algoritmos aplicados a grafos ponderados e explorar como eles podem ser implementados utilizando a linguagem Python. Um dos exemplos mais palpáveis da aplicação desses algoritmos reside na otimização de rotas terrestres, um problema comum na vida cotidiana de muitos, onde ferramentas de navegação se tornaram quase indispensáveis para motoristas.

Além disso, consideraremos o papel desses algoritmos em projetos de infraestrutura, como na instalação de antenas celulares, construção e manutenção de vias públicas e redes de internet. Mas como esses algoritmos podem ser aplicados em contextos menos grandiosos, como na

distribuição eficiente de eletricidade em uma propriedade rural? Seja ligando moradias, estábulos ou pomares, o objetivo é sempre minimizar os custos associados à instalação de fios elétricos e garantir que a energia chegue a todos os pontos necessários de forma econômica.

Ao longo desta aula, você será introduzido a como calcular a rota mais eficiente entre dois pontos, buscando sempre a solução de menor custo, como já introduzido anteriormente. Os algoritmos que veremos com mais detalhes são fundamentais para a resolução de problemas de otimização, demonstram a aplicabilidade prática e o impacto significativo da teoria dos grafos.

Para praticarmos o que veremos nesta aula, suponhamos que você esteja desenvolvendo um software para uma empresa especializada em paisagismo que atende grandes projetos, como parques públicos, residências de luxo e zoológicos. O objetivo principal dessa ferramenta é otimizar o uso de materiais de pavimentação, reduzindo ao máximo os custos de remodelação dessas áreas vastas.

Para alcançar esse objetivo, o software necessita de dados específicos sobre as dimensões das áreas a serem trabalhadas e as distâncias entre elas. A complexidade da interface gráfica não é uma prioridade neste momento, mas sim a eficácia do algoritmo em calcular a maneira mais econômica de conectar todas as áreas, resultando em uma árvore geradora mínima que indique o caminho de menor custo para a distribuição dos materiais de pavimentação (Takenaka, 2021).

Está preparado para avançar nessa jornada de aprendizado? Vamos lá!

Vamos Começar!

Nesta aula, vamos explorar três algoritmos fundamentais para a manipulação de grafos: Kruskal, Prim e Dijkstra. Enquanto Kruskal e Prim são utilizados para encontrar uma árvore geradora mínima, que conecta todos os pontos (ou vértices) com o menor custo possível, Dijkstra é empregado para determinar o caminho mais curto partindo de um ponto específico.

Para ilustrar a aplicação prática desses algoritmos, consideremos uma situação hipotética em um hotel fazenda, onde os hóspedes ficam acomodados em chalés dispersos pela propriedade. Durante um dia chuvoso, você e seus amigos percebem a ausência de caminhos pavimentados entre os chalés, causando desconforto e sujeira, já que a equipe de limpeza acaba trazendo lama para dentro dos chalés. Diante disso, surge uma discussão curiosa sobre como seria se decidissem pavimentar os caminhos de forma minimalista, mantendo o aspecto natural do local, conforme desejo do proprietário.

A questão proposta na brincadeira é determinar quais caminhos pavimentar para garantir que a equipe de limpeza possa acessar todos os chalés sem interferir excessivamente no ambiente natural. Esse cenário pode ser modelado como um problema de grafos, onde os chalés representam os vértices e os possíveis caminhos entre eles, as arestas. Aqui, a direção dos caminhos é irrelevante, permitindo movimentação livre em qualquer sentido, e o objetivo é conectar todos os chalés com a menor distância total de pavimentação. Para resolver essa

situação hipotética, os algoritmos de Kruskal ou Prim poderiam ser aplicados para identificar a configuração ótima de caminhos a serem pavimentados, enquanto o entendimento das distâncias entre os chalés seria crucial para o cálculo preciso do projeto. Parte superior do formulário.

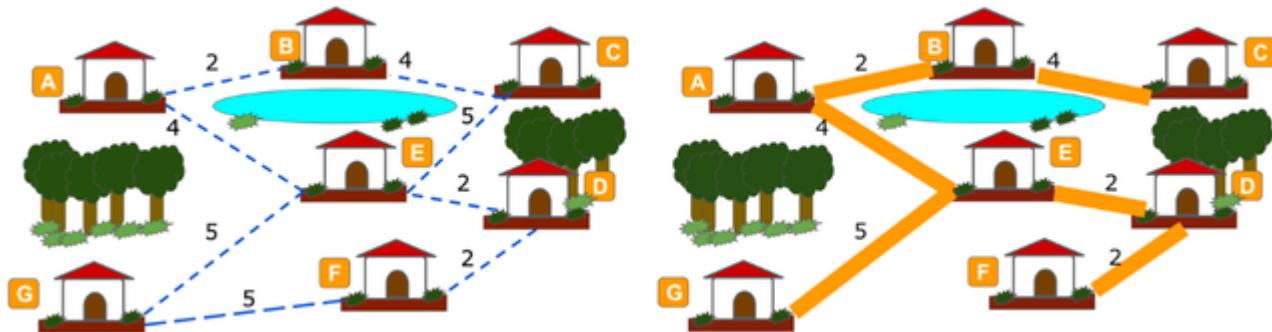


Figura 1 | Conexão de chalés. Fonte: adaptada de Takenaka (2021).

Algoritmo de Kruskal e Prim

Para resolver esse problema, aplicando, por exemplo, o Algoritmo de Kruskal, devemos introduzir o conceito de árvore geradora mínima. Uma árvore geradora é uma estrutura que une todos os vértices de um grafo sem formar ciclos. Em grafos com pesos nas arestas, buscamos a árvore geradora mínima (MST), que é a configuração que conecta todos os vértices com o menor custo total possível. A ideia é selecionar as arestas de menor peso que não criem ciclos até que todos os vértices estejam interligados (Borin, 2020). O algoritmo de Kruskal, desenvolvido por Joseph Kruskal em 1956, é uma metodologia eficaz para encontrar essa árvore geradora mínima, priorizando as arestas de menor peso e evitando a formação de ciclos no processo.

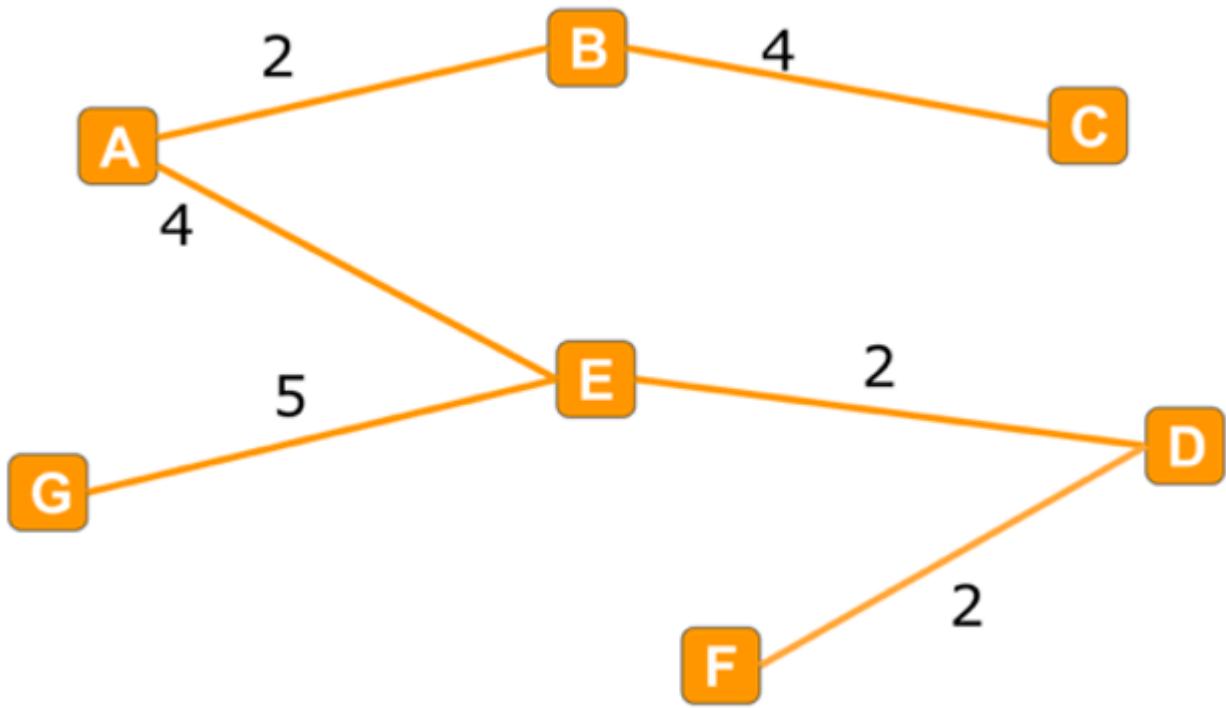
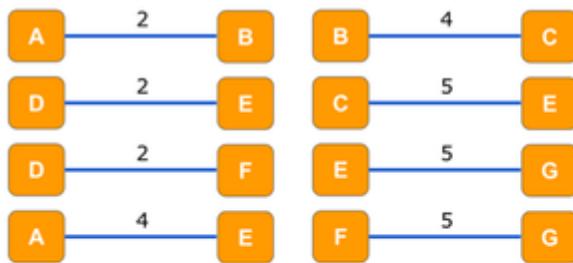


Figura 2 | Árvore geradora de custo mínimo. Fonte: adaptada de Takenaka (2021).



Arestas são postas em ordem crescente de pesos e são selecionadas para a MST.

Exceto a aresta {C,E} por fechar um ciclo.

Exceto a aresta {F,G} por fechar um ciclo.

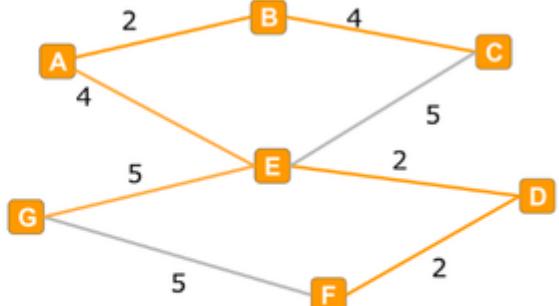


Figura 3 | Algoritmo de Kruskal. Fonte: adaptada de Takenaka (2021).

O pseudocódigo do algoritmo de Kruskal (Figura 4) detalha a formação de uma árvore geradora mínima (MST) a partir de um grafo composto por vértices e arestas. Inicialmente, a MST é representada por um conjunto vazio, que em Python é indicado por `set()`. Em seguida, para cada

vértice do grafo, cria-se um conjunto contendo o próprio vértice. Procede-se então à avaliação de cada aresta, seguindo uma ordem de peso crescente. Para cada aresta avaliada, compara-se os conjuntos dos vértices conectados por ela; se esses conjuntos são disjuntos (ou seja, não compartilham elementos, indicando ausência de intersecção (\emptyset)), conclui-se que a inclusão dessa aresta na MST não formará ciclos (Lambert, 2022). Dessa forma, a aresta é adicionada ao conjunto de arestas da MST e os conjuntos dos vértices ligados por ela são unidos, utilizando a operação de união (*union*).

KRUSKAL(G):

MST = \emptyset

Para cada vértice $v \in G.V$:

CRIE CONJUNTO(v)

Para cada aresta $(u, v) \in G.E$ ordenada pelo peso (u, v) :

se CONJUNTO(u) \neq CONJUNTO(v):

MST = MST $\cup \{(u, v)\}$

UNION(u, v)

retorna MST

Figura 4 | Pseudocódigo Kruskal. Fonte: adaptada de Takenaka (2021).

A seguir, apresentamos um exemplo de implementação do algoritmo de Kruskal em Python:

```
class Grafo:  
    def __init__(self, vertices):  
        self.V = vertices # Número de vértices  
        self.grafo = [] # Lista para armazenar as  
        arestas  
  
    # Adiciona uma aresta ao grafo
```

ESTRUTURA DE DADOS

```
def adicionar_aresta(self, u, v, w):
    self.grafo.append([u, v, w])

    # Função para encontrar o representante do
    # subconjunto de um elemento 'i'
    def encontrar(self, pai, i):
        if pai[i] == i:
            return i
        return self.encontrar(pai, pai[i])

    # Função que realiza a união de dois
    # subconjuntos 'x' e 'y'
    def unir(self, pai, rank, x, y):
        raiz_x = self.encontrar(pai, x)
        raiz_y = self.encontrar(pai, y)

        # União por rank para otimização
        if rank[raiz_x] < rank[raiz_y]:
            pai[raiz_x] = raiz_y
        elif rank[raiz_x] > rank[raiz_y]:
            pai[raiz_y] = raiz_x
        else:
            pai[raiz_y] = raiz_x
            rank[raiz_x] += 1

    # Função principal que roda o algoritmo de
    Kruskal
    def kruskal(self):
        resultado = [] # Guarda a árvore de spanning
        mínima
        i, e = 0, 0 # Índice usado para as arestas
        resultantes e arestas selecionadas,
        respectivamente

        # Ordena todas as arestas em ordem
        crescente de peso
        self.grafo = sorted(self.grafo, key=lambda item:
        item[2])

        pai = []
        rank = []

        # Inicializa os subconjuntos
        for no in range(self.V):
            pai.append(no)
            rank.append(0)
```

```
# Número de arestas resultantes será igual a V-
```

```
1
```

```
while e < self.V - 1:  
    u, v, w = self.grafo[i]  
    i = i + 1  
    x = self.encontrar(pai, u)  
    y = self.encontrar(pai, v)
```

```
# Se a inclusão dessa aresta não causa ciclo,  
adicionamos ela ao resultado
```

```
if x != y:  
    e = e + 1  
    resultado.append([u, v, w])  
    self.unir(pai, rank, x, y)
```

```
# Imprime as arestas selecionadas
```

```
print()  
for u, v, peso in resultado:  
    print( % (u, v, peso))
```

```
# Exemplo de utilização do algoritmo de Kruskal
```

```
g = Grafo(4)  
g.adicionar_aresta(0, 1, 10)  
g.adicionar_aresta(0, 2, 6)  
g.adicionar_aresta(0, 3, 5)  
g.adicionar_aresta(1, 3, 15)  
g.adicionar_aresta(2, 3, 4)  
  
g.kruskal()
```

A classe **Grafo** é utilizada para representar o grafo e contém métodos para adicionar arestas, encontrar e unir subconjuntos, além do algoritmo principal de Kruskal.

No método **kruskal()**, as arestas são ordenadas por peso e, em seguida, são adicionadas ao resultado se não formarem um ciclo.

Os métodos **encontrar()** e **unir()** são usados para encontrar o representante de um subconjunto e para unir dois subconjuntos, respectivamente.

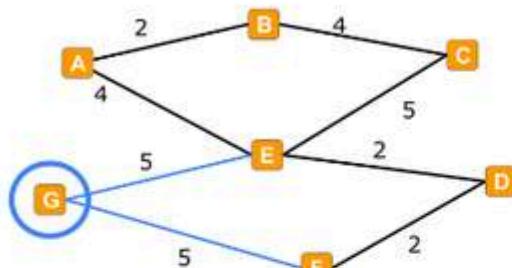
O método **adicionar_aresta()** é utilizado para adicionar uma aresta ao grafo.

O código de exemplo cria um grafo com 4 vértices e 5 arestas e executa o algoritmo de Kruskal sobre ele, imprimindo as arestas selecionadas para a árvore de *spanning* mínima (Alves, 2021).

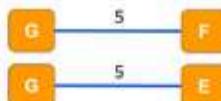
Já o método de Prim é outra estratégia para determinar uma Árvore Geradora Mínima (MST) dentro de um grafo, desenvolvido inicialmente por Vojtěch Jarník no final dos anos 1920 para o cálculo de custos associados à instalação elétrica. Posteriormente, foi redescoberto e popularizado por Robert Clay Prim em 1957, passando a ser conhecido por esse nome. Embora o custo mínimo encontrado por esse algoritmo coincida com o identificado pelo método de Kruskal, as árvores geradas por cada um dos métodos podem variar (Takenaka, 2021).

Para implementar o algoritmo de Prim, inicia-se selecionando arbitrariamente um vértice u do grafo G para atuar como a raiz da MST. A partir desse ponto, e enquanto a MST não englobar todos os vértices de G , a estratégia consiste em escolher, de forma contínua, a aresta de menor custo que conecte um vértice já incluído na MST a um vértice ainda não visitado, expandindo assim a árvore até que todos os vértices sejam abrangidos. Esse processo será exemplificado nas Figuras 5, 6, 7 e 8, simulando os chalés e caminhos dentro de um hotel fazenda, para demonstrar a formação da MST através da seleção criteriosa das arestas baseando-se nos menores custos.

Escolha de um vértice para começar e acrescente em uma fila ordenada por custo as arestas que partem do vértice escolhido {G,E,5} e {G,F,5}



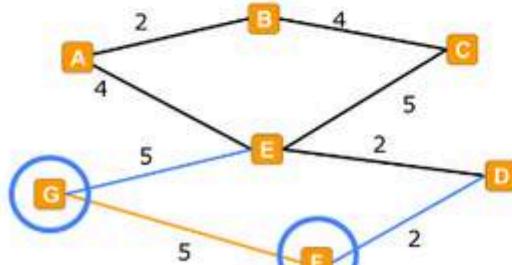
Selecione a aresta de menor custo da fila cuja origem é um vértice da MST e cujo destino é um vértice que não está na MST



Remova {G,F,5} da fila e a inclua na MST



Como agora F faz parte da MST, acrescente na fila ordenada por custo as arestas que partem do vértice F {F,D,2}



Selecione a aresta de menor custo da fila cuja origem é um vértice da MST e cujo destino é um vértice que não está na MST

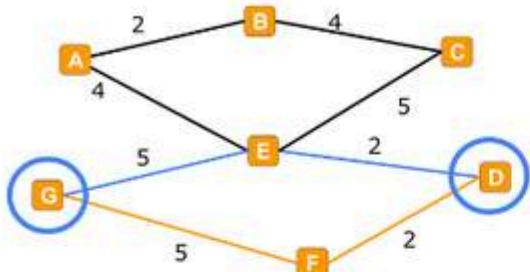


Remova {F,D,2} da fila e a inclua na MST



Figura 5 | Grafo não direcionado ponderado - Parte 1. Fonte: adaptada de Takenaka (2021).

Acrescente na fila ordenada por custo as arestas que partem do vértice D {D,E,2}



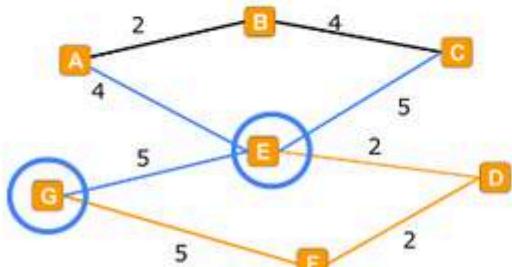
Selecione a aresta de menor custo da fila cuja origem é um vértice da MST e cujo destino é um vértice que não está na MST



Remova {D,E,2} da fila e a inclua na MST



Acrescente na fila ordenada por custo as arestas que partem do vértice E



Selecione a aresta de menor custo da fila cuja origem é um vértice da MST e cujo destino é um vértice que não está na MST



Remova da fila a aresta selecionada e a insira na MST

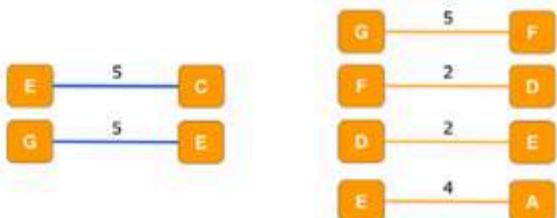
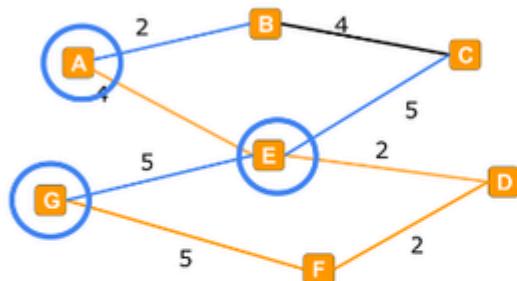
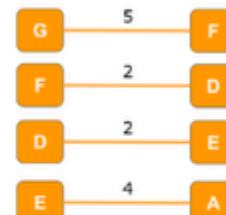


Figura 6 | Grafo não direcionado ponderado - Parte 2. Fonte: adaptada de Takenaka (2021).

Acrescente na fila ordenada por custo as arestas que partem do vértice recém incluído na MST, ou seja, A



Selecione a aresta de menor custo da fila cuja origem é um vértice da MST e cujo destino é um vértice que não está na MST



Remova da fila a aresta selecionada e a insira na MST

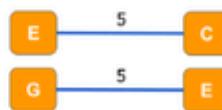
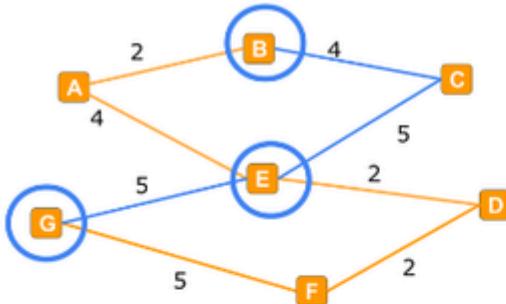
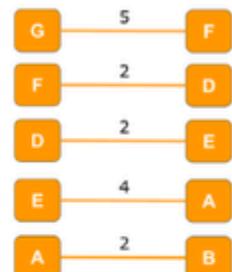


Figura 7 | Grafo não direcionado ponderado - Parte 3. Fonte: adaptada de Takenaka (2021).

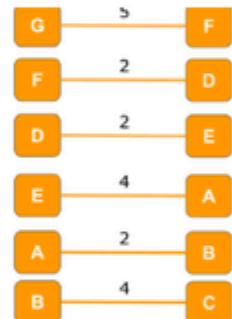
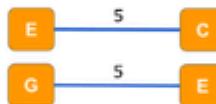
Acrescente na fila ordenada por custo as arestas que partem do vértice recém incluído na MST, ou seja, B



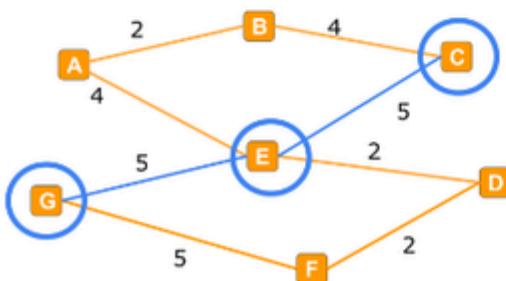
Selecione a aresta de menor custo da fila cuja origem é um vértice da MST e cujo destino é um vértice que não está na MST



Remova da fila a aresta selecionada e a insira na MST



Árvore MST



$5 + 2 + 2 + 4 + 2 + 4 = 19$

Figura 8 | Grafo não direcionado ponderado - Parte 4. Fonte: adaptada de Takenaka (2021).

O pseudocódigo do método de Prim é representado a seguir:

PRIM(G):

MST = {INICIO}

ENQUANTO MST.V ≠ G.V:

aresta (u, v) de menor peso tal que u ∈ MST.V e v ∈ G.V – MST.V

MST = MST ∪ {(u, v)}

retorna MST

Figura 9 | Pseudocódigo de Prim. Fonte: adaptada de Takenaka (2021).

Siga em Frente...

Algoritmos de Dijkstra

Agora, suponhamos que durante a organização de sua viagem ao hotel fazenda você e seus amigos perceberam que, partindo de sua cidade, existiam diversas rotas possíveis para alcançar o destino. Considerando a limitação de tempo disponível, era essencial escolher a rota mais eficiente. Para isso, vocês consultaram o mapa para examinar as opções de trajeto. A jornada começaria no ponto de origem A e concluiria no destino final Z (Takenaka, 2021).

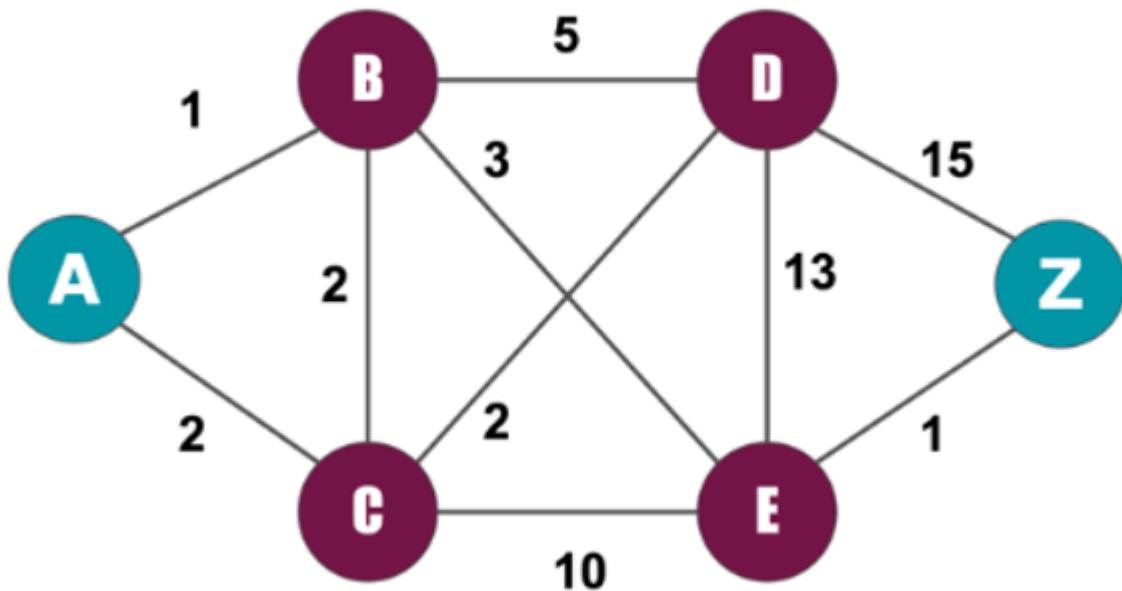


Figura 10 | Grafo que representa uma região com cidades e rotas. Fonte: adaptada de Takenaka (2021).

O algoritmo de Dijkstra, desenvolvido por Edsger Dijkstra em 1959, é projetado para encontrar o caminho de menor custo partindo de um vértice inicial até os demais vértices de um grafo (Szwarcfiter; Markenzon, 2020). Inicialmente, define-se para cada vértice uma estimativa de distância infinita e um antecessor não definido, indicados por $[\emptyset, \emptyset]$, pois ainda não se conhece o caminho até eles. Para o vértice de origem A, ajusta-se esses valores para $[\emptyset, 0]$, indicando que não possui antecessor e sua distância para si mesmo é zero. Todos os vértices são então colocados em uma fila para que o processo de atualização das distâncias e dos antecessores seja realizado de forma iterativa (Takenaka, 2021).

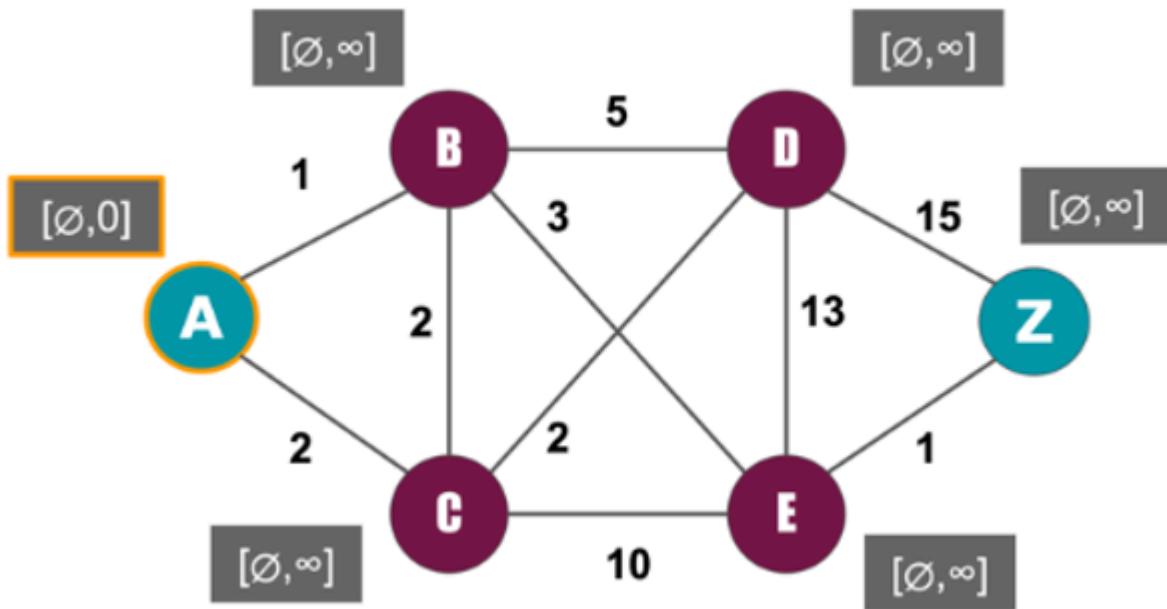


Figura 11 | Algoritmo de Dijkstra – ponto inicial com o antecessor e a distância estabelecidos: $[0,0]$. Fonte: adaptada de Takenaka (2021).

Na sequência do algoritmo de Dijkstra, o primeiro vértice a ser removido da fila é o de menor distância, que neste caso é A, com distância zero, pois todos os outros vértices têm distância inicialmente estimada como infinita. Prosseguindo, analisam-se os vértices adjacentes à A, recalculando as suas distâncias a partir de A. Para cada adjacente, a nova distância é determinada pela soma da distância de A (zero) com o peso da aresta que os conecta. Assim, atualiza-se a distância dos vértices adjacentes à A e define-se A como seu antecessor. Esse processo é repetido para identificar o caminho de menor custo de A para os outros vértices do grafo (Takenaka, 2021).

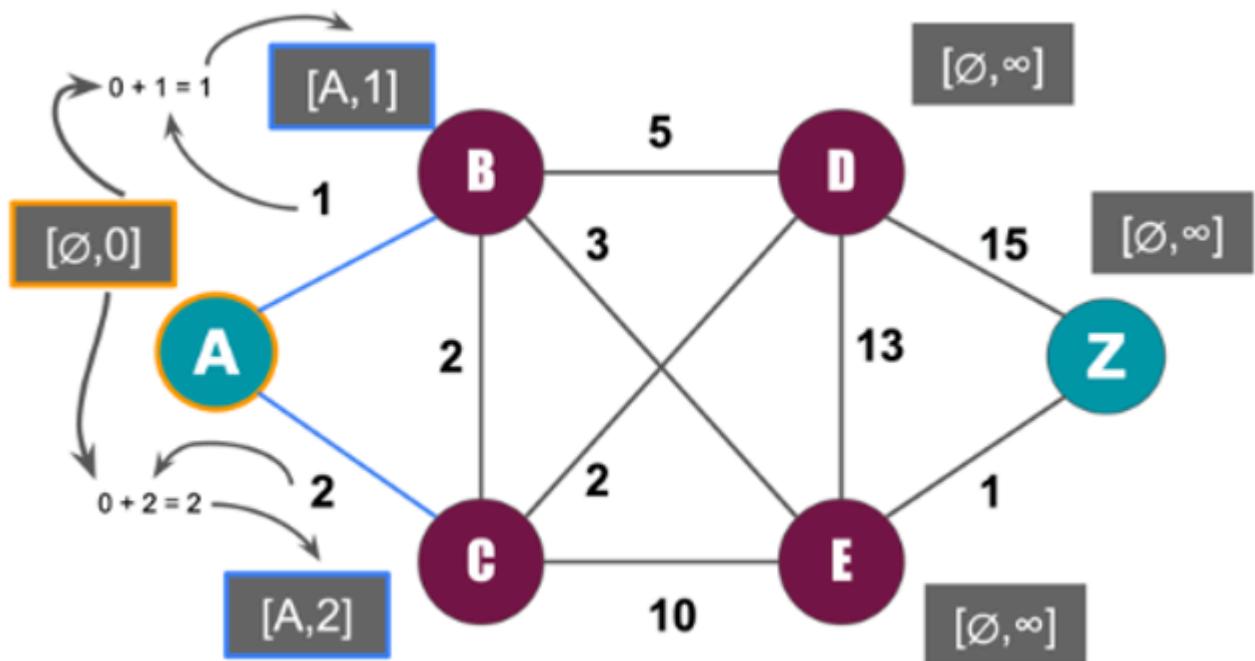


Figura 12 | Algoritmo de Dijkstra – atribuição do antecessor e da distância para B e C. Fonte: adaptada de Takenaka (2021).

Na etapa descrita, o algoritmo de Dijkstra prossegue escolhendo o próximo vértice com a menor distância acumulada, que é B com distância 1. Em seguida, examina-se cada vizinho de B (C e D), calculando a soma da distância acumulada até B com o peso das arestas que conectam B a esses vizinhos. Se essa soma resultar em uma distância menor do que a atualmente registrada para o vizinho, atualizam-se os valores de distância e antecessor para esse vizinho.

Por exemplo, ao analisar a aresta {B, D, 5}, verifica-se que a distância acumulada até D (inicialmente infinita) pode ser reduzida para 6 (1 de B + 5 da aresta), atualizando assim o antecessor de D para B e sua distância para 6. Por outro lado, ao analisar a aresta {B, C, 2}, a distância acumulada até C já era 2 (menor do que a nova distância calculada de 3), portanto, mantém-se a informação de distância e antecessor atual para C. Esse processo é repetido até que todas as distâncias mínimas dos vértices sejam determinadas (Takenaka, 2021).

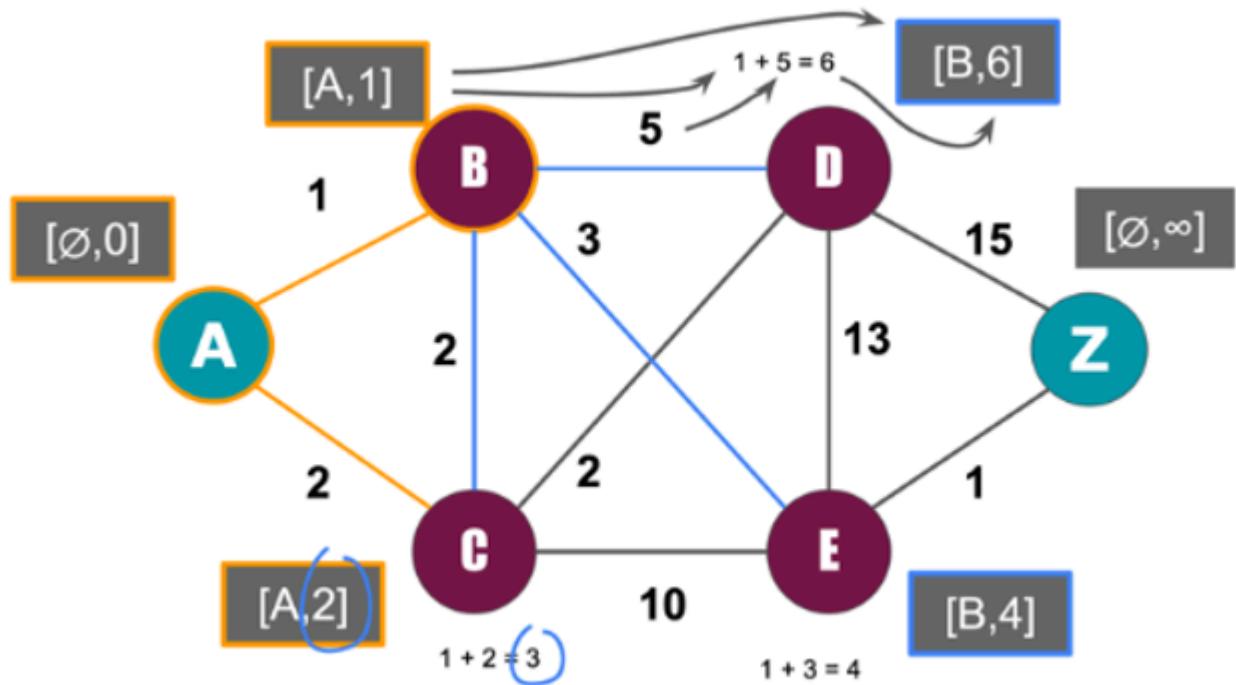


Figura 13 | Algoritmo de Dijkstra – cálculo da soma da distância do antecessor com o custo da aresta. Fonte: adaptada de Takenaka (2021).

Nessa fase do algoritmo de Dijkstra, as distâncias mínimas para os vértices remanescentes na fila são comparadas, identificando-se C como o vértice com a menor distância acumulada (2) partindo de A. Prosseguindo a análise a partir de C, os cálculos das distâncias são atualizados para seus vizinhos, D e E, seguindo as etapas previamente descritas.

Ao examinar os vizinhos de C, observa-se que a distância acumulada para D pode ser reduzida de 6 para 4, atualizando o antecessor de D para C. Essa atualização ocorre porque a distância acumulada até C (2) somada ao peso da aresta conectando C a D resulta em um valor menor do que a distância acumulada anteriormente registrada para D. Quanto ao vértice E, os valores de distância e antecessor se mantêm inalterados, indicando que não houve caminho mais curto encontrado a partir de C até E nessa etapa. Esse processo ilustra como o algoritmo de Dijkstra progride, escolhendo sistematicamente o vértice com a menor distância acumulada e atualizando as distâncias dos vértices adjacentes conforme necessário (Takenaka, 2021).

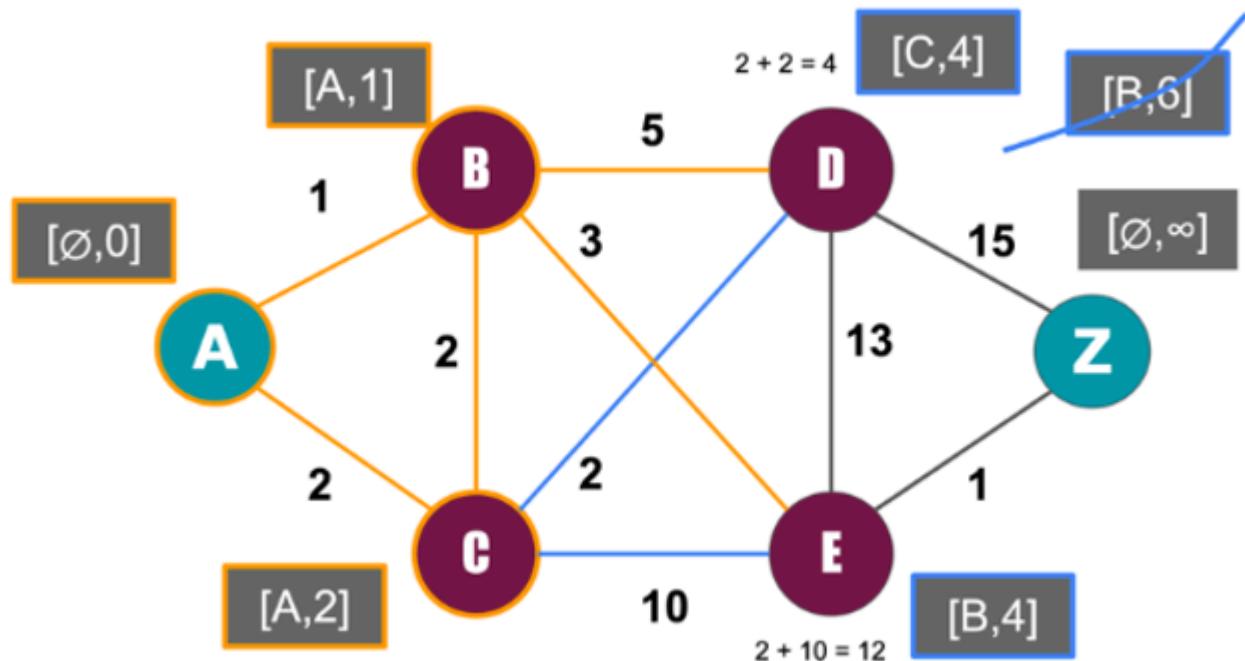


Figura 14 | Algoritmo de Dijkstra – novos valores para D e manter os valores para E. Fonte: adaptada de Takenaka (2021).

O processo prossegue até que todos os vértices tenham sido analisados e suas distâncias mínimas em relação ao ponto de partida A estabelecidas. No final, cada vértice apresenta um valor de distância mínima que indica o caminho mais curto a partir de A, além de seu antecessor nesse caminho. Por exemplo, o trajeto mais curto de A para D tem uma distância total de 4, seguindo a sequência de antecessores de D até A, passando por C. Similarmente, o caminho mais curto de A para Z possui uma distância mínima de 5, traçando o percurso inverso através dos antecessores de Z até A, que passa por E e B. Esse método ilustra como o algoritmo de Dijkstra calcula o caminho mais curto entre um vértice inicial e todos os outros vértices em um grafo (Takenaka, 2021).

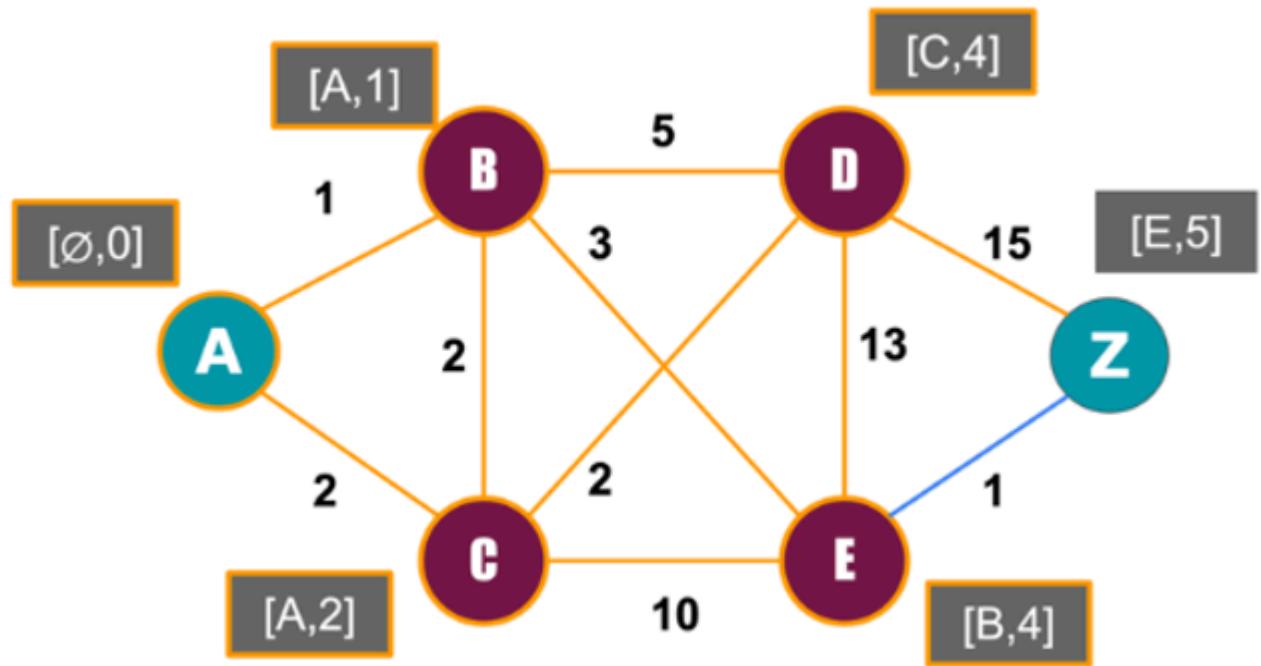


Figura 15 | Algoritmo de Dijkstra - final. Fonte: adaptada de Takenaka (2021).

Por fim, apresentamos o pseudocódigo do algoritmo de Dijkstra:

DIJKSTRA(G, S):

FILA = \emptyset

Para cada vértice v de $G.V$:

$dist[v] = \text{INFINITO}$

$antecessor[v] = \text{INDEFINIDO}$

insere v na FILA

$dist[\text{fonte}] = 0$

Enquanto FILA não está vazia:

u = vértice na fila FILA com a menor distância ($dist[u]$)

remove u da FILA

Para cada vizinho v de u :

$nova_distancia = dist[u] + dist\ de\ (u, v)$

if $nova_distancia < dist[v]$:

$dist[v] = nova_distancia$

$antecessor[v] = u$

retorna $dist[], antecessor[]$

Figura 16 | Pseudocódigo de Dijkstra. Fonte: adaptada de Takenaka (2021).

Por fim, um exemplo da implementação do algoritmo de Dijkstra em Python:

```
import sys

class Grafo:
    def __init__(self, vertices):
        self.V = vertices
        self.grafo = [[0] * vertices for _ in range(vertices)]

    # Função que encontra o vértice com a distância mínima do conjunto de vértices ainda não
    # incluídos no caminho mais curto
    def distancia_minima(self, distancias, visitados):
        minimo = sys.maxsize

        for v in range(self.V):
            if distancias[v] < minimo and visitados[v] == False:
                minimo = distancias[v]
                vertice_minimo = v

        return vertice_minimo

    # Função que imprime o caminho mais curto a partir do vértice inicial para o vértice alvo
    def imprimir_caminho(self, antecessores, j):
        if antecessores[j] == -1:
            print(j, end="")
            return
        self.imprimir_caminho(antecessores, antecessores[j])
        print(j, end="")

    # Função que implementa o algoritmo de Dijkstra
    def dijkstra(self, origem):
        distancias = [sys.maxsize] * self.V
        distancias[origem] = 0
        visitados = [False] * self.V
        antecessores = [-1] * self.V

        for _ in range(self.V):
            u = self.distancia_minima(distancias, visitados)
            visitados[u] = True

            for v in range(self.V):
                if self.grafo[u][v] > 0 and visitados[v] == False and distancias[v] > distancias[u] + self.grafo[u][v]:
                    distancias[v] = distancias[u] + self.grafo[u][v]
                    antecessores[v] = u
```

```
print()
for i in range(self.V):
    print(f'{i}\t{distancias[i]}\t\t', end='')
    self.imprimir_caminho(antecessores, i)
print()
```

```
# Exemplo de utilização do algoritmo de Dijkstra
g = Grafo(9)
g.grafo = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
            [4, 0, 8, 0, 0, 0, 0, 11, 0],
            [0, 8, 0, 7, 0, 4, 0, 0, 2],
            [0, 0, 7, 0, 9, 14, 0, 0, 0],
            [0, 0, 0, 9, 0, 10, 0, 0, 0],
            [0, 0, 4, 14, 10, 0, 2, 0, 0],
            [0, 0, 0, 0, 0, 2, 0, 1, 6],
            [8, 11, 0, 0, 0, 0, 1, 0, 7],
            [0, 0, 2, 0, 0, 0, 6, 7, 0]]]

g.dijkstra(0)
```

O exemplo de utilização cria um grafo com 9 vértices e executa o algoritmo de Dijkstra a partir do vértice 0, imprimindo as distâncias da origem para todos os vértices e os caminhos mais curtos correspondentes.

Vamos Exercitar?

Você foi designado para desenvolver uma funcionalidade em uma aplicação para uma empresa de paisagismo. Essa funcionalidade visa calcular o custo mínimo de materiais de pavimentação para projetos de paisagismo em áreas extensas, como parques públicos, mansões e zoológicos. A empresa busca remodelar essas áreas utilizando a menor quantidade possível de material. Para isso, você pode optar por implementar tanto o algoritmo de Kruskal quanto o de Prim, ambos capazes de encontrar a árvore geradora de custo mínimo. Pratique sua implementação desses algoritmos para contribuir com o projeto.

Exemplo de solução, fornecida por Takenaka (2021):

Algoritmo de Kruskal: Paisagismo

```
class Grafo:
```

ESTRUTURA DE DADOS

```
def __init__(self, orientado=False):
    # dict em que as chaves são os nomes de vértices origem
    # e os valores são os pares de vértice destino e custo
    self._lista_de_adjacencias = dict()
    self.orientado = orientado

@property
def vertices(self):
    return set(self._lista_de_adjacencias.keys())

def arestas(self, v_origem=None):
    if v_origem:
        # obtém arestas do vértice
        return self.varestas(v_origem)

    # retorna arestas do grafo inteiro
    arestas_do_grafo = []
    for v_origem in self.vertices:
        # acumula as arestas de todos os vértices
        arestas_do_grafo += self.varestas(v_origem)
    return arestas_do_grafo

def varestas(self, v_origem):
    # retorna arestas de um vértice
    arestas = []
    for v_destino, custo in self._lista_de_adjacencias[v_origem]:
        arestas.append((v_origem, v_destino, custo))
    return arestas

def inserir_aresta(self, u, v, custo):
    """
    # cria uma chave `u` com valor lista vazia no dicionário,
    # se chave `u` não existir
    self._lista_de_adjacencias.setdefault(u, [])

    # cria uma chave `v` com valor lista vazia no dicionário,
    # se chave `v` não existir
    self._lista_de_adjacencias.setdefault(v, [])

    # adiciona a aresta ao vértice `u`
    self._lista_de_adjacencias[u].append((v, custo))
    if not self.orientado:
```

ESTRUTURA DE DADOS

```
# se é um grafo orientado
# adiciona a aresta ao dicionário do vértice `v`
self._lista_de_adjacencias[v].append((u, custo))

def inserir_arestas(self, arestas):

    for aresta in arestas:
        self.inserir_aresta(*aresta)

def imprimir(self):
    total = 0
    for u, v, custo in self.arestas():
        print(.format(u, v, custo), end=' ')
        total += custo
    if not self.orientado:
        # divide por 2 para descontar a duplicidade
        total = total / 2
    print()
    print(.format(total))

#72
def kruskal(grafo):
    # cria conjunto de arestas da MST
    arestas_mst = set()

    # cria um dicionário para armazenar
    # o conjunto de vértices para cada vértice
    # que inicialmente é o próprio vértice
    conjuntos = {v: {v} for v in grafo.vertices}

    # ordena todas as arestas do grafo
    arestas_ordenadas = sorted(
        grafo.arestas(),
        key=lambda aresta: aresta[-1]
    )
    # para cada aresta, em ordem crescente
    for aresta in arestas_ordenadas:
        # obtém u, v e custo, da aresta
        u, v, custo = aresta
        if conjuntos[u].isdisjoint(conjuntos[v]):
            # os conjuntos de u e o de v são conjuntos disjuntos
            # adiciona aresta ao conjunto MST
            arestas_mst.add(aresta)
            # une os conjuntos v e u
            uniao = conjuntos[u] | conjuntos[v]
            # conjunto de u, conterá conjunto de v
```

ESTRUTURA DE DADOS

```
conjuntos[u] = uniao  
# conjunto de v, conterá conjunto de u  
conjuntos[v] = uniao
```

```
mst = Grafo()  
mst.inserir_arestas(arestas_mst)  
return mst
```

```
#106  
def solicitar_locais():  
    qtd = int(input())  
    locais = []  
    for i in range(qtd):  
        local = input()  
        locais.append(local)  
    return locais
```

```
def solicitar_distancias(locais):  
    from itertools import combinations  
    distancias = []  
    for u, v in combinations(locais, 2):  
        distancia = input(  
            .format(  
                u, v))  
        distancias.append((u, v,  
                           float(distancia)))  
    return distancias
```

```
print()  
arestas = [  
    ('Parquinho', 'Lago', 2.2),  
    ('Quadras', 'Lago', 5.5),  
    ('Parquinho', 'Quadras', 8.6),  
    ('Lanchonete', 'Lago', 6),  
]  
g1 = Grafo()  
g1.inserir_arestas(arestas)  
mst = kruskal(g1)  
mst.imprimir()  
  
print()  
print()  
locais = solicitar_locais()
```

```
arestas = solicitar_distancias(locais)
g2 = Grafo()
g2.inserir_arestas(arestas)
mst = kruskal(g2)
mst.imprimir()
```

Saiba mais

Algoritmos em grafos:

- Leitura do capítulo: CORMEN, T. [Algoritmos de Grafos](#). In: CORMEN, T. **Algoritmos - Teoria e Prática**.

Algoritmo de Kruskal e Prim:

- MORRISON, R. [Diferença entre Kruskal e Prim](#). **Strephonsays**.

Algoritmos de Dijkstra:

- FELIPPE, G. [O Algoritmo de Dijkstra](#). **Akira Dev**.

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

BORIN, V. P. **Estrutura de dados**. 1. ed. São Paulo: Contentus, 2020.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

TAKENAKA, R. M. **Introdução a grafos**. Estrutura de Dados, 2021. Disponível em: <http://bit.ly/497p4MU>. Acesso em: 6 fev. 2024.

Aula 5

GRAFOS E SUAS OPERAÇÕES

Videoaula de Encerramento



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Nesta videoaula, faremos um resumo e encerramento da unidade sobre grafos, abrangendo desde a introdução até operações com grafos, caminhos e ciclos, além de algoritmos para grafos em Python. Esse conteúdo é fundamental para quem deseja aprimorar suas habilidades em estruturas de dados e algoritmos, essenciais para resolver problemas complexos em diversas áreas da computação e análise de dados. Convidamos você a assistir a esta aula para consolidar seu aprendizado e preparar-se para aplicar esses conhecimentos em sua prática profissional.

Ponto de Chegada

Olá, estudante!

Para desenvolver a competência desta Unidade, que é conhecer e compreender as estruturas de dados do tipo grafos e como representá-los graficamente, é essencial que você se familiarize com os conceitos fundamentais de grafos, explorando as operações básicas que podem ser realizadas com eles, bem como os caminhos e ciclos que podem ser identificados dentro de uma estrutura de grafo. Adicionalmente, você irá aplicar seu conhecimento desenvolvendo algoritmos para grafos em Python, consolidando o aprendizado teórico através de prática de programação.

Durante a unidade, iniciamos com uma introdução a grafos, estabelecendo a base para o seu entendimento sobre o que são vértices e arestas e como esses elementos se combinam para formar diferentes tipos de grafos, como dirigidos e não dirigidos, ponderados e não ponderados. Compreender essas diferenças é importante para a aplicação correta dos grafos em problemas variados (Takenaka, 2021).

Avançamos para as operações em grafos, onde você aprende a realizar tarefas fundamentais como a inserção e remoção de vértices e arestas, além de entender conceitos importantes como

grau de um vértice, subgrafos e grafos complementares. Essas operações são a base para manipulação de estruturas de grafos em diversas aplicações.

Na sequência, exploramos caminhos e ciclos, elementos essenciais para a compreensão de como navegar dentro de um grafo. Aprender a identificar caminhos, ciclos e componentes conectados permite que você analise a estrutura de um grafo de maneira mais profunda, identificando possíveis problemas ou otimizações em redes de comunicação, logística e outros sistemas (Lambert, 2022).

Finalmente, na parte de algoritmos para grafos em Python, você tem a oportunidade de colocar em prática os conceitos aprendidos, implementando algoritmos como a busca em profundidade (DFS), busca em largura (BFS), Dijkstra, Prim e Kruskal. A aplicação prática desses algoritmos consolida seu entendimento sobre a teoria dos grafos e abre portas para solução de problemas complexos com eficiência (Borin, 2020).

É Hora de Praticar!



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Uma empresa de logística deseja otimizar as rotas de entrega de seus veículos para garantir a eficiência e reduzir custos operacionais. A empresa opera em uma cidade onde as entregas são feitas a partir de um depósito central para vários pontos de entrega espalhados pela cidade. O objetivo é encontrar a rota mais curta que permita ao veículo sair do depósito, realizar todas as entregas e retornar ao depósito com o menor percurso possível.

Dados:

O depósito central é representado pelo vértice **D**.

Os pontos de entrega são representados pelos vértices **A, B, C, E, e F**.

As distâncias entre os pontos são representadas em uma matriz de adjacência:

Caminhos para chegar a cada ponto a partir de 'D':

Para 'A': Início em 'D'

Para 'B': Começa em 'D', passa por 'A'

Para 'C': Começa em 'D', passa por 'A'

Para 'E': Começa em 'D', passa por 'A', depois 'B'

Para 'F': Começa em 'D', passa por 'A', 'B', e então 'E'

Desafio:

Usar o Algoritmo de Dijkstra para encontrar a rota mais curta do depósito **D** para todos os pontos de entrega, considerando que o veículo precisa retornar ao depósito após realizar todas as

ESTRUTURA DE DADOS

entregas.

Para refletir sobre o conteúdo abordado, considere as seguintes perguntas:

1. Como a escolha entre um grafo dirigido e um não dirigido pode impactar a solução de um problema específico?
2. De que maneira os algoritmos de busca em grafos, como DFS e BFS, podem ser aplicados em situações reais, como em sistemas de recomendação ou na otimização de rotas?
3. Qual a importância de algoritmos como Dijkstra, Prim e Kruskal na resolução de problemas práticos e como você poderia aplicá-los em um projeto pessoal ou profissional?

Ao longo desta unidade, ao dominar os fundamentos e a aplicação prática dos grafos, você desenvolve uma competência valiosa que ampliará suas habilidades na resolução de problemas complexos, tanto no âmbito acadêmico quanto profissional.

Segue um exemplo de como poderíamos implementar essa solução em Python:

```
# Definindo a matriz de adjacência que representa o grafo
# 0 indica que não há caminho direto entre os vértices
grafo = {
    'D': {'A': 2, 'B': 4},
    'A': {'D': 2, 'C': 1, 'B': 3},
    'B': {'A': 3, 'D': 4, 'E': 2},
    'C': {'A': 1, 'F': 4},
    'E': {'B': 2, 'F': 3},
    'F': {'C': 4, 'E': 3}
}

def dijkstra(grafo, inicio):
    menor_distancia = {}
    antecessor = {}
    nao_visitados = grafo
    infinito = float('inf')
    caminho = []

    for vertice in nao_visitados:
        menor_distancia[vertice] = infinito
    menor_distancia[inicio] = 0

    while nao_visitados:
        vertice_minimo = None
        for vertice in nao_visitados:
            if vertice_minimo is None:
                vertice_minimo = vertice
            elif menor_distancia[vertice] < menor_distancia[vertice_minimo]:
                vertice_minimo = vertice

        for vizinho in grafo[vertice_minimo]:
            distancia = menor_distancia[vertice_minimo] + grafo[vertice_minimo][vizinho]
            if distancia < menor_distancia[vizinho]:
                menor_distancia[vizinho] = distancia
                antecessor[vizinho] = vertice_minimo
```

ESTRUTURA DE DADOS

```
for vertice_adjacente, peso in grafo[vertice_minimo].items():
    if peso + menor_distancia[vertice_minimo] < menor_distancia[vertice_adjacente]:
        menor_distancia[vertice_adjacente] = peso + menor_distancia[vertice_minimo]
        antecessor[vertice_adjacente] = vertice_minimo
    nao_visitados.pop(vertice_minimo)

vertice_atual = inicio
while vertice_atual != None:
    caminho.append(vertice_atual)
    vertice_atual = antecessor.get(vertice_atual, None)

return menor_distancia, caminho

distancias, caminho = dijkstra(grafo, 'D')
print(distancias)
print(caminho)
```

Inicialização: Define-se a matriz de adjacência do grafo e inicializa-se o algoritmo com distâncias infinitas para todos os vértices, exceto o ponto de partida (D), que recebe distância 0.

Processamento: O algoritmo itera sobre os vértices não visitados, selecionando o vértice com a menor distância acumulada em cada passo. Atualiza-se a menor distância para os vértices adjacentes, baseando-se nas distâncias dos vértices já processados.

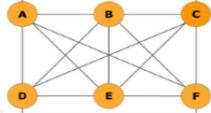
Resultado: O algoritmo retorna a menor distância de D para todos os outros vértices e o caminho a partir do depósito, permitindo à empresa de logística otimizar suas rotas de entrega.

Este infográfico é uma ferramenta didática que pode ajudar a visualizar e compreender melhor os conceitos abordados na unidade:

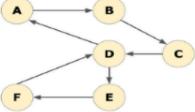
Tipos de Grafos

Grafo não direcionado

Um grafo não direcionado desconexo é utilizado para descrever cenários em que alguns elementos do grafo não estão interligados entre si.



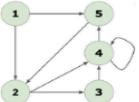
Grafo orientado



Um grafo orientado, também conhecido como grafo dirigido, direcionado ou digrafo, caracteriza-se por apresentar relações assimétricas entre seus vértices.

Grafo orientado com ciclo

Um ciclo Hamiltoniano é uma trilha que não repete os vértices, exceto o início e o fim, passando por todos os vértices, enquanto um grafo Hamiltoniano é aquele que permite um ciclo Hamiltoniano.



Grafo topológico



É um grafo acíclico usado para modelar sequências de tarefas e suas dependências

Matriz de adjacência

Uma das representações numéricas é a matriz de adjacência, que é uma representação computacional compreensível pelas linguagens de programação.

	A	B	C	D	E	F
A	0	1	1	1	1	1
B	1	0	1	1	1	1
C	1	1	0	1	1	1
D	1	1	1	0	1	1
E	1	1	1	1	0	1
F	1	1	1	1	1	0

Lista de adjacência

A	B	C	D	E
A → B	B → C	C → D	D → E	
B → C	C → D	D → E	E → B	
C → D	D → E	E → C		
D → E	E → B			
E → B				

Representação de todas arestas ou arcos de um grafo em uma lista.

BORIN, V. P. **Estrutura de dados**. 1. ed. São Paulo: Contentus, 2020.

LAMBERT, K. A. **Fundamentos de Python: estruturas de dados**. São Paulo: Cengage Learning, 2022.

TAKENAKA, R. M. **Introdução a grafos**. Estrutura de Dados, 2021. Disponível em: <http://bit.ly/497p4MU>. Acesso em: 6 fev. 2024.

Unidade 4

ESTRUTURAS DE DADOS AVANÇADAS E ANÁLISE DE DADOS

Aula 1

Maps e Hash

Maps e Hash



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Nesta videoaula, mergulharemos nos conceitos fundamentais de *maps* e *hash map*, explorando a dinâmica chave-valor e a estrutura das tabelas *hash*. Você aprenderá como essas estruturas otimizam a busca e o armazenamento de dados, uma habilidade essencial para qualquer desenvolvedor. Além disso, demonstraremos a implementação prática de tabelas *hash* em Python, preparando você para aplicar esses conceitos em soluções reais. Junte-se a nós para aprimorar suas habilidades de programação e entender por que essas técnicas são importantes na computação.

Bons estudos!

Ponto de Partida

Olá, estudante!

Desejamos-lhes boas-vindas à aula sobre tabelas *hash*, uma estrutura de dados fundamental para a eficiência computacional, tanto na recuperação de informações quanto na segurança de

ESTRUTURA DE DADOS

dados através de algoritmos de criptografia. As tabelas de espalhamento são projetadas para otimizar a inserção, remoção e, sobretudo, a recuperação rápida de dados.

Nesta aula, iremos desvendar como funcionam as tabelas *hash*, desde a implementação até o cálculo da posição dos elementos com a função *hash*, explorando a relação chave-valor que define essa estrutura. Além disso, iremos aplicar esses conceitos em Python, demonstrando na prática como essa estrutura pode mapear entradas para posições específicas de um vetor com eficiência surpreendente. Prepare-se para aprofundar seus conhecimentos sobre uma das estruturas de dados que tornam possíveis operações rápidas e eficientes, essenciais para profissionais da tecnologia da informação.

Para assimilar o conteúdo desta aula, suponhamos que você é um desenvolvedor de software na equipe de TI de uma empresa que produz acessórios veiculares. Você enfrentará o desafio de modernizar o sistema de controle de fabricação e armazenamento. A empresa categoriza seus variados produtos em 15 classes distintas, com cada produto marcado por um identificador único de 10 dígitos, onde os dois últimos dígitos indicam sua classe. Com o aumento da produção, a necessidade de um método automatizado para gerenciar o inventário tornou-se essencial. Seu primeiro projeto envolve criar um software capaz de contar automaticamente os produtos em estoque por classe, além de permitir a consulta da quantidade de itens de uma classe específica através do código de identificação do produto. Essa solução deve oferecer uma recuperação de dados eficiente, agilizando o processo de contagem e consulta de produtos armazenados.

O programa que você desenvolverá em Python utilizará uma estrutura de dados otimizada para o rápido acesso às informações de cada classe de produto. O foco estará em incrementar a contagem de produtos por classe quando novos itens são adicionados ao estoque e em fornecer uma maneira simples de consultar o número de itens em uma classe específica, baseando-se no identificador do produto. Essa implementação não apenas facilitará a gestão do inventário como também suportará o crescimento contínuo da produção, substituindo o método manual anterior por um sistema informatizado, eficiente e escalável.

Preparado para aprender mais sobre essa estrutura de dados?

Então, bons estudos!

Vamos Começar!

Conceitos de *maps* e *hash map*: chave-valor

As estruturas de dados são fundamentais no desenvolvimento de algoritmos, pois permitem a resolução de uma ampla gama de problemas computacionais que, sem elas, seriam intransponíveis. Cada estrutura de dados tem suas próprias vantagens e limitações, o que influencia diretamente a escolha de qual estrutura utilizar em determinadas situações. Um dos principais desafios ao empregar estruturas de dados para o armazenamento de informações é a

gestão eficiente de grandes volumes de dados, que podem alcançar milhões ou até bilhões de registros. Sem uma estratégia adequada para a recuperação desses dados, o processo de busca pode se tornar extremamente lento e ineficiente, aumentando significativamente o tempo de execução das operações computacionais.

Para ilustrar o desafio do armazenamento eficiente, imagine uma estrutura destinada a conter todas as palavras formáveis em um idioma. Dada a vastidão de possibilidades, localizar uma palavra específica, como "caderno", em uma estrutura linear ou hierárquica, pode ser excessivamente demorado. Isso se deve à necessidade de percorrer a estrutura até encontrar a palavra, sem saber sua posição prévia. Mesmo com algoritmos de busca rápidos, o processo torna-se lento com o aumento do volume de dados, pois esses algoritmos perdem eficiência à medida que o conjunto de dados cresce.

Esse problema evidencia a limitação de estruturas de dados que não associam o conteúdo ao seu índice de forma direta. Uma solução promissora para essa questão é o uso de **tabelas hash**, que oferecem vantagens significativas na inserção e remoção de dados, apesar de desafios como colisões, onde dois itens distintos são mapeados para a mesma posição.

Tabelas *hash*

As tabelas *hash* operam pelo princípio de mapear entradas a posições específicas numa tabela, utilizando funções *hash* para determinar esse mapeamento. Esse processo pode ser realizado por funções *hash* padrão ou funções *map* disponíveis em algumas linguagens e bibliotecas, não se limitando estritamente a implementações de *hash*. Por exemplo, armazenar os números 235, 578, 1024, 96, 32 com uma função *hash* baseada no módulo 10 resultaria em cada número sendo posicionado em um índice correspondente ao resto da sua divisão por 10. Assim, o número 32 seria colocado no índice 2, 235 no índice 5, e assim sucessivamente, demonstrando como os índices na tabela *hash* são definidos pelo resultado dessa divisão (Figura 1).

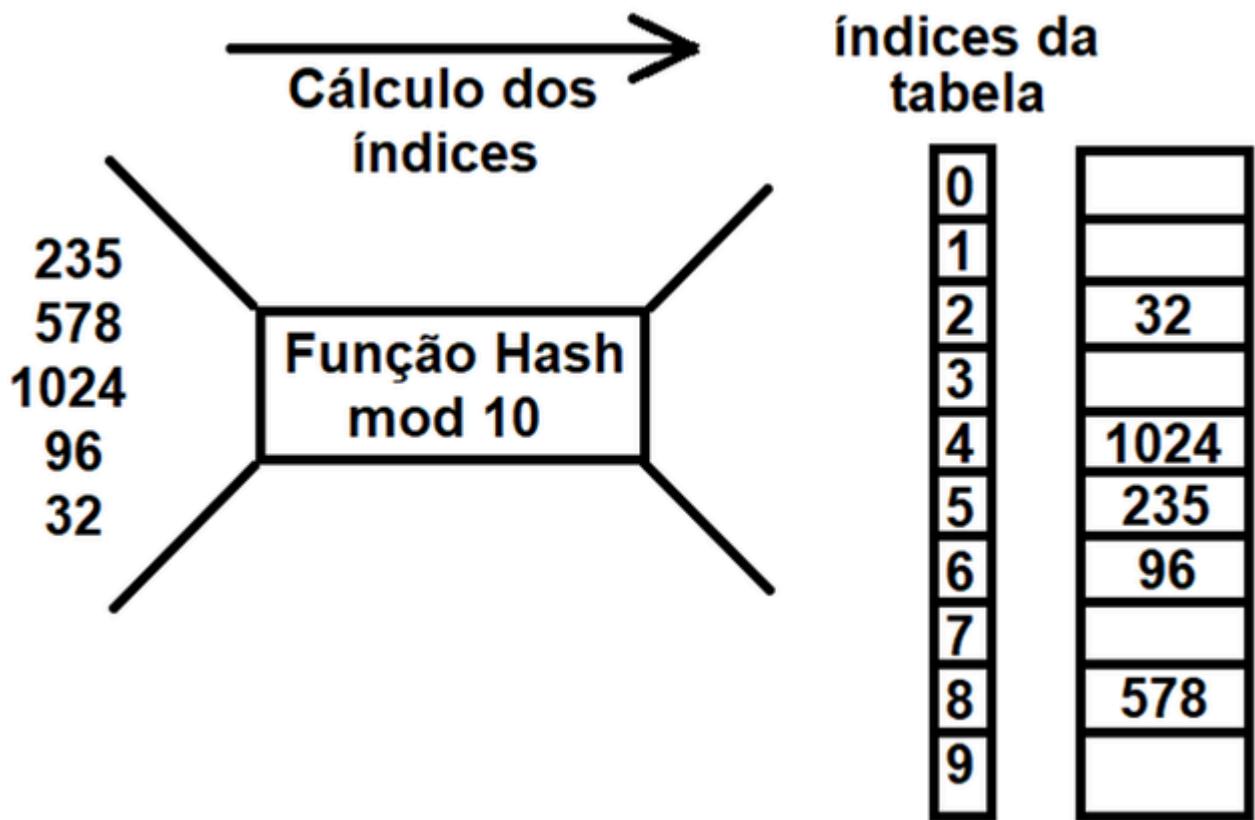


Figura 1 | Exemplo de utilização de função hash. Fonte: adaptada de Dias (2021).

Imagine que você precisa localizar o número 1024 em um vetor onde os números estão dispostos aleatoriamente. Na pior das hipóteses, você teria que examinar cada posição do vetor até encontrar o número desejado. Embora isso possa ser gerenciável em vetores de tamanho reduzido, em vetores maiores, o tempo computacional necessário para realizar múltiplas buscas torna-se proibitivamente alto.

A figura anterior (Figura 1) serve para ilustrar como funciona o *hash*, mas sua aplicabilidade prática é limitada. Utilizar um vetor ordenado poderia ser mais adequado simplesmente para armazenar números. Então, em que situações o *hash* se torna realmente útil? Geralmente, o *hash* é utilizado quando os dados armazenados consistem em um conjunto de informações e uma dessas informações serve para gerar a chave *hash*, idealmente algo único. Isso significa que o valor de retorno da função *hash* é determinado pela entrada específica.

Tomemos como exemplo um sistema de cadastro de clientes, um projeto típico na área de computação. Os dados dos clientes podem ser eficientemente armazenados usando uma tabela *hash*, otimizando o gerenciamento das informações. Utilizando o CPF do cliente para gerar uma chave *hash*, a recuperação dos dados desse cliente torna-se direta e rápida. Ao fornecer o CPF, recalcula-se a chave com a função *hash*, localizando assim onde os dados do cliente estão armazenados. Esse método é conhecido como chave-valor, onde a chave é gerada a partir de um valor de entrada específico para a função *hash*.

CHAVE - VALOR

VALOR: 12,345,678,935 (CPF)

FUNÇÃO HASH: MOD 10

CHAVE: 5 (RESTO DA DIVISÃO CPF POR 10)

Figura 2 | Exemplo de conceito chave-valor. Fonte: adaptada de Dias (2021).

Antes de avançarmos, é importante compreender como funciona uma tabela *hash*. Essa estrutura armazena valores em posições definidas por chaves, que são determinadas aplicando-se o valor à função *hash* específica da tabela, resultando na chave de armazenamento. Esse processo otimiza operações, reduzindo o tempo computacional e podendo também economizar memória. A escolha da função *hash* é essencial, visto que o desempenho da busca e a eficiência da tabela dependem dela, não sendo afetados pelo tamanho da tabela, mas sim pela eficácia da função *hash* e pelo manejo de colisões, que ocorrem quando valores distintos resultam na mesma chave.

Por exemplo, se adicionarmos o número 18 à tabela e ele gerar a chave 8, onde o número 578 já está armazenado, enfrentamos uma **colisão**. Uma maneira de resolver isso é transformar cada espaço de armazenamento em uma lista, adicionando elementos conforme ocorrem colisões. Esse método, embora resolva o problema das colisões, pode aumentar o tempo de busca se muitos elementos forem mapeados para a mesma chave, formando listas longas que exigem busca sequencial. Portanto, a gestão eficaz das colisões é fundamental para manter a eficiência da recuperação de informações em uma tabela *hash*.

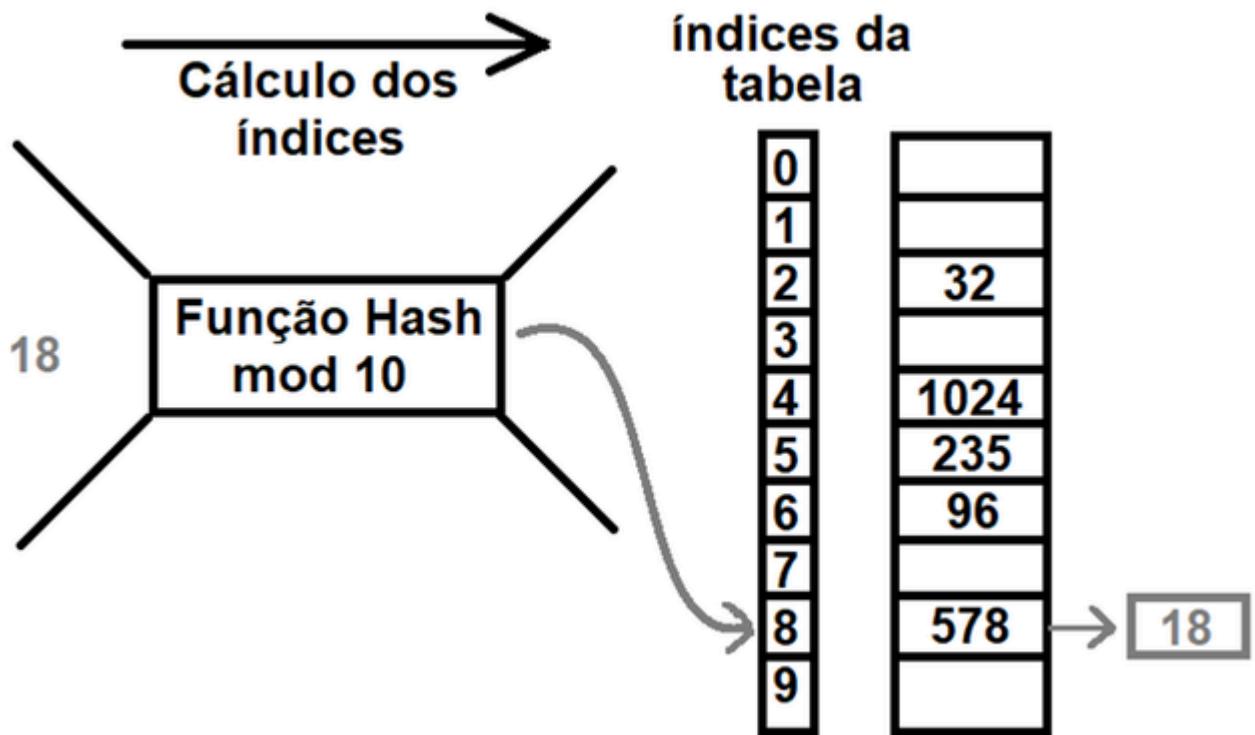


Figura 3 | Tratando colisões na tabela hash. Fonte: adaptada de Dias (2021).

Para minimizar colisões em uma tabela *hash*, é essencial escolher uma função *hash* adequada. O objetivo é alcançar uma distribuição uniforme, onde cada chave tem igual probabilidade de ser selecionada, idealmente distribuindo a chance de alocação para um elemento em 10% por chave, se tivermos 10 elementos, exemplificando um *hash* uniforme simples. Alcançar uma função *hash* perfeita, que distribua as entradas uniformemente sem colisões, é desafiador. Portanto, entender e aplicar funções *hash* eficazes é essencial.

Uma abordagem é o método da divisão, onde a chave '*K*' é mapeada para uma das '*m*' posições usando a seguinte fórmula:

$$h(k) = k \bmod m$$

$$h(k) = k \bmod m$$

Onde ***mod*** representa o módulo. Escolher ***m*** como um número primo não próximo de uma potência de 2 ajuda a evitar padrões previsíveis de colisão.

Outra técnica é o método da multiplicação, definido por:

$$h(k) = m((k * A) \bmod 1)$$

Onde o resultado é baseado na parte fracionária da multiplicação de '*K*' por uma constante '*A*' entre 0 e 1, com '*m*' sendo o número de posições desejadas. Nesse método, a escolha de '*m*' é

menos crítica, mas a constante ' A ' é essencial, com valores como '0,618034' mostrando-se eficazes. Ambos os métodos visam uma distribuição equilibrada das chaves pela tabela, reduzindo o impacto das colisões na eficiência da recuperação de dados.

Siga em Frente...

Implementação de tabelas *hash* em Python

Entendendo a importância das funções *hash* e seu impacto no desempenho, é essencial escolher a função adequada, uma tarefa que pode ser desafiadora devido à complexidade de gerar chaves distribuídas uniformemente, especialmente sem conhecer previamente os dados a serem armazenados. Ao lidar com um conjunto hipotético de 1000 números, a escolha entre o método da divisão e o da multiplicação para a função *hash* requer análise cuidadosa. O método da divisão pode ser atrativo, mas sua eficácia depende da seleção de um número primo como divisor. Por outro lado, o método da multiplicação, menos sensível à escolha de m , depende da constante A para otimizar a distribuição das chaves.

A ideia de utilizar um conjunto dinâmico de funções *hash* sugere uma abordagem mais flexível e potencialmente mais eficaz em distribuir uniformemente as chaves, minimizando colisões. No Python, dicionários empregam *hashing*, servindo como base para implementações de tabelas *hash* eficientes. Explorar um exemplo de código que simula a tabela *hash* e ilustra o manejo (ou a falta dele) de colisões pode enriquecer o entendimento sobre as funções *hash* em contextos práticos, destacando as considerações necessárias para otimizar o armazenamento e a recuperação de dados, portanto observe o código a seguir:

```
# Parte 1: Inicialização da tabela hash como
# um dicionário vazio
 hashtable = {}
 m = 10 # Define o número de índices (ou
 #'buckets') na tabela hash

# Parte 2: Definição da função hash
def hashfunct(v, mh):
    # v é o valor a ser armazenado e mh é o
    # tamanho da tabela hash (m)
    return v % mh # Retorna o índice para
    # armazenamento usando o método da divisão

# Parte 3: Preenchimento inicial da tabela
# hash
for i in range(m):
    hashtable[i] = '' # Inicializa cada índice da
    # tabela com um espaço vazio
```

```
# Parte 4: Inserção de valores na tabela hash
hashtable[hashfunct(235, m)] = '235' # Insere
o valor '235' no índice calculado pela função
hash
hashtable[hashfunct(578, m)] = '578'
hashtable[hashfunct(1024, m)] = '1024'
hashtable[hashfunct(96, m)] = '96'
hashtable[hashfunct(32, m)] = '32'

# Exibe o estado atual da tabela hash após as
inserções
print(hashtable)

# Parte 5: Inserção de um valor adicional sem
tratamento de colisões
hashtable[hashfunct(18, m)] = '18' # Insere
'18', substituindo qualquer valor previamente
armazenado no mesmo índice

# Exibe o estado final da tabela hash,
mostrando o resultado da inserção sem
tratamento de colisões
print(hashtable)
```

O algoritmo apresentado inicia criando uma tabela *hash* como um dicionário vazio e estabelece o número de posições de forma paramétrica (Parte 1), uma escolha que permite flexibilidade. A função *hash* (Parte 2) calcula o índice para armazenamento dos dados usando o método da divisão pelo número 10, preparando o terreno para a inicialização dos índices da tabela com valores vazios (Parte 3). Os valores são então inseridos (Parte 4) usando essa função *hash*, que determina a posição específica de cada elemento na tabela.

A visualização do *hash* reflete o layout apresentado na Figura 1. Contudo, ao introduzir um novo valor que resulta em colisão (Parte 5), o algoritmo não dispõe de um mecanismo para tratá-la, podendo sobreescriver dados existentes sem alerta, como ilustrado na Figura 3. Isso destaca a necessidade de gerenciamento de colisões em tabelas *hash* para evitar a perda de informações. Para aprimorar o algoritmo, são necessárias duas modificações fundamentais: **detectar e tratar colisões**. A detecção de colisões é direta: se a chave específica no dicionário já contiver um valor, identifica-se a ocorrência de uma colisão.

```
# Parte 1: Inicialização e definição da função
hash
1 hashtable = {} # Cria um dicionário vazio
para a tabela hash.
2 m = 10 # Define o número de índices
disponíveis na tabela hash.
```

```
4 def hashfunct(v, mh): # Define a função
hash usando o método da divisão.
5   return v % mh # Retorna o índice baseado
no resto da divisão do valor pelo número total
de índices (m).

# Parte 2: Definição da função de inserção
com detecção de colisões
7 def insereTC(valor): # Define a função para
inserir valores na tabela hash.
8   if (hashtable[hashfunct(valor,m)] == ' '): #
Verifica se o índice calculado pela função
hash está vazio.
9     hashtable[hashfunct(valor,m)] = valor # Se estiver vazio, insere o valor no índice
calculado.
10  else: # Se o índice já estiver ocupado por
outro valor, uma colisão é detectada.
11    print() # Imprime uma mensagem
indicando a detecção de uma colisão.

# Parte 3: Preenchimento inicial dos índices
da tabela hash
13 for i in range(m): # Loop para preencher os
índices da tabela hash com espaços vazios.
14   hashtable[i] = ' ' # Inicializa cada índice
da tabela hash com um espaço vazio.

# Parte 4: Inserção de valores na tabela hash
17 # Inserções de testes com valores
específicos para demonstrar como os valores
são armazenados e como colisões são
detectadas.
18 insereTC(235)
19 insereTC(578)
20 insereTC(1024)
21 insereTC(96)
22 insereTC(32)

24 print(hashtable) # Imprime o estado atual
da tabela hash após as inserções.

# Parte 5: Teste de inserção que gera uma
colisão
```

```
26 insereTC(18) # Tenta inserir um valor que  
causará uma colisão.
```

```
28 print(hashtable) # Imprime o estado final  
da tabela hash, mostrando o resultado após a  
tentativa de inserção que gera colisão.
```

Nesse exemplo, desenvolve-se uma função que verifica se uma chave específica na tabela *hash* está livre. Se estiver, o elemento é adicionado; se não, o programa sinaliza a ocorrência de uma colisão e não procede com a inserção. Ao rodar o algoritmo após essa modificação, observa-se, conforme mostrado na Figura 4, que o elemento na posição chave 8 permanece inalterado, e o valor 18 não é inserido devido à detecção da colisão.

```
In [15]:
```

```
{0: ' ', 1: ' ', 2: 32, 3: ' ', 4: 1024, 5: 235, 6: 96, 7: ' ', 8: 578, 9: ' '}
```

```
Colisao detectada, tratar colisão...
```

```
{0: ' ', 1: ' ', 2: 32, 3: ' ', 4: 1024, 5: 235, 6: 96, 7: ' ', 8: 578, 9: ' '}
```

Figura 4 | Resultado do algoritmo com tratamento de colisão. Fonte: adaptada de Dias (2021).

Existem diversas abordagens para resolver colisões em tabelas *hash*. Atualmente, uma opção seria adicionar elementos que causam colisões ao final do dicionário ou, especificamente para esse contexto, colocá-los na posição $m+10$. O método de resolução de colisões depende muito das particularidades do problema em questão e deve ser adaptado conforme necessário.

Nesta aula, exploramos as tabelas *hash* e seus conceitos fundamentais, destacando sua aplicação prática em Python. Essas estruturas são essenciais na resolução de problemas computacionais, tornando-se um assunto relevante no estudo de estruturas de dados. Incentivamos a implementação e experimentação dos algoritmos discutidos para um aprendizado mais profundo. Continue explorando e boa sorte nos estudos.

Vamos Exercitar?

Para solucionar o problema de contagem e categorização de produtos da empresa, optamos pela estrutura de dados tabela *hash*. Essa escolha se deve à eficiência da tabela *hash* em recuperar informações rapidamente, evitando buscas complexas que seriam necessárias em estruturas de dados alternativas. A implementação segue o princípio do exemplo fornecido, adaptando-se ao contexto de 15 classes de produtos distintas. Utilizamos a função *hash* baseada no método da divisão, definindo $m = 15$, onde a classe 0 corresponde à classe 15. Experimentos demonstram que a aplicação do módulo 15 aos primeiros 8 dígitos do número de etiqueta gera uma chave que reflete os dois últimos dígitos, representando a classe do produto.

Ao determinar a posição da chave na tabela, incrementamos essa posição em uma unidade, sem armazenar o valor inicialmente proposto. Existem duas abordagens para o incremento da contagem de produtos: utilizar um vetor de inteiros, onde os valores são diretamente incrementados, ou empregar um dicionário, convertendo *strings* em inteiros (e vice-versa) durante o incremento. Optamos por demonstrar a solução utilizando um vetor de inteiros para ilustrar uma alternativa de implementação do *hash*.

Uma das maneiras de resolver o problema é apresentado a seguir:

```
# Importa a biblioteca array para manipular arrays
import array as arr

# Define o número de classes de produtos
m = 15

# Inicializa a tabela hash como um array de inteiros com 15 posições, todas começando com
# o valor 0
hashtable = arr.array('I', [0]*m)

# Função hash que utiliza o método da divisão para determinar a posição do produto na
# tabela hash
def hashfunct(v, mh):
    return v % mh # Retorna o resto da divisão de v por mh, que é o número de classes

# Função para inserir/incrementar a contagem de produtos na tabela hash
def insereTC(valor):
    # Acessa a posição na tabela hash determinada pela função hash e incrementa o valor
    hashtable[hashfunct(valor, m)] += 1

# Função para retornar a quantidade de produtos de uma determinada classe
def retornaV(valor):
    # Retorna o valor armazenado na posição da tabela hash correspondente à classe do
    # produto
    return hashtable[hashfunct(valor, m)]

# Testes
print(hashtable) # Exibe a tabela hash inicial
print()
x = int(input()) # Lê o número da etiqueta do produto
insereTC(x) # Insere o produto na tabela hash
print()
print(hashtable) # Exibe a tabela hash após a inserção
print()
x = int(input()) # Lê o número da etiqueta para busca
print()
```

```
print(retornaV(x)) # Exibe a quantidade de produtos da classe correspondente
```

Saiba mais

Conceitos de maps e hash map: chave-valor:

- SQUIRRELS, J. [HashMap: que tipo de mapa é esse?](#) Codegym.

Tabelas hash:

- GASPAR, W. [O que é e como funciona a estrutura de dados Tabela Hash?](#) WagnerGaspar.

Implementação de tabelas hash em Python:

- AWARI. [COMO usar a estrutura de dados hash table em python para otimizar seu código.](#) Awari.

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

BRAGANHOLO, V. **Árvore B**. Estruturas de Dados e Seus Algoritmos, 2020. Disponível em: <https://braganholo.github.io/material/ed/11-ArvoreB.pdf>. Acesso em: 30 jan. 2024.

DIAS, M. A. **Análise de dados estruturados**. Estrutura de Dados, 2021. Disponível em: <https://bit.ly/49g36r9>. Acesso em: 17 fev. 2024.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

Aula 2

Análise de Dados Estruturados

Análise de Dados Estruturados



Este conteúdo é um vídeo!

Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Olá, estudante!

Nesta videoaula, exploraremos as poderosas aplicações de grafos para análise de dados em Python, mergulhando na base de dados orientada a grafos. Veremos como essa abordagem revoluciona a visualização e interpretação de dados complexos, tornando-se essencial para sua prática profissional em campos que exigem análise de dados avançada. Descubra como transformar dados brutos em insights valiosos através de grafos.

Não perca essa oportunidade de aprimorar suas habilidades. Junte-se a nós nessa jornada de aprendizado!

Ponto de Partida

Olá, estudante!

Bem-vindo à jornada de descoberta da análise de dados estruturados através de grafos. Nesta aula, exploraremos o uso de grafos na análise de dados, investigaremos a composição das bases de dados fundamentadas em grafos e aprenderemos como visualizar essas estruturas complexas. Além disso, desvendaremos as funcionalidades e vantagens dos bancos de dados orientados a grafos, tudo isso utilizando a linguagem Python para construir nossos exemplos práticos.

Ao término desta aula, você terá uma compreensão sólida sobre a implementação e as situações ideais para empregar bancos de dados em grafos. Aprenderá sobre o design e os atributos essenciais desses bancos de dados, como visualizá-los eficientemente e, mais importante, identificará as aplicações práticas que se beneficiam dessa estrutura – como a modelagem de relações de afinidade em plataformas de *e-commerce* (Lambert, 2022).

Com a análise de dados se tornando cada vez mais importante no cenário corporativo, esta aula é uma oportunidade valiosa para adquirir habilidades procuradas por empresas inovadoras, preparando-o para enfrentar desafios reais no seu futuro profissional.

Para assimilarmos o conteúdo que será apresentado, imagine que você passou no processo seletivo do Banco KPL. Durante a semana inaugural de treinamento, você enfrentará um desafio

ESTRUTURA DE DADOS

prático: colaborar com uma equipe experiente do banco para desenvolver soluções inovadoras para problemas atuais. Sua equipe está na linha de frente da detecção de fraudes em transações com cartões de crédito, uma área crítica que demanda precisão e agilidade. As transações, que incluem uma variedade de operações como compras únicas, parcelamentos, saques e verificações on-line, estão apresentando um número significativo de falsos positivos e negativos.

Em outras palavras, transações legítimas estão sendo erroneamente sinalizadas como fraudulentas, enquanto algumas fraudulentas não estão sendo detectadas. Diante desse cenário, você tem carta branca para propor alterações no sistema de detecção de fraudes. Sua proposta deve fundamentar-se em análises robustas e considerar as falhas dos métodos atuais. Para estruturar sua solução, reflita sobre os conteúdos abordados na seção, em especial a utilização de grafos na representação e análise de dados. Por exemplo, imagine a aplicação de um modelo de grafos usando a biblioteca NetworkX para mapear as transações e identificar padrões suspeitos. Como você aplicaria essa ferramenta para reduzir os erros de classificação? Encorajamos você a integrar o conhecimento adquirido e inovar no combate à fraude no Banco KPL.

Entender a organização de dados em estruturas de grafos e como essa representação pode ser maximizada para análises avançadas é uma habilidade essencial para profissionais do seu campo. Está preparado para embarcar nessa jornada de aprendizado? Vamos iniciar!

Bons estudos.

Vamos Começar!

Aplicações de grafos para análise de dados em Python

Nesta aula, abordaremos os grafos e sua implementação em bancos de dados com uso de Python, culminando na exploração de suas aplicações-chave. Antes, revisaremos o conceito fundamental de grafos: estruturas que representam dados do mundo real através de vértices (os pontos como A, B, C, D apresentados na Figura 1) e arestas (as linhas que os conectam, como 1, 2, 3, 4), podendo ambos carregar variadas informações.

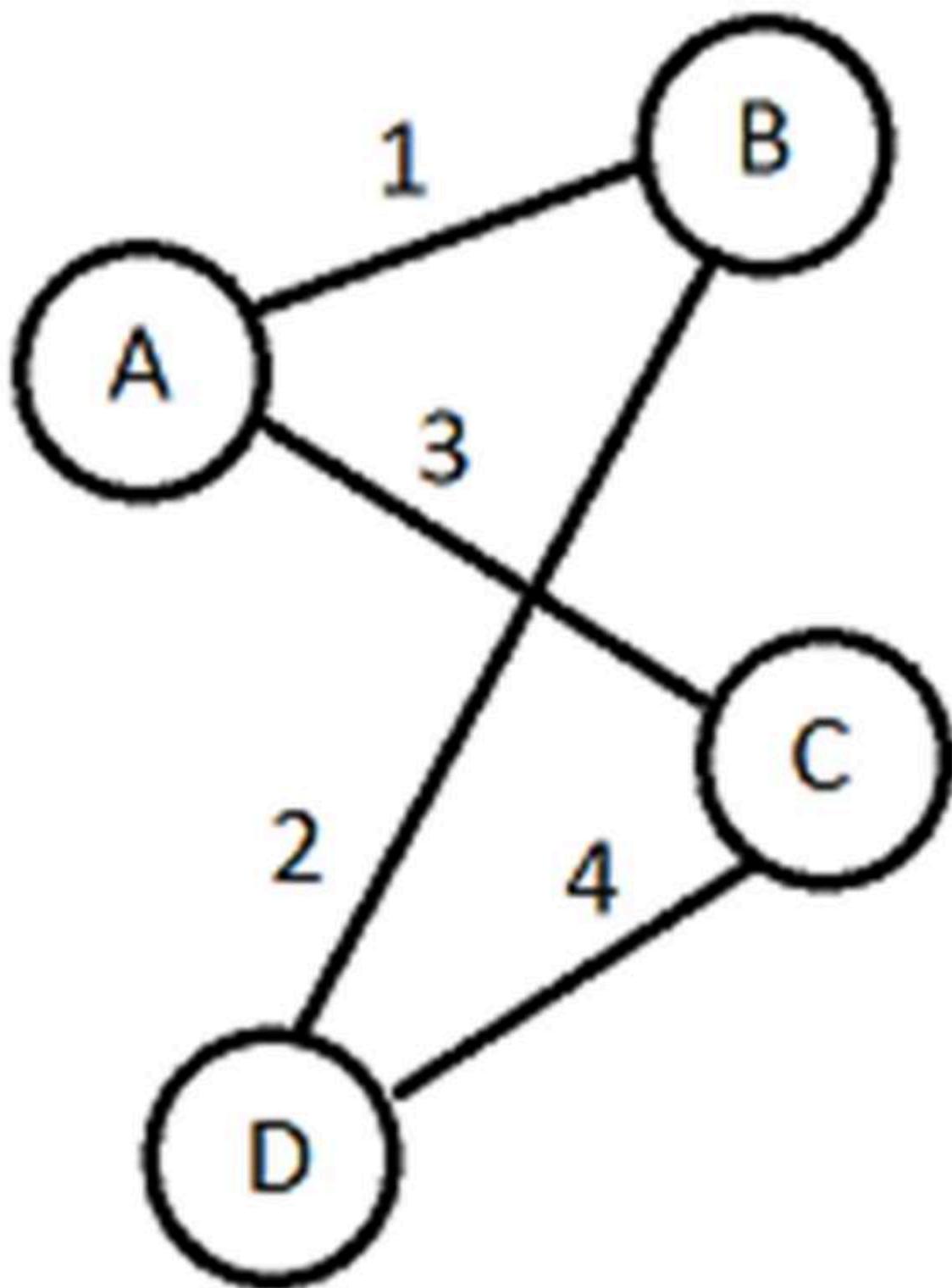


Figura 1 | Exemplo de grafo acíclico. Fonte: adaptada de Dias (2021).

ESTRUTURA DE DADOS

Ademais, exploraremos os bancos de dados relacionais tradicionais, que se baseiam em tabelas interligadas e consultas SQL, e os bancos de dados NoSQL, que utilizam estruturas distintas para relacionar dados. Discutiremos também o conceito de Big Data, caracterizado por seu imenso **volume, alta variedade** e rápida **velocidade** de coleta, que exige novas abordagens para armazenamento e análise devido à sua complexidade e tamanho (Dias, 2021).

As características volumosas, variadas e velozes do Big Data são comumente descritas pelos "3 Vs". Com a crescente prevalência do Big Data na coleta diária de informações em diversos mercados, os bancos de dados relacionais tradicionais mostraram-se inadequados para gerenciar tais dados com a agilidade requerida. A solução emergiu na forma de bancos de dados não relacionais, os NoSQL, que oferecem tempos de resposta menores devido às suas características intrínsecas, proporcionando uma gestão de dados mais eficiente em ambientes de Big Data (Dias, 2021).

Base de dados orientada a grafos

Dentre as variedades de bancos de dados NoSQL, destacam-se os orientados a grafos, que retomaram seu uso para armazenamento e análise de dados devido à crescente complexidade das relações entre dados. Essa prática, inicialmente introduzida em estudos científicos da década de 1960, encontrou uma nova relevância no cenário moderno. Os grafos são indicados para essa aplicação devido à sua capacidade natural de representar e analisar relações complexas de maneira eficiente.

Atualmente, os dados não são apenas volumosos, variados e rápidos, mas também crescentemente interconectados, exigindo uma estrutura de armazenamento que permita acessar e compreender essas conexões com eficácia. Ao converter dados para um modelo baseado em grafos, entidades de modelos de dados antigos tornam-se vértices, e as relações anteriormente manifestadas em chaves e tabelas tornam-se arestas. Esta construção, apesar de sua simplicidade conceitual, permite associar informações detalhadas tanto aos vértices quanto às arestas (Dias, 2021).

Com um grafo estruturado, pesquisar por relacionamentos ou informações entre dados torna-se uma questão de localizar os vértices apropriados e explorar as arestas conectadas. Isso possibilita a análise direta e eficiente das relações entre entidades, facilitando o entendimento das conexões intrincadas presentes nos dados. Diante de grafos de grande escala, com um número significativo de arestas, a análise pode ser conduzida por meio do agrupamento e da identificação de vértices-chave.

Ao se distanciar de um vértice central, o impacto de uma aresta pode ser proporcionalmente atenuado, permitindo um foco nas conexões mais relevantes. Em um banco de dados orientado a grafos, ao analisar a conexão de um consumidor com um produto, por exemplo, onde vértices representam consumidores e produtos e arestas indicam as interações de compra, a proximidade da relação é avaliada pelo número de vértices e arestas pelo qual se deve passar para conectar ambos. Um caminho mais curto sugere uma relação mais forte entre o consumidor e o produto. Esse exemplo ilustra como as relações em um banco de dados baseado

em grafos podem ser interpretadas para avaliar a relevância de produtos para os consumidores (Dias, 2021).

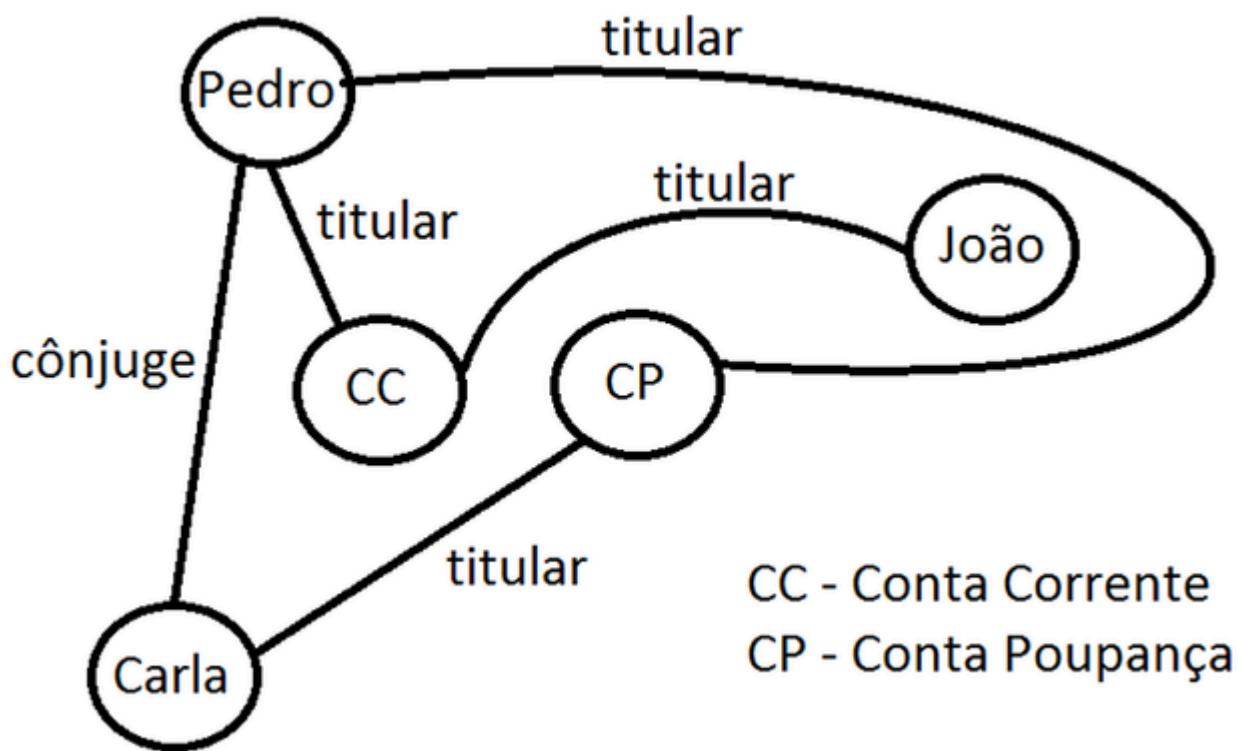


Figura 2 | Exemplo de banco de dados orientado a grafo. Fonte: adaptada de Dias (2021).

A Figura 2 ilustra um banco de dados baseado em grafos para clientes bancários, onde Pedro tem contas corrente e poupança, e Carla, que é sua cônjuge, também possui uma conta poupança, sugerindo uma possível conta conjunta. A relação de Carla e Pedro pode ser ajustada removendo a aresta "cônjugue" caso a conta não seja conjunta. João é identificado como titular de uma conta corrente. Essas informações são diretamente inferidas pelas conexões das arestas, enquanto em bancos de dados relacionais, seria necessário consultar múltiplas tabelas para reunir esses dados.

Siga em Frente...

Aplicações e visualização de dados em grafos para análise de dados em Python

A recuperação de informações em um banco de dados orientado a grafos destaca-se pela facilidade com que as arestas conectadas a um nó específico podem ser acessadas, seja essa estrutura representada por matrizes ou listas encadeadas. Diferentemente, em bancos de dados relacionais, a busca por informações envolve a procura de chaves em tabelas, o que pode se

ESTRUTURA DE DADOS

tornar um processo demorado, especialmente à medida que o tamanho das tabelas aumenta, impactando o tempo de resposta do sistema (Dias, 2021).

O Neo4j é um exemplo de um Sistema Gerenciador de Banco de Dados (SGBD) NoSQL baseado em grafos, escolhido por sua eficiência no manejo de grandes volumes de dados. Sua arquitetura é otimizada para armazenar e manipular vértices e arestas, tornando-o ideal para aplicações de Big Data com estruturas hierárquicas. O Cypher é a linguagem de consulta utilizada pelo Neo4j, análoga ao SQL para bancos de dados relacionais, e o pacote Py2Neo permite a integração com a linguagem Python, facilitando a manipulação do Neo4j dentro do ambiente Python (Alves, 2021).

Quando se trata de visualização, grafos pequenos são simples de serem representados visualmente, mas grafos maiores exigem ferramentas especializadas capazes de calcular e exibir as estruturas de forma comprehensível. Muitas bibliotecas em Python são projetadas para trabalhar com grafos e bancos de dados. Entre essas bibliotecas, daremos atenção especial à NetworkX, um pacote Python projetado para a criação, a manipulação e o estudo de redes e grafos complexos. NetworkX oferece estruturas de dados para uma variedade de tipos de grafos e redes, algoritmos de grafos, geradores de grafos e a possibilidade de personalizar nós e arestas com texto, imagens ou dados numéricos, permitindo uma análise detalhada e visualização de redes complexas (Dias, 2021).

Para começar a usar o pacote NetworkX, é preciso instalá-lo no ambiente Python. Isso pode ser feito através do comando no terminal:

```
pip install networkx
```

Após a instalação, o pacote está pronto para uso. O guia do pacote oferece uma visão completa sobre suas capacidades, exigindo um estudo detalhado para explorar todas as suas potencialidades. Nesta aula, focaremos na criação e visualização de um grafo básico para ilustrar como o pacote funciona. Vamos construir um grafo simples e mostrar como ele pode ser visualizado na tela.

```
# Importa as bibliotecas necessárias
import networkx as nx
import matplotlib.pyplot as plt

# Define uma classe para visualização de grafos
class visualizacaoGrafo:

    # Inicializador da classe
    def __init__(self):
        self.visual = [] # Lista para armazenar as arestas do grafo

    # Método para adicionar arestas ao grafo
```

```
def adicionaAresta(self, a, b):
    temp = [a, b] # Cria uma aresta temporária como uma lista de dois vértices
    self.visual.append(temp) # Adiciona a aresta à lista de visualização

# Método para desenhar o grafo
def desenhar(self):
    G = nx.Graph() # Cria um objeto grafo G utilizando NetworkX
    G.add_edges_from(self.visual) # Adiciona as arestas armazenadas na lista ao grafo
    # Desenha o grafo utilizando NetworkX, definindo a cor dos nós como cinza claro
    nx.draw_networkx(G, node_color='lightgrey')
    plt.show() # Exibe o grafo desenhado na tela

# Cria uma instância da classe de visualização de grafo
G = visualizacaoGrafo()
# Adiciona arestas ao grafo representando relações entre pessoas e contas
G.adicionaAresta('Pedro', 'CP')
G.adicionaAresta('Pedro', 'Carla')
G.adicionaAresta('Carla', 'CP')
G.adicionaAresta('Pedro', 'CC')
G.adicionaAresta('Joao', 'CC')
# Chama o método para desenhar o grafo
G.desenhar()
```

Nesse exemplo, a classe **visualizacaoGrafo** é utilizada para construir um grafo que ilustra as relações entre pessoas (Pedro, Carla, João) e suas contas (CP - Conta Poupança, CC - Conta Corrente). A lista visual armazena as arestas do grafo, onde cada aresta é uma relação entre dois vértices (por exemplo, uma pessoa e uma conta ou relações entre pessoas).

Após adicionar as arestas desejadas utilizando o método **adicionaAresta**, o método **desenhar** cria uma instância de um grafo com a biblioteca NetworkX, adiciona as arestas a partir da lista visual e, finalmente, desenha o grafo na tela utilizando a função **draw_networkx** da NetworkX, com a cor dos nós definida como cinza claro para melhor visualização.

Esse script é um exemplo prático de como representar e visualizar relações complexas de forma intuitiva e visual com grafos em Python.

A Figura 3 exibe o grafo gerado a partir das arestas definidas anteriormente. O layout dos vértices e das arestas varia automaticamente pelo software para prevenir sobreposições e garantir uma visualização clara. Calcular esses posicionamentos é uma tarefa complexa que requer expertise em computação gráfica, justificando o uso de bibliotecas como a NetworkX. Notavelmente, na construção do grafo, não especificamos parâmetros de posicionamento, evidenciando a capacidade da biblioteca de organizar os elementos do grafo de forma autônoma.

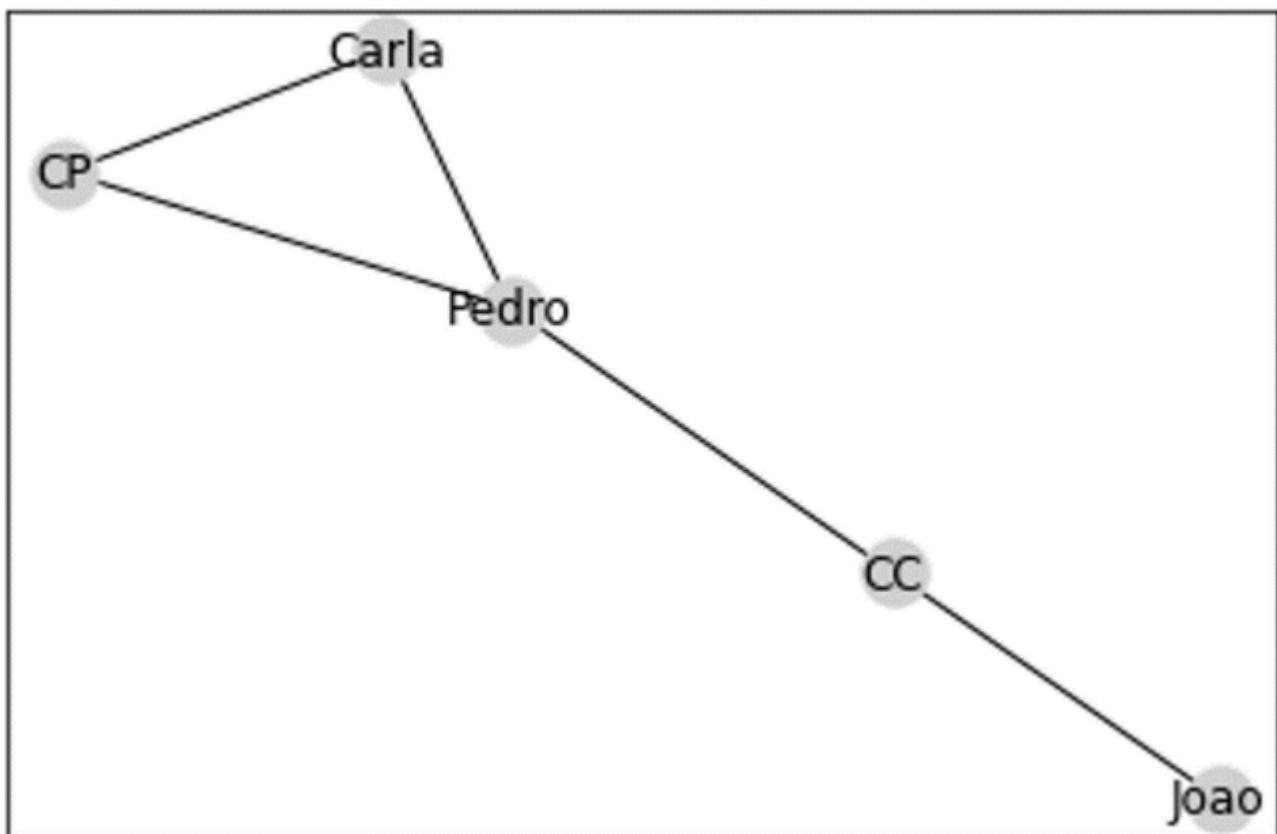


Figura 3 | Grafo resultante do algoritmo. Fonte: adaptada de Dias (2021).

Visualizar dados em um banco de dados orientado a grafos via Python pode simplificar significativamente a análise de conjuntos complexos de informações. Organizar grafos com muitos vértices e arestas apresenta desafios únicos, destacando a importância de ferramentas de visualização eficazes. Bancos de dados orientados a grafos são únicos por armazenarem não apenas os dados, mas também as relações entre eles, o que é crucial para várias aplicações práticas (Lambert, 2022).

Entre as aplicações notáveis, sistemas de recomendação se beneficiam enormemente dessa estrutura, permitindo recomendações personalizadas baseadas nas compras anteriores dos usuários. Na gestão de redes de TI, a visualização de grafos pode identificar rapidamente os pontos críticos da rede que exigem atenção especial para manter a continuidade do serviço. Além disso, o controle de acesso em sistemas informatizados pode ser otimizado com grafos, facilitando a verificação rápida das permissões de usuário (Szwarcfiter; Markenzon, 2020).

Nesta aula, introduzimos o conceito e a programação de bancos de dados orientados a grafos em Python, mostrando como a visualização de dados em grafos pode acelerar o acesso à informação e suportar a tomada de decisões. As aplicações discutidas ilustram a relevância desse conteúdo para profissionais visando o mercado de trabalho tecnológico.

Vamos Exercitar?

Durante sua semana de treinamento no Banco KPL, você foi encarregado de abordar a questão das classificações imprecisas nas transações de cartão de crédito. A equipe aponta que há uma confusão entre transações legítimas e fraudulentas, o que compromete a operação do banco.

A solução proposta concentra-se na implementação de um banco de dados orientado a grafos, inovando sobre o sistema relacional existente. A vantagem dos grafos reside na sua capacidade de capturar não apenas os dados das transações, mas também as relações entre elas, facilitando a detecção de padrões suspeitos e melhorando a segurança dos dados. A adoção dessa tecnologia NoSQL permitiria uma identificação mais eficaz e segura de comportamentos fraudulentos, modernizando os sistemas do banco (Dias, 2021).

```
# Importando as bibliotecas necessárias
# matplotlib.pyplot para a criação de gráficos
# networkx para manipulação e visualização de grafos
import matplotlib.pyplot as plt
import networkx as nx

# Definindo as arestas do grafo, que conectam clientes e lojas
edges = [('Loja 1','Cliente 1'),('Cliente 1','Cliente 2'),('Cliente','Loja 3')]

# Criando um objeto Grafo com a biblioteca networkx
G = nx.Graph()

# Adicionando as arestas definidas anteriormente ao grafo
G.add_edges_from(edges)

# Configurando o layout do grafo. O spring_layout posiciona os nós de forma que os que
# estão conectados fiquem próximos
pos = nx.spring_layout(G)

# Criando uma figura para o gráfico
plt.figure()

# Desenhando os nós do grafo com as posições definidas, coloração e borda
# O parâmetro 'node_color' define a cor interna dos nós
# O parâmetro 'node_size' define o tamanho dos nós
# O parâmetro 'alpha' define a transparência da cor do nó
nx.draw(G, pos, edge_color='black', width=1, linewidths=1,
        node_size=500, node_color='lightgrey', alpha=0.9)

# Definindo um dicionário de rótulos para os nós do grafo
labels = {node:node for node in G.nodes()}
```

```
# Desenhando os rótulos dos nós no gráfico com as posições definidas anteriormente  
nx.draw_networkx_labels(G, pos, labels)
```

```
# Personalizando e desenhando os rótulos das arestas com informações sobre transações  
# Os parâmetros 'font_size' e 'font_color' definem o tamanho e a cor do texto dos rótulos das  
arestas
```

```
edge_labels = {('Loja 1','Cliente 1'):'Transações: 2',  
               ('Cliente 1','Cliente 2'):'Transações: 5',  
               ('Cliente','Loja 3'):'Transações: 7'}
```

```
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')
```

```
# Removendo os eixos x e y para limpar a visualização do gráfico  
plt.axis('off')
```

```
# Exibindo o gráfico  
plt.show()
```

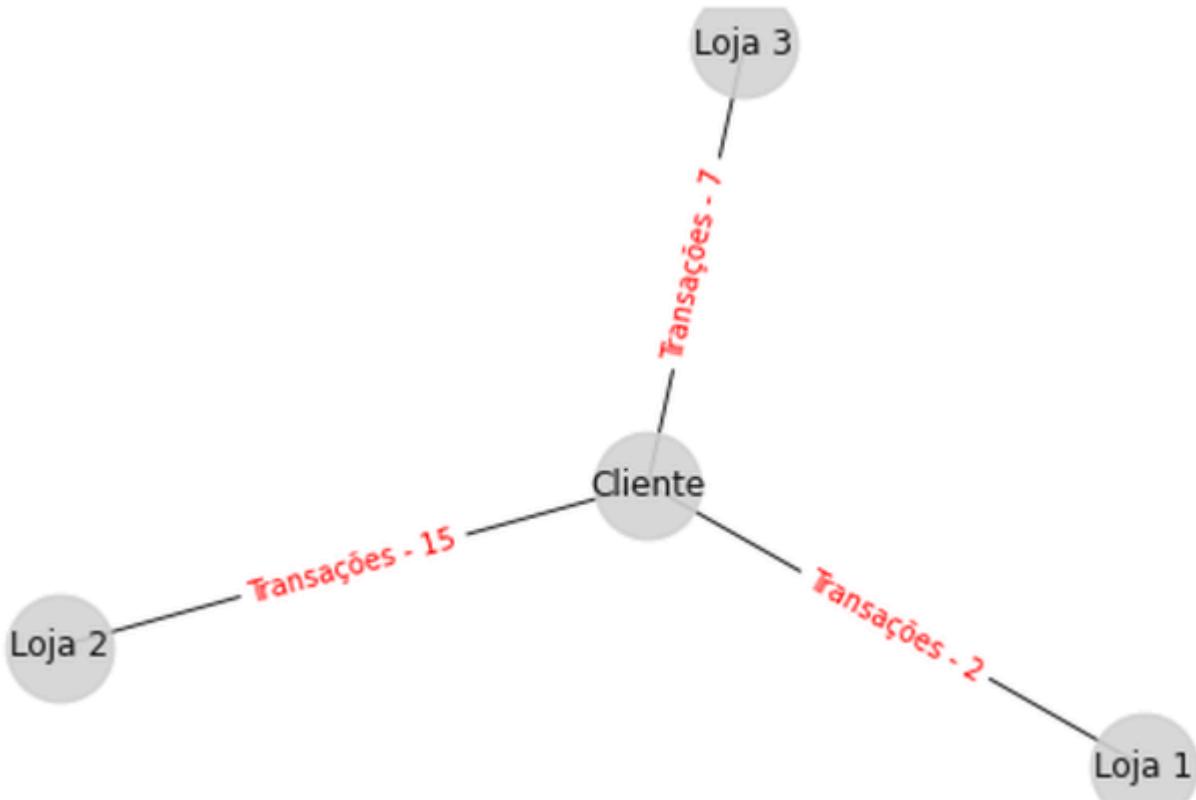


Figura 4 | Grafo exemplo gerado pelo algoritmo. Fonte: adaptada de Dias (2021).

Saiba mais

ESTRUTURA DE DADOS

Aplicações de grafos para análise de dados em Python:

- MIRANDA, D. [Análise de Dados em Grafo com Python na Área do Direito](#). Blog Diego Miranda.

Base de dados orientada a grafos:

- [BANCO de dados de grafos definido](#). Oracle.

Aplicações e visualização de dados em grafos para análise de dados em Python:

- [TUTORIAL de Python com Networkx](#): Aprenda a Criar e Analisar Redes com Facilidade. Awari.

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

DIAS, M. A. **Análise de dados estruturados**. Estrutura de Dados, 2021. Disponível em: <https://bit.ly/49g36r9>. Acesso em: 17 fev. 2024.

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

Aula 3

Introdução a Análise de Algoritmos e Estruturas de Dados

Introdução a Análise de Algoritmos e Estruturas de Dados

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Nesta videoaula, mergulharemos na análise de algoritmos e estruturas de dados, focando nas complexidades temporal e espacial. Você entenderá como a eficiência dos algoritmos é afetada pela maneira como os dados são estruturados e aprenderá a analisar a complexidade de diferentes estruturas de dados.

Esse conhecimento é fundamental para qualquer profissional da área de tecnologia, pois impacta diretamente no desempenho e na qualidade das soluções de software desenvolvidas. Junte-se a nós nesta jornada de aprendizado e descubra como otimizar seus algoritmos para alcançar melhores resultados.

Ponto de Partida

Bem-vindo à aula sobre análise de algoritmos e estruturas de dados. Ao longo desta unidade, exploramos diversas maneiras de armazenar e acessar dados, cada uma com suas funcionalidades e objetivos específicos. Nos aprofundamos na importância de escolher a estrutura de dados adequada para otimizar tanto o tempo de execução quanto o consumo de memória dos programas. Para isso, é importante entender a complexidade dos algoritmos e como aplicar esse conhecimento na seleção da estrutura de dados correta para a eficiência computacional.

Nesse contexto, vamos discutir um problema fictício enfrentado por uma empresa, que nos contratou para otimizar um algoritmo de gerenciamento de dados de clientes, atualmente implementado com uma lista encadeada não ordenada. Este estudo de caso nos permitirá aplicar os conceitos de análise de complexidade e explorar soluções alternativas, como o uso de tabelas *hash*, que oferecem melhor desempenho em operações de inserção, remoção e busca em comparação com as listas encadeadas.

Esta aula não só sintetiza os tópicos abordados ao longo da disciplina, mas também prepara você para enfrentar desafios semelhantes no mercado de trabalho. Com o domínio das estruturas de dados e da análise de complexidade, você estará apto a desenvolver soluções mais eficientes e a contribuir significativamente para projetos de desenvolvimento de software.

Prepare-se para mergulhar na análise de algoritmos e descobrir a vital importância das estruturas de dados na computação.

Bons estudos!

Vamos Começar!

Nesta aula, abordaremos os fundamentos das estruturas de dados e sua aplicação crítica em algoritmos, destacando a importância desses conceitos no contexto do desenvolvimento de software. Estruturas de dados, como vetores, listas, pilhas, filas, árvores e tabelas *hash*, são essenciais para o armazenamento e gerenciamento eficiente de dados, permitindo que algoritmos os processem de maneira eficaz.

Complexidades temporal e espacial em estrutura de dados

A eficiência de um algoritmo não se limita à sua lógica; ela também é influenciada pelo uso de recursos computacionais, como **tempo** de execução e consumo de **memória**. Esses aspectos são impactados pela escolha da estrutura de dados adequada. Diante de um problema específico e várias soluções algorítmicas possíveis, a seleção da estrutura de dados correta é essencial para otimizar o desempenho e a eficiência do algoritmo. Vamos explorar como identificar a estrutura mais apropriada para cada situação, considerando as características do problema e as soluções disponíveis (Dias, 2021).

Nesse contexto, a Figura 1 ilustra uma situação típica onde enfrentamos um problema específico com várias opções de algoritmos que podem ser aplicados como solução. A questão então se torna: como selecionar o algoritmo mais adequado?

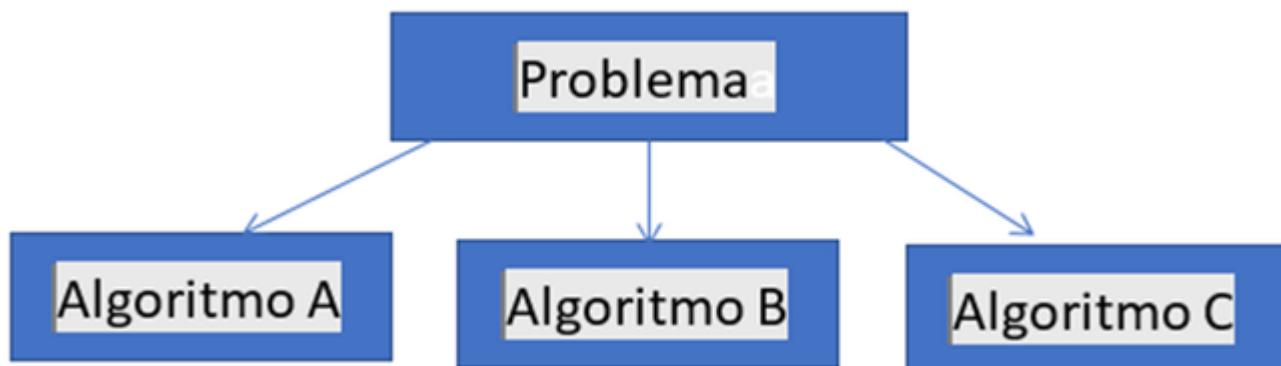


Figura 1 | Situação de escolha de algoritmos. Fonte: adaptada de Dias (2021).

Para determinar a complexidade de um algoritmo, efetuamos um cálculo matemático focando na operação que mais impacta a variável em análise. No caso da complexidade de tempo, consideramos as instruções que demandam maior tempo de execução. Da mesma forma, a complexidade de espaço é calculada levando em conta as estruturas de dados utilizadas e a alocação de memória dinâmica. Esse método de análise aplica-se igualmente às estruturas de dados.

Ao discutir como uma estrutura de dados influencia a eficiência de um algoritmo, é essencial compreender as implicações dessa escolha. Para um problema que envolve busca de dados específicos, a seleção da estrutura de dados adequada é essencial para facilitar a localização desses elementos. Utilizar uma lista encadeada pode limitar a eficiência da busca devido à sua natureza linear, onde, na ausência de ordenação, uma busca sequencial se faz necessária, geralmente resultando em menor eficiência. Alternativamente, estruturas hierárquicas como

ESTRUTURA DE DADOS

árvores podem oferecer vantagens na busca, embora possam introduzir complexidade adicional na manipulação.

A relevância das estruturas de dados no campo da computação está intimamente ligada à natureza dos dados atuais, onde sistemas computacionais gerenciam volumes enormes de informações a velocidades elevadas, caracterizando o cenário de Big Data (Dias, 2021). Um design de aplicação que não leva em conta esse contexto pode ver sua eficácia reduzida. As estruturas de dados são importantes por várias razões, incluindo:

- **Velocidade de processamento:** algoritmos que não empregam estruturas de dados apropriadamente podem falhar em processar dados eficientemente, mesmo em computadores potentes.
- **Busca:** a eficácia na busca por dados armazenados, minimizando o uso de recursos como tempo de processamento, é diretamente afetada pela escolha da estrutura de dados.
- **Requisições múltiplas:** estruturas de dados apropriadas são capazes de suportar um alto volume de requisições, um aspecto comum em softwares durante o processamento de dados.

Além disso, o uso de estruturas de dados contribui para uma gestão de memória computacional eficiente, permite a reutilização de código e oferece a flexibilidade para modelar e resolver uma ampla gama de problemas computacionais. Dessa forma, as estruturas de dados servem como fundamentos essenciais na construção de algoritmos computacionais eficazes para o processamento de dados.

Explorando a relevância das estruturas de dados, abordaremos suas complexidades de tempo e espaço, fundamentando nossa discussão na teoria de análise de complexidade de algoritmos. Conforme Cormen (2012), analisar um algoritmo envolve prever os recursos necessários para sua execução, principalmente o tempo de execução e a quantidade de memória utilizada, além de outros requisitos em casos específicos, como a largura de banda para comunicação. Essa análise requer a associação com um modelo computacional, considerando que o desempenho de um algoritmo pode variar em diferentes hardwares. A base comum para essas análises é o modelo de um computador de um único núcleo operando com memória RAM, uma premissa que uniformiza os cálculos de complexidade ao desconsiderar particularidades de máquinas específicas.

Esse modelo computacional serve de fundamento para um método matemático que permite determinar a complexidade dos algoritmos sem ser afetado pelas variáveis de um equipamento físico particular. Na prática, isso significa uma análise detalhada do algoritmo, instrução por instrução, justificada pela maneira como a memória RAM processa instruções através do núcleo do processador. Nesse contexto, o objetivo é identificar uma função de custo ou complexidade, denotada por $F(n)$, que reflete o esforço necessário para executar um algoritmo com uma entrada de tamanho ' n ', representando a complexidade de tempo. É importante notar que, embora denominada complexidade de tempo, essa medida não quantifica o tempo diretamente, mas sim o **número de execuções** de uma **operação** considerada **crítica**.

Da mesma forma, se $F(n)$ indicasse o volume de memória necessário para a execução de um algoritmo com uma entrada de tamanho n , estaríamos descrevendo a complexidade de espaço do algoritmo.

Além do tempo de execução e do consumo de memória, outros fatores podem influenciar a avaliação da eficiência de um algoritmo. Por exemplo, a análise pode incluir como a organização prévia dos dados de entrada afeta o desempenho de algoritmos de inserção ordenada (Dias, 2021). Em alguns casos, a eficiência é verificada simplesmente medindo o tempo de execução do algoritmo. O objetivo do modelo matemático é desvincular o cálculo da complexidade de variáveis externas, focando nas operações cruciais do algoritmo. É interessante refletir sobre as diversas métricas que podem ser usadas para comparar algoritmos em termos de estruturas, operações e linguagens de programação.

Siga em Frente...

Eficiência de algoritmos com dados estruturados

Para calcular a complexidade de um algoritmo, é essencial identificar a operação que mais impacta o desempenho em relação à métrica de interesse. O processo envolve determinar essa operação-chave e analisar como ela se comporta com diferentes tamanhos de entrada de dados.

Existem três cenários principais considerados na análise de algoritmos:

- **Melhor caso:** quando o algoritmo se beneficia das condições de entrada. Esse cenário é raro na prática.
- **Pior caso:** representa o desempenho mais lento possível, geralmente causado por condições de entrada desfavoráveis. Embora possa ocorrer, espera-se que não seja a norma.
- **Caso médio:** o cenário mais comum, baseado em uma distribuição conhecida das entradas. Essa análise é frequentemente usada para comparações.

O parâmetro "***n***estabilidade do seu tempo de execução, mesmo com grandes variações no número de entradas. Algoritmos eficazes mantêm um tempo de execução relativamente constante, enquanto algoritmos menos eficazes mostram um aumento significativo no tempo necessário para processar entradas maiores. Esses conceitos fundamentam a análise da complexidade e são essenciais para entender como realizar esses cálculos (Dias, 2021).

Para facilitar a compreensão da complexidade, tomaremos como exemplo o algoritmo de busca sequencial. Imagine um vetor com 7 posições destinadas a números inteiros. Pretende-se armazenar sete números nesse vetor e, posteriormente, realizar a busca por um número específico dentre os armazenados. A implementação dessa busca, utilizando a linguagem Python, é ilustrada a seguir (Lambert, 2022):

```
def buscaSeq(x, elem):
    n = len(x)
    for i in range (0,n):
        if x[i] == elem:
            return 1
    return 0
```

No código apresentado, se o elemento buscado for localizado, a função retorna 1; se não for encontrado, retorna 0. A etapa chave desse algoritmo é a comparação efetuada (*if*). Essa comparação é necessária para verificar se o elemento procurado corresponde a algum dos elementos armazenados na estrutura de dados (Alves, 2021).

Nesse exemplo, destacam-se duas considerações fundamentais. Primeiramente, suponhamos que temos o seguinte vetor:

$$X = [3, 2, 4, 5, 9]$$

$$X = [3, 2, 4, 5, 9]$$

Abordaremos as três condições típicas de análise:

- **Melhor caso:** ocorre quando o elemento procurado se encontra na primeira posição do vetor, demandando somente uma comparação. Um exemplo seria a busca do número **3** no vetor 'X'.
- **Pior caso:** acontece quando o elemento desejado está na última posição ou não está presente no vetor. Por exemplo, a busca pelo número **9** no vetor 'X'.
- **Caso médio:** este se verifica quando o elemento-alvo situa-se em algum ponto mediano do vetor, implicando que apenas metade do vetor necessita ser examinada. A busca pelo número **4** no vetor 'X'.

Para cada uma dessas situações, é possível estabelecer a função de complexidade $F(n)$:

- **Melhor caso:** $F(n) = 1$, já que apenas uma comparação é necessária.
- **Pior caso:** $F(n) = n$, implicando na necessidade de percorrer todos os elementos do vetor.
- **Caso médio:** este é estimado por $F(n) = n/2$, considerando uma análise que divide o vetor ao meio.

O código apresentado ilustra que, para a busca sequencial, a complexidade no caso médio é $F(n) = n/2$, caracterizando-se como uma complexidade linear. Isso significa que o tempo necessário para a execução do algoritmo cresce proporcionalmente ao tamanho da entrada 'n'. Ao examinar detalhadamente, observa-se que a complexidade específica de cada problema é calculada para um dado algoritmo, mas compreender a função de complexidade diretamente do cálculo matemático pode ser desafiador (Dias, 2021). Para facilitar, utiliza-se a notação de dominância assintótica, conhecida como **Big O**.

Introdução à análise de complexidade de estruturas de dados

A dominância assintótica indica que, na maioria dos casos, a função real de complexidade de um algoritmo é menor ou igual à função $F(n)$, permitindo afirmar genericamente a complexidade de tempo de um algoritmo como $O(F(n))$. Por exemplo, afirmar que a complexidade de tempo de um algoritmo é $O(n)$ implica que sua complexidade exata é dominada por uma função linear $F(n) = n$. Isso significa que, para uma entrada de 1000 elementos, o algoritmo realizará a operação mais custosa 1000 vezes; se a entrada for de um milhão, a operação ocorrerá um milhão de vezes. Por outro lado, um algoritmo com complexidade $O(1)$ executa um número fixo de operações, independentemente do tamanho da entrada, sendo classificado como de complexidade constante.

Para determinar o $F(n)$ ideal e classificar a complexidade de um algoritmo, analisa-se o termo de **maior grau** no polinômio resultante da função exata de complexidade. Se o maior grau for 1, a complexidade é

$$O(n); \text{ se for 2, é } O(n^2); \text{ se for 3, é } O(n^3)$$

, e assim sucessivamente. Comparar as curvas de complexidade de diferentes funções ajuda a entender o impacto na performance entre um algoritmo

$$O(n) \text{ e um } O(n^2)$$

, por exemplo. A Figura 2 oferece uma comparação simples do tempo de execução em função do número de entradas para quatro funções comuns de complexidade.

$$O(n); \text{ se for 2, é } O(n^2); \text{ se for 3, é } O(n^3)$$

$$O(n) \text{ e um } O(n^2)$$

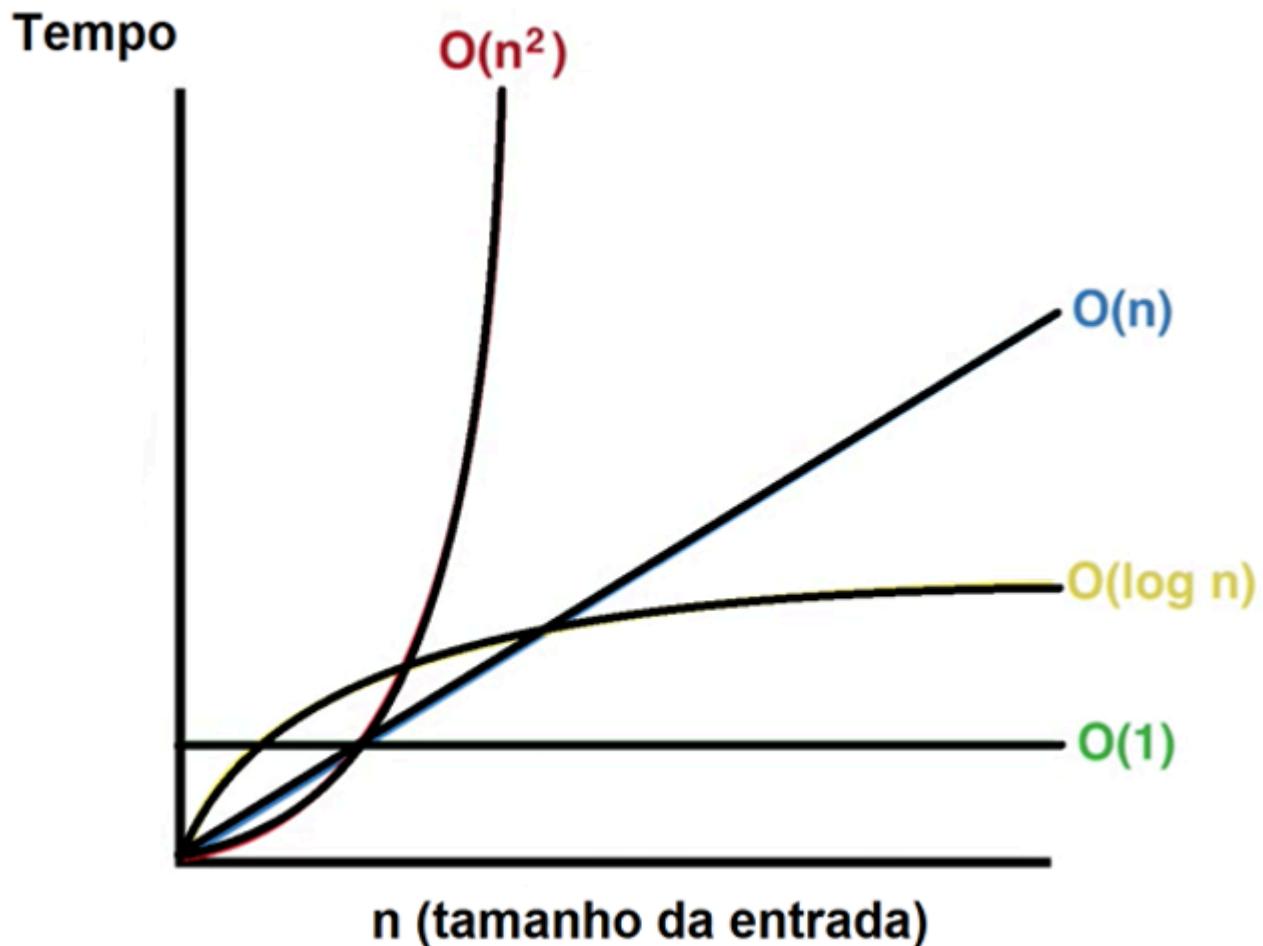


Figura 2 | Exemplos de funções para comparação de complexidade. Fonte: adaptada de Dias (2021).

A Figura 2 ilustra como o aumento do número de entradas ' n ' afeta distintamente as curvas de complexidade de tempo, usando a notação **Big O**. A escolha adequada de um algoritmo é essencial, visto que algoritmos com complexidade

$$O(n^2)$$

, por exemplo, podem ter um impacto significativo no desempenho. Embora alguns problemas exijam algoritmos de complexidade mais alta, a eficiência geral do algoritmo não deve ser ignorada. Felizmente, para muitos algoritmos, a análise de complexidade já foi realizada e está disponível em literatura especializada, evitando a necessidade de cálculos individuais. Observe o quadro a seguir, onde fornecemos o caso médio para efeito de análise de diversas estruturas:

$$O(n^2)$$

Estrutura	Operação	Complexidade
Lista	Busca Binária	$O(\log n)$

ESTRUTURA DE DADOS

	Inserção no Início	$O(1)$
	Inserção no Final	$O(1)$
	Inserção Ordenada	$O(n)$
	Consulta	$O(n)$
	Remoção	$O(n)$
	Esvaziar	$O(n)$
Pilha	Inserção	$O(1)$
	Remoção	$O(1)$
	Consulta	$O(n)$
	Esvaziar	$O(n)$
Fila	Inserção	$O(1)$
	Remoção	$O(1)$
	Consulta	$O(n)$
	Esvaziar	$O(n)$
Heap	Consulta (Max e Min)	$O(1)$
	Remoção (Max e Min)	$O(\log n)$
	Inserção	$O(\log n)$
Hash linear e quadrático	Inserção	$O(n)$
	Remoção	$O(n)$
	Busca	$O(n)$
Hash lista	Inserção	$O(1)$
	Busca com k elementos na posição da tabela	$O(k)$
	Remoção com k elementos na posição da tabela	$O(k)$
Árvore Binária	Busca	$O(n)$
	Inserção	$O(\log n)$
	Remoção	$O(\log n)$
Árvore AVL	Inserção	$O(\log n)$

	Remoção	$O(\log n)$
	Busca	$O(n)$

Tabela 1 | Complexidade dos algoritmos de estruturas de dados. Fonte: adaptada de Dias (2021).

A Tabela 1 detalha a complexidade de diversas operações nas estruturas de dados. Embora as diferenças entre as complexidades de cada função possam parecer pequenas, a escolha da estrutura adequada torna-se importante quando lidamos com um grande número de entradas. Por exemplo, pode ser mais eficaz utilizar uma tabela *hash* com listas do que armazenar os dados em listas simples.

Concluímos assim esta aula. Agora, você deve estar equipado para identificar as estruturas de dados mais apropriadas para diferentes algoritmos e decidir qual estrutura utilizar para solucionar problemas específicos. Espera-se que você leve adiante os conhecimentos adquiridos e aplique-os efetivamente em futuras disciplinas e no desenvolvimento de software, alcançando excelentes resultados.

Vamos Exercitar?

O problema apresentado no *Ponto de partida* está intrinsecamente ligado ao conteúdo abordado nesta aula, com base nos detalhes fornecidos pelo proprietário da empresa que solicitou sua análise e soluções:

- O algoritmo em questão é um sistema de gerenciamento de dados, que depende fundamentalmente de estruturas de dados para o armazenamento de informações.
- O cliente reporta que o sistema está lento, indicando uma falha na sua eficiência operacional.
- Especificamente, a lentidão é notada nas operações de busca e inserção de dados.
- A estrutura de dados utilizada é uma lista encadeada não ordenada.

Diante dessas informações, torna-se evidente que o problema está, direta ou indiretamente, associado à **escolha da estrutura de dados**, e que a substituição ou modificação dessa estrutura pode mitigar ou até resolver completamente o problema.

Insights para sua solução

A análise deve considerar a estrutura escolhida, oferecendo diversas vias de resolução. A lista encadeada tem a vantagem de permitir inserções constantes $O(1)$ no início ou no fim da lista. Contudo, a lentidão mencionada sugere que as inserções podem estar sendo feitas no fim da lista sem um ponteiro de referência direta, exigindo um percurso completo da lista a cada nova inserção, resultando em uma complexidade de inserção $O(n)$.

A questão da busca é mais simples de diagnosticar, pois em uma lista encadeada não ordenada, a busca por qualquer elemento é necessariamente sequencial, com uma complexidade também de $O(n)$.

Uma solução imediata poderia ser a adição de um ponteiro ao final da lista para agilizar as inserções para $O(1)$, reduzindo o tempo dessas operações. Alternativamente, organizar a lista de forma ordenada manteria a complexidade de inserção em $O(n)$, mas permitiria a aplicação de uma busca binária, reduzindo a complexidade de busca para $O(\log n)$.

A adoção de uma árvore binária para armazenamento e busca poderia ser outra solução viável, apresentando-se como uma boa alternativa para o problema da empresa.

A chave aqui é ponderar sobre as diferentes possibilidades de otimização das estruturas de dados. Ao avaliar a complexidade dos diversos algoritmos, várias conclusões e propostas de solução emergem. Qualquer solução proposta deve ser igualmente boa ou superior à atual, sendo imprescindível evitar alterações no software que introduzem estruturas de dados com complexidades de tempo de execução superiores à da lista encadeada não ordenada.

Saiba mais

Complexidades temporal e espacial em estrutura de dados:

- PEREIRA, W. [Introdução à complexidade de algoritmos. Blog Nagoya Foundation.](#)

Eficiência de algoritmos com dados estruturados:

- AZEVEDO, L. [A Importância da estrutura de dados e algoritmos bem estruturados na programação. Blog pessoal devleonardoazevedo.](#)

Introdução à análise de complexidade de estruturas de dados:

- HUANG, S. [O que é a notação Big O: complexidade de tempo e de espaço. FreeCodeCamp.](#)

Referências

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expressa, 2021.

CORMEN, T. **Algoritmos - Teoria e Prática**. Rio de Janeiro: GEN, 2012.

DIAS, M. A. **Análise de dados estruturados**. Estrutura de Dados, 2021. Disponível em: <https://bit.ly/49g36r9>. Acesso em: 17 fev. 2024.

LAMBERT, K. A. **Fundamentos de Python:** estruturas de dados. São Paulo: Cengage Learning, 2022.

Aula 4

Algoritmos de Ordenação e Busca Avançados

Algoritmos de Ordenação e Busca Avançados

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Bem-vindo a uma jornada pelo coração da computação eficiente! Nesta aula, vamos desvendar os segredos da eficiência dos algoritmos de ordenação e explorar detalhes da busca binária com suas variações. Esses conceitos não são apenas teorias abstratas, eles são ferramentas poderosas que aprimoram a performance e a competência profissional no desenvolvimento de software. Entenda essas técnicas para transformar a complexidade em simplicidade e a lentidão em rapidez. Bons estudos!

Ponto de Partida

Olá, estudante!

Nesta aula, iremos nos aprofundar no universo dos algoritmos, explorando a eficiência dos algoritmos de ordenação, a busca binária e suas variações, além das aplicações práticas e otimizações. Vamos descobrir não só como essas técnicas funcionam, mas também por que elas são tão essenciais no campo da computação e tecnologia.

Nesse mundo de dados em expansão, a habilidade de organizar e acessar informações de forma rápida e eficiente é o que distingue sistemas de ponta dos demais. A capacidade de ordenar e buscar dados eficientemente é fundamental, e é aqui que a importância dos algoritmos de ordenação e busca binária se destaca. Eles são a base para funções que vão desde a simples

ESTRUTURA DE DADOS

organização de contatos em seu telefone até operações complexas em bancos de dados corporativos.

Nossa discussão será guiada por um estudo de caso intrigante: uma empresa de *e-commerce* que enfrenta o desafio de otimizar seus processos de busca e classificação de produtos para melhorar a experiência do usuário. Como podemos ajudá-los a alcançar esse objetivo? Quais estratégias e algoritmos deveriam ser implementados para garantir que um cliente encontre o que procura com eficiência e precisão? Essas são as questões que nortearão nosso aprendizado hoje.

À medida que avançarmos, encorajo você a pensar criticamente sobre como esses algoritmos podem ser aplicados e otimizados em diferentes cenários. Mantenha-se atento, pois as habilidades que você desenvolverá aqui são altamente valorizadas no mercado de trabalho e podem ser a chave para inovações em sua futura carreira profissional.

Com isso em mente, prepare-se para mergulhar no mundo dos algoritmos. Esteja pronto para ser surpreendido, para desafiar o conhecimento convencional e para ver a teoria ganhar vida.

Vamos começar?

Bons estudos.

Vamos Começar!

Eficiência dos algoritmos de ordenação

A eficiência dos algoritmos de ordenação é um conceito fundamental na Ciência da Computação, especialmente quando se trata de otimizar o desempenho de aplicativos e sistemas. A eficiência é medida principalmente em termos de tempo de execução e uso de memória, sendo essencial para a escolha do algoritmo mais adequado para um determinado conjunto de dados ou aplicação. Nesta aula, usaremos como exemplo alguns algoritmos de ordenação conhecidos, discutindo sua eficiência, vantagens e limitações, com exemplos práticos e citações relevantes (Lambert, 2022).

Bubble Sort

O Bubble Sort é um dos algoritmos de ordenação mais simples, porém menos eficiente para grandes conjuntos de dados. Sua lógica consiste em repetidamente percorrer o vetor, comparar elementos adjacentes e trocá-los de lugar se estiverem na ordem errada. Esse processo se repete até que o vetor esteja ordenado. A eficiência do Bubble Sort é geralmente

$$O(n^2)$$

onde n é o número de elementos no vetor (Szwarcfiter; Markenzon, 2020).

$$O(n^2)$$

Exemplo:

Se tivermos um vetor [5, 3, 8, 4, 2], o Bubble Sort fará várias passagens para ordená-lo, resultando em [2, 3, 4, 5, 8] após as trocas necessárias.

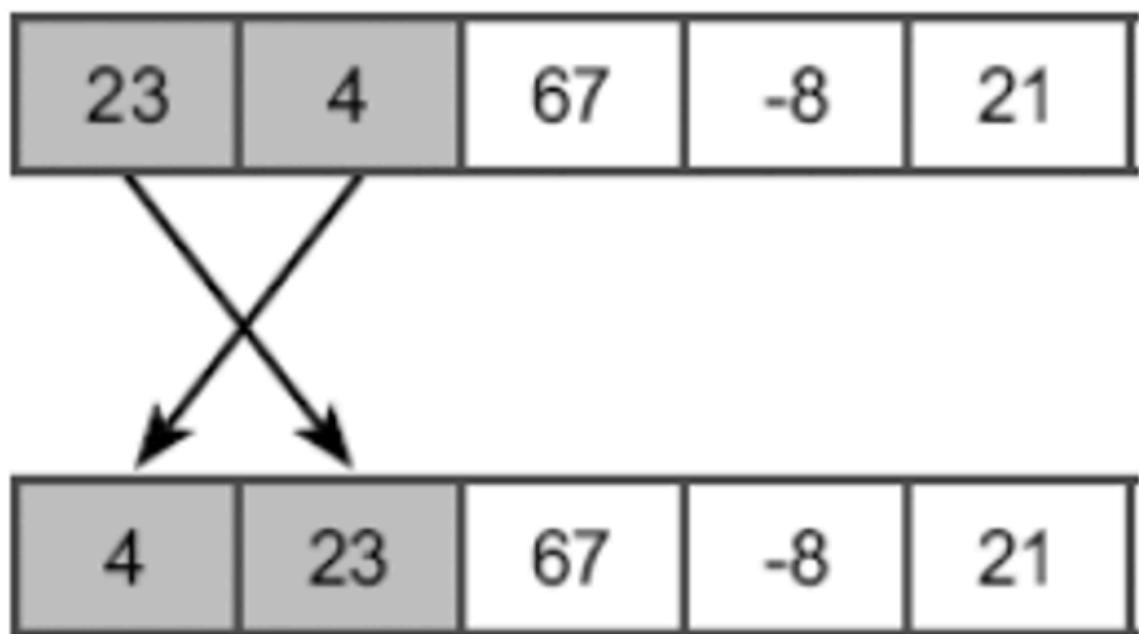


Figura 1 | Bubble Sort. Fonte: adaptada de Backes (2023).

Quick Sort

O Quick Sort é um algoritmo de ordenação muito mais eficiente que o Bubble Sort, especialmente para grandes conjuntos de dados. Utiliza uma abordagem de divisão e conquista, escolhendo um "pivô", e particionando o vetor em elementos menores que o pivô e elementos maiores que o pivô. O processo é repetido recursivamente para cada partição. A eficiência média do Quick Sort é

$$O(n \log n)$$

$$O(n \log n)$$

Exemplo:

Considerando o vetor [5, 3, 8, 4, 2], o Quick Sort pode escolher 5 como pivô e rearranjar o vetor em [3, 4, 2, 5, 8], seguindo com a ordenação das partições menores.

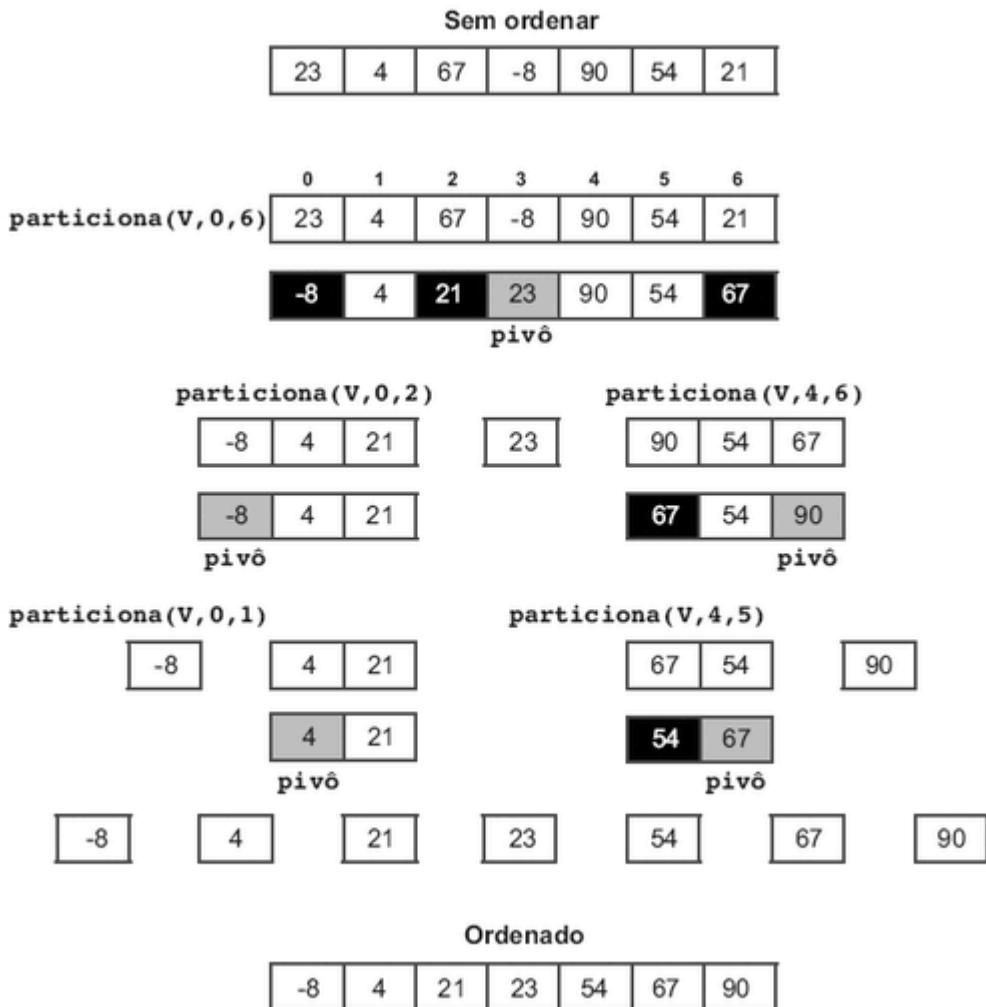


Figura 2 | Quick Sort. Fonte: adaptada de Backes (2023).

Merge Sort

O Merge Sort também segue a abordagem de divisão e conquista, dividindo o vetor em metades até que cada “*subvetor*” tenha apenas um elemento. Em seguida, ele combina (merge) esses “*subvetores*” de forma ordenada. A eficiência do Merge Sort é $O(n \log n)$, tornando-o altamente eficiente para grandes volumes de dados (Lambert, 2022).

$$O(n \log n)$$

Exemplo:

Para o vetor [5, 3, 8, 4, 2], o Merge Sort dividirá repetidamente o vetor até que tenhamos vetores de um elemento, para então combiná-los ordenadamente.

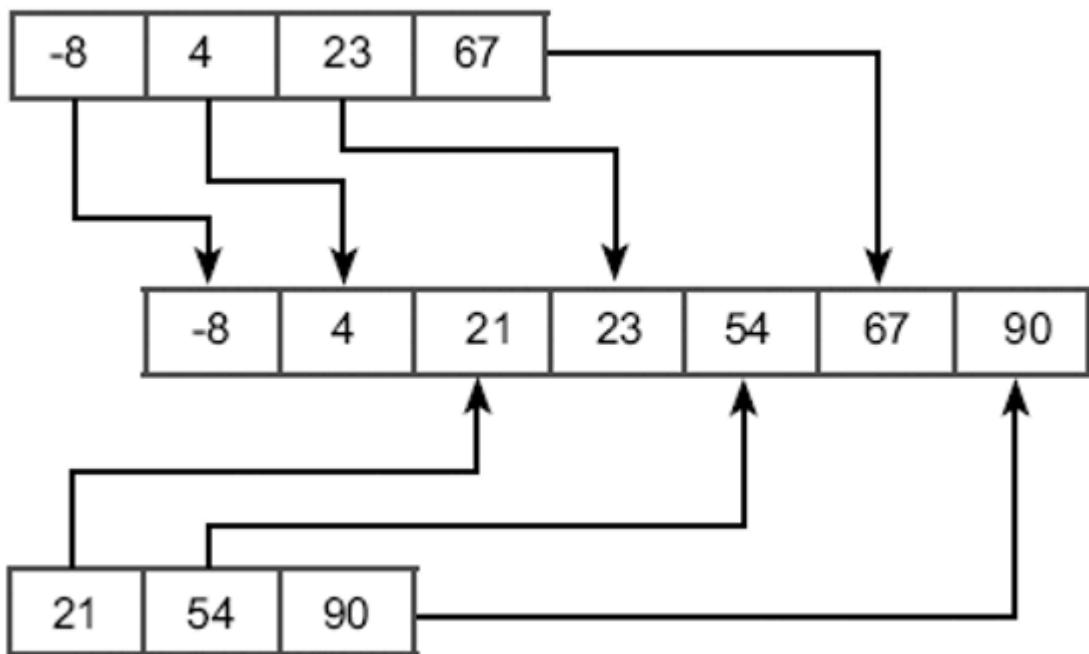


Figura 3 | Merge Sort. Fonte: adaptada de Backes (2023).

Insertion Sort

O Insertion Sort é particularmente eficiente para conjuntos de dados pequenos ou quase ordenados. Ele funciona "inserindo" cada elemento na posição correta do vetor já ordenado. Embora sua eficiência geral seja

$$O(n^2)$$

em conjuntos pequenos ou quase ordenados, ele pode ser mais rápido que algoritmos

$$O(n \log n)$$

devido à sua simplicidade (Lambert, 2022).

$$O(n^2)$$

$$O(n \log n)$$

Exemplo:

Com o vetor [5, 3, 8, 4, 2], o Insertion Sort começa com 5 e insere cada elemento subsequente na posição correta, resultando em [2, 3, 4, 5, 8].

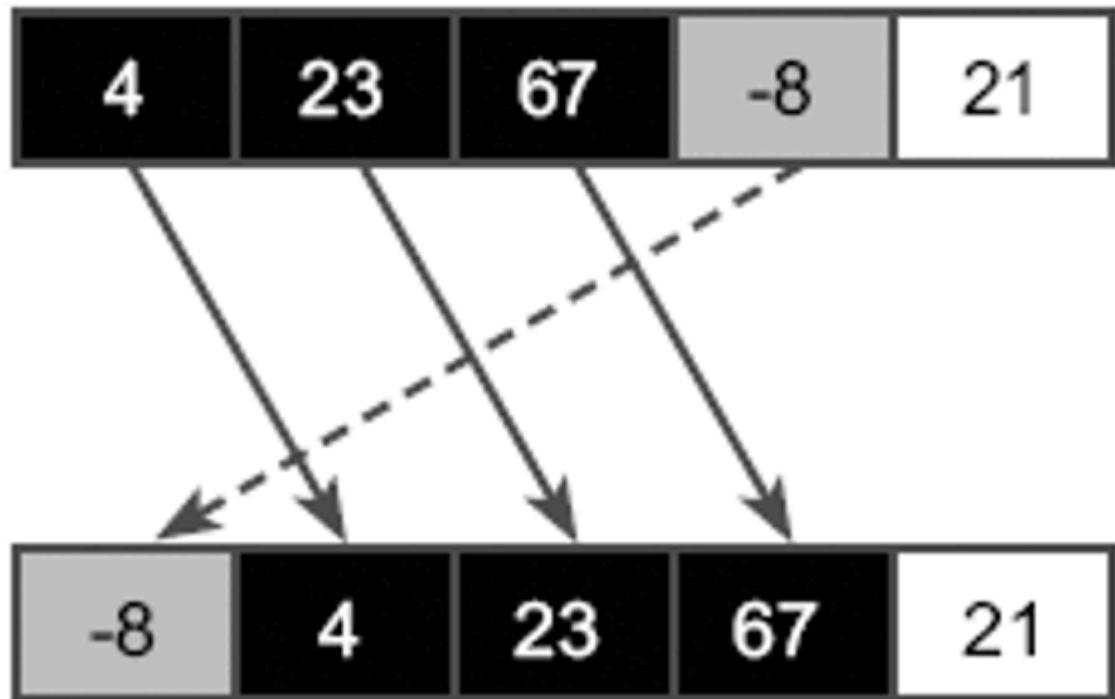


Figura 4 | Insertion Sort. Fonte: adaptada de Backes (2023).

Em síntese, algoritmos como o Quick Sort e o Merge Sort são preferíveis para grandes conjuntos de dados devido à sua eficiência

$$O(n \log n)$$

enquanto o Bubble Sort e o Insertion Sort podem ser úteis para dados pequenos ou quase ordenados. A compreensão dessas diferenças é essencial para a seleção do algoritmo mais eficiente para uma tarefa específica, otimizando assim o desempenho do aplicativo ou sistema.

$$O(n \log n)$$

Siga em Frente...

Busca binária e suas variações

A busca binária é um algoritmo eficiente para encontrar um elemento em um vetor ordenado. Sua eficiência vem da abordagem de "divisão e conquista" que utiliza, reduzindo significativamente o

número de comparações necessárias para encontrar um elemento em comparação com a busca linear. Iremos explorar o conceito de busca binária, suas variações e exemplos práticos para entender melhor sua eficiência (Lambert, 2022).

Busca binária

A busca binária começa comparando o elemento central do vetor ordenado com o valor de busca. Se o valor de busca for igual ao elemento central, a busca termina com sucesso. Se o valor de busca for menor, a busca continua na metade inferior do vetor; se for maior, na metade superior. Esse processo se repete até que o valor seja encontrado ou que a subseção se torne vazia (Szwarcfiter; Markenzon, 2020).

A eficiência da busca binária é
 $O(\log n)$

onde n é o número de elementos no vetor. Isso significa que, mesmo para grandes conjuntos de dados, o número de comparações necessárias para encontrar um elemento cresce muito lentamente à medida que o tamanho do conjunto de dados aumenta.

$O(\log n)$

Exemplo:

Considere um vetor ordenado [10, 20, 30, 40, 50, 60] e a busca pelo valor '30'. A busca binária começará comparando '30' com o elemento central '40'. Como '30' é menor, a busca continua na metade inferior '[10, 20, 30]'. O processo se repete até que '30' seja encontrado.

	0	1	2	3	4	5	6	7	8	9
Buscar '23'	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	9
23 > 16	2	5	8	12	16	23	38	56	72	91
Busque pela metade à direita	0	1	2	3	4	L=5	6	M=7	8	H=9
23 < 56	2	5	8	12	16	23	38	56	72	91
Busque pela metade à esquerda	0	1	2	3	4	L=5, M=5	H=6	7	8	9
Encontrado, retorne a posição (5)	2	5	8	12	16	23	38	56	72	91

Figura 5 | Busca binária. Fonte: adaptada de Binary (2023).

Variações da busca binária

Busca binária em vetores com valores duplicados: quando o vetor contém valores duplicados, a busca binária pode ser adaptada para encontrar a primeira ou a última ocorrência de um valor. Isso é feito ajustando a condição de parada e a direção da busca quando valores iguais são encontrados.

Busca binária em limites desconhecidos

Em casos onde o tamanho do vetor não é conhecido ou é infinito, a busca binária pode ser adaptada para identificar os limites superior e inferior dinamicamente, aumentando a faixa de busca exponencialmente até encontrar o intervalo que contém o valor de busca.

Busca binária em matrizes bidimensionais ordenadas

Quando aplicada a matrizes ordenadas (por linha e coluna), a busca binária pode ser adaptada para trabalhar com duas dimensões, selecionando pontos médios tanto nas linhas quanto nas colunas para encontrar o elemento desejado.

Logo, a busca binária e suas variações oferecem métodos eficientes para encontrar elementos em conjuntos de dados ordenados, adaptando-se a diferentes necessidades e estruturas de dados. Sua eficiência

$$O(\log n)$$

a torna uma escolha superior para busca em grandes conjuntos de dados em comparação com algoritmos de busca linear

$$O(n)$$

$$O(\log n)$$

$$O(n)$$

Aplicações práticas e otimizações

Os algoritmos de ordenação e a busca binária são fundamentais em computação e têm aplicações práticas em diversas áreas. Eles são usados para resolver problemas do mundo real, como organização de dados, pesquisa rápida de informações e otimização de processos. Além disso, há várias maneiras de otimizar esses algoritmos para torná-los mais eficientes. Vamos explorar algumas dessas aplicações e otimizações (Lambert, 2022).

Aplicações práticas

Bancos de dados: em bancos de dados, a ordenação é importante para indexação, o que permite buscas rápidas. A busca binária é frequentemente aplicada em índices ordenados para encontrar registros rapidamente.

Engenharia de software: algoritmos de ordenação são usados para gerenciar dependências em sistemas de *build*, como ordenar tarefas em um *makefile*.

Sistemas de recomendação: plataformas como Netflix e Amazon usam algoritmos de ordenação para classificar produtos ou filmes com base em certos critérios, como popularidade ou correspondência com o perfil do usuário.

Processamento de Linguagem Natural (PLN): algoritmos de busca são aplicados para encontrar palavras em dicionários de forma eficiente, o que é crucial para correção ortográfica e análise sintática.

Computação gráfica: em gráficos 3D, a ordenação é usada no algoritmo de pintor, que determina a ordem em que as superfícies são “*renderizadas*”.

Otimizações em algoritmos de ordenação

- Introsort: combinação de Quick Sort e Heap Sort. O Introsort começa com Quick Sort e muda para Heap Sort quando a profundidade da recursão excede um nível que depende do logaritmo do número de elementos sendo ordenados.
- Timsort: baseado em Merge Sort e Insertion Sort. É otimizado para aproveitar os padrões existentes nos dados, tornando-o eficiente para dados parcialmente ordenados.
- Radix Sort: em vez de comparações, utiliza a representação dos números para ordená-los. Isso é especialmente útil para ordenar inteiros ou *strings* e pode ser mais rápido que os algoritmos baseados em comparação.

Otimizações na busca binária

- *Exponential search*: antes de realizar uma busca binária, começa com uma busca exponencial para encontrar os limites onde o elemento pode estar. Isso pode ser mais rápido quando o elemento está próximo do início do vetor.
- Busca binária interpolada: em vez de escolher o meio do vetor, escolhe uma posição provável com base na distribuição dos valores. Isso pode ser mais eficiente quando os elementos estão distribuídos uniformemente.
- Busca binária em árvores binárias de busca (ABBs): em vez de vetores, a busca binária pode ser aplicada em ABBs, onde cada nó tem um valor, um filho à esquerda com valores menores e um filho à direita com valores maiores.

As aplicações práticas dos algoritmos de ordenação e busca binária são diversas e permeiam muitos aspectos da tecnologia e pesquisa. A escolha e otimização de um algoritmo dependem do contexto específico, como tamanho do conjunto de dados, sua distribuição e a frequência de buscas. Compreender e aplicar essas otimizações pode levar a melhorias significativas na eficiência e no desempenho dos sistemas computacionais.

Vamos Exercitar?

O estudo de caso apresentado para discussão na aula envolve a aplicação de algoritmos de ordenação e busca binária em um sistema de *e-commerce* para melhorar a eficiência da recuperação de produtos e a experiência do usuário durante a pesquisa de itens.

Você deve portanto identificar os principais gargalos de desempenho na recuperação e ordenação de produtos. Por exemplo, lentidão na busca de produtos, uso ineficiente de memória, tempo de carregamento elevado para listagens de produtos e ordenação ineficaz que afeta a usabilidade.

Insights para a solução:

- **Seleção de algoritmos de ordenação apropriados:** baseado no problema identificado, elabore um texto argumentando sobre a escolha de um algoritmo de ordenação apropriado. Por exemplo, se os produtos são adicionados frequentemente e as listagens não são muito grandes, o Insertion Sort pode ser adequado. Para listagens maiores e menos frequentemente atualizadas, algoritmos como Quick Sort ou Merge Sort podem ser mais eficientes.
- **Implementação da busca binária:** demonstre como implementar a busca binária para otimizar a recuperação de produtos, considerando que a lista de produtos está ordenada. Aborde variações da busca binária se o estudo de caso apresentar requisitos específicos, como a busca por um intervalo de preços.
- **Otimizações aplicadas:** proponha otimizações baseadas no contexto do estudo de caso, como a utilização de busca binária interpolada para conjuntos de dados uniformemente distribuídos ou a paralelização de algoritmos de ordenação em um ambiente de servidor com múltiplos núcleos.
- **Análise de resultados:** analise os resultados esperados com as mudanças propostas, como a melhora no tempo de busca e na eficiência de memória e como isso afeta a experiência do usuário final.

Nota: é importante que você teste sua solução com conjuntos de dados reais ou simulados para validar suas hipóteses. A atividade prática deve ser documentada em detalhe para facilitar sua revisão e seu estudo.

Saiba mais

Eficiência dos algoritmos de ordenação:

- LOBIANCO, C. [Eficiência de algoritmos: ordenando com Bubble Sort, Selection Sort e Random Sort](#). Blog Turing Talks.

Busca binária e suas variações:

- PANTUZA, G. [Busca binária](#). Blog Pantuza.

Aplicações práticas e otimizações:

- ARAUJO, R. [Busca local e otimização](#). Blog Ricardo Matsumura.

Referências

BACKES, A. R. **Algoritmos e estruturas de dados em Linguagem C**. Rio de Janeiro: LTC, 2023.

BINARY Search in Java. **Geek4Geeks**, 2023. Disponível em:

<https://www.geeksforgeeks.org/binary-search-in-java/>. Acesso em: 17 fev. 2024.

LAMBERT, K. K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.

Aula 5

ESTRUTURAS DE DADOS AVANÇADAS E ANÁLISE DE DADOS

Videoaula de Encerramento

Este conteúdo é um vídeo!



Para assistir este conteúdo é necessário que você acesse o AVA pelo computador ou pelo aplicativo. Você pode baixar os vídeos direto no aplicativo para assistir mesmo sem conexão à internet.

Dica para você

Aproveite o acesso para baixar os slides do vídeo, isso pode deixar sua aprendizagem ainda mais completa.

Nesta videoaula, concluímos nossa jornada através da unidade, abordando tópicos essenciais como *maps* e *hash*, análise de dados estruturados, a arte da análise de algoritmos e estruturas de dados e, por fim, algoritmos avançados de ordenação e busca. Você aprenderá a integrar esses conceitos para projetar e implementar soluções eficientes em Python, essenciais para qualquer profissional de tecnologia visando otimizar o armazenamento, a recuperação de dados e o desempenho de sistemas computacionais.

Bons estudos!

Ponto de Chegada

Olá, estudante! Para desenvolver a competência desta Unidade, que é conhecer e ser capaz de entender e projetar um banco de dados baseado em grafos, *maps* e *hash*, além de analisar a complexidade e eficiência de estrutura de dados, é essencial mergulhar nos conceitos fundamentais de *maps* e *hash map*, compreendendo a relação chave-valor, a estrutura e a implementação de tabelas *hash* em Python. Esse conhecimento é o alicerce para projetar sistemas de armazenamento de dados eficientes e otimizados (Dias, 2021).

Avançando, você explorou a aplicação de grafos na análise de dados em Python, aprendendo sobre a base de dados orientada a grafos e como essa estrutura pode ser utilizada para melhorar a visualização e análise de dados complexos. A compreensão dessas aplicações é importante para entender como os dados podem ser interconectados de forma eficaz, facilitando consultas e análises (Lambert, 2022).

A introdução à análise de algoritmos e estruturas de dados abre caminho para o estudo das complexidades temporal e de espaço em estruturas de dados, bem como a eficiência de algoritmos com dados estruturados. Esse segmento destaca a importância de escolher a estrutura de dados correta e otimizar algoritmos para garantir a máxima eficiência (Szwarcfiter; Markenzon, 2020).

Por fim, ao examinar algoritmos de ordenação e busca avançados, você adquiriu a habilidade de avaliar a eficiência dos algoritmos de ordenação, compreender as variações da busca binária e aplicar otimizações práticas. Esses conhecimentos são fundamentais para desenvolver soluções que respondam às necessidades específicas de performance e eficiência.

É Hora de Praticar!

Desenvolvendo um sistema de cache com *maps* e *hashes*

Descrição do problema

Você é um desenvolvedor de software em uma empresa que lida com grandes volumes de requisições a um banco de dados que armazena informações sobre produtos de um *e-commerce*. O tempo de resposta para as consultas de produto tem se tornado um gargalo, afetando a experiência do usuário no site. Para mitigar esse problema, foi proposto o desenvolvimento de um sistema de cache que armazene temporariamente os produtos mais acessados, reduzindo o número de acessos diretos ao banco de dados e, consequentemente, o tempo de resposta.

Objetivo da atividade

ESTRUTURA DE DADOS

Utilize a estrutura de dados *map*, implementada através de *hashes*, para criar um sistema de cache simples em Python. Esse sistema deve ser capaz de armazenar informações sobre os produtos (como ID, nome, descrição e preço) e fornecer uma maneira rápida de acessá-los com base no ID do produto.

Requisitos

1. Estrutura do produto: defina uma classe **Produto** com atributos para ID, nome, descrição e preço.
2. Implementação do cache: utilize um dicionário em Python como sua estrutura de *hash* para implementar o cache. A chave deve ser o ID do produto, e o valor deve ser uma instância da classe **Produto**.
3. Funcionalidades do cache:
 - Inserção: permita a adição de produtos ao cache.
 - Consulta: desenvolva uma função para consultar produtos pelo ID. Se o produto estiver no cache, retorne os detalhes do produto. Caso contrário, simule uma busca no "banco de dados" (pode ser uma simples impressão em tela), armazene o resultado no cache e então retorne os detalhes do produto.
 - Remoção (opcional): implemente uma funcionalidade para remover produtos do cache baseando-se no ID.

Pergunta para reflexão

Após implementar o sistema, reflita e responda: Por que o uso de *hashes*, através de um *map*, é uma boa estratégia para o desenvolvimento de um sistema de cache? Considere os aspectos de tempo de acesso, complexidade e eficiência na sua resposta.

Agora, reflita sobre as seguintes questões para aprofundar sua compreensão dos conteúdos da unidade:

1. Como a escolha entre uma tabela *hash* e uma base de dados orientada a grafos pode impactar o desempenho de um sistema de gerenciamento de dados?
2. De que maneira a análise de complexidade de um algoritmo auxilia na escolha da estrutura de dados mais adequada para um determinado problema?
3. Quais são os critérios para decidir entre implementar uma busca binária ou utilizar um algoritmo de ordenação em um conjunto de dados específico?

Essas perguntas guiam sua reflexão sobre a importância de entender profundamente as estruturas de dados e algoritmos, capacitando-o a projetar soluções eficazes e eficientes para problemas computacionais complexos.

Estrutura do produto:

Definir uma classe **Produto** com atributos para ID (identificador único do produto), nome, descrição e preço.

```
class Produto:  
    def __init__(self, id, nome, descricao, preco):  
        self.id = id
```

```
self.nome = nome
self.descricao = descricao
self.preco = preco
```

Implementação do cache

Utilizar um dicionário em Python como estrutura de *hash* para implementar o cache. As chaves serão os IDs dos produtos, e os valores serão instâncias da classe **Produto**.

```
cache = {}
```

Funcionalidades do cache

Inserção:

Função para adicionar produtos ao cache.

```
def adicionar_produto_ao_cache(produto):
    cache[produto.id] = produto
```

Consulta:

Função para consultar produtos pelo ID. Se o produto estiver no cache, retorna os detalhes do produto; caso contrário, simula uma busca no "banco de dados", armazena no cache e retorna os detalhes.

```
def consultar_produto(id):
    if id in cache:
        return cache[id]
    else:
        print()
        # Simulação de busca no banco e criação do produto
        produto = Produto(id, , 100.00)
        adicionar_produto_ao_cache(produto)
    return produto
```

Remoção (opcional):

Função para remover produtos do cache baseando-se no ID.

```
def remover_produto_do_cache(id):
    if id in cache:
        del cache[id]
```

Pergunta para reflexão:

Por que o uso de *hashes*, através de um *map*, é uma boa estratégia para o desenvolvimento de um sistema de cache?

Resposta:

O uso de *hashes* implementados por meio de um *map* é uma estratégia eficiente para sistemas de cache devido ao acesso rápido que proporciona. As operações de inserção, consulta e remoção têm complexidade de tempo aproximadamente constante $O(1)$ em casos médios, o que significa que o tempo de acesso aos dados é rápido e não varia significativamente com o aumento do número de entradas no cache. Essa característica é essencial para melhorar a performance e a experiência do usuário em aplicações que lidam com grande volume de dados e

ESTRUTURA DE DADOS

requisições, como um *e-commerce*, garantindo um acesso eficiente e reduzindo a carga sobre o banco de dados principal.

Este infográfico é uma ferramenta didática que pode ajudar a visualizar e compreender melhor os conceitos abordados na unidade:



ESTRUTURAS DE DADOS AVANÇADAS E ANÁLISE DE DADOS

MAPS E HASH

Maps são estruturas de dados que associam chaves a valores, facilitando a busca rápida de dados. Hashing é o processo de conversão de uma grande quantidade de dados em uma pequena quantidade de informação que representa esses dados, tornando as operações de maps eficientes.

Hashing: transforma dados de tamanho variável em um tamanho fixo de saída, usando uma função hash. É essencial para busca rápida, segurança de dados e estruturas eficientes como tabelas hash



ANÁLISE DE DADOS ESTRUTURADOS

A análise de dados estruturados envolve o exame e a interpretação de dados organizados em formatos facilmente acessíveis, como tabelas ou bancos de dados. Essa análise permite extrair *insights* valiosos, apoiando a tomada de decisão baseada em evidências.



Exemplo: Bancos NoSQL com grafos armazenam dados em nós e arestas, representando relacionamentos complexos de maneira intuitiva. São ideais para redes sociais, sistemas de recomendação e análise de conexões.



INTRODUÇÃO A ANÁLISE DE ALGORITMOS E ESTRUTURAS DE DADOS

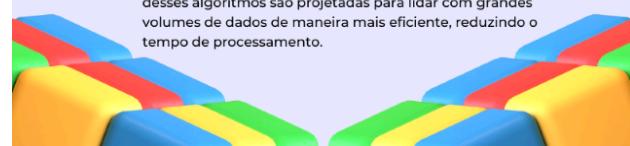


Esta área estuda como os algoritmos operam e utilizam recursos, como tempo e memória, para processar dados. Compreender esses conceitos ajuda a selecionar a estrutura de dados mais adequada e a otimizar algoritmos para resolver problemas de maneira eficiente.



ALGORITMOS DE ORDENAÇÃO E BUSCA AVANÇADOS

Algoritmos de ordenação organizam dados em uma sequência específica, enquanto algoritmos de busca localizam dados dentro de estruturas. Versões avançadas desses algoritmos são projetadas para lidar com grandes volumes de dados de maneira mais eficiente, reduzindo o tempo de processamento.



DIAS, M. A. **Análise de dados estruturados**. Estrutura de Dados, 2021. Disponível em: <https://bit.ly/49g36r9>. Acesso em: 17 fev. 2024.

ESTRUTURA DE DADOS

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cengage Learning, 2022.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2020.