

Tecnólogo Ciência de Dados

Guilherme Giacomini Teixeira

**LISTA ENCADEADA:**

Trabalho de Avaliação da Unidade 1 da Disciplina Estru- tura de Dados

Balneário Camboriú - SC 2025

Guilherme Giacomini Teixeira

**LISTA ENCADEADA:**

Trabalho de Avaliação da Unidade 1 da Disciplina Estru- tura de Dados

Trabalho de avaliação da unidade 1 da disciplina Estrutu- ra de Dados apresentado como requisito parcial para a obtenção da média no curso Ciência de Dados.

Professora: Vanessa Matias Leite

Tutor: João Henrique Correia dos Santos

Balneário Camboriú - SC 2025

**SUMÁRIO**

1. [INTRODUÇÃO 3](#_bookmark0)
2. [DESENVOLVIMENTO 4](#_bookmark1)
3. [RESULTADOS](#_bookmark2) 6
4. [CONCLUSÃO](#_bookmark3) 7
5. [REFERÊNCIAS](#_bookmark4) 8

# INTRODUÇÃO

Este relatório apresenta a atividade solicitada na disciplina Estrutura de Dados, realizada na Unidade 2, Aula 4, do curso de Ciência de Dados.

O objetivo da atividade foi aprender e implementar uma **lista encadeada** usan- do a linguagem de programação **Python**.

Neste trabalho, será implementada a função count\_nodes, que recebe uma lista encadeada como parâmetro e retorna o número de nós que ela contém. A ati- vidade também inclui a criação de uma lista encadeada, a adição de elementos, e a impressão da lista e da contagem de nós. Para a realização da atividade, foi utilizado o ambiente de desenvolvimento integrado (IDE) VS Code.

Neste relatório, serão descritos os passos seguidos para o desenvolvimento do projeto.

# DESENVOLVIMENTO

A linguagem Python é uma escolha popular e versátil, especialmente para a área de Ciên- cia de Dados, por sua sintaxe clara e legível. Ela é uma linguagem de programação de alto nível, interpretada, e com tipagem dinâmica e forte. Sua popularidade advém de sua facilidade de apren- dizado e da vasta quantidade de bibliotecas e frameworks disponíveis, que suportam desde o desenvolvimento web até a automação de tarefas e, de forma notável, a análise de dados e o aprendizado de máquina. No contexto da atividade, a escolha do Python se justifica por sua efi- ciência em lidar com estruturas de dados.

Para este projeto, foi utilizada a ferramenta VS Code para a implementação de uma lista encadeada. A atividade consistiu em implementar uma função chamada count\_nodes, que rece- be uma lista encadeada como parâmetro e retorna o número de nós presentes na lista. A função percorre a lista encadeada usando um loop para incrementar um contador. Ao final do percurso, o valor do contador é retornado.

Para a implementação, foram criadas as classes Node e LinkedList. Na classe LinkedList, foi implementado o método append para adicionar elementos à lista. Após adicionar os elemen- tos, a lista é impressa na tela e a função count\_nodes é chamada, exibindo o número de nós pre- sentes na lista.

A seguir, está o código da solução:

class Node:

def init (self, data): self.data = data self.next = None

class LinkedList: def init (self):

self.head = None

def append(self, data): new\_node = Node(data)

if self.head is None: self.head = new\_node return

last\_node = self.head while last\_node.next:

last\_node = last\_node.next last\_node.next = new\_node

def print\_list(self): current\_node = self.head while current\_node:

print(current\_node.data, end=" -> ") current\_node = current\_node.next

print("None")

# Implementar uma função para contar o número de nós em uma lista encadeada. def count\_nodes(linked\_list):

count = 0

current\_node = linked\_list.head while current\_node:

count += 1

current\_node = current\_node.next return count

# Exemplo de uso:

# Criar uma nova lista encadeada my\_list = LinkedList()

# Adicionar elementos à lista my\_list.append(1) my\_list.append(2) my\_list.append(3) my\_list.append(4)

# Imprimir a lista print("Lista encadeada:") my\_list.print\_list()

# Contar o número de nós num\_nodes = count\_nodes(my\_list)

print(f"Número de nós na lista: {num\_nodes}")

# RESULTADOS:

A atividade foi realizada com sucesso no ambiente de desenvolvimento VS Co- de, seguindo as etapas de implementação de uma lista encadeada em Python. O código desenvolvido foi capaz de criar a estrutura da lista, adicionar elementos a ela e, mais importante, contar o número de nós presentes.

A função count\_nodes implementada percorreu a lista encadeada do início ao fim, incrementando um contador a cada nó visitado. Ao final do percurso, o valor total do contador foi retornado, que correspondeu ao número de elementos adicionados à lista. O resultado final, impresso na tela do terminal, exibiu corretamente o número de nós, validando a funcionalidade da função e o entendimento do conceito por trás da estrutura de dados.

O desenvolvimento desta atividade prática solidificou o aprendizado sobre o funcionamento das listas encadeadas, mostrando como a manipulação de nós e re- ferências é fundamental para a sua operação. A implementação do código reforçou a teoria de que, para contar os elementos, é necessário percorrer a lista sequencial- mente, diferentemente de estruturas de dados que permitem acesso direto a qualquer elemento por meio de um índice.

# CONCLUSÃO

A atividade prática proporcionou a oportunidade de aplicar os conceitos teóri- cos de listas encadeadas, aprendidos na disciplina de Estrutura de Dados. A imple- mentação de um sistema de contagem de nós, usando a linguagem Python no am- biente de desenvolvimento VS Code, demonstrou a importância de entender como percorrer uma estrutura de dados de forma sequencial.

O projeto foi bem-sucedido, com a criação de um código funcional que atendeu a todos os requisitos propostos. A implementação da função count\_nodes validou o conhecimento sobre a travessia de uma lista encadeada e a lógica de contagem de seus elementos.

Em conclusão, este trabalho reforçou o aprendizado sobre as listas encadea- das e sua aplicação em problemas práticos, mostrando como o entendimento de es- truturas de dados é crucial para o desenvolvimento de softwares.

.

# REFERÊNCIAS

ALVES, W. P. **Programação Python: aprenda de forma rápida**. São Paulo: Ex- pressa, 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: Cen- gage Learning, 2022.

MARIANO, D. C. B. **Introdução à estrutura de dados em Python**. Estrutura de Da- dos, 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.

LEITE, V. M. **Listas Encadeadas**. Aula 2. In: UNOPAR ANHANGUERA. Fundamen- tos de Estruturas de Dados. [Material de curso]. Unidade 1, Fundamentos de Estru- turas de Dados. [S. l.]: Anhanguera Unopar, 2025. Acesso restrito.



Tecnólogo Ciência de Dados

Guilherme Giacomini Teixeira

**ÁRVORES AVL:**

Trabalho de Avaliação da Unidade 2 da Disciplina Estru- tura de Dados

Balneário Camboriú - SC 2025

Guilherme Giacomini Teixeira

**ÁRVORES AVL:**

Trabalho de Avaliação da Unidade 2 da Disciplina Estru- tura de Dados

Trabalho de avaliação da unidade 2 da disciplina Estrutu- ra de Dados apresentado como requisito parcial para a obtenção da média no curso Ciência de Dados.

Professora: Vanessa Matias Leite

Tutor: João Henrique Correia dos Santos

Balneário Camboriú - SC 2025

**SUMÁRIO**

1. [INTRODUÇÃO 3](#_bookmark5)
2. [DESENVOLVIMENTO 4](#_bookmark6)
3. [RESULTADOS](#_bookmark7) 6
4. [CONCLUSÃO](#_bookmark8) 7
5. [REFERÊNCIAS](#_bookmark9) 8
6. **INTRODUÇÃO**

Este relatório apresenta a atividade solicitada na disciplina Estrutura de Dados, realizada na Unidade 2, Aula 4, do curso de Ciência de Dados.

O objetivo da atividade foi implementar um sistema de gerenciamento de Poké- mon utilizando uma Árvore AVL usando uma linguagem de programação.

A árvore deve armazenar informações sobre cada Pokémon, como nome e va- lor de força. Além disso, o sistema deve ser capaz de realizar buscas por nome, re- mover Pokémon e listar todos os Pokémon em ordem decrescente de força. Neste relatório, serão descritos os passos seguidos para o desenvolvimento do projeto e a utilização do ambiente de desenvolvimento **VS Code**.

1. **DESENVOLVIMENTO**

A linguagem **Python** é uma escolha popular e versátil, especialmente para a área de Ciência de Dados, por sua sintaxe clara e legível. Ela é uma linguagem de programação de alto nível, interpretada, e com tipagem dinâmica e forte. Sua popu- laridade advém de sua facilidade de aprendizado e da vasta quantidade de bibliote- cas e frameworks disponíveis, que suportam desde o desenvolvimento web até a automação de tarefas e, de forma notável, a análise de dados e o aprendizado de máquina. No contexto da atividade, a escolha do Python se justifica pela sua eficiên- cia em lidar com estruturas de dados complexas como as Árvores AVL.

Para este projeto, foi utilizada a ferramenta **VS Code** para a implementação de uma Árvore AVL que gerencia informações de Pokémon, incluindo nome e força. O código desenvolvido contempla as funcionalidades de busca, remoção e listagem, garantindo que a estrutura da Árvore AVL seja mantida balanceada após cada ope- ração. A seguir, está o print da tela do VS Code com a implementação da solução.

Código do Projeto:

class NoDaArvore:

"""Representa um nó da Árvore AVL.""" def init (self, nome, forca):

self.nome = nome self.forca = forca self.left = None self.right = None self.height = 1

class ArvoreAVL:

"""Implementa a estrutura de dados Árvore AVL.""" def init (self):

self.raiz\_do\_no = None

def obter\_altura(self, no\_da\_arvore):

"""Retorna a altura de um nó. Retorna 0 se o nó for nulo.""" if not no\_da\_arvore:

return 0

return no\_da\_arvore.height

def obter\_balanceamento(self, no\_da\_arvore): """Calcula o fator de balanceamento de um nó.""" if not no\_da\_arvore:

return 0

return self.obter\_altura(no\_da\_arvore.left) - self.obter\_altura(no\_da\_arvo- re.right)

def rotacionar\_direita(self, no\_desbalanceado): """Realiza uma rotação para a direita.""" no\_ascendente = no\_desbalanceado.left subarvore\_em\_movimentacao = no\_ascendente.right

# Realiza a rotação

no\_ascendente.right = no\_desbalanceado no\_desbalanceado.left = subarvore\_em\_movimentacao

# Atualiza as alturas

no\_desbalanceado.height = 1 + max(self.obter\_altura(no\_desbalanceado.left), self.obter\_altura(no\_desbalanceado.right))

no\_ascendente.height = 1 + max(self.obter\_altura(no\_ascendente.left), self.ob- ter\_altura(no\_ascendente.right))

return no\_ascendente

def rotacionar\_esquerda(self, no\_desbalanceado): """Realiza uma rotação para a esquerda.""" no\_ascendente = no\_desbalanceado.right subarvore\_em\_movimentacao = no\_ascendente.left

# Realiza a rotação

no\_ascendente.left = no\_desbalanceado no\_desbalanceado.right = subarvore\_em\_movimentacao

# Atualiza as alturas

no\_desbalanceado.height = 1 + max(self.obter\_altura(no\_desbalanceado.left), self.obter\_altura(no\_desbalanceado.right))

no\_ascendente.height = 1 + max(self.obter\_altura(no\_ascendente.left), self.ob- ter\_altura(no\_ascendente.right))

return no\_ascendente

def inserir(self, no\_da\_arvore, nome, forca): """Insere um novo nó na árvore e a rebalanceia.""" if not no\_da\_arvore:

return NoDaArvore(nome, forca)

# A inserção é baseada na força para a ordenação if forca < no\_da\_arvore.forca:

no\_da\_arvore.left = self.inserir(no\_da\_arvore.left, nome, forca) elif forca > no\_da\_arvore.forca:

no\_da\_arvore.right = self.inserir(no\_da\_arvore.right, nome, forca) else: # Forças iguais, o critério pode ser o nome

if nome < no\_da\_arvore.nome:

no\_da\_arvore.left = self.inserir(no\_da\_arvore.left, nome, forca) elif nome > no\_da\_arvore.nome:

no\_da\_arvore.right = self.inserir(no\_da\_arvore.right, nome, forca) else:

return no\_da\_arvore # Pokémon já existe

# Atualiza a altura do nó pai

no\_da\_arvore.height = 1 + max(self.obter\_altura(no\_da\_arvore.left), self.ob- ter\_altura(no\_da\_arvore.right))

# Obtém o fator de balanceamento e realiza as rotações se necessário balance = self.obter\_balanceamento(no\_da\_arvore)

# Rotação LL

if balance > 1 and forca < no\_da\_arvore.left.forca: return self.rotacionar\_direita(no\_da\_arvore)

# Rotação RR

if balance < -1 and forca > no\_da\_arvore.right.forca: return self.rotacionar\_esquerda(no\_da\_arvore)

# Rotação LR

if balance > 1 and forca > no\_da\_arvore.left.forca: no\_da\_arvore.left = self.rotacionar\_esquerda(no\_da\_arvore.left) return self.rotacionar\_direita(no\_da\_arvore)

# Rotação RL

if balance < -1 and forca < no\_da\_arvore.right.forca: no\_da\_arvore.right = self.rotacionar\_direita(no\_da\_arvore.right) return self.rotacionar\_esquerda(no\_da\_arvore)

return no\_da\_arvore

def obter\_no\_de\_menor\_valor(self, no\_da\_arvore): """Encontra o nó com o menor valor na subárvore.""" if no\_da\_arvore is None or no\_da\_arvore.left is None:

return no\_da\_arvore

return self.obter\_no\_de\_menor\_valor(no\_da\_arvore.left)

def deletar(self, no\_da\_arvore, nome):

"""Remove um nó da árvore por nome e a rebalanceia.""" if not no\_da\_arvore:

return no\_da\_arvore

# Lógica de remoção para BST if nome < no\_da\_arvore.nome:

no\_da\_arvore.left = self.deletar(no\_da\_arvore.left, nome) elif nome > no\_da\_arvore.nome:

no\_da\_arvore.right = self.deletar(no\_da\_arvore.right, nome) else: # O nó a ser removido foi encontrado

if no\_da\_arvore.left is None: temp = no\_da\_arvore.right no\_da\_arvore = None return temp

elif no\_da\_arvore.right is None: temp = no\_da\_arvore.left no\_da\_arvore = None

return temp

temp = self.obter\_no\_de\_menor\_valor(no\_da\_arvore.right) no\_da\_arvore.nome = temp.nome

no\_da\_arvore.forca = temp.forca

no\_da\_arvore.right = self.deletar(no\_da\_arvore.right, temp.nome)

# Se a árvore tiver apenas um nó, não há necessidade de balanceamento if no\_da\_arvore is None:

return no\_da\_arvore

# Atualiza a altura do nó pai

no\_da\_arvore.height = 1 + max(self.obter\_altura(no\_da\_arvore.left), self.ob- ter\_altura(no\_da\_arvore.right))

# Obtém o fator de balanceamento e realiza as rotações se necessário balance = self.obter\_balanceamento(no\_da\_arvore)

# Rotação LL

if balance > 1 and self.obter\_balanceamento(no\_da\_arvore.left) >= 0: return self.rotacionar\_direita(no\_da\_arvore)

# Rotação LR

if balance > 1 and self.obter\_balanceamento(no\_da\_arvore.left) < 0: no\_da\_arvore.left = self.rotacionar\_esquerda(no\_da\_arvore.left) return self.rotacionar\_direita(no\_da\_arvore)

# Rotação RR

if balance < -1 and self.obter\_balanceamento(no\_da\_arvore.right) <= 0: return self.rotacionar\_esquerda(no\_da\_arvore)

# Rotação RL

if balance < -1 and self.obter\_balanceamento(no\_da\_arvore.right) > 0: no\_da\_arvore.right = self.rotacionar\_direita(no\_da\_arvore.right) return self.rotacionar\_esquerda(no\_da\_arvore)

return no\_da\_arvore

def buscar(self, no\_da\_arvore, nome):

"""Busca um Pokémon na árvore pelo nome."""

if not no\_da\_arvore or no\_da\_arvore.nome == nome: return no\_da\_arvore

if no\_da\_arvore.nome < nome:

return self.buscar(no\_da\_arvore.right, nome) return self.buscar(no\_da\_arvore.left, nome)

def listar\_em\_ordem(self, no\_da\_arvore):

"""Lista os Pokémon em ordem decrescente de força (travessia inversa).""" if no\_da\_arvore:

self.listar\_em\_ordem(no\_da\_arvore.right)

print(f"Nome: {no\_da\_arvore.nome}, Força: {no\_da\_arvore.forca}") self.listar\_em\_ordem(no\_da\_arvore.left)

1. **RESULTADOS:**

A atividade proposta foi concluída com sucesso. Através da implementação da Árvore AVL em Python, foi possível criar um sistema eficiente de gerenciamento de Pokémon. O código desenvolvido, seguindo as convenções de nomenclatura em por- tuguês, demonstrou a capacidade de manipular a estrutura de dados de forma eficaz.

As operações de inserção, busca, remoção e listagem foram implementadas de forma a manter o balanceamento da árvore, garantindo a complexidade de tempo logarítmica para cada uma delas, conforme exigido no problema. A funcionalidade de busca, por exemplo, permite localizar um Pokémon pelo nome, retornando seu valor de força. A listagem em ordem decrescente de força também foi corretamente imple- mentada, permitindo a visualização dos Pokémon mais fortes.

1. **CONCLUSÃO**

A atividade prática proporcionou a oportunidade de aplicar os conceitos teóri- cos de **Árvores AVL**, aprendidos na disciplina de **Estrutura de Dados**. A implemen- tação de um sistema de gerenciamento de Pokémon utilizando essa estrutura de da- dos, feita no ambiente de desenvolvimento **VS Code**, demonstrou a importância do rebalanceamento automático da árvore para a manutenção do desempenho.

O projeto foi bem-sucedido, com a criação de um código funcional que atendeu a todos os requisitos propostos. A implementação das funções de **inserção, busca, remoção e listagem** em ordem decrescente de força, com a garantia de que a com- plexidade de tempo se mantivesse logarítmica, confirmou a eficácia das Árvores AVL para a organização e manipulação de grandes volumes de dados de forma eficiente.

Em conclusão, este trabalho reforçou o aprendizado sobre as Árvores AVL e sua aplicação em problemas práticos, mostrando como estruturas de dados eficien- tes são cruciais para o desenvolvimento de softwares robustos e de alto desempenho.

.

1. **REFERÊNCIAS**

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

LAMBERT, K. A. **Fundamentos de Python:** estruturas de dados. São Paulo: Cen- gage 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.

LEITE, V. M. **Árvores AVL**. Aula 4. In: UNOPAR ANHANGUERA. *Estrutura de Dados*. [Material de curso]. Unidade 2. [S. l.]: Anhanguera Unopar, 2025. Acesso restrito.

LEITE, V. M. **Estrutura de dados árvores**. Aula 5. In: UNOPAR ANHANGUERA. *Estru- tura de Dados*. [Material de curso]. Unidade 2. [S. l.]: Anhanguera Unopar, 2025. Acesso res- trito.



Tecnólogo Ciência de Dados

Guilherme Giacomini Teixeira

**ALGORITMOS PARA GRAFOS EM PYTHON:**

Trabalho de Avaliação da Unidade 3 da Disciplina Estru- tura de Dados

Balneário Camboriú - SC 2025

Guilherme Giacomini Teixeira

**ALGORITMOS PARA GRAFOS EM PYTHON:**

Trabalho de Avaliação da Unidade 3 da Disciplina Estru- tura de Dados

Trabalho de avaliação da unidade 3 da disciplina Estrutu- ra de Dados apresentado como requisito parcial para a obtenção da média no curso Ciência de Dados.

Professora: Vanessa Matias Leite

Tutor: João Henrique Correia dos Santos

Balneário Camboriú - SC 2025

**SUMÁRIO**

1. [INTRODUÇÃO 3](#_bookmark10)
2. [DESENVOLVIMENTO 4](#_bookmark11)
3. [RESULTADOS](#_bookmark12) 6
4. [CONCLUSÃO](#_bookmark13) 7
5. [REFERÊNCIAS](#_bookmark14) 8
6. **INTRODUÇÃO**

Este relatório apresenta a atividade solicitada na disciplina de **Estrutura de Dados**, realizada na Unidade 3, Aula 4, do curso de **Ciência de Dados**. O objetivo da atividade foi implementar um algoritmo de caminho mínimo para grafos usando a lin- guagem de programação Python.

O projeto consiste em desenvolver uma aplicação para uma empresa de lo- gística, capaz de calcular a rota mais rápida entre dois pontos de entrega, conside- rando as diferentes distâncias entre eles. Para isso, é necessário implementar uma classe Grafo que contenha métodos para adicionar vértices e arestas com pesos. O grafo deve ser representado por listas de adjacência, e a classe precisa de um mé- todo para encontrar o caminho mais curto usando o algoritmo de Dijkstra.

Este relatório detalhará os passos seguidos para o desenvolvimento da apli- cação, incluindo a configuração do ambiente no VS Code e a implementação do códi- go para resolver o problema proposto.

1. **DESENVOLVIMENTO**

A linguagem Python é uma escolha popular e versátil para a área de Ciência de Dados, em grande parte devido à sua sintaxe clara e legível. É uma linguagem de programação de alto nível, interpretada e multiparadigma, que oferece suporte a pro- gramação orientada a objetos, funcional e procedural. Sua tipagem dinâmica e forte, juntamente com a vasta quantidade de bibliotecas e frameworks disponíveis, a tornam ideal para diversas aplicações, incluindo desenvolvimento web, automação, análise de dados e aprendizado de máquina.

No contexto deste projeto, o Python foi a linguagem escolhida para implemen- tar um algoritmo de caminho mínimo para grafos. A sua eficiência em lidar com es- truturas de dados complexas, como a representação de grafos por meio de listas de adjacência, justifica plenamente sua utilização para resolver o problema de encontrar a rota mais rápida em uma rede logística.

Para a implementação do projeto, a ferramenta VS Code foi utilizada para de- senvolver o programa. A atividade consistiu em criar uma aplicação que auxiliasse no mapeamento de rotas para uma nova empresa de logística, calculando a rota mais rápida entre dois pontos de entrega.

O projeto incluiu a criação de uma classe Grafo com métodos para adicionar vértices e arestas com pesos. O grafo foi representado usando listas de adjacência. O componente central da solução foi a implementação do algoritmo de Dijkstra para encontrar o caminho mais curto e o custo total entre um ponto de partida e um ponto de chegada fornecidos pelo usuário.

A seguir, está o código no VS Code com a implementação da solução em Py-

thon:

import heapq

# Classe para representar o Grafo class Grafo:

def init (self):

# O grafo é um dicionário onde as chaves são os vértices e os valores são dicionários de vizinhos

# {vertice: {vizinho: peso}} self.grafo = {}

def adicionar\_vertice(self, vertice):

# Adiciona um novo vértice ao grafo se ele ainda não existir if vertice not in self.grafo:

self.grafo[vertice] = {}

def adicionar\_aresta(self, origem, destino, peso):

# Adiciona aresta com peso. Se os vértices não existirem, eles são criados. self.adicionar\_vertice(origem)

self.adicionar\_vertice(destino)

# Aresta direcionada: origem -> destino self.grafo[origem][destino] = peso

def dijkstra(self, inicio, fim):

# Inicializa as distâncias de todos os vértices como infinito e a do início como 0 distancias = {vertice: float('inf') for vertice in self.grafo}

distancias[inicio] = 0

# Dicionário para rastrear o caminho mais curto caminhos = {vertice: [] for vertice in self.grafo} caminhos[inicio] = [inicio]

# Fila de prioridade (min-heap) para os vértices a serem visitados. # Armazena tuplas (distância, vértice)

fila\_prioridade = [(0, inicio)]

while fila\_prioridade:

distancia\_atual, vertice\_atual = heapq.heappop(fila\_prioridade)

# Se a distância atual for maior que a registrada, continue (já encontramos um caminho mais curto)

if distancia\_atual > distancias[vertice\_atual]: continue

# Se chegamos ao destino, o caminho mais curto foi encontrado if vertice\_atual == fim:

return caminhos[fim], distancias[fim]

# Explora os vizinhos do vértice atual

for vizinho, peso in self.grafo[vertice\_atual].items(): distancia = distancia\_atual + peso

# Se um caminho mais curto for encontrado, atualiza a distância e o caminho if distancia < distancias[vizinho]:

distancias[vizinho] = distancia

caminhos[vizinho] = caminhos[vertice\_atual] + [vizinho] heapq.heappush(fila\_prioridade, (distancia, vizinho))

# Se o destino não for alcançável a partir do início, retorna None

return None, float('inf')

# Função principal para executar o programa def main():

# Instancia o grafo grafo\_logistica = Grafo()

# Adiciona os vértices e arestas com os pesos, conforme o exemplo do problema. # Vértices: A, B, C, D, E

# Arestas: A->B(4), A->C(2), B->C(5), B->D(10), C->E(3), D->E(4), E->A(7)

grafo\_logistica.adicionar\_aresta('A', 'B', 4)

grafo\_logistica.adicionar\_aresta('A', 'C', 2)

grafo\_logistica.adicionar\_aresta('B', 'C', 5)

grafo\_logistica.adicionar\_aresta('B', 'D', 10)

grafo\_logistica.adicionar\_aresta('C', 'E', 3)

grafo\_logistica.adicionar\_aresta('D', 'E', 4)

grafo\_logistica.adicionar\_aresta('E', 'A', 7)

# Recebe do usuário os pontos de partida e chegada ponto\_partida = input("Digite o ponto de partida (ex: A): ").upper()

ponto\_chegada = input("Digite o ponto de chegada (ex: E): ").upper()

# Valida se os pontos de partida e chegada existem no grafo

if ponto\_partida not in grafo\_logistica.grafo or ponto\_chegada not in grafo\_logistica.grafo:

print("Um ou ambos os pontos não existem no grafo. Por favor, verifique e tente novamen-

te.")

return

# Utiliza o algoritmo de Dijkstra para encontrar o caminho mais curto caminho, custo = grafo\_logistica.dijkstra(ponto\_partida, ponto\_chegada)

# Imprime o resultado if caminho:

print(f"O caminho mais curto é: {' -> '.join(caminho)} com custo total de {custo}.") else:

print(f"Não há caminho do ponto {ponto\_partida} para o ponto {ponto\_chegada}.")

# Executa a função principal if name == " main ":

main()

1. **RESULTADOS:**

A atividade proposta, que envolveu a implementação de um algoritmo de cami- nho mínimo para grafos, permitiu a aplicação prática de conceitos teóricos de Estru- tura de Dados. O objetivo foi desenvolver uma solução em Python capaz de calcular a rota mais rápida entre dois pontos de entrega para uma empresa de logística, uti- lizando o algoritmo de Dijkstra.

A implementação, conforme o código desenvolvido, demonstrou ser eficaz na representação de um grafo por meio de listas de adjacência. A classe Grafo e o al- goritmo de Dijkstra foram capazes de receber um conjunto de dados estático, proces- sar os vértices e arestas com seus respectivos pesos e, a partir dos pontos de partida e chegada informados pelo usuário, determinar o caminho mais curto e o seu custo total.

A solução validou o exemplo fornecido na atividade , onde, para um grafo com os vértices A, B, C, D, E e arestas com pesos específicos, a rota mais curta de "A" para "E" foi corretamente identificada como "A -> C -> E", com um custo total de 5. Em conclusão, o projeto cumpriu todos os requisitos, demonstrando a capacidade do alu- no em aplicar os conceitos de grafos e o algoritmo de Dijkstra para resolver um pro- blema de otimização de rotas de forma eficiente.

1. **CONCLUSÃO**

A atividade prática proporcionou a oportunidade de aplicar os conceitos teóri- cos de grafos e o algoritmo de Dijkstra, aprendidos na disciplina de Estrutura de Da- dos. A implementação de um sistema de mapeamento de rotas para uma empresa de logística, feita no ambiente de desenvolvimento VS Code, demonstrou a importância de algoritmos de caminho mínimo para a otimização de processos e a tomada de decisões.

O projeto foi bem-sucedido, com a criação de um código funcional que atendeu a todos os requisitos propostos. A implementação da classe Grafo, a representação dos dados por listas de adjacência e a aplicação do algoritmo de Dijkstra para encon- trar o caminho mais curto confirmaram a eficácia dessas estruturas de dados e algo- ritmos para a organização e manipulação de informações de forma eficiente.

Em conclusão, este trabalho reforçou o aprendizado sobre grafos e sua aplica- ção em problemas práticos, mostrando como estruturas de dados eficientes são cru- ciais para o desenvolvimento de softwares robustos e de alto desempenho na área de Ciência de Dados.

1. **REFERÊNCIAS**

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

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

LAMBERT, K. A. **Fundamentos de Python**: estruturas de dados. São Paulo: Cenga- ge 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.](http://bit.ly/497p4MU) Acesso em: 6 fev. 2024.

LEITE, V. M. **Algoritmos para grafos em Python**. Aula 4. In: UNOPAR ANHAN- GUERA. Estrutura de Dados. [Material de curso]. Unidade 3, Grafos e suas Opera- ções. [S. l.]: Anhanguera Unopar, 2025. Acesso restrito.



Tecnólogo Ciência de Dados

Guilherme Giacomini Teixeira

**ANALISE DE DADOS ESTRUTURADOS:**

Trabalho de Avaliação da Unidade 4 da Disciplina Estru- tura de Dados

Balneário Camboriú – SC 2025

Guilherme Giacomini Teixeira

**ANALISE DE DADOS ESTRUTURADOS:**

Trabalho de Avaliação da Unidade 4 da Disciplina Estru- tura de Dados

Trabalho de avaliação da unidade 4 da disciplina Estrutu- ra de Dados apresentado como requisito parcial para a obtenção da média no curso Ciência de Dados.

Professora: Vanessa Matias Leite

Tutor: João Henrique Correia dos Santos

Balneário Camboriú - SC 2025

**SUMÁRIO**

1. [INTRODUÇÃO 3](#_bookmark15)
2. [DESENVOLVIMENTO 4](#_bookmark16)
3. [RESULTADOS](#_bookmark17) 6
4. [CONCLUSÃO](#_bookmark18) 7
5. [REFERÊNCIAS](#_bookmark19) 8
6. **INTRODUÇÃO**

Este relatório apresenta a atividade solicitada na disciplina de **Estrutura de Dados**, realizada na **Unidade 4, Aula 2** do curso de **Ciência de Dados**.

O objetivo da atividade foi aprender a realizar uma implementação para a análi- se de dados estruturados utilizando a linguagem de programação Python. A tarefa proposta consiste em modelar uma rede social como um grafo usando a biblioteca NetworkX , além de implementar uma classe para gerenciar usuários e suas cone- xões.

O projeto também envolve a utilização de algoritmos de detecção de comuni- dades e o cálculo de métricas de centralidade para entender a influência dos usuários na rede. Para o desenvolvimento, será utilizada a ferramenta **VSCode**, que permite a execução de código Python de forma eficiente.

Neste relatório, serão descritos os passos seguidos para o desenvolvimento do projeto, detalhando a criação das classes, a implementação dos métodos solici- tados e a análise dos resultados obtidos.

1. **DESENVOLVIMENTO**

A linguagem Python é amplamente utilizada em Ciência de Dados por sua sin- taxe clara e legível. É uma linguagem de alto nível, interpretada e multiparadigma, com um vasto ecossistema de bibliotecas. Para esta atividade, o Python foi escolhido por sua capacidade de lidar com estruturas de dados complexas, como os grafos, e por sua eficiência na análise de redes.

O projeto foi implementado utilizando o VSCode para criar um programa que modela uma rede social como um grafo. A solução inclui as seguintes classes e fun- cionalidades:

Classe SocialNetwork: Gerencia o grafo da rede social, armazenando usuários e suas conexões. Possui métodos para adicionar e remover usuários, conectar e des- conectar conexões, além de encontrar comunidades e calcular a centralidade dos usuários.

Classe User: Representa um usuário individual com um identificador único (user\_id) e dados associados.

A atividade também exige um script de teste para demonstrar a funcionalidade das classes, realizando operações como a adição de usuários, a criação de conexões e a análise de métricas da rede.

A seguir, o código no VSCode com a implementação do projeto:

import networkx as nx

# Classe para representar um usuário class User:

def init (self, user\_id, data=None): self.user\_id = user\_id

self.data = data if data is not None else {}

# Método para obter o ID do usuário def get\_user\_id(self):

return self.user\_id

# Método para atualizar os dados do usuário def set\_data(self, data):

self.data.update(data)

# Classe para a rede social class SocialNetwork:

def init (self):

# Inicializa um grafo não direcionado da biblioteca NetworkX self.graph = nx.Graph()

# Adiciona um usuário ao grafo

def add\_user(self, user\_id, user\_data=None): if not self.graph.has\_node(user\_id):

self.graph.add\_node(user\_id, data=user\_data) print(f"Usuário {user\_id} adicionado.")

else:

print(f"Usuário {user\_id} já existe.")

# Remove um usuário e suas conexões do grafo def remove\_user(self, user\_id):

if self.graph.has\_node(user\_id):

self.graph.remove\_node(user\_id) print(f"Usuário {user\_id} removido.")

else:

print(f"Usuário {user\_id} não encontrado.")

# Conecta dois usuários

def connect\_users(self, user1\_id, user2\_id):

if self.graph.has\_node(user1\_id) and self.graph.has\_node(user2\_id): if not self.graph.has\_edge(user1\_id, user2\_id):

self.graph.add\_edge(user1\_id, user2\_id) print(f"Usuários {user1\_id} e {user2\_id} conectados.")

else:

print(f"Usuários {user1\_id} e {user2\_id} já estão conectados.")

else:

print("Um ou ambos os usuários não existem.")

# Desconecta dois usuários

def disconnect\_users(self, user1\_id, user2\_id): if self.graph.has\_edge(user1\_id, user2\_id):

self.graph.remove\_edge(user1\_id, user2\_id) print(f"Usuários {user1\_id} e {user2\_id} desconectados.")

else:

print(f"Conexão entre {user1\_id} e {user2\_id} não existe.")

# Encontra comunidades no grafo def find\_communities(self):

# Utiliza o algoritmo de Louvain para detecção de comunidades

communities = nx.algorithms.community.louvain\_communities(self.graph, seed=42) print("Comunidades encontradas:")

for i, comm in enumerate(communities): print(f"Comunidade {i+1}: {comm}")

return communities

# Calcula a centralidade dos usuários

def user\_centralities(self, method='degree'): if method == 'degree':

centrality = nx.degree\_centrality(self.graph) elif method == 'betweenness':

centrality = nx.betweenness\_centrality(self.graph) elif method == 'closeness':

centrality = nx.closeness\_centrality(self.graph) else:

print("Método de centralidade inválido.") return {}

print(f"Centralidade ({method}) dos usuários:") for user, score in centrality.items():

print(f"Usuário {user}: {score:.4f}") return centrality

# Analisa um subgrafo

def analyze\_subgraph(self, user\_ids):

if all(self.graph.has\_node(uid) for uid in user\_ids): subgraph = self.graph.subgraph(user\_ids)

print(f"Análise do subgrafo com os usuários {user\_ids}:") print(f"Número de nós: {subgraph.number\_of\_nodes()}") print(f"Número de arestas: {subgraph.number\_of\_edges()}") return subgraph

else:

print("Um ou mais usuários não existem no grafo principal.") return None

# Script de teste para demonstrar a funcionalidade if name == " main ":

social\_net = SocialNetwork()

# Adicionando usuários

social\_net.add\_user("Alice", {"name": "Alice Silva", "interests": "Python, Data Science"}) social\_net.add\_user("Bob", {"name": "Bob Souza", "interests": "Machine Learning, AI"}) social\_net.add\_user("Charlie", {"name": "Charlie Santos", "interests": "Web Development"}) social\_net.add\_user("David", {"name": "David Costa", "interests": "Data Science, Statistics"}) social\_net.add\_user("Eve", {"name": "Eve Ferreira", "interests": "Python, Algorithms"}) social\_net.add\_user("Frank", {"name": "Frank Lima", "interests": "Data Visualization"})

print("-" \* 20)

# Conectando usuários social\_net.connect\_users("Alice", "Bob") social\_net.connect\_users("Alice", "David") social\_net.connect\_users("Bob", "David") social\_net.connect\_users("Bob", "Eve")

social\_net.connect\_users("Charlie", "Frank") social\_net.connect\_users("David", "Eve") social\_net.connect\_users("Alice", "Eve")

print("-" \* 20)

# Identificando comunidades social\_net.find\_communities()

print("-" \* 20)

# Calculando centralidades social\_net.user\_centralities(method='degree') social\_net.user\_centralities(method='betweenness') social\_net.user\_centralities(method='closeness')

print("-" \* 20)

# Analisando subgrafo social\_net.analyze\_subgraph(["Alice", "Bob", "David"])

print("-" \* 20)

# Removendo um usuário e desconectando usuários social\_net.remove\_user("Frank") social\_net.disconnect\_users("Alice", "Eve")

1. **RESULTADOS:**

Nesta atividade, o projeto foi implementado com sucesso utilizando o VSCode e a linguagem Python, seguindo a estrutura de classes proposta. A solução demons- trou de forma prática como as estruturas de dados de grafos podem ser aplicadas para modelar e analisar redes sociais.

Através do código, foi possível simular a adição e remoção de usuários e suas conexões, o que é fundamental para a representação dinâmica de uma rede. A uti- lização da biblioteca NetworkX simplificou a manipulação do grafo, permitindo a apli- cação de algoritmos complexos.

A análise do código-fonte e dos resultados obtidos no script de teste mostrou que as classes SocialNetwork e User foram capazes de gerenciar eficientemente os dados e as interações.

Os algoritmos de detecção de comunidades, como o de Louvain, permitiram identificar subgrupos de usuários com interesses ou conexões em comum. O cálculo de métricas de centralidade (grau, intermediação, proximidade) revelou a influência de cada usuário na rede, indicando quais seriam os nós mais importantes ou influen- tes.

O projeto demonstrou a importância de ferramentas como a biblioteca NetworkX e a linguagem Python para a Ciência de Dados, ao fornecer insights vali- osos sobre a estrutura e o comportamento de uma rede social.

1. **CONCLUSÃO**

A atividade prática proporcionou a oportunidade de aplicar os conceitos teóri- cos de análise de dados estruturados e grafos, aprendidos na disciplina de Estrutura de Dados. A implementação de um sistema de gerenciamento de rede social utilizan- do a biblioteca NetworkX e a linguagem Python no ambiente de desenvolvimento VS- Code demonstrou a importância dessas ferramentas para a modelagem e análise de interações complexas.

O projeto foi bem-sucedido, com a criação de um código funcional que atendeu a todos os requisitos propostos. A implementação das classes SocialNetwork e User, juntamente com os métodos para adicionar, remover e conectar usuários, foi funda- mental para simular o comportamento de uma rede social. A utilização de algoritmos de detecção de comunidades e o cálculo de métricas de centralidade confirmaram a eficácia dessa abordagem para extrair insights valiosos sobre a estrutura da rede e a influência de seus usuários.

Em conclusão, este trabalho reforçou o aprendizado sobre a aplicação de es- truturas de dados avançadas, como grafos, em problemas práticos. Ele mostrou co- mo a escolha de ferramentas e bibliotecas eficientes é crucial para o desenvolvimen- to de softwares robustos e para a análise de dados complexos de forma eficaz.

.

1. **REFERÊNCIAS**

ALVES, W. P. **Programação Python**: aprenda de forma rápida. São Paulo: Expres- sa, 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: Cenga- ge Learning, 2022.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmo**s.

3. ed. Rio de Janeiro: LTC, 2020.

LEITE, V. M. **Análise de dados estruturados**. Aula 2. In: UNOPAR ANHANGUERA. Estrutura de Dados. [Material de curso]. Unidade 4, Estruturas de Dados Avançados e Análise de Dados. [S. l.]: Anhanguera Unopar, 2025. Acesso restrito.