Tecnólogo Ciência de Dados

Guilherme Giacomini Teixeira

**LISTA ENCADEADA:**

Trabalho de Avaliação da Unidade 1 da Disciplina Estrutura de Dados

Balneário Camboriú - SC
2025

Guilherme Giacomini Teixeira

# LISTA ENCADEADA:

## Trabalho de Avaliação da Unidade 1 da Disciplina Estrutura de Dados

Trabalho de avaliação da unidade 1 da disciplina Estrutura 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

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** usando 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 atividade 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.

## 2 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 popularidade advém de sua facilidade de aprendizado 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 eficiência 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 recebe 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 elementos, a lista é impressa na tela e a função count_nodes é chamada, exibindo o número de nós presentes 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}")
```

## 3 RESULTADOS:

A atividade foi realizada com sucesso no ambiente de desenvolvimento VS Code, 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 referências é fundamental para a sua operação. A implementação do código reforçou a teoria de que, para contar os elementos, é necessário percorrer a lista sequencialmente, diferentemente de estruturas de dados que permitem acesso direto a qualquer elemento por meio de um índice.

## 4 CONCLUSÃO

A atividade prática proporcionou a oportunidade de aplicar os conceitos teóricos de listas encadeadas, aprendidos na disciplina de Estrutura de Dados. A implementação de um sistema de contagem de nós, usando a linguagem Python no ambiente 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 encadeadas e sua aplicação em problemas práticos, mostrando como o entendimento de estruturas de dados é crucial para o desenvolvimento de softwares.

.

## 5 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.

LEITE, V. M. **Listas Encadeadas**. Aula 2. In: UNOPAR ANHANGUERA. Fundamentos de Estruturas de Dados. [Material de curso]. Unidade 1, Fundamentos de Estruturas 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 Estrutura de Dados

Balneário Camboriú - SC
2025

Guilherme Giacomini Teixeira

# ÁRVORES AVL:

## Trabalho de Avaliação da Unidade 2 da Disciplina Estrutura de Dados

Trabalho de avaliação da unidade 2 da disciplina Estrutura 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

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 Pokémon utilizando uma Árvore AVL usando uma linguagem de programação.

A árvore deve armazenar informações sobre cada Pokémon, como nome e valor de força. Além disso, o sistema deve ser capaz de realizar buscas por nome, remover 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**.

## 2 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 popularidade advém de sua facilidade de aprendizado 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 pela sua eficiência 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 operação. 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)
```

## 3 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 português, 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 implementada, permitindo a visualização dos Pokémon mais fortes.

## 4 CONCLUSÃO

A atividade prática proporcionou a oportunidade de aplicar os conceitos teóricos de **Árvores AVL**, aprendidos na disciplina de **Estrutura de Dados**. A implementação de um sistema de gerenciamento de Pokémon utilizando essa estrutura de dados, 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 complexidade 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 eficientes são cruciais para o desenvolvimento de softwares robustos e de alto desempenho.

.

## 5 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.

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. *Estrutura de Dados*. [Material de curso]. Unidade 2. [S. l.]: Anhanguera Unopar, 2025. Acesso restrito.

Tecnólogo Ciência de Dados

Guilherme Giacomini Teixeira

# ALGORITMOS PARA GRAFOS EM PYTHON:

Trabalho de Avaliação da Unidade 3 da Disciplina Estrutura de Dados

Balneário Camboriú - SC
2025

Guilherme Giacomini Teixeira

# ALGORITMOS PARA GRAFOS EM PYTHON:

Trabalho de Avaliação da Unidade 3 da Disciplina Estrutura de Dados

Trabalho de avaliação da unidade 3 da disciplina Estrutura 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

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 linguagem de programação Python.

O projeto consiste em desenvolver uma aplicação para uma empresa de logística, capaz de calcular a rota mais rápida entre dois pontos de entrega, considerando 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 método para encontrar o caminho mais curto usando o algoritmo de Dijkstra.

Este relatório detalhará os passos seguidos para o desenvolvimento da aplicação, incluindo a configuração do ambiente no VS Code e a implementação do código para resolver o problema proposto.

## 2 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 programação 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 implementar um algoritmo de caminho mínimo para grafos. A sua eficiência em lidar com estruturas 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 desenvolver 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 Python:

```
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 novamente.")
        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()
```

## 3 RESULTADOS:

A atividade proposta, que envolveu a implementação de um algoritmo de caminho mínimo para grafos, permitiu a aplicação prática de conceitos teóricos de Estrutura 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, utilizando 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 algoritmo de Dijkstra foram capazes de receber um conjunto de dados estático, processar 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 aluno em aplicar os conceitos de grafos e o algoritmo de Dijkstra para resolver um problema de otimização de rotas de forma eficiente.

## 4 CONCLUSÃO

A atividade prática proporcionou a oportunidade de aplicar os conceitos teóricos de grafos e o algoritmo de Dijkstra, aprendidos na disciplina de Estrutura de Dados. 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 encontrar o caminho mais curto confirmaram a eficácia dessas estruturas de dados e algoritmos para a organização e manipulação de informações de forma eficiente.

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

## 5 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.

LEITE, V. M. **Algoritmos para grafos em Python**. Aula 4. In: UNOPAR ANHANGUERA. Estrutura de Dados. [Material de curso]. Unidade 3, Grafos e suas Operações. [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 Estrutura de Dados

Balneário Camboriú – SC
2025

Guilherme Giacomini Teixeira

# ANALISE DE DADOS ESTRUTURADOS:

## Trabalho de Avaliação da Unidade 4 da Disciplina Estrutura de Dados

Trabalho de avaliação da unidade 4 da disciplina Estrutura 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

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 análise 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 conexões.

O projeto também envolve a utilização de algoritmos de detecção de comunidades 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 solicitados e a análise dos resultados obtidos.

## 2 DESENVOLVIMENTO

A linguagem Python é amplamente utilizada em Ciência de Dados por sua sintaxe 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 funcionalidades:

Classe SocialNetwork: Gerencia o grafo da rede social, armazenando usuários e suas conexões. Possui métodos para adicionar e remover usuários, conectar e desconectar 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")
```

## 3 RESULTADOS:

Nesta atividade, o projeto foi implementado com sucesso utilizando o VSCode e a linguagem Python, seguindo a estrutura de classes proposta. A solução demonstrou 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 utilização da biblioteca NetworkX simplificou a manipulação do grafo, permitindo a aplicação 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 influentes.

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

## 4 CONCLUSÃO

A atividade prática proporcionou a oportunidade de aplicar os conceitos teóricos de análise de dados estruturados e grafos, aprendidos na disciplina de Estrutura de Dados. A implementação de um sistema de gerenciamento de rede social utilizando 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 fundamental 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 estruturas de dados avançadas, como grafos, em problemas práticos. Ele mostrou como a escolha de ferramentas e bibliotecas eficientes é crucial para o desenvolvimento de softwares robustos e para a análise de dados complexos de forma eficaz.

.

## 5 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 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.