

Tecnólogo Ciência de Dados

Guilherme Giacomini Teixeira

Árvores Avl:

Trabalho de Avaliação da Unidade 2 da Disciplina Estrutura de Dados

Balneário Camboriú - SC

2025

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

SUMÁRIO

[1 INTRODUÇÃO 3](#_Toc129809424)

[2 DESENVOLVIMENTO 4](#_Toc129809425)

[3 RESULTADOS](#_Toc129809426) 6

[4 CONCLUSÃO](#_Toc129809427) 7

[5 REFERÊNCIAS](#_Toc129809428) 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 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

# 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\_arvore.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.obter\_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.obter\_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.obter\_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.obter\_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)

# 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.

# 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.

.

# 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.