ALEXANDRE DIAMANTE

FUNDAMENTOS DE ESTRUTURAS DE DADOS
2 – ESTRUTURA DE DADOS ÁRVORES
4 - Árvores AVL

Trabalho apresentado para Atividade Prática na disciplina ESTRUTURA DE DADOS, do curso de CIÊNCIA DE DADOS, turno EAD da Instituição de Ensino Ampli Anhanguera.

Umuarama-PR

2024

```
class Node:
    def __init__(self, name, strength):
        self.name = name
        self.strength = strength
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    # Função para obter a altura do nó
    def get_height(self, node):
        if not node:
            return 0
        return node.height

    # Função para calcular o fator de balanceamento
    def get_balance(self, node):
        if not node:
            return 0
        return self.get_height(node.left) - self.get_height(node.right)

    # Rotação a direita
    def rotate_right(self, z):
        y = z.left
        T3 = y.right

        # Realiza a rotação
        y.right = z
        z.left = T3

        # Atualiza alturas
        z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))

        return y

    # Rotação a esquerda
    def rotate_left(self, z):
        y = z.right
        T2 = y.left

        # Realiza a rotação
        y.left = z
        z.right = T2

        # Atualiza alturas
        z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))

        return y
```

```
# Função para inserir um novo Pokémon na árvore AVL
def insert(self, node, name, strength):
    # Inserção normal na árvore binária de busca baseada na força
    if not node:
        return Node(name, strength)
    elif strength < node.strength:  # Note que agora comparamos pela força
        node.left = self.insert(node.left, name, strength)
    elif strength > node.strength:
        node.right = self.insert(node.right, name, strength)
    else:  # Se as forças forem iguais, usamos o nome como desempate
        if name < node.name:
            node.left = self.insert(node.left, name, strength)
        else:
            node.right = self.insert(node.right, name, strength)

    # Atualiza a altura do nó atual
    node.height = 1 + max(self.get_height(node.left), self.get_height(node.right))

    # Verifica o balanceamento do nó
    balance = self.get_balance(node)

    # Se o nó estiver desbalanceado, realiza as rotações apropriadas

    # Caso 1 - Rotação a direita
    if balance > 1 and self.get_balance(node.left) >= 0:
        return self.rotate_right(node)

    # Caso 2 - Rotação a esquerda
    if balance < -1 and self.get_balance(node.right) <= 0:
        return self.rotate_left(node)

    # Caso 3 - Rotação a esquerda-direita
    if balance > 1 and self.get_balance(node.left) < 0:
        node.left = self.rotate_left(node.left)
        return self.rotate_right(node)

    # Caso 4 - Rotação a direita-esquerda
    if balance < -1 and self.get_balance(node.right) > 0:
        node.right = self.rotate_right(node.right)
        return self.rotate_left(node)

    return node

# Função para encontrar o nó com o menor valor (utilizada na remoção)
def get_min_value_node(self, node):
    if node is None or node.left is None:
        return node
    return self.get_min_value_node(node.left)
```

```
    # Função para remover um Pokémon da árvore AVL
    def delete(self, node, name):
        # Busca o nó a ser removido baseado no nome
        if not node:
            return node
        if name < node.name:
            node.left = self.delete(node.left, name)
        elif name > node.name:
            node.right = self.delete(node.right, name)
        else:
            # Nó com apenas um filho ou nenhum filho
            if node.left is None:
                temp = node.right
                node = None
                return temp
            elif node.right is None:
                temp = node.left
                node = None
                return temp

            # Nó com dois filhos, obter o sucessor em ordem (menor na subárvore
direita)
            temp = self.get_min_value_node(node.right)

            # Copia os dados do sucessor para este nó e depois remove o sucessor
            node.name = temp.name
            node.strength = temp.strength
            node.right = self.delete(node.right, temp.name)

        if node is None:
            return node

        # Atualiza a altura do nó atual
        node.height = 1 + max(self.get_height(node.left), self.get_height(node.right))

        # Verifica o balanceamento do nó
        balance = self.get_balance(node)

        # Realiza as rotações para balancear a árvore

        # Caso 1 - Rotação a direita
        if balance > 1 and self.get_balance(node.left) >= 0:
            return self.rotate_right(node)

        # Caso 2 - Rotação a esquerda
        if balance < -1 and self.get_balance(node.right) <= 0:
            return self.rotate_left(node)

        # Caso 3 - Rotação a esquerda-direita
        if balance > 1 and self.get_balance(node.left) < 0:
```

```
            node.left = self.rotate_left(node.left)
            return self.rotate_right(node)

        # Caso 4 - Rotação a direita-esquerda
        if balance < -1 and self.get_balance(node.right) > 0:
            node.right = self.rotate_right(node.right)
            return self.rotate_left(node)

        return node

    # Função para buscar um Pokémon pelo nome
    def search(self, node, name):
        if node is None:
            return None
        if node.name == name:
            return node
        if name < node.name:
            return self.search(node.left, name)
        else:
            return self.search(node.right, name)

    # Função para listar os Pokémon em ordem decrescente de força
    def list_descending(self, node, result=None):
        if result is None:
            result = []
        if node is not None:
            self.list_descending(node.right, result)
            result.append((node.name, node.strength))
            self.list_descending(node.left, result)
        return result

# Testando a implementação
tree = AVLTree()
root = None

# Inserindo Pokémon
root = tree.insert(root, "Pikachu", 55)
root = tree.insert(root, "Charmander", 50)
root = tree.insert(root, "Bulbasaur", 45)
root = tree.insert(root, "Squirtle", 48)
root = tree.insert(root, "Jigglypuff", 20)

# Listando Pokémon em ordem decrescente de força
print("Pokémon em ordem decrescente de força:")
for pokemon in tree.list_descending(root):
    print(pokemon)

# Buscando um Pokémon
search_result = tree.search(root, "Charmander")
if search_result:
```

```
    print(f"Encontrado: {search_result.name} com força {search_result.strength}")
else:
    print("Pokémon não encontrado")

# Removendo um Pokémon
root = tree.delete(root, "Bulbasaur")

# Listando Pokémon após remoção
print("\nPokémon após remoção de Bulbasaur:")
for pokemon in tree.list_descending(root):
    print(pokemon)

# Testando a remoção de diferentes casos
# Inserindo mais Pokémon
root = tree.insert(root, "Gengar", 60)  # Maior força, será a nova raiz
root = tree.insert(root, "Eevee", 35)
root = tree.insert(root, "Snorlax", 100)  # Nova raiz após balanceamento
root = tree.insert(root, "Mewtwo", 120)

# Listando Pokémon antes das remoções
print("\nPokémon antes das remoções:")
for pokemon in tree.list_descending(root):
    print(pokemon)

# 1. Removendo uma folha (Pokémon sem filhos)
root = tree.delete(root, "Jigglypuff")  # Remoção de folha
print("\nApós remoção de Jigglypuff (uma folha):")
for pokemon in tree.list_descending(root):
    print(pokemon)

# 2. Removendo um Pokémon com um filho
root = tree.delete(root, "Eevee")  # Eevee tem apenas um filho ou nenhum
print("\nApós remoção de Eevee (um filho):")
for pokemon in tree.list_descending(root):
    print(pokemon)

# 3. Removendo um Pokémon com dois filhos
root = tree.delete(root, "Pikachu")  # Pikachu tem dois filhos
print("\nApós remoção de Pikachu (dois filhos):")
for pokemon in tree.list_descending(root):
    print(pokemon)

# Verificando se a árvore se manteve balanceada após as remoções
search_result = tree.search(root, "Snorlax")
if search_result:
    print(f"\nRaiz atual após balanceamento: {search_result.name} com força
{search_result.strength}")
else:
    print("\nErro: a árvore não está balanceada corretamente")
```

Publicado no Git:
https://github.com/alexandrediamante/cienciadedados.git