## **Relatório: Avaliação de Árvores Binárias de Procura**

### **Introdução**

Este relatório apresenta um estudo detalhado sobre a eficiência das árvores AVL em comparação com as árvores binárias de procura standards (BST). O objetivo é analisar como diferentes abordagens de implementação de árvores de busca influenciam o desempenho em operações de inserção, pesquisa e remoção. O trabalho incluiu a implementação de ambas as estruturas, a geração de conjuntos de dados para testes, a medição de tempos de execução para as operações mencionadas e a visualização dos resultados obtidos.

### **Objetivos do Estudo**

1. Estudar o efeito das árvores AVL na inserção ordenada e na pesquisa de elementos.
2. Avaliar a diferença entre árvores AVL e árvores binárias de procura standards (BST).
3. Implementar e testar a operação AVL-remove().
4. Analisar as operações de remoção de elementos em ambos os tipos de árvores.
5. Apresentar gráficos com desempenho e diferenças entre os dois casos.

### **Plano do Projeto**

O projeto foi dividido nas seguintes etapas:

1. **Implementação de Árvores BST e AVL**
2. **Geração de Dados de Teste**
3. **Análise de Desempenho**
4. **Implementação da Operação AVL-remove()**
5. **Estudo Comparativo das Operações de Remoção**
6. **Visualização dos Resultados**

### **Implementação das Árvores BST e AVL**

#### **Árvore Binária de Procura (BST)**

A árvore BST foi implementada com os métodos básicos de inserção, pesquisa e remoção:

class Node:

def \_\_init\_\_(self, key):

self.left = None

self.right = None

self.val = key

class BST:

def \_\_init\_\_(self):

self.root = None

def insert(self, key):

if self.root is None:

self.root = Node(key)

else:

self.\_insert(self.root, key)

* **Inserção:** Adicionar um novo nó na árvore mantendo a propriedade de ordem.
* **Pesquisa:** Localizar um nó específico na árvore.
* **Remoção:** Excluir um nó da árvore e ajustar a estrutura.

#### **Árvore AVL**

A árvore AVL foi implementada com os métodos de inserção, pesquisa e remoção, incluindo o balanceamento:

class AVLNode:

def \_\_init\_\_(self, key):

self.left = None

self.right = None

self.val = key

self.height = 1

class AVL:

def \_\_init\_\_(self):

self.root = None

def insert(self, key):

self.root = self.\_insert(self.root, key)

* **Inserção:** Adicionar um novo nó e balancear a árvore.
* **Pesquisa:** Localizar um nó específico mantendo o balanceamento.
* **Remoção:** Excluir um nó e rebalancear a árvore.

### 

### 

### **Geração de Dados de Teste**

Foram gerados dois tipos de conjuntos de dados para testar as árvores:

1. **Dados Ordenados:** Sequência de números em ordem crescente.
2. **Dados Aleatórios:** Sequência de números embaralhados aleatoriamente.

import random

def generate\_ordered\_data(size):

return list(range(size))

def generate\_random\_data(size):

data = list(range(size))

random.shuffle(data)

return data

### **Avaliação de Desempenho**

As operações de inserção, pesquisa e remoção foram medidas para ambas as árvores utilizando dados aleatórios. A medição de tempo foi realizada utilizando a biblioteca time:

import time

def measure\_insertion(tree, data):

start\_time = time.time()

for key in data:

tree.insert(key)

end\_time = time.time()

return end\_time - start\_time

def measure\_search(tree, data):

start\_time = time.time()

for key in data:

tree.search(key)

end\_time = time.time()

return end\_time - start\_time

def measure\_deletion(tree, data):

start\_time = time.time()

for key in data:

tree.delete(key)

end\_time = time.time()

return end\_time - start\_time

### **Comparação de Desempenho**

Foram utilizados três tamanhos de conjuntos de dados: 100, 1.000 e 10.000 elementos. Os tempos de execução foram medidos para cada operação e cada tipo de árvore:

sizes = [100, 1000, 10000]

results = {

"BST": {"insert": [], "search": [], "delete": []},

"AVL": {"insert": [], "search": [], "delete": []}

}

for size in sizes:

ordered\_data = generate\_ordered\_data(size)

random\_data = generate\_random\_data(size)

bst = BST()

avl = AVL()

results["BST"]["insert"].append(measure\_insertion(bst, random\_data))

results["BST"]["search"].append(measure\_search(bst, random\_data))

results["BST"]["delete"].append(measure\_deletion(bst, random\_data))

results["AVL"]["insert"].append(measure\_insertion(avl, random\_data))

results["AVL"]["search"].append(measure\_search(avl, random\_data))

results["AVL"]["delete"].append(measure\_deletion(avl, random\_data))

### **Visualização dos Resultados**

Os resultados das medições foram visualizados utilizando gráficos, que mostram o tempo de execução das operações em função do número de elementos:

import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))

plt.subplot(1, 3, 1)

plt.plot(sizes, results["BST"]["insert"], label='BST Insert')

plt.plot(sizes, results["AVL"]["insert"], label='AVL Insert')

plt.xlabel('Número de Elementos')

plt.ylabel('Tempo (segundos)')

plt.title('Tempo de Inserção')

plt.legend()

plt.subplot(1, 3, 2)

plt.plot(sizes, results["BST"]["search"], label='BST Search')

plt.plot(sizes, results["AVL"]["search"], label='AVL Search')

plt.xlabel('Número de Elementos')

plt.ylabel('Tempo (segundos)')

plt.title('Tempo de Pesquisa')

plt.legend()

plt.subplot(1, 3, 3)

plt.plot(sizes, results["BST"]["delete"], label='BST Delete')

plt.plot(sizes, results["AVL"]["delete"], label='AVL Delete')

plt.xlabel('Número de Elementos')

plt.ylabel('Tempo (segundos)')

plt.title('Tempo de Remoção')

plt.legend()

plt.tight\_layout()

plt.show()

### **Análise dos Gráficos**

#### **Inserção**

A inserção em árvores AVL é mais lenta devido ao overhead do rebalanceamento, mas a estrutura balanceada garante um desempenho consistente nas operações subsequentes.

#### **Pesquisa**

As árvores AVL mantêm um tempo de pesquisa baixo e consistente, enquanto as BSTs podem ter tempos de pesquisa mais elevados se estiverem desbalanceadas.

#### 

#### 

#### **Remoção**

As árvores AVL são eficientes na remoção devido ao rebalanceamento automático, enquanto as BSTs apresentam maior variabilidade no tempo de remoção, especialmente quando desbalanceadas.

### **Conclusões Finais**

* **AVL:** As árvores AVL são mais eficientes em operações de pesquisa devido ao balanceamento mantido após cada inserção e remoção. No entanto, apresentam um overhead maior na inserção e remoção devido às operações de rebalanceamento necessárias para manter a árvore balanceada.
* **BST:** As árvores BST têm um desempenho aceitável em dados aleatórios, com menor overhead durante a inserção e remoção, mas sofrem de desbalanceamento, o que resulta em tempos de pesquisa menos consistentes e potencialmente mais altos.
* **Recomendação:** Usar árvores AVL para aplicações que requerem alta performance e consistência nas operações de pesquisa, mesmo que isso implique um overhead maior em inserções e remoções. Árvores BST podem ser adequadas para conjuntos de dados menores ou quando a ordem de inserção é suficientemente aleatória para evitar desbalanceamento significativo.