# Relatório Técnico: Comparação de Árvores de Busca (BST, AVL, Rubro-Negra)

**Disciplina:** Estrutura de Dados

**Integrantes:**

- Fellipe Melhoranca B. Tomasella
- Inglid Pablina de A. Sandeski
- João Vitor de S. Costa
- Vitor Linsbinski de Oliveira

---

## 1. Introdução Teórica

As árvores de busca são estruturas de dados fundamentais para o armazenamento e recuperação eficiente de informações. Este trabalho analisa três variações:

### 1.1. Árvore Binária de Busca (BST)

A estrutura base onde cada nó possui até dois filhos. A propriedade fundamental é que todos os nós à esquerda possuem valores menores que a raiz, e todos à direita, valores maiores. Embora simples, não possui mecanismos de balanceamento, podendo degenerar para uma lista encadeada (O(n)) no pior caso.

### 1.2. Árvore AVL (Adelson-Velsky e Landis)

Uma árvore binária de busca **auto-balanceável**. Ela mantém a altura controlada através do **Fator de Balanceamento** (diferença de altura entre subárvores esquerda e direita), que deve ser sempre -1, 0 ou 1. Se esse fator for violado após uma inserção ou remoção, a árvore aplica **rotações** (simples ou duplas) para se reequilibrar, garantindo complexidade O(log n).

### 1.3. Árvore Rubro-Negra (Red-Black)

Outra árvore auto-balanceável, mas com critérios menos rígidos que a AVL. Ela utiliza **cores (Vermelho/Preto)** nos nós e um conjunto de 5 propriedades para garantir que o caminho mais longo da raiz a uma folha não seja mais que o dobro do caminho mais curto. Isso resulta em

menos rotações durante inserções/remoções, tornando-a ideal para aplicações com muitas escritas.

---

## 2. Implementação e Funções Principais

O projeto foi desenvolvido em **Python**, utilizando Programação Orientada a Objetos.

### 2.1. Estrutura dos Nós

Todos os nós armazenam:

- `id` (Chave inteira para ordenação)
- `valor` (Dado associado)
- `dado_extra` (Categoria opcional)
- Ponteiros `esquerda` e `direita`

### 2.2. Funções Implementadas

Para cada árvore, foram implementadas as operações:

- **Inserir(id, valor):** Adiciona um novo nó mantendo as propriedades da árvore.
- **Buscar(id):** Retorna o nó e o número de comparações realizadas.
- **Remover(id):** Remove o nó e reorganiza a árvore (lidando com sucessores e rebalanceamento).
- **Travessias:** Em-ordem (In-order), Pré-ordem e Pós-ordem.
- **Métricas:** Cálculo de altura, contagem de nós e rotações.

---

## 3. Análise de Complexidade

A tabela abaixo resume a complexidade de tempo (Big O) para as operações:

| Estrutura | Melhor Caso | Caso Médio | Pior Caso | Observação |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| **BST** | O(log n) | O(log n) | **O(n)** | Degenera com dados ordenados. |
| **AVL** | O(log n) | O(log n) | **O(log n)** | Estritamente balanceada. |
| **Rubro-Negra** | O(log n) | O(log n) | **O(log n)** | Balanceamento pragmático. |

---

## 4. Resultados Experimentais

Os testes foram realizados com conjuntos de dados de **100, 1.000 e 10.000** elementos, gerados aleatoriamente.

*(Nota: Insira aqui os gráficos gerados pelo Benchmark ou a tabela do CSV)*

**Observações dos Testes:**

1. **Inserção:** A BST é a mais rápida com dados aleatórios (menos overhead), mas a Rubro-Negra supera a AVL por fazer menos rotações.

2. **Busca:** A AVL tende a ser levemente mais rápida que a Rubro-Negra em grandes volumes, pois sua altura é estritamente menor.

3. **Pior Caso:** Ao inserir dados já ordenados (1, 2, 3...), a BST teve desempenho catastrófico (O(n)), enquanto AVL e Rubro-Negra mantiveram O(log n).

---

## 5. Conclusão

A análise comparativa permite concluir que:

- **BST** é adequada apenas para dados puramente aleatórios e onde a simplicidade de implementação é prioritária. Seu risco de degeneração a torna inviável para sistemas críticos.

- **AVL** é a melhor escolha para cenários com **muitas buscas e poucas alterações** (ex: dicionários, índices estáticos), devido ao seu balanceamento rígido que otimiza a leitura.

- **Rubro-Negra** é a melhor opção para **uso geral** (ex: `TreeMap` do Java, `std::map` do C++), pois oferece um excelente compromisso entre velocidade de busca e custo de atualização (rotações).

---

# 6. Anexos: Trechos de Código

## 6.1. Rotação na AVL

```
def rotacionar_direita(self, y):
    x = y.esquerda
    T2 = x.direita
    x.direita = y
    y.esquerda = T2
    y.altura = 1 + max(self.obter_altura_no(y.esquerda), self.obter_altura_no(y.direita))
    x.altura = 1 + max(self.obter_altura_no(x.esquerda), self.obter_altura_no(x.direita))
    return x
```

## 6.2. Propriedades da Rubro-Negra (Inserção)

```
def _consertar_insercao(self, k):
    while k.pai.cor == "VERMELHO":
        if k.pai == k.pai.pai.direita:
            u = k.pai.pai.esquerda # Tio
            if u.cor == "VERMELHO":
                u.cor = "PRETO"
                k.pai.cor = "PRETO"
                k.pai.pai.cor = "VERMELHO"
                k = k.pai.pai
            else:
                # ... Rotações ...
```

*(Nota: Adicione aqui os prints da tela do programa em execução)*