Implementação e Análise de Árvores Balanceadas: AVL e Rubro-Negra

Jerônimo Rafael Bezerra Filho
Matrícula: 20240039188
Estruturas de Dados Básicas II

Universidade Federal do Rio Grande do Norte
Link para o Repositório no GitHub

24 de outubro de 2025

Resumo

Este relatório detalha o projeto desenvolvido para a disciplina de Estruturas de Dados Básicas II, focado na implementação e análise de duas estruturas de dados avançadas: a Árvore AVL e a Árvore Rubro-Negra. O objetivo do trabalho foi implementar as operações fundamentais (inserção, remoção e busca) para ambas as árvores, garantindo que as propriedades de balanceamento de cada uma fossem mantidas corretamente após as modificações. O sistema permite ao usuário escolher qual estrutura de árvore deseja utilizar no momento da execução e testar suas funcionalidades.

Sumário

1 Introdução

Árvores Binárias de Busca (BST) são estruturas de dados fundamentais que permitem operações eficientes de busca, inserção e remoção. No entanto, seu desempenho, no pior caso, pode degenerar para o de uma lista ligada ($O(n)$) se os dados forem inseridos de forma ordenada. Para resolver esse problema, foram desenvolvidas as Árvores de Busca Balanceadas.

Este trabalho explora duas das mais importantes implementações dessas árvores: a Árvore AVL e a Árvore Rubro-Negra. O objetivo é implementar ambas as estruturas em C++, permitindo ao usuário interagir com elas e analisar empiricamente suas estratégias de balanceamento e complexidade de implementação.

2 Estruturas de Dados e Estratégias de Balanceamento

2.1 Árvore AVL (Adelson-Velsky e Landis)

A Árvore AVL foi a primeira estrutura de árvore de busca auto-balanceada inventada. Sua principal característica é a manutenção de uma propriedade de altura rigorosa.

2.1.1 Estrutura de Dados

O nó de uma Árvore AVL (`NoAVL`) armazena, além do dado e dos ponteiros para os filhos (`esquerda`, `direita`), um campo adicional: `altura`. A altura de um nó é definida como a maior distância (em número de arestas) entre ele e uma folha em sua subárvore.

2.1.2 Estratégia de Balanceamento

A AVL garante que, para qualquer nó na árvore, as alturas de suas duas subárvores (esquerda e direita) difiram em no máximo 1. Essa propriedade é mantida através do **Fator de Balanceamento (FB)**, calculado como:

$FB = altura(filho\_esquerda) - altura(filho\_direita)$

Após cada inserção ou remoção, o FB de cada nó ancestral é verificado. Se o FB de um nó se tornar -2 ou +2, a árvore está desbalanceada e rotações são aplicadas para corrigi-la. Existem quatro casos de rotação:

- **Rotação Simples à Direita (Caso LL):** Quando o desbalanceamento é +2 e o filho da esquerda tem FB +1.
- **Rotação Simples à Esquerda (Caso RR):** Quando o desbalanceamento é -2 e o filho da direita tem FB -1.
- **Rotação Dupla Esquerda-Direita (Caso LR):** Quando o desbalanceamento é +2 e o filho da esquerda tem FB -1. Uma rotação à esquerda é feita no filho da esquerda, seguida de uma rotação à direita no nó original.

- **Rotação Dupla Direita-Esquerda (Caso RL):** Quando o desbalanceamento é -2 e o filho da direita tem FB +1. Uma rotação à direita é feita no filho da direita, seguida de uma rotação à esquerda no nó original.

2.2 Árvore Rubro-Negra

A Árvore Rubro-Negra é outra árvore de busca auto-balanceada que, embora menos rígida que a AVL, garante o balanceamento através de um conjunto de propriedades baseadas em cores.

2.2.1 Estrutura de Dados

O nó (`NoRN`) armazena, além do dado e dos ponteiros (`esquerda`, `direita`), um ponteiro para o `pai` e um atributo de `cor` (VERMELHO ou PRETO). A implementação utiliza um nó sentinela (`T_NIL`) para representar todas as folhas nulas, simplificando a implementação das operações.

2.2.2 Estratégia de Balanceamento

O balanceamento é garantido pela manutenção de 5 propriedades fundamentais:

1. Todo nó é VERMELHO ou PRETO.
2. A raiz é sempre PRETA.
3. Toda folha (`T_NIL`) é PRETA.
4. Se um nó é VERMELHO, então ambos os seus filhos são PRETOS (um nó VERMELHO não pode ter um filho VERMELHO).
5. Para cada nó, todos os caminhos simples daquele nó até as folhas descendentes contêm o mesmo número de nós PRETOS (a "altura preta").

Após inserções ou remoções, se alguma dessas propriedades for violada (mais comumente a 4 ou a 2), a árvore é corrigida através de **recolorações** e **rotações** (funções `fixInserir` e `fixRemover`).

3 Descrição das Funções Implementadas

O sistema foi modularizado em classes (`ArvoreAVL` e `ArvoreRubroNegra`) e um arquivo `main.cpp` contendo os menus de interação.

- **Funções Públicas (Comuns):**
  - `inserir(int dado)`: Adiciona um novo nó à árvore e aciona as funções de balanceamento.
  - `remover(int dado)`: Remove um nó da árvore (usando o sucessor em-ordem para o caso de dois filhos) e aciona as funções de balanceamento.

  - `buscar(int dado)`: Procura um valor na árvore e informa ao usuário se foi encontrado.
  - `imprimir()`: Exibe a estrutura da árvore no console de forma visual.

- **Funções Auxiliares (AVL):**
  - `getAltura()`, `getFatorBalanceamento()`, `atualizarAltura()`: Funções de gerenciamento da altura.
  - `rotacaoDireita()`, `rotacaoEsquerda()`: Implementam as rotações.

- **Funções Auxiliares (Rubro-Negra):**
  - `rotacaoDireita()`, `rotacaoEsquerda()`: Implementam as rotações (com atualização de ponteiros de pai).
  - `fixInserir()`: Corrige a árvore após a inserção (casos do "tio").
  - `fixRemover()`: Corrige a árvore após a remoção (casos do "irmão").
  - `transplantar()`: Função auxiliar para a remoção, que substitui uma subárvore por outra.

4 Principais Desafios Encontrados

A implementação das duas árvores apresentou desafios distintos. Na **Árvore AVL**, o desafio foi garantir que a altura dos nós fosse recalculada corretamente na "volta"da recursão, para que o Fator de Balanceamento fosse detectado no nó ancestral correto, e aplicar a rotação dupla (LR/RL) na ordem correta.

Na **Árvore Rubro-Negra**, o principal desafio foi, sem dúvida, a implementação da função `fixRemover`. Enquanto a inserção (`fixInserir`) possui 3 casos principais (relativamente diretos), a remoção de um nó PRETO exige a análise de 4 casos principais, cada um com sub-casos e casos simétricos, para restaurar a "altura preta"da árvore. A lógica de `transplantar` e o gerenciamento do nó sentinela `T_NIL` também exigiram atenção especial.

5 Testes Realizados e Resultados

Para validar a corretude das implementações, foram executados testes específicos para forçar todos os casos de balanceamento e rebalanceamento, tanto na inserção quanto na remoção.

5.1 Teste A: AVL - Rotação Simples (RR)

Objetivo: Forçar uma rotação simples à esquerda (RR). **Passos:** Inserir 10, 20, 30. **Resultado:**

```
Inserindo 10...
--------------------
Inserindo 20...
```

```
--------------------
Inserindo 30...
Desbalanceamento: Direita-Direita (RR) em 10. -> (Rotação Esquerda
   em 10)
--------------------
Estrutura da Arvore AVL:
    |---[30] (h=1, fb=0)
|---[20] (h=2, fb=0)
|   |---[10] (h=1, fb=0)
--------------------
```

Análise: O programa detectou corretamente o desbalanceamento (RR) no nó 10 e aplicou a rotação à esquerda, promovendo o 20 a nova raiz.

5.2 Teste B: AVL - Rotação Dupla (LR)

Objetivo: Forçar uma rotação dupla esquerda-direita (LR). **Passos:** Inserir 30, 10, 20. **Resultado:**

```
Inserindo 30...
--------------------
Inserindo 10...
--------------------
Inserindo 20...
Desbalanceamento: Esquerda-Direita (LR) em 30. -> (Rotação Esquerda
    em 10)
 -> (Rotação Direita em 30)
--------------------
Estrutura da Arvore AVL:
    |---[30] (h=1, fb=0)
|---[20] (h=2, fb=0)
|   |---[10] (h=1, fb=0)
--------------------
```

Análise: O programa detectou o caso LR, aplicou a rotação à esquerda no 10, e depois a rotação à direita no 30, resultando na árvore balanceada.

5.3 Teste C: AVL - Remoção com Verificação de FB

Objetivo: Demonstrar que o FB é verificado e recalculado após a remoção. **Passos:** Inserir 10, 5, 20, 3, 7, 30. Em seguida, remover 30. **Resultado (Pós-Remoção):**

```
(Árvore antes de remover 30)
Estrutura da Arvore AVL:
        |---[30] (h=1, fb=0)
    |---[20] (h=2, fb=-1)
|---[10] (h=3, fb=0)
|   |   |---[7] (h=1, fb=0)
|   |---[5] (h=2, fb=0)
|       |---[3] (h=1, fb=0)
--------------------
Tentando remover 30...
--------------------
(Árvore após remover 30)
```

```
Estrutura da Arvore AVL:
    |---[20] (h=1, fb=0)
|---[10] (h=3, fb=1)
|   |   |---[7] (h=1, fb=0)
|   |---[5] (h=2, fb=0)
|       |---[3] (h=1, fb=0)
--------------------
```

Análise: Após a remoção do 30, o FB do nó 10 foi recalculado de 0 para 1. Como 1 está no intervalo [-1, 1], nenhuma rotação foi necessária.

5.4 Teste D: Rubro-Negra - Inserção (Tio Vermelho)

Objetivo: Forçar o caso de inserção com "tio"VERMELHO (Caso 1: Recoloração). **Passos:** Inserir 10, 5, 15. Depois, inserir 3. **Resultado:**

```
(Árvore antes de inserir 3)
|   |---[15,V]
|---[10,P]
    |---[5,V]

(Árvore após inserir 3)
|   |---[15,P]
|---[10,P]
    |---[5,P]
        |---[3,V]
```

Análise: Ao inserir 3, uma violação Vermelho-Vermelho ocorreu (3 e 5). O tio (15) era VERMELHO. O `fixInserir` aplicou a recoloração: 5 e 15 tornaram-se PRETOS, e 10 tornou-se a raiz PRETA.

5.5 Teste E: Rubro-Negra - Inserção (Tio Preto)

Objetivo: Forçar o caso de inserção com "tio"PRETO (Caso 2/3: Rotação). **Passos:** Inserir 10, 5, 3. **Resultado:**

```
(Árvore após inserir 3)
|   |---[10,V]
|---[5,P]
    |---[3,V]
```

Análise: Ao inserir 3, uma violação Vermelho-Vermelho ocorreu (3 e 5). O tio (`T_NIL`) era PRETO. O `fixInserir` aplicou uma rotação à direita no avô (10) e recoloriu os nós, promovendo 5 à raiz.

5.6 Teste F: Rubro-Negra - Remoção (Nó Preto)

Objetivo: Forçar o `fixRemover` ao remover um nó PRETO. **Passos:** Inserir 10, 5, 15. Remover 5 (vermelho). Remover 10 (preto). **Resultado:**

```
(Árvore após inserir 10, 5, 15)
|   |---[15,V]
|---[10,P]
```

```
    |---[5,V]

(Após remover 5 - nó Vermelho, caso simples)
|   |---[15,V]
|---[10,P]

(Após remover 10 - nó Preto, aciona fixRemover)
|---[15,P]
```

Análise: A remoção do nó VERMELHO 5 foi trivial. A remoção do nó PRETO 10 acionou o `fixRemover` no seu sucessor (15), que corrigiu as propriedades da árvore, deixando 15 como a nova raiz PRETA.

6 Conclusão

Este projeto permitiu uma compreensão prática e aprofundada das Árvores AVL e Rubro-Negra. A implementação revelou as diferenças fundamentais em suas estratégias de balanceamento: a AVL, com sua regra de altura rígida, e a Rubro-Negra, com suas propriedades de cor mais flexíveis, mas complexas de manter.

Ambas as implementações foram validadas através de testes que cobriram os principais casos de inserção e remoção. O desenvolvimento da lógica de rebalanceamento, especialmente a função de remoção da Árvore Rubro-Negra, provou ser o maior desafio, mas foi essencial para garantir a corretude e eficiência da estrutura de dados.