AED2 - Algoritmos e Estr. de Dados II

Aula 8: Árvores Rubro-Negras

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

1. O que é uma Árvore Rubro-Negra?

Uma **Árvore Rubro-Negra** (Red-Black Tree) é uma **árvore binária de busca auto-balanceada** onde cada nó armazena um bit extra: sua **cor** (Vermelho ou Preto).

- **Histórico:** Inventada por Leonidas Guibas e Robert Sedgewick em 1978, inspirada nas árvores 2-3-4 de Bayer (1972).
- **Por que essas cores?** Escolhidas por serem distinguíveis em impressoras preto-e-branco (caneta vermelha e preta).
- **Relação com B-Trees:** Uma RB-Tree é equivalente a uma árvore 2-3-4. Cada nó vermelho representa a metade inferior de um nó 3 ou 4 da B-Tree.
- **Overhead:** Apenas **1 bit** por nó – praticamente zero custo adicional de memória.

Ideia Central

As regras de cor garantem que **nenhum caminho da raiz até uma folha seja mais que 2× maior** que qualquer outro, mantendo a árvore **aproximadamente balanceada**.

2. Motivação – O Problema da BST Degenerada

Problema da BST degenerada: Inserir 1, 2, 3, 4, 5 em ordem gera uma **linha reta**. A busca degrada de $O(\log N)$ para $O(N)$.

Figure: BST degenerada = lista encadeada.

Balanceamento Automático

- **AVL (1962):** Balanceamento **rígido** ($|FB| \leq 1$). Busca mais rápida, mas **mais rotações** em inserção/remoção.
- **Rubro-Negra (1978):** Balanceamento **relaxado**. **Menos rotações**, ideal para cenários com **muitas escritas**.

Trade-off: RB é levemente mais alta que AVL (busca um pouco mais lenta), mas compensa com inserção/remoção mais eficientes.

4. Desempenho e Complexidade

Operação	BST	AVL	Rubro-Negra
Busca (melhor)	$O(\log N)$	$O(\log N)$	$O(\log N)$
Busca (pior)	$O(N)$	$O(\log N)$	$O(\log N)$
Inserção (pior)	$O(N)$	$O(\log N)$	$O(\log N)$
Remoção (pior)	$O(N)$	$O(\log N)$	$O(\log N)$
Altura máxima	N	$\approx 1{,}44 \log N$	$\leq 2 \log N$
Rotações (inserção)	0	Até $O(\log N)$	≤ 2

Comparação Numérica (para $N = 1.000.000$)

- **AVL:** altura ≈ 29
- **Rubro-Negra:** altura ≤ 40
- **BST degenerada:** altura $= 1.000.000$ **(inviável!)**

A RB-Tree garante **performance consistente** em todos os cenários, sem os picos de custo da AVL.

5. O Padrão da Indústria

Enquanto a AVL é matematicamente elegante, a **Árvore Rubro-Negra** é a escolha pragmática da indústria. Ela é a estrutura de dados por trás de:

- `java.util.TreeMap` e `TreeSet`.
- `std::map`, `set` e `std::multiset` do C++.
- O agendador de processos **Completely Fair Scheduler** do kernel Linux.
- `ConcurrentHashMap` do Java 8+ (buckets com muitos elementos viram RB-Trees).

Sua vantagem é realizar **menos rotações** que a AVL, trocando ajustes estruturais caros por simples mudanças de cor (**recoloração**).

6. As 5 Regras de Ouro

1. **Cor:** Todo nó é Vermelho ou Preto.
2. **Raiz:** A raiz **deve** ser Preta.
3. **Folhas:** Todas as folhas NIL são Pretas.
4. **Vermelho** $\Rightarrow$ **filhos Pretos:** Se um nó é Vermelho, **ambos** os filhos devem ser Pretos.
5. **Altura Negra igual:** Todo caminho até folha deve ter o **mesmo número de nós Pretos**.

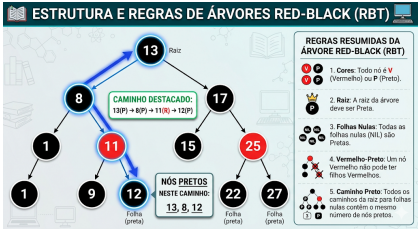


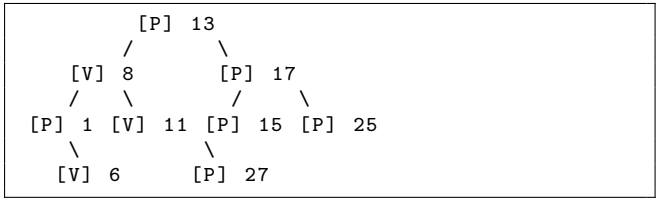
Figure: Regras rubro-negra.

Propriedade de Altura

O caminho mais longo (intercalando Vermelho-Preto) não é mais que $\mathbf{2}\times$ maior que o caminho mais curto (apenas Pretos).

- Isso garante que a árvore nunca fique "despencada" de um lado só.
- A altura h de uma árvore com n nós internos é sempre $h \leq 2\log(n+1)$.
- Busca, Inserção e Remoção permanecem $O(\log N)$.

8. Exemplo Visual – Árvore Rubro-Negra Válida



Verificação das propriedades:

- Raiz (13) é Preta ✓
- Folhas NIL são Pretas ✓
- Nenhum Vermelho tem filho Vermelho ✓
- Altura negra de todos os caminhos $= 3$ ✓

Caminhos e altura negra:

- 13→8→1→6→NIL: 3 pretos
- 13→8→11→NIL: 3 pretos
- 13→17→15→NIL: 3 pretos
- 13→17→25→27→NIL: 3 pretos

9. Lógica de Inserção – Visão Geral

Todo novo nó inserido começa como **Vermelho**. Por quê?

- Se inseríssemos **Preto**: violaríamos a Propriedade 5 (altura negra) em **todos** os caminhos passando por aquele nó – difícil de corrigir.
- Inserindo **Vermelho**: só violamos a Propriedade 4 se o **pai também for Vermelho** (dupla-vermelho). Isso é **local** e mais fácil de corrigir.

Algoritmo de Correção (Fix-Up)

Após inserir Z (Vermelho) com pai P (Vermelho), olhamos para o **Tio** U (irmão do pai):

- **Caso 1 – Tio Vermelho:** Recoloração (Pai e Tio $\rightarrow$ Preto, Avô $\rightarrow$ Vermelho). O problema **sobe** para o Avô.
- **Caso 2 – Tio Preto, configuração em linha (zig-zig):** Rotação simples + troca de cores.
- **Caso 3 – Tio Preto, configuração em joelho (zig-zag):** Rotação dupla.

10. Inserção – Caso 1: Tio Vermelho (Recoloração)

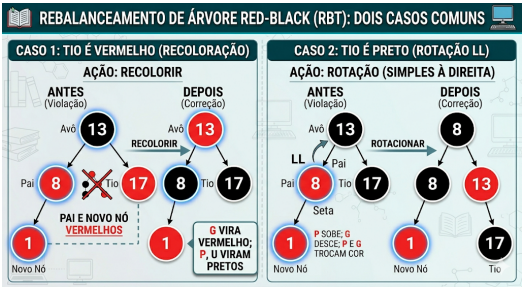


Figure: Caso do Tio Vermelho.

Passos da Recoloração:

1. Pinte Pai (P) de **Preto**.
2. Pinte Tio (U) de **Preto**.
3. Pinte Avô (G) de **Vermelho**.
4. Suba recursivamente para verificar novas violações em G.

11. Por que a Recoloração funciona?

Preservação da Altura Negra

A recoloração transforma 1 preto (Avô) e 2 vermelhos (Pai/Tio) em 1 vermelho (Avô) e 2 pretos (Pai/Tio).

- O número de nós pretos em qualquer caminho que passe pelo avô **não muda**.
- A violação de "duplo-vermelho" apenas **sobe na árvore**.
- Se chegar na raiz, pintamos a raiz de preto (Regra 2) e o balanceamento termina.

12. Inserção – Casos 2 e 3: Tio Preto (Rotação)

Quando o **Tio** é Preto (ou NIL), a recoloração não basta – precisamos de **rotações**.

Caso 2: Linha (zig-zig)

- Z e P estão no **mesmo lado** de G (ambos à esquerda ou ambos à direita).
- **Solução:** Rotação **simples** em G + trocar cores de P e G.
- **Resultado:** Árvore corrigida em 1 rotação.

Caso 3: Joelho (zig-zag)

- Z e P estão em **lados opostos** de G (ex: P é filho esquerdo, Z é filho direito).
- **Solução:** Rotação **dupla** – primeiro em P, depois em G.
- **Resultado:** Transforma em Caso 2, depois resolve.

4 Subcasos

Cada caso tem 2 variantes (esquerda/direita), totalizando 4 subcasos simétricos: EE, ED, DE, DD. O código trata os dois lados explicitamente.

13. Implementação Java – Estrutura do Nó

```java
class Node {
    int key;
    boolean isRed;     // true = Vermelho, false = Preto
    Node left, right, parent;

    Node(int key) {
        this.key = key;
        this.isRed = true;  // Novo no sempre começa Vermelho
    }
}
```

Comparação de Cores (OpenJDK style)

```java
boolean isRed(Node x) { return (x != null) && x.isRed; }
Node parentOf(Node x) { return (x == null) ? null : x.parent; }
```

14. Implementação Java – Nó Sentinela NIL

O que é o Sentinela?

Em vez de usar `null` para representar o fim dos caminhos, usamos um nó especial chamado **NIL**:

- O NIL é sempre de cor **Preta**.
- Todas as referências vazias apontam para este único objeto.

Vantagem: Simplifica enormemente o código dos algoritmos (Rotação e Fix-Up). Não precisamos checar se ‘pai.esq == null‘ o tempo todo, apenas olhamos a cor do nó retornado.

15. Implementação Java – Rotações

Rotação à Esquerda:

```
void rotateLeft(Node x) {
    Node y = x.right;
    x.right = y.left;
    if (y.left != null) y.left.parent = x;
    y.parent = x.parent;
    if (x.parent == null) root = y;
    else if (x == x.parent.left) x.parent.left = y;
    else x.parent.right = y;
    y.left = x; x.parent = y;
}
```

Rotação à Direita:

```
void rotateRight(Node x) {
    Node y = x.left;
    x.left = y.right;
    if (y.right != null) y.right.parent = x;
    y.parent = x.parent;
    if (x.parent == null) root = y;
    else if (x == x.parent.right) x.parent.right = y;
    else x.parent.left = y;
    y.right = x; x.parent = y;
}
```

16. Implementação Java – Método Insert

```
void insert(int key) {
    Node z = new Node(key);
    Node y = null, x = root;
    while (x != null) {
        y = x;
        x = (z.key < x.key) ? x.left : x.right;
    }
    z.parent = y;
    if (y == null) root = z;
    else if (z.key < y.key) y.left = z;
    else y.right = z;

    fixInsertion(z); // Chama o balanceamento
}
```

17. Implementação Java – FixInsertion (Resumido)

```java
void fixInsertion(Node z) {
    while (isRed(parentOf(z))) {
        if (parentOf(z) == leftOf(parentOf(parentOf(z)))) {
            Node y = rightOf(parentOf(parentOf(z))); // Tio
            if (isRed(y)) {                          // Caso 1
                parentOf(z).isRed = false;
                y.isRed = false;
                parentOf(parentOf(z)).isRed = true;
                z = parentOf(parentOf(z));
            } else {
                if (z == rightOf(parentOf(z))) {     // Caso 3
                    z = parentOf(z); rotateLeft(z);
                }
                parentOf(z).isRed = false;           // Caso 2
                parentOf(parentOf(z)).isRed = true;
                rotateRight(parentOf(parentOf(z)));
            }
        } else { /* Simetrico para o lado direito */ }
    }
    root.isRed = false;
}
```

18. Implementação Java – Remoção (Conceito)

A remoção é a operação mais complexa (8 casos).

Conceito de "Duplo Preto" (Double Black)

Ao remover um nó **Preto**, o caminho que passava por ele perde 1 preto, violando a Propriedade 5.

- O nó substituto (ou NIL) recebe uma carga extra: **"duplo preto"**.
- O balanceamento tenta "empurrar" esse preto para cima até encontrar um nó vermelho ou chegar na raiz.

Nota: Em ambientes reais, prefira usar implementações nativas como `java.util.TreeMap`.

19. Implementação Java – Método Remove

```
void remove(int key) {
    Node z = findNode(key);
    if (z == null) return;
    Node y = z, x;
    boolean yOriginalRed = y.isRed;
    if (z.left == null) {
        x = z.right; transplant(z, z.right);
    } else if (z.right == null) {
        x = z.left; transplant(z, z.left);
    } else {
        y = minimum(z.right);
        yOriginalRed = y.isRed;
        x = y.right;
        if (y.parent == z) x.parent = y;
        else {
            transplant(y, y.right); y.right = z.right;
            y.right.parent = y;
        }
        transplant(z, y); y.left = z.left; y.left.parent = y;
        y.isRed = z.isRed;
    }
    if (!yOriginalRed) fixDeletion(x);
}
```

20. Exemplo Passo a Passo – Inserção (1/2)

Inserindo: **10, 20, 30, 15, 25, 5, 1**

1. **Inserir 10 (V):** Raiz → repaint Preto. Árvore: [P]10
2. **Inserir 20 (V):** Filho direito de 10. OK.
3. **Inserir 30 (V):** Pai 20 é Vermelho, Tio NIL (Preto). Caso zig-zig → **rotação esquerda em 10**.

```
1          [P] 20
2         /      \
3    [V] 10      [V] 30
```

4. **Inserir 15 (V):** Pai 10 é Vermelho, Tio 30 é Vermelho → **recoloração**: 10→P, 30→P, 20→V. Raiz 20 repaint Preto.

21. Exemplo Passo a Passo – Inserção (2/2)

Continuação:

5. **Inserir 25 (V):** Pai 30 é Preto. OK, sem violação.
6. **Inserir 5 (V):** Pai 10 é Preto. OK.
7. **Inserir 1 (V):** Pai 5 é Vermelho, Tio 15 é Preto. Caso zig-zig → **rotação direita em 10** + troca de cores.

Resultado Final

```
          [P] 20
         /      \
     [P] 10      [P] 30
     /    \          \
  [P] 5   [P] 15    [V] 25
   /
[V] 1
```

TreeMap (Mapa Ordenado)

```
TreeMap<Integer, String> agenda = new TreeMap<>();
agenda.put(10, "Reuniao"); agenda.put(5, "Academia");
agenda.put(20, "Jantar");  agenda.put(15, "Estudar");

// Iteracao em ordem: 5, 10, 15, 20
for (var e : agenda.entrySet())
    System.out.println(e.getKey() + ": " + e.getValue());

agenda.ceilingKey(7);  // 10 (menor chave >= 7)
agenda.floorKey(7);    // 5  (maior chave <= 7)

// Ordem reversa
agenda.descendingMap(); // {20, 15, 10, 5}
```

TreeSet (Conjunto Ordenado)

```
TreeSet<Integer> set = new TreeSet<>();
set.add(10); set.add(5); set.add(20);

set.higher(10);  // 20 (proximo estrito > 10)
set.lower(10);   // 5  (anterior estrito < 10)
set.headMap(15); // [5, 10] (elementos < 15)
set.tailMap(10); // [10, 20] (elementos >= 10)
```

24. Aplicações Práticas – Linux e Java

- **Linux CFS Scheduler:** O kernel usa RB-Tree para ordenar processos pelo **vruntime**. O processo com menor vruntime (mais à esquerda) é o próximo.
- **Java Collections:** `TreeMap` e `TreeSet` são implementações diretas. No `ConcurrentHashMap`, buckets com ≥ 8 elementos viram RB-Trees.

- **C++ STL:** `std::map` e `std::set` garantem complexidade logarítmica com RB-Trees.
- **Nginx:** Gerenciamento de **timers** e conexões ativas.
- **Sistemas de arquivos:** ext3/4 do Linux usam para gerenciar blocos de memória.

26. Comparativo Final – Quando Usar Cada Uma?

Critério	HashMap	AVL	Rubro-Negra
Busca	$O(1)$	$O(\log N)$	$O(\log N)$
Inserção	$O(1)$	$O(\log N)$	$O(\log N)$
Ordenação	Não	Sim	Sim
Range Queries	Não	Sim	Sim
Rotações (ins.)	N/A	Mais	**Menos**
Uso ideal	Lookup puro	Read-heavy	Write-heavy + ordem

Regras de Ouro

- **HashMap:** Use por padrão quando só precisa de lookup/inserção/remoção rápida.
- **TreeMap (RB-Tree):** Use quando precisa de **ordem**, **range queries** (subMap, ceiling, floor) ou **próximo/anterior**.
- **AVL vs RB:** Se o cenário é **read-heavy** (muitas buscas, poucas alterações), AVL é ligeiramente melhor. Se há **muitas escritas**, RB-Tree é preferível.

27. Resumo Final

- RB-Trees são BSTs auto-balanceadas com **5 propriedades de cor** que garantem altura $O(\log N)$.
- Inserção: novo nó é **Vermelho**. Correção usa **recoloração** ou **rotações**.
- Remoção: usa conceito de **duplo preto**.
- Estrutura organizada **mais usada na indústria** (Linux, Java, C++).

28. Referências Bibliográficas

- **CLRS** – Introduction to Algorithms, Capítulo 13.
- **Sedgewick** – Algorithms, 4th Edition.
- **OpenJDK** – Source code de `java.util.TreeMap`.
- **Linux Kernel** – `include/linux/rbtree.h`.