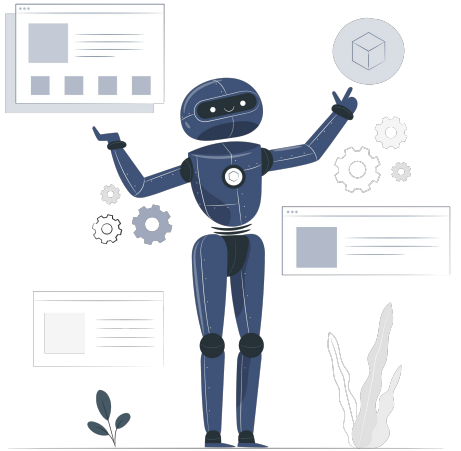
Árvores Rubro-negras

Lucas da Cruz Rezende

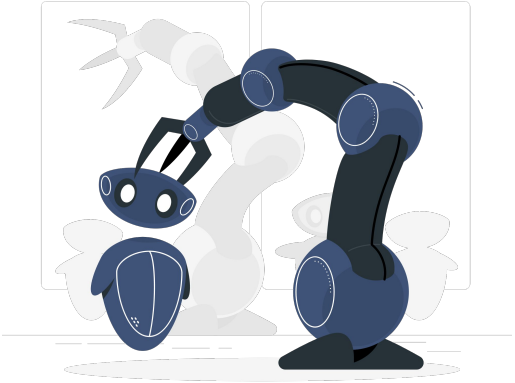
Introdução

As árvores rubro-negras são árvores binárias de busca balanceadas, onde cada nó é colorido de vermelho ou preto. Essas propriedades garantem que operações como inserção, remoção e busca sejam feitas em tempo logarítmico no pior caso.



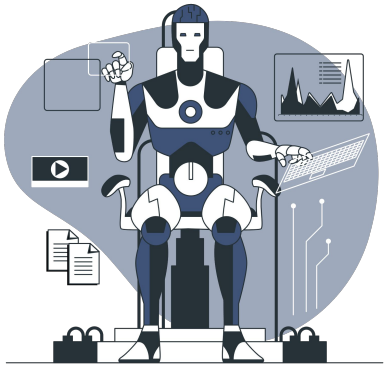
Motivação

- Manter o Balanceamento: Garante que a árvore esteja sempre balanceada, para operações rápidas.
- Facilidade de Implementação: Mais fácil de implementar do que outras árvores balanceadas.
- Desempenho Consistente: Mantém um tempo de resposta previsível para busca, inserção e remoção.
- Flexibilidade: Lida bem com inserções e remoções frequentes.
- Uso Comum: Usada em muitas bibliotecas para mapas e conjuntos devido à sua eficiência.



Áreas De Aplicação

- **Banco de Dados:** Índices rápidos para melhorar consultas.
- **Compiladores:** Gestão de tabelas de símbolos.
- **Sistemas Operacionais:** Memória e filas de prioridade eficientes.
- **Redes:** Roteamento de dados otimizado.
- **Linguagens:** Implementação de conjuntos e mapas eficientes (C++ std::map/set).



Regras Específicas Para Balanceamento.

1. Cada nó pode ser vermelho ou preto.
2. A raiz da árvore é sempre preta.
3. Nós nulos (folhas) são considerados pretos.
4. Se um nó é vermelho, seus filhos devem ser pretos. Em outras palavras, um nó vermelho nunca pode ter filhos vermelhos.
5. Qualquer caminho de um nó até um nó nulo deve conter o mesmo número de nós pretos.

Regras Específicas Para Balanceamento.

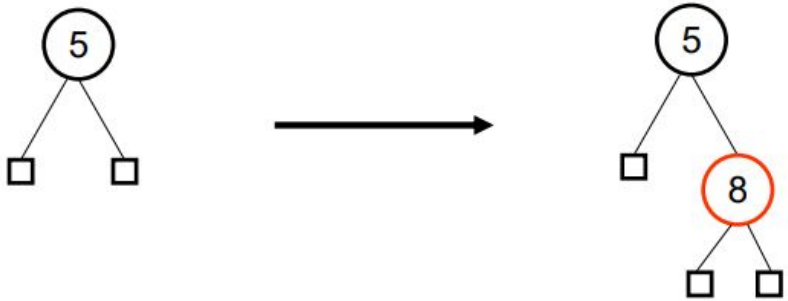
- Violação Vermelha: Ocorre quando um nó vermelho possui um filho vermelho, violando a Regra 4.
- Altura Preta: É o número de nós pretos desde o próprio nó até um nó nulo.
- Violação Preta: Acontece quando há duas alturas pretas diferentes para um nó, violando a Regra 5.

Regras Específicas Para Balanceamento.

Árvores rubro-negras, assim como as AVL, são autoajustáveis e mantêm um bom balanceamento, mas seus critérios são mais complexos. A inserção ou remoção de nós pode violar as regras, exigindo rotações e ajustes de cor. Apesar do balanceamento semelhante ao das AVL, a implementação das árvores rubro-negras é mais difícil de entender.

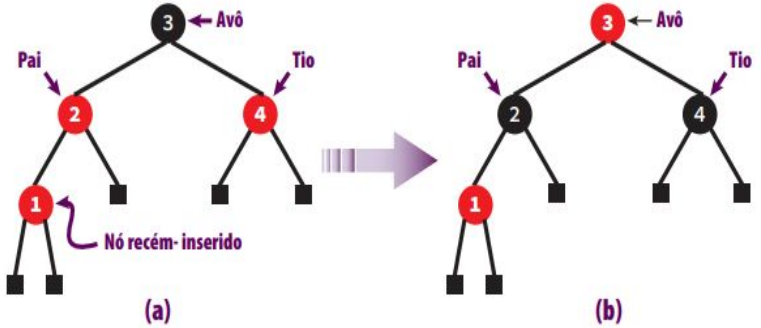
Inserção

Durante a inserção de um nó em uma árvore rubro-negra, as propriedades da árvore são mantidas ao definir a cor do novo nó como vermelha, e realizando rotações ou trocas de cores nos nós quando as regras da árvore forem violadas.



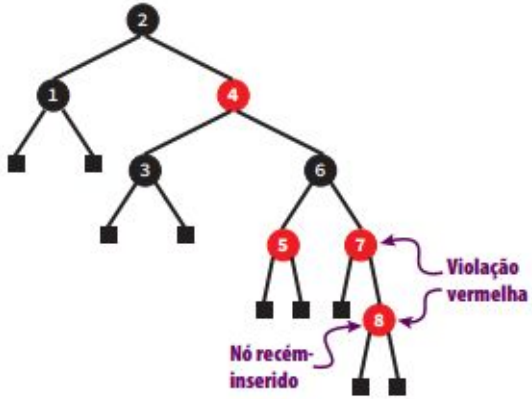
Caso 1 de Violação: Pai e Tio Vermelhos

Se o pai e o tio do nó recém-inserido forem vermelhos, o avô do nó deve ser preto. Nesse caso, basta trocar as cores do avô e de seus filhos, fazendo com que o avô se torne vermelho e seus filhos pretos, conforme mostrado na Figura



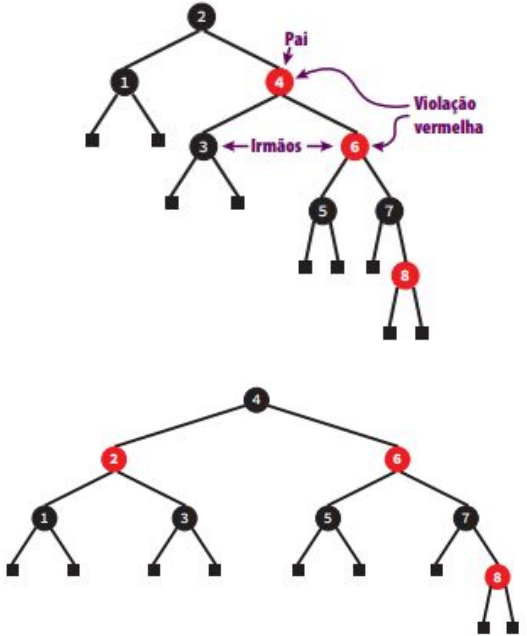
Caso 2 de Violação: Irmãos com Cores Diferentes e Pai Vermelho

Após a inserção de um nó contendo a chave 8 em uma árvore rubro-negra, ocorre uma violação de cor na subárvore com raiz no nó 6, que é corrigida temporariamente por uma mudança de cor. No entanto, essa mudança gera uma nova violação entre os nós 4 e 6. Como a alteração de cores não resolve essa nova violação , o caso se encaixa no Caso 2, onde apenas uma rotação pode solucionar o problema.



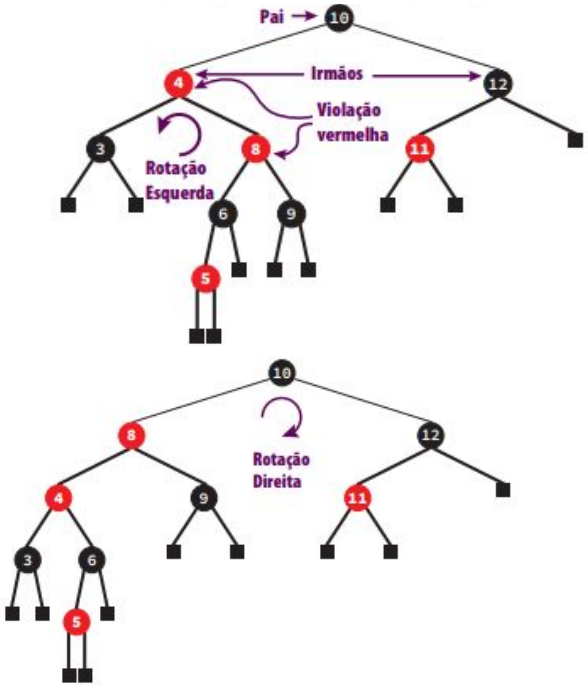
Caso 2 de Violação: Irmãos com Cores Diferentes e Pai Vermelho

Para resolver o problema uma rotação à esquerda simples é aplicada ao nó 4 em torno do nó 2. Com isso, o nó 4 se torna a nova raiz da subárvore, e o nó 2 se torna seu filho vermelho. Após a rotação, todas as violações de cor são corrigidas e as alturas pretas dos caminhos tornam-se iguais. Isso demonstra a importância de ajustar as cores juntamente com a rotação, garantindo o equilíbrio da árvore.



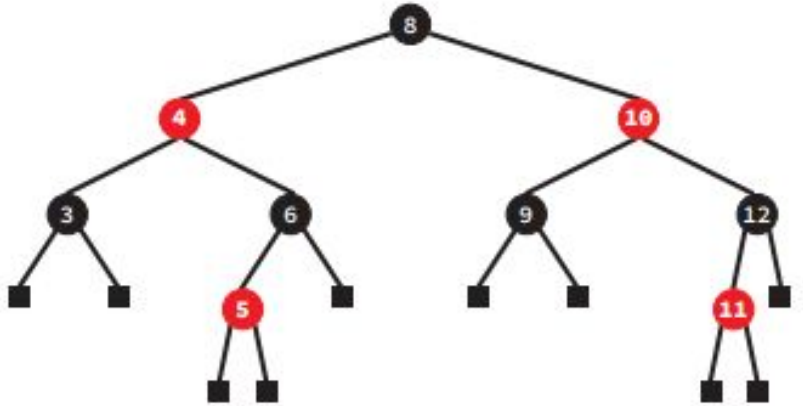
Caso 3 de Violação: Irmãos com Cores Diferentes e Pai Preto

No último caso de violação, quando o pai dos nós discutidos no Caso 2 é preto , uma rotação simples não é suficiente para resolver o problema. Nesse cenário, é necessário realizar uma rotação dupla, para corrigir a violação. A rotação dupla é crucial para restabelecer o equilíbrio da árvore.



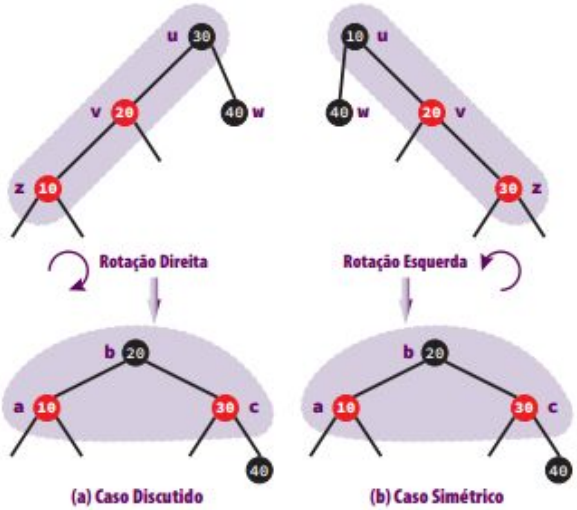
Caso 3 de Violação: Irmãos com Cores Diferentes e Pai Preto

Esse caso requer uma rotação dupla, e não apenas uma rotação simples, porque a primeira rotação aumenta a altura preta da subárvore esquerda do nó 4. Essa mudança desequilibraria a árvore, criando uma violação na contagem de nós pretos em alguns caminhos. A rotação dupla é usada justamente para evitar essa violação, garantindo que a altura preta permaneça consistente em todos os caminhos e que a árvore retorne ao seu estado equilibrado.



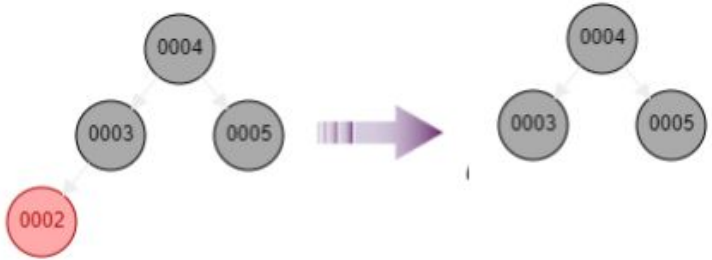
Simetria dos Casos de Violação

Cada caso descrito acima representa, de fato, dois casos combinados de violação.
O tratamento de cada par simétrico é similar, de forma que, do ponto de vista prático, para implementar a correção de um componente desse par, basta trocar filho esquerdo por filho direito e rotação esquerda por rotação direita e vice-versa.



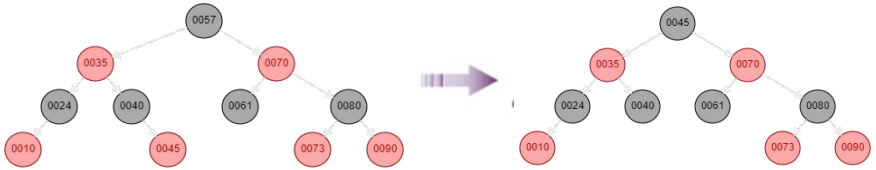
Remoção

Se o nó removido for vermelho, não ocorrerá nenhuma violação. Isso ocorre porque é impossível introduzir uma violação vermelha removendo-se um nó vermelho de uma árvore rubro-negra válida



Remoção

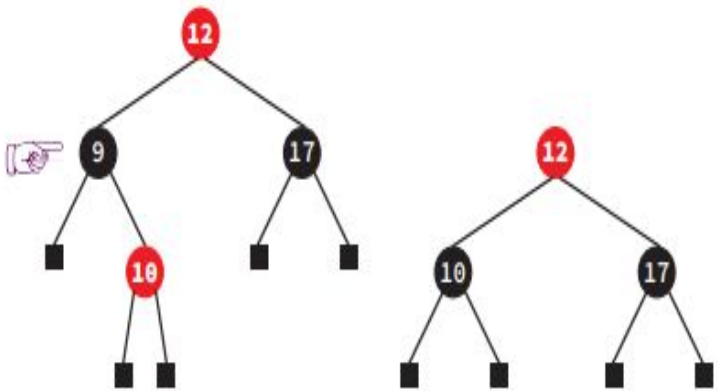
A remoção de um nó preto certamente causará uma violação preta do mesmo modo que a inserção de um nó preto causaria e é por isso que a inserção de nós pretos foi evitada. Além de causar violação preta, a remoção de um nó preto também pode causar uma violação vermelha



Caso 1: Nó Removido Preto com Filho Vermelho

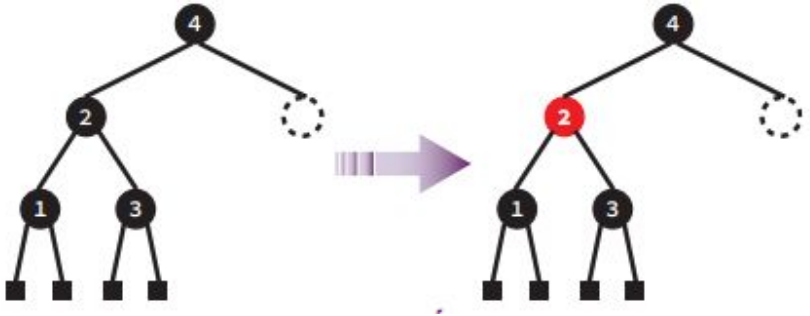
Quando um nó preto com um filho vermelho é removido, pode-se simplesmente colorir o filho vermelho de preto, evitando qualquer violação na árvore.

Se o nó a ser removido possui dois filhos, o procedimento envolve encontrar o sucessor imediato desse nó, copiar seu conteúdo para o nó a ser removido, e então remover o sucessor. Como o sucessor tem no máximo um filho, sua remoção pode ser tratada de forma simples, seguindo o procedimento descrito acima.



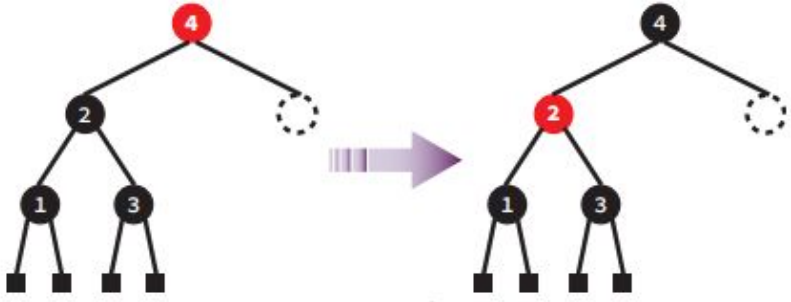
Caso 2: Irmão Preto e Sobrinho Preto

Se um nó é removido e o irmão é preto com ambos os filhos também pretos, a solução é relativamente simples: recolorir o irmão de vermelho. No entanto, isso pode causar uma violação preta que pode se propagar para cima na árvore



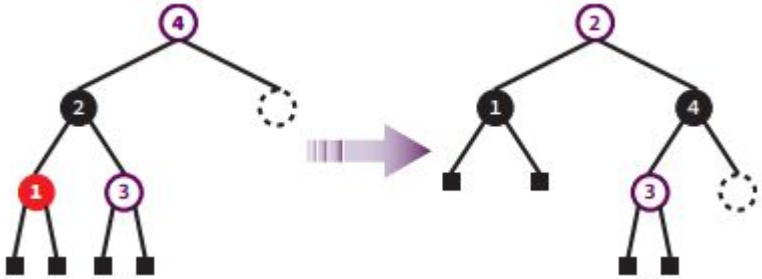
Caso 2: Irmão Preto e Sobrinho Preto

Nesse caso, pode-se recolorir o irmão do nó removido de vermelho sem causar uma violação vermelha. No entanto, isso pode causar uma violação preta, pois a altura preta da subárvore esquerda do nó 4 diminui, criando um desequilíbrio. Se o nó 4 era originalmente vermelho, recolorir o irmão de vermelho e o pai de preto resolve o problema, normalizando as alturas pretas até a raiz.



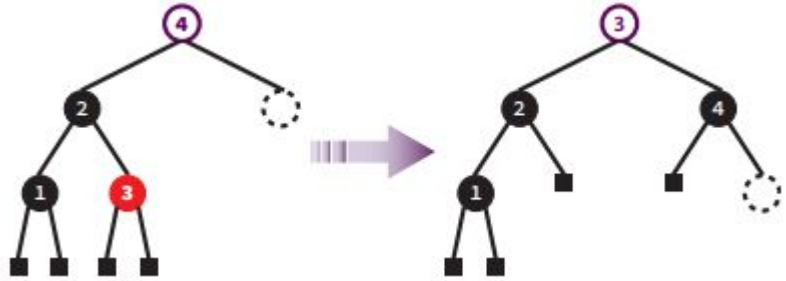
Caso 3: Irmão Preto e Sobrinho(s) Vermelho(s)

Quando um nó é removido e seu irmão preto tem um ou ambos os filhos vermelhos, há dois subcasos distintos. Se o filho da esquerda do irmão é vermelho, é necessária uma rotação simples. A cor do pai deve ser guardada antes da rotação, pois a rotação altera as cores dos nós pai e filho. Após a rotação, o novo pai é colorido com sua cor original e seus filhos se tornam pretos



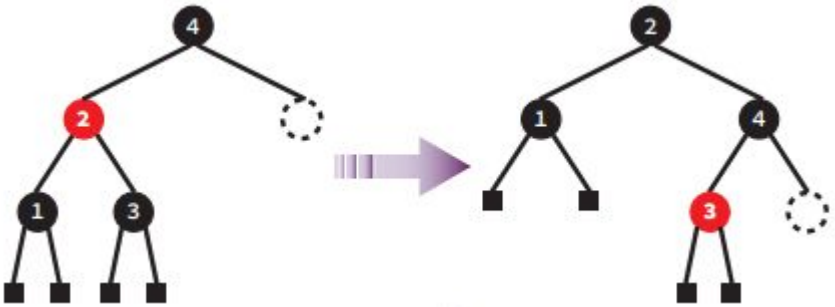
Caso 3: Irmão Preto e Sobrinho(s) Vermelho(s)

Após a rotação simples, a altura preta da subárvore é restaurada e nenhum ajuste adicional é necessário. Se o filho da direita do irmão do nó removido for vermelho, usa-se uma rotação dupla em vez de uma simples. As cores finais são semelhantes, e o filho da esquerda do irmão será preto nesse caso. Se o filho da esquerda for vermelho.



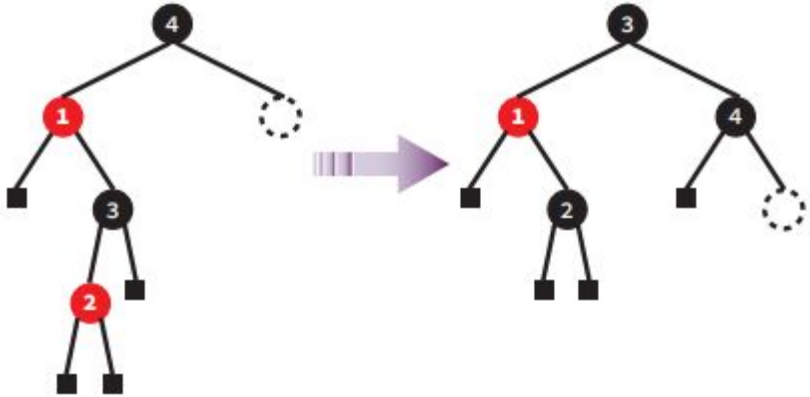
Caso 4: O Irmão do Nó Removido É Vermelho

Quando o irmão do nó removido é vermelho, realiza-se uma rotação simples e recolore-se o novo pai de preto e o filho da direita do irmão de vermelho. Se o irmão vermelho tem um filho não nulo, teste a cor dos filhos. Se ambos forem vermelhos, faça uma rotação dupla e recolore o novo pai e o filho da direita de preto e o filho da esquerda de vermelho.



Caso 4: O Irmão do Nó Removido É Vermelho

Quando há dois nós vermelhos, um deles se torna preto após a rotação para evitar violação da altura preta da subárvore. O último caso de violação ocorre quando o filho da direita do nó 4 é vermelho. Para resolver, realiza-se uma rotação simples do nó 3 em torno do nó 2, seguida de uma rotação dupla em torno do nó 6, o que reduz o problema ao caso anterior e restaura a estrutura e as cores desejadas.



Violações Devido a Inserções vs. Violações Devido a Remoções

Inserção	Remoção
Violações são corrigidas por meio de rotação e coloração de nós	Idem
Examinam-se as cores dos tios de um nó recém-inserido para verificar quando ocorre violação	Examina-se a cor do irmão do nó removido para verificar quando ocorre violação
A principal violação é a violação vermelha	A principal violação é a violação preta
Implementação é relativamente fácil	Implementação é bem mais complicada

Implementação - Inclusões e Definições

```cpp
#include <iostream> // Inclui a biblioteca para entrada e saída padrão
#include <list> // Inclui a biblioteca para usar listas

using namespace std; // Usar o namespace padrão para simplificar a sintaxe

enum Cor { VERMELHO, PRETO }; // Define um enum para as cores dos nós
```

Implementação - Estrutura do Nó

```
struct No {
    int valor; // Valor armazenado no nó
    Cor cor; // Cor do nó (VERMELHO ou PRETO)
    No *esquerda, *direita, *pai;
 // Ponteiros para o filho à esquerda, filho à direita e pai

    No(int valor) {
        this->valor = valor; // Inicializa o valor do nó
        esquerda = direita = pai = nullptr;
 // Inicializa os ponteiros para nullptr
        this->cor = VERMELHO; // Define a cor inicial como VERMELHO
    }
};
struct No {
    int valor; // Valor armazenado no nó
    Cor cor; // Cor do nó (VERMELHO ou PRETO)
    No *esquerda, *direita, *pai;
 // Ponteiros para o filho à esquerda, filho à direita e pai

    No(int valor) {
        this->valor = valor; // Inicializa o valor do nó
        esquerda = direita = pai = nullptr;
 // Inicializa os ponteiros para nullptr
        this->cor = VERMELHO; // Define a cor inicial como VERMELHO
    }
};
```

Implementação - Classe da Árvore Rubro-Negra

```
class ArvoreRN {
private:
    No* raiz; // Ponteiro para a raiz da árvore
    list<No*> ordemInsercao;
 // Lista para manter a ordem de inserção dos nós

protected:
    // Métodos para operações internas de manutenção da árvore
    void rotacaoEsquerda(No* &raiz, No* &no);
    void rotacaoDireita(No* &raiz, No* &no);
    void corrigirInsercao(No* &raiz, No* &no);
    void corrigirRemocao(No* &raiz, No* &no);
    No* minimo(No* no);
    void substituirNo(No* u, No* v);

public:
    ArvoreRN() { raiz = nullptr; }
 // Construtor da árvore, inicializa a raiz como nullptr

    // Métodos públicos para interagir com a árvore
    void inserir(int valor);
    void remover(int valor);
    void mostrarEmOrdem();
    void emOrdem();
    No* getRaiz() { return raiz; } // Retorna a raiz da árvore
};
```

Implementação - Rotação à Esquerda

```
void ArvoreRN::rotacaoEsquerda(No* &raiz, No* &no) {
    No* direitaNo = no->direita; // Nodo à direita do nó atual
    no->direita = direitaNo->esquerda;
 // Ajusta o filho esquerdo do nó direito para ser o filho direito do nó a
tual

    if (direitaNo->esquerda != nullptr)
        direitaNo->esquerda->pai = no;
 // Ajusta o pai do filho esquerdo do nó direito

    direitaNo->pai = no->pai; // Ajusta o pai do nó direito

    if (no->pai == nullptr)
        raiz = direitaNo; // Se o nó atual é a raiz, atualiza a raiz
    else if (no == no->pai->esquerda)
        no->pai->esquerda = direitaNo;
 // Ajusta o filho esquerdo do pai do nó atual
    else
        no->pai->direita = direitaNo;
 // Ajusta o filho direito do pai do nó atual

    direitaNo->esquerda = no;
 // Faz o nó atual ser o filho esquerdo do nó direito
    no->pai = direitaNo; // Ajusta o pai do nó atual
}
```

Implementação - Rotação à Direita

```
void ArvoreRN::rotacaoDireita(No* &raiz, No* &no) {
    No* esquerdaNo = no->esquerda; // Nodo à esquerda do nó atual
    no->esquerda = esquerdaNo->direita;
 // Ajusta o filho direito do nó esquerdo para ser o filho esquerdo do nó
atual

    if (esquerdaNo->direita != nullptr)
        esquerdaNo->direita->pai = no;
 // Ajusta o pai do filho direito do nó esquerdo

    esquerdaNo->pai = no->pai; // Ajusta o pai do nó esquerdo

    if (no->pai == nullptr)
        raiz = esquerdaNo; // Se o nó atual é a raiz, atualiza a raiz
    else if (no == no->pai->esquerda)
        no->pai->esquerda = esquerdaNo;
 // Ajusta o filho esquerdo do pai do nó atual
    else
        no->pai->direita = esquerdaNo;
 // Ajusta o filho direito do pai do nó atual

    esquerdaNo->direita = no;
 // Faz o nó atual ser o filho direito do nó esquerdo
    no->pai = esquerdaNo; // Ajusta o pai do nó atual
}
```

Implementação - Inserção de Nó

```
void ArvoreRN::inserir(int valor) {
    No* novoNo = new No(valor); // Cria um novo nó
    No* paiNo = nullptr; // Ponteiro para o pai do novo nó
    No* atual = raiz; // Ponteiro para percorrer a árvore

    // Encontra o local para inserir o novo nó
    while (atual != nullptr) {
        paiNo = atual; // Atualiza o pai do nó
        if (novoNo->valor < atual->valor)
            atual = atual->esquerda; // Vai para o filho à esquerda
        else
            atual = atual->direita; // Vai para o filho à direita
    }

    novoNo->pai = paiNo; // Define o pai do novo nó

    if (paiNo == nullptr)
        raiz = novoNo;
// Se a árvore estava vazia, o novo nó se torna a raiz
    else if (novoNo->valor < paiNo->valor)
        paiNo->esquerda = novoNo;
// Adiciona o novo nó como filho à esquerda
    else
        paiNo->direita = novoNo;
// Adiciona o novo nó como filho à direita

    novoNo->esquerda = novoNo->direita = nullptr;
// Define os filhos do novo nó como nullptr
    novoNo->cor = VERMELHO; // Define a cor do novo nó como vermelho

    ordemInsercao.push_back(novoNo);
// Adiciona o nó à lista de inserção

    corrigirInsercao(raiz, novoNo); // Corrige a árvore após a inserção
    mostrarEmOrdem(); // Exibe a árvore após a inserção
}
```

Implementação - Correção Após Inserção

```
void ArvoreRN::corrigirInsercao(No* &raiz, No* &no) {
    No* paiNo = nullptr; // Ponteiro para o pai do nó
    No* avoNo = nullptr; // Ponteiro para o avô do nó

    // Ajusta a árvore enquanto o nó não for a raiz e o pai for vermelho
    while ((no != raiz) && (no->cor != PRETO) && (no->pai->cor ==
 VERMELHO)) {
        paiNo = no->pai; // Define o pai do nó
        avoNo = no->pai->pai; // Define o avô do nó

        if (paiNo == avoNo->esquerda) {
            No* tioNo = avoNo->direita; // Define o tio do nó

            if (tioNo != nullptr && tioNo->cor == VERMELHO) {
                // Caso 1: Tio é vermelho
                avoNo->cor = VERMELHO;
 // Muda a cor do avô para vermelho
                paiNo->cor = PRETO; // Muda a cor do pai para preto
                tioNo->cor = PRETO; // Muda a cor do tio para preto
                no = avoNo; // Move o nó para o avô
            } else {
                if (no == paiNo->direita) {
                    // Caso 2: O nó está à direita do pai
                    rotacaoEsquerda(raiz, paiNo);
 // Realiza rotação à esquerda
                    no = paiNo; // Atualiza o nó
                    paiNo = no->pai; // Atualiza o pai
                }
                // Caso 3: O nó está à esquerda do pai
                rotacaoDireita(raiz, avoNo);
 // Realiza rotação à direita
                swap(paiNo->cor, avoNo->cor);
 // Troca as cores do pai e do avô
                no = paiNo; // Atualiza o nó
            }
        } else {
            No* tioNo = avoNo->esquerda; // Define o tio do nó
            if (tioNo != nullptr && tioNo->cor == VERMELHO) {
                // Caso 1: Tio é vermelho
                avoNo->cor = VERMELHO;
 // Muda a cor do avô para vermelho
                paiNo->cor = PRETO; // Muda a cor do pai para preto
                tioNo->cor = PRETO; // Muda a cor do tio para preto
                no = avoNo; // Move o nó para o avô
            } else {
                if (no == paiNo->esquerda) {
                    // Caso 2: O nó está à esquerda do pai
                    rotacaoDireita(raiz, paiNo);
 // Realiza rotação à direita
                    no = paiNo; // Atualiza o nó
                    paiNo = no->pai; // Atualiza o pai
                }
                // Caso 3: O nó está à direita do pai
                rotacaoEsquerda(raiz, avoNo);
 // Realiza rotação à esquerda
                swap(paiNo->cor, avoNo->cor);
 // Troca as cores do pai e do avô
                no = paiNo; // Atualiza o nó
            }
        }
    }

    raiz->cor = PRETO; // Garante que a raiz seja preta
}
```

Implementação - Substituir Nó

```
void ArvoreRN::substituirNo(No* u, No* v) {
    if (u->pai == nullptr)
        raiz = v; // Se o nó a ser substituído é a raiz, atualiza a raiz
    else if (u == u->pai->esquerda)
        u->pai->esquerda = v; // Substitui o nó à esquerda do pai
    else
        u->pai->direita = v; // Substitui o nó à direita do pai

    if (v != nullptr)
        v->pai = u->pai; // Atualiza o pai do nó substituto
}
```

Implementação - Remoção de Nó

```
void ArvoreRN::remover(int valor) {
    No* z = raiz;
    // Encontra o nó a ser removido
    while (z != nullptr) {
        if (valor < z->valor)
            z = z->esquerda;
        else if (valor > z->valor)
            z = z->direita;
        else
            break;
    }

    if (z == nullptr)
        return; // O valor não foi encontrado

    No* y = z; // Nó a ser removido
    Cor corOriginalY = y->cor;
 // Guarda a cor original do nó a ser removido
    No* x;

    if (z->esquerda == nullptr) {
        x = z->direita; // Se o nó não tem filho à esquerda
        substituirNo(z, z->direita);
    } else if (z->direita == nullptr) {
        x = z->esquerda; // Se o nó não tem filho à direita
        substituirNo(z, z->esquerda);
    } else {
```

```
        y = minimo(z->direita);
 // Encontrar o nó mínimo na subárvore direita
        corOriginalY = y->cor; // Guarda a cor original do nó mínimo
        x = y->direita;

        if (y->pai == z)
            x->pai = y;
        else {
            substituirNo(y, y->direita); // Substitui o nó mínimo
            y->direita = z->direita;
            y->direita->pai = y;
        }

        substituirNo(z, y); // Substitui o nó a ser removido
        y->esquerda = z->esquerda;
        y->esquerda->pai = y;
        y->cor = z->cor;
    }

    delete z; // Remove o nó da memória

    if (corOriginalY == PRETO)
        corrigirRemocao(raiz, x); // Corrige a árvore após a remoção

    ordemInsercao.remove(z); // Remove o nó da lista de inserção
    mostrarEmOrdem(); // Exibe a árvore após a remoção
}
```

Implementação - Correção Após Remoção

```
void ArvoreRN::corrigirRemocao(No* &raiz, No* &x) {
    while (x != raiz && (x == nullptr || x->cor == PRETO)) {
        if (x == x->pai->esquerda) {
            No* w = x->pai->direita; // Irmão do nó x
            if (w->cor == VERMELHO) {
                w->cor = PRETO;
                x->pai->cor = VERMELHO;
                rotacaoEsquerda(raiz, x->pai);
                w = x->pai->direita;
            }

            if ((w->esquerda == nullptr || w->esquerda->cor == PRETO) &&
                (w->direita == nullptr || w->direita->cor == PRETO)) {
                w->cor = VERMELHO;
                x = x->pai;
            } else {
                if (w->direita == nullptr || w->direita->cor == PRETO) {
                    if (w->esquerda != nullptr)
                        w->esquerda->cor = PRETO;
                    w->cor = VERMELHO;
                    rotacaoDireita(raiz, w);
                    w = x->pai->direita;
                }
                w->cor = x->pai->cor;
                x->pai->cor = PRETO;
                if (w->direita != nullptr)
                    w->direita->cor = PRETO;
                rotacaoEsquerda(raiz, x->pai);
                x = raiz;
            }
        } else {
            No* w = x->pai->esquerda; // Irmão do nó x
            if (w->cor == VERMELHO) {
                w->cor = PRETO;
                x->pai->cor = VERMELHO;
                rotacaoDireita(raiz, x->pai);
                w = x->pai->esquerda;
            }

            if ((w->direita == nullptr || w->direita->cor == PRETO) &&
                (w->esquerda == nullptr || w->esquerda->cor == PRETO)) {
                w->cor = VERMELHO;
                x = x->pai;
            } else {
                if (w->esquerda == nullptr || w->esquerda->cor ==
PRETO) {
                    if (w->direita != nullptr)
                        w->direita->cor = PRETO;
                    w->cor = VERMELHO;
                    rotacaoEsquerda(raiz, w);
                    w = x->pai->esquerda;
                }
                w->cor = x->pai->cor;
                x->pai->cor = PRETO;
                if (w->esquerda != nullptr)
                    w->esquerda->cor = PRETO;
                rotacaoDireita(raiz, x->pai);
                x = raiz;
            }
        }
    }
    if (x != nullptr)
        x->cor = PRETO; // Garante que o nó x seja preto
}
```

Prova

Passo Base:

- Para uma árvore com 1 nó
  - Altura: 0
  - Fórmula: $h \leq 2\log2(1+1)$
  - Cálculo: $h \leq 2 \times 1 = 2$
- A fórmula é verdadeira para uma árvore com 1 nó.

Prova

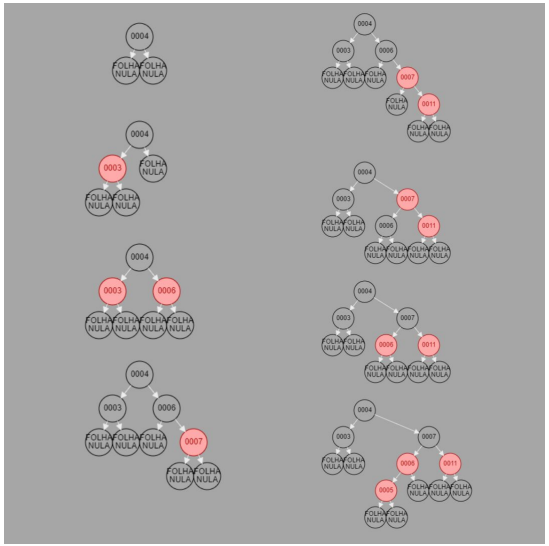
Passo Indutivo:

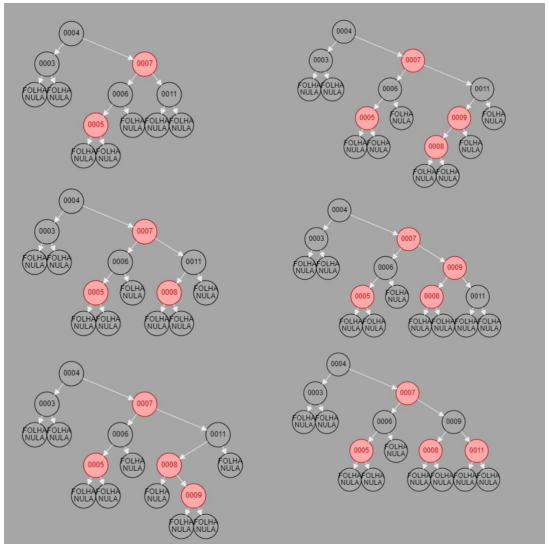
- Suponha que para uma árvore com nnn nós, a fórmula é verdadeira: $h \leq 2\log_2(n+1)$
  - Adicionamos um nó, resultando em n+1 nós.
  - A nova fórmula é: $h \leq 2\log_2((n+1)+1)$ $h \leq 2\log_2(n+2)$
  - A fórmula continua válida porque $\log_2(n+2)$ é muito semelhante a $\log_2(n+1)$.
- A altura de uma árvore rubro-negra com nnn nós é $O(\log n)$.

Exercício 1

Desenhe a árvore rubro-negra resultante da inserção (nesta ordem) dos nós contendo as chaves 4, 3, 6, 7, 11, 5, 8 e 9, considerando que árvore está inicialmente vazia.

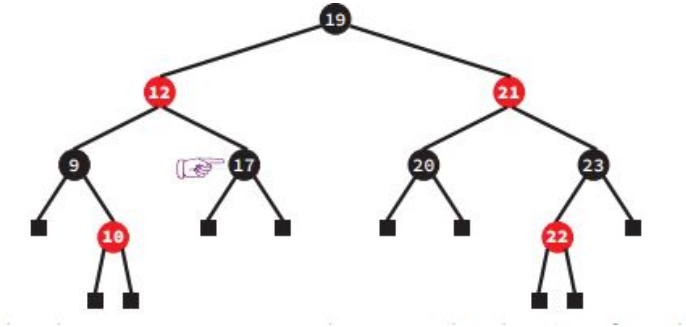
Exercício 1 Resolvido



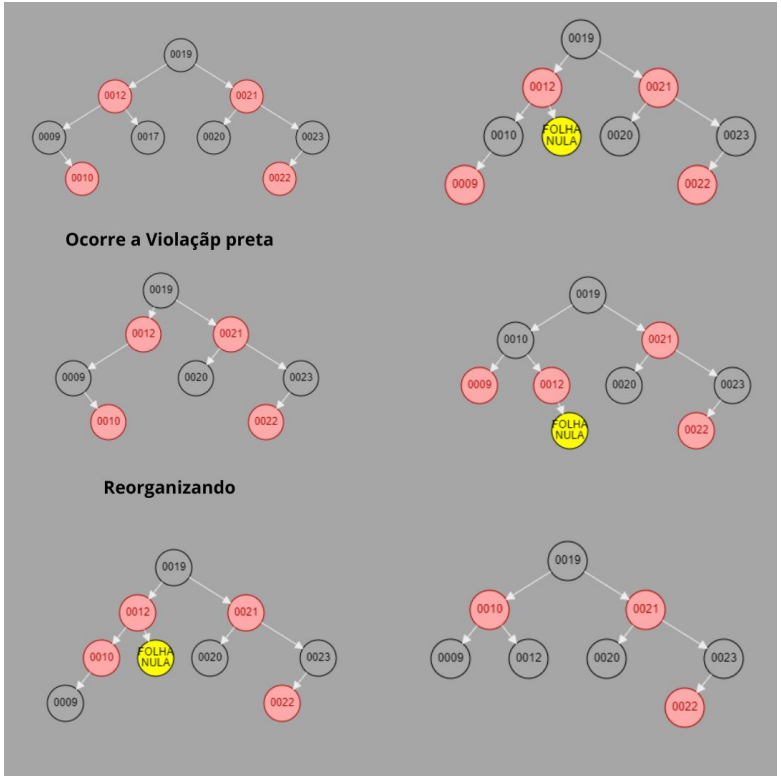


Exercício 2

(a) Que tipo de violação ocorre após a remoção do nó contendo a chave 17 na figura abaixo? (b) Como essa violação pode ser corrigida?



Exercício 2 Resolvido



OBRIGADO PELA ATENÇÃO!

Alguma pergunta?

Referências

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

KNUTH, Donald E. The Art of Computer Programming: Sorting and Searching. 2. ed. Boston: Addison-Wesley, 1998.

SEDGEWICK, Robert; WAYNE, Kevin. Algorithms. 4. ed. Boston: Addison-Wesley, 2011.

TARJAN, Robert E. Data Structures and Network Algorithms. Philadelphia: Society for Industrial and Applied Mathematics, 1983.

OLIVEIRA, Ulysses de. Estruturas de Dados Usando a Linguagem C: Volume 2: Busca e Ordenação. 1. ed. São Paulo: Érica, 2019.