Engenharia de Computação

Análise e Projeto de Algoritmos

Seminário - Árvore Rubro Negra

Professor: Anderson Grandi Pires

Aluno:
Lucas da Cruz Rezende

Leopoldina, MG

09 de setembro de 2024

Sumário

1 Introdução

As árvores rubro-negras são um tipo de árvore binária de busca balanceada que desempenham um papel crucial na ciência da computação, especialmente em algoritmos e estruturas de dados. Inventadas por Rudolf Bayer em 1972, essas árvores foram inicialmente chamadas de B-árvores binárias simétricas e receberam seu nome atual em 1978, graças ao trabalho de Leo J. Guibas e Robert Sedgewick.

Uma árvore rubro-negra é caracterizada por suas propriedades específicas de balanceamento, onde cada nó é colorido de vermelho ou preto. Essas propriedades garantem que a árvore permaneça aproximadamente balanceada, permitindo que operações básicas como inserção, remoção e busca sejam realizadas em tempo logarítmico no pior caso. A estrutura e as regras de coloração das árvores rubro-negras asseguram que nenhum caminho da raiz até uma folha seja mais do que duas vezes mais longo que qualquer outro, o que é essencial para a eficiência dos algoritmos que as utilizam.

2 Motivação e áreas de aplicação

As árvores rubro-negras são estruturas de dados fundamentais em ciência da computação, especialmente em algoritmos e estruturas de dados. Elas garantem que a árvore permaneça aproximadamente balanceada, assegurando que as operações básicas (inserção, remoção e busca) possam ser realizadas em tempo logarítmico, ou seja, O(logn). A principal motivação para o uso de árvores rubro-negras é a eficiência. Manter a árvore balanceada é crucial para sistemas que lidam com grandes volumes de dados, pois permite que as operações sejam realizadas de forma rápida e eficiente. Sem esse balanceamento, as operações poderiam se tornar muito lentas, especialmente em casos de pior cenário.

As árvores rubro-negras são amplamente utilizadas para implementar índices em sistemas de banco de dados. Esses índices permitem buscas rápidas e eficientes, melhorando significativamente o desempenho das consultas. Em compiladores, as árvores rubro-negras são usadas para gerenciar tabelas de símbolos. Essas tabelas armazenam informações sobre variáveis e funções durante o processo de compilação, facilitando a análise e a geração de código. Diversas estruturas de dados internas dos sistemas operacionais, como a gestão de memória e a implementação de filas de prioridade, utilizam árvores rubro-negras. Elas ajudam a garantir que essas operações sejam realizadas de maneira eficiente.

Em redes, as árvores rubro-negras são empregadas em algoritmos de roteamento e em estruturas de dados para armazenar informações de roteamento. Isso é essencial para garantir a eficiência e a rapidez na transmissão de dados. Muitas bibliotecas padrão de linguagens de programação, como C++ (std::map e std::set), utilizam árvores rubro-negras para implementar conjuntos e mapas. Isso permite que essas estruturas de dados sejam eficientes e rápidas.

As árvores rubro-negras são uma ferramenta poderosa e versátil em ciência da com-

putação. Sua capacidade de manter a eficiência em operações básicas as torna indispensáveis em diversas áreas, desde sistemas de banco de dados até compiladores e sistemas operacionais. A compreensão e o uso adequado dessa estrutura de dados podem trazer grandes benefícios para o desenvolvimento de software e a gestão de dados.

3 Conceitos e funcionamento

3.1 Conceitos

Uma árvore rubro-negra é um tipo de árvore binária de busca que segue um conjunto de regras específicas:

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.

Uma violação vermelha ocorre quando um nó vermelho possui um filho que também é vermelho, violando a Regra 4. Já a altura preta de um nó é o número de nós pretos desde o próprio nó até um nó nulo, e, de acordo com a Regra 5, todas as alturas pretas de um nó devem ser iguais. Uma violação preta acontece quando um nó apresenta duas alturas pretas diferentes.

Assim como as árvores AVL, as árvores rubro-negras são autoajustáveis e garantem um balanceamento aceitável. No entanto, os critérios de balanceamento das árvores rubro-negras são mais complexos e implícitos nas regras que definem sua estrutura. Quando um nó é inserido ou removido, violações dessas regras podem ocorrer, exigindo rotações e ajustes de cor para corrigir o problema.

Embora o balanceamento em árvores rubro-negras resulte em algo similar ao das árvores AVL, sua implementação é mais difícil de entender.

Nós serão representados como mostra a Figura 1.

Figura 1: Nós Usados para Representar Árvores Rubro-negras

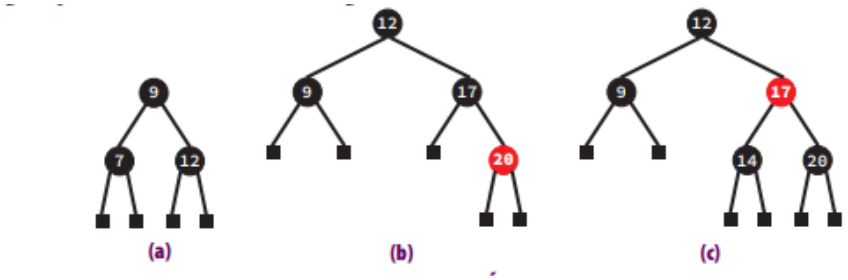

Figura 2: Exemplos de Árvores Rubro-negras

A Figura 3 ilustra três exemplos de árvores que violam as regras que definem as árvores rubro-negras:

- Na Figura 3 (a), ocorre uma violação da Regra 5, pois há uma diferença nas alturas pretas. Seguindo o caminho pelo filho esquerdo do nó 9 até o nó nulo, encontram-se 2 nós (o próprio 9 e o nó nulo). No entanto, pelo filho direito do nó 9 até qualquer filho do nó 12, encontram-se 3 nós pretos. Isso caracteriza uma violação preta.
- Na Figura 3 (b), a violação ocorre devido à Regra 4. O nó 17 é vermelho e não poderia ter um filho vermelho, o que caracteriza uma violação vermelha.
- Na Figura 3 (c), também há uma violação preta. A violação ocorre porque, em algum caminho da árvore, a quantidade de nós pretos difere entre dois trajetos até nós nulos, o que viola a regra de igualdade das alturas pretas.

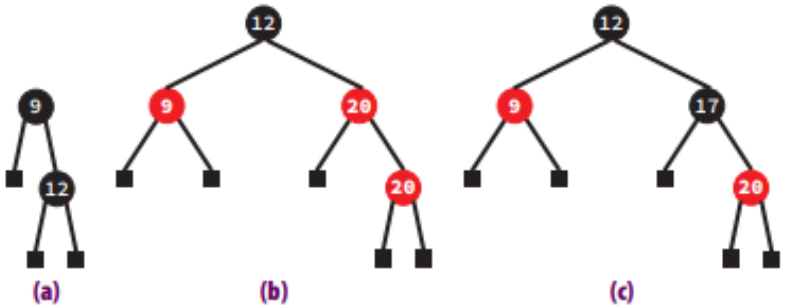

Figura 3: Violações de Regras em Árvores Rubro-negras

As regras 3 e 4 das árvores rubro-negras garantem que o menor caminho possível, obtido alternando nós vermelhos e pretos, é no máximo duas vezes maior do que o menor caminho composto apenas por nós pretos. Essa relação implica que as árvores rubro-negras são razoavelmente balanceadas, o que garante uma eficiência aceitável nas operações de busca, inserção e remoção.

3.2 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. Inserir nós como vermelhos preserva a altura preta da árvore (Regra 5), mas pode resultar em uma violação da regra das cores vermelhas. Caso o novo nó fosse preto, isso geraria um caminho mais longo de nós pretos, violando as propriedades da árvore. Portanto, o nó inserido deve ser vermelho, com o risco de ocorrer uma violação vermelha. No entanto, essa abordagem é vantajosa por dois motivos:

1. Há a possibilidade de não ocorrer nenhuma violação.
2. Uma violação vermelha é mais simples de resolver do que uma violação preta.

Se o pai do novo nó for preto, não haverá nenhuma violação. No entanto, se o pai for vermelho, ocorre uma violação vermelha e a árvore precisará ser reajustada. A Regra 2, que lida com a violação preta, só é violada quando a árvore está vazia e o novo nó se torna a raiz. A Regra 4, que trata da violação vermelha, é quebrada quando o pai do novo nó é vermelho.

Para determinar se ocorre uma violação vermelha após a inserção, verifica-se se algum nó vermelho no caminho de inserção tem um filho vermelho. Note que, logo após uma inserção, apenas um filho de um nó vermelho poderá ser vermelho, já que, se houvesse mais de um filho vermelho, uma violação vermelha já teria ocorrido antes, e a árvore não seria válida.

Após a inserção de um nó, existem três casos principais de violação vermelha que precisam ser tratados.

3.2.1 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 [ver Figura 4 (a)]. 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 4 (b).

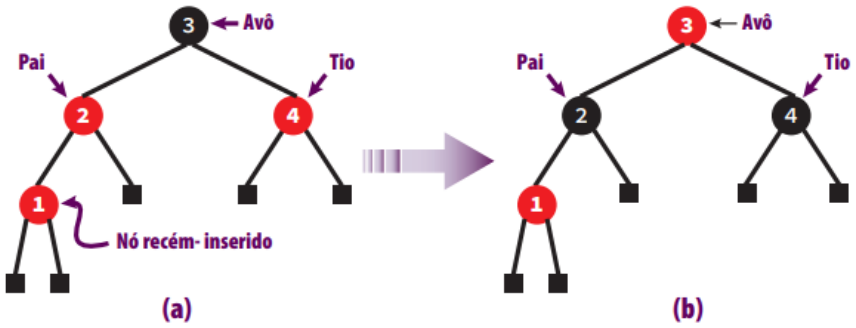

Figura 4: Caso 1 de Violação Devido a Inserção 1

Como a cor do avô passou a ser vermelha, pode ocorrer uma nova violação vermelha no nível superior da árvore. Isso acontece se o pai do avô (que não aparece na Figura 4) também for vermelho, gerando outra violação vermelha. Portanto, após resolver esse caso, o processo de ajustes continua até que não haja mais violações.

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

A Figura 5 mostra a árvore após a inserção do nó 8, e uma mudança de cor é necessária, já que a subárvore com raiz no nó 6 se enquadra no Caso 1. A mudança de cor, ilustrada na Figura 6, causa uma violação vermelha entre os nós 4 e 6. Nesse caso, a alteração de cores não resolve o problema, pois, se o nó 4 se tornar preto, a altura preta dos caminhos que passam por ele seria maior que a dos caminhos que passam pelo nó 1, violando as propriedades da árvore.

Portanto, como a situação corresponde ao Caso 2, apenas uma rotação pode resolver o problema.

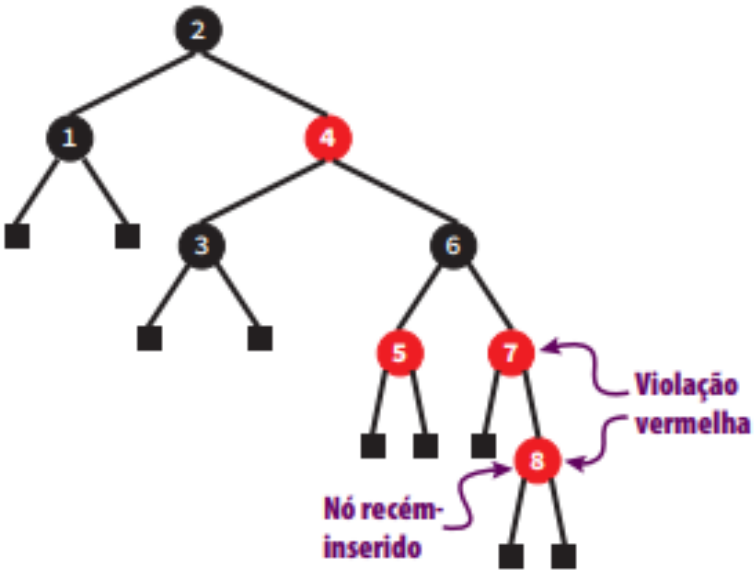

Figura 5: Caso 1 de Violação Devido a Inserção 2

A rotação utilizada para resolver o problema ilustrado na Figura 6 é uma rotação à esquerda simples do nó contendo 4 em torno do nó contendo 2, conforme visto na Figura 7. Após essa rotação, não há mais violação preta, pois todos os caminhos a partir da raiz da subárvore, tanto pela direita quanto pela esquerda, têm três nós pretos.

Como mostra a Figura 7, após a rotação, o nó contendo 4 torna-se a nova raiz, e o nó contendo 2 se torna seu filho vermelho. Agora, não há mais violação vermelha e as alturas

pretas são iguais. Esse exemplo demonstra a necessidade de ajustar as cores junto com a rotação.

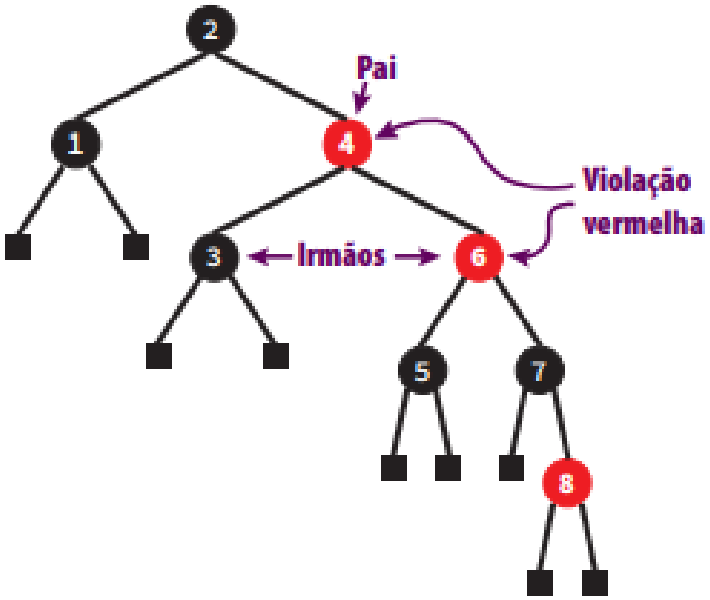

Figura 6: Caso 2 de Violação Devido a Inserção 1

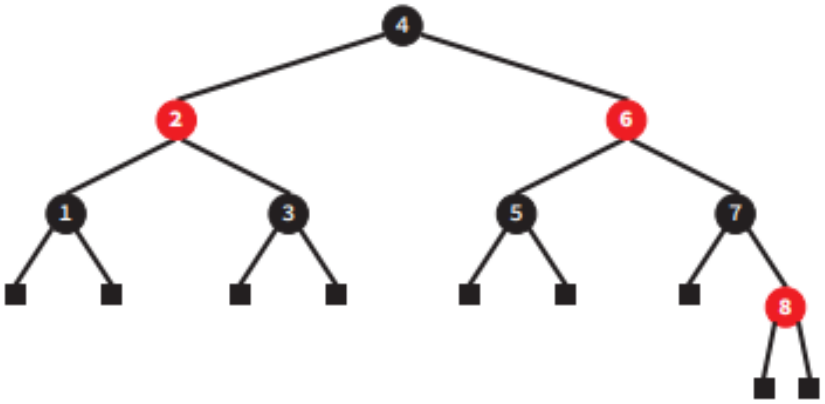

Figura 7: Caso 2 de Violação Devido a Inserção 2

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

O último caso de violação ocorre quando o pai dos nós discutidos no Caso 2 é preto, como ilustrado na Figura 8. Nesse cenário, uma rotação simples não é suficiente para resolver o problema. Em vez disso, é necessária uma rotação dupla, para corrigir a estrutura da árvore.

A Figura 9 mostra o resultado após a primeira rotação da rotação dupla, e a Figura 10 ilustra o resultado final após a segunda rotação. Essas rotações duplas ajustam a árvore para resolver a violação e restaurar as propriedades da árvore rubro-negra.

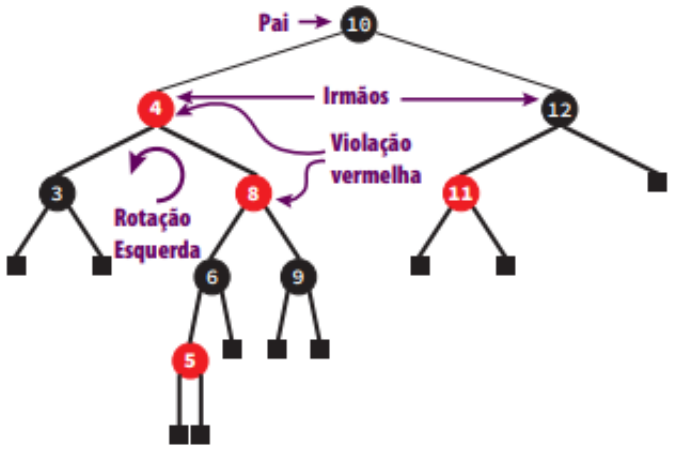

Figura 8: Caso 3 de Violação Devido a Inserção 1

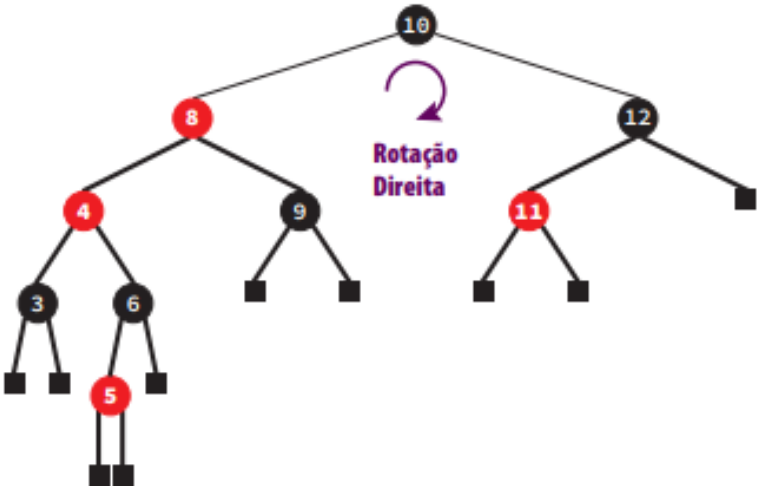

Figura 9: Caso 3 de Violação Devido a Inserção 2

Esse caso requer uma rotação dupla em vez de uma rotação simples porque a primeira rotação (como mostrado na Figura 9) pode aumentar a altura preta da subárvore esquerda do nó contendo 4, o que poderia introduzir uma violação preta na árvore.

A rotação dupla é utilizada para evitar essa violação, ajustando a estrutura da árvore de maneira que a altura preta permaneça balanceada e as propriedades da árvore rubro-negra sejam preservadas. A rotação dupla é necessária para garantir que a árvore permaneça válida após a correção da violação vermelha.

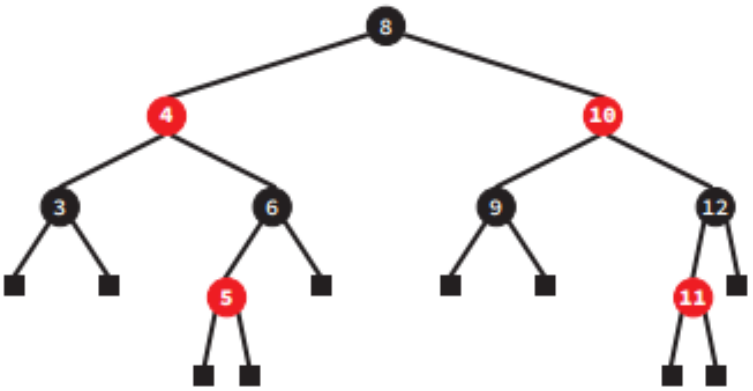

Figura 10: Caso 3 de Violação Devido a Inserção 3

3.2.4 Simetria dos Casos de Violação

Cada caso de violação após a inserção de um nó representa, na verdade, dois casos combinados de violação, resultando em seis possibilidades distintas, em vez de apenas três. A Figura 11 mostra a simetria do Caso 1, a Figura 12 mostra a simetria do Caso 2 e a Figura 13 ilustra o Caso 3 de violação.

O tratamento de cada par simétrico é similar: para corrigir um componente de um par, basta trocar o filho esquerdo pelo direito e substituir a rotação à esquerda pela rotação à direita, e vice-versa. Essa abordagem pode tornar a codificação extensa, justificando a escolha de uma implementação mais eficiente.

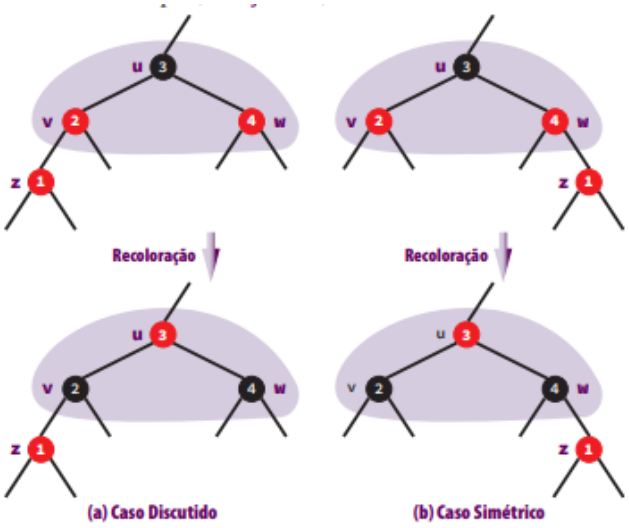

Figura 11: Simetria do Caso 1 de Violação Devido a Inserção

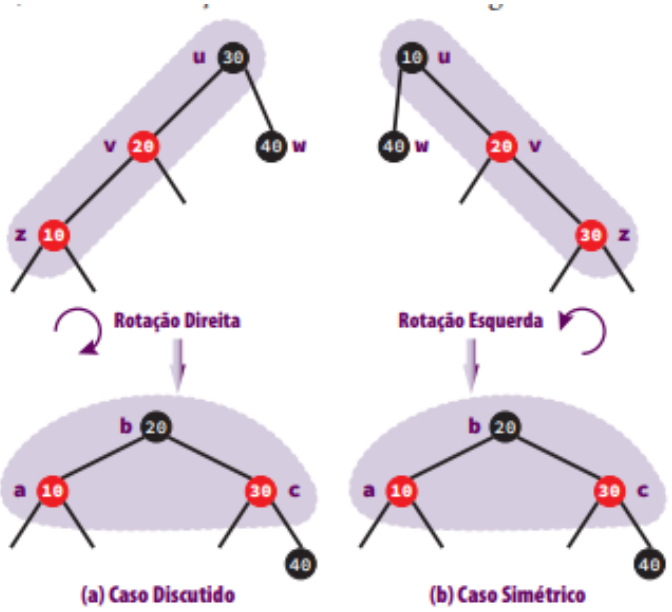

Figura 12: Simetria do Caso 2 de Violação Devido a Inserção

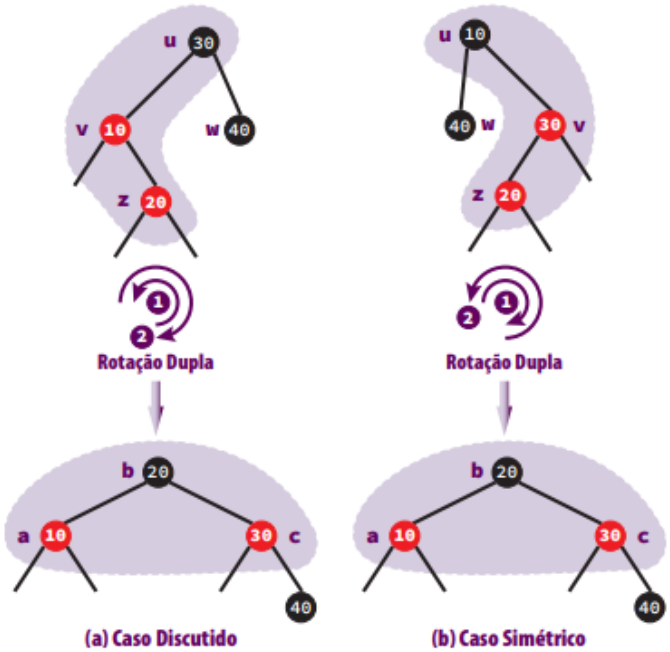

Figura 13: Simetria do Caso 3 de Violação Devido a Inserção

3.3 Remoção

Durante uma operação de inserção em uma árvore rubro-negra, a cor do novo nó pode ser escolhida para forçar e corrigir uma violação vermelha, que é relativamente fácil de resolver. No entanto, durante a remoção de um nó, a cor não pode ser escolhida e, portanto, a remoção de um nó vermelho não causa uma violação, pois não afeta a altura preta da árvore. Isso ocorre porque a altura preta não leva em conta nós vermelhos.

Por outro lado, a remoção de um nó preto pode causar uma violação preta, similar ao que ocorre quando um nó preto é inserido. Isso se deve ao fato de que a remoção de um nó preto pode reduzir a altura preta dos caminhos na árvore. Além disso, a remoção de um nó preto também pode causar uma violação vermelha.

Se o nó a ser removido tiver apenas um filho não nulo, ele pode ser substituído por esse filho. Se o nó não tiver filhos não nulos, ele é substituído por um nó nulo.

3.3.1 Caso 1: Nó Removido Preto com Filho Vermelho

Se o nó a ser removido é preto e tem um filho vermelho, basta colorir esse filho de preto, pois isso resolve o problema sem causar violações nas propriedades da árvore. Esse é um caso simples de ajuste, chamado Caso 1. Quando o nó a ser removido tem dois filhos, o sucessor imediato é encontrado e seu conteúdo é copiado para o nó original. Em seguida, o sucessor é removido, já que ele possui no máximo um filho.

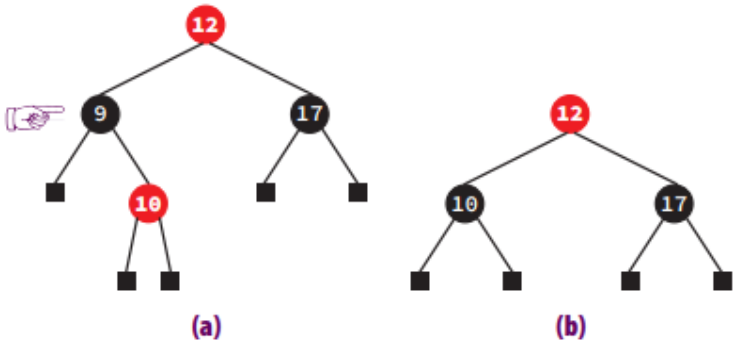

Figura 14: Caso 1 de Reajuste após Remoção em Árvore Rubro-negra

3.3.2 Caso 2: Irmão Preto e Sobrinho Preto

Quando um nó é removido e seu irmão é preto com ambos os filhos também pretos, o ajuste é relativamente simples, mas pode gerar uma propagação para cima na árvore. Esse é um dos casos de correção da estrutura da árvore e ocorre após a remoção de um nó, conforme ilustrado na Figura 15, onde o nó tracejado indica o nó removido.

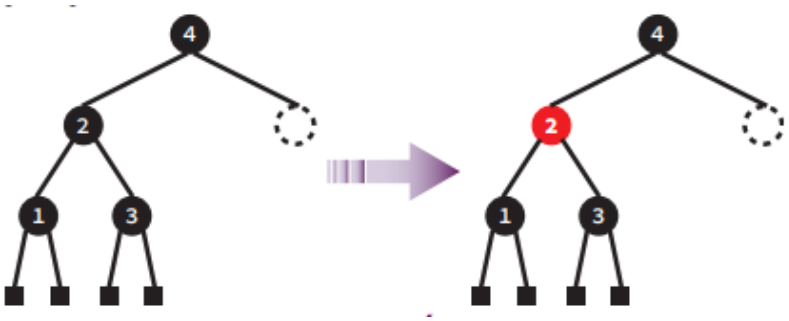

Figura 15: Remoção em Árvore Rubro-negra 1

Nesse caso, pode-se recolorir o irmão do nó removido (o nó com conteúdo 2) para vermelho sem violar as regras de cor da árvore. No entanto, há um risco de violação da altura preta, pois a subárvore à esquerda do nó com conteúdo 4 teve sua altura preta reduzida. Embora as alturas pretas de ambas as subárvores do nó contendo 4 fiquem equilibradas, se o nó 4 for a raiz de uma subárvore, pode haver uma violação na altura preta do pai dele, já que a subárvore do irmão do nó 4 não teve sua altura preta alterada.

Se o pai do nó removido (neste caso, o nó contendo 4) era originalmente vermelho, é possível recolorir o irmão de vermelho e o pai de preto. Com isso, as alturas pretas serão ajustadas, e o problema será resolvido até a raiz da árvore. Isso ocorre porque, ao ajustar as cores, as propriedades da árvore rubro-negra são mantidas sem alterar a estrutura geral da altura preta.

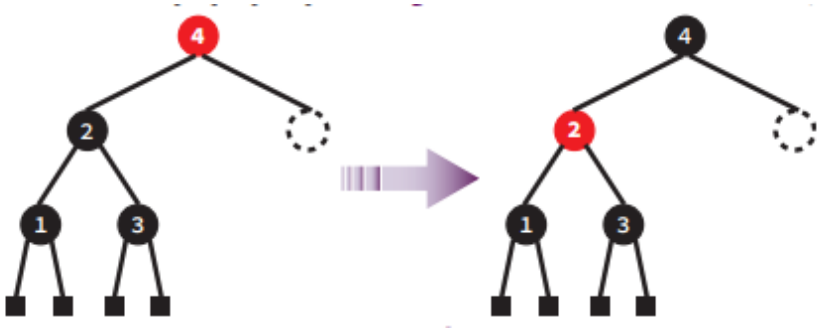

Figura 16: Remoção em Árvore Rubro-negra 2

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

Os casos mais complexos de correção de violações após a remoção de um nó ocorrem quando o irmão do nó removido é preto e pelo menos um dos filhos do irmão é vermelho. Existem dois subcasos principais nesse cenário.

No primeiro subcaso, se o filho esquerdo do irmão é vermelho, é suficiente realizar uma rotação simples. No entanto, a cor do pai pode ser vermelha ou preta, então é importante

preservar essa cor para restaurá-la após a rotação, pois o processo de rotação não leva em conta as cores anteriores do novo e do antigo pai. Após a rotação, o novo pai deve ser recolorido com sua cor original, enquanto seus filhos são coloridos de preto, garantindo que as propriedades da árvore rubro-negra sejam mantidas. A Figura 17 ilustra esse processo, onde o nó sem cor indica que sua cor não é relevante para a discussão.

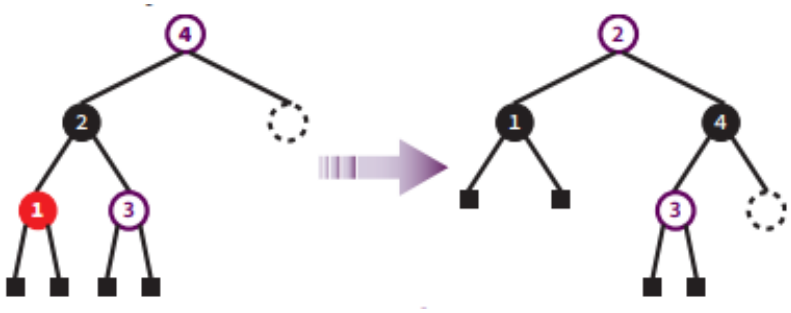

Figura 17: Remoção em Árvores Rubro-negras 3

Após a última alteração na configuração da subárvore, a altura preta é restaurada, e nenhum outro ajuste será necessário na árvore. O mesmo processo se aplica quando o filho direito do irmão do nó removido é vermelho, mas, neste caso, uma rotação dupla é utilizada em vez de uma rotação simples. As cores finais serão as mesmas. Lembre-se de que, nessa situação, o filho esquerdo do irmão deve ser preto. Se ele for vermelho, aplica-se a resolução do Caso 2 (como mostrado na Figura 18).

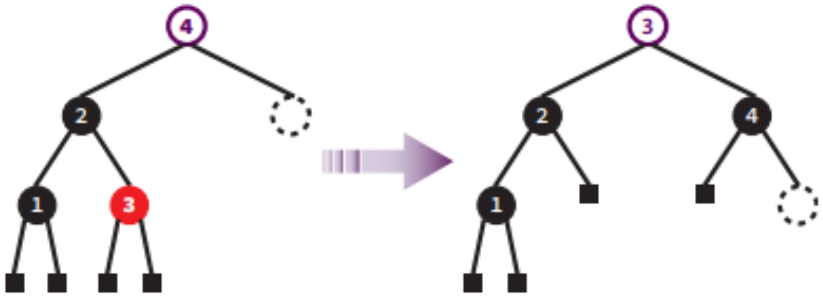

Figura 18: Remoção em Árvores Rubro-negras 4

Esses são os únicos cenários em que o irmão do nó removido é preto. No entanto, se o irmão for vermelho, a situação se complica um pouco mais, exigindo ajustes adicionais para corrigir as violações nas propriedades da árvore rubro-negra.

3.3.4 Caso 4: O Irmão do Nó Removido É Vermelho

Quando o nó removido tem um irmão vermelho, sempre é necessária pelo menos uma rotação para corrigir a estrutura da árvore. Se o irmão for vermelho, ambos os seus filhos

serão pretos. Nesse caso, uma rotação simples pode ser realizada, o novo pai é recolorido como preto e o filho direito do irmão é colorido como vermelho, restaurando o balanceamento da árvore, conforme mostrado na Figura 19. O nó contendo 3 é colorido de vermelho porque é seguro fazê-lo sem causar violações adicionais.

No segundo caso, em que o irmão vermelho tem um filho não nulo, basta identificar qual dos filhos é vermelho. Se ambos forem vermelhos, uma rotação dupla é realizada. Após a rotação, o novo pai é colorido de preto, o filho direito de preto, e o filho esquerdo de vermelho, como ilustra a Figura 20. Isso ajusta a altura preta da subárvore direita sem afetar a subárvore esquerda.

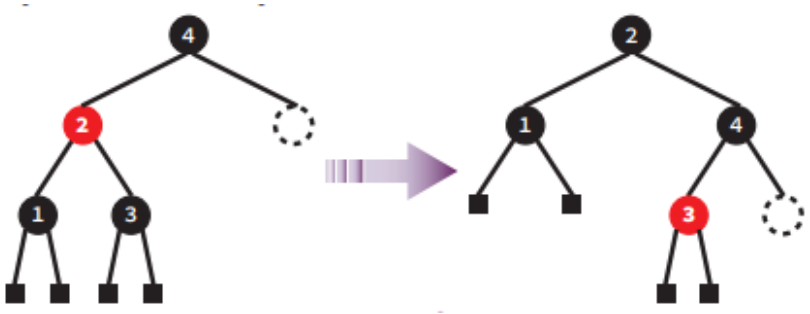

Figura 19: Remoção em Árvores Rubro-negras 5

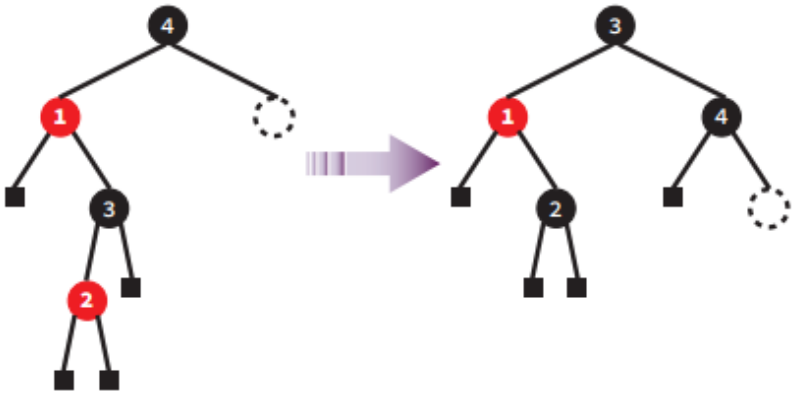

Figura 20: Remoção em Árvores Rubro-negras 6

Quando há dois nós vermelhos envolvidos, um deles é convertido para preto após a rotação, enquanto o outro é deixado vermelho para evitar qualquer violação da altura preta na subárvore de onde ele foi removido.

Finalmente, se o filho direito do nó contendo 4 for vermelho, esse cenário pode ser reduzido ao caso anterior. Uma rotação simples é realizada com o nó contendo 3 em torno do nó contendo 2, seguida por uma rotação dupla ao redor do nó contendo 6. Isso resulta na estrutura correta, com as cores restauradas conforme mostrado na Figura 21.

3.3.5 Violações Devido a Inserções vs. Violações Devido a Remoções

Inserção	**Remoção**
Examinam-se as cores dos tios do nó inserido para detectar violações.	Examinam-se as cores dos tios do nó inserido para detectar violações.
A principal violação é de cor vermelha.	A principal violação é de cor preta.
A implementação é simples.	A implementação é mais complexa.

Tabela 1: Comparação entre Inserção e Remoção em árvores balanceadas

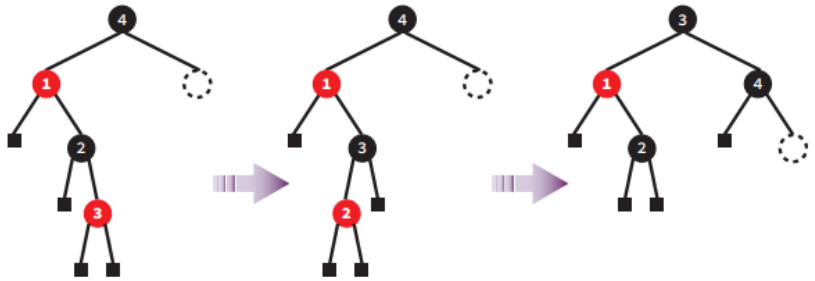

Figura 21: Remoção em Árvores Rubro-negras 7

4 Implementação

Abaixo está a implementação da árvore rubro-negra com inserção e remoção, bem como as funções para manutenção das propriedades de balanceamento.

4.1 Estrutura do Nó e Enumeração de Cores

Cada nó na árvore rubro-negra é representado pela estrutura No, que inclui:

- valor: O valor armazenado no nó.
- cor: A cor do nó (VERMELHO ou PRETO), essencial para manter o balanceamento da árvore.
- esquerda e direita: Ponteiros para os filhos esquerdo e direito, respectivamente.
- pai: Ponteiro para o pai do nó.

```
enum Cor { VERMELHO, PRETO };

struct No {
    int valor;
    Cor cor;
    No *esquerda, *direita, *pai;

    No(int valor) {
        this->valor = valor;
        esquerda = direita = pai = nullptr;
        this->cor = VERMELHO;
    }
};
```

Algoritmo 1: Definição do Nó e enumeração de cores

4.2 Funções de Rotação

As rotações são operações essenciais para manter a árvore balanceada. Elas garantem que as propriedades da árvore rubro-negra sejam preservadas.

Rotação à Esquerda: Ajusta os ponteiros para promover o nó direito a pai e o nó atual a filho esquerdo do novo pai.

Rotação à Direita: Ajusta os ponteiros para promover o nó esquerdo a pai e o nó atual a filho direito do novo pai.

```
void rotacaoEsquerda(No* &raiz, No* &no) {
    No* direitaNo = no->direita;
    no->direita = direitaNo->esquerda;

    if (direitaNo->esquerda != nullptr)
        direitaNo->esquerda->pai = no;

    direitaNo->pai = no->pai;

    if (no->pai == nullptr)
        raiz = direitaNo;
    else if (no == no->pai->esquerda)
        no->pai->esquerda = direitaNo;
    else
        no->pai->direita = direitaNo;

    direitaNo->esquerda = no;
    no->pai = direitaNo;
}

void rotacaoDireita(No* &raiz, No* &no) {
    No* esquerdaNo = no->esquerda;
```

```
    no->esquerda = esquerdaNo->direita;

    if (esquerdaNo->direita != nullptr)
        esquerdaNo->direita->pai = no;

    esquerdaNo->pai = no->pai;

    if (no->pai == nullptr)
        raiz = esquerdaNo;
    else if (no == no->pai->esquerda)
        no->pai->esquerda = esquerdaNo;
    else
        no->pai->direita = esquerdaNo;

    esquerdaNo->direita = no;
    no->pai = esquerdaNo;
}
```

Algoritmo 2: Rotação à Esquerda e à Direita

4.3 Função de Inserção e corrigir após inserção

A inserção de um novo nó segue o padrão de uma árvore binária de busca, mas com ajustes adicionais para manter as propriedades da árvore rubro-negra:

Passos da Inserção:

1. Inserção Padrão: Insere o nó como faria em uma árvore binária de busca.
2. Correção: Ajusta a árvore para manter suas propriedades após a inserção, chamando corrigirInsercao.

A função corrigirInsercao garante que a árvore mantenha suas propriedades após a inserção:

Passos da Correção:

1. Verificação de Propriedades: Verifica se o nó pai é vermelho e realiza as correções necessárias.
2. Rotações e Troca de Cores: Realiza rotações e troca de cores conforme necessário para restaurar as propriedades da árvore.

```
void inserir(int valor) {
    No* novoNo = new No(valor);
    No* paiNo = nullptr;
    No* atual = raiz;
```

```
5
6       while (atual != nullptr) {
7           paiNo = atual;
8           if (novoNo->valor < atual->valor)
9               atual = atual->esquerda;
10          else
11              atual = atual->direita;
12      }
13
14      novoNo->pai = paiNo;
15
16      if (paiNo == nullptr)
17          raiz = novoNo;
18      else if (novoNo->valor < paiNo->valor)
19          paiNo->esquerda = novoNo;
20      else
21          paiNo->direita = novoNo;
22
23      novoNo->esquerda = novoNo->direita = nullptr;
24      novoNo->cor = VERMELHO;
25
26      corrigirInsercao(raiz, novoNo);
27  }
28
29  void corrigirInsercao(No* &raiz, No* &no) {
30      No* paiNo = nullptr;
31      No* avoNo = nullptr;
32
33      while ((no != raiz) && (no->cor != PRETO) && (no->pai->cor ==
            VERMELHO)) {
34          paiNo = no->pai;
35          avoNo = no->pai->pai;
36
37          if (paiNo == avoNo->esquerda) {
38              No* tioNo = avoNo->direita;
39
40              if (tioNo != nullptr && tioNo->cor == VERMELHO) {
41                  avoNo->cor = VERMELHO;
42                  paiNo->cor = PRETO;
43                  tioNo->cor = PRETO;
44                  no = avoNo;
45              } else {
46                  if (no == paiNo->direita) {
47                      rotacaoEsquerda(raiz, paiNo);
48                      no = paiNo;
49                      paiNo = no->pai;
50                  }
51                  rotacaoDireita(raiz, avoNo);
52                  swap(paiNo->cor, avoNo->cor);
53                  no = paiNo;
```

```
54              }
55          } else {
56              No* tioNo = avoNo->esquerda;
57
58              if (tioNo != nullptr && tioNo->cor == VERMELHO) {
59                  avoNo->cor = VERMELHO;
60                  paiNo->cor = PRETO;
61                  tioNo->cor = PRETO;
62                  no = avoNo;
63              } else {
64                  if (no == paiNo->esquerda) {
65                      rotacaoDireita(raiz, paiNo);
66                      no = paiNo;
67                      paiNo = no->pai;
68                  }
69                  rotacaoEsquerda(raiz, avoNo);
70                  swap(paiNo->cor, avoNo->cor);
71                  no = paiNo;
72              }
73          }
74      }
75      raiz->cor = PRETO;
76  }
```

Algoritmo 3: Função de Inserção e Correção

4.4 Função de Remoção e corrigir após remoção

A remoção de um nó é mais complexa e envolve ajustes adicionais para garantir que a árvore permaneça balanceada:

Passos da Remoção:

1. Encontrar o Nó a Ser Removido: Localiza o nó a ser removido na árvore.
2. Substituir e Ajustar: Substitui o nó removido pelo seu sucessor ou predecessor e ajusta a árvore para manter suas propriedades.

A função corrigirRemocao ajusta a árvore após a remoção de um nó para garantir que as propriedades da árvore rubro-negra sejam mantidas:

Passos da Correção:

1. Verificação de Propriedades: Verifica se o nó é preto e realiza correções conforme necessário.
2. Rotações e Troca de Cores: Realiza rotações e troca de cores para restaurar o balanceamento da árvore.

```
void remover(int valor) {
    No* z = raiz;
    No* y;
    No* x;

    while (z != nullptr && z->valor != valor) {
        if (valor < z->valor)
            z = z->esquerda;
        else
            z = z->direita;
    }

    if (z == nullptr)
        return;

    y = z;
    Cor corOriginalY = y->cor;

    if (z->esquerda == nullptr) {
        x = z->direita;
        substituirNo(z, z->direita);
    } else if (z->direita == nullptr) {
        x = z->esquerda;
        substituirNo(z, z->esquerda);
    } else {
        y = minimo(z->direita);
        corOriginalY = y->cor;
        x = y->direita;

        if (y->pai == z)
            x->pai = y;
        else {
            substituirNo(y, y->direita);
            y->direita = z->direita;
            y->direita->pai = y;
        }

        substituirNo(z, y);
        y->esquerda = z->esquerda;
        y->esquerda->pai = y;
        y->cor = z->cor;
    }

    delete z;

    if (corOriginalY == PRETO)
        corrigirRemocao(raiz, x);
}
void corrigirRemocao(No* &raiz, No* x) {
    while (x != raiz && (x == nullptr || x->cor == PRETO)) {
```

```
51          if (x == x->pai->esquerda) {
52              No* w = x->pai->direita;
53
54              if (w->cor == VERMELHO) {
55                  w->cor = PRETO;
56                  x->pai->cor = VERMELHO;
57                  rotacaoEsquerda(raiz, x->pai);
58                  w = x->pai->direita;
59              }
60
61              if ((w->esquerda == nullptr || w->esquerda->cor == PRETO) &&
62                  (w->direita == nullptr || w->direita->cor == PRETO)) {
63                  w->cor = VERMELHO;
64                  x = x->pai;
65              } else {
66                  if (w->direita == nullptr || w->direita->cor == PRETO) {
67                      if (w->esquerda != nullptr)
68                          w->esquerda->cor = PRETO;
69                      w->cor = VERMELHO;
70                      rotacaoDireita(raiz, w);
71                      w = x->pai->direita;
72                  }
73                  w->cor = x->pai->cor;
74                  x->pai->cor = PRETO;
75                  if (w->direita != nullptr)
76                      w->direita->cor = PRETO;
77                  rotacaoEsquerda(raiz, x->pai);
78                  x = raiz;
79              }
80          } else {
81              No* w = x->pai->esquerda;
82
83              if (w->cor == VERMELHO) {
84                  w->cor = PRETO;
85                  x->pai->cor = VERMELHO;
86                  rotacaoDireita(raiz, x->pai);
87                  w = x->pai->esquerda;
88              }
89
90              if ((w->direita == nullptr || w->direita->cor == PRETO) &&
91                  (w->esquerda == nullptr || w->esquerda->cor == PRETO)) {
92                  w->cor = VERMELHO;
93                  x = x->pai;
94              } else {
95                  if (w->esquerda == nullptr || w->esquerda->cor == PRETO) {
96                      if (w->direita != nullptr)
97                          w->direita->cor = PRETO;
98                      w->cor = VERMELHO;
99                      rotacaoEsquerda(raiz, w);
100                     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;
}
```

Algoritmo 4: Rotação à Esquerda e à Direita

5 Prova

Lema 1: Qualquer árvore rubro-negra com raiz r tem pelo menos $2^{AP(r)} - 1$ nós não nulos, sendo que $AP(r)$ é a altura preta da árvore.

Prova por indução:

Base da indução: Se a altura preta $AP(r) = 0$, então não há nós internos, ou seja, $n = 0$. Isso é verdade, pois

$$2^0 - 1 = 0.$$

Assim, a base da indução está provada.

Hipótese indutiva: Suponha que o lema seja válido para $AP(r) < k$.

Conclusão: Precisamos mostrar que o lema também vale para $AP(r) = k + 1$. Se $AP(r) = k$, então cada subárvore de r tem altura preta $k - 1$. Assim, usando a hipótese indutiva, o número mínimo de nós dessa árvore é dado por:

$$2^{k-1} - 1 + 2^{k-1} - 1 + 1 = 2^k - 1 = 2^{(k+1)-1} - 1$$

Isso completa a prova para $AP(r) = k + 1$.

Lema 2: Seja a a altura de uma árvore rubro-negra cuja raiz é r. Então, $AP(r) \geq \frac{a}{2}$.

Prova: Considere um caminho desde a raiz até uma folha. Se existir um nó vermelho nesse caminho, deve haver um nó preto correspondente, pois a Regra 4 exige que não haja dois nós vermelhos consecutivos. Assim, pelo menos metade dos nós nesse caminho deve ser preto. Portanto, a altura preta da raiz, $AP(r)$, deve ser pelo menos metade da altura total a, isto é, $AP(r) \geq \frac{a}{2}$.

Teorema F.1: Qualquer árvore rubro-negra com n nós não nulos possui altura a tal que:

$$a \leq 2 \cdot \log_2(n + 1).$$

Prova: Seja a a altura de uma árvore rubro-negra com raiz r. De acordo com o Lema F.1, temos que:

$$n \geq 2^{AP(r)} - 1.$$

De acordo com o Lema F.2, temos que:

$$AP(r) \geq \frac{a}{2}.$$

Substituindo a altura preta $AP(r)$ na inequação do Lema F.1, obtemos:

$$n \geq 2^{\frac{a}{2}} - 1.$$

Adicionando 1 a ambos os lados, temos:

$$n + 1 \geq 2^{\frac{a}{2}}.$$

Aplicando o logaritmo de base 2 em ambos os lados, obtemos:

$$\log_2(n+1) \geq \frac{a}{2}.$$

Multiplicando ambos os lados por 2, temos:

$$2 \cdot \log_2(n+1) \geq a.$$

Assim, obtemos a desigualdade desejada:

$$a \leq 2 \cdot \log_2(n+1).$$

Corolário F.1: A altura de qualquer árvore rubro-negra com n nós é $O(\log n)$.

Prova: De acordo com o Teorema F.1, temos que:

$$a \leq 2 \cdot \log_2(n+1).$$

Portanto, a altura a é proporcional ao logaritmo de n, e a prova é trivial.

Qualquer operação básica sobre uma árvore rubro-negra tem complexidade temporal $O(\log n)$. Como foi visto na Seção 4.2, rotações alteram apenas alguns poucos ponteiros, de modo que o custo temporal é $O(1)$. O custo temporal de uma alteração de cor de algum nó também é $O(1)$. Portanto, o custo de reestruturação de um único nó de uma árvore rubro-negra é $O(1)$.

O número máximo de operações de recoloração de nós após uma inserção é igual à altura máxima de uma árvore rubro-negra. Assim, de acordo com o Teorema F.1, o número máximo de tais operações é menor ou igual a $2 \cdot \log_2(n+1)$.

Árvores AVL e árvores rubro-negras permitem que todas as operações básicas em tabelas de busca tenham custo temporal $O(\log n)$. Árvores AVL são mais balanceadas do que árvores rubro-negras, mas requerem mais rotações em operações de inserção e remoção. Portanto, se houver mais inserções e remoções do que buscas, é melhor usar árvores rubro-negras. De outra forma, é melhor usar árvores AVL.

6 Exercícios

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.

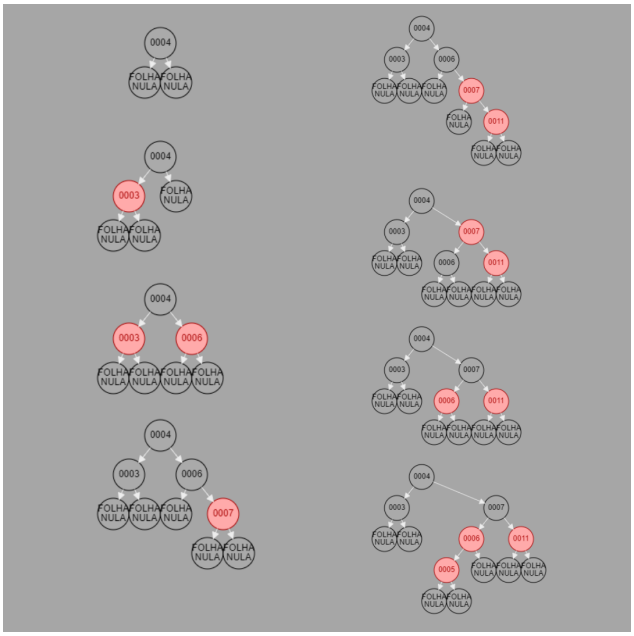

Figura 22: Resolvendo exercício 1

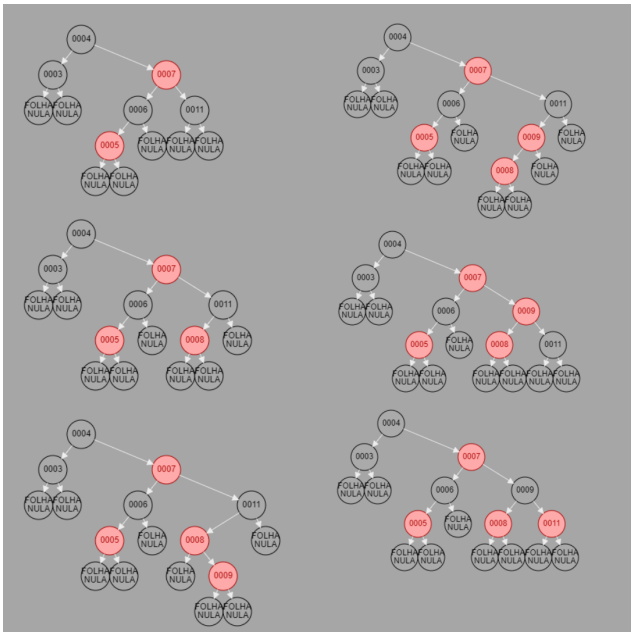

Figura 23: Resolvendo exercício 1

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?

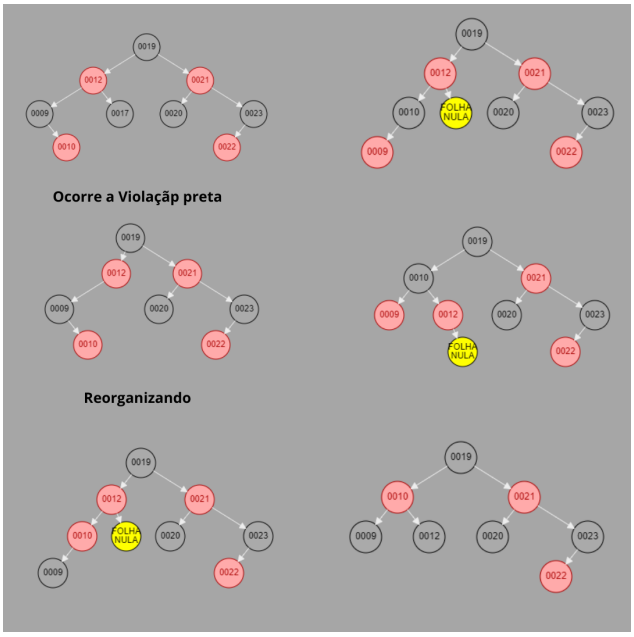

Figura 24: Resolvendo exercício 2

7 Conclusão

A árvore rubro-negra se destaca como uma poderosa estrutura de dados, oferecendo um equilíbrio eficiente entre inserções, exclusões e buscas. Sua capacidade de garantir o balanceamento parcial após operações complexas, mantendo a altura da árvore limitada a O(log n), é fundamental para sua eficácia em uma ampla gama de aplicações computacionais. Além disso, seu mecanismo de cores, com regras rigorosas, assegura que o desempenho das operações não se degrade, mesmo em casos extremos. Dada sua eficiência e versatilidade, a árvore rubro-negra permanece uma escolha popular em implementações de bases de dados, compiladores e outros sistemas críticos onde o gerenciamento otimizado de dados é essencial. Assim, sua relevância teórica e prática continua a ter um impacto significativo no campo da ciência da computação.

Link para Repositório do GitHub do projeto: github.com/Rezenddyr/Trabalho-APAL

8 Referência Bibliográfica

- 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.