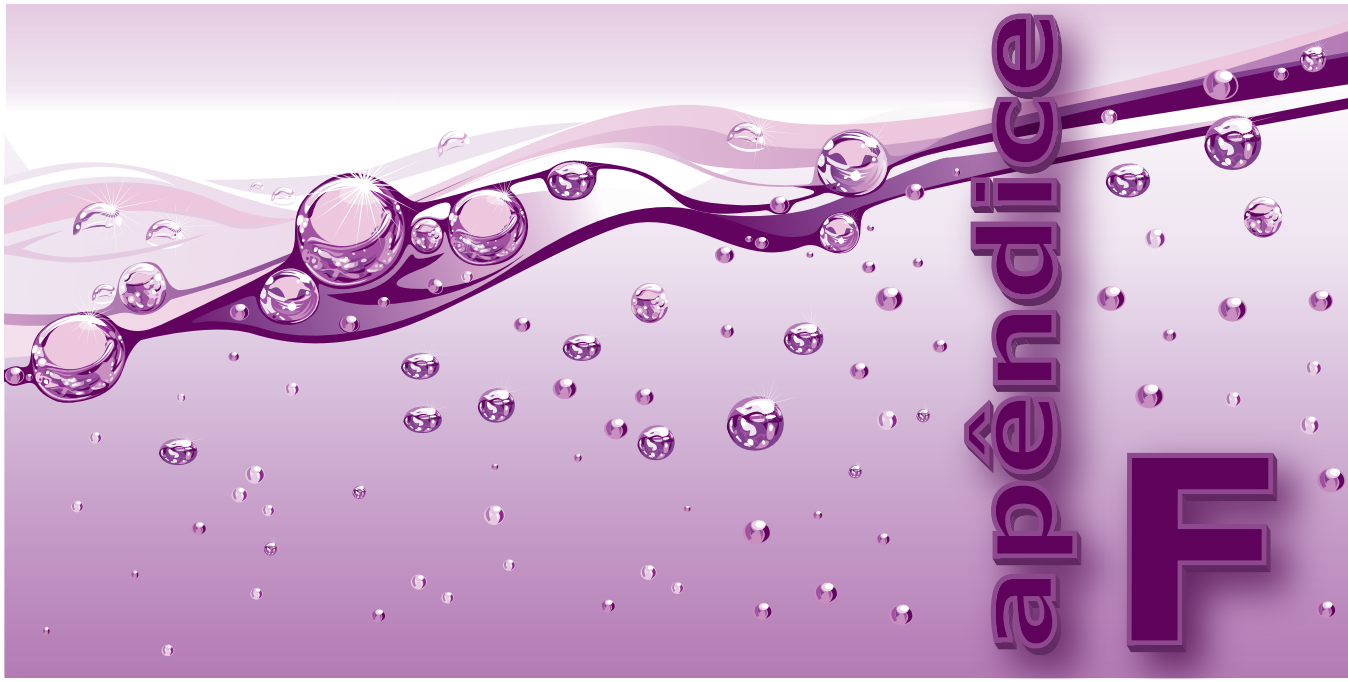


Árvores Rubro-negras

Conteúdo

RVORE RUBRO-NEGRA é um tipo de árvore balanceada, que é bem mais complicada para rebalancear do que árvores AVL. Portanto se você tem alguma dúvida referente a rotações e correções de balanceamento, sugere-se que você estude novamente **Seção 4.4** antes de prosseguir.

Não custa insistir que árvores rubro-negras são bastante complicadas de entender e implementar. Pior, se você insistir, ao final deste apêndice você irá se decepcionar, pois, em termos de análise assintótica (i.e., notação ó), essas árvores apresentam exatamente o mesmo custo temporal que árvores AVL para operações básicas de busca, inserção e remoção. A bem da verdade, árvores AVL são até um pouco mais eficientes. Enfim prossiga nesta seção apenas se você for um rubro-negro fanático...

F.1 Conceitos

Uma **árvore rubro-negra** é uma árvore binária de busca que obedece às seguintes regras:

[1] Cada nó é considerado vermelho ou preto.

[2] A raiz da árvore é sempre um nó preto.

[3] Qualquer nó nulo é preto.

[4] Se um nó for vermelho, seus filhos serão sempre pretos. Ou seja, um nó vermelho não pode ter qualquer filho vermelho.

[5] Qualquer caminho de um nó para 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 (i.e., quando a **Regra 4** é violada). Por sua vez, uma **altura preta** de um nó é o número de nós pretos encontrados desde o próprio nó até um nó nulo. De acordo com a **Regra 5**, todas as alturas pretas de um nó devem ser iguais. Uma **violação preta** ocorre quando um nó possui duas alturas pretas diferentes. A **altura preta de uma árvore rubro-negra** é a altura preta de sua raiz.

Assim como árvores AVL, árvores rubro-negras também são árvores autoajustáveis que apresentam balanceamentos aceitáveis. Mas os critérios de balanceamento de árvores rubro-negras são bem mais complexos e não são aparentes à primeira vista. Esses critérios são implícitos nas regras que definem essas árvores e funcionam de modo semelhante aos critérios que definem árvores AVL. Quer dizer, no caso de árvores AVL, um rebalanceamento pode ser necessário após uma operação de inserção ou remoção de nós se tal operação provocar um desbalanceamento da árvore. Por outro lado, quando um nó de uma árvore rubro-negra é inserido ou removido, as regras que definem esse tipo de árvore podem ser violadas, de modo que rotações e alterações de cor podem ser necessárias para corrigir eventuais violações. Do ponto de vista prático, o resultado obtido com árvores rubro-negras é semelhante àquele obtido com árvores AVL. Mas infelizmente, as semelhanças entre árvores rubro-negras e árvores AVL param por aqui porque árvores rubro-negras são muito mais difíceis de entender e implementar[1]. Portanto se você apresentar alguma dificuldade em balanceamento de árvores, sugere-se que você reveja a **Seção 4.4** antes de prosseguir.

Até aqui, as representações gráficas de árvores apresentadas neste livro não levaram em consideração nós nulos. Mas agora, para ser fiel à definição de árvore rubro-negra apresentada acima e facilitar seu entendimento, eles farão parte das representações gráficas de árvores rubro-negras. Nós pretos serão representados por círculos pretos e nós vermelhos serão representados por círculos acinzentados, como mostra a **Figura F–1**.

1 Árvores rubro-negras foram inventadas por Rudolf Bayer em 1972, mas a tarefa de colori-las coube a Leonidas Guibas e Robert Sedgewick em 1978 (v. **Bibliografia**). Segundo Sedgewick, a cor vermelha foi escolhida porque era a melhor cor reproduzida pela impressora a laser com a qual trabalhavam. Por sua vez, Guibas relata que vermelho e preto eram as cores das canetas que utilizavam enquanto desenhavam as famígeras árvores.

Figura F–1: Nós Usados para Representar Árvores Rubro-negras

A **Figura F–2** mostra três exemplos de árvores rubro-negras. Note que, como ilustra a **Figura F–2 (a)**, uma árvore rubro-negra pode conter apenas nós pretos. Verifique que as três árvores da **Figura F–2** satisfazem as cinco regras que definem árvores rubro-negras.

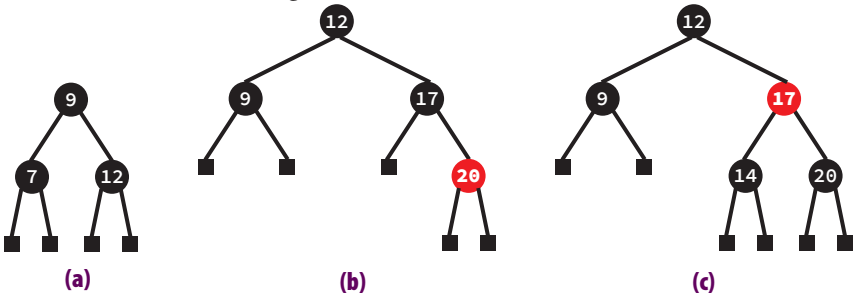


Figura F–2: Exemplos de Árvores Rubro-negras

A **Figura F–3** mostra três exemplos de árvores que não são consideradas rubro-negras por violarem algumas das regras que definem essas árvores.

A árvore da **Figura F–3 (a)** viola a **Regra 5**, pois o número de nós encontrados no caminho até um nó nulo seguindo o filho esquerdo do nó contendo a chave *9* é *2* (considerando o próprio nó *9* e o nó nulo encontrados nesse caminho). Por outro lado, seguindo o filho direito do nó que contém *9* até qualquer filho do nó contendo *12*, o número de nós pretos encontrados é *3*. Logo esse é um exemplo de violação preta. A árvore da **Figura F–3 (b)** viola a **Regra 4** porque o nó *17* é vermelho e, portanto, não poderia ter um filho vermelho. Logo esse é um exemplo de violação vermelha. A árvore da **Figura F–3 (c)** também contém uma violação preta (tente explicar por que).

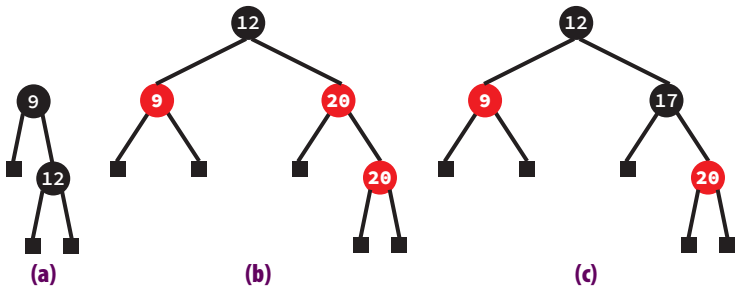


Figura F–3: Violações de Regras em Árvores Rubro-negras

As regras **3** e **4** de definição de árvores rubro-negras implicam que o menor caminho (i.e., aquele obtido alternando-se nós vermelhos e pretos) é apenas duas vezes maior do que o menor caminho obtido apenas com nós pretos. Portanto árvores rubro-negras são razoavelmente balanceadas.

F.2 Inserção

As propriedades de uma árvore rubro-negra são mantidas durante a inserção de um nó fazendo-se com que a cor do novo nó seja vermelha e girando-se ou alterando-se cores de alguns nós quando as regras forem violadas após a inserção. Inserindo-se sempre nós vermelhos, preservam-se as alturas pretas dos nós da árvore (**Regra 5**),

mas pode ocorrer uma violação vermelha. Se o novo nó fosse preto, seria sempre criado um caminho maior de nós pretos. Portanto o novo nó deve ser vermelho, mas assim procedendo, corre-se o risco de uma violação vermelha. Mesmo assim, essa abordagem é vantajosa por duas razões:

1. Existe a chance de não haver nenhuma violação.
2. Violação vermelha é mais fácil de resolver do que violação preta.

Se o pai de um novo nó for preto, não haverá violação. Do contrário, haverá violação vermelha e a árvore precisará ser reajustada. A **Regra 2** (violação preta) é violada apenas quando a árvore está vazia e o novo nó se torna raiz. A **Regra 4** (violação vermelha) é violada quando o pai do novo nó é vermelho.

Para determinar se ocorre violação vermelha após uma inserção, verifica-se se algum nó vermelho no caminho de inserção (v. **Seção 4.4.1**) possui algum filho vermelho. Note que, logo após uma inserção, apenas um filho de um nó vermelho poderá ser vermelho, pois, quando não, uma violação vermelha já teria ocorrido antes e a árvore em questão não seria uma árvore rubro-negra válida.

Após a inserção de um nó, existem três casos de violação vermelha que serão discutidos em seguida.

F.2.1 Caso 1 de Violação: Pai e Tio Vermelhos

Nesse caso, o pai e o tio do nó recém-inserido são examinados e, se eles forem vermelhos, o pai deles (avô do nó recém-inserido) deve ser preto [v. **Figura F–4 (a)**]. Assim pode-se apenas trocar as cores do avô e dos seus filhos de maneira que ele se torne vermelho e seus filhos se tornem pretos, como mostra a **Figura F–4 (b)**.

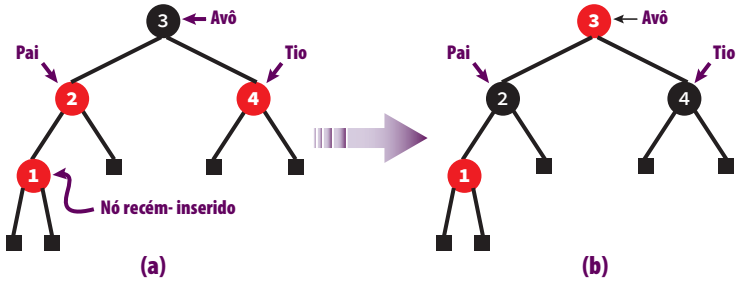


Figura F–4: Árvore Rubra-Negra: Caso 1 de Violação Devido a Inserção 1

Como a cor do referido avô passou a ser vermelha, pode ocorrer outra violação vermelha no próximo nível superior da árvore. Quer dizer, como a cor do avô passou a ser vermelha, é possível que seu pai (que não aparece na **Figura F–4**) também seja vermelho e então ocorre outra violação vermelha. Portanto após resolver este caso, continua-se percorrendo o caminho de inserção no sentido inverso (i.e., em direção à raiz) à procura de novas violações.

F.2.2 Caso 2 de Violação: Irmãos com Cores Diferentes e Pai Vermelho

A **Figura F–5** mostra a configuração de uma árvore rubro-negra após a inserção de um nó contendo *8* como chave. Nessa situação, uma mudança de cor precisa ser efetuada porque a configuração da subárvore cuja raiz é o nó *6* se enquadra no **Caso 1** descrito acima. Essa mudança de cor é mostrada na **Figura F–6**. Observe que tal mudança de cor causa uma violação vermelha envolvendo os nós com chaves *4* e *6*. Essa nova violação vermelha não se encaixa mais no **Caso 1**, de modo que uma mudança de cores não resolve o problema. Isso ocorre porque se o nó com a chave *4* se tornar preto, todos os caminhos que passam por esse nó até um nó nulo encontrarão um número de nós pretos maior do que nos caminhos que passam pelo nó *1*. Por exemplo, a altura preta do nó *2* seria *3* passando pelo nó *1* e *4* passando pelo nó *4 se esse último nó fosse preto*. A situação mostrada na **Figura F–6** corresponde exatamente ao **Caso 2** e, de acordo com o que foi exposto, alteração de cores não resolve esse caso. Ou seja, esse caso só pode ser resolvido por meio de rotação.

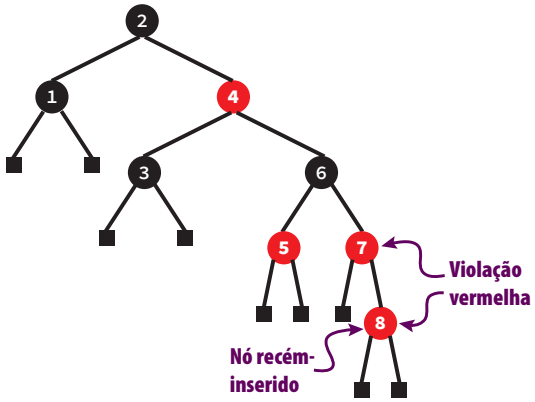


Figura F–5: Árvore Rubra-Negra: Caso 1 de Violação Devido a Inserção 2

A rotação utilizada para resolver o problema específico ilustrado na **Figura F–6** é uma rotação esquerda simples do nó contendo *4* em torno do nó que contém *2*, como se vê na **Figura F–7**. Note que, agora, não há nenhuma violação preta, pois cada caminho da raiz da subárvore em apreço, quer pela direita quer pela esquerda, encontra sempre três nós pretos.

Observe que, como mostra a **Figura F–7**, quando ocorre a rotação esquerda em torno do nó contendo *2*, o nó que contém *4* torna-se a nova raiz da árvore e o nó contendo *2* se torna seu filho vermelho. Note que, agora, não há mais nenhuma violação vermelha e que todas as alturas pretas são as iguais. Esse exemplo mostra ainda por que uma função de rotação simples deve também alterar cores do modo como foi feito nesse exemplo (v. **Seção F.4**).

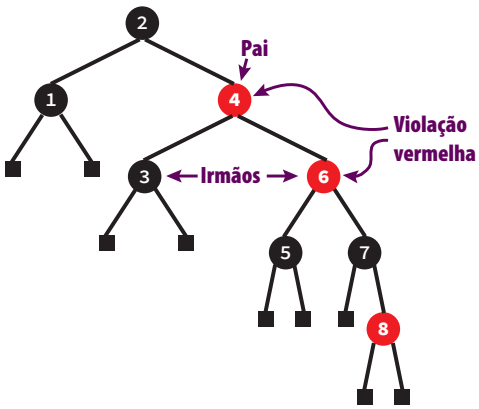


Figura F–6: Árvore Rubra-Negra: Caso 2 de Violação Devido a Inserção 1

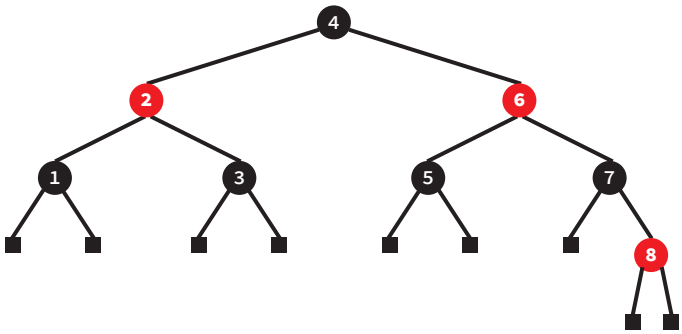


Figura F–7: Árvore Rubra-Negra: Caso 2 de Violação Devido a Inserção 2

F.2.3 Caso 3 de Violação: Irmãos com Cores Diferentes e Pai Preto

O último caso de violação ocorre quando o pai dos irmãos discutidos no **Caso 2** é preto, como mostra a **Figura F–8**. Nesse caso, uma rotação simples não resolve o problema. É nesse caso que a rotação dupla, que será descrita na **Seção F.4**, se faz necessária para produzir o resultado mostrado na **Figura F–10**. A **Figura F–9** mostra o resultado após a primeira dessas rotações que compõem a rotação dupla mencionada.

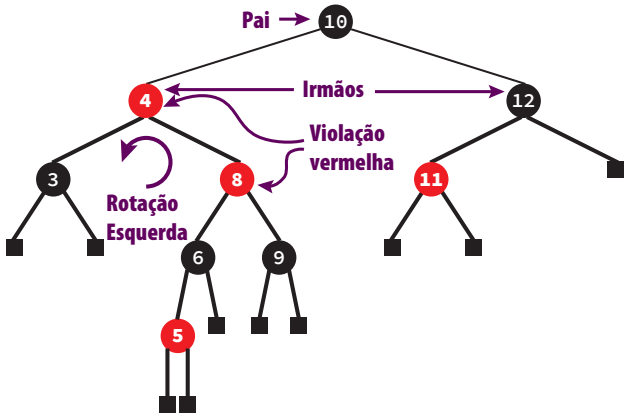


FIGURA F–8: ÁRVORE RUBRA-NEGRA: CASO 3 DE VIOLAÇÃO DEVIDO A INSERÇÃO 1

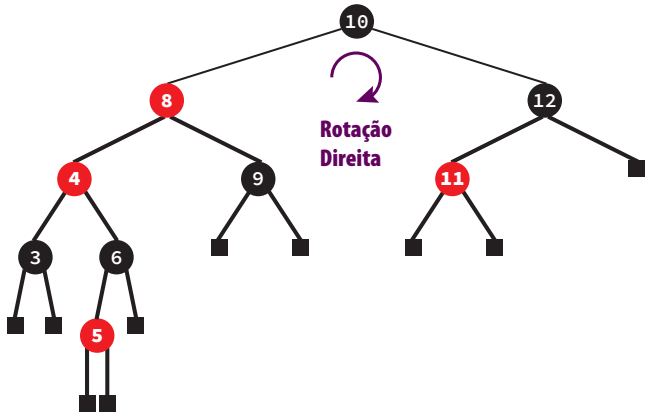


FIGURA F–9: ÁRVORE RUBRA-NEGRA: CASO 3 DE VIOLAÇÃO DEVIDO A INSERÇÃO 2

Agora, por que esse caso requer uma rotação dupla e não apenas uma rotação simples? Para responder essa questão, note que a primeira rotação (v. **Figura F–9**) aumenta a altura preta da subárvore esquerda do nó contendo *4*. Isso poderia introduzir uma violação preta na árvore. Assim a rotação dupla é utilizada para evitar que essa violação ocorra.

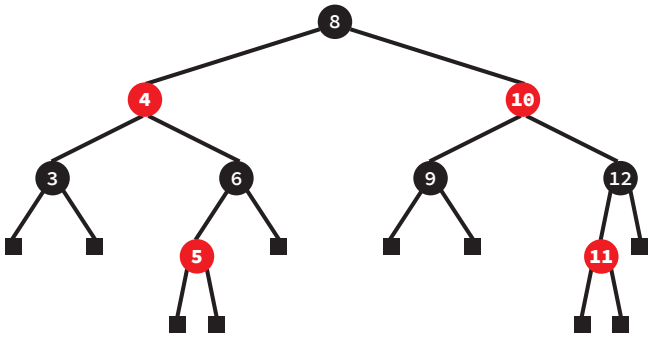


FIGURA F–10: ÁRVORE RUBRA-NEGRA: CASO 3 DE VIOLAÇÃO DEVIDO A INSERÇÃO 3

F.2.4 Simetria dos Casos de Violação

Cada caso descrito acima representa, de fato, dois casos combinados de violação, de modo que, após a inserção de um nó, existem seis (e não apenas três) possibilidades de violação. A **Figura F–11** mostra a simetria do **Caso 1** de violação descrito acima. Por sua vez, a **Figura F–12** mostra a simetria do **Caso 2** e a **Figura F–13** ilustra o **Caso 3** 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. Isso tornaria a codificação bastante longa, o que justifica a escolha da implementação a ser desenvolvida aqui (v. **Seção F.4**).

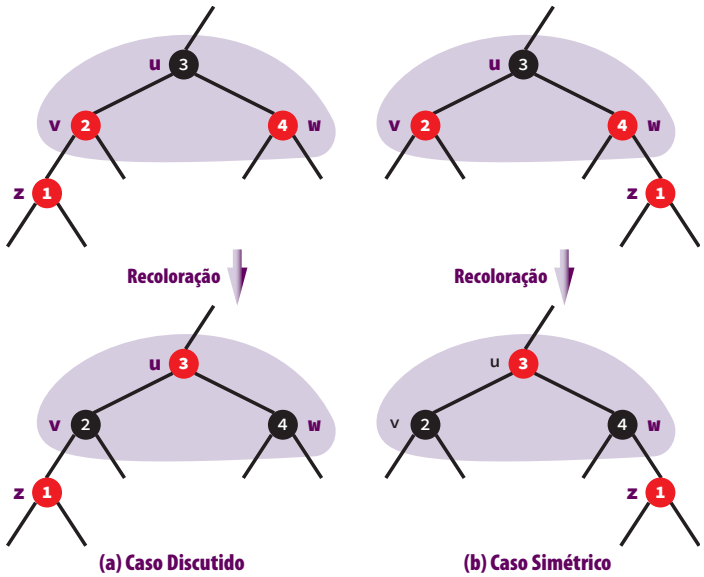


FIGURA F–11: SIMETRIA DO CASO 1 DE VIOLAÇÃO DEVIDO A INSERÇÃO

Inserção em árvores rubro-negras é uma operação *relativamente* fácil, se comparada com remoção nessas árvores. Violação vermelha é razoavelmente simples de testar e corrigir sem causar problemas em níveis superiores da árvore. Agora, prepare-se, vem aí a remoção em árvores rubro-negras...

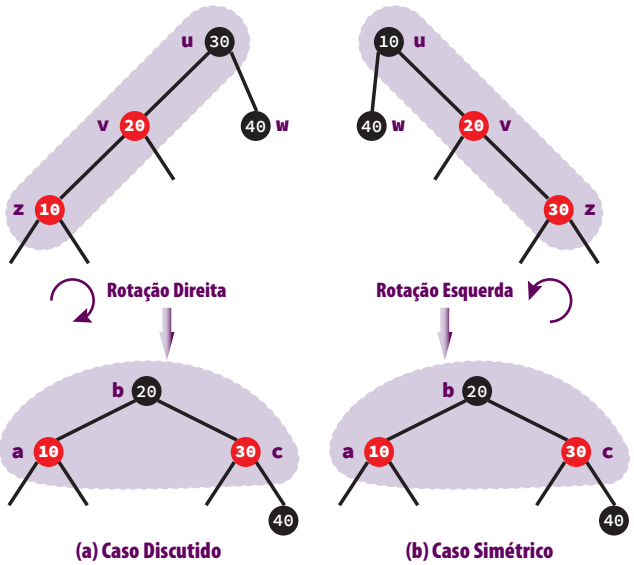


FIGURA F–12: SIMETRIA DO CASO 2 DE VIOLAÇÃO DEVIDO A INSERÇÃO

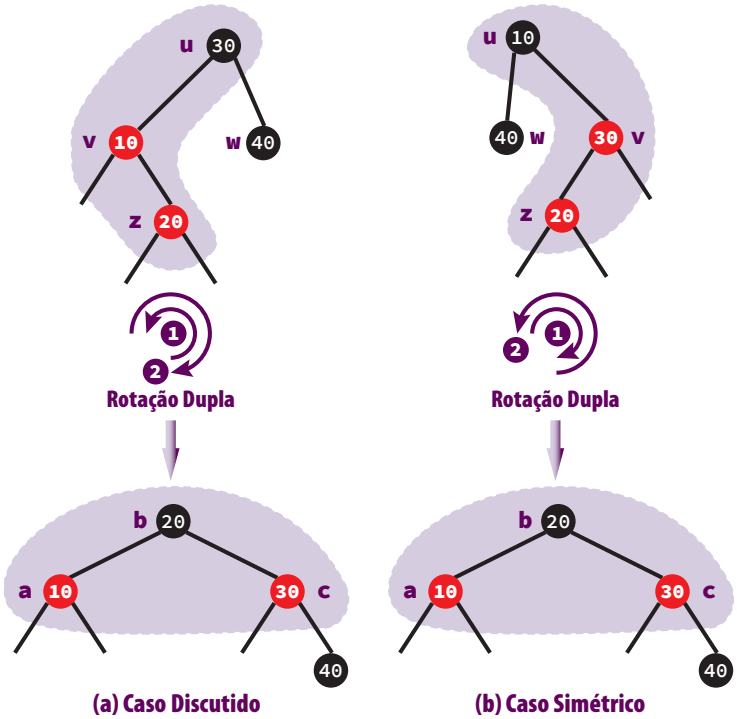


Figura F–13: Simetria do Caso 3 de Violação Devido a Inserção

F.3 Remoção

Durante uma operação de inserção, pode-se selecionar a cor de um novo nó. Nesse caso, uma violação vermelha é forçada e, então, corrigida. Tais violações vermelhas são relativamente fáceis de corrigir. Agora, quando se remove um nó, não se tem a opção de escolher sua cor. 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. Em outras palavras, como nenhum nó vermelho é levado em conta na altura preta de um nó, a remoção de um nó vermelho não tem consequência na altura preta de qualquer nó. Portanto a remoção de um nó vermelho não pode violar as regras de uma árvore rubro-negra.

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.

Como foi visto na **Seção 4.1.1**, se o nó a ser removido tem apenas um filho não nulo, ele pode ser substituído por esse filho. Por outro lado, se o nó a ser removido não possui nenhum filho não nulo, ele pode ser substituído por um nó nulo.

F.3.1 Caso 1: Nó Removido Preto com Filho Vermelho

Se o nó que será removido é preto e tem um filho vermelho, pode-se colorir esse filho com preto sem introduzir nenhuma violação, uma vez que o caso de remoção de um nó preto foi reduzido ao caso de remoção do nó vermelho, que não causa nenhuma violação. Esse caso trivial de reajuste, denominado **Caso 1**, é mostrado na **Figura F–14**.

Se o nó a ser removido possui dois filhos, encontra-se seu sucessor imediato e copia-se seu conteúdo para o nó a ser removido. Então usando o procedimento descrito acima, remove-se o referido sucessor, pois sabe-se que ele tem no máximo um filho (v. **Seção 4.1.1**).

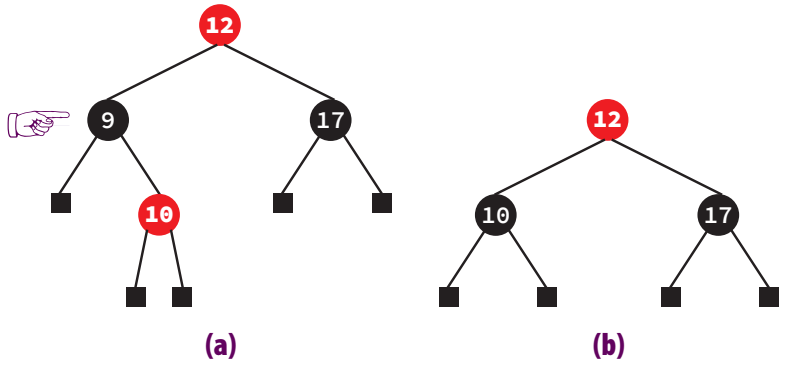


FIGURA F–14: CASO 1 DE REAJUSTE APÓS REMOÇÃO EM ÁRVORE RUBRO-NEGRA

F.3.2 Caso 2: Irmão Preto e Sobrinho Preto

Se um nó é removido, o irmão desse nó é preto e seus filhos são ambos pretos, tem-se um caso relativamente trivial de resolver, mas que pode se propagar para cima na árvore. Esse caso é ilustrado na **Figura F–15**, na qual o nó tracejado é aquele que foi removido.

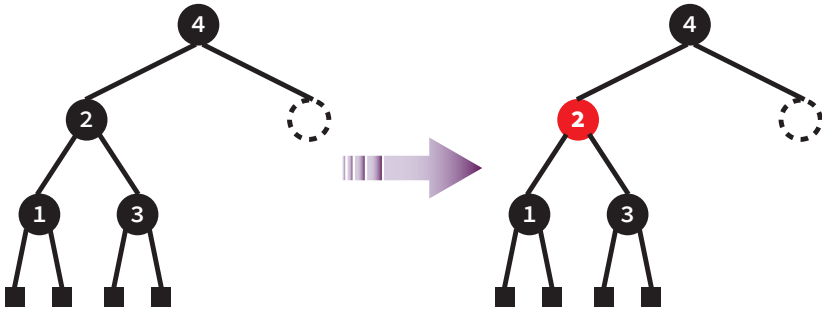


FIGURA F–15: REMOÇÃO EM ÁRVORE RUBRO-NEGRA 1

Nesse caso, pode-se simplesmente recolorir o irmão do nó removido (i.e., o nó contendo *2*) com vermelho sem introduzir uma violação vermelha, mas ainda existe a possibilidade de que ocorra uma violação preta porque a altura preta da subárvore esquerda do nó com conteúdo *4* foi diminuída. Isso faz com que ambas as subárvores do nó contendo *4* tenham a mesma altura preta, mas se o nó com conteúdo *4* for raiz de uma subárvore, então seu pai apresentará uma violação preta visto que a subárvore do irmão do nó com conteúdo *4* não teve sua altura preta alterada.

Agora, se o pai do nó removido (o nó contendo *4* nesse caso) era originalmente vermelho, pode-se recolorir o irmão com vermelho, o pai com preto e o serviço estará completo porque as alturas pretas estarão normalizadas até a raiz da árvore (**Exercício**: explique por que). A **Figura F–16** mostra essa última situação.

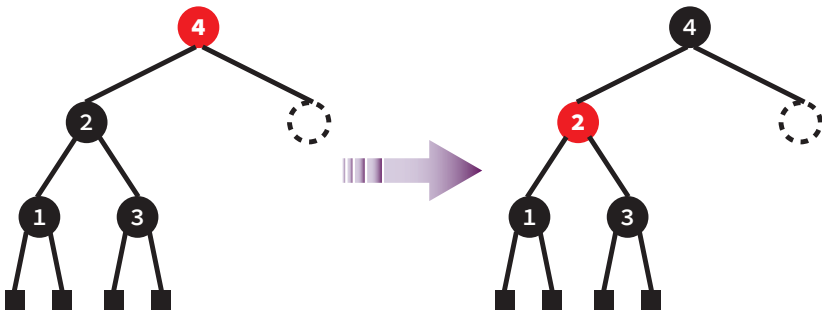


FIGURA F–16: REMOÇÃO EM ÁRVORE RUBRO-NEGRA 2

F.3.3 Caso 3: Irmão Preto e Sobrinho(s) Vermelho(s)

Os casos mais difíceis de correção de violação após remoção são aqueles em que o irmão do nó removido é preto e algum ou seus dois filhos são vermelhos. Aqui, existem dois subcasos distintos. Se o filho da esquerda do irmão for vermelho, então só é necessário executar uma rotação simples. Entretanto o pai poderia ser vermelho ou preto. Assim é preciso guardar sua cor para restaurá-la depois, porque a rotação não considera as cores antigas do novo e do antigo pai. Depois da rotação, o novo pai é recolorido com sua cor original e seus filhos se tornam pretos, como mostra a **Figura F–17**. Nessa figura, o nó sem coloração significa que ele pode ser de qualquer cor (i. e, sua cor não importa nesta discussão).

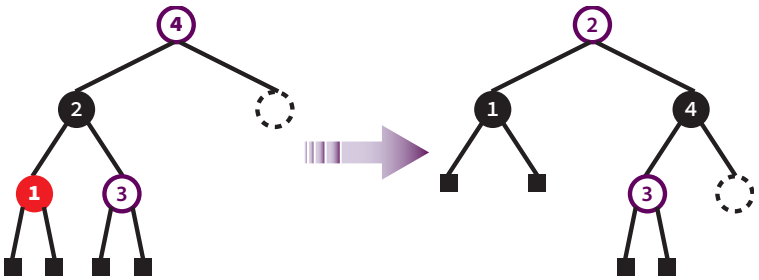


FIGURA F–17: REMOÇÃO EM ÁRVORES RUBRO-NEGRAS 3

A última alteração na configuração da subárvore restaura sua altura preta da árvore e nenhum ajuste na árvore será mais necessário. O mesmo processo é seguido quando o filho da direita do irmão do nó removido é vermelho, exceto que, desta vez, usa-se uma rotação dupla em vez de uma rotação simples. As cores finais são idênticas, e lembre-se que o filho da esquerda do irmão seria preto nesse caso. Se ele for vermelho, então a resolução do **Caso 2** é aplicada (**Figura F–18**).

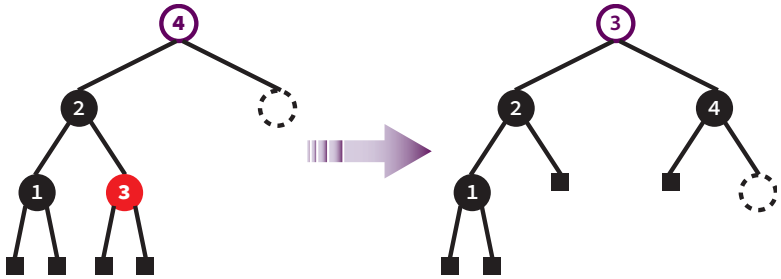


FIGURA F–18: REMOÇÃO EM ÁRVORE RUBRO-NEGRA 4

Esses são os únicos casos nos quais o irmão do nó removido é preto. Se esse irmão for vermelho, a situação é um pouco pior.

F.3.4 Caso 4: O Irmão do Nó Removido É Vermelho

Cada caso em que o nó removido possui um irmão vermelho requer pelo menos uma rotação. Se esse irmão for vermelho, então ambos os seus filhos serão pretos. Assim pode-se realizar uma rotação simples, recolorir o novo pai com preto e recolorir o filho da direita do irmão com vermelho para restaurar a árvore, como mostra a **Figura F–19**.

Examinando-se a **Figura F–19**, vê-se como o procedimento prescrito realmente restaura o balanceamento corrigindo as alturas pretas. Nesse caso, o nó contendo *3* se torna vermelho porque ele é o único nó que se tem certeza que pode se tornar vermelho sem propagar violações.

O segundo caso de irmão vermelho considera o filho não nulo do irmão. Como esse filho não é um nó nulo, pode-se simplesmente testar um de seus filhos e agir de acordo com qual deles é vermelho. Se o outro filho for

vermelho, executa-se uma rotação dupla, colore-se o novo pai com preto, seu filho da direita com preto e seu filho da esquerda com vermelho, como mostra a **Figura F–20**. Isso restaura a altura preta da subárvore direita sem alterar a altura preta da subárvore esquerda.

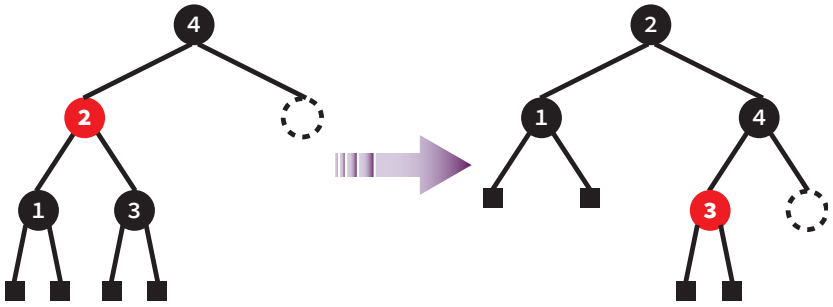


Figura F–19: Remoção em Árvore Rubro-negra 5

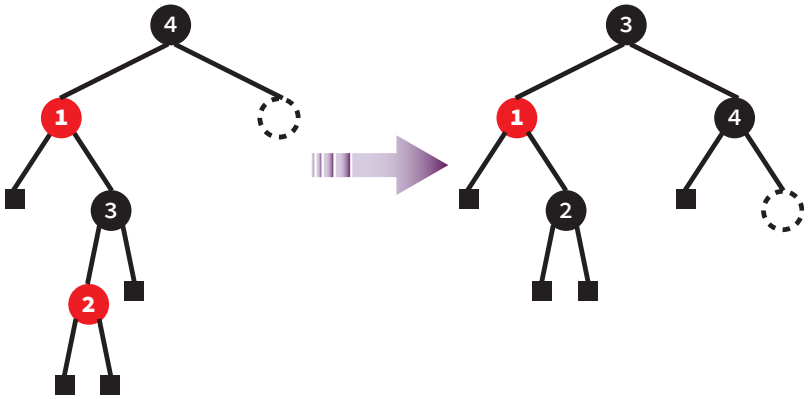


Figura F–20: Remoção em Árvore Rubro-negra 6

Quando há dois nós vermelhos para lidar, torna-se um deles preto após a rotação e deixa-se de lado o outro para evitar violação da altura preta da subárvore da qual ele foi removido.

O último caso de violação ocorre quando o filho da direita do nó contendo *4* é vermelho. Esse caso pode ser reduzido ao caso anterior por meio de uma rotação simples do nó contendo *3* em torno do nó contendo *2* seguida de uma rotação dupla em torno do nó contendo *6* (o caso anterior), o que resulta na estrutura desejada, com as mesmas cores acima (v. **Figura F–21**).

F.3.5 Violações Devido a Inserções vs. Violações Devido a Remoções

A **Tabela F–1** compara as operações de inserção e remoção em árvores rubro-negras em termos de restruturação após cada uma delas.

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

Tabela F–1: Inserção versus Remoção em Árvore Rubro-negra

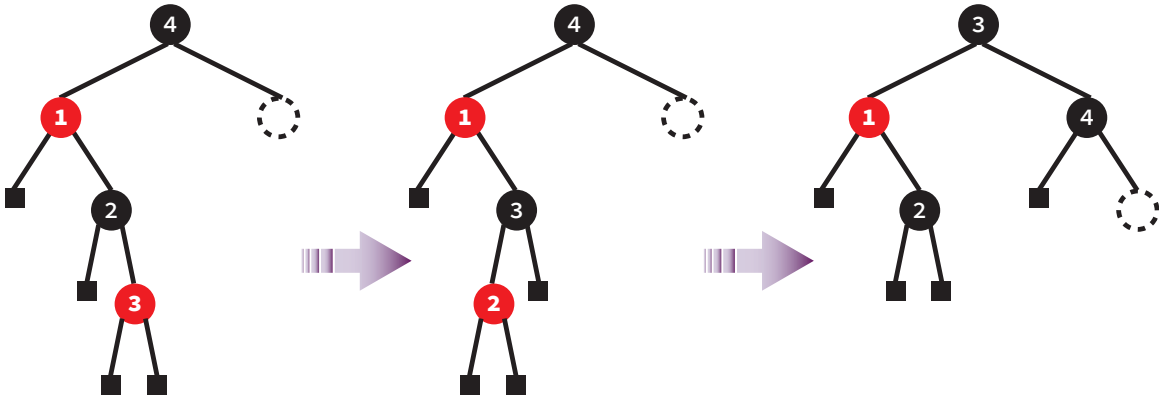


FIGURA F–21: REMOÇÃO EM ÁRVORE RUBRO-NEGRA 7

F.4 Implementação

F.4.1 Tipos

As seguintes definições de tipos serão utilizadas nesta implementação de árvores rubro-negras:

```
typedef enum {ESQUERDA = 0, DIREITA = 1} tDirecao;

typedef enum {PRETO = 0, VERMELHO = 1} tCor;

typedef struct noRN {
           tCor         cor; /* Indica se a cor do nó */
           tConteudo    conteudo; /* Conteúdo do nó */
           struct noRN *filho[2]; /* Os filho do nó */
        } tNoRN;

typedef struct {
           tNoRN *raiz; /* Raiz da árvore */
           int    nNos; /* Número de nós da árvore */
        } tArvoreRN;
```

A seguinte função verifica se o nó cujo endereço é recebido como parâmetro é vermelho:

```
static int EhVermelho(tNoRN *pNo)
{
   return pNo && pNo->cor == VERMELHO;
}
```

Não é necessário escrever uma função similar à função `EhVermelho()` para verificar se um nó é preto, pois, se um nó não é vermelho, ele é preto e vice-versa.

Observação: Para manter o código mais simples de entender, a implementação a apresentada nesta seção não faz tratamento de exceções com `ASSEGURA`, como tem sido feito anteriormente e como deveria ser feito aqui também. Evidentemente, na prática, você não deve proceder assim.

F.4.2 Rotações

Rotações em árvores rubro-negras são similares a rotações em árvores AVL, mas, aqui, se aproveita a operação de rotação para efetuar alterações de cores nos nós que são protagonistas. Revisando a **Seção F.2**, você entenderá a razão pela qual os nós são coloridos do modo que são após sofrerem rotações. Além disso, as quatro rotações utilizadas para restaurar árvores rubro-negras são condensadas em apenas duas rotações para simplificar a codificação das operações de restauração da árvore que serão apresentadas mais adiante.

Se você observar atentamente as funções `RotacaoDireita()` e `RotacaoEsquerda()` apresentadas na **Seção 4.2**, constatará que elas são simétricas. Quer dizer, cada uma delas pode ser obtida a partir da outra trocando-se *esquerda* por *direita* e vice-versa. Assim essas duas funções podem ser resumidas numa única função com a introdução de um parâmetro adicional que indica a direção da rotação (i.e., se ela é à direta ou à esquerda).

A função `RotacaoSimples()` efetua uma rotação simples sobre um nó, representado pelo parâmetro `x`, de uma árvore rubro-negra na direção especificada pelo parâmetro `dir`. Essa função retorna a nova raiz da árvore após a rotação. Note que, como foi dito acima, assume-se que todos os nós são válidos (i.e., não há tratamento de exceção).

```
static tNoRN *RotacaoSimples(tNoRN *x, tDirecao dir)
{
   tNoRN *y; /* Nova raiz após a rotação */

      /* Faz y apontar para o filho da raiz que está do lado oposto da rotação */
   y = x->filho[!dir];

      /* O novo filho de x que estará do lado oposto da */
      /* rotação será o filho de y do lado da rotação   */
   x->filho[!dir] = y->filho[dir];

      /* O filho da nova raiz do lado da rotação será a antiga raiz */
   y->filho[dir] = x;

   y->cor = PRETO; /* A nova raiz será preta */

      /* A antiga raiz será vermelha para não causar violação preta da nova raiz */
   x->cor = VERMELHO;

   return y;
}
```

A **Figura F–22** ilustra duas chamadas da função `RotacaoSimples()`.

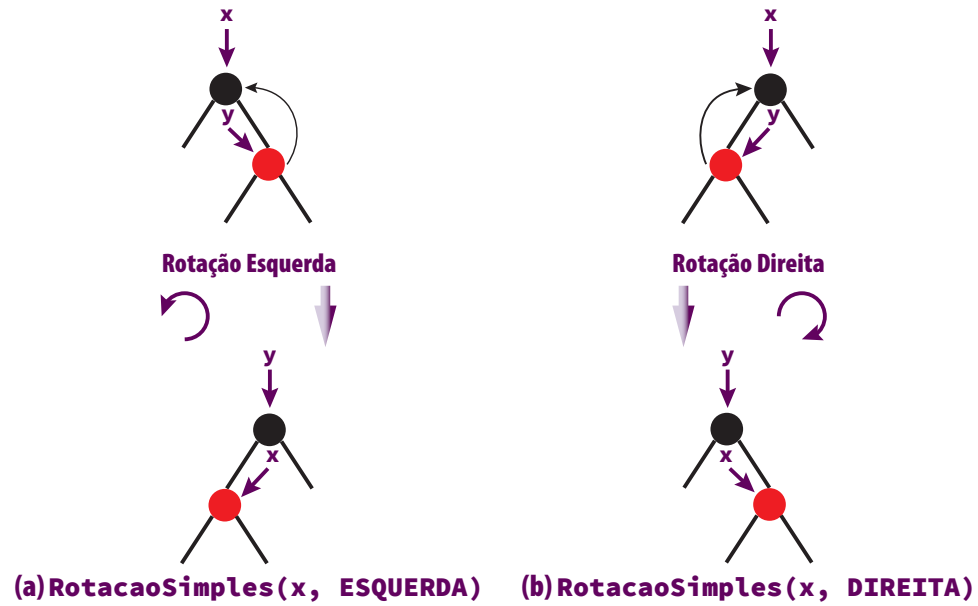


FIGURA F–22: ROTAÇÕES SIMPLES EM ÁRVORES RUBRO-NEGRAS

As rotações duplas de interesse em restauração de árvores rubro-negras envolvem três nós denominados *avô*, *pai* e *filho* devido aos graus de parentesco que mantêm entre si de acordo com a terminologia de árvores. Conforme o nome da rotação sugere, numa rotação dupla ocorrem duas rotações: a primeira rotação é do neto oposto da

rotação em torno de seu pai e a segunda rotação ocorre entre esse neto e seu avô na direção especificada pelo segundo parâmetro. A seguinte função implementa essa rotação dupla.

```
static tNoRN *RotacaoDupla(tNoRN *x, tDirecao dir)
{
      /* Efetua uma rotação em torno do filho oposto de x */
      /* na direção contrária especificada por 'dir'      */
   x->filho[!dir] = RotacaoSimples(x->filho[!dir], !dir);

      /* Agora gira o filho */
   return RotacaoSimples(x, dir);
}
```

A **Figura F–23** ilustra duas chamadas da função `RotacaoDupla()`.

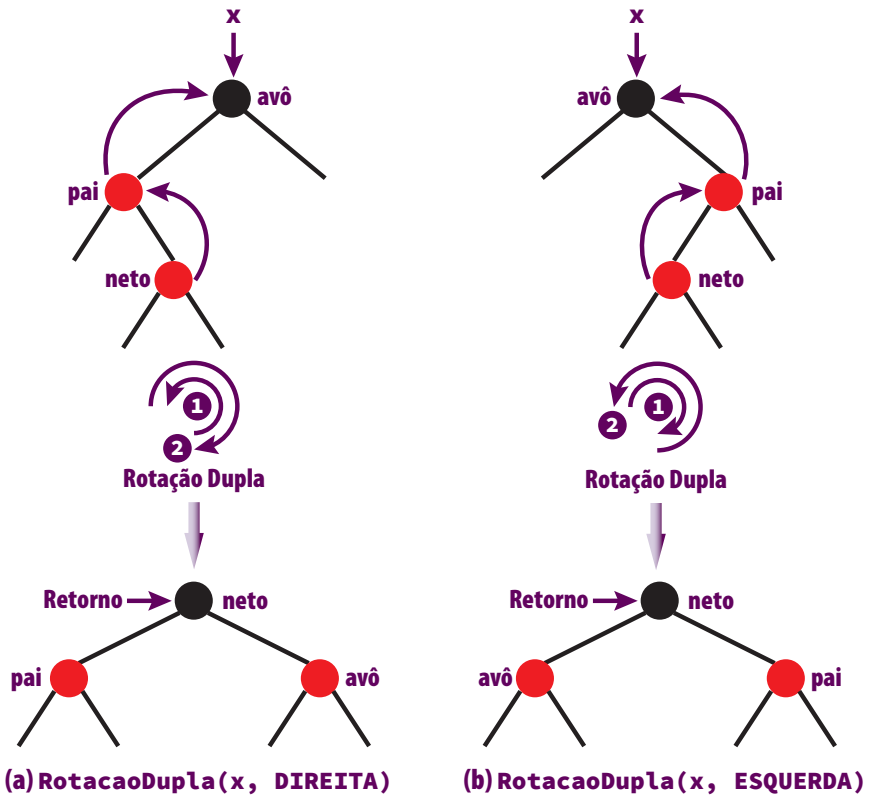


FIGURA F–23: ROTAÇÕES DUPLAS EM ÁRVORES RUBRO-NEGRAS

F.4.3 Inserção

Antes de implementar a função que realiza inserção e lida com violações, lembre-se que um nó recém-inserido nunca é rebalanceado. A função apenas retorna o novo nó quando ele chega a uma folha e deixa o rebalanceamento para os nós acima. Essa função certifica-se que, se existe uma violação, ela está na subárvore. Desse modo, podem-se efetuar as rotações sem usar um indicador que sinaliza para o algoritmo que ele deve continuar efetuando rotações para cima. Na realidade, nenhum indicador é necessário, pois é possível fazer mudanças de cores no caminho de subida da árvore mesmo que apenas uma rotação simples ou dupla possa ser efetuada.

A função, a ser apresentada adiante, que implementa inserção em árvores rubro-negras usa recursão para descer na árvore até encontrar a folha na qual será efetuada a inserção. Na fase de decréscimo da função, os ponteiros usados no caminho de inserção (v. **Seção 4.4.1**) podem ser acessados e usados para corrigir eventuais violações.

A função `InsereRN()` apresentada a seguir serve como interface para a função `InsereRNRec()` que, de fato, efetua a maior parte da tarefa de inserção de um nó numa árvore rubro-negra com os eventuais ajustes que sejam necessários na árvore. Os parâmetros da função `InsereRN()` representam a árvore rubro-negra na qual será feita a inserção e o conteúdo do nó que será inserido. Essa função retorna `1`, se a inserção for bem-sucedida ou `0`, em caso contrário.

```
int InsereRN(tArvoreRN *arvore, const tConteudo *conteudo)
{
   tNoRN *p;

      /* De fato, quem faz a inserção é a função InsereRNRec()*/
   p = InsereRNRec(arvore->raiz, conteudo);

      /* Altera a raiz da árvore apenas se ocorreu inserção */
   if (p) {
         /* Altera a raiz da árvore para refletir a inserção */
      arvore->raiz = p;

      arvore->raiz->cor = PRETO; /* Assegura que a raiz é preta */

      ++arvore->nNos; /* Mais um nó foi inserido */
   }

   return p ? 1 : 0;
}
```

A função `InsereRNRec()` que efetivamente efetua a inserção de nós e eventuais correções em árvores rubro-negras é definida a seguir. Essa função tem como parâmetros a raiz da árvore rubro-negra na qual será feita a inserção e o conteúdo do nó que será inserido. Ela retorna o endereço da raiz da árvore após a inserção, se a inserção for bem-sucedida ou **NULL**, em caso contrário. A inserção é malsucedida quando a chave do novo nó já existe na árvore, pois a chave é considerada primária.

```
static tNoRN *InsereRNRec(tNoRN *raiz, const tConteudo *conteudo )
{
   tDirecao dir; /* Direção (esquerda/direita) de descida */
   int      compara; /* Resultado de comparação de chaves */

      /* Se a árvore estiver vazia, cria uma raiz e retorna-a */
   if (!raiz)
      return NovoNoRN(conteudo);

      /* Compara a chave a ser inserida com a chave que se encontra na raiz */
   compara = strcmp(raiz->conteudo.chave, conteudo->chave);

      /* Se a chave já existir, ela não será inserida, pois ela é primária */
   if (!compara)
      return NULL; /* Chave já existe */

      /*                                                              */
      /* É preciso descer na árvore para encontrar o local da inserção */
      /*                                                              */

      /* 'dir' indicará se a descida será pela esquerda */
      /* (dir = 0) ou pela direita (dir = 1)            */
   dir = (tDirecao) (compara < 0);

      /* Chamada recursiva para inserção à direita ou esquerda de uma folha */
   raiz->filho[dir] = InsereRNRec(raiz->filho[dir], conteudo);

      /* Se for necessário corrigir alguma violação,  */
      /* a correção ocorrerá na instrução if a seguir */
```

```
      /* Verifica se o nó inserido ainda é vermelho */
   if (EhVermelho(raiz->filho[dir])) {
         /* Verifica se o irmão do nó inserido é vermelho */
      if (EhVermelho(raiz->filho[!dir])) {
            /* Caso 1 de violação. Esse caso é corrigido por */
            /* meio de troca de cores entre pai e filhos     */
         raiz->cor = VERMELHO; /* Pai se torna comunista */

            /* Filhos se tornam pretos */
         raiz->filho[ESQUERDA]->cor = PRETO;
         raiz->filho[DIREITA]->cor = PRETO;
      } else {
            /* Casos 2 e 3 de violação. Aqui, apenas rotações resolvem. */
         if (EhVermelho(raiz->filho[dir]->filho[dir])) {
               /* O filho esquerdo do filho esquerdo ou o filho direito do filho */
               /* direito é vermelho.  Uma simples rotação na direção contrária  */
               /* resolve essa violação.                                         */
            raiz = RotacaoSimples(raiz, !dir);
         } else if (EhVermelho(raiz->filho[dir]->filho[!dir])) {
               /* O filho direito do filho esquerdo ou o filho esquerdo do */
               /* filho direito é vermelho. Uma rotação dupla na direção   */
               /* contrária resolve essa violação.                         */
            raiz = RotacaoDupla(raiz, !dir);
         }
      }
   }

   return raiz;
}
```

É recomendável que você explore um exemplo que considere todos os casos de violação para entender como a função `InsereRNRec()` funciona. Assim sugere-se que use um número razoável de chaves escolhidas ao acaso e execute manualmente a referida função.

F.4.4 Remoção

A função `RemoveRN()` remove de uma árvore rubro-negra um nó que contém a chave especificada como parâmetro. Os parâmetros dessa função representam a árvore rubro-negra na qual será feita a remoção e a chave do nó que será removido. Essa função retorna `1`, se a remoção for bem-sucedida ou `0`, de outro modo. Note que a raiz é colorida com preto ao final se a árvore não for vazia. Observe ainda que essa função usa duas variáveis indicadoras que são passadas como parâmetros para a função `RemoveRNRec()` (v. adiante):

- `concluido` — essa variável indicará se será necessário checar possíveis violações (`concluido` igual a `0`) ou não (`concluido` igual a `1`). Em princípio, supõe-se que há violações. As funções `RemoveRNRec()` e `CorrigeRemocao()` (v. adiante) confirmarão ou não essa suposição. Esse indicador é necessário para informar à função que reajusta a árvore rubro-negra após uma remoção quando ela deve encerrar os reajustes na árvore.
- `houveRemocao` — essa variável indicará se houve de fato remoção após uma chamada da função `RemoveRNRec()`.

```
int RemoveRN(tArvoreRN *arvore, tChave chave)
{
   int concluido = 0, /* Indicará se será necessário checar possíveis violações */
                      /* (concluido = 0) ou não (concluido = 1). Em princípio,  */
                      /* supõe-se que sim. RemoveRNRec() confirmará ou não      */
                      /* essa suposição                                         */
```

```
      houveRemocao = 0; /* Indicará se houve remoção */

      /* Chama a função que realmente efetua a remoção */
   arvore->raiz = RemoveRNRec( arvore->raiz, chave, &concluido, &houveRemocao );

      /* A raiz da árvore deve ser preta */
   if (arvore->raiz)
      arvore->raiz->cor = PRETO;

   return houveRemocao;
}
```

A função recursiva `RemoveRNRec()`, que efetivamente realiza a remoção de nós e eventuais correções em árvores rubro-negras, é definida a seguir. Ela retorna o endereço da raiz da árvore após a remoção, se a remoção for bem-sucedida ou **NULL**, se a árvore tornar-se vazia. Os parâmetros dessa função são:

- `raiz` (entrada/saída) — ponteiro para a raiz da árvore na qual será feita a remoção; é importante observar que esse parâmetro pode ser alterado
- `chave` (entrada/saída) — chave do nó a ser removido
- `concluido` (entrada/saída) v indica se a árvore precisará ser reajustada
- `removido` (entrada/saída) — indica quando ocorre remoção

```
static tNoRN *RemoveRNRec(tNoRN *raiz, tChave chave, int *concluido, int *removido)
{
   tNoRN   *sucessor, /* Apontará para o sucessor de um nó */
           *novaRaiz; /* Um ponteiro auxiliar */
   tDirecao dir;  /* Direção (esquerda/direita) de descida */
   int      compara; /* Resultado de comparação de chaves */

   if (!raiz) { /* Árvore vazia */
      *concluido = 1; /* Não há mais o que fazer */
   } else {
         /* Compara a chave do nó corrente com a chave do nó a ser removido */
      compara = strcmp(raiz->conteudo.chave, chave);

         /* Verifica se a chave foi encontrada */
      if (!compara) { /* Chave encontrada */
         if (!raiz->filho[ESQUERDA] || !raiz->filho[DIREITA]) {
            /*                                                    */
            /* Caso trivial: O nó não tem filho ou tem apenas um filho */
            /*                                                    */

               /* Faz 'novaRaiz' apontar para o filho direito da raiz se o     */
               /* filho esquerdo for NULL. Se o filho esquerdo da raiz não for */
               /* NULL, 'novaRaiz' apontará para ele. Se os 2 filhos forem     */
               /* NULL, 'novaRaiz' também será NULL.                           */
            novaRaiz = raiz->filho[!raiz->filho[ESQUERDA]];

               /* Se a nova raiz for vermelha, torna-a preta */
            if (EhVermelho(novaRaiz)) {
                  /* Lembre-se que nós nulos são pretos. Então não há */
                  /* possibilidade de corrupção de memória aqui.      */
               novaRaiz->cor = PRETO;
            }

            free(raiz); /* Libera a antiga raiz */

            *concluido = 1; /* Não há mais nenhuma correção a ser feita */

            return novaRaiz; /* Retorna a nova raiz */
         } else { /* O nó tem dois filhos */
```

```
            /*** Encontra o sucessor do nó ***/

               /* Primeiro, desce-se até o filho direito */
            sucessor = raiz->filho[DIREITA];

               /* Agora, desce-se sempre à esquerda até encontrar uma folha */
            while (sucessor->filho[ESQUERDA])
               sucessor = sucessor->filho[ESQUERDA];

            /* Sucessor encontrado */

               /* Copia o conteúdo do sucessor para o nó que seria removido */
            raiz->conteudo = sucessor->conteudo;

               /* Copia a chave do sucessor para a chave do nó a ser removido */
            strcpy(chave, sucessor->conteudo.chave);
         }

         *removido = 1; /* Houve remoção */
      }
      /*                                                    */
      /* A chave ainda não foi encontrada e a busca continua */
      /*                                                    */

         /* Obtém a direção da descida (0 = esquerda; 1 = direita) */
      dir = compara < 0;

         /* Efetua a remoção na subárvore indicada por 'dir' */
      raiz->filho[dir] = RemoveRNRec(raiz->filho[dir], chave, concluido, removido);

         /* Efetua eventuais correções de violações  */
         /* se o parâmetro 'concluido' assim indicar */
      if (!*concluido)
         raiz = CorrigeRemocao(raiz, dir, concluido);
   }

   return raiz;
}
```

Como a função `RemoveRNRec()` remove apenas nós sem dois filhos, deve-se alterar o indicador representado pelo parâmetro `concluido` apenas depois de uma remoção de nó vermelho ou se o **Caso 1** for aplicável. De outro modo, esse parâmetro será alterado pela função `CorrigeRemocao()` (v. adiante).

A função `CorrigeRemocao()`, que é chamada por `RemoveRNRec()`, lida com todos os casos de violação que possam surgir após a remoção de um nó preto de uma árvore rubro-negra. A função `CorrigeRemocao()`, apresentada a seguir, corrige eventuais violações ocorridas numa remoção de nó de uma árvore rubro-negra. Os parâmetros dessa função são:

- `raiz` (entrada) — ponteiro para a raiz da árvore na qual será feita a correção
- `dir` (entrada) — direção (esquerda/direita) da correção
- `concluido` (entrada/saída) — indica quando a correção está concluída

A referida função retorna o endereço da raiz da árvore após passar pelas devidas correções.

```
static tNoRN *CorrigeRemocao(tNoRN *raiz, tDirecao dir, int *concluido)
{
   tNoRN *pai = raiz,
         *filho = raiz->filho[!dir],
         *r;
   tCor   guardaCor; /* Guarda a cor de um nó */
```

```
      /* Determina qual é o caso de violação e o corrige */
   if (filho && !EhVermelho(filho)) {
         /* Casos de irmão preto */
      if (!EhVermelho(filho->filho[ESQUERDA]) && !EhVermelho(filho->filho[DIREITA])){
         if (EhVermelho(pai))
            *concluido = 1;

         pai->cor = PRETO;
         filho->cor = VERMELHO;
      } else {
         guardaCor = raiz->cor;

         if (EhVermelho(filho->filho[!dir]))
            pai = RotacaoSimples(pai, dir);
         else
            pai = RotacaoDupla(pai, dir);

         pai->cor = guardaCor;
         pai->filho[ESQUERDA]->cor = PRETO;
         pai->filho[DIREITA]->cor = PRETO;

         *concluido = 1;
      }
   } else if (filho->filho[dir] != NULL) {
         /* Casos de irmão vermelho */
      r = filho->filho[dir];

      if (!EhVermelho(r->filho[ESQUERDA]) && !EhVermelho(r->filho[DIREITA])) {
         pai = RotacaoSimples(pai, dir);
         pai->filho[dir]->filho[!dir]->cor = VERMELHO;
      } else {
         if (EhVermelho(r->filho[dir]))
            filho->filho[dir] = RotacaoSimples(r, !dir);

         pai = RotacaoDupla(pai, dir);
         filho->filho[dir]->cor = PRETO;
         pai->filho[!dir]->cor = VERMELHO;
      }

      pai->cor = PRETO;
      pai->filho[dir]->cor = PRETO;

      *concluido = 1;
   }

   return pai;
}
```

A função `CorrigeRemocao()` não é tão longa como em algumas implementações de árvores rubro-negras. A economia de código é obtida notando-se similaridades entre os casos e atribuindo-se cores fora dos casos para evitar repetição de código. Para um melhor entendimento dessa função, compare os casos descritos na **Seção F.3** com suas traduções no código.

F.4.5 Implementações Ascendente e Descendente

Tanto na operação de inserção quanto na operação de remoção, reajustes podem ser necessários para resolver possíveis violações. Como foi visto nas seções anteriores, esses reajustes fazem com que os nós que constituem os caminhos de inserção ou remoção fossem visitados no sentido inverso (i. e, para cima, em direção à raiz da árvore). Essas operações de ajuste são denominadas **ascendentes** exatamente porque elas sobem na árvore em direção à raiz.

Existe outra forma de corrigir eventuais violações de uma árvore rubro-negra após inserção ou remoção, que é denominada **descendente**. Essa forma de ajuste é **preventiva** (em vez de **corretiva**), pois ela tenta assegurar que uma inserção ou remoção não causa violação, em vez de permitir que ela possa ocorrer para então corrigi-la.

Nesta seção optou-se pela implementação corretiva que é mais fácil de entender e implementar e, portanto, é mais didática.

F.5 Avaliação

Lema F.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: A prova será feita por indução sobre a altura preta da árvore cuja raiz é r.

Base da indução. Se $AP(r) = 0$, tem-se que n deve ser 0, pois, nesse caso, não há nó interno. De fato $2^0 - 1 = 0$, o que prova a base da indução.

Hipótese indutiva. Suponha que o lema vale para $AP(r) < k$.

Conclusão. Deve-se mostrar que o lema vale para $AP(r) = k + 1$. Se $AP(r) = k$, cada subárvore de r tem altura preta $k - 1$. Logo, usando-se 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 mostra que o lema vale para $AP(r) = k + 1$. ■

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

Prova: Se existir um nó vermelho no caminho desde a raiz até uma folha, deve haver um nó preto correspondente. (De outra maneira, haveria dois nós vermelhos seguidos nesse caminho, o que contraria a **Regra 4**). Portanto pelo menos a metade dos nós nesse caminho deve ser preta e, assim, $AP(r) \geq a/2$. ■

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**, $n \geq 2^{AP(r)} - 1$ e, de acordo com o **Lema F.2**, $AP(r) \geq a/2$. Portanto tem-se que:

$$\begin{aligned} n &\geq 2^{AP(r)} - 1 && \Rightarrow \\ n &\geq 2^{a/2} - 1 && \Rightarrow \\ n + 1 &\geq 2^{a/2} && \Rightarrow \\ \log_2 (n + 1) &\geq a/2 && \Rightarrow \\ a &\leq 2 \cdot \log_2 (n + 1) && \blacksquare \end{aligned}$$

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

Prova: À luz do **Teorema F.1**, a prova é trivial. ■

Em consonância com o **Corolário F.1** e com o que foi exposto na **Seção F.5**, qualquer operação básica sobre uma árvore rubro-negra tem avaliação 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 obviamente $O(1)$. Portanto o custo de reestruturação de um único nó de uma árvore rubro-negra tem custo temporal $O(1)$. 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 **Teorema F.1**, o número máximo de tais operações é menor do que 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 busca, é melhor usar árvores rubro-negras. De outra forma, é melhor usar árvores AVL.

Como a altura de uma árvore rubro-negra com *n* nós é *O(log n)*, a função `RemoveRN()` sem a chamada de `CorrigeRemocao()` tem custo temporal *O(log n)*. No interior de `CorrigeRemocao()`, cada um dos casos de correção termina após executar um número constante de alterações de cor e no máximo três rotações. O **Caso 3** é o único caso no qual o laço **while** pode ser repetido e então o ponteiro `pai` move-se para cima da árvore no máximo *O(log n)* vezes, sem executar nenhuma rotação. Assim a função `CorrigeRemocao()` tem custo temporal *O(log n)* e executa no máximo três rotações, de forma que o custo temporal de `RemoveRN()` também é *O(log n)*.

Resumindo, em termos de eficiência, os prós e contras de árvores rubro-negras são quase os mesmos de árvores AVL. Em termos de custo de implementação, apesar de serem um pouco complicadas, árvores AVL levam uma enorme vantagem com relação a árvores rubro-negras, pois as regras que definem uma árvore AVL são bem mais simples do que aquelas que definem árvores rubro-negras. A esta altura, o leitor que chegou até aqui talvez se sinta muito frustrado exatamente por tê-lo feito. Se você se sente assim, talvez fique consolado pelo fato de o autor também se sentir assim. É provável que o grande atrativo para árvores rubro-negras, seja exatamente seu nome exótico. Afinal, quem não se sente atraído por uma estrutura de dados colorida?

F.6 Comparando Árvores Binárias de Busca [Revisita]

Este livro discutiu quatro tipos de árvores binárias de busca:

[1] Árvore binária de busca ordinária (v. **Seção 4.1**)

[2] Árvore AVL (v. **Seção 4.4**)

[3] Árvore rubro-negra (presente apêndice)

[4] Árvore afunilada (v. **Seção 4.5**)

Cada abordagem de implementação de árvore binária de busca possui vantagens e desvantagens dependendo do contexto no qual elas são utilizadas. Esta seção indica em que situação cada uma delas é a mais apropriada.

Atualmente, árvores binárias de busca ordinárias são usadas apenas com o propósito didático, pois, além de serem mais fáceis de implementar, elas servem como base para implementações mais sofisticadas e complicadas.

Em qualquer implementação de tabela de busca, o que importa é o custo com que as operações básicas de busca, inserção e remoção são executadas. Mas deve-se ainda levar em consideração a dificuldade com que um programador se depara durante a própria implementação.

O foco de árvores afuniladas é em cada nó individualmente, enquanto que, em árvores balanceadas, o foco é na altura da árvore constituída pelo conjunto de nós. Quer dizer, o enfoque da técnica de afunilamento é nos elementos em si e não no formato da árvore de busca. Essa técnica funciona bem quando alguns elementos são acessados com muito mais frequência do que outros. Se elementos próximos da raiz forem acessados com a mesma frequência com que elementos em níveis bem mais baixos são acessados, então árvore afunilada não é uma boa escolha para tabela de busca. Nesse caso, é melhor utilizar uma árvore com autobalanceamento (i.e., árvore AVL ou rubro-negra).

Na **Seção F.5**, foi visto que, em termos de custos temporal e espacial, árvores AVL e árvores rubro-negras são equiparáveis, mas a implementação de árvores rubro-negras é indubitavelmente bem mais complicada do que

a implementação de árvores AVL. Assim é melhor usar a **Navalha de Occam** (o **Princípio da Parcimônia**) e ficar distante de árvores rubro-negras.

A título de comparação, a **Tabela F–2** mostra as alturas das árvores binárias de busca criadas para o arquivo `CEPs.bin` descrito no **Apêndice A**. Como esse arquivo possui *673.580* registros, a profundidade (ideal) de uma árvore perfeitamente balanceada seria igual a *20* (i. e, log_2 *673580*), que, conforme foi discutido na **Seção 4.4**, não faz sentido tentar obter na prática.

Abordagem	Profundidade da árvore
Árvore Binária Ordinária	*3.376*
Árvore AVL	*24*
Árvore Rubro-negra	*27*
Árvore Afunilada	*445*

Tabela F–2: Desempenho de Árvores Binárias de Busca com o Arquivo CEPs.bin

A **Tabela F–2** mostra que os resultados relativos obtidos condizem com a expectativa. Ou seja, era esperado que uma árvore binária ordinária fosse bastante desbalanceada, visto que muitas chaves (CEPs) no arquivo `CEPs.bin` estão ordenadas. Por outro lado, o fato de a árvore AVL e a árvore rubro-negra possuírem alturas próximas da profundidade ideal, com pequena vantagem para a árvore AVL, também era esperado. Enfim a implementação de tabela de busca com árvore afunilada parece ruim, visto que a profundidade dessa árvore é cerca de 20 vezes maior do que a profundidade ideal. Mas novamente, lembre-se que árvore afunilada não é árvore balanceada, de forma que uma profundidade ainda maior do que essa ainda seria aceitável.

A análise amortizada de árvores afuniladas, apresentada no **Capítulo 5**, garante que *m* operações consecutivas são executadas com custo temporal máximo $O(m \cdot log\ n)$, mas essa garantia não exclui a possibilidade de uma única operação ter custo temporal $O(n)$. Portanto embora essa garantia de custo temporal não seja tão enfática quanto nos casos de árvores AVL ou rubro-negras, que apresentam custos temporais $O(log\ n)$ no pior caso, o desempenho líquido de árvores afuniladas quando se considera uma sequência de operações é o mesmo exibido por árvores balanceadas.

F.7 Checando Árvores Rubro-negras

Problema: Escreva uma função que verifica se uma árvore supostamente rubro-negra é realmente uma árvore rubro-negra.

Solução: A seguinte definição de tipo enumeração facilitará o entendimento da função solicitada.

```
typedef enum { VIOLACAO_NENHUMA = 1, VIOLACAO_ARVORE_DE_BUSCA,
               VIOLACAO_VERMELHA, VIOLACAO_PRETA } tViolacao;
```

A função `EhRubroNegra()` verifica se uma árvore supostamente rubro-negra é realmente rubro-negra. Seu único parâmetro é um ponteiro para a raiz da árvore e ela retorna um valor do tipo `tViolacao` indicando se ocorreu alguma violação das regras de definição de árvores rubro-negras.

```
static tViolacao EhRubroNegra(tNoRN *raiz)
{
   int    alturaEsq,
          alturaDir;
   tNoRN *noEsquerda,
         *noDireita;

      /* Se a árvore estiver vazia, ela é rubro-negra */
```

```
   if (!raiz)
      return 1;

   noEsquerda = raiz->filho[0];
   noDireita = raiz->filho[1];

      /* Checa violação vermelha */
   if (EhVermelho(raiz))
      if (EhVermelho(noEsquerda) || EhVermelho(noDireita))
         return VIOLACAO_VERMELHA;

   alturaEsq = EhRubroNegra(noEsquerda);
   alturaDir = EhRubroNegra(noDireita);

      /* Checa alturas pretas */
   if (alturaEsq && alturaDir && alturaEsq != alturaDir)
      return VIOLACAO_PRETA;

      /* Conta apenas nós pretos */
   if (alturaEsq && alturaDir)
      return EhVermelho(raiz) ? alturaEsq : alturaEsq + 1;
   else
      return VIOLACAO_PRETA;
}
```

A função `EhRubroNegra()` é relativamente simples. Ela visita cada nó na árvore e executa certos testes no nó e em seus filhos. O primeiro teste é para verificar se um nó vermelho tem filhos vermelhos. O segundo teste assegura que a árvore é uma árvore de busca binária válida. O último teste conta os nós pretos ao longo de um caminho e assegura que todos os caminhos têm a mesma altura preta.

A função `AvaliaArvoreRubroNegra()` chama `EhRubroNegra()` para testar uma árvore rubro-negra e apresenta o resultado na tela. Os parâmetros da função `AvaliaArvoreRubroNegra()`, que será apresentada adiante são:

- `arvore` (entrada) — ponteiro que representa a árvore
- `compara` (entrada) — ponteiro para uma função que compara conteúdos dos nós de acordo com o mesmo critério de comparação de **qsort()**

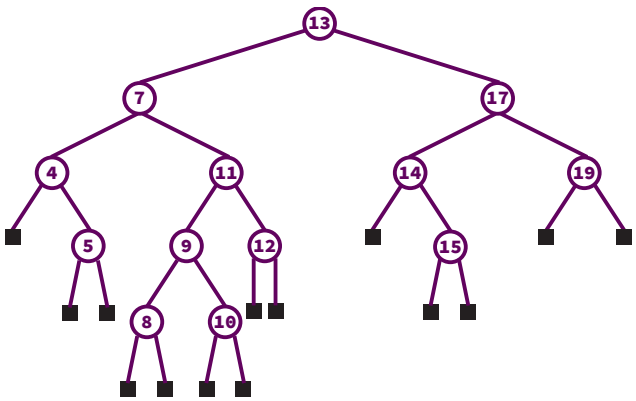
```
int AvaliaArvoreRubroNegra(tArvoreRN *arvore, tCompara compara)
{
   tViolacao resultado;

      /* Checa se a árvore é uma árvore binária de busca */
   if (!EhArvoreBinDeBuscaRN(arvore->raiz, compara))
      resultado = VIOLACAO_ARVORE_DE_BUSCA;
   else
      resultado = EhRubroNegra(arvore->raiz);

   switch (resultado) {
      case VIOLACAO_VERMELHA:
         puts("\n\t>>> Ocorreu uma violacao vermelha <<<");
         return 0;

      case VIOLACAO_PRETA:
         puts("\n\t>>> Ocorreu uma violacao preta <<<");
         return 0;

      case VIOLACAO_ARVORE_DE_BUSCA:
         puts("Essa nao e' uma arvore binaria de busca <<<<");
         return 0;
```

```
      default:
         puts( "\n\t>>> Essa e' uma legitima arvore rubro-negra <<<" );
         return 1;
   }
}
```

A função `AvaliaArvoreRubroNegra()` chama `EhArvoreBinDeBuscaRN()` para checar se a árvore recebida como parâmetro é de fato uma árvore de busca. Essa última função é semelhante à função `EhArvoreBinDeBusca()` apresentada na **Seção 4.7.3**.

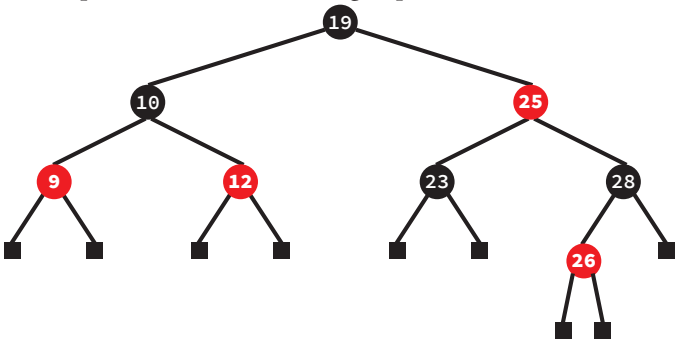
F.8 Exercícios de Revisão

1. Quais são as regras que uma árvore binária de busca deve obedecer para ser considerada uma árvore rubro-negra?
2. Explique o que coloração de nós em árvores rubro-negras tem a ver com o balanceamento dessas árvores.
3. (a) Existe alguma árvore rubro-negra contendo apenas nós pretos? (b) Existe árvore rubro-negra contendo apenas nós vermelhos? Explique seu raciocínio.
4. Se os nós de uma árvore rubro-negra podem ser todos pretos, por que existem nós vermelhos?
5. A ordem com que um conjunto de chaves é inserido numa árvore rubro-negra influencia a eficiência das operações básicas sobre essa árvore?
6. (a) O que é caminho de inserção? (b) O que é caminho de remoção? (c) O que é caminho inverso de inserção? (d) O que é caminho inverso de remoção?
7. Mostre que, numa árvore rubro-negra, o maior caminho de um nó para uma folha tem comprimento, no máximo, igual a duas vezes o tamanho do menor caminho desse nó para uma folha.
8. (a) Desenhe uma árvore rubro-negra cujos nós possuam balanceamento de uma árvore AVL. (b) Desenhe uma árvore rubro-negra cujos nós não possuam balanceamento de uma árvore AVL.
9. (a) Qual é o maior número de nós não nulos de uma árvore rubro-negra com altura preta igual a h. (b) Qual é o menor número possível de nós nesse caso?
10. Quais são as propriedades que podem ser violadas logo após a inserção de um novo nó numa árvore rubro-negra?
11. Suponha que os nós de uma árvore rubro-negra contenham apenas chaves inteiras cujos valores são: *10*, *20*, *30*, *40*, *50*, *60*, *70* e *80*. Desenhe essa árvore quando os nós são inseridos nessa mesma ordem.
12. Atribua uma cor para cada nó da árvore rubro-negra da figura abaixo. (Os nós nulos já estão coloridos.)



13. Mostre que, se uma árvore binária de busca é constituída por três nós que formam uma cadeia, como nas figuras abaixo, ela não pode constituir uma árvore rubro-negra para quaisquer cores que sejam atribuídas aos seus nós.

14. (a) Apresente a árvore resultante da inserção de um nó com conteúdo *27* na árvore rubro-negra ilustrada na figura abaixo. (b) Que tipo de violação ocorre logo após essa inserção?



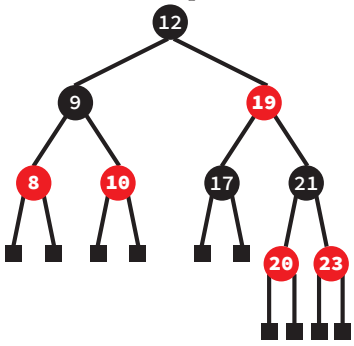
15. (a) Apresente a árvore resultante da remoção do nó contendo a chave *12* na árvore rubro-negra ilustrada na figura do exercício **14**. (b) Que tipo de violação ocorre logo após essa remoção?

16. (a) Apresente a árvore resultante da remoção do nó contendo a chave *25* na árvore rubro-negra ilustrada na figura do exercício **14**. (b) Ocorre alguma violação após essa remoção?

17. Por que se o nó removido numa árvore rubro-negra for vermelho, não ocorrerá nenhuma violação?

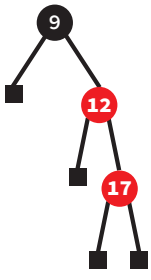
18. Por que se diz que árvore rubro-negra é uma árvore com autobalanceamento se a definição de árvores dessa natureza sequer menciona balanceamento?

19. (a) O que é altura preta de um nó? (b) Qual é a altura preta de cada nó da árvore rubro-negra da figura abaixo?

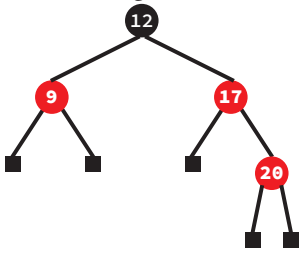


20. Defina: (a) violação preta e (b) violação vermelha.

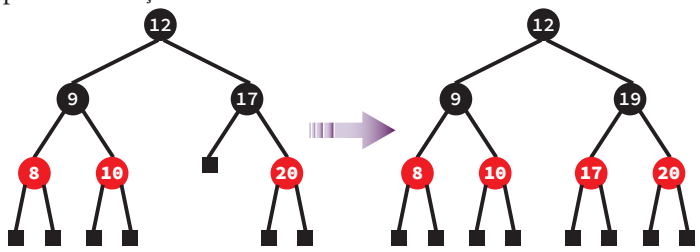
21. Quando o nó contendo a chave *17* é inserido na árvore da figura abaixo, uma das regras que definem uma árvore rubro-negra é violada. (a) Qual é essa regra? (b) Essa violação correspondente a qual dos casos na **Seção F.2**? (c) Como essa violação é corrigida?



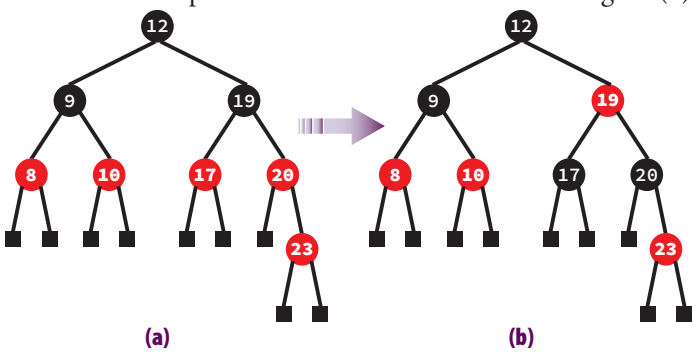
22. Na figura abaixo, um nó com chave igual a *20* é inserido, o que faz com que a árvore rubro-negra se torne inválida. (a) Que violação é essa? (b) Como corrigi-la?



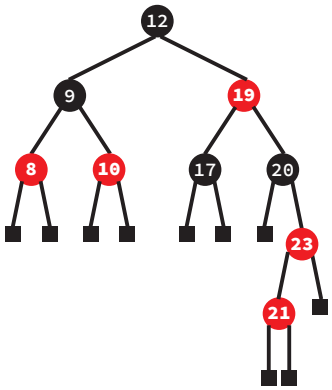
23. Na árvore rubro-negra da figura (a) abaixo, é inserido um nó com chave igual a *19*. De acordo com o procedimento descrito na **Seção F.2**, a árvore resultante é aquela mostrada na figura (b) abaixo. Descreva as operações efetuada para a obtenção dessa última árvore.



24. O nó contendo a chave *23* acaba de ser inserido na árvore ilustrada na figura (a) abaixo. Como, após essa inserção, esta árvore é reestruturada para resultar na árvore ilustrada na figura (b) abaixo?



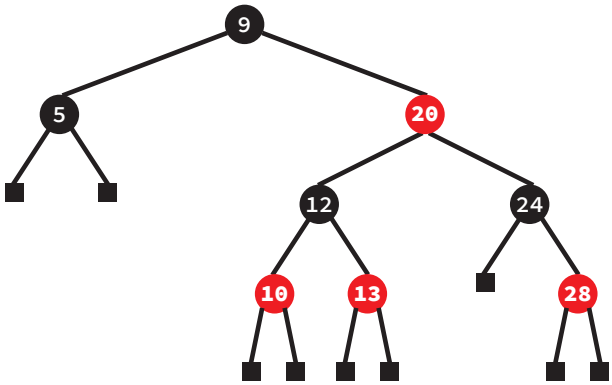
25. Na figura a seguir a árvore rubro-negra ilustrada era válida antes da inserção do nó contendo a chave *21*. (a) Que tipo de violação ocorreu com essa árvore para torná-la inválida após a inserção desse nó? (b) Mostre como corrigir essa árvore de modo a torná-la uma árvore rubro-negra válida novamente.



26. Desenhe a árvore rubro-negra resultante da inserção (nesta ordem) dos nós contendo as chaves *10*, *20*, *30* e *17*, considerando que árvore está inicialmente vazia.

27. 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.

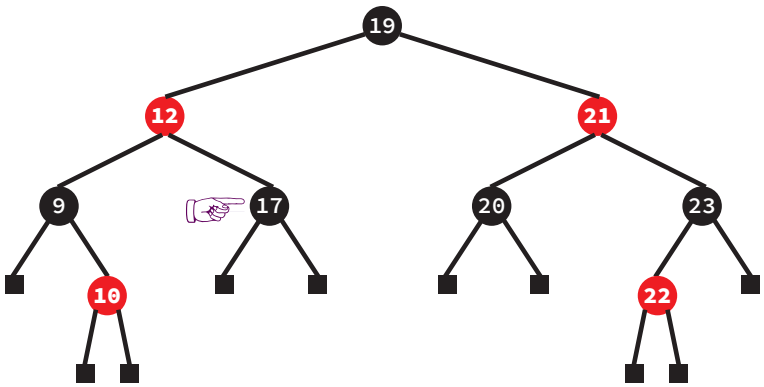
28. Apresente a árvore resultante da inserção dos nós contendo as chaves *12*, *22*, *32* e *15*, nessa ordem, na árvore rubro-negra da figura abaixo.



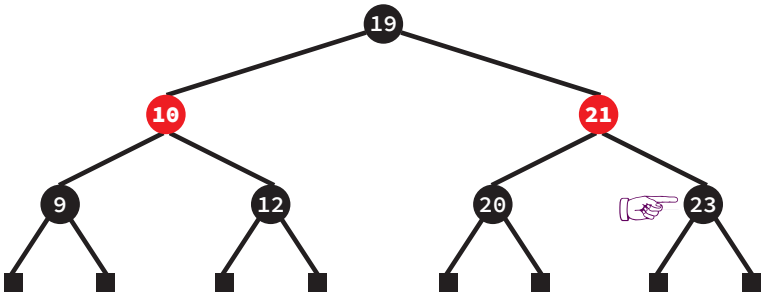
29. Por que é impossível que a remoção de nó vermelho cause uma violação das regras para árvores rubro-negras?

30. Uma árvore rubro-negra contém apenas nós pretos. Qual é o formato dessa árvore?

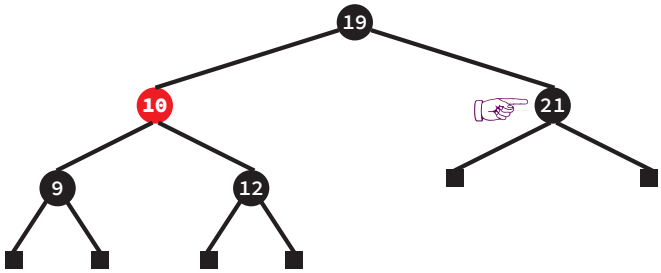
31. (a) Que tipo de violação ocorre após a remoção do nó contendo a chave *17* na figura abaixo? (b) Como essa violação pode ser corrigida?



32. (a) Que tipo de violação ocorre após a remoção do nó contendo a chave *23* na figura abaixo? (b) Como essa violação pode ser corrigida?



33. (a) Que tipo de violação ocorre após a remoção do nó contendo a chave 21 na figura abaixo? (b) Como essa violação pode ser corrigida?



34. Uma maneira alternativa de implementação de árvores rubro-negras pode utilizar, na definição do tipo de nó, um campo adicional que armazena um ponteiro para o pai de cada nó. Esse campo permite acessar nós em níveis superiores após uma inserção ou remoção de modo a corrigir eventuais violações. (a) Qual é a desvantagem dessa abordagem? (b) Qual é a vantagem dessa abordagem?

35. Existe outra alternativa de implementação de operações de inserção e remoção em árvores rubro-negras que não usa recursão ou campo adicional. Como é essa implementação?

36. Descreva os casos de violação de regras de árvores rubro-negras devido a uma inserção.

37. Para que serve o tipo `tDirecao` usado na implementação de árvores rubro-negras?

38. Por que a função `RotacaoSimples()` altera as cores dos nós participantes de uma operação de rotação?

39. A função `RotacaoDupla()` altera a cor de algum nó?

40. Por que existem duas funções para efetuar rotações simples em nós na implementação de árvores AVL e apenas uma função com a mesma finalidade na implementação de árvores rubro-negras?

41. Por que é mais complicado corrigir violações devido a remoção do que violações devido a inserção em árvores rubro-negras?

42. Descreva as implementações ascendente e descendente de árvores rubro-negras.

43. Por que cada novo nó a ser inserido numa árvore rubro-negra possui inicialmente a cor vermelha (e não preta)?

44. Por que a remoção de um nó vermelho nunca viola as regras de definição de árvores rubro-negras?

45. Por que a remoção de um nó preto pode causar uma violação vermelha?

46. Suponha que todos os nós de uma árvore rubro-negra sejam pretos. Qual deverá ser o formato dessa árvore?

47. Apresente uma comparação entre árvores AVL e árvores rubro-negras.

48. Como se verifica se uma árvore binária pode ser classificada como uma árvore rubro-negra?

F.9 Exercícios de Programação

EP1 Uma maneira alternativa de implementação de árvores rubro-negras utiliza, na definição do tipo de nó, um campo adicional que armazena o pai de cada nó. Esse campo permite subir na árvore após uma inserção ou remoção de modo a corrigir eventuais violações. Implemente funções que efetuem inserção e remoção em árvores rubro-negras usando essa abordagem.

EP2 Acrescente instruções de tratamento de exceção às funções de implementação de árvores rubro-negras apresentadas na **Seção F.4**.

F.10 Respostas e Sugestões para os Exercícios de Revisão

1. Consulte a **Seção F.1**.

2. Formalmente, consulte a **Seção F.5**. Informalmente, a **Regra 5** assegura que uma árvore rubro-negra que não contém nenhum nó vermelho é balanceada, pois cada caminho da raiz até uma folha contém o mesmo

número de nós pretos. Acrescentando-se nós vermelhos, a **Regra 4** garante que num caminho da raiz até uma folha com k nós pretos, há no máximo k nós vermelhos, de modo que a adição de nós vermelhos no máximo duplica a altura da árvore.

3. (a) Sim. Por exemplo, a árvore da **Figura F–24** é uma árvore rubro-negra legítima. (b) Não, pois isso violaria a regra que requer que a raiz da árvore seja preta.

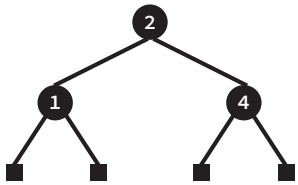


FIGURA F–24: QUESTÃO 3 (A)

4. Pela **Regra 1**, a raiz de uma árvore rubro-negra deve ser preta. (Além disso, um nó vermelho não pode ter filho vermelho.)
5. Não.
6. Consulte a **Seção 4.4.1**.
7. Use um raciocínio semelhante àquele usado na prova do **Lema F.2**.
8. (a) V. **Figura F–25 (a)**. (b) V. **Figura F–25 (b)**.

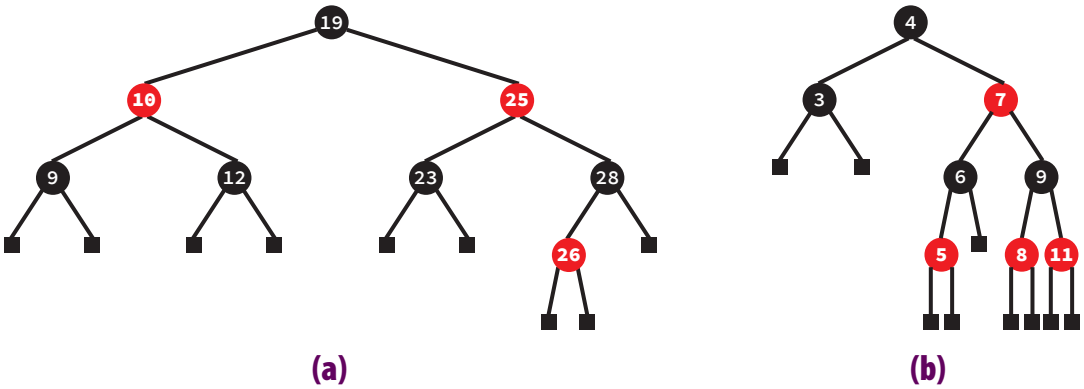


FIGURA F–25: QUESTÃO 8

9. (a) O número máximo de nós que uma árvore binária pode ter é $2^p - 1$, sendo p a profundidade (ou altura da árvore), Mas, de acordo com o **Lema F.2**, $h \geq p/2$, sendo h a altura preta da árvore. Assim o número máximo de nós numa árvore rubro-negra de altura preta h é $2^{2h} - 1$. (b) V. **Lema F.1**.
10. Consulte a **Seção F.2**.
11. V. **Figura F–26**.

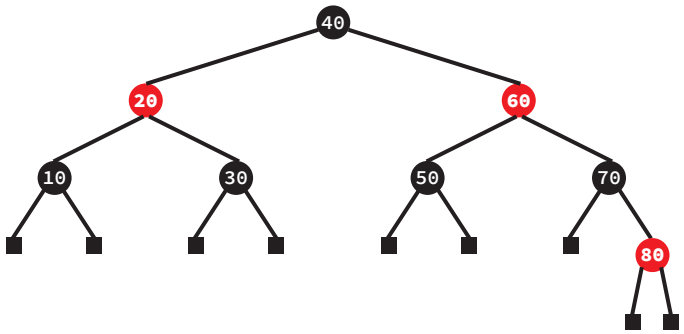


FIGURA F–26: QUESTÃO 11

12. V. **Figura F–27**.

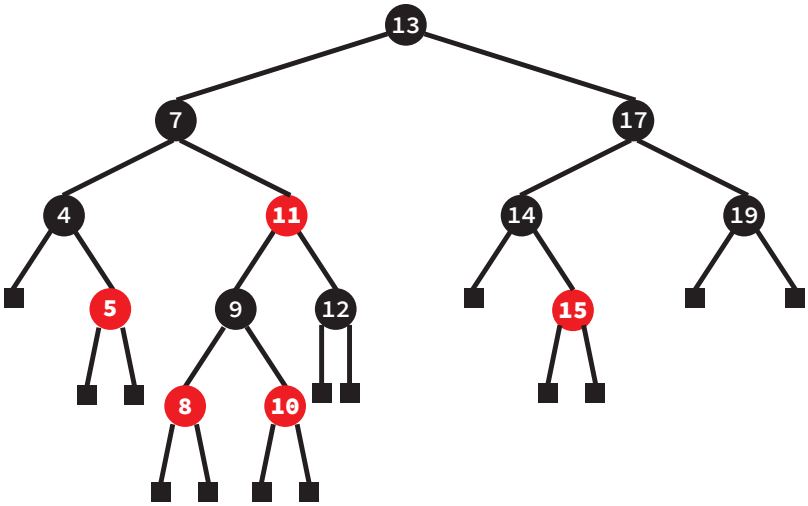
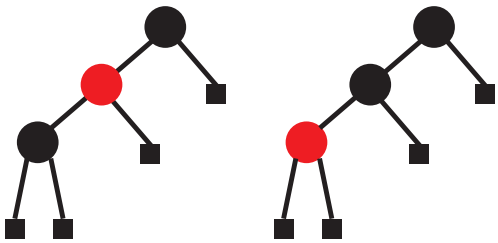


FIGURA F–27: QUESTÃO 12

13. Em primeiro lugar, o raciocínio empregado para a árvore inclinada à esquerda vale para a outra árvore. Assim o foco será na árvore inclinada à esquerda. A raiz dessa árvore deve ser preta (**Regra 1**), mas os demais nós não podem ser todos pretos (pois isso violaria a **Regra 5**) nem todos vermelhos (pois isso violaria a **Regra 4**). Portanto restam as opções mostradas na figura **Figura F–28**. Mas essas alternativas também violam a **Regra 5**. Conclusão: tais árvores rubro-negras não podem existir.



(a) Opção 1 **(b) Opção 2**

FIGURA F–28: QUESTÃO 13

14. (a) V. **Figura F–29**. (b) Violação vermelha.

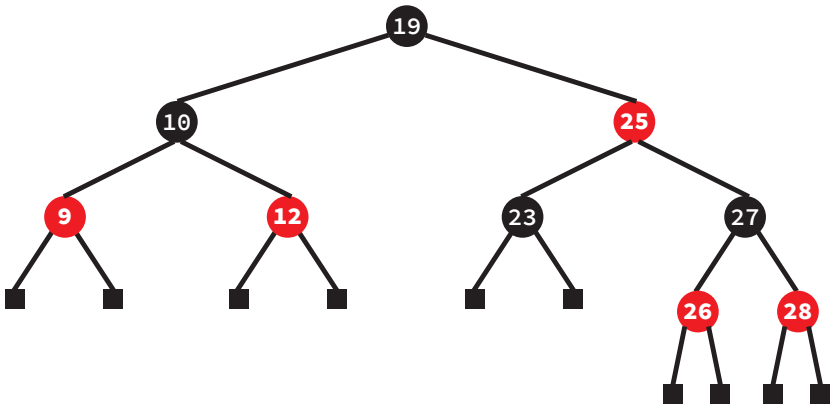


FIGURA F–29: QUESTÃO 14 (a)

15. (a) V. **Figura F–30**. (b) Nenhuma.

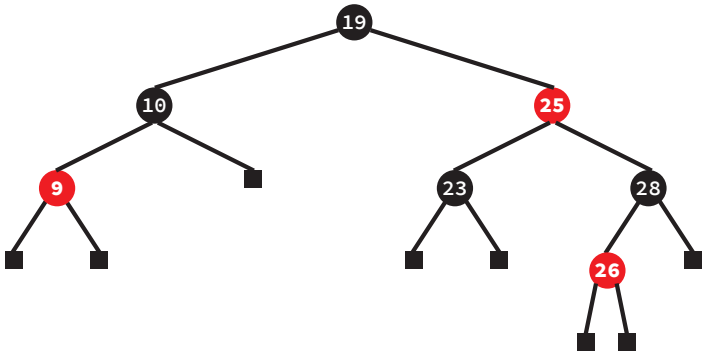


FIGURA F–30: QUESTÃO 15 (a)

16. (a) V. **Figura F–31**. (b) Não.

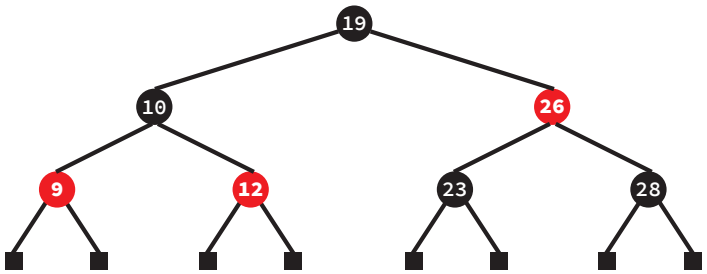


FIGURA F–31: QUESTÃO 16 (a)

17. Consulte a **Seção F.3**.

18. As regras de definição de árvores rubro-negras requerem, implicitamente, que elas sejam balanceadas.

19. (a) Consulte a **Seção F.1**. (b) Raiz: *2*; nó com chaves *9*, *17*, *19* e *21*: *1*; demais nós: *0*.

20. Consulte a **Seção F.1**.

21. A **Regra 4** é violada visto que, nesse caso, têm-se nós vermelhos duplicados. Logo essa inserção requer reestruturação da árvore, que é obtida por meio de rotação, como mostra a **Figura F–32**.

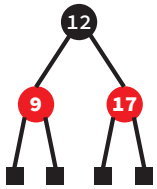


FIGURA F–32: QUESTÃO 21

22. Esse caso é corrigido por meio de recoloração de nós (em vez de rotação), o que faz com que a árvore adquira a configuração mostrada na **Figura F–33**.

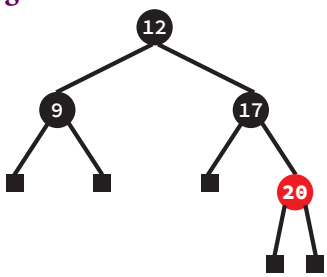


FIGURA F–33: QUESTÃO 22

23. V. **Figura F–34**.

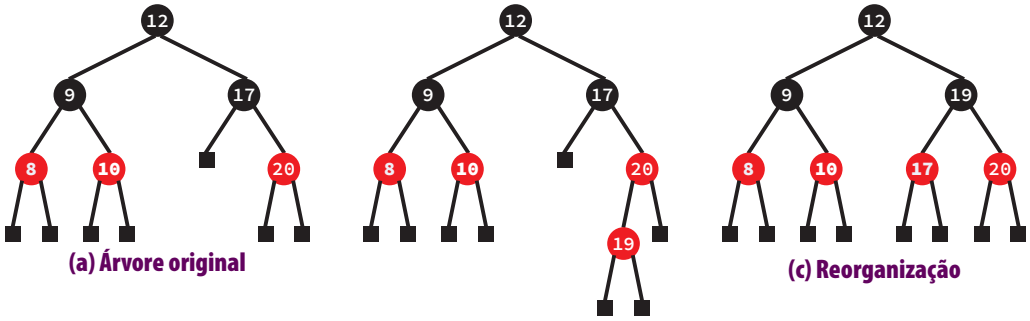


Figura F–34: Questão 23

24. V. **Figura F–35**.

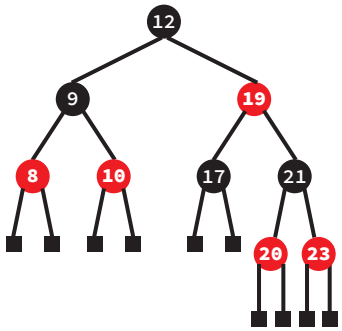


Figura F–35: Questão 24

25. (a) Violação vermelha. (b) V. **Figura F–36**.

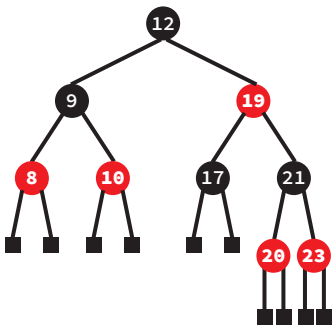


Figura F–36: Questão 25 (b)

26. A árvore rubro-negra resultante da inserção (nesta ordem) dos nós contendo as chaves *10*, *20*, *30* e *17* é apresentada na **Figura F–37**.

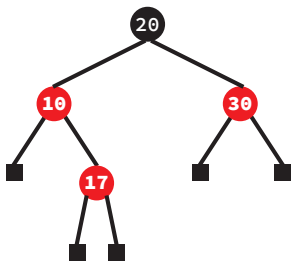


Figura F–37: Questão 26

27. V. **Figura F–38**.

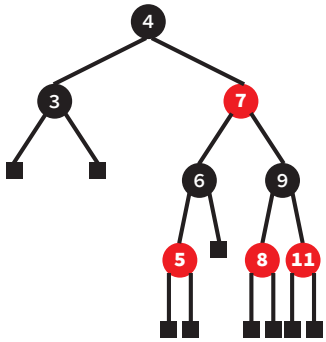


Figura F–38: Questão 27

28. V. **Figura F–39**.

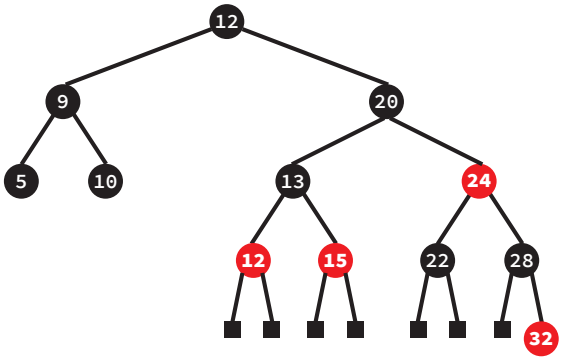


Figura F–39: Questão 28

29. Consulte a **Seção F.3**.

30. Essa árvore só terá um nó (raiz) ou será constituída por uma raiz com dois filhos (v. resposta da questão **3**).

31. V. **Figura F–40**.

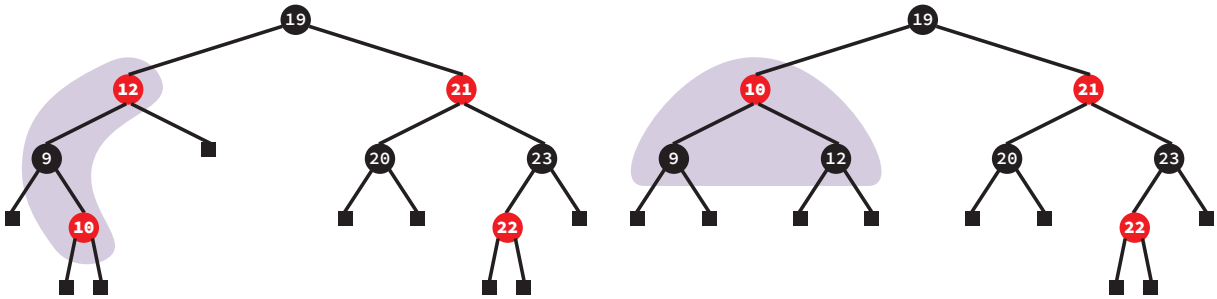


Figura F–40: Questão 31

32. V. **Figura F–41**.

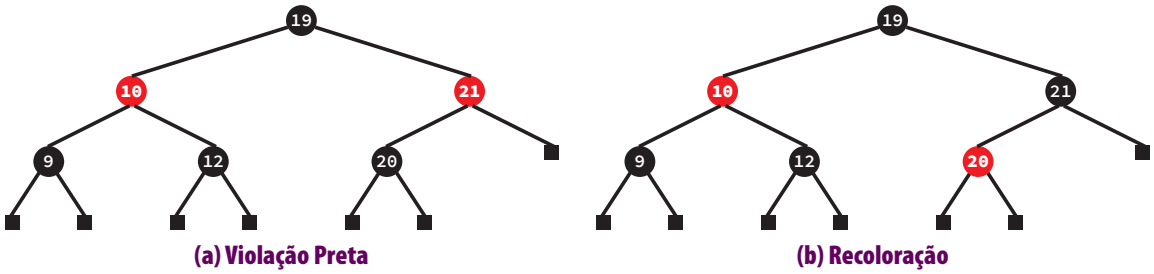


FIGURA F–41: QUESTÃO 32

33. V. **Figura F–42**.

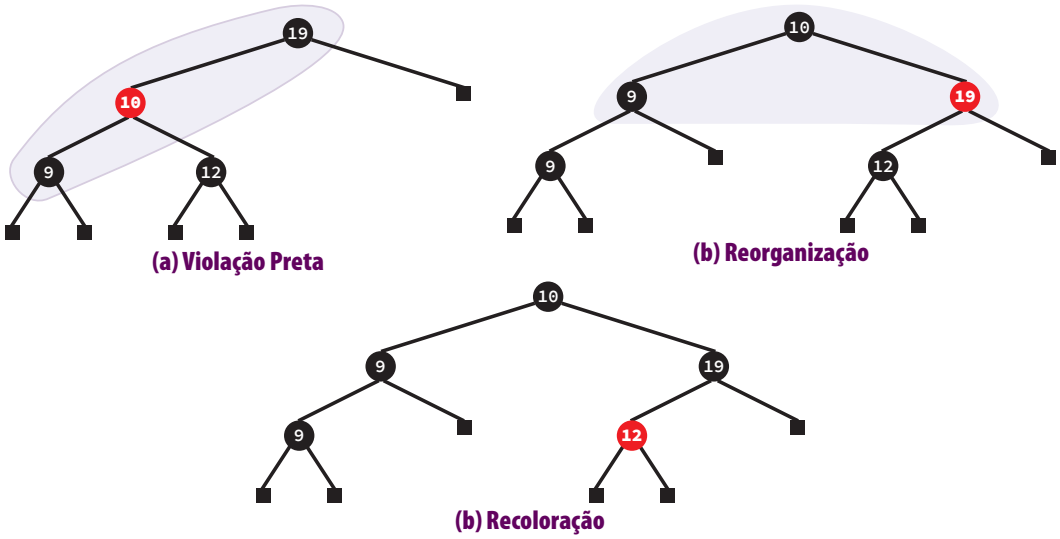


FIGURA F–42: QUESTÃO 33

34. (a) Dispêndio maior de memória. (b) Esse ponteiro permite subir até a raiz da árvore durante uma operação de inserção ou remoção sem o uso de pilha ou recursão.

35. Essa implementação deve usar uma pilha que armazena os nós visitados durante uma dessas operações.

36. Consulte a **Seção F.2**.

37. As constantes dessa enumeração servem para especificar se uma rotação será à esquerda ou à direita.

38. A nova raiz é colorida com preto para satisfazer a **Regra 2** e a antiga raiz é colorida com vermelho para não causar violação preta da nova raiz.

39. Sim (indiretamente).

40. Consulte a **Seção F.4.2**.

41. Consulte a **Seção F.3**.

42. Consulte a **Seção F.4.5**.

43. Consulte a **Seção F.2**.

44. Consulte a **Seção F.3**.

45. Consulte a **Seção F.3**.

46. Essa árvore deverá ser uma árvore binária perfeita (ou repleta).

47. Consulte a **Seção F.6**.

48. Consulte a **Seção F.7**.