

O Capítulo 12 mostrou que uma árvore de busca binária de altura  $h$  pode suportar qualquer das operações básicas de conjuntos dinâmicos — como `SEARCH`, `PREDECESSOR`, `SUCCESSOR`, `MINIMUM`, `MAXIMUM`, `INSERT` e `DELETE` — no tempo  $O(h)$ . Assim, as operações de conjuntos são rápidas se a altura da árvore de busca é pequena. Todavia, se a altura da árvore é grande, a execução dessas operações poderá ser mais lenta do que com uma lista ligada. Árvores vermelho-preto são um dos muitos esquemas de árvores de busca que são “balanceadas” de modo a garantir que operações básicas de conjuntos dinâmicos demorem o tempo  $O(\lg n)$  no pior caso.

## 13.1 PROPRIEDADES DE ÁRVORES VERMELHO-PRETO

Uma **árvore vermelho-preto** é uma árvore de busca binária com um bit extra de armazenamento por nó: sua *cor* — ela pode ser **VERMELHA** ou **PRETA**. Restringindo as cores dos nós em qualquer caminho simples da raiz até uma folha, as árvores vermelho-preto asseguram que o comprimento de nenhum desses caminhos seja maior que duas vezes o de qualquer outro, de modo que a árvore é aproximadamente **balanceada**.

Cada nó da árvore contém agora os atributos *cor*, *chave*, *esquerda*, *direita* e *p*. Se um filho ou o pai de um nó não existir, o atributo do ponteiro correspondente do nó contém o valor `NIL`. Trataremos esses valores `NIL` como se fossem ponteiros para folhas (nós externos) da árvore de busca binária e os nós normais que portam chaves como nós internos da árvore.

Uma árvore vermelho-preto é uma árvore de busca binária que satisfaz as seguintes **propriedades vermelho-preto**:

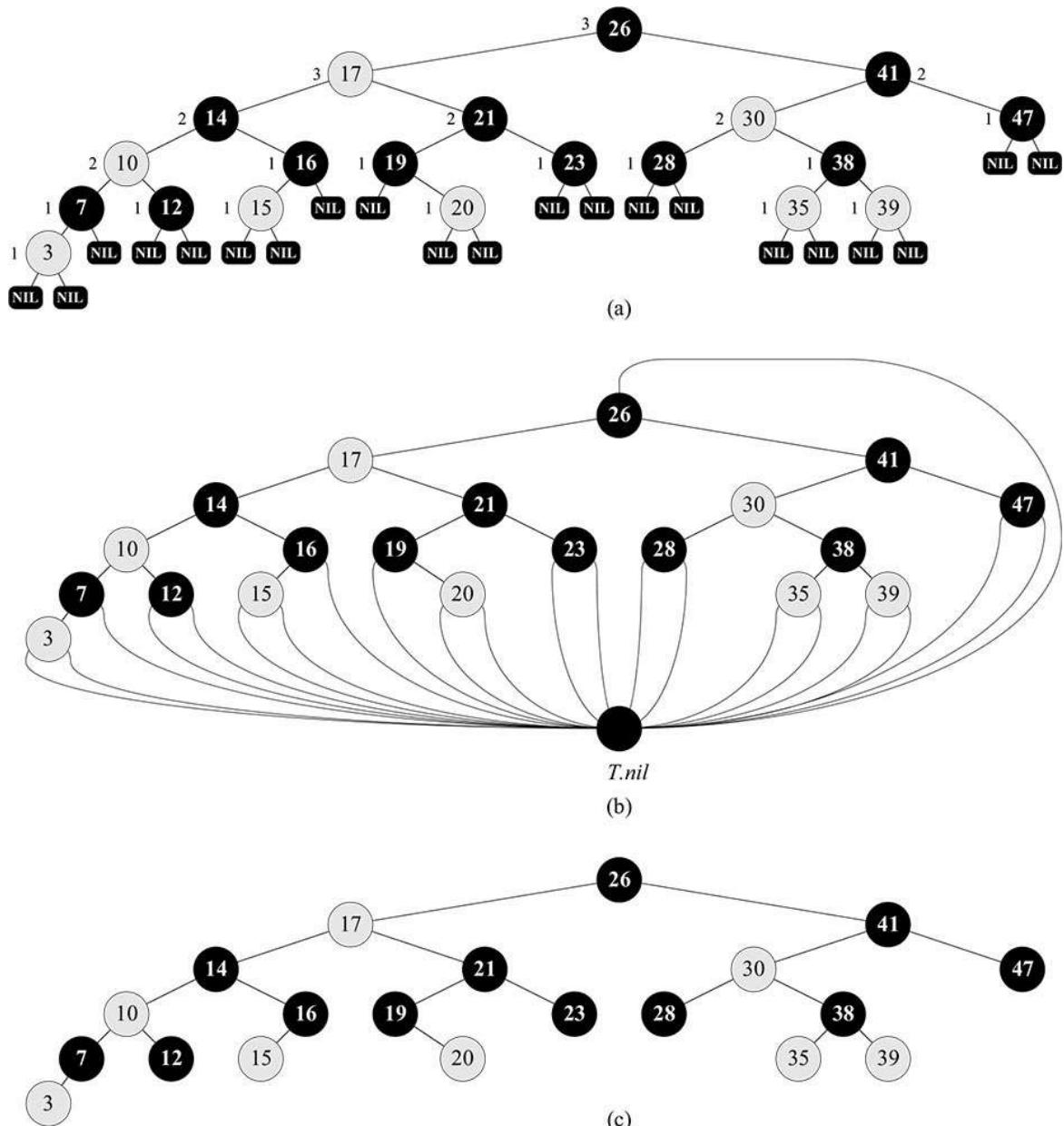
1. Todo nó é vermelho ou preto.
2. A raiz é preta.
3. Toda folha (`NIL`) é preta.
4. Se um nó é vermelho, então os seus filhos são pretos.
5. Para cada nó, todos os caminhos simples do nó até folhas descendentes contêm o mesmo número de nós pretos.

A Figura 13.1 mostra um exemplo de árvore vermelho-preto.

Por questão de conveniência no tratamento das condições de fronteira em código de árvores vermelho-preto, usamos uma única sentinela para representar `NIL` (veja p. 238). Para uma árvore vermelho-preto  $T$ , a sentinela  $T.nil$  é um objeto com os mesmos atributos que um nó comum na árvore. Seu atributo *cor* é **PRETO** e seus outros atributos — *p*, *esquerda*, *direita* e *chave* — podem adotar valores arbitrários. Como mostra a Figura 13.1(b), todos os ponteiros para `NIL` são substituídos por ponteiros para a sentinela  $T.nil$ .

Usamos a sentinela para poder tratar um filho `NIL` de um nó  $x$  como um nó comum cujo pai é  $x$ . Se bem que poderíamos adicionar um nó de sentinela distinto para cada `NIL` na árvore, de modo que o pai de cada `NIL` fosse bem definido, essa abordagem desperdiçaria espaço. Em vez disso, usamos a única sentinela  $T.nil$  para representar todos os

nós `NIL` — todas as folhas e o pai da raiz. Os valores dos atributos *p*, *esquerda*, *direita* e *chave* da sentinela são irrelevantes, embora, por conveniência, possamos defini-los durante o curso de um procedimento.



**Figura 13.1** Uma árvore vermelho-preto com nós pretos em negrito e nós vermelhos em cinzento. Todo nó em uma árvore vermelho-preto é vermelho ou preto, os filhos de um nó vermelho são pretos, e todo caminho simples de um nó até uma folha descendente contém o mesmo número de nós pretos. **(a)** Toda folha, mostrada como um `NIL`, é preta. Cada nó não `NIL` é marcado com sua altura preta: nós `NIL`s têm altura preta igual a 0. **(b)** A mesma árvore vermelho-preto, mas com cada `NIL` substituído pela única sentinela `T.nil`, que é sempre preta, e cujas alturas pretas são omitidas. O pai da raiz também é a sentinela. **(c)** A mesma árvore vermelho-preto, mas com folhas e o pai da raiz omitidos completamente. Utilizaremos esse estilo de representação no restante deste capítulo.

Em geral, limitamos nosso interesse aos nós internos de uma árvore vermelho-preto, já que eles contêm os valores de chaves. No restante deste capítulo, omitiremos as folhas quando desenharmos árvores vermelho-preto, como mostra a Figura 13.1(c).

Denominamos o número de nós pretos em qualquer caminho simples de um nó  $x$ , sem incluir esse nó, até uma folha, por **altura preta** do nó, denotada por  $bh(x)$ . Pela propriedade 5, a noção de altura preta é bem definida, já que

todos os caminhos simples descendentes que partem do nó têm o mesmo número de nós pretos. Definimos a altura preta de uma árvore vermelho-preto como a altura preta de sua raiz.

O lema a seguir, mostra por que as árvores vermelho-preto dão boas árvores de busca.

### Lema 13.1

Uma árvore vermelho-preto com  $n$  nós internos tem, no máximo, a altura  $2 \lg(n + 1)$ .

**Prova** Começamos mostrando que a subárvore com raiz em qualquer nó  $x$  contém no mínimo  $2^{bh(x)} - 1$  nós internos. Provamos essa afirmativa por indução sobre a altura de  $x$ . Se a altura de  $x$  é 0, então  $x$  deve ser uma folha ( $T.nil$ ), e a subárvore com raiz em  $x$  realmente contém no mínimo  $2^{bh(x)} - 1 = 2^0 - 1 = 0$  nós internos. Para a etapa indutiva, considere um nó  $x$  que tenha altura positiva e considere um nó interno  $x$  com dois filhos. Cada filho tem uma altura preta  $bh(x)$  ou  $bh(x) - 1$ , dependendo de sua cor ser vermelha ou preta, respectivamente. Visto que a altura de um filho de  $x$  é menor que a altura do próprio  $x$ , podemos aplicar a hipótese indutiva para concluir que cada filho tem, no mínimo,  $2^{bh(x)} - 1 - 1$  nós internos. Assim, a subárvore com raiz em  $x$  contém, no mínimo,  $(2^{bh(x)} - 1 - 1) + (2^{bh(x)} - 1 - 1) + 1 = 2^{bh(x)} - 1$  nós internos, o que prova a afirmativa.

Para completar a prova do lema, seja  $h$  a altura da árvore. De acordo com a propriedade 4, no mínimo metade dos nós em qualquer caminho simples da raiz até uma folha, não incluindo a raiz, deve ser preta. Consequentemente, a altura preta da raiz deve ser, no mínimo,  $h/2$ ; assim,

$$n \geq 2^{h/2} - 1 .$$

Passando o valor 1 para o lado esquerdo e tomado logaritmos em ambos os lados, temos  $\lg(n + 1) \geq h/2$  ou  $h \leq 2 \lg(n + 1)$ .

Uma consequência imediata desse lema é que podemos implementar as operações de conjuntos dinâmicos `SEARCH`, `MINIMUM`, `MAXIMUM`, `SUCCESSOR` e `PREDECESSOR` no tempo  $O(\lg n)$  em árvores vermelho-preto, já que cada execução no tempo  $O(h)$  em uma árvore de busca de altura  $h$  (como mostra o Capítulo 12) e em qualquer árvore vermelho-preto em  $n$  nós é uma árvore de busca com altura  $O(\lg n)$ . (Claro que as referências a `NIL` nos algoritmos do Capítulo 12 teriam de ser substituídas por  $T.nil$ .) Embora os algoritmos `TREE-INSERT` e `TREE-DELETE` do Capítulo 12 sejam executados no tempo  $O(\lg n)$  quando é dada uma árvore vermelho-preto como entrada, eles não suportam diretamente as operações de conjuntos dinâmicos `INSERT` e `DELETE`, já que não garantem que a árvore de busca binária modificada será uma árvore vermelho-preto. Porém, veremos nas Seções 13.3 e 13.4 como suportar essas duas operações no tempo  $O(\lg n)$ .

## Exercícios

---

- 13.1-1** Desenhe, no estilo da Figura 13.1(a), a árvore de busca binária completa de altura 3 nas chaves {1, 2, ..., 15}. Adicione as folhas `NIL` e dê três cores diferentes aos nós, de tal modo que as alturas pretas das árvores vermelho-preto resultantes sejam 2, 3 e 4.
- 13.1-2** Desenhe a árvore vermelho-preto que resulta após a chamada a `TREE-INSERT` na árvore da Figura 13.1 com chave 36. Se o nó inserido for vermelho, a árvore resultante é uma árvore vermelho-preto? E se ele for preto?
- 13.1-3** Vamos definir uma **árvore vermelho-preto relaxada** como uma árvore de busca binária que satisfaz as propriedades vermelho-preto 1, 3, 4 e 5. Em outras palavras, a raiz pode ser vermelha ou preta. Considere uma árvore vermelho-preto relaxada  $T$  cuja raiz é vermelha. Se colorirmos a raiz de  $T$  de preto, mas não fizermos nenhuma outra mudança em  $T$ , a árvore resultante é uma árvore vermelho-preto?
- 13.1-4** Suponha que “absorvemos” todo nó vermelho em uma árvore vermelho-preto em seu pai preto, de modo que os filhos do nó vermelho se tornem filhos do pai preto. (Ignore o que acontece com as chaves.) Quais são os

graus possíveis de um nó preto depois que todos os seus filhos vermelhos são absorvidos? O que você pode dizer sobre as profundidades das folhas da árvore resultante?

- 13.1-5** Mostre que o comprimento do mais longo caminho simples de um nó  $x$  em uma árvore vermelho-preto até uma folha descendente é, no máximo, duas vezes o do caminho simples mais curto do nó  $x$  até uma folha descendente.
- 13.1-6** Qual é o maior número possível de nós internos em uma árvore vermelho-preto com altura preta  $k$ ? Qual é o menor número possível?
- 13.1-7** Descreva uma árvore vermelho-preto em  $n$  chaves que permita a maior razão possível entre nós internos vermelhos e nós internos pretos. Qual é essa razão? Qual árvore tem a menor razão possível e qual é essa razão?

## 13.2 ROTACÕES

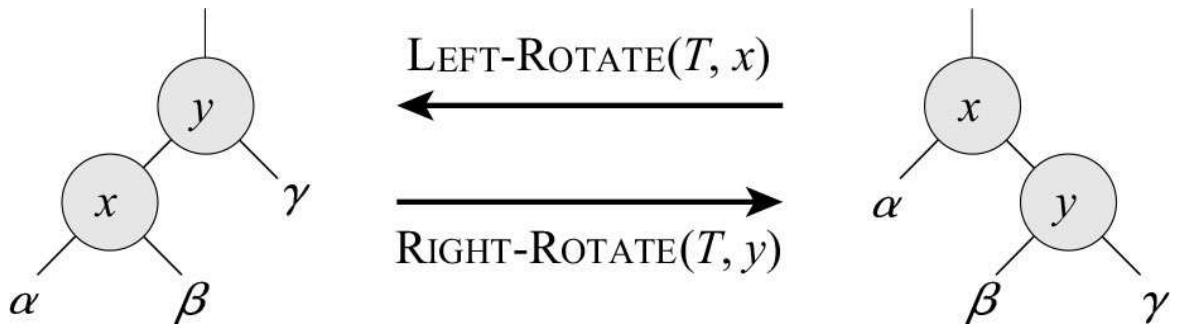
As operações de árvores de busca TREE-INSERT e TREE-DELETE, quando executadas em uma árvore vermelho-preto com  $n$  chaves, demoram o tempo  $O(\lg n)$ . Como elas modificam a árvore, o resultado pode violar as propriedades vermelho-preto enumeradas na Seção 13.1. Para restabelecer essas propriedades, devemos mudar as cores de alguns nós na árvore e também mudar a estrutura de ponteiros.

Mudamos a estrutura de ponteiros por meio de *rotação*, uma operação local em uma árvore de busca que preserva a propriedade de árvore de busca binária. A Figura 13.2 mostra os dois tipos de rotações: rotações para a esquerda e rotações para a direita. Quando fazemos uma rotação para a esquerda em um nó  $x$ , supomos que seu filho à direita  $y$  não é  $T.nil$ ;  $x$  pode ser qualquer nó na árvore cujo filho à direita não é  $T.nil$ . A rotação para a esquerda “pivota” ao redor da ligação de  $x$  para  $y$ . Transforma  $y$  na nova raiz da subárvore, com  $x$  como filho à esquerda de  $y$  e o filho à esquerda de  $y$  como filho à direita de  $x$ .

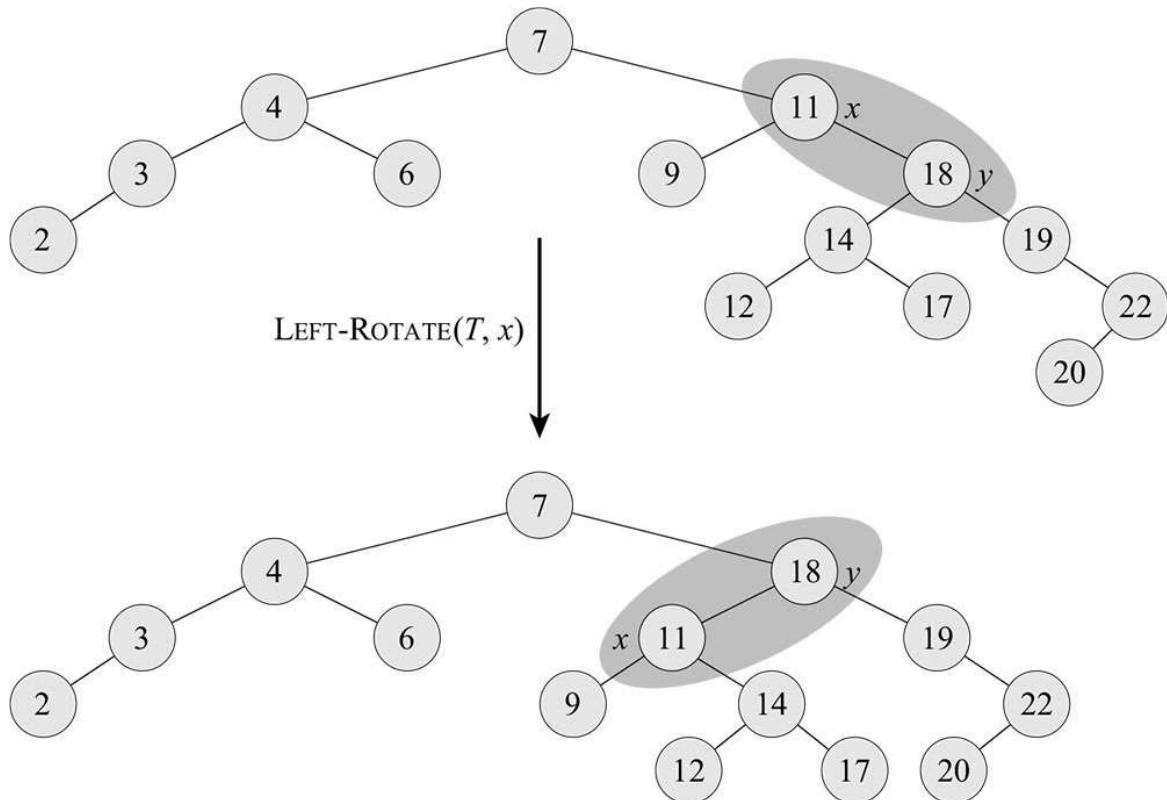
O pseudocódigo para LEFT-ROTATE supõe que  $x.direita \neq T.nil$  e que o pai da raiz é  $T.nil$ .

```
LEFT-ROTATE( $T; x$ )
1  $y = x.direita$            // define  $y$ 
2  $x.direita = y.esquerda$  // transforma a subárvore à esquerda de  $y$  na subárvore à direita de  $x$ 
3 if  $y.esquerda \neq T.nil$ 
4    $y.esquerda.p = x$ 
5  $y.p = x.p$                // liga o pai de  $x$  a  $y$ 
6 if  $x.p == T.nil$ 
7    $T.raiz = y$ 
8 elseif  $x == x.p.esquerda$ 
9    $x.p.esquerda = y$ 
10 else  $x.p.direita = y$ 
11  $y.esquerda = x$          // coloca  $x$  à esquerda de  $y$ 
12  $x.p = y$ 
```

A Figura 13.3 mostra um exemplo de como LEFT-ROTATE modifica uma árvore de busca binária. O código para RIGHT-ROTATE é simétrico. LEFT-ROTATE e RIGHT-ROTATE são executados no tempo  $O(1)$ . Somente ponteiros são alterados por uma rotação; todos os outros atributos em um nó permanecem os mesmos.



**Figura 13.2** As operações de rotação em uma árvore de busca binária. A operação  $\text{LEFT-ROTATE}(T, x)$  transforma a configuração dos dois nós à direita na configuração à esquerda mudando um número constante de ponteiros. A operação inversa  $\text{RIGHT-ROTATE}(T, y)$  transforma a configuração à esquerda na configuração à direita. As letras  $\alpha$ ,  $\beta$  e  $\gamma$  representam subárvores arbitrárias. Uma operação de rotação preserva a propriedade de árvore de busca binária: as chaves em  $\alpha$  precedem  $x.chave$ , que precede as chaves em  $\beta$ , que precedem  $y.chave$ , que precedem as chaves em  $\gamma$ .



**Figura 13.3** Um exemplo de como o procedimento  $\text{LEFT-ROTATE}(T, x)$  modifica uma árvore de busca binária. Os percursos de árvore em inordem da árvore de entrada e a árvore modificada produzem a mesma listagem de valores de chaves.

## Exercícios

**13.2-1** Escreva pseudocódigo para  $\text{RIGHT-ROTATE}$ .

**13.2-2** Demonstre que, em toda árvore de busca binária de  $n$  nós, existem exatamente  $n - 1$  rotações possíveis.

**13.2-3** Sejam  $a$ ,  $b$  e  $c$  nós arbitrários nas subárvores  $\alpha$ ,  $\beta$  e  $\gamma$ , respectivamente, na árvore da direita da Figura 13.2. Como as profundidades de  $a$ ,  $b$  e  $c$  mudam quando é realizada uma rotação para a esquerda no nó  $x$  na

figura?

- 13.2-4** Mostre que qualquer árvore de busca binária arbitrária de  $n$  nós pode ser transformada em qualquer outra árvore de busca binária arbitrária de  $n$  nós por meio de  $O(n)$  rotações. (*Sugestão:* Primeiro, mostre que, no máximo,  $n - 1$  rotações para a direita são suficientes para transformar a árvore em uma cadeia orientada para a direita.)

**13.2-5** ★

Dizemos que uma árvore de busca binária  $T_1$  pode ser *convertida para a direita* na árvore de busca binária  $T_2$  se for possível obter  $T_2$  de  $T_1$  por meio de uma série de chamadas a `RIGHR-ROTATE`. Dê um exemplo de duas árvores  $T_1$  e  $T_2$  tais que  $T_1$  não possa ser convertida para a direita em  $T_2$ . Em seguida, mostre que, se uma árvore  $T_1$  pode ser convertida para a direita em  $T_2$ , ela pode ser convertida para a direita por meio de  $O(n_2)$  chamadas a `RIGHT-ROTATE`.

## 13.3 INSERÇÃO

Podemos inserir um nó em uma árvore vermelho-preto de  $n$  nós no tempo  $O(\lg n)$ . Para tal, usamos uma versão ligeiramente modificada do procedimento `TREE-INSERT` (Seção 12.3) para inserir o nó  $z$  na árvore  $T$  como se ela fosse uma árvore de busca binária comum e depois colorimos  $z$  de vermelho. (O Exercício 13.3-1 pede que você explique por que escolhemos que o nó  $z$  é vermelho, em vez de preto.) Para garantir que as propriedades vermelho-preto serão preservadas, chamamos um procedimento auxiliar `RB-INSERT-FIXUP` para colorir novamente os nós e executar rotações. A chamada `RB-INSERT( $T, z$ )` insere o nó  $z$  — cuja *chave* considera-se já ter sido inserida — na árvore vermelho-preto  $T$ .

```

RB-INSERT( $T, z$ )
1       $y = T.nil$ 
2       $x = T.raiz$ 
3      while  $x \neq T.nil$ 
4           $y = x$ 
5          if  $z.chave < x.chave$ 
6               $x = x.esquerda$ 
7          else  $x = x.direita$ 
8
9           $z.p = y$ 
10         if  $y == T.nil$ 
11              $T.raiz = z$ 
12         elseif  $z.chave < x.chave$ 
13              $y.esquerda = z$ 
14         else  $y.direita = z$ 
15          $z.esquerda = T.nil$ 
16          $z.direita = T.nil$ 
17          $z.cor = \text{RED}$ 
18
19         RB-INSERT-FIXUP( $T, z$ )

```

Há quatro diferenças entre os procedimentos TREE-INSERT e RB-INSERT. Primeiro, todas as instâncias de NIL em TREE-INSERT são substituídas por  $T.nil$ . Em segundo lugar, definimos  $z.esquerda$  e  $z.direita$  como  $T.nil$  nas linhas 14 e 15 de RB-INSERT, a fim de manter a estrutura de árvore adequada. Em terceiro lugar, colorimos  $z$  de vermelho na linha 16. Em quarto lugar, visto que colorir  $z$  de vermelho pode causar uma violação de uma das propriedades vermelho-preto, chamamos RB-INSERT-FIXUP( $T, z$ ) na linha 17 de RB-INSERT para restaurar as propriedades vermelho-preto.

```

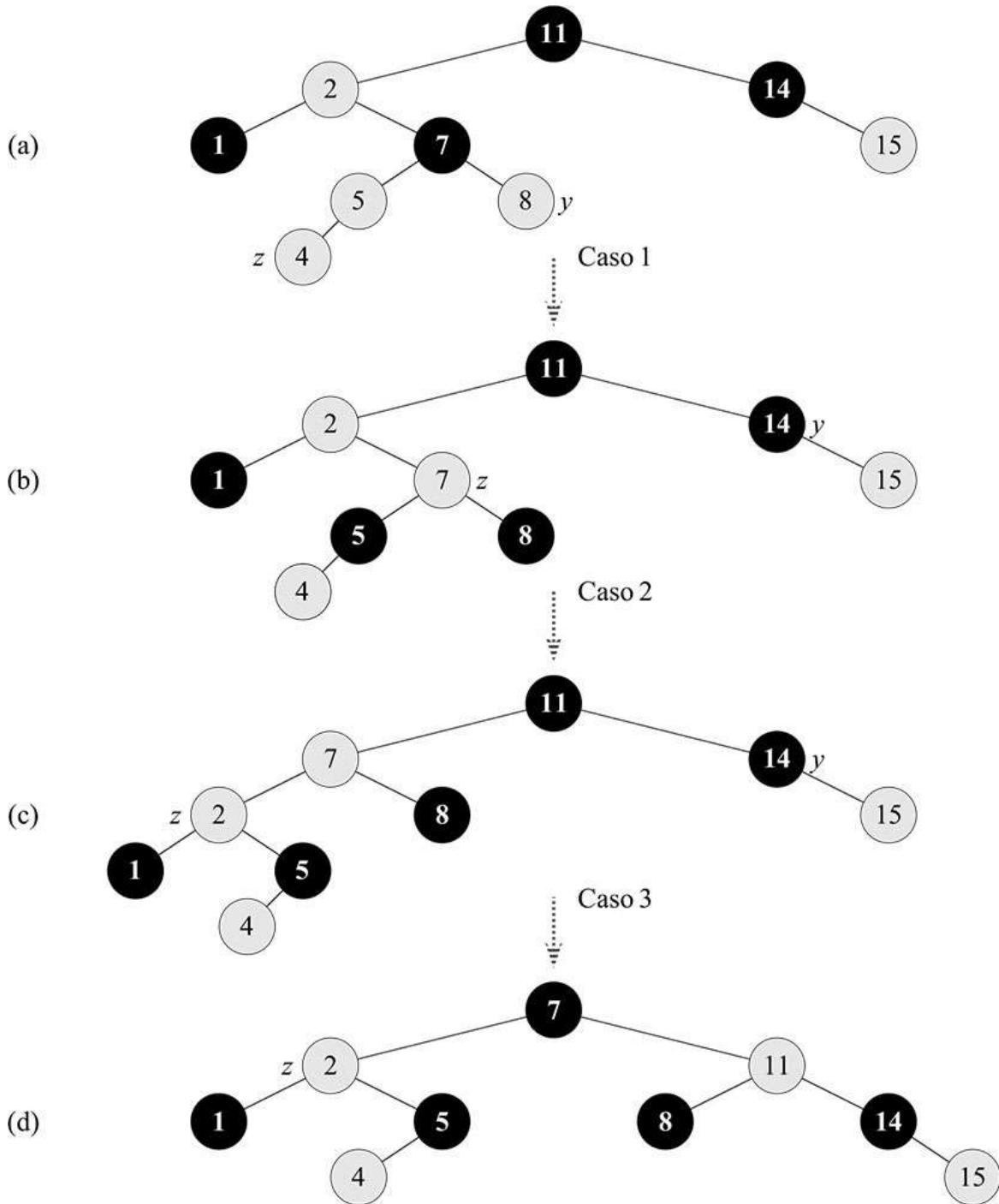
RB-INSERT-FIXUP( $T, z$ )
1      while  $z.p.cor == \text{VERMELHO}$ 
2          if  $z.p == z.p.p.esquerda$ 
3               $y = z.p.p.direita$ 
4              if  $y.cor == \text{VERMELHO}$ 
5                   $z.p.cor = \text{PRETO}$                                 // caso 1
6                   $y.cor = \text{PRETO}$                                 // caso 1
7                   $z.p.p.cor = \text{VERMELHO}$                       // caso 1
8                   $z = z.p.p$                                     // caso 1
9              else if  $z == z.p.direita$ 
10              $z = z.p$                                       // caso 2
11             LEFT-ROTATE( $T, z$ )                          // caso 2
12              $z.p.cor = \text{PRETO}$                                 // caso 3
13              $z.p.p.cor = \text{VERMELHO}$                       // caso 3
14             RIGHT-ROTATE( $T, z.p.p$ )                     // caso 3
15         else (igual à cláusula then
16             com "direita" e "esquerda" trocadas)
17
18          $T.raiz.cor = \text{PRETO}$ 

```

Para entender como RB-INSERT-FIXUP funciona, desmembraremos nosso exame do código em três etapas principais. Primeiro, determinaremos quais violações das propriedades vermelho-preto são introduzidas em RB-INSERT quando o nó  $z$  é inserido e colorido de vermelho. Em segundo lugar, examinaremos a meta global do laço **while** das linhas 1–15. Por fim, exploraremos cada um dos três casos<sup>1</sup> dentro do corpo do laço **while** e veremos como eles cumprem essa meta. A Figura 13.4 mostra como RB-INSERT-FIXUP funciona em uma amostra de árvore vermelho-preto.

Quais das propriedades vermelho-preto podem ser violadas na chamada a RB-INSERT-FIXUP? A propriedade 1 certamente continua válida, bem como a propriedade 3, já que ambos os filhos do nó vermelho recém-inserido são a sentinela  $T.nil$ . A propriedade 5, que diz que o número de nós pretos é igual em todo caminho simples de um dado nó, também é satisfeita porque o nó  $z$  substitui a sentinela (preta), e o nó  $z$  é vermelho com filhos sentinelas. Assim, as únicas propriedades que poderiam ser violadas são a propriedade 2, que exige que a raiz seja preta, e a propriedade 4, que diz que um nó vermelho não pode ter um filho vermelho. Ambas as violações possíveis se devem a  $z$  ser colorido de vermelho. A propriedade 2 é violada se  $z$  é a raiz, e a propriedade 4 é violada se o pai de  $z$  é vermelho. A Figura 13.4(a) mostra uma violação da propriedade 4 após a inserção do nó  $z$ .

---



**Figura 13.4** A operação de RB-INSERT-FIXUP. **(a)** Um nó  $z$  depois da inserção. Como  $z$  e seu pai  $z.p$  são vermelhos, ocorre uma violação da propriedade 4. Visto que o tio  $y$  de  $z$  é vermelho, o caso 1 no código se aplica. Colorimos novamente os nós e movimentamos o ponteiro  $z$  para cima na árvore, resultando na árvore mostrada em **(b)**. Mais uma vez,  $z$  e seu pai são vermelhos, mas o tio  $y$  de  $z$  é preto. Como  $z$  é o filho à direita de  $z.p$ , o caso 2 se aplica. Executamos uma rotação para a esquerda e a árvore resultante é mostrada em **(c)**. Agora,  $z$  é o filho à esquerda de seu pai, e o caso 3 se aplica. Colorindo novamente e executando uma rotação para a direita, é produzida a árvore em **(d)**, que é uma árvore vermelho-preto válida.

O laço **while** nas linhas 1–15 mantém o seguinte invariante de três partes no início de cada iteração do laço:

- O nó  $z$  é vermelho.
- Se  $z.p$  é a raiz, então  $z.p$  é preto.

- c. Se a árvore violar qualquer das propriedades vermelho-preto, ela violará no máximo uma delas, e a violação será da propriedade 2 ou da propriedade 4. Se a árvore violar a propriedade 2 é porque  $z$  é a raiz e é vermelho. Se a árvore violar a propriedade 4 é porque  $z$  e  $z.p$  são vermelhos.

A parte (c), que trata das violações de propriedades vermelho-preto, é mais fundamental para mostrar que RB-INSERT-FIXUP restaura as propriedades vermelho-preto que as partes (a) e (b), que utilizamos no caminho para entender situações no código. Como nos concentraremos no nó  $z$  e nós próximos a ele na árvore, é útil saber pela parte (a) que  $z$  é vermelho. Usaremos a parte (b) para mostrar que o nó  $z.p.p$  existe quando nos referimos a ele nas linhas 2, 3, 7, 8, 13 e 14.

Lembre-se de que precisamos mostrar que um invariante de laço é verdadeiro antes da primeira iteração do laço, que cada iteração mantém o invariante de laço e que o invariante de laço nos dá uma propriedade útil ao término do laço.

Começamos com os argumentos de inicialização e término. Então, à medida que examinarmos com mais detalhes como o corpo do laço funciona, demonstraremos que o laço mantém o invariante em cada iteração. Durante o processo, também demonstraremos que cada iteração do laço tem dois resultados possíveis: o ponteiro  $z$  sobe a árvore ou executamos algumas rotações e o laço termina.

**Inicialização:** Antes da primeira iteração do laço, começamos com uma árvore vermelho-preto sem nenhuma violação e acrescentamos um nó vermelho  $z$ . Mostramos que cada parte do invariante é válida no momento em que RB-INSERT-FIXUP é chamado:

- a. Quando RB-INSERT-FIXUP é chamado,  $z$  é o nó vermelho que foi acrescentado.
- b. Se  $p[z]$  é a raiz, então  $z.p$  começou preto e não mudou antes da chamada de RB--INSERT-FIXUP.
- c. Já vimos que as propriedades 1, 3 e 5 são válidas quando RB-INSERT-FIXUP é chamado. Se a árvore violar a propriedade 2, a raiz vermelha deve ser o nó  $z$  recém-acrescentado, que é o único nó interno na árvore. Como o pai e ambos os filhos de  $z$  são a sentinela, que é preta, a árvore tampouco viola a propriedade 4. Assim, essa violação da propriedade 2 é a única violação de propriedades vermelho-preto na árvore inteira. Se a árvore violar a propriedade 4, como os filhos do nó  $z$  são sentinelas pretas e a árvore não tinha nenhuma outra violação antes de  $z$  ser acrescentado, a violação tem de ser porque  $z$  e  $z.p$  são vermelhos. Além disso, a árvore não viola nenhuma outra propriedade vermelho-preto.

**Término:** Quando o laço termina, é porque  $z.p$  é preto. (Se  $z$  é a raiz, então  $z.p$  é a sentinela  $T.nil$ , que é preta.)

Assim, a árvore não viola a propriedade 4 no término do laço. Pelo invariante de laço, a única propriedade que poderia deixar de ser válida é a propriedade 2. A linha 16 restaura também essa propriedade, de modo que, quando RB-INSERT-FIXUP termina, todas as propriedades vermelho-preto são válidas.

**Manutenção:** Na realidade, precisamos considerar seis casos no laço **while**, mas três deles são simétricos aos outros três, dependendo de a linha 2 determinar que o pai  $z.p$  de  $z$  é um filho à esquerda ou um filho à direita do avô  $z.p.p$  de  $z$ . Damos o código somente para a situação na qual  $z.p$  é um filho à esquerda. O nó  $z.p.p$  existe, já que, pela parte (b) do invariante de laço, se  $z.p$  é a raiz, então  $z.p$  é preto. Visto que entramos em uma iteração de laço somente se  $z.p$  é vermelho, sabemos que  $z.p$  não pode ser a raiz. Consequentemente,  $z.p.p$  existe.

Distinguimos o caso 1 dos casos 2 e 3 pela cor do irmão do pai de  $z$ , ou “tio”. A linha 3 faz  $y$  apontar para o tio  $z.p.p.direita$  de  $z$ , e a linha 4 testa a cor de  $y$ . Se  $y$  é vermelho, então executamos o caso 1. Do contrário, o controle passa para os casos 2 e 3. Em todos os três casos, o avô  $z.p.p$  de  $z$  é preto, já que seu pai  $z.p$  é vermelho, e a propriedade 3 é violada apenas entre  $z$  e  $z.p$ .

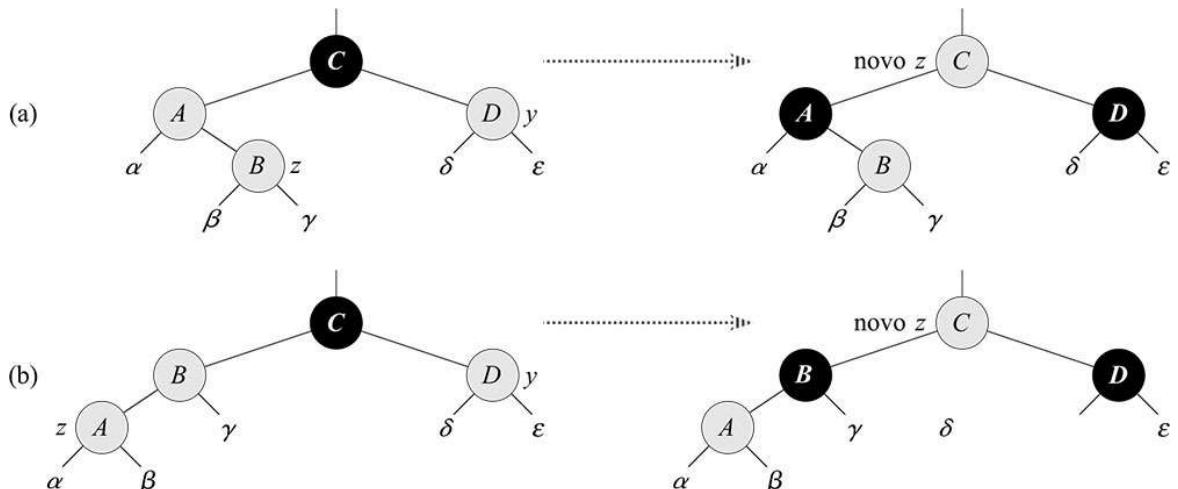
### Caso 1: o tio de $y$ de $z$ é vermelho

A Figura 13.5 mostra a situação para o caso 1 (linhas 5–8), que ocorre quando  $z.p$  e  $y$  são vermelhos. Como  $z.p.p$  é preto, podemos colorir  $z.p$  e  $y$  de preto, o que corrige o problema de  $z$  e  $z.p$  serem vermelhos, e podemos colorir  $z.p.p$  de vermelho, mantendo assim a propriedade 5. Então repetimos o laço **while** com  $z.p.p$  como o novo nó  $z$ . O ponteiro  $z$  sobe dois níveis na árvore. Agora mostramos que o caso 1 mantém o invariante de laço no início da próxima iteração. Usamos  $z$  para denotar o nó  $z$  na iteração atual, e  $z' = z.p.p$  para denotar o nó que será denominado  $z$  no teste da linha 1 na iteração seguinte.

- Como essa iteração colore  $z.p.p$  de vermelho, o nó  $z'$  é vermelho no início da próxima iteração.
- O nó  $z'.p$  é  $z.p.p.p$  nessa iteração, e a cor desse nó não se altera. Se esse nó é a raiz, ele era preto antes dessa iteração e permanece preto no início da próxima iteração.
- Já mostramos que o caso 1 mantém a propriedade 5 e não introduz uma violação das propriedades 1 ou 3.

Se o nó  $z'$  é a raiz no início da próxima iteração, então o caso 1 corrigiu a única violação da propriedade 4 nessa iteração. Como  $z'$  é vermelho e é a raiz, a propriedade 2 passa a ser a única violada, e essa violação se deve a  $z'$ .

Se o nó  $z'$  não é a raiz no início da próxima iteração, então o caso 1 não criou uma violação da propriedade 2. O caso 1 corrigiu a única violação da propriedade 4 que existia no início dessa iteração. Então, transformou  $z'$  em vermelho e deixou  $z'.p$  como estava. Se  $z'.p$  era preto, não há nenhuma violação da propriedade 4. Se  $z'.p$  era vermelho, colorir  $z'$  de vermelho criou uma violação da propriedade 4 entre  $z'$  e  $z'.p$ .



**Figura 13.5** O caso 1 do procedimento RB-INSERT-FIXUP. A propriedade 4 é violada, já que  $z$  e seu pai  $z.p$  são vermelhos. A mesma ação é adotada se (a)  $z$  é um filho à direita ou (b)  $z$  é um filho à esquerda. Cada uma das subárvores,  $\alpha$ ,  $\beta$ ,  $g$ ,  $d$  e  $e$  tem uma raiz preta e cada uma tem a mesma altura preta. O código para o caso 1 muda as cores de alguns nós, preservando a propriedade 5: todos os caminhos simples descendentes de um nó até uma folha têm o mesmo número de pretos. O laço while continua com o avô  $z.p.p$  do nó  $z$  como o novo  $z$ . Qualquer violação da propriedade 4 só pode ocorrer agora entre o novo  $z$ , que é vermelho, e seu pai, que também é vermelho.

### Caso 2: o tio $y$ de $z$ é preto e $z$ é um filho à direita

### Caso 3: o tio $y$ de $z$ é preto e $z$ é um filho à esquerda

Nos casos 2 e 3, a cor do tio  $y$  de  $z$  é preta. Distinguimos os dois casos conforme  $z$  seja um filho à direita ou à esquerda de  $z.p$ . As linhas 10 e 11 constituem o caso 2, que é mostrado na Figura 13.6, juntamente com o caso 3. No caso 2, o nó  $z$  é um filho à direita de seu pai. Usamos imediatamente uma rotação para a esquerda para transformar a situação no

caso 3 (linhas 12–14), na qual o nó  $z$  é um filho à esquerda. Como  $z$  e  $z.p$  são vermelhos, a rotação não afeta a altura preta dos nós nem a propriedade 5. Quer entremos no caso 3 diretamente ou por meio do caso 2, o tio  $y$  de  $z$  é preto, já que, do contrário, teríamos executado o caso 1. Além disso, o nó  $z.p.p$  existe, visto que demonstramos que esse nó existia no momento em que as linhas 2 e 3 foram executadas e, após  $z$  subir um nível na linha 10 e depois descer um nível na linha 11, a identidade de  $z.p.p$  permanece inalterada. No caso 3, executamos algumas mudanças de cores e uma rotação para a direita, o que preserva a propriedade 5; em seguida, visto que não temos mais dois nós vermelhos em uma linha, encerramos. O corpo do laço **while** não é executado outra vez, já que agora  $z.p$  é preto.

Agora, mostramos que os casos 2 e 3 mantêm o invariante de laço. (Como acabamos de demonstrar,  $z.p$  será preto no próximo teste na linha 1 e o corpo do laço não será executado novamente.)

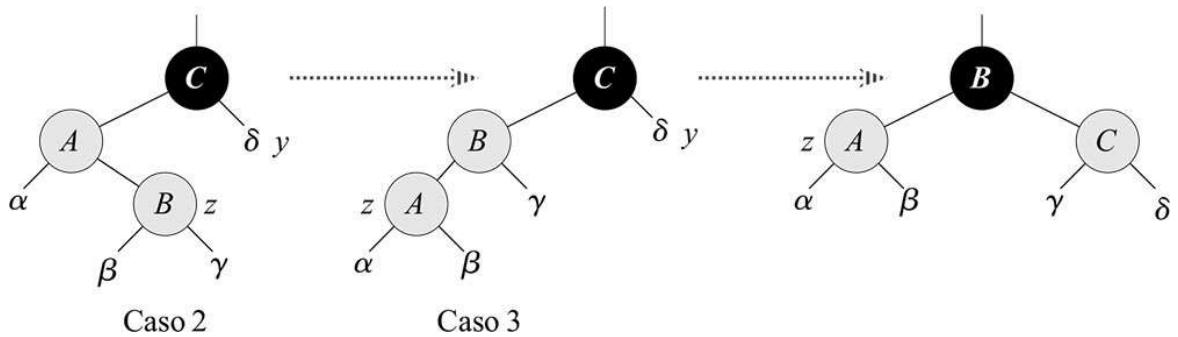
- O caso 2 faz  $z$  apontar para  $z.p$ , que é vermelho. Nenhuma mudança adicional em  $z$  ou em sua cor ocorre nos casos 2 e 3.
- O caso 3 torna  $z.p$  preto, de modo que, se  $z.p$  é a raiz no início da próxima iteração, ele é preto.
- Como ocorre no caso de 1, as propriedades 1, 3 e 5 são mantidas nos casos 2 e 3.

Visto que o nó  $z$  não é a raiz nos casos 2 e 3, sabemos que não há nenhuma violação da propriedade 2. Os casos 2 e 3 não introduzem uma violação da propriedade 2, já que o único nó que se tornou vermelho torna-se um filho de um nó preto pela rotação no caso 3.

Os casos 2 e 3 corrigem a única violação da propriedade 4 e não introduzem outra violação. Mostrando que cada iteração do laço mantém o invariante, também mostramos que RB--INSERT-FIXUP restaura corretamente as propriedades vermelho-preto.

## Análise

Qual é o tempo de execução de RB-INSERT? Visto que a altura de uma árvore vermelho-preto em  $n$  nós é  $O(\lg n)$ , as linhas 1–16 de RB-INSERT levam o tempo  $O(\lg n)$ . Em RB-INSERT-FIXUP, o laço **while** só é repetido se o caso 1 ocorrer, e então o ponteiro  $z$  sobe dois níveis na árvore.



**Figura 13.6** Casos 2 e 3 do procedimento RB-INSERT-FIXUP. Como no caso 1, a propriedade 4 é violada no caso 2 ou no caso 3 porque  $z$  e seu pai  $z.p$  são vermelhos. Cada uma das subárvores,  $\alpha$ ,  $\beta$ ,  $\gamma$  e  $\delta$  tem uma raiz preta ( $\alpha$ ,  $\beta$  e  $\gamma$  pela propriedade 4, e  $\delta$  porque, caso contrário, estaríamos no caso 1) e cada uma tem a mesma altura preta. Transformamos o caso 2 no caso 3 por uma rotação para a esquerda, o que preserva a propriedade 5: todos os caminhos simples descendentes de um nó até uma folha têm o mesmo número de pretos. O caso 3 provoca algumas mudanças de cores e uma rotação para a direita, o que também preserva a propriedade 5. Em seguida, o laço **while** termina porque a propriedade 4 é satisfeita: não há mais dois nós vermelhos em seguida.

Portanto, o número total de vezes que o laço **while** pode ser executado é  $O(\lg n)$ . Assim, RB-INSERT demora um tempo total  $O(\lg n)$ . Além disso, ele nunca executa mais de duas rotações, já que o laço **while** termina se o caso 2 ou o caso 3 for executado.

## Exercícios

---

- 13.3-1** Na linha 16 de RB-INSERT, atribuímos o nó  $z$  recém-inserido com vermelho. Note que, se tivéssemos optado por atribuir  $z$  com preto, a propriedade 4 de uma árvore vermelho-preto não seria violada. Por que não optamos por definir  $z$  como preto?
- 13.3-2** Mostre as árvores vermelho-preto que resultam após a inserção sucessiva das chaves 41, 38, 31, 12, 19, 8 em uma árvore vermelho-preto inicialmente vazia.
- 13.3-3** Suponha que a altura preta de cada uma das subárvores  $\alpha, \beta, \gamma, d$ , nas Figuras 13.5 e 13.6 seja  $k$ . Identifique cada nó em cada figura com sua altura preta para verificar se a transformação indicada preserva a propriedade 5.
- 13.3-4** O professor Teach está preocupado que RB-INSERT-FIXUP possa atribuir  $T.nil.cor$  como VERMELHO, caso em que o teste da linha 1 não faria o laço terminar quando  $z$  fosse a raiz. Mostre que a preocupação do professor é infundada, demonstrando que RB-INSERT-FIXUP nunca atribui  $T.nil.cor$  com VERMELHO.
- 13.3-5** Considere uma árvore vermelho-preto formada pela inserção de  $n$  nós com RB-INSERT. Mostre que, se  $n > 1$ , a árvore tem, no mínimo, um nó vermelho.
- 13.3-6** Sugira como implementar RB-INSERT de maneira eficiente se a representação para árvores vermelho-preto não incluir nenhum armazenamento para ponteiros superiores.

## 13.4 ELIMINAÇÃO

Como as outras operações básicas em uma árvore vermelho-preto de  $n$  nós, a eliminação de um nó demora o tempo  $O(\lg n)$ . Eliminar um nó de uma árvore vermelho-preto é um pouco mais complicado que inserir um nó.

O procedimento para eliminar um nó de uma árvore vermelho-preto é baseado no procedimento RB-DELETE (Seção 12.3). Primeiro, precisamos customizar a sub-rotina TRANSPLANT que TREE-DELETE chama, de modo que ela se aplique a uma árvore vermelho-preto:

```
RB-TRANSPLANT( $T, u, v$ )
1   if  $u.p == T.nil$ 
2        $T.raiz = v$ 
3   elseif  $u == u.p.esquerda$ 
4        $u.p.esquerda = v$ 
5   else  $u.p.direita = v$ 
6    $v.p = u.p$ 
```

Há duas diferenças entre o procedimento RB-TRANSPLANT e o procedimento TRANSPLANT. A primeira é que a linha 1 referencia a sentinela  $T.nil$  em vez de  $\text{NIL}$ . A segunda é que a atribuição a  $.p$  na linha 6 ocorre incondicionalmente: podemos atribuir a  $v.p$  mesmo que aponte para a sentinela. De fato, exploraremos a capacidade de atribuir a  $.p$  quando  $= T.nil$ .

O procedimento RB-DELETE é como o procedimento TREE-DELETE, porém com linhas adicionais de pseudocódigo. Algumas dessas linhas adicionais rastreiam um nó  $y$  que poderia causar violações das propriedades vermelho-preto. Quando queremos eliminar o nó  $z$  e  $z$  tem menos do que dois filhos,  $z$  é removido da árvore e queremos que  $y$  seja  $z$ . Quando  $z$  tem dois filhos,  $y$  deve ser o sucessor de  $z$ , e  $y$  passa para a posição de  $z$  na árvore. Também lembramos a cor de  $y$  antes de ele ser eliminado da árvore ou passar para dentro dela, e rastreamos o nó  $x$  que passa para a posição original de  $y$  na árvore porque o nó  $x$  também poderia causar violações das propriedades vermelho-preto. Após

eliminar o nó  $z$ , RB-DELETE chama um procedimento auxiliar RB-DELETE-FIXUP, que muda as cores e executa rotações para restaurar as propriedades vermelho-preto.

```

RB-DELETE( $T, z$ )
1    $y = z$ 
2    $y\text{-cor-original} = y.\text{cor}$ 
3   if  $z.\text{esquerda} == T.\text{nil}$ 
4      $x = z.\text{direita}$ 
5     RB-TRANSPLANT( $T, z, z.\text{direita}$ )
6   elseif  $z.\text{direita} == T.\text{nil}$ 
7      $x = z.\text{esquerda}$ 
8     RB-TRANSPLANT( $T, z, z.\text{esquerda}$ )
9   else  $y = \text{TREE-MINIMUM}(z.\text{direita})$ 
10     $y\text{-cor-original} = y.\text{cor}$ 
11     $x = y.\text{direita}$ 
12    if  $y.p == z$ 
13       $x.p = y$ 
14    else RB-TRANSPLANT( $T, y, y.\text{direita}$ )
15       $y.\text{direita} = z.\text{direita}$ 
16       $y.\text{direita}.p = y$ 
17      RB-TRANSPLANT( $T, z, y$ )
18       $y.\text{esquerda} = z.\text{esquerda}$ 
19       $y.\text{esquerda}.p = y$ 
20       $y.\text{cor} = z.\text{cor}$ 
21    if  $y\text{-cor-original} == \text{PRETO}$ 
22      RB-DELETE-FIXUP( $T, x$ )

```

Embora RB-DELETE contenha quase duas vezes o número de linhas de pseudocódigo de TREE-DELETE, os dois procedimentos têm a mesma estrutura básica. Podemos encontrar cada linha de TREE-DELETE dentro de RB-DELETE (se substituirmos  $T.\text{nil}$  por NIL e as chamadas a RB-TRANSPLANT por chamadas a TRANSPLANT) se executado sob as mesmas condições.

Apresentamos a seguir, as outras diferenças entre os dois procedimentos:

- Mantemos o nó  $y$  como o nó que é retirado da árvore ou que é passado para dentro dela. A linha 1 faz  $y$  apontar para o nó  $z$  quando  $z$  tiver menos que dois filhos e, portanto, é removido. Quando  $z$  tem dois filhos, a linha 9 faz  $y$  apontar para o sucessor de  $z$  exatamente como em TREE-DELETE, e  $y$  passa para a posição de  $z$  na árvore.
- Como a cor do nó  $y$  pode mudar, a variável  $y\text{-cor-original}$  armazena a cor de  $y$  antes de ocorrer qualquer mudança. As linhas 2 e 10 definem essa variável imediatamente após atribuições a  $y$ . Quando  $z$  tem dois filhos, então  $y \neq z$  e o nó  $y$  passa para a posição original do nó  $z$  na árvore vermelho-preto; a linha 20 dá a  $y$  a mesma cor de  $z$ . Precisamos salvar a cor original de  $y$  para testá-la no final de RB-DELETE; se o nó era preto, remover ou mover  $y$  poderá causar violações das propriedades vermelho-preto.
- Como discutimos, rastreamos o nó  $x$  que passa para a posição original do nó  $y$ . As atribuições nas linhas 4, 7 e 11 fazem  $x$  apontar para o único filho de  $y$  ou, se  $y$  não tiver filhos, para a sentinela  $T.\text{nil}$ . (Lembre-se de que dissemos, na Seção 12.3, que  $y$  não tem nenhum filho à esquerda.)
- Visto que o nó  $x$  passa para a posição original de  $y$ , o atributo  $x.p$  é sempre definido para apontar para a posição original do pai de  $y$  na árvore, mesmo que  $x$  seja, de fato, a sentinela  $T.\text{nil}$ . A menos que  $z$  seja o pai original de  $y$  (o que ocorre somente quando  $z$  tiver dois filhos e seu sucessor  $y$  for o filho à direita de  $z$ ), a atribuição a  $x.p$  ocorre na linha 6 de RB-TRANSPLANT.

(Observe que, quando RB-TRANSPLANT é chamado nas linhas 5, 8 ou 14, o terceiro parâmetro passado é o mesmo que  $x$ .)

Entretanto, quando o pai original de  $y$  é  $z$ , não queremos que  $x.p$  aponte para o pai original de  $y$ , visto que estamos eliminando aquele nó da árvore. Como o nó  $y$  subirá para ocupar a posição de  $z$  na árvore, atribuir  $y$  a  $x.p$  na linha 13 faz com que  $x.p$  aponte para a posição original do pai de  $y$ , mesmo que  $x = T.nil$ .

- Por fim, se o nó  $y$  era preto, pode ser que tenhamos introduzido uma ou mais violações das propriedades vermelho-preto e, por isso, chamamos RB-DELETE-FIXUP na linha 22 para restaurar as propriedades vermelho-preto. Se  $y$  era vermelho, as propriedades vermelho--preto ainda são válidas quando  $y$  é eliminado ou movido, pelas seguintes razões:
  1. Nenhuma altura preta na árvore mudou.
  2. Nenhum par de nós vermelhos tornou-se adjacente. Como  $y$  toma o lugar de  $z$  na árvore, juntamente com a cor de  $z$ , não podemos ter dois nós vermelhos adjacentes na nova posição de  $y$  na árvore. Além disso, se  $y$  não era o filho à direita de  $z$ , então  $x$ , o filho à direita original de  $y$ , substitui  $y$  na árvore. Se  $y$  é vermelho, então  $x$  deve ser preto; portanto, substituir  $y$  por  $x$  não pode fazer com que dois nós vermelhos se tornem adjacentes.
  3. Visto que  $y$  não poderia ter sido a raiz se fosse vermelho, a raiz permanece preta.

Se o nó  $y$  era preto, poderão surgir três problemas, que a chamada de RB-DELETE-FIXUP remediará. Primeiro, se  $y$  era a raiz e um filho vermelho de  $y$  se torna a nova raiz, violamos a propriedade 2. Segundo, se  $x$  e  $y.p$  (que agora também é  $x.p$ ) eram vermelhos, então violamos a propriedade 4. Terceiro, mover  $y$  pela árvore faz com que qualquer caminho simples que continha  $y$  anteriormente tenha um nó preto a menos. Assim, a propriedade 5 agora é violada por qualquer ancestral de  $y$  na árvore. Podemos corrigir a violação da propriedade 5 dizendo que o nó  $x$ , que agora ocupa a posição original de  $y$ , tem um preto “extra”. Isto é, se somarmos 1 à contagem de nós pretos em qualquer caminho simples que contenha  $x$ , então, por essa interpretação, a propriedade 5 se mantém válida. Quando extraímos ou movimentamos o nó preto  $y$ , “impomos” sua negritude ao nó  $x$ . O problema é que agora o nó  $x$  não é nem vermelho nem preto, o que viola a propriedade 1. Em vez disso, o nó  $x$  é “duplamente preto” ou “vermelho e preto” e contribui com 2 ou 1, respectivamente, para a contagem de nós pretos em caminhos simples que contêm  $x$ . O atributo *cor* de  $x$  ainda será VERMELHO (se  $x$  é vermelho e preto) ou PRETO (se  $x$  é duplamente preto). Em outras palavras, a consequência desse preto extra em um nó é que  $x$  apontará para o nó em vez de para o atributo *cor*.

Agora podemos ver o procedimento RB-DELETE-FIXUP e examinar como ele devolve as propriedades vermelho-preto à árvore de busca.

```

RB-DELETE-FIXUP( $T; x$ )
1   while  $x \neq T.raiz$  and  $x.cor == PRETO$ 
2     if  $x == x.p.esquerda$ 
3        $w = x.p.direita$ 
4       if  $w.cor == VERMELHO$ 
5          $w.cor = PRETO$                                 // caso 1
6          $x.p.cor = VERMELHO$                          // caso 1
7         LEFT-ROTATE( $T, x.p$ )                      // caso 1
8          $w = x.p.direita$                            // caso 1
9       if  $w.esquerda.cor == PRETO$  and  $w.direita.cor == PRETO$ 
10       $w.cor = VERMELHO$                             // caso 2
11       $x = x.p$                                  // caso 2
12    else if  $w.direita.cor == PRETO$ 
13       $w.esquerda.cor = PRETO$                      // caso 3
14       $w.cor = VERMELHO$                           // caso 3
15      RIGHT-ROTATE( $T, w$ )                        // caso 3
16       $w = x.p.direita$                          // caso 3

```

O procedimento RB-DELETE-FIXUP restaura as propriedades 1, 2 e 4. Os Exercícios 13.4-1 e 13.4-2 pedem que você mostre que o procedimento restaura as propriedades 2 e 4 e, assim, no restante desta seção focalizaremos a propriedade 1. O objetivo do laço **while** nas linhas 1–22 é mover o preto extra para cima na árvore até

1.  $x$  apontar para um nó vermelho e preto, caso em que colorimos  $x$  (isoladamente) de preto na linha 23;
2.  $x$  apontar para a raiz, caso em que simplesmente “removemos” o preto extra; ou
3. que, executadas as operações adequadas de rotações e novas colorações, saímos do laço.

Dentro do laço **while**,  $x$  sempre aponta para um nó não raiz duplamente preto. Determinamos na linha 2 se  $x$  é um filho à esquerda ou um filho à direita de seu pai  $x.p$ . (Já fornecemos o código para a situação na qual  $x$  é um filho à esquerda; a situação na qual  $x$  é um filho à direita — linha 22 — é simétrica.) Mantemos um ponteiro  $w$  para o irmão de  $x$ . Visto que o nó  $x$  é duplamente preto, o nó  $w$  não pode ser  $T.nil$  porque, caso contrário, o número de pretos no caminho simples de  $x.p$  até a folha  $w$  (simplesmente preta) seria menor que o número no caminho simples de  $x.p$  até  $x$ .

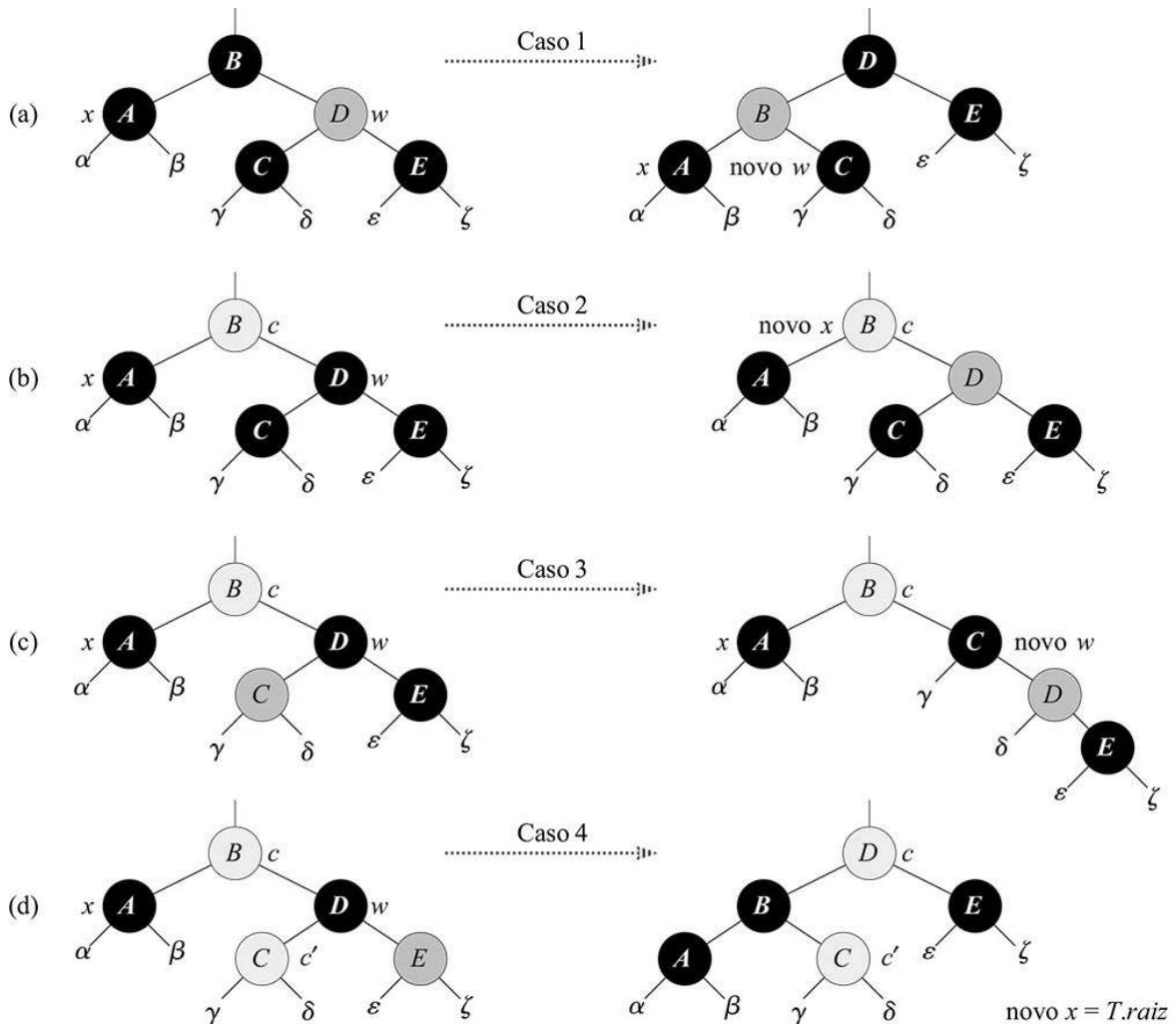
Os quatro casos<sup>2</sup> no código aparecem na Figura 13.7. Antes de examinar cada caso em detalhes, vamos ver, de um modo mais geral, como podemos comprovar que, em cada um dos casos, a transformação preserva a propriedade 5. A ideia-chave é que, em cada caso, a transformação aplicada preserva o número de nós pretos (incluindo o preto extra de  $x$ ) da raiz da subárvore (inclusive) mostrada até cada uma das subárvores  $\alpha, \beta, \dots$ . Assim, se a propriedade 5 é válida antes da transformação, continua a ser válida depois dela. Por exemplo, na Figura 13.7(a), que ilustra o caso 1, o número de nós pretos da raiz até a subárvore  $\alpha$  ou  $\beta$  é 3, antes e também depois da transformação. (Mais uma vez, lembre-se de que o nó  $x$  adiciona um preto extra.) De modo semelhante, o número de nós pretos da raiz até qualquer das subárvores  $\gamma, d, e$  é 2, antes e também depois da transformação. Na Figura 13.7(b), a contagem deve envolver o valor  $c$  do atributo *cor* da raiz da subárvore mostrada, que pode ser VERMELHO ou PRETO. Se definirmos contador(VERMELHO) = 0 e contador(PRETO) = 1, o número de nós pretos da raiz até  $\alpha$  é 2 + contador( $c$ ), antes e também depois da transformação. Nesse caso, após a transformação, o novo nó  $x$  tem o atributo *cor*  $c$ , mas na realidade é vermelho e preto (se  $c$  = VERMELHO) ou duplamente preto (se  $c$  = PRETO). Os outros casos podem ser verificados de maneira semelhante (veja o Exercício 13.4-5.)

### Caso 1: o irmão $w$ de $x$ é vermelho

O caso 1 (linhas 5–8 de RB-DELETE-FIXUP e Figura 13.7(a)) ocorre quando o nó  $w$ , o irmão do nó  $x$ , é vermelho. Visto que  $w$  deve ter filhos pretos, podemos trocar as cores de  $w$  e  $x.p$  e depois executar uma rotação para a esquerda em  $x.p$  sem violar qualquer das propriedades vermelho-preto. O novo irmão de  $x$ , que é um dos filhos de  $w$  antes da rotação, agora é preto e, assim, convertemos o caso 1 no caso 2, 3 ou 4.

Os casos 2, 3 e 4 ocorrem quando o nó  $w$  é preto; eles são distinguidos pelas cores dos filhos de  $w$ .





**Figura 13.7** Os casos no laço **while** do procedimento RB-DELETE-fixup. Nós em preto têm atributos *cor* PRETO, nós sombreados em tom mais escuro têm atributos *cor* VERMELHO e nós sombreados em tom mais claro têm atributos *cor* representados por *c* e *c'*, que podem ser VERMELHO ou PRETO. As letras *a*, *b*, ..., representam subárvore arbitrárias. Cada caso transforma a configuração à esquerda na configuração à direita mudando algumas cores e/ou executando uma rotação. Qualquer nó apontado por *x* tem um preto extra e é duplamente preto ou vermelho e preto. Somente o caso 2 faz o laço se repetir. (a) O caso 1 é transformado no caso 2, 3 ou 4 trocando as cores dos nós *B* e *D* e executando uma rotação para a esquerda. (b) No caso 2, o preto extra representado pelo ponteiro *x* é deslocado para cima na árvore colorindo o nó *D* de vermelho e ajustando *x* para apontar para o nó *B*. Se entrarmos no caso 2 por meio do caso 1, o laço **while** termina, já que o novo nó *x* é vermelho e preto, e portanto o valor *c* de seu atributo *cor* é VERMELHO. (c) O caso 3 é transformado no caso 4 trocando as cores dos nós *C* e *D* e executando uma rotação para a direita. (d) O caso 4 remove o preto extra representado por *x* mudando algumas cores e executando uma rotação para a esquerda (sem violar as propriedades vermelho-preto) e, então, o laço termina.

### Caso 2: o irmão *w* de *x* é preto e os filhos de *w* são pretos

No caso 2 (linhas 10–11 de RB-DELETE-FIXUP e Figura 13.7(b)), os filhos de *w* são pretos. Visto que *w* também é preto, retiramos um preto de *x* e também de *w*, deixando *x* com apenas um preto e deixando *w* vermelho. Para compensar a remoção de um preto de *x* e de *w*, gostaríamos de adicionar um preto extra a *x.p*, que era originalmente vermelho ou preto. Fazemos isso repetindo o laço **while** com *x.p* como o novo nó *x*. Observe que, se entrarmos no caso 2 por meio do caso 1, o novo nó *x* será vermelho e preto, já que o *x.p* original era vermelho. Consequentemente, o valor *c* do atributo *cor* do novo nó *x* é VERMELHO, e o laço termina quando testa a condição de laço. Então colorimos o novo nó *x* de preto (simplesmente) na linha 23.

### Caso 3: o irmão *W* de *X* é preto, o filho à esquerda de *W* é vermelho e o filho à direita de *W* é preto

O caso 3 (linhas 13–16 e Figura 13.7(c)) ocorre quando  $w$  é preto, seu filho à esquerda é vermelho e seu filho à direita é preto. Podemos permutar as cores de  $w$  e de seu filho à esquerda  $w.esquerda$  e então executar uma rotação para a direita em  $w$  sem violar qualquer das propriedades vermelho-preto. O novo irmão  $w$  de  $x$  é agora um nó preto com um filho à direita vermelho e, assim, transformamos o caso 3 no caso 4.

#### Caso 4: o irmão $w$ de $x$ é preto e o filho à direita de $w$ é vermelho

O caso 4 (linhas 17–21 e Figura 13.7(d)) ocorre quando o irmão  $w$  do nó  $x$  é preto e o filho à direita de  $w$  é vermelho. Fazendo algumas mudanças de cores e executando uma rotação para a esquerda em  $x.p$ , podemos remover o preto extra em  $x$ , tornando-o unicamente preto, sem violar qualquer das propriedades vermelho-preto. Definir  $x$  como a raiz faz o laço **while** terminar ao testar a condição de laço.

## Análise

Qual é o tempo de execução de RB-DELETE? Visto que a altura de uma árvore vermelho-preto de  $n$  nós é  $O(\lg n)$ , o custo total do procedimento sem a chamada a RB-DELETE-FIXUP demora o tempo  $O(\lg n)$ . Dentro de RB-DELETE-FIXUP, cada um dos casos 1, 3 e 4 leva ao término depois de executar um número constante de mudanças de cores e no máximo três rotações. O caso 2 é o único no qual o laço **while** pode ser repetido, e então o ponteiro  $x$  se move para cima na árvore no máximo  $O(\lg n)$  vezes sem executar nenhuma rotação. Assim, o procedimento RB-DELETE-FIXUP demora o tempo  $O(\lg n)$  e executa no máximo três rotações e, portanto, o tempo global para RB-DELETE também é  $O(\lg n)$ .

## Exercícios

---

- 13.4-1** Mostre que, após a execução de RB-DELETE-FIXUP, a raiz da árvore tem de ser preta.
- 13.4-2** Mostre que, se  $x$  e  $x.p$  são vermelhos em RB-DELETE, então a propriedade 4 é restabelecida pela chamada a RB-DELETE-FIXUP( $T, x$ ).
- 13.4-3** No Exercício 13.3-2, você determinou a árvore vermelho-preto que resulta da inserção sucessiva das chaves 41, 38, 31, 12, 19, 8 em uma árvore inicialmente vazia. Agora, mostre as árvores vermelho-preto que resultam da eliminação sucessiva das chaves na ordem 8, 12, 19, 31, 38, 41.
- 13.4-4** Em quais linhas do código de RB-DELETE-FIXUP poderíamos examinar ou modificar a sentinela  $T.nil$ ?
- 13.4-5** Em cada um dos casos da Figura 13.7, dê a contagem de nós pretos da raiz da subárvore mostrada até cada uma das subárvores  $\alpha, \beta, \dots, \gamma$ , e confirme que cada contagem permanece a mesma depois da transformação. Quando um nó tiver um atributo  $cor\ c$  ou  $c'$ , use a notação  $contagem(c)$  ou  $contagem(c')$  simbolicamente em sua contagem.
- 13.4-6** Os professores Skelton e Baron estão preocupados porque, no início do caso 1 de RB-DELETE-FIXUP, o nó  $x.p$  poderia não ser preto. Se os professores estão corretos, as linhas 5–6 estão erradas. Mostre que  $x.p$  deve ser preto no início do caso 1 e, portanto, os professores não precisam se preocupar.
- 13.4-7** Suponha que um nó  $x$  seja inserido em uma árvore vermelho-preto com RB-INSERT e então imediatamente eliminado com RB-DELETE. A árvore vermelho-preto resultante é igual à árvore vermelho-preto inicial? Justifique sua resposta.