Capítulo 13

Árvores vermelho-preto

O Capítulo 12 mostrou que uma árvore de pesquisa binária de altura h pode implementar quaisquer 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 pesquisa é pequena; porém, se a altura da árvore é grande, o desempenho dessas operações pode não ser melhor do que seria no caso de uma lista ligada. As árvores vermelho-preto constituem um entre muitos esquemas de árvores de pesquisa que são "balanceadas" com o objetivo de garantir que as 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 pesquisa binária com um bit extra de armazenamento por nó: sua ***cor***, que pode ser VERMELHO ou PRETO. Restringindo o modo como os nós podem ser coloridos em qualquer caminho desde a raiz até uma folha, as árvores vermelho-preto asseguram que nenhum desses caminhos será maior que duas vezes o comprimento de qualquer outro, de forma que a árvore é aproximadamente ***balanceada***.

Cada nó da árvore contém agora os campos *cor*, *chave*, *esquerda*, *direita* e p. Se um filho ou o pai de um nó não existir, o campo do ponteiro correspondente do nó conterá o valor NIL. Trataremos esses valores NIL como ponteiros para nós externos (folhas) da árvore de pesquisa binária e, portanto, os nós normais que conduzem chaves serão tratados como nós internos da árvore.

Uma árvore de pesquisa binária é uma árvore vermelho-preto se satisfaz às 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 ambos os seus filhos são pretos.
5. Para cada nó, todos os caminhos desde um nó até as 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 uso de condições limites no código de árvores vermelho-preto, usamos uma única sentinela para representar NIL (ver Seção 10.2). Para uma árvore vermelho-preto T, a sentinela $nil[T]$ é um objeto com os mesmos campos que um nó comum na árvore. Seu campo *cor* é PRETO e seus outros campos – *p*, *esquerda*, *direita* e *chave* – podem ser definidos como valores arbitrários. Como mostra a Figura 13.1(b), todos os ponteiros para NIL são substituídos por ponteiros para a sentinela $nil[T]$.

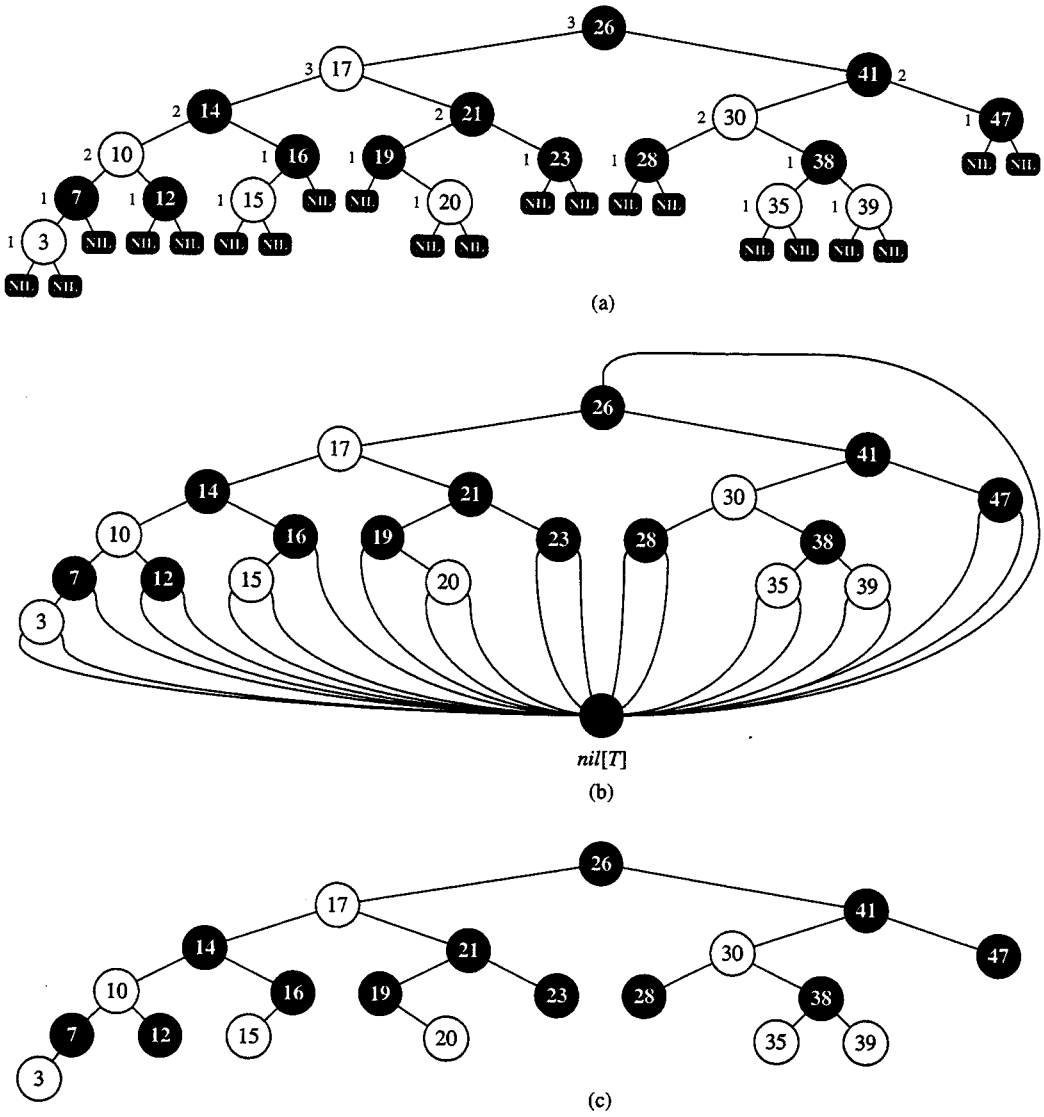


FIGURA 13.1 Uma árvore vermelho-preto com nós pretos escurecidos e nós vermelhos sombreados. Todo nó em uma árvore vermelho-preto é vermelho ou preto, os filhos de um nó vermelho são ambos pretos, e todo caminho simples desde um nó até uma folha descendente contém o mesmo número de nós pretos. **(a)** Toda folha, mostrada como um NIL, é preta. Cada nó não NIL é marcado com sua altura de preto: os nós NIL têm altura de preto igual a 0. **(b)** A mesma árvore vermelho-preto, mas com cada NIL substituído pela única sentinela $nil[T]$, que é sempre preta, e com alturas de preto 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

Usamos a sentinela de modo a podermos tratar um filho NIL de um nó x como um nó comum cujo pai é x. Poderíamos adicionar um nó de sentinela distinto para cada NIL na árvore, de forma que o pai de cada NIL fosse bem definido, mas essa abordagem desperdiçaria espaço. Em

vez disso, usamos a única sentinela *nil*[*T*] para representar todos os nós NIL – todas as folhas e o pai da raiz. Os valores dos campos *p*, *esquerda*, *direita* e *chave* da sentinela são irrelevantes, embora possamos defini-los durante o curso de um procedimento de acordo com nossa conveniência.

Em geral, limitamos nosso interesse aos nós internos de uma árvore vermelho-preto, pois 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).

Chamamos o número de nós pretos em qualquer caminho desde um nó x, sem incluir esse nó, até uma folha, de ***altura de preto*** do nó, denotada por bh(x). Pela propriedade 5, a noção de altura de preto é bem definida, pois todos os caminhos descendentes a partir do nó têm o mesmo número de nós pretos. Definimos a altura de preto de uma árvore vermelho-preto como a altura de preto de sua raiz.

O lema a seguir mostra por que as árvores vermelho-preto constituem boas árvores de pesquisa.

Lema 13.1
Uma árvore vermelho-preto com n nós internos tem altura no máximo $2 \lg(n + 1)$.

Prova Primeiro, vamos mostrar que a subárvore com raiz em qualquer nó x contém pelo menos $2^{bh(x)} - 1$ 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 (*nil*[*T*]), e a subárvore com raiz em x realmente contém pelo menos $2^{bh(x)} - 1 = 2^0 - 1 = 0$ nós internos. Para a etapa indutiva, considere um nó x que tem altura positiva e é um nó interno com dois filhos. Cada filho tem uma altura de preto bh(x) ou bh(x) – 1, dependendo de sua cor ser vermelha ou preta, respectivamente. Tendo em vista 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 pelo menos $2^{bh(x)-1} - 1$ nós internos. Desse modo, a subárvore com raiz em x contém pelo menos $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ 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, pelo menos metade dos nós em qualquer caminho simples desde a raiz até uma folha, não incluindo a raiz, deve ser preta. Conseqüentemente, a altura de preto da raiz deve ser pelo menos $h/2$; desse modo,

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

Movendo-se o valor 1 para o lado esquerdo e usando-se logaritmos em ambos os lados, obtém-se $\lg(n + 1) \geq h/2$, ou $h \leq 2 \lg(n + 1)$. ■

Uma conseqüência imediata desse lema é que as operações sobre conjuntos dinâmicos SEARCH, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR podem ser implementadas no tempo $O(\lg n)$ em árvores vermelho-preto, pois elas podem ser criadas para execução no tempo $O(h)$ em uma árvore de pesquisa de altura h (como mostra o Capítulo 12), e qualquer árvore vermelho-preto sobre n nós é uma árvore de pesquisa com altura $O(\lg n)$. (Evidentemente, referências a NIL nos algoritmos do Capítulo 12 teriam de ser substituídas por *nil*[*T*].) 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 oferecem suporte direto para as operações de conjuntos dinâmicos INSERT e DELETE, pois não garantem que a árvore de pesquisa binária modificada será uma árvore vermelho-preto. Porém, veremos nas Seções 13.3 e 13.4 que essas duas operações podem de fato ser admitidas no tempo $O(\lg n)$.

Exercícios

13.1-1
Desenhe a árvore de pesquisa binária completa de altura 3 sobre as chaves $\{1, 2, ..., 15\}$. Adicione as folhas NIL e defina as cores dos nós de três modos diferentes, tais que as alturas de preto das árvores vermelho-preto resultantes sejam 2, 3 e 4.

13.1-2
Desenhe a árvore vermelho-preto que resulta da chamada a TREE-INSERT sobre a árvore da Figura 13.1 com chave 36. Se o nó inserido for vermelho, a árvore resultante será uma árvore vermelho-preto? E se ele for preto?

13.1.3
Vamos definir uma ***árvore vermelho-preto relaxada*** como uma árvore de pesquisa binária que satisfaz às 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 será uma árvore vermelho-preto?

13.1.4
Vamos que todo nó vermelho em uma árvore vermelho-preto seja "absorvida" por seu pai preto, de forma que os filhos do nó vermelho se tornem filhos do pai preto. (Ignore o que acontece às chaves.) Quais são os graus possíveis de um nó preto depois que todos os seus filhos vermelhos são absorvidos? O que se pode dizer sobre as profundidades das folhas da árvore resultante?

13.1-5
Mostre que o mais longo caminho simples desde um nó x em uma árvore vermelho-preto até uma folha descendente tem comprimento no máximo duas vezes igual ao caminho simples mais curto desde o 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 de preto k? Qual é o menor número possível?

13.1-7
Descreva uma árvore vermelho-preto sobre n chaves que permita a maior razão possível de nós internos vermelhos para nós internos pretos. Qual é essa razão? Qual árvore tem a menor razão possível, e qual é essa razão?

13.2 Rotações

As operações sobre árvores de pesquisa TREE-INSERT e TREE-DELETE, quando executadas sobre uma árvore vermelho-preto com n chaves, demoram o tempo $O(\lg n)$. Considerando-se que 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 através de ***rotação***, uma operação local em uma árvore de pesquisa que preserva a propriedade de árvores de pesquisa binária. A Figura 13.2 mostra os dois tipos de rotações: rotações à esquerda e rotações à direita. Quando fazemos uma rotação à esquerda em um nó x, supomos que seu filho da direita y não é $nil[T]$. A rotação à esquerda "faz o pivô" em torno da ligação de x para y. Ela faz de y a nova raiz da subárvore, tendo x como filho da esquerda de y e o filho da esquerda de y como filho da direita de x.

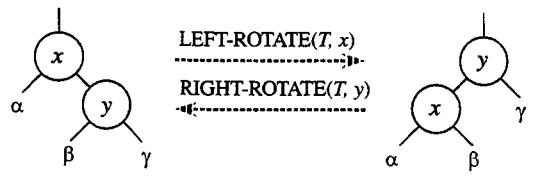


FIGURA 13.2 As operações de rotação em uma árvore de pesquisa binária. A operação LEFT-ROTATE(T, x) transforma a configuração dos dois nós da esquerda na configuração da direita, mudando um número constante de ponteiros. A configuração da direita pode ser transformada na configuração da esquerda pela operação inversa RIGHT-ROTATE(T, y). As letras α, β e γ representam subárvores arbitrárias. Uma operação de rotação preserva a propriedade de árvores de pesquisa binária: as chaves em α precedem *chave*[x], que precede as chaves em β, que precedem *chave*[y], que precede as chaves em γ

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

```
LEFT-ROTATE(T, x)
 1 y ← direita[x]              ▷ Define y.
 2 direita[x] ← esquerda[y]    ▷ Faz da subárvore esquerda de y a subárvore direita de x.
 3 p[esquerda[y]] ← x
 4 p[y] ← p[x]                 ▷ Liga o pai de x a y.
 5 if p[x] = nil[T]
 6   then raiz[T] ← y
 7   else if x = esquerda[p[x]]
 8           then esquerda [p[x]] ← y
 9           else direita[p[x]] ← y
10 esquerda[y] ← x             ▷ Coloca x à esquerda de y.
11 p[x] ← y
```

A Figura 13.3 mostra como LEFT-ROTATE opera. O código para RIGHT-ROTATE é simétrico. Tanto LEFT-ROTATE quanto RIGHT-ROTATE são executados no tempo $O(1)$. Somente os ponteiros são alterados por uma rotação; todos os outros campos em um nó permanecem os mesmos.

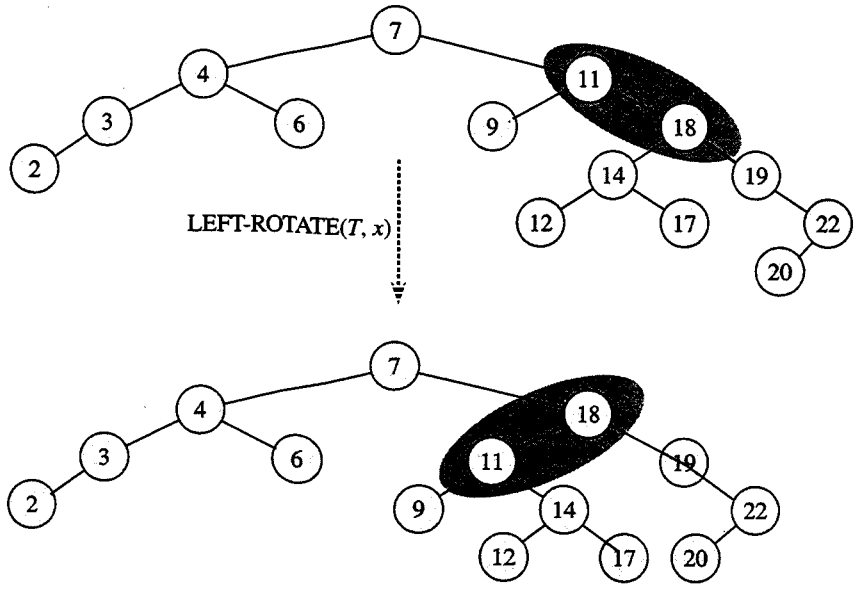


FIGURA 13.3 Um exemplo de como o procedimento LEFT-ROTATE(T, x) modifica uma árvore de pesquisa binária. Os percursos de árvore em ordem da árvore de entrada e a árvore modificada produzem a mesma listagem de valores de chaves

Exercícios

13.2-1
Escreva pseudocódigo para RIGHT-ROTATE.

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

13.2-3
Sejam a, b e c nós arbitrários nas subárvores α, β e γ, respectivamente, na árvore da esquerda da Figura 13.2. De que modo as profundidades de a, b e c mudam quando é realizada uma rotação à esquerda sobre o nó x na figura?

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

13.2-5 ⋆
Dizemos que uma árvore de pesquisa binária T_1 pode ser ***convertida à direita*** na árvore de pesquisa binária T_2 se é possível obter T_2 a partir de T_1 por meio de uma série de chamadas a RIGHT-ROTATE. Dê um exemplo de duas árvores T_1 e T_2 tais que T_1 não possa ser convertida à direita em T_2. Em seguida, mostre que, se uma árvore T_1 pode ser convertida à direita em T_2, ela pode ser convertida à direita com o uso de $O(n^2)$ chamadas a RIGHT-ROTATE.

13.3 Inserção

A inserção de um nó em uma árvore vermelho-preto de n nós pode ser realizada no tempo $O(\lg n)$. 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 pesquisa binária comum, e depois colorimos z de vermelho. Para garantir que as propriedades vermelho-preto serão preservadas, chamamos então um procedimento auxiliar RB-INSERT-FIXUP para recolorir os nós e executar rotações. A chamada RB-INSERT(T, z) insere o nó z, cujo campo *chave* já deve ser preenchido, na árvore vermelho-preto T.

```
RB-INSERT(T, z)
 1  y ← nil[T]
 2  x ← raiz[T]
 3  while x ≠ nil[T]
 4    do y ← x
 5       if chave[z] < chave[x]
 6         then x ← esquerda[x]
 7         else x ← direita[x]
 8  p[z] ← y
 9  if y = nil[T]
10    then raiz[T] ← z
11    else if chave[z] < chave[y]
12           then esquerda[y] ← z
13           else direita[y] ← z
14  esquerda[z] ← nil[T]
15  direita[z] ← nil[T]
16  cor[z] ← VERMELHO
17  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 $nil[T]$. Em segundo lugar, definimos $esquerda[z]$ e $direita[z]$ como $nil[T]$ 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, tendo em vista 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 cor[p[z]] = VERMELHO
 2      do if p[z] = esquerda[p[p[z]]]
 3          then y ← direita[p[p[z]]]
 4               if cor[y] ← VERMELHO
 5                 then cor[p[z]] ← PRETO                ▷ Caso 1
 6                      cor[y] ← PRETO                   ▷ Caso 1
 7                      cor[p[p[x]]] ← VERMELHO          ▷ Caso 1
 8                           z ← p[p[z]]                 ▷ Caso 1
 9                 else  if z = direita[p[z]]
10                         then z ← p[z]                 ▷ Caso 2
11                              LEFT-ROTATE(T, z)        ▷ Caso 2
12                      cor[p[z]] ← PRETO                ▷ Caso 3
13                           cor[p[p[z]]] ← VERMELHO     ▷ Caso 3
14                      RIGHT-ROTATE(T, p[p[z]])         ▷ Caso 3
15          else (igual a cláusula then
                      com "direita" e "esquerda" trocadas)
16  cor[raiz[T]] ← PRETO
```

Para entender como RB-INSERT-FIXUP funciona, desmembraremos nosso exame do código em três etapas principais. Primeiro, determinaremos que 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 loop **while** das linhas 1 a 15. Por fim, exploraremos cada um dos três casos[1] em que o loop **while** é dividido e veremos como eles alcançam essa meta. A Figura 13.4 mostra como RB-INSERT-FIXUP opera sobre 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 a ser válida, bem como a propriedade 3, pois ambos os filhos do nó vermelho recém-inserido são a sentinela $nil[T]$. A propriedade 5, que nos diz que o número de nós pretos é igual em todo caminho a partir de um dado nó, também é satisfeita, porque o nó z substitui a sentinela (preta), e o nó z é vermelho com filhos sentinelas. Desse modo, as únicas propriedades que poderiam ser violadas são a propriedade 2, que exige que a raiz seja preta, e a propriedade 4, que nos diz que um nó vermelho não pode ter um filho vermelho. Ambas as violações possíveis se devem ao fato de 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 depois que o nó z é inserido.

O loop **while** nas linhas 1 a 15 mantém o seguinte invariante de três partes:

No início de cada iteração do loop,

a. O nó z é vermelho.

b. Se $p[z]$ é a raiz, então $p[z]$ é preto.

[1] O caso 2 recai no caso 3, e por conseguinte esses dois casos não são mutuamente exclusivos.

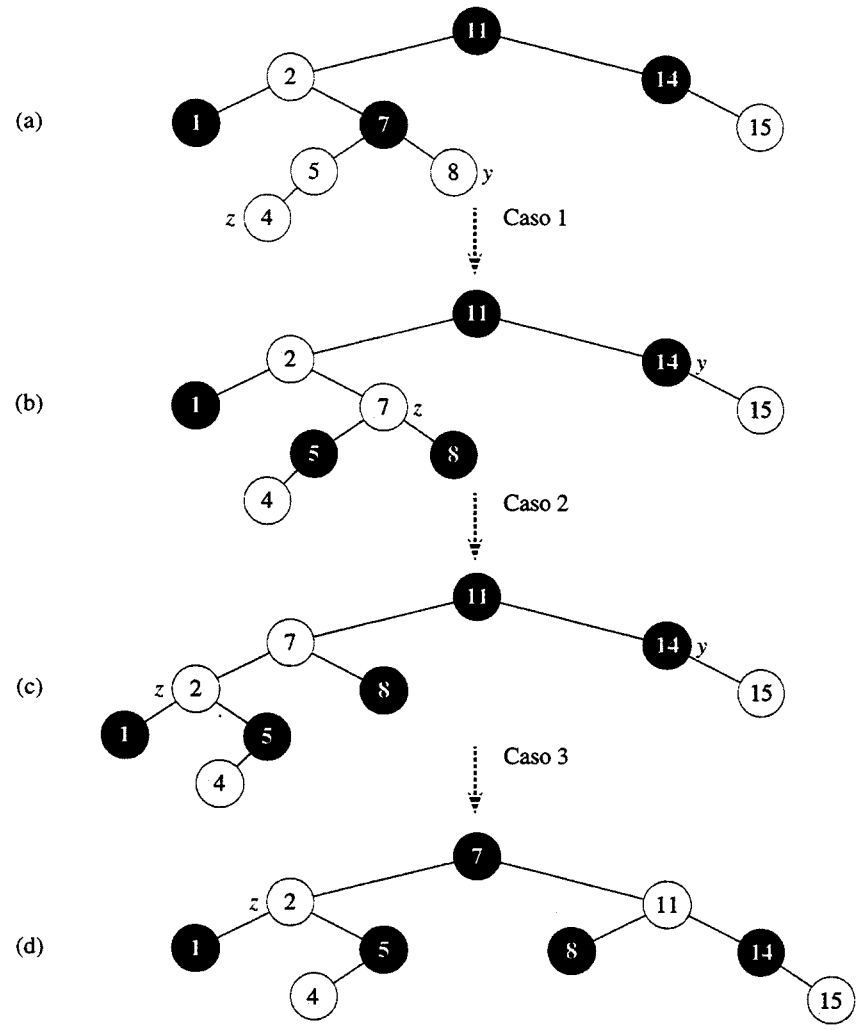


FIGURA 13.4 A operação de RB-INSERT-FIXUP. (a) Um nó z depois da inserção. Tendo em vista que z e seu pai $p[z]$ são ambos vermelhos, ocorre uma violação da propriedade 4. Tendo em vista que o tio y de z é vermelho, pode ser aplicado o caso 1 no código. Os nós são novamente coloridos, e o ponteiro z é movido para cima na árvore, resultando na árvore mostrada em (b). Mais uma vez, z e seu pai são ambos vermelhos, mas o tio y de z é preto. Como z é o filho da direita de $p[z]$, o caso 2 pode ser aplicado. Uma rotação à esquerda é executada, e a árvore resultante é mostrada em (c). Agora, z é o filho da esquerda de seu pai, e o caso 3 pode ser aplicado. Uma rotação à direita produz a árvore em (d), que é uma árvore vermelho-preto válida

c. Se existe uma violação das propriedades vermelho-preto, existe no máximo uma violação, e ela é uma violação da propriedade 2 ou da propriedade 4. Se há uma violação da propriedade 2, ela ocorre porque z é a raiz e é vermelho. Se há uma violação da propriedade 4, ela ocorre porque tanto z quanto $p[z]$ são vermelhos.

A parte (c), que lida com violações de propriedades vermelho-preto, é mais importante 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. Tendo em vista que nos concentraremos no nó z e nós próximos a ele na árvore, é útil saber da parte (a) que z é vermelho. Usaremos a parte (b) para mostrar que o nó $p[p[z]]$ existe quando fazemos referência a ele nas linhas 2, 3, 7, 8, 13 e 14.

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

Começamos com os argumentos de inicialização e término. Em seguida, à medida que examinarmos com mais detalhes como o corpo do loop funciona, demonstraremos que o loop mantém o invariante em cada iteração. Ao longo do caminho, também demonstraremos que há dois resultados possíveis de cada iteração do loop: o ponteiro z sobe a árvore, ou então algumas rotações são executadas e o loop termina.

Inicialização: Antes da primeira iteração do loop, começamos com uma árvore vermelho-preto sem violações e adicionamos 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 adicionado.

b. Se $p[z]$ é a raiz, então $p[z]$ 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 há uma violação da propriedade 2, então a raiz vermelha deve ser o nó z recém-adicionado, o qual é o único nó interno na árvore. Como o pai e ambos os filhos de z são a sentinela, que é preta, também não há uma violação da propriedade 4. Assim, essa violação da propriedade 2 é a única violação de propriedades vermelho-preto na árvore inteira.

Se há uma violação da propriedade 4, como os filhos do nó z são sentinelas pretas e a árvore não tinha outras violações antes de z ser adicionado, a violação tem de ter ocorrido porque tanto z quanto $p[z]$ são vermelhos. Além disso, não há outras violações de propriedades vermelho-preto.

Término: Quando o loop termina, ele o faz porque $p[z]$ é preto. (Se z é a raiz, então $p[z]$ é a sentinela $nil[T]$, que é preta.) Desse modo, não há violação da propriedade 4 no término do loop. Pelo loop invariante, a única propriedade que poderia deixar de ser válida é a propriedade 2. A linha 16 também restaura essa propriedade, de forma que, quando RB-INSERT-FIXUP termina, todas as propriedades vermelho-preto são válidas.

Manutenção: Na realidade, há seis casos há seis casos a considerar no loop **while**, mas três deles são simétricos aos outros três, dependendo do pai $p[z]$ de z ser um filho da esquerda ou um filho da direita do avô $p[p[z]]$ de z, o que é determinado na linha 2. Fornecemos o código apenas para a situação na qual $p[z]$ é um filho da esquerda. O nó $p[p[z]]$ existe pois, pela parte (b) do loop invariante, se $p[z]$ é a raiz, então $p[z]$ é preto. Tendo em vista que entramos em uma iteração de loop somente se $p[z]$ é vermelho, sabemos que $p[z]$ não pode ser a raiz. Conseqüentemente, $p[p[z]]$ existe.

O caso 1 se distingue dos casos 2 e 3 pela cor do irmão do pai de z, ou "tio". A linha 5 faz y apontar para o tio $direita[p[p[z]]]$ de z, e é feito um teste na linha 4. Se y é vermelho, então o caso 1 é executado. Do contrário, o controle passa para os casos 2 e 3. Em todos os três casos, o avô $p[p[z]]$ de z é preto, pois seu pai $p[z]$ é vermelho, e a propriedade 3 é violada apenas entre z e $p[z]$.

Caso 1: o tio de y de z é vermelho

A situação para o caso 1 (linhas 5 a 8) é mostrada na Figura 13.5. O caso 1 é executado quando tanto $p[z]$ quanto y são vermelhos. Tendo em vista que $p[p[z]]$ é preto, podemos colorir tanto $p[z]$ quanto y de preto, corrigindo assim o problema de z e $p[z]$ serem ambos vermelhos, e também colorimos $p[p[z]]$ de vermelho, mantendo assim a propriedade 5. Em seguida, repetimos o loop **while** com $p[p[z]]$ como o novo nó z. O ponteiro z sobe dois níveis na árvore.

Agora mostramos que o caso 1 mantém o loop invariante no início da próxima iteração. Usamos z para denotar o nó z na iteração atual, e $z' = p[p[z]]$ para denotar o nó z no teste da linha 1 na iteração seguinte.

a. Como essa iteração torna a cor de $p[p[z]]$ vermelha, o nó z' é vermelho no início da próxima iteração.

b. O nó $p[z']$ é $p[p[p[z]]]$ 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.

c. Já mostramos que o caso 1 mantém a propriedade 5, e é claro que ele não introduz uma violação da propriedade 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'.

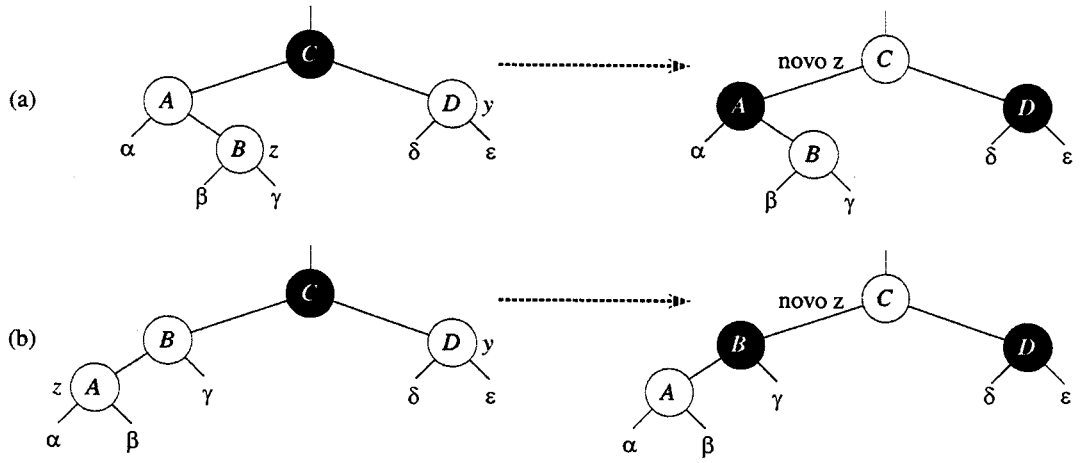


FIGURA 13.5 O caso 1 do procedimento RB-INSERT. A propriedade 4 é violada, pois z e seu pai $p[z]$ são ambos vermelhos. A mesma ação é adotada se **(a)** z é um filho da direita ou **(b)** z é um filho da esquerda. Cada uma das subárvores, $\alpha, \beta, \gamma, \delta$ e ε tem uma raiz preta, e cada uma tem a mesma altura de preto. O código para o caso 1 altera as cores de alguns nós, preservando a propriedade 5: todos os caminhos descendentes desde um nó até uma folha têm o mesmo número de pretos. O loop **while** continua com o avô $p[p[z]]$ 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

Se 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. Em seguida, ele tornou z' vermelho e deixou $p[z']$ como estava. Se $p[z']$ era preto, não há nenhuma violação da propriedade 4. Se $p[z']$ era vermelho, colorir z' de vermelho criou uma violação da propriedade 4 entre z' e $p[z']$.

Caso 2: o tio y de z é preto e z é um filho da direita
Caso 3: o tipo y de z é preto e z é um filho da esquerda

Nos casos 2 e 3, a cor do tio y de z é preta. Os dois casos se distinguem pelo fato de z ser um filho da direita ou da esquerda de $p[z]$. 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 da direita de seu pai. Usamos imediatamente uma rotação à esquerda para transformar a situação no caso 3 (linhas 12 a 14), na qual o nó z é um filho da esquerda. Como tanto z quanto $p[z]$ são vermelhos, a rotação não afeta nem a altura de preto dos nós nem a propriedade 5. Quer entremos no caso 3 diretamente ou através do caso 2, o tio y de z será preto, pois, do contrário, teríamos executado o caso 1. Além disso, o nó $p[p[z]]$ existe, pois 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 $p[p[z]]$ permanece inalterada. No caso 3, executamos algumas mudanças de co-

res e uma rotação à direita, o que preserva a propriedade 5; em seguida, tendo em vista que não temos mais dois nós vermelhos em uma linha, encerramos. O corpo do loop **while** não é executado outra vez, pois agora $p[z]$ é preto.

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

a. O caso 2 faz z apontar para $p[z]$, que é vermelho. Nenhuma mudança adicional em z ou em sua cor ocorre nos casos 2 e 3.

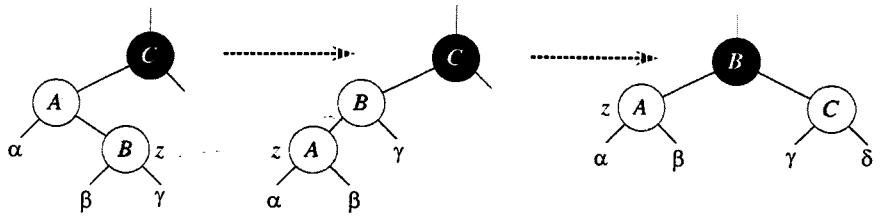
b. O caso 3 torna $p[z]$ preto, de forma que, se $p[z]$ é a raiz no início da próxima iteração, ele é preto.

c. Como ocorre no caso de 1, as propriedades 1, 3 e 5 são mantidas nos casos 2 e 3.

Tendo em vista 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, pois o único nó que se tornou vermelho passa a ser um filho de um nó preto pela rotação no caso 3.

Os casos 2 e 3 corrigem a única violação da propriedade 4, e eles não introduzem outra violação.

Tendo mostrado que cada iteração do loop mantém o invariante, mostramos que RB-INSERT-FIXUP restaura corretamente as propriedades vermelho-preto.



Caso 2 Caso 3

FIGURA 13.6 Os casos 2 e 3 do procedimento RB-INSERT. Como no caso 1, a propriedade 4 é violada no caso 2 ou no caso 3, porque z e seu pai $p[z]$ são ambos vermelhos. Cada uma das subárvores, α, β, γ e δ tem uma raiz preta (α, β e γ pela propriedade 4, e δ porque, em caso contrário, estaríamos no caso 1), e cada uma tem a mesma altura de preto. O caso 2 é transformado no caso 3 por uma rotação à esquerda, o que preserva a propriedade 5: todos os caminhos descendentes de um nó até uma folha têm o mesmo número de pretos. O caso 3 causa algumas mudanças de cores e uma rotação à direita, o que também preserva a propriedade 5. Em seguida, o loop **while** termina, porque a propriedade 4 é satisfeita: não há mais dois nós vermelhos em uma linha

Análise

Qual é o tempo de execução de RB-INSERT? Tendo em vista que a altura de uma árvore vermelho-preto sobre n nós é $O(\lg n)$, as linhas 1 a 16 de RB-INSERT levam o tempo $O(\lg n)$. Em RB-INSERT-FIXUP, o loop **while** só se repete se o caso 1 é executado, e então o ponteiro z sobe dois níveis na árvore. O número total de vezes que o loop **while** pode ser executado é portanto $O(\lg n)$. Desse modo, RB-INSERT demora um tempo total $O(\lg n)$. É interessante observar que ele nunca executa mais de duas rotações, pois o loop **while** termina se o caso 2 ou o caso 3 é executado.

Exercícios

13.3-1

Na linha 16 de RB-INSERT, definimos a cor do nó recém-inserido como vermelho. Note que, se tivéssemos optado por definir a cor de z como preto, a propriedade 4 de uma árvore vermelho-preto não seria violada. Por que não optamos por definir a cor de z como preta?

13.3-2

Mostre as árvores vermelho-preto que resultam após a inserção bem-sucedida das chaves 41, 38, 31, 12, 19, 8 em uma árvore vermelho-preto inicialmente vazia.

13.3-3

Suponha que a altura de preto de cada uma das subárvores $\alpha, \beta, \gamma, \delta, \varepsilon$ nas Figuras 13.5 e 13.6 seja k. Identifique cada nó em cada figura com sua altura de preto, a fim de verificar se a propriedade 5 é preservada pela transformação indicada.

13.3-4

O professor Teach está preocupado com o fato de RB-INSERT-FIXUP poder definir $cor[nil[T]]$ como VERMELHO, em cujo caso o teste da linha 1 não faria o loop terminar quando z fosse a raiz. Mostre que a preocupação do professor é infundada, demonstrando que RB-INSERT-FIXUP nunca define $cor[nil[T]]$ como 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 pelo menos um nó vermelho.

13.3-6

Sugira como implementar RB-INSERT de maneira eficiente, se a representação para árvores vermelho-preto não incluir nenhum espaço de 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 $O(\lg n)$. A eliminação de um nó de uma árvore vermelho-preto é apenas ligeiramente mais complicada que a inserção de um nó.

O procedimento RB-DELETE é uma modificação secundária do procedimento TREE-DELETE (Seção 12.3). Após extrair um nó, ele 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 if esquerda[z] = nil[T] or direita[z] = nil[T]
 2    then y ← z
 3    else y ← TREE-SUCCESSOR(z)
 4 if esquerda[y] ≠ nil[T]
 5    then x ← esquerda[y]
 6    else x ← direita[y]
 7 p[x] ← p[y]
 8 if p[y] = nil[T]
 9    then raiz[T] ← x
10     else if y = esquerda[p[y]]
11          then esquerda[p[y]] ← x
12          else direita[p[y]] ← x
13  if y ≠ z
14    then chave[z] ← chave[y]
```

```
15          copia dados satélite de y em z
16  if cor[y] = PRETO
17    then RB-DELETE-FIXUP(T, x)
18  return y
```

Existem três diferenças entre os procedimentos TREE-DELETE e RB-DELETE. Primeiro, todas as referências a NIL em TREE-DELETE foram substituídas por referências à sentinela *nil*[*T*] em RB-DELETE. Segundo, o teste para determinar se x é NIL na linha 7 de TREE-DELETE foi removido, e a atribuição $p[x] \leftarrow p[y]$ é executada de modo incondicional na linha 7 de RB-DELETE. Desse modo, se x é a sentinela *nil*[*T*], o ponteiro de seu pai aponta para o pai do nó y extraído. Terceiro, uma chamada a RB-DELETE-FIXUP é efetuada nas linhas 16 e 17 se y é preto. Se y é vermelho, as propriedades vermelho-preto ainda são válidas quando y é extraído, pelas seguintes razões:

- Nenhuma altura preta na árvore mudou.
- Nenhum nó vermelho se tornou adjacente.
- Como y não poderia ter sido a raiz se fosse vermelho, a raiz permanece preta.

O nó x repassado a RB-DELETE-FIXUP é um entre dois nós: ou o nó que era o único filho de y antes de y ser extraído, se y tinha um filho que não era a sentinela *nil*[*T*] ou, se y não tinha filhos, x é a sentinela *nil*[*T*]. Nesse último caso, a atribuição incondicional na linha 7 garante que o pai de x agora é o nó que anteriormente era o pai de y, quer seja x um nó interno de transporte de chave ou a sentinela *nil*[*T*].

Agora podemos examinar o modo como o procedimento RB-DELETE-FIXUP restaura as propriedades vermelho-preto para a árvore de pesquisa.

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ raiz[T] e cor[x] = PRETO
 2    do if x = esquerda[p[x]]
 3         then w ← direita[p[x]]
 4              if cor[w] = VERMELHO
 5                then cor[w] ← PRETO                          ▷ Caso 1
 6                     cor[p[x]] ← VERMELHO                    ▷ Caso 1
 7                     LEFT-ROTATE(T, p[x])                    ▷ Caso 1
 8                     w ← direita[p[x]]                       ▷ Caso 1
 9              if cor[esquerda[w]] = PRETO e cor[direita[w]] = PRETO
10                then cor[w] ← VERMELHO                       ▷ Caso 2
11                     x ← p[x]                                ▷ Caso 2
12                else if cor[direita[w]] = PRETO
13                       then cor[esquerda[w]] ← PRETO         ▷ Caso 3
14                            cor[w] ← VERMELHO                ▷ Caso 3
15                            RIGHT-ROTATE(T, w)               ▷ Caso 3
16                            w ← direita[p[x]]                ▷ Caso 3
17                            cor[w] ← cor[p[x]]               ▷ Caso 4
18                            cor[p[x]] ← PRETO                ▷ Caso 4
19                            cor[direita[w]] ← PRETO          ▷ Caso 4
20                            LEFT-ROTATE(T, p[x])             ▷ Caso 4
21                            x ← raiz[T]                      ▷ Caso 4
22         else (igual à cláusula then com "direita" e "esquerda" trocadas)
23  cor[x] ← PRETO
```

Se o nó y extraído em RB-DELETE é preto, três problemas podem surgir. Primeiro, se y era a raiz e um filho vermelho de y se torna a nova raiz, violamos a propriedade 2. Segundo, se tanto x quanto $p[y]$ (que agora também é $p[x]$) eram vermelhos, então violamos a propriedade 4. Terceiro, a remoção de y faz qualquer caminho que continha y anteriormente ter um nó preto a menos. Desse modo, a propriedade 5 agora é violada por qualquer ancestral de y na árvore. Podemos corrigir esse problema dizendo que o nó x tem um preto "extra". Isto é, se adicionarmos 1 à contagem de nós pretos em qualquer caminho que contenha x então, sob essa interpretação, a propriedade 5 se mantém válida. Quando extraímos o nó preto y, "empurramos" sua característica de preto sobre seu filho. O problema é que agora o nó x não é vermelho nem preto, violando assim 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 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, o preto extra em um nó é refletido no fato de x apontar para o nó, e não no atributo *cor*.

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

1. x apontar para um nó vermelho e preto, em cujo caso iremos colorir x (unicamente) de preto na linha 23,
2. x apontar para a raiz, e nesse caso o preto extra pode ser simplesmente "removido", ou
3. podem ser executadas operações adequadas de rotação e de nova coloração.

Dentro do loop **while**, x sempre aponta para um nó duplamente preto não raiz. Determinamos na linha 2 se x é um filho da esquerda ou um filho da direita de seu pai $p[x]$. (Fornecemos o código para a situação na qual x é um filho da esquerda; a situação na qual x é um filho da direita – linha 22 – é simétrica.) Mantemos um ponteiro w para o irmão de x. Como o nó x é duplamente preto, o nó w não pode ser $nil[T]$; caso contrário, o número de pretos no caminho desde $p[x]$ até a folha w (unicamente preta) seria menor que o número no caminho de $p[x]$ até x.

Os quatro casos[2] no código estão ilustrados na Figura 13.7. Antes de examinar cada caso em detalhes, vamos dar uma olhada mais geral na maneira como podemos verificar se a transformação em cada um dos casos preserva a propriedade 5. A idéia chave é que em cada caso o número de nós pretos (incluindo o preto extra de x) desde (e inclusive) a raiz da subárvore mostrada até cada uma das subárvores α, β, ..., ζ é preservado pela transformação. Desse modo, se a propriedade 5 é válida antes da transformação, ela continua a ser válida daí em diante. Por exemplo, na Figura 13.7(a), que ilustra o caso 1, o número de nós pretos desde a raiz até a subárvore α ou β é 3, tanto antes quanto após a transformação. (Mais uma vez, lembre-se de que o nó x adiciona um preto extra.) De modo semelhante, o número de nós pretos desde a raiz até qualquer das subárvores γ, δ, ε e ζ é 2, tanto antes quanto 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, então o número de nós pretos desde a raiz até α é 2 + contador(c), tanto antes quanto após a transformação. Nesse caso, depois da transformação, o novo nó x tem o atributo *cor* igual a c, mas esse nó é realmente vermelho e preto (se c = VERMELHO) ou duplamente preto (se c = PRETO). Os outros casos podem ser verificados de maneira semelhante (ver Exercício 13.4-5.)

[2] Como em RB-INSERT-FIXUP, os casos em RB-DELETE-FIXUP não são mutuamente exclusivos.

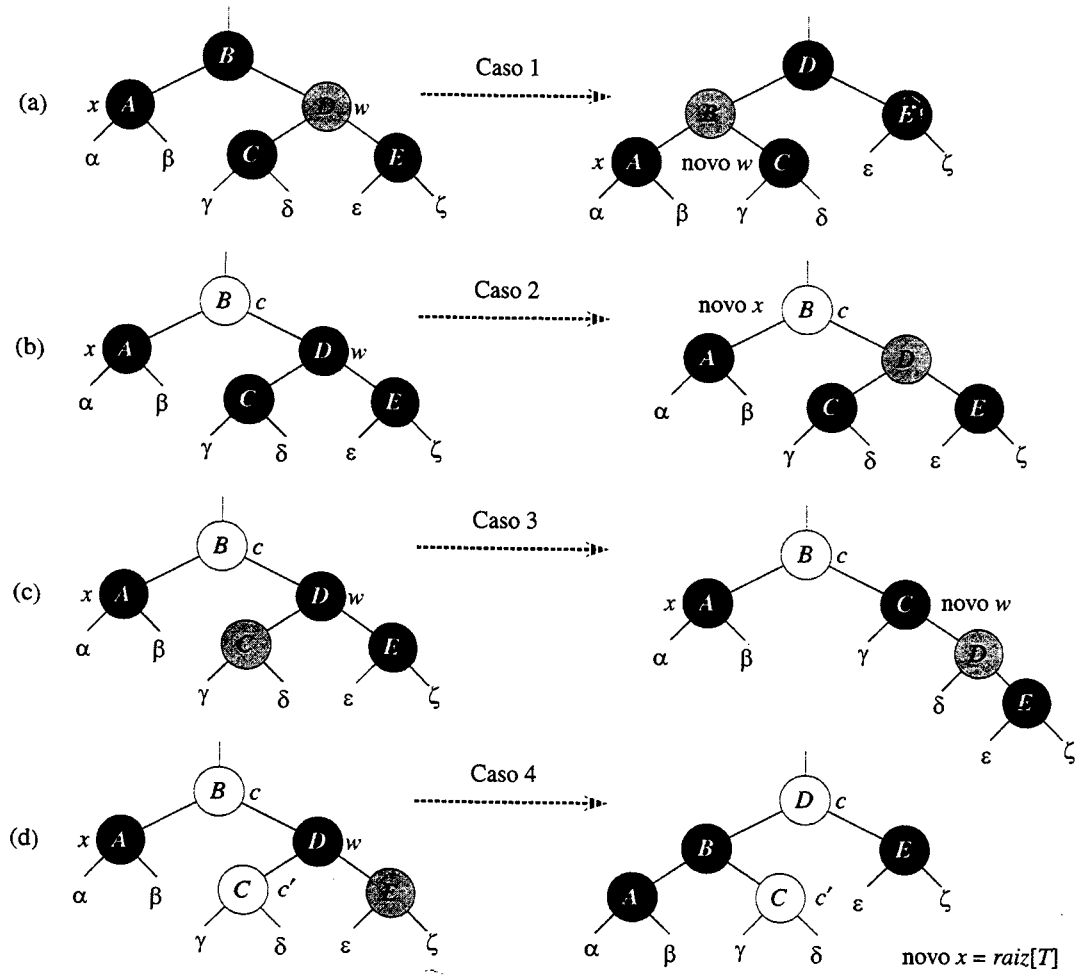


FIGURA 13.7 Os casos no loop **while** do procedimento RB-DELETE-FIXUP. Os nós escurecidos têm atributos *cor* PRETO, nós fortemente sombreados têm atributos *cor* VERMELHO e nós ligeiramente sombreados têm atributos *cor representados por c* e *c'*, que podem ser VERMELHO ou PRETO. As letras $\alpha, \beta, \ldots, \zeta$ representam subárvores arbitrárias. Em cada caso, a configuração da esquerda é transformada na configuração da direita pela mudança de algumas cores e/ou execução de uma rotação. Qualquer nó indicado por x tem um preto extra e é duplamente preto ou vermelho e preto. O único caso que faz o loop se repetir é o caso 2. (**a**) O caso 1 é transformado no caso 2, 3 ou 4 pela troca das cores dos nós B e D e pela execução de uma rotação à esquerda. (**b**) No caso 2, o preto extra representado pelo ponteiro x é movido para cima na árvore, colorindo-se o nó D de vermelho e definindo-se x de modo a apontar para o nó B. Se entrarmos no caso 2 através do caso 1, o loop **while** terminará, pois 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 pela troca das cores dos nós C e D e pela execução de uma rotação à direita. (**d**) No caso 4, o preto extra representado por x pode ser removido mudando-se algumas cores e executando-se uma rotação à esquerda (sem violar as propriedades vermelho-preto), e o loop se encerra

Caso 1: o irmão w de x é vermelho

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

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

Caso 2: o irmão w de x é preto, e ambos os filhos de w são pretos

No caso 2 (linhas 10 e 11 de RB-DELETE-FIXUP e Figura 13.7(b)), ambos os filhos de w são pretos. Tendo em vista que w também é preto, retiramos um preto de x e de w, deixando x com apenas um preto e deixando w vermelho. Para compensar, a remoção de um preto de x e de w, seria interessante adicionar um preto extra a $p[x]$, que era originalmente vermelho ou preto. Fazemos isso repetindo o loop **while** com $p[x]$ como o novo nó x. Observe que, se entrarmos no caso 2 através do caso 1, o novo nó x será vermelho e preto, pois o $p[x]$ original era vermelho. Conseqüentemente, o valor c do atributo *cor* do novo nó x é VERMELHO, e o loop termina quando testa a condição de loop. O novo nó x é então colorido (unicamente) de preto na linha 23.

Caso 3: o irmão w de x é preto, o filho da esquerda de w é vermelho e o filho da direita de w é preto

O caso 3 (linhas 13 a 16 e Figura 13.7(c)) ocorre quando w é preto, seu filho da esquerda é vermelho e seu filho da direita é preto. Podemos alternar as cores de w e de seu filho da esquerda *esquerda*$[w]$, e depois executar uma rotação à direita sobre w sem violar qualquer das propriedades vermelho-preto. O novo irmão w de x é agora um nó preto com um filho da direita vermelho, e desse modo transformamos o caso 3 no caso 4.

Caso 4: o irmão w de x é preto, e o filho da direita de w é vermelho

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

Análise

Qual é o tempo de execução de RB-DELETE? Tendo em vista 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 termina depois de executar um número constante de mudanças de cores e no máximo três rotações. O caso 2 é o único caso em que o loop **while** pode ser repetido, e então o ponteiro x se move para cima na árvore no máximo $O(\lg n)$ vezes e nenhuma rotação é executada. Desse modo, o procedimento RB-DELETE-FIXUP demora o tempo $O(\lg n)$ e executa no máximo três rotações, e assim 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 $p[y]$ são ambos vermelhos em RB-DELETE, então a propriedade 4 é restabelecida pela chamada RB-DELETE-FIXUP(T, x).

13.4-3

No Exercício 13.3-2, você encontrou 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 *nil*$[T]$?

13.4-5

Em cada um dos casos da Figura 13.7, forneça a contagem de nós pretos desde a raiz da subárvore mostrada até cada uma das subárvores $\alpha, \beta, \ldots, \zeta$, e confirme que cada contagem permanece a mesma depois da transformação. Quando um nó tiver um atributo *cor* c ou c', use a notação contador(c) ou contador(c') simbolicamente em sua contagem.

13.4-6

Os professores Skelton e Baron estão preocupados com o fato de que, no início do caso 1 de RB-DELETE-FIXUP, o nó $p[x]$ não poderia ser preto. Se os professores estão corretos, então as linhas 5 e 6 estão erradas. Mostre que $p[x]$ deve ser preto no início do caso 1, e portanto os professores não têm nenhuma razão para se preocupar.

13.4-7

Suponha que um nó x seja inserido em uma árvore vermelho-preto com RB-INSERT e depois imediatamente eliminado com RB-DELETE. A árvore vermelho-preto resultante é igual à árvore vermelho-preto inicial? Justifique sua resposta.

Problemas

13-1 Conjuntos dinâmicos persistentes

Durante o curso de um algoritmo, às vezes achamos que precisamos manter versões anteriores de um conjunto dinâmico à medida que ele é atualizado. Tal conjunto é chamado ***persistente***. Um modo de implementar um conjunto persistente é copiar o conjunto inteiro sempre que ele é modificado, mas essa abordagem pode reduzir a velocidade de um programa e também consumir muito espaço. Às vezes, podemos fazer muito melhor que isso.

Considere um conjunto persistente S com as operações INSERT, DELETE e SEARCH, que implementamos usando árvores de pesquisa binária como mostra a Figura 13.8(a). Mantemos uma raiz separada para cada versão do conjunto. Para inserir a chave 5 no conjunto, criamos um novo nó com chave 5. Esse nó se torna o filho da esquerda de um novo nó com chave 7, pois não podemos modificar o nó existente com chave 7. De modo semelhante, o novo nó com chave 7 se torna o filho da esquerda de um novo nó com chave 8, cujo filho da direita é o nó existente com chave 10. O novo nó com chave 8 se torna, por sua vez, o filho da direita de uma nova raiz r' com chave 4, cujo filho da esquerda é o nó existente com chave 3. Desse modo, copiamos apenas parte da árvore e compartilhamos alguns dos nós com a árvore original, como mostra a Figura 13.8(b).

Suponha que cada nó da árvore tenha os campos *chave*, *esquerda* e *direita*, mas nenhum campo pai. (Consulte também o Exercício 13.3-6.)

a. No caso geral de uma árvore de pesquisa binária persistente, identifique os nós que precisam ser alterados para se inserir uma chave k ou eliminar um nó y.

b. Escreva um procedimento PERSISTENT-TREE-INSERT que, dada uma árvore persistente T e uma chave k a ser inserida, retorne uma nova árvore persistente T' que seja o resultado da inserção de k em T.

c. Se a altura da árvore de pesquisa binária persistente T é h, quais são os requisitos de tempo e espaço da sua implementação de PERSISTENT-TREE-INSERT? (O requisito de espaço é proporcional ao número de novos nós alocados.)

d. Suponha que tivéssemos incluído o campo pai em cada nó. Nesse caso, PERSISTENT-TREE-INSERT precisaria executar uma operação de cópia adicional. Prove que PERSISTENT-TREE-INSERT exigiria então o tempo e o espaço $\Omega(n)$, onde n é o número de nós na árvore.

e. Mostre como usar árvores vermelho-preto para garantir que o tempo de execução do pior caso e o espaço são $O(\lg n)$ por inserção ou eliminação.

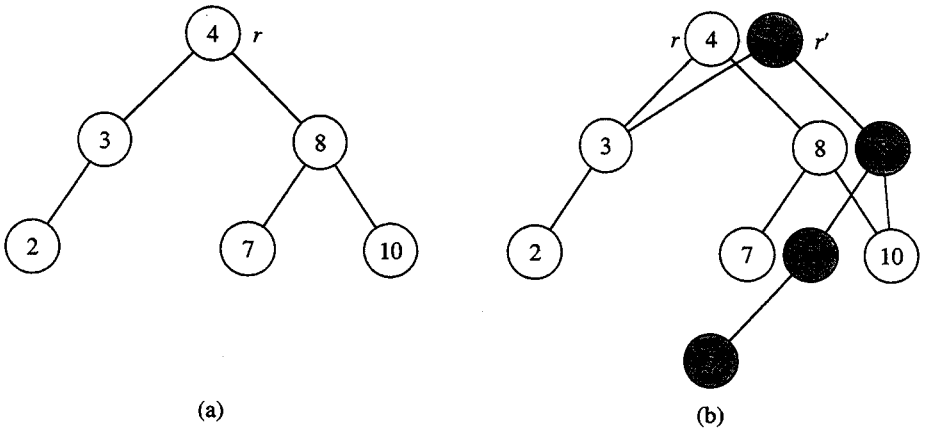


FIGURA 13.8 (a) Uma árvore de pesquisa binária com chaves 2, 3, 4, 7, 8, 10. (b) A árvore de pesquisa binária persistente que resulta da inserção da chave 5. A versão mais recente do conjunto consiste nos nós acessíveis a partir da raiz r', e a versão anterior consiste nos nós acessíveis a partir de r. Os nós fortemente sombreados são adicionados quando a chave 5 é inserida

13-2 Operação de junção sobre árvores vermelho-preto

A operação de ***junção*** utiliza dois conjuntos dinâmicos S_1 e S_2 e um elemento x tal que, para qualquer $x_1 \in S_1$ e $x_2 \in S_2$, temos $chave[x_1] \leq chave[x] \leq chave[x_2]$. Ela retorna um conjunto $S = S_1 \cup \{x\} \cup S_2$. Neste problema, investigamos como implementar a operação de junção sobre árvores vermelho-preto.

a. Dada uma árvore vermelho-preto T, armazenamos sua altura de preto como o campo $bh[T]$. Mostre que esse campo pode ser mantido por RB-INSERT e RB-DELETE sem exigir espaço de armazenamento extra na árvore e sem aumentar os tempos de execução assintóticos. Mostre que, enquanto descemos em T, podemos determinar a altura de preto de cada nó que visitamos no tempo $O(1)$ por nó visitado.

Desejamos implementar a operação RB-JOIN(T_1, x, T_2), o que destrói T_1 e T_2 e retorna uma árvore vermelho-preto $T = T_1 \cup \{x\} \cup T_2$. Seja n o número total de nós em T_1 e T_2.

b. Suponha que $bh[T_1] \geq bh[T_2]$. Descreva um algoritmo de tempo $O(\lg n)$ que encontre um nó preto y em T_1 com a maior chave entre os nós cuja altura de preto é $bh[T_2]$.

c. Seja T_y a subárvore com raiz em y. Descreva de que modo T_y pode ser substituído por $T_y \cup \{x\} \cup T_2$ no tempo $O(1)$ sem destruir a propriedade de árvore de pesquisa binária.

d. Que cor devemos usar em x para que as propriedades vermelho-preto 1, 3 e 5 sejam mantidas? Descreva de que maneira as propriedades 2 e 4 podem ser impostas no tempo $O(\lg n)$.

e. Demonstre que nenhuma generalidade é perdida tomando-se a hipótese da parte (b). Descreva a situação simétrica que surge quando $bh[T_1] \leq bh[T_2]$.

f. Mostre que o tempo de execução de RB-JOIN é O(lg n).

13-3 Árvores AVL

Uma ***árvore AVL*** é uma árvore de pesquisa binária que é de ***altura balanceada***: Para cada nó x, as alturas das subárvores esquerda e direita de x diferem por no máximo 1. Para implementar uma árvore AVL, mantemos um campo extra em cada nó: $h[x]$ é a altura de nó x. Como ocorre no caso de qualquer outra árvore de pesquisa binária T, supomos que $raiz[T]$ aponta para o nó raiz.

a. Prove que uma árvore AVL com n nós tem altura $O(\lg n)$. (*Sugestão:* Prove que, em uma árvore AVL de altura h, existem pelo menos F_h nós, onde F_h é o h-ésimo número de Fibonacci.)

b. No caso da inserção em uma árvore AVL, primeiro um nó é inserido no lugar apropriado na ordem da árvore de pesquisa binária. Depois dessa inserção, a árvore pode não ser mais de altura balanceada. Especificamente, as alturas dos filhos da esquerda e da direita de algum nó podem diferir por 2. Descreva um procedimento BALANCE(x), que toma uma subárvore com raiz em x cujos filhos da esquerda e da direita são de altura balanceada e têm alturas que diferem por no máximo 2, isto é, $|h[direita[x]] - h[esquerda[x]]| \leq 2$, e altera a subárvore com raiz em x para ser de altura balanceada. (*Sugestão:* Use rotações.)

c. Usando a parte (b), descreva um procedimento recursivo AVL-INSERT(x, z), que toma um nó x dentro de uma árvore AVL e um nó z recentemente criado (cuja chave já foi preenchida) e adiciona z à subárvore com raiz em x, mantendo a propriedade de que x é a raiz de uma árvore AVL. Como no procedimento TREE-INSERT da Seção 12.3, suponha que $chave[z]$ já foi preenchida e que $esquerda[z]$ = NIL e $direita[z]$ = NIL; suponha também que $h[z] = 0$. Desse modo, para inserir o nó z na árvore AVL T, chamamos AVL-INSERT($raiz[T]$, z).

d. Dê um exemplo de uma árvore AVL de n nós, na qual uma operação AVL-INSERT resulta na execução de $\Omega(\lg n)$ rotações.

13-4 Treaps

Se inserirmos um conjunto de n itens em uma árvore de pesquisa binária, a árvore resultante pode ficar terrivelmente desbalanceada, levando a tempos de pesquisa longos. Porém, como vimos na Seção 12.4, árvores de pesquisa binária construídas aleatoriamente tendem a ser balanceadas. Portanto, uma estratégia que em média constrói uma árvore balanceada para um conjunto fixo de itens é permutar os itens ao acaso e depois inseri-los nessa ordem na árvore.

E se não tivermos todos os itens ao mesmo tempo? Se recebermos os itens um de cada vez, ainda poderemos construir aleatoriamente uma árvore de pesquisa binária a partir deles? Examinaremos uma estrutura de dados que responde de forma afirmativa a essa pergunta. Um ***treap*** é uma árvore de pesquisa binária com uma forma modificada de ordenar os nós. A Figura 13.9 mostra um exemplo. Como de hábito, cada nó x na árvore tem um valor de chave $chave[x]$. Além disso, atribuímos $prioridade[x]$, que é um número aleatório escolhido independentemente para cada nó. Supomos que todas as prioridades são distintas e também que todas as chaves são distintas. Os nós do treap são ordenados de forma que as chaves obedeçam à propriedade de árvore de pesquisa binária, e que as prioridades obedeçam à propriedade de ordem de heap mínimo:

- Se v é um filho da esquerda de u, então $chave[v] < chave[u]$.
- Se v é um filho da direita de u, então $chave[v] > chave[u]$.
- Se v é um filho de u, então $prioridade[v] > prioridade[u]$.

(Essa combinação de propriedades é o motivo pelo qual a árvore é chamada um "treap"; ela tem características de uma árvore – tree, em inglês – de pesquisa binária e de um heap.)

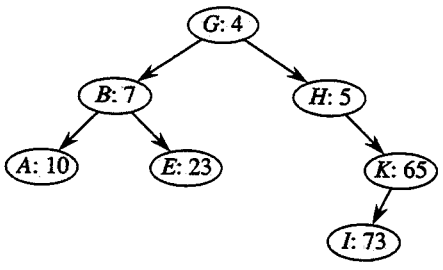


FIGURA 13.9 Um treap. Cada nó x é identificado com $chave[x]$:$prioridade[x]$. Por exemplo, a raiz tem chave G e prioridade 4

É conveniente pensar em treaps do modo a seguir. Suponha que inserimos nós $x_1, x_2, \ldots, x_n$, com chaves associadas, em um treap. Então, o treap resultante é a árvore que teria sido formada se os nós fossem inseridos em uma árvore de pesquisa binária normal na ordem dada por suas prioridades (escolhidas ao acaso), isto é, $prioridade[x_i] < prioridade[x_j]$ significa que x_i foi inserido antes de x_j.

a. Mostre que, dado um conjunto de nós $x_1, x_2, \ldots, x_n$, com chaves e prioridades associadas (todas distintas), existe um único treap associado a esses nós.

b. Mostre que a altura esperada de um treap é $\Theta(\lg n)$ e, conseqüentemente, o tempo para procurar um valor no treap é $\Theta(\lg n)$.

c. Explique como TREAP-INSERT funciona. Explique a idéia em linguagem comum e forneça o pseudocódigo. (*Sugestão:* Execute o procedimento habitual de inserção em árvore de pesquisa binária e depois execute rotações para restaurar a propriedade de ordem de heap mínimo.)

d. Mostre que o tempo de execução esperado de TREAP-INSERT é $\Theta(\lg n)$.

TREAP-INSERT executa uma pesquisa e depois uma seqüência de rotações. Embora essas duas operações tenham o mesmo tempo de execução esperado, elas têm custos diferentes na prática. Uma pesquisa lê informações do treap sem modificá-lo. Em contraste, uma rotação altera os ponteiros pai e filho dentro do treap. Na maioria dos computadores, as operações de leitura são muito mais rápidas que as operações de gravação. Desse modo, seria interessante que TREAP-INSERT executasse poucas rotações. Mostraremos que o número esperado de rotações executadas é limitado por uma constante.

Para fazê-lo, precisaremos de algumas definições, que estão ilustradas na Figura 13.11. A ***espinha esquerda*** de uma árvore de pesquisa binária T é o caminho desde a raiz até o nó com a menor chave. Em outras palavras, a espinha esquerda é o caminho desde a raiz que consiste apenas em arestas da esquerda. Simetricamente, a ***espinha direita*** de T é o caminho desde a raiz que consiste somente em arestas da direita. O ***comprimento*** de uma espinha é o número de nós que ela contém.

e. Considere que o treap T imediatamente após x é inserido com o uso de TREAP-INSERT. Seja C o comprimento da espinha direita da subárvore esquerda de x. Seja D o comprimento da espinha esquerda da subárvore direita de x. Prove que o número total de rotações que foram executadas durante a inserção de x é igual a $C + D$.

Agora calcularemos os valores esperados de C e D. Sem perda de generalidade, supomos que as chaves são 1, 2, ..., n, pois estamos comparando essas chaves apenas entre si.

Para os nós x e y, onde $y \neq x$, seja $k = chave[x]$ e $i = chave[y]$. Definimos variáveis indicadoras aleatórias

$X_{i,k} = I\{y$ está na espinha direita da subárvore esquerda de x (em $T)\}$.

f. Mostre que $X_{i,k} = 1$ se e somente se $prioridade[y] > prioridade[x]$, $chave[y] < chave[x]$ e, para todo z tal que $chave[y] < chave[z] < chave[x]$, temos $prioridade[y] < prioridade[z]$.

g. Mostre que

$$\Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}.$$

h. Mostre que

$$E[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}.$$

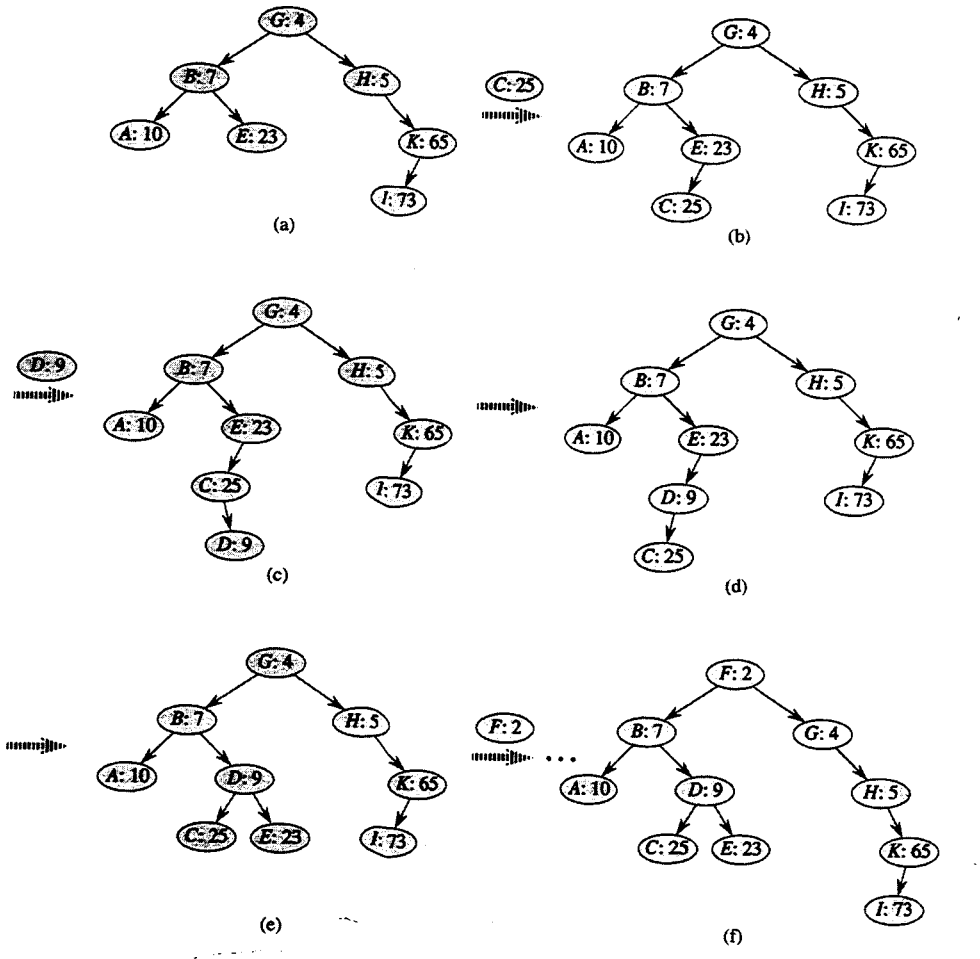


FIGURA 13.10 A operação de TREAP-INSERT. **(a)** O treap original, antes da inserção. **(b)** O treap depois da inserção de um nó com chave C e prioridade 25. **(c)–(d)** As fases intermediárias quando se insere um nó com chave D e prioridade 9. **(e)** O treap depois de terminada a inserção das partes (c) e (d). **(f)** O treap depois da inserção de um nó com chave F e prioridade 2

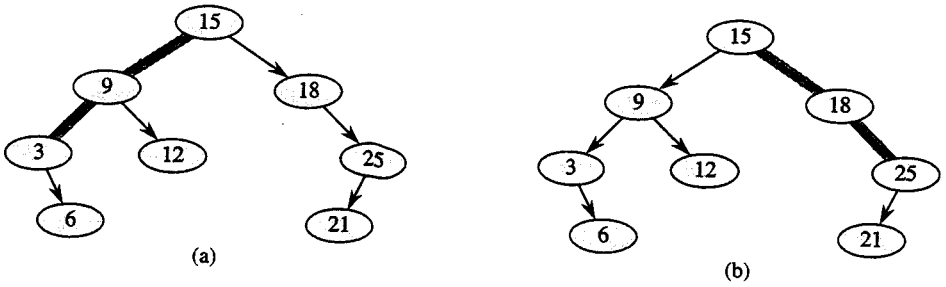


FIGURA 13.11 Espinhas de uma árvore de pesquisa binária. A espinha esquerda está sombreada em (a), e a espinha direita está sombreada em (b)

i. Use um argumento de simetria para mostrar que

$$E[D] = 1 - \frac{1}{n-k+1}.$$

j. Conclua que o número esperado de rotações executadas quando se insere um nó em um treap é menor que 2.

Notas do capítulo

A idéia de balancear uma árvore de pesquisa se deve a Adel'son-Vel'skiĭ e Landis [2], que apresentaram em 1962 uma classe de árvores de pesquisa balanceadas chamada "árvores AVL", descritas no Problema 13-3. Outra classe de árvores de pesquisa, chamadas "árvores 2-3", foi apresentada por J. E. Hopcroft em 1970. Em uma árvore 2-3, o equilíbrio é mantido pela manipulação dos graus de nós na árvore. Uma generalização de árvores 2-3 apresentada por Bayer e McCreight [32], chamadas árvores B, é o tópico do Capítulo 18.

As árvores vermelho-preto foram criadas por Bayer [31] sob o nome "árvores B binárias simétricas". Guibas e Sedgewick [135] estudaram profundamente suas propriedades e introduziram a convenção de cores vermelho/preto. Andersson [15] fornece uma variação de árvores vermelho-preto mais simples de codificar. Weiss [311] chama essa variação de árvores AA. Uma árvore AA é semelhante a uma árvore vermelho-preto, exceto pelo fato de que os filhos da esquerda nunca podem ser vermelhos.

Os treaps foram propostos por Seidel e Aragon [271]. Eles são a implementação padrão de um dicionário em Leda, uma coleção bem implementada de estruturas de dados e algoritmos.

Existem muitas outras variações sobre árvores binárias balanceadas, inclusive as árvores de peso balanceado [230], as árvores de k vizinhos [213] e as árvores de bode expiatório [108]. Talvez as mais curiosas sejam as "árvores oblíquas" introduzidas por Sleator e Tarjan [281], que são "auto-ajustáveis". (Uma boa descrição de árvores oblíquas é dada por Tarjan [292].) As árvores oblíquas mantêm o equilíbrio sem qualquer condição explícita de equilíbrio, como cor. Em vez disso, "operações oblíquas" (que envolvem rotações) são executadas dentro da árvore toda vez que se efetua um acesso. O custo amortizado (consulte o Capítulo 17) de cada operação sobre uma árvore de n nós é $O(\lg n)$.

As listas de saltos [251] constituem uma alternativa às árvores binárias balanceadas. Uma lista de saltos é uma lista ligada que é ampliada com uma série de ponteiros adicionais. Cada operação de dicionário é executada no tempo esperado $O(\lg n)$ em uma lista de saltos de n itens.