# Capítulo 9

# Árvores Rubro-Negras

Neste capítulo apresentamos as árvores rubro-negras, uma versão das árvores binárias de busca com altura logarítmica. As árvores rubro-negras são uma das estruturas de dados mais utilizadas. Elas aparecem como a principal estrutura de busca em muitas implementações de bibliotecas, incluindo a Java Collections Framework e várias implementações da C++ Standard Template Library. Elas também são utilizadas no kernel do sistema operacional Linux. Existem várias razões para a popularidade das árvores rubro-negras:

1. Uma árvore rubro-negra com $n$ elementos tem altura de no máximo $2\log n$.

2. As operações add($x$) e remove($x$) em uma árvore rubro-negra são executadas em tempo $O(\log n)$ no *pior caso*.

3. O número de rotações amortizadas realizadas durante uma operação add($x$) ou remove($x$) é constante.

As duas primeiras dessas propriedades já colocam as árvores rubro-negras à frente de skiplists, treaps, e árvores scapegoat. Skiplists e treaps dependem de randomização, e seus tempos de execução $O(\log n)$ são apenas esperados. As árvores scapegoat têm limite de altura garantido, mas as operações add($x$) e remove($x$) executam apenas em tempo amortizado $O(\log n)$. A terceira propriedade é a cereja do bolo. Ela nos diz que o tempo necessário para adicionar ou remover um elemento $x$ é limitado

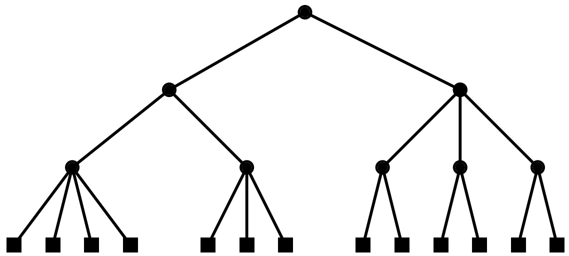

Figura 9.1: Uma árvore 2-4 de altura 3.

pelo tempo necessário para encontrar $x$.[1]

No entanto, as propriedades legais das árvores rubro-negras vêm com um preço: complexidade de implementação. Manter um limite de $2\log n$ na altura não é fácil. Isso exige uma análise cuidadosa de vários casos. Devemos garantir que a implementação faça exatamente a coisa certa em cada caso. Uma mudança de cor ou rotação mal feita produz um bug que pode ser muito difícil de entender e rastrear.

Em vez de pular diretamente para a implementação das árvores rubro-negras, primeiro passaremos alguma base sobre uma estrutura de dados relacionada: a árvore 2-4. Isso dará uma visão de como as árvores rubro-negras foram descobertas e porque a manutenção eficiente delas pode ser possível.

## 9.1 Árvores 2-4

Uma árvore 2-4 é uma árvore enraizada com as seguintes propriedades:

**Propriedade 9.1** (Altura)**.** Todas as folhas têm a mesma profundidade.

**Propriedade 9.2** (grau)**.** Cada nó interno tem 2, 3 ou 4 filhos.

Um exemplo de uma árvore 2-4 é mostrado na Figura 9.1. As propriedades das árvores 2-4 implicam que sua altura é logarítmica no número de folhas:

[1] Note que, nesse sentido, as skiplists e treaps também têm essa propriedade. Veja os exercicios 4.6 e 7.5.

**Lema 9.1.** *Uma árvore 2-4 com n folhas tem altura máxima de* $\log n$.

*Demonstração.* O limite inferior de 2 no número de filhos de um nó interno implica que, se a altura de uma árvore 2-4 for $h$, então ela tem no mínimo $2^h$ folhas. Em outras palavras,

$$n \geq 2^h \quad .$$

Tirando o log de ambos os lados desta desigualdade resulta em $h \leq \log n$. □

### 9.1.1 Adicionando uma folha

Adicionar uma folha a uma árvore 2-4 é fácil (veja Figura 9.2). Se nós queremos adicionar uma folha $u$ como filho de algum nó $w$ no penúltimo nível, então simplesmente fazemos $u$ um filho de $w$. Isso certamente mantém a propriedade da altura, mas poderia violar a propriedade do grau; se $w$ tinha quatro filhos antes de adicionar $u$, então $w$ agora tem cinco filhos. Neste caso, nós *dividimos* $w$ em dois nós, $w$ e $w'$, tendo dois e três filhos, respectivamente. Mas agora $w'$ não tem pai, então, recursivamente, fazemos $w'$ um filho do pai de $w$. Novamente, isso pode fazer com que o pai de $w$ tenha muitos filhos, caso em que o dividimos. Este processo continua até alcançar um nó com menos de quatro filhos, ou até dividir a raiz, $r$, em dois nós $r$ e $r'$. Nesse último caso, criamos uma nova raiz que tem $r$ e $r'$ como filhos. Isso aumenta a profundidade de todas as folhas simultaneamente e, portanto, mantém a propriedade da altura.

Uma vez que a altura da árvore 2-4 nunca é maior que $\log n$, o processo de adicionar uma folha termina após $\log n$ passos, no máximo.

### 9.1.2 Removendo uma folha

Remover uma folha da árvore 2-4 é um pouco mais complicado (veja Figura 9.3). Para remover uma folha $u$ do seu pai $w$, apenas o removemos. Se $w$ tivesse apenas dois filhos antes da remoção de $u$, então $w$ seria deixado com apenas um filho e violaria a propriedade do grau.

Para corrigir isso, olhamos para o irmão de $w$, $w'$. O nó $w'$ certamente existe, dado que o pai de $w$ tenha pelo menos dois filhos. Se $w'$ tem três

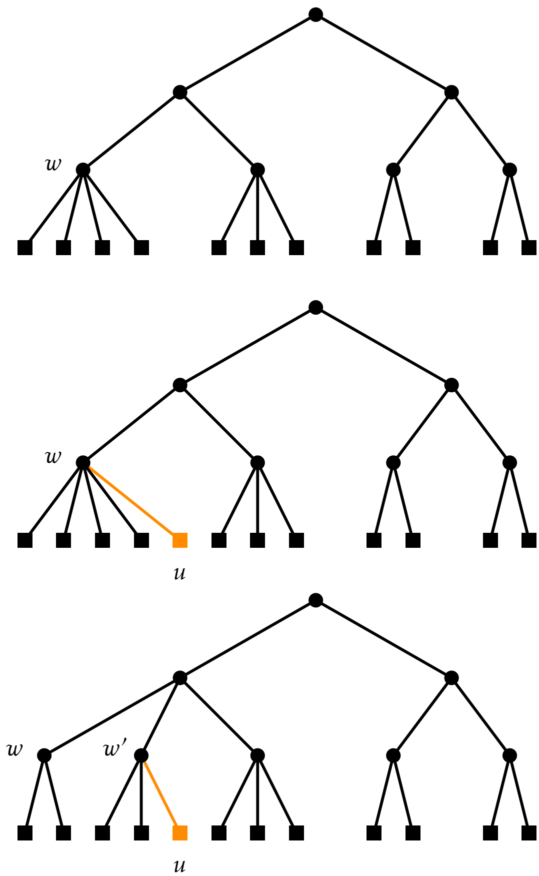


Figura 9.2: Adicionando uma folha na árvore 2-4. Este processo termina depois de uma divisão porque *w.parent* tem um grau inferior a 4 antes da adição.

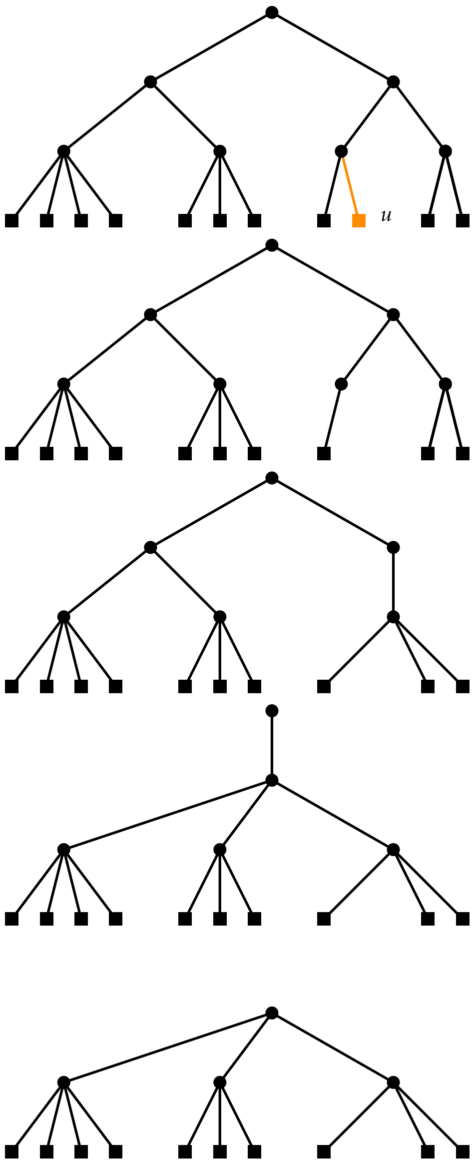


Figura 9.3: Removendo uma folha de uma árvore 2-4. Este processo vai até a raiz porque cada um dos antecessores $u$ e seus irmãos têm somente dois filhos.

ou quatro filhos, então nós levamos um desses filhos de $w'$ para $w$. Agora $w$ tem dois filhos, $w'$ tem dois ou três filhos e assim terminamos.

Por outro lado, se $w'$ tiver apenas dois filhos, então nós *mesclamos* $w$ e $w'$ em um único nó, $w$, que tem três filhos. Em seguida, removemos $w'$ do pai de $w'$ recursivamente. Esse processo acaba quando alcançamos um nó, $u$, onde $u$ ou seu irmão tem mais de dois filhos, ou quando chegamos à raiz. No último caso, se a raiz é deixada com apenas um filho, então nós excluímos a raiz e fazemos do seu filho a nova raiz. Mais uma vez, isso diminui simultaneamente a altura de cada folha e, portanto, mantém a propriedade de altura.

Novamente, uma vez que a altura da árvore nunca é mais de $\log n$, o processo de remoção de uma folha termina após um máximo de $\log n$ passos.

## 9.2 Árvore Rubro-Negra: Uma Árvore 2-4 Simulada

Uma árvore rubro-negra é uma árvore binária de busca na qual cada nó, $u$, tem uma *cor* que é *vermelha* ou *preta*. O vermelho é representado pelo valor 0 e preto pelo valor 1.

Antes e depois de qualquer operação em uma árvore rubro-negra, as duas seguintes propriedades são satisfeitas. Cada propriedade é definida tanto em termos de cores vermelhas e pretas, como em termos dos valores numéricos 0 e 1.

**Propriedade 9.3** (altura preta)**.** Há o mesmo número de nós pretos em cada caminho da raiz para a folha. (A soma das cores em qualquer caminho da raiz para a folha é a mesma.)

**Propriedade 9.4** (sem-borda-vermelha)**.** Dois nós vermelhos nunca são adjacentes. (Para qualquer nó $u$, exceto a raiz, $u.colour + u.parent.colour \geq 1$.)

Observe que sempre podemos colorir a raiz, $r$, de uma árvore rubro-negra sem violar nenhuma dessas duas propriedades, então assumiremos que a raiz é preta, e os algoritmos para atualizar uma árvore rubro-negra irão manter isso. Outro truque que simplifica a árvore rubro-negra é tratar os nós externos (representados por *nil*) como nós pretos. Desta forma,

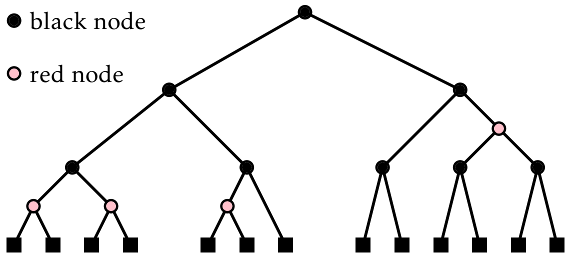


Figura 9.4: Um exemplo de uma árvore rubro-negra com altura preta 3. Os nós externos (*nil*) são desenhados como quadrados.

todo nó real, $u$, de uma árvore rubro-negra tem exatamente dois filhos, cada uma com uma cor bem definida. Um exemplo de árvore rubro-negra é mostrado em Figura 9.4.

### 9.2.1 Árvores Rubro-Negras e Árvores 2-4

No começo, pode parecer surpreendente que uma árvore rubro-negra possa ser eficientemente atualizada para manter as propriedades altura-preta e sem-borda-vermelha, e parece incomum mesmo considerar estas como propriedades úteis. Contudo, árvores rubro-negras foram projetadas para ser uma simulação eficiente de árvores 2-4 como árvores binárias.

Consulte a Figura 9.5. Considere qualquer árvore rubro-negra, $T$, tendo $n$ nós e executando a seguinte transformação: remova cada nó vermelho $u$ e conecte os dois filhos de $u$ diretamente ao pai (preto) de $u$. Após essa transformação nós ficamos com uma árvore $T'$ tendo apenas nós pretos.

Cada nó interno em $T'$ tem dois, três ou quatro filhos: Um nó preto que começou com dois filhos pretos ainda terá dois filhos pretos após essa transformação. Um nó preto que começou com um filho vermelho e um preto terá três filhos depois dessa transformação. Um nó preto que começou com dois filhos vermelhos terá quatro filhos após essa transformação. Além disso, a propriedade altura-preta agora garante que cada caminho da raiz até a folha em $T'$ tem o mesmo comprimento. Em outras palavras, $T'$ é uma árvore 2-4!

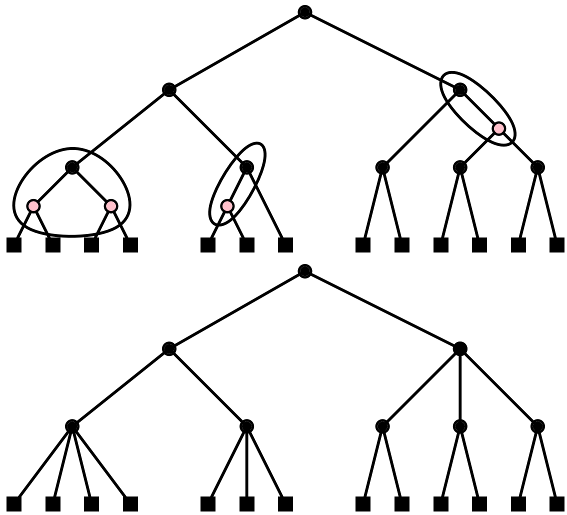

Figura 9.5: Toda árvore rubro-negra possui uma árvore 2-4 correspondente.

A árvore 2-4 $T'$ tem $n+1$ folhas que correspondem aos $n+1$ nós externos da árvore rubro-negra. Portanto, esta árvore tem, no máximo, altura $\log(n+1)$. Agora, cada caminho da raiz até a folha na árvore 2-4 corresponde a uma rota da raiz da árvore rubro-negra $T$ até um nó externo. O primeiro e último nó neste caminho são pretos, e no máximo um, a cada dois nós internos, é vermelho. Então esta rota tem no máximo $\log(n+1)$ nós pretos e no máximo $\log(n+1)-1$ nós vermelhos. Portanto, a rota mais longa da raiz para qualquer nó *interno* em $T$ é no máximo

$$2\log(n+1)-2 \leq 2\log n \quad ,$$

para qualquer $n \geq 1$. Isso prova a propriedade mais importante da árvore rubro-negra:

**Lema 9.2.** *A altura da árvore rubro-negra com n nós é no máximo* $2\log n$.

Agora que vimos a relação entre árvores 2-4 e árvores rubro-negras, não é difícil acreditar que podemos manter, eficientemente, uma árvore rubro-negra ao adicionar e remover elementos.

Já vimos que adicionar um elemento em uma *Árvore Binária de Busca* pode ser feito adicionando uma nova folha. Portanto, para implementar add($x$) em uma árvore rubro-negra, precisamos de um método para simular a divisão de um nó com cinco filhos em uma árvore 2-4. Um nó de árvore 2-4 com cinco filhos é representado por um nó preto que tem dois filhos vermelhos, um dos quais também tem um filho vermelho. Nós podemos dividir esse nó colorindo-o de vermelho e colorindo dois filhos pretos. Um exemplo disso é mostrado em Figura 9.6.

Da mesma forma, implementar o método remove($x$) requer um método de mesclagem de dois nós, e pegar emprestado um filho de um irmão. Mesclar dois nós é o inverso de uma divisão (mostrado em Figura 9.6), e envolve colorir, de vermelho, dois irmãos pretos e colorir, de preto, o pai vermelho. Pegar emprestado de um irmão é o procedimento mais complicado e envolve rotações e recolorações de nó.

Claro, durante todo isso, ainda devemos manter as propriedades sem-borda-vermelha e altura-preta. Embora não seja mais surpreendente que isso possa ser feito, há uma grande quantidade de casos que precisam ser considerados se tentarmos fazer uma simulação direta de uma árvore 2-4 através de uma árvore rubro-negra. Em algum momento, torna-se mais simples ignorar a árvore 2-4 subjacente e trabalhar diretamente para manter as propriedades da árvore rubro-negra.

### 9.2.2 Árvore Rubro-Negra inclinada para a esquerda

Não existe uma definição única de árvores rubro-negras. Em vez disso, existe uma família de estruturas que conseguem manter as propriedades altura-preta e sem-borda-vermelha durante as operações add($x$) e remove($x$). Diferentes estruturas fazem isso de diferentes maneiras. Aqui, implementamos a estrutura de dados que chamamos de RedBlackTree. Esta estrutura implementa uma variante particular das árvores rubro-negras que satisfaz uma propriedade adicional:

**Propriedade 9.5** (inclinada para a esquerda)**.** Em qualquer nó $u$, se *u.left* for preto, então *u.right* é preto.

Observe que a árvore rubro-negra mostrada em Figura 9.4 não satisfaz a propriedade inclinada-para-esquerda; ela é violada pelo pai do nó vermelho no caminho mais à direita.

O motivo para manter a propriedade inclinada-para-esquerda é que

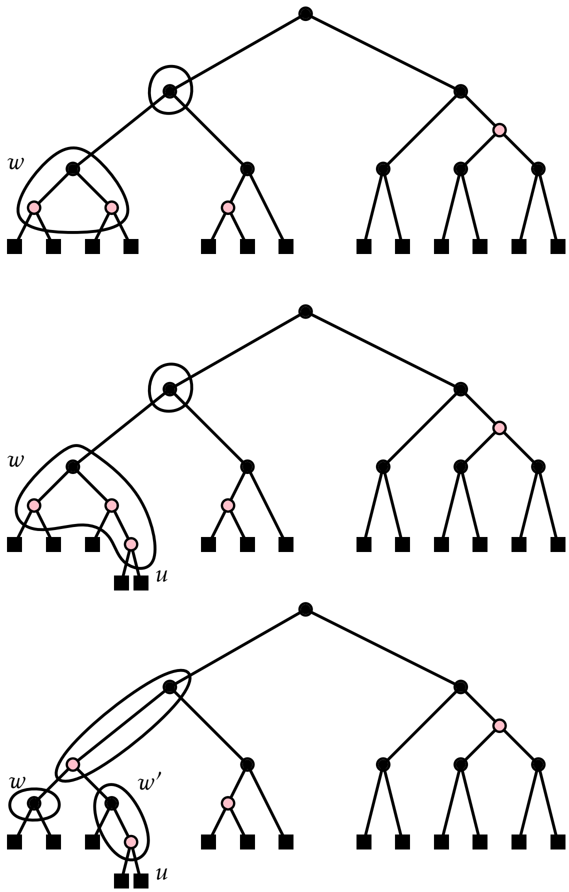


Figura 9.6: Simulando uma operação de divisão na árvore 2-4 durante uma adição em uma árvore rubro-negra. (Isto simula a adição em árvore 2-4 mostrada na Figura 9.2.)

ela reduz o número de casos encontrados ao atualizar a árvore durante add($x$) e remove($x$) operações. Em termos de árvores 2-4, isso implica que cada árvore 2-4 tem uma representação única: um nó de grau dois torna-se um nó preto com dois filhos pretos. Um nó de grau três torna-se um nó preto cujo filho esquerdo é vermelho e cujo filho direito é preto. Um nó de grau quatro torna-se um nó preto com dois filhos vermelhos.

Antes de descrever a implementação de $add(x)$ e remove($x$) em detalhe, apresentamos algumas sub-rotinas simples usadas por esses métodos que estão ilustradas na Figura 9.7. As duas primeiras sub-rotinas são para manipular cores, preservando a propriedade altura-negra. O método push_black($u$) recebe como entrada um nó preto $u$ que tem dois filhos vermelhos e então colore $u$ de vermelho e seus dois filhos de preto. O método pull_black($u$) inverte esta operação:

push_black($u$)
  $u.colour \leftarrow u.colour - 1$
  $u.left.colour \leftarrow u.left.colour + 1$
  $u.right.colour \leftarrow u.right.colour + 1$

pull_black($u$)
  $u.colour \leftarrow u.colour + 1$
  $u.left.colour \leftarrow u.left.colour - 1$
  $u.right.colour \leftarrow u.right.colour - 1$

O método flip_left($u$) troca as cores de $u$ e $u.right$ e então executa uma rotação à esquerda em $u$. Este método inverte as cores desses dois nós, bem como sua relação pai-filho:

flip_left($u$)
  swap_colours($u, u.right$)
  rotate_left($u$)

A operação flip_left($u$) é especialmente útil para restaurar a propriedade inclinada-pra-esquerda em um nó $u$ que a viola (porque $u.left$ é preto

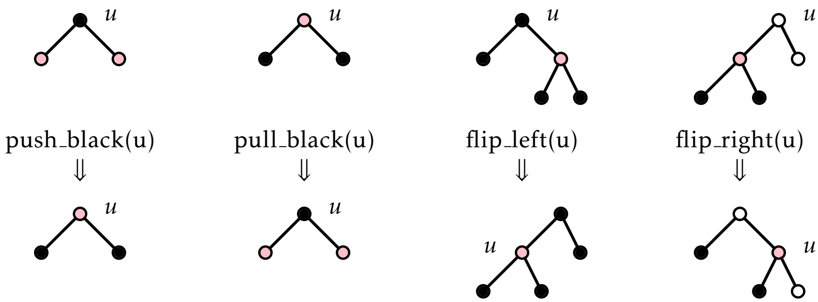


Figura 9.7: Flips, pulls e pushes

e *u.right* é vermelho). Neste caso especial, podemos ter certeza de que esta operação preserva ambas as propriedades altura-preta e sem-borda-vermelha. A operação flip_right($u$) é simétrica com flip_left($u$), quando os papéis da esquerda e direita são invertidos.

```
flip_right(u)
    swap_colours(u, u.left)
    rotate_right(u)
```

### 9.2.3 Adição

Para implementar add($x$) em uma RedBlackTree, nós executamos um inserção padrão em BinarySearchTree para adicionar uma nova folha, $u$, com $u.x = x$ e definir $u.colour = red$. Observe que isso não altera a altura preta de qualquer nó, por isso não viola a propriedade de altura-preta. No entanto, pode violar a propriedade inclinada-para-esquerda (se $u$ é o filho direito de seu pai), e isso pode violar a propriedade sem-borda-vermelha (se o pai de $u$ é *vermelho*). Para restaurar essas propriedades, chamamos o método add_fixup($u$).

```
add(x)
    u ← new_node(x)
    u.colour ← red
```

```
    if add_node(u) then
        add_fixup(u)
        return true
    return false
```

Ilustrado em Figura 9.8, o método add_fixup($u$) recebe como entrada um nó $u$ cuja cor é o vermelho e que pode violar as propriedades sem-borda-vermelha e/ou inclinada-para-esquerda. A seguinte discussão é provavelmente impossível de se explicar sem se referir a Figura 9.8 ou recriá-la em um pedaço de papel. De fato, o leitor talvez deseje estudar essa figura antes de continuar.

Se $u$ é a raiz da árvore, podemos colorir $u$ de preto para restaurar as propriedades. Se o irmão de $u$ também for vermelho, então o pai de $u$ deve ser preto, de modo que as propriedades inclinada-para-esquerda e sem-borda-vermelha se mantenham.

Caso contrário, primeiro determinamos se o pai de $u$, $w$, viola a propriedade inclinada-para-esquerda, e se for o caso, executamos uma operação flip_left($w$) e definimos $u = w$. Isso nos deixa em um estado bem definido: $u$ é o filho esquerdo de seu pai, $w$, então $w$ agora satisfaz a propriedade inclinada-para-esquerda. Tudo o que resta é garantir a propriedade sem-borda-vermelha em $u$. Nós apenas temos que nos preocupar em caso de $w$ ser vermelho, pois em caso contrário, $u$ já satisfaz a propriedade sem-borda-vermelha.

Uma vez que ainda não terminamos, $u$ é vermelho e $w$ é vermelho. A propriedade sem-borda-vermelha (que só é violada por $u$ e não por $w$) implica que o avô de $u$, $g$, existe e é preto. Se o filho direito de $g$ for vermelho, então a propriedade inclinada-para-esquerda garante que ambos os filhos de $g$ são vermelhos, e uma chamada de push_black($g$) faz $g$ vermelho e $w$ preto. Isso restaura a propriedade sem-borda-vermelha em $u$, mas pode fazer com que seja violada em $g$, então todo o processo recomeça com $u = g$.

Se o filho direito de $g$ for preto, então uma chamada para flip_right($g$) faz $w$ o pai (preto) de $g$ e dá a $w$ dois filhos vermelhos, $u$ e $g$. Isso garante que $u$ satisfaça a propriedade sem-borda-vermelha e $g$ satisfaça a propriedade inclinada-para-esquerda. Neste caso, podemos parar.

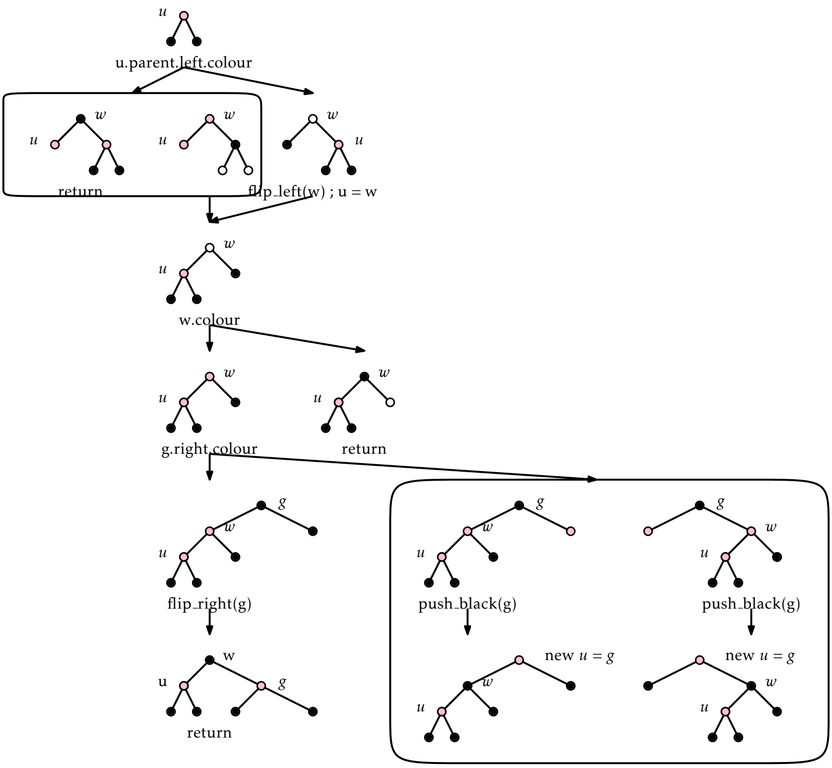


Figura 9.8: Uma única rodada no processo de fixação da propriedade 2 após uma inserção.

```
add_fixup(u)
  while u.colour = red do
    if u = r then
      u.colour ← black
      return
    w ← u.parent
    if w.left.colour = black then
      flip_left(w)
      u ← w
      w ← u.parent
    if w.colour = black then
      return # red-red edge is eliminated - done
    g ← w.parent
    if g.right.colour = black then
      flip_right(g)
      return
    push_black(g)
    u ← g
```

O método insert_fixup($u$) leva tempo constante por iteração e cada iteração termina ou move $u$ para mais perto da raiz. Assim sendo, o método insert_fixup($u$) termina após $O(\log n)$ iterações em tempo $O(\log n)$.

### 9.2.4 Remoção

A operação remove($x$) na RedBlackTree é a mais complicada para implementar, e isso é verdade para todas as variantes conhecidas de árvore rubro-negra. Assim como a operação remove($x$) em uma ÁrvoreBinári-aDeBusca, esta operação resume-se a encontrar um nó $w$ com apenas um filho, $u$, e retirar $w$ da árvore fazendo $w.parent$ adotar $u$.

O problema com isso é que, se $w$ for preto, então a propriedade altura-preta agora será violada em $w.parent$. Podemos evitar esse problema temporariamente adicionando $w.colour$ em $u.colour$. Claro, isso introduz dois outros problemas: (1) se ambos $u$ e $w$ começarem preto, então $u.colour + w.colour = 2$ (duplo preto), que é uma cor inválida. Se $w$ for vermelho, então ele é substituído por um nó preto $u$, que pode violar a propriedade inclinada-para-esquerda em $u.parent$. Ambos os problemas podem ser resolvidos com uma chamada do método remove_fixup($u$).

```
remove(x)
    u ← find_last(x)
    if u = nil or u.x ≠ x then
        return false
    w ← u.right
    if w = nil then
        w ← u
        u ← w.left
    else
        while w.left ≠ nil do
            w ← w.left
        u.x ← w.x
        u ← w.right
    splice(w)
    u.colour ← u.colour + w.colour
    u.parent ← w.parent
    remove_fixup(u)
    return true
```

O método remove_fixup($u$) recebe como entrada um nó $u$ cuja cor é preta (1) ou duplo-preto (2). Se $u$ for duplo-preto, então remove_fixup($u$) executa uma série de rotações e operações de recoloração que movem o nó duplo preto pra cima da árvore até que possa ser eliminado. Durante este processo, o nó $u$ muda até que, no final deste processo, $u$ aponta para a raiz da subárvore que foi alterada. A raiz desta subárvore pode ter mudado de cor. Em particular, pode ter ido de vermelho para preto, então o método remove_fixup($u$) termina ao verificar se o pai de $u$ viola a propriedade inclinada-para-esquerda, e se for o caso, a corrige.

```
remove_fixup(u)
    while u.colour > black do
```

```
    if u = r then
      u.colour ← black
    else if u.parent.left.colour = red
      u ← remove_fixup_case1(u)
    else if u = u.parent.left
      u ← remove_fixup_case2(u)
    else
      u ← remove_fixup_case3(u)
  if u ≠ r then    # restore left-leaning property, if needed
    w ← u.parent
    if w.right.colour = red and w.left.colour = black then
      flip_left(w)
```

O método remove_fixup($u$) é ilustrado em Figura 9.9. Mais uma vez, o seguinte texto será difícil, se não impossível, de se explicar sem se referir a Figura 9.9. Cada iteração do loop em remove_fixup($u$) processa o nó duplo-preto $u$ com base em um de quatro casos:

Caso 0: $u$ é a raiz. Este é o caso mais fácil de tratar. Nós recolorimos $u$ para ser preto (isso não viola nenhuma das propriedades das árvores rubro-negras).

Caso 1: O irmão de $u$, $v$, é vermelho. Nesse caso, o irmão de $u$ é o filho esquerdo de seu pai, $w$ (pela propriedade inclinada-para-esquerda). Nós realizamos um right-flip direito em $w$ e depois prosseguimos para a próxima iteração. Observe que esta ação faz com que o pai de $w$ viole a propriedade inclinada-para-esquerda e a profundidade de $u$ aumente. No entanto, também implica que a próxima iteração estará no Caso 3 com $w$ colorido de vermelho. Ao examinar o Caso 3 abaixo, veremos que o processo irá parar durante a próxima iteração.

```
remove_fixup_case1(u)
  flip_right(u.parent)
  return u
```

Caso 2: O irmão de $u$, $v$, é preto, e $u$ é o filho esquerdo de seu pai, $w$. Nesse caso, chamamos pull_black($w$), fazendo $u$ preto, $v$ vermelho e escu-

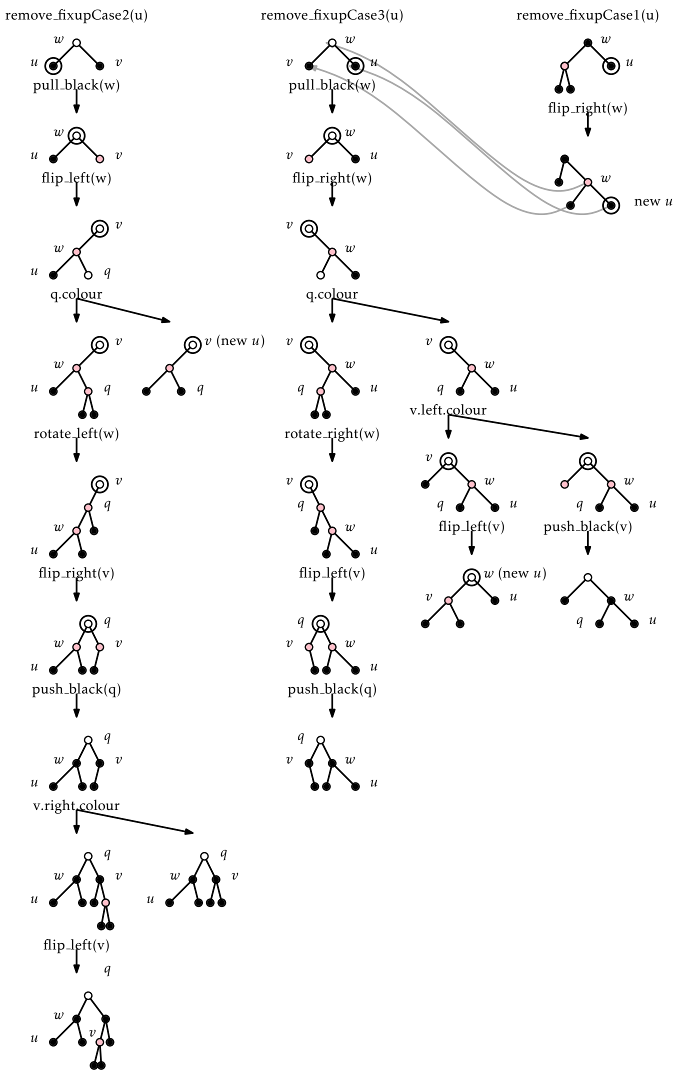


Figura 9.9: Uma única rodada no processo de eliminar um nó duplo-preto após uma remoção.

recendo a cor de $w$ para preto ou duplo-preto. Neste ponto, $w$ não satisfaz a propriedade inclinada-para-esquerda, então nós chamamos flip_left($w$) para corrigir isso.

Neste ponto, $w$ é vermelho e $v$ é a raiz da subárvore com a qual nós começamos. Precisamos verificar se $w$ faz com que a propriedade sem-borda-vermelha seja violada. Fazemos isso inspecionando o filho direito de $w$, $q$. Se $q$ é preto, então $w$ satisfaz a propriedade sem-borda-vermelha e podemos continuar a próxima iteração com $u = v$.

Caso contrário ($q$ é vermelho), tanto a propriedade sem-borda-vermelha quanto a inclinada-para-esquerda são violadas em $q$ e $w$, respectivamente. A propriedade inclinada-para-esquerda é restaurada com uma chamada de rotate_left($w$), mas a propriedade sem-borda-vermelha ainda é violada. Neste ponto, $q$ é o filho esquerdo de $v$, $w$ é o filho esquerdo de $q$, $q$ e $w$ são vermelhos e $v$ é preto ou duplo-preto. A flip_right($v$) faz $q$ o pai de ambos $v$ e $w$. Seguindo com um push_black($q$), colore $v$ e $w$ de preto e define a cor de $q$ de volta para a cor original de $w$.

Neste ponto, o nó duplo-preto foi eliminado e as propriedade sem-borda-vermelha e altura-preta estão restabelecidas. Apenas um problema possível: o filho direito de $v$ pode ser vermelho, caso no qual a propriedade inclinada-para-esquerda seria violada. Verificamos isso e executamos um flip_left($v$) para corrigi-la, se necessário.

```
remove_fixup_case2(u)
    w ← u.parent
    v ← w.right
    pull_black(w)
    flip_left(w)
    q ← w.right
    if q.colour = red then
        rotate_left(w)
        flip_right(v)
        push_black(q)
        if v.right.colour = red then
            flip_left(v)
        return q
    else
```

```
        return v
```

Caso 3: o irmão de $u$ é preto e $u$ é o filho certo de seu pai, $w$. Este caso é simétrico ao Caso 2 e é tratado principalmente da mesma maneira. As únicas diferenças vêm do fato de que a propriedade inclinada-para-esquerda é assimétrica, por isso requer uma manipulação diferente.

Como antes, começamos com uma chamada de pull_black($w$), o que torna $v$ vermelho e $u$ preto. Uma chamada de flip_right($w$) promove $v$ para a raiz da subárvore. Neste ponto $w$ é vermelho, e o código se ramifica de duas maneiras dependendo da cor do filho esquerdo de $w$, $q$.

Se $q$ estiver vermelho, o código termina exatamente da mesma maneira que o Caso 2 termina, mas é ainda mais simples já que não há perigo de $v$ não satifazer a propriedade inclinada-para-esquerda.

O caso mais complicado ocorre quando $q$ é preto. Nesse caso, nós examinamos a cor do filho esquerdo de $v$. Se for vermelho, então $v$ tem dois filhos vermelhos e seu preto extra podem ser postos pra baixo com uma chamada de push_black($v$). Neste ponto, $v$ agora tem a cor original de $w$, e assim terminamos.

Se o filho esquerdo de $v$ for preto, então $v$ viola a propriedade inclinada-para-esquerda, e restauramos isso com uma chamada de flip_left($v$). Depois, devolvemos o nó $v$ para que a próxima iteração de remove_fixup($u$) continue com $u = v$.

```
remove_fixup_case3(u)
    w ← u.parent
    v ← w.left
    pull_black(w)
    flip_right(w) # w is now red
    q ← w.left
    if q.colour = red then # q-w is red-red
        rotate_right(w)
        flip_left(v)
        push_black(q)
        return q
    else
```

```
        if v.left.colour = red then
            push_black(v)
            return v
        else # ensure left-leaning
            flip_left(v)
            return w
```

Cada iteração de remove_fixup($u$) leva tempo constante. Os casos 2 e 3 terminam ou movem $u$ para mais perto da raiz da árvore. O Caso 0 (onde $u$ é a raiz) sempre termina, o e Caso 1 leva imediatamente para Caso 3, que também termina. Como a altura da árvore é de no máximo $2\log n$, concluímos que existem no máximo $O(\log n)$ iterações de remove_fixup($u$), então remove_fixup($u$) é executado em tempo $O(\log n)$.

## 9.3 Resumo

O seguinte teorema resume o desempenho da estrutura de dados RedBlackTree:

**Teorema 9.1.** *Uma RedBlackTree implementa a interface SSet e suporta as operações* add($x$), remove($x$) *e* find($x$) *em tempo de pior caso* $O(\log n)$ *por operação.*

Não incluído no teorema acima é o seguinte bônus extra:

**Teorema 9.2.** *Começando com uma RedBlackTree vazia, qualquer seqüência de $m$ operações* add($x$) *e* remove($x$) *resultam em um tempo total gasto de* $O(m)$ *durante todas as chamadas de* add_fixup($u$) *e* remove_fixup($u$).

Nós apenas esboçamos uma prova do Teorema 9.2. Comparando add_fixup($u$) e remove_fixup($u$) com os algoritmos para adicionar ou remover uma folha em uma árvore 2-4, podemos nos convencer de que essa propriedade é herdada de uma árvore 2-4. Em particular, se pudermos mostrar que o tempo total gasto dividindo, mesclando e pegando emprestado em uma árvore 2-4 é $O(m)$, então isso implica Teorema 9.2.

A prova deste teorema para árvores 2-4 usa o método potencial de análise amortizada.[2] Defina o potencial de um nó interno $u$ em uma árvore 2-4 como

$$\Phi(u) = \begin{cases} 1 & \text{se } u \text{ tem 2 filhos} \\ 0 & \text{se } u \text{ tem 3 filhos} \\ 3 & \text{se } u \text{ tem 4 filhos} \end{cases}$$

e o potencial de uma árvore 2-4 como a soma dos potenciais dos seus nós. Quando ocorre uma divisão, é porque um nó com quatro filhos se torna dois nós, com dois e três filhos. Isso significa que o potencial total cai em $3 - 1 - 0 = 2$. Quando ocorre uma mesclagem, dois nós que tinham dois filhos são substituídos por um nó com três filhos. O resultado é uma queda de $2 - 0 = 2$ no potencial. Portanto, para cada divisão ou mesclagem, o potencial diminui em dois.

Agora perceba que, se ignorarmos a divisão e a mesclagem de nós, existe apenas um número constante de nós cujo número de filhos é alterado pela adição ou remoção de uma folha. Ao adicionar um nó, um nó tem o número de filhos aumentado em um, aumentando o potencial por no máximo três. Durante a remoção de uma folha, um nó tem seu número de filhos diminuido em um, aumentando o potencial por até um, e dois nós podem estar envolvidos em uma operação de pegar emprestado, aumentando seu potencial total por até um.

Para resumir, cada mesclagem e divisão causam o potencial de cair por ao menos dois. Ignorando a mesclagem e a divisão, cada adição ou remoção faz com que o potencial aumente por até três, e o potencial é sempre não negativo. Portanto, o número de divisões e fusões causadas por $m$ adições ou remoções em uma árvore inicialmente vazia é no máximo $3m/2$. Teorema 9.2 é uma conseqüência dessa análise e a correspondência entre árvores 2-4 e árvores rubro-negras.

## 9.4 Discussão e Exercícios

Árvores rubro-negras foram introduzidas pela primeira vez por Guibas e Sedgewick [38]. Apesar da sua alta complexidade de implementação, elas são encontradas em algumas das bibliotecas e aplicações mais utilizadas.

[2] Veja a prova de Lema 2.2 e Lema 3.1 para outras aplicações do método potencial.

A maioria das apostilas de algoritmos e e estruturas de dados discutem algumas variantes de árvores rubro-negras.

Andersson [6] descreve uma versão de árvores balanceadas que são semelhantes às árvores rubro-negras, mas tem a restrição adicional de qualquer nó ter no máximo um filho vermelho. Isso implica que essas árvores simulam árvores 2-3 ao invés de árvores 2-4. Elas são significativamente mais simples, no entanto, que a estrutura RedBlackTree apresentada neste capítulo.

Sedgewick [64] descreve duas versões de árvores rubro-negras inclinada para esquerda. Estas usam a recursão junto com uma simulação de divisão de cima para baixo e mesclando em árvores 2-4. A combinação dessas duas técnicas fazem um código curto e elegante.

Uma estrutura de dados relacionada e mais antiga é a *ávore AVL* [3]. As árvores de AVL são *balanceada em altura*: Em cada nó $u$, as alturas da subárvore enraizada em *u.left* e da subárvore enraizada em *u.right* diferem por mais de um. Segue-se imediatamente que, se $F(h)$ for o número mínimo de folhas em uma árvore de altura $h$, então $F(h)$ obedece a recorrência de Fibonacci

$$F(h) = F(h-1) + F(h-2)$$

com casos base $F(0) = 1$ e $F(1) = 1$. Isso significa que $F(h)$ é aproximadamente $\varphi^h/\sqrt{5}$, onde $\varphi = (1+\sqrt{5})/2 \approx 1.61803399$ é o *coeficiente de ouro*. (Mais precisamente, $|\varphi^h/\sqrt{5} - F(h)| \le 1/2$.) Argumentando como na prova de Lema 9.1, isso implica

$$h \le \log_\varphi n \approx 1.440420088 \log n \ ,$$

então as árvores AVL têm altura menor do que árvores rubro-negras. O balanceamento de altura pode ser mantido durante as operações add($x$) e remove($x$) ao caminhar de volta ao caminho da raiz e realizar uma operação de rebalanceamento em cada nó $u$ onde a altura das subarvores esquerda e direita de $u$ diferem em dois. Veja Figura 9.10.

As variantes de Andersson e de Sedgewick da árvore rubro-negra e as árvores AVL são mais simples de implementar do que a estrutura RedBlackTree definida aqui. Infelizmente, nenhuma delas pode garantir que o tempo amortizado gasto rebalanceamento é $O(1)$ por atualização. Em particular, não há análogo ao Teorema 9.2 para essas estruturas.

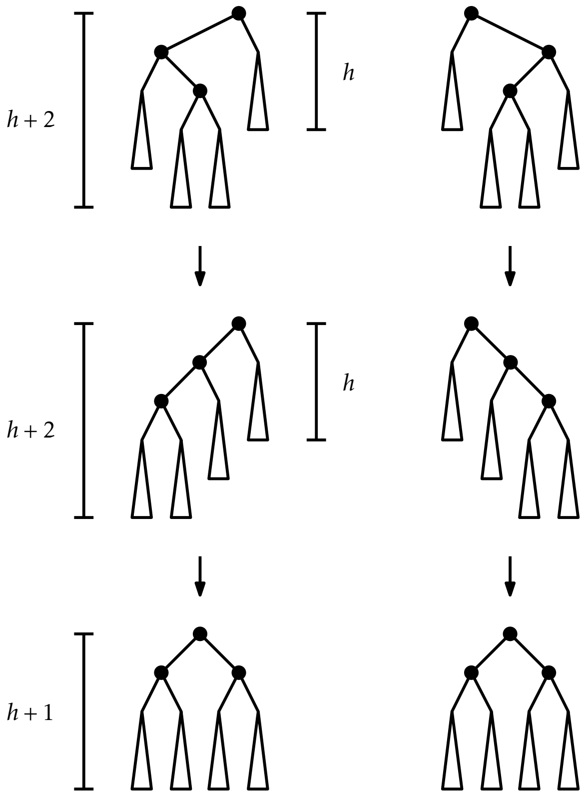


Figura 9.10: Reequilibrando em uma árvore AVL. No máximo, duas rotações são necessárias para converter um nó cujas subárvores tenham uma altura de $h$ e $h+2$ em um nó cujas subárvores têm uma altura de no máximo $h+1$.

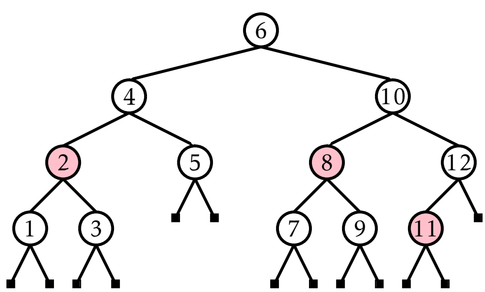


Figura 9.11: Uma árvore rubro-negra para praticar.

**Exercício 9.1.** Ilustre a árvore 2-4 que corresponde à RedBlackTree da Figura 9.11.

**Exercício 9.2.** Ilustre a adição de 13, depois de 3.5, e depois de 3.3 na RedBlackTree da Figura 9.11.

**Exercício 9.3.** Ilustre a remoção de 11, depois de 9, e depois de 5 na RedBlackTree da Figura 9.11.

**Exercício 9.4.** Mostre que, para valores arbitrariamente grandes de $n$, há árvores rubro-negras com $n$ nós com altura de $2\log n - O(1)$.

**Exercício 9.5.** Considere as operações push_black($u$) e pull_black($u$). O que fazem essas operações para a árvore 2-4 subjacente que está sendo simulada pela árvore rubro-negra?

**Exercício 9.6.** Mostre que, para valores arbitrariamente grandes de $n$, existem sequências de operações add($x$) e remove($x$) que ocasionam árvores rubro-negras com $n$ nós de altura $2\log n - O(1)$.

**Exercício 9.7.** Por que o método remove($x$) na implementação RedBlackTree executa a atribuição $u.parent \leftarrow w.parent$? Isso já não deveria estar feito pela chamada de splice($w$)?

**Exercício 9.8.** Suponha que uma árvore 2-4, $T$, tenha $n_\ell$ folhas e $n_i$ nós internos.

1. Qual é o valor mínimo de $n_i$ em função de $n_\ell$?

2. Qual é o valor máximo de $n_i$ em função de $n_\ell$?

3. Se $T'$ é uma árvore rubro-negra que representa $T$, então, quantos nós vermelhos tem $T'$?

**Exercício 9.9.** Suponha que você tenha uma árvore binária de busca com $n$ nós e altura de no máximo $2\log n - 2$. É sempre possível colorir os nós vermelhos e pretos de modo que a árvore satisfaça as propriedades altura-preta e sem-borda-vermelha? Se positivo, também pode ser feito para satisfazer a propriedade inclinada-para-esquerda?

**Exercício 9.10.** Suponha que você tenha duas árvores rubro-negras $T_1$ e $T_2$ de mesma altura preta, $h$, e tal que a maior chave em $T_1$ é menor do que a menor chave em $T_2$. Mostre como combinar $T_1$ e $T_2$ em uma única árvore rubro-negra em tempo $O(h)$.

**Exercício 9.11.** Estenda sua solução para Exercício 9.10 para o caso em que as duas árvores $T_1$ e $T_2$ têm alturas pretas diferentes, $h_1 \neq h_2$. O tempo de execução deve ser $O(\max\{h_1, h_2\})$.

**Exercício 9.12.** Prove que, durante uma operação add($x$), uma árvore AVL deve executar no máximo uma operação de rebalanceamento (que envolve no máximo duas rotações; veja Figura 9.10). Dê um exemplo de uma árvore AVL e uma operação remove($x$) naquela árvore que requer operações de rebalanceamento da ordem de $\log n$

**Exercício 9.13.** Implemente uma classe AVLTree que implementa árvores AVL conforme descrito acima. Compare seu desempenho com o da implementação RedBlackTree. Qual implementação tem uma operação find($x$) mais rápida?

**Exercício 9.14.** Projete e implemente uma série de experimentos que comparam a performance relativa de find($x$), $add(x)$ e $remove(x)$ para as implementações SSet de SkiplistSSet, ScapegoatTree, Treap e RedBlack-Tree. Certifique-se de incluir vários cenários de teste, incluindo casos em que os dados são aleatórios, já ordenado, são removidos em ordem aleatória, são removidos em ordem ordenada, e assim por diante.