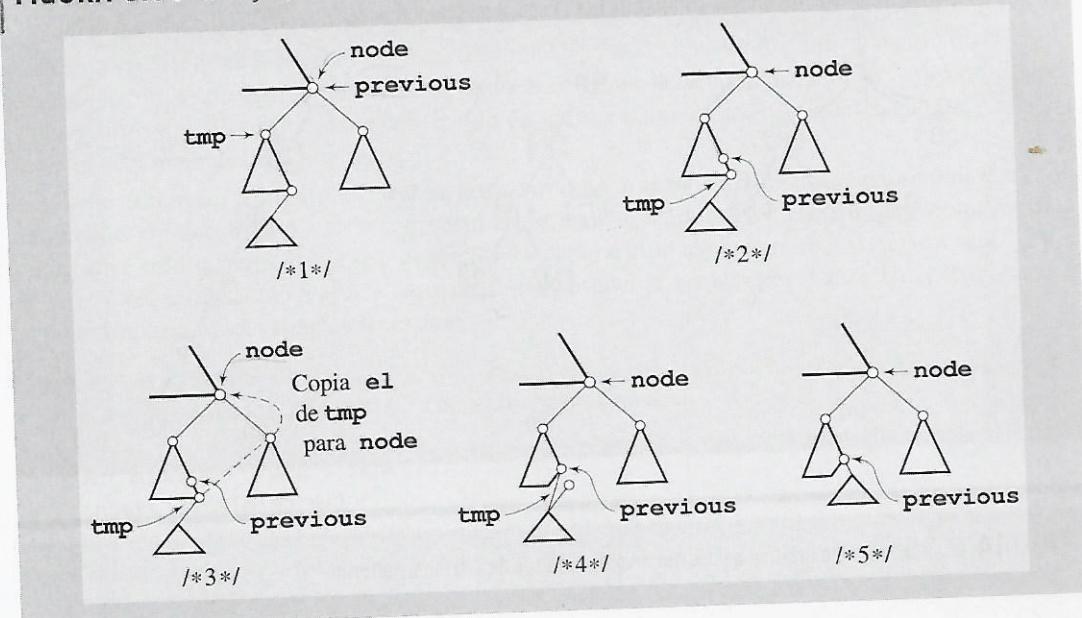


bivariantes, e o resultado da análise classificou-se entre “as mais difíceis de todas as análises exatas de algoritmos que foram realizadas até a data”. Em consequência, a confiança nos resultados experimentais não é surpreendente.

FIGURA 6.33 Remoção por cópia.



6.7 Balanceando uma árvore

No início deste capítulo, dois argumentos favoráveis às árvores foram apresentados; ambos são bem apropriados para representar a estrutura hierárquica de certo domínio, e o processo de busca é muito mais rápido usando árvores do que listas ligadas. O segundo argumento, no entanto, nem sempre se mantém. Tudo depende de como é a árvore. A Figura 6.34 mostra três árvores binárias de busca. Todas armazenam os mesmos dados, mas, obviamente, a da Figura 6.34a é a melhor, e a da 6.34c a pior.

No pior caso, três testes são necessários na primeira, e seis na última para localizar um objeto. O problema com as árvores nas Figuras 6.34b e c é que são algo assimétricas ou aparadas de um lado, isto é, os objetos nelas não são distribuídos uniformemente na medida em que a árvore na Figura 6.34c praticamente se torna uma lista ligada, embora, formalmente, seja ainda uma árvore. Tal situação não aparece em uma árvore balanceada.

Uma árvore binária é *balanceada em altura* ou simplesmente *balanceada* se a diferença na altura de ambas as subárvore de qualquer nó na árvore é zero ou um. Por exemplo, para o nó K na Figura 6.34b, a diferença entre as alturas de suas subárvore ser igual a um é aceitável. Para o nó B, no entanto, esta diferença é três, o que significa que a árvore inteira é desbalanceada. Para o mesmo nó B na 6.34c, a diferença é a pior possível: cinco. Além disso, uma árvore é considerada *perfeitamente balanceada* se é balanceada e todas as folhas se encontram em um ou em dois níveis.

A Figura 6.35 mostra como muitos nós podem ser armazenados em árvores binárias de diferentes alturas. Uma vez que cada nó pode ter dois filhos, o número de nós em certo nível é o dobro do de ascendentes que residem no nível prévio (exceto, naturalmente, a raiz). Por exemplo, se 10.000 elementos são armazenados em uma árvore perfeitamente balanceada, então a árvore é de altura $\lceil \lg(10.001) \rceil = \lceil 13,289 \rceil = 14$. Em termos práticos, isto significa que, se 10.000 elementos são arma-

zenados em uma árvore perfeitamente balanceada, no máximo 14 nós têm que ser verificados para localizar um elemento particular. Esta é uma diferença substancial, comparada aos 10.000 testes necessários em uma lista ligada (no pior caso). Em consequência, é válido o esforço para construir uma árvore balanceada ou modificar uma existente de modo que seja balanceada.

FIGURA 6.34 Diferentes árvores binárias de busca com a mesma informação.

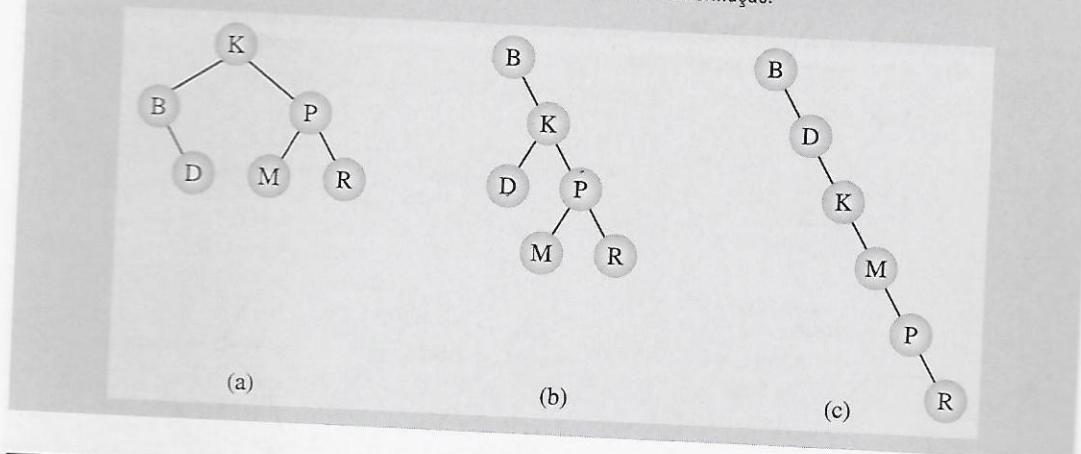


FIGURA 6.35 Número máximo de nós em árvores binárias de diferentes alturas.

Altura	Nós em um nível	Nós em todos os níveis
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 2$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
:		
11	$2^{10} = 1.024$	$2.047 = 2^{11} - 1$
:		
14	$2^{13} = 8.192$	$16.383 = 2^{14} - 1$
:		
h	2^{h-1}	$n = 2^h - 1$
:		

Há um número de técnicas para balancear apropriadamente uma árvore binária. Algumas consistem em reestruturar constantemente a árvore quando os elementos chegam e levam a uma árvore desbalanceada. Outras, em reordenar os próprios dados e então construir uma árvore se uma ordenação dos dados garante que a árvore resultante está平衡ada. Esta seção apresenta uma técnica simples deste tipo.

A árvore parecida com lista ligada da Figura 6.34c é o resultado de uma corrente de dados particular. Assim, se os dados chegam na ordem ascendente ou descendente, a árvore se parece com uma lista ligada. A árvore da Figura 6.34b é desbastada lateralmente porque o primeiro elemento que chega

é a letra B, que precede quase todas as outras letras, exceto A; a subárvore esquerda de B é garantida de ter somente um nó. A árvore da Figura 6.34a parece muito bem, pois a raiz contém um elemento próximo do meio de todos os possíveis elementos, e P está mais ou menos no meio de K e Z. Isto nos leva a um algoritmo baseado na técnica binária de busca.

Quando os dados chegarem, armazene-os em uma matriz. Se todos os possíveis dados chegaram, ordene a matriz usando um dos algoritmos eficientes discutidos no Capítulo 9. Agora, designe para a raiz o elemento do meio na matriz. A matriz agora consiste em duas submatrizes: uma entre o seu início e o elemento escolhido para a raiz e uma entre esta e a extremidade da matriz. O filho à esquerda da raiz é tomado do meio da primeira submatriz, seu filho à direita, um elemento no meio da segunda. Agora, a construção do nível que contém os filhos da raiz está concluída. O próximo nível, com os filhos dos filhos da raiz, é construído da mesma maneira, usando quatro submatrizes e os elementos do meio de cada um deles.

Nesta descrição, primeiro a raiz é inserida em uma árvore inicialmente vazia, depois seu filho à esquerda, então seu filho à direita, e assim por diante, nível por nível. Uma implementação deste algoritmo é muito simplificada se a ordem de inserção é mudada: primeiro, insere-se a raiz, depois seu filho à esquerda, então o filho à esquerda deste à esquerda, e assim por diante. Isto permite usar a seguinte implementação simples recursiva:

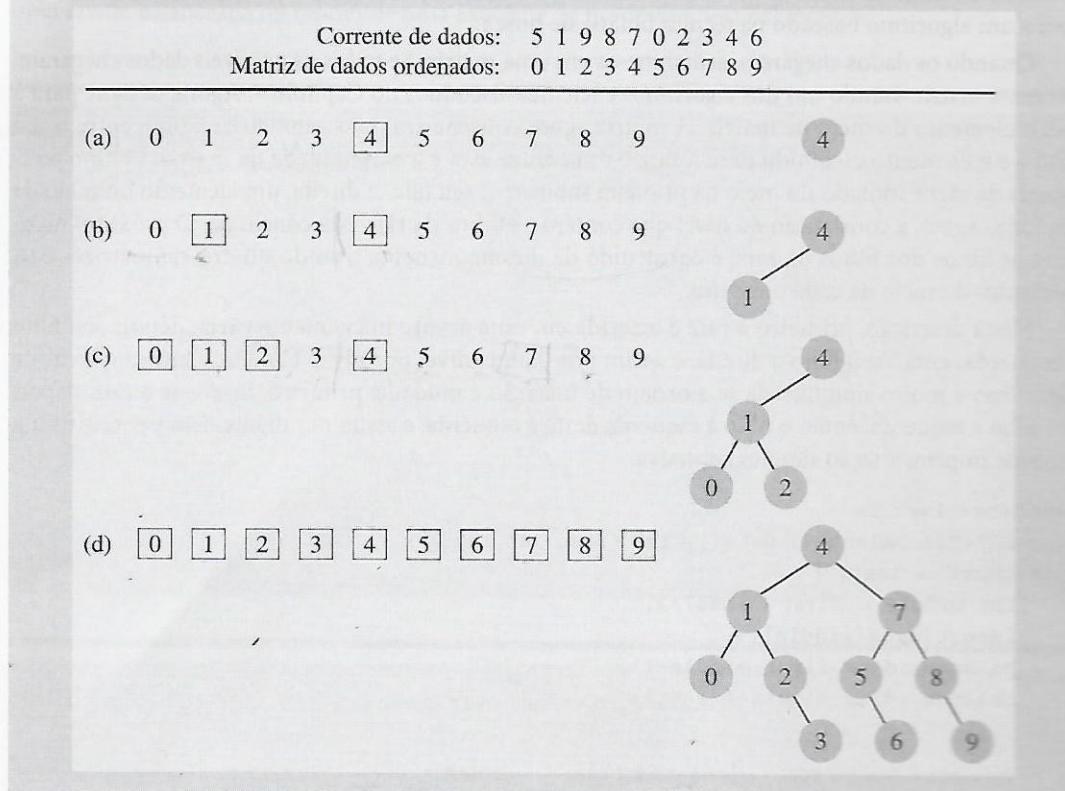
```
template<class T>
void BST<T>::balance(T data[], int first, int last) {
    if (first <= last) {
        int middle = (first + last)/2;
        insert(data[middle]);
        balance (data, first, middle-1);
        balance (data, middle+1, last);
    }
}
```

Um exemplo de aplicação de `balance()` é mostrado na Figura 6.36. Primeiro, o número 4 é inserido (Figura 6.36a), então 1 (Figura 6.36b), depois 0 e 2 (Figura 6.36c) e, finalmente, 3, 7, 5, 6, 8 e 9 (Figura 6.36d).

Este algoritmo tem um sério inconveniente: todos os dados devem ser colocados em uma matriz antes que a árvore possa ser criada; e podem ser armazenados em uma matriz diretamente a partir da entrada. Neste caso, o algoritmo pode ser impróprio quando a árvore tem que ser usada enquanto os dados nela a ser incluídos ainda estão chegando. Mas os dados podem ser transferidos a partir de uma árvore desbalanceada para uma matriz usando o percurso em in-ordem. A árvore pode agora ser removida e recriada usando `balance()`. Isto pelo menos não exige o uso de qualquer algoritmo de ordenação para colocar os dados em ordem.

6.7.1 O Algoritmo DSW

O algoritmo discutido na seção anterior era um tanto ineficiente, pois exigia uma matriz adicional que necessitava ser ordenada antes que a construção de uma árvore perfeitamente balanceada começasse. Para evitar a ordenação, ele exigia a demolição e então a reconstrução da árvore, o que é ineficiente, exceto para árvores relativamente pequenas. Há, no entanto, algoritmos que exigem pequena armazenagem adicional para variáveis intermediárias e o uso de procedimentos sem ordenação. O elegante algoritmo DSW foi idealizado por Colin Day e mais tarde melhorado por Quentin F. Stout e Bette L. Warren.

FIGURA 6.36 Criando uma árvore binária de busca a partir de uma matriz ordenada.

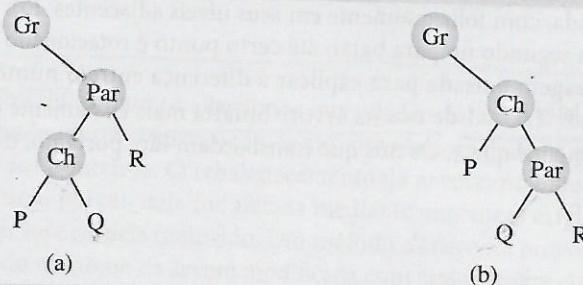
O essencial para as transformações de árvores neste algoritmo é a *rotação*, introduzida por Adel'son-Vel'skii e Landis (1962). Há dois tipos de rotação, a esquerda e a direita, simétricas uma à outra. A rotação à direita do nó Ch com relação ao seu ascendente Par é realizada de acordo com o seguinte algoritmo:

```
rotateRight (Gr, Par, Ch)
  if Par nao e a raiz da arvore // isto e, se Gr nao e nulo
    o avo Gr do filho Ch se torna o ascendente de Ch;
    a subarvore direita de Ch se torna a subarvore esquerda do ascendente Par de Ch;
    o no Ch obtém Par como seu filho à direita;
```

As etapas envolvidas nesta operação composta são mostradas na Figura 6.37. A terceira é o núcleo da rotação, quando Par, o nó ascendente do filho Ch, torna-se o filho de Ch, e os papéis de um ascendente e seu filho se modificam. No entanto, esta troca de papéis não pode afetar a principal propriedade da árvore, ou seja, é uma árvore de busca. A primeira e a segunda etapas de `rotateRight()` são necessárias para assegurar que, depois da rotação, a árvore permaneça uma árvore de busca.

Basicamente, o algoritmo DSW transfigura uma árvore binária arbitrária em uma árvore平衡ada com uma lista ligada chamada *espinha dorsal*. Então, esta árvore alongada é transformada em uma série de passagens em uma árvore perfeitamente balanceada, rotacionando repetidamente cada segundo nó da espinha dorsal ao redor de seu ascendente.

FIGURA 6.37 Rotação à direita do filho Ch ao redor do ascendente Par.

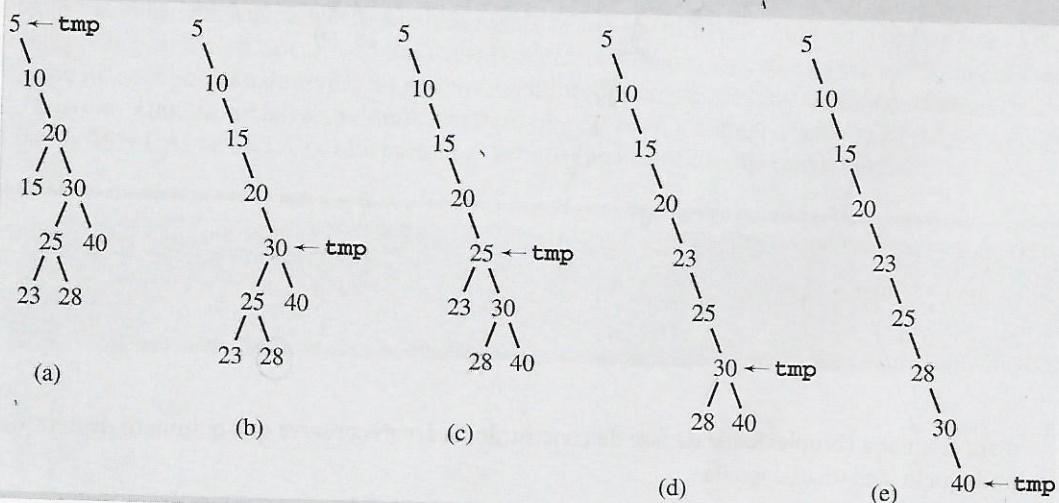


Na primeira fase, uma espinha dorsal é criada usando a seguinte rotina:

```
createBackbone(root)
    tmp = root;
    while (tmp != 0)
        if tmp tem um filho à esquerda
            gire esse filho ao redor de tmp; // por isso o filho à esquerda
                                            // se torna o ascendente de tmp;
            ajuste tmp para o filho que se tornou o ascendente;
        else ajuste tmp para o filho à direita;
```

Este algoritmo está ilustrado na Figura 6.38. Note que uma rotação exige o conhecimento sobre o ascendente de tmp; assim, outro ponteiro tmp que ser mantido quando implementado o algoritmo.

FIGURA 6.38 Transformando uma árvore binária de busca em uma espinha dorsal.



No melhor caso, quando a árvore já é uma espinha dorsal, o laço while é executado n vezes e nenhuma rotação é realizada. No pior, quando a raiz não tem um filho à direita, o laço while executa $2n - 1$ vezes com $n - 1$ rotações realizadas, onde n é o número de nós na árvore. O tempo de execução da primeira fase é $O(n)$. Neste caso, para cada nó, exceto aquele com o menor valor, o filho

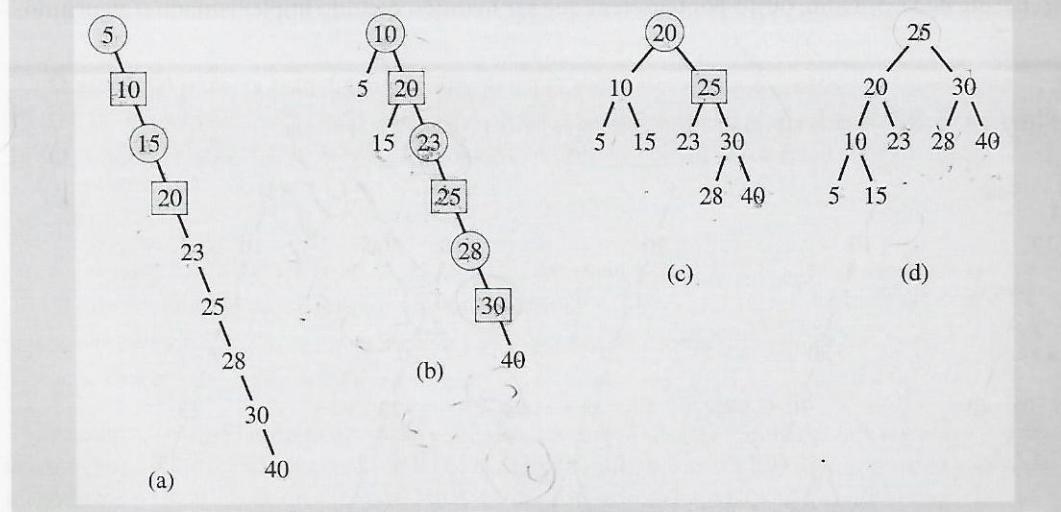
à esquerda de `tmp` é rotacionado ao redor de `tmp`. Depois que todas as rotações terminarem, `tmp` apontará para a raiz `e`, depois de n iterações, ela descerá a espinha dorsal para se tornar nula.

Na segunda fase, a espinha dorsal é transformada em uma árvore, mas, desta vez, a árvore está perfeitamente balanceada, com folhas somente em seus níveis adjacentes. Em cada passe para baixo na espinha dorsal, cada segundo nó para baixo até certo ponto é rotacionado ao redor de seu ascendente. A primeira passagem é usada para explicar a diferença entre o número n de nós na árvore corrente e o número $2^{\lfloor \lg(n+1) \rfloor} - 1$ de nós na árvore binária mais exatamente completa, onde $\lfloor x \rfloor$ é o mais próximo inteiro menor que x . Os nós que transbordam são, portanto, tratados separadamente.

```
createPerfectTree(n)
    n = número de nós;
    m = 2 $^{\lfloor \lg(n+1) \rfloor} - 1$ ;
    faça n-m rotações começando do topo da espinha dorsal;
    while (m > 1)
        m = m/2;
        faça m rotações começando do topo da espinha dorsal;
```

A Figura 6.39 mostra um exemplo. A espinha dorsal na Figura 6.38e tem nove nós e é pré-processada por uma passagem fora do laço para ser transformada na espinha dorsal mostrada na Figura 6.39b. Agora, duas passagens são executadas. Em cada espinha dorsal, os nós a serem promovidos em um nível pelas rotações à esquerda aparecem como quadrados; seus ascendentes, ao redor do qual são girados, são círculos.

FIGURA 6.39 Transformando uma espinha dorsal em uma árvore perfeitamente balanceada.



Para calcular a complexidade da fase de construção da árvore, observe que o número de iterações realizadas pelo laço `while` iguala

$$(2^{\lg(m+1)-1} - 1) + \dots + 15 + 7 + 3 + 1 = \sum_{i=1}^{\lg(m+1)-1} (2^i - 1) = m - \lg(m+1)$$

O número de rotações pode ser dado agora pela fórmula

$$n - m + (m - \lg(m+1)) = n - \lg(m+1) = n - \lfloor \lg(n+1) \rfloor$$

isto é, o número de rotações é $O(n)$. Criar uma espinha dorsal exige também no máximo $O(n)$ rotações, por isso o custo do rebalanceamento global com o algoritmo DSW é ótimo em termos de tempo, porque cresce linearmente com n e exige uma pequena e fixa quantidade de armazenagem.

6.7.2 Árvores AVL

As duas seções anteriores discutiram os algoritmos que rebalancearam a árvore globalmente; cada nó poderia ter sido envolvido no rebalanceamento tanto movendo os dados a partir dos nós ou reatribuindo novos valores aos ponteiros. O rebalanceamento da árvore, no entanto, pode ser realizado localmente se apenas uma porção dela for afetada mediante mudanças exigidas depois que um elemento for inserido na árvore ou dela removido. Um método clássico foi proposto por Adel'son-Vel'skii e Landis, que é celebrado no nome da árvore modificada com este método: a árvore AVL.

Uma árvore AVL (originalmente chamada árvore admissível) é aquela na qual as alturas das subárvores esquerda e direita de cada nó diferem no máximo por um. Por exemplo, todas as árvores da Figura 6.40 são AVL. Os números nos nós indicam os *fatores de balanceamento*, que são as diferenças entre as alturas das subárvores esquerda e direita. Um fator de平衡amento é a altura da subárvore direita menos a altura da subárvore esquerda. Para uma árvore AVL, todos os fatores de balanceamento devem ser +1, 0 ou -1. Note que a definição de árvore AVL é a mesma que a de árvore balanceada; no entanto, o seu conceito sempre inclui implicitamente as técnicas para balanceamento da árvore. Além disso, diferente dos dois métodos previamente discutidos, a técnica para balancear as árvores AVL não garante que a árvore resultante esteja perfeitamente balanceada.

A definição de uma árvore AVL indica que o número mínimo de nós em uma árvore é determinado por uma equação de recorrência

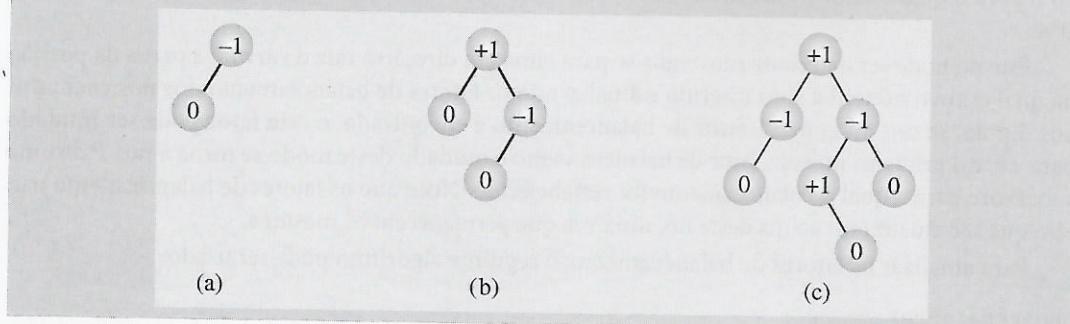
$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$

onde $AVL_0 = 0$ e $AVL_1 = 1$ são as condições iniciais.¹ Esta fórmula leva aos seguintes limites na altura h de uma árvore AVL, dependendo do número de nós n (veja o Apêndice A.5):

$$\lg(n+1) \leq h < 1,44\lg(n+2) - 0,328$$

Em consequência, h é limitado por $O(\lg n)$; a busca de pior caso exige $O(\lg n)$ comparações. Para uma árvore binária perfeitamente balanceada de mesma altura, $h = \lceil \lg(n+1) \rceil$. Por esta razão, o tempo de busca no pior caso em uma árvore AVL é 44% pior (ela exige 44% mais comparações) do que na configuração de árvore do melhor caso. Estudos empíricos indicam que o número médio de busca está muito mais perto do melhor caso que do pior, e é igual a $\lg n + 0,25$ para n grande (Knuth, 1998). As árvores AVL são, portanto, definitivamente estudos de importância.

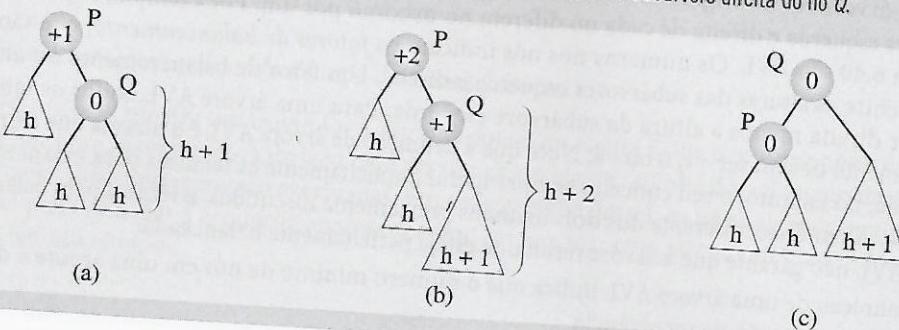
FIGURA 6.40 Exemplos de árvores AVL.



1. Os números gerados por esta fórmula de recorrência são chamados *números de Leonardo*.

Se o fator de balanceamento de qualquer nó em uma árvore AVL se torna menor do que -1 ou maior que 1, a árvore tem que ser balanceada. Uma árvore AVL pode se tornar desbalanceada em quatro situações, mas somente duas necessitam ser analisadas; as outras são simétricas. O primeiro caso, resultado de inserir um nó na subárvore direita do filho à direita, está ilustrado na Figura 6.41. As alturas das subárvores participantes estão indicadas dentro dessas subárvores. Na árvore AVL da Figura 6.41a, um nó é inserido em algum lugar da subárvore direita de Q (Figura 6.41b), o que perturba o balanceamento da árvore P. Neste caso, o problema pode ser facilmente retificado rotacionando o nó Q ao redor de seu ascendente P (Figura 6.41c), de modo que o fator de balanceamento tanto de P quanto de Q se torna zero, o que é ainda melhor do que no princípio.

FIGURA 6.41 Balanceando uma árvore depois da inserção de um nó na subárvore direita do nó Q.



O segundo caso, resultado de inserir um nó na subárvore esquerda do filho à direita, é mais complexo. Um nó é inserido na árvore da Figura 6.42a; a árvore resultante é mostrada na Figura 6.42b e com mais detalhes na Figura 6.42c. Note que o fator de balanceamento de R também pode ser -1. Para trazer a árvore de volta ao平衡amento, uma dupla rotação é realizada. O balanço da árvore P é restaurado rotacionando-se R ao redor do nó Q (Figura 6.42d) e então rotacionando-se R novamente, desta vez ao redor do nó P (Figura 6.42e).

Nestes dois casos, a árvore P é considerada uma árvore única. No entanto, P pode ser parte de uma árvore AVL maior; pode ser um filho de algum outro nó na árvore. Se um nó é inserido na árvore e o balanceamento de P é perturbado e então restabelecido, é necessário trabalho extra com o(s) predecessor(es) de P? Felizmente não. Note que as alturas das árvores nas Figuras 6.41c e 6.42a, resultantes das rotações, são iguais a $h + 2$. Isto significa que o fator de balanceamento do ascendente da nova raiz (Q na Figura 6.41c e R na 6.42a) permanece o mesmo de antes da inserção, e as mudanças feitas na subárvore P são suficientes para restaurar o balanceamento da árvore AVL inteira. O problema é encontrar um nó P para o qual o fator de balanceamento se torne inaceitável depois de um nó ter sido inserido na árvore.

Este nó pode ser detectado movendo-se para cima em direção à raiz da árvore a partir da posição na qual o novo nó tenha sido inserido e atualizando os fatores de balanceamento dos nós encontrados. Então, se um nó com ± 1 fator de balanceamento é encontrado, o este fator pode ser mudado para ± 2 , e o primeiro nó cujo fator de balanceamento é mudado deste modo se torna a raiz P de uma subárvore para a qual o balanceamento foi restabelecido. Note que os fatores de balanceamento não têm que ser atualizados acima deste nó, uma vez que permanecem os mesmos.

Para atualizar os fatores de balanceamento, o seguinte algoritmo pode ser usado:

```
updateBalanceFactors()
    Q = nó que acabou de ser inserido;
    P = antecessor de Q;
```

```

if Q é filho à esquerda de P
    P->balanceFactor--;
else P->balanceFactor++;
while P não é raiz e P->balanceFactor ≠ ±2
    Q = P;
    P = antecessor de P;
    if Q->balanceFactor is 0
        return;
    if Q é filho à esquerda de P
        P->balanceFactor--;
    Else P->balanceFactor++;
    if P->balanceFactor is ±2
        rebalancear a subárvore com raiz em P;

```

Na Figura 6.43a, um caminho é marcado com um fator de balanceamento igual a +1. A inserção de um novo nó no final deste caminho resulta em uma árvore não balanceada (Figura 6.43b), e o balanceamento é restabelecido por uma rotação à esquerda (Figura 6.43c).

Contudo, se os fatores de balanceamento no caminho do recém-inserido nó até a raiz da árvore são todos zero, têm que ser atualizados, mas nenhuma rotação é necessária para quaisquer nós encontrados. Na Figura 6.44a, a árvore AVL tem um caminho de todos os fatores de balanceamento iguais a zero. Depois de um nó ter sido anexado ao fim deste caminho (Figura 6.44b), nenhuma mudança é feita na árvore, exceto para atualizar os fatores de balanceamento de todos os nós ao longo do caminho.

FIGURA 6.42 Balanceando uma árvore depois da inserção de nó na subárvore esquerda do nó Q.

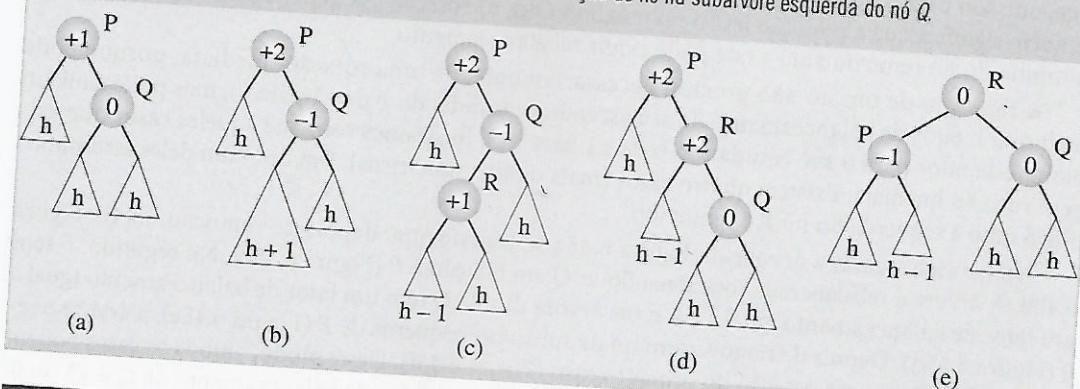


FIGURA 6.43 Um exemplo de inserção de um novo nó (b) em uma árvore AVL (a), que exige uma rotação (c) para restaurar o balanceamento da altura.

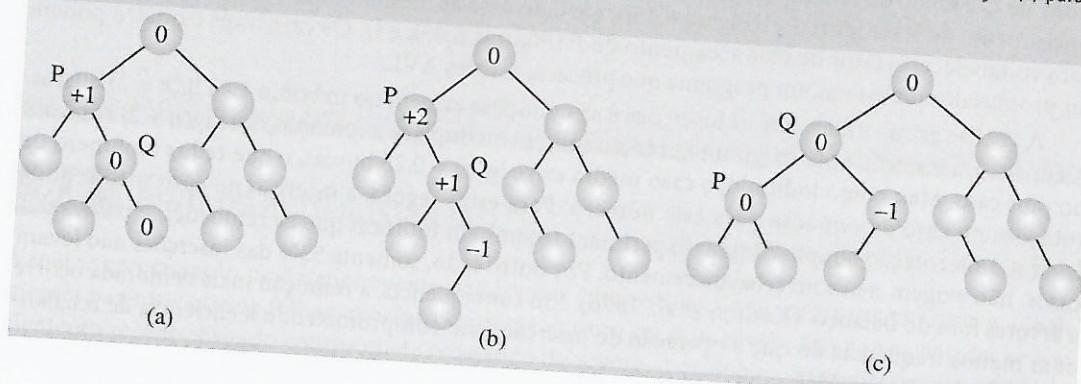
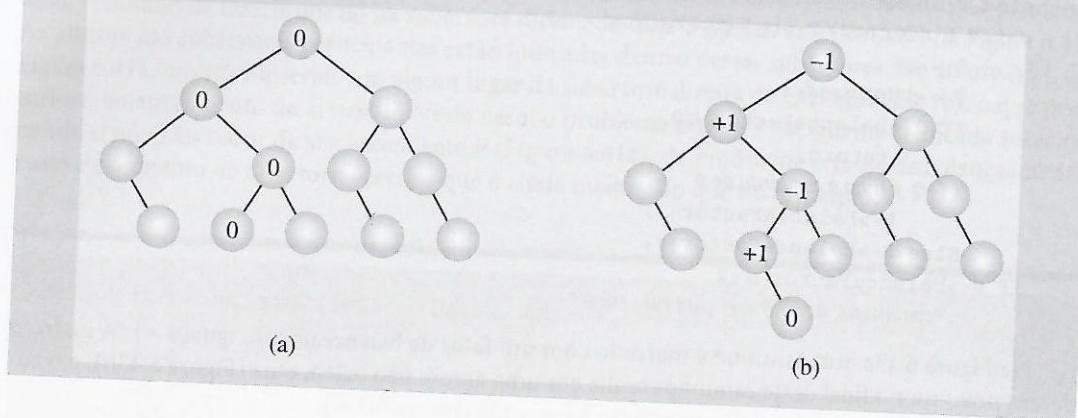


FIGURA 6.44 Em uma árvore AVL, (a) um novo nó é inserido, (b) não exigindo ajuste de altura.



A remoção pode ser mais demorada que a inserção. Primeiro, aplicamos `deleteByCopying()` para remover o nó. Esta técnica nos permite reduzir o problema de remover um nó com dois descendentes para removê-lo com no máximo um descendente.

Depois de um nó ter sido removido da árvore, os fatores de平衡amento são atualizados a partir do ascendente do nó removido até a raiz. Para cada nó nesse caminho cujo fator de balanceamento se torne ± 2 , uma rotação simples ou dupla tem que ser realizada para restabelecer o balanceamento da árvore. Muito importante: o rebalanceamento não para depois que o primeiro nó de P é encontrado para o qual o fator de balanceamento se tornaria ± 2 , como no caso com a inserção. Isto também significa que a remoção leva a no máximo $O(\lg n)$ rotações, já que, no pior caso, cada nó no caminho do nó removido até a raiz pode exigir rebalanceamento.

A remoção de um nó não precisa, necessariamente, de uma rotação imediata, porque pode melhorar o fator de balanceamento do seu ascendente (mudando-o de ± 1 para 0), mas pode também piorar este fator para o avô (mudando-o de ± 1 para ± 2). Ilustramos somente aqueles casos que exigem rotação imediata. Existem quatro casos (mais quatro simétricos). Em cada um deles assumimos que o filho à esquerda do nó P é removido.

No primeiro caso, a árvore da Figura 6.45a se transforma, depois da remoção, na da Figura 6.45b. A árvore é rebalanceada rotacionando-se Q em relação a P (Figura 6.45c). No segundo, P tem um fator de balanceamento igual a $+1$, e sua árvore direita Q tem um fator de balanceamento igual a 0 (Figura 6.45d). Depois de remover um nó na subárvore esquerda de P (Figura 6.45e), a árvore é rebalanceada pela mesma rotação do primeiro caso (Figura 6.45f). Desta forma, ambos os casos podem ser processados juntos em uma implementação depois que o fator de balanceamento de Q é $+1$ ou 0. Se Q é -1 , temos dois outros casos, mais complexos. No terceiro, a subárvore R de Q tem um fator de balanceamento igual a -1 (Figura 6.45g). Para rebalancear a árvore, primeiro R é rotacionado ao redor de Q e então ao redor de P (Figuras 6.45h e i). O quarto difere do terceiro quanto ao fator de balanceamento de R ser igual a $+1$ (Figura 6.45j), em cujo caso as mesmas duas rotações são necessárias para restabelecer o fator de balanceamento de P (Figuras 6.45k e l). Os casos três e quatro podem ser processados juntos em um programa que processe árvores AVL.

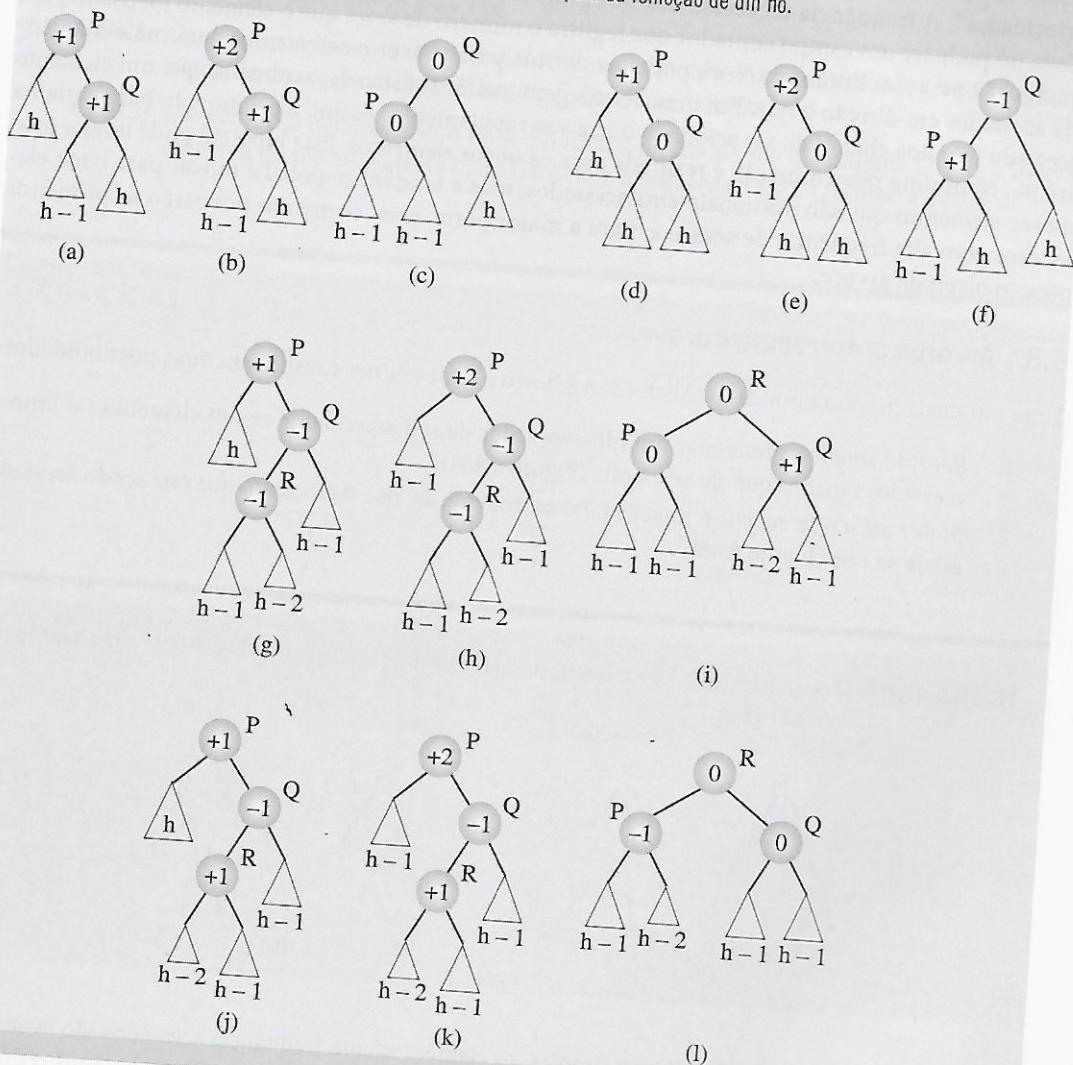
A análise acima indica que as inserções e as remoções exigem no máximo $1,44 \lg(n+2)$ buscas. Além disso, a inserção pode exigir uma rotação simples ou dupla, e a remoção, $1,44 \lg(n+2)$ rotações no pior caso. Mas, como indicado, o caso médio exige $\lg(n) + 0,25$ buscas, o que reduz o número de rotações no caso da remoção para este número. Para estar segura, a inserção no caso médio pode levar a uma rotação simples/dupla. Experimentos também indicam que as remoções, em 78% dos casos, não exigem nenhum rebalanceamento. Por outro lado, somente 53% das inserções não levam a árvores fora do balanço (Karlton et al., 1976). Em consequência, a remoção mais demorada ocorre com menos frequência do que a operação de inserção, não comprometendo a eficiência de rebalanceamento das árvores AVL.

Estas árvores podem ser estendidas permitindo-se diferença na altura $\Delta > 1$ (Foster, 1973). Não é surpresa que a altura do pior caso aumenta com Δ e

$$h = \begin{cases} 1,81 \lg(n) - 0,71 & \text{se } \Delta = 2 \\ 2,15 \lg(n) - 1,13 & \text{se } \Delta = 3 \end{cases}$$

Como os experimentos indicam, o número médio de nós visitados aumenta pela metade na comparação de árvores AVL puras ($\Delta = 1$), mas a quantidade de reestruturação pode ser diminuída por um fator de 10.

FIGURA 6.45 Rebalanceamento de uma árvore AVL depois da remoção de um nó.



6.8

Árvores autoajustadas

A maior preocupação no平衡amento das árvores é impedir-las de se tornar assimétricas e, idealmente, permitir que as folhas ocorram somente em um ou dois níveis. Em consequência, se um elemento recém-chegado compromete o balanço da árvore, o problema é imediatamente retificado