THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

ALGORITMOS

TRADUÇÃO DA 2ª EDIÇÃO AMERICANA

Teoria e Prática

REVISORA TÉCNICA
JUSSARA PIMENTA MATOS
Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da USP e Consultora em Engenharia de Software

TRADUÇÃO
VANDENBERG D. DE SOUZA

6ª Tiragem

ELSEVIER

EDITORA CAMPUS

Do original
Introduction to algorithms – Second Edition
Tradução autorizada do idioma inglês da edição publicada por The MIT Press

Editoração Eletrônica
Estúdio Castellani
Revisão Gráfica
Jane Castellani

Projeto Gráfico
Elsevier Editora Ltda.
A Qualidade da Informação.
Rua Sete de Setembro, 111 – 16º andar
20050-006 Rio de Janeiro RJ Brasil
Telefone: (21) 3970-9300 FAX: (21) 2507-1991
E-mail: *info@elsevier.com.br*
Escritório São Paulo:
Rua Elvira Ferraz, 198
04552-040 Vila Olímpia São Paulo SP
Tel.: (11) 3841-8555

ISBN 85-352-0926-3
(Edição original: ISBN 0-07-013151-1)

CIP-Brasil. Catalogação-na-fonte.
Sindicato Nacional dos Editores de Livros, F.

A385

Algoritmos : teoria e prática / Thomas H. Cormen... [*et al.*]; tradução da segunda edição [americana] Vandenberg D. de Souza. – Rio de Janeiro : Elsevier, 2002 – 6ª Reimpressão.

Tradução de: Introduction to algorithms
ISBN 85-352-0926-3

1. Programação (Computadores). 2. Algoritmos de computador. I. Cormen. Thomas H.

01-1674

CDD – 005.1
CDU – 004.421

04 05 06 07

9 8 7 6

Capítulo 18

Árvores B

As árvores B são árvores de pesquisa balanceadas projetadas para funcionar bem em discos magnéticos ou outros dispositivos de armazenamento secundário de acesso direto. As árvores B são semelhantes às árvores vermelho-preto (Capítulo 13), mas são melhores para minimizar operações de E/S de disco. Muitos sistemas de bancos de dados usam árvores B ou variações de árvores B para armazenar informações.

As árvores B diferem significativamente das árvores vermelho-preto pelo fato de que os nós de árvores B podem ter muitos filhos, desde uma dezena até milhares. Isto é, o "fator de ramificação" de uma árvore B pode ser bastante grande, embora normalmente seja determinado por características da unidade de disco usado. As árvores B são semelhantes às árvores vermelho-preto no fato de que toda árvore B de n nós tem altura $O(\lg n)$, embora a altura de uma árvore B possa ser consideravelmente menor que a altura de uma árvore vermelho-preto, porque seu fator de ramificação pode ser muito maior. Então, as árvores B também podem ser usadas para implementar muitas operações sobre conjuntos dinâmicos no tempo $O(\lg n)$.

As árvores B generalizam árvores de pesquisa binária de maneira natural. A Figura 18.1 mostra uma árvore B simples. Se um nó interno x de uma árvore B contém $n[x]$ chaves, então x tem $n[x] + 1$ filhos. As chaves no nó x são usadas como pontos de divisão que separam o intervalo de chaves manipuladas por x em $n[x] + 1$ subintervalos, cada qual manipulado por um filho de x. Quando procuramos por uma chave em uma árvore B, tomamos uma decisão de $(n[x] + 1)$ modos, com base em comparações com as $n[x]$ chaves armazenadas no nó x. A estrutura de nós de folhas difere da estrutura de nós internos; examinaremos essas diferenças na Seção 18.1.

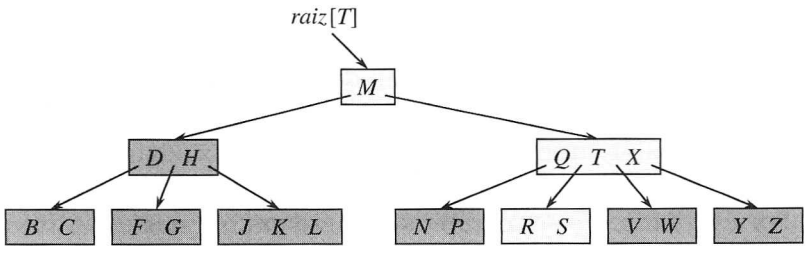

FIGURA 18.1 Uma árvore B cujas chaves são as consoantes do alfabeto. Um nó interno x contendo $n[x]$ chaves tem $n[x] + 1$ filhos. Todas as folhas estão na mesma profundidade na árvore. Os nós levemente sombreados são examinados em uma pesquisa que busca a letra R

A Seção 18.1 fornece uma definição precisa de árvores B e prova que a altura de uma árvore B cresce apenas de forma logarítmica com o número de nós que contém. A Seção 18.2 descreve como procurar por uma chave e inserir uma chave em uma árvore B, e a Seção 18.3 discute a eliminação. Porém, antes de prosseguirmos, precisamos indagar por que estruturas de dados projetadas para o trabalho em um disco magnético são avaliadas de maneira diferente das estruturas de dados projetadas para atuar na memória de acesso aleatório principal.

Estruturas de dados no espaço de armazenamento secundário

Existem muitas tecnologias diferentes disponíveis para fornecer capacidade de memória em um sistema de computador. A ***memória primária*** (ou ***memória principal***) de um sistema de computador consiste normalmente em chips de memória de silício. Essa tecnologia é em geral duas ordens de magnitude mais dispendiosa por bit armazenado que a tecnologia de armazenamento magnético, como fitas ou discos. Um sistema de computador típico tem ***armazenamento secundário*** baseado em discos magnéticos; a quantidade de tal armazenamento secundário com freqüência excede a quantidade de memória primária em pelo menos duas ordens de magnitude.

A Figura 18.2 mostra uma unidade de disco típica. A unidade consiste em várias ***lâminas***, que giram a uma velocidade constante em torno de um ***eixo*** comum. A superfície de cada lâmina é coberta por um material magnetizável. O lâmina é lida ou gravada por uma ***cabeça*** na extremidade de um ***braço***. Os braços estão fisicamente conectados, ou "associados", e podem mover suas cabeças para dentro ou para fora na direção do eixo. Quando uma determinada cabeça está estacionária, a superfície que passa sob ela é chamada uma ***trilha***. As cabeças de leitura/gravação estão alinhadas verticalmente o tempo todo e, portanto, o conjunto de trilhas sob elas é acessado simultaneamente. A Figura 18.2(b) mostra esse conjunto de trilhas, conhecido como ***cilindro***.

Embora os discos sejam mais econômicos e tenham maior capacidade que a memória principal, eles são muito, muito mais lentos, porque têm peças móveis. Há dois componentes para o movimento mecânico: a rotação da lâmina e o movimento do braço. Na época em que esta edição foi escrita, discos comerciais giram a velocidades de 5.400 a 15.000 revoluções por minuto (RPM), sendo 7.200 RPM a mais comum. Apesar de 7.200 RPM parecer rápido, uma rotação demora 8,33 milissegundos, quase cinco ordens de magnitude demorada que os tempos de acesso de 100 nanossegundos comumente encontrados na memória de silício. Em outras palavras, se tivéssemos de esperar uma rotação completa para um item específico cair sob a cabeça de leitura/gravação, poderíamos acessar a memória principal quase 100.000 vezes durante esse período! Em média, temos de esperar apenas por metade de uma rotação mas, ainda assim, a diferença em tempos de acesso para a memória de silício *versus* discos é enorme. A movimentação dos braços também demora algum tempo. Na época em que escrevemos, os tempos médios de acesso para discos comerciais estão no intervalo de 3 a 9 milissegundos.

Para amortizar o tempo gasto na espera por movimentos mecânicos, os discos acessam não apenas um item, mas vários de cada vez. As informações são divididas em diversas ***páginas*** de bits de igual tamanho que apareçam sucessivamente dentro de cilindros, e cada leitura ou gravação de disco inclui uma ou mais páginas inteiras. Para um disco típico, uma página pode ter 2^{11} a 2^{14} bytes de comprimento. Uma vez que a cabeça de leitura/gravação está posicionada corretamente e o disco gira até o início da página desejada, a leitura ou gravação de um disco magnético é inteiramente eletrônica (exceto pela rotação do disco), e grandes quantidades de dados podem ser lidas ou gravadas com rapidez.

Muitas vezes, demora mais tempo para se obter acesso a uma página de informações e fazer a leitura da página de um disco que o tempo necessário para o computador examinar todas as informações lidas. Por essa razão, examinaremos separadamente neste capítulo os dois componentes principais do tempo de execução:

- O número de acessos ao disco.
- O tempo de CPU (ou de computação).

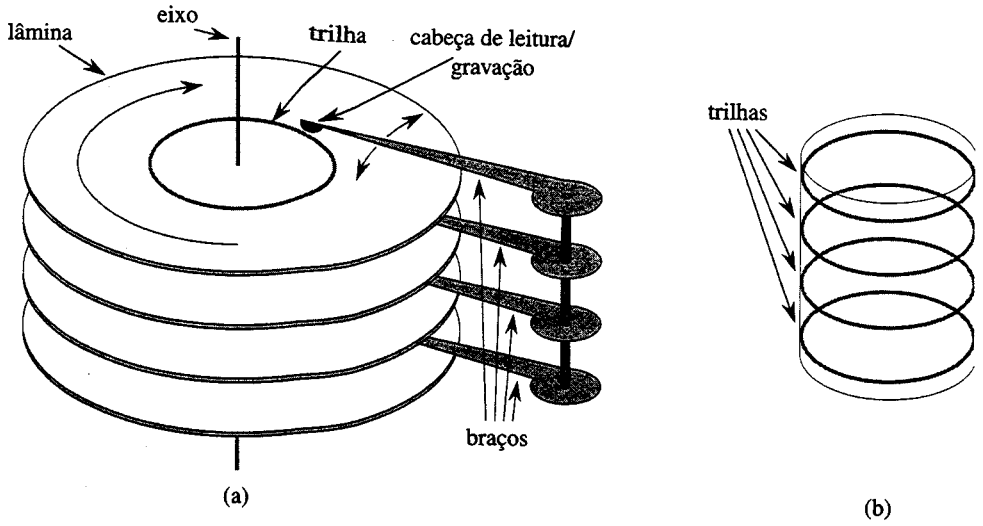

FIGURA 18.2 (a)Uma unidade de disco típica. Ela é composta por várias lâminas que giram em torno de um eixo. Cada lâmina é lida e gravada com uma cabeça na extremidade de um braço. Os braços são associados de modo a moverem suas cabeças em conjunto. Aqui, os braços giram em torno de um eixo pivô comum. Uma trilha é a superfície que passa sob a cabeça de leitura/gravação quando ela é estacionária. (b) Um cilindro consiste em um conjunto de trilhas concêntricas

O número de acessos ao disco é medido em termos do número de páginas de informações que precisam ser lidas do disco ou gravadas nele. Observamos que o tempo de acesso ao disco não é constante – ele depende da distância entre a trilha atual e a trilha desejada, e também do estado inicial de rotação do disco. Contudo, usaremos o número de páginas lidas ou gravadas como uma aproximação inicial bruta do tempo total gasto no acesso ao disco.

Em uma aplicação típica de árvore B, a quantidade de dados manipulados é tão grande que os dados não cabem todos na memória principal de uma só vez. Os algoritmos de árvores B copiam páginas selecionadas do disco para a memória principal conforme necessário e gravam novamente em disco as páginas que foram alteradas. Como os algoritmos de árvores B só precisam de um número constante de páginas na memória principal em qualquer instante, o tamanho da memória principal não limita o tamanho das árvores B que podem ser manipuladas.

Modelamos operações de disco em nosso pseudocódigo da maneira ilustrada a seguir. Seja x um ponteiro para um objeto. Se o objeto estiver atualmente na memória principal do computador, então poderemos fazer referência aos campos do objeto do modo usual: por exemplo, $chave[x]$. Contudo, se o objeto referenciado por x residir no disco, então teremos de executar a operação DISK-READ(x) para ler o objeto x e inseri-lo na memória principal, antes que seus campos possam ser referenciados. (Supomos que, se x já estiver na memória principal, então DISK-READ(x) não exigirá nenhum acesso ao disco; ele será "não operacional".) De modo semelhante, a operação DISK-WRITE(x) é usada para gravar quaisquer alterações que tenham sido efetuadas nos campos do objeto x. Ou seja, o padrão típico de trabalho com um objeto é dado a seguir.

```
x ← um ponteiro para algum objeto
DISK-READ(x)
operações que têm acesso e/ou modificam os campos de x
DISK-WRITE(x)        ▷ Omitida se nenhum campo de x foi alterado.
outras operações que têm acesso mas não modificam campos de x
```

O sistema só pode manter um número limitado de páginas na memória principal em qualquer instante. Presumiremos que as páginas que não estão mais em uso são retiradas da memória principal pelo sistema; nossos algoritmos de árvores B ignorarão essa questão.

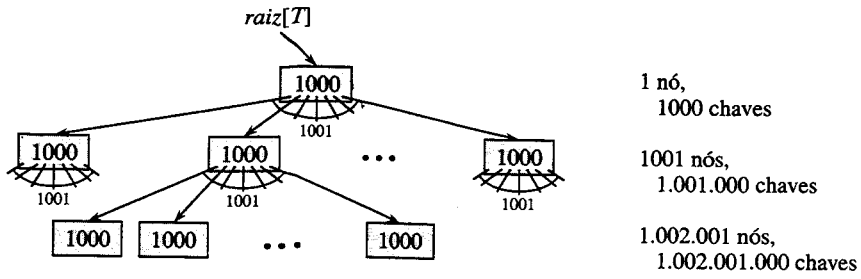

FIGURA 18.3 Uma árvore B de altura 2 contendo mais de um bilhão de chaves. Cada nó interno e cada folha contém 1000 chaves. Existem 1001 nós na profundidade 1 e mais de um milhão de folhas na profundidade 2. Mostramos dentro de cada nó x o valor de $n[x]$, o número de chaves em x

Tendo em vista que, na maioria dos sistemas, o tempo de execução de um algoritmo de árvore B é determinado principalmente pelo número de operações DISK-READ e DISK-WRITE que executa, é sensato usar essas operações de forma eficiente, fazendo-as ler ou gravar o máximo de informações possíveis. Desse modo, um nó de árvore B é normalmente tão grande quanto uma página de disco inteira. O número de filhos que um nó de árvore B pode ter é então limitado pelo tamanho de uma página de disco.

Para uma grande árvore B armazenada em um disco, fatores de ramificação entre 50 e 2000 são usados com freqüência, dependendo do tamanho de uma chave em relação ao tamanho de uma página. Um grande fator de ramificação reduz drasticamente tanto a altura da árvore quanto o número de acessos ao disco necessários para encontrar qualquer chave. A Figura 18.3 mostra uma árvore B com um fator de ramificação igual a 1001 e altura 2 que pode armazenar mais de um bilhão de chaves; não obstante, como o nó de raiz pode ser mantido permanentemente na memória principal, no máximo apenas *dois* acessos ao disco são exigidos para encontrar qualquer chave nessa árvore!

18.1 Definição de árvores B

Para manter tudo em termos simples, supomos – como fizemos no caso das árvores de pesquisa binária e no caso das árvores vermelho-preto – que quaisquer "informações satélite" associadas a uma chave estão armazenadas no mesmo nó em que está a chave. Na prática, realmente seria possível armazenar com cada chave apenas um ponteiro para outra página de disco contendo as informações satélite correspondentes a essa chave. O pseudocódigo deste capítulo pressupõe implicitamente que as informações satélite associadas a uma chave, ou o ponteiro para tais informações satélite, viajam com a chave sempre que a chave é deslocada de um nó até outro nó. Outra organização de árvore B comumente utilizada, conhecida como ***árvore B+***, armazena todas as informações satélite nas folhas e só armazena ponteiros de chaves e filhos nos nós internos, maximizando assim o fator de ramificação dos nós internos.

Uma ***árvore B*** T é uma árvore enraizada (com raiz identificada por $raiz[T]$) que tem as propriedades a seguir.

1. Todo nó x tem os seguintes campos:
   a. $n[x]$, o número de chaves atualmente armazenadas no nó x,
   b. as próprias $n[x]$ chaves, armazenadas em ordem não decrescente, de modo que $chave_1[x] \leq chave_2[x] \leq ... \leq chave_{n[x]}[x]$ e
   c. $folha[x]$, um valor booleano que é TRUE se x é uma folha, e FALSE se x é um nó interno.
2. Cada nó interno x também contém $n[x] + 1$ ponteiros $c_1[x], c_2[x], ..., c_{n[x]+1}[x]$ para seus filhos. Os nós de folhas não têm filhos, e assim seus campos c_i são indefinidos.

3. As chaves $chave_i[x]$ separam os intervalos de chaves armazenadas em cada subárvore: se k_i é qualquer chave armazenada na subárvore com raiz $c_i[x]$, então

$$k_1 \leq chave_1[x] \leq k_2 \leq chave_2[x] \leq \ldots \leq chave_{n[x]}[x] \leq k_{n[x]+1} .$$

4. Toda folha tem a mesma profundidade, que é a altura h da árvore.

5. Existem limites inferiores e superiores sobre o número de chaves que um nó pode conter. Esses limites podem ser expressos em termos de um inteiro fixo $t \geq 2$ chamado ***grau mínimo*** da árvore B:

   a. Todo nó diferente da raiz deve ter pelo menos $t - 1$ chaves. Desse modo, todo nó interno diferente da raiz tem pelo menos t filhos. Se a árvore é não vazia, a raiz deve ter pelo menos uma chave.

   b. Todo nó pode conter no máximo $2t - 1$ chaves. Então, um nó interno pode ter no máximo $2t$ filhos. Dizemos que um nó é ***completo*** se ele contém exatamente $2t - 1$ chaves.[1]

A árvore B mais simples ocorre quando $t = 2$. Todo nó interno tem então 2, 3 ou 4 filhos, e temos uma ***árvore 2-3-4***. Na prática, porém, em geral são utilizados valores de t muito maiores.

A altura de uma árvore B

O número de acessos ao disco exigidos para a maioria das operações em uma árvore B é proporcional à altura da árvore B. Agora, vamos analisar a altura de uma árvore B no pior caso.

Teorema 18.1

Se $n \geq 1$, então, para qualquer árvore B T de n nós de altura h e grau mínimo $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2} .$$

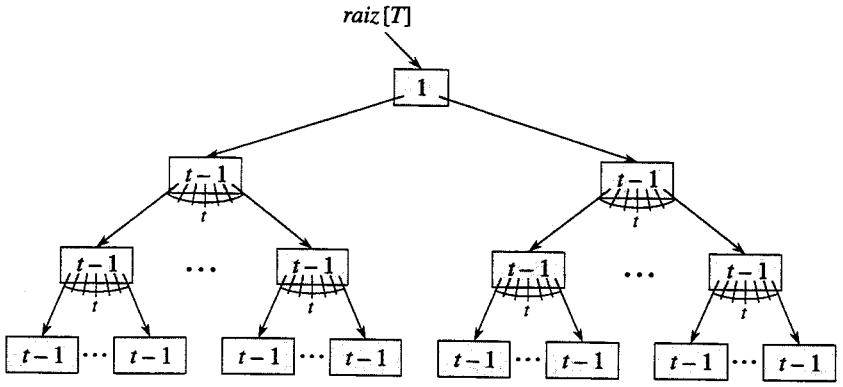

FIGURA 18.4 Uma árvore B de altura 3 contendo um número mínimo possível de chaves. Mostramos dentro de cada nó x o valor de $n[x]$

[1] Outra variante comum em uma árvore B, conhecida como uma ***árvore B****, exige que cada nó interno esteja pelo menos 2/3 completo, em vez de pelo menos metade completo, como uma árvore B exige.

Prova Se uma árvore B tem altura h, o número de seus nós é minimizado quando a raiz contém uma chave e todos os outros nós contêm $t-1$ chaves. Nesse caso, existem 2 nós na profundidade 1, $2t$ nós na profundidade 2, $2t^2$ nós na profundidade 3 e assim por diante, até a profundidade h, em que existem pelo menos $2t^{h-1}$ nós. A Figura 18.4 ilustra tal árvore para $h = 3$. Desse modo, o número n de chaves satisfaz à desigualdade

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 . \end{aligned}$$

Por álgebra simples, obtemos $t^h \leq (n+1)/2$. O uso de logaritmos de base t de ambos os lados prova o teorema. ■

Aqui vemos a capacidade das árvores B, quando comparadas a árvores vermelho-preto. Embora a altura da árvore cresça na proporção $O(\lg n)$ em ambos os casos (lembre-se de que t é uma constante), para as árvores B, a base do logaritmo pode ser muitas vezes maior. Desse modo, as árvores B poupam um fator de aproximadamente $\lg t$ sobre as árvores vermelho-preto no número de nós examinados para a maioria das operações de árvores. Tendo em vista que o exame de um nó arbitrário em uma árvore normalmente exige um acesso ao disco, o número de acessos ao disco é substancialmente reduzido.

Exercícios

18.1-1
Por que não permitimos um grau mínimo $t = 1$?

18.1-2
Para que valores de t a árvore da Figura 18.1 é uma árvore B válida?

18.1-3
Mostre todas as árvores B válidas de grau mínimo 2 que representam $\{1, 2, 3, 4, 5\}$.

18.1-4
Como uma função do grau mínimo t, qual é o número máximo de chaves que podem ser armazenadas em uma árvore B de altura h?

18.1-5
Descreva a estrutura de dados que resultaria se cada nó preto em uma árvore vermelho-preto absorvesse seus filhos vermelhos, incorporando os filhos de seus filhos a seus próprios filhos.

18.2 Operações básicas sobre árvores B

Nesta seção, apresentamos os detalhes das operações B-TREE-SEARCH, B-TREE-CREATE e B-TREE-INSERT. Nesses procedimentos, adotamos duas convenções:

- A raiz da árvore B está sempre na memória principal, de forma que uma operação DISK-READ na raiz nunca é exigida; porém, uma operação DISK-WRITE da raiz é exigida sempre que o nó de raiz é modificado.

- Quaisquer nós repassados como parâmetros já devem ter tido uma operação DISK-READ executada sobre eles.

Os procedimentos que apresentamos são todos algoritmos de "uma passagem" que prosseguem em sentido descendente a partir da raiz da árvore, sem terem de subir de volta.

Pesquisa em uma árvore B

Pesquisar em uma árvore B é muito semelhante a pesquisar em uma árvore de pesquisa binária, exceto pelo fato de que, em vez de tomar uma decisão de ramificação binária ou de "duas vias" em cada nó, tomamos uma decisão de ramificação de várias vias, de acordo com o número de filhos do nó. Mais precisamente, em cada nó interno x, tomamos uma decisão de ramificação de $(n[x] + 1)$ vias.

B-TREE-SEARCH é uma generalização direta do procedimento TREE-SEARCH definido para árvores de pesquisa binária. B-TREE-SEARCH toma como entrada um ponteiro para o nó de raiz x de uma subárvore e uma chave k a ser pesquisada nessa subárvore. A chamada de nível superior é portanto da forma B-TREE-SEARCH($raiz[T]$, k). Se k está na árvore B, B-TREE-SEARCH retorna o par ordenado (y, i) consistindo em um nó y e um índice i tal que $chave_i[y] = k$. Caso contrário, o valor NIL é retornado.

```
B-TREE-SEARCH(x, k)
1 i ← 1
2 while i ≤ n[x] e k > chave_i[x]
3    do i ← i + 1
4 if i ≤ n[x] e k = chave_i[x]
5    then return (x, i)
6 if folha[x]
7    then return NIL
8    else DISK-READ(c_i[x])
9         return B-TREE-SEARCH(c_i[x], k)
```

Usando um procedimento de pesquisa linear, as linhas 1 a 3 encontram o menor i tal que $k \leq chave_i[x]$, ou então definem i como $n[x] + 1$. As linhas 4 e 5 verificam se agora descobrimos a chave, retornando se tivermos descoberto. As linhas 6 a 9 encerram uma pesquisa malsucedida (se x é uma folha) ou usam a recursão para pesquisar a subárvore apropriada de x, depois de executar a necessária operação DISK-READ sobre esse filho.

A Figura 18.1 ilustra a operação de B-TREE-SEARCH; os nós levemente sombreados são examinados durante uma pesquisa pela chave R.

Como no procedimento TREE-SEARCH para árvores de pesquisa binária, os nós encontrados durante a recursão formam um caminho descendente desde a raiz da árvore. O número de páginas de disco às quais B-TREE-SEARCH tem acesso é portanto $\Theta(h) = \Theta(\log, n)$, onde h é a altura da árvore B e n é o número de chaves na árvore B. Tendo em vista que $n[x] < 2t$, o tempo tomado pelo loop **while** das linhas 2 e 3 dentro de cada nó é $O(t)$, e o tempo total da CPU é $O(th) = O(t \log_t n)$.

Como criar uma árvore B vazia

Para construir uma árvore B, primeiro utilizamos B-TREE-CREATE para criar um nó de raiz vazio, e depois chamamos B-TREE-INSERT para adicionar novas chaves. Esses dois procedimentos usam um procedimento auxiliar ALLOCATE-NODE, que aloca uma página de disco para ser usada como um novo nó no tempo $O(1)$. Podemos presumir que um nó criado por ALLOCATE-NODE não exige nenhuma operação DISK-READ, pois ainda não existe nenhuma informação útil armazenada no disco para esse nó.

```
B-TREE-CREATE(T)
1 x ← ALLOCATE-NODE( )
2 folha[x] ← TRUE
3 n[x] ← 0
4 DISK-WRITE(x)
5 raiz[T] ← x
```

B-TREE-CREATE exige $O(1)$ operações de disco e o tempo de CPU $O(1)$.

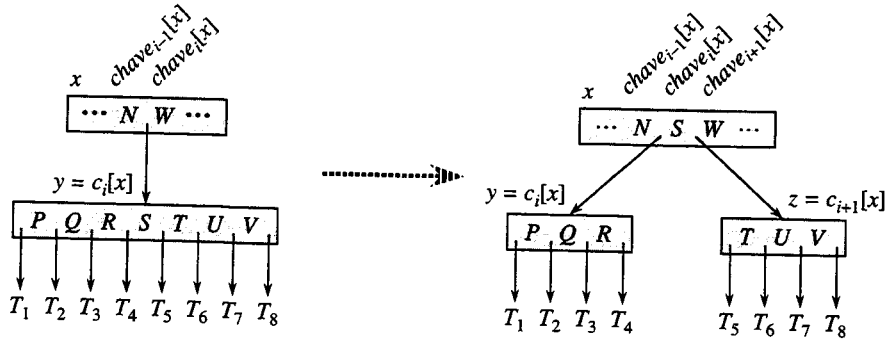

FIGURA 18.5 A divisão de um nó com $t = 4$. O nó y é dividido em dois nós, y e z, e a chave mediana S de y é movida para cima até o pai de y

A inserção de uma chave em uma árvore B

Inserir uma chave em uma árvore B é significativamente mais complicado que inserir uma chave em uma árvore de pesquisa binária. Como ocorre no caso das árvores de pesquisa binária, procuramos pela posição de folha em que devemos inserir a nova chave. Porém, como em uma árvore B, não podemos simplesmente criar um novo nó de folha e inseri-lo, pois a árvore resultante deixaria de ser uma árvore B válida. Em vez disso, inserimos a nova chave em um nó de folha existente. Tendo em vista que não podemos inserir uma chave em um nó de folha completo, introduzimos uma operação que ***divide*** um nó completo y (que tem $2t - 1$ chaves) ao redor de sua ***chave mediana*** $chave_t[y]$ em dois nós que têm $t - 1$ chaves cada. A chave mediana se desloca para cima até o pai de y para identificar o ponto de divisão entre as duas novas árvores. Contudo, se o pai de y também está completo, ele deve ser dividido antes de ser possível inserir a nova chave e, portanto, essa necessidade de dividir nós completos pode se propagar para cima por toda a árvore.

Como no caso de uma árvore de pesquisa binária, podemos inserir uma chave em uma árvore B em uma única passagem para baixo na árvore, desde a raiz até uma folha. Para fazer isso, não esperamos descobrir se realmente precisaremos dividir um nó completo a fim de fazer a inserção. Em vez disso, à medida que descemos a árvore procurando pela posição à qual pertence a nova chave, dividimos cada nó completo que encontramos pelo caminho (inclusive a própria folha). Desse modo, sempre que queremos dividir um nó completo y, temos a certeza de que seu pai não é completo.

A divisão de um nó em uma árvore B

O procedimento B-TREE-SPLIT-CHILD toma como entrada um nó interno *não completo* x (que se presume estar na memória principal), um índice i e um nó y (que também se presume estar na memória principal) tal que $y = c_i[x]$ é um filho *completo* de x. Então, o procedimento divide esse filho em dois e ajusta x de modo que ele tenha agora um filho adicional. (Para dividir uma raiz completa, primeiro transformaremos a raiz em um filho de um novo nó raiz vazio, de modo a podermos usar B-TREE-SPLIT-CHILD. Assim, a árvore cresce uma unidade em altura; a divisão é o único meio pelo qual a árvore cresce.)

A Figura 18.5 ilustra esse processo. O nó completo y é dividido aproximadamente em sua chave mediana S, que é deslocada para cima até o nó x pai de y. As chaves em y que são maiores que a chave mediana são inseridas em um novo nó z, o qual se torna um novo filho de x.

```
B-TREE-SPLIT-CHILD(x, i, y)
 1 z ← ALLOCATE-NODE( )
 2 folha[z] ← folha[y]
 3 n[z] ← t - 1
 4 for j ← 1 to t - 1
 5    do chave_j[z] ← chave_{j + t}[y]
 6 if not folha[y]
 7    then for j ← 1 to t
 8            do c_j[z] ← c_{j + t}[y]
 9 n[y] ← t - 1
10 for j ← n[x] + 1 downto i + 1
11    do c_{j + 1}[x] ← c_j[x]
12 c_{i + 1}[x] ← z
13 for j ← n[x] downto i
14    do chave_{j + 1}[x] ← chave_j[x]
15 chave_i[x] ← chave_t[y]
16 n[x] ← n[x] + 1
17 DISK-WRITE(y)
18 DISK-WRITE(z)
19 DISK-WRITE(x)
```

B-TREE-SPLIT-CHILD funciona pelo método direto de "recortar e colar". Aqui, y é o i-ésimo filho de x e é o nó que está sendo dividido. O nó y tem originalmente $2t$ filhos ($2t - 1$ chaves), mas é reduzido a t filhos ($t - 1$ chaves) por essa operação. O nó z "adota" os t maiores filhos ($t - 1$ chaves) de y, e z se torna um novo filho de x, posicionado logo após y na tabela de filhos de x. A chave mediana de y se desloca para cima até se tornar a chave em x que separa y e z.

As linhas 1 a 8 criam o nó z e dão a ele as $t - 1$ chaves maiores e os t filhos correspondentes de y. A linha 9 ajusta a contagem de chaves para y. Finalmente, as linhas 10 a 16 inserem z como um filho de x, movem a chave mediana de y para cima até x, a fim de separar y de z, e ajustam a contagem de chaves de x. As linhas 17 a 19 gravam todas as páginas de disco modificadas. O tempo de CPU usado por B-TREE-SPLIT-CHILD é (t), devido aos loops nas linhas 4-5 e 7-8. (Os outros loops são executados para $O(t)$ iterações.) O procedimento executa $O(1)$ operações de disco.

Inserção de uma chave em uma árvore B em uma única passagem pela árvore

A inserção de uma chave k em uma árvore B denominada T de altura h é feita em uma única passagem descendente na árvore, exigindo $O(h)$ acessos ao disco. O tempo de CPU exigido é $O(th) = O(t \log_t n)$. O procedimento B-TREE-INSERT utiliza B-TREE-SPLIT-CHILD para garantir que a recursão nunca descerá até um nó completo.

```
B-TREE-INSERT(T, k)
1 r ← raiz[T]
2 if n[r] = 2t - 1
3    then s ← ALLOCATE-NODE( )
4         raiz[T] ← s
5         folha[s] ← FALSE
6         n[s] ← 0
```

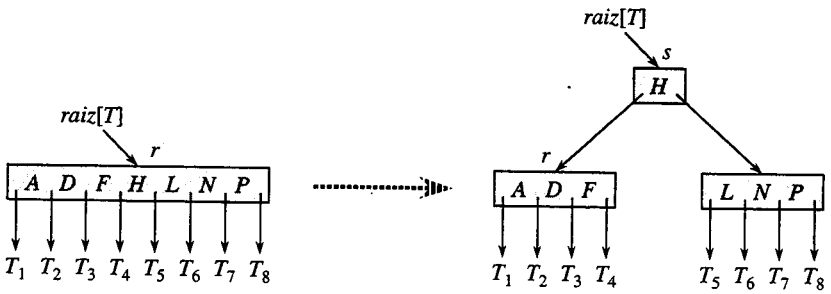

FIGURA 18.6 Divisão da raiz com $t = 4$. O nó de raiz r é dividido em dois, e é criado um novo nó de raiz s. A nova raiz contém a chave mediana de r e tem as duas metades de r como filhos. A árvore B cresce em altura uma unidade quando a raiz é dividida

```
7       c1[s] ← r
8       B-TREE-SPLIT-CHILD(s, 1, r)
9       B-TREE-INSERT-NONFULL(s, k)
10    else B-TREE-INSERT-NONFULL(r, k)
```

As linhas 3 a 9 tratam o caso no qual o nó de raiz r é completo: a raiz é dividida e um novo nó s (que tem dois filhos) se torna a raiz. A divisão da raiz é o único modo de aumentar a altura de uma árvore B. A Figura 18.6 ilustra esse caso. Diferente de uma árvore de pesquisa binária, uma árvore B aumenta em altura na parte superior, em vez de aumentar na parte inferior. O procedimento termina chamando B-TREE-INSERT-NONFULL para executar a inserção da chave k na árvore com raiz no nó de raiz não-completo. B-TREE-INSERT-NONFULL executa a recursão descendo a árvore conforme necessário, e garante em todos os momentos que o nó ao qual ele recorre não está completo, através de uma chamada a B-TREE-SPLIT-CHILD à medida que se torna necessário.

O procedimento recursivo auxiliar B-TREE-INSERT-NONFULL insere a chave k no nó x, que se presume ser não cheio quando o procedimento é chamado. A operação de B-TREE-INSERT e a operação recursiva de B-TREE-INSERT-NONFULL garantem que essa hipótese é verdadeira.

```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if folha[x]
 3    then while i ≥ 1 e k < chave_i[x]
 4           do chave_{i+1}[x] ← chave_i[x]
 5              i ← i − 1
 6       chave_{i+1}[x] ← k
 7       n[x] ← n[x] + 1
 8       DISK-WRITE(x)
 9    else while i ≥ 1 e k < chave_i[x]
10           do i ← i − 1
11       i ← i + 1
12       DISK-READ(c_i[x])
13       if n[c_i[x]] = 2t − 1
14          then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15               if k > chave_i[x]
16                 then i ← i + 1
17       B-TREE-INSERT-NONFULL(c_i[x], k)
```

O procedimento B-TREE-INSERT-NONFULL funciona da maneira descrita a seguir. As linhas 3 a 8 tratam o caso no qual x é um nó de folha, inserindo a chave k em x. Se x não é um nó de folha, então devemos inserir k no nó de folha apropriado na subárvore com raiz no nó interno x.

Nesse caso, as linhas 9 a 11 determinam o filho de x para o qual a recursão é descendente. A linha 13 detecta se a recursão desceria até um filho completo, e nesse caso a linha 14 utiliza B-TREE-SPLIT-CHILD para dividir esse filho em dois filhos não-completos, e as linhas 15 e 16 determinam qual dos dois filhos é agora o filho correto para o qual se deve descer. (Observe que não há necessidade de uma operação DISK-READ($c_i[x]$) após a linha 16 incrementar i, pois a recursão descerá nesse caso até um filho que acaba de ser criado por B-TREE-SPLIT-CHILD.) Portanto, o efeito final das linhas 13 a 16 é garantir que o procedimento nunca recorrerá até um nó completo. Então, a linha 17 utiliza a recursão para inserir k na subárvore apropriada. A Figura 18.7 ilustra os diversos casos de inserção em uma árvore B.

O número de acessos ao disco executados por B-TREE-INSERT é $O(h)$ para uma árvore B de altura h, pois somente $O(1)$ operações DISK-READ e DISK-WRITE são executados entre chamadas a B-TREE-INSERT-NONFULL. O tempo total de CPU usado é $O(th) = O(t \log_t n)$. Tendo em vista que B-TREE-INSERT-NONFULL é recursiva de final, ele pode ser implementado de modo alternativo como um loop **while**, demonstrando que o número de páginas que precisam estar na memória principal em qualquer instante é $O(1)$.

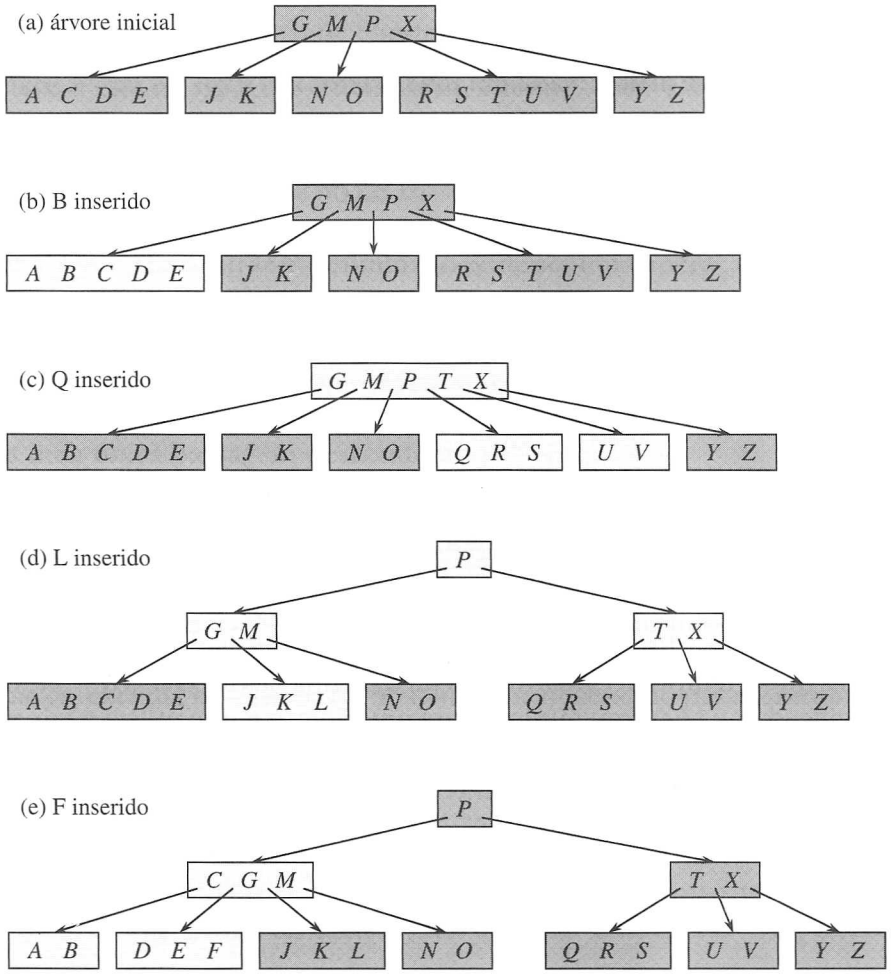

FIGURA 18.7 Inserção de chaves em uma árvore B. O grau mínimo t para essa árvore B é 3; assim, um nó pode conter no máximo 5 chaves. Nós que são modificados pelo processo de inserção estão levemente sombreados. (**a**) A árvore inicial para este exemplo. (**b**) O resultado da inserção de B na árvore inicial; essa é uma inserção simples em um nó de folha. (**c**) O resultado da inserção de Q na árvore anterior. O nó $R\,S\,T\,U\,V$ é dividido em dois nós contendo $R\,S$ e $U\,V$, a chave T é movida para cima até a raiz e Q é inserido na metade mais à esquerda das duas (o nó $R\,S$). (**d**) O resultado da inserção de L na árvore anterior. A raiz é dividida de qualquer modo, pois ela é completa, e a árvore B cresce uma unidade em altura. Então, L é inserida na folha que contém $J\,K$. (**e**) O resultado da inserção de F na árvore anterior. O nó $A\,B\,C\,D\,E$ é dividido antes de F ser inserido na metade mais à direita das duas (o nó $D\,E$)

Exercícios

18.2-1

Mostre os resultados da inserção das chaves

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

nessa ordem em uma árvore B vazia com grau mínimo 2. Desenhe apenas as configurações da árvore imediatamente antes de ter de dividir algum nó, e desenhe também a configuração final.

18.2-2

Explique sob quais circunstâncias, se houver, operações DISK-READ ou DISK-WRITE redundantes são executadas durante o curso da execução de uma chamada a B-TREE-INSERT. (Uma operação DISK-READ redundante é uma operação DISK-READ para uma página que já está na memória. Uma operação DISK-WRITE redundante grava em disco uma página de informações idêntica à que já está armazenada lá.)

18.2-3

Explique como encontrar a chave mínima armazenada em uma árvore B e como encontrar o predecessor de uma dada chave armazenada em uma árvore B.

18.2-4 ★

Suponha que as chaves $\{1, 2, ..., n\}$ sejam inseridas em uma árvore B vazia com grau mínimo 2. Quantos nós tem a árvore B final?

18.2-5

Tendo em vista que nós de folhas não exigem nenhum ponteiro para filhos, eles poderiam usar um valor de t diferente (maior) de nós internos para o mesmo tamanho de página de disco. Mostre como modificar os procedimentos de criação e inserção em uma árvore B para tratar essa variação.

18.2-6

Suponha que B-TREE-SEARCH seja implementado para usar a pesquisa binária em lugar da pesquisa linear dentro de cada nó. Mostre que isso torna o tempo de CPU necessário igual a $O(\lg n)$, independentemente do modo como t poderia ser escolhido como uma função de n.

18.2-7

Suponha que o hardware de disco nos permite escolher arbitrariamente o tamanho de uma página de disco, mas que o tempo necessário para ler a página de disco seja $a + bt$, onde a e b são constantes especificadas, e t é o grau mínimo para uma árvore B que utiliza páginas do tamanho selecionado. Descreva como escolher t para minimizar (aproximadamente) o tempo de pesquisa na árvore B. Sugira um valor ótimo de t para o caso no qual $a = 5$ milissegundos e $b = 10$ microssegundos.

18.3 Eliminação de uma chave de uma árvore B

A eliminação de uma árvore B é análoga à inserção, embora um pouco mais complicada, porque uma chave pode ser eliminada de qualquer nó – não apenas de uma folha – e a eliminação de um nó interno exige que os filhos do nó sejam reorganizados. Como na inserção, devemos nos resguardar contra a possibilidade da eliminação produzir uma árvore cuja estrutura viole as propriedades de árvores B. Da mesma maneira que tivemos de assegurar que um nó não ficasse grande demais devido à inserção, devemos assegurar que um nó não ficará pequeno demais durante a eliminação (a não ser pelo fato de a raiz poder ter menos que o número mínimo t-1 de chaves, embora ela não tenha permissão para ter mais que o número máximo $2t - 1$ de chaves). Da mesma maneira que um algoritmo de inserção simples poderia ter de retornar se um nó no caminho onde a chave fosse inserida estivesse completo, uma abordagem simples para a eliminação po-

deria ter de retornar se um nó (diferente da raiz) ao longo do caminho no qual a chave tivesse de ser eliminada apresentasse o número mínimo de chaves.

Suponha que o procedimento B-TREE-DELETE tenha de eliminar a chave k da subárvore com raiz em x. Esse procedimento está estruturado para garantir que, sempre que B-TREE-DELETE for chamado recursivamente em um nó x, o número de chaves em x seja pelo menos o grau mínimo t. Observe que essa condição exige uma chave além do mínimo exigido pelas condições usuais de árvores B, de forma que às vezes uma chave talvez tenha de ser movida para dentro de um nó filho, antes da recursão descer até esse filho. Essa condição reforçada nos permite eliminar uma chave da árvore em uma única passagem descendente sem a necessidade de "voltar" (com uma única exceção, que explicaremos em breve). A especificação a seguir para eliminação de uma árvore B deve ser interpretada com a compreensão de que, se acontecer de o nó raiz x se tornar um nó interno sem nenhuma chave (essa situação pode ocorrer nos casos 2c e 3b a seguir), então x será eliminada e o único filho de x, $c_1[x]$, se tornará a nova raiz da árvore, diminuindo a altura da árvore em uma unidade e preservando a propriedade que afirma que a raiz da árvore contém no mínimo uma chave (a menos que a árvore esteja vazia).

Descreveremos como a eliminação funciona, em vez de apresentarmos o pseudocódigo. A Figura 18.8 ilustra os diversos casos de eliminação de chaves de uma árvore B.

1. Se a chave k está no nó x e x é uma folha, elimine a chave k de x.
2. Se a chave k está no nó x e x é um nó interno, faça o seguinte.
   - ***a.*** Se o filho y que precede k no nó x tem pelo menos t chaves, então encontre o predecessor k' de k na subárvore com raiz em y. Elimine recursivamente k', e substitua k por k' em x. (Encontrar k' e eliminá-lo pode ser uma operação executada em uma única passagem descendente.)
   - ***b.*** Simetricamente, se o filho z que segue k no nó x tem pelo menos t chaves, então encontre o sucessor k' de k na subárvore com raiz em z. Elimine recursivamente k', e substitua k por k' em x. (Encontrar k' e eliminá-lo pode ser uma operação executada em uma única passagem descendente.)
   - ***c.*** Caso contrário, se tanto y quanto z têm apenas $t-1$ chaves, faça a intercalação de k e todos os itens z em y, de modo que x perca tanto k quanto o ponteiro para z, e y contenha agora $2t-1$ chaves. Em seguida, libere z e elimine recursivamente k de y.
3. Se a chave k não estiver presente no nó interno x, determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k, se k estiver absolutamente na árvore. Se $c_i[x]$ tiver somente $t-1$ chaves, execute o passo 3a ou 3b conforme necessário para garantir que desceremos até um nó contendo pelo menos t chaves. Em seguida, encerre efetuando uma recursão sobre o filho apropriado de x.
   - ***a.*** Se $c_i[x]$ tiver somente $t-1$ chaves, mas tiver um irmão com t chaves, forneça a $c_i[x]$ uma chave extra, movendo uma chave de x para baixo até $c_i[x]$, movendo uma chave do irmão esquerdo ou direito imediato de $c_i[x]$ para dentro de x, e movendo o ponteiro do filho apropriado do irmão para $c_i[x]$.
   - ***b.*** Se $c_i[x]$ e todos os irmãos de $c_i[x]$ têm $t-1$ chaves, faça a intercalação de $c_i[x]$ com um único irmão, o que envolve mover uma chave de x para baixo até o novo nó intercalado, a fim de se tornar a chave mediana para esse nó.

Tendo em vista que a maioria das chaves em uma árvore B se encontra nas folhas, podemos esperar que, na prática, as operações de eliminação sejam usadas com maior freqüência para eliminar chaves de folhas. O procedimento B-TREE-DELETE atua então em uma passagem descendente pela árvore, sem ter de voltar a subir. Contudo, quando elimina uma chave em um nó interno, o procedimento efetua uma passagem descendente através da árvore, mas pode ter de retornar ao nó do qual a chave foi eliminada, a fim de substituir a chave por seu predecessor ou sucessor (casos 2a e 2b).

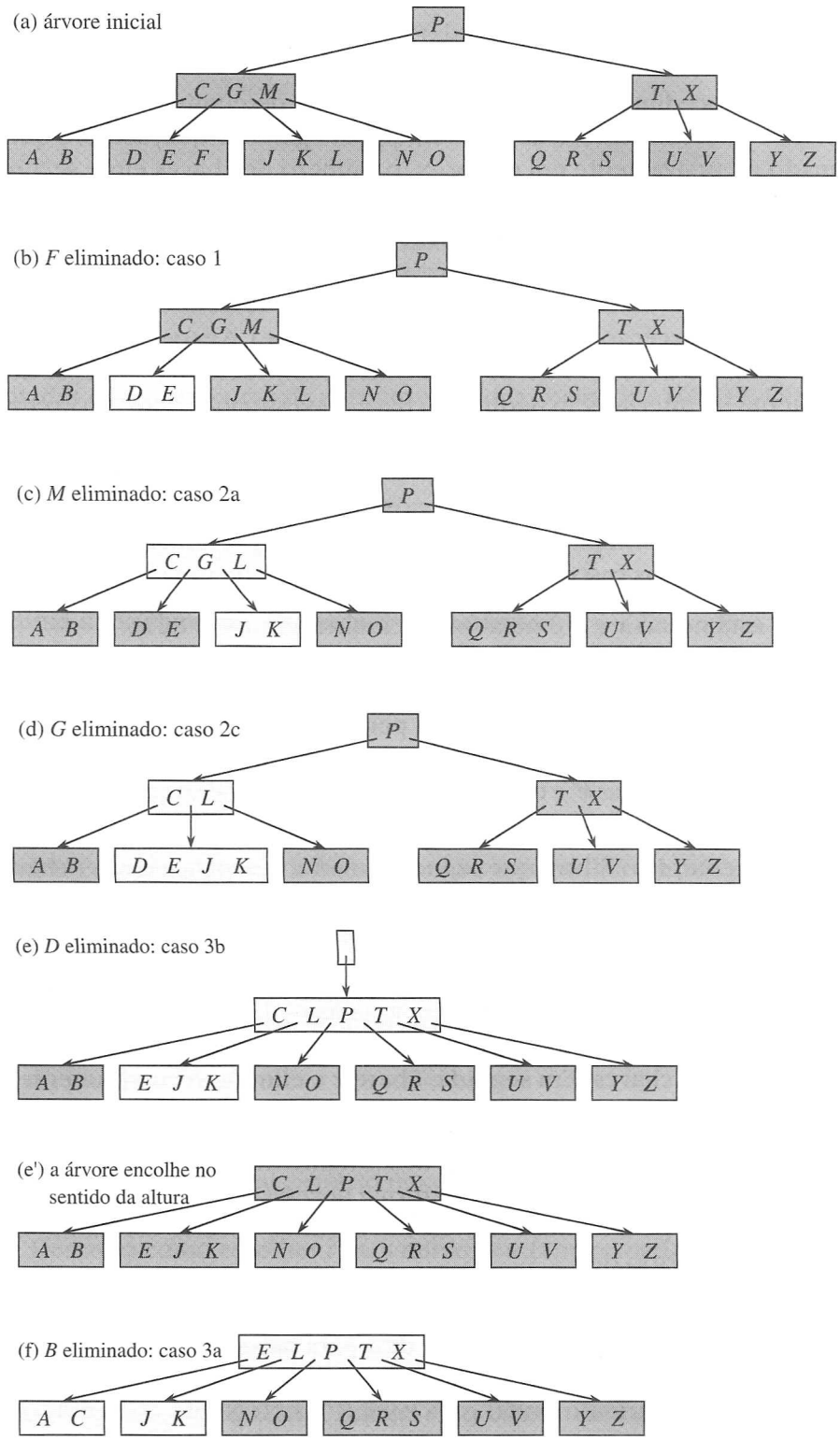

FIGURA 18.8 Eliminação de chaves de uma árvore B. O grau mínimo para essa árvore B é $t = 3$; assim, um nó (diferente da raiz) não pode ter menos de 2 chaves. Nós que são modificados estão levemente sombreados. (**a**) A árvore B da Figura 18.7(e). (**b**) Eliminação de *F*. Esse é o caso 1: a simples eliminação de uma folha. (**c**) Eliminação de *M*. Esse é o caso 2a: o predecessor *L* de *M* é movido para cima, a fim de ocupar a posição de *M*. (**d**) Eliminação de *G*. Esse é caso 2c: *G* é empurrado para baixo até formar o nó *DEGJK*, e então *G* é eliminado dessa folha (caso 1). (**e**) Eliminação de *D*. Esse é o caso 3b: a recursão não pode descer até o nó *CL*, porque ele tem apenas 2 chaves; assim, *P* é empurrado para baixo e intercalado com *CL* e *TX* para formar *CLPTX*; em seguida, *D* é eliminado de uma folha (caso 1). (**e'**) Após (d), a raiz é eliminada e a árvore encolhe uma unidade em altura. (**f**) Eliminação de *B*. Esse é o caso 3a: *C* é movido para preencher a posição de *B*, e *E* é movido para preencher a posição de *C*

Embora esse procedimento pareça complicado, ele envolve apenas $O(h)$ operações de disco para uma árvore B de altura h, pois somente $O(1)$ chamadas a DISK-READ e DISK-WRITE são efetuadas entre invocações recursivas do procedimento. O tempo de CPU necessário é $O(th) = O(t \log_t n)$.

Exercícios

18.3-1
Mostre os resultados da eliminação de C, P e V, nessa ordem, da árvore da Figura 18.8(f).

18.3-2
Escreva o pseudocódigo para B-TREE-DELETE.

Problemas

18-1 Pilhas no espaço de armazenamento secundário
Considere a implementação de uma pilha em um computador que tem uma quantidade relativamente pequena de memória primária rápida e uma quantidade relativamente grande de espaço de armazenamento em disco mais lento. As operações PUSH e POP são admitidas sobre valores de uma única palavra. A pilha a que desejamos dar suporte pode crescer até se tornar muito maior do que seria possível caber na memória, e assim a maior parte dela terá de ser armazenada em disco.

Uma implementação de pilha simples, embora ineficiente, mantém a pilha inteira no disco. Mantemos na memória um ponteiro de pilha, o qual é o endereço de disco do elemento do topo da pilha. Se o ponteiro tiver o valor p, o elemento superior será a (p mod m)-ésima palavra na página $\lfloor p / m \rfloor$ do disco, onde m é o número de palavras por página.

Para implementar a operação PUSH, incrementamos o ponteiro da pilha, lemos a página apropriada no disco e a inserimos na memória, copiamos o elemento a ser empilhado para a palavra apropriada na página e gravamos a página de novo no disco. Uma operação POP é semelhante. Decrementamos o ponteiro da pilha, lemos a página apropriada no disco e retornamos à posição superior da pilha. Não precisamos gravar de novo a página, pois ela não foi modificada.

Pelo fato de operações de disco serem relativamente dispendiosas, usamos o número total de acessos ao disco como uma medida de mérito para qualquer implementação. Também levamos em conta o tempo da CPU, mas debitamos $\Theta(m)$ por qualquer acesso de disco a uma página de m palavras.

a. Assintoticamente, qual é o número de acessos ao disco no pior caso para n operações de pilhas com o uso dessa implementação simples? Qual é o tempo de CPU para n operações de pilhas? (Expresse sua resposta em termos de m e n para esta parte e para as partes subseqüentes.)

Agora, considere uma implementação de pilha na qual mantemos na memória uma única página da pilha. (Também mantemos uma pequena quantidade de memória para controlar qual página está atualmente na memória.) Podemos executar uma operação de pilha somente se a página de disco relevante reside na memória. Se necessário, a página atualmente na memória pode ser gravada no disco e a nova página lida do disco para a memória. Se a página de disco relevante já estiver na memória, então não será necessário nenhum acesso ao disco.

b. Qual é o número de acessos ao disco no pior caso exigido para n operações PUSH? Qual é o tempo de CPU?

c. Qual é o número de acessos ao disco no pior caso exigido para n operações de pilhas? Qual é o tempo de CPU?

Agora, suponha que a pilha seja implementa mantendo-se duas páginas na memória (além de um número pequeno de palavras para contabilidade).

d. Descreva como gerenciar as páginas da pilha de modo que o número amortizado de acessos ao disco para qualquer operação de pilha seja $O(1/m)$ e o tempo de CPU amortizado para qualquer operação de pilha seja $O(1)$.

18-2 Junção e divisão de árvores 2-3-4

A operação de ***junção*** toma dois conjuntos dinâmicos S' e S'' e um elemento x tal que, para qualquer $x' \in S'$ e $x'' \in S''$, temos $chave[x'] < chave[x] < chave[x'']$. Ela retorna um conjunto $S = S' \cup \{x\} \cup S''$. A operação de ***divisão*** é como uma junção "inversa": dado um conjunto dinâmico S e um elemento $x \in S$, ela cria um conjunto S' que consiste em todos os elementos de $S - \{x\}$ cujas chaves são menores que $chave[x]$, e um conjunto S'' que consiste em todos os elementos em $S - \{x\}$ cujas chaves são maiores que $chave[x]$. Neste problema, investigaremos como implementar essas operações sobre árvores 2-3-4. Supomos por conveniência que os elementos consistem apenas em chaves e que todos os valores de chaves são distintos.

a. Mostre como manter, para todo nó x de uma árvore 2-3-4, a altura da subárvore com raiz em x como um campo $altura[x]$. Certifique-se de que sua implementação não afeta os tempos de execução assintóticos de pesquisa, inserção e eliminação.

b. Mostre como implementar a operação de junção. Dadas duas árvores 2-3-4 T' e T'' e uma chave k, a junção deve ser executada no tempo $O(1 + |h' - h''|)$, onde h' e h'' são as alturas de T' e T'', respectivamente.

c. Considere o caminho p desde a raiz de uma árvore 2-3-4 T até uma dada chave k, o conjunto S' de chaves em T que são menores que k, e o conjunto S'' de chaves em T que são maiores que k. Mostre que p divide S' em um conjunto de árvores $\{T'_0, T'_1, \ldots, T'_m\}$ e um conjunto de chaves $\{k'_1, k'_2, \ldots, k'_m\}$, onde para $i = 1, 2, \ldots, m$, temos $y < k'_i < z$ para quaisquer chaves $y \in T'_{i-1}$ e $z \in T'_i$. Qual é o relacionamento entre as alturas de T'_{i-1} e T'_i? Descreva o modo como p divide S'' em conjuntos de árvores e chaves.

d. Mostre como implementar a operação de divisão sobre T. Utilize a operação de junção para montar as chaves de S' em uma única árvore 2-3-4 T' e as chaves de S'' em uma única árvore 2-3-4 T''. O tempo de execução da operação de divisão deve ser $O(\lg n)$, onde n é o número de chaves em T. (*Sugestão*: Os custos para as operações de junção devem se encaixar.)

Notas do capítulo

Knuth [185], Aho, Hopcroft e Ullman [5] e ainda Sedgewick [269] apresentam discussões adicionais de esquemas de árvores balanceadas e árvores B. Comer [66] fornece uma pesquisa ampla de árvores B. Guibas e Sedgewick [135] discutem os relacionamentos entre vários tipos de esquemas de árvores balanceadas, inclusive árvores vermelho-preto e árvores 2-3-4.

Em 1970, J. E. Hopcroft criou as árvores 2-3, precursoras das árvores B e das árvores 2-3-4, nas quais todo nó interno tem dois ou três filhos. As árvores B foram introduzidas por Bayer e McCreight em 1972 [32]; eles não explicaram a escolha desse nome.

Bender, Demaine e Farach-Colton [37] estudaram como fazer árvores B funcionarem bem na presença de efeitos de hierarquia de memória. Seus algoritmos ***sem memória do cache*** funcionam de forma eficiente sem conhecer explicitamente os tamanhos de transferência de dados dentro da hierarquia de memória.