18 ÁRVORES B

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

As B-árvores são diferentes das árvores vermelho-preto no sentido de que os nós de B-árvores podem ter muitos filhos, de alguns até milhares. Isto é, o "fator de ramificação" de uma B-árvore pode ser bastante grande, embora normalmente, dependa de características da unidade de disco utilizada. As B-árvores são semelhantes às árvores vermelho-preto no sentido de que toda B-árvore de n nós tem altura $O(\lg n)$. Todavia, a altura exata de uma B-árvore pode ser consideravelmente menor que a altura de uma árvore vermelho-preto porque seu fator de ramificação e, por consequência, a base do logaritmo que expressa sua altura pode ser muito maior. Portanto, também podemos usar B-árvores para implementar muitas operações de conjuntos dinâmicos no tempo $O(\lg n)$.

B-árvores generalizam árvores de busca binária de modo natural. A Figura 18.1 mostra uma B-árvore simples. Se um nó interno x de uma B-árvore contém $x.n$ chaves, então x tem $x.n + 1$ filhos. As chaves no nó x servem como pontos de divisão que separam a faixa de chaves manipulada por x em $x.n + 1$ subfaixas, cada uma tratada por um filho de x. Quando procuramos uma chave em uma B-árvore, tomamos uma decisão de $(x.n + 1)$ vias, com base em comparações com as $x.n$ chaves armazenadas no nó x. A estrutura de nós de folhas é diferente da estrutura de nós internos; examinaremos essas diferenças na Seção 18.1.

A Seção 18.1 dá uma definição precisa de B-árvores e prova que o aumento da altura de uma B-árvore é apenas logarítmico de acordo com o número de nós que ela contém. A Seção 18.2 descreve como procurar uma chave e como inserir uma chave em uma B-árvore, e a Seção 18.3 discute eliminação. Porém, antes de prosseguir, precisamos perguntar por que avaliamos estruturas de dados projetadas para funcionar em um disco de modo diferente das estruturas de dados projetadas para funcionar em memória principal de acesso aleatório.

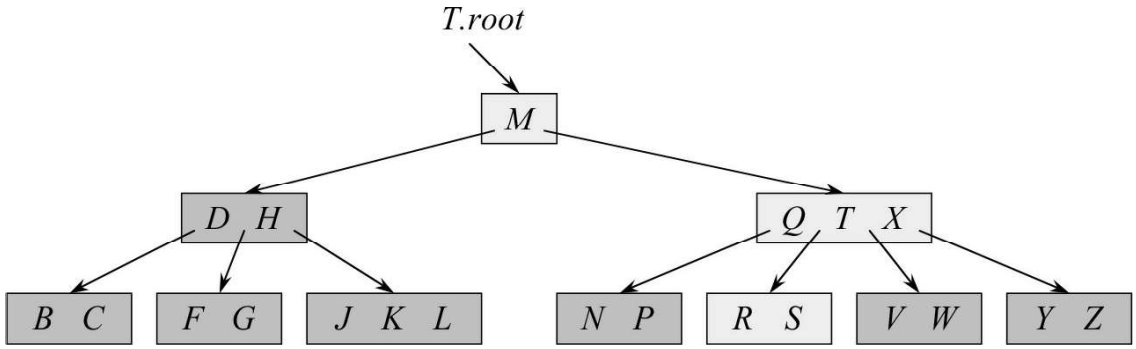


Figura 18.1 Uma B-árvore cujas chaves são as consoantes do alfabeto latino. Um nó interno x contendo $x.n$ chaves tem $x.n + 1$ filhos. Todas as folhas estão na mesma profundidade na árvore. Os nós sombreados em tom mais claro são examinados em uma busca pela letra R.

Estruturas de dados em armazenamento secundário

Os sistemas de computador aproveitam várias tecnologias que fornecem capacidade de memória. A ***memória primária*** (ou ***memória principal***) de um sistema de computador normalmente, consiste em chips de memória de silício. Em geral, essa tecnologia é mais de uma ordem de grandeza mais cara por bit armazenado que a tecnologia de armazenamento magnético, como fitas ou discos. A maioria dos sistemas de computador também tem ***armazenamento secundário*** baseado em discos magnéticos; muitas vezes, a quantidade de tal armazenamento secundário é no mínimo duas ordens de grandeza maior que a quantidade de memória primária. A Figura 18.2 mostra uma unidade de disco típica. A unidade consiste em uma ou várias ***lâminas*** que giram a uma velocidade constante em torno de um ***eixo*** comum. Um material magnetizável cobre a superfície de cada lâmina. A unidade de disco lê ou grava cada lâmina por meio de uma ***cabeça*** na extremidade de um ***braço***. Os braços podem movimentar suas cabeças aproximando-se ou afastando-se do fuso. Quando determinada cabeça está estacionária, a superfície que passa sob ela é denominada ***trilha***. Várias lâminas aumentam somente a capacidade da unidade de disco, mas não seu desempenho.

Embora os discos sejam mais baratos e tenham maior capacidade que a memória principal, eles são muito, muito mais lentos porque têm peças mecânicas móveis.[1] O movimento mecânico tem dois componentes: a rotação da lâmina e o movimento do braço. Na época da redação desta edição, a velocidade de rotação dos discos comerciais era 5.400-15.000 revoluções por minuto (RPM). O que existia, normalmente no comércio eram velocidades de 15.000 RPM em unidades de disco de grau de servidor, 7.200 RPM em unidades de disco de computadores de mesa e 5.400 RPM em unidades de disco de laptops. Embora a velocidade de 7.200 RPM pareça alta, uma rotação demora 8,33 milissegundos, tempo que é mais de cinco ordens de grandeza maior que os tempos de acesso de 50 nanossegundos (mais ou menos) comumente encontrados em memória de silício. Em outras palavras, se temos de esperar uma rotação completa para um item específico cair sob a cabeça de leitura/gravação, podemos acessar a memória principal mais de 100.000 vezes durante esse mesmo período. Em média, temos de esperar somente metade de uma rotação, mas ainda assim, a diferença entre tempos de acesso para memórias de silício e para discos é enorme. A movimentação dos braços também demora algum tempo. Na época em que redigimos esta edição, os tempos de acesso médios para discos comerciais estavam na faixa de 8-11 milissegundos.

Para amortizar o tempo gasto na espera de movimentos mecânicos, os discos acessam vários itens de cada vez, e não apenas um. As informações são divididas em várias ***páginas*** de bits de igual tamanho que aparecem consecutivamente dentro de trilhas, e cada leitura ou gravação de disco é de uma ou mais páginas inteiras. Para um disco típico, uma página pode ter 2^{11} a 2^{14} bytes de comprimento. Tão logo a cabeça de leitura/gravação esteja posicionada corretamente e o disco tenha girado até o início da página desejada, a leitura ou gravação em disco magnético é inteiramente eletrônica (exceto a rotação do disco) e o disco pode ler ou gravar rapidamente grande quantidade de dados.

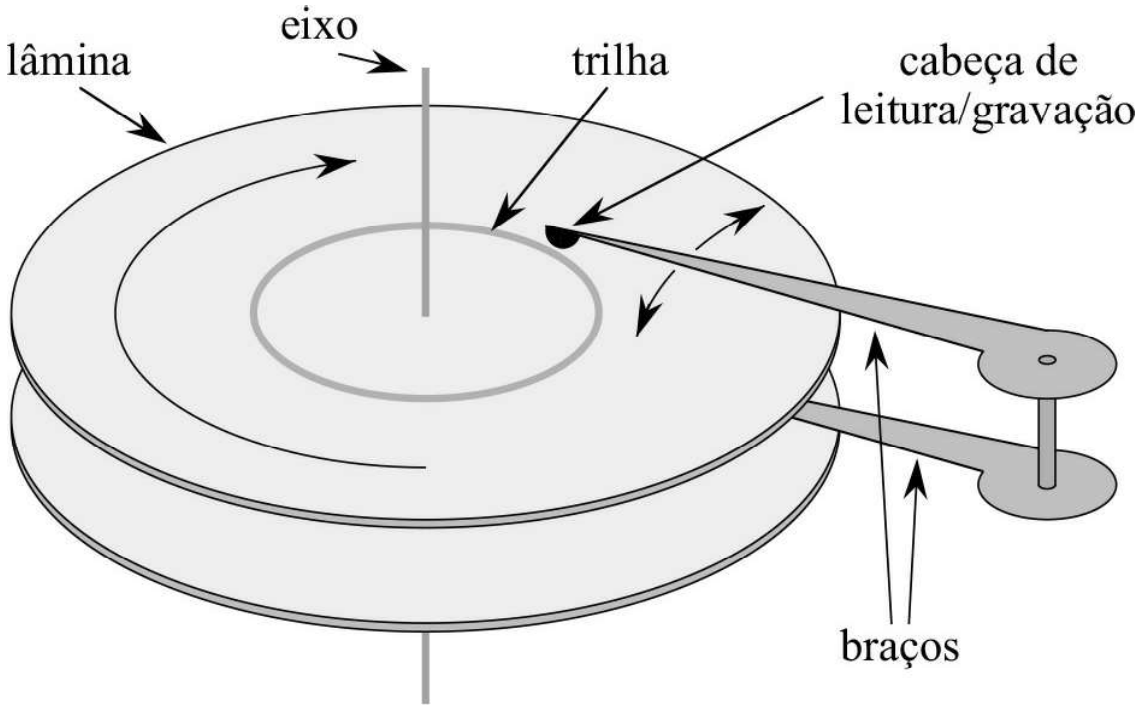


Figura 18.2 Uma unidade de disco típica. Ela é composta por uma ou mais lâminas (aqui mostramos duas lâminas) que giram em torno de um fuso. Cada lâmina é lida e gravada com uma cabeça na extremidade de um braço. Os braços giram ao redor de um eixo pivô comum. Uma trilha é a superfície que passa sob a cabeça de leitura/gravação quando a cabeça está estacionária.

Muitas vezes, acessar uma página de informações em um disco e ler essa página demora mais que examinar todas as informações lidas. Por essa razão, neste capítulo estudaremos separadamente os dois componentes principais do tempo de execução:

- o número de acessos ao disco, e
- o tempo de CPU (ou de computação).

Medimos o número de acessos ao disco em termos do número de páginas de informações que precisam ser lidas do disco ou nele gravadas. Observamos que o tempo de acesso ao disco não é constante — depende da distância entre a trilha atual e a trilha desejada, e também da posição de rotação inicial do disco. Não obstante, usaremos o número de páginas lidas ou gravadas como uma aproximação de primeira ordem do tempo total gasto no acesso ao disco.

Em uma aplicação típica de B-árvore, a quantidade de dados manipulados é tão grande que os dados não cabem todos na memória principal de uma só vez. Os algoritmos de B-árvores copiam páginas selecionadas do disco para a memória principal conforme necessário e gravam novamente em disco as páginas que foram alteradas. Algoritmos de B-árvores mantêm somente um número constante de páginas na memória principal em qualquer instante; assim, o tamanho da memória principal não limita o tamanho das B-árvores 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, poderemos referenciar os atributos do objeto do modo usual: por exemplo, $x.chave$. Contudo, se o objeto referenciado por x residir no disco, teremos de executar a operação DISK-READ(x) para ler o objeto x para a memória principal antes de podermos referenciar seus atributos. (Supomos que, se x já está na memória principal, DISK-READ(x) não requer nenhum acesso ao disco; é uma "não operação" ("no-op").) De modo semelhante, a operação DISK-WRITE(x) é usada para gravar quaisquer alterações que tenham sido feitas nos atributos do objeto x. Isto é, o padrão típico de trabalho com um objeto é o seguinte:

```
x = um ponteiro para algum objeto
DISK-READ(x)
operações que acessam e/ou modificam os atributos de x
DISK-WRITE(x)                    // omitida se nenhum atributo de x foi alterado
outras operações que acessam mas não modificam atributos de x
```

O sistema pode manter somente um número limitado de páginas na memória principal em qualquer instante. Suporemos que, o sistema descarrega da memória principal as páginas que não estão mais em uso; nossos algoritmos de B-árvores ignorarão essa questão.

Visto que na maioria dos sistemas o tempo de execução de um algoritmo de B-árvore depende primariamente do número de operações DISK-READ e DISK-WRITE que executa, seria bom que cada uma dessas operações lesse ou gravasse o máximo possível de informações. Assim, um nó de B-árvore é, normalmente, tão grande quanto uma página de disco inteira, e esse tamanho limita o número de filhos que um nó de B-árvore pode ter.

Para uma B-árvore grande armazenada em disco, normalmente, os fatores de ramificação estão entre 50 e 2.000, dependendo do tamanho de uma chave em relação ao tamanho de uma página. Um fator de ramificação grande reduz drasticamente a altura da árvore e também o número de acessos ao disco necessários para encontrar qualquer chave. A Figura 18.3 mostra uma B-árvore com um fator de ramificação de 1.001 e altura 2 que pode armazenar mais de um bilhão de chaves; não obstante, visto que podemos manter o nó de raiz permanentemente na memória principal, podemos encontrar qualquer chave dessa árvore com, no máximo, dois acessos ao disco.

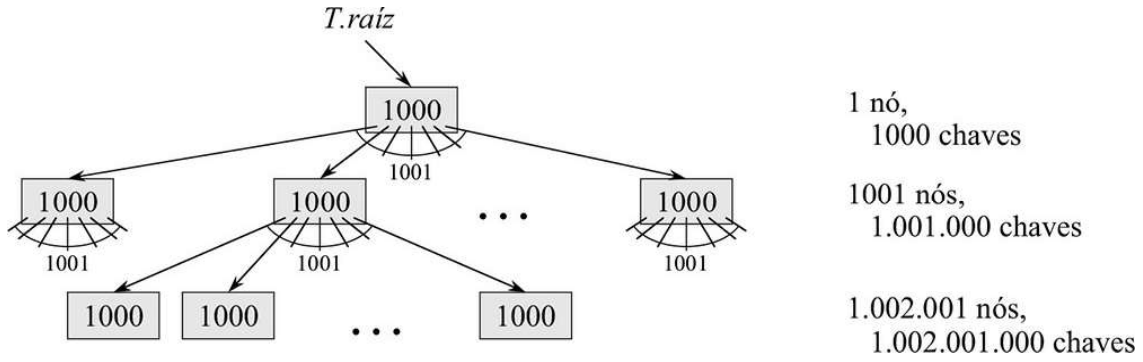


Figura 18.3 Uma B-árvore de altura 2 contendo mais de um bilhão de chaves. Dentro de cada nó x aparece $x.n$, o número de chaves em x. Cada nó interno e folha contêm 1.000 chaves. Essa B-árvore contém 1.001 nós na profundidade 1 e mais de um milhão de folhas na profundidade 2.

18.1 DEFINIÇÃO DE B-ÁRVORES

Por questão de simplicidade, supomos — como fizemos para as árvores de busca binária e árvores vermelho-preto que quaisquer "informações satélites" associadas a uma chave residem no mesmo nó que a chave. Na prática, poderíamos até mesmo armazenar com cada chave apenas um ponteiro para uma outra página de disco que contenha as informações satélites para essa chave. O pseudocódigo neste capítulo, supõe implicitamente que as informações satélites associadas a uma chave, ou o ponteiro para tais informações satélites, acompanham a chave sempre que esta passar de nó para nó. Uma variante comum de B-árvore, conhecida como ***B-árvore***+, armazena todas as informações satélites nas folhas e apenas chaves e ponteiros de filhos nos nós internos, o que maximiza o fator de ramificação dos nós internos.

Uma ***B-árvore***+ T é uma árvore enraizada (cuja raiz é $T.raiz$) que tem as seguintes propriedades:

1. Todo nó x tem os seguintes atributos:

a. $x.n$, o número de chaves atualmente armazenadas no nó x,
b. as próprias $x.n$ chaves, $x.chave_1$, $x.chave_2$, ... , $x.chave_{x.n}$, armazenadas em ordem não decrescente, de modo que $x.chave_1 \leq x.chave_2 \leq \ldots \leq x.chave_{x.n}$,
c. $x.folha$, um valor booleano que é VERDADEIRO se x é uma folha e FALSO se x é um nó interno.

2. Cada nó interno x também contém $x.n + 1$ ponteiros $x.c_1$, $x.c_2$, ..., $x.c_{x.n+1}$ para seus filhos. Os nós de folhas não têm filhos e, assim, seus atributos c_i são indefinidos.
3. As chaves $x.chave_i$ separam as faixas de chaves armazenadas em cada subárvore: se k_i é qualquer chave armazenada na subárvore com raiz $c_i x$, então

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

4. Todas as folhas têm a mesma profundidade, que é a altura h da árvore.
5. Os nós têm limites inferiores e superiores para o número de chaves que podem conter. Expressamos esses limites em termos de um inteiro fixo $t \geq 2$ denominado ***grau mínimo*** da B-árvore:
   a. Todo nó, exceto a raiz, deve ter no mínimo $t - 1$ chaves. Assim, todo nó interno, exceto a raiz, tem no mínimo t filhos. Se a árvore é não vazia, a raiz deve ter no mínimo uma chave.
   b. Todo nó pode conter no máximo $2t - 1$ chaves. Portanto, um nó interno pode ter no máximo $2t$ filhos. Dizemos que um nó está ***cheio*** se contém exatamente $2t - 1$ chaves.[2]

A B-árvore mais simples ocorre quando $t = 2$. Então, todo nó interno tem dois, três ou quatro filhos, e temos uma ***árvore 2-3-4***. Todavia, na prática, valores muito mais altos de t produzem B-árvores de menor altura.

A altura de uma B-árvore

O número de acessos ao disco exigidos para a maioria das operações em uma B-árvore é proporcional à altura da B-árvore. Analisamos agora a altura do pior caso de uma B-árvore.

Teorema 18.1

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

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

Prova A raiz de uma B-árvore T contém no mínimo uma chave, e todos os outros nós contêm no mínimo $t - 1$ chaves. Assim, T, cuja altura é h, tem no mínimo dois nós na profundidade 1, no mínimo $2t$ nós na profundidade 2 e no mínimo $2t_2$ nós na profundidade 3 ,e assim por diante, até que na profundidade h ela tem no mínimo $2t_h$-1 nós. A Figura 18.4 ilustra tal árvore para $h = 3$. Assim, o número n de chaves satisfaz a 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 $t_h \leq (n + 1)/2$. Tomando logaritmos em base t de ambos os lados prova-se o teorema.

Vemos aqui, o poder de B-árvores em comparação com 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 B-árvores a base do logaritmo pode ser muitas vezes maior. Assim, B-árvores poupam um fator de aproximadamente $\lg t$ em relação a árvores vermelho-preto no que se refere ao número de nós examinados para a maioria das operações de árvore. Como, normalmente, temos de acessar o disco para examinar um nó arbitrário em uma árvore, as B-árvores evitam uma quantidade substancial de acessos ao disco.

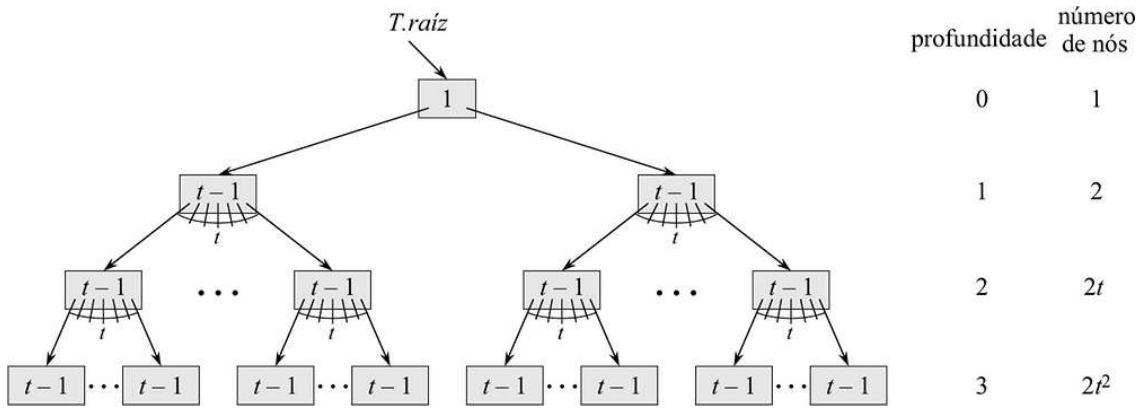


Figura 18.4 Uma B-árvore de altura 3 contendo um número mínimo possível de chaves. Mostramos $x.n$ dentro de cada nó x.

Exercícios

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

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

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

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

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 vermelhos a seus próprios filhos pretos.

18.2 Operações básicas em B-árvores

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 B-árvore está sempre na memória principal, de modo que nunca precisamos executar uma operação Disk-Read na raiz; porém, temos de executar uma operação Disk-Write da raiz, sempre que o nó de raiz for modificado.
- Qualquer nó só poderá ser passado como parâmetro após a execução de uma operação Disk-Read nesse mesmo nó.

Os procedimentos que apresentamos são algoritmos de "uma passagem" cuja execução ocorre na direção descendente em relação à raiz da árvore, sem ter de retornar.

Busca em uma B-árvore

Executar uma busca em uma B-árvore é muito semelhante a executar uma busca em uma árvore de busca binária, exceto 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 exatamente, em cada nó interno x, tomamos uma decisão de ramificação de $(x.n + 1)$ vias.

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

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

Usando um procedimento de busca linear, as linhas 1-3 encontram o menor índice i tal que $k \leq x.chave_i$ ou definem i como $x.n + 1$. As linhas 4-5 verificam se agora descobrimos a chave e a retornam se a tivermos descoberto. Caso contrário, as linhas 6-9 terminam a busca sem sucesso (se x é uma folha) ou executam uma recursão para procurar a subárvore adequada de x, após executar a necessária operação DISK-READ naquele filho.

A Figura 18.1 ilustra a operação de B-TREE-SEARCH. O procedimento examina os nós sombreados em tom mais claro durante uma operação de busca da chave R.

Como ocorre no procedimento TREE-SEARCH para árvores de busca binária, os nós encontrados durante a recursão formam um caminho simples descendente desde a raiz da árvore. Portanto, o procedimento B-TREE-SEARCH acessa $O(h) = O(\log_t n)$ páginas de disco, onde h é a altura da B-árvore e n é o número de chaves na B-árvore. Visto que $x.n < 2t$, o laço **while** das linhas 2-3 demora o tempo $O(t)$ dentro de cada nó, e o tempo total de CPU é $O(th) = O(t \log_t n)$.

Criando uma B-árvore vazia

Para construir uma B-árvore T, primeiro utilizamos B-TREE-CREATE para criar um nó de raiz vazio e depois chamamos B-TREE-INSERT para acrescentar 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 considerar que um nó criado por ALLOCATE-NODE não requer nenhuma operação DISK-READ, já que ainda não existe nenhuma informação útil armazenada no disco para esse nó.

```
B-TREE-CREATE(T)
    x = ALLOCATE-NODE( )
    x.folha = TRUE
    x.n = 0
    DISK-WRITE(x)
    T.raiz = x
```

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

Inserindo uma chave em uma B-árvore

Inserir uma chave em uma B-árvore é significativamente mais complicado que inserir uma chave em uma árvore de busca binária. Quando se trata de árvores de busca binária, procuramos a posição de folha na qual inserir a nova chave. Porém, quando se trata de uma B-árvore, não podemos simplesmente criar um novo nó de folha e inseri-lo, já que a árvore resultante deixaria de ser uma B-árvore válida. Em vez disso, inserimos a nova chave em um nó de folha existente. Visto que não podemos inserir uma chave em um nó de folha que está cheio, recorremos a uma operação que ***reparte*** um nó cheio y (que tem $2t$ - 1 chaves) em torno de sua ***chave mediana*** $chave_t[y]$ em dois nós que têm somente t - 1 chaves cada. A chave mediana sobe para dentro do pai de y para identificar o ponto de repartição entre as duas novas árvores. Porém, se o pai de y também está cheio, temos de reparti-lo antes de podermos inserir a nova chave e, assim, podemos acabar repartindo nós cheios por toda a árvore acima.

Como ocorre com uma árvore de busca binária, podemos inserir uma chave em uma B-árvore em uma única passagem descendente pela árvore da raiz até uma folha. Para tal, não esperamos para verificar se realmente precisamos repartir um nó cheio para executar a inserção. Em vez disso, à medida que descemos a árvore à procura da posição à qual pertence a nova chave, repartimos cada nó cheio que encontramos pelo caminho (inclusive a própria folha). Assim, sempre que queremos repartir um nó cheio y, temos a certeza de que seu pai não está cheio.

Repartindo um nó em uma B-árvore

O procedimento B-TREE-SPLIT-CHILD toma como entrada um nó interno x *não cheio* (que consideramos estar na memória principal) e um índice i tal que $x.c_i$ (que também consideramos estar na memória principal) que é um filho *cheio* de x. Então, o procedimento reparte esse filho em dois e ajusta x de modo que ele tenha um filho adicional. Para repartir uma raiz cheia, primeiro transformaremos a raiz em um filho de um novo nó de raiz vazio para podermos usar B-TREE-SPLIT-CHILD. Assim, a altura da árvore aumenta de uma unidade; repartir é o único meio de a árvore crescer.

A Figura 18.5 ilustra esse processo. Repartimos o nó cheio $y = x.c_i$ em torno de sua chave mediana S, que sobe para o nó x, pai de y. As chaves em y que são maiores que a chave mediana passam para um novo nó z, que se torna um novo filho de x.

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

B-TREE-SPLIT-CHILD funciona pelo método direto de "recortar e colar". Aqui, x é o nó que está sendo repartido e y é o i-ésimo filho de x (definido na linha 2). 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 toma 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 sobe e torna-se a chave em x que separa y e z.

As linhas 1-9 criam o nó z e dão a ele as t - 1 chaves maiores e os t filhos correspondentes de y. A linha 10 ajusta a contagem de chaves para y. Finalmente, as linhas 11-17 inserem z como um filho de x, passam a chave mediana de y para cima até x, para separar y de z, e ajustam a contagem de chaves de x. As linhas 18-20 gravam todas as páginas de disco modificadas. O tempo de CPU usado por B-TREE-SPLIT-CHILD é (t), devido aos laços nas linhas 5-6 e 8-9. (Os outros laços são executados para $O(t)$ iterações.) O procedimento executa $O(1)$ operações de disco.

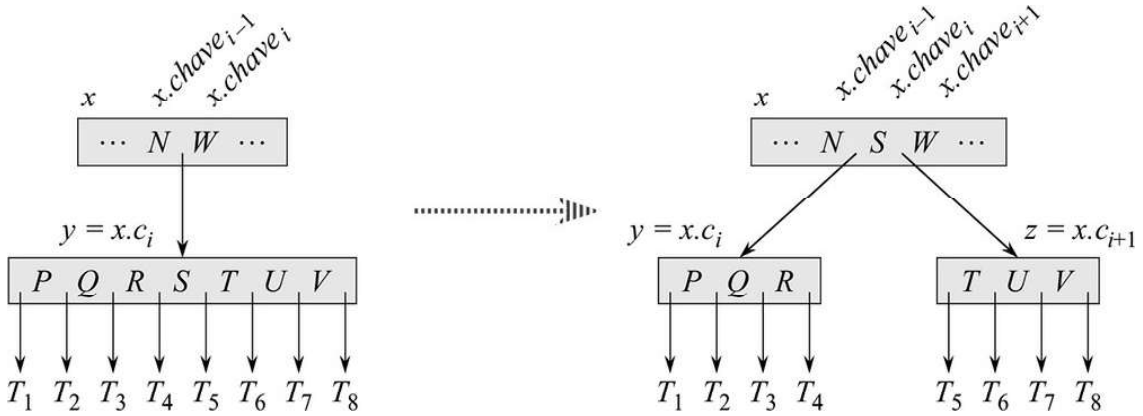


Figura 18.5 Repartição de um nó com $t = 4$. O nó $y = x.c_i$ é repartido em dois nós, y e z, e a chave mediana S de y sobe para dentro do pai de y.

Inserindo uma chave em uma B-árvore em uma única passagem descendente pela árvore

Inserir uma chave k em uma B-árvore T de altura h em uma única passagem descendente pela árvore requer $O(h)$ acessos ao disco. O tempo de CPU requerido é $O(th) = O(t \log^t n)$. O procedimento B-TREE-INSERT utiliza B-TREE-SPLIT-CHILD para garantir que a recursão nunca desça até um nó cheio.

```
B-TREE-INSERT(T,k)
1     r = T.raíz
2     if r.n == 2t − 1
3             s = ALLOCATE-NODE()
4             T.raíz = s
5             s.folha = FALSE
6             s.n = 0
7             s.c_1 = r
8             B-TREE-SPLIT-CHILD(s, 1)
9             B-TREE-INSERT-NONFULL(s, k)
10    else B-TREE-INSERT-NONFULL(r, k)
```

As linhas 3-9 tratam o caso no qual o nó de raiz r está cheio: a raiz é repartida e um novo nó s (que tem dois filhos) se torna a raiz. Repartir a raiz é o único modo de aumentar a altura de uma B-árvore. A Figura 18.6 ilustra esse caso. Diferentemente de uma árvore de busca binária, a altura de uma B-árvore aumenta em cima, em vez de embaixo. O procedimento termina chamando B-TREE-INSERT-NONFULL para inserir a chave k na árvore com raiz no nó de raiz não cheio. B-Tree-Insert-Nonfull executa recursão árvore abaixo conforme necessário e garante todas as vezes que o nó no qual executa a recursão não está cheio chamando B-TREE-SPLIT-CHILD quando necessário.

O procedimento recursivo auxiliar B-TREE-INSERT-NONFULL insere a chave k no nó x, suposto não cheio quando o procedimento é chamado. A operação de B-TREE-INSERT e a operação recursiva de B-TREE-INSERT-NONFULL garantem que tal suposição seja válida.

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

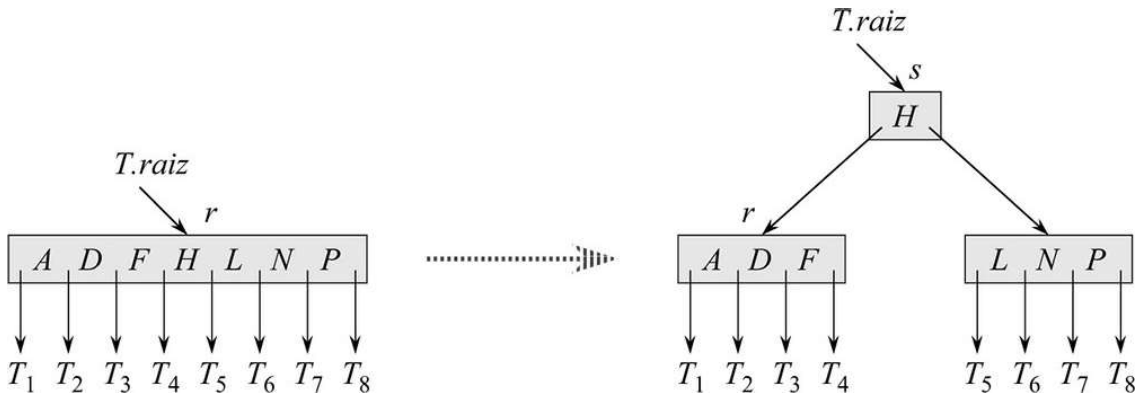


Figura 18.6 Repartição da raiz com $t = 4$. O nó de raiz r é repartido em dois, e um novo nó de raiz s é criado. A nova raiz contém a chave mediana de r e tem como filhos as duas metades de r. A altura da B-árvore aumenta de uma unidade quando a raiz é repartida.

O procedimento B-TREE-INSERT-NONFULL funciona da maneira descrita a seguir. As linhas 3-8 tratam o caso no qual x é um nó folha inserindo a chave k em x. Se x não é um nó folha, devemos inserir k no nó folha adequado na subárvore com raiz no nó interno x. Nesse caso, as linhas 9-11 determinam o filho de x para o qual a recursão é descendente. A linha 13 detecta se a recursão desceria até um filho cheio, caso em que a linha 14 usa B-TREE-SPLIT-CHILD para repartir esse filho em dois filhos não cheios, e as linhas 15-16 determinam qual dos dois filhos é agora o filho correto para o qual descer. (Observe que não há nenhuma necessidade de uma operação DISK-READ($x.c_i$) após a linha 16 incrementar i, já que nesse caso a recursão descerá até um filho que acabou de ser criado por B-TREE-SPLIT-CHILD.) Portanto, o efeito líquido das linhas 13-16 é garantir que o procedimento nunca executará uma recursão em um nó cheio. Então, a linha 17 executa recursão para inserir k na subárvore adequada. A Figura 18.7 ilustra os vários casos de inserção em uma B-árvore.

Para uma B-árvore de altura h, B-TREE-INSERT executa $O(h)$ acessos a disco, já que ocorrem somente $O(1)$ operações DISK-READ e DISK-WRITE entre chamadas a B-TREE-INSERT-NONFULL. O tempo total de CPU usado é $O(th) = O(t \log t\, n)$. Visto que B-TREE-INSERT-NONFULL é recursivo de cauda, podemos implementá-lo alternativamente como um laço

while, demonstrando assim que o número de páginas que precisam estar na memória principal em qualquer instante é $O(1)$.

Exercícios

18.2-1 Mostre os resultados da inserção das chaves

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

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

18.2-2 Explique sob quais circunstâncias, se houver, ocorrem operações redundantes DISK-READ ou DISK-WRITE 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 ali armazenada.)

18.2-3 Explique como encontrar a chave mínima armazenada em uma B-árvore e como encontrar o predecessor de uma determinada chave armazenada em uma B-árvore.

18.2-4 ★

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

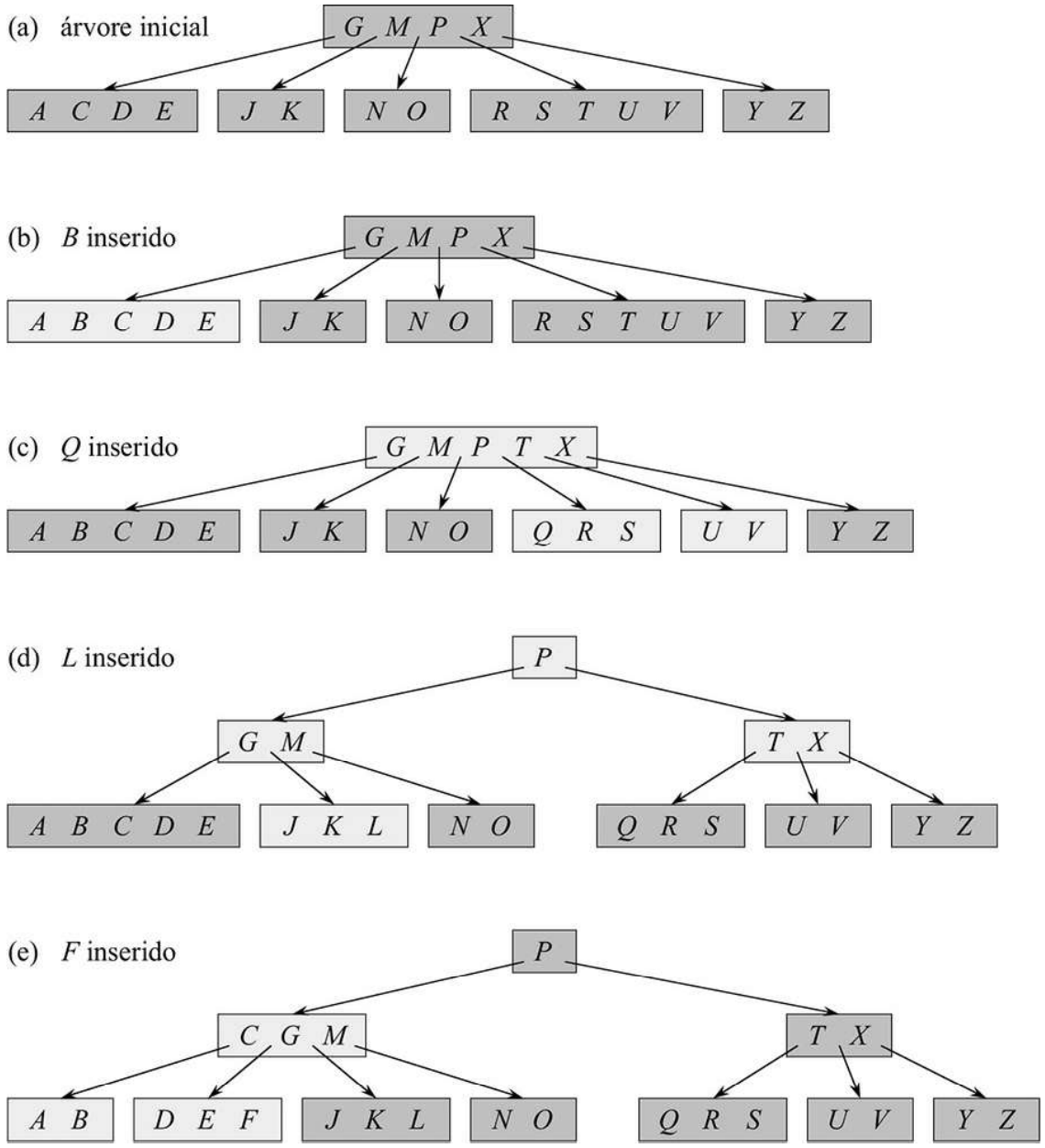


Figura 18.7 Inserção de chaves em uma B-árvore. O grau mínimo t para essa B-árvore é 3; assim, um nó pode conter no máximo cinco chaves. Nós que são modificados pelo processo de inserção estão sombreados em tom mais claro. **(a)** A árvore inicial para este exemplo. **(b)** O resultado da inserção de 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 repartido em dois nós contendo $R\ S$ e $U\ V$, a chave T é passada para cima até a raiz e Q é inserido na metade mais à esquerda das duas metades (o nó $R\ S$). **(d)** O resultado da inserção de L na árvore anterior. A raiz é repartida imediatamente, já que está cheia, e a altura da B-árvore aumenta de uma unidade. Então, L é inserido na folha que contém $J\ K$. **(e)** O resultado da inserção de F na árvore anterior. O nó $A\ B\ C\ D\ E$ é repartido antes de F ser inserido na metade mais à direita das duas metades (o nó $D\ E$).

18.2-5 Visto que nós folha não exigem nenhum ponteiro para filhos, eles poderiam usar um valor t diferente (maior) do número de nós internos para o mesmo tamanho de página de disco. Mostre como modificar os procedimentos de criação e inserção em uma B-árvore para tratar essa variação.

18.2-6 Suponha que quiséssemos implementar B-TREE-SEARCH para usar busca binária em vez de busca linear dentro de cada nó. Mostre que essa alteração resulta no tempo de CPU requerido $O(\lg n)$, independentemente do modo como t poderia ser escolhido em função de n.

18.2-7 Suponha que o hardware de disco nos permita 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 B-árvore que utiliza páginas do tamanho selecionado. Descreva como escolher t para minimizar (aproximadamente) o tempo de busca da B-árvore. Sugira um valor ótimo de t para o caso em que a = 5 milissegundos e b = 10 microssegundos.

18.3 Eliminar uma chave em uma B-árvore

Eliminar uma chave em uma B-árvore procedimento análogo à inserção, mas é um pouco mais complicada porque podemos eliminar uma chave de qualquer nó — não apenas uma folha — e, quando eliminamos uma chave de um nó interno, temos de rearranjar os filhos do nó. Como na inserção, devemos nos prevenir para que a eliminação não produza uma árvore cuja estrutura viole as propriedades de B-árvores. Exatamente como tivemos de assegurar que um nó não ficasse demasiadamente grande devido à inserção, temos de garantir que um nó não fique demasiadamente pequeno durante a eliminação (exceto a raiz, que poder ter menos que o número mínimo t - 1 de chaves). Assim como um algoritmo de inserção simples poderia ter de retroceder se um nó no caminho até o ponto de inserção da chave estivesse cheio, uma abordagem de eliminação simples poderia ter de retroceder se um nó (exceto a raiz) no caminho até o ponto de eliminação da chave tivesse o número mínimo de chaves.

O procedimento B-Tree-Delete elimina a chave k da subárvore com raiz em x. Projetamos esse procedimento para garantir que sempre que ele chamar a si mesmo recursivamente em um nó x o número de chaves em x é, pelo menos, o grau mínimo t. Observe que essa condição requer uma chave a mais que o mínimo exigido pelas condições usuais de B-árvores, de modo que, às vezes, será preciso passar uma chave para dentro de um nó filho, antes de a recursão descer até esse filho. Essa condição reforçada nos permite eliminar uma chave da árvore em uma única passagem descendente sem ter de "retroceder" (com uma única exceção, que explicaremos). Você deve interpretar a especificação dada a seguir para eliminação em uma B-árvore entendendo que, se alguma vez o nó de raiz x se tornar um nó interno sem nenhuma chave (essa situação pode ocorrer nos casos 2c e 3b, então eliminamos x, e o único filho de x, $x.c_1$, se torna a nova raiz da árvore, o que reduz a altura da árvore de uma unidade e preserva a propriedade de a raiz da árvore conter no mínimo uma chave (a menos que a árvore esteja vazia).

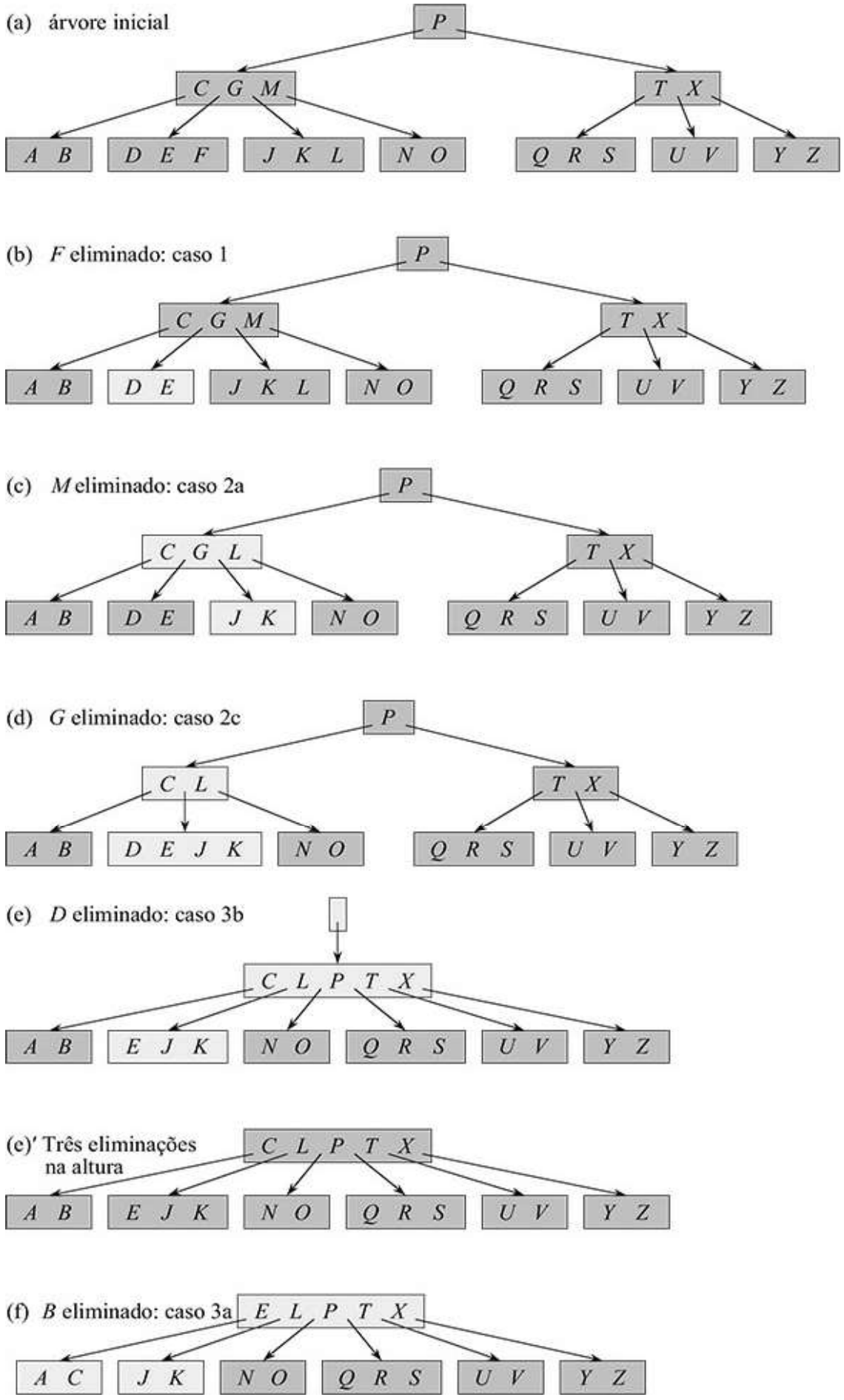
Descrevemos como a eliminação funciona, em vez de apresentarmos o pseudocódigo. A Figura 18.8 ilustra os vários casos de eliminação de chaves em uma B-árvore.

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 no mínimo 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. (Podemos encontrar k' e eliminá-lo em uma única passagem descendente.)
   - ***b.*** Se y tiver menos que t chaves, então, simetricamente, examine o filho z que segue k no nó x. Se z tiver no mínimo t chaves, encontre o sucessor k' de k na subárvore com raiz em z. Elimine k' recursivamente e substitua k por k' em x. (Podemos encontrar k' e eliminá-lo em uma única passagem descendente.)
   - ***c.*** Caso contrário, y e z têm apenas t - 1 chaves, junte k e todo o z com y, de modo que x perde k e também o ponteiro para z, e agora y contém $2t$ - 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_ix da subárvore adequada que deve conter k, se k estiver na árvore. Se $x.c_i$ tiver somente t - 1 chaves, execute a etapa 3a ou 3b conforme necessário para garantir que desceremos até um nó que contém no mínimo t chaves. Então, termine executando recursão no filho adequado de x.
   - ***a.*** Se $x.c_i$ tiver somente t - 1 chaves, mas tiver um irmão imediato com no mínimo t chaves, dê a $x.c_i$ uma chave extra passando uma chave de x para baixo até $x.c_i$, passando uma chave do irmão imediato à esquerda ou à

direita de $x.c_i$ para cima até x, e passando o ponteiro de filho adequado do irmão para $x.c_i$.

b. Se $x.c_i$ e os irmãos imediatos de $x.c_i$ têm t - 1 chaves, junte $x.c_i$ com um irmão, o que envolve passar uma chave de x para baixo até o novo nó resultante da junção, que assim se torna a chave mediana para esse nó.


(a) árvore inicial
P
C G M
T X
A B
D E F
J K L
N O
Q R S
U V
Y Z
(b) F eliminado: caso 1
P
C G M
T X
A B
D E
J K L
N O
Q R S
U V
Y Z
(c) M eliminado: caso 2a
P
C G L
T X
A B
D E
J K
N O
Q R S
U V
Y Z
(d) G eliminado: caso 2c
P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z
(e) D eliminado: caso 3b
C L P T X
A B
E J K
N O
Q R S
U V
Y Z
(e)′ Três eliminações na altura
C L P T X
A B
E J K
N O
Q R S
U V
Y Z
(f) B eliminado: caso 3a
E L P T X
A C
J K
N O
Q R S
U V
Y Z

18.8 Eliminação de chaves em uma B-árvore. O grau mínimo para essa B-árvore é $t = 3$; portanto, um nó (exceto a raiz) não pode ter menos de duas chaves. Nós que são modificados estão sombreados em tom mais claro. (**a**) A B-árvore da Figura 18.7(e). (**b**) Eliminação de *F*. Esse é o caso 1: eliminação simples de uma folha. (**c**) Eliminação de *M*. Esse é o caso 2a: o predecessor *L* de *M* passa para cima e ocupa a posição de *M*. (**d**) Eliminação de *G*. Esse é o caso 2c: empurramos *G* para baixo para formar o nó *D E G J K* e depois eliminamos *G* dessa folha (caso 1). (**e**) Eliminação de *D*. Esse é o caso 3b: a recursão não pode descer até o nó *C L* porque ele tem apenas duas chaves; assim, empurramos *P* para baixo e o intercalamos com *C L* e *T X* para formar *C L P T X*; então, eliminamos *D* de uma folha (caso 1). (**e'**) Após (e), eliminamos a raiz, e a altura da árvore encolhe uma unidade. (**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*.

Visto que a maioria das chaves em uma B-árvore se encontra nas folhas, podemos esperar que, na prática, operações de eliminação são usadas na maior parte das vezes para eliminar chaves de folhas. Então, o procedimento B-TREE-DELETE age em uma passagem descendente pela árvore, sem ter de retroceder. Contudo, quando elimina uma chave em um nó interno, o procedimento efetua uma passagem descendente pela árvore, mas pode ter de retornar ao nó do qual a chave foi eliminada para substituir a chave por seu predecessor ou sucessor (casos 2a e 2b).

Embora pareça complicado, esse procedimento envolve apenas $O(h)$ operações de disco para uma B-árvore de altura h, já que 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*, em ordem, da árvore da Figura 18.8(f).

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

Problemas

18-1 Pilhas em 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 armazenamento mais lento em disco. As operações PUSH e POP funcionam com valores de uma única palavra. A pilha que desejamos suportar pode crescer até seu tamanho tornar-se tão grande que não cabe mais na memória; por isso, parte dela tem de ser armazenada em disco.

Uma implementação de pilha simples, mas ineficiente, mantém a pilha inteira no disco. Mantemos na memória um ponteiro de pilha, que é o endereço de disco do elemento do topo da pilha. Se o ponteiro tiver o valor p, o elemento do topo é a (p mod m)-ésima palavra na página p/m do disco, onde m é o número de palavras por página. Para implementar a operação PUSH, incrementamos o ponteiro da pilha, lemos a página adequada para a memória de disco, copiamos o elemento a ser inserido na pilha para a palavra adequada na página e gravamos a página de novo no disco. Uma operação POP é semelhante. Decrementamos o ponteiro da pilha, lemos a página adequada no disco e voltamos ao topo da pilha. Não precisamos gravar de novo a página, já que ela não foi modificada.

Como operações de disco são relativamente caras, contamos dois custos para qualquer implementação: o número total de acessos ao disco e o tempo total de CPU. Qualquer acesso de disco a uma página de m palavras incorre em gastos de um acesso de disco e mais (m) tempo de CPU.

a. Assintoticamente, qual é o número de acessos ao disco do pior caso para n operações de pilhas usando essa 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 subsequentes.)

Agora, considere uma implementação de pilha na qual mantemos uma página da pilha na memória. (Mantemos também 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 residir na memória. Se necessário, podemos gravar a página atualmente na memória no disco e ler a nova página do disco para a memória. Se a página de disco relevante já estiver na memória, não será necessário nenhum acesso ao disco.

b. Qual é o número de acessos ao disco do pior caso exigido para n operações PUSH? Qual é o tempo de CPU?

c. Qual é o número de acessos ao disco do pior caso exigido para n operações de pilha? Qual é o tempo de CPU?

Agora, suponha que implementemos a pilha mantendo duas páginas na memória (além de um pequeno número 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 repartição 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 $x'.chave < x.chave < x''.chave$. A junção retorna um conjunto $S = S' \cup \{x\} \cup S''$. A operação de ***repartição*** 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 $x.chave$, e um conjunto S'' que consiste em todos os elementos em $S - \{x\}$ cujas chaves são maiores que $x.chave$. Neste problema, investigaremos como implementar essas operações em árvores 2-3-4. Consideramos 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 atributo $x.altura$. Certifique-se de que sua implementação não afeta os tempos de execução assintóticos de busca, 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 operação de 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 simples p da 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 reparte S' em um conjunto de árvores $\{T'_0, T'_1, ..., k'_m\}$ e um conjunto de chaves $\{k'_1, k'_2, ..., k'_m\}$ onde, para $i = 1, 2, ..., m$, temos $y < k'_i < z$ para quaisquer chaves $y \in T'_{i-1}$ e $z \in T'_i$. Qual é a relação entre as alturas de T'_{i-1} e T'_i? Descreva o modo como p reparte S'' em conjuntos de árvores e chaves.

d. Mostre como implementar a operação de repartição em 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 repartição deve ser $O(\lg n)$, onde n é o número de chaves em T. (*Sugestão*: Os custos para as operações de junção devem se cancelar.)

NOTAS DO CAPÍTULO

Knuth [211], Aho, Hopcroft e Ullman [5] e Sedgewick [306] apresentam discussões adicionais de esquemas de B-árvoresalanceadas e B-árvores. Comer [74] dá um levantamento abrangente de B-árvores. Guibas e Sedgewick [155] discutem as relações entre vários tipos de esquemas de B-árvoresalanceadas, inclusive árvores vermelho-preto e árvores 2-3-4.

Em 1970, J. E. Hopcroft criou as árvores 2-3, precursoras das B-árvores e das árvores 2-3-4, nas quais todo nó interno tem ou dois ou três filhos. Bayer e McCreight [35] apresentaram as B-árvores em 1972; eles não explicaram a escolha desse nome.

Bender, Demaine e Farach-Colton [40] estudaram como fazer B-árvores funcionarem bem na presença de efeitos de hierarquia de memória. Seus algoritmos ***sem consciência de cache*** funcionam eficientemente sem conhecer explicitamente os tamanhos de transferência de dados dentro da hierarquia de memória.

---

1 Na época da redação desta edição, unidades de disco de estado sólido tinham acabado de chegar ao mercado de consumo. Embora sejam mais rápidas que as unidades de disco mecânicas, custam mais por gigabyte e têm menor capacidade que as unidades de disco mecânicas.

2 Uma outra variante comum de uma B-árvore, conhecida como ***B-árvore****, exige que cada nó interno esteja no mínimo 2/3 cheio, em vez de no mínimo metade cheio, como exige uma B-árvore.