Capítulo 12

Árvores de pesquisa binária

As árvores de pesquisa são estruturas de dados que admitem muitas operações de conjuntos dinâmicos, inclusive SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT e DELETE. Desse modo, uma árvore de pesquisa pode ser usada como um dicionário e também como uma fila de prioridades.

As operações básicas sobre uma árvore de pesquisa binária demoram um tempo proporcional à altura da árvore. No caso de uma árvore binária completa com n nós, tais operações são executadas em um tempo $\Theta(\lg n)$ no pior caso. Porém, se a árvore é uma cadeia linear de n nós, as mesmas operações demoram um tempo $\Theta(n)$ no pior caso. Veremos na Seção 12.4 que a altura de uma árvore de pesquisa binária construída aleatoriamente é $O(\lg n)$, de forma que as operações básicas sobre conjuntos dinâmicos demoram o tempo $\Theta(\lg n)$ sobre tal árvore.

Na prática, nem sempre podemos garantir que as árvores de pesquisa binária são construídas aleatoriamente, mas existem variações cujo desempenho no pior caso em operações básicas tem uma boa garantia. O Capítulo 13 apresenta uma dessas variações, as árvores vermelho-preto, que têm altura $O(\lg n)$. O Capítulo 18 introduz as árvores B (B-trees), que são boas particularmente para manutenção de bancos de dados em um espaço de armazenamento secundário (disco) de acesso aleatório.

Depois de apresentar as propriedades básicas de árvores de pesquisa binária, as próximas seções mostram como percorrê-las para imprimir seus valores em seqüência ordenada, procurar por um valor, encontrar o elemento mínimo ou máximo, encontrar o predecessor ou o sucessor de um elemento e inserir ou eliminar elementos em uma árvore de pesquisa binária. As propriedades matemáticas básicas das árvores são apresentadas no Apêndice B.

12.1 O que é uma árvore de pesquisa binária?

Uma árvore de pesquisa binária é organizada, conforme seu nome sugere, em uma árvore binária, como mostra a Figura 12.1. Uma árvore desse tipo pode ser representada por uma estrutura de dados encadeada em que cada nó é um objeto. Além de um campo *chave* e de dados satélite, cada nó contém campos *esquerdo*, *direito* e p que apontam para os nós correspondentes a seu filho da esquerda, seu filho da direita e a seu pai, respectivamente. Se um filho ou o pai estiver ausente, o campo apropriado conterá o valor NIL. O nó raiz é o único nó na árvore cujo campo pai é NIL.

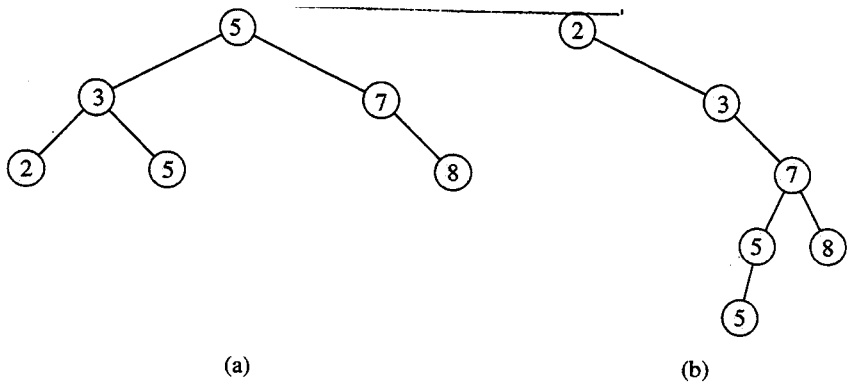

FIGURA 12.1 Árvores de pesquisa binária. Para qualquer nó x, as chaves na subárvore esquerda de x são no máximo $chave[x]$, e as chaves na subárvore direita de x são no mínimo $chave[x]$. Árvores de pesquisa binária diferentes podem representar o mesmo conjunto de valores. O tempo de execução do pior caso para a maioria das operações em árvores de pesquisa é proporcional à altura da árvore. (**a**) Uma árvore de pesquisa binária em 6 nós com altura 2. (**b**) Uma árvore de pesquisa binária menos eficiente, com altura 4 e que contém as mesmas chaves

As chaves em uma árvore de pesquisa binária são sempre armazenadas de modo a satisfazerem à ***propriedade de árvore de pesquisa binária***:

> Seja x um nó em uma árvore de pesquisa binária. Se y é um nó na subárvore esquerda de x, então $chave[y] \leq chave[x]$. Se y é um nó na subárvore direita de x, então $chave[x] \leq chave[y]$.

Desse modo, na Figura 12.1(a), a chave da raiz é 5, as chaves 2, 3 e 5 em sua subárvore esquerda não são maiores que 5, e as chaves 7 e 8 em sua subárvore direita não são menores que 5. A mesma propriedade é válida para todo nó na árvore. Por exemplo, a chave 3 na Figura 12.1(a) não é menor que a chave 2 em sua subárvore esquerda e não é maior que a chave 5 em sua subárvore direita.

A propriedade de árvore de pesquisa binária nos permite imprimir todas as chaves em uma árvore de pesquisa binária em seqüência ordenada, por meio de um simples algoritmo recursivo, chamado ***percurso de árvore em ordem***. O nome desse algoritmo deriva do fato de que a chave da raiz de uma subárvore é impressa entre os valores de sua subárvore esquerda e aqueles de sua subárvore direita. (De modo semelhante, um ***percurso de árvore de pré-ordem*** imprime a raiz antes dos valores em uma ou outra subárvore, e um ***percurso de árvore de pós-ordem*** imprime a raiz depois dos valores contidos em suas subárvores.) Para usar o procedimento a seguir com o objetivo de imprimir todos os elementos em uma árvore de pesquisa binária T, chamamos INORDER-TREE-WALK($raiz[T]$).

```
INORDER-TREE-WALK(x)
1 if x ≠ NIL
2    then INORDER-TREE-WALK(esquerda[x])
3         print chave[x]
4         INORDER-TREE-WALK(direita[x])
```

Como um exemplo, o percurso de árvore em ordem imprime as chaves em cada uma das duas árvores de pesquisa binária da Figura 12.1 na ordem 2, 3, 5, 5, 7, 8. A correção do algoritmo decorre por indução diretamente da propriedade de árvore de pesquisa binária.

Ele demora o tempo $\Theta(n)$ para percorrer uma árvore de pesquisa binária de n nós, pois, após a chamada inicial, o procedimento é chamado de forma recursiva exatamente duas vezes para cada nó na árvore – uma vez para seu filho da esquerda e uma vez para seu filho da direita. O

teorema a seguir apresenta uma prova mais formal de que o tempo para executar um percurso de árvore em ordem é linear.

Teorema 12.1
Se x é a raiz de uma subárvore de n nós, então a chamada INORDER-TREE-WALK(x) demora o tempo $\Theta(n)$.

Prova Seja $T(n)$ o tempo tomado por INORDER-TREE-WALK quando ele é chamado na raiz de uma subárvore de n nós. INORDER-TREE-WALK demora um espaço de tempo pequeno e constante em uma subárvore vazia (para o teste $x \neq$ NIL) e então $T(0) = c$ para alguma constante positiva c.

Para $n > 0$, suponha que INORDER-TREE-WALK seja chamado em um nó x cuja subárvore esquerda tem k nós e cuja subárvore direita tem $n - k - 1$ nós. O tempo para executar INORDER-TREE-WALK(x) é $T(n) = T(k) + T(n-k-1) + d$ para alguma constante positiva d que reflete o tempo para executar INORDER-TREE-WALK(x), excetuando-se o tempo gasto em chamadas recursivas.

Usamos o método de substituição para mostrar que $T(n) = \Theta(n)$, provando que $T(n) = (c + d)n + c$. Para $n = 0$, temos $(c + d) \cdot 0 + c = c = T(0)$. Para $n > 0$, temos

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c\,, \end{aligned}$$

o que completa a prova.

Exercícios

12.1-1
Trace árvores de pesquisa binária de altura 2, 3, 4, 5 e 6 sobre o conjunto de chaves {1, 4, 5, 10, 16, 17, 21}.

12.1-2
Qual a diferença entre a propriedade de árvore de pesquisa binária e a propriedade de heap mínimo (Seção 6.1)? A propriedade de heap pode ser usada para imprimir as chaves de uma árvore de n nós na seqüência ordenada em tempo $O(n)$? Explique como, ou por que não.

12.1-3
Forneça um algoritmo não recursivo que execute um percurso de árvore em ordem. (*Sugestão:* Existe uma solução fácil que usa uma pilha como uma estrutura de dados auxiliar e uma solução mais complicada, embora elegante, que não emprega nenhuma pilha mas pressupõe que é possível testar a igualdade entre dois ponteiros.)

12.1-4
Forneça algoritmos recursivos que executem caminhos de árvores de pré-ordem e pós-ordem no tempo $\Theta(n)$ em uma árvore de n nós.

12.1-5
Mostre que, considerando-se que a ordenação de n elementos demora o tempo $\Omega(n \lg n)$ no pior caso no modelo de comparação, qualquer algoritmo baseado em comparação para construção de uma árvore de pesquisa binária a partir de uma lista arbitrária de n elementos demora o tempo $\Omega(n \lg n)$ no pior caso.

12.2 Consultas em uma árvore de pesquisa binária

A operação mais comum executada sobre uma árvore de pesquisa binária é procurar por uma chave armazenada na árvore. Além da operação SEARCH, árvores de pesquisa binária podem admitir consultas como MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR. Nesta seção, examinaremos essas operações e mostraremos que cada uma delas pode ser admitida no tempo $O(h)$ sobre uma árvore de pesquisa binária de altura h.

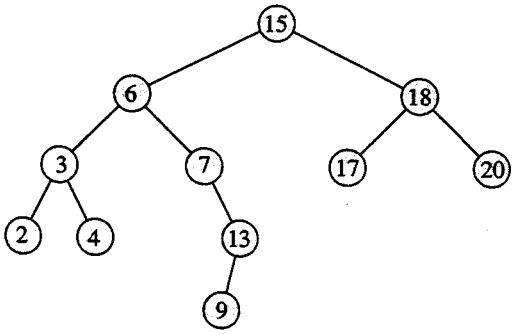

FIGURA 12.2 Consultas em uma árvore de pesquisa binária. Para procurar pela chave 13 na árvore, o caminho $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ é seguido a partir da raiz. A chave mínima na árvore é 2, que pode ser encontrada seguindo-se os ponteiros da *esquerda* a partir da raiz. A chave máxima 20 é encontrada seguindo-se os ponteiros da *direita* a partir da raiz. O sucessor do nó com chave 15 é o nó com chave 17, pois ele é a chave mínima na subárvore direita de 15. O nó com chave 13 não tem nenhuma subárvore direita, e assim seu sucessor é seu ancestral mais baixo cujo filho da esquerda também é um ancestral. Nesse caso, o nó com chave 15 é seu sucessor

Como pesquisar

Usamos o procedimento a seguir para procurar por um nó com uma determinada chave em uma árvore de pesquisa binária. Dado um ponteiro para a raiz da árvore e uma chave k, TREE-SEARCH retorna um ponteiro para um nó com chave k, se existir algum; caso contrário, ele retorna NIL.

```
TREE-SEARCH(x, k)
1 if x = NIL or k = chave[x]
2    then return x
3 if k < chave[x]
4    then return TREE-SEARCH(esquerda[x], k)
5    else return TREE-SEARCH(direita[x], k)
```

O procedimento começa sua pesquisa na raiz e traça um caminho descendente na árvore, como mostra a Figura 12.2. Para cada nó x que encontra, ele compara a chave k com $chave[x]$. Se as duas chaves são iguais, a pesquisa termina. Se k é menor que $chave[x]$, a pesquisa continua na subárvore esquerda de x, pois a propriedade de árvore de pesquisa binária implica que k não poderia estar armazenada na subárvore direita. Simetricamente, se k é maior que $chave[x]$, a pesquisa continua na subárvore direita. Os nós encontrados durante a recursão formam um caminho descendente a partir da raiz da árvore, e portanto o tempo de execução de TREE-SEARCH é $O(h)$, onde h é a altura da árvore.

O mesmo procedimento pode ser escrito de forma iterativa, "estendendo-se" a recursão para o interior de um loop **while**. Na maioria dos computadores, essa versão é mais eficiente.