da qual, e do $caminho_1 = 0$, obtemos $2 \ln n = 2 \ln 2 \lg n = 1.386 \lg n$, como uma aproximação para o $caminho_n$ (veja a seção A.4 no Apêndice A). Esta é uma aproximação para o número médio de comparações em uma árvore média. Este número é $O(\lg n)$, que está mais perto do melhor que do pior caso, e também indica que há pouco espaço para melhorias, pois $caminho_{melhor}/caminho_n \approx 0{,}7215$, e o comprimento do caminho médio no melhor caso é diferente por somente 27,85% do comprimento do caminho esperado no caso médio. A busca em uma árvore binária é, em consequência, muito eficiente na maioria dos casos, mesmo sem balancear a árvore. No entanto, isto é verdade somente para árvores criadas aleatoriamente, porque, naquelas altamente desbalanceadas e alongadas, cujas formas se parecem com listas ligadas, o tempo de pesquisa é $O(n)$, o que é inaceitável, considerando que uma eficiência $O(\lg n)$ pode ser obtida.

6.4 Percurso em árvores

Este é o processo de visitar cada nó da árvore exatamente uma vez. O percurso pode ser interpretado como colocar todos os nós em uma linha ou a linearização de uma árvore.

A definição de percurso especifica somente uma condição – visitar cada nó somente uma vez –, mas não a ordem na qual os nós são visitados. Por isso existem tantos percursos quantas permutações dos nós; para uma árvore com n nós, existem $n!$ percursos diferentes. A maioria deles, no entanto, é algo caótico e não indica muita regularidade, de modo que implementar tais cruzamentos não traz generalidade: para cada n um conjunto separado de procedimentos de percurso precisa ser implementado, e somente poucos deles podem ser usados para um diferente número de dados. Por exemplo, dois possíveis percursos da árvore da Figura 6.6c que podem ser de algum uso são as sequências 2, 10, 12, 20, 13, 25, 29, 31 e 29, 31, 20, 12, 2, 25, 10, 13. A primeira lista os números pares, e depois os ímpares em ordem ascendente; a segunda lista todos os nós de nível a nível da direita para a esquerda, do mais baixo até a raiz. A sequência 13, 31, 12, 2, 10, 29, 20, 25 não indica qualquer regularidade na ordem dos números ou na dos nós cruzados. Ela é apenas uma sequência aleatória de nó para nó provavelmente inútil. Não obstante, todas essas sequências são resultado de três percursos legítimos de $8! = 40.320$. Em face de tal abundância de percursos e da aparente inutilidade da maioria deles, gostaríamos de restringir nossa atenção a duas classes somente: percursos em extensão e profundidade.

6.4.1 Percurso em extensão

Percurso em extensão é visitar cada nó começando do nível mais baixo (ou mais alto) e movendo para baixo (ou para cima) nível a nível, visitando nós em cada nível da esquerda para a direita (ou da direita para a esquerda). Existem, portanto, quatro possibilidades, e uma delas – percurso em extensão de cima para baixo, da esquerda para a direita, na árvore da Figura 6.6c – resulta na sequência 13, 10, 25, 2, 12, 20, 31, 29.

A implementação deste tipo de percurso é direta quando uma fila é usada. Considere um percurso em extensão de cima para baixo, da esquerda para a direita. Depois que um nó é visitado, seus filhos, se houver algum, são colocados no final da fila e o nó no início da fila é visitado. Considerando que para um nó no nível n seus filhos estão no nível $n + 1$, colocando-se esses filhos no final da fila, eles serão visitados depois que todos os nós do nível n assim o forem. Deste modo, a restrição de que todos os nós no nível n precisam ser visitados antes de visitar quaisquer nós no nível $n + 1$ será satisfeita.

Uma implementação da função-membro correspondente é mostrada na Figura 6.10.

6.4.2 Percurso em profundidade

Percurso em profundidade prossegue, tanto quanto possível, à esquerda (ou direita), então se move para trás até a primeira encruzilhada, vai um passo para a direita (ou esquerda) e novamente, tanto

FIGURA 6.10 Implementação do percurso em extensão do topo para baixo, da esquerda para a direita.

```
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>*> queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                queue.enqueue(p->left);
            if (p->right != 0)
                queue.enqueue(p->right);
        }
    }
}
```

quanto possível, para a esquerda (ou direita). Repetimos este processo até que todos os nós sejam visitados. Esta definição, no entanto, não especifica com clareza exatamente quando os nós são visitados: antes de prosseguir para baixo na árvore ou depois de mover para trás? Existem algumas variações do percurso em profundidade.

Há três tarefas de interesse neste tipo de percurso:

V – visitar um nó
L – percorrer a subárvore esquerda (do inglês *left*)
R – percorrer a subárvore direita (do inglês *right*)

Um percurso organizado ocorre se estas tarefas são realizadas na mesma ordem para cada nó. As três tarefas podem estar ordenadas em 3! = 6 modos, portanto, há seis possíveis percursos em profundidade:

VLR VRL
LVR RVL
LRV RLV

Se o número de ordenações diferentes ainda parece grande, pode ser reduzido para três percursos, onde o movimento é sempre da esquerda para a direita e a atenção é focalizada na primeira coluna. A esses percursos são dados os seguintes nomes padrão:

VLR – cruzamento de árvore em pré-ordem
LVR – cruzamento de árvore em in-ordem
LRV – cruzamento de árvore em pós-ordem

Pequenas e elegantes funções podem ser implementadas diretamente a partir das descrições simbólicas desses três percursos, como mostrado na Figura 6.11.

Essas funções podem parecer simplistas, mas seu real poder reside na recursão; aliás, na recursão dupla. O trabalho é realmente feito pelo sistema na pilha em tempo de execução. Isto simplifica a codificação, mas coloca uma pesada carga sobre o sistema. Para melhor entender este processo, o percurso de árvore em in-ordem é discutido em detalhes.

No percurso em in-ordem, a subárvore esquerda do nó corrente é visitada primeiro, então o próprio nó, e, finalmente, a subárvore à direita. Tudo isto vale, obviamente, se a árvore não está vazia. Antes de analisar a pilha de execução, a saída dada pelo cruzamento em in-ordem é determinada por referência à Figura 6.12. As seguintes etapas correspondem às letras nessa figura:

(a) O nó 15 é a raiz na qual `inorder()` é chamada pela primeira vez. A função chama a si mesma para o filho esquerdo do nó 15, o nó 4.

(b) O nó 4 é não nulo, assim, `inorder()` é chamada no nó 1. Devido ao nó 1 ser uma folha (isto é, suas duas subárvores estão vazias), invocações de `inorder()` nas subárvores não resultam em outras chamadas recursivas de `inorder()`, pois a condição na instrução `if` não é satisfeita. Assim, depois que a chamada de `inorder()` para subárvore esquerda é terminada, o nó 1 é visitado e então uma rápida chamada de `inorder()` é executada para a subárvore direita nula do nó 1. Depois de encerrar a chamada para o nó 4, este é visitado. Este nó tem uma subárvore direita nula; `inorder()` é chamada, portanto, somente para verificar isto, e, logo depois de encerrar a chamada para o nó 15, este é visitado.

(c) O nó 15 tem uma subárvore direita, assim, `inorder()` é chamada para o nó 20.

(d) `inorder()` é chamada para o nó 16, ele é visitado, e então para a sua subárvore esquerda nula, que é visitada após o nó 16. Depois de uma rápida chamada a `inorder()` na subárvore direita nula do nó 16 e do retorno à chamada no nó 20, este é também visitado.

(e) `inorder()` é chamada no nó 25, depois na sua subárvore esquerda vazia, e então o nó 25 é visitado, e finalmente `inorder()` é chamada na subárvore direita vazia do nó 25.

Se a visita inclui imprimir os valores armazenados nos nós, então a saída é:

```
1 4 15 16 20 25
```

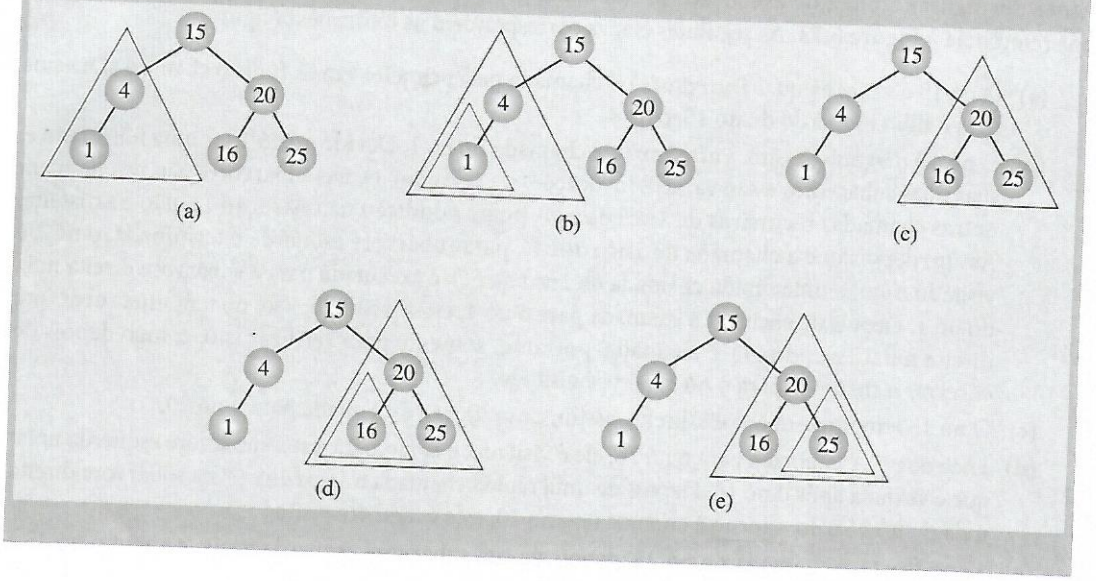
FIGURA 6.11 Implementação do percurso em profundidade.

```
template<class T>
void BST<T>::inorder(BSTNode<T> *p) {
    if (p != 0) {
        inorder(p->left);
        visit(p);
        inorder(p->right);
    }
}

template<class T>
void BST<T>::preorder(BSTNode<T> *p) {
    if (p != 0) {
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}

template<class T>
void BST<T>::postorder(BSTNode<T>* p) {
    if (p != 0) {
        postorder(p->left);
        postorder(p->right);
        visit(p);
    }
}
```

FIGURA 6.12 Percurso de árvore in-ordem.



A chave para o percurso é que as três tarefas, L, V e R, são realizadas para cada nó separadamente. Isto significa que o percurso da subárvore direita de um nó é mantido pendente até que as primeiras duas tarefas, L e V, sejam realizadas. Se as últimas duas estão terminadas, podem ser canceladas, como na Figura 6.13.

Para apresentar o modo como `inorder()` trabalha, o comportamento da pilha em tempo de execução é observado. O número entre parênteses na Figura 6.14 indica o endereço de retorno mostrado no lado esquerdo do código para `inorder()`.

```
      template<class T>
      void BST<T>::inorder(BSTnode<T> *node) {
           if (node != 0) {
/*1*/           inorder(node->left);
/*2*/           visit(node);
/*3*/           inorder(node->right);
/*4*/      }
      }
```

Um retângulo com uma seta para cima e um número indica o valor corrente de `node` colocado na pilha. Por exemplo, ↑4 significa que `node` aponta para o nó da árvore cujo valor é o número 4. A Figura 6.14 mostra as mudanças da pilha em tempo de execução quando `inorder()` é executada para a árvore da Figura 6.12.

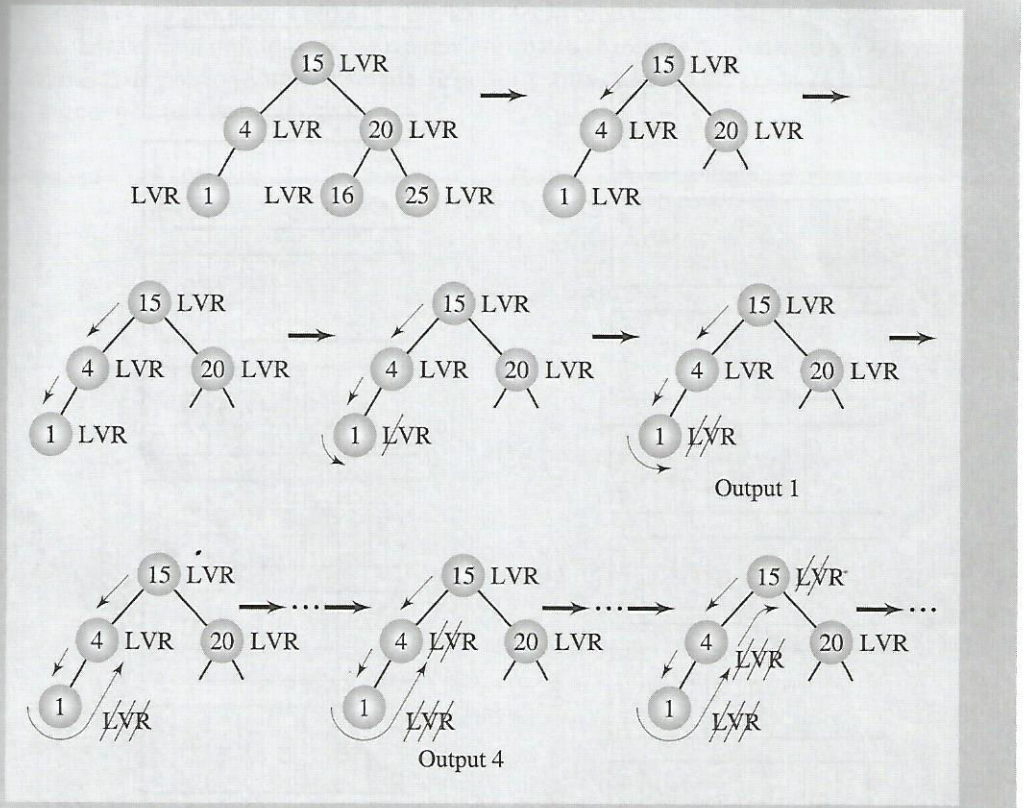
(a) Inicialmente, a pilha em tempo de execução está vazia (ou melhor, assume-se que está vazia, desprezando-se o que estiver armazenado nela antes da primeira chamada para `inorder()`).

(b) Sobre a primeira chamada, o endereço de retorno de `inorder()` e o valor de `node`, ↑15, são colocados na pilha em tempo de execução. A árvore, apontada por `node`, não está vazia, a condição na instrução `if` é satisfeita e `inorder()` é chamada outra vez com o nó 4.

(c) Antes que ela seja executada, o endereço de retorno, (2), e o valor corrente de `node`, ↑4, são colocados na pilha. Como `node` não é nulo, `inorder()` está para ser invocado pelo filho esquerdo de node, ↑1.

(d) Primeiro, o endereço de retorno, (2), e o valor de `node` são colocados na pilha.

FIGURA 6.13 Detalhes de diversas primeiras etapas do percurso em in-ordem.



(e) `inorder()` é chamada com o filho esquerdo do nó 1. O endereço (2) e o valor corrente do parâmetro de `node`, nulo, são armazenados na pilha. Como `node` é nulo, `inorder()` é imediatamente deixado; na saída, o registro de ativação é removido da pilha.

(f) O sistema vai agora à sua pilha em tempo de execução, restaura o valor de `node`, ↑1, executa a instrução sob (2) e imprime o número 1. Como `node` não está completamente processado, seu valor e o endereço (2) ainda estão na pilha.

(g) Com o filho à direita de `node` ↑1, a instrução sob (3) é executada, é a próxima chamada para `inorder()`. Primeiro, no entanto, o endereço (4) e o valor corrente de `node`, nulo, são colocados na pilha. Como `node` é nulo, `inorder()` é deixado; na saída, a pilha é limpa.

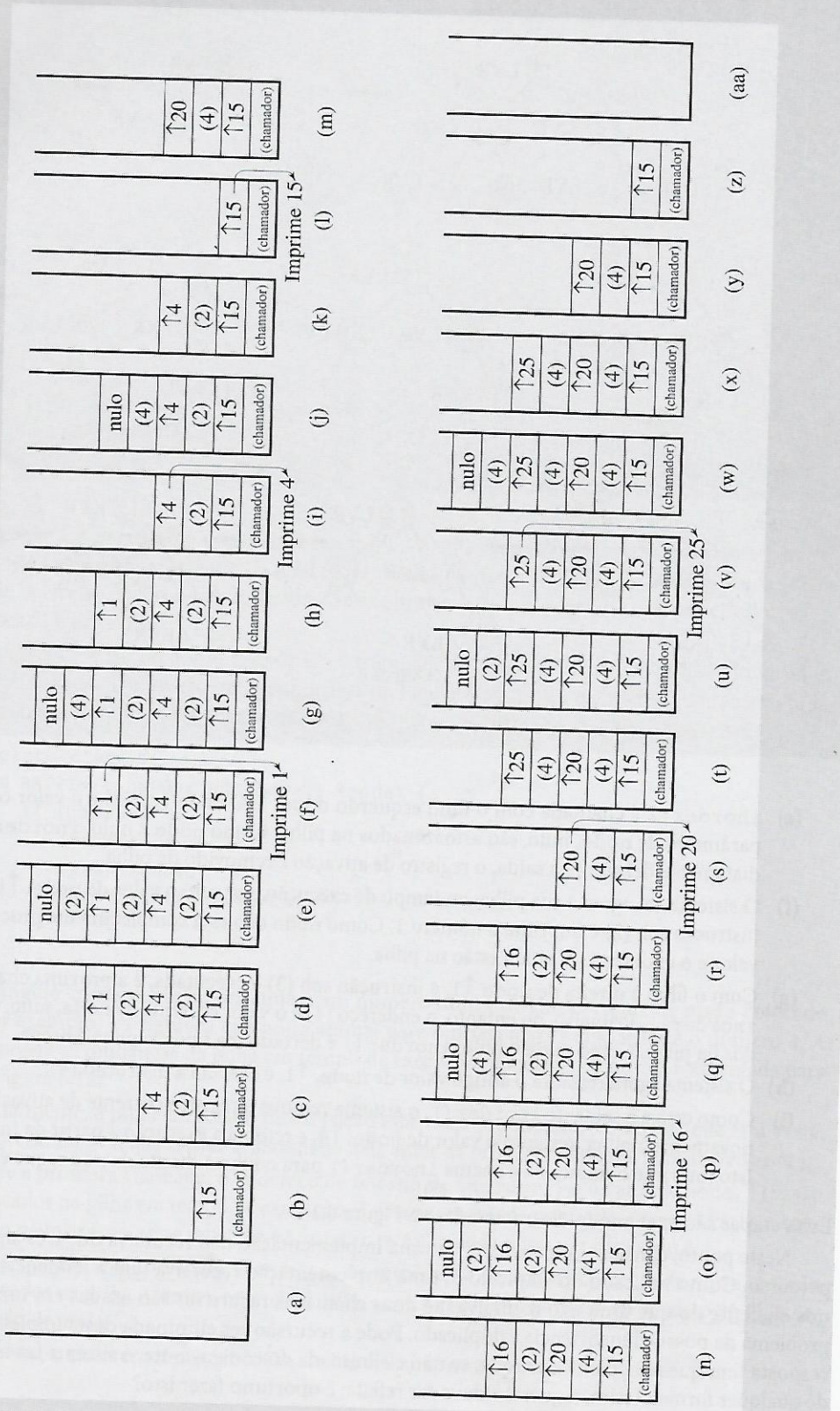
(h) O sistema agora restaura o antigo valor de `node`, ↑1, e executa a instrução (4).

(i) Como esta é a saída de `inorder()`, o sistema remove o registro corrente de ativação e remete novamente à pilha, restaura o valor de `node`, ↑4, e retoma a execução a partir da instrução (2). Isto imprime o número 4 e chama `inorder()` para o filho à direita de `node`, que é nulo.

Essas etapas são apenas o início, mostradas na Figura 6.14.

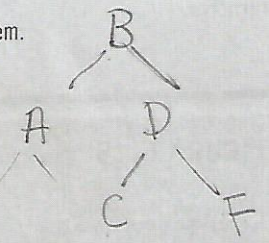
Neste ponto, considere o problema de uma implementação não recursiva dos três algoritmos de percurso. Como indicado no Capítulo 5, uma implementação recursiva tem a tendência de ser menos eficiente do que uma não recursiva. Se duas chamadas recursivas são usadas em uma função, o problema da possível ineficiência é duplicado. Pode a recursão ser eliminada desta implementação? A resposta tem que ser positiva, porque, se não é eliminada do código-fonte, o sistema faz isto para nós de qualquer forma. Assim, a questão deve ser refeita: é oportuno fazer isto?

FIGURA 6.14 Mudanças na pilha em tempo de execução durante o percurso em in-ordem.



Veja primeiro uma versão não recursiva do percurso de arvore em pré-ordem, mostrada na Figura 6.15. A função `iterativePreorder()` é duas vezes tão grande quanto `preorder()`, mas é ainda pequena e legível; no entanto, usa pesadamente a pilha. Em consequência, funções de suporte são necessárias para processar a pilha, e a implementação total não é tão pequena. Embora duas chamadas recursivas sejam omitidas, existem agora até quatro chamadas por iteração do laço `while`: até duas chamadas de `push()`, uma chamada de `pop()` e uma chamada de `visit()`. É difícil considerar isto como melhoria da eficiência.

FIGURA 6.15 Uma implementação não recursiva no percurso de árvore em pré-ordem.




```
template<class T>
void BST<T>::iterativePreorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T> *p = root;
    if (p != 0) {
        travStack.push(p);
        while (!travStack.empty()) {
            p = travStack.pop();
            visit(p);
            if (p->right != 0)
                travStack.push(p->right);
            if (p->left != 0)   // filho a esquerda colocado depois do da direita
                travStack.push(p->left); // estar no topo
                                         // da pilha;
        }
    }
}
```

Nas implementações recursivas dos três percursos, note que a única diferença está na ordem das linhas de código. Por exemplo, em `preorder()`, primeiro um nó é visitado e então há chamadas para a subárvore esquerda e para a direita. Por outro lado, em `postorder()`, visitar um nó precede ambas as chamadas. Podemos transformar tão facilmente a versão não recursiva de um percurso em pré-ordem da esquerda para a direita em uma não recursiva de um percurso em pós-ordem da esquerda para a direita? Infelizmente, não. Em `iterativePreorder()`, a visita ocorre antes que ambos os filhos sejam colocados na pilha. Mas esta ordem, na realidade, não importa. Se o filho é colocado primeiro e então o nó é visitado, isto é, se `visit(p)` é colocado depois que ambos chamam `push()`, a implementação resultante é ainda um cruzamento em pré-ordem. O que importa aqui é que `visit()` tem que seguir `pop()`, e esta última tem que preceder ambas as chamadas de `push()`. Em consequência, as implementações não recursivas dos percursos in-ordem e pós-ordem têm que ser desenvolvidas de forma independente.

Uma versão não recursiva do percurso em pós-ordem pode ser obtida com certa facilidade se observamos que a sequência gerada por um percurso em pós-ordem da esquerda para a direita (uma ordem LRV) é a mesma que a sequência inversa gerada por um percurso em pré-ordem da direita para a esquerda (uma ordem VRL). Neste caso, a implementação de `iterativePreorder()` pode ser adotada para criar `iterativePostorder()`. Isto significa que duas pilhas têm que ser usadas, uma para visitar cada nó na ordem inversa depois que um percurso em pré-ordem, da direita para a esquerda, é terminado. É, no entanto, possível desenvolver uma função para o percurso em pós-ordem que coloca na pilha um nó que tem dois descendentes, uma vez antes de percorrer sua subárvore esquerda e outra antes de percorrer sua subárvore direita. Um ponteiro auxiliar `q` é usado para

distinguir estes dois casos. Os nós com um descendente são colocados somente uma vez, e as folhas não necessitam ser colocadas de modo algum (Figura 6.16).

Um percurso não recursivo em in-ordem de árvore também é complicado. Uma implementação possível é dada na Figura 6.17. Em casos assim, podemos claramente ver o poder da recursão: `iterativeInorder()` é quase ilegível, e sem uma explicação completa não é fácil determinar o propósito desta função. Por outro lado, a função `inorder()` recursiva demonstra imediatamente um propósito e é lógica. Em consequência, `iterativeInorder()` pode ser defendida em um caso apenas: se é mostrado que existe um ganho substancial no tempo de execução e que a função é chamada com frequência em um programa. Caso contrário, `inorder()` é preferível à sua contraparte iterativa.

FIGURA 6.16 Uma implementação não recursiva do percurso de árvore em pós-ordem.

```
template<class T>
void BST<T>::iterativePostorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T>* p = root, *q = root;
    while (p != 0) {
        for ( ; p->left != 0; p = p->left)
            travStack.push(p);
        while (p->right == 0 || p->right == q) {
            visit(p);
            q = p;
            if (travStack.empty())
                return;
            p = travStack.pop();
        }
        travStack.push(p);
        p = p->right;
    }
}
```

FIGURA 6.17 Uma implementação não recursiva do percurso de árvore em in-ordem.

```
template<class T>
void BST<T>::iterativeInorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T> *p = root;
    while (p != 0) {
        while (p != 0) {            // empilha o filho a direita (se houver algum)
            if (p->right)           // e o proprio no quando indo
                travStack.push(p->right); // para a esquerda;
            travStack.push(p);
            p = p->left;
        }
        p = travStack.pop();        // extrai um no sem filho a esquerda
        while (!travStack.empty() && p->right == 0) { // visite-o
```

FIGURA 6.17 Uma implementação não recursiva do percurso de árvore em in-ordem. (*continuação*)

```
            visit(p);                  // e todos os nos sem filho
            p = travStack.pop();       // a direita;
        }
        visit(p);                      // visite tambem o primeiro no com
        if (!travStack.empty())        // um filho à direita (se houver algum);
            p = travStack.pop();
        else p = 0;
    }
}
```

6.4.3 Percurso em profundidade sem pilha

Árvores alinhadas

As funções de percurso analisadas na seção anterior foram tanto recursivas como não recursivas, mas ambos os tipos usaram uma pilha tanto implícita quanto explicitamente para armazenar a informação sobre os nós cujos processamentos não tinham sido terminados. No caso das funções recursivas, a pilha em tempo de execução foi utilizada. No das variantes não recursivas uma pilha explicitamente definida e mantida pelo usuário foi usada. A questão é que algum tempo adicional tem que ser gasto para manter a pilha, e mais algum espaço ser colocado à parte para a própria árvore. No pior caso, quando a árvore é desequilibrada desfavoravelmente, a pilha pode conter informação sobre quase todo nó da árvore, uma séria questão para árvores muito grandes.

É mais eficiente incorporar a pilha como parte da árvore. Isto é feito incorporando *linhas* em um nó. Linhas são ponteiros para o predecessor e para o sucessor do nó, de acordo com o percurso na ordem de entrada, e as árvores cujos nós usam linhas são chamadas *árvores alinhadas*. Quatro ponteiros são necessários para cada nó da árvore, que novamente tomam espaço valioso.

O problema pode ser resolvido sobrecarregando ponteiros existentes. Nas árvores, os ponteiros à esquerda e à direita são para filhos, mas também podem ser usados como ponteiros para os predecessores e para os sucessores, desta forma sendo sobrecarregados com significado. Como distinguimos esses significados? Para um operador sobrecarregado, o contexto é sempre um fator de desambiguação. Nas árvores, no entanto, um novo membro de dados tem que ser usado para indicar o significado corrente dos ponteiros.

Como um ponteiro pode apontar para um nó de cada vez, o esquerdo é tanto ponteiro para o filho à esquerda como para um predecessor. Analogamente, o ponteiro direito aponta tanto para a subárvore à direita como para o sucessor (Figura 6.18a).

A Figura 6.18a sugere que ambos os ponteiros para os predecessores e para os sucessores têm que ser mantidos, o que nem sempre é o caso. Pode ser suficiente usar somente uma linha, como mostrado na implementação do percurso em in-ordem de uma árvore alinhada, que exige apenas ponteiros para os sucessores (Figura 6.18b).

A função é relativamente simples. A linha tracejada na Figura 6.18b indica a ordem na qual `p` acessa os nós na árvore. Note que somente uma variável, `p`, é necessária para percorrer a árvore. Nenhuma pilha é necessária; em consequência, espaço é economizado. Mas é realmente economizado? Como indicado, os nós exigem um membro de dados que indicam como o ponteiro à direita está sendo usado. Na implementação de `threadedInorder()`, o membro de dados booleano `sucessor` desempenha este papel, como mostrado na Figura 6.19. Por isso, `sucessor` exige somente um bit da memória do computador, insignificante em comparação com os outros campos. No entanto, os detalhes são altamente dependentes da implementação. O sistema operacional quase certamente recheia