FIGURA 6.19 Implementação de uma árvore alinhada genérica e o percurso em in-ordem de uma árvore alinhada. (*continuação*)

```
    ThreadedTree()    {
        root = 0;
    }
    void insert(const T&);        // Figura 6.24
    void inorder();
    . . . . . .
protected:
    ThreadedNode<T>* root;
    . . . . . .
};

#endif
template<class T>
void ThreadedTree<T>::inorder() {
    ThreadedNode<T> *prev, *p = root;
    if (p != 0) {                 // processe somente arvores não vazias;
        while (p->left != 0)      // va para o no mais a esquerda;
            p = p->left;
        while (p != 0) {
            visit(p);
            prev = p;
            p = p->right;         // va para o no a direita e somente
            if (p != 0 && prev->successor == 0) // se ele e um
                while (p->left != 0)// descendente vá para o
                    p = p->left;  // no mais a esquerda, caso contrario
        }                         // visite o sucessor;
    }
}
```

O percurso em pós-ordem é apenas levemente mais complicado. Primeiro, é criado um nó falso, que tem a raiz como seu descendente à esquerda. No processo de percurso uma variável pode ser usada para verificar o tipo da ação corrente. Se a ação é percurso à esquerda e o nó corrente tem um descendente à esquerda, o descendente é percorrido; caso contrário, a ação é mudada para percurso à direita. Se a ação é percurso à direita e o nó corrente tem um descendente não alinhado à direita, o descendente é percorrido e a ação é mudada para percurso à esquerda; caso contrário, a ação muda para visitar um nó. Se a ação é visitar um nó, então o nó corrente é visitado e, posteriormente, seu sucessor pós-ordem tem que ser encontrado. Se o ascendente do nó corrente é acessível através de uma linha (isto é, o nó corrente é o filho à esquerda do ascendente), o percurso é estabelecido para continuar com o descendente à direita do ascendente. Se o nó corrente não tem descendente à direita, ele é o final da cadeia de nós estendida à direita. Primeiro, o início da cadeia é atingido através da linha do nó corrente, depois as referências à direita dos nós na cadeia são invertidas, e, finalmente, a cadeia é varrida para trás, cada nó é visitado, e, em seguida, as referências à direita são restauradas aos seus ajustes prévios.

Percurso por transformação da árvore

O primeiro conjunto de algoritmos de percurso analisado previamente neste capítulo necessitou de uma pilha a fim de reter a informação necessária para o processamento bem-sucedido. As árvores alinhadas incorporaram uma pilha como parte da árvore, ao custo de estender os nós por um campo

para fazer uma distinção entre a interpretação do ponteiro à direita como um ponteiro para o filho ou para o sucessor. Dois de tais campos de rótulo são necessários se tanto o sucessor quanto o predecessor são considerados. No entanto, é possível cruzar uma árvore sem usar qualquer pilha ou linha. Existem muitos desses algoritmos, todos possíveis fazendo-se mudanças temporárias na árvore durante o percurso. Essas mudanças consistem em reatribuir novos valores para alguns ponteiros, mas a árvore pode temporariamente perder sua estrutura, que necessita ser restaurada antes que o percurso termine. A técnica é ilustrada por um elegante algoritmo idealizado por Joseph M. Morris aplicado ao percurso em in-ordem.

Primeiro, é fácil notar que o percurso em in-ordem é muito simples para árvores degeneradas, nas quais nenhum nó tem um filho à esquerda (veja a Figura 6.1e). Nenhuma subárvore esquerda tem que ser considerada para qualquer nó. Em consequência, as três etapas usuais, LVR (visitar a subárvore esquerda, visitar o nó, visitar a subárvore direita), para cada nó no percurso em in-ordem se tornam duas etapas, VR. Nenhuma informação sobre o status corrente do nó que está sendo processado necessita ser retida antes de percorrer seu filho esquerdo, simplesmente porque não há um filho à esquerda. O algoritmo de Morris leva em conta esta observação temporariamente transformando a árvore de modo que o nó que está sendo processado não tenha filho à esquerda; por isso, esse nó pode ser visitado e sua subárvore à direita processada. O algoritmo pode ser sumarizado como:

```
MorrisInorder ()
  while não terminado
    if o nó não tem descendente à esquerda
       visite-o;
       vá para a direita;
    else faça desse nó o filho à direita do nó mais à direita no seu descendente à esquerda;
         vá para esse descendente à esquerda;
```

FIGURA 6.20 Implementação do algoritmo de Morris para o percurso em in-ordem.

```
template<class T>
void BST<T>::MorrisInorder() {
    BSTNode<T> *p = root, *tmp;
    while (p != 0)
        if (p->left == 0) {
            visit(p);
            p = p->right;
        }
        else {
            tmp = p->left;
            while (tmp->right != 0 && // va para o no mais a direita
                   tmp->right != p)   // da subárvore esquerda ou
                tmp = tmp->right;     // para o ascendente temporario
            if (tmp->right == 0) {    // de p; se 'true'
                tmp->right = p;       // o no mais a direita foi
                p = p->left;          // atingido, faca-o um
            }                         // ascendente temporário da
            else {                    // raiz corrente, caso contrario
                                      // um descendente temporario
                visit(p);             // foi encontrado; visite o no p
                tmp->right = 0;       // e entao corte o ponteiro
```

FIGURA 6.20 Implementação do algoritmo de Morris para o percurso em in-ordem. *(continuação)*

```
                                    // direito da corrente
                p = p->right;       // ascendente, por onde ele
            }                       // cessa de ser um ascendente;
        }
}
```

Este algoritmo percorre com sucesso a árvore, mas somente uma vez, pois destrói sua estrutura original. Em consequência, alguma informação precisa ser retida para permitir que a árvore seja restaurada à sua forma original. Isto é obtido retendo-se o ponteiro esquerdo do nó movido para baixo em sua subárvore à direita, como no caso dos nós 10 e 5 na Figura 6.21.

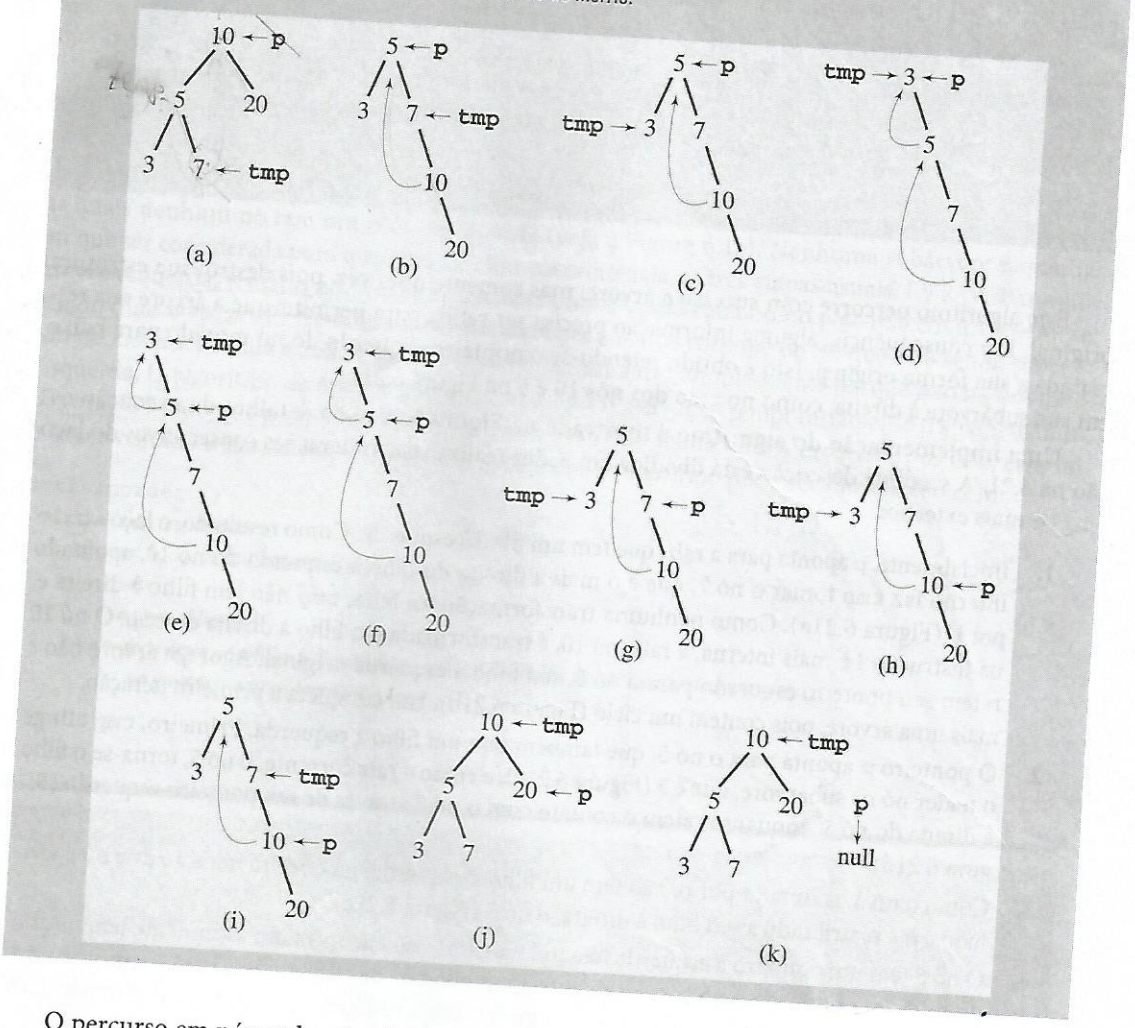
Uma implementação do algoritmo é mostrada na Figura 6.20, e os detalhes da execução estão na 6.21. A seguinte descrição está dividida em ações realizadas em iterações consecutivas do laço `while` mais externo:

1. Inicialmente, `p` aponta para a raiz, que tem um filho à esquerda. Como resultado, o laço `while` interno faz `tmp` tomar o nó 7, que é o mais à direita do filho à esquerda do nó 10, apontado por `p` (Figura 6.21a). Como nenhuma transformação foi feita, `tmp` não tem filho à direita e, na instrução `if` mais interna, a raiz, nó 10, é transformada em filho à direita de `tmp`. O nó 10 retém seu ponteiro esquerdo para o nó 5, seu filho à esquerda original. Agora, a árvore não é mais uma árvore, pois contém um ciclo (Figura 6.21b). Isto completa a primeira iteração.
2. O ponteiro `p` aponta para o nó 5, que também tem um filho à esquerda. Primeiro, `tmp` atinge o maior nó na subárvore, que é 3 (Figura 6.21c), e então a raiz corrente, o nó 5, torna-se o filho à direita do nó 3, enquanto retém o contato com o nó 3 através de seu ponteiro esquerdo (Figura 6.21d).
3. Como o nó 3, apontado por `p`, não tem um filho à esquerda, na terceira iteração este nó é visitado e `p` é reatribuído a seu filho à direita, o nó 5 (Figura 6.21e).
4. O nó 5 tem um ponteiro à esquerda não nulo; assim, `tmp` encontra um ascendente temporário do nó 5, que é o mesmo nó correntemente apontado por `tmp` (Figura 6.21f). A seguir, o nó 5 é visitado e a configuração da árvore na Figura 6.21b é restabelecida, ajustando-se o ponteiro direito do nó 3 para nulo (Figura 6.21g).
5. O nó 7, apontado agora por `p`, é visitado, e `p` se move para baixo, para seu filho à direita (6.21h).
6. `tmp` é atualizado para apontar para o ascendente temporário do nó 10 (Figura 6.21i). A seguir, este nó é visitado e então restabelecido ao seu status de raiz, anulando-se o ponteiro direito do nó 7 (Figura 6.21j).
7. Finalmente, o nó 20 é visitado sem trabalho posterior, pois não tem filho à esquerda nem sua posição alterada.

Isto completa a execução do algoritmo de Morris. Note que existem sete iterações do laço `while` mais externo para somente cinco nós na árvore da Figura 6.21. A razão disto é o fato de haver dois filhos na árvore; assim, o número de iterações extras depende do número de filhos à esquerda na árvore inteira. O algoritmo tem um desempenho pior para árvores com grande número de tais filhos.

O percurso em pré-ordem é fácil de obter, a partir do percurso em in-ordem, movendo `visit()` da cláusula `else` mais interna para a cláusula `if` mais interna. Deste modo, um nó é visitado antes de uma transformação de árvore.

FIGURA 6.21 Percurso de árvore com o método de Morris.



O percurso em pós-ordem também pode ser obtido, a partir do percurso em in-ordem , primeiro criando um nó falso, cujo descendente à esquerda é a árvore que está sendo processada e o à direita é nulo. Então, esta árvore temporariamente estendida é percorrida como em in-ordem, exceto que, na cláusula `else` mais interna, depois de encontrar um ascendente temporário, nós entre `p->left` (incluído) e `p` (excluído) estendido para a direita em uma árvore modificada são processados na ordem inversa. Para processá-los em tempo constante, a cadeia de nós é varrida para baixo e os ponteiros direitos são invertidos para apontar para os ascendentes dos nós. Então, a mesma cadeia é varrida para cima, cada nó é visitado e os ponteiros direitos são restaurados aos seus ajustes originais.

Quão eficientes são os procedimentos de percurso discutidos nesta seção? Todos operam no tempo $\Theta(n)$, a implementação alinhada exige $\Theta(n)$ mais espaço para as linhas do que as árvores binárias de pesquisa não alinhadas, e tanto os percursos recursivos como os não recursivos exigem $O(n)$ de espaço adicional (na pilha do tempo de execução ou na definida pelo usuário). Diversas dúzias de rodadas em árvores geradas aleatoriamente de 5.000 nós indicam que, para as rotinas de percurso em pré-ordem e in-ordem (recursiva, iterativa, de Morris e alinhada), a diferença no tempo de execução é somente da ordem de 5% a 10%. Os percursos de Morris têm uma vantagem inegável sobre os outros: não exigem espaço adicional. Os percursos recursivos baseiam-se na pilha em tempo de execução, que pode transbordar com árvores muito profundas. Os percursos iterativos também usam uma pilha e, embora ela possa transbordar, o problema não é tão imediato como no caso da pilha

em tempo de execução. As árvores alinhadas usam nós maiores do que os usados por árvores não alinhadas, o que usualmente não deve ser problema. Mas tanto a implementação iterativa quanto a alinhada são muito menos intuitivas do que as recursivas; em consequência, a clareza da implementação e os tempos de execução comparáveis claramente favorecem, na maioria das situações, as implementações recursivas sobre outras implementações.

6.5 Inserção

Busca em árvore binária não modifica a árvore. Ela varre a árvore de um modo predeterminado para acessar algumas das chaves da árvore ou todas, mas a própria árvore permanece inalterada depois da operação. Os percursos de árvores podem modificá-las, mas também deixá-las na mesma condição. Modificá-las ou não depende das ações prescritas por `visit()`. Existem certas operações que sempre fazem mudanças sistemáticas na árvore, tais como adicionar nós, removê-los, modificar elementos, fundir árvores e balancear árvores para reduzir sua altura. Esta seção trata somente da inserção de um nó em uma árvore binária de busca.

Para inserir um novo nó com a chave `el`, um nó da árvore com uma extremidade sem saída tem que ser atingido, e o novo nó tem de ser anexado a ele. Um nó da árvore é encontrado usando a mesma técnica que a pesquisa de árvore usou: a chave `el` é comparada com a chave de um nó que está sendo examinado atualmente durante uma varredura de árvore. Se `el` é menor que a chave, o filho da esquerda (se houver) de `p` é testado. Se o filho de `p` a ser testado está vazio, a varredura será interrompida e o novo nó torna-se este filho. O procedimento está ilustrado na Figura 6.22. A Figura 6.23 contém o algoritmo para inserir um nó.

FIGURA 6.22 Inserindo nós em árvores binárias de busca.

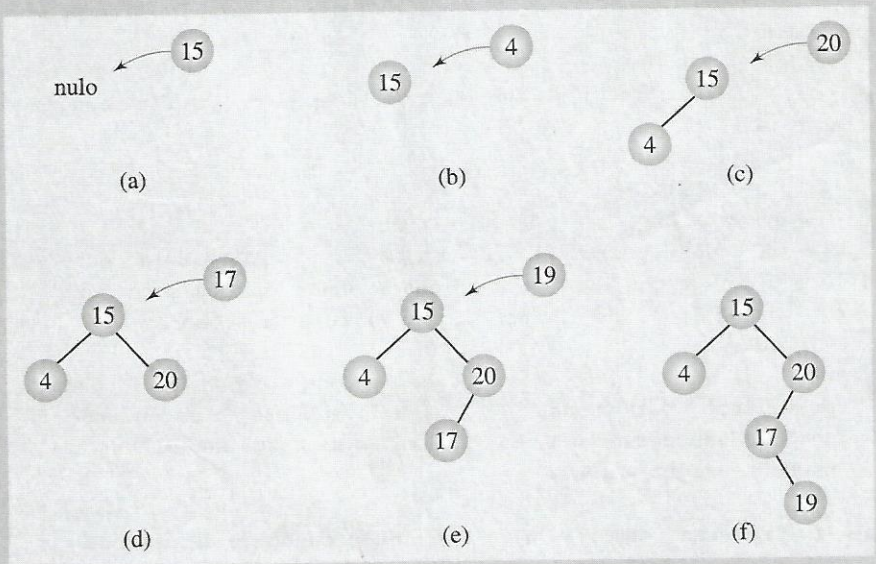


FIGURA 6.23 Implementação do algoritmo de inserção.

```
template<class T>
```