**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.

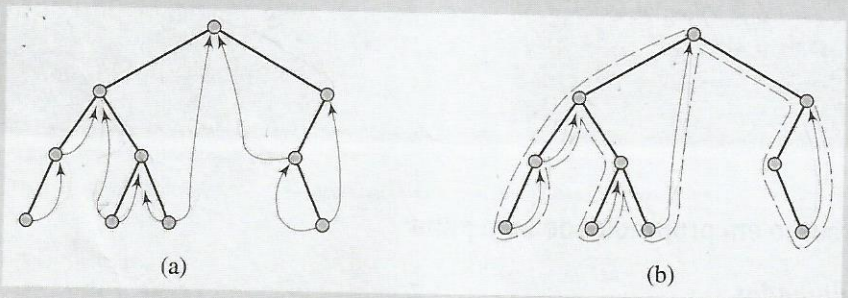
Como um ponteiro pode apontar para um nó de cada vez, o esquerdo é tanto ponteiro para o filho à esquerda como para um predecessor. Analogamente, o ponteiro direito aponta tanto para a subárvore à direita como para o sucessor (Figura 6.18a).

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

uma estrutura de bits com bits adicionais para um alinhamento apropriado das palavras da máquina. Se for assim, `sucessor` necessita de pelo menos um byte, se não uma palavra inteira, invalidando o argumento de economizar espaço usando árvores alinhadas.

**FIGURA 6.18** (a) Uma árvore alinhada e (b) um caminho do percurso em in-ordem em uma árvore alinhada com somente sucessores à direita.




As árvores alinhadas podem também ser usadas para percorrer em pré-ordem e pós-ordem. No percurso em pré-ordem, o nó corrente é visitado primeiro, depois o percurso continua com seu descendente esquerdo, se houver algum, ou com o descendente direito, se houver. Se o nó corrente é uma folha, linhas são usadas para ir através da cadeia de seus já visitados sucessores na ordem de entrada a fim de reiniciar o percurso com o descendente à direita do último sucessor.

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

```
//************************    genThreaded.h  *********************
//              Árvore binária alinhada genérica de busca

#ifndef THREADED_TREE
#define THREADED_TREE

template<class T>
class ThreadedNode {
public:
    ThreadedNode() {
        left = right = 0;
    }
    ThreadedNode(const T& el, ThreadedNode *l = 0, ThreadedNode *r = 0) {
        el = e; left = 1; right = r; successor = 0;
    }
    T el;
    ThreadedNode *left, *right;
    unsigned int successor : 1;
};

template<class T>
class ThreadedTree {
public:
```

**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