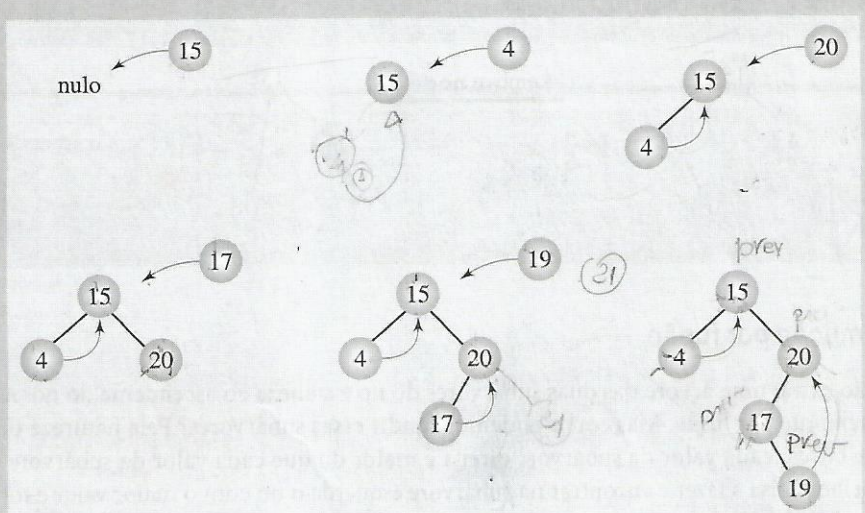
Ao analisarmos o problema de percorrer as árvores binárias, três abordagens foram apresentadas: cruzamento com o auxílio de uma pilha, com o auxílio de linhas e através da transformação da árvore. A primeira abordagem não modifica a árvore durante o processo. A terceira a modifica, mas a restaura para a mesma condição de quando a iniciou. Somente a segunda abordagem necessita de alguma operação preparatória na árvore para se tornar praticável: exige linhas. Essas linhas podem ser criadas cada vez antes de o percurso iniciar sua tarefa, e removidas cada vez que é terminado. Se o percurso não é realizado com frequência, isto se torna uma opção viável. Outra abordagem é manter as linhas em todas as operações na árvore ao inserir um novo elemento na árvore binária de busca.

A função para inserir um nó em uma árvore alinhada é uma simples extensão de `insert()` para as árvores binárias de busca regulares, usada para ajustar as linhas sempre que possível. Esta função é para o percurso de árvore em in-ordem, e cuida somente dos sucessores, não dos predecessores.

Um nó com um filho à direita tem um sucessor em algum lugar na sua subárvore à direita. Em consequência, não necessita de uma linha de sucessor. Tais linhas são necessárias para permitir escalar a árvore, não para descê-la. Um nó sem filho à direita tem seu sucessor em algum lugar acima dele. Exceto para um nó, todos os nós sem filhos à direita terão linhas para os seus sucessores. Se um nó se torna o filho à direita de outro, ele herda o sucessor de seu novo ascendente. Se um nó se torna um filho à esquerda de outro, este ascendente se torna seu sucessor. A Figura 6.24 mostra a implementação deste algoritmo. As primeiras poucas inserções estão na Figura 6.25.

FIGURA 6.25 Inserindo nós em uma árvore alinhada.



6.6 Remoção

Remoção de um nó é outra operação necessária para manter uma árvore binária de busca. O nível de complexidade em realizar a operação depende da posição do nó a ser removido da árvore. É muito mais difícil remover um nó que tem duas subárvores do que uma folha; a complexidade do algoritmo de remoção é proporcional ao número de filhos que o nó tem. Existem três casos de remoção de um nó da árvore binária de busca:

1. O nó é uma folha e não tem filhos. Este é o caso mais fácil de se tratar. O ponteiro apropriado de seu ascendente é ajustado para nulo e o nó removido por `delete`, como na Figura 6.26.

2. O nó tem um filho. Este caso não é complicado. O ponteiro do ascendente para o nó é reajustado para apontar para o filho do nó. Deste modo, os filhos do nó são elevados em um nível e todos os antecessores perdem um grau de descendência em suas designações de parentesco. Por exemplo, o nó que contém 20 (veja Figura 6.27) é removido ajustando o ponteiro direito do seu ascendente, que contém 15, para apontar para o único filho de 20, que é 16.
3. O nó tem dois filhos. Neste caso, nenhuma operação de uma etapa pode ser realizada, pois os ponteiros direito e esquerdo do ascendente não podem apontar para ambos os filhos do nó ao mesmo tempo. Esta seção discute duas soluções diferentes para este problema.

FIGURA 6.26 Removendo uma folha.

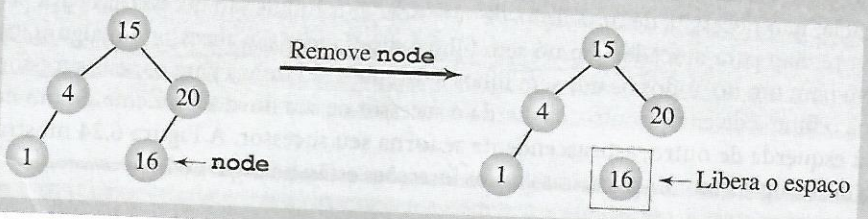
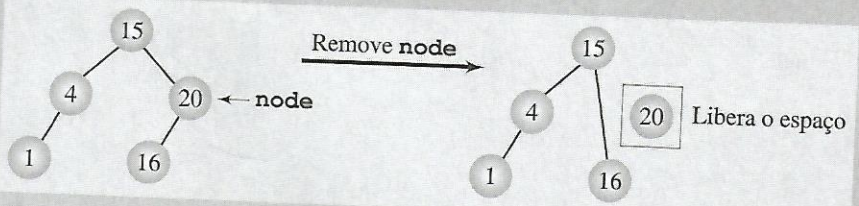


FIGURA 6.27 Removendo um nó com um filho.



6.6.1 Remoção por fusão

Esta solução extrai uma árvore das duas subárvores do nó e a anexa ao ascendente do nó. A técnica é chamada *remoção por fusão*. Mas como podemos fundir essas subárvores? Pela natureza das árvores binárias de busca, cada valor da subárvore direita é maior do que cada valor da subárvore esquerda; assim, a melhor coisa a fazer é encontrar na subárvore esquerda o nó com o maior valor e torná-lo um ascendente da subárvore direita. Simetricamente, pode-se encontrar na subárvore direita o nó com o menor valor e torná-lo um ascendente da subárvore esquerda.

O nó desejado é aquele mais à direita da subárvore esquerda. Ele pode ser localizado movendo-se ao longo da subárvore e tomando os ponteiros direitos até que nulo seja encontrado. Isto significa que este nó não terá um filho direito, e não há perigo de violar a propriedade das árvores binárias de busca na árvore original ajustando-se aquele ponteiro direito do nó mais à direita para a árvore direita. (O mesmo pode ser feito ajustando-se o ponteiro esquerdo do nó mais à esquerda para a subárvore direita da subárvore esquerda). A Figura 6.28 representa esta operação, e a 6.29 mostra a implementação do algoritmo.

Pode parecer que `findAndDeleteByMerging()` contém código redundante. Em vez de chamar `search()` antes de invocar `deleteByMerging()`, `findAndDeleteByMerging()` parece se esquecer de `search()` e pesquisa pelo nó a ser removido usando seu código privativo. Mas usar `search()` na função `findAndDeleteByMerging()` é uma tremenda simplificação. A função

2. O nó tem um filho. Este caso não é complicado. O ponteiro do ascendente para o nó é reajustado para apontar para o filho do nó. Deste modo, os filhos do nó são elevados em um nível e todos os antecessores perdem um grau de descendência em suas designações de parentesco. Por exemplo, o nó que contém 20 (veja Figura 6.27) é removido ajustando o ponteiro direito do seu ascendente, que contém 15, para apontar para o único filho de 20, que é 16.
3. O nó tem dois filhos. Neste caso, nenhuma operação de uma etapa pode ser realizada, pois os ponteiros direito e esquerdo do ascendente não podem apontar para ambos os filhos do nó ao mesmo tempo. Esta seção discute duas soluções diferentes para este problema.

FIGURA 6.26 Removendo uma folha.

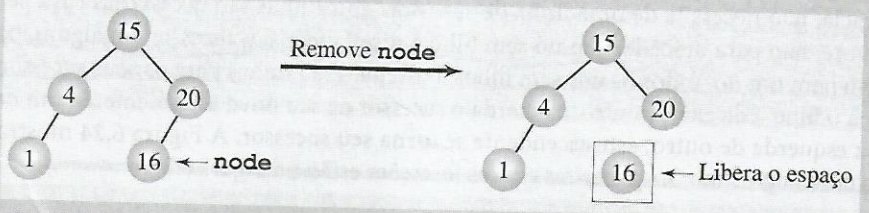
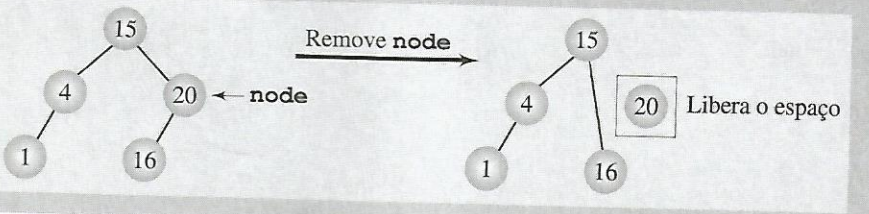


FIGURA 6.27 Removendo um nó com um filho.



6.6.1 Remoção por fusão

Esta solução extrai uma árvore das duas subárvores do nó e a anexa ao ascendente do nó. A técnica é chamada *remoção por fusão*. Mas como podemos fundir essas subárvores? Pela natureza das árvores binárias de busca, cada valor da subárvore direita é maior do que cada valor da subárvore esquerda; assim, a melhor coisa a fazer é encontrar na subárvore esquerda o nó com o maior valor e torná-lo um ascendente da subárvore direita. Simetricamente, pode-se encontrar na subárvore direita o nó com o menor valor e torná-lo um ascendente da subárvore esquerda.

O nó desejado é aquele mais à direita da subárvore esquerda. Ele pode ser localizado movendo-se ao longo da subárvore e tomando os ponteiros direitos até que nulo seja encontrado. Isto significa que este nó não terá um filho direito, e não há perigo de violar a propriedade das árvores binárias de busca na árvore original ajustando-se aquele ponteiro direito do nó mais à direita para a árvore direita. (O mesmo pode ser feito ajustando-se o ponteiro esquerdo do nó mais à esquerda para a subárvore direita da subárvore esquerda). A Figura 6.28 representa esta operação, e a 6.29 mostra a implementação do algoritmo.

Pode parecer que `findAndDeleteByMerging()` contém código redundante. Em vez de chamar `search()` antes de invocar `deleteByMerging()`, `findAndDeleteByMerging()` parece se esquecer de `search()` e pesquisa pelo nó a ser removido usando seu código privativo. Mas usar `search()` na função `findAndDeleteByMerging()` é uma tremenda simplificação. A função

FIGURA 6.29 Implementação do algoritmo para remoção por fusão. (*continuação*)

```
template<class T>
void BST<T>::findAndDeleteByMerging(const T& el) {
    BSTNode<T> *node = root, *prev = 0;
    while (node != 0) {
        if (node->el == el)
            break;
        prev = node;
        if (el < node->el)
            node = node->left;
        else node = node->right;
    }
    if (node != 0 && node->el == el)
        if (node == root)
            deleteByMerging(root);
        else if (prev->left == node)
            deleteByMerging(prev->left);
        else deleteByMerging(prev->right);
    else if (root != 0)
        cout << "element " << el << " nao esta na arvore\n";
    else cout << "a arvore esta vazia\n";
}
```

A Figura 6.30 mostra cada etapa desta operação, evidenciando que mudanças são feitas quando `findAndDeleteByMerging()` é executada. Os números nesta figura correspondem àqueles colocados nos comentários do código da Figura 6.29.

FIGURA 6.30 Detalhes da remoção por fusão.

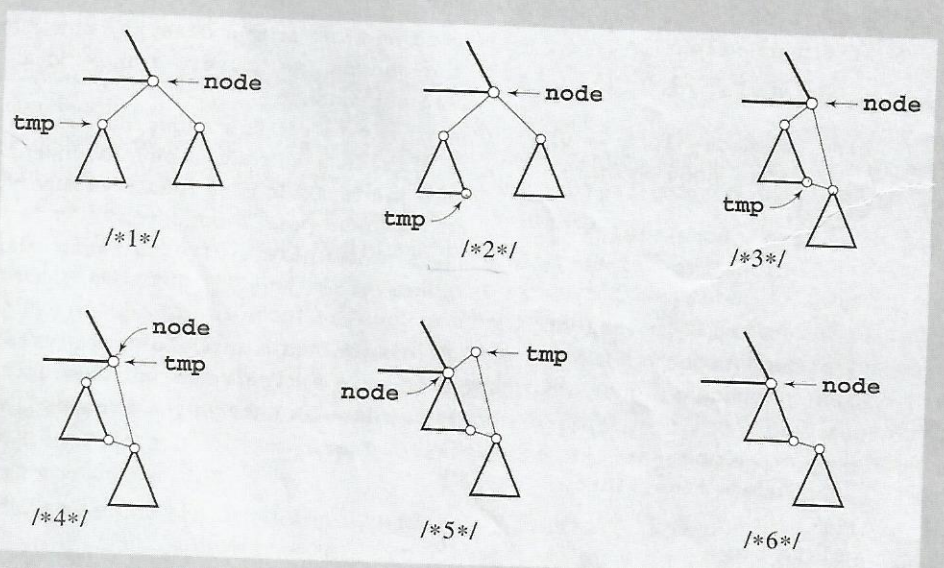
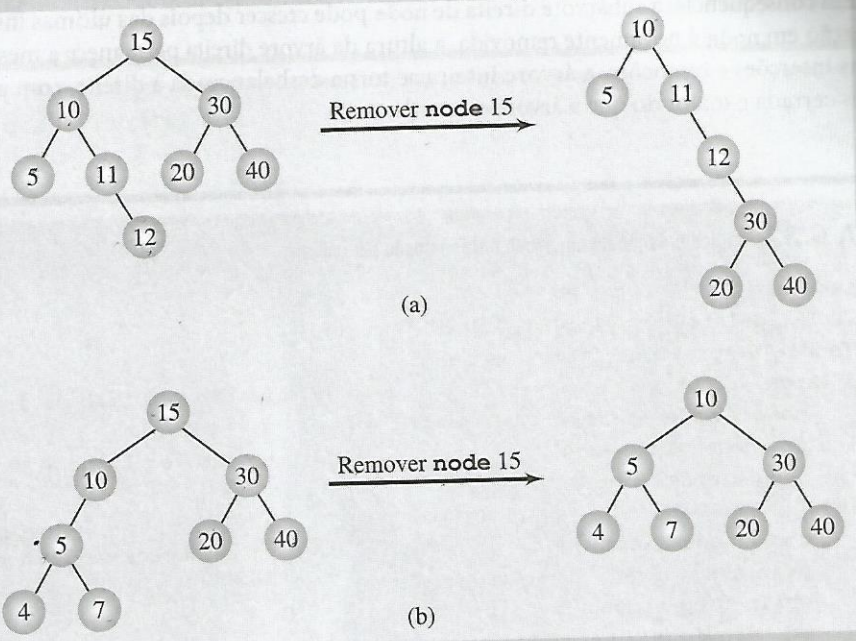


FIGURA 6.31 A altura de uma árvore pode ser (a) estendida e (b) reduzida depois da remoção por fusão.



O algoritmo para a remoção por fusão pode resultar no aumento da altura da árvore. Em alguns casos, a nova árvore pode ser altamente desbalanceada, como a Figura 6.31a ilustra. Algumas vezes, a altura pode ser reduzida (veja a Figura 6.31b). Este algoritmo não é necessariamente ineficiente, mas está longe da perfeição. Seria preferível um algoritmo que não permitisse à árvore aumentar sua altura ao remover um de seus nós.

6.6.2 Remoção por cópia

Outra solução, chamada *remoção por cópia*, foi proposta por Thomas Hibbard e Donald Knuth. Se o nó tem dois filhos, ele pode ser reduzido a um dos dois casos simples: o nó é uma folha ou tem somente um filho não vazio. Isto pode ser feito substituindo por seu predecessor imediato (ou sucessor) a chave que está sendo removida. Como já indicado no algoritmo remoção por fusão, um predecessor de chave é a chave do nó mais à direita na subárvore esquerda (e, analogamente, seu sucessor imediato é a chave do nó mais à esquerda na subárvore direita). Primeiro, o predecessor tem que ser localizado. Isto é feito, novamente, movendo-se uma etapa para a esquerda, primeiro atingindo a raiz da subárvore esquerda do nó e então movendo-se tão à direita quanto possível. A seguir, a chave do nó localizado substitui a chave a ser removida. Este é onde um dos dois casos simples entra em ação. Se o nó mais à direita é uma folha, o primeiro caso se aplica; no entanto, se ele tem um filho, o segundo caso é relevante. Deste modo, a remoção por cópia remove uma chave k_1 sobrescrevendo-a por outra chave k_2 e então removendo o nó que contém k_2, enquanto a remoção por fusão consistiu na remoção de uma chave k_1 junto com o nó que a continha.

Para implementar o algoritmo, duas funções podem ser usadas. Uma, `deleteByCopying()`, está ilustrada na Figura 6.32. A segunda, `findAndDeleteByCopying()`, é como `findAndDeleteByMerging()`, mas chama `deleteByCopying()`, em vez de `deleteByMerging()`. Um acompanhamento etapa por etapa é mostrado na Figura 6.33, e os números sobre os diagramas referem-se aos indicados nos comentários incluídos na implementação de `deleteByCopying()`.

Este algoritmo não aumenta a altura da árvore, mas ainda causa um problema se é aplicado muitas vezes junto com a inserção. Ele é assimétrico; sempre remove o nó do predecessor imediato em node, possivelmente reduzindo a altura da subárvore esquerda e deixando a subárvore direita não afetada. Em consequência, a subárvore direita de node pode crescer depois das últimas inserções e, se a informação em node é novamente removida, a altura da árvore direita permanece a mesma. Depois de muitas inserções e remoções, a árvore inteira se torna desbalanceada à direita, com a árvore direita mais cerrada e maior do que a árvore esquerda.

FIGURA 6.32 Implementação do algoritmo para remoção por cópia.

```
template<class T>
void BST<T>::deleteByCopying(BSTNode<T>*& node) {
    BSTNode<T> *previous, *tmp = node;
    if (node->right == 0)                    // o no nao tem filho a direita;
        node = node->left;
    else if (node->left == 0)                // o no nao tem filho a esquerda;
        node = node->right;
    else {
        tmp = node->left;                    // o no tem ambos os filhos;
        previous = node;                     // 1.
        while (tmp->right != 0) {            // 2.
            previous = tmp;
            tmp = tmp->right;
        }
        node->el = tmp->el;                  // 3.
        if (previous == node)
            previous->left= tmp->left;
        else previous ->right = tmp->left;   // 4.
    }
    delete tmp;                              // 5.
}
```

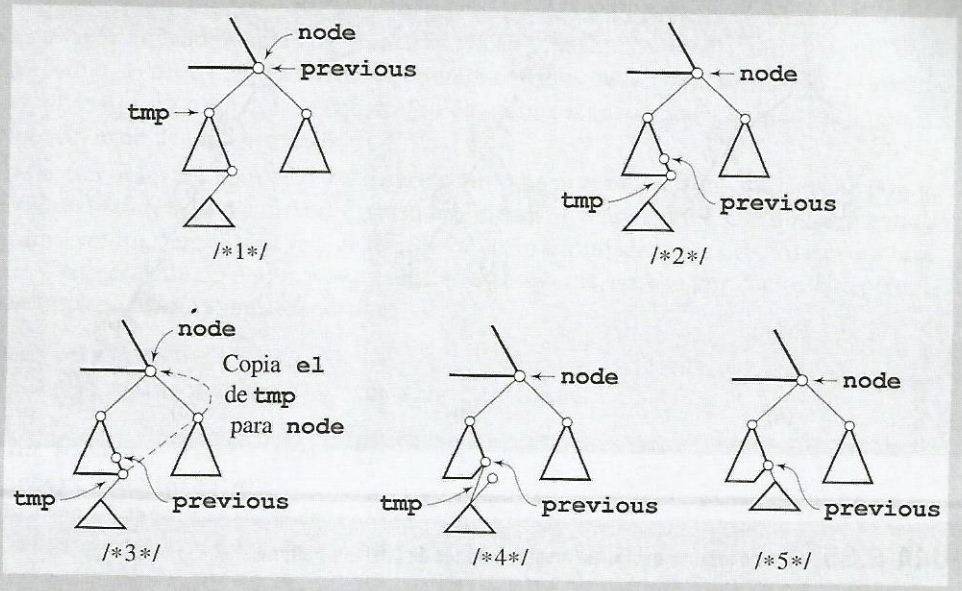
Para evitar este problema, uma melhoria simples pode tornar o algoritmo simétrico. O algoritmo pode, alternativamente, remover o predecessor da informação em node da subárvore à esquerda e seu sucessor da subárvore à direita. A melhoria é significativa. As simulações realizadas por Jeffrey Eppinger mostram que um comprimento de caminho interno esperado para muitas inserções e remoções assimétricas é $\Theta(n \lg^3 n)$ para n nós, e, quando as remoções simétricas são usadas, o comprimento do caminho interno esperado se torna $\Theta(n \lg n)$. Os resultados teóricos obtidos por J. Culberson confirmam essas conclusões. De acordo com Culberson, as inserções e as remoções simétricas dão $\Theta(n\sqrt{n})$ para o comprimento de caminho interno esperado e $\Theta(\sqrt{n})$ para o tempo médio de busca (comprimento médio do caminho), enquanto as remoções simétricas levam a $\Theta(\lg n)$ para o tempo médio de busca, e, como antes, $\Theta(n \lg n)$ para o comprimento de caminho interno médio.

Esses resultados podem ser de importância moderada para as aplicações práticas. Os experimentos mostram que, para uma árvore binária de 2.048 nós, somente depois de 1,5 milhões de inserções e remoções simétricas o comprimento do caminho interno se torna pior do que em uma árvore gerada aleatoriamente.

Os resultados teóricos são apenas fragmentários por causa da extraordinária complexidade do problema. Arne Jonassen e Donald Knuth analisaram o problema das inserções e remoções aleatórias para uma árvore de somente três nós, que exigiram o uso das funções de Bessel e equações integrais

bivariantes, e o resultado da análise classificou-se entre "as mais difíceis de todas as análises exatas de algoritmos que foram realizadas até a data". Em consequência, a confiança nos resultados experimentais não é surpreendente.

FIGURA 6.33 Remoção por cópia.



6.7 Balanceando uma árvore

No início deste capítulo, dois argumentos favoráveis às árvores foram apresentados; ambos são bem apropriados para representar a estrutura hierárquica de certo domínio, e o processo de busca é muito mais rápido usando árvores do que listas ligadas. O segundo argumento, no entanto, nem sempre se mantém. Tudo depende de como é a árvore. A Figura 6.34 mostra três árvores binárias de busca. Todas armazenam os mesmos dados, mas, obviamente, a da Figura 6.34a é a melhor, e a da 6.34c a pior. No pior caso, três testes são necessários na primeira, e seis na última para localizar um objeto. O problema com as árvores nas Figuras 6.34b e c é que são algo assimétricas ou aparadas de um lado, isto é, os objetos nelas não são distribuídos uniformemente na medida em que a árvore na Figura 6.34c praticamente se torna uma lista ligada, embora, formalmente, seja ainda uma árvore. Tal situação não aparece em uma árvore balanceada.

Uma árvore binária é *balanceada em altura* ou simplesmente *balanceada* se a diferença na altura de ambas as subárvores de qualquer nó na árvore é zero ou um. Por exemplo, para o nó K na Figura 6.34b, a diferença entre as alturas de suas subárvores ser igual a um é aceitável. Para o nó B, no entanto, esta diferença é três, o que significa que a árvore inteira é desbalanceada. Para o mesmo nó B na 6.34c, a diferença é a pior possível: cinco. Além disso, uma árvore é considerada *perfeitamente balanceada* se é balanceada e todas as folhas se encontram em um ou em dois níveis.

A Figura 6.35 mostra como muitos nós podem ser armazenados em árvores binárias de diferentes alturas. Uma vez que cada nó pode ter dois filhos, o número de nós em certo nível é o dobro do de ascendentes que residem no nível prévio (exceto, naturalmente, a raiz). Por exemplo, se 10.000 elementos são armazenados em uma árvore perfeitamente balanceada, então a árvore é de altura $\lceil \lg(10.001) \rceil = \lceil 13{,}289 \rceil = 14$. Em termos práticos, isto significa que, se 10.000 elementos são arma-