`search()` retorna um ponteiro para o nó que contém `el`. Em `findAndDeleteByMerging()`, é importante ter este ponteiro armazenado especificamente em um dos ponteiros do ascendente do nó. Em outras palavras, um chamador para `search()` fica satisfeito se pode acessar o nó a partir de qualquer direção, enquanto `findAndDeleteByMerging()` quer acessá-lo tanto do seu membro de dados do ponteiro direito como esquerdo do ascendente. Caso contrário, o acesso à subárvore inteira que tem este nó como sua raiz seria perdido. Uma razão para isto é que `search()` focaliza a chave do nó, e `findAndDeleteByMerging()` no próprio nó como um elemento de uma estrutura maior, ou seja, uma árvore.

FIGURA 6.28 Sumário da remoção por fusão.

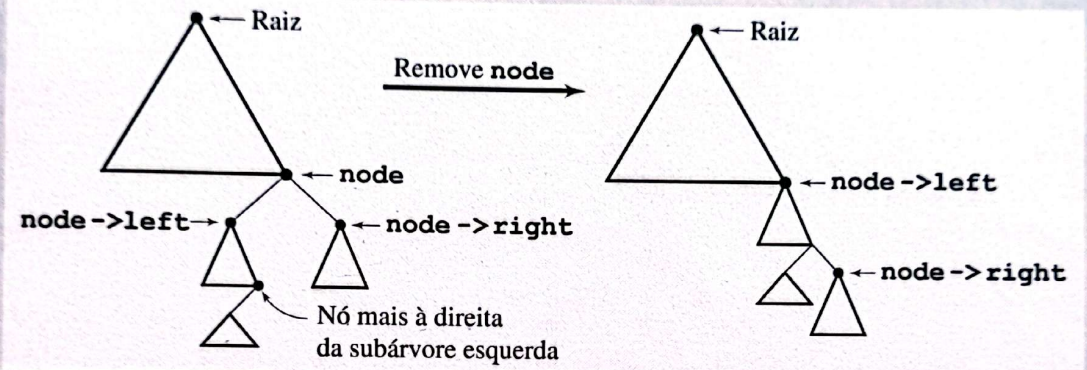

FIGURA 6.29 Implementação do algoritmo para remoção por fusão.

```
template<class T>
void BST<T>::deleteByMerging(BSTNode<T>*& node) {
    BSTNode<T> *tmp = node;
    if (node != 0) {
        if (!node->right)            // o no nao tem filho a direita; seu filho
            node = node->left;       // a esquerda (se houver) e anexado a
                                     // seu ascendente;
        else if (node->left == 0)    // o no nao tem filho a esquerda; seu filho
            node = node->right;      // a direita e anexado a seu ascendente;
        else {                       // esteja preparado para fundir as subarvores;
            tmp = node->left;        // 1. mova-se para a esquerda
            while (tmp->right != 0) // 2. e entao para a direita tanto quanto
                                     // possivel;
                tmp = tmp->right;
            tmp->right =             // 3. estabeleca o vinculo entre
                node->right;         // o no mais a direita da subarvore
                                     //esquerda e da subarvore direita;
            tmp = node;              // 4.
            node = node->left;       // 5.
        }
        delete tmp;                  // 6.
    }
}
```