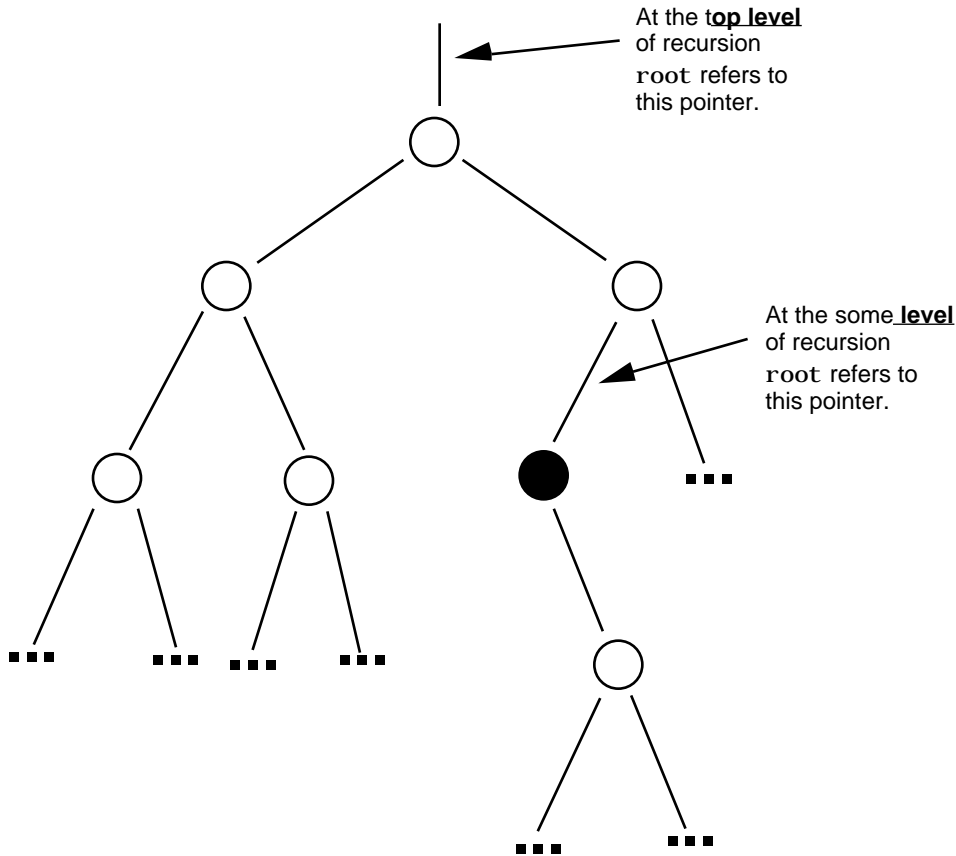

Removing an item from a Binary Search Tree

The Remove function is more involved because it requires rearranging the nodes in the tree. First we find the node we want to remove. There are two possibilities: either the node has no more than one child, or the node has both children.

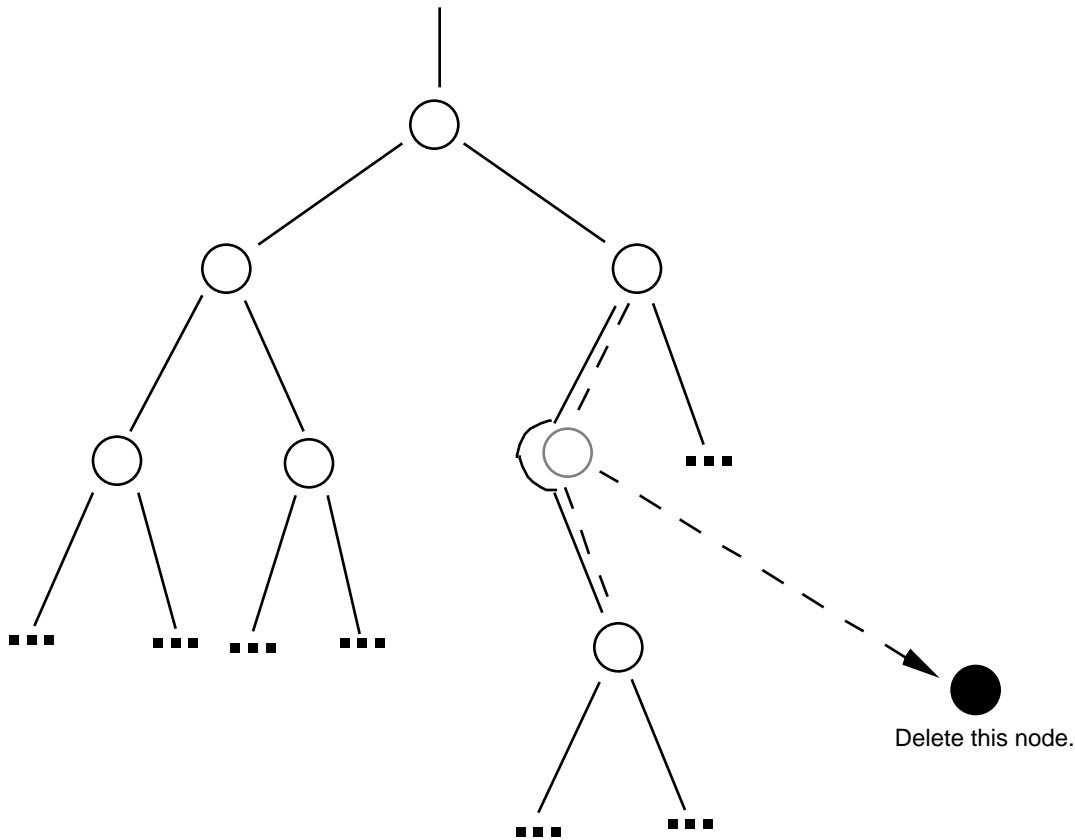
In the first case, we can rearrange the pointers to bypass the node by appending its only child directly to the removed node's parent.

Find the node to delete:



The display on the following page shows how the pointers are rearrange to delete the node.

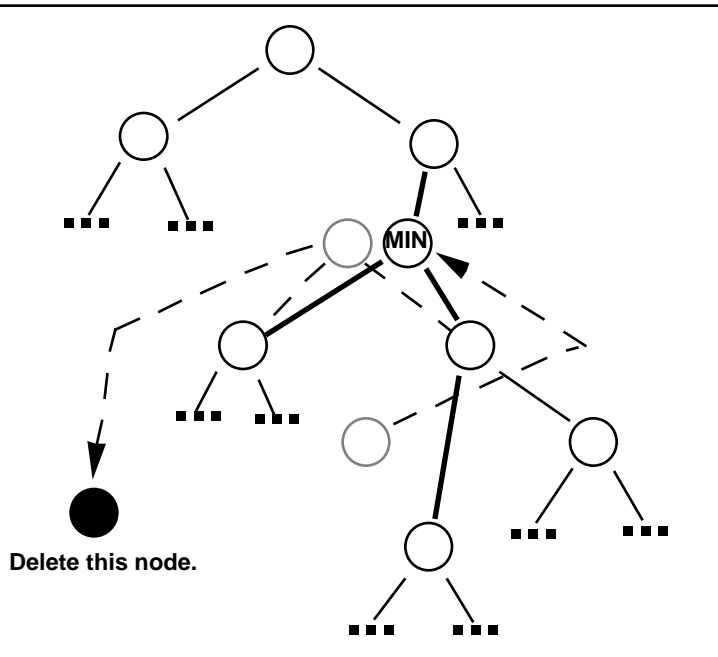
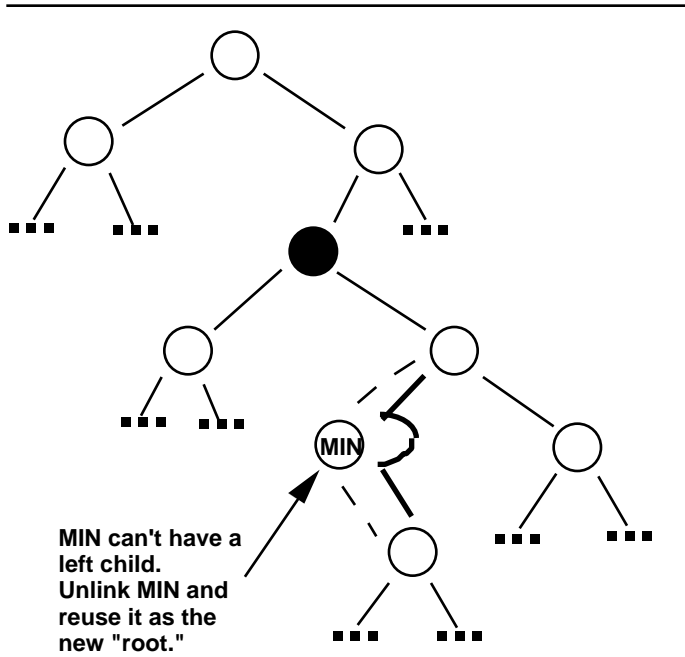
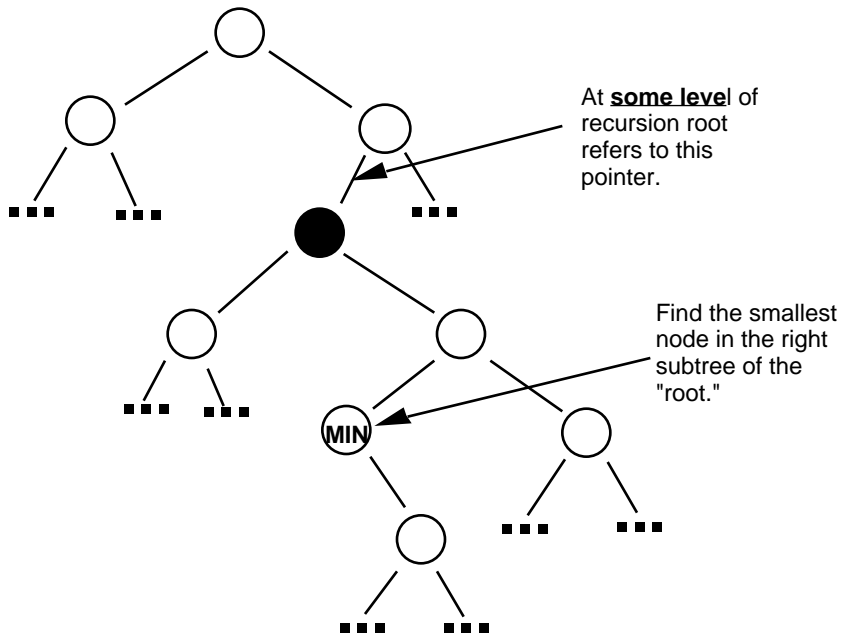
Rearrange pointers and delete the node:



The second case – when both the node’s children are present – requires some extra work. Our approach here is to replace the node with the smallest element of its right subtree (the same would work for the largest element of its left subtree.) The order property of the binary search tree will still hold because the new node, coming from the right subtree, is larger than any node in the left subtree, and since it is the smallest node in the right subtree, it is smaller than all remaining nodes there.

The algorithm is to find the node containing the smallest element of any tree by starting at the root and going left as long as possible. This node does not have a left child, so we can easily unlink it from the tree. We can put this node in place of the removed node to fill the gap.

On the following page are the steps to find and remove a node with two subtrees attached.



This method is implemented in the following recursive code:

```
bool Remove(TNode * & root, const KeyType & key)
// pre: T is a binary search tree
// post: The key node is removed from the tree if it is found, and true is returned
//       If key is not found then the tree is not altered, and false is returned
{
    if (root == NULL) // Base case: the tree is empty
        return false;

    if (root->key == key) // Base case: found the node, root refers to the
    { // pointer to the node to be removed.
        TNode * oldroot(root); // Save the node to be removed
        if (root->left == NULL) // No left child
            root = root->right; // Replace root pointer with pointer to child
        else if (root->right == NULL) // No right child
            root = root->left; // Replace root pointer with pointer to child
        if (oldroot != root) // One of the two if's above was true
        {
            delete oldroot;
            return true;
        }
        TNode * parent(root);
        TNode * next(root->right);
        while (next->left != NULL) // Find the smallest element
        { // in the right subtree
            parent = next; // and keep track of its parent
            next = next->left;
        }
        if (parent == root) // Unlink this node from the tree
            root->right = next->right; // it doesn't have a left child
        else // so just connect its right subtree
            parent->left = next->right; // to its parent
        next->left = root->left; // Unlink this smallest element and
        next->right = root->right; // make it the new root
        delete root;
        root = next;
        return true;
    }
    else if (key < root->key) // Recursive case: Remove
        return Remove(root->left, key); // from the left subtree
    else if (key > root->key) // Recursive case: Remove
        return Remove(root->right, key); // from the right subtree
}
```