Figure 12.3   Inserting a node with key 13 into a binary search tree. The simple path from the root down to the position where the node is inserted is shown in blue. The new node and the link to its parent are highlighted in orange.

trailing pointer y, because by the time it finds the NIL where z belongs, the search has proceeded one step beyond the node that needs to be changed. Lines 8–13 set the pointers that cause z to be inserted.

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

Deletion

The overall strategy for deleting a node z from a binary search tree T has three basic cases and, as we'll see, one of the cases is a bit tricky.

- If z has no children, then simply remove it by modifying its parent to replace z with NIL as its child.

- If z has just one child, then elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.

- If z has two children, find z's successor y—which must belong to z's right subtree—and move y to take z's position in the tree. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree. Because y is z's successor, it cannot have a left child, and y's original right child moves into y's original position, with the rest of y's original right subtree following automatically. This case is the tricky one because, as we'll see, it matters whether y is z's right child.

The procedure for deleting a given node z from a binary search tree T takes as arguments pointers to T and z. It organizes its cases a bit differently from the three cases outlined previously by considering the four cases shown in Figure 12.4.

- If z has no left child, then as in part (a) of the figure, replace z by its right child, which may or may not be NIL. When z's right child is NIL, this case deals with
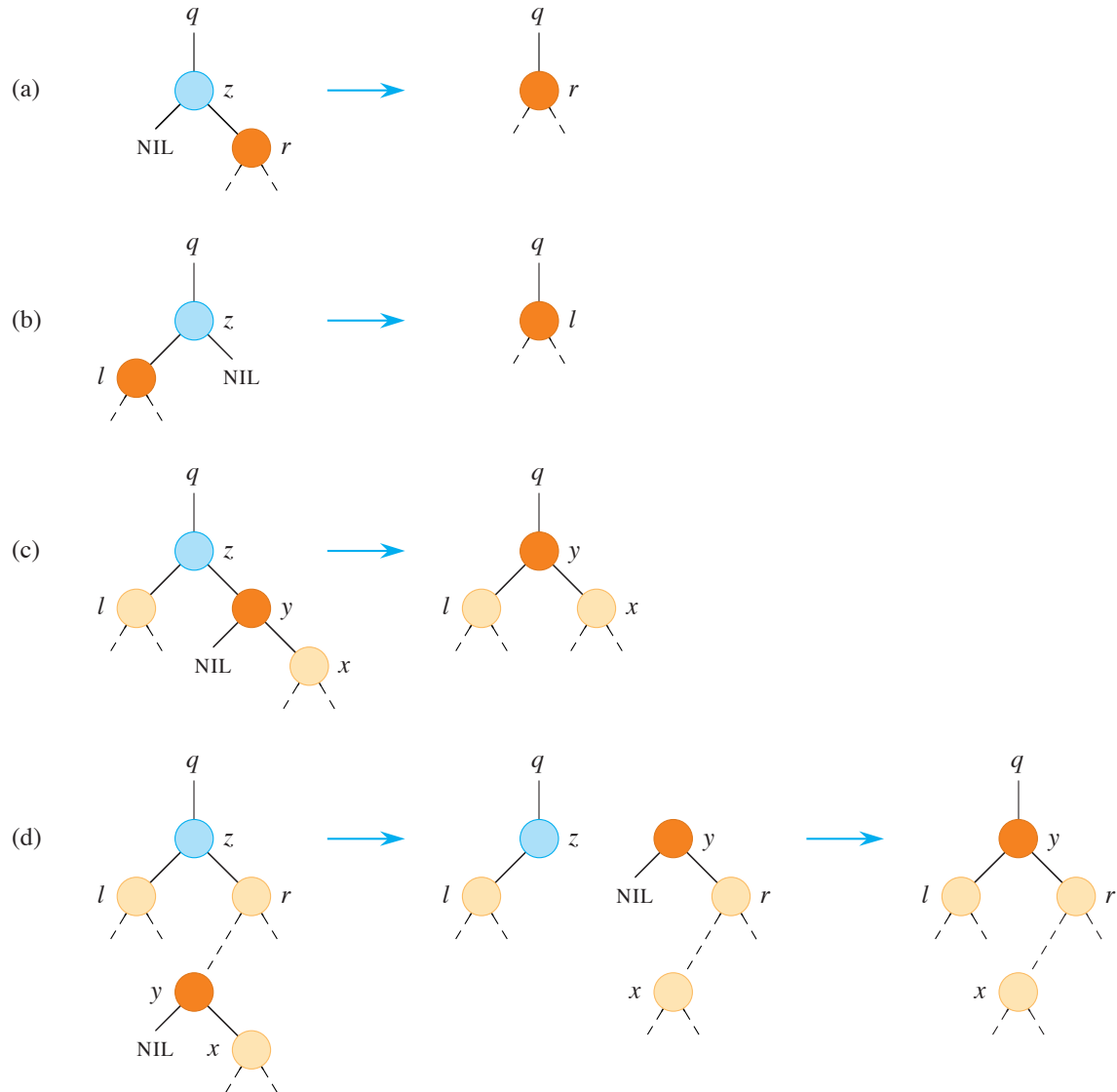
Figure 12.4   Deleting a node z, in blue, from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. The node that will replace node z in its position in the tree is colored orange. **(a)** Node z has no left child. Replace z by its right child r, which may or may not be NIL. **(b)** Node z has a left child l but no right child. Replace z by l. **(c)** Node z has two children. Its left child is node l, its right child is its successor y (which has no left child), and y's right child is node x. Replace z by y, updating y's left child to become l, but leaving x as y's right child. **(d)** Node z has two children (left child l and right child r), and its successor $y \neq r$ lies within the subtree rooted at r. First replace y by its own right child x, and set y to be r's parent. Then set y to be q's child and the parent of l.

the situation in which z has no children. When z's right child is non-NIL, this case handles the situation in which z has just one child, which is its right child.

- Otherwise, if z has just one child, then that child is a left child. As in part (b) of the figure, replace z by its left child.

- Otherwise, z has both a left and a right child. Find z's successor y, which lies in z's right subtree and has no left child (see Exercise 12.2-5). Splice node y out of its current location and replace z by y in the tree. How to do so depends on whether y is z's right child:

  - If y is z's right child, then as in part (c) of the figure, replace z by y, leaving y's right child alone.
  - Otherwise, y lies within z's right subtree but is not z's right child. In this case, as in part (d) of the figure, first replace y by its own right child, and then replace z by y.

As part of the process of deleting a node, subtrees need to move around within the binary search tree. The subroutine TRANSPLANT replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child. TRANSPLANT allows v to be NIL instead of a pointer to a node.

TRANSPLANT(T, u, v)

1   **if** $u.p ==$ NIL
2       $T.root = v$
3   **elseif** $u == u.p.left$
4       $u.p.left = v$
5   **else** $u.p.right = v$
6   **if** $v \neq$ NIL
7       $v.p = u.p$

Here is how TRANSPLANT works. Lines 1–2 handle the case in which u is the root of T. Otherwise, u is either a left child or a right child of its parent. Lines 3–4 take care of updating $u.p.left$ if u is a left child, and line 5 updates $u.p.right$ if u is a right child. Because v may be NIL, lines 6–7 update $v.p$ only if v is non-NIL. The procedure TRANSPLANT does not attempt to update $v.left$ and $v.right$. Doing so, or not doing so, is the responsibility of TRANSPLANT's caller.

The procedure TREE-DELETE on the facing page uses TRANSPLANT to delete node z from binary search tree T. It executes the four cases as follows. Lines 1–2 handle the case in which node z has no left child (Figure 12.4(a)), and lines 3–4

handle the case in which z has a left child but no right child (Figure 12.4(b)). Lines 5–12 deal with the remaining two cases, in which z has two children. Line 5 finds node y, which is the successor of z. Because z has a nonempty right subtree, its successor must be the node in that subtree with the smallest key; hence the call to TREE-MINIMUM($z.right$). As we noted before, y has no left child. The procedure needs to splice y out of its current location and replace z by y in the tree. If y is z's right child (Figure 12.4(c)), then lines 10–12 replace z as a child of its parent by y and replace y's left child by z's left child. Node y retains its right child (x in Figure 12.4(c)), and so no change to $y.right$ needs to occur. If y is not z's right child (Figure 12.4(d)), then two nodes have to move. Lines 7–9 replace y as a child of its parent by y's right child (x in Figure 12.4(c)) and make z's right child (r in the figure) become y's right child instead. Finally, lines 10–12 replace z as a child of its parent by y and replace y's left child by z's left child.

TREE-DELETE(T, z)

```
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)        // replace z by its right child
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)         // replace z by its left child
 5  else y = TREE-MINIMUM(z.right)       // y is z's successor
 6      if y ≠ z.right                   // is y farther down the tree?
 7          TRANSPLANT(T, y, y.right)    // replace y by its right child
 8          y.right = z.right            // z's right child becomes
 9          y.right.p = y                //      y's right child
10      TRANSPLANT(T, z, y)              // replace z by its successor y
11      y.left = z.left                  // and give z's left child to y,
12      y.left.p = y                     //      which had no left child
```

Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREE-MINIMUM in line 5. Thus, TREE-DELETE runs in $O(h)$ time on a tree of height h.

In summary, we have proved the following theorem.

Theorem 12.3
The dynamic-set operations INSERT and DELETE can be implemented so that each one runs in $O(h)$ time on a binary search tree of height h. ∎