Both of these procedures run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a path downward from the root.

Successor and predecessor

Given a node in a binary search tree, it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than $key[x]$. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2       then return TREE-MINIMUM (right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5       do x ← y
6          y ← p[y]
7  return y
```

The code for TREE-SUCCESSOR is broken into two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in the right subtree, which is found in line 2 by calling TREE-MINIMUM($right[x]$). For example, the successor of the node with key 15 in Figure 12.2 is the node with key 17.

On the other hand, as Exercise 12.2-6 asks you to show, if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. In Figure 12.2, the successor of the node with key 13 is the node with key 15. To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; this is accomplished by lines 3–7 of TREE-SUCCESSOR.

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since we either follow a path up the tree or follow a path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.

Even if keys are not distinct, we define the successor and predecessor of any node x as the node returned by calls made to TREE-SUCCESSOR(x) and TREE-PREDECESSOR(x), respectively.

In summary, we have proved the following theorem.

Theorem 12.2

The dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be made to run in $O(h)$ time on a binary search tree of height h.

Exercises 12.2-1

Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?

a. 2, 252, 401, 398, 330, 344, 397, 363.
b. 924, 220, 911, 244, 898, 258, 362, 363.
c. 925, 202, 911, 240, 912, 245, 363.
d. 2, 399, 387, 219, 266, 382, 381, 278, 363.
e. 935, 278, 347, 621, 299, 392, 358, 363.

Exercises 12.2-2

Write recursive versions of the TREE-MINIMUM and TREE-MAXIMUM procedures.

Exercises 12.2-3

Write the TREE-PREDECESSOR procedure.

Exercises 12.2-4

Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

Exercises 12.2-5

Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

Exercises 12.2-6

Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

Exercises 12.2-7

An inorder tree walk of an n-node binary search tree can be implemented by finding the minimum element in the tree with TREE-MINIMUM and then making n-1 calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

Exercises 12.2-8

Prove that no matter what node we start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

Exercises 12.2-9

Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $key[y]$ is either the smallest key in T larger than $key[x]$ or the largest key in T smaller than $key[x]$.

12.3 Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold. As we shall see, modifying the tree to insert a new element is relatively straight-forward, but handling deletion is somewhat more intricate.

Insertion

To insert a new value v into a binary search tree T, we use the procedure TREE-INSERT. The procedure is passed a node z for which $key[z] = v$, $left[z] =$ NIL, and $right[z] =$ NIL. It modifies T and some of the fields of z in such a way that z is inserted into an appropriate position in the tree.

```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4       do y ←  x
 5          if key[z] < key[x]
```

```
 6            then x ← left[x]
 7            else x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                ▷ Tree T was empty
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
```

Figure 12.3 shows how TREE-INSERT works. Just like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and traces a path downward. The pointer x traces the path, and the pointer y is maintained as the parent of x. After initialization, the **while** loop in lines 3–7 causes these two pointers to move down the tree, going left or right depending on the comparison of $key[z]$ with $key[x]$, until x is set to NIL. This NIL occupies the position where we wish to place the input item z. Lines 8–13 set the pointers that cause z to be inserted.

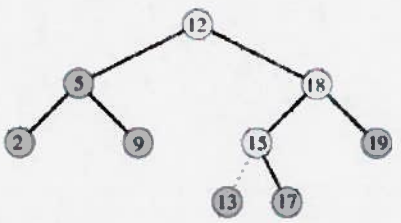

Figure 12.3: Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

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 procedure for deleting a given node z from a binary search tree takes as an argument a pointer to z. The procedure considers the three cases shown in Figure 12.4. If z has no children, we modify its parent $p[z]$ to replace z with NIL as its child. If the node has only a single child, we "splice out" z by making a new link between its child and its parent. Finally, if the node has two children, we splice out z's successor y, which has no left child (see Exercise 12.2-5) and replace z's key and satellite data with y's key and satellite data.

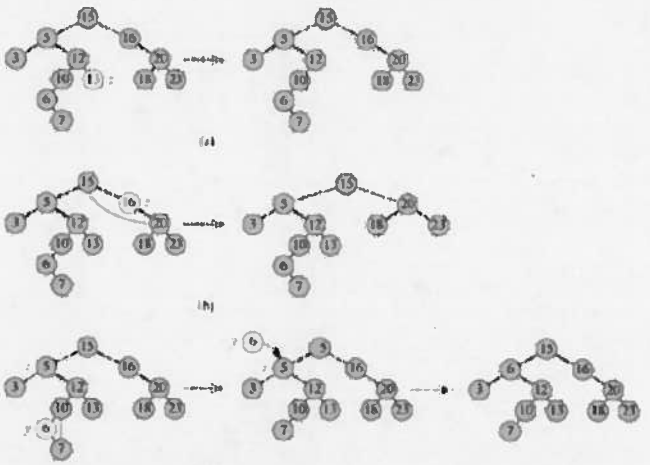

Figure 12.4: Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. *(a)* If z has no children, we just remove it. *(b)* If z has only one child, we splice out z. *(c)* If z has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data.

The code for TREE-DELETE organizes these three cases a little differently.

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2       then y ← z
 3       else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5       then x ← left[y]
 6       else x ← right[y]
 7  if x ≠ NIL
 8       then p[x] ← p[y]
 9  if p[y] = NIL
10       then root[T] ← x
11       else if y = left[p[y]]
12                then left[p[y]] ← x
13                else right[p[y]] ← x
14  if y ≠ z
15       then key[z] ← key[y]
16            copy y's satellite data into z
17  return y
```

In lines 1–3, the algorithm determines a node y to splice out. The node y is either the input node z (if z has at most 1 child) or the successor of z (if z has two children). Then, in lines 4–6, x is set to the non-NIL child of y, or to NIL if y has no children. The node y is spliced out in lines 7–13 by modifying pointers in $p[y]$ and x. Splicing out y is somewhat complicated by the need for proper handling of the boundary conditions, which occur when x = NIL or when y is the root. Finally, in lines 14–16, if the successor of z was the node spliced out, y's key and satellite data are moved to z, overwriting the previous key and satellite data. The node y is returned in line 17 so that the calling procedure can recycle it via the free list. The procedure 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 made to run in $O(h)$ time on a binary search tree of height h.

Exercises 12.3-1

Give a recursive version of the TREE-INSERT procedure.

Exercises 12.3-2

Suppose that a binary search tree is constructed by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree.

Exercises 12.3-3

We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

Exercises 12.3-4

Suppose that another data structure contains a pointer to a node y in a binary search tree, and suppose that y's predecessor z is deleted from the tree by the procedure TREE-DELETE. What problem can arise? How can TREE-DELETE be rewritten to solve this problem?

Exercises 12.3-5

Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.