

---

## 12 Binary Search Trees

The search tree data structure supports each of the dynamic-set operations listed on page 250: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, you can use a search tree both as a dictionary and as a priority queue.

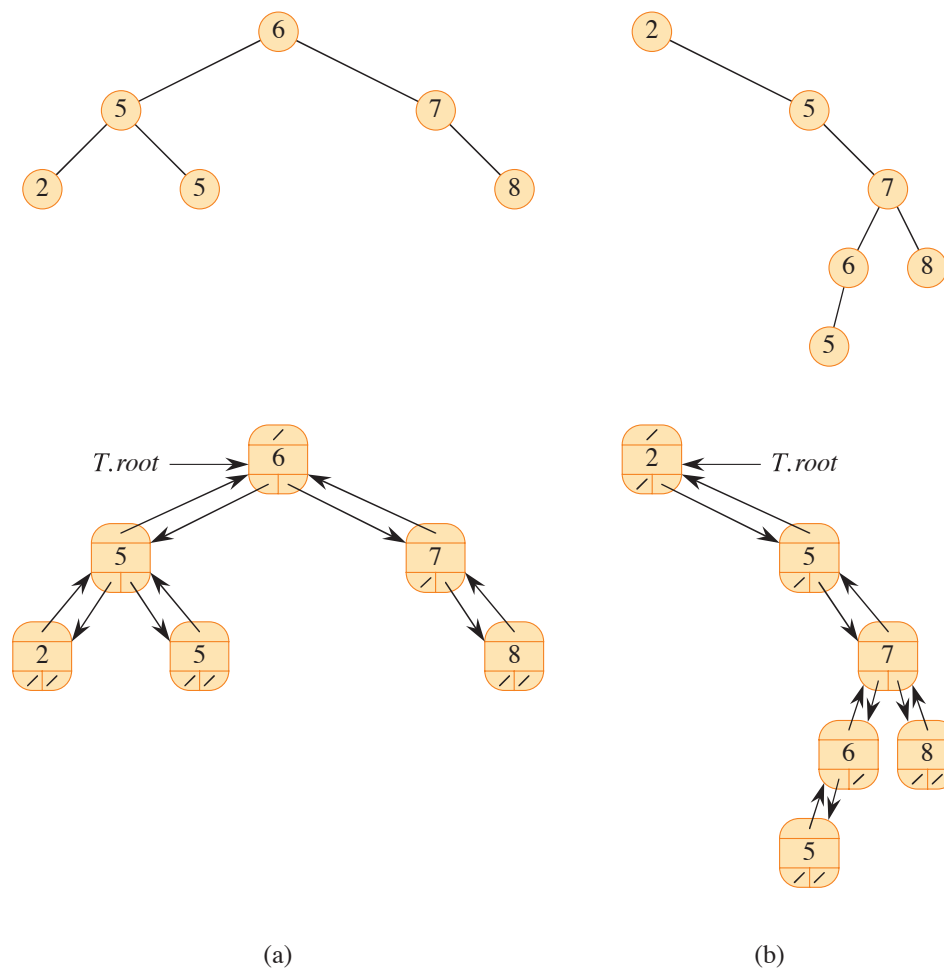
Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with  $n$  nodes, such operations run in  $\Theta(\lg n)$  worst-case time. If the tree is a linear chain of  $n$  nodes, however, the same operations take  $\Theta(n)$  worst-case time. In Chapter 13, we'll see a variation of binary search trees, red-black trees, whose operations guarantee a height of  $O(\lg n)$ . We won't prove it here, but if you build a binary search tree on a random set of  $n$  keys, its expected height is  $O(\lg n)$  even if you don't try to limit its height.

After presenting the basic properties of binary search trees, the following sections show how to walk a binary search tree to print its values in sorted order, how to search for a value in a binary search tree, how to find the minimum or maximum element, how to find the predecessor or successor of an element, and how to insert into or delete from a binary search tree. The basic mathematical properties of trees appear in Appendix B.

---

### 12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 12.1. You can represent such a tree with a linked data structure, as in Section 10.3. In addition to a *key* and satellite data, each node object contains attributes *left*, *right*, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate attribute contains the value NIL. The tree itself has an attribute *root*



**Figure 12.1** Binary search trees. For any node  $x$ , the keys in the left subtree of  $x$  are at most  $x.key$ , and the keys in the right subtree of  $x$  are at least  $x.key$ . Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. The top figure shows how to view the tree conceptually, and the bottom figure shows the *left*, *right*, and *p* attributes in each node, in the style of Figure 10.6 on page 266. **(b)** A less efficient binary search tree, with height 4, that contains the same keys.

that points to the root node, or NIL if the tree is empty. The root node  $T.root$  is the only node in a tree  $T$  whose parent is NIL.

The keys in a binary search tree are always stored in such a way as to satisfy the *binary-search-tree property*:

Let  $x$  be a node in a binary search tree. If  $y$  is a node in the left subtree of  $x$ , then  $y.key \leq x.key$ . If  $y$  is a node in the right subtree of  $x$ , then  $y.key \geq x.key$ .

Thus, in Figure 12.1(a), the key of the root is 6, the keys 2, 5, and 5 in its left subtree are no larger than 6, and the keys 7 and 8 in its right subtree are no smaller than 6. The same property holds for every node in the tree. For example, looking at the root's left child as the root of a subtree, this subtree root has the key 5, the key 2 in its left subtree is no larger than 5, and the key 5 in its right subtree is no smaller than 5.

Because of the binary-search-tree property, you can print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an *inorder tree walk*, given by the procedure INORDER-TREE-WALK. This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree. (Similarly, a *preorder tree walk* prints the root before the values in either subtree, and a *postorder tree walk* prints the root after the values in its subtrees.) To print all the elements in a binary search tree  $T$ , call INORDER-TREE-WALK( $T.root$ ). For example, the inorder tree walk prints the keys in each of the two binary search trees from Figure 12.1 in the order 2, 5, 5, 6, 7, 8. The correctness of the algorithm follows by induction directly from the binary-search-tree property.

```

INORDER-TREE-WALK( $x$ )
1  if  $x \neq \text{NIL}$ 
2      INORDER-TREE-WALK( $x.left$ )
3      print  $x.key$ 
4      INORDER-TREE-WALK( $x.right$ )

```

It takes  $\Theta(n)$  time to walk an  $n$ -node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree—once for its left child and once for its right child. The following theorem gives a formal proof that it takes linear time to perform an inorder tree walk.

**Theorem 12.1**

If  $x$  is the root of an  $n$ -node subtree, then the call INORDER-TREE-WALK( $x$ ) takes  $\Theta(n)$  time.

**Proof** Let  $T(n)$  denote the time taken by INORDER-TREE-WALK when it is called on the root of an  $n$ -node subtree. Since INORDER-TREE-WALK visits all  $n$  nodes of the subtree, we have  $T(n) = \Omega(n)$ . It remains to show that  $T(n) = O(n)$ .

Since INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree (for the test  $x \neq \text{NIL}$ ), we have  $T(0) = c$  for some constant  $c > 0$ .

For  $n > 0$ , suppose that INORDER-TREE-WALK is called on a node  $x$  whose left subtree has  $k$  nodes and whose right subtree has  $n - k - 1$  nodes. The time to perform INORDER-TREE-WALK( $x$ ) is bounded by  $T(n) \leq T(k) + T(n - k - 1) + d$  for some constant  $d > 0$  that reflects an upper bound on the time to execute the body of INORDER-TREE-WALK( $x$ ), exclusive of the time spent in recursive calls.

We use the substitution method to show that  $T(n) = O(n)$  by proving that  $T(n) \leq (c + d)n + c$ . For  $n = 0$ , we have  $(c + d) \cdot 0 + c = c = T(0)$ . For  $n > 0$ , we have

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &\leq ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c, \end{aligned}$$

which completes the proof. ■

## Exercises

### 12.1-1

For the set  $\{1, 4, 5, 10, 16, 17, 21\}$  of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

### 12.1-2

What is the difference between the binary-search-tree property and the min-heap property on page 163? Can the min-heap property be used to print out the keys of an  $n$ -node tree in sorted order in  $O(n)$  time? Show how, or explain why not.

### 12.1-3

Give a nonrecursive algorithm that performs an inorder tree walk. (*Hint:* An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that you can test two pointers for equality.)

### 12.1-4

Give recursive algorithms that perform preorder and postorder tree walks in  $\Theta(n)$  time on a tree of  $n$  nodes.

### 12.1-5

Argue that since sorting  $n$  elements takes  $\Omega(n \lg n)$  time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of  $n$  elements takes  $\Omega(n \lg n)$  time in the worst case.

---

## 12.2 Querying a binary search tree

Binary search trees can support the queries **MINIMUM**, **MAXIMUM**, **SUCCESSOR**, and **PREDECESSOR**, as well as **SEARCH**. This section examines these operations and shows how to support each one in  $O(h)$  time on any binary search tree of height  $h$ .

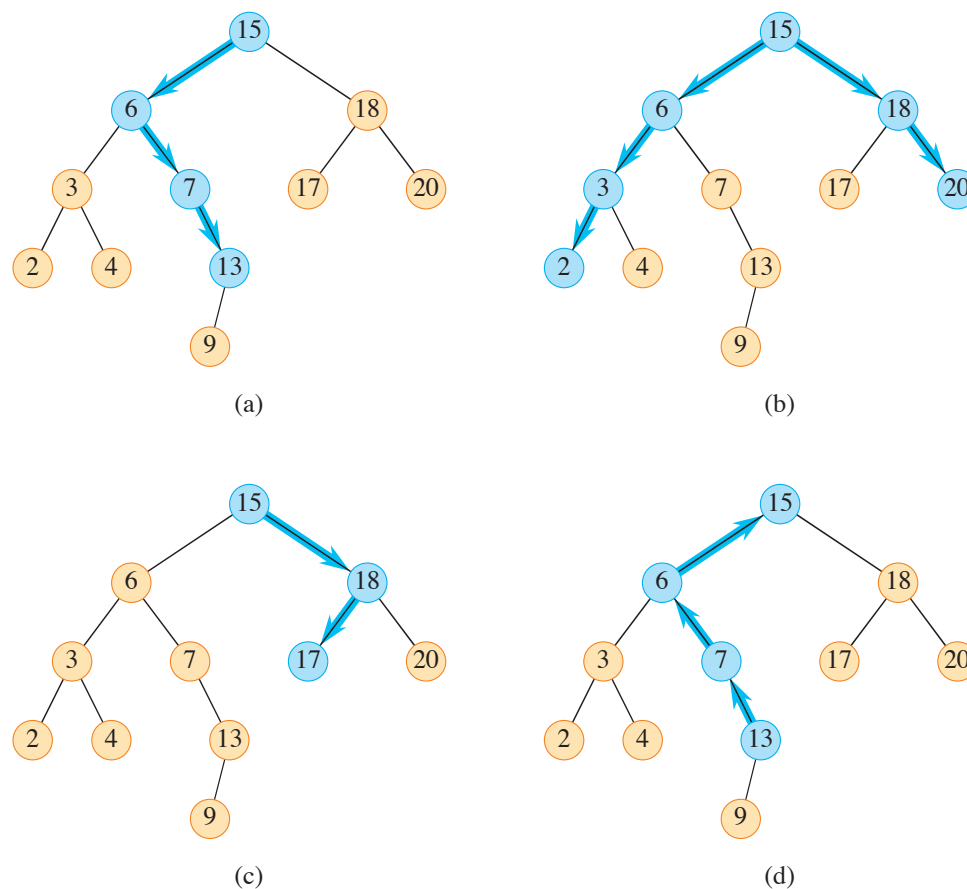
### Searching

To search for a node with a given key in a binary search tree, call the **TREE-SEARCH** procedure. Given a pointer  $x$  to the root of a subtree and a key  $k$ , **TREE-SEARCH**( $x, k$ ) returns a pointer to a node with key  $k$  if one exists in the subtree; otherwise, it returns **NIL**. To search for key  $k$  in the entire binary search tree  $T$ , call **TREE-SEARCH**( $T.root, k$ ).

```
TREE-SEARCH( $x, k$ )
1  if  $x == \text{NIL}$  or  $k == x.key$ 
2      return  $x$ 
3  if  $k < x.key$ 
4      return TREE-SEARCH( $x.left, k$ )
5  else return TREE-SEARCH( $x.right, k$ )
```

```
ITERATIVE-TREE-SEARCH( $x, k$ )
1  while  $x \neq \text{NIL}$  and  $k \neq x.key$ 
2      if  $k < x.key$ 
3           $x = x.left$ 
4      else  $x = x.right$ 
5  return  $x$ 
```

The **TREE-SEARCH** procedure begins its search at the root and traces a simple path downward in the tree, as shown in Figure 12.2(a). For each node  $x$  it encounters, it compares the key  $k$  with  $x.key$ . If the two keys are equal, the search terminates. If  $k$  is smaller than  $x.key$ , the search continues in the left subtree of  $x$ , since the binary-search-tree property implies that  $k$  cannot reside in the right subtree. Symmetrically, if  $k$  is larger than  $x.key$ , the search continues in the right subtree. The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of **TREE-SEARCH** is  $O(h)$ , where  $h$  is the height of the tree.



**Figure 12.2** Queries on a binary search tree. Nodes and paths followed in each query are colored blue. (a) A search for the key 13 in the tree follows the path  $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$  from the root. (b) The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. (c) The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. (d) The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

Since the TREE-SEARCH procedure recurses on either the left subtree or the right subtree, but not both, we can rewrite the algorithm to “unroll” the recursion into a **while** loop. On most computers, the ITERATIVE-TREE-SEARCH procedure on the facing page is more efficient.

### Minimum and maximum

To find an element in a binary search tree whose key is a minimum, just follow *left* child pointers from the root until you encounter a NIL, as shown in Figure 12.2(b).

The TREE-MINIMUM procedure returns a pointer to the minimum element in the subtree rooted at a given node  $x$ , which we assume to be non-NIL.

```
TREE-MINIMUM( $x$ )
1  while  $x.left \neq \text{NIL}$ 
2       $x = x.left$ 
3  return  $x$ 

TREE-MAXIMUM( $x$ )
1  while  $x.right \neq \text{NIL}$ 
2       $x = x.right$ 
3  return  $x$ 
```

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If node  $x$  has no left subtree, then since every key in the right subtree of  $x$  is at least as large as  $x.key$ , the minimum key in the subtree rooted at  $x$  is  $x.key$ . If node  $x$  has a left subtree, then since no key in the right subtree is smaller than  $x.key$  and every key in the left subtree is not larger than  $x.key$ , the minimum key in the subtree rooted at  $x$  resides in the subtree rooted at  $x.left$ .

The pseudocode for TREE-MAXIMUM is symmetric. Both TREE-MINIMUM and TREE-MAXIMUM run in  $O(h)$  time on a tree of height  $h$  since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

### Successor and predecessor

Given a node in a binary search tree, how can you 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  $x.key$ . Regardless of whether the keys are distinct, we define the *successor* of a node as the next node visited in an inorder tree walk. The structure of a binary search tree allows you to determine the successor of a node without comparing keys. The TREE-SUCCESSOR procedure on the facing page returns the successor of a node  $x$  in a binary search tree if it exists, or NIL if  $x$  is the last node that would be visited during an inorder walk.

The code for TREE-SUCCESSOR has two cases. If the right subtree of node  $x$  is nonempty, then the successor of  $x$  is just the leftmost node in  $x$ 's right subtree, which line 2 finds by calling TREE-MINIMUM( $x.right$ ). For example, the successor of the node with key 15 in Figure 12.2(c) 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

```

TREE-SUCCESSOR( $x$ )
1  if  $x.right \neq \text{NIL}$ 
2      return TREE-MINIMUM( $x.right$ ) // leftmost node in right subtree
3  else // find the lowest ancestor of  $x$  whose left child is an ancestor of  $x$ 
4       $y = x.p$ 
5      while  $y \neq \text{NIL}$  and  $x == y.right$ 
6           $x = y$ 
7           $y = y.p$ 
8      return  $y$ 

```

left child is also an ancestor of  $x$ . In Figure 12.2(d), the successor of the node with key 13 is the node with key 15. To find  $y$ , go up the tree from  $x$  until you encounter either the root or a node that is the left child of its parent. Lines 4–8 of TREE-SUCCESSOR handle this case.

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

In summary, we have proved the following theorem.

### Theorem 12.2

The dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be implemented so that each one runs in  $O(h)$  time on a binary search tree of height  $h$ . ■

## Exercises

### 12.2-1

You are searching for the number 363 in a binary search tree containing numbers between 1 and 1000. Which of the following sequences *cannot* 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.



**12.2-2**

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

**12.2-3**

Write the TREE-PREDECESSOR procedure.

**12.2-4**

Professor Kilmer claims to have discovered a remarkable property of binary search trees. Suppose that the search for key  $k$  in a binary search tree ends up at 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 Kilmer 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.

**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.

**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.)

**12.2-7**

An alternative method of performing an inorder tree walk of an  $n$ -node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making  $n - 1$  calls to TREE-SUCCESSOR. Prove that this algorithm runs in  $\Theta(n)$  time.

**12.2-8**

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

**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  $y.key$  is either the smallest key in  $T$  larger than  $x.key$  or the largest key in  $T$  smaller than  $x.key$ .

## 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. We'll see that modifying the tree to insert a new element is relatively straightforward, but deleting a node from a binary search tree is more complicated.

### Insertion

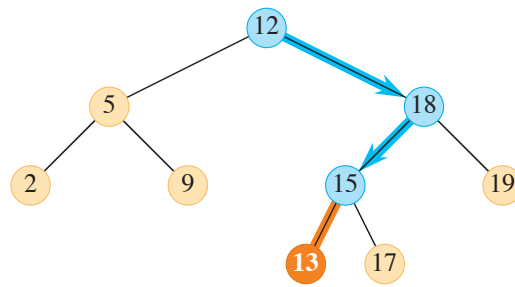
The TREE-INSERT procedure inserts a new node into a binary search tree. The procedure takes a binary search tree  $T$  and a node  $z$  for which  $z.key$  has already been filled in,  $z.left = \text{NIL}$ , and  $z.right = \text{NIL}$ . It modifies  $T$  and some of the attributes of  $z$  so as to insert  $z$  into an appropriate position in the tree.

```

TREE-INSERT( $T, z$ )
1   $x = T.root$            // node being compared with  $z$ 
2   $y = \text{NIL}$            //  $y$  will be parent of  $z$ 
3  while  $x \neq \text{NIL}$      // descend until reaching a leaf
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$              // found the location—insert  $z$  with parent  $y$ 
9  if  $y == \text{NIL}$ 
10      $T.root = z$        // tree  $T$  was empty
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 

```

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 the pointer  $x$  traces a simple path downward looking for a NIL to replace with the input node  $z$ . The procedure maintains the *trailing pointer*  $y$  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  $z.key$  with  $x.key$ , until  $x$  becomes NIL. This NIL occupies the position where node  $z$  will go. More precisely, this NIL is a *left* or *right* attribute of the node that will become  $z$ 's parent, or it is  $T.root$  if tree  $T$  is currently empty. The procedure needs the



**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  $h$ .

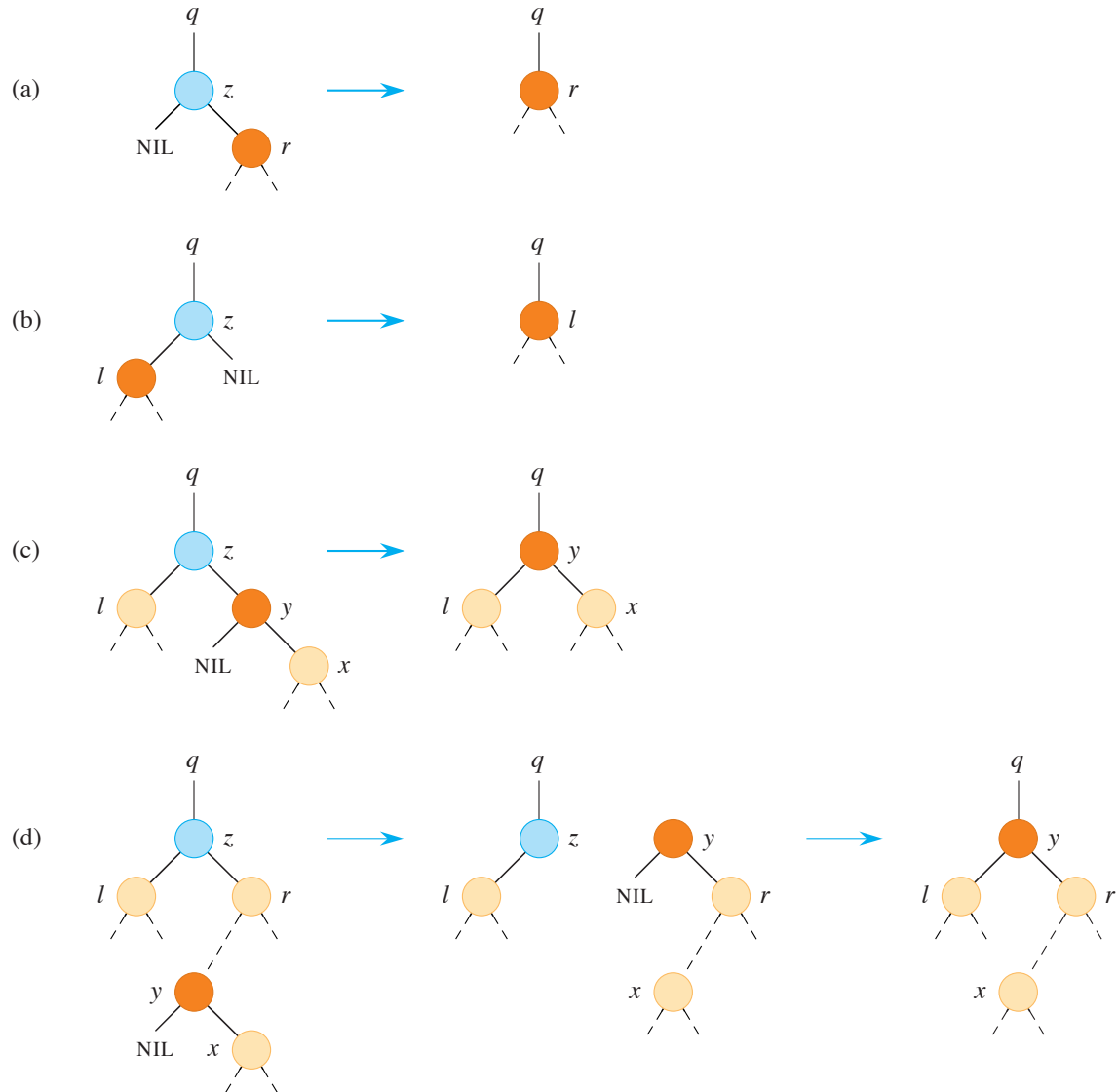
### Deletion

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

```

Each line of  $\text{TREE-DELETE}$ , including the calls to  $\text{TRANSPLANT}$ , takes constant time, except for the call to  $\text{TREE-MINIMUM}$  in line 5. Thus,  $\text{TREE-DELETE}$  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  $\text{INSERT}$  and  $\text{DELETE}$  can be implemented so that each one runs 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.

**12.3-2**

Suppose that you construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is 1 plus the number of nodes examined when the value was first inserted into the tree.

**12.3-3**

You 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?

**12.3-4**

When TREE-DELETE calls TRANSPLANT, under what circumstances can the parameter  $v$  of TRANSPLANT be NIL?

**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.

**12.3-6**

Suppose that instead of each node  $x$  keeping the attribute  $x.p$ , pointing to  $x$ 's parent, it keeps  $x.succ$ , pointing to  $x$ 's successor. Give pseudocode for TREE-SEARCH, TREE-INSERT, and TREE-DELETE on a binary search tree  $T$  using this representation. These procedures should operate in  $O(h)$  time, where  $h$  is the height of the tree  $T$ . You may assume that all keys in the binary search tree are distinct. (*Hint:* You might wish to implement a subroutine that returns the parent of a node.)

**12.3-7**

When node  $z$  in TREE-DELETE has two children, you can choose node  $y$  to be its predecessor rather than its successor. What other changes to TREE-DELETE are necessary if you do so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be minimally changed to implement such a fair strategy?

---

**Problems**
**12-1 Binary search trees with equal keys**

Equal keys pose a problem for the implementation of binary search trees.

- a.* What is the asymptotic performance of TREE-INSERT when used to insert  $n$  items with identical keys into an initially empty binary search tree?

Consider changing TREE-INSERT to test whether  $z.key = x.key$  before line 5 and to test whether  $z.key = y.key$  before line 11. If equality holds, implement one of the following strategies. For each strategy, find the asymptotic performance of inserting  $n$  items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, which compares the keys of  $z$  and  $x$ . Substitute  $y$  for  $x$  to arrive at the strategies for line 11.)

- b.* Keep a boolean flag  $x.b$  at node  $x$ , and set  $x$  to either  $x.left$  or  $x.right$  based on the value of  $x.b$ , which alternates between FALSE and TRUE each time TREE-INSERT visits  $x$  while inserting a node with the same key as  $x$ .
- c.* Keep a list of nodes with equal keys at  $x$ , and insert  $z$  into the list.
- d.* Randomly set  $x$  to either  $x.left$  or  $x.right$ . (Give the worst-case performance and informally derive the expected running time.)

**12-2 Radix trees**

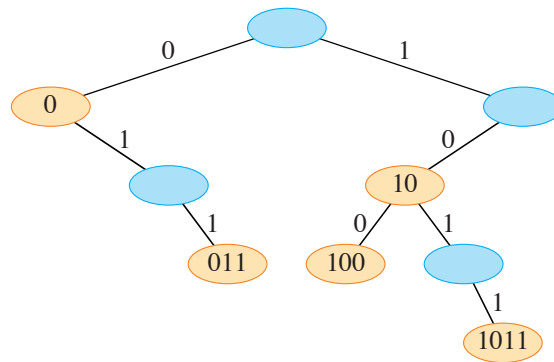
Given two strings  $a = a_0a_1 \dots a_p$  and  $b = b_0b_1 \dots b_q$ , where each  $a_i$  and each  $b_j$  belongs to some ordered set of characters, we say that string  $a$  is *lexicographically less than* string  $b$  if either

1. there exists an integer  $j$ , where  $0 \leq j \leq \min\{p, q\}$ , such that  $a_i = b_i$  for all  $i = 0, 1, \dots, j-1$  and  $a_j < b_j$ , or
2.  $p < q$  and  $a_i = b_i$  for all  $i = 0, 1, \dots, p$ .

For example, if  $a$  and  $b$  are bit strings, then  $10100 < 10110$  by rule 1 (letting  $j = 3$ ) and  $10100 < 101000$  by rule 2. This ordering is similar to that used in English-language dictionaries.

The *radix tree* data structure shown in Figure 12.5 (also known as a *trie*) stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key  $a = a_0a_1 \dots a_p$ , go left at a node of depth  $i$  if  $a_i = 0$  and right if  $a_i = 1$ . Let  $S$  be a set of distinct bit strings whose lengths sum to  $n$ . Show how to use a radix tree to sort  $S$  lexicographically in  $\Theta(n)$  time. For the example in Figure 12.5, the output of the sort should be the sequence 0, 011, 10, 100, 1011.





**Figure 12.5** A radix tree storing the bit strings 1011, 10, 011, 100, and 0. To determine each node's key, traverse the simple path from the root to that node. There is no need, therefore, to store the keys in the nodes. The keys appear here for illustrative purposes only. Keys corresponding to blue nodes are not in the tree. Such nodes are present only to establish a path to other nodes.

### 12-3 Average node depth in a randomly built binary search tree

A **randomly built binary search tree** on  $n$  keys is a binary search tree created by starting with an empty tree and inserting the keys in random order, where each of the  $n!$  permutations of the keys is equally likely. In this problem, you will prove that the average depth of a node in a randomly built binary search tree with  $n$  nodes is  $O(\lg n)$ . The technique reveals a surprising similarity between the building of a binary search tree and the execution of RANDOMIZED-QUICKSORT from Section 7.3.

Denote the depth of any node  $x$  in tree  $T$  by  $d(x, T)$ . Then the **total path length**  $P(T)$  of a tree  $T$  is the sum, over all nodes  $x$  in  $T$ , of  $d(x, T)$ .

*a.* Argue that the average depth of a node in  $T$  is

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T).$$

Thus, you need to show that the expected value of  $P(T)$  is  $O(n \lg n)$ .

*b.* Let  $T_L$  and  $T_R$  denote the left and right subtrees of tree  $T$ , respectively. Argue that if  $T$  has  $n$  nodes, then

$$P(T) = P(T_L) + P(T_R) + n - 1.$$

*c.* Let  $P(n)$  denote the average total path length of a randomly built binary search tree with  $n$  nodes. Show that

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n-1) .$$

d. Show how to rewrite  $P(n)$  as

$$P(n) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n) .$$

e. Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-3, conclude that  $P(n) = O(n \lg n)$ .

Each recursive invocation of randomized quicksort chooses a random pivot element to partition the set of elements being sorted. Each node of a binary search tree partitions the set of elements that fall into the subtree rooted at that node.

f. Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree. (The order in which comparisons are made may differ, but the same comparisons must occur.)

#### 12-4 Number of different binary trees

Let  $b_n$  denote the number of different binary trees with  $n$  nodes. In this problem, you will find a formula for  $b_n$ , as well as an asymptotic estimate.

a. Show that  $b_0 = 1$  and that, for  $n \geq 1$ ,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

b. Referring to Problem 4-5 on page 121 for the definition of a generating function, let  $B(x)$  be the generating function

$$B(x) = \sum_{n=0}^{\infty} b_n x^n .$$

Show that  $B(x) = xB(x)^2 + 1$ , and hence one way to express  $B(x)$  in closed form is

$$B(x) = \frac{1}{2x} (1 - \sqrt{1 - 4x}) .$$

The *Taylor expansion* of  $f(x)$  around the point  $x = a$  is given by

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!} (x - a)^k,$$

where  $f^{(k)}(x)$  is the  $k$ th derivative of  $f$  evaluated at  $x$ .

c. Show that

$$b_n = \frac{1}{n+1} \binom{2n}{n}$$

(the  $n$ th **Catalan number**) by using the Taylor expansion of  $\sqrt{1-4x}$  around  $x = 0$ . (If you wish, instead of using the Taylor expansion, you may use the generalization of the binomial theorem, equation (C.4) on page 1181, to noninteger exponents  $n$ , where for any real number  $n$  and for any integer  $k$ , you can interpret  $\binom{n}{k}$  to be  $n(n-1)\cdots(n-k+1)/k!$  if  $k \geq 0$ , and 0 otherwise.)

d. Show that

$$b_n = \frac{4^n}{\sqrt{\pi n^{3/2}}} (1 + O(1/n)).$$

---

## Chapter notes

Knuth [261] contains a good discussion of simple binary search trees as well as many variations. Binary search trees seem to have been independently discovered by a number of people in the late 1950s. Radix trees are often called “tries,” which comes from the middle letters in the word *retrieval*. Knuth [261] also discusses them.

Many texts, including the first two editions of this book, describe a somewhat simpler method of deleting a node from a binary search tree when both of its children are present. Instead of replacing node  $z$  by its successor  $y$ , delete node  $y$  but copy its key and satellite data into node  $z$ . The downside of this approach is that the node actually deleted might not be the node passed to the delete procedure. If other components of a program maintain pointers to nodes in the tree, they could mistakenly end up with “stale” pointers to nodes that have been deleted. Although the deletion method presented in this edition of this book is a bit more complicated, it guarantees that a call to delete node  $z$  deletes node  $z$  and only node  $z$ .

Section 14.5 will show how to construct an optimal binary search tree when you know the search frequencies before constructing the tree. That is, given the frequencies of searching for each key and the frequencies of searching for values that fall between keys in the tree, a set of searches in the constructed binary search tree examines the minimum number of nodes.