2.3.1 Binary Search Trees

Definition 2.3 [Binary search tree] A *binary search tree* is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:

1. Every element has a key and no two elements have the same key (i.e., the keys are distinct).
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees. □

A *binary search tree* can support the operations search, insert, and delete among others. In fact, with a binary search tree, we can search for a data element both by key value and by rank (i.e., find an element with key x, find the fifth-smallest element, delete the element with key x, delete the fifth-smallest element, insert an element and determine its rank, and so on).

There is some redundancy in the definition of a binary search tree. Properties 2, 3, and 4 together imply that the keys must be distinct. So, property 1 can be replaced by the property: The root has a key.

Some examples of binary trees in which the elements have distinct keys are shown in Figure 2.11. The tree of Figure 2.11(a) is not a binary search tree, despite the fact that it satisfies properties 1, 2, and 3. The right subtree fails to satisfy property 4. This subtree is not a binary search tree, as its right subtree has a key value (22) that is smaller than that in the subtree's root (25). The binary trees of Figure 2.11(b) and (c) are binary search trees.

Searching a Binary Search Tree

Since the definition of a binary search tree is recursive, it is easiest to describe a recursive search method. Suppose we wish to search for an element with key x. An element could in general be an arbitrary structure that has as one of its fields a *key*. We assume for simplicity that the element just consists of a *key* and use the terms element and key interchangeably. We begin at the root. If the root is 0, then the search tree contains no elements and the search is unsuccessful. Otherwise, we compare x with the key in the root. If x equals this key, then the search terminates successfully. If x is less than the key in the root, then no element in the right subtree can have key value x, and only the left subtree is to be searched. If x is larger than the key in the root, only the right subtree needs to be searched. The subtrees can be searched recursively as in Algorithm 2.4. This function assumes a linked

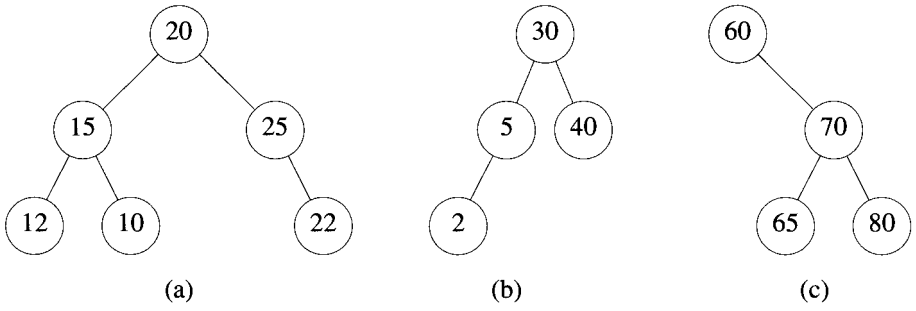


Figure 2.11 Binary trees

representation for the search tree. Each node has the three fields *lchild*, *rchild*, and *data*. The recursion of Algorithm 2.4 is easily replaced by a **while** loop, as in Algorithm 2.5.

```
1  Algorithm Search(t, x)
2  {
3      if (t = 0) then return 0;
4      else if (x = t → data) then return t;
5           else if (x < t → data) then
6                    return Search(t → lchild, x);
7                else return Search(t → rchild, x);
8  }
```

Algorithm 2.4 Recursive search of a binary search tree

If we wish to search by rank, each node should have an additional field $leftsize$, which is one plus the number of elements in the left subtree of the node. For the search tree of Figure 2.11(b), the nodes with keys 2, 5, 30, and 40, respectively, have $leftsize$ equal to 1, 2, 3, and 1. Algorithm 2.6 searches for the kth-smallest element.

As can be seen, a binary search tree of height h can be searched by key as well as by rank in $O(h)$ time.

```
1   Algorithm ISearch(x)
2   {
3       found := false;
4       t := tree;
5       while ((t ≠ 0) and not found) do
6       {
7           if (x = (t → data)) then found := true;
8           else if (x < (t → data)) then t := (t → lchild);
9                else t := (t → rchild);
10      }
11      if (not found) then return 0;
12      else return t;
13  }
```

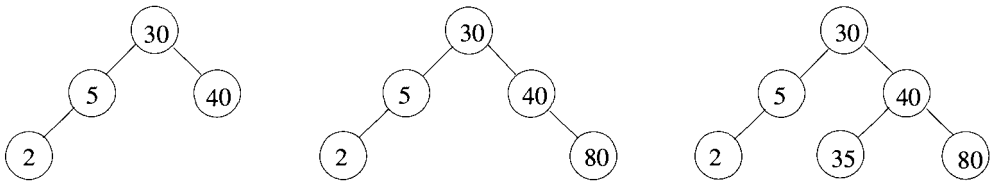
Algorithm 2.5 Iterative search of a binary search tree

```
1   Algorithm Searchk(k)
2   {
3       found := false; t := tree;
4       while ((t ≠ 0) and not found) do
5       {
6           if (k = (t → leftsize)) then found := true;
7           else if (k < (t → leftsize)) then t := (t → lchild);
8                else
9                {
10                   k := k − (t → leftsize);
11                   t := (t → rchild);
12               }
13      }
14      if (not found) then return 0;
15      else return t;
16  }
```

Algorithm 2.6 Searching a binary search tree by rank

Insertion into a Binary Search Tree

To insert a new element x, we must first verify that its key is different from those of existing elements. To do this, a search is carried out. If the search is unsuccessful, then the element is inserted at the point the search terminated. For instance, to insert an element with key 80 into the tree of Figure 2.12(a), we first search for 80. This search terminates unsuccessfully, and the last node examined is the one with key 40. The new element is inserted as the right child of this node. The resulting search tree is shown in Figure 2.12(b). Figure 2.12(c) shows the result of inserting the key 35 into the search tree of Figure 2.12(b).



(a) (b) (c)

Figure 2.12 Inserting into a binary search tree

Algorithm 2.7 implements the insert strategy just described. If a node has a *leftsize* field, then this is updated too. Regardless, the insertion can be performed in $O(h)$ time, where h is the height of the search tree.

Deletion from a Binary Search Tree

Deletion of a leaf element is quite easy. For example, to delete 35 from the tree of Figure 2.12(c), the left-child field of its parent is set to 0 and the node disposed. This gives us the tree of Figure 2.12(b). To delete the 80 from this tree, the right-child field of 40 is set to 0; this gives the tree of Figure 2.12(a). Then the node containing 80 is disposed.

The deletion of a nonleaf element that has only one child is also easy. The node containing the element to be deleted is disposed, and the single child takes the place of the disposed node. So, to delete the element 5 from the tree of Figure 2.12(b), we simply change the pointer from the parent node (i.e., the node containing 30) to the single-child node (i.e., the node containing 2).

```
1   Algorithm Insert(x)
2   // Insert x into the binary search tree.
3   {
4       found := false;
5       p := tree;
6       // Search for x. q is the parent of p.
7       while ((p ≠ 0) and not found) do
8       {
9           q := p; // Save p.
10          if (x = (p → data)) then found := true;
11          else  if (x < (p → data)) then p := (p → lchild);
12                else p := (p → rchild);
13      }

14      // Perform insertion.
15      if (not found) then
16      {
17          p := new TreeNode;
18          (p → lchild) := 0; (p → rchild) := 0; (p → data) := x;
19          if (tree ≠ 0) then
20          {
21              if (x < (q → data)) then (q → lchild) := p;
22              else (q → rchild) := p;
23          }
24          else tree := p;
25      }
26  }
```

Algorithm 2.7 Insertion into a binary search tree

When the element to be deleted is in a nonleaf node that has two children, the element is replaced by either the largest element in its left subtree or the smallest one in its right subtree. Then we proceed to delete this replacing element from the subtree from which it was taken. For instance, if we wish to delete the element with key 30 from the tree of Figure 2.13(a), then we replace it by either the largest element, 5, in its left subtree or the smallest element, 40, in its right subtree. Suppose we opt for the largest element in the left subtree. The 5 is moved into the root, and the tree of Figure 2.13(b) is obtained. Now we must delete the second 5. Since this node has only one child, the pointer from its parent is changed to point to this child. The tree of Figure 2.13(c) is obtained. We can verify that regardless of whether the replacing element is the largest in the left subtree or the smallest in the right subtree, it is originally in a node with a degree of at most one. So, deleting it from this node is quite easy. We leave the writing of the deletion procedure as an exercise. It should be evident that a deletion can be performed in $O(h)$ time if the search tree has a height of h.

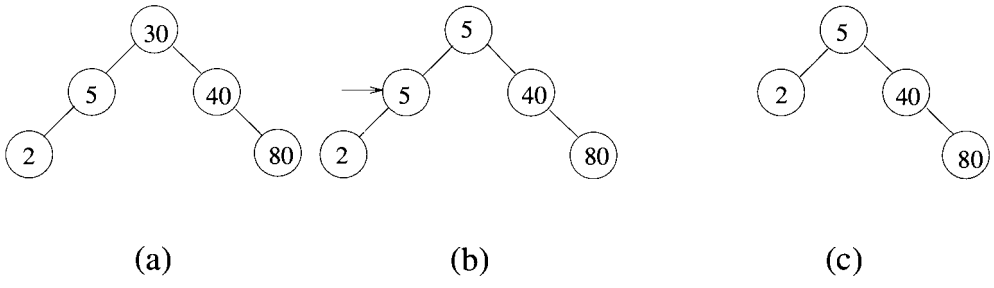


Figure 2.13 Deletion from a binary search tree

Height of a Binary Search Tree

Unless care is taken, the height of a binary search tree with n elements can become as large as n. This is the case, for instance, when Algorithm 2.7 is used to insert the keys $[1, 2, 3, \ldots, n]$, in this order, into an initially empty binary search tree. It can, however, be shown that when insertions and deletions are made at random using the procedures given here, the height of the binary search tree is $O(\log n)$ on the average.

Search trees with a worst-case height of $O(\log n)$ are called *balanced search trees.* Balanced search trees that permit searches, inserts, and deletes to be performed in $O(\log n)$ time are listed in Table 2.1. Examples include AVL trees, 2-3 trees, Red-Black trees, and B-trees. On the other hand splay trees

take $O(\log n)$ time for each of these operations in the *amortized* sense. A description of these balanced trees can be found in the book by E. Horowitz, S. Sahni, and D. Mehta cited at the end of this chapter.

Data structure	search	insert	delete
Binary search tree	$O(n)$ (wc) $O(\log n)$ (av)	$O(n)$ (wc) $O(\log n)$ (av)	$O(n)$ (wc) $O(\log n)$ (av)
AVL tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
2-3 tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Red-Black tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
B-tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Splay tree	$O(\log n)$ (am)	$O(\log n)$ (am)	$O(\log n)$ (am)

Table 2.1 Summary of dictionary implementations. Here (wc) stands for worst case, (av) for average case, and (am) for amortized cost.

2.3.2 Cost Amortization

Suppose that a sequence I1, I2, D1, I3, I4, I5, I6, D2, I7 of insert and delete operations is performed on a set. Assume that the *actual cost* of each of the seven inserts is one. (We use the terms *cost* and *complexity* interchangeably.) By this, we mean that each insert takes one unit of time. Further, suppose that the delete operations D1 and D2 have an actual cost of eight and ten, respectively. So, the total cost of the sequence of operations is 25.

In an amortization scheme we charge some of the actual cost of an operation to other operations. This reduces the charged cost of some operations and increases that of others. The *amortized cost* of an operation is the total cost charged to it. The cost transferring (amortization) scheme is required to be such that the sum of the amortized costs of the operations is greater than or equal to the sum of their actual costs. If we charge one unit of the cost of a delete operation to each of the inserts since the last delete operation (if any), then two units of the cost of D1 get transferred to I1 and I2 (the charged cost of each increases by one), and four units of the cost of D2 get transferred to I3 to I6. The amortized cost of each of I1 to I6 becomes two, that of I7 becomes equal to its actual cost (that is, one), and that of each of D1 and D2 becomes 6. The sum of the amortized costs is 25, which is the same as the sum of the actual costs.

Now suppose we can prove that no matter what sequence of insert and delete operations is performed, we can charge costs in such a way that the amortized cost of each insertion is no more than two and that of each deletion

is no more than six. This enables us to claim that the actual cost of any insert/delete sequence is no more than $2 * i + 6 * d$, where i and d are, respectively, the number of insert and delete operations in the sequence. Suppose that the actual cost of a deletion is no more than ten and that of an insertion is one. Using actual costs, we can conclude that the sequence cost is no more than $i + 10 * d$. Combining these two bounds, we obtain $\min\{2 * i + 6 * d,\ i + 10 * d\}$ as a bound on the sequence cost. Hence, using the notion of cost amortization, we can obtain tighter bounds on the complexity of a sequence of operations.

The amortized time complexity to perform insert, delete, and search operations in splay trees is $O(\log n)$. This amortization is over n operations. In other words, the total time taken for processing an arbitrary sequence of n operations is $O(n \log n)$. Some operations may take much longer than $O(\log n)$ time, but when amortized over n operations, each operation costs $O(\log n)$ time.

EXERCISES

1. Write an algorithm to delete an element x from a binary search tree t. What is the time complexity of your algorithm?

2. Present an algorithm to start with an initially empty binary search tree and make n random insertions. Use a uniform random number generator to obtain the values to be inserted. Measure the height of the resulting binary search tree and divide this height by $\log_2 n$. Do this for $n = 100, 500, 1,000, 2,000, 3,000, \ldots, 10,000$. Plot the ratio height/$\log_2 n$ as a function of n. The ratio should be approximately constant (around 2). Verify that this is so.

3. Suppose that each node in a binary search tree also has the field *leftsize* as described in the text. Design an algorithm to insert an element x into such a binary search tree. The complexity of your algorithm should be $O(h)$, where h is the height of the search tree. Show that this is the case.

4. Do Exercise 3, but this time present an algorithm to delete the element with the kth-smallest key in the binary search tree.

5. Find an efficient data structure for representing a subset S of the integers from 1 to n. Operations we wish to perform on the set are

   - **INSERT**(i) to insert the integer i to the set S. If i is already in the set, this instruction must be ignored.
   - **DELETE** to delete an arbitrary member from the set.
   - **MEMBER**(i) to check whether i is a member of the set.