CHAPTER 10

Efficient Binary Search Trees

10.1 OPTIMAL BINARY SEARCH TREES

Binary search trees were introduced in Chapter 5. In this section, we consider the construction of binary search trees for a static set of elements. That is, we make no additions to or deletions from the set. Only searches are performed.

A sorted list can be searched using a binary search. For this search, we can construct a binary search tree with the property that searching this tree using the function *iterSearch* (Program 5.17) is equivalent to performing a binary search on the sorted list. For instance, a binary search on the sorted element list (5, 10, 15) (*for convenience, all examples in this chapter show only an element's key rather than the complete element*) corresponds to using function *iterSearch* on the binary search tree of Figure 10.1. Although this tree is a full binary tree, it may not be the optimal binary search tree to use when the probabilities with which different elements are searched are different.

To find an optimal binary search tree for a given collection of elements, we must first decide on a cost measure for search trees. When searching for an element at level l,

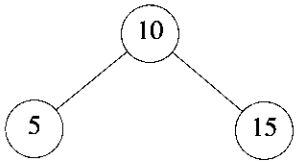


Figure 10.1: Binary search tree corresponding to a binary search on the list (5, 10, 15)

function *iterSearch* makes l iterations of the **while** loop. Since this **while** loop determines the cost of the search, it is reasonable to use the level number of a node as its cost.

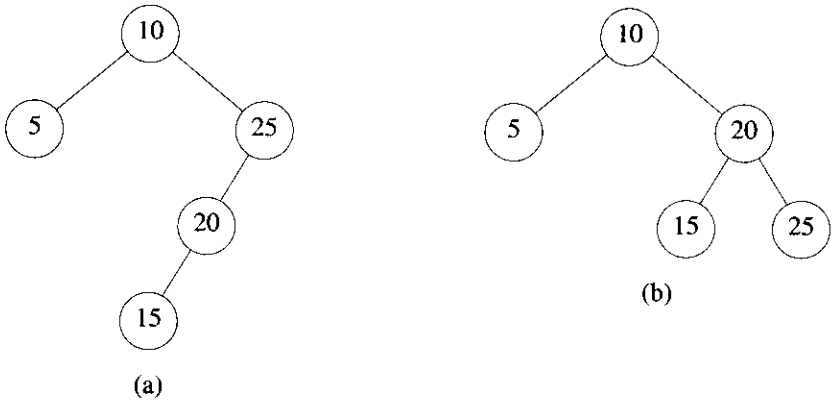


Figure 10.2: Two binary search trees

Example 10.1: Consider the two search trees of Figure 10.2. The second of these requires at most three comparisons to decide whether the element being sought is in the tree. The first binary tree may require four comparisons, since any search key k such that $10 < k < 20$ will test four nodes. Thus, as far as worst-case search time is concerned, the second binary tree is more desirable than the first. To search for a key in the first tree takes one comparison for the 10, two for each of 5 and 25, three for 20, and four for 15. Assuming that each key is searched for with equal probability, the average number of comparisons for a successful search is 2.4. For the second binary search tree this amount

is 2.2. Thus, the second tree has a better average behavior, too.

Suppose that each of 5, 10, 15, 20 and 25 is searched for with probablility 0.3, 0.3, 0.05, 0.05 and 0.3, respectively. The average number of comparisons for a successful search in the trees of Figure 10.2 (a) and (b) is 1.85 and 2.05, respectively. Now, the first tree has better average behavior than the second tree! □

In evaluating binary search trees, it is useful to add a special "square" node at every null link. Doing this to the trees of Figure 10.2 yields the trees of Figure 10.3. Remember that every binary tree with n nodes has $n + 1$ null links and therefore will have $n + 1$ square nodes. We shall call these nodes *external* nodes because they are not part of the original tree. The remaining nodes will be called *internal* nodes. Each time a binary search tree is examined for an identifier that is not in the tree, the search terminates at an external node. Since all such searches are unsuccessful searches, external nodes will also be referred to as *failure nodes*. A binary tree with external nodes added is an *extended binary tree*. The concept of an extended binary tree as just defined is the same as that defined in connection with leftist trees in Chapter 9. Figure 10.3 shows the extended binary trees corresponding to the search trees of Figure 10.2.

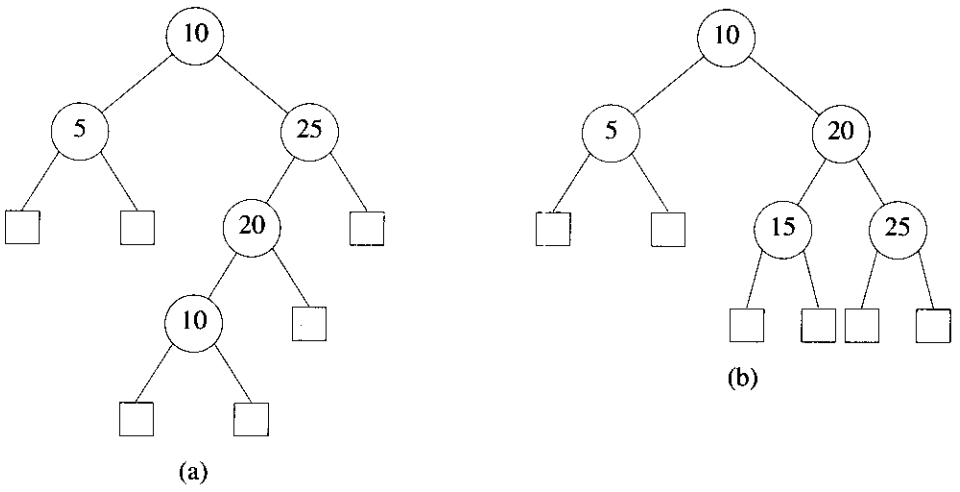


Figure 10.3: Extended binary trees corresponding to search trees of Figure 10.2

We define the *external path length* of a binary tree to be the sum over all external nodes of the lengths of the paths from the root to those nodes. Analogously, the *internal path length* is the sum over all internal nodes of the lengths of the paths from the root to

those nodes. The internal path length, I, for the tree of Figure 10.3(a) is

$$I = 0 + 1 + 1 + 2 + 3 = 7$$

Its external path length, E, is

$$E = 2 + 2 + 4 + 4 + 3 + 2 = 17$$

Exercise 1 of this section shows that the internal and external path lengths of a binary tree with n internal nodes are related by the formula $E = I + 2n$. Hence, binary trees with the maximum E also have maximum I. Over all binary trees with n internal nodes, what are the maximum and minimum possible values for I? The worst case, clearly, is when the tree is skewed (i.e., when the tree has a depth of n). In this case,

$$I = \sum_{i=0}^{n-1} i = n(n-1)/2$$

To obtain trees with minimal I, we must have as many internal nodes as close to the root as possible. We can have at most 2 nodes at distance 1, 4 at distance 2, and in general, the smallest value for I is

$$0 + 2 * 1 + 4 * 2 + 8 * 3 + \cdots +$$

One tree with minimal internal path length is the complete binary tree defined in Section 5.2. If we number the nodes in a complete binary tree as in Section 5.2, then we see that the distance of node i from the root is $\lfloor \log_2 i \rfloor$. Hence, the smallest value for I is

$$\sum_{1 \le i \le n} \lfloor \log_2 i \rfloor = O(n \log_2 n)$$

Let us now return to our original problem of representing a static element set as a binary search tree. Let $a_1, a_2, \cdots, a_n$ with $a_1 < a_2 < \cdots < a_n$ be the element keys. Suppose that the probability of searching for each a_i is p_i. The total cost of any binary search tree for this set of keys is

$$\sum_{1 \le i \le n} p_i \cdot \text{level}(a_i)$$

when only successful searches are made. Since unsuccessful searches (i.e., searches for keys not in the table) will also be made, we should include the cost of these searches in our cost measure, too. Unsuccessful searches terminate with algorithm *iterSearch* (Program 5.17) returning a 0 pointer. Every node with an empty subtree defines a point at which such a termination can take place. Let us replace every empty subtree by a failure node. The keys not in the binary search tree may be partitioned into $n + 1$ classes E_i, $0 \le i \le n$. E_0 contains all keys X such that $X < a_1$. E_i contains all keys X such that

$a_i < X < a_{i+1}$, $1 \le i < n$, and E_n contains all keys X, $X > a_n$. It is easy to see that for all keys in a particular class E_i, the search terminates at the same failure node, and it terminates at different failure nodes for keys in different classes. The failure nodes may be numbered 0 to n, with i being the failure node for class E_i, $0 \le i \le n$. If q_i is the probability that the key being sought is in E_i, then the cost of the failure nodes is

$$\sum_{0 \le i \le n} q_i \cdot (\text{level(failure node } i) - 1)$$

Therefore, the total cost of a binary search tree is

$$\sum_{1 \le i \le n} p_i \cdot \text{level}(a_i) + \sum_{0 \le i \le n} q_i \cdot (\text{level (failure node } i) - 1) \qquad (10.1)$$

An *optimal binary search tree* for $a_1, \cdots, a_n$ is one that minimizes Eq. (10.1) over all possible binary search trees for this set of keys. Note that since all searches must terminate either successfully or unsuccessfully, we have

$$\sum_{1 \le i \le n} p_i + \sum_{0 \le i \le n} q_i = 1$$

Example 10.2: Figure 10.4 shows the possible binary search trees for the key set (a_1,a_2,a_3)= (5, 10, 15). With equal probabilities, $p_i = q_j = 1/7$ for all i and j, we have

cost (tree a) = 15/7; cost (tree b) = 13/7
cost (tree c) = 15/7; cost (tree d) = 15/7
cost (tree e) = 15/7

As expected, tree b is optimal. With $p_1 = 0.5$, $p_2 = 0.1$, $p_3 = 0.05$, $q_0 = 0.15$, $q_1 = 0.1$, $q_2 = 0.05$, and $q_3 = 0.05$ we have

cost (tree a) = 2.65; cost (tree b) = 1.9
cost (tree c) = 1.5; cost (tree d) = 2.05
cost (tree e) = 1.6

Tree c is optimal with this assignment of p's and q's. □

How does one determine the optimal binary search tree? We could proceed as in Example 10.2 and explicitly generate all possible binary search trees, then compute the cost of each tree, and determine the tree with minimum cost. Since the cost of an n-node binary search tree can be determined in $O(n)$ time, the complexity of the optimal binary search tree algorithm is $O(n\,N(n))$, where $N(n)$ is the number of distinct binary search trees with n keys. From Section 5.11 we know that $N(n) = O(4^n/n^{3/2})$. Hence, this brute-force algorithm is impractical for large n. We can find a fairly efficient algorithm by making some observations about the properties of optimal binary search trees.

Let $a_1 < a_2 < \cdots < a_n$ be the n keys to be represented in a binary search tree. Let

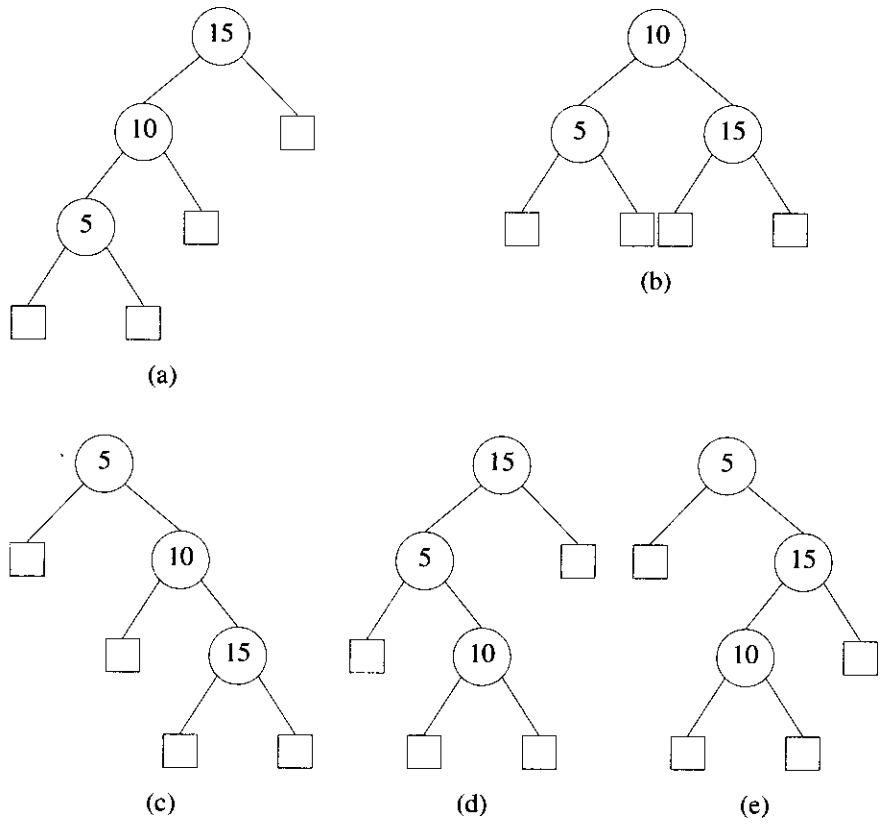


Figure 10.4: Binary search trees with three elements

T_{ij} denote an optimal binary search tree for $a_{i+1}, \cdots, a_j$, $i < j$. By convention T_{ii} is an empty tree for $0 \le i \le n$, and T_{ij} is not defined for $i > j$. Let c_{ij} be the cost of the search tree T_{ij}. By definition c_{ii} will be 0. Let r_{ij} be the root of T_{ij}, and let

$$w_{ij} = q_i + \sum_{k=i+1}^{j} (q_k + p_k)$$

be the weight of T_{ij}. By definition $r_{ii} = 0$, and $w_{ii} = q_i$, $0 \le i \le n$. Therefore, $T_{0n}$is an optimal binary search tree for $a_1, \cdots, a_n$. Its cost is c_{0n}, its weight is w_{0n}, and its root is r_{0n}.

If T_{ij} is an optimal binary search tree for $a_{i+1}, \cdots, a_j$, and $r_{ij} = k$, then k satisfies the inequality $i < k \leq j$. T_{ij} has two subtrees L and R. L is the left subtree and contains the keys $a_{i+1}, \cdots, a_{k-1}$, and R is the right subtree and contains the keys $a_{k+1}, \cdots, a_j$ (Figure 10.5). The cost c_{ij} of T_{ij} is

$$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R) \tag{10.2}$$

where weight (L) = weight $(T_{i,k-1}) = w_{i,k-1}$, and weight (R) = weight $(T_{kj}) = w_{kj}$.

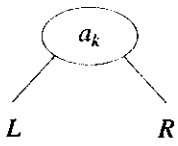


Figure 10.5: An optimal binary search tree T_{ij}

From Eq. (10.2) it is clear that if c_{ij} is to be minimal, then cost$(L) = c_{i,k-1}$ and cost$(R) = c_{kj}$, as otherwise we could replace either L or R by a subtree with a lower cost, thus getting a binary search tree for $a_{i+1}, \cdots, a_j$ with a lower cost than c_{ij}. This violates the assumption that T_{ij} is optimal. Hence, Eq. (10.2) becomes

$$\begin{aligned} c_{ij} &= p_k + c_{i,k-1} + c_{kj} + w_{i,k-1} + w_{kj} \\ &= w_{ij} + c_{i,k-1} + c_{kj} \end{aligned} \tag{10.3}$$

Since T_{ij} is optimal, it follows from Eq. (10.3) that $r_{ij} = k$ is such that

$$w_{ij} + c_{i,k-1} + c_{kj} = \min_{i<l\leq j}\{w_{ij} + c_{i,l-1} + c_{lj}\}$$

or

$$c_{i,k-1} + c_{kj} = \min_{i<l\leq j}\{c_{i,l-1} + c_{lj}\} \tag{10.4}$$

Equation (10.4) gives us a means of obtaining T_{0n} and c_{0n}, starting from the knowledge that $T_{ii} = \phi$ and $c_{ii} = 0$.

Example 10.3: Let $n = 4$ and $(a_1, a_2, a_3, a_4) = (10, 15, 20, 25)$. Let $(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$ and $(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$. The p's and q's have been multiplied by 16 for convenience. Initially, $w_{ii} = q_i$, $c_{ii} = 0$, and $r_{ii} = 0$, $0 \leq i \leq 4$. Using Eqs. (10.3) and (10.4), we get

$$
\begin{aligned}
w_{01} &= p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 8 \\
c_{01} &= w_{01} + \min\{c_{00} + c_{11}\} = 8 \\
r_{01} &= 1 \\
w_{12} &= p_2 + w_{11} + w_{22} = p_2 + q_2 + w_{11} = 7 \\
c_{12} &= w_{12} + \min\{c_{11} + c_{22}\} = 7 \\
r_{12} &= 2 \\
w_{23} &= p_3 + w_{22} + w_{33} = p_3 + q_3 + w_{22} = 3 \\
c_{23} &= w_{23} + \min\{c_{22} + c_{33}\} = 3 \\
r_{23} &= 3 \\
w_{34} &= p_4 + w_{33} + w_{44} = p_4 + q_4 + w_{33} = 3 \\
c_{34} &= w_{34} + \min\{c_{33} + c_{44}\} = 3 \\
r_{34} &= 4
\end{aligned}
$$

Knowing $w_{i,i+1}$ and $c_{i,i+1}$, $0 \le i < 4$, we can use Eqs. (10.3) and (10.4) again to compute $w_{i,i+2}$, $c_{i,i+2}$, $r_{i,i+2}$, $0 \le i < 3$. This process may be repeated until w_{04}, c_{04}, and r_{04} are obtained. The table of Figure 10.6 shows the results of this computation. From the table, we see that $c_{04} = 32$ is the minimal cost of a binary search tree for a_1 to a_4. The root of tree T_{04} is a_2. Hence, the left subtree is T_{01} and the right subtree T_{24}. T_{01} has root a_1 and subtrees T_{00} and T_{11}. T_{24} has root a_3; its left subtree is therefore T_{22} and right subtree T_{34}. Thus, with the data in the table it is possible to reconstruct T_{04}. Figure 10.7 shows T_{04}. □

Example 10.3 illustrates how Eq. (10.4) may be used to determine the c's and r's, as well as how to reconstruct T_{0n} knowing the r's. Let us examine the complexity of this function to evaluate the c's and r's. The evaluation function described in Example 10.3 requires us to compute c_{ij} for $(j - i) = 1, 2, \cdots, n$ in that order. When $j - i = m$, there are $n - m + 1$ c_{ij}'s to compute. The computation of each of these c_{ij}'s requires us to find the minimum of m quantities (see Eq. (10.4)). Hence, each such c_{ij} can be computed in time $O(m)$. The total time for all c_{ij}'s with $j - i = m$ is therefore $O(nm - m^2)$. The total time to evaluate all the c_{ij}'s and r_{ij}'s is

$$\sum_{1 \le m \le n} (nm - m^2) = O(n^3)$$

Actually we can do better than this using a result due to D. E. Knuth that states that the optimal l in Eq. (10.4) may be found by limiting the search to the range $r_{i,j-1} \le l \le r_{i+1,j}$. In this case, the computing time becomes $O(n^2)$ (see Exercise 3). Function *obst* (Program 10.1) uses this result to obtain in $O(n^2)$ time the values of w_{ij}, r_{ij}, and c_{ij}, $0 \le i \le j \le n$. The actual tree T_{0n} may be constructed from the values of

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Figure 10.6: Computation of c_{04} and r_{04}. The computation is carried out by row from row 0 to row 4

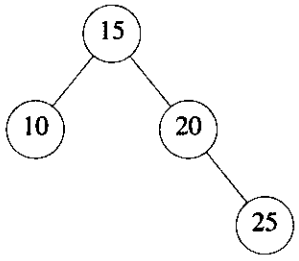


Figure 10.7: Optimal binary search tree for Example 10.3

r_{ij} in O(n) time. The algorithm for this is left as an exercise.

Function *obst* (Program 10.1) computes the cost $c[i][j] = c_{ij}$ of optimal binary search trees T_{ij} for keys $a_{i+1}, \cdots, a_j$. It also computes $r[i][j] = r_{ij}$, the root of T_{ij}. $w[i][j] = w_{ij}$ is the weight of T_{ij}. The two-dimensional arrays c, r and w are global arrays of type **int**. The inputs to this function are the success and failure probability arrays, $p[\ ]$ and $q[\ ]$ and the number of keys n. The array elements $p[0]$ and $a[0]$ are not used.

```
void obst(double *p, double *q, int n)
{
    int i, j, k, m;
    for (i = 0; i < n; i++) {/* initialize */
        /* 0-node trees */
        w[i][i] = q[i]; r[i][i] = c[i][i] = 0;
        /* one-node trees */
        w[i][i+1] = q[i] + q[i+1] + p[i+1];
        r[i][i+1] = i + 1;
        c[i][i+1] = w[i][i+1];
    }
    w[n][n] = q[n]; r[n][n] = c[n][n] = 0;

    /* find optimal trees with m > 1 nodes */
    for (m = 2; m <= n; m++)
        for (i = 0; i <= n- m; i++)
        {
            j = i + m;
            w[i][j] = w[i][j-1] + p[j] + q[j];
            k = KnuthMin(i,j);
            /* KnuthMin returns a value k in the range
               [r[i][j-1], r[i+1][j]] minimizing
               c[i][k-1]+c[k][j] */
            c[i][j] = w[i][j] + c[i][k-1] + c[k][j];
                                  /* Eq. (10.3) */
            r[i][j] = k;
        }
}
```

Program 10.1: Finding an optimal binary search tree

EXERCISES

1. (a) Prove by induction that if T is a binary tree with n internal nodes, I its internal path length, and E its external path length, then $E = I + 2n, n \geq 0$.

   (b) Using the result of (a), show that the average number of comparisons s in a successful search is related to the average number of comparisons, u, in an unsuccessful search by the formula

   $$s = (1 + 1/n)u - 1, n \geq 1$$

2. Use function *obst* (Program 10.1), to compute w_{ij}, r_{ij}, and c_{ij}, $0 \leq i < j \leq 4$, for the key set $(a_1, a_2, a_3, a_4) = (5, 10, 15, 20)$, with $p_1 = 1/20$, $p_2 = 1/5$, $p_3 = 1/10$, $p_4 = 1/20$, $q_0 = 1/5$, $q_1 = 1/10$, $q_2 = 1/5$, $q_3 = 1/20$, and $q_4 = 1/20$. Using the r_{ij}'s, construct the optimal binary search tree.

3. (a) Complete function *obst* by providing the code for function *KnuthMin*.

   (b) Show that the computing time complexity of *obst* is $O(n^2)$.

   (c) Write a C function to construct the optimal binary search tree T_{0n} given the roots r_{ij}, $0 \leq i < j \leq n$. Show that this can be done in time $O(n)$.

4. Since, often, only the approximate values of the p's and q's are known, it is perhaps just as meaningful to find a binary search tree that is nearly optimal (i.e., its cost, Eq. (10.1), is almost minimal for the given p's and q's). This exercise explores an $O(n \log n)$ algorithm that results in nearly optimal binary search trees. The search tree heuristic we shall study is

   Choose the root a_k such that $|w_{0,k-1} - w_{k,n}|$ is as small as possible. Repeat this process to find the left and right subtrees of a_k.

   (a) Using this heuristic obtain the resulting binary search tree for the data of Exercise 2. What is its cost?

   (b) Write a C function implementing the above heuristic. The time complexity of your function should be $O(n \log n)$.

   An analysis of the performance of this heuristic may be found in the paper by Mehlhorn (see the References and Selected Readings section).

10.2 AVL TREES

Dynamic collections of elements may also be maintained as binary search trees. In Chapter 5, we saw how insertions and deletions can be performed on binary search trees. Figure 10.8 shows the binary search tree obtained by entering the months JANUARY to DECEMBER in that order into an initially empty binary search tree by using function

insert (Program 5.21).

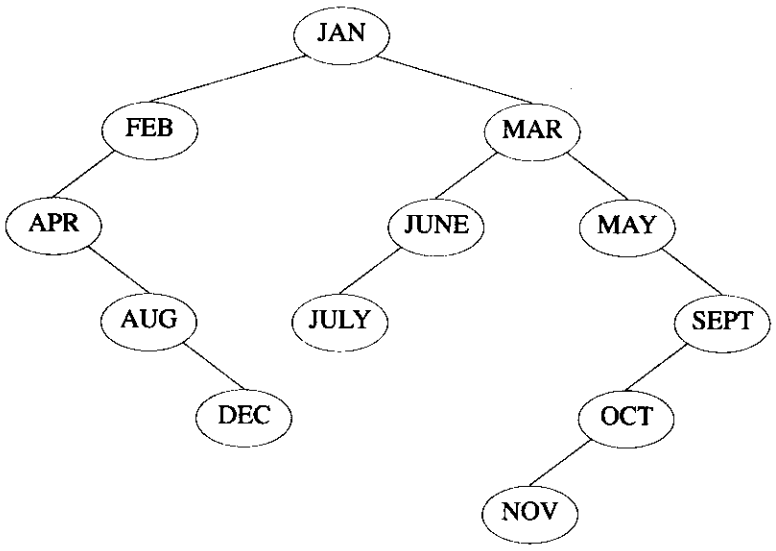


Figure 10.8: Binary search tree obtained for the months of the year

The maximum number of comparisons needed to search for any key in the tree of Figure 10.8 is six for NOVEMBER. The average number of comparisons is (1 for JANUARY + 2 each for FEBRUARY and MARCH + 3 each for APRIL, JUNE and MAY + $\cdots$ + 6 for NOVEMBER)/12 = 42/12 = 3.5. If the months are entered in the order JULY, FEBRUARY, MAY, AUGUST, DECEMBER, MARCH, OCTOBER, APRIL, JANUARY, JUNE, SEPTEMBER, NOVEMBER, then the tree of Figure 10.9 is obtained.

The tree of Figure 10.9 is well balanced and does not have any paths to a leaf node that are much longer than others. This is not true of the tree of Figure 10.8, which has six nodes on the path from the root to NOVEMBER and only two nodes on the path to APRIL. Moreover, during the construction of the tree of Figure 10.9, all intermediate trees obtained are also well balanced. The maximum number of key comparisons needed to find any key is now 4, and the average is 37/12 $\approx$ 3.1. If the months are entered in lexicographic order, instead, the tree degenerates to a chain as in Figure 10.10. The maximum search time is now 12 key comparisons, and the average is 6.5. Thus, in the worst case, searching a binary search tree corresponds to sequential searching in a sorted linear list. When the keys are entered in a random order, the tree tends to be balanced as in

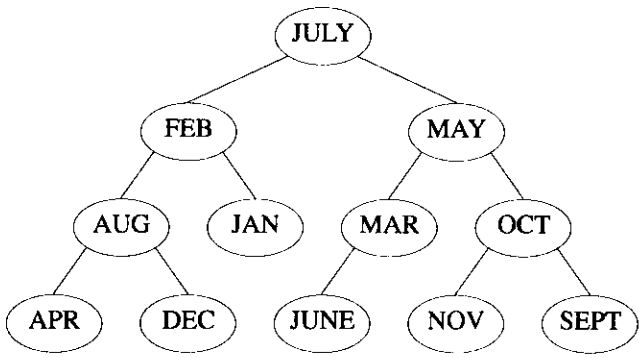


Figure 10.9: A balanced tree for the months of the year

Figure 10.9. If all permutations are equally probable, then the average search and insertion time is O(log n) for an n-node binary search tree.

From our earlier study of binary trees, we know that both the average and maximum search time will be minimized if the binary search tree is maintained as a complete binary tree at all times. However, since we are dealing with a dynamic situation, it is difficult to achieve this ideal without making the time required to insert a key very high. This is so because in some cases it would be necessary to restructure the whole tree to accommodate the new entry and at the same time have a complete binary search tree. It is, however, possible to keep the tree balanced to ensure both an average and worst-case search time of O(log n) for a tree with n nodes. In this section, we study one method of growing balanced binary trees. These balanced trees will have satisfactory search, insertion and deletion time properties. Other ways to maintain balanced search trees are studied in later sections.

In 1962, Adelson-Velskii and Landis introduced a binary tree structure that is balanced with respect to the heights of subtrees. As a result of the balanced nature of this type of tree, dynamic retrievals can be performed in O(log n) time if the tree has n nodes in it. At the same time, a new key can be entered or deleted from such a tree in time O(log n). The resulting tree remains height-balanced. This tree structure is called an AVL tree. As with binary trees, it is natural to define AVL trees recursively.

Definition: An empty tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees respectively, then T is *height-balanced* iff (1) T_L and T_R are height-balanced and (2) $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively. □

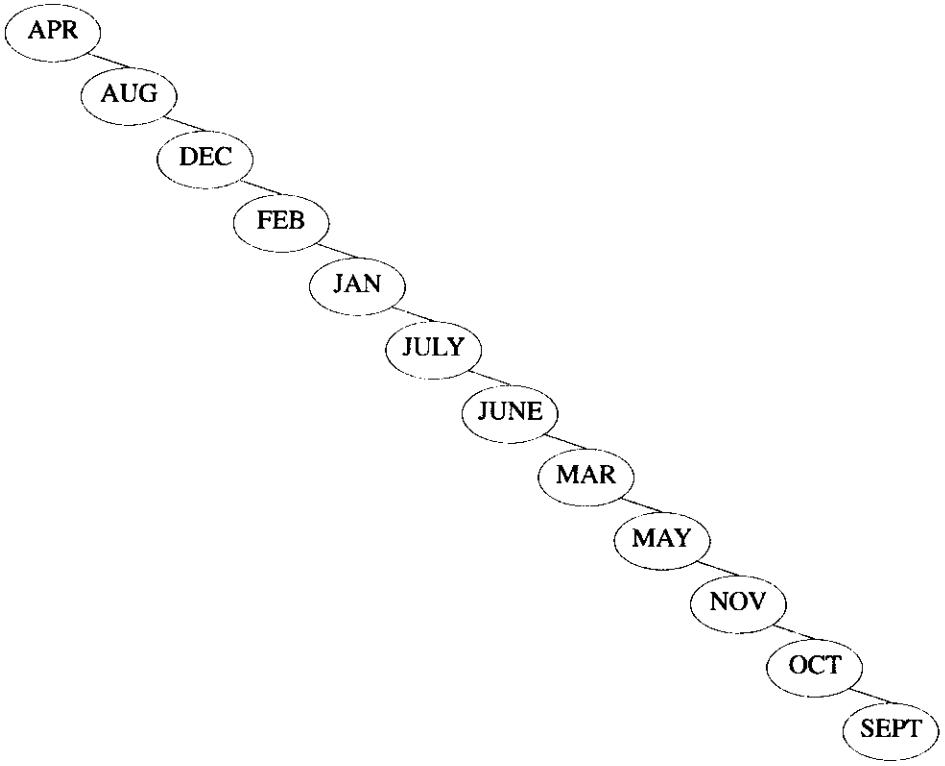


Figure 10.10: Degenerate binary search tree

The definition of a height-balanced binary tree requires that every subtree also be height-balanced. The binary tree of Figure 10.8 is not height-balanced, since the height of the left subtree of the tree with root APRIL is 0 and that of the right subtree is 2. The tree of Figure 10.9 is height-balanced while that of Figure 10.10 is not. To illustrate the processes involved in maintaining a height-balanced binary search tree, let us try to construct such a tree for the months of the year. This time let us assume that the insertions are made in the following order: MARCH, MAY, NOVEMBER, AUGUST, APRIL, JANUARY, DECEMBER, JULY, FEBRUARY, JUNE, OCTOBER, SEPTEMBER. Figure 10.11 shows the tree as it grows and the restructuring involved in keeping the tree balanced. The numbers above each node represent the difference in heights between the left and right subtrees of that node. This number is referred to as the balance factor of the node.

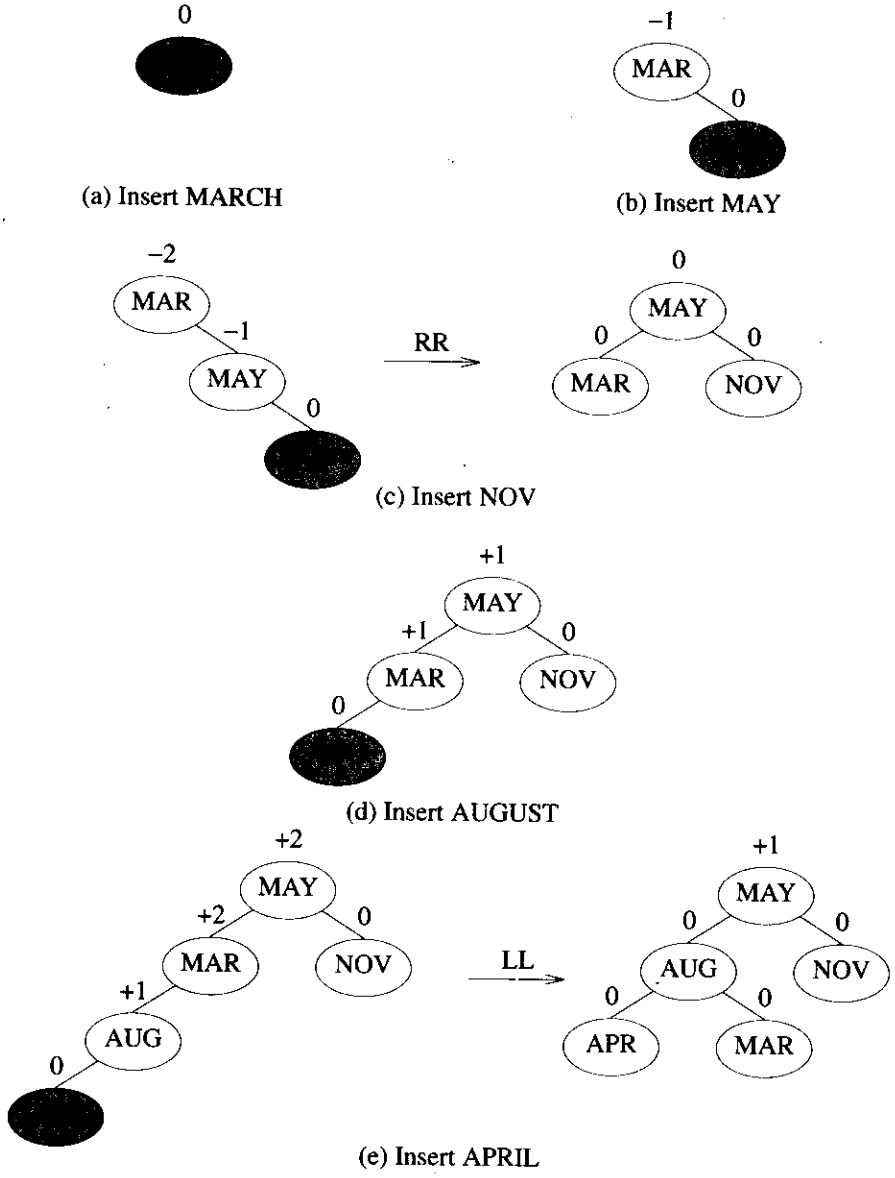
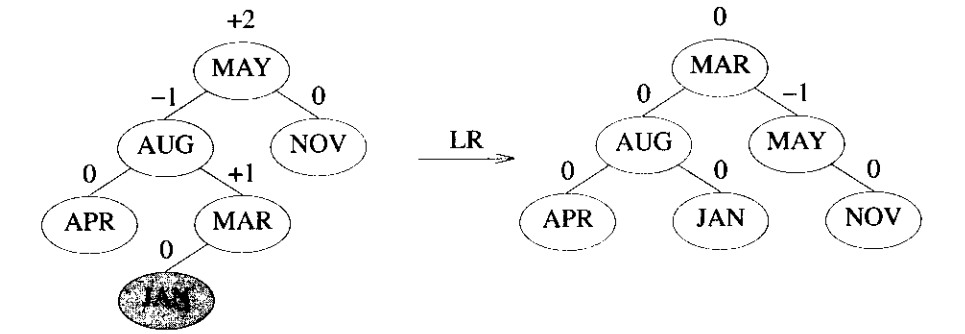
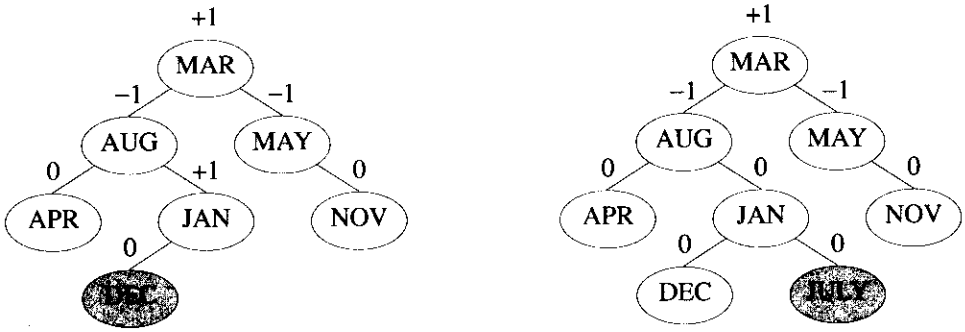


Figure 10.11: Balanced trees obtained for the months of the year (continued on next page)



(f) Insert JANUARY



(g) Insert DECEMBER

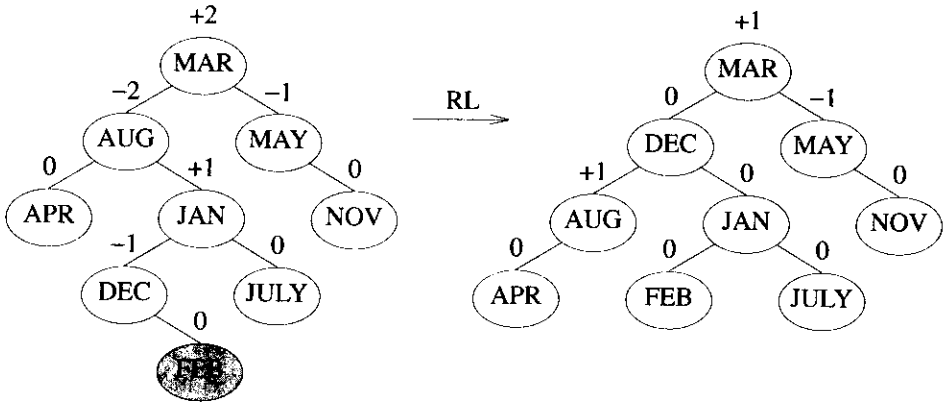
(h) Insert JULY

Figure 10.11: Balanced trees obtained for the months of the year (continued on next page)

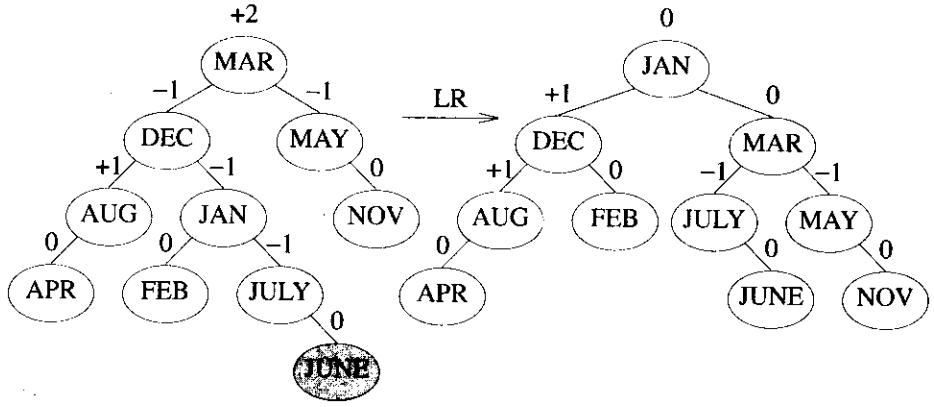
Definition: The *balance factor*, $BF(T)$, of a node T in a binary tree is defined to be $h_L - h_R$, where h_L and h_R, respectively, are the heights of the left and right subtrees of T. For any node T in an AVL tree, $BF(T) = -1, 0$, or 1. □

Inserting MARCH and MAY results in the binary search trees (a) and (b) of Figure 10.11. When NOVEMBER is inserted into the tree, the height of the right subtree of MARCH becomes 2, whereas that of the left subtree is 0. The tree has become unbalanced. To rebalance the tree, a rotation is performed. MARCH is made the left child of MAY, and MAY becomes the root (Figure 10.11(c)). The introduction of AUGUST leaves the tree balanced (Figure 10.11(d)).

The next insertion, APRIL, causes the tree to become unbalanced again. To



(i) Insert FEBRUARY



(j) Insert JUNE

Figure 10.11: Balanced trees obtained for the months of the year (continued on next page)

rebalance the tree, another rotation is performed. This time, it is a clockwise rotation. MARCH is made the right child of AUGUST, and AUGUST becomes the root of the subtree (Figure 10.11(e)). Note that both of the previous rotations were carried out with respect to the closest parent of the new node that had a balance factor of ±2. The insertion of JANUARY results in an unbalanced tree. This time, however, the rotation involved is somewhat more complex than in the earlier situations. The common point,

however, is that the rotation is still carried out with respect to the nearest parent of JANUARY with a balance factor ±2. MARCH becomes the new root. AUGUST, together with its left subtree, becomes the left subtree of MARCH. The left subtree of MARCH becomes the right subtree of AUGUST. MAY and its right subtree, which have keys greater than MARCH, become the right subtree of MARCH. (If MARCH had had a nonempty right subtree, this could have become the left subtree of MAY, since all keys would have been less than MAY.)

Inserting DECEMBER and JULY necessitates no rebalancing. When FEBRUARY is inserted, the tree becomes unbalanced again. The rebalancing process is very similar to that used when JANUARY was inserted. The nearest parent with balance factor ±2 is AUGUST. DECEMBER becomes the new root of that subtree. AUGUST, with its left subtree, becomes the left subtree. JANUARY, with its right subtree, becomes the right subtree of DECEMBER; FEBRUARY becomes the left subtree of JANUARY. (If DECEMBER had had a left subtree, it would have become the right subtree of AUGUST.) The insertion of JUNE requires the same rebalancing as in Figure 10.11(f). The rebalancing following the insertion of OCTOBER is identical to that following the insertion of NOVEMBER. Inserting SEPTEMBER leaves the tree balanced.

In the preceding example we saw that the addition of a node to a balanced binary search tree could unbalance it. The rebalancing was carried out using four different kinds of rotations: LL, RR, LR, and RL (Figure 10.11 (e), (c), (f), and (i), respectively). LL and RR are symmetric, as are LR and RL. These rotations are characterized by the nearest ancestor, A, of the inserted node, Y, whose balance factor becomes ±2. The following characterization of rotation types is obtained:

LL: new node Y is inserted in the left subtree of the left subtree of A

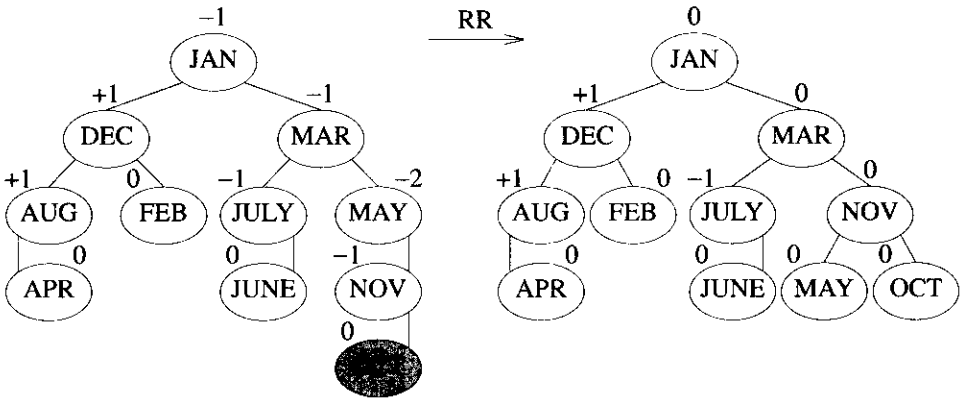
LR: Y is inserted in the right subtree of the left subtree of A

RR: Y is inserted in the right subtree of the right subtree of A

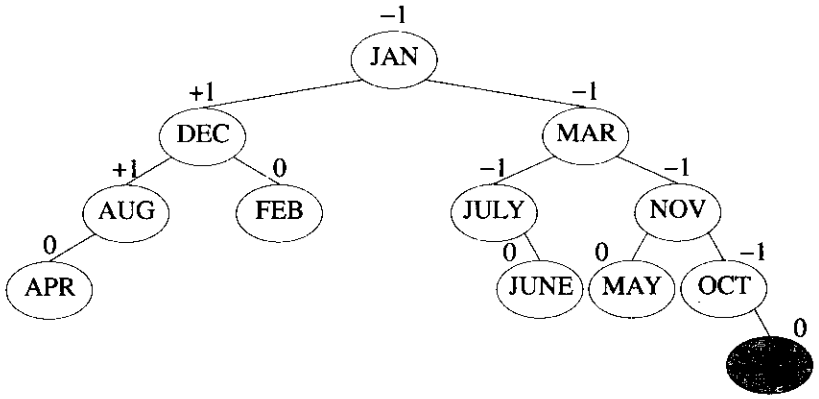
RL: Y is inserted in the left subtree of the right subtree of A

A moment's reflection will show that if a height-balanced binary tree becomes unbalanced as a result of an insertion, then these are the only four cases possible for rebalancing. Figures 10.12 and 10.13 show the LL and LR rotations in terms of abstract binary trees. The RR and RL rotations are symmetric. The root node in each of the trees of the figures represents the nearest ancestor whose balance factor has become ±2 as a result of the insertion. In the example of Figure 10.11 and in the rotations of Figures 10.12 and 10.13, notice that the height of the subtree involved in the rotation is the same after rebalancing as it was before the insertion. This means that once the rebalancing has been carried out on the subtree in question, examining the remaining tree is unnecessary. The only nodes whose balance factors can change are those in the subtree that is rotated.

The transformations done to remedy LL and RR imbalances are often called *single rotations*, while those done for LR and RL imbalances are called *double rotations*. The



(k) Insert OCTOBER



(l) Insert SEPTEMBER

Figure 10.11: Balanced trees obtained for the months of the year

transformation for an LR imbalance can be viewed as an RR rotation followed by an LL rotation, while that for an RL imbalance can be viewed as an LL rotation followed by an RR rotation.

To carry out the rotations of Figures 10.12 and 10.13, it is necessary to locate the node A around which the rotation is to be performed. As remarked earlier, this is the nearest ancestor of the newly inserted node whose balance factor becomes ± 2. For a node's balance factor to become ± 2, its balance factor must have been ± 1 before the insertion. Therefore, before the insertion, the balance factors of all nodes on the path

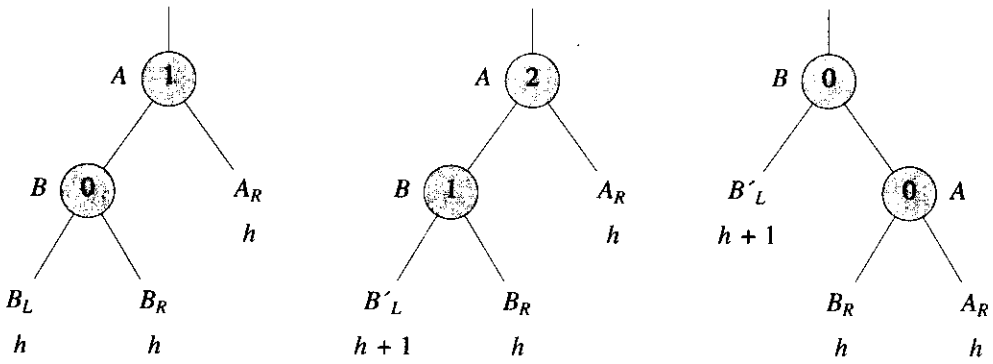
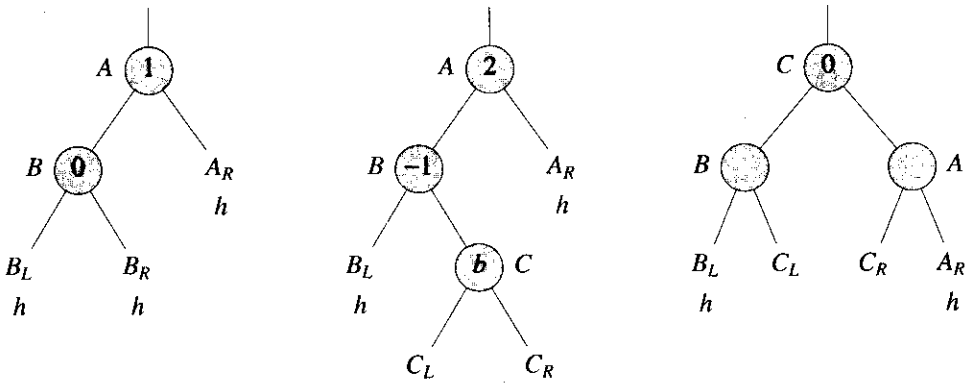


Figure 10.12: An LL rotation

from A to the new insertion point must have been 0. With this information, the node A is readily determined to be the nearest ancestor of the new node having a balance factor ± 1 before insertion. To complete the rotations, the address of F, the parent of A, is also needed. The changes in the balance factors of the relevant nodes are shown in Figures 10.12 and 10.13. Knowing F and A, these changes can be carried out easily.

What happens when the insertion of a node does not result in an unbalanced tree (see Figure 10.11 (a), (b), (d), (g), (h), and (l))? Although no restructuring of the tree is needed, the balance factors of several nodes change. Let A be the nearest ancestor of the new node with balance factor ± 1 before insertion. If, as a result of the insertion, the tree did not get unbalanced, even though some path length increased by 1, it must be that the new balance factor of A is 0. If there is no ancestor A with a balance factor ± 1 (as in Figure 10.11 (a), (b), (d), (g), and (l)), let A be the root. The balance factors of nodes from A to the parent of the new node will change to ± 1 (see Figure 10.11 (h); A = JANUARY). Note that in both cases, the procedure to determine A is the same as when rebalancing is needed. The remaining details of the insertion-rebalancing process are spelled out in function *avlInsert* (Program 10.2). The function *leftRotation* (Program 10.3) gives the code for the *LL* and *LR* rotations. The code for the *RR* and *RL* rotations is symmetric and we leave it as an exercise. The type definitions in use are:



(a) Before insertion (b) After inserting into B_R (c) After LR rotation

$b=0 \Rightarrow bf(B)=bf(A)=0$ after rotation
$b=1 \Rightarrow bf(B)=0$ and $bf(A)=-1$ after rotation
$b=-1 \Rightarrow bf(B)=1$ and $bf(A)=0$ after rotation

Figure 10.13: An LR rotation

```
typedef struct {
        int key;
        } element;
typedef struct treeNode *treePointer;
        struct {
               treePointer leftChild;
               element     data;
               short int   bf;
               treePointer rightChild;
               } treeNode;
```

The pointer to the tree *root* is set to *NULL* before to the first call of *avlInsert*. We also set *unbalanced* to *FALSE* **before each call** to *avlInsert*. The function call is *avlInsert(&root, x, &unbalanced)*.

To really understand the insertion algorithm, you should apply it to the example of Figure 10.11. Once you are convinced that it keeps the tree balanced, then the next question is how much time does it take to make an insertion? An analysis of the algorithm reveals that if h is the height of the tree before insertion, then the time to insert a

```
void avlInsert(treePointer *parent, element x,
                                    int *unbalanced)
{
    if (!*parent) {/* insert element into null tree */
        *unbalanced = TRUE;
        MALLOC(*parent, sizeof(treeNode));
        (*parent)→leftChild =
                    (*parent)→rightChild = NULL;
        (*parent)→bf = 0; (*parent)→data = x;
    }
    else if (x.key < (*parent)→data.key) {
        avlInsert(&(*parent)→leftChild, x, unbalanced);
        if (*unbalanced)
        /* left branch has grown higher */
            switch ((*parent)→bf) {
                case -1: (*parent)→bf = 0;
                         *unbalanced = FALSE; break;
                case  0: (*parent)→bf = 1; break;
                case  1: leftRotation(parent,unbalanced);
            }
     }
     else if (x.key > (*parent)→data.key) {
        avlInsert(&(*parent)→rightChild, x, unbalanced);
        if (*unbalanced)
            /* right branch has grown higher */
            switch((*parent)→bf) {
                case 1 : (*parent)→bf = 0;
                         *unbalanced = FALSE; break;
                case 0 : (*parent)→bf = -1; break;
                case -1: rightRotation(parent, unbalanced);
            }
        }
        else {
            *unbalanced = FALSE;
            printf("The key is already in the tree");
        }
}
```

Program 10.2: Insertion into an AVL tree

```
void leftRotation(treePointer *parent, int *unbalanced)
{
    treePointer grandChild, child;
    child = (*parent)→leftChild;
    if (child→bf == 1) {
        /* LL rotation */
        (*parent)→leftChild = child→rightChild;
        child→rightChild = *parent;
        (*parent)→bf = 0;
        (*parent) = child;
    }
    else {
        /* LR rotation */
        grandChild = child→rightChild;
        child→rightChild = grandChild→leftChild;
        grandChild→leftChild = child;
        (*parent)→leftChild = grandChild→rightChild;
        grandChild→rightChild = *parent;
        switch(grandChild→bf) {
            case 1: (*parent)→bf = -1;
                    child→bf = 0;
                    break;
            case 0: (*parent)→bf = child→bf = 0;
                    break;
            case -1: (*parent)→bf = 0;
                     child→bf = 1;
        }
        *parent = grandChild;
    }
    (*parent)→bf = 0;
    *unbalanced = FALSE;
}
```

Program 10.3: Left rotation function

new identifier is O(h). This is the same as for unbalanced binary search trees, although the overhead is significantly greater now. In the case of binary search trees, however, if there were n nodes in the tree, then h could be n (Figure 10.10) and the worst case insertion time would be O(n). In the case of AVL trees, since h is at most O(log n), the worst case insertion time is O(log n). To see this, let N_h be the minimum number of nodes in a

height balanced tree of height h. In the worst case, the height of one of the subtrees is $h - 1$ and the height of the other is $h - 2$. Both these subtrees are also height balanced. Hence, $N_h = N_{h-1} + N_{h-2} + 1$ and $N_0 = 0$, $N_1 = 1$ and $N_2 = 2$. Notice the similarity between this recursive definition for N_h and the definition of the Fibonacci numbers $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, and $F_1 = 1$. In fact, we can show (Exercise 2) that $N_h = F_{h+2} - 1$ for $h \geq 0$. From Fibonacci number theory we know that $F_h \approx \phi^h / \sqrt{5}$ where $\phi = (1 + \sqrt{5})/2$. Hence, $N_h \approx \phi^{h+2} / \sqrt{5} - 1$. This means that if there are n nodes in the tree, then its height, h, is at most $\log_\phi (\sqrt{5}(n + 1)) - 2$. Therefore, the worst case insertion time for a height balanced tree with n nodes is O(log n).

The exercises show that it is possible to find and delete an element with a specified key and to find and delete the element with the kth smallest key from a height-balanced tree in O(log n) time. Results of an empirical study of deletion in height-balanced trees may be found in the paper by Karlton et al. (see the References and Selected Readings section). Their study indicates that a random insertion requires no rebalancing, a rebalancing rotation of type LL or RR, and a rebalancing rotation of type LR and RL, with probabilities 0.5349, 0.2327, and 0.2324, respectively. Figure 10.14 compares the worst-case times of certain operations on sorted sequential lists, sorted linked lists, and AVL trees.

Operation	Sequential list	Linked list	AVL tree
Search for element with key k	O(log n)	O(n)	O(log n)
Search for jth item	O(1)	O(j)	O(log n)
Delete element with key k	O(n)	O(1)[1]	O(log n)
Delete jth element	O($n - j$)	O(j)	O(log n)
Insert	O(n)	O(1)[2]	O(log n)
Output in order	O(n)	O(n)	O(n)

1. Doubly linked list and position of k known
2. Position for insertion known

Figure 10.14: Comparison of various structures

EXERCISES

1. (a) Convince yourself that Figures 10.12 and 10.13 together with the cases for the symmetric rotations RR and RL takes care of all the possible situations that may arise when a height-balanced binary tree becomes unbalanced as a result of an insertion. Alternately, come up with an example that is not covered by any of the cases in this figure.

(b) Draw the transformations for the rotation types RR and RL.

2. Show that the LR rotation of Figure 10.13 is equivalent to an RR rotation followed by an LL rotation and that an RL rotation is equivalent to an LL rotation followed by an RR rotation.
3. Prove by induction that the minimum number of nodes in an AVL tree of height h is $N_h = F_{h+2} - 1$, $h \geq 0$.
4. Complete *avlInsert* (Program 10.2) by filling in the code needed to rebalance the tree in case of a right imbalance.
5. Start with an empty AVL tree and perform the following sequence of insertions: DECEMBER, JANUARY, APRIL, MARCH, JULY, AUGUST, OCTOBER, FEBRUARY, NOVEMBER, MAY, JUNE. Use the strategy of *avlInsert* to perform each insert. Draw the AVL tree following each insertion and state the rotation type (if any) for each insert.
6. Assume that each node in an AVL tree has the data member *lsize*. For any node, a, $a \rightarrow lsize$ is the number of nodes in its left subtree plus one. Write a C function to locate the kth smallest key in the tree. Show that this can be done in $O(\log n)$ time if there are n nodes in the tree.
7. Rewrite the insertion function *avlInsert with the added assumption that each node has an lsize* data member as in Exercise 6. Show that the insertion time remains $O(\log n)$.
8. Write a C function to list the elements of an AVL tree in ascending order of key. Show that this can be done in $O(n)$ time if the tree has n nodes.
9. Write an algorithm to delete the element with key k from an AVL tree. The resulting tree should be restructured if necessary. Show that the time required for this is $O(\log n)$ when there are n nodes in the tree. [**Hint**: If k is not in a leaf, then replace k by the largest value in its left subtree or the smallest value in its right subtree. Continue until the deletion propagates to a leaf. Deletion from a leaf can be handled using the reverse of the transformations used for insertion.]
10. Do Exercise 9 for the case when each node has an *lsize* data member and the kth smallest key is to be deleted.
11. Complete Figure 10.14 by adding a column for hashing.
12. For a fixed k, $k \geq 1$, we define a height-balanced tree $HB(k)$ as below:

    Definition: An empty binary tree is an $HB(k)$ tree. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is $HB(k)$ iff (a) T_L and T_R are $HB(k)$ and (b) $|h_L - h_R| \leq k$, where h_L and h_R are the heights of T_L and T_R, respectively. □

    (a) Obtain the rebalancing transformations for $HB(2)$.

    (b) Write an insertion algorithm for $HB(2)$ trees.

10.3 RED-BLACK TREES

10.3.1 Definition

A *red-black tree* is a binary search tree in which every node is colored either red or black. The remaining properties satisfied by a red-black tree are best stated in terms of the corresponding extended binary tree. Recall, from Section 9.2, that we obtain an extended binary tree from a regular binary tree by replacing every null pointer with an external node. The additional properties are

RB1. The root and all external nodes are colored black.

RB2. No root-to-external-node path has two consecutive red nodes.

RB3. All root-to-external-node paths have the same number of black nodes.

An equivalent definition arises from assigning colors to the pointers between a node and its children. The pointer from a parent to a black child is black and to a red child is red. Additionally,

RB1´. Pointers from an internal node to an external node are black.

RB2´. No root-to-external-node path has two consecutive red pointers.

RB3´. All root-to-external-node paths have the same number of black pointers.

Notice that if we know the pointer colors, we can deduce the node colors and vice versa. In the red-black tree of Figure 10.15, the external nodes are shaded squares, black nodes are shaded circles, red nodes are unshaded circles, black pointers are thick lines, and red pointers are thin lines. Notice that every path from the root to an external node has exactly two black pointers and three black nodes (including the root and the external node); no such path has two consecutive red nodes or pointers.

Let the *rank* of a node in a red-black tree be the number of black pointers (equivalently the number of black nodes minus 1) on any path from the node to any external node in its subtree. So the rank of an external node is 0. The rank of the root of Figure 10.15 is 2, that of its left child is 2, and of its right child is 1.

Lemma 10.1: Let the length of a root-to-external-node path be the number of pointers on the path. If P and Q are two root-to-external-node paths in a red-black tree, then $length\,(P) \leq 2length\,(Q)$.

Proof: Consider any red-black tree. Suppose that the rank of the root is r. From RB1 the last pointer on each root-to-external-node path is black. From RB2´ no such path has two consecutive red pointers. So each red pointer is followed by a black pointer. As a

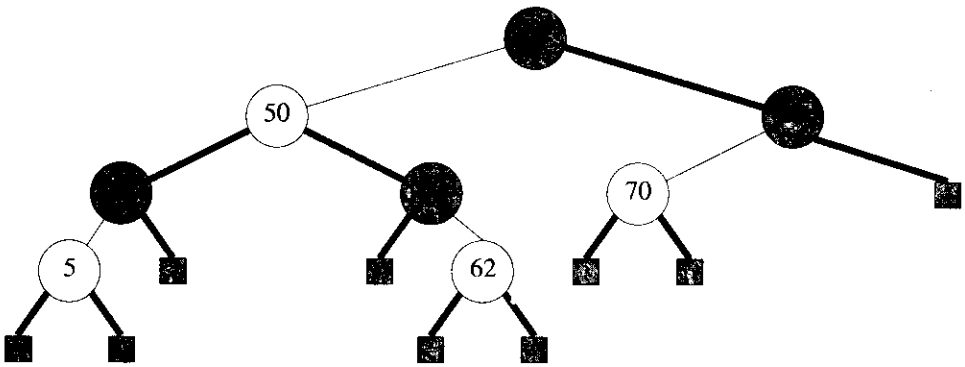


Figure 10.15: A red-black tree

result, each root-to-external-node path has between r and $2r$ pointers, so $length(P) \leq 2length(Q)$. To see that the upper bound is possible, consider the red-black tree of Figure 10.15. The path from the root to the left child of 5 has length 4, while that to the right child of 80 has length 2. □

Lemma 10.2: Let h be the height of a red-black tree (excluding the external nodes), let n be the number of internal nodes in the tree, and let r be the rank of the root.

(a) $h \leq 2r$

(b) $n \geq 2^r - 1$

(c) $h \leq 2\log_2(n+1)$

Proof: From the proof of Lemma 10.1, we know that no root-to-external-node path has length $> 2r$, so $h \leq 2r$. (The height of the red-black tree of Figure 10.15 with external nodes removed is $2r = 4$.)

Since the rank of the root is r, there are no external nodes at levels 1 through r, so there are $2^r - 1$ internal nodes at these levels. Consequently, the total number of internal nodes is at least this much. (In the red-black tree of Figure 10.15, levels 1 and 2 have $3 = 2^2 - 1$ internal nodes. There are additional internal nodes at levels 3 and 4.)

From (b) it follows that $r \leq \log_2(n+1)$. This inequality together with (a) yields (c). □

Since the height of a red-black tree is at most $2\log_2(n+1)$, search, insert, and

delete algorithms that work in $O(h)$ time have complexity $O(\log n)$.

Notice that the worst-case height of a red-black tree is more than the worst-case height (approximately $1.44\log_2(n+2)$) of an AVL tree with the same number of (internal) nodes.

10.3.2 Representation of a Red-Black Tree

Although it is convenient to include external nodes when defining red-black trees, in an implementation null pointers, rather than physical nodes, represent external nodes. Further, since pointer and node colors are closely related, with each node we need to store only its color or the color of the two pointers to its children. Node colors require just one additional bit per node, while pointer colors require two. Since both schemes require almost the same amount of space, we may choose between them on the basis of actual run times of the resulting red-black tree algorithms.

In our discussion of the insert and delete operations, we will explicitly state the needed color changes only for the nodes. The corresponding pointer color changes may be inferred.

10.3.3 Searching a Red-Black Tree

We can search a red-black tree with the code we used to search an ordinary binary search tree (Program 5.17). This code has complexity $O(h)$, which is $O(\log n)$ for a red-black tree. Since we use the same code to search ordinary binary search trees, AVL trees, and red-black trees and since the worst-case height of an AVL tree is least, we expect AVL trees to show the best worst-case performance in applications where search is the dominant operation.

10.3.4 Inserting into a Red-Black Tree

Elements may be inserted using the strategy used for ordinary binary trees (Program 5.21). When the new node is attached to the red-black tree, we need to assign the node a color. If the tree was empty before the insertion, then the new node is the root and must be colored black (see property RB1). Suppose the tree was not empty prior to the insertion. If the new node is given the color black, then we will have an extra black node on paths from the root to the external nodes that are children of the new node. On the other hand, if the new node is assigned the color red, then we might have two consecutive red nodes. Making the new node black is guaranteed to cause a violation of property RB3, while making the new node red may or may not violate property RB2. We will make the

new node red.

If making the new node red causes a violation of property RB2, we will say that the tree has become imbalanced. The nature of the imbalance is classified by examining the new node u, its parent pu, and the grandparent gu of u. Observe that since property RB2 has been violated, we have two consecutive red nodes. One of these red nodes is u, and the other must be its parent; therefore, pu exists. Since pu is red, it cannot be the root (as the root is black by property RB1); u must have a grandparent gu, which must be black (property RB2). When pu is the left child of gu, u is the left child of pu and the other child of gu is black (this case includes the case when the other child of gu is an external node); the imbalance is of type LLb. The other imbalance types are LLr (pu is the left child of gu, u is the left child of pu, the other child of gu is red), LRb (pu is the left child of gu, u is the right child of pu, the other child of gu is black), LRr, RRb, RRr, RLb, and RLr.

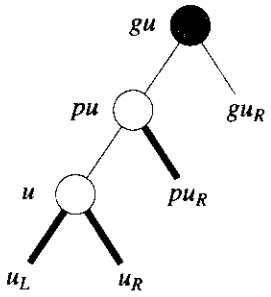
Imbalances of the type XYr (X and Y may be L or R) are handled by changing colors, while those of type XYb require a rotation. When we change a color, the RB2 violation may propagate two levels up the tree. In this case we will need to reclassify at the new level, with the new u being the former gu, and apply the transformations again. When a rotation is done, the RB2 violation is taken care of and no further work is needed.

Figure 10.16 shows the color changes performed for LLr and LRr imbalances; these color changes are identical. Black nodes are shaded, while red ones are not. In Figure 10.16(a), for example, gu is black, while pu and u are red; the pointers from gu to its left and right children are red; gu_R is the right subtree of gu; and pu_R is the right subtree of pu. Both LLr and LRr color changes require us to change the color of pu and of the right child of gu from red to black. Additionally, we change the color of gu from black to red provided gu is not the root. Since this color change is not done when gu is the root, the number of black nodes on all root-to-external-node paths increases by 1 when gu is the root of the red-black tree.

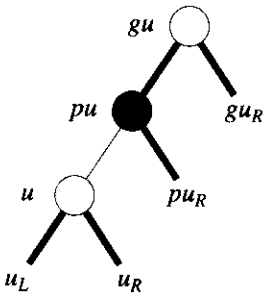
If changing the color of gu to red causes an imbalance, gu becomes the new u node, its parent becomes the new pu, its grandparent becomes the new gu, and we continue to rebalance. If gu is the root or if the color change does not cause an RB2 violation at gu, we are done.

Figure 10.17 shows the rotations performed to handle LLb and LRb imbalances. In Figures 10.17(a) and (b), u is the root of pu_L. Notice the similarity between these rotations and the LL (refer to Figure 10.12) and LR (refer to Figure 10.13) rotations used to handle an imbalance following an insertion in an AVL tree. The pointer changes are the same. In the case of an LLb rotation, for example, in addition to pointer changes we need to change the color of gu from black to red and of pu from red to black.

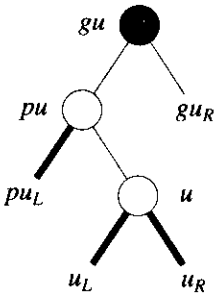
In examining the node (or pointer) colors after the rotations of Figure 10.17, we see that the number of black nodes (or pointers) on all root-to-external-node paths is unchanged. Further, the root of the involved subtree (gu before the rotation and pu after) is black following the rotation; therefore, two consecutive red nodes cannot exist on the



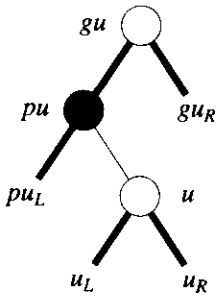
(a) LLr imbalance



(b) After LLr color change



(c) LRr imbalance

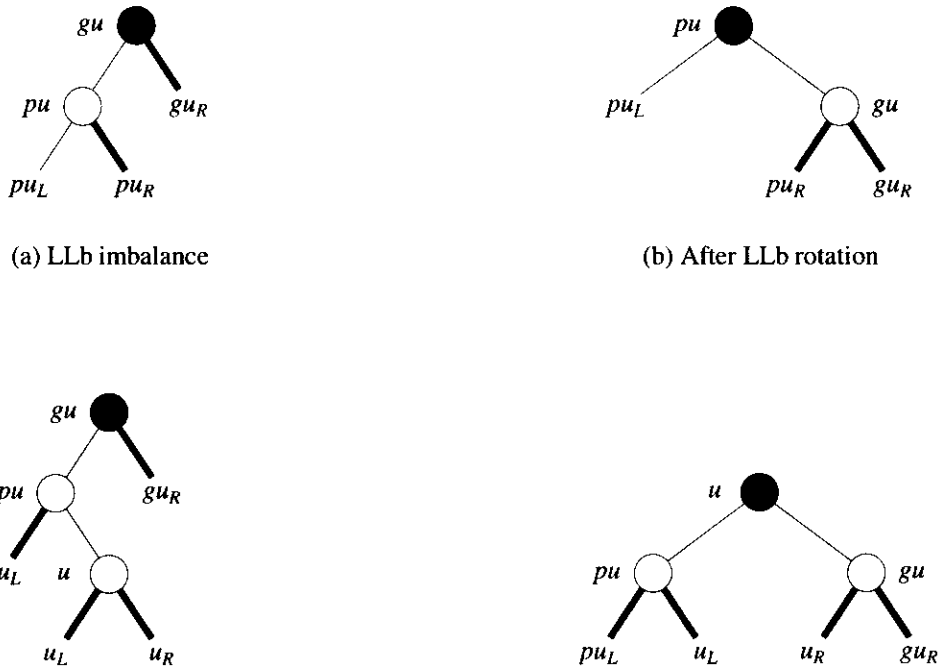


(d) After LRr color change

Figure 10.16: LLr and LRr color changes

path from the tree root to the new *pu*. Consequently, no additional rebalancing work is to be done. *A single rotation (preceded possibly by* O(log *n) color changes) suffices to restore balance following an insertion!*

Example 10.4: Consider the red-black tree of Figure 10.18(a). External nodes are shown for convenience. In an actual implementation, the shown black pointers to external nodes are simply null pointers and external nodes are not represented. Notice that all root-to-external-node paths have three black nodes (including the external node) and two black pointers.



(a) LLb imbalance

(b) After LLb rotation

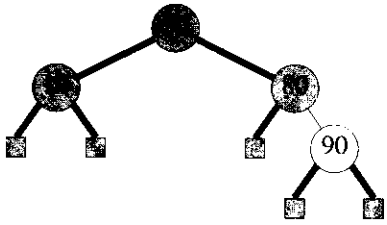
(c) LRb imbalance

(d) After LRb rotation

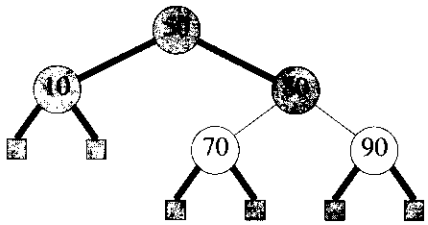
Figure 10.17: LLb and LRb rotations for red-black insertion

To insert 70 into this red-black tree, we use the algorithm of Program 5.18. The new node is added as the left child of 80. Since the insertion is done into a nonempty tree, the new node is assigned the color red. So the pointer to it from its parent (80) is also red. This insertion does not result in a violation of property RB2, and no remedial action is necessary. Notice that the number of black pointers on all root-to-external-node paths is the same as before the insertion.

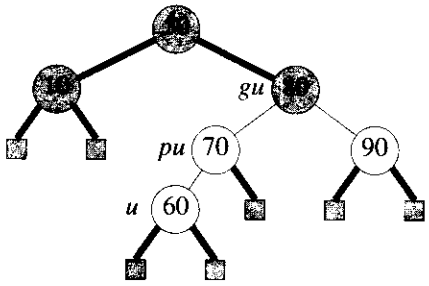
Next insert 60 into the tree of Figure 10.18(b). The algorithm of Program 5.18 attaches a new node as the left child of 70, as is shown in Figure 10.18(c). The new node is red, and the pointer to it is also red. The new node is the *u* node, its parent (70) is *pu*, and its grandparent (80) is *gu*. Since *pu* and *u* are red, we have an imbalance. This imbalance is classified as an LLr imbalance (as *pu* is the left child of *gu*, *u* is the left child of *pu*, and the other child of *gu* is red). When the LLr color change of Figure 10.16(a) and (b) is performed, we get the tree of Figure 10.18(d). Now *u*, *pu*, and *gu* are



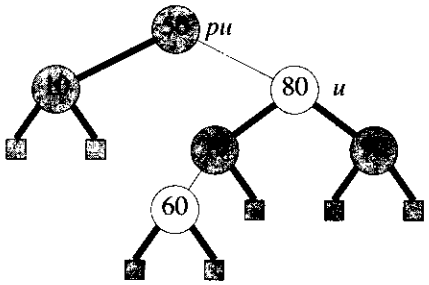
(a) Initial



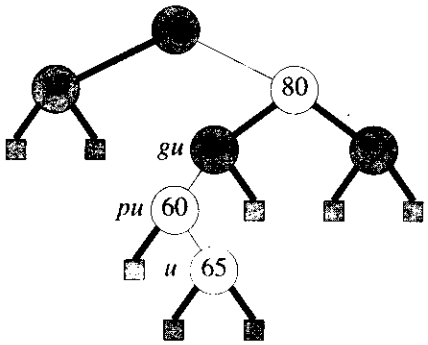
(b) Insert 70



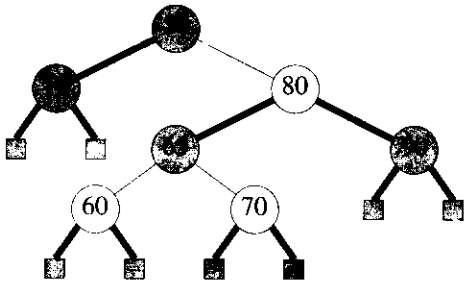
(c) Insert 60



(d) LLr color change



(e) Insert 65

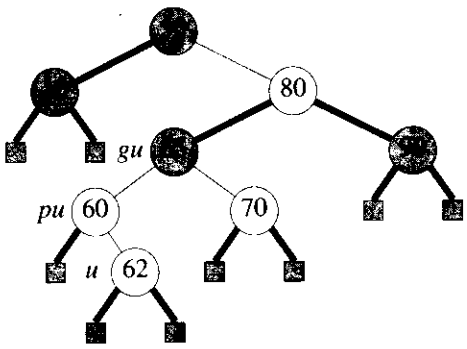


(f) LRb rotation

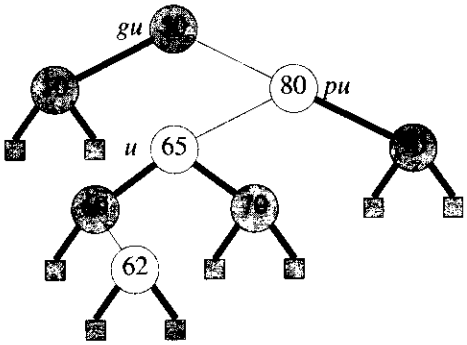
Figure 10.18: Insertion into a red-black tree (continued on next page)

each moved two levels up the tree. The node with 80 is the new *u* node, the root becomes *pu*, and *gu* is **NULL**. Since there is no *gu* node, we cannot have an RB2 imbalance at this location and we are done. All root-to-external-node paths have exactly two black pointers.

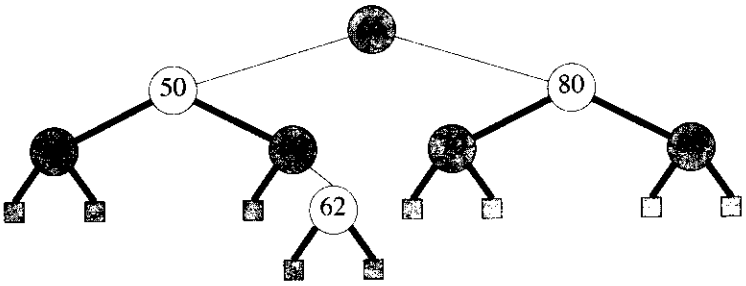
Now insert 65 into the tree of Figure 10.18(d). The result appears in Figure 10.18(e). The new node is the *u* node. Its parent and grandparent are, respectively, the *pu* and *gu* nodes. We have an LRb imbalance that requires us to perform the rotation of Figures 10.17(c) and (d). The result is the tree of Figure 10.18(f).



(g) Insert 62



(h) LRr color change



(i) RLb rotation

Figure 10.18: Insertion into a red-black tree

Finally, insert 62 to obtain the tree of Figure 10.18(g). We have an LRr imbalance that requires a color change. The resulting tree and the new *u*, *pu*, and *gu* nodes appear

in Figure 10.18(h). The color change just performed has caused an RLb imbalance two levels up, so we now need to perform an RLb rotation. The rotation results in the tree of Figure 10.18(i). Following a rotation, no further work is needed, and we are done. □

10.3.5 Deletion from a Red-Black Tree

The development of the deletion transformations is left as an exercise.

10.3.6 Joining Red-Black Trees

In Section 5.7.5, we defined the following operations on binary search trees: *threeWayJoin*, *twoWayJoin*, and *split*. Each of these can be performed in logarithmic time on red-black trees. The operation *threeWayJoin* (A, x, B) (A corresponds to *small*, x to *mid*, and B to *big*) can be performed as follows.

Case 1: If A and B have the same rank, then let C be constructed by creating a new root with pair x, *leftChild* A, and *rightChild* B. Both links are made black. The rank of C is one more than the ranks of A and B.

Case 2: If $rank(A) > rank(B)$, then follow *rightChild* pointers from A to the first node Y that has rank equal to $rank(B)$. Properties RB1 to RB3 guarantee the existence of such a node. Let $p(Y)$ be the parent of Y. From the definition of Y, it follows that $rank(p(Y)) = rank(Y) + 1$. Hence, the pointer from $p(Y)$ to Y is a black pointer. Create a new node, Z, with pair x, *leftChild* Y (i.e., node Y and its subtrees become the left subtree of Z) and *rightChild* B. Z is made the right child of $p(Y)$, and the link from $p(Y)$ to Z has color red. The links from Z to its children are made black. Note that this transformation does not change the number of black pointers on any root-to-external-node path. However, it may cause the path from the root to Z to contain two consecutive red pointers. If this happens, then the transformations used to handle this in a bottom-up insertion are performed. These transformations may increase the rank of the tree by one.

Case 3: The case $rank(A) < rank(B)$ is similar to Case 2.

Analysis of threeWayJoin: The correctness of the function just described is easily established. Case 1 takes O(1) time; each of the remaining two cases takes $O(|rank(A) - rank(B)|)$ time under the assumption that the rank of each red-black tree is known prior to computing the join. Hence, a three-way join can be done in $O(\log n)$ time, where n is the number of nodes in the two trees being joined. A two-way join can be performed in a similar manner. Note that there is no need to add parent data members to the nodes to perform a join, as the needed parents can be saved on a stack as we move from the root to the node Y. □

A two-way join may be done in a similar fashion.

10.3.7 Splitting a Red-Black Tree

We now turn our attention to the split operation. Assume for simplicity that the splitting key, i, is actually present in the red-black tree A. Under this assumption, the split operation *split* (A, i, B, x, C) (see Section 5.7.5, A corresponds to *theTree*, i to k, B to *small*, x to *mid*, and C to *big*) can be performed as in Program 10.4.

Step 1: Search A for the node P that contains the element with key i. Copy this element to the parameter *mid*. Initialize B and C to be the left and right subtrees of P, respectively.

Step 2:

```
for (Q = parent(P); Q; P = Q, Q = parent(Q))
   if (P == Q→leftChild)
      C = threeWayJoin(C, Q→data, Q→rightChild)
   else B = threeWayJoin(Q→leftChild, Q→data, B);
}
```

Program 10.4: Splitting a red-black tree

We first locate the splitting element, x, in the red-black tree. Let P be the node that contains this element. The left subtree of P contains elements with key less than i. B is initialized to be this subtree. All elements in the right subtree of P have a key larger than i, and C is initialized to be this subtree. In Step 2, we trace the path from P to the root of the red-black tree A. During this traceback, two kinds of subtrees are encountered. One of these contains elements with keys that are larger than i as well as all keys in C. This happens when the traceback moves from a left child of a node to its parent. The other kind of subtree contains elements with keys that are smaller than i, as well as smaller than all keys in B. This happens when we move from a right child to its parent. In the former case a three-way join with C is performed, and in the latter a three-way join with B is performed. One may verify that the two-step procedure outlined here does indeed implement the split operation for the case when i is the key of an element in the tree A. It is easily extended to handle the case when the tree A contains no element with key i.

Analysis of split: Call a node red if the pointer to it from its parent is red. The root and all nodes that have a black pointer from their parent are black. Let $r(X)$ be the rank of node X in the unsplit tree. First, we shall show that during a split, if P is a black node in the unsplit tree, and $Q \neq 0$, then

$$r(Q) \geq \max\{r(B), r(C)\}$$

where P, Q, B, and C are as defined at the start of an iteration of the **for** loop in Step 2.

From the definition of rank, the inequality holds at the start of the first iteration of the **for** loop regardless of the color of P. If P is red initially, then its parent, Q, exists and is black. Let q' be the parent of Q. If $q' = 0$, then there is no Q at which the inequality is violated. So, assume $q' \neq 0$. From the definition of rank and the fact that Q is black, it follows that $r(q') = r(Q) + 1$. Let B' and C' be the trees B and C following the three-way join of Step 2. Since $r(B') \leq r(B) + 1$ and $r(C') \leq r(C) + 1$, $r(q') = r(Q) + 1 \geq \max\{r(B), r(C)\} + 1 \geq \max\{r(B'), r(C')\}$. So, the inequality holds the first time Q points to a node with a black child P (i.e., at the start of the second iteration of the **for** loop, when $Q = q'$).

Having established the induction base, we can proceed to show that the inequality holds at all subsequent iterations when Q points to a node with a black child P. Suppose Q is currently pointing to a node q with a black child $P = p$. Assume that the inequality holds. We shall show that it will hold again the next time Q is at a node with a black P. For there to be such a next time, the parent q' of q must exist. If q is black, the proof is similar to that provided for a black Q and a red P in the induction base.

If q is red, then q' is black. Further, for there to be a next time when Q is at a node with a black P, q must have a grandparent q'', as when Q moves to q' and P to q, $Q = q'$ has a red child $P = q$. Let B' and C' represent the B and C trees following the iteration that begins with $P = p$ and $Q = q$. Similarly, let B'' and C'' represent these trees following the iteration that begins with $P = q$ and $Q = q'$.

Suppose that C is joined with q and its right subtree R to create C'. If $r(C) = r(R)$, then $r(C') = r(C) + 1$, and C' has two black children (recall that when the rank increases by one, the root has two black children). If $C'' = C'$, then $B = B'$ is combined with q' and its left subtree L' to form B''. Since $r(L') \leq r(q')$, $r(B'') \leq \max\{r(B), r(L')\} + 1$, and $r(q'') = r(q') + 1 = r(q) + 1$, $r(B'') \leq r(q'')$. Also, $r(C'') = r(C') = r(C) + 1 \leq r(q) + 1 \leq r(q'')$. So, the inequality holds when $Q = q''$. If $C'' \neq C'$, then C' is combined with q' and its right subtree R' to form C''. If $r(R') \geq r(C')$, then $r(C'') \leq r(R') + 1 \leq r(q') + 1 = r(q'')$. If $r(R') < r(C')$, then $r(C'') = r(C')$, as C' has two black children, and the join of C', q', and R' does not increase the rank. Once again, $r(C'') \leq r(q'')$, and the inequality holds when $Q = q''$.

If $r(C) > r(R)$ and $r(C') = r(C)$, then $r(q'') = r(q) + 1 \geq \max\{r(B), r(C)\} + 1 \geq \max\{r(B''), r(C'')\}$. If $r(C') = r(C) + 1$, then C' has two black children, and $r(C'') \leq r(q) + 1 = r(q'')$. Also, $r(B'') \leq r(q'')$. So, the inequality holds when $Q = q''$. The case $r(C) < r(R)$ is similar.

The case when B is joined with q and its left subtree L is symmetric.

Using the rank inequality just established, we can show that whenever Q points to a node with a black child, the total work done in Step 2 of the splitting algorithm from initiation to the time Q reaches this node is $O(r(B) + r(C) + r(Q))$. Here, B and C are, respectively, the current red-black trees with values smaller and larger than the splitting value. Since $r(Q) \geq \max\{r(B), r(C)\}$, the total work done in Step 2 is $O(r(Q))$. From

this, it follows that the time required to perform a split is O(log n), where n is the number of nodes in the tree to be split. □

EXERCISES

1. Start with an empty red-black tree and insert the following keys in the given order: 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1. Draw figures similar to Figure 10.18 depicting your tree immediately after each insertion and following the rebalancing rotation or color change (if any). Label all nodes with their color and identify the rotation type (if any) that is done.
2. Do Exercise 1 using the insert key sequence: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15.
3. Do Exercise 1 using the insert key sequence: 20, 10, 5, 30, 40, 57, 3, 2, 4, 35, 25, 18, 22, 21.
4. Do Exercise 1 using the insert key sequence: 40, 50, 70, 30, 42, 15, 20, 25, 27, 26, 60, 55.
5. Draw the RRr and RLr color changes that correspond to the LLr and LRr changes of Figure 10.16.
6. Draw the RRb and RLb rotations that correspond to the LLb and LRb changes of Figure 10.17.
7. Let T be a red-black tree with rank r. Write a function to compute the rank of each node in the tree. The time complexity of your function should be linear in the number of nodes in the tree. Show that this is the case.
8. Compare the worst-case height of a red-black tree with n nodes and that of an AVL tree with the same number of nodes.
9. Develop the deletion transformations for a red-black tree. Show that a deletion from a red-black tree requires at most one rotation.
10. (a) Use the strategy described in this section to obtain a C function to compute a three-way join. Assume the existence of a function *rebalance* (X) that performs the necessary transformations if the tree pointer to node X is the second of two consecutive red pointers. The complexity of this function may be assumed to be O(*level* (X)).
    (b) Prove the correctness of your function.
    (c) What is the time complexity of your function?
11. Obtain a function to perform a two-way join on red-black trees. You may assume the existence of functions to search, insert, delete, and perform a three-way join. What is the time complexity of your two-way join function?

12. Use the strategy suggested in Program 10.4 to obtain a C function to perform the split operation in a red-black tree T. The complexity of your algorithm must be $O(height(T))$. Your function must work when the splitting key i is present in T and when it is not present in T.

13. Complete the complexity proof for the split operation by showing that whenever Q has a black child, the total work done in Step 2 of the splitting algorithm from initiation to the time that Q reaches the current node is $O(r(Q))$.

14. Program the search, insert, and delete operations for AVL trees and red-black trees.

    (a) Test the correctness of your functions.

    (b) Generate a random sequence of n inserts of distinct values. Use this sequence to initialize each of the data structures. Next, generate a random sequence of searches, inserts, and deletes. In this sequence, the probability of a search should be 0.5, that of an insert 0.25, and that of a delete 0.25. The sequence length is m. Measure the time needed to perform the m operations in the sequence using each of the above data structures.

    (c) Do part (b) for $n = 100$, 1000, 10,000, and 100,000 and $m = n$, $2n$, and $4n$.

    (d) What can you say about the relative performance of these data structures?

10.4 SPLAY TREES

We have studied balanced search trees that allow one to perform operations such as search, insert, delete, join, and split in $O(\log n)$ worst-case time per operation. In the case of priority queues, we saw that if we are interested in amortized complexity rather than worst-case complexity, simpler structures can be used. This is also true for search trees. Using a splay tree, we can perform the operations in $O(\log n)$ amortized time per operation. In this section, we develop two varieties of splay trees—bottom up and top down. Although the amortized complexity of each operation is $O(\log n)$ for both varieties, experiments indicate that top-down splay trees are faster than bottom-up splay trees by a constant factor.

10.4.1 Bottom-Up Splay Trees

A *bottom-up splay tree* is a binary search tree in which each search, insert, delete, and join operation is performed in the same way as in an ordinary binary search tree (see Chapter 5) except that each of these operations is followed by a *splay*. In a split, however, we *first* perform a splay. This makes the split very easy to perform. A splay consists of a sequence of rotations. For simplicity, we assume that each of the operations is

always successful. A failure can be modeled as a different successful operation. For example, an unsuccesful search may be modeled as a search for the element in the last node encountered in the unsuccessful search, and an unsuccessful insert may be modeled as a successful search. With this assumption, the start node for a splay is obtained as follows:

(1) *search*: The splay starts at the node containing the element being sought.

(2) *insert*: The start node for the splay is the newly inserted node.

(3) *delete*: The parent of the physically deleted node is used as the start node for the splay. If this node is the root, then no splay is done.

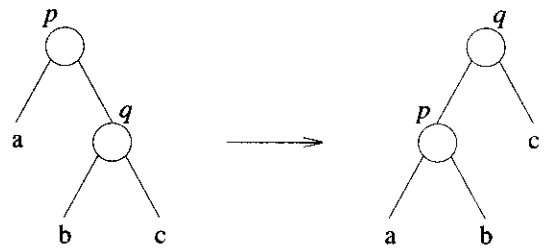
(4) *threeWayJoin*: No splay is done.

(5) *split*: Suppose that we are splitting with respect to the key i and that key i is actually present in the tree. We first perform a splay at the node that contains i and then split the tree. As we shall see, splitting following a splay is very simple.

Splay rotations are performed along the path from the start node to the root of the binary search tree. These rotations are similar to those performed for AVL trees and red-black trees. Let q be the node at which the splay is being performed. Initially, q is the node at which the splay starts. The following steps define a splay:

(1) If q is either 0 or the root, then the splay terminates.

(2) If q has a parent, p, but no grandparent, then the rotation of Figure 10.19 is performed, and the splay terminates.



a, b, and c are subtrees

Figure 10.19: Rotation when q is a right child and has no grandparent

(3) If q has a parent, p, and a grandparent, gp, then the rotation is classified as LL (p is the left child of gp, and q is the left child of p), LR (p is the left child of gp, and q

is the right child of p), RR, or RL. The RR and RL rotations are shown in Figure 10.20. LL and LR rotations are symmetric to these. The splay is repeated at the new location of q.

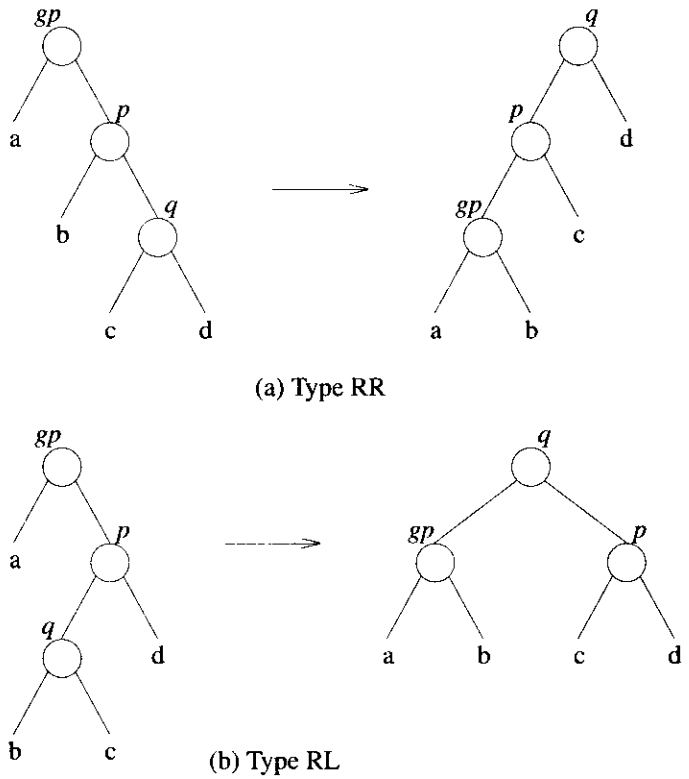
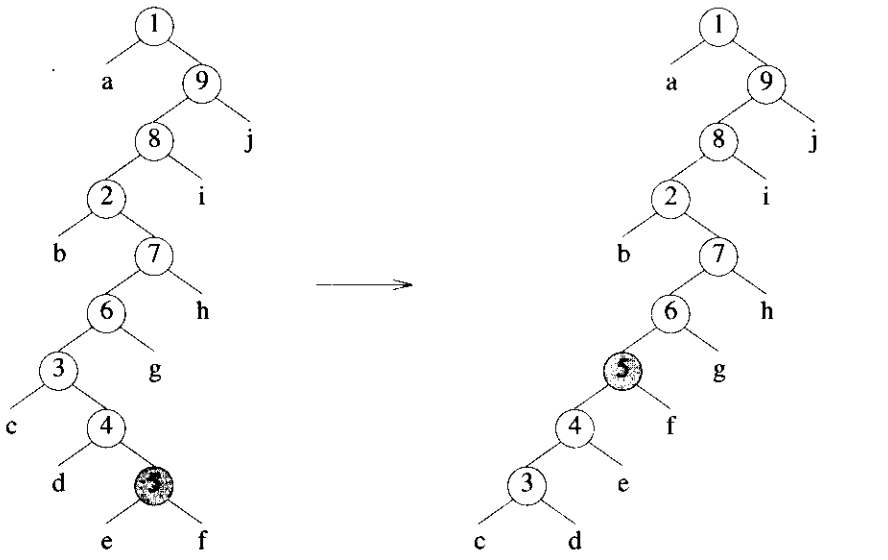


Figure 10.20: RR and RL rotations

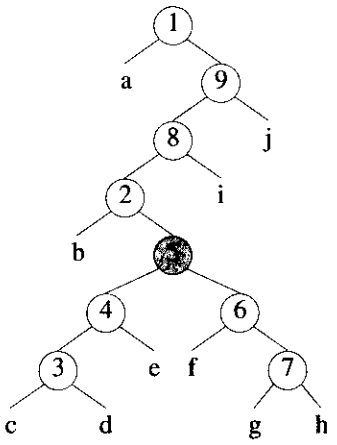
Notice that all rotations move q up the tree and that following a splay, q becomes the new root of the search tree. As a result, splitting the tree with respect to a key, i, is done simply by performing a splay at i and then splitting at the root. Figure 10.21 shows a binary search tree before, during, and after a splay at the shaded node.

In the case of Fibonacci heaps, we obtained the amortized complexity of an operation by using an explicit cross-charging scheme. The analysis for splay trees will use a *potential* technique. Let P_0 be the initial potential of the search tree, and let P_i be its potential following the ith operation in a sequence of m operations. The amortized time

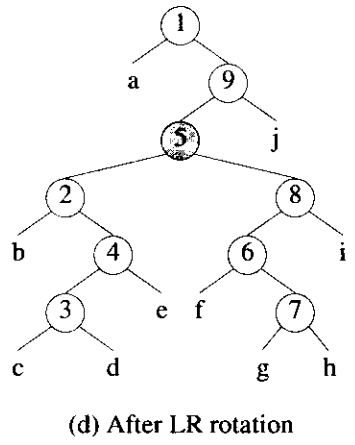


(a) Initial search tree

(b) After RR rotation

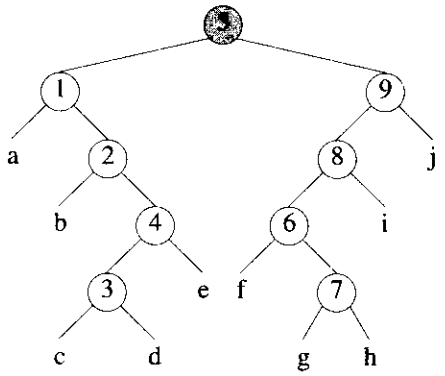


(c) After LL rotation



(d) After LR rotation

Figure 10.21: Rotations in a splay beginning at shaded node (continued on next page)



(e) After RL rotation

Figure 10.21: Rotations in a splay beginning at the shaded node

for the ith operation is defined to be

$$\text{(actual time for the } i\text{th operation)} + P_i - P_{i-1}$$

That is, the amortized time is the actual time plus the change in the potential. Rearranging terms, we see that the actual time for the ith operation is

$$\text{(amortized time for the } i\text{th operation)} + P_{i-1} - P_i$$

Hence, the actual time needed to perform the m operations in the sequence is

$$\sum_i \text{(amortized time for the } i\text{th operation)} + P_0 - P_m$$

Since each operation other than a join involves a splay whose actual complexity is of the same order as that of the whole operation, and since each join takes O(1) time, it is sufficient to consider only the time spent performing splays.

Each splay consists of several rotations. We shall assign to each rotation a fixed cost of one unit. The choice of a potential function is rather arbitrary. The objective is to use one that results in as small a bound on the time complexity as is possible. We now define the potential function we shall use. Let the size, $s(i)$, of the subtree with root i be the total number of nodes in it. The rank, $r(i)$, of node i is equal to $\lfloor \log_2 s(i) \rfloor$. The potential of the tree is $\sum_i r(i)$. The potential of an empty tree is defined to be zero.

Suppose that in the tree of Figure 10.21(a), the subtrees a, b, $\cdots$, j are all empty. Then $(s(1), \cdots, s(9)) = (9, 6, 3, 2, 1, 4, 5, 7, 8)$; $r(3) = r(4) = 1$; $r(5) = 0$; and $r(9) = 3$. In Lemma 10.3 we use r and r', respectively, to denote the rank of a node before and

after a rotation.

Lemma 10.3: Consider a binary search tree that has n elements/nodes. The amortized cost of a splay operation that begins at node q is at most $3(\lfloor \log_2 n \rfloor - r(q)) + 1$.

Proof: Consider the three steps in the definition of a splay:

(1) In this case, q either is 0 or the root. This step does not alter the potential of the tree, so its amortized and actual costs are the same. This cost is 1.

(2) In this step, the rotation of Figure 10.19 (or the symmetric rotation for the case when q is the left child of p) is performed. Since only the ranks of p and q are affected, the potential change, ΔP, is $r'(p) + r'(q) - r(p) - r(q)$. Further, since $r'(p) \leq r(p)$, $\Delta P \leq r'(q) - r(q)$. The amortized cost of this step (actual cost plus potential change) is, therefore, no more than $r'(q) - r(q) + 1$.

(3) In this step only the ranks of q, p, and gp change. So, $\Delta P = r'(q) + r'(p) + r'(gp) - r(q) - r(p) - r(gp)$. Since, $r(gp) = r'(q)$,

$$\Delta P = r'(p) + r'(gp) - r(q) - r(p) \quad \text{-- (1)}$$

Consider an RR rotation. From Figure 10.20(a), we see that $r'(p) \leq r'(q)$, $r'(gp) \leq r'(q)$, and $r(q) \leq r(p)$. So, $\Delta P \leq 2(r'(q) - r(q))$. If $r'(q) > r(q)$, $\Delta P \leq 3(r'(q) - r(q)) - 1$. If $r'(q) = r(q)$, then $r'(q) = r(q) = r(p) = r(gp)$. Also, $s'(q) > s(q) + s'(gp)$. Consequently, $r'(gp) < r'(q)$. To see this, note that if $r'(gp) = r'(q)$, then $s'(q) > 2^{r(q)} + 2^{r'(gp)} = 2^{r(q)+1}$, which violates the definition of rank. Hence, from (1), $\Delta P \leq 2(r'(q) - r(q)) - 1 = 3(r'(q) - r(q)) - 1$. So, the amortized cost of an RR rotation is at most $1+3(r'(q) - r(q)) - 1 = 3(r'(q) - r(q))$.

This bound may be obtained for LL, LR, and RL rotations in a similar way.

The lemma now follows by observing that Steps 1 and 2 are mutually exclusive and can occur at most once. Step 3 occurs zero or more times. Summing up over the amortized cost of a single occurrence of Steps 1 or 2 and all occurrences of Step 3, we obtain the bound of the lemma. □

Theorem 10.1: The total time needed for a sequence of m search, insert, delete, join, and split operations performed on a collection of initially empty splay trees is $O(m \log n)$, where n, $n > 0$, is the number of inserts in the sequence.

Proof: Since none of the splay trees has more than n nodes, no node has rank more than $\lfloor \log_2 n \rfloor$. A search (excluding the splay) does not change the rank of any node and hence does not affect the potential of the splay tree involved. An insert (excluding the splay) increases, by one, the size of every node on the path from the root to the newly inserted node. This causes the ranks of the nodes with size $2^k - 1$ to change. There are at most $\lfloor \log_2 n \rfloor + 1$ such nodes on any insert path. So, each insert (excluding the splay) increases the potential by at most this much. Each join increases the total potential of all

the splay trees by at most $\lfloor \log_2 n \rfloor$. Deletions do not increase the potential of the involved splay tree except for any increase that results from the splay step. The split operation (excluding the splay step) reduces the overall potential by an amount equal to the rank of the tree just before the split (but after the splay that precedes it). So, the potential increase, PI, attributable to the m operations (exclusive of that attributable to the splay steps of the operations) is $O(m \log n)$.

From our definition of the amortized cost of a splay operation, it follows that the time for the sequence of operations is the sum of the amortized costs of the splays, the potential change $P_0 - P_m$, and PI. From Lemma 10.3, it follows that the sum of the amortized costs is $O(m \log n)$. The initial potential, P_0, is 0, and the final potential, P_m, is ≥ 0. So, the total time is $O(m \log n)$. □

10.4.2 Top-Down Splay Trees

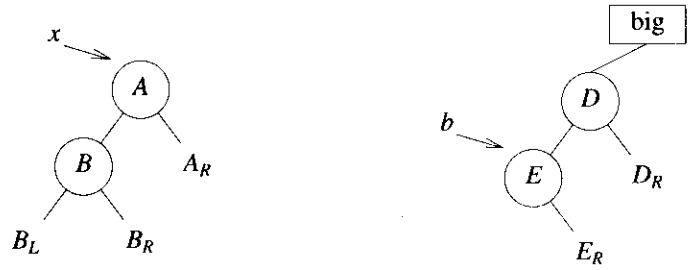
As in the case of a bottom-up splay tree, a *threeWayJoin* is implemented the same in a top-down splay tree as in Section 5.7.5. For the remaining operations, let the splay node be as defined for a bottom-up splay tree. For each operation, we follow a path from the root to the splay node as in Section 5.7.5. However, as we follow this path, we partition the binary search tree into three components—a binary search tree *small* of elements whose key is less than that of the element in the splay node, a binary search tree *big* of elements whose key is greater than that of the element in the splay node, and the splay node. Notice that in this downward traversal of the path from the root to the splay node, we do not actually know which node is the splay node until we get to it. So, the downward traversal is done using the key k associated with the operation that is being performed.

For the partitioning, we begin with two empty binary search trees *small* and *big*. It is convenient to give these trees a header node that is deleted at the end. Let s and b, respectively, be initialized to the header nodes of *small* and *big*. The downward traversal to the splay node begins at the root. Let x denote the node we currently are at. We begin with x being the tree root. There are 7 cases to consider:

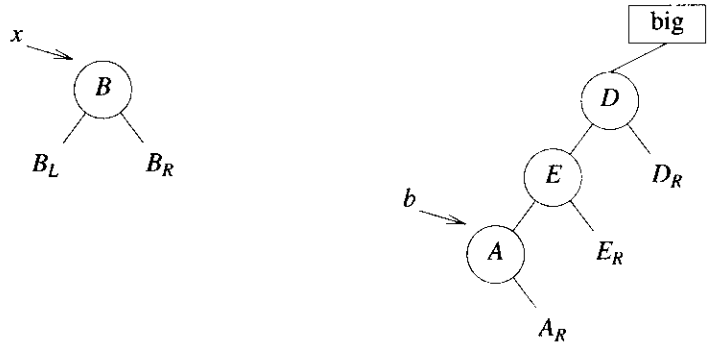
Case 0: *x is the splay node.*
Terminate the partitioning process.

Case L: *The splay node is the left child of x.*
In this case, x and its right subtree contain keys that are greater than that in the splay node. So, we make x the left child of b ($b \rightarrow leftChild = x$) and set $b = x$ and $x = x \rightarrow leftChild$. Notice that this automatically places the right subtree of x into *big*. Figure 10.22 shows a schematic for this case.

Case R: *The splay node is the right child of x.*
This case is symmetric to Case L. Now, x and its left subtree contain keys that



(a) Before L transformation



(b) After L transformation

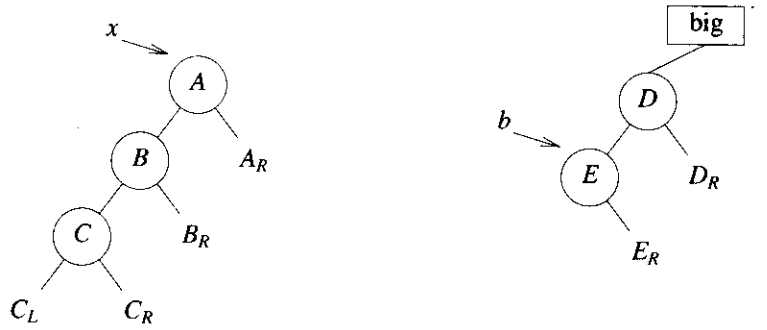
Figure 10.22: Case L for a top-down splay tree

are less than that in the splay node. So, we make x the right child of s ($s \rightarrow rightChild = x$) and set $s = x$ and $x = x \rightarrow rightChild$. Notice that this automatically places the left subtree of x into *small*.

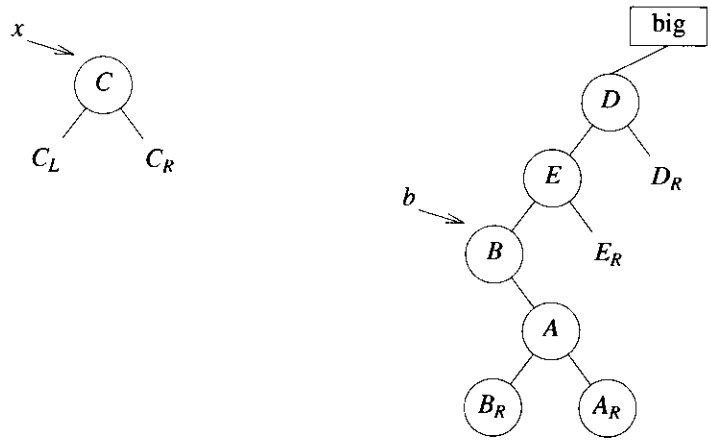
Case LR: *The splay node is in the right subtree of the left child of x.*
This case is handled as Case L followed by Case R.

Case RL: *The splay node is in the left subtree of the right child of x.*
This case is handled as Case R followed by Case L.

Case LL: *The splay node is in the left subtree of the left child of x.*
This case is *not* handled as two applications of Case L. Instead we perform an LL rotation around x. Figure 10.23 shows a schematic for this case. The shown transformation is accomplished by the following code fragment:



(a) Before LL transformation



(b) After LL transformation

Figure 10.23: Case LL for a top-down splay tree

```
b->leftChild  = x->leftChild;
b = b->leftChild;
x->leftChild = b->rightChild;
b->rightChild = x;
x  = b->leftChild;
```

Case RR: *The splay node is in the right subtree of the right child of x.*
This case is symmetric to Case LL.

The above transformations are applied repeatedly until terminated by an application of Case 0. Upon termination, x is the splay node. Now, the left subtree of x is made the right subtree of s and the right subtree of x is made the left subtree of b. Finally, the header nodes of the small and big trees are deleted.

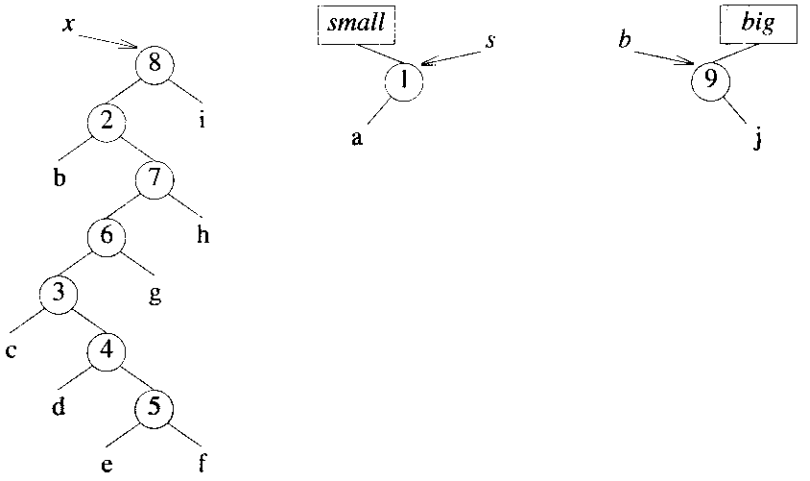
In case we were performing a split operation and x contains the split key, we return *small*, $x \rightarrow data$, and *big* as the result of the split. For a search, insert and delete, we make *small* and *big*, respectively, the left and right subtrees of x and the tree rooted at x is the new binary search tree (we assume that in the downward quest for the splay node, the remaining tasks associated with the search, insert and delete operations have been done).

Example 10.5: Suppose we are searching for the key 5 in the top-down splay tree of Figure 10.21(a). Although we don't know this at this time, the splay node is the shaded node. The path from the root to the splay node is determined by comparing the search key 5 with the key in the current node. We start with the current node pointer x at the root and two empty splay trees—*small* and *big*. These empty splay trees have a header node. The variables s and b, respectively, point to these header nodes. Since the splay node is in the left subtree of the right child of x, an RL transformation is called for. The search tree as well as the trees *small* and *big* following the RL transformation are shown in Figure 10.24(a).

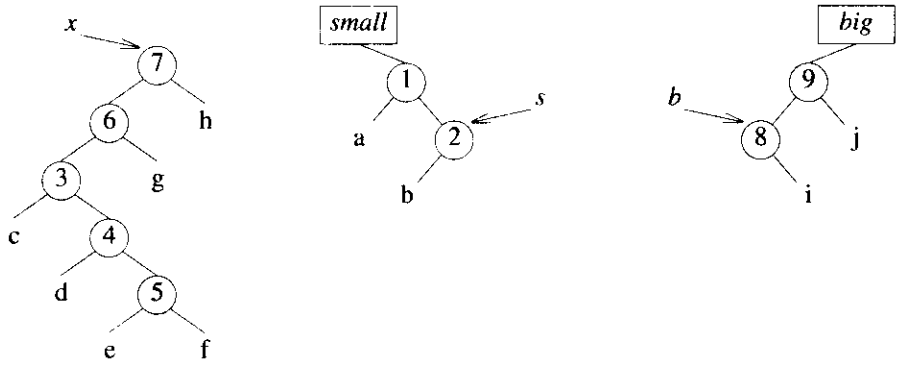
Now, since the splay node is in the right subtree of the left child of the new x, an LR transformation is made and we obtain the configuration of Figure 10.24(b). Next, we make an LL transformation (Figure 10.24(c)) and an RR transformation (Figure 10.24(d)). Now, x is at the splay node. The left subtree of x is made the right subtree of s and the right subtree of x is made the left subtree of b (Figure 10.24(e)). Finally, we delete the header nodes and make the small and big trees subtrees of x as shown in Figure 10.24(f). □

EXERCISES

1. Obtain figures corresponding to Figures 10.19 and 10.20 for the symmetric bottom-up splay tree rotations.
2. What is the maximum height of a bottom-up splay tree that is created as the result of n insertions made into an initially empty splay tree? Give an example of a sequence of inserts that results in a splay tree of this height.
3. Complete the proof of Lemma 10.3 by providing the proof for the case of an RL rotation. Note that the proofs for LL and LR rotations are similar to those for RR and RL rotations, respectively, as the rotations are symmetric.



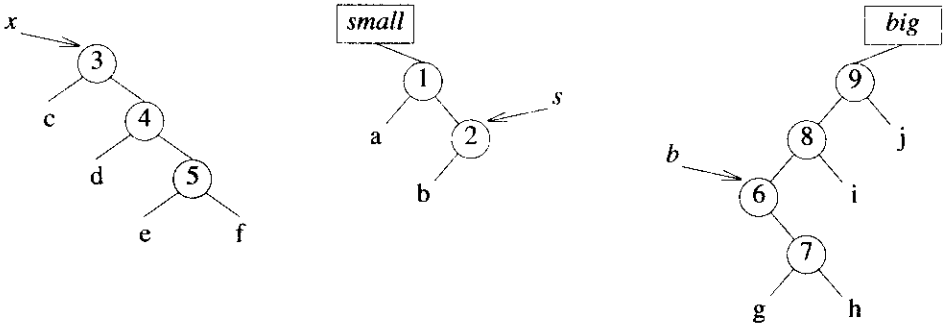
(a) After RL transformation



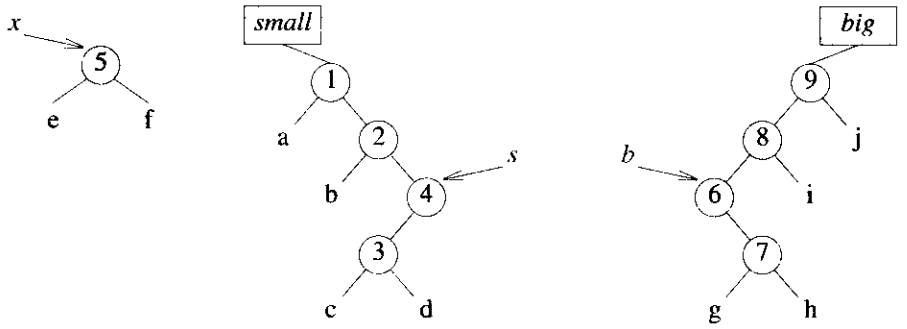
(b) After LR transformation

Figure 10.24: Example for top-down splay tree (continued on next page)

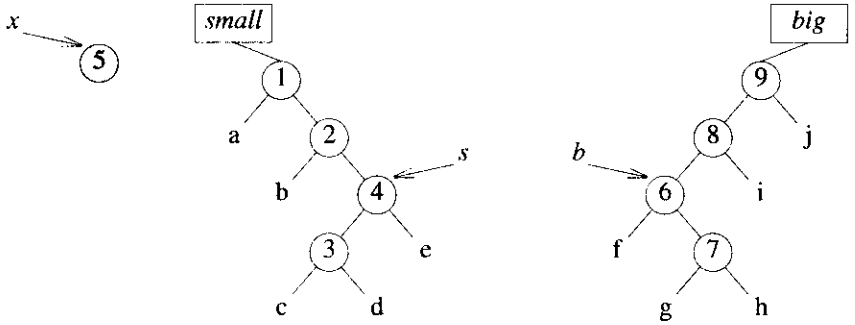
4. Explain how a two-way join should be performed in a bottom-up splay tree so that the amortized cost of each splay tree operation remains O(log n).
5. Explain how a split with respect to key i is to be performed when key i is not present in the bottom-up splay tree. The amortized cost of each bottom-up splay tree operation should be O(log n).



(c) After LL transformation

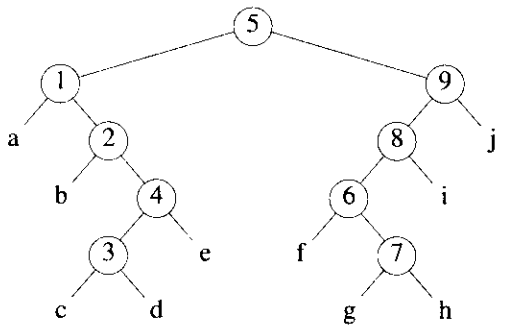


(d) After RR transformation



(e) After moving subtrees of splay node

Figure 10.24: Example for top-down splay tree (continued on next page)



(f) Final search tree

Figure 10.24: Example for top-down splay tree

6. Implement the bottom-up splay tree data structure. Test all functions using your own test data.
7. [***Sleator and Tarjan***] Suppose we modify the definition of $s(i)$ used in connection with the complexity analysis of bottom-up splay trees. Let each node i have a positive weight $p(i)$. Let $s(i)$ be the sum of the weights of all nodes in the subtree with root i. The rank of this subtree is $\log_2 s(i)$.
   (a) Let t be the root of a splay tree. Show that the amortized cost of a splay that begins at node q is at most $3(r(t) - r(q)) + 1$, where r is the rank just before the splay.
   (b) Let S be a sequence of n inserts and m searches. Assume that each of the n inserts adds a new element to the splay tree and that all searches are successful. Let $p(i)$, $p(i) > 0$, be the number of times element i is sought. The $p(i)$'s satisfy the following equality:

$$\sum_{i=1}^{n} p(i) = m$$

   Show that the total time spent on the m searches is

$$O(m + \sum_{i=1}^{n} p(i) \log(m/p(i)))$$

   Note that since $\Omega(m + \sum_{i=1}^{n} p(i) \log(m/p(i)))$ is an information theoretic bound on the search time in a static search tree (the optimal binary search

tree of Section 10.1 is an example of such a tree), bottom-up splay trees are optimal to within a constant factor for the representation of a static set of elements.

8. Obtain figures corresponding to Figures 10.22 and 10.23 for the top-down splay tree transformations R, RR, RL, and LR.
9. What is the maximum height of a top-down splay tree that is created as the result of n insertions made into an initially empty splay tree? Give an example of a sequence of inserts that results in a splay tree of this height.
10. Implement the top-down splay tree data structure. Test all functions using your own test data.

10.5 REFERENCES AND SELECTED READINGS

The $O(n^2)$ optimum binary search tree algorithm is from "Optimum binary search trees," by D. Knuth, *Acta Informatica*, 1:1, 1971, pp. 14-25. For a discussion of heuristics that obtain in $O(n \log n)$ time nearly optimal binary search trees, see "Nearly optimal binary search trees," by K. Mehlhorn, *Acta Informatica*, 5, 1975, pp. 287-295; and "Binary search trees and file organization," by J. Nievergelt, *ACM Computing Surveys*, 6:3, 1974, pp. 195-207.

The original paper on AVL trees by G. M. Adelson-Velskii and E. M. Landis appears in *Dokl. Acad. Nauk.*, SSR (Soviet Math), 3, 1962, pp. 1259-1263. Additional algorithms to manipulate AVL trees may be found in "Linear lists and priority queues as balanced binary trees," by C. Crane, Technical Report STAN-CS-72-259, Computer Science Dept., Stanford University, Palo Alto, CA, 1972; and *The Art of Computer Programming: Sorting and Searching* by D. Knuth, Addison-Wesley, Reading, MA, 1973 (Section 6.2.3).

Results of an empirical study of height-balanced trees appear in "Performance of height-balanced trees," by P. L. Karlton, S. H. Fuller, R. E. Scroggs, and E. B. Koehler, *CACM*, 19:1, 1976, pp. 23-28.

Splay trees were invented by D. Sleator and R. Tarjan. Their paper "Self-adjusting binary search trees," *JACM*, 32:3, 1985, pp. 652-686, provides several other analyses of splay trees, as well as variants of the basic splaying technique discussed in the text. Our analysis is modeled after that in *Data Structures and Network Algorithms*, by R. Tarjan, SIAM Publications, Philadelphia, PA, 1983.

For more on binary search trees, see Chapters 10 through 14 of "Handbook of data structures and applications," edited by D. Mehta and S. Sahni, Chapman & Hall/CRC, Boca Raton, 2005.

CHAPTER 11

Multiway Search Trees

11.1 m-WAY SEARCH TREES

11.1.1 Definition and Properties

Balanced binary search trees such as AVL and red-black trees allow us to search, insert, and delete in O(log n) time, where n is the number of elements. While this may seem to be a remarkable accomplishment, we can improve the performance of search structures by capitalizing on the exhorbitant time it takes to make a memory access (whether to main memory or to disk) relative to the time it takes to perform an arithmetic or logic operation in a modern computer. An access to main memory typically takes approximately 100 times the time to do an arithmetic operation while an access to disk takes about 10,000 times the time for an arithmetic operation. Because of this significant mismatch between processor speed and memory access time, data is typically moved from main memory to cache (fast memory) in units of a cache-line size (of the order of 100 bytes) and from disk to main memory in units of a block (several kilo bytes). For uniformity with disks, we say that main memory is organized into blocks; the size of each block being equal to that of a cache line. AVL and red-black trees are unable to

take advantage of this large unit (i.e., block) in which data is moved from slow memory (main or disk) to faster memory (cache or main) since the node size is typically only a few bytes. Consider an AVL tree with 1,000,000 elements. It's height may be as much as $\lfloor 1.44\log_2(n+2) \rfloor$, which is 28. To search this tree for an element with a specified key, we must access those nodes that are on the search path from the root to the node that contains the desired element. This path may contain 28 nodes and if each of these 28 nodes lies in a different memory block, a total of 28 memory accesses and 28 compares are made in the worst case. Most of the search time is spent on memory access! To improve performance, we must reduce the number of memory accesses. Notice that if halving the number of memory accesses resulted in a doubling of the number of comparisons, we would still achieve a reduction in total search time. Since the number of memory accesses is closely tied to the height of the search tree, we must reduce tree height. To break the $\log_2(n+1)$ barrier on tree height resulting from the use of binary search trees, we must use search trees whose degree is more than 2. In practice, we use the largest degree for which the tree node fits into a block (whether cache line or disk block).

Definition: An *m-way search tree* is either empty or satisfies the following properties:

(1) The root has at most m subtrees and has the following structure:

$$n, A_0, (E_1,A_1), (E_2,A_2), \cdots, (E_n,A_n)$$

where the A_i, $0 \le i \le n < m$, are pointers to subtrees, and the E_i, $1 \le i \le n < m$, are elements. Each element E_i has a key $E_i.K$.

(2) $E_i.K < E_{i+1}.K$, $1 \le i < n$.

(3) Let $E_0.K = -\infty$ and $E_{n+1}.K = \infty$. All keys in the subtree A_i are less than $E_{i+1}.K$ and greater than $E_i.K$, $0 \le i \le n$.

(4) The subtrees A_i, $0 \le i \le n$, are also m-way search trees. □

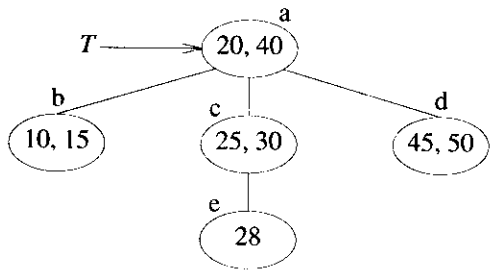
We may verify that binary search trees are two-way search trees. A three-way search tree is shown in Figure 11.1. For convenience, only keys are shown in this figure as well as in all remaining figures in this chapter.

In a tree of degree m and height h, the maximum number of nodes is

$$\sum_{0 \le i \le h-1} m^i = (m^h - 1)/(m - 1)$$

Since each node has at most $m - 1$ elements, the maximum number of elements in an m-way tree of height h is $m^h - 1$. For a binary tree with $h = 3$ this quantity is 7. For a 200-way tree with $h = 3$ we have $m^h - 1 = 8 * 10^6 - 1$.

To achieve a performance close to that of the best m-way search trees for a given number of elements n, the search tree must be balanced. The particular varieties of balanced m-way search trees we shall consider here are known as B-trees and B^+-trees.



node	schematic format
a	2, b, (20, c), (40, d)
b	2, 0, (10, 0), (15, 0)
c	2, 0, (25, e), (30, 0)
d	2, 0, (45, 0), (50, 0)
e	1, 0, (28, 0)

Figure 11.1: Example of a three-way search tree

11.1.2 Searching an m-Way Search Tree

Suppose we wish to search an m-way search tree for an element whose key is x. We begin at the root of the tree. Assume that this node has the structure given in the definition of an m-way search tree. For convenience, assume that $E_0.K = -\infty$ and $E_{n+1}.K = +\infty$. By searching the keys of the root, we determine i such that $E_i.K \leq x < E_{i+1}.K$. If $x = E_i.K$, then the search is complete. If $x \neq E_i.K$, then from the definition of an m-way search tree, it follows that if x is in the tree, it must be in subtree A_i. So, we move to the root of this subtree and proceed to search it. This process continues until either we find x or we have determined that x is not in the tree (the search leads us to an empty subtree). When the number of elements in the node being searched is small, a sequential search is used. When this number is large, a binary search may be used. A high-level description of the algorithm to search an m-way search tree is given in Program 11.1.

EXERCISES

1. Draw a sample 5-way search tree.
2. What is the minimum number of elements in an m-way search tree whose height is h?
3. Write an algorithm to insert an element whose key is x into an m-way search tree. What is the complexity of your algorithm?
4. Write an algorithm to delete the element whose key is x from an m-way search tree. What is the complexity of your algorithm?

```
/* search an m-way search tree for an element with key x,
   return pointer to the element if found, return NULL otherwise */
E_0.K = - MAXKEY;
for (*p = root; p; p = A_i)
{
    Let p have the format n, A_0, (E_1, A_1), ..., (E_n, A_n);
    E_{n+1}.K = MAXKEY;
    Determine i such that E_i.K ≤ x < E_{i+1}.K;
    if ( x == E_i.K) return E_i;
}
/* x is not in the tree */
return NULL;
```

Program 11.1: Searching an m-way search tree

11.2 B-TREES

11.2.1 Definition and Properties

The implementation of a database management system often relies upon either B-trees or B^+-trees (Section 11.3) to facilitate quick insertion into, deletion from, and searching of a database. A knowledge of these structures is crucial to understanding how commercial database management systems function. In defining a B-tree, it is convenient to extend m-way search trees by the addition of external nodes. An external (or failure) node is added wherever we otherwise have a **NULL** pointer. An external node represents a node that can be reached during a search only if the element being sought is not in the tree. Nodes that are not external nodes are called *internal* nodes.

Definition: A *B-tree of order m* is an m-way search tree that either is empty or satisfies the following properties:

(1) The root node has at least two children.

(2) All nodes other than the root node and external nodes have at least $\lceil m/2 \rceil$ children.

(3) All external nodes are at the same level. □

Observe that when $m = 3$, all internal nodes of a B-tree have a degree that is either 2 or 3 and when $m = 4$, the permissible degrees for these nodes are 2, 3 and 4. For this reason, a B-tree of order 3 is known as a 2-3 tree and a B-tree of order 4 is known as a

2-3-4 tree. A B-tree of order 5 is not a 2-3-4-5 tree as a B-tree of order 5 cannot have nodes whose degree is 2 (except for the root). Also, notice that all B-trees of order 2 are full binary trees. Hence, B-trees of order 2 exist only when the number of key values is $2^k - 1$ for some k. However, for any $n \geq 0$ and any $m > 2$, there is always a B-tree of order m that contains n keys.

Figure 11.2 shows a 2-3 tree (i.e., a B-tree of order 3) and Figure 11.3 shows a 2-3-4 tree (i.e., a B-tree of order 4). Notice that each (internal) node of a 2-3 tree can hold 2 elements while each such node of a 2-3-4 tree can hold 3 elements. In the figures, only the keys are shown. Note also that although Figures 11.2 and 11.3 show external nodes, external nodes are introduced only to make it easier to define and talk about B-trees. External nodes are not physically represented inside a computer. Rather, the corresponding child pointer of the parent of each external node is set to **NULL**.

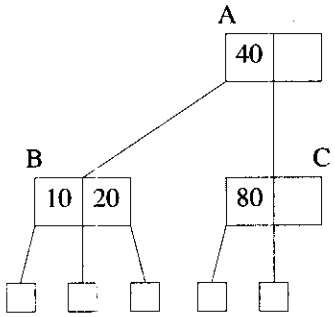


Figure 11.2: Example of a 2-3 tree

11.2.2 Number of Elements in a B-Tree

A B-tree of order m in which all external nodes are at level $l+1$ has at most $m^l - 1$ keys. What is the minumum number, N, of elements in such a B-tree? From the definition of a B-tree we know that if $l > 1$, the root node has at least two children. Hence, there are at least two nodes at level 2. Each of these nodes must have at least $\lceil m/2 \rceil$ children. Thus, there are at least $2\lceil m/2 \rceil$ nodes at level 3. At level 4 there must be at least $2\lceil m/2 \rceil^2$ nodes, and continuing this argument, we see that there are at least $2\lceil m/2 \rceil^{l-2}$ nodes at level l when $l > 1$. All of these nodes are internal nodes. If the keys in the tree are $K_1, K_2, \cdots, K_N$ and $K_i < K_{i+1}$, $1 \leq i < N$, then the number of external nodes is $N + 1$. This is so because failures occur for $K_i < x < K_{i+1}$, $0 \leq i \leq N$, where $K_0 = -\infty$ and $K_{N+1} = +\infty$. This results in $N + 1$ different nodes that one could reach while searching

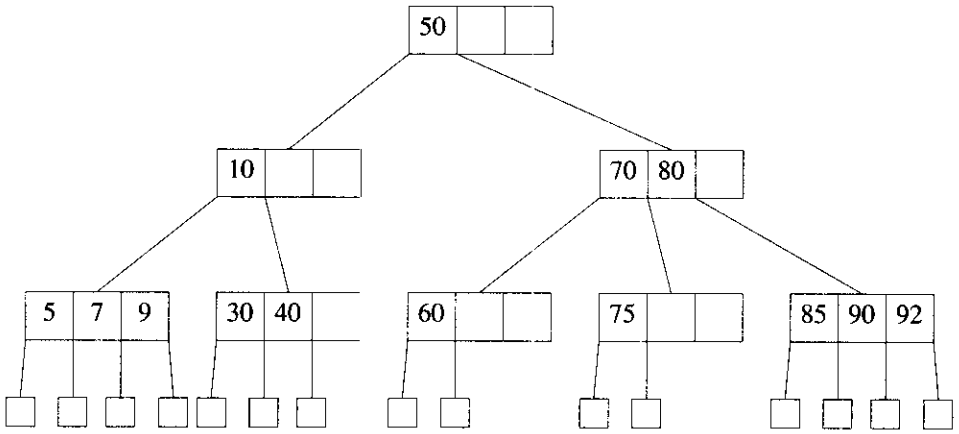


Figure 11.3: Example of a 2-3-4 tree

for a key x that is not in the B-tree. Therefore, we have

$$\begin{aligned} N+1 &= \text{number of external nodes} \\ &= \text{number of nodes at level } (l+1) \\ &\geq 2\lceil m/2 \rceil^{l-1} \end{aligned}$$

so $N \geq 2\lceil m/2 \rceil^{l-1} - 1, l \geq 1$.

This in turn implies that if there are N elements (equivalently, keys) in a B-tree of order m, then all internal nodes are at levels less than or equal to l, $l \leq \log_{\lceil m/2 \rceil}\{(N+1)/2\} + 1$. If a B-tree node can be examined with a single memory access, the maximum number of accesses that have to be made for a search is l. Using a B-tree of order $m = 200$, which is quite practical for a disk resident B-tree, a tree with $N \leq 2 \times 10^6 - 2$ will have $l \leq \log_{100}\{(N+1)/2\} + 1$. Since l is an integer, we obtain $l \leq 3$. For $N \leq 2 \times 10^8 - 2$ we get $l \leq 4$.

To search a B-tree with a number of memory accesses equal to the B-tree height we must be able to examine a B-tree node with a single memory access. This means that the size of a node should not exceed the size of a memory block (i.e., size of a cache line or disk block). For main-memory resident B-trees an m in the tens is practical and for disk resident B-trees an m in the hundreds is practical.

11.2.3 Insertion into a B-Tree

The insertion algorithm for B-trees first performs a search to determine the leaf node, p, into which the new key is to be inserted. If the insertion of the new key into p results in p having m keys, the node p is split. Otherwise, the new p is written to the disk, and the insertion is complete. To split the node, assume that following the insertion of the new element, p has the format

$$m, A_0, (E_1,A_1), \cdots, (E_m,A_m), \quad \text{and} \quad E_i < E_{i+1},\ 1 \le i < m$$

The node is split into two nodes, p and q, with the following formats:

$$\text{node } p\text{: } \lceil m/2 \rceil - 1, A_0, (E_1A_1), \cdots, (E_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1}) \tag{11.5}$$

$$\text{node } q\text{: } m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, (E_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}), \cdots, (E_m, A_m)$$

The remaining element, $E_{\lceil m/2 \rceil}$, and a pointer to the new node, q, form a tuple $(E_{\lceil m/2 \rceil}, q)$. This is to be inserted into the parent of p.

Inserting into the parent may require us to split the parent, and this splitting process can propagate all the way up to the root. When the root splits, a new root with a single element is created, and the height of the B-tree increases by one. A high-level description of the insertion algorithm for a disk resident B-tree is given in Program 11.2.

Example 11.1: Consider inserting an element with key 70 into the 2-3 tree of Figure 11.2. First we search for this key. If the key is already in the tree, then the existing element with this key is replaced by the new element. Since 70 is not in our example 2-3 tree, the new element is inserted and the total number of elements in the tree increases by 1. For the insertion, we need to know the leaf node encountered during the search for 70. Note that whenever we search for a key that is not in the 2-3 tree, the search encounters a unique leaf node. The leaf node encountered during the search for 70 is the node C, with key 80. Since this node has only one element, the new element may be inserted here. The resulting 2-3 tree is shown in Figure 11.4(a).

Next, consider inserting an element with key 30. This time the search encounters the leaf node B. Since B is full, it is necessary to split B. For this, we first symbolically insert the new element into B to get the key sequence 10, 20, 30. Then the overfull node is split using Eq. 11.5. Following the split, B has the key sequence 10 and the new node, D, has 30. The middle element, whose key is 20, together with a pointer to the new node D is inserted into the parent A of B. The resulting 2-3 tree is shown in Figure 11.4(b).

Finally, consider the insertion of an element with key 60 into the 2-3 tree of Figure 11.4(b). The leaf node encountered during the search for 60 is node C. Since C is full, a new node, E, is created. Node E contains the element with the largest key (80). Node C contains the element with the smallest key (60). The element with the median key (70), together with a pointer to the new node, E, is to be inserted into the parent A of C. Again, since A is full, a new node, F, containing the element with the largest key among

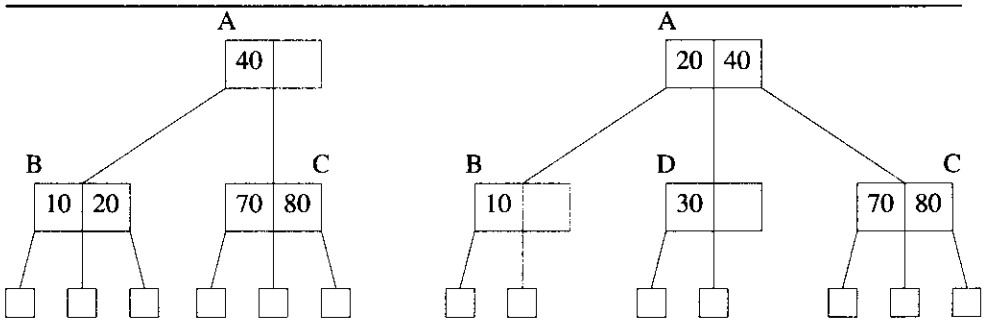
```
/* insert element x into a disk resident B-tree */
Search the B-tree for an element E with key x.K.
if such an E is found, replace E with x and return;
Otherwise, let p be the leaf into which x is to be inserted;
q = NULL;
for (e = x; p; p = p→parent())
{/* (e, q) is to be inserted into p */
    Insert (e, q) into appropriate position in node p;
    Let the resulting node have the form: n, A0, (E1, A1), ···, (En, An);
    if ( n <= m - 1) { /* resulting node is not too big */
        write node p to disk; return;
    }
    /* node p has to be split */
    Let p and q be defined as in Eq. (11.5);
    e = E[m/2];
    write nodes p and q to the disk;
}
/* a new root is to be created */
Create a new node r with format 1, root, (e, q);
root = r;
write root to disk;
```

Program 11.2: Insertion into a B-tree

{20, 40, 70} is created. As before, A contains the element with the smallest key. B and D remain the left and middle children of A, respectively, and C and E become these children of F. If A had a parent, then the element with the median key 40 and a pointer to the new node, F, would be inserted into this parent node. Since A does not have a parent, we create a new root, G, for the 2-3 tree. Node G contains the element with key 40, together with child pointers to A and F. The new 2-3 tree is as shown in Figure 11.5. □

Analysis of B-tree Insertion: For convenience, assume the B-tree is disk resident. If h is the height of the B-tree, then h disk accesses are made during the top-down search. In the worst case, all h of the accessed nodes may split during the bottom-up splitting pass. When a node other than the root splits, we need to write out two nodes. When the root splits, three nodes are written out. If we assume that the h nodes read in during the top-down pass can be saved in memory so that they are not to be retrieved from disk during the bottom-up pass, then the number of disk accesses for an insertion is at most h (downward pass) + $2(h - 1)$ (nonroot splits) + 3 (root split) = $3h + 1$.

The average number of disk accesses is, however, approximately $h + 1$ for large



(a) 70 inserted (b) 30 inserted

Figure 11.4: Insertion into the 2-3 tree of Figure 11.2

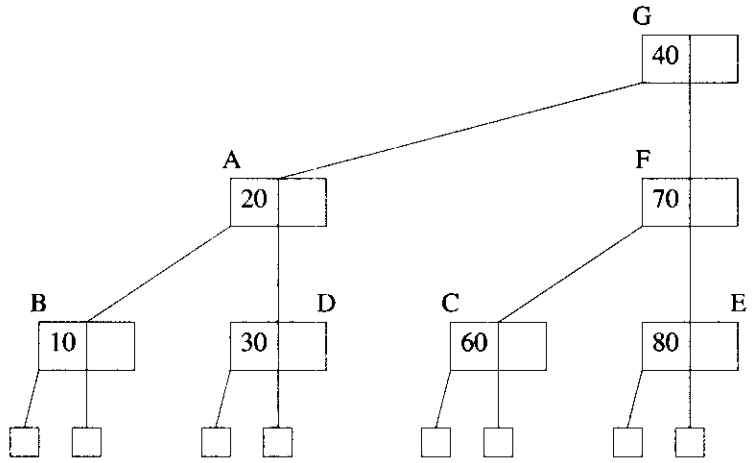
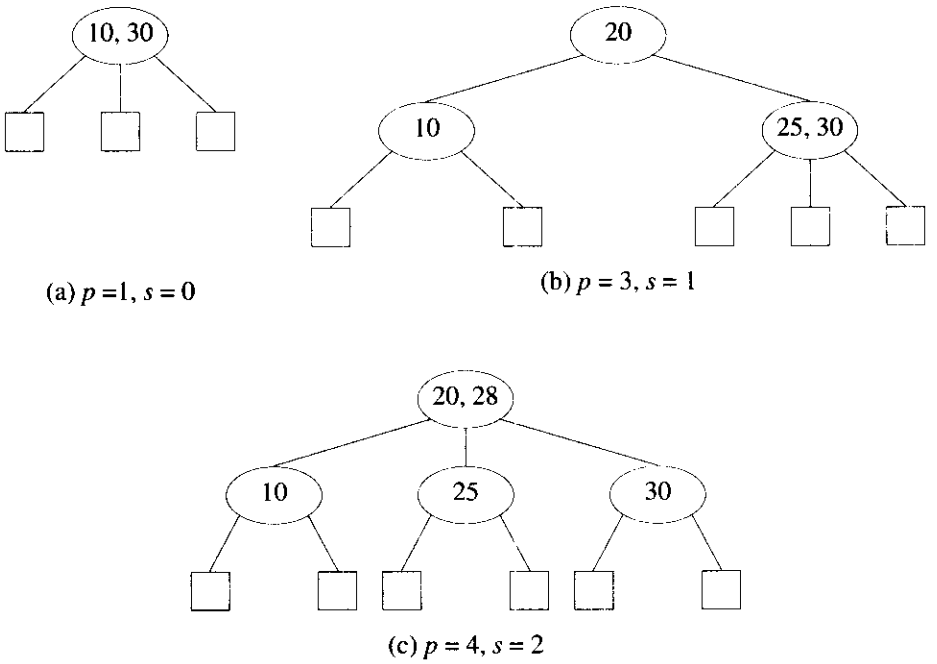


Figure 11.5: Insertion of 60 into the 2-3 tree of Figure 11.4(b)

m. To see this, suppose we start with an empty B-tree and insert N values into it. The total number of nodes split is at most $p - 2$, where p is the number of internal nodes in the final B-tree with N entries. This upper bound of $p - 2$ follows from the observation that each time a node splits, at least one additional node is created. When the root splits, two additional nodes are created. The first node created results from no splitting, and if a B-tree has more than one node, then the root must have split at least once. Figure 11.6

shows that $p - 2$ is the tightest upper bound on the number of nodes split in the creation of a p-node B-tree when $p > 2$ (note that there is no B-tree with $p = 2$). A B-tree of order m with p nodes has at least $1 + (\lceil m/2 \rceil - 1)(p - 1)$ keys, as the root has at least one key and remaining nodes have at least $\lceil m/2 \rceil - 1$ keys each. The average number of splits, s_{avg}, may now be determined as follows:



(a) $p = 1, s = 0$

(b) $p = 3, s = 1$

(c) $p = 4, s = 2$

Figure 11.6: B-trees of order 3

$$\begin{aligned} s_{avg} &= (\text{total number of splits})/N \\ &\leq (p-2)/\{1 + (\lceil m/2 \rceil - 1)(p-1)\} \\ &< 1/(\lceil m/2 \rceil - 1) \end{aligned}$$

For $m = 200$ this means that the average number of node splits is less than 1/99 per key inserted. The number of disk accesses in an insertion is $h + 2s - 1$, where s is the number of nodes that are split during the insertion. So, the average number of disk accesses is $h + 2s_{avg} + 1 < h + 101/99 \approx h + 1$. □

11.2.4 Deletion from a B-Tree

For convenience, assume we are deleting from a B-tree that resides on disk. Suppose we are to delete the element whose key is x. First, we search for this key. If x is not found, no element is to be deleted. If x is found in a node, z, that is not a leaf, then the position occupied by the corresponding element in z is filled by an element from a leaf node of the B-tree. Suppose that x is the ith key in z (i.e., $x = E_i.K$). Then E_i may be replaced by either the element with smallest key in the subtree A_i or the element with largest key in the subtree A_{i-1}. Both of these elements are in leaf nodes. In this way the deletion from a nonleaf node is transformed into a deletion from a leaf. For example, if we are to delete the element with key 20 that is in the root of Figure 11.6 (c), then this element may be replaced by either the element with key 10 or the element with key 25. Both are in leaf nodes. Once the replacement is done, we are faced with the problem of deleting either the 10 or the 25 from a leaf.

There are four possible cases when deleting from a leaf node p. In the first, p is also the root. If the root is left with at least one element, the changed root is written to disk and we are done. Otherwise, the B-tree is empty following the deletion. In the remaining cases, p is not the root. In the second case, following the deletion, p has at least $\lceil m/2 \rceil - 1$ elements. The modified leaf is written to disk, and we are done.

In the third case (rotation), p has $\lceil m/2 \rceil - 2$ elements, and its nearest sibling, q, has at least $\lceil m/2 \rceil$ elements. To determine this, we examine only one of the two (at most) nearest siblings that p may have. p is deficient, as it has one less than the minimum number of elements required. q has more elements than the minimum required. A rotation is performed. In this rotation, the number of elements in q decreases by one, and the number in p increases by one. As a result, neither p nor q is deficient following the rotation. The rotation leaves behind a valid B-tree. Let r be the parent of p and q. If q is the nearest right sibling of p, then let i be such that E_i is the ith element in r, all elements in p have a key that is less than $E_i.K$, and all those in q have a key that is greater than $E_i.K$. For the rotation, E_i becomes the rightmost element in p, E_i is replaced, in r, by the first (i.e., smallest) element in q, and the leftmost subtree of q becomes the rightmost subtree of p. The changed nodes p, q, and r are written to disk, and the deletion is complete. The case when q is the nearest left sibling of p is similar.

Figure 11.7 shows the rotation cases for a 2-3 tree. A "?" denotes a situation in which the presence or absence of an element is irrelevant. a, b, c, and d denote the children (i.e., roots of subtrees) of nodes.

In the fourth case (combine) for deletion, p has $\lceil m/2 \rceil - 2$ elements, and its nearest sibling q has $\lceil m/2 \rceil - 1$ elements. So, p is deficient, and q has the minimum number of elements required by a nonroot node. Now, nodes p and q and the in-between element E_i in the parent r are combined to form a single node. The combined node has $(\lceil m/2 \rceil - 2) + (\lceil m/2 \rceil - 1) + 1 = 2\lceil m/2 \rceil - 2 \leq m - 1$ elements, which will, at most, fill the node. The combined node is written to disk. The combining operation reduces the number of elements in the parent node, r, by one. If the parent does not become deficient

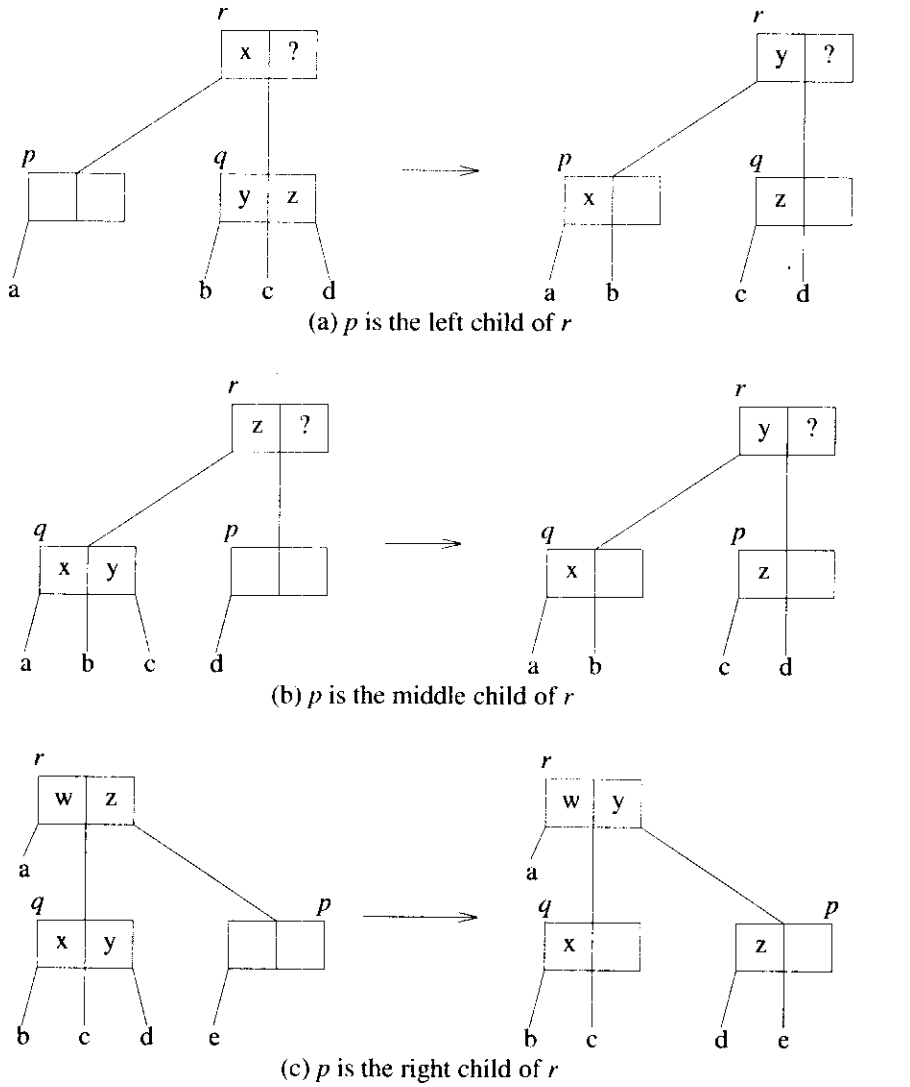


Figure 11.7: The three cases for rotation in a 2-3 tree

(i.e., it has at least one element if it is the root and at least $\lceil m/2 \rceil - 1$ elements if it is not the root), the changed parent is written to disk, and we are done. Otherwise, if the deficient parent is the root, it is discarded, as it has no elements. If the deficient parent is not the root, it has exactly $\lceil m/2 \rceil - 2$ elements. To remove this deficiency, we first

attempt a rotation with one of r's nearest siblings. If this is not possible, a combine is done. This process of combining can continue up the B-tree only until the children of the root are combined.

Figure 11.8 shows the two cases for a combine in a 2-3 tree when p is the left child of r. We leave it as an exercise to obtain the figures for the cases when p is a middle child and when p is a right child.

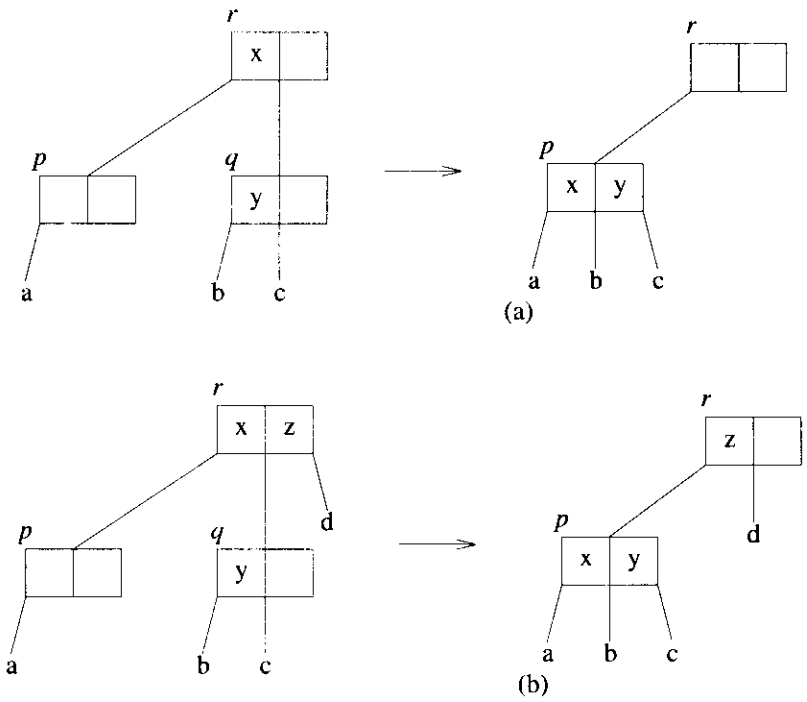


Figure 11.8: Combining in a 2-3 tree when p is the left child of r

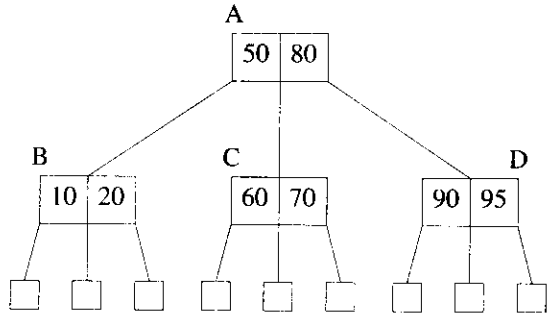
A high-level description of the deletion algorithm is provided in Program 11.3.

Example 11.2: Let us begin with the 2-3 tree of Figure 11.9(a). Suppose that the two element fields in a node of a 2-3 tree are called *dataL* and *dataR*. To delete the element with key 70, we must merely delete this element from node C. The result is shown in Figure 11.9(b). To delete the element with key 10 from the 2-3 tree of Figure 11.9(b), we need to shift *dataR* to *dataL* in node B. This results in the 2-3 tree of Figure 11.9(c).

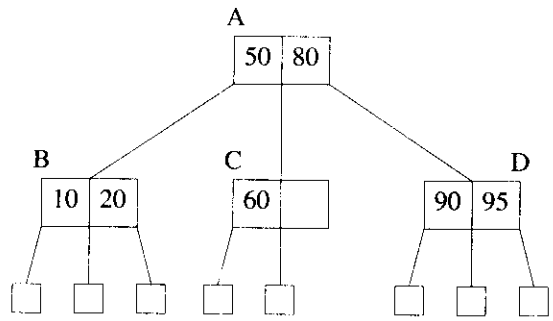
Next consider the deletion of the element with key 60. This leaves node C deficient. Since the right sibling, D, of C has 3 elements, we are in case 3 and a rotation

```
/* delete element with key x */
Search the B-tree for the node p that contains the element whose key is x;
if there is no such p return;  /* no element to delete */
Let p be of the form n, A_0, (E_1,A_1), ···, (E_n,A_n) and E_i.K = x;
if p is not a leaf {
    Replace E_i with the element with the smallest key in subtree A_i;
    Let p be the leaf of A_i from which this smallest element was taken;
    Let p be of the form n, A_0, (E_1,A_1), ···, (E_n,A_n);
    i = 1;
}
/* delete E_i from node p, a leaf */
Delete (E_i, A_i) from p; n--;
while ((n < ⌈m/2⌉ - 1) && p != root)
    if p has a nearest right sibling q {
        Let q: n_q, A_0^q, (E_1^q, A_1^q), ···, (E_{n_q}^q, A_{n_q}^q);
        Let r: n_r, A_0^r, (E_1^r, A_1^r), ···, (E_{n_r}^r, A_{n_r}^r) be the parent of p and q;
        Let A_j^r = q and A_{j-1}^r = p;
        if (n_q >= ⌈m/2⌉) {/* rotation */
            (E_{n+1}, A_{n+1}) = (E_j^r, A_0^q); n = n + 1; /* update node p */
            E_j^r = E_1^q; /* update node r */
            (n_q, A_0^q, (E_1^q, A_1^q), ···) = (n_q-1, A_1^q, (E_2^q, A_2^q), ···);
                        /* update node q */
            write nodes p, q and r to disk; return;
        } /* end of rotation */
        /* combine p, E_j^r, and q */
        s = 2* ⌈m/2⌉ - 2;
        write s, A_0, (E_1, A_1), ···, (E_n, A_n), (E_j^r, A_0^q), (E_1^q, A_1^q), ···, (E_{n_q}^q, A_{n_q}^q)
        to disk as node p;
        /* update for next iteration */
        (n, A_0 ···) = (n_r-1, A_0^r, ···, (E_{j-1}^r, A_{j-1}^r), (E_{j+1}^r, A_{j+1}^r) ···)
        p = r;
    } /* end of if p has a nearest right sibling */
    else {/* node p must have a left sibling */
        /* this is symmetric to the case where p has a right sibling,
           and is left as an exercise */
    } /* end of if-else and while */
if ( n) write p: (n, A_0, ···, (E_n,A_n))
else root = A_0; /* new root */
```

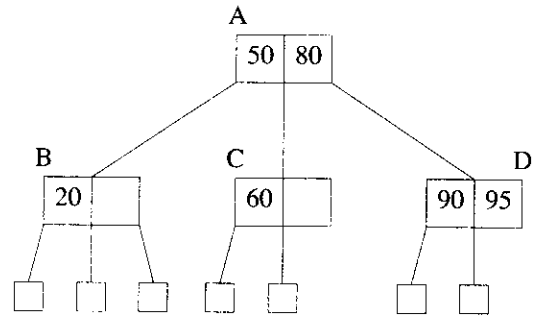
Program 11.3: Deletion from a B-tree that resides on disk



(a) Initial 2-3 tree



(b) 70 deleted



(c) 10 deleted

Figure 11.9: Deletion from a 2-3 tree (continued on next page)

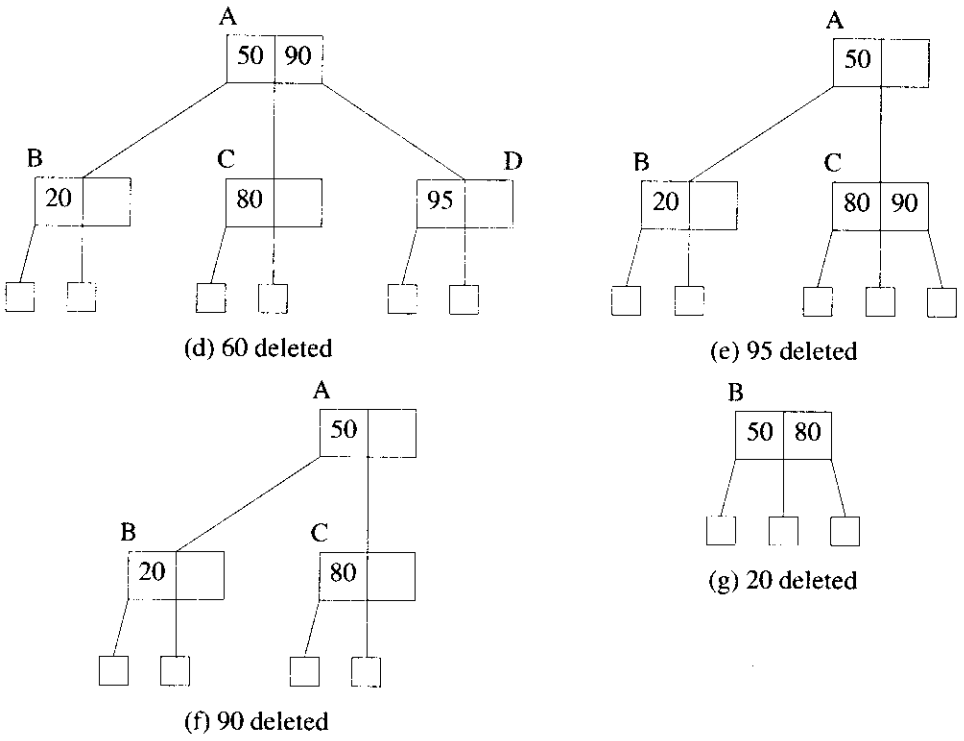


Figure 11.9: Deletion from a 2-3 tree

is performed. In this rotation, we move the the in-between element (i.e., the element whose key is 80) of the parent A of C and D to the *dataL* position of C and move the smallest element of D (i.e., the element whose key is 20) into the in-between position of the parent A of C and D (i.e., the *dataR* position of A). The resulting 2-3 tree takes the form shown in Figure 11.9(d). When the element with key 95 is deleted, node D becomes deficient. The rotation performed when the 60 was deleted is not possible now, as the left sibling, C, has the minimum number of elements required by a node in a B-tree of order 3. We now are in case 4 and must combine nodes C and D and the in-between element (90) in the parent A of C and D. For this, we move the 90 into the left sibling, C, and delete node D. Notice that in a combine, one node is deleted, whereas in a rotation, no node is deleted. The deletion of 95 results in the 2-3 tree of Figure 11.9(e). Deleting the element with key 90 from this tree results in the 2-3 tree of Figure 11.9(f). Now consider deleting the element with key 20 from this tree. Node B becomes

deficient. At this time, we examine B's right sibling, C. If C has excess elements, we can perform a rotation similar to that done during the deletion of 60. Otherwise, a combine is performed. Since C doesn't have excess elements, we proceed in a manner similar to the deletion of 95 and do a combine. This time the elements with keys 50 and 80 are moved into B, and node C is deleted. This, however, causes the parent node A to become deficient. If the parent had not been a root, we would examine its left or right sibling, as we did when nodes C (deletion of 60) and D (deletion of 95) became empty. Since A is the root, it is simply deleted, and B becomes the new root (Figure 11.9(g)). Recall that a root is deficient iff it has no element. □

Analysis of B-tree Deletion: Once again, we assume a disk-resident B-tree and that disk nodes accessed during the downward search pass may be saved in a stack in main memory, so they do not need to be reaccessed from the disk during the upward restructuring pass. For a B-tree of height h, h disk accesses are made to find the node from which the key is to be deleted and to transform the deletion to a deletion from a leaf. In the worst case, a combine takes place at each of the last $h - 2$ nodes on the root-to-leaf path, and a rotation takes place at the second node on this path. The $h - 2$ combines require this many disk accesses to retrieve a nearest sibling for each node and another $h - 2$ accesses to write out the combined nodes. The rotation requires one access to read a nearest sibling and three to write out the three nodes that are changed. The total number of disk accesses is $3h$.

The deletion time can be reduced at the expense of disk space and a slight increase in node size by including a delete bit, F_i, for each element, E_i, in a node. Then we can set $F_i = 1$ if E_i has not been deleted and $F_i = 0$ if it has. No physical deletion takes place. In this case a delete requires a maximum of $h + 1$ accesses (h to locate the node containing the element to be deleted and 1 to write out this node with the appropriate delete bit set to 0). With this strategy, the number of nodes in the tree never decreases. However, the space used by deleted entries can be reused during further insertions (see Exercises). As a result, this strategy has little effect on search and insert times (the number of levels increases very slowly when m is large). The time taken to insert an item may even decrease slightly because of the ability to reuse deleted element space. Such reuses would not require us to split any nodes. □

EXERCISES

1. Show that all B-trees of order 2 are full binary trees.
2. Use the insertion algorithm of Program 11.2 to insert an element with key 40 into the 2-3 tree of Figure 11.9(a). Show the resulting 2-3 tree.
3. Use the insertion algorithm of Program 11.2 to insert elements with keys 45, 95, 96, and 97, in this order, into the 2-3-4 tree of Figure 11.3. Show the resulting 2-3-4 tree follwoing each insert.

4. Use the deletion algorithm of Program 11.3 to delete the elements with keys 90, 95, 80, 70, 60 and 50, in this order, from the 2-3 tree of Figure 11.9(a). Show the resulting 2-3 tree following each deletion.
5. Use the deletion algorithm of Program 11.3 to delete the elements with keys 85, 90, 92, 75, 60, and 70 from the 2-3-4 tree of Figure 11.3. Show the resulting 2-3-4 tree following each deletion.
6. (a) Insert elements with keys 62, 5, 85, and 75 one at a time into the order-5 B-tree of Figure 11.10. Show the new tree after each element is inserted. Do the insertion using the insertion process described in the text.

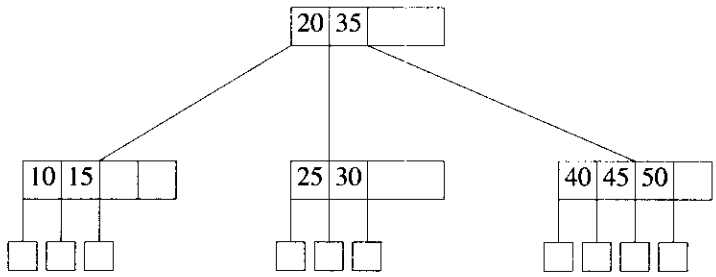


Figure 11.10: B-tree of order 5

   (b) Assuming that the tree is kept on a disk and one node may be retrieved at a time, how many disk accesses are needed to make each insertion? State any assumptions you make.
   (c) Delete the elements with keys 45, 40, 10, and 25 from the order-5 B-tree of Figure 11.10. Show the tree following each deletion. The deletions are to be performed using the deletion process described in the text.
   (d) How many disk accesses are made for each of the deletions?
7. Complete Figure 11.8 by adding figures for the cases when p is the middle child and right child of its parent.
8. Complete the symmetric case of Program 11.3.
9. Develop 2-3 tree functions to search, insert and delete. Test your functions using your own test data.
10. Develop 2-3-4 tree functions to search, insert and delete. Test your functions using your own test data.

11. Write insertion and deletion algorithms for B-trees assuming that with each element is associated an additional data member, *deleted*, such that *deleted* = **FALSE** iff the corresponding element has not been deleted. Deletions should be accomplished by setting *deleted* = **FALSE**, and insertions should make use of deleted space whenever possible without restructuring the tree.

12. Write algorithms to search and delete keys from a B-tree by position; that is, *get* (k) finds the kth smallest key, and *delete* (k) deletes the kth smallest key in the tree. (**Hint**: To do this efficiently, additional information must be kept in each node. With each pair (E_i,A_i) keep $N_i = \Sigma_{j=0}^{i-1}$ (number of elements in the subtree $A_j + 1$).) What are the worst-case computing times of your algorithms?

13. The text assumed a node structure that was sequential. However, we need to perform the following functions on a B-tree node: search, insert, delete, join, and split.

    (a) Explain why each of these functions is important during a search, insert, and delete operation in the B-tree.

    (b) Explain how a red-black tree could be used to represent each node. You will need to use integer pointers and regard each red-black tree as embedded in an array.

    (c) What kind of performance gain/loss do you expect using red-black trees for each node instead of a sequential organization? Try to quantify your answer.

14. Modify Program 11.2 so that when node p has m elements, we first check to see if either the nearest left sibling or the nearest right sibling of p has fewer than $m - 1$ elements. If so, p is not split. Instead, a rotation is performed moving either the smallest or largest element in p to its parent. The corresponding element in the parent, together with a subtree, is moved to the sibling of p that has space for another element.

15. [***Bayer and McCreight***] Suppose that an insertion has been made into node p and that it has become over-full (i.e., it now contains m elements). Further, suppose that ps nearest right sibling q is full (i.e., it contains $m - 1$ elements). So, the elements in p and q together with the in-between element in the parent of p and q make $2m$ elements. These $2m$ elements may be partitioned into three nodes p, q, and r containing $\lfloor (2m-2)/3 \rfloor$, $\lfloor (2m-1)/3 \rfloor$, and $\lfloor 2m/3 \rfloor$ elements, respectively, plus two in-between elements (one for p and q and the other for q and r). So, we may split p and q into 3 nodes p, q, and r that are almost two-thirds full, replace the former in-between element for p and q with the new one, and then insert the in-between element for q and r together with a pointer to the new node r into the parent of p and q. The case when q is the nearest left sibling of p is similar.

    Rewrite Program 11.2 so that node splittings occur only as described here.