10 SEARCH TREES

OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Variations in binary search trees—static and dynamic
- Ways of building trees of each type to ensure that they remain balanced

We have discussed the non-linear data structure, tree, and one of its most popular variations, the search tree, in Chapter 7. The binary search tree (BST) is one of the fundamental data structures extensively used for searching the target in a set of ordered data. BSTs are widely used for retrieving data from databases, look-up tables, and storage dictionaries. It is the most efficient search technique having a time complexity that is logarithmic to the size of the set. There are two cases with respect to BST construction. The first case is a set of keys and the probabilities with which they are searched, which is known in advance. The second is when knowledge about the keys is not available in advance and the keys occur dynamically. These two cases lead to the following two kinds of search trees:

1. *Static BST*—is one that is not allowed to update its structure once it is constructed. In other words, the static BST is an offline algorithm, which is presumably aware of the access sequence beforehand.
2. *Dynamic BST*—is one that changes during the access sequence. We assume that the dynamic BST is an online algorithm, which does not have prior information about the sequence.

In this chapter, we shall study about these two BSTs and the concept of symbol tables.

10.1 SYMBOL TABLE

While compilers and assemblers are scanning a program, each identifier must be examined to determine if it is a keyword. This information concerning the keywords in a programming language is stored in a symbol table. Consider the following C++ statement:

```
int limit;
```

When a compiler processes this statement, it will identify that `int` is a keyword and `limit` is an identifier. However, a question arises as to how a compiler classifies them as a keyword and a user-defined identifier. For identifying `int` as a keyword, the compiler is provided with a table of keywords. For faster search through a list of keywords, the symbol table is used as an efficient data structure.

The symbol table is a kind of a 'keyed table' which stores <key, information> pairs with no additional logical structure.
The operations performed on symbol tables are the following:

1. Inserting the <key, information> pairs into the collection.
2. Removing the <key, information> pairs by specifying the key.
3. Searching for a particular key.
4. Retrieving the information associated with a key.

When a compiler stores information that can be retrieved by some unique key value, it means we are using a keyed table. The field that contains the value by which we want to retrieve the information is the *key field*. When keyed tables are used in a compiler and an assembler, where the key (the symbol) is the programmer's identifier and the information is the location assigned by the assembler to that identifier, the keyed tables are called *symbol tables*.

10.1.1 Representation of Symbol Table

There are two different techniques for implementing a keyed table, namely, the symbol table and the tree table.

Static Tree Tables

When symbols are known in advance and no insertion and deletion is allowed, such a structure is called a *static tree table*. An example of this type of table is a reserved word table in a compiler. This table is searched once for every occurrence of an identifier in a program. If an identifier is not present in the reserved word table, then it is searched for in another table. When we know the keys and their probable frequencies of being searched, we can optimize the search time by building an optimal binary search tree (OBST). The keys have history associated with their use, which is referred to as their *probability of occurrence*. There are four options for searching:

1. Static tree table can be stored as a sorted sequential list and binary search ($O(\log_2 n)$) can be used to search a symbol.
2. Balanced BST can be used to find symbols having equal probabilities.
3. Hash tables, having the search time O(1), can be used to store a symbol table.
4. OBST is used when different symbols are searched with different probabilities.

Dynamic Tree Tables

A dynamic tree table is used when symbols are not known in advance but are inserted as they come and deleted if not required. *Dynamic keyed tables* are those that are built on-the-fly. The keys have no history associated with their use. The dynamically built tree that is a balanced BST is the best choice.

Let us now look into each of these trees in detail.

10.2 OPTIMAL BINARY SEARCH TREE

Before we study OBSTs, let us revise BSTs. A BST is one of the most important data structures in computer science. When arrays are used to store ordered data, we use the very efficient searching technique, that is, binary search. However, its insertion and deletion algorithms are inefficient as they require shifting of data in the array. An alternative is to use a linked list to store ordered data, which although provides efficient insertion and deletion algorithms, its sequential searching algorithm is inefficient. Therefore, a BST is the only data structure left that not only has an efficient searching algorithm but also efficient insertion and deletion algorithms.

A BST can be defined as a key-based tree with the following properties:

1. Every element has a key, and no two elements have the same key (i.e., keys are unique).
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are greater than the key in the root.
4. Each subtree in itself is a BST.

A BST has a few problems which are to be overcome. Consider the BST shown in Fig. 10.1.

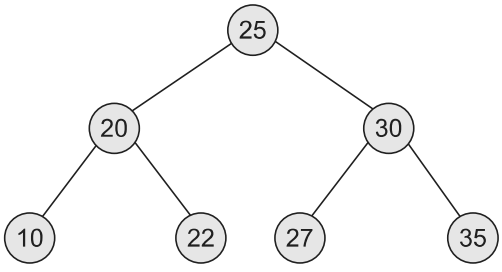


Fig. 10.1 Sample binary search tree

The inorder traversal produces 10, 20, 22, 25, 27, 30, 35. For the same set of keys, depending on their sequence of arrival, the other two search trees can be constructed as in Fig. 10.2.

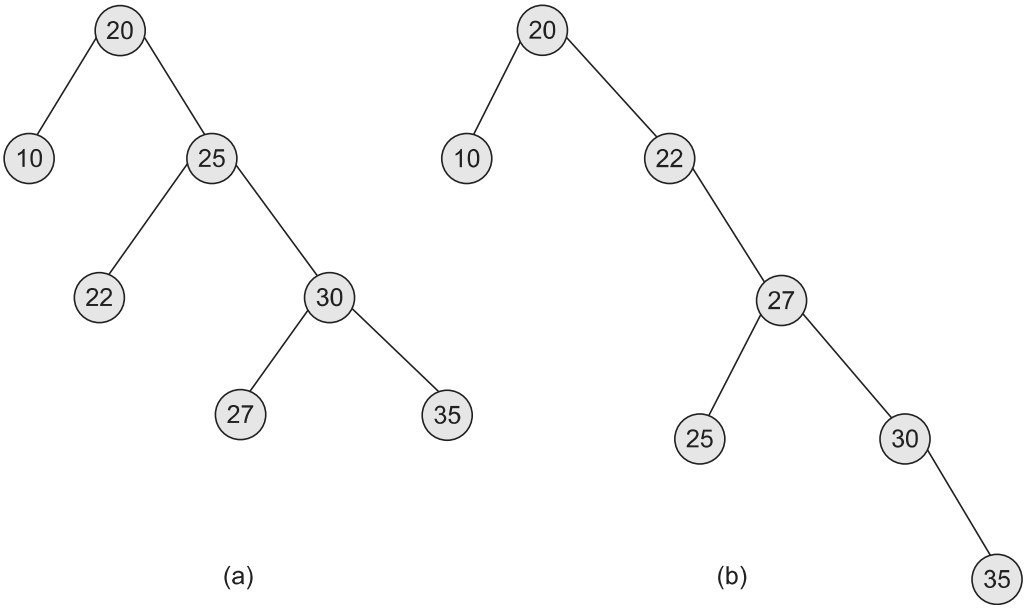


Fig. 10.2 Sample BSTs for keys (25, 10, 35, 27, 35, 20, 22) (a) Sample 1 (b) Sample 2

Note that the left BST in Fig. 10.2(a) requires utmost four comparisons to search the target in the tree, whereas for searching the target in the right tree (Fig. 10.2b), the maximum comparisons needed are five. We can say that the first BST has a better average behaviour than the second.

When the target is at the root (level 0), we need just one comparison; if the target is at level 1, we need two comparisons, and so on. Since the number of comparisons, or in other words iterations, through the search loop determines the cost of search, the cost should be minimum, that is, optimal. Hence, the optimality criteria for a static BST can be stated as minimizing the cost of the BST under a given access sequence. Such a cost can be defined as follows:

$$\text{Cost(T)} = \sum_{i=1}^{n} l(a_i) \tag{10.1}$$

Here, the total number of nodes are n, and $l(a_i)$ is the length of the i^{th} key, a.

Here, we assume that all the keys are searched with equal probabilities. However, in reality, the keys are searched with different probabilities, and it should be taken care of while constructing the tree so that the keys searched more often should require less time as compared to those searched rarely. This can be achieved by placing the more frequently searched key nodes closer to the root as compared to those that are searched rarely, to reduce the total number of average searches. A node is said to be closer to the root when its path length is lesser than that of the other nodes.

In brief, cost of a tree is computed with respect to its node's probability of search and path length. Hence,

$$\text{Cost(T)} = \sum_{i=1}^{n} W_i \times L_i \tag{10.2}$$

where,

W_i = frequency or probability (also called as weight of the i^{th} node)

L_i = level of a particular node calculated from the root node treated from level 0

Assume that there are four keys {*P*, *Q*, *R*, *S*} that are to be searched with probabilities 0.1, 0.2, 0.4, and 0.3, respectively. There are 14 possible BSTs. A few of them are shown in Fig. 10.3.

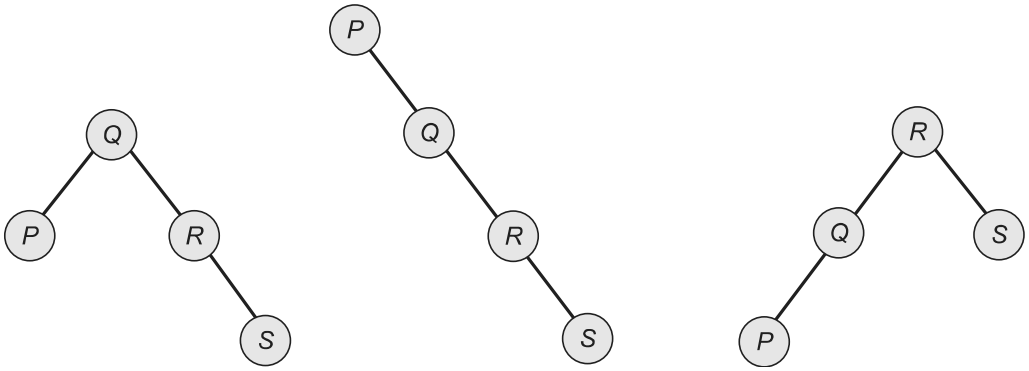


Fig.10.3 Three sample BSTs for keys {*P*, *Q*, *R*, *S*}

Now, we need to find out which of these 14 trees is the optimal one. One way to do this is to construct all possible BSTs. However, as the number of keys (n) increases, the total number of search trees also increases. So this approach is unrealistic for a large n. An alternative is to use a general algorithm.

Consider the keys $\{k_1, k_2, \ldots, k_n\}$ such that $k_1 < k_2 < k_3 < \ldots < k_n$. Every successful search for the key k_i has the probability $p(i)$. In addition, every unsuccessful search for the key x has the probability of failure $q(i)$ for $0 \le i \le n$, and $k_i < x < k_{i+1}$. We can add a fictitious node as a child for every leaf node.

For the BSTs in Fig. 10.4, all the keys represent internal nodes; all successful searches will always end at an internal node; all squares denote external nodes, which are fictitious; all unsuccessful searches will end at some external node. If there are n keys, there are $n + 1$ external nodes. So all the keys that are not a part of a BST belong to one of $(n + 1)$ equivalence classes E_i for $0 \le i \le n$. The class E_0 contains all keys $m < k_1$. The class E_1 contains all keys m such that $k_1 < m < k_2$. In general, the class E_i contains all keys m such that $k_i < m < k_{i+1}$. So if an unsuccessful search reaches at the node E_i at level l, it means that $l - 1$ comparisons are already performed. Hence, the cost of such node is $q(i) \times (\text{level}(E_i) - 1)$. Similarly, every successful search that stops at the key k_i at level l has the cost $p(i) \times \text{level}(k_i)$.

Hence, the cost of a BST is given as follows:

$$\sum_{1 \le i \le n} p(i) \times \text{level}(k_i) + \sum_{0 \le i \le n} q(i) \times \text{level}(E_i) - 1 \tag{10.3}$$

Equation (10.3) defines the cost of a BST in terms of the probabilities of successful and unsuccessful searches and the level of a node. Now, let us define an OBST. We need a BST with an optimal cost. An *OBST* is a BST with the minimum cost. Let us see how to build it by taking Example 10.1.

EXAMPLE 10.1 Given the keys = {while, do, if} and probabilities $p(i) = q(i) = 1/7$ for all i. Compute the cost of all possible BSTs and find the OBST.

Solution We get five possible BSTs for the given keys as shown in Fig. 10.4.

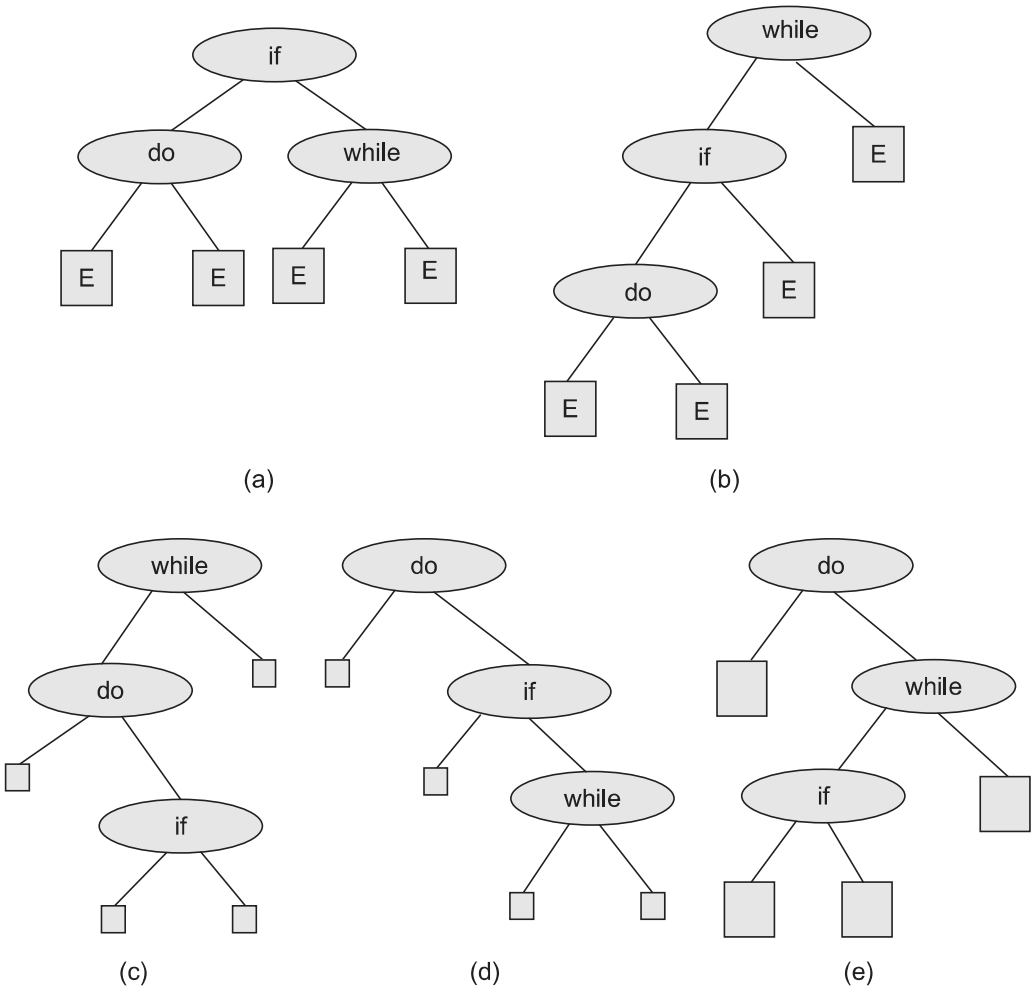


Fig. 10.4 BSTs for the keys {do, while, if} (a) BST1 (b) BST2 (c) BST3 (d) BST 4 (e) BST 5

Let us compute the cost of each BST.
For Fig. 10.4(a),

$$\text{Cost} = \sum_{1 \le i \le 3} p(i) \times \text{level}(k_i) + \sum_{0 \le i \le 3} q(i) \times \text{level}(E_i) - 1 = \text{A} + \text{B}$$

$$\text{A} = \sum_{1 \le i \le 3} p(i) \times \text{level}(k_i) = p_1 \times \text{level}(k_1) + p_2 \times \text{level}(k_2) + p_3 \times \text{level}(k_3)$$

$$= 1/7(2 + 2 + 1)$$

$$= 5/7$$

$$\text{B} = \sum_{0 \le i \le 3} q(i) \times (\text{level}(E_i) - 1) = q_0 \times \text{level}(E_0) - 1 + q_1 \times \text{level}(E_1) - 1$$

$$+ q_2 \times \text{level}(E_2) - 1 + q_3 \times \text{level}(E_3) - 1$$

$$= 1/7(2 + 2 + 2 + 2)$$

$$= 8/7$$

Therefore, cost = (5/7) + (8/7) = 13/7
For Fig. 10.4(b), total cost = A + B

where $$\text{A} = \sum_{1 \le i \le 3} p(i) \times \text{level}(k_i) = \frac{1}{7}(1 + 2 + 3)$$

$$= 6/7$$

$$\text{B} = \sum_{0 \le i \le 3} q(i) \times \text{level}(E_i) = \frac{1}{7}(3 + 3 + 2 + 1)$$

$$= 9/7$$

Therefore, cost = (6/7) + (9/7) = 15/7

Similarly, for Figs 10.4(c)–(e), the cost of each subtree = 15/7.

The cost of the tree in Fig. 10.4(a) is the least; hence, it is the OBST.

Practically, we cannot use such an approach to find an OBST as we will need to draw all possible BSTs and then find the cost of all BSTs. As the number of keys increases, the number of BSTs also increases. Dynamic programming approach can be used to construct an OBST by considering the probabilities of both successful and unsuccessful searches for the given set of keys.

We construct an OBST step-by-step using the following three formulae:

$$w(i, j) = p(j) + q(j) + w(i, j - 1)$$

$$c(i, j) = \min_{(i < a \le j)} \{c(i, a - 1) + c(a, j)\} + w(i, j)$$

$$r(i, j) = a$$

where,

$w(i, j)$is the weight of node (i, j)

$c(i, j)$is the cost of node (i, j)

$c(i, a-1)$is the cost of left subtree

$c(a, j)$is the cost of right subtree

$r(i, j)$is the root of the tree

The dynamic programming approach can be used to construct an OBST stepwise, where the principal of optimality should hold at each step. Assume that there are n keys $\{k_1, k_2, \ldots, k_n\}$ where $k_1 < k_2 < k_3 < \ldots < k_n$. So at some step, if k_a is the root of a tree, then the resultant tree is as in Fig. 10.5.

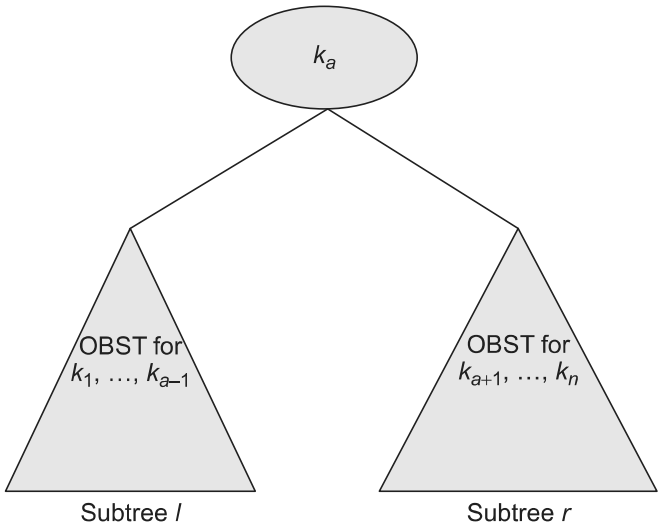


Fig. 10.5 Resultant OBST

Since this is a BST, the left subtree l has keys $k_1, k_2, \ldots, k_{a-1}$, and external nodes $E_0, E_1, \ldots, E_{a-1}$.

Therefore, using Eq. (10.3), the cost of the left subtree l is

$$\text{Cost}(l) = \sum_{1 \le i\,(a-1)} p(i) \times \text{level}(k_a) + \sum_{0 \le i(a-1)} q(i) \times \text{level}(E_i) - 1 \qquad (10.4)$$

Similarly, the cost of right subtree using Eq. (10.3) is

$$\text{Cost}(r) = \sum_{(a+1) \le i \le n} p(i) \times \text{level}(k_a) + \sum_{a \le i \le n} q(i) \times \text{level}(E_i) - 1 \qquad (10.5)$$

Therefore, the cost of the tree in Fig. 10.5 is the sum of probability of the node k_a, cost of the left subtree l, cost of the right subtree r, weight of the nodes from 0 to $a-1$, and weight of the nodes from a to n. In notation, this can be stated as follows:

$$p(a) + \text{cost}(l) + \text{cost}(r) + w(0, a-1) + w(a, n) \tag{10.6}$$

Cost(l) and cost(r) are determined considering their roots at level 1. If cost(l) is minimum and cost(r) is also minimum, then the cost of Eq. (10.5) is also minimum, and thus, we can conclude that the tree in Fig. 10.4(a) is optimal.

Let c(i, j) denote the cost of an OBST t_{ij} having keys k_{i+1}, …, k_j and external nodes E_i, …, E_j.. So for the left subtree l of OBST in Fig. 10.5, cost (l) = $c(0, a-1)$ and for its right subtree r, cost(r) = $c(a, n)$.

Hence,

$$p(a) + c(0, a-1) + c(a, n) + w(0, a-1) + w(a, n) \tag{10.7}$$

Equation (10.4) gives the cost of a tree having nodes from k_0 to k_n. In general, we can write an equation that gives the cost for a subtree having nodes from k_i to k_j as

$$p(a) + c(i, a-1) + c(a, j) + w(i, a-1) + w(a, j) \tag{10.8}$$

Obviously, Eq. (10.7) gives the minimum cost only if a is chosen properly. So we have to solve Eq. (10.6) for different values of a and then select the minimum. Hence, we can generalize Eq. (10.7) to get the following equation:

$$c(i,j) = \min_{i < a \le j} [c(i,a-1) + c(a,j) + w(i,a-1) + p(a) + w(a,j)] \tag{10.9}$$

The steps to find OBST are as follows:

1. We begin by considering all unsuccessful probabilities as initially there are no nodes in the tree. $c(i, i) = 0$, $r(i, i) = 0$, and $w(i, i) = q(i)$ for $0 \le i \le n$, where n is the number of keys.
2. Compute $c(i, j)$ for $j - i = 1$, that is, we are constructing a node of level 1. In addition, compute $w(i, j) = p(j) + q(j) + w(i, j-1)$, and the root $r(i, j)$ is the value of a which minimizes $c(i, j)$.

$$c(i,j) = \min_{i < a \le j} [c(i,a-1) + c(a,j) + w(i,j)]$$

3. Compute $c(i, j)$ for $j - i = 2$. In addition, compute $w(i, j)$ and $r(i, j)$ as in the previous step.
4. Continue the process till $j - i = n$. Here, w_{on}, c_{on}, and r_{on} denote the weight, cost, and root of OBST, respectively.

5. Finally, we can construct an OBST having the root $r_{on} = a$, which means that the key k_a is the root.

In general, let r_{ij} be any node in an OBST, whose value is a. Then, its left node is $r_{i,a-1}$ and its right node is $r_{a,j}$. It is shown in Fig. 10.6.

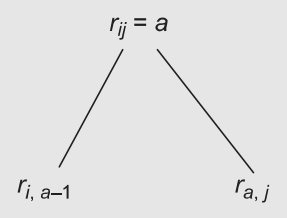


Fig. 10.6 OBST

Using this, we can construct a tree until we get $r_{ij} = 0$ at all the nodes and these are the *external nodes* of a tree.

The initial cost table of the dynamic programming algorithm for constructing an OBST is shown in Fig. 10.7.

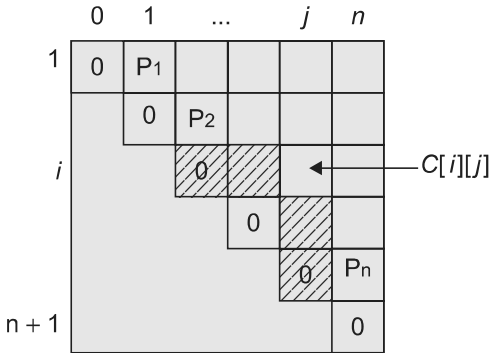


Fig. 10.7 Initial cost table of OBST

The values needed for computing `C[i][j]` are shaded in Fig. 10.7. They are the values in row `i` and to the left of column `j`, and the values in column `j` and the rows below row `i`.

Consider an OBST tree node having the following structure:

```
class leaf
{
   char name[10];
} leaf[max];
```

Program Code 10.1 implements the logic for building an OBST and computing its cost using C++.

PROGRAM CODE 10.1

```
#include<stdio.h>
#define max 20
int i, j, k, n, min, r[max][max];
float p[max], q[max], w[max][max], c[max][max];
void OBST();
void print(int, int);
void print_tab();
main()
{
   cout << "\n Enter no. of leaves in tree:"
   cin >> n;
   cout << "\n Enter leaf label";
   for(i = 1; i <= n; i++)
      cin >> leaf[i].name;
   cout << "\n Enter the probability of successful
   search:";
   for(i = 1; i <= n; i++)
   {
      cout << "p["<<i<<"]";
      cin >> sp[i];
   }
   cout << "\n Enter the probability of unsuccessful
   search: ";
   for(i = 0; i <= n; i++)
   {
      cout << "q["<<i<<"]";
      cin >> q[i];
   }
   cout << "\ninput:\n<<Leaf("<<n<<")";
   for(i = 1; i <= n; i++)
   {
      cout << "leaf["<<i<<"].name";
      cout << "np(1:"<<n<<")";
   }
   for(i = 1; i <= n; i++)
   {
      cout << "p["<<i<<"]";
      cout << "\nq(0:"<<n<<")=";
   }
   for(i = 0; i <= n; i++)
```

```
      cout << "\t<<q[i]";
   OBST();
   print_tab();
   print(0, n);
}

void OBST()
{
   for(i = 0; i < n; i++)
   {
      r[i][i] = c[i][i] = 0; w[i][i] = q[i];
      w[i][i + 1] = p[i + 1] + q[i + 1] + w[i][i];
      c[i][i + 1] = w[i][i + 1];
      r[i][i + 1] = i + 1;
   }
   c[n][n] = 0.0; r[n][n] = 0.0; w[n][n] = q[n];
   for(i = 2; i <= n; i++)
   {
      for(j = 0; j <= n - i; j++)
      {
         w[j][j + i] = w[j][j + i - 1] + p[j + i] + q[j
         + i];
         c[j][j + i] = 999;
         for(k = j + 1; k < j + i; k++)
            if(c[j][j + i] > c[j][k - 1] + c[k][j + i])
            {
               c[j][j + i] = c[j][k - 1] + c[k][j + i];
               r[j][j + i] = k;
            }
         c[j][j+i]+=w[j][j+i];
      }
   }
}

void print(int l, int rr)
{
   if(l >= rr) return;
   if(r[l][r[l][rr] - 1] != 0)
      cout <<  "\nleft child of "<<leaf[r[l][rr]].name
```

```
      <<"\t"<<leaf[r[l][r[l][rr] - 1]].name;
   if(r[r[l][rr]][rr] != 0)
      cout << "\nright child of"<< leaf[r[l][rr]].name
      <<"\t" <<leaf[r[r[l][rr]][rr]].name;
   print(l,r[l][rr] - 1);
   print(r[l][rr],rr);
}

void print_tab()
{
   cout << "\noutput:\n";
   cout  <<"-----------------------------------------
   -------------\n";
   for(i = 0; i <= n; i++)
      cout << "w" << i << i << "=" << w[i][i] << "\n";
   for(i = 0; i <= n; i++)
      cout << "w" << i << i << "=" << c[i][i] << "\n";
   for(i = 0; i <= n; i++)
      cout << "w" << i << i << "=" << r[i][i] << "\n";
   cout << "-----------------------------------------
   --------------\n";
   k = 1;
   while(k <= n)
   {
      for(i = 0, j = i + k; i < n, j <= n; i++, j++)
        cout << "w" << i << j << "=" << w[i][j] << "\n";
      for(i = 0, j = i + k; i < n, j <= n; i++, j++)
        cout << "C" << i << j << "=" << c[i][j] << "\n";
      for(i = 0, j = i + k; i < n, j <= n; i++, j++)
        cout << "R" << i << j << "=" << r[i][j] << "\n";
      cout  <<"-----------------------------------------
      ----------------\n";
      k++;
           }
    cout << "\nOBST:c[0][n]<<w[0][n]<<leaf[r[0][n]]
    .name"
    cout << \nOBST:c[0][%d] = %0.2f w[0][%d] = %0.2f
    r[0][%d] = %s", n, c[0][n], n, w[0][n], n, leaf[r[0]
    [n]].name);
}
```

Program Code 10.1 is the implementation of the OBST construction through the dynamic approach we just discussed. Let us see its working with Example 10.2.

EXAMPLE 10.2 Find an OBST using a dynamic programming for $n = 4$ and keys $(k_1 < k_2 < k_3 < k_4)$ = (do, if, int, while) given that $p(1:4) = (3, 3, 1, 1)$ and $q(0:4) = (2, 3, 1, 1, 1)$.

Solution

Step 1: Initially, $c(i, i) = 0$, $r(i, i) = 0$, and $w(i, i) = q(i)$ for $0 \le i \le 4$.
Hence, $w(0, 0) = 2$, $w(1, 1) = 3$, $w(2, 2) = w(3, 3) = w(4, 4) = 1$
This is shown in Table 10.1.

Table 10.1 OBST computation for Example 10.2 after step 1

	0	1	2	3	4	Initial values ←
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						
2						
3						
4						

Step 2: $w(i, j) = p(j) + q(j) + w(i, j - 1)$

$$c(i,j) = \min_{i < a \le j} [c(i,a-1) + c(a,j) + w(i,j)]$$

$r = (i, j)$ = value of a which minimizes $c(i, j)$

Let us compute $c(i, j)$ for $j - i = 1$

$$w(0, 1) = p(1) + q(1) + w(0, 0)$$
$$= 3 + 3 + 2 = 8$$
$$c(0, 1) = w(0, 1) + \min[c(0, 0) + c(1, 1)] \quad \text{for } a = 1$$
$$= 8 + [0 + 0] = 8$$
$$r(0, 1) = 1$$
$$w(1, 2) = p(2) + q(2) + w(1, 1) = 3 + 1 + 3 = 7$$
$$c(1, 2) = w(1, 2) + \min[c(1, 1) + c(2, 2)] = 7 + [0 + 0] = 7 \quad \text{for } a = 2$$

$r(1, 2) = 2$
$w(2, 3) = p(3) + q(3) + w(2, 2) = 1 + 1 + 1 = 3$
$c(2, 3) = w(2, 3) + \min[c(2, 2) + c(3, 3)] = 3 + [0 + 0] = 3$ for $a = 3$
$r(2, 3) = 3$
$w(3, 4) = p(4) + q(4) + w(3, 3) = 1 + 1 + 1 = 3$
$c(3, 4) = w(3, 4) + \min[c(3, 3) + c(4, 4)] = 3 + [0 + 0] = 3$
$r(3, 4) = 4$

This computation is shown in Table 10.2.

Table 10.2 OBST computation for Example 10.2 after step 2

	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	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	Here $j - i = 1$, that is, while calculating c_{ij}, a took only one value, that is, j.
2					
3					
4					

Step 3: Compute $c(i, j)$ for $j - i = 2$

$w(0, 2) = p(2) + q(2) + w(0, 1)$
$= 3 + 1 + 8 = 12$
$c(0, 2) = w(0, 2) + \min[c(0, 0) + c(1, 2)$ for $a = 1$,
$c(0, 1) + c(2, 2)$ for $a = 2]$
$= 12 + \min[0 + 7, 8 + 0]$
$= 12 + 7 = 19$
$r(0, 2) = 1$
$w(1, 3) = p(3) + q(3) + w(1, 2) = 1 + 1 + 7 = 9$
$c(1, 3) = w(1, 3) + \min[c(1, 1) + c(2, 3)$ for $a = 2$,
$c(1, 2) + c(3, 3)$ for $a = 3]$
$= 9 + \min[0 + 3, 7 + 0]$
$= 9 + 3 = 12$
$r(1, 3) = 2$
$w(2, 4) = p(4) + q(4) + w(2, 3) = 1 + 1 + 3 = 5$

$$c(2, 4) = w(2, 4) + \min[c(2, 2) + c(3, 4) \text{ for } a = 3 \text{ , } c(2, 3) + c(4, 4)]$$
$$\text{for } a = 4$$
$$= 5 + \min[0 + 3, 3 + 0]$$
$$= 5 + 3 = 8$$
$$r(2, 4) = 3$$

Table 10.3 shows this computation.

Table 10.3 OBST computation for Example 10.2 after step 3

	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	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$	Here, $j - i = 2$, that is, while calculating c_{ij}, a took two values.	
3					
4					

Step 4: Compute $c(i, j)$ for $j - i = 3$

$$w(0, 3) = p(3) + q(3) + w(0, 2) = 1 + 1 + 12 = 14$$
$$c(0, 3) = w(0, 3) + \min[c(0, 0) + c(1, 3) \quad \text{for } a = 1, c(0, 1) +$$
$$c(2, 3) \quad \text{for } a = 2, c(0, 2) + c(3, 4) \quad \text{for } a = 3]$$
$$= 14 + \min[0 + 12, 8 + 3, 19 + 3]$$
$$= 14 + \min[12, 11, 22]$$
$$= 14 + 11 = 25$$
$$r(0, 3) = 2$$
$$w(1, 4) = p(4) + q(4) + w(1, 3) = 1 + 1 + 9 = 11$$
$$c(1, 4) = w(1, 4) + \min[c(1, 1) + c(2, 4) \quad \text{for } a = 2, c(1, 2) +$$
$$c(3, 4) \quad \text{for } a = 3, c(1, 3) + c(4, 4) \quad \text{for } a = 4]$$
$$= 11 + \min[0 + 8, 7 + 3, 12 + 0]$$
$$= 11 + \min[8, 10, 12]$$
$$= 11 + 8 = 19$$
$$r(1, 4) = 2$$

This computation is shown in Table 10.4.

Table 10.4 OBST computation for Example 10.2 after step 4

	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	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$	Here $j - i = 3$, that is, while calculating c_{ij}, a took three values.		
4					

Step 5: Compute $c(i, j)$ for $j - i = 4$

$$w(0, 4) = p(4) + q(4) + w(0, 3) = 1 + 1 + 14 = 16$$

$$c(0, 4) = w(0, 4) + \min[c(0, 0) + c(1, 4) \quad \text{for } a = 1, c(0, 1) + c(2, 4) \quad \text{for } a = 2,$$
$$c(0, 2) + c(3, 4) \quad \text{for } a = 3, c(0, 3) + c(4, 4) \quad \text{for } a = 4]$$
$$= 16 + \min[0 + 19, 8 + 8, 19 + 3, 25 + 0]$$
$$= 16 + \min[19, 16, 22, 25]$$
$$= 16 + 16 = 32$$

$$r(0, 4) = 2$$

All these computations are shown in Table 10.5.

Table 10.5 OBST computation for Example 10.2 after step 5

	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	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

In the last step, we obtained $w_{04} = 16$, $c_{04} = 32$, $r_{04} = 2$, which denote that for the given keys = (do, if, int, while), an OBST has weight 16, cost 32, and root k_2 = if.

In Table 10.5, row i and column j shows the result of $w(j, i+j)$, $c(j, i+j)$, and $r(j, i+j)$, respectively. The calculation proceeds row-by-row.

The r values are shown in Fig. 10.8.

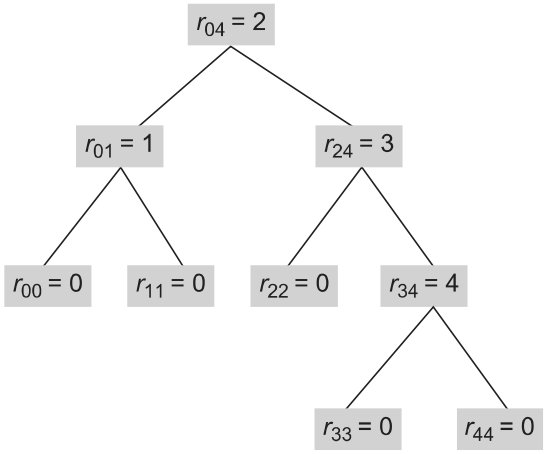


Fig. 10.8 Tree and r values

Let us construct an OBST as shown in Fig. 10.9 from the calculations based on these r values.

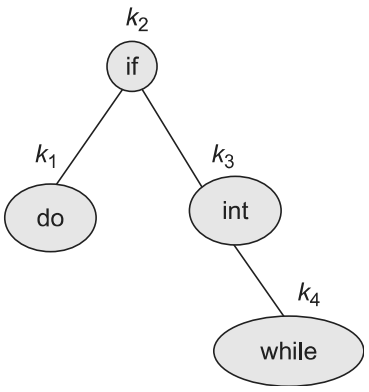


Fig. 10.9 OBST for Example 10.2

Let us now see another example of OBST construction through dynamic approach in Example 10.3.

EXAMPLE 10.3 Find an OBST using the dynamic programming approach for $n = 4$, keys = (count, float, if, while). Compute $w(i, j)$, $r(i, j)$, and $c(i, j)$ for $0 \le i \le j \le 4$ given that $p(1) = 1/20$, $p(2) = 1/5$, $p(3) = 1/10$, $p(4) = 1/20$, $q(0) = 1/5$, $7(1) = 1/10$, $q(2) = 1/5$, $q(3) = 1/20$, and $q(4) = 1/20$. Using $r(i, j)$, construct an OBST.

Solution

$$p(1:4) = \frac{1}{20}, \frac{1}{5}, \frac{1}{10}, \frac{1}{20} = (0.05, 0.2, 0.1, 0.05)$$

$$q(0:4) = \frac{1}{5}, \frac{1}{10}, \frac{1}{5}, \frac{1}{20}, \frac{1}{20} = (0.2, 0.1, 0.2, 0.05, 0.05)$$

Step 1: $c(i, i) = 0$, $r(i, i) = 0$, and $w(i, i) = q(i)$ for $0 \le i \le 4$
Hence, $w_{00} = 0.2$, $w_{11} = 0.1$, $w_{22} = 0.2$, $w_{33} = 0.05$, $w_{44} = 0.05$.

Step 2: $w(i, j) = q(j) + p(j) + w(i, j - 1)$

$$c(i, j) = \min_{i < a \le j} [c(i, a - 1) + c(a, j) + w(i, j)]$$

$r(i, j)$ = value of a which minimizes $c(i, j)$
Compute $c(i, j)$ for $j - i = 1$.
$w(0, 1) = p(1) + q(1) + w(0, 0)$
$= 0.05 + 0.1 + 0.2 = 0.35$
$c(0, 1) = w(0, 1) + \min[c(0, 0) + c(1, 1)]$
$= 0.35 + [0 + 0] = 0.35$
$r(0, 1) = 1$
$w(1, 2) = p(2) + q(2) + w(1, 1) = 0.2 + 0.2 + 0.1 = 0.5$
$c(1, 2) = w(1, 2) + \min[c(1, 1) + c(2, 2)] = 0.5 + [0 + 0] = 0.5$
$r(1, 2) = 2$
$w(2, 3) = p(3) + q(3) + w(2, 2) = 0.1 + 0.05 + 0.2 = 0.35$
$c(2, 3) = w(2, 3) + \min[c(2, 2) + c(3, 3)] = 0.35 + [0 + 0] = 0.35$
$r(2, 3) = 3$
$w(3, 4) = p(4) + q(4) + w(3, 3) = 0.05 + 0.05 + 0.05 = 0.15$
$c(3, 4) = w(3, 4) + \min[c(3, 3) + c(4, 4)] = 0.15 + [0 + 0] = 0.15$
$r(3, 4) = 4$

Step 3: Compute $c(i, j)$ for $j - i = 2$.
$w(0, 2) = p(2) + q(2) + w(0, 1)$
$= 0.2 + 0.2 + 0.35 = 0.75$

$$
\begin{aligned}
c(0, 2) &= w(0, 2) + \min[c(0, 0) + c(1, 2), c(0, 1) + c(2, 2)] \\
&= 0.75 + \min[0 + 0.5, 0.35 + 0] \\
&= 0.75 + 0.35 = 1.10 \\
r(0, 2) &= 2 \\
w(1, 3) &= p(3) + q(3) + w(1, 2) = 0.1 + 0.05 + 0.5 = 0.65 \\
c(1, 3) &= w(1, 3) + \min[c(1, 1) + c(2, 3), c(1, 2) + c(3, 3)] \\
&= 0.65 + \min[0 + 0.35, 0.5 + 0] \\
&= 0.65 + 0.35 = 1.00 \\
r(1, 3) &= 2 \\
w(2, 4) &= p(4) + q(4) + w(2, 3) = 0.05 + 0.05 + 0.35 = 0.45 \\
c(2, 4) &= w(2, 4) + \min[c(2, 2) + c(3, 4), c(2, 3) + c(4, 4)] \\
&= 0.45 + \min[0 + 0.15, 0.35 + 0] \\
&= 0.45 + 0.15 = 0.60 \\
r(2, 4) &= 3
\end{aligned}
$$

Step 4: Compute $c(i, j)$ for $j - i = 3$.

$$
\begin{aligned}
w(0, 3) &= p(3) + q(3) + w(0, 2) \\
&= 0.1 + 0.05 + 0.75 = 0.90 \\
c(0, 3) &= w(0, 3) + \min[c(0, 0) + c(1, 3), c(0, 1) + c(2, 3), c(0, 2) + c(3, 3)] \\
&= 0.9 + \min[0 + 1, 0.35 + 0.35, 1.1 + 0] \\
&= 0.9 + 0.7 = 1.6 \\
r(0, 3) &= 2 \\
w(1, 4) &= p(4) + q(4) + w(1, 3) = 0.05 + 0.05 + 0.65 = 0.75 \\
c(1, 4) &= w(1, 4) + \min[c(1,1) + c(2, 4), c(1, 2) + c(3, 4), c(1, 3) + c(4, 4)] \\
&= 0.75 + \min[0 + 0.6, 0.5 + 0.15, 1 + 0] \\
&= 0.75 + \min[0.6, 0.65, 1] \\
&= 0.75 + 0.6 = 1.35 \\
r(1, 4) &= 2
\end{aligned}
$$

Step 5: Compute $c(i, j)$ for $j - i = 4$.

$$
\begin{aligned}
w(0, 4) &= p(4) + q(4) + w(0, 3) = 0.05 + 0.05 + 0.9 = 1.00 \\
c(0, 4) &= w(0, 4) + \min[c(0, 0) + c(1, 4), c(0, 1) + c(2, 4), c(0, 2) + c(3, 4), \\
&\qquad c(0, 3) + c(4, 4)] \\
&= 1 + \min[0 + 1.35, 0.35 + 0.6, 1.1 + 0.15, 1.6 + 0] \\
&= 1 + \min[1.35, 0.95, 1.25, 1.6] \\
&= 1 + 0.95 = 1.95 \\
r(0, 4) &= 2
\end{aligned}
$$

Hence, for the keys (k_1, k_2, k_3, k_4) = (count, float, if, while), an OBST has weight $w_{04} = 1$, cost $c_{04} = 1.95$ and root $r_{04} = 2$.

These calculations can be written in the table form as in Table 10.6.

Table 10.6 OBST computations for Example 10.3

	0	1	2	3	4
0	$w_{00} = 0.2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 0.1$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 0.2$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 0.05$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 0.05$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 0.35$ $c_{01} = 0.35$ $r_{01} = 1$	$w_{12} = 0.5$ $c_{12} = 0.5$ $r_{12} = 2$	$w_{23} = 0.35$ $c_{23} = 0.35$ $r_{23} = 3$	$w_{34} = 0.15$ $c_{34} = 0.15$ $r_{34} = 4$	
2	$w_{02} = 0.75$ $c_{02} = 1.1$ $r_{02} = 2$	$w_{13} = 0.65$ $c_{13} = 1$ $r_{13} = 2$	$w_{24} = 0.45$ $c_{24} = 0.6$ $r_{24} = 3$		
3	$w_{03} = 0.9$ $c_{03} = 1.6$ $r_{03} = 2$	$w_{14} = 0.75$ $c_{14} = 1.35$ $r_{14} = 2$			
4	$w_{04} = 1$ $c_{04} = 1.95$ $r_{04} = 2$				

Figure 10.10 shows the calculated *r* values.

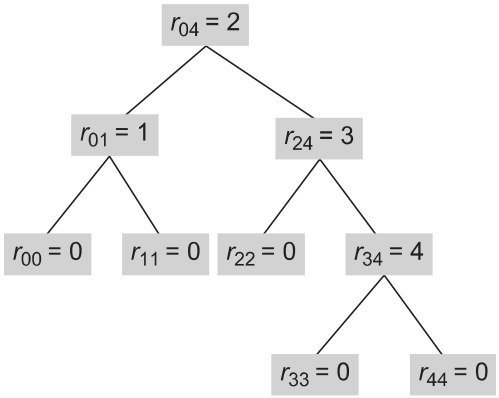


Fig. 10.10 Keys and *r* value

Figure 10.11 is the OBST obtained for Example 10.3 based on these *r* values.

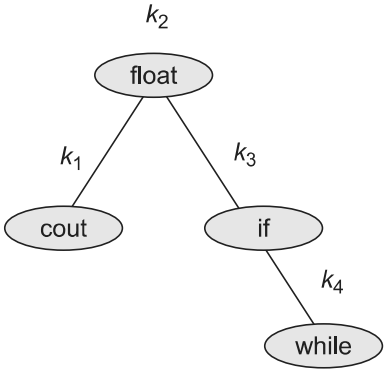


Fig. 10.11 OBST for Example 10.3

10.3 AVL TREE (HEIGHT-BALANCED TREE)

In many applications, insertions and deletions occur frequently with no predictable order. Sometimes, it is important to optimize the search times by keeping the tree balanced at all times. The resulting BST is called AVL tree. It was described by two Russian mathematicians G. M. Adelson-Velskii and E. M. Landis in 1962.

An *AVL tree* is a BST where the heights of the left and right subtrees of the root differ by utmost 1 and the left and right subtrees are again AVL trees. The formal definition is as follows:

Definition: An empty tree is height-balanced, if T is a non-empty binary tree with T_L and T_R as its left and right subtrees, respectively, with the following properties:

1. T_L and T_R are height-balanced.
2. $-1 \leq |h_L - h_R| \leq 1$, where h_L and h_R are the heights of T_L and T_R, respectively.

In an AVL tree with n nodes, the searches, insertions, and deletions can all be achieved in time O(log n), even in the worst case. To keep the tree height-balanced, we have to find out the balance factor of each node in the tree after every insertion or deletion.

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

For example, consider the BST as shown in Fig. 10.12.

BF(Fri) = 0
BF(Mon) = +1
BF(Sun) = +2

Because BF(Sun) = +2, the tree is no longer height-balanced, and it should be restructured.

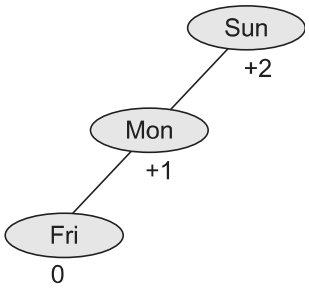


Fig. 10.12 Unbalanced BST

If a node is inserted or deleted from a balanced tree, then it may become unbalanced. So to rebalance it, the position of some nodes can be changed in proper sequence. This can be achieved by performing rotations of nodes. For example, consider the BST as in Fig. 10.13.

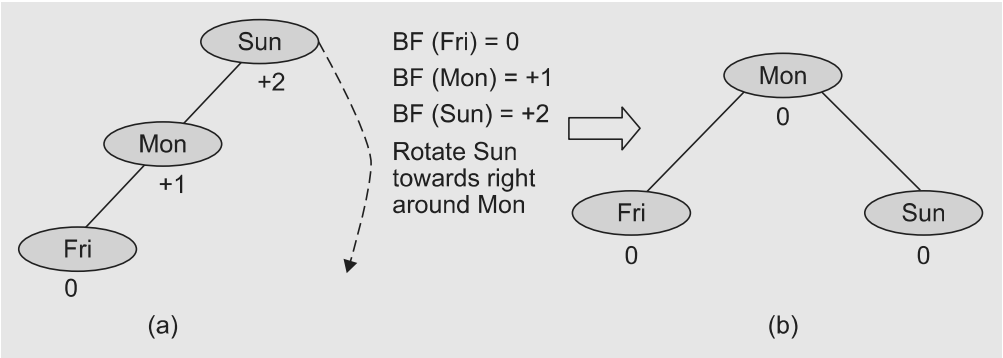


Fig. 10.13 Balancing a tree by rotating towards right (a) Unbalanced tree (b) Balanced tree

Similarly, the rotation can be performed towards left as shown in Fig. 10.14.

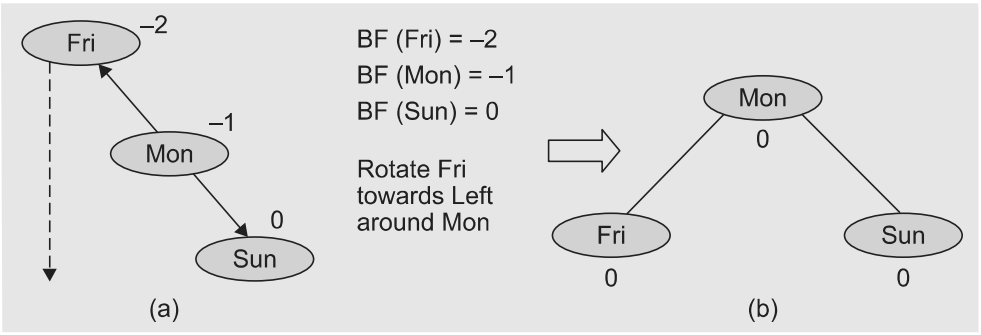


Fig. 10.14 Balancing a tree by rotating towards left (a) Unbalanced tree (b) Balanced tree

Let X be an inserted node and A be an unbalanced node after insertion whose BF = ±2. It depends on the scenario whether a rotation should be performed towards left or right. An unbalanced tree is balanced using one of the following four ways: (a) Left of left (LL) (b) Right of right (RR) (c) Left of right (LR) (d) Right of left (RL).

Case 1: LL (Left of Left) Consider the BST in Fig. 10.15. Note that the nodes drawn as squares represent subtrees.

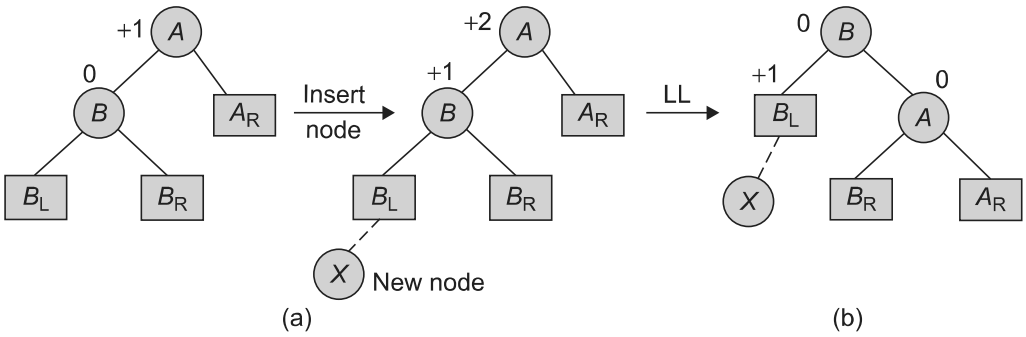


Fig. 10.15 Case LL for unbalanced tree due to insertion at left of left of a node
(a) Unbalanced tree due to increase in height of B_L (b) Balanced tree

Suppose the node A in Fig. 10.15 becomes unbalanced when X is inserted to the left of left of A, that is, in the left subtree of the left subtree of A, then the rule of rotation as in Fig. 10.15 should be used for balancing.

As shown in Fig. 10.15, B_L is to the left of the left child of A. When the height of B_L increases, then node A becomes unbalanced. To rebalance the tree, node B becomes the root of the subtree. As it is a BST and $B_L < B$ in a rebalanced tree, B_L remains the left child of B. As $B < A$, node A becomes the right child of B. As $A < A_R$, A_R remains the right child of A. Now, the question is where to place B_R. Because $B_R > B$, it will be placed to the right of B. However, $B_R < A$, so it will be placed to the left of A. Hence, B_R becomes the left child of A. Thus, in Fig.10.15, right rotation of A is performed around the node B.

Case 2: RR (Right of Right) When X is inserted to the right of right of A, that is, in the right subtree of the right subtree of A, the rule of rotation as in Fig. 10.16 should be used for balancing.

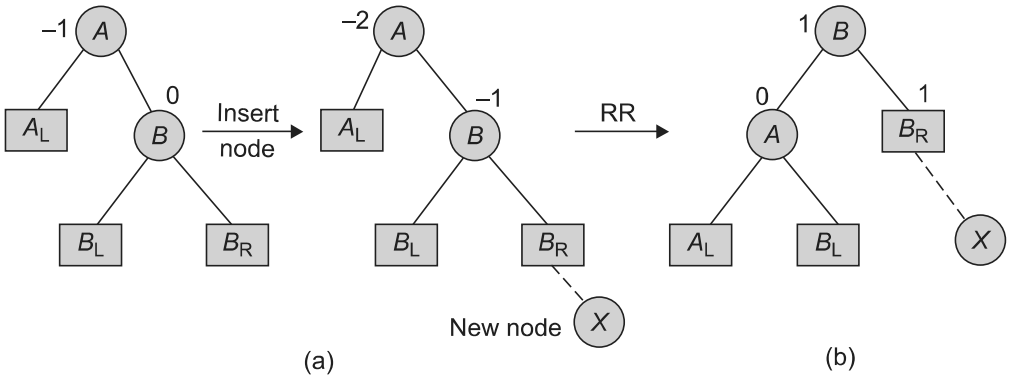


Fig. 10.16 Case RR for unbalanced tree due to insertion at right of right of a node
(a) Unbalanced tree due to increase in height of B_R (b) Balanced tree

As shown in Fig. 10.16, B_R is to the right of the right child of A. When the height of B_R increases, then node A becomes unbalanced. To rebalance the tree, node B is made the root of the tree. As it is a BST and $B_R > B$ in the rebalanced tree, B_R remains the right child of B. As $A < B$ in the rebalanced tree, A becomes the left child of B. As $A_L < A$, A_L remains the left child of A. Now, the question is where to place B_L. As $B_L < B$, it will be on the left side of B. Since $B_L >$ A, it will be to the right of A, and B_L becomes the right child of A. In other words, B_L is less than B and greater than A. Thus, it should be inserted in the left of B and right of A. Thus, in Fig. 10.16, left rotations of A are performed around B.

Case 3: LR (Left of Right) When X is inserted to the left of right of A, that is, in the left subtree of the right subtree of A, the rules of rotation as in Fig. 10.17 should be used for balancing.

In Fig. 10.17, the case LR is depicted using three different scenarios. Scenario 1 depicts a simplified tree where A has no right child, B has no left child, and C has no

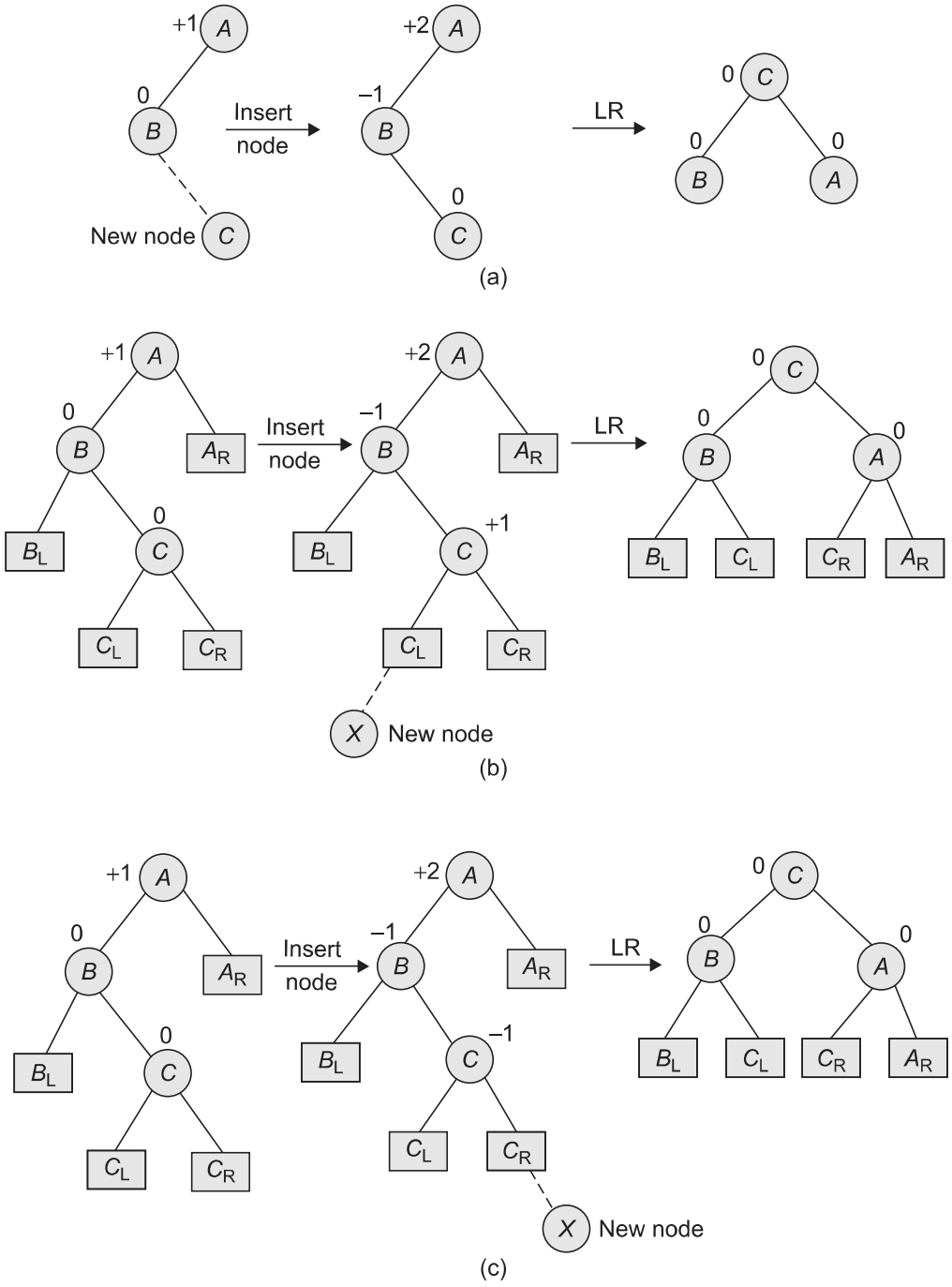


Fig. 10.17 Case LR for unbalanced tree due to insertion in left of right of a node
(a) LR rotation (b) Scenario 2—LR rotation after insertion of new node
(c) Scenario 3

children. Node A is unbalanced due to the insertion of C to the right of the left child of A. To rebalance the tree, C becomes the root of the subtree. As $C < A$, A becomes the

right child of C. As $B < C$, B remains the left child of C and thus remains at its position in the subtree.

In scenario 2, node A is unbalanced due to the increase in the height of C_L. Figure 10.17(c) depicts how node A is unbalanced due to the increase in the height of C_R. For both the cases, solution is the same. In the rebalanced trees, node C becomes the root of the subtree. As $C < A < A_R$, A becomes the right child of C and A_R remains the right child of A. In addition, as $B_L < B < C$, B becomes the left child of C and B_L remains the left child of B. Till this step, the subtree looks as shown in Fig.10.18.

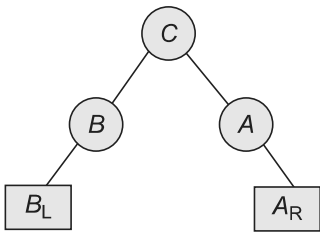


Fig. 10.18 Partial subtree in the LR case

Now, the question is where to place C_L and C_R. As $C_L < C$, it will be placed on the left of C. Since $C_L > B$, it becomes the right child of B. Similarly, $C_R > C$, so it will be inserted to the right of C. Since $C_R < A$, it becomes the left child of A.

To summarize the case LR, node C becomes the root of the rebalanced subtree. As $C_L < C$ and $B < C$, they are placed on the left of C. As $C_R > C$ and $A > C$, they are placed on the right of C.

Case 4: RL (Right of left) When X is inserted to the right of left of A, that is, in the right subtree of the left subtree of A, the rules of rotation as in Fig. 10.19 should be used for balancing.

In Fig. 10.19, the case RL is depicted using three different scenarios. In scenario 1, it is considered that A has no left child, B has no right child, and C has no children. Hence, Fig. 10.19(a) looks simplified. Here, node A becomes unbalanced due to the insertion of node C. To rebalance it, node C becomes the root of the subtree. As $A < C$, A becomes the left child of C and remains at the same position in the rebalanced tree.

Figure 10.19(b) depicts how node A becomes unbalanced due to the increase in the height of C_L. In Fig. 10.19(c), scenario 3 depicts how node A becomes unbalanced due to the increase in the height of C_R. In both Scenarios 2 and 3, solution is the same. To rebalance the subtrees, node C becomes the root of the subtrees. As $A_L < A < C$, A_L remains the left child of A, and A becomes the left child of C. As $C < B < B_R$, B_R remains the right child of B, and B becomes the right child of C. Upto this step, the subtree looks as shown in Fig.10.20.

Now, the question is where to place C_L and C_R. As $C_L < C$, it will be inserted to the left side of C. As $C_L > A$, it becomes the right child of A. Similarly, as $C_R > C$, C_R becomes the right child of C. However, $C_R < B$; hence, it becomes the left child of B.

To summarize the case RL, node C becomes the root of the rebalanced subtree. As $A < C$ and $C_L < C$, they are placed on the left of C. In addition, $B > C$ and $C_R > C$; hence, they are placed to the right of C.

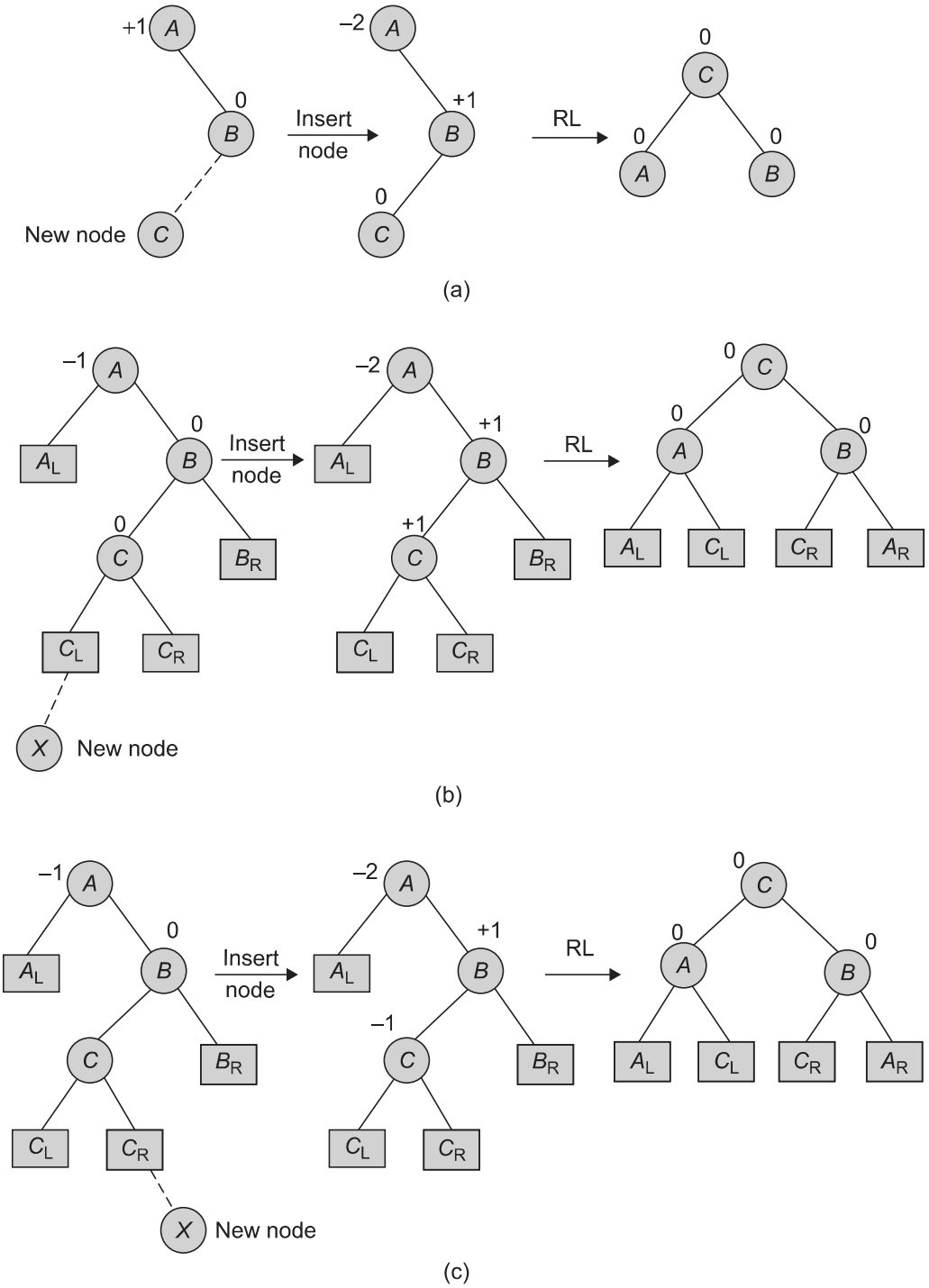


Fig.10.19 Case RL for unbalancing due to insertion in right of left of a node
(a) Scenario 1 (b) Scenario 2 (c) Scenario 3

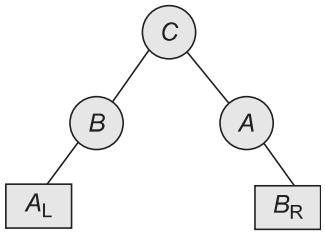


Fig. 10.20 Partial subtree in case RL

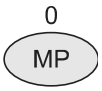
Let us see an example to illustrate the process involved in maintaining a height-balanced BST in Example 10.4.

EXAMPLE 10.4 Consider a list of subjects studied in a computer engineering course. Assume that the insertions are made in the following order:

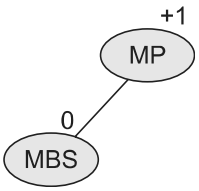
MP, MBS, MMT, NCP, AI, ACA, OOCS, DC, DS, OOP, OOMD

Solution: The steps of insertions and the brief explanations are listed and illustrated as follows.

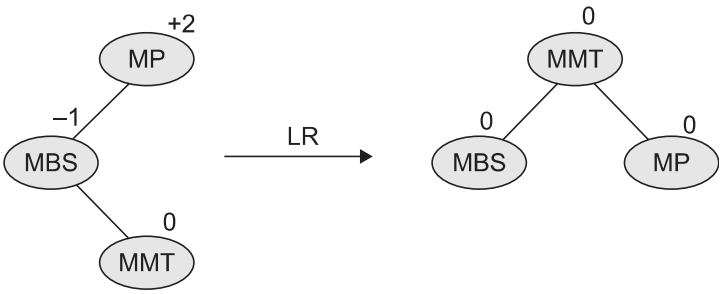
(a) Insert MP.



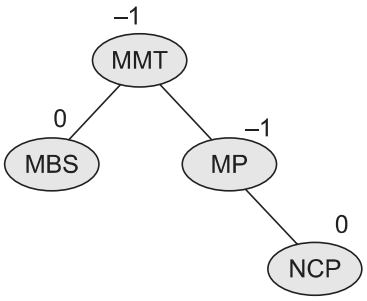
(b) Insert MBS.



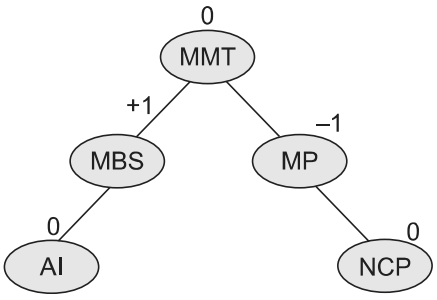
(c) Insert MMT. In the BST, MMT is placed to the right of left of MP, and MP is unbalanced. Hence go for LR rotation for rebalancing.



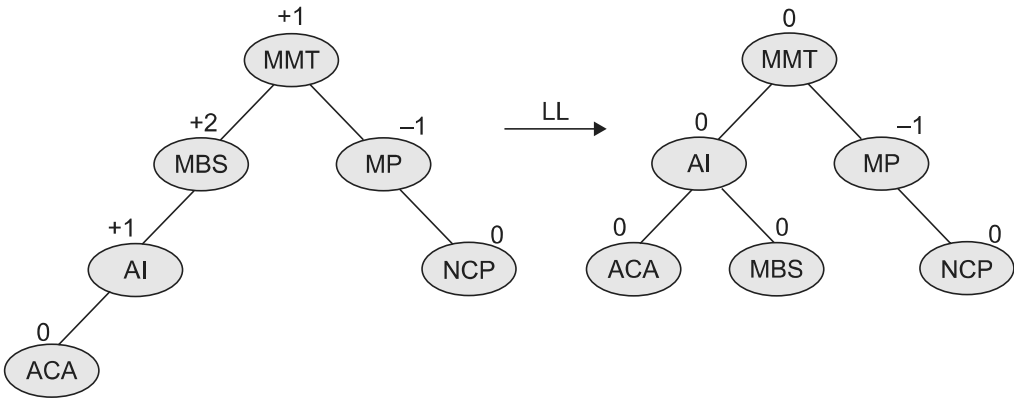
(d) Insert NCP.



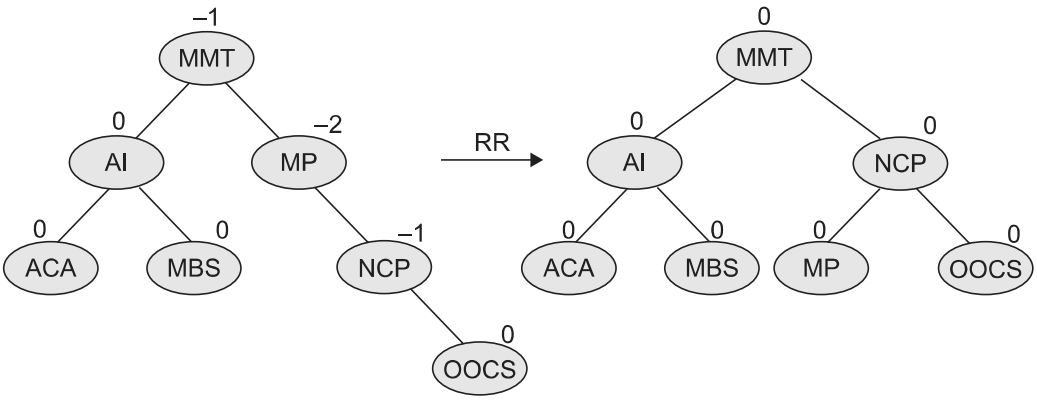
(e) Insert AI.



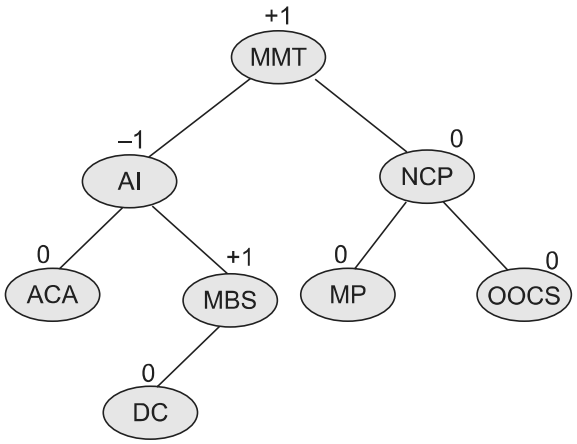
(f) Insert ACA. ACA is placed to the left of left of MBS, and MBS is unbalanced. Hence use LL rotation to rebalance it.



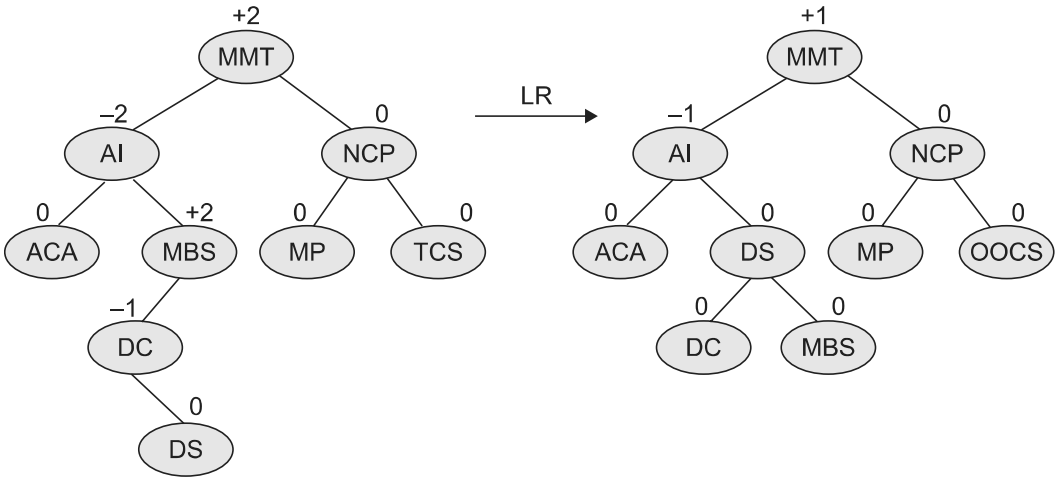
(g) Insert OOCS. OOCS is placed to the right of right of MP and now MP is unbalanced. Hence use RR rotation for rebalancing.



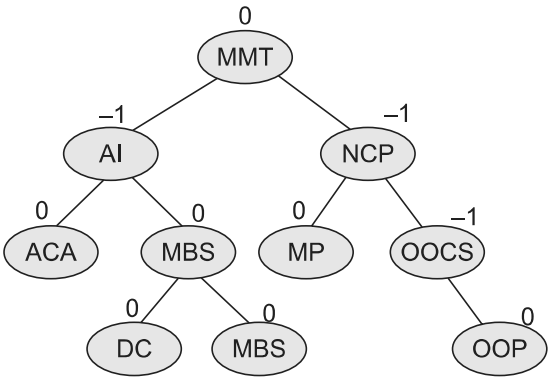
(h) Insert DC.



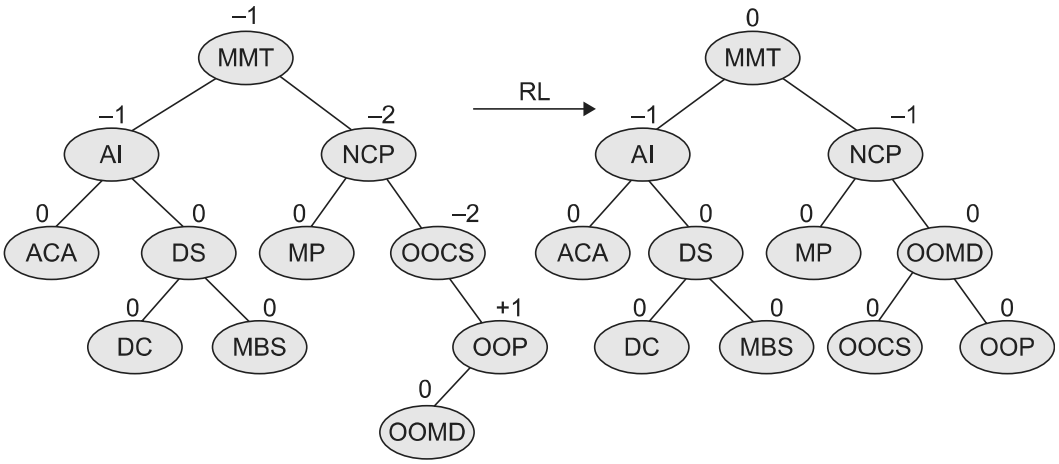
(i) Insert DS. DS is placed to the right of left of MBS due to which MBS is unbalanced. Hence use rotation LR for rebalancing.



(j) Insert OOP.



(k) Insert OOMD. OOMD is placed to the left of right of OOCS because of which OOCS is unbalanced. Hence use RL rotation for rebalancing.



10.3.1 Implementation of AVL Technique

Example 10.4 demonstrates the working procedure to balance a BST. Let us write an algorithm for it. In general, a node in a tree stores data of and pointers to the left and right children. In an AVL tree, each node stores these three fields. To simplify the work, we can store the height of its subtree in each node. Hence, we will define each AVL tree node having as four fields.

Consider an AVL tree node that has the following structure:

```
class AVLNode
{
   KeyType key;
```

```
    AVLNode *Left, *Right;
    int height;
};
```

Assume that `getNode()` is a function which allocates memory for a new `AVLNode`, initializes all the fields, and returns a pointer to it.

Let `height(n)` be a function that returns the height of the subtree with root *n*, otherwise returns –1 if `null`. Let `balancefactor(n)` be a function which returns the balance factor of node *n* in its tree. Note that in Example 10.3 we considered the following: when the balance factor of a node *n* is +2, the tree is unbalanced due to the increase in the height of the left subtree and there are only two possibilities, either go for LL or LR. When the balance factor of node *n* is –2, then the tree is unbalanced due to the increase in the height of the right subtree and there are only two possibilities, either go for RR or RL.

Let us write a function `insert()` which will insert a given key in the AVL tree at its proper position, and rebalance the tree if needed using one of the four rotations: `LL`, `RR`, `LR`, `RL`. This is illustrated in Program Code 10.2.

PROGRAM CODE 10.2

```
AVLNode *AVLNode :: insert(int NewKey, AVLNode *root)
{
   AVLNode *NewNode;
   int lh, rh;
   root->height = height(root);
   if(root == null)
   {
      NewNode = new AVLNode;
      NewNode->key = NewKey;
      NewNode->left = null;
      NewNode->right = null;
      root = NewNode;
   }
   else
   {
      if(NewKey < root->key)
      {
         root->left = insert(NewKey,root->left);
         if(balancefactor(root) == 2)
         {
            // Tree is unbalanced due to increase
```

```
            // in height of left subtree
            if(NewKey < root->left->key)
            {
               cout << "\n LL rotation \n";
               root = LL(root);
            }
            else
            {
               cout << "\n LR rotation \n";
               root = LR(root);
            }
         }
      }
      else if(NewKey > root->key)
      {
         root->right = insert(NewKey, root->right);
         if(balancefactor(root) == -2)
         {
            // Tree is unbalanced due to increase
            // in height of right subtree
            if(NewKey > root->right->key)
            {
               cout << "\n RR rotation \n";
               root = RR(root);
            }
            else
            {
               cout << "\n RL rotation \n";
               root = RL(root);
            }
         }
      }
      else
         cout << "Duplicate key";
   }
   // After insertion, modify field height of the root
   root->height = height(root);
   return root;
}
```

We now know the four rules of rotation. Let us write a code for Case 1: LL. Consider the scenario as in Fig. 10.21 where Program Code 10.3 simulates its operations.

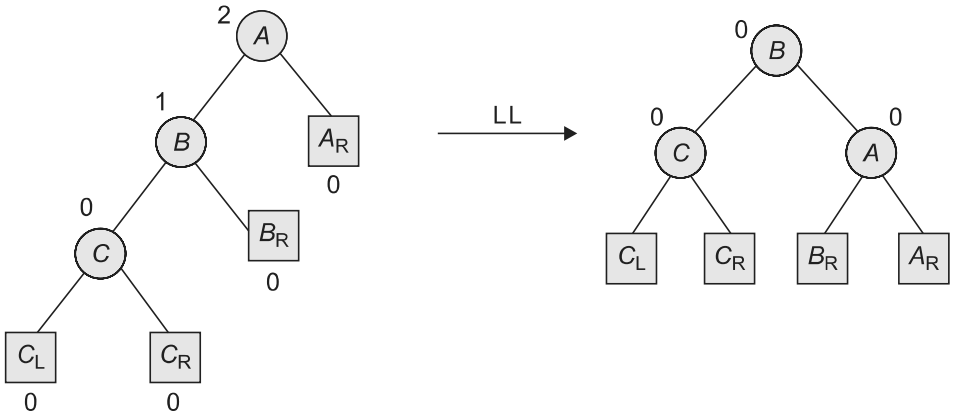


Fig. 10.21 Scenario for case LL

PROGRAM CODE 10.3

```
// rotation: Left
AVLNode *AVLNode :: Left(AVLNode *A)
//function is called with unbalanced node as a parameter
{
   AVLNode *B;
   B = A->right;
   A->right = B->left;
   B->left = A;
   A->height = height(A);
   B->height = height(B);
   return B;       // Set new root to B
}

// Rotation : Right
AVLNode *AVLNode::Right(AVLNode *A)
{
   AVLNode *B;
   B = A->left;
   A->left = B->right;
   B->right = A;
   A->height = height(A);
   B->height = height(B);
```

```
    return B;       // Set new root to B
}
// Case 1 of rotation : LL
AVLNode *AVLNode :: LL(AVLNode *root)
{
    root = Right(root);
    return root;
}

// Case 2 of rotation : RR
AVLNode *AVLNode :: RR(AVLNode *root)
{
    root = Left(root);
    return root;
}
```

Similarly, the function `RR()` can be written for Case 2. Program Code 10.3 shows the simulation of RR. Now, consider Case 3: LR (Fig. 10.22a) and Case 4: RL (Fig. 10.22b).

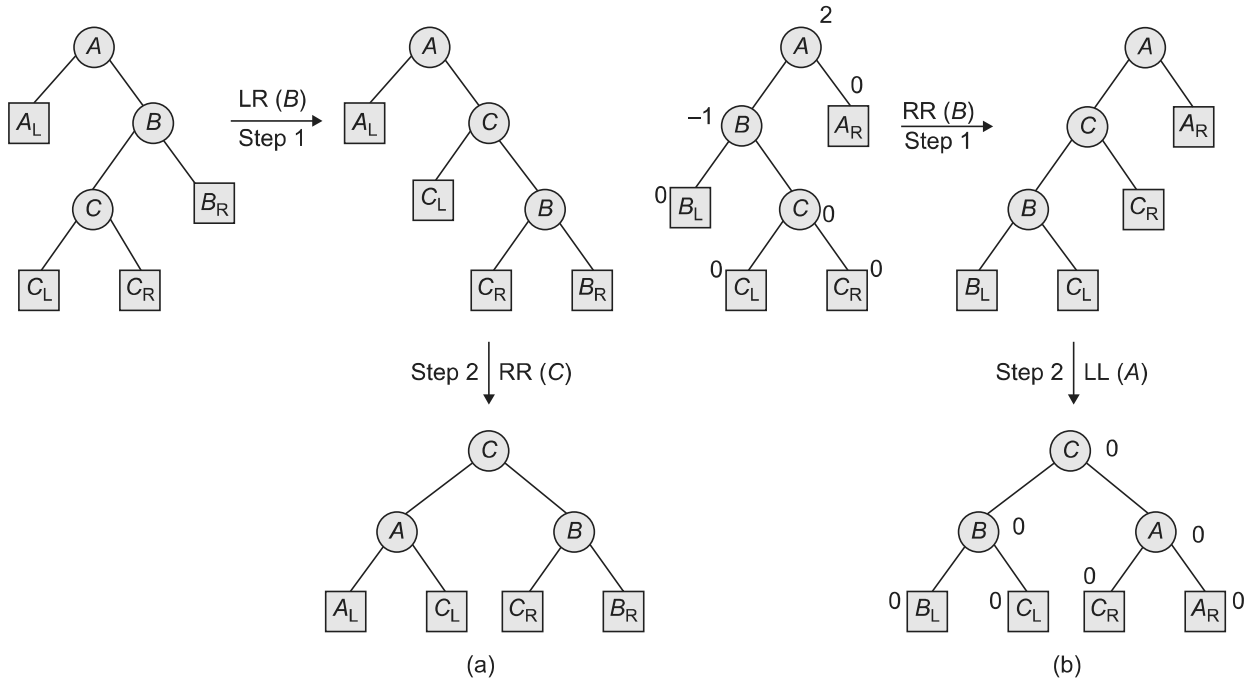


Fig. 10.22 Scenario for case (a) LR (b) RL

We can use the functions `LL()` and `RR()` to write the `LR()` and `RL()` functions as in Program Code 10.4.

PROGRAM CODE 10.4

```
// Case 3 of rotation:LR
AVLNode *AVLNode :: LR(AVLNode *root)
{
   root->left = Left(root->left);
   root = Right(root);
   return root;
}

//* Case 4 of rotation : RL
AVLNode *AVLnode :: RL(AVLNode *root)
{
   root->right = Right(root->right);
   root = Left(root);
   return root;
}
```

Similarly, the function `RL()` can be written for case 4. Program Code 10.4 depicts the simulation of it.

10.3.2 Insertions and Deletions in AVL Tree

Insertions and deletions in AVL tree are performed as in BSTs and followed by rotations to correct the imbalances in the outcome trees. In the case of insertions, one rotation is sufficient. In the case of deletions, utmost O(log$n$) rotations are needed from the first point of discrepancy going up towards the root.

Figure 10.23 demonstrates the deletion of a node in a given AVL tree. The original tree is shown in Fig. 10.23(a). Figure 10.23(b) shows the tree after deletion of node 4. Note that in Fig. 10.23(c), the imbalance at node 3 implies an LL rotation around node 2 and the imbalance at node 5 in Fig. 10.23(d) implies a n RR rotation around node 8.

Program Code 10.5 illustrates a function to delete an element from AVL tree.

Examples 10.5–10.7 illustrate the construction of an AVL tree for different sets of data.

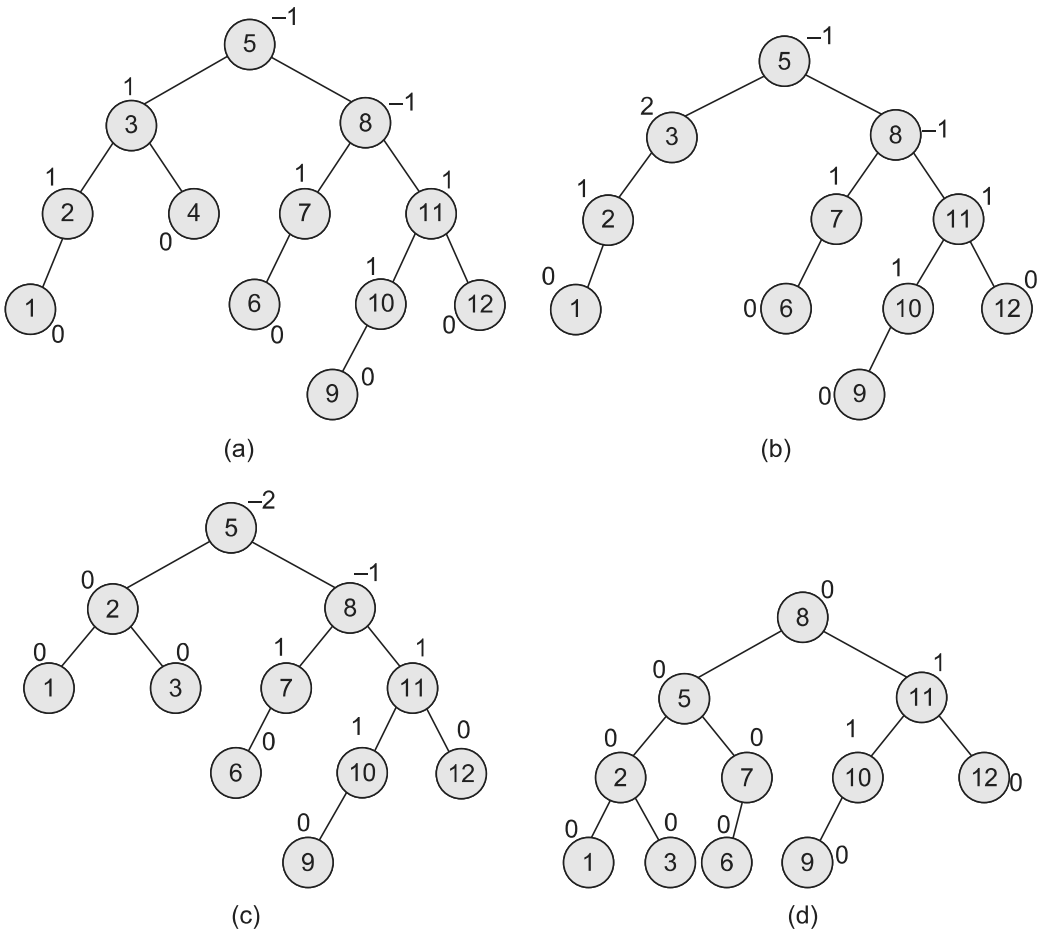


Fig. 10.23 Deletion of a node in AVL tree (a) Original tree (b) After deletion of 4 (*c*) LL rotation around node 2 (d) RR rotation around node 8

PROGRAM CODE 10.5

```
//Function to delete an element from AVL tree
AVLNode *AVLNode :: del(AVLNode *root,int dval)
{
   AVLNode *temp;
   if (root != null)
   {
      if (dval < root->key)
      {
         root->left = del(root->left,dval);
         if(balancefactor(root) == -2)
         {
            if(balancefactor(root->right) <= 0)
```

```
            {
               cout << "\n RR rotation \n";
               root = RR(root);
            }
            else
            {
               cout << "\n RL rotation \n";
               root = RL(root);
            }
         }
      }
      else if(dval > root->key)
      {
         root->right = del(root->right, dval);
         if(balancefactor(root) == 2)
         {
            if(balancefactor(root->left) >= 0)
            {
               cout << "\n LL rotation \n";
               root = LL(root);
            }
            else
            {
               cout << "\n LR rotation \n";
               root = LR(root);
            }
         }
      }
      else
      {
         if(root->right == null)      // No right tree
            return(root->left);
         else
         {
            // find leftmost of right
            temp = root->right;
            while(temp->left != null)
               temp = temp->left;
            root->key = temp->key;
            temp->right = del(root->right, temp->key);
            if(balancefactor(root) == 2)
            {
               if(balancefactor(root->left) >= 0)
```

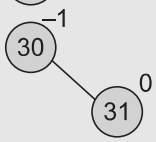
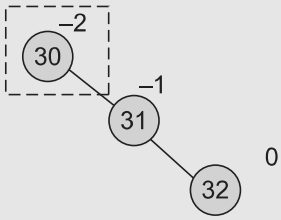
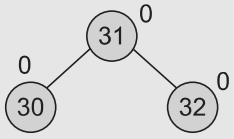
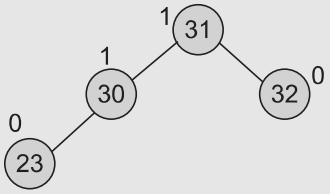
```
                root = LL(root);
             else
                root = LR(root);
          }
       }
    }
    else
       return null;
    // Update height of root node
    root->height = height(root);
    return(root);
 }
```

EXAMPLE 10.5 Construct an AVL tree for the following data:

30, 31, 32, 23, 22, 28, 24, 29, 26, 27, 34, 36

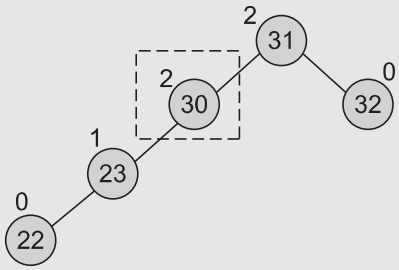
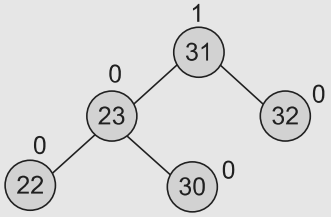
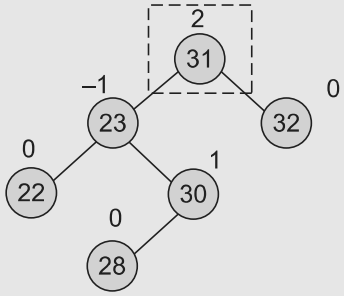
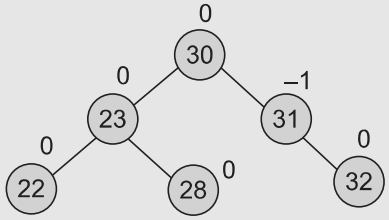
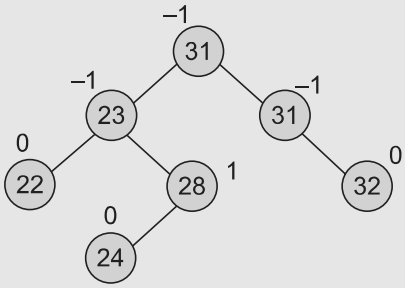
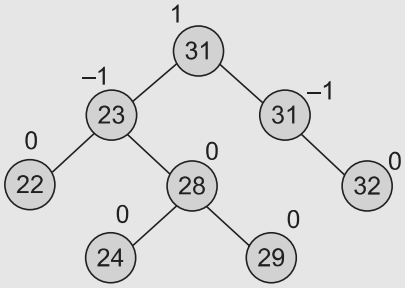
Solution Let us solve and show the balance factor and the type of rotation performed (if any) at each insertion. Table 10.7 demonstrates the same through the steps stated here.

Table 10.7 Construction of AVL tree for Example 10.5

Data inserted	AVL tree after insertion of BF	Rotation performed	Rebalanced AVL tree
30		→	No balancing required
31		→	No balancing required
32		RR →	
23		→	No balancing required

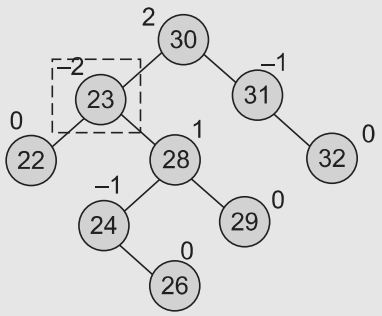
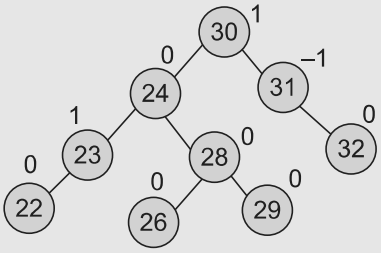
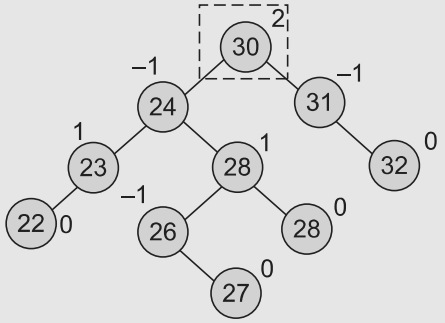
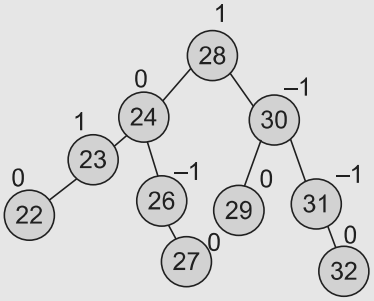
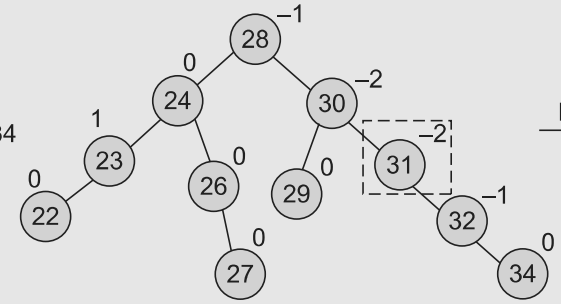
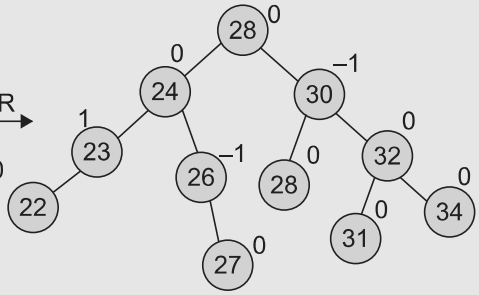
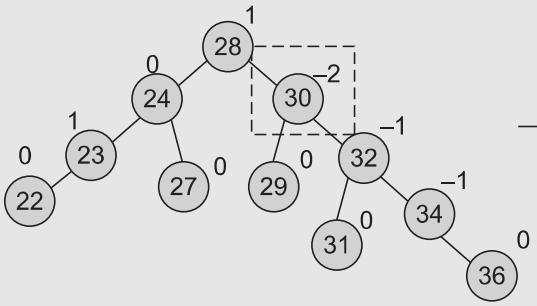
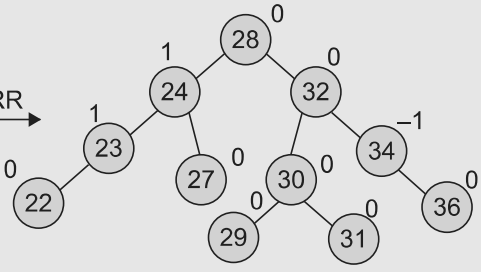
(Continued)

Table 10.7 (Continued)

Data inserted	AVL tree after insertion of BF	Rotation performed	Rebalanced AVL tree
22		LL	
28		LR	
24		→	No balancing required
29		→	No balancing required

(Continued)

Table 10.7 (Continued)

Data inserted	AVL tree after insertion of BF	Rotation performed	Rebalanced AVL tree
26		RL	
27		LR	
34		RR	
36		RR	

EXAMPLE 10.6 Construct an AVL tree for the following data:

STA, ADD, LDA, MOV, JMP, TRIM, XCHG, MVI, DIV, NOP, IN, JNZ

Solution Figure 10.24 demonstrates the steps involved to construct the AVL tree for the given sequence.

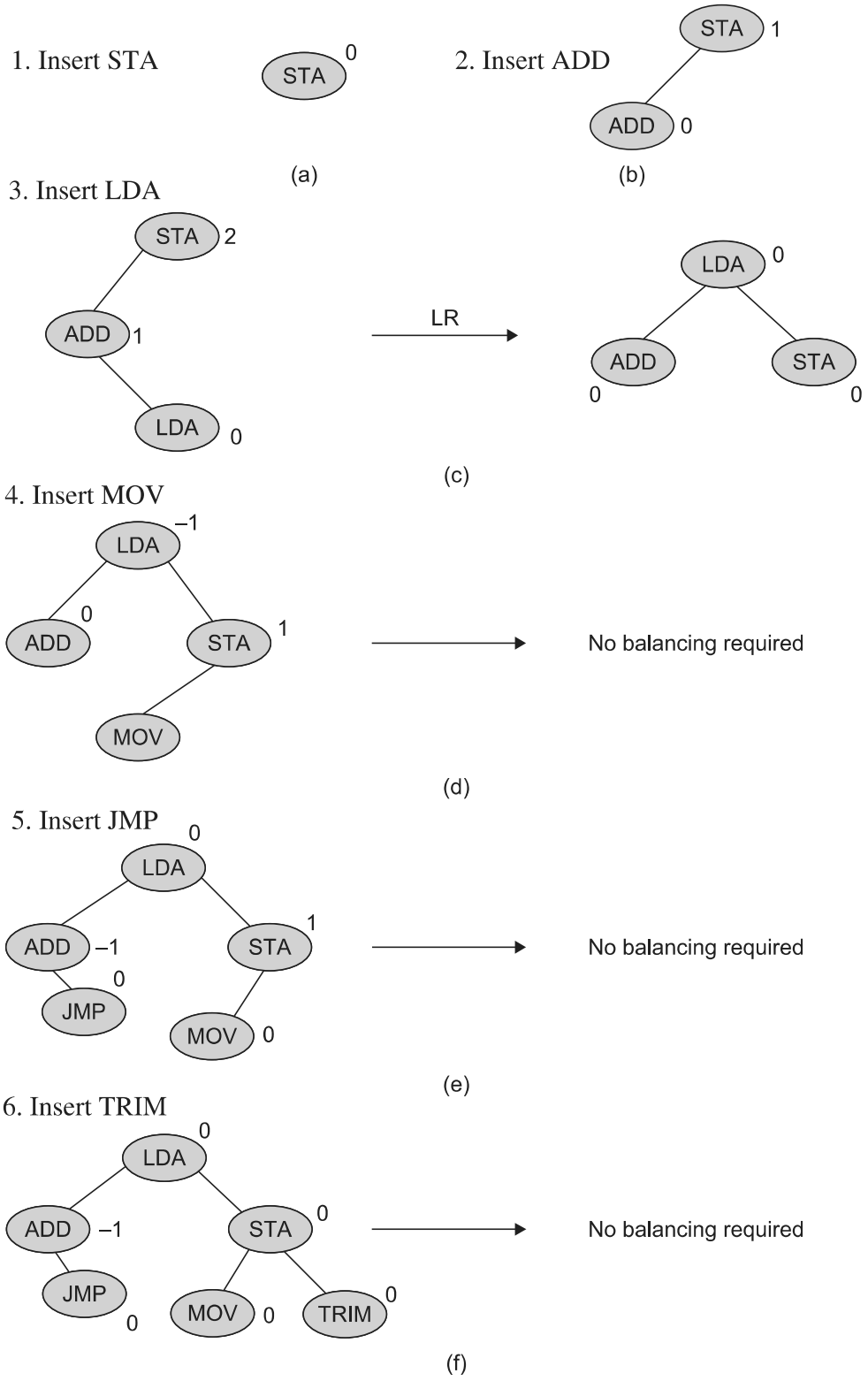
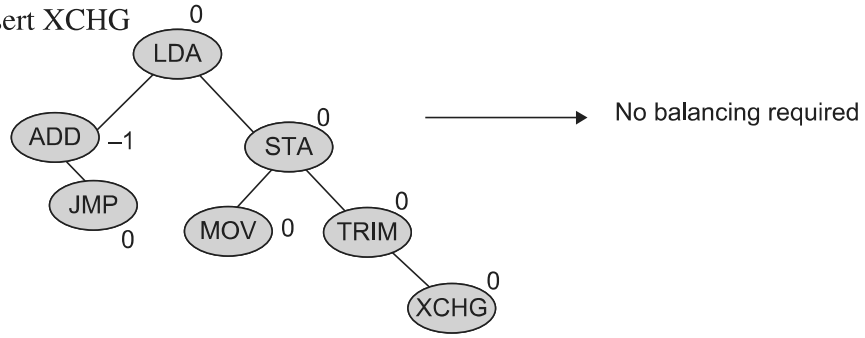


Fig.10.24 Construction of AVL tree for Example 10.6 (a) Key = STA (b) Key = ADD (c) Key = LDA (d) Key = MOV (e) Key = JMP (f) Key = TRIM

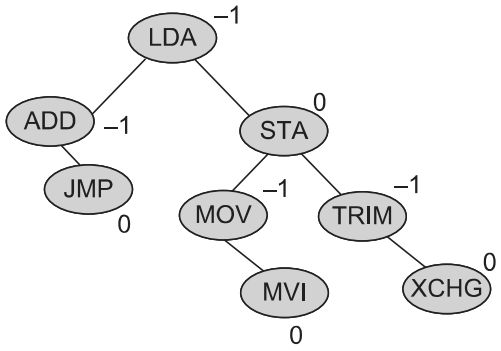
(Continued)

7. Insert XCHG



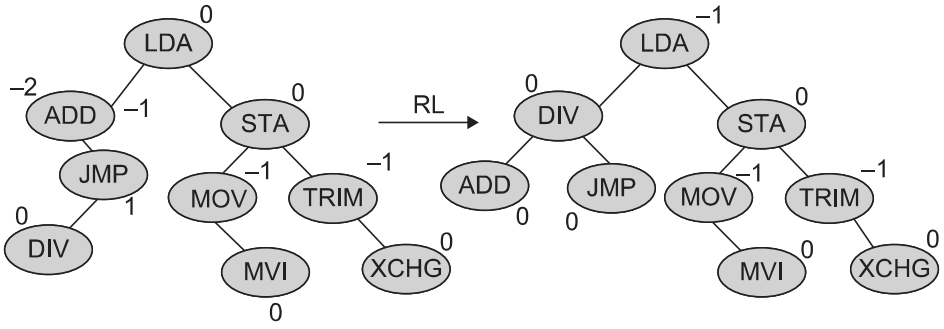
(g)

8. Insert MVI



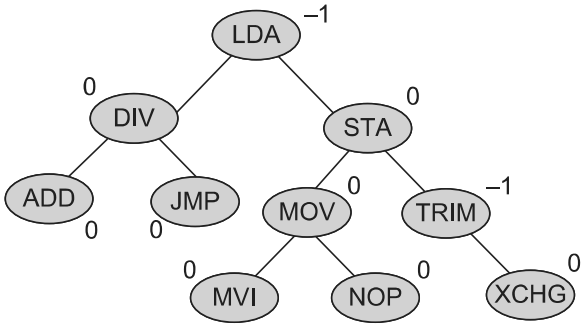
(h)

9. Insert DIV



(i)

10. Insert NOP

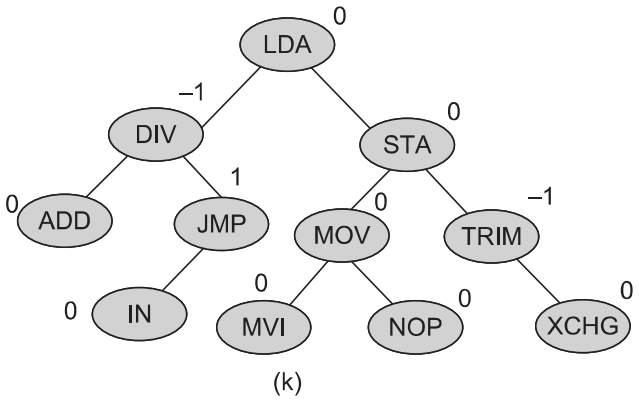


(j)

Fig.10.24 (Continued) (g) Key = XCHG (h) Key = MVI (i) Key = DIV (j) Key = NOP

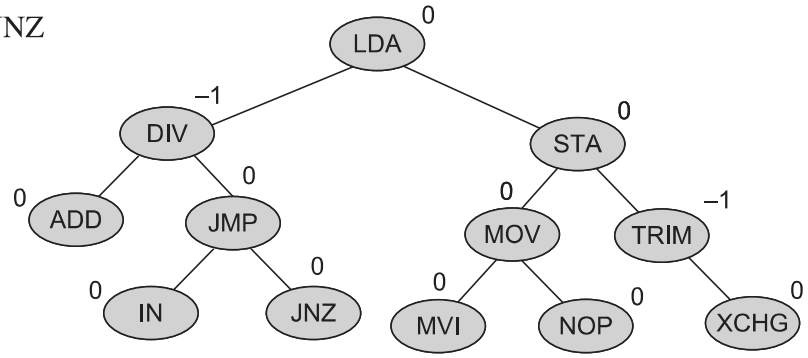
(Continued)

11. Insert IN



(k)

12. Insert JNZ



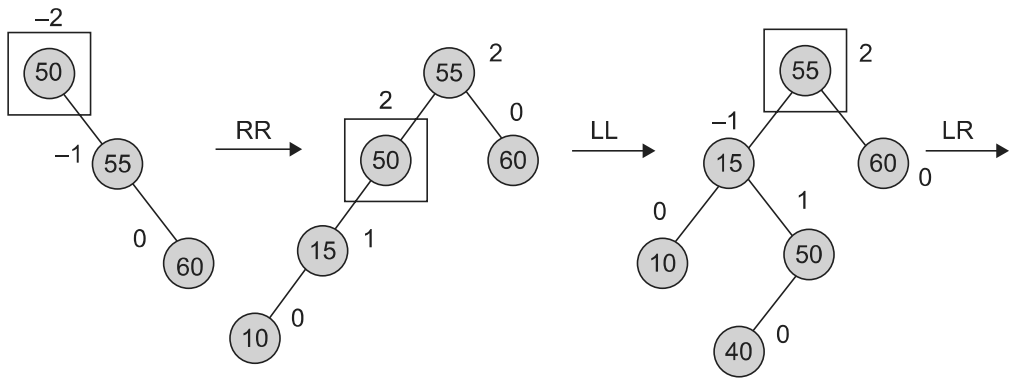
(l)

Fig.10.24 (Continued) (k) Key = IN (l) Key = JNZ

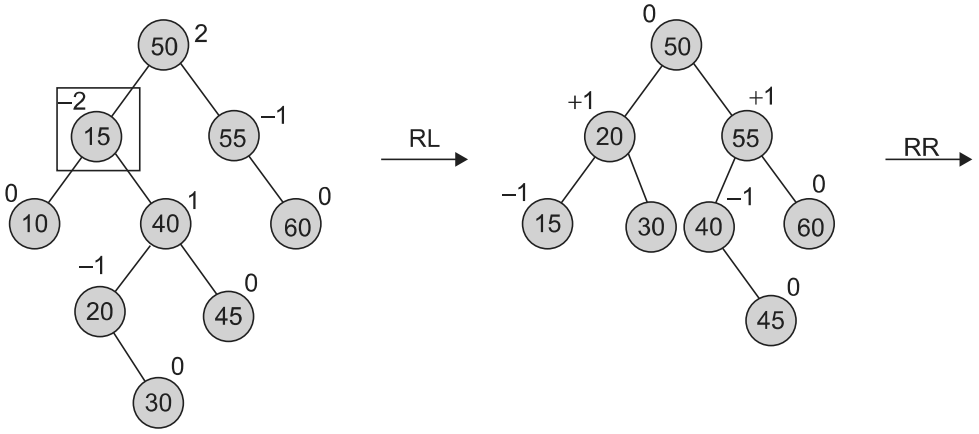
EXAMPLE 10.7 Construct an AVL tree for the set of keys = {50, 55, 60, 15, 10, 40, 20, 45, 30, 70, 80}.

Solution Figure 10.25 demonstrates the construction of an AVL tree for the given set of keys.

1. After insertion of (50, 55, and 60):



2. After insertion of (15, 10, 40, 20, 45, and 30):



3. After insertion of 70 and 80:

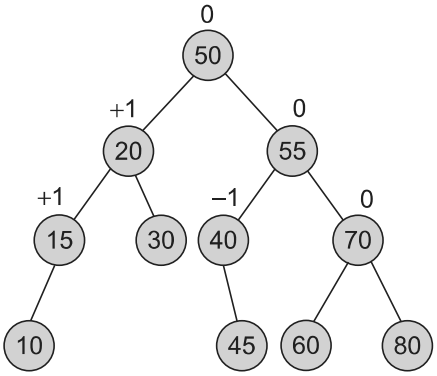


Fig.10.25 Construction of AVL tree for Example 10.7

RECAPITULATION

- Search trees are of great importance in an algorithm design.
- It is always desirable to keep the search time of each node in a tree minimal.
- OBST maintains the optimal average search time of all the nodes.
- In an AVL tree, after insertion of each node, it is checked whether the tree is balanced or not. If unbalanced, it is rebalanced immediately.
- Rebalancing of AVL tree is performed using one of the four rotations: LL, RR, LR, RL.
- AVL trees work by ensuring that all nodes of the left and right subtrees differ in height by utmost 1, which ensures that a tree cannot get too deep.
- Compilers use hash tables to keep track of the declared variables in a source code called as a symbol table.
- Unbalancing of an AVL tree due to insertion is removed in a single rotation. However, unbalancing due to the deletion may require multiple steps for balancing.

KEY TERMS

AVL tree An AVL tree is a BST where the heights of the left and right subtrees of the root differ by utmost 1. In addition, the left and right subtrees of the root are again AVL trees.

Keyed table Keyed tables are a very useful data structure. They store <key, information> pairs with no additional logical structure.

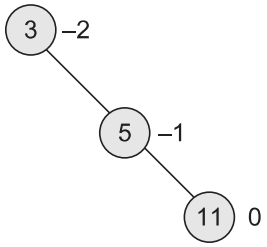
OBST Optimal binary search tree is a binary search tree having an average search time of all keys as the optimal value.

Symbol table While compilers and assemblers scan a program, each identifier must be examined to determine if it is a keyword. This information concerning the keywords and identifier in a programming language is stored in a table called symbol table.

EXERCISES

Multiple choice questions

1. Which of the following is true?
   (a) The cost of searching an AVL tree is O(log$n$) but that of a BST is O(n).
   (b) The cost of searching an AVL tree is O(log$n$) but that of a complete binary tree is O($n$log$n$).
   (c) The cost of searching a BST is O(log$n$) but that of an AVL tree is O(n).
   (d) The cost of searching an AVL tree is O(log$n$) but that of a BST is O(n)
2. In the following AVL tree, the structure has to be balanced, so we have to rotate it



   (a) clockwise
   (b) counter clockwise
   (c) in both the directions
   (d) none of the above
3. What is the maximum height of an AVL tree with seven nodes?

Note: Assume that the height of a tree with a single node is 0.

   (a) 2
   (b) 3
   (c) 4
   (d) 5
4. The worst case height of an AVL tree with n nodes is
   (a) 1.44log(n + 2)
   (b) 2.44log(n + 2)
   (c) 3.44log(n + 2)
   (d) 1.44log(n + 2)
5. What will be the time complexity for inserting a node into an AVL tree?
   (a) O(n)
   (b) O(log$n$)
   (c) n
   (d) n^2
6. Which of the following properties of OBST is true?
   (a) The left subtree of a node contains only the nodes with keys less than the node's key.
   (b) The right subtree of a node contains only the nodes with keys greater than the node's key.
   (c) Both the left and right subtrees must also be BSTs.
   (d) All of the above
7. To find the cost of the given OBST,