7 TREES

OBJECTIVES

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

- Hierarchical representation of data using trees
- Binary search trees (BSTs) that allow both rapid retrievals by key and inorder traversals
- The use of trees as a flexible data structure for solving a wide range of problems

In computer science, a *tree* is a widely used data structure that emulates a tree structure with a set of linked nodes. Trees are used popularly in computer programming. They can be used for improving database search times (binary search trees, AVL trees, red–black trees), in game programming (minmax trees, decision trees, path finding trees), 3D graphics programming (binary trees, quadtrees, octrees), arithmetic scripting languages (arithmetic precedence trees), data compression (Huffman trees), and even file systems (btrees, sparse indexed trees, trie trees). Let us learn about trees in this chapter.

7.1 INTRODUCTION

Let us first revise the classification of data structures as *linear* and *non-linear*. A data structure is said to be *linear* if its elements form a sequence or a linear list. In a linear data structure, every data element has a unique successor and a unique predecessor. There are two basic ways of representing linear structures in memory. One way is to have the relationship between the elements by means of pointers (links), called as *linked lists*. Another way is using sequential organization, that is, *arrays*.

Non-linear data structures are used to represent the data containing hierarchical or network relationship between the elements. Trees and graphs are examples of non-linear data structures. In non-linear data structures, every data element may have more than one predecessor as well as successor. Elements do not form any particular linear sequence.

Non-linear data structures are capable of expressing more complex relationships than linear data structures. In general, wherever the hierarchical relationship among data is to be preserved, the tree is used. Well-known examples of such structures are family trees, hierarchy of positions in organization, and so on. *Tree*, a non-linear data structure, is a

means to maintain and manipulate data in many applications. Consider the following example:

The operating system of a computer system organizes files into directories and subdirectories. Directories are also referred to as *folders*. The operating system organizes folders and files using a tree structure as in Fig. 7.1.

A folder contains other folders (subfolders) and files. This can be viewed as the tree drawn in Fig. 7.1. Note that the root here is desktop. The common uses of trees include the following:

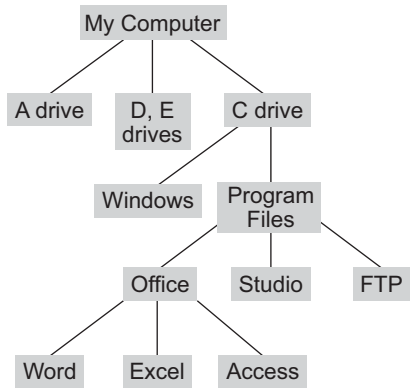


Fig. 7.1 Folder and subfolders organization

1. Manipulating hierarchical data
2. Making information easily searchable
3. Manipulating sorted lists of data

A tree is a graph called the directed acyclic graph. So let us first discuss the basic terminology related to trees.

7.1.1 Basic Terminology

We should first learn about a general graph because trees can be viewed as restricted graphs. A graph G consists of a non-empty set V, a set E, and a mapping from the set E to set V. Here, V is the set of *nodes*, also called as *vertices points*, of the graph, and E is the set of edges of the graph. For *finite graphs*, V and E are finite. We can represent them as $G = (V, E)$.

Adjacent Nodes

If an edge $e \in E$ is associated with a pair of nodes (a, b) where $a, b \in V$, then it is said that the edge e joins or connects the nodes a and b. Any two nodes that are connected with an edge are called as *adjacent nodes*.

Directed and Undirected Graphs

In a graph $G(V, E)$, an edge that is directed from one node to another is called a *directed edge*, whereas an edge that has the no specific direction is called an *undirected edge*. A graph where every edge is directed is called as a *directed graph* or *diagraph*. A graph where every edge is undirected is called as an *undirected graph*. If some of edges are directed and some are undirected in a graph, the graph is called as a *mixed graph*.

A city map showing only the one-way streets is an example of a directed graph where the intersections are vertices and the edges are streets. A map showing only the two-way streets is an example of an undirected graph, and a map showing all the one-way and two-way streets is an example of a mixed graph.

Let (V, E) be a graph and let $e \in E$ be a directed edge associated with the ordered pair of nodes (a, b). Then, the edge e is said to be initiating or originating in the node a and terminating or ending in the node b. The nodes a and b are also called the *initial* and *terminal nodes* respectively, of the edge e. An edge $e \in E$ that joins the nodes a and b, be it directed or undirected, is said to be *incident* to the nodes a and b, respectively.

An edge of a graph that joins a node to itself is called a *loop* (sling). Note that this loop is different from the loop in a program. The direction of the loop has no significance.

Parallel Edges and Multigraph

The graph given in Fig. 7.2(a) has only one edge between any pair of nodes. In the directed edges, the two possible edges between the pair of nodes that are opposite in direction are considered distinct. In some directed as well as undirected graphs, there may exist more than one edge incident to the same pair of nodes, say a and b.

In Fig. 7.2(b), the edges e_1, e_2, and e_3 are incident to vertices a and b. Such edges are called as *parallel edges*. Here, e_1, e_2, and e_3 are three parallel edges. In addition, e_5 and e_6 are two parallel edges. Any graph that contains parallel edges is called a *multigraph*. On the other hand, a graph that has no parallel edges is called a *simple graph*.

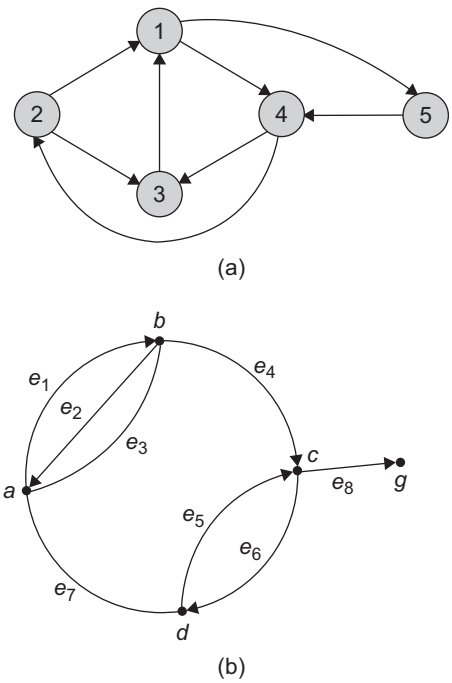


Fig. 7.2 Classification based on edges (a) Simple graph (b) Multigraph

Weighted Graph

A graph where weights are assigned to every edge is called a *weighted graph*. Weights can also be assigned to vertices. A graph of areas and streets of a city may be assigned weights according to its traffic density. A graph of areas and connecting roads may be assigned weights such that the distance between the cities is assigned to edges and area population is assigned to vertices.

Null Graph and Isolated Vertex

In a graph, a node that is not adjacent to any other node is called an *isolated node*. A graph containing only isolated nodes is called a *null graph*. Hence, the set of edges is an empty set in a null graph.

Let V = set of students, E = {there exists an edge incident to two students if they share books}. Let $V = \{a, b, c\}$. If no two students among a, b, and c shares books, then the graph $G = \{V, E\}$ is represented as shown in Fig. 7.3(a).

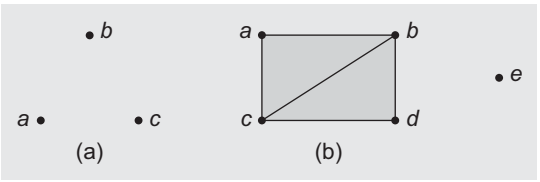


Fig. 7.3 Classification based on nodes (a) Null graph (b) Graph with isolated vertex

Here, G is a null graph, and a, b, and c are isolated vertices. In Fig. 7.3(b), $V = \{a, b, c, d, e\}$ and $E = \{(a, b), (a, c), (b, c), (c, d), (b, d)\}$, and e is an isolated vertex.

Degree of Vertex

In a directed graph, for any node V, the number of edges that have V as its initial node is called the *outdegree* of the node V. In other words, the number of edges incident from a node is its *outdegree* (*outgoing degree*), and the number of edges incident to it is an *indegree* (*incoming degree*). The sum of indegree and outdegree is the total degree of a node (*vertex*). In an undirected graph, the total degree or degree of a node is the number of edges incident to the node. The isolated vertex degree is zero. The degree of vertex a in Fig. 7.4 is 3, whereas the degree of vertex f is 1. For vertex 1 in Fig. 7.2(a), the incoming degree is 2 and the outgoing degree is 2.

Paths and Circuits

Let $G = (V, E)$ be a simple graph. Consider a sequence of edges of G such that the terminal node of any edge in the sequence is the initial node of the next edge, if any, in the sequence (Fig. 7.4).

Here, $G = (V, E)$, $V = \{a, b, c, d\}$, and $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$.

An example of such a sequence is given by $\{e_1, e_2, e_4, e_5\}$.

The sequence $\{e_1, e_2, e_4, e_5\}$ can also be written as $\{a, b, c, d, b\}$.

In addition, $\{e_6, e_2, e_1, e_3, e_4, e_2, e_5\}$ is another sequence. Note that not all edges and nodes appearing in a sequence need to be distinct. In addition, for a given graph, any arbitrary set of nodes such as $\{a, f, b\}$ that is written in any order does not give a sequence as required. In fact, each node appearing in the sequence must be adjacent to the nodes appearing just before and after it in the sequence, except for the first and the last nodes.

Consider the graph in Fig. 7.5.

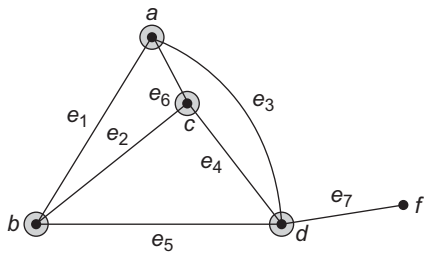


Fig. 7.4 Graph G

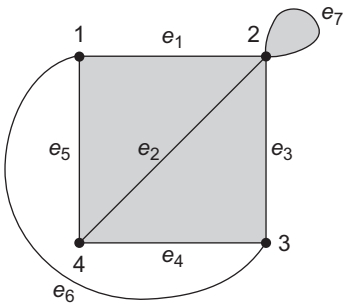


Fig. 7.5 Graph with self loop

A sequence of edges of a graph such that the terminal node of any edge in the sequence is the initial node of the edge, if any, appearing next in the sequence, defines the path of a graph. The number of edges appearing in the path is called the *length of the path*.

Example paths for the graph in Fig. 7.5 are as follows:

$P_1 = \{(2, 4)\}$, also written as $\{e_2\}$
$P_2 = \{(2, 3), (3, 1), (1, 4)\}$, also written as $\{e_3, e_6, e_5\}$
$P_3 = \{e_1, e_2, e_4, e_3, e_1, e_5\}$ or $\{(1, 2), (2, 4), (4, 3), (3, 2), (2, 1), (1, 4)\}$

A path where no edge is traversed more than once is called a *simple path* (or edge simple path). A path where no vertex is traversed (visited) more than once is called an *elementary path* (node simple path). For example, $\{e_1, e_2, e_4, e_6, e_5\}$ is a simple path but not elementary as the vertex 1 is traversed (visited) twice.

A path that originates and ends at the same node is called a *cycle* (circuit). A cycle is elementary if each node is traversed once (except origin) and is simple if every edge of the cycle is traversed once. For example, the following are the cycles for the graph in Fig. 7.5.

$C_1 = \{(2, 2)\}$, also represented as $\{e_7\}$
$C_2 = \{(1, 2), (2, 4), (4, 1)\}$, also represented as $\{1, 2, 4, 1\}$ or $\{e_1, e_2, e_5\}$
$C_3 = \{e_3, e_2, e_5, e_6\}$ or $\{3, 2, 4, 1, 3\}$

Here, the cycle $\{e_1, e_2, e_5\}$ in both simple and elementary cycles is also referred to as a *closed path*.

Connectivity

A graph is said to be connected if and only if there exists a path between every pair of vertices. Some examples are shown in Figs 7.6(a) and (b).

The graph $G = (V, E)$ drawn in Fig. 7.6(a) with $V = \{a, b, c, d, e\}$ is a disconnected graph. It contains two connected components. A connected graph has a single connected component. The graph shown in Fig. 7.6(b) is a connected graph.

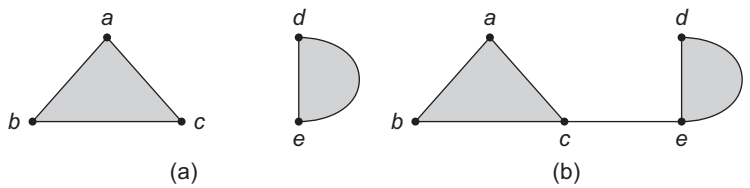


Fig. 7.6 Graph connectivity (a) Graph with two connected components (b) Connected graph with one connected component

Acyclic Graph

A simple graph that does not have any cycles is called *acyclic graph*. Such graphs do not have any loops.

Trees

A class of graphs that is acyclic is termed as *trees*.

Let us now discuss an important class of graphs called *trees* and its associated terminology.

Trees are useful in describing any structure that involves hierarchy. Familiar examples of such structures are family trees, the hierarchy of positions in an organization, and so on.

Forest and Trees

A *forest* is a graph that contains no cycles, and a connected forest is a *tree*. For example, Fig. 7.7 shows a forest with three components, each of which is a tree.

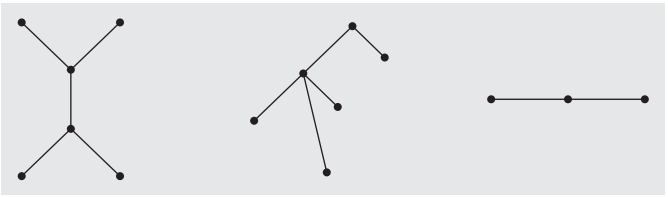


Fig. 7.7 Forest with three trees

Note that trees and forests are simple graphs. The following terminology belongs to trees.

Directed tree An acyclic directed graph is a *directed tree*.

Root A directed tree has one node called its *root*, with indegree zero, whereas for all other nodes, the indegree is 1.

Terminal node (leaf node) In a directed tree, any node that has an outdegree zero is a *terminal node*. The terminal node is also called as *leaf node* (or *external node*).

Branch node (internal node) All other nodes whose outdegree is not zero are called as *branch nodes*.

Level of node The level of any node is its path length from the root. The level of the root of a directed tree is zero, whereas the level of any node is equal to its distance from the root. Distance from the root is the number of edges to be traversed to reach the root.

7.1.2 General Tree

A tree T is defined recursively as follows:

1. A set of zero items is a tree, called the *empty tree* (or null tree).
2. If $T_1, T_2, ..., T_n$ are n trees for $n > 0$ and R is a *node*, then the set T containing R and the trees $T_1, T_2, ..., T_n$ are a tree. Within T, R is called the *root* of T, and $T_1, T_2, ..., T_n$ are called *subtrees*.

The tree in Fig. 7.8(a) is the empty tree since there are no nodes. The tree in Fig. 7.8(b) has only one node, the *root*. The tree in Fig. 7.8(c) has 16 nodes. The root node has four subtrees. The roots of these subtrees are called the *children* of the root. There are 16 nodes in the tree, so there are 15 non-empty subtrees. The nodes with no subtrees are called *terminal nodes* or more commonly, *leaves*. These are 10 leaves in the tree in Fig. 7.8(c).

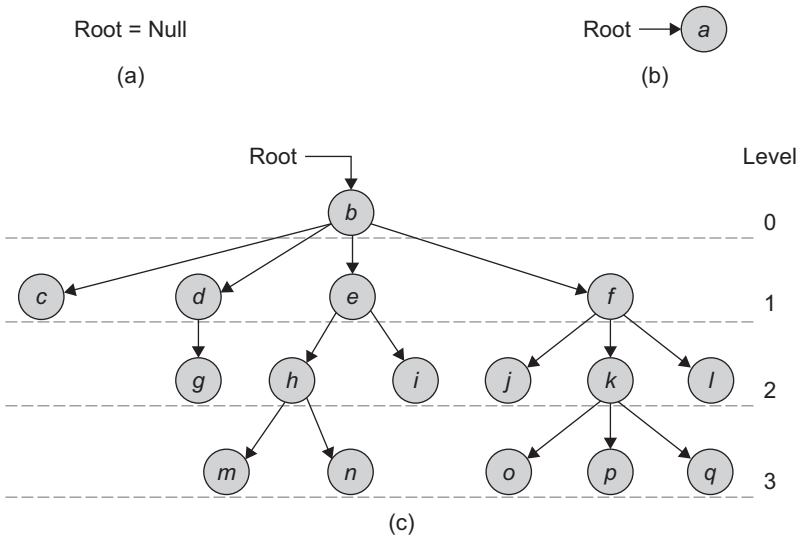


Fig. 7.8 Degree of a tree (a) Empty tree—degree undefined (b) Tree with a single node—degree 0 (c) Tree of height 3—degree 4

The degree of a node is the number of subtrees it has. Thus, the degree of the nodes in Fig. 7.8(c) ranges from zero to four. By definition, the degree of each leaf node is zero. The *degree of a tree* is the maximum degree of a node in the tree. As the tree in Fig. 7.8(a) has no nodes, there is no maximum degree of a node, and the degree of the tree is not defined. The tree in Fig. 7.8(b) has degree zero, and the tree in Fig. 7.8(c) has degree four.

Since family relationships can be modelled as trees, we often call the root of a tree (or subtree) the *parent*, and the roots of the subtrees the *children*. Consequently, the children of the same node are called *siblings*.

The advantage of the relationship between a parent and its children is that a directed edge (or, simply an edge) extends from a parent to its children. Thus, the edges connect a root with the roots of each subtree. For example, in Fig. 7.8(c), an edge extends from the root *b* to each of the nodes *c*, *d*, *e*, and *f*. Similarly, edges extend from *e* to *i* and from *d* to *g*. An undirected edge extends in both directions between a parent and a child. Thus, the undirected edges would also extend from *i* to *e* and from *g* to *d*.

A directed path (or simply path) is a sequence of directed edges $e_1, e_2, ..., e_n$, where the node at the end of one edge serves as the beginning of the next edge. An undirected path is a similar sequence of undirected edges.

For example, in Fig. 7.8(c), one path containing three edges begins at the root and extends through nodes *f*, *k*, and *p*. Similarly, the path beginning at node *h* and containing the nodes *e*, *b,* and *d* is an undirected path. In this chapter and chapters 8 and 9, *edge* refers to a directed edge from a parent to its child. Following the analogy of family hierarchies, if a path exists from one node to another, it is common to state that the first node is an ancestor of the second, and the second is a descendent of the first.

The *length of a path* is the number of edges it contains (which is one less than the number of nodes on the path). The *depth* or *level of a node* is the length of a node, which is the length of a directed path from the root to that node. The *height of a tree* is the length of the path from the root to a node at the lowest level. In other words, the height of a tree is the maximum path length in the tree. Thus, the level of the root of a tree is zero, and the level of each child of the root is one. Equivalently, the height of a tree is the largest level number of any node in the tree.

There are three common ways to symmetrically order (or list) the nodes in a tree: preorder, inorder, and postorder. For each of these orderings, an empty tree gives rise to an empty list, and the tree with one node yields the list with one node. For trees with more than one node, the following statements are true:

1. The preorder list contains the root followed by the preorder list of nodes of the subtrees of the root from left to right.
2. The inorder list contains the inorder list of the leftmost subtree, the root, and the inorder list of each of the other subtrees from left to right.
3. The postorder list contains the postorder list of subtrees of the root from left to right followed by the root.

Figure 7.9 shows a tree whose nodes are labelled with numbers rather than letters.

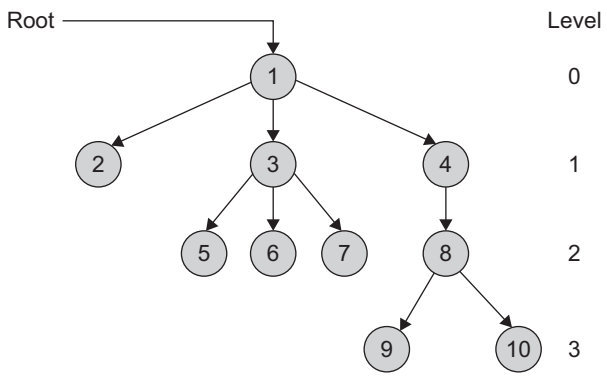


Fig. 7.9 Sample tree

The following is a list of terms for review using the example tree in Fig. 7.9.

Subtrees The nodes labelled 2, 3, and 4 are the roots of the subtrees (children) of the node labelled 1. The nodes labelled 5, 6, and 7 are the roots of the subtrees (children) of the node labelled 3. The node labelled 8 is the root of the subtree (child) of the node labelled 4. The nodes labelled 9 and 10 are the roots of the subtrees (children) of the node labelled 8.

Leaves The nodes labelled 2, 5, 6, 7, 9, and 10 are the terminal nodes or leaf nodes.

Degree The nodes labelled 1 and 3 have degree 3. The node labelled 8 has degree 2. The node labelled 4 has degree 1. All the leaf nodes have degree 0. The degree of the tree is 3, because the maximum degree of any node is 3.

Levels The level number appears on the right of the tree. The level of the root is 0, the level of the nodes labelled 2, 3, and 4 is 1, the level of the nodes labelled 5, 6, 7, and 8 is 2, and that of the nodes labelled 9 and 10 is 3.

Family relationships The node labelled 1 is the parent of the nodes labelled 2, 3, and 4. The node labelled 3 is the parent of the nodes labelled 5, 6, and 7. The node labelled 4 is the parent of the node labelled 8, which in turn is the parent of the nodes labelled 9 and 10. The nodes labelled 2, 3, and 4 are siblings similar to the nodes labelled 9 and 10. Note that the node labelled 8 is not the sibling of the nodes labelled 5, 6, and 7.

Paths and path lengths Paths exist from all parents to children. A unique path exists from the root to each leaf node as shown in Fig. 7.9. Since any sub-path is a path, all the paths are represented. This is shown in the next page:

$1 \rightarrow 2$	Length: 1
$1 \rightarrow 3 \rightarrow 5$	Length: 2
$1 \rightarrow 3 \rightarrow 6$	Length: 2
$1 \rightarrow 3 \rightarrow 7$	Length: 2
$1 \rightarrow 4 \rightarrow 8 \rightarrow 9$	Length: 3
$1 \rightarrow 4 \rightarrow 8 \rightarrow 10$	Length: 3

Height and depth The height of the tree is 3, the maximum level. The depth of the nodes labelled 2, 3, and 4 is 1. The depth of the nodes labelled 5, 6, 7, and 8 is 2. The depth of the nodes labelled 9 and 10 is 3, which is the same as the height of the tree. The depth of the nodes on the lowest level is always the same as the height of the tree.

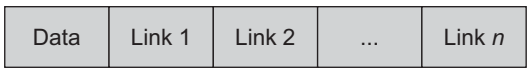
Orderings The preorder, inorder, and postorder orderings of the nodes are given in the following sequence:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 4 \rightarrow 8 \rightarrow 9 \rightarrow 10$ (preorder)
$2 \rightarrow 1 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 8 \rightarrow 10 \rightarrow 4$ (inorder)
$2 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 3 \rightarrow 9 \rightarrow 10 \rightarrow 8 \rightarrow 4 \rightarrow 1$ (postorder)

We shall learn about these orderings in Section 7.7.

7.1.3 Representation of a General Tree

We can use either a sequential organization or a linked organization for representing a tree. If we wish to use a generalized linked list, then a node must have a varying number of fields depending upon the number of branches. However, it is simpler to use algorithms for the data where the node size is fixed.



For a fixed size node, we can use a node with data and pointer fields as in a generalized linked list.

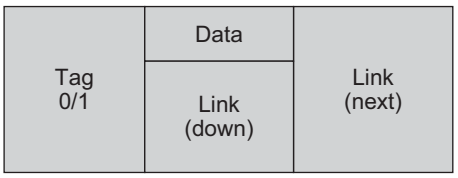


Figure 7.10 shows a sample tree.

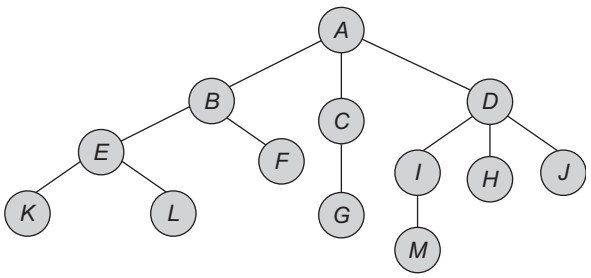


Fig. 7.10 Sample tree

The list representation of this tree is shown in Fig. 7.11.

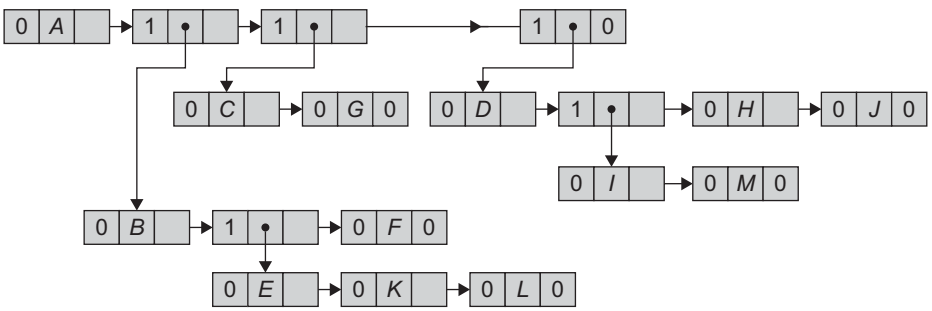


Fig. 7.11 List representation

7.2 TYPES OF TREES

In this section, we shall study some important types of trees.

1. Free tree
2. Rooted tree
3. Ordered tree
4. Regular tree
5. Binary tree
6. Complete tree
7. Position tree

Free tree A free tree is a connected, acyclic graph. It is an undirected graph. It has no node designated as a root. As it is connected, any node can be reached from any other node through a unique path. The tree in Fig. 7.12 is an example of a free tree.

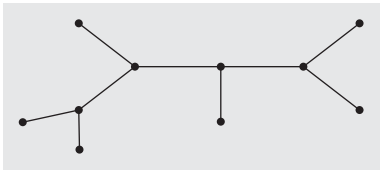


Fig. 7.12 Free tree

Rooted tree Unlike free tree, a *rooted tree* is a directed graph where one node is designated as root, whose incoming degree is zero, whereas for all other nodes, the incoming degree is one (Fig. 7.13).

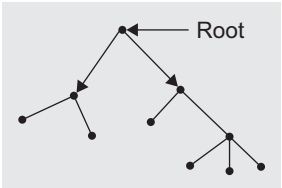


Fig. 7.13 Rooted tree

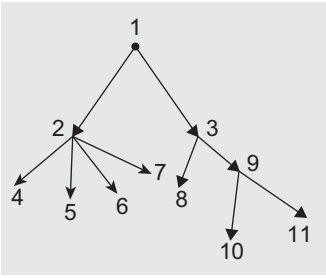


Fig. 7.14 Ordered tree

Ordered tree In many applications, the relative order of the nodes at any particular level assumes some significance. It is easy to impose an order on the nodes at a level by referring to a particular node as the first node, to another node as the second, and so on. Such ordering can be done from left to right (Fig. 7.14). Just like nodes at each level, we can prescribe order to edges. If in a directed tree, an ordering of a node at each level is prescribed, then such a tree is called an *ordered tree*.

Regular tree A tree where each branch node vertex has the same outdegree is called a *regular tree*. If in a directed tree, the outdegree of every node is less than or equal to *m*, then the tree is called an *m-ary tree*. If the outdegree of every node is exactly equal to *m* (the branch nodes) or zero (the leaf nodes), then the tree is called a *regular m-ary tree*.

Binary tree A binary tree is a special form of an *m*-ary tree. Since a binary tree is important, it is frequently used in various applications of computer science.

We have defined an m-ary tree (general tree). A binary tree is an m-ary position tree when $m = 2$. In a binary tree, no node has more than two children.

Complete tree A tree with n nodes and of depth k is complete if and only if its nodes correspond to the nodes that are numbered from 1 to n in the full tree of depth k.

A binary tree of height h is complete if and only if one of the following holds good:

1. It is empty.
2. Its left subtree is complete of height $h-1$ and its right subtree is completely full of height $h-2$.
3. Its left subtree is completely full of height $h-1$ and its right subtree is complete of height $h-1$.

A binary tree is completely full if it is of height h and has $(2^{h+1} - 1)$ nodes.

Full binary tree A binary tree is a *full binary tree* if it contains the maximum possible number of nodes in all levels. Figure 7.15 shows a full binary tree of height 2.

In a full binary tree, each node has two children or no child at all. The total number of nodes in a full binary tree of height h is $2^{h+1}-1$ considering the root at level 0.

It can be calculated by adding the number of nodes of each level as in the following equation:

$$2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$$

Figure 7.15 has $2^{2+1}-1 = 8-1 = 7$ nodes.

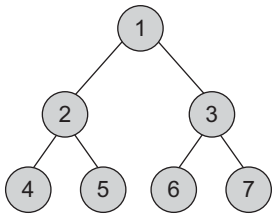


Fig. 7.15 Full binary tree

Complete binary tree A binary tree is said to be a *complete binary tree* if all its levels except the last level have the maximum number of possible nodes, and all the nodes of the last level appear as far left as possible. In a complete binary tree, all the leaf nodes are at the last and the second last level, and the levels are filled from left to right.

Figure 7.16 is a complete binary tree.

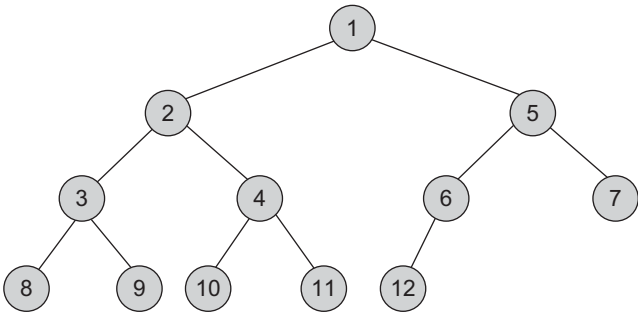


Fig. 7.16 Complete binary tree

Applications such as the priority queue and heap sort use the complete binary tree.

Left skewed binary tree If the right subtree is missing in every node of a tree, we call it a *left skewed tree* (Fig. 7.17).

If the left subtree is missing in every node of a tree, we call it as *right subtree* (Fig. 7.18).

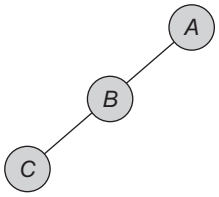


Fig. 7.17 Left skewed tree

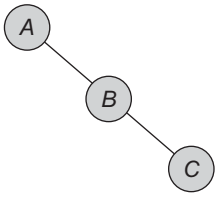


Fig. 7.18 Right skewed tree

Strictly binary tree If every non-terminal node in a binary tree consists of non-empty left and right subtrees, then such a tree is called a *strictly binary tree*.

In Fig. 7.19, the non-empty nodes *D* and *E* have left and right subtrees. Such expression trees are known as *strictly binary trees*.

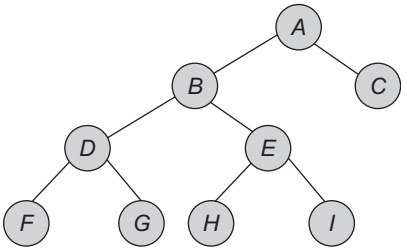


Fig. 7.19 Strictly binary tree

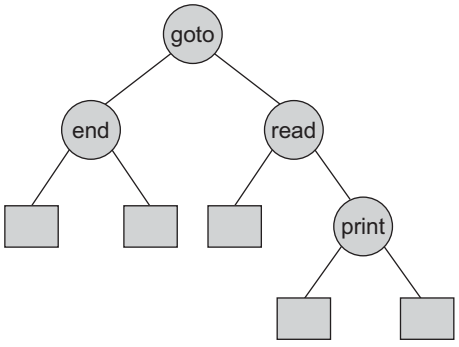


Fig. 7.20 Extended binary tree

Extended binary tree A binary tree *T* with each node having zero or two children is called an *extended binary tree*. The nodes with two children are called *internal nodes*, and those with zero children are called *external nodes*. Trees can be converted into extended trees by adding a node (Fig. 7.20).

Position tree A position tree, also known as a suffix tree, is one that represents the suffixes of a string *S* and such representation facilitates string operations being performed faster. Such a tree's edges are labelled with strings, such that each suffix of *S* corresponds to exactly one path from the tree's root to a leaf node. The space and time requirement is linear in the length of *S*. After its construction, several operations can be performed quickly, such as locating a substring in *S*, locating a substring if a certain number of mistakes are allowed, locating matches for a regular expression pattern, and so on.

7.3 BINARY TREE

One of the most commonly used classes of trees is a binary tree. A binary tree has the degree two, with each node having at most two children. This makes the implementation of trees easier. In addition, binary trees have a wide range of applications. We shall study these in this section.

Definition A binary tree

1. is either an empty tree or
2. consists of a node, called *root*, and two children, *left* and *right*, each of which is itself a binary tree.

The definition is recursive as we have defined a binary tree in terms of itself. All the internal nodes of a binary tree are themselves the roots of smaller binary trees (Fig. 7.21).

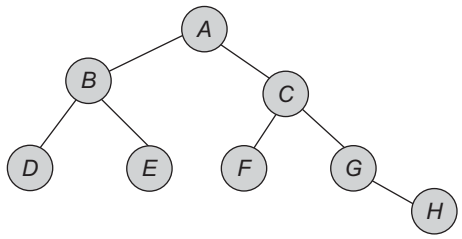


Fig. 7.21 Binary tree

Let us consider the two distinct binary trees in Fig. 7.22.

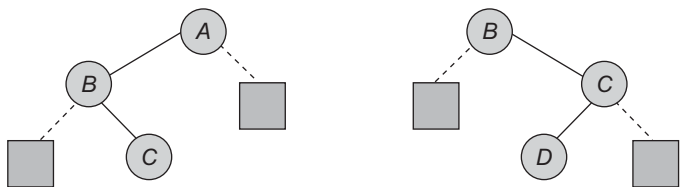


Fig. 7.22 Two binary trees

The definition implies that every non-empty node has two children, either of which may be empty. Here *A*'s right child and *B*'s left child are empty trees (represented by shaded boxes). Usually, empty trees in a binary tree are not shown.

7.3.1 Properties of a Binary Tree

A *tree* is a connected acyclic graph. In many ways, a tree is the simplest non-trivial type of graph. It has several good properties such as the fact that there exists a unique path between every two vertices. The following theorems list some simple properties of trees:

Let *T* be a tree. Then the following properties hold true:

1. There exists a unique path between every two vertices.
2. The number of vertices is one more than the number of edges in the tree.
3. A tree with two or more vertices has at least two leaves.

Let us refer to Fig. 7.23 for proving these properties.

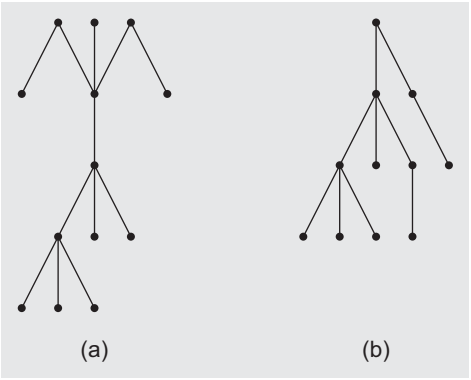


Fig. 7.23 Binary trees (a) Sample 1 (b) Sample 2

Property 1

Property 1 comes from the definition of a tree. As a tree is a connected graph, there exists at least one path between every two vertices. However, if there are two or more paths between a pair of vertices, there would be a circuit in the graph and so the graph cannot be a tree.

Property 2

Property 2 can be proved using mathematical induction. Let there be a tree T with the total number of edges e and the total number of vertices v.

Induction step A tree with one vertex contains no edge, and a tree with two vertices has one edge.

Induction hypothesis Let us consider that there is an edge $\{a, b\}$ in T such that the removal of the edge $\{a, b\}$ divides T into two disjoint trees T_1 and T_2, where T_1 contains the vertex a and all the vertices whose paths to a in T do not contain the edge $\{a, b\}$, and T_2 contains the vertex b and all the vertices whose paths to b do not contain the edge $\{a, b\}$.

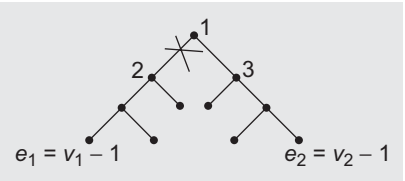
Since both T_1 and T_2 have utmost $v - 1$ vertices, it follows from the hypothesis that

$$\text{for } T_1 \Rightarrow e_1 = v_1 - 1 \text{ and}$$
$$\text{for } T_2 \Rightarrow e_2 = v_2 - 1,$$

where e_1 and e_2 are the number of edges and v_1 and v_2 are the number of vertices in T_1 and T_2, respectively.

Thus $e_1 + e_2 = v_1 + v_2 - 2$

Since $e = e_1 + e_2 + 1$ and $v = v_1 + v_2$, we have $e = v - 1$, as shown in the following figure:



Property 3

Property 3 follows from Property 2, that is, the sum of degrees of the vertices in any graph is equal to $2e$, which is equal to $2v - 2$, in a tree. Since a tree with more than one vertex cannot have any isolated vertex, there must be at least two vertices of indegree 1 in the tree.

Other Properties

1. The maximum number of nodes of level i in a binary tree is 2^{i-1}, where $i \geq 1$.
2. The maximum number of nodes of depth d in a binary tree is 2^{d-1} , where $d \geq 1$.

Let us prove these properties using induction. Assume the root is only one node at level 1.

Hence, the maximum number of nodes is 2^{i-1}, that is, $2^{1-1} = 2^0 = 1$.

By induction hypothesis, let i be any arbitrary positive integer greater than 1. Then, the maximum number of nodes on level $i - 1$ is $2^{i-1-1} = 2^{i-2}$.

Hence, it is proved that the maximum number of nodes at level i is 2^{i-1}.

Note: If we assume the root at level 0, then the expression is 2^i.

Since each node in a binary tree has a maximum degree 2, the maximum number of nodes at level i is 2^{i-1}.

The maximum number of nodes of depth d of a binary tree is given by

$$\sum_{i=1}^{d}(\text{maximum no. of nodes at level } i) = \sum_{i=1}^{d} 2^{d-1}$$

Relation Between Number of Leaf Nodes and Degree-2 Nodes

In any non-empty tree T, if there are n_0 leaf nodes and n_2 nodes of degree 2, then

$$n_0 = n_2 + 1$$

Let n_1 be the number of nodes of degree 1 and n be the total number of nodes.
Since all nodes in T are with the utmost degree 2, then

$$n = n_0 + n_1 + n_2 + \ldots \tag{7.1}$$

If the number of branches is B, then $n = B + 1$. For a binary tree, all the branches stem out from a node of degree 1 or 2. Thus,

$$\begin{aligned} B &= n_1 + 2 \times n_2 \\ \text{So, } n &= B + 1 \\ B &= n_1 + 2 \times n_2 + 1 \end{aligned} \tag{7.2}$$

Subtracting Eq. (7.2) from Eq. (7.1) we get

$$n_0 = n_2 + 1$$

Binary Tree With n Nodes Having n + 1 External Nodes

Taking the base case of a tree with only one node, that is the root, it has two external nodes or null links. So, if $n = 1$, then the number of external nodes is $n + 1$, that is, 2.

From this base case, if there are n internal nodes where the left subtree has L nodes, then the right subtree has $n - L - 1$ internal nodes (1 for the root).

By induction hypothesis, the number of external nodes of the left subtree is $L + 1$.

The number of external nodes of the right subtree is $(n - L - 1) + 1 = n - L$. So, the total number of external nodes is $L + 1 + n - L = n + 1$.

7.4 BINARY TREE ABSTRACT DATA TYPE

We have defined a binary tree. Let us now define it as an abstract data type (ADT), which includes a list of operations that process it.

```
ADT btree
 1. Declare create()→btree
 2. makebtree(btree, element, btree)→btree
 3. isEmpty(btree)→boolean
 4. leftchild(btree)→btree
 5. rightchild(btree)→btree
 6. data(btree)→element
 7. for all l,r ∈ btree, e ∈ element, Let
 8. isEmpty(create) = true
 9. isEmpty(makebtree(l,e,r)) = false
10. leftchild(create()) = error
11. rightchild(create()) = error
12. leftchild(makebtree(l,e,r)) = l
13. rightchild(makebtree(l,e,r)) = r
14. data(makebtree(l,e,r)) = e
15. end
end btree
```

The six functions with their domains and ranges are declared in lines 1 through 6. Lines 7 through 14 are the set of axioms that describe how the functions are related.

The `create()` operation creates an empty binary tree; the `isEmpty()` operation checks whether `btree` is empty or not and returns the Boolean value true or false, respectively; `leftchild(btree)` and `rightchild(btree)` return the left and right subtrees, respectively; `data(btree)` returns the data element.

Program Code 7.1 states the class definition of the operations that process the tree ADT.

PROGRAM CODE 7.1

```
class TreeNode
{
   public:
      char Data;
      TreeNode *Lchild;
      TreeNode *Rchild;
};

class BinaryTree
{
   private:
      TreeNode *Root;
   public:
      BinaryTree(){Root = Null};
      // constructor creates an empty tree
      TreeNode * GetNode();
      void InsertNode(TreeNode*);
      void DeleteNode( TreeNode*);
;
```

Operations on binary tree The basic operations on a binary tree can be as listed as follows:

1. Creation—Creating an empty binary tree to which the 'root' points
2. Traversal—Visiting all the nodes in a binary tree
3. Deletion—Deleting a node from a non-empty binary tree
4. Insertion—Inserting a node into an existing (may be empty) binary tree
5. Merge—Merging two binary trees
6. Copy—Copying a binary tree
7. Compare—Comparing two binary trees
8. Finding a replica or mirror of a binary tree

7.5 REALIZATION OF A BINARY TREE

In this section, we shall study the basic realization of a binary tree and discuss its capabilities for supporting various operations. The implementation of a binary tree should represent the hierarchical relationship between a parent node and its left and right children. We have studied the elementary data structures such as linked list and arrays. Now, we shall extend these concepts to the binary tree structures. We shall give more emphasis to the linked implementation as it is more popular than the corresponding sequential structure due to the following two main reasons:

1. A binary tree has a natural implementation in a linked storage.
2. The linked structure is more convenient for insertions and deletions.

Let us study both the implementations.

7.5.1 Array Implementation of Binary Trees

One of the ways to represent a tree using an array is to store the nodes level-by-level, starting from the level 0 where the root is present. Such a representation requires sequential numbering of the nodes, starting with the nodes on level 0, then those on level 1, and so on.

We have defined a complete tree. A complete binary tree of height h has $(2^{h+1} - 1)$ nodes in it. The nodes can be stored in a one-dimensional array, *tree*, with the node numbered at the location *tree*(i). An array of size $2^{h+1} - 1$ is needed for the same.

The root node is stored in the first memory location as the first element in the array. The following rules can be used to decide the location of any i^{th} node of a tree:
For any node with index i, $0 \le i \le n - 1$,

1. Parent(i) $= \lfloor (t - 1)/2 \rfloor$ if $i \neq 0$; if $i = 0$, then it is the root that has no parent.
2. Lchild(i) $= 2 \times i + 1$ if $2i + 1 \le n - 1$; if $2i \ge n$, then i has no left child.
3. Rchild(i) $= 2i + 2$ if $2i + 2 \le n - 1$; if $(2i + 1) \ge n$, then i has no right child.

Let us consider the complete binary tree in Fig. 7.24.

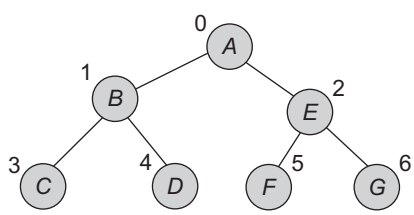


Fig. 7.24 Complete binary tree

The representation of the binary tree in Fig. 7.24 using an array is as follows:

0	1	2	3	4	5	6	7	8
A	B	E	C	D	F	G	–	–

Let us consider one more example as in Fig. 7.25.

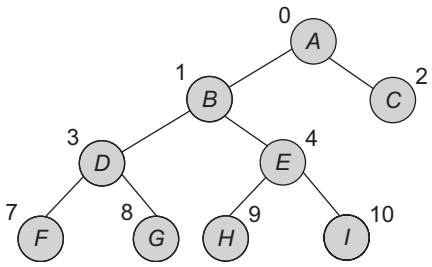
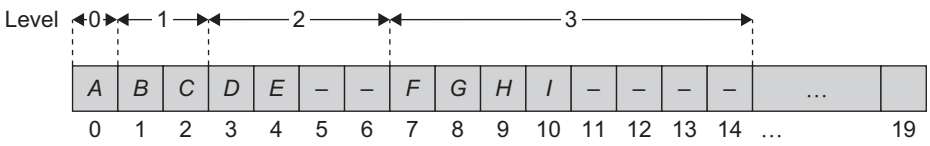


Fig. 7.25 Tree with 11 nodes

Now, the array representation of the tree in Fig. 7.25 is as follows:



Let us consider one more example of a skewed tree as in Fig. 7.26.

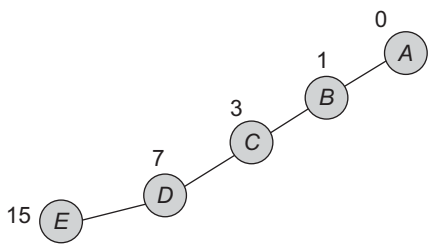
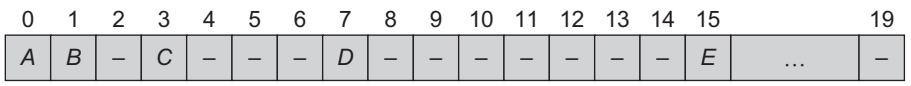


Fig. 7.26 Sample skewed tree

This tree has the following array representation:



This representation of binary trees using an array seems to be the easiest. Certainly, it can be used for all binary trees. However, such a representation has certain drawbacks. In most of the representations, there will be a lot of unused space. For complete binary trees, the representation seems to be good as no space in an array is wasted between the nodes. Certainly, the space is wasted as we generally declare an array of some arbitrary maximum limit. From the examples, we can make out that for the skewed tree, however, less than half of the array is only used and more is left unused. In the worst case, a skewed tree of depth k will require $2^{k+1} - 1$ locations of array, and occupy just a few of them.

In addition, even though the representation seems to be good for complete binary trees, it is not useful for many other binary trees. In addition, the representation has drawbacks of sequential representation, which have been discussed. A major drawback of sequential representation is that the data movement of potentially many nodes is needed when the insertion or deletion of a node occurs. Here, the movement of nodes is needed to reflect the change in the level number of these nodes.

These problems can be overcome by the use of linked representation.

Advantages The various merits of representing binary trees using arrays are as follows:

1. Any node can be accessed from any other node by calculating the index.
2. Here, the data is stored without any pointers to its successor or predecessor.
3. In the programming languages, where dynamic memory allocation is not possible (such as BASIC, FORTRAN), array representation is the only means to store a tree.

Disadvantages The various demerits when representing binary trees using arrays are as follows:

1. Other than full binary trees, majority of the array entries may be empty.
2. It allows only static representation. The array size cannot be changed during the execution.
3. Inserting a new node to it or deleting a node from it is inefficient with this representation, because it requires considerable data movement up and down the array, which demand excessive amount of processing time.

7.5.2 Linked Implementation of Binary Trees

Binary tree has a natural implementation in a linked storage. In a linked organization, we wish that all the nodes should be allocated dynamically. Hence, we need each node with data and link fields. Each node of a binary tree has both a left and a right subtree. Each node will have three fields—Lchild, Data, and Rchild. Pictorially, this node is shown in Fig. 7.27.

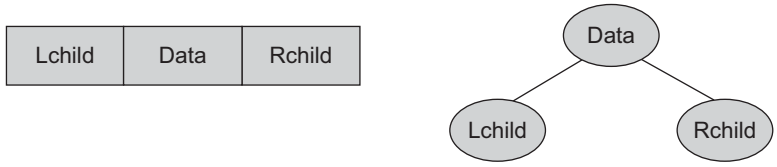


Fig. 7.27 Tree node

A node does not provide information about the parent node. However, it is still adequate for most of the applications. If needed, the fourth parent field can be included. The binary tree in Fig. 7.28 will have the linked representation as in Fig. 7.29. The root of the tree is stored in the data member *root* of the tree. This data member provides an access pointer to the tree.

Here, 0 (zero) stored at Lchild or Rchild fields represents that the respective child is not present. Let us consider one more example as in Fig. 7.29.

In this node structure, Lchild and Rchild are the two link fields to store the addresses of left child and right child of a node; data is the information content of the node. With this representation, if we know the address of the root node, then using it, any other node can be accessed.

Each node of a binary tree (as the root of some subtree) has both left and right subtrees, which can be accessed through pointers as follows.

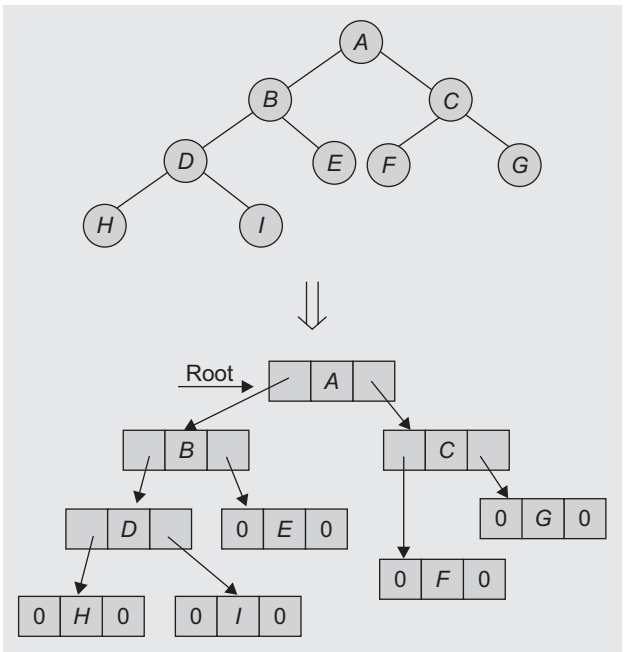


Fig. 7.28 Sample tree 1 and its linked representation

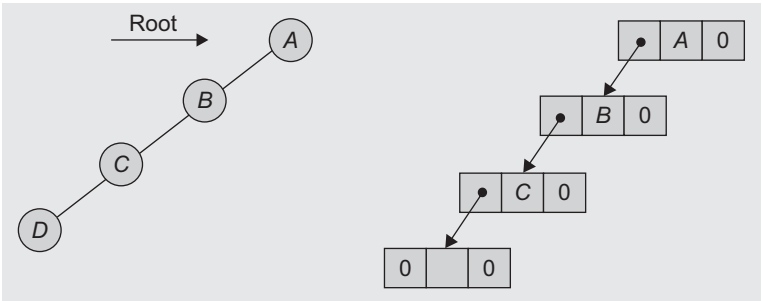


Fig. 7.29 Sample tree 2 and its linked representation

Program Code 7.1 described the ADT for a binary tree. Program Code 7.2 is a class for the binary tree that shows specifications for the ADT. Implementation of these functions is covered in the forthcoming topics.

PROGRAM CODE 7.2

```
class TreeNode
{
   public:
      char Data;
         TreeNode *Lchild;
      TreeNode *Rchild;
};

class BinaryTree
{
   private:
      TreeNode *Root;
   public:
      BinaryTree() {Root = Null;}      // constructor
      // int BTree_Equal(BinaryTree, BinaryTree);
      TreeNode *GetNode();
      void InsertNode(TreeNode*);
      void DeleteNode(TreeNode*);
      void Postorder(TreeNode*);
      void Inorder(TreeNode*);
      void Preorder(TreeNode*);
      TreeNode *TreeCopy();
      void Mirror();
      int TreeHeight(TreeNode*);
      int CountLeaf(TreeNode*);
      int CountNode(TreeNode*);
      void BFS_Tree();
      void DFS_Tree();
      TreeNode *Create_Btree_InandPre_Traversal(char
      preorder[max], char inorder[max]);
      void Postorder_Non_Recursive(void);
      void Inorder_Non_Recursive();
      void Preorder_Non_Recursive();
      int BTree_Equal(BinaryTree, BinaryTree);
      TreeNode *TreeCopy(TreeNode*);
      void Mirror(TreeNode*);
};
```

Using this declaration for a linked representation, the binary tree representation can be logically viewed as in Fig. 7.30. The physical representation shows the memory allocation of nodes.

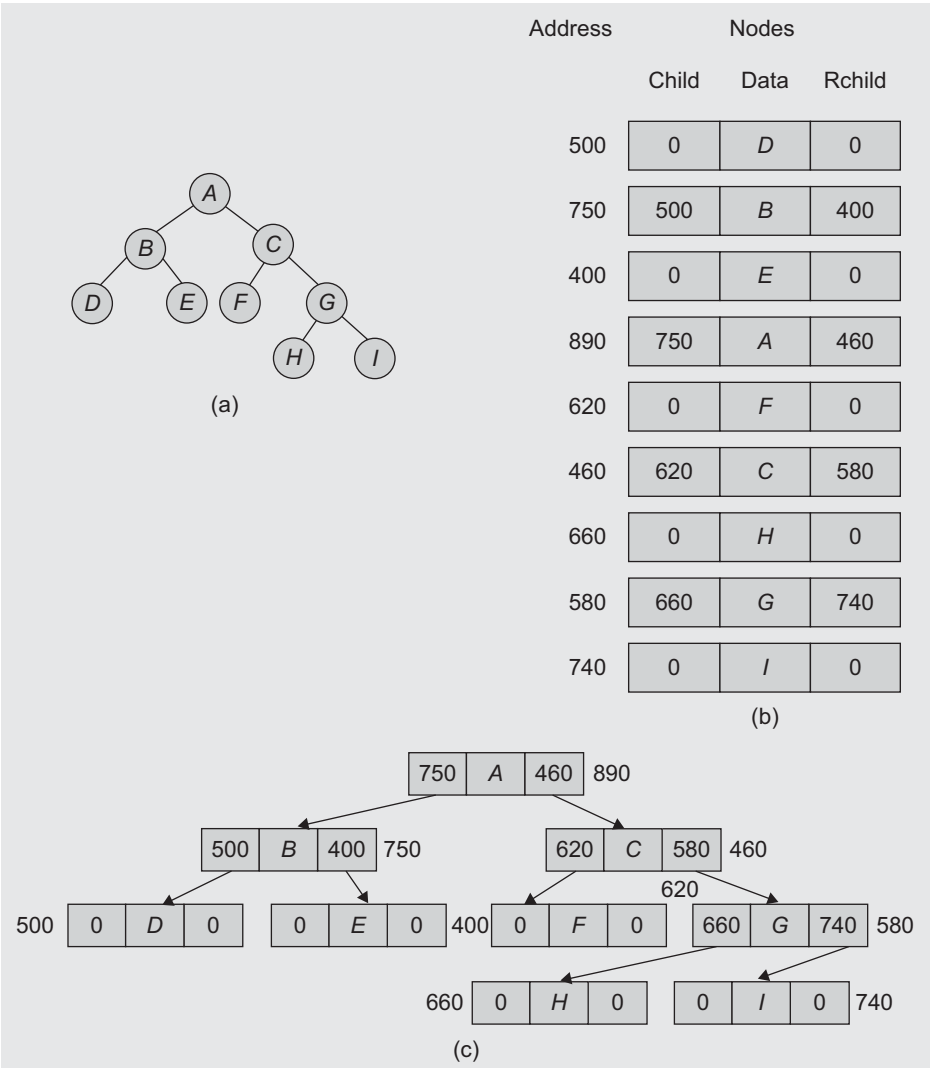


Fig. 7.30 Tree and its views (a) Binary tree (b) Physical view (c) Logical view

Advantages The merits of representing binary trees through liked representations are as follows:

1. The drawbacks of the sequential representation are overcome in this representation. We may or may not know the tree depth in advance. In addition, for unbalanced trees, the memory is not wasted.

2. Insertion and deletion operations are more efficient in this representation.
3. It is useful for dynamic data.

Disadvantages The demerits of representing binary trees through linked representation are as follows:

1. In this representation, there is no direct access to any node. It has to be traversed from the root to reach to a particular node.
2. As compared to sequential representation, the memory needed per node is more. This is due to two link fields (left child and right child for binary trees) in the node.
3. The programming languages not supporting dynamic memory management would not be useful for this representation.

7.6 INSERTION OF A NODE IN BINARY TREE

The `insert()` operation inserts a new node at any position in a binary tree. The node to be inserted could be a branch node or a leaf node. The branch node insertion is generally based on some criteria that are usually in the context of a special form of a binary tree. Let us study a commonly used case of inserting a node as a leaf node.

The insertion procedure is a two-step process.

1. Search for the node whose child node is to be inserted. This is a node at some level *i*, and a node is to be inserted at the level $i + 1$ as either its left child or right child. This is the node after which the insertion is to be made.
2. Link a new node to the node that becomes its parent node, that is, either the Lchild or the Rchild.

This is represented in Fig. 7.31.

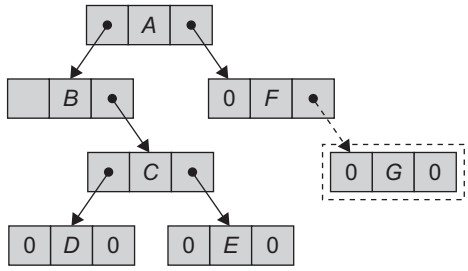


Fig. 7.31 Insertion of node *G* as the Rchild of node *F*

7.7 BINARY TREE TRAVERSAL

Traversal is a frequently used operation. Traversal of a tree means *stepping through the nodes of a tree* by means of the connections between parents and children, which is also called *walking the tree*, and the action is *a walk of the tree*. Traversal means visiting every

node of a binary tree. There are many operations that are often performed on a tree such as search a node, print some information, insert a node, delete a node, and so on. All such operations need the traversal through a tree.

This operation is used to visit each node (exactly once). A full traversal of a tree visits nodes of a tree in a certain linear order. This linear order could be familiar and useful. For example, if the binary tree contains an arithmetic expression, then its traversal may give us the expression in infix, postfix, or prefix notations.

There are various traversal methods. For a systematic traversal, it is better to visit each node (starting from the root) and its two subtrees in the same way. In other words, when traversing, we need to treat each node and its subtree in the same fashion. If we let L, D, and R stand for moving left, data, and moving right, respectively (Fig. 7.32), when at a node, then there are six possible combinations—LDR, LRD, DLR, DRL, RDL, and RLD.

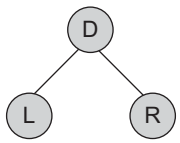


Fig. 7.32 Components of a subtree

Consider the binary tree shown in Fig. 7.33. This tree represents a binary tree. We have studied all the notations of an expression tree and its inter-conversions in Chapter 3.

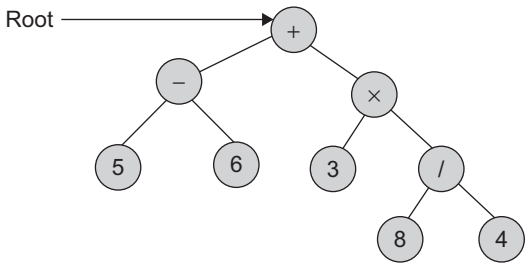


Fig. 7.33 A binary tree representing an arithmetic expression

Let us see the result of each of the six traversals.

LDR:	5 – 6 + 3 × 8 / 4
LRD:	5 6 – 3 8 4 / × +
DLR:	+ – 5 6 × 3 / 8 4
DRL:	+ × / 4 8 3 – 6 5
RDL:	4 / 8 × 3 + 6 – 5
RLD:	4 8 / 3 × 6 5 – +

We can notice that DLR and RLD, LDR and RDL, and LRD and DRL are mirror symmetric. If we adopt the convention that traversing is done left before right, only then, the three traversals, that is, LDR, LRD, and DLR, are fundamental. These are called as

inorder, *postorder*, and *preorder* traversals because there is a natural correspondence between these traversals producing the infix, postfix, and prefix forms of an arithmetic expression, respectively.

7.7.1 Preorder Traversal

In this traversal, the root is visited first followed by the left subtree in preorder and then the right subtree in preorder. The tree characteristics lead to naturally implement the tree traversals recursively. It can be defined in the following steps:

Preorder (DLR) Algorithm

1. Visit the root node, say *D*.
2. Traverse the left subtree of the node in preorder.
3. Traverse the right subtree of the node in preorder.

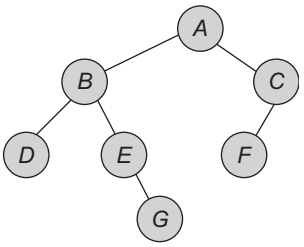


Fig. 7.34 Binary tree

Let us consider the tree in Fig. 7.34.

A preorder traversal of the tree in Fig. 7.34 visits the node in a sequence: *A B D E G C F*. For an expression tree, the preorder traversal yields a prefix expression (Fig. 7.35).

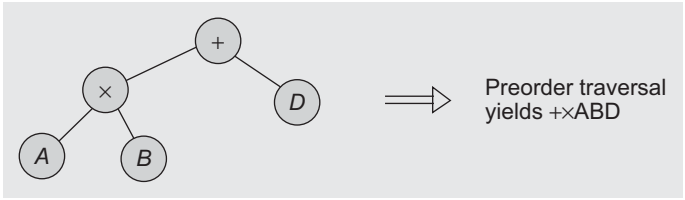
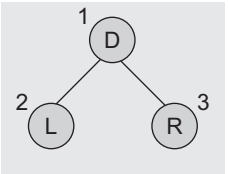


Fig. 7.35 Expression tree and its preorder traversal

The preorder traversal says, 'visit a node, traverse left, and continue moving. When you cannot continue, move right and begin again or move back, until you can move right and stop'. The `Preorder()` function can be written as both recursive and non recursive.

```
void BinaryTree :: Preorder(TreeNode*)
{
   if(Root != Null)
   {
      cout << Root->Data;
      Preorder(Root->Lchild);
      Preorder(Root->Rchild);
   }
}
```



Let us consider the tree in Fig. 7.36. This tree contains an arithmetic expression with the binary operators add (+), multiply (×), divide (/), exponentiation (^), and variables *A*, *B*, *C*, *D*, and *E*.

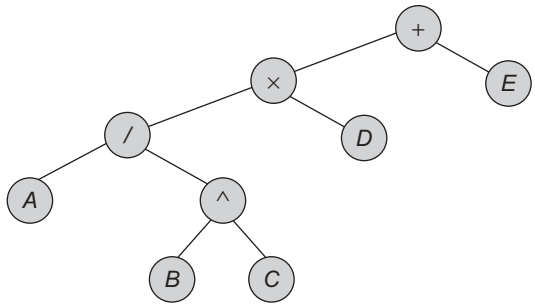


Fig. 7.36 Binary tree for expression

It gives us the prefix expression +×/*A* ^ *BCDE*. The preorder traversal is also called as *depth-first traversal*.

7.7.2 Inorder Traversal

In this traversal, the left subtree is visited first in inorder followed by the root and then the right subtree in inorder. This can be defined as the following:

Inorder (LDR) Algorithm

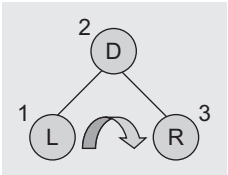
1. Traverse the left subtree of the root node in inorder.
2. Visit the root node *node*.
3. Traverse the right subtree of the root node in inorder.

Let us consider the binary tree in Fig. 7.34. An *inorder traversal* of a tree visits the node in the following sequence.

Inorder sequence: D B E G A F C An inorder expression traversal of the tree in Fig. 7.35, which is an expression tree, yields an inorder expression as $A \times B + D$ and for Fig. 7.36 yields an inorder expression as $((A/B \text{ ^ } C) \times D) + E$.

The `Inorder()` function simply calls for moving down the tree towards the left until it can no longer proceed. So next, we visit the node, move one node to the right, and continue again. If we cannot move, move one node to the right and continue again. If we cannot move to the right, go back one more node and then continue. The inorder traversal is also called as *symmetric traversal*. This traversal can be written as a recursive function as follows:

```
void BinaryTree :: Inorder(TreeNode*)
{
   if(Root != Null)
   {
      Inorder(Root→Lchild);
      cout << Root→Data;
      Inorder(Root→Rchild);
   }
}
```



7.7.3 Postorder Traversal

In this traversal, the left subtree is visited first in postorder followed by the right subtree in postorder and then the root. This is defined as the following:

Postorder (LRD) Algorithm

1. Traverse the root's left child (subtree) of the root node in postorder.
2. Traverse the root's right child (subtree) of the root node in postorder.
3. Visit the root node.

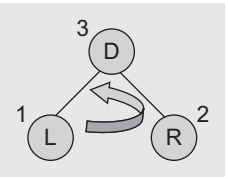
Let us consider the binary tree in Fig. 7.34. The postorder traversal yields the following sequence:

Postorder sequence: D G E B F C A For the expression tree in Fig. 7.35, the postorder traversal yields a postfix expression as the following:

$$= A\ B \times D +$$

For the tree in Fig. 7.36, the postfix expression by the postorder traversal is *ABC××/ D×E+*. The postorder traversal says, "traverse left and continue again. When you cannot continue, move right and begin again or move back until you can move right and visit the node."

```
void BinaryTree :: Postorder(TreeNode*)
{
   if(Root != Null)
   {
      Postorder(Root→Lchild);
      Postorder(Root→Rchild);
      cout << Root→Data;
   }
}
```



7.7.4 Non-recursive Implementation of Traversals

We have defined the recursive traversals. These are easy to read and understand. However, a language translator such as a compiler will be burdened to carry out the execution. Let us go for the other non-recursive approach for these algorithms. Let us write a non-recursive implementation using stacks. Here, at the time of left or right traversal, when the left or right node is the root of the other subtree, the current node is to be stacked for further traversal.

Non-recursive Preorder Algorithm

```
1. Tmp = Root
2. while(stack not empty) do
   begin
      if(Tmp is not null) then
      begin
         visit(Tmp→Data)
         Push(Tmp→Lchild)
```

```
         Tmp = Tmp->Lchild
      else if(stack is empty)
         then exit
      else
         Tmp = Pop()
         Tmp = Tmp->Rchild
      end
   end
3. stop
```

The C++ implementation of the non-recursive preorder algorithm is given in Program Code 7.3.

PROGRAM CODE 7.3

```
void BinaryTree :: Preorder_Non_Recursive()
{
   TreeNode *Tmp = Root;
   stack S;
   while(1)
   {
      while(Tmp != Null)
      // traverse left till left is null and push
      {
         S.Push(Tmp);
         cout << Tmp ->Data;
         Tmp = Tmp->Lchild;
      }
      if(S.IsEmpty()) return;
      //if stack is not empty then pop one and go to
      right
      Tmp = S.Pop();
      Tmp = Tmp->Rchild;
      // if stack is empty stop the process
   }
}
```

We need `stack` and `queue` for tree operations such as depth-first and breadth-first traversals. Program Code 7.4 uses these two data structures:

PROGRAM CODE 7.4

```
class stack
{
   public:
      TreeNode *stk[max];
```

```
      int data,top;
      stack *S;
   public:
      stack()
      {
         top = -1;
      }
      int IsEmpty()
      {
         if(top == -1)
            return 1;
         else
            return 0;
      }
      void Push(TreeNode *x)
      {
         stk[top++] = x;
      }
      TreeNode *Pop()
      {
         TreeNode *x;
         x = stk[top--];
         return(x);
      }
};

class stack1
{
   public:
      char stk1[max];
      int data,top;
   public:
      stack1()
      {
         top = -1;
      }
      int IsEmpty1()
      {
         if(top == -1)
            return 1;
        else
            return 0;
      }
```

```
      void Push1(char x)
      {
         stk1[top++] = x;
      }
      char Pop1()
      {
         char x;
         x = stk1[top--];
         return(x);
      }
};

class queue
{
   TreeNode *que[max];
   int data, rear, front;
   public:
      queue()
      {
         rear = front = -1;
      }
      int Empty()
      {
         if(rear == front)
            return 1;
         else
            return 0;
      }
      int Full()
      {
         if(rear == max)
            return 1;
         else
            return 0;
      }
     void Add(TreeNode *x)
      {
         if(Full())
            cout << "\n Queue Overflow";
        else
          que[++rear] = x;
      }
      TreeNode *Del()
```

```
      {
         TreeNode *x;
         if(Empty())
         {
            cout << "\n Queue is empty";
            //return -1;
         }
         else
         {
            x = que[front++];
            return(x);
         }
      }
};
```

Non-recursive Inorder Algorithm

Algorithm 7.1 is for non-recursive inorder traversal of a binary tree.

ALGORITHM 7.1

```
1. Tmp = Root
2. while(1) do
   begin
      while(Tmp != Null) then
      begin
         push(Tmp)
         Tmp = Tmp→Lchild
      end
      if(stack is empty) then exit
      Tmp = Pop()
      visit(Tmp→data)
      Tmp = Tmp→Rchild
   end while
3. Stop
```

The C++ implementation of Algorithm 7.1 is stated in Program Code 7.5.

PROGRAM CODE 7.5

```
void BinaryTree :: Inorder_Non_Recursive()
{
   TreeNode *Tmp;
   stack S;
   Tmp = Root;
  while(1)
```

```
    {
       while(Tmp != Null)
       {
          S.Push(Tmp);
          Tmp = Tmp->Lchild;
       }
       if(S.IsEmpty())
          return;
          //if stack is not empty then pop one and go to
right
       Tmp = S.Pop();
       cout << Tmp->Data;
       Tmp = Tmp->Rchild;
    }
}
```

Non-recursive Postorder Algorithm

Algorithm 7.2 is for non-recursive postorder traversal of a binary tree. As compared to earlier algorithms, in postorder traversal, we require the `Pop` operation when returning from the left and right subtrees.

ALGORITHM 7.2

1. When we return from the left subtree, perform the following operations:
   (a) `Tmp = Pop`
   (b) `Tmp = Tmp→Rchild`.
2. When we return from the right subtree, perform the following operations:
   (a) `Tmp = Pop`
   (b) `Print Tmp→data` (that is, visit and process the node, if required)
   (c) `Tmp = Pop`

Hence, we need to differentiate between the `return` operation from the left subtree and right subtree. Let us use the stack that stores the status: 'L' for left, and 'R' for right. For performing the extra `Pop` operation while returning from the right subtree, we need to assign `Tmp = Null`. Program Code 7.6 demonstrates this.

PROGRAM CODE 7.6

```
void BinaryTree :: Postorder_Non_Recursive(void)
{
   TreeNode * Tmp = Root;
   stack S;
```

```
    stack1 S1;
    char flag;
    // stack S stores the node and S1 stores the flag 'L' or 'R'
    while(1)
    {
       while(Tmp != Null)
       // traverse tree left till not Null
       {
          S.Push(Tmp);
          S1.Push1('L');
          // push node in S and 'L' in S1
          Tmp = Tmp->Lchild;
       }
       if(S.IsEmpty())
          return;
       else
       {
          Tmp = S.Pop();
          //pop node
          flag = S1.Pop1();
          if(flag == 'R')
          // if flag is 'R' display data
          {
             cout << Tmp->Data;
             Tmp = Null;
          }
          else      // if flag is 'L'
          {
             S.Push( Tmp);
             // push Tmp with flag 'R'
             S1.Push1('R');
             Tmp = Tmp->Rchild;
             // move to right
          }
       }
    }
}
```

7.7.5 Formation of Binary Tree from its Traversals

Sometimes, we need to construct a binary tree if its traversals are known. From a single traversal, a unique binary tree cannot be constructed. However, if two traversals are

known, then the corresponding tree can be drawn uniquely. Let us examine these possibilities and then chalk out the algorithm for the same.

The basic principle for formulation is as follows:

1. If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given, then the last node is the root node.
2. Once the root node is identified, all the nodes in all left and right subtrees of the root node can be identified.
3. Same techniques can be applied repeatedly to form the subtrees.

We can conclude that for the binary tree, construction and traversals are essential out of which one should be inorder traversal and another should be preorder or postorder traversal. Alternatively, for the given preorder and postorder traversals, the binary tree cannot be obtained uniquely.

Consider the following sequences of traversal as in Example 7.1.

EXAMPLE 7.1 Construct a binary tree using the following two traversals:

Inorder : *D B H E A I F J C G*
Preorder: *A B D E H C F I J G*

Solution From the preorder traversal, it is evident that *A* is the root node. In addition, in the inorder traversal, all the nodes that are to the left side of *A* belong to the left subtree and those to the right side of *A* belong to the right subtree (Fig. 7.37).

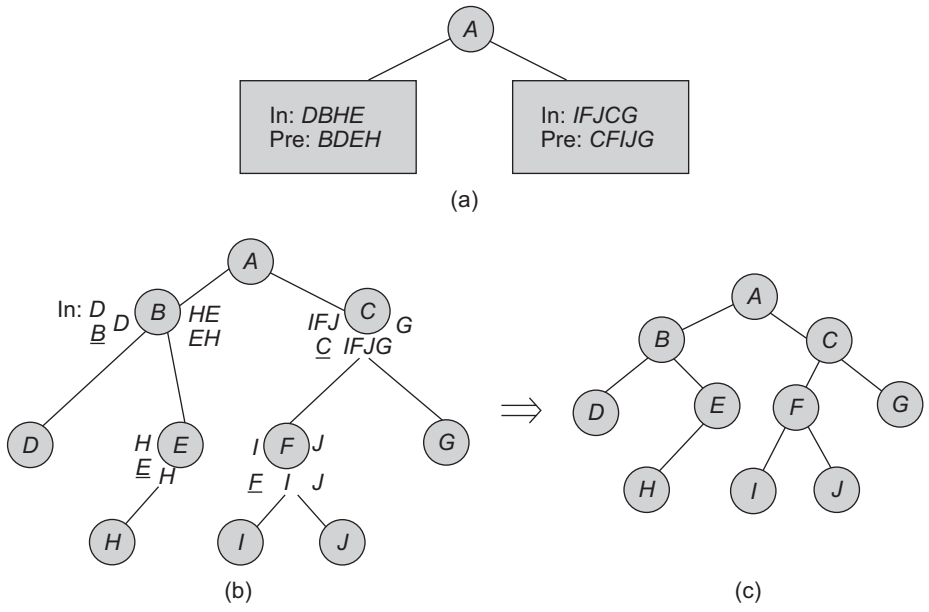


Fig. 7.37 Binary tree from inorder and preorder traversals (a) Two subtrees as a being the root from two traversals (b) Repeated application (c) Final binary tree

Example 7.2 shows the construction of a binary tree from a sequence of inorder and postorder traversals.

EXAMPLE 7.2 Construct a binary tree from its inorder and postorder traversals.

Inorder : 1 2 3 4 5 6 7 8 9
Postorder: 1 3 5 4 2 8 7 9 6

Solution As 6 is the last node traversed in postorder, 6 is the root (Fig. 7.38). The final binary tree constructed is as in Fig. 7.39.

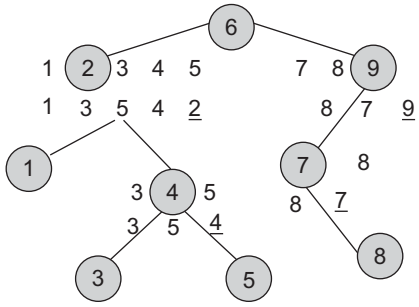


Fig. 7.38 Sample tree

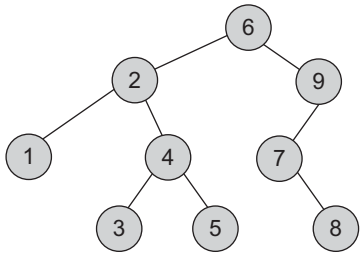


Fig. 7.39 Final binary tree

Using the inorder and preorder traversals, a binary tree can be constructed. Program Code 7.7 is the implementation of the same.

PROGRAM CODE 7.7

```
// code to construct tree using preorder and inorder
// sequences
class TreeNode
{
   public:
      char Data;
      TreeNode *Lchild, *Rchild;
};

//Function to create a tree using preorder and inorder
sequences
TreeNode *BinaryTree :: Create_Btree_InandPre_Traversal
(char preorder[max], char inorder[max])
{
   //to store divided inorder and preorder sequence
   char in1[max],in2[max],pre1[max],pre2[max];
```

```
    TreeNode *Tmp;
    int i,j,k;
    if(strlen(preorder) == 0)
       return Null;
       Tmp = new TreeNode;
       Tmp->Data = preorder[0];
       //following code is for dividing inorder sequence
       for(i = 0;inorder[i] != preorder[0]; i++)
          in1[i] = inorder[i];
       in1[i] = '\0';
       i++;
       k = 0;
       for(j = i; inorder[j] != '\0'; j++)
          in2[k++] = inorder[j];
       in2[k] = '\0';
       cout << " in " << in1 << "   " << in2;
      //following code is for dividing preorder sequence
       i = j = 0;k = 1;
       for(k = 1; preorder[k] != '\0'; k++)
       {
          if(strchr(in1,preorder[k]) != Null)
          // strchr function used to check char is
          present in string or not
             pre1[i++]=preorder[k];
          else
             pre2[j++]=preorder[k];
       }
       pre1[i]='\0';
       pre2[j]='\0';
       Tmp->Lchild = Create_Btree_InandPre_Traversal
       (pre1,in1);
       Tmp->Rchild = Create_Btree_InandPre_Traversal
       (pre2,in2);
       return Tmp;
}
```

7.7.6 Breadth- and Depth-first Traversals

As defined earlier, we know that the traversal of a tree means visiting through the nodes of a tree. A traversal where the node is visited before its children is called a *breadth-first traversal*; a walk where the children are visited prior to the parent is called a *depth-first traversal*.

Depth-first Traversal

A traversal where the children are visited (operated on) before the parent is called the *depth-first traversal*. We have already seen a few ways to traverse the elements of a tree. For example, look at the tree in Fig. 7.40.

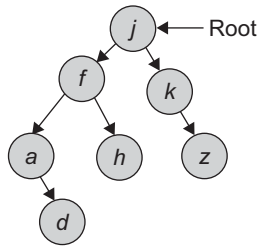


Fig. 7.40 Sample tree

A preorder traversal would visit the elements in the order: *j*, *f*, *a*, *d*, *h*, *k*, *z*.

This type of traversal is called a *depth-first traversal* as it tries to go deeper in the tree before exploring the siblings. For example, the traversal visits all the descendants of *f* (i.e., keeps going deeper) before visiting *f*'s sibling *k* (and any of *k*'s descendants).

The two other traversal orders are *inorder* and *postorder*. An inorder traversal would give us the following sequence: *a*, *d*, *f*, *h*, *j*, *k*, *z*. A postorder traversal would give us the following sequence: *d*, *a*, *h*, *f*, *z*, *k*, *j*. These traversals also try to go *deeper* first.

For example, the inorder traversal visits *a* and *d* before it explores *a*'s sibling *h*. Likewise, it visits all of the *j*'s left subtree (i.e., *a*, *d*, *f*, *h*) before exploring *j*'s right subtree (i.e., *k*, *z*). The same is true for the postorder traversal. It visits all of the *j*'s left subtree (i.e., *d*, *a*, *h*, *f*) before exploring any part of the right subtree (i.e., *z*, *k*).

Let us see how it is implemented non-recursively using stack. Program Code 7.8 is the implementation of non-recursive depth-first traversal using stack.

PROGRAM CODE 7.8

```
void BinaryTree :: DFS_Tree()
{
   stack S;
   TreeNode *Tmp=Root;
   do
   {
      cout << Tmp->Data;
      if(Tmp->Rchild != Null)
         S.Push(Tmp->Rchild);
      if(Tmp->Lchild != Null)
        S.Push(Tmp->Lchild);
      if(S.IsEmpty()) break;
      Tmp = S.Pop();
   }
   while(1);
}
```

Breadth-first Traversal

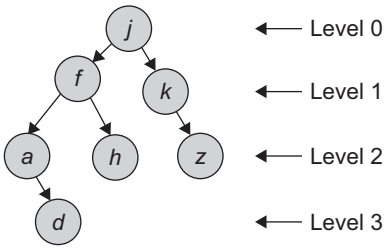


Fig. 7.41 Tree of level 3

Depth-first is not the only way to go through the elements of a tree. Another way is to go through them level-by-level (Fig. 7.41). For example, each element exists at a certain level (or depth) in the tree.

So, if we want to visit the elements level-by-level (and left to right, as usual), we would start at level 0 with *j*, then go to level 1 for *f* and *k*, then go to level 2 for *a*, *h*, and *z*, and finally go to level 3 for *d*. This level-by-level traversal is called a *breadth-first traversal* because we explore the breadth, that is, the full width of the tree at a given level, before going deeper. One may think about why we should ever traverse a tree breadth wise. Well, there are many reasons for the same.

Tree of officers Suppose you have a tree representing some command structure as in Fig. 7.42.

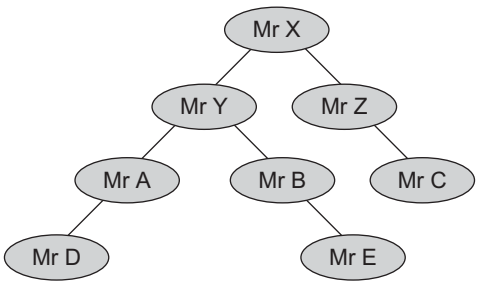


Fig. 7.42 Officers tree

This tree is meant to represent who is in charge of the lower ranking of officers. For example, Mr X is directly responsible for Mr Y and Mr Z. People of the same rank are at the same level in the tree. However, to distinguish between people of the same rank, those with more experience are on the left and those with less experience are on the right (i.e., experience from left to right). One way the command would follow to trace the path is to list the officers in the tree in the breadth-first order. This would give the following result:

Mr X at the top level, say manager; then his subordinates, say department heads as.

Mr Y and Mr Z and their subordinates as Mr A, Mr B, Mr C, Mr D, and Mr E.

In this case, traversing the tree breadth-first makes more sense as we want to print the results post wise from the highest level. As we have seen, the tree traversals go deeper in the tree first using stack as a helper data structure. Instead, if we are going to implement a breadth-first traversal of a tree, we will need the queue as a helper data structure. Let us consider the tree drawn as in Fig. 7.43.

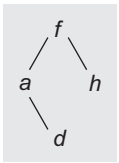


Fig. 7.43 Breadth-first traversal—sample tree

When we are at element *f*, that is the only time we have the access to its two immediate children, *a* and *h*. So, when we are at *f*, we need the data structure that holds its children. Obviously then, *f* must have been in the data structure before them, since we would have put *f* in when we were at *f*'s parent. So, if we put the parent in the data structure before its children, we need to select the data structure that will give us the order we need. A queue will give us the order we want! A queue enforces the first-in-first-out (FIFO) order, and we have to process the parent first before its descendants.

Non-recursive breadth-first traversal is implemented in Program Code 7.9 using a queue.

PROGRAM CODE 7.9

```
void BinaryTree :: BFS_Tree()
{
   queue Q;
   TreeNode *Tmp = Root;
   do
   {
      cout << Tmp->Data;
      if((Tmp->Lchild) != Null)
         Q.Add(Tmp->Lchild);
      if(Tmp->Rchild != Null)
         Q.Add(Tmp->Rchild);
      if(Q.Empty()) break;
      Tmp = Q.Del();
   }
   while(1);
}
```

7.8 OTHER TREE OPERATIONS

Using traversal as a basic operation, many other operations can be performed on a tree, such as finding the height of the tree, computing the total number of nodes, leaf nodes, and so on. Let us study a few of such operations.

7.8.1 Counting Nodes

`CountNode()` is the function that returns the total count of nodes in a linked binary tree.

```
int  BinaryTree :: CountNode(TreeNode *Root)
{
   if(Root == Null)
      return 0;
   else
      return(1 + CountNode(Root->Rchild) + CountNode(Root->Lchild));
}
```

7.8.2 Counting Leaf Nodes

The `CountLeaf()` operation counts the total number of leaf nodes in a linked binary tree. *Leaf nodes* are those with no left or right children.

```
int BinaryTree :: CountLeaf(TreeNode *Root)
{
   if(Root == Null)
      return 0;
```

```
   else if((Root->Rchild == Null) && (Root->Lchild == Null))
      return(1);
   else
      return(CountLeaf(Root->Lchild) + CountLeaf(Root->Rchild));
}
```

7.8.3 Computing Height of Binary Tree

The `TreeHeight()` operation computes the height of a linked binary tree. *Height* of a tree is the maximum path length in the tree. We can get the path length by traversing the tree depthwise. Let us consider that an empty tree's height is 0 and the tree with only one node has the height 1.

```
int BinaryTree :: TreeHeight(TreeNode *Root)
{
   int heightL, heightR;
   if(Root == Null)
      return 0;
   if(Root->Lchild == Null && Root->Rchild == Null)
      return 0;
   heightL = TreeHeight(Root->Lchild);
   heightR = TreeHeight(Root->Rchild);
   if(heightR > heightL)
      return(heightR + 1);
   return(heightL + 1);
}
```

7.8.4 Getting Mirror, Replica, or Tree Interchange of Binary Tree

The `Mirror()` operation finds the mirror of the tree that will interchange all left and right subtrees in a linked binary tree.

```
void BinaryTree :: Mirror(TreeNode *Root)
{
   TreeNode *Tmp;
   if(Root != Null)
   {
      Tmp = Root->Lchild;
      Root->Lchild = Root->Rchild;
      Root->Rchild = Tmp;
      Mirror(Root->Lchild);
      Mirror(Root->Rchild);
   }
}
```

7.8.5 Copying Binary Tree

The `TreeCopy()` operation makes a copy of the linked binary tree. The function should allocate the necessary nodes and copy the respective contents into them.

```
TreeNode *BinaryTree :: TreeCopy()
{
   TreeNode *Tmp;
   if(Root == Null)
```

```
      return Null;
   Tmp = new TreeNode;
   Tmp→Lchild = TreeCopy(Root→Lchild);
   Tmp→Rchild = TreeCopy(Root→Rchild);
   Tmp→Data = Root→Data;
   return Tmp;
}
```

7.8.6 Equality Test

The `BTree_Equal()` operation checks whether two binary trees are equal. Two trees are said to be equal if they have the same topology, and all the corresponding nodes are equal. The same topology refers to the fact that each branch in the first tree corresponds to a branch in the second tree in the same order and vice versa.

```
int BinaryTree :: BTree_Equal(Binarytree T1 , BinaryTree T2)
{
   if(Root == Null && T2.Root == Null)
      return 1;
   return(Root && T2.Root);
   &&(Root->Data == T2.Root->Data);
   &&BTree_Equal(Root->Lchild ,T2.Root->Lchild);
   &&BTree_Equal(Root->Rchild, T2.Root->Rchild));
}
```

7.9 CONVERSION OF GENERAL TREE TO BINARY TREE

A general tree is one where each node can have an outgoing degree n, where $n \geq 0$. Each node may have many applications such as charts, genesis, networks, and so on. In this section, we shall study that every general tree can be represented as a binary tree. We can make out from the study of representations of trees that the representation of a binary tree is easier than the general tree representation.

Binary trees are the trees where the maximum degree of any node is two. Any general tree can be represented as a binary tree using the following algorithm:

1. All nodes of a general tree will be the nodes of a binary tree.
2. The root T of a general tree is the root of a binary tree.
3. To obtain a binary tree, we use a relationship between the nodes that can have the following two characteristics:
   (a) The first or the leftmost child–parent relationship
   (b) Node-next right sibling relationship

Use the following steps to obtain T' from T:

1. Connect (insert an arrow from) each node to its right sibling (if one exists).
2. Disconnect (remove arrows from) each node from (to) all but the leftmost child.

Examples 7.3 and 7.4 demonstrate the conversion of a general tree into a binary tree.

EXAMPLE 7.3 Convert the general tree in Fig. 7.44 into its corresponding binary tree.

Solution In this tree, the leftmost child of 2 is 3 and the next right child of 2 is 4. The binary tree corresponding to the tree is obtained by connecting together all siblings of each node (Fig. 7.45) and deleting all links from a node to its children except for the link to its leftmost child. The binary tree obtained is shown in Fig. 7.46.

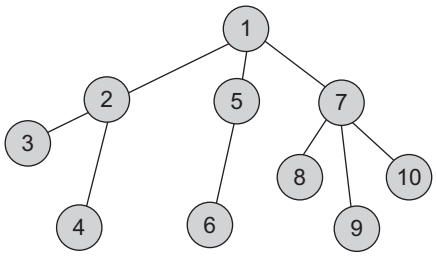


Fig. 7.44 General tree

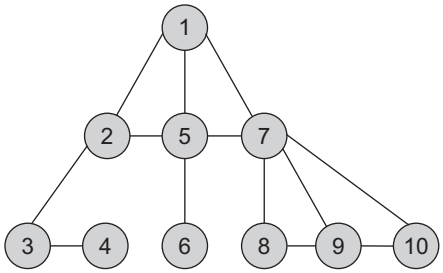


Fig. 7.45 Step 1

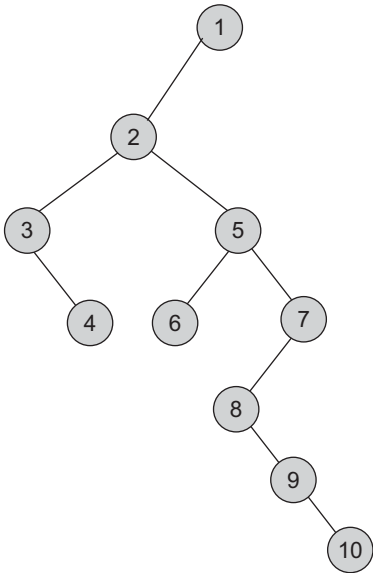


Fig. 7.46 Binary tree for tree in Fig. 7.44

EXAMPLE 7.4 Convert the general tree in Fig. 7.47 into its corresponding binary tree.

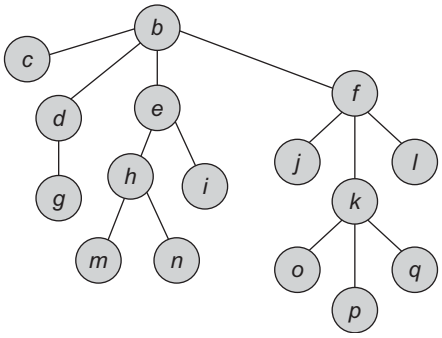


Fig. 7.47 General tree

Solution The binary tree representation of Fig. 7.47 is shown in Fig. 7.48.

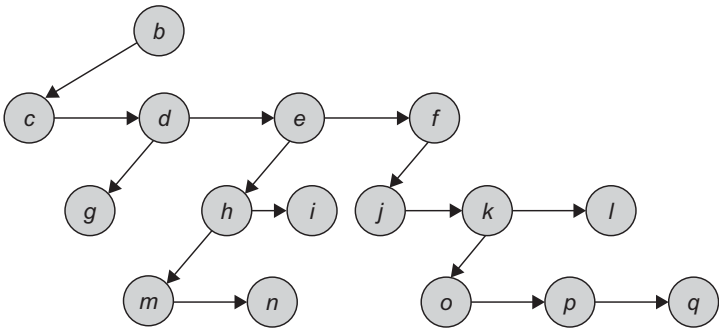


Fig. 7.48 Binary tree for tree in Fig. 7.47

This binary tree can also be drawn in a more familiar format as in Fig. 7.49.

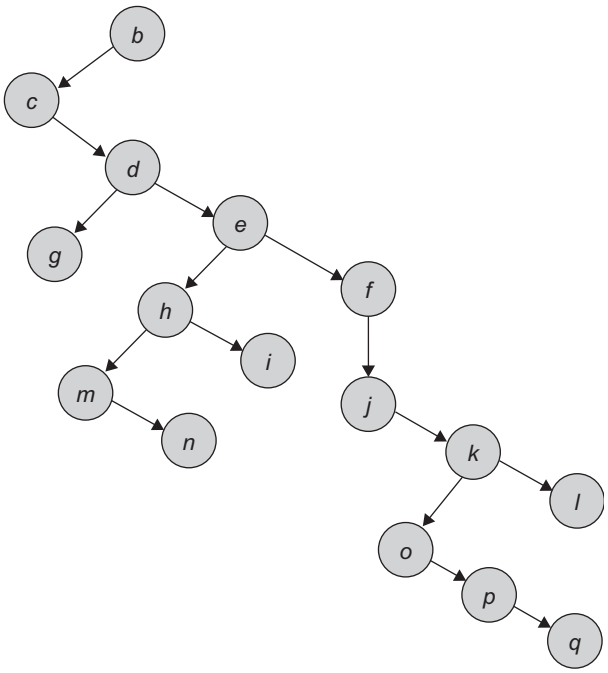


Fig. 7.49 Resultant binary tree in format 2

Note that if the order of the children in a tree is not important (unordered tree), then any of the children of a node could be its leftmost child and any of its siblings could be its next right siblings. For the sake of definiteness, we choose the nodes based upon how the tree is drawn. The node structure for a binary tree can be shown as in Fig. 7.50.

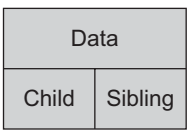


Fig. 7.50 Node structure for binary tree

In addition, notice that the transformation from the resultant binary tree to the original n-ary tree is reversible. That is, given a binary tree representation of a general tree, we can re-create the general tree. A left node is the leftmost child of its parent. A right node is a sibling of its parent

Example 7.5 illustrates the conversion of a given tree to a binary tree.

EXAMPLE 7.5 Convert the following tree in Fig. 7.51 into a binary tree.

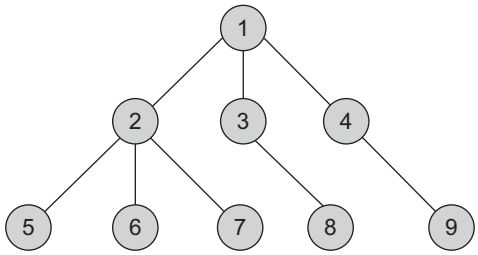


Fig. 7.51 Given general tree

Solution Let us connect the siblings and drop all the pointers from the parent to the children except to the first child as in Fig. 7.52.

Now every child becomes a left child, every sibling becomes a right child, and the resultant tree is a binary tree as in Fig. 7.53.

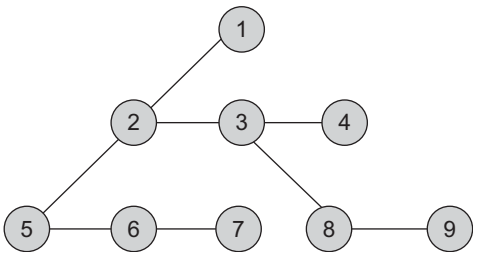


Fig. 7.52 Step 1

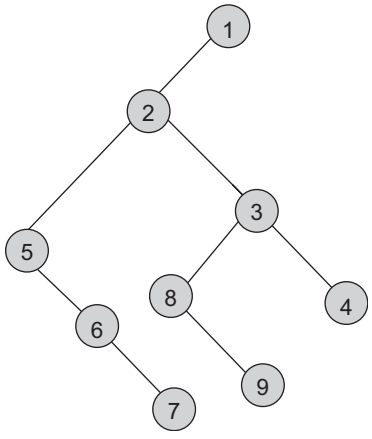


Fig. 7.53 Resultant binary tree

7.10 BINARY SEARCH TREE

We know that the sequential search with O(n) searches is slower compared to the binary search with O($\log_2 n$) searches. If the list is an ordered list stored in a contiguous sequential storage, the binary search is faster. Though the list is stored, if it is stored in a linked list, the binary search cannot work as it does not support direct access. However, when we frequently need to make changes in the list, that is, inserting a new entry or

deleting an old entry, then it is much slower to use a contiguous sequential list than a linked list, as `insert()` and `delete()` operations need data movement. On the other hand, in a linked organization, we need only a few pointer manipulations for insertion and deletions. If so, we can then find an implementation for an ordered list where we can search quickly (as with binary search) and insert or delete elements quickly (as with linked list). A binary tree provides an excellent solution to this problem.

1. We can make entries of an ordered list into the nodes of a binary tree. We shall see that we can search a target key in $O(\log_2 n)$ steps, and in addition, we can insert and delete the key in time $O(\log_2 n)$.
2. The binary search tree (BST) is a binary tree with the property that the value in a node is greater than any value in a node's left subtree and less than any value in the node's right subtree.
3. This property guarantees fast search time provided the tree is relatively balanced.

The BSTs are classified as *static trees* and *dynamic trees*. *Static tree* is a BST where the set of values in the nodes is known in advance and never changes. *Dynamic tree* is a BST where the values in a tree may change over time. We shall study the ways of building and balancing these search trees to guarantee that the trees remain balanced so that the search time is minimum.

Binary search tree A BST is a binary tree that is either empty or where every node contains a key and satisfies the following conditions:

1. The key in the left child of a node, if it exists, is less than the key in its parent node.
2. The key in the right child of a node, if it exists, is greater than the key in its parent node.
3. The left and right subtrees of a node are again BSTs.

The definition ensures that no two entries in a BST can have equal keys. It is possible to change the definition to allow entries with equal keys but doing so makes an algorithm more complicated. We assume that all keys are unique.

Figure 7.54 represents two BSTs.

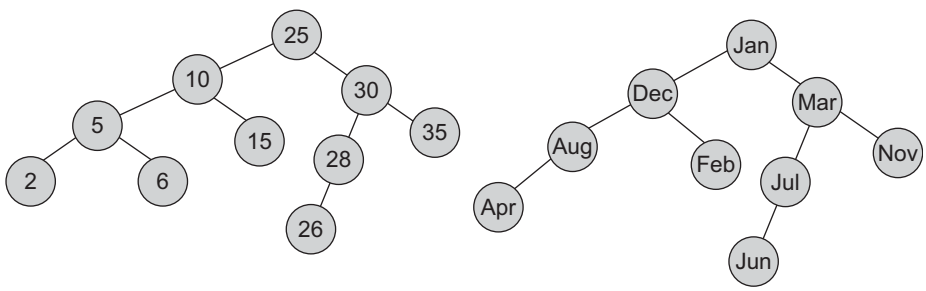


Fig. 7.54 Binary search trees

The following are the operations commonly performed on a BST:

1. Searching a key
2. Inserting a key
3. Deleting a key
4. Traversing the tree

Program Code 7.10 demonstrates the class for a BST showing the node structure and the function prototypes to operate on.

PROGRAM CODE 7.10

```
class TreeNode
{
   <data type> Key;
   TreeNode *Lchild, *Rchild;
};

class BSTree
{
   private:
      TreeNode *Root;
   public:
      BSTree() {Root = Null;}      // constructor
      void InsertNode(int Key);
      void DeleteNode(int key);
      void Search(int Key);
      bool IsEmpty();
};
```

7.10.1 Inserting a Node

To insert a new node into a BST, the keys should remain in proper order so that the resulting tree satisfies the definition of a BST.

Let us consider the insertion of the keys Esha, Beena, Deepa, Gilda, Amit, Geeta, and Chetan, into an initially empty tree in the given order as shown in Fig. 7.55.

If the tree is empty, then the first entry, Esha, when inserted, becomes the root, as shown in Fig. 7.55(a). Since Beena is less than Esha, insertion goes into the left subtree of Esha, and so on for all keys. If the tree is not empty, then we must compare the key with the one in the root. `Insert()` function can be written both recursively as well as non-recursively.

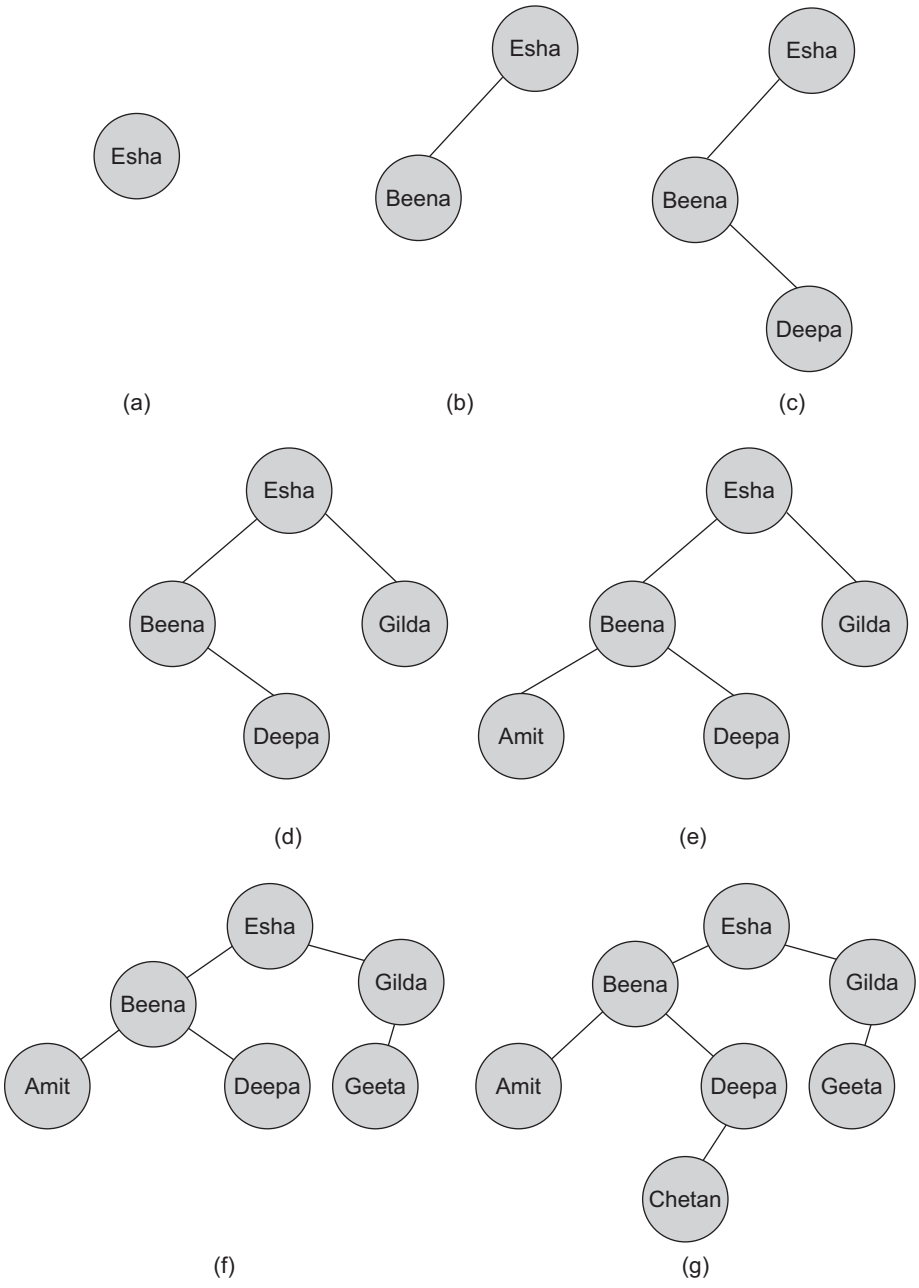


Fig. 7.55 Insertion in BST (a) Insert Esha (b) Insert Beena (c) Insert Deepa (d) Insert Gilda (e) Insert Amit (f) Insert Geeta (g) Insert Chetan

The BST ADT was given in Program Code 7.10. The code for the function to insert a node is listed in Program Code 7.11.

PROGRAM CODE 7.11

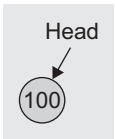
```
TreeNode *BSTree :: Insert(int Key)
{
   TreeNode *Tmp, NewNode;
   NewNode = new BSTNode;
   NewNode->Data = Key;
   NewNode->Lchild = NewNode->Rchild = Null:
   if(Root == Null)
   {
      Root = NewNode;
      return;
   }
   Tmp = Root;
   while(Tmp != Null)
   {
      if(Tmp->Data < Key)
      {
         if(Tmp->Lchild == Null)
         {
            Tmp->Lchild = NewNode;
            return;
         }
         Tmp = Tmp->Lchild;
         else if(Tmp->Rchild == Null)
         {
            Tmp->Rchild = NewNode;
            return;
         }
      }
   }
   Tmp = Tmp->Rchild;
}
```

Initially, the tree is empty. The tree is built through the `Insert()` function. Example 7.6 shows the construction of a BST from a given set of elements.

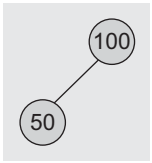
EXAMPLE 7.6 Build a BST from the following set of elements—100, 50, 200, 300, 20, 150, 70, 180, 120, 30—and traverse the tree built in inorder, postorder, and preorder.

Solution The BST is constructed through the following steps:

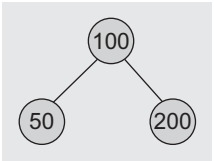
Step 1: Initially, `Root = Null`. Now let us insert 100.



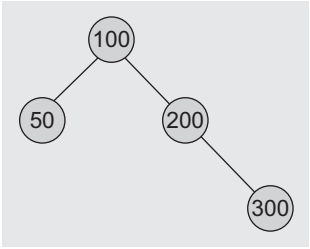
Step 2: Insert 50. As it is less than the root, that is, 100, and its left child is `Null`, we insert it as a left child of the root.



Step 3: Insert 200. As it is greater than the root, that is, 100, and its right child is `Null`, we insert it as a right child of the root.

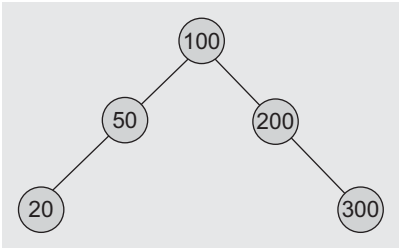


Step 4: Insert 300. As it is greater than the root, that is, 100, we move right to 200. It is greater than 200, and its right child is `Null`, so we insert it as a right child of 200.

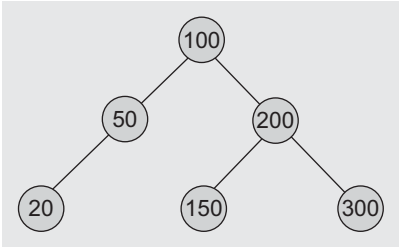


Similarly, we insert the other nodes.

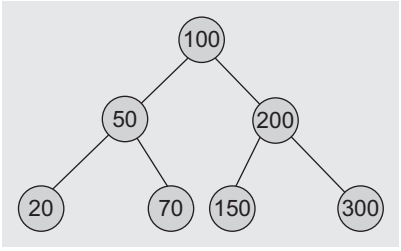
Step 5: Insert 20.



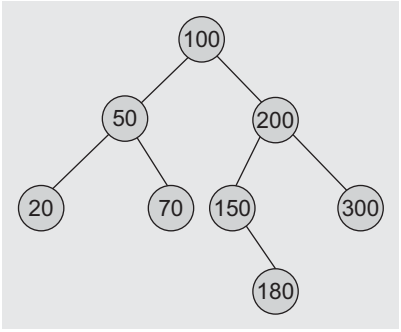
Step 6: Insert 150.



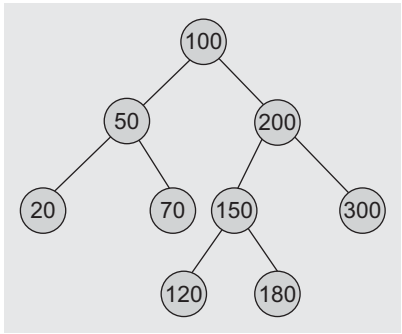
Step 7: Insert 70.



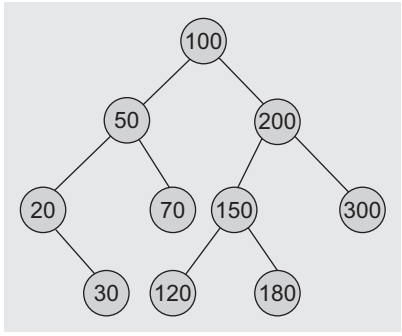
Step 8: Insert 180.



Step 9: Insert 120.



Step 10: Insert 30.



Traverse the built tree in inorder, postorder, and preorder and display the sequence of numbers.

Preorder: 100 50 20 30 70 200 150 120 180 300
Inorder: 20 30 50 70 100 120 150 180 200 300
Postorder: 30 20 70 50 120 180 150 300 200 100

Note that the inorder traversal of a BST generates the data in ascending order.

7.10.2 Searching for a Key

To search for a target key, we first compare it with the key at the root of the tree. If it is the same, then the algorithm ends. If it is less than the key at the root, search for the target key in the left subtree, else search in the right subtree. Let us, for example, search for the key 'Saurabh' in Fig. 7.56.

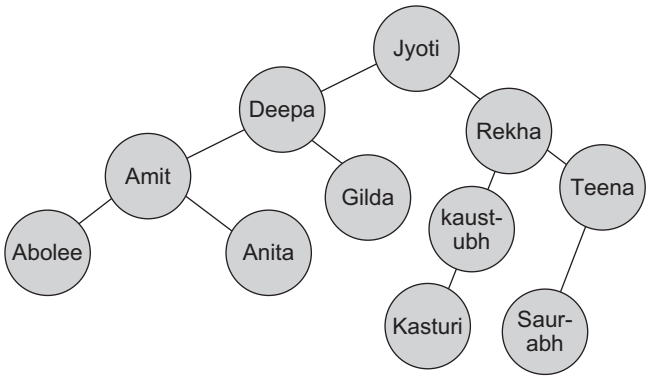


Fig. 7.56 Binary search tree

We first compare 'Saurabh' with the key of the root, 'Jyoti'. Since 'Saurabh' comes after 'Jyoti' in alphabetical order, we move to the right side and next compare it with the key 'Rekha'. Since 'Saurabh' comes after 'Rekha', we move to the right again and compare with 'Teena'. Since 'Saurabh' comes before 'Teena', we move to the left.

Now the question is to identify what event will be the terminating condition for the search. The solution is if we find the key, the function finishes successfully. If not, we continue searching until we hit an empty subtree.

Program Code 7.12 shows the implementation of `search()` function, both non-recursive and recursive implementations.

PROGRAM CODE 7.12

```
TreeNode *BSTree :: Search(int Key)
{
   TreeNode *Tmp = Root;
   while(Tmp)
   {
      if(Tmp->Data == Key)
         return Tmp;
      else if(Tmp->data < Key)
         Tmp = Tmp->Lchild;
      else
         Tmp = Tmp->Rchild;
   }
   return Null;
}
```

```
class BSTree
{
   private:
      TreeNode * Root;
      TreeNode*BSTree :: Rec_Search(TreeNode *root,
      int key);
   public:
      BSTree() {Root = Null;}      // constructor
      void InsertNode(int Key);
      void DeleteNode(int key);
      void Search(int Key);
      bool IsEmpty();
      TreeNode* BSTree:: Recursive_Search(int key)
      {
         Rec_Search(Root, int Key);
      }

};
TreeNode *BSTree :: Rec_Search(TreeNode *root, int key)
{
   if(root == Null)
      return(root);
   else
   {
      if(root->Data < Key)
         root = Rec_Search(root->Lchild);
      else if(root->data > Key)
         root = Rec_Search(root->Rchild);
   }
}
```

The class with recursive function is given in this program code.

7.10.3 Deleting a Node

Deletion of a node is one of the frequently performed operations. Let *T* be a BST and *X* be the node of key *K* to be deleted from *T*, if it exists in the tree. Let *Y* be a parent node of *X*. There are three cases when a node is to be deleted from a BST. Let us consider each case:

1. *X* is a leaf node.
2. *X* has one child.
3. *X* has both child nodes.

Case 1: Leaf node deletion If the node to be deleted, say *X*, is a leaf node, then the process is easy. We need to change the child link of the parent node, say *Y* of node to be deleted to `Null`, and free the memory occupied by the node to be deleted and then return. Consider the following tree given in Fig. 7.57. Here, 5 is the node to be deleted.

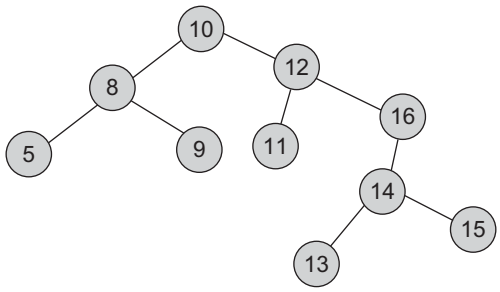


Fig. 7.57 Binary search tree

After deleting the node with data = 5, the BST becomes as in Fig. 7.58.

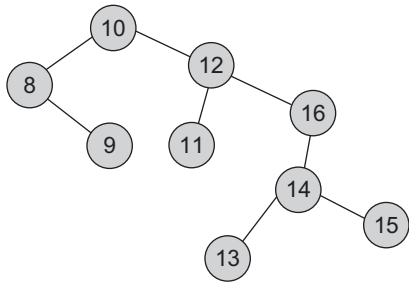


Fig. 7.58 BST after deletion of node with data = 5

Case 2(a): Node not having right subtree If the node to be deleted has a single child link, that is, either right child or left child is `Null` and has only one subtree, the process is still easy. If there is no right subtree, then just link the left subtree of the node to be deleted to its parent and free its memory. If *X* denotes the node to be deleted and *Y* is its parent with *X* as a left child, then we need to set *Y*->Lchild = *X*->Lchild and free the memory. If *X* denotes the node to be deleted and *Y* is its parent with *X* as a right child, then we need to set *Y*->Rchild = *X*->Lchild and free the memory. Let the node to be deleted be with data = 16 and data = 8; the resultant tree is as shown in Figs 7.59(a) and (b), respectively.

Case 2(b): Node not having left subtree If there is no left subtree, then just link the right subtree of the node to be deleted to its parent and free its memory. If *X* denotes the node to be deleted and *Y* is its parent with *X* as a left child, then we need to set *Y*->Lchild = *X*->Rchild and free the memory. If *X* denotes the node to be deleted and *Y* is its parent with *X* as a right child, then we need to set *Y*->Rchild = *X*->Rchild and free the memory. Let the node to be deleted be with data = 5 and data = 12; the resultant tree is as in Figs 7.59(c) and (d), respectively.

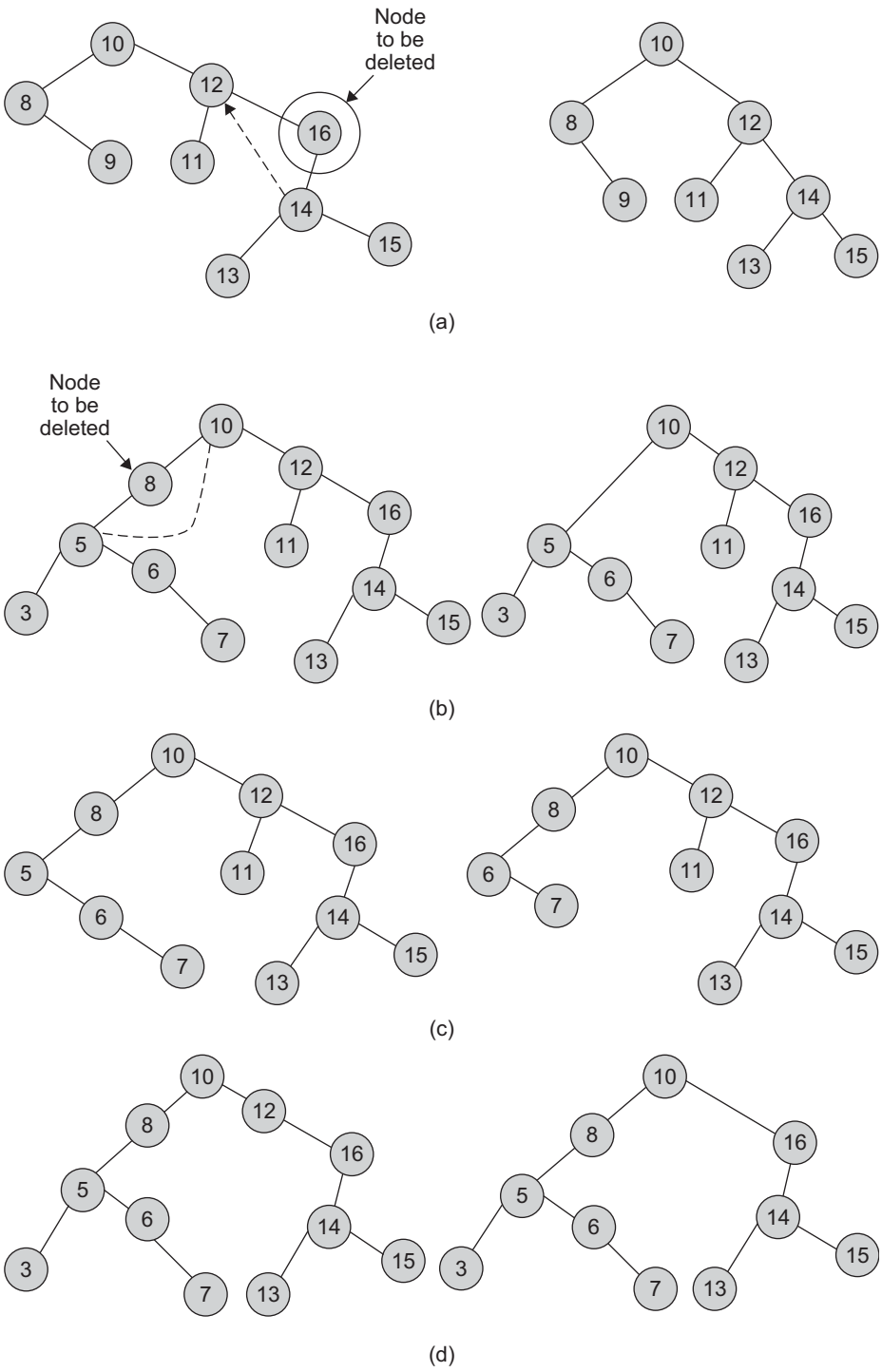


Fig. 7.59 Resultant tree after deletion of node with no left or right subtree
(a) Delete 16 (b) Delete 8 (c) Delete 5 (d) Delete 12

Case 3: Node having both subtrees Consider the case when the node to be deleted has both right and left subtrees. This problem is more difficult than the earlier cases. The question is which subtrees should the parent of the deleted node be linked to, what should be done with the other subtrees, and where should the remaining subtrees be linked. One of the solutions is to attach the right subtree in place of the deleted node, and then attach the left subtree onto an appropriate node of the right subtree. This is pictorially shown in Fig. 7.60.

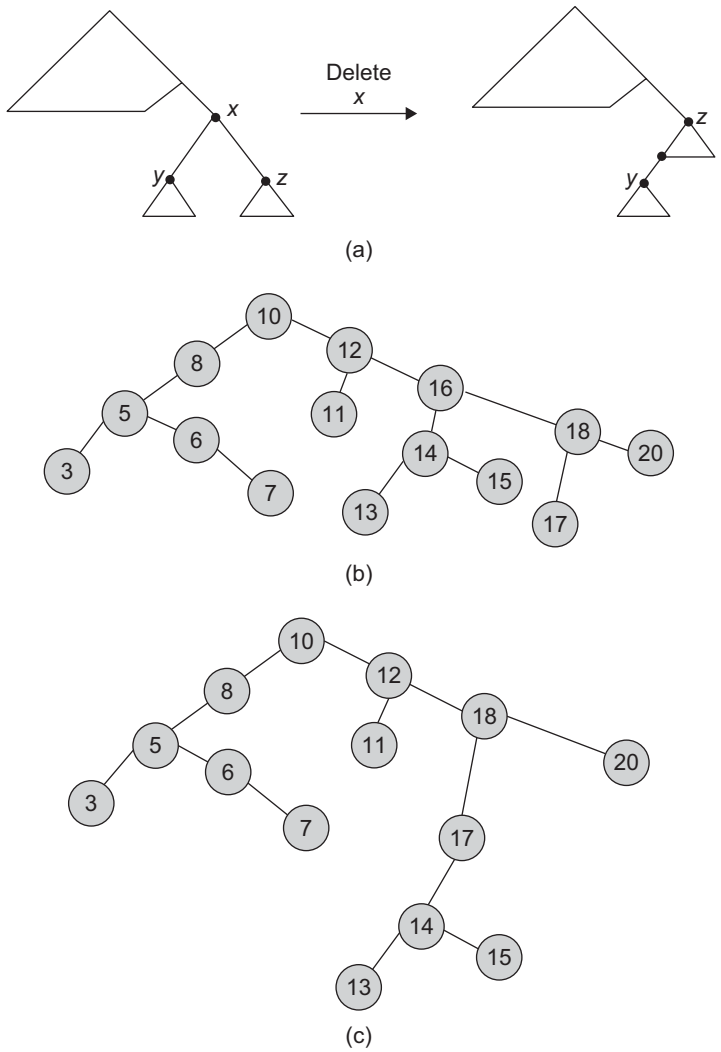


Fig. 7.60 Deleting node with both subtrees (a) Delete *x* (b) Delete 16 (c) Resultant tree after deletion

Another way to delete *X* from *T* is by first deleting the inorder successor of the node *X*, say *Z*, then replace the data content in the node *X* by the data content in the node *Z* (successor of the node *X*). Inorder successor means the node that comes after the node *X* during the inorder traversal of *T*.

Let us consider Fig. 7.61, and let the node to be deleted be the node with data 12.

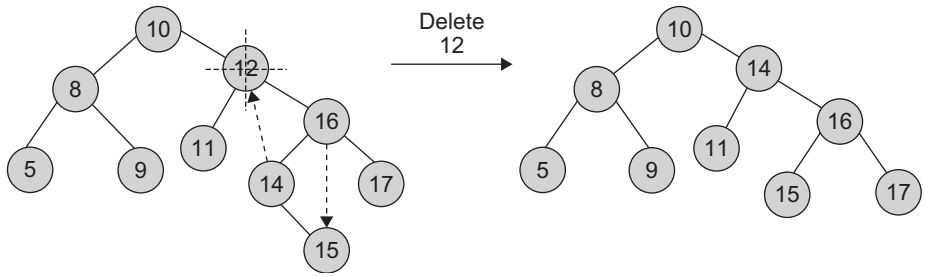


Fig. 7.61 Deleting the node with data = 12

In this process, we are actually trying to maintain the properties and the structure of a binary tree as much as possible. While deleting a node with both subtrees, we attempt searching the best suitable node to place at the deleted node. There are two alternatives to achieve so:

1. One can search for the largest data in the deleted node's left subtree and replace the deleted node with it.
2. One can search for the smallest data from the deleted node's right subtree and replace the deleted node with it.

Program Code 7.13 includes implementation of the `delete()` function with all cases such as the node to be deleted being a leaf node or the node having one child or the node having both child nodes.

PROGRAM CODE 7.13

```
// function to delete a node from BST
TreeNode *BSTree :: del(int deldata)
{
   int found = 0;
   int flag;
   TreeNode *temp = Root, *parent, *x;
   if(Root == Null)
   {
      cout << endl << "\t BST is empty";
      return Null;
   }
```

```
    else
    {
       parent = temp;
       //Search a BST node to be deleted & its parent
       while(temp != Null)
       {
          if(temp->Data == deldata )
             break;      // found
          if(deldata < temp->Data)
          {
             parent = temp;
             temp = temp->Lchild;
          }
          else
          {
             parent = temp;
             temp = temp->Rchild;
          }
       }      // end of search
       if(temp == Null)
          return(Null);
       else
       {
          //case of BST node having right children
          if(temp->Rchild != Null)
          {
             //find leftmost of right BST node
             //cout << "\n Temp is having right child";
             parent = temp;
             x = temp->Rchild;
             while(x->Lchild != Null)
             {
                parent = x;
                x = x->Lchild;
             }
             temp->Data = x->Data;
             temp = x;
          }
          //case of BST node being a leaf Node
        if(temp->Lchild == Null && temp->Rchild == Null)
          {
             //cout << "\n Leaf node";
            if(temp != root)
```

```
            {
               if(parent->lLchild == temp)
                  parent->Rchild = Null;
               else
                  parent->Rchild = Null;
            }
            else
               root = Null;
            delete temp;
            return(root);
         }
         else if(temp->Lchild!=Null&&temp->Rchild ==
         Null)
            //case of BST node having left children
         {
            //cout << "\n only left";
            if(temp != root)
            {
               if(parent->Lchild == temp)
                  parent->Lchild = temp->Lchild;
               else
                  parent->Rchild = temp->Lchild;
            }
            else
               root = temp->Lchild;
            delete temp;
            return(root);
         }
      }
   }
}
```

7.10.4 Binary Tree and Binary Search Tree

We have studied both binary tree and BST. A BST is a special case of the binary tree. The comparison of both yields the following points:

1. Both of them are trees with degree two, that is, each node has utmost two children. This makes the implementation of both easy.
2. The BST is a binary tree with the property that the value in a node is greater than any value in a node's left subtree and less than any value in a node's right subtree.

3. The BST guarantees fast search time provided the tree is relatively balanced, whereas for a binary tree, the search is relatively slow.

Consider the binary tree and BST shown in Fig. 7.62.

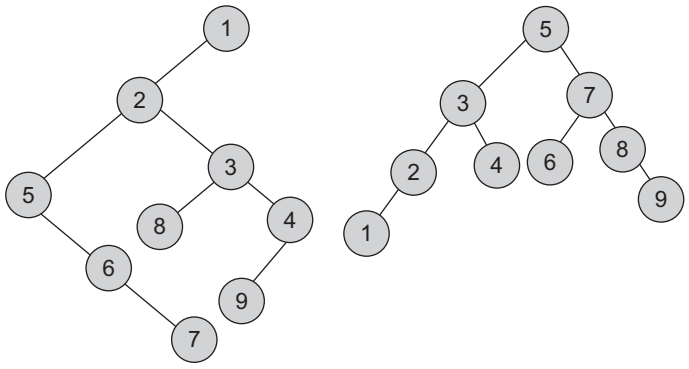


Fig. 7.62 Binary tree and binary search tree

In these trees, if we search for a node with data = 7, the number of searches varies in both trees as we need 5 comparisons for binary tree and 2 comparisons for BST.

Given a binary tree with no duplicates, we can construct a BST from a binary tree. The process is easy; one can traverse the binary tree and construct a BST for it by inserting each node in an initially empty BST.

7.11 THREADED BINARY TREE

We have studied the linked implementation of binary trees and the fundamental operations such as inserting a node, deleting a node, and traversing the tree. There are two key observations—first is that for all leaf nodes and those with one child the Lchild and/or Rchild fields are set to `Null`. The second observation is in the traversal process. The traversal functions use stack to store information about those nodes whose processing has not been finished. In case of non-recursive traversals, user-defined stack is used, and in case of recursive traversals internal stack is used. There is additional time for processing, but additional space for storing the stack is required. This is not a perceptible problem when a tree is of larger size.

To solve this problem, we can modify the node structure to hold information about other nodes in the tree such as parent, sibling, and so on. A.J. Perlis and C. Thornton have suggested replacing all the `Null` links by pointers, called *threads*. A tree with a thread is called a *threaded binary tree* (TBT). Note that both threads and tree pointers

are pointers to nodes in the tree. The difference is that the threads are not structural pointers of the tree. They can be removed but still the tree does not change. *Tree pointers* are the pointers that join and hold the tree together. Threads utilize the `Null` pointers' waste space to improve the processing efficiency. One such application is to use these `Null` pointers to make traversals faster. In a left `Null` pointer, we store a pointer to the node's inorder successor. This allows us to traverse the tree both left to right and right to left without recursion.

Though advantageous, we need to differentiate between a thread and a pointer. In the pictorial representation in Fig. 7.63, we draw the threads as dashed lines and the non-threads as solid lines.

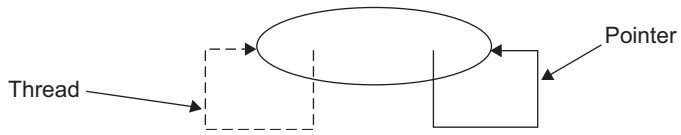


Fig. 7.63 Representation of threads and pointers

However, we need to differentiate between the thread and the pointer in actual implementation, that is, in the memory representation of a tree.

Let us use two additional fields—`Lbit` and `Rbit` to distinguish between a thread and a pointer.

Let us also use a function `IsThread()` that returns `true` if the pointer is a thread, and `false` if it is the conventional pointer to the child in the tree.

`Lbit`(node) = 1 if `Left`(node) is a thread

= 0 if `Left`(node) is a child pointer

`Rbit`(node) = 1 if `Right`(node) is a thread

= 0 if `Right`(node) is a child pointer

```
Class TBTNode
{
   boolean Lbit, Rbit;
   <Datatype> Data;
   TBTNode *Left, *Right;
}; y
```

Let us consider a tree and also a tree with threads as in Figs 7.64(a) and (b), respectively.

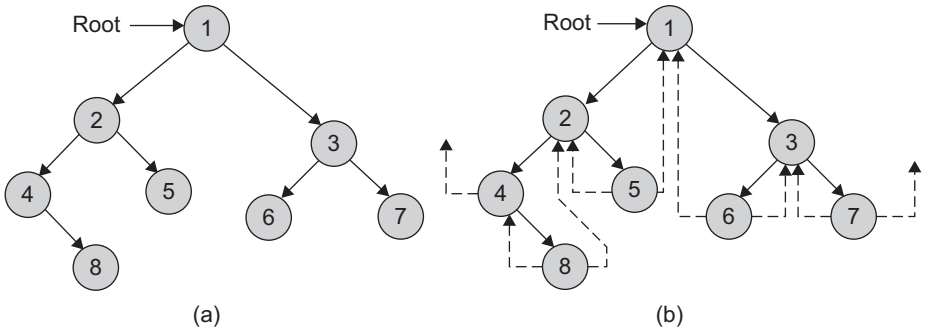


Fig. 7.64 Threaded binary tree (a) Tree (b) Corresponding threaded binary tree

In Fig. 7.64, note that the two threads Left(4) and Right(7) have been left dangling. To avoid such dangling pointers, we use an additional node, a *head node* of all threaded trees. The tree *T* is the left subtree of the head node. An empty tree is represented in Fig. 7.65.

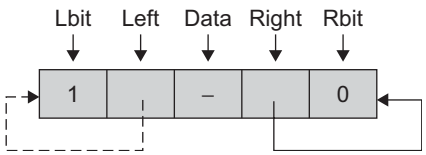


Fig. 7.65 An empty threaded binary tree

The tree in Fig. 7.64 has its TBT drawn. In the TBT as in Fig. 7.65, two threads that are the left thread of a node with data = 4 and the right thread of a node with data = 7 are dangling as they remain unassigned. To avoid this, a head node is added in the tree. The tree in Fig. 7.64 can be redrawn as in Fig. 7.66.

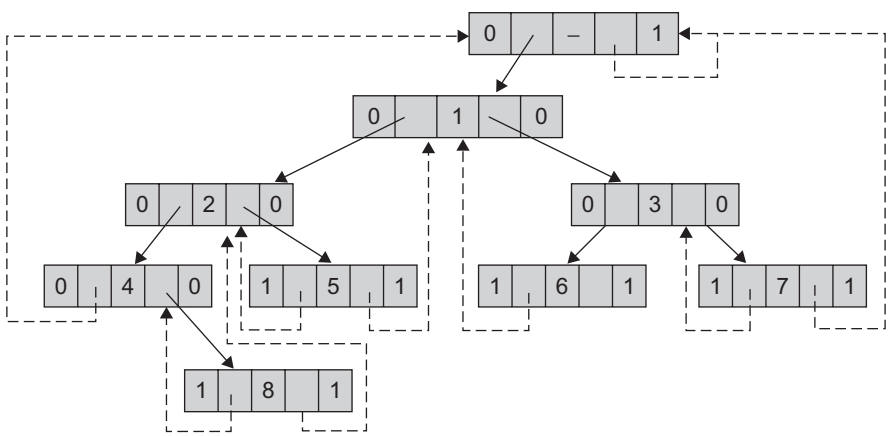


Fig. 7.66 Memory representation of TBT in Fig. 7.64

Here, 1 (true) shows it's a thread and 0 (false) represents that it's not a thread but a pointer to the child subtree.

7.11.1 Threading a Binary Tree

To build a threaded tree, first build a standard binary tree. Then, traverse the tree, changing the `Null` right pointers to point to their successors, for inorder TBT. Let `succ(N)` be a function that returns the inorder successor of the node *N*. Let `pred(N)` be a function that is an inorder predecessor of node *N*. The successor of *N* is the node that would be printed after *N* when traversing the tree in inorder. The predecessor of *N* is the node that immediately precedes the node *N* when traversing the tree in inorder. Hence, for inorder TBT, we replace `Right(N)` (if `Null`) to `succ(N)` and replace `Left(N)` (if `Null`) to `pred(N)`. In Fig. 7.66, the inorder successor of 8 is 2, so the right child of 8 is made a thread which is pointing to 2 and `Rbit` field is made 4.

If we are given a binary tree, it is natural to think how to set threads so that the tree becomes a threaded tree. Threading a binary tree is an interesting problem. The first idea could be to find each `Null` pointer and insert the proper thread. However, when we reach a `Null` pointer, we have no way to determine what the proper thread is. The proper approach would be based on taking any non-leaf (branch) node and setting the threads that would point to it. The successor and predecessor of a node *A* are defined as follows:

1. *successor*—the leftmost node in *A*'s right subtree
2. *predecessor*—the rightmost node in *A*'s left subtree

The algorithm must traverse the tree level-by-level, setting all the threads that should point to each node as it processes the node. Therefore, each thread set before the node containing the thread is processed. In fact, if the node is a leaf node, it need not be processed at all.

Let us use a queue to traverse the tree by level. We need to traverse the tree once using the helper data structure for threading, and it can later be traversed without any helper data structure such as stack. After the threads are inserted to the node being processed, its children go on the queue. During preprocessing, the thread to the header must be inserted in the tree's leftmost node as the left thread, and the thread to the header must be inserted in the tree's rightmost node as the right thread.

In fact, there are three ways to thread a binary tree while corresponding to inorder, preorder, and postorder traversals.

1. The TBT corresponding to inorder traversal is called *inorder threading*.
2. The TBT corresponding to preorder traversal is called *preorder threading*.
3. The TBT corresponding to postorder traversal is called *postorder threading*.

To build a TBT, there is one more method that is popularly used. In this method, the threads are created while building the binary tree. Program Code 7.14 is the C++ code to demonstrate this method.

PROGRAM CODE 7.14

```
// function to create inorder threaded binary search tree
void ThreadedBinaryTree :: create()
{
   Char ans;
   int flag;
   TBTNode *node, *temp;
   Head = new TBTNode;      // create head
   Head->Left = Head;
   Head->Right = Head;
   Head->Rbit = Heat->Lbit = 1;
   // create root for TBST
   Root = new TBTNode;
   cout << "\n Enter data for root";
   cin >> Root->data;
   Root->Left = Head;
   Root->Right = Head;
   // attach root to left of Head
   Head->Left = Root;
   // make thread bit of root 0
   Root->Lbit = Root->Rbit = 0;
   do
   {
      //  create new node for a tree
      node = newTBTNode;
      cout << "\n Enter data";
      cin >> node- data;
      node->Lbit = node->Rbit = 1;
      temp = Root;
      while
      {
         if(node->data < temp->data)
         {
            if(temp->Lbit == 1)
            // check leaf node and attach
            {
               node->Left = temp->Left;
               node->Right = temp;
               // attach node to left of temp
               temp->Lbit = 0;
```

```
                temp->Left = node;
                break;
             }
             else
                temp = temp->Left;
          }
          else
          {
             if(temp->Rbit == 1)      // is thread?
             {
                node->Left = temp;
                node->Right = temp->Right;
                // attaching node to right of temp
                temp->Right = node;
                temp->Rbit = 0;
                break;
             }
             else
                temp = temp->Right;
          }
       }       // end of while
       cout >> "Do you want to add more?";
       cin >> ans;
    }
    while(ans == 'y'||ans == 'Y');
}      // end of create
```

Sample Run

1. Insert root data 50 (Fig. 7.67).

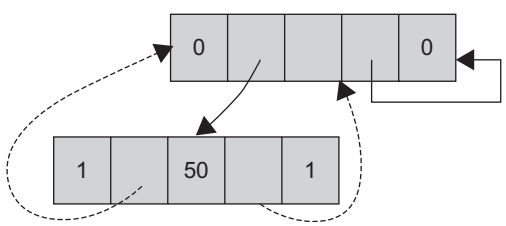


Fig. 7.67 Insert root 50

2. Attach the node with data 30 (Fig. 7.68).

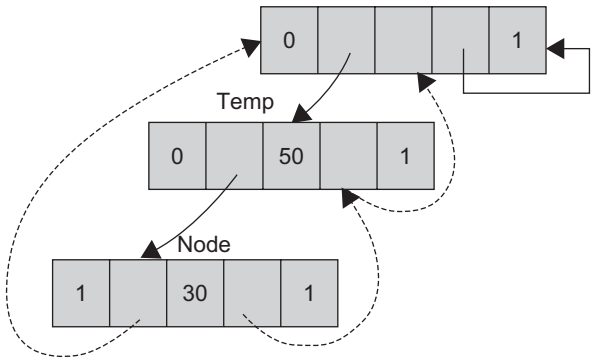


Fig. 7.68 Insert 30

Attach the left of `temp` copy to the left of the `temp` node to make the right child of the node as `temp`.

3. Attach the node with data 60 (Fig. 7.69).

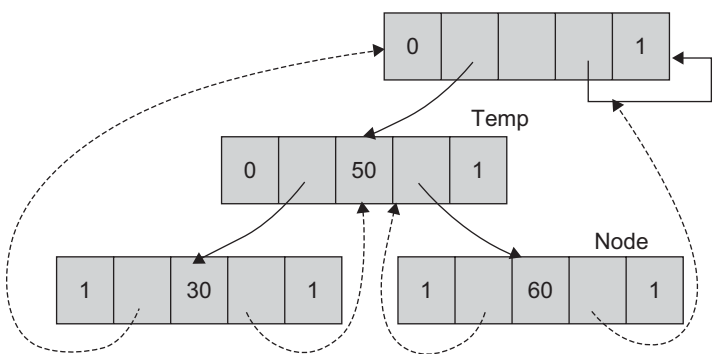


Fig. 7.69 Insert 60

Copy the right of `temp` to the right of the node to make the left of node as `temp`. Attach the node to the right of temp to make `Rbit` 0.

4. Attach the node with data 55 (Fig. 7.70).

Copy the left of `temp` to the left of node. Make the right of node as `temp`. Then, attach `tnode` to the left of `temp`. Make the `Lbit` of `temp` 0.

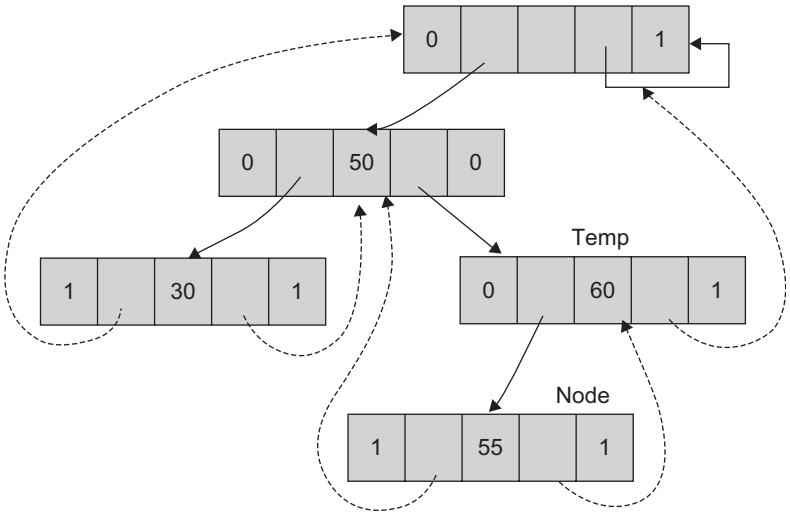


Fig. 7.70 Insert 55

The implementation of function for creating a TBT is given in Program Code 7.15.

PROGRAM CODE 7.15

```
//Function to create a tree as per user choice
void TBTree::create()
{
   TBTNode *temp,*prev;
   char ch,x;
   Root = Null;
   do
   {
      temp = new TBTNode;
      temp->left = temp->right = head;
      temp->lbit = temp->rbit = 0;
      cout << "\nEnter the char data:";
      cin >> temp->data;
      if(Root == Null)
      {
         Root = temp;
         head->left = Root;
         head->lbit = 1;
      }
      else
     {
```

```
            prev = Root;
            while(prev != Null)
            {
               cout << "Left child or Right child of (r/l):
               " << prev->data << " :";
               cin >> x;
               if(x == 'l' || x == 'L')
               {
                  if(prev->lbit == 0)
                  {
                     temp->left = prev->left;
                     prev->left = temp;
                     prev->lbit = 1;
                     temp->right = prev;
                     break;
                  }
                  else
                     prev = prev->left;
               }
               else
               {
                  if(x == 'r' || x == 'R')
                  {
                     if(prev->rbit == 0)
                     {
                        temp->right = prev->right;
                        prev->right = temp;
                        prev->rbit = 1;
                        temp->left = prev;
                        break;
                     }
                     else
                        prev = prev->right;
                  }
               }
            }
         cout <<"Do you want to Add more?";
         cin >> ch;
      }
      while(ch == 'y' || ch == 'Y');
   }
```

7.11.2 Right-threaded Binary Tree

In a right-threaded binary tree each `Null` right link is replaced by a special link to the successor of that node under inorder traversal, called a *right thread*. The right thread will help us to traverse freely in inorder since we need to only follow either an ordinary link or a thread to find the next node to visit. When we replace each `Null` left link by a special link to the predecessor of the node (left thread) under inorder traversal, the result is fully a TBT.

7.11.3 Inorder Traversal

It can be realized that the inorder traversal in an inorder TBT is very easy. However, the other traversals are a bit difficult. If the preorder or postorder threading of a binary tree is known, then the corresponding traversal can be obtained efficiently.

The code for inorder traversal of a TBT is listed in Program Code 7.16.

PROGRAM CODE 7.16

```
// Traverse a threaded tree in inorder
void TBTree::Inorder()
{
   TBTNode *temp;
   temp = Root;
   int flag = 0;
   if(Root == Null)
   {
      cout << "\nTree not present";
   }
   else
   {
      while(temp != head)
      {
         if(temp->lbit == 1 && flag == 0)
         // go to left till Lbit is 1(till child)
         {
            temp = temp->left;
         }
         else
         {
            cout << temp->data << " ";     // display data
            if(temp->rbit == 1)     // go to right by child
            {
               temp = temp->right;
               flag = 0;
```

```
            }
            else      // go to right by thread
            {
               temp = temp->right;
               flag = 1;
            }
         }
      }
   }
}
```

Note that this traversal does not use stack, whereas for non-threaded binary tree, we require a stack as an intermediate data structure.

The computing time is O(n) for a binary tree with n nodes.

Example of inorder traversal of threaded binary tree Figure 7.71 shows a TBT whose inorder traversal sequence is—Megha, Amit, Arvind, Varsha, Abolee, Hemant, Saurabh.

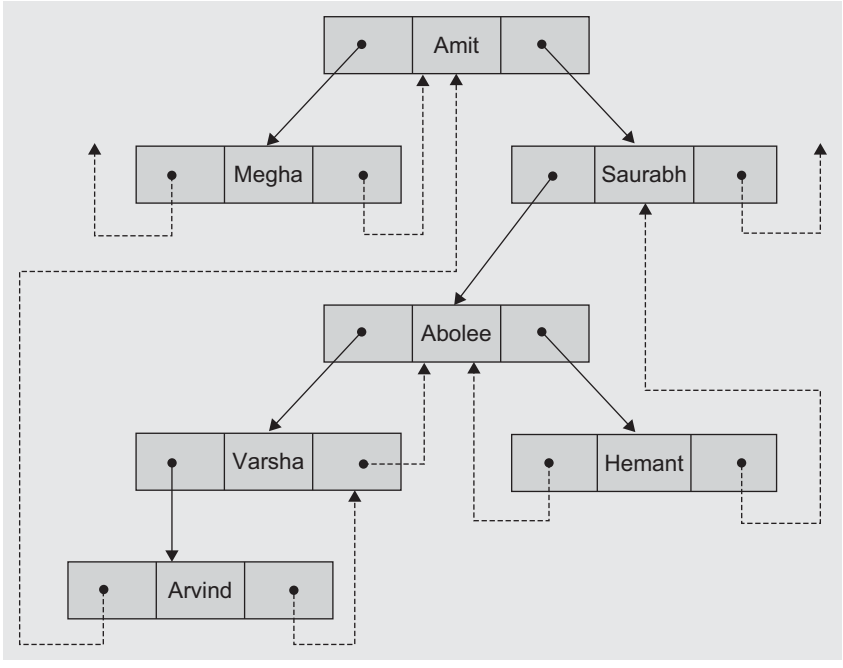


Fig. 7.71 Inorder traversal of a TBT

7.11.4 Preorder Traversal

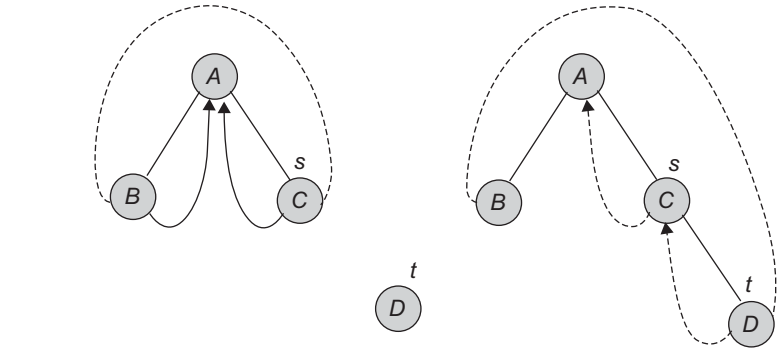
These threads also simplify the algorithm for preorder and postorder traversals. Program Code 7.17 is the C++ routine for preorder traversal of a TBT.

PROGRAM CODE 7.17

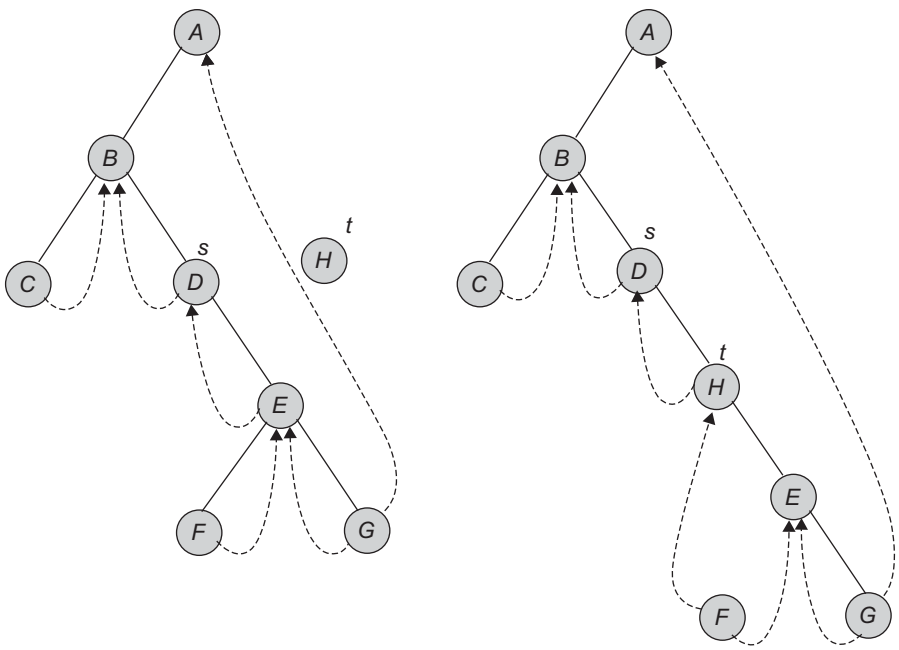
```
// Traverse a threaded tree in preorder
void TBTree :: preorder()
{
   TBTNode *temp;
   int flag = 0;
   temp = Root;
   while(temp != head)
   {
      if(flag == 0) cout << temp->data <<" ";
      if(temp->lbit == 1 && flag == 0)     // go left till
      lbit is 1
      {
         temp = temp->left;
      }
      else if(temp->rbit == 1)      // go to right by child
      {
         temp = temp->right;
         flag = 0;
      }
      else      // go to right by thread
      {
         temp = temp->right;
         flag = 1;
      }
   }
}      //End of function
```

7.11.5 Insert to Right of a Node

Consider Figs 7.72(a) and (b). We want to insert the node *t* to the right of the node *s* in both the threaded trees.



(a)



(b)

Fig. 7.72 Inserting nodes in a TBT (a) Inserting node *D* (b) Inserting node *H*

7.11.6 Deleting a Node

Consider Fig. 7.73. We want to delete the node labelled *D* from the TBT.

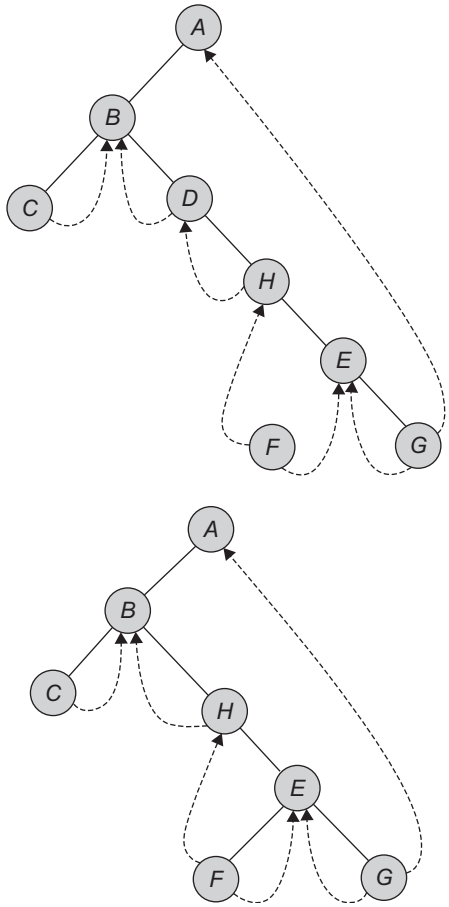


Fig. 7.73 Deleting a node from a TBT

7.11.7 Pros and Cons

A TBT has some advantages and disadvantages over a non-threaded binary tree. They are as follows:

1. The traversal for a TBT is straightforward. No recursion or stack is needed. Once we locate the leftmost node, we loop following the thread to the next node. When we find the null thread, the traversal is complete.

2. At any node, the node's successor and predecessor can be located. In case of non-threaded binary tree, this task is time consuming and difficult. In addition, stack is needed for the same.
3. Threads are usually more upward, whereas links are downward. Thus, in a threaded tree, we can traverse in either direction, and the nodes are in fact circularly linked. Hence, any node can be reached from any other node.
4. Insertions into and deletions from a threaded tree are time consuming as the link and thread are to be manipulated.

7.12 APPLICATIONS OF BINARY TREES

There is a vast set of applications of the binary tree in addition to searching. The applications discussed in this section are gaming, expression tree, Huffman tree for coding, and decision trees.

7.12.1 Expression Tree

A binary tree storing or representing an arithmetic expression is called as *expression tree*. The leaves of an expression tree are *operands*. Operands could be variables or constants. The branch nodes (internal nodes) represent the operators. A binary tree is the most suitable one for arithmetic expressions as it contains either binary or unary operators. The expression tree for expression *E*, is shown in Fig. 7.74.

$$\text{Let } E = ((A \times B) + (C - D))/(C - E)$$

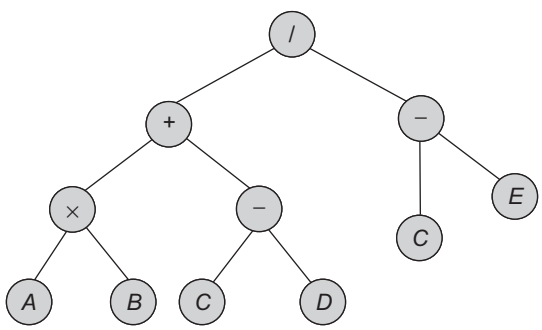


Fig. 7.74 Expression tree for $E = ((A \times B) + (C - D))/(C - E)$

We have studied that the Polish notations are very useful in the compilation process. There is a close relationship between binary trees and expressions in prefix and postfix notations.

In the expression tree as in Fig. 7.74, an infix expression is represented by representing the node as an operator, and the left and right subtrees are the left and right operands of that operator.

If we traverse this tree in preorder, we visit the nodes in the order of: $/ + \times AB - CD - CE$, and this is a prefix form of the infix expression. On the other hand, if we traverse the tree in postorder, the nodes are visited in the following order: $AB \times CD - E - /$, which is a postfix equivalent of the infix notation.

Example 7.7 represents the postfix equivalent of a given infix notation.

EXAMPLE 7.7 Represent $AB + D \times EFAD \times + / + C +$ as an expression tree.

Solution Figure 7.75 represents the given expression in the form of a tree.

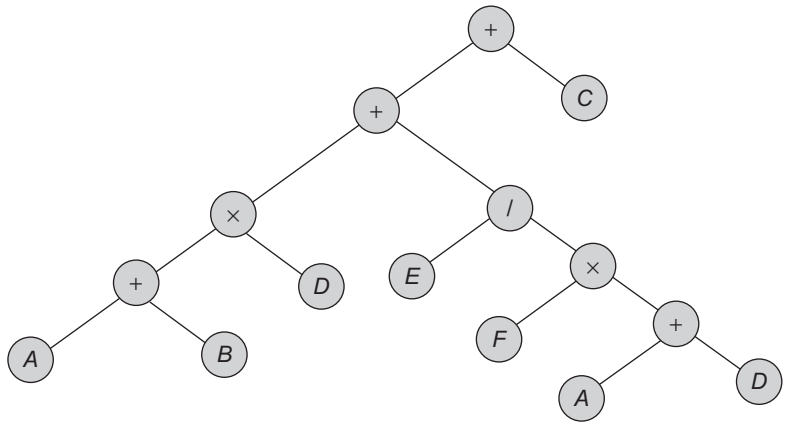


Fig. 7.75 Expression tree for $E = AB + D \times EFAD \times + / + C +$

Construction of Expression Tree

We have studied the binary tree representation of an expression. Let us study how to construct a tree when the infix expression is given. First, the infix expression is converted to a postfix expression. Use Algorithm 7.3 to construct an expression tree.

ALGORITHM 7.3

(Scan the postfix expression from left to right.)

1. Get one token from expression E.
2. Create a node say `curr` for it.

3. If (symbol is operand) then
   (a) push a node `curr` onto a stack.
4. else if (symbol is operator) then
   (a) T_2 = `pop()`
   T_1 = `pop()`
   Here T_1 and T_2 are pointers to left trees and right subtrees of the operator, respectively.
   (b) Attach T_1 to left and T_2 to the right of `curr`.
   (c) Form a tree whose root is the operator and T_1 and T_2 are left and right children, respectively.
   (d) Push the node `curr` having attached left and right subtrees onto a stack.
5. Repeat steps 1–4 till the end of expression.
6. Pop the node curr from the stack, which is a pointer to the root of expression tree.

Example 7.8 shows the steps to construct an expression tree for a given expression, *E*.

EXAMPLE 7.8 Represent $E = (a + b \times c)/d$ as an expression tree.

Solution Let us consider the expression $E = (a + b \times c)/d$

Postfix expression = $abc \times + d/$

The following steps of operations are performed:

1. The operands *a*, *b*, *c* will be pushed onto the stack by forming a one-node tree of each and pushing a pointer to each onto a stack (Fig. 7.76).

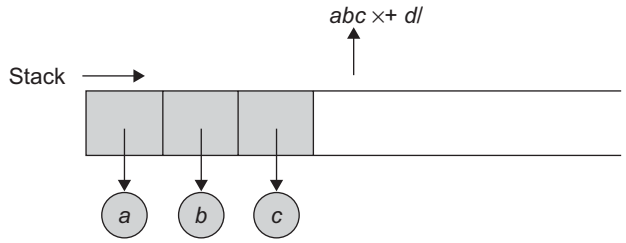


Fig. 7.76 Step 1

2. When the operator × has been encountered, the top two pointers are popped. A tree is formed with × as a root and the two popped pointers as children. The pointer to the root is pushed onto a stack (Fig. 7.77).

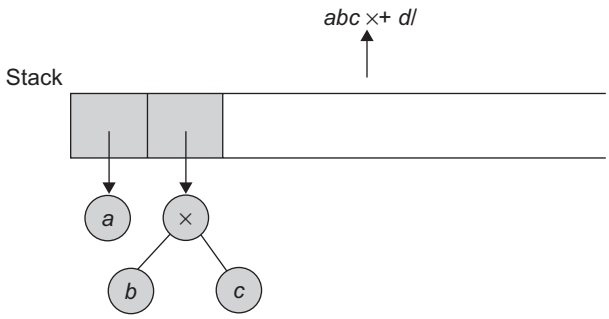


Fig. 7.77 Step 2

3. After the operator + has been encountered, the procedure as in step 2 is executed (Fig. 7.78).

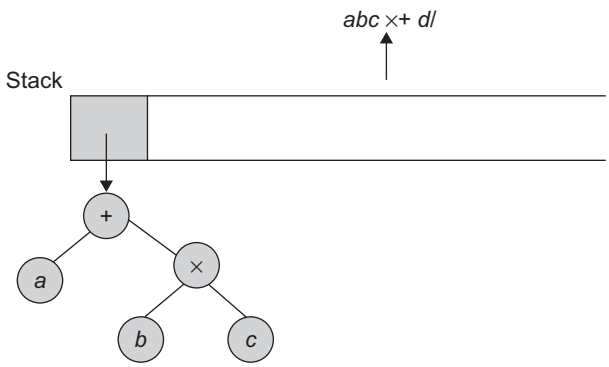


Fig. 7.78 Step 3

4. As the operand *d* has been encountered, it is pushed as a pointer to the one-node tree (Fig. 7.79).

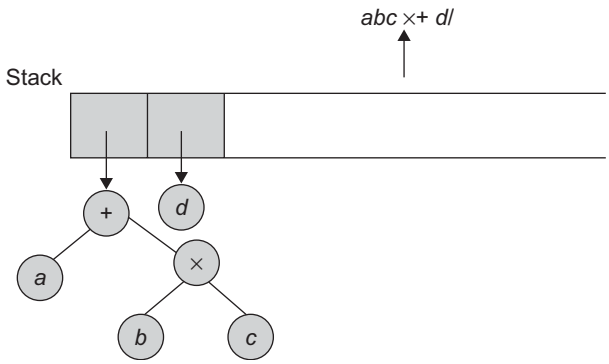


Fig. 7.79 Step 4

5. After the operator has been encountered, it follows the procedure as in step 2 (Fig. 7.80).

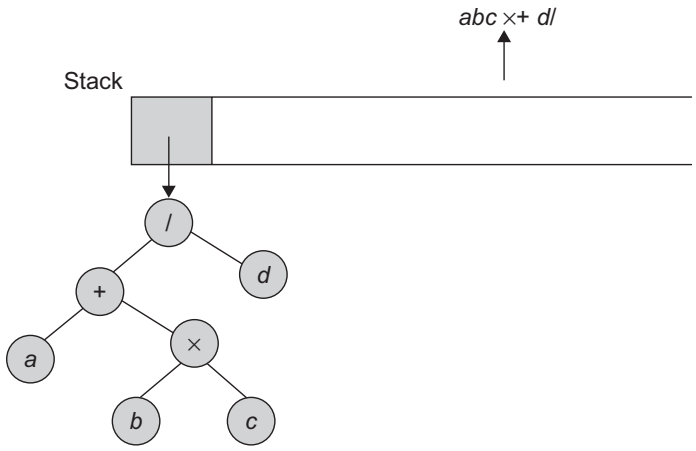


Fig. 7.80 Step 5

6. Pop the stack as the expression has been processed. This returns a pointer to the expression tree's root.

7.12.2 Decision Tree

In practice there are many applications which use trees to store data for rapid retrieval, the most useful application being decision making. These applications, along with a tree as one of the data structures, often oblige some additional structures on the tree. Consider an example tree, a BST. In the BST, the data in a left or right subtree has a particular relationship with the data in the node (such as being greater than or smaller than the data).

We can use trees to arrange the outcomes of various decisions in the required order. We can denote these actions in the form of a tree, called the decision tree.

The *decision tree* is a classifier in the form of a tree where each node is either a branch node or a leaf node. Here the leaf node denotes the value of the target attribute (class) of examples and the branch node is a decision node that denotes some test to be carried out and takes a decision based on a single attribute value, with one branch and subtree for each possible outcome of the test. A decision tree can be used for classification by starting at the root of the tree and moving through it until a leaf node that provides the classification of the instance is reached. The decision tree training is a typical inductive approach to gain the knowledge on classification.

For example, consider the execution of a C program. The initial part of the program contains pre-processor directives followed by global variables and functions including

the main function. Initially, the operating systems need to load the code and constants, initialize variables (if any), read the input data, and print the required information. The sequence of these actions depends on the code written. Generally, most programs involve more than simple input and output statements. The program includes conditional statements that use `if` or `case` statements. It may also include unconditional loops (`for` loop) or conditional loops (`while` and `repeat` loops).

In such cases, the program execution flow depends on the results of testing the values of the variables and expressions. For example, after testing a Boolean expression in an if statement, the program may execute the statements following it or the statements in the corresponding `else` part. Similarly, after examining a `while` condition, the program may repeat the code within the loop, or may continue with the code following the loop.

We can visualize these different ways in which a program may execute through a decision tree. Execution of a C program starts with a call to the function `main()`, or we can represent the root of the decision tree with the code that is always run at the start of a program till the first conditional statement. However, at the first conditional statement, the program executes one code segment or another depending upon the value of the condition. In such a situation, the decision tree can be drawn with a child of the root for each code option that the program code follows. For an `if` statement, there are two children of the root—one if the Boolean expression is true where the `then` clause is executed, and another in case the expression is false. For a `case` statement, a different child is drawn for each different case identified by the code, because different paths are followed for each of these situations. Figure 7.81 illustrates all such cases.

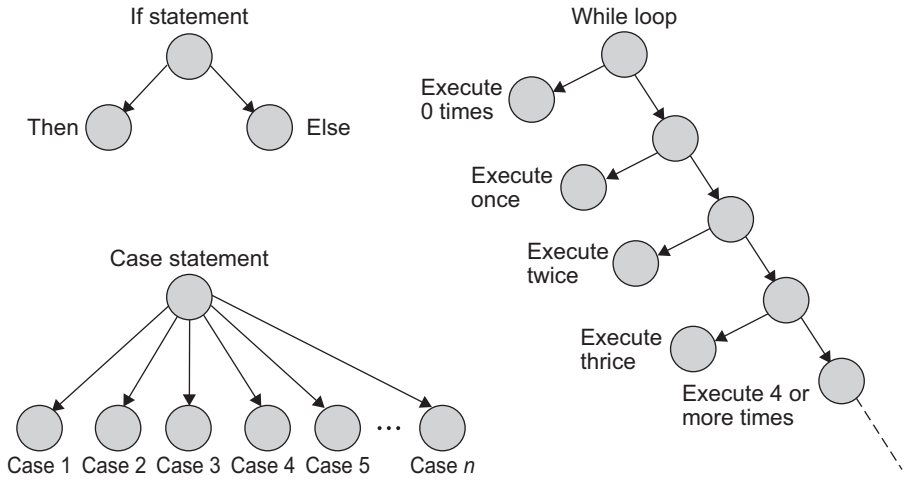


Fig. 7.81 Decision trees for program structures

Figure 7.80 represents pseudo-structures for the Pascal language. In the figure, for a `while` loop, the body of the loop might be executed 0, 1, 2, 3, or more times, after which the program execution continues with the code that follows the `while` statement. Conversely, the execution flow first tests the `exit` condition and then either exits the loop or starts the body of the loop for the first time. If the loop is executed for the first time, then the execution arrives at the `exit` condition a second time and either exits the loop or continues with the loop a second time. The work within the loop continues until the system tests the `exit` condition and determines that the loop should not continue. In tracing the program execution, we add each individual decision into another branch inside a general decision tree.

The example demonstrates the usefulness of the decision trees to demonstrate and test all possible execution paths that might be followed within a program or a piece of code. Decision trees not only provide a mechanism for demonstrating the code execution but they also provide a structure for examining how general algorithms might work. Consider an example of searching an element in a sorted list. We can use a decision tree to demonstrate the working of searching of member in the list. For binary search to be applied on the sorted list, various comparisons are required for deciding which set of elements are to be further searched or whether the search is to be terminated. In a BST, when an element is initially compared with the root, if the search is successful, the process terminates. If the element is lesser than the root, then it is searched in the left subtree, and otherwise in right subtree.

The advantages of decision trees are the following:

1. Decision trees are most suitable for listing all possible decisions from the current state.
2. They are suitable for classification without the need for many computations.

Decision trees are popularly used in expert systems. Although decision trees seem to be very useful, they suffer from a few drawbacks, such as the following:

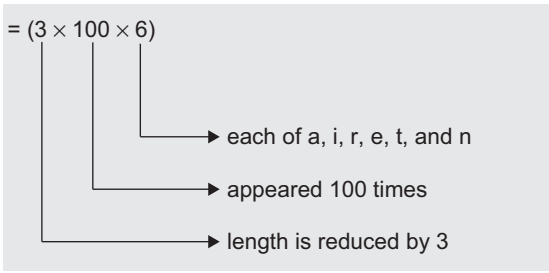
1. Decision trees are prone to errors in classification problems with more classes
2. They can be computationally expensive for complex problems.

7.12.3 Huffman's Coding

One of the most important applications of the binary tree is in communication. Consider an example of transmitting an English text. We need to represent this text by a sequence of 0s and 1s. To represent the message made of English letters in binary, we represent each alphabet in the binary form, that is, as a sequence of 0s and 1s. Each alphabet must be represented with a unique binary code. We need to assign a code, each of length 5 to each letter in the alphabet as $2^4 < 26 < 2^5$. Now to send a message, we have to simply transmit a long string of 0s and 1s containing the sequences for the letters in the message. At the receiving end, the message received will be divided into sequences of length 5, and the corresponding message is recognized.

Let us consider that a sequence of 1000 letters is to be sent. Now, the total bits to be transmitted will be 1000×5, as we represent each letter with 5 bits. It may happen that among those 1000 letters, the letters a, i, r, e, t, and n appeared maximum number of times in the sequence.

It is observed that the letters in the alphabet are not used with uniform frequencies. For example, the letters e and t are used more frequently than x and z. Hence, we may represent the more frequently used letters with shorter sequences and less frequently used letters with longer sequences so that the overall length of the string will be reduced. In this example, if a, i, r, e, t, and n are assigned say in the sequence of length 2, and let us assume that each one of them appeared 100 times among the sequence of 1000 letters. Now, the length will be reduced by a factor



Such a coding is called as *variable length coding*. Even though the variable length coding reduces the overall length of the sequence to be transmitted, an interesting problem arises. When we represent the letters by the sequences of various lengths, there is the question of how one at the receiving end can unambiguously divide a long string of 0s and 1s into the sequences corresponding to the letters. Let us consider an example. Let us use the sequence 00 to represent the letter 'a', 01 to represent letter 'n', and 0001 to represent the letter 't'. Suppose we want to transmit a text of two letters 'an' by transmitting the sequence 0001. Now, at the receiving end, it is difficult to determine whether the transmitted sequence was 'an' or 't'. This is because 00 is a prefix of the code 0001. We must assign variable sequences to the letters such that no code is the prefix of the other.

A set of sequences is said to be a *prefix code* if no sequence in the set is the prefix of another sequence in the set. For example, the set {000, 001, 01, 10, 11} is a prefix code, whereas the set {1, 00, 000, 0001} is not. Hence we must use prefix codes to represent the letters in alphabet. If we represent the letters in the alphabet by the sequences in a prefix code, it will always be possible to divide a received string into sequences representing the letters in a message unambiguously. One of the most useful applications of binary tree is in generating the prefix codes for a given binary tree. We label the two edges incident from each branch node with 0 and 1. To each leaf,

assign a code that is a sequence of labels of the edges in the path from the root to that of leaf (Fig. 7.82).

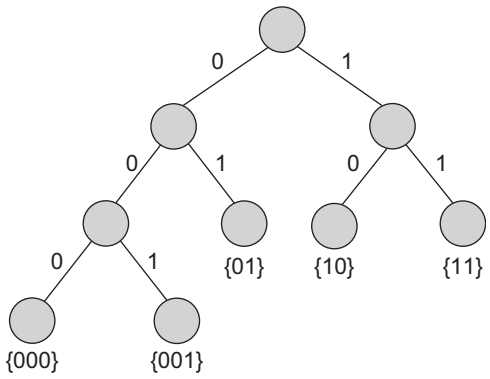


Fig. 7.82 Prefix codes

It is always possible to divide a received sequence of 0s and 1s into the sequences that are in a prefix code. Starting at the root of the binary tree, we shall trace a downward path in the tree according to the bits in the received sequence. At a branch node, we shall follow the edge labelled with 0 if we encounter a 0 in the received sequence, and we shall follow the edge labelled with a 1 if we encounter a 1 in the received sequence. When the downward path reaches a leaf, it shows that the prefix code has been detected. For the next sequence, we should return to the root of the tree. This process clearly assures that the variable length code, which is the prefix code, has no ambiguity.

Now the problem is about constructing a binary tree. Suppose we are given a set of weights $w_1, w_2, ..., w_n$. Let us assume that $w_1 \leq w_2 \leq ... \leq w_n$. A binary tree that has n leaves with weights $w_1, w_2, ..., w_n$ assigned to the leaves is called as binary tree for weight $w_1, w_2, ..., w_n$. Our aim is to assign smaller code to the leaf of higher weights, as the weights here denote the frequency of occurrence. The length of sequence of bits assigned to a leaf node is path length of that node. Hence, we want lesser path length to the leaf nodes with higher weights.

Let the weight of a tree T be denoted by w(T). The weight of a binary tree for weights $w_1, w_2, ..., w_n$ is given by

$$w(\mathrm{T}) = \sum_{i=1}^{n} w_i L(w_i), \text{ where}$$

$L(w_i)$ = path length of node of weight w_i.

A binary tree for weights $w_1, w_2, ..., w_n$ is said to be an optimal binary tree if its weight is minimum. Hence, our aim is to construct a tree such that w(T) is minimum.

D.A. Huffman has given a very elegant procedure to construct an optimal binary tree. Suppose we want an optimal tree for the weights w_1, w_2, ..., w_n. Let a be a branch node of largest path length in the tree. Suppose the weights assigned to the sons of a are w_b and w_c. Thus, $l(w_b) \geq l(w_1)$, and $l(w_b) \geq l(w_2)$. In addition, since the tree is optimal, we should have $l(w_b) \leq l(w_1)$, and $l(w_b) \leq l(\mathrm{w}_2)$.

Huffman's algorithm The algorithm is given as follows:

1. Organize the data into a row as ascending order frequency weights. Each character is the leaf node of a tree.
2. Find two nodes with the smallest combined weights and join them to form the third node. This will form a new two-level tree. The weight of the new third node is the addition of two nodes.
3. Repeat step 2 till all the nodes on every level are combined to form a single tree.

7.12.4 Game Trees

One of the most exciting applications of trees is in games such as tic-tac-toe, chess, nim, checkers, go, and so on. We shall consider tic-tac-toe as an example for explaining this application of trees.

The game starts with an empty board and each time a player tries for the best move from the given board position. Each player is initially assigned a symbol of either 'X' or 'O'. Depending on the board position the user has to decide how good the position seems to be for a player. For implementation we need to compute a value say, `WinValue`, which of course will have the largest possible value for a winning position, and the smallest value for a losing position. An example of such a `WinValue` computation could be the difference between the number of rows, columns, and diagonals that are left open for one player and those left open for the opponent game partner. Here we can omit the values 9 and –9 as they represent the values for a position that wins and loses, respectively. While computing this value we need not further search for other possible board positions that might result from the current positions, as it just estimates a motionless board position. We can write a function while implementing this game that computes and returns the `WinValue`. Let us name such a function as `ComputeWinValue()`. Considering all the possible positions, it is possible to construct a tree of the possible board positions that may result from each possible move, called a *game tree*.

Now with a given board position, we need to determine the next best move and for that we need to consider all the possible moves and respective resulting positions after the move. For a player the best move is the one that results in a board position with the highest `WinValue`. Careful observation leads to the conclusion that this calculation however, does not always yield the best move. A sample position and the five possible moves that player with symbol X can make from that current position is shown in Fig. 7.83.

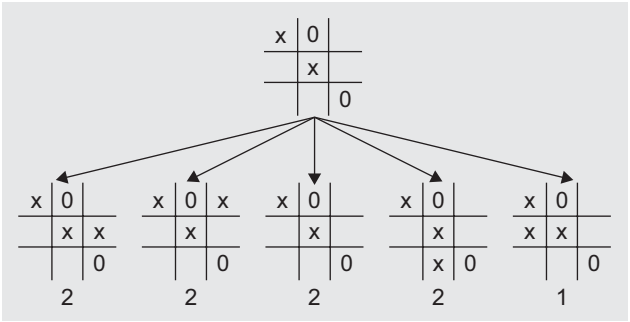


Fig. 7.83 An example game tree and WinValues of each possible move

Now if we compute `WinValue` for the five resulting positions, using the `ComputeWinValue()` function, we get the values as shown in Fig. 7.83. Among them four of the moves result with the same maximum `WinValue`. One can note that the move in the fourth position definitely leads to the victory for the player with the marking symbol X, and the other three moves would lead to the victory of the opponent with the symbol 0.

This shows that the move that yields the smallest `WinValue` is better than the moves that yield a higher `WinValue`. The static `ComputeWinValue()` function, therefore, is not sufficient to guess the result of the game. Hence we need to revise this function. We can have such a function for simple games such as tic-tac-toe, but often, games such as chess are too complex for static functions to determine the best possible move computation.

The best way to predict and play is to look ahead of several moves so as to get a significantly better choice of the next move. Let the variable `LookAhead` be the number of future moves to be taken care of. Considering all the possible positions, it is possible to construct a tree of the possible board positions that may result from each possible move as shown in Fig. 7.84 which shows the game tree for a tic-tac-toe game considering a look-ahead of level of 2.

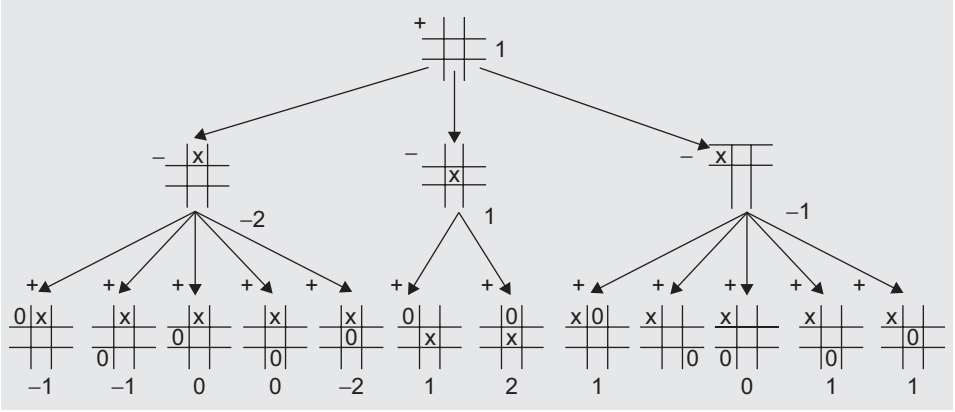


Fig. 7.84 A game tree for tic-tac-toe

Here the height of the tree is the maximum level of the nodes which represents the look-ahead level of the game. Let us denote the player who must begin the game with the '+' symbol (plus sign) and his or her opponent as '–' symbol (minus sign). Now we need to compute the best move for '+' from the root position. The remaining nodes of the tree may be designated as '+' nodes or '–' nodes, depending upon which player must move from that node's position. Each node of Fig. 7.84 is marked as a '+' node or '–' node, depending upon which player must move from that node's position. Consider the case where the game positions of all the child nodes of a '+' node have been evaluated for player '+'. Then obviously, a '+' should select the move that paves the way to the maximum `WinValue`. Thus, the value of a '+' node to player '+' is the maximum of the values of its child nodes. On the other hand, once '+' moves, '–' will choose the move that results in the minimum of the values of its child nodes.

For a player with the symbol 0, Fig. 7.85 shows the best possible moves.

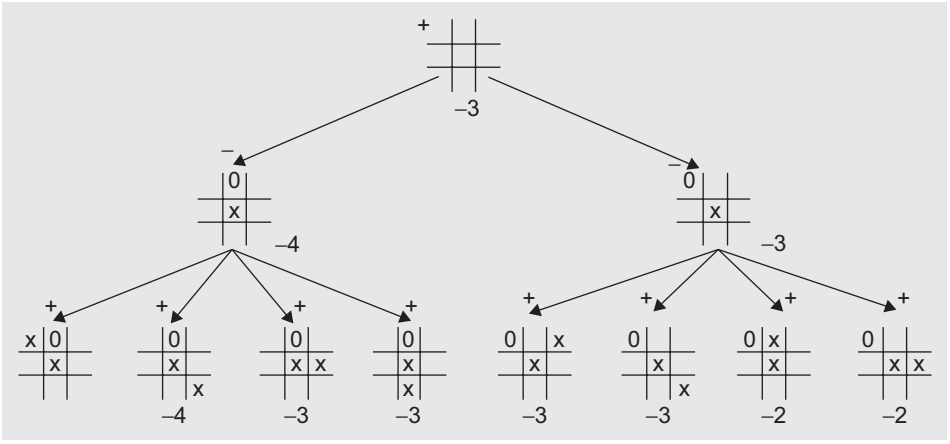


Fig. 7.85 Game tree showing best moves for a player with symbol 0

Note that the designation of '+' and '–' depends on whose move is being calculated. The best move for a player from a given position may by determined by first constructing the game tree and applying a static `ComputeWinValue()` function to the leaf nodes. Each node of the game tree must include a representation of the board and an indication of whether the node is a '+'node or a '–' node.

RECAPITULATION

- Non-linear data structures are those where every data element may have more than one predecessor as well as successor. Elements do not form any particular linear sequence. Tree and graph are two examples of non-linear data structure. Non-linear data structures are capable of expressing more complex relationship than linear data structure.

- Tree, a non-linear data structure, is a mean to maintain and manipulate data in many applications. Wherever the hierarchical relationship among data is to be preserved, tree is used.
- A binary tree is a special form of a tree. It is important and frequently used in various applications of computer science. A binary tree has degree two, and each node has utmost two children. This makes the implementation of tree easier. The implementation of binary tree should represent the hierarchical relationship between the parent node and its left and right children.
- Binary tree has the natural implementation in a linked storage. In a linked organization, we wish that all nodes should be allocated dynamically. Hence, we need each node with data and link fields. Each node of a binary tree has both a left and a right subtree. Each node will have three fields Lchild, Data, and Rchild.
- The operations on a binary tree include insert node, delete node, and traverse tree. Traversal is one of the key operations. Traversal means visiting every node of a binary tree. There are various traversal methods. For a systematic traversal, it is better to visit each node (starting from root) and its both subtrees in the same way.
- Let L represent the left subtree, R represent the right subtree, and D be node data. Three traversals are fundamental: LDR, LRD, and DLR. These are called as *inorder*, *postorder*, and *preorder* traversals because there is a natural correspondence between these traversals producing the infix, postfix, and preorder forms of an arithmetic expressions, respectively. In addition, a traversal where the node is visited before its children are visited is called a *breadth-first traversal*; a walk where the children are visited prior to the parent is called a *depth-first traversal*.
- The binary search tree is a binary tree with the property that the value in a node is greater than any value in a node's left subtree and less than any value in the node's right subtree. This property guarantees fast search time provided the tree is relatively balanced.
- The key applications of tree include the following: expression tree, gaming, Huffman coding, and decision tree.

KEY TERMS

Binary search tree A binary search tree (BST) is a binary tree that is either empty or where every node contains a key and satisfies the following conditions:

1. The key in the left child of a node, if it exists, is less than the key in its parent node.
2. The key in the right child of a node, if it exists, is greater than the key in its parent node.
3. The left and the right subtrees of the node are again BSTs.

Binary tree A binary tree has degree two, each node has atmost two children. A binary tree is either: an empty tree; or consists of a node, called root and two children, left and right, each of which are themselves binary trees.

Breadth- and depth-first traversals A traversal where the node is visited before its children are visited is called a breadth-first traversal; a walk where the children are visited prior to the parent is called a depth-first traversal.

Decision tree Decision tree is a classifier in the form of a tree structure, where each node is either:

a leaf node—indicates the value of the target attribute (class) of examples; or a decision node—specifies some test to be carried out on a single attribute-value, with one branch and sub-tree for each possible outcome of the test.

Expression tree A binary tree storing or representing an arithmetic expression is called as an expression tree. The leaves of an expression tree are operands. Operands could be variables or constants. The branch nodes (internal nodes) represent the operators.

Inorder traversal In this traversal, the left subtree is visited first in inorder, then the root, and finally the right subtree in inorder.

Non-linear data structures Non-linear data structures are used to represent the data containing hierarchical or network relationship between the elements. Trees and graphs are examples of non-linear data structure.

Pre-order traversal In this traversal, the root is visited first, then the left subtree in preorder, and finally the right subtree in preorder.

Threaded binary tree A.J. Perlis and C. Thornton have suggested to replace all the null links in binary tree by pointers, called threads. A tree with thread is called as threaded binary tree.

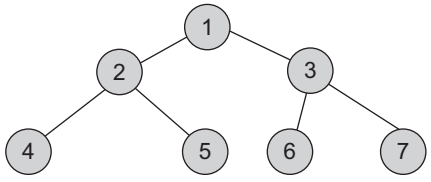
Tree traversal Traversal of tree means stepping through the nodes of a tree by means of the connections between parents and children; it is also called walking the tree, and the action is called the walk of the tree.

Tree Tree, a non-linear data structure, is a mean to maintain and manipulate data in many applications. Non-linear data structures are capable of expressing more complex relationship than linear data structure. A class of graphs that are acyclic are termed as trees. Trees are useful in describing any structure that involves hierarchy.

EXERCISES

Multiple choice questions

1. Consider the following tree:



If the postorder traversal gives ($ab - cd +$), then the label of the nodes 1, 2, 3, 4, 5, 6 will be
   (a) $+, -, \times, a, b, c, d$
   (b) $a, -, b, +, c, \times, d$
   (c) $a, b, c, d, -, \times, +$
   (d) $-, a, b, +, \times, c, d$

2. A list of integers is read one at a time, and a BST is constructed. Next, the tree is traversed and the integers are printed. Which traversal would print the result in the original order of the input?
   (a) Preorder
   (b) Postorder
   (c) Inorder
   (d) None of the above

3. A binary tree T has n leaf nodes. The number of nodes of degree 2 in T is
   (a) $\log_2 n$
   (b) $n - 1$
   (c) n
   (d) 2^n

4. Which is the most efficient tree for accessing data from a database?
   (a) BST
   (b) B-tree
   (c) OBST
   (d) AVL tree

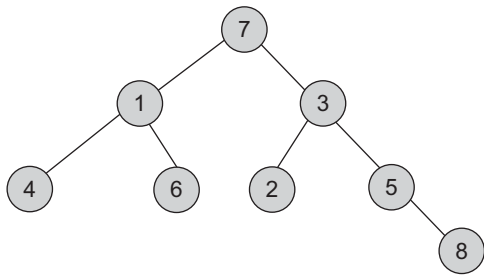
5. A binary tree where every non-leaf node has non-empty left and right subtrees is called a strictly binary tree. Such a tree with 10 leaves
   (a) cannot have more than 19 nodes.

(b) has exactly 19 nodes.
(c) has exactly 17 nodes.
(d) cannot have more than 17 nodes.

6. The depth of a complete binary tree with n nodes is
   (a) $\log_2 (n + 1) - 1$
   (b) $\log_2 n$
   (c) $\log_2 (n - 1) + 1$
   (d) $\log_2 n + 1$
7. Which of the following traversal techniques lists the nodes of a BST in ascending order?
   (a) Postorder
   (b) Inorder
   (c) Preorder
   (d) All of a, b, c
8. A binary tree has a height of 5. What is the minimum number of nodes it can have?
   (a) 31
   (b) 15
   (c) 5
   (d) 1
9. A binary tree is generated by inserting an inorder as 50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24. The number of nodes in the left and right subtree, respectively is given by
   (a) (4, 7)
   (b) (7, 4)
   (c) (8, 3)
   (d) (3, 8)
10. A BST contains the values 1, 2, 3, 4, 5, 6, 7, 8. The tree is traversed in preorder and the values are printed. The valid output is
    (a) 53124786
    (b) 53126487
    (c) 53241678
    (d) 53124768
11. In _____ traversal, the right subtree is processed last.
    (a) a preorder
    (b) an inorder
    (c) a postorder
    (d) (a) or (b)

Review questions

1. Consider the binary tree in the following figure.



   (a) What structure is represented by the binary tree?
   (b) Give the different steps for deleting the node with key 5 so that the structure is preserved.
   (c) Outline a procedure in pseudo code to delete an arbitrary node from such a binary tree with n nodes that preserves the structure. What is the worst case time complexity of your procedure?
2. Prove by the principal of mathematical induction that for any binary tree where every non-leaf node has 2 descendants, the number of leaves in the tree is one more than the number of non-leaf nodes.
3. A 3-ary tree is a tree where every internal node has exactly 3 children. Use induction to prove that the number of leaves in a 3-ary tree with n internal nodes is $2(n - 1) + 3$.
4. A rooted tree with 12 nodes has its numbers from 1 to 12 in preorder. When the tree is traversed in postorder, the nodes are visited in following order: 3, 5, 4, 2, 7, 8, 6, 10, 11, 12, 9, 1. Reconstruct the original tree from this information, that is, find the parent of each node. Show the tree diagrammatically.
5. What is the number of binary trees with 3 nodes which when traversed in postorder give the sequence A, B, C? Draw all these binary trees.
6. A size-balanced binary tree is a binary tree where for every node, the difference between the

number of nodes in the left and right subtree is utmost 1. The distance of a node from the root is the length of the path from the root to the node. The height of a binary tree is the maximum distance of a leaf node from the root.

(a) Prove by induction on h that a size-balance binary tree of height h contains at least $2h$ nodes.

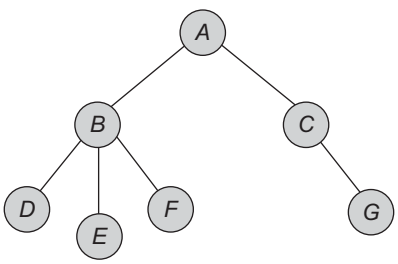
(b) In a size-balanced tree of height $h \leq 1$, how many nodes are at a distance $h - 1$ from the root?

7. Let $\boldsymbol{A}$ be an $n \times n$ matrix such that the elements in each row and each column are arranged in ascending order. Draw a decision tree that finds first, second, and third smallest elements in minimum number of comparisons.
8. In a binary tree, a full node is defined to be a node with 2 children. Use induction on the height of a binary tree to prove that the number of full nodes plus one is equal to the number of leaves.
9. (a) Draw a BST (initially empty) that results from inserting the records with the keys

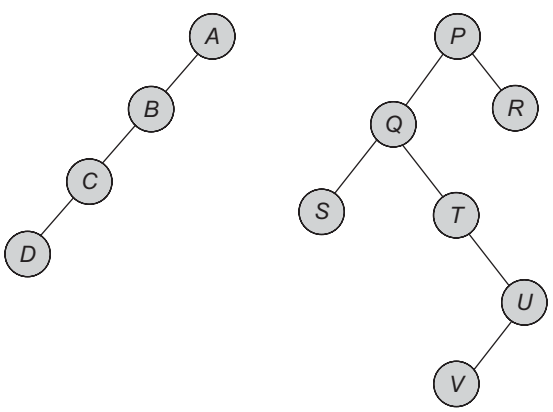
EASY QUESTION

(b) Delete the key Q from the constructed BST.

10. Write a recursive function in C++ that creates a mirror image of a binary tree.
11. What is a BST? Write a recursive C++ function to search for an element in a given BST. Write a non-recursive version of the same.
12. Write a non-recursive C++ function to traverse a binary tree containing integers in preorder.
13. Write a C++ function for insertion of a node into a BST.
14. Write C++ function that traverses a TBT in inorder.
15. Represent a binary tree using pointers and write a function to traverse and point nodes of a tree level-by-level.
16. Represent a binary tree using pointers and write a function to traverse a given tree in inorder.
17. Given the following inorder and postorder sequences of nodes of binary tree, draw the corresponding binary tree. Show the steps.
    (a) Inorder : 1 3 5 6 4 2 7
    (b) Postorder : 6 5 4 3 7 2 1
18. From the given traversals, construct the binary tree.
    (a) Inorder: D B F E A G C L J H K
    (b) Postorder: D F E B G L J K H C A
19. Write a pseudocode C++ for non-recursive postorder and inorder traversal for binary tree.
20. List down the steps to convert a general tree to a binary tree. Convert the following general tree to a binary tree.



21. Explain the array representation of binary trees using the following figures and state and explain the limitations of this representation.



(a) (b)

22. Write a pseudocode for deleting a node from a BST. Simulate your algorithm with a BST of 10 nodes and show the deletion process. Especially, show the deletion of the interior nodes and not just the leaf nodes.

23. Write a C ++ function to find the following:
    (a) Height of a given binary tree
    (b) Width (breadth) of a binary tree

Answers to multiple choice questions

1. (a) The postorder traversal yields 4, 5, 2, 6, 7, 3, 1. Comparing with a, b, $-$, c, d, $\times$, $+$, we get the labels of nodes 1, 2, 3, 4, 5, 6, 7 as $+$, $-$, $\times$, a, b, c, d, respectively.
2. (d) 3. (b) 4. (c) 5. (b) A regular (strictly) binary tree with n leaves must have $(2n - 1)$ nodes. 6. (a) 7. (b) 8. (c) 9. (b) 10. (d) 11. (d)