16 CHAPTER Trees

16.1 Introduction

Trees form one of the most widely used subclass of graphs. This is due to the fact that ma of the applications of graph theory, directly or indirectly, involve trees. Tree occurs in situatio where many elements are to be organized into some short of hierarchy. In computer science, tre are useful in organizing and storing data in a database.

In this chapter we introduce the basic terminology of tree. We look at subtrees of trees e rooted trees and binary trees and also many applications of trees.

16.2. Trees and their Properties

A tree is a connected acyclic graph *i.e.* a connected graph having no cycle. Its edges are call **branches**. Fig. 16.1. are examples of trees with atmost five vertices. Fig. 16.2. (*a*) and (*b*) are n trees, since they have cycles.

A tree with only one vertex is called a **trivial tree** otherwise T is a **nontrivial tree**.

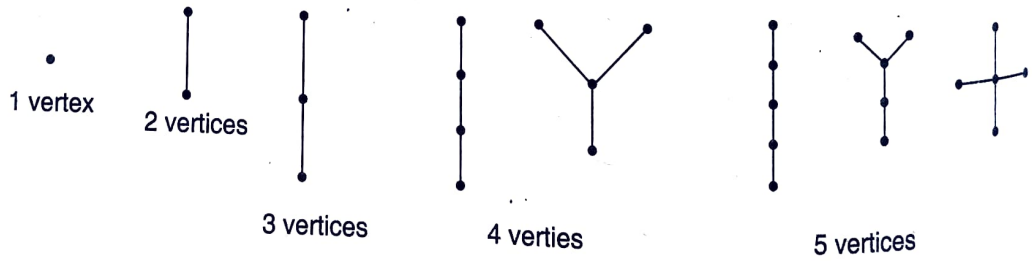


Fig. 16.1

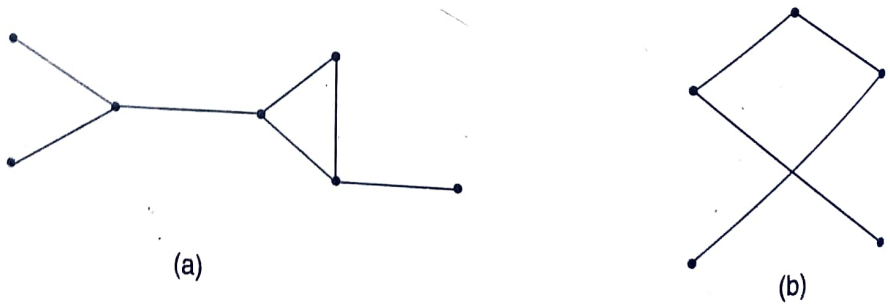


Fig. 16.2

A collection of trees is called a Forest.

Characterisations

Trees have many equivalent characterisations, any of which could be taken as the definiti Such characterisation are useful because we need only verify that a graph satisfies any one of th to prove that it is a tree, after which we can use all other properties. A few simple and import theorems on the general properties of trees are given below.

Theorem 16.1. There is one and only one path between every pair of vertices in a tree, T.

Proof. Since T is a connected graph, there must exist at least one path between every pair of vertices in T. Let there are two distinct paths between two vertices u and v of T but union of these two paths will contain a cycle and then T cannot be a tree.

Conversely

Theorem 16.2. If in a graph G there is one and only one path between every pair of vertices, G is a tree.

Proof. Since there exists a path between every pair of vertices then G is connected. A cycle in a graph (with two or more vertices) implies that there is at least one pair of vertices u, v such that there are two distinct paths between u and v. Since G has one and only one path between every pair of vertices, G can have no cycles, Therefore, G is a tree.

Theorem 16.3. A tree T with n vertices has $n - 1$ edges.

Proof. The theorem is proved by induction on n, the number of vertices of T.

Basis of Induction: When $n = 1$ then T has only one vertex. Since it has no cycles, T cannot have any edge *i.e.* it has $e = 0 = n - 1$.

Inductive step: Suppose the theorem is true for $n = k \geq 2$ where k is some positive integer. We use this to show that the result is true for $n = k + 1$. Let T be a tree with $k + 1$ vertices and let uv be edge of T. Then if we remove the edge uv from T we obtain the graph $T - uv$. Then the graph is disconnected since $T - uv$ contains no (u, v) path. If there were a path, say $u, v_1, v_2, \ldots, v$, from u to v then when we added back the edge uv there would be a cycle $u, v_1, v_2 \ldots, v, u$ in T.

Thus, $T - uv$ is disconnected. The removal of an edge from a graph can disconnect the graph into at most two components. So $T - uv$ has two components, say, T_1 and T_2. Since there were no cycles in T to begin with, both components are connected and are without cycles. Thus, T_1 and T_2 are trees and each has fewer than n vertices. This means that we can apply the induction hypothesis to T_1 and T_2, to give

$$e(T_1) = v(T_1) - 1$$
$$e(T_2) = v(T_2) - 1$$

But the construction of T_1 and T_2 by removal of a single edge from T gives that

$$e(T) = e(T_1) + e(T_2) + 1$$

and that

$$v(T) = v(T_1) + v(T_2)$$

it follows that

$$e(T) = v(T_1) - 1 + v(T_2) - 1 + 1 = v(T) - 1 = k + 1 - 1 = k$$

Thus T has k edges, as required.

Hence by the principle of mathematical induction the theorem is proved.

Theorem 16.4. For any positive integer n, if G is a connected graph with n vertices and $n - 1$ edges, then G is a tree.

Proof. Let n be a positive integer and suppose G is a particular but arbitrarily chosen graph that is connected and has n vertices and $n - 1$ edges. We know that a tree is a connected graph without cycles. We have proved in previous theorem that a tree has $n - 1$ edges. We have to prove the converse that if G has no cycles and n-1 edges, then G is connected. We decompose G into k components, $G_1, G_2, \ldots G_k$. Each component is connected and it has no cycles since G has no cycles. Hence, each C_k is a tree. Now, $e_i = n_i - 1$ and $\sum_{i=1}^{k} e_i = \sum_{i=1}^{k} (n_i - 1) = n - k$ or $e = n - k$. Then it follows that $k = 1$ or G has only one component. Hence, G is a tree.

Theorem 16.5 A graph is a tree if and only if it is minimally connected

Proof. Let G be a graph with n vertices and we assume that G is a tree with n vertices. We know that a tree with n number of vertices has $(n-1)$ number of edges. If one edge is deleted from G, then it has $(n-2)$ edges and G becomes disconnected. Hence G is a minimally connected graph.

Conversely, let G be a minimally connected graph with n number of vertices. The number of edges of $G \geq n-1$ as G is a connected graph. If possible, let G be not a tree. Then G contains a circuit and G becomes still connected if one edge of this circuit is removed from G. This contradicts our hypothesis that G is a minimally connected graph. Hence G is a tree.

The results of the preceding theorems can be summarised by saying that the following are five different but equivalent definitions of tree. A graph with n vertices is called a tree if

1. G is connected and has no cycles (acyclic)
2. G is connected and has n-1 edges
3. G is a acyclic and has n-1 edges
4. There is exactly one path between every pair of vertices in G
5. G is a minimally connected graph.

Rooted Trees

A rooted tree is a tree in which a particular vertex is distinguished from the others and is called the **root.** In contrast to natural trees, which have their roots at the bottom, in graph theory rooted trees are typically drawn with their roots at the top. First, we place the root at the top. Under the root and on the same level, we place the vertices that can be reached from the root on a simple path of length 1. Under each of these vertices and on the same level, we place vertices that can be reached from the root on a simple path of length 2. We continue in this way until the entire tree is drawn. We give definitions of some terms related to it.

1. The **level** of a vertex is the number of edges along the unique path between it and the root. The level of the root is defined as 0. The vertices immediately under the root are said to be in level 1 and so on.
2. The **height** of a rooted tree is the maximum level to any vertex of the tree. The **depth** of a vertex v in a tree is the length of the path from the root to v.
3. Given any internal vertex v of a rooted tree, the **children** of v are all those vertices that are adjacent to v and are one level further away from the root than v. If w is a child of v, the v is called the **parent** of w, and two vertices that are both children of the same parent are called **siblings.**
4. If the vertex u has no children, then u is called a **leaf** (pendant or a **terminal vertex**). A non-pendant vertex in a is called an **internal vertex.**
5. The **descendants** of the vertex u is the set consisting of all the children of u together with the descents of those children. Given vertices v and w, if v lies on the unique path between w and the root, then v is an **ancestor** of w and w is a descendant of v.

These terms are illustrated in Fig. 16.3.

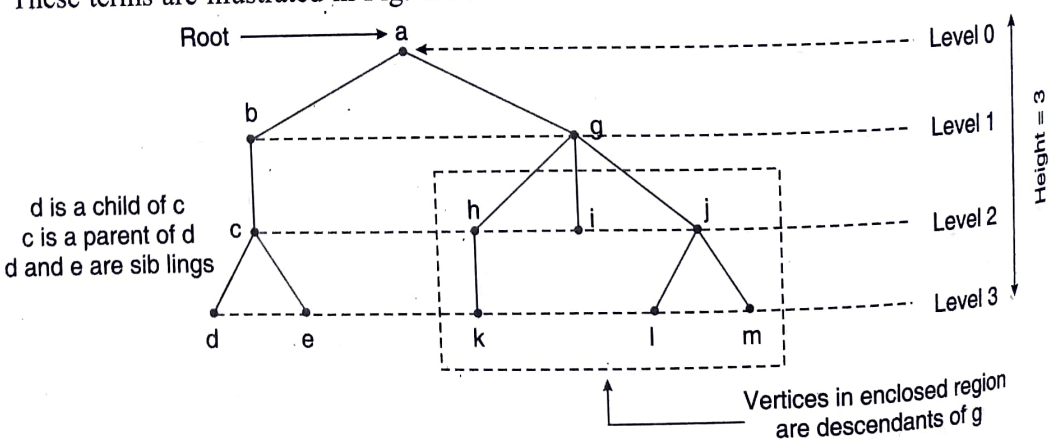


Fig. 16.3

Example 1. Consider the rooted tree in Fig. 16.4.

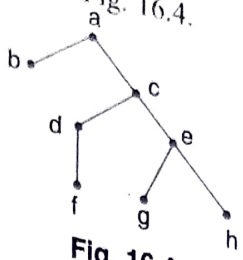


Fig. 16.4

(a) What is the root of T?

(b) Find the leaves and the internal vertices of T.

(c) What are the levels of c and e.

(d) Find the children of c and e.

(e) Find the descendants of the vertices a and c.

Solution. (*a*) Vertex a is distinguished as the only vertex located at the top of the tree. Therefore a is the root.

(*b*) The leaves are those vertices that have no children. These are b, f, g and h. The internal vertices are c, d and e.

(*c*) The levels of c and e are 1 and 2 respectively.

(*d*) The children of c are d and e and of e are g and h.

(*e*) The descendants of a are b, c, d, e, f, g, h. The descendants of c are d, e, f, g, h.

Definition. A rooted tree is an ***m*-ary** if every internal vertex has at most m children. A m-ary tree is a **full *m*-ary** tree if every internal vertex has exactly m children. In particular, the 2-ary tree is called **binary tree**.

The relationship between i, the number of internal vertices and l, the number of leaves of a full m-ary tree can be proved by using the following theorem.

Theorem 16.6. A full m-ary tree with i internal vertex has $n = mi + 1$ vertices.

Proof. Since the tree is a full m-ary, each internal vertex has m children and the number of internal vertex is i, the total number of vertex except the root is mi.

Therefore, the tree has $n = mi + 1$ vertices.

Since l is the number of leaves, we have $n = l + i$.Using the two equalities $n = mi + 1$ and $n = l + i$, the following results can easily be deduced.

A full m-ary tree with

(*i*) n vertices has $i = (n\text{-}1) / m$ internal vertices and $l = [(m - 1) n + 1] / m$ leaves

(*ii*) i internal vertices has $n = mi + 1$ vertices and $l = (m - 1) i + 1$ leaves

(*iii*) l leaves has $n = (ml - 1) / (m - 1)$ vertices and $i = (l - 1) / (m - 1)$ internal vertices.

Theorem 16.7. There are at most m^h leaves in an m-ary tree of height h.

Proof. We prove the theorem by mathematical induction.

Basis of Induction: For $h = 1$, The tree consists of a root with no more than m children, each of which is a leaf. Hence there are no more than $m^1 = m$ leaves in an m-ary of height 1.

Induction Hypothesis: We assume that the result is true for all m-ary trees of heights less than h.

Induction Step: Let T be an m-ary tree of height h. The leaves of T are the leaves of subtrees of T obtained by deleting the edges from the root to each of the vertices of level 1. Each of these

subtrees has height less than or equal $h-1$. So by the inductive hypothesis, each of these rooted trees has at most m^{h-1} leaves. Since there are at most m such subtrees, each with a maximum of m^{h-1} leaves, there are at most $m.m^{h-1} = m^h$.

This proves the theorem.

Example 2. *Every notrivial tree T has at least two vertices of degree 1.*

Solution. Let n = the number of vertices of T ($n \geq 2$) and m = the number of vertices of degree 1.

Let $v_1, v_2, v_3, \ldots v_m$ denote the m vertices of degree 1. Then each of the remaining $n-m$ vertices $v_{m+1}, v_{m+2}, \ldots v_n$ has at least degree 2.

Thus, $\deg(v_i) = 1$ for $i = 1, 2, \ldots, m$.

≥ 2 for $i = m+1, m+2, \ldots, n$.

$$\sum_{i=1}^{n} \deg(v_i) = \sum_{i=1}^{m} \deg(v_i) + \sum_{i=m+1}^{n} \deg(v_i) = m + \sum_{i=m+1}^{n} \deg(v_i)$$

$$\geq m + 2(n-m) = 2n - m$$

Again $$\sum_{i=1}^{n} \deg(v_i) = 2e = 2(n-1) = 2n-2$$

Hence $2n - 2 \geq 2n - m \Rightarrow m \geq 2$.

This proves that T contains at least two vertices of degree 1.

Example 3. *In a complete n-ary tree with i internal vertices, the number of leaf vertex p is given by* $p = (n-1)(x-1)/n$.

Solution. Since the tree is complete n-ary having i internal vertices, total number of vertices is $x = n.i + 1$ so that $i = (x-1)/n$. ...(1)

Again p is the number of leaves in the trees so $x = i + p + 1$. ...(2)

Eliminating i from (1) and (2), we get

$P = (n-1)(x-1)/n$.

Example 4. *A tree has two vertices of degree 2, one vertex of degree 3 and three vertices of degree 4. How many vertices of degree 1 does it have?*

Solution. Let x be the required number. Now, total number of vertices $= 2+1+3+x = 6+x$

Hence the number of edges is $6 + x - 1 = 5 + x$ [In a tree $|E| = |V| - 1$]

The total degree of the tree $= 2 \times 2 + 1 \times 3 + 3 \times 4 + 1 \times x = 19 + x$

So, the number of edges is $\dfrac{19+x}{2}$ [$2e = \Sigma \deg(v_i)$]

Now, $$\frac{19+x}{2} = 5 + x$$

$$19 + x = 10 + 2x$$

or $$x = 9$$

Thus, there are 9 vertices of degree one in the tree.

16.3. Spanning Tree

A subgraph T of a connected graph $G(V, E)$ is called a spanning tree if (*i*) T is a tree and (*ii*) T includes every vertex of G *i.e.* $V(T) = V(G)$. If $|V| = n$ and $|E| = m$, then the spanning tree of G must have n vertices and hence $n-1$ edges. We must remove $m - (n-1)$ edges from G to obtain a spanning tree. In removing these edges one must ensure that the resulting graph remain connected and further there is no circuit in it.

Example 5. Find all spanning trees of the graph G shown in Fig. 16.5

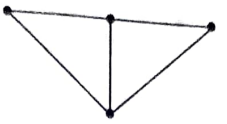


Fig. 16.5

Solution. The graph G has four vertices and hence each spanning tree must have $4 - 1 = 3$ edges. Thus each tree can be obtained by deleting two of the five edges of G. This can be done in 10 $\left(\binom{5}{2}\right)$ ways, except that two of the ways lead to disconnected graphs. Thus there are eight spanning trees as shown in Fig. 16.6.

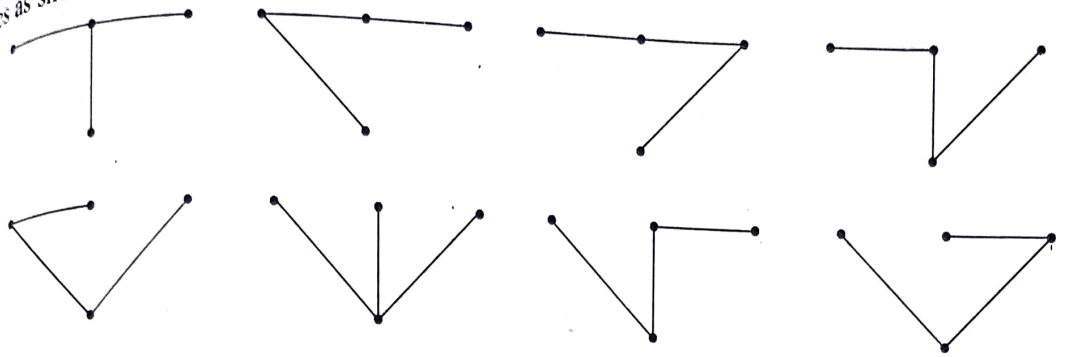


Fig. 16.6

Theorem 16.8. A simple graph G has a spanning tree if and only if G is connected.

Proof. First, suppose that a simple graph G has a spanning tree T. T contains every vertex of G. Let a and b be vertices of G. Since a and b are also vertices of T and T is a tree, there is a path P between a and b. Since T is subgraph, P also serves as path between a and b in G. Hence G is connected.

Conversely, suppose that G is connected. If G is not a tree, it must contain a simple circuit. Remove an edge from one of these simple circuits. The resulting subgraph has one fewer edge but still contains all the vertices of G and is connected. If this subgraph is not a tree, it has a simple circuit ; so as before, remove an edge that is in a simple circuit. Repeat this process until no simple circuit remain. This is possible because there are only a finite number of edges in the graph, the process terminates when no simple circuits remain. Thus we eventually produce an acyclic subgraph T which is a tree. The tree is a spanning tree since it contains every vertex of G

Example 6(*i*). Find all spanning tree of the graph G shown in Fig. 16.7

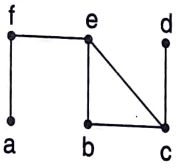


Fig. 16.7

Solution. The graph G is connected. It has 6 edges and 6 vertices and hence each spanning tree must have $6 - 1 = 5$ edges. So $6 - 5 = 1$ edges has to be deleted from G. The graph G has one cycle $c\ b\ e\ c$, removal of any edge of the cycle gives a tree. There are three trees which contain all the vertices of G and hence spanning trees.

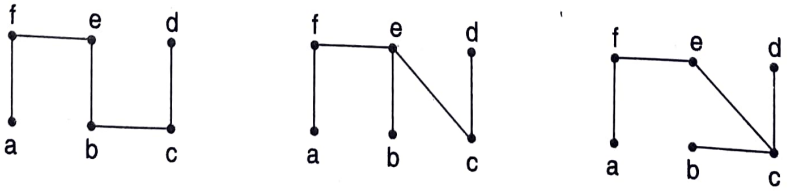
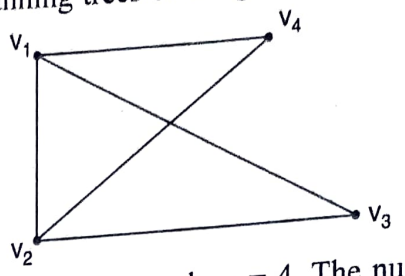


Fig. 16.8

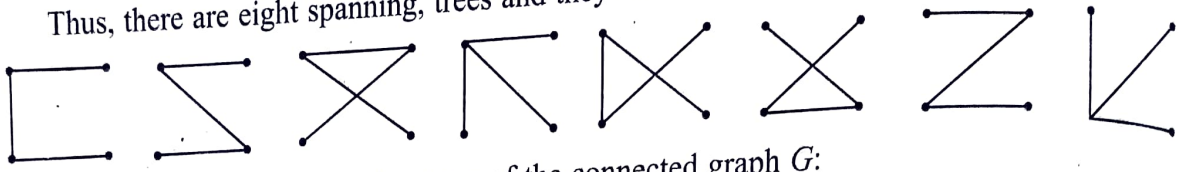
Note that a spanning tree of a graph need not be unique.

Example 6(*ii*). Find all the spanning trees of the graph shown below.

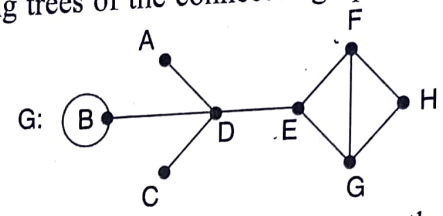


Solution. The number of vertices in the graph, $n = 4$. The number of edges, $m = 5$. So, the number of edges to be detected to get the spanning trees $= m - n + 1 = 5 - 4 + 1 = 2$.

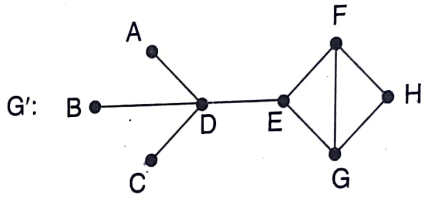
Thus, there are eight spanning, trees and they are



Example 7. Find all spanning trees of the connected graph G:



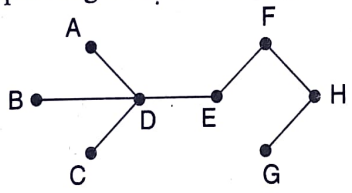
Solution. Since the vertex B contains self-loop, we remove the self-loop from the vertex B, and G becomes



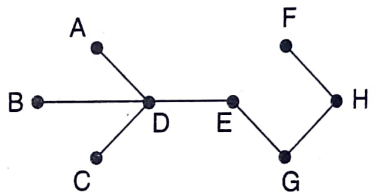
The graph is connected and it has 9 edges and 8 vertices so $9 - (8 - 1) = 2$ edges has to be deleted from the graph to get a spanning tree which is connected and does not contain cycle.

The spanning trees of the graph G' are given below:

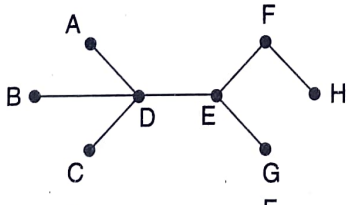
(*i*)



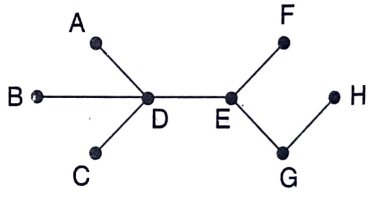
(*ii*)



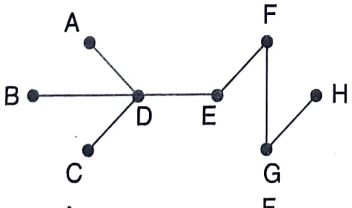
(*iii*)



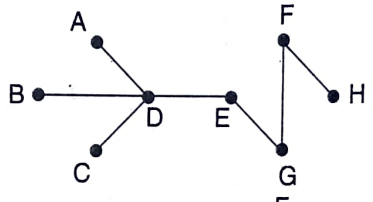
(*iv*)



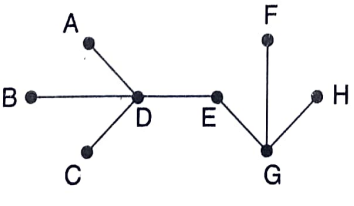
(*v*)



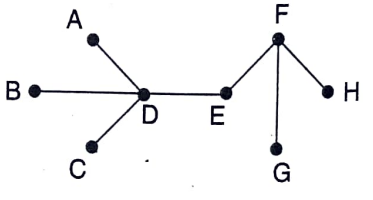
(*vi*)



(*vii*)



(*viii*)

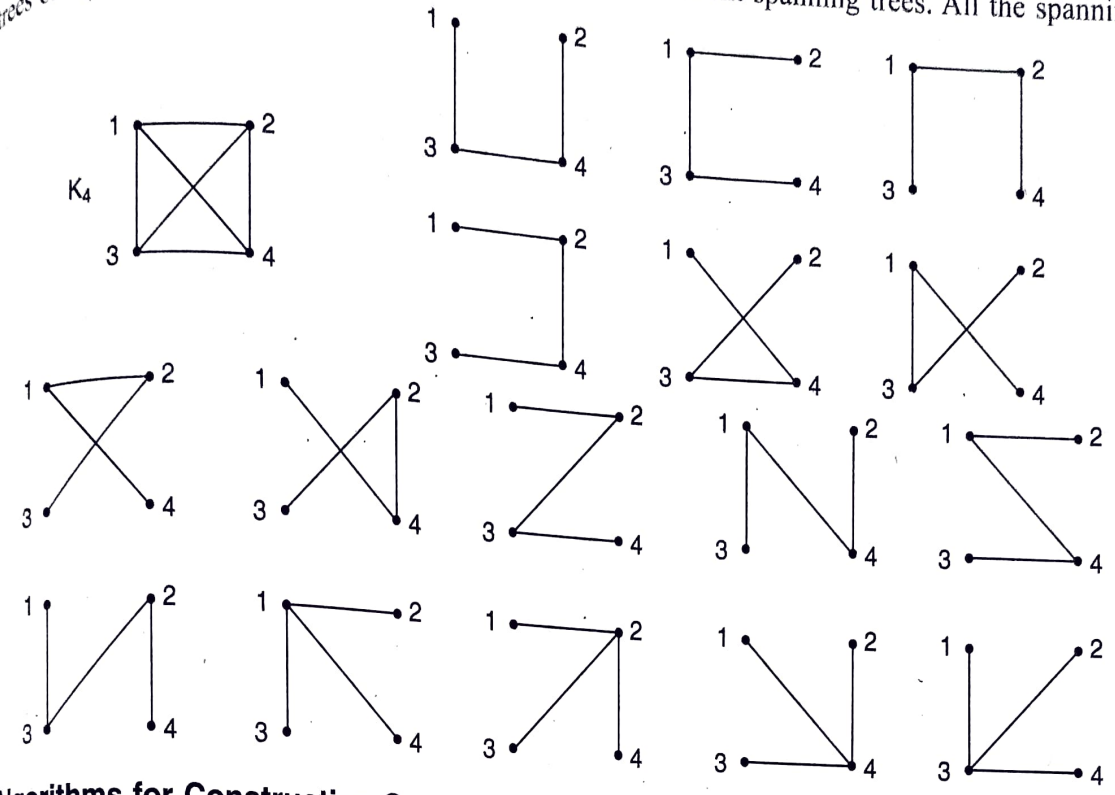


The number of different spanning trees on the complete graph K_n can be found form Calay's theorem which is given below without proof.

Caylay's Theorem 16.9. The complete graph K_n has n^{n-2} different spanning trees.

Example 8. Give all the spanning trees of K_4.

Solution. Here $n = 4$, so there will be $4^{4-2} = 16$ different spanning trees. All the spanning trees of K_4 are shown below



Algorithms for Constructing Spanning Trees

Instead of constructing spanning trees by removing edges, spanning tree can be built up by successively adding edges. Two algorithms based on this principle for finding a spannig tree are Breath-first Search (BFS) and Depth-first Search (DFS).

BFS Algorithm

The idea of BFS is to visit all vertices on a given level before going into the next level until all are visited.

Procedure.

(*i*) Arbitrarily choose a vertex and designate it as the root. Then add all edges incident to this vertex, such that the addition of edges does not produce any cycle.

(*ii*) The new vertices added at this stage become the vertices at level 1 in the spanning tree, arbitrarily order them.

(*iii*) Next, for each vertex at level 1, visited in order, add each edge incident to this vertex to the tree as long as it does not produce any cycle.

(*iv*) Arbitrarily order the children of each vertex at level 1. This produces the vertices at level 2 in the tree.

(*v*) Continue the same procedure until all the vertices in the tree have been added.

(*vi*) The procedure ends, since there are only a finite number of edges in the graph.

(*vii*) A spanning tree is produced since we have produced a tree without cycle containing every vertex of the graph.

Example 9. Use BFS algorithm to find a spanning tree of graph G of Fig. 16.9.

Solution. (*i*) Choose the vertex a to be the root.

(*ii*) Add edges incident with all vertices adjacent to a, so that edges $\{a, b\}$, $\{a, c\}$ are added. The two vertices b and c are in level 1 in the tree.

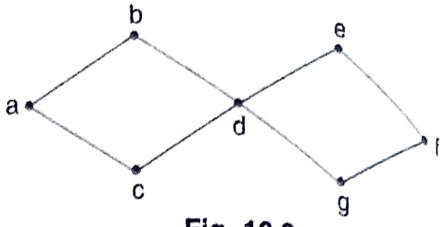


Fig. 16.9

(*iii*) Add edges from these vertices at level 1 to adjacent vertices not already in the tree. Hence the edge $\{c, d\}$ is added. The vertex d is in level 2. (why $\{b, d\}$ is not joined ?)

(*iv*) Add edge from d in level 2 to adjacent vertices not already in the tree. The edge $\{d, e\}$ and $\{d, g\}$ are added. Hence e and g are in level 3.

(*v*) Add edge from e at level 3 to adjacent vertices not already in the tree and hence $\{e, f\}$ is added.

The steps of Breath fast procedure are shown in Fig. 16.10

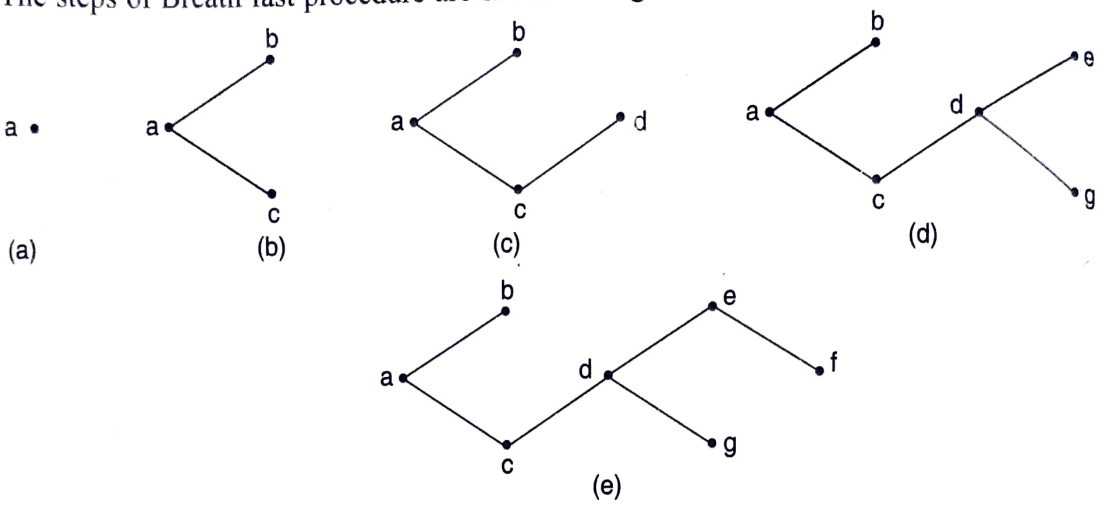


Fig. 16.10

Hence Fig.16.10. (*e*) is the required spanning tree.

DFS Algorithm

An alternative to Breath–first search is Depth-first search which proceeds to successive levels in a tree at the earliest possible opportunity. DFS is also called back **tracking**.

Procedure.

(*i*) Arbitrarily choose a vertex from the vertices of the graph and designate it as the root.

(*ii*) Form a path star ting at this vertex by successively adding edges as long as possible where each new edge is incident with the last vertex in the path without producing any cycle.

(*iii*) If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree.

Otherwise, move back to the next to last vertex in the path, and, if possible, form a new path starting at this vertex passing through vertices that were not already visited.

(*iv*) If this cannot be done, move back another vertex in the path, that is two vertices back in the path, and repeat.

(*v*) Repeat this procedure, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new paths that are as long as possible until no more edges can be added.

(*vi*) This process ends since the graph has a finite number of edges and is connected. A spanning tree is produced.

Note (*i*) For DFS and BFS, discard all parallel edges and self loops from the given connected graph.

(*ii*) If the given connected graph G is a directed graph then we construct the corresponding undirected graph.

Example 10. Find a spanning tree of the graph of Fig. 16.11 using Depth–first search algorithm.

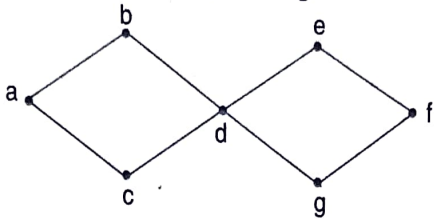


Fig. 16.11

Solution. Choose the vertex a. Form a path by successively adding edges incident with vertices not already in the path as long as possible. This produces the path *a-c-d-e-f-g*.

Now back track to *f*. There is no path beginning at *f* containing vertices not already visited. Similarly, after back track at *e*, there is no path. So move back track at *d* and form the path $d - b$. This produces the required spanning tree which is shown in Fig. 16.12.

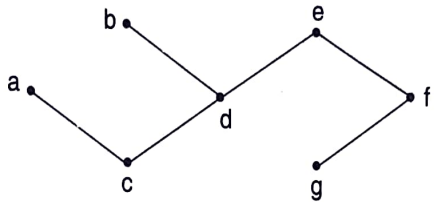


Fig. 16.12

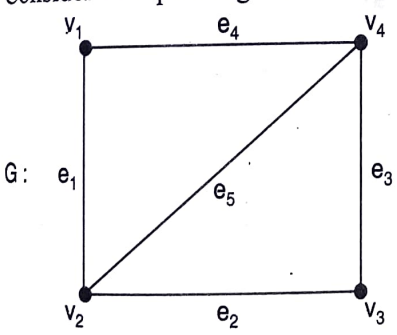
Note: The basic idea of BFS is to visit all vertices sequentially on a given level before it goes on the next level.

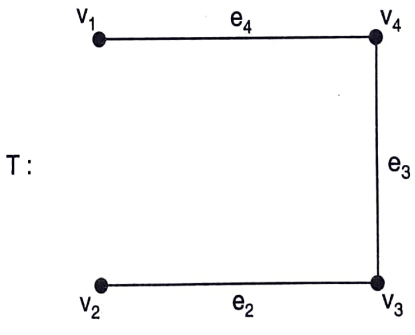
The DFS proceeds successively to higher levels at the first opportunity.

Fundamental Circuits

Let T be a spanning tree of a graph G. Then the edges of G that are in T are called **branches** of G. The edge of G that is not in T is called a **chord** of G with respect to T. A circuit formed by adding a chord e to a spanning tree T of a graph is called a Fundamental circuit of G with respect to spanning tree T relative to chord e. The cut set containing exactly one branch of T is called fundamental cut set of G to w *r.t.T*.

Consider the spanning tree T of the graph G as shown below:





The branches of G are e_2, e_3 and e_4.

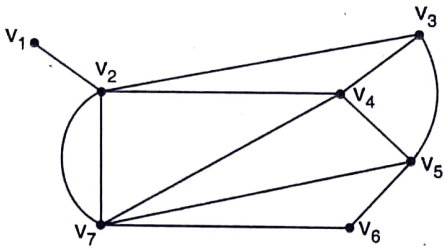
The chords of G are e_1 and e_5.

If the chord e_1 is added to the spanning tree, then one circuit $v_1 - v_4 - v_3 - v_2 - v_1$ is formed and is known as fundamental circuit. If the chord e_5 is added, then the circuit $v_2 - v_3 - v_4 - v_2$ is another fundamental circuit.

Notes

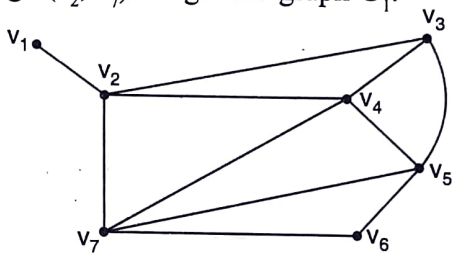
1. Fundamental circuit is defined with respect to a spanning tree.
2. Fundamental circuit with respect to a spanning tree in a graph is not unique.
3. A given circuit may be fundamental circuit with respect to one spanning tree but not so with respect to other spanning tree in the same graph.
4. If G is a connected simple graph with n vertices and e edges, it has $r = e - (n - 1)$ chords with respect to any spanning tree T, so it has r fundamental circuits with respect to T.
5. For every branch there is corresponding cutset since removal of any branch from a spanning tree breaks the spanning tree into two trees.

Example 11. Find fundamental circuits for the graphs shown below.



Solution: We first use *BFS* algorithm to find the spanning tree. First delete all loops and parallel edges. Deleting one parallel edge (v_2, v_7) we get the graph G_1.

G_1 :



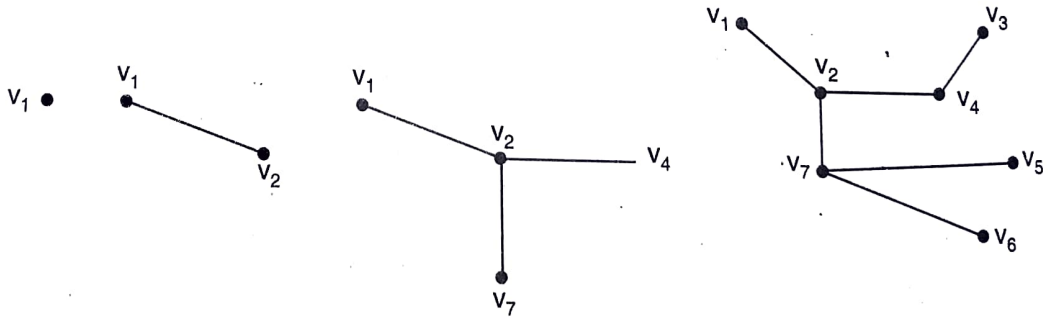
(*i*) Choose the vertex v_1 to be the root.

(*ii*) Add edge incident with all vertices to v_1, so that edge (v_1, v_2) is added.

(*iii*) Add edges from this vertex v_2 to adjacent vertices not already in the tree. Hence (v_2, v_4) and (v_2, v_7) are added.

(*iv*) Add edges from v_4 and v_7 to adjacent vertices not already in tree. Hence (v_7, v_6), (v_7, v_5) and (v_4, v_3) are added and we get a spanning tree.

The steps of *BFS* procedure are shown on the next page.



After converting the given graph in simple graph, we have $e = 11$ and $n = 7$. So there are $r = 11 - 7 + 1 = 5$ fundamental circuits and these are $v_2 - v_3 - v_4 - v_2$ relative to edge (v_2, v_3) $v_4 - v_7 - v_2 - v_4$ relative to edge (v_4, v_7), $v_4 - v_5 - v_7 - v_2 - v_4$ relative to edge (v_4, v_5), $v_5 - v_6 - v_7 - v_5$ relative to edge (v_5, v_6) and $v_3 - v_4 - v_2 - v_7 - v_5 - v_3$ relative to edge (v_3, v_5).

Rank and Nullity

If n be the number of vertices, e the number of edges and $k(n \geq k, k = 1$ for connected graph) the number of components of a graph G then rank and nullity is defined as

$$\text{Rank } r = n - k$$

$$\text{Nullity} = e - n + k$$

In general, for any graph of n vertices and e edges, the number of edges to be removed to make it acyclic is called nullity of the graph and equal to the number of chords with respect to any spanning tree of G.

The complement of nullity *i.e.*, the number of understroyed edges is called rank which gives number of branches of G with respect to any spanning tree.

Weighted Graph

A weighted graph is a graph G in which each edge e has been assigned a non-negative number $w(e)$, called the weight (or length) of e. Fig. 16.13 shows a weighted graph. The weight (or length) of a path in such a weighted graph G is defined to be the sum of the weights of the edges in the path. Many optimisation problems amount to finding, in a suitable weighted graph, a certain type of subgraph with minimum (or maximum) weight.

Minimal Spanning Trees

Let G be a connected weightd graph. The weight of a spanning tree of G is the sum of the weights of the edges. A minimal spanning tree of G is a spanning tree of G with minimum weight. The weighted graph G of Fig. 16.13. shows six cities and the costs of laying railway links between certain pairs of cities. We want to set up railway links between the cities at minimum costs. The solution can be represented by a subgraph. This subgraph must be a spanning tree since it covers all the vertices (so that each city is in the road system), it must be connected (so that any city can be reached from any other), it must have unique simple path between each pair of vertices. Thus what is needed is a spanning tree the sum of whose weights is minimum *i.e.*, a minimal spanning tree.

Algorithm for Minimal Spanning Trees

There are several methods available for actually finding a minimal spanning tree in a given graph. Two algorithms due to Kruskal and Prim of finding a minimal spanning tree for a connected weighted graph where no weight is negative are presented below. These algorithms are example of **greedy algorithms.** A greedy algorithm is a procedure that makes an optimal choice at each of its steps without regard to previous choices.

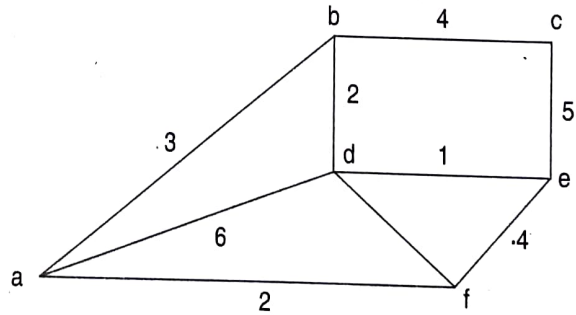


Fig. 16.13

Kruskal's Algorithm

This algorithm provides an acyclic subgraph T of a connected weighted graph G which is a minimal spanning tree of G. The algorithm involves the following steps:

Input : A connected weighted graph G.

Output : A minimal spanning tree T.

Step 1. List all the edges (which do not form a loop) of G in non-decreasing order of their weights.

Step 2. Select an edge of minimum weight (If more than one edge of minimum weight, arbitrarily choose one of them). This the first edge of T.

Step 3. At each stage, select an edge of minmum weight from all the remaining edges of G if it does not form a circuit with the previously selected edges in T. Include the edge in T.

Step 4. Repeat step 3 until $n - 1$ edges have been selected, when n is the number of vertices in G.

The following examples illustrate these steps.

Example 12. Show how Kruskal's algorithm find a minimal spanning tree for the graph of Fig. 16.14.

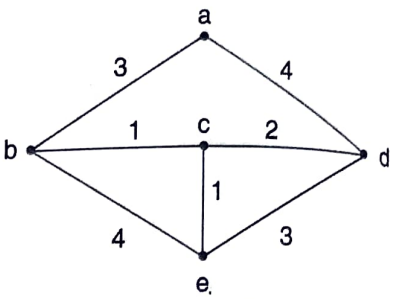


Fig. 16.14

Solution:

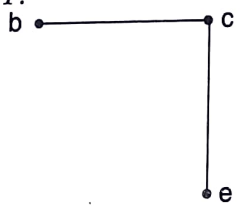
Step 1 : List the edges in non-decreasing order of their weights, as in Table 16.1

Edge :	(b, c)	(c, e)	(c, d)	(a, b)	(e, d)	(a, d)	(b, e)
Weight :	1	1	2	3	3	4	4

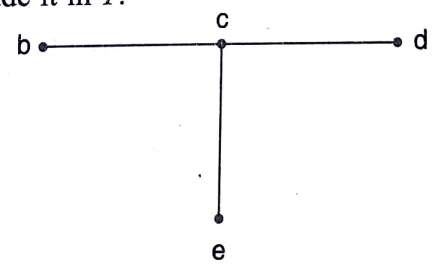
Table 16.1

Step 2 : Select the edge (b, c) since it has the smallest weight, include it in T.

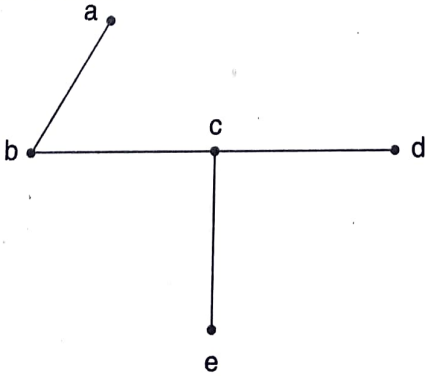
Step 3. Select an edge with the next smallest weight (c, e) since it does not form circuit with the existing edges in T, so include it in T.



Step 4. Select an edge with the next smallest weight (c, d) since it does not form circuit with the existing edges in T, so include it in T.



Step 5. Select an edge with the next smallest weight (a, b) since it does not form circuit with the existing edges in T, so include it in T.



Since G contains 5 vertices and we have chosen 4 edges, we stop the algorithm and the minimal spanning tree is produced.

Example 13. Show how Kruskal's algorithm finds a minimal spanning tree of the graph of Fig. 16.15.

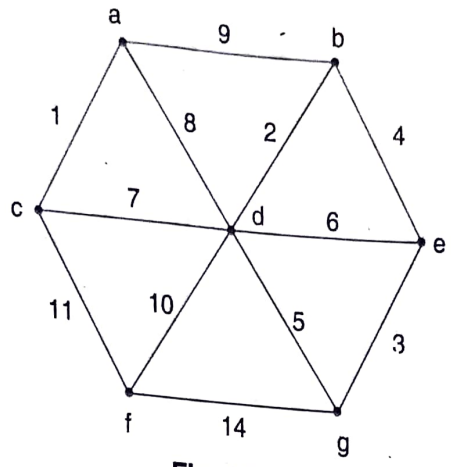


Fig. 16.15

Solution:

Step 1 : List the edges in non-decreasing order of their weights, as in Table 16.2

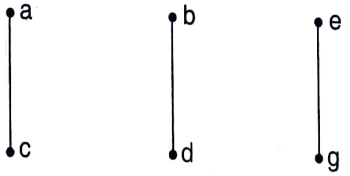
Edge :	(a, c)	(b, d)	(e, g)	(b, e)	(d, g)	(d, e)	(d, c)	(a, d)	(a, b)	(d, f)	(c, f)	(f, g)
Weight	1	2	3	4	5	6	7	8	9	10	11	12

Table 16.2

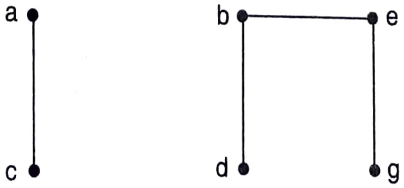
Step 2 : Select the edge (a, c) since it has the smallest weight, include it in T.

Step 3. Select an edge with the next smallest weight (b, d) since it does not form cycle with the existing edges in T, so include it in T.

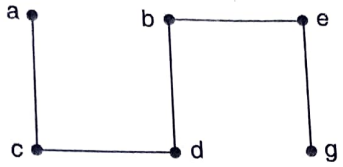
Step 4. Select an edge with the next smallest weight (e, g) since it does not form cycle with the existing edges in T, so include it in T.



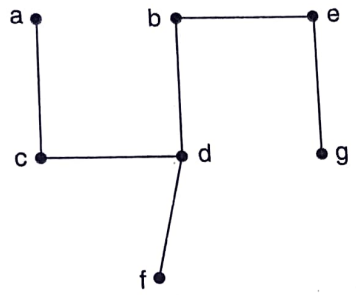
Step 5. Select an edge with the next smallest weight (b, e) since it does not form cycle with the existing edges in T, so include it in T.



Step 6. Select an edge with the next smallest weight (d, c) since it does not form cycle with the existing edges in T, so include it in T.

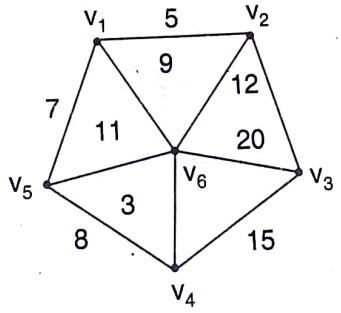


Step 7. Select an edge with the next smallest weight (d, f) since it does not form cycle with the existing edges in T, so include it in T.



Since G contains 7 vertices and we have chosen 6 edges, the process terminants and the minimal spanning tree is produced.

Example 14. Using Kruskal's algorithm find a spanning tree with minimum weight from the graph below. Also calculate the total weight of spanning tree.



Solution.

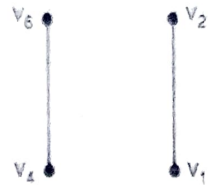
Step 1. List the edges in non-decreasing order of their weight as in Table 16.3

Edge	(v_4, v_6)	(v_1, v_2)	(v_1, v_5)	(v_5, v_4)	(v_1, v_6)	(v_2, v_3)	(v_5, v_6)	(v_6, v_2)	(v_4, v_3)	(v_3, v_6)
Weight	3	5	7	8	9	10	11	12	15	20

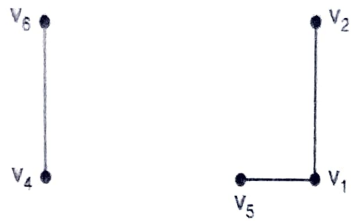
Table 16.3

Step 2. Select the edge (v_4, v_6) since it has the smallest weight 3, include it in T

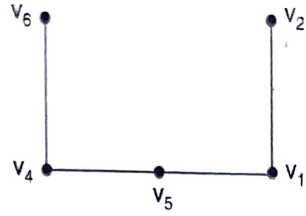
Step 3. Select an edge with the next smallest weight (v_1, v_2) since it does not form cycle with existing so T, include it in T.



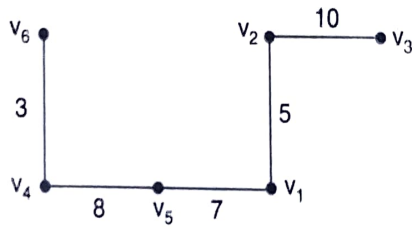
Step 4. Select an edge with the next smallest weight (v_1, v_5) since it does not form cycle with existing edges in T, so include it in T



Step 5. Select an edge with the next smallest weight (v_5, v_4) since it does not form cycle with existing edges in T, so include it in T



Step 6. Select an edge with the next smallest weight (v_2, v_3) since it does not cycle with the existing edges in T, so include it *in* T



Since G contains 6 vertices and we have chosen 5 edges, we stop the algorithm and the minimal Spanning tree is produced. The weight of the spanning tree is 3 + 8 + 7 + 5 10 = 33.

Prim's Algorithm

Initially the algorithm starting at a designated vertex chooses an edge with minimum weight and considers this edge and associated vertices as part of the desired tree. Then iterate, looking for an edge with minimum weight not yet selected that has one of its vertices in the tree while the other vertex is not. The process terminates when $n - 1$ edges have been selected from a graph of n vertices to form a minimal spanning tree. The algorithm involves the following steps.

Input : A connected weighted graph G.

Output : A minimal spanning tree T.

Step 1. Remove all self loops (if exist) and all parallel edges between two vertices except the one with minimum weight. Select any vertex in G. Among all the edges incident with the selected vertex, choose an edge of minimum weight. Include it in T.

Step 2. At each stage, choose an edge of smallest weight joining a vertex already included in T and a vertex not yet included, if it does not form a circuit with the edges in T. Include it in T.

Step 3. Repeat until all the vertices of G are included.

Example 15 (*a*). Find the minimal spanning tree of the weighted graph of Fig. 16.16 Prim's algorithm.

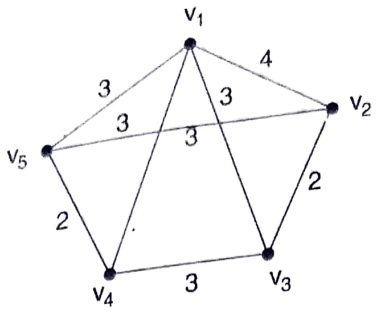
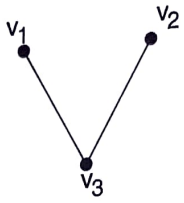


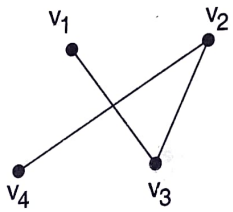
Fig. 16.16

Solution. 1. We choose the vertex v_1. Now edge with smallest weight incident on v_1 is (v_1, v_5) so we choose the edge [or (v_1, v_5)].

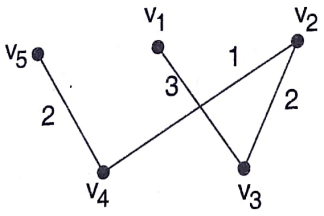
2. Now $w(v_1, v_2) = 4$, $w(v_1, v_4) = 3$, $w(v_1, v_5) = 3$, $w(v_3, v_2) = 2$ and $w(v_3, v_4) = 3$. We choose the edge (v_3, v_2) since it is minimum.



3. Again $w(v_1, v_5) = 3$, $w(v_2, v_4) = 1$ and $w(v_3, v_4) = 3$. We choose the edge (v_2, v_4)



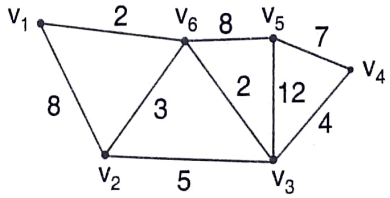
4. Now we choose the edge (v_4, v_5). Now all the vertices are convered. The minimal spanning tree is produced.



The weight of the minimal spanning tree is

$$3 + 2 + 1 + 2 = 8$$

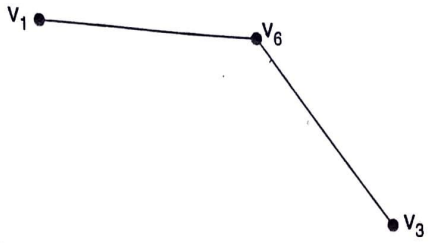
Example 16 (*a*). Describe Prim's algorithm and use this to find out the minimal spanning tree of the following graph.



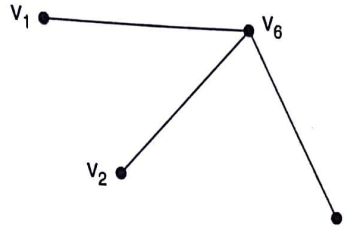
Solution. In order to find out the minimal spanning tree the following steps are followed.

Step 1. We choose the vertex v_1 as the starting vertex. The edge with smallest weight incident on v_1 is (v_1, v_6), as we choose the edge and include it in T (spanning tree).

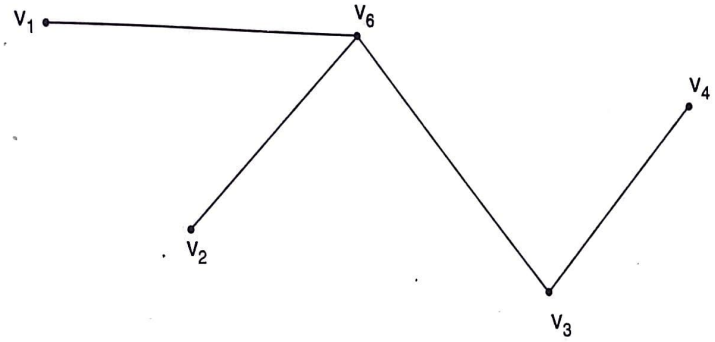
Step 2. Now $w(v_1, v_2) = 8$, $w(v_6, v_2) = 3$, $w(v_6, v_3) = 2$, and $w(v_6, v_5) = 8$. We choose the edge (v_6, v_3) since it is minimum and include it in T.



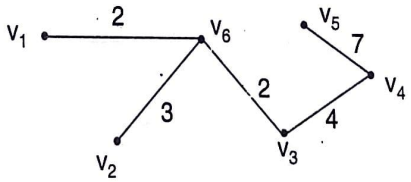
Step 3. Again $w(v_1, v_2) = 8$, $w(v_1, v_2) = 3$, $w(v_6, v_5) = 8$, $w(v_3, v_2) = 5$, $w(v_3, v_4) = 4$ and $w(v_3, v_5) = 12$. We choose (v_6, v_2) since it in minimum and include in T.



Step 4. Now among elligible edges, $w(v_6, v_5) = 8$, $w(v_3, v_5) = 12$, $w(v_3, v_4) = 4$ we choose (v_3, v_4) since it is minimum and include it in T.



Step 5. Only the vertex v_5 is not yet included. Now $(v_6, v_5) = 8$, $w(v_3, v_5) = 12$, $w(v_4, v_5) = 7$. We choose (v_4, v_5) since it is minimum and include it in T.



The minimal spanning tree is obtained and the total weight of the tree is

$2 + 2 + 3 + 4 + 7 = 18$

Prim's Algorithm in Tabular Form

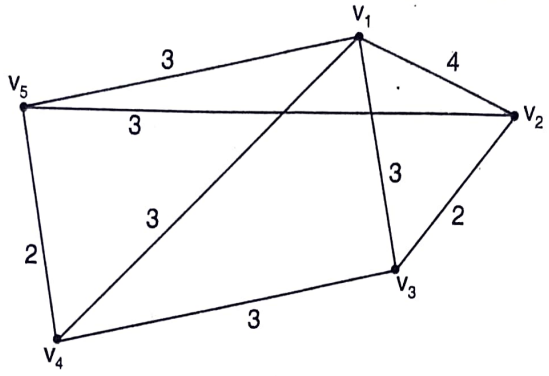
Step 1: Remove all self-loops (if exist) and all parallel edges between two vertices except the one with minimum weight. Label the vertices by $v_1, v_2, \ldots v_n$. Tabulate the given weights of the edges of G in an $n \times n$ table. This table is symmetric with respect to the diagonal and the diagonal is empty. If there is no edge connecting v_i to v_j, set the weight of the edge (v_i, v_j) as ∞.

Step 2: Start from the vertex v_1 *i.e.* first row and connect it to the vertex which has smallest entry in 1st row of the table, say, v_k, Then a tree v_1, v_k is formed.

Step 3: Connect the tree v_1, v_k to the vertex v_j ($\neq v_1, v_k$) that has smallest entry among all entries in 1st row and kth row. Thus a tree v_1, v_k, v_j is formed.

Step 4: Continue this process of selection of vertices until $(n-1)$ edges are selected (since a tree with n vertices has $n-1$ edges). These edges form a minimal spanning tree T in G.

Example 16 (*b*). Find by Prim's algorithm a minimal spanning tree from the following graph.



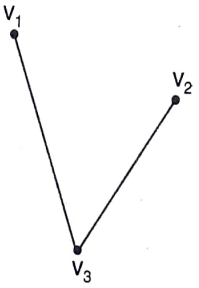
Solution: The given graph is a connected weighted graph having 5 vertices. So a minimal spanning tree has 4 edges.

Step 1: The tabulated form of the weights of its edges are given below:

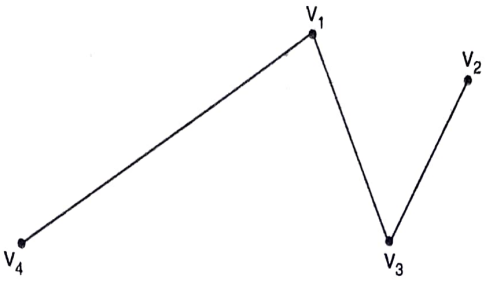
	v_1	v_2	v_3	v_4	v_5
v_1	–	4	3	3	3
v_2	4	–	2	∞	3
v_3	3	2	–	3	∞
v_4	3	∞	3	–	2
v_5	3	3	∞	2	–

Step 2: Start form the vertex v_1. The smallest entry in v_1-row, *i.e.*, the 1st row of the table is 3, which corresponds to the vertex v_3 or v_4 or v_5. Take any one of these three vertices, say v_3, thus the tree v_1, v_3 is formed as shown below:

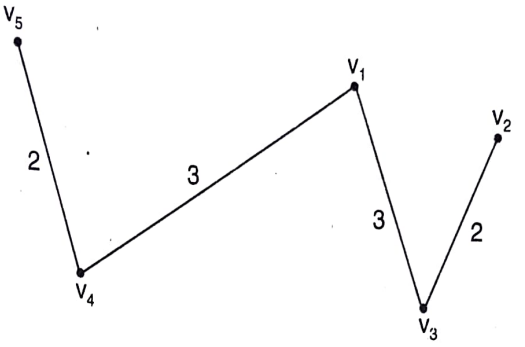
Step 3: Smallest entry in v_1-row and v_3-row, *i.e.*, 1st and 3rd rows is 2 which corresponds to v_2. Then the tree v_1, v_3, v_2 is formed as shown below:



Step 4: Smallest entry in v_1-row, v_2-row and v_3-row, *i.e.*, 1st, 2nd and 3rd rows is 3 (we do not take 2 since it corresponds to v_2 or v_3) which corresponds to v_4 or v_5 (except v_1, v_3 which are already selected). Take anyone of these two vertices, say, v_4. Thus the tree v_1, v_3, v_2, v_4 is formed as shown below:



Step 5: The smallest entry in v_1-row, v_2-row, v_3-row and v_4-row, *i.e.*, 1st, 2nd, 3rd and 4th row is 2 which corresponds to v_5 (except v_2, v_3 which are already taken). Thus the tree v_1, v_3, v_2, v_4, v_5 is formed as shown below:



Since this tree has 4 (= 5–1) edges, this is the required minimal spanning tree. The weight of this minimal spanning tree is 2 + 3 + 3 + 2 = 10.

Difference between Prim's and Kruskal's Algorithm

In prim's algorithm at every step edges of minimum weight that are incident to a vertex, already in the tree and not forming a cycle, are chosen.

But in Kruskal's algorithm, edges of minimum weight that are not necessarily incident to a vertex already in the tree and not forming a cycle is chosen.

16.4. Binary Tree

A binary tree is a rooted tree in which each vertex has atmost two children. Each child in a binary tree is designated either a **left child** or a **right child** (not both), and an internal vertex has at most one left and one right child. A **full binary** is a tree in which each internal vertex has exactly two children.

Given an internal vertex v of a binary tree T, the **left subtree** of v is the binary tree whose root is the left child of v, whose vertices consist of the left child of v and all its descendants, and whose edges consist of all those edges of T that connect the vertices of the left subtree together. The **right subtree** of v is defined analogously.

Fig. 16.17 (*a*) is a binary tree and Fig. 16.17 (*b*) is a full binary tree since each of its internal vertices has two children.

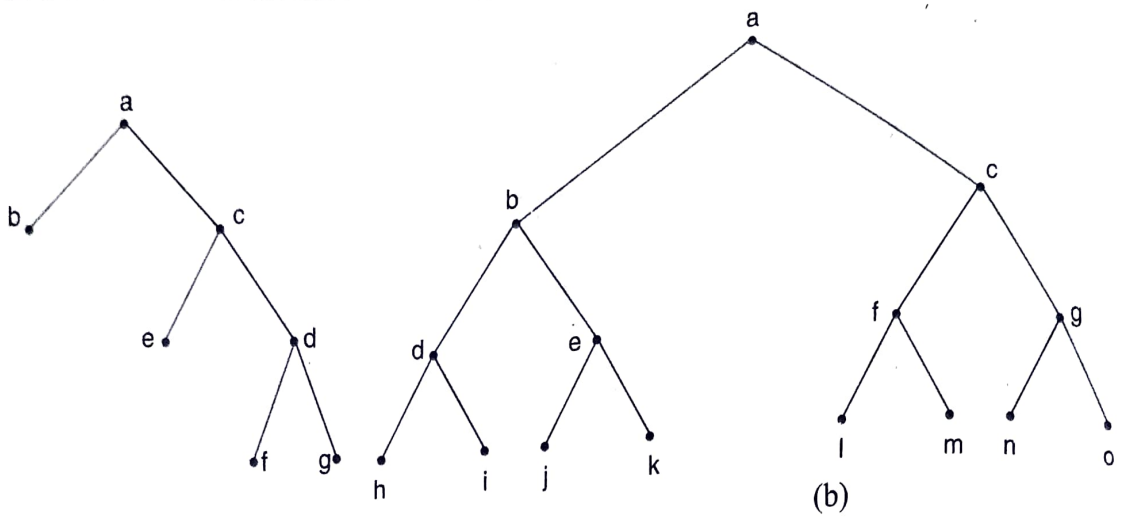


Fig. 16.17

A tree in which there is exactly one vertex of degree two and each of the other vertices is of degree one or three is called a binary tree. The vertex of degree two is called root of the tree.

Example 17. What are the left and right children of b shown in Fig. 16.18? What are the left and right subtrees of a?

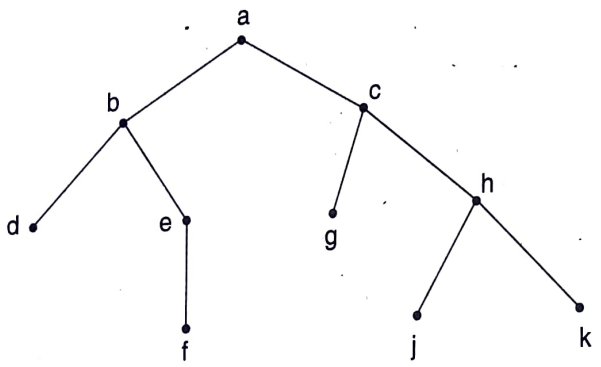


Fig. 16.18

Solution. The left child of b is d and the right child is e. The left subtree of the vertex a consists of the vertices b, d, e and f and the right subtree of a consists of the vertices c, g, h, j and k whose figures are shown in Fig. 16.19 (a) and (b) respectively.

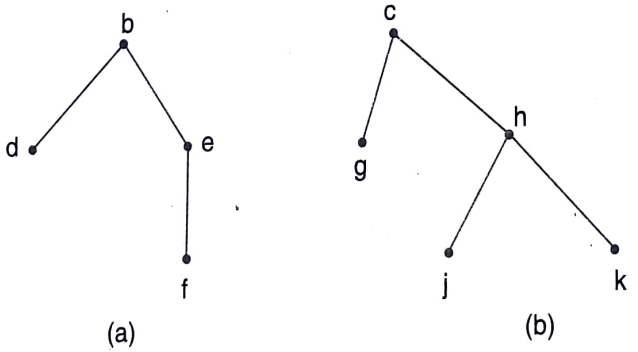


Fig. 16.19

Properties of Binary Trees

Theorem 16.9 The number of vertices in a binary tree is always odd.

Proof : Consider a binary tree on n vertices. This tree possess exactly one vertex of degree two and other vertices of degree one or three. Thus there are n-1 vertices of odd degrees. Since the number of vertices of odd degree in a graph is always even, it follows than n-1 is even and hence n is odd.

Theorem 16.10. In any binary tree T on n vertices, the number of pendent vertices is equal to $(n + 1)/2$.

Proof: Let the number of pendant vertices in a binary tree on n vertices be k. There is exactly one vertex of degree 2, k vertices of degree 1 and the remaining $(n - k - 1)$ vertices of degree 3. We know

Sum of degrees of vertices = 2 × no. of edges.

$3 \times (n - k - 1) + 2 \times 1 + k \times 1 = 2\,(n - 1)$

i.e., $3n - 2k - 1 = 2n - 2$

or $k = (n + 1)/2$

Theorem 16.11 The number of internal vertices in a binary tree is one less than the number of pendant vertices.

Proof. Let the binary tree contains $x + y$ vertices, where x = number of pendant vertices, y = number of non-pendant vertices *i.e.*, internal vertices in the binary tree. Therefore, the total number of vertices $n = x + y$.

By property 2, $x = (n + 1)/2$ So, $x = (x + y + 1)/2 => y = x - 1$.

This shows that

Number of internal vertices = Number of pendant vertices – 1.

Theorem 16.12 Prove that the maximum number of vertices on level n of a binary tree is 2^n where $n \geq 0$.

Proof. The root of a binary tree is on level 0 and there can only one vertex at this level. The maximum number of vertices on level 1 is 2, on level 2 is $4 = 2^2$ and so on. We prove the theorem by mathematical induction.

Basis of induction : When $n = 0$, the only vertex is the root. Thus the maximum number of vertices on level $n = 0$ is $2^0 = 1$.

Induction hypothesis : We assume that the theorem is true for level k, where $n \geq k \geq 0$. So, The maximum number of vertices on level k is 2^k.

Induction Step : Since each vertex in binary tree has maximum degree 2, then the maximum number of vertices on level $k + 1$ is twice the maximum number of level k. Hence, the maximum number of vertices at level $k = 1$ is $2.2^k = 2^{k+1}$. Hence, the theorem is proved.

Theorem 16.13 Prove that the maximum number of vertices in a binary tree of height h is $2^{h+1} - 1$, $h \geq 0$.

Proof. It is known that the maximum number of vertices on level n of a binary tree is 2^n. So, the maximum number of vertices in a binary of height h is

$$1 + 2 + 2^2 + \ldots + 2^h = \frac{2^{h+1} - 1}{2 - 1}$$

$$= 2^{h+1} - 1$$

Theorem 16.14. Show that the minimum height of a binary tree on n vertices is $\log_2 (n + 1) - 1$ and maximum possible height is $\frac{n-1}{2}$.

Proof. Let T be a binary tree of n vertices. The maximum number of vertices in the binary tree of height h is $2^{h+1} - 1$. But T has n vertices.

$$2^{h+1} - 1 \geq n$$

i.e., $$h \geq \log_2(n + 1) - 1$$

But h is an integer, hence the minimum possible height of an n-vertex binary tree is min $n = \lceil \log_2(n + 1) - 1 \rceil$ where $\lceil n \rceil$ denotes the smallest integer greater than or equal to n.

Let l denotes the maximum possible height of T. We have the root of T at zero level, 2 vertices at level 1, 2 vertices at level 2, ..., 2 vertices at level l. We have at least $1 + (2 + 2 + \ldots\ldots$ l times$) = 1 + 2l$. But T has n vertices

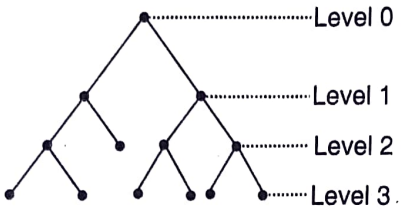
So $$1 + 2l \leq n \quad i.e., \quad l \leq \frac{n-1}{2}$$

But n is odd, so $\frac{n-1}{2}$ is an integer.

$\therefore$ The number of possible value of l is $\frac{n-1}{2}$

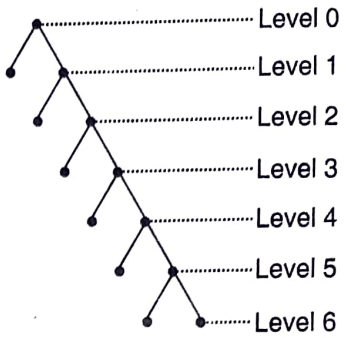
$$\max h = \frac{n-1}{2}$$

For example : For $n = 13$



(a) 3-level, 13-vertex binary tree

$$\min h = \lceil (\log_2 14) - 1 \rceil = 3$$



(b) 6-level, 13-vertex binary tree

$$\max h = \frac{13-1}{2} = 6$$

Theorem 16.15. If T is full binary tree with i internal vertices, then T has $i + 1$ terminal vertices and $2i + 1$ total vertices.

Proof. The vertices of T consists of the vertices that are children (of some parent 0) and the vertices that are not children (of any parent). There is one nonchild – the root. Since there are 1 internal vertices, each parent having two children, there are $2i$ children, thus there are total $2i + 1$ vertices and the number of terminal vertices is $(2i + 1) - i = i + 1$.

Complete binary tree

If all the leaves of a full binary tree are at level d, then we call such a tree as a complete binary tree of depth d. A complete binary tree of depth of 3 is shown in Fig. 16.20. If T is a complete binary tree with n vertices, then the vertices at any level l are given the label numbers ranging from $2l$ to $2^{l+1} - 1$ or from 2^l to n if n is less than $2^{l+1} - 1$.

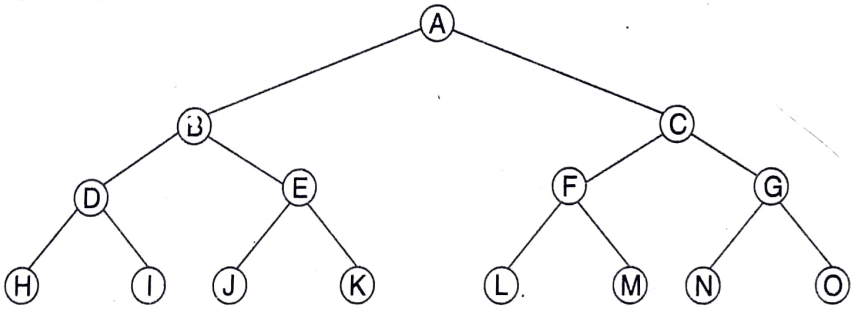


Fig. 16.20. A complete binary tree.

16.5. Tree Traversal

A traversal of a tree is a process to traverse a tree in a systematic way so that each vertex is visited exactly once. Three commonly used traversals are preorder, postorder and inorder. We describe here these three process that may be used to traverse a binary tree.

Preorder Traversal: The preorder traversal of a binary tree is defined recursively as follows.

(*i*) Visit the root

(*ii*) Traverse the left subtree in preorder

(*iii*) Traverse the right subtree in preorder

Postorder Traversal: The postorder traversal of a binary tree is defined recursively as follows

(*i*) Traverse the left subtree in postorder

(*ii*) Traverse the right subtree in postorder

(*iii*) Visit the root

Inorder Traversal: The inorder traversal of a binary tree is defined recursively as follows

(*i*) Traverse in inorder the left subtree

(*ii*) Visit the root

(*iii*) Traverse in inorder the right subtree

For example, we find the Preorder, Inorder and Post order traversal of the following tree *T*.

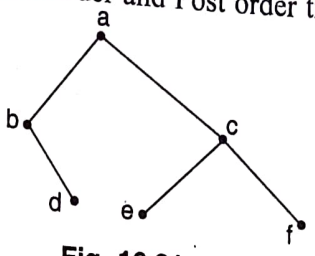


Fig. 16.21

Preorder Traversal

1. First visit the root **a**

2. Traverse the left subtree and visit the root **b**. Now traverse the left subtree with b as the root; it is empty. The traverse the right subtree of *b* and visit the root **d**. There is no subtree of *d*.

3. Then back to the root *a*, traverse the right subtree and visit the root **c**. Traverse the left subtree with *c* as the root and visit the root **e**. The subtree of *e* is empty. Then traverse the right subtree of *c* and visit the root **f**. The root f has no subtree.

All the vertices have been covered. Hence output is **a b d c e f**

Inorder Traversal

1. First traverse the left subtree with root a in inorder. Again traverse the left subtree with root *b* in inorder. It is empty, so visit **b**. Now traverse right subtree of *b*. Then traverse left subtree of *d* which is empty, visit **d**. The right subtree of *d* is empty.

2. Back to a and visit **a**.

3. Traverse the right subtree of a in inorder. Again traverse the left subtree of *c* inorder. Then the left subtree of *e* in inorder; it is empty. So visit **e**. Traverse the right subtree of *c*. There is no subtree of *f*, so visit **f**, so visit **f**. Back *c* and visit **c**.

All the vertices have been covered. Hence output is **b d a e f c.**

Postorder Traversal

1. Traverse the left subtree with root a in post order.

2. Traverse the left subtree with root *b* in postorder. The left subtree of *b* is empty. Traverse the right subtree. Since *d* has no subtree, visit **d**. Then back to *b* and visit **b**.

Then back to a and traverse the right subtree of *a*. Traverse the left subtree with *c* as root. Since *e* has no subtree, visit **e**. Traverse the right subtree with *c* as root. Since *f* has no subtree, visit f. Then back to *c* and visit **c**.

3. Back to the root a and visit **a**.

All the vertices have been covered. Hence output is **d b e f c a**

Given an order of traversal of a tree it is possible to construct a tree. For example consider the following order:

Inorder = $d\ b\ e\ a\ c$

We can construct the binary trees shown below in Fig. 16.22 using this order of traversal:

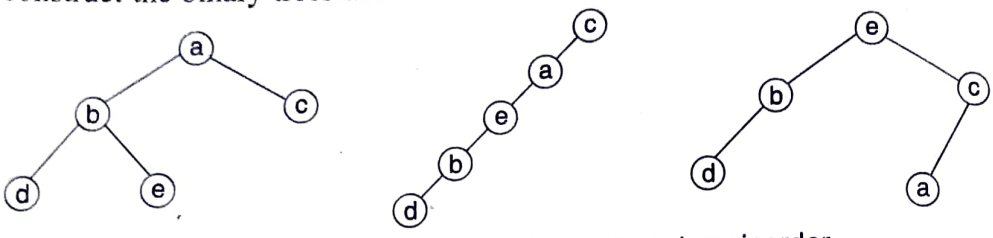


Fig. 16.22. Binary trees constructed using given inorder

Therefore, we can conclude that given only one order of traversal of a tree it is possible to construct a number of binary trees, a unique binary tree is not possible to be constructed. For construction of a unique binary tree we require two orders in which one has to be inorder, the other can be preorder or postorder.

To Draw a Unique Binary Tree when Inorder and Preorder Traversal of the Tree is Given.

1. The root of T is obtained by choosing the first vertex in its preorder.

2. The left child of the root vertex is obtained as follows. First use the inorder traversal to find the vertices in the left subtree of the binary tree (all the vertices to the left of this vertex in the inorder traversal are the part of the left subtree). The left child of the root is obtained by selecting the first vertex in the preorder traversal of the left subtree Draw the left child.

3. Use the inorder traversal to find the vertices in the right subtree of the binary tree (all the vertices to the right of the first vertex are the part of the right subtree). Then the right child is obtained by selecting the first vertex in the preorder traversal of the right subtree. Draw the right child.

4. The procedure is repeated recursively until every vertex is not visited in preorder.

Example 18. Given the preorder and inorder traversal of a binary tree, draw the unique tree

Preorder : $g\ b\ q\ a\ c\ p\ d\ e\ r$

Inorder : $q\ b\ c\ a\ g\ p\ e\ d\ r$

Solution. Here g is the first vertex in preorder traversal, thus g is the root of the tree. Using inorder traversal, left subtree of g consists of the vertices q, b, c and a. Then the left child g is b since b is the first vertex in the preorder traversal in the left subtree. Similarly, right subtree of g consists of the vertices p, e, d and r. then the right child of g is p since p is the vertex in the preorder traversal in the right subtree.

Repeating the above process with each node, we finally obtain the required tree as shown in Fig. 16.23.

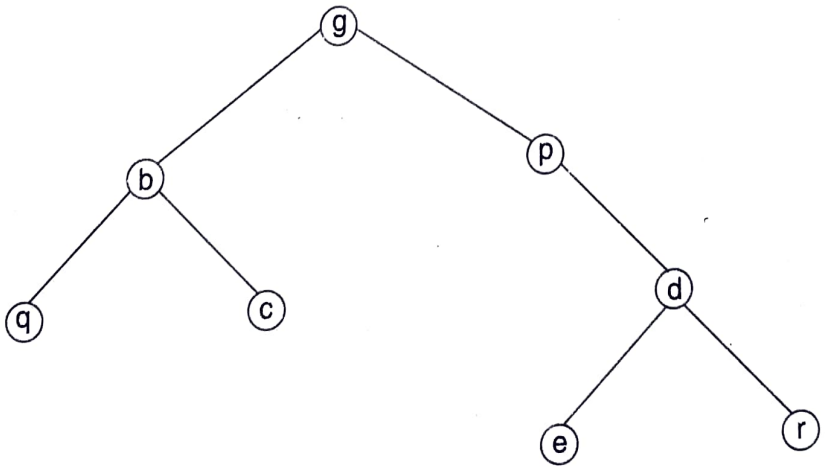


Fig. 16.23

To Draw a Unique Binary Tree when Inorder and Postorder Traversal of the Tree is Given.

1. The root of the binary tree is obtained by choosing the last vertex in the postorder traversal
2. The right child of the root vertex is obtained as follows. First use the inorder traversal to find the vertices in the right subtree (All the vertices right to the root vertex in the inorder traversal are the vertices of the right subtree). The right child of the root is obtained by selecting the last vertex in the postorder traversal. Draw the right child.
3. Use the inorder traversal to find the vertices in the left subtree of the binary tree. Then the left child is obtained by selecting the last vertex in the post order traversal of the left subtree. Draw the left child.
4. The process is repeated recursively until every vertex is not visited in postorder.

Example 19. Given the postorder and inorder traversal of a binary tree, draw the unique binary tree

Postorder : *d e c f b h i g a*

Inorder : *d c e b f a h g i*

Solution. Here a is the last vertex in postorder traversal, thus a is the root of the tree. Using inorder traversal, right subtree of root vertex a consists of the vertices *h, g* and *i*. The right child of *a* is *g* since *g* is the last vertex in the post order traversal in the right subtree. Similarly, left subtree of a consists of the vertices *d, c, e, b* and *f* then the left child of *a* is *b* since *b* is the last vertex in the postorder traversal in the left subtree. Repeating the above process with each vertex, we finally obtain the required tree as shown in Fig. 16.24.

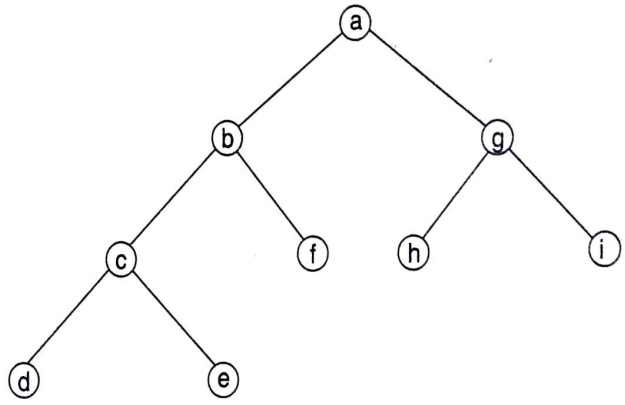


Fig. 16.24

Representation of Algebraic Structure by Binary Trees

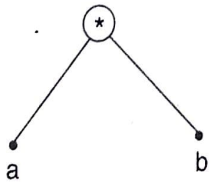
Binary trees are used to represent algebraic expressions, the vertices of the tree are labled with the numbers, veriables, or operations that make up the expression. The leaves of the tree can be labled with numbers or variables. Operations such as addition, subtraction, multiplication, division, or exponentiation can only be assigned to internal vertices. The operation at each vertex operates on its left and right subtrees from left to right.

Example 20. Use a binary tree to represent the expression

(*i*) $a * b$ (*ii*) $(a + b) / c$

(*iii*) $(a + b) * (c / d)$ (*iv*) $(a + b) * c) + (d / e)$

Solution. (*i*)



(*ii*)

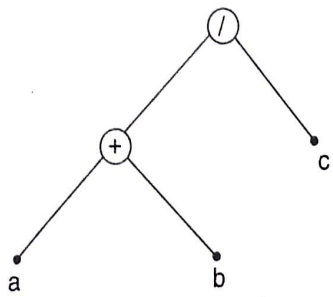


Fig. 16.25

(*iii*)

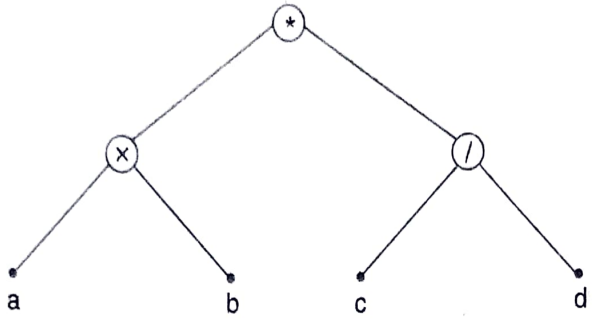


Fig. 16.26

(*iv*)

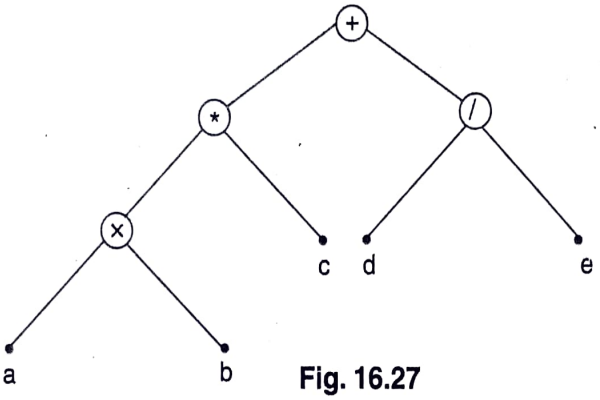


Fig. 16.27

Example 21. Determine the value of the expression represented in a binary tree shown in Fig. 16.28.

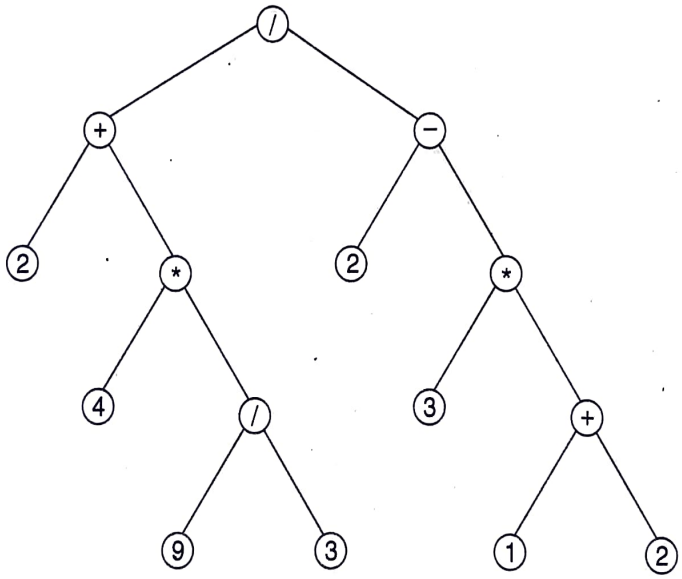


Fig. 16.28

Solution. The expression represented by the binary tree is (9/3 * 4 + 2) / (((1 + 2) * 3) – 2) and the value is (3 * 4 + 2) / ((3 * 3) – 2)

= (12 + 2) / (9 – 2)

= 14 / 7 = 2.

Infix, Prefix and Postfix Notation of an Arithmetic Expression

We know that even for fully parenthesised expression a repeated scanning of the expression is still required in order to evaluate the expression. This phenomenon is due to the fact that operators appear with the operands inside the expression. We can represent expressions in three different ways. They are Infix, Prefix and Postfix forms of an expression.

Infix Notation

The notation used in writing the operator between its operands is called infix notation. The infix form of an algebraic expression is the inorder traversal of the binary tree representing the expression. It gives the original expression with the elements and operations in the same order as they originally occurred. To make the infix forms of an expression unambiguous it is necessary to include parentheses in the inorder traversal, whenever we encounter an operation.

Prefix Notation

The repeated scanning of an infix expression is avoided if it is converted first to an equivalent parenthesis – free of **polish notation.** The prefix form of an expression is the preorder traversal of the binary tree representing the given expression. The expression in prefix notation are unambiguous, so that no parentheses are needed in such expression.

Post fix Notation

The post fix form of an expression is the post order traversal of the binary tree representing the given expression. Expressions written in post fix form are said to be in **reverse polish notation.** Expressions in this notation are unambiguous, so that parentheses are not needed.

Table below gives the equivalent forms of several fully parenthesised expressions. Note that in both the prefix and post fix equivalents of such an infix expression, the variable names are all in the same relative position.

Infix	*Prefix*	*Postfix*
$(x * y) + z)$	$+ * xyz$	$xy* z +$
$((x + y) * (z + t)$	$* + xy + zt$	$xy + zt + *$
$((x + y * z) - (u / v + w))$	$- + x * yz + / uvw$	$xyz* + uv/w + -$

Example 22. Represent the expression as a binary tree and write the prefix and postfix forms of the expression.

$$A * B - C \uparrow D + E / F$$

Solution. The binary tree representing the given expression is shown below.

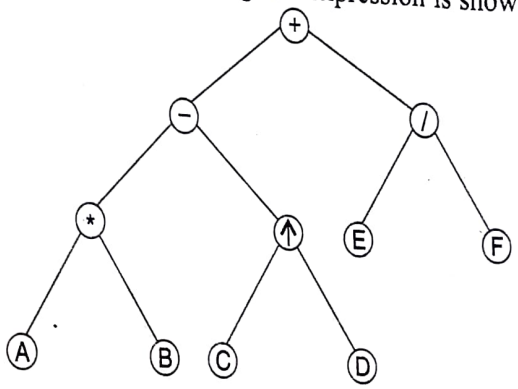


Fig. 16.29

Prefix : $+ - * AB \uparrow CD / EF$

Postifix : $AB * CD \uparrow - EF / +$

Evaluating prefix and postfix form of an expression

To evaluate an expression in prefix form, proceed as follows. Move from left to right until we find a string of the form Fxy, where F is the symbol for a binary operator and x and y are two operands. Evaluate xFy and substitue the result for the string Fxy. Consider the result as a new operand and continue this procedure until only one number remains.

When an expression is in postfix form, it is evaluated in a manner similar to that used for prefix form, except that the operator symbol is after its operands rather than before them.

Example 23. What is the value of

(*a*) prefix expression * – 84 + 6 / 42

(*b*) postfix expression 823* – 2 ↑ 63 / +

Solution. (*a*) The evaluation is carried out in the following sequence of steps.

1. * – 84 + 6 / 42
2. *4 + 6 / 42 since the first string in the F*xy* is – 84 and 8 – 4 = 4
3. *4 + 62 replacing / 42 by 4 / 2 = 2
4. *48 replacing + 62 = 8
5. 32 replacing *48 by 4* 8 = 32

(*b*) The evaluation is carried out in the following sequence of steps.

1. 823* – 2 ↑ 63 / +
2. 86 – 2 ↑ 63 / + replacing 23 * by 2 * 3 = 6
3. 22 ↑ 63 / + replacing 86 – by 8 – 6 = 2
4. 463 / replacing 2 ↑ 2 by $2^2 = 4$
5. 42 + replacing 63 / by 6 / 3 = 2
6. 6 replacing 42 + by 4 + 2 = 6

Binary Search Trees

A binary search tree is basically a binary tree, and therefore it can be traversed in preorder, postorder, and inorder. If we traverse a binary search tree in inorder and print the identifiers contained in the vertices of the tree, we get a sorted list of identifiers in the ascending order.

Binary trees are used extensively in computer science to store elements from an ordered set such as a set of numbers or a set of strings. Suppose we have a set of strings and numbers. We call them as keys. We are interested in two of the many operations that can be performed on this set.

1. Ordering (or sorting) the set.
2. Searching the ordered set to locate a certain key and, in the event of not finding the key in the set, adding it at the right position so that the ordering of the set is maintained.

Definition. A binary search tree is a binary tree T in which data are associated with the vertices. The data are arranged so that, for each vertex v in T, each data item in the left subtree of v is less than the data item in v and each data item in the right subtree of v is greater than the data item in v. Thus, A binary search tree for a set S is a **labled binary tree** in which each vertex v is labelled by an element $l(v) \in S$ such that

1. for each vertex u in the left subtree of v, $l(u) < l(v)$,
2. for each vertex u in the right subtree of v, $l(u) > l(v)$, and
3. for each element $a \in S$, there is exactly one vertex v such that $l(v) = a$

The binary tree T in Fig. 16.30. is a binary search tree since every vertex in T exceeds every number in its left subtree and is less than every number in its right subtree.

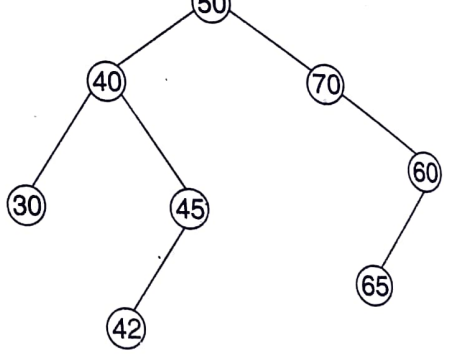


Fig. 16.30 A binary search tree

Creating a Binary Search Tree

The following recursive procedure is used to form the binary search tree for a list of items. To start, we create a vertex and place the first item in the list in this vertex and assign this as the key of the root. To add a new item, first compare it with the keys of vertices already in the tree, starting at the root and moving to the left if the item is less than the key of the respective vertex if this has a left child, or moving to the right if the item is greater than the key of the respective vertex if this vertex has a right child. When the item is less than the respective vertex and this vertex has no left

child, then a new vertex with this item as its key is inserted as a new left child. Similarly, when the item is greater than the respective vertex and this vertex has no right child, then a new vertex with this item as its key is inserted as a new right child. In this way, we store all the items in the list in the tree and thus create a binary search tree.

Example 24. Form a binary search tree

(*i*) for the data 16, 24, 7, 5, 8, 20, 40, 3 in the given order.

(*ii*) for the words *if, then, end, begin, else* (used as keywords in ALGOL) in lexicographic order.

Solution. (*i*) We begin by selecting the number 16 to be the root. Since the next number 24 is greater than 16, add a right child of the root and level it with 24. We choose next element in the list 7 and again start at the root and compare it with 16. Since 7 is less than 16, add a left child of the root and level it with 7. We compare 5 to 7, since 7 is greater than 5, then we move further down to the left child of 7 and level the vertex to 5. Similar procedure is followed for left out numbers in the list. The Fig. 16.31 shows the binary search tree.

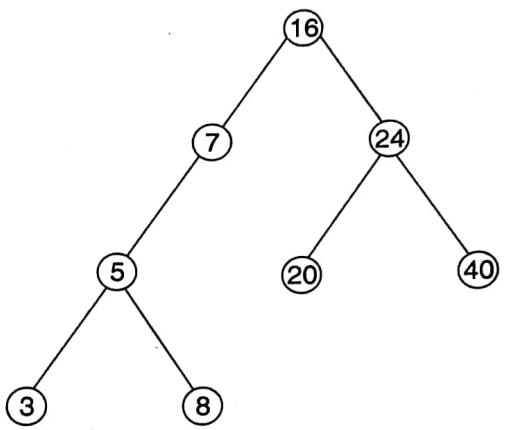


Fig. 16.31

(*ii*) We start the word *if* as the key of the root. Since *then* comes after *if* (in alphabetical order), add a right child of the root with key *then*. Since the next word *end* comes before *if*, add a left child of the root with key *end*. The next word *begin* is compared with *if*. Since *begin* is before *if* we move down to the left child of *if*, which is the vertex lebelled *end*. We compare *end* with *begin*. Since *begin* comes before *end*, then we move further down to the left child of *end* and level with key *begin*. Similarly *else* comes after *begin*, we move further down to the right child of *begin* and level with key *else*. The Fig. 16.32 shows the binary search tree.

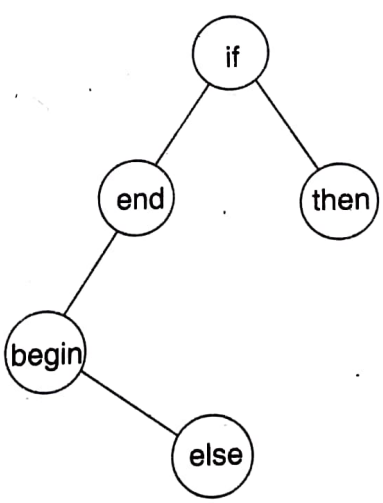


Fig. 16.32

Storage Representation of Binary Tree

This section discusses two ways of representing a binary tree in computer memory. The first way uses a single array, called the sequential representation of binary tree. The second is called the link representation.

Sequential Representation

We can represent the vertices of a binary tree as array elements and access the vertices using array notations. The advantage is that we need not use a chain of pointers connecting the widely separated vertices.

Consider the almost complete binary tree shown in Fig. 16.33.

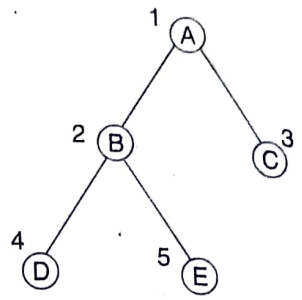


Fig. 16.33

Array Positions	1	2	3	4	5
Vertices	A	B	C	D	E

Fig. 16.34

Note the we assigned numbers for all the nodes. We can assign numbers in such a way that the root is assigned the number 1, a left child is assigned twice the number assigned to its father, a right child is assigned one more than twice the number assigned to its father. We can keep the vertices of an almost complete binary tree in an array. Fig. 16.34 shows vertices kept in an array.

By this convention, we can map vertex i to ith index in the array, and the parent of vertex i will get mapped at an index $i/2$ where as left child of vertex i gets mapped at an index $2i$ and right child gets mapped at an index $2i + 1$. The sequential representation can be extended to general binary trees. We do this by identifying an almost complete binary tree that contains the binary tree being represented. An almost complete binary tree containing the binary tree in Fig. 16.35. is shown in Fig. 16.36.

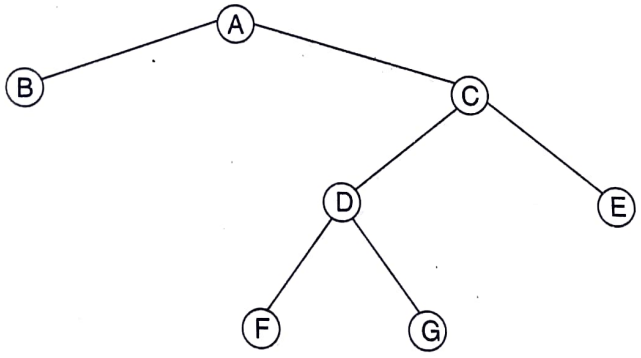


Fig. 16.35 A binary tree

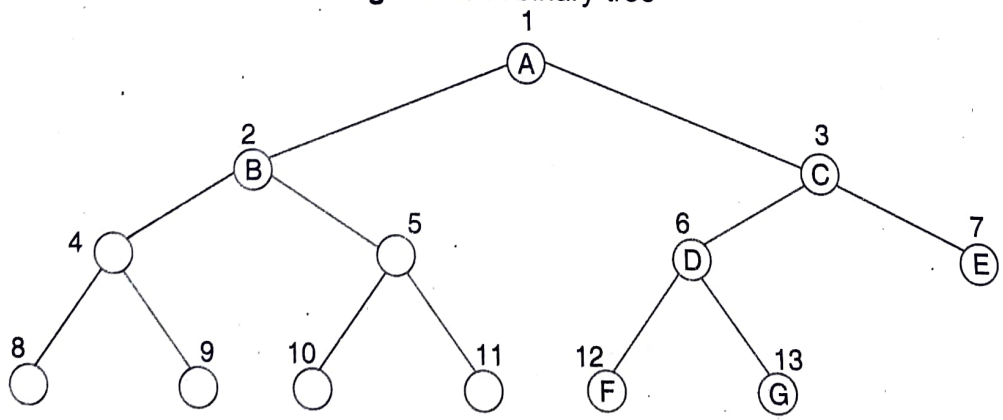


Fig. 16.36 An almost complete binary tree containing the binary tree in Fig. 16.35

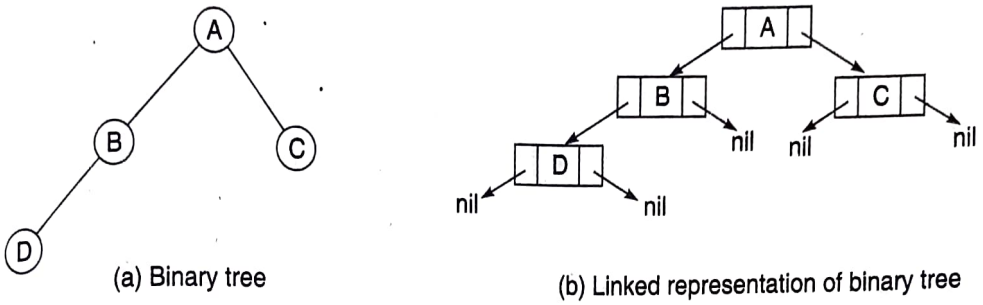
The vertices kept in an array is shown below :

Array Positions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	·····	40
Vertices	A	B	C			D	E					F	G				....	

Linked Representation

An array representation of a binary tree is not suitable for frequent insertions and deletions, and therefore we find that eventhough no storage is wasted if the binary tree is a complete one when array representation is used, it makes insertion and deletion in a tree is costly. Computer representation of trees based on linked allocation seems to more popular because of the ease with which nodes can be inserted in and deleted from a tree, and because tree structures can grow to an arbitary size. Therefore instead of using an array representation, one can use a linked representation, in which every node is represented as a structure having 3 fields, one for holding data, one for linking it with left sub-tree and the one for linking it with right sub-tree as shown below. A general tree can easily be converted into an equivalent binary tree by using the natural correspondence algorithm.

Where LLINK or RLINK contain a pointer to the left sub-tree respectively of the node in question. DATA contains the information which is to be associated with this particular node. Each pointer can have a value of NULL. An example of abinary tree as a graph and its corrosponding linked representation in memory are given in Fig. 16.37.



(a) Binary tree

(b) Linked representation of binary tree

Fig. 16.37

SOLVED EXAMPLES

Example 25. Does there exist trees for the degree sequence (*i*) 1, 1, 1, 1, 2, 2, 2, 3, 3 (*ii*) 4, 4, 3, 2, 1, 1, 1, 1, 1 ? If Yes, then draw the tree-else explain why such a tree cannot exit ?

Solution. (*i*) The figure drawn below is a tree having the degree sequence given in (*i*)

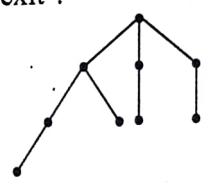


Fig. 16.38

(*ii*) Here edges = $1 \times 5 + 2 \times 1 + 3 \times 1 + 4 \times 2 = 18$ and the number of edges = $2e = 2(n-1) = 2 \times (9 - 1 = 16)$ So, the condition Edges = $2(n-1)$ is not satisfied. Hence these does not exist a tree with degree sequence given in (*ii*).

Example 26. Let T be a tree having 32 edges. On removing a certain edge from T, two disjoint trees T_1 and T_2 are obtained. If the number of vertices in T_1 is double that of T_2, find the number of edges in T_1 and T_2.

Solution. Given tree T has 32 edges. We know that removal of an edge does not affect the total number of vertices of a graph. Let n be the number of vertices of the tree T_2.

Then the number of vertices of T_1 is $2n$. Hence, the number of edges of T_1 and T_2 and $2n - 1$ and $n - 1$ respectively.

$\therefore \quad (2n-1) + (n-1) = 32 \quad \Rightarrow \quad n = 11.$

Hence the number of vertices of T_2 is 11 and that of T_1 is 22. Consequently, the number of edges of T_1 is $22 - 1 = 21$, and that of T_2 is $21 - 11 = 10$.

Example 27. Which trees are complete bipartite graphs?

Solution. Let T be a tree which is a complete bipartite graph. Let $T = K_{m,n}$. Then the number of vertices in T is $m + n$. Hence it has $m + n - 1$ number of edges. Again the graph $K_{m,n}$ has $m \times n$ number of edges.

$\therefore \quad mn = m + n - 1 \Rightarrow (m-1)(n-1) = 0 \Rightarrow m = 1, n = 1.$

It follows that T is either $K_{1,n}$ or $K_{m,1}$ *i.e.* T is a star. For example,

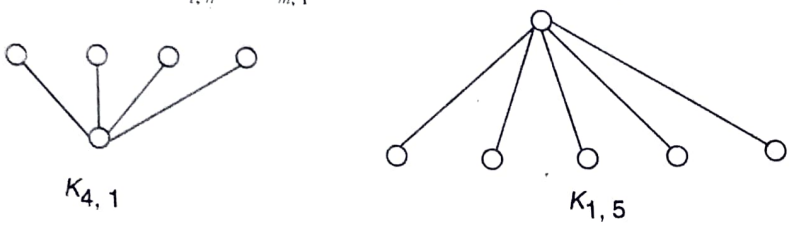


Fig. 16.39

are Complete Bipartite Trees

Example 28. Obtain a minimal spanning tree of the following graph using Kruskal's algorithm:

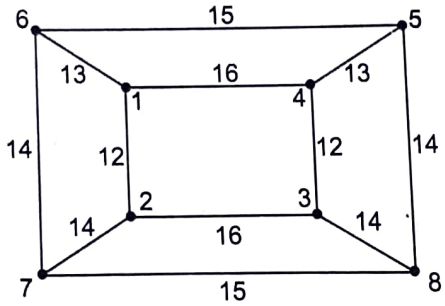


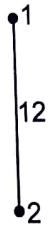
Fig. 16.40

Solution. Here the given connected weighted graph G (say) contains 8 vertices. So the minimal spanning tree has 7 edges.

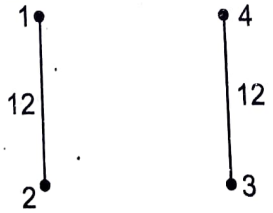
Step 1. List all edges (except self-loops, if any) of the graph G in order of nondecreasing weight.

Edges :	(1, 2)	(3, 4)	(1, 6)	(4, 5)	(2, 7)	(3, 8)	(6, 7)	(5, 8)	(5, 6)	(7, 8)	(1, 4)	(2, 3)
Weight :	12	12	13	13	14	14	14	14	15	15	16	16

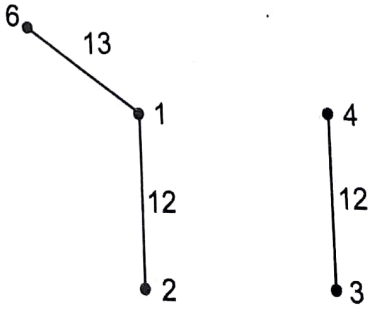
Step 2. Select the smallest edge (1, 2), as shown below:



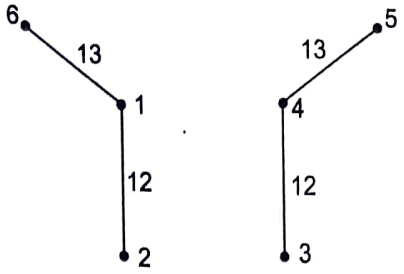
Step 3. Select the next smallest edge (3, 4), because it does not form any cycle with (1, 2).



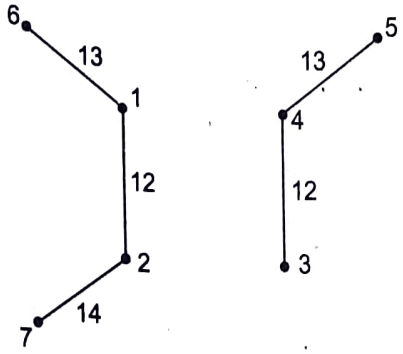
Step 4. Select the next smallest edge (1, 6), since it does not form any cycle with the previously selected edges.



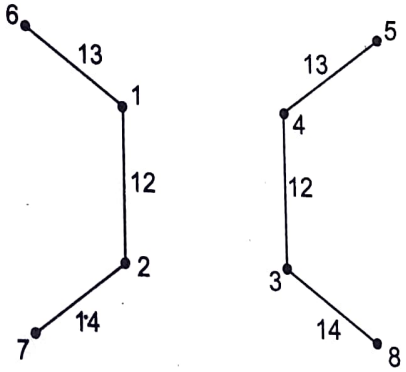
Step 5. Select the next smallest edge (4, 5), since it does not form any circuit with the previously selected edges.



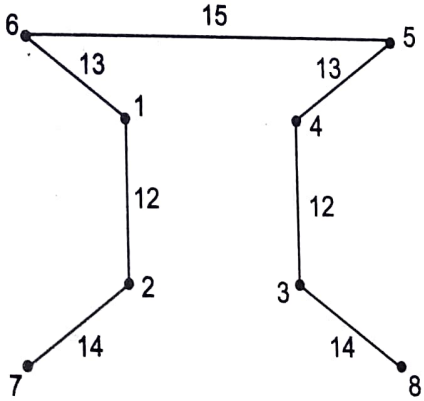
Step 6. Select the next smallest edge (2, 7), because it does not form any cycle with the previously selected edges.



Step 7. Select the next smallest edge (3, 8), because it does not form any cycle with the previously selected edges.



Step 8. We reject the next smallest edges (6, 7) and (5, 8), since each of these edges forms a cycle with the previously selected edges.



Since this tree has 7 (= 8 – 1) edges, this is the required minimal spanning tree. Weight of this minimal spanning tree = 12 + 12 + 13 + 14 + 14 + 15 = 93.

Example 29. Find by Prim's algorithm a minimal spanning tree from the following connected weighted graph:

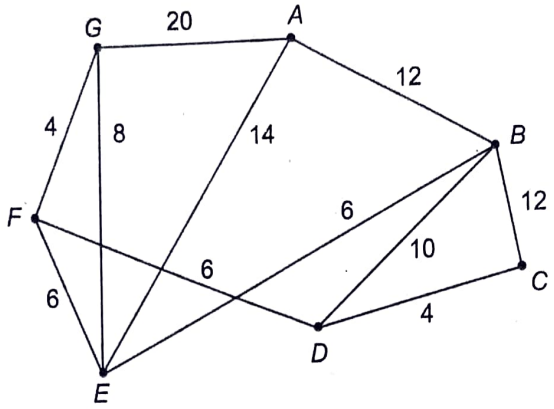


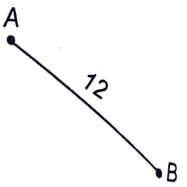
Fig. 16.41

Solution. The given graph is connected weighted graph having 7 vertices, so a minimal spanning tree of this graph has 6 (= 7 – 1) edges.

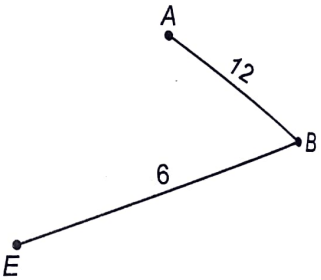
Step 1. The tabulated form of the weights of its edges are given below:

	A	*B*	*C*	*D*	*E*	*F*	*G*
A	–	12	∞	∞	14	∞	20
B	12	–	12	10	6	∞	∞
C	∞	12	–	4	∞	∞	∞
D	∞	10	4	–	∞	6	∞
E	14	6	∞	∞	–	6	8
F	∞	∞	∞	6	6	–	4
G	20	∞	∞	∞	8	4	–

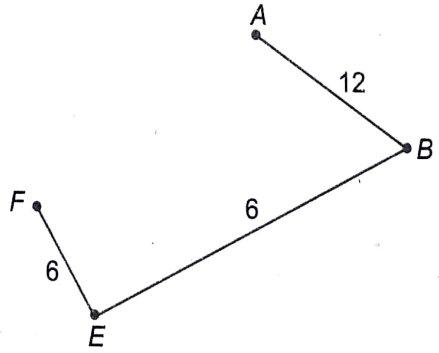
Step 2. Start from the vertex *A*. The smallest entry in *A*-row, *i.e.*, the 1st row of the table is 12 which corresponds to the vertex *B*. Thus the tree *AB* is formed as shown below:



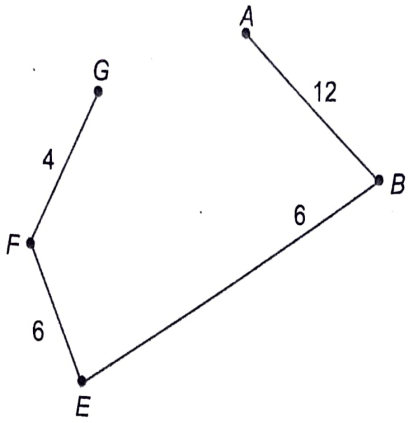
Step 3. Smallest entry in *A*-row and *B*-row, *i.e.*, in 1 st and 2nd rows is 6 which corresponds to the vertex *E*. Thus the tree *ABC* is formed as shown below:



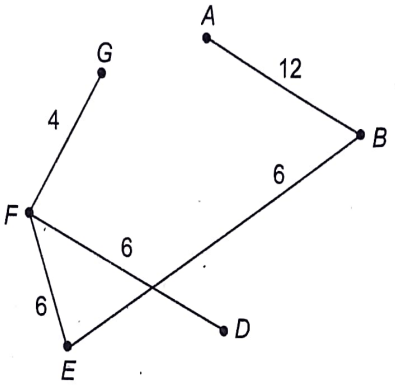
Step 4. Smallest entry in *A*-row, *B*-row and *E*-row, *i.e.*, in 1st, 2nd and 5th rows is 6 which corresponds to the vertex *F* (except *B* and *E* which are already selected). Thus the tree *ABEF* is formed.



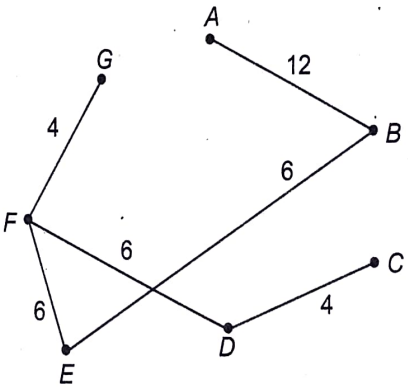
Step 5. Smallest entry in A-row, B-row, E-row and F-row, *i.e.*, in 1st, 2nd, 5th and 6th rows is 4 which corresponds to the vertex G. Hence the tree $ABEFG$ is formed.



Step 6. Smallest entry in A-row, B-row, E-row F-row and G-row is 6 (we are not taking 4 since it corresponds to F or G which are already selected) which corresponds to D in F-row (except B, E and F which are already selected). Thus the following tree is formed:



Step 7. Smallest entry in A-row, B-row, D-row E-row, F-row and G-row is 4 which corresponds to the vertex C in D-row (except the vertices F and G which are already selected). Hence the following tree is formed:



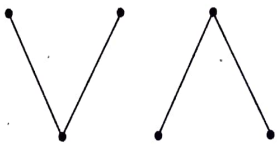
Since this tree has 7 vertices and 6 (= 7 – 1) edges, this is the required minimal spanning tree. Weight of this minimal spanning tree = 12 + 6 + 6 + 4 + 6 + 4 = 38.

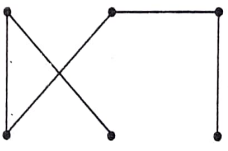
TREE PROBLEM SET

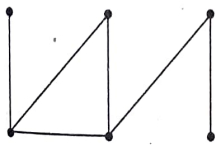
Problem Set 16.1.

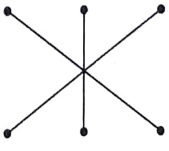
1. Which of the following graphs are tree ?

(*a*) 

(*b*) 

(*c*) 

(*d*) 

(*e*) 

(*f*)