14.1 Introduction

The word tree suggests branching out from a root and never completing a cycle. Trees form one of the most widely used subclass of graphs. This is due to the fact that many of the applications of graph theory, directly or indirectly, involve trees. Tree occurs in situations where many elements are to be organised into some short of hierarchy. In computer science, trees are useful in organising and storing data in a database.

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

14.2. Trees and Their Properties

A tree is a connected acyclic graph *i.e.* a connected graph having no cycle. Its edges are called **branches**. Fig. 14.1. are examples of trees with atmost five vertices. Fig. 14.2. (*a*) and (*b*) are not trees, since they have cycles.

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

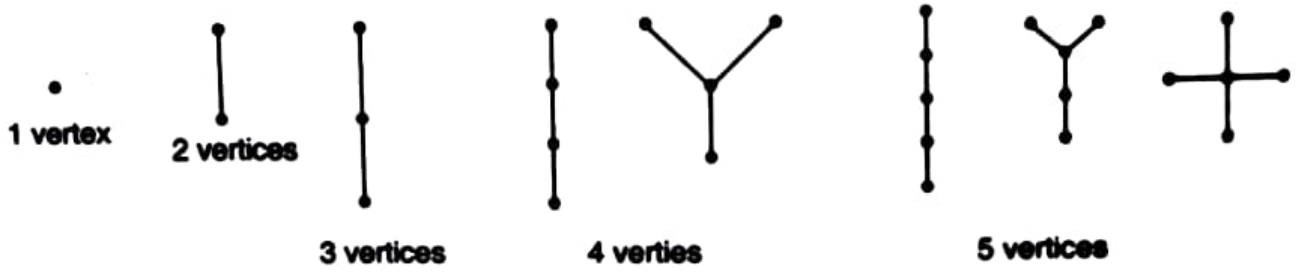


Fig. 14.1

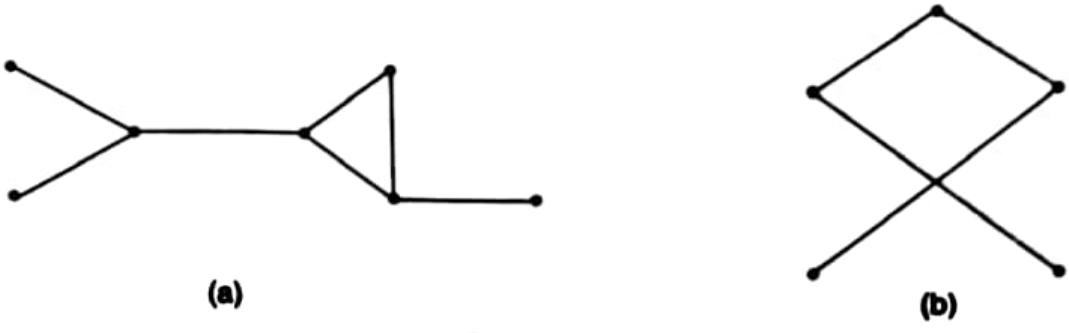


Fig. 14.2

Characterisations

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

Theorem 14.1. There is one and only one path between every pair of vertices *in a tree, T.*

Conversely

Theorem 14.2. If in a graph G there is one and only one path between every pair of vertices, s a tree.

Theorem 14.3. A tree T with n vertices has n-1 edges.

Theorem 14.4. For any positive integer n, if G is a connected graph with n vertices and $n-1$ edges, then G is a tree.

Theorem 14.5 *A graph is a tree if and only if it is minimally connected*

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** (or a **terminal vertex**). If u has either one or two children, then u 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. 14.3.

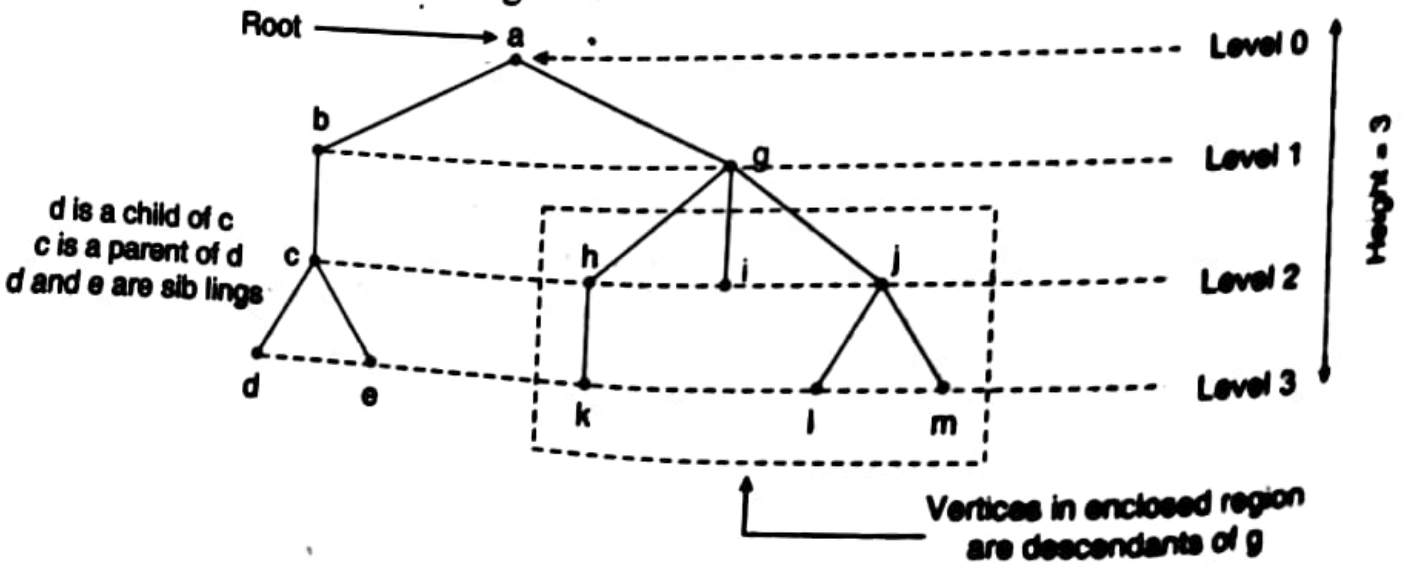


Fig. 14.3

Example 1. Consider the rooted tree in Fig. 14.4.

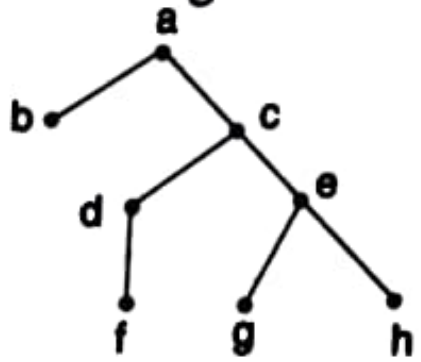


Fig. 14.4

(*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**.

Theorem 14.6. A full m-ary tree with i internal vertex has $n = mi + 1$ vertices.

Theorem 14.7. There are at most m^h leaves in an m-ary tree of height h.

14.3. 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. 14.5

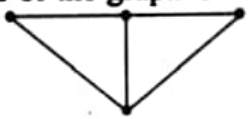


Fig. 14.5

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 ($^5C_2$) ways, except that two of the ways lead to disconnected graphs. Thus there are eight spanning trees as shown in Fig. 14.6.

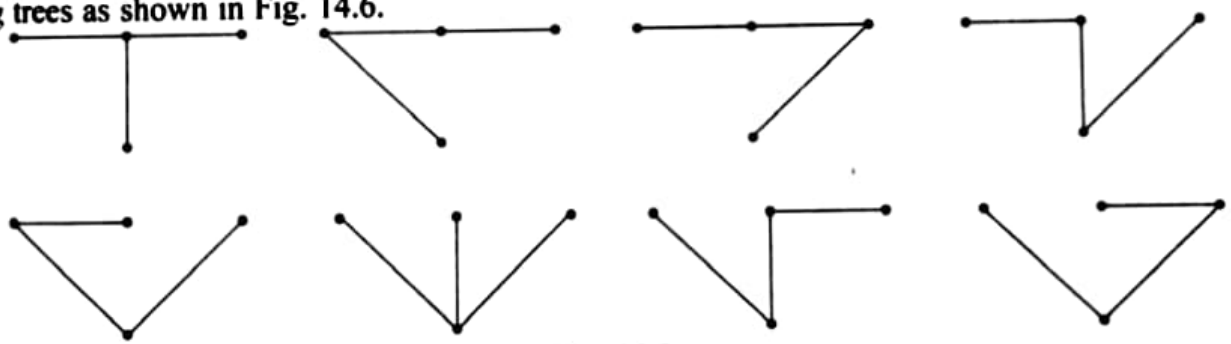


Fig. 14.6

Theorem 14.8. A simple graph G has a spanning tree if and only if G is connected.

Example 6(*i*). Find all spanning tree of the graph G shown in Fig. 14.7

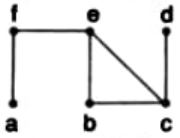


Fig. 14.7

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.

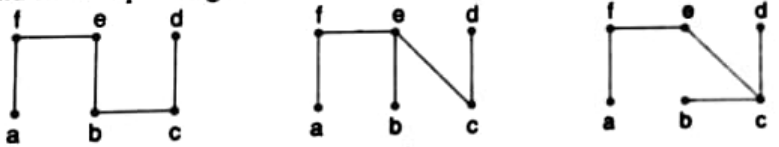
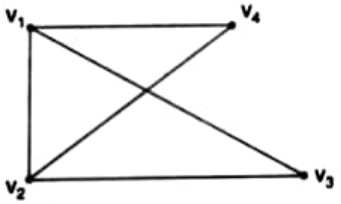


Fig. 14.8

[Note that a spanning tree of a graph need not be unique]

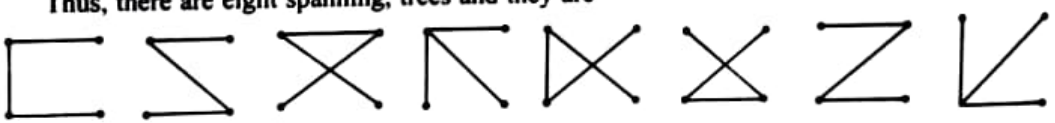
In general, if G is a connected graph with n vertices and m edges, spanning tree of G must have $n - 1$ edges. Hence, the number of edges to be deleted from G before a spanning tree is obtained must be $m - (n - 1) = m - n + 1$.

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 = $m - n + 1 = 5 - 4 + 1 = 2$.

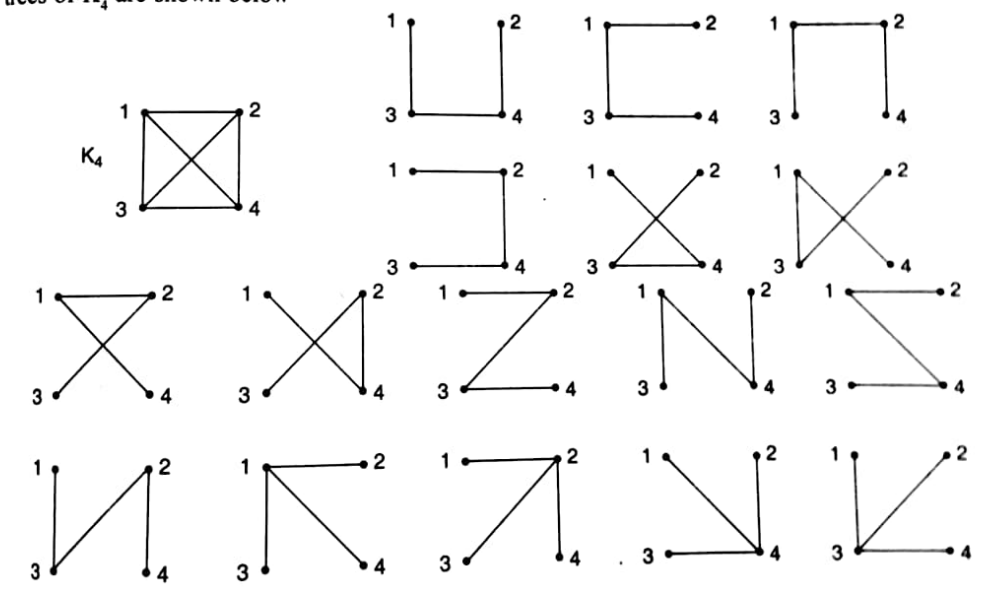
Thus, there are eight spanning, trees and they are



Caylay's Theorem 14.9. 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

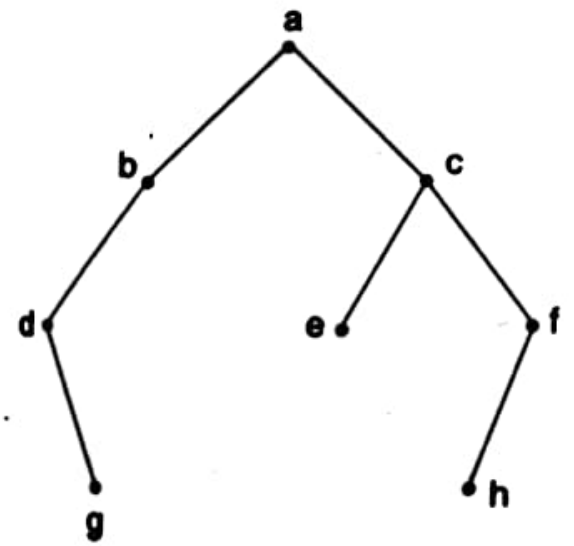


14.4. Binary Tree

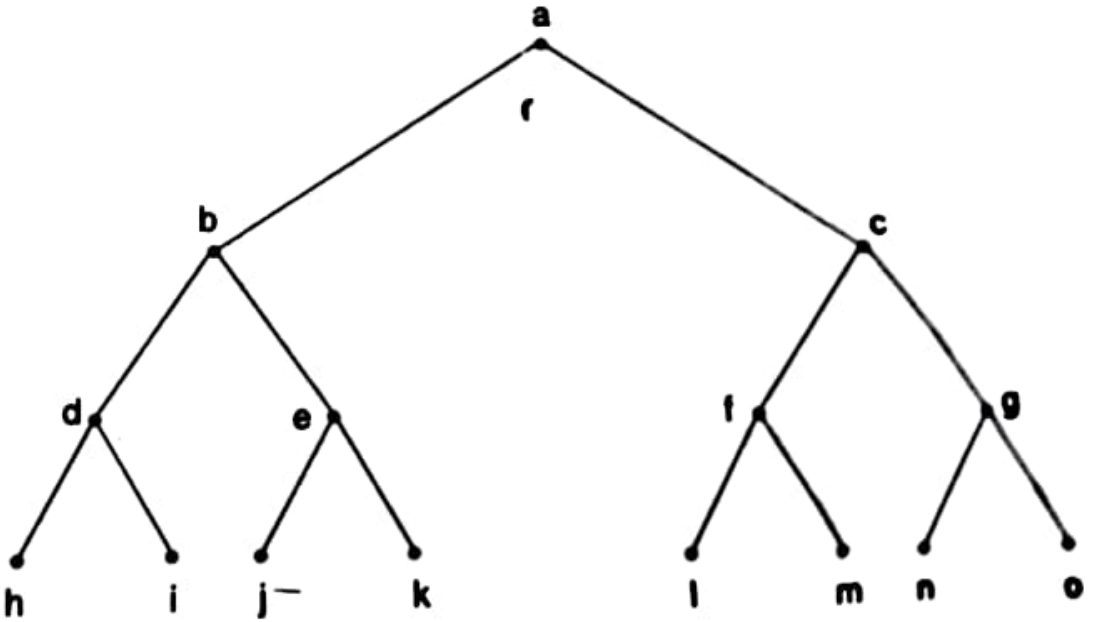
A binary tree is a rooted tree in which each vertex has atmost two children. Each child in a binary tree is designated either a **left child** or a **right child** (not both), and an internal vertex has at most one left and one right child. A **full binary** is a tree in which each internal vertex has 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. 14.17 (*a*) is a binary tree and Fig. 14.17 (*b*) is a full binary tree since each of its internal vertices has two children.



(a)



(b)

Fig. 14.17

Example 17. What are the left and right children of b shown in Fig. 14.18? What are the left and right subtrees of a?

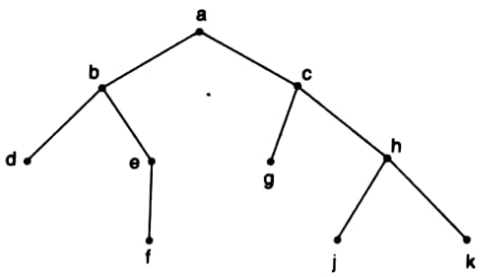


Fig. 14.18

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. 14.19 (a) and (b) respectively.

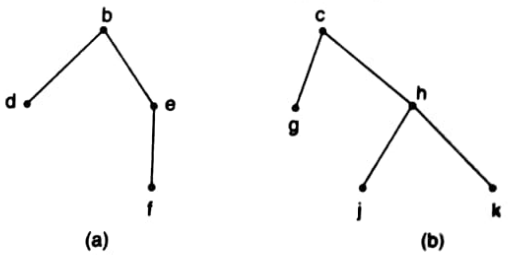


Fig. 14.19

Properties of Binary Trees

Theorem 14.9 The number of vertices in a full binary tree is always odd.

Theorem 14.10. In any binary tree T on n vertices, the number of pendent vertices is equal to $(n + 1)/2$

Theorem 14.11 The number of internal vertices in a binary tree is one less than the number of pendant vertices.

Theorem 14.12 Prove that the maximum number of vertices on level n of a binary tree is 2^n where $n \geq 0$.

Theorem 14.13 Prove that the maximum number of vertices in a binary tree of height h is $2^{h+1} - 1$, $h \geq 0$.

Theorem 14.15. If T is full binary tree with i internal vertices, then T has $i + 1$ terminal vertices and $2i + 1$ total vertices.

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. 14.20. 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 $2^{l+1} - 1$.

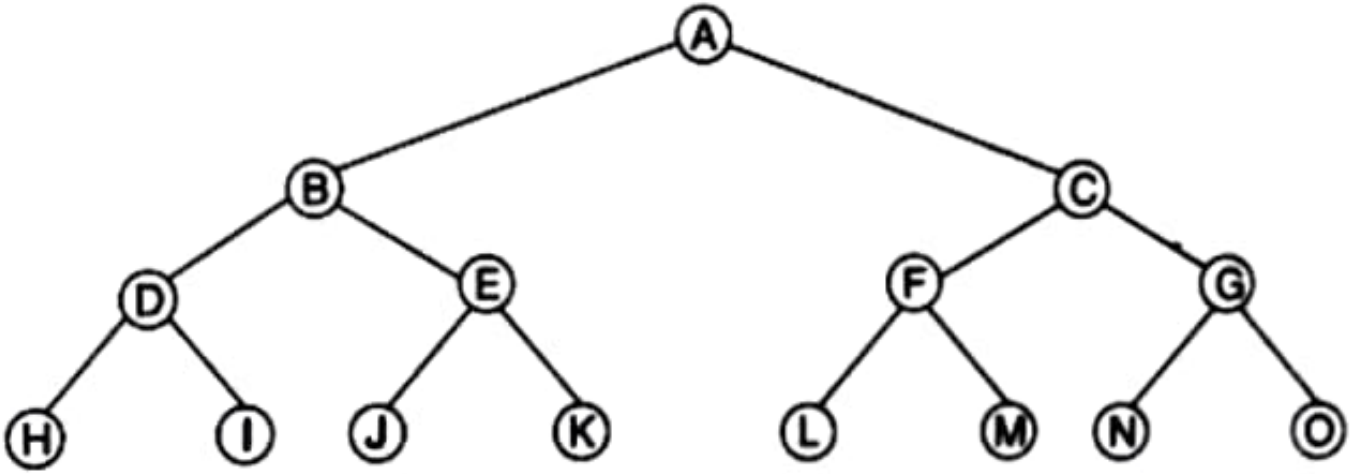


Fig. 14.20. A complete binary tree

14.5. 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*.

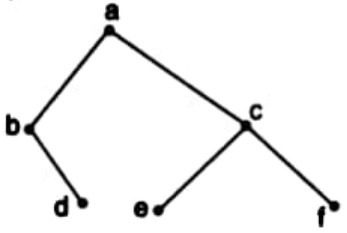


Fig. 14.21

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.**

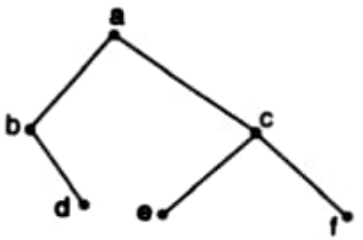


Fig. 14.21

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. 14.22 using this order of traversal:

Fig. 14.22. 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 construction of a unique binary tree

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. 14.23.

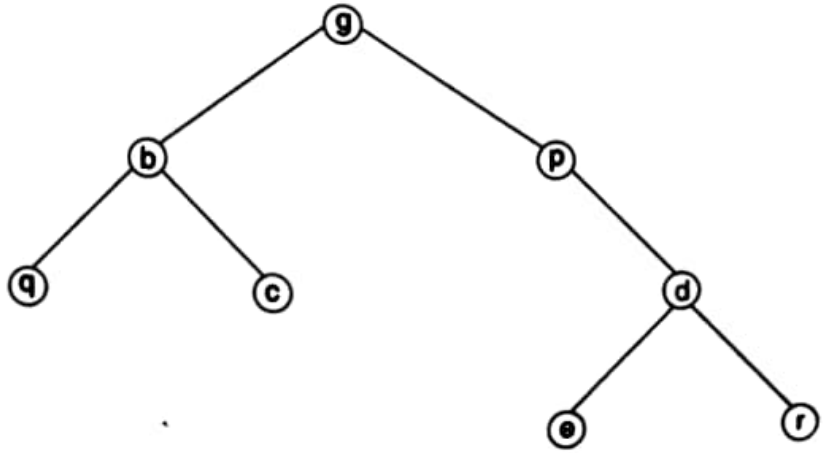


Fig. 14.23

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. 14.24.

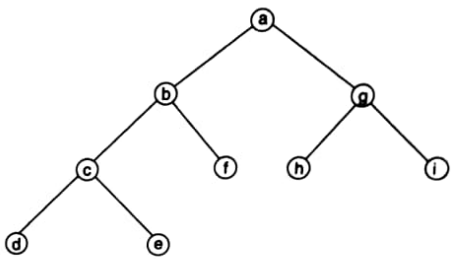


Fig. 14.24

Example 21. Determine the value of the expression represented in a binary tree shown in Fig. 14.28.

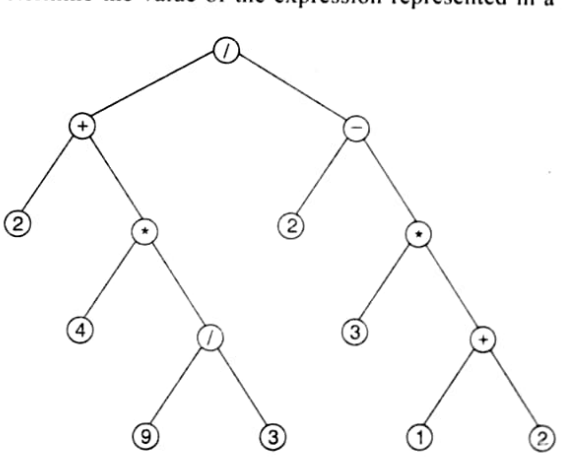


Fig. 14.28

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.

Representation of Algebraic Structure by Binary Trees

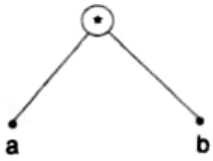
Binary trees are used to represent algebraic expressions, the vertices of the tree are labled with the numbers, veriables, or operations that make up the expression. The leaves of the tree can be labled with numbers or variables. Operations such as addition, subtraction, multiplication, division, or 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*)

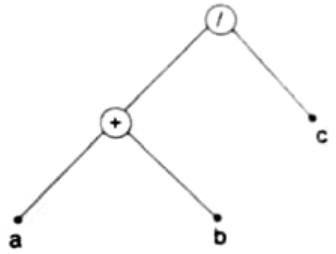


Fig. 14.25

(*iii*)

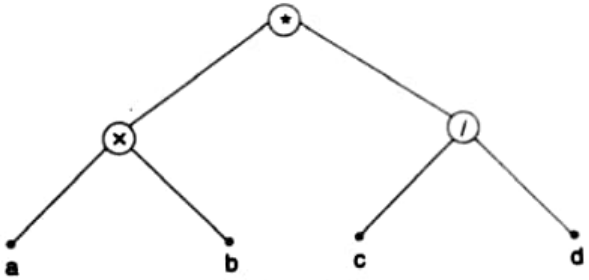


Fig. 14.26

(*iv*)

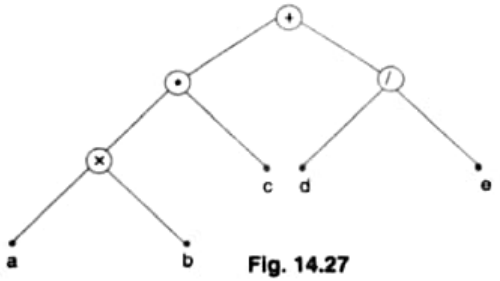


Fig. 14.27

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 postion.

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.

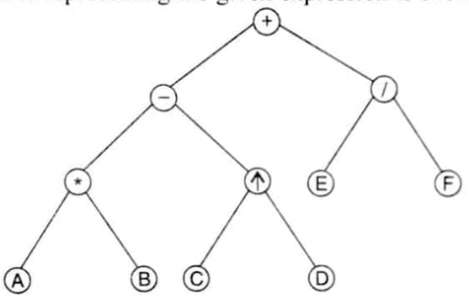


Fig. 14.29

Prefix : + − * AB ↑ CD / EF

Postifix : AB * CD ↑ − 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) postifix 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 Fxy 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. $823* - 2 \uparrow 63 / +$
2. $86 - 2 \uparrow 63 / +$ replacing $23 *$ by $2 * 3 = 6$
3. $22 \uparrow 63 / +$ replacing $86 -$ by $8 - 6 = 2$
4. $463 /$ replacing $2 \uparrow 2$ 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. 14.30. 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.

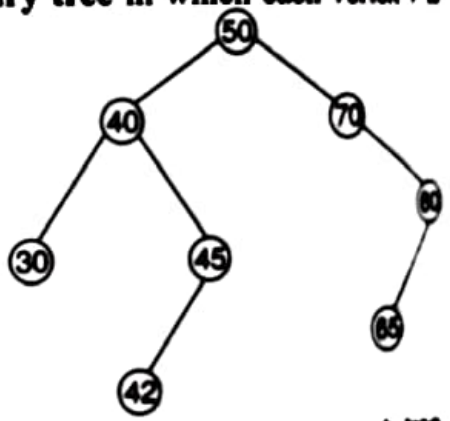


Fig. 14.30 A binary search tree

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 20. 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. 14.31 shows the binary search tree.

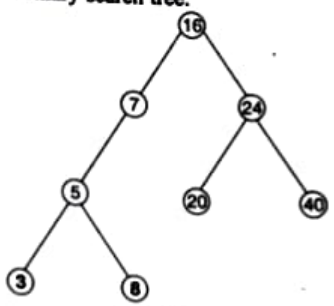


Fig. 14.31

(*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. 14.32 shows the binary search tree.

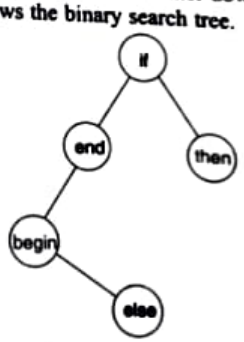


Fig. 14.32

Algorithms for Constructing Spanning Trees

An algorithm for finding a spanning tree based on the proof of above theorem would not be very efficient, it would involve the time-consuming process of finding cycles. 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

In this algorithm a rooted tree will be constructed, and the underlying undirected graph of this rooted forms the spanning tree. The idea of BFS is to visit all vertices on a given level before going into the next level.

Procedure. 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. The new vertices added at this stage become the vertices at level 1 in the spanning tree, arbitrarily order them. 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. Arbitrarily order the children of each vertex at level 1. This produces the vertices at level 2 in the tree. Continue the same procedure until all the vertices in the tree have been added. The procedure ends, since there are only a finite number of edges in the graph. A spanning tree is produced since we have produced a tree without cycle containing every vertex of the graph.

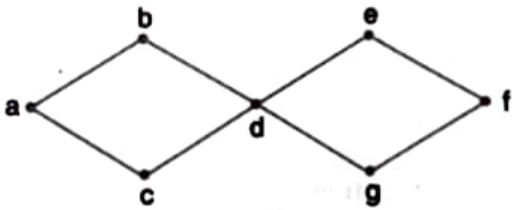


Fig. 14.9

Example 9. Use BFS algorithm to find a spanning tree of graph G of Fig. 14.9.

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.

(*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. 14.10

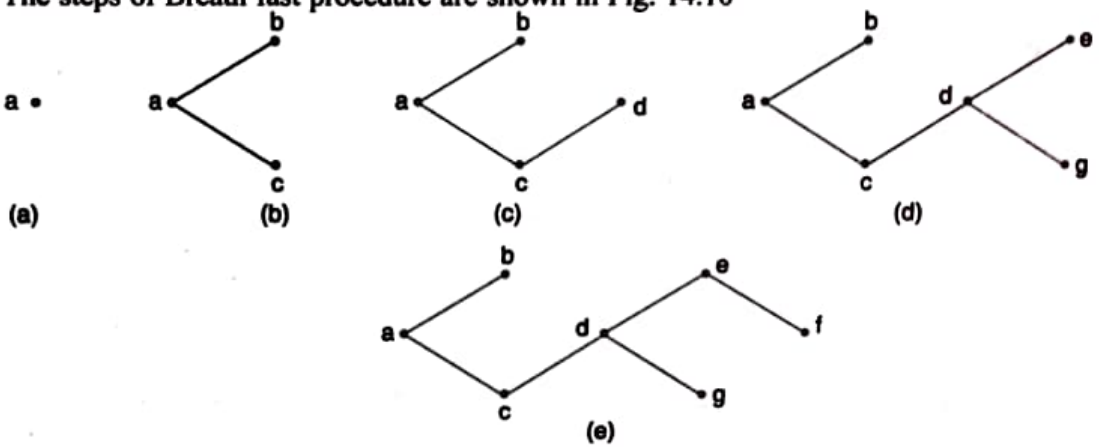


Fig. 14.10

Hence Fig.14.10. (*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. Arbitrarily choose a vertex from the vertices of the graph and designate it as the root. 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. 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. If this can not be done, move back another vertex in the path, that is two vertices back in the path, and repeat. 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. This process ends since the graph has a finite number of edges and is connected. A spanning tree is produced.

Example 10. Find a spanning tree of the graph of Fig. 14.11 using Depth–first search algorithm.

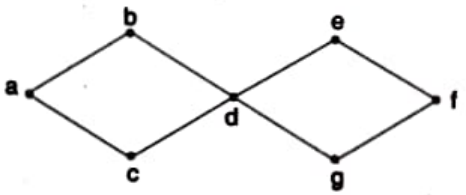


Fig. 14.11

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. 14.12.

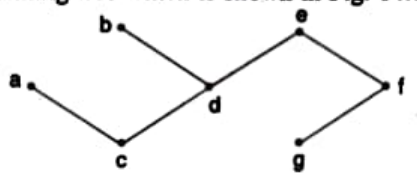


Fig. 14.12

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.

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. 14.13 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. 14.13. 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.

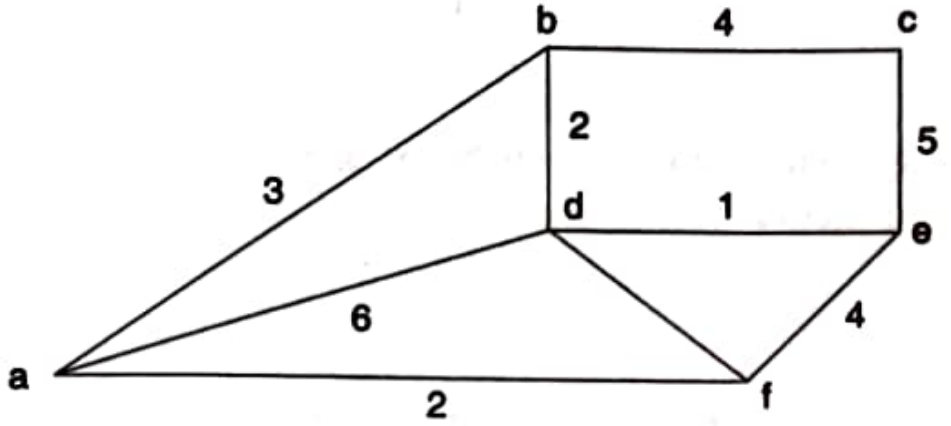


Fig. 14.13

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.

Example 12. Show how Kruskal's algorithm find a minimal spanning tree for the graph of Fig. 14.14.

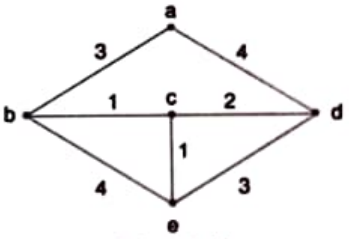


Fig. 14.14

Solution:

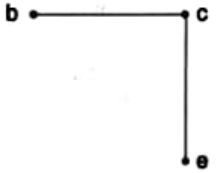
Step 1 : List the edges in non-decreasing order of their weights, as in Table 14.1

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

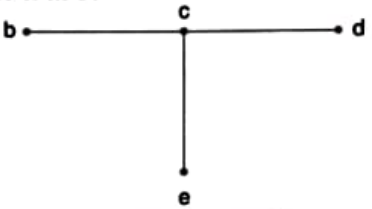
Table 14.1

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

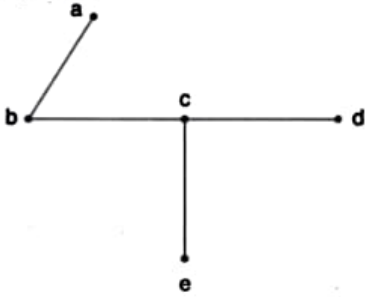
Step 3. Select an edge with the next smallest weight (c, e) since it does not form circuit with the existing edges in T, so include it in T.



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.

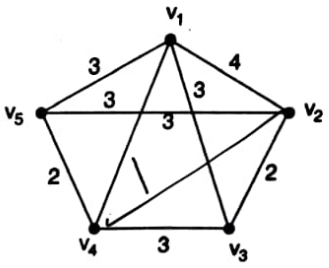
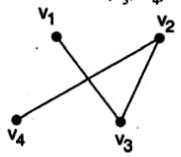
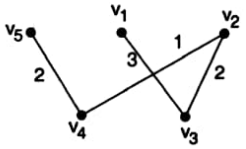


Fig. 14.16

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

$3 + 2 + 1 + 2 = 8$

Huffman's Algorithm

For a given sequence of letters:

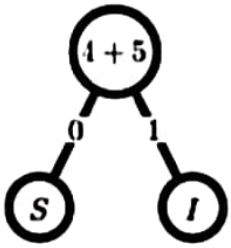
1. Count the frequency of occurrance for the letters in the sequence.
2. Sort the frequencies into increasing order
3. **Build the Huffman tree:**
   - Choose the two smallest values, make a (sub) binary with these values.
   - Accumulate the sum of these values
   - Replace the sum in place of original two smallest values and repeat from 1.
   - *Construction of tree is a bottom up insertion of sub-trees at each iteration.*

(2)

o build the Huffman tree, we sort the frequencies into increasi
rder (4, 5, 7, 8, 12, 29).

S	4
I	5
O	7
T	8
P	12
E	29

Then we choose the two smallest values S and I (4 and 5), a
onstruct a binary tree with labeled edges:

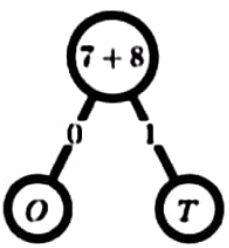


(3)

Next, we replace the two smallest values S (4) and I (5) with th
sum, getting a new sequence, (7, 8, 9, 12, 29).

O	7
T	8
SI	9
P	12
E	29

We again take the two smallest values, O and T, and construct
abeled binary tree:

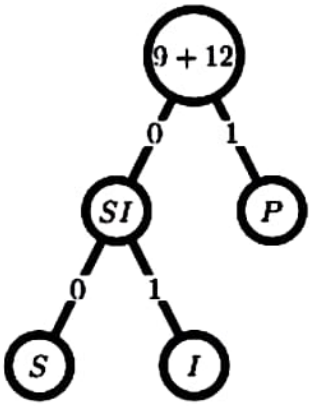


(4)

We now have the frequencies (15, 9, 12, 29) which must be sorted into (9, 12, 15, 29)

$$\begin{array}{cc} SI & 9 \\ P & 12 \\ OT & 15 \\ E & 29 \end{array}$$

and the two lowest, which are IS (9) and P (12), are selected once again:

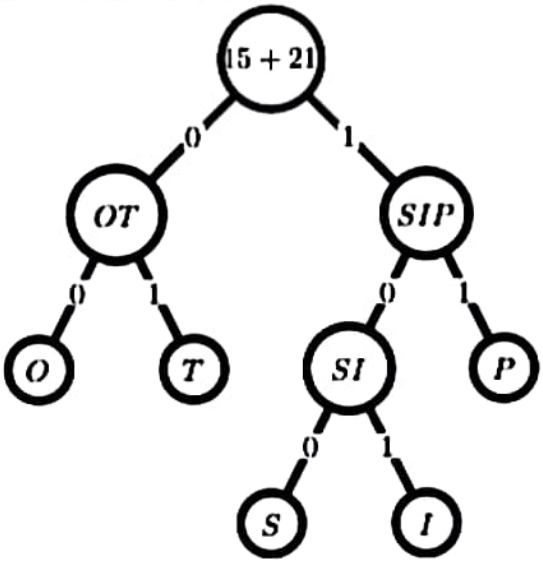


(5)

We now have the frequencies (21,15, 29) which must be sorted into (15, 21, 29)

$$\begin{array}{cc} OT & 15 \\ SIP & 21 \\ E & 29 \end{array}$$

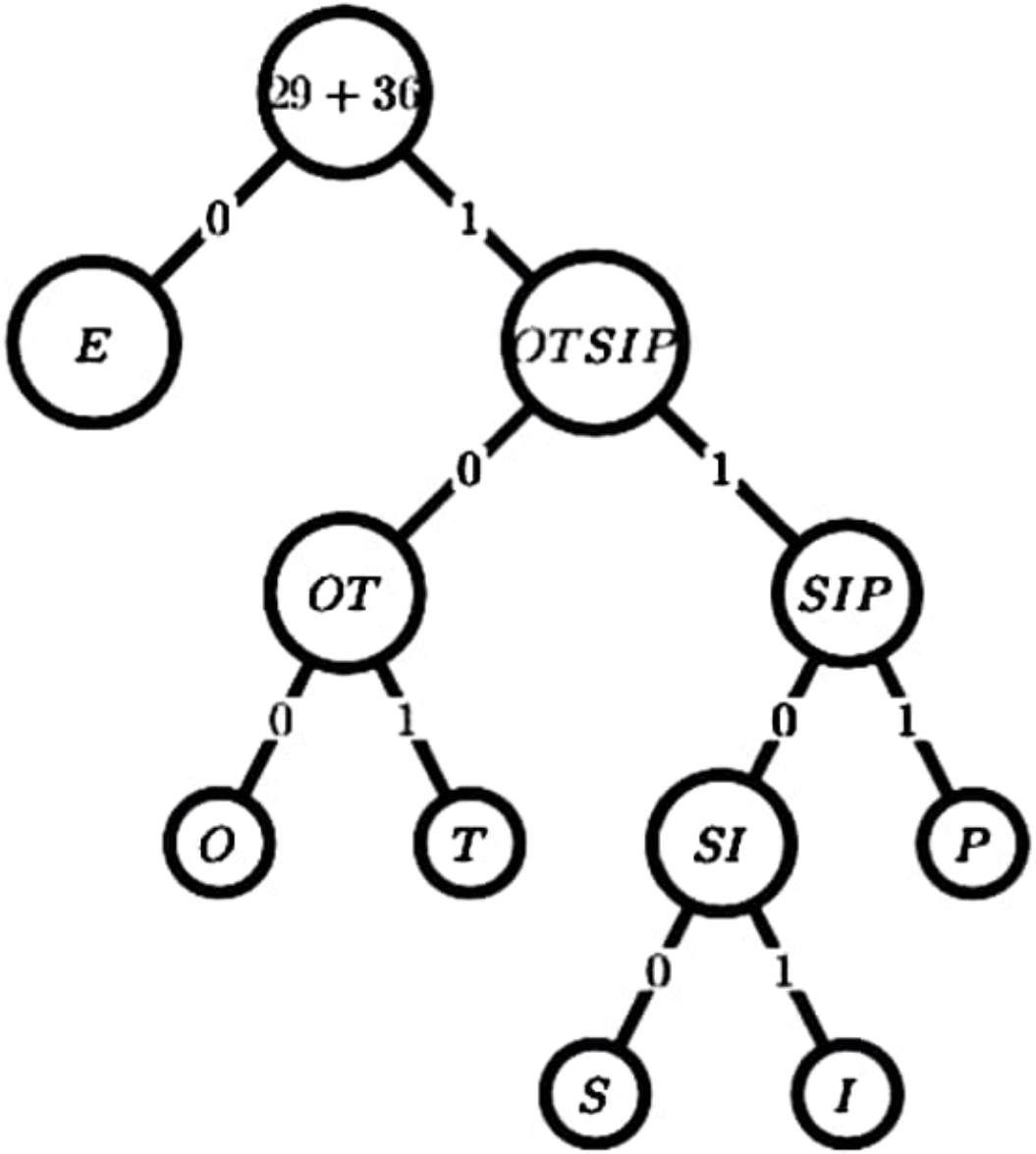
Now, we combine the two lowest which are OT (15) and ISP (21):



(5): Final Tree

The two remaining frequencies, 36 and 29, are now combined into the final tree.

E	29
$OTSIP$	36



GAME TREES

Trees are used in the analysis of two person games like chess, checkers and tic-tac-toe. In these games players alternate moves and each player tries to outsmart the opponent. The trees are used in the development of many computer programs that allow people to play against computers or even computers against computers. Here we discuss trees that are used to develop game playing strategies.

Example 17. *To understand the game trees in a better way. Let us consider a game tree as shown in fig. The values shown are obtained from an evaluation function.*

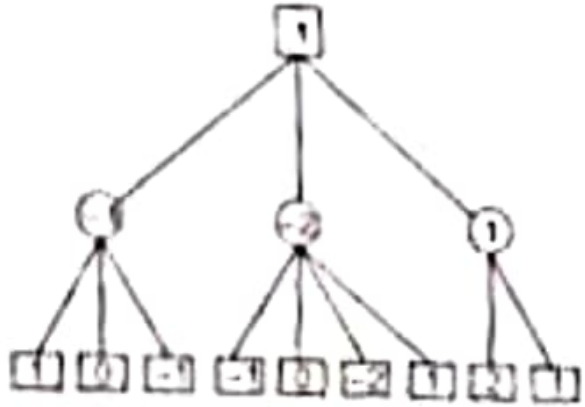


Fig. 29 (a)

The game tree is expanded by two levels. The first player is represented by a box and the second player is represented by a circle. A path represents a sequence of moves. If a position is shown in a square, it is the first player's move. If a position is shown in a circle, it is the second player's move. A terminal vertex represents the end of the game. In our game tree, if the terminal vertex is a circle, the first player makes the last move and lost the game. If the terminal vertex is a box, the second player lost the game.

Minimax strategy of Game Playing. It is a look ahead strategy. Here one player is called a maximizer (who seeks maximum of its children) and the other is called a minimizer (who seeks minimum of its children).

Both the opponent (the maximizer and minimizer fight it out to see that the opponent gets the minimum benefit while they get the maximum benefit.

A static evaluation function E is constructed that assigns each possible game position the value E(P) of the position to the first player. After the vertices at the lowest level are assigned values by using the function E, the minimax strategy can be applied to generate the values of other vertices.

It is assumed that the static evaluation function returns a value from −10 to +10, where a value of 0 indicates a win for the maximizer and a value of −10 a win for the minimizer. A value of 0 indicates a tie or draw. When using a game tree we should use a depth first search. If the game tree is so large that it is not feasible to reach a terminal vertex, the level to which depth first search has carried out can be limited. The search is said to be a K-level search if we limit the search to K levels below the given vertex.

Example 18. *Apply the minimax to find the value of the root in the following game tree using a two level, depth first minimax search. The values shown are obtained from an evaluation function.*

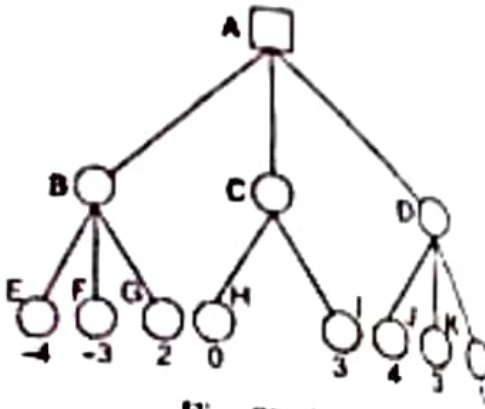


Fig. 29 (b)

Sol. Let us assume that the maximizer will have to play first followed by the minimizer. Before, the maximizer moves to B, C, or D, he will have to think which move would be highly beneficial to him.

(i) If A moves to B, it is the minimizer who will have to play next. The minimizer always tries to give the minimum benefit to the other player and hence he will move to E. Thus, the value − 4 is backed up at B.

(ii) If A moves to C, then the minimizer will move to H and the value 0 is backed up at C.

(iii) If A moves to D, then the minimizer will move to L and the value 1 is backed up at D.

The maximizer will now have to choose between B, C or D with the values − 4, 0 and 1. The maximizer will choose node D because it gives him a value of 1, that is better than − 4 and 0.

The tree with the backed up values is shown in Fig.

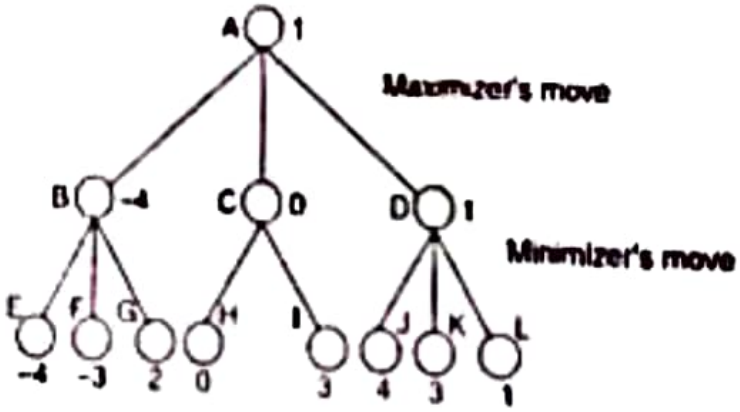


Fig. 29 (c)

Now, assume that the minimizer will have to play first followed by the maximizer. In this case, the tree with the backed up values is shown in Fig.

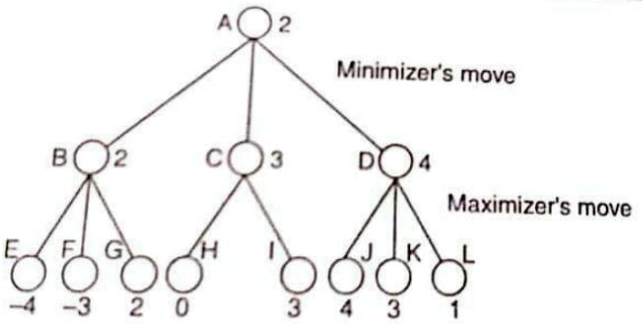

A 2
Minimizer's move
B 2
C 3
D 4
Maximizer's move
E F G H I J K L
−4 −3 2 0 3 4 3 1

Fig. 29 (*d*)