UNIT-IV

TREES:

Definition of trees:

A tree is defined as a finite set of one or more nodes such that

(i) there is a specially designated node called the root and

(ii) the rest of the nodes could be partitioned into t disjoint sets (t ≥ 0) each set representing a tree Ti , i = 1, 2, . . . t known as subtree of the tree.

A node in the definition of the tree represents an item of information, and the links between the nodes termed as branches, represent an association between the items of information.

Figure below illustrates a tree. The definition of the tree emphasizes on the aspect of (i) connectedness and (ii) absence of closed loops or what are termed cycles.

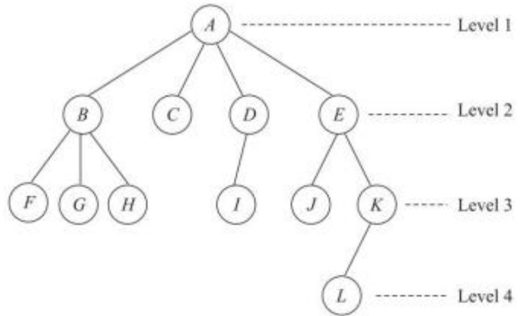


Fig. An example tree

Basic terminologies of trees :

There are several basic terminologies associated with the tree. The specially designated node called **root node**.

The number of subtrees of a node is known as the **degree** of the node. Nodes that have zero degree are called **leaf nodes** or **terminal nodes**.

The rest of them are called as **non-terminal** nodes. These nodes which hang from branches emanating from a node are known a **children** and the node from which the branches emanate is known as the **parent node**.

Children of the same parent node are referred to as **siblings**. The **ancestors** of a given node are those nodes that occur on the path from the root to the given node.

The **degree of a tree** is the maximum degree of the node in the tree. The **level** of a node is defined by letting the root to occupy level 1 . The rest of the nodes occupy various levels depending on their association.

Thus if a parent node occupies level i, its children should occupy level i+1. This renders the tree to have a hierarchical structure with root occupying the top most level of 1.

The **height or depth** of a tree is defined to be the maximum level of any node in the tree.

Depth of a node to be the length of the longest path from the root node to that node, which yields the relation, **depth of the tree = height of the tree – 1** .

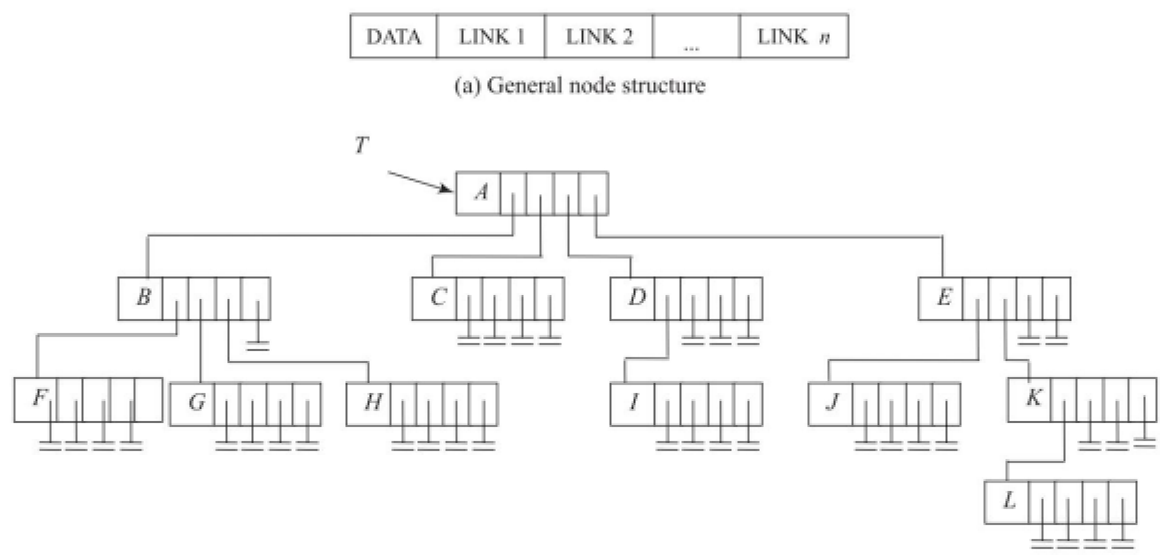
A forest is a set of zero or more disjoint trees. The removal of the root node from a tree results in a forest (of its subtrees!).

Representation of Trees:

A common representation of a tree to suit its storage in the memory of a computer, is a **list**. The tree of above Fig. could be represented in its list form as (A (B(F,G,H), C, D(I), E(J,K(L)))).

The root node comes first followed by the list of subtrees of the node. This is repeated for each subtree in the tree.

This list form of a tree, is a naïve representation of the tree as a linked list. The node structure of the linked list is shown in Fig. (a).



(a) General node structure

(b) Linked list representation of the tree shown in Fig. 8.1

The DATA field of the node stores the information content of the tree node. A fixed set of LINK fields accommodate the pointers to the child nodes of the given node.

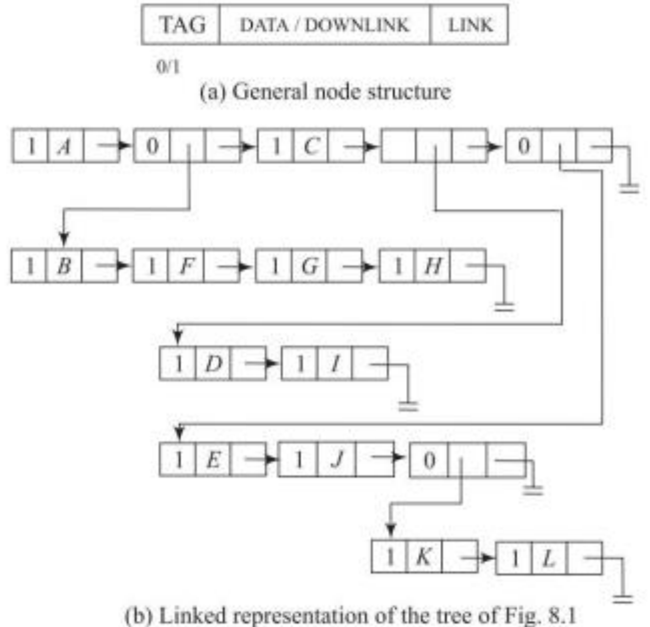
In fact the maximum number of links the node would require is equal to the degree of the tree. The linked representation of the tree illustrated in Fig. (b).

Disadvantage: wastage of space by way of null pointers.

An **alternative representation** would be to use a node structure as shown in Fig.below. Here TAG =1 indicates that the next field (DATA / DOWN LINK) is occupied by data (DATA) and TAG = 0 indicates that the same is used to hold a link (DOWN LINK).

The node structure of the linked list holds a DOWNLINK whenever it encounters a child node which gives rise to a subtree. Thus the root node A has four child nodes, three of which viz., B, D and E give rise to subtrees.

Note the DOWN LINK active fields of the nodes in these cases with TAG set to 0. In contrast, observe the linked list node corresponding to C which has no subtree. The DATA field records C with TAG set to 1.



(a) General node structure

(b) Linked representation of the tree of Fig. 8.1

BINARY TREES:

Basic terminologies:

A binary tree has the characteristic of all nodes having at most two branches, that is, all nodes have a **degree of at most 2**. A binary tree can therefore be empty or consist of a root node and two disjointed binary trees termed **left subtree** and **right subtree**. Figure below illustrates a binary tree.

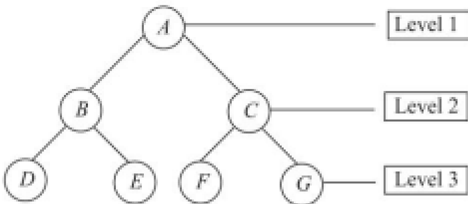


Fig. An example of binary tree

The distinction between trees and binary trees

The distinction between trees and binary trees are while a binary tree can be empty with zero nodes, a tree can never be empty.

Again while the ordering of the subtrees in a tree is immaterial, in a binary tree the distinction of left and right subtrees are very clearly maintained.

Properties:

Some important observations regarding binary trees.

(i) The maximum number of nodes on level i of a binary tree is 2i–1, i ≥ 1.

(ii) The maximum number of nodes in a binary tree of height h is 2h – 1, h ≥ 1.

(iii) For any non-empty binary tree, if to is the number of terminal nodes and t2 is the number of nodes of degree 2, then to = t2 + 1.

These observations could be easily verified on the binary tree shown in above Fig. The maximum number of nodes on level 3 is 232 = 22 = 4.

Also with the height of the binary tree being 3, the maximum number of nodes = 23 1 = 7. Again to = 4 and t2 = 3 which yields to = t2 + 1.4

Types of binary trees:

Full binary tree: A binary tree of height h which has all its permissible maximum number of nodes viz., 2^h-1 intact is known as a full binary tree of height h. Figure (a) illustrates a full binary tree of height 4. Note the specific method of numbering the nodes.

complete binary tree: A binary tree with n' nodes and height h is complete if its nodes correspond to the nodes which are numbered 1 to n (n'≤ n) in a full binary tree of height h.

In other words, a complete binary tree is one in which its nodes follow a sequential numbering that increments from a left-to-right and top-to-bottom fashion. A full binary tree is therefore a special case of a complete binary tree.

Also, the height of a complete binary tree with n elements has a height h given by
$h = \lceil \log2 (n + 1) \rceil$.

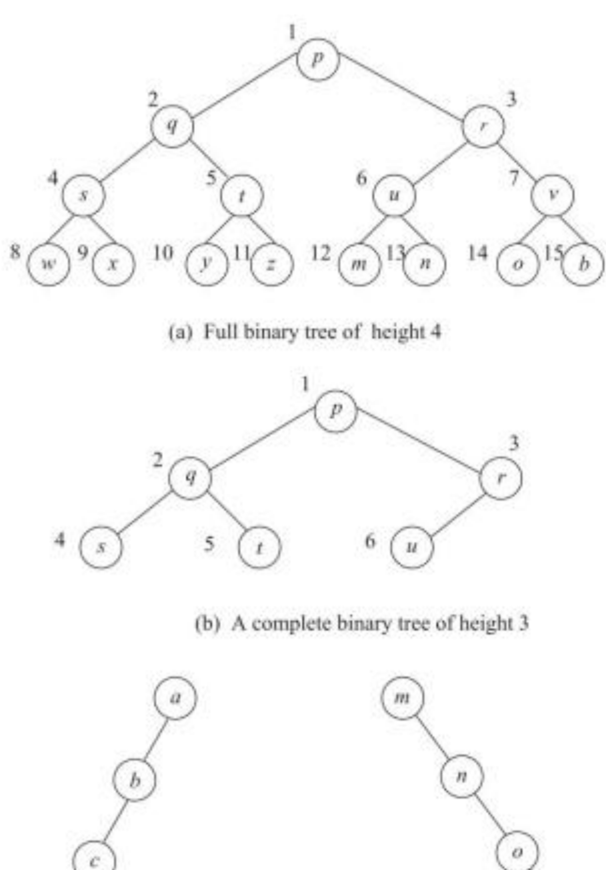
A complete binary tree obeys the following properties with regard to its node numbering: (i) If a parent node has a number i then its left child has the number 2i (2i≤n). If 2i > n then i has no left child.

(ii) If a parent node has a number i, then its right child has the number 2i + 1 (2i + 1 ≤ n). If 2i + 1 > n then i has no right child.

(iii) If a child node (left or right) has a number i then the parent node has the number
$\lfloor i/2 \rfloor$ if i ≠ 1. If i = 1 then i is the root and hence has no parent.

Figure (b) illustrates an example complete binary tree.

Skewed binary tree: A binary tree which is dominated solely by left child nodes or right child nodes is called a skewed binary tree or more specifically left skewed binary tree or right skewed binary tree respectively. Figure (c) illustrates examples of skewed binary trees.



(a) Full binary tree of height 4

(b) A complete binary tree of height 3

(c) Skewed binary tree

<u>Representation of Binary Trees:</u>

A binary tree could be represented using a **sequential data structure** (arrays) as well as **linked data structure**.

i) <u>Array representation of binary trees:</u>

To represent the binary tree as an array, the sequential numbering system emphasized by a complete binary tree is used. Consider the binary tree shown in Fig. (a) below. The array representation is as shown in Fig. (b) below.

The association of numbers pertaining to parent and left/ right child nodes makes it convenient to access the appropriate cells of the array. However, the missing nodes in the binary tree and hence the corresponding array locations, are left empty in the array.

This obviously leads to a lot of wastage of space. However, the array representation ideally suits a full binary tree due to its non-wastage of space.

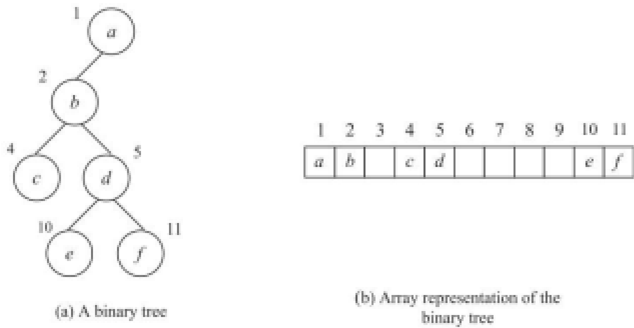


Fig. Array representation of a binary tree

ii) Linked representation of binary trees:

The linked representation of a binary tree has the node structure shown in Fig. (a) below. Here, the node, besides the DATA field, needs two pointers LCHILD and RCHILD to point to the left and right child nodes respectively. The tree is accessed by remembering the pointer to the root node of the tree.

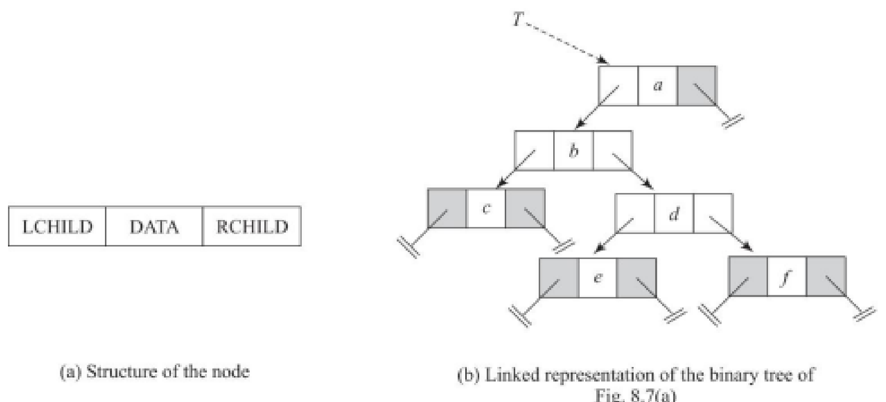


Fig. Linked representation of a binary tree

In the binary tree T shown in Fig. (b), LCHILD (T) refers to the node storing b and RCHILD (LCHILD (T)) refers to the node storing d and so on.

The following are some of the important observations regarding the linked representation of a binary tree:

(i) If a binary tree has n nodes then the number of pointers used in its linked representation is 2 * n.

(ii) The number of null pointers used in the linked representation of a binary tree with n nodes is n + 1.

BINARY TREE TRAVERSALS:

A traversal of a binary tree is where its nodes are visited in a particular but repetitive order, rendering a linear order of the nodes or information represented by them.

A traversal is governed by three actions; viz. **Move left (L), Move Right (R)** and **Process Node (P).** In all, it yields six different combinations of LPR, LRP, PLR, PRL and RLP. Of these, three have emerged significant. They are,

i) LPR - Inorder traversal

ii) LRP - Postorder traversal

iii) PLR - Preorder traversal

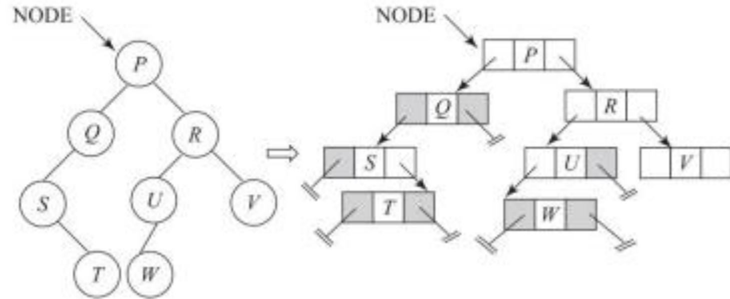


Fig. Binary tree to demonstrate Inorder, Postorder and Preorder traversals

Inorder Traversal:

The traversal keeps moving left in the binary tree until one can move no further, processes the node and moves to the right to continue its traversal again. In the absence of any node to the right, it retracts backwards by a node and continues the traversal.

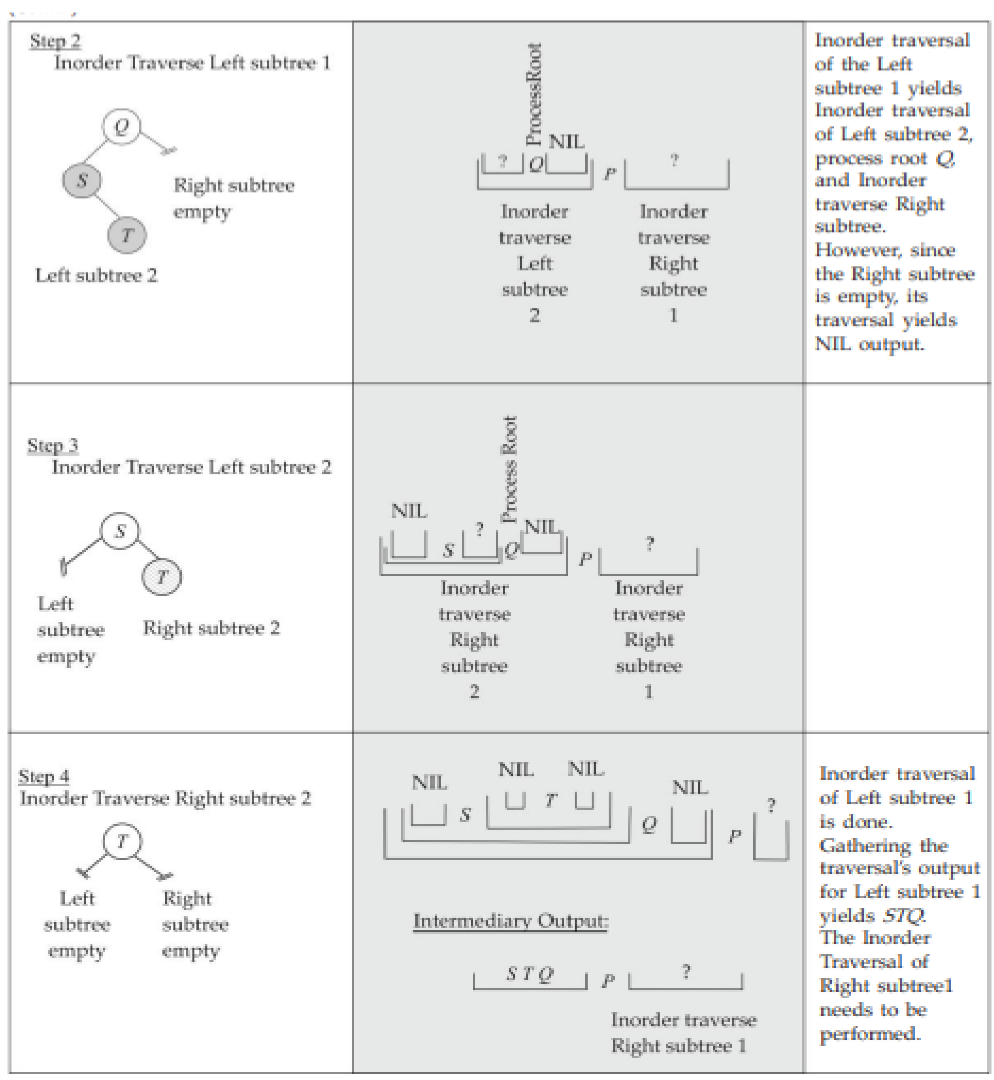
Algorithm : Recursive procedure to perform Inorder traversal of a binary tree

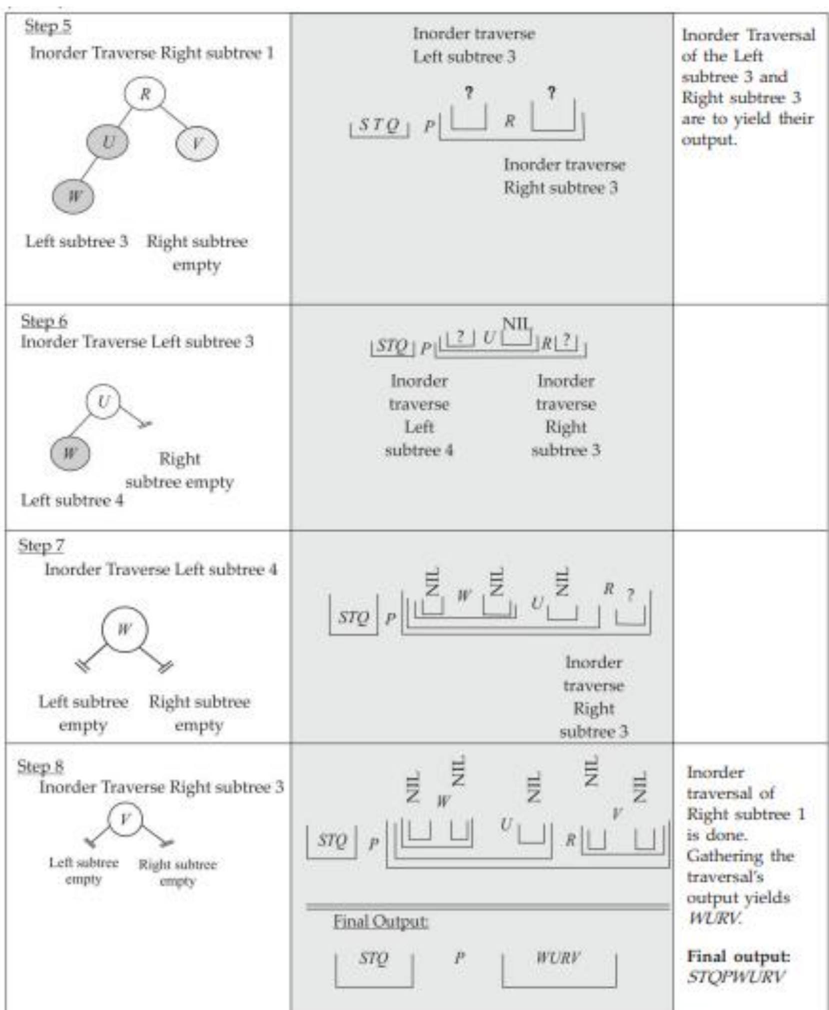
```
procedure INORDER_TRAVERSAL (NODE)
        If NODE ≠ NIL then
        {
        call INORDER_TRAVERSAL (LCHILD(NODE));
        print (DATA (NODE)) ;
        call INORDER_TRAVERSAL (RCHILD(NODE));
        }
end INORDER_TRAVERSAL.
```

Table : Inorder traversal of binary tree shown in above Fig.

Binary Tree	Inorder Traversal Output	Remarks
Step 1 Inorder Traverse Binary Tree Node P Q R S U V T W Left subtree 1 Right subtree 1	Process Root ? P ? Inorder traverse Left subtree 1 Inorder traverse Right subtree 1	Inorder traversals of the Left and Right subtrees of the root node are to yield their output.

(*Contd.*)


Step 2
Inorder Traverse Left subtree 1
Q
S
T
Right subtree empty
Left subtree 2
ProcessRoot
NIL
?
Q
P
?
Inorder traverse Left subtree 2
Inorder traverse Right subtree 1
Inorder traversal of the Left subtree 1 yields Inorder traversal of Left subtree 2, process root Q, and Inorder traverse Right subtree.
However, since the Right subtree is empty, its traversal yields NIL output.
Step 3
Inorder Traverse Left subtree 2
S
T
Left subtree empty
Right subtree 2
Process Root
NIL
S
?
Q
NIL
P
?
Inorder traverse Right subtree 2
Inorder traverse Right subtree 1
Step 4
Inorder Traverse Right subtree 2
T
Left subtree empty
Right subtree empty
NIL
S
NIL
T
NIL
Q
NIL
P
?
Intermediary Output:
S T Q
P
?
Inorder traverse Right subtree 1
Inorder traversal of Left subtree 1 is done.
Gathering the traversal's output for Left subtree 1 yields STQ.
The Inorder Traversal of Right subtree1 needs to be performed.



Postorder Traversal:

The traversal proceeds by keeping to the left until it is no further possible, turns right to begin again or if there is no node to the right, processes the node and retraces its direction by one node to continue its traversal.

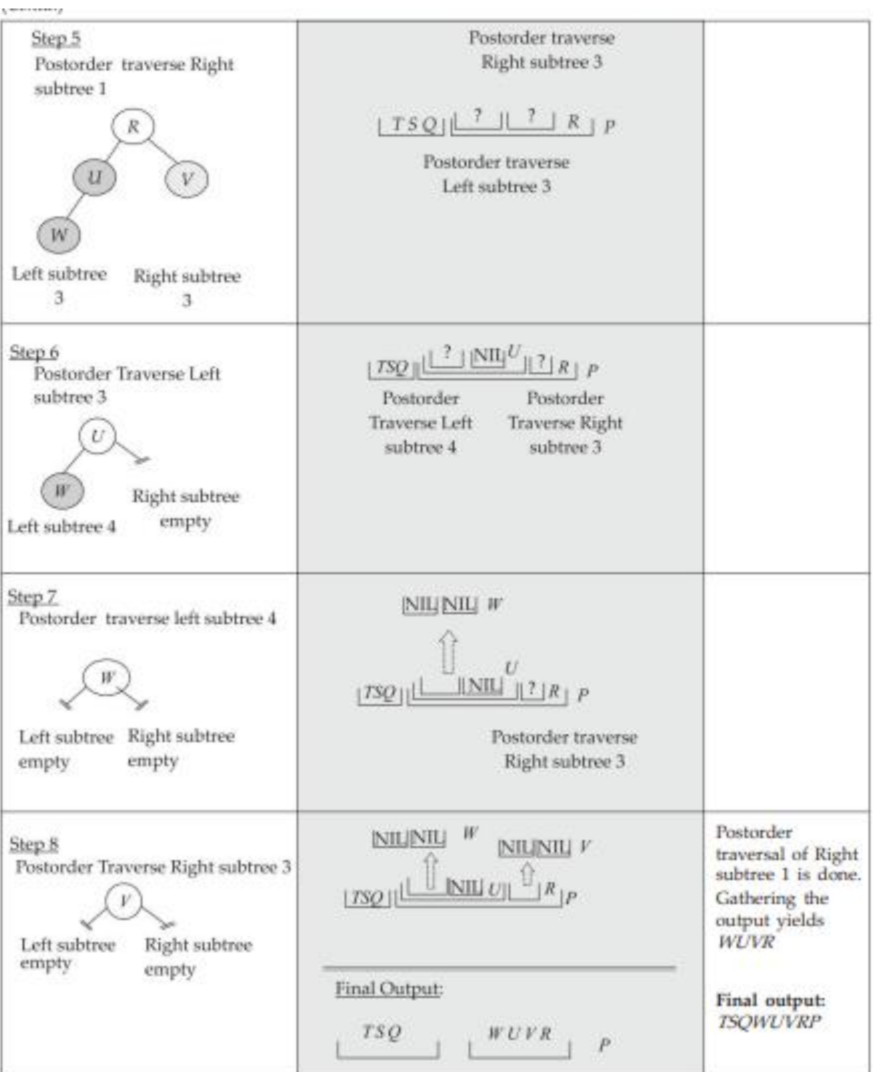
Algorithm: Recursive procedure to perform Postorder traversal of a binary tree

```
procedure POSTORDER_TRAVERSAL (NODE)
        /* NODE refers to the Root node of the binary tree in its first call to the
        procedure. Root node is the starting point of the traversal */
    If NODE ≠ NIL then
    {
    call POSTORDER_TRAVERSAL (LCHILD(NODE));
        / * Postorder traverse the left subtree (L) */
    call POSTORDER_TRAVERSAL (RCHILD(NODE));
        /* Postorder traverse the right subtree (R)*/
     print (DATA (NODE)) ;          /* Process node (P) */
    }
end POSTORDER_TRAVERSAL.
```

Table: Postorder traversal of binary tree shown in above Fig.

Binary Tree	Postorder Traversal Output	Remarks
Step 1 Postorder traverse Binary Tree Node P Q R S U V T W Left subtree 1 Right subtree 1	Process Root ? ? P Postorder traverse Left subtree 1 Postorder traverse Right subtree 1	Postorder Traversal of the Left and Right subtrees of the root is yet to yield their output.
Step 2 Postorder Traverse Left subtree 1 Q S Right subtree empty T Left subtree 2	Postorder traverse Left subtree 2 ? NIL Q ? P Postorder traverse Right subtree 1	
Step 3 Postorder Traverse Left subtree 2 S T Left subtree empty Right subtree 2	NIL ? S NIL Q ? P Postorder traverse Right subtree 2 Postorder traverse Right subtree 1	
Step 4 Postorder traverse Right subtree 2 T Left subtree empty Right subtree empty	NIL NIL NIL T S NIL Q ? P Intermediary Output: T S Q ? P	Postorder traversal of Left subtree 1 is done. Gathering the traversal's output for Left subtree 1 yields TSQ

(Contd.)



Preorder Traversal:

The traversal processes every node as it moves left until it can move no further. Now it turns right to begin again or if there is no node in the right, retracts until it can move right to continue its traversal.

Algorithm: Recursive procedure to perform Preorder traversal of a binary tree procedure

```
PREORDER_TRAVERSAL (NODE)
        /* NODE refers to the Root node of the binary tree in its first call to the
        procedure. Root node is the starting point of the traversal */
  If NODE ≠ NIL then
      {
      print (DATA (NODE)) ; /* Process node (P) */
      call PREORDER_TRAVERSAL (LCHILD(NODE));
          / * Preorder traverse the left subtree (L) */
      call PREORDER_TRAVERSAL (RCHILD(NODE));
          /* Preorder traverse the right subtree (R)*/
          }
```

end PREORDER_TRAVERSAL.

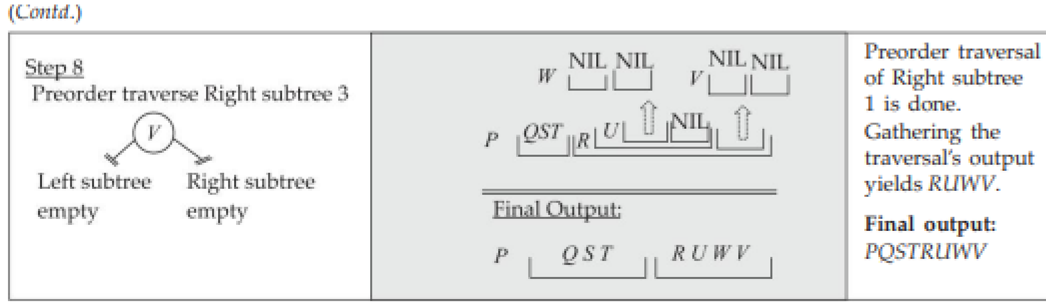
Table: Preorder traversal of binary tree shown in above Fig.

Binary Tree	Preorder Traversal Output	Remarks
Step 1 Preorder traversal of Binary Tree Node → P; Q, R; S, U, V; T, W Left subtree 1 Right subtree 1	Process Root P ? ? Preorder traverse Left subtree 1 / Preorder traverse Right subtree 1	Preorder traversals of the Left subtree 1 and Right subtree 1 to yield their output.
Step 2 Preorder traverse Left subtree 1 Q, S, T Right subtree empty Left subtree 2	Preorder traverse Left subtree 2 P Q ? NIL ? Preorder traverse Right subtree 1	

(Contd.)

Step 3 Preorder traverse Left subtree 2 S, T Left subtree empty Right subtree 2	Preorder traverse Right subtree 2 P Q S NIL ? NIL ? Preorder traverse Right subtree 1	
Step 4 Preorder traverse Right subtree 2 T Left subtree empty Right subtree empty	T NIL NIL P Q S NIL ⇧ NIL ? Intermediary Output: P QST ?	Preorder traversal of Left subtree 1 is done. Gathering the traversal's output yields *QST*
Step 5 Preorder traverse Right subtree 1 R, U, V, W Left subtree 3 Right subtree 3	Preorder traverse Left subtree 3 P QST R ? ? Preorder traverse Right subtree 3	
Step 6 Preorder traverse Left subtree 3 U, W Right subtree empty Left subtree 4	Preorder traverse Left subtree 4 P QST R U ? NIL ? Preorder traverse Right subtree 3	
Step 7 Preorder traverse left subtree 4 W Left subtree empty Right subtree empty	W NIL NIL P QST R U ⇧ NIL ? Preorder traverse Right subtree 3	

(Contd.)



Some significant observations pertaining to the traversals of a binary tree are the following,

(i) Given a preorder traversal of a binary tree, the root node is the first occurring item in the list.

(ii) Given a postorder traversal of a binary tree, the root node is the last occurring item in the list.

(iii) Inorder traversal does not directly reveal the root node of the binary tree.

(iv) An inorder traversal coupled with any one of preorder or post order traversal helps trace back the structure of the binary tree

BINARY SEARCH TREES:

Definition :

A binary search tree T may be an empty binary tree. If non-empty, then for a set S, a binary search tree T satisfies the following norms:

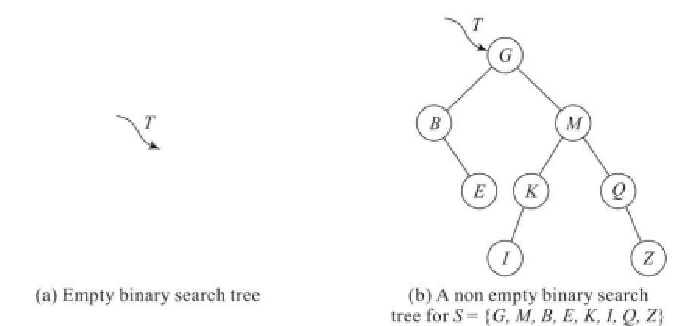
(i) all keys of the binary search tree must be distinct

(ii) all keys in the left subtree of T are less than the root element

(iii) all keys in the right subtree of T are greater than the root element and

(iv) the left and right subtrees of T are also binary search trees.

- The inorder traversal of a binary search tree T yields the elements of the associated set S in the ascending order.



(a) Empty binary search tree

(b) A non empty binary search tree for $S = \{G, M, B, E, K, I, Q, Z\}$

Representation of a binary search tree:

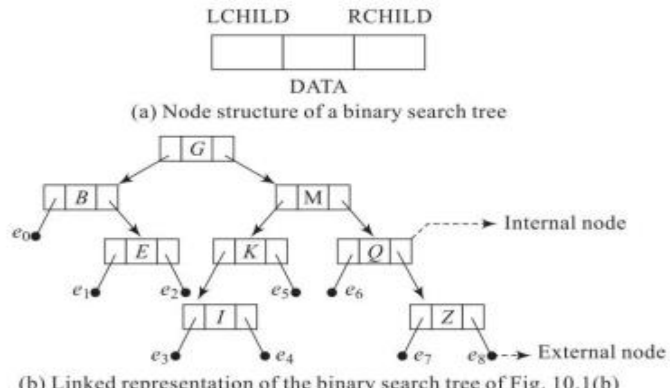
A binary search tree is commonly represented using a linked representation in the same way as that of a binary tree.

The node structure and the linked representation of the binary search tree shown in above Fig. is illustrated in below Fig.

However, the null pointers of the nodes may be represented using fictitious nodes called **external nodes**.

Thus a linked representation of a binary search tree is viewed as a bundle of external nodes which represent the null pointers and **internal nodes** which represent the keys.

Such a binary tree is referred to as an **extended binary tree**. Obviously, the number of external nodes in a binary search tree comprising n internal nodes is n+1. The path from the root to an external node is called as an **external path**.



(a) Node structure of a binary search tree

(b) Linked representation of the binary search tree of Fig. 10.1(b)

Binary Search Trees Operations:

Retrieval from a binary search tree Let T be a binary search tree. To retrieve a key u from T, u is first compared with the root key r of T.

If u = r then the search is done. If u < r then the search begins at the left subtree of T. If u > r then the search begins at the right subtree of T.

The search is repeated recursively in the left and right sub-subtrees with u compared against the respective root keys, until the key u is either found or not found.

If the key is found the search is termed successful and if not found, is termed unsuccessful.

While all successful searches terminate at the appropriate internal nodes in the binary search tree, all unsuccessful searches terminate only at the external nodes in the appropriate portion of the binary search tree. Hence external nodes are also referred to as failure nodes.

Algorithm : Procedure to retrieve ITEM from a binary search tree T

procedure FIND_BST(T, ITEM, LOC)

/* LOC is the address of the node containing ITEM which is to be retrieved from the binary search tree T. In case of unsuccessful search the procedure prints the message ITEM not found and returns LOC as NIL.*/

if T = NIL then

{

```
   print ( binary search tree T is empty);
  exit;
 }                    /* exit procedure*/
else
 LOC = T;
 while (LOC ≠ NIL) do
 case
          : ITEM = DATA(LOC)
                   return (LOC);            /* ITEM found in node LOC*/
          : ITEM < DATA(LOC)
                   LOC = LCHILD(LOC);   /* search left subtree*/
          : ITEM > DATA(LOC)
                   LOC= RCHILD(LOC);    /* search right subtree*/
  Endcase
 endwhile
If (LOC=NIL) then
{
          print(ITEM not found);
        return (LOC)
}           /* unsuccessful search*/
end FIND_BST15
```

Example: Consider the set S = {416, 891, 456, 765, 111, 654, 345, 256, 333} whose associated binary search tree T is shown in Fig below.

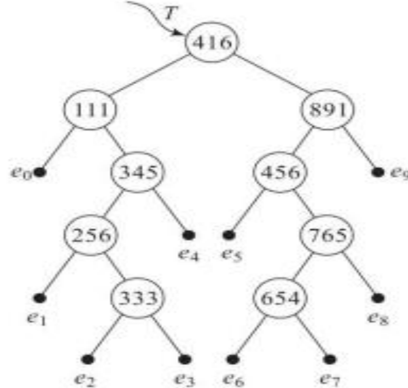


Table: Trace of Algorithm for the retrieval of ITEM=333

LOC	ITEM <, =, > DATA(LOC)?	Updated LOC
Initially	333 < 416	LOC = LCHILD(#(416)) = #(111)
LOC = T = #(416)		
LOC = #(111)	333 > 111	LOC = RCHILD(#(111)) = #(345)
LOC = #(345)	333 < 345	LOC = LCHILD(#(345)) = #(256)
LOC = #(256)	333 > 256	LOC = RCHILD(#(256)) = #(333)
LOC = #(333)	333 = 333	RETURN(#(333)) Element found and node returned

Table: Trace of Algorithm for the retrieval of ITEM=777

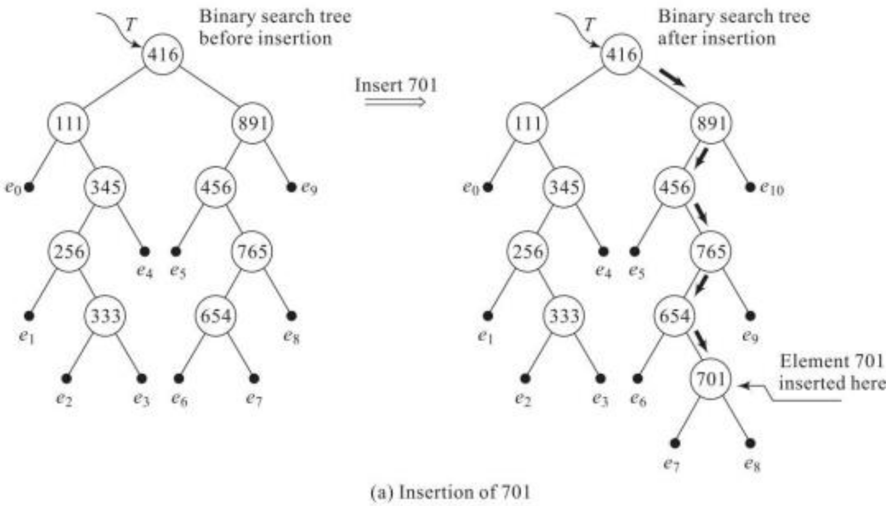
LOC	ITEM <, =, > DATA(LOC)?	Updated LOC
Initially LOC = T = # (416)	777 > 416	LOC = RCHILD(# (416)) = # (891)
LOC = # (891)	777 < 891	LOC = LCHILD(# (891)) = # (456)
LOC = # (456)	777 > 456	LOC = RCHILD(# (456)) = # (765)
LOC = # (765)	777 > 765	LOC = RCHILD(# (765)) = NIL
LOC = NIL		Element not found RETURN (NIL)

Insertion into a binary search tree:

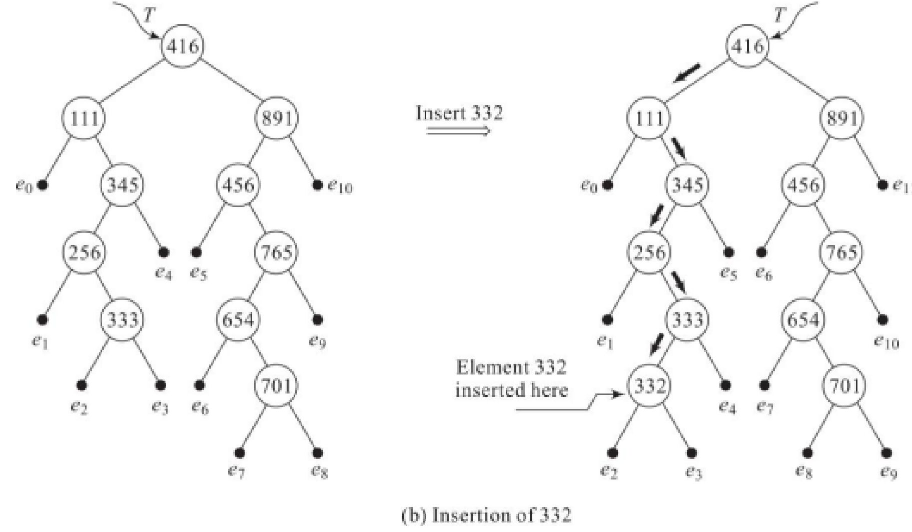
The insertion of a key into a binary search tree is similar to the retrieval operation. The insertion of a key u initially proceeds as if it were trying to retrieve the key from the binary search tree, but on reaching the null pointer (failure node) which it is sure to encounter since key u is not present in the tree, a new node containing the key u is inserted at that position.

Example: Let us insert keys 701 and 332 into the binary search tree T associated with set S ={ 416, 891, 456, 765, 111, 654, 345, 256, 333}, shown in Fig below, Figure (a) shows the insertion of 701.

Note how the operation moves down the tree in the path shown and when it encounters a failure node e7, the key 701 is inserted as the right child of node containing 654. Again the insertion of 332 which follows a similar procedure is illustrated in Fig. (b).



(a) Insertion of 701



(b) Insertion of 332

Deletion from a binary search tree:

For the deletion of a key from the binary search tree we first search for the node containing the key by undertaking a retrieval operation.

But once the node is identified, the following cases are tested before the node containing the key u is appropriately deleted from the binary search tree T:

(i) key u is a leaf node

(ii) key u has a lone subtree (left subtree or right subtree only)

(iii) key u has both left subtree and right subtree

Case (i) If the key u to be deleted is a leaf node then the deletion is trivial since the appropriate link field of the parent node of key u only needs to be set as NIL.

Case (ii) If the key u to be deleted has either a left subtree or a right subtree (but not both) then the link of the parent node of u is set to point to the appropriate subtree.

Case (iii) If the key u to be deleted has both a left subtree and a right subtree, then the problem is complicated. In this case since the right subtree comprises keys that are greater than u, the parent node of key u is now set to point to the right subtree of u.

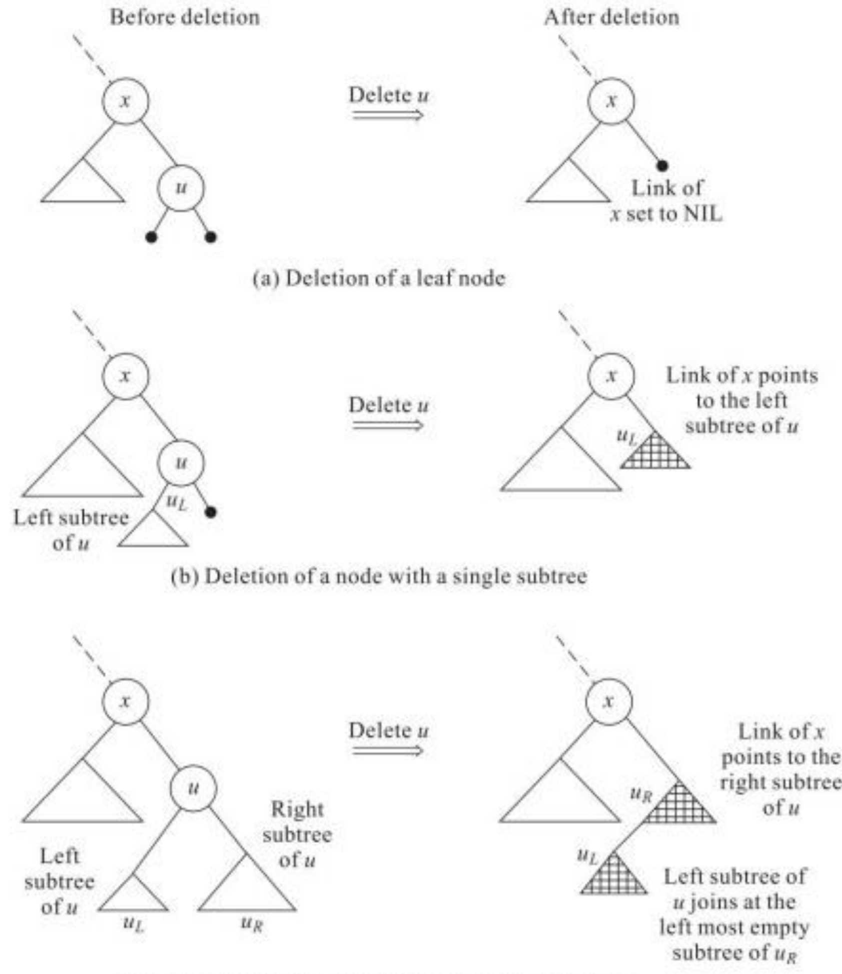
Since all the keys of the left subtree of u are less than that of the right subtree of u, we move as far left as possible in the right subtree of u until an empty left subtree is found and link the left subtree of u at that position.

Algorithm : Procedure to delete a node NODE_U from a binary search tree given its parent node NODE_X

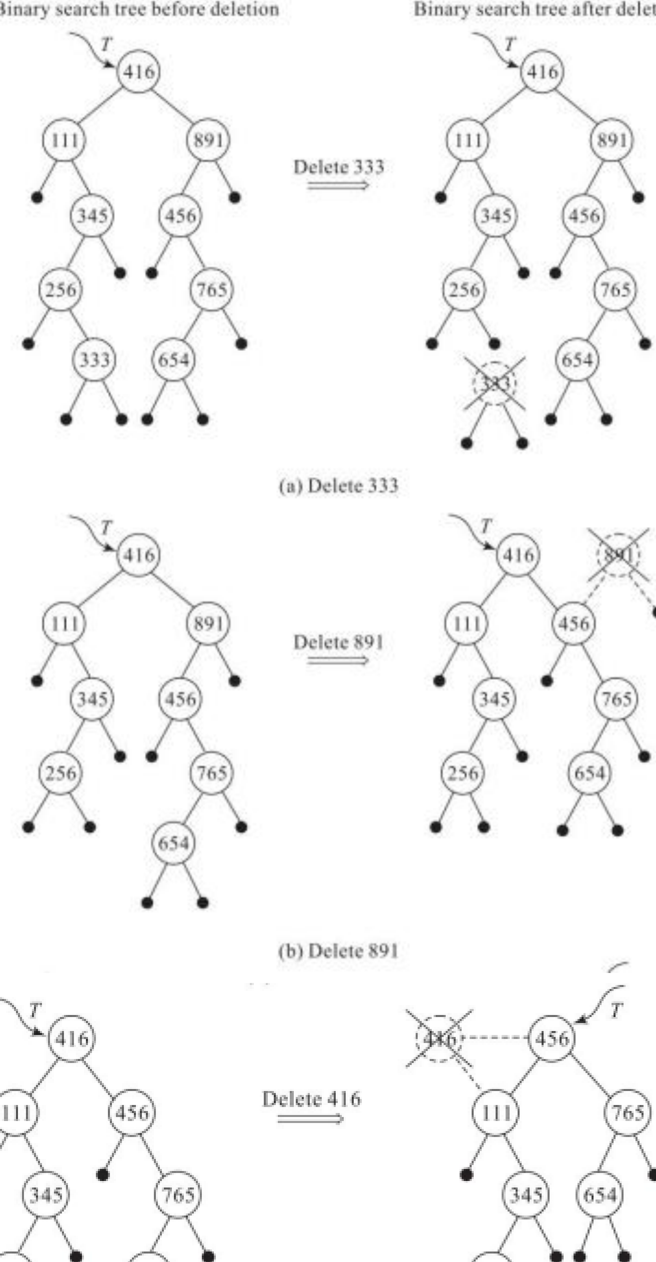
procedure DELETE(NODE_U, NODE_X)

/* NODE_U is the node which is to be deleted from the binary search tree. NODE_X is the parent node for which NODE_U may be the left child or the right child. Procedure DELETE is applicable for deletion of all non-empty nodes other than the root (i.e.) NODE_U ≠ NIL and NODE_X ≠ NIL */

```
case
    :LCHILD(NODE_U )=RCHILD(NODE_U )=NIL:
         Set RCHILD(NODE_X) or LCHILD(NODE_X)to NIL based on whether NODE_U is the right
child or left child of NODE_X respectively;
        call RETURN(NODE_U );

 :LCHILD(NODE_U )< > NIL and RCHILD(NODE_U )<> NIL:

          Set RCHILD(NODE_X) or LCHILD(NODE_X) to RCHILD(NODE_U) based on whether
NODE_U is the right child or left child of NODE_X respectively;

         TEMP= RCHILD(NODE_U);
          while (LCHILD(TEMP) <> NIL) do
         TEMP=LCHILD(TEMP);
          endwhile
         LCHILD(TEMP) = LCHILD(NODE_U);
          call RETURN (NODE_U);

 :LCHILD (NODE_U)< > NIL and RCHILD(NODE_U) = NILTEMP=LCHILD(NODE_U);
         Set RCHILD(NODE_X) or LCHILD(NODE_X) to TEMP based on whether NODE_U is the right
child or left child of NODE_X respectively;
          call RETURN(NODE_U);

  :LCHILD(NODE_U) = NIL and RCHILD(NODE_U) < > NIL:
         TEMP=RCHILD(NODE_U);
         Set RCHILD(NODE_X) or LCHILD(NODE_X )to TEMP based on whether NODE_U is the right
child or left child of NODE_X respectively;
         call RETURN(NODE_U );
 endcase
end DELETE
```



Example: Delete keys 333, 891 and 416 in the order given, from the binary search tree T associated with set S = {416, 891, 456, 765, 111, 654, 345, 256, 333} shown in below Fig.



(a) Delete 333

(b) Delete 891

(c) Delete 416

GRAPHS:

Definitions and Basic Terminologies:

Graph A graph G = (V, E) consists of a finite non empty set of vertices V also called points or nodes and a finite set E of unordered pairs of distinct vertices called edges or arcs or links.

Example Figure below illustrates a graph. Here V = {a, b, c, d} and E = {(a, b), (a, c), (b, c), (c, d)}. However it is convenient to represent edges using labels as shown in the figure.

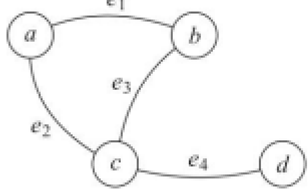


Fig. A graph

V : Vertices : {a, b, c, d}

E : Edges : {e1, e2, e3, e4}

- A graph G = (V, E) where E = ∅, is called as **a null** or **empty graph**.
- A graph with one vertex and no edges is called **a trivial graph**.

Multigraph:

A multigraph G = (V, E) also consists of a set of vertices and edges except that E may contain **multiple edges** (i.e.) edges connecting the same pair of vertices, or may contain **loops or self edges** (i.e.) an edge whose end points are the same vertex.

Example Figure below illustrates a multigraph Observe the multiple edges e1, e2 connecting vertices a, b and e5, e6, e7 connecting vertices c, d respectively. Also note the self edge e4.

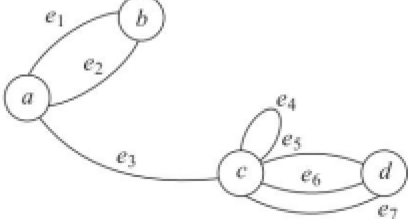


Fig. A multigraph

Directed and undirected graphs:

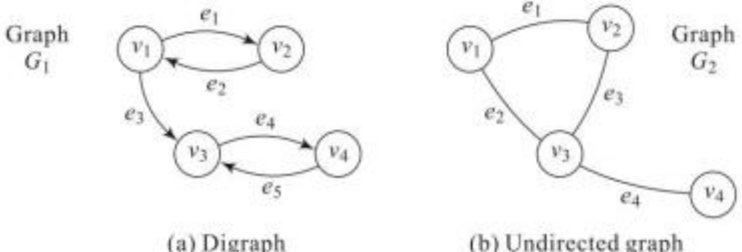
A graph whose definition makes reference to unordered pairs of vertices as edges is known as an undirected graph.

The edge eij of such an undirected graph is represented as (vi , vj) where vi , vj are distinct vertices. Thus an undirected edge (vi , vj) is equivalent to (vj , vi).

On the other hand, **directed graphs or digraphs** make reference to edges which are directed (i.e.) edges which are ordered pairs of vertices.

The edge eij is referred to as which is distinct from < vj , vi > where vi , vj are distinct vertices.

Example: Figure below illustrates a digraph and an undirected graph.



(a) Digraph (b) Undirected graph

In Fig. (a), e1 is a directed edge between v1 and v2, (i.e.) e1 = <v1,v2>, whereas in Fig. (b) e1 is an undirected edge between v1 and v2, (i.e.) e1 = (v1, v2).

The list of vertices and edges of graphs G1 and G2 are:

Vertices (G1) : {v1, v2, v3, v4}

Vertices (G2) : {v1, v2, v3, v4}

Edges (G1) : {<v1,v2>,<v2,v1>,<v1,v3>,<v3,v4>,<v4,v3> } or {e1, e2, e3, e4, e5}

Edges (G2) : {(v1, v2) (v1, v3) (v2, v3) (v3, v4)} or {e1, e2, e3, e4}

- In the case of an undirected edge (vi , vj) in a graph, the vertices vi , vj are said to be **adjacent** or the edge (vi , vj) is said to be **incident** on vertices vi , vj .
- Thus in Fig. (b) vertices v1, v3 are adjacent to vertex v2 and edges e1: (v1, v2), e3: (v2, v3) are incident on vertex v2.
- On the other hand, if is a directed edge, then vi is said to be adjacent to vj and vj is said to be adjacent from vi . The edge is incident to both vi and vj .

Complete graphs:

The number of distinct unordered pairs (vi , vj), vi ≠ vj in a graph with n vertices is $^nC_2$= (n(n-1))/2. An n vertex undirected graph with exactly (n(n-1))/2 edges is said to be complete.

Example Figure below illustrates a complete graph. The undirected graph with 4 vertices has all its $^4C_2$ = 6 edges intact.

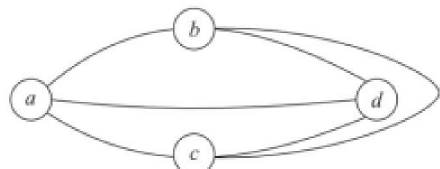
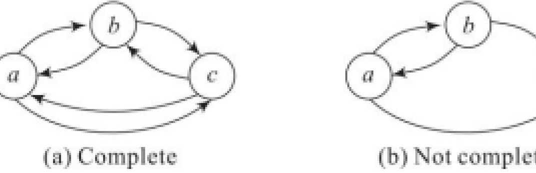


Fig. A complete graph

In the case of a digraph with n vertices, the maximum number of edges is given by $^nP_2$ = n. (n-1).

Such a graph with exactly n.(n-1) edges is said to be a complete digraph.

Example Figure (a) illustrates a digraph which is complete and Fig. (b) a graph which is not complete.

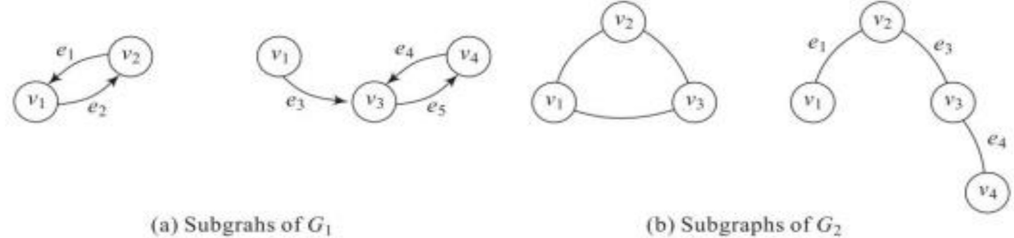


(a) Complete (b) Not complete

Subgraph:

A subgraph $G^{1'} = (V^{1'}, E^{1'})$ of a graph G = (V, E) is such that $V^1 \subseteq V$ and $E^1 \subseteq E$.

Example Figure below illustrates some subgraphs of the directed and undirected graphs shown in Fig. (Graphs G1 and G2)



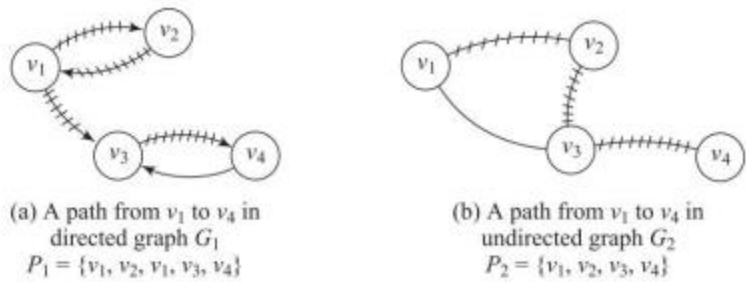
(a) Subgrahs of G_1 (b) Subgraphs of G_2

Path:

A path from a vertex vi to vertex vj in an undirected graph G is a sequence of vertices such that (vi , v_{l1}) (v_{l1} , v_{l2}), ..(v_{lk} , vj) are edges in G.

If G is directed then the path from vi to vj more specially known as a directed path consists of edges < vi , v_{l1}> < v_{l1} , v_{l2}> ..< v_{lk} , vj > in G.

Example Figure (a) illustrates a path P1 from vertex v1 to v4 in graph G1 and Fig (b) illustrates a path P2 from vertex v1 to v4 of graph G2.



(a) A path from v_1 to v_4 in directed graph G_1 $P_1 = \{v_1, v_2, v_1, v_3, v_4\}$

(b) A path from v_1 to v_4 in undirected graph G_2 $P_2 = \{v_1, v_2, v_3, v_4\}$

The length of a path is the number of edges on it. Example In above Fig. the length of path P1 is 4 and the length of path P2 is 3.

A simple path is a path in which all the vertices except possibly the first and last vertices are distinct.

Example In graph G2, the path from v1 to v4 given by {(v1, v2), (v2, v3), (v3, v4)} and written as {v1, v2, v3, v4} is a simple path where as the path from v3 to v4 given by {(v3, v1), (v1, v2), (v2, v3), (v3, v4)} and written as {v3, v1, v2, v3, v4} is not a simple path but a path due to the repetition of vertices.

A cycle is a simple path in which the first and last vertices are the same. A cycle is also known as a **circuit**, **elementary cycle**, **circular path** or **polygon**.

Example In graph G2 the path {v1, v2, v3, v1} is a cycle. Also, in graph G1 the path {v1, v2, v1} is a cycle or more specifically a directed cycle.

Connected graphs:

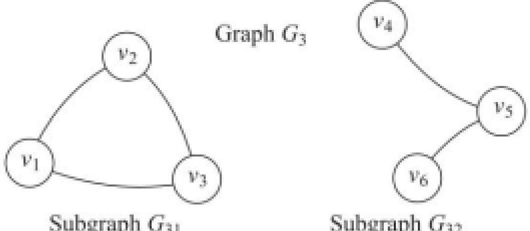
Two vertices vi , vj in a graph G are said to be connected only if there is a path in G between vi and vj .

In an undirected graph if vi and vj are connected then it automatically holds that vj and vi are also connected. An undirected graph is said to be a connected graph if every pair of distinct vertices vi , vj are connected.

Example Graph G2 is connected where as graph G3 shown in below Fig. is not connected.

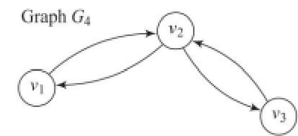
In the case of an undirected graph which is not connected, the **maximal connected subgraph** is called as a **connected component** or simply a component.

Example Graph G3 has two connected components viz., graph G31 and G32.

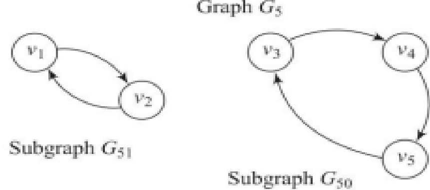


A directed graph is said to be **strongly connected** if every pair of distinct vertices vi , vj are connected (by means of a directed path). Thus if there exists a directed path from vi to vj then there also exists a directed path from vj to vi .

Example Graph G4 shown in Fig. is strongly connected.



However, the digraph shown in Fig. is not strongly connected but is said to possess two strongly connected components. A strongly connected component is a maximal subgraph that is strongly connected.



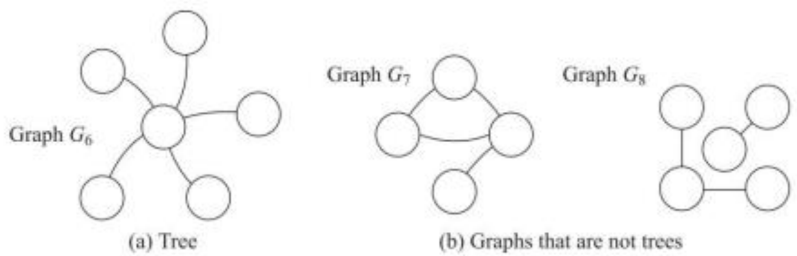
Trees:

A tree is defined to be a connected acyclic graph. The following properties are satisfied by a tree:

(i) There exists a path between any two vertices of the tree, and

(ii) No cycles must be present in the tree. In other words, trees are acyclic.

Example Figure (a) illustrates a tree. Figure (b) illustrates graphs which are not trees due to the violation of the property of acyclicity and connectedness respectively.



(a) Tree (b) Graphs that are not trees

Degree:

The degree of a vertex in an undirected graph is the number of edges incident to that vertex.

A vertex with degree one is called as a **pendant vertex** or **end** vertex.

A vertex with degree zero and hence has no incident edges is called an **isolated** vertex. **Example** In graph G2 the degree of vertex v3 is 3 and that of vertex v2 is 2.

In the case of digraphs, we define the **indegree** of a vertex v to be the number of edges with v as the head and the **outdegree** of a vertex to be number of edges with v as the tail. **Example** In graph G1 the indegree of vertex v3 is 2 and the out degree of vertex v4 is 1.

Isomorphic graphs:

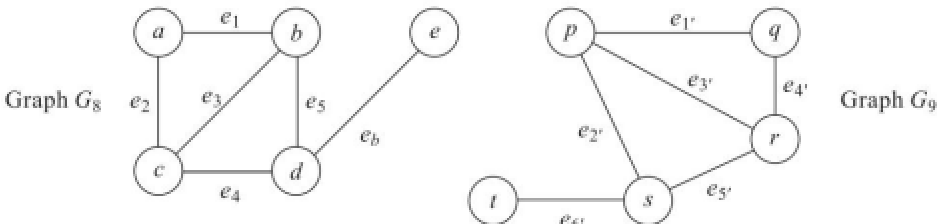
Two graphs are said to be isomorphic if,

(i) they have the same number of vertices

(ii) they have the same number of edges

(iii) they have an equal number of vertices with a given degree

Example Figure illustrates two graphs which are isomorphic.



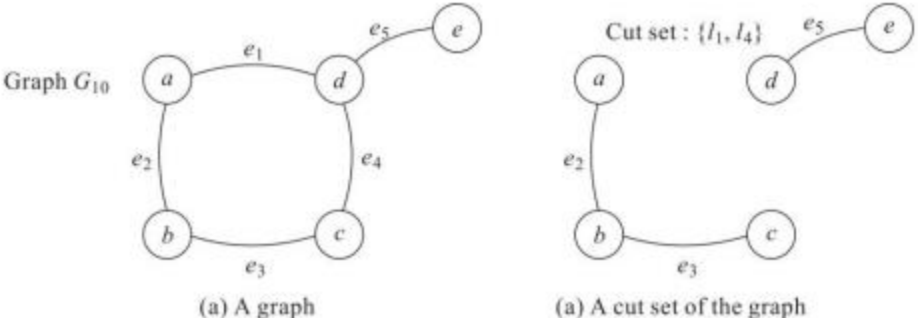
The property of isomorphism can be verified on the lists of vertices and edges of the two graphs G8 and G9 when superimposed as shown below:

Vertices (G_8) :	a	b	c	d	e	
	↕	↕	↕	↕	↕	
Vertices (G_9) :	q	p	r	s	t	
Degree of the vertices :	2	3	3	3	1	
Edges (G_8) :	e_1	e_2	e_3	e_4	e_5	e_6
Edges (G_9) :	e_1'	e_4'	e_3'	e_2'	e_5'	e_6'

Cut set:

A cut set in a connected graph G is the set of edges whose removal from G leaves G disconnected, provided the removal of no proper subset of these edges disconnects the graph G. Cut sets are also known as **proper cut set** or **cocycle** or **minimal cut set**.

Example Figure below illustrates the cut set of the graph G10. The cut set {e1, e4} disconnects the graph into two components as shown in the figure. {e5} is also another cut set of the graph.

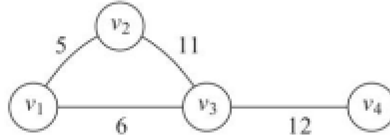


(a) A graph (a) A cut set of the graph

Labeled graphs:

A graph G is called a labeled graph if its edges and / or vertices are assigned some data. In particular if the edge e is assigned a non negative number l(e) then it is called the weight or length of the edge e.

Example Figure below illustrates a labeled graph. A graph with weighted edges is also known as a network.

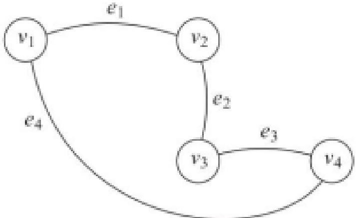


Eulerian graph:

A walk starting at any vertex going through each edge exactly once and terminating at the start vertex is called an Eulerian walk or Euler line.

The Koenigsberg bridge problem was in fact a problem of obtaining an Eulerian walk for the graph concerned. The solution to the problem is, an Eulerian walk is possible only if the degree of each vertex in the graph is even.

Given a connected graph G, G is an Euler graph iff all the vertices are of even degree. Example Figure below illustrates an Euler graph. {e1, e2, e3, e4} shows a Eulerian walk. The even degree of the vertices may be noted.

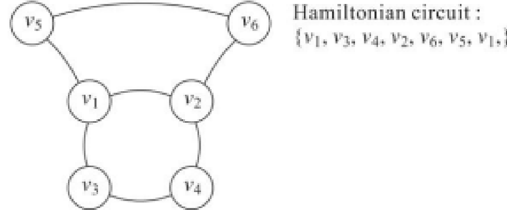


Hamiltonian circuit:

A Hamiltonian circuit in a connected graph is defined as a closed walk that traverses every vertex of G exactly once, except of course the starting vertex at which the walk terminates.

A circuit in a connected graph G is said to be Hamiltonian if it includes every vertex of G. If any edge is removed from a Hamiltonian circuit then what remains is referred to as a Hamiltonian path. Hamiltonian path traverses every vertex of G.

Example Figure below illustrates a Hamiltonian circuit.



Representations of Graphs:

The representation of graphs in a computer can be categorized as

1. sequential representation and
2. linked representation.

Of the two, though sequential representation has several methods, all of them follow a matrix representation thereby calling for their implementation using arrays. The linked representation of a graph makes use of a singly linked list as its fundamental data structure.

1. Sequential representation of graphs

The sequential or the matrix representation of graphs have the following methods:

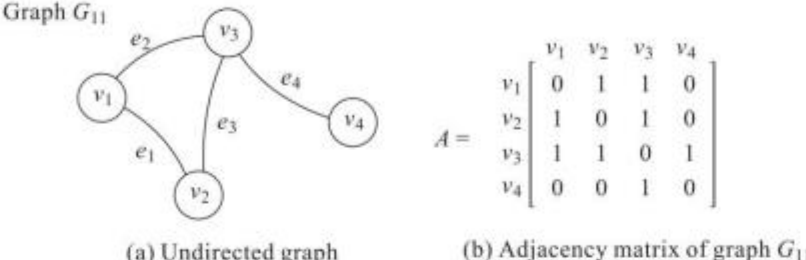
(i) Adjacency matrix representation
(ii) Incidence matrix representation
(iii) Circuit matrix representation
(iv) Cut set matrix representation
(v) Path matrix representation

i) Adjacency matrix representation :

The adjacency matrix of a graph G with n vertices is an n X n symmetric binary matrix given by A = [aij] defined as aij = 1 if the i th and jth vertices are adjacent (i.e.) there is an edge connecting the i th and j th vertices

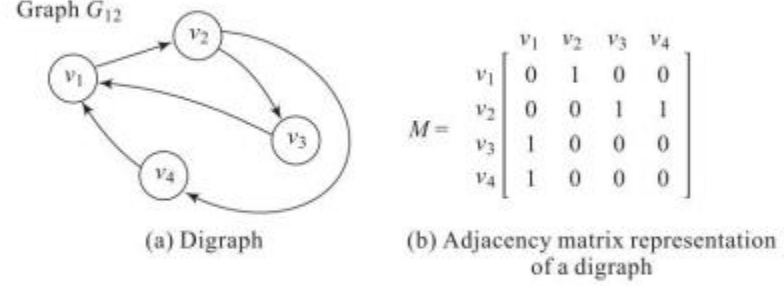
= 0 otherwise, (i.e.) if there is no edge linking the vertices.

Example Figure (a) illustrates an undirected graph whose adjacency matrix is shown in Fig. (b). It can easily be seen that while adjacency matrices of undirected graphs are symmetric, nothing can be said about the symmetricity of the adjacency matrix of digraphs.



(a) Undirected graph (b) Adjacency matrix of graph G_{11}

Example Figure below illustrates a digraph and its adjacency matrix representation.



(a) Digraph (b) Adjacency matrix representation of a digraph

ii) Incidence matrix representation :

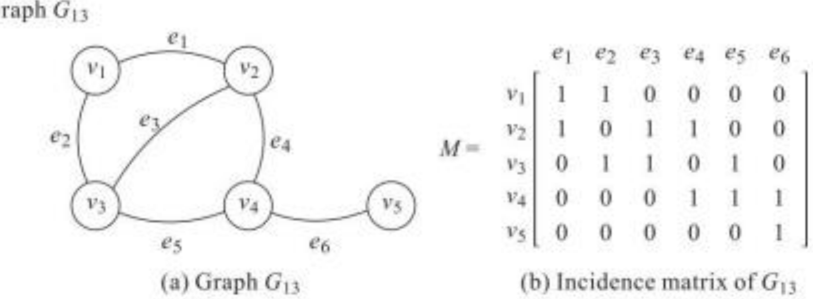
Let G be a graph with n vertices and e edges. Define an n X e matrix M = [mij] whose n rows correspond to n vertices and e columns correspond to e edges, as

mij = 1 if the j th edge ej is incident on the i th vertex vi ,

= 0 otherwise

Matrix M is known as the incidence matrix representation of the graph G.

Example Consider the graph G13 shown in below Fig. (a), the incidence matrix representation for the graph is given in Fig. (b).



(a) Graph G_{13} (b) Incidence matrix of G_{13}

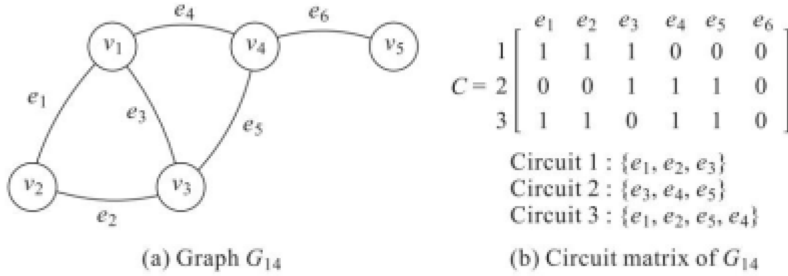
iii) Circuit matrix representation :

For a graph G let the number of different circuits be t and the number of edges be e. Then the circuit matrix C= [Cij] of G is a t X e matrix defined as

Cij = 1 if the i th circuit includes the j th edge,

= 0 otherwise

Example Consider the graph G14 shown in below Fig. (a). The circuits for this graph expressed in terms of their edges are 1: {e1, e2, e3} 2: {e3, e4, e5} 3: {e1, e2, e5, e4}. The circuit matrix C of order 3 X 6 is shown in Fig. (b).



	e_1	e_2	e_3	e_4	e_5	e_6
1	1	1	1	0	0	0
2	0	0	1	1	1	0
3	1	1	0	1	1	0

(a) Graph G_{14} (b) Circuit matrix of G_{14}

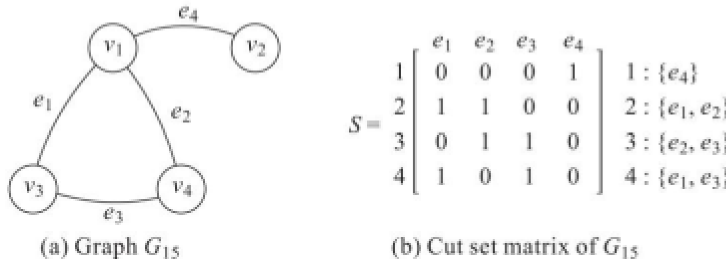
iv) Cut set matrix representation:

For a graph G, a matrix S = [sij] whose rows correspond to cut sets and columns correspond to edges of the graph is defined to be a cut set matrix if

sij = 1 if the i th cut set contains the j th edge,

= 0 otherwise

Example Consider the graph G15 shown in below Fig. (a). The cut sets of the graph are 1:{e4} 2:{e1, e2} 3:{e2, e3} and 4:{e1, e3}. The cut set matrix representation is shown in Fig. (b).



	e_1	e_2	e_3	e_4
1	0	0	0	1
2	1	1	0	0
3	0	1	1	0
4	1	0	1	0

(a) Graph G_{15} (b) Cut set matrix of G_{15}

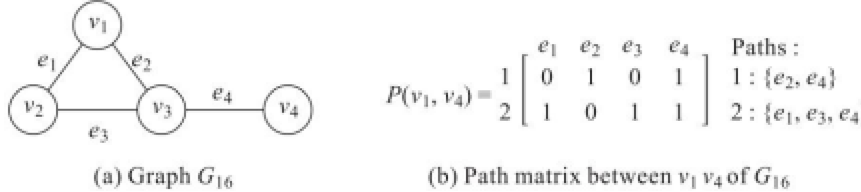
v) Path matrix representation :

A path matrix is generally defined for a specific pair of vertices in a graph. If (u, v) is a pair of vertices then the path matrix denoted as P(u,v) = [pij] is given by

pij = 1 if the jth edge lies in the i th path between vertices u and v,

= 0 otherwise

Example Consider the graph G16 shown in Fig. (a). The paths between vertices v1 and v4 are 1:{e2, e4} and 2:{e1, e3, e4}. The path matrix representation is shown in Fig (b).



	e_1	e_2	e_3	e_4
1	0	1	0	1
2	1	0	1	1

(a) Graph G_{16} (b) Path matrix between v_1 v_4 of G_{16}

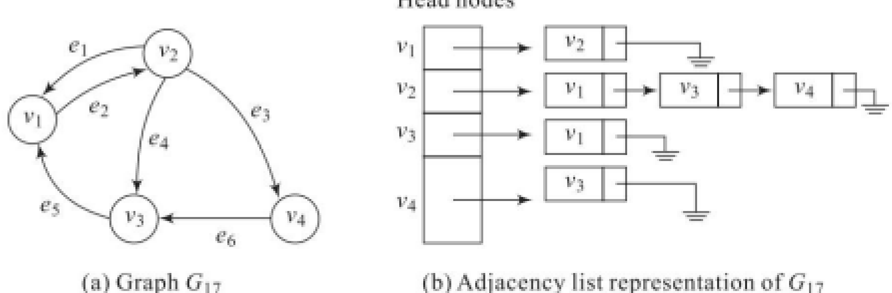
Of all these sequential representations, adjacency matrix representation represents the graph best and is the most widely used representation.

2. **Linked representation of graphs** :

The linked representation of graphs is referred to as **adjacency list representation** and is comparatively efficient with regard to adjacency matrix representation.

Given a graph G with n vertices and e edges, the adjacency list opens n head nodes corresponding to the n vertices of graph G, each of which points to a singly linked list of nodes, which are adjacent to the vertex representing the head node.

Example Figure below illustrates a graph and its adjacency list representation. It can easily be seen that if the graph is undirected, then the number of nodes in the singly linked lists put together is 2e where as in the case of digraphs the number of nodes is just e, where e is the number of edges in the graph.



(a) Graph G_{17} (b) Adjacency list representation of G_{17}

Graph Traversals:

Graphs support the following traversals:

1. Breadth first Traversal, and
2. Depth first Traversal.

A traversal, to recall, is a systematic walk which visits the nodes comprising the data structure (graphs in this case) in a specific order.

1. Breadth first traversal:

The breadth first traversal starts from a vertex u which is said to be visited. Now all nodes vi , adjacent to u are visited.

The unvisited vertices wij adjacent to each of vi are visited next and so on. The traversal terminates when there are no more nodes to visit.

The process calls for the maintenance of a queue to keep track of the order of nodes whose adjacent nodes are to be visited.

Algorithm: Breadth first traversal

Procedure BFT(s)

/* s is the start vertex of the traversal in an undirected graph G */

/* Q is a queue which keeps track of the vertices whose adjacent nodes are to be visited */

/* Vertices which have been visited have their ' visited' flags set to 1 (i.e.) visited (vertex) = 1. Initially, visited (vertex) = 0 for all vertices of graph G */

Initialize queue Q;

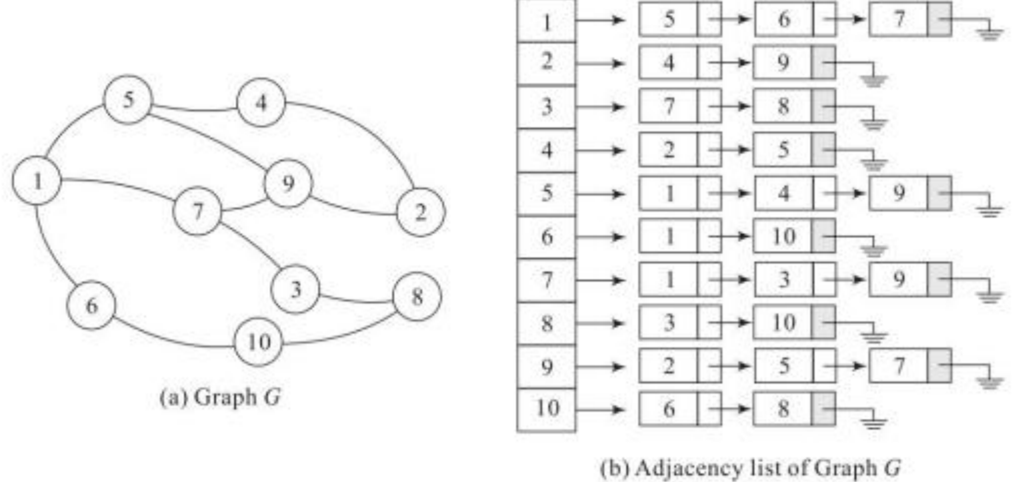
visited(s) = 1;

call ENQUEUE (Q,s); /* insert s into Q */

```
        while not EMPTY_QUEUE(Q) do                 /* process until Q is empty */
                call DEQUEUE (Q,s)                  /* delete s from Q*/
                print (s);                          /* output vertex visited */
                 for all vertices v adjacent to s do
                    if (visited (v) = 0) then
                          {
                           call ENQUEUE (Q, v);
                          visited (v) =1;
                          }
                  end
         endwhile
end BFT.
```

Breadth first traversal as its name indicates traverses the successors of the start node, generation after generation in a horizontal or linear fashion.

Example: Consider the undirected graph G shown in below Fig. (a) and its adjacency list representation shown in Fig.(b). The trace of procedure BFT(1) where the start vertex is 1, is shown in Table below.



(a) Graph *G*

(b) Adjacency list of Graph *G*

Fig. A graph and its adjacency list representation to demonstrate breadth first traversal

Table: Trace of the Breadth first traversal procedure on graph G.

Current Vertex	Queue *Q*	Traversal output	Status of visited flag of vertices {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} of graph *G*
1 (Start vertex)	1		1 2 3 4 5 6 7 8 9 10 1 0 0 0 0 0 0 0 0 0
1	5 6 7	1	1 2 3 4 5 6 7 8 9 10 1 0 0 0 1 1 1 0 0 0
5	5 7 4 9	1 5	1 2 3 4 5 6 7 8 9 10 1 0 0 1 1 1 1 0 1 0
6	7 4 9 10	1 5 6	1 2 3 4 5 6 7 8 9 10 1 0 0 1 1 1 1 0 1 1
7	4 9 10 3	1 5 6 7	1 2 3 4 5 6 7 8 9 10 1 0 1 1 1 1 1 0 1 1
4	9 10 3 2	1 5 6 7 4	1 2 3 4 5 6 7 8 9 10 1 1 1 1 1 1 1 0 1 1
9	10 3 2	1 5 6 7 4 9	1 2 3 4 5 6 7 8 9 10 1 0 1 1 1 1 1 0 1 1
10	3 2 8	1 5 6 7 4 9 10	1 2 3 4 5 6 7 8 9 10 1 1 1 1 1 1 1 1 1 1
3	2 8	1 5 6 7 4 9 10 3	1 2 3 4 5 6 7 8 9 10 1 1 1 1 1 1 1 1 1 1
2	8	1 5 6 7 4 9 10 3 2	1 2 3 4 5 6 7 8 9 10 1 1 1 1 1 1 1 1 1 1
8		1 5 6 7 4 9 10 3 2 8	1 2 3 4 5 6 7 8 9 10 1 1 1 1 1 1 1 1 1 1
		1 5 6 7 4 9 10 3 2 8	**Breadth first traversal ends**

The breadth first traversal starts from vertex 1 and visits vertices 5,6,7 which are adjacent to it, while enqueuing them into queue Q.

In the next shot, vertex 5 is dequeued and its adjacent, unvisited vertices 4, 9 are visited next and so on. The process continues until the queue Q which keeps track of the adjacent vertices is empty.

Depth first traversal:

The depth first traversal of an undirected graph starts from a vertex u which is said to be visited. Now, all the nodes vi adjacent to vertex u are collected and the first occurring vertex v1 is visited, deferring the visits to other vertices.

The nodes adjacent to v1 viz., w1k are collected and the first occurring adjacent vertex viz., w11 is visited deferring the visit to other adjacent nodes and so on. The traversal progresses until there are no more visits possible.

Algorithm : Depth first traversal

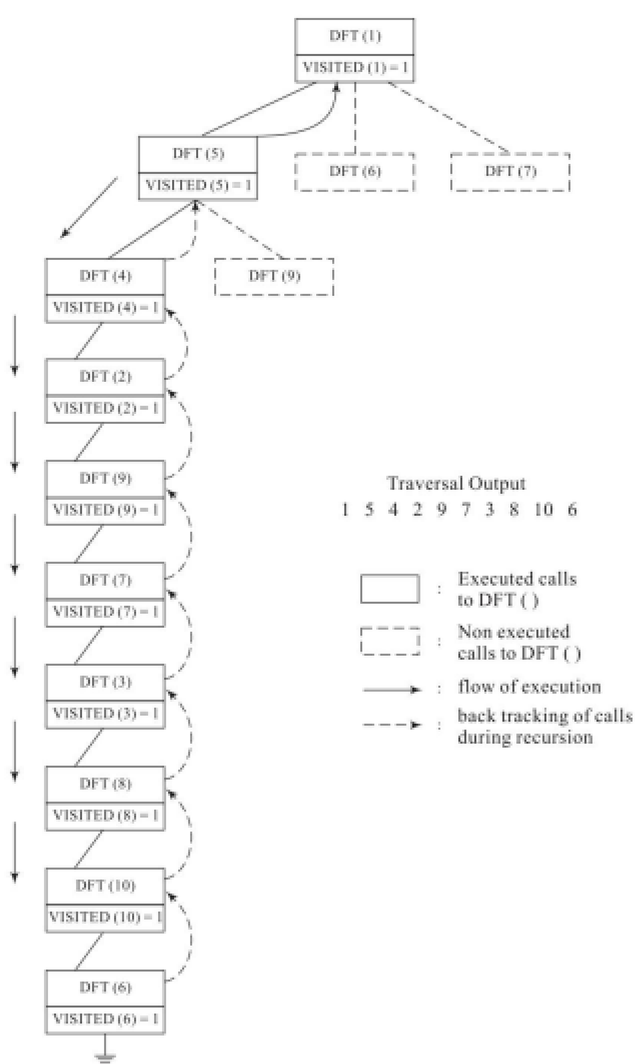
```
Procedure DFT(s)                /* s is the start vertex */
     visited(s) = 1;
    print (s);                  /* Output visited vertex */
    for each vertex v adjacent to s do
       if visited(v) = 0 then
        call DFT(v);
     end
    end DFT
```

The depth first traversal as its name indicates visits each node, that is, the first occurring among its adjacent nodes and successively repeats the operation, thus moving deeper and deeper into the graph.

In contrast, breadth first traversal moves sideways or breadth ways in the graph. **Example** illustrates a depth first traversal of a undirected graph.

Consider the undirected graph G and its adjacency list representation shown in above Fig.

Figure shows a tree of recursive calls which represents a trace of the procedure DFT(1) on the graph G with start vertex 1.



The tree of recursive calls illustrates the working of the DFT procedure. The first call DFT(1) visits start vertex 1 and releases 1 as the traversal output.

Vertex 1 has vertices 5, 6, 7 as its adjacent nodes. DFT(1) now invokes DFT(5), visiting vertex 5 and releasing it as the next traversal output.

However DFT(6) and DFT(7) are kept in waiting for DFT(5) to complete its execution. Such procedure calls waiting to be executed are shown in broken line boxes in the tree of recursive calls.

Now DFT(5) invokes DFT(4) releasing vertex 4 as the traversal output while DFT(9) is kept in abeyance. Note that though vertex 1 is an adjacent node of vertex 5, since no DFT( ) calls to vertices already visited are invoked, DFT(1) is not called for.

The process continues until DFT(6) completes its execution with no more nodes left to visit. During recursion the calls made to DFT( ) procedure are indicated using solid arrows in the forward direction.

Once DFT(6) finishes execution, back tracking takes place which is indicated using broken arrows in the reverse direction. Once DFT(1) completes execution the traversal output is gathered to be **1 5 4 2 9 7 3 8 10 6.**