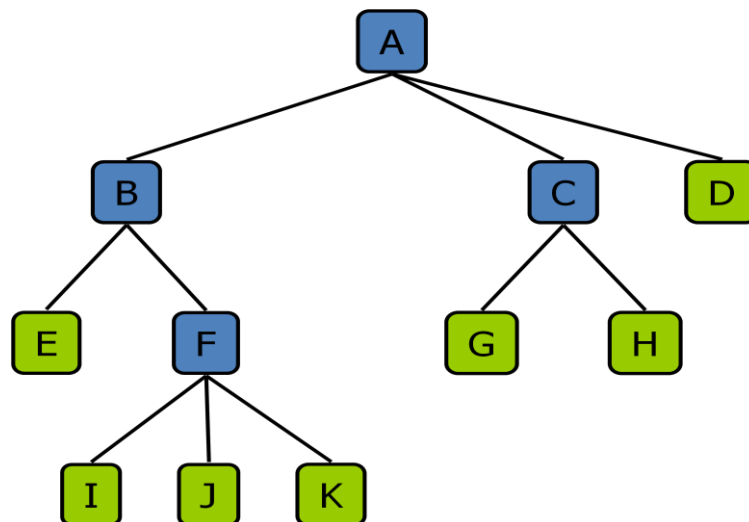


UNIT - III

TREES:

- 1) A tree is non – linear, hierarchical Data Structure.
- 2) A tree is a finite non empty set of elements. It is an abstract model of a hierarchical structure consists of nodes with a parent- child relation.
- 3) A tree T is defined recursively as follows:
 1. A set of zero items is a tree, called the empty tree (or null tree).
 2. If T_1, T_2, \dots, T_n are n trees for $n > 0$ and R is a node, then the set T containing R and the trees T_1, T_2, \dots, T_n are a tree. Within T , R is called the root of T , and T_1, T_2, \dots, T_n are called subtrees.

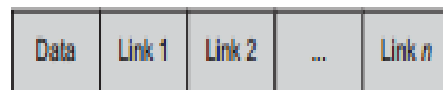
Tree Terminology



- ✚ **Root:** node without parent (A)
- ✚ **Subtree:** tree consisting of a node and its descendants
- ✚ **Siblings:** nodes share the same parent (B,C,D) (E,F) (I,J,K) (G,H)
- ✚ **Internal node:** node with at least one child (A, B, C, F)
- ✚ **LEAVES(External node):** node without children (E, I, J, K, G, H, D)
- ✚ **Ancestors of a node:** parent, grandparent, grand-grandparent, etc.
- ✚ **Descendant of a node:** child, grandchild, grand-grandchild, etc.
- ✚ **Depth of a node:** number of ancestors
- ✚ **Height of a tree:** maximum depth of any node (3)
- ✚ **Degree of a node:** the number of its children
- ✚ **Degree of a tree:** the maximum number of its node.

Representation of a General Tree

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



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

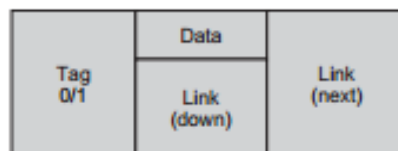


Figure 7.10 shows a sample tree.

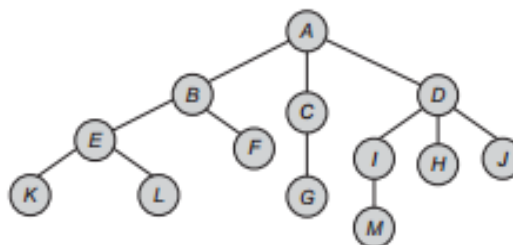


Fig. 7.10 Sample tree

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

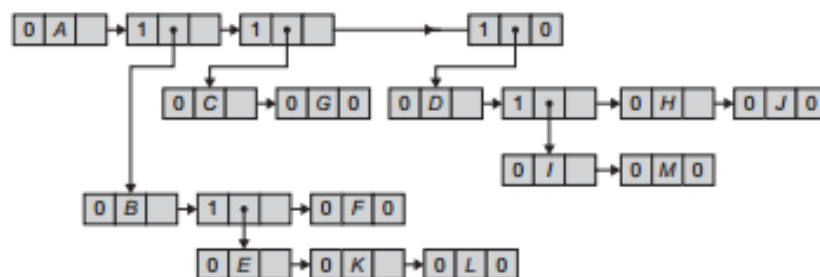
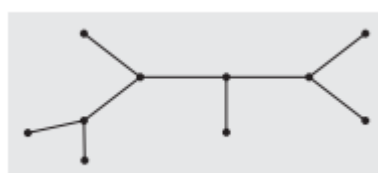


Fig. 7.11 List representation

TYPES OF TREES

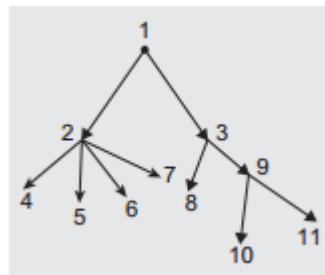
1. Free tree: A free tree is a connected, acyclic graph. It is an undirected graph. It has no node designated as a root. As it is connected, any node can be reached from any other node through a unique path. The tree in Fig. is an example of a free tree.



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



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



- 1. Regular tree** A tree where each branch node vertex has the same outdegree is called a *regular tree*. If in a directed tree, the outdegree of every node is less than or equal to m , then the tree is called an *m-ary tree*. If the outdegree of every node is exactly equal to m (the branch nodes) or zero (the leaf nodes), then the tree is called a *regular m-ary tree*.
- 2. Binary tree** A binary tree is a special form of tree. Each and every node in the binary tree contains 2 or less than 2 children except leaf nodes.

Partial Binary tree: Each and every node in the tree contains less than 2 children except leaf nodes.

Full Binary tree: Each and every node in the tree (Except Leaf nodes) contains exactly 2 nodes is called Full Binary tree.

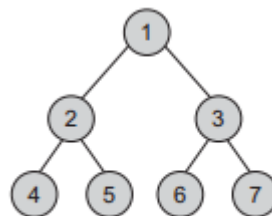


Fig. 7.15 Full binary tree

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

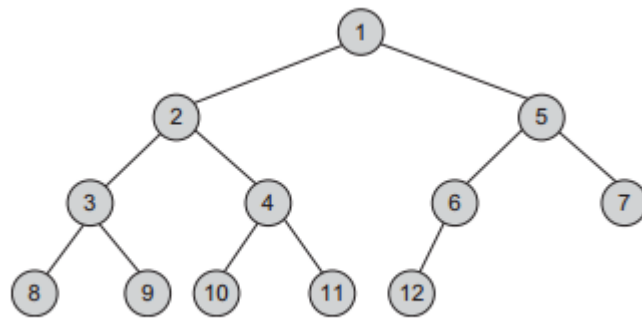


Fig. 7.16 Complete binary tree

4. **Left skewed binary tree** :If the right subtree is missing in every node of a tree, we call it a *left skewed tree* (Fig. 7.17). If the left subtree is missing in every node of a tree, we call it as *right subtree* (Fig. 7.18).

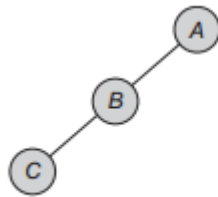


Fig. 7.17 Left skewed tree

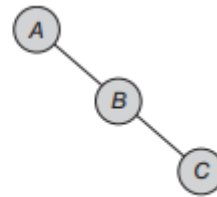


Fig. 7.18 Right skewed tree

5. **Strictly binary tree** If every non-terminal node in a binary tree consists of non-empty left and right subtrees, then such a tree is called a *strictly binary tree*. In Fig. 7.19, the non-empty nodes D and E have left and right subtrees. Such expression trees are known as *strictly binary trees*.

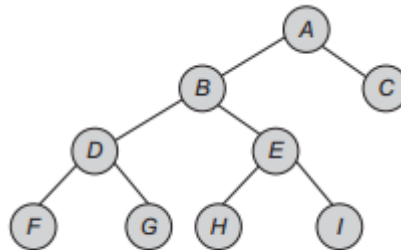


Fig. 7.19 Strictly binary tree

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

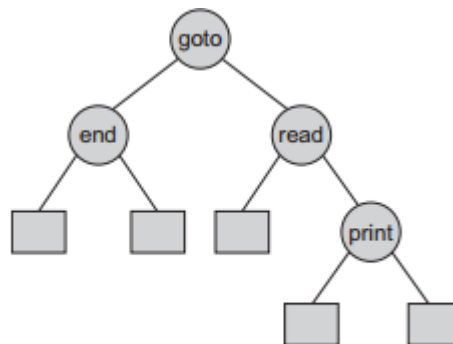


Fig. 7.20 Extended binary tree

BINARY TREE ABSTRACT DATA TYPE(ADT):

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

ADT btree

1. Declare create() \rightarrow Ebtree
 2. makebtree(btree, element, btree) \rightarrow Ebtree
 3. isEmpty(btree) \rightarrow Eboolean
 4. leftchild(btree) \rightarrow Ebtree
 5. rightchild(btree) \rightarrow Ebtree
 6. data(btree) \rightarrow Eelement
 7. for all l,r \in Ebtree, e \in Eelement, Let
 8. isEmpty(create) = true
 9. isEmpty(makebtree(l,e,r)) = false
 10. leftchild(create()) = error
 11. rightchild(create()) = error
 12. leftchild(makebtree(l,e,r)) = l
 13. rightchild(makebtree(l,e,r)) = r
 14. data(makebtree(l,e,r)) = e
 15. end
- end btree

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

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

BINARY TREE ADT CLASS:

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

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

REPRESENTATION OF A BINARY TREE

Array implementation of Binary Trees

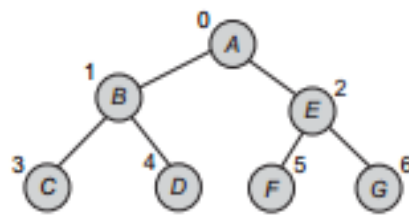


Fig. 7.24 Complete binary tree

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

0	1	2	3	4	5	6	7	8
A	B	E	C	D	F	G	-	-

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

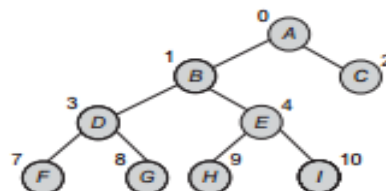


Fig. 7.25 Tree with 11 nodes

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

Level	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	19
	A	B	C	D	E	-	-	F	G	H	I	-	-	-	-	...	

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

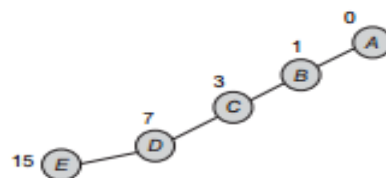


Fig. 7.26 Sample skewed tree

This tree has the following array representation:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	19
A	B	-	C	-	-	-	D	-	-	-	-	-	-	-	E	...	-

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

Linked implementation of Binary Trees

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



Fig. 7.27 Tree node

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

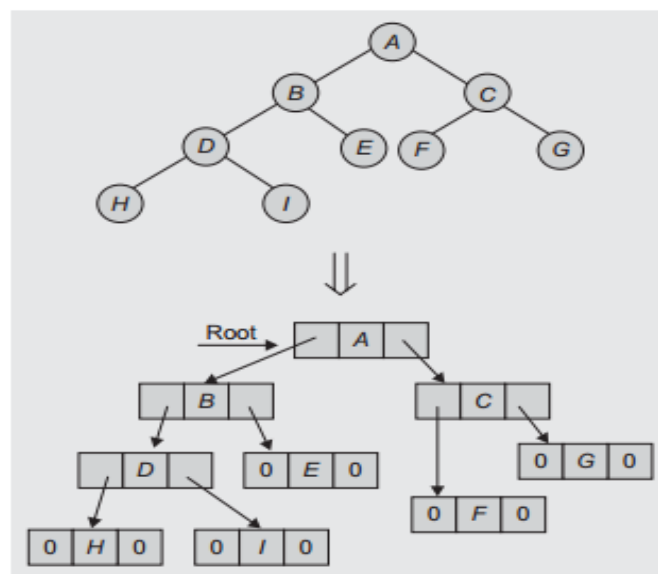


Fig. 7.28 Sample tree 1 and its linked representation

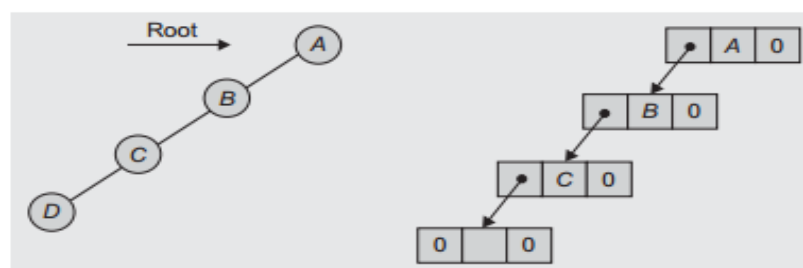


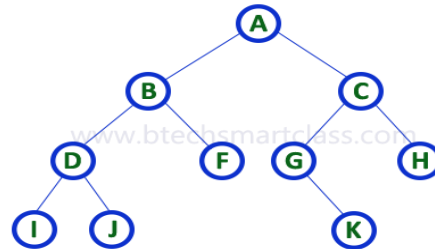
Fig. 7.29 Sample tree 2 and its linked representation

BINARY TREE TRAVERSAL:

Traversal means visiting each and every node in the tree. There are three types of binary tree traversals.

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

Consider the following binary tree...



1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit '**I**' then go for its root node '**D**' and later we visit D's right child '**J**'. With this we have completed the left part of node B. Then visit '**B**' and next B's right child '**F**' is visited. With this we have completed left part of node A. Then visit root node '**A**'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit '**G**' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node '**C**' and next visit C's right child '**H**' which is the right most child in the tree so we stop the process.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

The inorder traversal is also called as *symmetric traversal*. This traversal can be written as a recursive function as follows:

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

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

The Preorder() function can be written as both recursive and non recursive.

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

2. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

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

BINARY SEARCH TREE

Binary Search Tree, is a node-based binary tree data structure which has the following Properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

There must be no duplicate nodes.

EX:

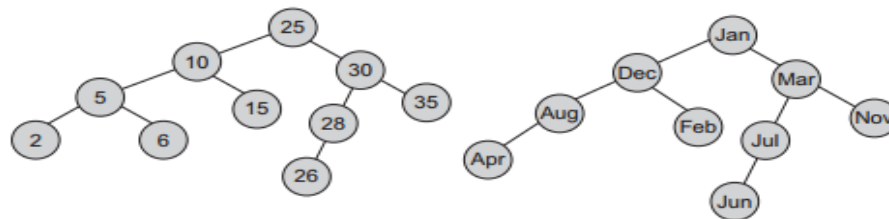


Fig. 7.54 Binary search trees

Applications of binary trees

- **Binary Search Tree** - Used in *many* search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
- **Binary Space Partition** - Used in almost every 3D video game to determine what objects need to be rendered.
- **Binary Tries** - Used in almost every high-bandwidth router for storing router-tables.
- **Hash Trees** - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.
- **Heaps** - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality-of-Service in routers, and A* (*path-finding algorithm used in AI applications, including robotics and video games*). Also used in heap-sort.
- **Huffman Coding Tree (Chip Uni)** - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
- **GGM Trees** - Used in cryptographic applications to generate a tree of pseudo-random numbers.
- **Syntax Tree** - Constructed by compilers and (implicitly) calculators to parse expressions.
- **Treap** - Randomized data structure used in wireless networking and memory allocation.
- **T-tree** - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so.

GRAPHS:

Graph is a collection of vertices and arcs which connects vertices in the graph

Graph is a collection of nodes and edges which connects nodes in the graph

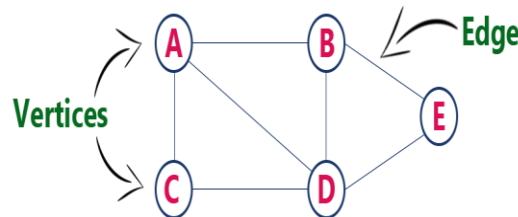
Generally, a graph G is represented as $G = (V, E)$, where V is set of vertices and E is set of edges.

Example

The following is a graph with 5 vertices and 7 edges.

This graph G can be defined as $G = (V, E)$

Where $V = \{A, B, C, D, E\}$ and $E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}$.



Graph Terminology

We use the following terms in graph data structure...

Vertex

An individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge** - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A, B) is equal to edge (B, A).
2. **Directed Edge** - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A, B) is not equal to edge (B, A).
3. **Weighted Edge** - A weighted edge is an edge with cost on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin

If an edge is directed, its first endpoint is said to be origin of it.

Destination

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

Incident

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

GRAPH ABSTRACT DATA TYPE:

Graphs as non-linear data structures represent the relationship among data elements, having more than one predecessor and/or successor. A graph G is a collection of nodes (*vertices*) and arcs joining a pair of the nodes (*edges*). Edges between two vertices represent the relationship between them. For finite graphs, V and E are finite. We can denote the graph as $G = (V, E)$. Let us define the graph ADT. We need to specify both sets of vertices and edges. Basic operations include creating a graph, inserting and deleting a vertex, inserting and deleting an edge, traversing a graph, and a few others.

A graph is a set of vertices and edges $\{V, E\}$ and can be declared as follows:

```
graph
create() ∅ Graph
insert_vertex(Graph, v) ∅ Graph
delete_vertex(Graph, v) ∅ Graph
insert_edge(Graph, u, v) ∅ Graph
delete_edge(Graph, u, v) ∅ Graph
is_empty(Graph) ∅ Boolean;
end graph
```

These are the primitive operations that are needed for storing and processing a graph.

Create

The create operation provides the appropriate framework for the processing of graphs. The create() function is used to create an empty graph. An empty graph has both V and E as null sets. The empty graph has the total number of vertices and edges as zero. However, while implementing, we should have V as a non-empty set and E as an empty set as the mathematical notation normally requires the set of vertices to be non-empty.

Insert Vertex

The insert vertex operation inserts a new vertex into a graph and returns the modified graph. When the vertex is added, it is isolated as it is not connected to any of the vertices in the graph through an edge. If the added vertex is related with one (or more) vertices in the graph, then the respective edge(s) are to be inserted. Figure 8.1(a) shows a graph $G(V, E)$, where $V = \{a, b, c\}$ and $E = \{(a, b), (a, c), (b, c)\}$, and the resultant graph after inserting the node d . The resultant graph G is shown in Fig. 8.1(b). It shows the inserted vertex with resultant $V = \{a, b, c, d\}$. We can show the adjacency relation with other vertices by adding the edge. So now, E would be $E = \{(a, b), (a, c), (b, c), (b, d)\}$ as shown in Fig.

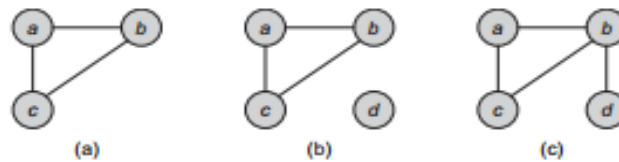


Fig. 8.1 Inserting a vertex in a graph (a) Graph G (b) After inserting vertex d (c) After adding an edge

Is_empty:

The is empty operation checks whether the graph is empty and returns true if empty else returns false. An empty graph is one where the set V is a null set. These are the basic operations on graphs, and a few more include getting the set of adjacent nodes of a vertex or an edge and traversing a graph. Checking the adjacency between vertices means verifying the relationship between them, and the relationship is maintained using a suitable data structure.

Delete Vertex

The delete vertex operation deletes a vertex and all the incident edges on that vertex and returns the modified graph. Figure 8.2(a) shows a graph $G(V, E)$ where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d)\}$, and the resultant graph after deleting the node c is shown in Fig. 8.2(b) with $V = \{a, b, d\}$ and $E = \{(a, b), (b, d)\}$.

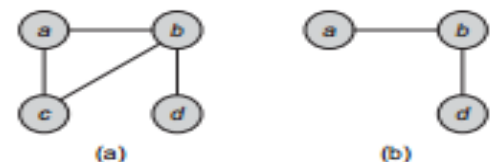


Fig. 8.2 Deleting a vertex from a graph (a) Graph G (b) Graph after deleting vertex c

Insert Edge

The insert edge operation adds an edge incident between two vertices. In an undirected graph, for adding an edge, the two vertices u and v are to be specified, and for a directed graph along with vertices, the start vertex and the end vertex should be known Figure 8.3(a) shows a graph $G(V, E)$ where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d)\}$ and the resultant graph after inserting the edge (c, d) is shown in Fig. 8.3(b) with $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d), (c, d)\}$.

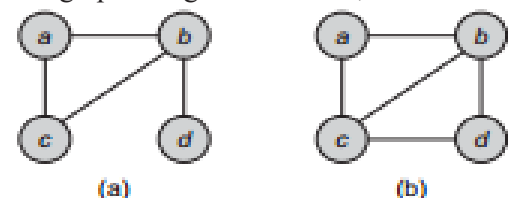


Fig. 8.3 Inserting an edge in a graph (a) Graph G (b) After inserting edge (c, d)

Delete Edge

The delete edge operation removes one edge from the graph. Let the graph G be $G(V, E)$. Now, deleting the edge (u, v) from G deletes the edge incident between vertices u and v and keeps the incident vertices u, v .

Figure 8.4(a) shows a graph $G(V, E)$ where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (b, d)\}$. The resultant graph after deleting the edge (b, d) is shown in Fig. 8.4(b) with $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c)\}$.

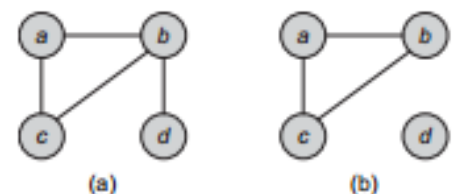


Fig. 8.4 Deleting edge in graph (a) Graph G (b) Graph after deleting the edge (b, d)

Graph traversal is also known as searching through a graph.

REPRESENTATION OF GRAPHS:

There are two standard representations of a graph given as follows:

1. Adjacency matrix (sequential representation) and
2. Adjacency list (linked representation)

Using these two representations, graphs can be realized using the adjacency matrix, adjacency list, or adjacency multi list.

Adjacency Matrix:

Adjacency matrix is a square, two-dimensional array with one row and one column for each vertex in the graph. An entry in row i and column j is 1 if there is an edge incident between vertex i and vertex j , and is 0 otherwise. If a graph is a weighted graph, then the entry 1 is replaced with the weight. It is one of the most common and simple representations of the edges of a graph; programs can access this information very efficiently. For a graph $G = (V, E)$, suppose $V = \{1, 2, \dots, n\}$. The adjacency matrix for G is a two dimensional $n \times n$ Boolean matrix A and can be represented as $A[i][j] = \{1 \text{ if there exists an edge } \langle i, j \rangle, 0 \text{ if edge } \langle i, j \rangle \text{ does not exist}\}$. The adjacency matrix A has a natural implementation as in the following: $A[i][j]$ is 1 (or true) if and only if vertex i is adjacent to vertex j . If the graph is undirected, then $A[i][j] = A[j][i] = 1$. If the graph is directed, we interpret 1 stored at $A[i][j]$, indicating that the edge from i to j exists and not indicating whether or not the edge from j to i exists in the graph. The graphs G_1 , G_2 , and G_3 of Fig. 8.5 are represented using the adjacency matrix in Fig. 8.6, among which G_2 is a directed graph.

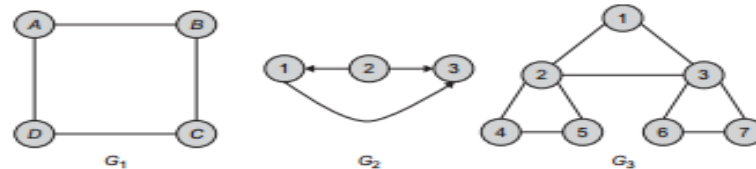


Fig. 8.5 Graphs G_1 , G_2 , and G_3

	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

G_1

	1	2	3
1	0	0	1
2	1	0	1
3	0	0	0

G_2

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	1	1	1	0	0
3	1	1	0	0	0	1	1
4	0	1	0	0	1	0	0
5	0	1	0	1	0	0	0
6	0	0	1	0	0	0	1
7	0	0	1	0	0	1	0

G_3

Fig. 8.6 Adjacency matrix for G_1 , G_2 , and G_3 of Fig. 8.5

For a weighted graph, the matrix A is represented as $A[i][j] = \{\text{weight} \quad \text{if the edge } \langle i, j \rangle \text{ exists}$

$0 \quad \text{if there exists no edge } \langle i, j \rangle\}$

Here, weight is the label associated with the edge of the graph. For example, Figs 8.7(a) and (b) show the weighted graph and its associated adjacency matrix.

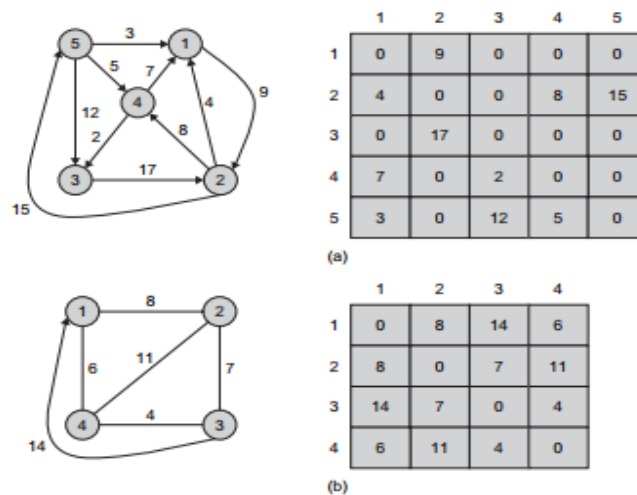


Fig. 8.7 Adjacency matrix (a) Directed weight graph and its adjacency matrix (b) Undirected weight graph and its adjacency matrix

ADJACENCY LIST:

In this representation, the n rows of the adjacency list are represented as n -linked lists, one list per vertex of the graph. The adjacency list for a vertex i is a list of all vertices adjacent to it. One way of achieving this is to go for an array of pointers, one per vertex. For example, we can represent the graph G by an array Head , where $\text{Head}[i]$ is a pointer to the adjacency list of vertex i . For list, each node of the list has at least two fields: vertex and link. The vertex field contains the vertex id, and the link field stores a pointer to the next node that stores another vertex adjacent to i . Figure 8.8(b) shows an adjacency list representation for a directed graph in Fig.

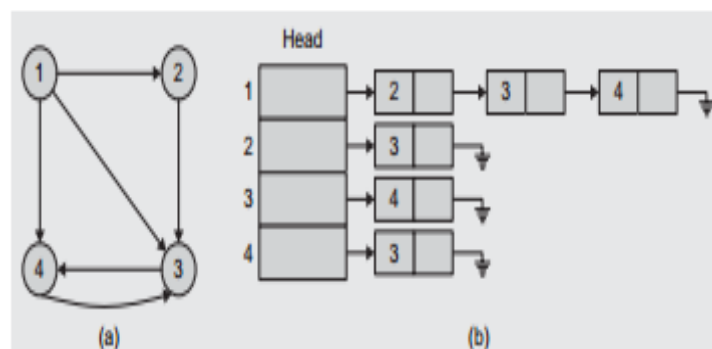


Fig. 8.8 Adjacency list representation (a) Graph G_1 (b) Adjacency list for G_1

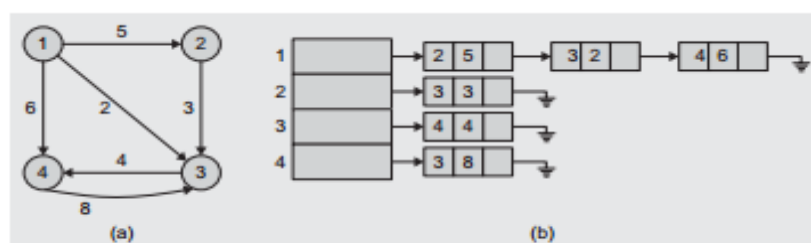


Fig. 8.9 Adjacency list of weighted graph (a) Weighted graph G_2 (b) Adjacency list of G_2

ADJACENCY MULTILIST:

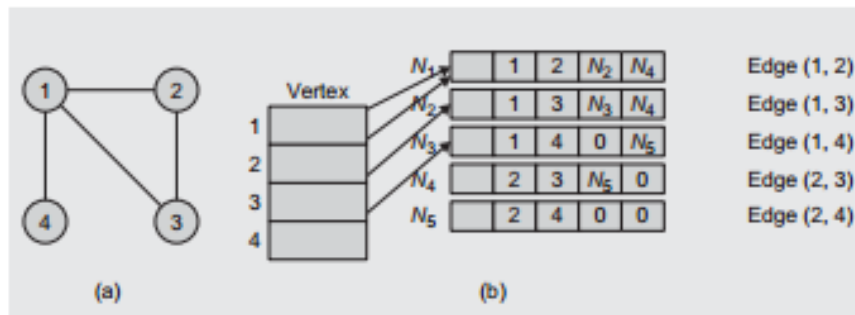


Fig. 8.10 Adjacency multilist (a) Graph G_1 (b) Adjacency multilist for G_1

For Fig. 8.10, the lists are as follows:

Vertex 1: $N_1 \rightarrow N_2 \rightarrow N_3$
Vertex 2: $N_1 \rightarrow N_4 \rightarrow N_5$
Vertex 3: $N_2 \rightarrow N_5$
Vertex 4: $N_3 \rightarrow N_5$

GRAPH TRAVERSAL:

Visiting all the vertices and edges in a systematic fashion called *graph traversal*. There are two types of Graph travels —**depth-first traversal** and **breadth-first traversal**. Traversal of a graph is commonly used to search a vertex or an edge through the graph; hence, it is also called a *search technique*. Consequently, depth-first and breadth-first traversals are popularly known as **depth-first search (DFS)** and **breadth-first search (BFS)**, respectively.

Depth-first Search:

In DFS, as the name indicates, from the currently visited vertex in the graph, we keep searching deeper whenever possible. All the vertices are visited by processing a vertex and its descendents before processing its adjacent vertices.

For non-recursive implementation, whenever we reach a node, we shall push it (vertex or node address) onto the stack. We would then pop the vertex, process it, and push all its adjacent vertices onto the stack. Suppose we have a directed graph G where all the vertices are initially marked as unvisited. In a graph, we can reach any vertex more than once through different paths. Hence, to assure that each vertex is visited once, we mark each as visited whenever it is processed. Let us use an array say `visited` for the same. Initially, all vertices are marked unvisited. Marking `visited[i]` to 0 indicates that the vertex i is unvisited. Whenever we push the vertex say j onto the stack, we mark it visited by setting its `visited[j]` to 1.

The recursive algorithm for DFS can be outlined as in Algorithm 8.1.

Algorithm 8.1 shows the recursive working of DFS of a graph.

When we need to show its equivalent non-recursive code, we need to use a stack. No recursive DFS can be implemented by using a stack for pushing all unvisited vertices adjacent to the one being visited and popping the stack to find the next unvisited vertex.

Consider the graph in Fig. 8.12(a) and its adjacency list in Fig. 8.12(b).

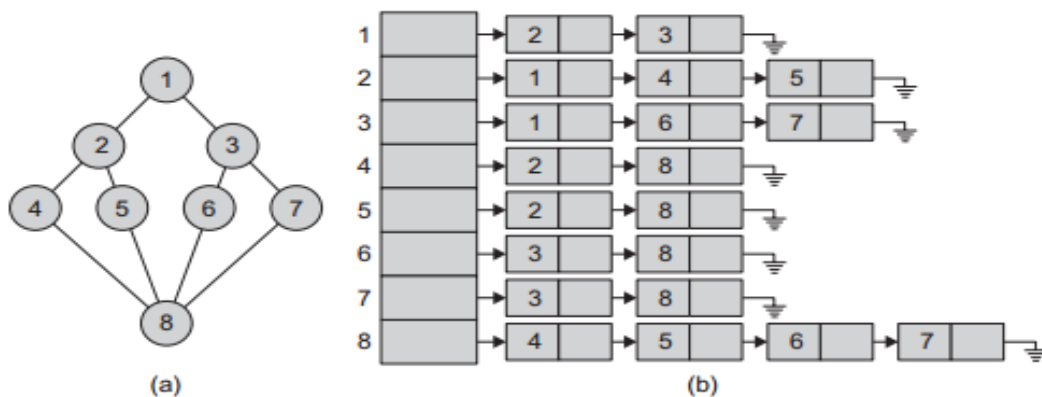


Fig. 8.12 Sample graph for traversal (a) Graph G (b) Adjacency list representation of G

The DFS traversal will be 1, 2, 4, 8, 5, 6, 3, 7. Another possible traversal could be 1, 3, 7, 8, 6, 5, 2, 4. Let us now consider the graph in Fig. 8.13.

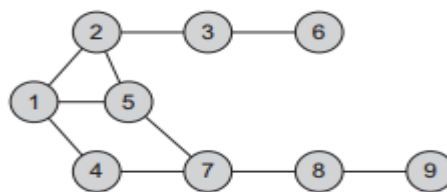
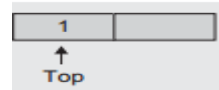


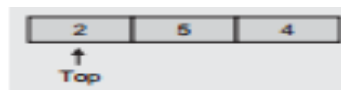
Fig. 8.13 Sample graph

Let us traverse the graph using a non-recursive algorithm that uses stack. Let 1 be the start vertex. Note that the stack is empty initially.

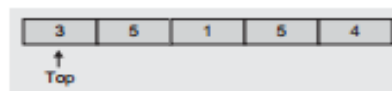
1. Initially, $V = \text{set of visited vertices} = \emptyset$. Push 1 onto the stack.



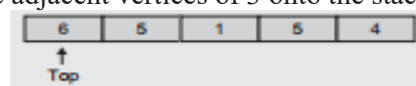
2. As the stack is not empty, $\text{vertex} = \text{pop}()$; we get 1. As 1 is not visited, mark it as visited. Now $V = \{1\}$. Push all the adjacent vertices of 1 onto the stack. Since the stack is not empty, $\text{vertex} = \text{pop}()$; we get 2.



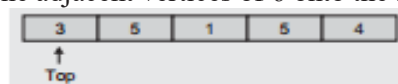
3. As 2 is not visited, mark it as visited, and now $V = \{1, 2\}$. Then, push all the adjacent vertices of 2.



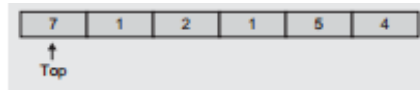
4. Since the stack is not empty, $\text{vertex} = \text{pop}()$; we get 3. As 3 is not visited, mark it as visited. Now $V = \{1, 2, 3\}$. We then push all the adjacent vertices of 3 onto the stack.



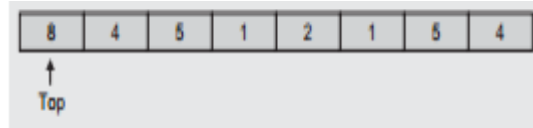
5. Since the stack is not empty, $\text{vertex} = \text{pop}()$; we get 6. As 6 is not visited, mark it as visited. Now $V = \{1, 2, 3, 6\}$. We then push all the adjacent vertices of 6 onto the stack.



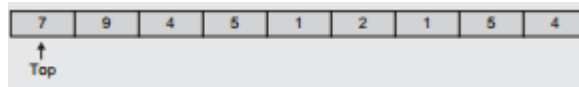
6. Since the stack is not empty, $\text{vertex} = \text{pop}()$; we get 3. As 3 is visited, pop again $\text{vertex} = \text{pop}()$; we then get 5. As 5 is not visited, mark it as visited. Now $V = \{1, 2, 3, 6, 5\}$. Push all the adjacent vertices onto the stack.



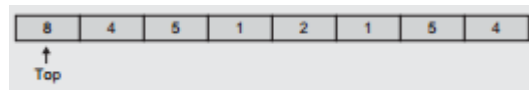
7. As the stack is not empty, $\text{vertex} = \text{pop}()$; we get 7, which is not visited. Hence, $V = \{1, 2, 3, 6, 5, 7\}$; we now push all the adjacent vertices of 7 onto the stack.



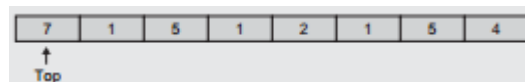
8. As the stack is not empty, $\text{vertex} = \text{pop}()$; we get 8, which is not visited. Hence, $V = \{1, 2, 3, 6, 5, 7, 8\}$. Push all the adjacent vertices of 8 onto the stack.



9. As the stack is not empty, $\text{vertex} = \text{pop}() = 7$, which is visited; $\text{vertex} = \text{pop}() = 9$, which is not visited. Hence, $V = \{1, 2, 3, 6, 5, 7, 8, 9\}$. Push all the adjacent vertices of 9 onto the stack.



10. As the stack is not empty, $\text{vertex} = \text{pop}() = 8$, which is visited; so again $\text{vertex} = \text{pop}() = 4$, which is not visited. Hence, $V = \{1, 2, 3, 6, 5, 7, 8, 9, 4\}$. Push all the adjacent vertices of 4 onto the stack.



11. The stack is not empty. So the following operations yield:

$\text{vertex} = \text{pop}()$ we get 7, visited
 $\text{vertex} = \text{pop}()$ we get 1, visited
 $\text{vertex} = \text{pop}()$ we get 5, visited
 $\text{vertex} = \text{pop}()$ we get 1, visited
 $\text{vertex} = \text{pop}()$ we get 2, visited
 $\text{vertex} = \text{pop}()$ we get 1, visited
 $\text{vertex} = \text{pop}()$ we get 5, visited
 $\text{vertex} = \text{pop}()$ we get 4, visited

12. The stack is now empty, Hence, we stop. The set $V = \{1, 2, 3, 6, 5, 7, 8, 9, 4\}$ represents the order in which they are visited. Hence, the DFS of the graph (Fig. 8.13) gives the sequence as 1, 2, 3, 6, 5, 7, 8, 9, and 4. This is shown in Fig. 8.14.

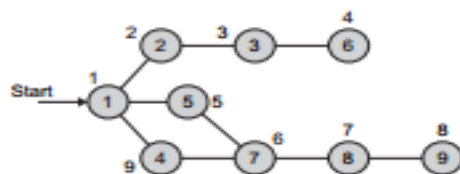


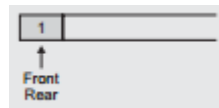
Fig. 8.14 Depth-first traversal for graph in Fig. 8.13

Breadth-first Search:

Another systematic way of visiting the vertices is the breadth-first search (BFS). The BFS differs from DFS in a way that all the unvisited vertices adjacent to i are visited after visiting the start vertex i and marking it visited. Next, the unvisited vertices adjacent to these vertices are visited and so on until the entire graph has been traversed. The approach is called 'breadth-first' because from the vertex i that we visit, we search as broadly as possible by next visiting all the vertices adjacent to i . For example, the BFS of the graph of Fig. 8.13 results in visiting the nodes in the following order: 1, 2, 3, 4, 5, 6, 7, and 8. This search algorithm uses a queue to store the vertices of each level of the graph as and when they are visited. These vertices are then taken out from the queue in sequence, that is, first in first out (FIFO), and their adjacent vertices are visited until all the vertices have been visited.

Let us traverse the graph using a non-recursive algorithm that uses a queue. Let 1 be the start vertex. Initially, the queue is empty, and the initial set of visited vertices, $V = \phi$.

1. Add 1 to the queue. Mark 1 as visited. $V = \{1\}$.



2. As the queue is not empty, $\text{vertex} = \text{delete}()$ from queue, and we get 1. Add all the un-visited adjacent vertices of 1 to the queue. In addition, mark them as visited. Now, $V = \{1, 2, 5, 4\}$



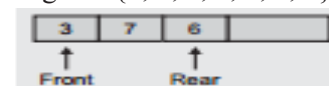
3. As the queue is not empty, $\text{vertex} = \text{delete}()$ and we get 2. Add all the adjacent, un-visited vertices of 2 to the queue and mark them as visited. Now $V = \{1, 2, 5, 4, 3\}$.



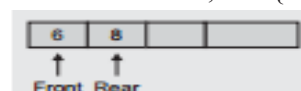
4. As the queue is not empty, $\text{vertex} = \text{delete}()$ from queue, and we get 5. Now, add all the adjacent, un-visited vertices adjacent to 5 to the queue and mark them as visited. Now, $V = \{1, 2, 5, 4, 3, 7\}$.



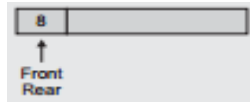
5. As the queue is not empty, $\text{vertex} = \text{delete}()$ from queue, and we get 4. Now, add all the adjacent, not visited vertices adjacent to 4 to the queue. The vertices 1 and 7 are adjacent to 4 and hence are already visited. Now the next element we get from the queue is 3. Now, we add all the un-visited vertices adjacent to 3 to the queue, making $V = \{1, 2, 5, 4, 3, 7, 6\}$



6. As the queue is not empty, $\text{vertex} = \text{delete}()$ and we get 7. Add all the adjacent, un- visited vertices of 7 to the queue and mark them as visited. Now, $V = \{1, 2, 5, 4, 3, 7, 6, 8\}$



7. As the queue is not empty, $\text{vertex} = \text{delete}()$, and we get 6. Then, add all the un-visited adjacent vertices of 6 to the queue and mark them as visited. Now $V = \{1, 2, 5, 4, 3, 7, 6, 8\}$



8. As queue is not empty, $\text{vertex} = \text{delete}()$ and we get 8. Add its adjacent un-visited vertices to the queue and mark them as visited. $V = \{1, 2, 5, 4, 3, 7, 6, 8, 9\}$.



9. As the queue is not empty, $\text{vertex} = \text{delete}() = 9$. Here, note that no adjacent vertices of 9 are un-visited. 10. As the queue is empty, we stop. The sequence in which the vertices are visited by the BFS is 1, 2, 5, 4, 3, 7, 6, 8, 9 This is represented in Fig. 8.18.

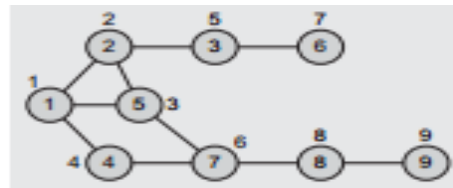


Fig. 8.18 Breadth-first search sequence for the graph in Fig. 8.13

SPANNING TREE:

DEF: A tree is a connected graph with no cycles. A spanning tree is a sub-graph of G that has all vertices of G and is a tree. A minimum spanning tree of a weighted graph G is the spanning tree of G whose edges sum to minimum weight. There can be more than one minimum spanning tree for a graph. Figure 8.19 shows a graph, one of its spanning trees, and a minimum spanning tree.

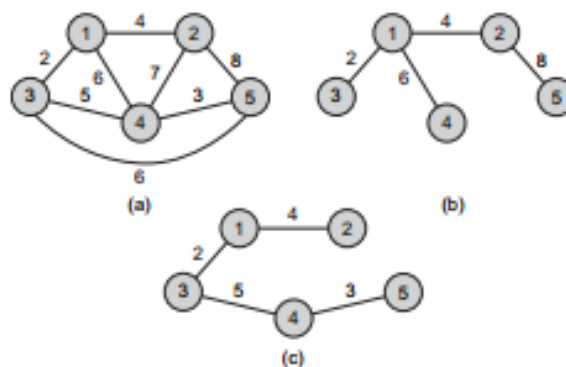


Fig. 8.19 Spanning trees (a) Graph (b) Spanning tree (c) Minimum spanning tree

There are two popular methods used to compute the minimum spanning tree of a graph is

1. Prim's algorithm
2. Kruskal's algorithm

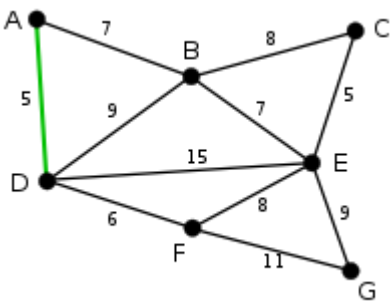
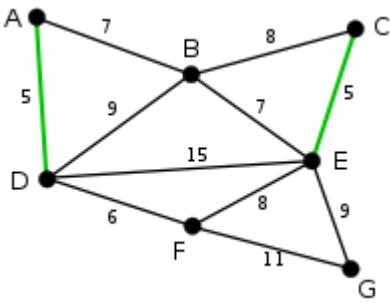
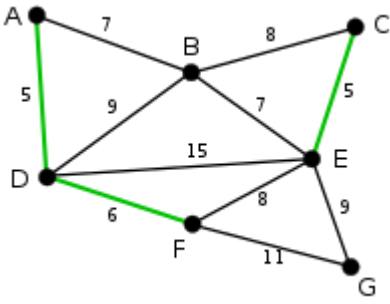
1) Prim's algorithm:

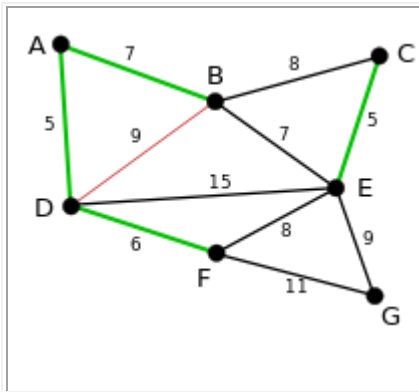
Prim's algorithm is a greedy **algorithm** that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

2) Kruskal's algorithm:

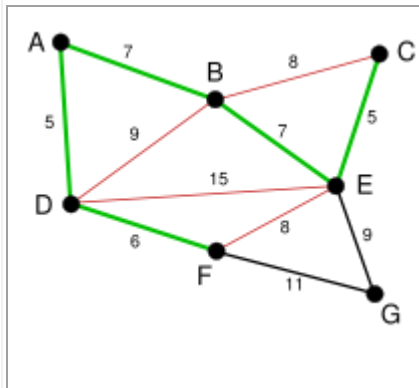
Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest*

Example

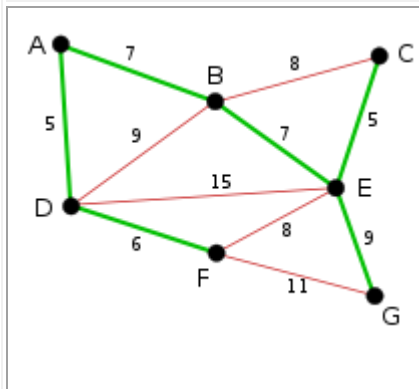
Image	Description
	<p>AD and CE are the shortest edges, with length 5, and AD has been arbitrarily chosen, so it is highlighted.</p>
	<p>CE is now the shortest edge that does not form a cycle, with length 5, so it is highlighted as the second edge.</p>
	<p>The next edge, DF with length 6, is highlighted using much the same method.</p>



The next-shortest edges are **AB** and **BE**, both with length 7. **AB** is chosen arbitrarily, and is highlighted. The edge **BD** has been highlighted in red, because there already exists a path (in green) between **B** and **D**, so it would form a cycle (**ABD**) if it were chosen.



The process continues to highlight the next-smallest edge, **BE** with length 7. Many more edges are highlighted in red at this stage: **BC** because it would form the loop **BCE**, **DE** because it would form the loop **DEBA**, and **FE** because it would form **FEBAD**.



Finally, the process finishes with the edge **EG** of length 9, and the minimum spanning tree is found.

HASHING:

Def:

Hashing is a method of directly computing the address of the record with the help of a key by using a suitable mathematical function called the *hash function*. A *hash table* is an array-based structure used to store <key, information> pairs.

Hashing is the process of indexing and retrieving element (data) in a data structure to provide faster way of finding the element using the hash key.

Hash Table:

Hash table is an array which maps a key (data) into the datastructure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity

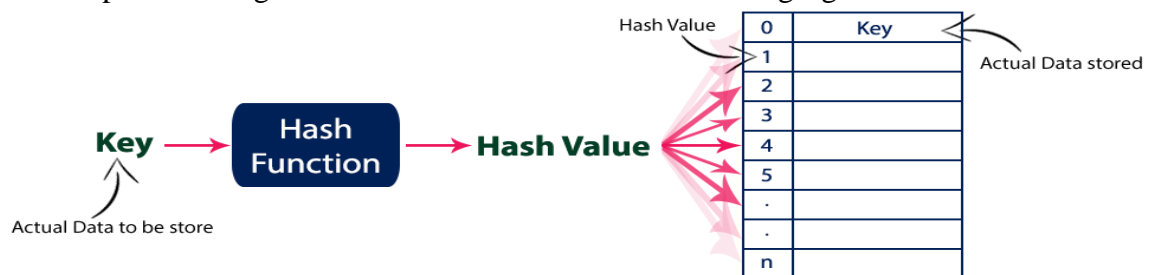
Hash Function:

Hash function is a function that maps a key in the range $[0 \text{ to } \text{Max} - 1]$, the result of which is used as an index (or address) in the hash table for storing and retrieving records

One more way to define a hash function is as the function that transforms a key into an address.

The address generated by a hashing function is called the *home address*. All home addresses refer to a particular area of the memory called the *prime area*.

Basic concept of hashing and hash table is shown in the following figure...



Bucket :A bucket is an index position in a hash table that can store more than one record. Tables 11.1 and 11.2 show a bucket of size 1 and size 2, respectively. When the same index is mapped with two keys, both the records are stored in the same bucket. The assumption is that the buckets are equal in size.

Table 11.1 Table with bucket size 1

Index	Bucket of size 1
0	Alka
1	Bindu
2	
3	Deven
4	Ekta
5	
6	Govind
⋮	⋮
13	Monika
⋮	⋮
18	Sharmila
⋮	⋮
25	Zinat

Table 11.2(a) Table with bucket size 2

Index	Bucket of size 2	
0	Alka	Abhay
1	Bindu	Babali
2		
3	Deepa	Deven
4	Ekta	Esha
5		
6	Govind	Gopal
⋮	⋮	⋮
13	Monika	Meera
⋮	⋮	⋮
18	Sharmila	Sindhu
⋮	⋮	⋮
25	Zinat	Ziya

Probe: Each action of address calculation and check for success is called as a *probe*.

Collision: The result of two keys hashing into the same address is called collision.

Synonym: Keys that hash to the same address are called synonyms.

Overflow: The result of many keys hashing to a single address and lack of room in the bucket is known as an overflow. Collision and overflow are synonymous when the bucket is of size 1.

Open or external hashing When we allow records to be stored in potentially unlimited space, it is called as *open* or *external hashing*.

Closed or internal hashing When we use fixed space for storage eventually limiting the number of records to be stored, it is called as *closed* or *internal hashing*.

Hash function Hash function is an arithmetic function that transforms a key into an address which is used for storing and retrieving a record.

Perfect hash function The hash function that transforms different keys into different addresses is called a *perfect hash function*. The worth of a hash function depends on how well it avoids collision.

HASH FUNCTIONS:

To store a record in a hash table, a hash function is applied to the key of the record being stored, returning an index within the range of the hash table. The record is stored at that index position, if it is empty. With direct addressing, a record with key K is stored in slot K . With hashing, this record is stored at the location $\text{Hash}(K)$, where $\text{Hash}(K)$ is the function. The hash function $\text{Hash}(K)$ is used to compute the slot for the key K .

Good Hash Function

A good hash function is one which satisfies the assumption. that any given record is equally likely to hash into any of the slots, independent of whether any other record has been already hashed to it or not.

This assumption is known as *simple uniform hashing*.

Features of a Good Hashing Function

1. Addresses generated from the key are uniformly and randomly distributed.
2. Small variations in the value of the key will cause large variations in the record addresses to distribute records (with similar keys) evenly.
3. The hashing function must minimize the occurrence of collision.

There are many methods of implementing hash functions:

1. Division Method

One of the required features of the hash function is that the resultant index must be within the table index range. One simple choice for a hash function is to use the modulus division indicated as MOD (the operator % in C/C++). The function MOD returns the remainder when the first parameter is divided by the second parameter. The result is negative only if the first parameter is negative and the parameters must be integers. The function returns an integer. If any parameter is NULL, the result is NULL.

$\text{Hash}(\text{Key}) = \text{Key} \% M$ Key is divided by some number M , and the remainder is used as the hash address.

2. Multiplication Method:

3. Extraction Method
4. Mid-square Hashing
5. Folding Technique
6. Rotation
7. Universal Hashing

COLLISION RESOLUTION STRATEGIES:

No hash function is perfect. If $\text{Hash}(\text{Key1}) = \text{Hash}(\text{Key2})$, then Key1 and Key2 are synonyms and if bucket size is 1, we say that collision has occurred. As a consequence, we have to store the record Key2 at some other location. A search is made for a bucket in which a record is stored containing Key2, using one of the several collision resolution strategies. The collision resolution strategies are as follows:

1. Open addressing
 - (a) Linear probing
 - (b) Quadratic probing
 - (c) Double hashing
 - (d) Key offset
2. Separate chaining (or linked list)
3. Bucket hashing (defers collision but does not prevent it)

The most important factors to be taken care of to avoid collision are the table size and choice of the hash function.