Chapter-5
Basic Traversal and Search Techniques

5.1 Techniques for Binary Trees

Binary Tree

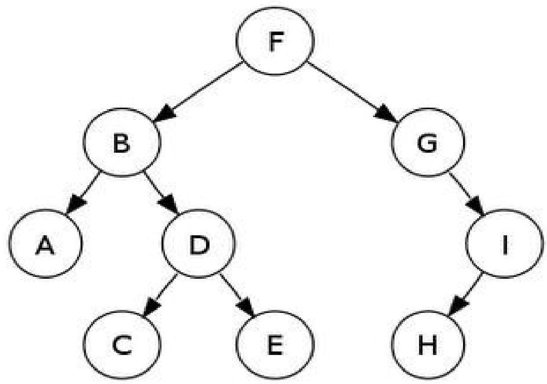
A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left sub tree and the right sub tree.
In a traversal of a binary tree, each element of the binary tree is visited exactly at once. During the visiting of an element, all actions like clone, display, evaluate the operator etc is taken with respect to the element. When traversing a binary tree, we need to follow linear order i.e. L, D, R where
L->Moving left
D->printing the data
R->moving right

We have three traversal techniques on binary tree. They are

- *In order*
- *Post order*
- *Pre order*

Examples

For fig: 1



In order: A-B-C-D-E-F-G-H-I
Post order: A-C-E-D-B-H-I-G-F
Pre order: F-B-A-D-C-E-G-I-H

Preorder, post order and in order algorithms

Algorithm preorder(x)

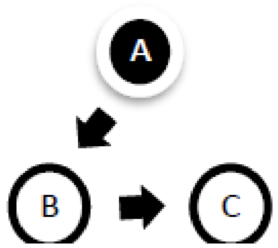
Input: x is the root of a sub tree.

1. If x ≠ NULL

2. **Then** output key(x);

3. Preorder (left(x));

4. Preorder (right(x));



Algorithm postorder(x)

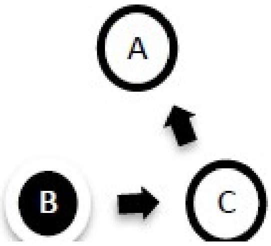
Input: x is the root of a subtree

1. **If** x ≠ NULL

2. **Then** postorder(left(x));;

3. Postorder(right(x));

4. Outputkey(x);



Algorithm inorder(x)

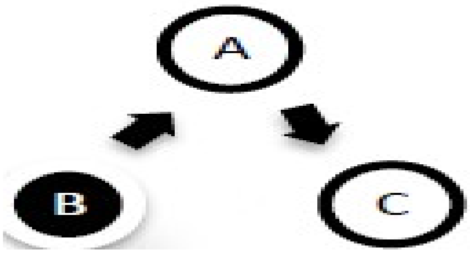
Input: x is the root of a subtree

1. If x≠ null

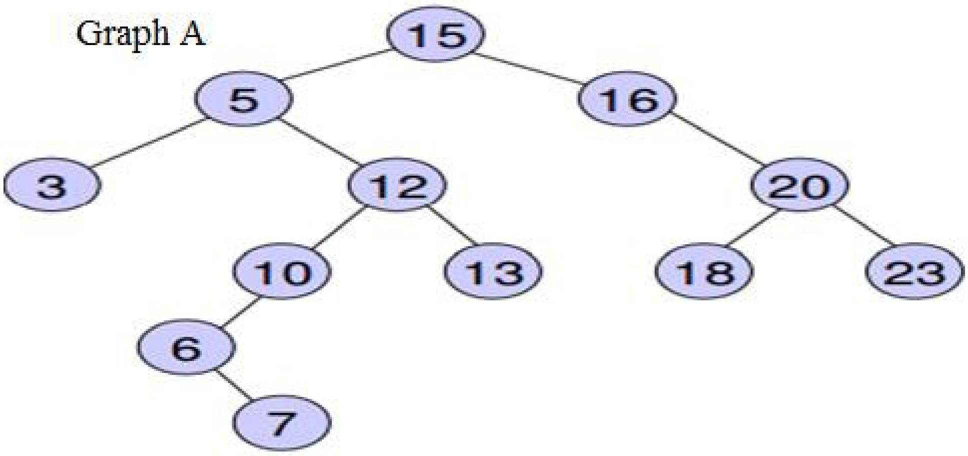
2. **Then** inorder(left(x));

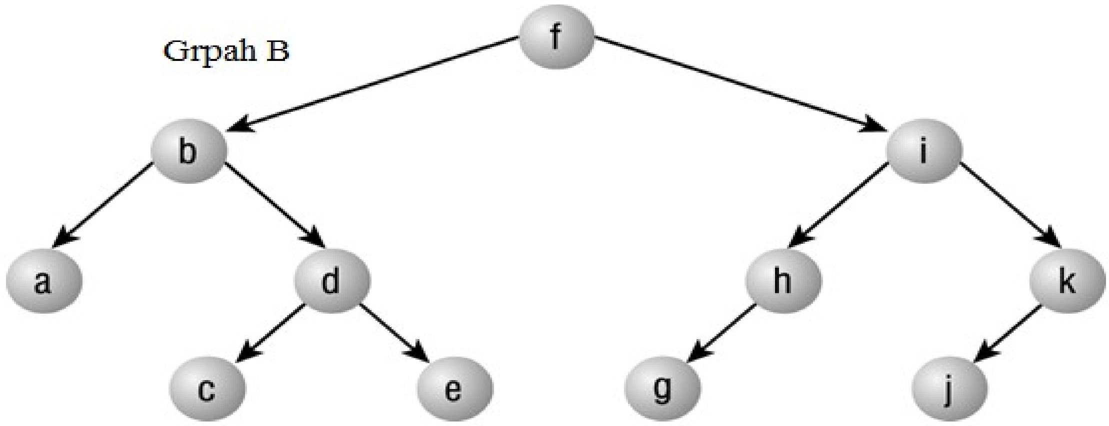
3. Outputkey(x);

4. Inorder(right(x));



Exercises



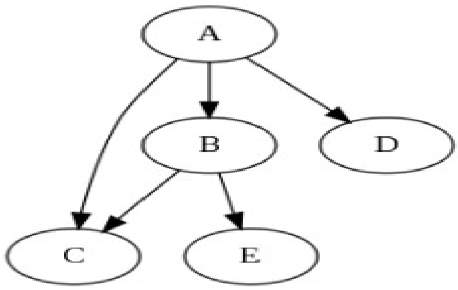


5.2 Techniques for Graphs

Graph: The sequence of edges that connected two vertices.
A graph is a pair (*V, E*), where
V is a set of nodes, called vertices
E is a collection (can be duplicated) of pairs of vertices, called edges
Vertices and edges are data structures and store elements.

Types of graphs: Graphs are of three types.

a. *Directed/Undirected:* In a directed graph the direction of the edges must be considered



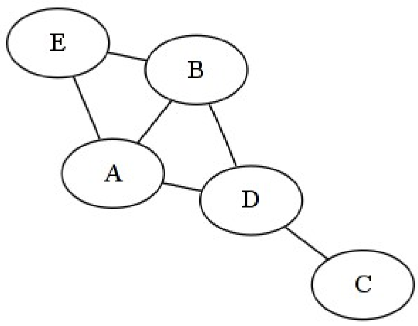


Fig 5.1

Fig 5.2

b. *Weighted/ Unweighted*: A weighted graph has values on its edge.

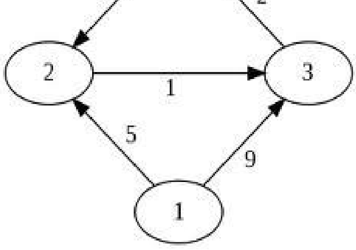


Fig 5.3

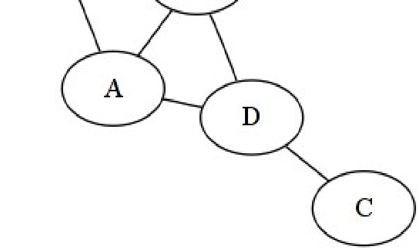
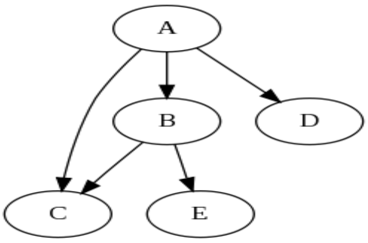
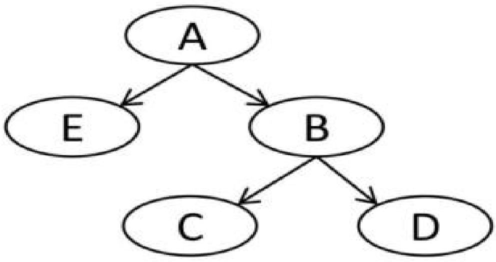


Fig 5.4

c. *Cyclic/Acyclic*: A **cycle** is a path that begins and ends at same vertex and A graph with no cycles is **acyclic.**

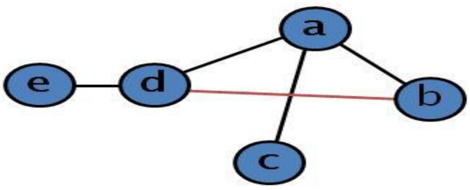




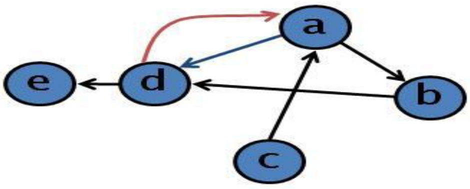
Representation of graphs

Graphs can be represented in three ways

(i) **Adjacency Matrix:** A *V* x *V* array, with matrix[*i*][*j*] storing whether there is an edge between the *ith* vertex and the *jth* vertex. This matrix is also called as "Bit matrix" or "Boolean Matrix"

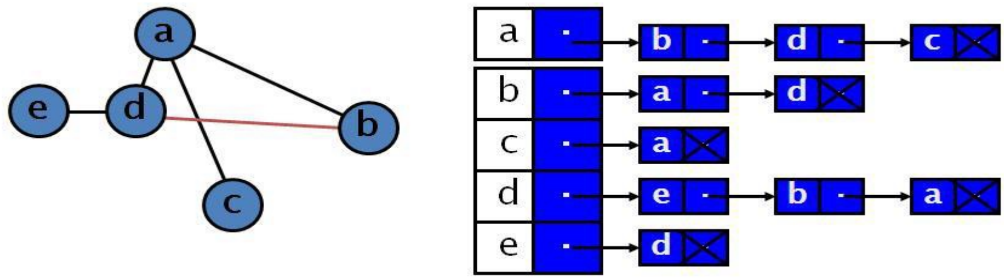


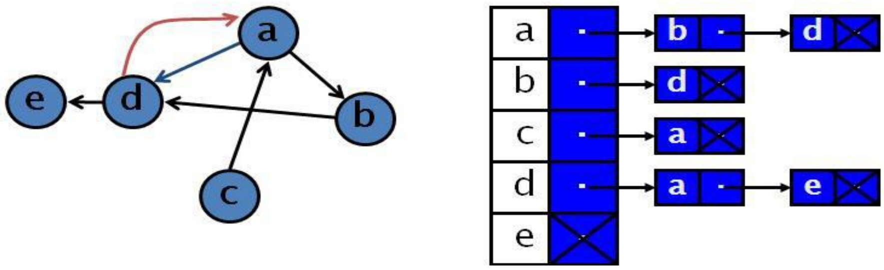
	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	1	0
c	1	0	0	0	0
d	1	1	0	0	1
e	0	0	0	1	0



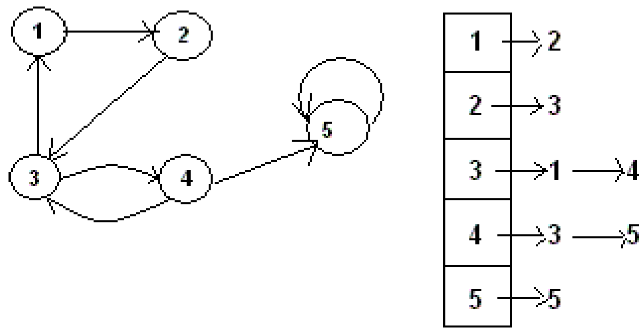
	a	b	c	d	e
a	0	1	0	1	0
b	0	0	0	1	0
c	1	0	0	0	0
d	1	0	0	0	1
e	0	0	0	0	0

(ii) **Adjacency list:** One linked list per vertex, each storing directly reachable vertices .





(iii) Linked List or Edge list:



Graph traversal techniques

“The process of traversing all the nodes or vertices on a graph is called graph traversal”.

We have two traversal techniques on graphs

- DFS
- BFS

Depth First Search

The DFS explore each possible path to its conclusion before another path is tried. In other words go as a far as you can (if u don’t have a node to visit), otherwise, go back and try another way. Simply it can be called as “backtracking”.

Steps for DFS

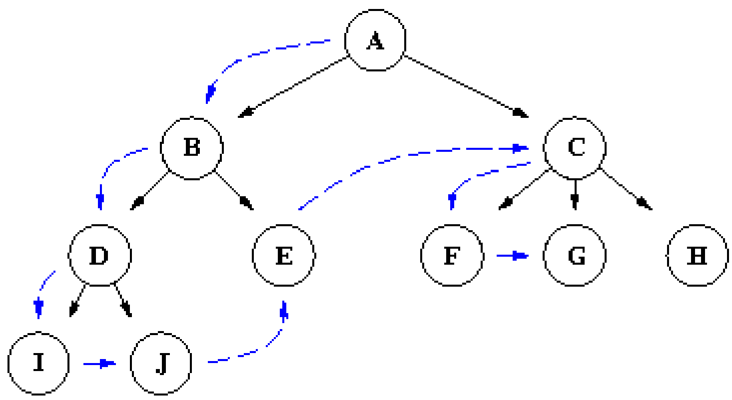
(i) Select an unvisited node ‘v’ visits it and treats it as the current node.

(ii) Find an unvisited neighbor of current node, visit it and make it new current node

(iii) If the current node has no unvisited neighbors, backtrack to its parent and make it as a new current node

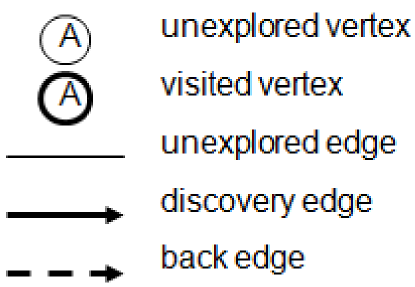
(iv) Repeat steps 2 and 3 until no more nodes can be visited

(v) Repeat from step 1 for remaining nodes also.



Implementation of DFS

DFS (Vertex)
{
Mark **u** as visiting
For each vertex V directly reachable from **u**
If **v** is unvisited
DFS (**v**)
}



Unexplored vertex: The node or vertex which is not yet visited.

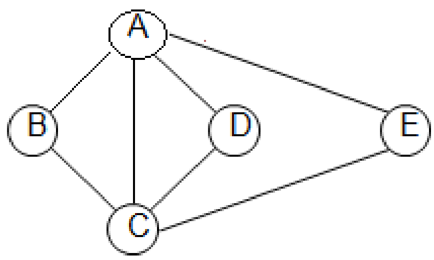
Visited vertex: The node or vertex which is visited is called 'visited vertex' i.e. can be called as "current node".

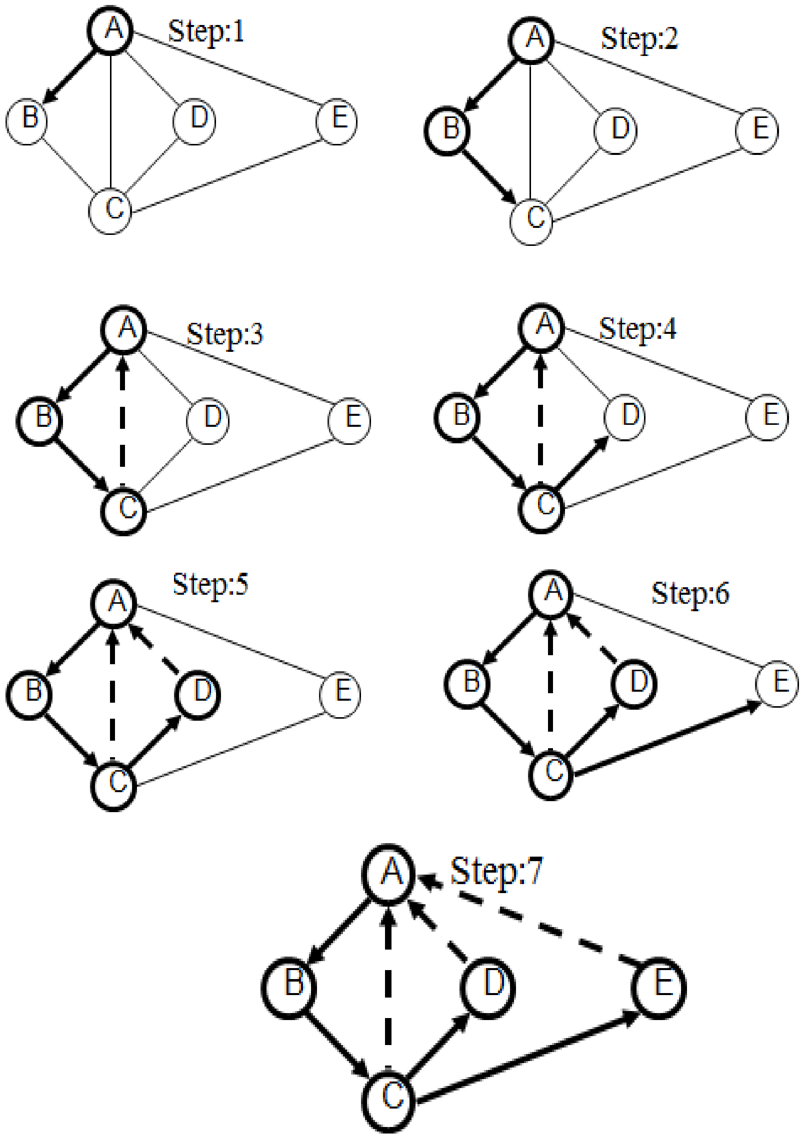
Unexplored edge: The edge or path which is not yet traversed.

Discovery edge: It is opposite to unexplored edge, the path which is already traversed is known as discovery edge.

Back edge: If the current node has no unvisited neighbors we need to backtrack to its parent node. The path used in back tracking is called back edge.

For the following graph the steps for tracing are as follows:

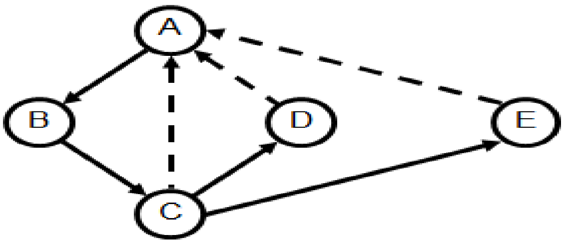




Properties of DFS

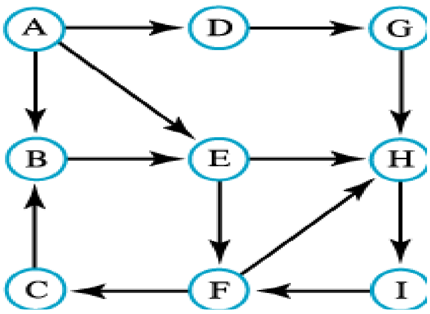
i) DFS (G, *v*) visits all the vertices and edges in the connected component of *v*.

ii) The discovery edges labeled by DFS (G, *v*) form a spanning tree of the connected component of *v*.



Tracing of graph using Depth First Search

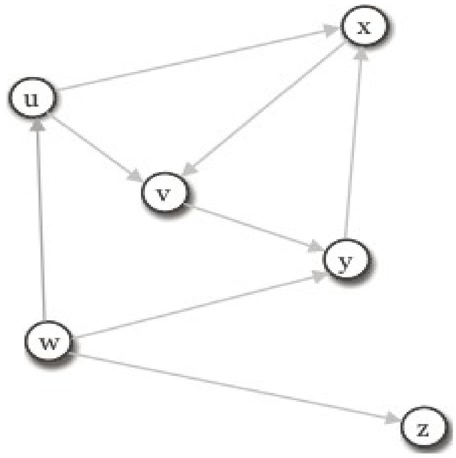
(a)



topVertex	nextNeighbor	Visited vertex	vertexStack (top to bottom)	traversalOrder (front to back)
		A	A	A
A			A	
	B	B	BA	AB
B			BA	
	E	E	EBA	ABE
E			EBA	
	F	F	FEBA	ABEF
F			FEBA	
	C	C	CFEBA	ABEFC
C			FEBA	
F			FEBA	
	H	H	HFEBA	ABEFCH
H			HFEBA	
	I	I	IHFEBA	ABEFCHI
I			HFEBA	
H			FEBA	
F			EBA	
E			BA	
B			A	
A			A	
	D	D	DA	ABEFCHID
D			DA	
	G		GDA	ABEFCHIDG
G			DA	
D			A	
A			*empty*	ABEFCHIDG

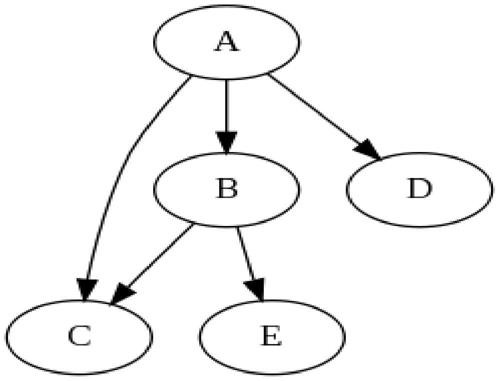
Exercise

1.



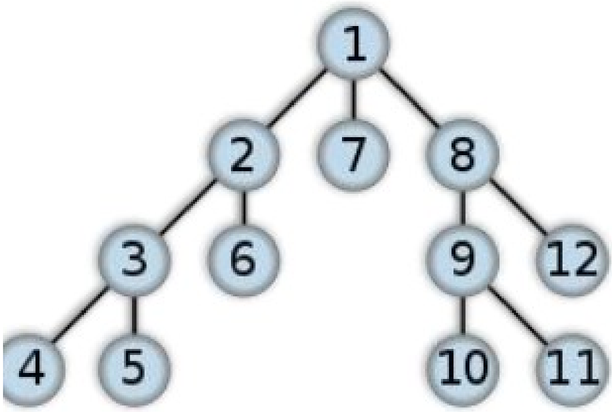
Depth: W-U-V-Y-X-Z

2.



Depth: A-B-C-E-D

3



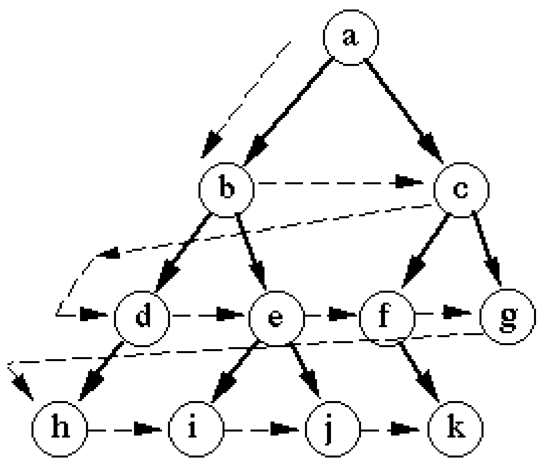
Depth: 1-2-3-4-5-6-7-8-9-10-11-12.

5.3 Breadth First Search

It is one of the simplest algorithms for searching or visiting each vertex in a graph. In this method each node on the same level is checked before the search proceeds to the next level. BFS makes use of a queue to store visited vertices, expanding the path from the earliest visited vertices
Breadth: a-b-c-d-e-f-g-h-i-j-k

Steps for BFS:

1. Mark all the vertices as unvisited.
2. Choose any vertex say ‘v’, mark it as visited and put it on the end of the queue.
3. Now, for each vertex on the list, examine in same order all the vertices adjacent to ‘v’
4. When all the unvisited vertices adjacent to v have been marked as visited and put it on the end (rear of the queue) of the list.
5. Remove a vertex from the front of the queue and repeat this procedure.
6. Continue this procedure until the list is empty.

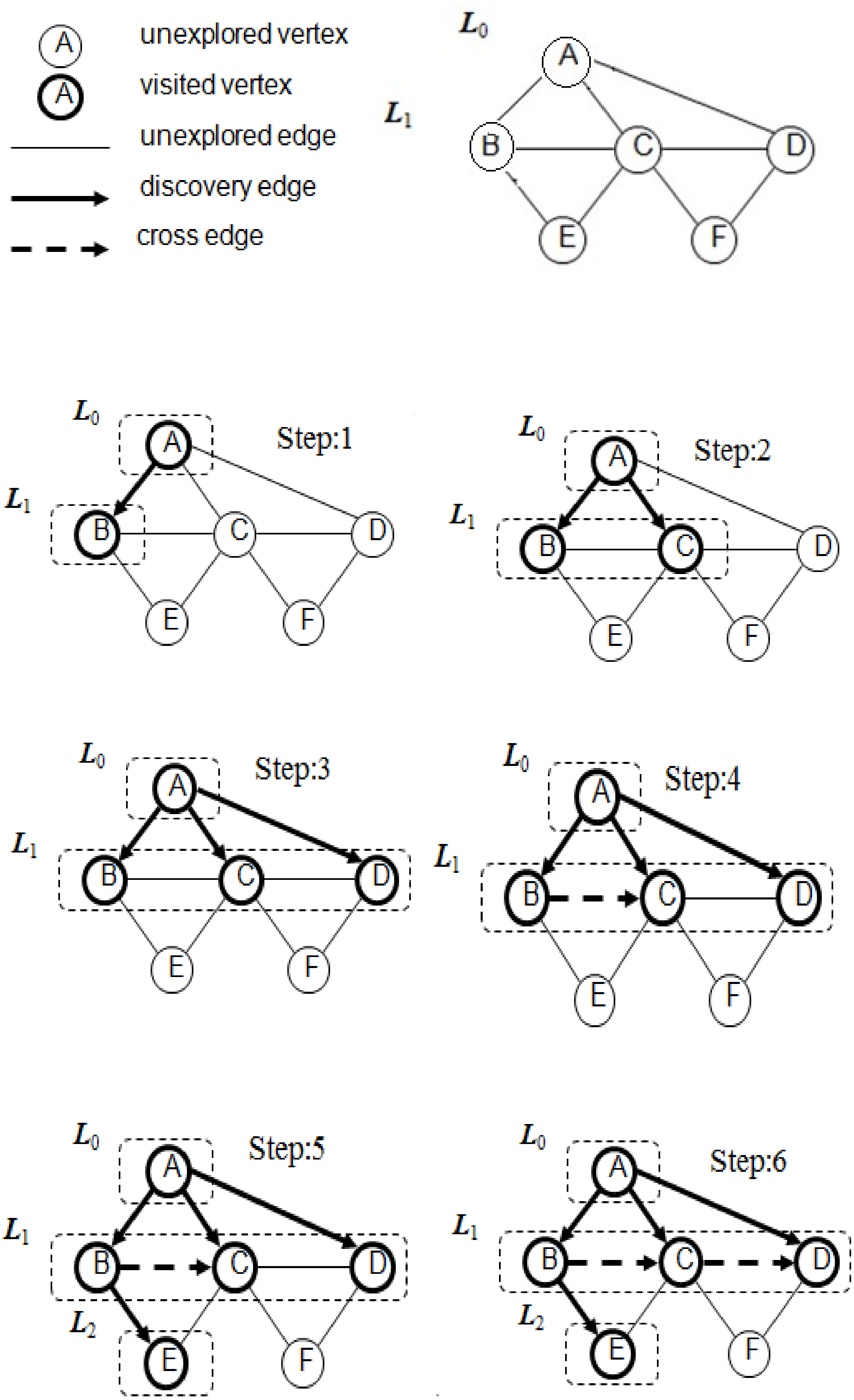


Breadth-first search

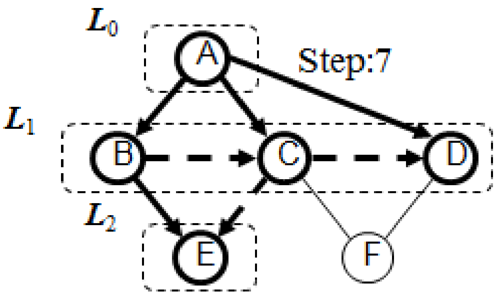
Implementation of BFS

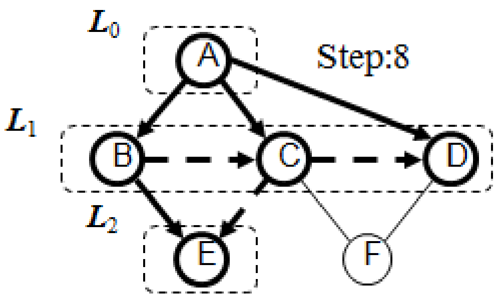
While queue Q not empty
De queue the first vertex **u** from Q
For each vertex **v** directly reachable from **u**
If **v** is unvisited
En queue **v** to Q
Mark **v** as visited
৲ Initially all vertices except the start vertex are marked as unvisited and the queue contains the start vertex only.

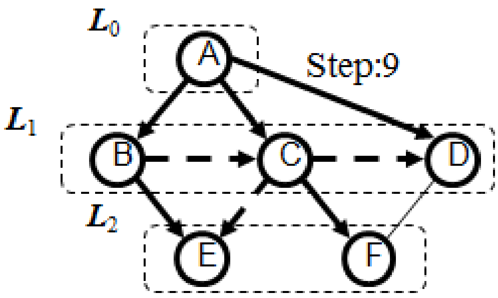
Explored vertex: A vertex is said to be explored if all the adjacent vertices of **v** are visited.
Example 1: Breadth first search for the following graph:

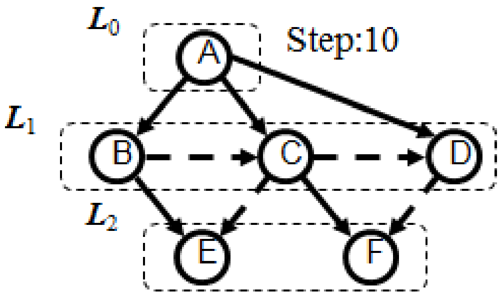


unexplored vertex
visited vertex
unexplored edge
discovery edge
cross edge
L_0
L_1
L_2
Step:1
Step:2
Step:3
Step:4
Step:5
Step:6





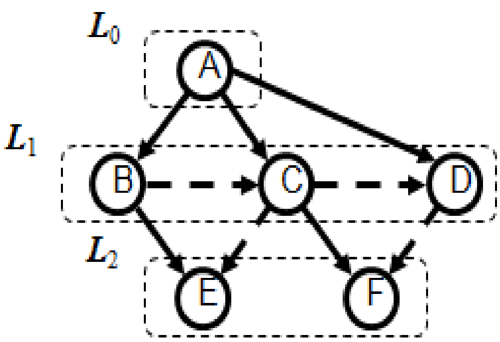




Properties of BFS

Notation: ***Gs*** (connected component of ***s***)

i) ***BFS*** (***G, s***) visits all the vertices and edges of ***Gs***
ii) The discovery edges labeled by ***BFS*** (***G, s***) form a spanning tree ***Ts*** of ***G***
iii) For each vertex ***v*** in ***Li***
a. The path of ***Ts*** from ***s*** to ***v*** has ***i*** edges
b. Every path from ***s*** to ***v*** in ***Gs*** has at least ***i*** edges.

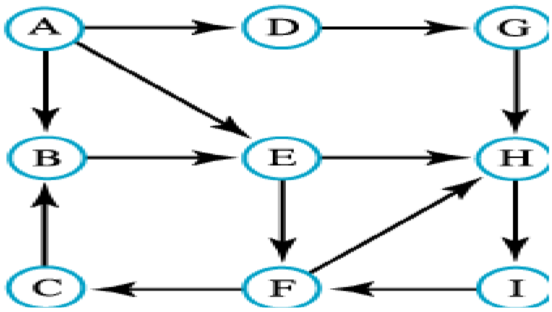


Complexity of BFS

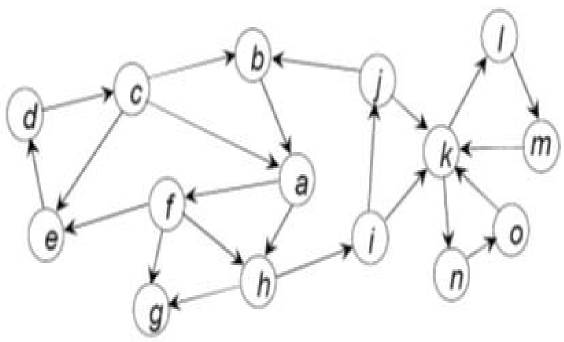
Step1: read a node from the queue O (v) times.
Step2: examine all neighbors, i.e. we examine all edges of the currently read node. Not oriented graph: 2*E edges to examine
*Hence the complexity of BFS is O (V + 2*E)*

Tracing of graph using Breadth first search:

(a)

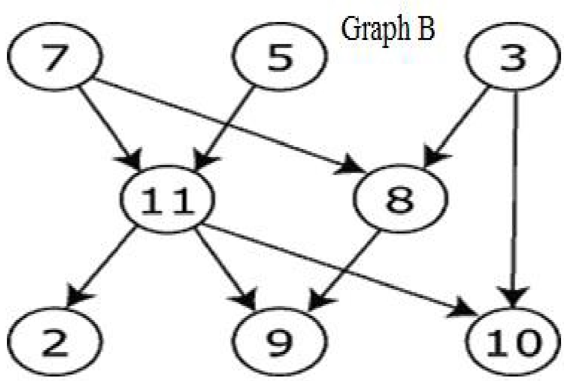


frontVertex	nextNeighbor	Visited vertex	vertexQueue	traversalOrder
		A	A	A
A			*empty*	
	B	B	B	A B
	D	D	B D	A B D
	E	E	B D E	A B D E
B			D E	
D			E	
	G	G	E G	A B D E G
E			G	
	F	F	G F	A B D E G F
	H	H	G F H	A B D E G F H
G			F H	
F			H	
	C	C	H C	A B D E G F H C
H			C	
	I	I	C I	A B D E G F H C I
C			I	
I			*empty*	



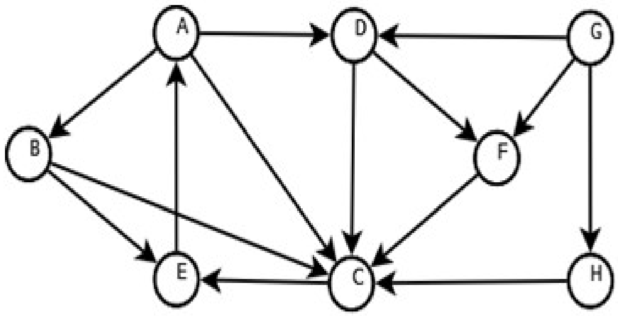
BFS: 7-11-8-2-9-10-5-3

BFS: a f h e g i d j k c l n b m o



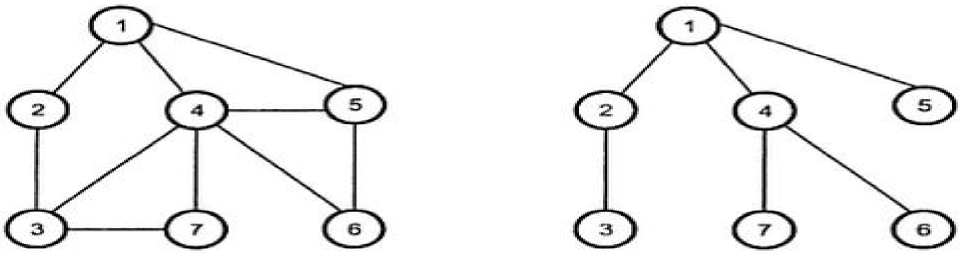
BFS: A-B-C-D-E-F-G-H

Graph:C



5.4 Connected Components and Spanning Trees

***Connected component*:** If G is connected undirected graph, then we can visit all the vertices of the graph in the first call to BFS. The sub graph which we obtain after traversing the graph using BFS represents the connected component of the graph.



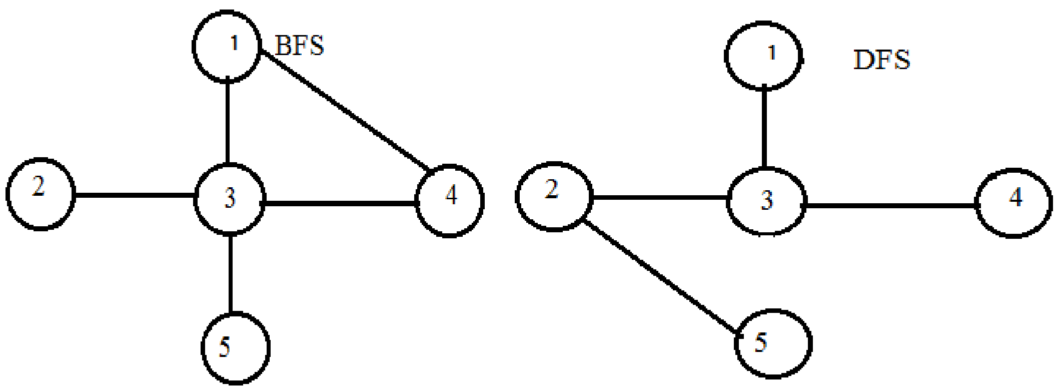
Graph G

Connected component of G obtained by breadth first search

Thus BFS can be used to determine whether G is connected. All the newly visited vertices on call to BFS represent the vertices in connected component of graph G. The sub graph formed by theses vertices make the connected component.

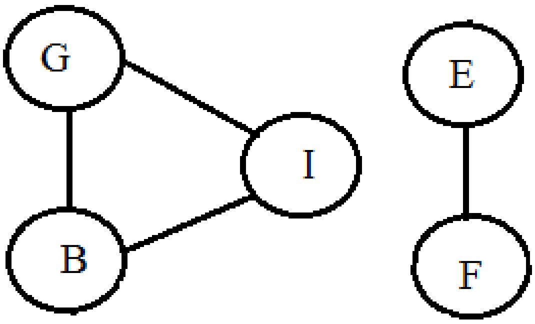
Spanning tree of a graph*:* Consider the set of all edges (u, w) where all vertices w are adjacent to u and are not visited. According to BFS algorithm it is established that this set of edges give the spanning tree of G, if G is connected. We obtain depth first search spanning tree similarly

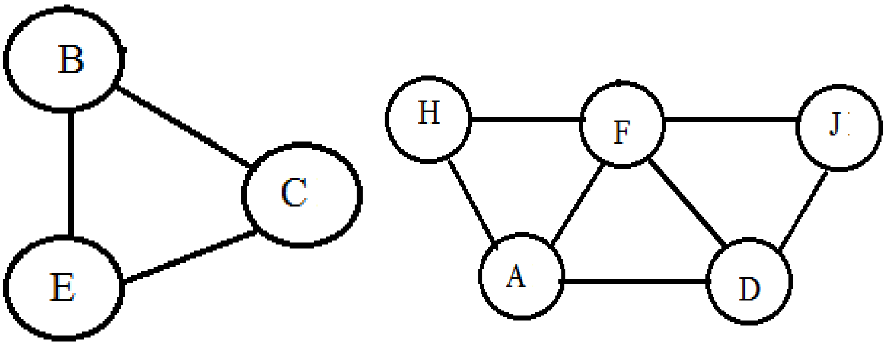
These are the BFS and DFS spanning trees of the graph G



Bi-connected Components

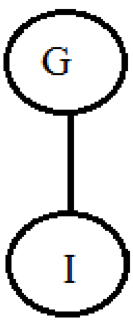
A connected undirected graph is said to be bi-connected if it remains connected after removal of any one vertex and the edges that are incident upon that vertex.
In this we have two components.
i. *Articulation point*: Let G= (V, E) be a connected undirected graph. Then an articulation point of graph ‘G’ is a vertex whose articulation point of graph is a vertex whose removal disconnects the graph ‘G’. It is also known as “cut point”.
ii. *Bi-connected graph:* A graph ‘G’ is said to be bi-connected if it contains no-articulation point.

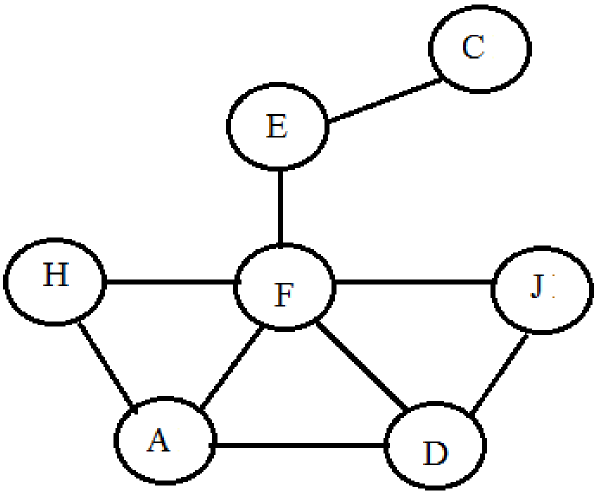




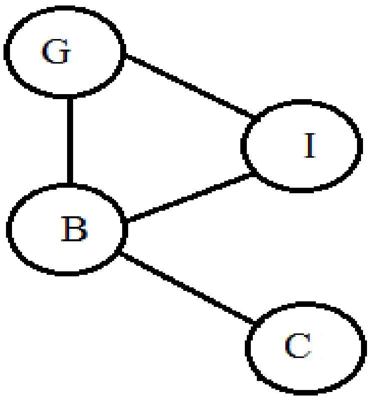
Articulation points for the above undirected graph are B, E, F

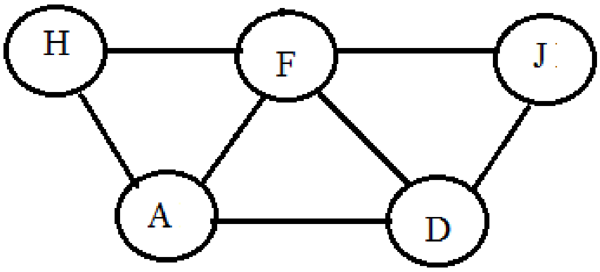
i) After deleting vertex B and incident edges of B, the given graph is divided into two components



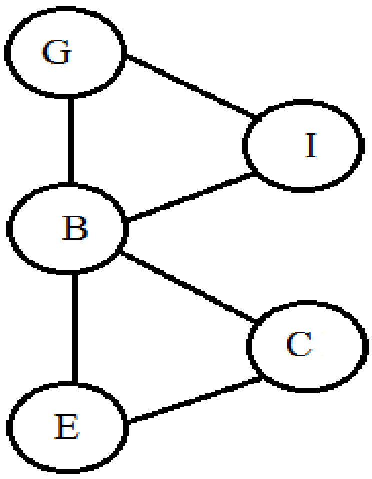


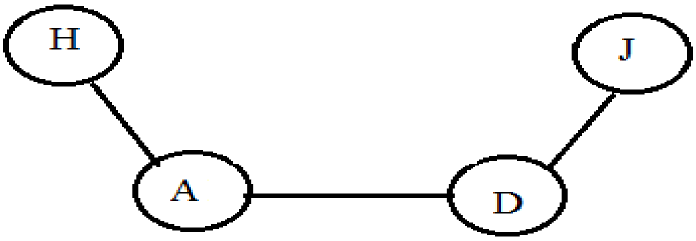
ii) After deleting the vertex E and incident edges of E, the resulting components are





iii) After deleting vertex F and incident edges of F, the given graph is divided into teo components.





Note: If there exists any articulation point, it is an undesirable feature in communication network where joint point between two networks failure in case of joint node fails.

Algorithm to construct the Bi- Connected graph

1. For each articulation point 'a' do

2. Let B1, B2, B3 ….Bk are the Bi-connected components

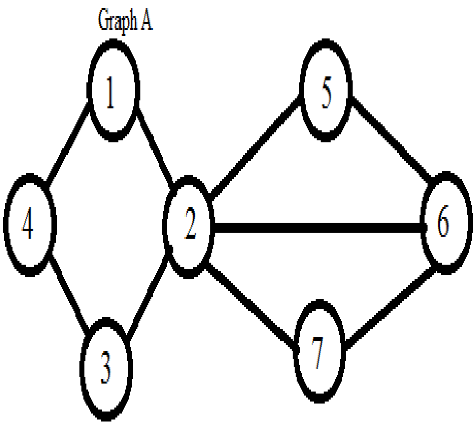
3. Containing the articulation point 'a'

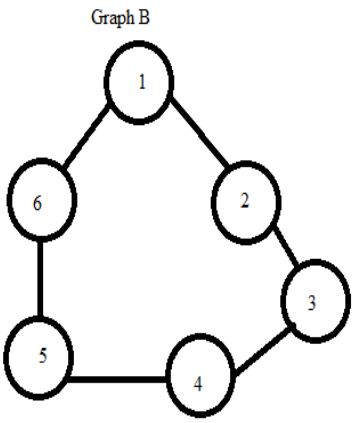
4. Let Vi E Bi, Vi # a i<=i<=k

5. Add (Vi,Vi+1) to Graph G.

Vi-vertex belong Bi
Bi-Bi-connected component
i- Vertex number 1 to k
a- articulation point

Exercise





****