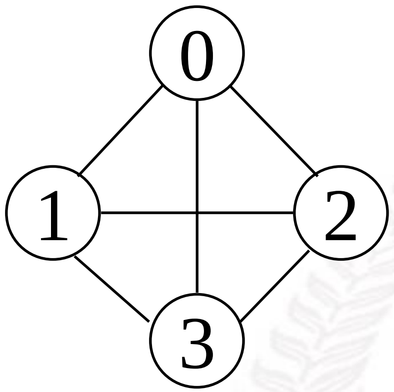
GRAPHS

Definitions

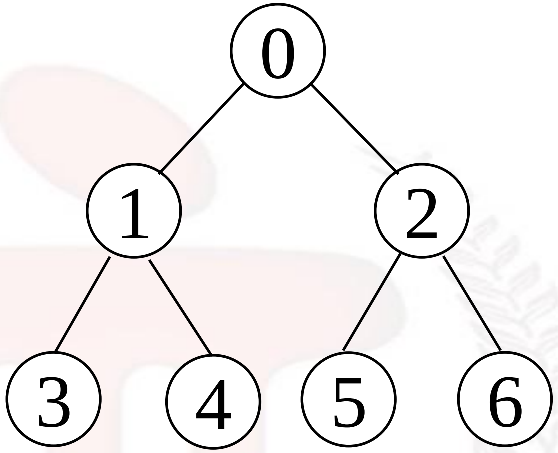
- A graph, $G=(V, E)$, consists of two sets:
  - a finite set of ***vertices(V)***, and
  - a finite, possibly empty set of edges***(E)***
  - $V(G)$ and $E(G)$ represent the sets of vertices and edges of G, respectively
- Undirected graph
  - The pairs of vertices representing any edge is ***unordered***
  - e.g., (v_0, v_1) and (v_1, v_0) represent the same edge   (v0, v1) = (v1,v0)
- Directed graph
  - Each edge as a directed pair of vertices   <v0, v1> != <v1,v0>
  - e.g. $<v_0, v_1>$ represents an edge, v_0 is the tail and v_1 is the head

Examples for Graph



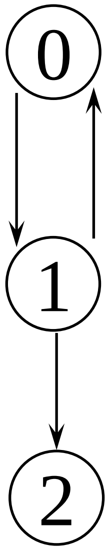
G_1

complete graph



G_2

incomplete graph



G_3

$V(G_1)=\{0,1,2,3\}$ $E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

$V(G_2)=\{0,1,2,3,4,5,6\}$ $E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$

$V(G_3)=\{0,1,2\}$ $E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

complete undirected graph: $n(n-1)/2$ edges

complete directed graph: $n(n-1)$ edges

Complete Graph

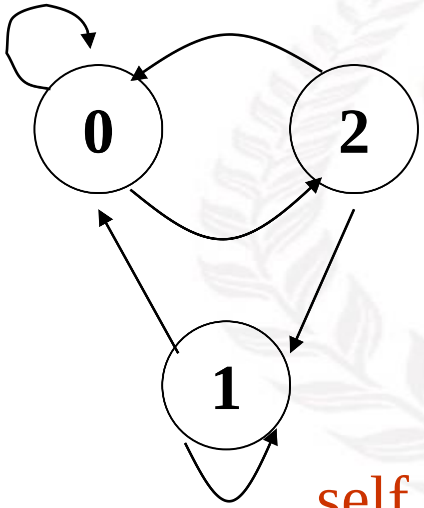
A complete graph is a graph that has the maximum number of edges

- for undirected graph with n vertices, the maximum number of edges is n(n-1)/2
- for directed graph with n vertices, the maximum number of edges is n(n-1)
- example: G1 (previous slide) is a complete graph

Adjacent and Incident

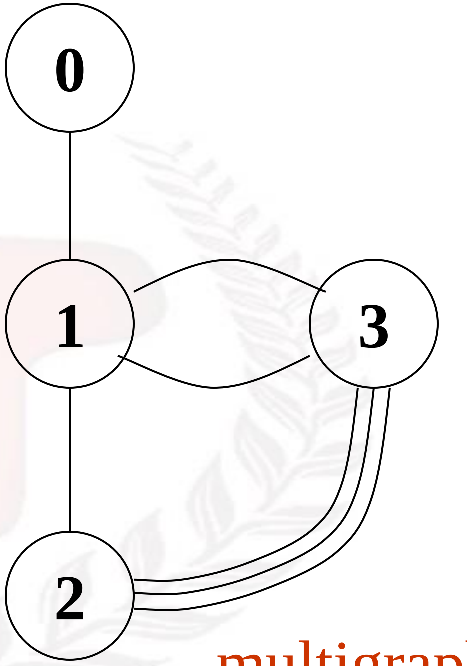
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

***Figure 6.3:** Example of a graph with feedback loops and a multigraph (p.260)*



self edge

(a)



(b)

multigraph:
multiple occurrences of the same edge

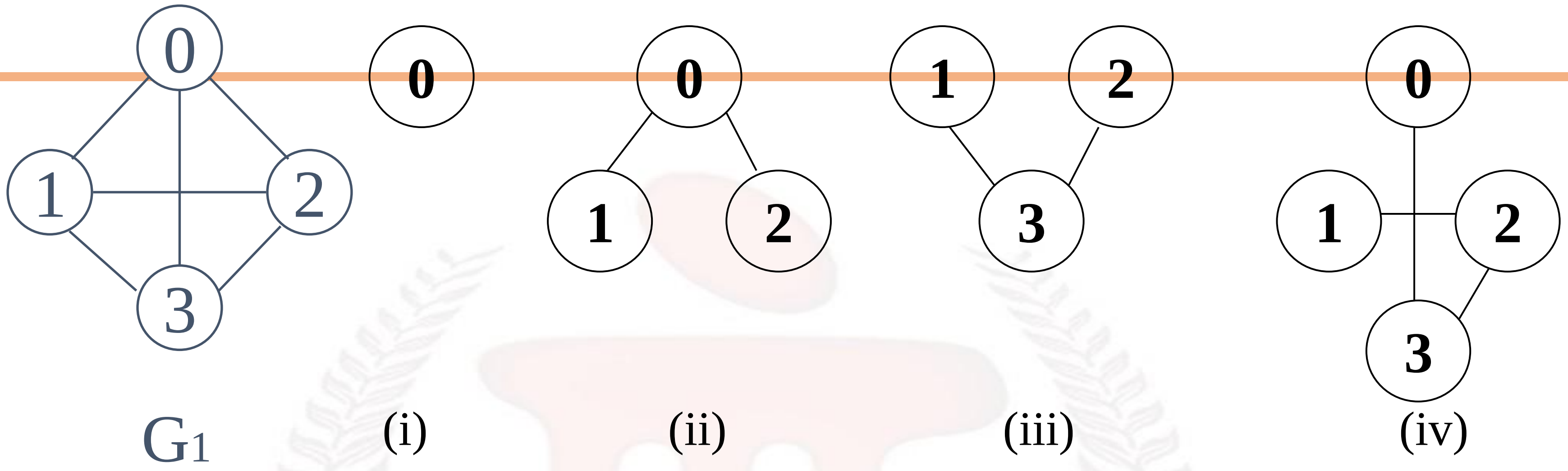
Subgraph and Path

- A subgraph of G is a graph G’ such that V(G’) is a subset of V(G) and E(G’) is a subset of E(G)

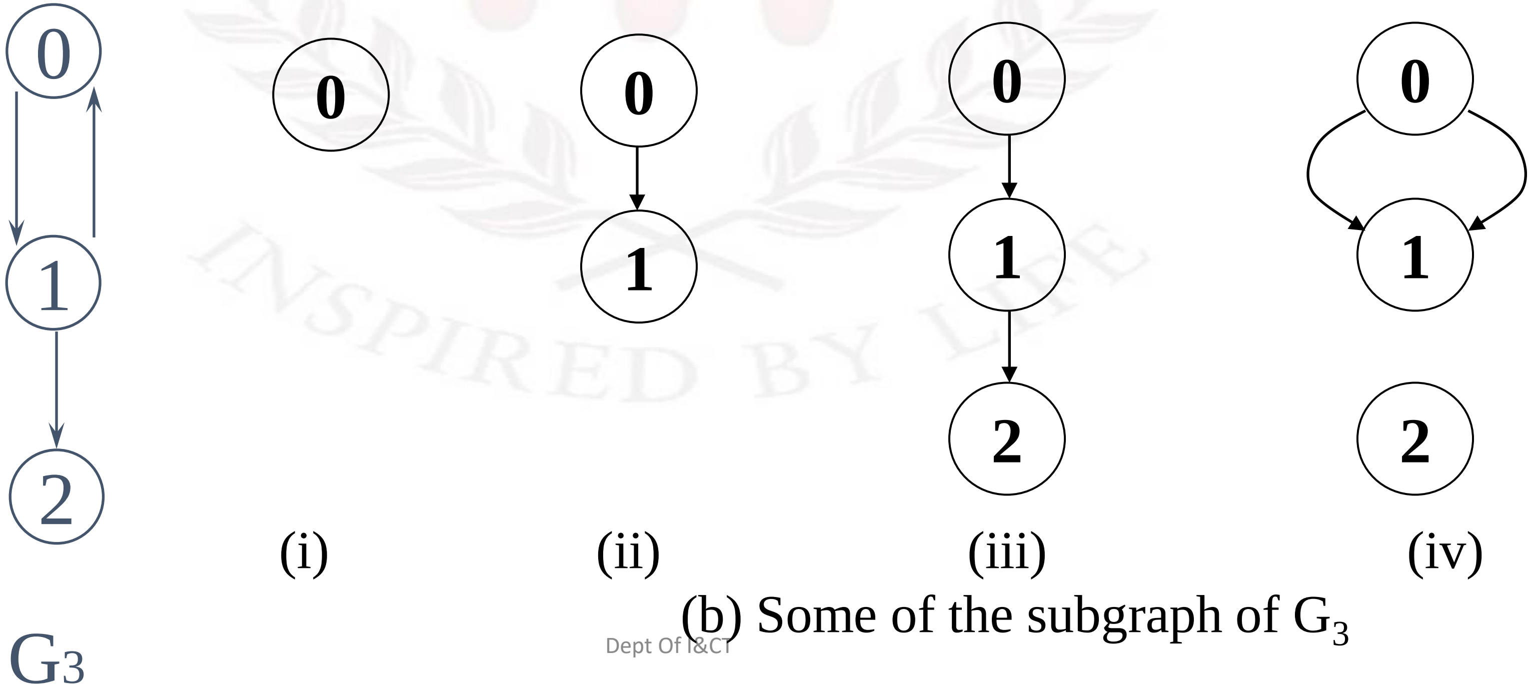
- A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) are edges in an undirected graph

- The length of a path is the number of edges on it

Figure 6.4: subgraphs of G_1 and G_3 (p.261)



(a) Some of the subgraph of G_1



(b) Some of the subgraph of G_3

Simple Path and Cycle

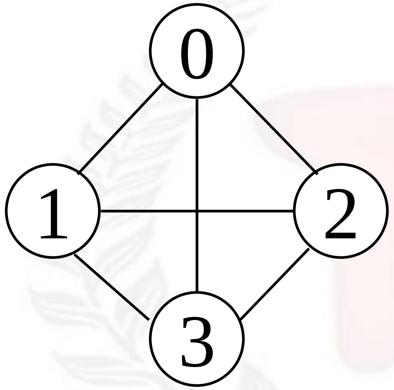
- A simple path is a path in which all vertices, except possibly the first and the last, are distinct

- A cycle is a simple path in which the first and the last vertices are the same

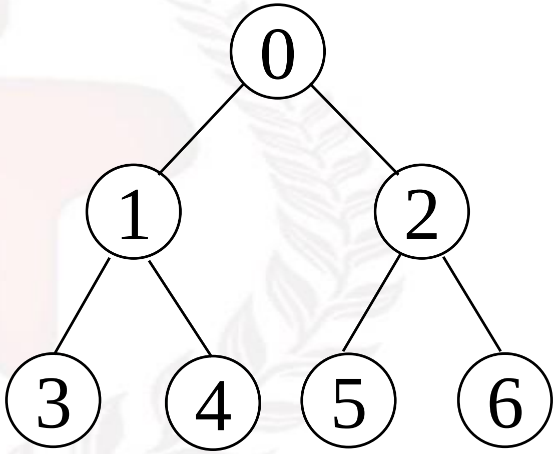
- In an undirected graph G, ***two vertices***, v_0 and v_1, are connected if there is a path in G from v_0 to v_1

- An undirected ***graph*** is connected if, for every pair of distinct vertices v_i, vj, there is a path from vi to v_j

connected



G_1



G_2

tree (acyclic graph)

Degree of an undirected graph

- The degree d_i of vertex i is the number of edges incident on vertex i.

- In an undirected graph, if d_i is the degree of a vertex i, n is the number of vertices and e is the number of edges, then number of edges e is

$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

Degree of a directed graph

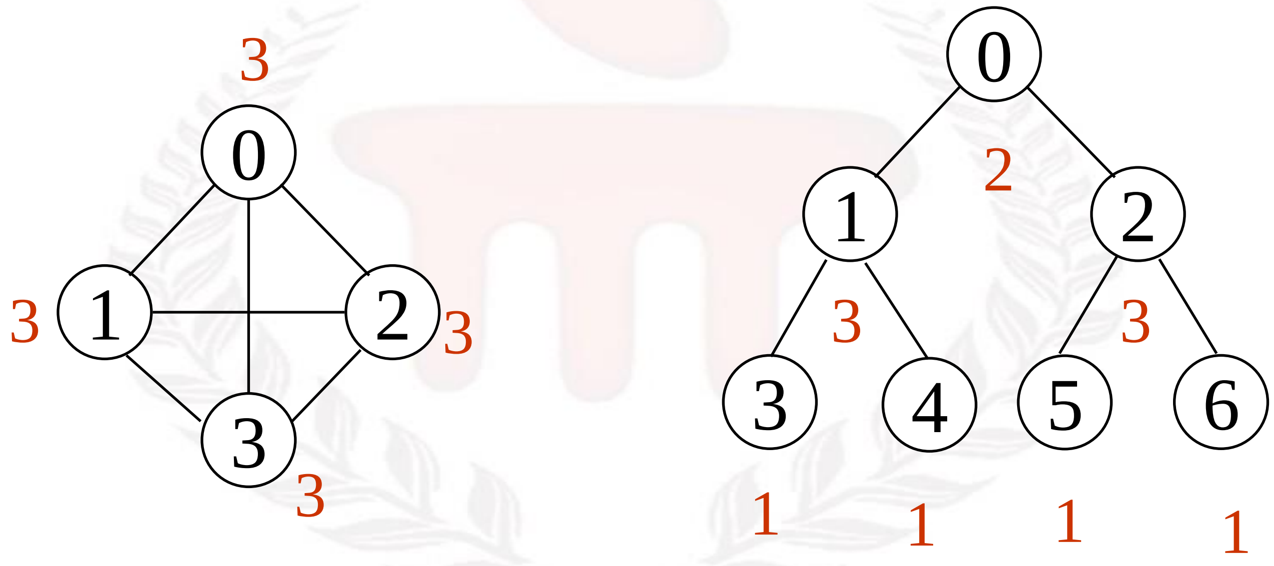
in – degree of vertex i: Let G be a digraph .The in-degree d_i^{in} of vertex i is the number of edges incident to i.

Out – degree of vertex i: The out- degree d_i^{out} of vertex i is the number of edges incident from this vertex.

$$e=\sum_{i=1}^{n} d_i^{in} = \sum_{i=1}^{n} d_i^{out}$$

Undirected graph

degree



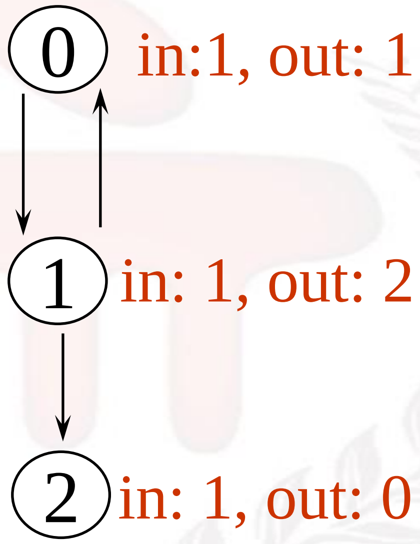
G1

G2

directed graph

in-degree

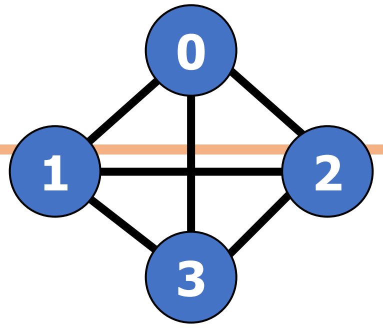
out-degree



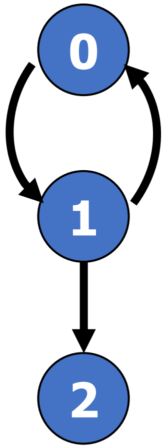
G_3

Graph representations

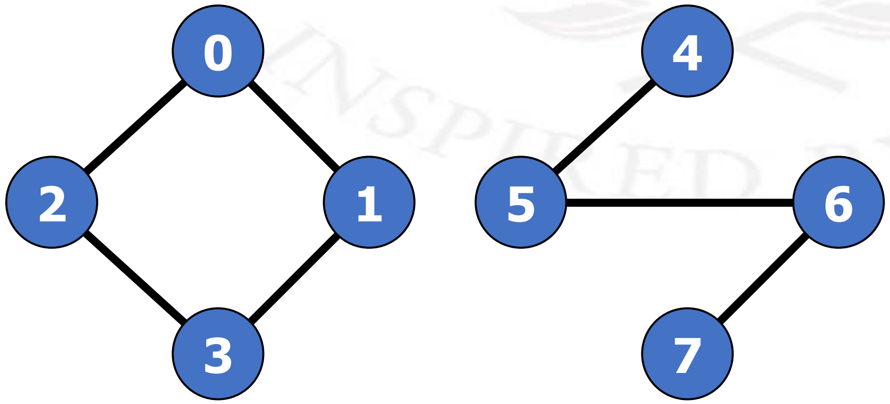
- Adjacency matrices
- Adjacency lists



G1

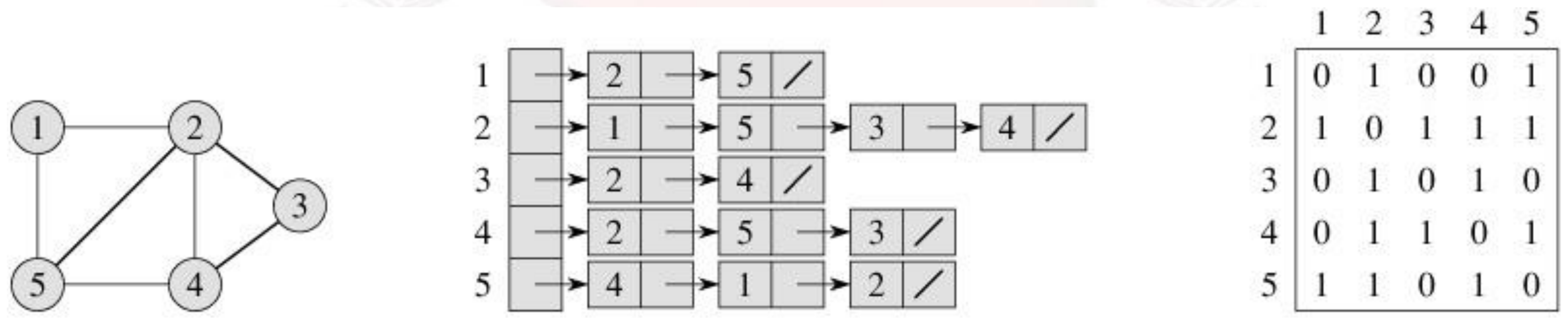


G2



G3

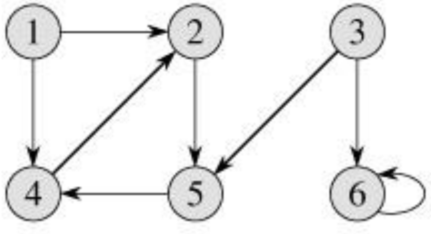
Graph representation – undirected



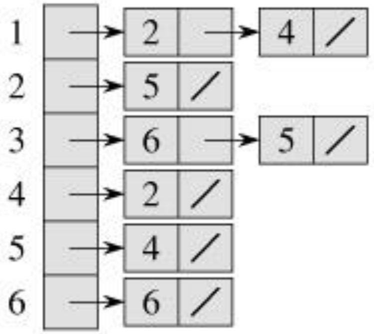
(a) (b) (c)

graph Adjacency list Adjacency matrix

Graph representation – directed



(a)



(b)

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

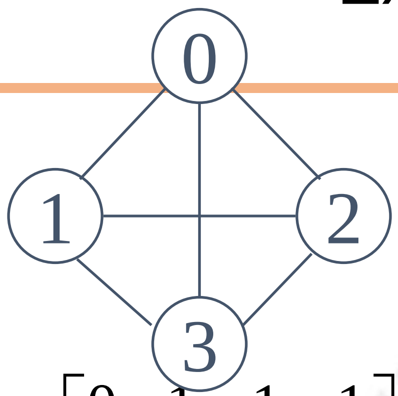
(c)

graph   Adjacency list   Adjacency matrix

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (vi, vj) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix

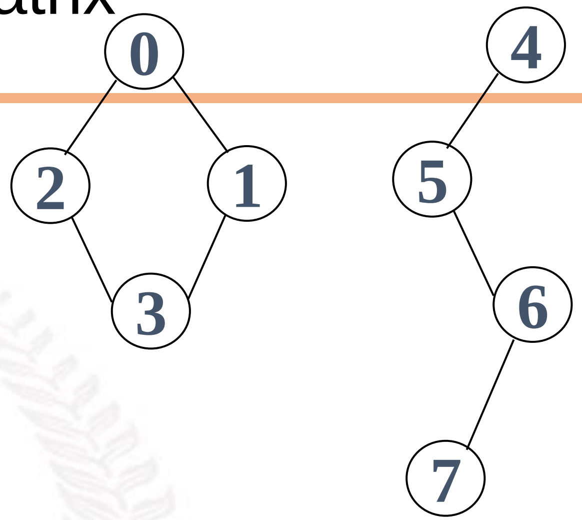


$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

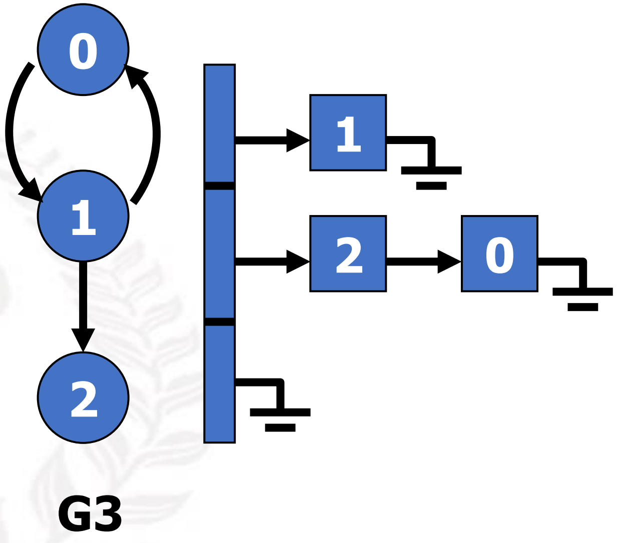
G_4

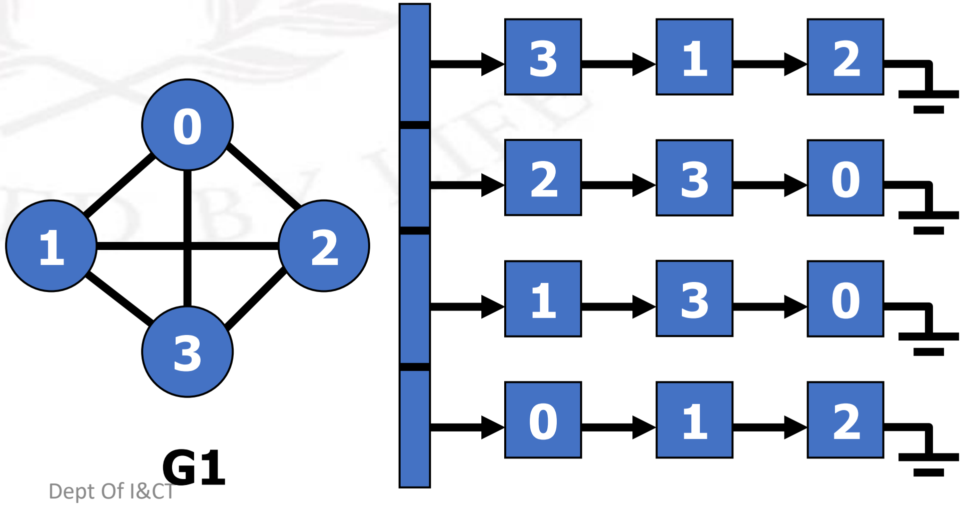
symmetric

undirected: $n^2/2$

directed: n^2

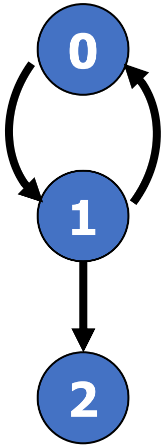
Adjacency lists





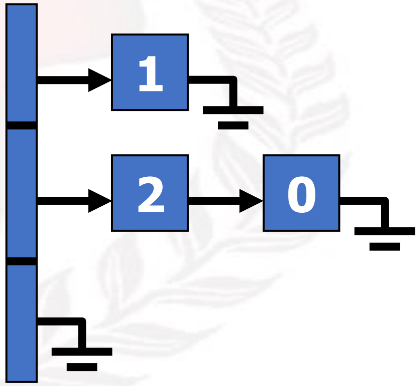
G1

Adjacency lists



G3

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$



Graph Operations

- Traversal
  Given G=(V,E) and vertex v, find all $w \in V$, such that w connects v.
  - Depth First Search (DFS)
    preorder tree traversal
  - Breadth First Search (BFS)
    level order tree traversal

Graph Traversals: DFS BFS

Some Graph Operations

- Traversal

  Given $G=(V,E)$ and vertex v, find all $w \in V$, such that w connects v.

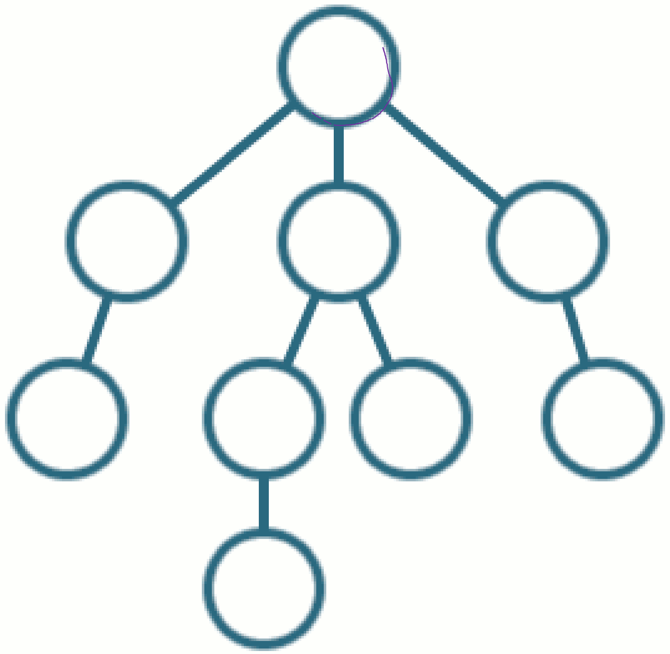
  - Depth First Search (DFS)

    preorder tree traversal

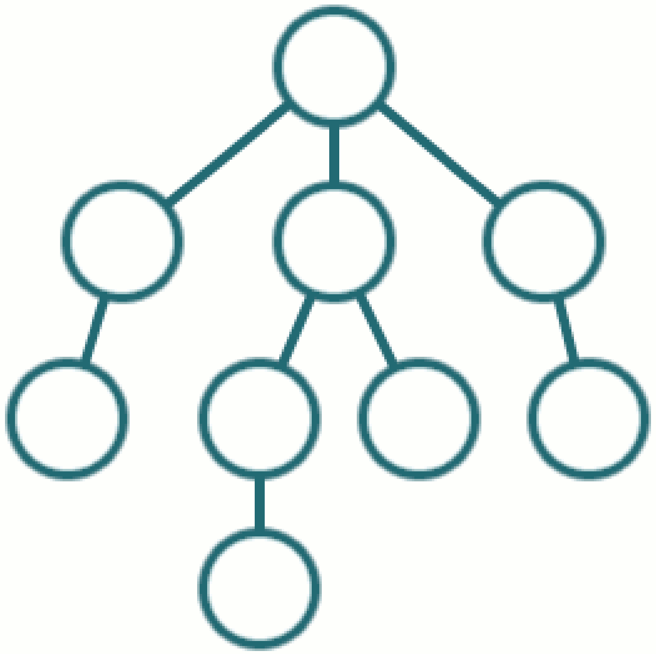
  - Breadth First Search (BFS)

    level order tree traversal

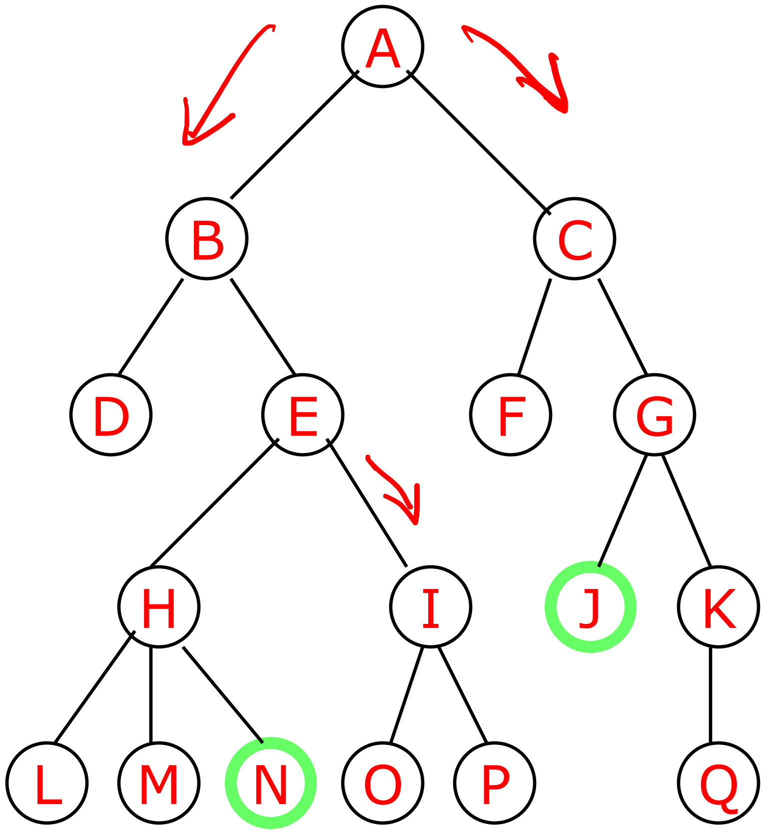
DFS



BFS



Depth-first search



- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q

DFS Algorithm

Mark all the n vertices as not visited.

insert source into stack and mark it visited

while(Stack is not empty)

{

delete Stack element into variable u

place all the adjacent (not visited) vertices of u into Stack and also mark them visited

print u

}

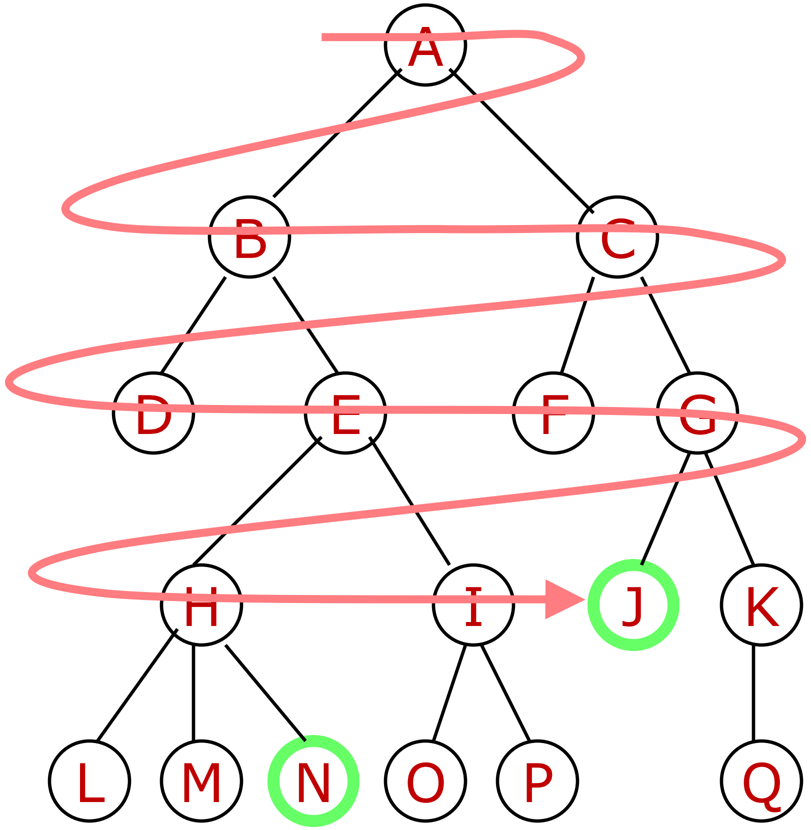
```
void dfs(int a[20][20],int n, int source)
{  int visited[10],u,v,i;
   for(i=1;i<=n;i++)        visited[i]=0;
   int S[20],top=-1;
   S[++top]=source;
   visited[source]=1;
   while(top>=0)
   {  u=S[top--];
      for(v=1;v<=n;v++)
      {   if(a[u][v]==1 && visited[v]==0)
          {
             visited[v]=1;          S[++top]=v;
          }
      }
      printf(" %d ",u);
   }
}
```

Breadth first search

It is so named because

It discovers all vertices at distance k from s before discovering vertices at distance k+1.

Breadth-first search



- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q

Algorithm BFS

Mark all the n vertices as not visited.

insert source into Q and mark it visited

while(Q is not empty)

{

delete Q element into variable u

place all the adjacent (not visited) vertices of u into Q and also mark them visited

print u

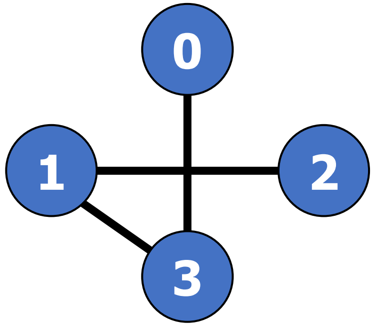
}

```
void bfs(int a[20][20],int n,int source)
{
int visited[10],u,v,i;

  for(i=1;i<=n;i++)       visited[i]=0;
     int Q[20],f=-1,r=-1;
    Q[++r]=source;    visited[source]=1;
    while(f<r)
    {
        u=Q[++f];
        for(v=1;v<=n;v++)
        {   if(a[u][v]==1 && visited[v]==0)
            {
                visited[v]=1;
                Q[++r]=v;
                  }
        }
        printf(" %d ",u);
    }
}
```

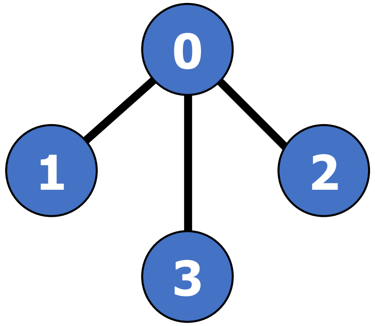
```
int main()
{
    int a[20][20],source, n,i,j;
    printf("Enter the no of vertices: ");
            scanf("%d",&n);

    printf("Enter the adjacency matrix: ");
    for(i=1;i<=n;i++)
            for(j=1;j<=n;j++)
                    scanf("%d",&a[i][j]);

      printf("Enter the source vertex: ");
            scanf("%d",&source);

    printf("\n BFS: ");    bfs(a,n,source);
    printf("\n DFS: ");    dfs(a,n,source);
    return 1;
}
```

Spanning Tree (ST)

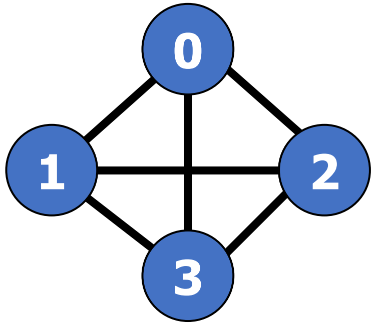
- A spanning tree is a minimal subgraph G', such that V(G')=V(G) and G' is connected. Spanning Tree is always acyclic.



ST1(G)



ST2(G)



G1