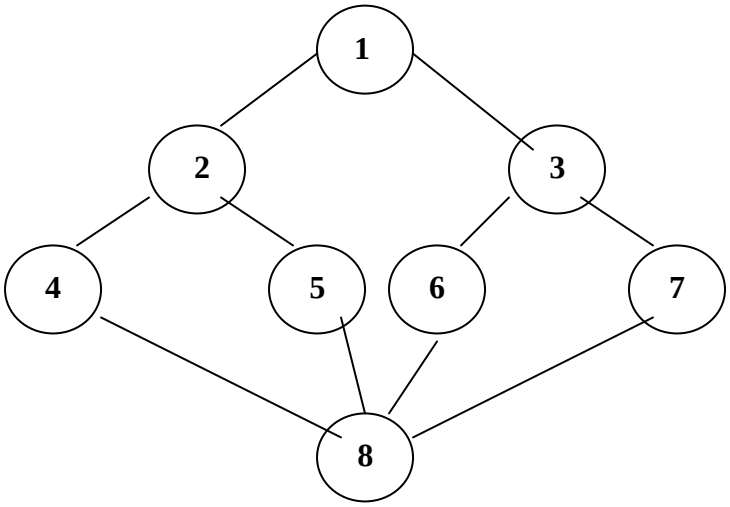
Assignment No : - B-10

Aim: To study graph representation in memory and traversal of graph.

Title: - Graph representation using adjacency matrix.

Problem Statement: There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight takes to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Justify the storage representation used.

Theory: GRAPH: A Graph 'G' consists of two sets V & E where, V is finite non-empty set of vertices &E is set of pairs of vertices called edges(G) represents set of vertices in graph 'G' & E(G) represents set of edges in graph 'G'. We can write G = (V, E) to represent a graph. For example, **Graph 'G1'**<1,2>, <2,1>



In above graph G = (V,E),V = {1, 2, 3, 4, 5, 6, 7, 8}

E = { (1,2), (1,3), (2,4), (2,5), (3,6), ((3,7), (4,8), (5,8), (6,8), (7,8) }

UNDIRECTED GRAPH: A graph in which pair of vertices (edge) is unordered, i.e. pairs (1,2) & (2,1) represent the same edge in above graph 'G1'

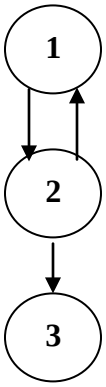
DIRECTED GRAPH : A graph in which each pair is represented by a directed pair. In Graph G2 given below pairs <1,2>, <2,1> & <2,3> are directed pairs. In pair <1,2>, vertex 1 is tail and vertex 2 is head of edge <1,2>. Thus the pairs <1,2>, <2,1> represent two different edges.

GRAPH REPRESENTATION: Commonly used representations to represent a graph in computer memory are,

1. Adjacency Matrix

2. Adjacency List

Graph G2



ADJACENCY MATRIX REPRESENTATION OF A GRAPH: Let G = (v,E) be a graph with 'v' vertices v >= 1. The adjacency matrix of graph G is a 2-dimensional n x n array say A[n:n]., with the property that,

A(i, j) = 1 iff the edge (Vi, Vj) is in E(G) for undirected graph or the edge <Vi, Vj> is in E(G) for directed graph and A(i, j) = 0 if there is no such edge in graph G from vertex Vi to Vj.

The adjacency matrix (symmetric matrix) for undirected graph G1 is,

Vertex			**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	0	1	1	0	0	0	0	0		
2	1	0	0	1	1	0	0	0		
3	1	0	0	0	0	1	1	0		
4	0	1	0	0	0	0	0	1		
5	0	1	0	0	0	0	0	1		
6	0	0	1	0	0	0	0	1		
7	0	0	1	0	0	0	0	1		
8	0	0	0	1	1	1	1	0		

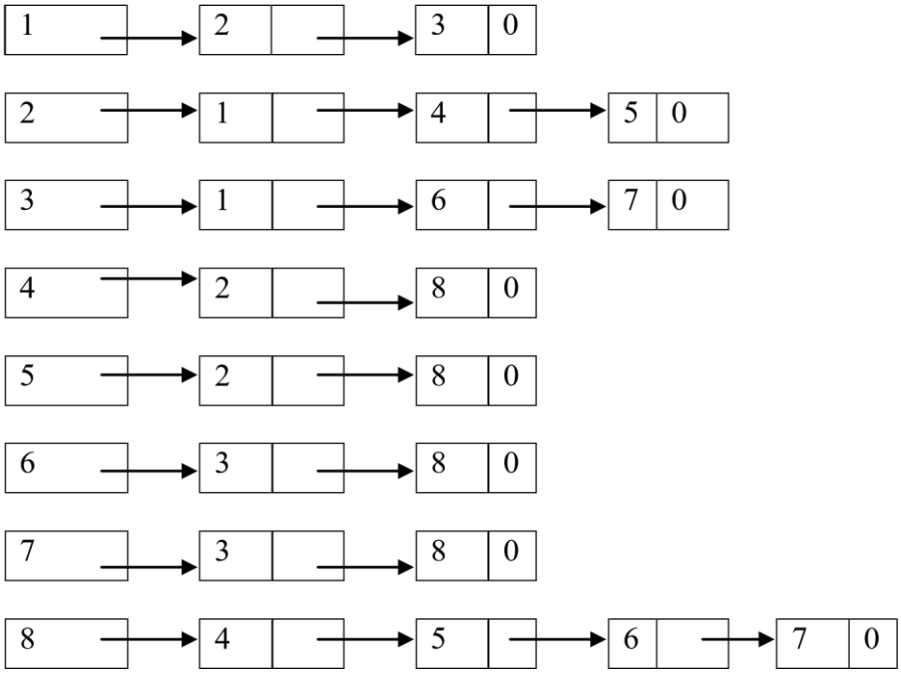
The adjacency matrix (non-symmetric matrix) for directed graph G2 is,

Vertex	**1**	**2**	**3**
1	0	1	0
2	1	0	1
3	0	0	0

ADJACENCY LIST REPRESENTATION OF A GRAPH: In this representation the 'n' rows of adjacency matrix are represented as 'n' linked lists. There is one list for each vertex in a graph. The nodes in list i represent the vertices that are adjacent to vertex i. The adjacency lists for undirected graph G1 and digraph G2 are given on next page.

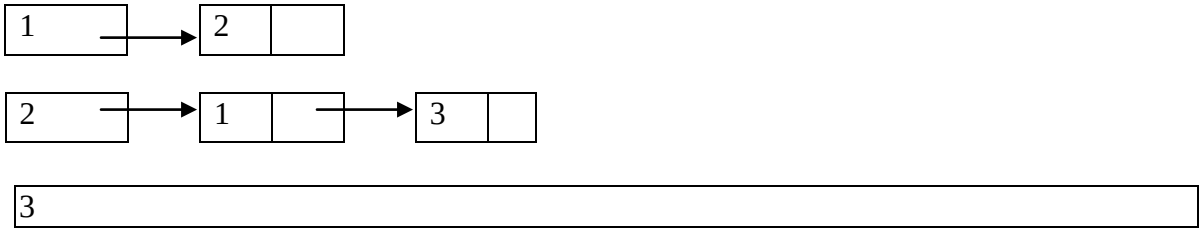
Adjacency list for undirected graph G1:

Head Nodes Adjacency List nodes



Adjacency list for directed graph G2:

Head Nodes Adjacency List nodes



TRAVERSAL OF A GRAPH: Let G = (V,E) be an undirected graph and a vertex v in V(G), we are interested in visiting all vertices in G that are reachable from vertex v i.e. all vertices connected to v. There are two ways to do this:

1) Depth First Search (DFS) and
2) Breadth First search (BFS)

DEPTH FIRST SEARCH: DFS is a recursive algorithm in which we start from a given vertex v, next an unvisited vertex w adjacent to v is selected and Depth First Search from w is initiated. When a vertex u is reached such that all of it's adjacent vertices have been visited, we back up to the

immediate previous vertex visited which has an unvisited vertex w adjacent to it and initiate a DFS from w and so on.
For example, if we start from vertex 1 in graph G1 and apply DFS procedure, the vertices will be visited in following sequence: **1 2 4 8 5 6 3 7**

BREADTH FIRST SEARCH*:* Starting at vertex v & making it as visited; Breadth First Search differs from Depth First Search in that all unvisited vertices adjacent to v are visited next. Then unvisited vertices adjacent to these vertices (visited in previous pass) are visited and so on. For example, if we start from vertex 1 in graph G1 and apply BFS procedure, the vertices will be visited in following sequence: 1 2 3 4 5 6 7 8

ANALYSIS OF ALGORITHMS:

1) Time Complexity:

For the construction of an undirected graph with 'n' vertices using adjacency matrix is **O (n^2),** since for every pair (i, j) we need to store either '0' or '1' in an array of size n x n.

For the construction of an undirected graph with 'v' vertices & 'e' edges using adjacency list is O **(v + e)**, since for every vertex v in G we need to store all adjacent edges to vertex v.

For DFS and BFS traversals of an undirected graph with 'n' vertices using adjacency matrix is **O (n^2),** since we visit to every value in an array of size n x n.

For DFS and BFS traversals of an undirected graph with 'v' vertices using adjacency list **O(e),** since we visit to each node in adjacency list exactly ones which is equal to number of edges in a graph.

2) Space Complexity :

Additional space required for DFS and BFS to store the intermediate elements in STACK and QUEUE respectively while traversing the graph .

ALGORITHMS:

Algorithm Create ()
// This algorithm is used to read input undirected graph from user.

```
1.  {
2.  Read( no of nodes as n);
3.  Repeat()
4.  {
5.  Write("Enter staring and ending vertices and its cost");
6.  Read(v1 , v2 and c);
7.  cost[v1][v2]=cost[v2][v1]=c;
8.  Write("Do You Want To Enter More Edges ");
9.  }until(false);
```

```
10.         }
```

Algorithm Display ()

// This algorithm is used to print undirected graph given by user.

```
1.      {
2.      for( i=1 to n) do
3.      for(j=1 to n ) do
4.      Write(cost[i][j]);
5.      }
```

Algorithm DFS ( ) // This algorithm is used to traverse graph in DFS order**.**

```
1.                      {
2.                      for(i=1 to n) do
3.                      visit[i]=0;
4.                      Read(Starting vertex in v1);
5.                      push(v1);
6.                      while(top!=-1)        // until stack  is not empty
7.                      {
8.                      v1=pop();
9.                      if(visit[v1]==0)     // v1 is unvisited
10.                     {
11.                     Write(v1);
12.                     visit[v1]=1;
13.                     for(v2=1 to n)                  // find neighbors
14.                     if(g[v1][v2]==1)      // neighbor exist
15.                     push(v2);
16.                     }
17.                     }
18.                     }
```

Algorithm BFS( ) // algorithm to traverse given graph in BFS order.

```
1.          {
2.          Read(Starting vertex in v1);
3.          for(i=1 to n) do
4.          visit[i]=0;      // All vertices are unvisited.
5.          add(v1);
6.          visit[v1]= 1;
7.          while(f!=r) // till queue is not empty
8.          {
9.          v1=remove(); //queued node
10.         Write(v1);
```

```
11.          for(v2=1 to n) do
12.          {  //each link of that node if not visited should be added into queue
14.           if(g[v1][v2]==1 && visit[v2]==0)
15.           {
16.           add(v2);
17            visit[v2]=1;
18.             }
19.             }
20.             }
21.             }
```

Flowchart:

Conclusion: