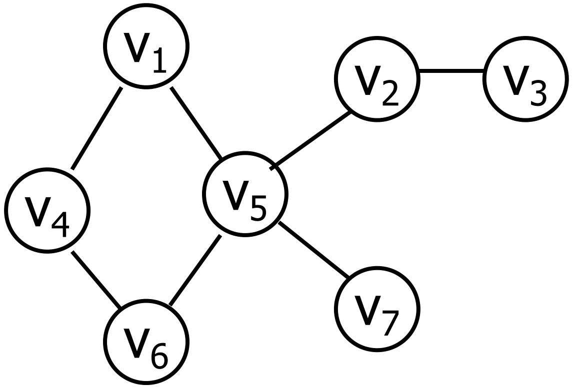
Graphs

Alok Kumar Jagadev

What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other.
- The set of edges describes relationships among the vertices.



Formal Definition of Graphs

- A graph *G* is defined as follows: G=(V, E)

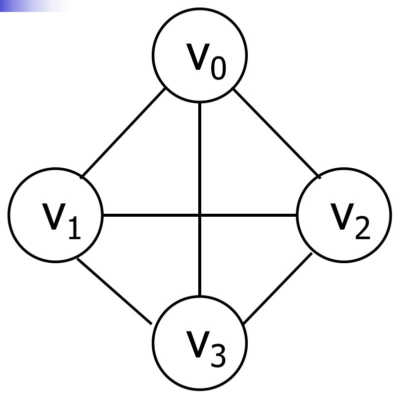
  V(G)*:* a finite, nonempty set of vertices

  E(G)*:* a set of edges (pairs of vertices)

  a set E that is a subset of V×V i.e. E is a set of pairs of the form (x, y) where x and y are nodes in V.

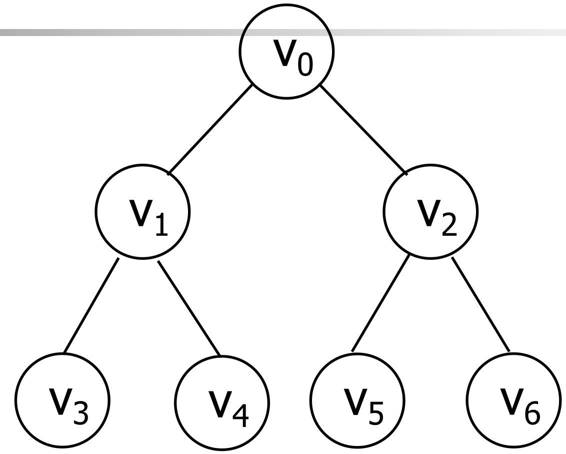
- An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A directed graph is one in which each edge is a directed pair of vertices, $(v_0, v_1) \neq (v_1, v_0)$

Examples for Graph



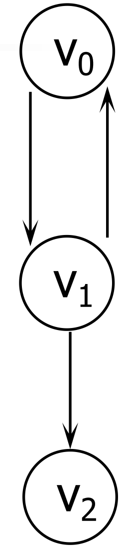
G_1

complete graph



G_2

incomplete graphs



G_3

$V(G_1)=\{v_0, v_1, v_2, v_3\}$

$V(G_2)=\{v_0, v_1, v2, v_3, v_4, v_5, v_6\}$

$V(G_3)=\{v_0, v_1, v_2\}$

$E(G_1)=\{(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_2), (v_1, v_3), (v_2, v_3)\}$

$E(G_2)=\{(v_0, v_1), (v_0, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_5), (v_2, v_6)\}$

$E(G_3)=\{(v_0, v_1), (v_1, v_0), (v_1, v_2)\}$

<u>complete undirected graph</u>: n(n-1)/2 edges

<u>complete directed graph</u>: n(n-1) edges

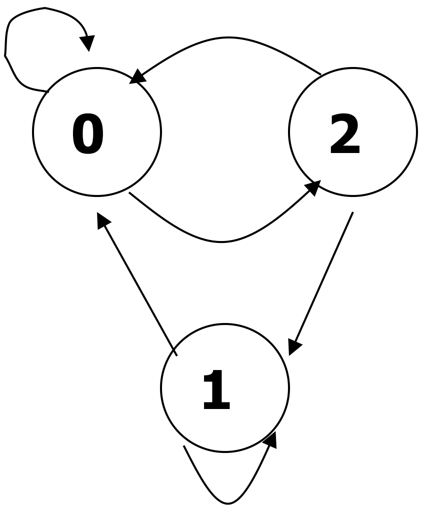
Complete Graph

- A complete graph is a graph that has the maximum number of edges
  - For undirected graph with n vertices: maximum number of edges: n(n-1)/2
  - For directed graph with n vertices, the maximum number of edges: n(n-1)
  - Example: G1 is a complete graph.

Adjacent and Incident

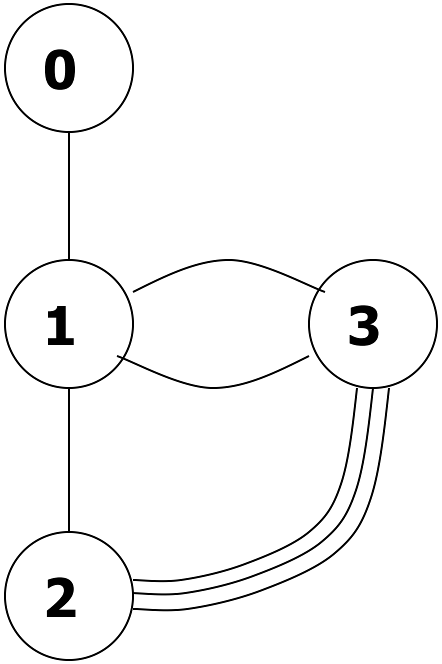
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If (v_0, v_1) is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0.
  - The edge (v_0, v_1) is incident on v_0 and v_1.

Example: Graph with feedback loops and a multigraph



self edge

(a)



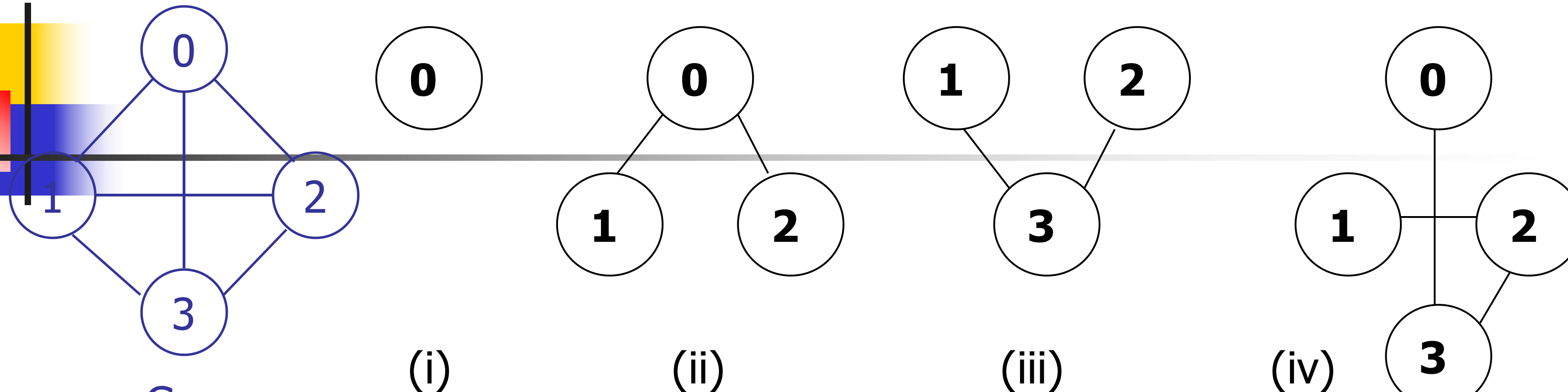
(b)

multigraph:
multiple occurrences
of the same edge

Subgraph and Path

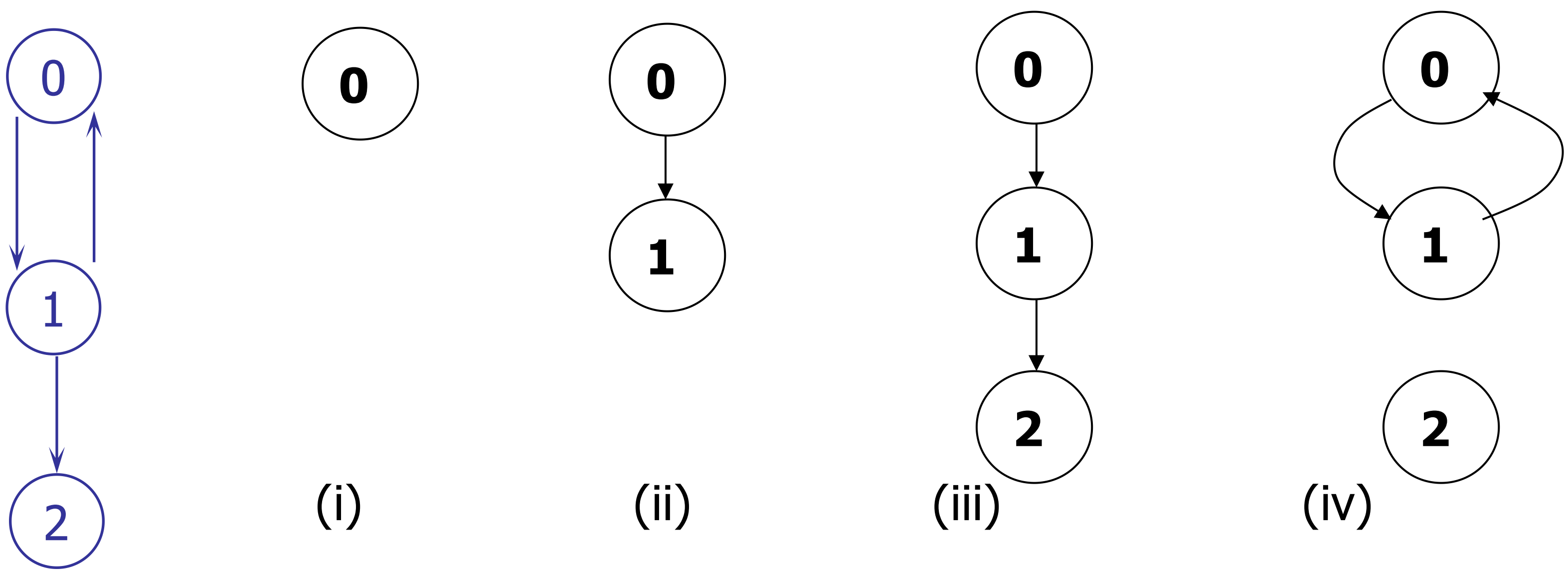
- A subgraph of G is a graph, S, such that V(S) is a subset of V(G) and E(S) is a subset of E(G).
- A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p , v_{i1} , v_{i2} , ... , v_{in} , v_q such that (v_p , v_{i1}), (v_{i1} , v_{i2}), ... , (v_{in} , v_q) are edges in an undirected graph.
- The length of a path is the number of edges on the path.

Subgraphs of G_1 and G_3



G_1

(a) Some of the subgraph of G_1



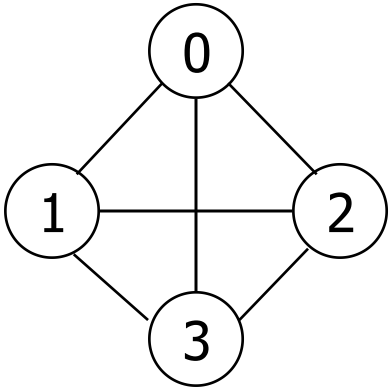
(b) Some of the subgraph of G_2

G_2

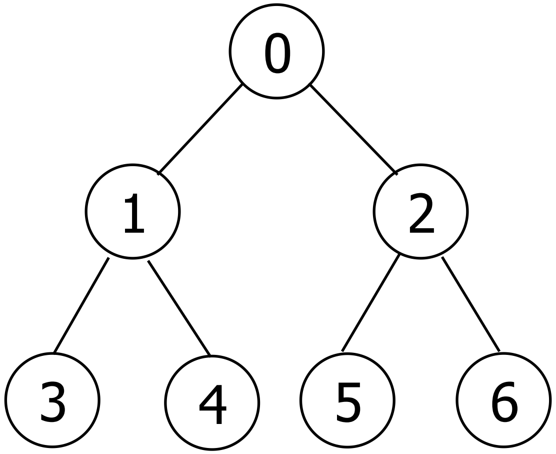
Some Terminology

- 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 from v_0 to v_1 in G.
- An undirected graph is connected if, for every pair of distinct vertices v_i, v_j, there is a path from v_i to v_j .

Connected



G_1



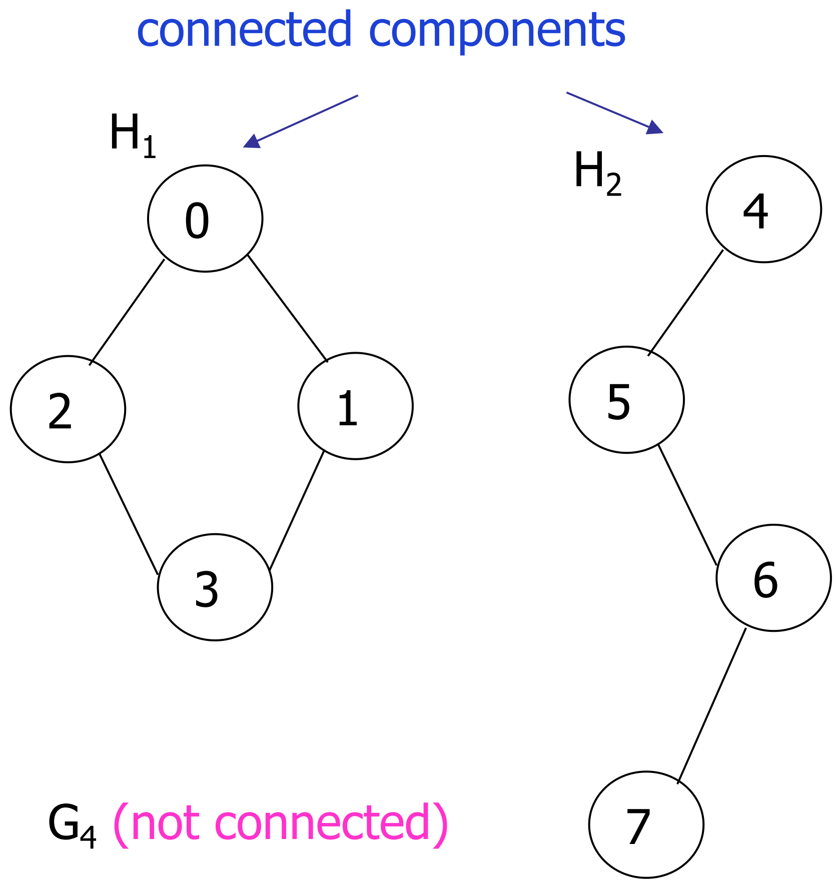
G_2

tree (acyclic graph)

Connected Component

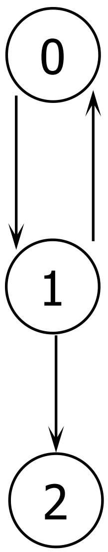
- A connected component or simply component of an undirected graph is a subgraph in which each pair of virtices is connected with each other via a path.
- A tree is a graph that is connected and acyclic.
- A directed graph is strongly connected if there is a directed path from v_i to v_j and also from v_j to v_i.

A graph with two connected components



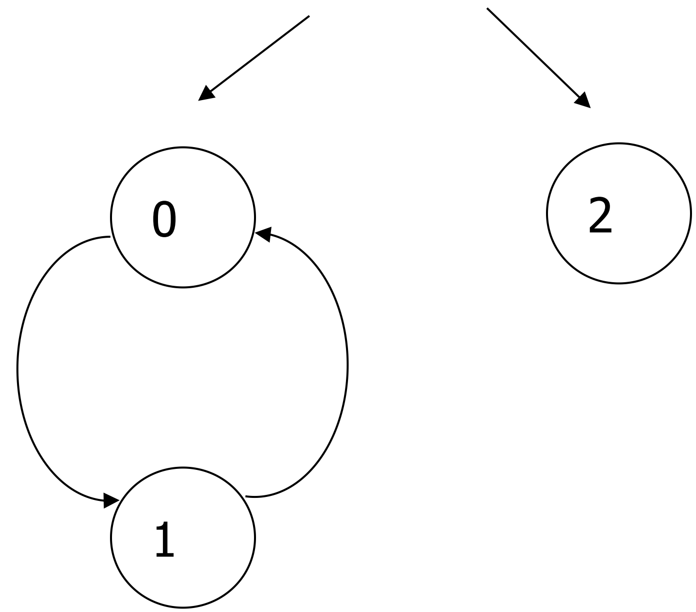
Strongly connected components

not strongly connected



G3

strongly connected components



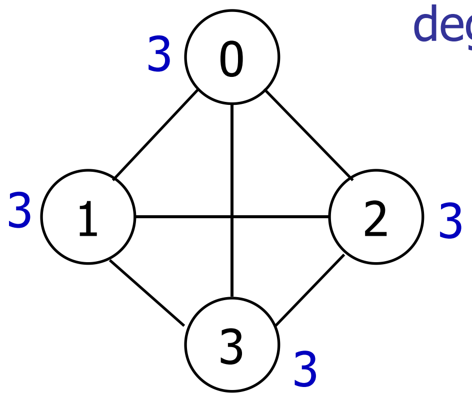
Degree

- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex v is the number of edges that have v as the head
  - the out-degree of a vertex v is the number of edges that have v as the tail
  - if d_i is the degree of a vertex *i* in a graph G with n vertices and e edges, the number of edges is

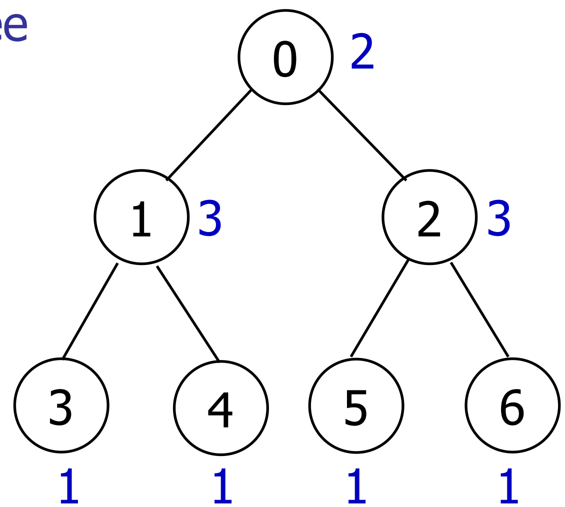
$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

Degrees in Graph

degree



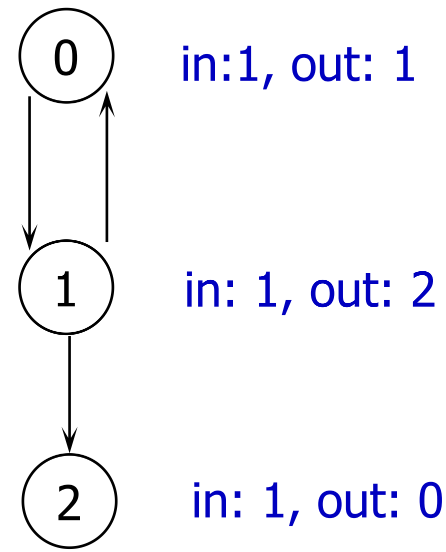
G1



G2

directed graph

in-degree
out-degree



G3

ADT for Graph

structure of Graph

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all graph $\in$ Graph, v, v_1 and v_2 $\in$ Vertices

Graph create(): return an empty graph

Graph insertVertex(graph, v): return a graph with v inserted. v has no incident edge.

Graph insertEdge(graph, v_1, v_2): return a graph with new edge between v_1 and v_2

Graph deleteVertex(graph, v): return a graph in which v and all edges incident to it are removed

Graph deleteEdge(graph, v_1, v_2): return a graph in which the edge (v_1, v_2) is removed

Boolean isEmpty(graph): if (graph==empty graph) return TRUE else return FALSE

List adjacent(graph, v): return a list of all vertices that are adjacent to v

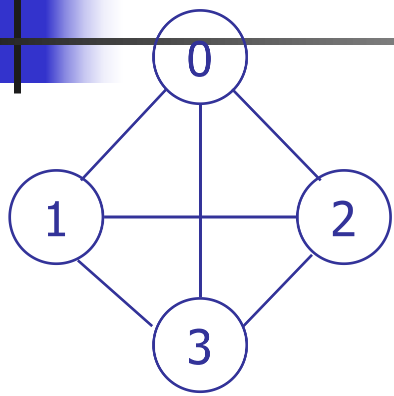
Graph Representations

- Adjacency Matrix
- Adjacency Lists
- Adjacency Multilists

Adjacency Matrix

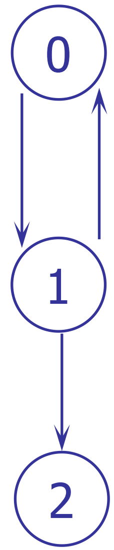
- Let G=(V, E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional $n \times n$ array, say adjMat.
- If the edge (v_i, v_j) is in E(G), adjMat[i][j]=1
- If there is no such edge in E(G), adjMat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency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

Examples for Adjacency Matrix



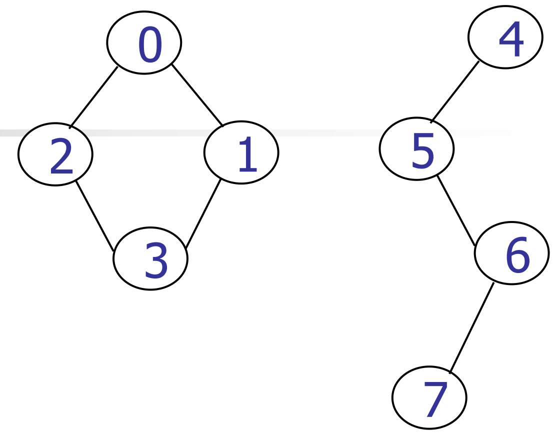
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G1



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

G2



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

G4

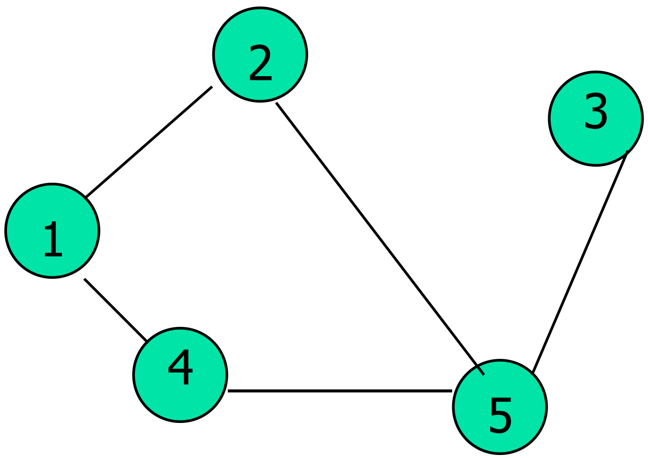
symmetric

Merits of Adjacency Matrix

- From the adjacency matrix, to determine the connection of vertices is easy.
- The degree of a vertex is $\sum_{j=0}^{n-1} adjMat[i][j]$
- For a digraph, the row sum is the out_degree, while the column sum is the in_degree

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

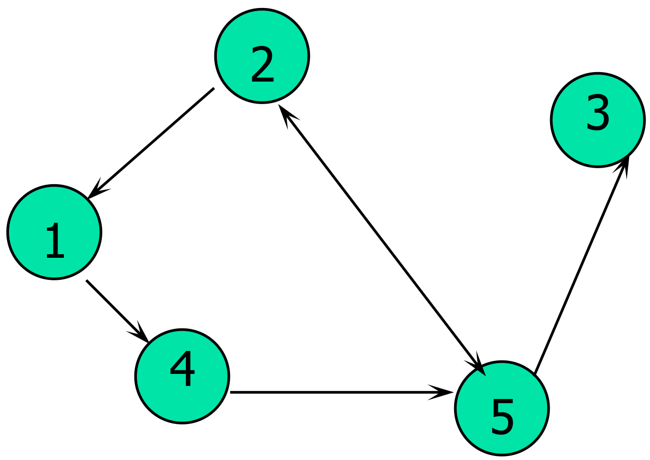
Adjacency Matrix Properties



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

- Diagonal entries are zero.
- Adjacency matrix of an undirected graph is symmetric.
  - A(i, j) = A(j, i) for all i and j.

Adjacency Matrix (Digraph)



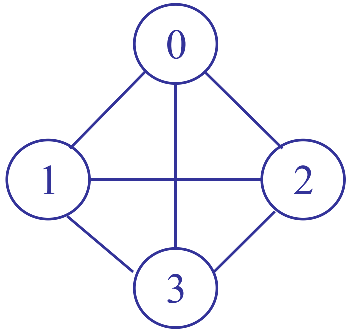
	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	1
3	0	0	0	0	0
4	0	0	0	0	1
5	0	1	1	0	0

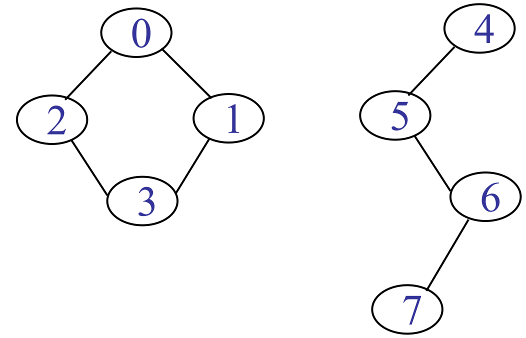
- Diagonal entries are zero.
- Adjacency matrix of a digraph need not be symmetric.

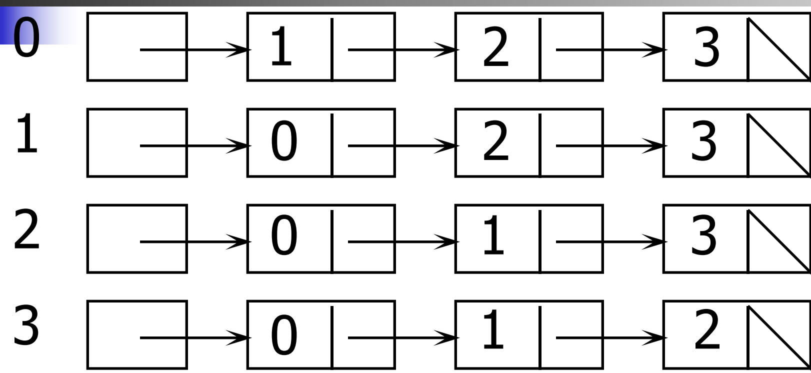
Data Structures for Adjacency Lists

Each row in adjacency matrix is represented as an adjacency list.

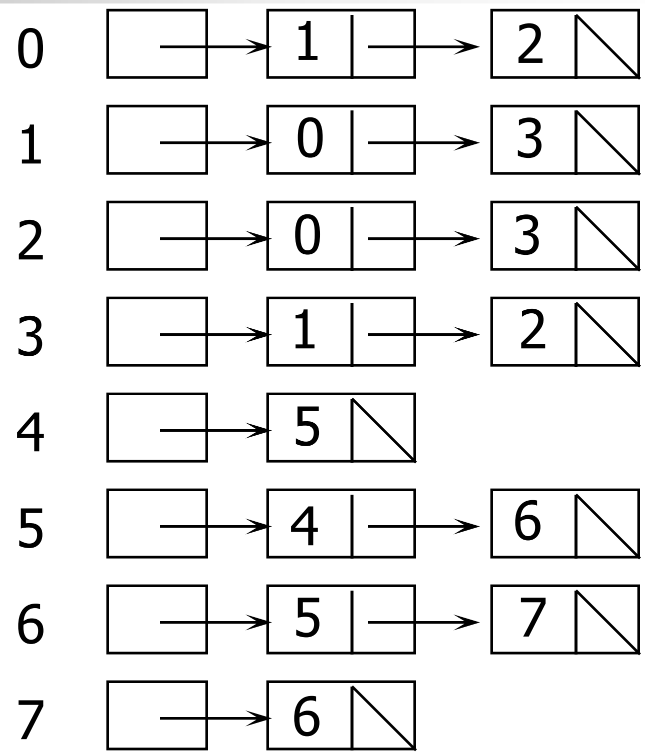
```
#define MAX_VERTICES 50

struct node {
    int vertex;
    struct node *link;
};

struct node * graph[MAX_VERTICES];
int n=0;                    // vertices currently in use
```



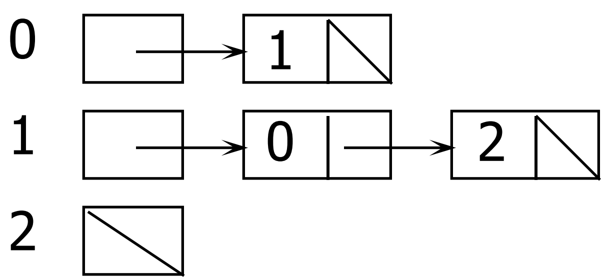




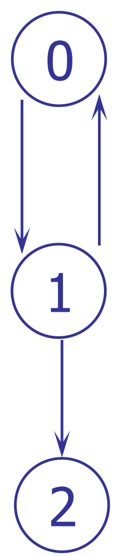
G1



G2



G3



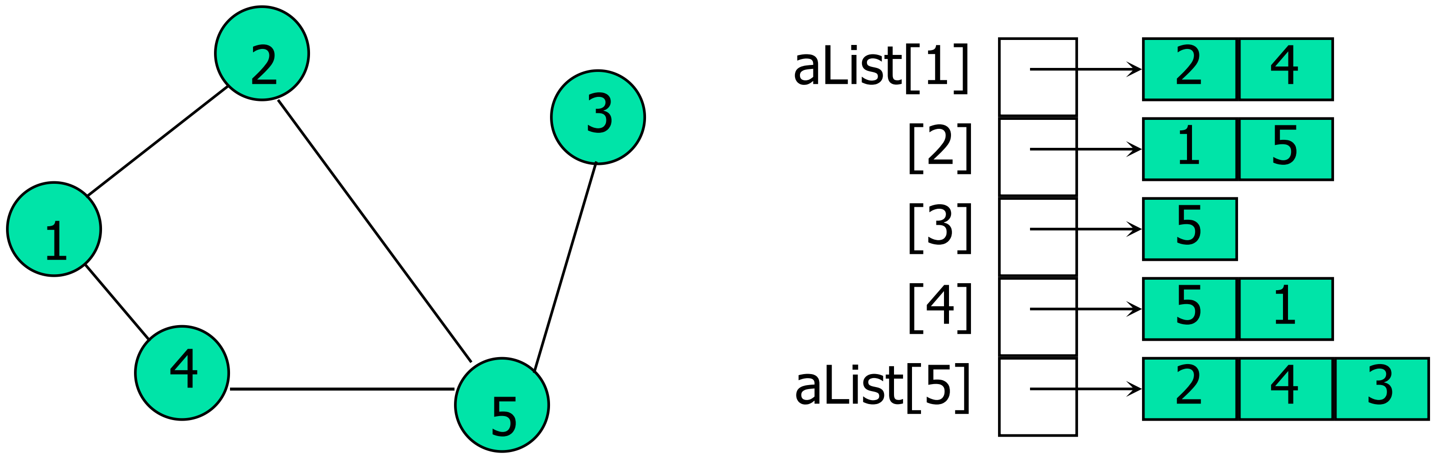
Array Length = n

# of chain nodes = 2e (undirected graph)

# of chain nodes = e (digraph)

Array Adjacency Lists

- Each adjacency list is an array list.



Array Length = n

# of list elements = 2e (undirected graph)

# of list elements = e (digraph)

Weighted Graphs

- Cost adjacency matrix.
  - C(i, j) = cost of edge (i, j)
- Adjacency lists => each list element is a pair (adjacent vertex, edge weight)

Graph Traversal Techniques

- The connectivity problem, as well as many other graph problems, can be solved using graph traversal techniques
- There are two standard graph traversal techniques:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)

Graph Traversal (Contd.)

- In both DFS and BFS, the nodes of the undirected graph are visited in a systematic manner so that every node is visited exactly once.
- Both BFS and DFS give rise to a tree:
  - When a node x is visited, it is labeled as visited, and it is added to the tree
  - If the traversal got to node x from node y, y is viewed as the parent of x, and x a child of y.

Depth-First Search

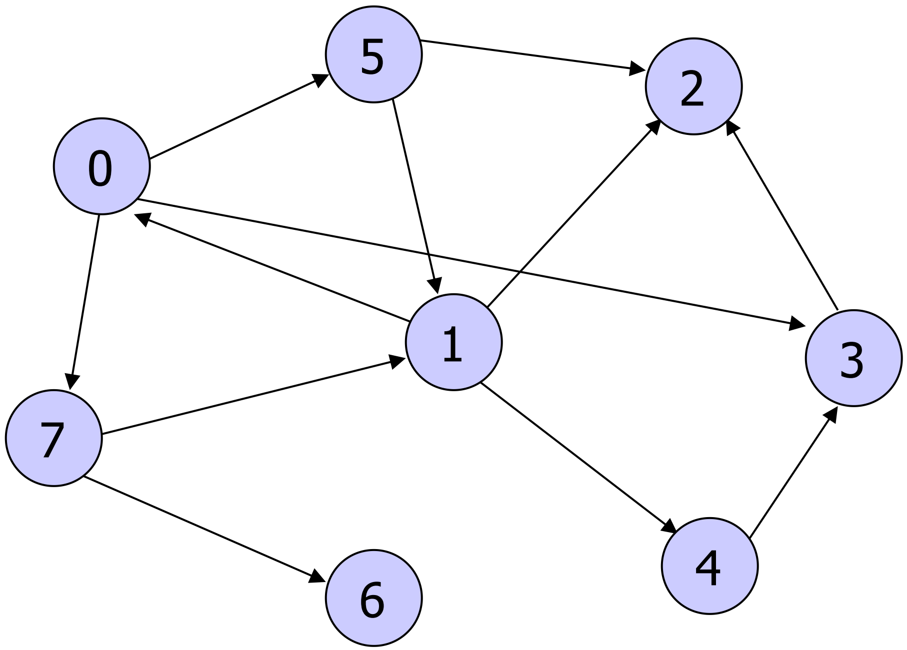
DFS follows the following rules:

1. Select an unvisited node x, visit it, and treat as the current node
2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
4. Repeat steps 2 and 3 until no more nodes can be visited.
5. If there are still unvisited nodes, repeat from step 1.

DFS (Pseudo Code)

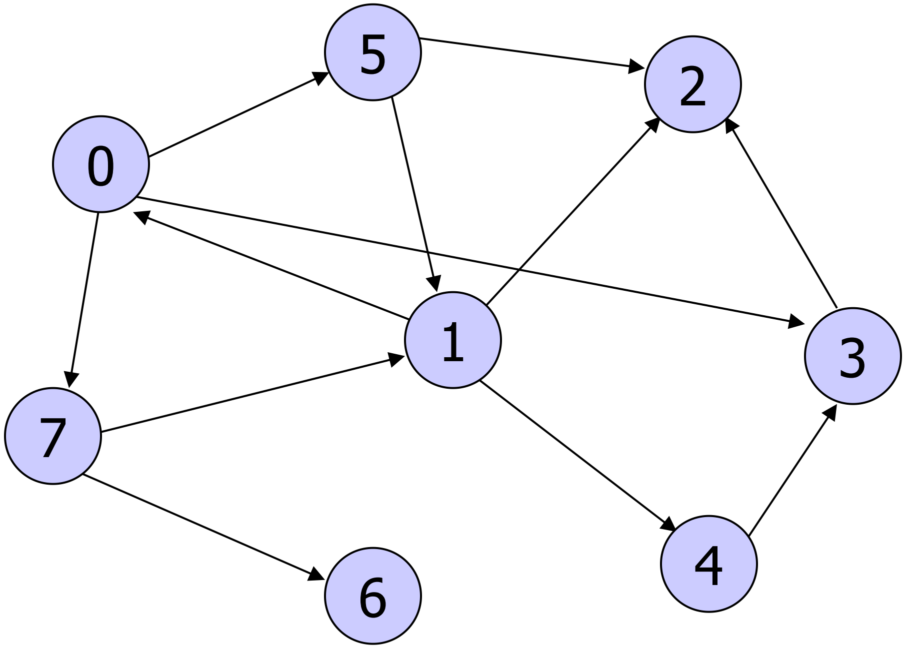
```
DFS(input: Graph G, Node v) {
    if (v = = NULL)
        return;
    push(v);
    while (stack is not empty) {
        pop(v);
        if (v has not yet been visited)
            mark_and_visit(v);
        for (each w adjacent to v)
            if (w has not yet been visited)
                push(w);
    }
}
```

Example



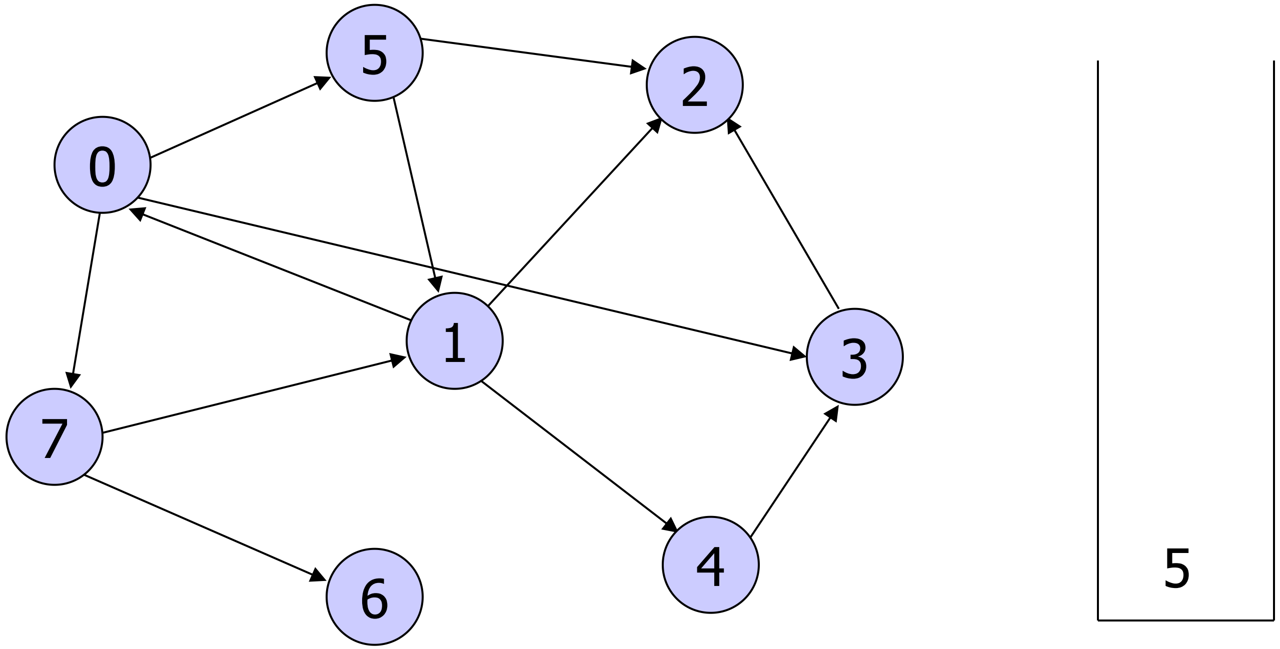
Policy: Visit adjacent nodes in <u>increasing index order</u>

DFS: Start with Node 5



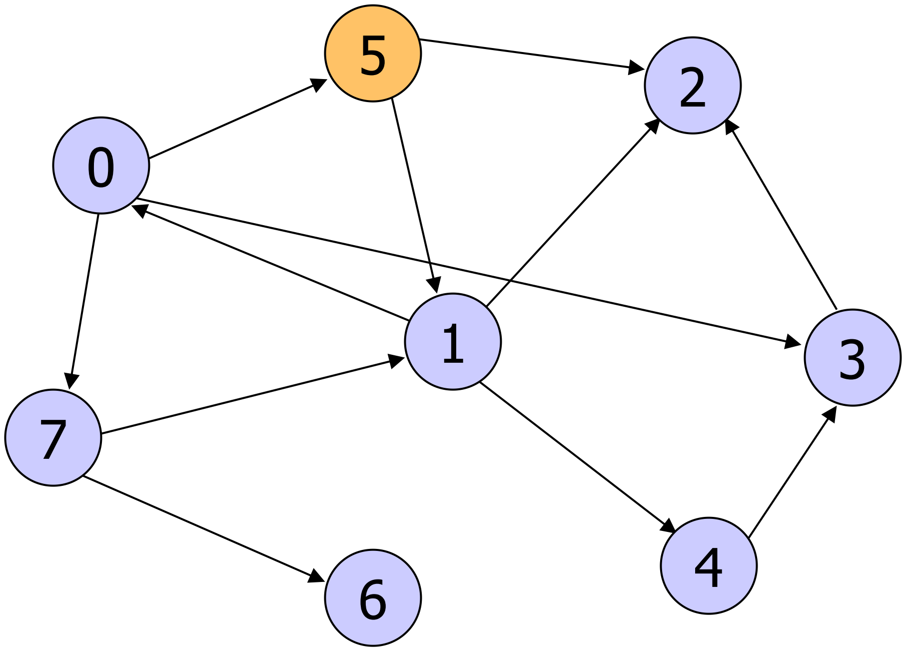
5 1 0 3 2 7 6 4

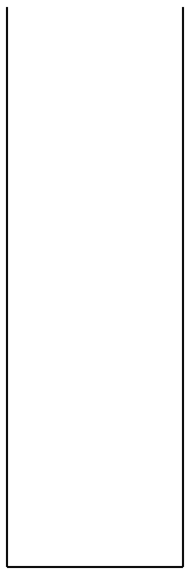
DFS: Start with Node 5



push (5)

DFS: Start with Node 5

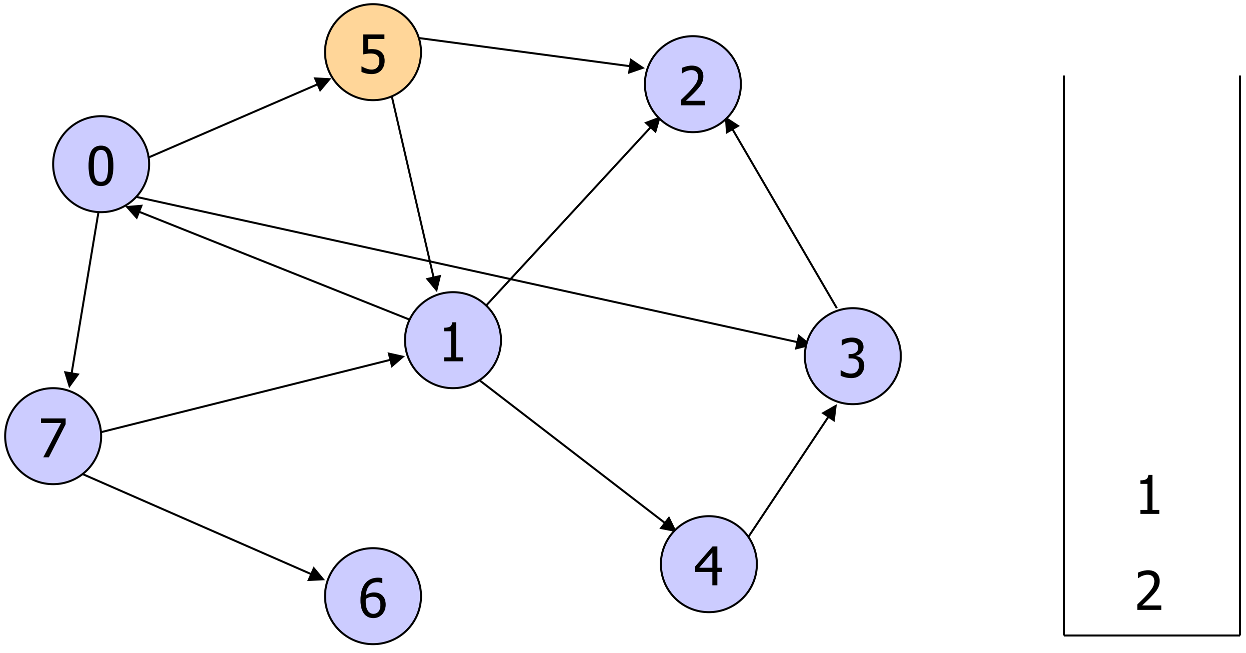




pop/visit/mark (5)

5

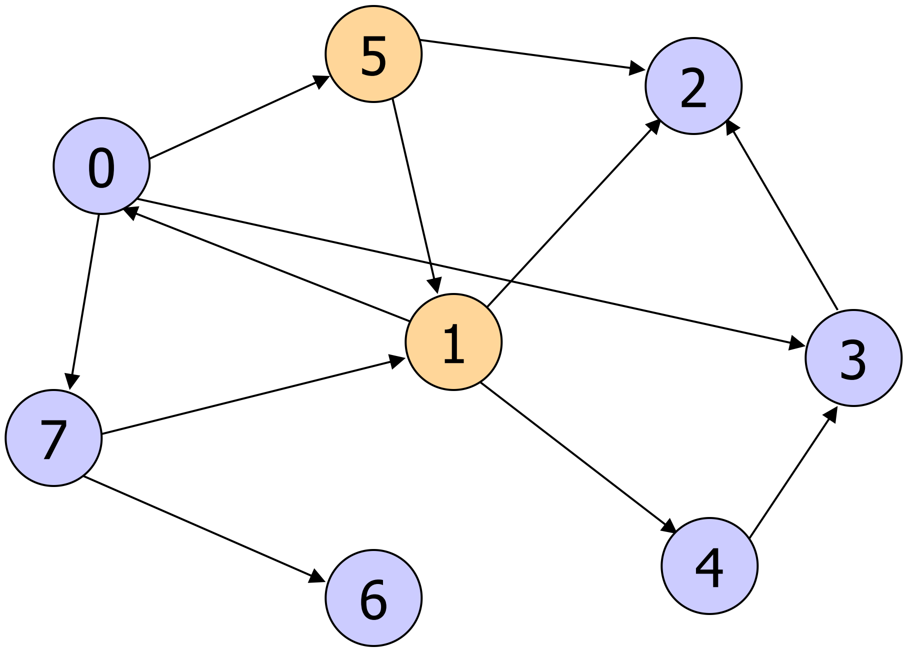
DFS: Start with Node 5

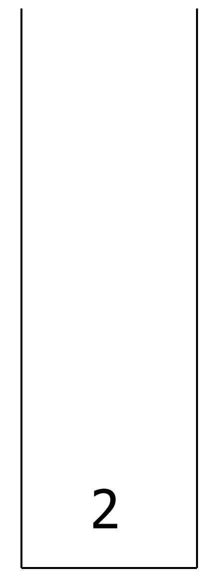


push (2), push (1)

5

DFS: Start with Node 5

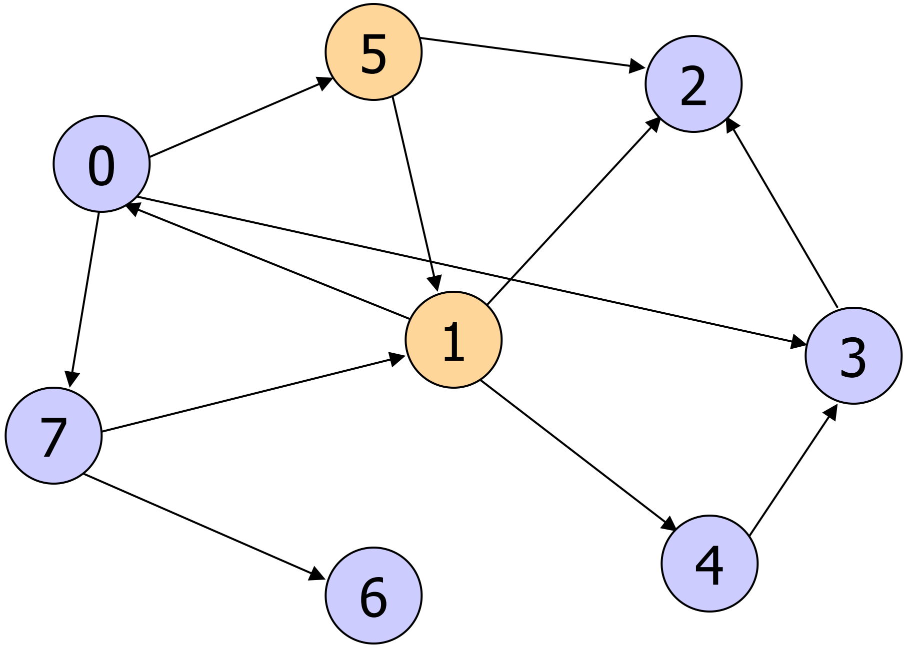


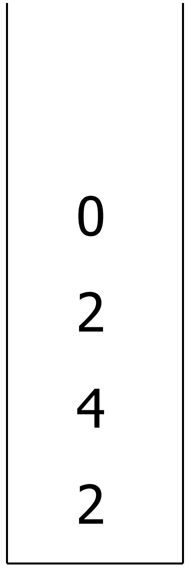


pop/visit/mark (1)

5 1

DFS: Start with Node 5

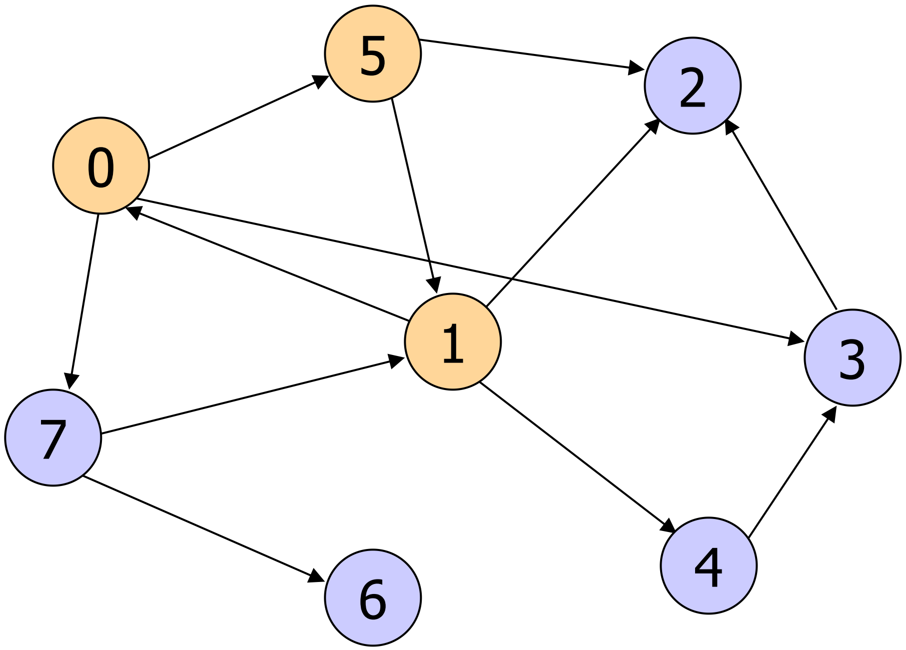


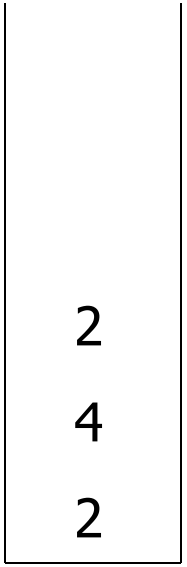


pop/visit/mark (1)

5 1

DFS: Start with Node 5

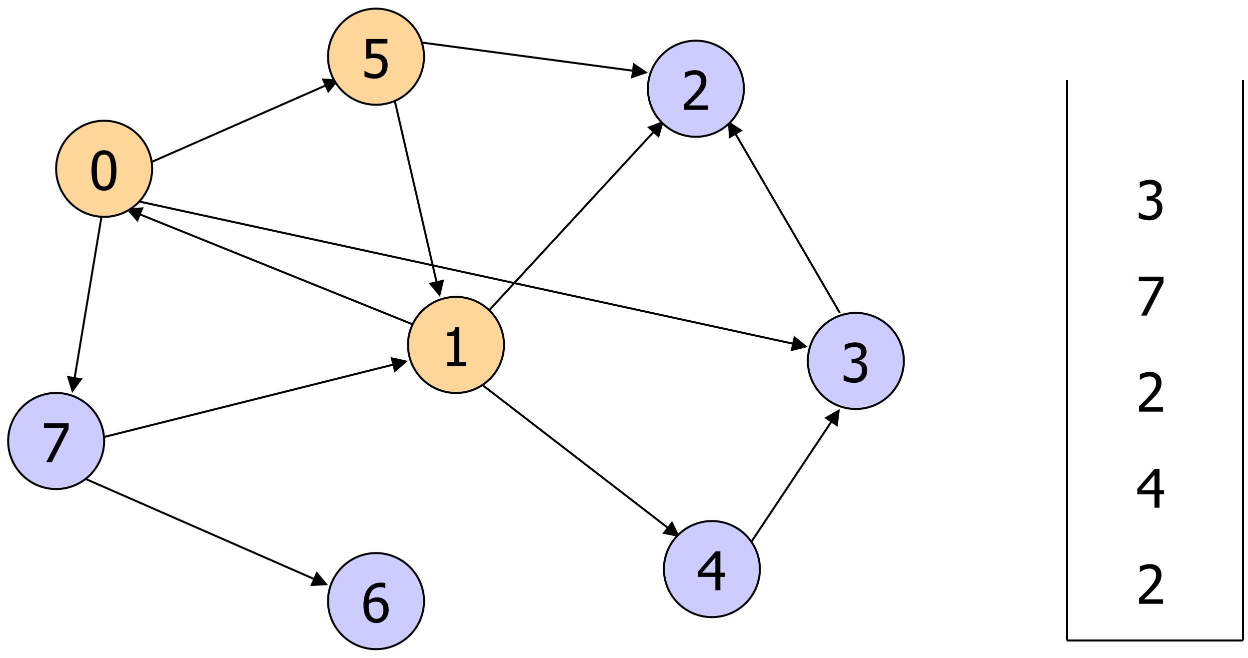




pop/visit/mark(0)

5 1 0

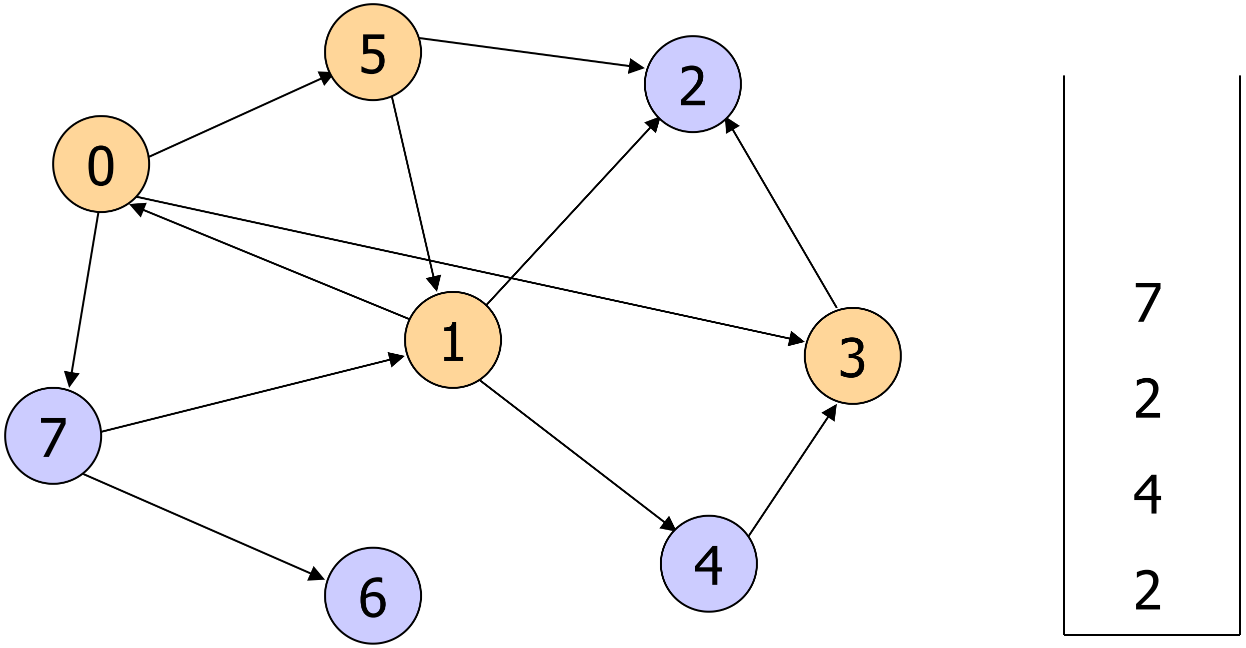
DFS: Start with Node 5



push(7), push(3)

5 1 0

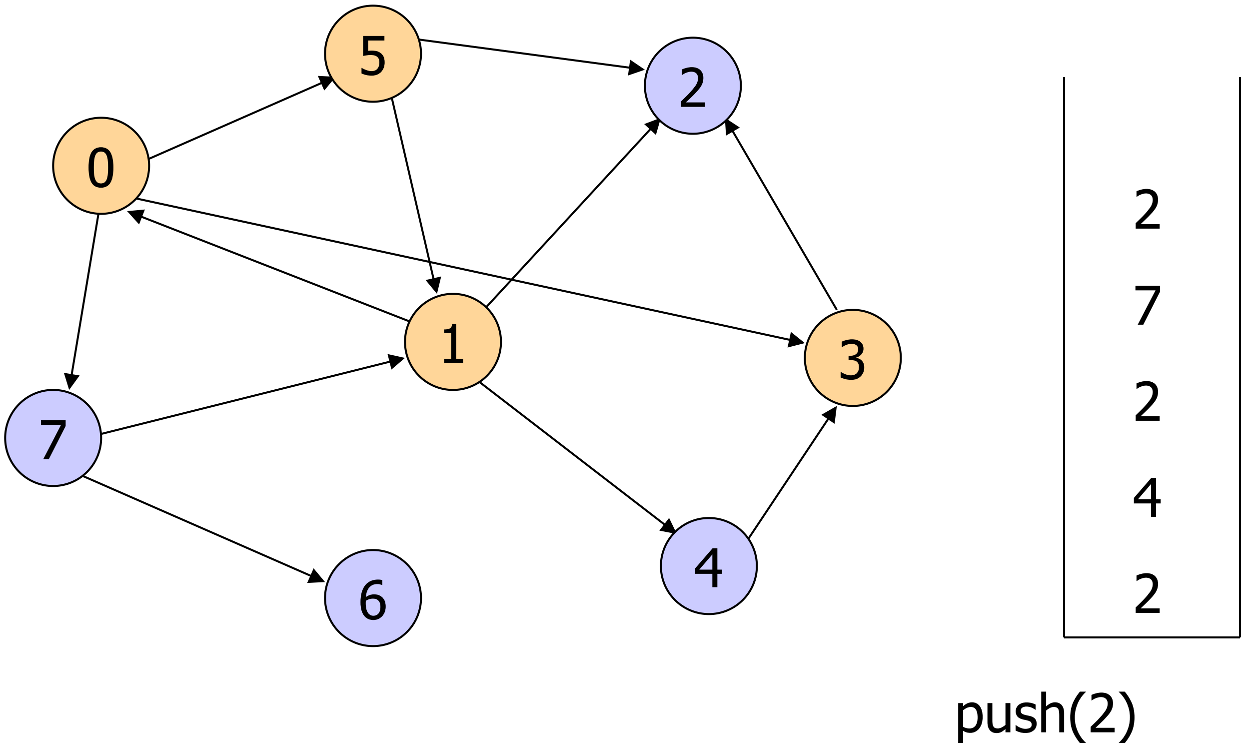
DFS: Start with Node 5



pop/visit/mark(3)

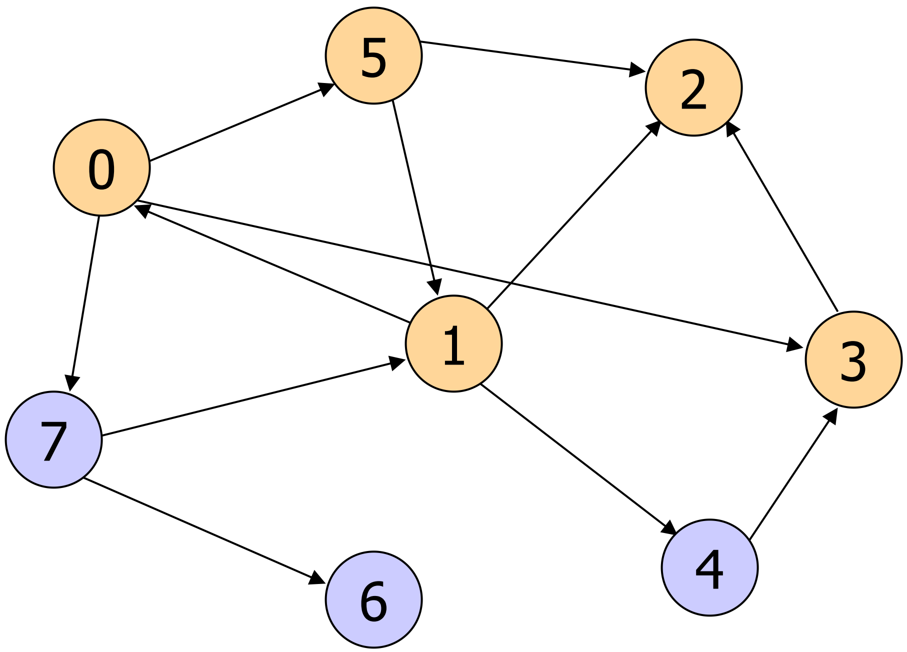
5 1 0 3

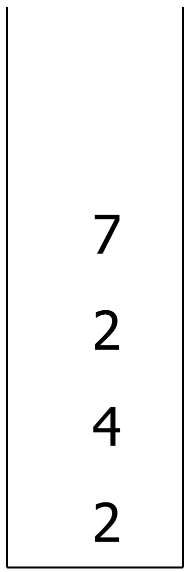
DFS: Start with Node 5



5 1 0 3

DFS: Start with Node 5

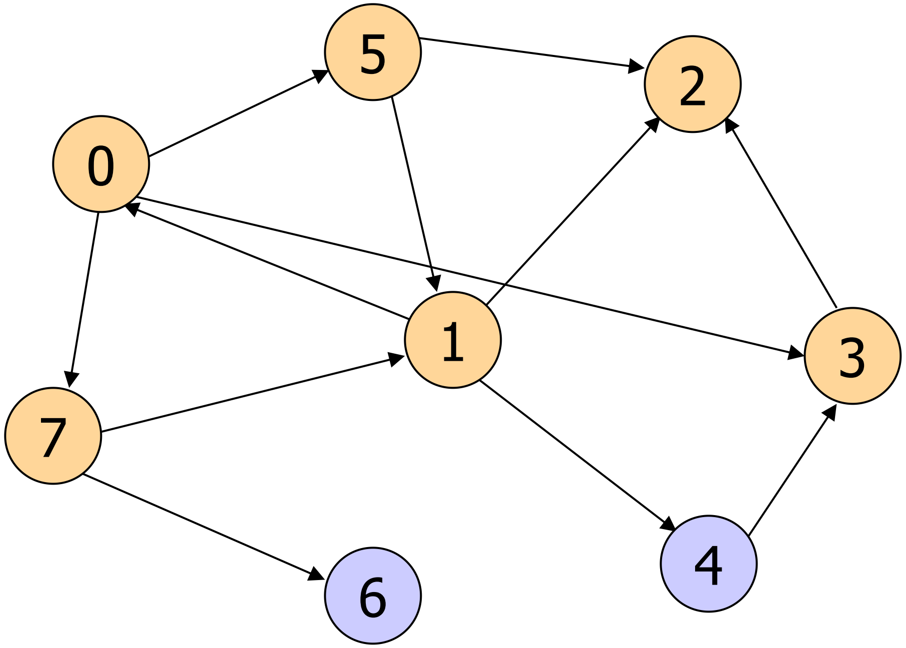


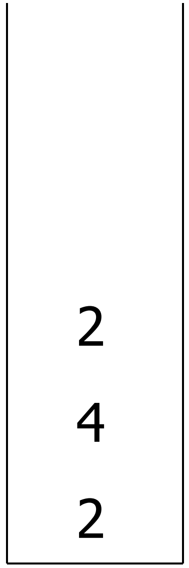


pop/mark/visit(2)

5 1 0 3 2

DFS: Start with Node 5

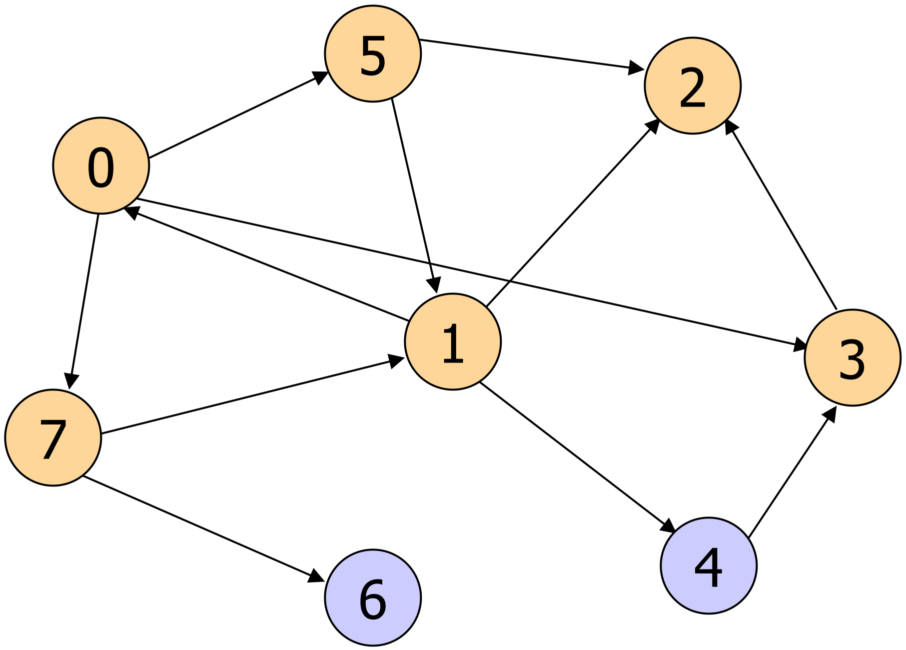




pop/mark/visit(7)

5 1 0 3 2 7

DFS: Start with Node 5



6

2

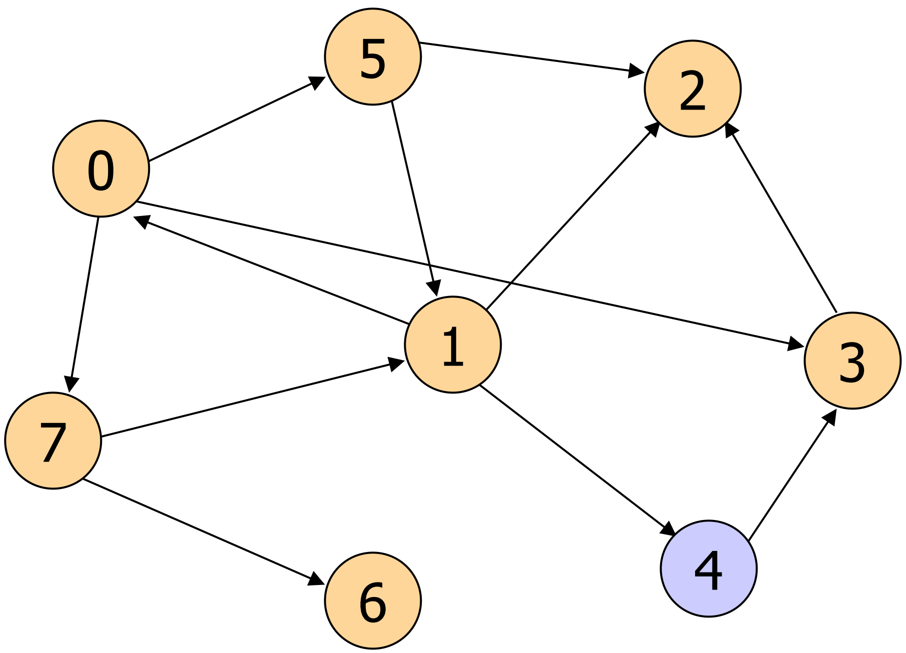
4

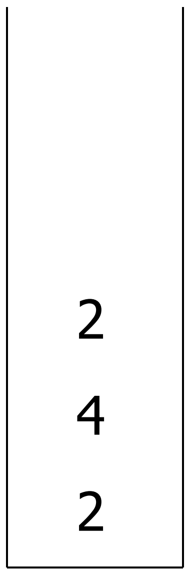
2

push(6)

5 1 0 3 2 7

DFS: Start with Node 5

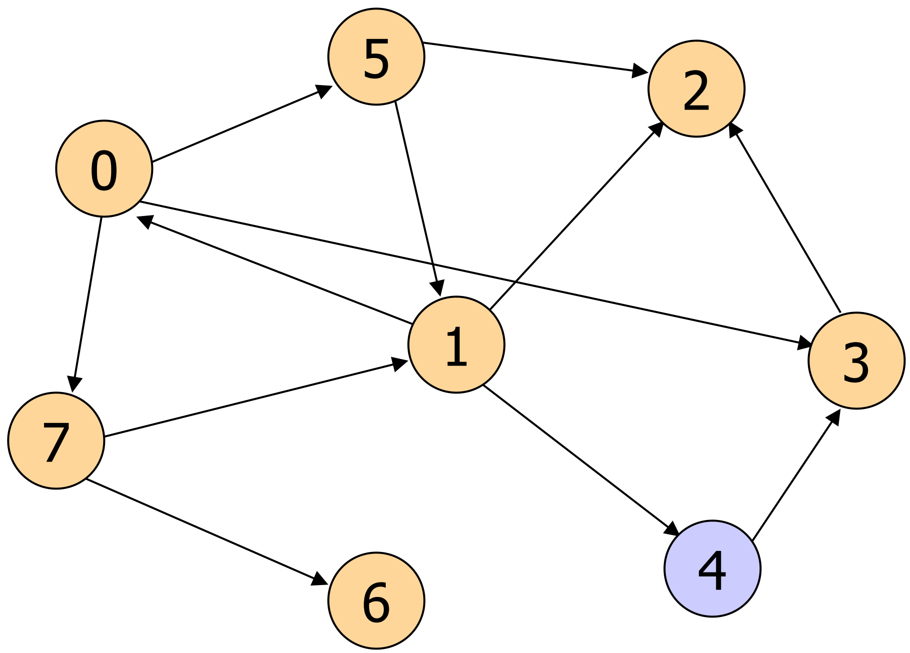


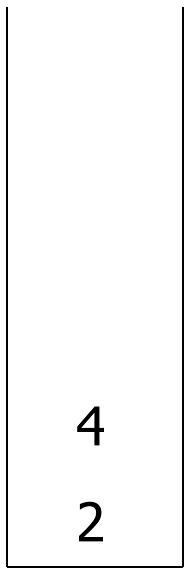


pop/mark/visit(6)

5 1 0 3 2 7 6

DFS: Start with Node 5

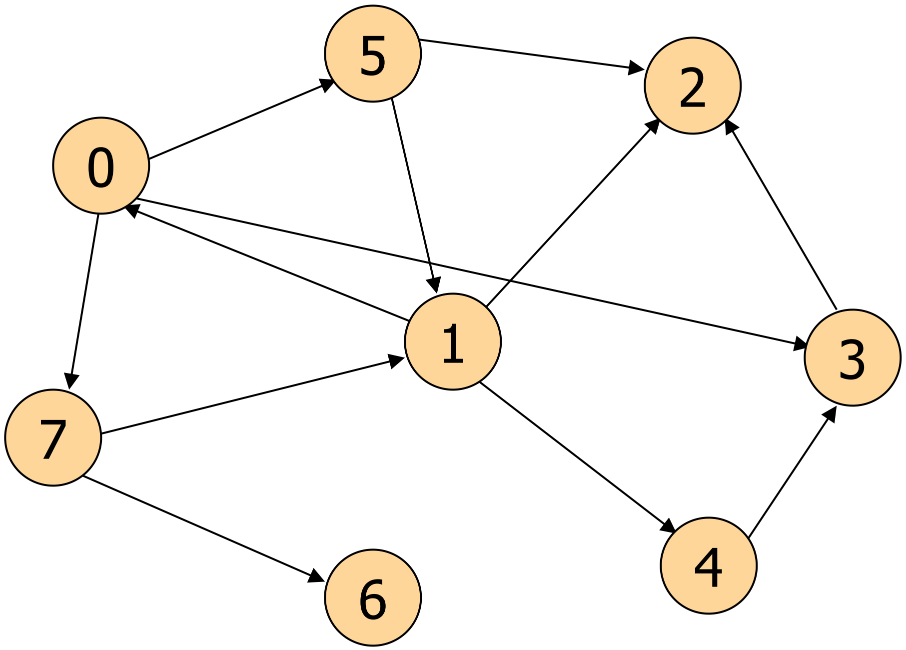


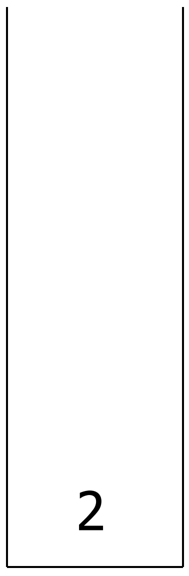


pop/do not visit(2)

5 1 0 3 2 7 6

DFS: Start with Node 5

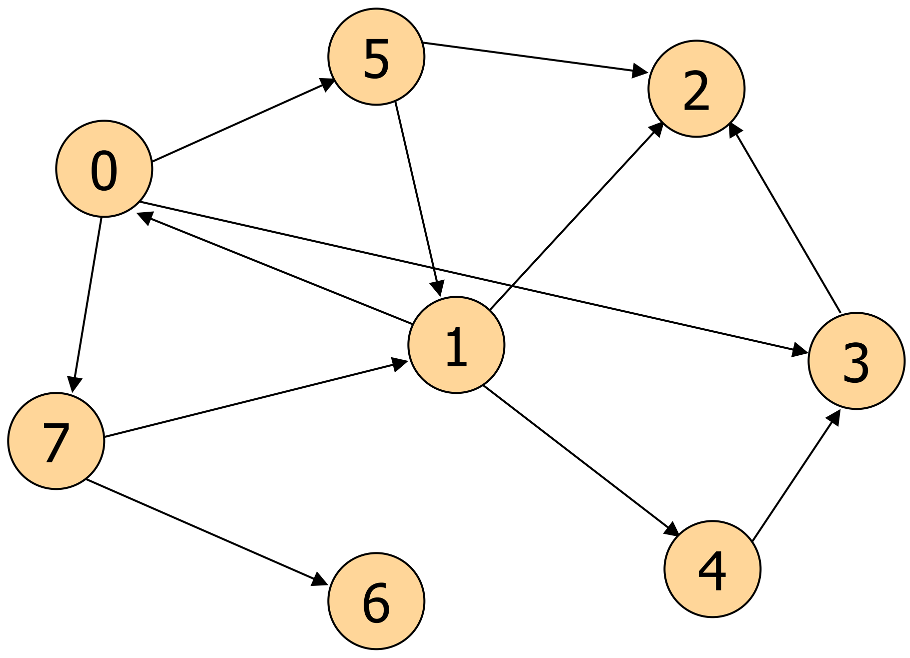


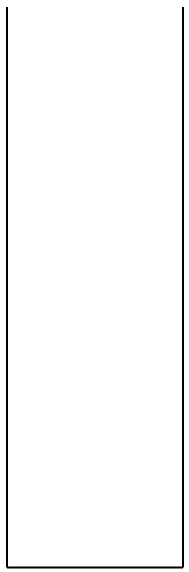


pop/mark/visit(6)

5 1 0 3 2 7 6 4

DFS: Start with Node 5

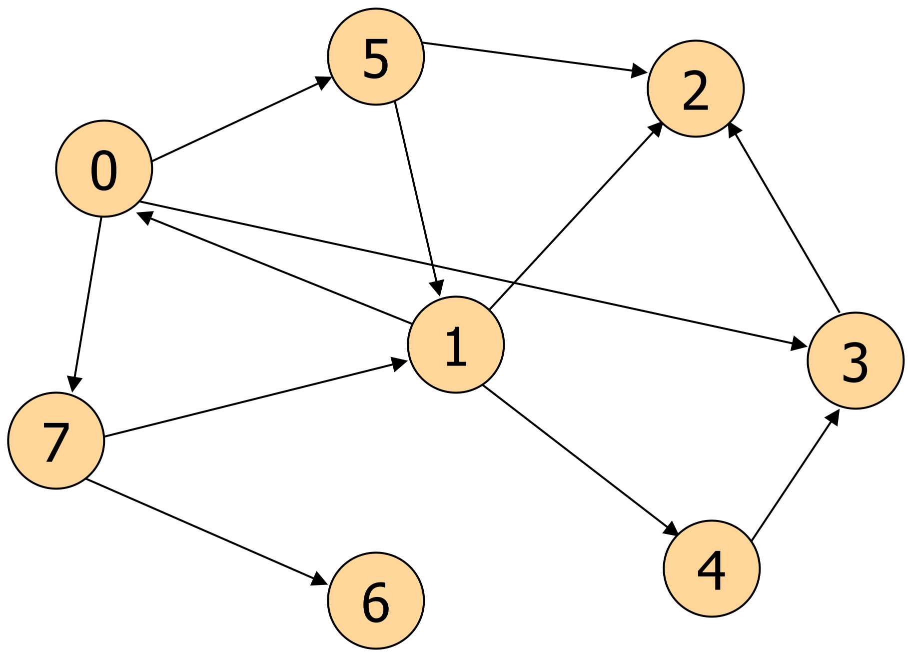




pop/do not visit(2)

5 1 0 3 2 7 6 4

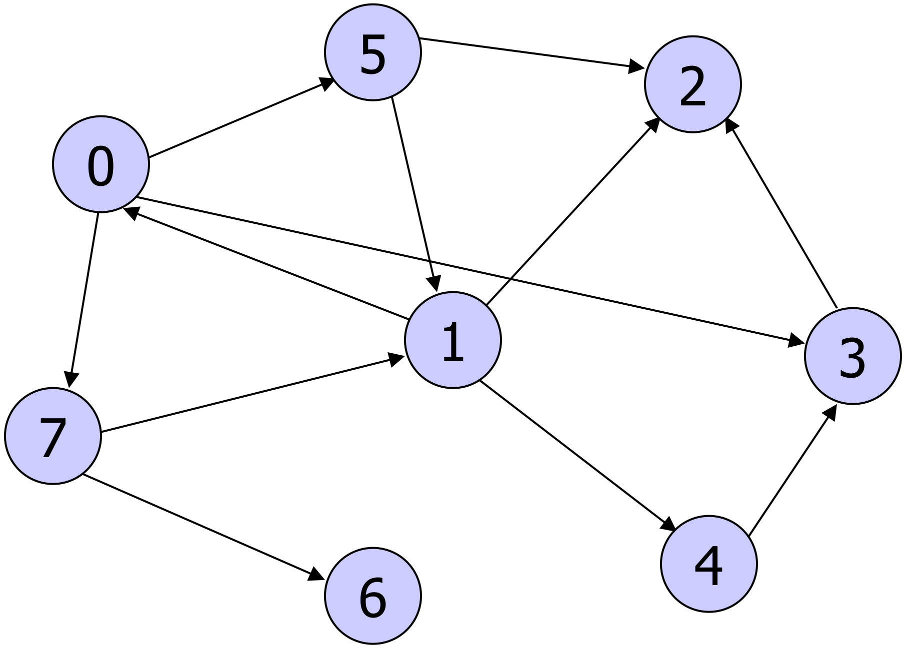
DFS: Start with Node 5



Stack empty, process completed

5 1 0 3 2 7 6 4

DFS: Start with Node 5
Note: edge (0, 3) removed

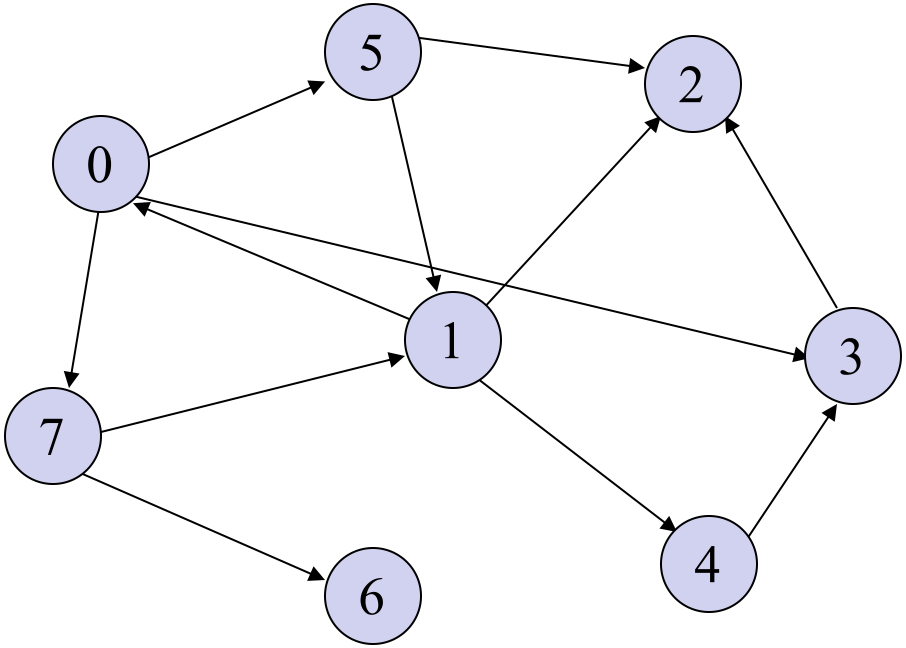


5 1 0 7 6 2 4 3

DFS (Pseudo Code) Policy: Don’t push nodes twice

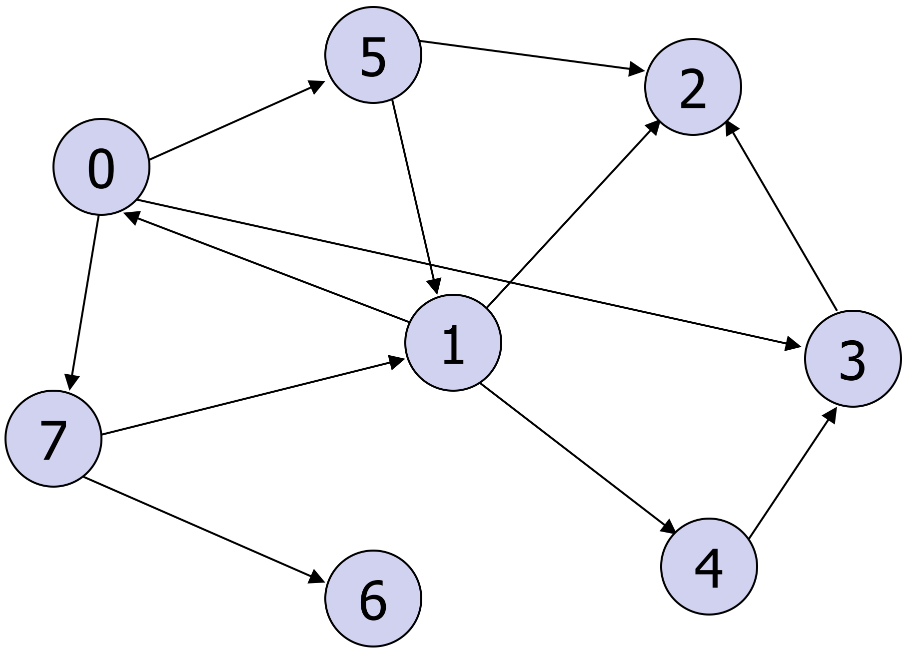
```
DFS(input: Graph G, Node v) {
    if (v = = NULL)
        return;
    push(v);
    while (stack is not empty) {
        pop(v);
        if (v has not yet been visited)
            mark_and_visit(v);
        for (each w adjacent to v)
            if (w has not yet been visited && not yet stacked)
                push(w);
    }
}
```

DFS (Don’t push nodes twice). Start with Node 5



5 1 0 3 7 6 4 2

DFS: Start with Node 5



2 3 6 7 0 4 1 5

Breadth-First Search

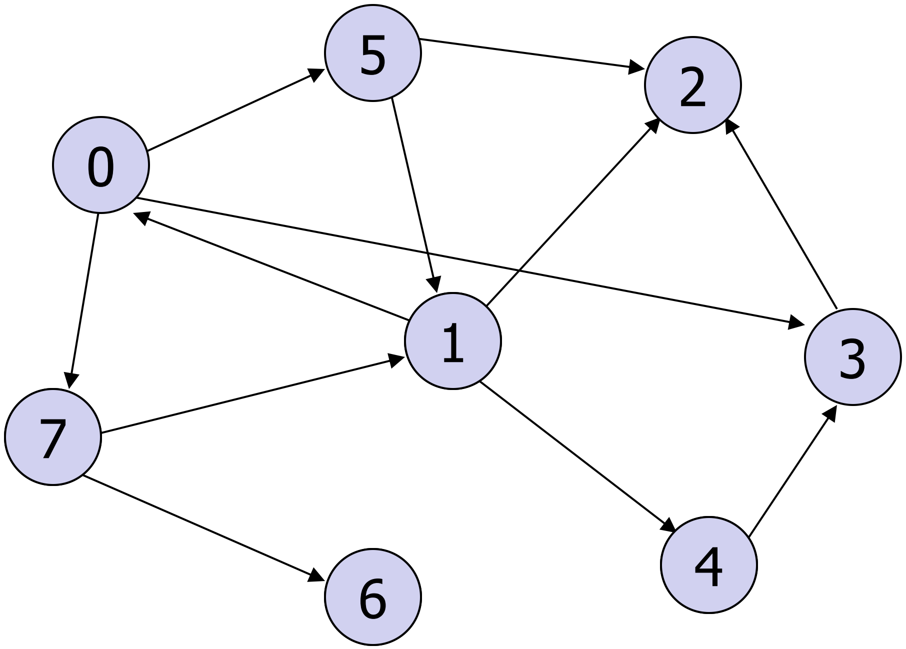
BFS follows the following rules:

1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
2. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z.
3. The newly visited nodes from this level form a new level that becomes the next current level.
4. Repeat step 2 & 3 until no more nodes can be visited.
5. If there are still unvisited nodes, repeat from Step 1.

BFS (Pseudo Code)

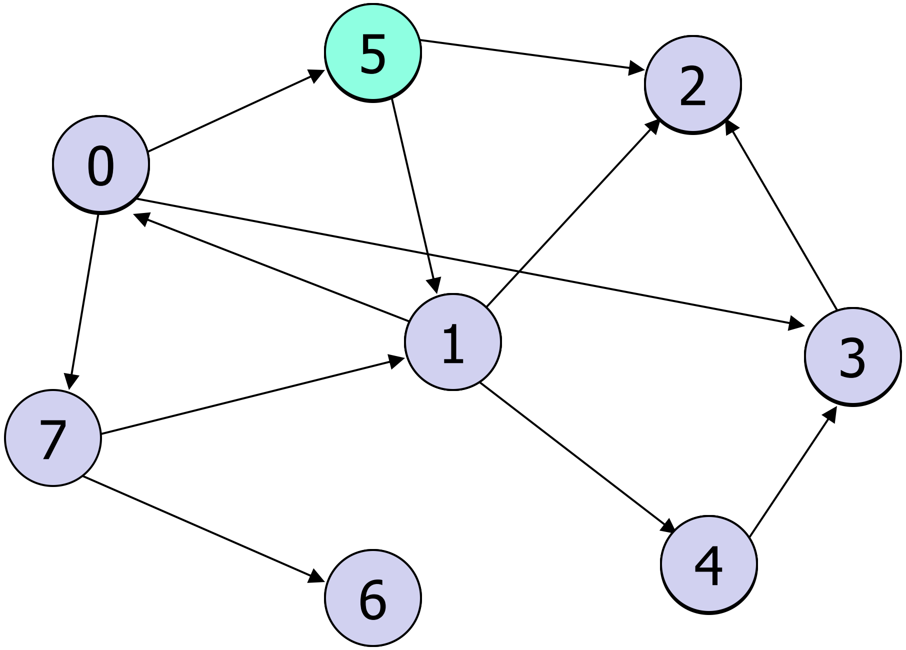
```
BFS(input: graph G, Node v) {
    Queue Q;   Integer x, z, y;
     if (v == NULL)    return;
     enqueue(v);
     while (queue is not empty) {
          dequeue(v);
          if (v has not yet been visited)
                mark_and_visit(v);
         for (each w adjacent to v)
               if (w has not yet been visited && has not been queued)
                    enqueue(w);
     }
}
```

BFS: Start with Node 5



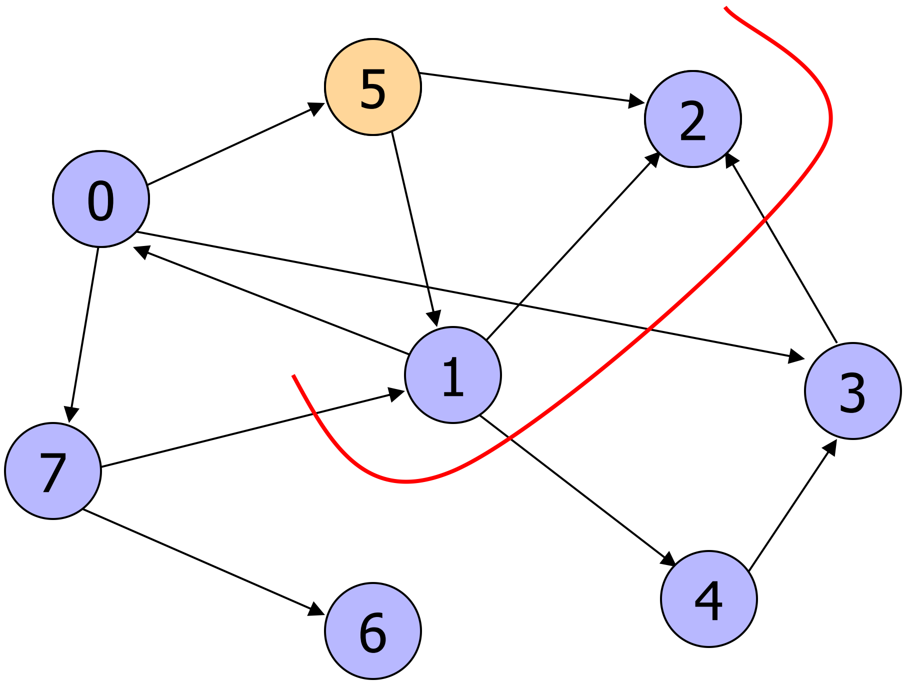
5 1 2 0 4 3 7 6

BFS: Start with Node 5



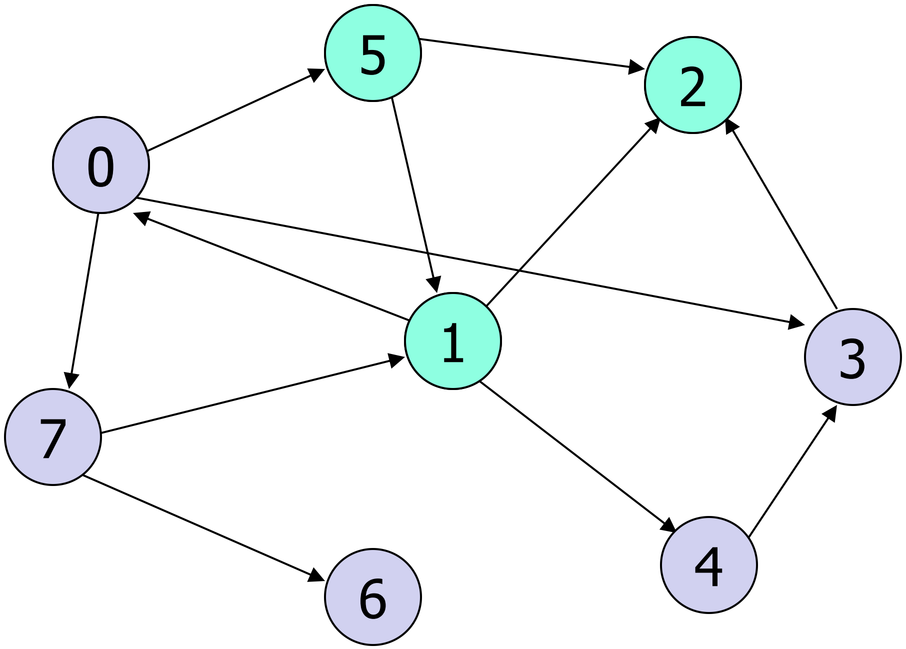
5

BFS: Node one-away



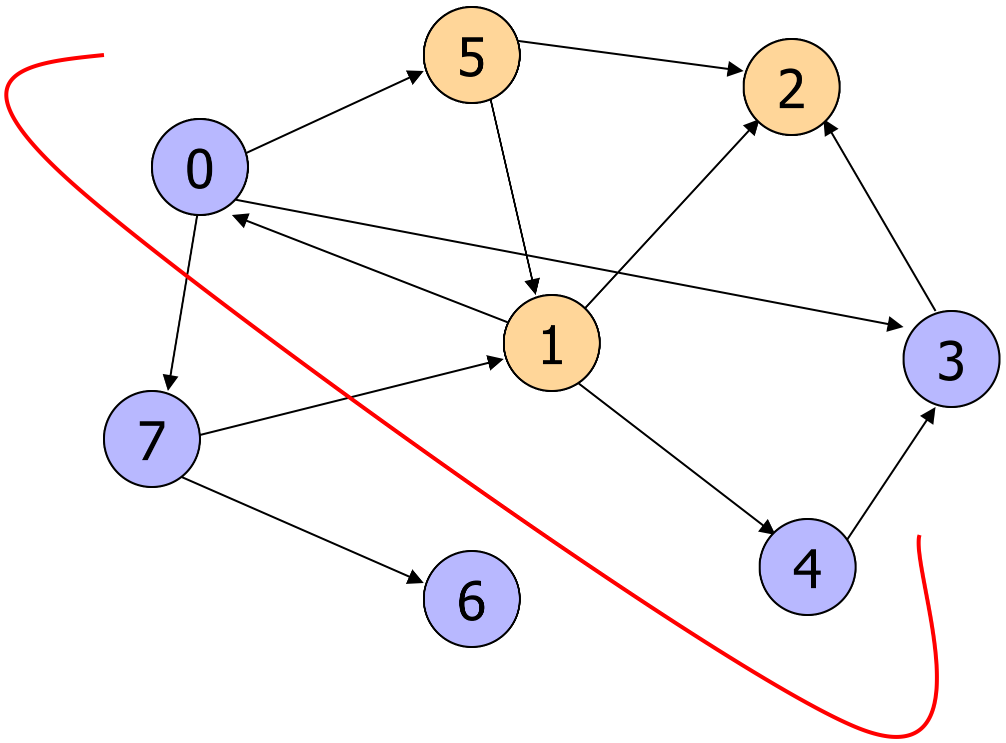
5

BFS: Visit 1 and 2



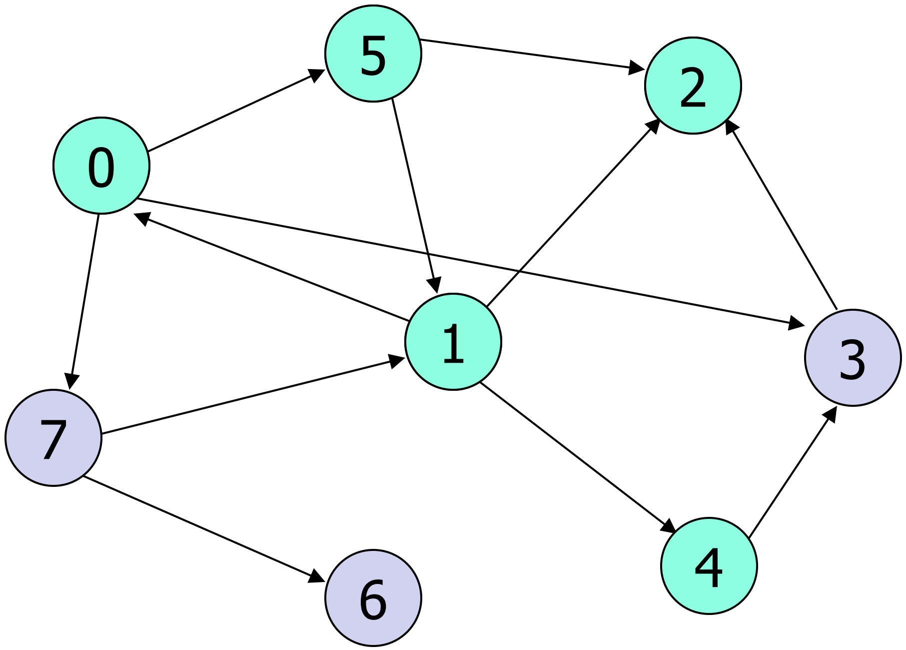
5 1 2

BFS: Nodes two-away



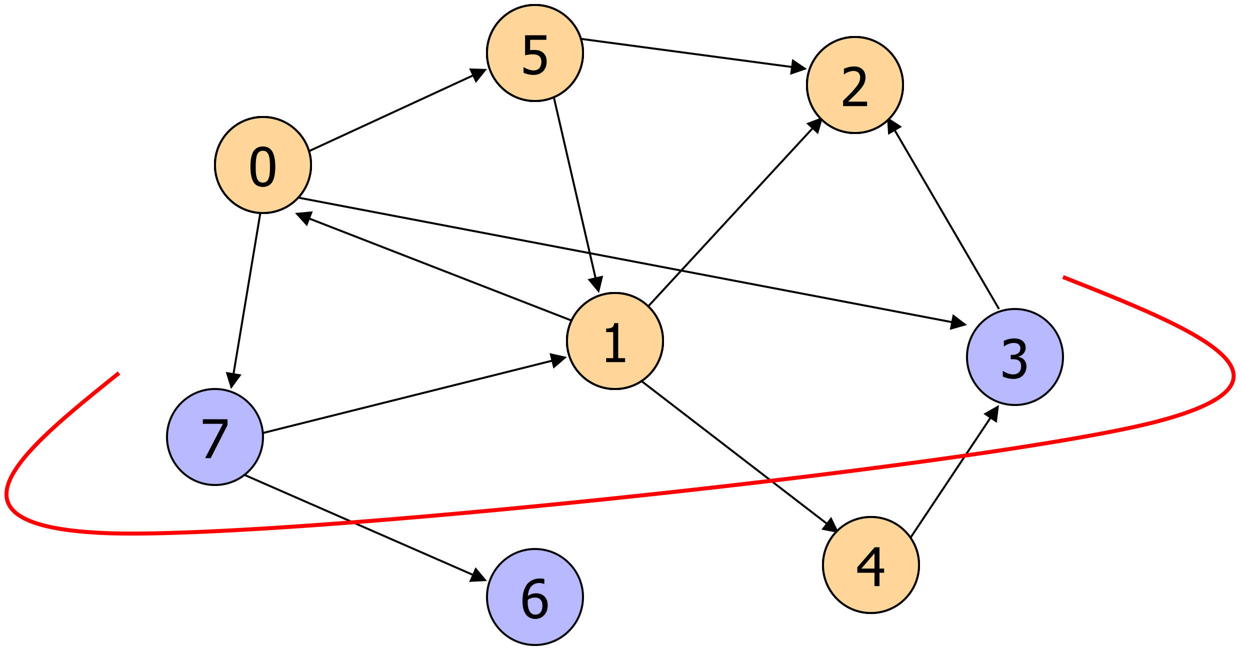
5 1 2

BFS: Visit 0 and 4



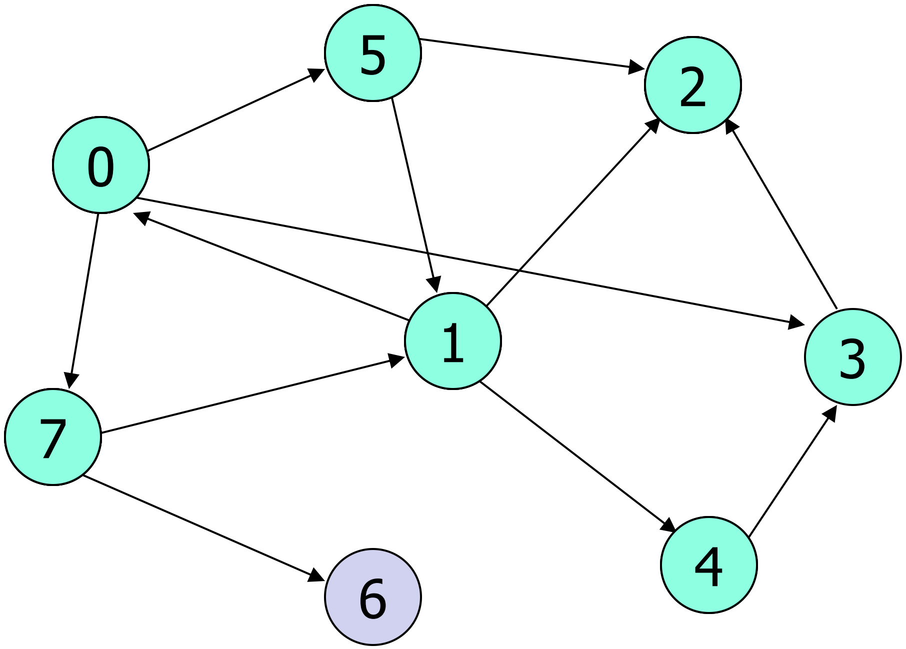
5 1 2 0 4

BFS: Nodes three-away



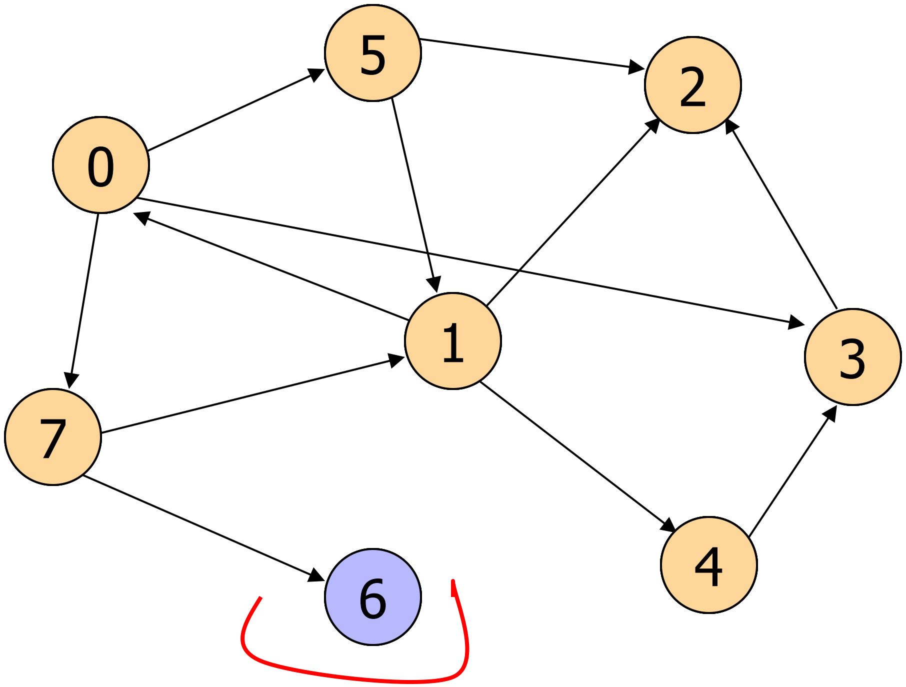
5 1 2 0 4

BFS: Visit nodes 3 and 7



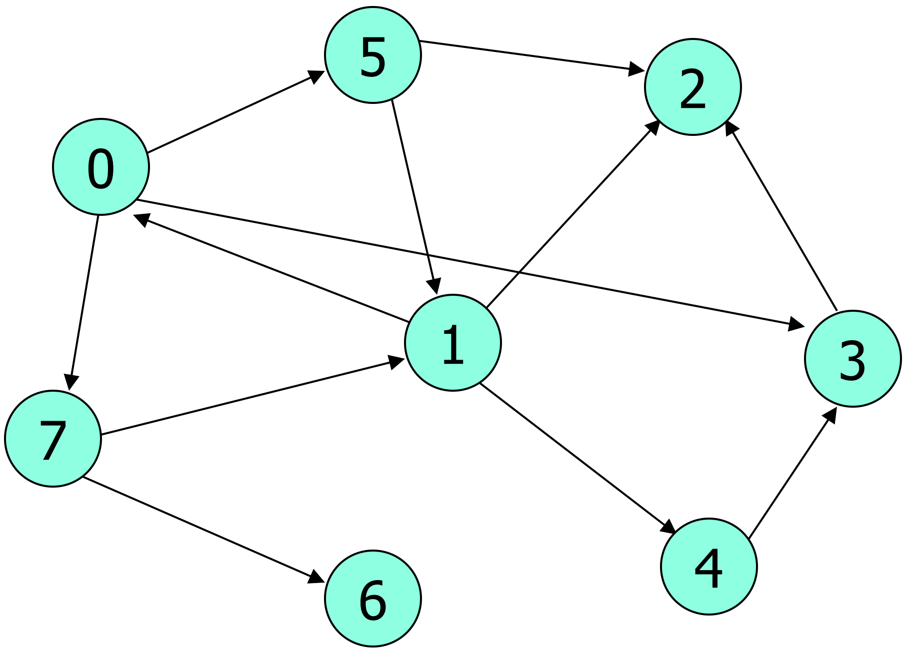
5 1 2 0 4 3 7

BFS: Node four-away



5 1 2 0 4 3 7

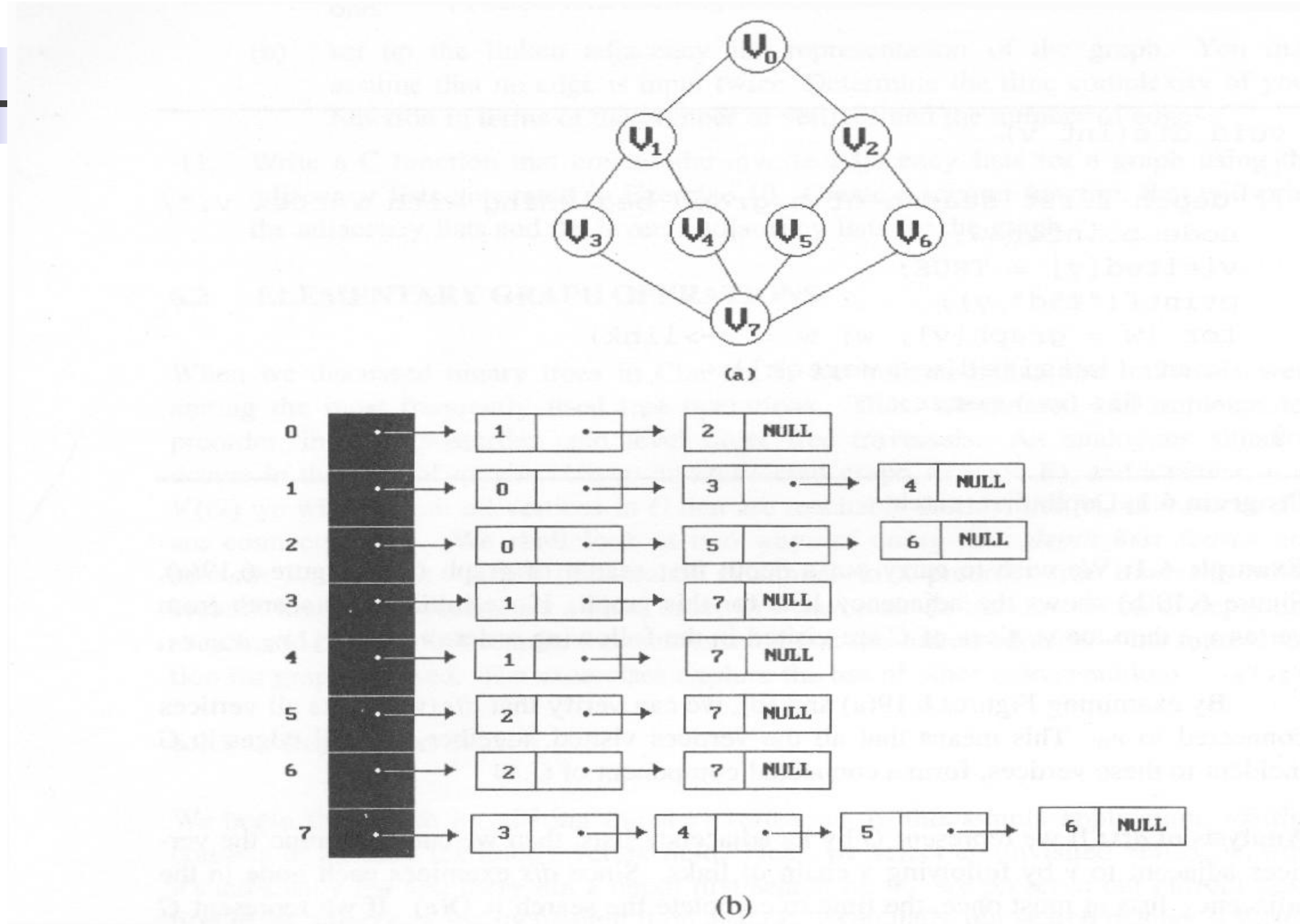
BFS: Visit 6



5 1 2 0 4 3 7 6

Graph G and its adjacency lists

depth first search: v0, v1, v3, v7, v4, v5, v2, v6



breadth first search: v0, v1, v2, v3, v4, v5, v6, v7

Topological sort

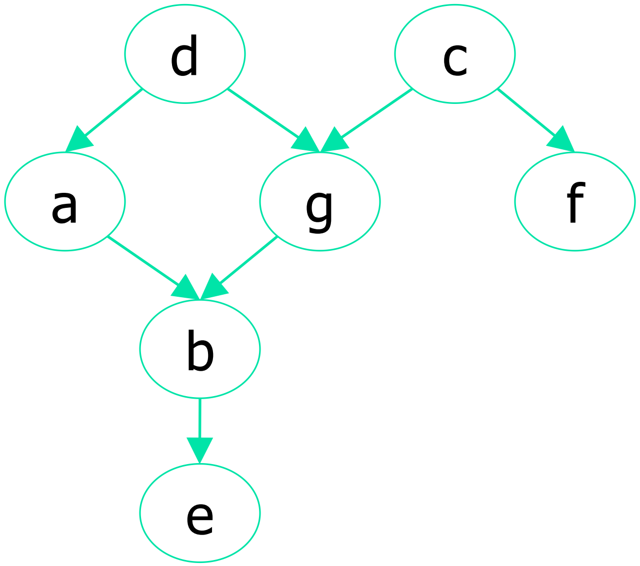
- Given a set of tasks and a set of dependencies (precedence constraints) of form "task A must be done before task B"
- Topological sort: An ordering of the tasks that conforms with the given dependencies
- Goal: Find a topological sort of the tasks or decide that there is no such ordering

Topological Sort

- Sorting technique over DAGs (Directed Acyclic Graphs)
- It creates a linear sequence (ordering) for the nodes such that:
  - If u has an outgoing edge to v ➔ then u must finish before v starts
- Very common in ordering jobs or tasks

Examples

- Scheduling: When scheduling *task graphs* in distributed systems, usually we first need to sort the tasks topologically
...and then assign them to resources.



Topological Sort Example

A job consists of 10 tasks with the following precedence rules:

Must start with 7, 5, 4 or 9.

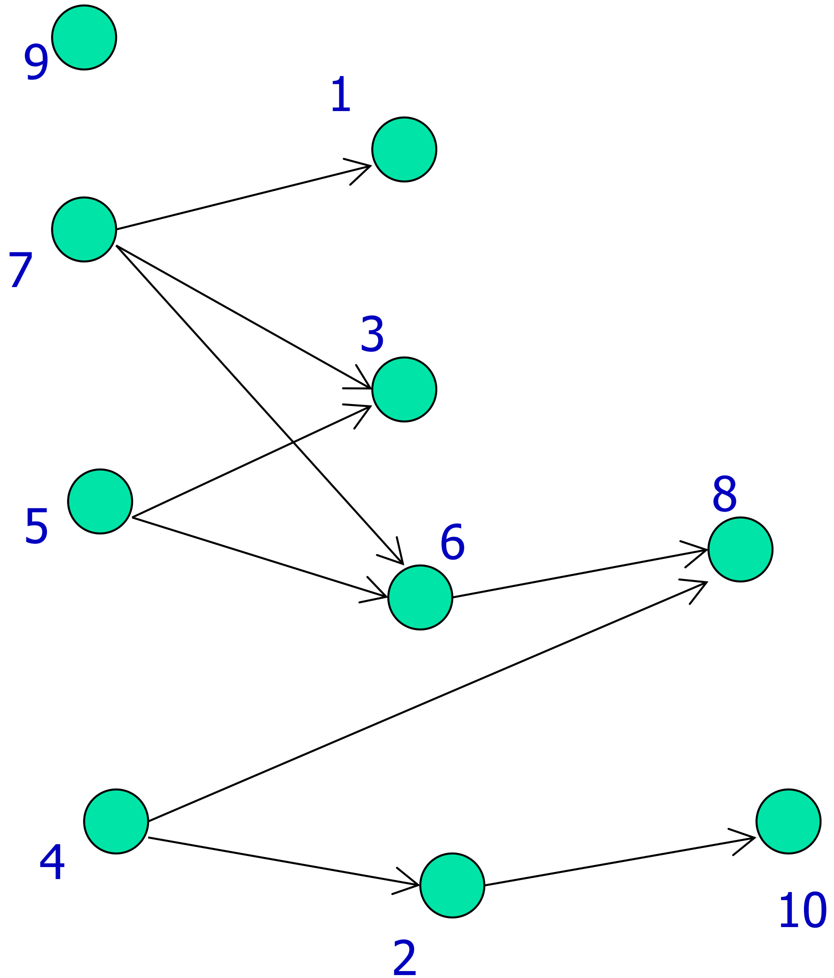
Task 1 must follow 7.

Tasks 3 & 6 must follow both 7 & 5.

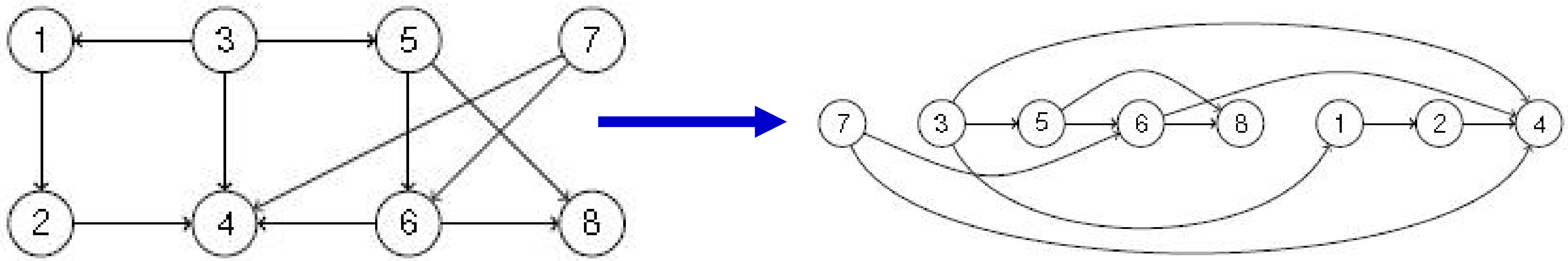
8 must follow 6 & 4.

2 must follow 4.

10 must follow 2.

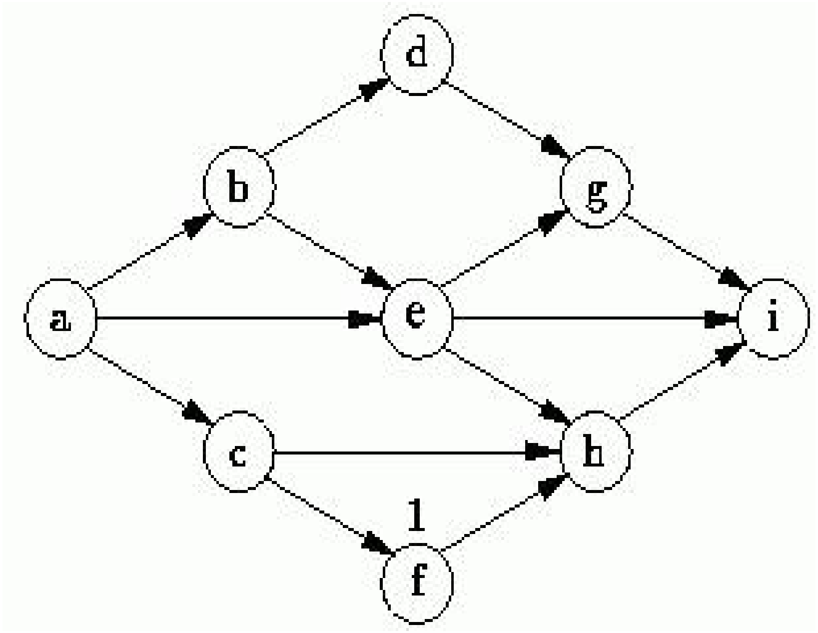


Topological Sort Example



Topological Sort is not unique

- Topological sort is not unique.
- The following are all topological sort of the graph below:



s1 = {a, b, c, d, e, f, g, h, i}

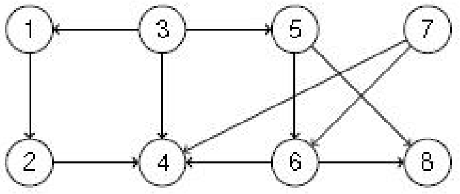
s2 = {a, c, b, f, e, d, h, g, i}

s3 = {a, b, d, c, e, g, f, h, i}

s4 = {a, c, f, b, e, h, d, g, i}

etc.

Topological Sort Algorithm

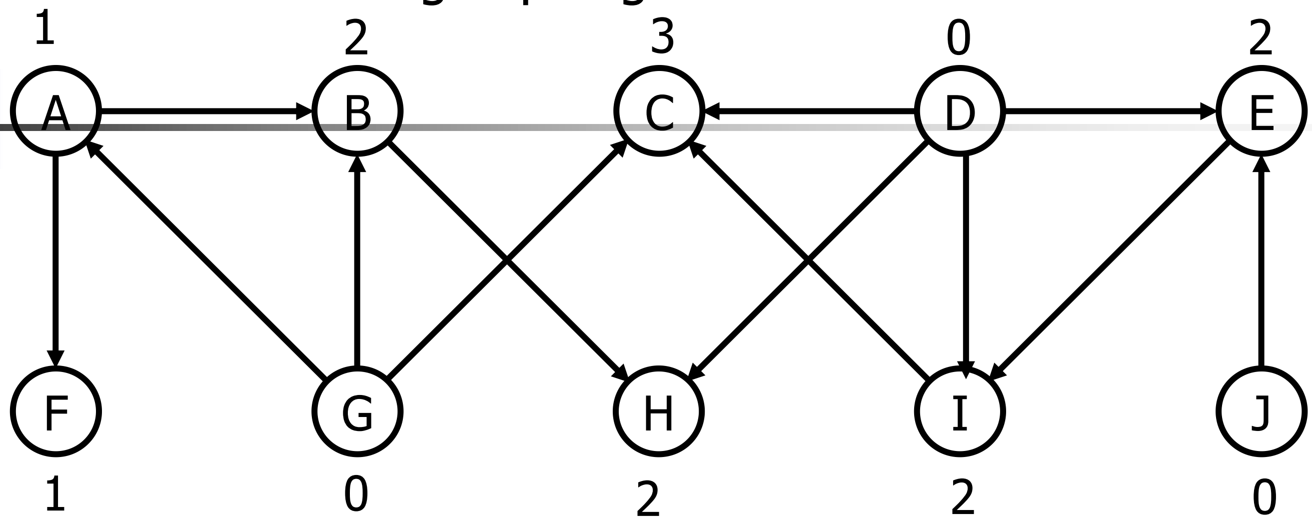


- One way to find a topological sort is to consider in-degrees of the vertices.
- The first vertex must have in-degree zero -- every DAG must have at least one vertex with in-degree zero.
- The Topological sort algorithm is:

```
int topologicalOrderTraversal( ) {
    int numVisitedVertices = 0;
    while(there are more vertices to be visited){
        if(there is no vertex with in-degree 0)
             break;
        else {
               select a vertex v that has in-degree 0;
               visit v;
               numVisitedVertices++;
               delete v and all its emanating edges;
        }
    }
   return numVisitedVertices;
}
```

Topological Sort Example

- Demonstrating Topological Sort.



D	G	A	B	F	H	J	E	I	C