Graphs

Graph

- A graph is a collection of nodes (or vertices) and edges (or arcs)
  - Each node contains an element
  - Each edge connects two nodes together (or possibly the same node to itself) and may contain an edge attribute

- A directed graph is one in which the edges have a direction
- An undirected graph (*graph*) is one in which the edges do not have a direction

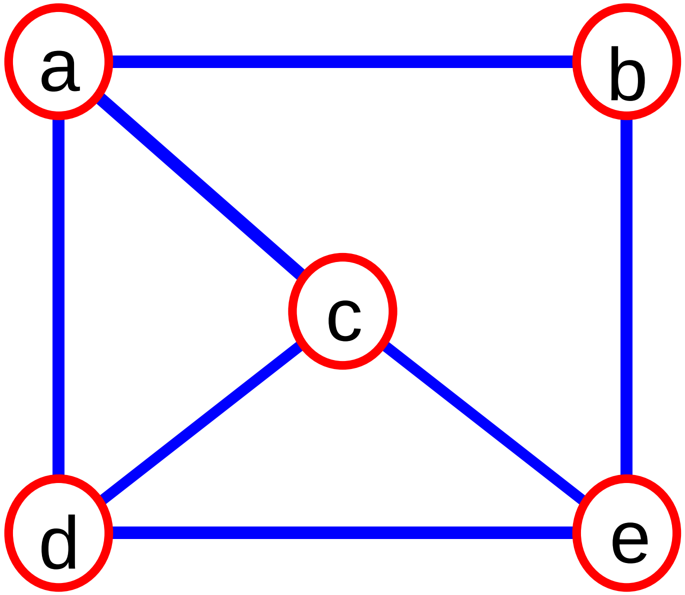
What is a Graph?

- A graph G = (V,E) is composed of:

    V: set of vertices

    E: set of edges connecting the vertices in V

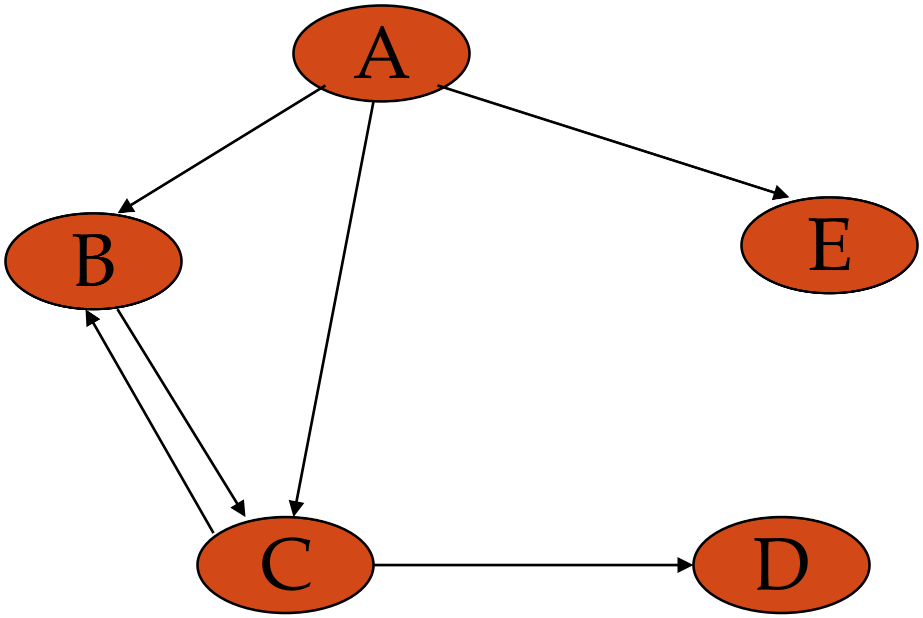
- An edge e = (u,v) is a pair of vertices
- Example:



V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d),
(b,e),(c,d),(c,e),(d,e)}

a digraph (Oriented or Directed graph)



V = [A, B, C, D, E]

E = [<A,B>, <B,C>, <C,B>, <A,C>, <A,E>, <C,D>]

Directed vs Undirected graph

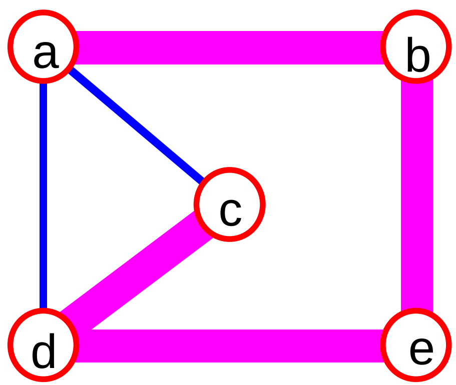
- An undirected graph is one in which the pair of vertices in a edge is unordered, (v0, v1) = (v1,v0)
- A directed graph is one in which each edge is a directed pair of vertices, <v0, v1> != <v1,v0>

Graph terminologies

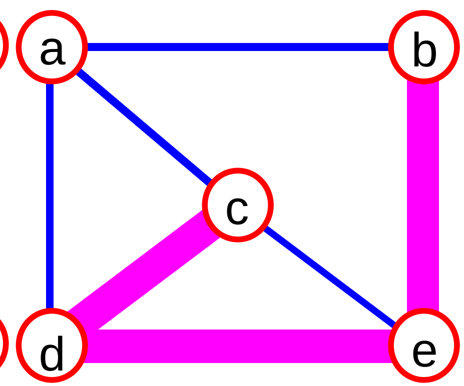
- The size of a graph is the number of *nodes* in it
- The empty graph has size zero (no nodes)
- If two nodes are connected by an edge, they are neighbors (and the nodes are adjacent to each other)
- The degree of a node is the number of edges it has
- For directed graphs,
  - If a directed edge goes from node S to node D, we call S the source and D the destination of the edge
    - The edge is an out-edge of S and an in-edge of D
    - S is a predecessor of D, and D is a successor of S
  - The in-degree of a node is the number of in-edges it has
  - The out-degree of a node is the number of out-edges it has

Graph terminologies

- path: sequence of vertices $v_1, v_2, v_3 \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.



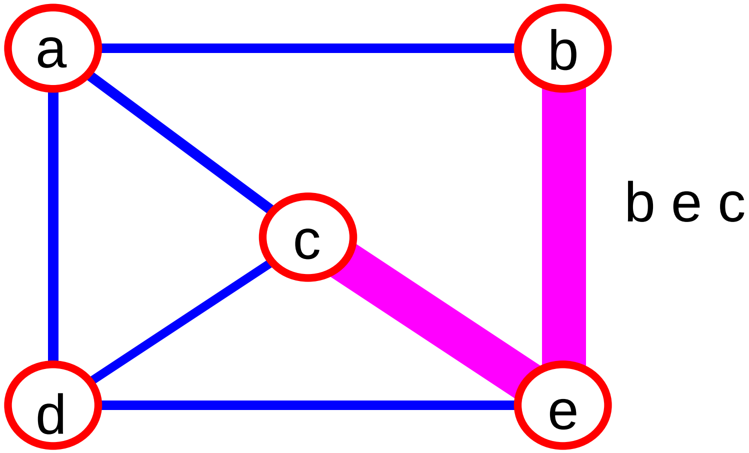
a b e d c



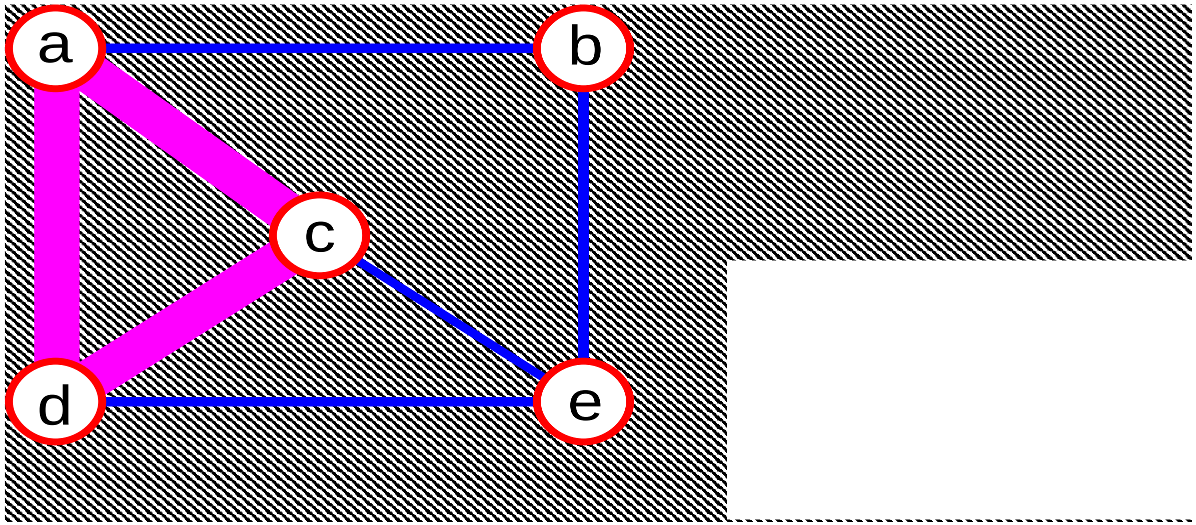
b e d c

More Terminology

- simple path: no repeated vertices

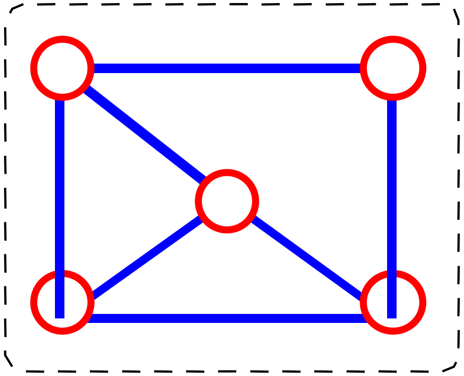


- cycle: simple path, except that the last vertex is the same as the first vertex

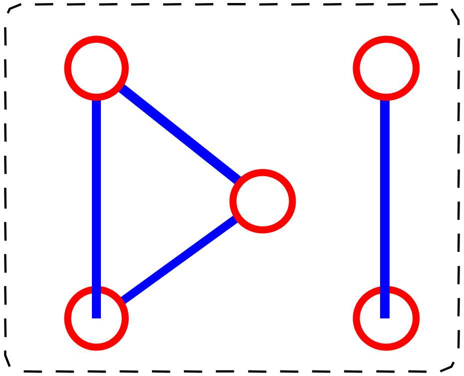


Even More Terminology

- connected graph: any two vertices are connected by some path



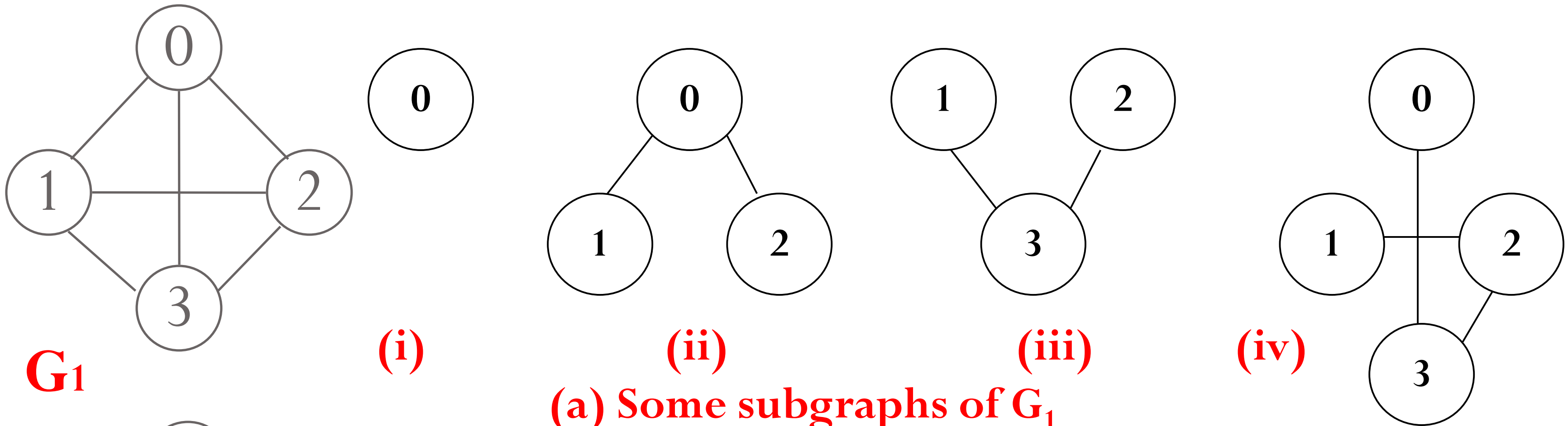
connected



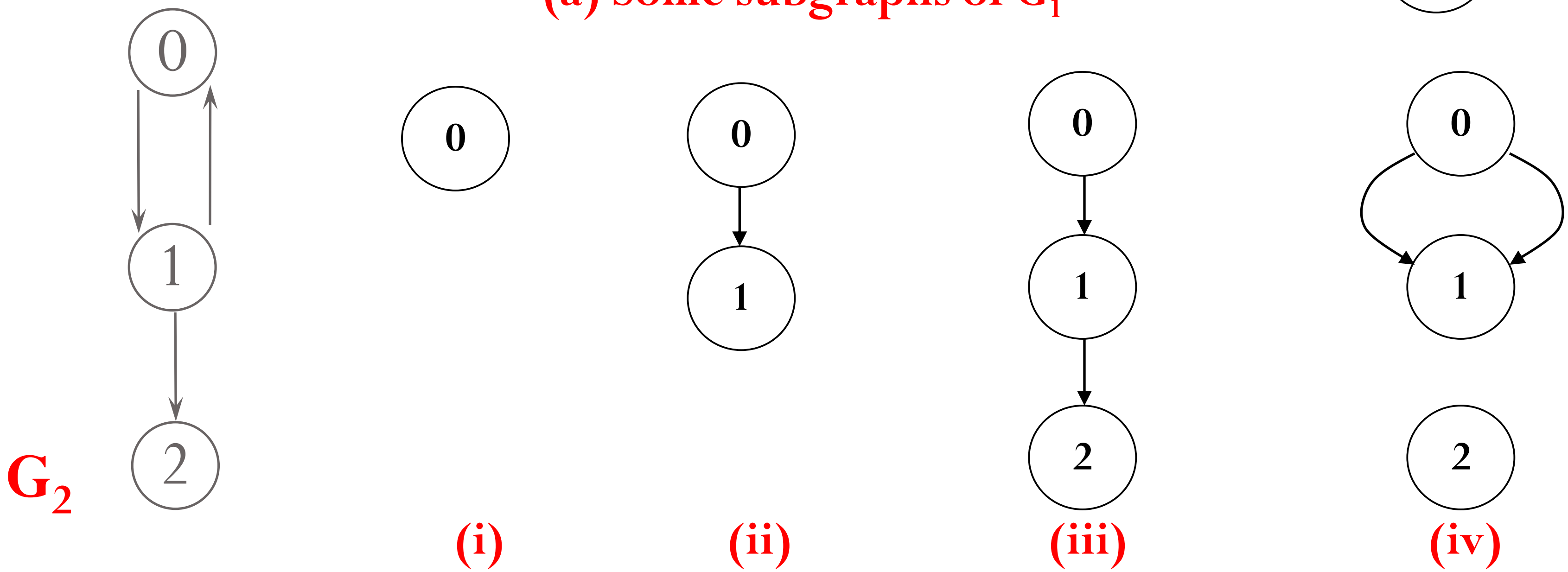
not connected

- subgraph: subset of vertices and edges forming a graph

Subgraph Examples



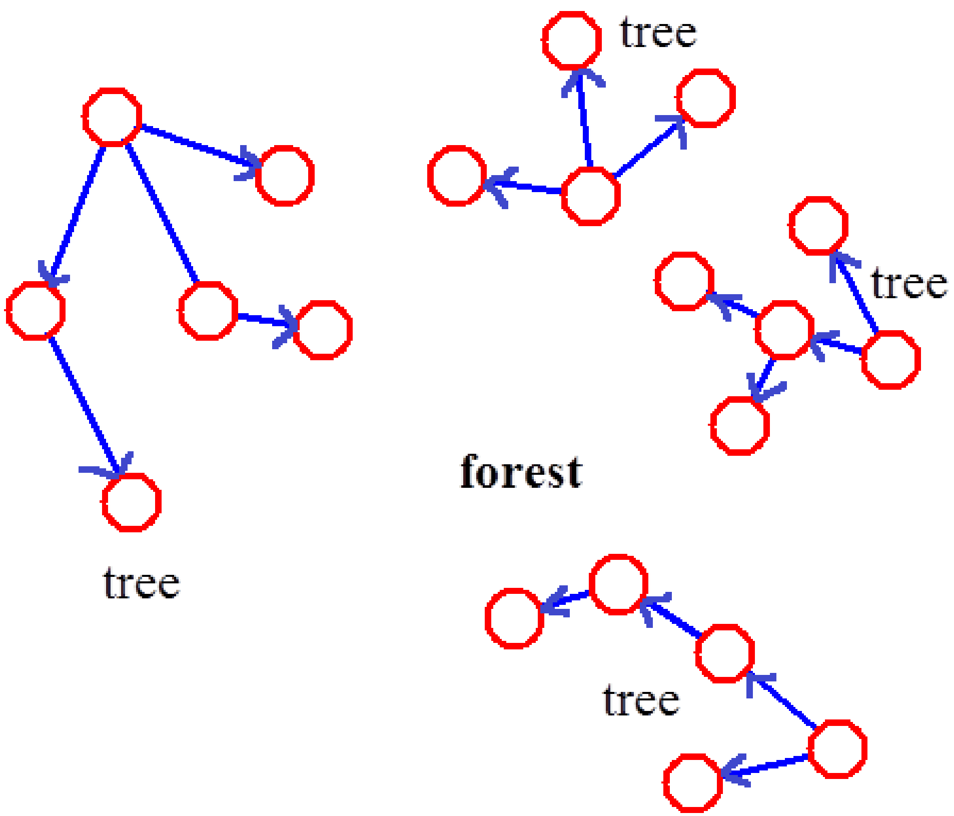
(a) Some subgraphs of G_1



(b) Some subgraphs of G_2

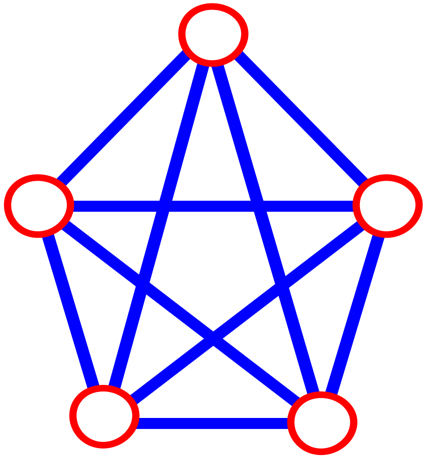
More…

- tree - connected digraph without cycles
- forest - collection of trees



Connectivity

- Let **n** = #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n**-1 edges, however, we would have counted each edge twice! Therefore, intuitively, m = **n**(**n** - 1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



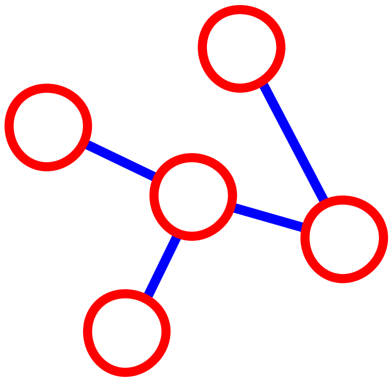
n = 5
m = (5 ∗ 4)/2 = 10

More on Connectivity

n = #vertices

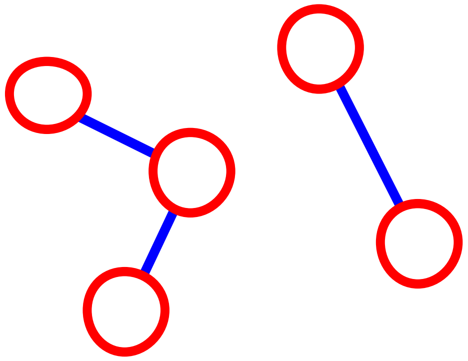
m = #edges

- For a tree $m = n - 1$



$n = 5$
$m = 4$

If $m < n - 1$, G is not connected



$n = 5$
$m = 3$

Graph Terminologies

- An undirected graph is **connected** if there is a path from every node to every other node
- A *directed graph* is **strongly connected** if there is a path from every node to every other node
- A directed graph is **weakly connected** if it is not strongly connected but the underlying undirected graph is connected
- Node X is **reachable** from node Y if there is a path from Y to X

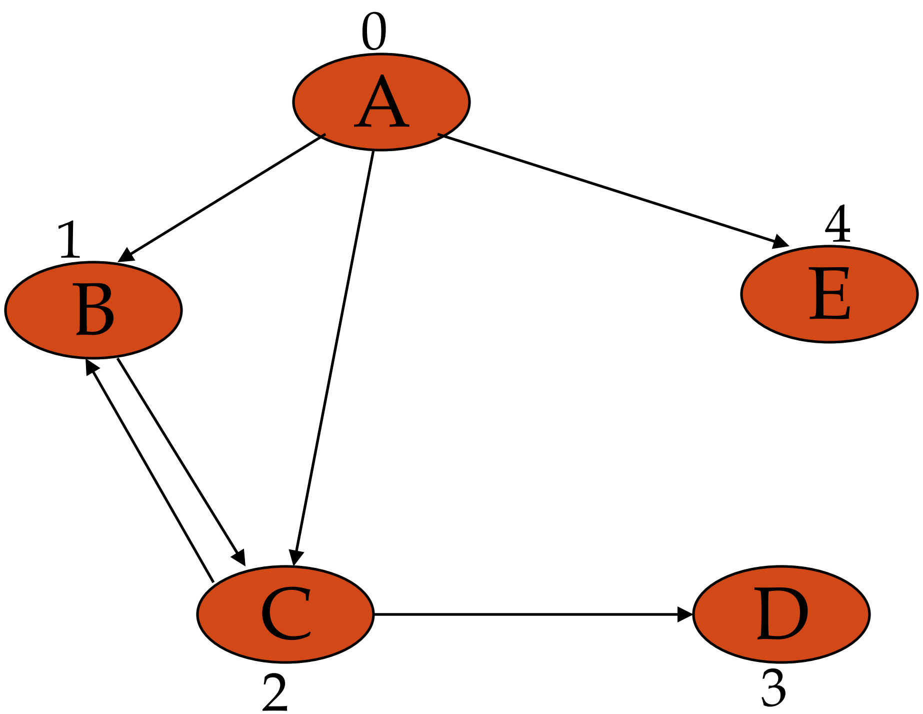
graph data structures

- storing the vertices
  - each vertex has a unique identifier and, maybe, other information
  - for efficiency, associate each vertex with a number that can be used as an index
- storing the edges
  - adjacency matrix – represent all possible edges
  - adjacency lists – represent only the existing edges

storing the vertices

- when a vertex is added to the graph, assign it a number
  - vertices are numbered between 0 and n-1
- graph operations start by looking up the number associated with a vertex
- many data structures to use
  - for small graphs a vector can be used
    - search will be O(n)

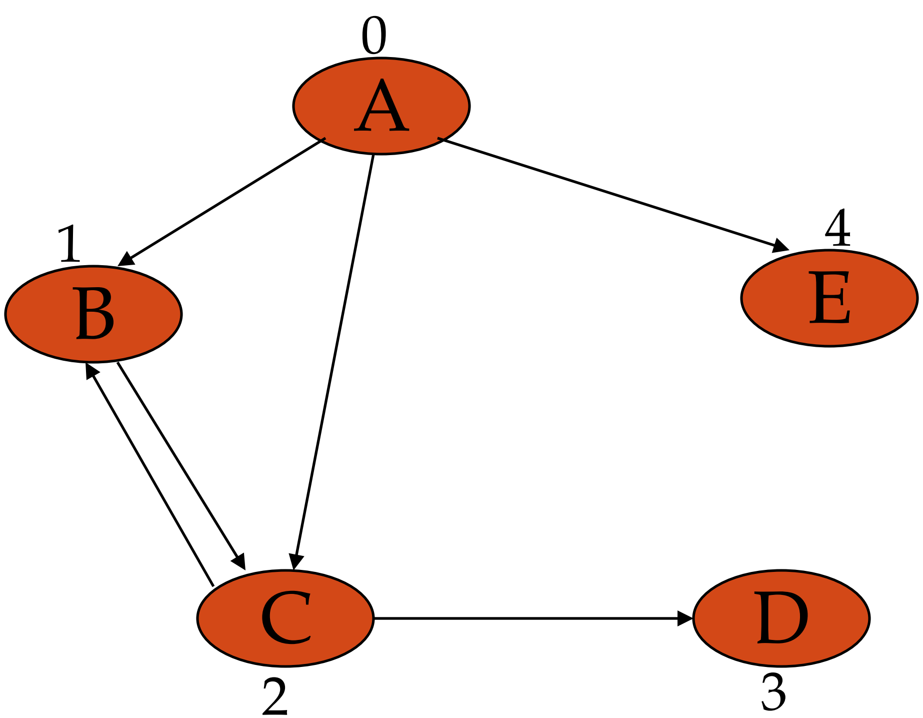
the vertex vector



0	A
1	B
2	C
3	D
4	E

adjacency matrix

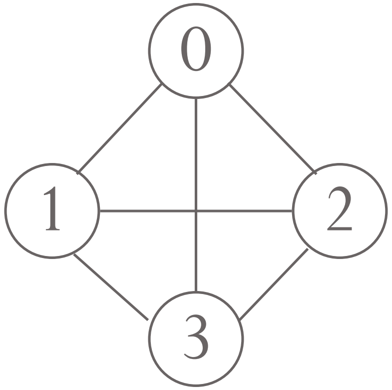
```
A_nxn, where n:# of vertices
A[i][j]=1 if (i,j) is an edge
        =0 otherwise
```



0	A
1	B
2	C
3	D
4	E

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

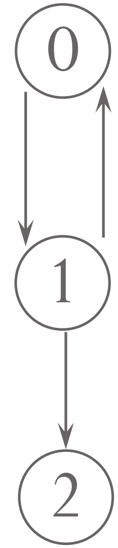
Examples for Adjacency Matrix



G_1

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

symmetric

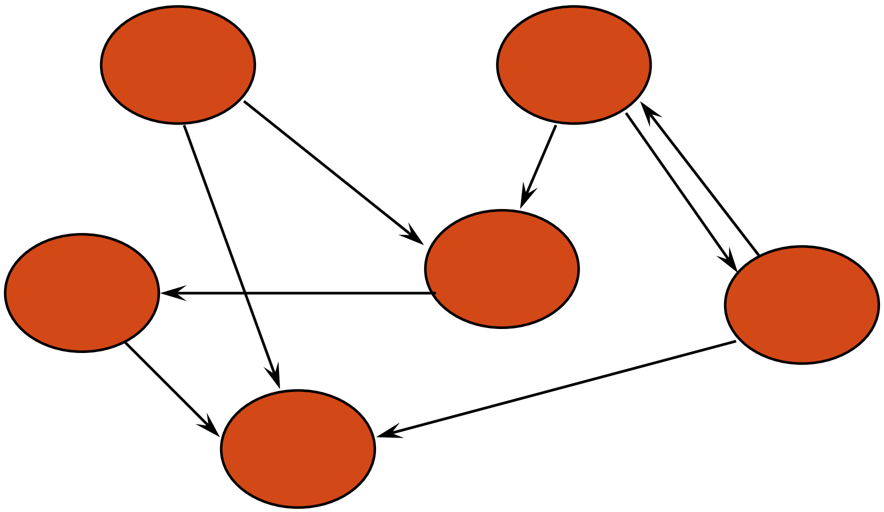


G_2

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

Asymmetric

Space required

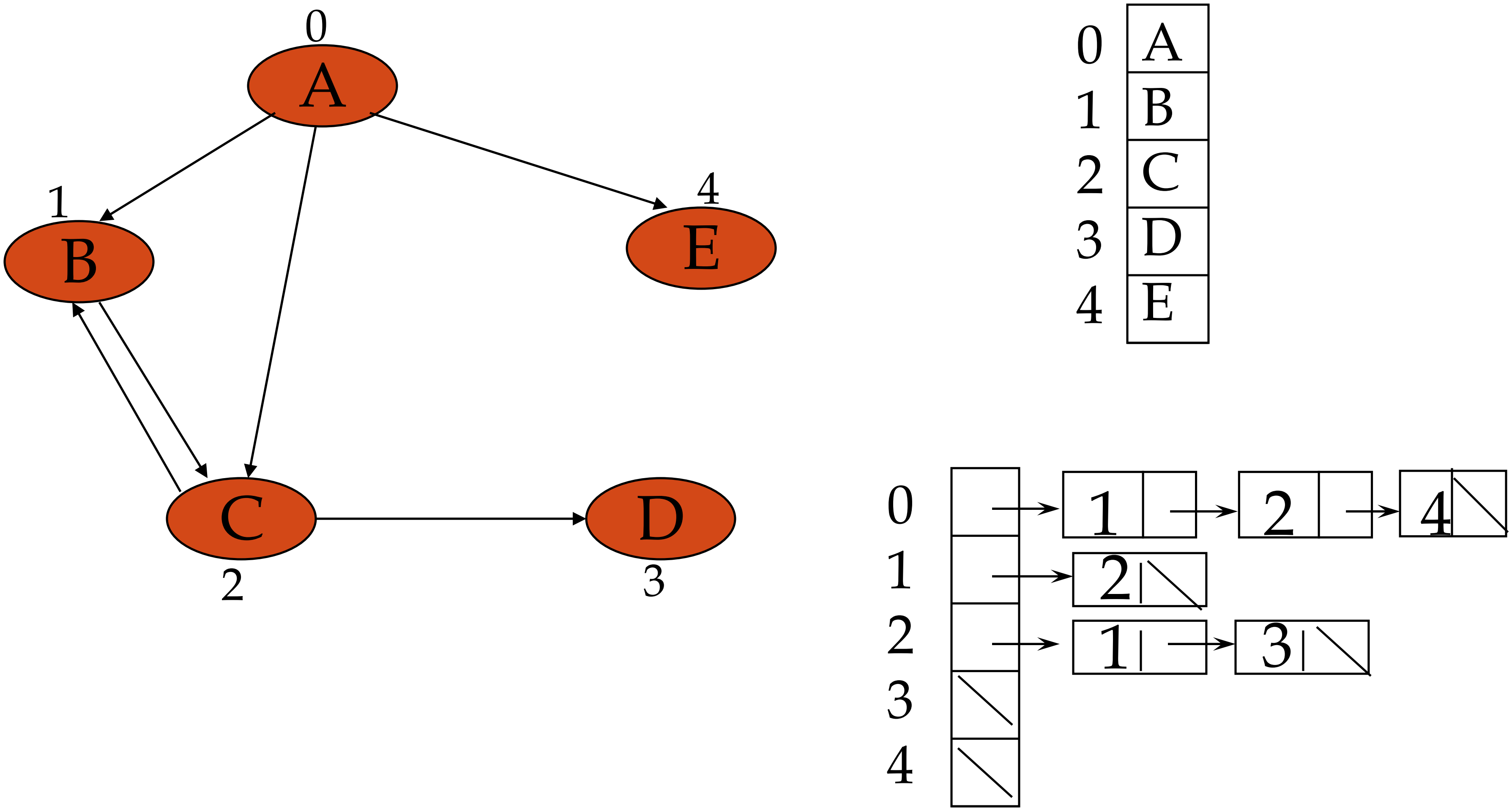


a n^2 matrix is needed for a graph with n vertices

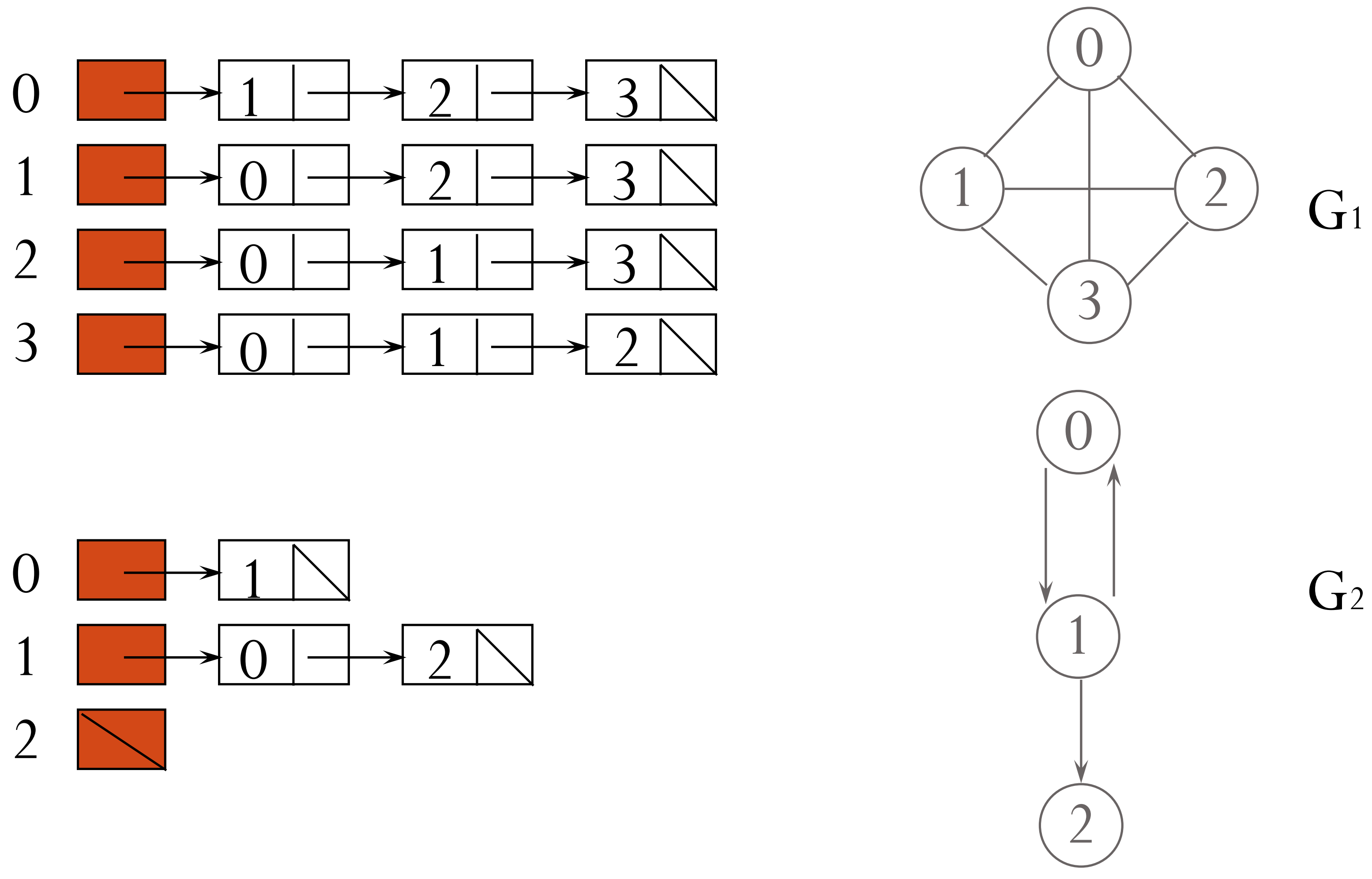
many graphs are “sparse”

- degree of “sparseness” is key factor in choosing a data structure for edges
  - **adjacency matrix** requires space for **all possible** edges
  - **adjacency list** requires space for **existing edges only**
- affects amount of memory space needed
- affects efficiency of graph operations

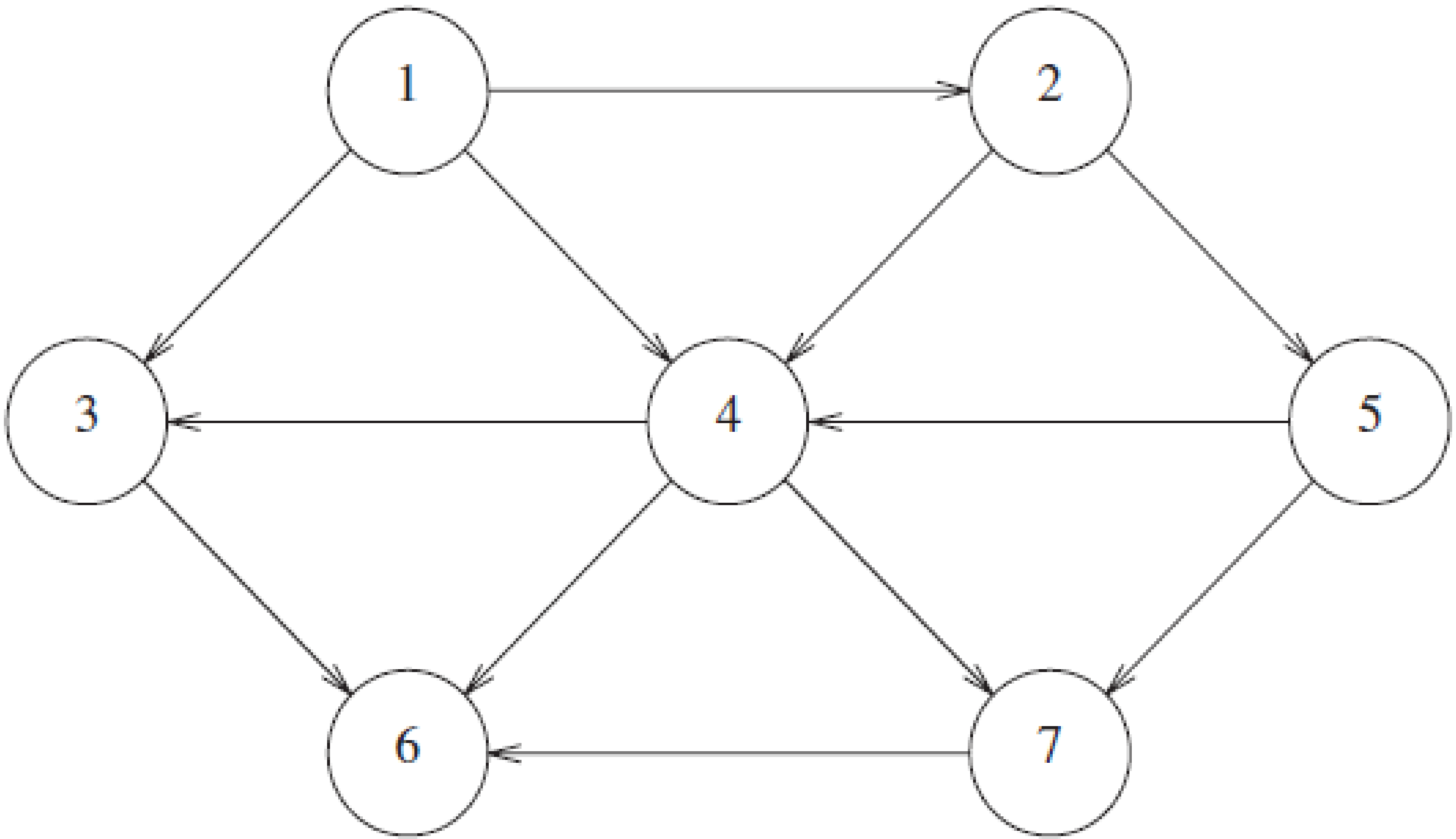
adjacency lists



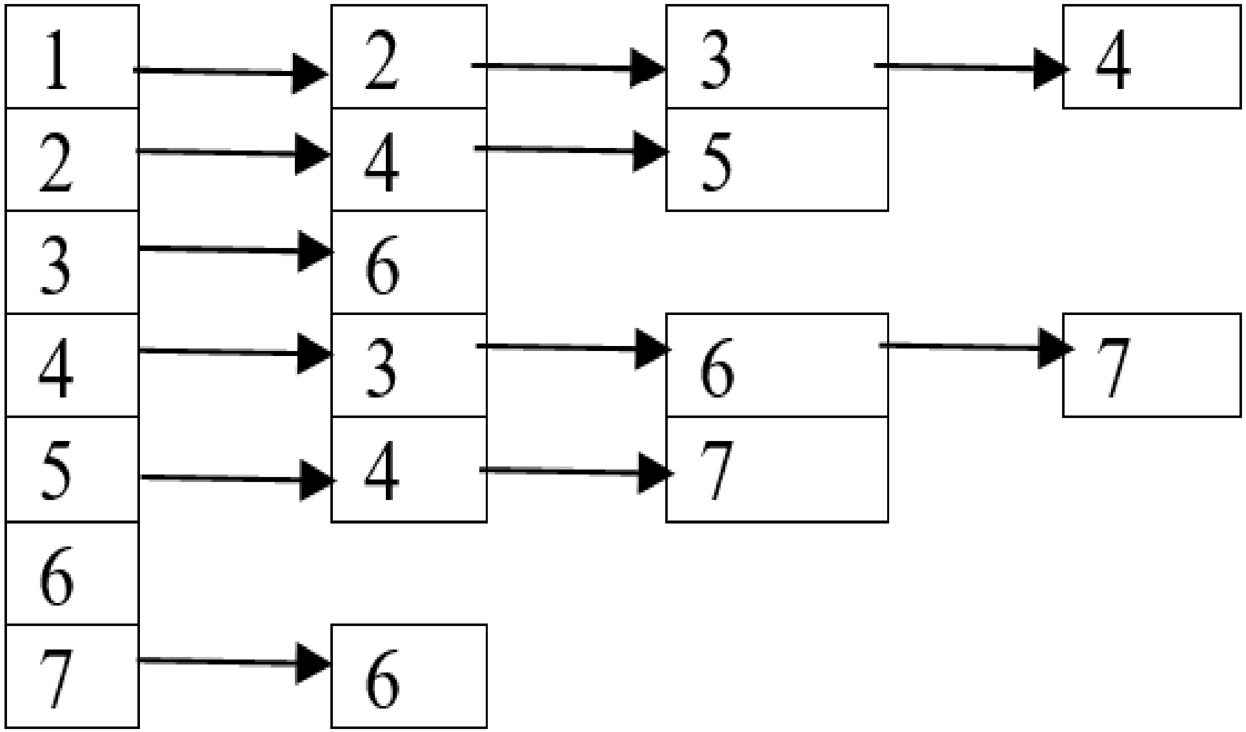
adjacency lists



Represent the following graphs in adjacency list, adjacency matrix :



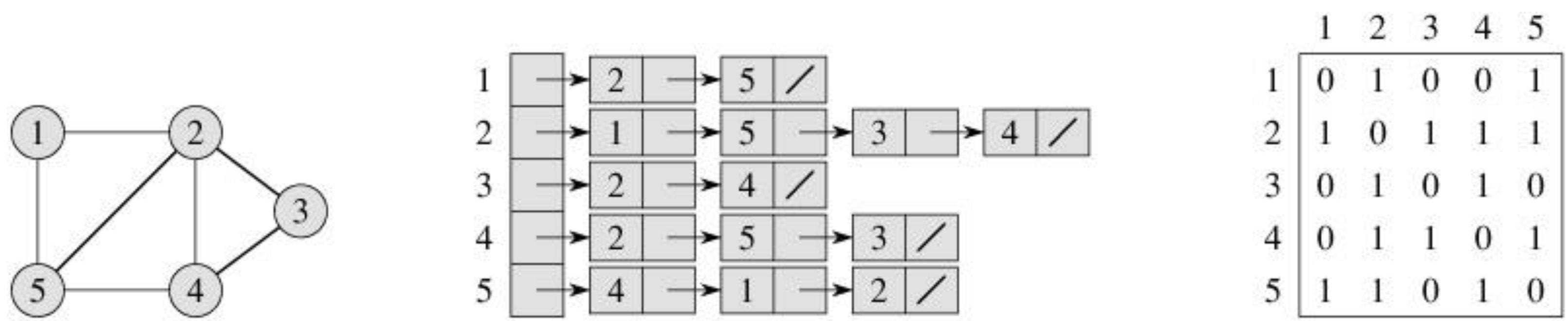
Adjacency list:



Adjacency Matrix :

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

Graph representation – undirected

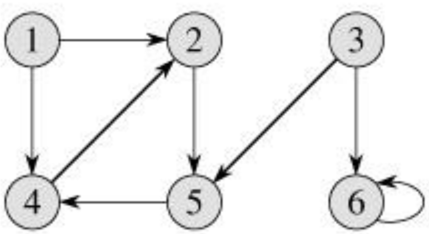


(a) (b) (c)

graph Adjacency list Adjacency matrix

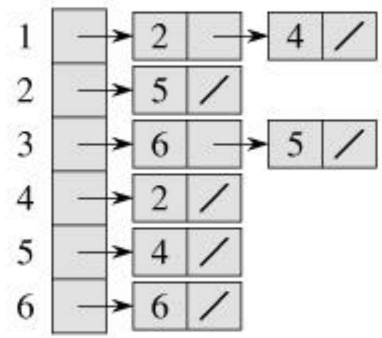
ref. Introduction to Algorithms by Thomas Cormen

Graph representation – directed



(a)

graph



(b)

Adjacency list

	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)

Adjacency matrix

ref. Introduction to Algorithms by Thomas Cormen

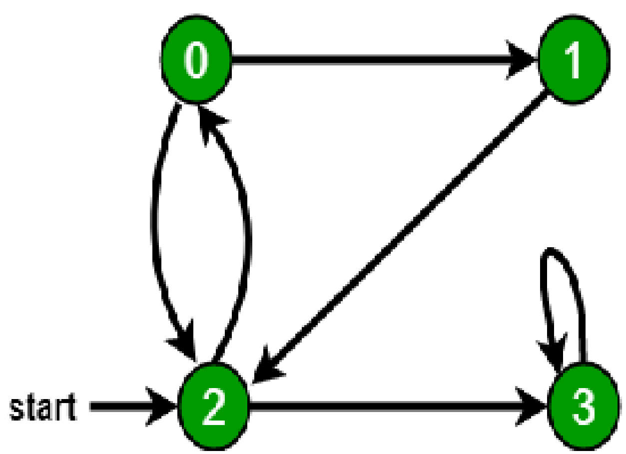
Graph Traversal

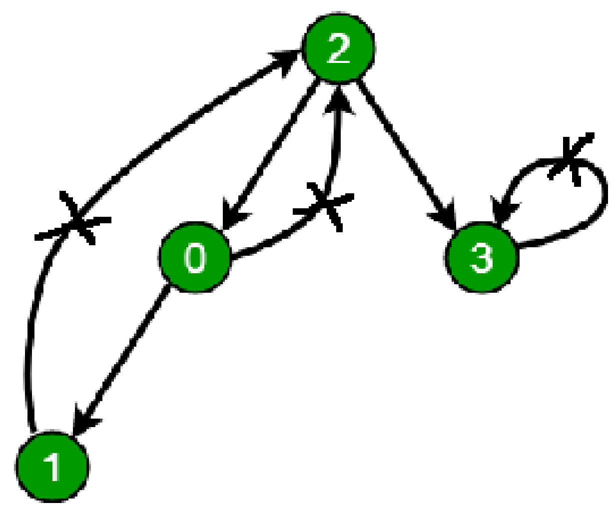
Graph Traversal

- BFS (Breadth First Search)
  - Start from a vertex, visit all the reachable vertices in a breadth first manner
  - Uses Queue for non-recursive implementation
- DFS (Depth First Search)
  - Start from a vertex, visit all the reachable vertices in a depth first manner
  - Uses Stack for non-recursive implementation

Graph Traversal-DFS

- Depth First Traversal (or Search) for a graph is similar to **Depth First Traversal** of a **tree**.
- The only difference here is, unlike trees, graphs may contain **cycles**, so we may come to the same node again.
- To avoid processing a node more than once, we use a **Boolean visited array**.





Graph Traversal-DFS

- For example, in the following graph, we start traversal from vertex 2.
- When we come to vertex 0, we look for all adjacent vertices of it.
- 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process.
- A Depth First Traversal of the following graph is 2, 0, 1, 3.

Depth-First Search

dfs (Node v)

1.[Push v on the stack STK]

```
push(STAK,v);
```

2.Repeat steps 3 to 5 while STAK is not empty

3.[Pop top element from STAK]

```
u=pop(STAK);
```

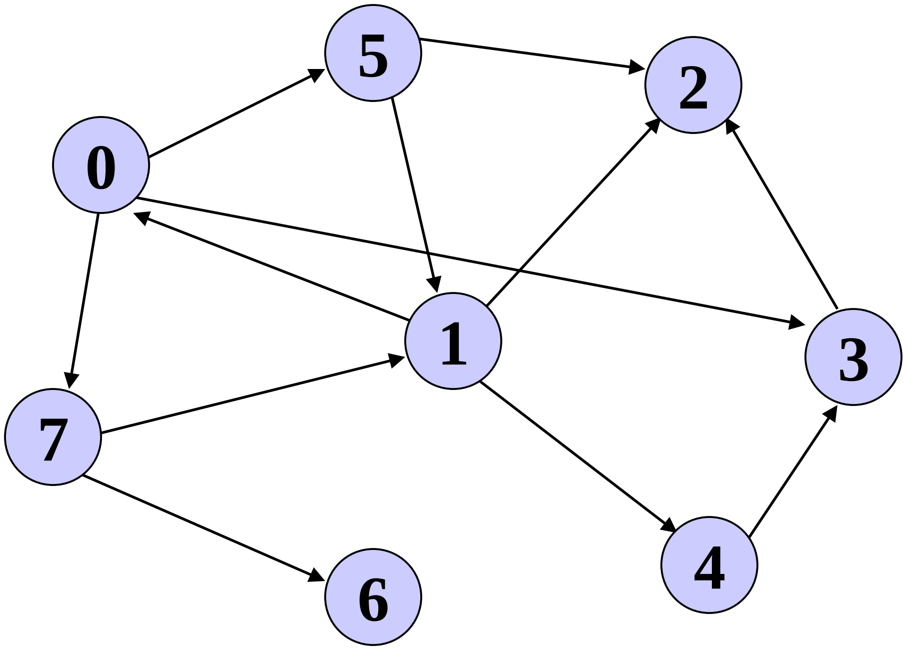
4.If u is not in visited list

Add u to the list of visited nodes

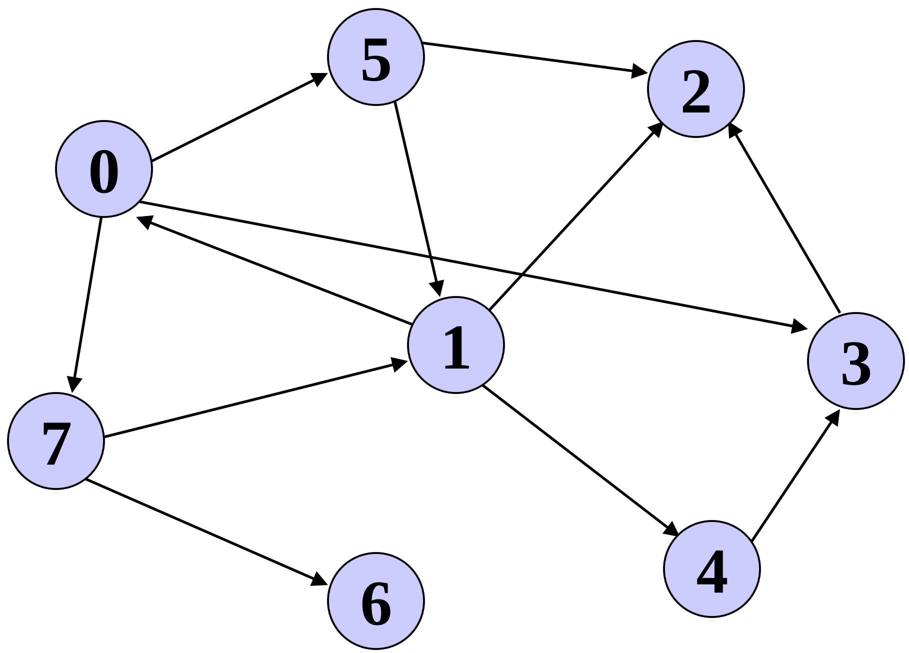
5.For each w adjacent to u

If w is not visited then Push(STAK,w);

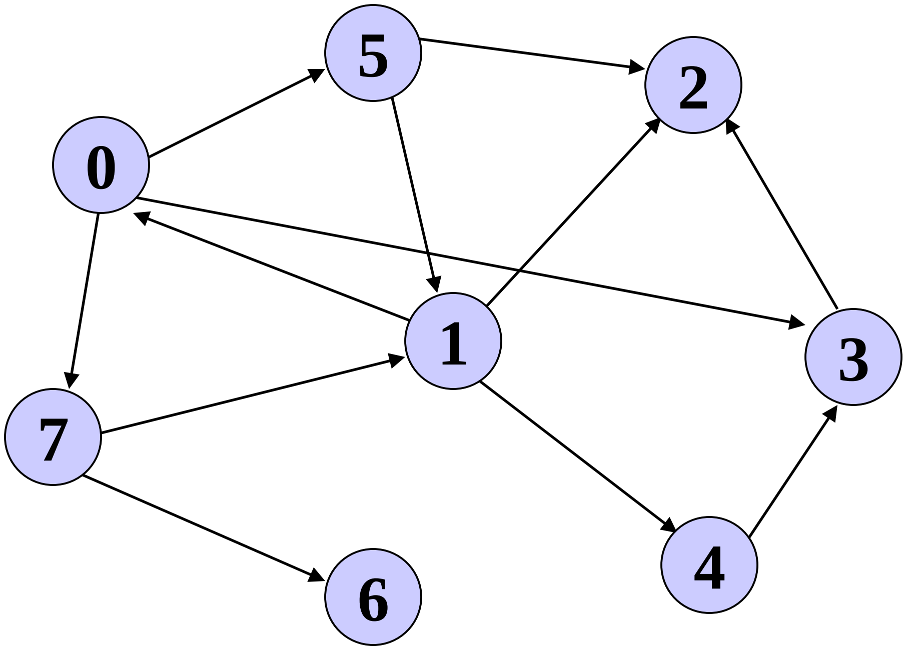
Example

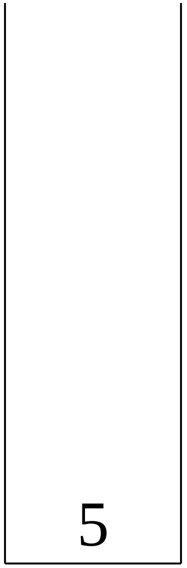


DFS: Start with Node 5



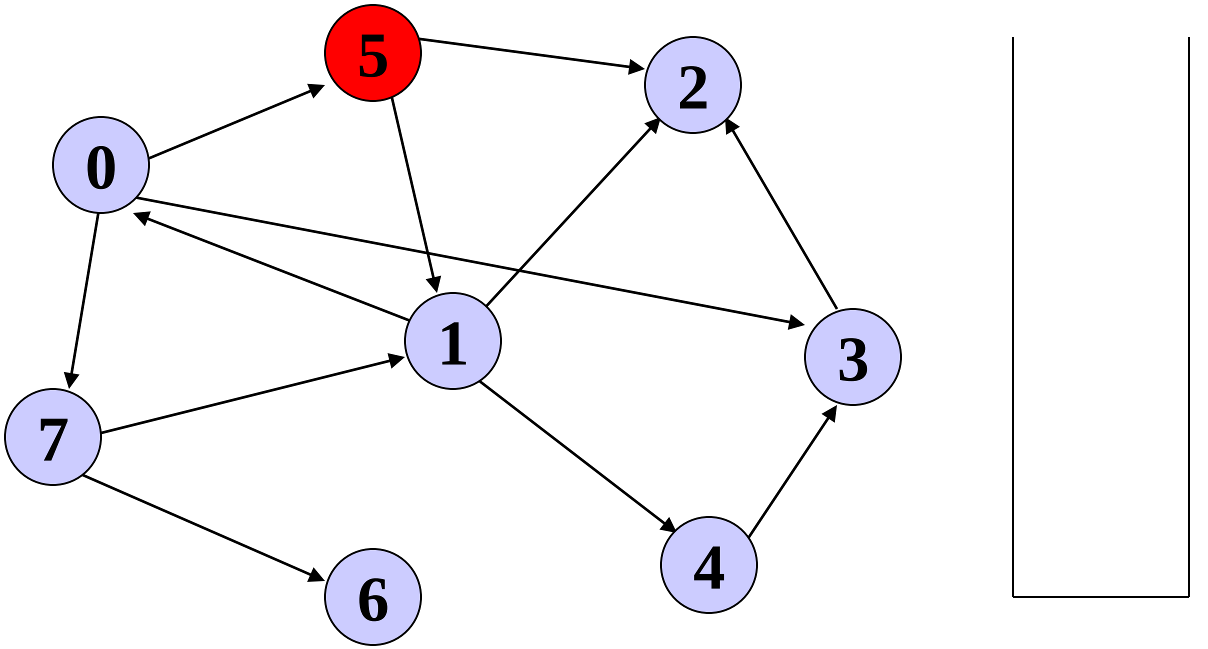
DFS: Start with Node 5





Push 5

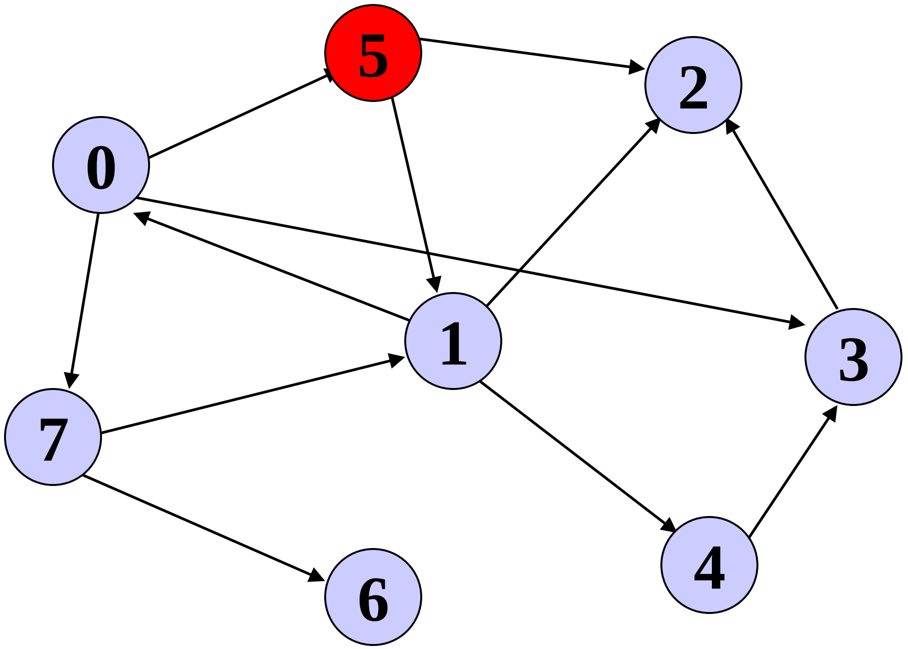
DFS: Start with Node 5

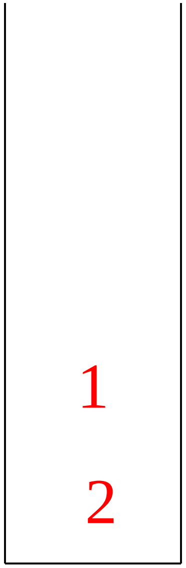


Pop/Visit 5

Visited: 5

DFS: Start with Node 5

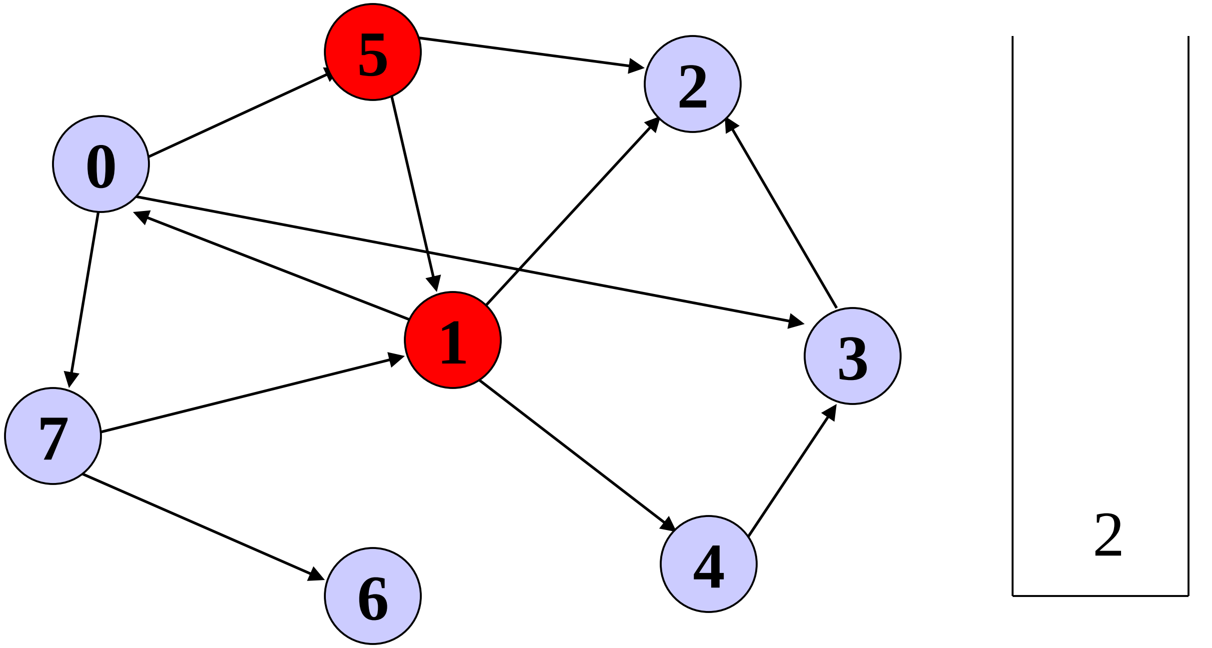




Push 2, Push 1

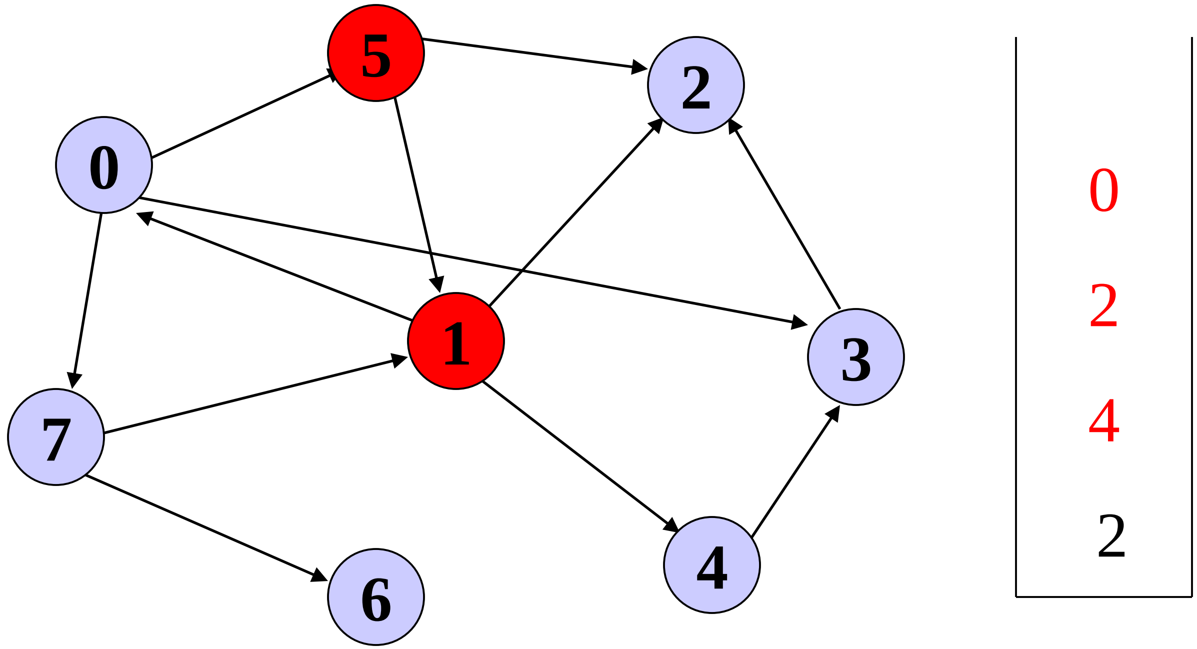
Visited: 5

DFS: Start with Node 5



Visited: 5 1

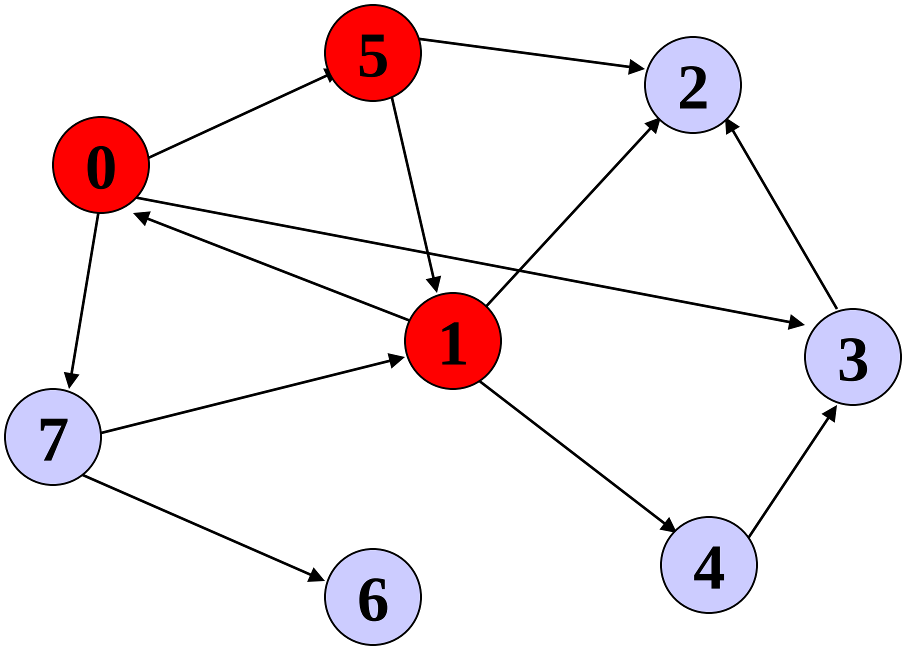
DFS: Start with Node 5

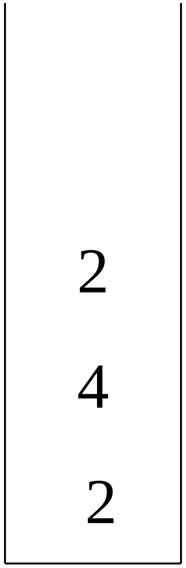


Push 4, Push 2, Push 0

Visited: 5 1

DFS: Start with Node 5

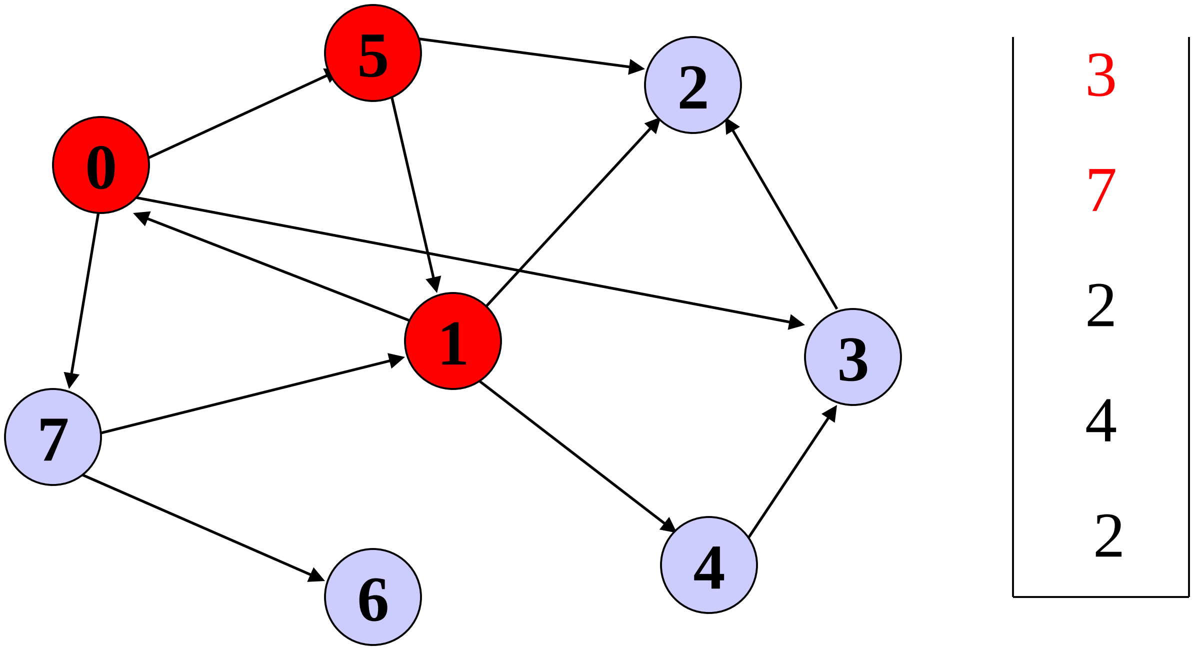




Pop/Visit 0

Visited: 5 1 0

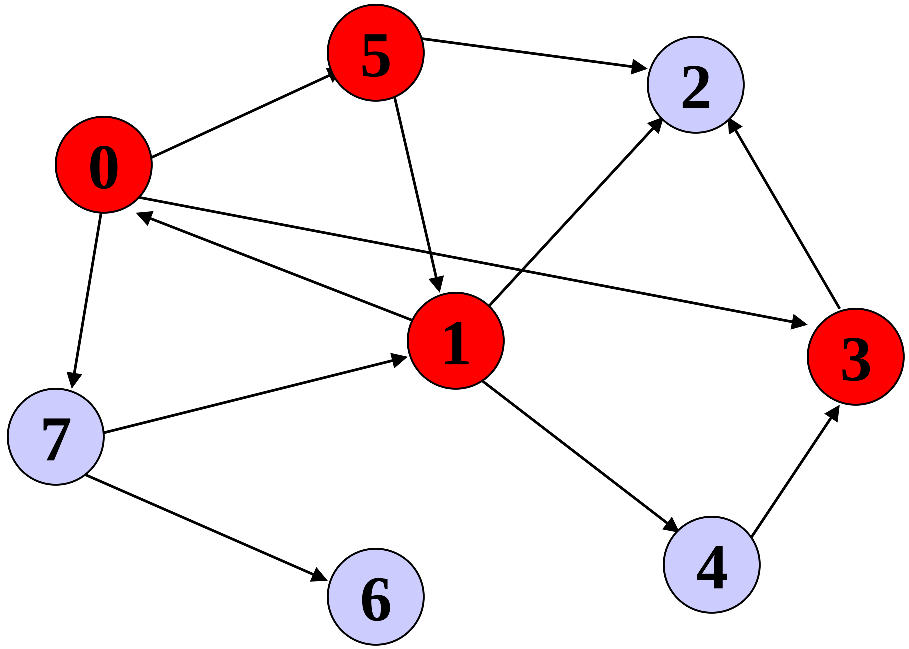
DFS: Start with Node 5

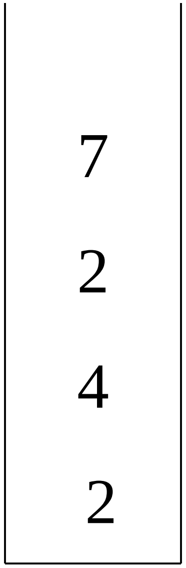


Push 7, Push 3

Visited: 5 1 0

DFS: Start with Node 5

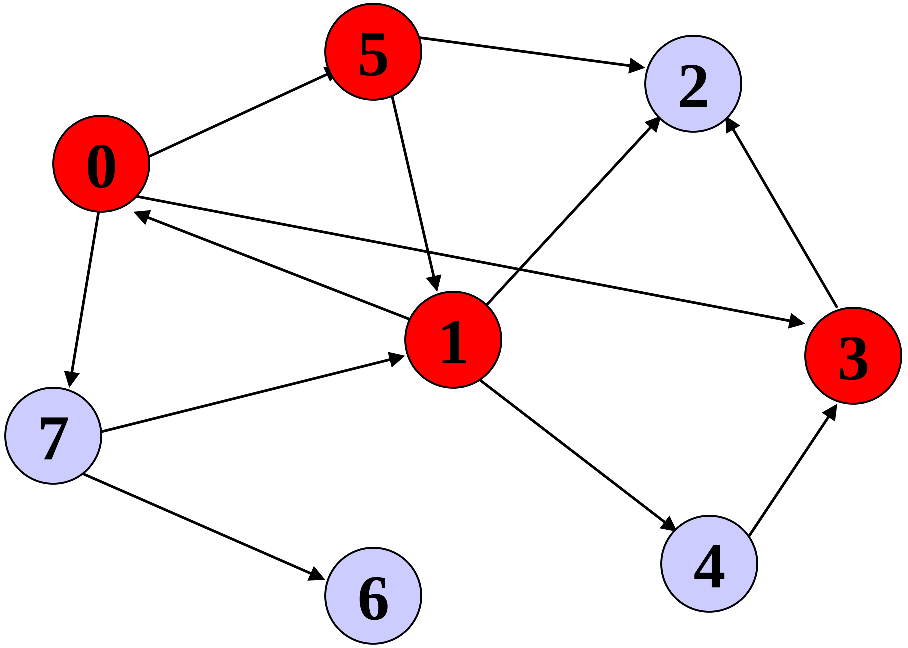


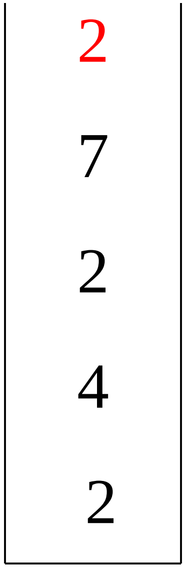


Pop/Visit 3

Visited: 5 1 0 3

DFS: Start with Node 5

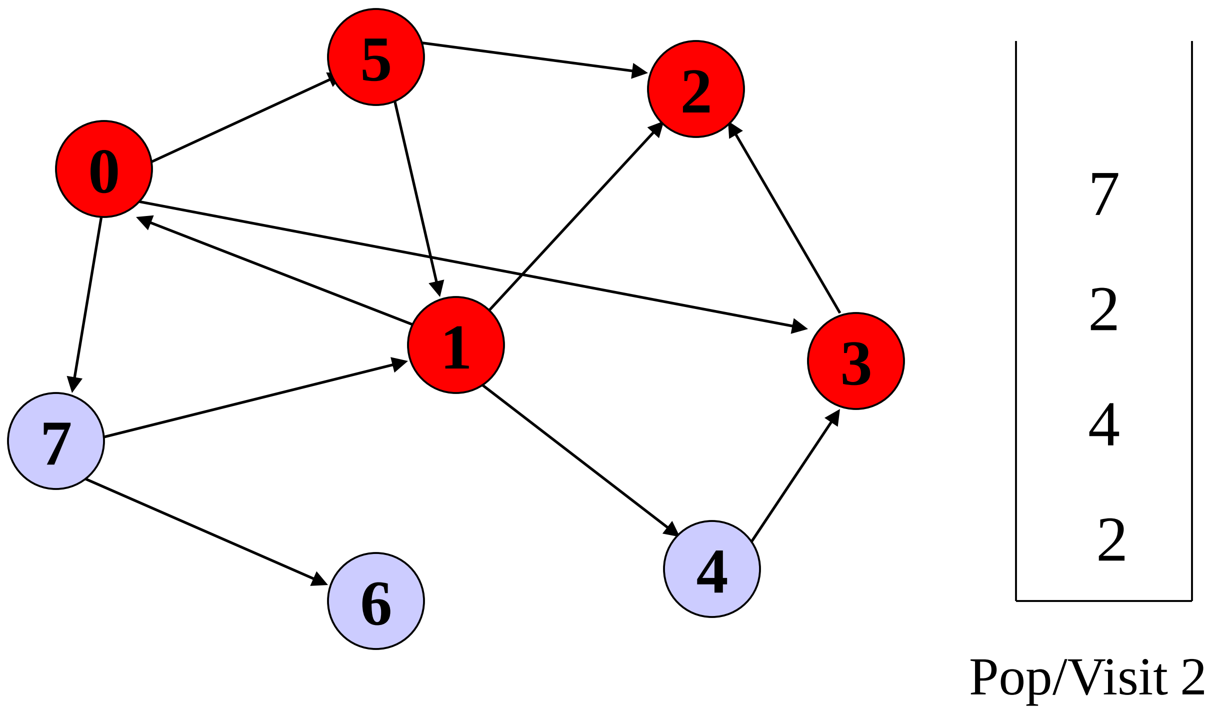




Push 2

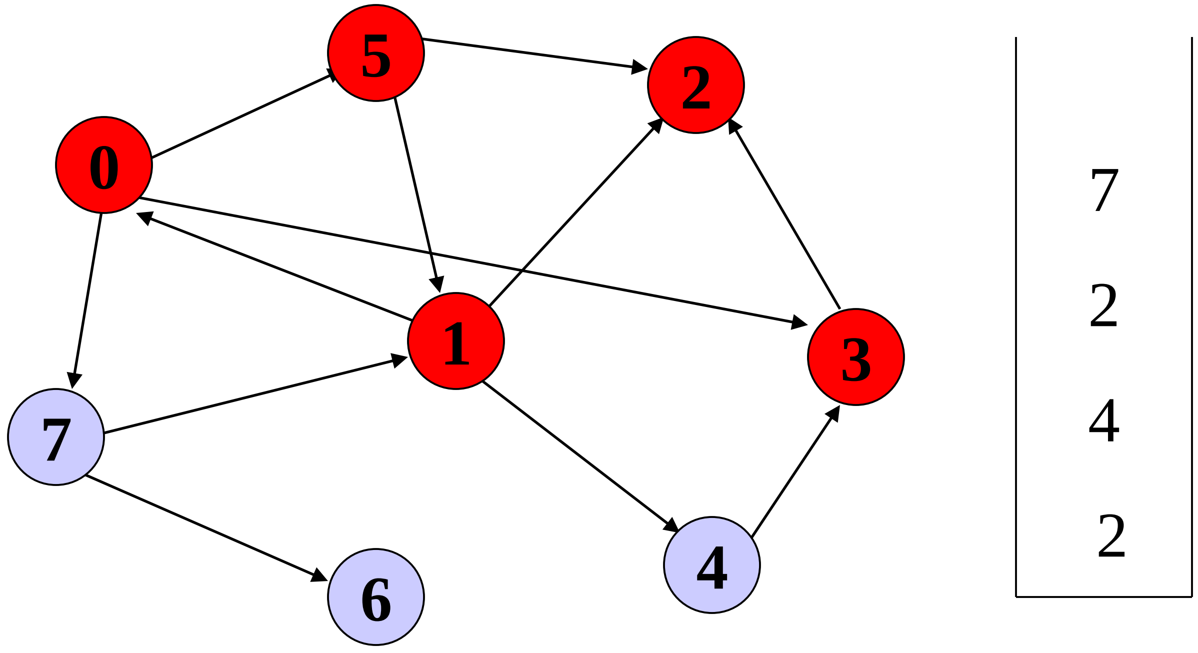
Visited: 5 1 0 3

DFS: Start with Node 5



Visited: 5 1 0 3 2

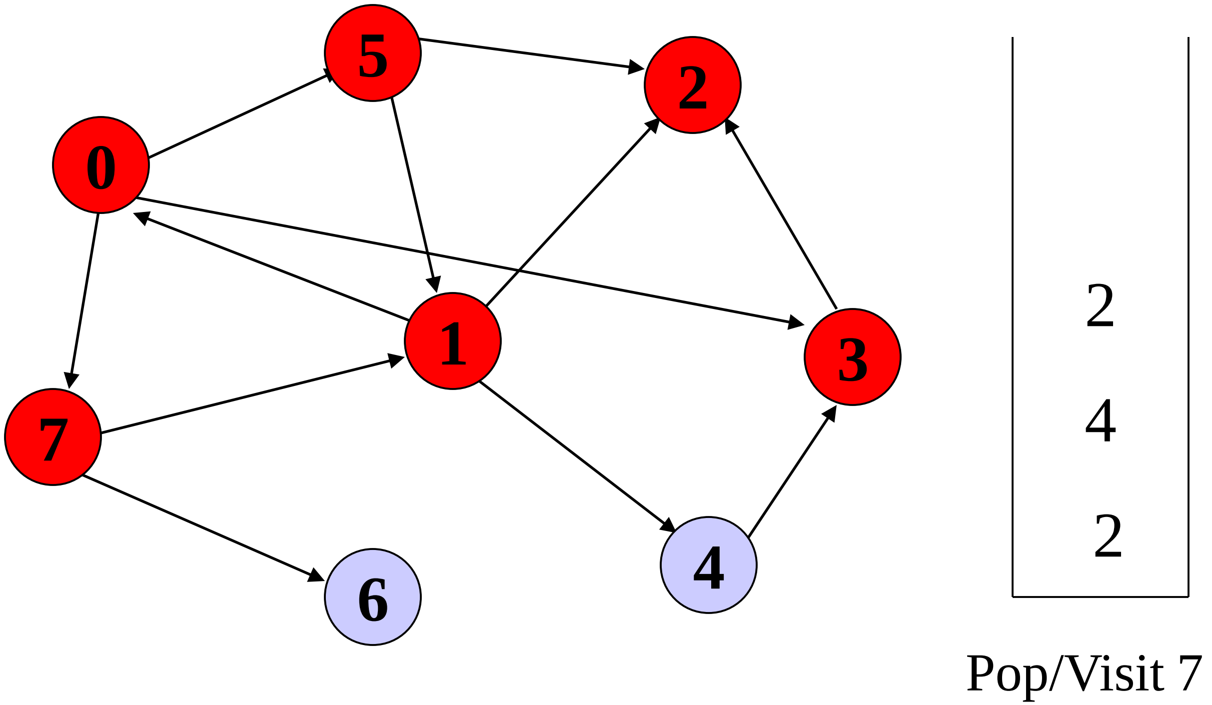
DFS: Start with Node 5



No push operation

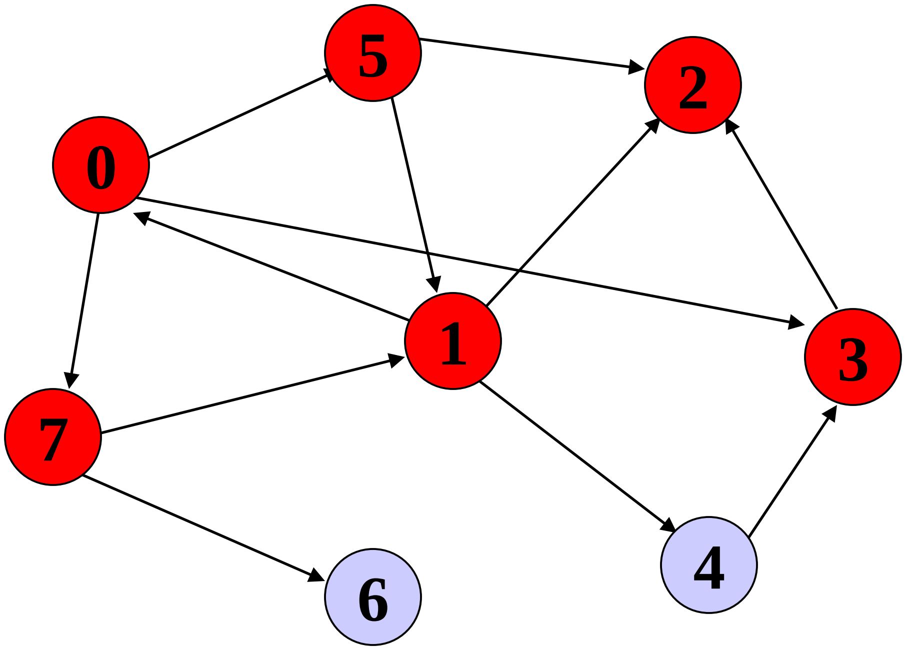
Visited: 5 1 0 3 2

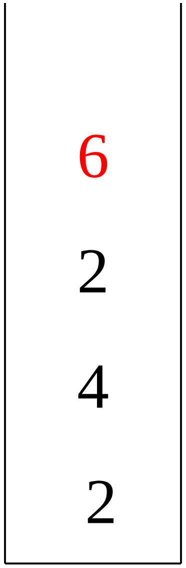
DFS: Start with Node 5



Visited: 5 1 0 3 2 7

DFS: Start with Node 5

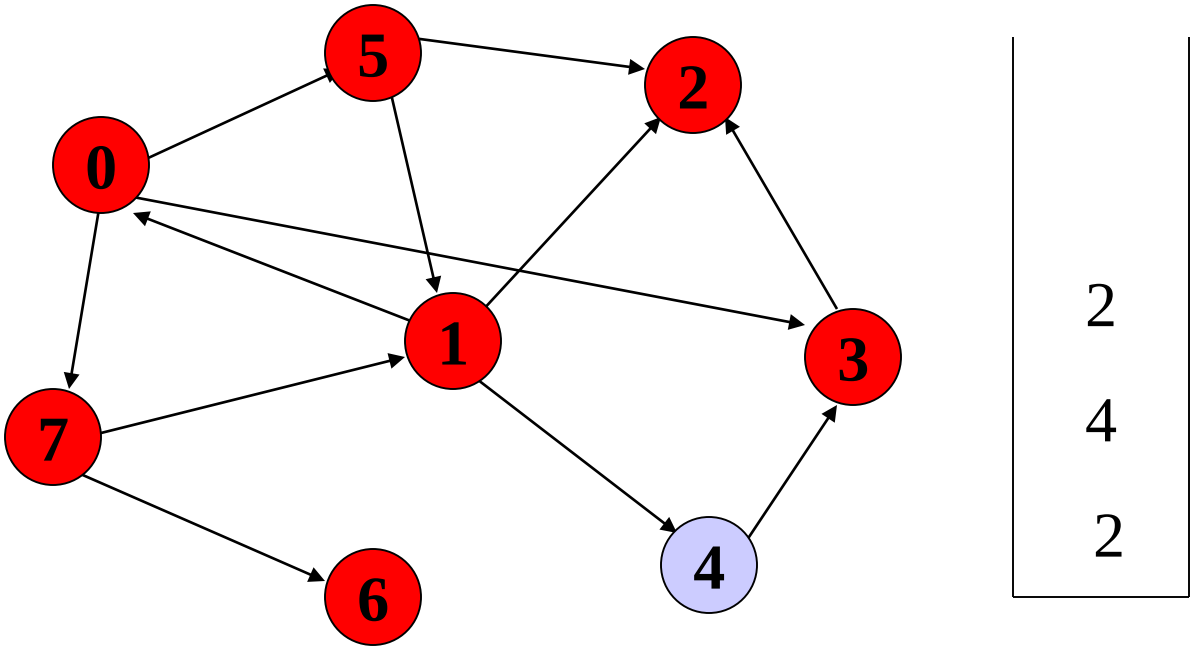




Push 6

Visited: 5 1 0 3 2 7

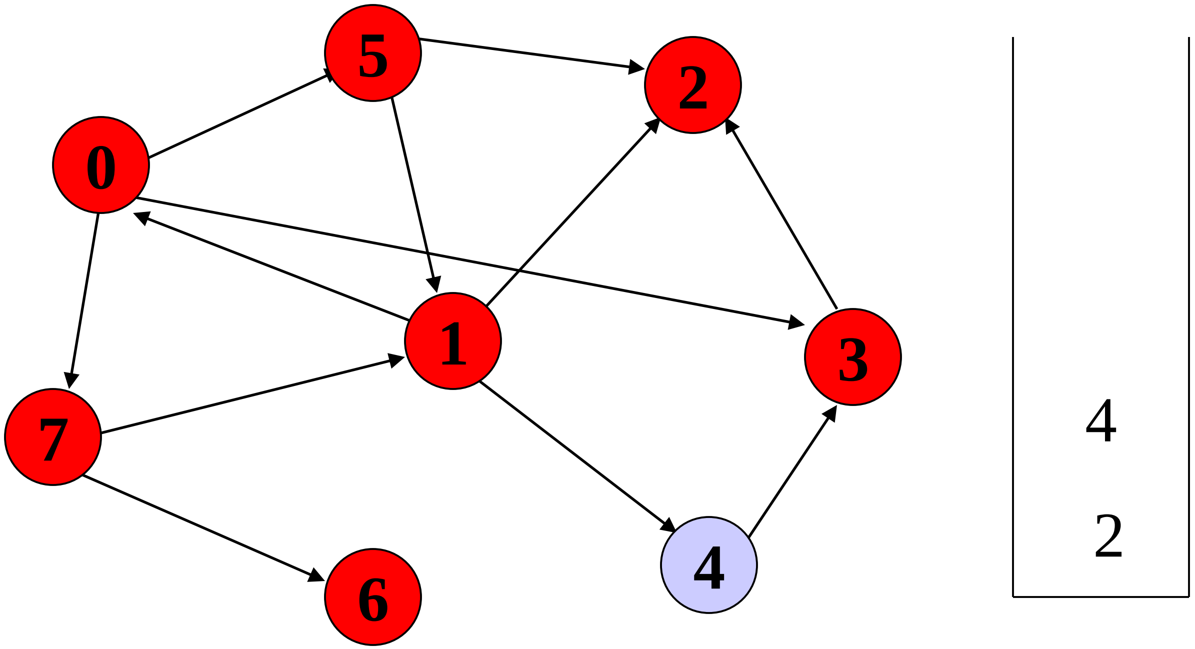
DFS: Start with Node 5



Pop/Visit 6

Visited: 5 1 0 3 2 7 6

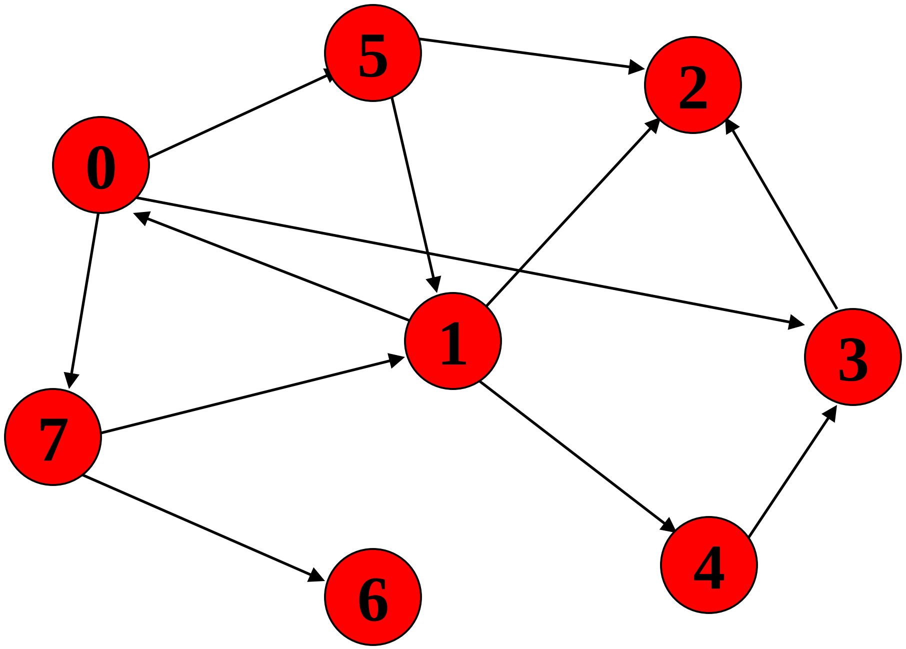
DFS: Start with Node 5

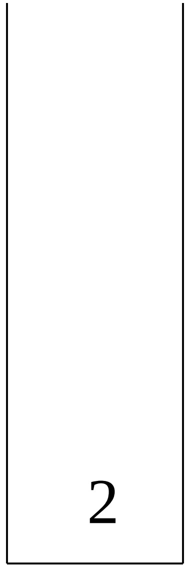


Pop (don’t visit) 2

Visited: 5 1 0 3 2 7 6

DFS: Start with Node 5

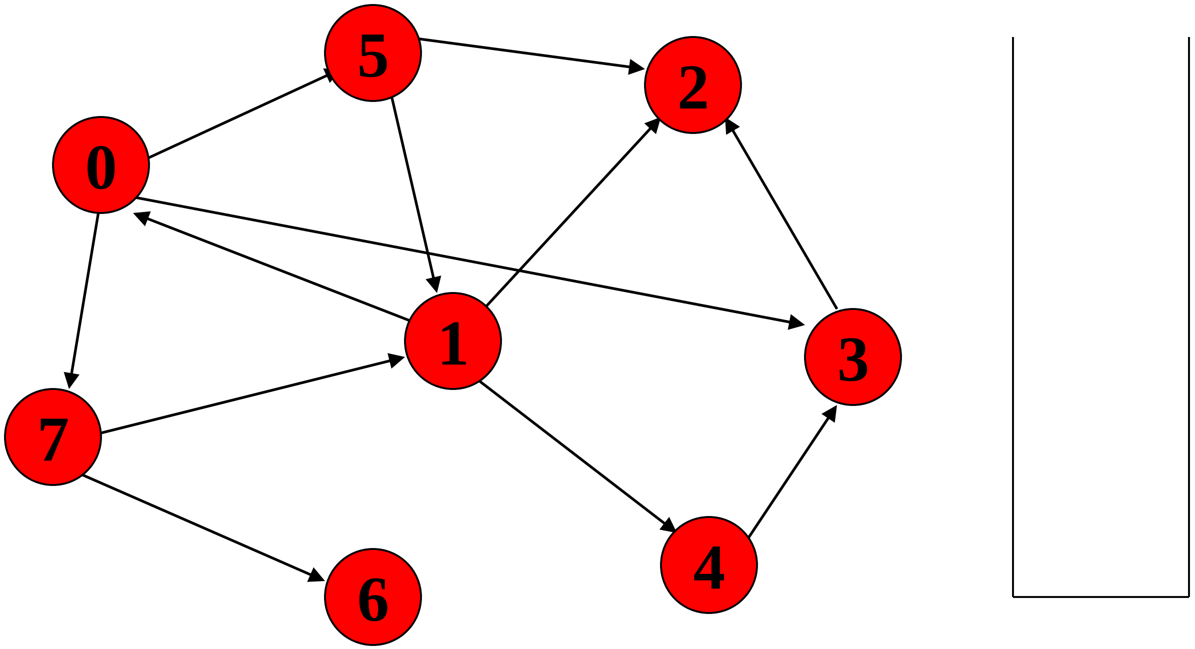




Pop/Visit 4

Visited: 5 1 0 3 2 7 6 4

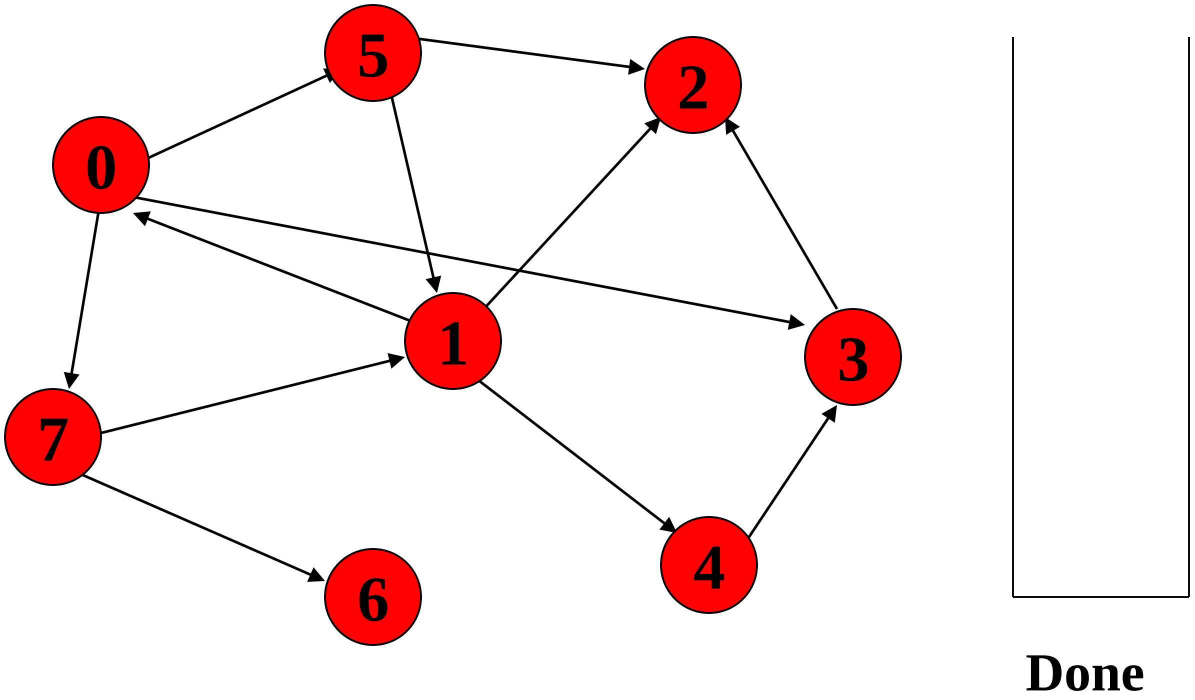
DFS: Start with Node 5



Pop (don’t visit) 2

Visited: 5 1 0 3 2 7 6 4

DFS: Start with Node 5



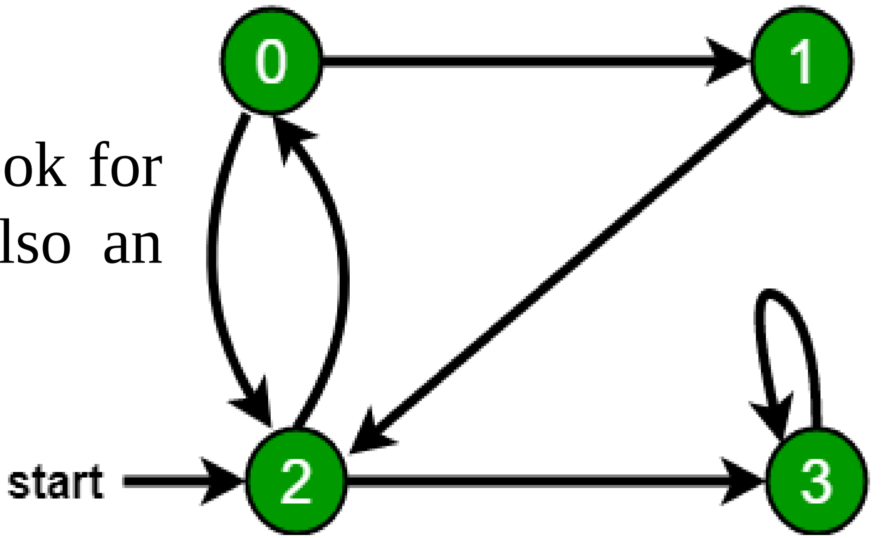
Visited: 5 1 0 3 2 7 6 4

Graph Traversal-BFS

- Breadth First Traversal (or Search) for a graph is similar to Level order Traversal of a tree.
- The only difference here is, unlike trees, graphs may contain cycles, so we may come to the same node again.
- To avoid processing a node more than once, we use a **Boolean visited array**.
- For simplicity, it is assumed that all vertices are reachable from the starting vertex.

Graph Traversal-BFS

- For example, in the given graph, we start traversal from vertex 2.
- When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0.

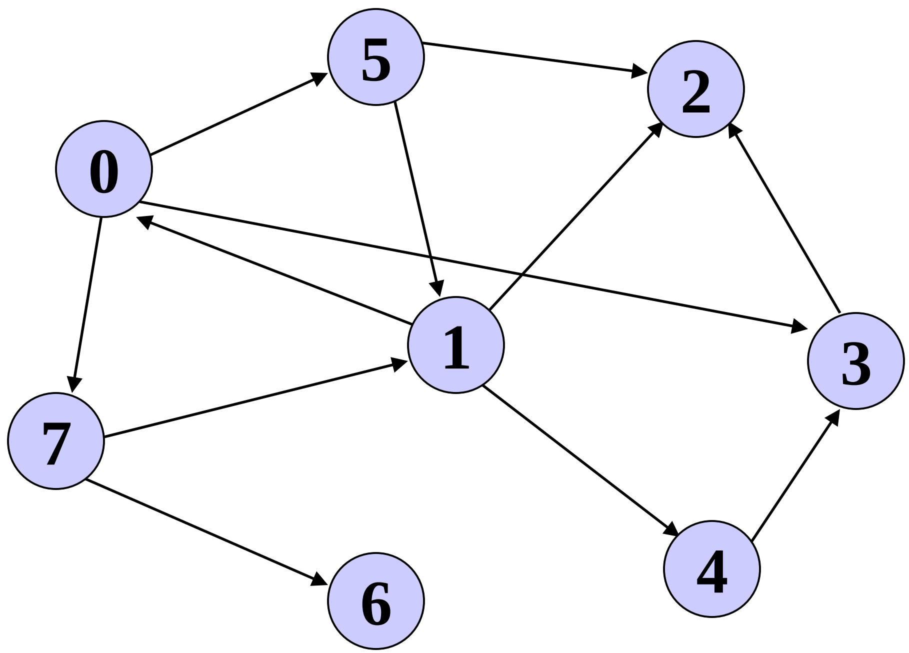


- If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process.
- A BFS traversal of the following graph is 2, 0, 3, 1.

Breadth-first Search

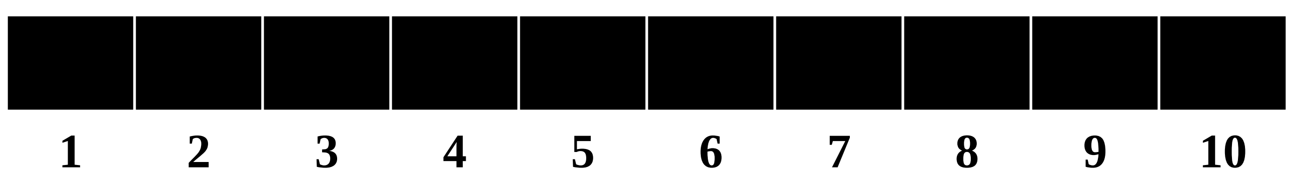
```
bfs (Node v)
1.[add v to QUEUE]
     enqueue(QUEUE,v);
2.Repeat steps 3 to 5 while QUEUE is not
  empty
3.[Remove one element from QUEUE]
     u=dequeue(QUEUE);
4.If u is not in list of visited nodes then
     Add u to list of visited nodes
5.For each w adjacent to u
    If w is not visited then
     enqueue(QUEUE,w);
```

BFS: Start with Node 5



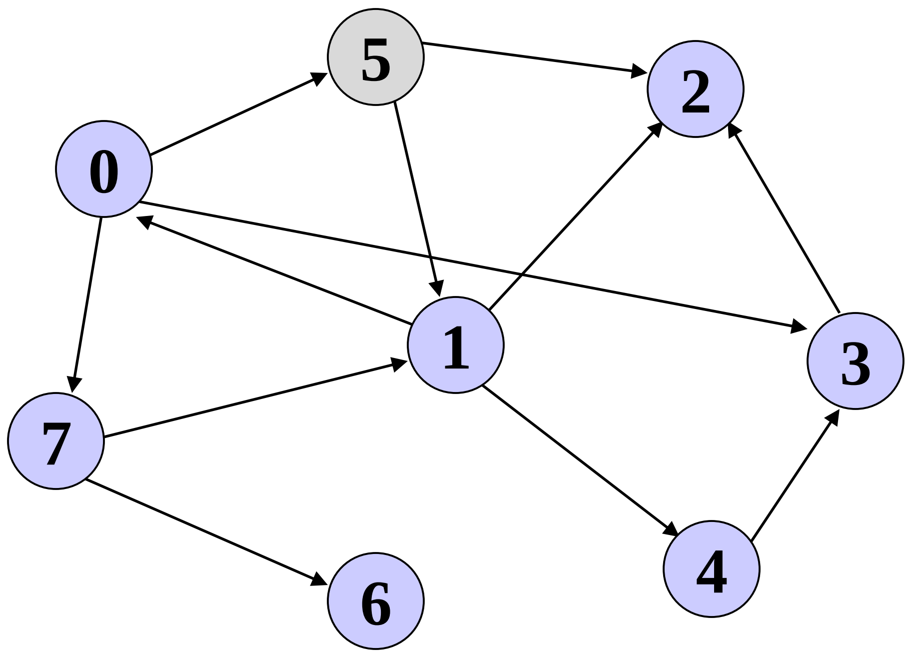
F=0

R=0



BFS: Start with Node 5

Visited:



F=1

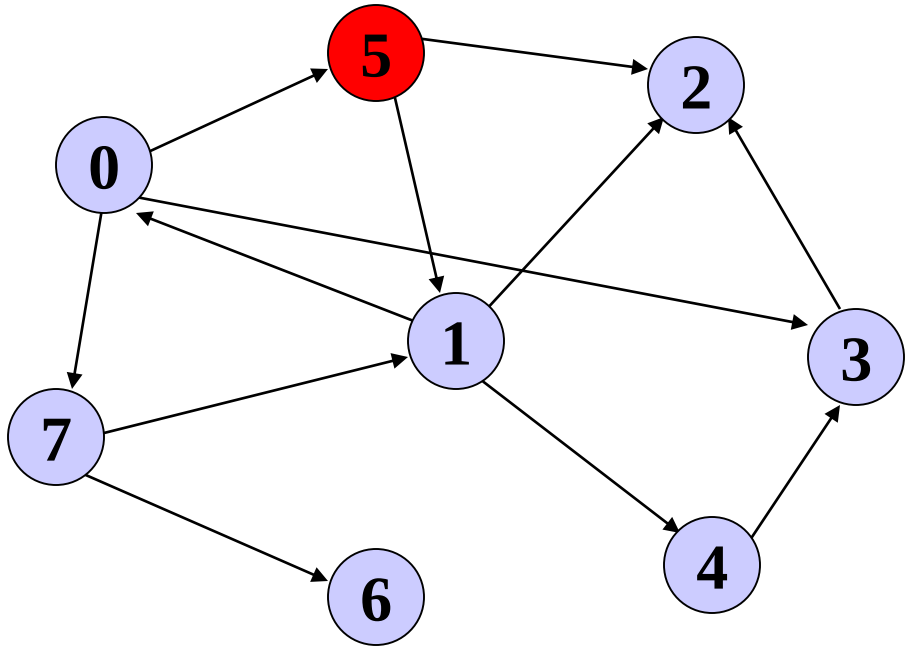
R=1

5									
1	2	3	4	5	6	7	8	9	10

Enqueue 5

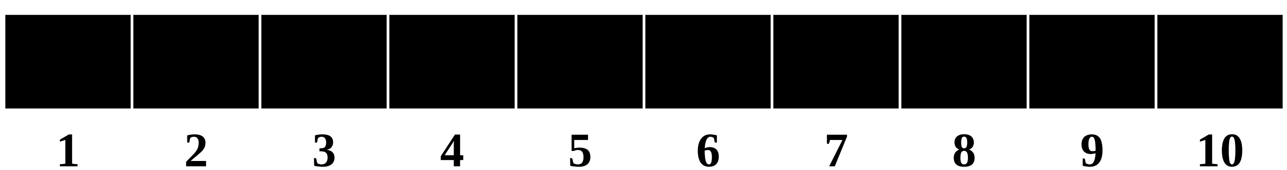
BFS: Start with Node 5

Visited: 5



F=0

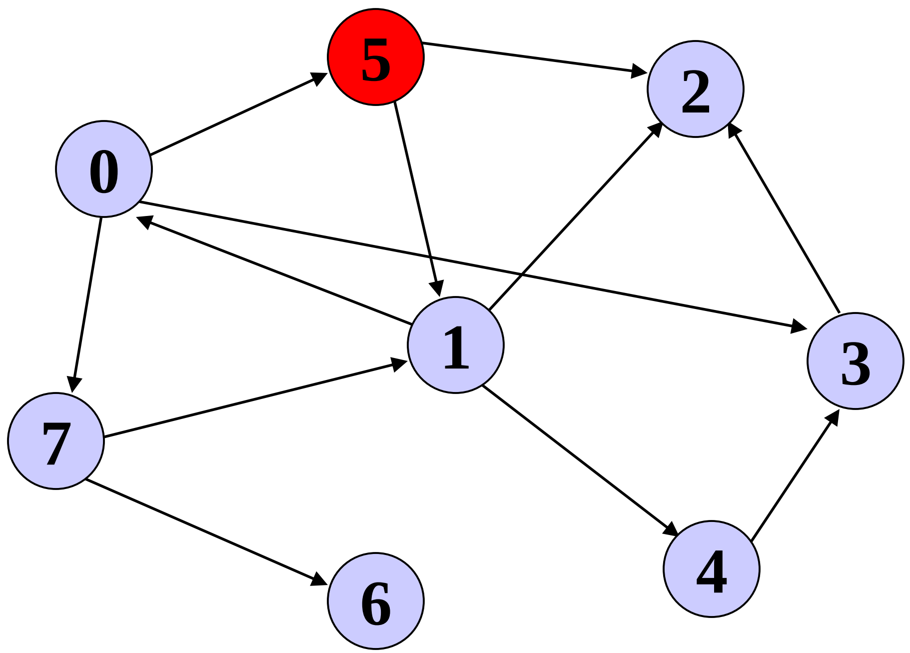
R=0



Dequeue/Visit 5

BFS: Start with Node 5

Visited: 5



F=1

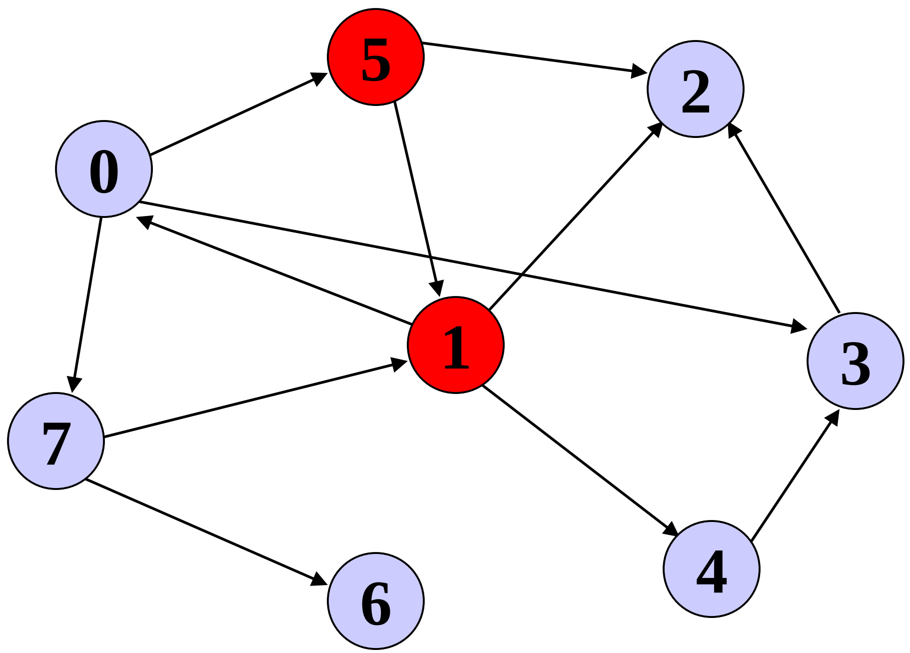
R=2

1	2	3	4	5	6	7	8	9	10
1	2								

Enqueue 1, 2

BFS: Start with Node 5

Visited: 5 1



F=2

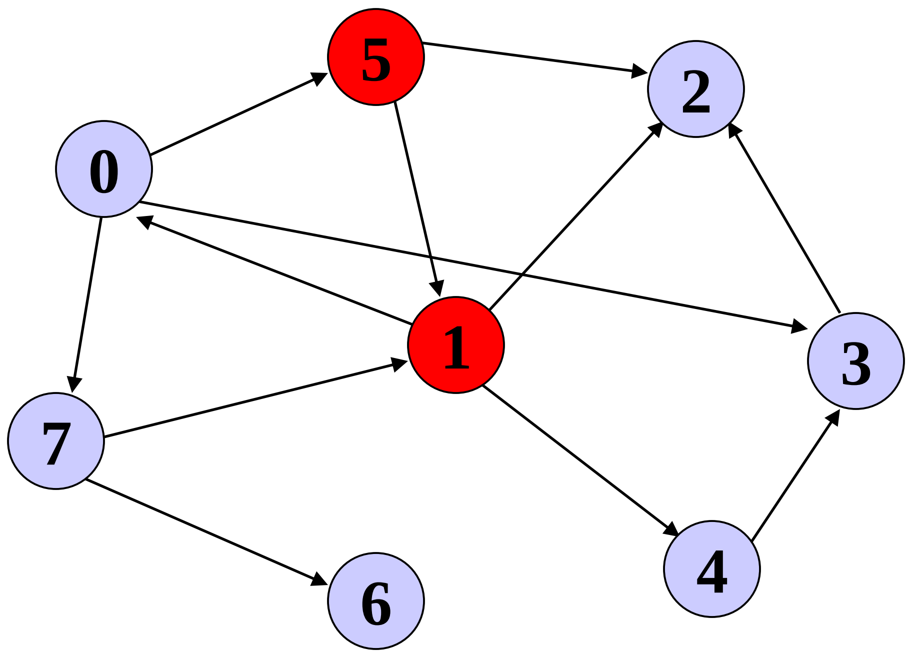
R=2

1	2	3	4	5	6	7	8	9	10
	2								

Dequeue/Visit 1

BFS: Start with Node 5

Visited: 5 1



F=2

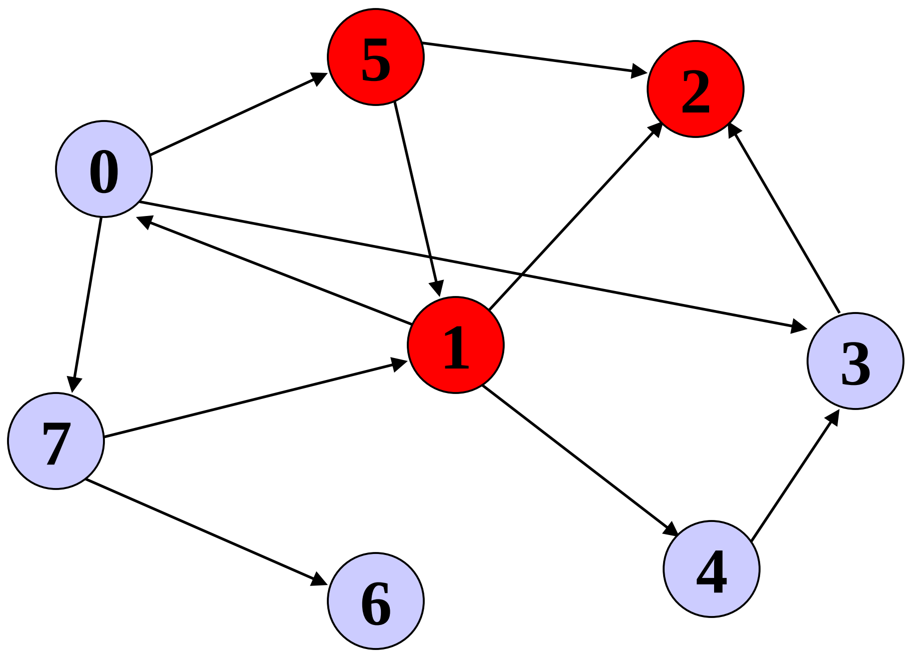
R=5

1	2	3	4	5	6	7	8	9	10
	2	0	2	4					

Enqueue 0, 2, 4

BFS: Start with Node 5

Visisted: 5 1 2



F=3

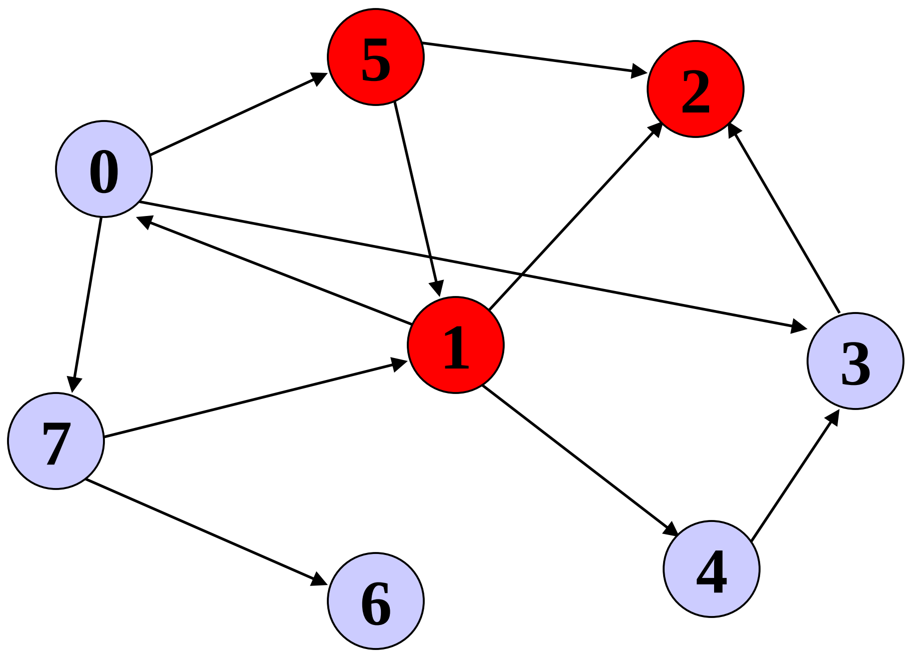
R=5

1	2	3	4	5	6	7	8	9	10
		0	2	4					

Dequeue/Visit 2

BFS: Start with Node 5

Visited: 5 1 2



F=3

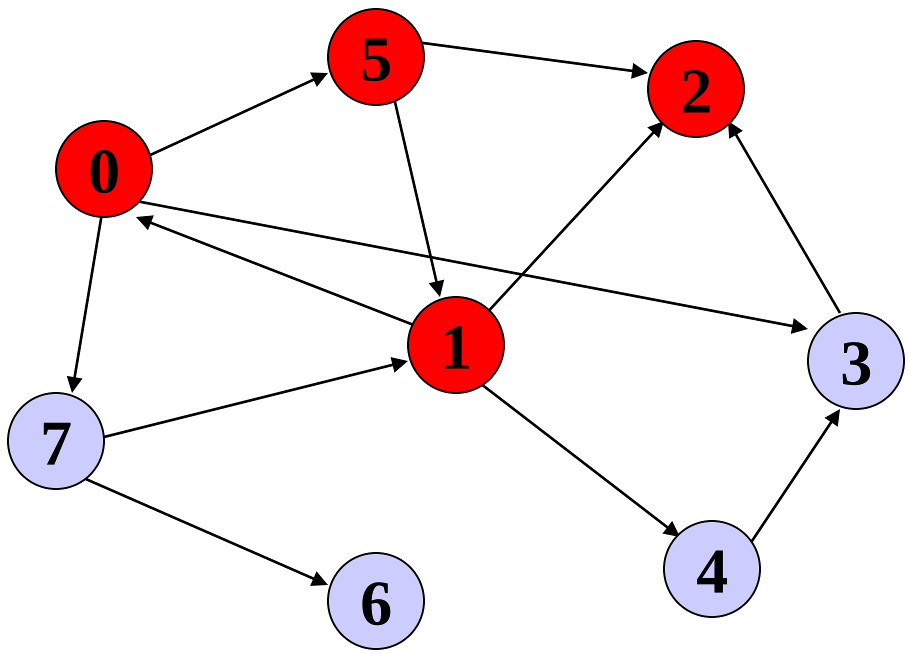
R=5

1	2	3	4	5	6	7	8	9	10
		0	2	4					

No Enqueue operation

BFS: Start with Node 5

Visited: 5 1 2 0



F=4

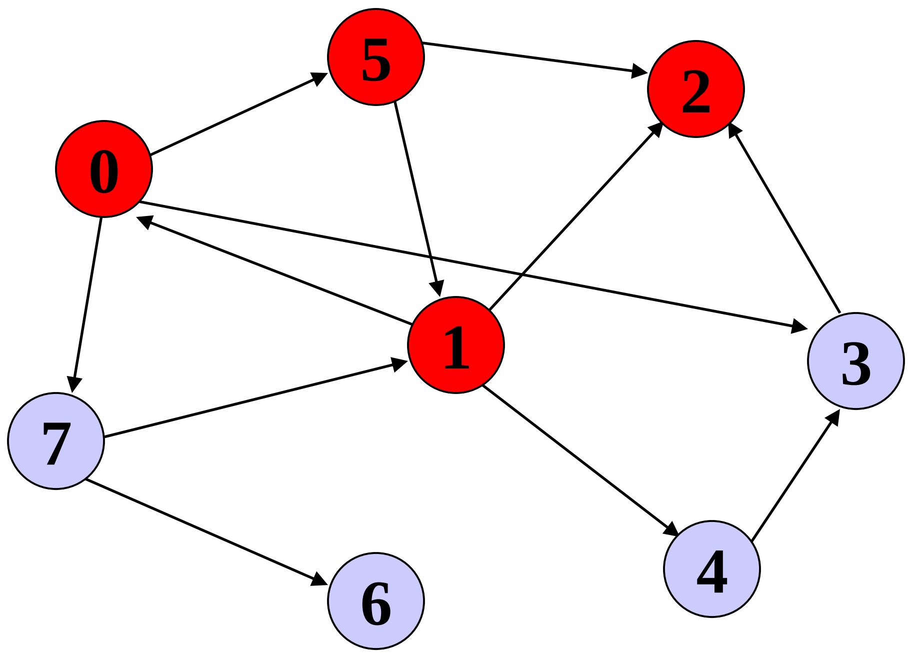
R=5

			2	4					
1	2	3	4	5	6	7	8	9	10

Dequeue/Visit 0

BFS: Start with Node 5

Visited: 5 1 2 0



F=4

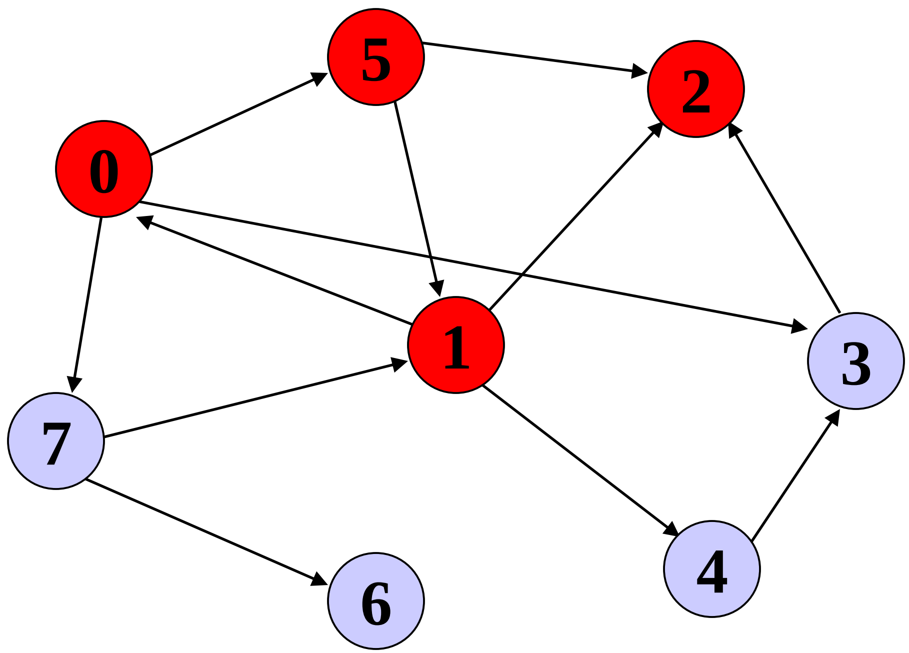
R=7

			2	4	3	7			
1	2	3	4	5	6	7	8	9	10

Enqueue 3, 7

BFS: Start with Node 5

Visited: 5 1 2 0



F=5

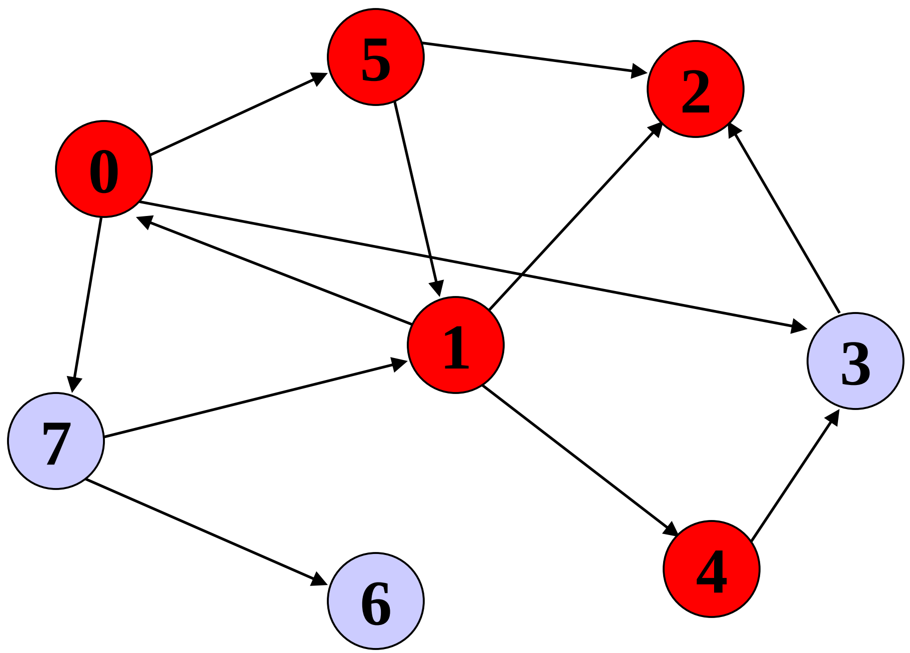
R=7

1	2	3	4	5	6	7	8	9	10
				4	3	7			

Dequeue (don't visit) 2

BFS: Start with Node 5

Visisted: 5 1 2 0 4



F=6

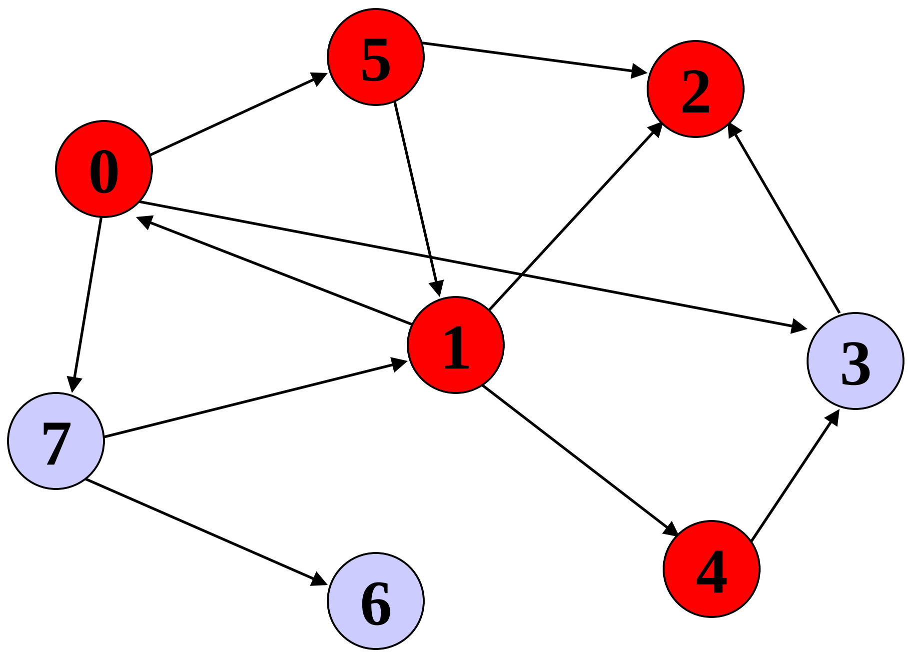
R=7

1	2	3	4	5	6	7	8	9	10
					3	7			

Dequeue/Visit 4

BFS: Start with Node 5

Visisted: 5 1 2 0 4



F=6

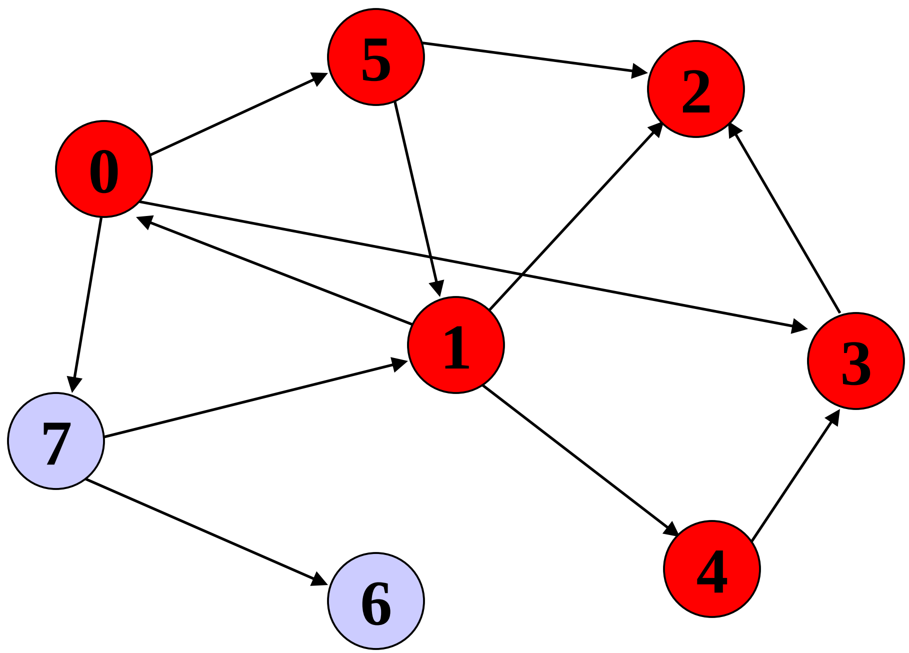
R=8

1	2	3	4	5	6	7	8	9	10
					3	7	3		

Enqueue 3

BFS: Start with Node 5

Visisted: 5 1 2 0 4 3



F=7

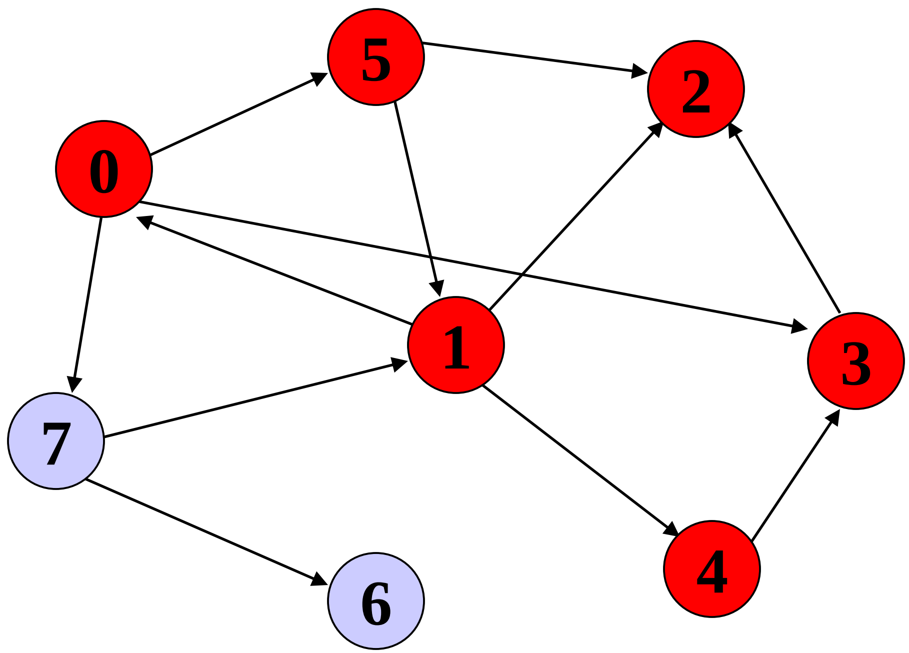
R=8

1	2	3	4	5	6	7	8	9	10
						7	3		

Dequeue/Visit 3

BFS: Start with Node 5

Visisted: 5 1 2 0 4 3



F=7

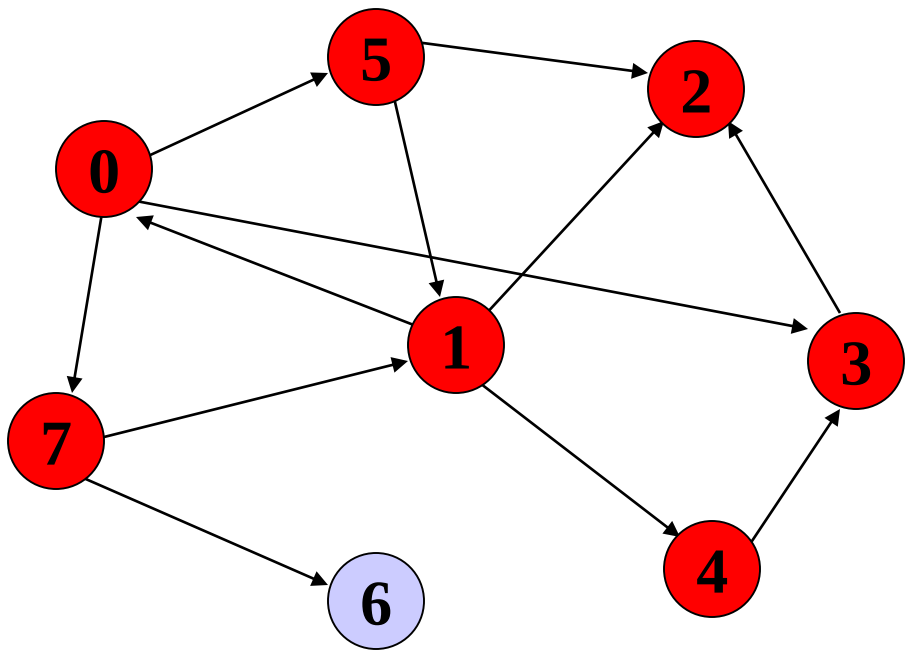
R=8

						7	3		
1	2	3	4	5	6	7	8	9	10

No Enqueue operation

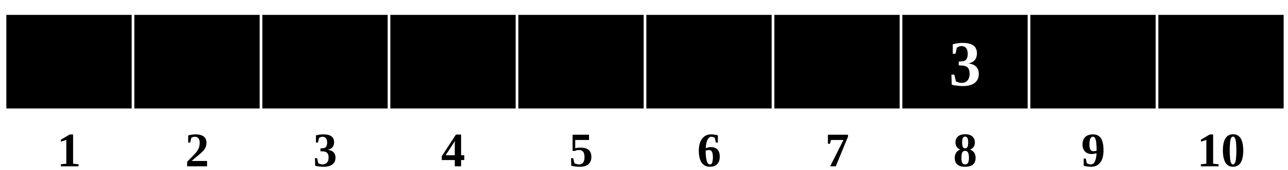
BFS: Start with Node 5

Visited: 5 1 2 0 4 3 7



F=8

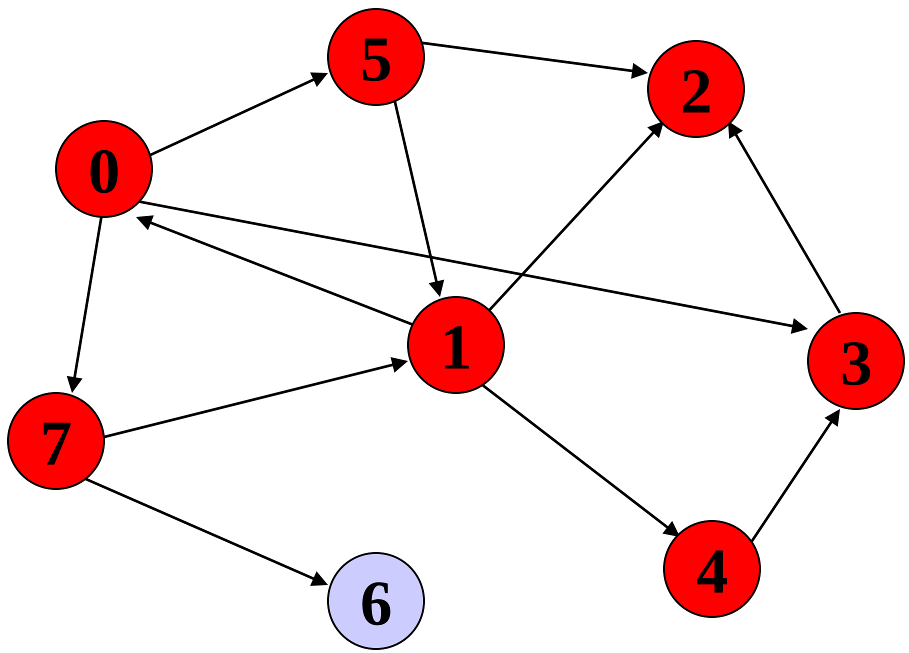
R=8



Dequeue/Visit 7

BFS: Start with Node 5

Visited: 5 1 2 0 4 3 7



F=8

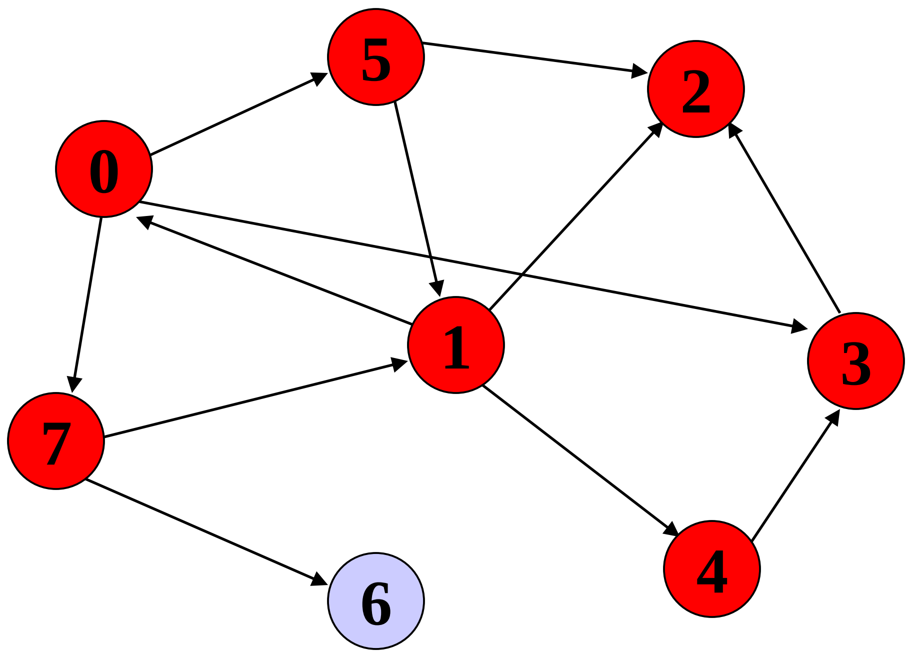
R=9

1	2	3	4	5	6	7	8	9	10
							3	6	

Enqueue 6

BFS: Start with Node 5

Visited: 5 1 2 0 4 3 7



F=9

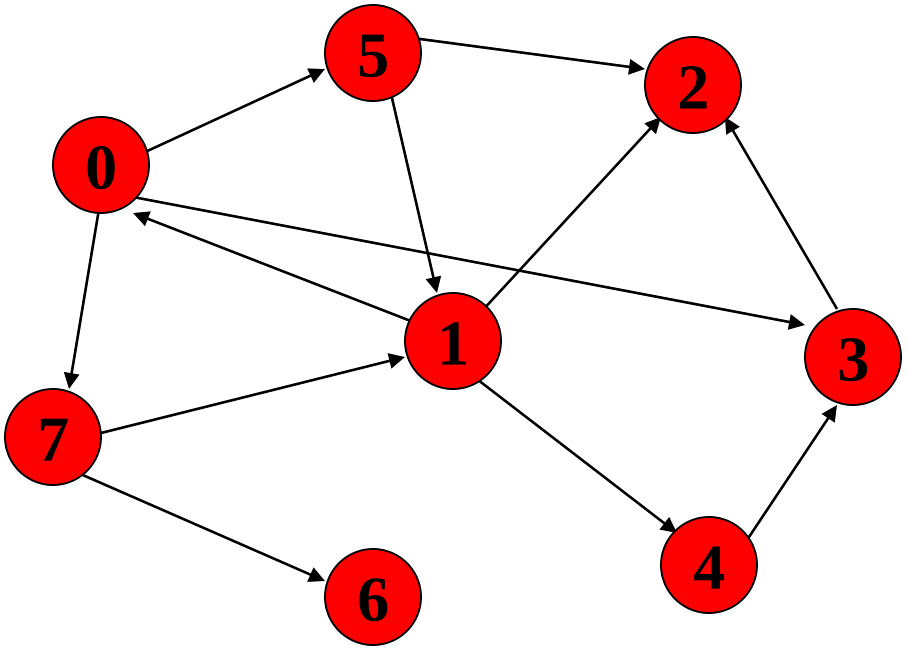
R=9

1	2	3	4	5	6	7	8	9	10
								6	

Dequeue (don’t visit) 3

BFS: Start with Node 5

Visisted: 5 1 2 0 4 3 7 6



F=0

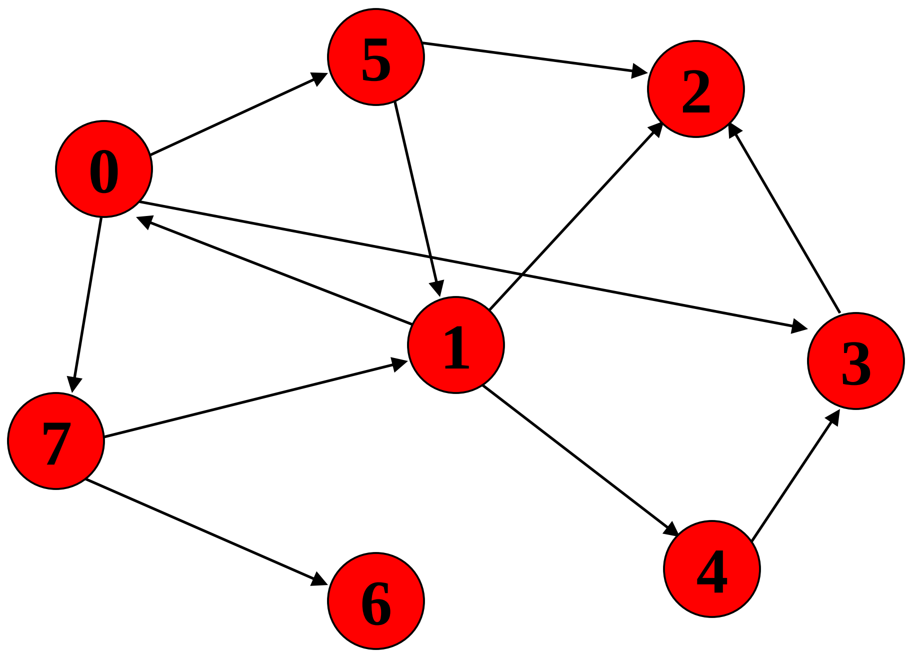
R=0

1 2 3 4 5 6 7 8 9 10

Dequeue/Visit 6

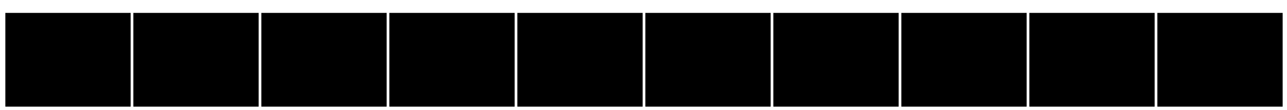
BFS: Start with Node 5

Visisted: 5 1 2 0 4 3 7 6



F=0

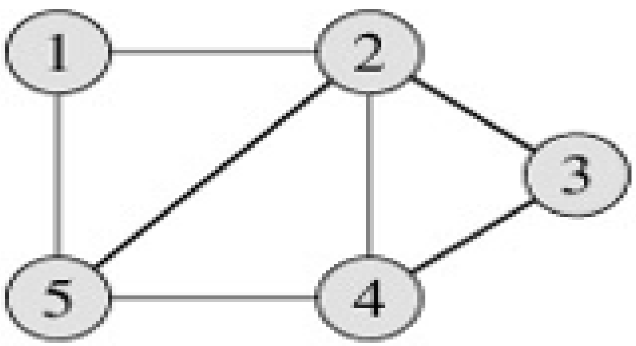
R=0

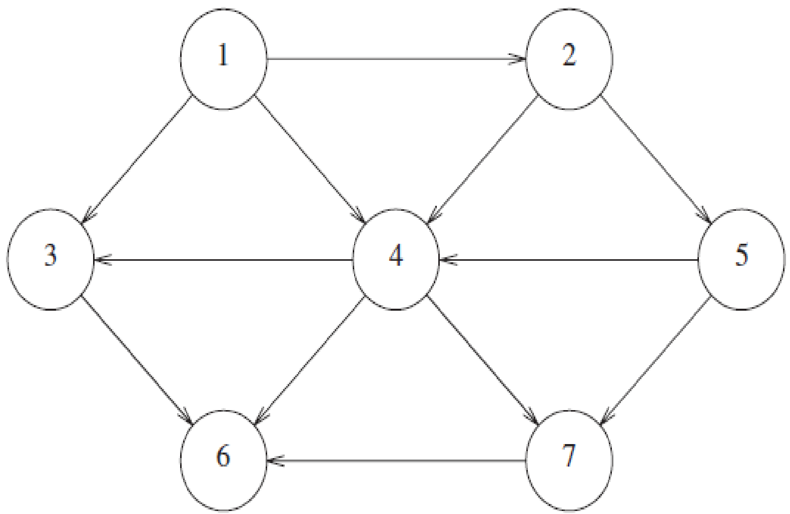


1 2 3 4 5 6 7 8 9 10

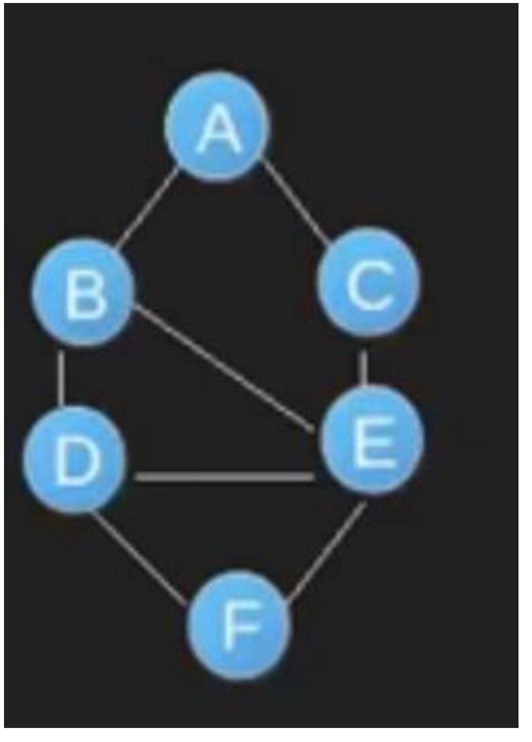
No Enqueue operation

Represent the given graph in adjacency list and adjacency matrix. Also, Identify the BFS and DFS traversal for the given graph.





Represent the given graph in adjacency list and adjacency matrix. Also, Identify the BFS and DFS traversal for the given graph.



Graph Traversal-DFS Applications

1) For an unweighted graph, DFS traversal of the graph produces the **minimum spanning tree** and all pair shortest path tree.

2) **Detecting cycle in a graph**: A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

3) **Path Finding:** We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.

ii) Use a stack S to keep track of the path between the start vertex and the current vertex.

iii) As soon as destination vertex z is encountered, return the path as the contents of the stack.

Graph Traversal-BFS Applications

1) Shortest Path and Minimum Spanning Tree for unweighted graph: In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

Graph Traversal-BFS Applications

3) Crawlers in Search Engines: Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.

4) Social Networking Websites: In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.

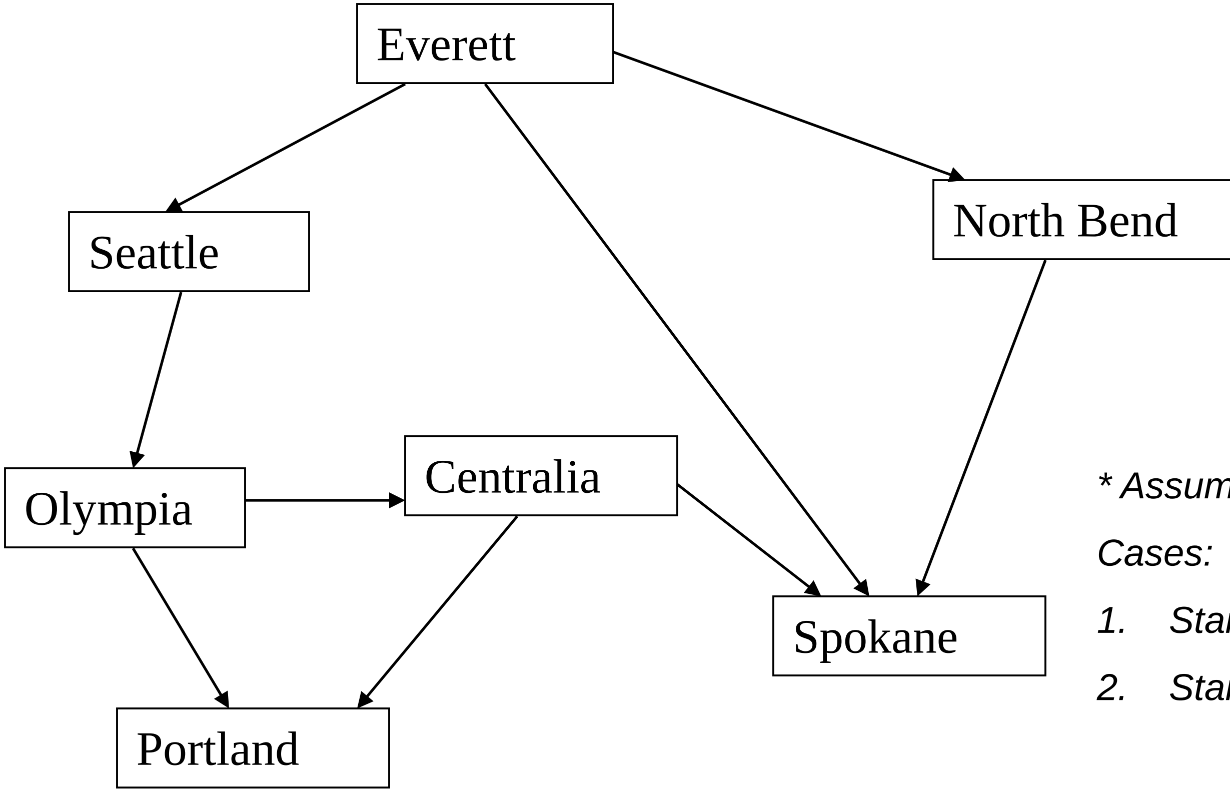
Graph Traversal-BFS Applications

5) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.

6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) **Path Finding:** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

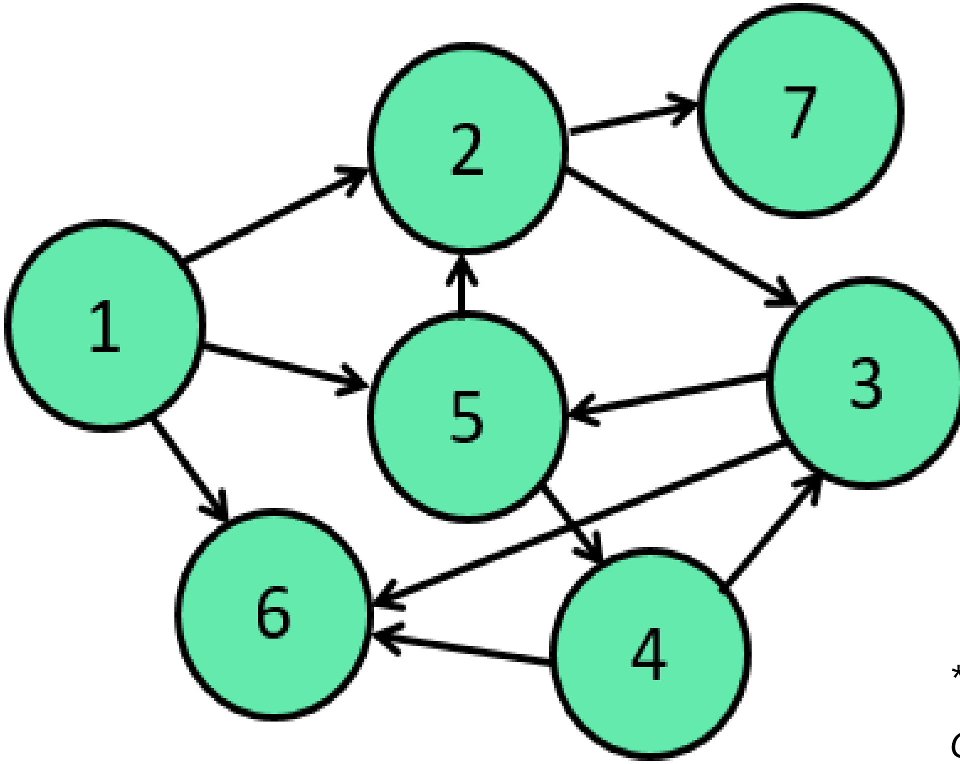
Practice Problem: DFS & BFS



** Assume – edge value 1 for all*

Cases:

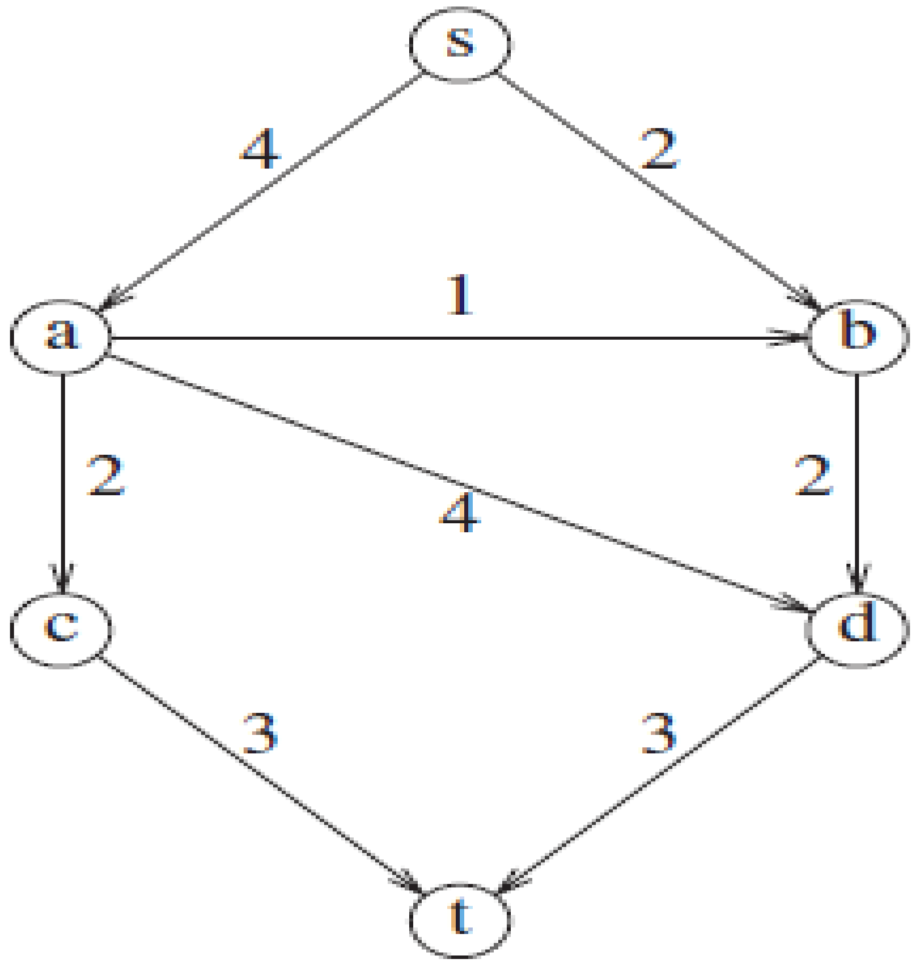
1. *Start with Everett*
2. *Start with Olympia*



** Assume – edge value 1 for all*

Cases:

1. Start with 1


s
a
b
c
d
t
4
2
1
2
4
2
3
3

BFS : (s, a, b, c, d, t)

DFS: (s, a, c, t, b, d)

Thank You.