Data Structure

http://smartlead.hallym.ac.kr

Instructor: Jin Kim

010-6267-8189(033-248-2318)

jinkim@hallym.ac.kr

Office Hours

늠름한 허스키

The Graph ADT (1/13)

- Introduction
  - A graph problem example: Köenigsberg bridge problem

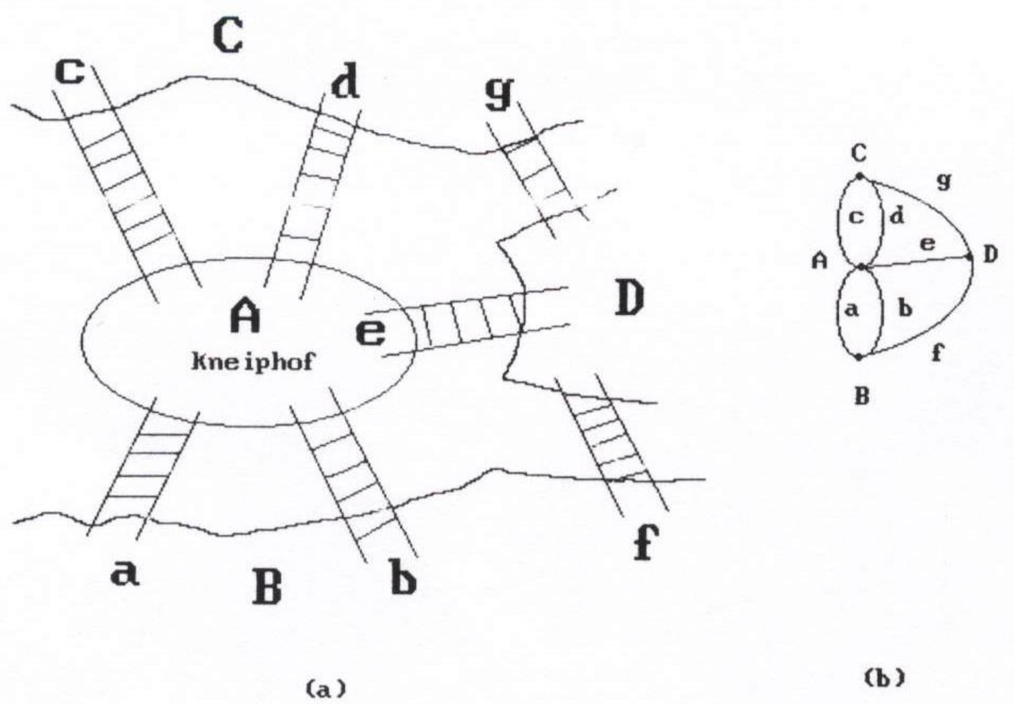


Figure 6.1: The bridges of Koenigsberg

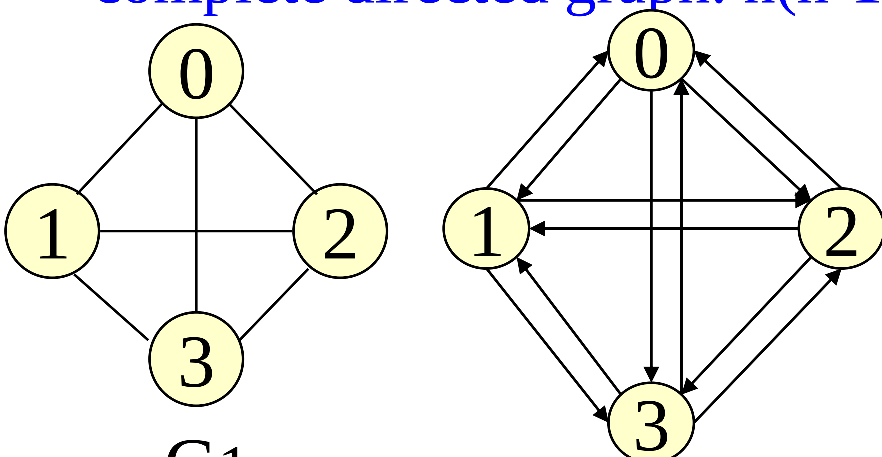
The Graph ADT (2/13)

- Definitions
  - A graph G consists of two sets 그래프는
    - a finite, nonempty set of vertices $V(G)$ 정점의 집합과
    - a finite, possible empty set of edges $E(G)$ 간선의 집합 이다
  - $G(V,E)$ represents a graph
  - An undirected graph(무방향 그래프) is one in which the pair of vertices in an edge is unordered, $(v_0, v_1) = (v_1, v_0)$
  - A directed graph(유방향 그래프)is one in which each edge is a directed pair of vertices, $<v_0, v_1> \; != \; <v_1, v_0>$

tail $\longrightarrow$ head

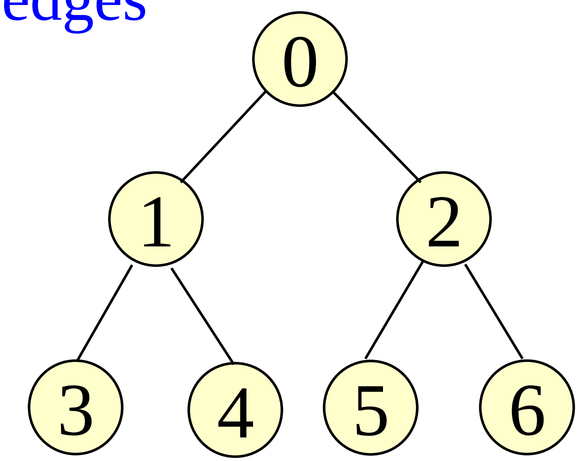
The Graph ADT (3/13)

- Examples for Graph
  - complete undirected graph(완전 무방향 그래프): n(n-1)/2 edges
  - complete directed graph: n(n-1) edges

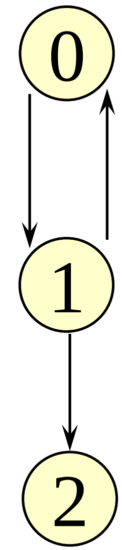


G1

complete graph



G2



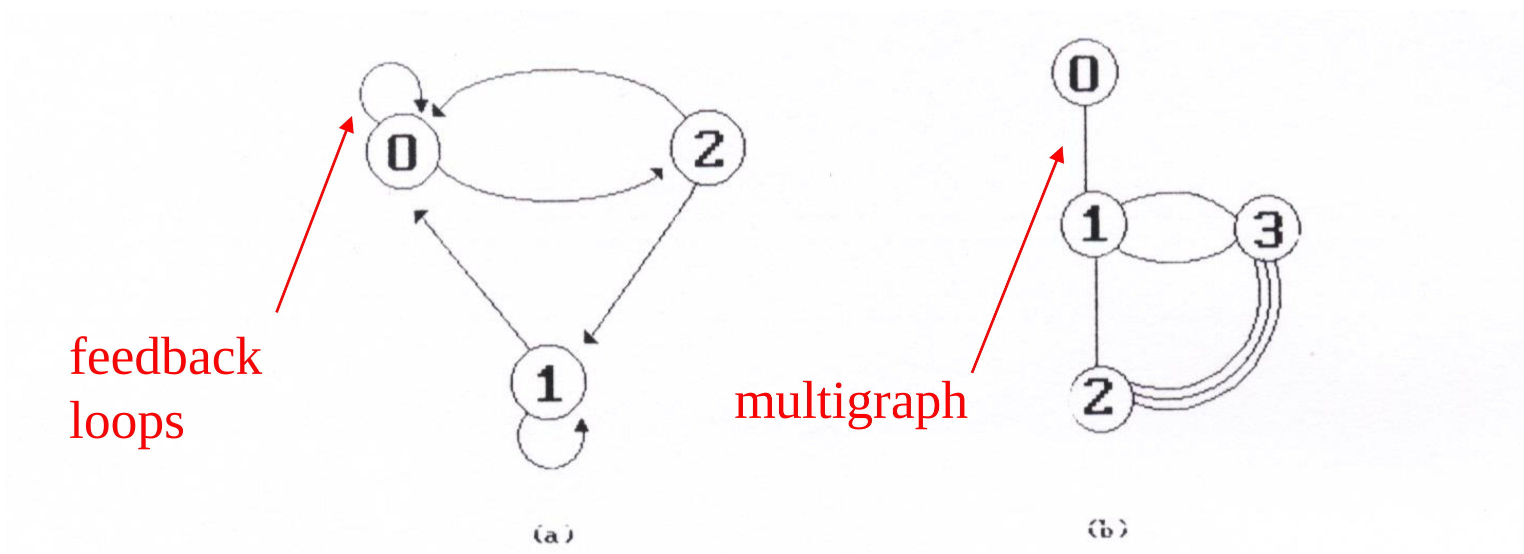
G3

incomplete graph

G1={V,E}, V(G1)={0,1,2,3} E(G1)={(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)}
G2={V,E}, V(G2)={0,1,2,3,4,5,6} E(G2)={(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)}
G3={V,E}, V(G3)={0,1,2} E(G3)={<0,1>,<1,0>,<1,2>}

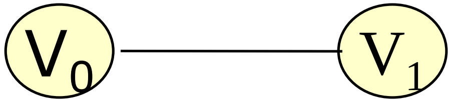
The Graph ADT (4/13)

- Restrictions on graphs (그래프에서 제한)
  - A graph may not have an edge from a vertex, *i*, back to itself. Such edges are known as *self loops(자기루프) 안됨*
  - A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data referred to as a multigraph(멀티그래프) 안됨

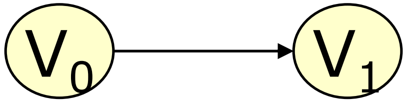


The Graph ADT (5/13)

- Adjacent(인접한) and Incident(부속한)
- If (v_0, v_1) is an edge in an undirected graph,
  - V_0 and v_1 are adjacent(정점들은 인접)
  - The edge (v_0, v_1) is incident(간선은 부속) on vertices v_0 and v_1

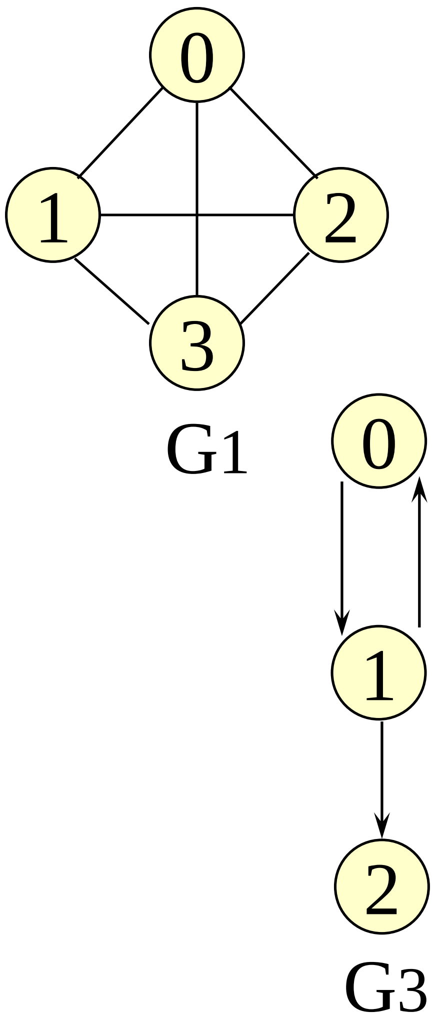


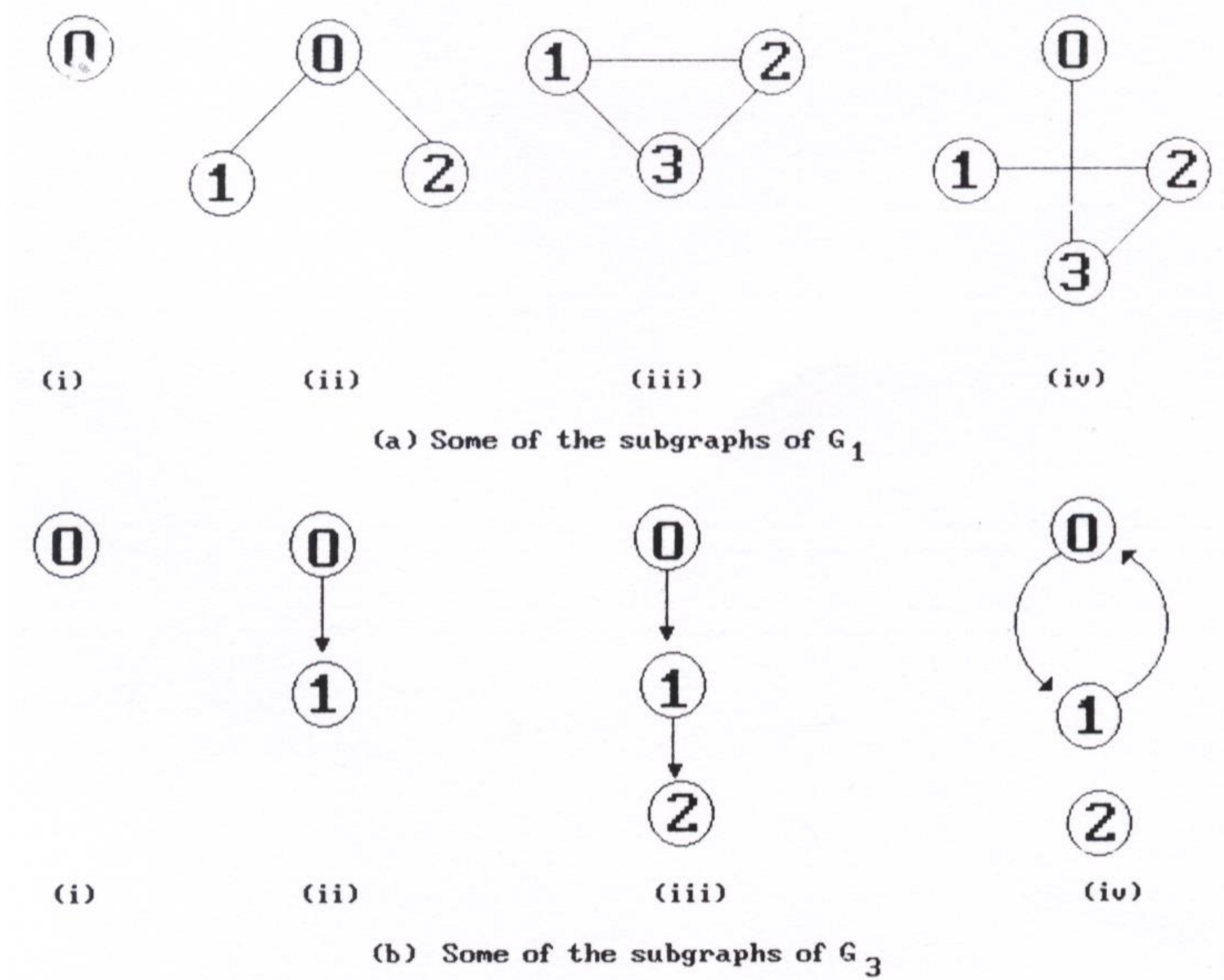
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1



The Graph ADT (6/13)

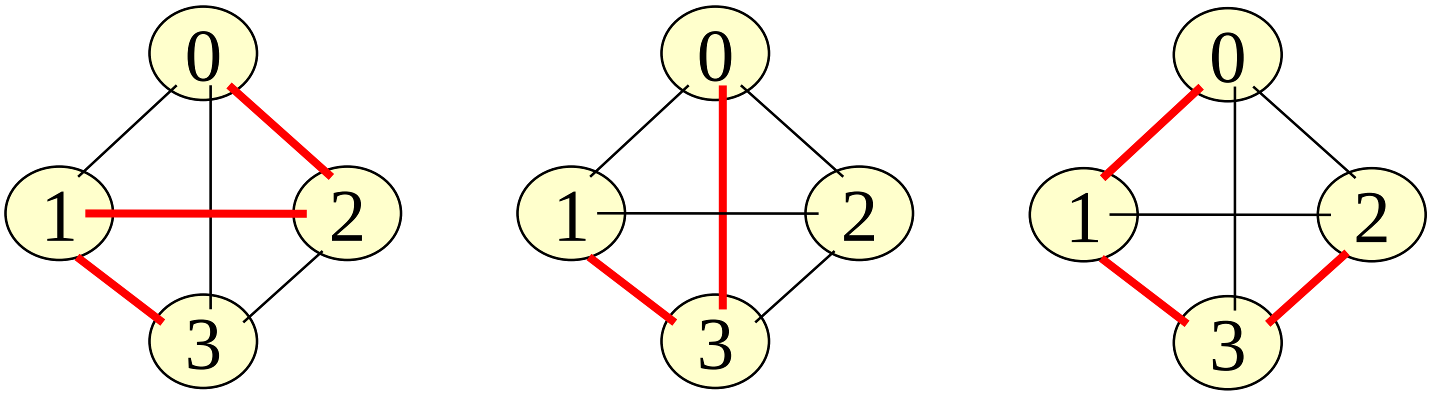
- A subgraph(서브그래프) of *G* is a graph *G*' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.





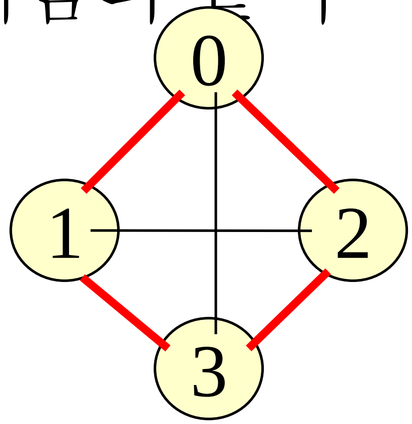
The Graph ADT (7/13)

- Path(경로)
  - 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.
    - A path such as (0, 2), (2, 1), (1, 3) is also written as 0, 2, 1, 3
  - The length of a path (경로의길이)is the number of edges on it



The Graph ADT (8/13)

- Simple path and cycle(단순 경로와 사이클)
  - simple path (simple directed path): a path in which all vertices, except possibly the first and the last, are distinct. (경로의 처음과 끝이 다름, 중간에 중복정점없음)
  - A cycle is a simple path in which the first and the last vertices are the same.(경로의 처음과 끝이 같음→사이클)

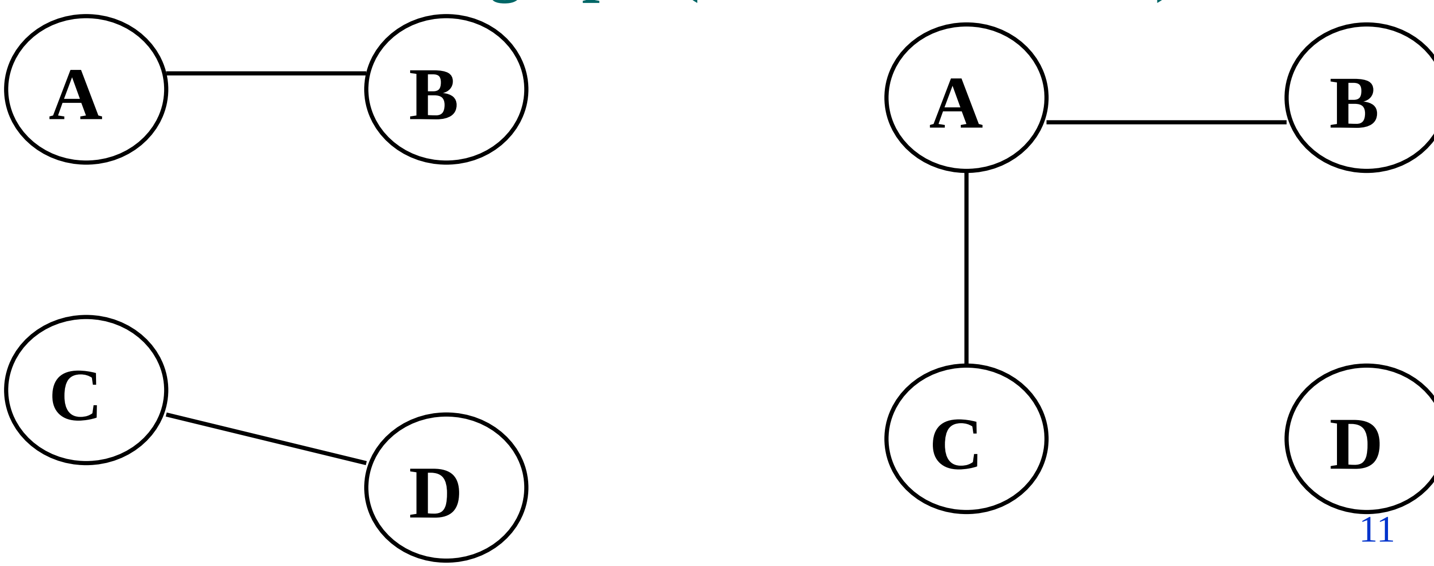


Connected and Disconnected graphs
연결그래프, 단절그래프

Connected graph(연결그래프): There is a path between each two vertices 어느 두 정점사이에도 경로존재

Disconnected graph(단절그래프) : There are at least two vertices not connected by a path. 그렇지 않을 경우

Examples of disconnected graphs(단절그래프의 예):

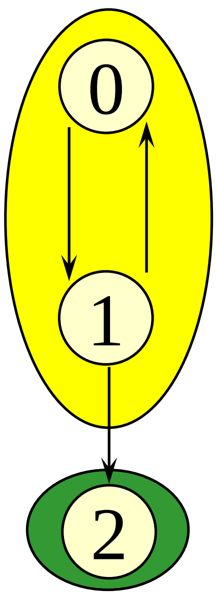


The Graph ADT (9/13)

- Connected graph(연결그래프)-무방향그래프
  - 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, v_j, there is a path from v_i to 퍼
  - 연결그래프의 어느 두 정점을 선택해도, 그 두 정점사이에 경로가 존재.
- Connected component(연결요소)-무방향 그래프
  - A connected component of an undirected graph is a maximal connected subgraph. 연결요소는 무방향그래프에서 최대연결부분그래프를 의미
  - A tree is a graph that is connected and acyclic *(i.e, has no cycle)*. (트리는 그래프의 일종. 연결되었고 사이클이 존재하지 않음)
  - 매우 중요

The Graph ADT (10/13)

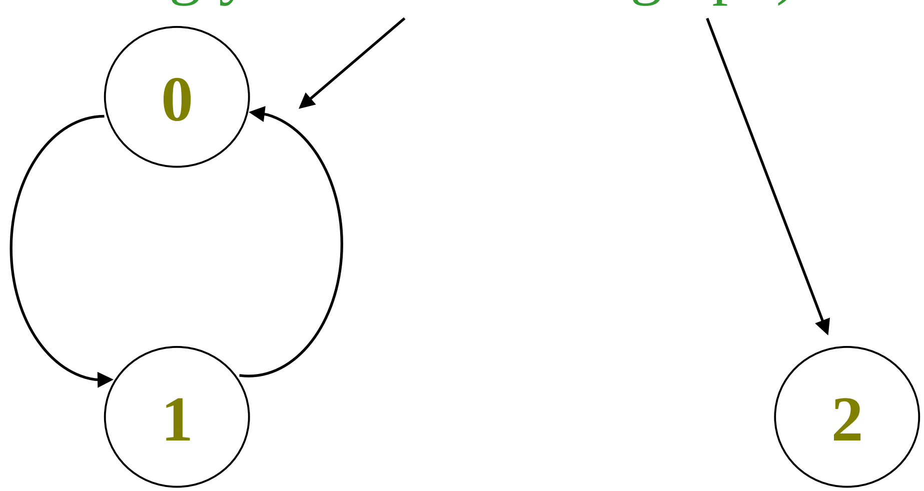
- Strongly Connected Component(강력연결요소)-유방향그래프
  - 유방향그래프의 어느 두 정점을 선택해도 양방향으로 경로가 존재하면 강력연결그래프라고 함. A directed graph is strongly connected if there is a directed path from *vi* to *vj* and also from *vj* to *vi*
  - A strongly connected component is a maximal subgraph that is strongly connected강력연결요소는 강력하게 연결된



G3

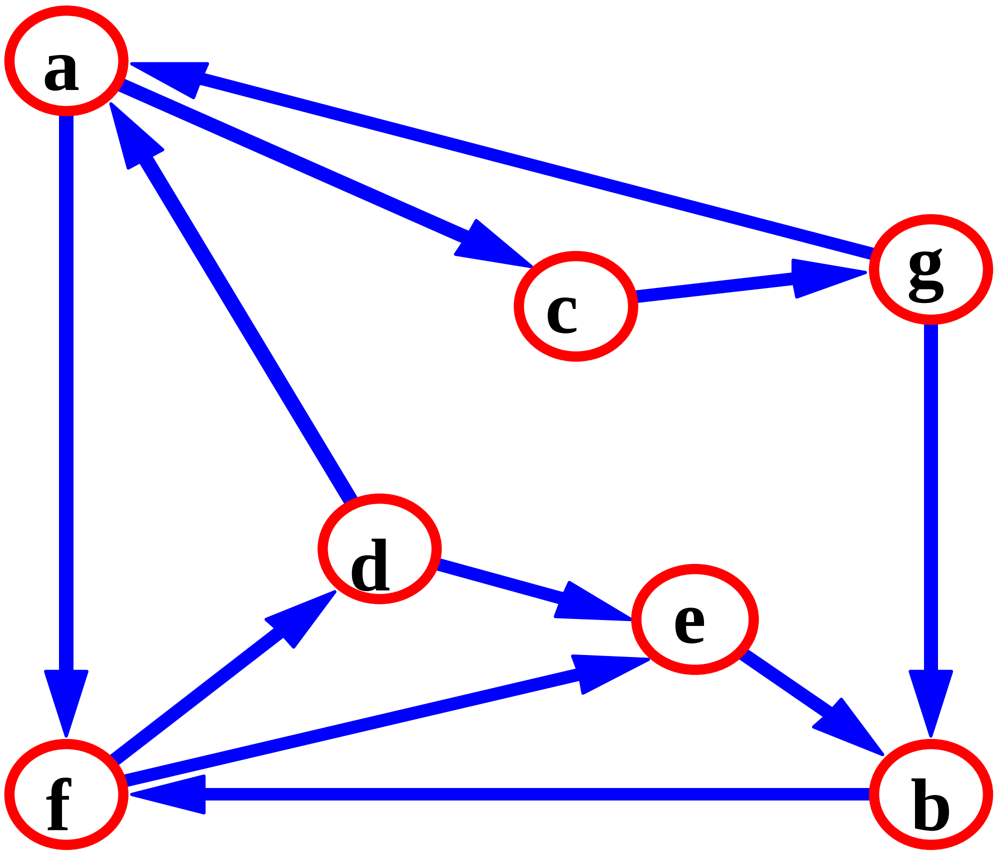
not strongly connected

strongly connected component강력연결요소
(maximal strongly connected subgraph)



Strongly Connected Directed graphs
강력연결그래프(유방향그래프)

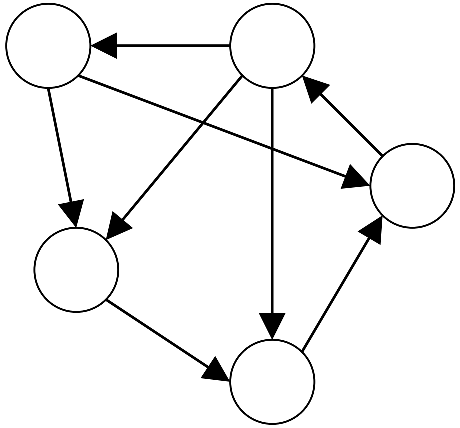
- Every pair of vertices are reachable from each other



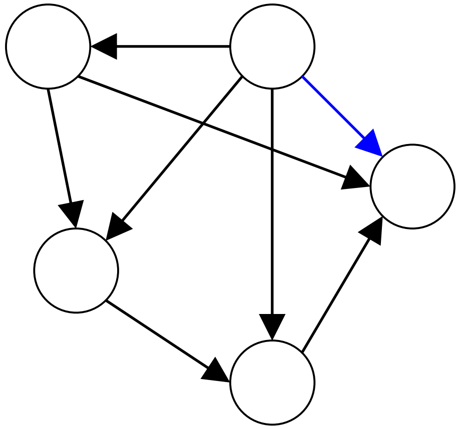
Strongly-Connected
강력연결그래프(유방향)

Graph *G* is ***strongly connected*** if, for every *u* and *v* in *V*, there is some path from *u* to *v* and some path from *v* to *u*.

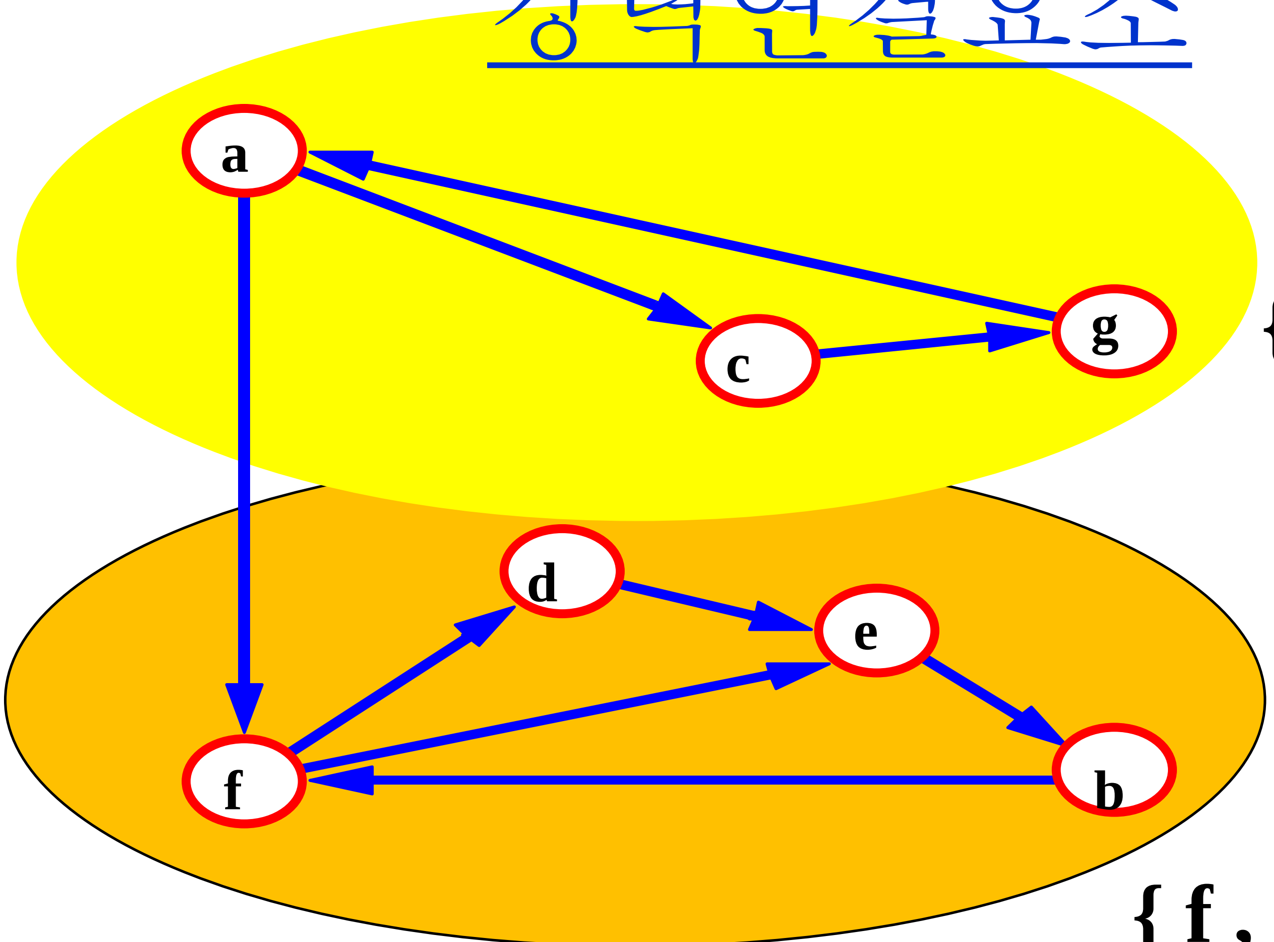
Strongly Connected



Not Strongly Connected



Strongly Connected Components
강력연결요소



{ a , c , g }

{ f , d , e , b }

The Graph ADT (11/13)

- Degree(차수)
  - The degree of a vertex is the number of edges incident to that vertex.
- For directed graph
  - in-degree (v)(진입차수) : the number of edges that have v as the head
  - out-degree (v)(진출차수) : 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

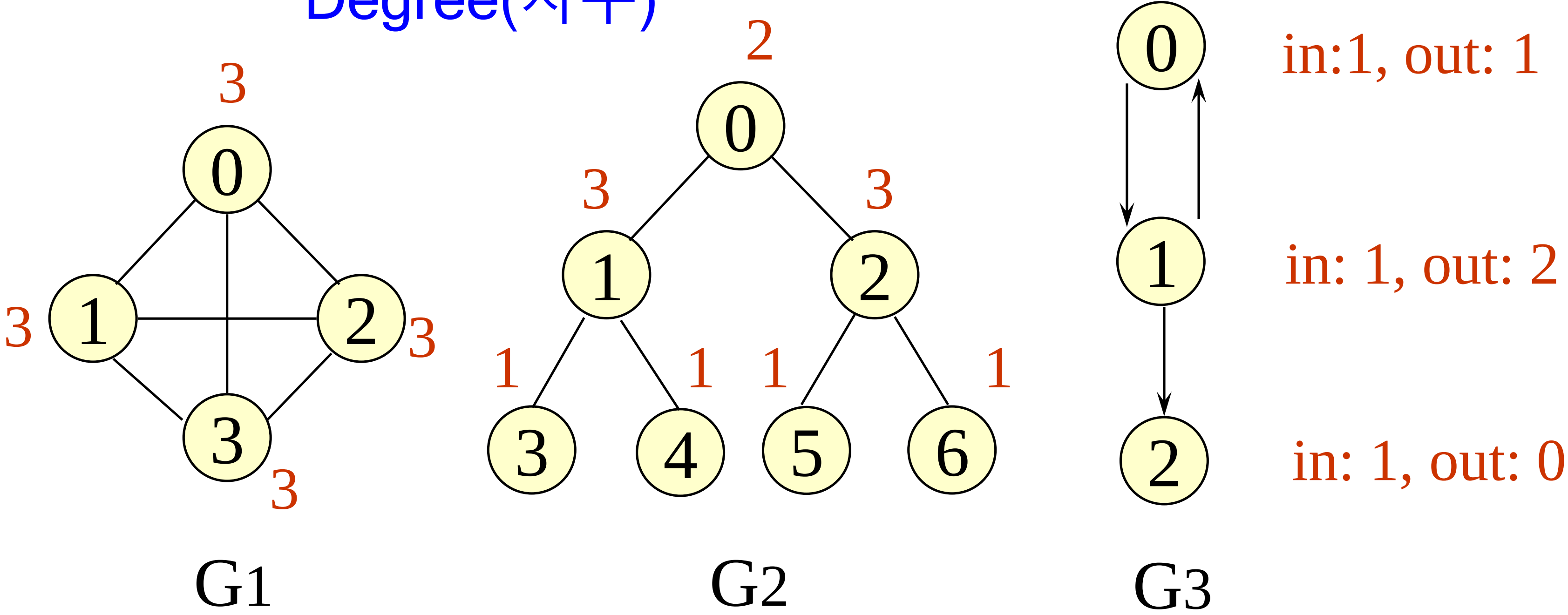
$$e = \left(\sum_{0}^{n-1} d_i\right) / 2$$

The Graph ADT (12/13)

- We shall refer to a directed graph as a *digraph*. When we us the term *graph*, we assume that it is an undirected graph 일반적으로 유방향그래프를 digraph라고도 함. 그래프라고 통칭할 때는 무방향그래프라 약속하자.

directed graph(유방향)
in-degree & out-degree
진입차수&진출차수

undirected graph(무방향)
Degree(차수)



The Graph ADT (13/13)

- Abstract Data Type *Graph*

ADT Graph

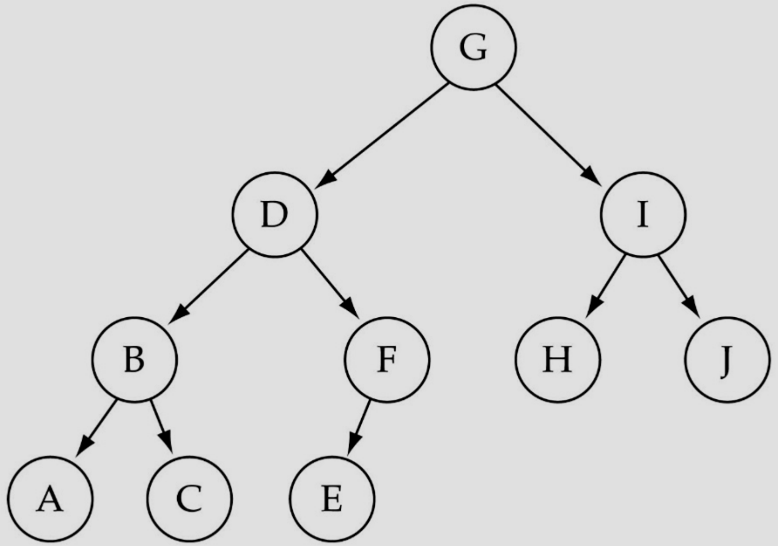
데이타 : 공백이 아닌 정점(vertex)의 유한집합(V)과 간선(edge)의 집합(E).
　　　　여기서 간선은 두 정점의 쌍.

연산 내용 : G∈Graph, u,v ∈V;
　createG() ::= create an empty graph;
　insertVertex(G, v) ::= insert vertex v into G;
　insertEdge(G, u, v) ::= insert edge (u, v) into G;
　deleteVertex(G, v) ::= delete vertex v and all edges incident on v from G;
　deleteEdge(G, u, v) ::= delete edge (u, v) from G;
　isEmpty(G) ::= **if** G has no vertex **then return** true, **else return** false;
　adjacent(G, v) ::= **return** set of all vertices adjacent to v;
End Graph

Trees vs graphs

- Trees are special cases of graphs!!(트리는 그래프의 특수한 종류)

(c) Graph3 is a directed graph.



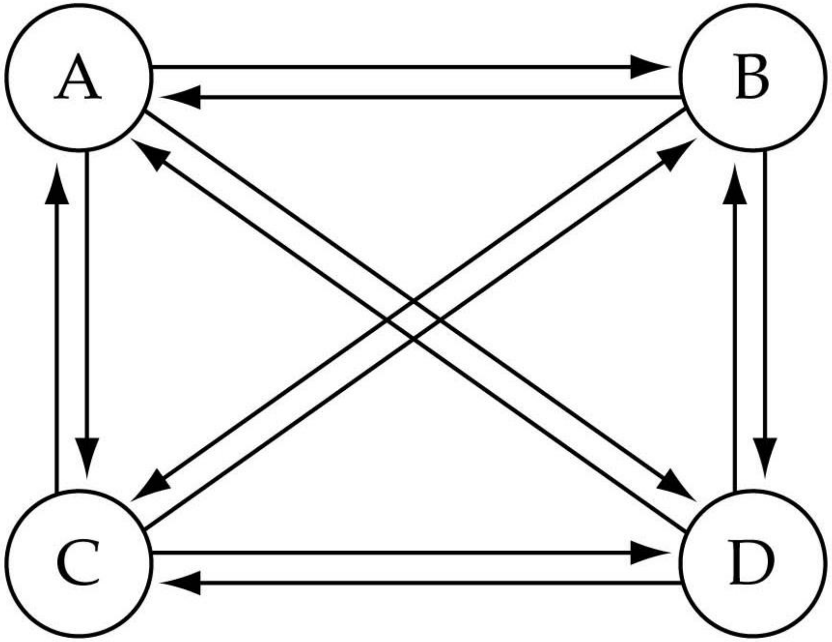
V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, J), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

Graph terminology 그래프용어

- What is the number of edges in a complete directed graph(유방향완전연결그래프) with N vertices? 간선의 개수?

*N * (N-1)*

$O(N^2)$



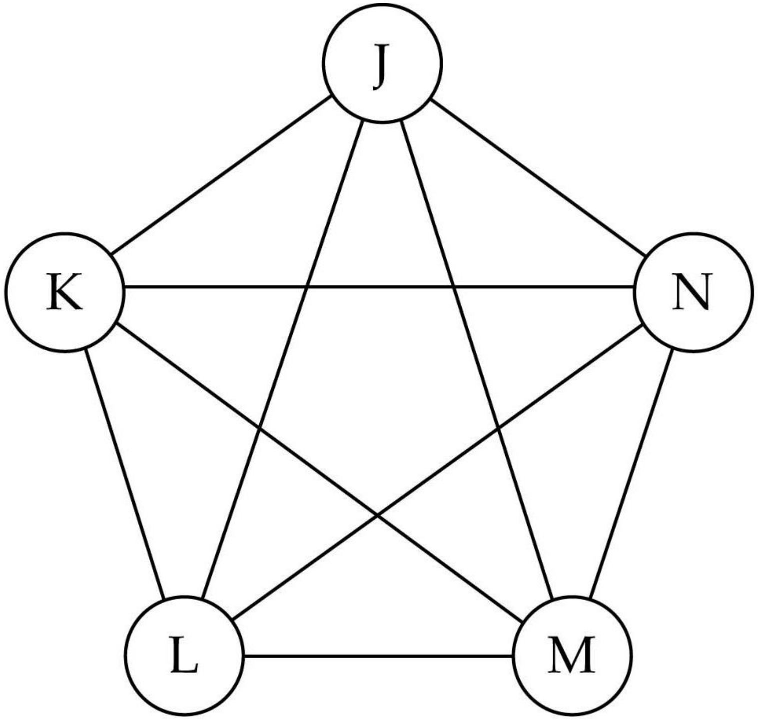
(a) Complete directed graph.

Graph terminology(그래프용어)

- What is the number of edges in a complete undirected graph with N vertices? 무방향완전연결그래프

  *N * (N-1) / 2(최대간선의 개수)*

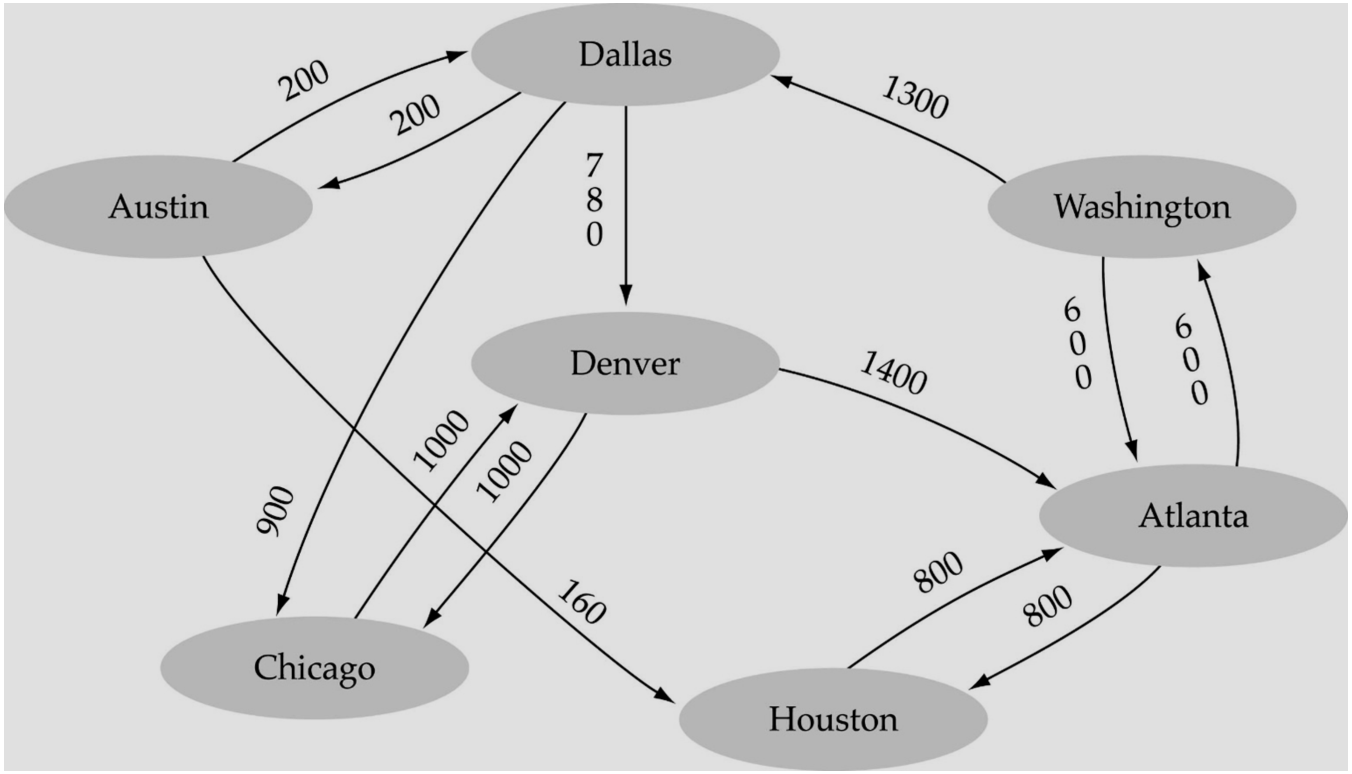
$$O(N^2)$$



(b) Complete undirected graph.

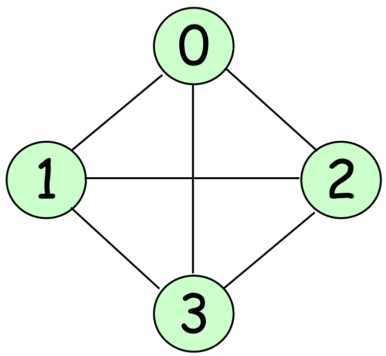
Graph terminology (그래프용어)

- Weighted graph(가중치그래프): a graph in which each edge carries a value



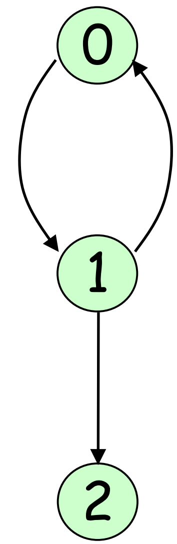
Adjacency Matrix(인접행렬로표현)

- Adjacency Matrix is a two dimensional n×n array.(인접행렬은 이차원배열)



$$
\begin{array}{c|cccc|}
 & 0 & 1 & 2 & 3 \\
0 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
2 & 1 & 1 & 0 & 1 \\
3 & 1 & 1 & 1 & 0
\end{array}
$$

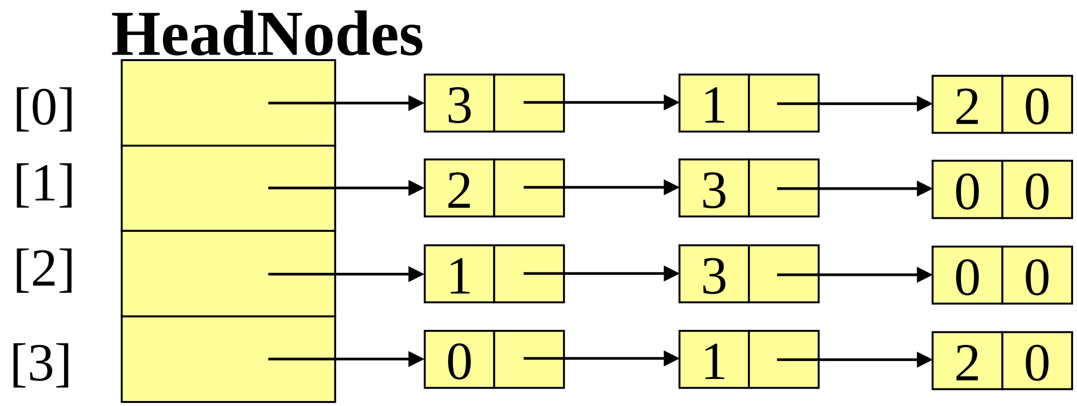
(a) G_1

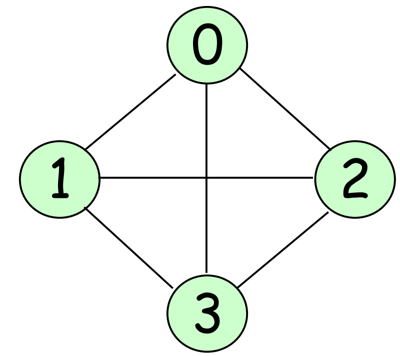


$$
\begin{array}{c|ccc|}
 & 0 & 1 & 2 \\
0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
2 & 0 & 0 & 0
\end{array}
$$

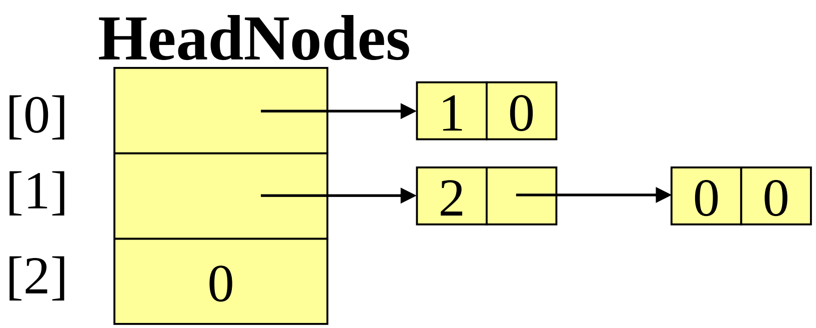
(b) G_3

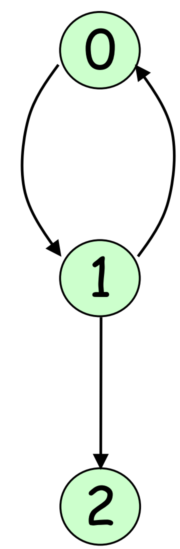
Adjacency Lists(리스트표현)





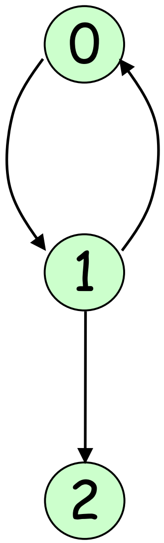
(a) G_1

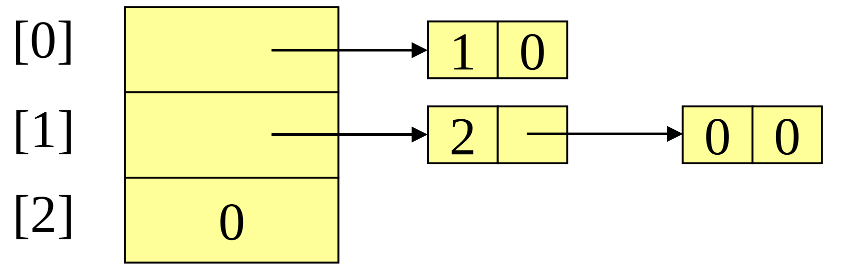


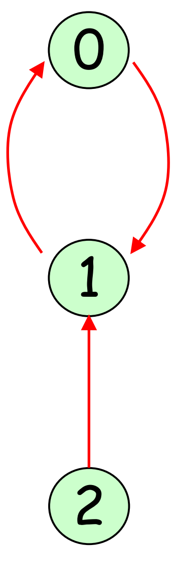


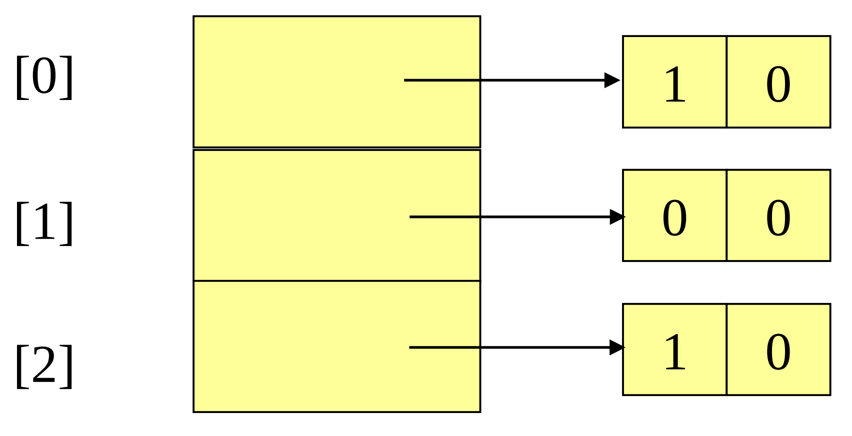
(b) G_3

Inverse Adjacency Lists(역인접리스트)



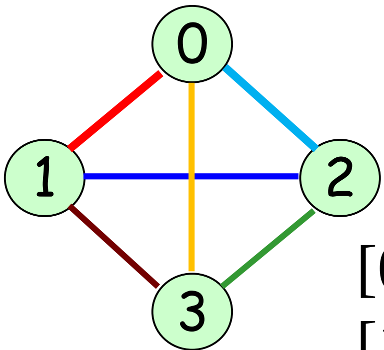






Multilists(멀티리스트로 표현)

m	vertex1	vertax2	list1	list2



Six edges

HeadNodes

[0]
[1]
[2]
[3]

	m	vertex1	vertax2	list1	list2	
N0		0	1	N1	N3	edge (0, 1)
N1		0	2	N2	N3	edge (0, 2)
N2		0	3	0	N4	edge (0, 3)
N3		1	2	N4	N5	edge (1, 2)
N4		1	3	0	N5	edge (1, 3)
N5		2	3	0	0	edge (2, 3)

The lists are

Vertex 0: N0 -> N1 -> N2

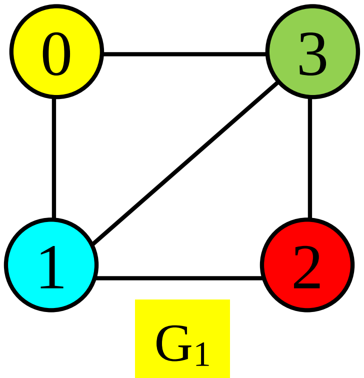
Vertex 1: N0 -> N3 -> N4

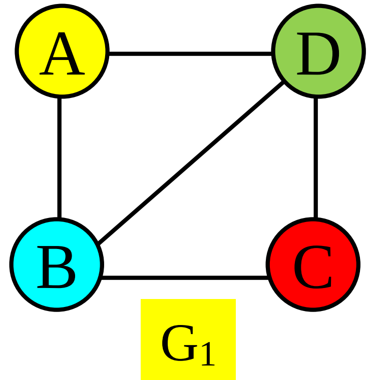
Vertex 2: N1 -> N3 -> N5

Vertex 3: N2 -> N4 -> N5

Six edges여섯개 간선

Adjacency Lists: Sequential Representation
인접리스트 : 선형표현





vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]

리스트 시작과
배열 크기 정보

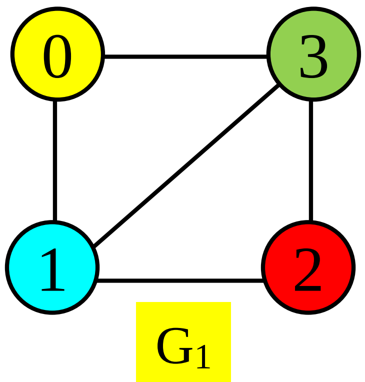
간선 정보

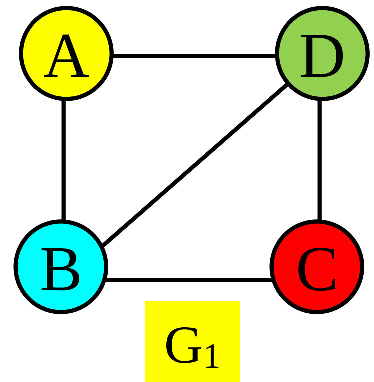
vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12	15	B	D	A	C	D	B	D	A	B	C

리스트 시작과
배열 크기 정보

간선 정보

Adjacency Lists: Sequential Representation
인접리스트 : 선형표현





vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]

리스트 시작과
배열 크기 정보

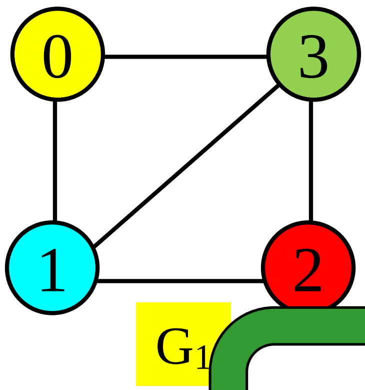
간선 정보

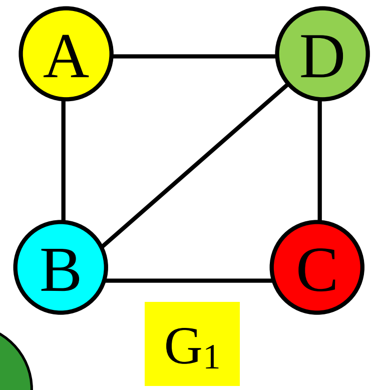
vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12	15	B	D	A	C	D	B	D	A	B	C

리스트 시작과
배열 크기 정보

간선 정보

Adjacency Lists: Sequential Representation
인접리스트 : 선형표현





vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5					1	3								

리스트 시작과
배열 크기 정보

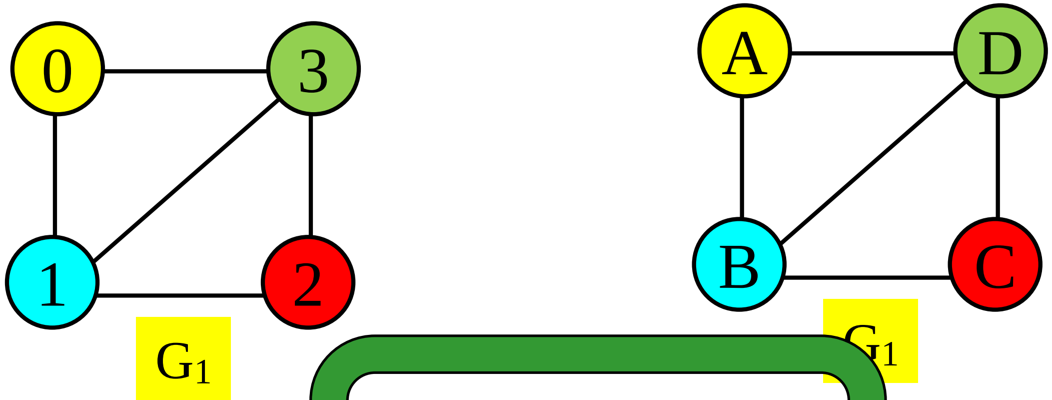
간선 정보

vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12	15	B	D	A	C	D	B	D	A	B	C

리스트 시작과
배열 크기 정보

간선 정보

Adjacency Lists: Sequential Representation
인접리스트 : 선형표현



vertex	[0]	[	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7				1	3	0	2	3					

리스트 시작과
배열 크기 정보

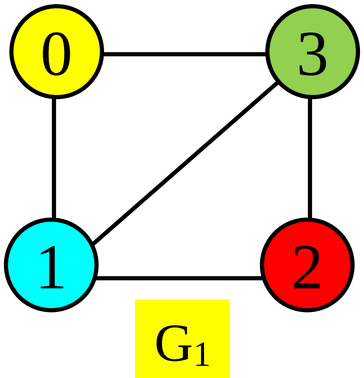
간선 정보

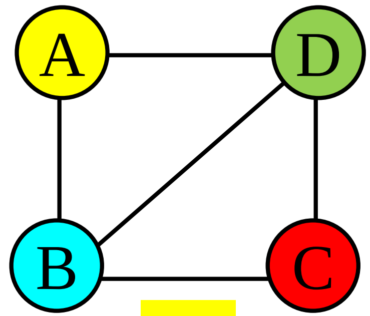
vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12	15	B	D	A	C	D	B	D	A	B	C

리스트 시작과
배열 크기 정보

간선 정보

Adjacency Lists: Sequential Representation
인접리스트 : 선형표현





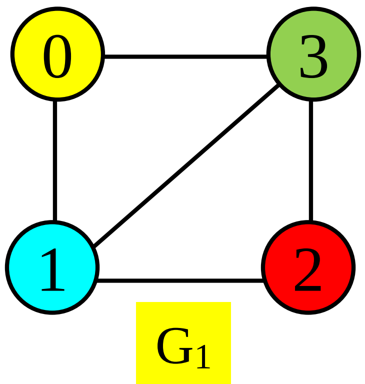
vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10			1	3	0	2	3	1	3			

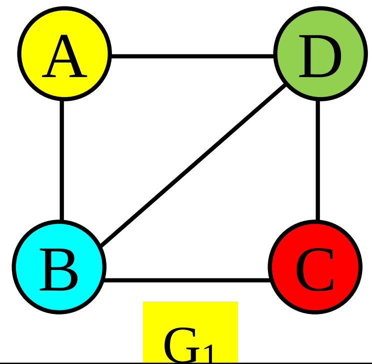
리스트 시작과 배열 크기 정보 간선 정보

vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12	15	B	D	A	C	D	B	D	A	B	C

리스트 시작과 배열 크기 정보 간선 정보

Adjacency Lists: Sequential Representation
인접리스트 : 선형표현





vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12		1	3	0	2	3	1	3	0	1	2

리스트 시작과
배열 크기 정보

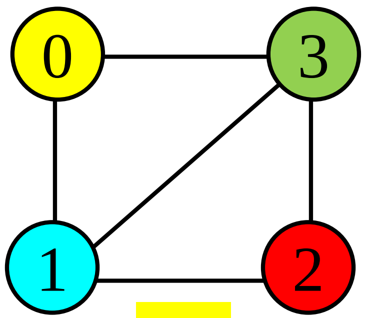
간선 정보

vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12	15	B	D	A	C	D	B	D	A	B	C

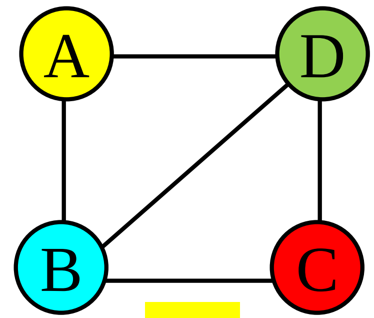
리스트 시작과
배열 크기 정보

간선 정보

Adjacency Lists: Sequential Representation
인접리스트 : 선형표현



G_1



vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12	15	1	3	0	2	3	1	3	0	1	2

리스트 시작과
배열 크기 정보

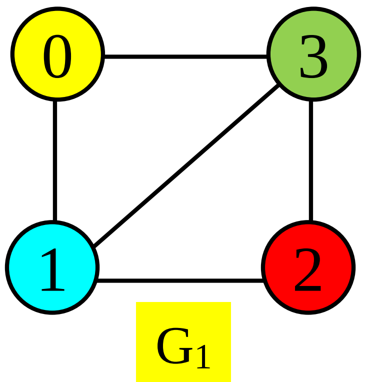
간선 정보

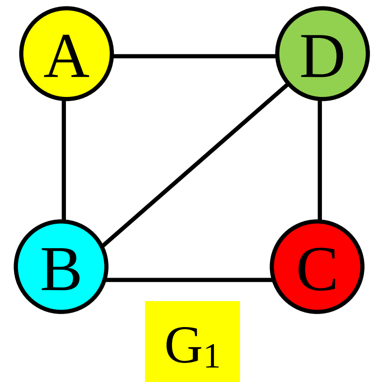
vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12	15	B	D	A	C	D	B	D	A	B	C

리스트 시작과
배열 크기 정보

간선 정보

Adjacency Lists: Sequential Representation
인접리스트 : 선형표현





vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12	15	1	3	0	2	3	1	3	0	1	2

리스트 시작과 배열 크기 정보 | 간선 정보

vertex	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	5	7	10	12	15	B	D	A	C	D	B	D	A	B	C

리스트 시작과 배열 크기 정보 | 간선 정보

Pros and Cons of Adjacency Matrices 인접행렬

- Pros: (장점)
  - Simple to implement(구현하기 간단하다)
  - Easy and fast to tell if a pair (I,j) is an edge: simply check if A[i][j] is 1 or 0 ((i,j)에 간선이 있는지 알기쉽다)
- Cons: (단점)
  - No matter how few edges the graph has, the matrix takes $O(n^2)$ in memory (아무리 간선수가적어도 $O(n^2)$개의 방이 필요)

Pros and Cons of Adjacency Lists 인접리스트

- Pros:(장점, for)
  - Saves on space (memory): the representation takes as many memory words as there are nodes and edge.(공간이 절약된다)
- Cons: (단점, against)
  - It can take up to O(n) time to determine if a pair of nodes (I,j) is an edge: one would have to search the linked list L[i], which takes time proportional to the length of L[i]. (i,j)사이에 간선이 있음을 알기위해 연결리스트탐색시간필요

Adjacency matrix vs. adjacency list representation

- **Adjacency matrix(인접행렬)**
  - Good for dense graphs(밀집그래프에 좋다) -- $|E| \sim O(|V|^2)$
  - Memory requirements: $O(|V| + |E|) = O(|V|^2)$
  - Connectivity between two vertices can be tested quickly
- **Adjacency list(인접리스트)**
  - Good for sparse graphs(희소그래프에좋다) -- $|E| \sim O(|V|)$
  - Memory requirements: $O(|V| + |E|) = O(|V|)$
  - Vertices adjacent to another vertex can be found quickly(이웃한정점을 알기쉽다)

Graph searching(그래프탐색) 순회방문을 생각

- *Problem(문제)*: find a path between two nodes of the graph (e.g., Austin and Washington) 두 정점사이에 경로가 존재할까?
- *Methods*: Depth-First-Search (DFS) or Breadth-First-Search (BFS) 깊이우선탐색, 너비우선탐색

Overview

- Goal
  - To systematically visit the nodes of a graph
    정점들을 효율적으로 방문하고 싶다.
- A tree is a directed, acyclic, graph (DAG)(트리는 유방향, 사이클없는 그래프)
- If the graph is a tree,(그래프가 트리이면)
  - DFS is exhibited by preorder, postorder, and (for binary trees) inorder traversals(전위,중위,후위순회)깊이우선탐색은 전위, 중위,후위선회로 탐색
  - BFS is exhibited by level-order traversal너비우선탐색은 (레벨순서방문)

Depth-First-Search (DFS)깊이우선탐색

- What is the idea behind DFS?
  - Travel as far as you can down a path(갈수있을때까지내려가자)
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)막다른 골목이면 표시하고 되돌아나오자.
- DFS can be implemented efficiently using a *stack*

(깊이우선탐색은 스택사용하면 효과적임)

Depth-First-Search (DFS) *(cont.)*

(1) 정점 i를 방문한다.

(2) 정점 i에 인접한 정점 중에서 아직 방문하지 않은 정점이 있으면, 이 정점들을 모두 스택에 저장한다.

(3) 스택에서 정점을 삭제하여 새로운 i를 설정하고, 단계 (1)을 수행한다.

(4) 스택이 공백이 되면 연산을 종료한다.

$$\text{visited}[i] = \begin{cases} \text{true}, & \text{방문하였음} \\ \text{false}, & \text{방문하지 않았음} \end{cases}$$

Depth First Search(스택사용)

```
DFS(i)
  // i 는 시작 정점
  for (i←0; i<n; i ← i +1) do {
      visited[i] ← false;     // 모든 정점을 방문 안한 것으로 마크
  }
  createStack();     //방문할 정점을 저장하는 스택
  push(Stack, i);     // 시작 정점 i 를 스택에 저장
  while (not isEmpty(Stack)) do {     // 스택이 공백이 될 때까지 반복 처리
      j ← pop(Stack);
      if (visited[j] = false) then {     //정점 j를 아직 방문하지 않았다면
          visit j;                              // 직접 j를 방문하고
          visited[j] ← true;            // 방문 한 것으로 마크
          for (each k ∈ adjacency(j)) do {     // 정점 j에 인접한 정점 중에서
              if (visited[k] = false) then       // 아직 방문하지 않은 정점들을
                  push(Stack, k);                    // 스택에 저장
          }
      }
  }
end DFS()
```

Breadth-First-Searching (BFS)너비우선탐색

- What is the idea behind BFS?
  - Look at all possible paths at the same depth before you go at a deeper level 같은 레벨의 모든 원소를 방문하고 다음레벨로가자.
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex) 막다른 골목이면 표시하고 되돌아 나가자

Breadth-First-Searching (BFS) (cont.)

(1) 정점 i를 방문한다.

(2) 정점 i에 인접한 정점 중에서 아직 방문하지 않은 정점이 있으면, 이 정점들을 모두 큐에 저장한다.

(3) 큐에서 정점을 삭제하여 새로운 i를 설정하고, 단계 (1)을 수행한다.

(4) 큐가 공백이 되면 연산을 종료한다.

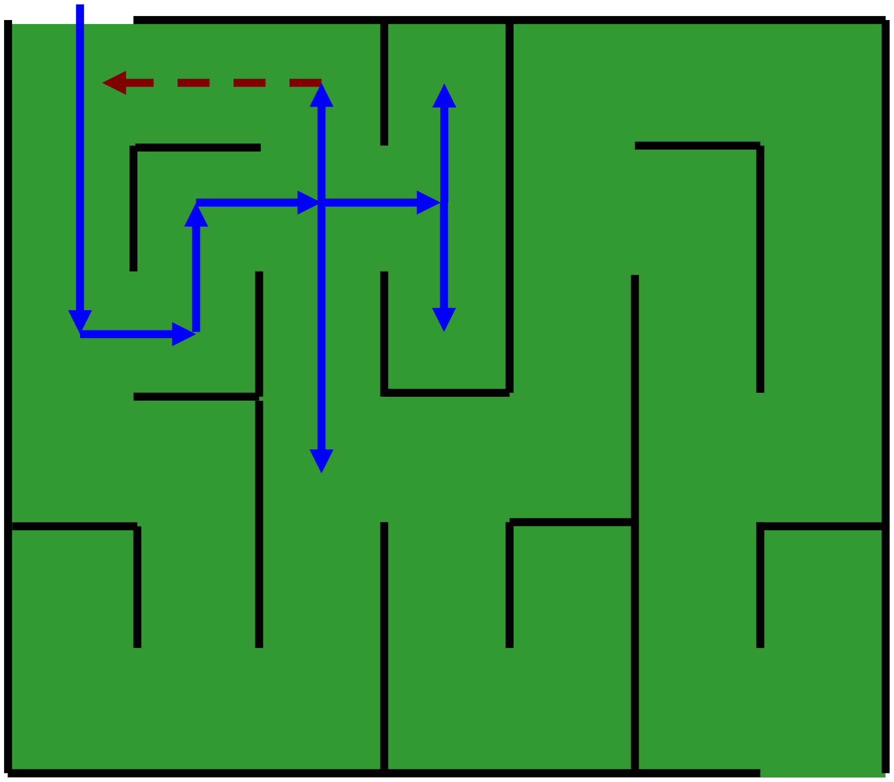
Breadth First Search(큐사용)

```
BFS(i)
  // i는 시작 정점
  for (i←0; i<n; i ←i+1) do {
      visited[i] ← false;    // 모든 정점을 방문 안 한 것으로 마크
  }
  visited[i] ← true;
  createQ();    // 방문할 정점을 저장하는 큐
  enquere(Q, i);
  while (not isEmpty(Q)) do {
      j ← dequeue(Q);
      if (visited[j] = false) then {
          visit j;
          visited[j] ← true;
      }
      for (each k ∈ adjacency(j)) do {
          if (visited[k] = false) then {
              enqueue(Q, k);
          }
      }
  }
end BFS()
```

DFS: Example1

DFS and Maze Traversal
미로찾기는 깊이우선탐색

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



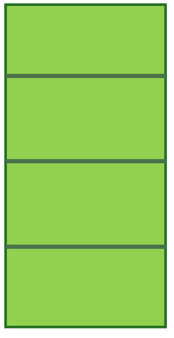
Maze 미로

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

maze

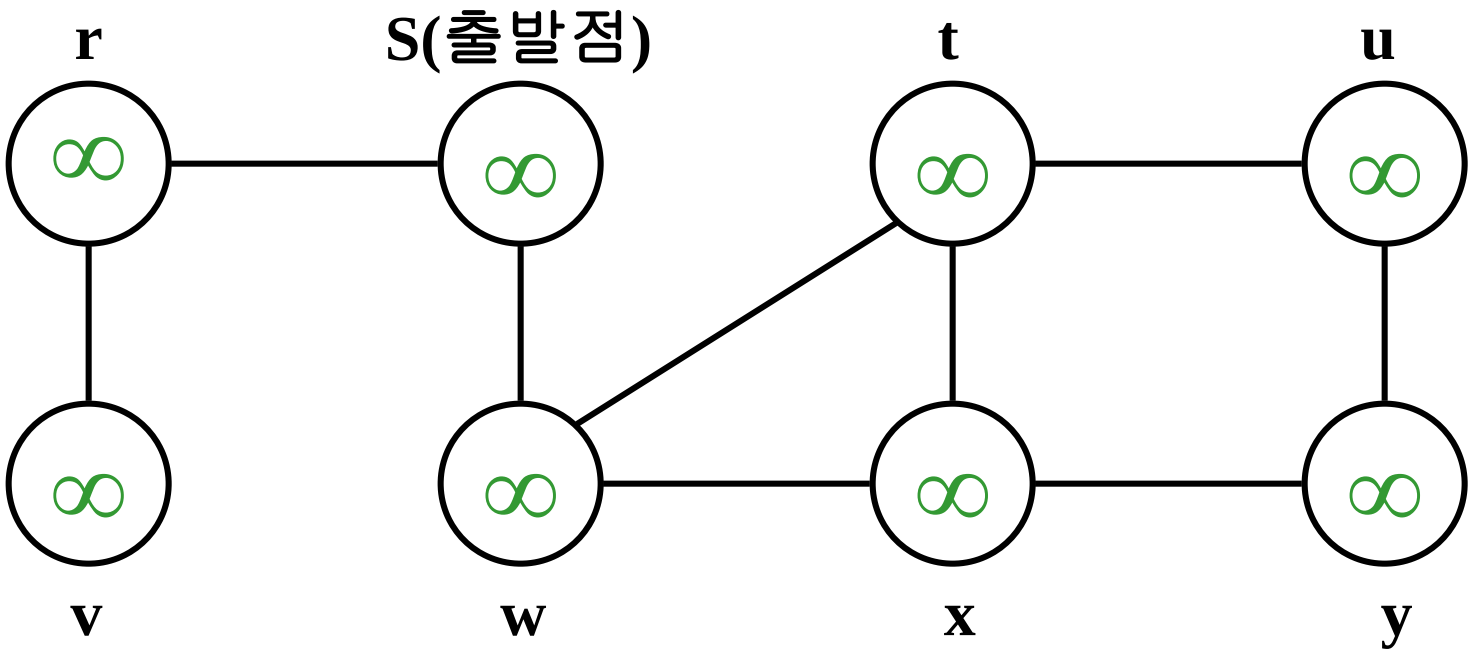
	1	2	3	4	5	6
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0

mark



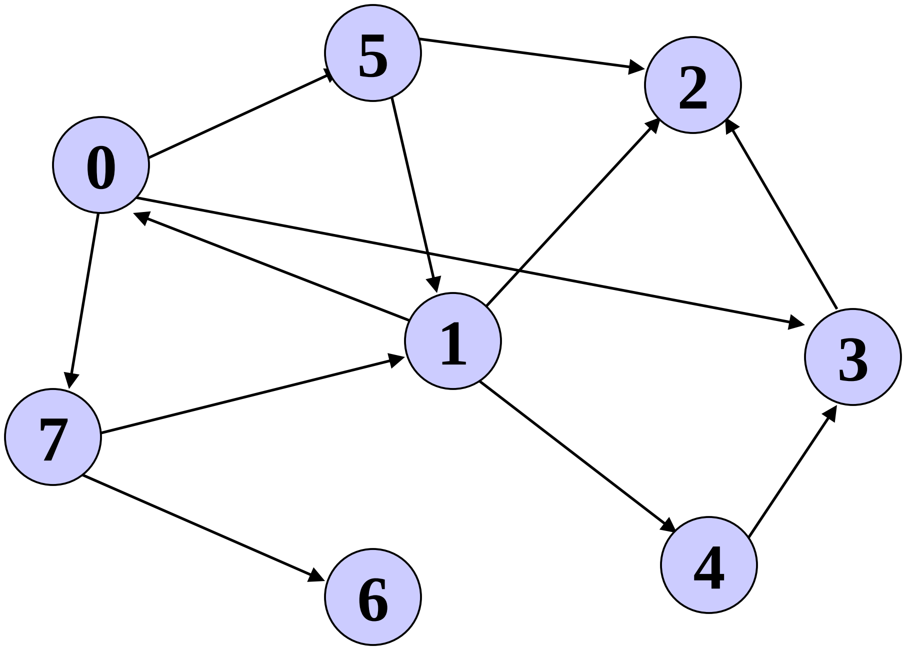
stack

Depth-First Search: Example



DFS: Example2

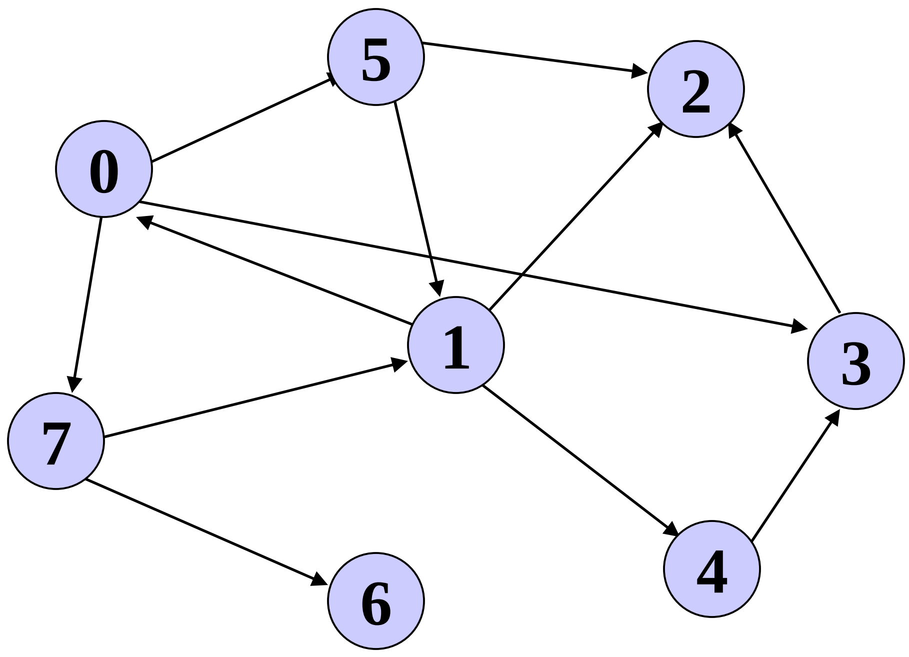
Example



Policy: Visit adjacent nodes in increasing index order

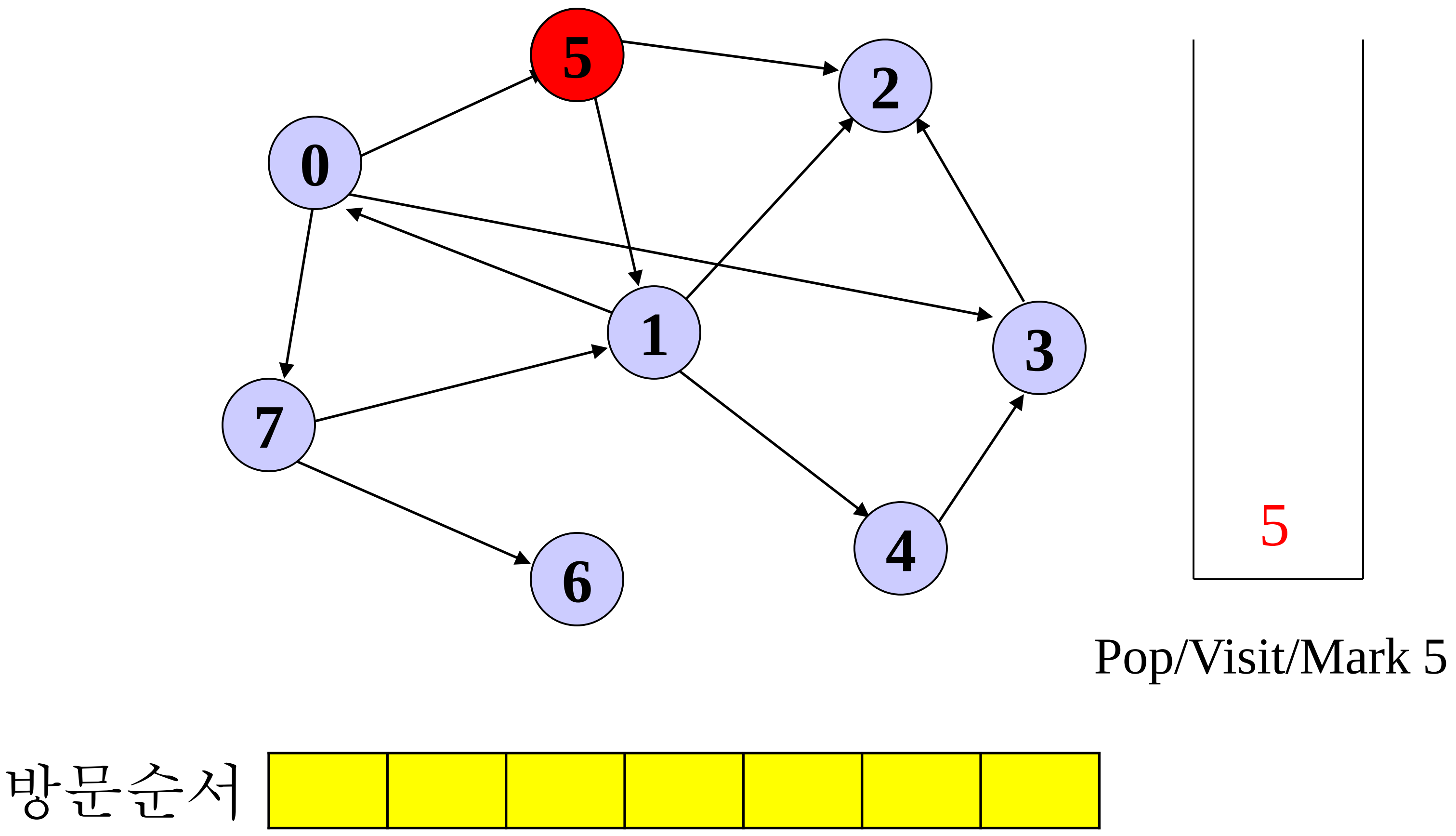
이웃노드를 작은 순으로 방문

Preorder DFS: Start with Node 5
노드5시작점

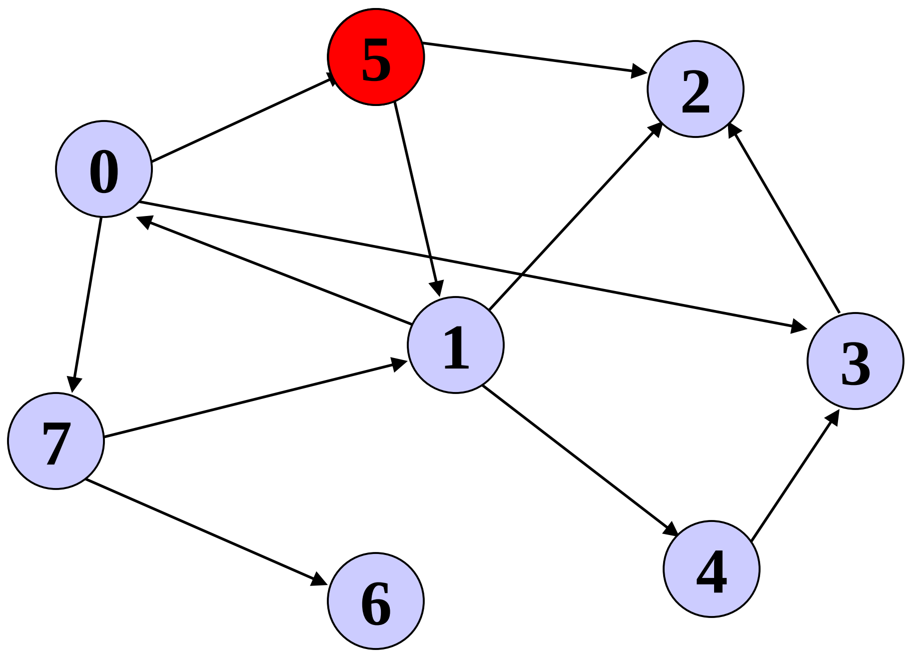


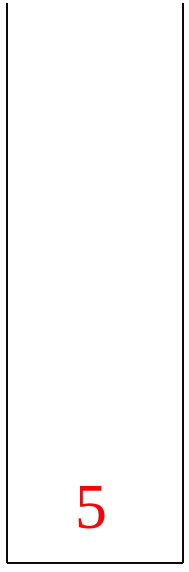
5 1 0 3 2 7 6 4

Preorder DFS: Start with Node 5



Preorder DFS: Start with Node 5

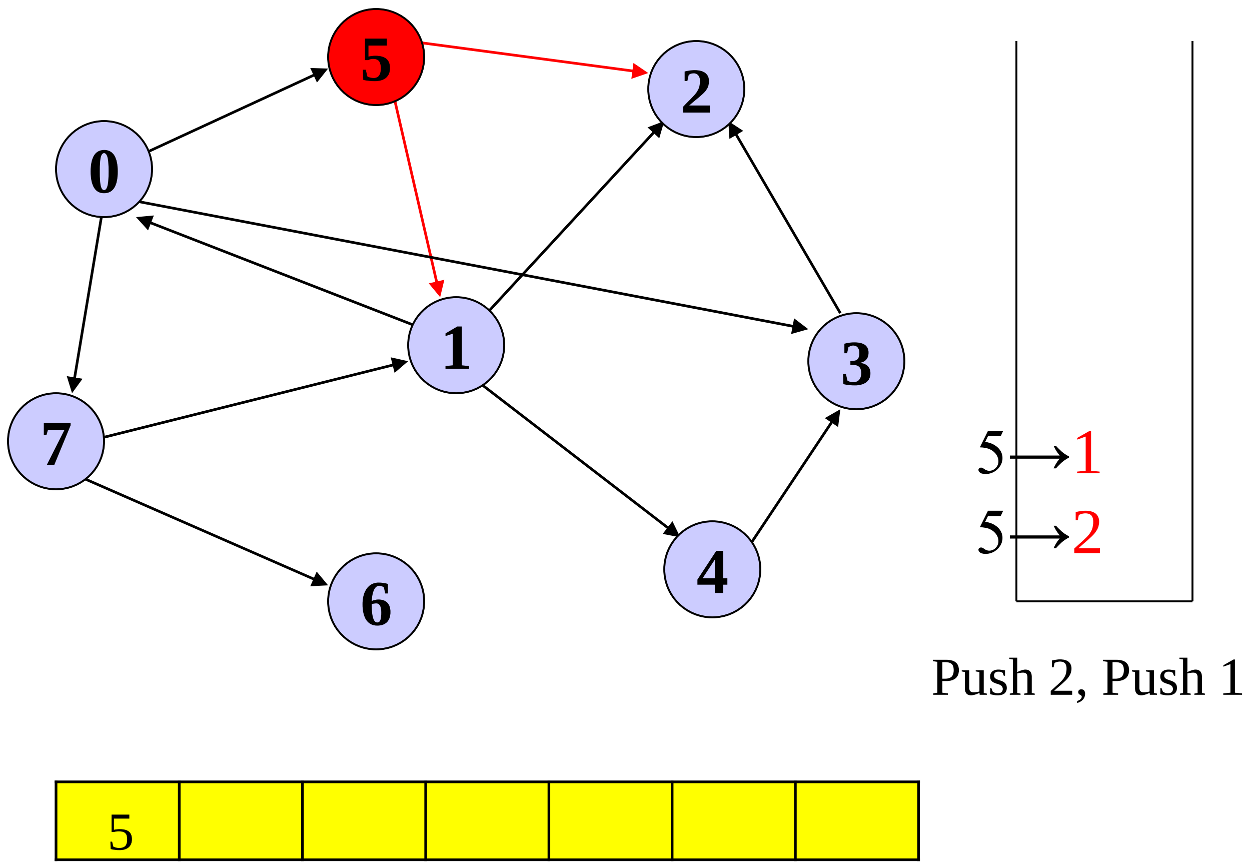




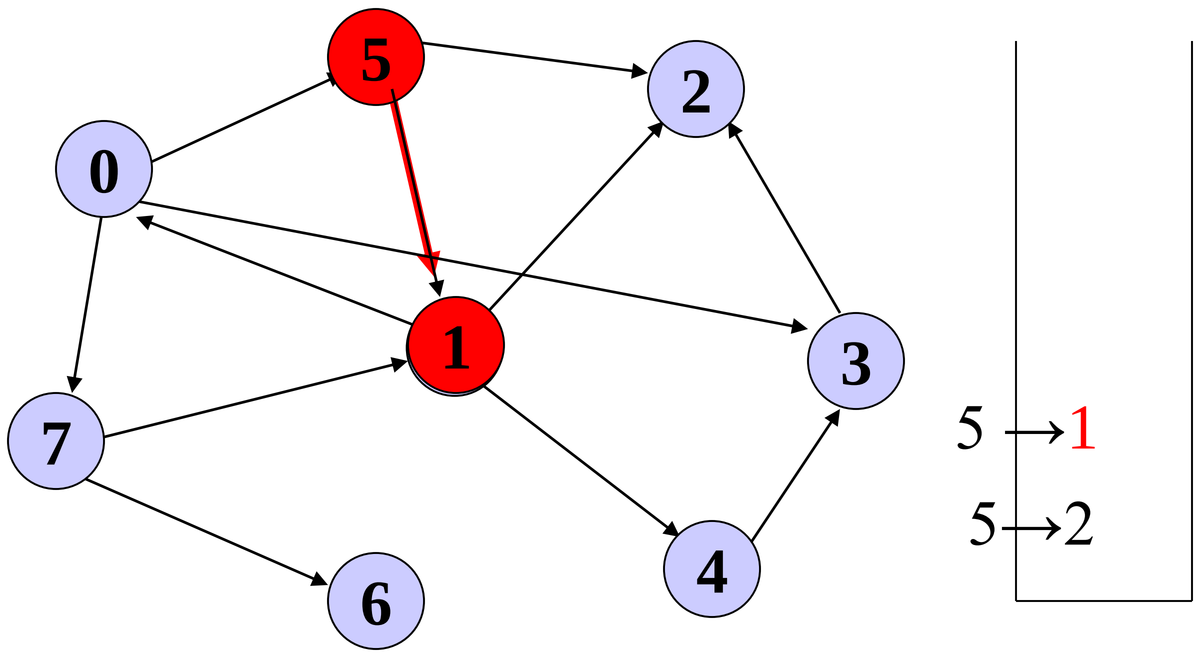
Pop/Visit/Mark 5

방문순서	5						

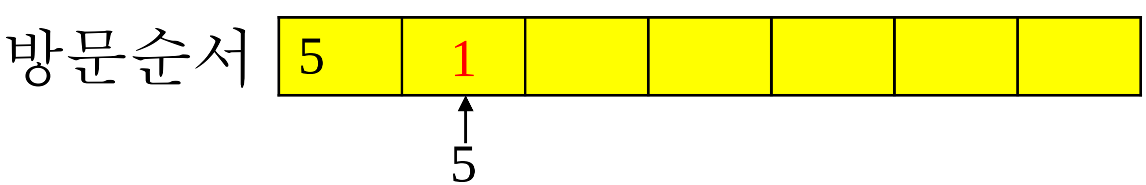
Preorder DFS: Start with Node 5



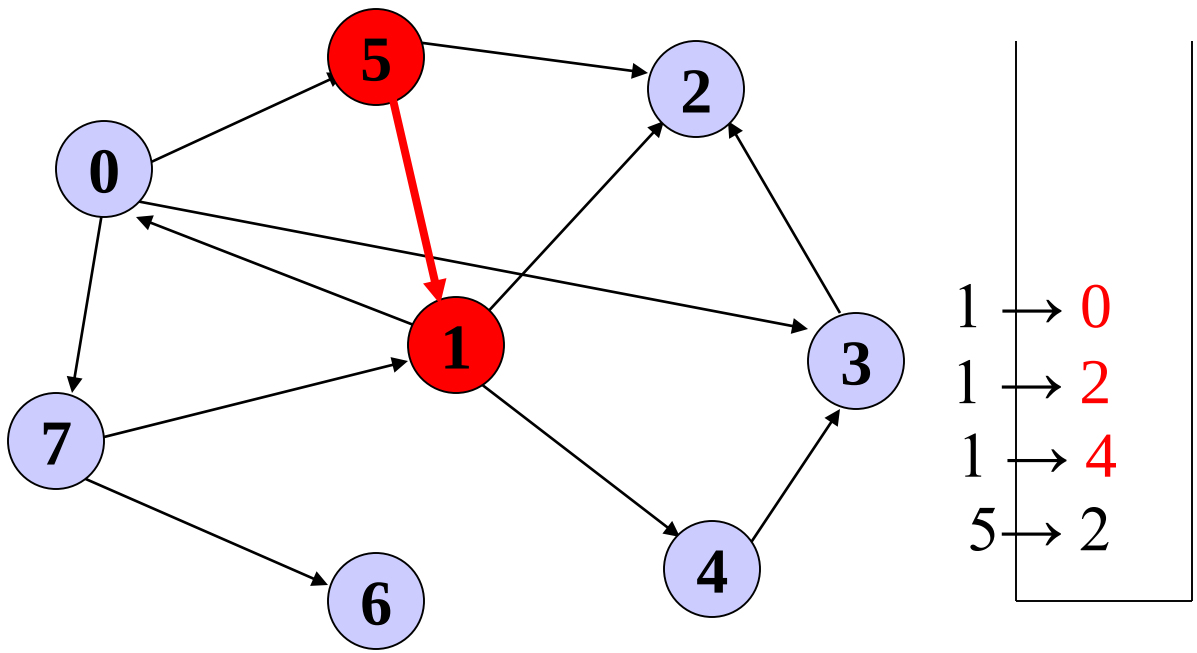
Preorder DFS: Start with Node 5



Pop/Visit/Mark 1



Preorder DFS: Start with Node 5

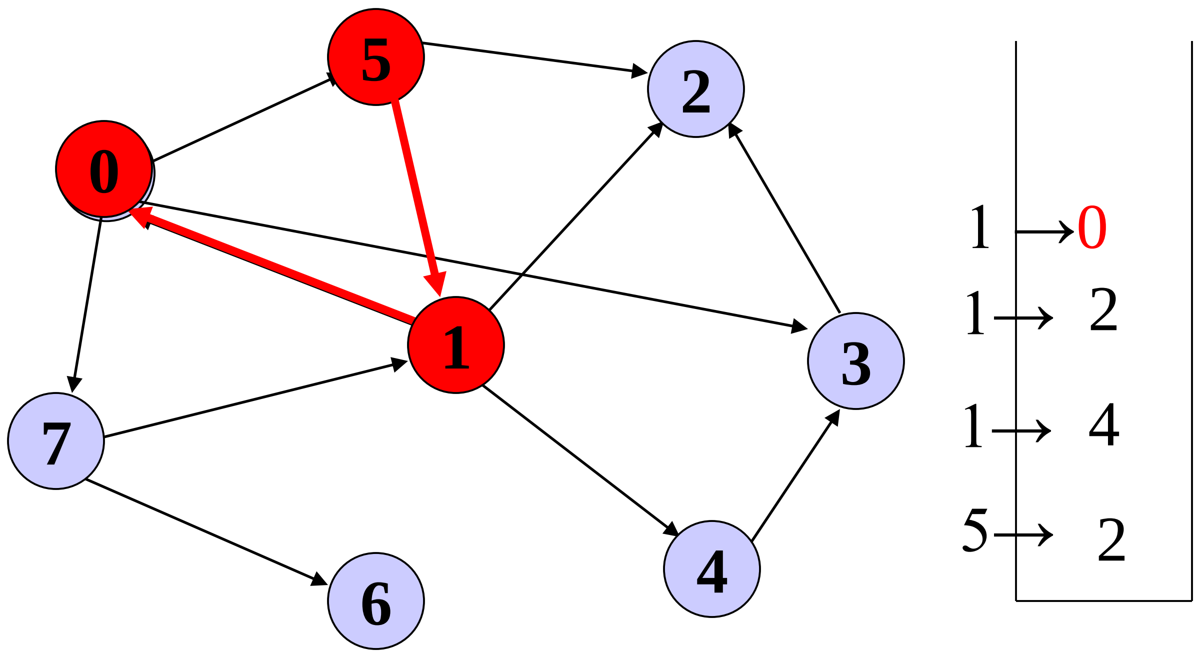


Push 4, Push 2,
Push 0

방문순서	5	1					

5

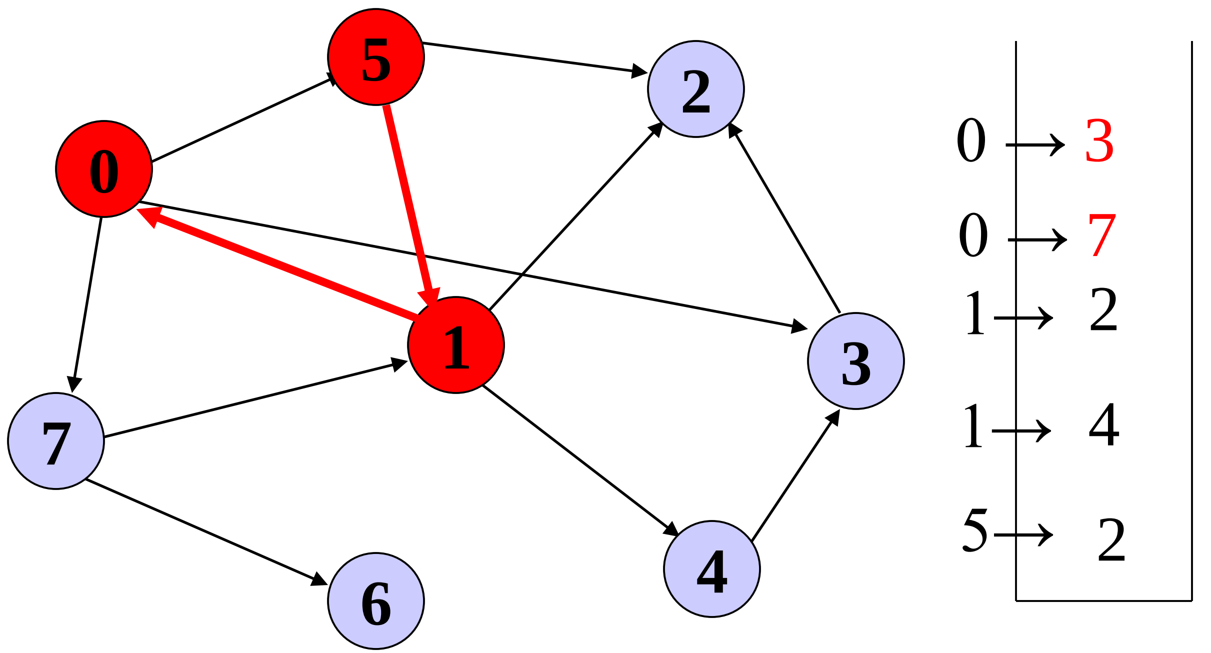
Preorder DFS: Start with Node 5



Pop/Visit/Mark 0

방문순서	5	1	0				
		5	1				

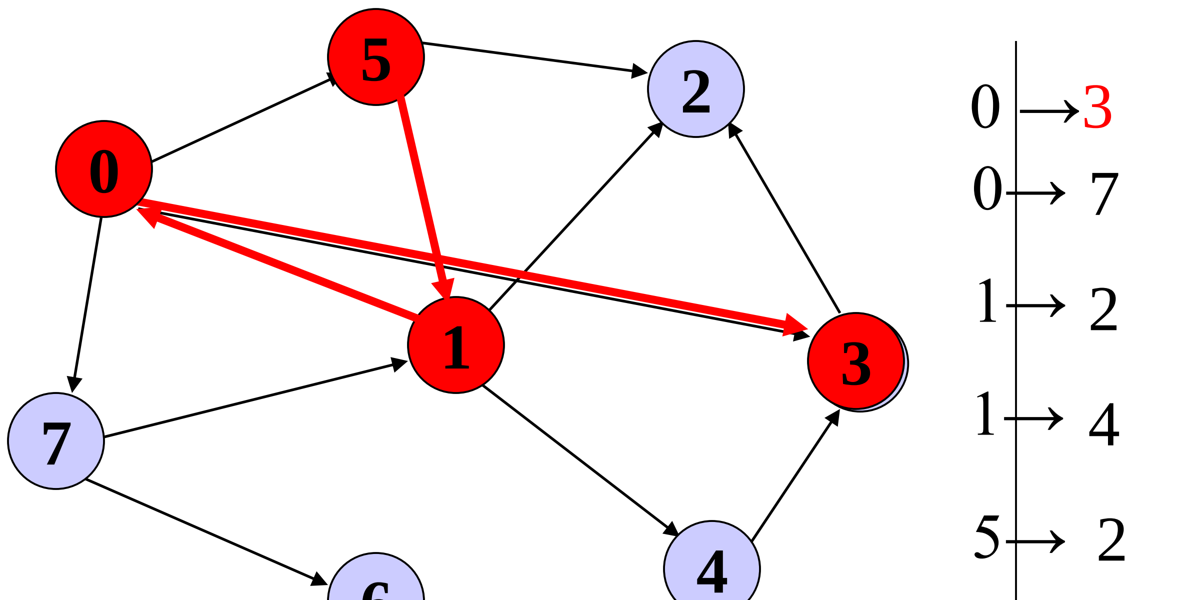
Preorder DFS: Start with Node 5



Push 7, Push 3

방문순서							
	5	1	0				
		5	1				

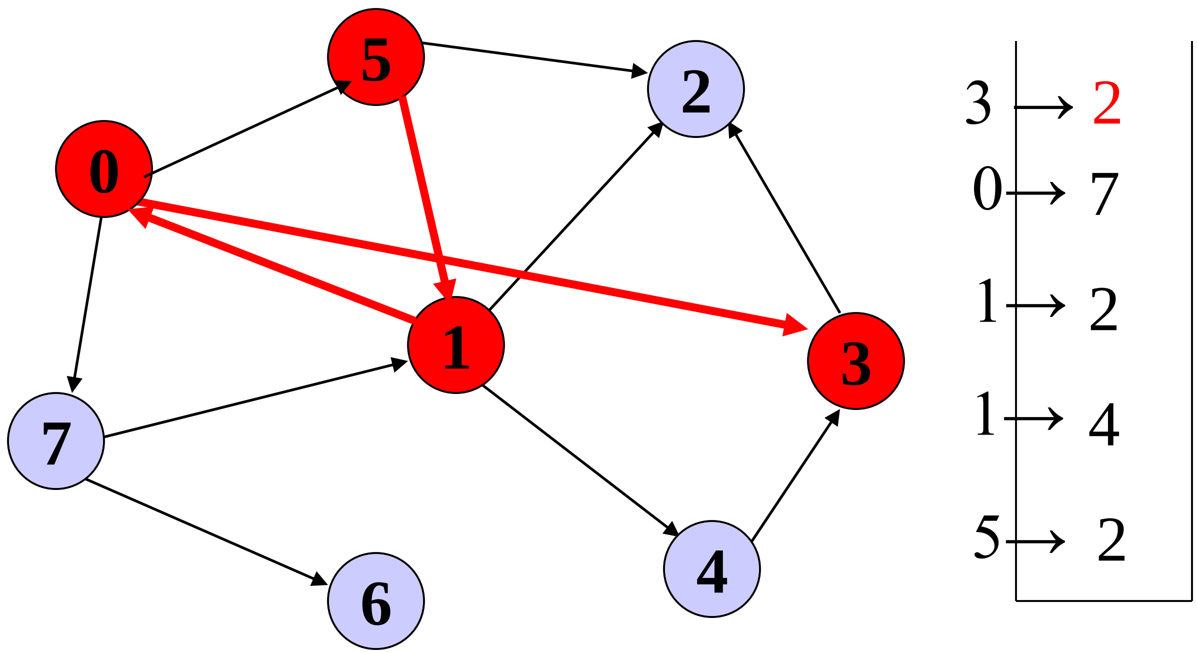
Preorder DFS: Start with Node 5



Pop/Visit/Mark 3

방문순서	5	1	0	3			
		5	1	0			

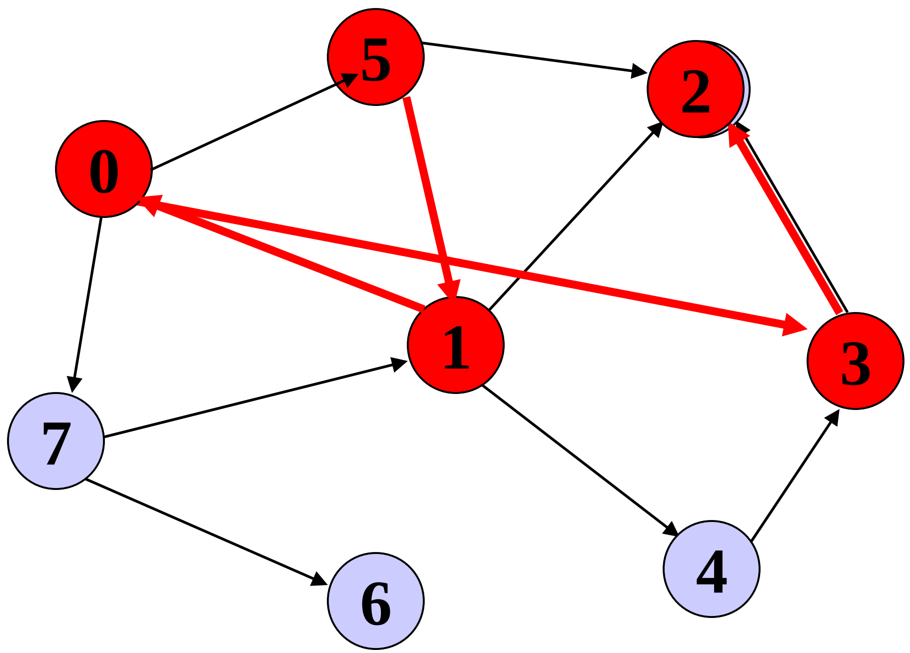
Preorder DFS: Start with Node 5



Push 2

방문순서	5	1	0	3			
		5	1	0			

Preorder DFS: Start with Node 5



3 → 2
0 → 7
1 → 2
1 → 4
5 → 2

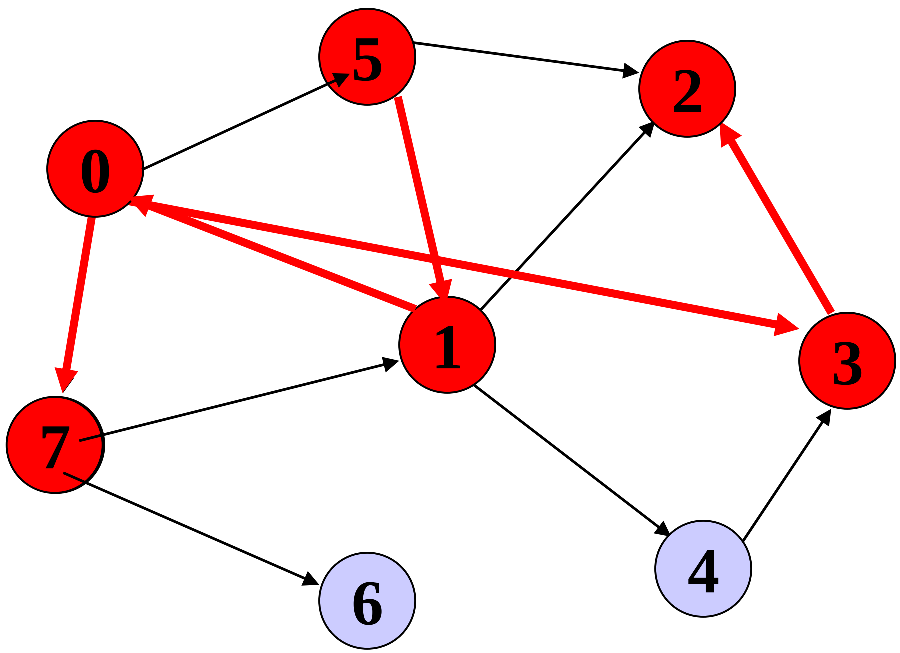
Pop/Mark/Visit 2

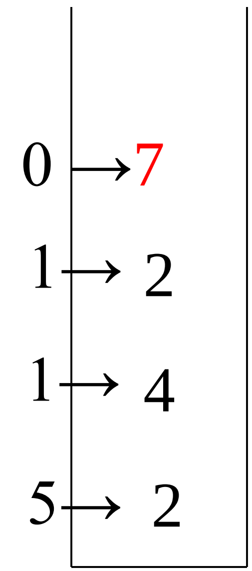
방문순서

5	1	0	3	2		
	5	1	0	3		

Push할 이웃노드없음

Preorder DFS: Start with Node 5





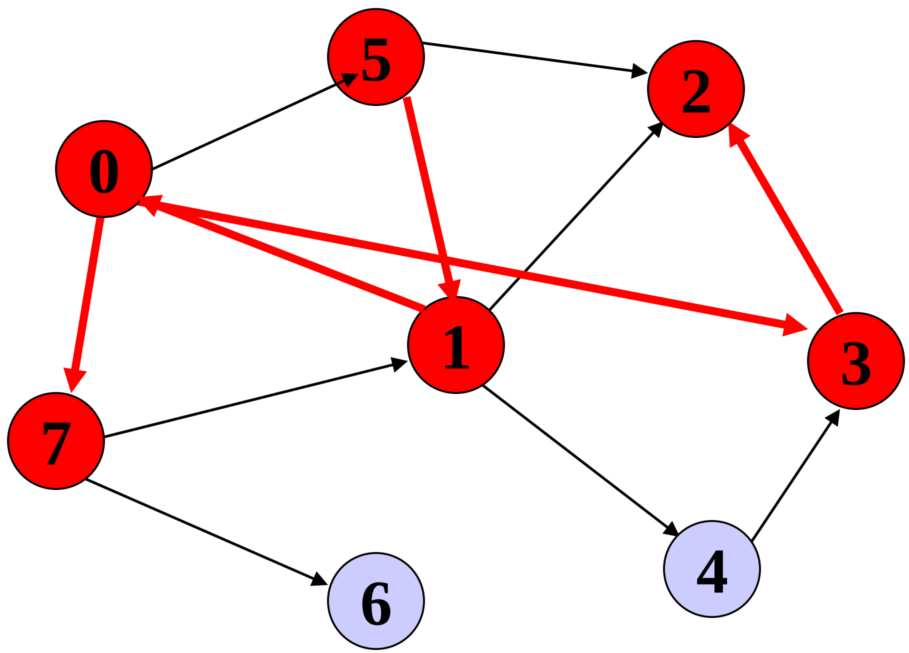
Pop/Mark/Visit 7

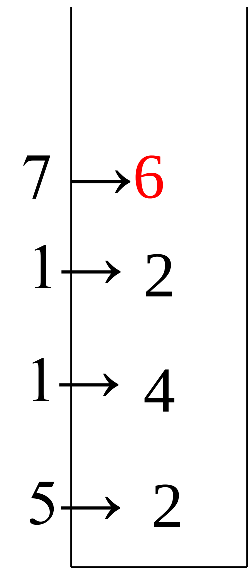
방문순서

5	1	0	3	2	7	
	5	1	0	3	0	

5 1 0 3 2 7

Preorder DFS: Start with Node 5

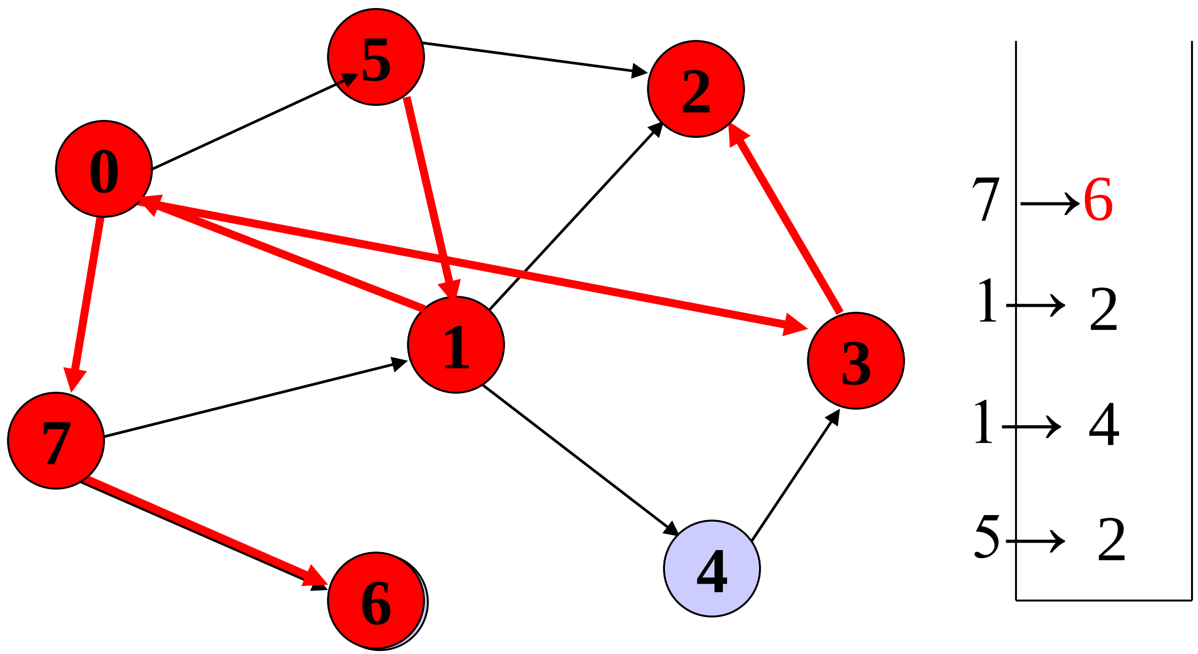




Push 6

방문순서	5	1	0	3	2	7	
		5	1	0	3	0	

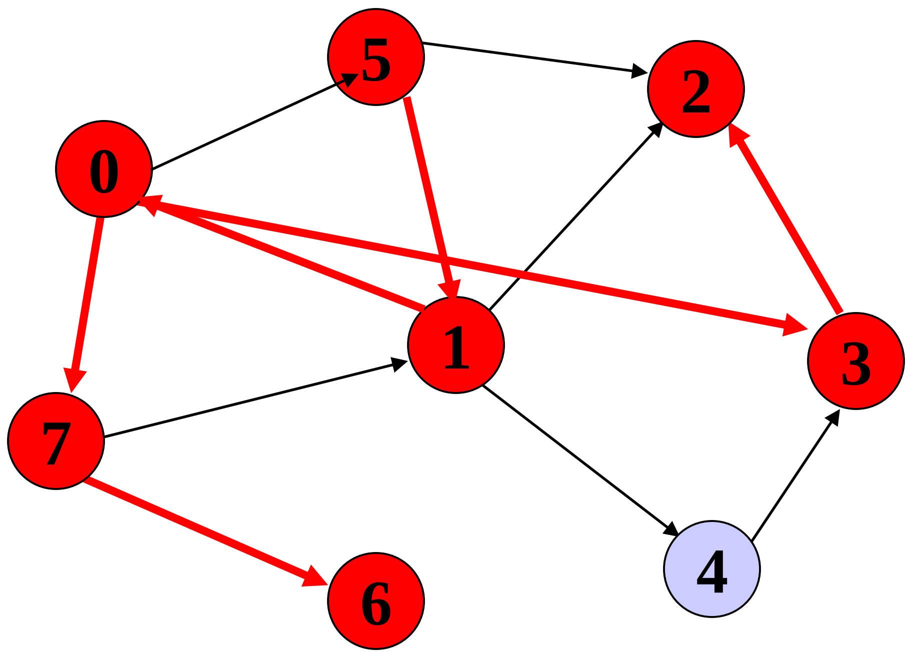
Preorder DFS: Start with Node 5

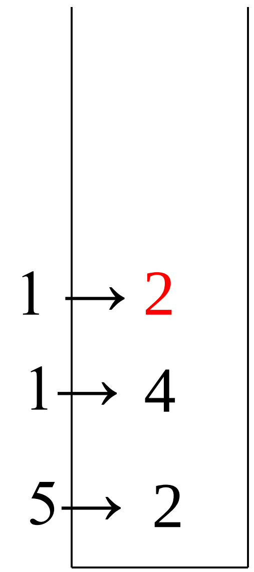


Pop/Mark/Visit 6

방문순서	5	1	0	3	2	7	6
		↑ 5	↑ 1	↑ 0	↑ 3	↑ 0	↑ 7

Preorder DFS: Start with Node 5

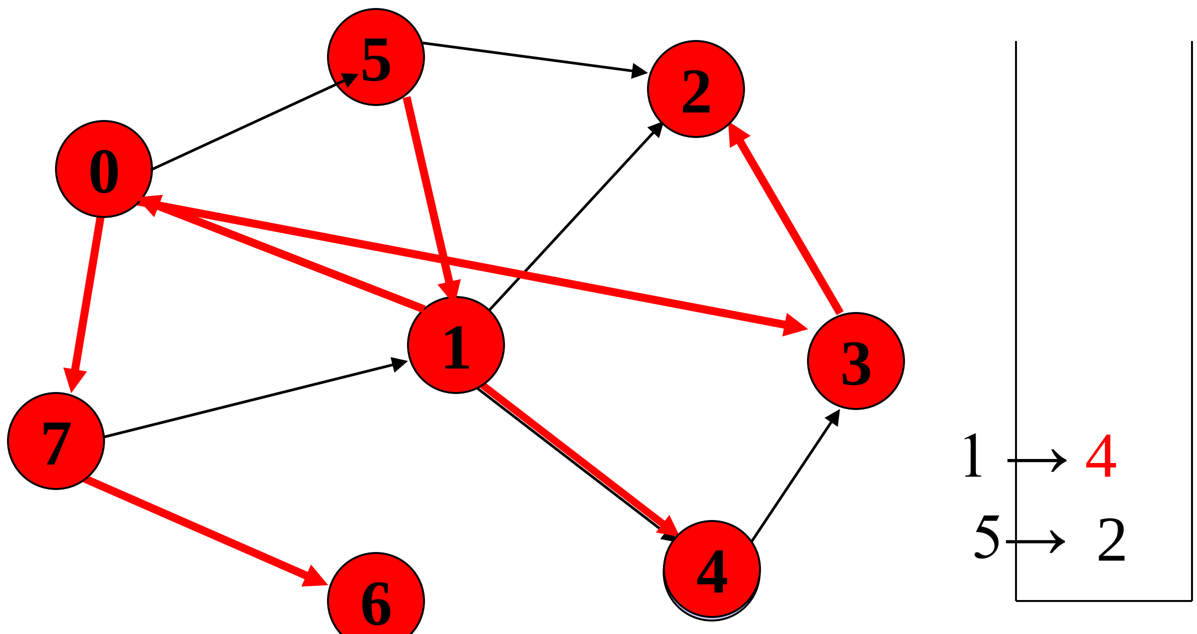




Pop (don’t visit) 2

방문순서	5	1	0	3	2	7	6
		5	1	0	3	0	7

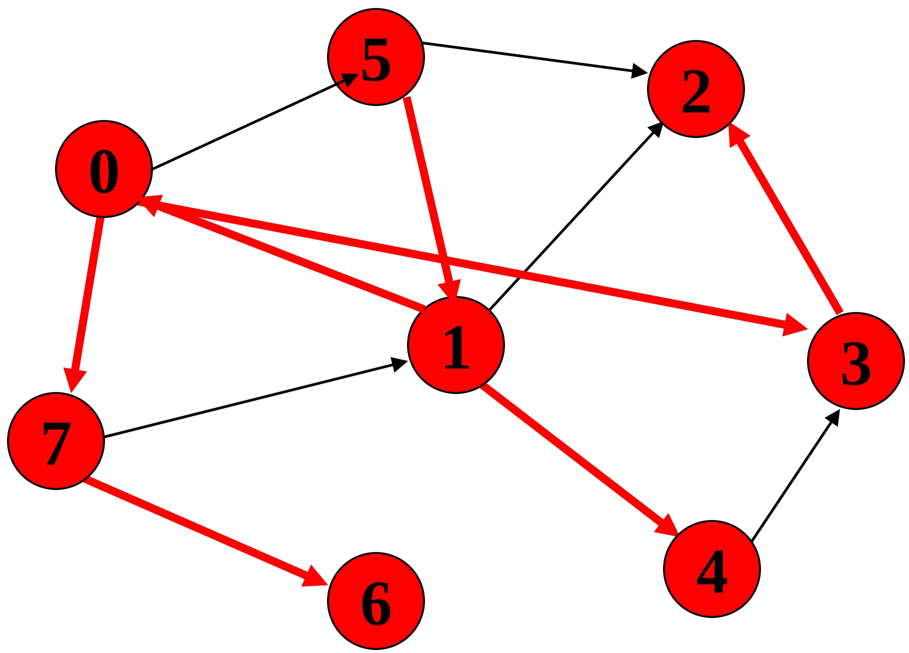
Preorder DFS: Start with Node 5

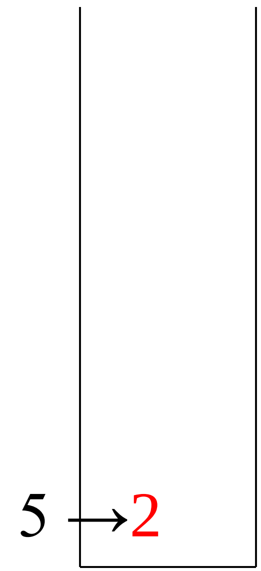


Pop/Mark/Visit 4

방문순서	5	1	0	3	2	7	6	4
		5	1	0	3	0	7	1

Preorder DFS: Start with Node 5

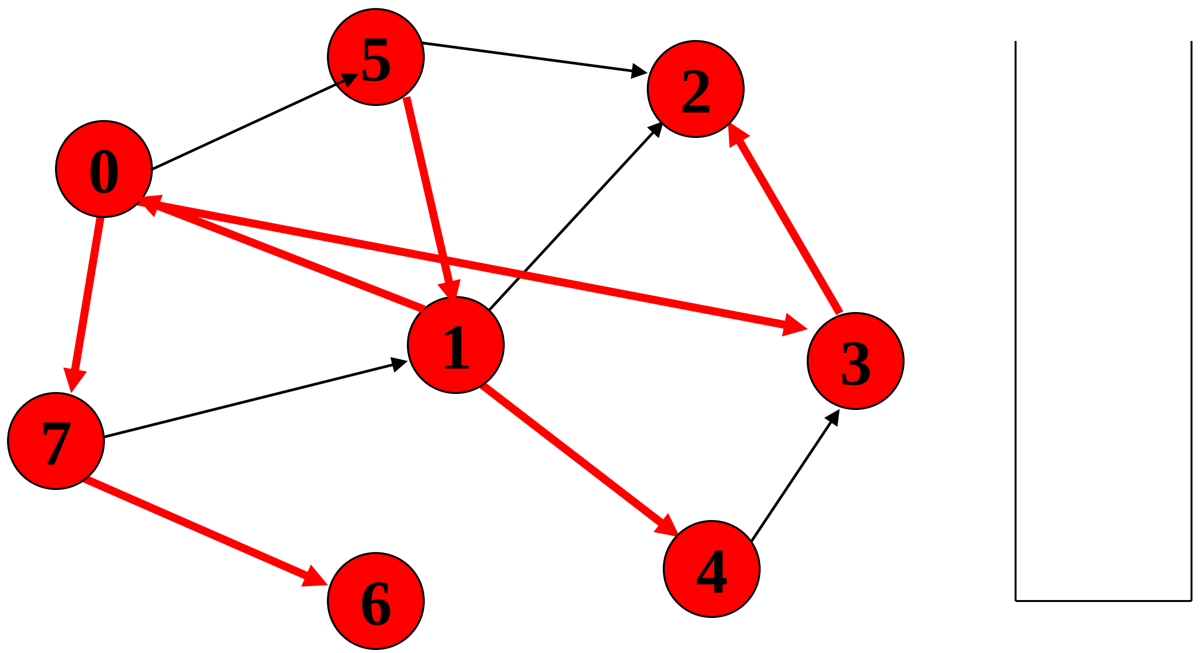




Pop (don't visit) 2

방문순서	5	1	0	3	2	7	6	4
		5	1	0	3	0	7	1

Preorder DFS: Start with Node 5



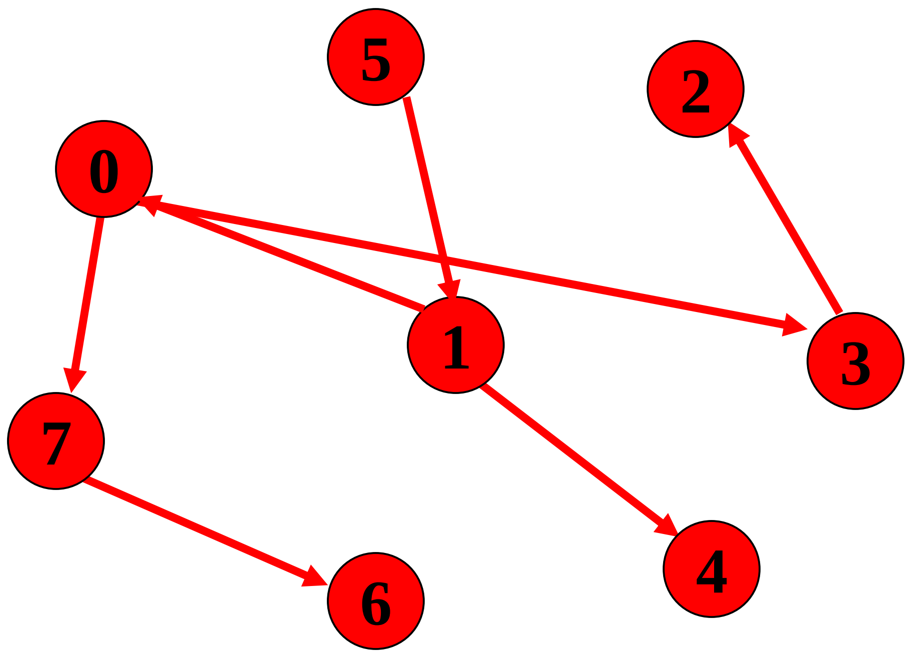
Done

5 1 0 3 2 7 6 4

5 1 0 3 0 7 1

Spanning Tree(간선트리)

DFS방식으로 그래프의 노드를 방문했을때 얻게되는 트리를 간선트리라한다



5 1 0 3 2 7 6 4

5 1 0 3 2 7 6 4

5 1 0 3 0 7 1

5 1 0 3 2 7 6 4

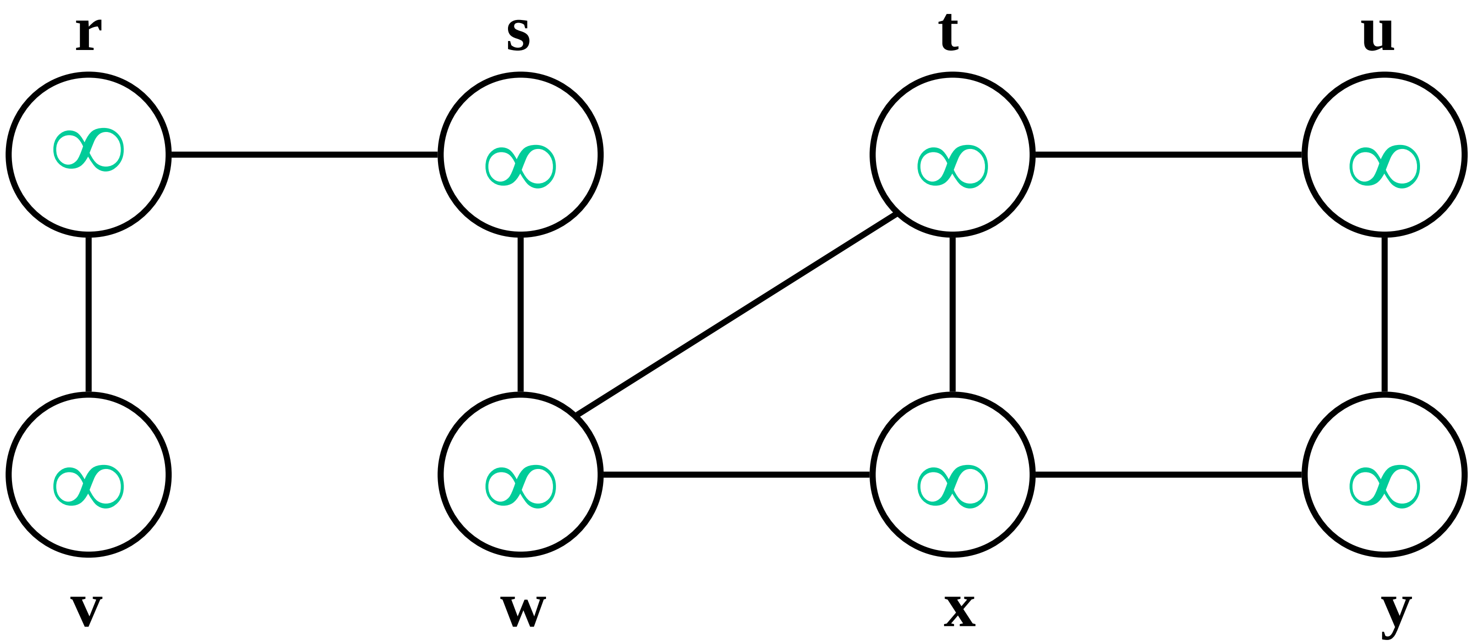
BFS방식으로 노드순회를 마친 결과

Breadth-first Search너비우선탐색

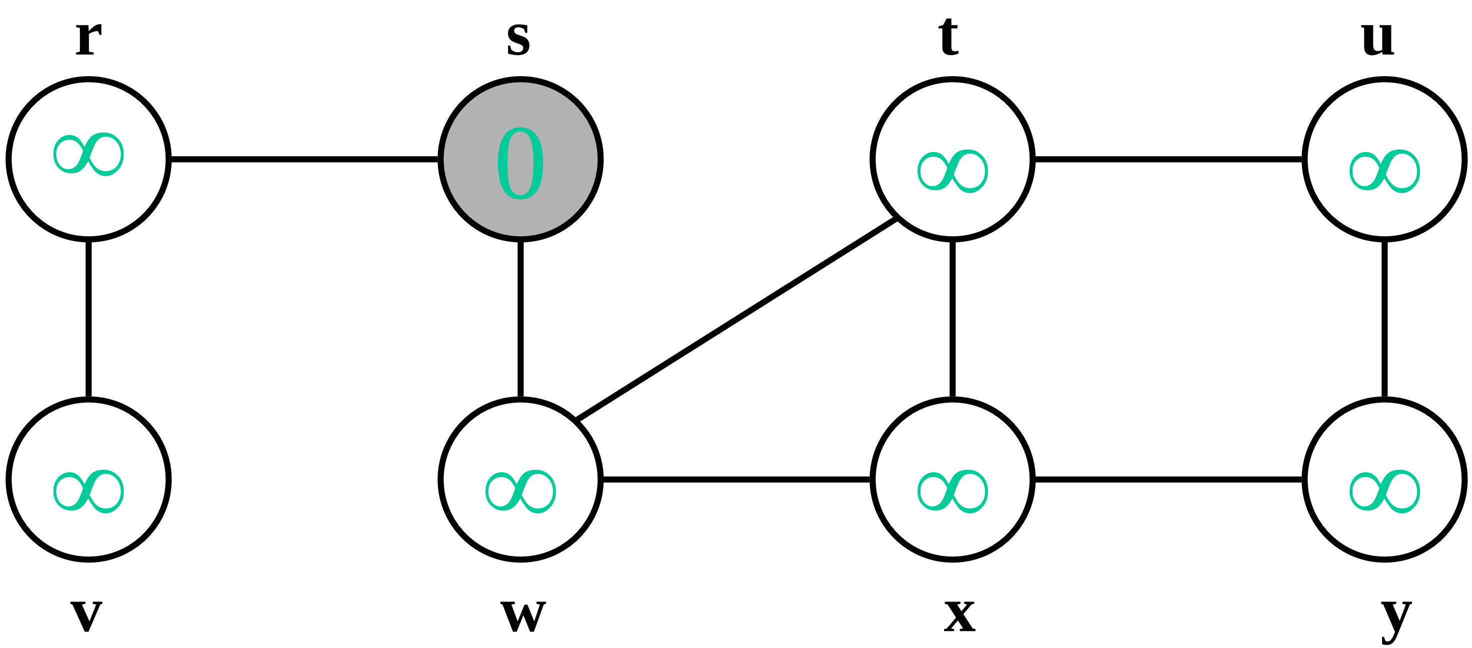
- Ripples in a pond연못의 동심원을 생각.
- Visit designated node
- Then visited unvisited nodes a distance *i* away, where *i* = 1, 2, 3, etc.
- For nodes the same distance away, visit nodes in systematic manner (eg. increasing index order)출발점에서 같은 거리만큼 떨어진 정점들을 모두 방문

BFS(너비우선탐색): Example1

Breadth-First Search: Example

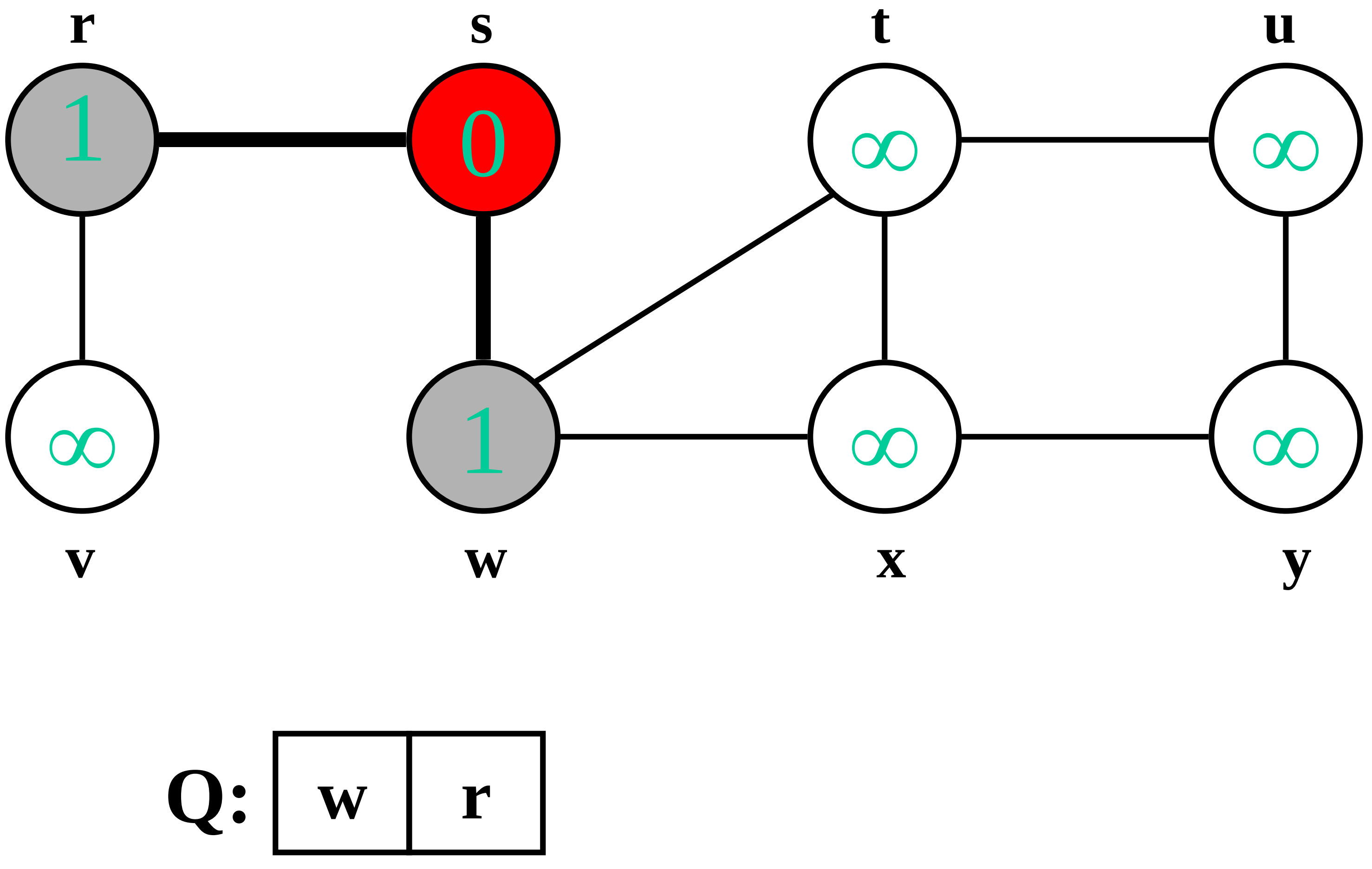


Breadth-First Search: Example

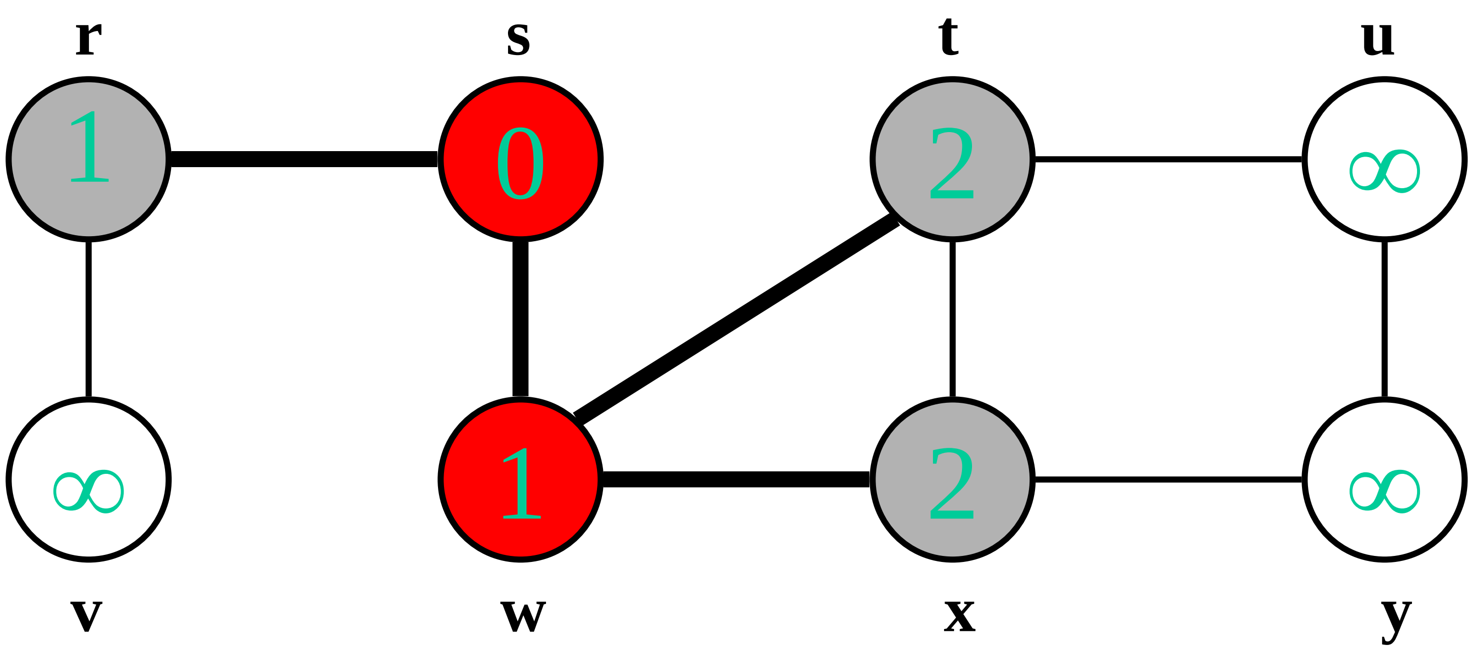


Q: s

Breadth-First Search: Example

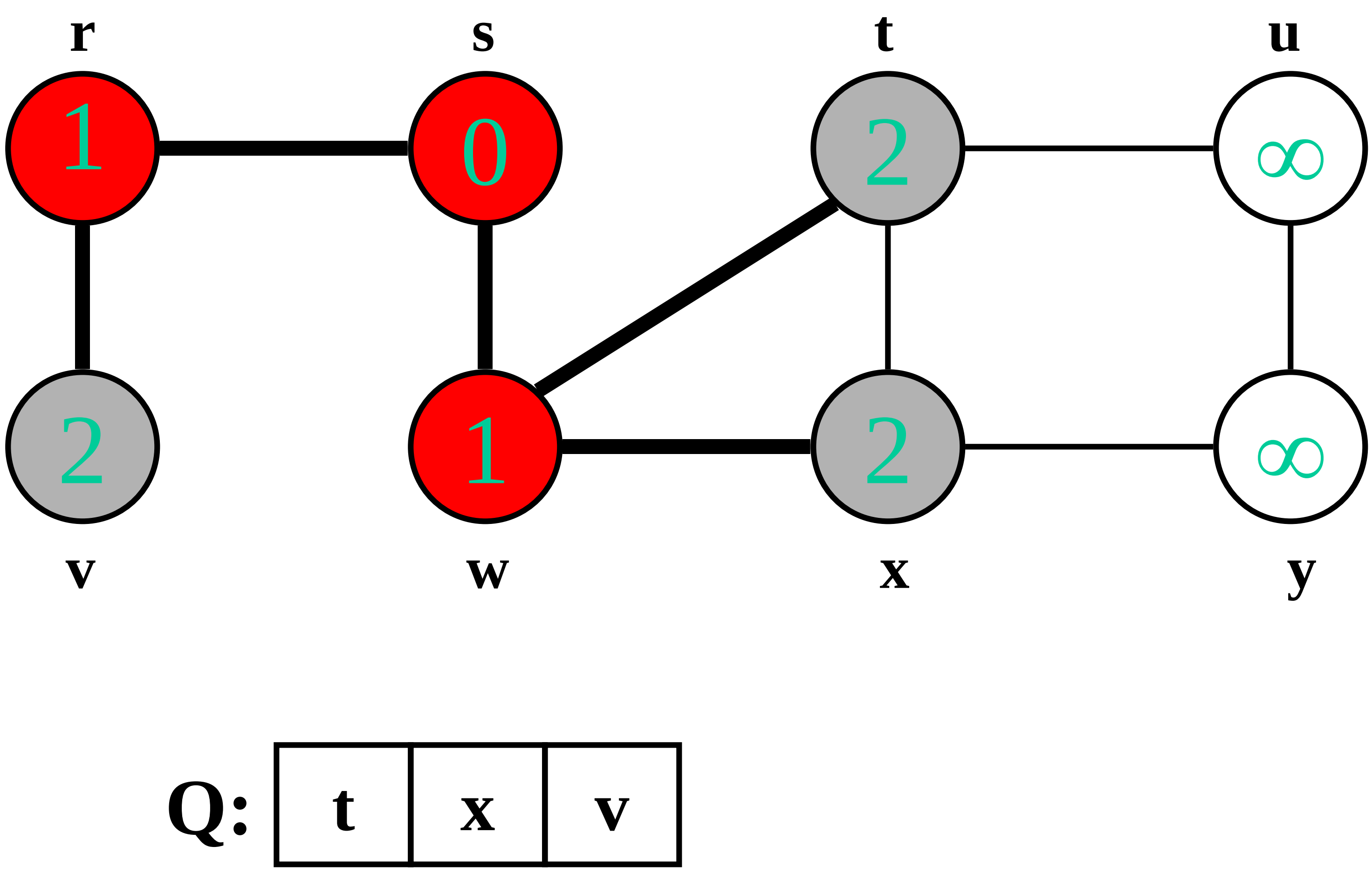


Breadth-First Search: Example

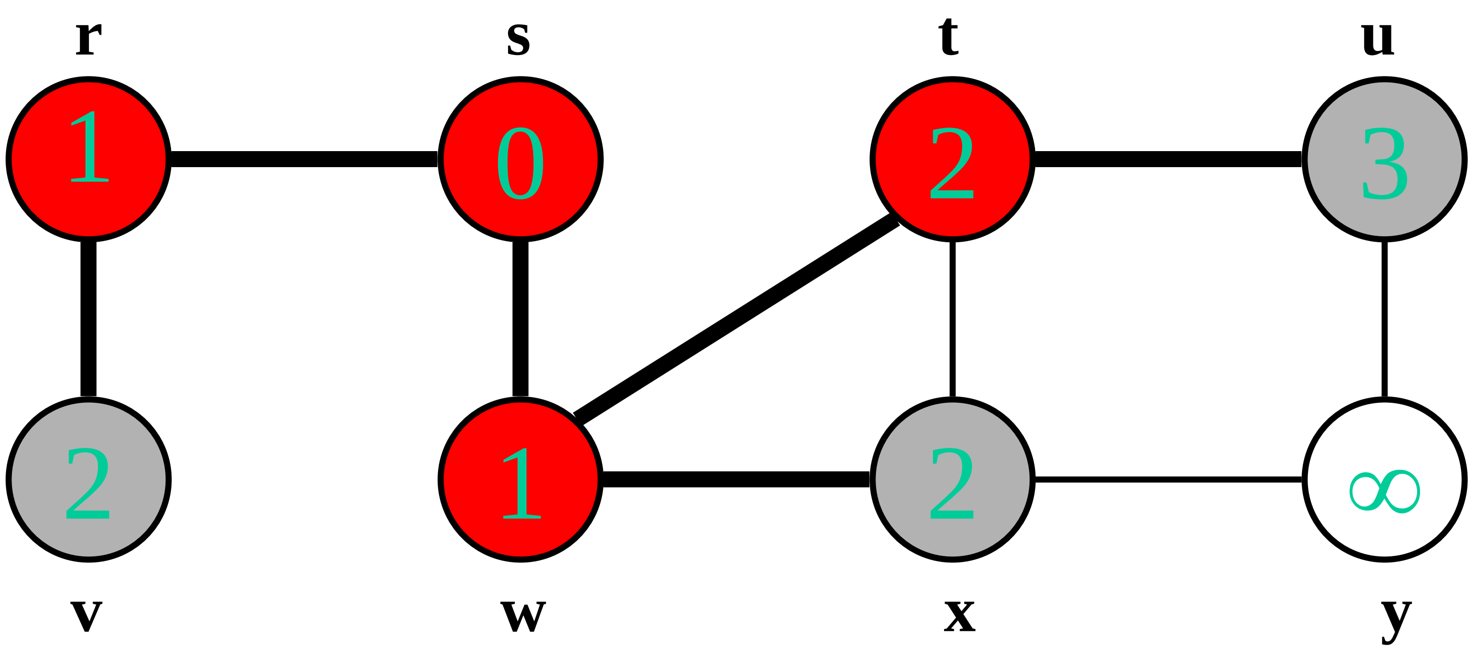


Q: | r | t | x |

Breadth-First Search: Example



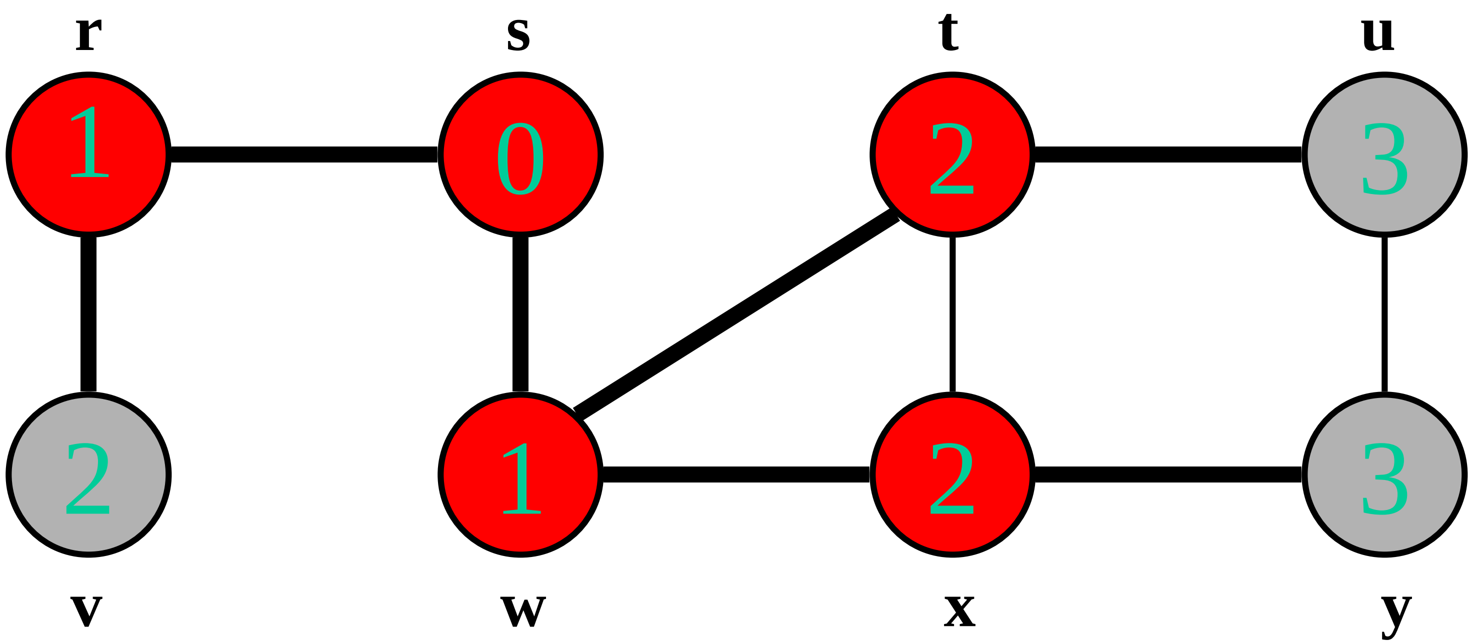
Breadth-First Search: Example



Q:

x	**v**	**u**

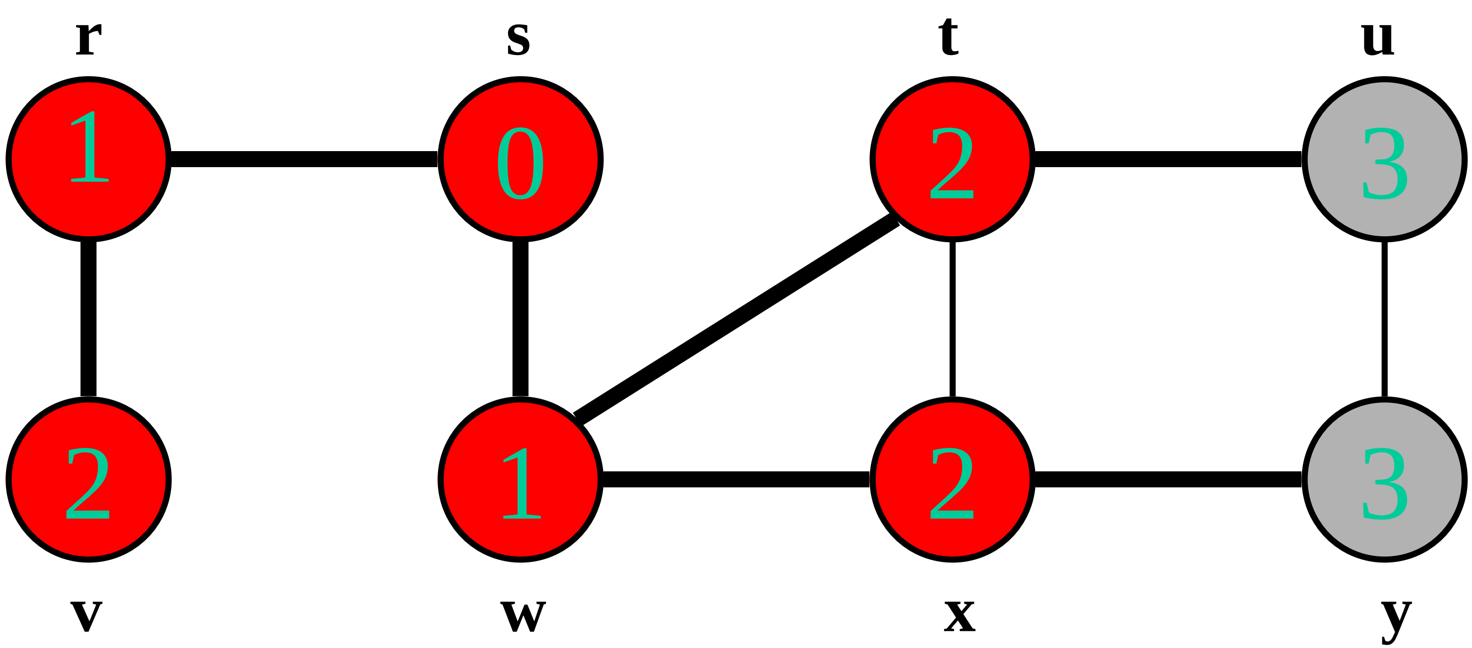
Breadth-First Search: Example



Q:

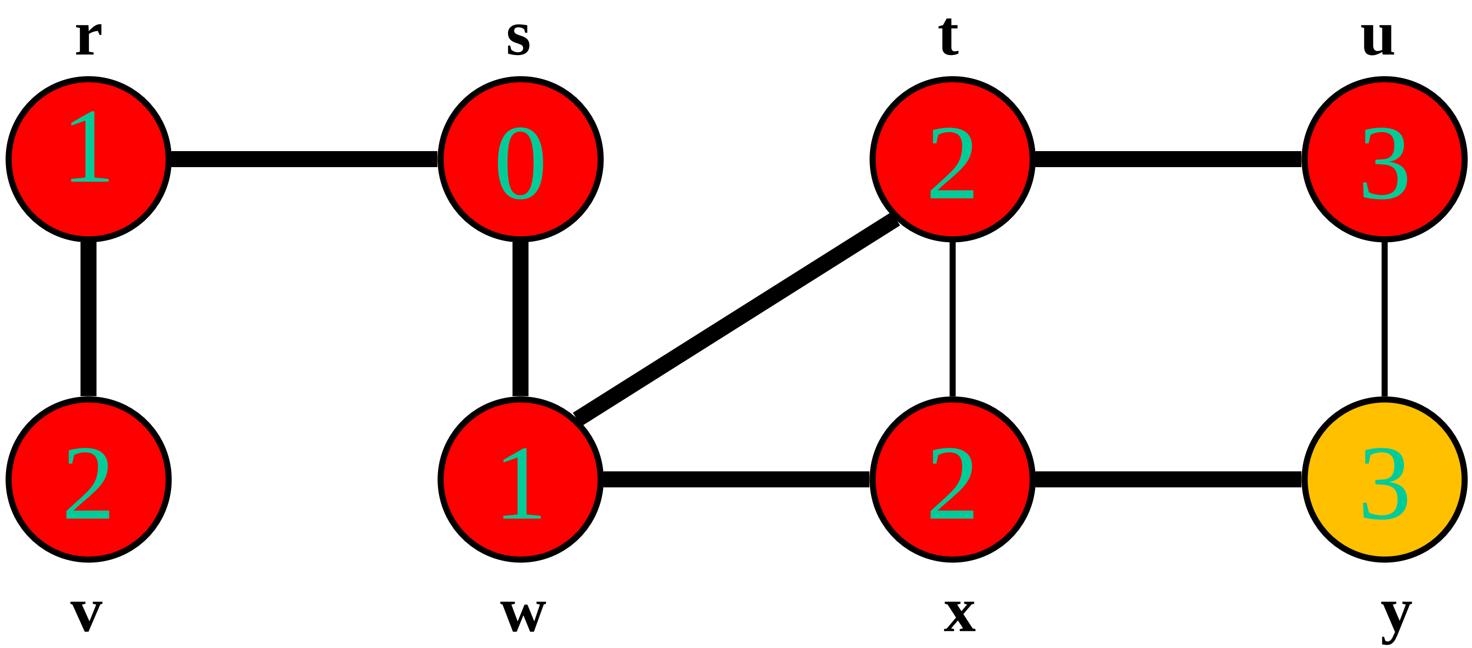
v	u	y

Breadth-First Search: Example



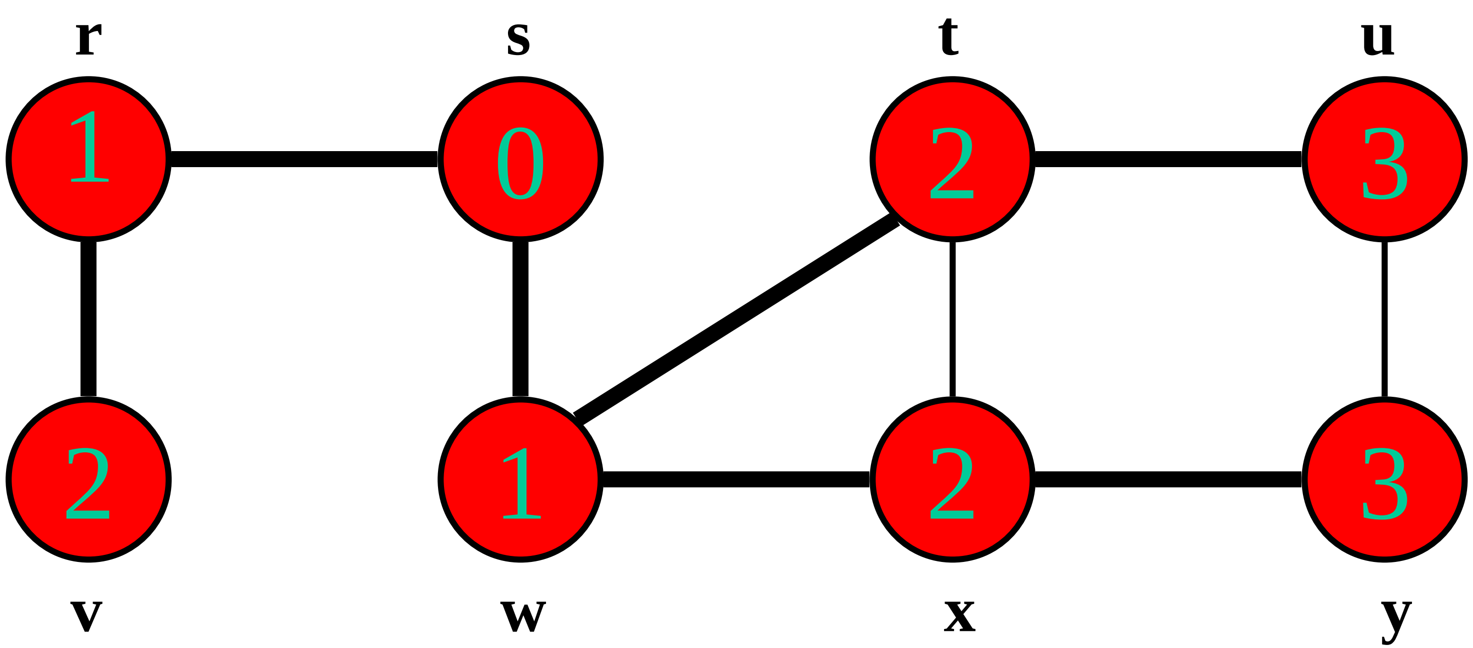
Q: | u | y |

Breadth-First Search: Example



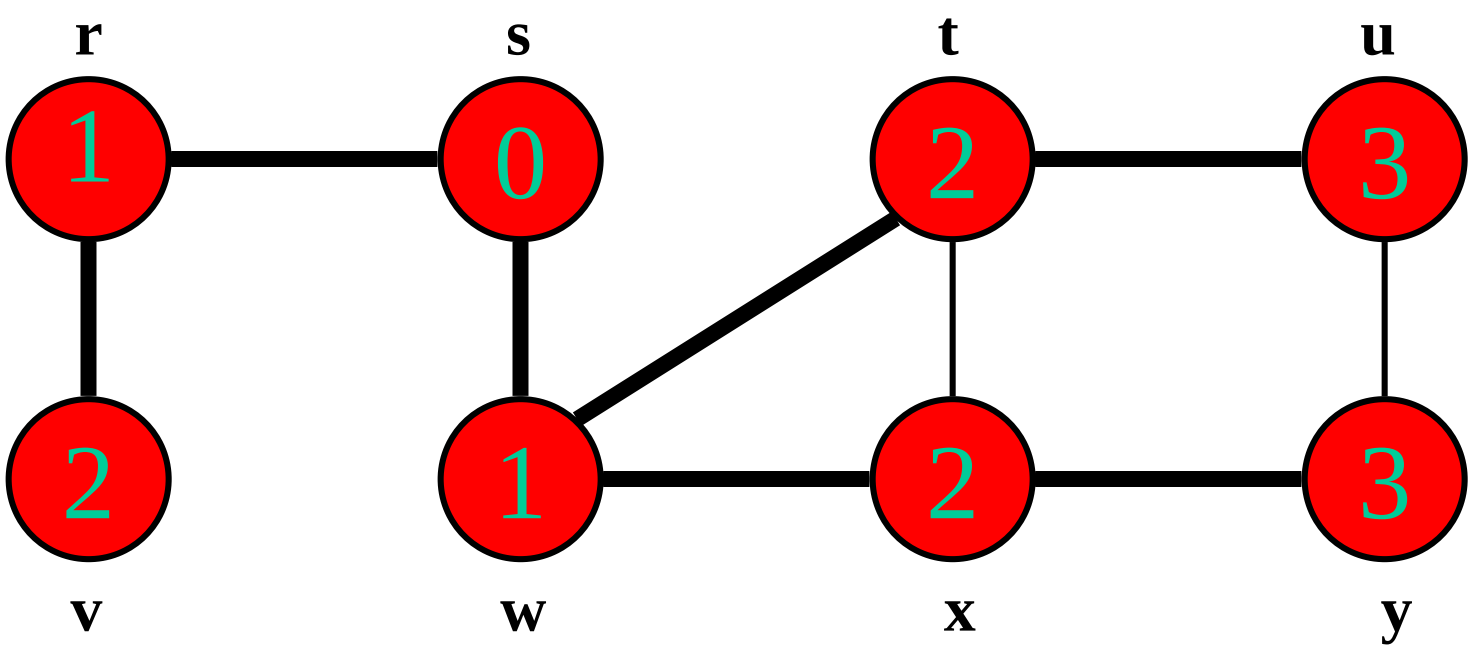
Q: y

Breadth-First Search: Example



Q: ∅

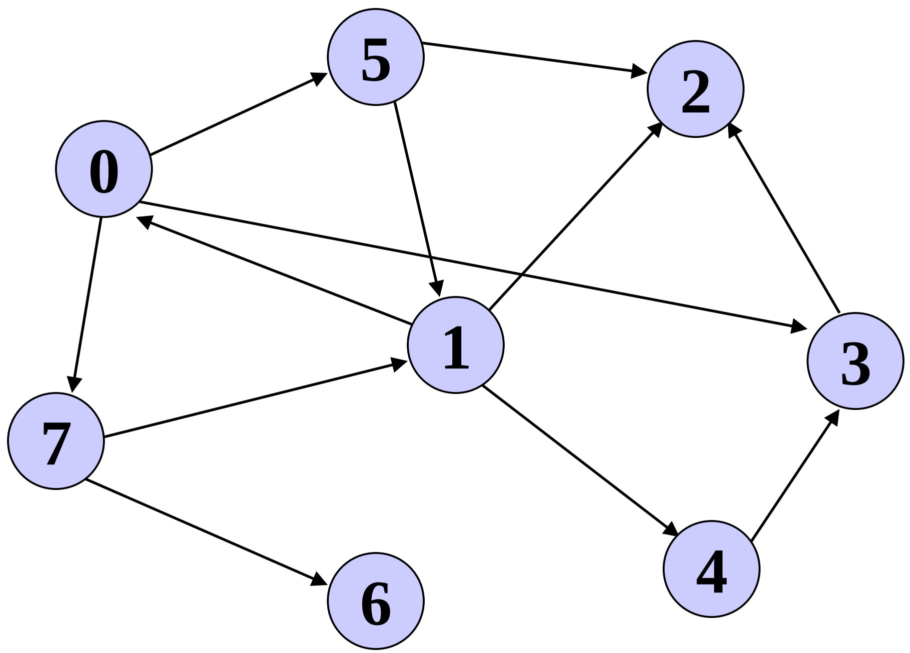
Breadth-First Search: Example



Q: Ø

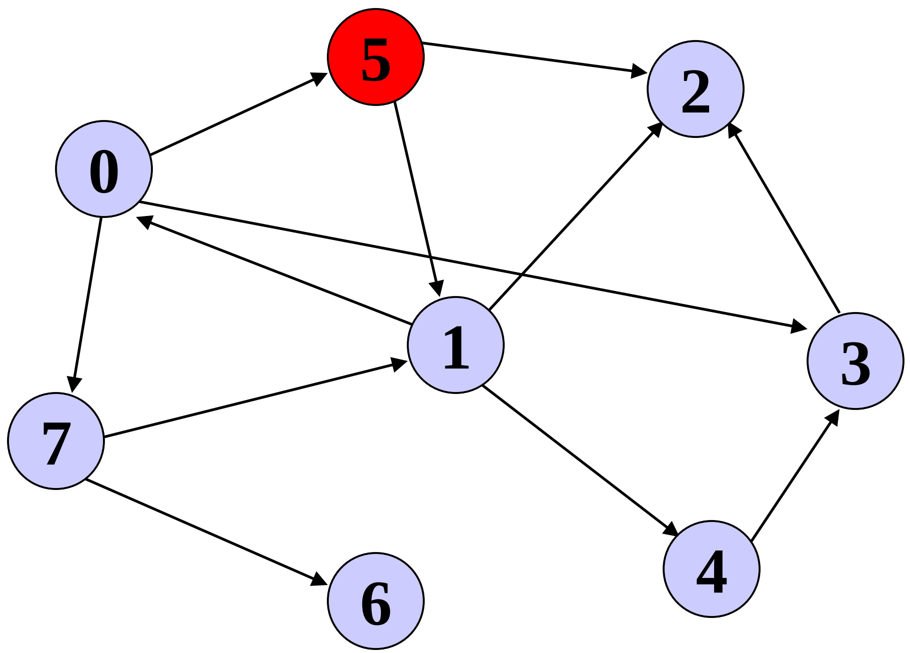
BFS(너비우선탐색): Example2

BFS: Start with Node 5



5 1 2 0 4 3 7 6

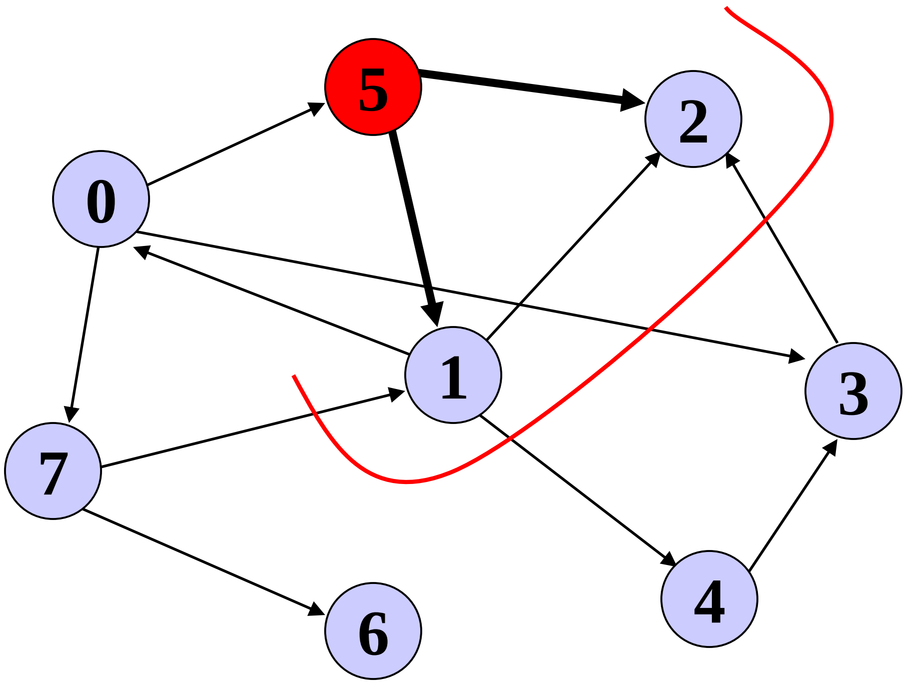
BFS: Start with Node 5



방문순서 5

Queue

BFS: Node one-away

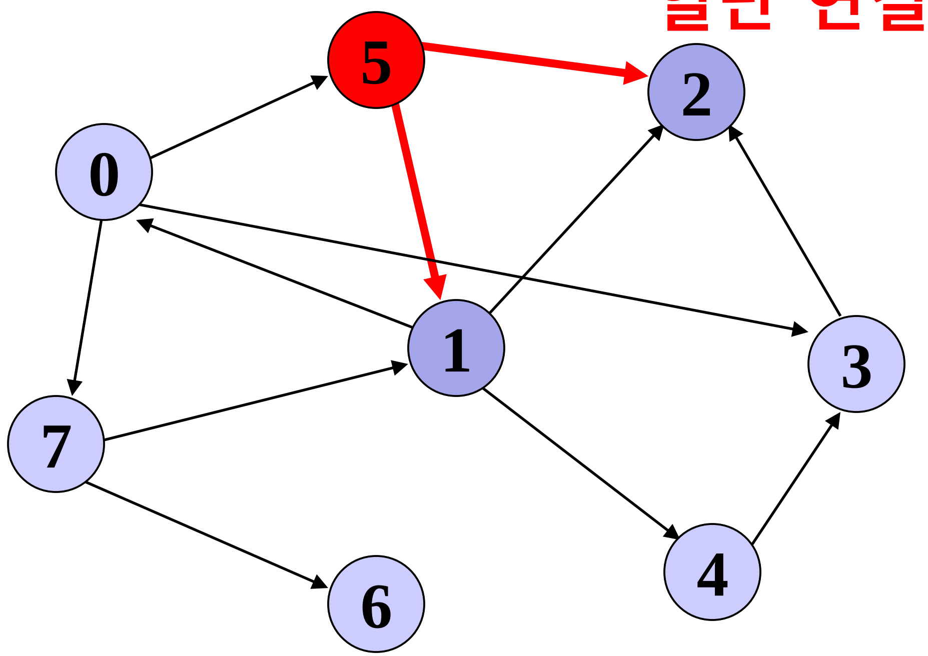


5

Queue

BFS: Visit 1 and 2

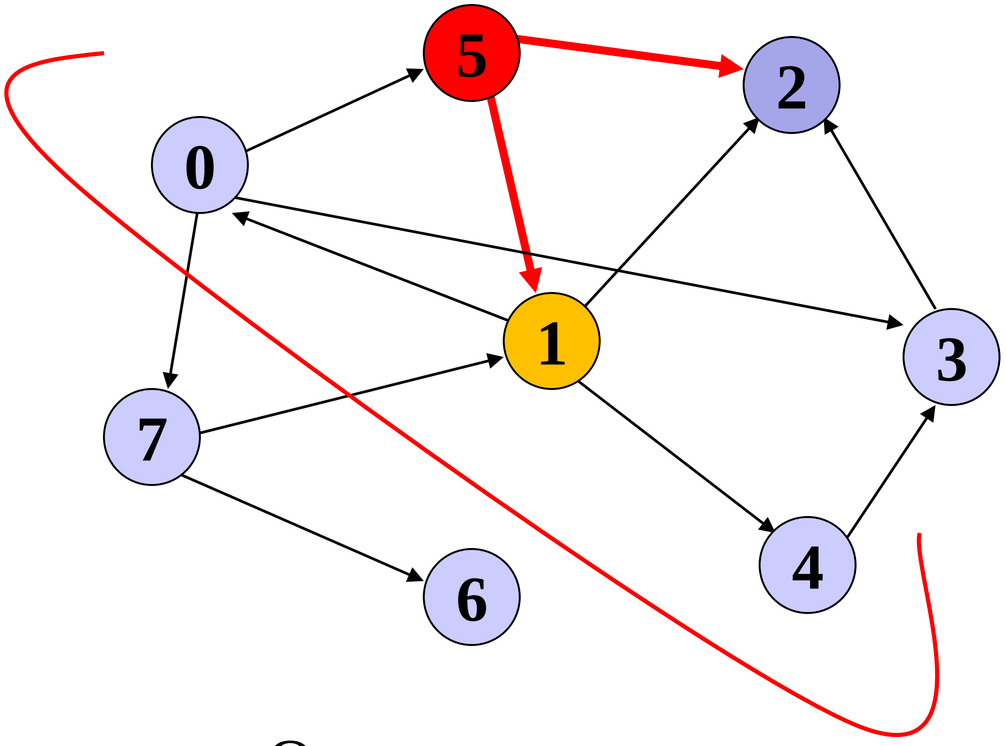
이웃노드 1,2를 빨강색으로
일단 연결



Queue 5 1 2

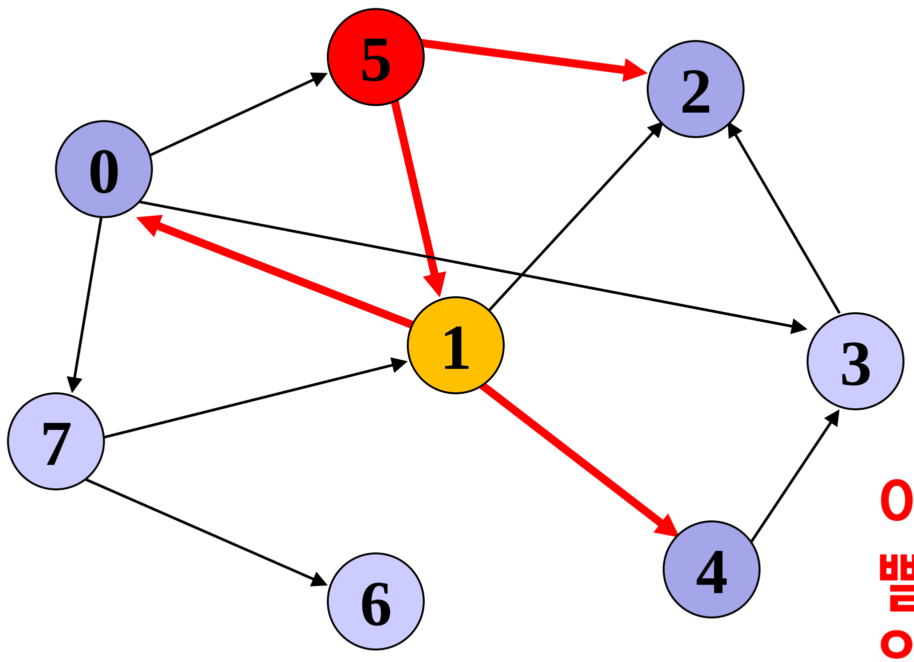
큐에 이웃인 1,2를 삽입
그리고 dequeue

BFS: Nodes two-away



Queue 5 1 2

BFS: Visit 0 and 4

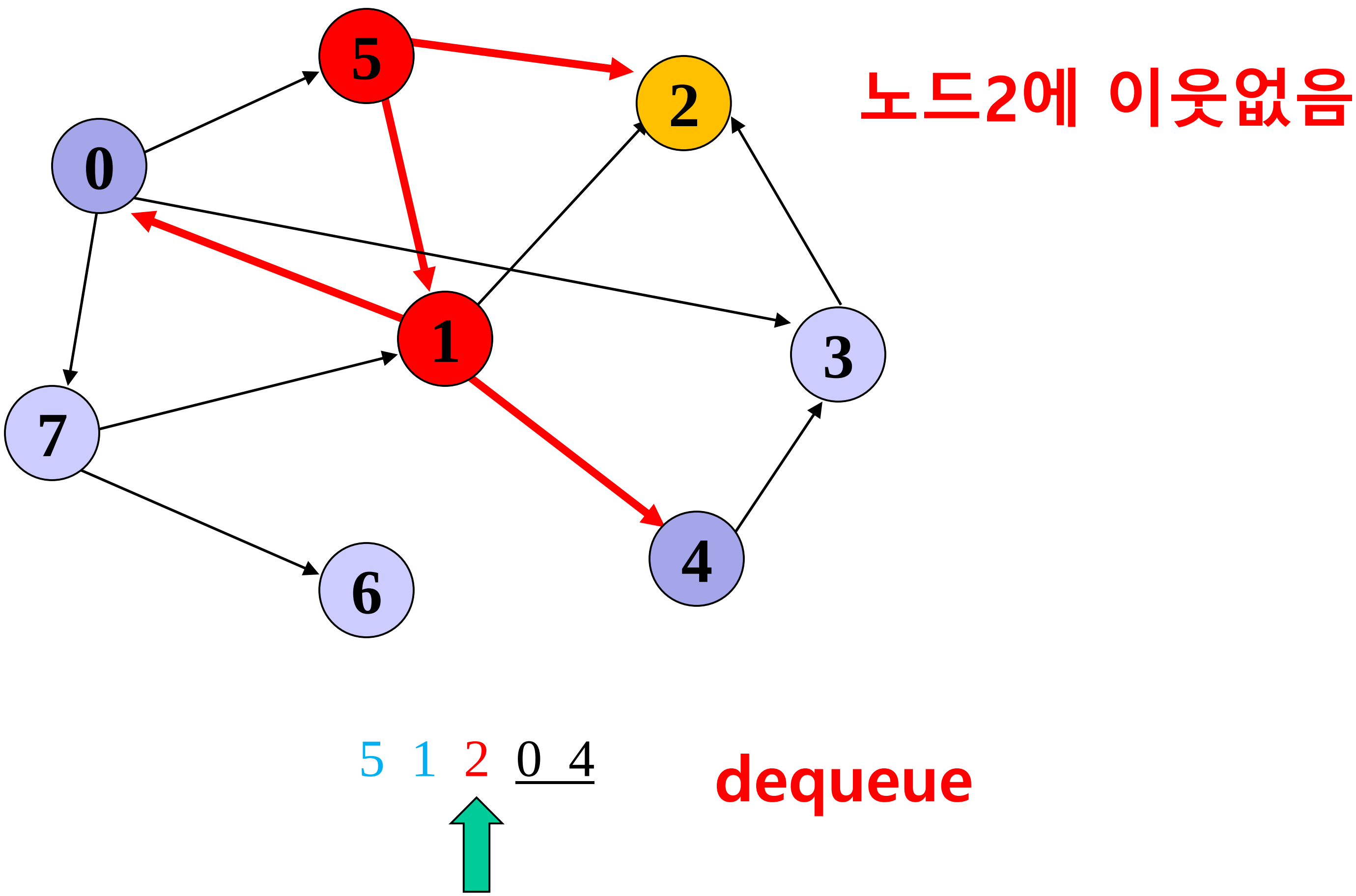


이웃노드 0,4를
빨강색으로
일단 연결

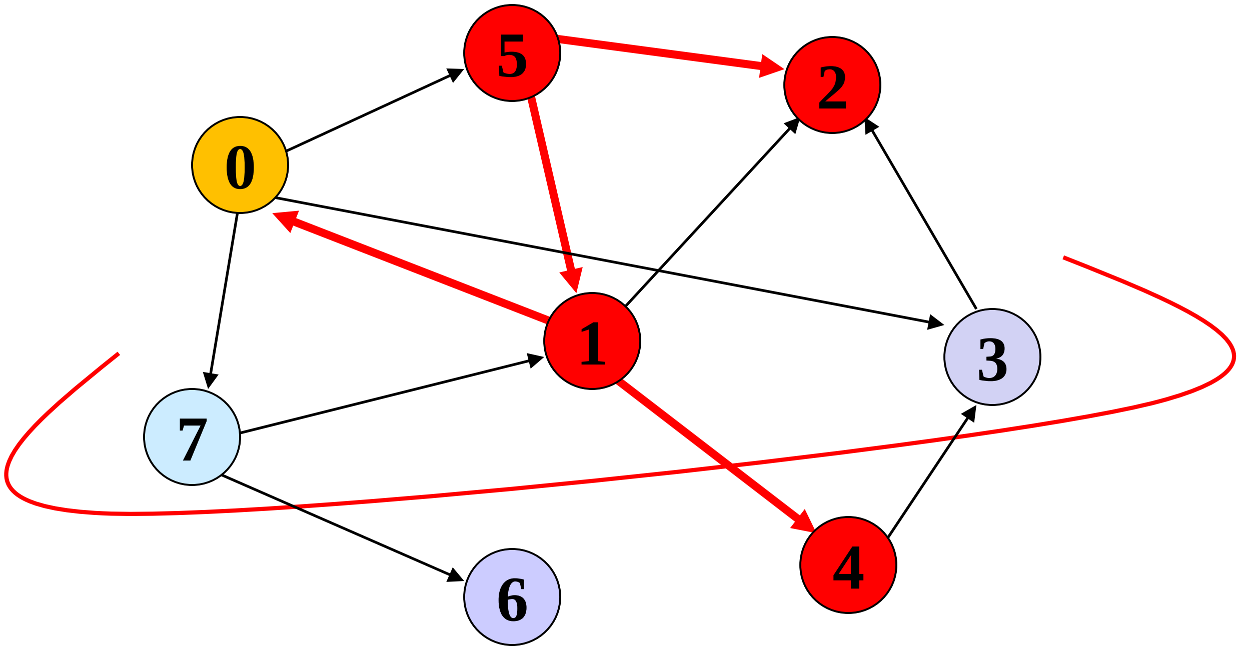
5 1 2 0 4

큐에 이웃인 0,4를 삽입
그리고 dequeue

BFS: Visit no node from 2



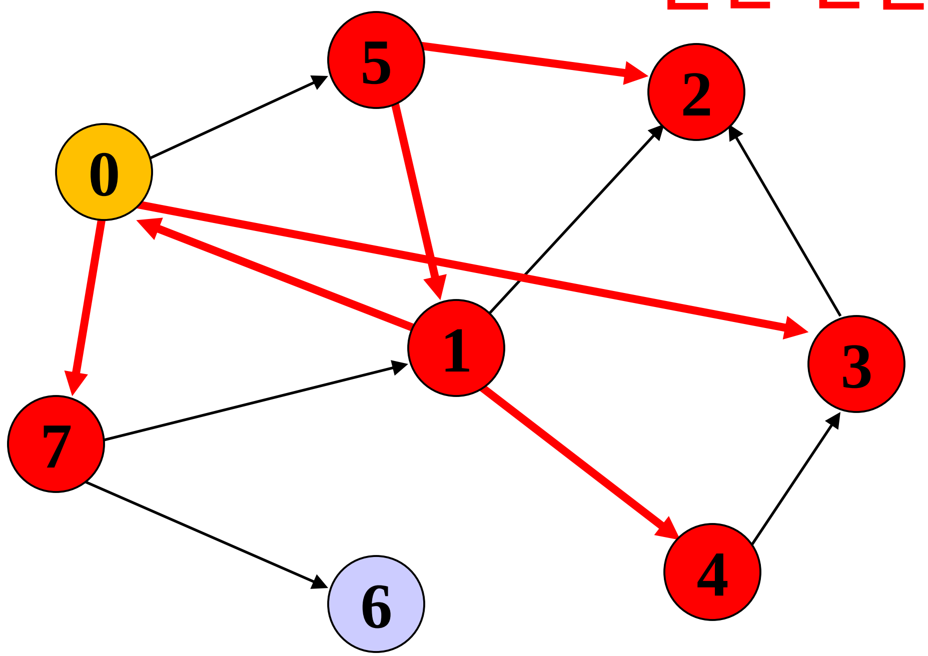
BFS: Nodes three-away



5 1 2 0 4

BFS: Visit nodes 3 and 7

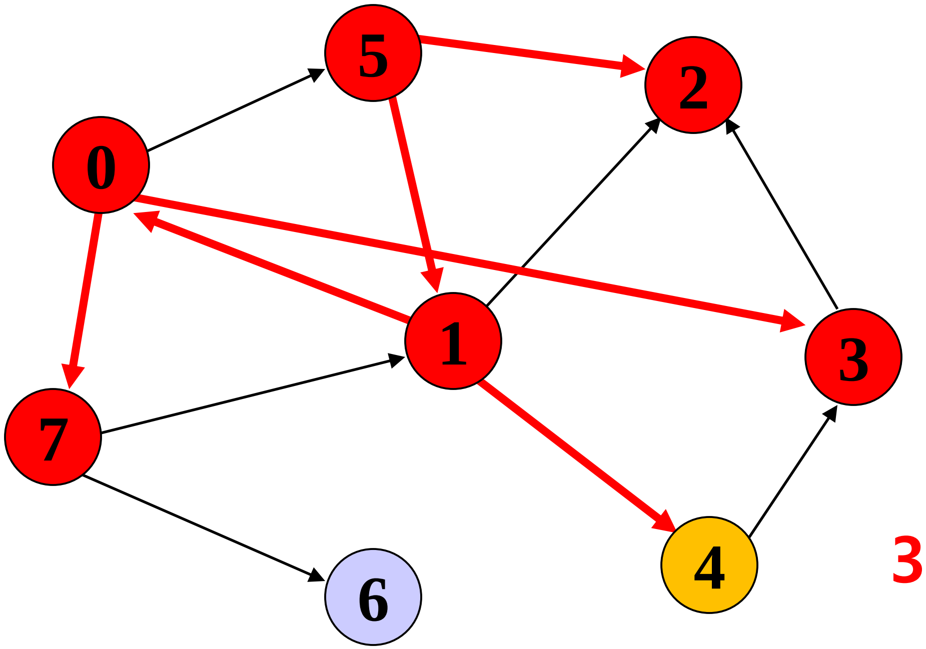
이웃노드 3,7을 빨강색으로
일단 연결



5 1 2 0 4 3 7

큐에 이웃인 3, 7을 삽입
그리고 dequeue

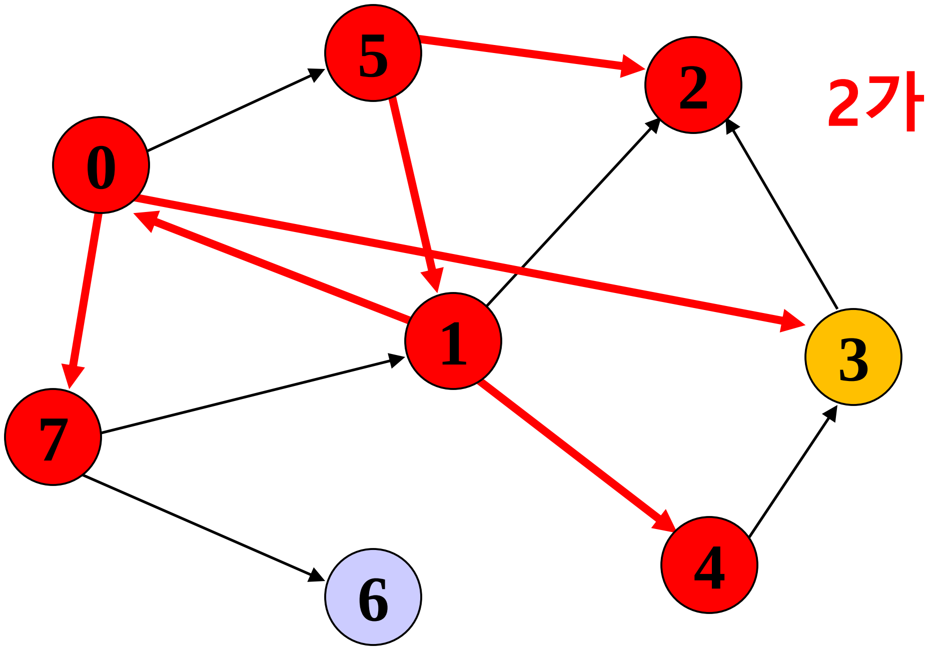
BFS: Visit no node from 4



3이 이미 빨강색

5 1 2 0 4 3 7 dequeue

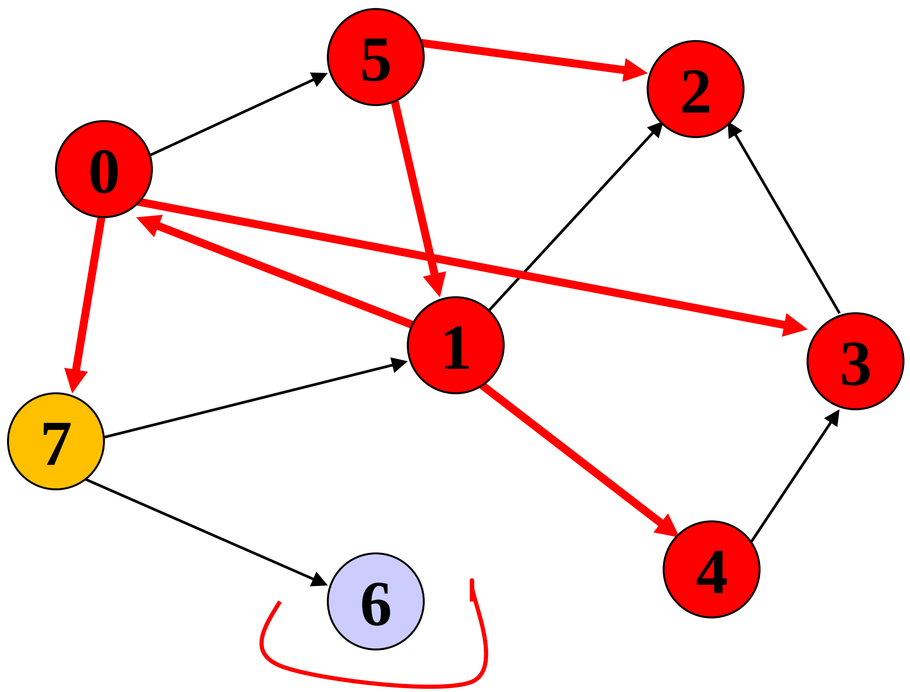
BFS: Visit no nodes from 3



2가 이미 빨강색

5 1 2 0 4 3 7 **dequeue**

BFS: Node four-away

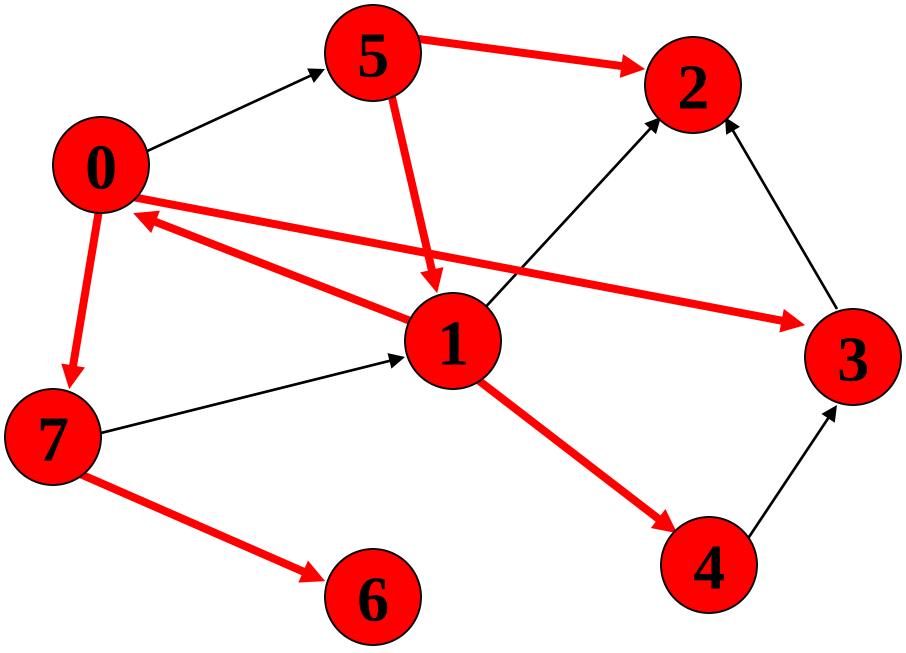


이웃노드 6를
빨강색으로
일단 연결

5 1 2 0 4 3 7 6

큐에 이웃인 6을 삽입
그리고 dequeue

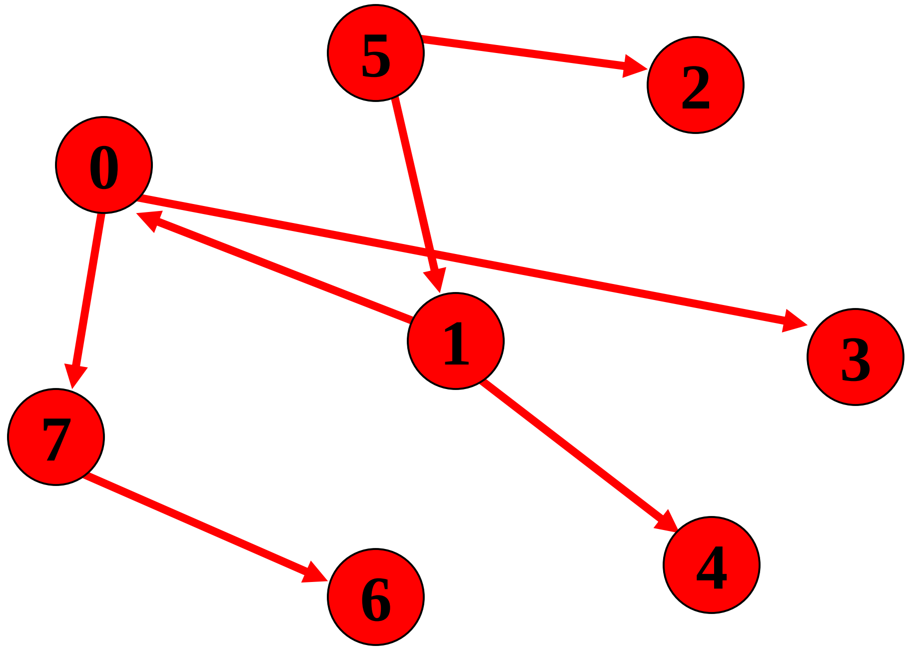
BFS: Visit 6



5 1 2 0 4 3 7 6

Spanning Tree(간선트리)

BFS방식으로 그래프의 노드를 방문했을때 얻게되는 트리를 간선트리라한다



5 1 2 0 4 3 7 6

BFS방식으로 노드순회를 마친 결과

감사합니다.