자료구조론

10장 그래프(graph)

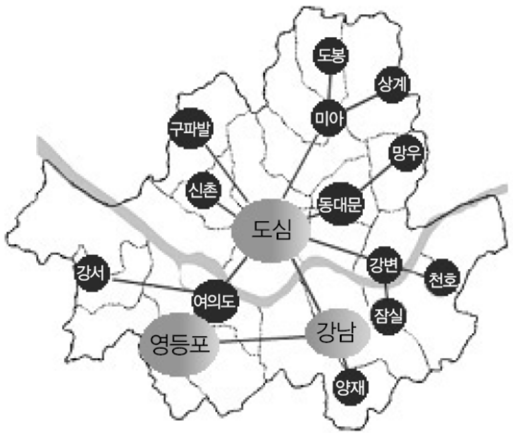
❑ 이 장에서 다룰 내용

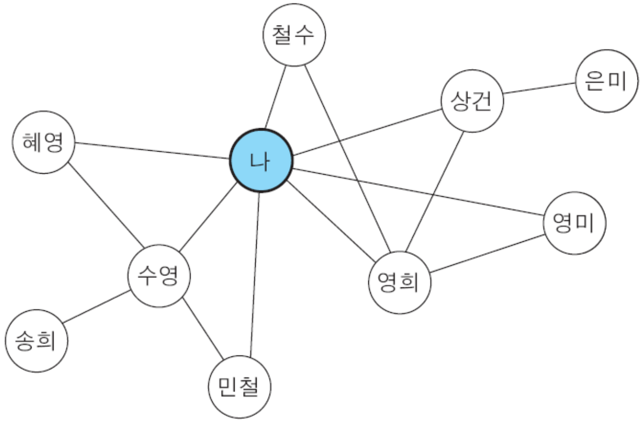
- 그래프의 구조
- 그래프의 구현
- 그래프 순회
- 신장 트리와 최소비용 신장 트리

❑ 그래프

❖ 그래프(graph)

- 현상이나 사물을 정점(vertex)과 간선(edge)으로 표현한 것으로서, 정점은 대상을 나타내고, 간선은 대상들 간의 관계를 나타냄
- 원소들 간의 多:多의 관계를 표현할 수 있다.
  - 선형 자료구조나 트리 자료구조로는 多:多의 관계를 표현 못함
- 예)





- 그래프 G는 다음과 같이 정의할 수 있다.

  G = (V, E)

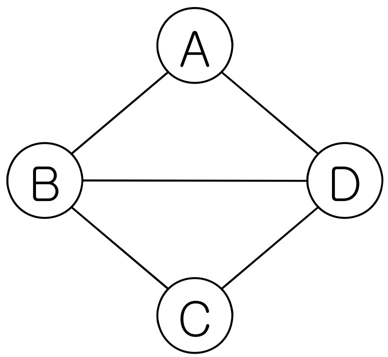
  - V는 정점들의 집합
  - E는 간선들의 집합

❑ 그래프 종류

❖ 무방향 그래프(undirected graph)

- 간선에 방향이 없는 그래프
- 정점 v_i와 정점 v_j 사이의 간선을 (v_i, v_j)로 표현
  - (v_i, v_j)와 (v_j, v_i)는 동일한 간선을 나타냄
- 예)

G1:



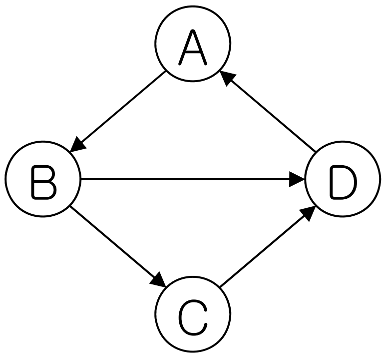
|V| 집합 V의 크기
|E| 집합 E의 크기
V(G) : 그래프 G의 정점 집합
E(G) : 그래프 G의 간선 집합

V(G1) = {A, B, C, D}
E(G1) = {(A,B), (A,D), (B,C), (B,D), (C,D)}

❑ 그래프 종류

- 방향 그래프(directed graph)
  - 다이그래프(digraph)라고도 부름
  - 간선에 방향이 있는 그래프
  - 정점 v_i에서 정점 v_j를 연결하는 간선 즉, $v_i \rightarrow v_j$를 $<v_i, v_j>$로 표현
    - v_i를 꼬리(tail), v_j를 머리(head)라고 한다.
    - $<v_i, v_j>$와 $<v_j, v_i>$는 서로 다른 간선을 나타냄
  - 예)

G3:



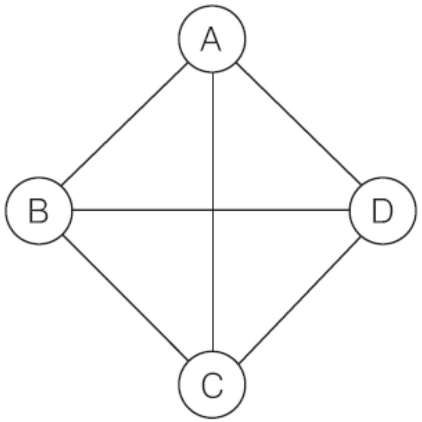
V(G3) = {A, B, C, D}

E(G3) = {<A,B>, <D,A>, <B,C>, <B,D>, <C,D>}

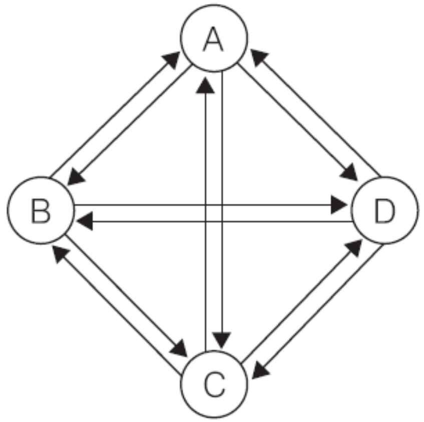
❑ 그래프 종류

❖ 완전 그래프(complete graph)

- 각 정점에서 다른 모든 정점으로의 간선이 존재하는 그래프
  - 주어진 정점 수에 대해 간선 수가 최대
- 정점이 n개인 완전 그래프의 간선 수
  - undirected graph인 경우 n(n-1)/2 개
  - directed graph인 경우 n(n-1) 개
- 예)



G5

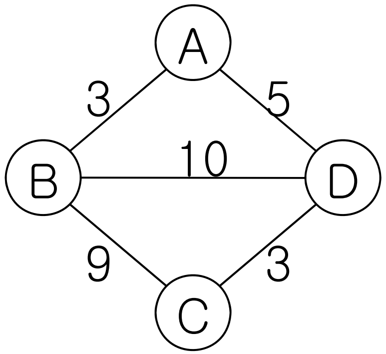


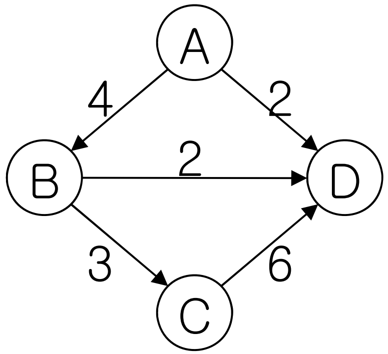
G6

❑ 그래프 종류

❖ 가중 그래프(weighted graph)

- 간선에 가중치(weight)가 주어진 그래프
- 가중 그래프의 간선에 주어진 가중치는 두 정점 사이의 비용, 거리, 시간 등을 의미하는 값이다.
- 예)

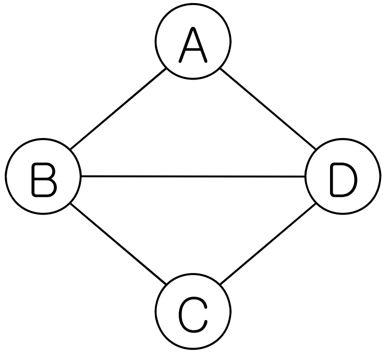




❑ 그래프 용어

❖ 인접(adjacent), 부속(incident)

- 두 정점 v_i 와 v_j 사이에 간선 (v_i, v_j) 가 존재하면,
  - v_i 와 v_j 는 인접하다(adjacent).
  - (v_i, v_j)는 v_i 와 v_j 에 부속되어 있다(incident).
- 예)



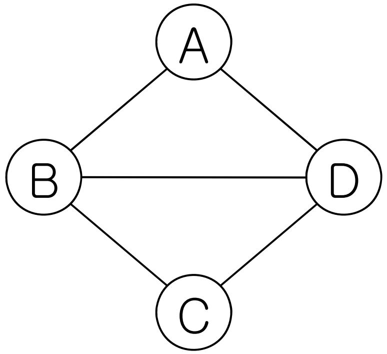
  - 정점 A와 인접한 정점은 B와 D
  - 정점 A에 부속되어 있는 간선은 (A, B)와 (A, D)
  - 간선 (B, C)는 B와 C에 부속되어 있다.

❑ 그래프 용어

❖ 차수(degree)

- 정점의 차수 = 정점에 부속된(incident) 간선의 수

  예)



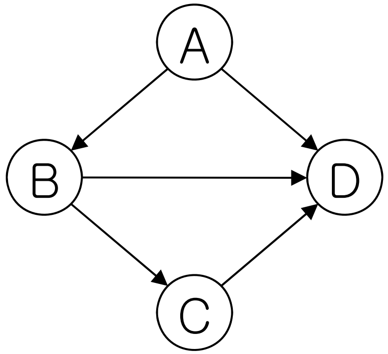
정점 A의 차수는 2

정점 B의 차수는 3

- 방향 그래프의 경우

  정점의 차수 = 진입차수(in-degree) + 진출차수(out-degree)

  예)



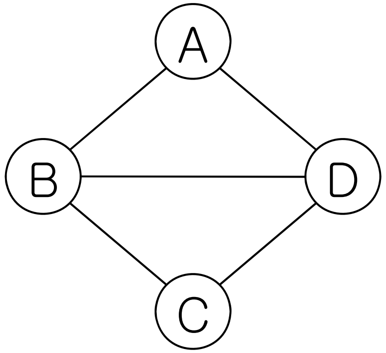
정점 B의 차수는 3

(진입차수 1 + 진출차수 2)

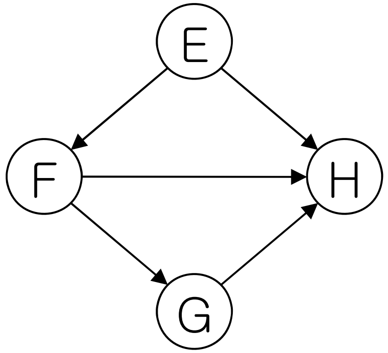
❑ 그래프 용어

❖ 경로(path)

- 그래프에서 한 정점에서 한 정점으로 간선을 따라 도달할 수 있는 길
- 정점 v_i에서 정점 v_j까지의 경로란 v_i에서 v_j에 이르기까지 간선으로 연결된 정점들을 순서대로 나열한 리스트
- 예)



A에서 C까지의 경로:
A-B-C, A-B-D-A-B-C, ...
B에서 C까지의 경로:
B-C, B-D-C, ...



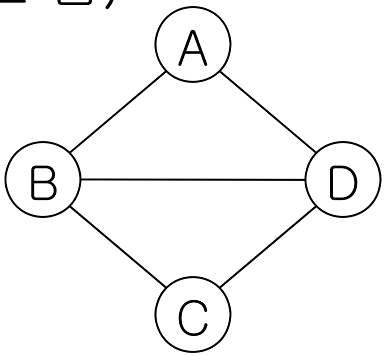
F에서 E까지의 경로:
없음
E에서 H까지의 경로:
E-H, E-F-G-H, ...

- 경로 길이(path length) : 경로를 구성하는 간선의 수
  - 경로 A-B-C의 길이 = 2

❑ 그래프 용어

❖ 단순 경로(simple path)

- 모두 다른 정점들로 구성된 경로(단, 시작 정점과 마지막 정점은 동일해도 됨)



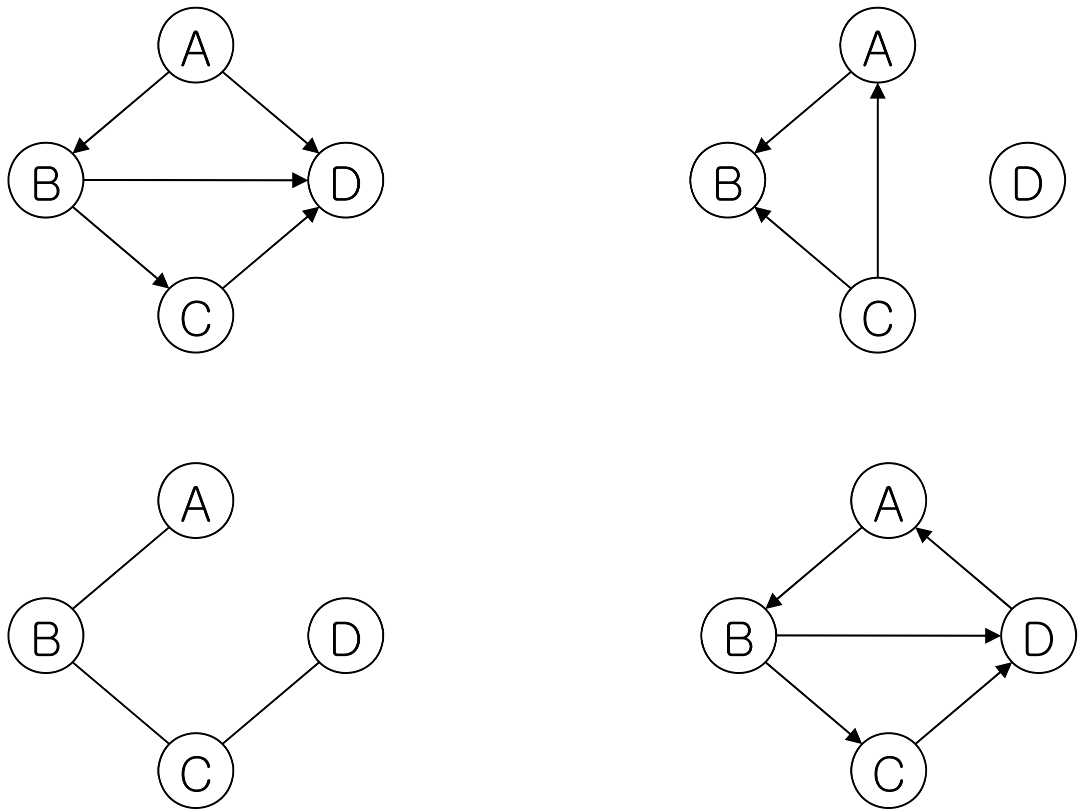
  - 단순 경로 예) A-B-C, B-A-D-C, A-B-C-D-A
  - 단순 경로가 아닌 예) A-B-D-A-B-C

- 사이클(cycle) : 시작 정점과 마지막 정점이 같은 단순 경로
  - 예) A-B-C-D-A

❑ 그래프 용어

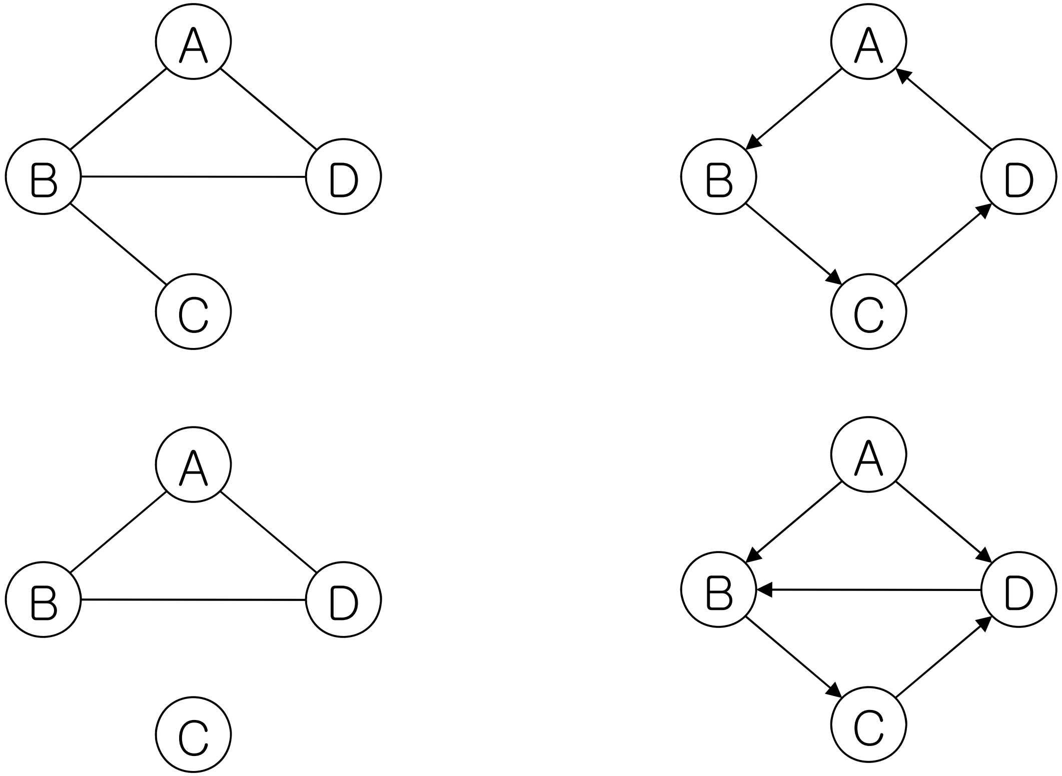
❖ DAG(directed acyclic graph)

- 사이클이 없는 방향 그래프
- 예) 다음 네 개의 그래프 중 DAG인 것을 모두 고르시오.



❑ 그래프 용어

- 연결(connected)
  - 정점 v_i에서 v_j까지 경로가 있으면 v_i와 v_j가 연결되었다고 함
- 연결 그래프(connected graph)
  - 모든 쌍의 정점들 사이에 경로가 있는 그래프
  - 예) 다음 네 개의 그래프 중 연결 그래프인 것을 모두 고르시오.

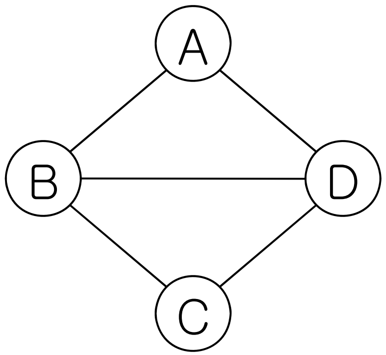


❑ 그래프 용어

- 부분그래프(subgraph)
  - 원래의 그래프에서 정점이나 간선의 일부를 취해 만든 그래프
  - 그래프 G와 부분그래프 G'의 관계

$$V(G') \subseteq V(G), \quad E(G') \subseteq E(G)$$

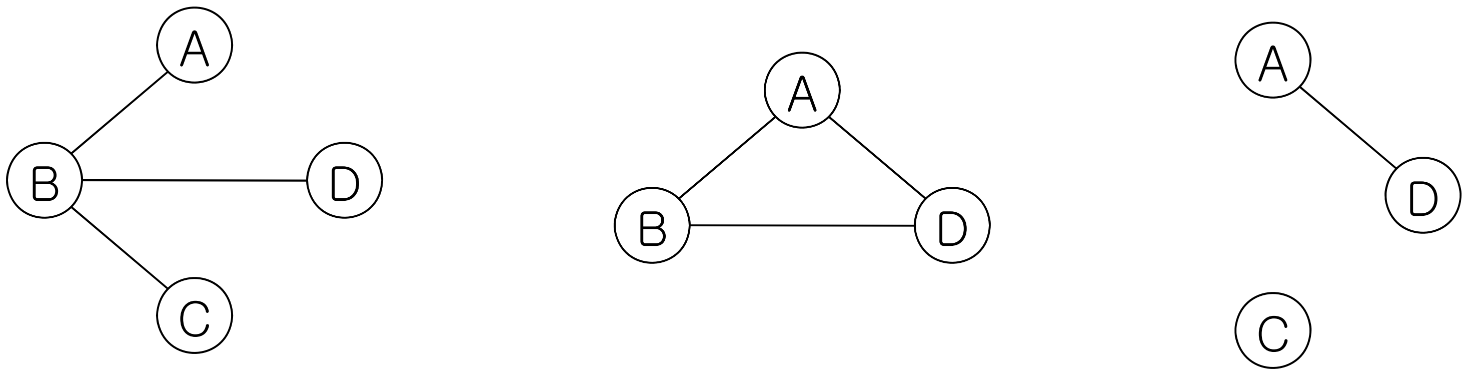
G:



$V(G) = \{A, B, C, D\}$

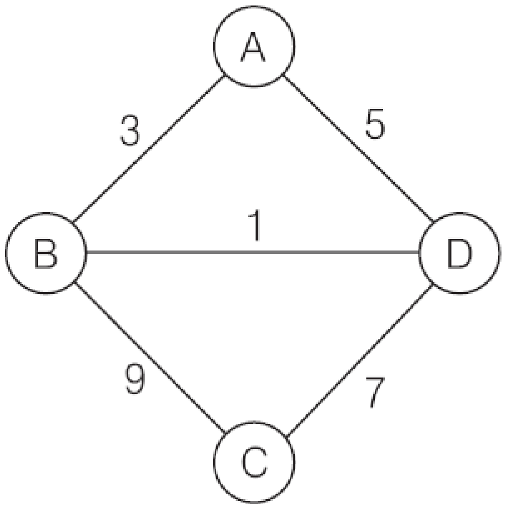
$E(G) = \{(A,B),(A,D),(B,C),(B,D),(C,D)\}$

  - 그래프 G의 부분그래프 예

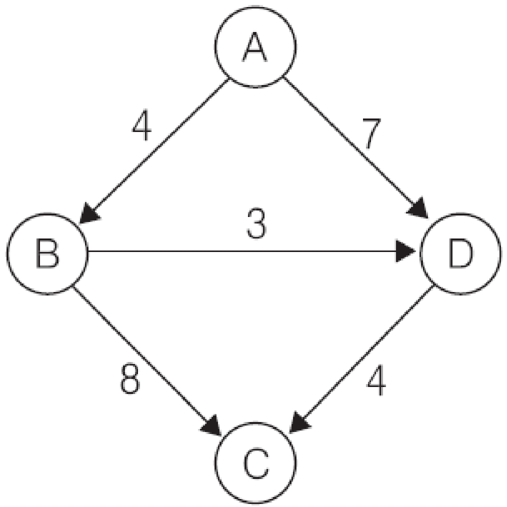


❑ 그래프

- 가중 그래프(weight graph) , 네트워크(network)
  - 정점을 연결하는 간선에 가중치(weight)를 할당한 그래프



G7



G8

❑ 그래프

❖ 그래프 추상 자료형

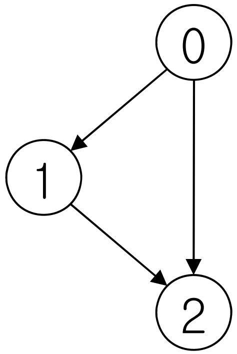
```
ADT Graph
데이터 : 정점의 집합과 간선의 집합
연산 :
    G∈Graph; u,v∈V(G);
    createGraph() ::= create an empty Graph;
        // 공백 그래프의 생성 연산
    isEmpty(G) ::= if (G have no vertex) then return true;    else return false;
        // 그래프 G가 정점이 없는 공백 그래프인지를 검사하는 연산
    insertVertex(G, v) ::= insert vertex v into G;
        // 그래프 G에 정점 v를 삽입하는 연산
    insertEdge(G, u, v) ::= insert edge (u,v) into G;
        // 그래프 G에 간선 (u,v)를 삽입하는 연산
    deleteVertex(G, v) ::= delete vertex v and all edges incident on v from G;
        // 그래프 G에서 정점 v를 삭제하고 그에 부속된 모든 간선을 삭제하는 연산
    deleteEdge(G, u, v) ::= delete edges (u,v) from G;
        // 그래프 G에서 간선 (u,v)를 삭제하는 연산
    adjacent(G, v) ::= return set of all vertices adjacent to v;
        // 정점 v에 인접한 모든 정점을 반환하는 연산
End Graph
```

❑ 그래프의 구현

❖ 그래프 구현 방법 2가지

- 인접 행렬(adjacency matrix)

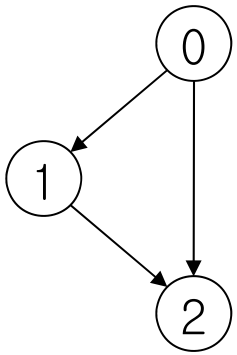
순차 자료구조로 구현

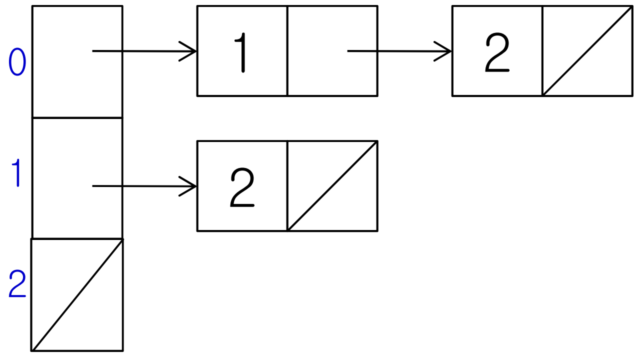


	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0

- 인접 리스트(adjacency list)

연결 자료구조로 구현



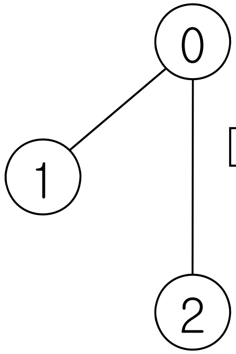


❑ 그래프의 구현 – 인접 행렬

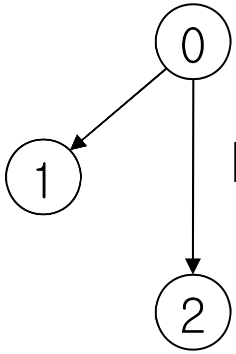
G = (V, E)에 대해
정점 수 |V| = n,
간선 수 |E| = e 라고 하자.

❖ 인접 행렬(adjacency matrix)

- 2차원 배열로 표현한 행렬에 두 정점의 인접 여부(간선 존재 여부)를 저장
  - n x n 정방행렬 A를 사용
  - A_{ij} : 정점 i와 j가 인접하면 1, 아니면 0



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



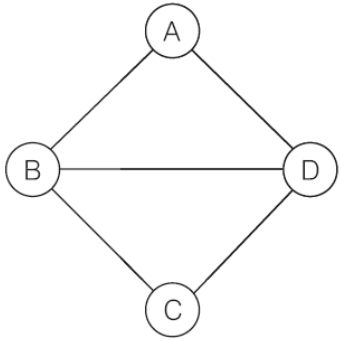
$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{array}$$

- 무방향 그래프의 인접 행렬
  - i행의 합 = i열의 합 = 정점 i의 차수
- 방향 그래프의 인접 행렬
  - i행의 합 = 정점 i의 out-degree
  - i열의 합 = 정점 i의 in-degree

❑ 그래프의 구현 – 인접 행렬

- 예)

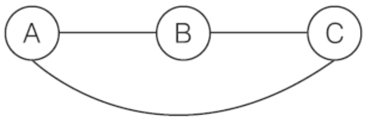
G1



	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	1
D	1	1	1	0

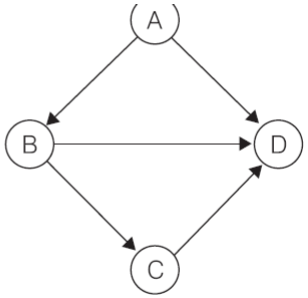
1+0+1+1=3 ⇨ 정점 B의 차수

G2



	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

G3

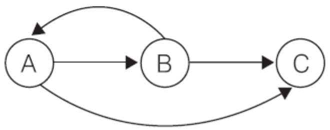


	A	B	C	D
A	0	1	0	1
B	0	0	1	1
C	0	0	0	1
D	0	0	0	0

0+0+1+1=2 ⇨ 정점 B의 진출차수

1+0+0+0=1 ⇨ 정점 B의 진입차수

G4



	A	B	C
A	0	1	1
B	1	0	1
C	0	0	0

❑ 그래프의 구현 – 인접 행렬

- 인접 행렬 표현의 단점
  - 정점이 n개이면 항상 n x n개의 메모리 사용하므로 희소 그래프인 경우 메모리 낭비가 심함
    - ➢ **희소 그래프 :** 정점의 개수에 비해서 간선의 개수가 매우 적어 인접 행렬이 희소 행렬(sparse matrix)이 됨
  - 희소 그래프인 경우 수행 시간 면에서 비효율적인 경우가 있음 – 인접 행렬의 의미 없는 내용을 모두 검사해야 하는 경우

예) 다음 그래프에서 정점 0의 차수를 구하는 연산

0	0	0	1	0	0	0	0	0	…	0
0	0	0	0	1	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	…	0
1	0	0	0	0	0	0	0	0	…	0
0	1	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	…	0
…										
0	0	0	0	0	0	0	0	0	…	0

❑ 그래프의 구현 – 인접 리스트

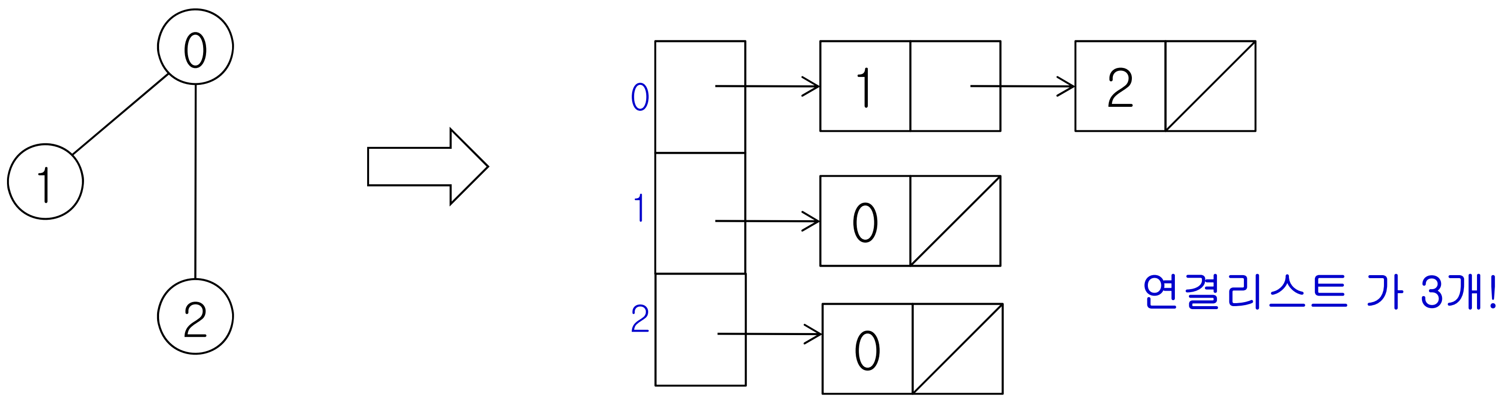
G = (V, E)에 대해
정점 수 |V| = n,
간선 수 |E| = e 라고 하자.

- 인접 리스트(adjacency list)
  - 각 정점에 대해 인접한 정점들이 무엇인지를 연결 리스트로 표현
    - n개의 연결 리스트가 필요
    - 연결 리스트의 노드 구성

```
class Node {
    int vertex;
    Node link;
}
```

➢ 정점 필드
➢ 다음 노드를 연결하는 링크 필드

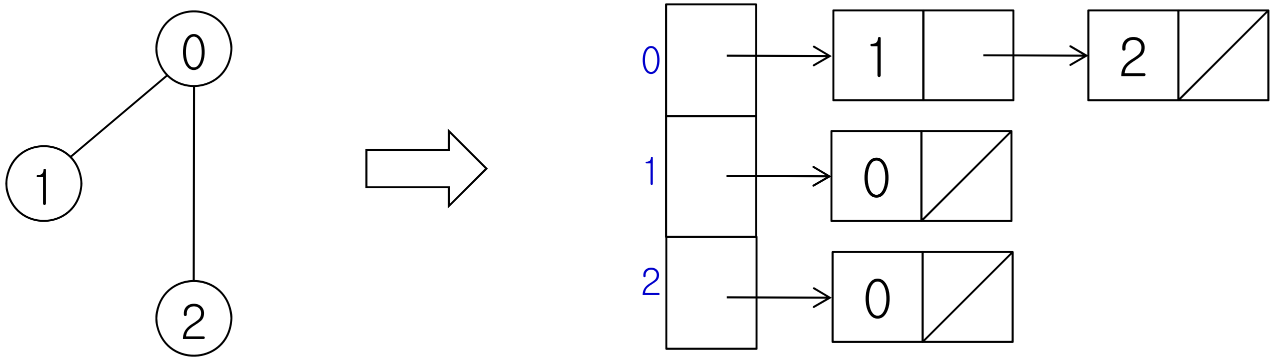
- 연결 리스트가 n개이므로 연결리스트의 첫 노드를 가리키는 변수 n개가 필요
  ➢ 이 변수들을 크기가 n인 배열로 구현



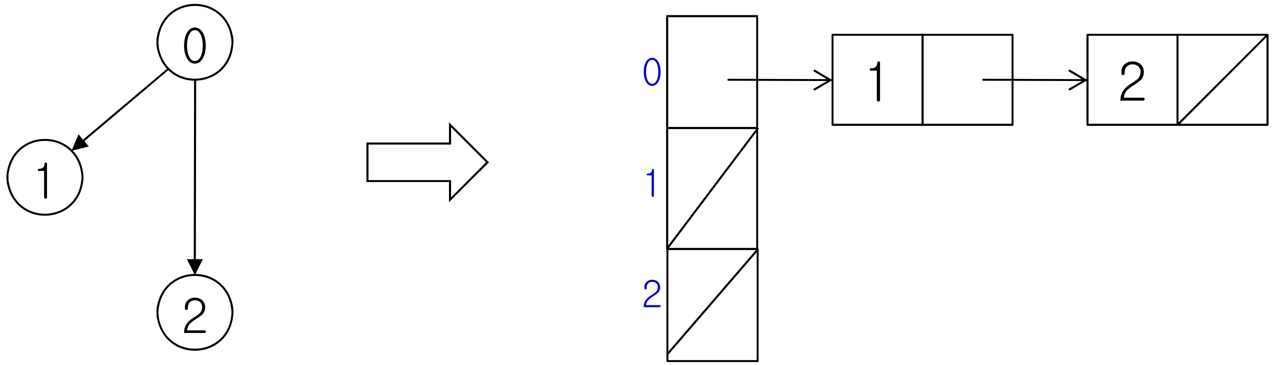
❑ 그래프의 구현 – 인접 리스트

G = (V, E)에 대해
정점 수 |V| = n,
간선 수 |E| = e 라고 하자.

- 무방향 그래프의 인접 리스트
  - 전체 노드 개수 : 2e
  - 정점 v 에 대한 리스트의 노드 개수 = v 의 degree

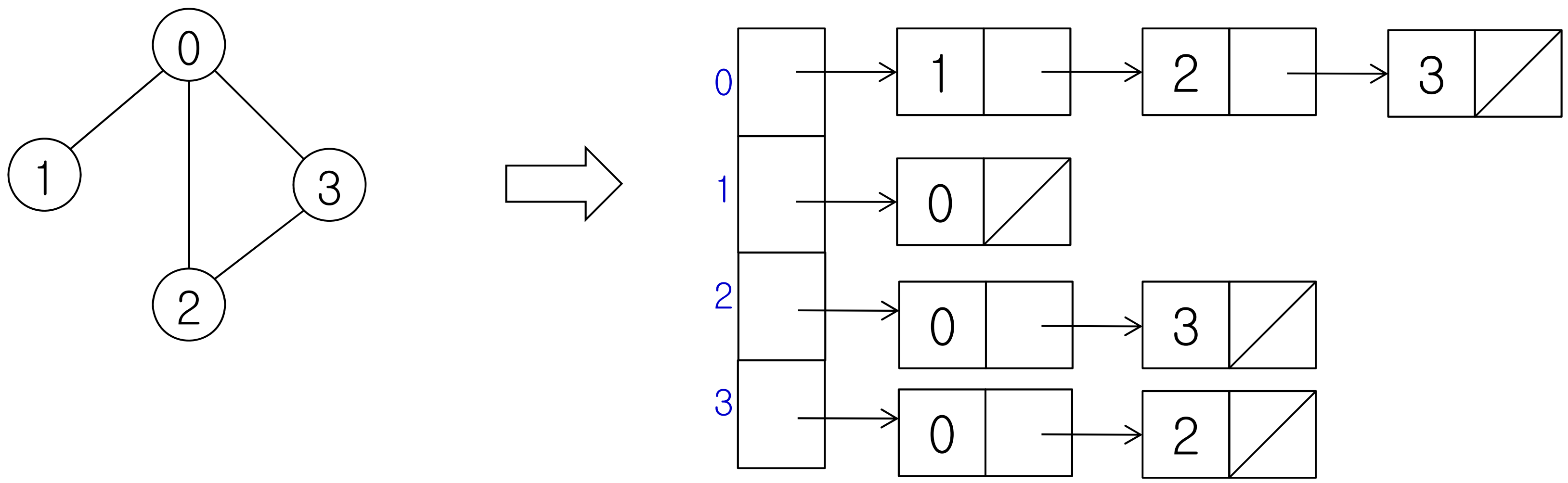


- 방향 그래프의 인접 리스트
  - 전체 노드 개수 : e
  - 정점 v 에 대한 리스트의 노드 개수 = v 의 out-degree



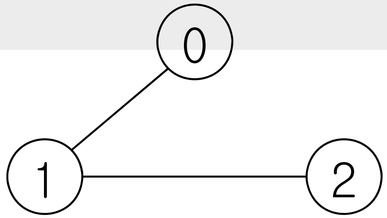
❑ 그래프의 구현 – 인접 리스트

- 인접 리스트로 구현한 무방향 그래프 예

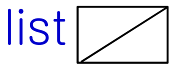


❑ 그래프의 구현 – 인접 리스트

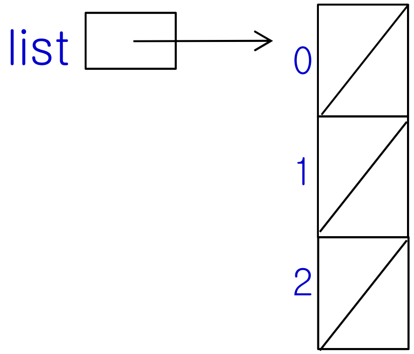
- 인접 리스트 구현 예)



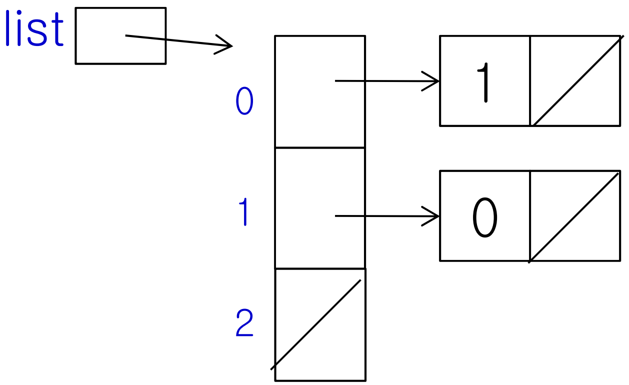
(1) Node[] list;



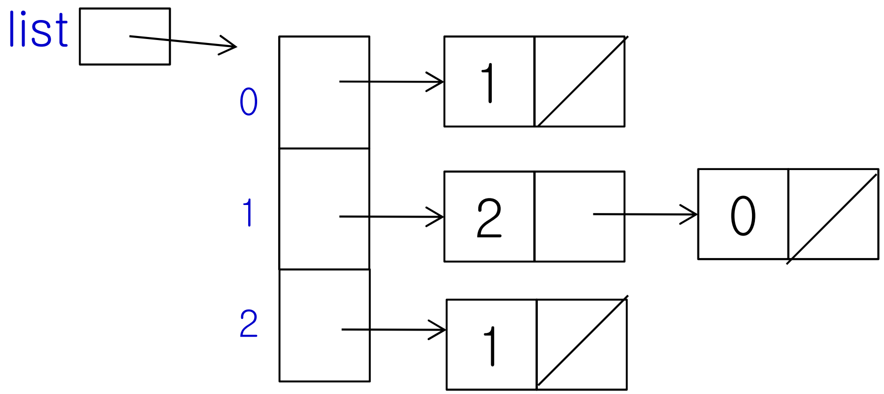
(2) list = new Node[3];



(3) 간선 (0, 1) 삽입



(4) 간선 (1, 2) 삽입



❑ 그래프의 구현

❖ 인접 행렬, 인접 리스트로 구현한 그래프 프로그램

- 문제를 간단히 하기 위해 정점 수가 n이면 정점 번호는 0, 1, 2, ..., n-1로 가정

```
public class Ex10_1 {
    public static void main(String[] args) {

        MatrixGraph g1 = new MatrixGraph(4);
        g1.insertEdge(0,3);     g1.insertEdge(0,1);     g1.insertEdge(1,3);
        g1.insertEdge(1,2);     g1.insertEdge(1,0);     g1.insertEdge(2,3);
        g1.insertEdge(2,1);     g1.insertEdge(3,2);     g1.insertEdge(3,1);
        g1.insertEdge(3,0);
        System.out.println("그래프 G1의 인접행렬 : ");
        g1.printAdjMatrix();

        ListGraph g2 = new ListGraph(4);
        g2.insertEdge(0,3);     g2.insertEdge(0,1);     g2.insertEdge(1,3);
        g2.insertEdge(1,2);     g2.insertEdge(1,0);     g2.insertEdge(2,3);
        g2.insertEdge(2,1);     g2.insertEdge(3,2);     g2.insertEdge(3,1);
        g2.insertEdge(3,0);
        System.out.println("그래프 G2의 인접리스트 : ");
        g2.printAdjList();
    }
}
```

❑ 그래프의 구현

```java
public class MatrixGraph {
    private int[][] matrix;
    private int n;  // vertex 개수

    public MatrixGraph(int n) {
        matrix = new int[n][n];
        this.n = n;
    }

    public void insertEdge(int v1, int v2) {
        if(v1<0 || v1>=n || v2<0 || v2>=n)
            System.out.println("그래프에 없는 정점입니다!");
        else matrix[v1][v2] = 1;
    }

    public void printAdjMatrix() {
        for(int i=0; i<n; i++) {
            for(int j=0; j<n; j++)
                System.out.print(matrix[i][j] + " ");
            System.out.println();
        }
    }
}
```

❑ 그래프의 구현

```
public class ListGraph {
    private Node[] list;
    private int n;   // vertex 개수

    public ListGraph(int n) {
        list = new Node[n];
        this.n = n;
    }

    public void insertEdge(int v1, int v2) {
        if(v1<0 || v1>=n || v2<0 || v2>=n)
            System.out.println("그래프에 없는 정점입니다!");
        else {
            // vertex가 v2인 새로운 노드를 v1의 연결리스트 맨 앞에 삽입
            Node newNode = new Node();
            newNode.vertex = v2;
            newNode.link = list[v1];
            list[v1] = newNode;
        }
    }
```

❑ 그래프의 구현

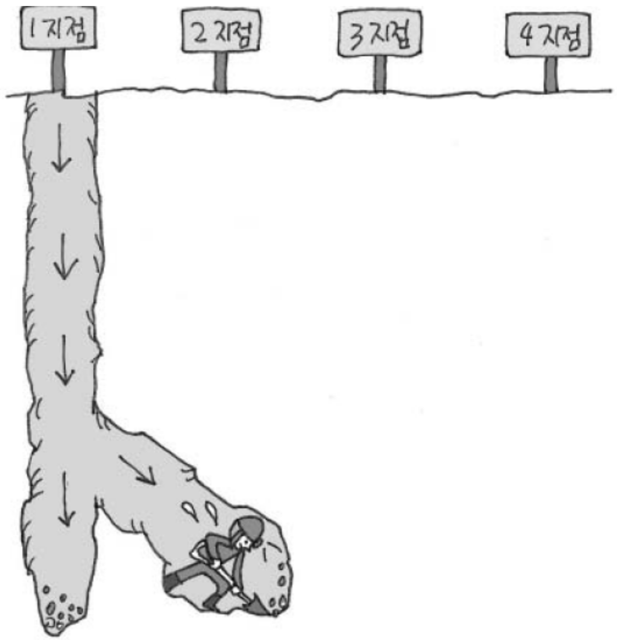
```
    public void printAdjList() {
        for(int i=0; i<n; i++)  {
            System.out.print("정점 " + i + "의 인접리스트");
            for(Node temp = list[i]; temp != null; temp = temp.link)
                System.out.print(" -> "+ temp.vertex);
            System.out.println();
        }
    }

    private class Node {
        int vertex;
        Node link;
    }
}
```

❑ 그래프 순회

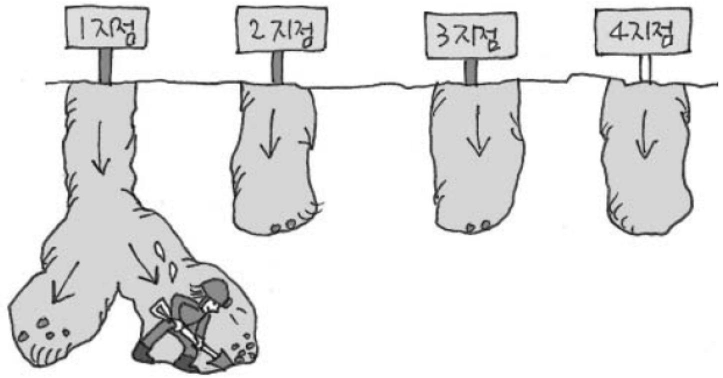
- 그래프 순회(graph traversal)
  - 그래프 탐색(graph search)
  - 어떤 정점에서 시작하여 그래프에 있는 모든 정점들을 한번씩 방문
- 그래프 순회(탐색) 방법
  - 깊이 우선 탐색(Depth First Search : DFS)
  - 너비 우선 탐색(Breadth First Search : BFS)

❑ 그래프 순회

- 그래프 순회 방법을 우물 파기에 비유해보자.
  - 한 지점을 골라서 팔 수 있을 때까지 계속해서 깊게 파다가 아무리 땅을 파도 물이 나오지 않으면, 밖으로 나와 다른 지점을 골라서 다시 깊게 땅을 파는 방법 ➔ 깊이 우선 탐색
  - 여러 지점을 고르게 파보고 물이 나오지 않으면, 파놓은 구덩이들을 다시 조금씩 더 깊게 파는 방법 ➔ 너비 우선 탐색



(a) 깊이 우선 탐색의 예



(b) 너비 우선 탐색의 예

❑ 그래프 순회

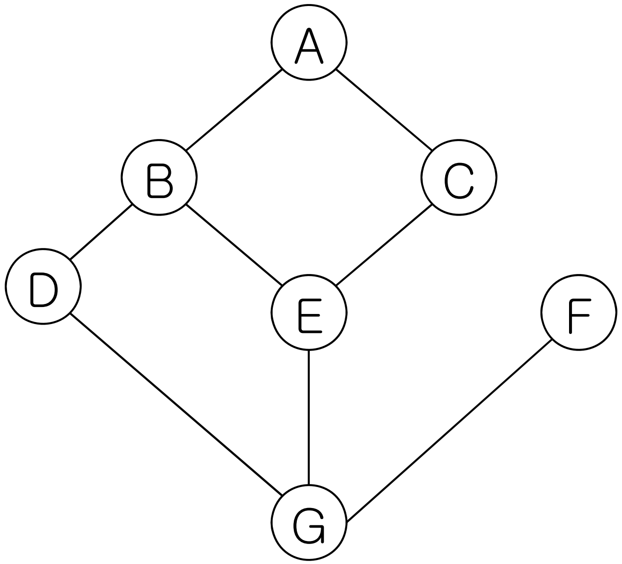
❖ 깊이 우선 탐색(Depth First Search : DFS)

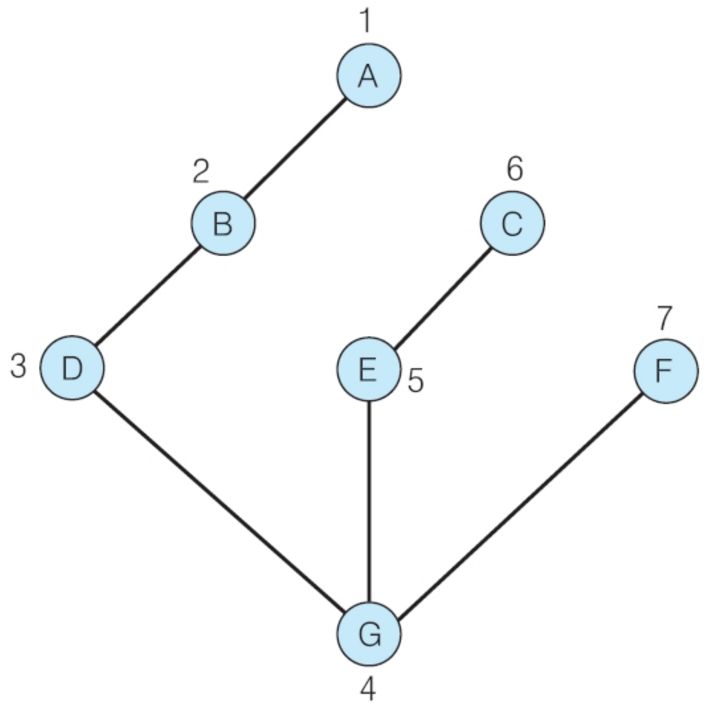
- G의 모든 정점을 순회하는 DFS 알고리즘(재귀적으로 구현)

```
DFS(G)   // 깊이우선탐색 방법으로 그래프 G를 순회
    for each v ∈ V(G)
        visited[v] ← false;
    for each v ∈ V(G)
        if (visited[v] = false) then aDFS(v);
end DFS()

aDFS(v)   // v를 시작 정점으로 하여 그래프 G를 깊이우선탐색
    visited[v] ← true;
    v 방문;
    for each x ∈ L(v)     ▷ L(v) : 정점 v의 인접 리스트
        if (visited[x] = false) then aDFS(x);
end aDFS()
```

❑ 그래프 순회

- 예) 깊이우선탐색 순서 : A–B–D–G–E–C–F





❖ 너비 우선 탐색(Breadth First Search : BFS)

- 순회 방법
  - 가까운 정점들을 먼저 방문하고 멀리 있는 정점들은 나중에 방문하는 순회 방법
  - 시작 정점으로부터 인접한 정점들을 모두 차례로 방문하고 나서, 방문했던 정점을 시작으로 하여 다시 인접한 정점들을 차례로 방문하는 방식
  - 인접한 정점들에 대해서 차례로 다시 너비 우선 탐색을 반복해야 하므로 선입선출의 구조를 갖는 큐를 사용

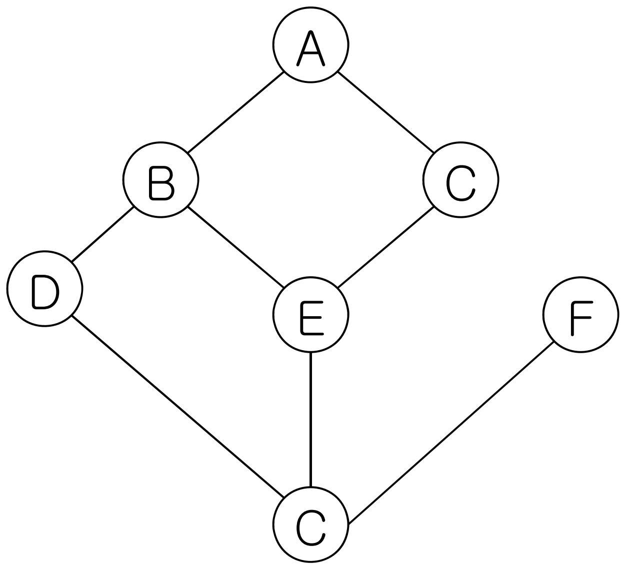
❑ 그래프 순회

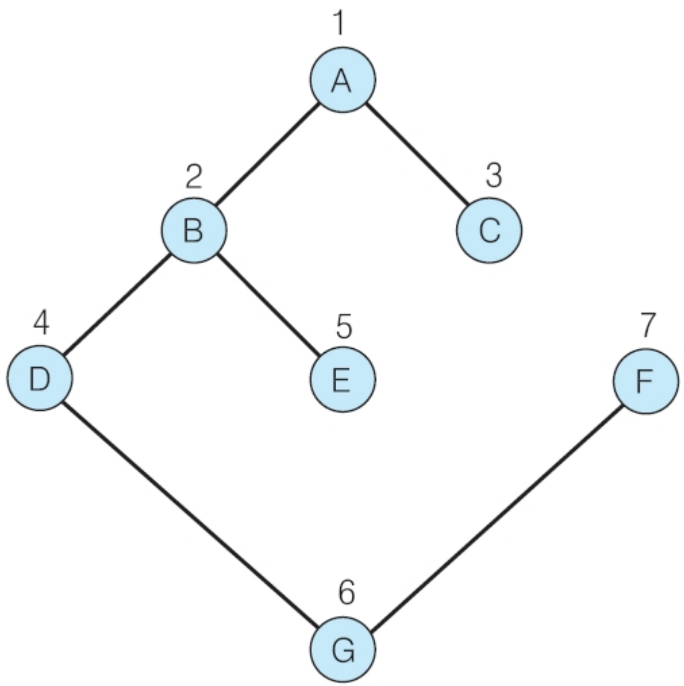
- G의 정점들을 순회하는 BFS 알고리즘(큐 이용)

```
BFS(s)  // s를 시작정점으로 하여 그래프 G를 너비우선탐색
    for each v ∈ V(G)
        visited[v] ← false;
    visited[s] ← true;
    s 방문;
    Q←createQueue();
    enQueue(Q, s);

    while (not isEmpty(Q)) do {
        v ← deQueue(Q);
        for each x ∈ L(v)    ▷ L(v) : 정점 v의 인접 리스트
            if (visited[x] = false) then {
                visited[x] ← true;
                x 방문;
                enQueue(Q, x);
            }
    }
end BFS()
```

❑ 그래프 순회

- 너비우선탐색 순서 : A-B-C-D-E-G-F

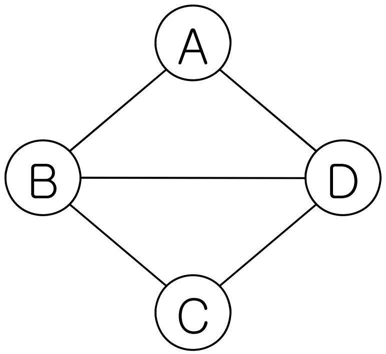




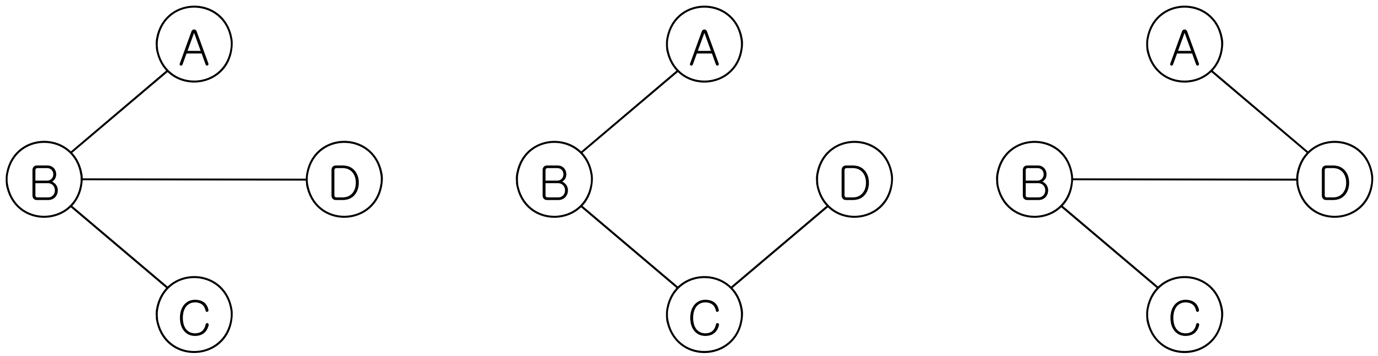
❑ 신장 트리와 최소 비용 신장 트리

❖ 신장 트리(spanning tree)

- n개의 정점으로 이루어진 **무방향 연결 그래프**의 subgraph 중에서, 정점은 n개 전부 포함하고, 간선은 n-1개만 포함하여 모든 정점들이 연결되도록 한 것
- 신장 트리는 최소 개수의 간선으로 그래프의 모든 정점들이 연결되도록 함
- 예) G1:

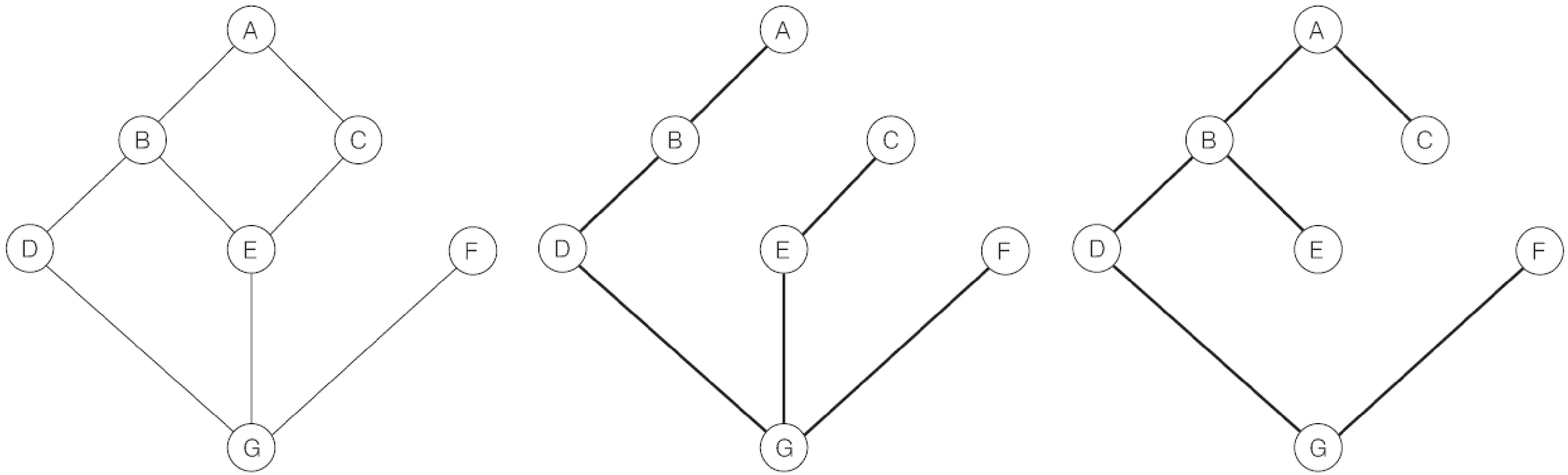


G1의 신장 트리들:



❑ 신장 트리와 최소 비용 신장 트리

- 깊이 우선 신장 트리(depth first spanning tree)
  - 깊이 우선 탐색을 이용하여 생성된 신장 트리
- 너비 우선 신장 트리(breadth first spanning tree)
  - 너비 우선 탐색을 이용하여 생성된 신장 트리
- 예) 그래프 G9의 깊이 우선 신장 트리와 너비 우선 신장 트리

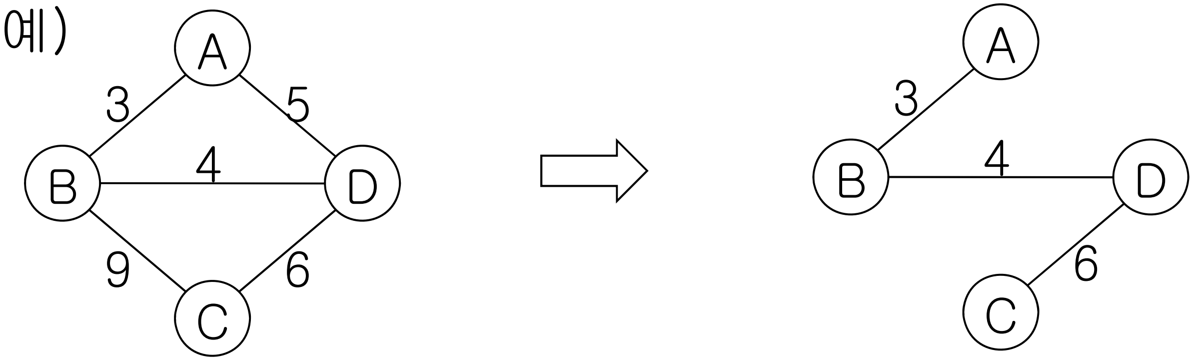


G9　　G9의 깊이 우선 신장 트리　　G9의 너비 우선 신장 트리

❑ 신장 트리와 최소 비용 신장 트리

❖ 최소 비용 신장 트리(minimum cost spanning tree)

- 무방향 가중 연결 그래프(undirected weighted connected graph)의 신장 트리들 중, 간선들의 가중치 합이 최소인 신장 트리
- 예)



❖ 최소 비용 신장 트리를 구하는 알고리즘

- Kruskal 알고리즘
- Prim 알고리즘

❑ 최소 비용 신장 트리 – Kruskal 알고리즘

❖ Kruskal 알고리즘

- 정점들은 그대로 두고 간선이 하나도 없는 상태에서 시작하여 가중치가 낮은 순서로 간선을 삽입하면서 최소 비용 신장 트리를 만든다.
- Kruskal 알고리즘 수행 과정

⑴ 그래프 G의 모든 간선을 가중치에 따라 오름차순으로 정렬한다.

⑵ 그래프 G에 가중치가 가장 작은 간선을 고른다.

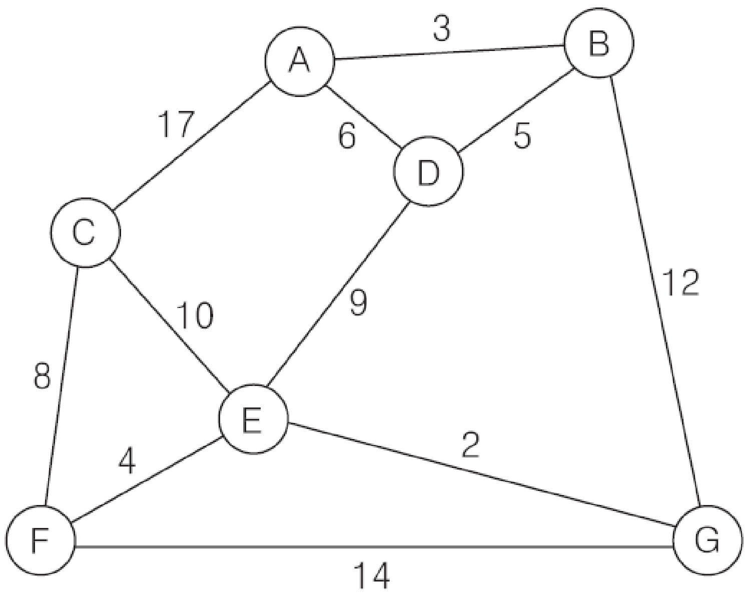
이 간선이 사이클을 형성하지 않으면 트리 간선 집합 T에 삽입하고, 사이클을 형성하면 버리고 가중치가 다음으로 작은 간선을 선택한다.

⑶ T가 n-1개의 간선을 포함할 때까지 ⑵를 반복한다.

❑ 최소 비용 신장 트리 – Kruskal 알고리즘

- 예) G10의 최소 비용 신장 트리 구하기 – Kruskal 알고리즘
- 초기 상태 : 그래프 G10의 간선을 가중치에 따라서 오름차순 정렬

초기 상태 : 그래프 G10의 간선을 가중치에 따라 오름차순으로 정렬한다.

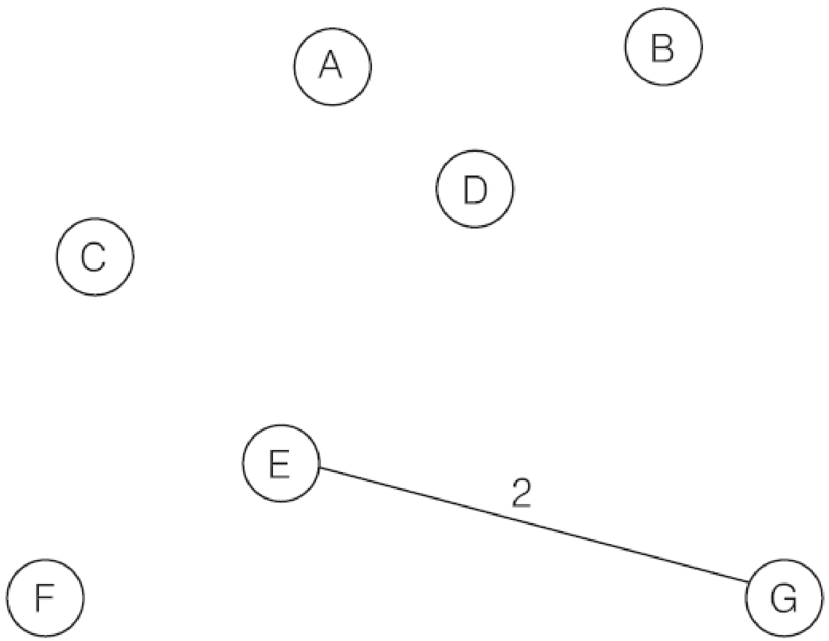


가중치	간선
2	(E, G)
3	(A, B)
4	(E, F)
5	(B, D)
6	(A, D)
8	(C, F)
9	(D, E)
10	(C, E)
12	(B, G)
14	(F, G)
17	(A, C)

간선의 수 : 11개

❑ 최소 비용 신장 트리 – Kruskal 알고리즘

① 가중치가 가장 작은 간선 (E,G) 삽입

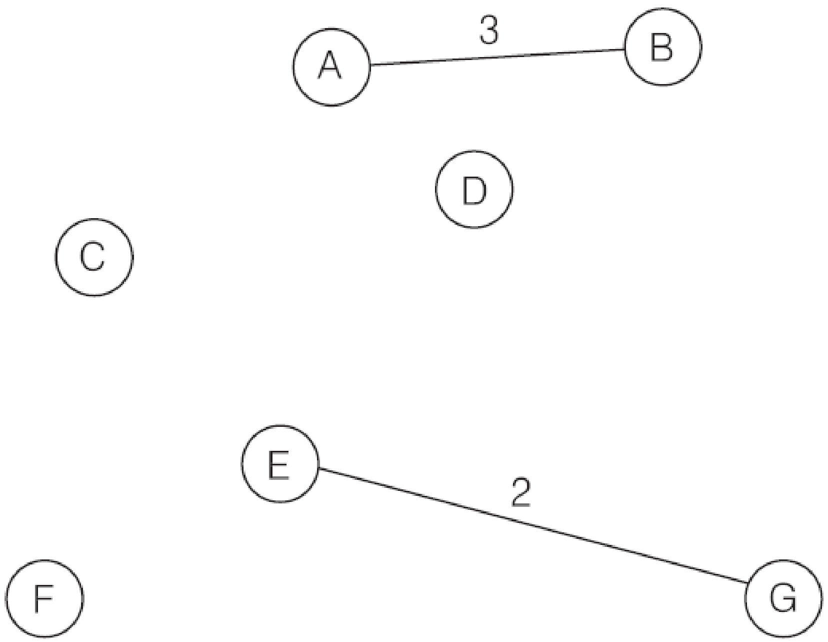


가중치	간선
2	(E, G)
3	(A, B)
4	(E, F)
5	(B, D)
6	(A, D)
8	(C, F)
9	(D, E)
10	(C, E)
12	(B, G)
14	(F, G)
17	(A, C)

삽입한 간선의 수 : 1개

❑ 최소 비용 신장 트리 – Kruskal 알고리즘

② 나머지 간선 중에서 가중치가 가장 작은 간선 (A,B) 삽입

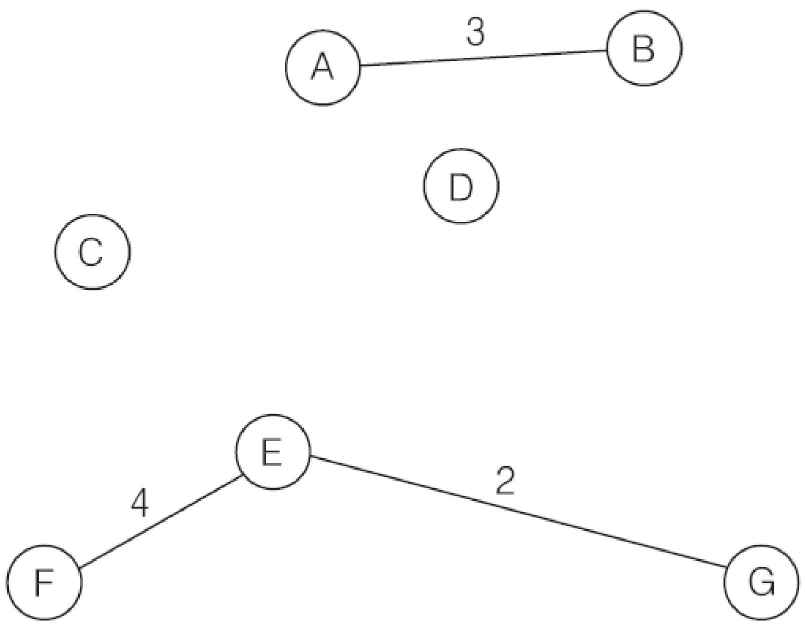


가중치	간선
2	(E, G)
3	(A, B)
4	(E, F)
5	(B, D)
6	(A, D)
8	(C, F)
9	(D, E)
10	(C, E)
12	(B, G)
14	(F, G)
17	(A, C)

삽입한 간선의 수 : 2개

❑ 최소 비용 신장 트리 – Kruskal 알고리즘

③ 나머지 간선 중에서 가중치가 가장 작은 간선 (E,F) 삽입

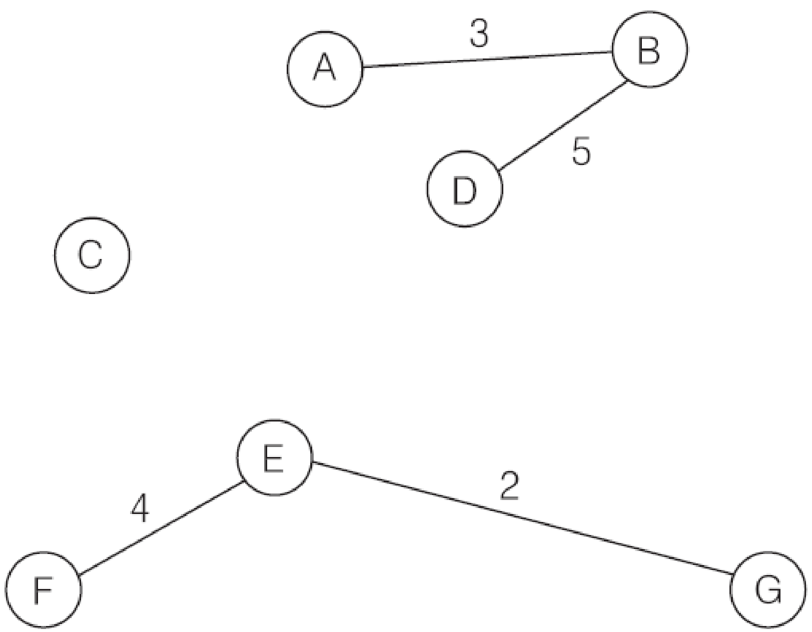


가중치	간선
2	(E, G)
3	(A, B)
4	(E, F)
5	(B, D)
6	(A, D)
8	(C, F)
9	(D, E)
10	(C, E)
12	(B, G)
14	(F, G)
17	(A, C)

삽입한 간선의 수 : 3개

❑ 최소 비용 신장 트리 – Kruskal 알고리즘

④ 나머지 간선 중에서 가중치가 가장 작은 간선 (B,D) 삽입

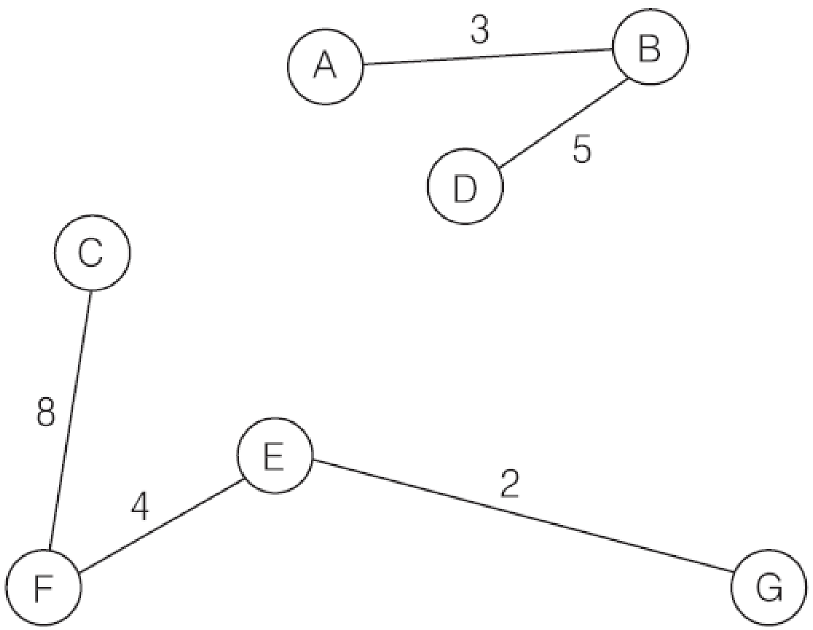


가중치	간선
2	(E, G)
3	(A, B)
4	(E, F)
5	(B, D)
6	(A, D)
8	(C, F)
9	(D, E)
10	(C, E)
12	(B, G)
14	(F, G)
17	(A, C)

삽입한 간선의 수 : 4개

❑ 최소 비용 신장 트리 – Kruskal 알고리즘

⑤ 나머지 간선 중에서 가중치가 가장 작은 간선 (A,D)를 삽입하면 A-B-D의 사이클이 생성되므로 삽입할 수 없다. 그 다음으로 가중치가 가장 작은 간선 (C,F) 삽입



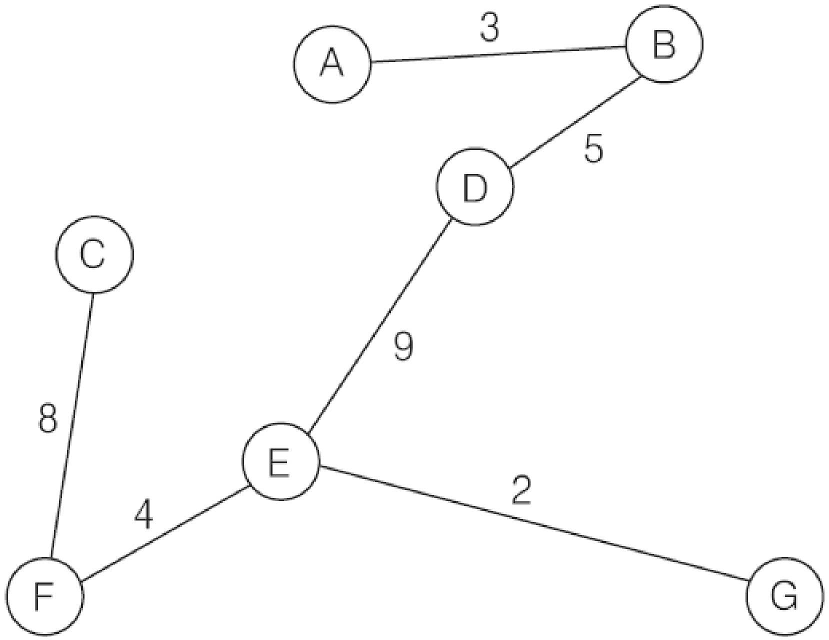
가중치	간선
2	(E, G)
3	(A, B)
4	(E, F)
5	(B, D)
~~6~~	~~(A, D)~~
8	(C, F)
9	(D, E)
10	(C, E)
12	(B, G)
14	(F, G)
17	(A, C)

삽입한 간선의 수 : 5개

❑ 최소 비용 신장 트리 – Kruskal 알고리즘

⑥ 나머지 간선 중에서 가중치가 가장 작은 간선 (D,E) 삽입

- 트리 간선 집합 T가 6개의 간선을 포함하므로 최소 비용 신장 트리 완성



가중치	간선
2	(E, G)
3	(A, B)
4	(E, F)
5	(B, D)
~~6~~	~~(A, D)~~
8	(C, F)
9	(D, E)
10	(C, E)
12	(B, G)
14	(F, G)
17	(A, C)

삽입한 간선의 수 : 6개

❑ 최소 비용 신장 트리 – Prim 알고리즘

❖ Prim 알고리즘

- 하나의 정점에서 시작하여 모든 정점이 포함될 때까지 최소 비용 신장 트리를 확장해 나간다.
- Prim 알고리즘 수행 과정

(1) 그래프 G에서 시작 정점을 선택하여 트리 정점 집합 S에 삽입한다.

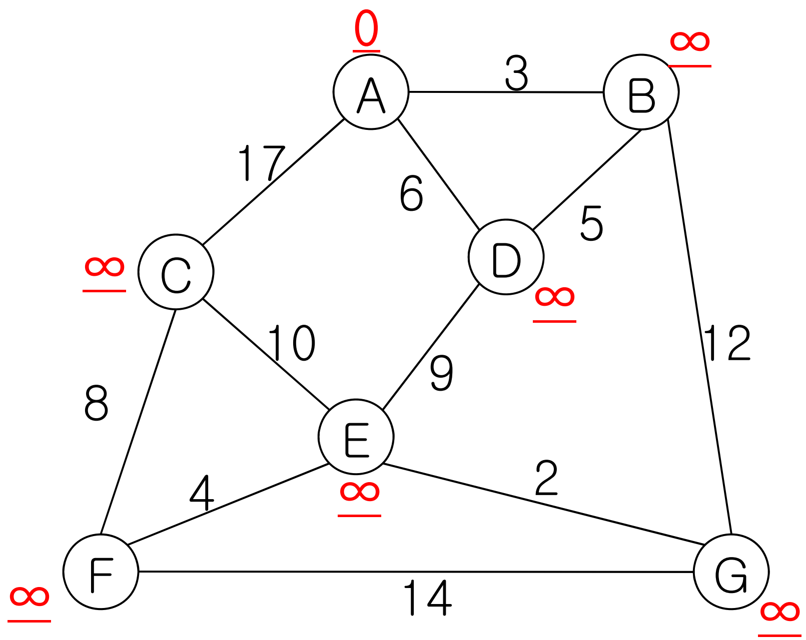
(2) 트리 정점과 트리 밖 정점에 부속된 간선들 중에서 가중치가 가장 작은 간선을 선택하여 트리를 확장한다.

이 때 선택된 간선의 한쪽 끝인 트리 밖 정점을 S에 삽입한다.

(3) S가 n 개의 정점을 포함할 때까지 (2)를 반복한다.

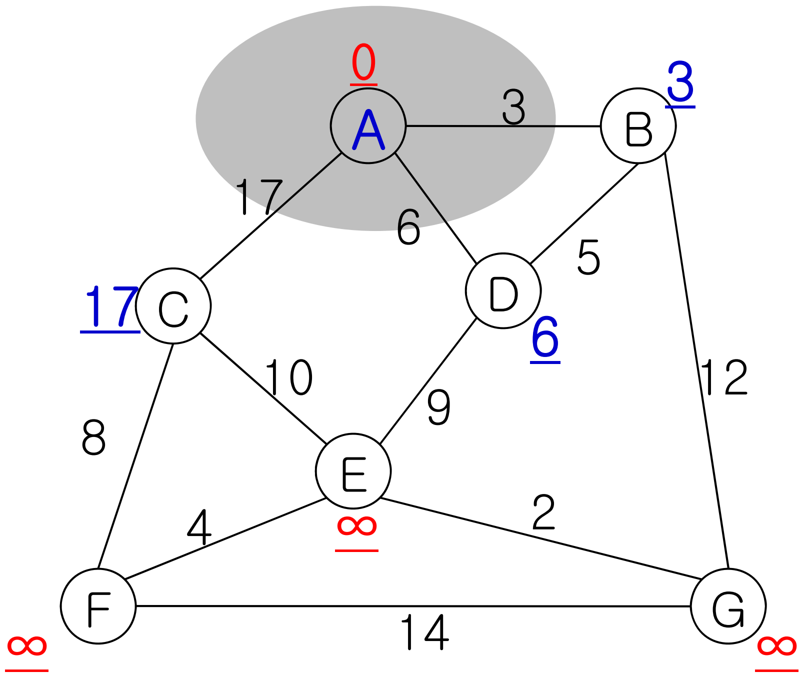
❑ 최소 비용 신장 트리 – Prim 알고리즘

- 예) G10의 최소 비용 신장 트리 구하기 – Prim 알고리즘
- 초기 상태
  - A를 시작 정점으로 선택
  - 트리 밖 각 정점 v를 트리 정점과 연결하는 간선의 가중치 중 최소값(이 값을 d[v]라고 부르자)을 ∞로 초기화
  - 시작 정점 A에 대한 d값은 0으로 초기화



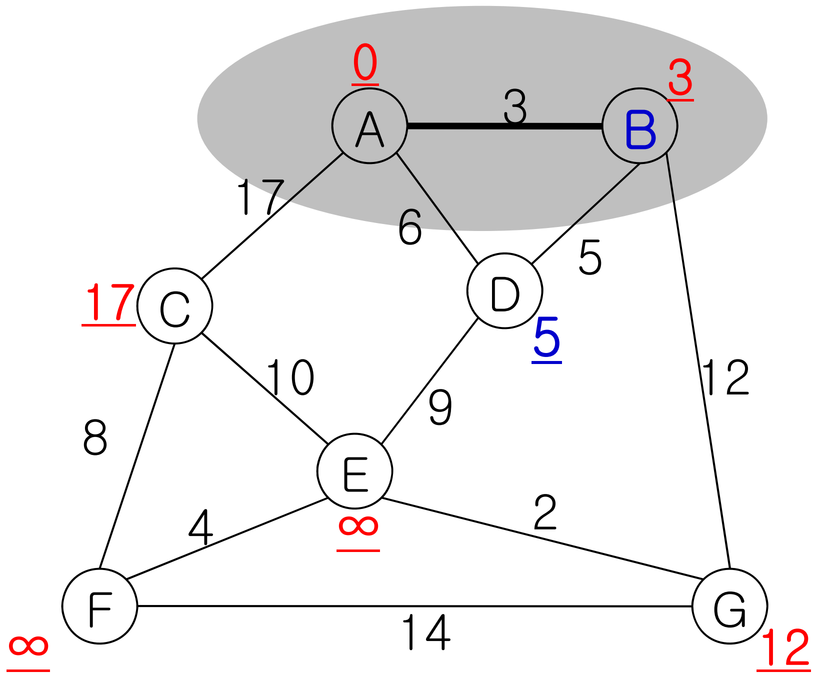
❑ 최소 비용 신장 트리 – Prim 알고리즘

① d 값이 가장 작은 트리 밖 정점(즉, A)을 골라 트리 정점 집합 T에 삽입하고, A와 인접한 트리 밖 정점의 d 값을 조정한다.



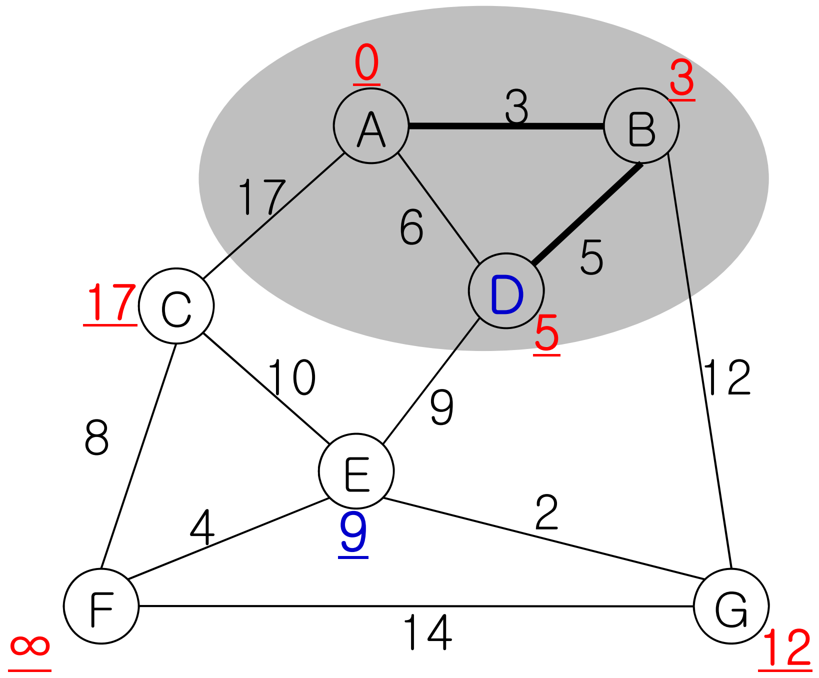
❑ 최소 비용 신장 트리 – Prim 알고리즘

② d 값이 가장 작은 트리 밖 정점(즉, B)을 골라 트리 정점 집합 T에 삽입하고, B와 인접한 트리 밖 정점의 d 값을 조정한다.



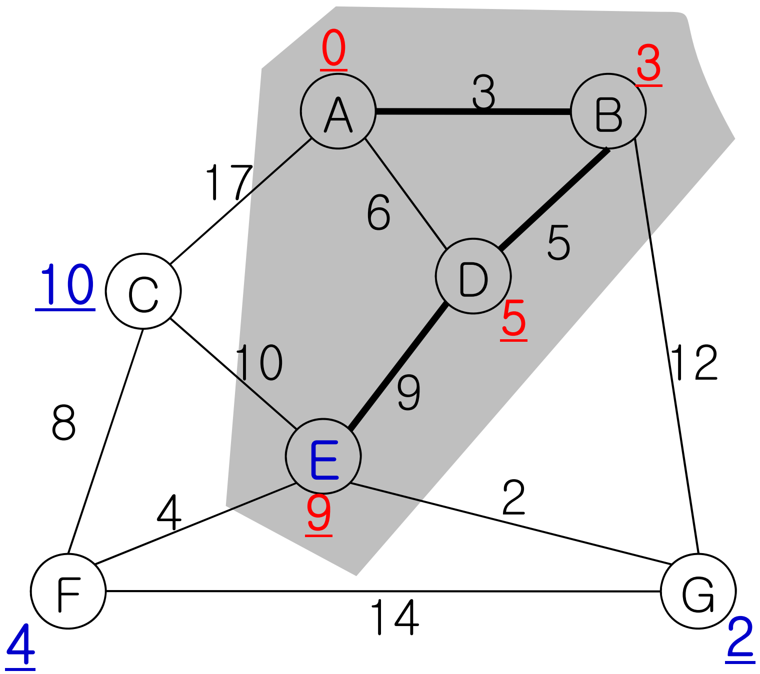
❑ 최소 비용 신장 트리 – Prim 알고리즘

③ d 값이 가장 작은 트리 밖 정점(즉, D)을 골라 트리 정점 집합 T에 삽입하고, D와 인접한 트리 밖 정점의 d 값을 조정한다.



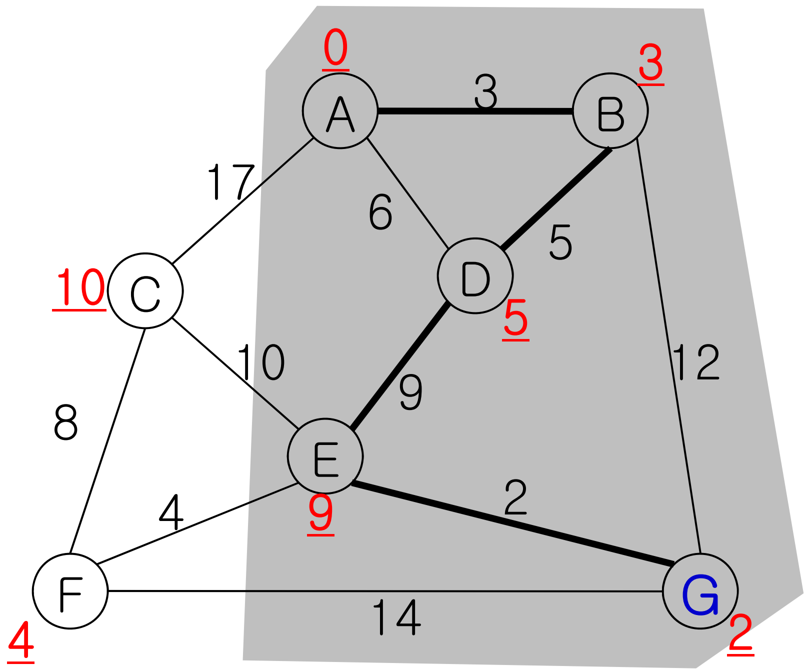
❑ 최소 비용 신장 트리 – Prim 알고리즘

④ d 값이 가장 작은 트리 밖 정점(즉, E)을 골라 트리 정점 집합 T에 삽입하고, E와 인접한 트리 밖 정점의 d 값을 조정한다.



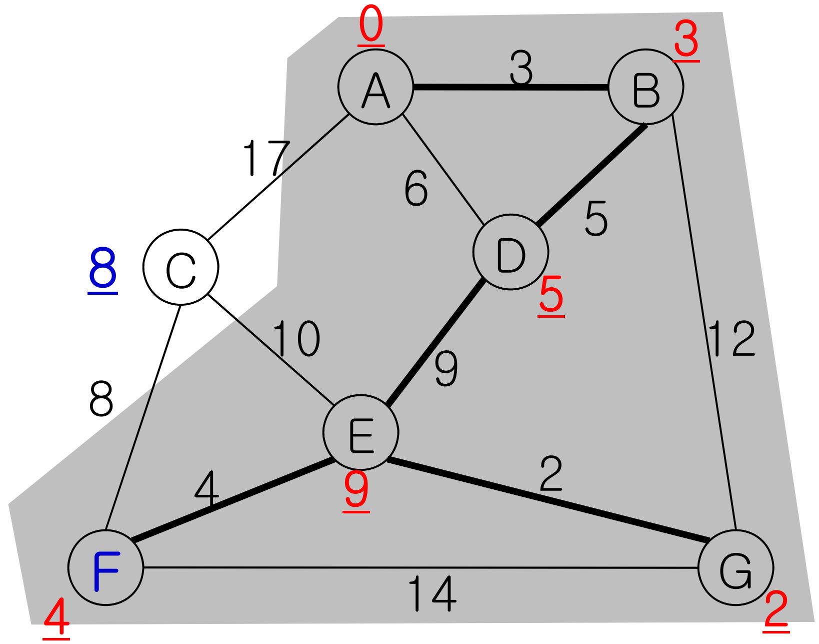
❑ 최소 비용 신장 트리 – Prim 알고리즘

⑤ d 값이 가장 작은 트리 밖 정점(즉, G)을 골라 트리 정점 집합 T에 삽입하고, G와 인접한 트리 밖 정점의 d 값을 조정한다.



❑ 최소 비용 신장 트리 – Prim 알고리즘

⑥ d 값이 가장 작은 트리 밖 정점(즉, F)을 골라 트리 정점 집합 T에 삽입하고, F와 인접한 트리 밖 정점의 d 값을 조정한다.



❑ 최소 비용 신장 트리 – Prim 알고리즘

⑦ d 값이 가장 작은 트리 밖 정점(즉, C)을 골라 트리 정점 집합 T에 삽입하면 알고리즘 종료