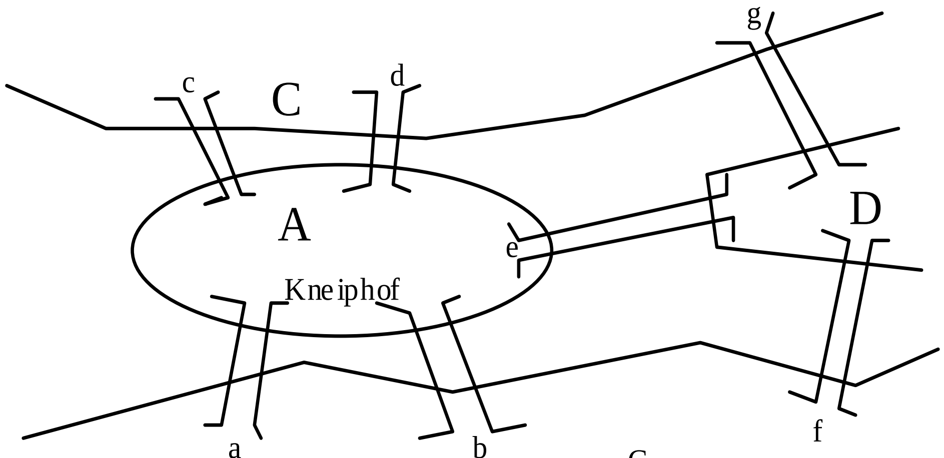
제 6 장 그래프(Graph)

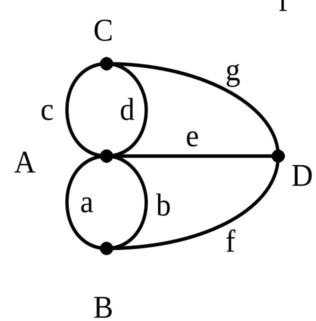
6.1 그래프 추상 데이터 타입

(1) 개요

- **그래프 사용의 예:** Konigsberg **문제 – 오일러 행로**



(a)

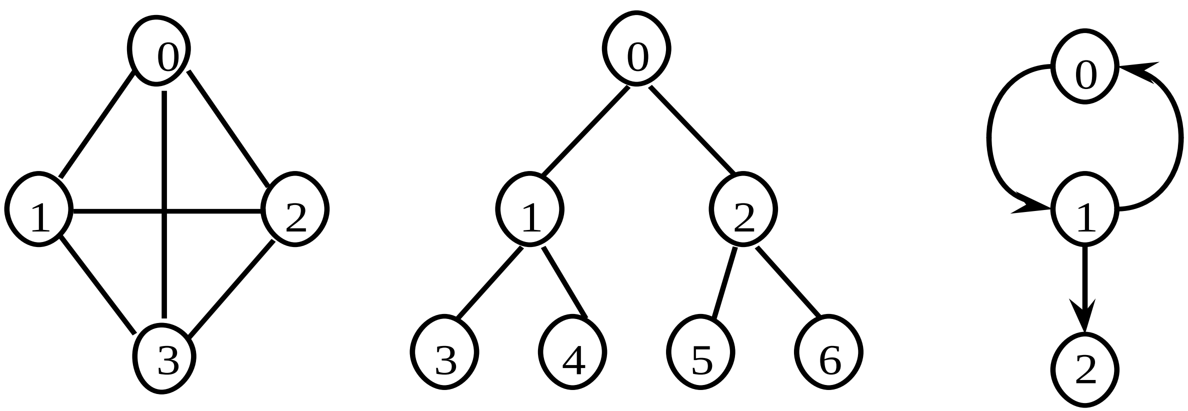


(b)

그래프의 정의

(2) **정의**

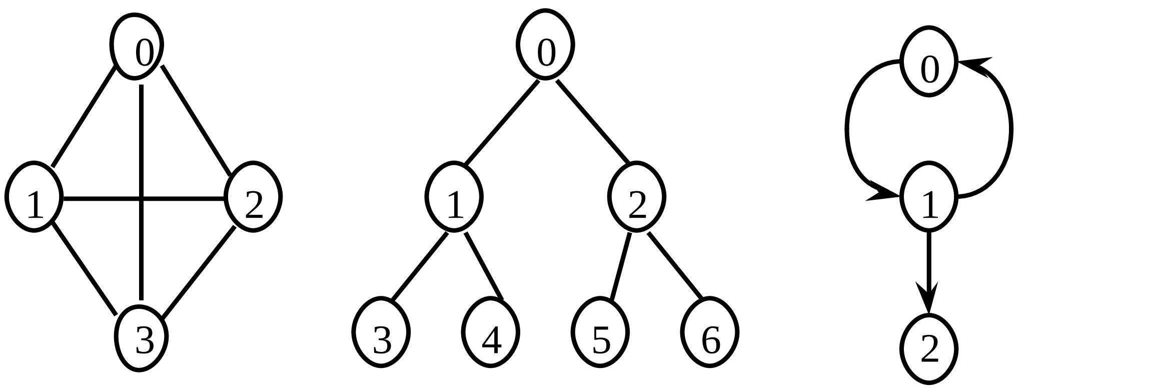
- G = (V, E)
  V: **공집합이 아닌 정점**(vertex) **들의 유한집합**
  E: **정점들의 쌍인 간선**(edge)**들의 집합**
- **예**: **그림** 6.2



(a) G_1 (b) G_2 (c) G_3

무방향그래프와 방향그래프

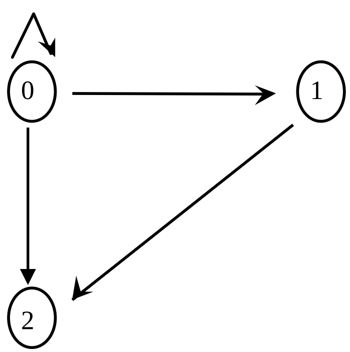
- **무방향그래프**: G_1, G_2
  $V(G_1) = \{ 0, 1, 2, 3 \}$
  $E(G_1) = \{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)\}$
- **방향그래프**: G_3
  $V(G_3) = \{ 0, 1, 2 \}$
  $E(G_3) = \{<0,1>, <1,0>, <1,2>\}$
  $<u, v>$**에서** u: **꼬리**, v: **머리**



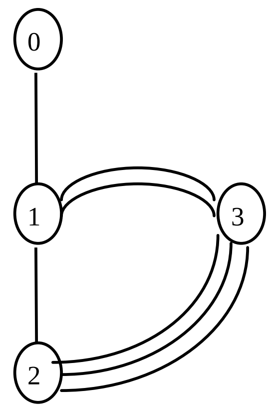
(a) G_1 (b) G_2 (c) G_3

용어 및 정의

- **일반적으로 그래프는 자기 루프나 중복 간선을 가질 수 없다.**



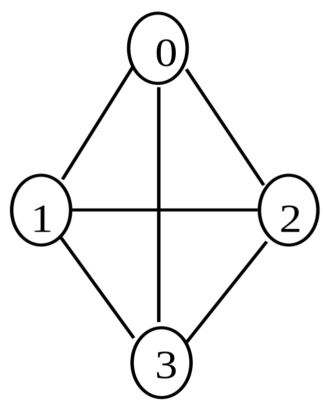
(a) 자기 간선을 가진 그래프



(b) 다중 그래프

완전 그래프

- **n개의 정점을 갖는 무방향그래프에서 간선의 최대수:** n(n-1)/2
- **최대 간선수를 갖는 그래프를 완전 그래프**(complete graph)**라 한다.**(그림 6.2(a))

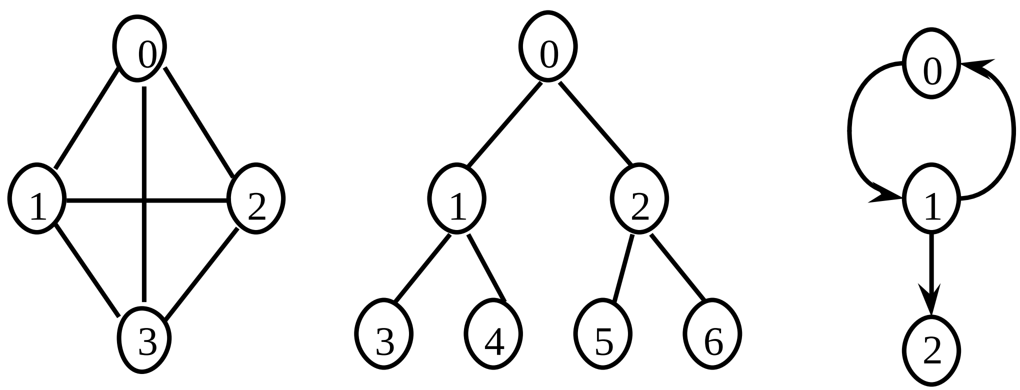


(a) G_1

인접과 부속

- 만약 (u, v)가 E(G)의 간선이라면, u 와 v 는 인접한다(adjacent) 라고 하며 간선 (u, v)는 정점 u 와 v 에 부속된다(incident) 고 한다.

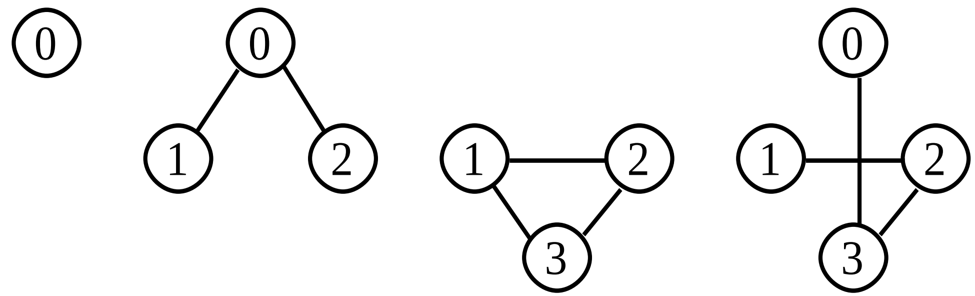
- 만약 <u, v>의 경우, u 는 v 에 인접하다고 하고, v 는 u 로부터 인접한다고 한다. 그리고 간선 <u, v>는 정점 u 와 v 에 부속된다.



(a) G_1 (b) G_2 (c) G_3

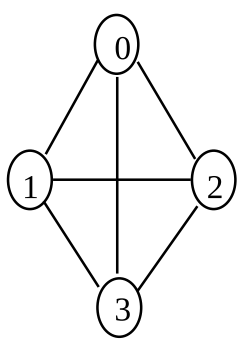
부분 그래프(subgraph)

- 부분 그래프: G'
  V(G') ⊆ V(G) 와
  E(G') ⊆ E(G)

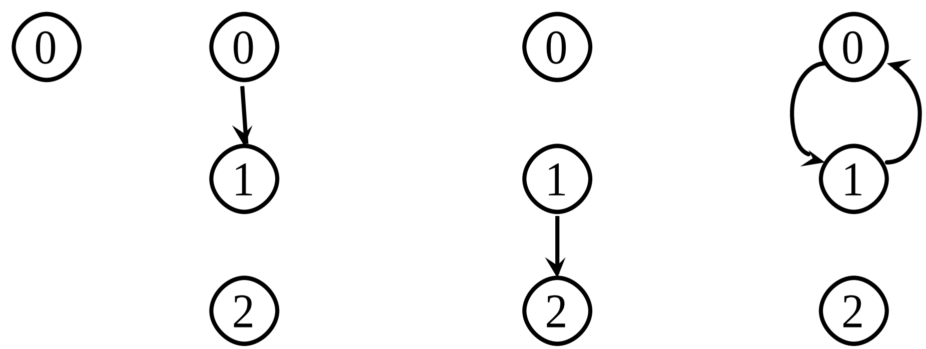


(i) (ii) (iii) (iv)

(a) G_1의 부분 그래프들



(a) G_1



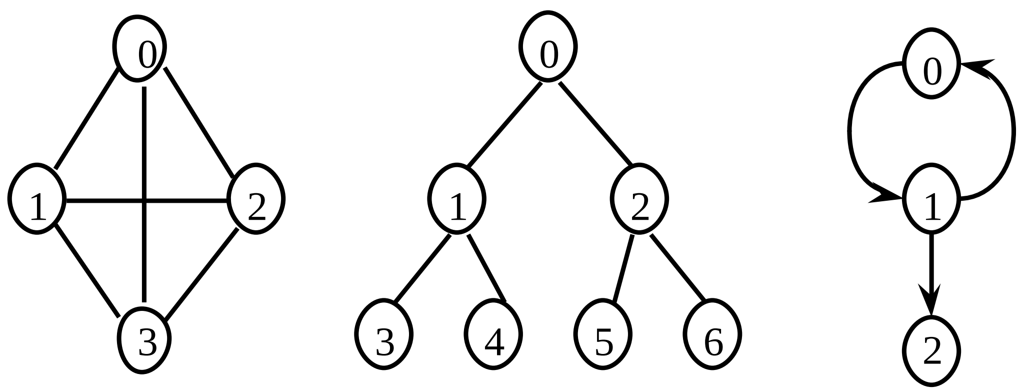
(i) (ii) (iii) (iv)

(b) G_3의 부분 그래프들

경로(path)

- **경로:** **그래프에서** $(u, i_1),(i_1, i_2),\ldots,(i_k, v)$**를** $E(G)$**에 속한 간선들이 라고 할 때 정점** u **에서부터 정점** v **까지의 경로란 정점들** $u, i_1, i_2, \ldots, i_k, v$ **를 의미한다.**

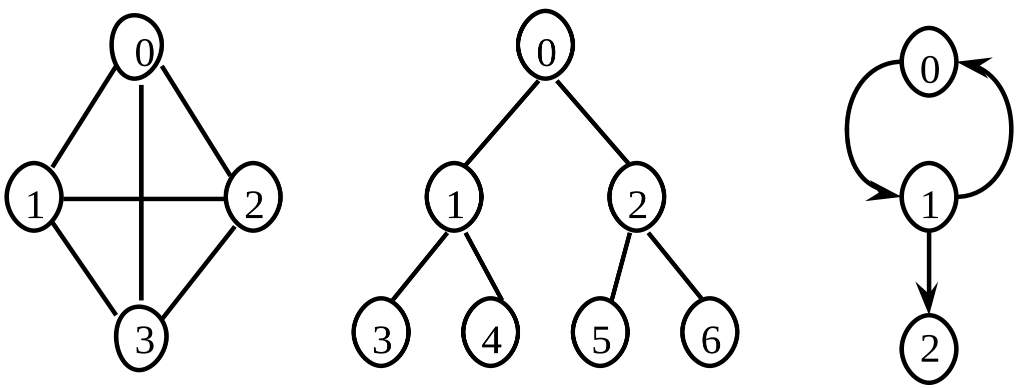
  방향 그래프인 경우, $<u, i_1>,<i_1, i_2>,\ldots,<i_k, v>$ **로 구성된다.**
- **경로의 길이: 경로 상에 있는 간선들의 수**
- 예: G_1에서 0−1−2−0−3, 0−1−2−3, G_3에서 0−1−2



(a) G_1 (b) G_2 (c) G_3

단순 경로와 사이클

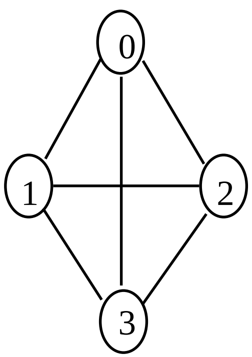
- **단순 경로**(simple path): **처음과 마지막을 제외한 모든 정점들이 서로 다른 경로**
- **단순 방향 경로**(simple directed path): **방향 그래프에서의 단순 경로**
- **사이클**(cycle): **처음과 마지막이 같은 단순 경로**
- 예: 단순경로: G_1에서 0-1-2-3, G_3에서 0-1-2
  사이클 : G_1에서 0-1-2-3-0, G_3에서 0-1-0



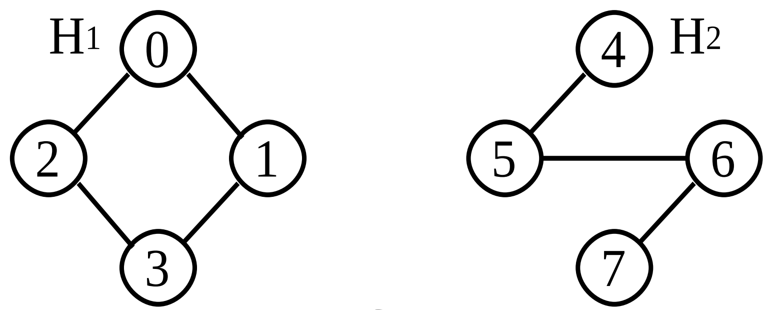
(a) G_1　　(b) G_2　　(c) G_3

연결 그래프(connected graph)

- **연결**(connected): **무방향 그래프** G**에서 정점** u**로부터** v**까지의 경로가 있으면**, <u>u**와** v**는 연결되었다**</u>**고 한다**.
- **그래프** G**에서 임의의 두 정점 사이에 경로가 존재하면**, G**를 연결그래프라** 한**다**.
- **예**: G_1: 연결 그래프, G_4: 비연결 그래프



(a) G_1

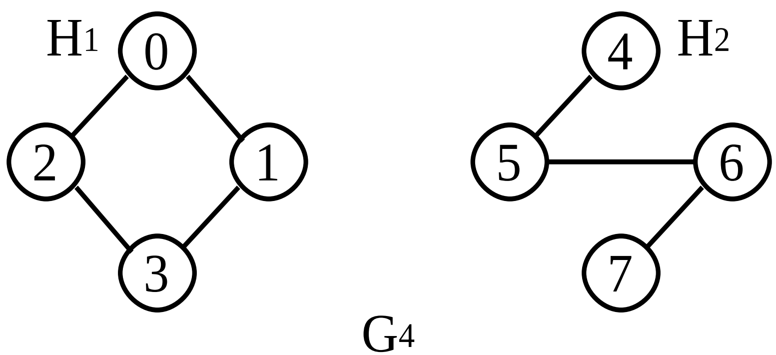


G_4

연결 요소(conneted component)

- **연결 요소: 무방향 그래프에서 최대 연결된 부분 그래프**

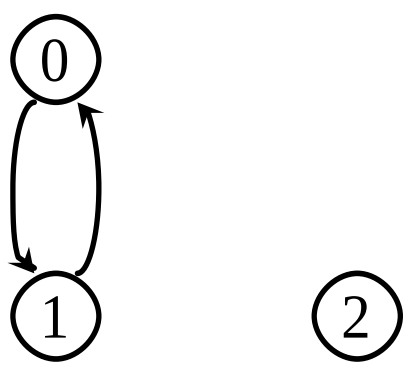
  예: $H_1 = \{0, 1, 2, 3\}$, $H_2 = \{4, 5, 6, 7\}$



강력 연결 요소

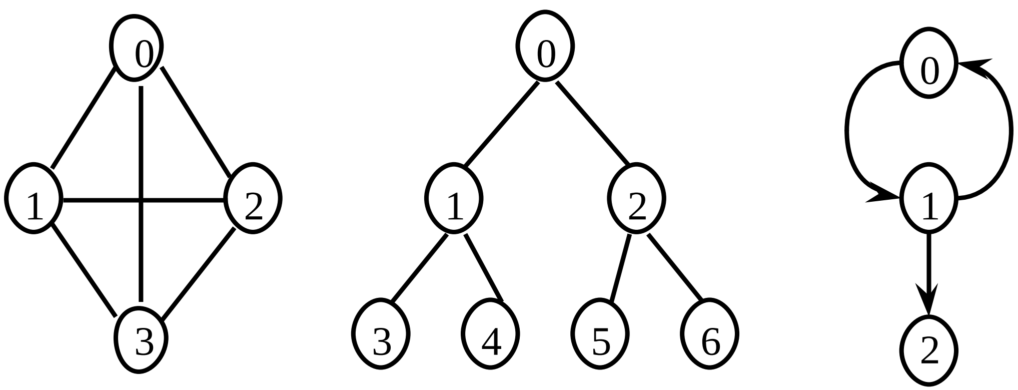
- **강력 연결**(strongly connected): **방향 그래프의 두 정점** u**와** v**사**이**에서**, u**에서** v**로**, v**에서** u**로의 방향 경로가 존재하면** u**와** v**는 강력 연결되었다고 한다**.
- **방향 그래프** G**에서 임의의 두 정점이 강력 연결되어 있다면** G**를 강력 연결 그래프라 한다**.
- **강력 연결 요소**: **방향 그래프에서 최대 강력 연결된 부분 그래프**

  예: {0, 1}, {2}



정점의 차수

- **정점의 차수**(degree): **정점에 부속한 간선들의 수**
- **방향 그래프에서는 진입 차수**(indegree)**와 진출 차수** (outdegree)**로 구분한다**.
- **간선의 수를** e , **정점** i **의 차수를** d_i **라 할 때**, $\sum d_i = 2e$**이다**.
- 예: G_1에서 $d_2 = 3$, G_2에서 $d_2 = 3$,
  G_3에서 진입차수 $d_1 = 1$, 진출차수 $d_1 = 2$



(a) G_1 (b) G_2 (c) G_3

(3) 그래프 표현법

1) **인접행렬**(adjacency matrix)

- G=(V, E)**에서 정점의 수가** n**일 때**, G**의 인접행렬은** n x n **의** 2**차원 배열이다**.

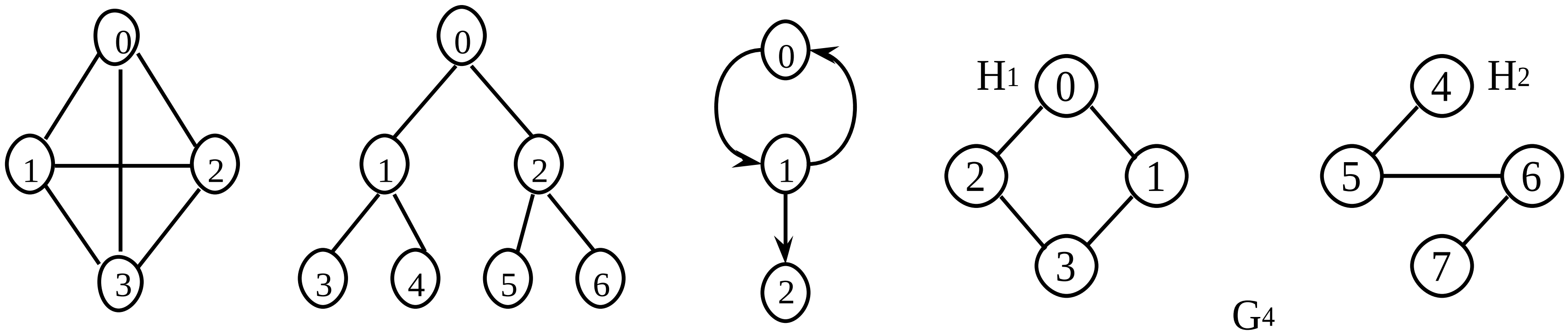
  무방향 그래프: **만약** (i, j)**가 간선이면**, A[i][j] = 1,

  그렇지 않으면, A[i][j] = 0.

  방향 그래프: **만약** <i, j>**가 간선이면**, A[i][j] = 1,

  그렇지 않으면, A[i][j] = 0.

인접 행렬 표현의 예



(a) G_1　　(b) G_2　　(c) G_3

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

(a) G_1

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

(b) G_3

$$
\begin{array}{cc}
\begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}
\end{array}
$$

(c) G_4

인접 행렬의 성질

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

(a) G_1

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

(b) G_3

$$\begin{array}{cc} \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

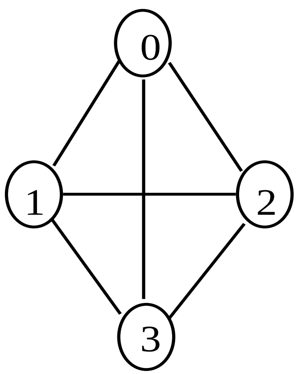
(c) G_4

- 무방향 그래프에서 $\sum_{j=0}^{n-1} A[i][j]$ = 정점 i 의 차수
- 방향 그래프에서 $\sum_{j=0}^{n-1} A[i][j]$ = 정점 i 의 진출 차수

인접 리스트(adjacency list)

2) 인접리스트

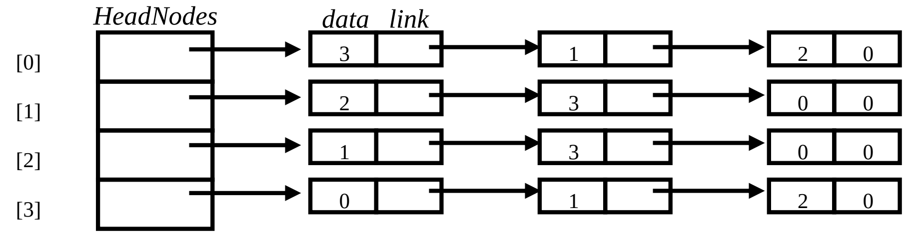
- 인접행렬의 n행들을 n개의 연결리스트로 표현한다.



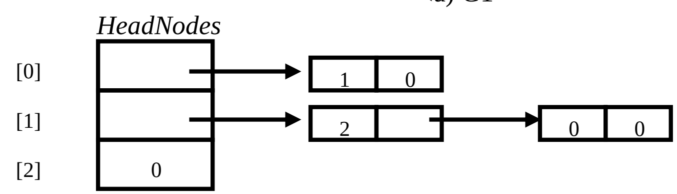
(a) G_1

$$
\begin{array}{c}
 \\ 0 \\ 1 \\ 2 \\ 3
\end{array}
\begin{array}{c}
\begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\
\left(\begin{array}{cccc}
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0
\end{array}\right)
\end{array}
$$

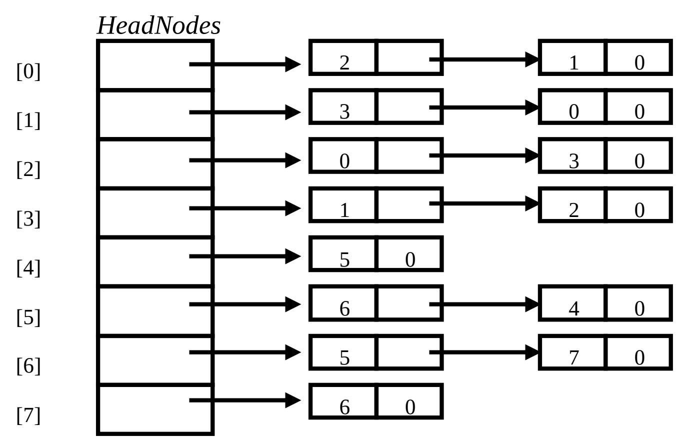
(a) G_1



(a) G1



(b) G3



c) G4

인접리스트

인접 리스트의 클래스 선언

- 클래스 선언

```
typedef ChainNode* ChainNodePointer ;
class Graph {
  private:
    ChainNodePointer *HeadNodes;  // first 들의 배열 선언
    int n;                          // 정점의 수
  public:
    Graph(const int vertices = 0): n(vertices)
    { HeadNodes = new ChainNodePointer[n];};
}
```

이때 HeadNode[i]가 i 번째 정점 리스트의 first 역할
(즉, 첫 노드를 가리킨다)

인접 행렬과 인접 리스트의 특징

- **인접행렬을 사용하는 알고리즘들은 행렬 내의 n^2 개의 항들을 조사해야 하므로 적어도 $O(n^2)$의 시간이 필요하다. 희소 그래프의 경우 인접행렬의 항들이 대부분 0 이므로 인접행렬을 사용하면 불리하게 된다. 따라서 인접리스트를 사용하는 것이 좋다.**

- **인접리스트의 경우, 무방향 그래프에서 n개의 헤드노드와 2e개의 리스트 노드가 필요하므로, 그래프의 모든 간선과 정점들을 확인하는데 $O(n+e)$ 시간이 소요된다.**

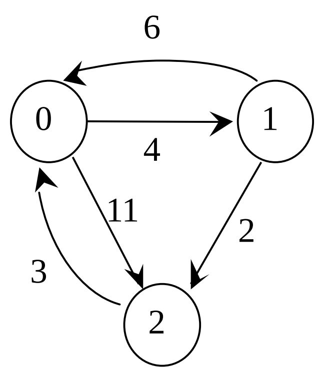
가중치 그래프

- 가중치 그래프: 간선에 가중치를 갖는 그래프
- 표현 방법:

1) **인접 행렬**을 사용할 **경우**, 간선이 존재함을 나타내는 1 **대신 가중치**를 저장하고, 간선이 없는 경우, 응용에 따라 0을 저장하거나 ∞(아주 큰 값)를 저장한다.

2) **인접 리스트**를 사용할 **경우**, 각 **노드에** weight **필드를 추가**하여 가중치를 이 필드에 저장한다.

- 가중치 그래프를 나타내는 인접행렬의 사용 예:



(a) 예제 방향그래프

A	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

6.2 기본적인 그래프 연산

(1) 깊이 우선 탐색(DFS: Depth First Search)

- 연결 그래프, 무방향 그래프

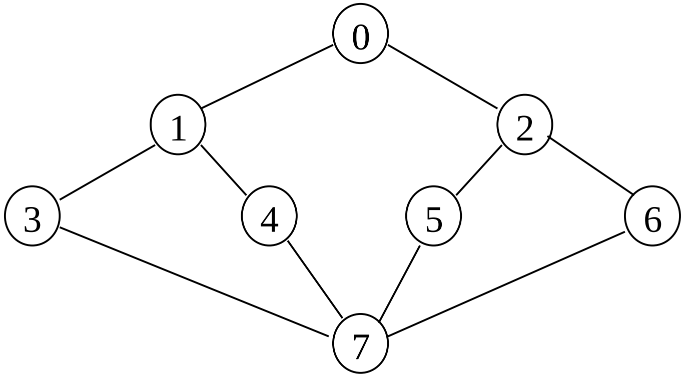
- 단계:

1. 출발 정점 v를 방문
2. v에 인접하고 방문하지 않은 한 정점 w를 선택
3. w를 시작점으로 다시 깊이 우선 탐색 시작
4. 모든 인접 정점을 방문한 정점 u에 도달하면, u를 방문하기 위해 사용된 간선 (w, u)상의 정점 w로 되돌아감
5. 정점 w로부터 다시 깊이 우선 탐색 시작
6. 방문이 안된 정점으로 더 이상 갈 수 없을 때 종료

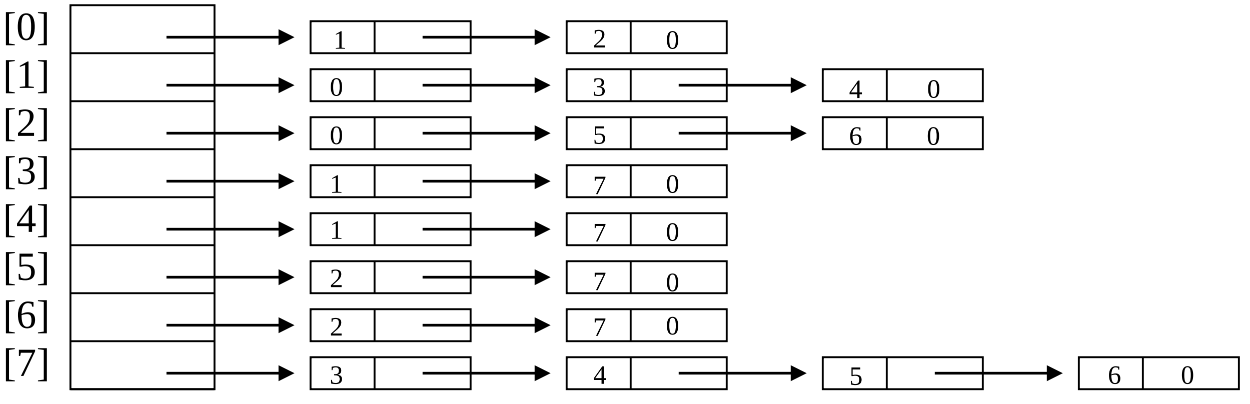
- 스택을 사용

깊이 우선 탐색의 예

- 예제 6.1: 0 – 1 – 3 – 7 – 4 – 5 – 2 – 6 순으로 방문



(a)



(b)

깊이 우선 탐색 알고리즘

```
void Graph::DFS() // 드라이버
{
   visited = new bool[n];       // visited를 Graph의 Boolean* 데이타 멤버로
                                 // 선언.
   for(int i=0; i<n; i++) visited[i] = false; // 초기에는 방문된 정점이 없음
   DFS(0); // 정점 0에서 탐색을 시작
   delete[] visited;
}
void Graph::DFS(const int v) // 실제 탐색 수행
// 정점 v에서 도달 가능하면서 아직 방문되지 않은 모든 정점들을 방문
{
   visited[v] = true;
   for(v에 인접한 각 정점 w에 대해) // 실제 코드는 그래프 표현 방법에 좌우됨
      if(!visited[w]) DFS(w);
}
```

깊이 우선 탐색 알고리즘의 분석

- **DFS**의 분석**:** 시간 복잡도

  인접 리스트를 사용할 때 - 리스트 노드를 한번씩 조사해야 하므로
  리스트 노드의 수는 **2e** 개 이므로 **O(e)** 시간 걸린다.

  인접 행렬을 사용할 때 - 한 정점에 인접한 정점들을 조사하기 위해
  O(n) 시간**,** 따라서 전체 원소들을 조사해야 하므로
  $O(n^2)$ 시간 걸린다.

너비 우선 탐색

(2) 너비 우선 탐색**(BFS: Breadth First Search)**

- 연결 그래프**,** 무방향 그래프

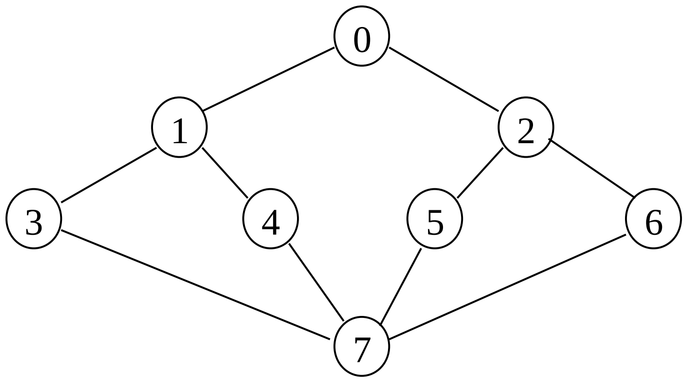
- 단계:

1. 시작 정점 **v**를 방문
2. v에 인접한 모든 정점들을 방문
3. 새롭게 방문한 정점들에 인접하면서 아직 방문하지 못한 정점들을 방문
4. 더 이상 방문할 정점이 없으면 종료

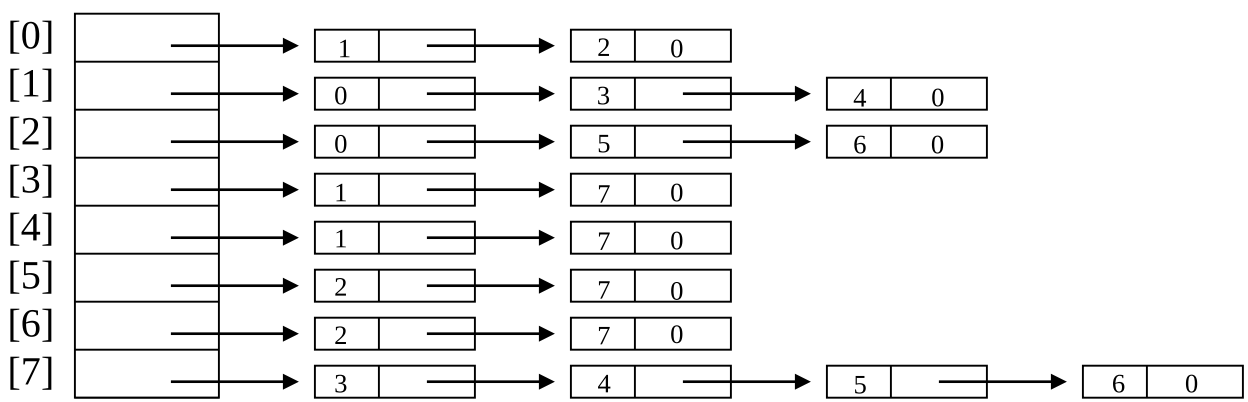
- 큐를 사용

너비 우선 탐색의 예

- 예제 6.2: 0 – 1 – 2 – 3 – 4– 5 – 6 – 7 순으로 방문



(a)



(b)

너비 우선 탐색 알고리즘

```
void Graph::BFS(int v)
// 정점 v에서 시작하여 너비 우선 탐색을 수행.
// v 방문시 visited[i]는 TRUE가 됨. 이 알고리즘은 큐를 사용함
{
   visited = new bool[n];
   for(int i=0; i<n; i++) visited[i] = false; // 초기에는 방문한 정점이 없음
   visited[v] = true;
   Queue<int> q; // q는 큐임
   q.Push(v);      // 정점을 큐에 삽입
   while(!q.IsEmpty()) {
     v = *q.Pop(v);    // 정점 v를 큐에서 삭제
     for(v에 인접한 모든 정점 w에 대해) // 실제 코드는 그래프 표현 방법에 좌우됨
       if(!visited[w]) {
          q.Push(w);
          visited[w] = true;
       }
   } // while 루프의 끝
   delete [] visited;
}
```

너비 우선 탐색 알고리즘의 분석

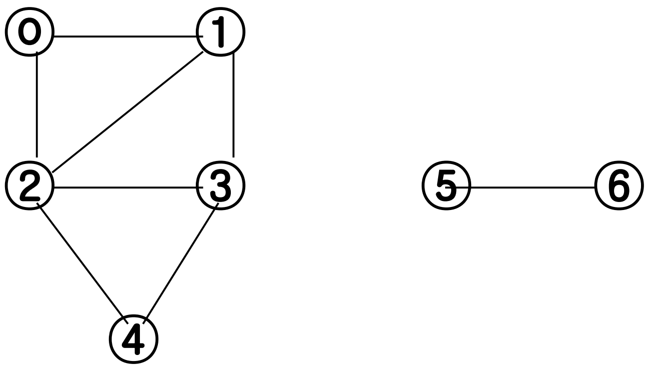
- BFS의 분석: 시간 복잡도

  인접 리스트를 사용할 때 – 리스트 노드를 한번씩 조사해야 하므로 리스트 노드의 수는 2e 개 이므로 O(e)시간 걸린다.

  인접 행렬을 사용할 때 – 한 정점에 인접한 정점들을 조사하기 위해 O(n) 시간, 따라서 전체 원소들을 조사해야 하므로 $O(n^2)$ 시간 걸린다.

(3) 연결 요소

- 예:



{0, 1, 2, 3, 4} {5, 6}

- 방문되지 않은 1개의 정점에서 DFS나 BFS를 사용하여 방문되는 정점들을 1개의 연결 요소로 결정한다.
- 방문되지 않은 정점들이 없을 때까지 이 과정을 반복한다.

연결 요소 알고리즘

```
void Graph::Components()
// 그래프의 연결요소들을 결정
{
 // visited는 Graph의 Boolean* 데이타 멤버로 선언되는 것으로 가정.
   visited = new bool[n];
   for(int i=0; i<n; i++) visited[i] = false;
   for(i=0; i<n; i++)
     if(!visited[i]) {
       DFS(i); // 하나의 요소를 발견
       OutputNewComponent();
     }
   delete [] visited;
}
```

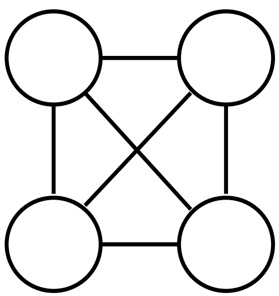
연결 요소 알고리즘의 분석

- 분석:

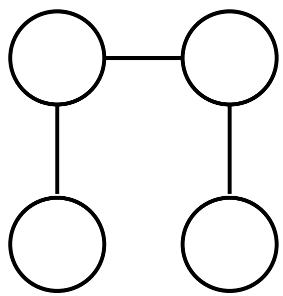
G가 인접리스트인 경우, 모든 노드들을 1번씩 방문하므로 O(e)시간 걸리고, for 반복문에 O(n)시간이 걸리므로 총 시간은 O(n+e)이다. G가 인접행렬로 표현된 경우, $O(n^2)$ 시간 걸린다.

(4) 신장 트리

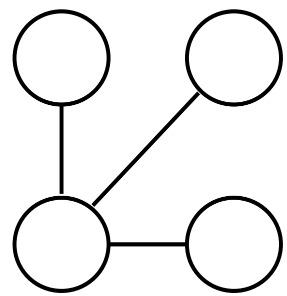
- 신장 트리: 연결 그래프 **G**에서 모든 정점들을 포함하며 트리인 부분 그래프



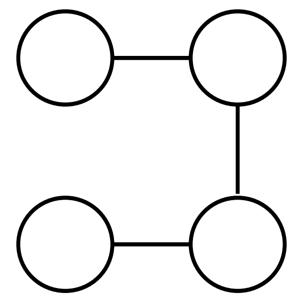
초기 그래프



신장 트리-1



신장 트리-2

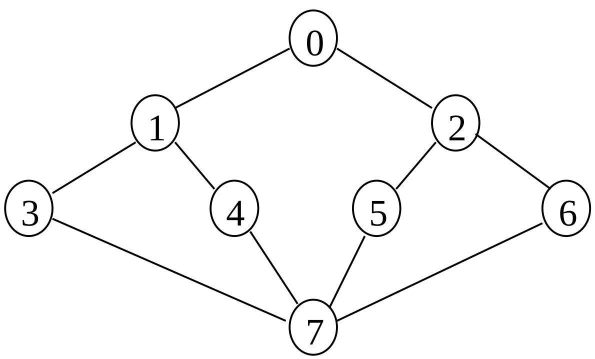


신장 트리-3

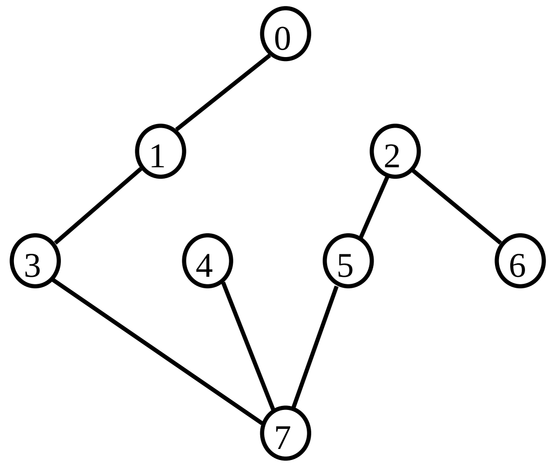
- 방법: 임의의 정점에서 출발하여 **DFS** 또는 **BFS**로 그래프를 탐색할 때, 정점 **v**에서 인접하며 방문되지 않은 정점 **w**가 존재할 때, 간선 **(v, w)**의 집합을 구한다.

깊이 우선과 너비 우선 신장 트리

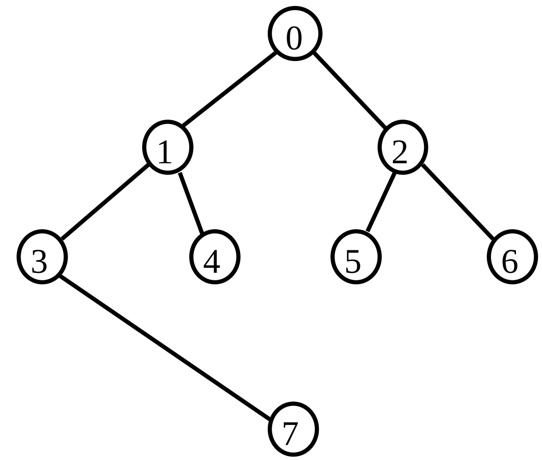
- 깊이 우선 신장 트리: **DFS**를 이용
  너비 우선 신장 트리: **BFS**를 이용



(a)



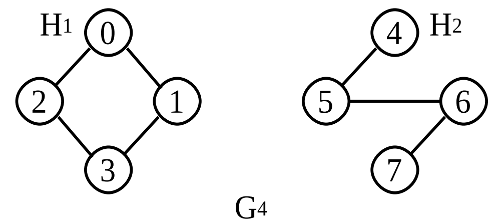
(a) *DFS*(0) 신장 트리

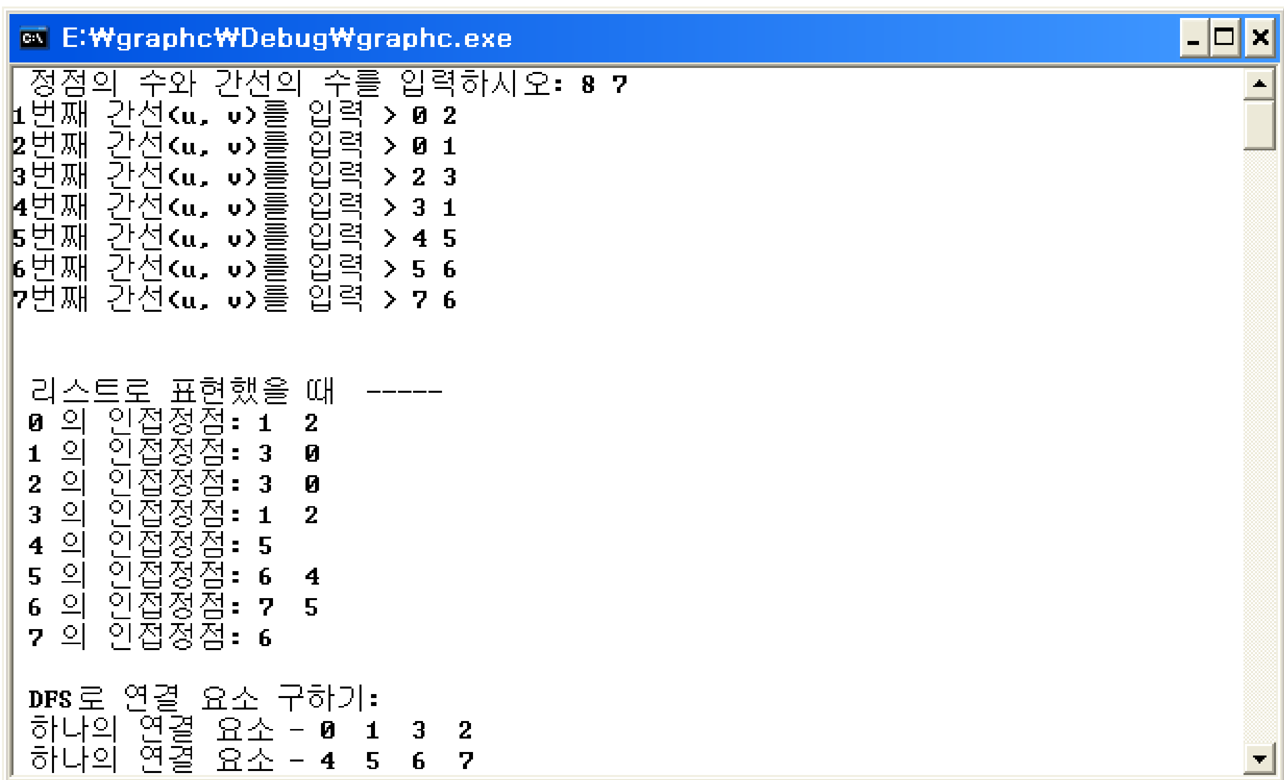


(b) *BFS*(0) 신장 트리

예제 프로그램

- ftp 사이트의 graph 폴더
- 그래프를 인접 리스트로 표현하며, 연결 요소를 출력한다.



- 

E:₩graphc₩Debug₩graphc.exe

```
 정점의 수와 간선의 수를 입력하시오: 8 7
1번째 간선<u, v>를 입력 > 0 2
2번째 간선<u, v>를 입력 > 0 1
3번째 간선<u, v>를 입력 > 2 3
4번째 간선<u, v>를 입력 > 3 1
5번째 간선<u, v>를 입력 > 4 5
6번째 간선<u, v>를 입력 > 5 6
7번째 간선<u, v>를 입력 > 7 6


 리스트로 표현했을 때  -----
 0 의 인접정점: 1  2
 1 의 인접정점: 3  0
 2 의 인접정점: 3  0
 3 의 인접정점: 1  2
 4 의 인접정점: 5
 5 의 인접정점: 6  4
 6 의 인접정점: 7  5
 7 의 인접정점: 6

 DFS로 연결 요소 구하기:
 하나의 연결 요소 - 0  1  3  2
 하나의 연결 요소 - 4  5  6  7
```

예제 프로그램의 main 함수

```
void main()
{
   int n, e ;  // n: 정점의 수, e: 간선의 수
   int k, u, v ;
   cout << " 정점의 수와 간선의 수를 입력하시오: ";
   cin >> n >> e ;
   Graph g(n);  // Graph 클래스의 객체 g 를 생성

   for(int i=0; i<e; i++) {
        k = i+1;
        cout << k << "번째 간선(u, v)를 입력 > ";
        cin >> u >> v ;
        g.InsertEdge(u, v);    // 인접 리스트에 간선 (u, v)를 삽입한다.
   }
   g.PrintVertex();  // 입력된 그래프의 연결리스트 출력
   g.Components();   // 입력된 그래프의 연결 요소 구하기
   cout << "₩n₩n";
}
```