네번째 강의

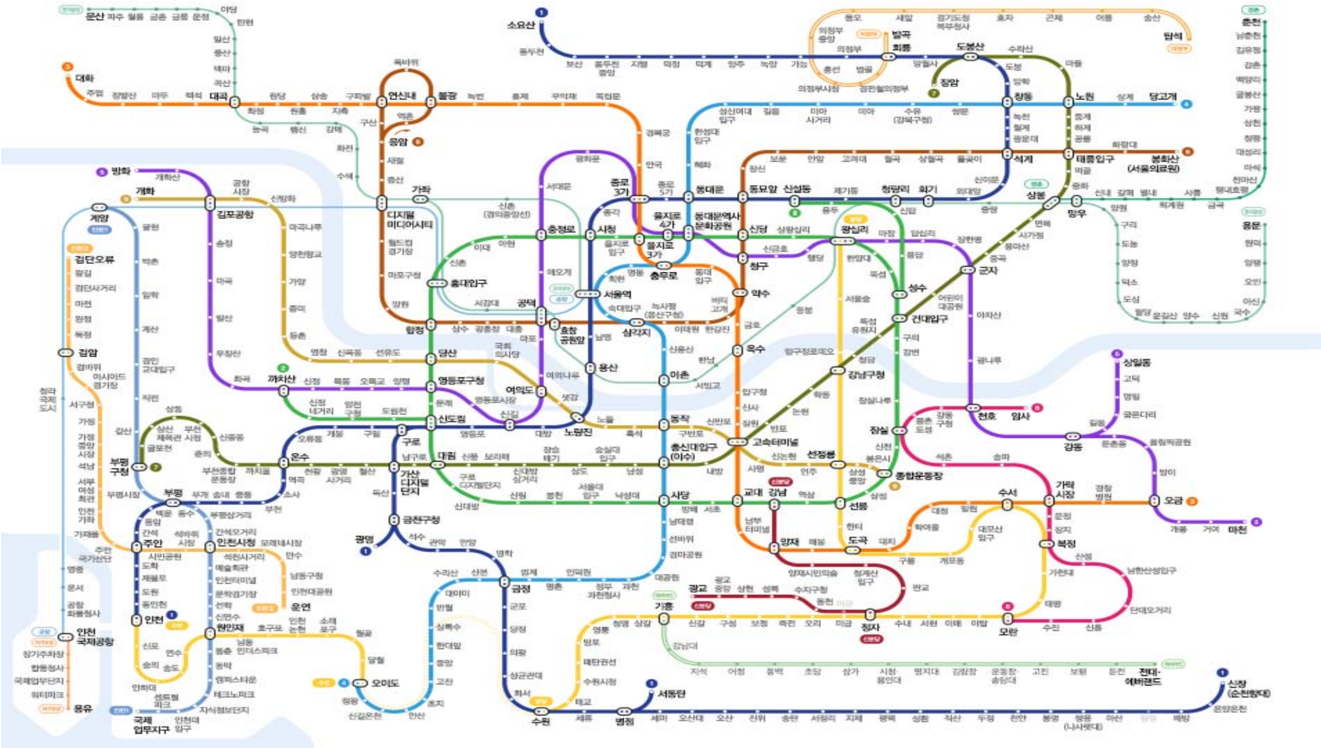
그래프

목차

- 그래프(**Graph**)의 정의
- 그래프의 표현법
- 그래프에 적용가능한 연산
- 실습
- 스스로 프로그래밍
- 첨삭 지도

1. 그래프의 정의

- 그래프(graph)란?
  - 연결되어 있는 객체 간의 관계를 표현하는 자료구조
  - 그래프의 예: 지하철 노선도, 친구 관계, 컴퓨터 네트워크 등



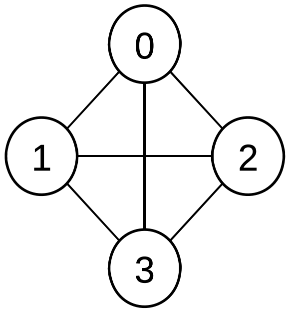
그래프 데이터 타입

- 그래프 G의 두 가지 구성 요소
  - V(G): G에 포함된 vertex(정점)들의 집합
  - E(G): G에 포함된 edge(간선, 에지)들의 집합
  - G = (V, E) 라고 쓰기도 함.

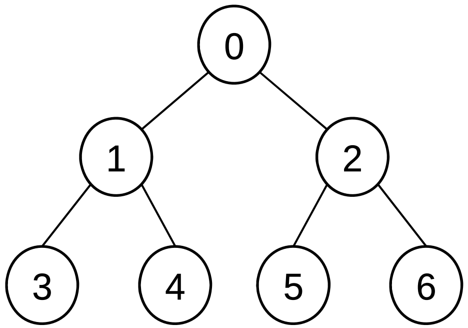
- 무방향성 그래프(undirected graph)
  - Vertex의 쌍을 나타내는 edge가 방향성이 없음
  - (u, v) , (v, u): 동일한 edge를 표현

- 방향성 그래프(Directed graph)
  - 각 edge에 방향성이 존재하는 그래프
  - <u, v>: u → v인 edge를 표현
    - u = tail, v = head

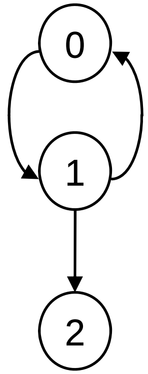
그래프의 예



G_1



G_2



G_3

$V(G_1) = \{0, 1, 2, 3\}$

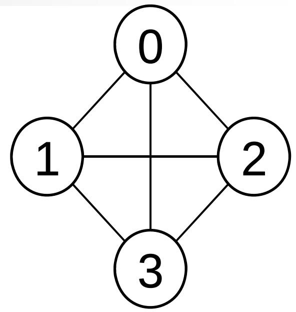
$E(G_1) = \{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)\}$

$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$

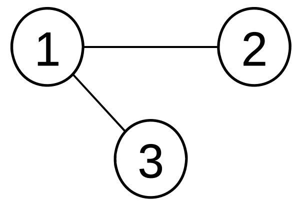
$E(G_2) = \{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)\}$

$V(G_3) = \{0, 1, 2\}$ $E(G_3) = \{<0,1>, <1,0>, <1,2>\}$

그래프에서 사용되는 용어들(1)



- 완전 그래프(Complete graph)
  - Edge의 수가 최대인 그래프
  - n개의 vertex → 최대 edge 수 = n(n–1)/2
- 부분 그래프(Subgraph)
  - $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$일 경우, G'는 G의 부분 그래프



- Vertex u에서 v까지의 경로(path)
  - $(u, v_{i1}), (v_{i1}, v_{i2}), \ldots, (v_{in}, v)$가 그래프의 edge라고 가정
  - u에서 v까지의 경로 = $u, v_{i1}, v_{i2}, \ldots, v_{in}, v$
- 경로의 길이 = 경로 상에 있는 edge의 수
- 단순 경로(simple path)
  - 처음과 마지막을 제외한 vertex가 다른 경로
  - 방향성 그래프: Simple directed path
- 사이클(cycle): 처음과 마지막이 동일한 단순 경로

그래프에서 사용되는 용어들(2)

- 연결(connected)
  - Vertex u와 v사이에 경로가 존재할 경우, u와 v는 연결
  - 방향성 그래프: strongly connected
- 연결 요소(connected component)
  - Maximal connected subgraph
  - 방향성 그래프: strongly connected component
- 트리 = Connected acyclic graph
- Vertex v의 차수(degree)
  - v에 부속된 edge의 수
  - 방향성 그래프
    - in-degree = v가 head가 되는 edge의 수
    - out-degree = v가 tail이 되는 edge의 수
- Digraph = Directed Graph의 준말

2. 그래프 표현법

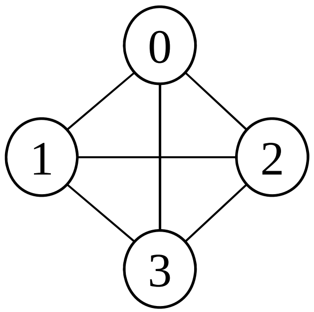
- 인접 행렬(Adjacency Matrix)
  - int A[n][n];   ← n: vertex의 수

- 인접 리스트(Adjacency List)

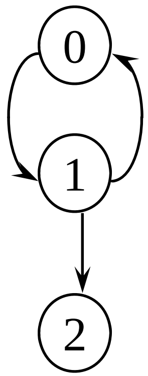
```
struct node {
    int       vertex;
    struct node   *link;
};
struct node *graph[n];
```

인접 행렬(Adjacency Matrix)

- n개의 vertex를 갖는 그래프 G의 인접 행렬의 구성
  - A[n][n]
    - (u, v) $\in$ E(G) 이면, A[u][v] = 1
    - Otherwise, A[u][v] = 0
  - 무방향성 그래프: A[][]는 대칭 행렬
  - 방향성 그래프: A[][]는 비대칭 행렬



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

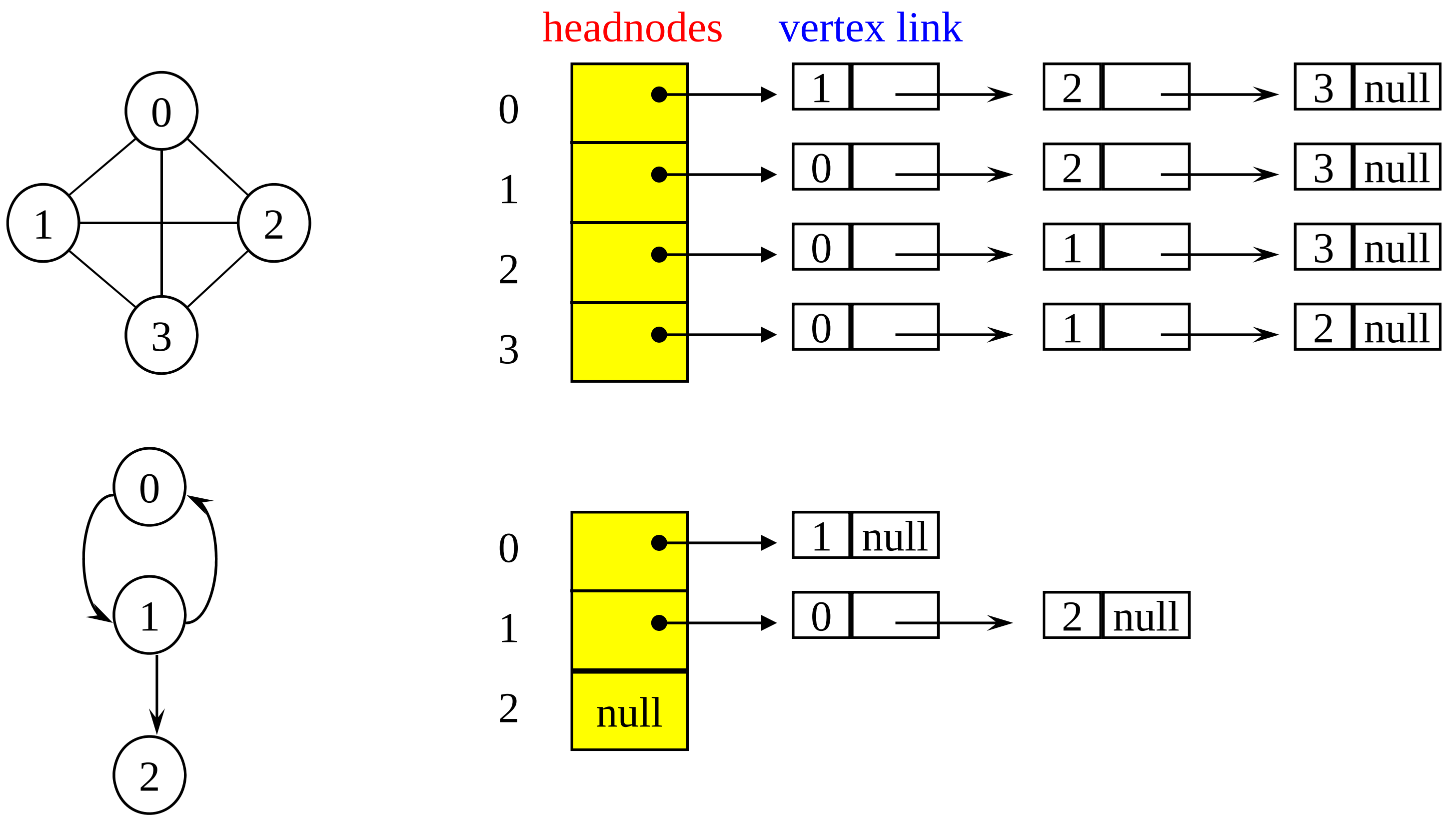


$$\begin{array}{c c} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \left[\begin{array}{ccc} 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} \\ 0 & \mathbf{0} & 0 \end{array}\right] \end{array}$$

대칭 행렬:
A[n(n-1)/2]로 구현 가능

인접 리스트(Adjacency List)

- 인접 행렬의 n 행들을 n개의 연결 리스트로 표현
  - 즉, 그래프 G의 각 vertex에 대해 한 개의 연결 리스트가 존재



3. 그래프에 적용가능한 연산

1. 깊이 우선 탐색(Depth First Search)
2. 너비 우선 탐색(Breadth First Search)
3. 연결 요소(Connected Components)
4. 신장 트리(Spanning Trees)

3.1 Depth First Search

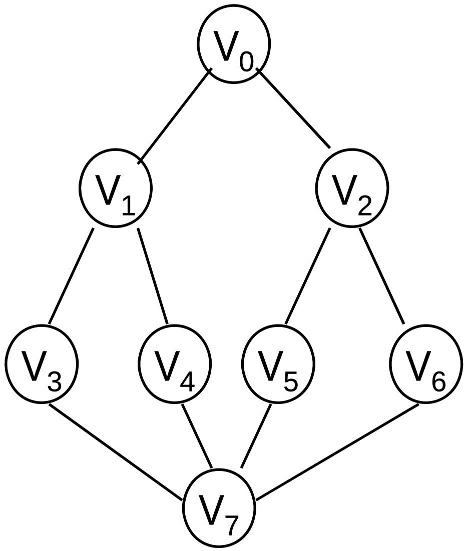
- 알고리즘
  - 출발 vertex, v의 인접 리스트부터 방문
  - v에 인접하면서 아직 방문하지 않은 vertex, w를 선택
  - w를 시작점으로 하여 다시 깊이 우선 탐색 시작
  - recursion을 이용하여 구현

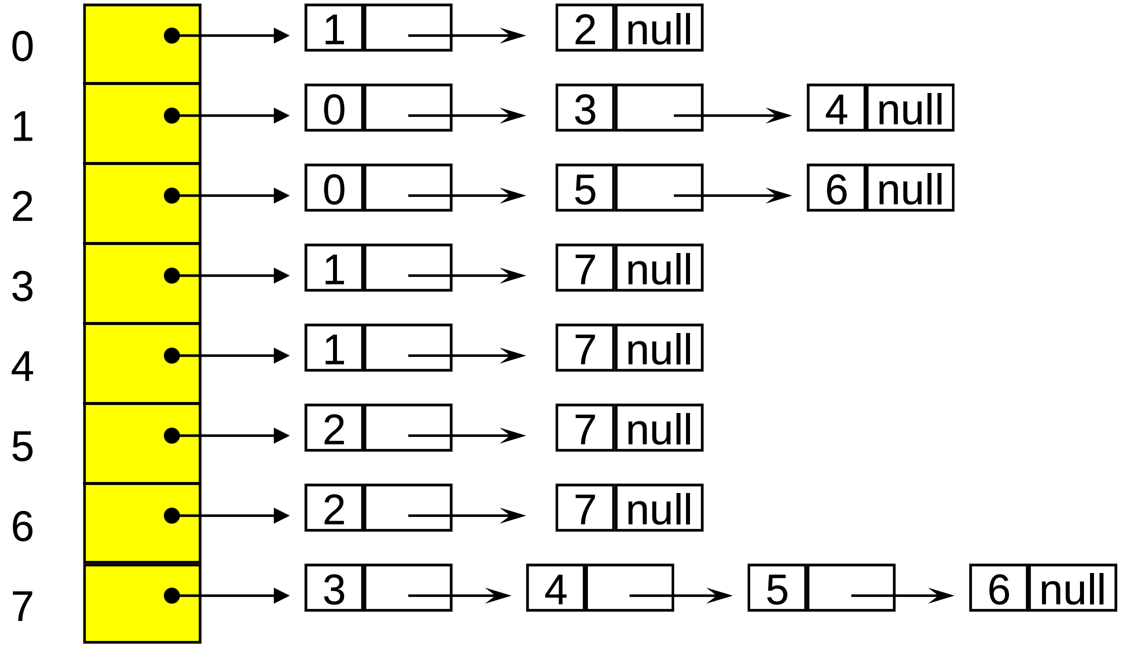
```
#define FALSE     0
#define TRUE      1
short int visited[MAX_VERTICES];

void dfs(int v)
{
        struct node *w;
        visited[v] = TRUE;
        printf("%5d", v);
        for (w = graph[v]; w; w = w→link)
                if (!visited[w→vertex]) dfs(w→vertex);
}
```

Depth First Search의 동작 과정

struct node *graph[];

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





depth first search order : V_0, V_1, V_3, V_7, V_4, V_5, V_2, V_6

3.2 Breadth First Search

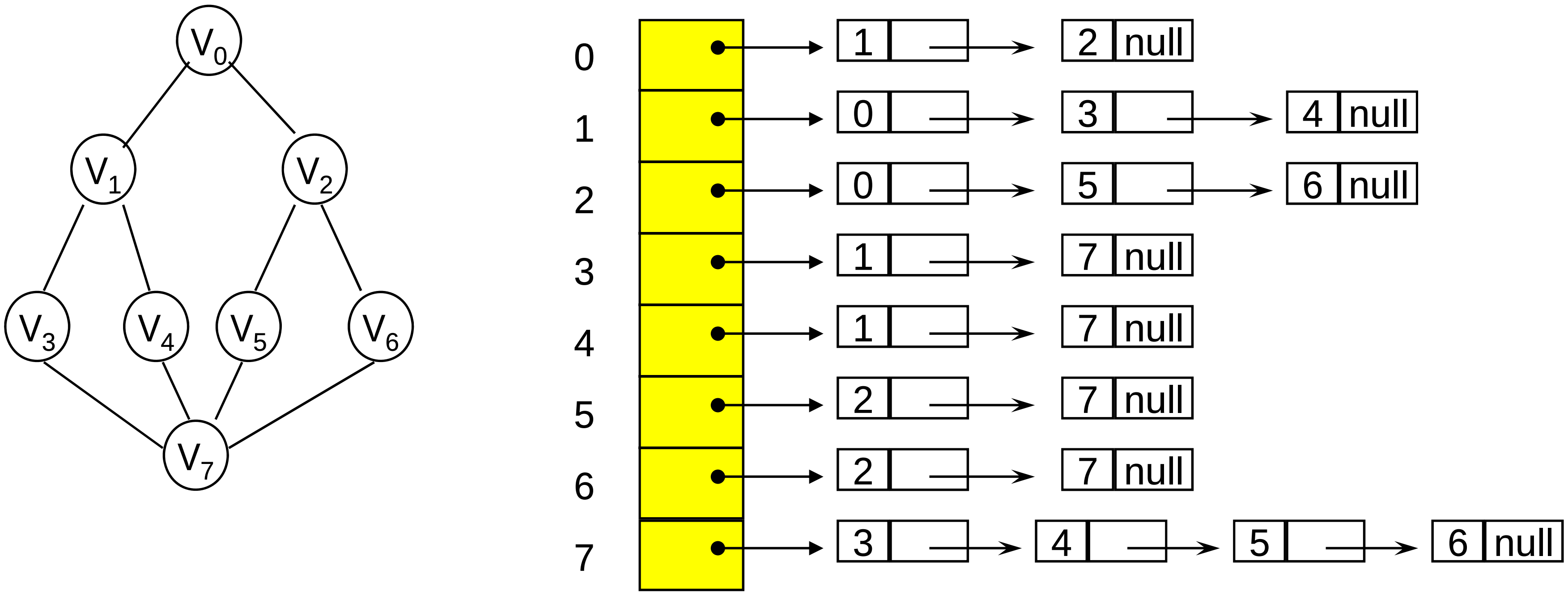
- 알고리즘
  - 출발 vertex, v의 인접 리스트부터 방문
  - v에 인접한 모든 vertex들을 먼저 방문
  - 그 다음, v에 인접한 첫번째 vertex에 인접한 vertex중에서 아직 방문하지 않은 vertex들을 다시 차례대로 방문 (Queue를 이용)

```
struct queue {
        int vertex;
        struct queue *link;
};
void addq(....);
int deleteq(....);
```

Breadth First Search

```
void bfs(int v)
{
// Vertex v부터 탐색 수행. visited[] 배열은 FALSE로 초기화. 큐를 사용
    node_pointer   w;
    struct queue   *front = NULL, *rear = NULL;

     printf("%5d", v);
     visited[v] = TRUE;
     addq(v);
     while (front) {
           v = deleteq();
           for (w = graph[v]; w; w = w→link)
                     if (!visited[w→vertex]) {
                                printf("%5d", w→vertex);
                                addq(&front, &rear, w→vertex);
                                visited[w→vertex] = TRUE;
}   }                     }
```

Breadth First Search의 동작 과정



breadth first search order : V_0, V_1, V_2, V_3, V_4, V_5, V_6, V_7

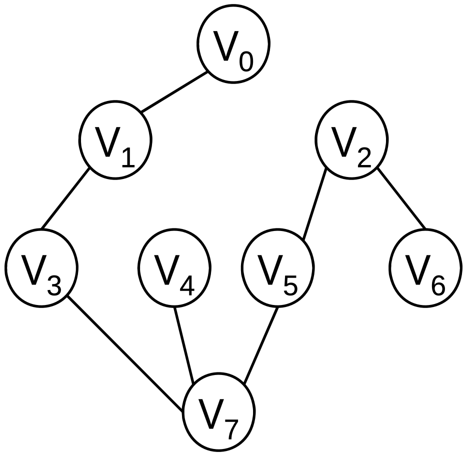
3.3 Connected Components

- 무방향성 그래프가 연결되어 있는지 검사?
  - dfs(0)나 bfs(0)를 호출한 후, 방문되지 않은 vertex가 존재하는지를 검사
- 그래프의 connected component들을 출력?

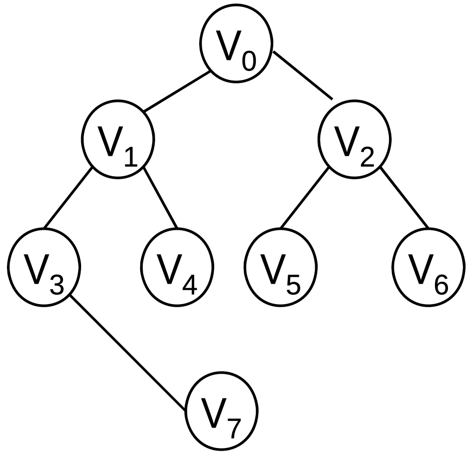
```
void connected (void)
{
   // 그래프의 연결 요소들을 한 줄씩 출력
   int i
   for (i = 0; i < n; i++)
         if (!visited[i]) {
               dfs(i);  printf("₩n");
         }
}
```

3.4 Spanning Trees

- 정의
  - 그래프 G에 포함된 edge들로 구성되며, G의 모든 vertex들을 포함하는 트리
- DFS나 BFS를 이용하여 spanning tree 구성
  - Depth first spanning tree
  - Breadth first spanning tree



depth first spanning tree



breadth first spanning tree

3.5 최소 비용 신장트리

- 정의
  - 가중치가 부여된 무방향 그래프에서 신장 트리의 비용 = 신장 트리를 구성하는 edge들의 비용의 합
  - Minimum cost spanning tree : 가장 비용이 적은 spanning tree

- 응용 분야
  - 도로 건설: 도시를 모두 연결하면서 도로의 길이가 최소가 되도록 하는 문제
  - 통신
  - 배관

Prim의 Algorithm

- 알고리즘의 개요
  - 단 하나의 vertex만 갖는 트리 T에서 시작
  - 각 단계에서 트리를 구성하도록 한번에 하나의 edge를 선택
  - T에 포함된 vertex와 포함되지 않은 vertex를 연결하는 edge들 중에서 비용이 최소인 edge를 T에 추가.
    - 추가된 edge의 vertex를 T에 포함시킴.

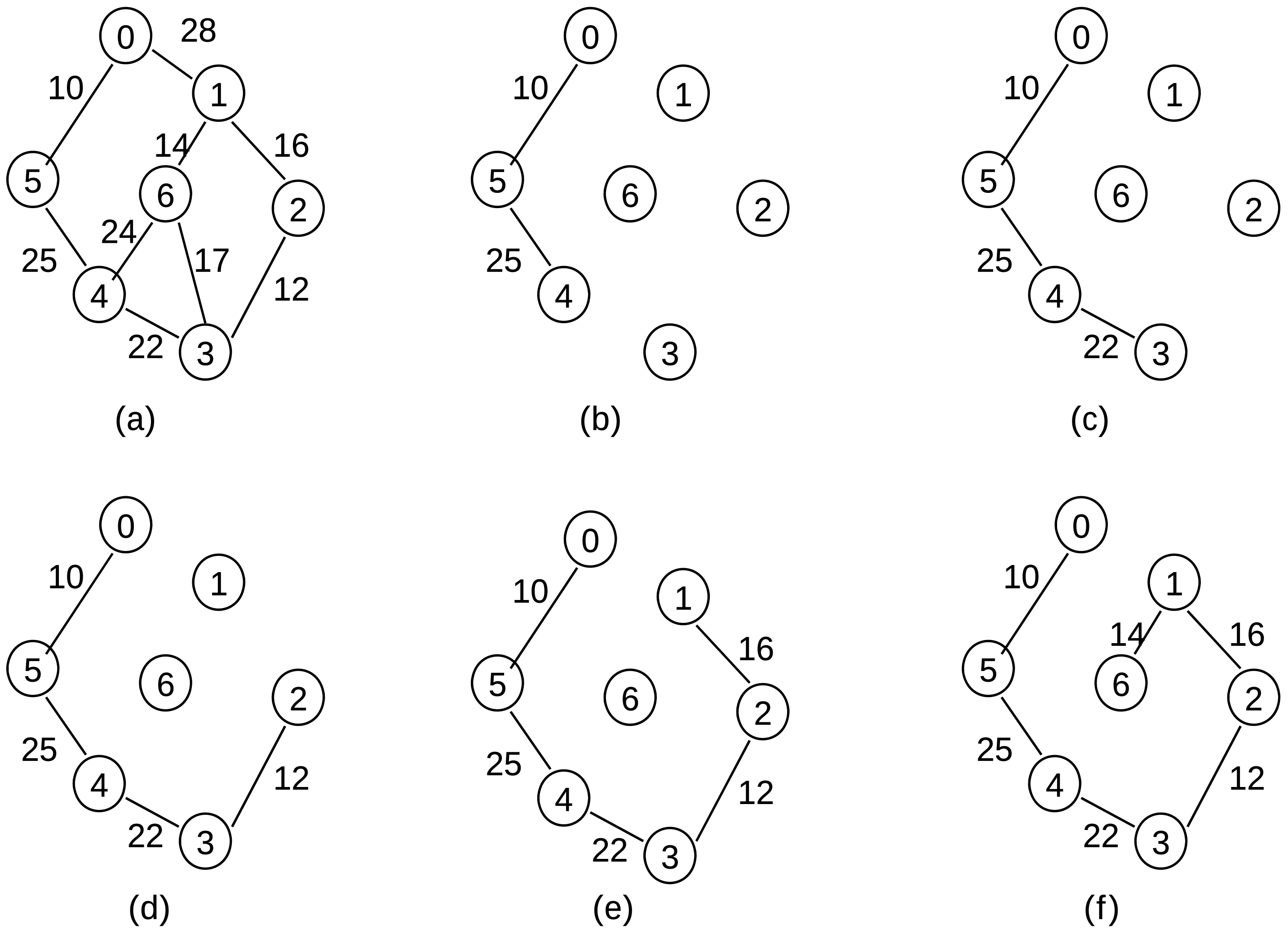
- 시간 복잡도 = **$O(n^2)$**

Prim Algorithm의 모습

```
T = {};
TV = {0};      // vertex 0에서 시작. 초기에는 edge 없음.
while (T contains fewer than n-1 edges) {
   let (u,v) be a least cost edge such that
         u ∈ TV and v ∉ TV;
   if (there is no such edge)   break;
   add v to TV;
   add (u,v) to T;
}
if ( T contains fewer than n-1 edges )
   printf("No spanning tree₩n");
```

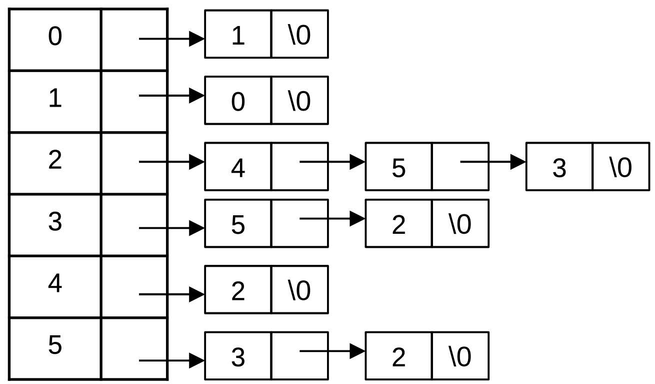
시간 복잡도 = $O(n^2)$

Prim Algorithm의 동작 예



그래프 – 실습 1

- 0부터 n-1까지의 vertex로 구성된 그래프에서 n과 무방향성 에지들을 차례대로 입력받아 인접 리스트를 구성하라.

<table>
<tr><th>입력</th><th>인접 리스트</th></tr>
<tr><td>6
0 1
2 3
2 5
5 3
2 4
-1 -1</td><td>
</td></tr>
</table>

그래프 – 실습 2

- 실습 **1**의 그래프에 대해 다음 연산을 수행하라.
  - Degree가 가장 큰 노드의 degree와 그 노드에 연결된 노드들의 리스트를 출력하라.
    - 앞의 그래프의 경우:
      - 노드 = 2
      - Degree = 3
      - 리스트 = 4 → 5 → 6
  - dfs() 함수를 구현한 후, dfs(0)을 실행하여 그 결과를 확인하라.
  - dfs() 함수를 이용하여 연결 요소를 출력하라.