**10. Graph**

**10-1. 그래프의 이해**

**# 정의**

: 연결되어 있는 원소 간의 관계를 표현하는 자료구조

ex) 인맥 지도, 수도 배관 배수 시스템, 물질의 분자 구조

**그래프 G는 집합(set) 두 개로 구성**

- 정점(vertex 또는 Node)

- 간선(Edge)

**스케치, 원, 그림, 화이트이(가) 표시된 사진

자동 생성된 설명**

**G = (V, E)**

V : 그래프에 있는 정점들의 집합을 의미함.(대상 : 대상물, 개념 등)

E : 정점을 연결하는 간선들의 집합을 의미함.(대상들 간의 관계)

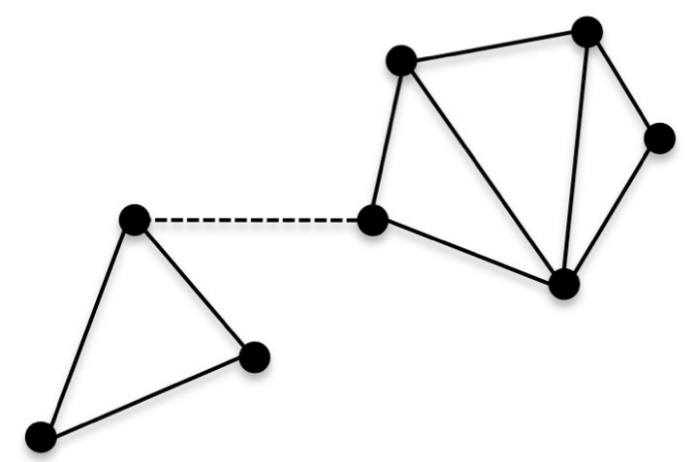
**# 용어**

**1. 연결(Connected)**

: 그래프에서 두 정점 Vi와 Vj까지의 경로가 있으면, 정점 Vi와 Vj가 연결되었다고 한다.

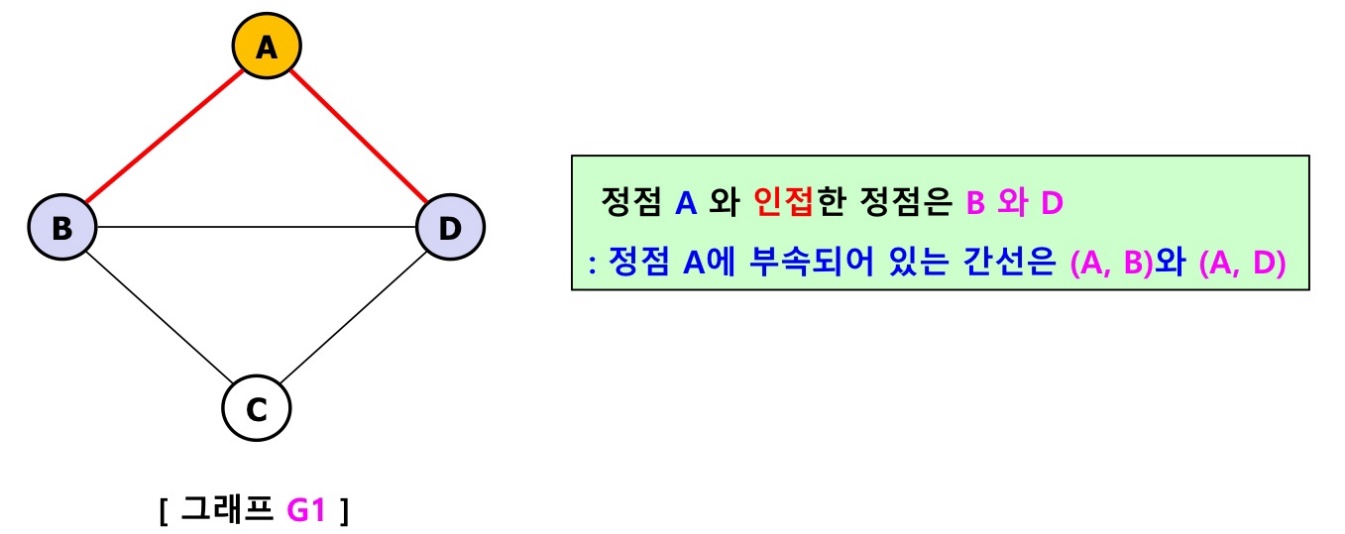
- 연결 그래프(Connected Graph) : 떨어져 있는 정점이 없는 그래프

- 단절 그래프(Disconnected Graph) : 연결되지 않은 정점이 있는 그래프



**2. 인접과 부속(Adjacent and Incident)**

: 그래프에서 두 정점 Vi와 Vj가 연결되어 간선(Vi, Vj)가 있을 때 두 정점 Vi와 Vj를 인접(adjacent)되어 있다고 하고, 간선(Vi, Vj)는 정점 Vi와 Vj에 부속(incident)되어 있다고 한다.



**3. 차수(Degree)**

: 정점에 부속되어 있는 간선의 수

- 유향 그래프에서는 정점에 부속된 간선의 방향에 따라서

🡪 진입차수(in-degree), 진출 차수(out-degree)

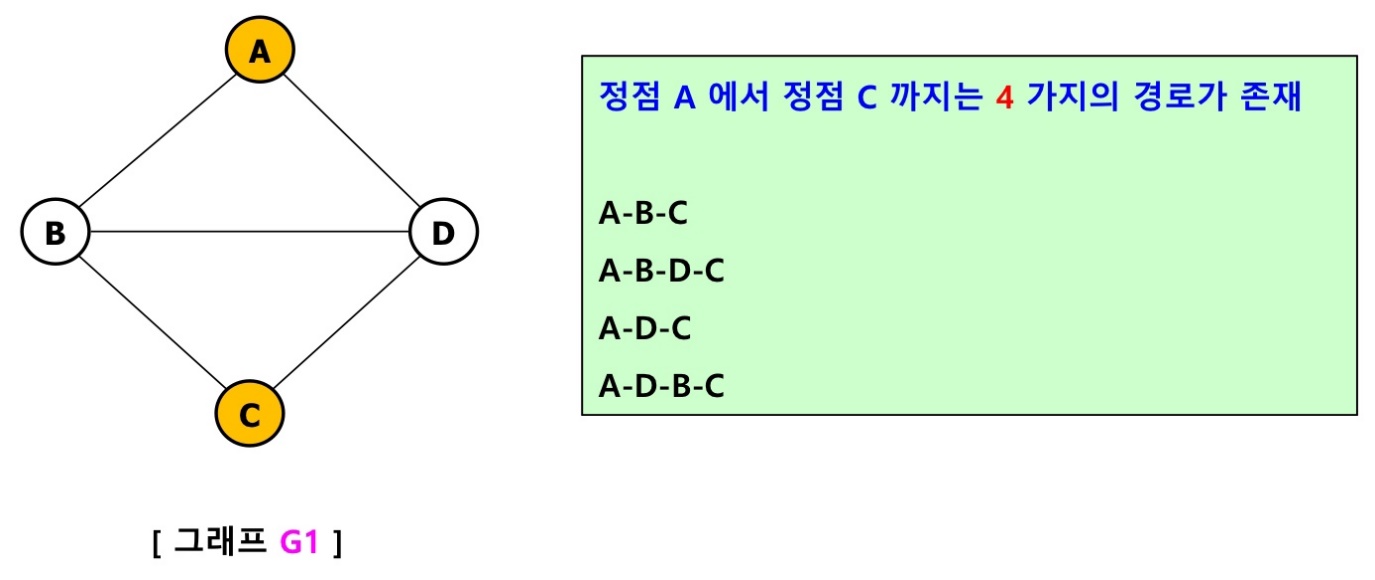
- 유향 그래프에서의 정점의 차수는 진입 차수와 진출 차수를 합한 값이다.

**텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

자동 생성된 설명**

**4. 경로(Path)**

: 그래프에서 간선을 따라 갈 수 있는 길을 순서대로 나열 한 것



**- 경로 길이(Path Length)**

: 경로를 구성하는 간선의 수

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

**- 순환(Cycle)**

: 단순 경로 중에서 경로의 시작 정점과 마지막 정점이 같은 경로

\* 단순 경로 : 모두 다른 정점으로 구성된 경로

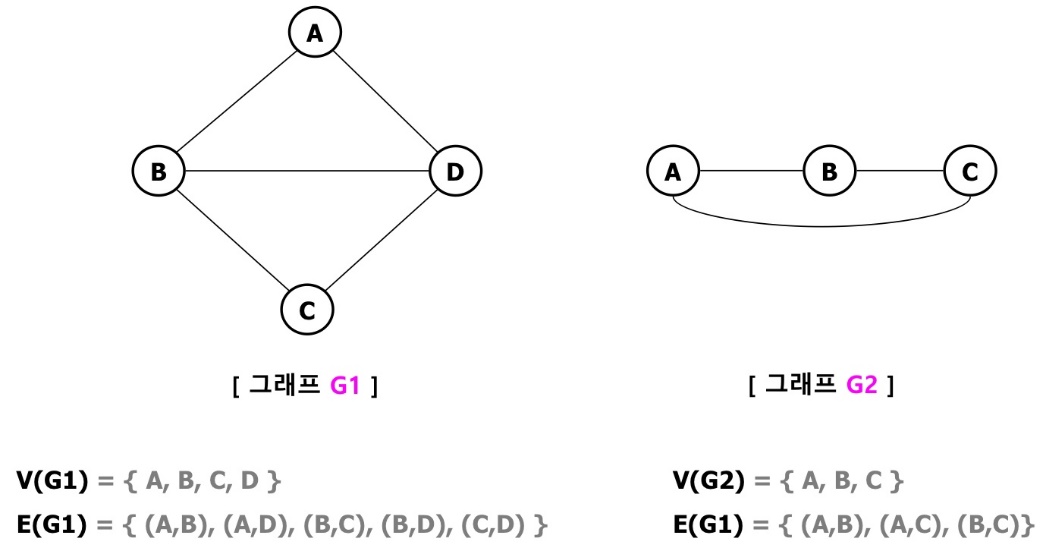
**도표, 원, 라인이(가) 표시된 사진

자동 생성된 설명**

**# 종류**

**1. 무향 그래프(Undirected Graph)**

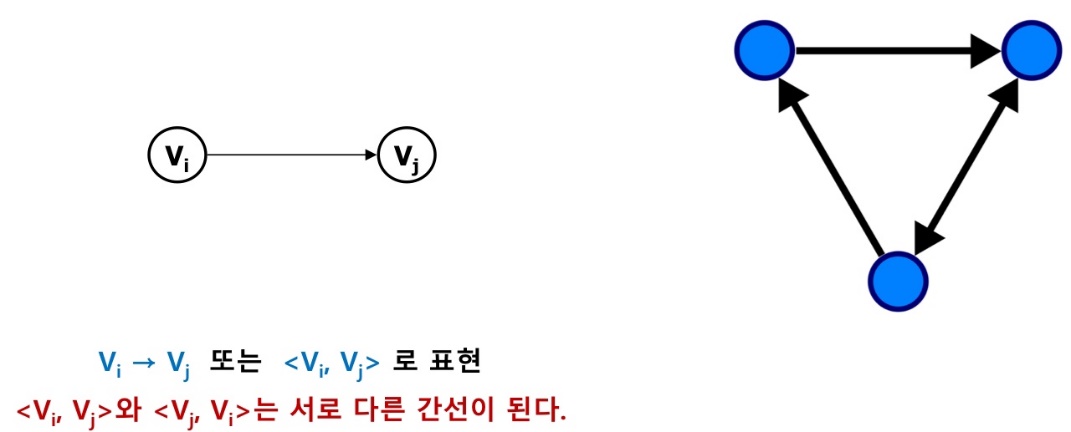
: 두 정점을 연결하는 간선에 방향성이 없는 그래프

****

**2. 유향 그래프(Directed Graph)**

: 두 정점을 연결하는 간선에 방향성이 있는 그래프

ex) 정점Vi에서 정점Vj를 연결하는 간선



텍스트, 도표, 스케치, 원이(가) 표시된 사진

자동 생성된 설명

**3. 가중치 그래프(Weight Graph)**

: 두 정점을 연결하는 간선에 가중치를 할당한 그래프

- 가중치는 두 정점 사이의 거리 또는 지나는 시간이 될 수도 있다.

- 또한 음수인 경우도 존재한다.

도표, 라인, 원이(가) 표시된 사진

자동 생성된 설명

**4. 완전 그래프(Complete Graph)**

: 모든 정점들 사이에 1:1로 직접 연결된 간선을 지닌 그래프

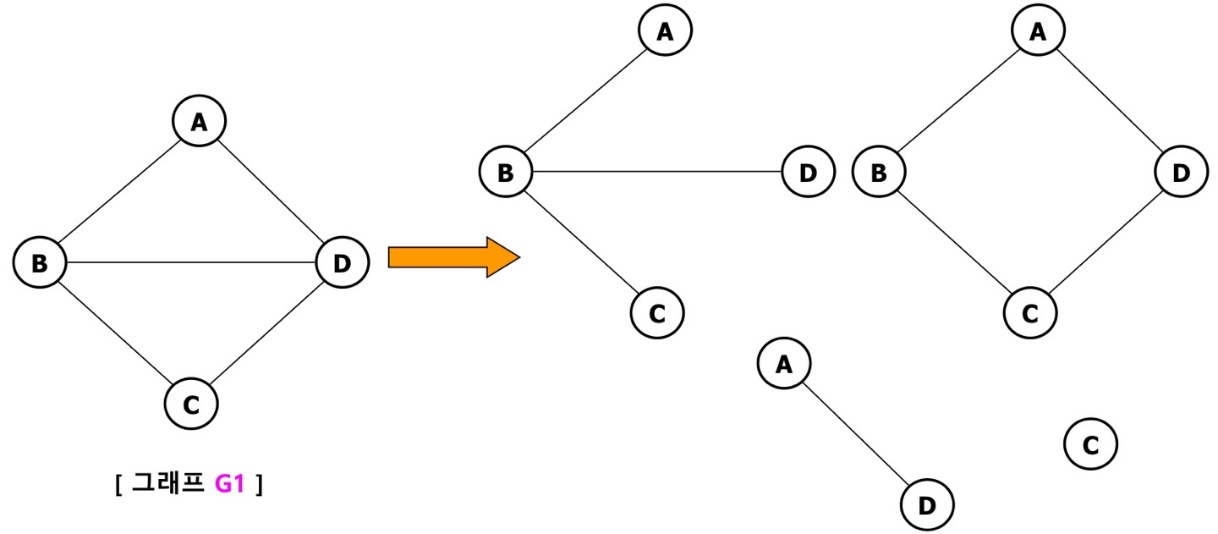
도표, 라인, 텍스트, 종이접기이(가) 표시된 사진

자동 생성된 설명

**5. 부분 그래프(Sub Graph)**

: 원래의 그래프에서 일부의 정점이나 간선을 제외하여 만든 그래프

- 부분 그래프는 원래의 그래프에 없는 정점이나 간선을 포함하지 않는다.

****

\* 트리(Tree) : 순환이 없는 연결된 그래프

**# 표현**

**1. 인접 행렬(Adjacent Matrix)**

: 순차 자료구조를 이용하는 2차원 배열의 방법

🡪 그래프의 두 정점을 연결한 간선의 유무를 행렬로 저장

\* N개의 정점을 가진 그래프: N x N 정방 행렬

\* 행렬의 행과 열: 그래프의 정점

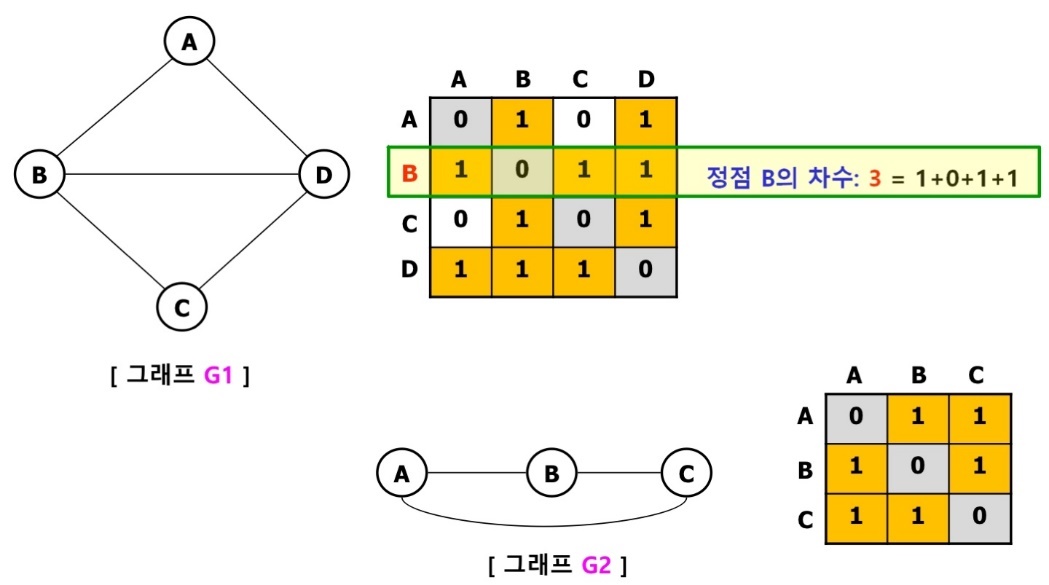
\* 행렬 값: 두 정점이 인접되어 있으면 1, 인접되어 있지 않으면 0

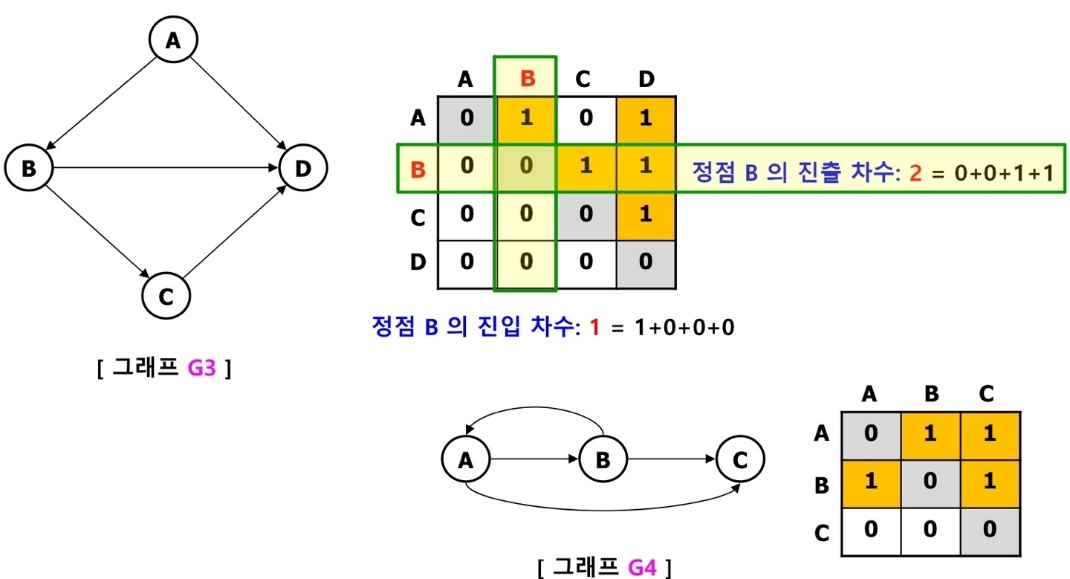
**- 무향 그래프**

: 행 i의 합 = 열 i의 합 = 정점 i의 차수

**- 유향 그래프**

: 행 i의 합 = 정점 i의 진출 차수/ 열 i의 합 = 정점 i의 진입 차수





**2. 인접 리스트(Adjacent List)**

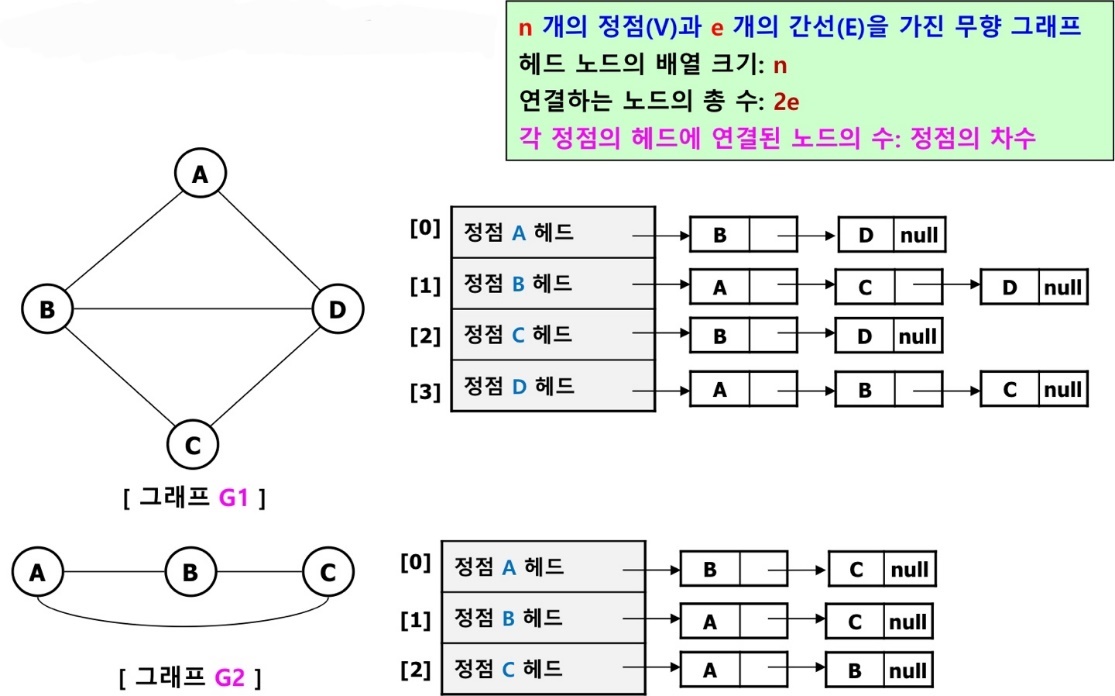
: 각 정점에 대한 인접 정점들을 연결하여 만든 단순 연결 리스트

\* 정점의 헤드 노드: 정점에 대한 리스트의 시작을 표현

\* 인접 리스트의 각 노드: 정점을 저장하는 필드와 다음 인접 정점을 연결하는 링크 필드로 구성

\* 각 정점의 차수만큼 노드를 연결: 리스트 내의 노드들은 인접 정점에 대해서 오름 차순으로 연결

**- 무향 그래프**

****

**- 유향 그래프**

텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

**# 순회(Traversal)**

\* 트리의 순회 : 깊이 우선순회(DFS), 너비 우선 순회(BFS)

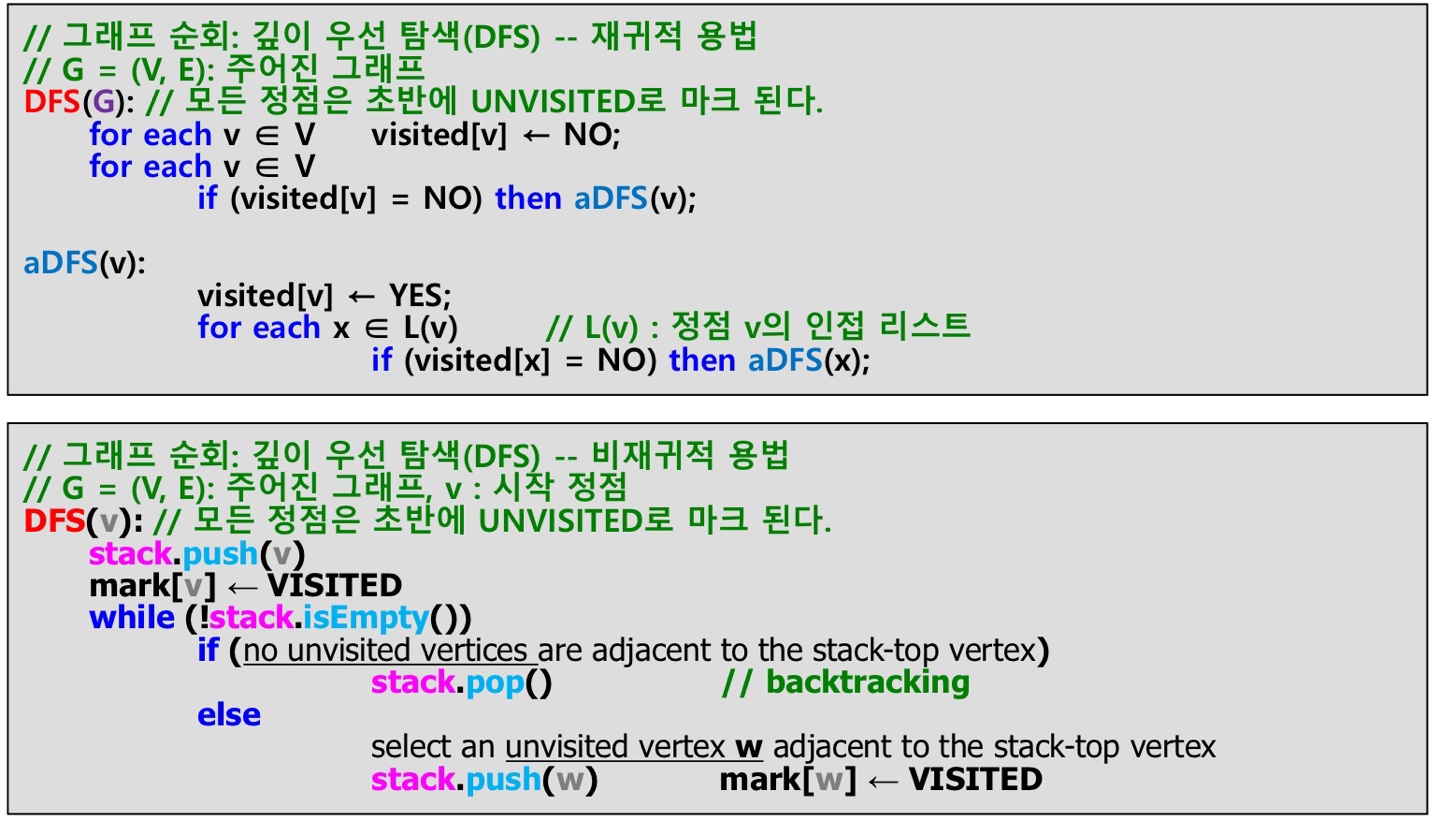
**1. 그래프 탐색(Graph Search)**

: 하나의 정점에서 시작하여 그래프에 있는 모든 정점을 한번씩 방문한다.

텍스트, 도표, 라인, 원이(가) 표시된 사진

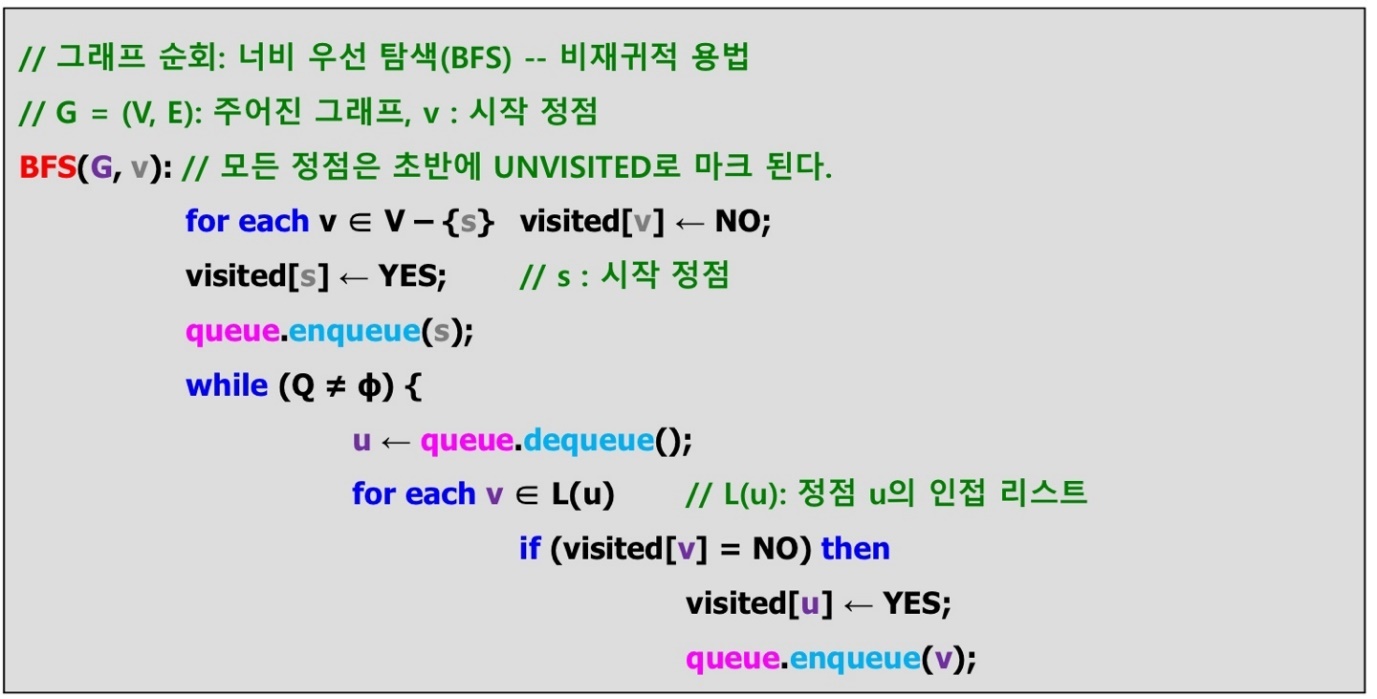
자동 생성된 설명

**2. 깊이 우선 탐색(DFS)**

**텍스트, 도표, 그림, 스케치이(가) 표시된 사진

자동 생성된 설명**

**3. 너비 우선 탐색(BFS)**



**도표, 텍스트, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명**

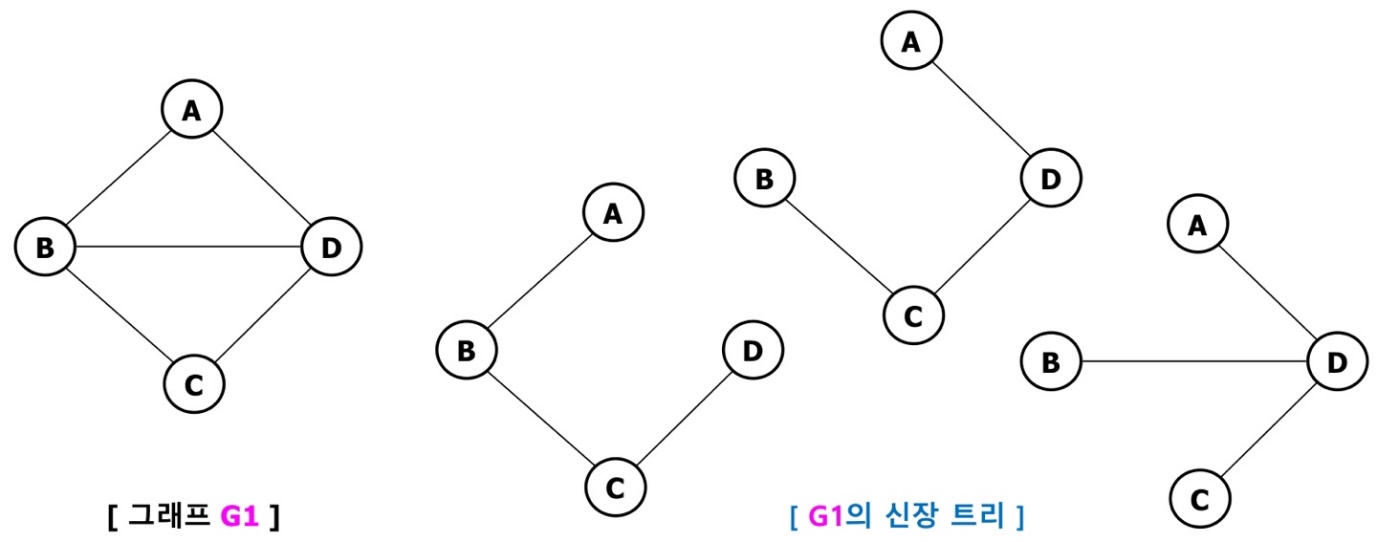
**10-2. 최소 신장 트리**

**10-2-1. 신장 트리(Spanning Tree)**

**# 정의**

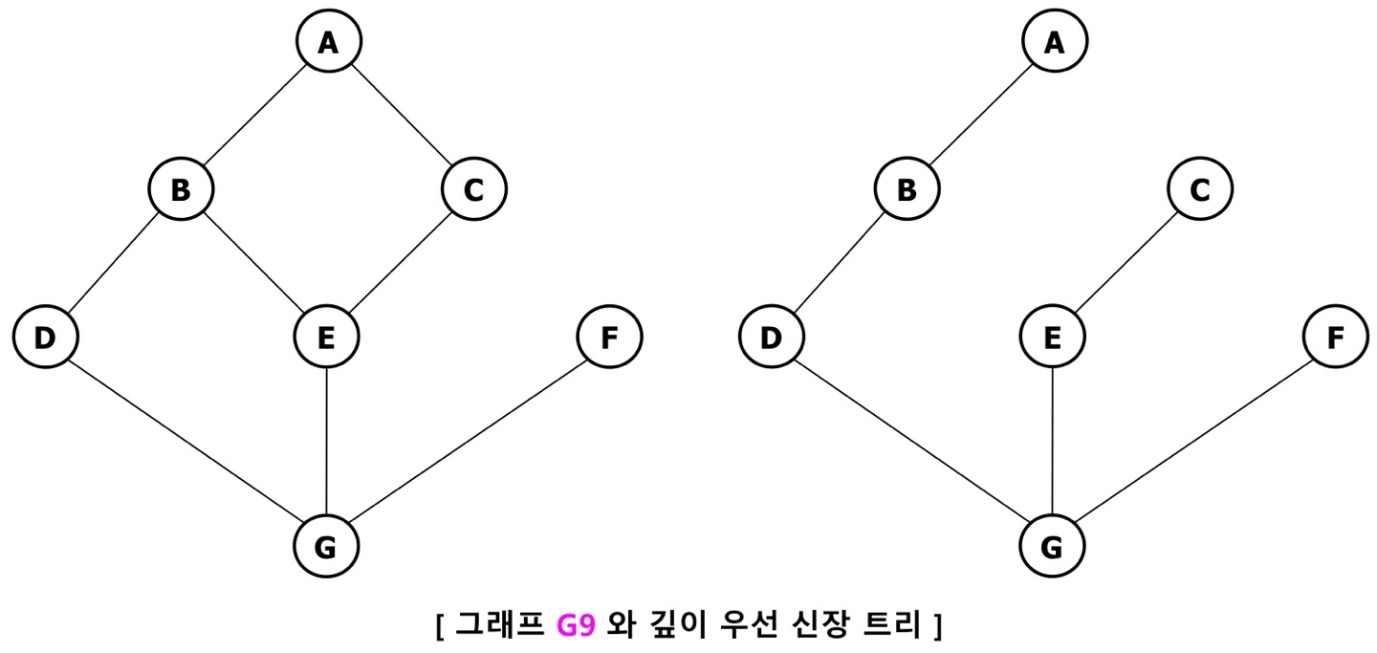
: N개의 정점으로 이루어진 무향 그래프 G에서 N개의 모든 정점과 N-1개의 간선으로 만들어진 트리

- 그래프의 관점에서 트리는 사이클이 없는 단순 연결 그래프



**# 깊이 우선 신장 트리(Depth First Spanning Tree)**

: 깊이 우선 탐색을 이용하여 생성된 신장 트리





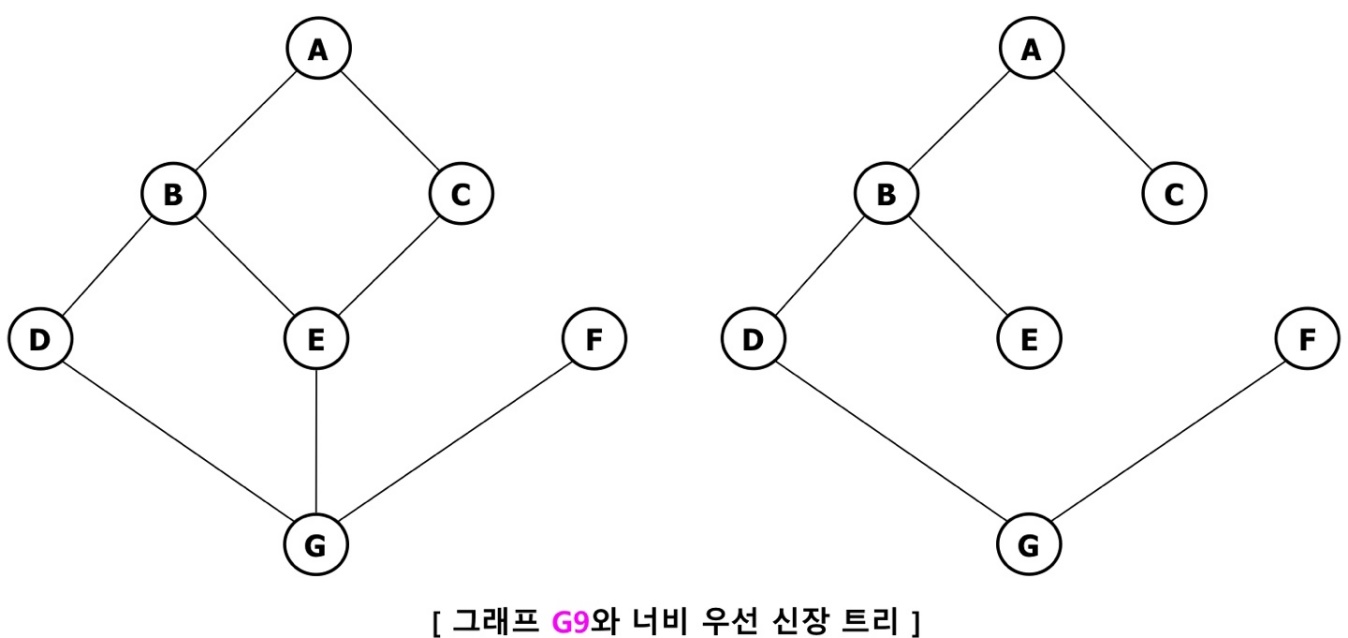
- 만들어진 T는 DFS 신장 트리라 한다.

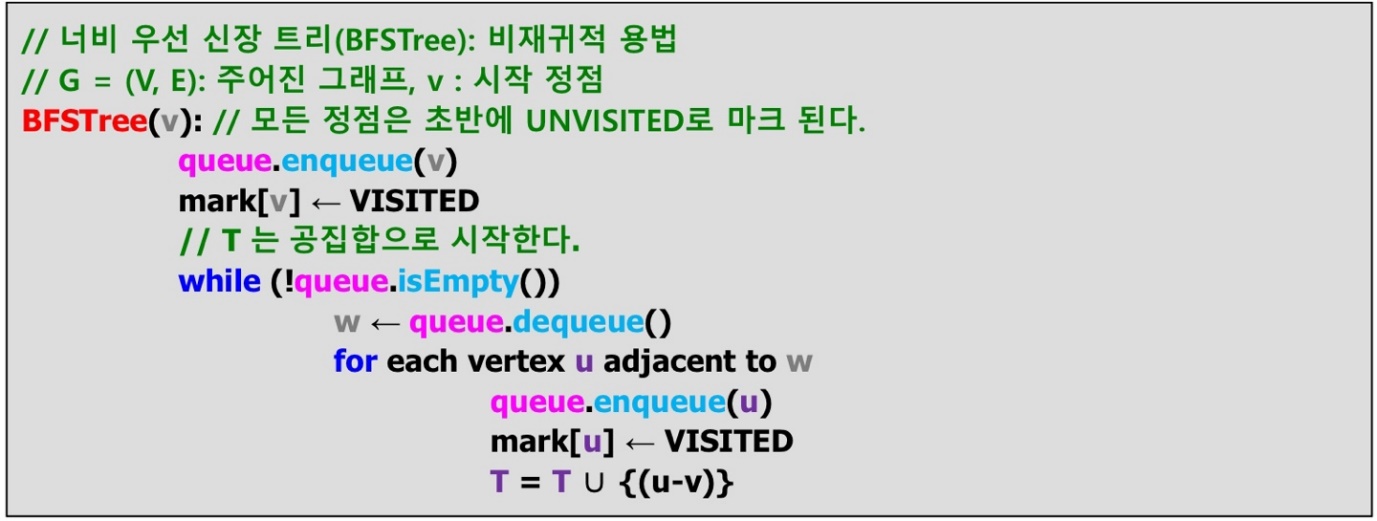
- 보통은 간선을 {u, v} 또는 (u, v)로 표현한다.

- 여기서는 좀 더 직관적인 (u-v), (u->v)로 표현하기로 한다.

**# 너비 우선 신장 트리(Breadth First Spanning Tree)**

: 너비 우선 탐색을 이용하여 생성된 신장 트리





- 만들어진 T는 BFS 신장 트리라 한다.

- 보통은 간선을 {u, v} 또는 (u, v)로 표현한다.

- 여기서는 좀 더 직관적인 (u-v), (u->v)로 표현하기로 한다.

**# 최소 신장 트리(Minimum Spanning Tree)**

: 무향 가중치 그래프에서 신장 트리를 구성하는 간선들의 가중치 합이 최소인 신장 트리

**- 신장 트리의 비용(Cost of a Spanning Tree)**

: 신장 트리를 구성하는 간선 가중치의 합

* **최소신장트리: 비용을 최소로 하는 신장 트리**

**- 가중치 그래프의 간선에 주어진 가중치**

: 비용이나 거리, 시간을 의미하는 값

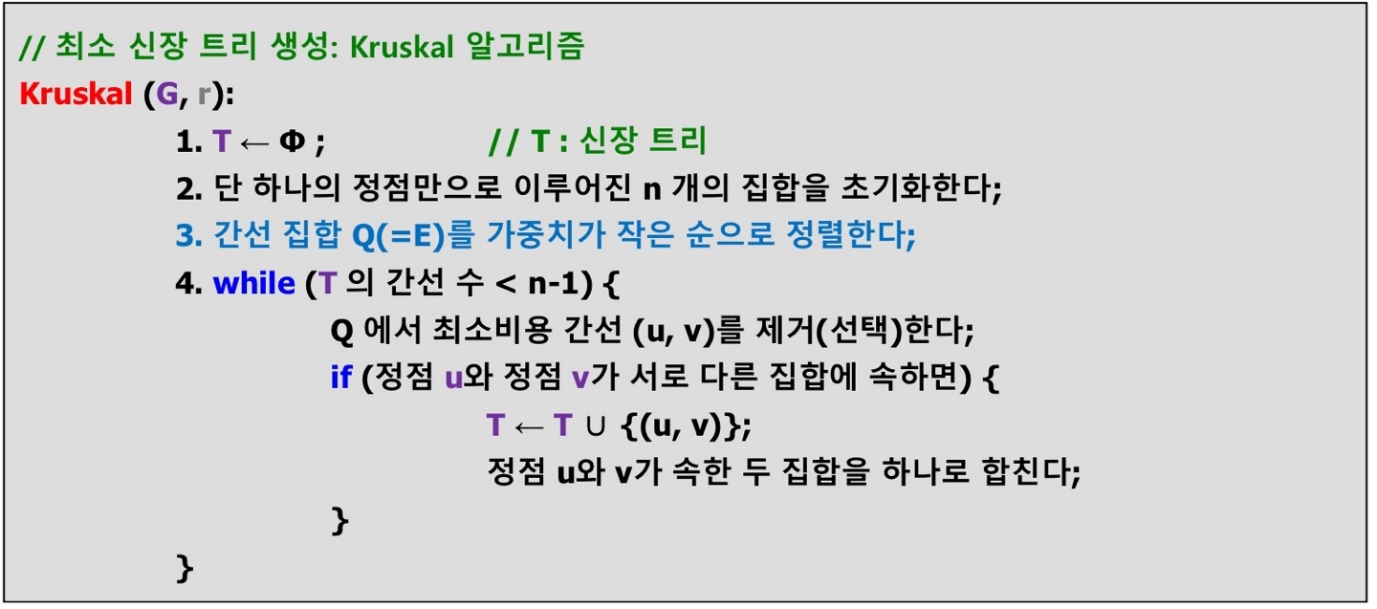
ex) 최소 신장 트리 생성 알고리즘 : Kruskal, Prim 알고리즘

**10-2-2. Kruskal 알고리즘**

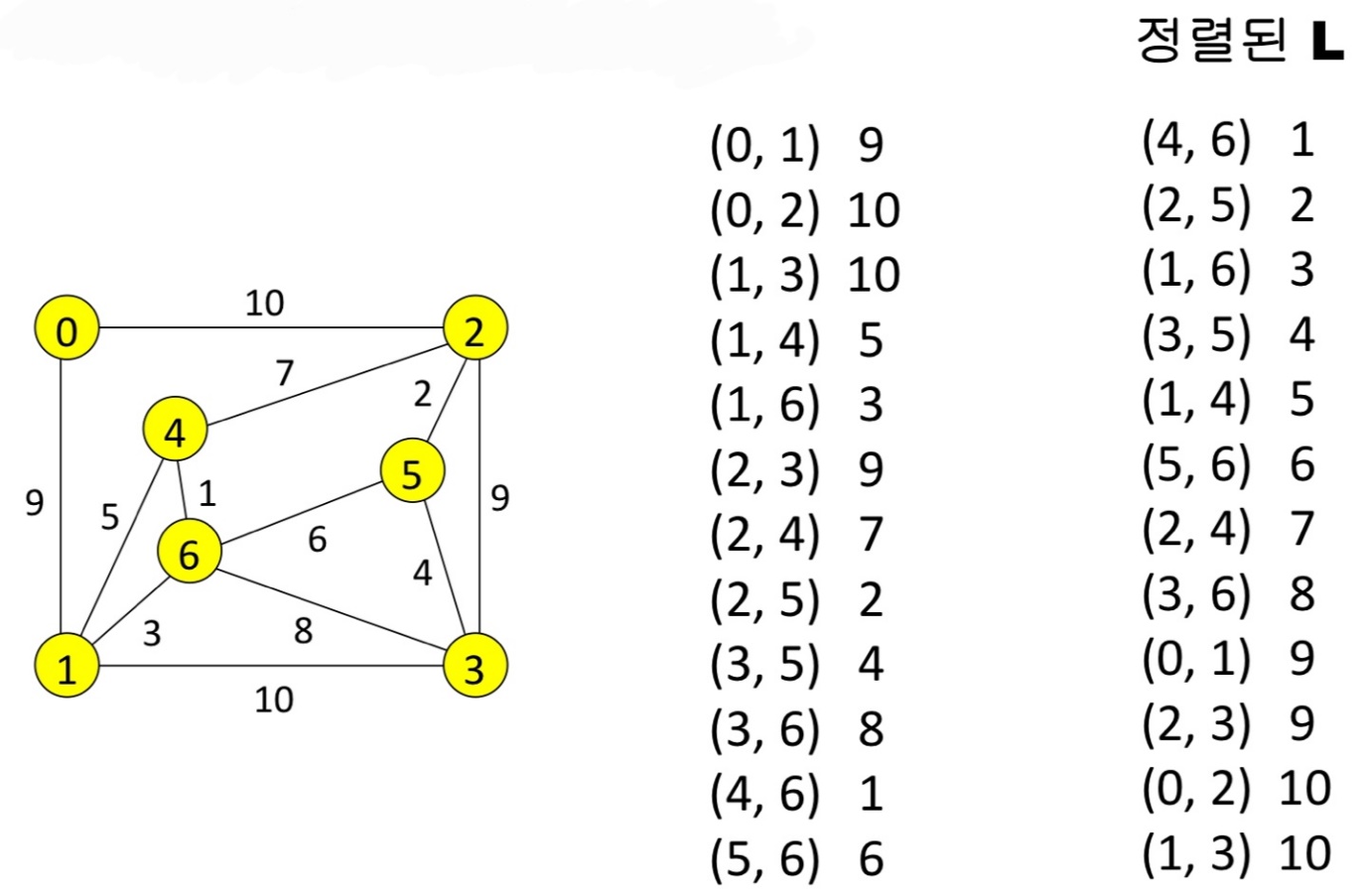
**# 정의**

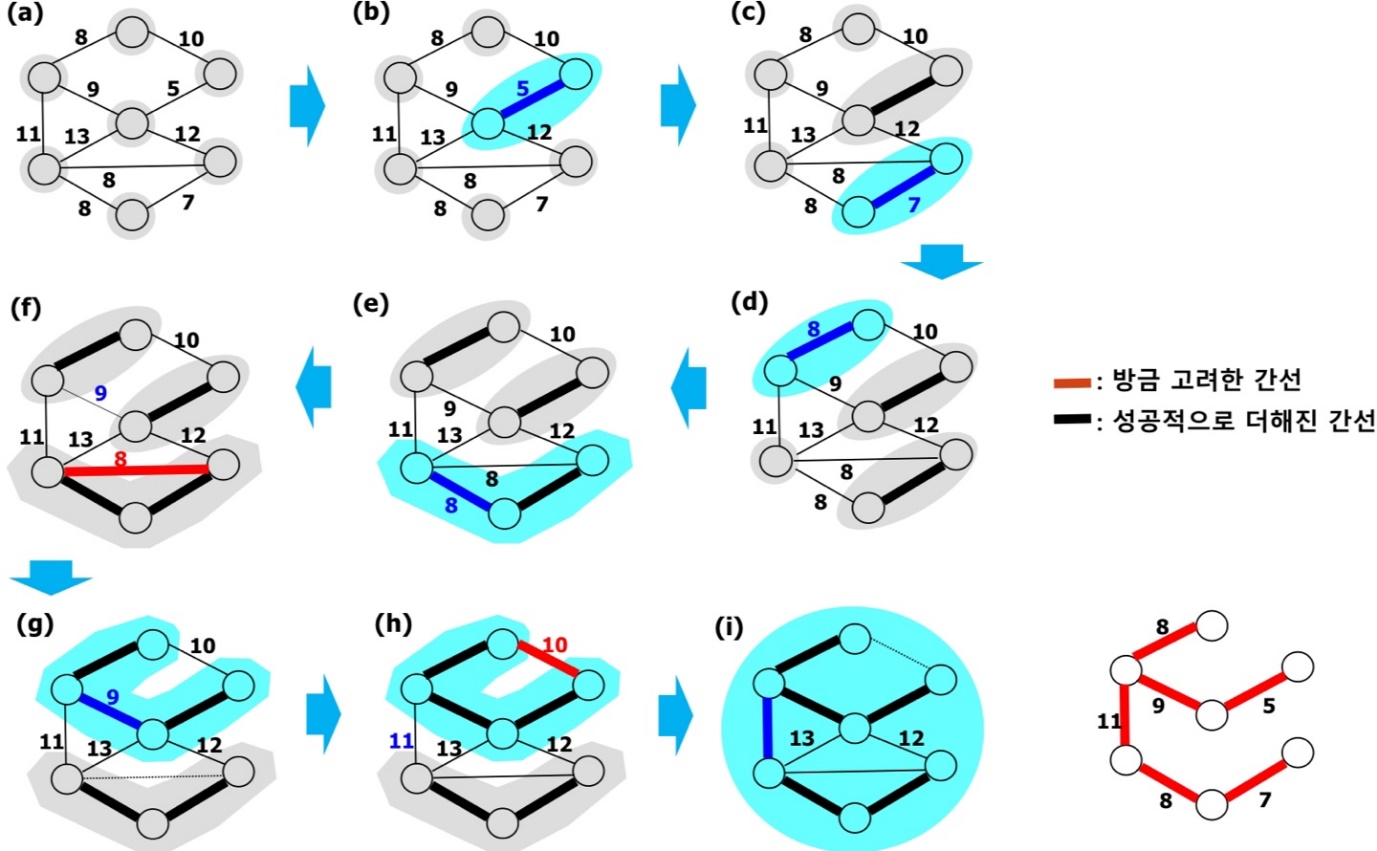
: 탐욕 알고리즘의 예

ex) 탐욕 알고리즘(Greedy Algorithm) : 우선 눈앞의 이익을 최대화하는 선택을 계속하는 알고리즘

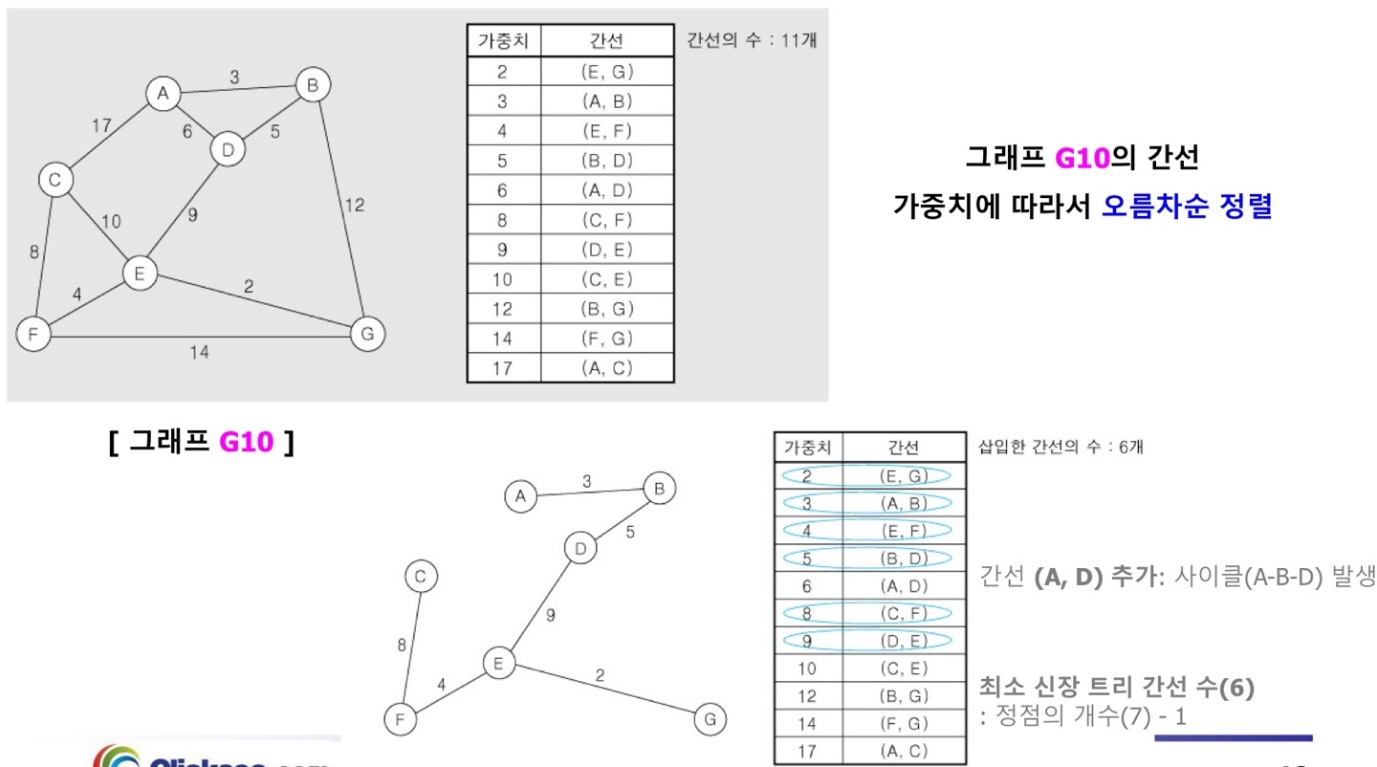


**# 알고리즘**

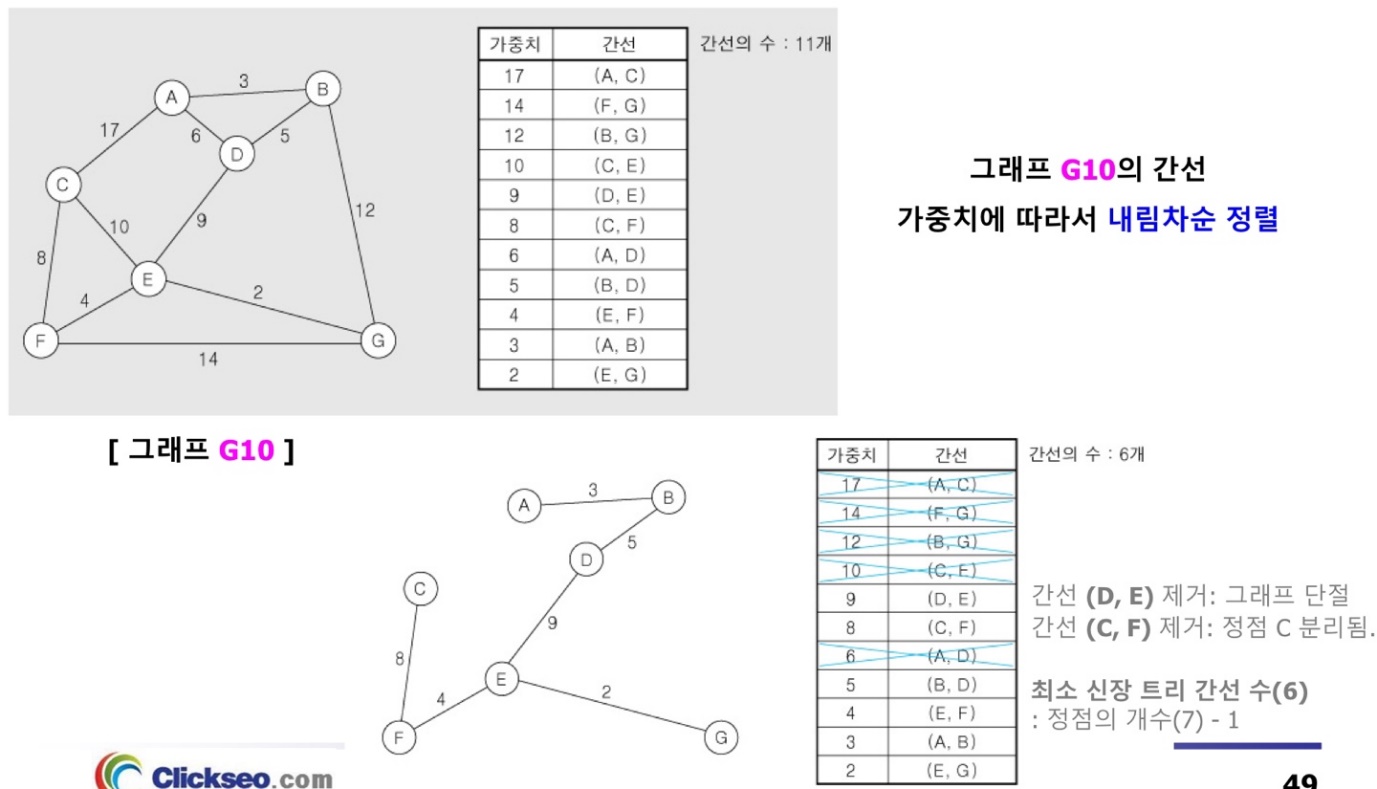
****

****

**1. 간선 가중치에 따라서 오름차순 정렬**

****

**2. 간선 가중치에 따라서 내림차순 정렬**

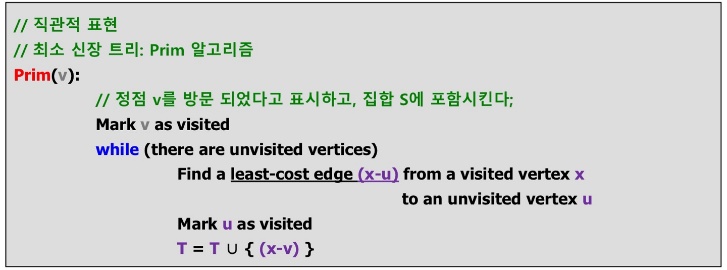
****

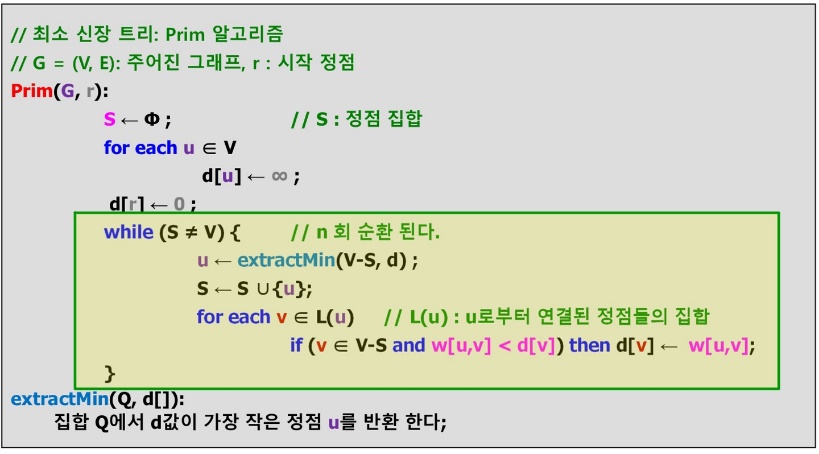
**10-2-3. Prim 알고리즘**

**# 정의**

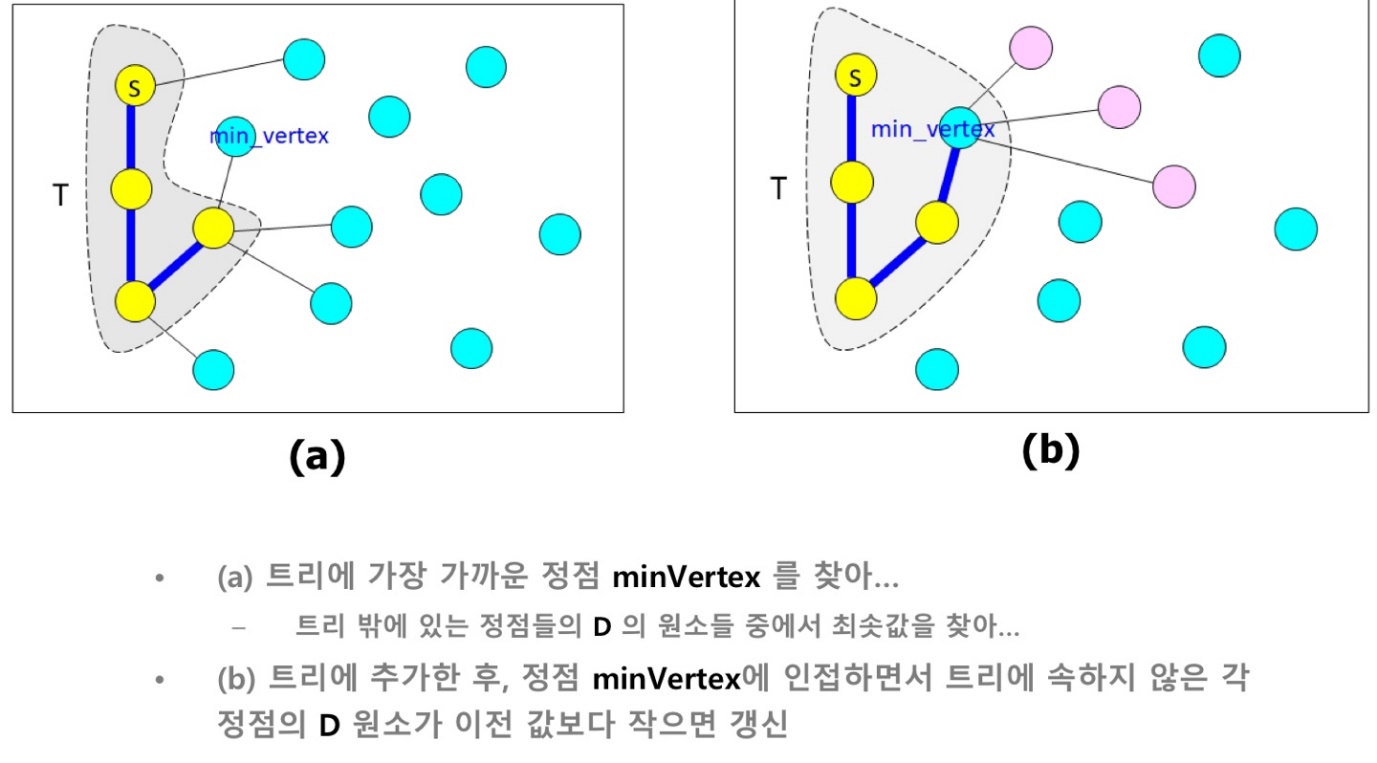
: 간선을 정렬하지 않고, 하나의 정점에서 시작하여 트리를 확장해 나가는 방법

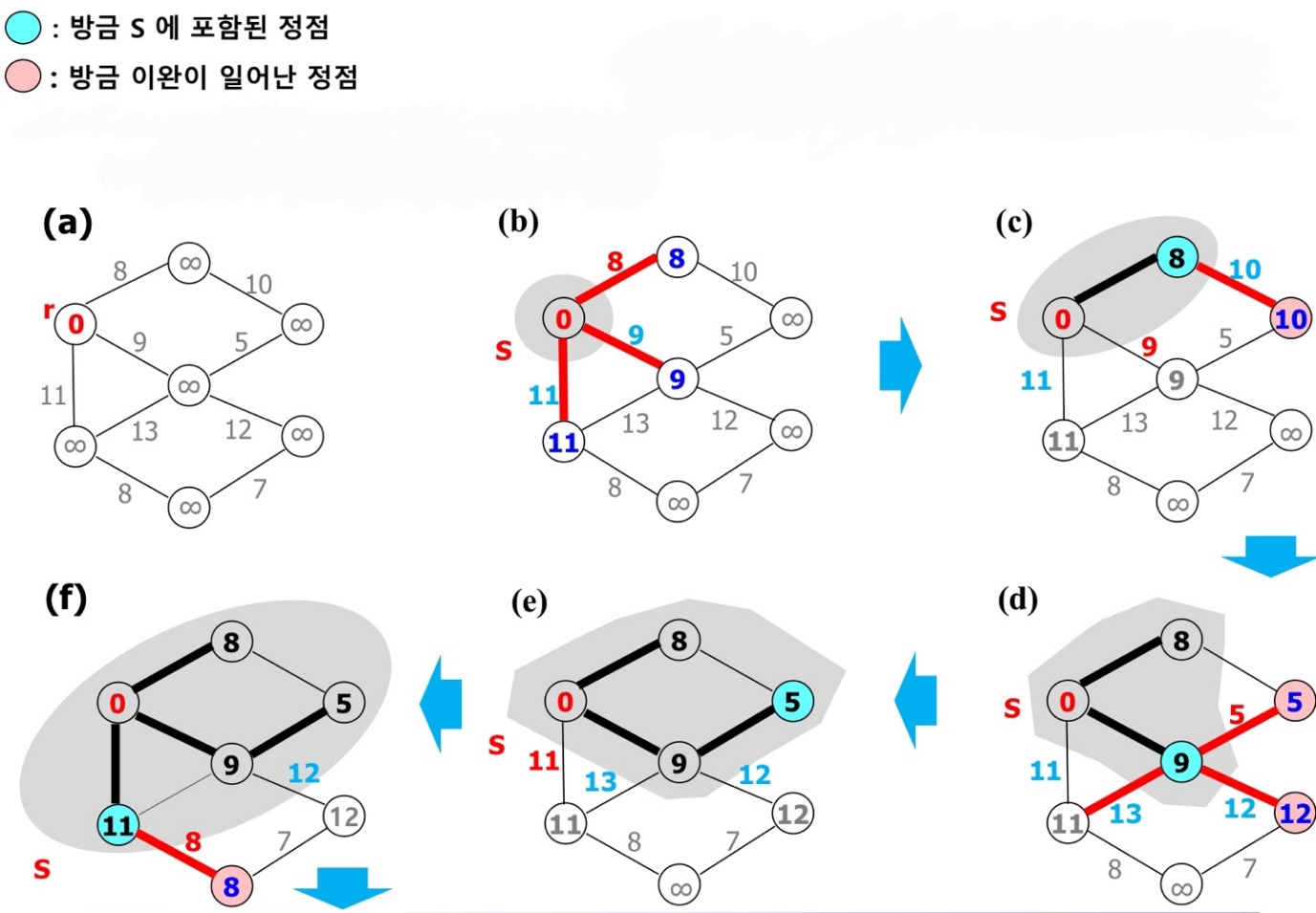
(탐욕 알고리즘의 예)





**# 알고리즘**

****

****

**도표, 스크린샷, 원, 디자인이(가) 표시된 사진

자동 생성된 설명**

**10-3. 위상 정렬(Topological sorting)**

**# 정의**

**- 조건**: 순환이 없는 유향 그래프

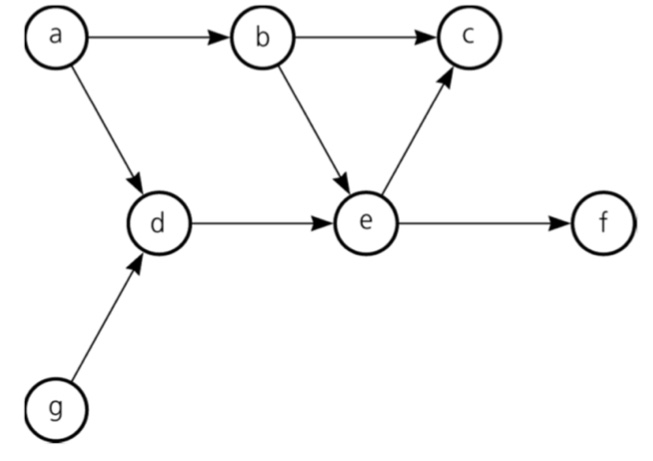
**- 위상 순서**

1. 간선 (x->y)가 존재하면 정점 x는 정점 y에 앞선다.

2. 대개 한 방향 그래프에는 서로 다른 위상 순서가 여럿 존재한다.

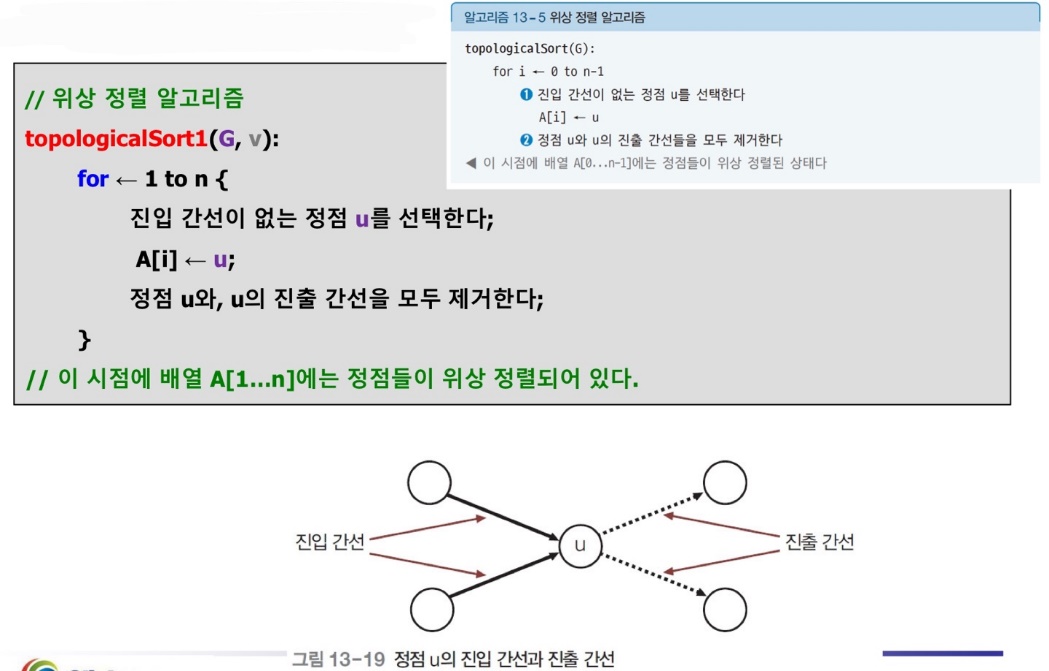
**- 위상 정렬**: 주어진 방향 그래프 G의 위상 순서 중 하나를 찾는다.

ex) 라면 끓이기 작업의 선후 관계&가능한 라면 끓이기 순서의 예

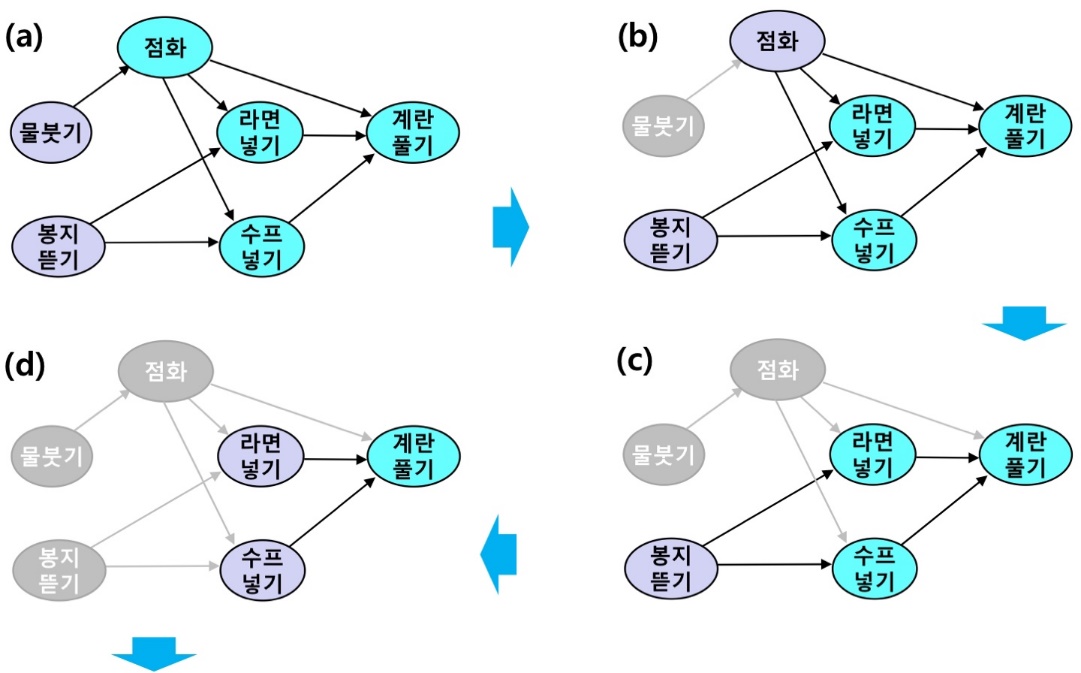
 스케치, 폰트, 원, 라인 아트이(가) 표시된 사진

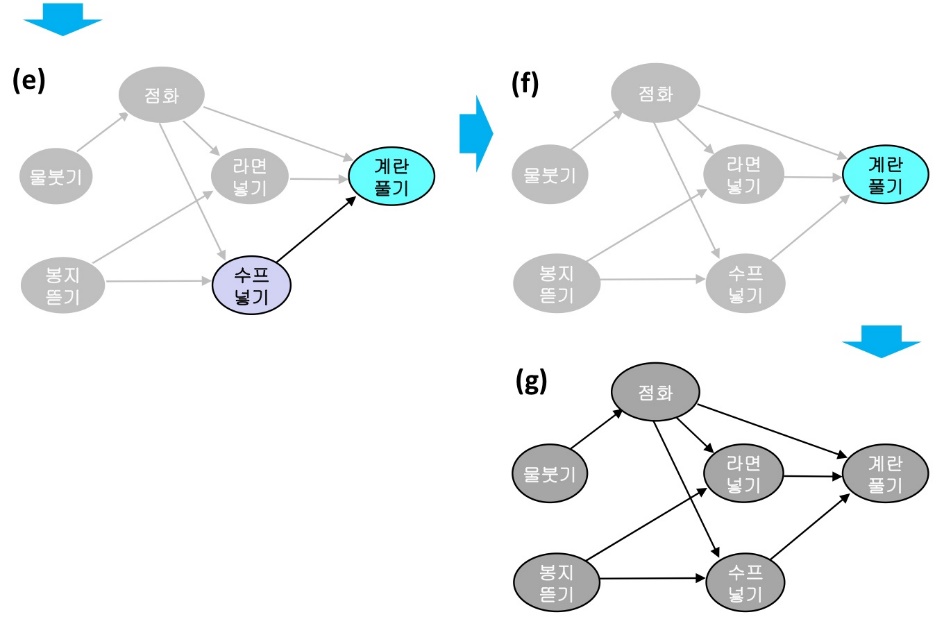
자동 생성된 설명

**# 알고리즘**

****

****

****

****

**10-4. 최단 경로(Shortest Paths)**

- 조건 : 간선의 가중치가 있는 유향 그래프

\* 무향 그래프는 각 간선에 대해 양쪽으로 유향 간선이 있는 그래프로 생각.

🡪 즉, 무향간선(u, v)는 유향 간선(u, v)와 (v, u)을 의미한다고 가정하면 된다.

**# 두 정점 사이의 최단 경로**

: 두 정점 사이의 경로들 중 간선의 가중치 합이 최소인 경로

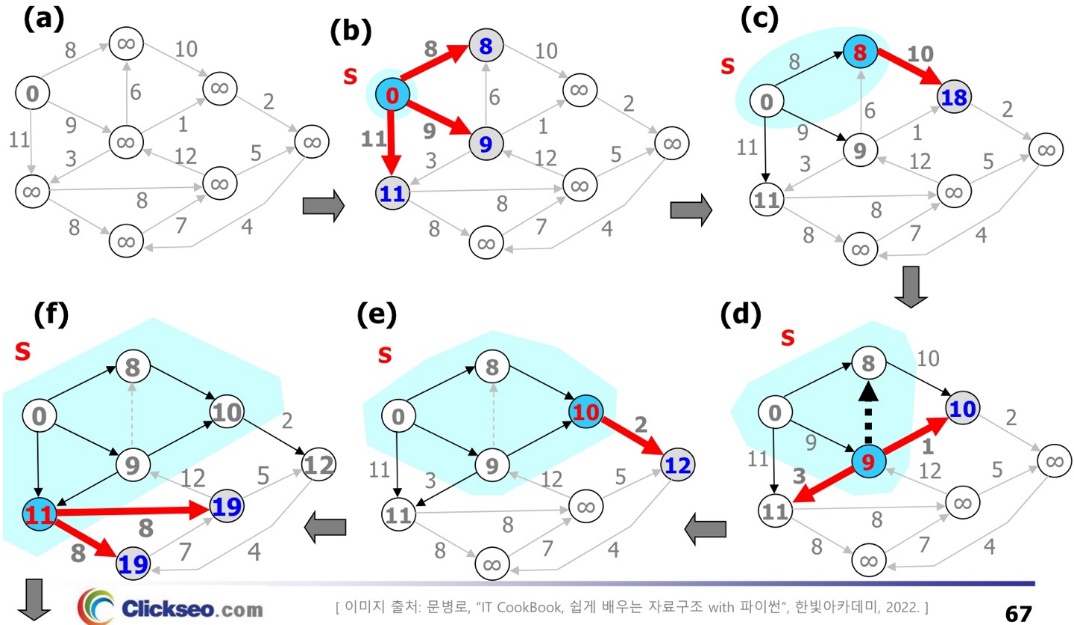
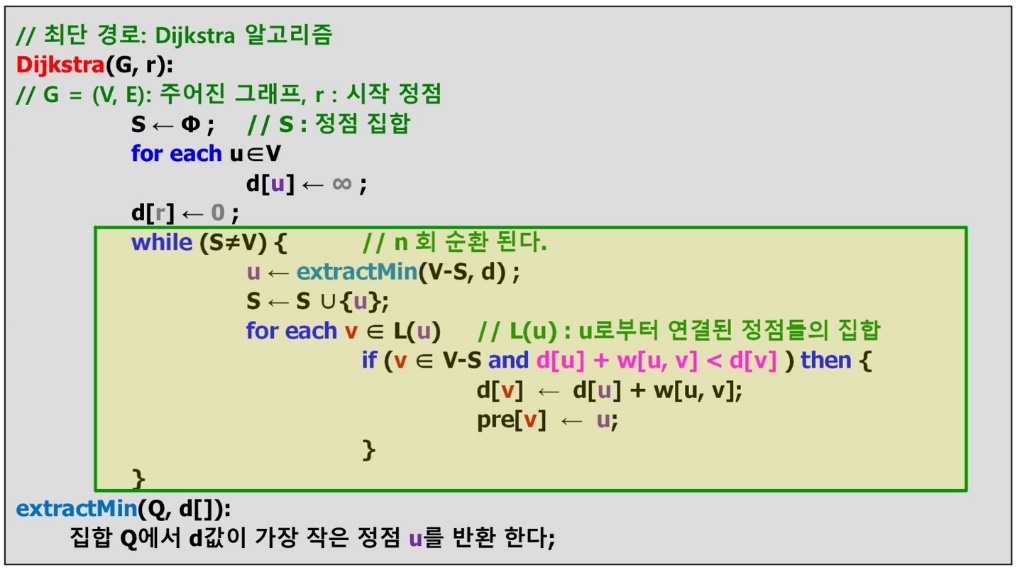
\* 간선 가중치의 합이 음인 싸이클이 있으면 문제가 정의되지 않는다.

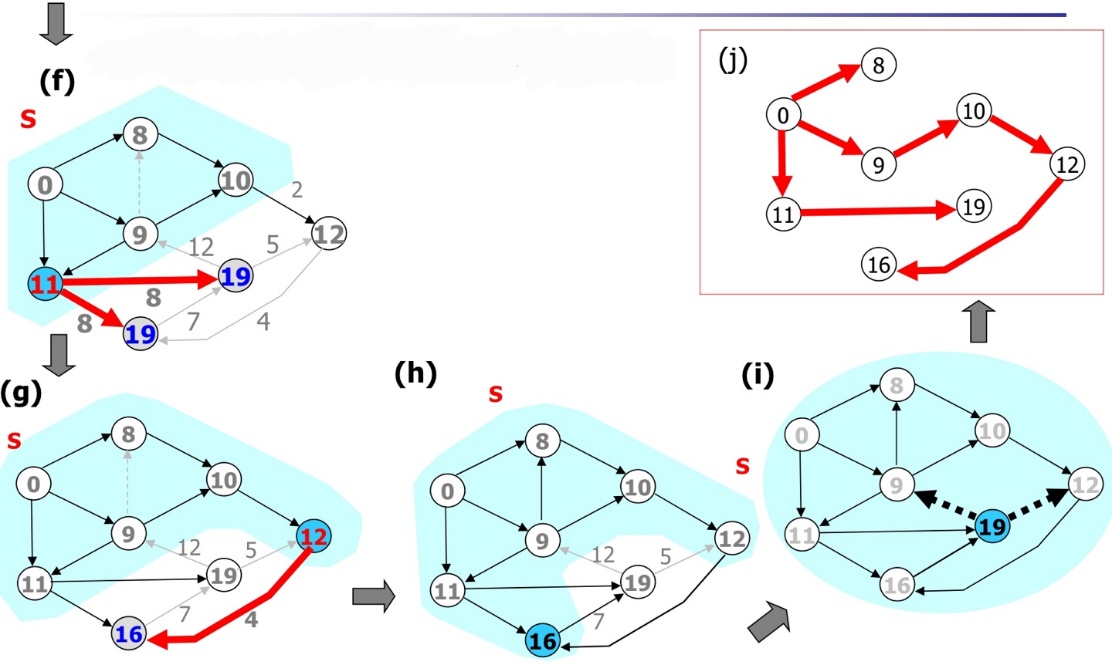
**# 단일 시작점 최단 경로**

: 단일 시작점으로부터 각 정점에 이르는 최단경로를 구한다.

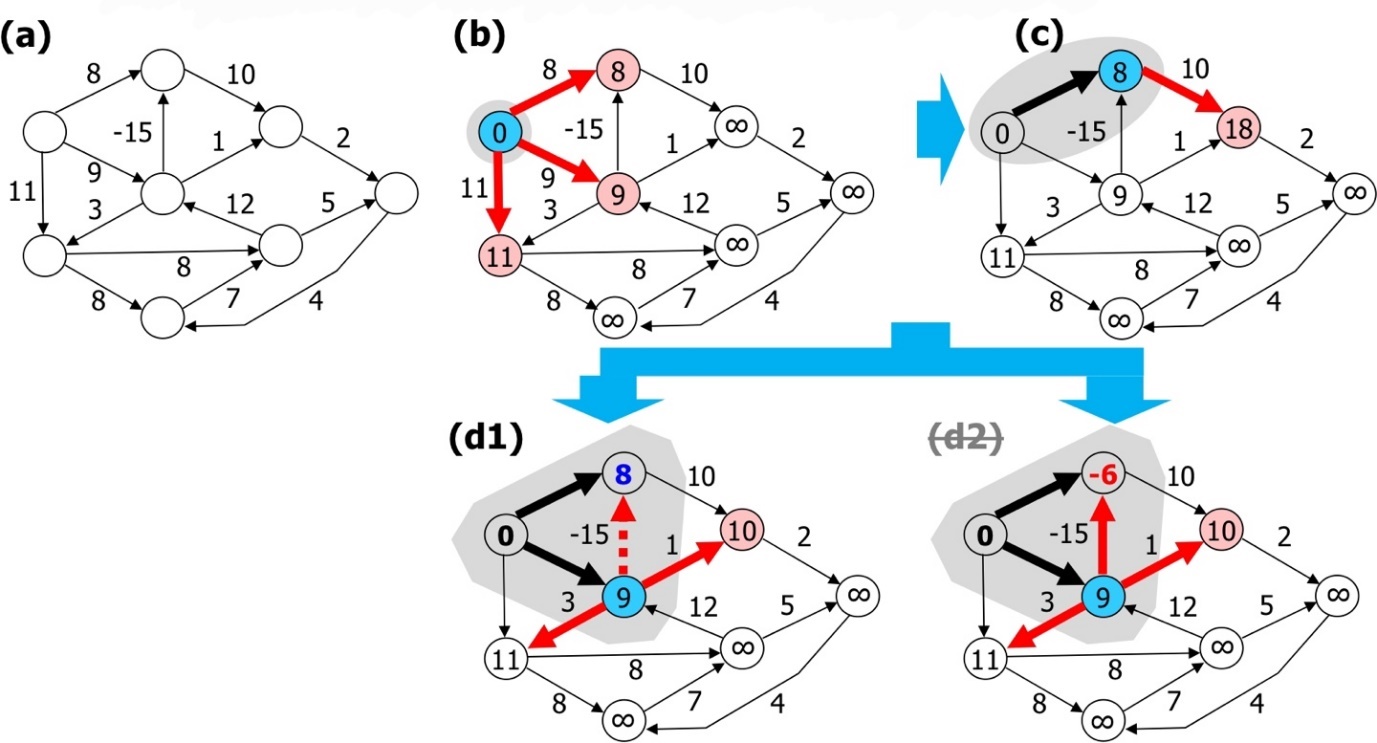
**1. Dijkstra 알고리즘**

: 음의 가중치를 허용하지 않는 최단 경로

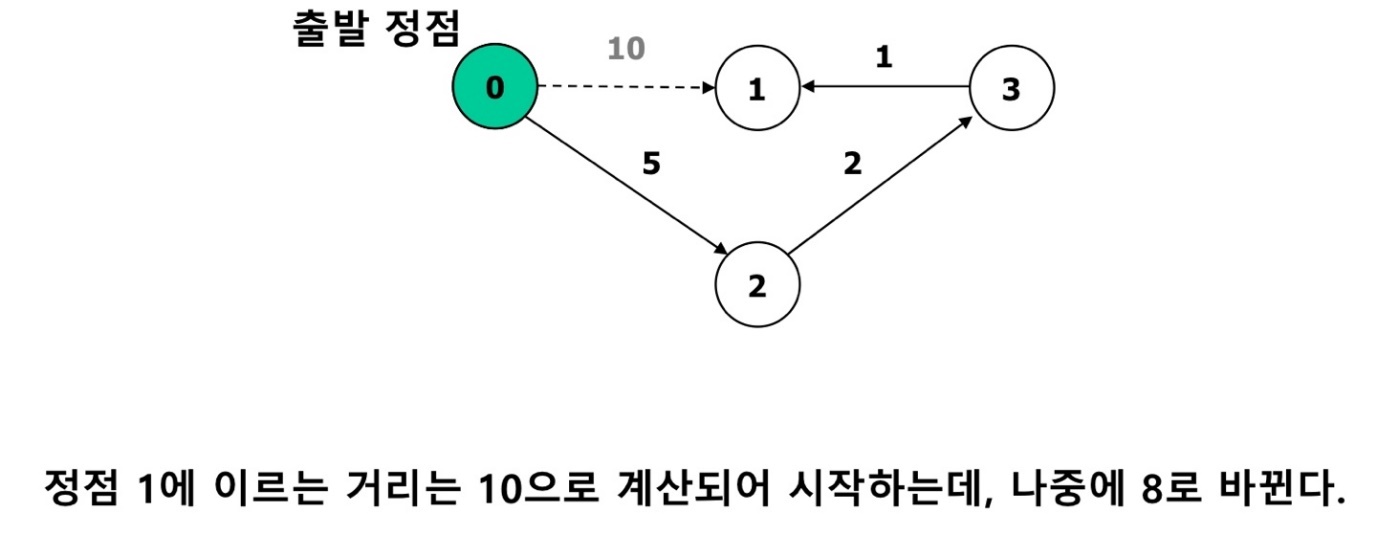




**ex1) 알고리즘이 작동하지 않은 예**



**ex2) 이완의 예**

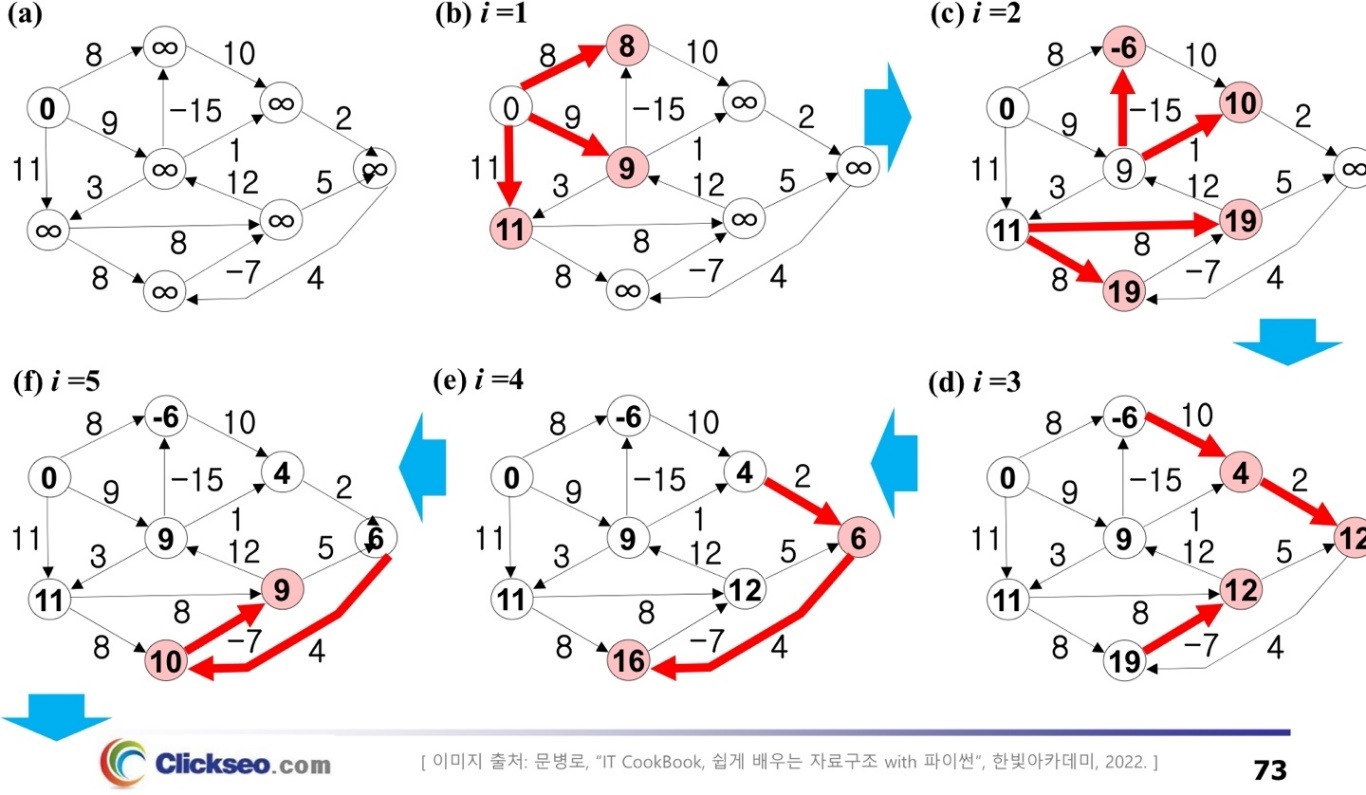


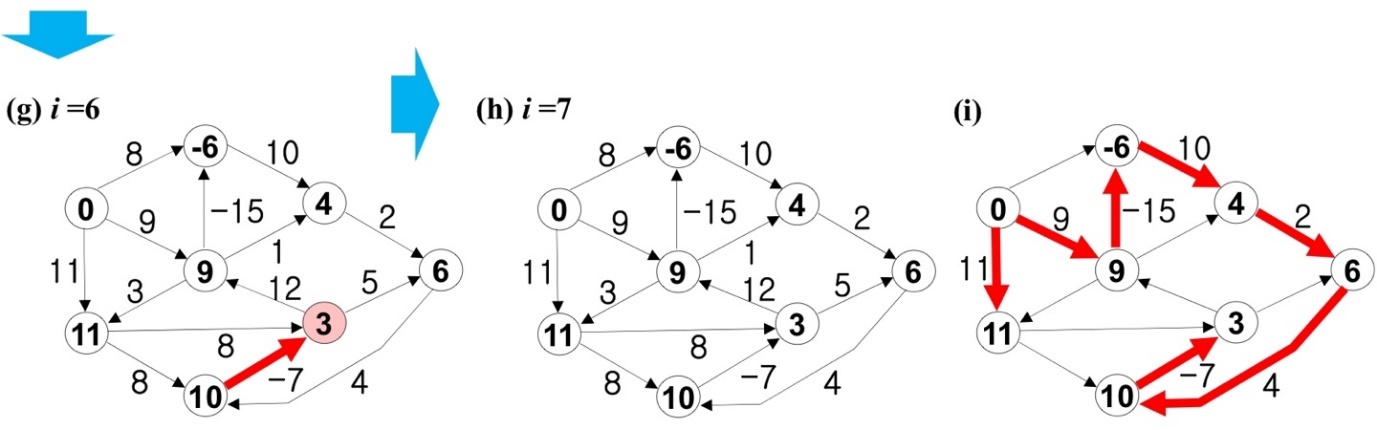
**2-1. Bellman-Ford 알고리즘**

: 음의 가중치를 허용하는 최단 경로

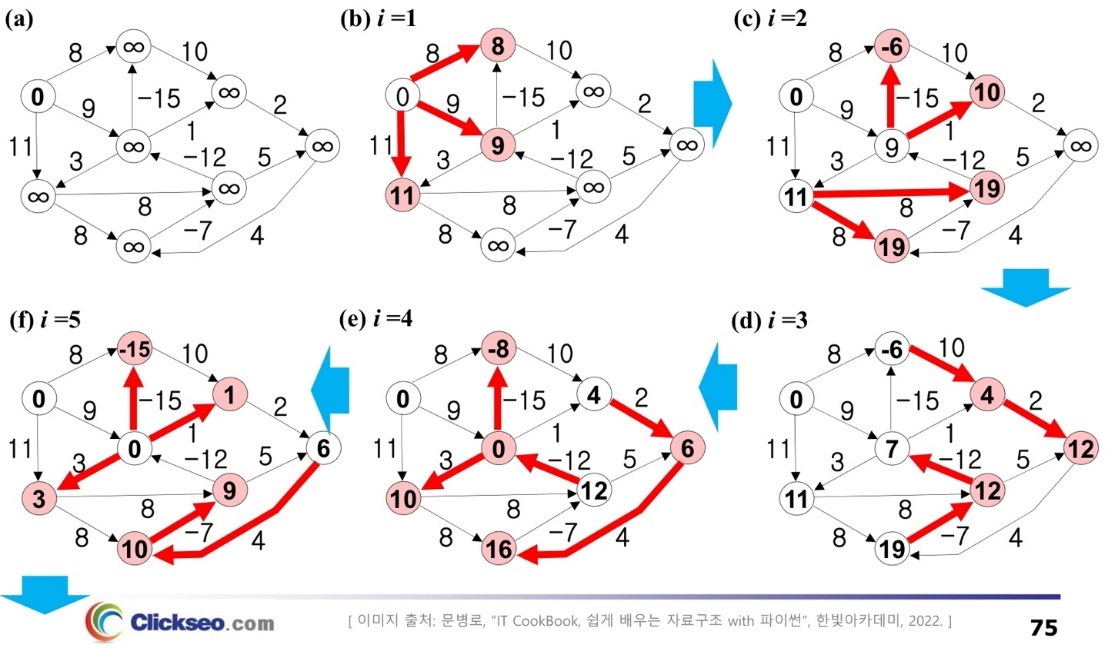
텍스트, 스크린샷, 폰트이(가) 표시된 사진

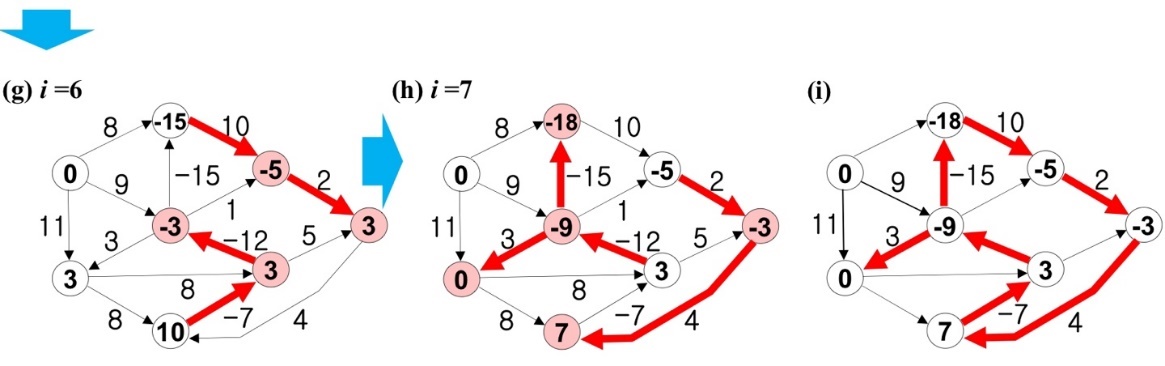
자동 생성된 설명





**ex1) 음의 사이클이 있는 경우**

****

****

**2-2. Bellman-Ford 알고리즘(동적 프로그래밍)**

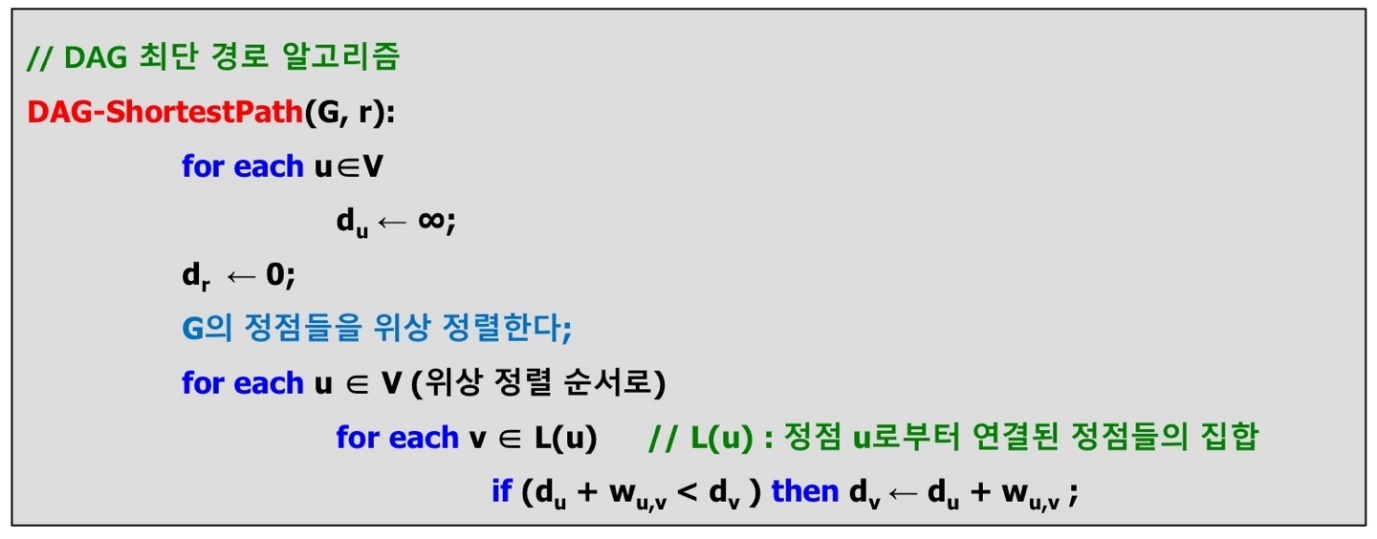
**텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

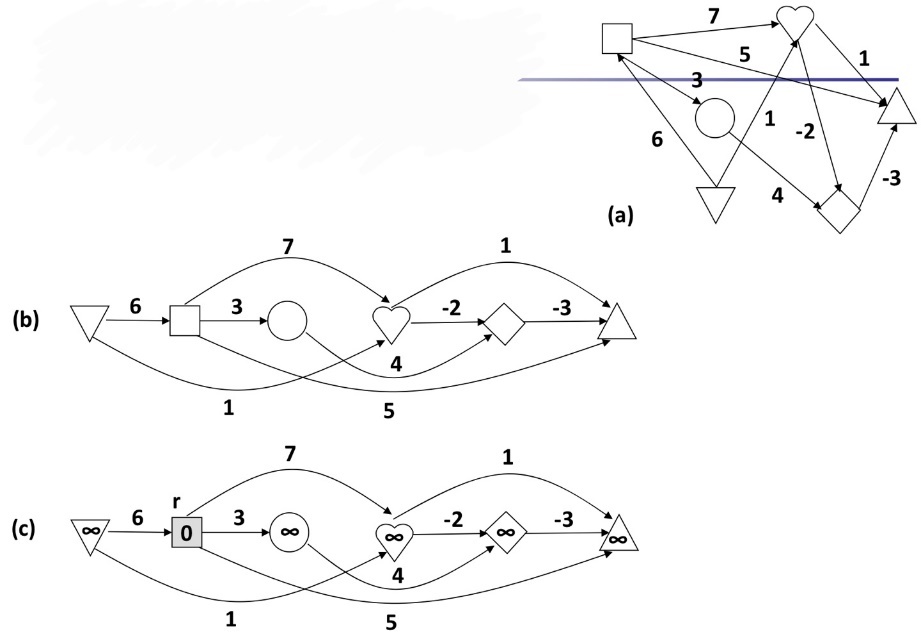
자동 생성된 설명**

**3. 순환이 없는 유향 그래프(DAG)의 최단 경로**

: DAG에서의 최단경로는 선형시간에 구할 수 있다.

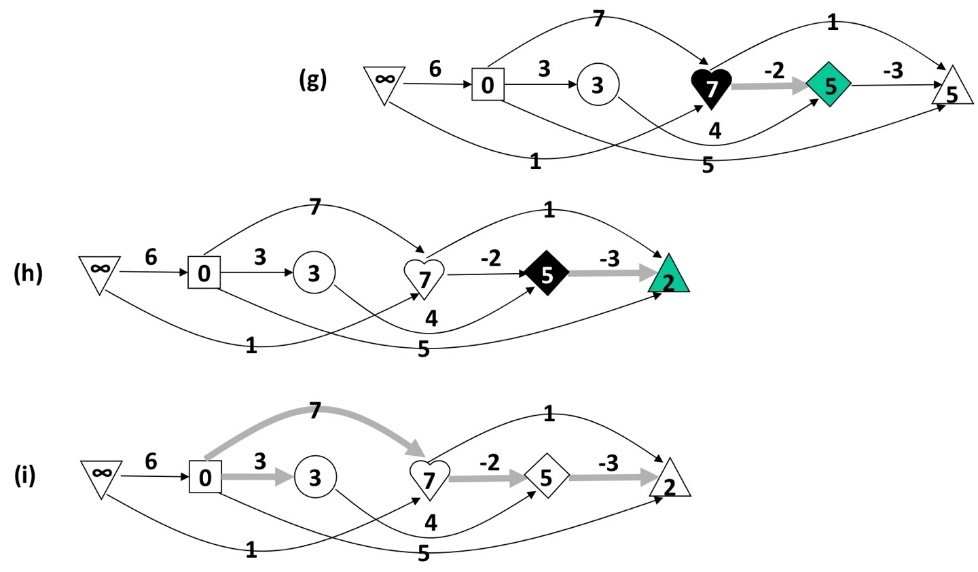
**\*DAG : Directed Acyclic Graph**

****

****

**그림, 스케치, 도표, 원이(가) 표시된 사진

자동 생성된 설명**

****

**# 모든 쌍 최단경로(Floyd-Warshall 알고리즘)**

: 모든 정점 쌍 사이의 최단경로를 모두 구한다.

ex) Road Atlas, 네비게이션 시스템, 네트워크 커뮤니케이션



텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**10-5. DFS, BFS**

#include <iostream>

#include <stack>

#include <queue>

using namespace std;

// GNode class

class GNode {

private:

int \_\_vertex; // 정점

int \_\_weight; // 가중치

GNode\* \_\_link;

friend class GraphType;

public:

GNode(int vertex, int weight);

};

// 그래프 노드(C): 그래프 노드 생성

GNode::GNode(int vertex = 0, int weight = 0)

: \_\_vertex(vertex), \_\_weight(weight), \_\_link(nullptr) {}

// GraphType class

class GraphType {

private:

int \_\_vertex; // 정점의 개수

GNode\*\* \_\_adjSList; // 인접 리스트

public:

GraphType(int vertex);

~GraphType(void);

void insertEdge(int vertex1, int vertex2, int weight);

void DFSAdjSList(int vertex);

void BFSAdjSList(int vertex);

void printAdjSList(void) const;

};

// GraphType: 생성자(소멸자)와 메소드 정의

GraphType::GraphType(int vertex) : \_\_vertex(vertex) {

\_\_adjSList = new GNode \* [vertex + 1];

memset(\_\_adjSList, 0, sizeof(GNode\*) \* (vertex + 1));

}

// graphDestroy : 그래프 삭제

GraphType::~GraphType(void) {

GNode\* old;

for (int i = 0; i < \_\_vertex; i++) {

old = \_\_adjSList[i];

while (old) {

\_\_adjSList[i] = old->\_\_link;

delete old;

old = \_\_adjSList[i];

}

}

delete[] \_\_adjSList;

}

// insertEdge : 간선 추가

void GraphType::insertEdge(int row, int col, int weight) {

if (row >= \_\_vertex || col >= \_\_vertex) {

// cout << "그래프에 없는 정점입니다!!!" << endl;

return;

}

GNode\* newGNode = new GNode(col, weight);

if (\_\_adjSList[row] == nullptr)

\_\_adjSList[row] = newGNode;

else {

GNode\* rNode = \_\_adjSList[row];

while (rNode->\_\_link)

rNode = rNode->\_\_link;

rNode->\_\_link = newGNode;

}

}

// printAdjMatrix : 그래프 전체 출력

void GraphType::printAdjSList(void) const {

char ch;

for (int i = 0; i < \_\_vertex; i++) {

ch = i + 65;

cout << "\t정점 " << ch << "의 인접 리스트";

GNode\* tNode = \_\_adjSList[i];

while (tNode) {

cout.width(3);

ch = tNode->\_\_vertex + 65;

cout << ch << " ->>";

tNode = tNode->\_\_link;

}

cout << " NULL" << endl;

}

}

//그래프 순회: 깊이 우선 탐색(DFS)

void GraphType::DFSAdjSList(int vertex){

//각 정점의 방문(Visited) 여부: true, false

int\* pVisited = new int[\_\_vertex];

memset(pVisited, 0, sizeof(int) \* \_\_vertex);

stack<int> S;

S.push(vertex);// Start Vertex

pVisited[vertex] = true; //정점 방문

char ch = vertex + 65;

cout.width(3); cout << ch;

int v = vertex;

GNode\* w;

while (!S.empty()) {

w = \_\_adjSList[v];

while (w){//인접한 정점이 있는 동안...

// 방문 안한 인접 정점에 대해서 수행

if (!pVisited[w->\_\_vertex]) {

S.push(w->\_\_vertex);

pVisited[w->\_\_vertex]=true;//정점 방문

ch = w->\_\_vertex + 65;

cout.width(3); cout << ch;

v = w->\_\_vertex;

w = \_\_adjSList[v];

}

else w = w->\_\_link;

}

//방문안한 인접 정점이 없으면 스택 pop

v = S.top(); S.pop();

}

cout << endl;

delete[] pVisited;

}

//그래프 순회: 너비 우선 탐색(BFS)

void GraphType::BFSAdjSList(int vertex) {

// 각 정점의 방문(Visited) 여부 : true, false

int\* pVisited = new int[\_\_vertex];

memset(pVisited, 0, sizeof(int) \* \_\_vertex);

queue<int> Q;

Q.push(vertex);// Start Vertex

pVisited[vertex] = true;// 정점 방문

char ch = vertex + 65;

cout.width(3); cout << ch;

int v = vertex;

GNode\* w;

while (!Q.empty()) {

v = Q.front(); Q.pop();

w = \_\_adjSList[v];

while (w){//인접한 정점이 있는 동안..

//방문 안한 인접 정점에 대해서 수행

if (!pVisited[w->\_\_vertex]) {

Q.push(w->\_\_vertex);

pVisited[w->\_\_vertex] = true;//정점 방문

ch = w->\_\_vertex + 65;

cout.width(3); cout << ch;

}

w = w->\_\_link;

}

}

cout << endl;

delete[] pVisited;

}

int main(void) {

// G9 : 무향 그래프

GraphType G9 = GraphType(7);

// 정점: A(0)

G9.insertEdge(0, 1, 0); // A(0) - B(1)

G9.insertEdge(0, 2, 0); // A(0) - C(2)

// 정점: B(1)

G9.insertEdge(1, 0, 0); // B(1) - A(0)

G9.insertEdge(1, 3, 0); // B(1) - D(3)

G9.insertEdge(1, 4, 0); // B(1) - E(4)

// 정점: C(2)

G9.insertEdge(2, 0, 0); // C(2) - A(0)

G9.insertEdge(2, 4, 0); // C(2) - E(4)

// 정점: D(3)

G9.insertEdge(3, 1, 0); // D(3) - B(6)

G9.insertEdge(3, 6, 0); // D(3) - G(6)

// 정점: E(4)

G9.insertEdge(4, 1, 0); // E(4) - B(1)

G9.insertEdge(4, 2, 0); // E(4) - C(2)

G9.insertEdge(4, 6, 0); // E(4) - G(6)

// 정점: F(5)

G9.insertEdge(5, 6, 0); // F(5) - G(6)

// 정점: G(6)

G9.insertEdge(6, 5, 0); // G(6) - F(5)

G9.insertEdge(6, 4, 0); // G(6) - E(4)

G9.insertEdge(6, 3, 0); // G(6) - D(3)

cout << "\n##### 그래프(G9): 인접 리스트 #####\n" << endl;

G9.printAdjSList();

printf("\n##### 그래프(G9): 깊이 우선 탐색(DFS) #####\n\n");

G9.DFSAdjSList(0);

printf("\n##### 그래프(G9): 너비 우선 탐색(BFS) #####\n\n");

G9.BFSAdjSList(0);

return 0;

}

**10-6. AdjMatrix**

/\*

그래프 표현(인접 행렬): 알고리즘 구현

파일명: GraphAdjMatrix.cpp

- main : 그래프 생성 및 정점 추가

클래스 : GraphType

- 그래프 생성.삭제: GraphType, ~GraphType

- 그래프 간선 추가: insertEdge

- 그래프 전체 출력: printAdjMatrix

\*/

#include <iostream>

using namespace std;

// GraphType class

class GraphType {

private:

int \_\_vertex; // 정점의 개수

int\*\* \_\_adjMatrix; // 인접 행렬

public:

GraphType(int vertex);

~GraphType(void);

void insertEdge(int vertex1, int vertex2, int weight);

void printAdjMatrix(void) const;

};

// 생성자: 그래프 삭제

GraphType::GraphType(int vertex) : \_\_vertex(vertex) {

// 인접 행렬: 2차원 배열 동적 메모리 할당

\_\_adjMatrix = new int\* [vertex + 1];

for (int i = 0; i < vertex; i++) {

\_\_adjMatrix[i] = new int[vertex];

memset(\_\_adjMatrix[i], 0, sizeof(int) \* vertex);

}

\_\_adjMatrix[vertex] = nullptr;

}

// 소멸자: 그래프 삭제

GraphType::~GraphType(void) {

for (int i = 0; i < \_\_vertex; i++)

delete[] \_\_adjMatrix[i];

delete[] \_\_adjMatrix;

\_\_vertex = 0;

}

// 그래프 간선 추가

void GraphType::insertEdge(int row, int col, int weight = 1) {

if (row >= \_\_vertex || col >= \_\_vertex) {

// cout << "그래프에 없는 정점입니다!!!" << endl;

return;

}

\_\_adjMatrix[row][col] = weight;

}

// 그래프 전체 출력

void GraphType::printAdjMatrix(void) const {

for (int i = 0; i < \_\_vertex; i++) {

cout << "\n\t";

for (int j = 0; j < \_\_vertex; j++) {

cout.width(3);

cout << \_\_adjMatrix[i][j];

}

}

cout << endl;

}

int main(void)

{

// G2 : 무향 그래프

GraphType G2 = GraphType(3);

// 정점: A(0)

G2.insertEdge(0, 1, 1); // A(0) - B(1)

G2.insertEdge(0, 2, 1); // A(0) - C(2)

// 정점: B(1)

G2.insertEdge(1, 0, 1); // B(1) - A(0)

G2.insertEdge(1, 2, 1); // B(1) - C(2)

// 정점: C(2)

G2.insertEdge(2, 1, 1); // C(2) - B(1)

G2.insertEdge(2, 0, 1); // C(2) - A(0)

cout << "\n##### 그래프(G2): 인접 행렬 #####" << endl;

G2.printAdjMatrix();

// G4 : 유향 그래프

GraphType G4 = GraphType(3);

// 정점(A)

G4.insertEdge(0, 1, 1); // A(0) - B(1)

G4.insertEdge(0, 2, 1); // A(0) - C(2)

// 정점(B)

G4.insertEdge(1, 0, 1); // B(1) - A(0)

G4.insertEdge(1, 2, 1); // B(1) - C(2)

cout << "\n##### 그래프(G4): 인접 행렬 #####" << endl;

G4.printAdjMatrix();

return 0;

}

**10-7. AdjList**

/\*

그래프 표현(인접 리스트): 알고리즘 구현

파일명: AdjList.cpp

- main : 그래프 생성 및 정점 추가

클래스 : GraphType

- 그래프 생성.삭제: GraphType, ~GraphType

- 그래프 간선 추가: insertEdge

- 그래프 전체 출력: printAdjList

\*/

#include <iostream>

using namespace std;

//GNode class

class GNode {

private:

int \_\_vertex;//정점

int \_\_weight;//가중치

GNode\* \_\_link;

friend class GraphType;

public:

GNode(int vertex, int weight);

};

//그래프 노드(C): 그래프 노드 생성

GNode::GNode(int vertex, int weight)

:\_\_vertex(vertex), \_\_weight(weight), \_\_link(nullptr){}

// GraphType class

class GraphType {

private:

int \_\_vertex; // 정점의 개수

GNode\*\* \_\_adjSList; // 인접 리스트

public:

GraphType(int vertex);

~GraphType(void);

void insertEdge(int vertex1, int vertex2, int weight);

void printAdjList(void) const;

};

//생성자(소멸자)와 메소드 정의

GraphType::GraphType(int vertex) : \_\_vertex(vertex) {

// 인접 행렬: 2차원 배열 동적 메모리 할당

\_\_adjSList=new GNode\*[vertex + 1];

memset(\_\_adjSList, 0, sizeof(GNode\*) \* (vertex + 1));

}

//graphDestroy: 그래프 삭제

GraphType::~GraphType(void) {

GNode\* old;

for (int i = 0; i < \_\_vertex; i++) {

old = \_\_adjSList[i];

while (old) {

\_\_adjSList[i] = old->\_\_link;

delete old;

old = \_\_adjSList[i];

}

}

delete[]\_\_adjSList;

}

//insertEdge: 그래프 간선 추가

void GraphType::insertEdge(int row, int col, int weight = 1) {

if (row >= \_\_vertex || col >= \_\_vertex) {

// cout << "그래프에 없는 정점입니다!!!" << endl;

return;

}

GNode\* newGNode = new GNode(col, weight);

if (\_\_adjSList[row] == nullptr)

\_\_adjSList[row] = newGNode;

else {

GNode\* rNode = \_\_adjSList[row];

while (rNode->\_\_link)

rNode = rNode->\_\_link;

rNode->\_\_link = newGNode;

}

}

//printAdjList: 그래프 전체 출력

void GraphType::printAdjList(void) const {

char ch;

for (int i = 0; i < \_\_vertex; i++) {

ch = i + 65;

cout << "\t정점 " << ch << "의 인접 리스트";

GNode\* tNode = \_\_adjSList[i];

while (tNode){

cout.width(3);

ch = tNode->\_\_vertex + 65;

cout << ch << "->>";

tNode = tNode->\_\_link;

}

cout << " NULL" << endl;

}

}

int main(void)

{

// G2 : 무향 그래프

GraphType G2 = GraphType(3);

// 정점: A(0)

G2.insertEdge(0, 1, 0); // A(0) - B(1)

G2.insertEdge(0, 2, 0); // A(0) - C(2)

// 정점: B(1)

G2.insertEdge(1, 0, 0); // B(1) - A(0)

G2.insertEdge(1, 2, 0); // B(1) - C(2)

// 정점: C(2)

G2.insertEdge(2, 1, 0); // C(2) - B(1)

G2.insertEdge(2, 0, 0); // C(2) - A(0)

cout << "\n##### 그래프(G2): 인접 행렬 #####" << endl;

G2.printAdjList();

// G4 : 유향 그래프

GraphType G4 = GraphType(3);

// 정점(A)

G4.insertEdge(0, 1, 0); // A(0) - B(1)

G4.insertEdge(0, 2, 0); // A(0) - C(2)

// 정점(B)

G4.insertEdge(1, 0, 0); // B(1) - A(0)

G4.insertEdge(1, 2, 0); // B(1) - C(2)

cout << "\n##### 그래프(G4): 인접 행렬 #####" << endl;

G4.printAdjList();

return 0;

}

**10-8. AdjsList(DFS, BFS) ver.Seo**

/\*

그래프 표현(인접 리스트) : 알고리즘 구현

파일명 : GraphAdjSListTraversal.cpp

- main : 그래프 생성 및 간선 추가

클래스 : GNode

클래스 : GraphType

- 그래프 생성.소멸 : GraphType, ~GraphType

- 그래프 간선 추가 : insertEdge

- 그래프 순회 : DFSAdjSList, BFSAdjSList

- 그래프 전체 출력 : printAdjSList

\*/

#include <iostream>

#include <stack>

#include <queue>

using namespace std;

// GNode class

class GNode {

private:

int \_\_vertex; // 정점

int \_\_weight; // 가중치

GNode\* \_\_link;

friend class GraphType;

public:

GNode(int vertex, int weight);

};

// 그래프 노드(C): 그래프 노드 생성

GNode::GNode(int vertex = 0, int weight = 0)

: \_\_vertex(vertex), \_\_weight(weight), \_\_link(nullptr) {}

// GraphType class

class GraphType {

private:

int \_\_vertex; // 정점의 개수

GNode\*\* \_\_adjSList; // 인접 리스트

public:

GraphType(int vertex);

~GraphType(void);

void insertEdge(int vertex1, int vertex2, int weight);

void DFSAdjSList(int vertex);

void BFSAdjSList(int vertex);

void printAdjSList(void) const;

};

// 그래프 생성(생성자)

GraphType::GraphType(int vertex) : \_\_vertex(vertex) {

\_\_adjSList = new GNode \* [vertex + 1];

memset(\_\_adjSList, 0, sizeof(GNode\*) \* (vertex + 1));

}

// 그래프 삭제(소멸자)

GraphType::~GraphType(void) {

GNode\* old;

for (int i = 0; i < \_\_vertex; i++) {

old = \_\_adjSList[i];

while (old) {

\_\_adjSList[i] = old->\_\_link;

delete old;

old = \_\_adjSList[i];

}

}

delete[] \_\_adjSList;

}

// 그래프 간선 추가

void GraphType::insertEdge(int row, int col, int weight) {

if (row >= \_\_vertex || col >= \_\_vertex) {

// cout << "그래프에 없는 정점입니다!!!" << endl;

return;

}

GNode\* newGNode = new GNode(col, weight);

if (\_\_adjSList[row] == nullptr)

\_\_adjSList[row] = newGNode;

else {

GNode\* rNode = \_\_adjSList[row];

while (rNode->\_\_link)

rNode = rNode->\_\_link;

rNode->\_\_link = newGNode;

}

}

// 그래프 순회: 깊이 우선 탐색(DFS)

void GraphType::DFSAdjSList(int vertex) {

}

// 그래프 순회: 너비 우선 탐색(BFS)

void GraphType::BFSAdjSList(int vertex) {

}

// 그래프 전체 출력

void GraphType::printAdjSList(void) const {

char ch;

for (int i = 0; i < \_\_vertex; i++) {

ch = i + 65;

cout << "\t정점 " << ch << "의 인접 리스트";

GNode\* tNode = \_\_adjSList[i];

while (tNode) {

cout.width(3);

ch = tNode->\_\_vertex + 65;

cout << ch << " ->>";

tNode = tNode->\_\_link;

}

cout << " NULL" << endl;

}

}

int main(void)

{

// G9 : 무향 그래프

GraphType G9 = GraphType(7);

// 정점: A(0)

G9.insertEdge(0, 1, 0); // A(0) - B(1)

G9.insertEdge(0, 2, 0); // A(0) - C(2)

// 정점: B(1)

G9.insertEdge(1, 0, 0); // B(1) - A(0)

G9.insertEdge(1, 3, 0); // B(1) - D(3)

G9.insertEdge(1, 4, 0); // B(1) - E(4)

// 정점: C(2)

G9.insertEdge(2, 0, 0); // C(2) - A(0)

G9.insertEdge(2, 4, 0); // C(2) - E(4)

// 정점: D(3)

G9.insertEdge(3, 1, 0); // D(3) - B(6)

G9.insertEdge(3, 6, 0); // D(3) - G(6)

// 정점: E(4)

G9.insertEdge(4, 1, 0); // E(4) - B(1)

G9.insertEdge(4, 2, 0); // E(4) - C(2)

G9.insertEdge(4, 6, 0); // E(4) - G(6)

// 정점: F(5)

G9.insertEdge(5, 6, 0); // F(5) - G(6)

// 정점: G(6)

G9.insertEdge(6, 3, 0); // G(6) - D(3)

G9.insertEdge(6, 4, 0); // G(6) - E(4)

G9.insertEdge(6, 5, 0); // G(6) - F(5)

cout << "\n##### 그래프(G9): 인접 리스트 #####\n" << endl;

G9.printAdjSList();

printf("\n##### 그래프(G9): 깊이 우선 탐색(DFS) #####\n\n");

G9.DFSAdjSList(6);

printf("\n##### 그래프(G9): 너비 우선 탐색(BFS) #####\n\n");

G9.BFSAdjSList(6);

return 0;

}