

그래프

Data Structures and Algorithms

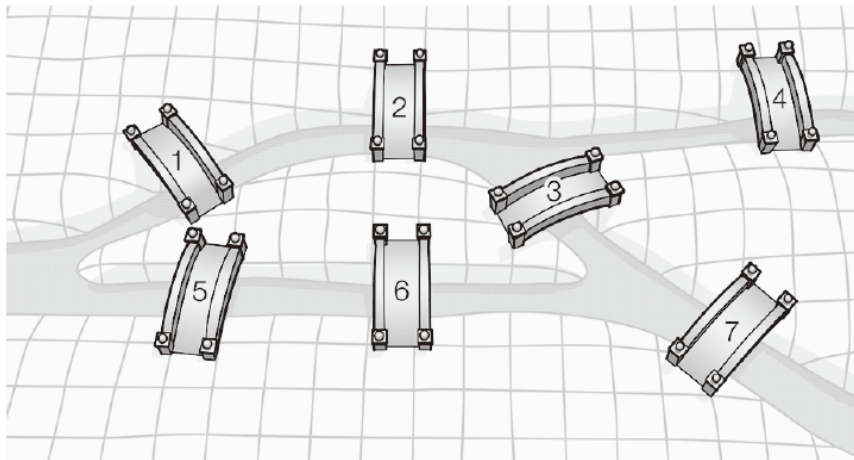
목차

- 그래프와 종류
- 인접 리스트 기반 그래프
- 탐색
- 최소 비용 신장 트리

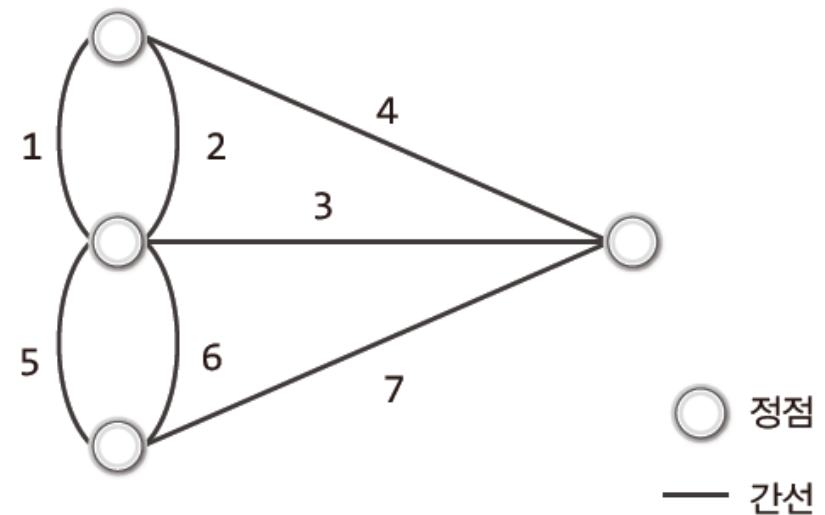
그래프와 종류

The Story

- Euler's Bridge (Koenigsberg); 한 붓 그리기
 - 모든 다리를 한만 건너서 처음 출발했던 장소로 돌아올 수 있는가?
 - 정점에 연결된 간선의 수가 짝수여야 함



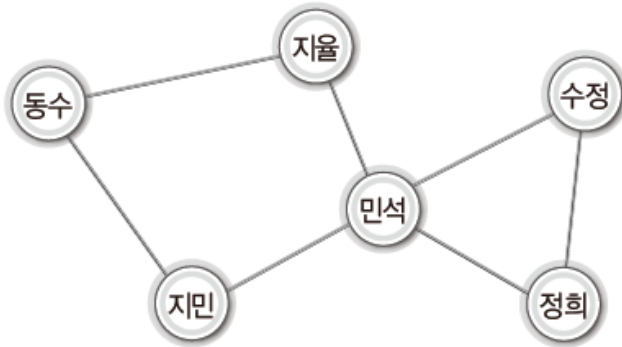
쾨니히스베르크의 다리 문제



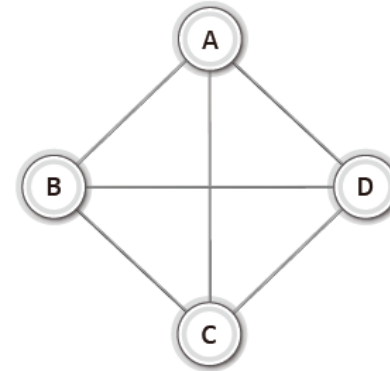
다리 문제의 재 표현

종류

- 무방향 그래프: 방향성 없음

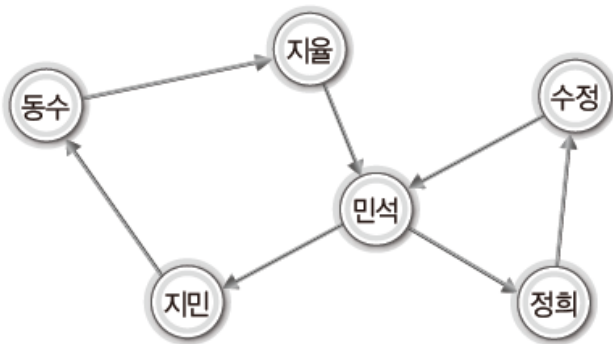


무방향 그래프의 예

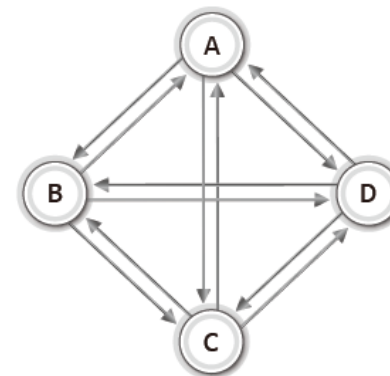


무방향 완전 그래프의 예

- 방향 그래프: 방향성 있음



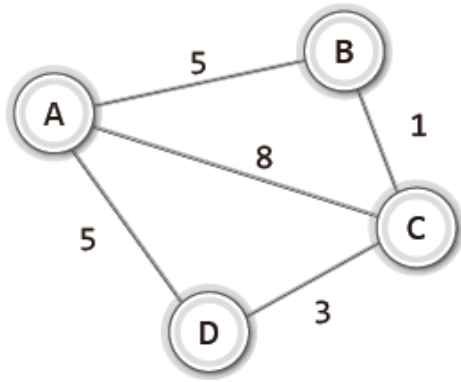
방향 그래프의 예



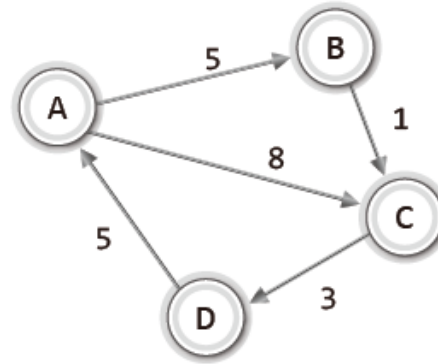
방향 완전 그래프의 예

가중치 그래프와 부분 그래프

- 가중치 그래프: 비용을 포함하는 간선

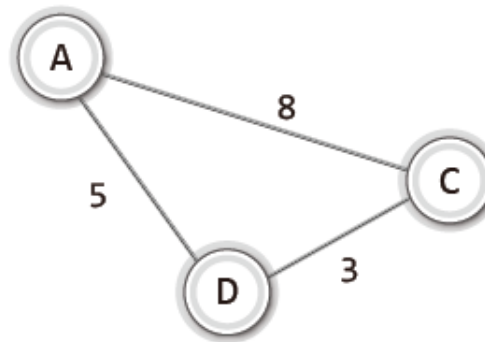
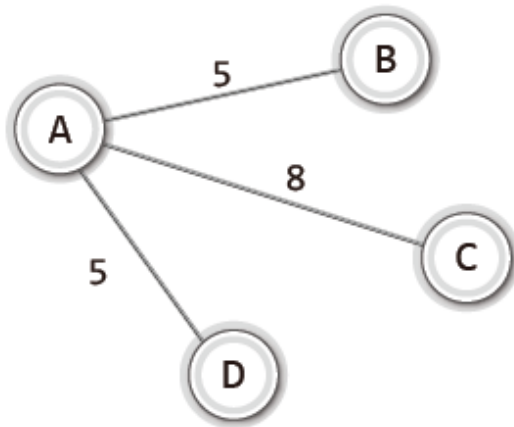


무방향 가중치 그래프의 예



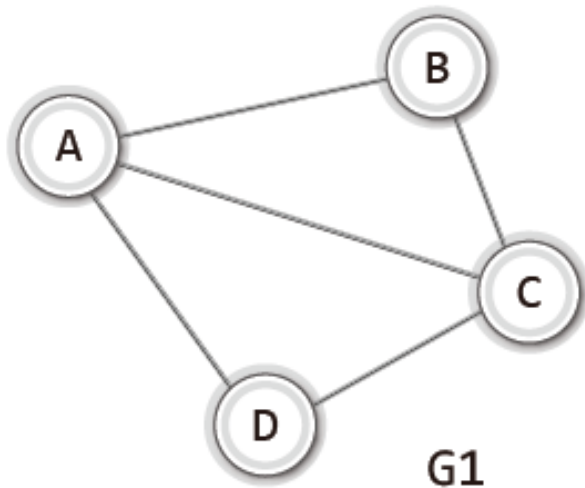
방향 가중치 그래프의 예

- 부분 그래프: 일부 정점과 간선으로 구성된 그래프



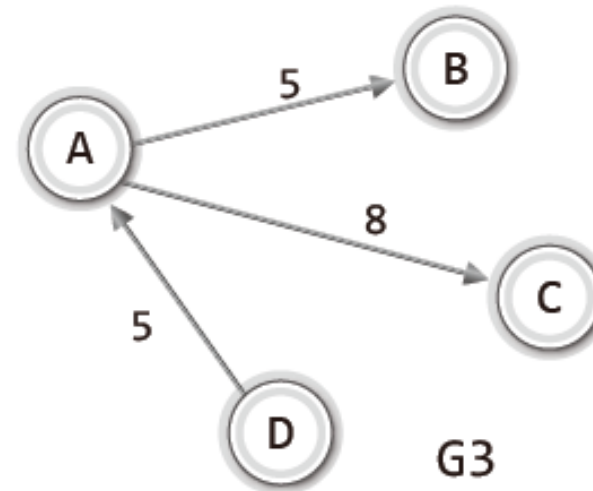
집합으로 표현

- 그래프 G 의 정점 집합: $V(G)$
- 그래프 G 의 간선 집합: $E(G)$



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

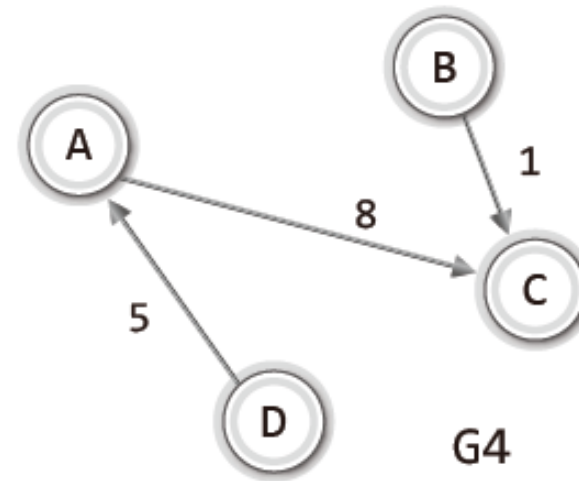
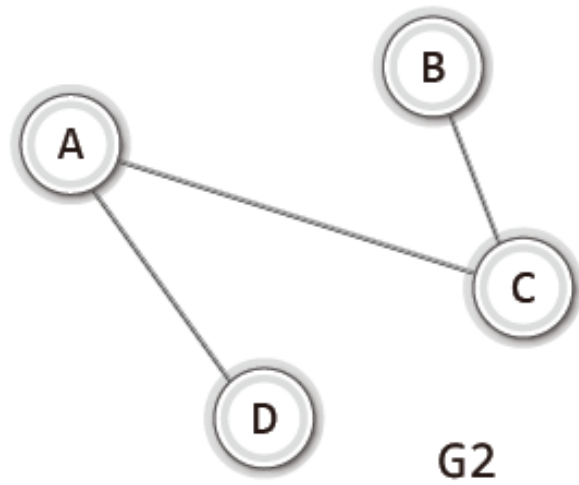
$$E(G1) = \{(A, B), (A, C), (A, D), \\ (B, C), (C, D)\}$$



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

$$E(G3) = \{\langle A, B \rangle, \langle A, C \rangle, \langle D, A \rangle\}$$

집합으로 표현 Ex



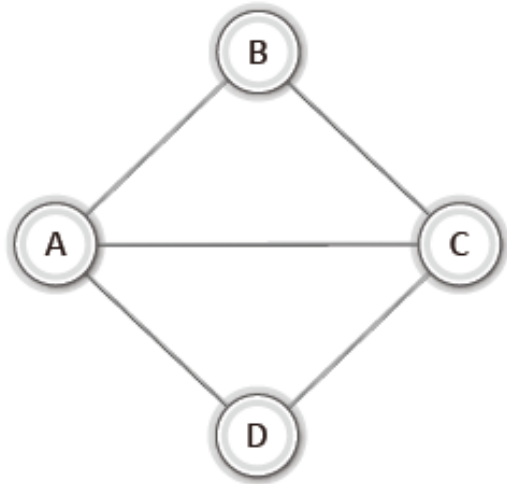
ADT

```
enum {A, B, C, D, E, F, G, H, I, J};
```

```
enum {SEOUL, INCHEON, DAEGU, BUSAN, KWANGJU};
```

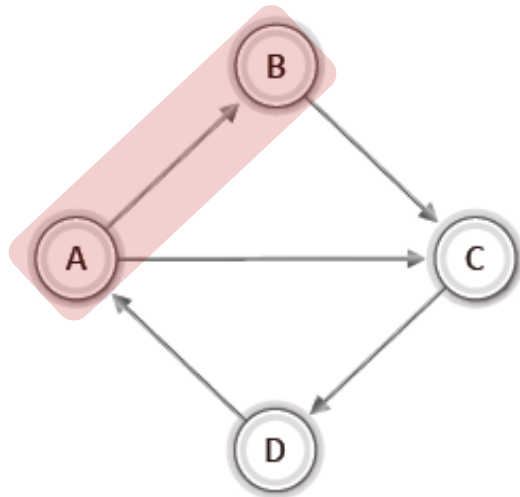
- `void GraphInit(UALGraph * pg, int nv);`
 - 그래프의 초기화를 진행한다.
 - 두 번째 인자로 정점의 수를 전달한다.
- `void GraphDestroy(UALGraph * pg);`
 - 그래프 초기화 과정에서 할당한 리소스를 반환한다.
- `void AddEdge(UALGraph * pg, int fromV, int toV);`
 - 매개변수 fromV와 toV로 전달된 정점을 연결하는 간선을 그래프에 추가한다.
- `void ShowGraphEdgeInfo(UALGraph * pg);`
 - 그래프의 간선정보를 출력한다.

구현: 인접 행렬



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

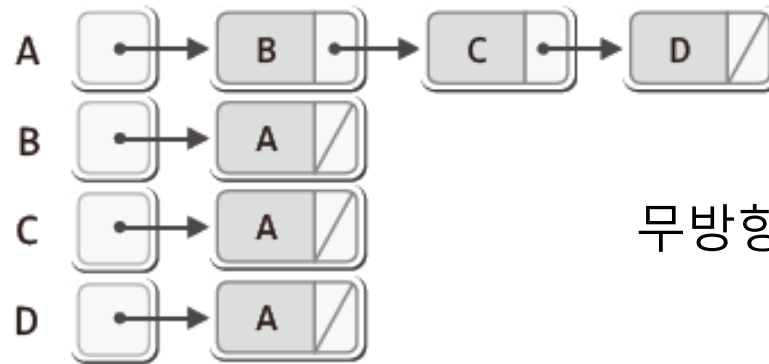
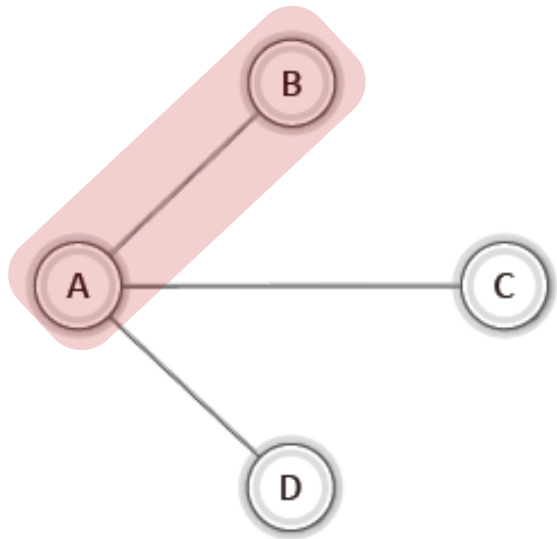
무방향 그래프



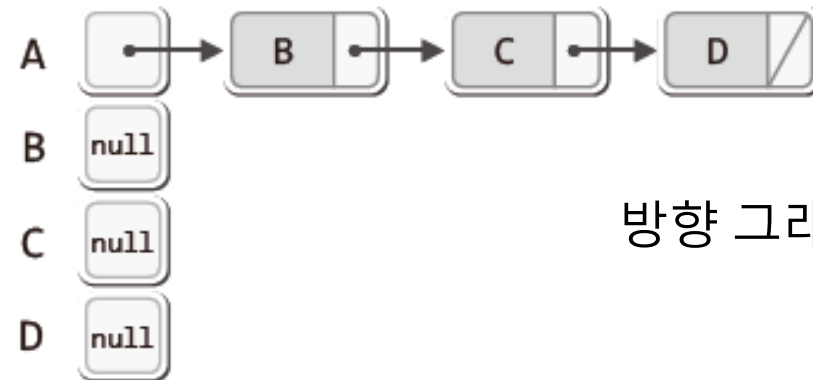
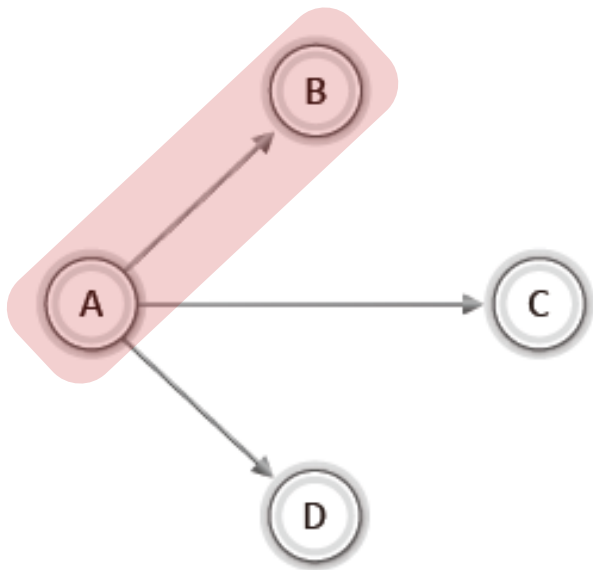
	A	B	C	D
A	0	1	1	0
B	0	0	1	0
C	0	0	0	1
D	1	0	0	0

방향 그래프

구현: 인접 리스트



무방향 그래프



방향 그래프

인접 리스트 기반 그래프

헤더

```
#include "DLinkedList.h"
```

```
enum {A, B, C, D, E, F, G, H, I, J}; // 정점의 이름들을 상수화
```

```
typedef struct _ual  
{  
    int numV;    // 정점의 수  
    int numE;    // 간선의 수  
    List * adjList; // 간선의 정보  
} ALGraph;
```

```
void GraphInit(ALGraph * pg, int nv); // 그래프의 초기화
```

```
void GraphDestroy(ALGraph * pg); // 그래프의 리소스 해제
```

```
void AddEdge(ALGraph * pg, int fromV, int toV); // 간선의 추가
```

```
void ShowGraphEdgeInfo(ALGraph * pg); // 유틸리티 함수: 간선의 정보 출력
```

main.c

```
#include <stdio.h>
#include "ALGraph.h"

int main(void)
{
    ALGraph graph;
    GraphInit(&graph, 5);    // A, B, C, D, E의 정점 생성

    AddEdge(&graph, A, B);
    AddEdge(&graph, A, D);
    AddEdge(&graph, B, C);
    AddEdge(&graph, C, D);
    AddEdge(&graph, D, E);
    AddEdge(&graph, E, A);

    ShowGraphEdgeInfo(&graph);

    GraphDestroy(&graph);
    return 0;
}
```

Can you visualize it?

구현: 초기화와 소멸

```
void GraphInit(ALGraph * pg, int nv)
{
    int i;
    // 정점 개수 만큼 리스트 배열을 생성
    pg->adjList = (List*)malloc(sizeof(List)*nv); // 간선 정보 저장용 리스트
    pg->numV = nv; // 정점의 수는 nv에 저장된 값을 결정
    pg->numE = 0; // 초기의 간선 수는 0개
    // 정점 개수 만큼 리스트 초기화
    for(i=0; i<nv; i++)
    {
        ListInit(&(pg->adjList[i]));
        SetSortRule(&(pg->adjList[i]), WhoIsPrecede);
    }
}

int WhoIsPrecede(int data1,
int data2)
{
    if(data1 < data2)
        return 0;
    else
        return 1;
}

void GraphDestroy(ALGraph * pg) // 그래프 리소스의 해제
{
    if(pg->adjList != NULL)
        free(pg->adjList); // 동적 할당된 연결 리스트 소멸
}
```

구현: 간선 정보 출력

```
void AddEdge(ALGraph * pg, int fromV, int toV) // 간선의 추가
{
    LInsert(&(pg->adjList[fromV]), toV); // 무방향이기 때문에
    LInsert(&(pg->adjList[toV]), fromV); // 양쪽의 노드에서 연결
    pg->numE += 1;
}

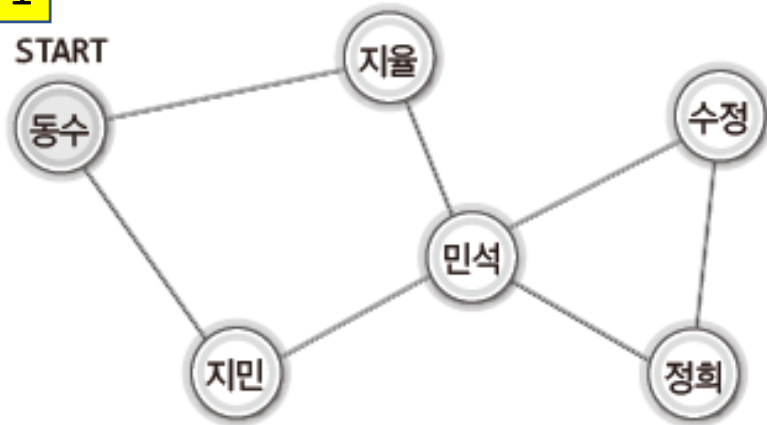
void ShowGraphEdgeInfo(ALGraph * pg) // 유틸리티 함수: 간선의 정보 출력
{
    int vx;
    for(int i=0; i<pg->numV; i++)
    {
        printf("%c와 연결된 정점: ", i + 65);
        if(LFirst(&(pg->adjList[i]), &vx))
        {
            printf("%c ", vx + 65);
            while(LNext(&(pg->adjList[i]), &vx))
                printf("%c ", vx + 65);
        }
        printf("\n");
    }
}
```


탐색

DFS

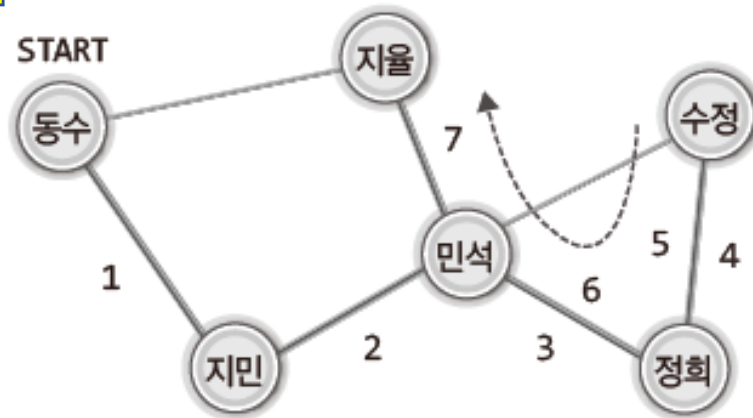
DFS: 깊이 우선 탐색

1



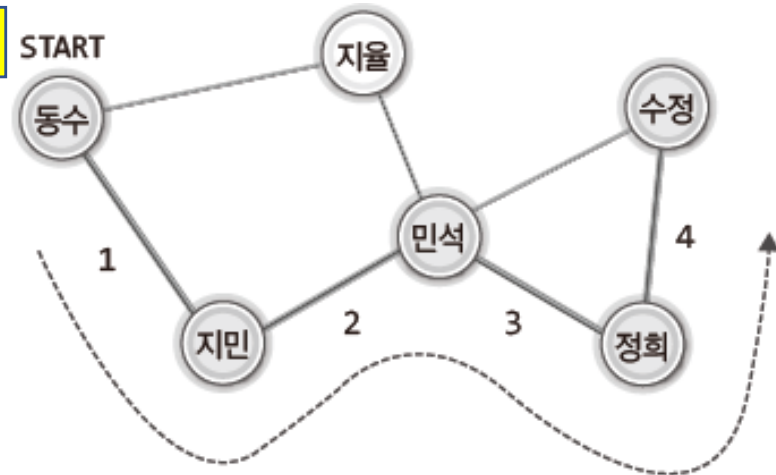
동수로부터 비상 연락망 가동!

3



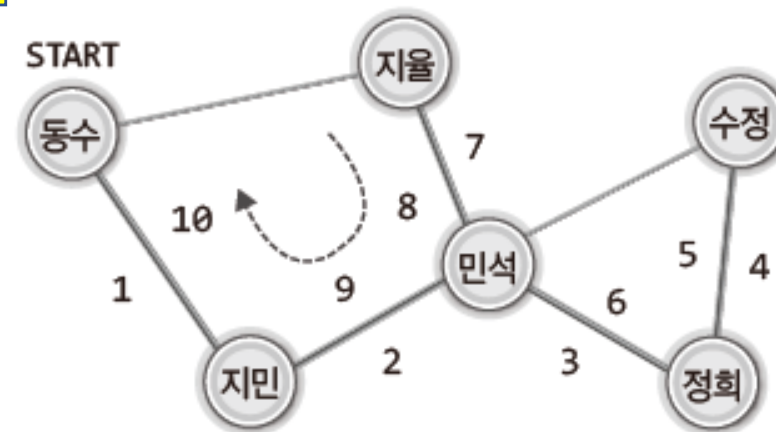
연락 할 곳이 없으면 역으로 되돌아
가면서 연락 취할 곳을 찾음

2



한 사람에게만 연락

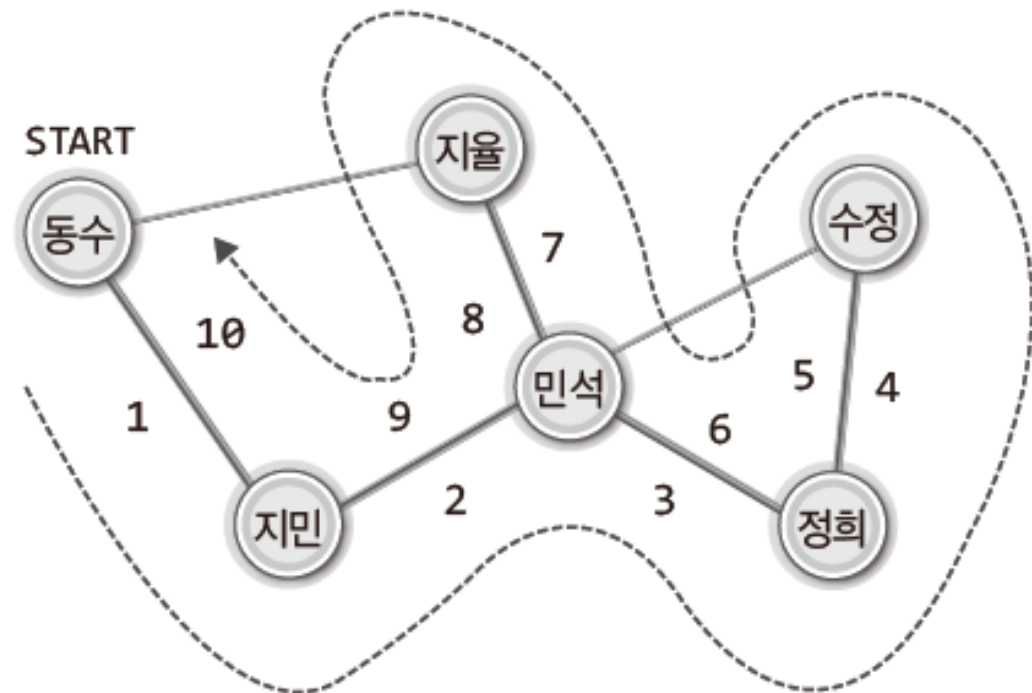
4



시작 점으로 되돌아 오면 연락
끝!

DFS: 깊이 우선 탐색 핵심

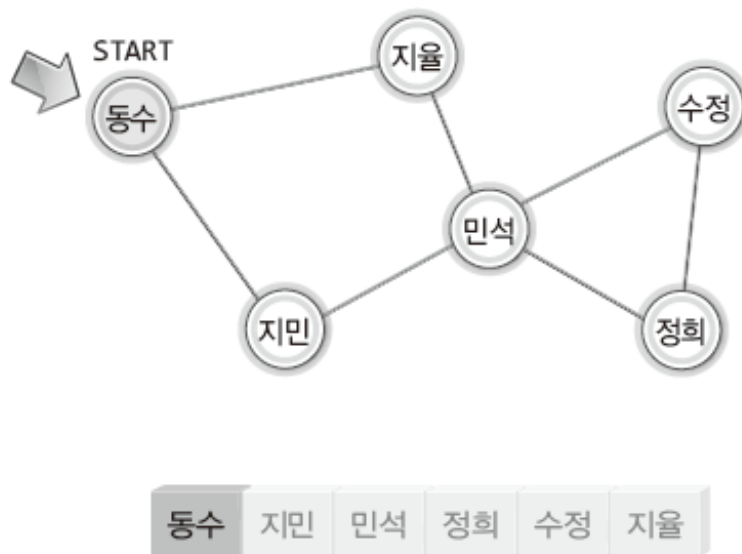
- 하나의 노드만 탐색
- 탐색 할 노드가 없으면 되돌아 감
- 처음 탐색을 시작한 노드로 되돌아 오면 끝



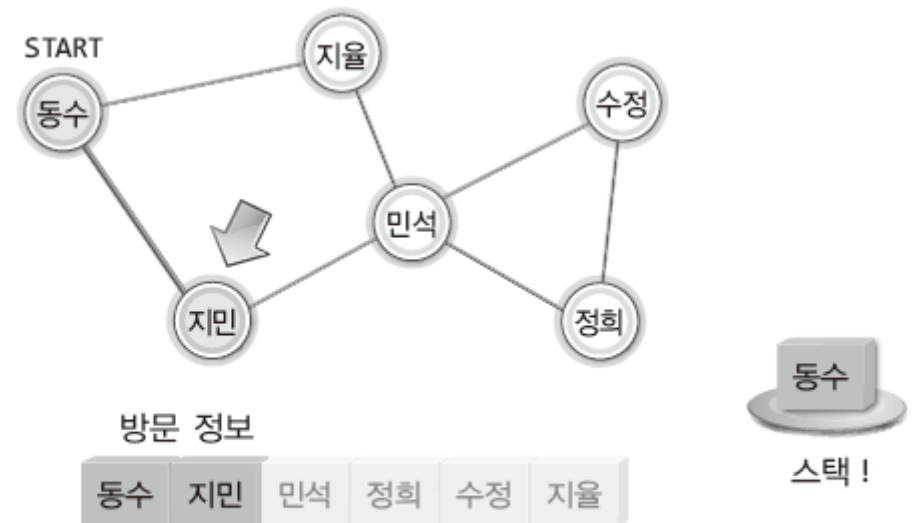
DFS 모델 1

- 배열: 방문 정보 추적 용
- 스택: 경로 정보 추적 용

1



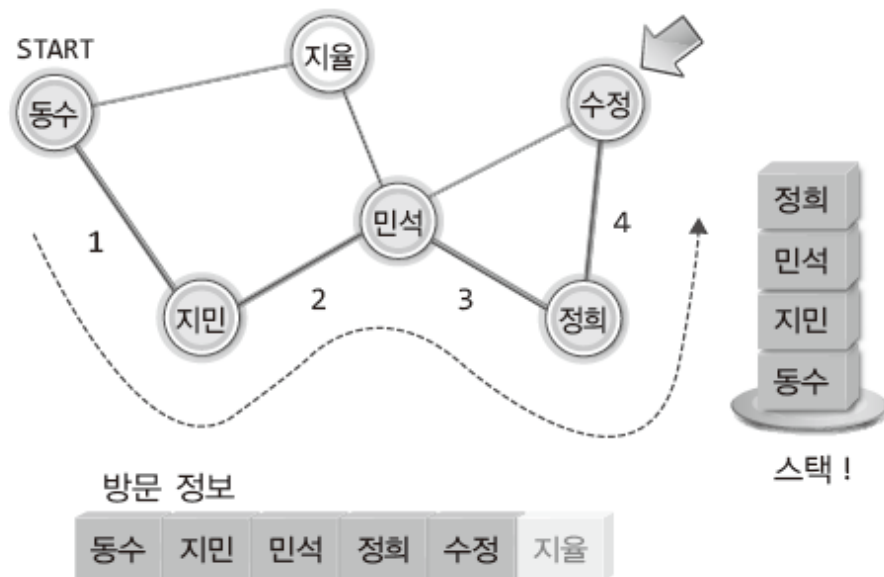
2



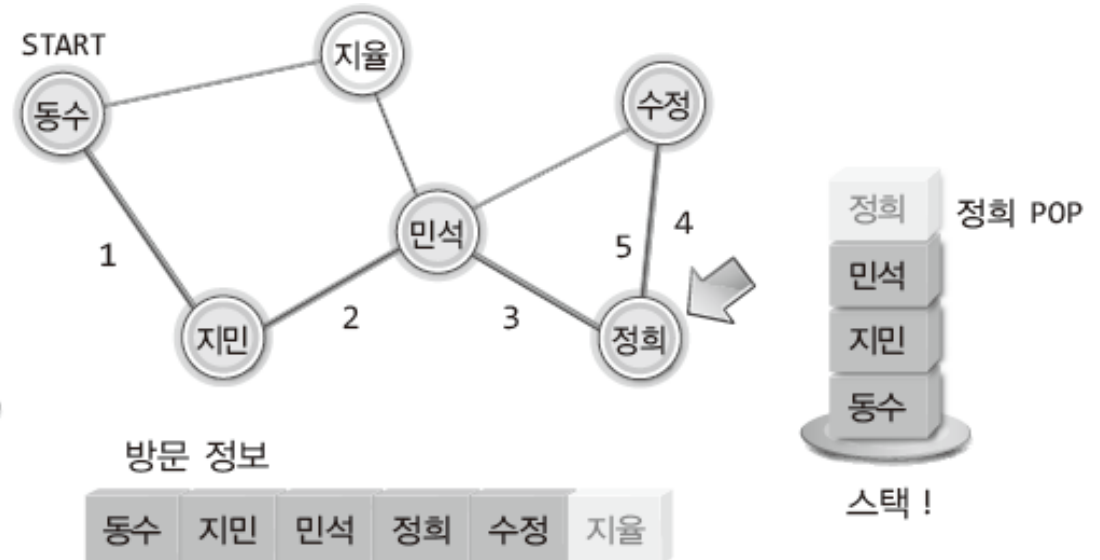
DFS 모델 2

- 되돌아가기 위해 직전 노드에 대한 정보를 스택에서 얻음

3



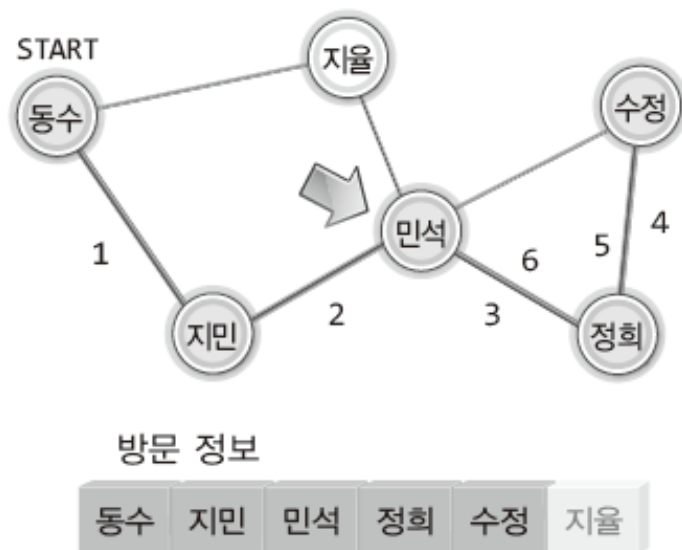
4



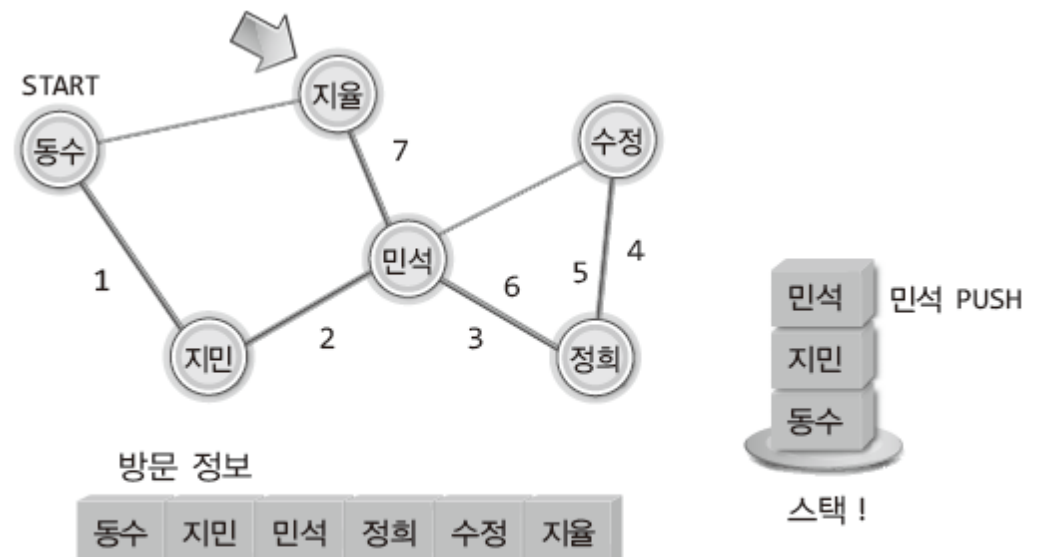
DFS 모델 3

- 이전 방문과 상관없이 연결된 다른 노드 방문 시 스택에 저장

5

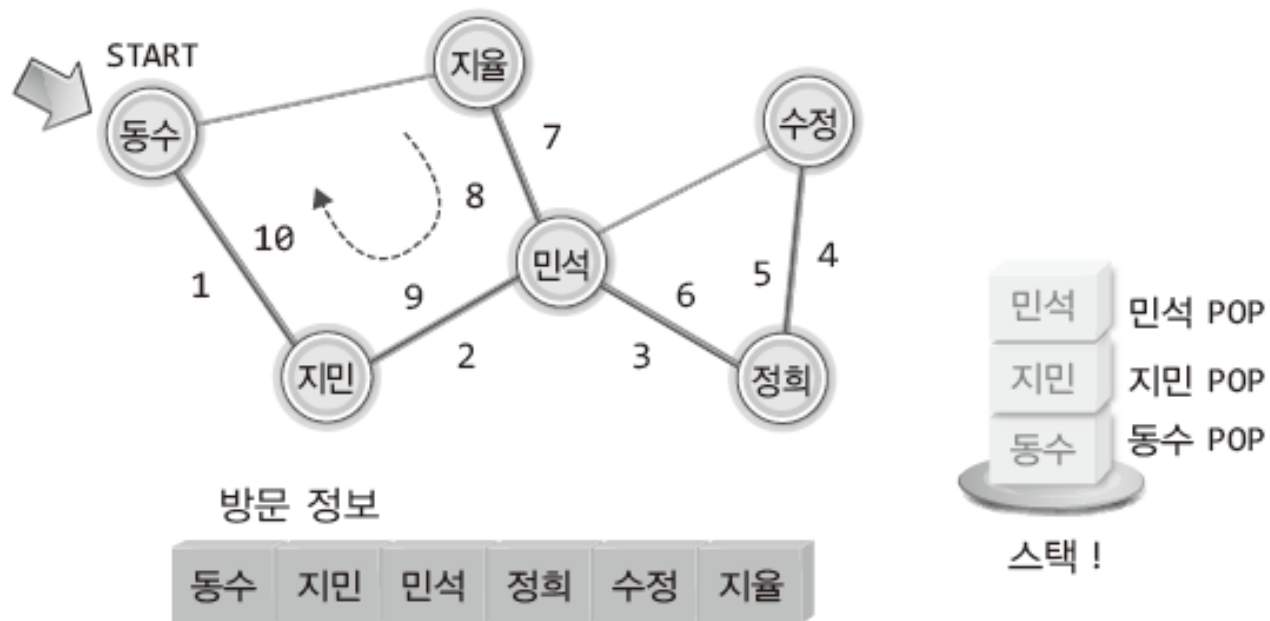


6



DFS 모델 4

- 스택을 반복적으로 pop하면 시작에 도달 가능



파일 구성

- 깊이 우선 탐색의 실제 구현
 - void DFSShowGraphVertex(ALGraph * pg, int startV);
 - 그래프의 모든 정점 정보를 출력하는 함수
 - DFS를 기반으로 정의가 된 함수
- 구현 결과를 반영한 파일의 구성
 - 그래프: ALGraphDFS.h, ALGraphDFS.c (기본 그래프 정의의 확장)
 - 스택: ArrayBaseStack.h, ArrayBaseStack.c
 - 이중연결리스트: DLinkedList.h, DLinkedList.c
 - 메인: DFSMain.c

구현: 헤더

```
#include "DLinkedList.h"
```

```
enum {A, B, C, D, E, F, G, H, I, J}; // 정점의 이름들을 상수화
```

```
typedef struct _ual  
{  
    int numV;    // 정점의 수  
    int numE;    // 간선의 수  
    List * adjList; // 간선의 정보  
    int * visitInfo; // 탐색한 정점 정보  
} ALGraph;
```

```
void GraphInit(ALGraph * pg, int nv); // 그래프의 초기화
```

```
void GraphDestroy(ALGraph * pg); // 그래프의 리소스 해제
```

```
void AddEdge(ALGraph * pg, int fromV, int toV); // 간선의 추가
```

```
void ShowGraphEdgeInfo(ALGraph * pg); // 유틸리티 함수: 간선의 정보 출력
```

```
void DFShowGraphVertex(ALGraph * pg, int startV); // 정점의 정보 출력
```

구현: visitInfo 관련

```
void GraphInit(ALGraph * pg, int nv)
{
    . . . .
    // 정점의 수를 길이로 하여 배열을 할당
    pg->visitInfo = (int *)malloc(sizeof(int) * pg->numV);
    // 배열의 모든 요소를 0으로 초기화!
    memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
}

void GraphDestroy(ALGraph * pg)
{
    . . . .
    // 할당된 배열의 소멸!
    if(pg->visitInfo != NULL)
        free(pg->visitInfo);
}
```

구현: Helper Function

- DFSShowGraphVertex 의 구현에 필요한 함수
 - 방문한 정점 정보를 그래프의 멤버 visitInfo가 가리키는 배열에 등록

// 방문한 정점의 정보를 기록 및 출력

```
int VisitVertex(ALGraph * pg, int visitV)
{
    if(pg->visitInfo[visitV] == 0) // visitV에 처음 방문일 때 '참'인 if문
    {
        pg->visitInfo[visitV] = 1; // visitV에 방문한 것으로 기록
        printf("%c ", visitV + 65); // 방문한 정점의 이름을 출력
        return TRUE;                // 방문 성공!
    }
    return FALSE;                  // 방문 실패!
}
```

구현: DFSShowGraphVertex

```
void DFSShowGraphVertex(ALGraph * pg, int startV)
{
    Stack stack;    int visitV = startV;    int nextV; StackInit(&stack);
    VisitVertex(pg, visitV); // 시작 정점 방문
    SPush(&stack, visitV); // 방문 후 정점 스택에 삽입
    while(LFirst(&(pg->adjList[visitV]), &nextV) == TRUE) // 연결된 정점 정보 파악
    {
        int visitFlag = FALSE;
        if(VisitVertex(pg, nextV) == TRUE) {
            . . . . . // 방문을 시도했는데 방문에 처음이면
        } else {
            . . . . . // 방문했었다면
        }

        if(visitFlag == FALSE) { // 연결된 정점과의 방문이 모두 완료되었다면,
            if(SIsEmpty(&stack) == TRUE) // 스택이 비면! 종료!
                break;
            else
                visitV = SPop(&stack); // 되돌아 가기 위한 POP 연산!
        }
    }
    memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
}
```

구현: DFShowGraphVertex

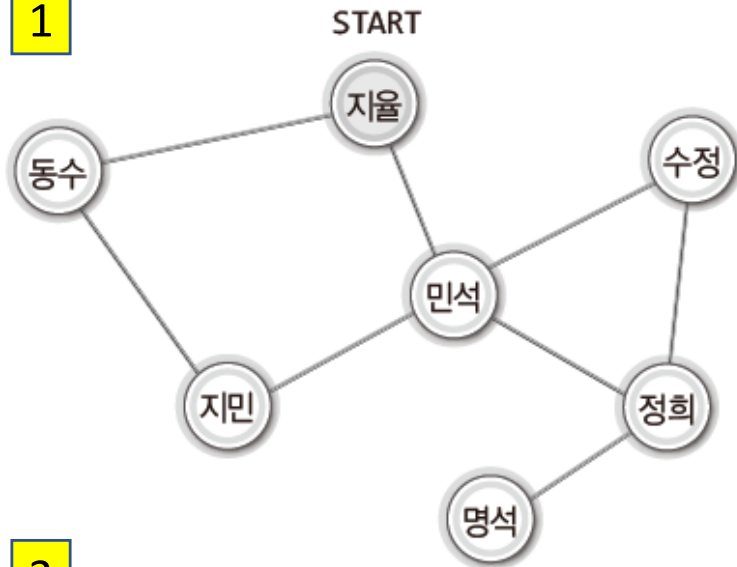
```
while(LFirst(&(pg->adjList[visitV]), &nextV) == TRUE) // 연결된 정점 정보 파악
{
    int visitFlag = FALSE;
    if(VisitVertex(pg, nextV) == TRUE) { // 방문을 시도했는데 방문에 처음이면
        SPush(&stack, visitV);           // 현재 정점에서 이동하므로 스택에 저장!
        visitV = nextV;
        visitFlag = TRUE;
    } else {                             // 방문했었다면
        while(LNext(&(pg->adjList[visitV]), &nextV) == TRUE)
        {
            // 연결된 다른 정점을 찾아서 방문을 시도하는 일련의 과정!
            if(VisitVertex(pg, nextV) == TRUE)
            {
                SPush(&stack, visitV);
                visitV = nextV;
                visitFlag = TRUE;
                break;
            }
        }
    }
}
```

탐색

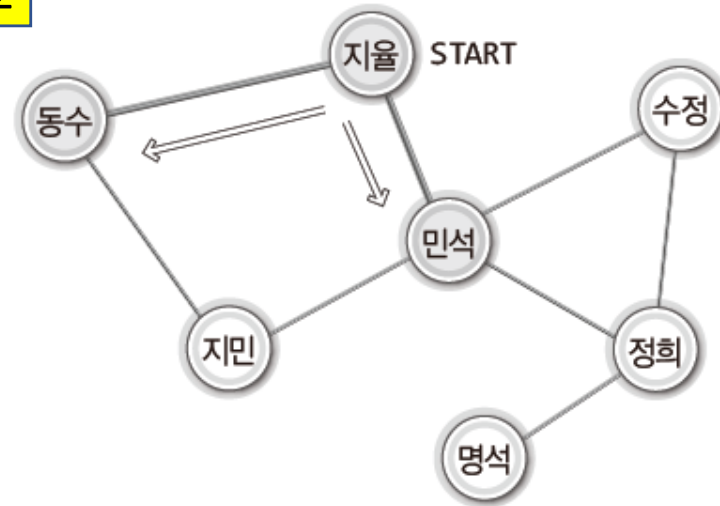
BFS

BFS: 너비 우선 탐색

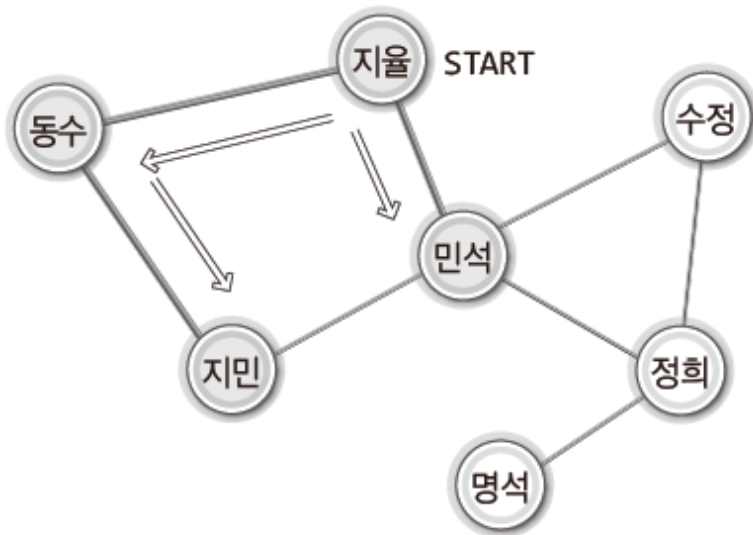
1



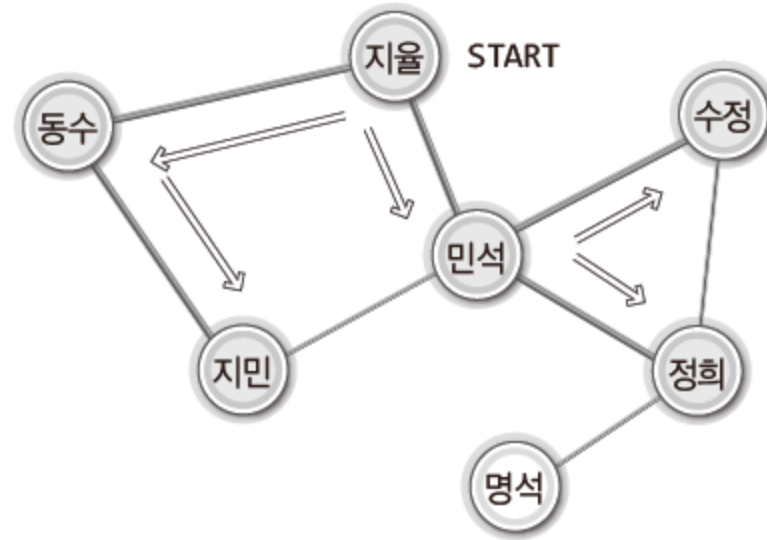
2



3



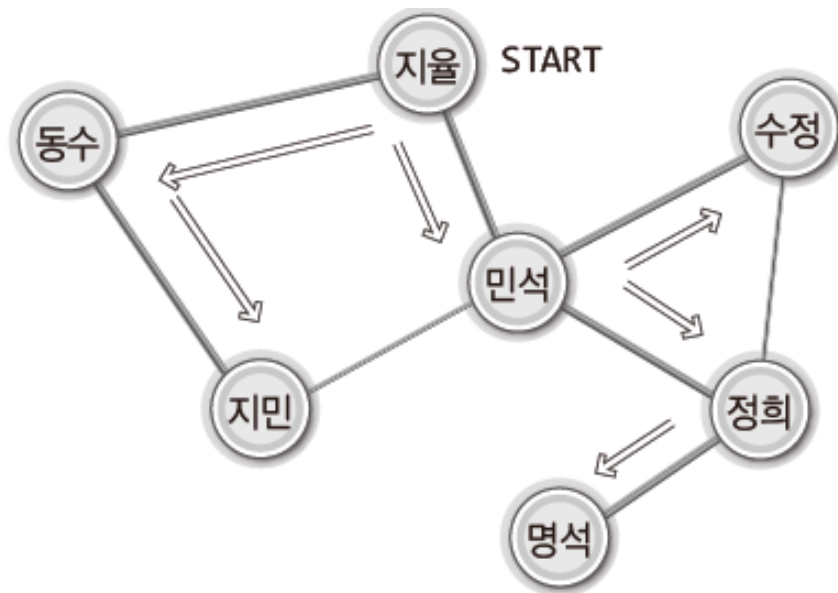
4



BFS: 너비 우선 탐색

- 노드에 연결된 한 노드를 탐색
- 노드에 연결된 다른 노드가 있다면 그 노드를 탐색
- 방문했던 자식 노드에서 반복

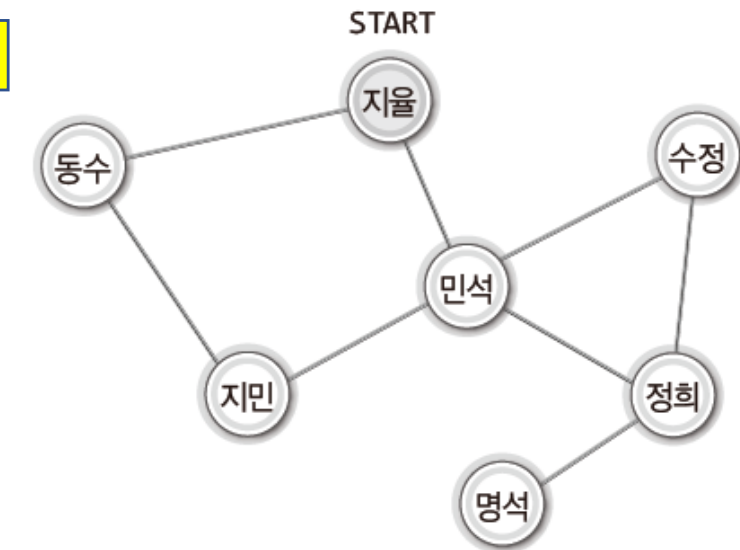
5



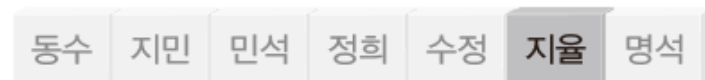
BFS 모델 1

- 큐: 방문 차례 기록
- 배열: 방문정보 기록

1



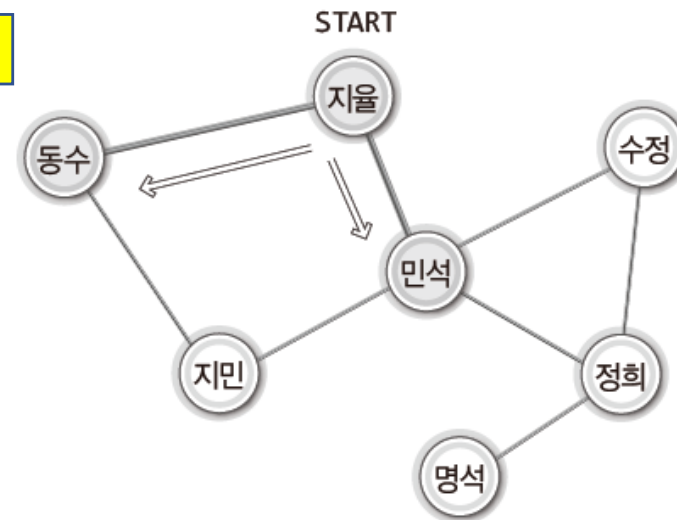
방문 정보



Queue



2



방문 정보



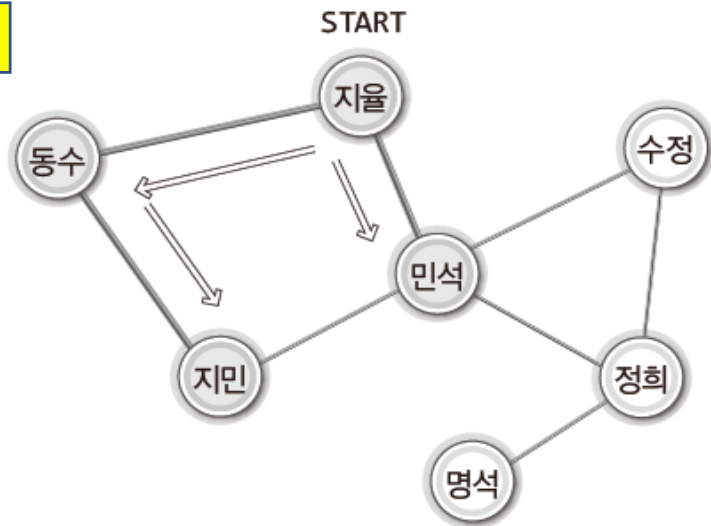
Queue



동수, 민석 enqueue

BFS 모델 2

3



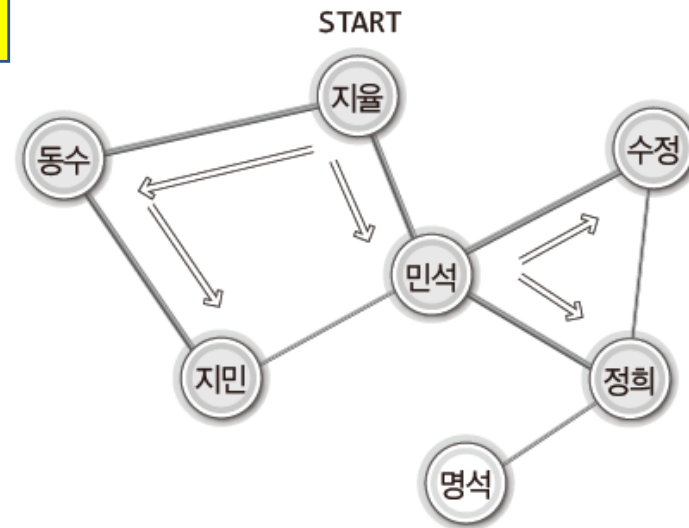
방문 정보



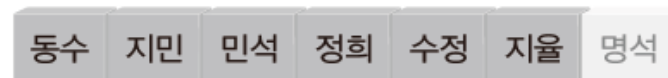
Queue



4



방문 정보

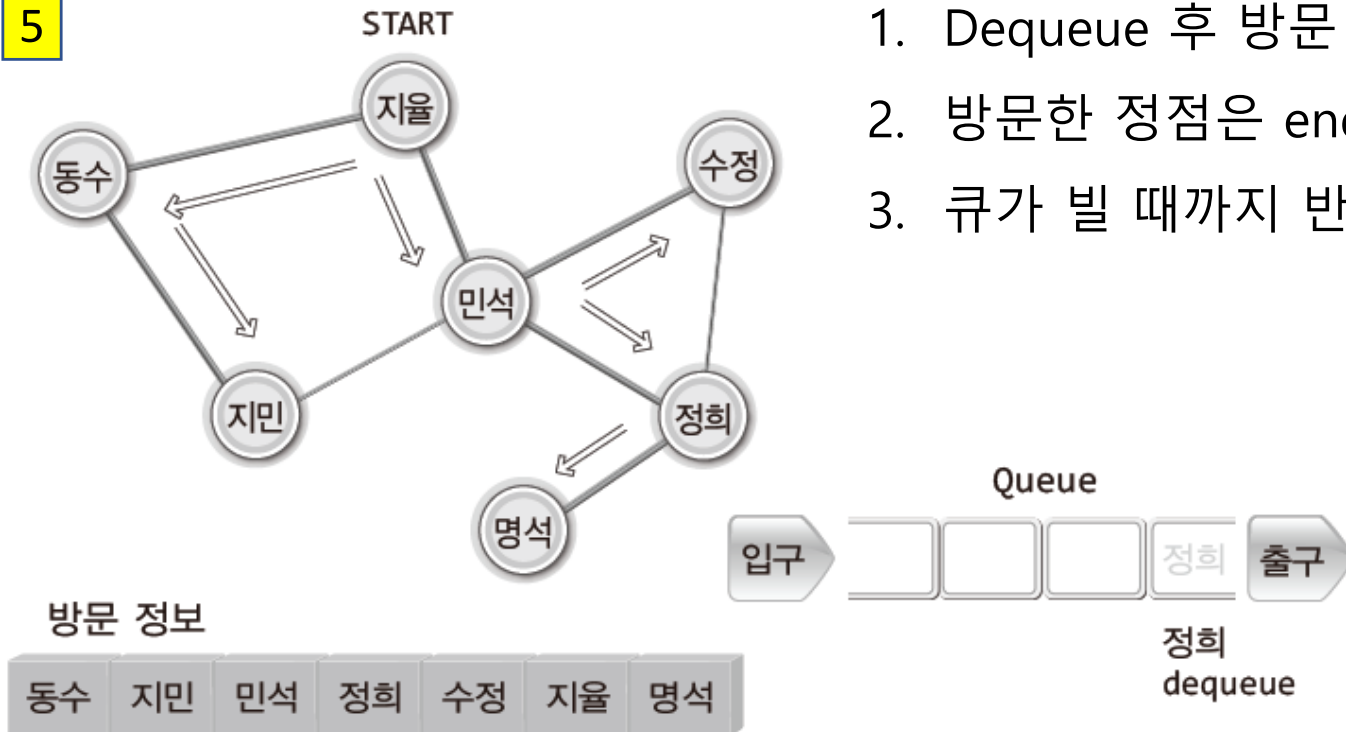


Queue



BFS 모델 3

5



1. Dequeue 후 방문
2. 방문한 정점은 enqueue
3. 큐가 빌 때까지 반복

명석의 정보도 큐에 enqueue, dequeue 됨
예시에서는 명석의 정보가 마지막에 enqueue
Dequeue 되면 종료

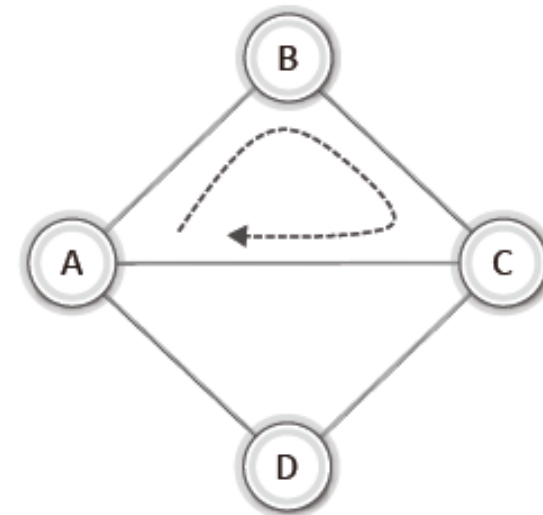
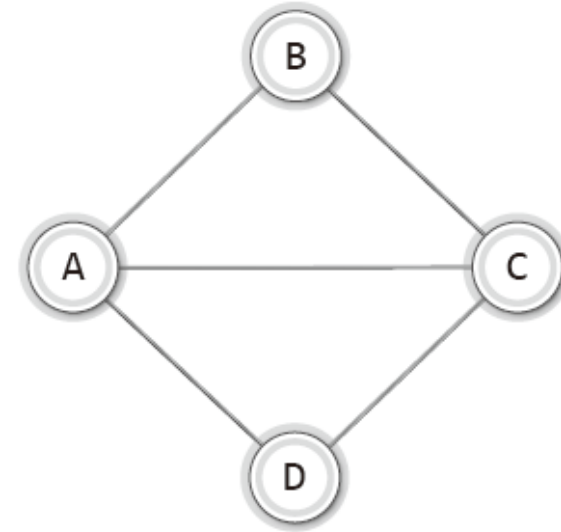
구현: Helper Function (헤더는 DFS와 동일)

```
// Breadth First Search: 정점의 정보 출력
void BFShowGraphVertex(ALGraph * pg, int startV)
{
    Queue queue;      int visitV = startV;      int nextV;
    QueueInit(&queue); // DFS를 위한 큐의 초기화
    VisitVertex(pg, visitV); // 시작 정점 탐색
                                // visitV에 연결된 정점 정보 얻음
    while(LFirst(&(pg->adjList[visitV]), &nextV) == TRUE)
    {
        if(VisitVertex(pg, nextV) == TRUE)
            Enqueue(&queue, nextV);
                                // 계속해서 visitV에 연결된 정점 정보 얻음
        while(LNext(&(pg->adjList[visitV]), &nextV) == TRUE){
            if(VisitVertex(pg, nextV) == TRUE)
                Enqueue(&queue, nextV);
        }
        if(QIsEmpty(&queue) == TRUE) // 큐가 비면 BFS 종료
            break;
        else
            visitV = Dequeue(&queue);
    }
    memset(pg->visitInfo, 0, sizeof(int) * pg->numV); // 탐색 정보 초기화
}
```

최소 비용 신장 트리

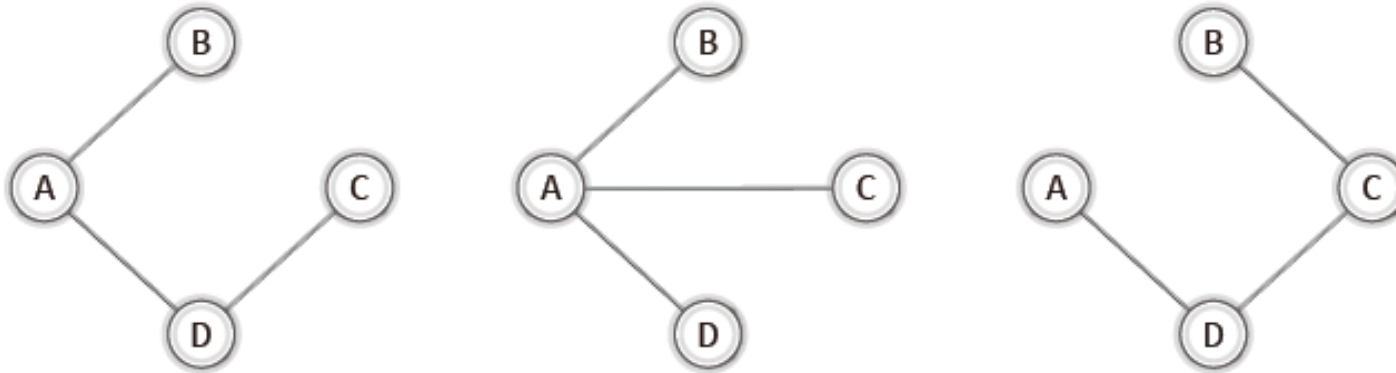
Cycle: 순환 고리

- 정점 B에서 정점 D에 이르는 단순 경로
 - B-A-D
 - B-C-D
 - B-A-C-D
 - B-C-A-D
- 중복의 처리
 - 단순 경로는 중복 간선 불포함
 - 반례: B-A-C-B-A-D (B-A 간선이 2번, 단순경로 X)
- 사이클
 - 시작점과 끝점이 같은 단순 경로

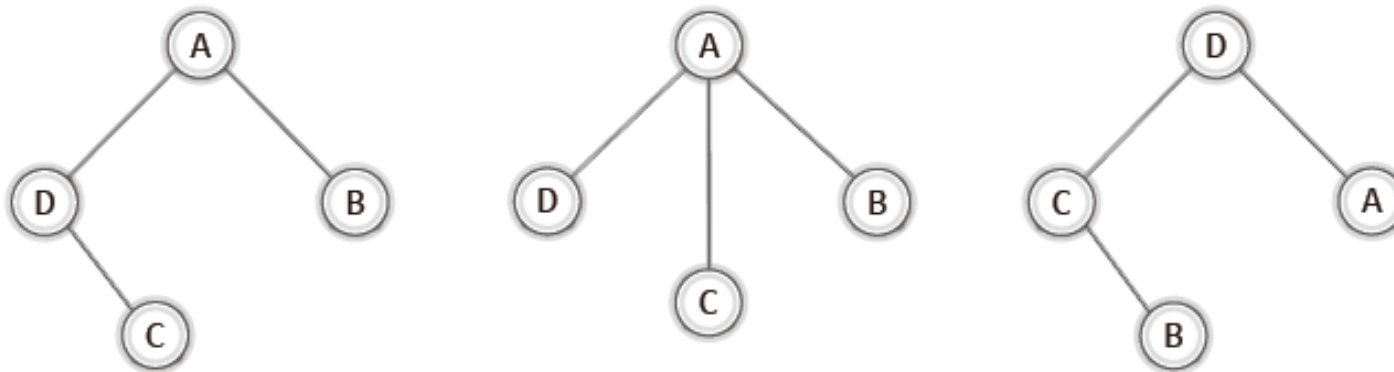


사이클이 없는 그래프

- 신장트리: 경로 중에 '사이클'이 없는 그래프



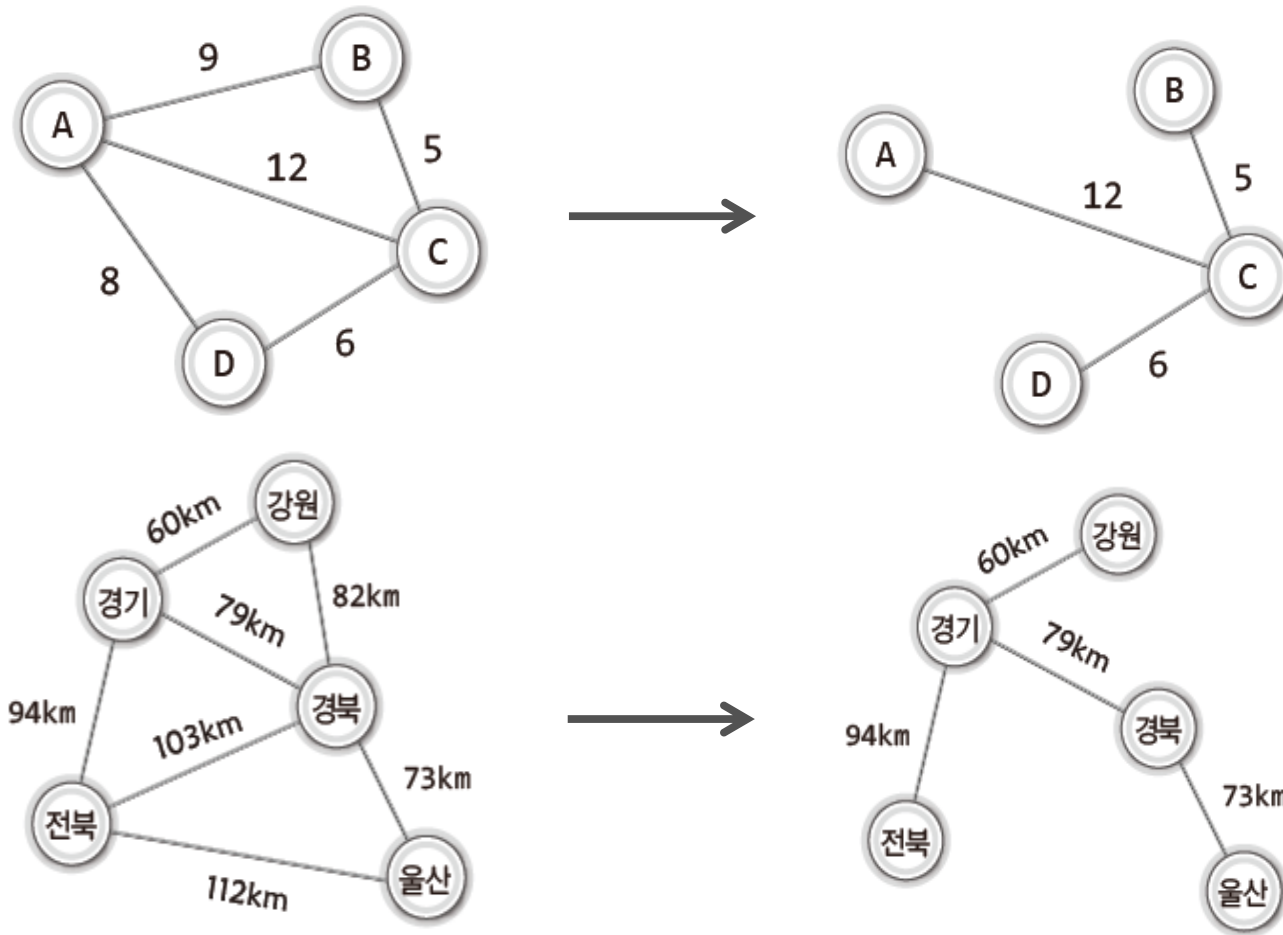
- 신장트리 vs 신장 그래프
 - 90도 회전



최소 비용 신장 트리 (Minimum spanning Tree)

- 특징
 - 그래프의 모든 정점이 간선으로 연결
 - 사이클이 없음

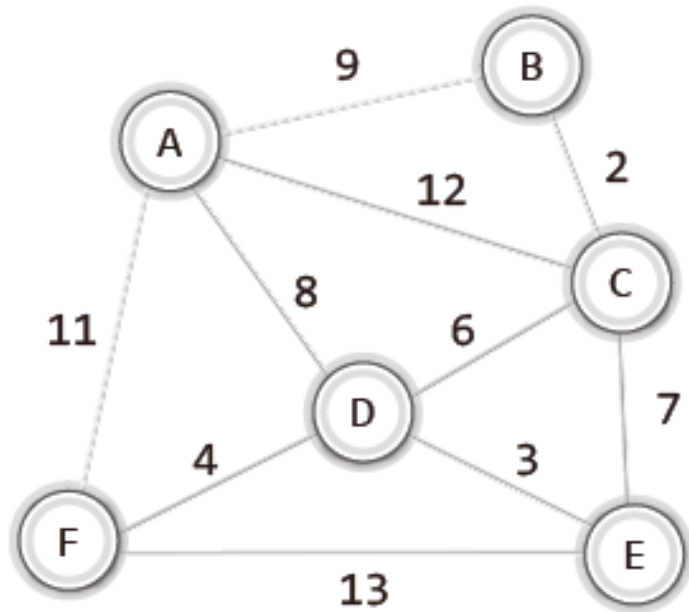
• 예



Kruskal 알고리즘 (낮음->높음) 1

1. 간선 가중치로 정렬
2. MST가 되도록 간선 추가 (사이클은 배제)

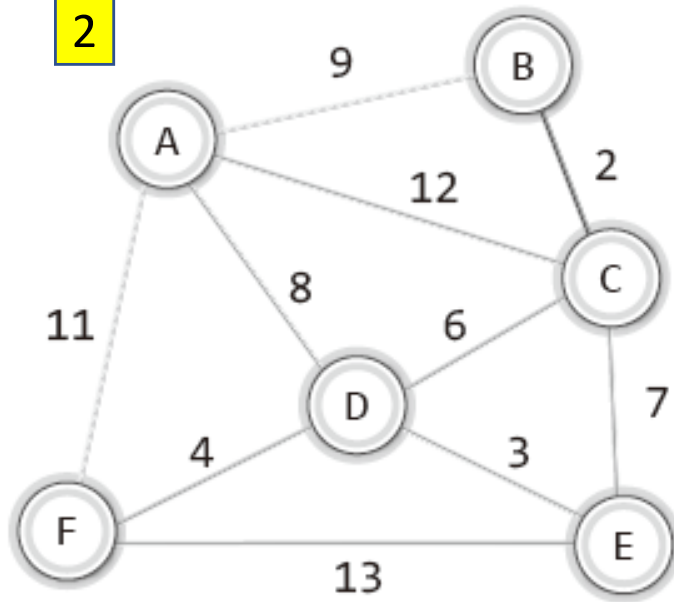
1



2, 3, 4, 6, 7, 8, 9, 11, 12, 13

가중치의 오름차순 정렬

2



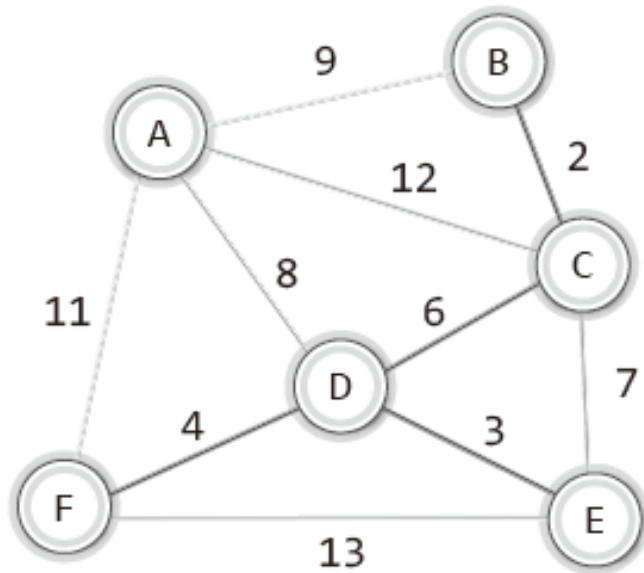
2, 3, 4, 6, 7, 8, 9, 11, 12, 13

가중치의 오름차순 정렬

Kruskal 알고리즘 (낮음->높음) 2

- 종료 조건: 간선의 수 + 1 = 정점의 수

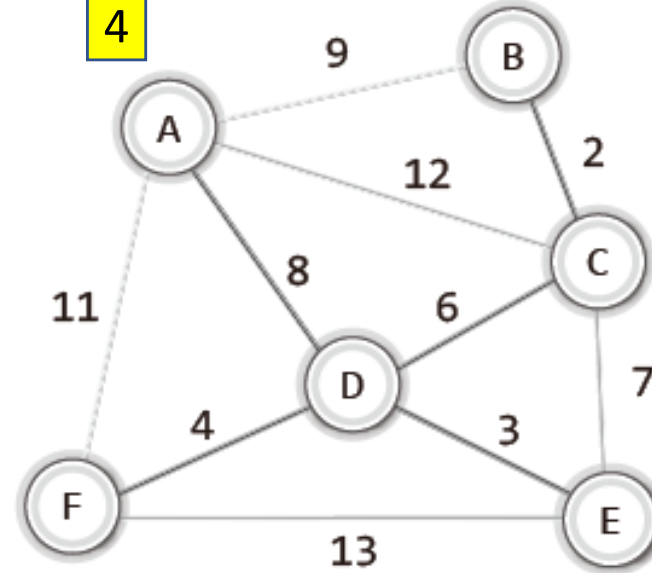
3



2, 3, 4, 6, 7, 8, 9, 11, 12, 13

가중치의 오름차순 정렬

4



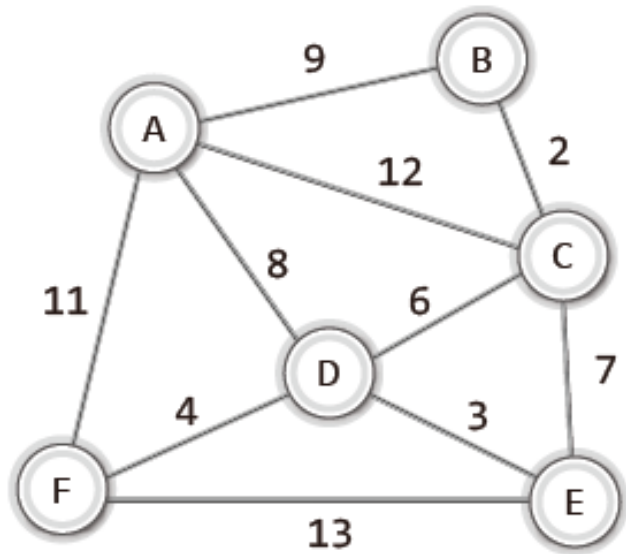
2, 3, 4, 6, 7, 8, 9, 11, 12, 13

가중치의 오름차순 정렬

Kruskal 알고리즘 (낮음->높음) 1

- 높은 가중치 순으로 정렬
- MST가 되도록 (섬을 만들지 않는) 간선 삭제

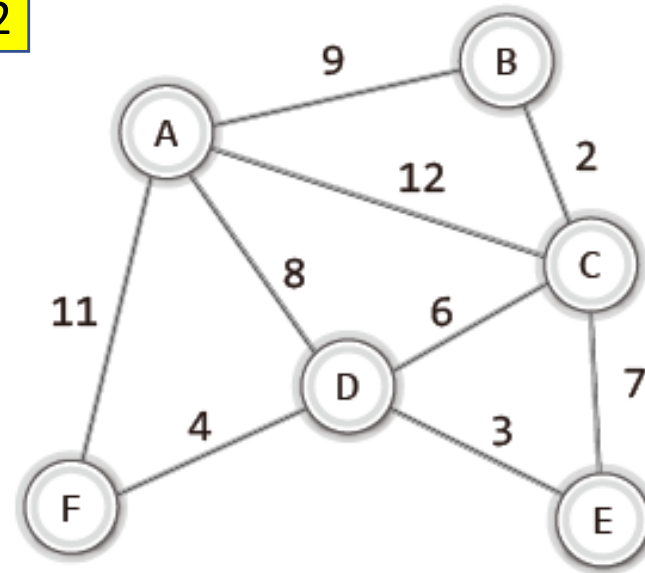
1



13, 12, 11, 9, 8, 7, 6, 4, 3, 2

가중치의 내림차순 정렬

2



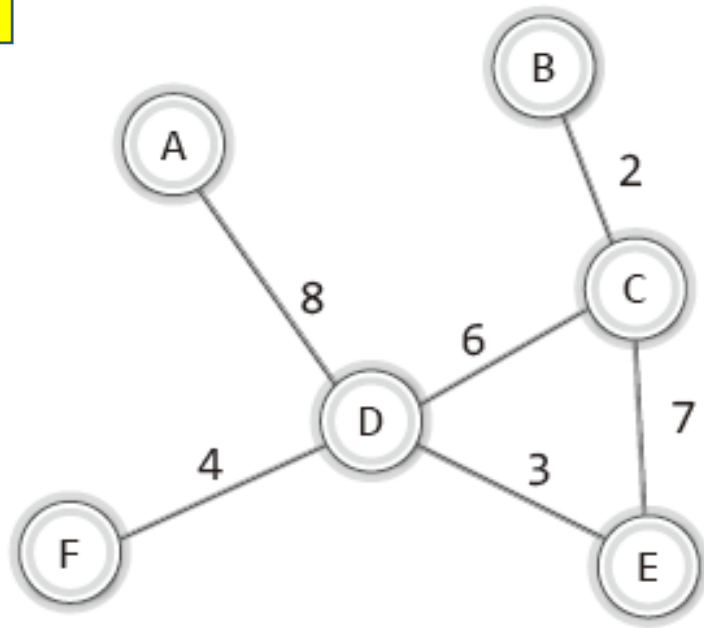
13, 12, 11, 9, 8, 7, 6, 4, 3, 2

가중치의 내림차순 정렬

Kruskal 알고리즘 (낮음->높음) 2

- 종료 조건: 간선의 수 + 1 = 정점의 수

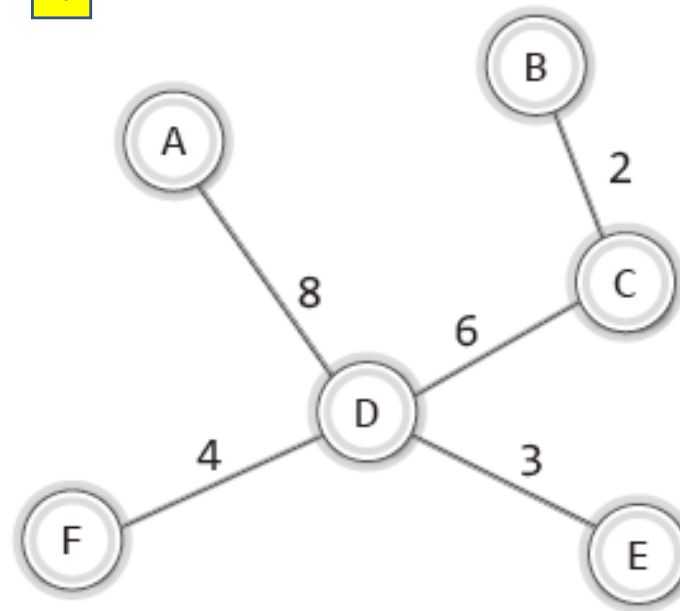
3



이동
→
↓ ↓ ↓
13, 12, 11, 9, 8, 7, 6, 4, 3, 2

가중치의 내림차순 정렬


4



↓
13, 12, 11, 9, 8, 7, 6, 4, 3, 2

가중치의 내림차순 정렬

구현

- 내림 차순 정렬, 간선 삭제 방식
 - 가중치가 포함된 간선을 정의한 구조체 추가 (ALEdge.h)
 - 코드
 - DLinkedList.h, DLinkedList.c
 - ArrayBaseStack.h, ArrayBaseStack.c
 - ALGraphDFS.h, ALGraphDFS.c. 크루스칼 알고리즘의 핵심 코드
-  수정!
- ALGraphKruskal.h, ALGraphKruskal.c 가중치 그래프의 구현 결과
 - AND ???
 - 가중치로 간선을 정렬 (Priority queue, Heap)
 - 삭제 후에도 두 정점을 연결하는 경로 존재 여부

Recap: Queue Vs. Priority Queue/Heap

- Queue: 들어간 순서 기반 Dequeue 연산

3, 31, 2, 0, 20, 4 → 3, 31, 2, 0, 20, 4

- Priority Queue: 우선순위에 따라 Dequeue 연산

3, 31, 2, 0, 20, 4 → 31, 20, 4, 3, 2, 0

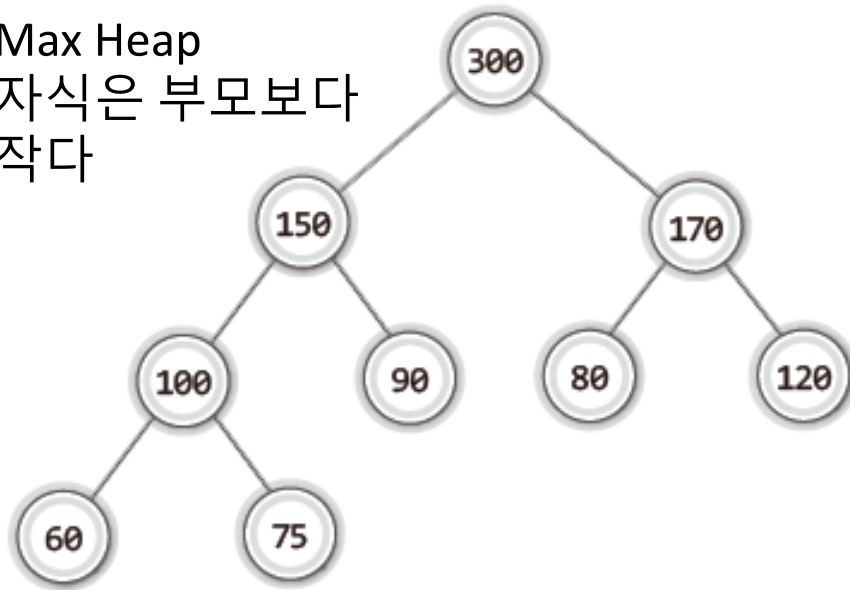
- 구현 방법

- 배열: 최악의 경우 삽입 위치 판단을 위해 모든 데이터 비교
- 연결리스트: 상동
- Heap (힙): 삽입 기준이 있음

Recap: Heap

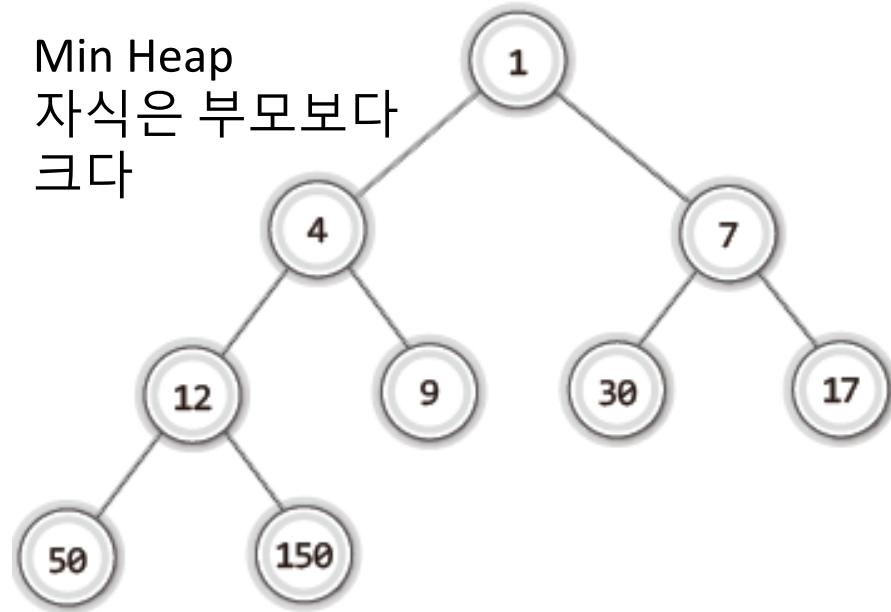
Max Heap

자식은 부모보다 작다



Min Heap

자식은 부모보다 크다



단, 완전 이진 트리이어야 함

리프노드외는 비어 있는 노드가 없어야 하고
왼쪽에서 오른쪽으로 채워져야 함

삽입 방법 ($O(\log n)$):

1. 끝에 삽입
2. 부모와 비교
3. 필요하면 자리 바꿈
4. 2번부터 반복

삽입 방법 ($O(\log n)$):

1. 루트를 제거
2. 끝을 루트로 이동
3. 자식과 비교
4. 필요하면 자리 바꿈
5. 2번부터 반복

구현: 헤더

```
enum {A, B, C, D, E, F, G, H, I, J};

typedef struct _ual
{
    int numV;
    int numE;
    List * adjList;
    int * visitInfo;
    PQueue pqueue;    // 간선의 가중치 정보 저장
} ALGraph;

typedef struct _edge
{
    int v1;    // 첫 번째 정점
    int v2;    // 두 번째 정점
    int weight; // 가중치
} Edge;

void GraphInit(ALGraph * pg, int nv); // 그래프의 초기화
void GraphDestroy(ALGraph * pg); // 그래프의 리소스 해제
void AddEdge(ALGraph * pg, int fromV, int toV, int weight); // 간선의 추가
void ShowGraphEdgeInfo(ALGraph * pg); // 간선의 정보 출력
void DFShowGraphVertex(ALGraph * pg, int startV); // Depth First Search:
정점의 정보 출력
void ConKruskalMST(ALGraph * pg); // 크루스칼 최소 비용 신장 트리의 구성
void ShowGraphEdgeWeightInfo(ALGraph * pg); // 간선의 가중치 정보 출력
```


구현: 수정된 부분

```
void GraphInit(ALGraph * pg, int nv)
{
    int i;
    pg->adjList = (List*)malloc(sizeof(List)*nv);
    pg->numV = nv; pg->numE = 0;

    for(i=0; i<nv; i++) {
        ListInit(&(pg->adjList[i]));
        SetSortRule(&(pg->adjList[i]), WhoIsPrecede);
    }

    pg->visitInfo = (int *)malloc(sizeof(int) * pg->numV);
    memset(pg->visitInfo, 0, sizeof(int) * pg->numV);

    PQueueInit(&(pg->pqueue), PQWeightComp); // 우선순위 큐의 초기화
}

int PQWeightComp(Edge d1, Edge d2)
{
    return d1.weight - d2.weight;
}
```

가중치 기준 내림차순으로

간선 정보 꺼내기 위한 정의!

구현: 수정된 부분

```
void AddEdge(ALGraph * pg, int fromV, int toV, int weight)
{
    Edge edge = {fromV, toV, weight};    // 간선의 정보 생성

    LInsert(&(pg->adjList[fromV]), toV);
    LInsert(&(pg->adjList[toV]), fromV);
    pg->numE += 1;

    PEnqueue(&(pg->pqueue), edge); // 간선의 정보를 우선순위 큐에 저장
}
```

구현: ConKruskalMST

```
void ConKruskalMST(ALGraph * pg)
{
    Edge recvEdge[20];    // 복원할 간선의 정보 저장
    Edge edge;
    int eidx = 0;         // MST 간선의 수 + 1 == 정점의 수
    int i;                // 크루스칼 알고리즘 기반 MST

    while(pg->numE+1 > pg->numV) // MST일 때까지 아래의 while문을 반복
    {
        edge = PDequeue(&(pg->pqueue)); //가중치 순으로 간선 정보 획득!
        RemoveEdge(pg, edge.v1, edge.v2); //획득한 정보의 간선 실제 삭제!

        if(!IsConnVertex(pg, edge.v1, edge.v2)) // 두 정점 연결 경로 확인
        {
            RecoverEdge(pg, edge.v1, edge.v2);
            recvEdge[eidx++] = edge; // 없으면 간선 복원
        }
    }

    for(i=0; i<eidx; i++) // 우선순위 큐에서 삭제된 간선의 정보를 회복
        PEnqueue(&(pg->pqueue), recvEdge[i]);
}
```

구현: 삭제와 복원

// 간선의 소멸: ConKruskalMST Helper function

```
void RemoveEdge(ALGraph * pg, int fromV, int toV)
{
    RemoveWayEdge(pg, fromV, toV); // 무방향 그래프
    RemoveWayEdge(pg, toV, fromV);
    (pg->numE)--;
}
```

// ConKruskalMST Helper function

```
void RecoverEdge(ALGraph * pg, int fromV, int toV)
{
    LInsert(&(pg->adjList[fromV]), toV); // 간선 가중치 저장 안함
    LInsert(&(pg->adjList[toV]), fromV);
    (pg->numE)++;
}
```

구현 : 방향 그래프를 위한 helper function

```
// 한쪽 방향의 간선 소멸: ConKruskalMST Helper function
void RemoveWayEdge(ALGraph * pg, int fromV, int toV)
{
    int edge;

    if(LFirst(&(amp;pg->adjList[fromV]), &edge)){
        if(edge == toV){
            LRemove(&(amp;pg->adjList[fromV]));
            return;
        }

        while(LNext(&(amp;pg->adjList[fromV]), &edge)){
            if(edge == toV){
                LRemove(&(amp;pg->adjList[fromV]));
                return;
            }
        }
    }
}
```

구현: 연결 확인

```
int IsConnVertex(ALGraph * pg, int v1, int v2)
{
    // 두 정점이 연결되어 있다면 TRUE, 그렇지 않다면 FALSE 반환
    Stack stack;      int visitV = v1;      int nextV;
    StackInit(&stack); VisitVertex(pg, visitV); SPush(&stack, visitV);
    while(LFirst(&(pg->adjList[visitV]), &nextV) == TRUE){
        int visitFlag = FALSE;
        if(nextV == v2){ // 순회 중 목표를 찾으면 TRUE
            memset(pg->visitInfo, 0, sizeof(int) * pg->numV); // 초기화
            return TRUE; // 목표를 찾으면 TRUE
        }
        if(VisitVertex(pg, nextV) == TRUE){
            SPush(&stack, visitV);
            visitV = nextV;
            visitFlag = TRUE;
        } else {
            while(LNext(&(pg->adjList[visitV]), &nextV) == TRUE){
                /* 음영의 탐색 작업 반복 */
            }
        }
        if(visitFlag == FALSE){
            if(SIsEmpty(&stack) == TRUE). break;
            else visitV = SPop(&stack);
        }
    }
    memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
    return FALSE; // 목표를 못 찾음
}
```