그래프,DFS,BFS (1)

WEEK 5 백민희

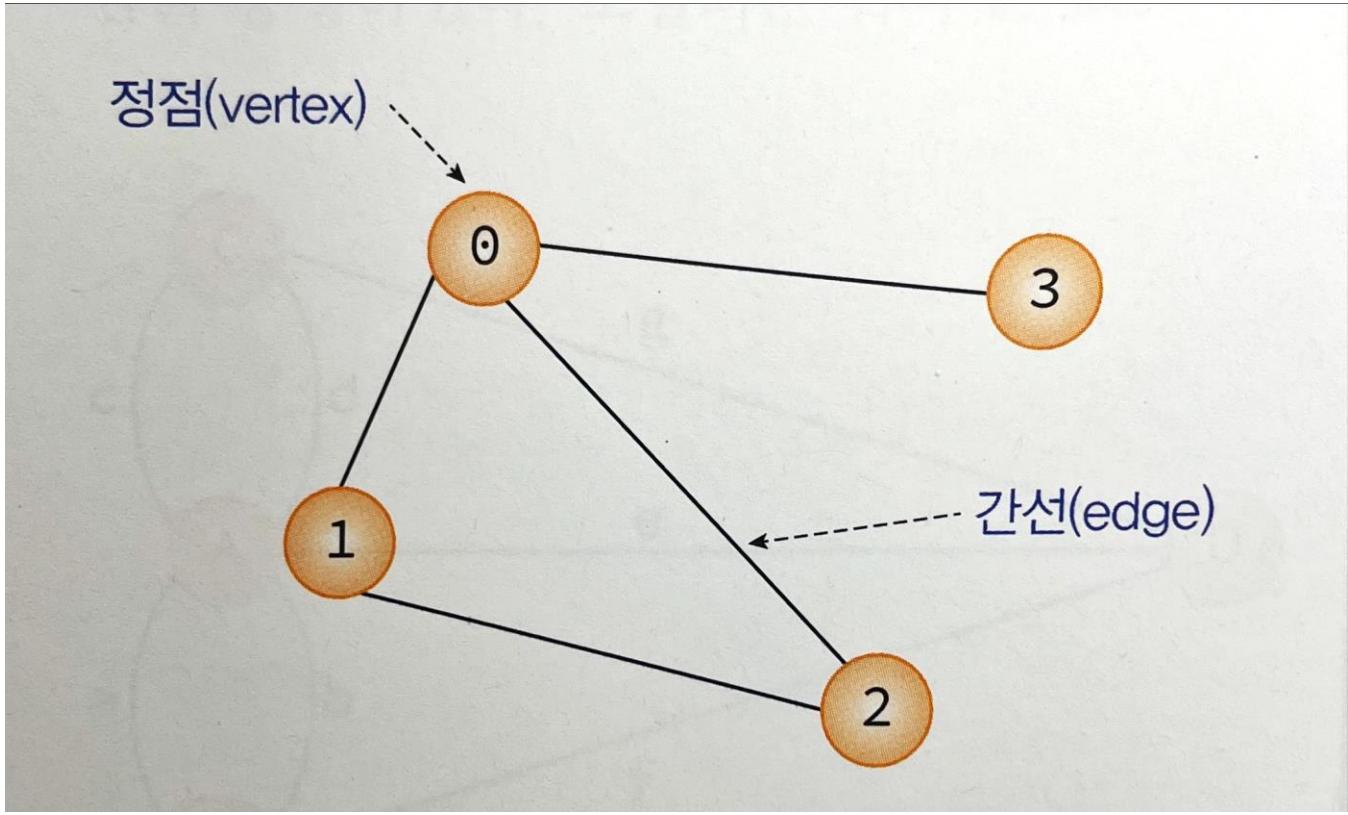
목차

1. 그래프 기본 개념&종류
2. 그래프의 표현 방법(인접행렬/인접리스트)
3. 그래프의 탐색(DFS/BFS)

그래프 기본 개념&종류

그래프 정의

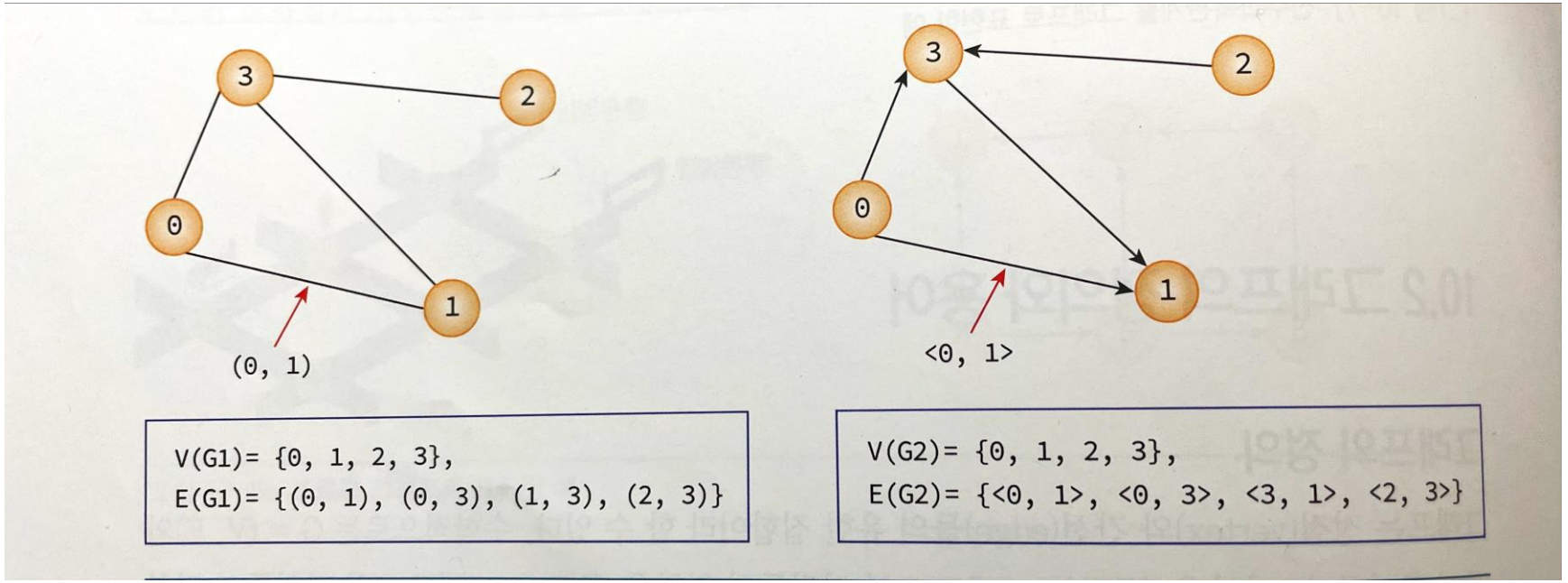
- 그래프:정점과 간선들의 유한 집합
- 수학적으로 G=(V, E)로 표시한다.
- V(G) 그래프 G의 정점들의 집합
- E(G) 그래프 G의 간선들의 집합
- 정점:여러가지 특성을 가질 수 있는 객체를 의미
- 간선:이러한 정점들 간의 관계를 의미함
- 정점=노드, 간선=링크



그래프를 집합으로 표현

- V(G1)={0, 1, 2, 3}
- E(G1)={ (0,1), (0,2), (0,3), (1,2) }

무방향 그래프와 방향 그래프

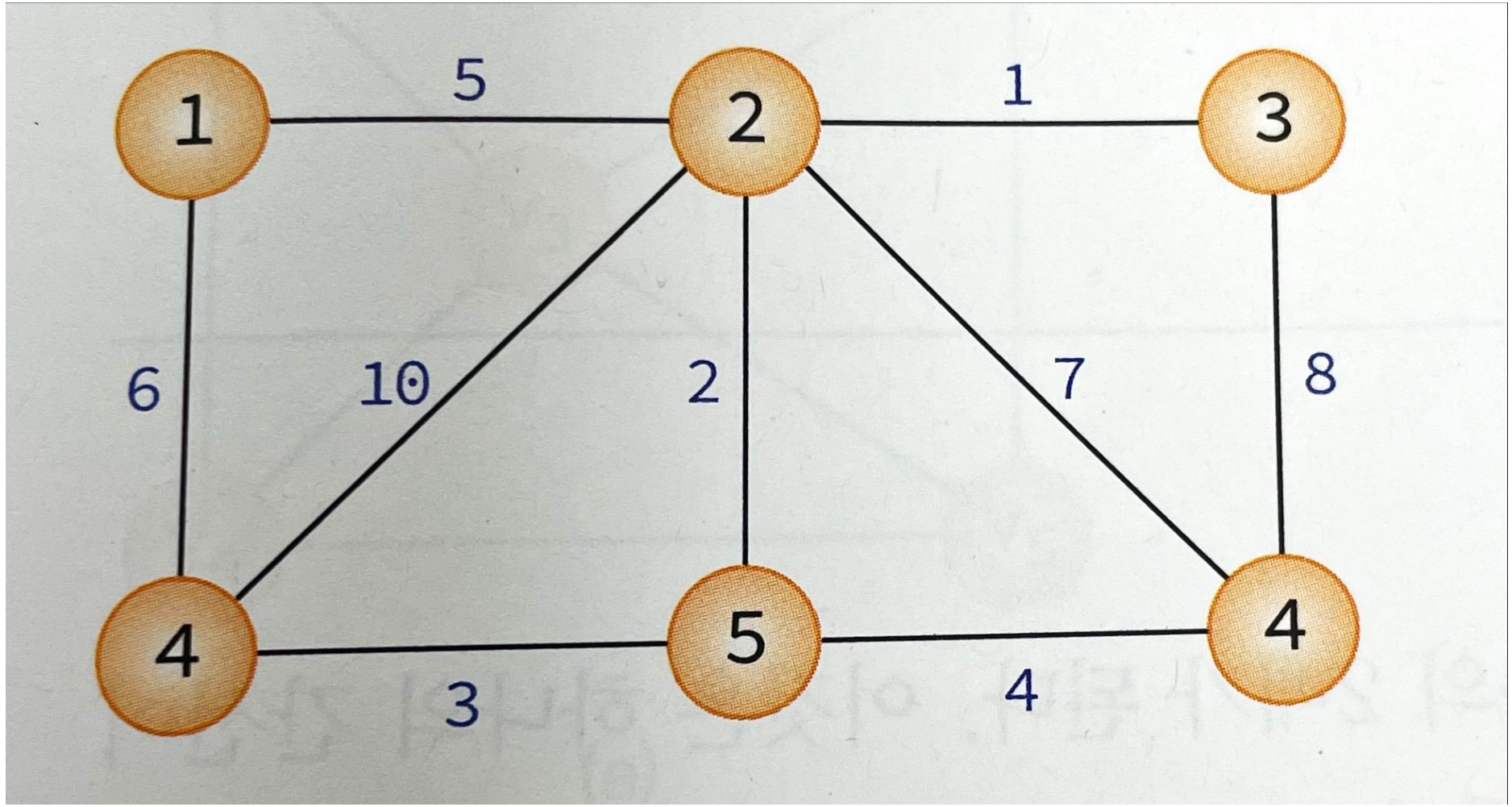


- 무방향 그래프의 간선:간선을 통해서 양방향으로 갈 수 있음

(A,B)로 표시, (A,B)와 (B,A)는 동일한 간선

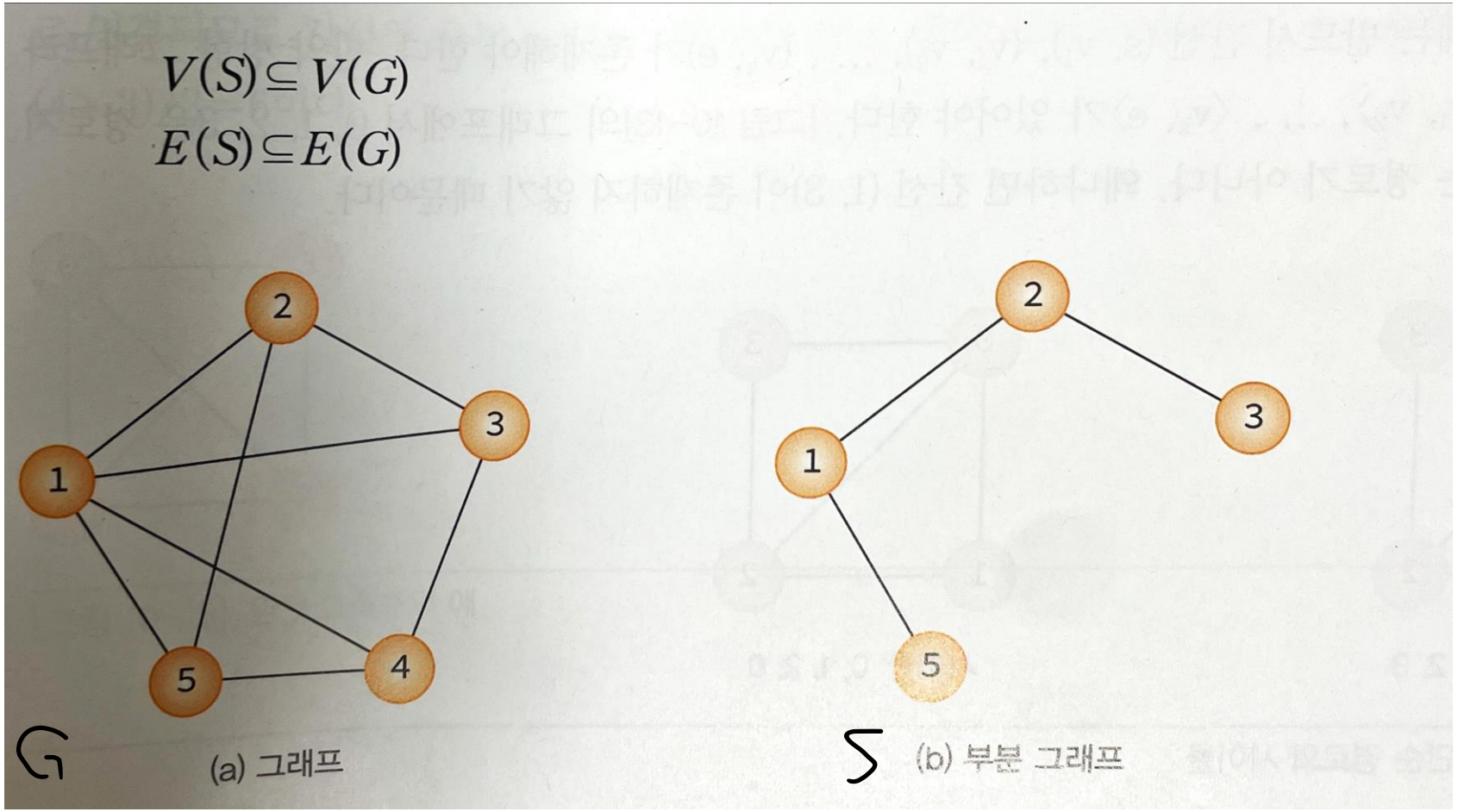
- 양방향 그래프의 간선:간선에 방향성이 존재하는 그래프로서 간선을 통해 한쪽 방향으로만 갈 수 있다.
- 정점 A에서 정점 B로만 갈 수 있는 간선은 <A,B>로 표현
- <A,B>, <B,A>는 서로 다른 간선

네트워크



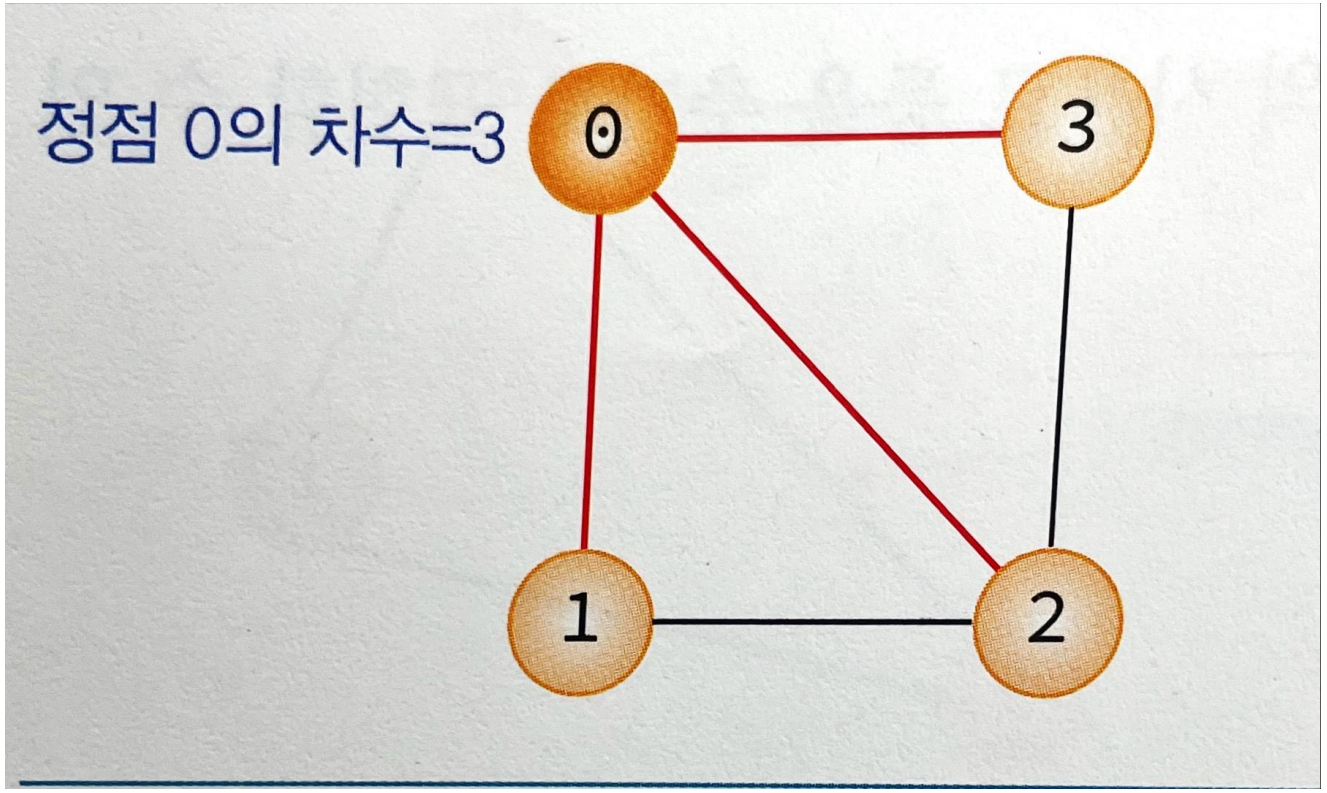
- 간선에 가중치를 할당하면 간선의 역할이 두 정점간의 연결 유무뿐만 아니라 연결 강도까지 나타낼 수 있으므로 복잡한 관계를 표현할 수 있게 된다.

부분그래프



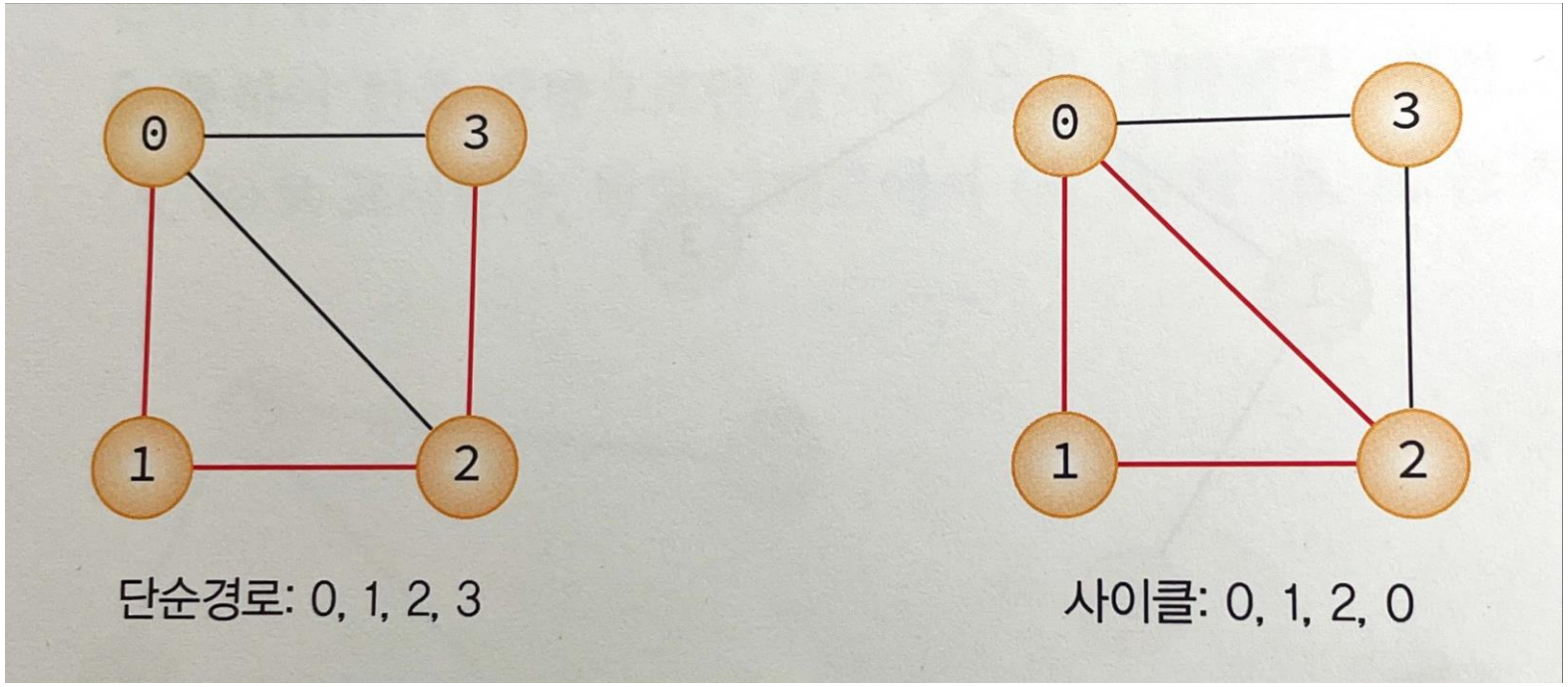
- 어떤 그래프의 정점의 일부와 간선의 일부로 이루어진 그래프

정점의 차수



- 인접 정점:간선에 의해 직접 연결된 정점
- 정점 0의 인접 정점은 정점1, 정점2, 정점 3
- 무방향 그래프에서 정점의 차수는 그 정점에 인접한 정점의 수를 말함

경로

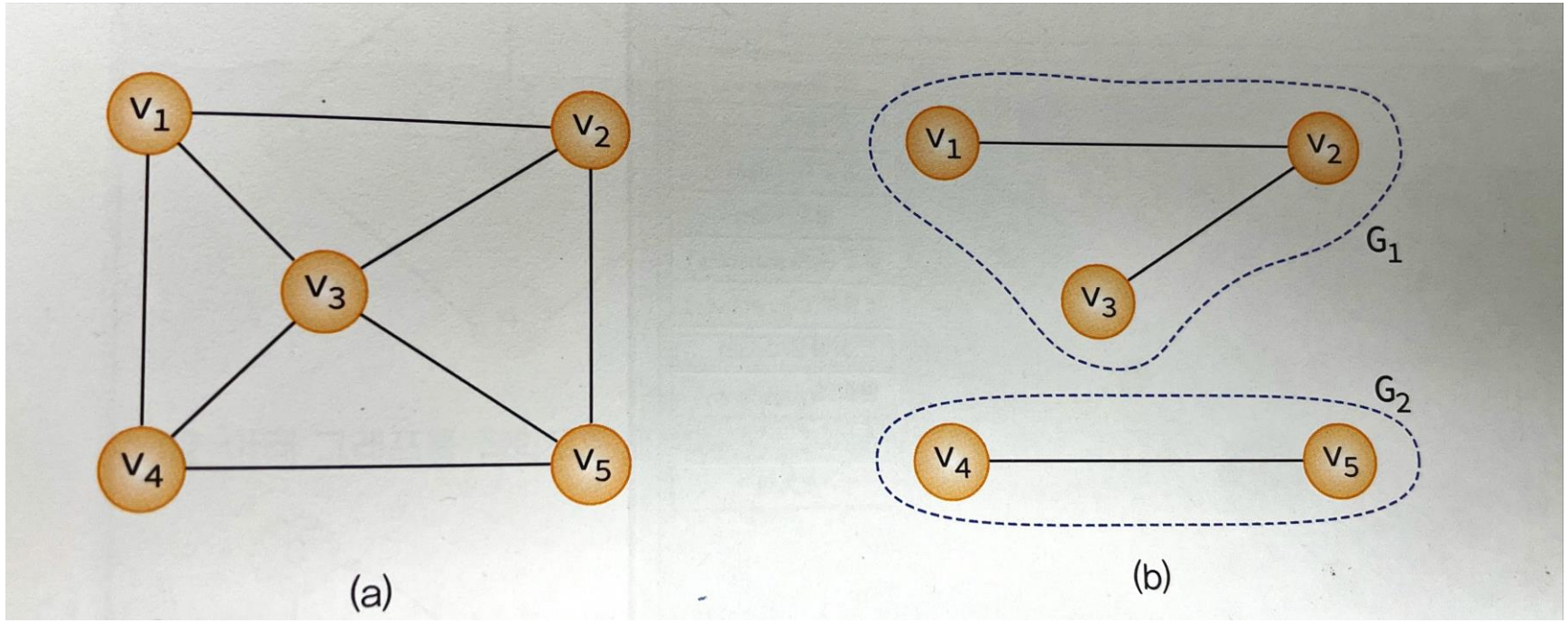


경로 중에서 반복되는 간선이 없을 경우 단순 경로,
단순 경로의 시작 정점과 종료 정점이 동일하면 사이클

- 무방향 그래프에서 정점 s로부터 정점 e까지의 경로는 정점의 나열 s,v1,v2, … vk,e로서, 나열된 정점들 간에는 반드시 간선 (s,v1), (v1,v2), …, (vk,e)가 존재해야한다. 방향그래프도 마찬가지

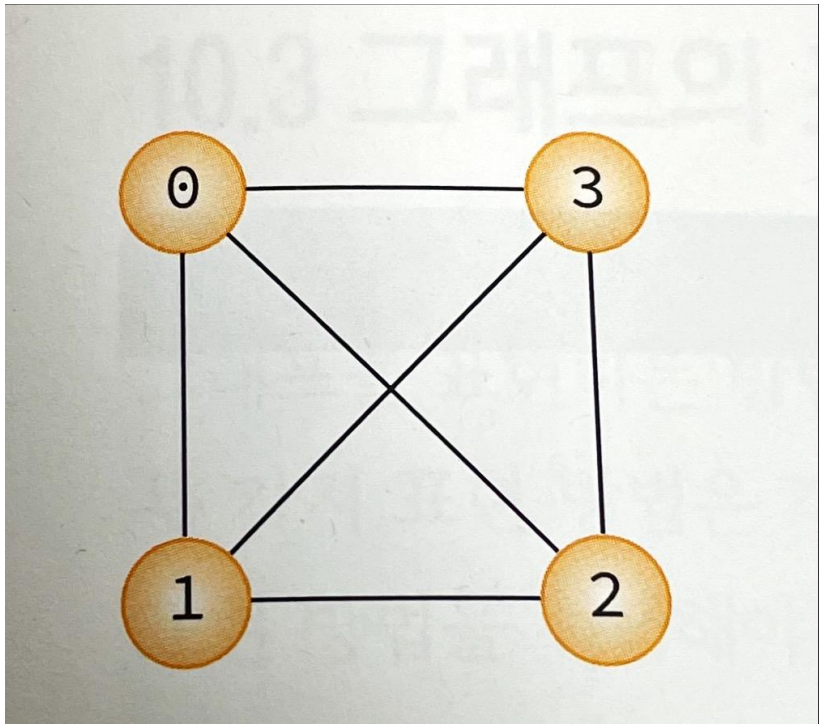
- 그래프에서 0,1,2,3 은 경로지만 0,1,3,2는 존재하지 않음
- 간선 (1,3)이 존재하지 않기 때문

연결그래프



- 무방향 그래프 G에 있는 모든 정점쌍에서 항상 경로가 존재한다면 G는 연결그래프, 그렇지 않으면 비연결그래프
- 트리는 사이클을 가지지 않은 연결그래프

완전그래프



- 완전그래프

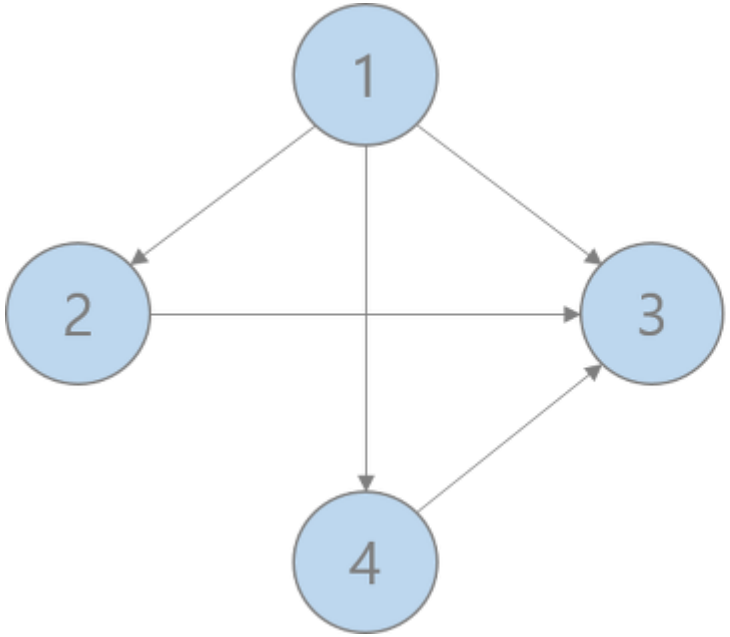
:그래프에 속해있는 모든 정점이 서로 연결되어있는 그래프

무방향 완전 그래프의 정점 수를 n 이라고 하면
간선의 수는 n*(n-1)/2

그래프의 표현 방법-인접행렬

- 인접행렬을 adj[][]라고 한다면 adj[i][j]에 대해 다음과 같이 정의
- adj[i][j] : 노드 i에서 노드 j로 가는 간선이 있으면 1, 아니면 0

간선에 방향이 있는 유향그래프의 연결관계를 인접행렬로 나타냄

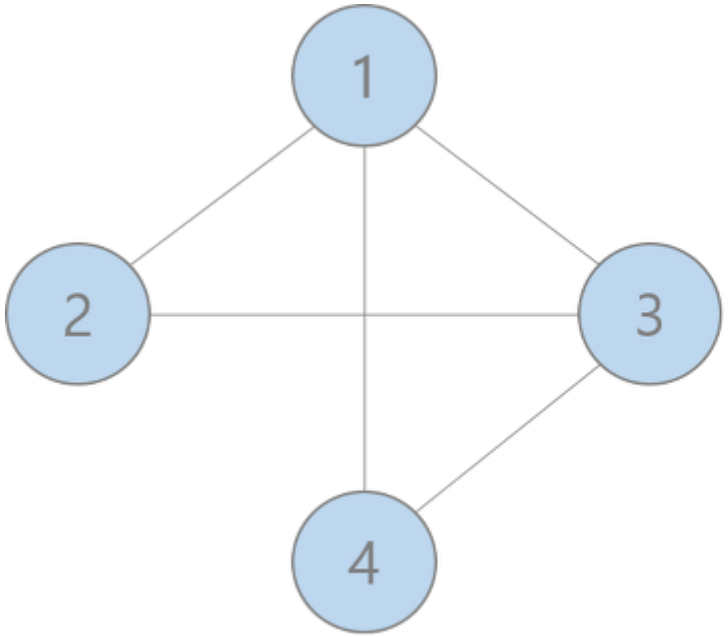


	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

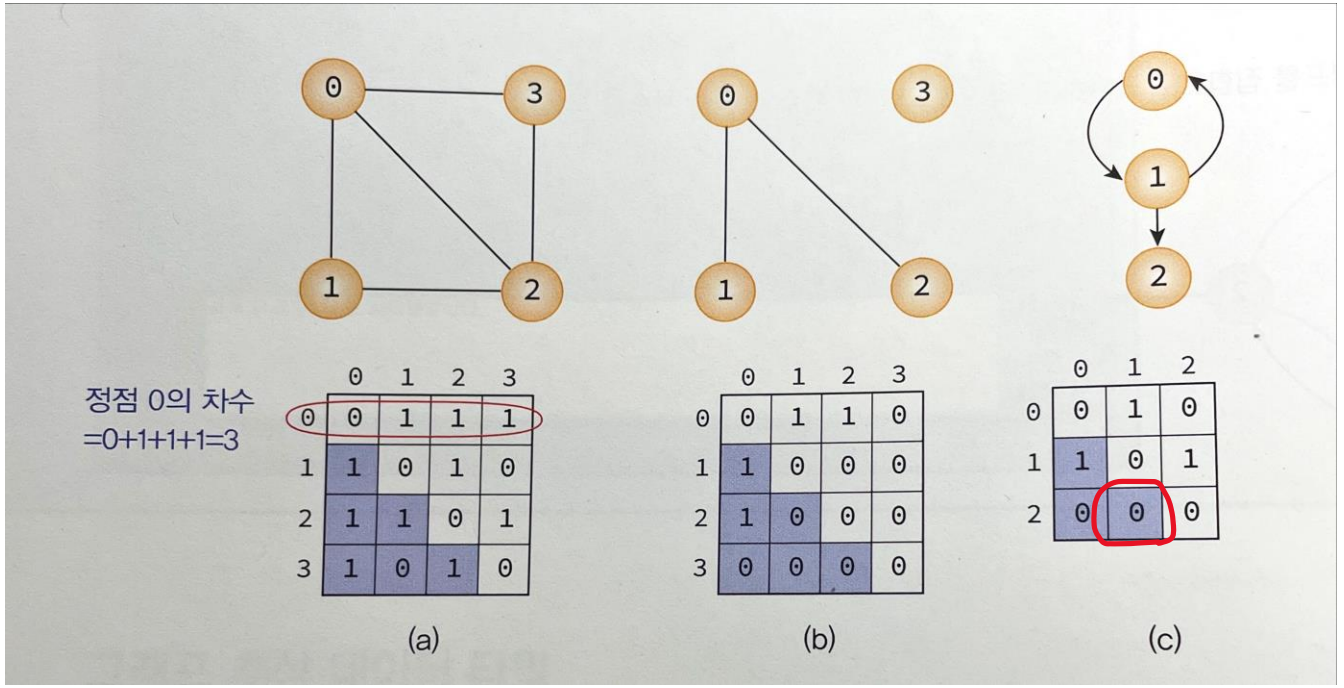
그래프의 표현 방법-인접행렬

- 인접행렬을 adj[][]라고 한다면 adj[i][j]에 대해 다음과 같이 정의
- adj[i][j] : 노드 i에서 노드 j로 가는 간선이 있으면 1, 아니면 0

- 간선에 방향이 없는 무향그래프의 연결관계
- **대각 성분**(adj[i][j]에서 i와 j가 같은 원소들)**을 기준으로 대칭**인 성질을 갖게 됨(자체간선을 허용하지 않는 그래프 기준)



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0



- 무방향 그래프:배열의 상위 삼각이나 하위 삼각만 저장하면 메모리 절약할 수 있다
- (c)처럼 방향 그래프의 인접행렬은 일반적으로 대칭이 아님
- n개의 정점을 가지는 그래프를 인접행렬로 표현하려면 간선수와 무관하게 n^2개의 메모리 공간이 필요하다.
- 인접행렬은 a처럼 그래프에 간선이 많은 밀집 그래프를 표현하는데 적합, b처럼 그래프 내에 적은 숫자의 간선만을 가지는 희소그래프 경우 메모리 낭비가 커서 적합하지 않음

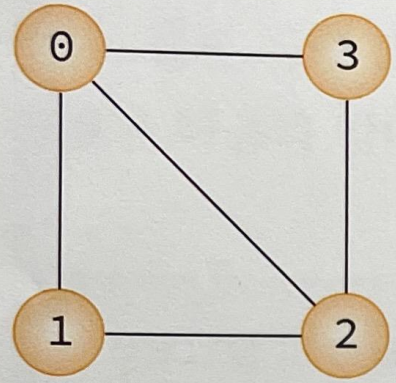
인접행렬의 장점&단점

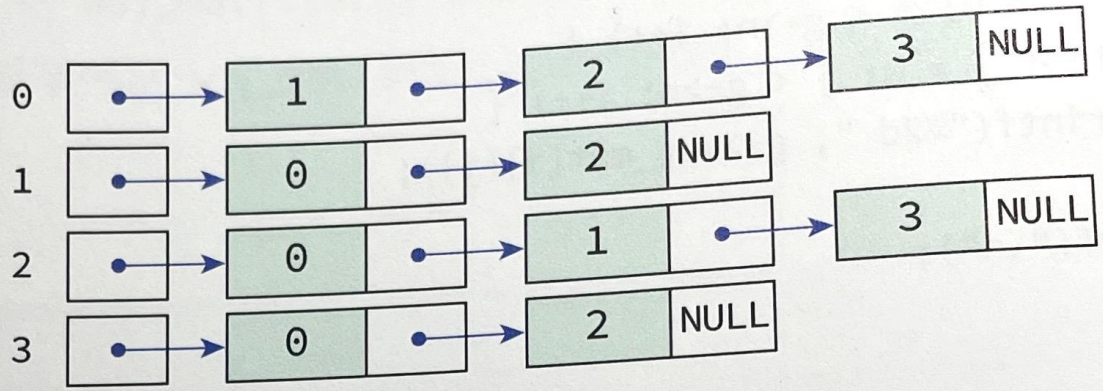
- 두 정점을 연결하는 간선의 존재 여부를 O(1)시간 안에 즉시 알 수 있다.
- 정점 U와 V를 연결하는 정점이 있는지 알려면 M[u][v]의 값을 조사하면 됨
- 정점의 차수는 O(N)의 연산에 의해 알 수 있다.

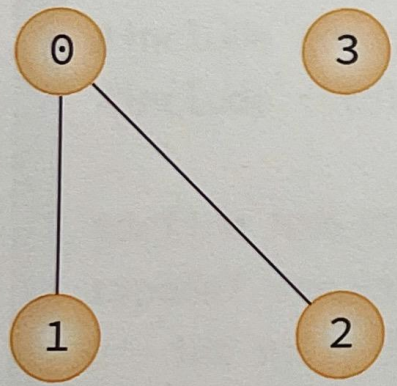
그래프의 표현 방법-인접리스트

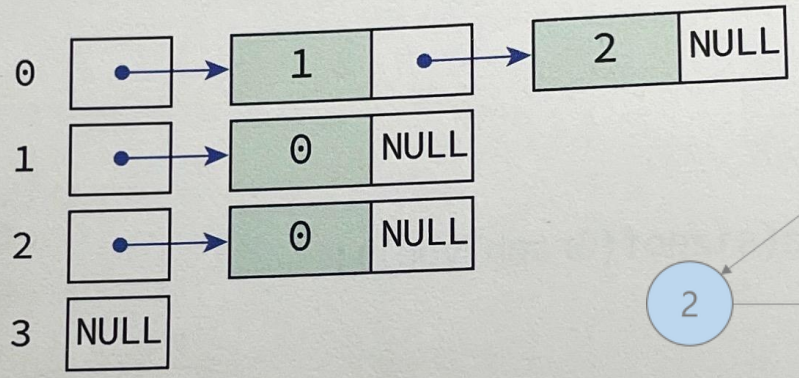
- 인접리스트:각각의 정점에 인접한 정점들을 연결리스트로 표시한 것
- 각 연결리스트의 노드들은 인접 정점을 저장
- 각 연결리스트들은 헤더 노드를 가지고 있고 헤더노드들은 하나의 배열로 구성됨
- 정점의 번호만 알면 이 번호의 배열의 인덱스로 하여 각 정점의 연결리스트에 쉽게 접근 가능

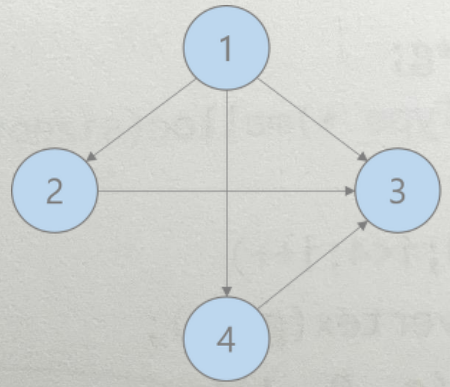
- 무방향 그래프 경우 간선(i, j)는 정점 i의 연결리스트에 인접 정점 j로서 표현되고 정점 j의 연결리스트에 인접 정점 i로 다시 표현됨
- 연결리스트 내에서 정점들의 순서 상관 없다.

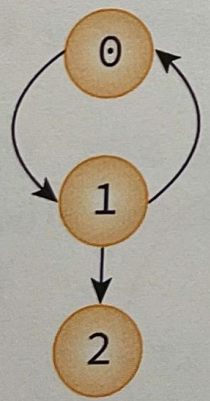

0
3
1
2

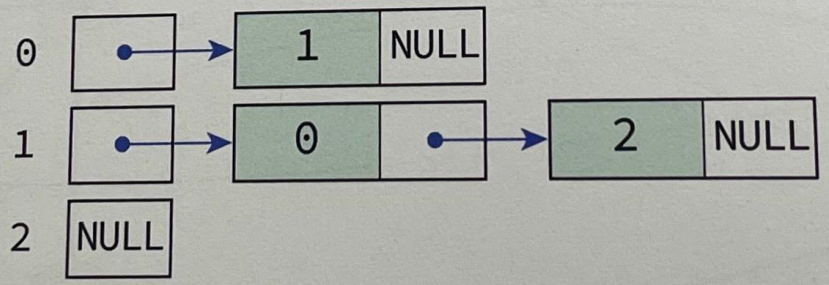

0
1
2
3
NULL

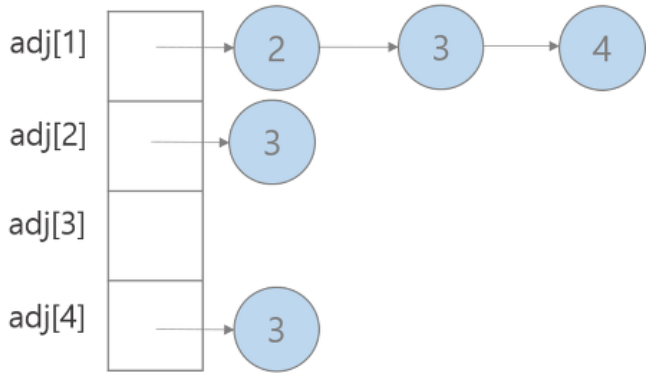

0
3
1
2


0
1
2
3
NULL


1
2
3
4


0
1
2


0
1
2
NULL


adj[1]
adj[2]
adj[3]
adj[4]
2
3
4
3
3

그래프의 탐색-DFS(인접행렬)

```java
import java.util.*;

public class DFS_Array {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt(); // 정점의 개수
        int m = sc.nextInt(); // 간선의 개수
        int v = sc.nextInt(); // 탐색을 시작할 정점의 번호

        boolean visited[] = new boolean[n + 1]; // 방문 여부를 검사할 배열

        int[][] adjArray = new int[n+1][n+1];

        // 두 정점 사이에 여러 개의 간선이 있을 수 있다.
        // 입력으로 주어지는 간선은 양방향이다.
        for(int i = 0; i < m; i++) {
            int v1 = sc.nextInt();
            int v2 = sc.nextInt();

            adjArray[v1][v2] = 1;
            adjArray[v2][v1] = 1;
        }

        System.out.println("DFS - 인접행렬 / 재귀로 구현");
        dfs_array_recursion(v, adjArray, visited);
        Arrays.fill(visited, false); // 스택 DFS를 위해 visited 배열 초기화

```

```java
    //DFS - 인접행렬 / 재귀로 구현
    public static void dfs_array_recursion(int v, int[][] adjArray, boolean[] visited) {
        int l = adjArray.length-1;
        visited[v] = true;
        System.out.print(v + " ");

        for(int i = 1; i <= l; i++) {
            if(adjArray[v][i] == 1 && !visited[i]) {
                dfs_array_recursion(i, adjArray, visited);
            }
        }
    }

```

그래프의 탐색-DFS(인접리스트)

```java
import java.util.*;

public class DFS_List {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt(); // 정점의 개수
        int m = sc.nextInt(); // 간선의 개수
        int v = sc.nextInt(); // 탐색을 시작할 정점의 번호

        boolean visited[] = new boolean[n + 1]; // 방문 여부를 검사할 배열

        LinkedList<Integer>[] adjList = new LinkedList[n + 1];

        for (int i = 0; i <= n; i++) {
            adjList[i] = new LinkedList<Integer>();
        }

        // 두 정점 사이에 여러 개의 간선이 있을 수 있다.
        // 입력으로 주어지는 간선은 양방향이다.
        for (int i = 0; i < m; i++) {
            int v1 = sc.nextInt();
            int v2 = sc.nextInt();
            adjList[v1].add(v2);
            adjList[v2].add(v1);
        }
```

```
        for (int i = 1; i <= n; i++) { // 방문 순서를 위해 오름차순 정렬
            Collections.sort(adjList[i]);
        }

        System.out.println("DFS - 인접리스트");
        dfs_list(v, adjList, visited);
    }

    // DFS - 인접리스트 - 재귀로 구현
    public static void dfs_list(int v, LinkedList<Integer>[] adjList, boolean[] visited) {
        visited[v] = true; // 정점 방문 표시
        System.out.print(v + " "); // 정점 출력

        Iterator<Integer> iter = adjList[v].listIterator(); // 정점 인접리스트 순회
        while (iter.hasNext()) {
            int w = iter.next();
            if (!visited[w]) // 방문하지 않은 정점이라면
                dfs_list(w, adjList, visited); // 다시 DFS
        }
    }
}
```

dfs시간복잡도

- 그래프가 인접 리스트로 표현된 경우 O(n+e)
- 인접 행렬로 표현된 경우 O(n^2)이다.

- 희소 그래프인 경우 인접 리스트의 사용이 인접 행렬보다 유리
- 희소 그래프는 그래프 내에 적은 수의 간선을 가지는 그래프로
- 인접 행렬을 사용하면 메모리의 낭비가 크기 때문

그래프의 탐색-BFS(인접행렬)

```java
import java.util.*;

public class BFS_Array {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt(); // 정점의 개수
        int m = sc.nextInt(); // 간선의 개수
        int v = sc.nextInt(); // 탐색을 시작할 정점의 번호

        boolean visited[] = new boolean[n + 1]; // 방문 여부를 검사할 배열

        int[][] adjArray = new int[n+1][n+1];

        // 두 정점 사이에 여러 개의 간선이 있을 수 있다.
        // 입력으로 주어지는 간선은 양방향이다.
        for(int i = 0; i < m; i++) {
            int v1 = sc.nextInt();
            int v2 = sc.nextInt();

            adjArray[v1][v2] = 1;
            adjArray[v2][v1] = 1;
        }

        System.out.println("BFS - 인접행렬");
        bfs_array(v, adjArray, visited);
    }
```

```java
// BFS - 인접행렬
public static void bfs_array(int v, int[][] adjArray, boolean[] visited) {
    Queue<Integer> q = new LinkedList<>();
    int n = adjArray.length - 1;

    q.add(v);
    visited[v] = true;

    while (!q.isEmpty()) {
        v = q.poll();
        System.out.print(v + " ");

        for (int i = 1; i <= n; i++) {
            if (adjArray[v][i] == 1 && !visited[i]) {
                q.add(i);
                visited[i] = true;
            }
```

그래프의 탐색-BFS(인접리스트)

```java
import java.util.*;

public class BFS_List {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt(); // 정점의 개수
        int m = sc.nextInt(); // 간선의 개수
        int v = sc.nextInt(); // 탐색을 시작할 정점의 번호

        boolean visited[] = new boolean[n + 1]; // 방문 여부를 검사할 배열

        LinkedList<Integer>[] adjList = new LinkedList[n + 1];

        for (int i = 0; i <= n; i++) {
            adjList[i] = new LinkedList<Integer>();
        }

        // 두 정점 사이에 여러 개의 간선이 있을 수 있다.
        // 입력으로 주어지는 간선은 양방향이다.
        for (int i = 0; i < m; i++) {
            int v1 = sc.nextInt();
            int v2 = sc.nextInt();
            adjList[v1].add(v2);
            adjList[v2].add(v1);
        }

```

```
        for (int i = 1; i <= n; i++) {
            Collections.sort(adjList[i]); // 방문 순서를 위해 오름차순 정렬
        }

        System.out.println("BFS - 인접리스트");
        bfs_list(v, adjList, visited);
    }

    // BFS - 인접리스트
    public static void bfs_list(int v, LinkedList<Integer>[] adjList, boolean[] visited) {
        Queue<Integer> queue = new LinkedList<Integer>();
        visited[v] = true;
        queue.add(v);

        while(queue.size() != 0) {
            v = queue.poll();
            System.out.print(v + " ");

            Iterator<Integer> iter = adjList[v].listIterator();
            while(iter.hasNext()) {
                int w = iter.next();
                if(!visited[w]) {
                    visited[w] = true;
                    queue.add(w);
                }
            }
```

bfs시간복잡도

- 정점의 수가 n이고, 간선의 수가 e인 그래프의 경우
- 그래프가 인접 리스트로 표현된 경우 O(n+e)
- 인접 행렬로 표현된 경우 O(n^2)이다.
- 희소 그래프인 경우 인접 리스트의 사용이 인접 행렬보다 유리
- (희소 그래프는 그래프 내에 적은 수의 간선을 가지는 그래프로 인접 행렬을 사용하면 메모리의 낭비가 크기 때문)
-