그래프

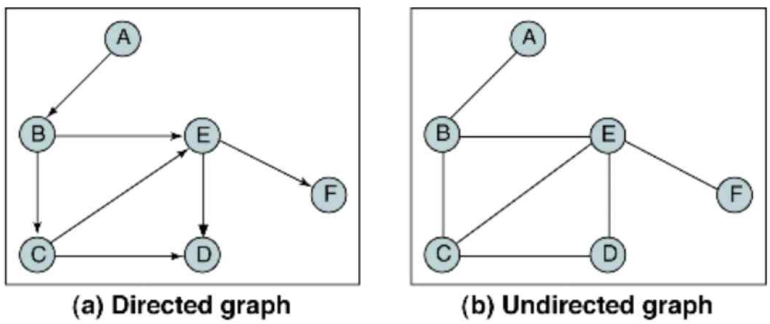
그래프란? "비선형 구조"

선형구조는 자료를 저장하고 꺼내는 것에 초점이 맞춰져 있고, 비선형구조는 표현에 초점이 맞춰져 있습니다.
이번 자료구조인 그래프는 바로 연결 관계에 초점이 맞춰져 있습니다.

그래프에서 사용되는 용어들을 정리
노드(Node): 연결 관계를 가진 각 데이터를 의미합니다.
정점(Vertex)이라고도 합니다.
간선(Edge): 노드 간의 관계를 표시한 선.
인접 노드(Adjacent Node): 간선으로 직접 연결된 노드(또는 정점)

◓ 그래프는 유방향 그래프와 무방향 그래프 두 가지가 있다

무방향 그래프에 대해서만 학습함.
유방향 그래프(Directed Graph): 방향이 있는 간선을 갖습니다. 간선은 단방향 관계를 나타내며, 각 간선은 한 방향으로만 진행할 수 있습니다.
무방향 그래프(Undirected Graph)는 방향이 없는 간선을 갖습니다

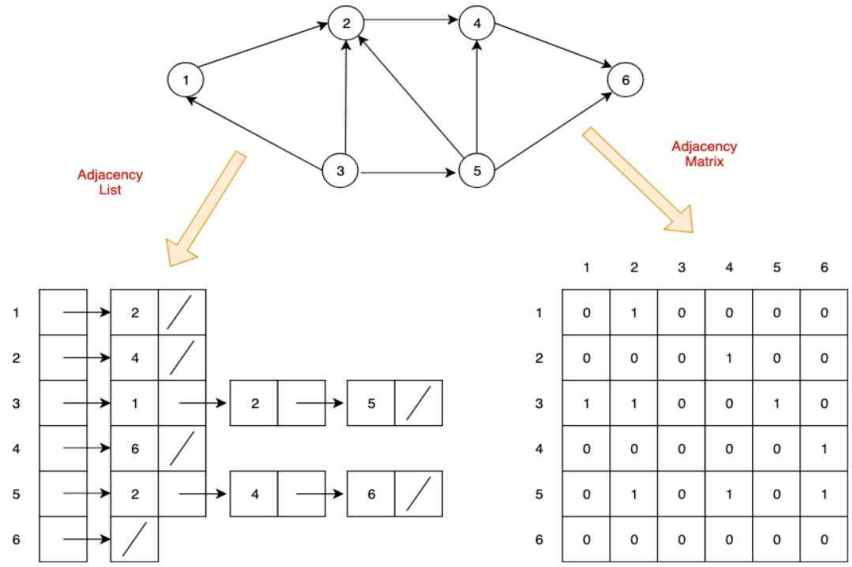


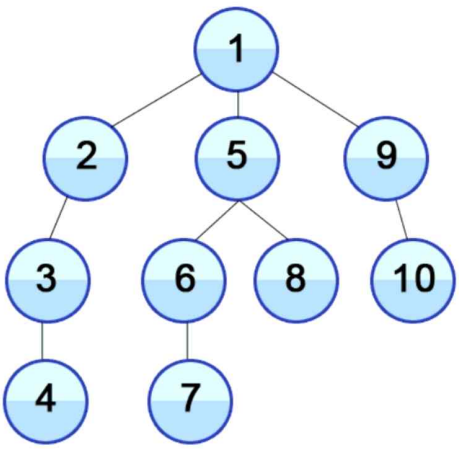
▼ 그래프의 표현 방법

이런 그래프라는 개념을 컴퓨터에서 표현하는 방법은 두 가지 방법이 있습니다!
1) 인접 행렬(Adjacency Matrix): 2차원 배열로 그래프의 연결 관계를 표현
2) 인접 리스트(Adjacnecy List): 링크드 리스트로 그래프의 연결 관계를 표현

시간 vs 공간

> 인접 행렬으로 표현하면 즉각적으로 0과 1이 연결되었는지 여부를 바로 알 수 있습니다.
> 그러나, 모든 조합의 연결 여부를 저장해야 되기 때문에 $O(\text{노드}^2)$ 만큼의 공간을 사용해야 합니다.
>
> 인접 리스트로 표현하면 즉각적으로 연결되었는지 알 수 없고, 각 리스트를 돌아봐야 합니다.
> 따라서 연결되었는지 여부를 알기 위해서 최대 $O(\text{간선})$ 만큼의 시간을 사용해야 합니다.
> 대신 모든 조합의 연결 여부를 저장할 필요가 없으니 $O(\text{노드} + \text{간선})$ 만큼의 공간을 사용하면 됩니다.





DFS란? 자료의 검색, 트리나 그래프를 탐색하는 방법. 한 노드를 시작으로 인접한 다른 노드를 재귀적으로 탐색해가고 끝까지 탐색하면 다시 위로 와서 다음을 탐색하여 검색한다. [컴퓨터인터넷IT용어대사전]

BFS란?
한 노드를 시작으로 인접한 모든 정점들을 우선 방문하는 방법. 더 이상 방문하지 않은 정점이 없을 때까지 방문하지 않은 모든 정점들에 대해서도 넓이 우선 검색을 적용한다. [컴퓨터인터넷IT용어대사전]

왜 DFS & BFS 를 배울까요?
정렬된 데이터를 이분 탐색하는 것처럼 아주 효율적인 방법이 있는 반면에,모든 경우의 수를 전부 탐색해야 하는 경우도 있습니다.DFS 와 BFS 는 그 탐색하는 순서에서 차이가 있습니다.DFS 는 끝까지 파고드는 것이고,BFS 는 갈라진 모든 경우의 수를 탐색해보고 오는 것이 차이점입니다.

DFS 는 끝까지 파고드는 것이라, 그래프의 최대 깊이 만큼의 공간을 요구합니다. 따라서 공간을 적게 씁니다. 그러나 최단 경로를 탐색하기 쉽지 않습니다.
BFS 는 최단 경로를 쉽게 찾을 수 있습니다! 모든 분기되는 수를 다 보고 올 수 있으니까요.
그러나, 모든 분기되는 수를 다 저장하다보니 공간을 많이 써야하고, 모든 걸 다 보고 오다보니 시간이더 오래걸릴 수 있습니다.

DFS는 Depth First Search 입니다.갈 수 있는 만큼 계속해서 탐색하다가 갈 수 없게 되면 다른 방향으로 다시 탐색하는 구조입니다.

stack과 재귀호출이용

```java
import java.util.ArrayList;
import java.util.Stack;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class DepthFirstSearch {

    public static void main(String[] args) {
        Map<Integer, List<Integer>> graph = new HashMap<>();
        graph.put(1, List.of(2, 5, 9));
        graph.put(2, List.of(1, 3));
        graph.put(3, List.of(2, 4));
        graph.put(4, List.of(3));
        graph.put(5, List.of(1, 6, 8));
        graph.put(6, List.of(5, 7));
        graph.put(7, List.of(6));
        graph.put(8, List.of(5));
        graph.put(9, List.of(1, 10));
        graph.put(10, List.of(9));

        ArrayList<Integer> dfsRecursiveResult = new ArrayList<>();
        dfsRecursive(graph, 1, dfsRecursiveResult) ;
        List<Integer> dfsStackResult = dfsStack(graph, 1);

        System.out.println(dfsRecursiveResult); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        System.out.println(dfsStackResult); // [1, 9, 10, 5, 8, 6, 7, 2, 3, 4]
    }

    private static void dfsRecursive(Map<Integer, List<Integer>> graph, int node, List<Integer> visited) {
        visited.add(node);

        for (int adj : graph.get(node)) {
            if (!visited.contains(adj)) {
                dfsRecursive(graph, adj, visited);
            }
        }

    }

    private static List<Integer> dfsStack(Map<Integer, List<Integer>> graph, int start) {
        List<Integer> visited = new ArrayList<>();
        Stack<Integer> stack = new Stack<>();
        stack.push(start);

        while (!stack.isEmpty()) {
            int top = stack.pop();
            visited.add(top);

            for (int adj : graph.get(top)) {
                if (!visited.contains(adj)) {
                    stack.push(adj);
                }
            }
        }

        return visited;
    }
}
```

```java
package dfs;

import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

public class DepthFirstSearch2 {

    public static void main(String[] args) {
        int[][] graph = {
                {0, 1, 0, 0, 1, 0, 0, 0, 0, 1},
                {1, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 1, 0, 1, 0, 0, 0, 0, 0, 0},
                {0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {1, 0, 0, 0, 0, 1, 0, 1, 0, 0},
                {0, 0, 0, 0, 1, 0, 1, 0, 0, 0},
                {0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
                {0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
                {1, 0, 0, 0, 0, 0, 0, 0, 1, 0}
        };

        List<Integer> dfsRecursiveResult = dfsRecursive(graph, 1, new ArrayList<>());
        List<Integer> dfsStackResult = dfsStack(graph, 1);

        System.out.println(dfsRecursiveResult); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        System.out.println(dfsStackResult); // [1, 9, 8, 5, 6, 7, 4, 2, 3, 10]
    }

    private static  void  dfsRecursive(int[][] graph, int node, List<Integer> visited) {
        visited.add(node);

        for (int adj = 1; adj <= graph.length; adj++) {
            if (graph[node - 1][adj - 1] == 1 && !visited.contains(adj)) {
                dfsRecursive(graph, adj, visited);
            }
        }

    }

    private static List<Integer> dfsStack(int[][] graph, int start) {
        List<Integer> visited = new ArrayList<>();
        Stack<Integer> stack = new Stack<>();
        stack.push(start);

        while (!stack.isEmpty()) {
            int top = stack.pop();
            visited.add(top);

            for (int adj = 1; adj <= graph.length; adj++) {
                if (graph[top - 1][adj - 1] == 1 && !visited.contains(adj)) {
                    stack.push(adj);
                }
            }
        }

        return visited;
    }
}
```

```java
import java.util.*;

public class Main {
    private static void dfs(int[][] graph, boolean[] visited, int vertex) {
        visited[vertex] = true;
        System.out.print(vertex + " ");

        for (int i = 0; i < graph.length; i++) {
            if (graph[vertex][i] == 1 && !visited[i]) {
                dfs(graph, visited, i);
            }
        }
    }

    public static void main(String[] args) {
        // 정점의 수
        int vertices = 4;

        // 인접 행렬 초기화
        int[][] graph = new int[vertices][vertices];

        // 간선 추가
        graph[0][1] = 1;
        graph[0][2] = 1;
        graph[1][2] = 1;
        graph[2][0] = 1;
        graph[2][3] = 1;
        graph[3][3] = 1;

        boolean[] visited = new boolean[vertices];

        // DFS 탐색 시작 (시작 정점: 2)
        System.out.println("DFS 탐색 결과 (시작 정점: 2):");
        dfs(graph, visited, 2);
    }
}
```

dfs기본
-모든 노드 방문하기

```
///////////////
import java.util.Stack;

public class Main {
    private static void dfs(int[][] graph, int startVertex) {
        int vertices = graph.length;
        boolean[] visited = new boolean[vertices];

        Stack<Integer> stack = new Stack<>();
        stack.push(startVertex);

        while (!stack.isEmpty()) {

            int currentVertex = stack.pop();        //스택에서  pop  -> 방문하지 않으면  방문배열에 추가

            if (!visited[currentVertex]) {
                System.out.print(currentVertex + " ");
                visited[currentVertex] = true;
            }

            //인접리스트 방문, 방문하지 않은 인접노드 스택에 담기
            for (int i = 0; i < vertices; i++) {
                if (graph[currentVertex][i] == 1 && !visited[i]) {
                    stack.push(i);
                }
            }
        }
    }

    public static void main(String[] args) {
        // 정점의 수
        int vertices = 4;

        // 인접 행렬 초기화
        int[][] graph = new int[vertices][vertices];

        // 간선 추가
        graph[0][1] = 1;
        graph[0][2] = 1;
        graph[1][2] = 1;
        graph[2][0] = 1;
        graph[2][3] = 1;
        graph[3][3] = 1;

        // DFS 탐색 시작 (시작 정점: 2)
        System.out.println("DFS 탐색 결과 (시작 정점: 2):");
        dfs(graph, 2);
    }
}
```