구디 아카데미 알고리즘 스터디 중급반 - 8주차

[멘토 유준혁]

스터디 주제

- 그래프와 트리 자료구조 및 트리의 종류
- 인접 행렬과 인접 리스트의 차이, 트리의 순회 방법
- DFS와 BFS 개념 및 특징 설명

그래프 (Graph) 란 무엇인가?

이번 주에는 오랜만에 자료구조를 살펴보려고 한다.

그 동안 했던 자료구조들이 모두 **선형적인 자료구조** 였던 반면,

이번 주는 **비선형 자료구조**에 해당하는 그래프와 트리에 대해서 알아보려고 한다.

우선 그래프의 사전적 정의를 먼저 살펴보자.

> 그래프 $G(V, E)$ 는
> 어떤 자료나 개념을 표현하는 노드 (node) 들의 집합 V 과
> 이들을 연결하는 간선 (edge) 들의 집합 E 로 구성된 자료구조

설명은 어렵게 되어있지만 활용 사례를 보면 감이 잡힐 것이다.

간단한 예로는 **사람들 간의 지인 관계, 여러 도시들을 연결하는 도로망, 혹은 지하철 노선도** 까지도 그래프의 한 예라고 할 수 있다.

이처럼 현실 세계의 사물이나 추상적인 개념 간의

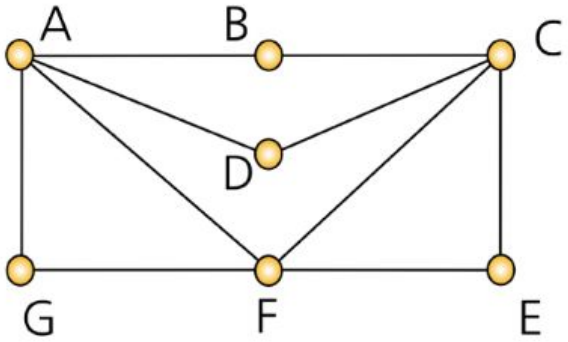
연결 관계를 표현하는 자료구조를 그래프라고 부른다.

한 번씩은 접해봤을 '한 붓 그리기' 역시 그래프에 속한다고 볼 수 있다.

어떤 그래프가 있을 때, 이 그래프의 모든 변을 단 한 번씩만 통과하는 경로를 오일러 경로라고 하는데, 여기서 같은 꼭짓점에서 시작하고 끝날 경우 이 경로를 오일러 회로라고 부른다. 오일러는 어떤 그래프가 오일러 회로를 가질 필요충분조건을 다음과 같이 언급하고 있다.

1. 연결된 그래프

2. 모든 꼭짓점은 짝수점



대표적인 오일로 회로

(한 붓 그리기의 가능 조건인 오일러 경로)

이런 단순한 그래프 외에도 가중치 그래프, 방향 그래프, DAG 등 여러 그래프가 존재한다.

모두 설명하고 지나가기엔 종류가 워낙 다양하여 트렐로에 사이트 링크를 올려두도록 하겠다.

그래프를 어떻게 표현하는가?

그래프의 정의를 알아보았으니 이번엔 그래프의 표현 방법을 알아볼 차례다.

기존의 선형 자료구조들은 대부분의 언어들에서 구현이 되어 있는 편이다.

기본적인 배열부터 스택이나 큐, 그리고 Collection 자료구조까지.

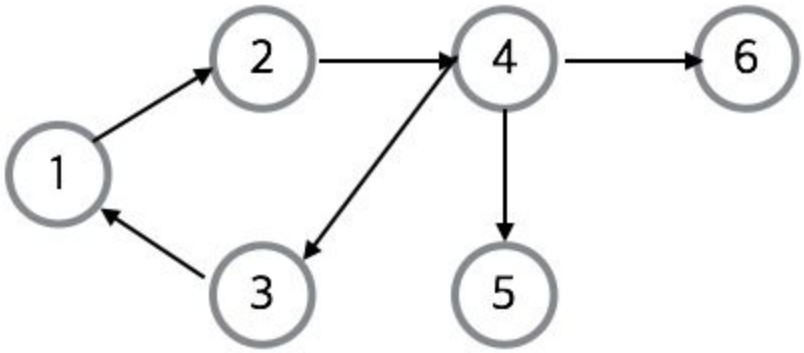
반면에 그래프의 경우 대부분 구현이 되어있지 않다.

구현이 되어있지 않다기보단, 구현 할 수 없다는 표현이 더 적절할 듯 싶다.

그래프라는 자료구조 자체가 워낙 다양한 방식으로 표현이 가능하기에 (트렐로 링크 확인)

고정된 클래스를 만들기 보단 사용자가 필요에 따라 직접 만들어 사용하는 편이다.

이 그래프를 표현함에 있어서 크게 두 가지의 방식이 있는데 바로 **인접 리스트와 인접 행렬** 이다.



(그래프 예시)

인접 리스트 표현

인접 리스트 표현은 그래프의 각 정점마다 해당 정점에서 나가는 간선의 목록을 저장하여 표현한다.

위의 예시 사진을 보면서 설명을 읽어주기를 바란다.

1번의 정점을 보면 2번과 연결 (단방향) 되어 있음을 알 수 있다.

이를 간단히 표현해보면 1 => 2 처럼 표현 할 수 있겠다.

마찬가지로 2 => 4, 3 => 1, 4 => 5, 6 … 으로 표현이 가능하다.

단, 이렇게 표현 할 경우 각 정점에서 이어지는 간선의 개수는 제각각일 것 이다.

(6번 정점의 경우 간선이 0개 이지만 4번 정점의 경우는 2개가 연결 되어 있다.)

따라서 정적으로 메모리를 할당하는 배열의 방식이 아닌, 동적으로 할당시키는 List를 활용하게 된다.

그래서 이런 식으로 표현하는 방식을 **인접 리스트** 라고 부른다.

이는 메모리를 효율적으로 사용하기에 공간 복잡도를 현저하게 낮출 수 있는 방식이다.

총 V 개의 정점에 E 개의 간선이 연결되어 있다고 한다면 $O(V+E)$ 만큼의 공간을 사용하게 된다.

인접 행렬 표현

인접 행렬 역시 각 정점에 연결된 간선의 정보를 저장함에 있어서 인접 리스트와 다른 점은 없다.

단순히 인접 행렬이 만들어진 이유는 인접 리스트의 단점 때문이다.

바로 두 정점이 연결되어 있는 지를 판단할 때 모든 정보를 일일이 뒤져야 한다는 점이다.

> *‘ 리스트도 인덱스로 접근이 가능하니까 탐색에 걸리는 시간 복잡도는 차이가 없지 않나요? ‘*

맞다. 인덱스를 알고있다면 시간 복잡도에 있어서 차이는 없다.

그렇지만 그 리스트에 해당 인덱스 (정점) 가 존재한다는 확신 할 수 없다는 게 문제다.

그래프 예시에서 6번 정점을 보자. 해당 정점은 다른 어느 정점과도 연결되어 있지 않기에

인접 리스트에 6번 정점에서 시작하는 간선이 존재하지 않을 것이다.

> *‘ List.contains(n) 을 사용하면 존재 여부를 알 수 있지 않나요? ’*

이것 역시 맞는 말이긴 하지만 **기본 클래스에서 제공되는 메서드는 마법이 아니다.**

해당 메서드를 사용하면 리턴 값이 바로 나오기에 O(1) 이 걸린다고 생각할 수 있겠지만,

내부 코드를 살펴보면 전혀 그렇지 않다.

오히려 존재하는 지 알기 위해 처음 부터 차례대로 확인함을 알 수 있다.

따라서 해당 값에 바로 접근이 가능한, 배열을 사용하게 되는 것이다.

```java
    public boolean contains(Object o) {
        return indexOf(o) >= 0;
    }

    /**
     * Returns the index of the first occurrence of the specified element
     * in this list, or -1 if this list does not contain the element.
     * More formally, returns the lowest index {@code i} such that
     * {@code Objects.equals(o, get(i))},
     * or -1 if there is no such index.
     */
    public int indexOf(Object o) {
        return indexOfRange(o, 0, size);
    }

    int indexOfRange(Object o, int start, int end) {
        Object[] es = elementData;
        if (o == null) {
            for (int i = start; i < end; i++) {
                if (es[i] == null) {
                    return i;
                }
            }
        } else {
            for (int i = start; i < end; i++) {
                if (o.equals(es[i])) {
                    return i;
                }
            }
        }
        return -1;
    }
```

(JAVA ArrayList의 contains 메서드)

두 정점이 연결되어 있으면 1, 아니라면 0 으로 표현한다고 가정하고 배열을 생성해보자.

가장 초기 단계인 배열의 크기를 설정할 때 어떻게 설정해야 할까?

그래프 예시에서 보면 알 수 있다. 바로 **그래프의 정점의 개수** 만큼 설정을 해주면 된다.

총 6개의 정점이 있으므로 이 배열의 크기는 6 * 6 만큼의 크기를 가지게 될 것이다.

그리고 나서 간선의 정보 대로 값을 채워나가게 될 것이다.

그렇다면 만약 총 100,000 개의 정점과 100개의 간선이 있다면 어떨까?

이 배열의 크기는 100,000 * 100,000 이므로 총 100억개의 정보를 저장하는 배열이 될 것이다.

JAVA에서 이 배열을 선언 및 초기화 하는 구문을 실행해보면 다음과 같은 에러가 나온다.

```java
import java.util.*;
public class Main {
    public static void main(String[] args) {
        int[][] arr = new int [100000][100000];
    }
}

```

```
<terminated> Main (1) [Java Application] C:₩Program Files₩Java₩jdk-11.0.1₩bin₩javaw.exe (2019. 12. 9. 오전 12:26:12)
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at Main.main(Main.java:4)
```

아직 간선에 대한 정보를 저장하지도 못했는데 배열 초기화 과정에서부터 에러가 나버리는 것이다.

인접 행렬로 그래프를 표현할 경우 공간 복잡도는 $O(V^2)$ 이기 때문이다.

여기서 알 수 있듯 인접 행렬과 인접 리스트의 차이는 배열과 리스트의 차이와 동일하다.

메모리를 동적으로 할당하느냐, 아니느냐의 차이 말이다.

따라서 정점의 개수가 많고 간선의 정보가 적다면 인접 리스트를,

정점의 개수가 적고 간선의 정보가 많다면 인접 행렬을 사용하면 된다.

트리 (Tree) 란 무엇인가?

그래프에 대한 긴 설명이 끝났으니 이번엔 트리를 할 차례다.

트리는 그래프와는 달리 간단히 설명하고 끝이 날 것이다.

트리가 그래프의 한 종류이기 때문이다.

따라서 여기선 트리와 그래프의 차이점을 주로 살펴보고

트리의 종류에 대해선 그래프와 마찬가지로 링크를 올려둘테니 개인적으로 확인하시길 바란다.

트리가 그래프의 한 종류라고 했으니 그래프의 정의가 트리에도 당연히 적용된다.

트리 역시 정점과 간선으로 이루어져 있다는 말이다.

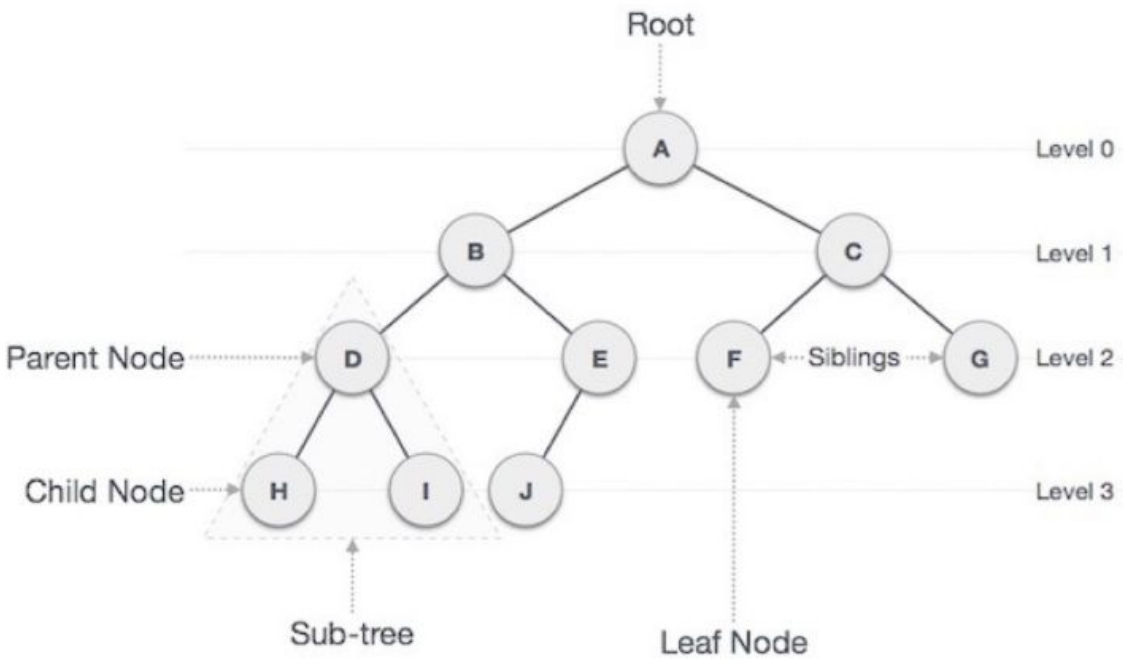
단, 차이점이 있다면 트리 같은 경우 계층 구조로 이루어져 있다는 점이다.

확실한 최상위 부모 (루트 노드) 와 이의 자식 노드들로 구성이 되어있다.

또한 사이클이 이루어지지 않는다. (3개 이상의 간선이 다각형을 이루는 구조)

더 이상 자식을 가지지 않는 노드를 Leaf Node (나뭇잎 노드) 라고 부른다.

간선의 개수는 정점의 개수 - 1 이다.



(트리 구조에 대한 설명)

트리의 표현 방법 역시 그래프와 동일하다.

인접 리스트 혹은 인접 행렬로 표현 할 수 있다는 말이다.

> **특이하게도 완전 이진 트리의 경우 1차원 배열으로도 표현이 가능하다.**
>
> **이 경우는 기회가 되면 설명하도록 하겠다.**

트리의 순회 방법

트리에 대한 설명이 끝났으니 이번엔 추가적으로 알아둬야 할 순회 방법에 대해 알아보자.

트리의 노드들을 방문하는 데는 대표적으로 다음의 4개의 방법이 존재한다.

1. **전위 순회 (Pre - Order)**
2. **중위 순회 (In - Order)**
3. **후위 순회 (Post - Order)**
4. **레벨 순회 (Level - Order)**

정보처리 자격증을 준비했다면 한 번쯤 봤을 것이다.

간략히 설명하자면

> **전위 순회 => 루트 / 왼쪽 / 오른쪽**
>
> **중위 순회 => 왼쪽 / 루트 / 오른쪽**
>
> **후위 순회 => 왼쪽 / 오른쪽 / 루트**
>
> **레벨 순회 => 계층 레벨이 가장 높은 곳부터 차례대로**

순으로 방문하는 것을 뜻한다.

실행 순서 같은 경우는 스터디 시간에 했던 설명을 떠올리시길 바라며

코드를 첨부하고 자료구조에 관한 이론은 끝마치도록 하겠다.

```
class Node {
	private int data;
	private Node left;
	private Node right;

	// Getter, Setter, toString();
	public Node(int data) {
		this.data = data;
	}

	public int getData() {
		return data;
	}

	public void setData(char data) {
		this.data = data;
	}

	public Node getLeft() {
		return left;
	}

	public void setLeft(Node left) {
		this.left = left;
	}

	public Node getRight() {
		return right;
	}

	public void setRight(Node right) {
		this.right = right;
	}

	public String toString() {
		return this.data + "";
	}
}
```

```
public class Main {
	static public void inorder(Node n) {
		if (n != null) {
			inorder(n.getLeft());
			System.out.print(n.getData()+" ");
			inorder(n.getRight());
		}
	}

	static public void preorder(Node n) {
		if (n != null) {
			System.out.print(n.getData() + " ");
			preorder(n.getLeft());
			preorder(n.getRight());
		}
	}

	static public void postorder(Node n) {
		if (n != null) {
			postorder(n.getLeft());
			postorder(n.getRight());
			System.out.print(n.getData() + " ");
		}
	}

	public static void main(String[] args) {
		/*
		 *      트리의 모습
		 *           1
		 *         /  \
		 *        /    \
		 *       2      3
		 *      / \    / \
		 *     4   5   6  7
		 *          \    /
		 *          8   9
		 */
```

```
            // 트리를 생성하고 자료를 넣는 과정
            Node rootNode = new Node(1);
            rootNode.setLeft(new Node(2));
            rootNode.setRight(new Node(3));
            rootNode.getLeft().setLeft(new Node(4));
            rootNode.getLeft().setRight(new Node(5));
            rootNode.getRight().setLeft(new Node(6));
            rootNode.getRight().setRight(new Node(7));
            rootNode.getLeft().getRight().setRight(new Node(8));
            rootNode.getRight().getRight().setLeft(new Node(9));

            // in-order, pre-order, post-order

System.out.println("\n\n\n=============inorder============");
            inorder(rootNode);


System.out.println("\n\n\n=============preorder============");
            preorder(rootNode);


System.out.println("\n\n\n=============postorder============");
            postorder(rootNode);
      }
}
```

그래서 DFS와 BFS는 뭔데?

드디어 본래의 목적인 DFS와 BFS를 공부해볼 차례다.

DFS (depth - first search) 와 BFS (breadth - first search) 가 뭐길래 이렇게 뜸을 들였을까?

용어의 의미에서 느끼셨겠지만 **DFS와 BFS는 그래프 탐색 방법 중 하나다.**

그래프의 모든 정점을 방문하는 방식을 두 가지로 나눈 것이다.

> **DFS (깊이 우선 탐색)는 말 그대로 깊이를 우선적으로 탐색하기 때문에 현재의 레벨과는 상관없이 더 이상 자식 노드가 없는 곳까지 내려가서 탐색을 하는 방법을 뜻한다.**
>
> **BFS (넓이 우선 탐색)는 깊이 보단 넓이 (레벨) 를 우선적으로 탐색하여 원을 그리듯이 탐색하는 방법을 뜻한다.**

우리는 트리의 순회 방법에 대해서 이미 알아봤었다.

대표적인 4가지 방법 중 1, 2, 3번째 방법이 DFS에 해당하고,

4번째의 Level - Order 가 바로 BFS에 해당한다.

(위의 코드에서는 Level - Order (BFS) 가 없다. 깃허브의 대표 예제 소스를 확인하길 바란다.)

기본적인 정보는 알았으니 이번엔 구현 방법의 차이와 특징에 대해서 살펴보자.

DFS의 구현 방법과 특징

DFS는 그래프의 모든 정점을 발견하는 가장 단순하고 고전적인 방법이다.

> **아직 방문하지 않는 정점으로 향하는 간선이 존재한다면 해당 간선을 따라가고,**
>
> **더이상 갈 곳이 없다면 다시 돌아가는 방법이다.**

미로에 갇혔을 때 탈출하는 가장 쉬운 방법이기도 하다.

트리 순회 방법 코드를 보시면 아시겠지만 DFS는 보통 재귀 함수를 사용한다.

더 이상 갈 곳이 없을 때 다시 돌아간다는 특성이 재귀 함수를 구현하면 간단히 해결되기 때문이다.

따라서 다음과 같은 규칙만 지킨다면 10줄 이내의 코드로 DFS를 구현할 수 있게 된다.

1. **더 깊이 들어갈 수 있고 그 곳이 방문하지 않은 곳이면 해당 정점에서 다시 탐색**
2. **더 깊이 들어갈 수 없다면 특정한 코드 실행 후 방문했음을 체크하고 복귀**
3. **탐색할 곳이 없을 때 까지 반복**

이런 단순한 방법을 잘 활용한다면

위상 정렬, 백트래킹, 오일러 서킷, 절단점 찾기, 사이클 여부 등 여러 문제들을 풀 수 있게 된다.

특히나 위상 정렬 같은 경우 코딩테스트에서 꾸준히 나오므로 잘 익혀두자.

BFS의 구현 방법과 특징

드디어 마지막인 BFS를 설명할 시간이다.

DFS가 초기의 정점에서 얼마나 깊이 들어갔는 지를 체크하지 않으면서 마구잡이로 탐색을 했다면

BFS는 반대로 Level (깊이) 을 중점적으로 체크하면서 탐색하는 방식을 뜻한다.

그렇다면 왜 Level을 끌고가면서 탐색을 하는 걸까?

어차피 모든 정점을 탐색하는 데 굳이 Level이 필요할까?

이렇게 탐색을 하는 주된 이유는 BFS 탐색을 하면서 얻는 가장 처음 정점이 바로

초기 정점과의 최단 거리 정점이기 때문이다.

애초에 BFS는 모든 정점을 탐색하기 위한 방법이라기 보단

최단 거리 정점을 찾아내기 위한 방법에 더 가깝다고 본다.

최단 거리 정점을 찾으려다 보니 모든 정점을 탐색하는,

어찌 보면 인과 관계가 뒤집혔다고도 볼 수 있겠다.

BFS 프로세스 과정을 살펴보면 다음과 같다. **(현재의 Level을 n 이라고 가정한다.)**

1. **초기 정점에서 n 번째에 해당하는 정점을 모두 탐색한다.**
2. **조건과 맞지 않다면 n을 1 증가 시킨다.**
3. **탐색할 곳이 없을 때 까지 반복**

때문에 BFS를 구현하기 위해서 선형 자료구조인 큐를 사용하게 된다.

큐에 가장 처음의 정점을 넣어준 뒤 (0번째 Level 정점)

큐 맨 앞 부터 빼면서 해당 정점과 이어진 정점을 모두 큐에 add하면서 조건을 확인하면

결국 Level을 순차적으로 방문함과 동일하니까 말이다.

이렇게 그래프의 탐색 방법이 끝이 났다. 나머지는 예제를 충분히 풀어보며 감을 익혀보자.

PS.

중급 알고리즘 스터디 멘티 여러분들 프로젝트 하면서 스터디도 하느라 모두 고생하셨습니다.

다음 주가 마지막 스터디 시간이지만 이론은 현재 문서가 마지막일 것 같네요.

원래 기획했던 3개월 보다 못미치는 약 2개월을 기점으로 스터디가 마무리 되는데

스터디가 조금이나마 도움이 됐다면 좋겠네요.

평생 처음 맡아보는 멘토라 저 역시도 부족함을 많이 느껴서 아쉽네여.

더 친근하고 재밌게 진행을 하지 못한 건 죄송함다 ;ㅅ;

기회가 된다면 회식? 같은 것도 해보고 싶었는데.. 아쉬운 것 밖에 생각이 안나네요ㅋㅋㅋ

구디의 밤 때 재학생도 오나..? 모르겠네요 ㅇㅅㅇ 어찌됐든 다들 고생하셨습니다!

제가 준비한 것 외에도 훨씬 많은 알고리즘 분류가 존재하니까 개인적인 공부는 놓지 마시구요!

학원 남은 기간 뿌듯한 시간을 보내시길 바랄게요.

연말 잘 보내시고 새해 복 많이 받으세요~

구디아카데미 중급 알고리즘 스터디 1기 멘토 유준혁