



땅울림 자구 스터디

11주차

1

**Binary
Search Tree**

2

Vector

3

**Graph
Theory**

1. Binary Search Tree

1 BST

Binary Search(이진 탐색)

- 정렬된 데이터에서 값을 빠르게 찾기 위한 탐색
- 집합의 가운데 값과 목표값을 비교
- 목표값이 더 크다면 집합의 오른쪽을,
목표값이 더 작다면 집합의 왼쪽을 탐색

1 BST

Binary Search Tree(BST)

- 이진 탐색을 트리로 구현해놓은 자료구조
- 왼쪽 자식은 부모보다 작고, 오른쪽 자식은 부모보다 큼
- 균형 트리, 불균형 트리가 있음

1 BST

이진 탐색 vs 이진 탐색 트리

- 이진 탐색은 배열이므로 크기가 고정되어있고 삽입, 삭제가 힘들
- 이진 탐색 트리는 크기에 제한이 없고 삽입, 삭제에 유리

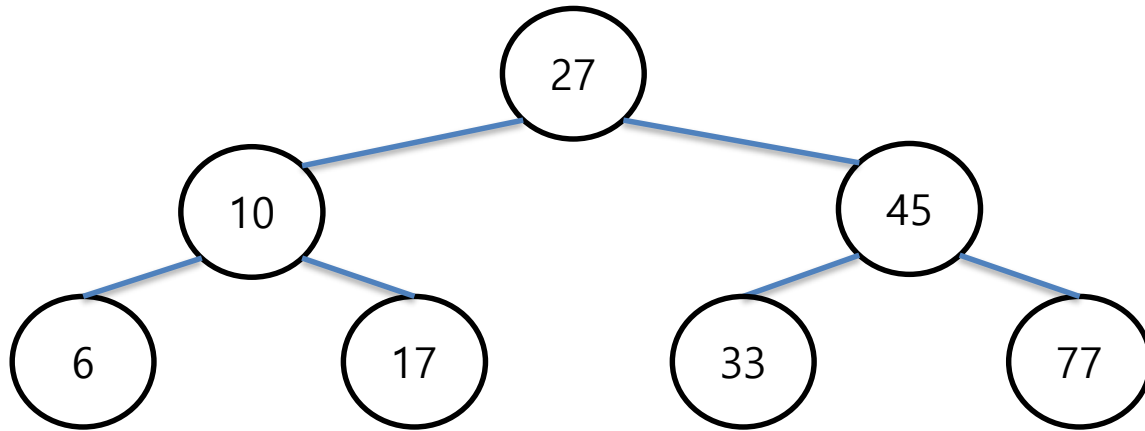
1 BST

이진 탐색 트리의 성능

	BST(불균형), 평균	BST(불균형), 최악	BST(균형)
탐색	$O(\log n)$	$O(n)$	$O(\log n)$
삽입	$O(\log n)$	$O(n)$	$O(\log n)$
삭제	$O(\log n)$	$O(n)$	$O(\log n)$

1 BST

불균형 이진 탐색 트리

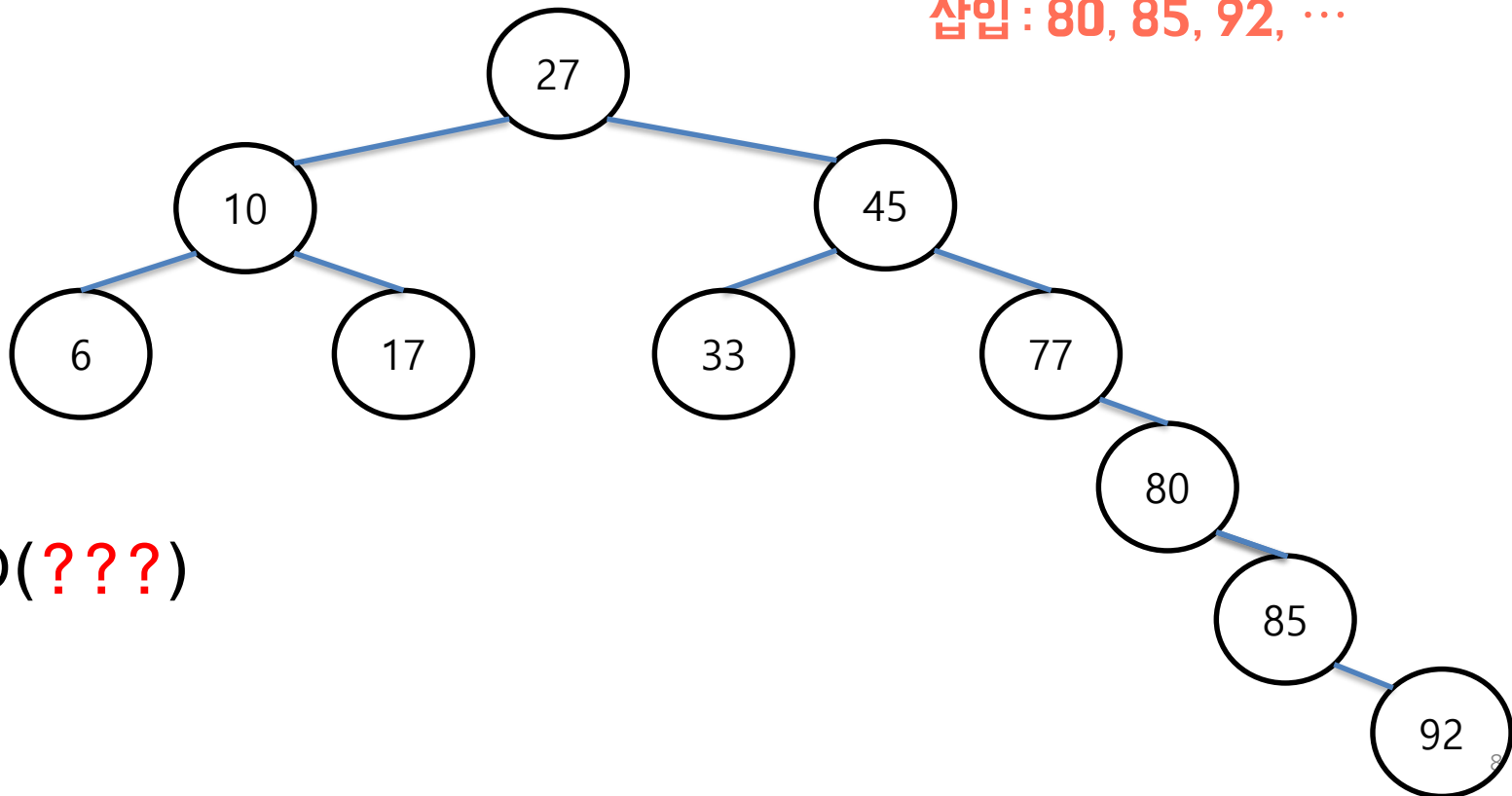


탐색 : $O(\log N)$

1 BST

불균형 이진 탐색 트리

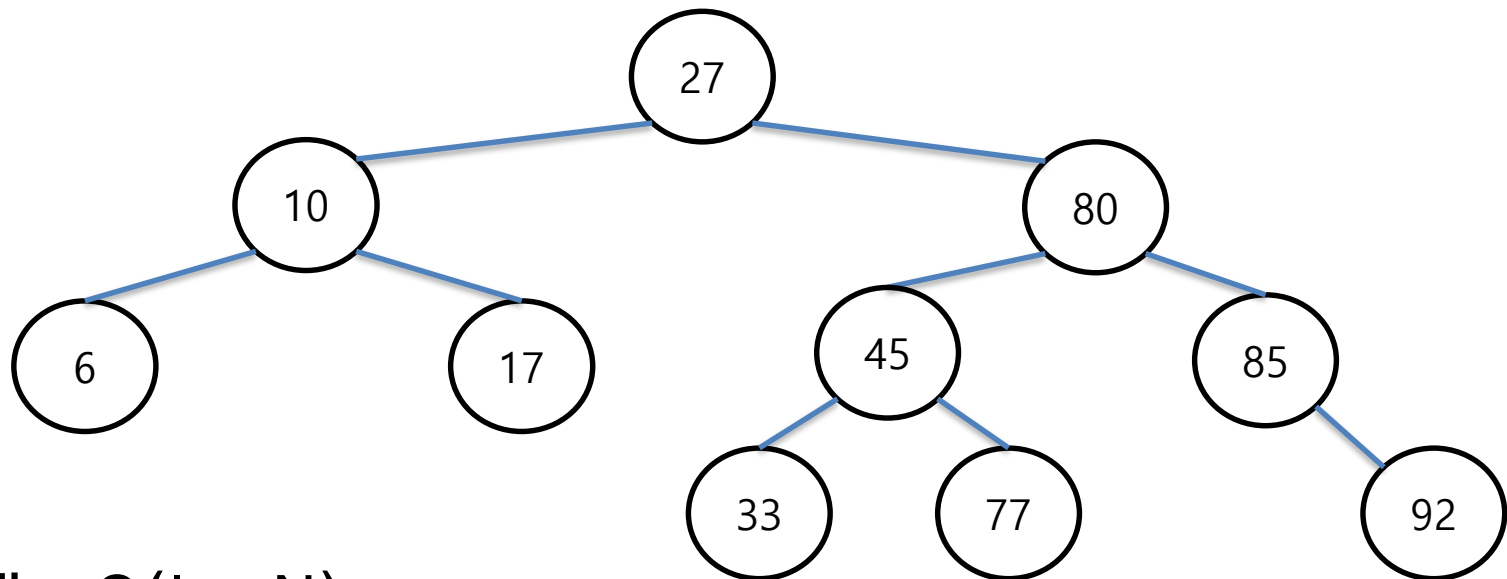
삽입 : 80, 85, 92, ...



탐색 : $O(???)$

1 BST

균형 이진 탐색 트리



탐색 : $O(\log N)$

1 BST

균형 이진 탐색 트리(AVL 트리)

- 이진 탐색 트리에서 일반적인 이진 트리처럼 데이터를 삽입, 삭제시키면 높이 균형이 맞지 않게 됨
- 노드를 삽입, 삭제할 때 회전을 통해 트리를 재구성해서 계속해서 높이를 $\log N$ 으로 유지시킴
- 삽입, 삭제, 탐색이 모두 $O(\log N)$

1 BST

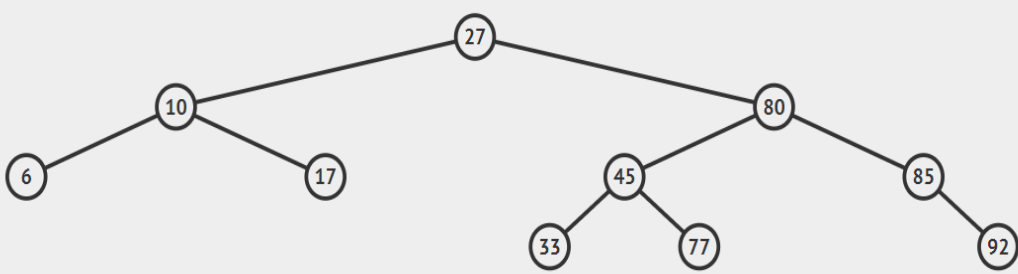
참고할만한 블로그

- C++ AVL 트리 알고리즘 완벽 소스 코드 및 정리!
- AVL Tree
- AVL트리(Adelson-Velskii / Landis)

AVL 트리 시뮬레이션 사이트

<https://visualgo.net/en/bst>

7 VISUALGO.NET / en /bst BINARY SEARCH TREE AVL TREE Exploration Mode Login



```
graph TD; 27((27)) --- 10((10)); 27 --- 80((80)); 10 --- 6((6)); 10 --- 17((17)); 80 --- 45((45)); 80 --- 85((85)); 45 --- 33((33)); 45 --- 77((77)); 85 --- 92((92));
```

1 Create
2 Empty
3 Insert(v)

Unbalanced Example | Balanced Example | Random | Skewed Left | Skewed Right

a,b,c 형식으로 여러 개
한꺼번에 insert 가능

Insert 80,85,92

The tree is now balanced.

```
insert v
check balance factor of this and its children
case1: this.rotateRight
case2: this.left.rotateLeft, this.rotateRight
case3: this.rotateLeft
case4: this.right.rotateRight, this.rotateLeft
this is balanced
```

2. Vector

2 vector

벡터 선언 방법

- `vector <int> v;`
- `vector <int> v(N);`
- `vector <int> v[9];`
- `vector <vector <int>> v;`

2 vector

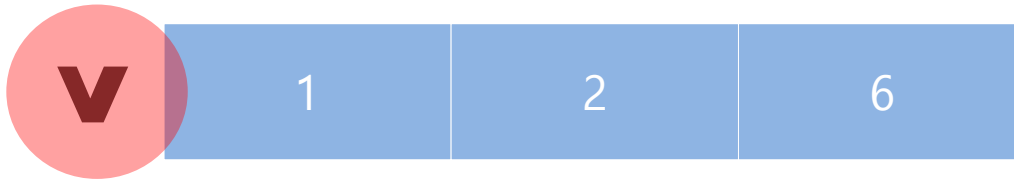
1) `vector<int> v;`

- 크기 0인 int형 1차원 벡터
- `v.push_back(a);` 가능
- `v[1] = a;` 불가능

2 vector

 : 일단 '핵'이라고..

1) `vector<int> v;` 처음엔 아무것도 없고 이름만 가지고 있음



```
v.push_back(1);
```

```
v.push_back(2);
```

```
v.push_back(6);
```

```
v[4] = 7; Error!
```

2 vector

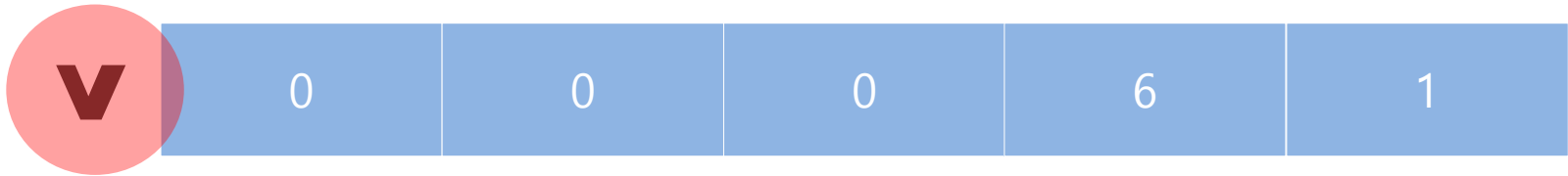
2) vector <int> v(N);

- 크기 N인 int형 1차원 벡터
- v.push_back(a); 가능
- v[1] = a; 가능(정해진 크기 안에서)
- v(N)의 괄호 안에는 변수도 사용 가능

2 vector

강괄호이므로 변수도
사용 가능

2) `vector<int> v(N);` 처음부터 크기 N짜리로 만듦
(N = 4)



`v.push_back(1);`

`v[3] = 6;`

2 vector

3) vector <int> v[9];

- 크기 9인 int형 2차원 벡터
- v[1].push_back(a); 가능
v[1][2] = a; 가능(크기가 확보됐을 때)
- v.push_back(a); 불가능
v[1] = a; 불가능
- v[9]의 대괄호 안에는 상수만 사용 가능

2 vector

대괄호[]는 변수 사용 불가능

3) `vector<int> v[3];` 크기 3짜리, 2차원 벡터 (행 3개)

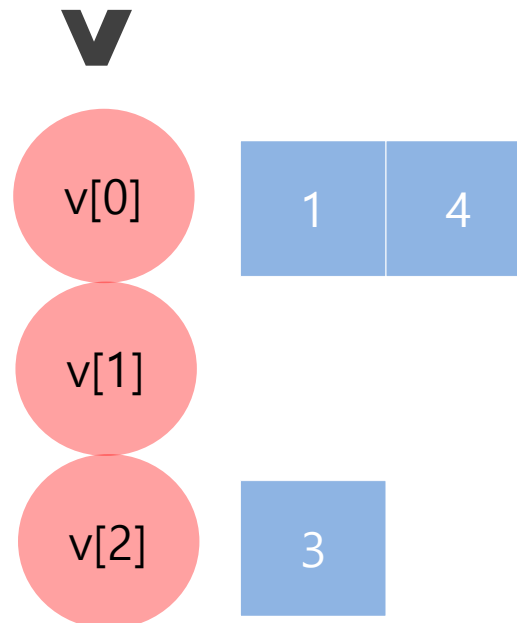
`v.push_back(1);` Error!

`v[2] = 6;` Error!

`v[0].push_back(1);`

`v[2].push_back(3);`

`v[0].push_back(4);`



2 vector

4) `vector<vector<int>> v(N);`

- 크기 N인 int형 2차원 벡터
- `v[1].push_back(a);` 가능
`v[1][2] = a;` 가능(크기가 확보됐을 때)
- `v.push_back(a);` 불가능
`v[1] = a;` 불가능
- `v(N)`의 괄호 안에는 변수도 사용 가능

2 vector

vector <int> v[9]과
vector <vector <int>> v(9)은 같다고 봐도 됨

차이점은 크기를 변수로 할수 있냐 없냐의 차이
둘 중 편한거 쓰면 됨

많이 헷갈리면 일단 이렇게만 기억
1차원으로 쓸 때 => vector <int> v(N);
2차원으로 쓸 때 => vector <int> v[N];

3. Graph Theory

3 Graph

Graph

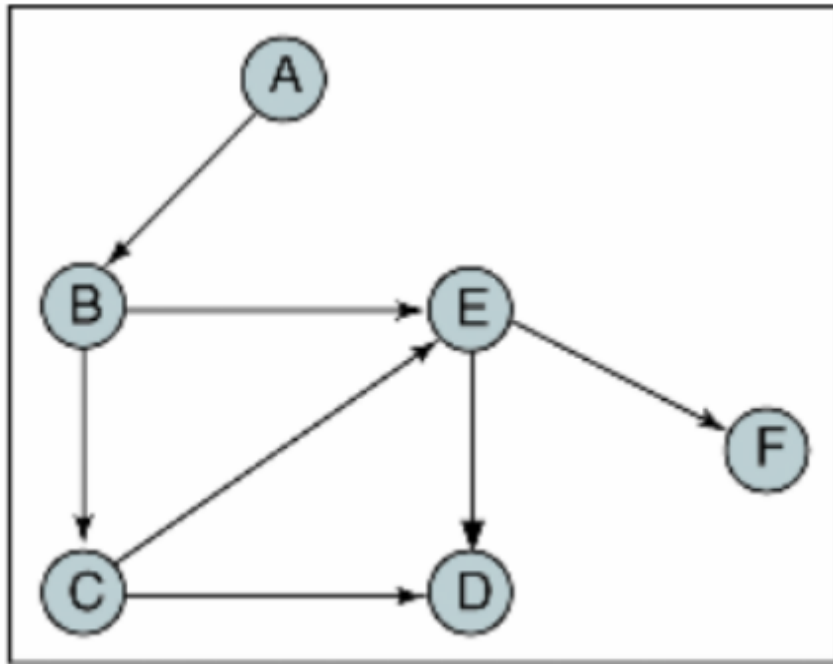
- 어떤 요소들 사이의 연결관계를 표현하기 위한 자료구조
- 노드와 노드를 연결하는 간선으로 이루어져있음
- 트리도 그래프의 일종
- 지도, 도로, 강의 커리큘럼 등 실생활에서의 수많은 것들을 그래프로 모델링할 수 있음

그래프와 트리의 차이

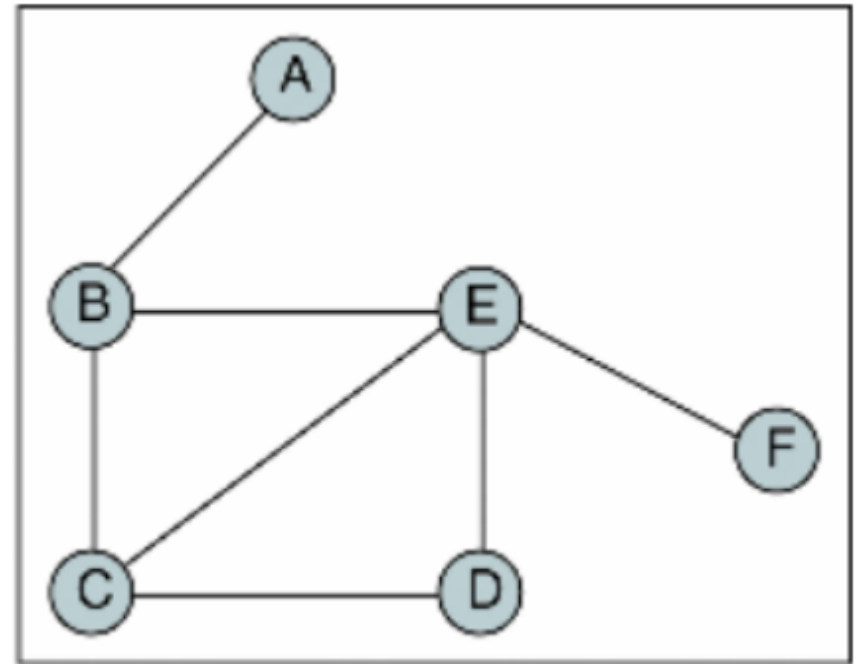
	그래프	트리
정의	노드와 간선들을 하나로 모아 놓은 자료구조	그래프의 한 종류
방향성	방향 그래프(Directed), 무방향 그래프(Undirected) 모두 존재	방향 그래프(Directed)
사이클	사이클(Cycle) 가능, 자체 간선(self-loop) 가능, 순환(Cyclic), 비순환(Acyclic) 모두 존재	사이클(Cycle) 불가능, 자체 간선(self-loop) 불가능, 비순환(Acyclic) 그래프
루트 노드	루트 노드의 개념이 없음	한 개의 루트 노드만 존재, 모든 자식은 하나의 부모 노드만 가짐
부모-자식	부모-자식의 개념이 없음	부모-자식 관계로 이루어짐
순회	DFS, BFS	DFS(pre,in,post order), BFS
간선의 수	그래프마다 간선의 수가 다름 (없을 수도 있음)	노드가 N개면 간선은 항상 N-1개
경로	두 노드 사이에도 경로가 다양하게 나올 수 있음	임의의 두 노드 간의 경로는 유일함

3 Graph

방향 그래프, 무방향 그래프



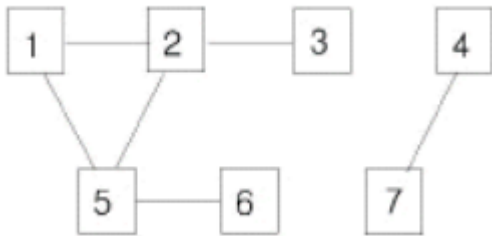
(a) Directed graph



(b) Undirected graph

3 Graph

그래프는 모두 연결되어있지 않을 수도 있음!



각각의 그룹을 “연결 요소” 라고 부름
위의 그림은 두 개의 연결요소로 이루어진 하나의 그래프

3 Graph

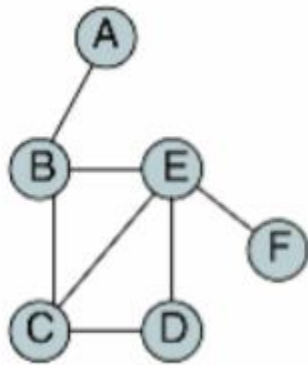
그래프의 표현

그래프를 코드로 표현하는 방법은 크게 두가지

1. 인접 행렬(이차원 배열)
2. 인접 리스트(이차원 벡터)
3. 간선 리스트(나중에)

1. 인접 행렬(이차원 배열)

모든 두 정점간의 연결관계를 이차원 배열로 표현



A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency matrix

장점 : 구현이 간단함

두 노드가 연결되어있는지 알고 싶을 때 $O(1)$ 에 가능

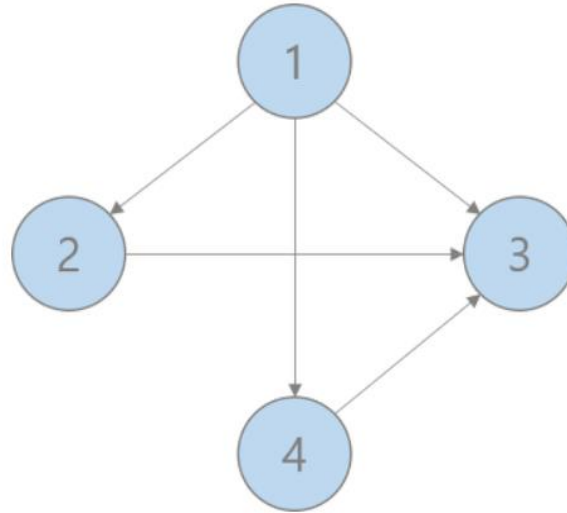
단점 : 공간을 많이 차지

한 노드와 연결된 모든 노드를 알고 싶으면 $O(V)$

모든 노드의 연결 관계를 탐색하려면 $O(V^2)$

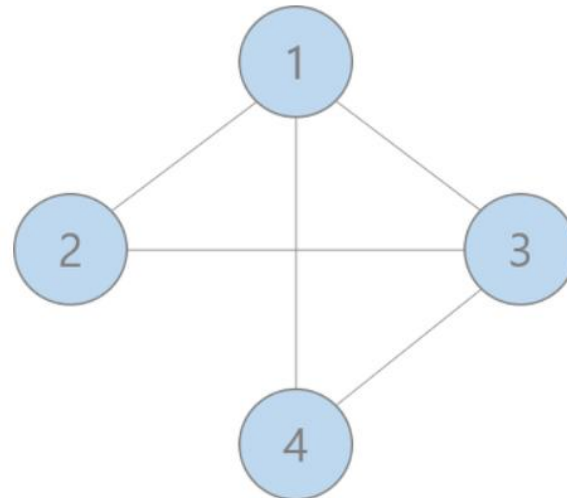
1. 인접 행렬(이차원 배열)

방향 그래프



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

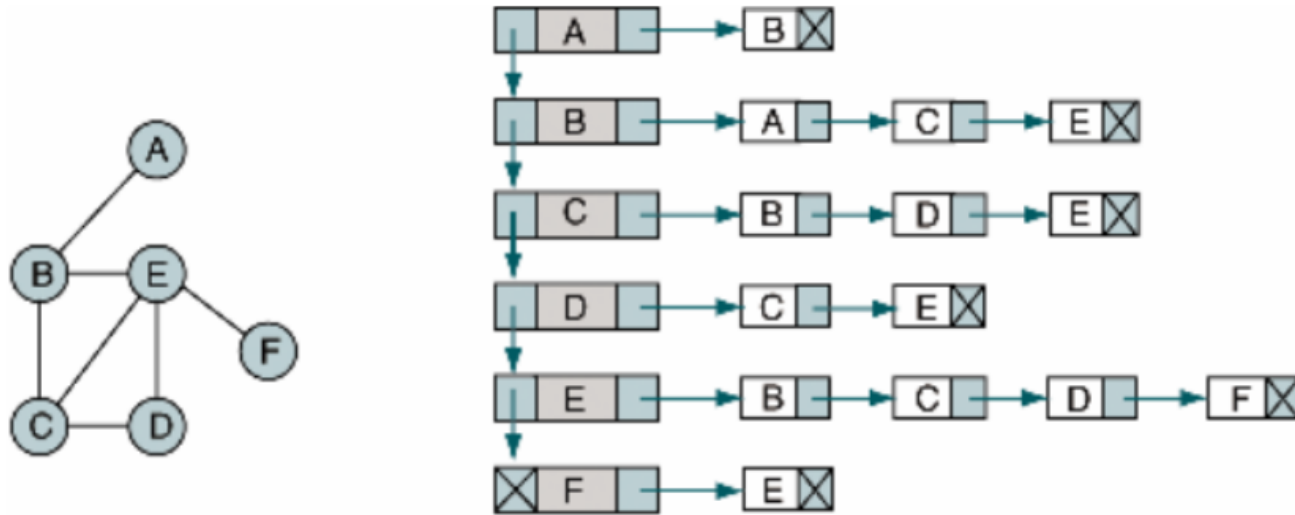
무방향 그래프



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

2. 인접 리스트(이차원 벡터)

실제로 연결된 노드들에 대한 정보만 저장

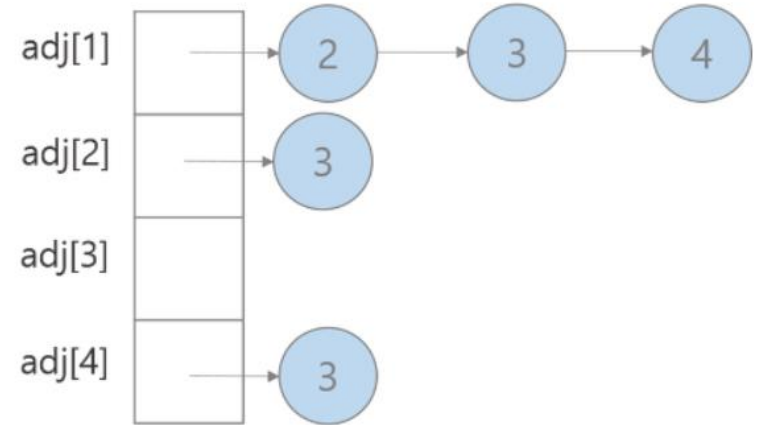
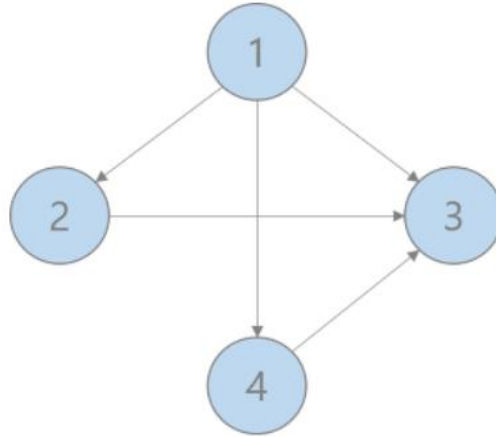


장점 : 간선의 개수만큼만 메모리 차지
모든 노드의 연결 관계를 $O(V)$ 에 탐색 가능

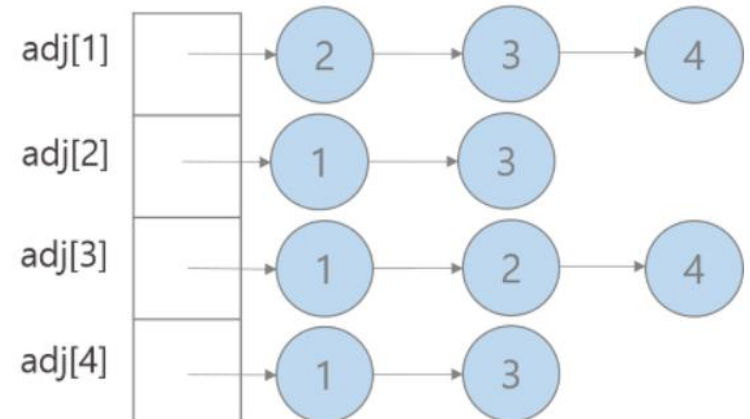
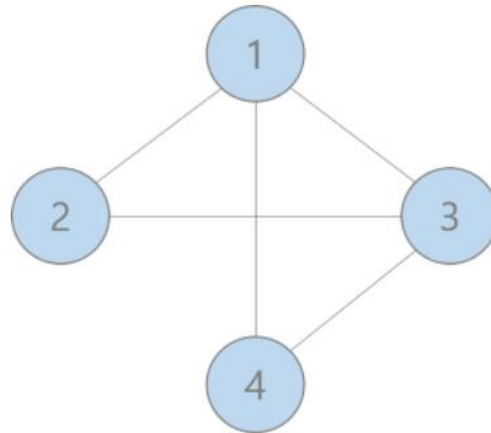
단점 : 어떤 두 노드가 연결되어있는지 찾으려면 $O(V)$

2. 인접 리스트(이차원 벡터)

방향 그래프



무방향 그래프



3 Graph

DFS, BFS 실습

1260_DFS와 BFS

참고 : <http://wanna-b.tistory.com/64>



감사합니다.

Made by 규정