Graph
&
Tree

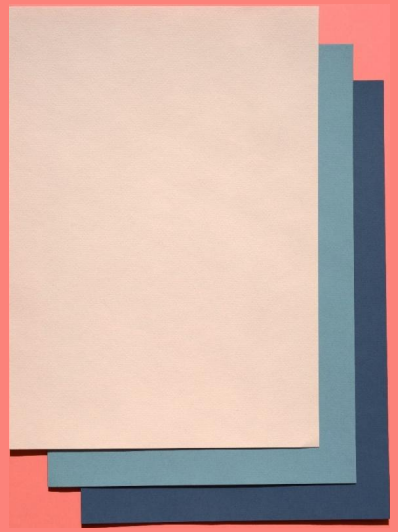
오늘의 학습 목표

01 그래프의 개념과 용어

02 트리의 개념(그래프와 트리의 차이)

03 이진트리의 개념과 순회

04 그 외의 트리(RB트리, AVL 트리)



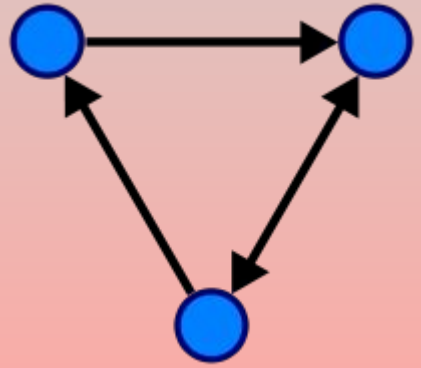
그래프란?

그래프(Graph)

- 그래프는 정점(vertex, node)과 간선(edge)로 구성된 자료구조이다.

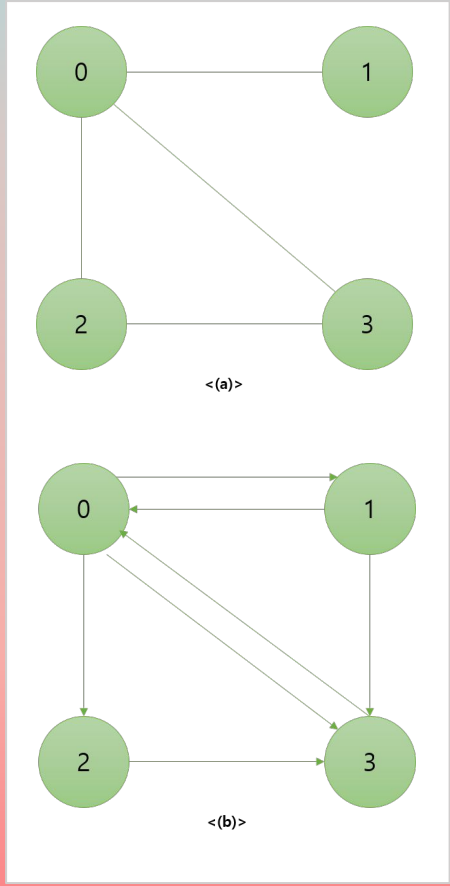
- 비선형 자료구조 이다.

  < – > 선형 자료구조(배열, 리스트, 스택, 큐, …)



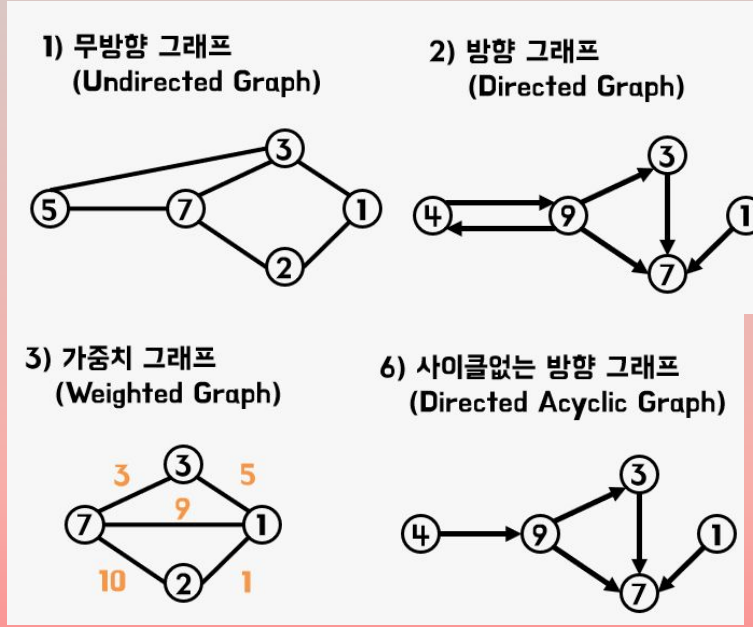
용어

- 정점(vertex 또는 node) : 위치의 개념
- 간선(edge) : 정점을 연결하는 선
- 인접 정점 : 간선에 의해 직접 연결된 정점
- 차수(degree) : 무방향 그래프에서 하나의 정점에 인접한 정점의 수
- 진입 차수(in-degree) : 방향 그래프에서 외부에서 오는 간선의 수
- 진출 차수(out-degree) : 방향 그래프에서 외부로 향하는 간선의 수
- 단순 경로 : 경로 중에서 반복되는 정점이 없는 경우
- 순환(cycle) : 단순 경로의 시작 정점과 종료 정점이 동일한 경우



그래프의 특징

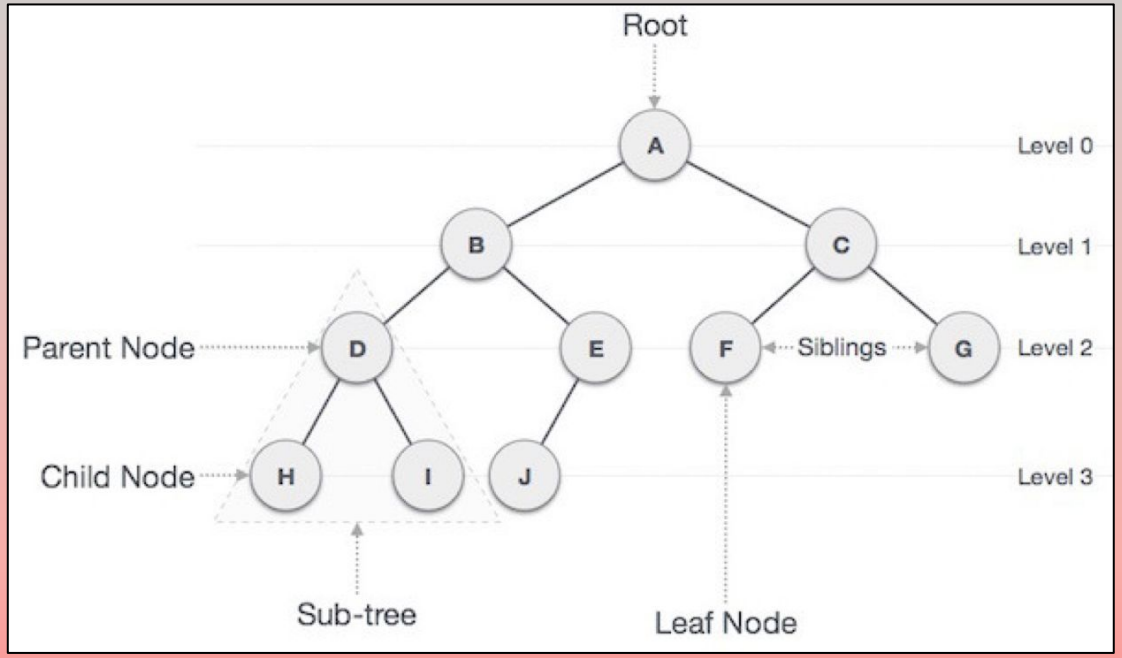
- 그래프는 네트워크 모델이다. <-> 계층 모델
- 노드들 사이에 무방향, 방향, 양방향 경로를 가질 수 있다.
- 부모 - 자식 관계라는 개념이 없다.
- 그래프 순회(탐색)은 **DFS**나 **BFS**로 이루어진다.
- 그래프는 순환(Cyclic) 혹은 비순환(Acyclic)이다.
- 그래프는 크게 방향 그래프와 무방향 그래프가 있다.
- 간선에 비용이나, 값이 할당된 그래프를
  가중치 그래프(**Weighted Graph**)라고 한다.



트리(Tree)

	그래프	트리
정의	노드(node)와 그 노드를 연결하는 간선(edge)을 하나로 모아 놓은 자료 구조	그래프의 한 종류 DAG(Directed Acyclic Graph, 방향성이 있는 비순환 그래프)의 한 종류
방향성	방향 그래프(Directed), 무방향 그래프(Undirected) 모두 존재	방향 그래프(Directed Graph)
사이클	사이클(Cycle) 가능, 자체 간선(self-loop)도 가능, 순환 그래프(Cyclic), 비순환 그래프(Acyclic) 모두 존재	사이클(Cycle) 불가능, 자체 간선(self-loop)도 불가능, 비순환 그래프(Acyclic Graph)
루트 노드	루트 노드의 개념이 없음	한 개의 루트 노드만이 존재, 모든 자식 노드는 한 개의 부모 노드 만을 가짐
부모-자식	부모-자식의 개념이 없음	부모-자식 관계 top-bottom 또는 bottom-top으로 이루어짐
모델	네트워크 모델	계층 모델
순회	DFS, BFS	DFS, BFS안의 Pre-, In-, Post-order
간선의 수	그래프에 따라 간선의 수가 다름, 간선이 없을 수도 있음	노드가 N인 트리는 항상 N-1의 간선을 가짐
경로	-	임의의 두 노드 간의 경로는 유일
예시 및 종류	지도, 지하철 노선도의 최단 경로, 전기 회로의 소자들, 도로(교차점과 일방 통행길), 선수 과목	이진 트리, 이진 탐색 트리, 균형 트리(AVL 트리, red-black 트리), 이진 힙(최대힙, 최소힙) 등

트리(Tree)

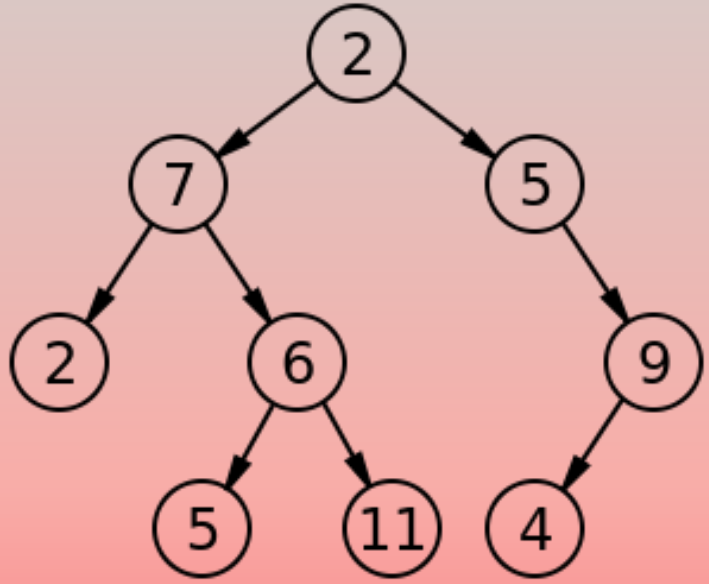


- 트리에서의 차수(**degree**)는 자식의 개수를 의미한다.
- 트리의 높이는 루트 노드에서 가장 깊숙히 있는 노드의 깊이

이진 트리(Binary Tree)

- 각 노드가 최대 두개의 자식을 갖는 트리

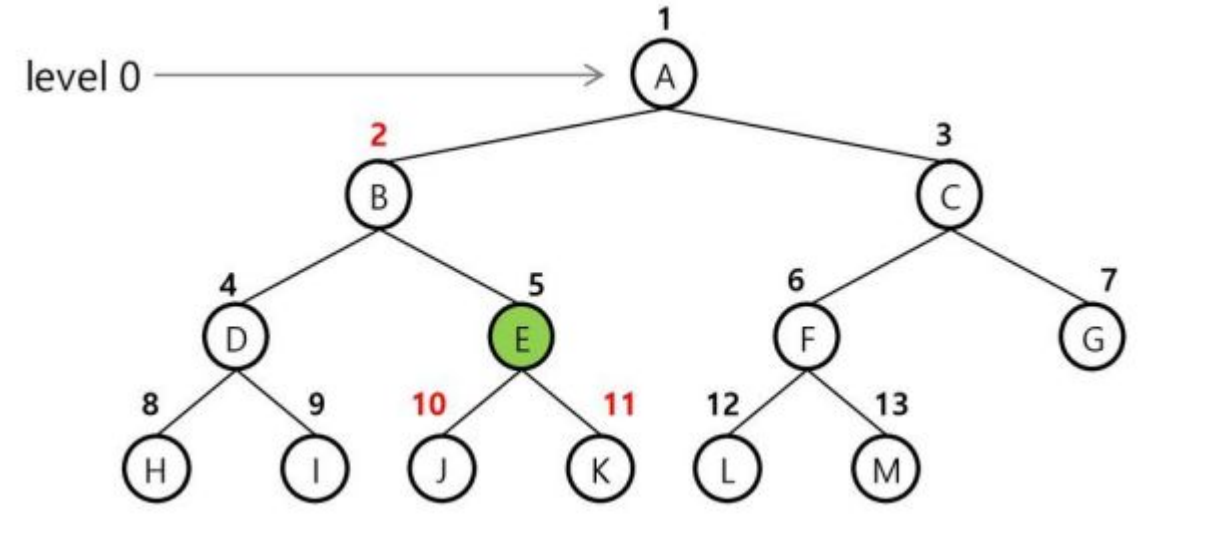
- 높이가 **h**인 이진 트리가 가질 수 있는 노드의
  최소 개수는 **h+1** 개, 최대 개수는 **2^(h+1) -1** 개가 된다.



- 노드 번호의 성질
- 노드 번호가 **i** 인 노드의 부모 노드 번호 **-> i / 2**
- 노드 번호가 **i** 인 노드의 왼쪽 자식 노드 번호 **-> 2 * i**
- 노드 번호가 **i** 인 노드의 오른쪽 자식 노드 번호 **-? 2 * i + 1**

이진 트리(Binary Tree)

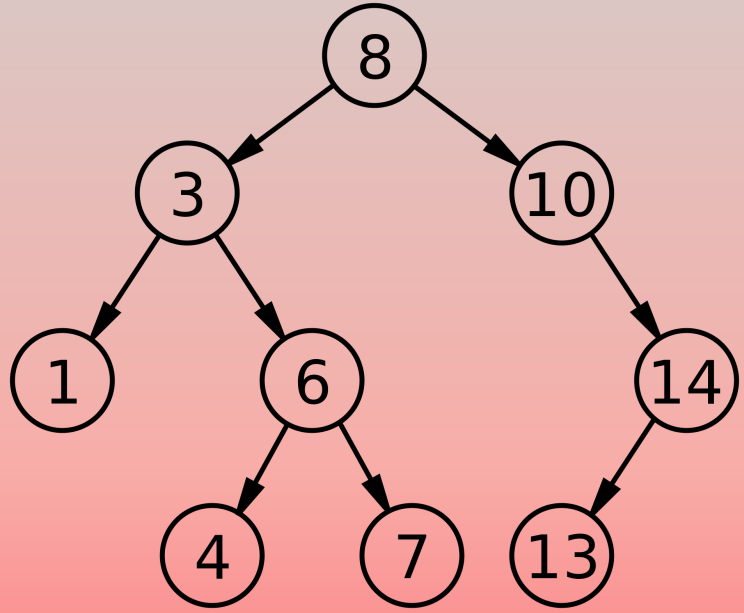
- 노드 번호의 성질
- 노드 번호가 i 인 노드의 부모 노드 번호 -> i / 2
- 노드 번호가 i 인 노드의 왼쪽 자식 노드 번호 -> 2 * i
- 노드 번호가 i 인 노드의 오른쪽 자식 노드 번호 -> 2 * i + 1
- 높이가 h 인 이진 트리를 위한 배열의 크기 -> 2^(h+1)



이진 탐색 트리(Binary Search Tree)

- 다음과 같은 속성을 가지는 이진 트리

1. 모든 왼쪽 자식들 < = 부모
2. 모든 오른쪽 자식들 > 부모

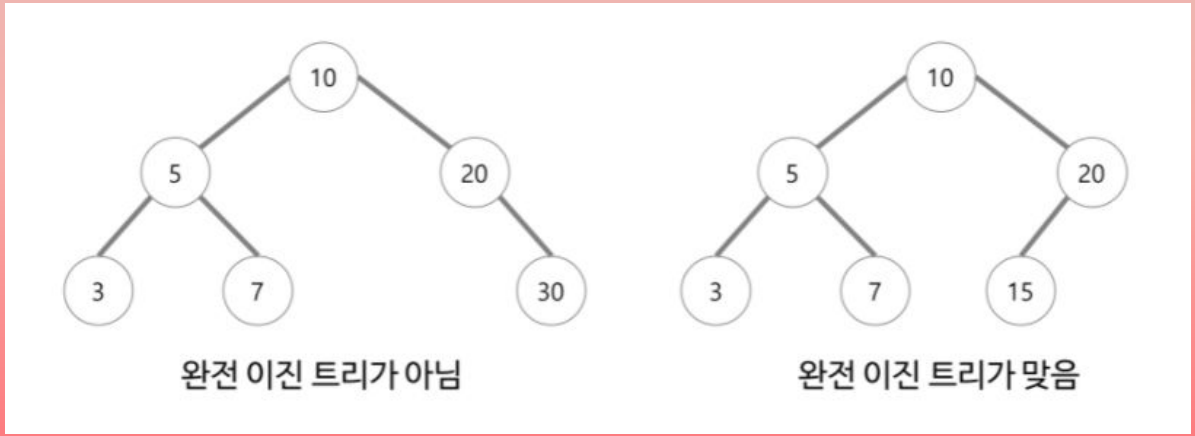


즉, 부모의 모든 왼쪽 자식들은 부모 보다 작거나 같아야 하며,
부모의 모든 오른쪽 자식들은 부모보다 커야 한다.

완전 이진 트리, 진 이진 트리, 포화 이진 트리

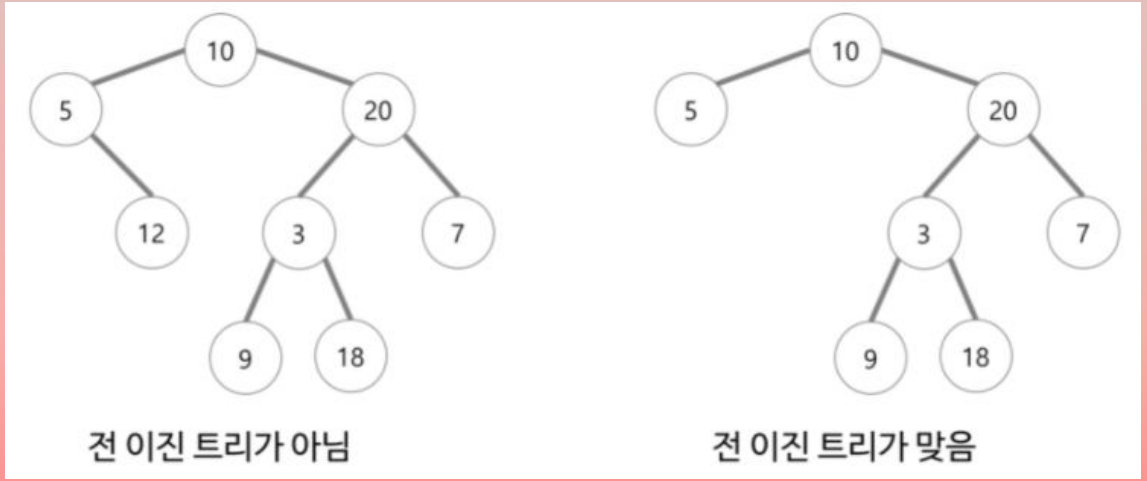
- 트리의 모든 높이에서 노드가 꽉 차 있는 이진 트리.
  즉, 마지막 레벨을 제외하고 모든 레벨이 완전히 채워져 있다.
- 마지막 레벨은 꽉 차 있지 않아도 되지만 노드가 왼쪽에서 오른쪽으로 채워져야 한다.

- 완전 이진 트리는 배열을 사용해 효율적으로 표현 가능하다.



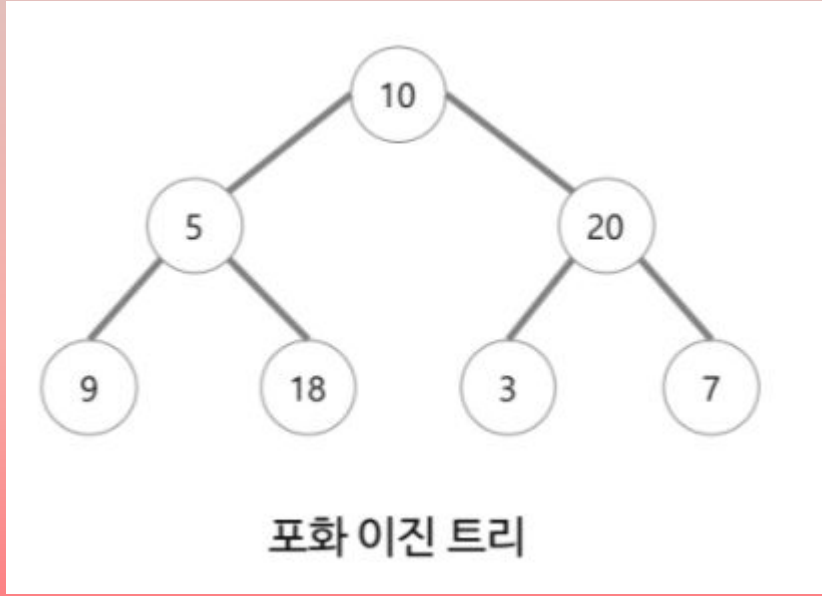
완전 이진 트리, ***진 이진 트리***, 포화 이진 트리

- **모든 노드가 0개 또는 2개의 자식 노드를 갖는 트리**



완전 이진 트리, 진 이진 트리, ***포화 이진 트리***

- 진 이진 트리이면서 완전 트리인 경우
- 노드의 개수가 정확히 **2^(k+1) -1** 개여야 한다. **(k**는 트리의 높이**)**



포화 이진 트리

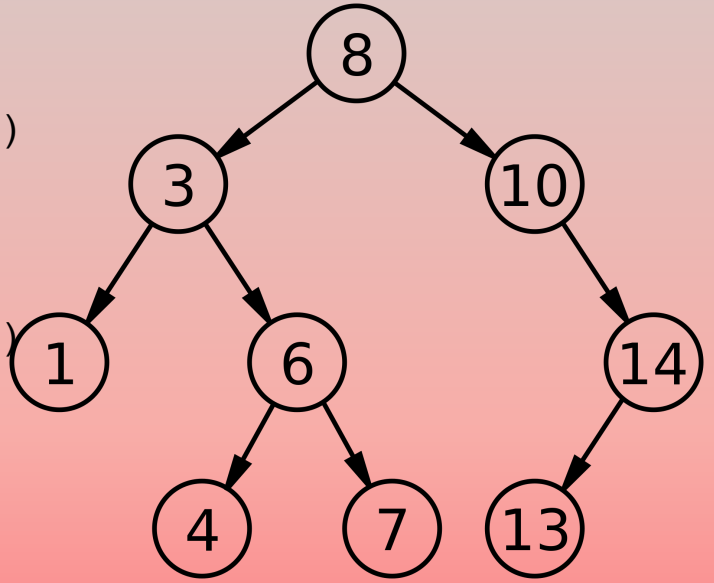
이진 트리의 순회

- **이진 트리에는 3가지의 순회 종류가 존재**

1. 전위 순회 (Preorder)
: 부모 -> 왼쪽 자식 노드(서브트리) -> 오른쪽 자식 노드(서브트리)
8 3 1 6 3 7 10 14 13

2. 중위 순회 (Inorder)
: 왼쪽 자식 노드(서브트리) **-> 부모** -> 오른쪽 자식 노드(서브트리)
1 3 4 6 7 8 10 13 14

3. 후위 순회 (Postorder)
: 왼쪽 자식 노드(서브트리) ->오른쪽 자식 노드(서브트리) -> **부모**
1 4 7 6 3 13 14 10 8



이진 트리의 순회 코드 구현

https://www.notion.so/12b0b14464234b98a5aded6e8dc43d1c

```
public class Node {
	private int data; //노드의 값
	private Node leftNode; //왼쪽 자식노드의 값
	private Node rightNode; //오른쪽 자식노드의 값
	public Node(int data, Node leftNode, Node rightNode) {
		this.data = data;
		this.leftNode = leftNode;
		this.rightNode = rightNode;
	}

	public void setLeftNode(Node node) {this.leftNode = node;}
	public void setRightNode(Node node) { this.rightNode = node;}
	public Node getLeftNode() { return this.leftNode;}
	public Node getRightNode() { return this.rightNode; }
}
```

이진 트리의 순회 코드 구현

https://www.notion.so/12b0b14464234b98a5aded6e8dc43d1c

```
    //전위
  public void preOrder(Node node) {
        if(node !=null) {
              System.out.print(node.getData() + " ");
              preOrder(node.getLeftNode());
              preOrder(node.getRightNode());
        }
}
```

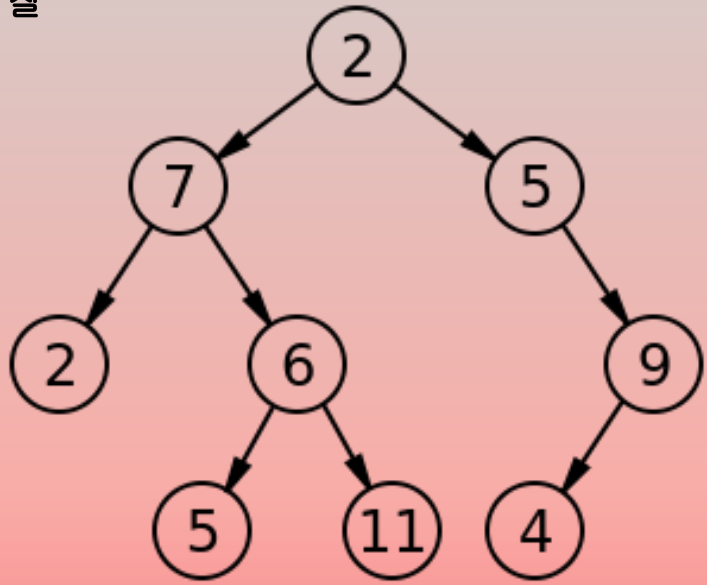
```
    //후위
  public void preOrder(Node node) {
        if(node !=null) {
              preOrder(node.getLeftNode());
              preOrder(node.getRightNode());
              System.out.print(node.getData() + " ");
        }
}
```

```
    //중위
  public void preOrder(Node node) {
        if(node !=null) {
              preOrder(node.getLeftNode());
              System.out.print(node.getData() + " ");
              preOrder(node.getRightNode());
        }
}
```

균형 트리 VS 비균형 트리

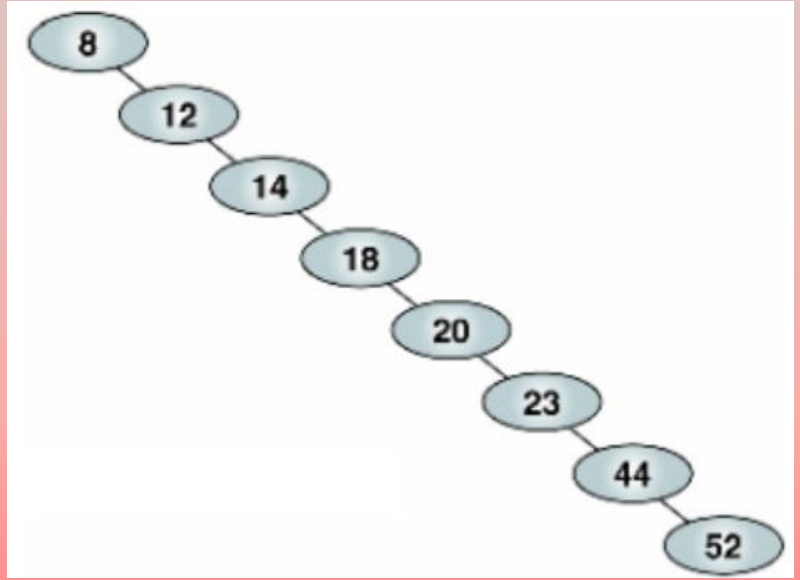
- **O(logN)** 시간에 **insert**와 **find**를 할 수 있을 정도로 균형이 잘 잡혀 있는 경우

- **ex)** 레드-블랙 트리, **AVL** 트리



AVL Tree

- 이진 검색 트리에서는 다음과 같은 문제점이 발생한다.



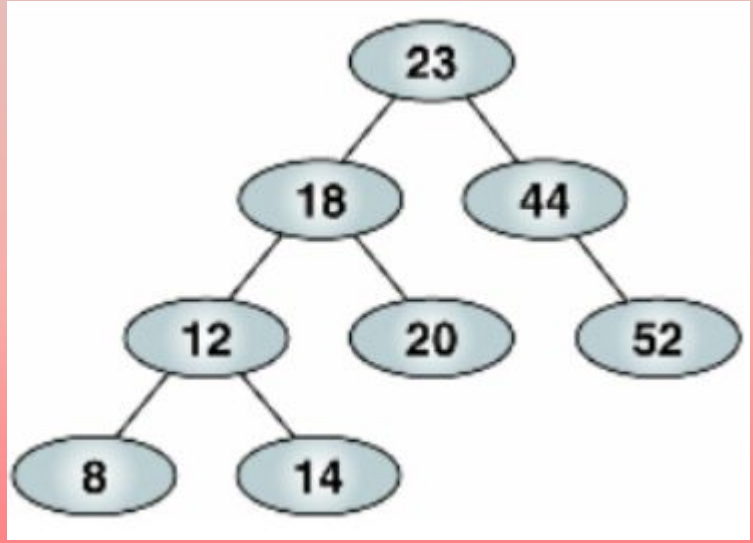
- 한 쪽으로 쏠려있어서 원하는 자료를 찾으려면 끝까지 탐색을 해야 한다.

AVL Tree

- 이진 탐색 트리이면서 동시에 균형을 유지하는 트리, 균형을 유지하기 때문에 이진 탐색 시의 효율성을 보장할 수 있다.
- 모든 노드의 왼쪽과 오른쪽 서브트리의 높이 차가 1 이하이다.

AVL 트리의 시간복잡도

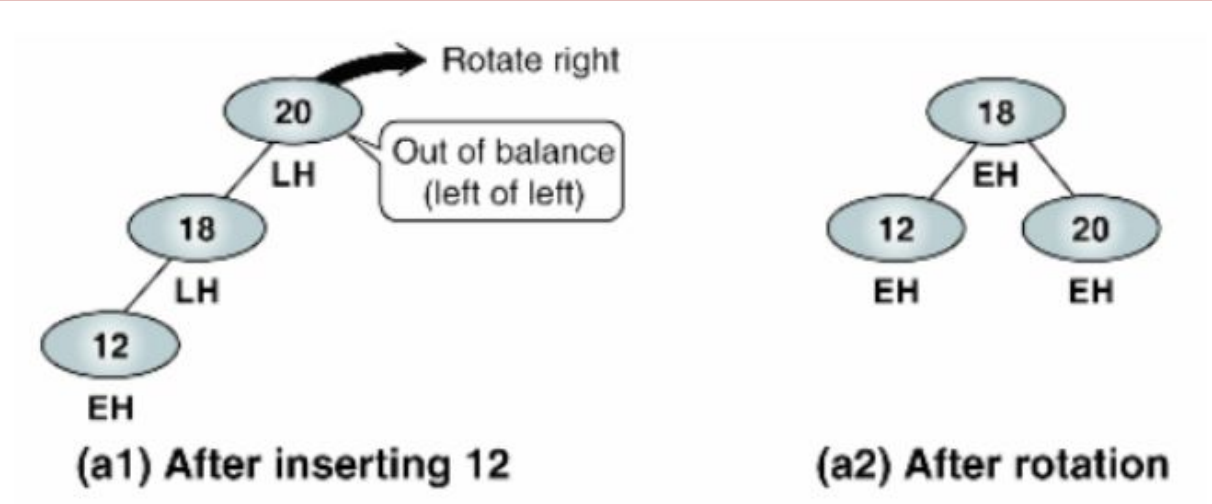
	평균	최악의 경우
공간	$O(n)$	$O(n)$
검색	$O(\log n)$	$O(\log n)$
삽입	$O(\log n)$	$O(\log n)$
삭제	$O(\log n)$	$O(\log n)$



AVL Tree

- 균형이 무너지는 유형에는 4가지가 있다. **LL, LR, RL, RR**

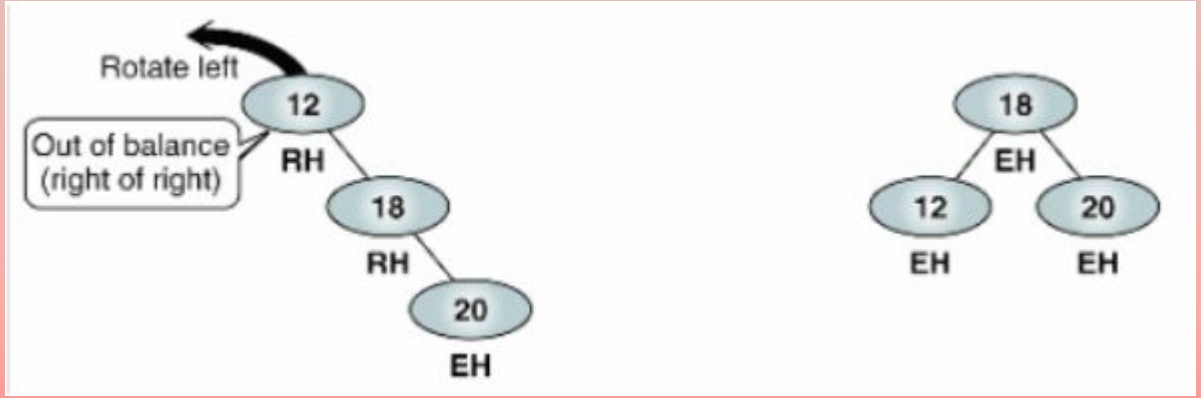
LL



AVL Tree

- 균형이 무너지는 유형에는 4가지가 있다. **LL, LR, RL, RR**

RR

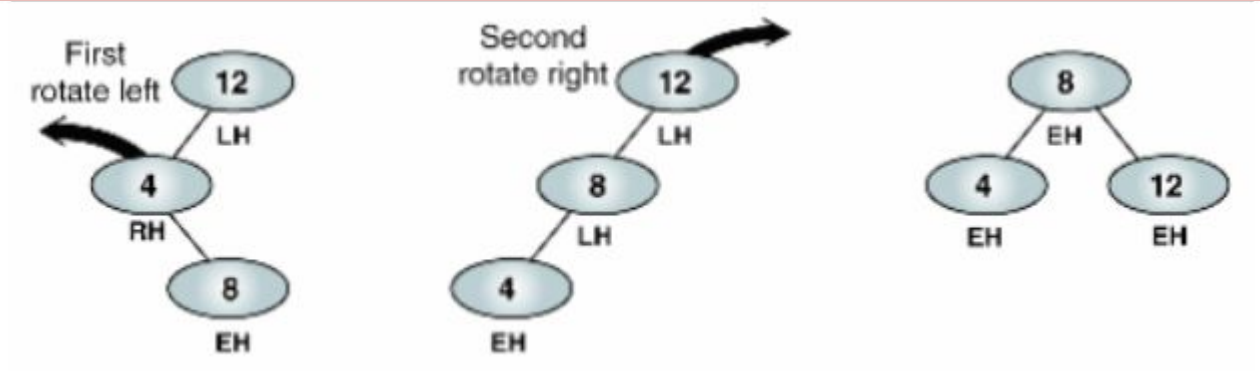


AVL Tree

https://www.zerocho.com/category/Algorithm/post/583cacb648a7340018ac73f1

- 균형이 무너지는 유형에는 4가지가 있다. **LL, LR, RL, RR**

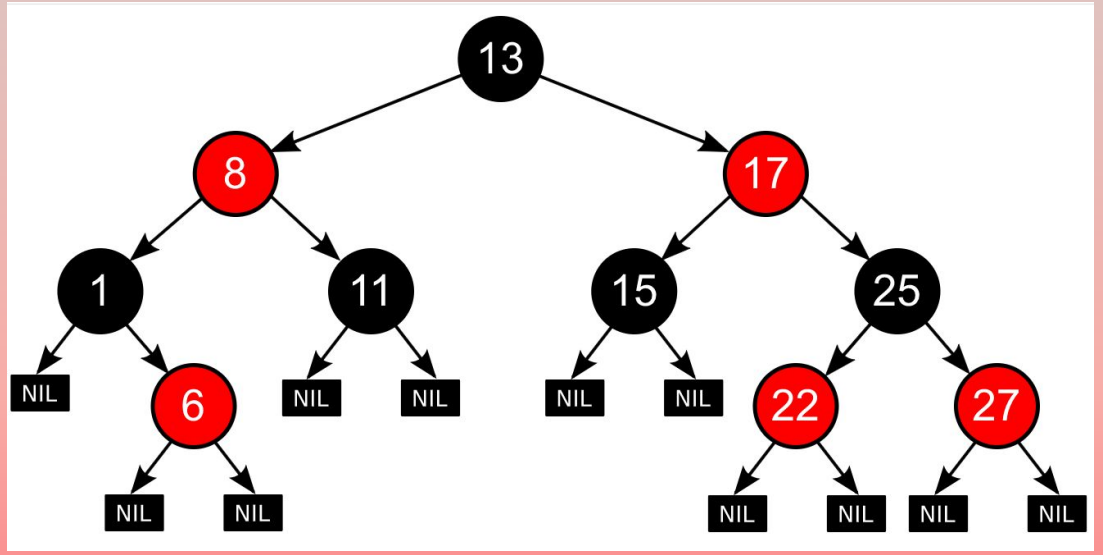
LR



Red-Black Tree

https://ko.wikipedia.org/wiki/%EB%A0%88%EB%93%9C-%EB%B8%94%EB%9E%99_%ED%8A%B8%EB%A6%AC

- 레드-블랙 트리는 각각의 노드가 레드나 블랙 인 색상 속성을 가지고 있는 이진 탐색 트리이다.



Red-Black Tree

https://ko.wikipedia.org/wiki/%EB%A0%88%EB%93%9C-%EB%B8%94%EB%9E%99_%ED%8A%B8%EB%A6%AC

1. 노드는 레드 혹은 블랙 중의 하나이다.
2. 루트 노드는 블랙이다.
3. 모든 리프 노드들(NIL)은 블랙이다.
4. 레드 노드의 자식노드 양쪽은 언제나 모두 블랙이다.
   (즉, 레드 노드는 연달아 나타날 수 없으며, 블랙 노드만이 레드 노드의 부모 노드가 될 수 있다)
5. 어떤 노드로부터 시작되어 그에 속한 하위 리프 노드에 도달하는 모든 경로에는 리프 노드를 제외하면 모두 같은 개수의 블랙 노드가 있다.

- AVL 트리는 레드-블랙 트리보다 더 엄격하게 균형이 잡혀 있기 때문에, 삽입과 삭제를 할 때 최악의 경우에는 더 많은 회전(rotations)이 필요하다.

Red-Black Tree

https://ko.wikipedia.org/wiki/%EB%A0%88%EB%93%9C-%EB%B8%94%EB%9E%99_%ED%8A%B8%EB%A6%AC

- 위 조건들을 만족하게 되면, 레드-블랙 트리는 가장 중요한 특성을 나타내게 된다: 루트 노드부터 가장 먼 잎노드 경로까지의 거리가, 가장 가까운 잎노드 경로까지의 거리의 두 배 보다 항상 작다. 다시 말해서 레드-블랙 트리는 개략적(roughly)으로 균형이 잡혀 있다(balanced). 따라서, 삽입, 삭제, 검색시 최악의 경우(worst-case)에서의 시간복잡도가 트리의 높이(또는 깊이)에 따라 결정되기 때문에 보통의 이진 탐색 트리에 비해 효율적이라고 할 수 있다.

- 왜 이런 특성을 가지는지 설명하기 위해서는, 네 번째 속성에 따라서, 어떤 경로에도 레드 노드가 연이어 나타날 수 없다는 것만 알고 있어도 충분하다. 최단 경로는 모두 블랙 노드로만 구성되어 있다고 했을 때, 최장 경로는 블랙 노드와 레드 노드가 번갈아 나오는 것이 될 것이다. 다섯 번째 속성에 따라서, 모든 경로에서 블랙 노드의 수가 같다고 했기 때문에 존재하는 모든 경로에 대해 최장 경로의 거리는 최단 경로의 거리의 두 배 이상이 될 수 없다.

Thank You