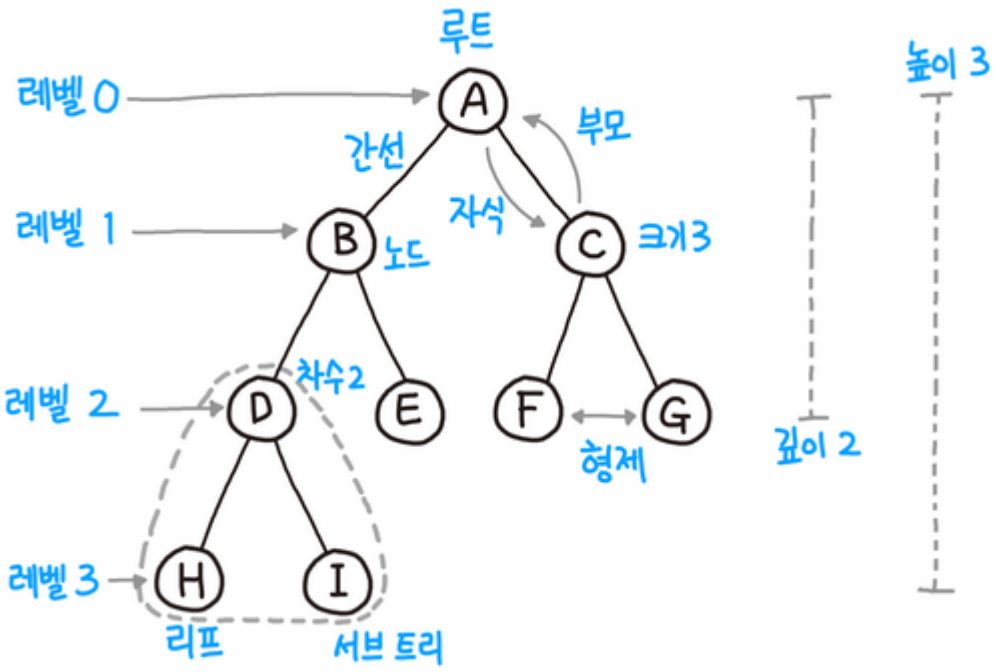
Tree

개요

- 비선형 자료구조로 계층적 구조를 나타냄
- 노드와 간선으로 이루어져 있으며, 최상위 노드는 루트(Root) 노드
- 각 노드는 하위 노드와 부모 노드를 가지며, 부모-자식 관계로 표현됨
- 노드는 자식 노드의 수에 따라 말단 노드(leaf node)와 비말단 노드(internal node)로 구분됨. (말단 노드는 자식이 없는 노드)
- 트리는 데이터 검색, 삽입, 삭제 등에 매우 유용함

용어



이진 트리

- 트리의 가장 간단한 형태
- 자식 노드의 개수(차수)를 최대 2개로 제한
- 두 자식 노드를 왼쪽 자식과 오른쪽 자식으로 구분
- 1개의 값과 왼쪽, 오른쪽 자식 노드를 각각 가리킬 2개의 포인터를 가진 구조로 구현 가능

- 일반적인 트리 구조(자식 노드의 개수가 2개 초과)에서 초과한 자식 하나마다 노드를 1개 추가하고, 새 노드의 왼쪽에 원래의 자식 노드, 오른쪽에 형제 노드를 배치해서 이진 트리로 변환 가능하므로 보통 트리는 이진 트리로 구현

이진 트리 종류

- 포화 이진 트리
  - 모든 리프 노드의 높이가 같고 리프 노드가 아닌 노드는 모두 2개의 자식을 갖는 이진 트리
  - 이진트리의 가장 깊은 리프의 높이가 N일 때, 노드 수의 최대치는 2^N-1개인데, 포화 이진 트리가 그 최대치임
- 완전 이진 트리
  - 모든 리프 노드의 높이가 최대 1 차이가 나고, 마지막 레벨을 제외하고 노드가 모두 채워져 있는 트리 (마지막 레벨도 채워져 있을 수 있음)
  - 오른쪽 자식이 있는 모든 노드는 왼쪽 자식이 있어야 함
  - 왼쪽에서 오른쪽으로 빠짐 없이 채워나가는 형태
  - 완전 이진 트리의 부분집합에 포화 이진 트리도 속함
    - N개의 노드를 가진 완전 이진 트리의 높이는 logN
  - 완전 이진 트리의 경우 왼쪽부터 빠짐없이 채워져 있다는 성질을 이용하여 배열로 구현할 수 있음
    - index 1번부터 시작하는 배열로, N번째 원소의 왼쪽 자식은 2N, 오른쪽 자식은 2N+1로 구성하면 됨.

완전 이진 트리 구현

1. Node 클래스
   - Node 클래스는 링크드리스트의 노드처럼 자체적으로 데이터 값 저장하고, 각 부모 / 자식 노드의 정보도 갖고 있음

```
class Node<T>{
    T value;
    Node left;
    Node right;

    public Node(T value){
```

```
        this.value = value;
    }
}
```

2. 이진 트리 클래스
    - 전체 이진 트리에 대한 정보를 갖는 클래스
    - 현재 루트가 무엇인지, 저장된 노드의 수가 몇 개인지 등의 정보를 갖고 있음

```
public class BinaryTree<T> {
    Node<T> root = null;
    int size = 0;

    public BinaryTree(){}
}
```

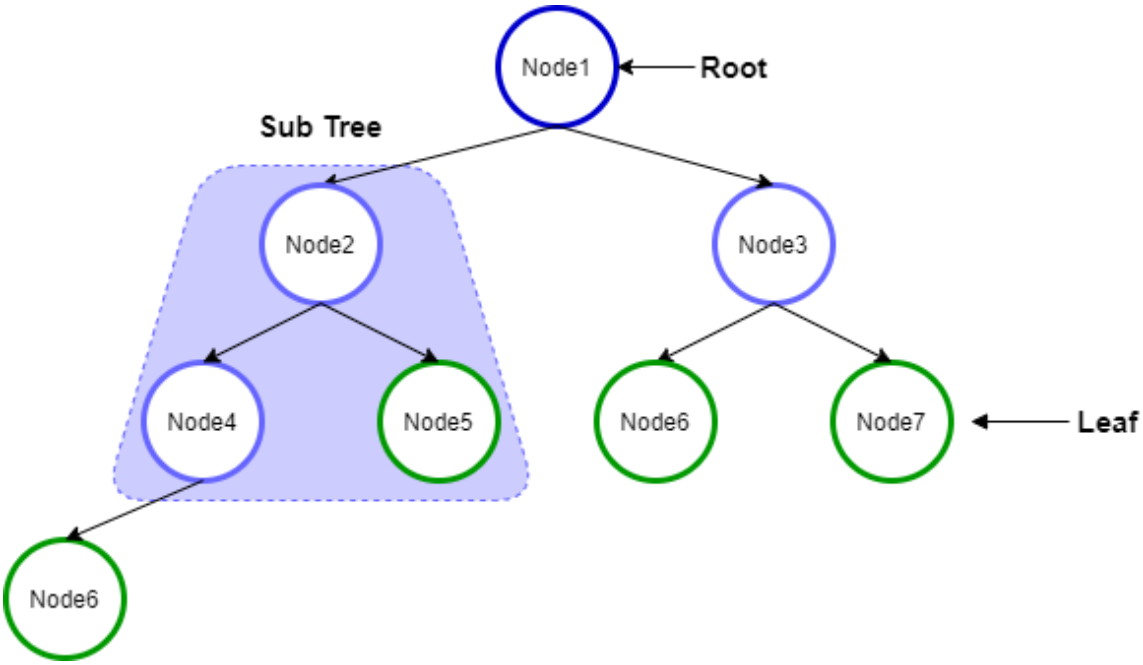
3. 데이터 노드 삽입
    - 데이터 삽입은 처음에 루트가 비어 있다면 루트에 삽입하고, 그 이후에는 완전 이진 트리이므로 루트의 왼쪽부터 차곡차곡 삽입
    - 레벨 순회 방식을 알아야 함

```
public boolean insert(T value){
        Node newNode = new Node(value);
        if(size == 0){
            root = newNode;
            return true;
        }
        Queue<Node<T>> q = new LinkedList<>();
        q.add(root);

        while(true){
            Node tmp = q.peek();
            if(tmp.left == null){
                tmp.left = newNode;
                newNode.parent=tmp;
                break;
            } else {
                q.add(tmp.left);
            }

            if(tmp.right == null){
                tmp.right = newNode;
                newNode.parent=tmp;
                break;
            } else {
                q.add(tmp.right);
            }
        }
```

```
        return true;
    }
```

이진 트리 순회 (boj 1991)

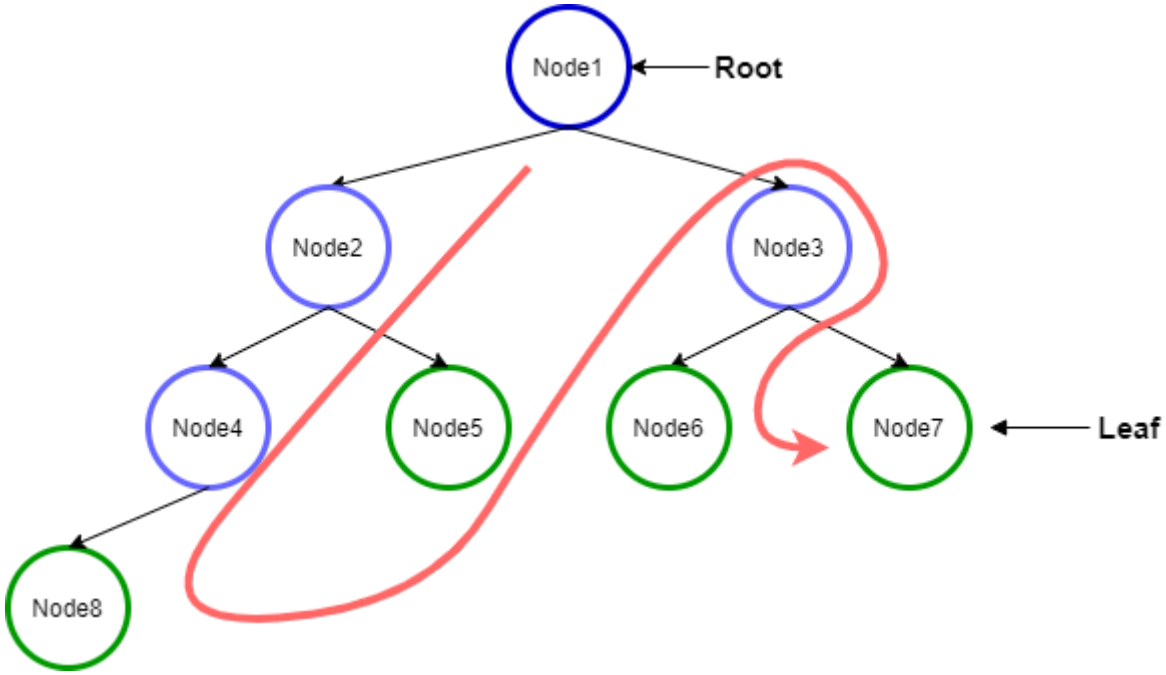
- 선형 구조(배열, 링크드리스트 등)에서는 순차적으로 데이터를 꺼내오기만 하면 됨
  - ex) 배열에선 index 0번부터 length-1까지 순차적으로 접근, 링크드리스트에선 노드가 선형적으로 연결돼있으므로, 다음 노드를 순차적으로 방문
- 트리는 하나의 노드가 복수의 다른 노드와 연결점을 갖는 비선형 구조
  - 동일한 깊이에 있는 노드라도, 어떤 데이터는 먼저 순회되고, 다른 데이터는 나중에 순회됨
- 트리의 구조는 재귀적이라는 특성도 있음
  - 큰 구조 안에 동일한 형태의 작은 구조로 나뉘어져 있음



  - 이 특성을 이용하여 전체 트리를 순회할 때, DFS로 쉽게 순회를 할 수 있음

전위 순회

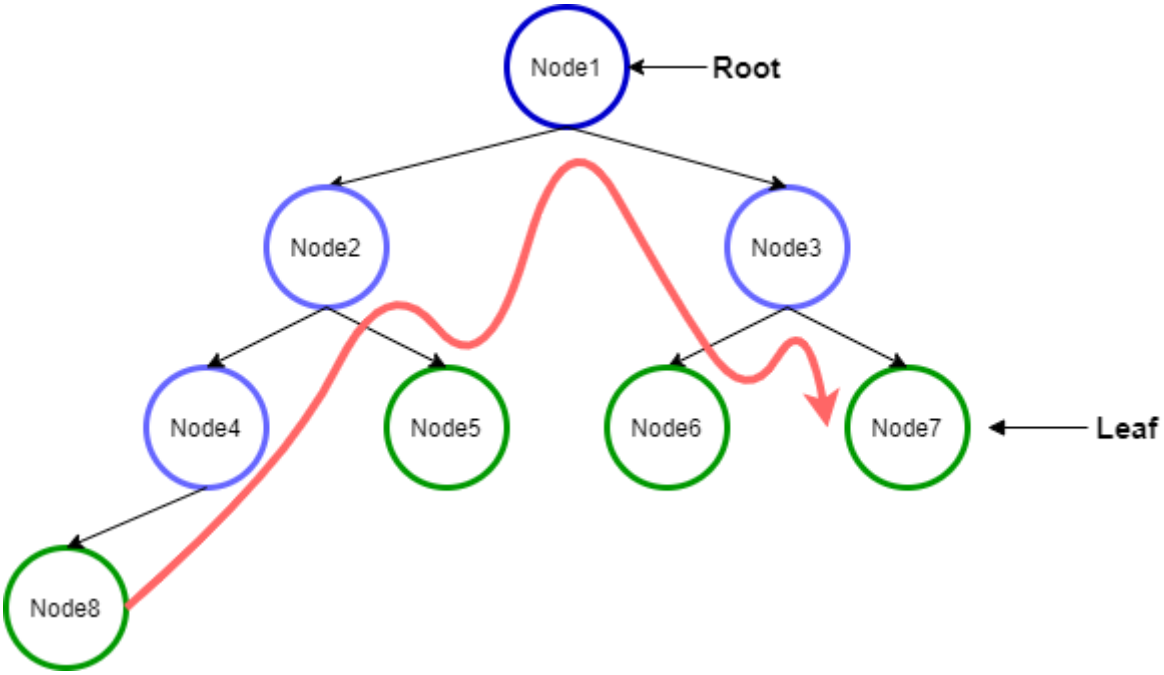
- 현재 자기 자신 노드를 제일 먼저 순회하고, Left → Right 순서로 순회
  - 아래 그림에서 1 → 2 → 4 → 8 → 5 → 3 → 6 → 7




```
// 현재 parameter로 전달된 노드가 비어있지 않다면
// 해당 노드의 값을 출력하고 left -> right 노드를 순회
public void preOrder(Node node){
        if(node != null){
            System.out.print(node.value);
            preOrder(node.left);
            preOrder(node.right);
        }
    }
```

중위 순회

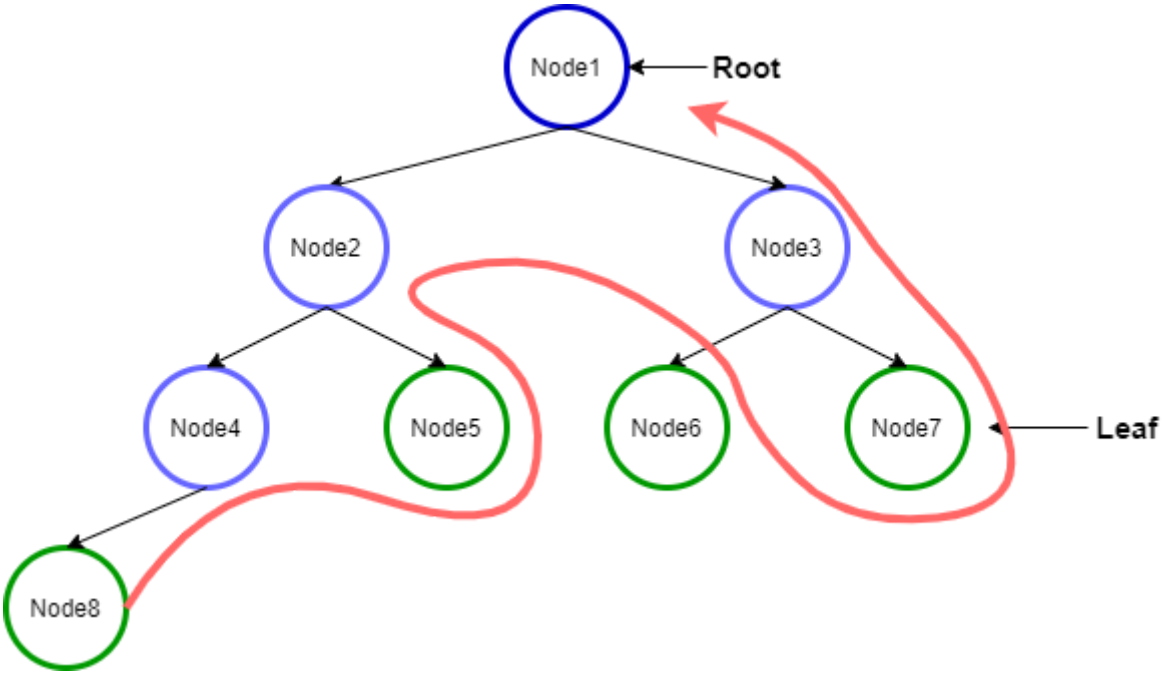
- 현재 자기 자신 노드를 중간에 순회하고, Left → 자기 자신 → Right 순서로 순회
  - 아래 그림에서 8 → 4 → 2 → 5 → 1 → 6 → 3 → 7




```
// 현재 parameter로 전달된 노드가 null이 아니라면
// 현재 노드의 left child 노드부터 순회하고
// 자신이 가진 값을 출력한 다음 right child 노드를 방문
public void inorder(Node node){
        if(node != null){
            inorder(node.left);
            System.out.print(node.value);
            inorder(node.right);
        }
    }
```

후위 순회

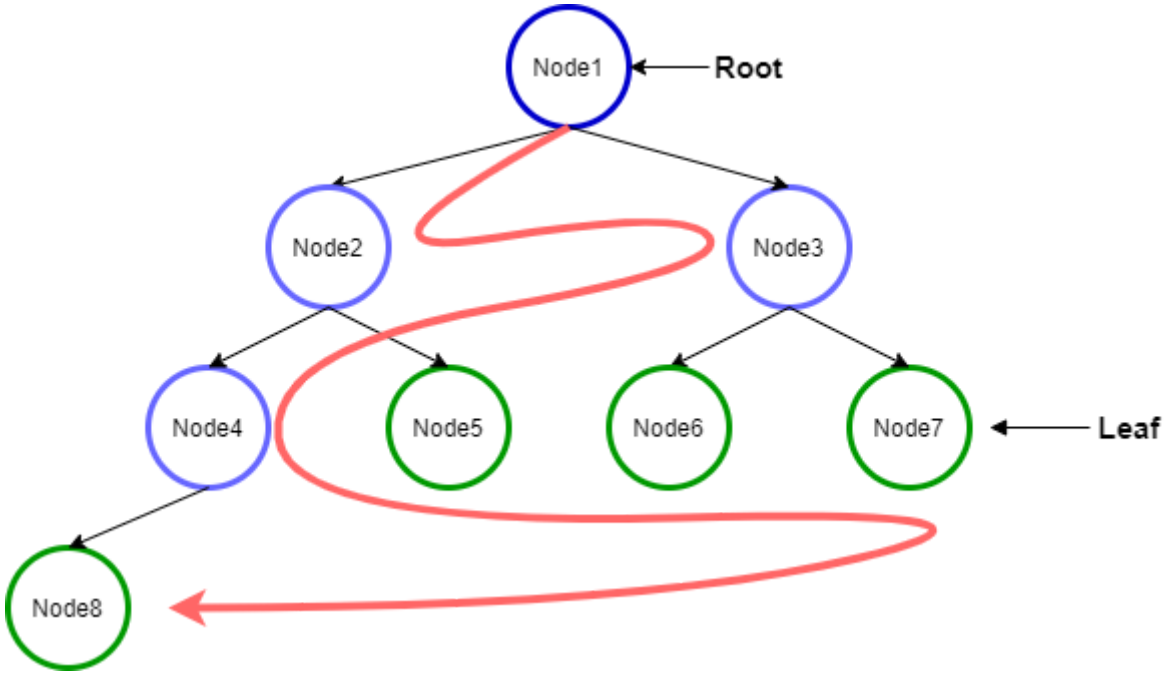
- 현재 자기 자신 노드를 가장 마지막에 순회
  - 아래 그림에서 8 → 4 → 5 → 2 → 6 → 7 → 3 → 1
- 후위 순회가 제일 중요
  - 리프에서 루트로 올라갈수록 작은 문제들의 결과를 통해 큰 문제를 해결하는 bottom-up이라고 한다면, 자식 노드들의 결과를 통하여 현재 노드의 결과를 구하는 방식이 필요할 수 있음
  - 따라서 DP나 세그먼트 트리로 문제를 해결할 때 모두 후위순회 사용!




```
// 현재 parameter로 전달된 노드가 null이 아니라면
// 현재 노드의 left child 노드부터 순회하고
// 자신이 가진 값을 출력한 다음 right child 노드를 방문
public void postOrder(Node node){
        if(node != null){
            postOrder(node.left);
            postOrder(node.right);
            System.out.print(node.value);
        }
    }
```

레벨 순회

- 다른 순회들과는 다르게, 레벨 순회는 형제 관계에 있는 노드 간에 방문 순서를 정함
  - 아래 그림에서 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8
  - 각 층을 모두 방문 후 자식 노드로 이어져서 방문
  - 최상단 노드부터 시작하여 자신의 자식 노드를 모두 방문하고 다음 단계로 넘어가야 하므로 Queue를 이용
  - 최상단 노드 방문 후, 자식 노드를 차례로 Queue에 넣고, 각 자식 노드를 방문할 때마다 해당 노드가 자식 노드를 갖고 있다면 Queue에 넣어서 순차적으로 방문
- 각 노드의 속성으로 부모 자식 관계를 알 수 있도록 되어있지만 형제는 설정하지 않아서 재귀적으로 풀기 어려움




```
public void levelOrder(Node root){
        Queue<Node<T>> q = new LinkedList<>();
        q.add(root);

        while(!q.isEmpty()){
            Node temp = q.peek();
            System.out.print(temp.value);
            if(temp.left != null){
                q.add(temp.left);
            }
            if(temp.right != null) {
                q.add(temp.right);
            }
            q.poll();
        }
    }
```

이진 트리 구현 전체 코드

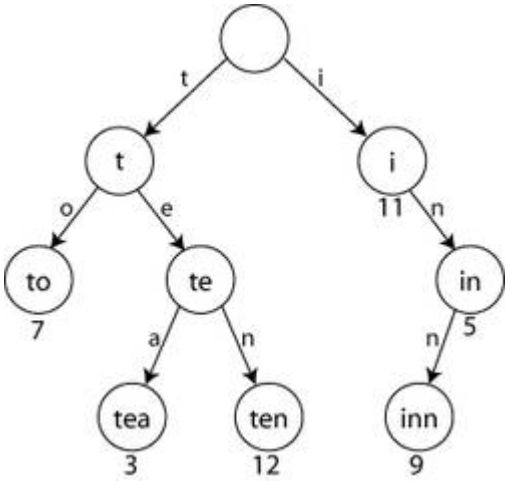
```
import java.util.LinkedList;
import java.util.Queue;

public class BinaryTree<T> {
    class Node<T>{
        T value;
        Node parent;
        Node left;
        Node right;

        public Node(T value){
```

```
            this.value = value;
        }
    }
    Node<T> root = null;
    int size = 0;

    public BinaryTree(){}

    public boolean insert(T value){
        Node newNode = new Node(value);
        if(size == 0){
            root = newNode;
            return true;
        }
        Queue<Node<T>> q = new LinkedList<>();
        q.add(root);

        while(true){
            Node tmp = q.peek();
            if(tmp.left == null){
                tmp.left = newNode;
                newNode.parent=tmp;
                break;
            } else {
                q.add(tmp.left);
            }

            if(tmp.right == null){
                tmp.right = newNode;
                newNode.parent=tmp;
                break;
            } else {
                q.add(tmp.right);
            }
        }
        return true;
    }

    public void inorder(Node node){
        if(node != null){
            inorder(node.left);
            System.out.print(node.value);
            inorder(node.right);
        }
    }

    public void preOrder(Node node){
        if(node != null){
            System.out.print(node.value);
            preOrder(node.left);
            preOrder(node.right);
        }
    }

    public void postOrder(Node node){
        if(node != null){
            preOrder(node.left);
            preOrder(node.right);
```

```
            System.out.print(node.value);
        }
    }

    public void levelOrder(Node root){
        Queue<Node<T>> q = new LinkedList<>();
        q.add(root);

        while(!q.isEmpty()){
            Node temp = q.peek();
            System.out.print(temp.value);
            if(temp.left != null){
                q.add(temp.left);
            }
            if(temp.right != null) {
                q.add(temp.right);
            }
            q.poll();
        }

    }
}
```

Trie (boj 14426)

- 여러 개의 문자열을 가지고 있다고 가정했을 때, 임의의 문자열이 가지고 있는 문자열 중 하나인지 알아내는 가장 효율적인 방법
  - 단순히 Brute Force로 비교를 한다면 시간이 매우 오래 걸림
  - 이진 탐색을 하면 단축시킬 수 있지만 정렬 자체에 시간이 오래 걸림
- 구조
  - 기본적으로 K진 트리 (K개의 문자)
  - 'tea'라는 문자열이 입력되었다면 순서대로 머릿글자 't'가 등록되고 그 다음 'e'가, 그 다음 'a'가 등록
  - 순차적으로 따라가며 문자열을 모두 찾았다면 그 위치에 표시하면 됨 (접두사 찾기에 많이 이용)
  - 이러한 트라이 구조는 찾고자 하는 문자열만큼 공간을 많이 사용하지만 그 문자열의 길이의 속도만큼 초고속 탐색이 가능



- 시간 복잡도
  - 문자열 집합의 개수와 상관 없이 찾고자 하는 문자열의 길이가 시간 복잡도 그 자체 (문자열 길이가 m이라면 시간 복잡도 `O(m)`)