Tree Structure

백설기 4차 스터디

AI6/1반 조해정

자료 구조 (data structure)

자료 구조

자료구조는 **여러 데이터들의 묶음을 저장**하고, **사용하는 방법을 정의**한 것이다.

데이터를 체계적으로 저장하고, **효율적으로 활용**하기 위해 자료구조를 사용한다.
대부분의 자료구조는 **특정한 상황**에 놓인 문제를 해결하는 데에 **특화**되어 있다.

신중히 선택한 자료 구조는 **보다 효율적인 알고리즘**을 사용할 수 있게 한다.
>> 문제를 빠르고 정확하게 해결할 수 있다.

종류:

- 단순구조 : 2진수, 정수/실수, 문자/문자열
- 선형구조 : 리스트(배열), 연결리스트(단순, 이중, 원형), 덱, 스택, 큐
- 비선형구조 : 트리(일반, 이진), 그래프(방향, 무방향)
- 파일구조 : 순차 파일, 색인 파일, 직접 파일

글, 그림 참조:
https://hanamon.kr/%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0%EB%9E%80-%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0%EB%A5%BC-%EB%B0%B0%EC%9A%B0%EB%8A%94-%EC%9D%B4%EC%9C%A0/

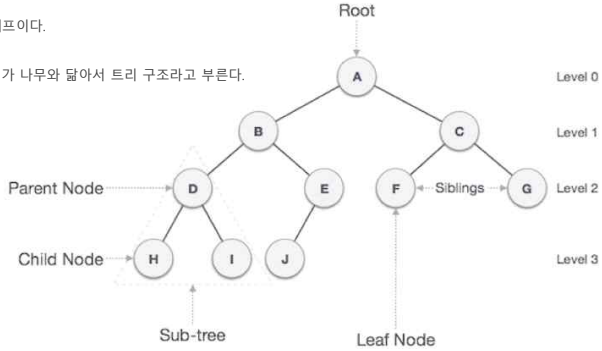
CONTENTS

Tree

트리는 **노드로 이루어진 자료구조**이다.
서로 다른 두 노드를 잇는 길이 하나 뿐인 그래프이다.

하나의 뿌리로부터 가지가 사방으로 뻗은 형태가 나무와 닮아서 트리 구조라고 부른다.

- **노드(Node)** :
  트리 구조를 이루는 모든 개별 데이터
- **간선(edge)** :
  노드를 연결하는 선
  link, branch 라고도 부름
- **부모 노드(Parent Node)** :
  두 노드가 상하관계로 연결되어 있을 때,
  상대적으로 루트에서 가까운 노드
- **자식 노드(Child Node)** :
  두 노드가 상하관계로 연결되어 있을 때,
  상대적으로 루트에서 먼 노드



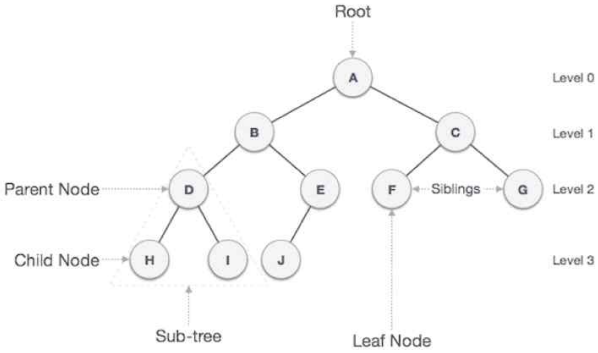
글, 그림 참조:
https://gmlwjd9405.github.io/2018/08/12/data-structure-tree.html
https://hanamon.kr/%ec%9e%90%eb%a3%8c%ea%b5%ac%ec%a1%b0-tree-%ed%8a%b8%eb%a6%ac/

Tree

- **루트 노드(Root Node)** :
  부모가 없는 노드, 항상 자식 노드를 가진다
  트리의 최상위 노드
- **리프 노드(Leaf Node)** :
  자식이 없는 노드
  잎 노드 또는 말단 노드라고 부름

- **내부 노드(Internal Node)** :
  잎 노드가 아닌 노드
- **형제(Sibling)** :
  같은 부모를 가지는 노드

- **서브 트리(Sub tree)** :
  root에서 뻗어나오는 큰 트리의 내부에,
  트리 구조를 갖춘 작은 트리
  트리의 자식도 트리, 트리의 자식의 자식도

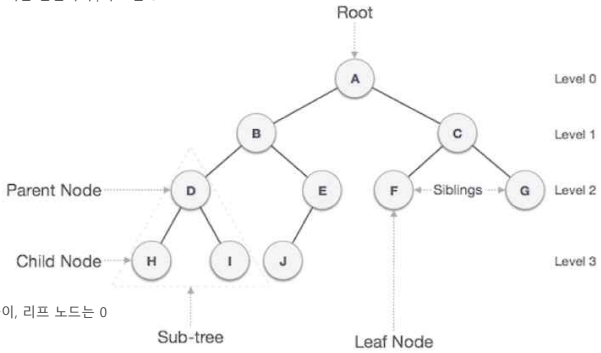


글, 그림 참조:
https://gmlwjd9405.github.io/2018/08/12/data-structure-tree.html
https://hanamon.kr/%ec%9e%90%eb%a3%8c%ea%b5%ac%ec%a1%b0-tree-%ed%8a%b8%eb%a6%ac/

Tree

- **노드의 깊이(Depth)** :
  루트에서 어떤 노드에 도달하기 위해 거쳐야 하는 간선의 수, 루트는 0
- **노드의 레벨(Level)** :
  트리의 특정 깊이를 가지는 노드의 집합
  Level 1의 노드들: {B, C}

- **노드의 크기(Size)** :
  자신을 포함한 모든 자손 노드의 개수
- **노드의 차수(Degree)** :
  각 노드가 지닌 가지의 수
  = 자식 노드의 개수
- **트리의 차수(Degree of Tree)** :
  트리의 최대 차수
- **트리의 높이(Height)** :
  리프 노드와 직간접적으로 연결된 노드의 높이, 리프 노드는 0
  보통 루트까지의 높이를 표현
  예제 트리의 경우, 루트 A의 높이=3



글, 그림 참조:
https://gmlwjd9405.github.io/2018/08/12/data-structure-tree.html
https://hanamon.kr/%ec%9e%90%eb%a3%8c%ea%b5%ac%ec%a1%b0-tree-%ed%8a%b8%eb%a6%ac/

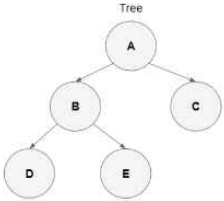
Tree

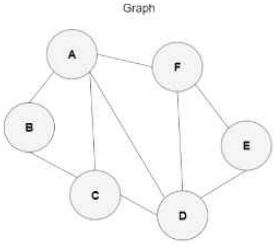
Tree

트리는
- 부모-자식의 관계를 가지는 계층형 모델
- 두 개의 노드 사이에 반드시 1개의 경로만을 가지며,
  사이클이 존재하지 않는 방향 그래프,
  '**최소 연결 트리**'라고 부르기도 한다.

트리의 특징 :
- 단방향, 비순환, 루트 노드 존재, 계층형
- 부모-자식 관계가 존재해 레벨이 존재
- 노드가 N개면 간선은 N-1개
- 완전이진트리의 경우,
  각 레벨 k에 존재하는 노드는 2^k 개

트리의 실사용 예 :
조직도, 가계도, 토너먼트 대진표, 디렉토리 구조

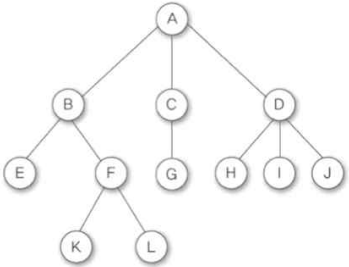




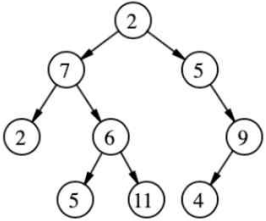
글, 그림 참조:
https://gmlwjd9405.github.io/2018/08/12/data-structure-tree.html
https://wooono.tistory.com/530
https://hanamon.kr/%ec%9e%90%eb%a3%8c%ea%b5%ac%ec%a1%b0-tree-%ed%8a%b8%eb%a6%ac/

이진 트리(Binary Tree)

트리의 **차수가 2 이하**가 되도록 하는 트리 -> 자식 노드의 갯수가 최대 2개
이 두 개의 자식 노드는 **왼쪽 자식 노드**와 **오른쪽 자식 노드**로 나눌 수 있다.
이진 트리는 **자료의 삽입, 삭제 방법**에 따라 나뉜다.



일반트리

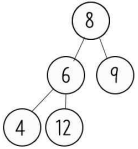


이진트리

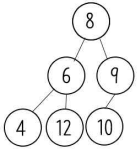
글, 그림 참조:
https://richard25.tistory.com/73

이진 트리의 유형

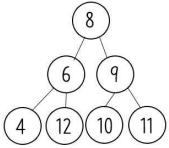
- **정 이진 트리(Full Binary Tree)**
  모든 노드가 0개 또는 2개의 자식 노드를 가진다.



- **완전 이진 트리(Complete Binary Tree)**
  마지막 레벨을 제외하고 모든 레벨이 완전히 채워져 있으며,
  마지막 레벨의 모든 노드는 가장 왼쪽부터 채워져 있다.



- **포화 이진 트리(Perfect Binary Tree)**
  모든 노드가 2개의 자식 노드를 갖고 있으며,
  모든 리프 노드가 동일한 깊이 또는 레벨을 갖는다.
  정 이진 트리이면서 완전 이진 트리이다.

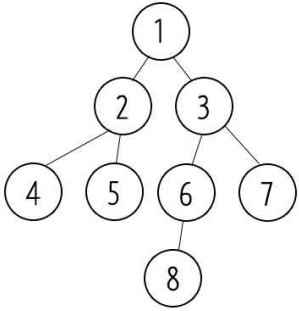


글, 그림 참조:
https://it-and-life.tistory.com/164

Binary Tree Traversal - Depth First

이진 트리의 깊이 우선 순회(Depth First Traversal) - 3가지

루트(Root)를 제일 먼저 순회 하냐, 중간에 하냐, 제일 나중에 하냐에 따라 달라진다.



- **전위순회 (Pre-order: Root-L-R)** : 루트를 제일 먼저 순회
  루트에서 시작해 왼쪽의 노드들을 순차적으로 둘러본 뒤, 왼쪽의 노드 탐색이 끝나면 오른쪽 노드를 탐색한다.
  1-2-4-5-3-6-8-7

- **중위순회 (In-order: L-Root-R)** : 루트를 중간에 순회
  제일 왼쪽 끝에 있는 노드부터 순회하기 시작하여, 루트를 기준으로 왼쪽에 있는 순회가 끝나면 루트를 거쳐 오른쪽에 있는 노드로 이동하여 마저 탐색한다.
  4-2-5-1-8-6-3-7

- **후위순회 (Post-order: L-R-Root)** : 루트를 제일 나중에 순회
  제일 왼쪽 끝에 있는 노드부터 순회하기 시작하여, 루트를 거치지 않고 오른쪽으로 이동해 순회한 뒤, 제일 마지막에 루트를 방문한다.
  4-5-2-8-6-7-3-1

글, 그림 참조:
https://it-and-life.tistory.com/164
https://hanamon.kr/%ec%9e%90%eb%a3%8c%ea%b5%ac%ec%a1%b0-%ed%83%90%ec%83%89-tree-traversal-%ed%8a%b8%eb%a6%ac-%ec%88%9c%ed%9a%8c/

Binary Tree Traversal - Depth First

순회 코드 구현 사고 흐름

뿌리 노드의 왼쪽 자식 노드는 왼쪽 서브트리의 뿌리 노드이다.
뿌리 노드의 오른쪽 자식 노드는 오른쪽 서브트리의 뿌리 노드이다.
어느 쪽의 서브트리이던, 리프노드까지 쭉 가지를 타고 내려간다.
리프 노드는 더 이상 자식 노드가 없기에 그 다음 순회를 행한다.

전위의 경우 :
뿌리 노드 출력
뿌리의 왼쪽 서브트리 순회
뿌리의 오른쪽 서브트리 순회
뿌리 노드가 리프 노드가 된다면 패스
(리프 노드는 자식 노드가 없다는 점을 이용해 판별)

>> !! **재귀함수**를 사용 !!

글, 그림 참조:
http://ejklike.github.io/2018/01/09/traversing-a-binary-tree-2.html

Binary Tree Traversal - Depth First

```python
# Node 클래스, 트리 자료 구조를 이루는 하나의 알갱이, 부모 노드가 되어 자식 노드 2개를 가진다.
class Node:
    # 초기화 시, 해당 노드의 값만 주어짐, 좌우 자식 노드는 비었다.
    # 3가지 속성: 노드값(self.data), 왼쪽 자식 노드(self.left), 오른쪽 자식 노드(self.right)
    def __init__(self, data):
        self.data = data
        self.left = self.right = None


# 이진 트리 클래스
class BinaryTree:
    # 처음엔 비어 있는 트리, 최상위 부모 노드인 root 만을 가진다.
    def __init__(self, root=None):
        self.root = root

    # 원소를 추가, 삭제, 탐색할 수 있도록 각종 method 를 추가.
    # insert(), delete(), find(), ...
    # 생략함
    # self.root = Node class instance
```

글, 그림 참조:
http://ejklike.github.io/2018/01/09/traversing-a-binary-tree-2.html

Binary Tree Traversal - Depth First

```
    # 재귀 함수로 순회 구현
    # 재귀 탈출 조건: 리프 노드
    # 순회 할 노드는 해당 서브 트리의 최상위 노드라 root 라고 표기

    # 전위 순회(Pre-order Traversal)
    # 루트 노드 -> 왼쪽 서브 트리 -> 오른쪽 서브 트리 순서 (Root-Left-Right)
    def preorder_traversal(self):
        def preorder(root):
            if root is None:    # 상위 노드가 리프 노드라 자식이 없는(None) 것
                pass
            else:
                print(root.data)
                preorder(root.left)
                preorder(root.right)
        preorder(self.root)     # 전체 이진 트리의 최상위 노드인 루트 노드부터 순회 시작
```

글, 그림 참조:
http://ejklike.github.io/2018/01/09/traversing-a-binary-tree-2.html

Binary Tree Traversal - Depth First

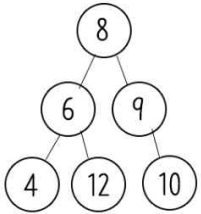
```
    # 중위 순회(In-order Traversal)
    # 왼쪽 서브 트리 -> 뿌리 노드 -> 오른쪽 서브 트리 (Left-Root-Right)
    def inorder_traversal(self):
        def inorder(root):
            if root is None:    # 상위 노드가 리프 노드라 자식이 없는(None) 것
                pass
            else:
                inorder(root.left)
                print(root.data)
                inorder(root.right)
        inorder(self.root)     # 전체 이진 트리의 최상위 노드인 루트 노드부터 순회 시작

    # 후위 순회(Pre-order Traversal)
    # 왼쪽 서브 트리 -> 오른쪽 서브 트리 -> 뿌리 노드 순서 (Left-Right-Root)
    def postorder_traversal(self):
        def postorder(root):
            if root is None:    # 상위 노드가 리프 노드라 자식이 없는(None) 것
                pass
            else:
                postorder(root.left)
                postorder(root.right)
                print(root.data)
        postorder(self.root)     # 전체 이진 트리의 최상위 노드인 루트 노드부터 순회 시작
```

글, 그림 참조:
http://ejklike.github.io/201

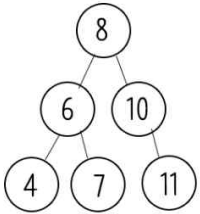
Binary Search Tree

이진 탐색 트리(Binary Search Tree)

- 모든 원소는 **서로 다른 유일한 값**을 갖는다. (검색 목적 자료구조이므로)
- **모든 왼쪽 자식**의 값이 루트나 부모**보다 작고, 모든 오른쪽 자식**의 값이 루트나 부모**보다 큰** 값을 가진다.
- 왼쪽 서브트리와 오른쪽 **서브트리 또한 이진 탐색 트리**이다.
  이진 탐색 트리를 통해 찾고자 하는 **원소를 보다 빠르고 효율적으로 찾거나 추가할 수 있다는 장점**이 있다.
- 이진 탐색 트리는 균형 잡힌 트리가 아닐 때(=편향 트리), 입력되는 값의 순서에 따라 한쪽으로 노드들이 몰리게 될 수 있다.



Binary tree

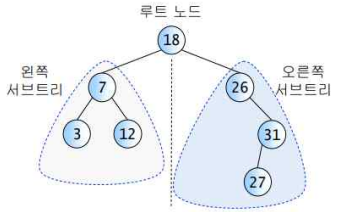


Binary search tree

글, 그림 참조:
https://it-and-life.tistory.com/164
https://richard25.tistory.com/73

이진 탐색 트리(Binary Search Tree)

이진 탐색 트리를 순회할 땐 **중위순회** 방법을 사용한다.
중위순회를 하게 되면 **정렬된 일련의 데이터**가 나오게 된다.
오른쪽의 예시의 경우, 중위 순회 결과는 '3-7-12-18-26-17-31'로 정렬되어 있다.



이진 탐색 트리의 연산

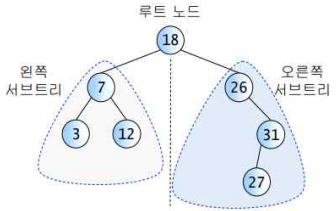
- **검색 : 타겟 데이터가 존재하는 지**
  찾고자하는 값과 현재 루트 노드의 값 비교
  타겟 값이 더 크다면 오른쪽 서브 트리로, 더 작다면 왼쪽 서브 트리로
  위 로직 **재귀적 반복 수행**, 루트 노드의 값이 타겟 데이터일 때까지 탐색

- **삽입 : 원하는 데이터를 삽입**
  새로운 데이터가 들어갈 자리가 비어있으면 그대로 값 대입
  비어있지 않은 경우 해당 자리의 노드 값과 비교
  삽입할 데이터보다 작다면 왼쪽 자식 트리로 이동, 보다 크면 오른쪽 자식 트리로 이동

글, 그림 참조:
https://velog.io/@haero_kim/%EC%9D%B4%EC%A7%84-%ED%83%90%EC%83%89-%ED%8A%B8%EB%A6%AC-Binary-Search-Tree

이진 탐색 트리의 연산

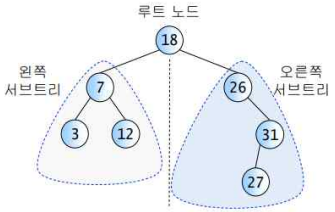
- **삭제 : 타겟 데이터를 삭제**
  1. 리프 노드를 삭제하는 경우
     그냥 삭제하면 됨
  2. 자식 노드가 하나인 노드를 삭제하는 경우
     해당 자식 노드를 삭제할 노드의 위치로 끌어올림
  3. 자식 노드가 두 개인 노드를 삭제하는 경우
     오른쪽 서브 트리의 MIN 값 또는 왼쪽 서브 트리의 MAX 값 중 하나를 삭제할 노드의 위치로 끌어올림



글, 그림 참조:
https://velog.io/@haero_kim/%EC%9D%B4%EC%A7%84-%ED%83%90%EC%83%89-%ED%8A%B8%EB%A6%AC-Binary-Search-Tree

이진 탐색 트리의 연산 시간 복잡도

이진 탐색 트리의 세 개 연산(검색, 삽입, 삭제)은 결국 트리를 순회하며
타겟 데이터의 위치를 찾는 연산이 공통적으로 필요하므로,
트리의 높이에 비례하여 시간 복잡도가 증가한다.



포화 이진 탐색 트리 :
모든 노드가 빈 자리 없이 꽉꽉 채워져 있는 이진 탐색 트리
가장 최적의 상황으로 구성된 트리

포화 이진 탐색 트리를 구성하고 있는 노드의 개수를 N, 높이를 H라고 했을 때,

$$N = 2^{(H+1)} - 1$$

노드의 개수를 통해 트리의 높이를 계산하면,

$$H = \log_2(N+1) - 1$$

따라서, 균형적으로 생성되어 있는 트리의 높이를 H라고 한다면, H에 비례하여 O(H) 시간 복잡도를 가지게 되는데,
이를 Big-O 시간 복잡도로 표현하면 **O(logN)**으로 표기할 수 있다.

글, 그림 참조:
https://velog.io/@haero_kim/%EC%9D%B4%EC%A7%84-%ED%83%90%EC%83%89-%ED%8A%B8%EB%A6%AC-Binary-Search-Tree

감사합니다