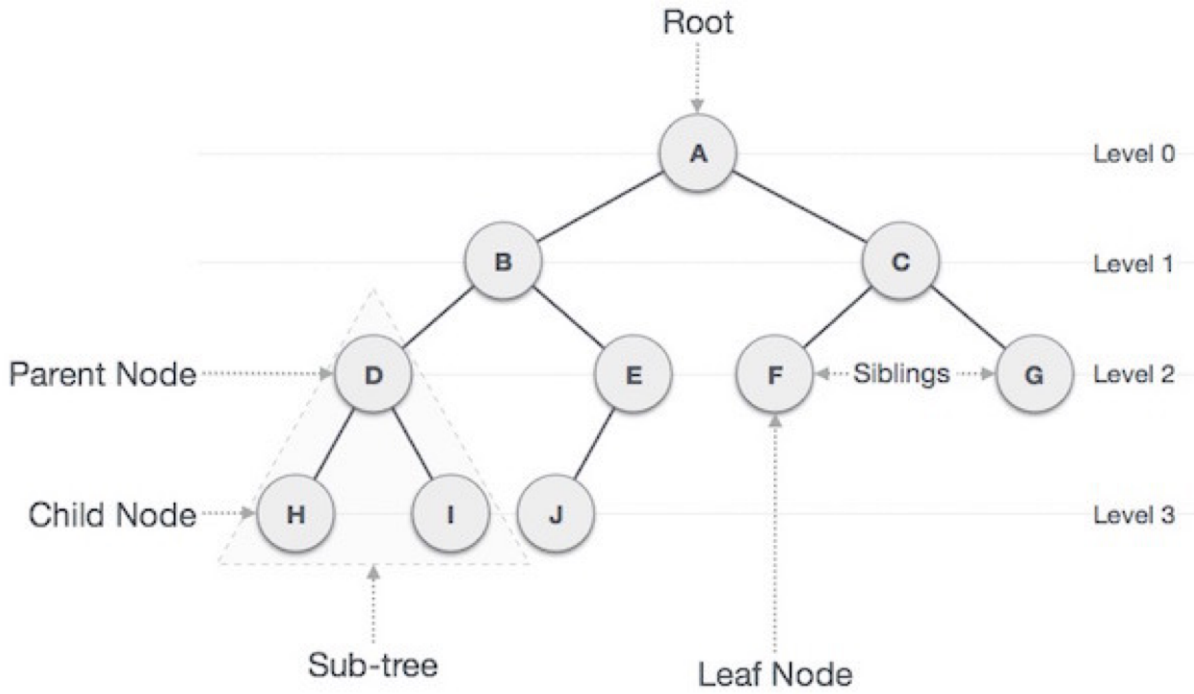
트리

트리 기본 개념



- **단말 노드(leaf node)**: 자식이 없는 노드, '말단 노드' 또는 '잎 노드'라고도 부른다.
- **내부(internal) 노드**: 단말 노드가 아닌 노드
- **간선(edge)**: 노드를 연결하는 선 (link, branch 라고도 부름)
- **형제(sibling)**: 같은 부모를 가지는 노드
- **노드의 깊이(depth)**: 루트에서 어떤 노드에 도달하기 위해 거쳐야 하는 간선의 수
- **노드의 레벨(level)**: 트리의 특정 깊이를 가지는 노드의 집합
- **트리의 높이(height)**: 루트 노드에서 가장 깊숙히 있는 노드의 깊이
- **노드의 크기(size)**: 자신을 포함한 모든 자손 노드의 개수

- 노드의 차수(degree): 하위 트리 개수 / 간선 수 (degree) = 각 노드가 지닌 가지의 수 ?
- 트리의 차수(degree of tree): 트리의 최대 차수 ?

특징

- 하나의 루트 노드를 갖는다.
- 수직적 계층적 관계를 표현
- 방향성이 있는 비순환 그래프. 순환하는 구조가 아님
- 특정 노드로 가는 경로는 하나다.

- 분할 정복 탐색 알고리즘에 쓰임
- 탐색 속도가 빠름

트리의 종류

- 이진 트리 (Binary tree)
- 이진 탐색 트리 (Binary search tree)
- 균형트리 (AVL tree, RB 트리)

이진 트리 (Binary Tree)

한 노드가 최대 2개의 자식 노드를 갖는 트리

크기 정렬이 되어있지 않음.

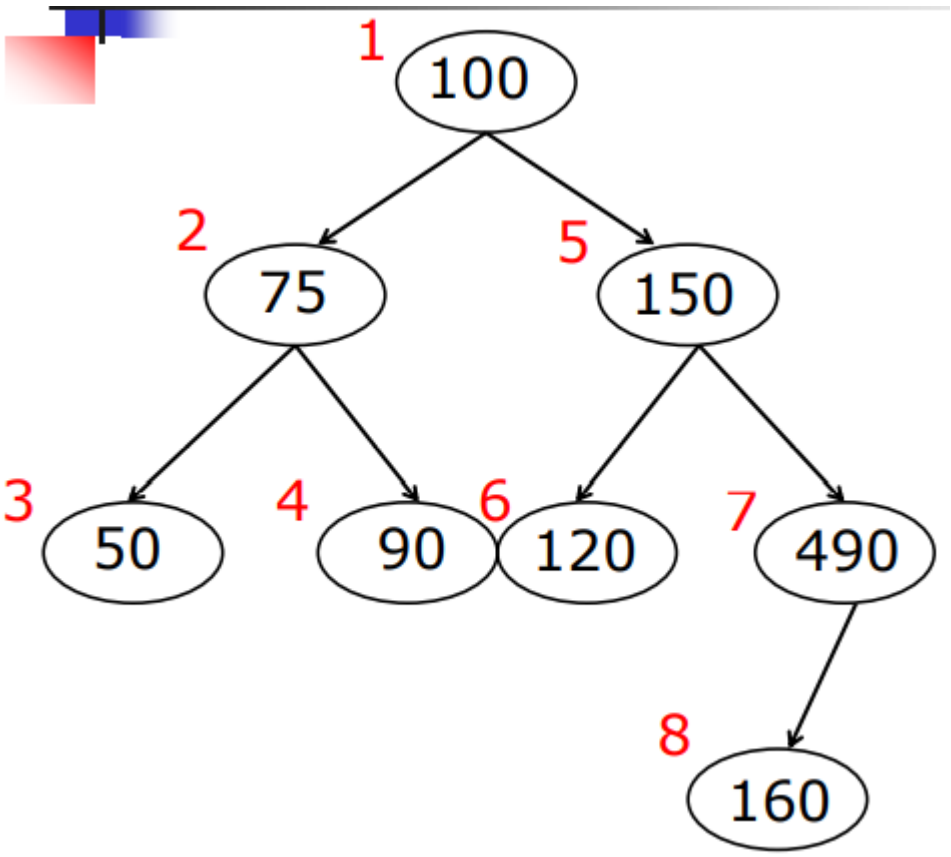
3가지 순회(traverse) 방법이 있음

- 전위(preorder) 후위(postorder) 중위(inorder)

전위 순회

1. 자기자신 방문(출력)
2. 왼쪽 자식 노드 호출

3. 없으면 오른쪽 자식 노드 호출

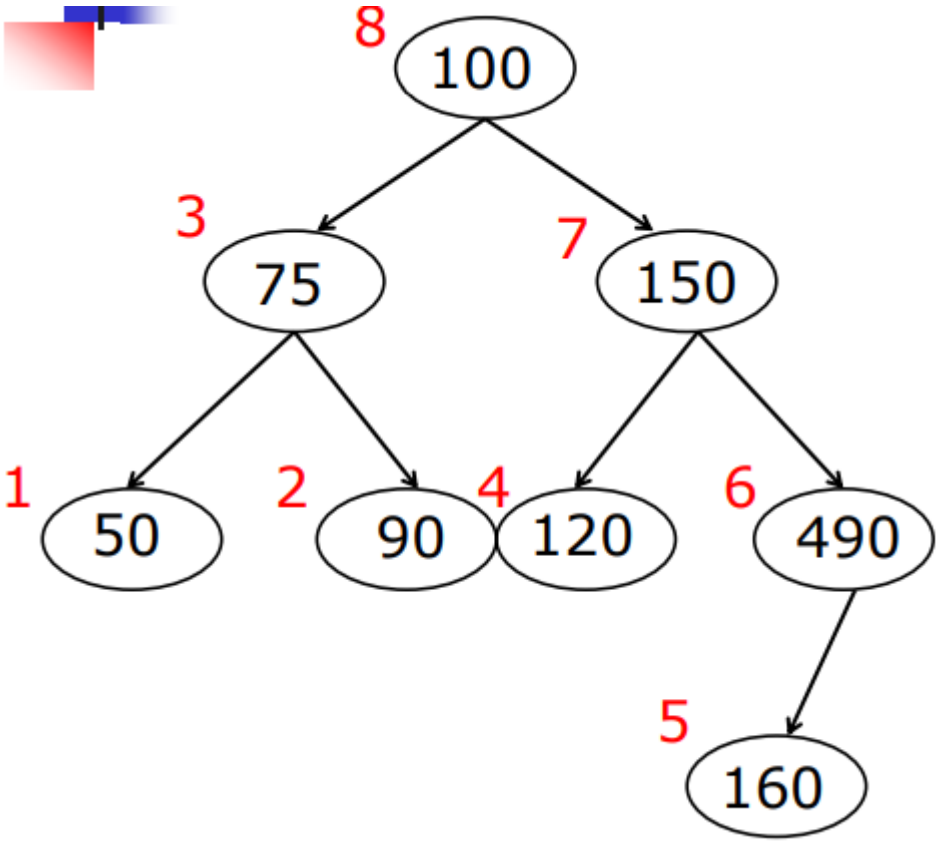


출력 순서 : 100 → 75 → 50 → 90 → 150 → 120 → 490 → 160

```
#노드 클래스 생성
class Node:
  def __init__(self,item) :
    self.item = item # 자기 자신
    self.left = None # 왼쪽 자식 노드
    self.right = None # 오른쪽 자식 노드

#preorder 전위 순회
def preorder(self,n): # n -> 노드
  if n != None:
    print(n.item,'',end='') #노드 방문
    if n.left:
      self.preorder(n.left) # 왼쪽 자식이 있으면 재귀로 왼쪽 자식 노드 방문
    if n.right:
      self.preorder(n.right) # 왼쪽 자식 방문뒤 오른쪽 자식 노드 방문
```

후위 순회

1. 왼쪽 자식 호출
2. 오른쪽 자식 호출
3. 자기자신 방문(출력)



순회(방문) 순서 : 50→ 90→ 75→120 → 160 → 490 →150 → 100

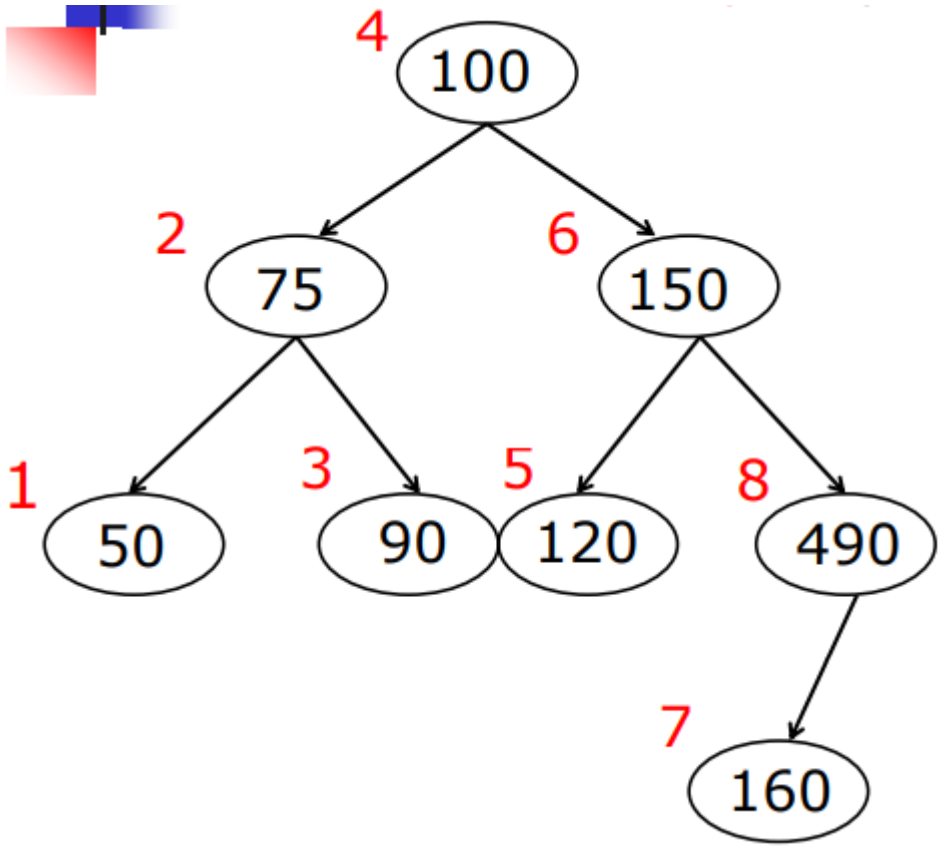
왼쪽에서 오른쪽으로 바닥 쓸면서 올라오는 방식

```
def postorder(self,n):# n = 노드
if n!= None:
  if n.left:
    self.postorder(n.left): # 왼쪽 자식이 있으면 호출
  if n.rignt:
    self.postorder(n.right): # 오른쪽 자식이 있으면 호출
  print(n.item,'',end='') # 노드 방문
```

중위순회

이진 탐색트리 크기순으로 나열 할때 쓰는 방법

1. 왼쪽 자식 노드 호출
2. 자기 자신 출력
3. 오른쪽 자식 노드 호출



순회 순서: 50 → 75 → 90→ 100 → 120 → 150 → 160 → 490

https://brunch.co.kr/@qqplot/131

이진 탐색 트리 BST

특징

- 이진 트리를 정렬한 버전. (좌에서 우로 오름 차순)
  - 자식 노드가 최대 2개

- 탐색 속도가 더 빠름
    - 시간 복잡도 = O(h) h: 트리의 높이
- 자료 입력과 삭제가 편함.
- 생성 → 삽입 → 삭제

이진 탐색 트리 클래스 정의 및 초기화

```
#노드 클래스 만들기
class Node(object):
    def __init__(self, data):
        self.data = data
        self.left = self.right = None
```

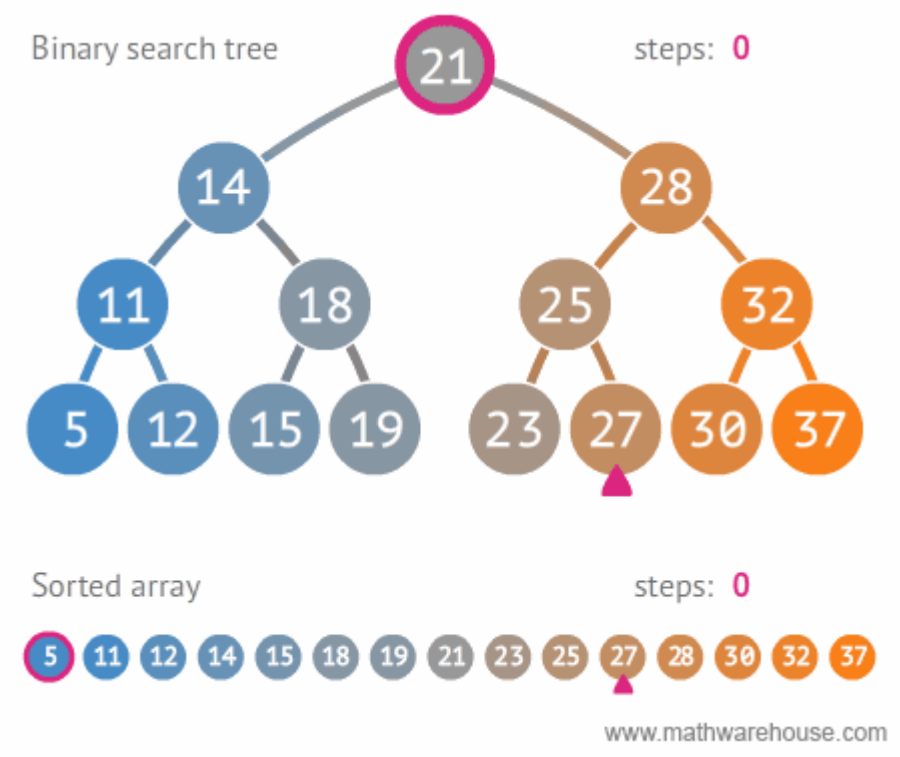
이진탐색 트리 클래스

```
class BinarySearchTree(object):
  def __init__(self):
    self.root = None
```

이진 탐색 트리 삽입

```
def insert(self, data):
        self.root = self._insert_value(self.root, data)
        return self.root is not None

    def _insert_value(self, node, data):
        #삽입하려는 노드가 비어있을 때
        if node is None:
                node = Node(data)
        #노드의 크기와 비교해서 왼쪽 자식 또는 오른쪽 자식에 삽입
        else:
            if data <= node.data:
                node.left = self._insert_value(node.left, data)
            else:
                node.right = self._insert_value(node.right, data)
        return node
```

탐색




```
def find(self, key):
        return self._find_value(self.root.key)

    def _find_value(self, root,key):
        if root is None or root.data == key:
```

```
            return root is not None
        elif key < root.data:
            return self._find_value(root.left,key)
        else:
            return self._find_value(root.right, key)
```

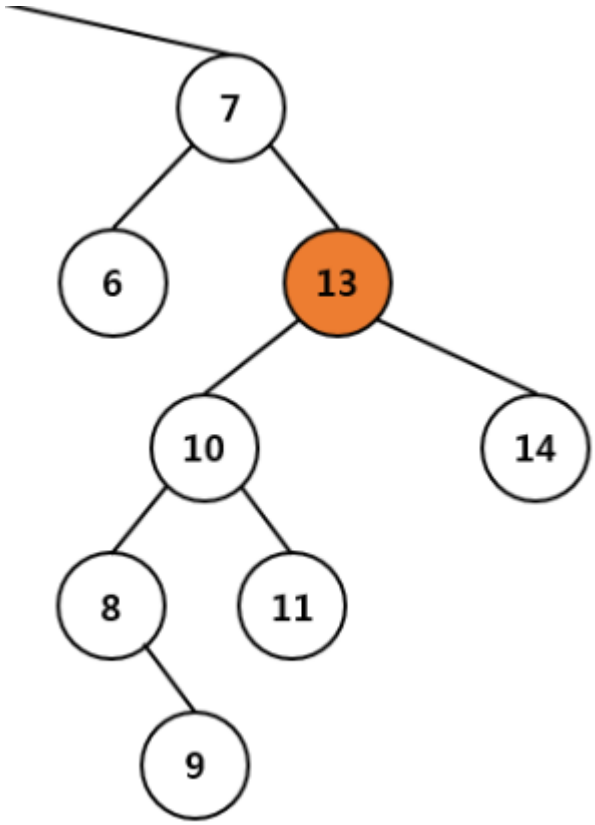
노드 삭제

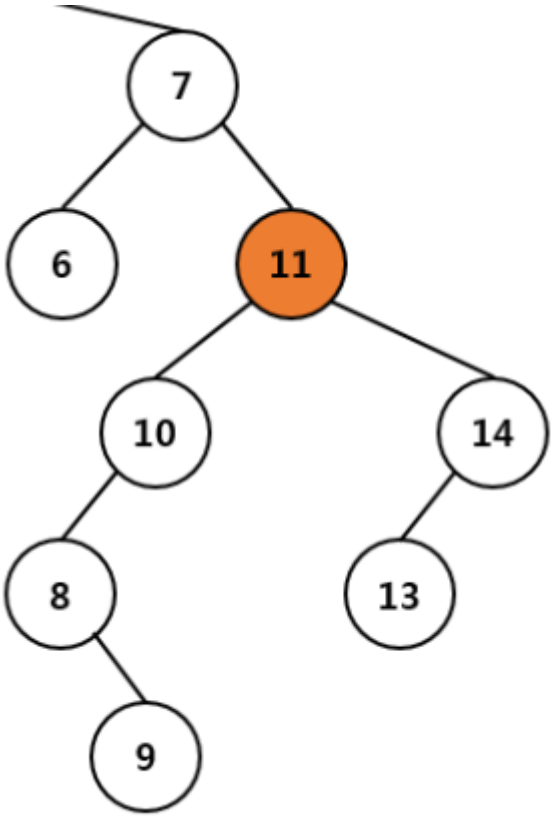
1. 자식 노드가 없는 노드 삭제
    - 해당 노드 바로 삭제
2. 자식이 하나인 노드 삭제
    - 해당 노드를 삭제하고 자식노드를 삭제한 노드 위치에 가지고 오기.
3. 자식노드가 두개인 노드 삭제
    a. 오른쪽 자식들 중 가장 왼쪽 자식(가장 작은 값)을 삭제할 노드 위치로 복사

    → 가장 왼쪽 자식 노드 삭제

    b. 왼쪽 자식들 중 가장 오른쪽 자식(가장 큰 값)을 삭제할 노드 위치로 가지고 오기.

    → → 가장 오른쪽 자식 노드 삭제


7
6
13
10
14
8
11
9


7
6
11
10
14
8
13
9

```
def delete(self, key):
        self.root, deleted = self._delete_value(self.root, key)
        return deleted

def _delete_value(self, node, key):
    #삭제할 노드가 없을때
    if node is None:
        return node, False

    deleted = False
    # 삭제할 노드를 찾았을때
    if key == node.data:
        deleted = True
        #자식 노드가 2개일때
        if node.left and node.right:
            #오른쪽 자식들중 가장 왼쪽에있는 자식으로 대체
            parent, child = node, node.right
            while child.left is not None:
                parent, child = child, child.left
            child.left = node.left
            if parent != node:
                parent.left = child.right
                child.right = node.right
            node = child     ###parent -> child 이동
        # 자식 노드가 하나일때
        elif node.left or node.right:
            node = node.left or node.right
        else:
            node = None
    # 삭제할 키 값 찾기
    elif key < node.data:
            node.left, deleted = self._delete_value(node.left, key)
    else:
            node.right, deleted = self._delete_value(node.right, key)
    return node, deleted
```

▼ 예제 코드 전체

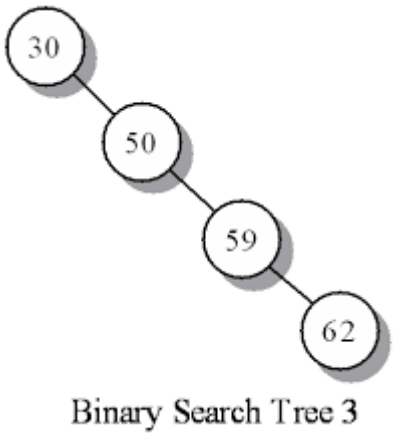
```
class Node(object):
    def __init__(self, data):
        self.data = data
        self.left = self.right = None

class BinarySearchTree(object):
    def __init__(self):
        self.root = None

    #노드 삽입
    def insert(self, data):
        self.root = self._insert_value(self.root, data)
```

```
        return self.root is not None

    def _insert_value(self, node, data):
        #삽입하려는 노드가 비어있을 때
        if node is None:
                node = Node(data)
        #노드의 크기와 비교해서 왼쪽 자식 또는 오른쪽 자식에 삽입
        else:
            if data <= node.data:
                node.left = self._insert_value(node.left, data)
            else:
                node.right = self._insert_value(node.right, data)
        return node

    # 노드 탐색
    def find(self, key):
        return self._find_value(self.root,key)

    def _find_value(self, root,key):
        if root is None or root.data == key:
            return root is not None
        elif key < root.data:
            return self._find_value(root.left,key)
        else:
            return self._find_value(root.right, key)

    # 노드 삭제

    def delete(self, key):
        self.root, deleted = self._delete_value(self.root, key)
        return deleted

    def _delete_value(self, node, key):
        if node is None:
            return node, False

        deleted = False
        if key == node.data:
            deleted = True
            if node.left and node.right:
                #replace the node to the leftmost of node.right
                parent, child = node, node.right
                while child.left is not None:
                    parent, child = child, child.left
                child.left = node.left
                if parent != node:
                    parent.left = child.right
                    child.right = node.right
                node = child
            elif node.left or node.right:
                node = node.left or node.right
            else:
                node = None
        elif key < node.data:
                node.left, deleted = self._delete_value(node.left, key)
        else:
                node.right, deleted = self._delete_value(node.right, key)
        return node, deleted
```

```
array = [40, 4, 34, 45, 14, 55, 48, 13, 15, 49, 47]
bst = BinarySearchTree()
for x in array:
    bst.insert(x)
# Find
print(bst.find(15)) # True
print(bst.find(17)) # False

# Delete
print(bst.delete(55)) # True
print(bst.delete(14)) # True
print(bst.delete(11)) # False
#출처: https://geonlee.tistory.com/72 [빠리의 택시 운전사:티스토리]
```

이진 탐색 트리 한계

- 정렬되어있는 데이터를 이진 탐색 트리로 만들면 리스트랑 다를게 없음.



Binary Search Tree 3

- 한쪽으로 치우친 모양이 나올 수있다. → 비효율적

AVL tree

이진 탐색트리가 한쪽으로 쏠리는 것을 막기위해 스스로 균형을 잡는 트리.

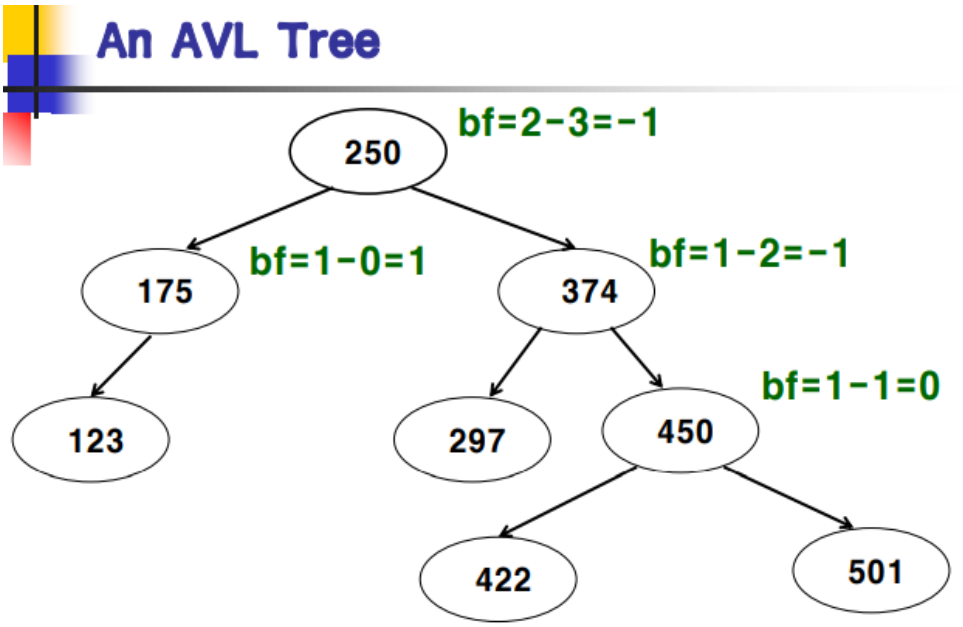
특징

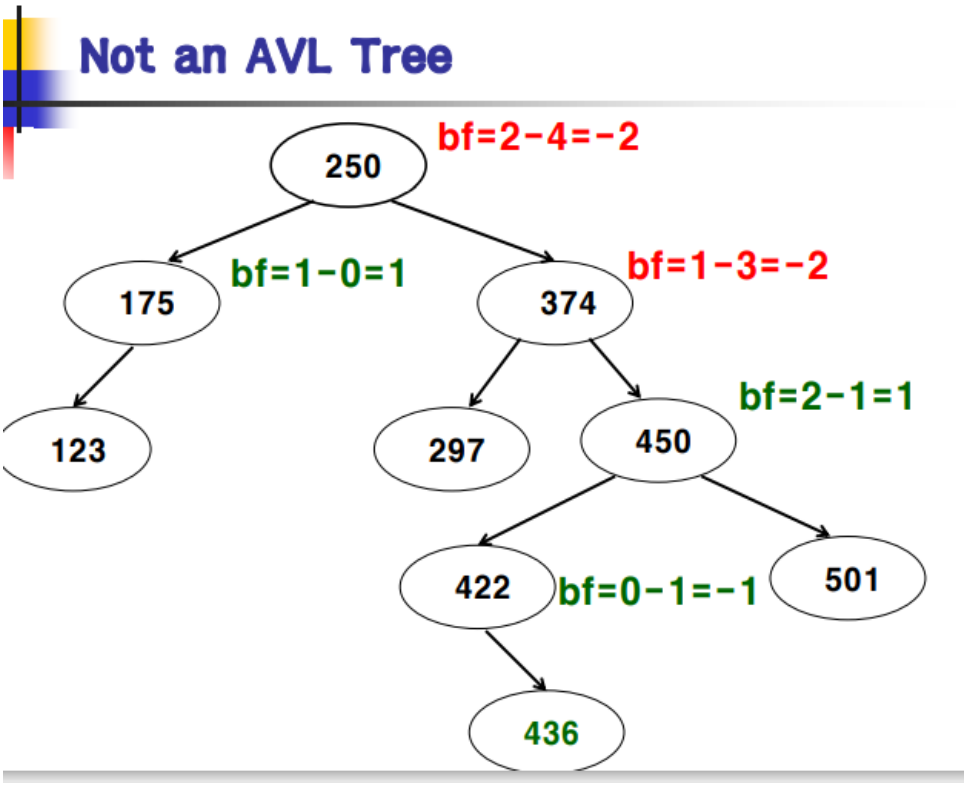
- 높이 균형성질 (height-balance tree)
- AVL트리의 높이는 log n

- 시간 복잡도 O(log n)
- 각 노드를 기준으로 좌우 레벨(높이)의 차가 2이상이 되지 않게 균형을 맞춘다.

BF(Balance Factor)

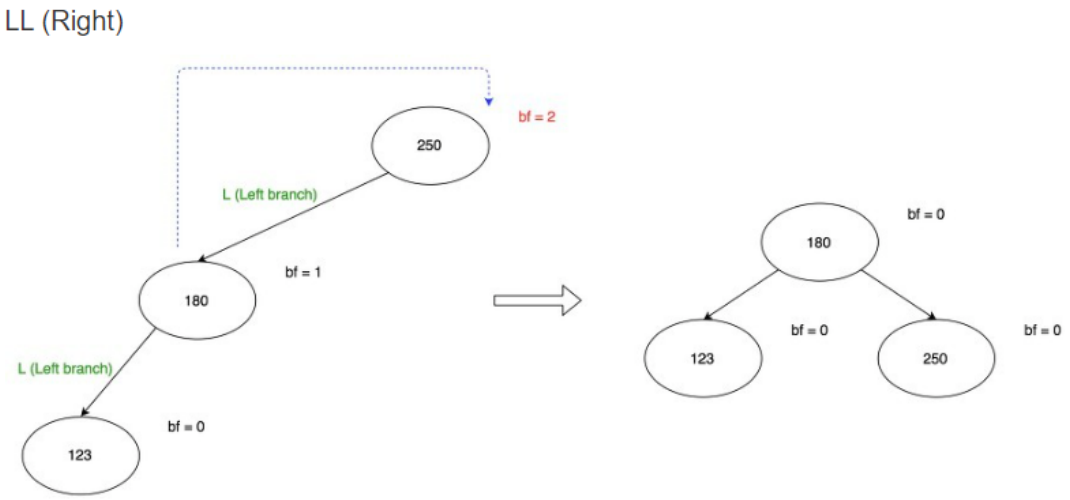
- 특정 노드의 왼쪽 서브트리의 높이에서 오른쪽 서브트리의 높이를 뺀 값.





AVL rotation

- Single Rotation - 회전 한번 만으로 bf에 1이하가 될 때
  - LL

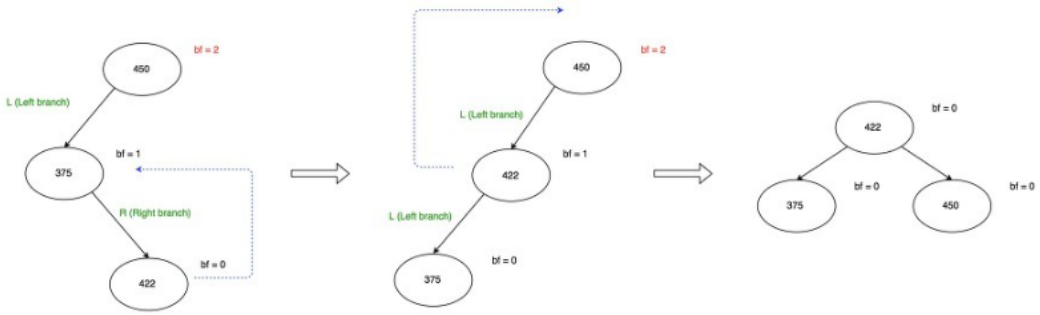



```
def _right_rotate(self, cur):
    v = cur #250
```

```
    w = cur.left #180
    t2 = w.right # none
    cur = w # 180
    w.right = v #250
    v.left = t2 # None
    v.height = 1 + max(self._get_height(v.left), self._get_height(v.right))
    w.height = 1 + max(self._get_height(w.left), self._get_height(w.right))
    return cur
```

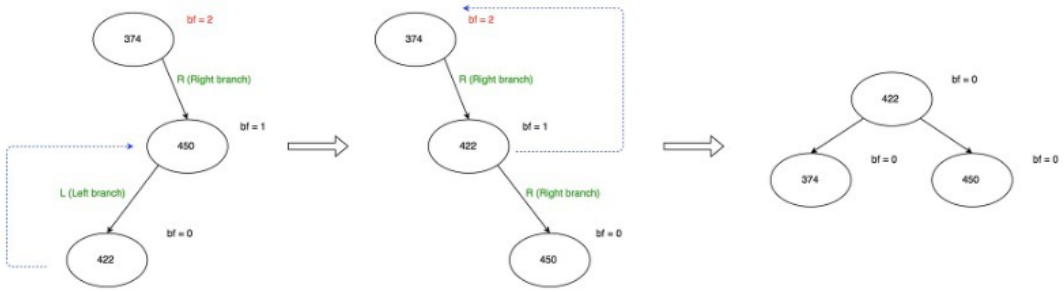
- Double Rotation - single rotation으로도 불균형 상태일때
  - LR

LR(Left & Right)



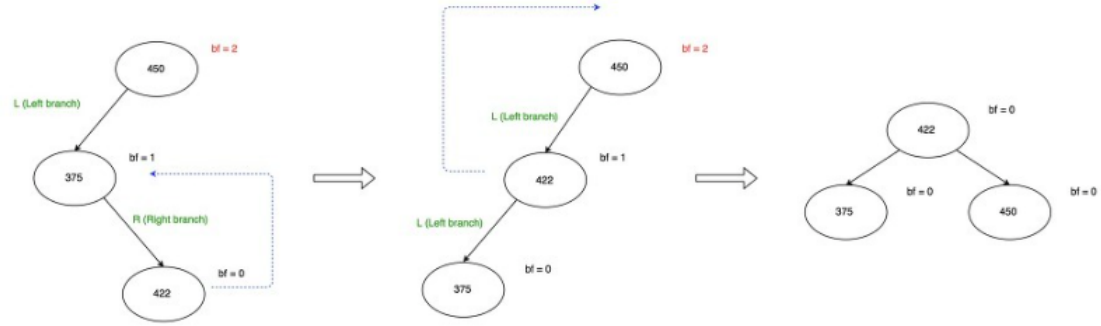
  - RL
  -

RL (Right & Left)



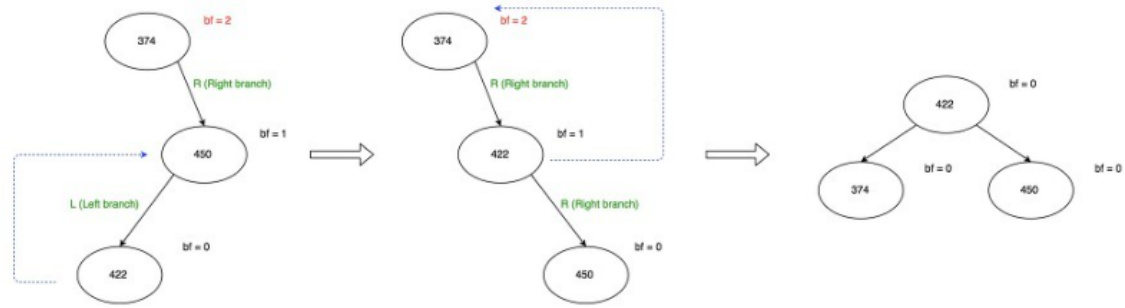
- Double Rotation - single rotation으로도 불균형 상태일때
  - LR

LR(Left & Right)



- RL

RL (Right & Left)



삽입

삽입 방식은 BST와 같다.

노드를 삽입한 뒤 bf를 확인

bf가 2이상인 노드가 생기면 회전을 통해 균형을 맞춘다

```
def __init__(self, val):
    self.root = TreeNode(val)

def insert(self, val):
    self.root = self._insert_node(self.root, val)

def _insert_node(self, cur, val):
    if not cur:
        return TreeNode(val) #노드 생성
    elif val < cur.val: # 왼쪽 노드로 보내기
        cur.left = self._insert_node(cur.left, val)
    elif val > cur.val: # 오른쪽 노드로 보내기
```

```
        cur.right = self._insert_node(cur.right, val)

    cur.height = 1 + max(self._get_height(cur.left),
                         self._get_height(cur.right))
    balance = self._get_balance(cur) #bf값 구하기

    #bf의 절대값이 2이상일 때 경우에 맞게 회전
    if balance > 1 and val > cur.left.val: # Left-Right case
        cur.left = self._left_rotate(cur.left)
        cur = self._right_rotate(cur)

    elif balance > 1 and val < cur.left.val: # Left-Left case
        cur = self._right_rotate(cur)

    elif balance < -1 and val > cur.right.val: # Right-Right case
        cur = self._left_rotate(cur)

    elif balance < -1 and val < cur.right.val: # Right-Left case
        cur.right = self._right_rotate(cur.right)
        cur = self._left_rotate(cur)
    return cur
```

삭제

BST 삭제 과정과 동일

삭제 해주면서 balance를 맞춰 주어야함.

삭제를 한 뒤에 bf가 2이상인 노드는 회전 (LL RR LR RL)

```
def delete(self, val): #val 없애려는 노드의 값
        self.root = self._delete_node(self.root, val)

def _delete_node(self, cur, val):
    if not cur: # 노드가 비었을때 False 반환
        return False
    elif cur == self.root and cur.val == val: # 지우려는 노드가 루트노드일때
        if cur.left and cur.right: # 자식 노드가 2개라면
            pre_val = self._find_predecessor(cur.left) # 왼쪽 노드중 최대 값인 자손 찾기
            self._delete_node(cur, pre_val) #
            cur.val = pre_val
        elif cur.left or cur.right: # 자식이 하나일때
            if cur.left:
                self.root = cur.left
            elif cur.right:
                self.root = cur.right
        else: # 제거하려는 노드를 찾았을 때
            self.root = None

    elif cur.left and cur.left.val == val: # 지우려는 값이 현재 노드의 왼쪽 자식에 있을때
        if cur.left.left and cur.left.right: # 왼쪽자식이 자식을 둘다 가지고 있을때
            pre_val = self._find_predecessor(cur.left.left)
```

```
            self._delete_node(cur, pre_val)
            cur.left.val = pre_val
            cur.left.height = 1 + \
                max(self._get_height(cur.left.left),
                    self._get_height(cur.left.right))
        elif cur.left.left or cur.left.right: # 왼쪽 자식이 자식을 하나만 가지고 있을때
            if cur.left.left:
                cur.left = cur.left.left
            elif cur.left.right:
                cur.left = cur.left.right
            cur.left.height = 1 + \
                max(self._get_height(cur.left.left),
                    self._get_height(cur.left.right))
        else: # 왼쪽 자식이 없을때
            cur.left = None
        cur.height = 1 + max(self._get_height(cur.left),
                             self._get_height(cur.right))

    elif cur.right and cur.right.val == val: #지우려는 값이 현재 노드의 오른쪽 자식에 있을때
        if cur.right.left and cur.right.right:# 오른쪽 자식이 자식을 2개가지고 있을때
            pre_val = self._find_predecessor(cur.right.left)
            self._delete_node(cur, pre_val)
            cur.right.val = pre_val
            cur.right.height = 1 + \
                max(self._get_height(cur.right.left),
                    self._get_height(cur.right.right))
        elif cur.right.left or cur.right.right: # 오른쪽자식이 1개의 자식을 가지고있을때
            if cur.right.left:
                cur.right = cur.right.left
            elif cur.right.right:
                cur.right = cur.right.right
            cur.right.height = 1 + \
                max(self._get_height(cur.right.left),
                    self._get_height(cur.right.right))
        else: #오른쪽 자식이 말단 노드일때 그 노드 삭제
            cur.right = None
        cur.height = 1 + max(self._get_height(cur.left),
                             self._get_height(cur.right))

    elif cur.val > val:
        cur.left = self._delete_node(cur.left, val)

    elif cur.val < val:
        cur.right = self._delete_node(cur.right, val)
```

https://8iggy.tistory.com/111

▼ AVL 전체 코드

```
class TreeNode:
    def __init__(self, val, left=None, right=None, height=1):
        self.val = val
        self.left = left
```

```
        self.right = right
        self.height = height # 높이를 뜻하는 height 속성 추가 기본값=1


class AVLtree:
    def __init__(self, val):
        self.root = TreeNode(val)

    def insert(self, val):
        self.root = self._insert_node(self.root, val)

    def delete(self, val):
        self.root = self._delete_node(self.root, val) # cur,val

    def _insert_node(self, cur, val):
        if not cur:
            return TreeNode(val) #노드 생성
        elif val < cur.val:
            cur.left = self._insert_node(cur.left, val)
        elif val > cur.val:
            cur.right = self._insert_node(cur.right, val)

        cur.height = 1 + max(self._get_height(cur.left),
                             self._get_height(cur.right))
        balance = self._get_balance(cur) #bf값 구하기

        #bf의 절대값이 2이상일 때 경우에 맞게 회전
        if balance > 1 and val > cur.left.val: # Left-Right case
            cur.left = self._left_rotate(cur.left)
            cur = self._right_rotate(cur)

        elif balance > 1 and val < cur.left.val: # Left-Left case
            cur = self._right_rotate(cur)

        elif balance < -1 and val > cur.right.val: # Right-Right case
            cur = self._left_rotate(cur)

        elif balance < -1 and val < cur.right.val: # Right-Left case
            cur.right = self._right_rotate(cur.right)
            cur = self._left_rotate(cur)
        return cur

    def _delete_node(self, cur, val):
        if not cur:
            return False
        elif cur == self.root and cur.val == val:
            if cur.left and cur.right:
                pre_val = self._find_predecessor(cur.left)
                self._delete_node(cur, pre_val)
                cur.val = pre_val
            elif cur.left or cur.right:
                if cur.left:
                    self.root = cur.left
                elif cur.right:
                    self.root = cur.right
            else:
                self.root = None
```

```
        elif cur.left and cur.left.val == val:
            if cur.left.left and cur.left.right:
                pre_val = self._find_predecessor(cur.left.left)
                self._delete_node(cur, pre_val)
                cur.left.val = pre_val
                cur.left.height = 1 + \
                    max(self._get_height(cur.left.left),
                        self._get_height(cur.left.right))
            elif cur.left.left or cur.left.right:
                if cur.left.left:
                    cur.left = cur.left.left
                elif cur.left.right:
                    cur.left = cur.left.right
                cur.left.height = 1 + \
                    max(self._get_height(cur.left.left),
                        self._get_height(cur.left.right))
            else:
                cur.left = None
            cur.height = 1 + max(self._get_height(cur.left),
                                 self._get_height(cur.right))

        elif cur.right and cur.right.val == val:
            if cur.right.left and cur.right.right:
                pre_val = self._find_predecessor(cur.right.left)
                self._delete_node(cur, pre_val)
                cur.right.val = pre_val
                cur.right.height = 1 + \
                    max(self._get_height(cur.right.left),
                        self._get_height(cur.right.right))
            elif cur.right.left or cur.right.right:
                if cur.right.left:
                    cur.right = cur.right.left
                elif cur.right.right:
                    cur.right = cur.right.right
                cur.right.height = 1 + \
                    max(self._get_height(cur.right.left),
                        self._get_height(cur.right.right))
            else:
                cur.right = None
            cur.height = 1 + max(self._get_height(cur.left),
                                 self._get_height(cur.right))

        elif cur.val > val:
            cur.left = self._delete_node(cur.left, val)

        elif cur.val < val:
            cur.right = self._delete_node(cur.right, val)

        balance = self._get_balance(cur)
        # Left-Left case
        if balance > 1 and self._get_balance(cur.left) >= 0:
            cur = self._right_rotate(cur)
        # Left-Right case
        elif balance > 1 and self._get_balance(cur.left) < 0:
            cur.left = self._left_rotate(cur.left)
            cur = self._right_rotate(cur)
        # Right-Left case
        elif balance < -1 and self._get_balance(cur.right) > 0:
```

```
            cur.right = self._right_rotate(cur.right)
            cur = self._left_rotate(cur)
        # Right-Right case
        elif balance < -1 and self._get_balance(cur.right) <= 0:
            cur = self._left_rotate(cur)
        return cur

    def _find_predecessor(self, cur):
        if cur.right:
            return self._find_predecessor(cur.right)
        else:
            return cur.val

    def _left_rotate(self, cur):
        v = cur
        w = cur.right
        t = w.left
        cur = w
        w.left = v
        v.right = t
        v.height = 1 + max(self._get_height(v.left), self._get_height(v.right))
        w.height = 1 + max(self._get_height(w.left), self._get_height(w.right))
        return cur

    def _right_rotate(self, cur):
        v = cur
        w = cur.left
        t2 = w.right
        cur = w
        w.right = v
        v.left = t2
        v.height = 1 + max(self._get_height(v.left), self._get_height(v.right))
        w.height = 1 + max(self._get_height(w.left), self._get_height(w.right))
        return cur
    #높이를 리턴
    def _get_height(self, cur):
        if not cur:
            return 0
        return cur.height
    #왼쪽 높이와 오른쪽 높이의 차를 반환
    def _get_balance(self, cur):
        if not cur:
            return 0
        return self._get_height(cur.left) - self._get_height(cur.right)

    def traverse(self):
        return self._print(self.root, [])

    def _print(self, cur, result):
        if cur:
            self._print(cur.left, result)
            result.append(cur.val)
            self._print(cur.right, result)
        return result

avl = AVLtree(3)
avl.insert(2)
avl.insert(5)
```

```
avl.insert(1)
avl.insert(4)
avl.insert(8)
avl.insert(7)
print(f'root balance: {avl._get_balance(avl.root)}, path: {avl.traverse()}')
avl.delete(4)
print(f'root balance: {avl._get_balance(avl.root)}, path: {avl.traverse()}')
```

그 밖의 트리

heap- 이진 트리의 꽉찬 형태. 말단 노드를 제외한 모든 중간 노드와 루트 노드의 자식이 2개

B-tree - AVL트리와 유사하지만 노드한개안에 데이터가 여러개들어감

공통문제

5639번: 이진 검색 트리

이진 검색 트리는 다음과 같은 세 가지 조건을 만족하는 이진 트리이다. 노드의 왼쪽 서브트리에 있는 모든 노드의 키는 노드의 키보다 작다. 노드의 오른쪽 서브트리에 있는 모든 노드의 키는 노드의 키보다

https://www.acmicpc.net/problem/5639

트리 문제들!

https://www.acmicpc.net/problemset?sort=ac_desc&algo=120