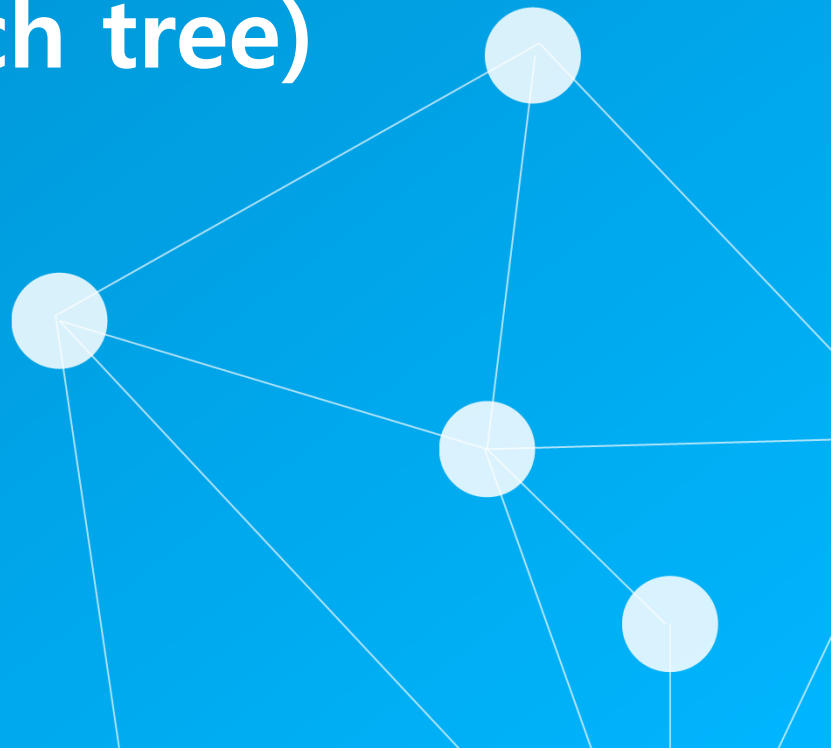

파이썬
자료구조

이진탐색트리
(binary search tree)



탐색트리

- 저장된 n개의 데이터에 대해 탐색, 삽입, 삭제, 갱신 등의 연산을 수행할 수 있는 자료구조
- 1차원 리스트나 연결리스트는 각 연산을 수행하는데 O(n) 시간이 소요
- 스택이나 큐는 특정 작업에 적합한 자료구조.
- 리스트 자료구조의 수행시간을 향상시키기 위한 트리 형태의 다양한 사전 자료구조
  - 이진탐색트리, AVL트리, 2-3트리, 레드블랙트리, B-트리

이진탐색

이진탐색(Binary Search):

정렬된 데이터의 중간에 위치한 항목을 기준으로 데이터를 두 부분으로 나누어 가며 특정 항목을 찾는 탐색방법

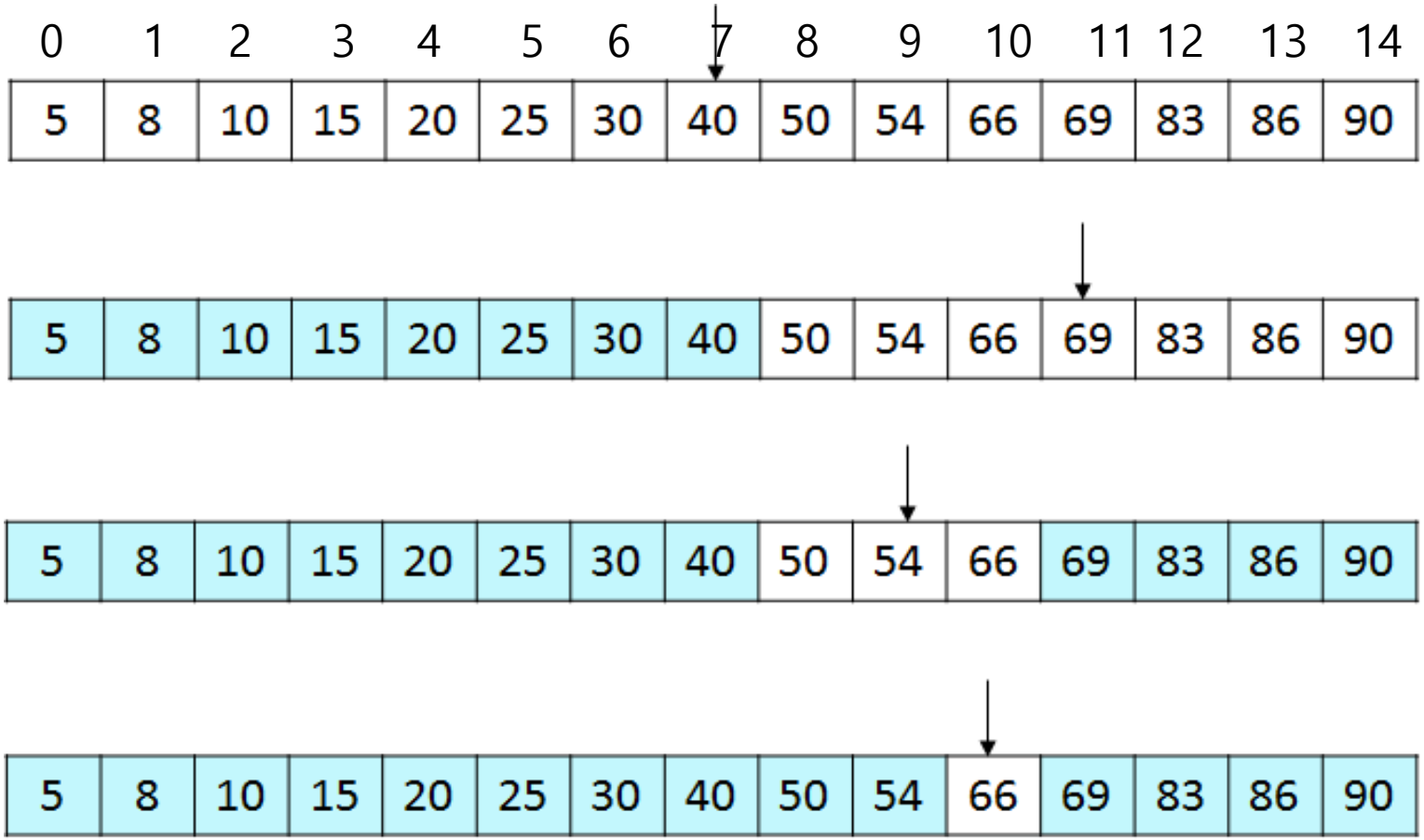
```
binary_search(left, right, t):
[1] if left > right: return None # 탐색 실패 (즉, t가 리스트에 없음)
[2] mid = (left + right) // 2      # 중간 항목의 인덱스 계산
[3] if a[mid] == t: return mid   # 탐색 성공
[4] if a[mid] > t:  binary_search(left, mid-1, t)  # 앞부분 탐색
[5] else: binary_search(mid+1, right, t)           # 뒷부분 탐색
```

이진탐색 (binary search)

```
# 반복 이용
def binarySearch(a, key):  # 반복
    left = 0  # a = [ .....]
    right = len(a)-1

    while left <= right:
        mid = (left + right)//2
#         print(a[mid])
        if key == a[mid]:
            return True  #return mid
        elif key < a[mid]:
            right = mid - 1
        else:
            left = mid + 1
    return False          # return -1
```

```
# 재귀 이용
def binarySearch1(a, key, left, right):
    if left > right:
        return False
    else:
        mid = (left + right)//2
#         print(a[mid])
        if key == a[mid]:
            return True
        elif key < a[mid]:
            return binarySearch1(a, key, left, mid-1)
        else:
            return binarySearch1(a, key, mid+1, right)
```

이진탐색으로 66을 찾는 과정



수행시간

- T(n) = 입력 크기 N인 정렬된 리스트에서 이진탐색을 하는데 수행되는 키 비교 횟수
- T(n)은 1번의 비교 후에 리스트의 1/2, 즉, 앞부분이나 뒷부분을 재귀호출하므로

$$T(n) \le T(n/2) + 1$$

$$T(1) = 1$$

- $T(n) \le T(n/2) + 1$

$\le [T((n/2)/2) + 1] + 1 = T(n/2^2) + 2$

$\le [T((n/2^2)/2) + 1] + 2 = T(n/2^3) + 3$

$\le \cdots \le T(n/2^k) + k$

$= T(1) + k$, if $n = 2^k$, $k = \log_2 n$

$= 1 + \log_2 n = O(\log n)$

탐색트리란?

트리를 구성해서 보다 빠르게 찾기위함

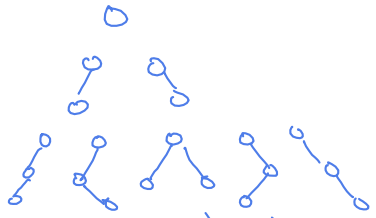
- 탐색트리는 탐색을 위한 트리 기반의 자료구조이다.

B(n) : n개의 노드를 가지는 서로 다른 이진트리의 개수. (B(0) = B(1) = 1)

B(1) = 1

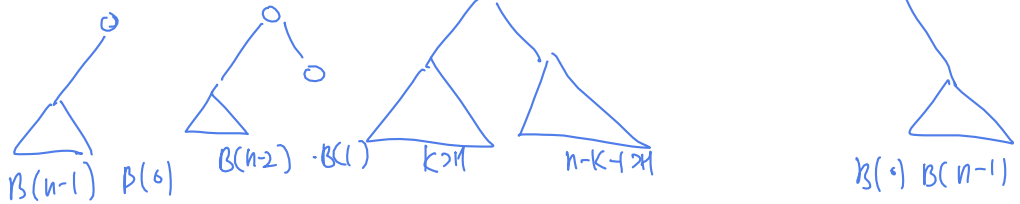
B(2) = 2

B(3) = 5



B(n) = B(n-1)B(0) + B(n-2)B(1) .... + B(k)B(n-k-1) + .. + B(0)B(n-1)

B(4) = B(0)B(3) + B(1)B(2) + B(2)B(1) + B(3)B(0) = 5+2+2+5 = 14



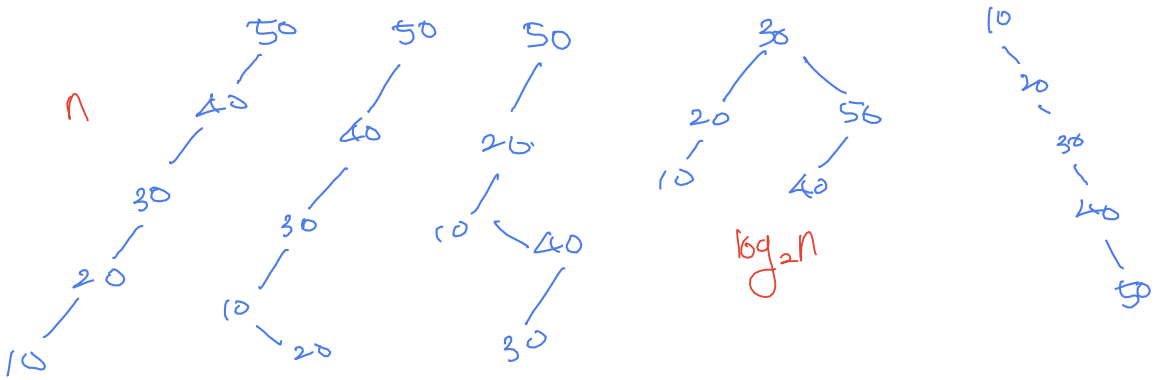
$$B(n) = \frac{2n C_n}{n+1}$$

↓

Catalan Number

주어진 n개 키 ⇒ 이진탐색트리

n= 5 10 20 30 40 50 높이 평균 $O(\log_2 n)$



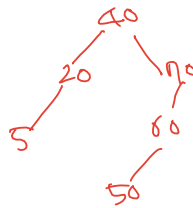
이진 탐색트리

Inorder traverse : 5 20 40 50 60 70

왼쪽 상위의 순서

30 20 5 70 60 40

30 70 60 40 20 5



postorder traverse : 5 20 50 60 70 40

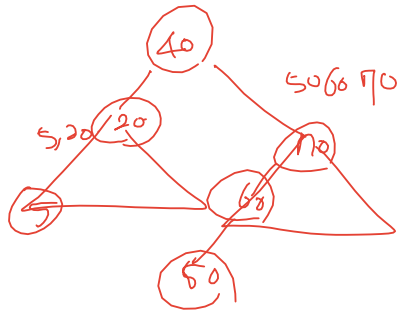
preorder traverse. 40 20 5 70 60 50

시험.

2개만 주어졌을경우

이진탐색트리 만들기

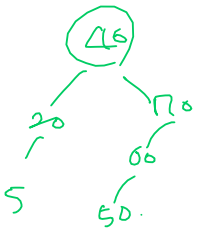
root: postorder traverse의 마지막원소



2개가 주어졌을경우

이진탐색트리 만들기

root: preorder traverse의 첫번째 원소



Expression (수식트리)

수식트리의 결과값 계산 => postorder traversal

- 수식 => 이진트리

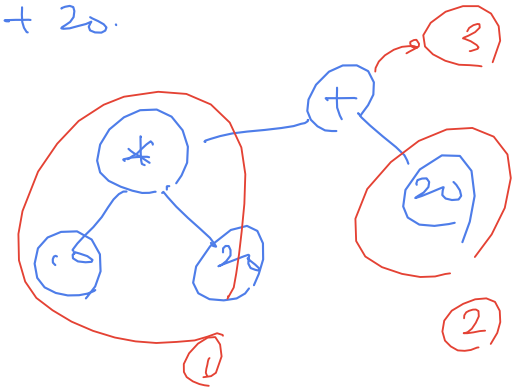
opr1 op opr2 => 루트: op

왼쪽피연산자 / 연산자 / 오른쪽피연산자.

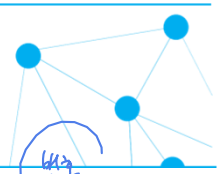
왼쪽 자식 노드 opr1

오른쪽 자식 노드 opr2

10 * 20 + 20.



이진탐색트리란?



ex) 도서 관리

책 : 제목, 저자, 분야

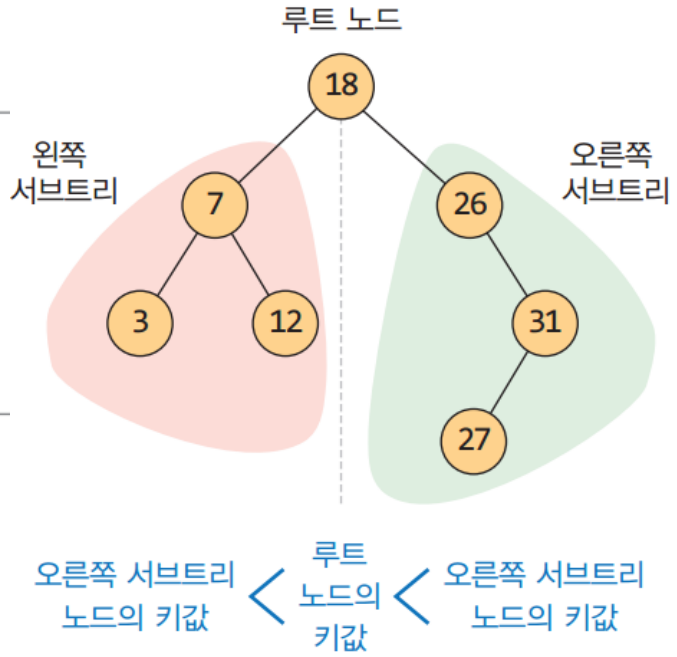
분류를 해서해서 책의 번호를 사용. (키값으로 줄)

맨위 그림과 같이 저장

- 탐색을 위한 트리 기반의 자료구조이다.
- 이진탐색트리
  - 이진트리기반의 탐색을 위한 자료 구조
  - 효율적인 탐색을 위한 이진트리 기반의 자료구조

정의 9.1 이진탐색트리

- 모든 노드는 유일한 키를 갖는다.
- 왼쪽 서브트리의 키들은 루트의 키보다 작다.
- 오른쪽 서브트리의 키들은 루트의 키보다 크다.
- 왼쪽과 오른쪽 서브트리도 이진탐색트리이다. (재귀)



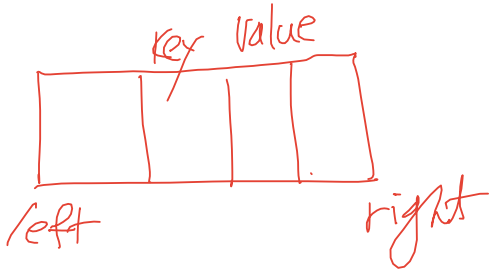
이진탐색트리의 연산

- 이진탐색트리: 노드 구조
- 탐색연산 (search)
- 삽입연산 (insert an element)
- 삭제연산 (delete an element with given key)
- 이진 탐색 트리의 성능 분석

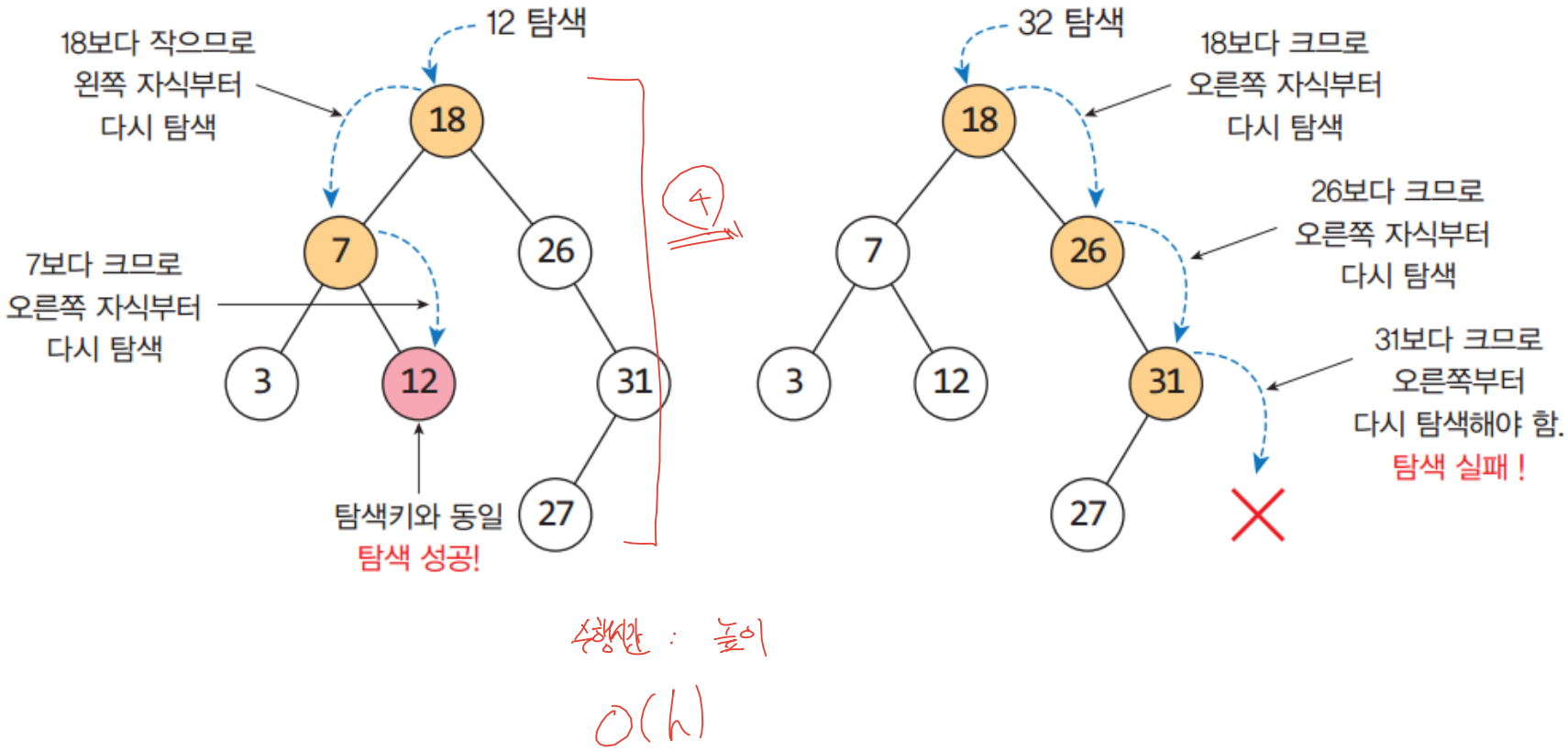
이진탐색트리: 노드 구조

- 노드의 구조
  - (탐색키, 키에 대한 값)의 형태

```
class TreeNode:
    def __init__(self, key, value, left=None, right=None):
        self.key = key          # 키 (key)
        self.value = value      # 값 (value)
        self.left = left
        self.right = right


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



탐색 연산: 키를 이용한 탐색



수행시간 : 높이

O(h)

탐색 연산: 순환과 반복

- 재귀(순환)와 반복 구조로 구현할 수 있음

```
# 반복 구조
    def search1(self, key):
        node = self.root
        while node is not None:
            if key == node.key:
                return node.value
            elif key < node.key:
                node = node.left
            else:
                node = node.right
        return None
```

```
# 재귀(순환) 이용
    def search2(self, key):
        return self._searchBst(self.root, key)

    def _searchSubtree(self, node, key):
        if node is None:
            return None
        elif key == node.key:
            return node.value
        elif key < node.key:
            return self._searchSubtree(node.left, key)
        else:
            return self._searchSubtree(node.right, key)
```

수행시간: O(h)
(h: 이진탐색트리 높이)

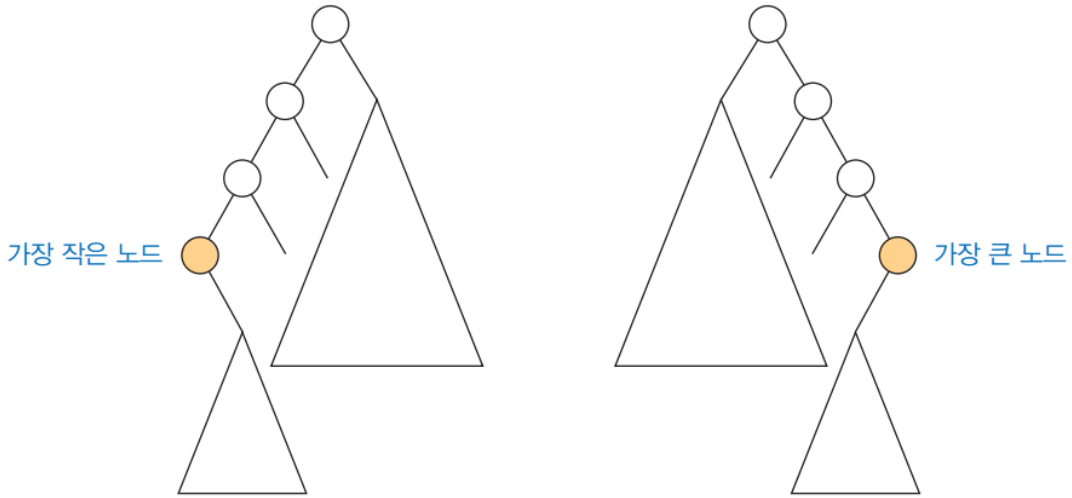
최대 키와 최소 키 찾는 연산

반복 이용

```
def maximum(self):
    node = self.root
    if node is None:
        return None
    while node.right != None:
        node = node.right
    return node.key
```

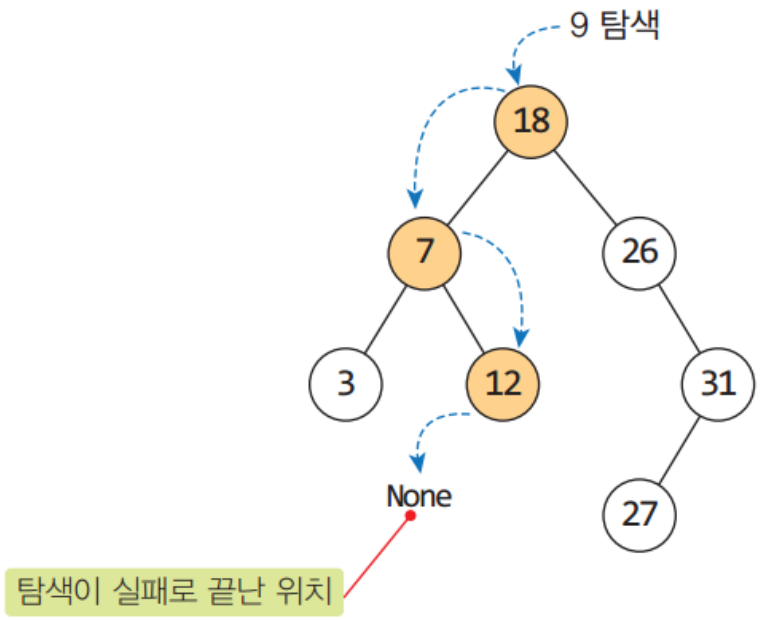
반복 이용

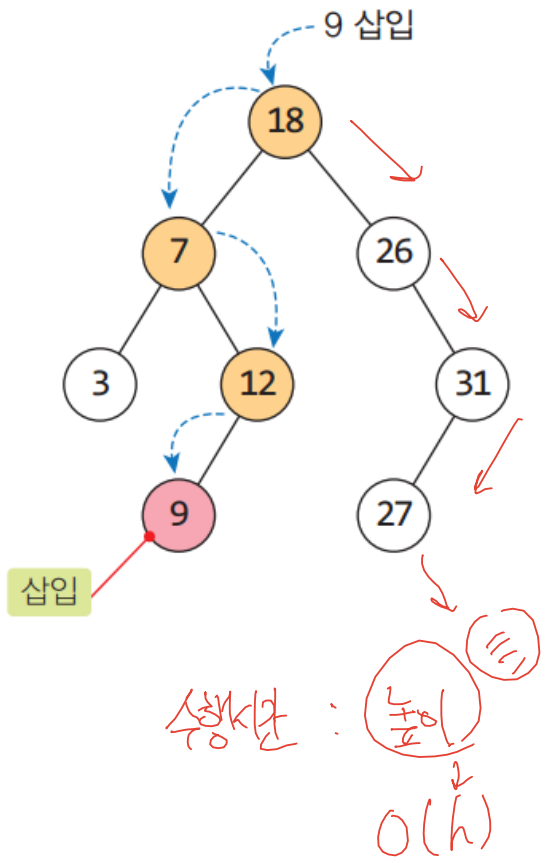
```
def minimum(self):
    node = self.root
    if node is None:
        return None
    while node.left != None:
        node = node.left
    return node.key
```



삽입 연산

- 탐 색에 실패한 위치 ➔ 노드를 삽입해야 하는 위치

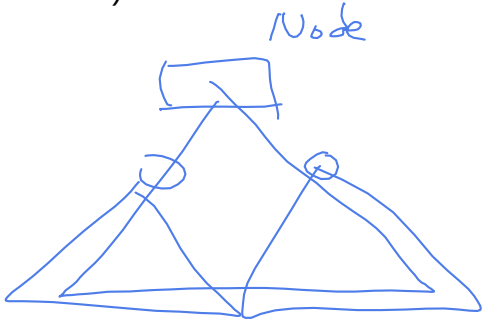




삽입 연산 알고리즘

```
def insert(self, key, value):
        self.root = self._insertSubtree(self.root, key, value)


def _insertSubtree(self, node, key, value): # 원소(key, value)가 저장된 노드를 반환
        if node == None:
            return TreeNode(key, value)
        elif key < node.key:            # 왼쪽 부트리에 노드를 삽입
            node.left = self._insertSubtree(node.left, key, value)
        elif key > node.key:            # 오른쪽 부트리에 노드를 삽입
            node.right = self._insertSubtree(node.right, key, value)
        else:
            pass
        return node
```

삽입한 후 서브트리의 루트를 반환



삭제 연산

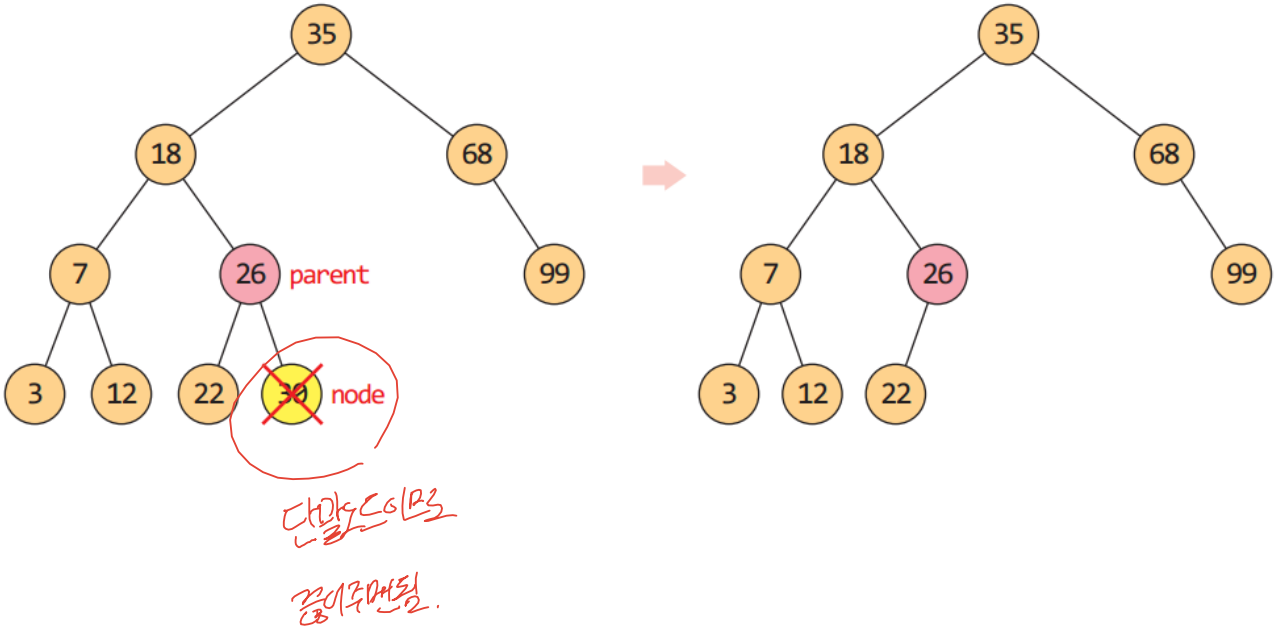
- 노드 삭제의 3가지 경우

  1. 삭제하려는 노드가 단말 노드일 경우

  2. 삭제하려는 노드가 하나의 왼쪽이나 오른쪽 서브 트리중 하나만 가지고 있는 경우 자식노드가

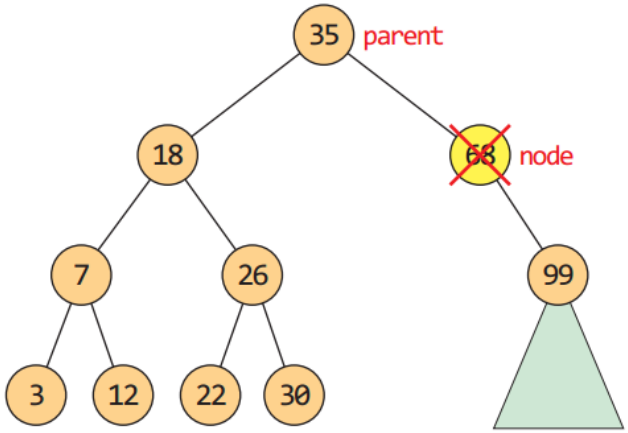
  3. 삭제하려는 노드가 두개의 서브 트리 모두 가지고 있는 경우
     의 자식노드가 두개를 갖음.

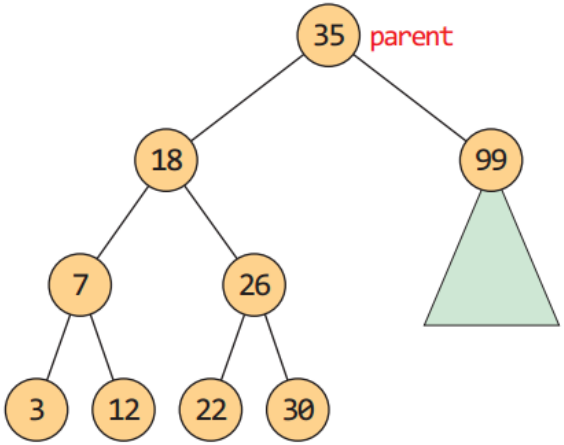
Case 1: 단말 노드 삭제



단말노드이므로
그냥지워짐.

Case2: 자식이 하나인 노드의 삭제

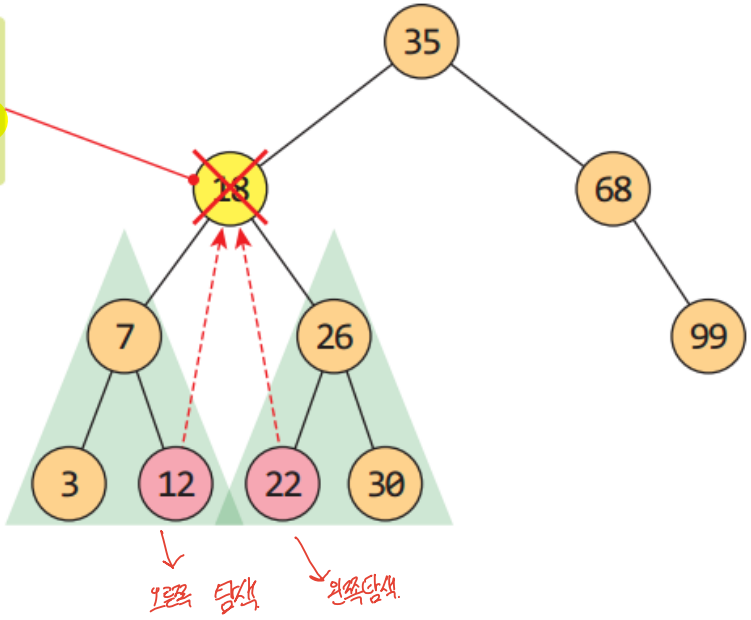




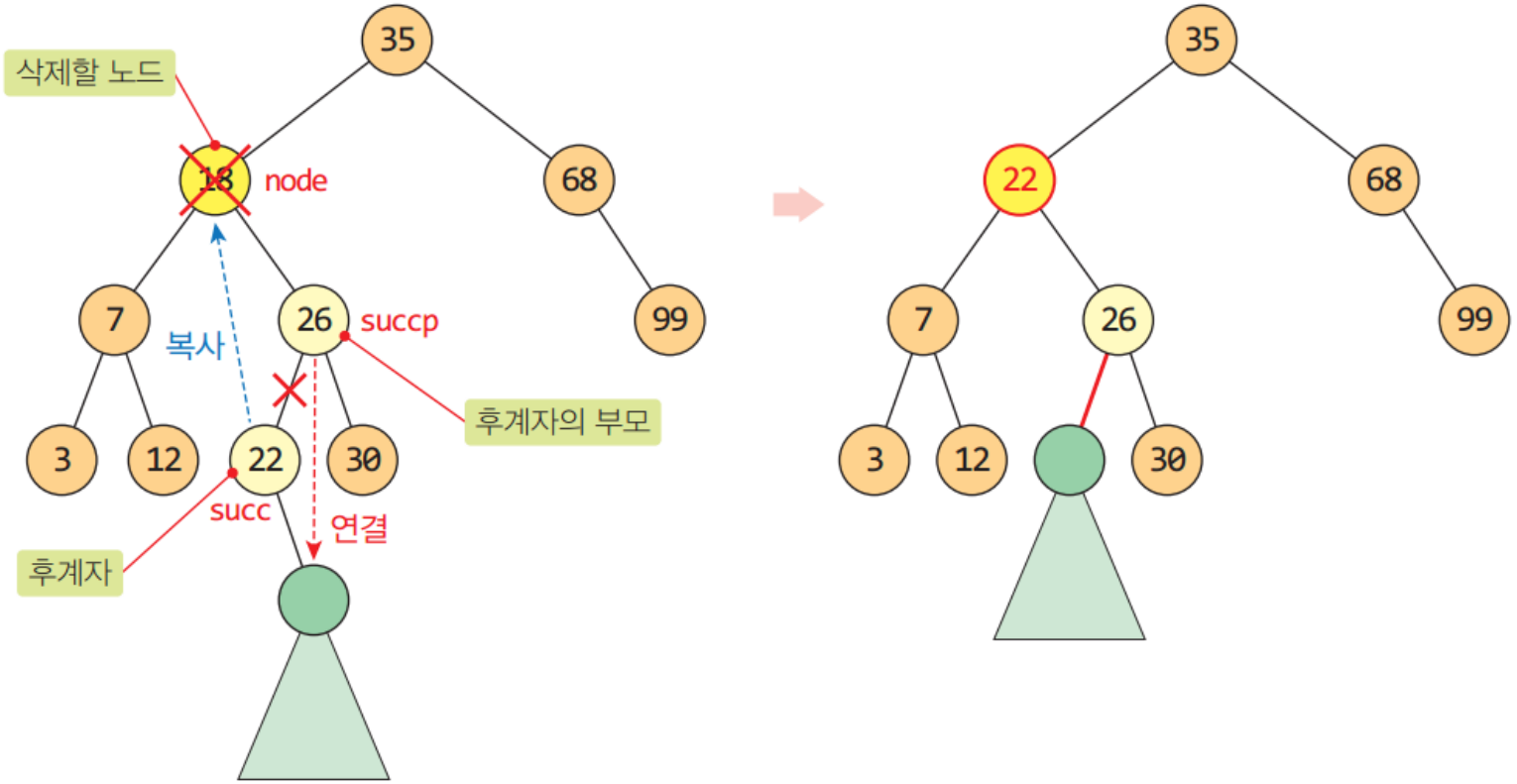
Case 3: 두 개의 자식을 가진 노드 삭제

- 가장 비슷한 값을 가진 노드를 삭제 위치로 가져옴
- 후계 노드의 선택

삭제할 위치에 왼쪽 서브트리의 가장 큰 노드나 오른쪽 서브트리의 가장 작은 노드가 들어가면 이진탐색트리의 조건을 계속 만족한다.



예) 노드 18 삭제


삭제할 노드
35
18
node
68
7
복사
26
succp
99
3
12
22
30
succ
후계자
연결
후계자의 부모
35
22
68
7
26
99
3
12
30

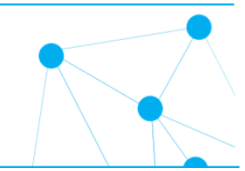
최소키 노드 찾기

```
# 주어진 키의 노드를 삭제하는데 이용
# node가 루트인 이진탐색트리에서 최소키 노드 반환

# 재귀 이용
def _minNode(self, node):
    if node.left == None:
        return node
    else:
        return self._minNode(node.left)

# 반복 이용
def _minNode(self, node):
    if node is None:
        return None
    while node.left != None:
        node = node.left
    return node
```

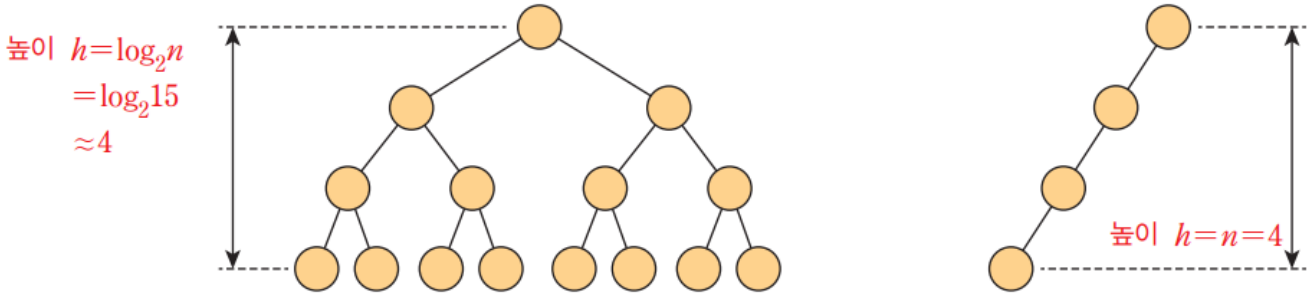
삭제 연산 (주어진 키의 노드 삭제)

```
def delete(self, key):
    self.root = self.deleteNode(self.root, key)

def _deleteNode(self,node,key):   # node가 루트인 이진탐색트리에서 key와 같은 키의 노드 삭제한 후 이진탐색트리의 루트를 반환
    if node == None:
        return None
    if key < node.key:    # 삭제할 키의 노드가 node의 왼쪽 부트리인 경우
        node.left = self._deleteNode(node.left, key)
        return node
    elif key > node.key:  # 삭제할 키의 노드가 node의 오른쪽 부트리인 경우
        node.right = self._deleteNode(node.right, key)
        return node
    else:                 # node가 삭제할 키의 노드인 경우
        if node.right == None:  # node의 오른쪽 자식노드가 없을 경우
            return node.left
        if node.left == None:    # node의 왼쪽 자식노드가 없을 경우
            return node.right

        rightMinNode = self._minNode(node.right)  # node의 오른쪽 부트리에서 최소키의 노드를 찾음
        node.key = rightMinNode.key               # node의 오른쪽 부트리에서 최소키의 노드를 복사 node에 복사
        node.value = rightMinNode.value
        node.right = self._deleteNode(node.right, node.key)   # node의 오른쪽 부트리에서 최소키의 노드를 삭제
        return node
```

이진탐색트리의 성능



- 탐색, 삽입, 삭제 연산의 시간 트리의 높이에 비례함



균형 잡힌 이진트리(포화이진트리)

경사 이진트리

연산	최악의 경우	가장 좋은 경우	평균적인 경우
탐색	O(n)	O(log n)	O(log n)
삽입	O(n)	O(log n)	O(log n)
삭제	O(n)	O(log n)	O(log n)

- N개의 노드가 있는 이진탐색트리의 높이가 가장 낮은 경우는 완전이진트리 형태일 때이고, 가장 높은 경우는 편향이진트리
- 따라서 이진트리의 높이 h는 아래와 같다.

$$\lceil \log (n+1) \rceil = \lfloor log_2 n \rfloor + 1 \approx \log\ n \leq h \leq n$$

- 가장 높이가 가장 이진탐색트리가 만들어질 경우:
  키들이 증가하는 순서대로 원소가 삽입될 경우

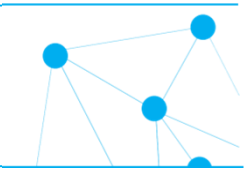
- 비어있는 이진탐색트리에 n개의 키들이 랜덤한 순서대로 삽입한다고 가정했을 때, 트리의 높이는 약 1.39 log n이다

효율적인 이진탐색트리

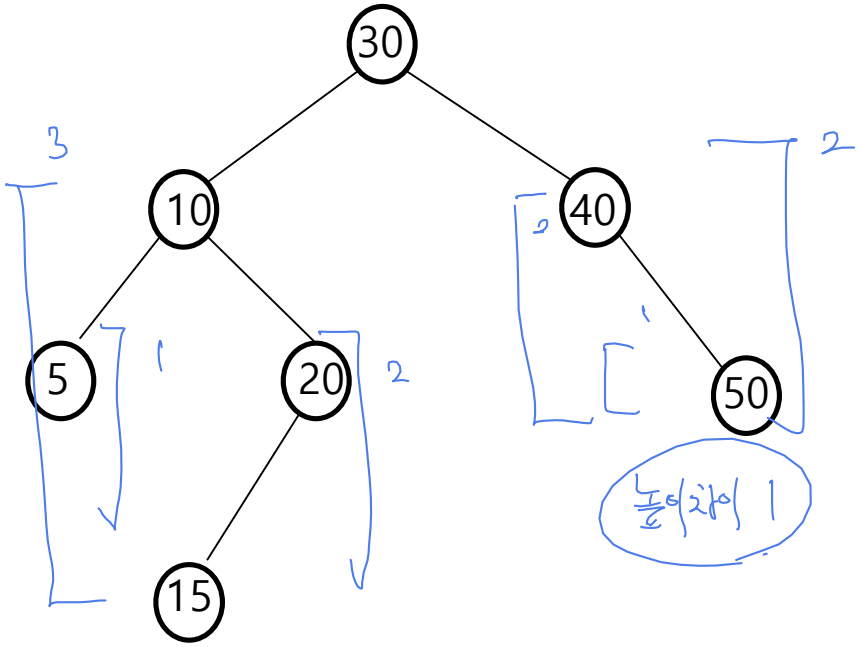
- Balanced binary search tree
  높이를 O(logn)으로 유지함
  - AVL 트리
  - Red-black 트리

연산	최악의 경우
탐색	O(log n)
삽입	O(log n)
삭제	O(log n)

AVL 트리



- 모든 노드의 왼쪽 부트리와 오른쪽 부트리의 높이 차이가 1 이하
- N개의 노드로 이루어진 AVL 트리의 높이 ≤ 1.44 $\log_2 n$



연산	최악의 경우
탐색	O(log n)
삽입	O(log n)
삭제	O(log n)