9차시

```
## 탐색트리란
# 탐색을 위한 트리 기반의 자료구조이다.

## 이진탐색트리
# 효율적인 탐색을 위한 이진트리 기반의 자료구조 이다.
# 사실은 비효율 적이고 해쉬가 효율적인 방법이다.
# 하지만 해쉬가 복잡하여 사용하기 힘드므로 트리를 사용하는 것이다.
# 삽입, 삭제, 탐색 : O(n)
# 완전이진트리로 정렬이 잘되어 있으면 : O(log n)

## 이진탐색트리의 성능
# 탐색, 삽입, 삭제 연산의 시간 트리의 높이에 비례함

# 노드의 구조, Binaray Search Tree Node
class BSTNode: # 이진탐색트리를 위한 노드 클래스
    def __init__(self, key, value): # 생성자 : 키와 값을 받음
        self.key = key
        self.value = value
        self.left = None # 왼쪽 자식에 대한 링크
        self.right = None # 오른쪽 자식에 대한 링크

# 이진탐색트리 탐색연산(순환 함수)
def search_bst(n, key): # 트리와 찾을 키값을 전달
    if n == None: # 현재 노드의 값이 비어있으면
        return None # 없음
    elif key == n.key: # 찾는 key값이 현재 노드의 key 값이면
        return n # 현재 노드 반환
    elif key < n.key: # 찾는 key 값이 현재 노드의 key값보다 작으면 왼쪽 자식 트리를 호출
        return search_bst(n.left, key)
    else: # 찾는 key 값이 현재 노드의 key 값이 크면
        return search_bst(n.right, key)

# 이진탐색트리 탐색연산(반복 함수)
def search_bst_iter(n, key):
    while n != None: # 현재 노드가 비어있을 때까지 반복
        if key == n.key: # key 값이랑 현재 노드의 key 값이랑 동일하면
            return n # 현재 노드 반환
        elif key < n.key: # 현재 노드의 key 값보다 작으면 현재 노드를 왼쪽 자식노드로 재정의
            n = n.left
        else: # 현재 노드의 key 값보다 크면 현재 노드를 오른쪽 자식 노드로 재정의
            n = n.right
    return None # 못찾으면 None

# 이진 탐색 트리 특정값 찾기
def search_value_bst(n, key):
    while n != None: # n이 있는 동안 반복
        if key == n.key: # 찾는 key을 찾으면
            return n.data # data 반복
        elif key < n.key:
            n = n.left
        else:
            n = n.right
    return None # 없으면 None 반환
```

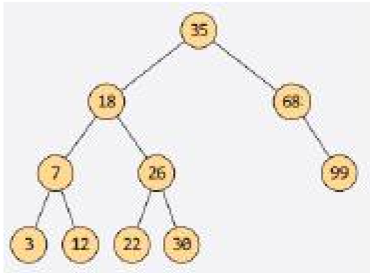
```
# 최대 값의 노드 탐색
def search_max_bst(n):
    while n != None and n.right != None: # 현재 노드가 존재하고, 현재 노드의 오른쪽 자식노드가 존재하면
        n = n.right # 오른쪽
    return n

# 최소값의 노드 탐색
def search_min_bst(n):
    while n != None and n.left != None: # 현재 노드가 존재하고, 현재 노드의 왼쪽 자식노드가 존재하면
        n = n.left
    return n

# 이진 탐색 트리 삽입 연산 (노드를 삽입함): 순환 구조 이용
def insert_bst(r, n): # (root, 삽입할 노드)를 매개변수로 사용
    if n.key < r.key: # 삽입할 노드의 키가 루트보다 작으면
        if r.left is None: # 루트의 왼쪽 자식이 없으면
            r.left = n # n은 루트의 자식이 됨
            return True
        else: # 루트의 왼쪽 자식이 있으면
            return insert_bst(r.left, n) # 왼쪽 자식에게 삽입하도록 함
    elif n.key > r.key: # 삽입할 노드의 키가 루트보다 크면
        if r.right is None: # 루트의 오른쪽 자식이 없으면
            r.right = n # n은 루트의 오른쪽 자식이 됨
            return True
        else: # 루트의 오른쪽 자식이 있으면
            return insert_bst(r.right, n) # 오른쪽 자식에게 삽입하도록 함
    else: # 키가 중복되면
        return False # 삽입하지 않음

# 삭제 알고리즘 1: 삭제하려는 노드가 단말 노드일 경우
def delete_bst_case1(parent, node, root): # (삭제할 노드의 부모노드, 삭제할 노드, 뿌리 노드)를 매개변수로 전달
    if parent is None: # 부모노드가 없다는 것을 뿌리노드 라는 것으로, 뿌리 노드를 None으로 만들어서 초기화시킨다.
        root = None # 공백 트리가 됨
    else: # 삭제할 노드가 뿌리 노드가 아니면
        if parent.left == node: # 삭제할 노드가 부모의 왼쪽 자식이면
            parent.left = None # 부모의 왼쪽 링크를 None
        else: # 오른쪽 자식이면
            parent.right = None # 부모의 오른쪽 링크를 None
    return root # root가 변경될 수 있으므로 반환

# 삭제 알고리즘 2 : 삭제하려는 노드가 하나의 왼쪽이나 오른쪽 서브트리 중 하나만 가지고 있는 경우
def delete_bst_case2(parent, node, root):
    # 삭제하려는 값의 자식노드 확보
    if node.left is not None: # 왼쪽 자식노드가 있으면
        child = node.left # chlid는 왼쪽 자식 노드 저장
    else: # 오른쪽 자식 노드가 있다면
        child = node.right # child에 오른쪽 자식 노드가 저장

    if node == root: # 만약, 뿌리노드를 삭제한다면
        root = child
    else:
        if node is parent.left: # 삭제하려는 노드가 부모의 왼쪽 자식이면
            parent.left = child # 분모의 왼쪽 자식 노드에 chile를 대입
        else: # 삭제하려는 노드가 부모의 오른쪽 자식이면
            parent.right = child # 분모의 오른쪽 자식 노드에 child를 대입
    return root
```

```
# 삭제 알고리즘 3: 삭제하려는 노드가 두개의 서브 트리 모두 가지고 있는 경우
def delete_bst_case3(parent, node, root):
    # 삭제하려는 노드를 기준으로 큰 값들 중 가장 작은 값 저장
    succp = node
    succ = node.right
    while(succ.left != None): # 계속 작은 값을 찾는다.
        succp = succ # 부모 노드 재저장
        succ = succ.left # 부모 노드의 왼쪽 자식 노드 저장

    if (succp.left == succ): # SUCC가 왼쪽 자식이면
        succp.left = succ.right # SUCC의 오른쪽 자식 연결, SUCC는 왼쪽 노드가 없기 때문에
    else: # SUCC가 오른쪽 자식이면
        succp.right =succ.right # SUCC의 왼쪽 자식 연결, SUCC는 왼쪽

    node.key = succ.key # 삭제할 노드의 키에 SUCC의 키를 대입
    node.value = succ.value # 삭제할 노드의 값에 SUCC의 값을 대입
    node = succ; # 노드에 SUCC 링크를 대입

    return root # 일관성을 위해 root 반환

# 이진탐색트리 삭제 연산(노드를 삭제함)
def delete_bst(root, key): # (뿌리노드, 삭제할 key)를 매개변수로 사용
    if root == None: # 공백 트리이면
        return None

    parent = None # 무보 변수 생성
    node = root # 현재 기준 노드에 뿌리 노드 대입
    while node != None and node.key != key: # parent 탐색, 기준 노드가 존재하고 기준노드의 key가 찾는 key가 아닌 경우에 반복
        parent = node # 기준 노드를 부모 노드에 저장
        if key < node.key: # 찾는 키가 기준 노드의 키 값보다 작으면
            node = node.left # 기준 노드에 왼쪽 자식 노드 연결
        else: # 찾는 키가 기준 노드의 키 값보다 크면
            node = node.right # 기준 노드에 오른쪽 자식노드 연결

    if node == None: # 삭제할 노드가 없음
        return None
    if node.left == None and node.right == None: # case 1: 단말 노드
        root = delete_bst_case1(parent, node, root)
    elif node.left == None or node.right == None: # case 2: 유일한 자식
        root = delete_bst_case2(parent, node, root)
    else: # case 3: 두 개의 자식
        root = delete_bst_case3(parent, node, root) # 변경된 루트 노드를 반환
    return root
```

연산	함수	최선의 경우 (균형트리)	최악의 경우 (경사트리)
키를 이용한 탐색	search_bst() search_bst_iter()	$O(\log_2 n)$	$O(n)$
값을 이용한 탐색	search_value_bst()	$O(n)$	$O(n)$
최대/최소 노드 탐색	search_max_bst() search_min_bst()	$O(\log_2 n)$	$O(n)$
삽입	insert_bst()	$O(\log_2 n)$	$O(n)$
삭제	delete_bst()	$O(\log_2 n)$	$O(n)$

```
# 순환을 이용해 트리의 노드 수를 계산하는 함수
def count_node(n):
    if n is None: # n이 None이면 공백 트리 --> 0을 반환
        return 0
    else: # 좌우 서브트리의 노드수의 합 +1을 반환 (순환이용)
        return 1 + count_node(n.left) + count_node(n.right)

# 중위 순회 함수, LVR
def inorder(n):
    if n is not None: # 비어있지 않다면
        inorder(n.left) # 왼쪽 서브트리 처리, L
        print(n.key, end=' ') # 루트노드 처리, V
        inorder(n.right) # 오른쪽 서브트리 처리, R

# 이진탐색트리를 이용한 맵 클래스, Binaray Search Tree
class BSTMap: # 이진탐색트리를 이용한 맵
    def __init__(self): # 생성자
        self.root = None # 트리의 루트 노드

    def isEmpty(self): # 맵 공백 검사
        return self.root == None

    def clear(self): # 맵 초기화
        self.root = None

    def size(self): # 레코드(노드) 수 계산
        return count_node(self.root)

    def search(self, key): # key에 해당하는 노드 반환
        return search_bst(self.root, key)

    def searchValue(self, key): # key에 해당하는 노드의 값을 반환
        return search_value_bst(self.root, key)

    def findMax(self):
        return search_max_bst(self.root)

    def findMin(self):
        return search_min_bst(self.root)

    def insert(self, key, value=None): # 삽입 연산
        n = BSTNode(key, value) # 키와 값으로 새로운 노드 생성
        if self.isEmpty(): # 공백이면
            self.root = n # 루트노드로 삽입
        else: # 공백이 아니면
            insert_bst(self.root, n) # insert_bst() 호출

    def delete(self, key): # delete_bst() 호출
        self.root = delete_bst(self.root, key) # 새로운 트리를 저장

    def display(self, msg = "BSTMap : "):
        print(msg, end='')
        inorder(self.root)
        print()
```

```
if __name__ == "__main__":
    map = BSTMap()
    data = [35, 18, 7, 26, 12, 3, 68, 22, 30, 99]

    print("[삽입 연산] : ", data)
    for key in data:
        map.insert(key)

    map.display("[중위 순회] : ")

    if map.search(26) != None:
        print('[탐색 26 ] : 성공')
    else:
        print('[탐색 26] : 실패')

    if map.search(25) != None:
        print('[탐색 25 ] : 성공')
    else:
        print('[탐색 25] : 실패')

    map.delete(3);
    map.display("[    3 삭제] : ")
    map.delete(68);
    map.display("[   68 삭제] : ")
    map.delete(18);
    map.display("[   18 삭제] : ")
    map.delete(35);
    map.display("[   35 삭제] : ")
```




```
'''
[삽입 연산] :  [35, 18, 7, 26, 12, 3, 68, 22, 30, 99]
[중위 순회] : 3 7 12 18 22 26 30 35 68 99
[탐색 30 ] : 성공
[탐색 25] : 실패
[    3 삭제] : 7 12 18 22 26 30 35 68 99
[   68 삭제] : 7 12 18 22 26 30 35 99
[   18 삭제] : 7 12 22 26 30 35 99
[   35 삭제] : 7 12 22 26 30 99
'''
```

```
## AVL 트리 : 균형이진탐색트리
# 모든 노드에서 왼쪽 서브트리와 오른쪽 서브트리의 높이 차가 1을 넘지 않은 이진탐색트리이다.
# 즉, 모든 노드의 균형 인수는 0이나 +1, -1이 되어야 한다.
# 최악,평균,최선 시간 복잡도 : O(log n)

# 탐색연산 : 이진탐색트리와 동일
# 삽입과 삭제 시 균형 상태가 깨질 수 있음, 따라서 삽입연산을 잘 시행해야함
# 균형이 깨지는 4가지 경우 : LL, LR, RR, RL 타입

from BinarySearchTree import *
from CircularQueue import CircularQueue

def count_node(n): # 순환을 이용해 트리의 노드 수를 계산하는 함수
    if n is None: # n이 None이면 공백 트리 --> 0을 반환
        return 0
    else: # 좌우 서브트리의 노드수의 합 +1을 반환 (순환이용)
        return 1 + count_node(n.left) + count_node(n.right)

def count_leaf(n): # 단말 노드(자식 노드가 없는 노드)의 수
    if n is None: # 공백 트리 --> 0을 반환
        return 0
    elif n.left is None and n.right is None: # 단말 노드이면 1을 반환
        return 1
    else: # 비단말 노드이면 좌우 서브트리의 결과값들을 합한다.
        return count_leaf(n.left) + count_leaf(n.right)

def count_height(n): # 트리의 높이를 구하는 함수
    if n is None: # 공백 트리 --> 0을 반환
        return 0
    hLeft = count_height(n.left) # 왼쪽 트리의 높이 계산
    hRight = count_height(n.right) # 오른쪽 트리의 높이 계산
    if (hLeft>hRight): # 더 높은 높이에 1을 더하여 반환
        return hLeft + 1
    else:
        return hRight + 1

def levelorder(root): # 레벨 순회 함수
    queue = CircularQueue() # 큐 객체 초기화
    queue.enqueue(root) # 최초에 큐에는 루트 노드만 들어있음.
    while not queue.isEmpty(): # 큐가 공백 상태가 아닌 동안 반복
        n = queue.dequeue() # 큐에서 맨 앞의 노드 n을 꺼냄
        if n is not None:
            print(n.key, end=' ') # 먼저 노드의 정보를 출력
            queue.enqueue(n.left) # n의 왼쪽 지식 노드를 큐에 삽입
            queue.enqueue(n.right) # n의 오른쪽 자식 노드를 큐에 삽입
```

```
# LL 회전 방법
def rotateLL(A):
    B = A.left # 시계방향으로 회전
    A.left = B.right
    B.right = A
    return A # 새로운 루트 A를 반환

# RR 회전 방법
def rotateRR(A):
    B = A.right # 반시계방향으로 회전
    A.right = B.left
    B.left = A
    return B # 새로운 루트 B를 반환

# RL 회전 방법
def rotateRL(A):
    B = A.right
    A.right = rotateLL(B) # LL회전
    return rotateRR(A) # RR회전

# LR 회전 방법
def rotateLR(A):
    B = A.left
    A.left = rotateRR(B) # RR회전
    return rotateLL(A) # LL회전

# 균형인수 계산
def calc_height_diff(A):
    if A is None: # 공백 트리 --> 0을 반환
        return 0
    hLeft = count_height(A.left) # 왼쪽 트리의 높이 계산
    hRight = count_height(A.right) # 오른쪽 트리의 높이 계산
    return hLeft - hRight

# 재균형인수 계산
def reBalance(parent): # 부모 노드의 균형 인수 계산, 왼쪽 - 오른쪽
    hDiff = calc_height_diff(parent)
    if hDiff > 1:
        if calc_height_diff(parent.left) > 0:
            parent = rotateLL(parent)
        else:
            parent = rotateLR(parent)
    elif hDiff < -1:
        if calc_height_diff(parent.right) < 0:
            parent = rotateRR(parent)
        else:
            parent = rotateRL(parent)
    return parent
```

```
def insert_avl(parent, node): # (부모노드, 자식노드)
    if node.key < parent.key: # 입력 노드의 키가 부모노드의 키보다 작을 때
        if parent.left != None: # 비어있지 않으면
            parent.left = insert_avl(parent.left, node) # 재귀호국, Parent.left = 재귀(parent.node)
        else:
            parent.left = node
        return reBalance(parent)
    elif node.key > parent.key: # 입력 노드의 키가 부모 노드의 키보다 클 때
        if parent.right != None:
            parent.right = insert_avl(parent.right, node)
        else:
            parent.right = node
        return reBalance(parent)
    else: # 같을 때
        print("중복된 키 에러")

class AVLMap(BSTMap):
    def __init__(self):
        super().__init__()

    def insert(self, key, value = None):
        n = BSTNode(key, value)
        if self.isEmpty():
            self.root = n
        else:
            self.root = insert_avl(self.root, n)

    def display(self, msg="AVLMap : "):
        print(msg, end=' ')
        levelorder(self.root)
        print()

if __name__ == "__main__":
    # node = [7, 8, 9, 2, 1, 5, 3, 6, 4]
    node = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    map = AVLMap()

    for i in node:
        map.insert(i)
        map.display("AVL(%d): "%i)

    print("노드의 개수 = %d"%count_node(map.root))
    print("단말의 개수 = %d"%count_leaf(map.root))
    print("트리의 높이 = %d"%count_height(map.root))
```

```
'''
AVL(7):   7
AVL(8):   7 8
AVL(9):   8 7 9
AVL(2):   8 7 9 2
AVL(1):   8 7 9
AVL(5):   8 7 9 5
AVL(3):   8 7 9
AVL(6):   8 7 9 6
AVL(4):   8 7 9
노드의 개수 = 3
단말의 개수 = 2
트리의 높이 = 2

AVL(0):   0
AVL(1):   0 1
AVL(2):   1 0 2
AVL(3):   1 0 2 3
AVL(4):   1 0 3 2 4
AVL(5):   3 1 4 0 2 5
AVL(6):   3 1 5 0 2 4 6
AVL(7):   3 1 5 0 2 4 6 7
AVL(8):   3 1 5 0 2 4 7 6 8
AVL(9):   3 1 7 0 2 5 8 4 6 9
노드의 개수 = 10
단말의 개수 = 5
트리의 높이 = 4
'''
```