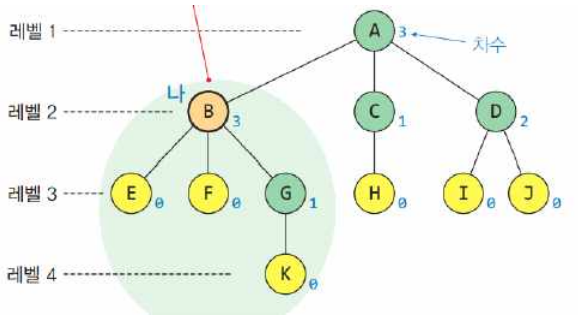
9차시

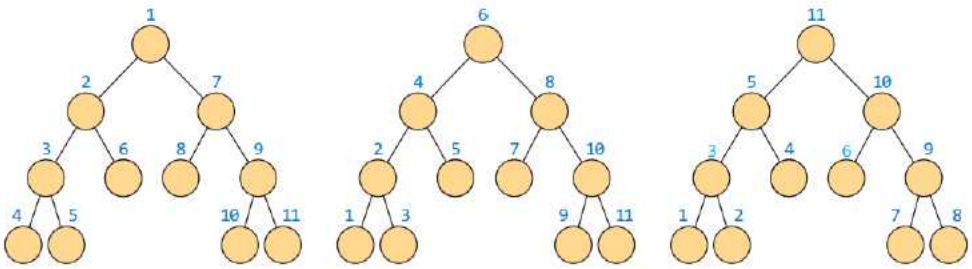
```
## 트리란
# 계층적인 자료의 표현에 적합한 자료구조, 예) 회사 조직도, 컴퓨터의 폴더 구조
# 트리를 표현하는 방법은 많지만 이진트리만 다루도록 한다.

## 트리의 용어
# 루트 노드 : 뿌리 노드
# 간선 또는 에지 : 선
# 부모 / 자식 / 형제 / 조상 / 자손노드
# 단말 / 비단말 노드 : 최종 노드이면 단말 노드이다.
# 노드의 차수 : 생성되는 자식 노드의 수
# 트리의 차수 : 뿌리노드의 차수
# 레벨 : 트리의 층수를 말한다. 뿌리노드가 1레벨
# 트리의 높이 : 트리가 이루는 층의 개수를 말한다.
# 포레스트 : 트리들의 집합
```



- 루트 노드: A
- B의 부모노드: A
- B의 자식 노드: E, F, G
- B의 자손 노드: E, F, G, K
- K의 조상 노드: G, B, A
- B의 형제 노드: C, D
- B의 차수: 3
- 단말 노드: E, F, K, H, I, J
- 비단말 노드: A, B, C, D, G
- 트리의 높이: 4
- 트리의 차수: 3

```
## 이진트리
# 모든 노드가 2개의 서브 트리를 갖는 트리로 서브 트리는 공집합일 수 있고, 순환적으로 정의된다.
# 노드의 개수가 n개면 간선의 개수는 n-1이다.
# 높이가 h이면 각 층의 노드의 수는 h ~ 2^(h-1)개의 노드를 가진다.
# 높이가 h이면 전체 노드의 수는 h ~ 2^h - 1개의 노드를 가진다.
# n개의 노드를 가진 이진 트리의 높이는 [log2(n+1)] ~ n 범위 안에 존재한다.
# 노드 i의 부모 노드 인덱스는 i // 2 , 왼쪽 자식 인덱스는 i*2 , 오른쪽 자식 인덱스는 i*2+1 이다.

## 포화 이진트리
# 트리의 각 레벨에 노드가 꽉 차있는 이진트리
# 노드의 번호
#         1
#    2         3
# 4    5    6    7

## 완전 이진트리
# 높이가 h일때 h-1까지는 꽉 차있고, h번째 레벨에는 순서대로 노드가 채워지는 경우

## 이진트리의 연산
# 순회 : 트리에 속하는 모든 노드를 한 번씩 방문하는 것으로 선형 자료구조는 순회가 단순하다.
# 이진트리의 기본 순회는 전위, 중위, 후위(V가 기준)으로 나뉜다.
# 각 노드의 방문 순서
#                 전위( VLR )                          중위( LVR )                          후위( LRV )
#                  1                  |                  6                  |                  11
#        2                  7         |        4                  8         |        5                  10
#    3        6        8        9     |    2        5        7        10    |    3        4        6        9
# 4    5                    10   11 | 1    3                    9    11 | 1    2                    7    8
# 가운데(V) -> 전반(L) -> 후반(R)    전반(L) -> 가운데(V) -> 후반(R)    전반(L) -> 후반(R) -> 가운데(V)
```

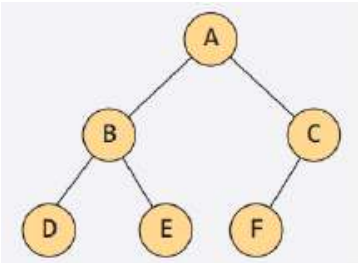
```
from CircularQueue import CircularQueue

# 이진 트리의 노드 표현
class TNode:
    def __init__(self, data, left=None, right=None):
        self.data = data # 값이 들어갈 data
        self.left = left # 왼쪽 자식 노드로 연결하는 링크
        self.right = right # 오른쪽 자식 노드로 연결하는 링크

# 전위 순회 함수의 응용 예) 노드의 레벨 계산, 구조화된 문서 출력
def preorder(n): # 전위 순회 함수, VLR
    if n is not None: # 비어있지 않다면
        print(n.data, end=' ') # 루트노드 처리, V
        preorder(n.left) # 왼쪽 서브트리 처리, L
        preorder(n.right) # 오른쪽 서브트리 처리, R

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

# 후위 순회 함수의 응용 예) 폴더 용량 계산
def postorder(n): # 후위 순회 함수, LRV
    if n is not None: # 비어있지 않다면
        postorder(n.left) # 왼쪽 서브트리 처리, L
        postorder(n.right) # 오른쪽 서브트리 처리, R
        print(n.data, end=' ') # 루트노드 처리, V
```



(a) 전위 순회 (b) 중위 순회 (c) 후위 순회

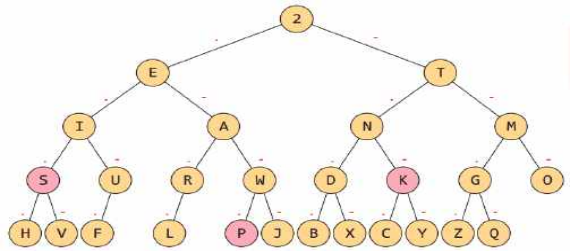
```
# 레벨 순회 함수 : (VRL) 전위 순회 알고리즘을 사용하여 큐를 이용하여 구현, 순환을 사용하지 않음
def levelorder(root): # 레벨 순회 함수
    queue = CircularQueue() # 큐 객체 초기화, 큐 -> (FIFO, LILO)
    queue.enqueue(root) # 최초에 큐에는 루트 노드만 들어있음. 루트노드를 처음으로 추가
    while not queue.isEmpty(): # 큐가 공백 상태가 아닌 동안 반복, 큐에 데이터가 없을 때까지 반복
        n = queue.dequeue() # 큐에서 맨 앞의 노드 n을 꺼냄, 큐에서 데이터를 꺼냄
        if n is not None: # 꺼낸 데이터가 비어있으면, 즉 최종노드이면 넘어감
            print(n.data, end=' ') # 먼저 노드의 정보를 출력, V
            queue.enqueue(n.left) # n의 왼쪽 지식 노드를 큐에 삽입, L
            queue.enqueue(n.right) # n의 오른쪽 자식 노드를 큐에 삽입, R
```

```
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) # 순환을 사용한다.

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
# 본문
if __name__ == "__main__":
    # 후위순회 순서대로 데이터를 입력
    d = TNode('D',None,None)
    e = TNode('E',None,None)
    b = TNode('B', d, e)
    f = TNode('F',None,None)
    c = TNode('C', f,None)
    root = TNode('A', b, c)

    print('\n      In-Order : ', end='')
    inorder(root) # 중위순회 LVR
    print('\n    Pre-Order : ', end='')
    preorder(root) # 전위순회 vLR
    print('\n   Post-Order : ', end='')
    postorder(root) # 추위 순회 LRV
    print('\n Level-Order : ', end='')
    levelorder(root) # 레벨 순회
    print()

    print(" 노드의 개수 = %d개"%count_node(root))
    print(" 단말의 개수 = %d개"%count_leaf(root))
    print(" 트리의 높이 = %d개"%count_height(root))

'''
     In-Order : D B E A F C
    Pre-Order : A B D E C F
   Post-Order : D E B F C A
  Level-Order : A B C D E F
  노드의 개수 = 6개
  단말의 개수 = 3개
  트리의 높이 = 3개
'''
```



```
## 모르스(모스) 코드 결정 트리
# 모르스(모스) 부호 : 점과 선의 조합으로 구성된 메시지 전달용 부호

## 모르스 부호 표
# [('A', '.-'), ('B', '-...'), ('C', '-.-.'), ('D', '-..'), ('E', '.'), ('F', '..-.'), ('G', '--.'), ('H', '....'),
#  ('I', '..'), ('J', '.---'), ('K', '-.-'), ('L', '.-..'), ('M', '--'), ('N', '-.'), ('O', '---'), ('P', '.--.'),
#  ('Q', '--.-'), ('R', '.-.'), ('S', '...'), ('T', '-'), ('U', '..-'), ('V', '...-'), ('W', '.--'), ('X', '-..-'),
#  ('Y', '-.--'), ('Z', '--..')]

## 인코딩 : 알파벳에서 모르스 코드로 변환, O(1) : 표에서 바로 찾으므로
## 디코딩 : 표에서 순차 탐색, O(n) : 표에서 탐색을 하므로
# 따라서 디코딩의 방법을 개선하기 위해 결정트리를 가져왔다.
# 결정 트리 : 여러 단계의 복잡한 조건을 갖는 문제에 대해 조건과 그에 따른 해결방법을 트리 형태로 나타낸 것
# 디코딩 시간 복잡도 O(n)

class TNode:
    def __init__(self, data, left, right):
        self.data = data
        self.left = left
        self.right = right

def make_morse_tree(): # 모수 부호코드를 결정 트리로 변환 구축하는 함수. 코드를 통해 경로를 찾고, 최종적으로 찾은 곳에 값을 이입
    root = TNode(None, None, None) # 뿌리노드 만들기
    for tp in table: # 모스 부호 table에서 하나씩 가져오기
        code = tp[1] # 모르스 코드, tp[0]는 문자를 뜻함
        node = root # 현재 뿌리 노드를 저장 해둠
        for c in code: # 모스 부호 코드를 처음부터 반환
            if c == '.': # '.' 문자이면
                if node.left == None: # 왼쪽 자식 노드가 비었으면 빈 노드 만들기
                    node.left = TNode(None, None, None)
                node = node.left # 왼쪽으로 이동
            elif c == '-': # 오른쪽으로 이동하면 -문자가 추가된다.
                if node.right == None: # 오른쪽 자식 노드가 비었으면
                    node.right = TNode(None, None, None) # 빈 노드 만들기
                node = node.right # 오른쪽으로 이동
        node.data = tp[0] # 최종적으로 찾은 곳에 해당 알파벳을 대입
    return root

def decode(root, code): # 모스코드 -> 알파벳, 매개변수로 모스부호 트리와 대코딩 할 모스 부호를 전달
    node = root # 뿌리 노드를 전달
    for c in code: # 디코딩할 모스부호에서 문자를 처음부터 가져오기
        if c =='.': # '.'이면 왼쪽으로 이동
            node = node.left
        elif c=='-': # '-'이면 오른쪽으로 이동
            node = node.right
    return node.data # 최종적으로 위치한 노드에 data, 즉 알파벳을 반환

def encode(ch): # 알파벳 -> 모스부호, 모스부호 테이블 사용
    idx = ord(ch) - ord('A') # 기준이 되는 문자A와의 차이를 이용하여 해당 문자의 노드 번호를 추출
    return table[idx][1] # 모스부호 테이블을 이용하여 해당 노드 번호의 알파벳을 반환
```

```
# 본문
if __name__ == "__main__":

    # 모스 부호 코드 테이블, 리스트 안에 튜플로 값을 넣어 선언
    table = [('A', '.-'), ('B', '-...'), ('C', '-.-.'), ('D', '-..'),
             ('E', '.'), ('F', '..-.'), ('G', '--.'), ('H', '....'),
             ('I', '..'), ('J', '.---'), ('K', '-.-'), ('L', '.-..'),
             ('M', '--'), ('N', '-.'), ('O', '---'), ('P', '.--.'),
             ('Q', '--.-'), ('R', '.-.'), ('S', '...'), ('T', '-'),
             ('U', '..-'), ('V', '...-'), ('W', '.--'), ('X', '-..-'),
             ('Y', '-.--'), ('Z', '--..')]
```




```
    morseCodeTree = make_morse_tree() # morseCodeTree에 결정 트리를 생성한다.
    str = input("입력 문장 : ")
    mlist = [] # 인코딩된 모스부호를 저장할 공간
    for ch in str : # 입력한 문자열이 끝날때까지 반복
        code = encode(ch) # 해당 알파벳을 모스부호로 인코딩
        mlist.append(code) # 인코딩한 알파벳 저장
    print("Morse Code: ", mlist) # 인코딩된 모스부호를 나열
    print("Decoding: ",end='')

    for code in mlist: # 인코딩된 모스부호를 하나씩 가져오기
        ch = decode(morseCodeTree, code) # 절정 트리와 모스 부로를 이용하여 다시 알파벳으로 디코딩
        print(ch, end='') # 디코딩한 값을 출력한다.
    print()

'''
입력 문장 : GAMEOVER
Morse Code:  ['--.', '.-', '--', '.', '---', '...-', '.', '.-.']
Decoding: GAMEOVER
'''
```

```
## 힙 트리

## 힙
# '더미'와 모습이 비슷한 완전 이진트리 기반의 자료구조로 가장 크거나 작은 값을 빠르게 찾아내도록 만들어진 구조이다.
# '더미'란 :
# 최대힙 : 부모 노드의 키값이 자식 노드의 키 값보다 크거나 같은 완전이진트리
# 최소힙 : 부모 노드의 키 값이 자식 노드의 키 값보다 작거나 같은 완전이진트리
# 즉, 완전 트리가 아니면 힙이 아님
# 완전 이진트리: 높이가 h일때 h-1까지는 꽉 차있고, h번째 레벨에는 순서대로 노드가 채워지는 경우
# 힙을 저장하는 효과적인 자료구조는 배열이다. 인덱스 0은 사용하지 않으며, 완전이진트리이므로 중간에 빈 칸이 없다.
# 우선순위 큐의 가장 좋은 구현 방법은 힙이다.
# 우선순위 큐 -> 힙 -> 배열

## 부모 노드와 자식 노드간의 인덱스 관계
# 노드 i의 부모 노드 인덱스는 i // 2 , 왼쪽 자식 인덱스는 i*2 , 오른쪽 자식 인덱스는 i*2+1 이다.
# 노드의 개수가 n개면 간선의 개수는 n-1이다.
# 높이가 h이면 층의 노드는 0 ~ 2^(h-1)개의 노드를 가진다.
# 높이가 h이면 전체 노드는 h ~ 2^(h-1)개의 노드를 가진다.
# n개의 노드를 가진 이진 트리의 높이는 [log2(n+1)] ~ n 범위 안에 존재한다.

## Upheap : 추가가 되는 구조로 말단에서 위로 올라가는 구조
# 시간 복잡도 O(log n) : 루트 노드까지 올라가야 하므로 트리 높이에 해당하는 이동이 필요
## Downheap : 삭제가 되는 구조로 루트 노드가 삭제되며, 마지막 레벨의 마지막 노드를 루트노드로 올려서 노드를 내리는 구조
# 부모의 교환이 끝나면 마지막 층의 순서를 재배열 한다.
# 시간 복잡도 O(log n) : 가장 아래 레벨까지 내려가야 하므로 역시 트리의 높이 만큼의 시간이 걸린다.

class MaxHeap: # 최대힙 클래스
    def __init__(self): # 생성자
        self.heap = [] # 리스트(배열)를 이용한 힙
        self.heap.append(0) # 0번 항목은 사용하지 않음 -> 부모, 자식 간에 인덱스를 찾기 위해서

    def size(self): # 힙의 크기
        return len(self.heap) -1 # 0 인덱스는 사용하지 않으므로 -1이 필요

    def isEmpty(self): # 공백 검사
        return len(self.heap) == 0

    def Parent(self, i): # 부모 노드 값 반환
        return self.heap[i//2] # 정수의 나누기므로 // 사용

    def Left(self, i): # 왼쪽 자식 노드 값 반환
        return self.heap[i*2]

    def Right(self, i): # 오른쪽 자식 노드 값 반환
        return self.heap[i*2+1]

    def display(self, msg='힙트리: '): # 출력 형식 지정
        print(msg, self.heap[1:]) # 리스트의 슬라이싱 이용, 0번 인덱스는 사용하지 않으므로 [1:] 사용
```

```
    def insert(self, n): # up-heap
        self.heap.append(n) # 맨 마지막 노드로 일단 삽입
        i = self.size() # 노드 n의 위치, 맨뒤에 있으므로 총길이만큼의 인덱스가 부여
        while (i != 1 and n > self.Parent(i)): # 삽입된 노드가 맨 처음이 아니거나 부모노드 값보다 크면 반복
            self.heap[i] = self.Parent(i) # 자식이 부모보다 크므로 부모의 값을 자식에 대입
            i = i // 2 # 부모의 인덱스로 초기화, 자식의 값을 부모에 저장하기 위해, 현재 사용중인 인덱스(자식)를 부모 인덱스로 변환
        self.heap[i] = n # 부모 인덱스에 삽입하려는 값을 데입, 스위칭의 알고리즘이다.

    def delete(self): # down-heap
        parent = 1
        child = 2
        if not self.isEmpty():
            hroot = self.heap[1] # 뿌리노드 값을 저장
            last = self.heap[self.size()] # 마지막 노드 값을 저장
            while (child <= self.size()): # 마지막 인덱스까지 반복(child)
                # 인덱스를 벗어나지 않고 형제 노드간에 오른쪽 값이 더 크면
                # 무모노드와 비교하고 값을 바꾸기 위해 자식 노드 중에서 큰 값을 찾는 과정
                if child < self.size() and self.Left(parent) < self.Right(parent):
                    child += 1
                if last >= self.heap[child]: # 마지막 노드값이 현재 기준 노드(child) 값보다 크면
                    break; # 삽입 위치를 찾음. down-heap 종료
                self.heap[parent] = self.heap[child] # 아니면 down-heap 계속
                parent = child
                child *= 2;

            self.heap[parent] = last # last값이 parent에 해당하는 값보다 크면 삽입 자리를 찾은것이므로 last를 대입
            self.heap.pop(-1) # 맨 마지막 노드 삭제
            return hroot # 저장해두었던 투르를 반환

if __name__ == "__main__":
    heap = MaxHeap() # MaxHeap 객체 생성
    data = [2,5,4,8,9,3,7,3] # 힘에 삽입할 데이터
    print("[삽입 연산] : " + str(data))
    for elem in data: # 모든 데이터를
        heap.insert(elem) # 힙에 삽입
    heap.display('[삽입 후]  : ') # 현재 힙 크리를 출력
    heap.delete() # 한번의 삭제 연산
    heap.display('[삭제 후]  : ') # 현재 힙 트리를 출력
    heap.delete() # 또 한번의 삭제 연산
    heap.display('[삭제 후]  : ') # 현재 힙 트리를 출력

'''
[삽입 연산] : [2, 5, 4, 8, 9, 3, 7, 3]
삽입과정
2
5 2
5 2 4
8 5 4 2
9 8 4 2 5
9 8 4 2 5 3
9 8 4 2 5 3 4
9 8 4 2 5 3 4 2
[삽입 후]  :  [9, 8, 7, 3, 5, 3, 4, 2]
[삭제 후]  :  [8, 5, 7, 3, 2, 3, 4]
[삭제 후]  :  [7, 5, 4, 3, 2, 3]
'''
```