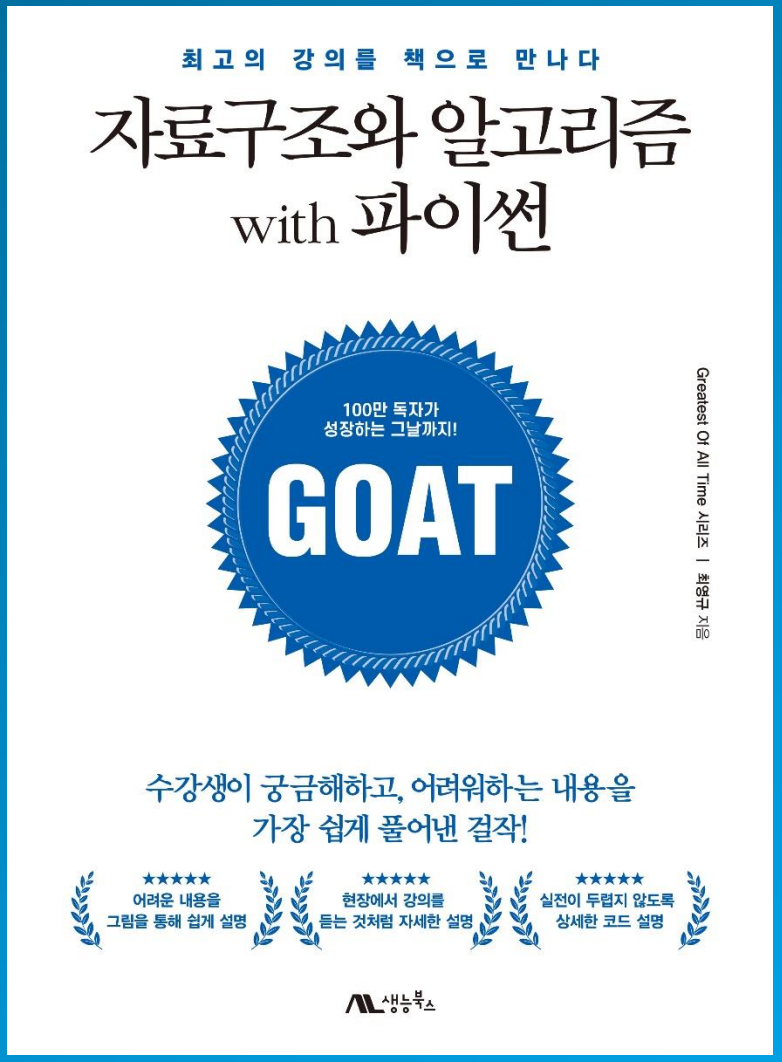
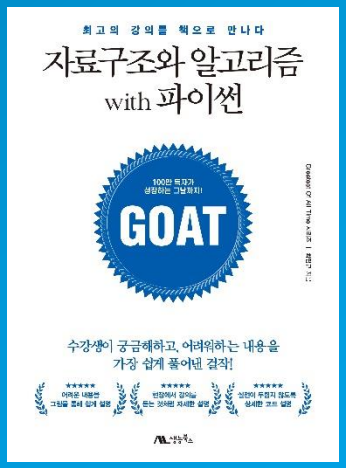



# SW알고리즘개발

**8. 주차별 강의계획(Weekly Schedule)**

| 주 (Week) | 강의주제 (Lecture subject) | 강의 세부내용 (Detailed contents) |
|---|---|---|
| 1 | 스택 | 이론 (1시간): 스택의 개념, 배열 구조로 스택 구현.<br>실습 (3시간):<br>실습환경 설정: Github 계정, Python 설치 및 통합 개발 환경(IDE) 설정.<br>파이썬으로 스택 구현, 괄호 검사, 시스템 스택과 순환 호출 실습. |
| 2 | 큐 | 이론 (1시간): 큐의 개념, 배열로 큐 구현, 덱의 개념.<br>실습 (3시간): 상속을 이용한 덱 구현, 파이썬에서 큐와 덱 사용 실습. |
| 3 | 리스트 | 이론 (1시간): 리스트의 개념, 배열 구조와 연결된 구조.<br>실습 (3시간): 파이썬 리스트 활용, 단순 및 이중 연결 리스트 구현 실습. |
| 4 | 트리 | 이론 (1시간): 트리의 개념, 이진 트리 구조와 연산.<br>실습 (3시간): 모스 코드 결정 트리 및 수식 트리 구현 실습. |
| 5 | 알고리즘 개요 | 이론 (1시간): 알고리즘의 정의, 중요성, 성능 분석 기법.<br>실습 (3시간): 다양한 알고리즘의 시간 복잡도 분석 및 구현 실습. |
| 6 | 정렬 | 이론 (1시간): 정렬의 개념, 선택 정렬, 삽입 정렬, 퀵 정렬, 기수 정렬.<br>실습 (3시간): 파이썬 정렬 함수 활용 및 정렬 알고리즘 구현 실습. |
| 7 | 탐색 | 이론 (1시간): 탐색의 개념, 순차 탐색, 이진 탐색.<br>실습 (3시간): 이진 탐색 트리 구현 및 탐색 알고리즘 실습. |




# 04 CHAPTER

# 트리

## 4장. 트리

04-1 트리란?

04-2 이진 트리

04-3 이진 트리의 연산

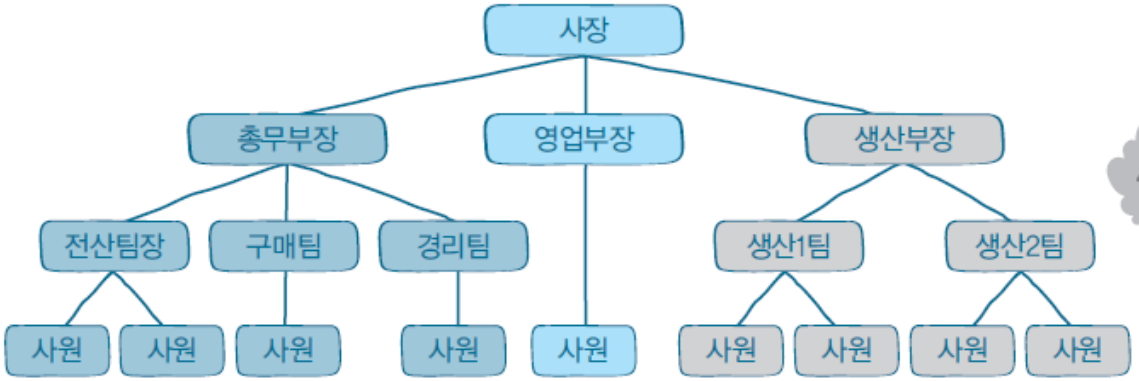
04-4 모스 코드 결정 트리

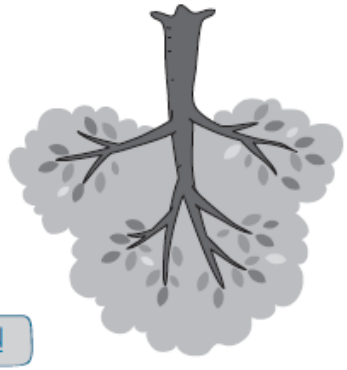
04-5 수식 트리

# 4.1 트리(Tree)란?

- 트리는 나무 모양의 자료구조
  - 계층적인 관계를 가진 자료의 표현에 매우 유용
  - 노드와 간선의 연결 관계
  - 순환적으로 정의되는 자료구조
  - 순환 호출을 사용하는 순환알고리즘 사용
  - 비선형 자료구조

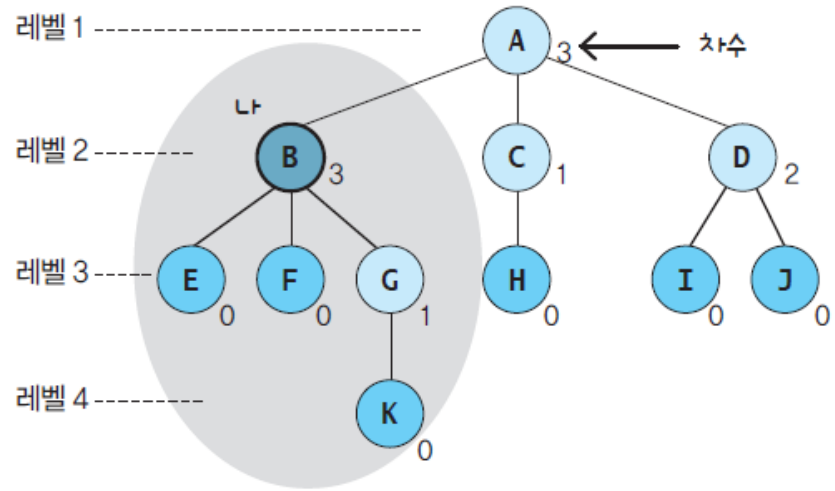
    (예) 회사의 조직도



- 응용
  - 운영체제의 파일 시스템, 탐색 트리, 우선순위 큐, 결정트리 등

# 트리 관련 용어

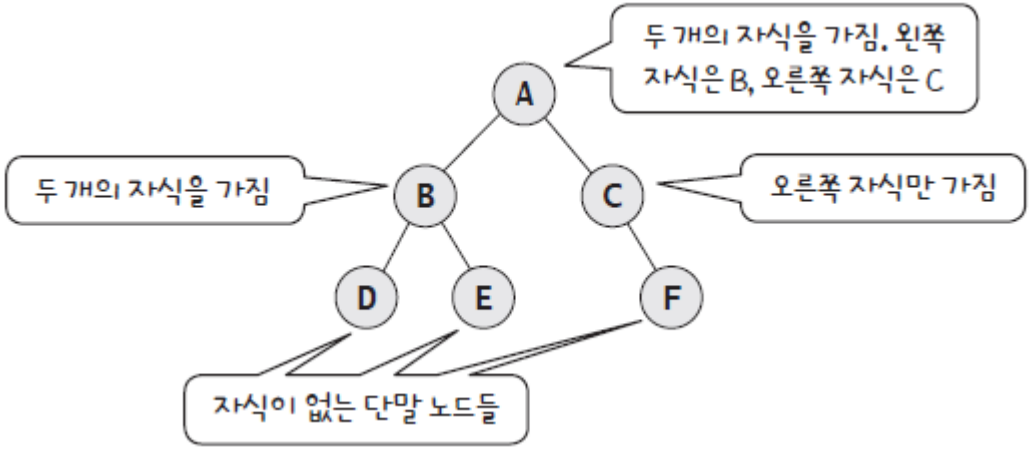



- 루트 노드: A
- B의 부모노드: A
- B의 자식 노드: E, F, G
- B의 자손 노드: E, F, G, K
- K의 조상 노드: G, B, A
- B의 형제 노드: C, D
- B의 차수: 3
- 단말 노드: E, F, K, H, I, J
- 비단말 노드: A, B, C, D, G
- 트리의 높이: 4
- 트리의 차수: 3

| 용어 | 설명 |
|---|---|
| 노드(Node) | 트리의 기본 단위. 데이터를 저장하며, 부모와 자식 관계로 연결됨 |
| 루트(Root) | 트리의 최상단에 있는 노드. 부모 노드가 없음 |
| 리프(Leaf) | 자식이 없는 노드. 말단 노드 |
| 간선(Edge) | 노드 간을 연결하는 선. 트리에서는 한 노드를 다른 노드에 연결함 |
| 부모(Parent) | 다른 노드(자식)를 직접 연결하고 있는 노드 |
| 자식(Child) | 다른 노드(부모)에 의해 연결된 노드 |
| 형제(Sibling) | 같은 부모를 공유하는 노드들 |
| 조상(Ancestor) | 어떤 노드에서 루트까지의 경로에 있는 모든 노드 |
| 자손(Descendant) | 어떤 노드로부터 하위로 연결된 모든 노드 |
| 서브트리(Subtree) | 트리 내에서 어떤 노드를 루트로 하는 하위 트리 |
| 높이(Height) | 루트에서 가장 깊은 리프 노드까지의 거리 (간선 수 기준) |
| 깊이(Depth) | 루트로부터 현재 노드까지의 거리 (간선 수 기준) |
| 차수(Degree) | 한 노드가 가진 자식 노드의 수 |
| 트리의 차수 | 트리 내 모든 노드 중에서 가장 큰 차수를 의미 |

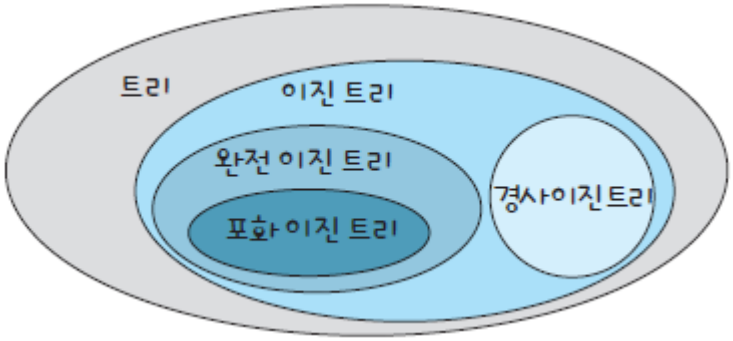
# 4.2 이진 트리(Binary tree)

- 모든 노드가 최대 2개의 자식만을 가질 수 있는 트리
  - 모든 노드의 차수가 2 이하로 제한
  - 왼쪽 자식과 오른쪽 자식은 반드시 구별되어야 함(순서가 존재!)

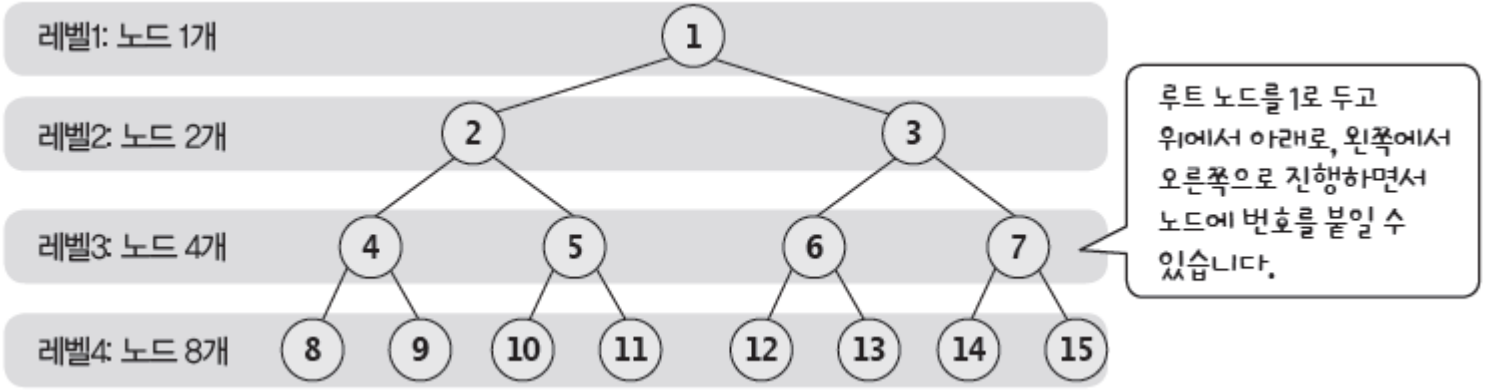



- 이진 트리의 예
  - **빠른 자료의 탐색**이 가능한 **이진 탐색 트리(binary search tree)**
  - **우선순위 큐**를 효과적으로 구현하는 **힙 트리(heap tree)**
  - **수식**을 트리 형태로 표현하여 계산하는 **수식 트리** 등

# 이진 트리의 종류



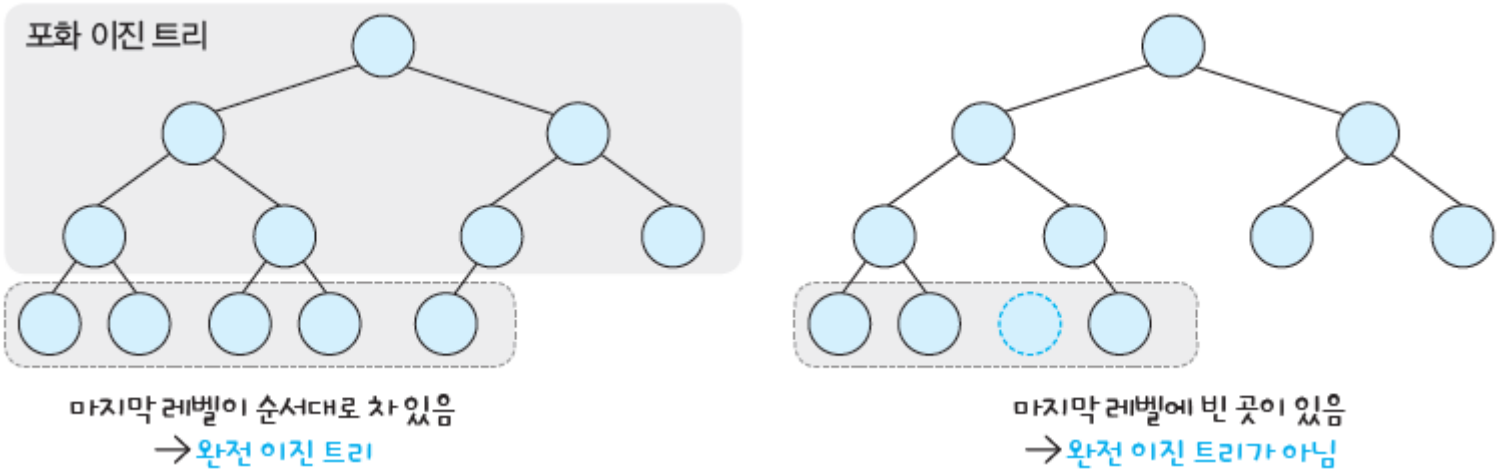

- 포화 이진 트리(full binary tree)




$$\text{전체 노드 개수} : 2^{1-1} + 2^{2-1} + 2^{3-1} + \cdots + 2^{k-1} = \sum_{i=0}^{k-1} 2^i = 2^k - 1$$

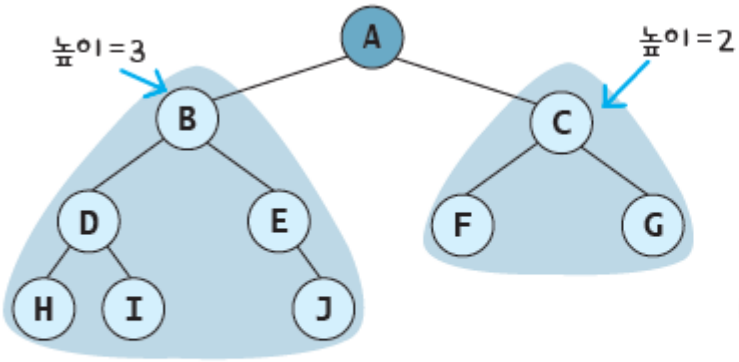
# 이진 트리의 종류

- 완전 이진 트리(complete binary tree)
  - 레벨 k-1까지는 포화이진트리,
  - 마지막 레벨 k에서는 왼쪽부터 오른쪽으로 노드가 순서대로
  - 예: 힙(heap)

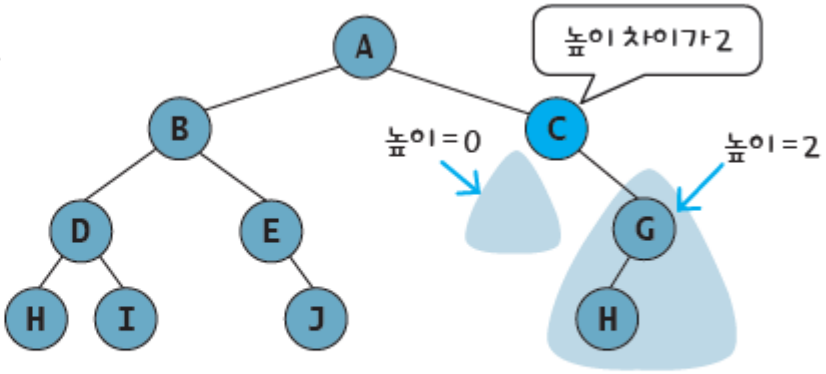


# 이진 트리의 종류

- 균형 이진 트리(balanced binary tree)
  - 모든 노드에서 좌우 서브 트리의 **높이 차이가 1 이하인** 트리
  - 아니면 → **경사 이진 트리**



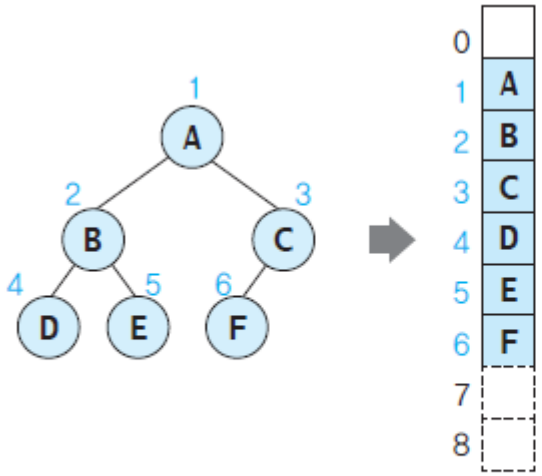

(a) 모든 노드의 좌우 서브 트리 높이 차이가 1 이하임 → 균형 이진 트리



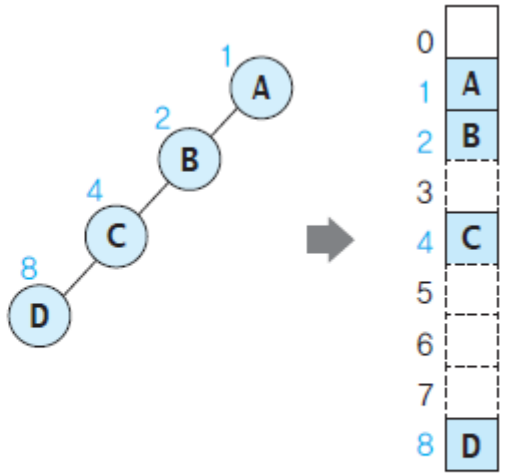

(b) 노드 C의 좌우 서브 트리 높이 차이가 2임(1 초과) → 균형 이진 트리가 아님

# 이진 트리의 배열 구조 표현

- 배열 구조
  - **트리의 높이가 k이면 배열의 길이가 $2^k$ - 1**
  - 이진 트리를 **포화 이진 트리의 일부**라고 생각하고 **번호 부여**




(a) 완전 이진 트리의 배열 표현. 중간에 빈 칸이 발생하지 않음.



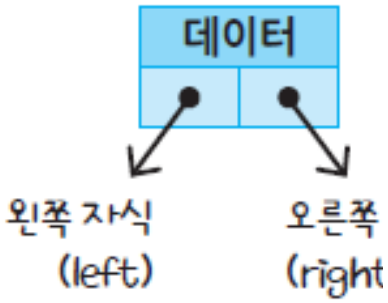

(b) 경사 이진 트리의 배열 표현. 중간에 빈 칸이 많이 발생할 수 있음.

- 노드 i의 부모 노드 인덱스 = i/2
- 노드 i의 왼쪽 자식 노드 인덱스 = 2i
- 노드 i의 오른쪽 자식 노드 인덱스 = 2i+1

파이썬에서는 나눗셈 연산자가 /와 //로 구분되어 있습니다. 정수 나눗셈을 위해서는 i//2를 써야합니다.

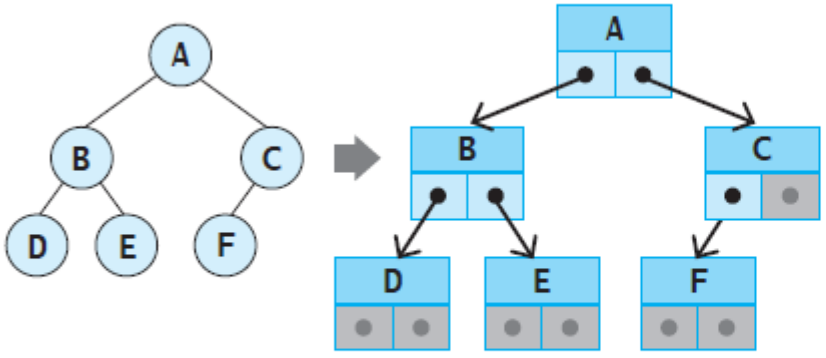
# 이진 트리의 링크 표현법

- 연결된 구조 표현: 링크 표현법
  - 연결된 구조의 이진 트리를 위한 노드의 구조

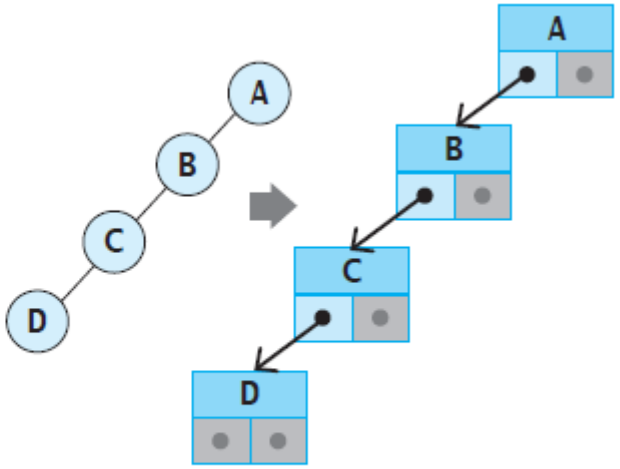



이진 트리를 위한 노드의 생성자

```
class BTNode:
    def __init__ (self, elem, left=None, right=None):
        self.data = elem
        self.left = left          # 왼쪽 자식을 위한 링크
        self.right = right        # 오른쪽 자식을 위한 링크
```



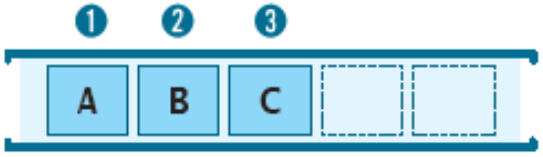

(a) 완전 이진 트리의 링크 표현



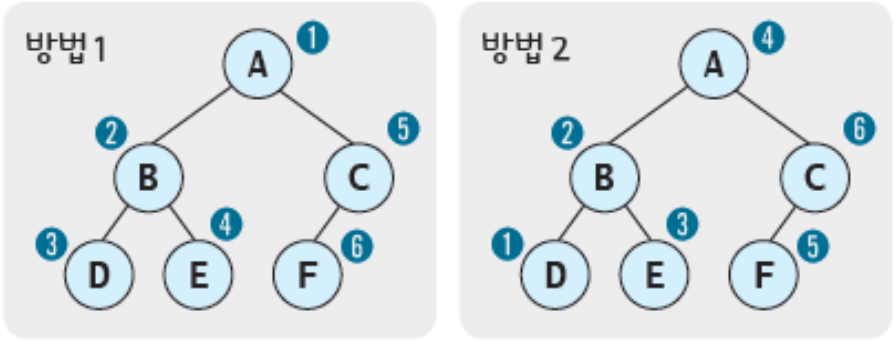

(b) 경사 이진 트리의 링크 표현

# 4.3 이진 트리의 연산

- 트리의 순회(traversal)
  - 트리의 모든 노드를 한 번씩 방문
  - 순환 기법(재귀 호출) 사용
  - 예: 트리의 모든 노드를 한 번씩 화면에 출력

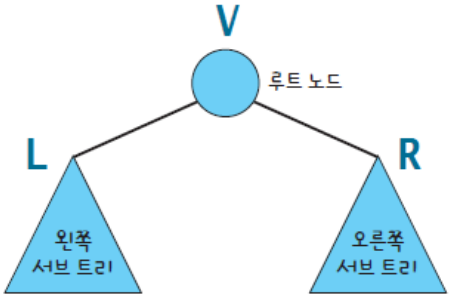



선형자료구조는
순회 방법이 단순합니다.



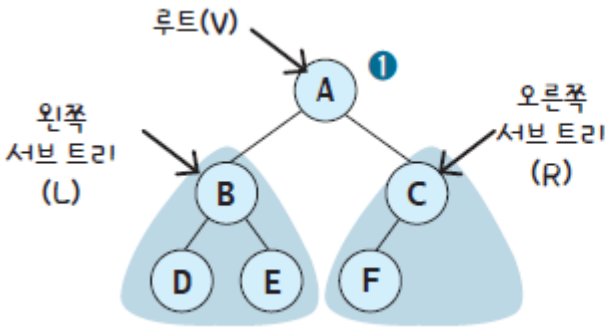

트리는 다양한 방법으로 순회할 수 있습니다.

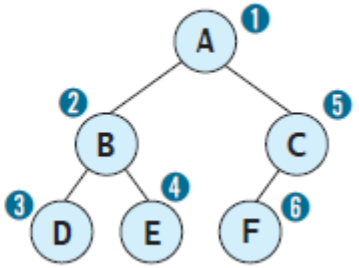
- 이진 트리의 표준 순회
  - 전위 순회(preorder traversal) : VLR
  - 중위 순회(inorder traversal) : LVR
  - 후위 순회(postorder traversal) : LRV



# 전위 순회(preorder)

- V → L → R (서브 트리도 같은 순회 방법을 적용)



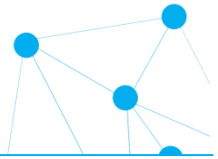




```
def preorder(n) :                    # 전위순회 함수
    if n is not None :
        print(n.data, end=' ')       # 노드를 방문해 처리할 연산들의 위치.
        preorder(n.left)             # 여기서는 노드의 데이터를 단순히 화면에 출력.
        preorder(n.right)            # 왼쪽 서브 트리 처리
                                     # 오른쪽 서브 트리 처리
```

전위순회 결과 --> A B D E C F

중첩된 괄호표현 --> ( A ( B ( D ) ( E ) ) ( C ( F ) ) )

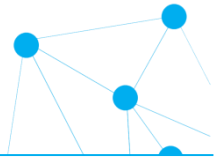
# 중위 순회(inorder)



- L → V → R


```
def inorder(n) :                    # 전위순회 함수
    if n is not None :
        inorder(n.left)             # 왼쪽 서브 트리 처리
        print(n.data, end=' ')      ← 노드에서 처리할 연산들의 위치
        inorder(n.right)            # 오른쪽 서브 트리 처리
```

# 후위 순회(postorder)

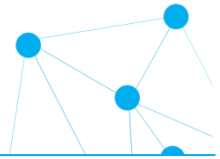


- L → R → V


```
def postorder(n) :
    if n is not None :
        postorder(n.left)
        postorder(n.right)
        print(n.data, end=' ')
```

← 노드에서 처리할 연산들의 위치

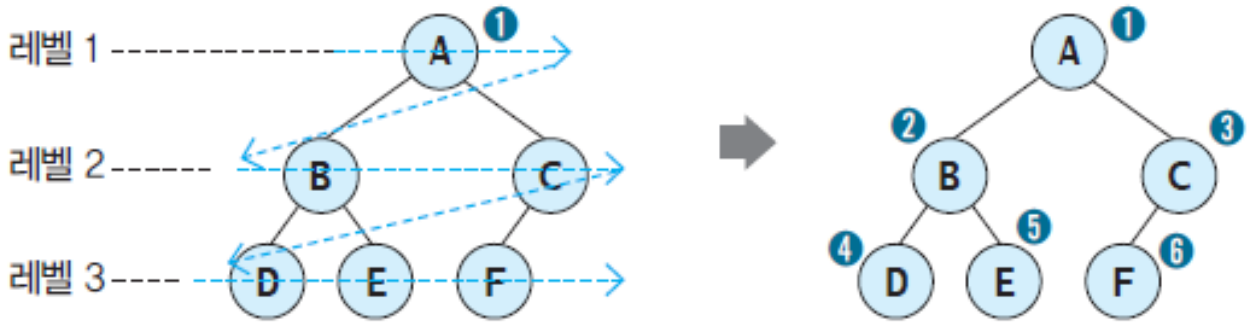
# 순회 방법의 선택

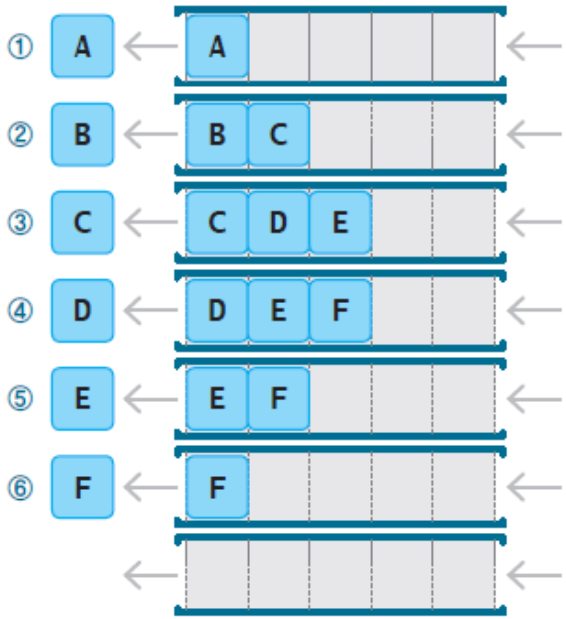


- 순회 방법은 어떻게 선택할까요?
  - 순서는 중요하지 않고 노드를 전부 방문하기만 하면 된다면? 어떤 방법도 상관이 없음
    (예) 노드의 수, 모든 노드의 순서 없는 출력 등
  - 자식을 먼저 처리해야 부모를 처리할 수 있다면? 후위 순회
    (예) 컴퓨터 폴더의 용량 계산
  - 부모가 처리되어야 자식을 처리할 수 있다면? 전위 순회
    (예) 노드의 레벨 계산

# 레벨 순회(level order)

- 레벨 순으로 노드를 방문 : 큐로 구현



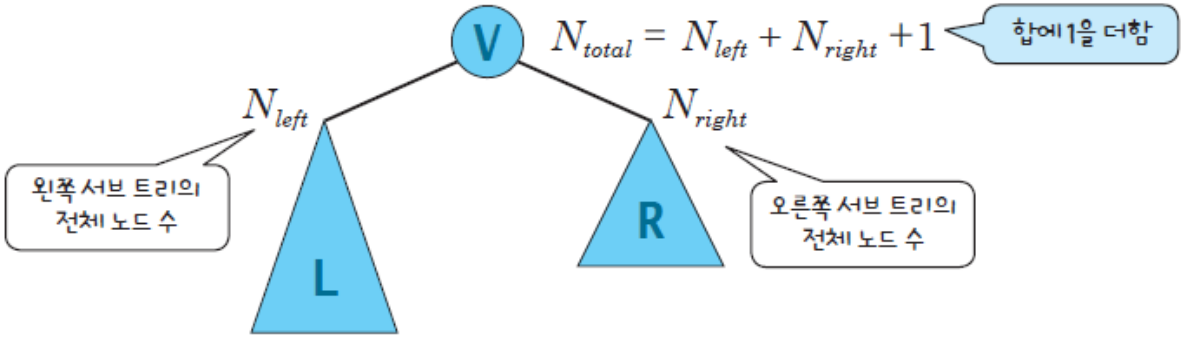




```python
from collections import deque
def levelorder(root):
    if root is None:
        return  # 빈 트리 처리
    q = deque()
    q.append(root)
    while q: # 큐가 공백이 될 때까지 반복
        node = q.popleft()
        print_data(node.data)  # 큐에서 노드가 나오는 순서가 방문 순서
        if node.left:
            q.append(node.left)
        if node.right:
            q.append(node.right)
```

# 이진 트리의 연산들

- 전체 노드의 수 구하기

> 이진 트리의 노드 개수는 왼쪽 서브 트리의 노드 수와 오른쪽 서브 트리의 노드 수의 합에 1(루트 노드)을 더하면 됩니다.

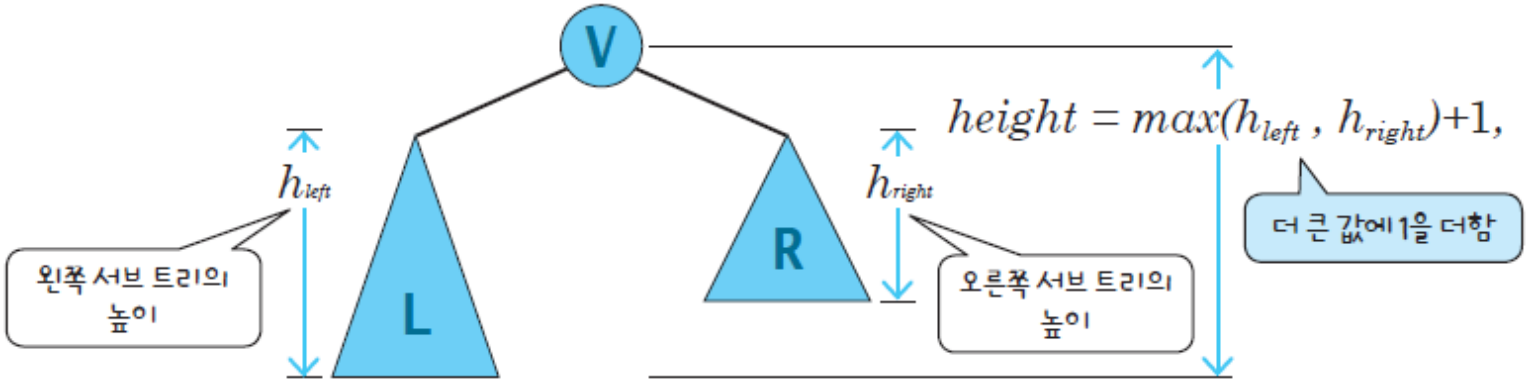



```
def count_node(n) :
    if n is None :  # n이 None이면 공백 트리 --> 0을 반환
        return 0
    else :          # 좌우 서브 트리의 노드 수의 합 + 1 (순환이용)
        return count_node(n.left) + count_node(n.right) + 1
```

그림 4.22

# 트리의 높이 구하기

이진 트리의 높이는 왼쪽 서브 트리의 높이와 오른쪽 서브 트리의 높이 중에서 큰 값에 1을 더한 값입니다.

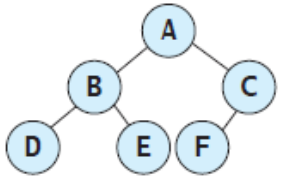



```
def calc_height(n) :
    if n is None :          # 공백 트리 --> 0을 반환
        return 0
    hLeft = calc_height(n.left)   # hLeft <- L의 높이
    hRight = calc_height(n.right)# hRight<- R의 높이
    if (hLeft > hRight) :  # 더 큰 값에 1을 더해 반환
        return hLeft + 1
    else: return hRight + 1
```

# 테스트 프로그램

```
d = BTNode('D', None, None)
e = BTNode('E', None, None)
b = BTNode('B', d, e)
f = BTNode('F', None, None)
c = BTNode('C', f, None)
root = BTNode('A', b, c)
```

단말 노드부터 루트까지 하나씩 노드를 만들어 트리를 구축



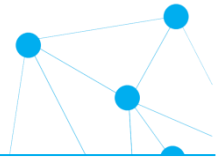

```
print('\n In-Order : ', end=''); inorder(root)
print('\n Pre-Order : ', end=''); preorder(root)
print('\n Post-Order : ', end=''); postorder(root)
print(\n Level-Order : ', end=''); levelorder(root)
print()

print(" 노드의 개수 = %d개" % count_node(root))
print(" 트리의 높이 = %d" % calc_height(root))
```

**실행 결과**

```
 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
 트리의 높이 = 3
```

중첩된 괄호 표현을 위해 괄호 ( , ) 추가

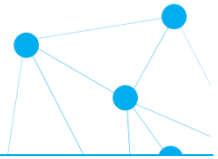
# 4.4 모스 코드 결정 트리



- 새뮤얼 모스(Samuel Morse) 코드
  - 도트(점)와 대시(선)의 조합으로 구성된 메시지 전달용 부호
  - SOS → ••• ——— •••

| 문자 | 부호 | 문자 | 부호 | 문자 | 부호 |
|---|---|---|---|---|---|
| A | • – | J | • – – – | S | • • • |
| B | – • • • | K | – • – | T | – |
| C | – • – • | L | • – • • | U | • • – |
| D | – • • | M | – – | V | • • • – |
| E | • | N | – • | W | • – – |
| F | • • – • | O | – – – | X | – • • – |
| G | – – • | P | • – – • | Y | – • – – |
| H | • • • • | Q | – – • – | Z | – – • • |
| I | • • | R | • – • | | |

# 문자를 모스 코드로 변환하는 과정



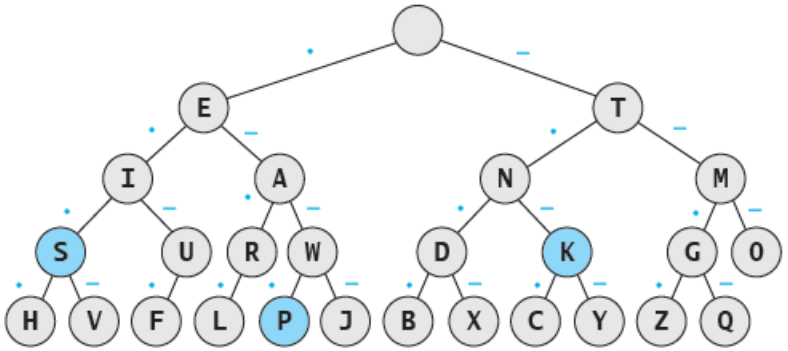
- 인코딩 함수 : 문자에 대응되는 코드를 표에서 찾아 순서대로 출력
- 디코딩 함수 : 모스 코드가 주어졌을 때 해당하는 알파벳을 추출
  - 표의 모든 항목을 하나씩 검사해야 함 → 비효율적
  - **결정 트리 사용** : 여러 단계의 복잡한 조건을 갖는 문제에 대해 조건과 그에 따른 해결방법을 트리 형태로 나타낸 것

```
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', '--..') ]

def encode(ch):
    idx = ord(ch)-ord('A')  # 리스트에서 해당 문자의 인덱스
    return table[idx][1]    # 해당 문자의 모스 부호 반환

def decode_simple(morse):
    for tp in table :               # 모스 코드 표의 모든 문자에 대해
        if morse == tp[1] :         # 찾는 코드와 같으면
            return tp[0]            # 그 코드의 문자를 반환
```

# 모스 코딩 디코딩을 위한 결정 트리(decision tree)

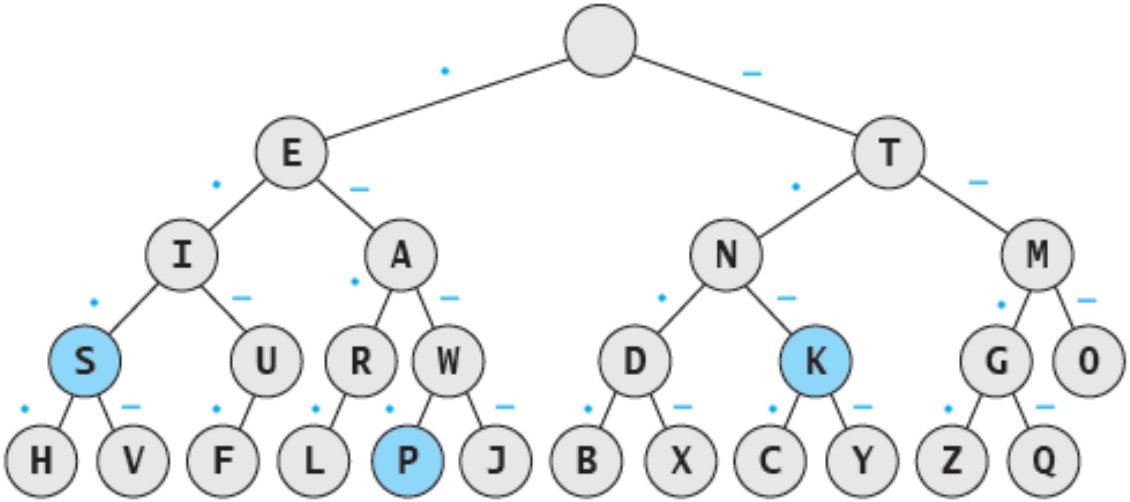
- **모스 코드를 위한 결정 트리 만들기 : 최대 트리의 높이 만큼만 비교 → 효율적**
  - 빈 루트 노드를 만들고 모스 코드표의 각 문자를 하나씩 트리에 추가
  - 문자를 추가할 때 루트부터 시작하여 트리를 타고 내려감.
    - 만약 타고 내려갈 자식 노드가 None이면 새로운 노드를 추가하는데, 노드만 추가할 뿐이지 그 노드의 문자는 아직 결정할 수 없음
  - 마지막 코드의 노드에 도달하면 그 노드에 문자를 할당

```
def make_morse_tree() -> BTNode:
    root = BTNode(None, None, None)
    for ch, code in MORSE_TABLE:
        cur = root
        for c in code:
            if c == '.':
                if cur.left is None:
                    cur.left = BTNode(None,None, None)
                cur = cur.left
            else:  # '-'
                if cur.right is None:
                    cur.right = BTNode(None, None, None)
                cur = cur.right
        cur.data = ch
    return root
```



# 결정 트리(decision tree) 기반 모스 코드의 디코딩

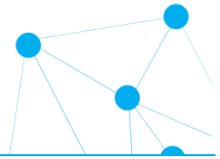
```
def decode(root, code):
    node = root                                  # 루트 노드에서 시작
    for c in code :                              # 각 부호에 대해
        if c == '·' : node = node.left           # 점( · ): 왼쪽으로 이동
        elif c=='-' : node = node.right          # 선(-): 오른쪽으로 이동
    return node.data                             # 문자 반환
```




예:

· · · : S
– · – : K
· – – · : P
– – · · · : 코드 없음

# 테스트 프로그램



```
morseCodeTree = make_morse_tree()
str = input("입력 문장 : ")
mlist = []
for ch in str:
    code = encode(ch)
    mlist.append(code)
print("Morse Code: ", mlist)
print("Decoding : ", end='')
for code in mlist:
    ch = decode(morseCodeTree, code)
    print(ch, end='')
print()
```

← 모스코드 결정트리를 만듦. morseCodeTree가 루트 노드

← 입력 문자열의 각 문자를 순서 대로 모스코드로 변환하여 리스트에 추가

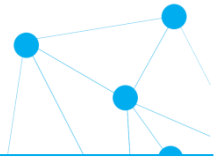
← 리스트의 모스 코드를 순서 대로 디코딩한 문자를 화면에 출력

**실행 결과**

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

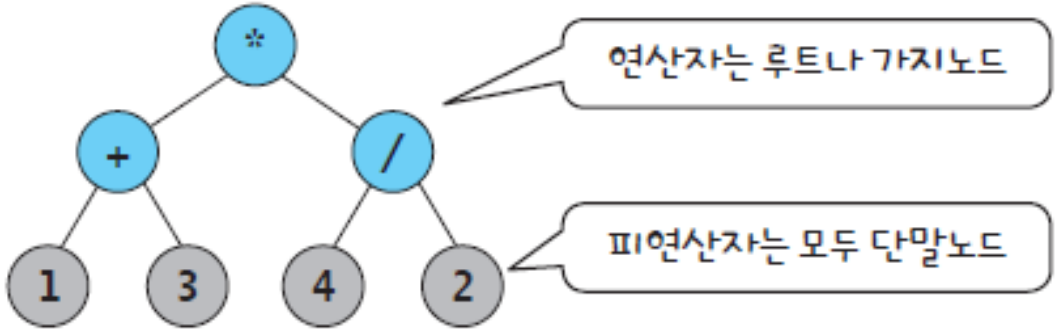
A M E O V E R

# 4.5 수식 트리



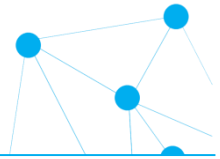
- 수식 트리(Expression Tree)
  - 산술식을 트리 형태로 표현한 이진 트리
  - 피연산자는 단말노드에 저장
  - 연산자는 루트나 중간 노드에 저장

```
(1 + 3) * (4 / 2)
```

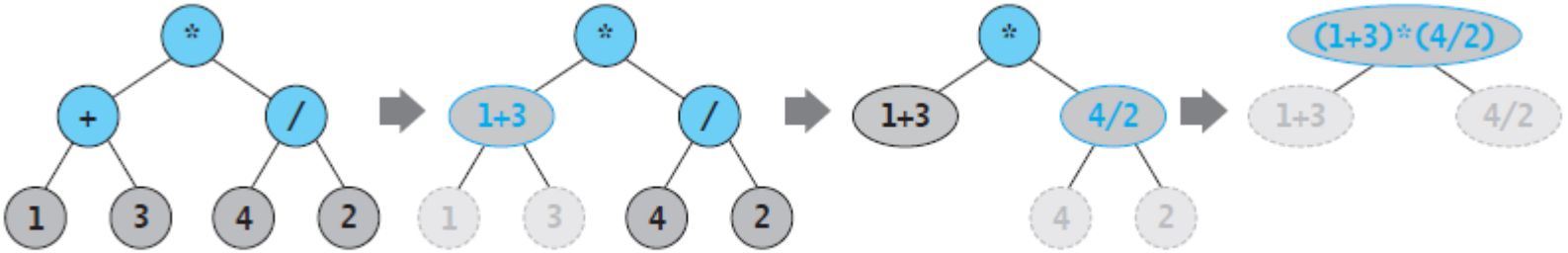



- 수식 트리의 계산
- 수식 트리 만들기

# 수식 트리의 계산



- 후위 순회




수식 트리 (a)왼쪽 서브 트리 계산 (b)오른쪽 서브 트리 계산 (c)루트 노드 계산

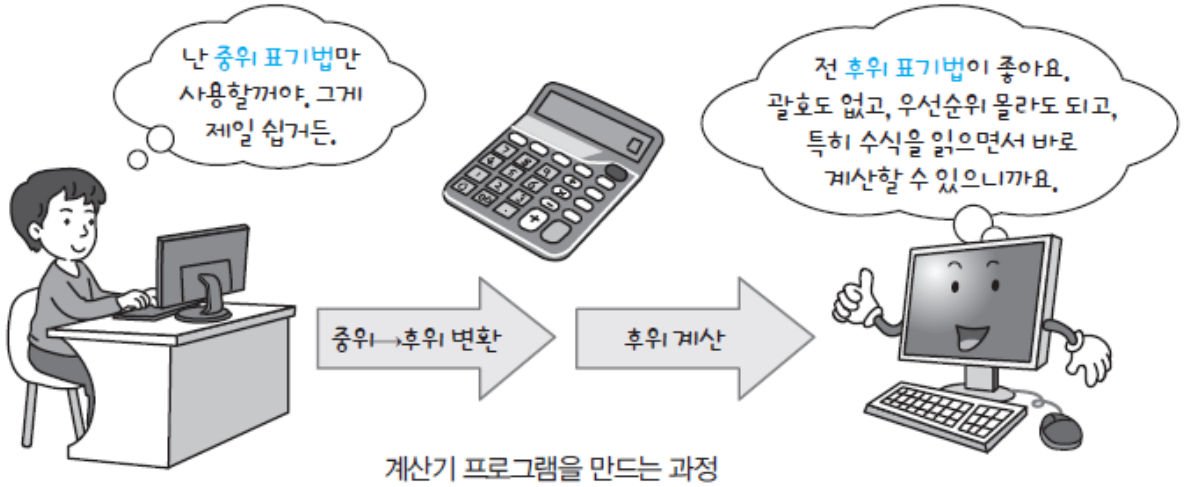
```
def evaluate(node) :
    if node is None :              # 공백 트리이면 0 반환
        return 0
    elif node.isLeaf() :           # 단말 노드이면 -> 피연산자
        return node.data           # 그 노드의 값(데이터) 반환
    else :                         # 루트나 가지노드라면 -> 연산자
        op1 = evaluate(node.left)
        op2 = evaluate(node.right)
        if node.data == '+' : return op1 + op2
        elif node.data == '-' : return op1 - op2
        elif node.data == '*' : return op1 * op2
        elif node.data == '/' : return op1 / op2
```

← 왼쪽과 오른쪽 서브트리를 먼저 계산해야 루트를 계산할 수 있음.

← 루트(현재 노드)를 처리. 후위순회

# 수식의 표현 방법

| 전위(prefix) | 중위(infix) | 후위(postfix) |
|---|---|---|
| **연산자** 피연산자1 피연산자2 | 피연산자1 **연산자** 피연산자2 | 피연산자1 피연산자2 **연산자** |
| + A B | A + B | A B + |
| + 5 * A B | 5 + A * B | 5 A B * + |




계산기 프로그램을 만드는 과정

- 후위 표기의 장점
  - 괄호를 사용하지 않음
  - 수식을 읽으면서 바로 계산
  - 연산자의 우선순위를 생각할 필요가 없음.

# 후위 표기 식으로 수식 트리 만들기

- 맨 뒤에서 앞으로 읽으면서 처리
- 입력 수식 : **1 3 + 4 2 / ***

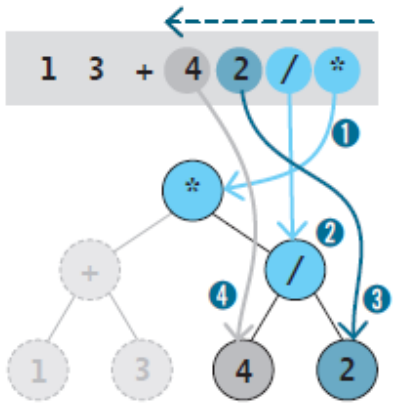
```
def buildETree(expr):
    if len(expr) == 0 :
        return None

    token = expr.pop()
    if token in "+-*/" :
        node = BTNode(token)
        node.right= buildETree(expr)
        node.left = buildETree(expr)
        return node
    else :
        return BTNode(float(token))
```

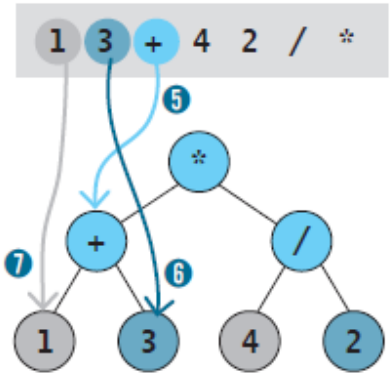
후위표기 수식을 expr로 전달. 예를 들어, 그림 4.25의 수식 트리는 ['1','3','+','4','2','/','*']와 같이 전달됨.

후위순회는 수식을 뒤에서 앞으로 처리. 따라서 pop()으로 맨 뒤의 요소를 꺼냄.

연산자이면 노드를 만들고, 오른쪽과 왼쪽순으로 서브트리를 순환호출을 이용해 만듦. 마지막으로 노드 반환.

피연산자이면 단말노드이므로 노드를 만들어 바로 반환



# 테스트 프로그램

```
str = input("입력(후위표기): ")      # 후위표기식 입력
expr = str.split()                  # 토큰 리스트로 변환
print("토큰분리(expr): ", expr)
root = buildETree(expr)  ← 후위 표기식을 수식트리로 만들고 루트를 반환
print('\n 전위순회: ', end=''); preorder(root)
print('\n 중위순회: ', end=''); inorder(root)
print('\n후위순회: ', end=''); postorder(root)
print('\n계산 결과 : ', evaluate(root))   # 수식 트리 계산
```

**실행 결과**

토큰 분리 결과(공백으로 분리)

```
입력(후위표기): 1 3 + 4 2 / *
토큰분리(expr):  ['1', '3', '+', '4', '2', '/', '*']

전위 순회: ( * ( + ( 1.0 ) ( 3.0 ) ) ( / ( 4.0 ) ( 2.0 ) ) )
중위 순회: 1.0 + 3.0 * 4.0 / 2.0
후위 순회: 1.0 3.0 + 4.0 2.0 / *
계산 결과 :  8.0
```

계산 결과