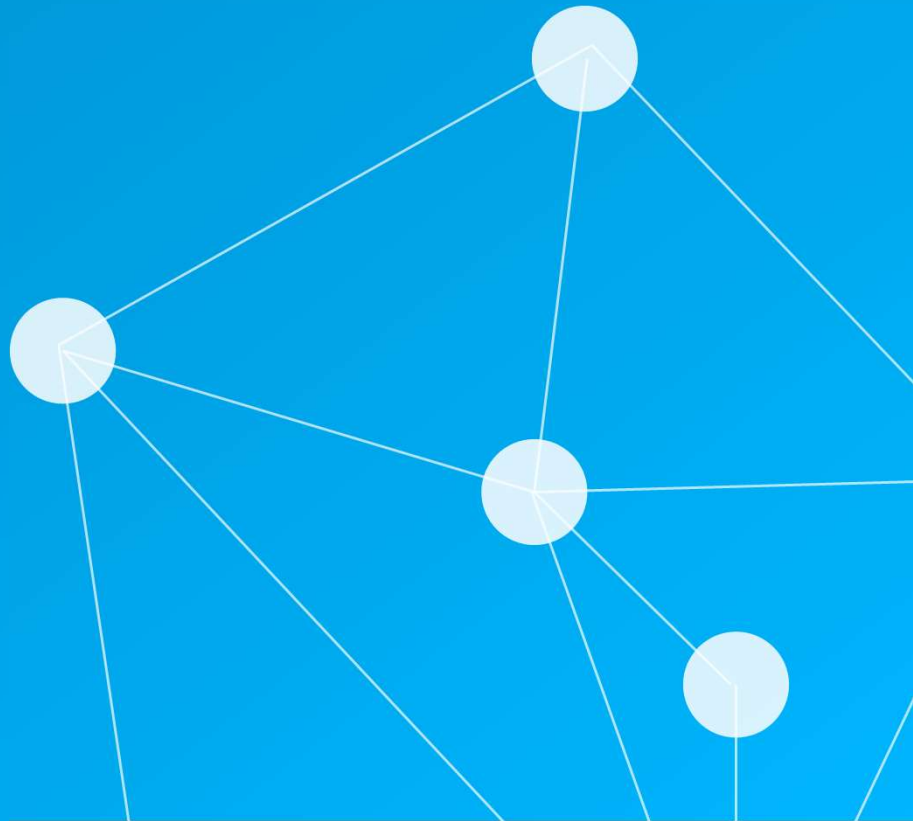
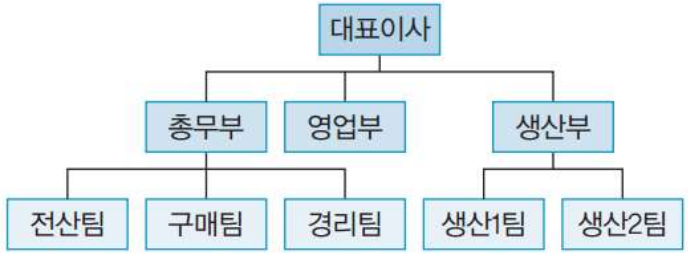
10주차 트리

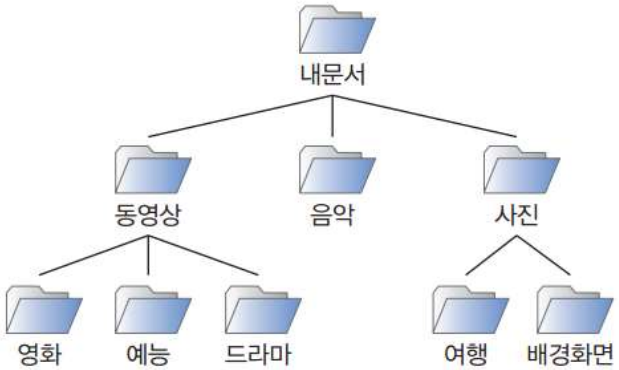


트리란?

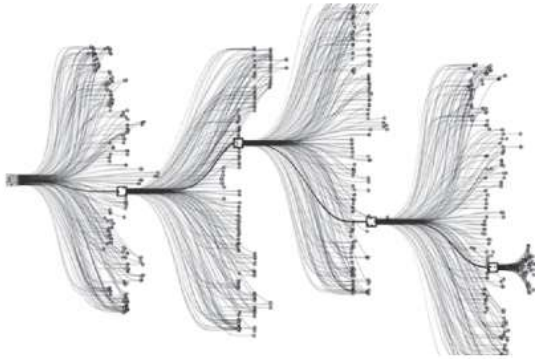
- 트리: 계층적인 자료의 표현에 적합한 자료 구조



(a) 회사의 조직도

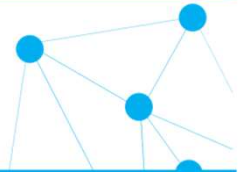


(b) 컴퓨터의 폴더 구조



(c) 인공지능 바둑 프로그램의
거대한 결정 트리(decision tree)

트리

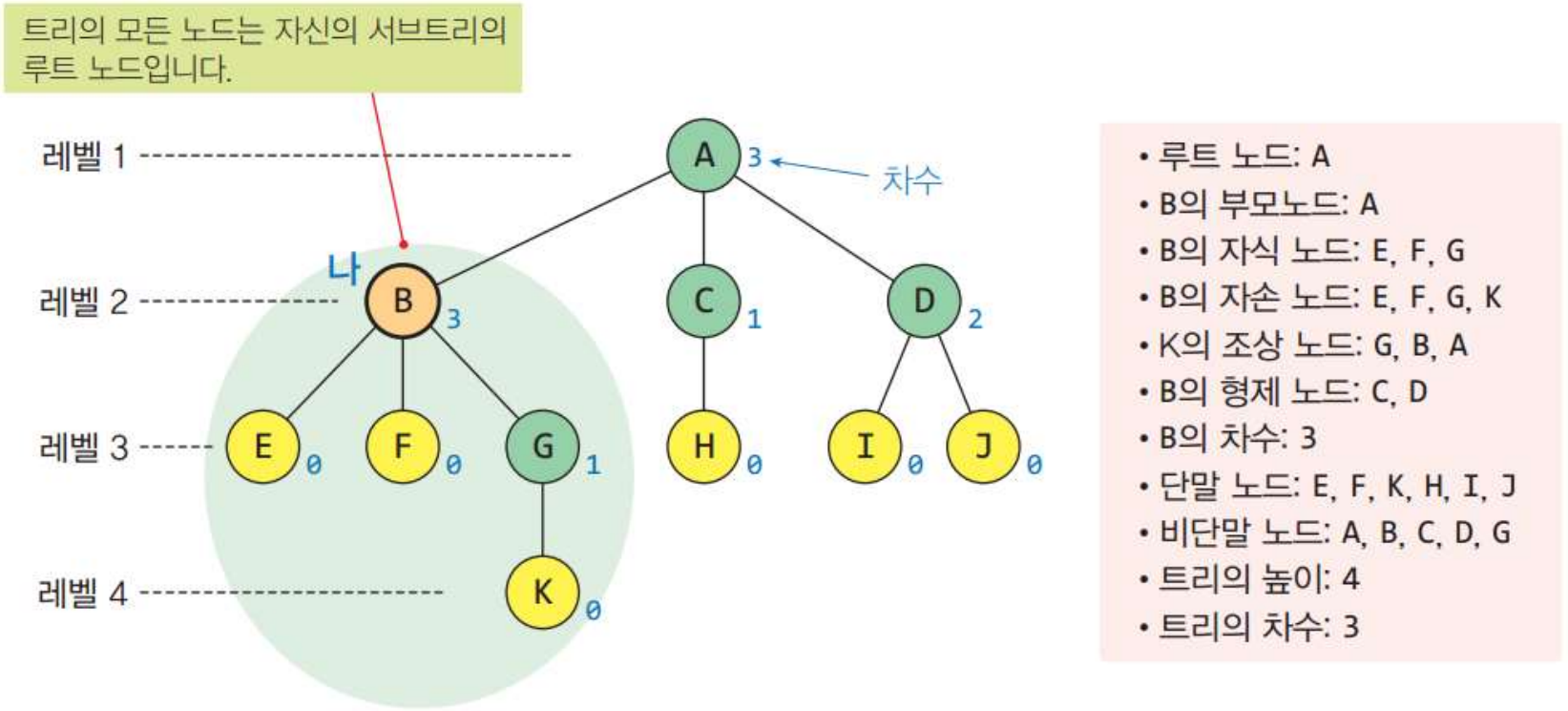


- 파이썬 리스트나 연결리스트: 데이터를 일렬로 저장하기 때문에 탐색 연산이 순차적으로 수행되는 단점
- 배열은 미리 정렬해 놓으면 이진탐색을 통해 효율적인 탐색이 가능하지만, 삽입이나 삭제 후에도 정렬 상태를 유지해야 하므로 삽입이나 삭제하는데 O(N) 시간 소요

Applications

- 조직이나 기관의 계층구조
- 컴퓨터 운영체제의 파일 시스템
- 트리는 일반적인 트리와 이진트리(Binary Tree)로 구분
- 다양한 탐색트리(Search Tree), 힙(Heap) 자료구조, 컴파일러의 수식을 위한 구문트리(Syntax Tree)등 광범위하게 응용

트리의 용어

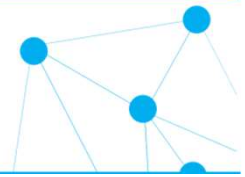


- 루트 노드: A
- B의 부모노드: A
- B의 자식 노드: E, F, G
- B의 자손 노드: E, F, G, K
- K의 조상 노드: G, B, A
- B의 형제 노드: C, D
- B의 차수: 3
- 단말 노드: E, F, K, H, I, J
- 비단말 노드: A, B, C, D, G
- 트리의 높이: 4
- 트리의 차수: 3

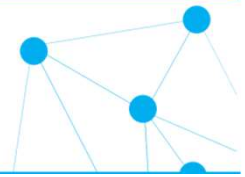
- 루트 노드
- 간선 또는 에지
- 부모 / 자식 / 형제
- 조상 / 자손
- 단말 / 비단말 노드

- 노드의 차수
- 트리의 차수
- 레벨
- 트리의 높이

용어

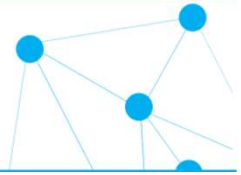


- 루트(Root) – 트리의 최상위에 있는 노드
- 자식(Child) – 노드 하위에 연결된 노드
- 차수(Degree) – 자식노드 수
- 부모(Parent) – 노드의 상위에 연결된 노드
- 이파리(Leaf) – 자식이 없는 노드
- 형제(Sibling) – 동일한 부모를 가지는 노드
- 조상(Ancestor) – 루트까지의 경로상에 있는 모든 노드들의 집합
- 후손(Descendant) –노드 아래로 매달린 모든 노드들의 집합
- 서브트리(Subtree) – 노드 자신과 후손노드로 구성된 트리



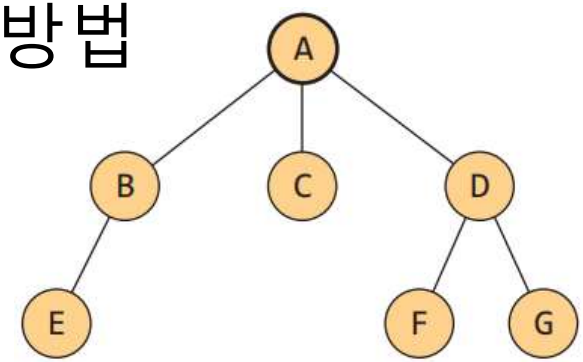
- 레벨(Level) – 루트는 레벨 1, 아래 층으로 내려가며 레벨이 1씩 증가
  - 레벨은 깊이(Depth)와 동일
- 높이(Height) – 트리의 최대 레벨
- 키(Key) – 탐색에 사용되는 노드에 저장된 정보

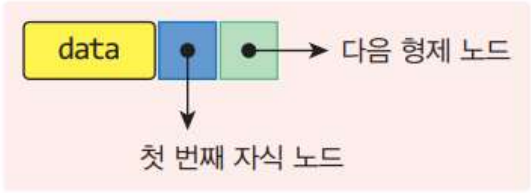
일반 트리의 표현 방법

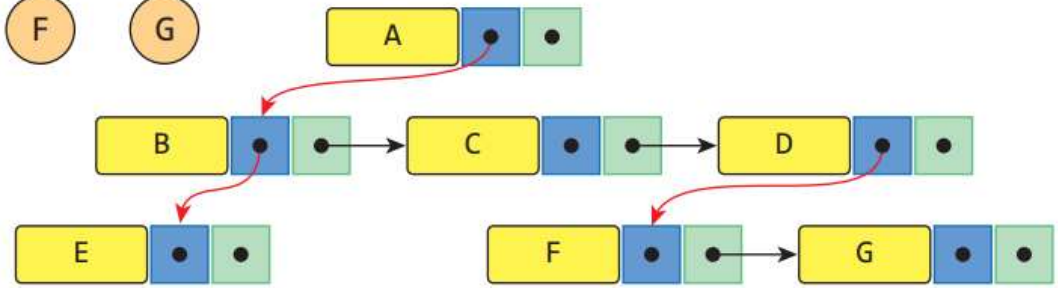


- 일반 트리의 노드

- 다른 방법

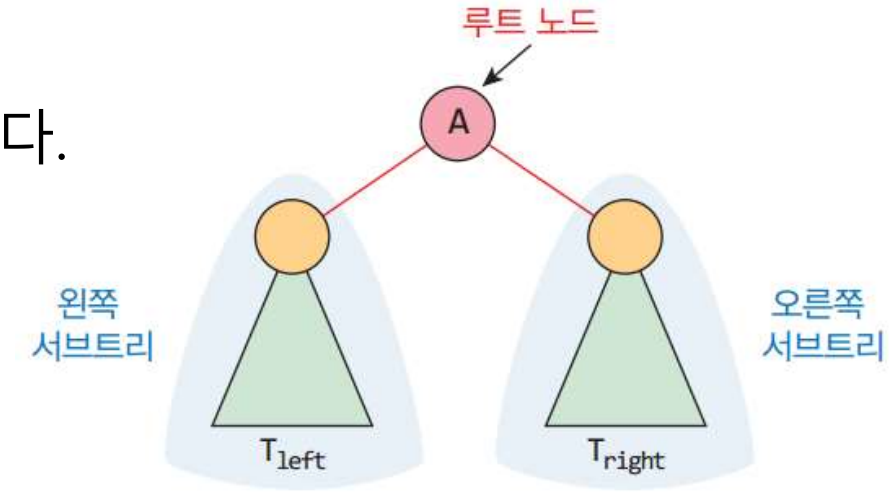






이진 트리

- 모든 노드가 2개의 서브 트리를 갖는 트리
  - 서브트리는 공집합일수 있다.
  - 이진트리는 순환적으로 정의된다.



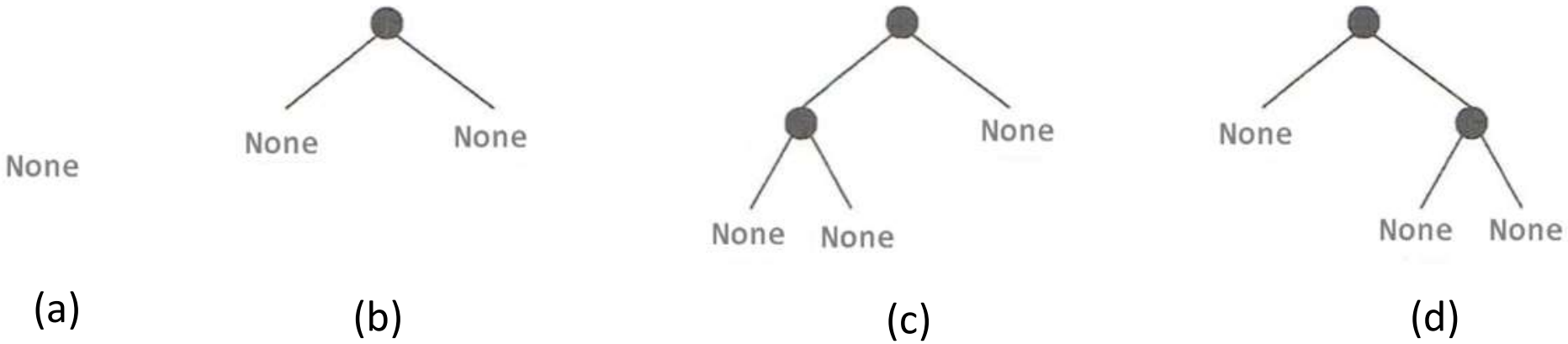
[정의] 이진트리는empty이거나, empty가 아니면, 루트노드와 2개의 이진트리인 왼쪽 서브트리와 오른쪽 서브트리로 구성된다.

(a) empty 트리

(b) 루트만 있는 이진트리

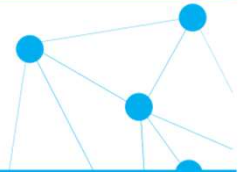
(c) 루트의 오른쪽 서브트리가 없는(empty) 이진트리

(d) 루트의 왼쪽 서브트리가 없는 이진트리

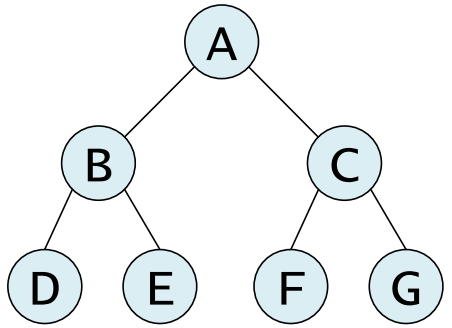


(a) (b) (c) (d)

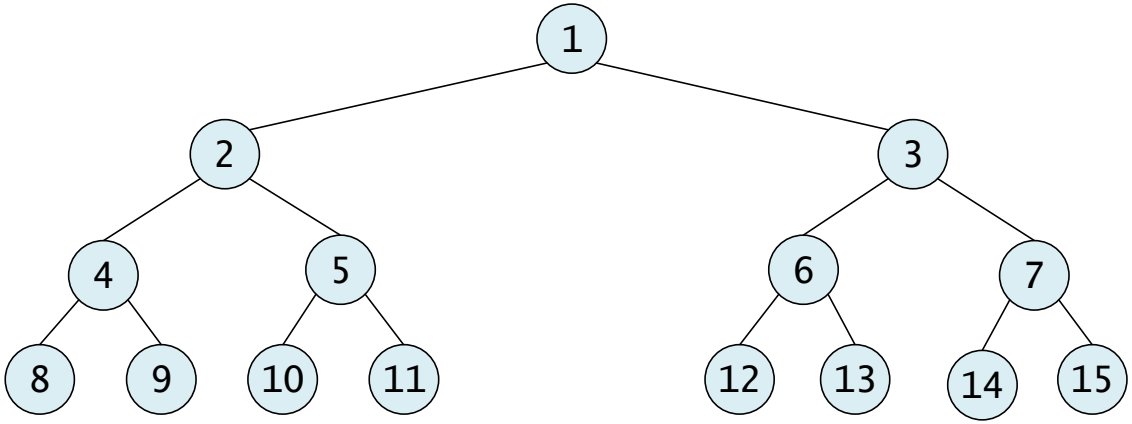
이진 트리의 분류



- 포화 이진 트리(full binary tree)
  - 트리의 각 레벨에 노드가 꽉 차있는 이진트리

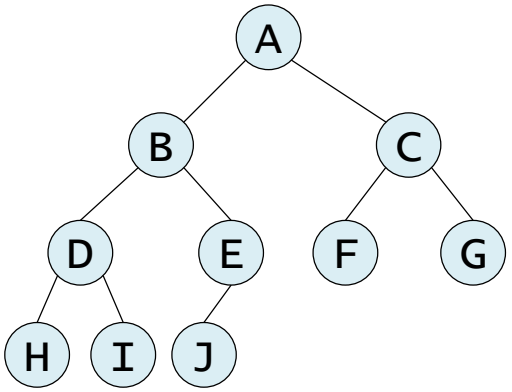


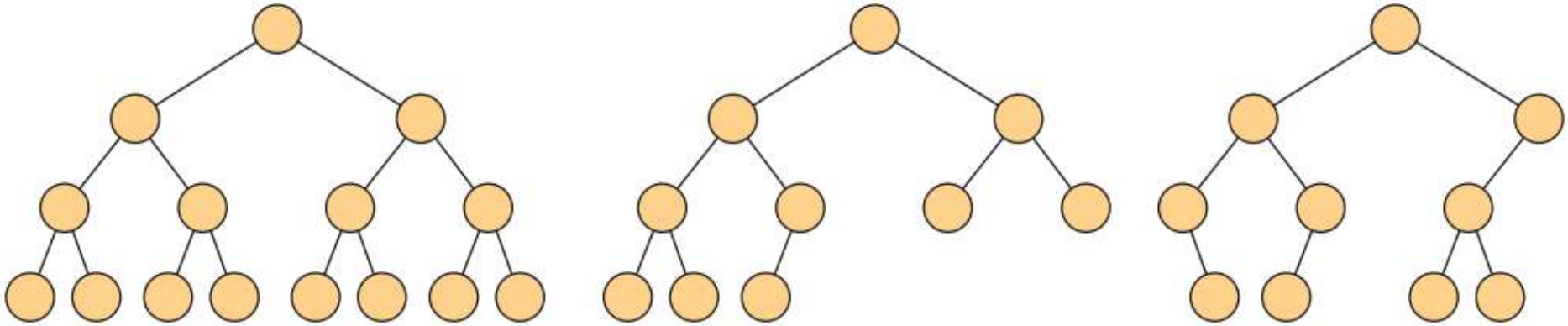
  - 노드의 번호



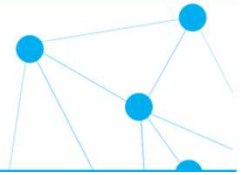
이진 트리의 분류

- 완전 이진 트리(complete binary tree)
  - 높이가 h일 때 레벨 1부터 h-1까지는 노드가 모두 채워짐
  - 마지막 레벨 h에서는 노드가 순서대로 채워짐

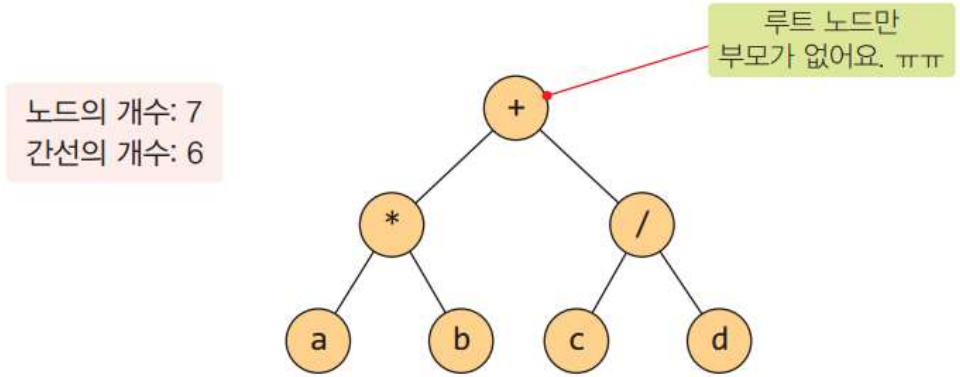




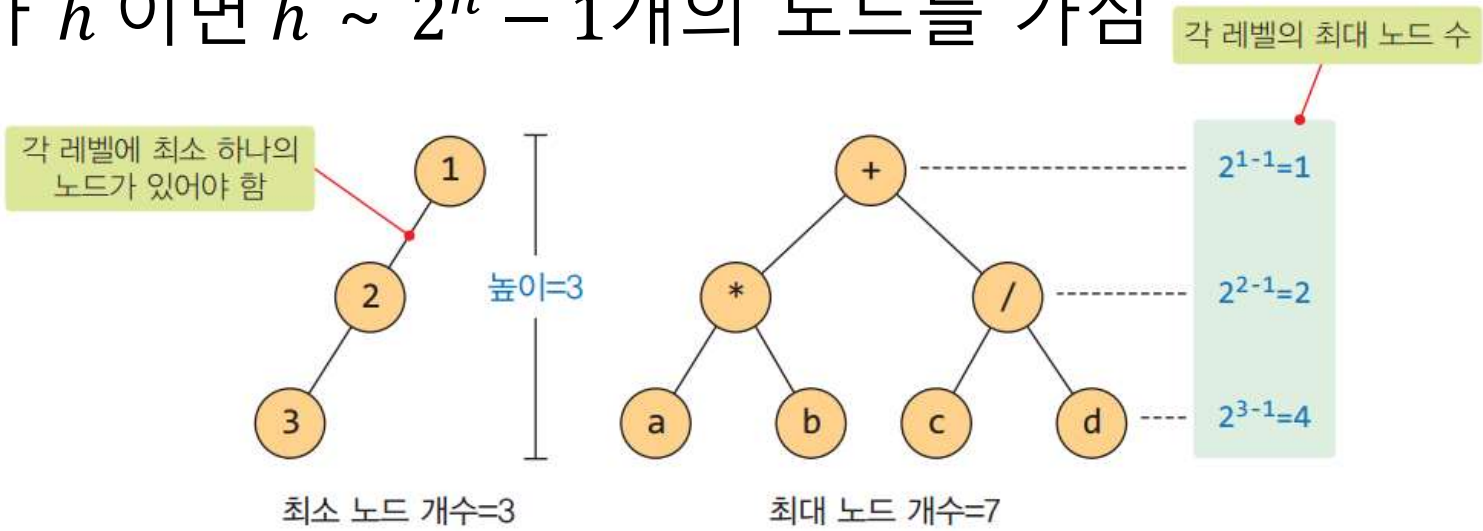
이진 트리의 성질



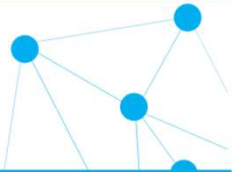
- 노드의 개수가 n개이면 간선의 개수는 n-1



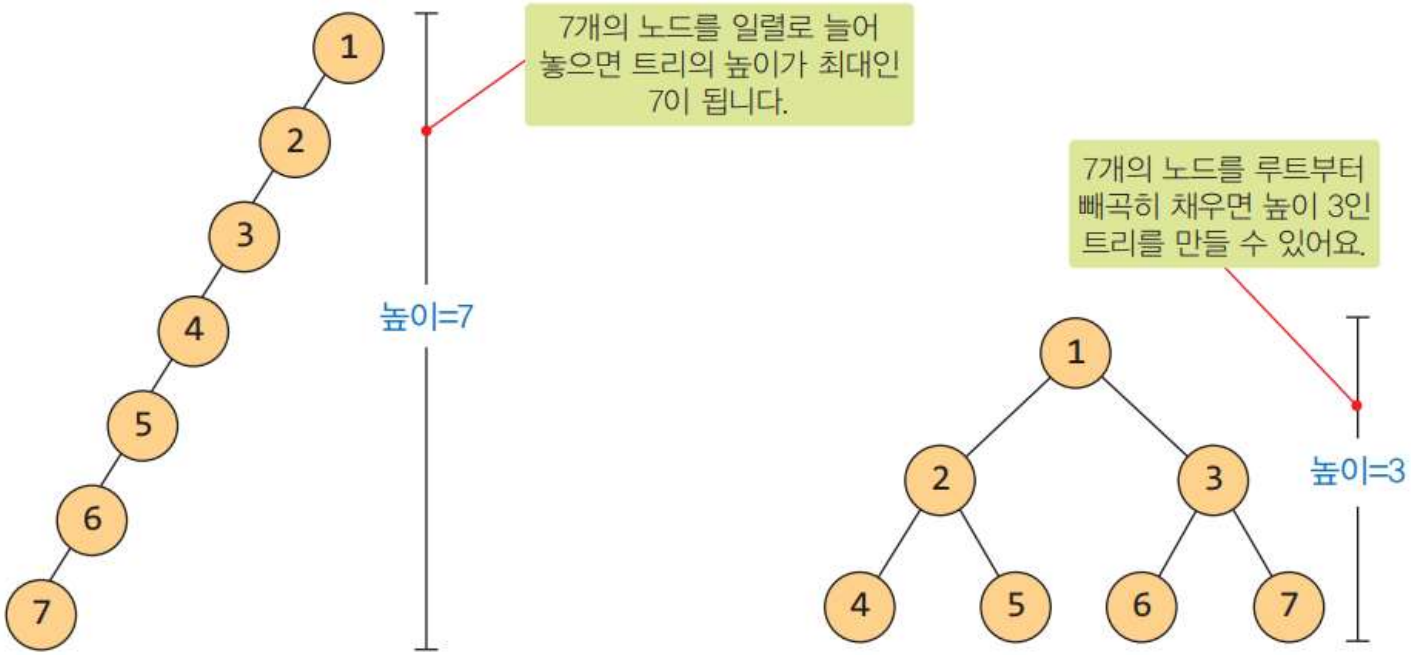
- 높이가 h 이면 $h \sim 2^h - 1$개의 노드를 가짐



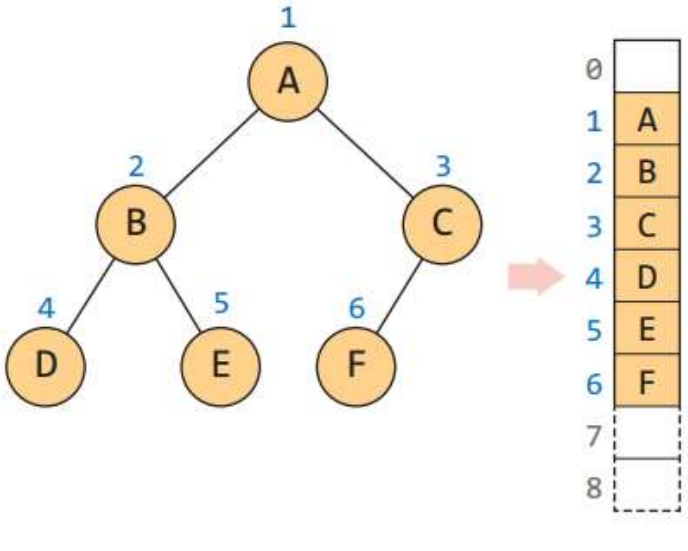
이진트리의 성질



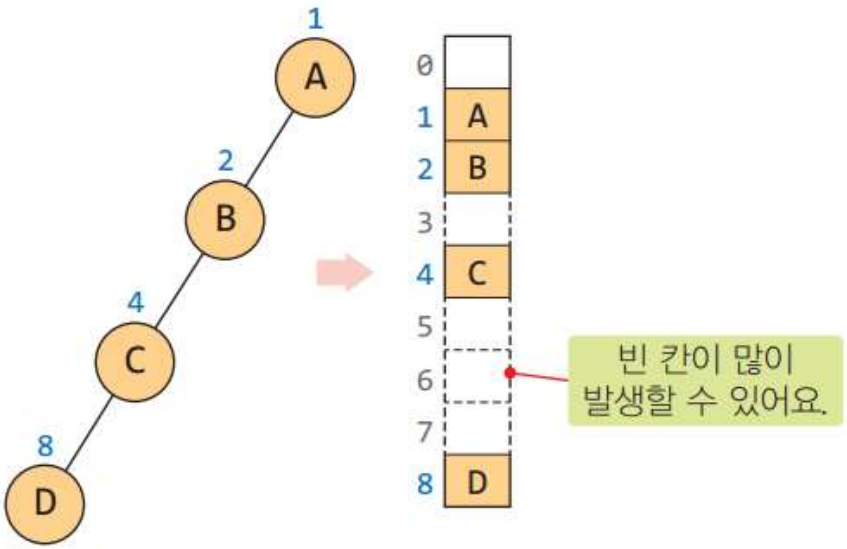
- n개 노드의 이진 트리 높이: $\lceil log_2(n+1) \rceil \sim n$



이진트리의 표현: 배열 표현법



완전이진트리의 배열 표현

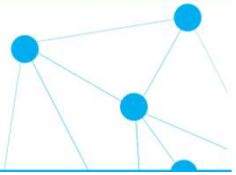


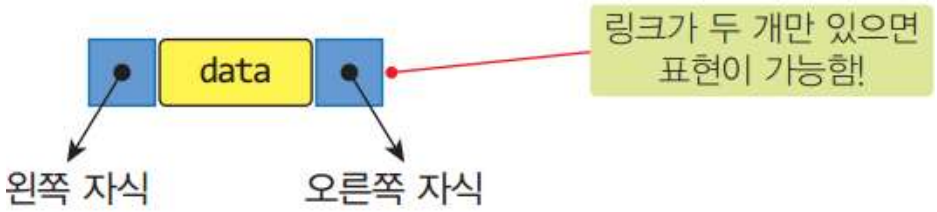
경사이진트리의 배열 표현

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

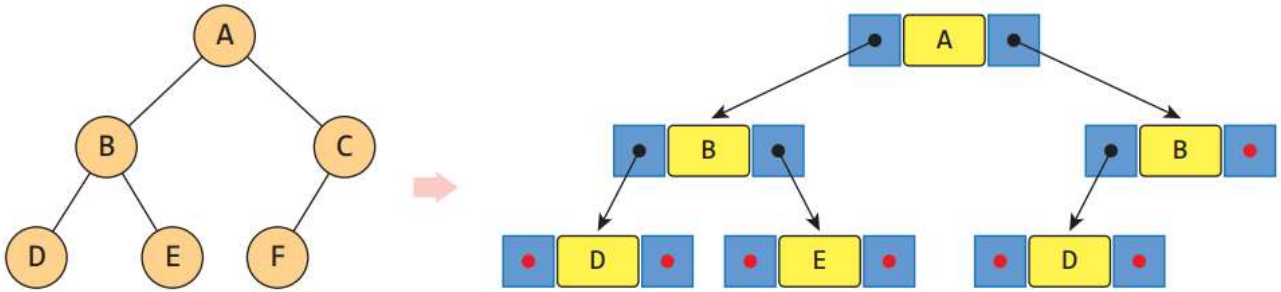
파이썬에서는 나눗셈 연산자가
/와 //로 구분되어 있습니다.
정수 나눗셈을 위해서는
i//2를 써야 합니다.

이진트리의 표현: 링크 표현법

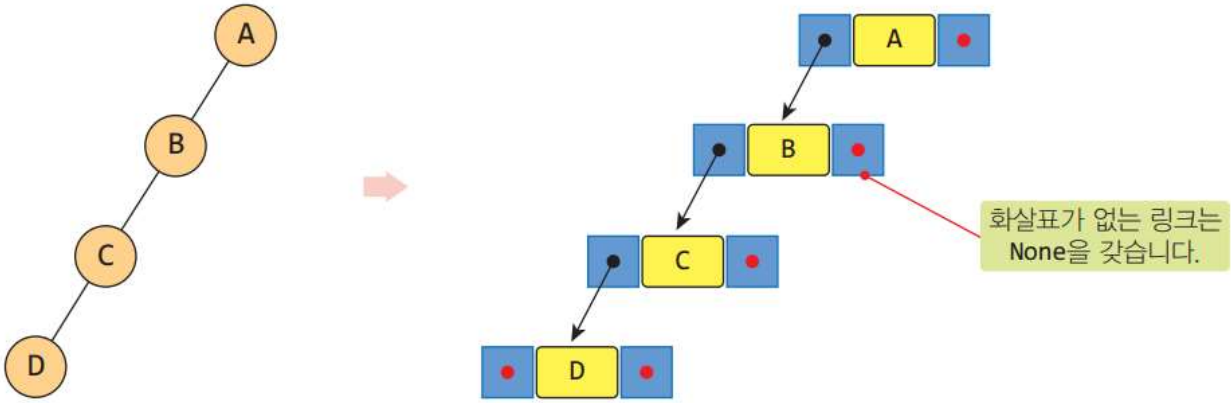





```
class TNode:
    def __init__ (self, data, left, right):
        self.data = data
        self.left = left
        self.right = right
```



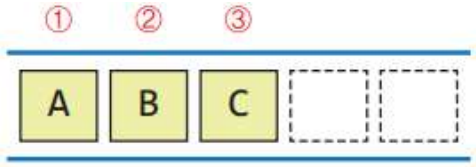
완전이진트리의 링크 표현



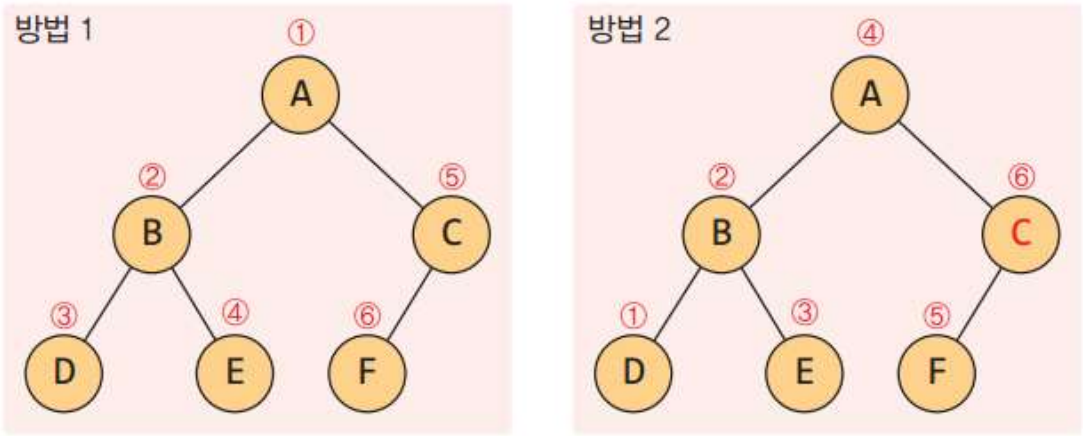
경사이진트리의 링크 표현

이진트리의 연산

- 순회(traversal)
  - 트리에 속하는 모든 노드를 한 번씩 방문하는 것
  - 선형 자료구조는 순회가 단순
  - 트리는 다양한 방법이 있음

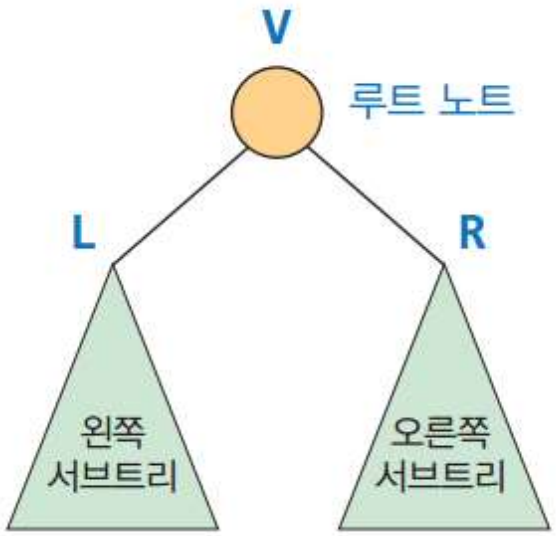


선형자료구조는
순회 방법이 단순하다.



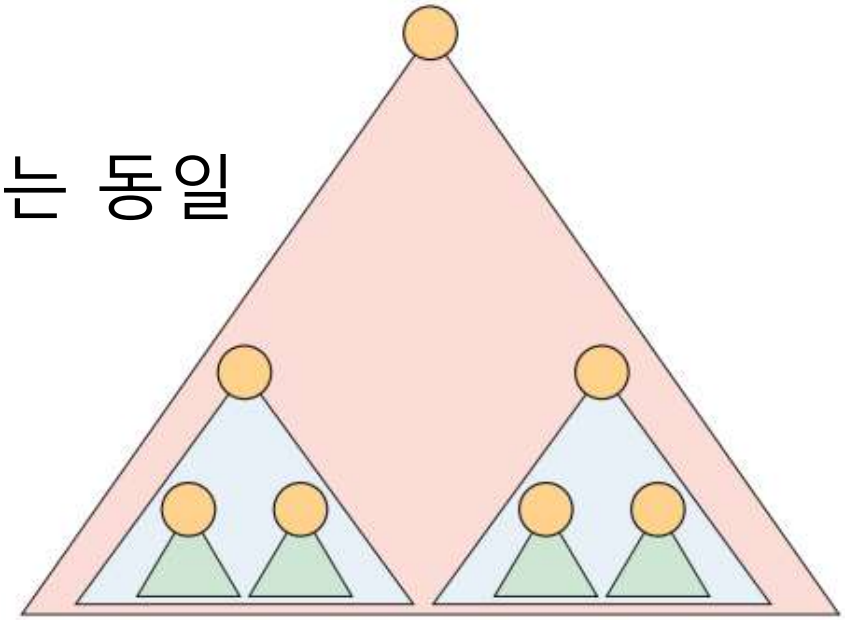
트리는 다양한 방법으로 순회할 수 있다.

이진트리의 기본 순회

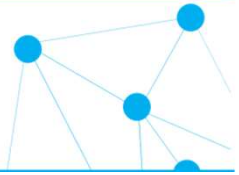


- 전위 순회(preorder traversal) : VLR
- 중위 순회(inorder traversal) : LVR
- 후위 순회(postorder traversal) : LRV

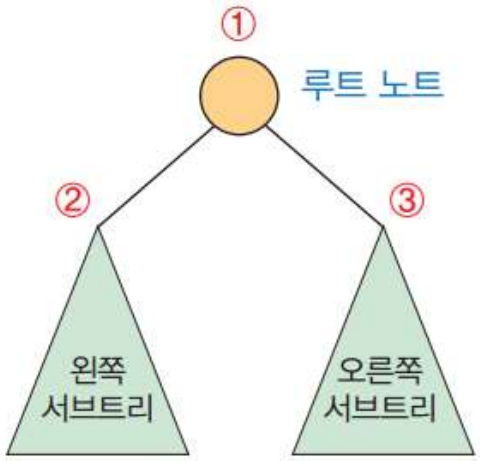
- 전체 트리나 서브 트리나 구조는 동일



전위 순회

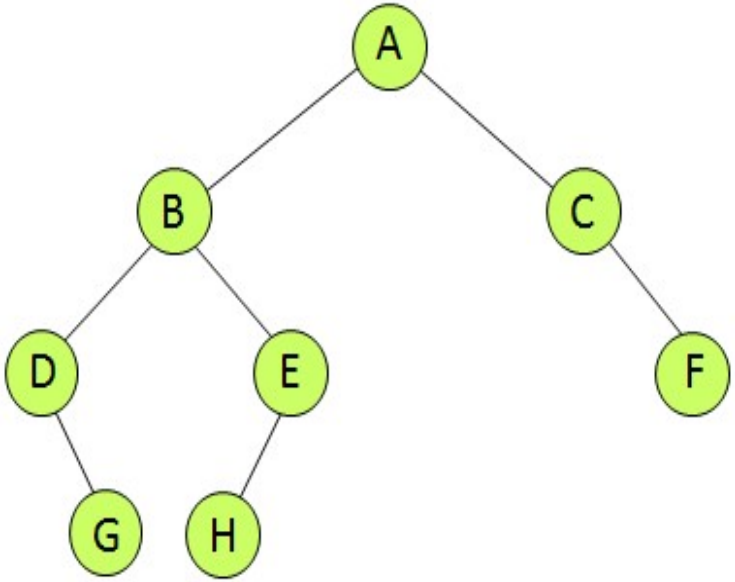


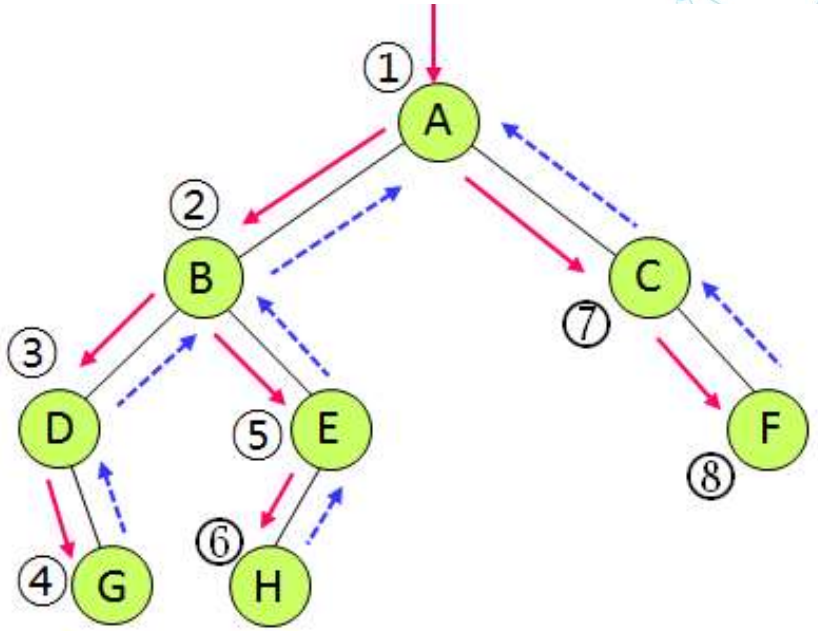
- 루트 → 왼쪽 서브트리 → 오른쪽 서브트리




```
def preorder(n) :                    # 전위 순회 함수
    if n is not None :
        print(n.data, end=' ')       # 먼저 루트노드 처리(화면 출력)
        preorder(n.left)             # 왼쪽 서브트리 처리
        preorder(n.right)            # 오른쪽 서브트리 처리
```

- 응용 예
  - 노드의 레벨 계산
  - 구조화된 문서 출력

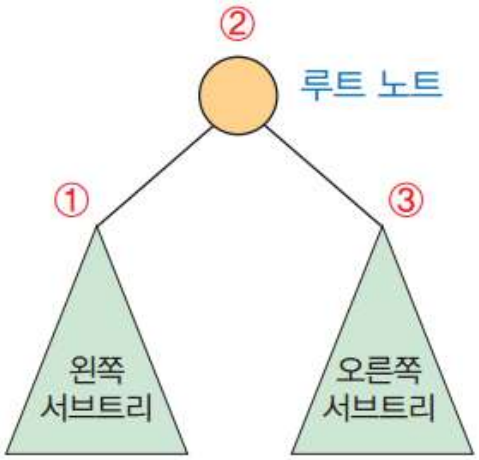




- 실선 화살표를 따라서A, B, D, G, E, H, C, F 순으로 방문
- 점선 화살표는 노드의 서브트리에 있는 모든 노드들을 방문한 후에 부모노드로 복귀
- 복귀하는 것은 프로그램에서 메소드 호출이 완료된 후에 리턴하는 것과 같음

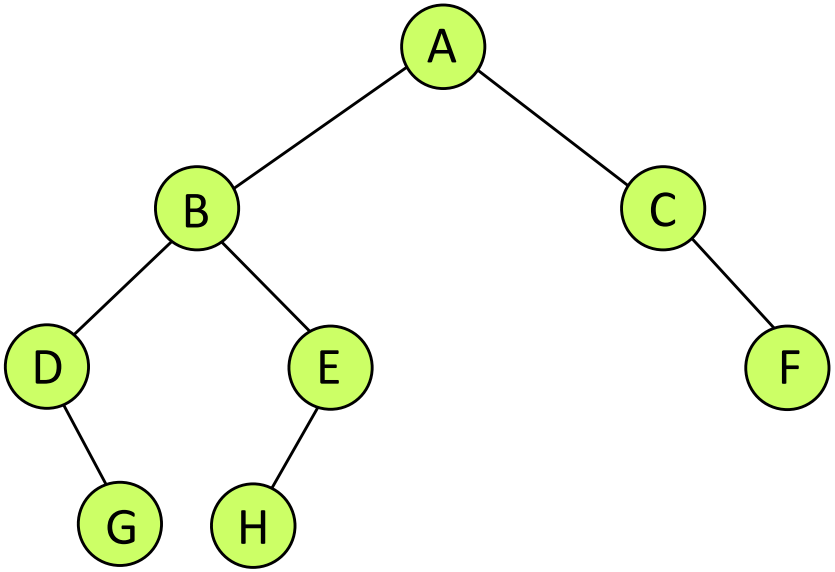
중위 순회

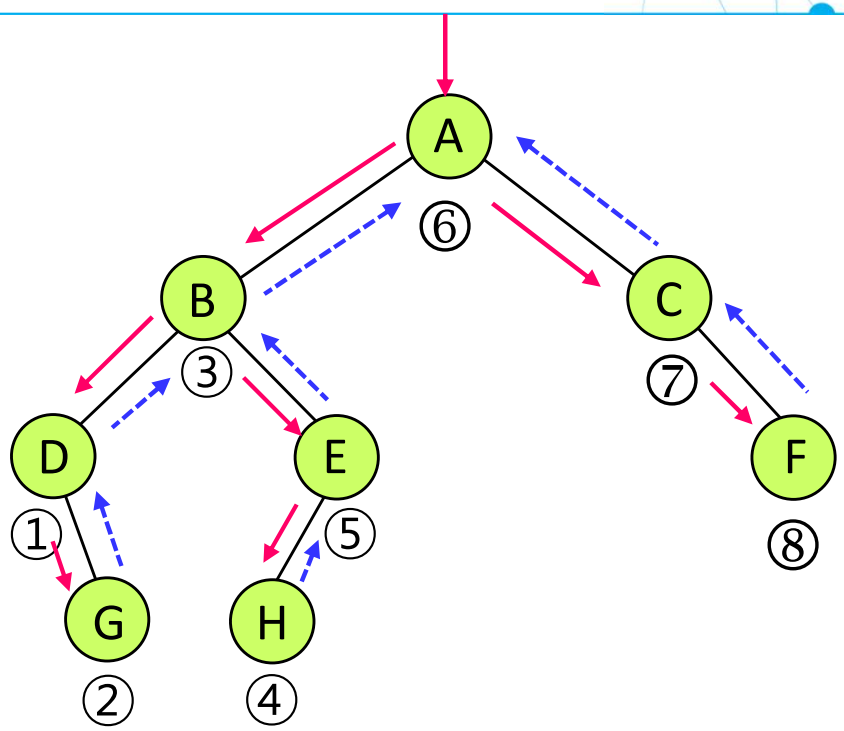
- 왼쪽 서브트리 → 루트 → 오른쪽 서브트리




```
def inorder(n) :                    # 중위 순회 함수
    if n is not None :
        inorder(n.left)             # 왼쪽 서브트리 처리
        print(n.data, end=' ')      # 루트노드 처리(화면 출력)
        inorder(n.right)            # 오른쪽 서브트리 처리
```

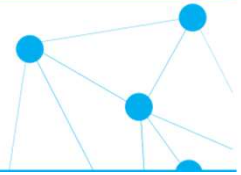
- 응용 예
  - 정렬



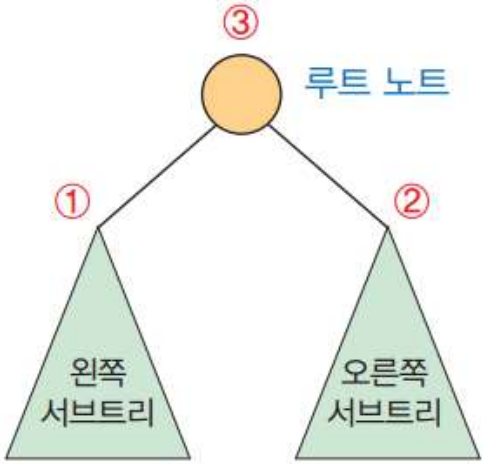


- 중위순회: D, G, B, H, E, A, C, F 순으로 방문

후위 순회

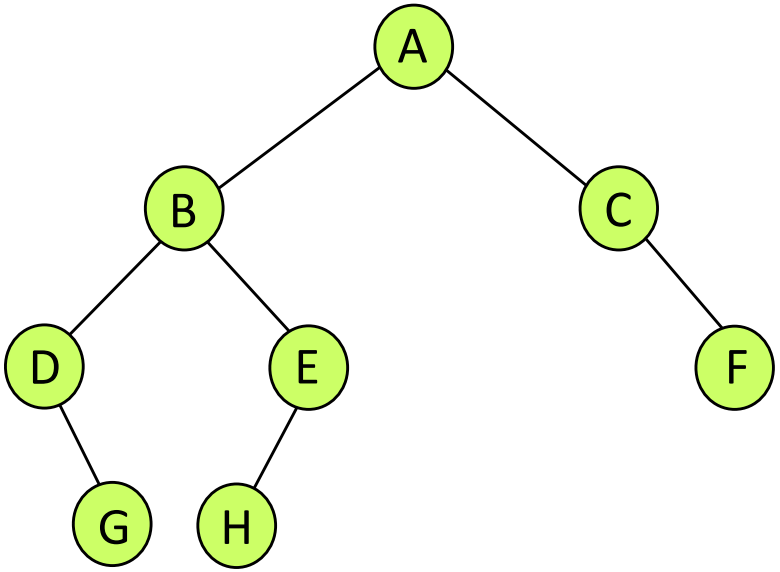


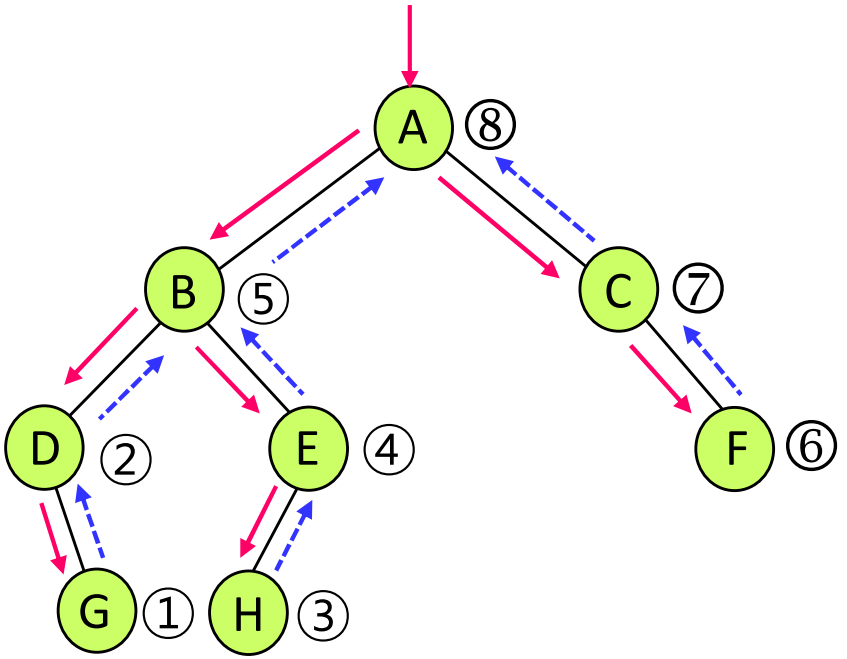
- 왼쪽 서브트리 → 오른쪽 서브트리 → 루트




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

- 응용 예
  - 폴더 용량 계산

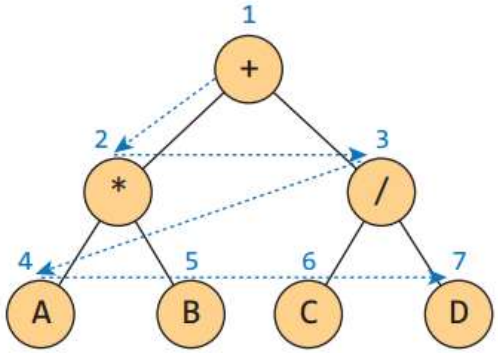


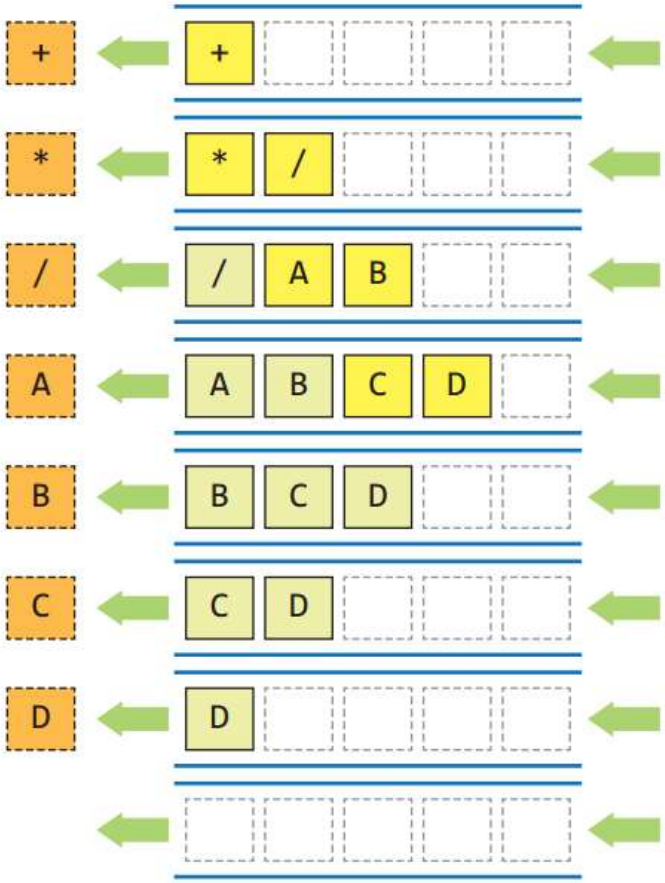


후위순회: G, D, H, E, B, F, C, A 순으로 방문

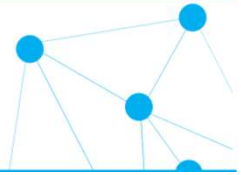
레벨 순회

- 노드를 레벨 순으로 검사하는 순회방법
  - 큐를 사용해 구현
  - 순환을 사용하지 않음





레벨 순회 알고리즘



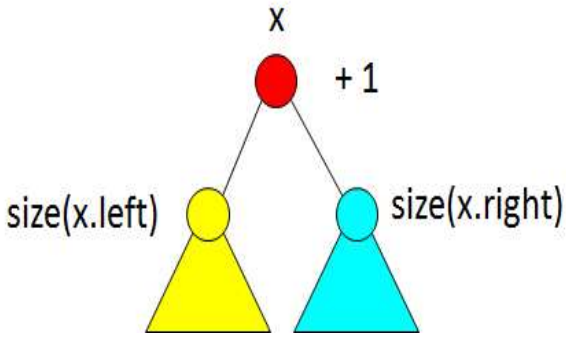
```
def levelorder(root) :
    queue = CircularQueue()              # 큐 객체 초기화
    queue.enqueue(root)                  # 최초에 큐에는 루트 노드만 들어있음.
    while not queue.isEmpty() :          # 큐가 공백상태가 아닌 동안,
        n = queue.dequeue()              # 큐에서 맨 앞의 노드 n을 꺼냄
        if n is not None :
          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

이진트리연산: 노드 개수, 단말 노드의 수

트리의 노드 수 = 1 +

(루트노드의 왼쪽 서브트리에 있는 노드 수) +

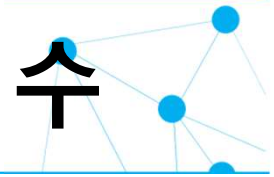
(루트노드의 오른쪽 서브트리에 있는 노드 수)



- 1은 루트노드 자신을 계산에 반영하는 것

```
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)
```

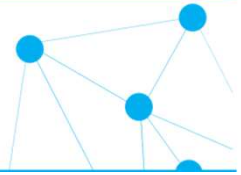
이진트리연산: 노드 개수, 단말 노드의 수



- 단말 노드의 수

```
def count_leaf(n) :
    if n is None :                                  # 공백 트리 --> 0을 반환
        return 0
    elif n.left is None and n.right is None :       # 단말노드 --> 1을 반환
        return 1
    else :                                          # 비단말 노드: 좌우 서브트리의 결과 합을 반환
        return count_leaf(n.left) + count_leaf(n.right)
```

이진트리연산 : 트리 높이

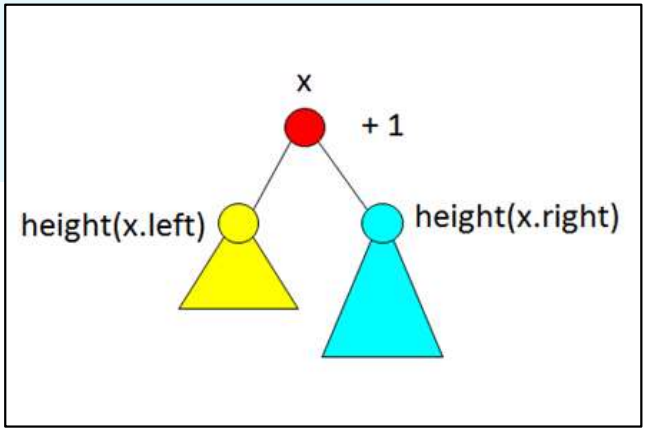


트리의 높이 = 1 +

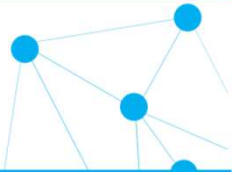
max (루트의 왼쪽 서브트리의 높이, 루트의 오른쪽 서브트리의 높이)

- 1은 루트노드 자신을 계산에 반영

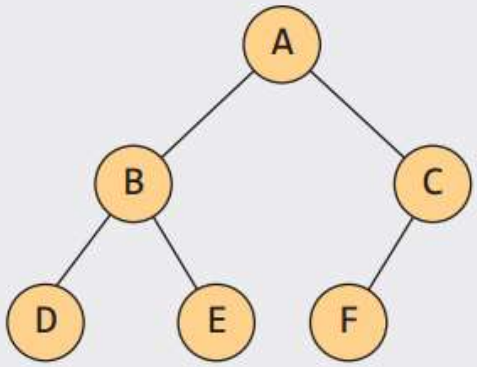
```
def calc_height(n) :
    if n is None :                      # 공백 트리 --> 0을 반환
        return 0
    hLeft = calc_height(n.left)         # 왼쪽 트리의 높이 --> hLeft
    hRight = calc_height(n.right)       # 오른쪽 트리의 높이 --> hRight
    if (hLeft > hRight) :               # 더 높은 높이에 1을 더해 반환.
        return hLeft + 1
    else:
        return hRight + 1
```



테스트 프로그램



```
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)
print('\n  Pre-Order : ', end='')
preorder(root)
print('\n Post-Order : ', end='')
postorder(root)
print('\nLevel-Order : ', end='')
levelorder(root)
print()

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




```
C:₩WINDOWS₩system32₩cmd.exe

  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 = 3
```

이진트리를 위한 BinaryTree 클래스

```
01 class BinaryTree:
02     class Node:
03         def __init__(self, item, left=None, right=None):
04             self.item  = item
05             self.left  = left
06             self.right = right
07
08     def __init__(self): # 트리 생성자
09         self.root = None
10
```

노드 생성자
항목과 왼쪽, 오른쪽 자식노드 레퍼런스

트리의 루트

```
11      def preorder(self, n): # 전위순회
12          if n != None:
13              print(str(n.item),' ', end='')
14              if n.left:
15                  self.preorder(n.left)
16              if n.right:
17                  self.preorder(n.right)
18
19      def inorder(self, n): # 중위순회
20          if n != None:
21              if n.left:
22                  self.inorder(n.left)
23              print(str(n.item),' ', end='')
24              if n.right:
25                  self.inorder(n.right)
26
27      def postorder(self, n): # 후위순회
28          if n != None:
29              if n.left:
30                  self.postorder(n.left)
31              if n.right:
32                  self.postorder(n.right)
33              print(str(n.item),' ', end='')
34
```

맨 먼저 노드 방문

왼쪽 서브트리 방문 후
오른쪽 서브트리 방문

왼쪽 서브트리 방문 후
노드 방문

왼쪽과 오른쪽 서브트리
모두 방문 후 노드 방문

```
35      def levelorder(self, root): # 레벨순회
36          q = []
37          q.append(root)
38          while len(q) != 0:
39              t = q.pop(0)
40              print(str(t.item), ' ', end='')
41              if t.left != None:
42                  q.append(t.left)
43              if t.right != None:
44                  q.append(t.right)
45
46      def height(self, root): # 트리 높이 계산
47          if root == None:
48              return 0
49          return max(self.height(root.left), self.height(root.right))+1
```

리스트로 큐 자료구조 구현

큐에서 첫 항목 삭제

삭제된 노드 방문

왼쪽자식, 오른쪽자식 큐에 삽입

두 자식노드의 높이 중 큰 높이 + 1

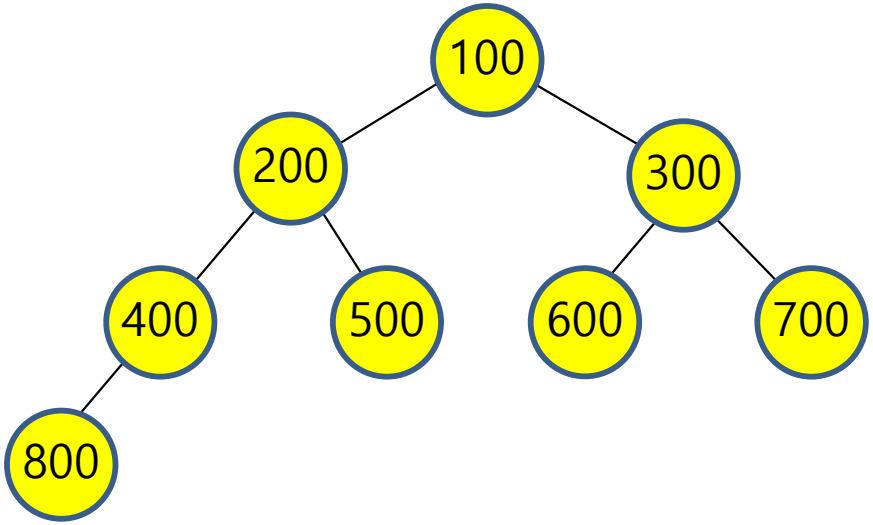
[프로그램 4-1] binary_tree.py

```
01 from binarytree import BinaryTree, Node
02 if __name__ == '__main__':
03     t = BinaryTree()
04     n1 = Node(100)
05     n2 = Node(200)
06     n3 = Node(300)
07     n4 = Node(400)
08     n5 = Node(500)
09     n6 = Node(600)
10     n7 = Node(700)
11     n8 = Node(800)
12     n1.left  = n2
13     n1.right = n3
14     n2.left  = n4
15     n2.right = n5
16     n3.left  = n6
17     n3.right = n7
18     n4.left  = n8
19     t.root = n1
```

이진트리 객체 생성

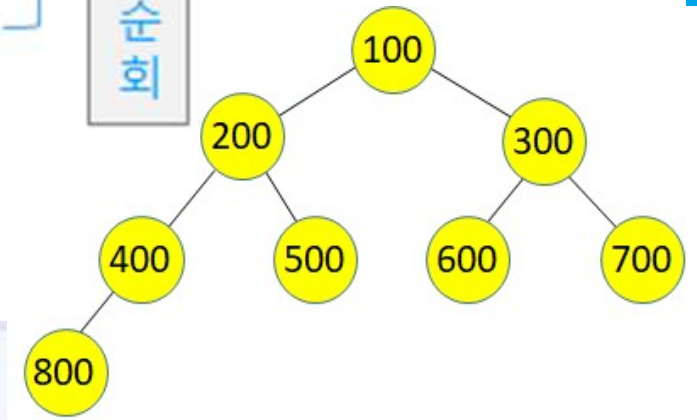
8개의 노드 생성

트리 만들기



```
20      print('트리 높이 =', t.height(t.root))
21      print('전위순회:\t', end='')
22      t.preorder(t.root)
23      print('\n중위순회:\t', end='')
24      t.inorder(t.root)
25      print('\n후위순회:\t', end='')
26      t.postorder(t.root)
27      print('\n레벨순회:\t', end='')
28      t.levelorder(t.root)
```

트리 높이 및 4가지 트리 순회



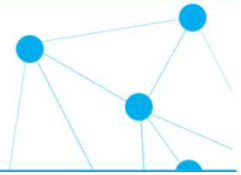
[프로그램 4-2] main.py

Console PyUnit

<terminated> main.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-3

```
트리 높이 = 4
전위순회:   100   200   400   800   500   300   600   700
중위순회:   800   400   200   500   100   600   300   700
후위순회:   800   400   500   200   600   700   300   100
레벨순회:   100   200   300   400   500   600   700   800
```

힙 트리



- 힙(Heap)이란?
  - "더미"와 모습이 비슷한 완전이진트리 기반의 자료 구조
  - 가장 큰(또는 작은) 값을 빠르게 찾아내도록 만들어진 자료 구조
  - 최대 힙, 최소 힙

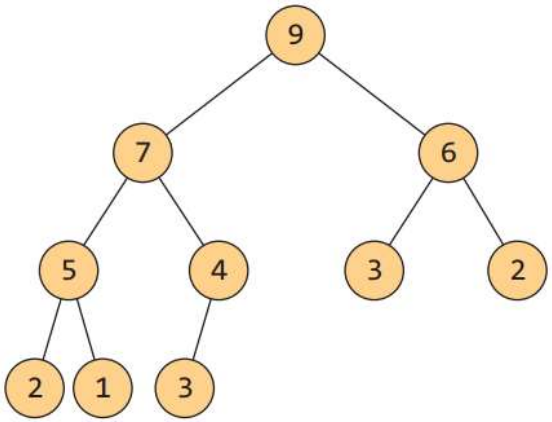
정의 8.2 최대 힙, 최소 힙의 정의

- 최대 힙(max heap): 부모 노드의 키 값이 자식 노드의 키 값보다 크거나 같은 완전이진트리 (key(부모노드) $\geq$ key(자식노드))
- 최소 힙(min heap): 부모 노드의 키 값이 자식 노드의 키 값보다 작거나 같은 완전이진트리 (key(부모노드) $\leq$ key(자식노드))

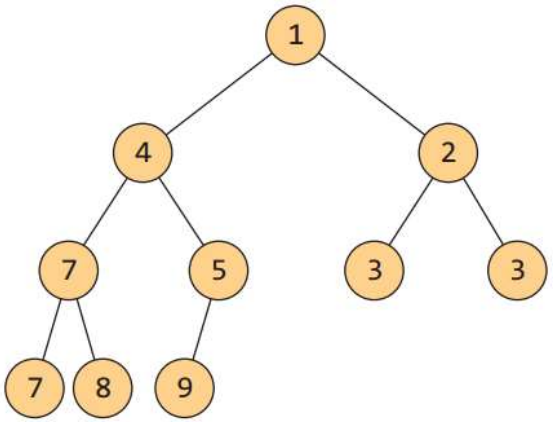
- 이진힙(Binary Heap)은 우선순위큐(Priority Queue)를 구현하는 가장 기본적인 자료구조이다.
- 우선순위큐(Priority Queue)
  - 가장 높은 우선순위를 가진 항목에 접근, 삭제와 임의의 우선순위를 가진 항목을 삽입을 지원하는 자료구조
- 스택이나 큐도 일종의 우선순위큐
  - 스택: 가장 마지막으로 삽입된 항목이 가장 높은 우선순위를 가지므로, 최근 시간일수록 높은 우선순위를 부여
  - 큐: 먼저 삽입된 항목이 우선순위가 더 높다. 따라서 이른 시간일수록 더 높은 우선순위를 부여

- 스택과 큐와 같은 우선순위큐가 있는데, 왜 또 다른 우선순위큐 자료구조가 필요할까?
  - 스택에 삽입되는 가장 마지막 항목의 우선순위는 스택에 있는 모든 항목들의 우선순위보다 높음
  - 큐에 가장 마지막 삽입되는 항목의 우선순위는 큐에 있는 모든 항목들의 우선수위보다 낮음
  - 삽입되는 항목이 임의의 우선순위를 가지면 스택이나 큐는 새 항목이 삽입될 때마다 저장되어 있는 항목들을 우선순위에 따라 정렬해야 하는 문제점이 있음

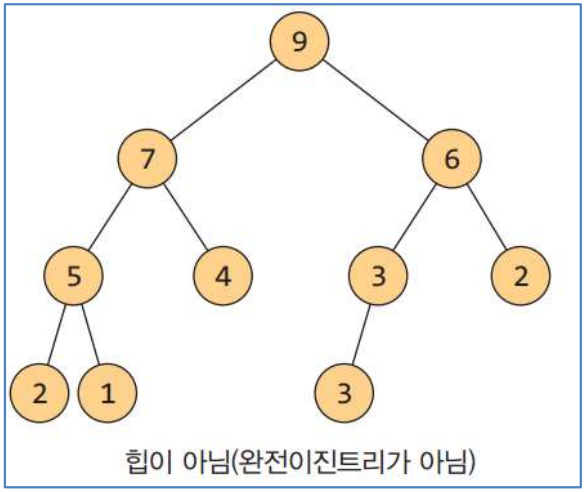
힙의 예



최대 힙(Max Heap)



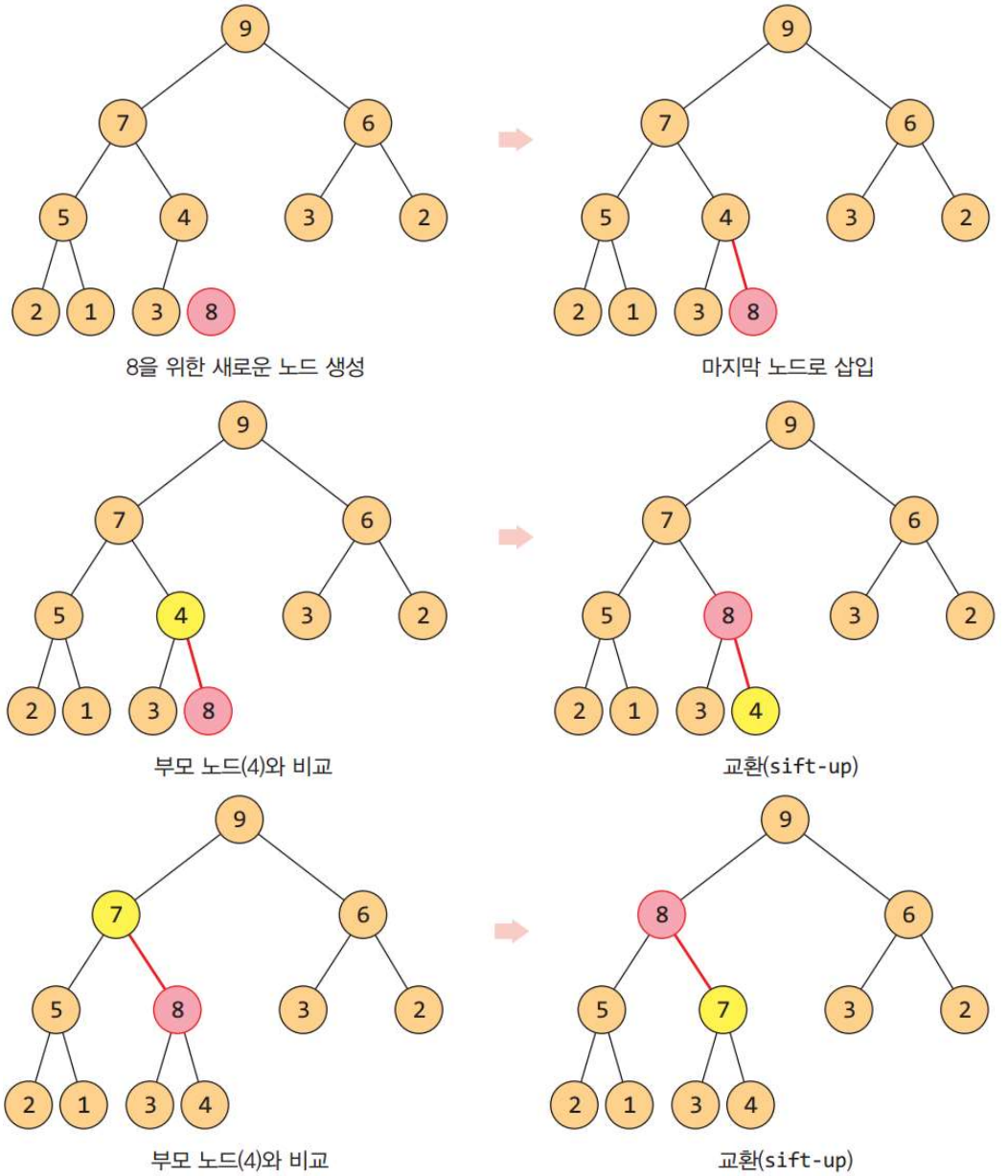
최소 힙(Min Heap)



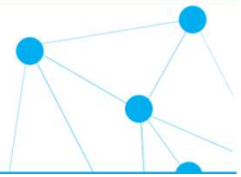
힙이 아님(완전이진트리가 아님)

힙의 연산: 삽입 연산

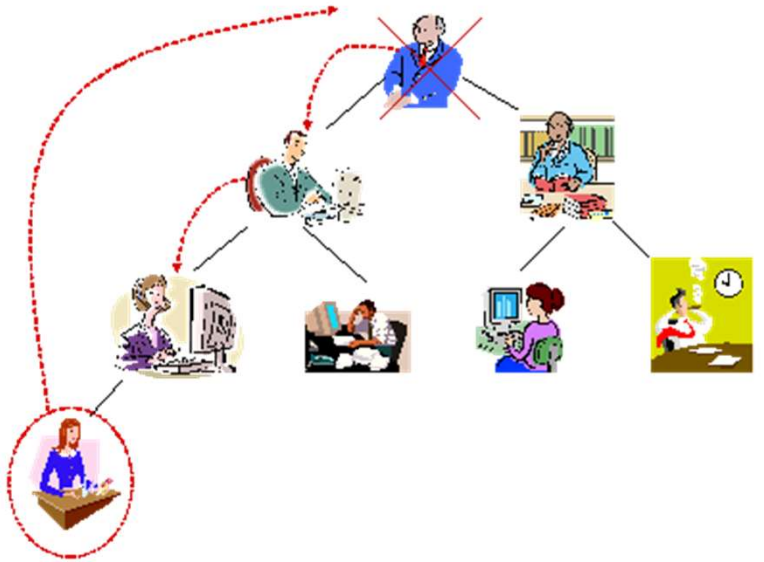
- Upheap
  - 회사에서 신입 사원이 들어오면 일단 말단 위치에 앉힘
  - 신입 사원의 능력을 봐서 위로 승진시킴
  - 시간 복잡도: $O(logn)$



삭제 연산



- 최대힙에서의 삭제 ➔ 항상 루트가 삭제됨
  - 가장 큰 키값을 가진 노드를 삭제하는 것
  - 시간 복잡도: $O(logn)$

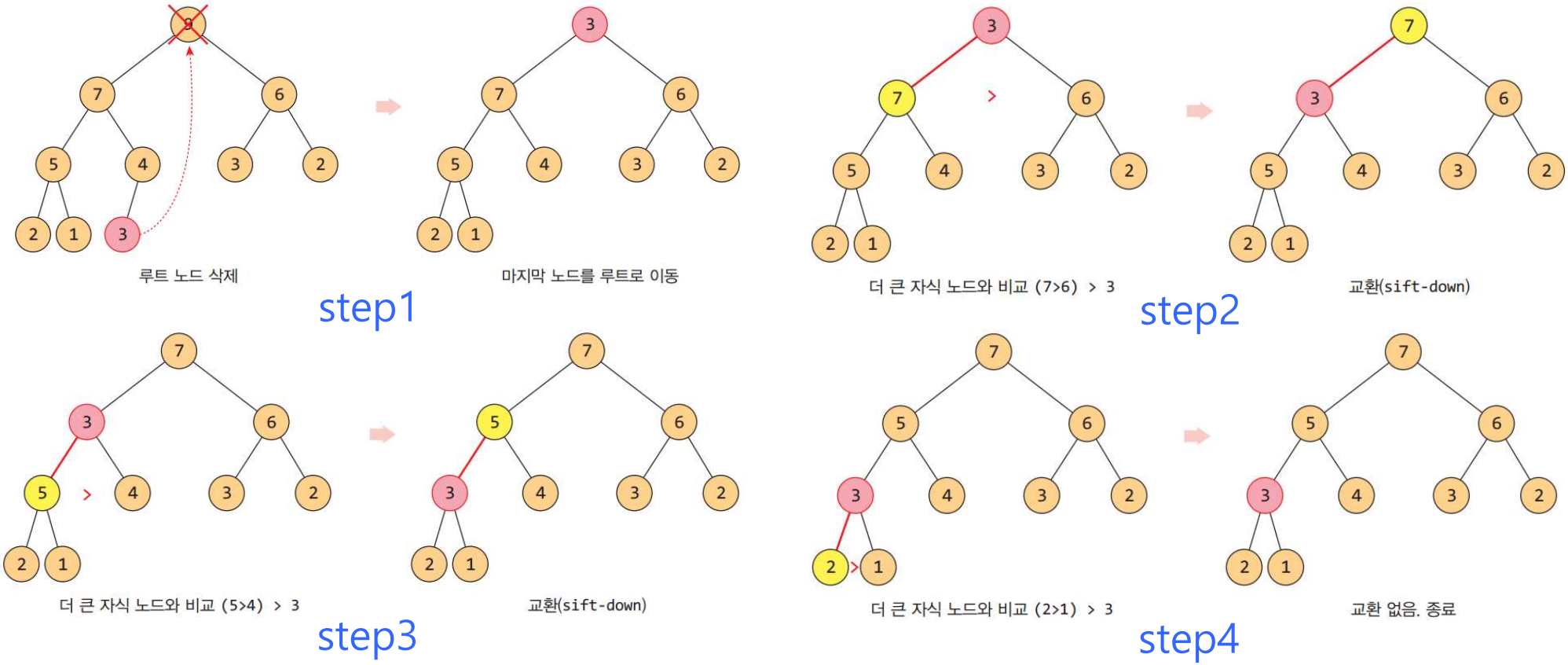


- 방법: downheap
  - 루트 삭제
  - 회사에서 사장의 자리가 비게 됨
  - 말단 사원을 사장 자리로 올림
  - 능력에 따라 강등 반복

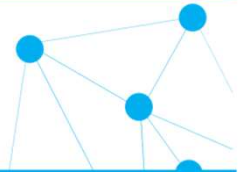
루트에서부터 단말노드까지의 경로에 있는 노드들을 교환하여 히프 성질을 만족시킨다.

힙의 연산: 삭제 연산

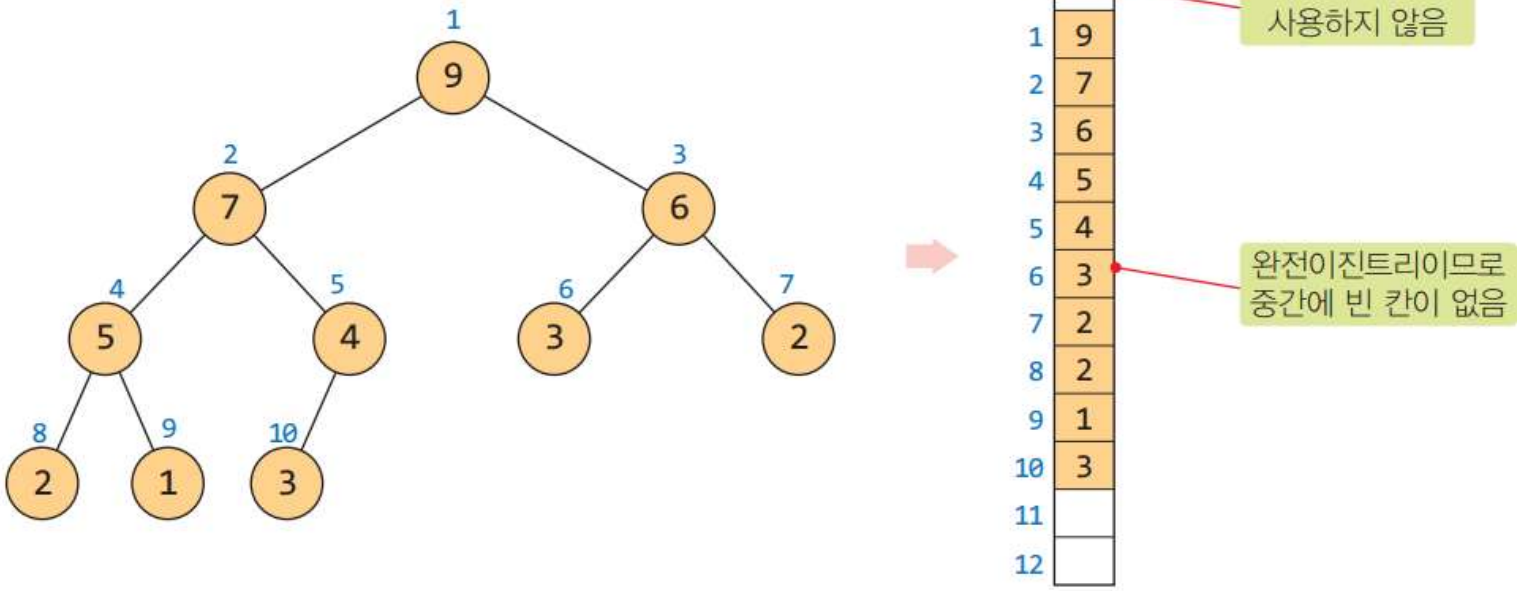
- Downheap



힙의 구현: 배열 구조

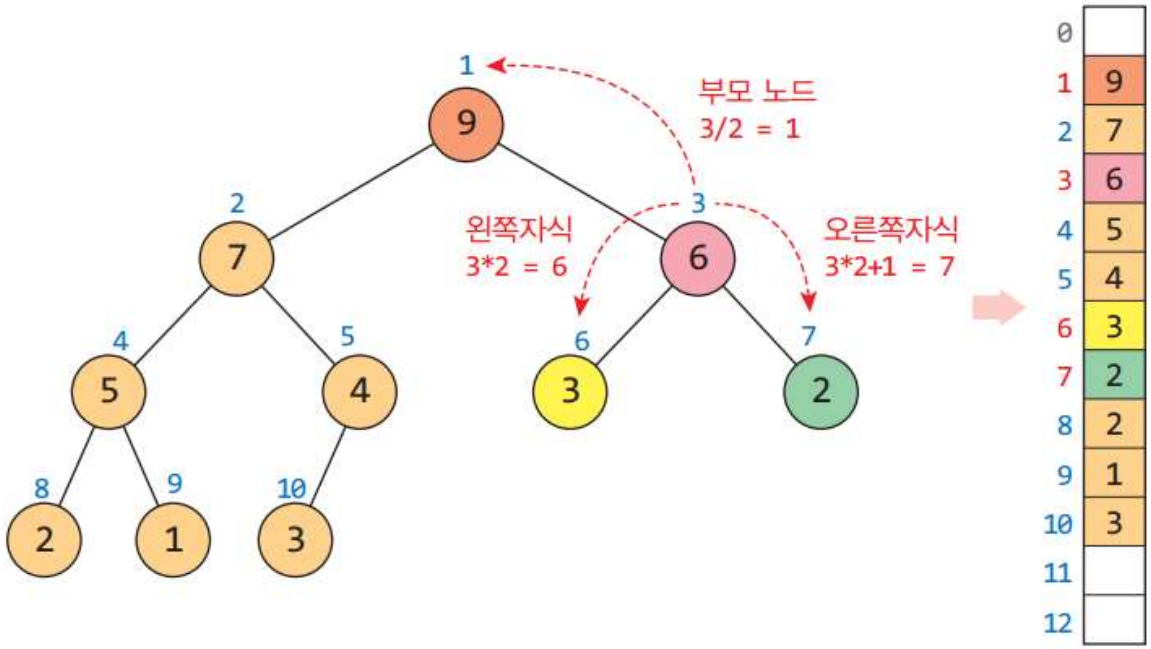


- 힙은 보통 배열을 이용하여 구현
  - 완전이진트리 ➔ 각 노드에 번호를 붙임 ➔ 배열의 인덱스

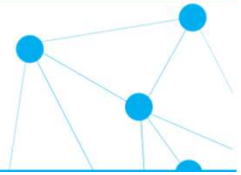


힙의 구현

- 부모노드와 자식노드의 관계
  - 왼쪽 자식의 인덱스 = (부모의 인덱스)*2
  - 오른쪽 자식의 인덱스 = (부모의 인덱스)*2 + 1
  - 부모의 인덱스 = (자식의 인덱스)/2



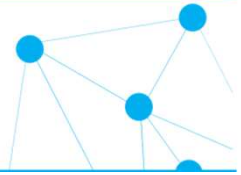
최대 힙의 구현



- 최대 힙 클래스

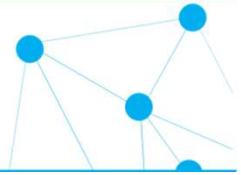
```
class MaxHeap :                                  # 최대힙 클래스
    def __init__ (self) :                        # 생성자
        self.heap = []                           # 리스트(배열)를 이용한 힙
        self.heap.append(0)                      # 0번 항목은 사용하지 않음

    def size(self) : return len(self.heap) - 1   # 힙의 크기
    def isEmpty(self) : return self.size() == 0  # 공백 검사
    def Parent(self, i) : return self.heap[i//2] # 부모노드 반환
    def Left(self, i) : return self.heap[i*2]    # 왼쪽 자식 반환
    def Right(self, i) : return self.heap[i*2+1] # 오른쪽 자식 반환
    def display(self, msg = '힙 트리: ') :
        print(msg, self.heap[1:])                # 파이썬 리스트의 슬라이싱 이용
```

최대 힙: 삽입 연산



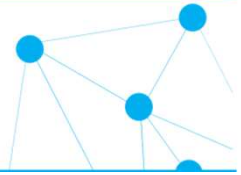
```
def insert(self, n) :
    self.heap.append(n)                    # 맨 마지막 노드로 일단 삽입
    i = self.size()                        # 노드 n의 위치
    while (i != 1 and n > self.Parent(i)):  # 부모보다 큰 동안 계속 업힙
        self.heap[i] = self.Parent(i)      # 부모를 끌어내림
        i = i // 2                         # i를 부모의 인덱스로 올림
    self.heap[i] = n                       # 마지막 위치에 n 삽입
```

최대 힙: 삭제 연산



```
def delete(self) :
    parent = 1
    child = 2
    if not self.isEmpty() :
        hroot = self.heap[1]                    # 삭제할 루트를 복사해 둠
        last = self.heap[self.size()]           # 마지막 노드
        while (child <= self.size()):           # 마지막 노드 이전까지
            # 만약 오른쪽 노드가 더 크면 child를 1 증가 (기본은 왼쪽 노드)
            if child<self.size() and self.Left(parent)<self.Right(parent):
                child += 1
            if last >= self.heap[child] :       # 더 큰 자식이 더 작으면
                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                # 맨 마지막 노드를 parent위치에 복사
        self.heap.pop(-1)                       # 맨 마지막 노드 삭제
        return hroot                            # 저장해두었던 루트를 반환
```

테스트 프로그램



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

```
선택 C:₩WINDOWS₩system32₩cmd.exe
[삽입 연산] : [2, 5, 4, 8, 9, 3, 7, 3]
[ 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]
```

상향식 힙만들기(Bottom-up Heap Construction)

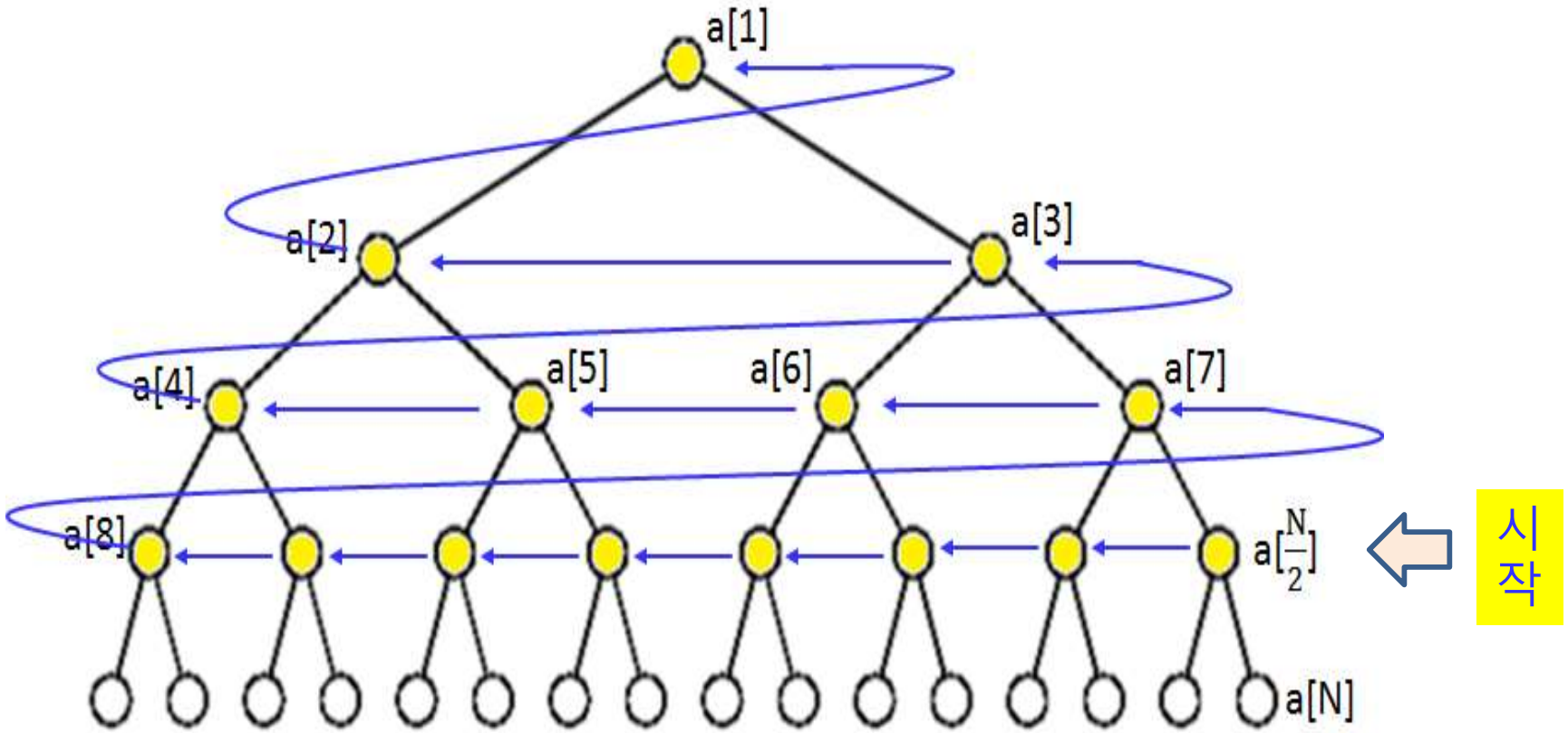
[핵심 아이디어]

- 상향식 방식으로 각 노드에 대해 힙속성을 만족하도록 부모와 자식을 서로 교환
- N개의 항목이 리스트에 임의의 순서로 저장되어 있을 때, 힙을 만들기 위해선 a[N//2]부터 a[1]까지 차례로 downheap을 각각 수행하여 힙속성을 충족시킨다.

- a[N//2+1]~a[N]에 대하여 downheap을 수행하지 않는 이유:

  이 노드들 각각은 이파리이므로, 각 노드 스스로가 힙의 크기가 1인 최소힙이기 때문

상향식 힙을 만드는 순서



이진힙을 위한 BHeap 클래스

```
01 class BHeap:
02     def __init__(self, a):
03         self.a = a
04         self.N = len(a) - 1
05
06     def create_heap(self): # 초기 힙 만들기
07         for i in range(self.N//2, 0, -1):
08             self.downheap(i)
09
```

이진힙 생성자
리스트 a
항목 수 N

heapq.heapify()와 동일함

```
10      def insert(self, key_value): # 삽입 연산
11          self.N += 1
12          self.a.append(key_value)
13          self.upheap(self.N)
14
```

heapq.push()와 동일함

새 항목을 힙 마지막에 추가

힙속성 회복시키기위해

```
15      def delete_min(self): # 최솟값 삭제
16          if self.N == 0:
17              print('힙이 비어 있음')
18              return None
19          minimum = self.a[1]
20          self.a[1], self.a[-1] = self.a[-1], self.a[1]
21          del self.a[-1]
22          self.N -= 1
23          self.downheap(1)
24          return minimum
25
```

heapq.pop()과 동일함

a[1]과 a[N] 교환

힙속성 회복시키기위해

```
26      def downheap(self, i): # 힙 내려가며 힙속성 회복
27          while 2*i <= self.N:
28              k = 2*i
29              if k < self.N and self.a[k][0] > self.a[k+1][0]:
30                  k += 1
31              if self.a[i][0] < self.a[k][0]:
32                  break
33              self.a[i], self.a[k] = self.a[k], self.a[i]
34              i = k
35
```

왼쪽, 오른쪽자식 중에서 승자 결정

힙속성 만족하면 루프 나가기

자식 승자와 현재 노드 교환

```
36      def upheap(self, j): # 힙 올라가며 힙속성 회복
37          while j > 1 and self.a[j//2][0] > self.a[j][0]:
38              self.a[j], self.a[j//2] = self.a[j//2], self.a[j]
39              j = j//2
40
41      def print_heap(self): # 힙 출력
42          for i in range(1, self.N+1):
43              print('[%2d' % self.a[i][0], self.a[i][1], ']', end='')
44          print('\n힙 크기 = ', self.N)
```

부모와 자식 교환

현재 노드가 한 층 올라감

[프로그램 4-3]
binary_heap.py

[프로그램 4-4] main.py

```
01 from binaryheap import BHeap
02 if __name__ == '__main__':
03     a = [None] * 1
04     a.append([90, 'watermelon'])
05     a.append([80, 'pear'])
06     a.append([70, 'melon'])
07     a.append([50, 'lime'])
08     a.append([60, 'mango'])
09     a.append([20, 'cherry'])
10     a.append([30, 'grape'])
11     a.append([35, 'orange'])
12     a.append([10, 'apricot'])
13     a.append([15, 'banana'])
14     a.append([45, 'lemon'])
15     a.append([40, 'kiwi'])
16     b = BHeap(a)
17     print('힙 만들기 전:')
18     b.print_heap()
19     b.create_heap()
20     print('최소힙:')
21     b.print_heap()
22     print('최솟값 삭제 후')
23     print(b.delete_min())
24     b.print_heap()
25     b.insert([5,'apple'])
26     print('5 삽입 후')
27     b.print_heap()
```

12개 항목의 리스트 생성

힙 객체 생성

힙 만들기 삭제 삽입 연산

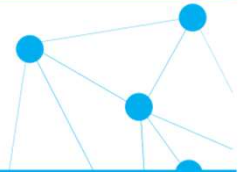
프로그램의 수행결과

Console PyUnit

<terminated> main.py [C:₩Users₩sbyang₩AppData₩Local₩Programs₩Python₩Python36-32₩python.exe]

```
힙 만들기 전:
[90 watermelon][80 pear][70 melon][50 lime][60 mango][20 cherry][30 grape][35 orange][10 apricot][15 banana][45 lemon][40 kiwi]
힙 크기 =  12
최소힙:
[10 apricot][15 banana][20 cherry][35 orange][45 lemon][40 kiwi][30 grape][80 pear][50 lime][60 mango][90 watermelon][70 melon]
힙 크기 =  12
최솟값 삭제 후
[10, 'apricot']
[15 banana][35 orange][20 cherry][50 lime][45 lemon][40 kiwi][30 grape][80 pear][70 melon][60 mango][90 watermelon]
힙 크기 =  11
5 삽입 후
[ 5 apple][35 orange][15 banana][50 lime][45 lemon][20 cherry][30 grape][80 pear][70 melon][60 mango][90 watermelon][40 kiwi]
힙 크기 =  12
```

힙의 복잡도 분석



- 삽입 연산에서 최악의 경우
  - 루트 노드까지 올라가야 하므로 트리의 높이에 해당하는 비교 연산 및 이동 연산이 필요하다.

  ➔ O(logn)

- 삭제연산 최악의 경우
  - 가장 아래 레벨까지 내려가야 하므로 역시 트리의 높이 만큼의 시간이 걸린다.

  ➔ O(logn)

|파이썬 heapq|

파이썬은 우선순위큐를 위한 heapq를 라이브러리로 제공

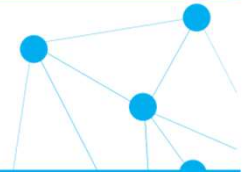
heapq에 선언된 메소드

- heapq.heappush(*heap*, *item*) # insert() 메소드와 동일
- heapq.heappop(*heap*) # delete_min() 메소드와 동일
- heapq.heappushpop(*heap*, *item*) # item 삽입 후 delete_min() 수행
- heapq.heapify(*x*) # create_heap() 메소드와 동일
- heapq.heapreplace(*heap*, *item*) # delete_min() 먼저 수행 후, *item* 삽입

이외에도 몇 개의 다른 메소드들이 있으나 힙의 항목 수가 많아지면 이 연산들은 매우 비효율적이서 사용하지 말 것을 권고

Applications

- 관공서, 은행, 병원, 우체국, 대형 마켓, 공항 등에서 이루어지는 업무와 관련된 이벤트 처리
- 컴퓨터 운영체제의 프로세스 처리
- 네트워크 라우터에서의 패킷 처리 등
- 실시간 급상승 검색어(데이터 스트림에서 Top k 항목 유지) 제공
- 허프만 코딩
- 힙정렬
- Prim의 최소신장트리 알고리즘과 Dijkstra의 최단경로 알고리즘에도 활용



감사합니다