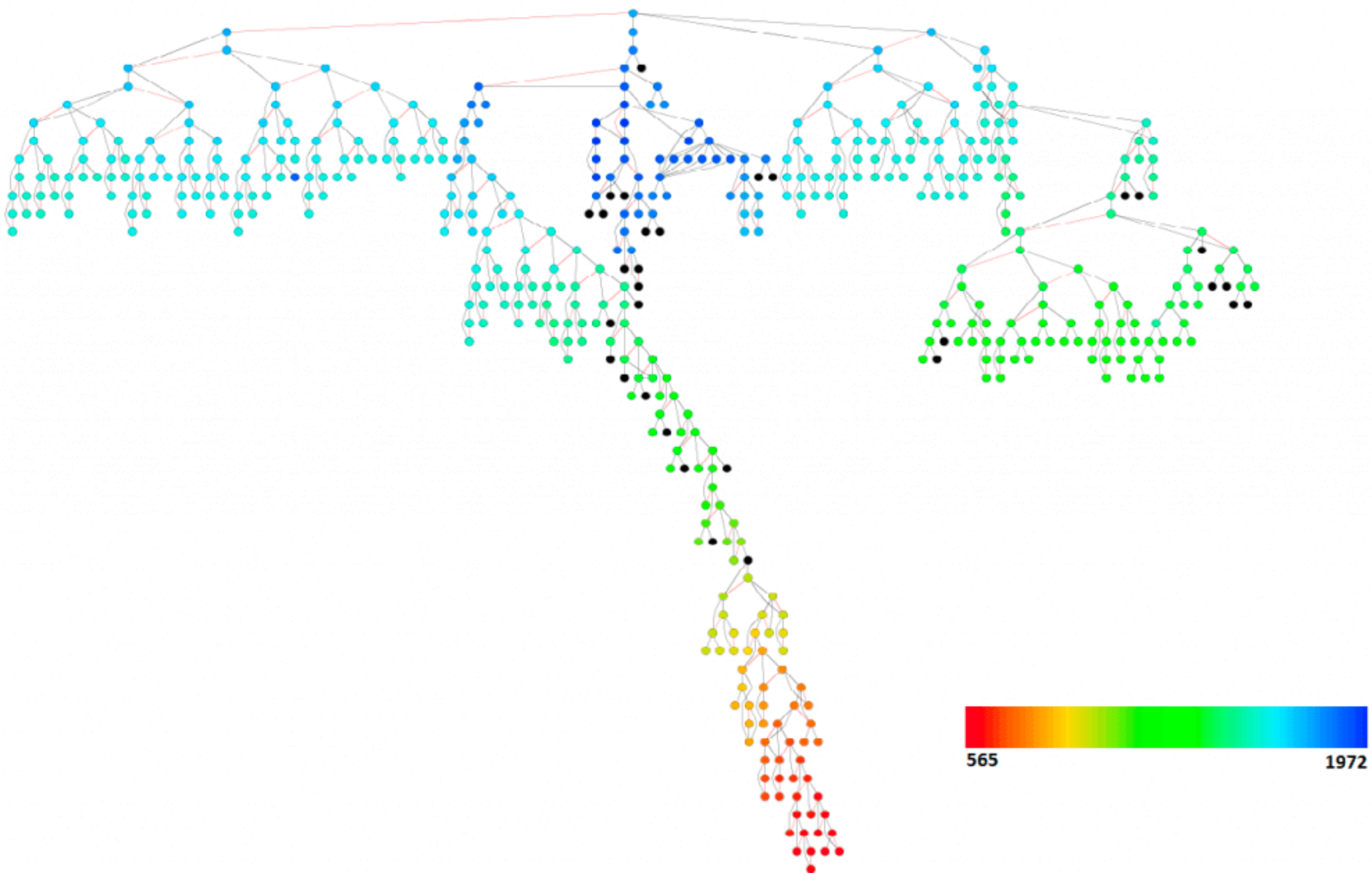


Tree

Jin Hyun Kim

Fall, 2019

In this Topic

- Tree
- Tree Traverse
- Threaded Tree

리스트의 단점

- 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

트리

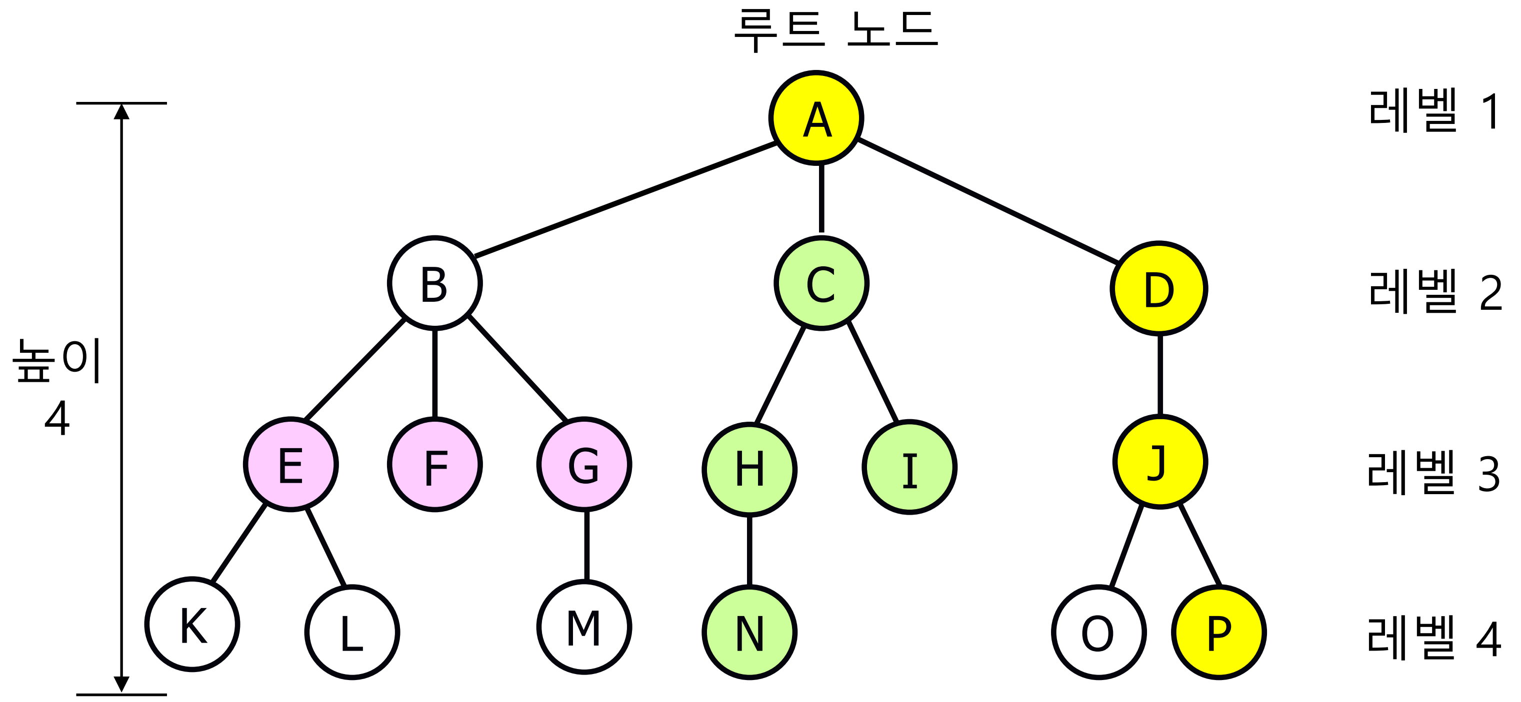
- 조직이나 기관의 계층구조
- 컴퓨터 운영체제의 파일 시스템
- 자바 클래스 계층구조 등
- 트리는 일반적인 트리와 이진트리(Binary Tree)로 구분
- 다양한 탐색트리(Search Tree), 힙(Heap) 자료구조, 컴파일러의 수식을 위한 구문트리(Syntax Tree) 등의 기본이 되는 자료구조로서 광범위하게 응용

트리

- 일반적인 트리(General Tree)는 실제 트리를 거꾸로 세워 놓은 형태의 자료구조
- HTML과 XML 의 문서 트리, 자바 클래스 계층구조, 운영체제의 파일시스템, 탐색트리, 이항(Binomial)힙, 피보나치(Fibonacci)힙에서 사용

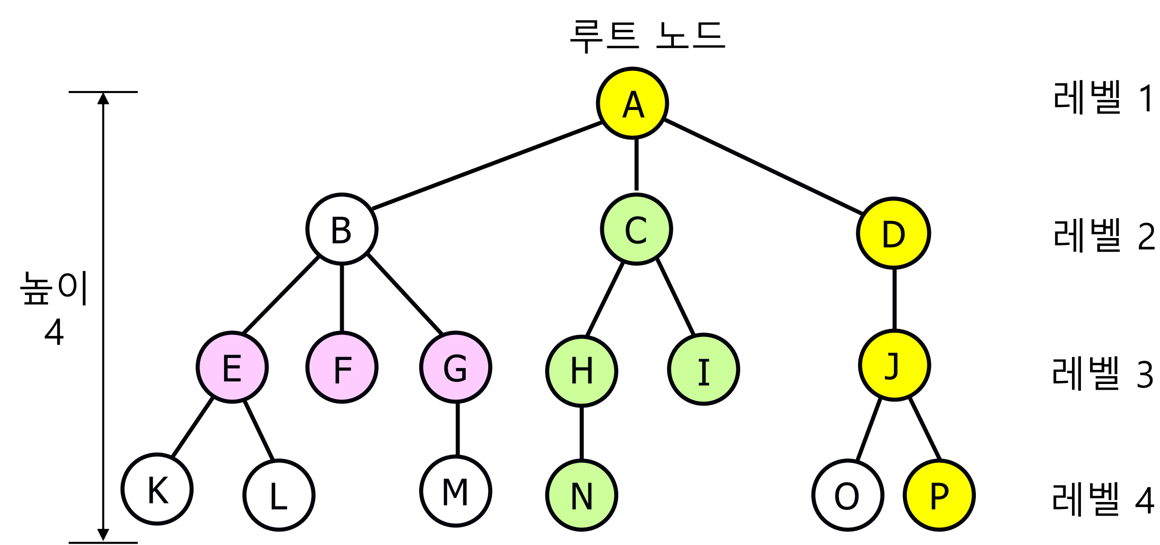
트리

- 일반적인 트리의 정의
  - 트리는 empty이거나, empty가 아니면 루트노드 R과 트리의 집합으로 구성되는데 각 트리의 루트노드는 R의 자식노드이다. 단, 트리의 집합은 공집합일 수도 있다



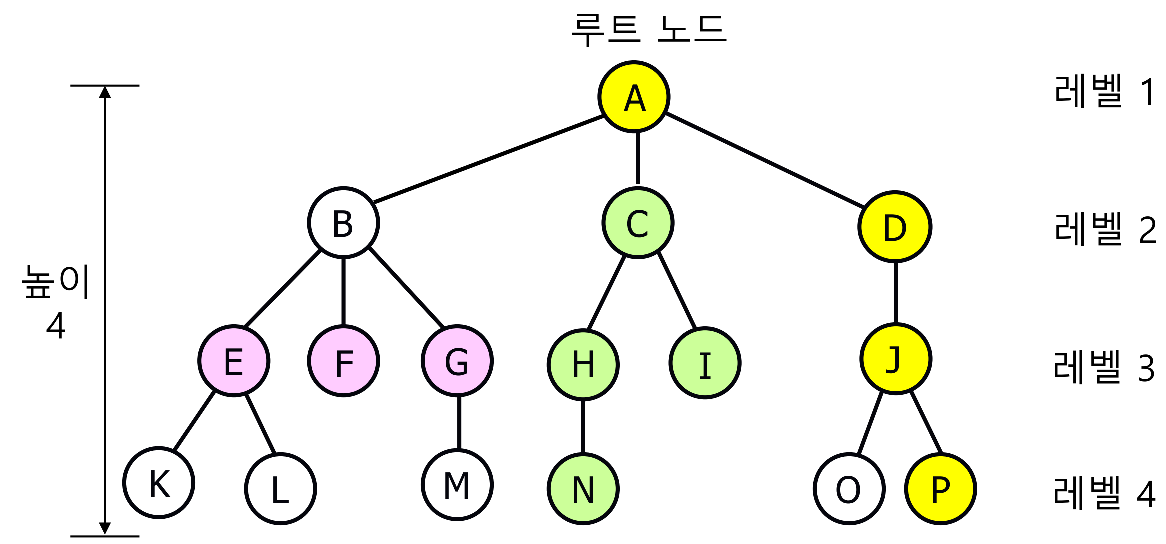
용어

- 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



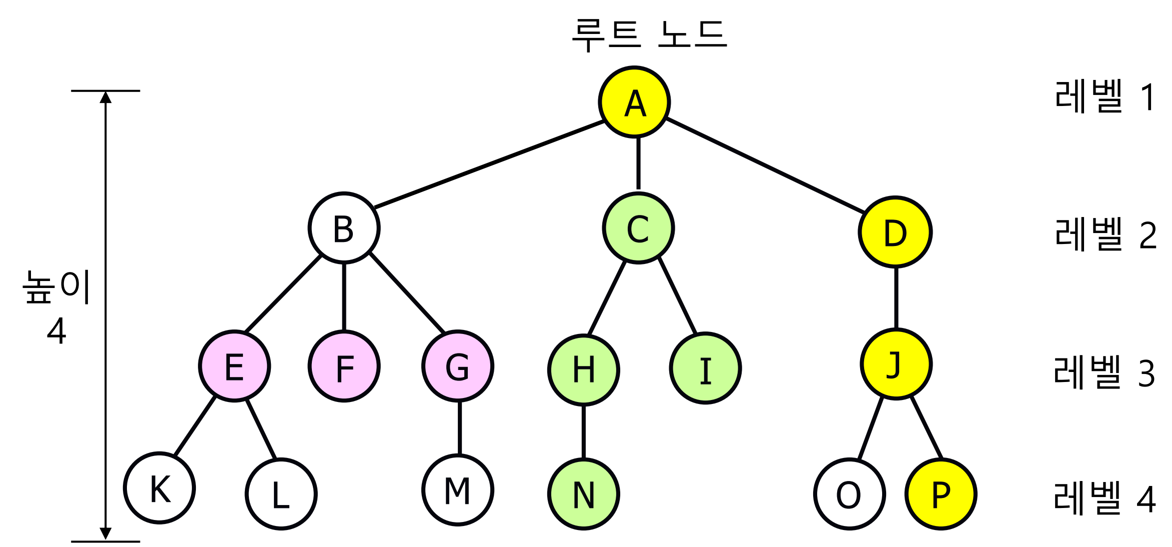
용어

- 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



용어

- A: 트리의 루트
- B, C, D: 각각 A의 자식
- A의 차수: 3
- B, C, D의 부모: A
- K, L, F, M, N, I, O, P: 이파리들
- E, F, G의 부모가 B로 모두 같으므로 이들은 서로 형제
- {B, C, D}, {H, I}, {K, L}, {O, P}도 각각 서로 형제들
- C의 자손: {H, I, N}
- C를 루트로 하는 서브트리는 C와 C의 자손들로 구성된 트리
- P의 조상: {J, D, A}
- 트리 높이: 4

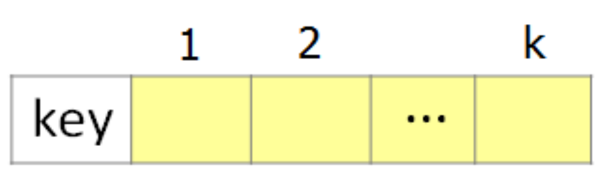


용어

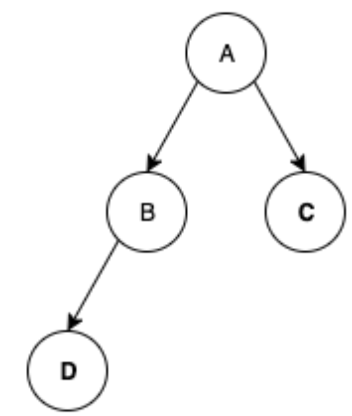
- 이파리: 단말(Terminal)노드 또는 외부(External)노드
- 내부(Internal)노드 또는 비 단말(Non-Terminal)노드: 이파리가 아닌 노드

특성

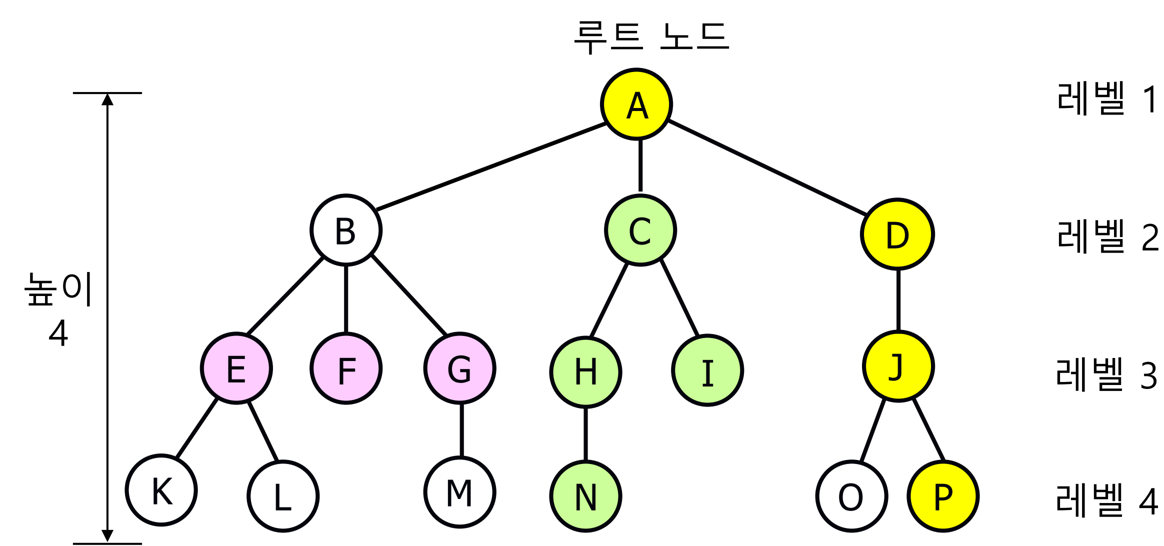
- 일반적인 트리를 메모리에 저장하려면 각 노드에 키와 자식 수만큼의 레퍼런스 저장 필요
    - 노드의 최대 차수가 k라면, k개의 레퍼런스 필드를 다음과 같이 선언해야



- N개의 노드가 있는 최대 차수가 k인 트리
    - None 레퍼런스 수 (자식으로 연결이 없는 연결의 수) = Nk – (N-1) = N(k-1) + 1
        - Nk = 총 레퍼런스의 수
        - **(N-1) = 트리에서 부모-자식을 연결하는 레퍼런스 수**



k = 2, N=4,
Non-Reference # = 4x2-(4-1)



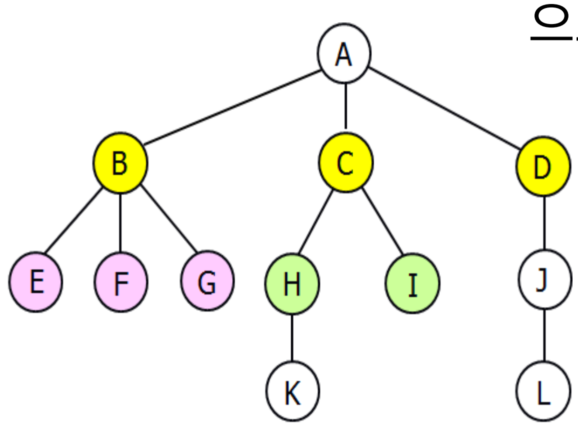
k = 3, N=16

왼쪽자식-오른쪽형제
(Left Child-Right Sibling, LCRS) 표현

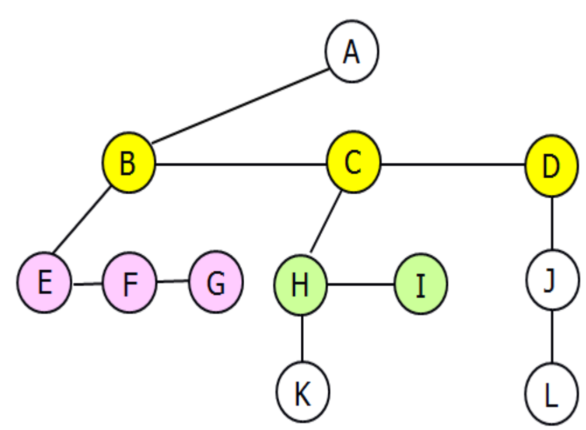
- 노드의 왼쪽 자식과 왼쪽 자식의 오른쪽 형제를 가리키는 2개의 레퍼런스만을 사용

key	
왼쪽 자식	오른쪽 형제

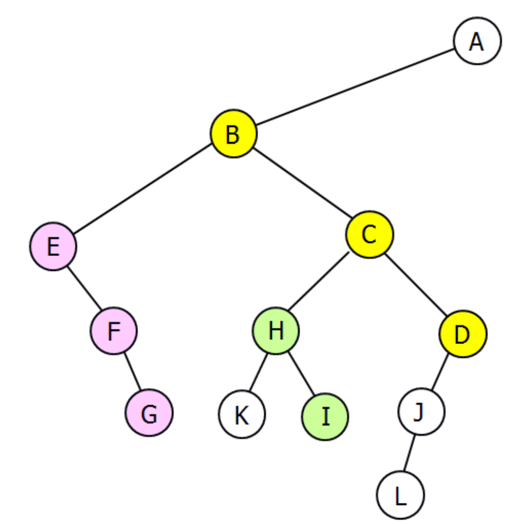
다중노드를 LCRS로



트리를 왼쪽자식-
오른쪽형제 표현
으로 변환



트리를 45° 시
계 방향으로 회
전

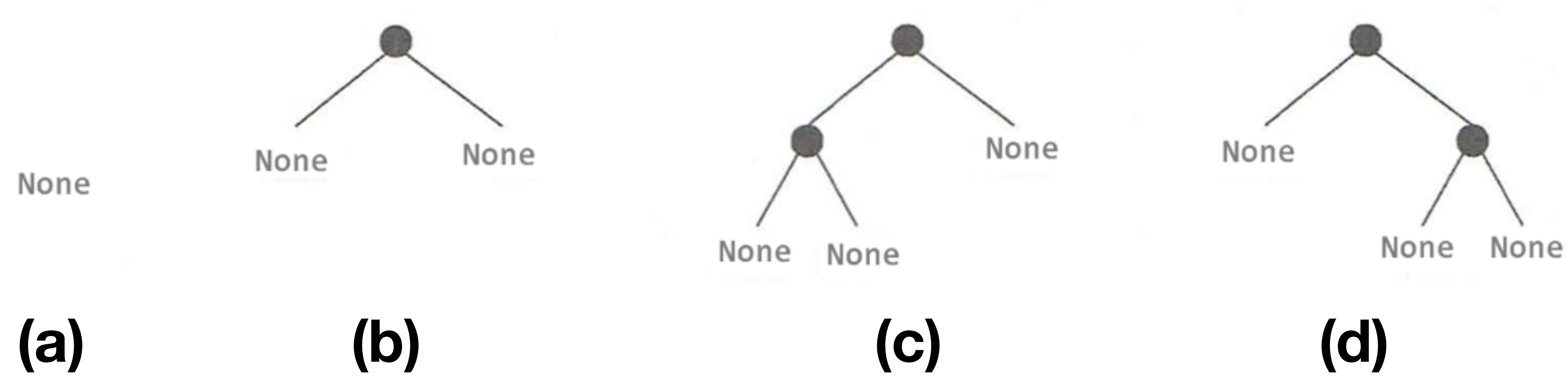


이진트리 (Binary Tree)

- 이진트리(Binary Tree): 각 노드의 자식 수가 2 이하인 트리
- 컴퓨터 분야에서 널리 활용되는 기본적인 자료구조
    - 이진트리가 데이터의 구조적인 관계를 잘 반영하고, 효율적인 삽입과 탐색을 가능하게 하며, 이진트리의 서브트리를 다른 이진트리의 서브트리와 교환하는 것이 쉽기 때문

이진트리 (Binary Tree)

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



(a) empty 트리

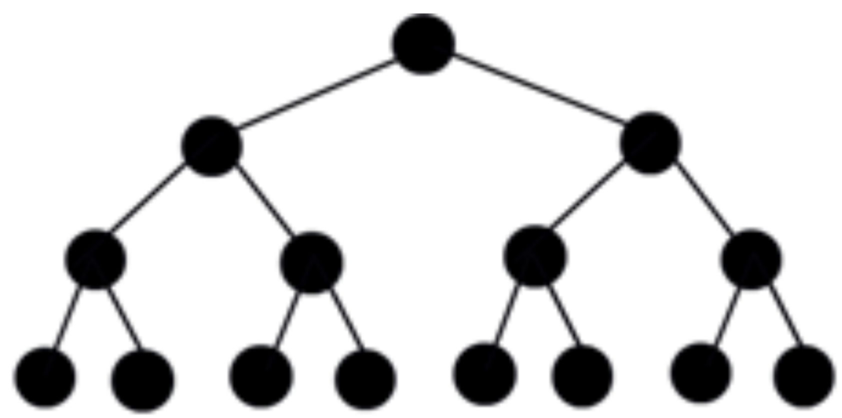
(b) 루트만 있는 이진트리

(c) 루트의 오른쪽 서브트리가 없는(empty) 이진트리

(d) 루트의 왼쪽 서브트리가 없는 이진트리

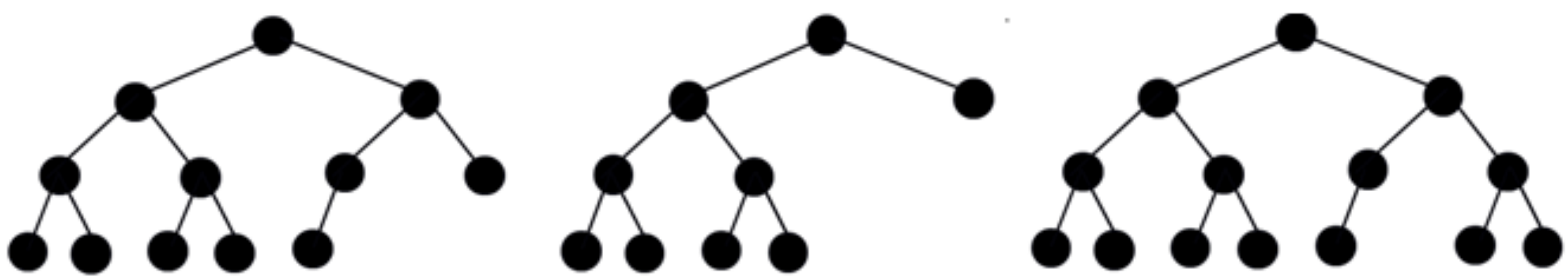
이진트리 (Binary Tree)

- 포화이진트리(Full Binary Tree): 각 내부노드가 2개의 자식노드를 가지는 트리



이진트리 (Binary Tree)

- 완전이진트리(Complete Binary Tree): 마지막 레벨을 제외한 각 레벨이 노드들로 꽉 차있고, 마지막 레벨에는 노드들이 왼쪽부터 빠짐없이 채워진 트리

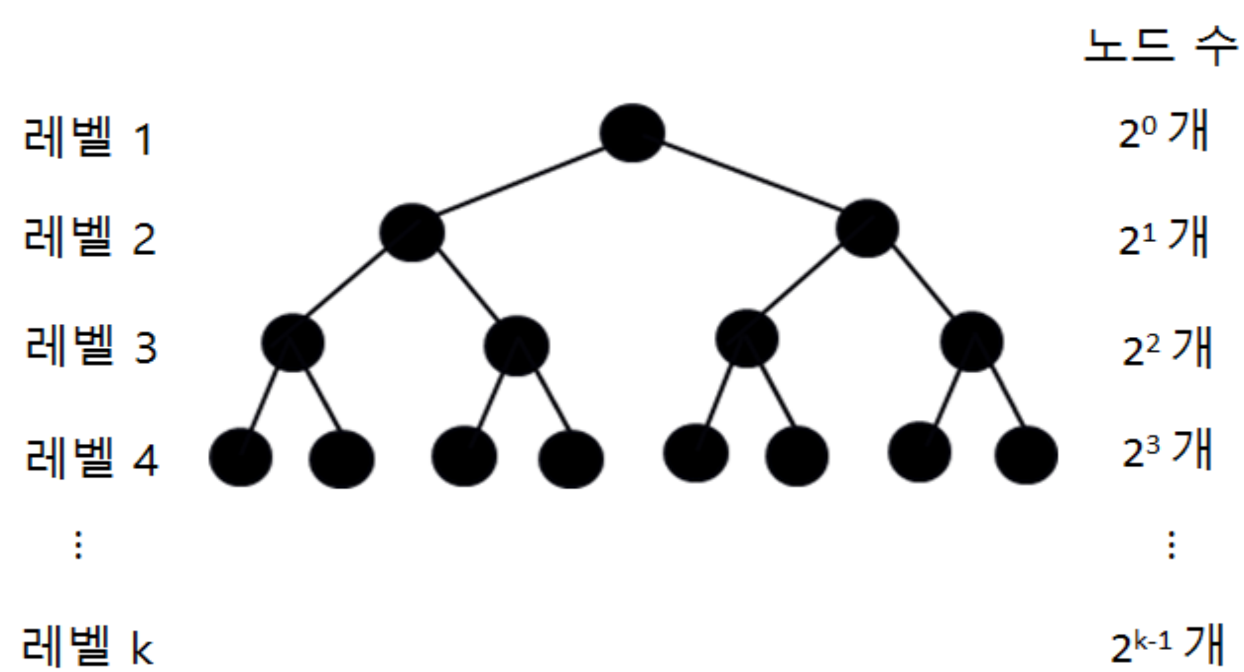


완전이진트리 불완전이진트리

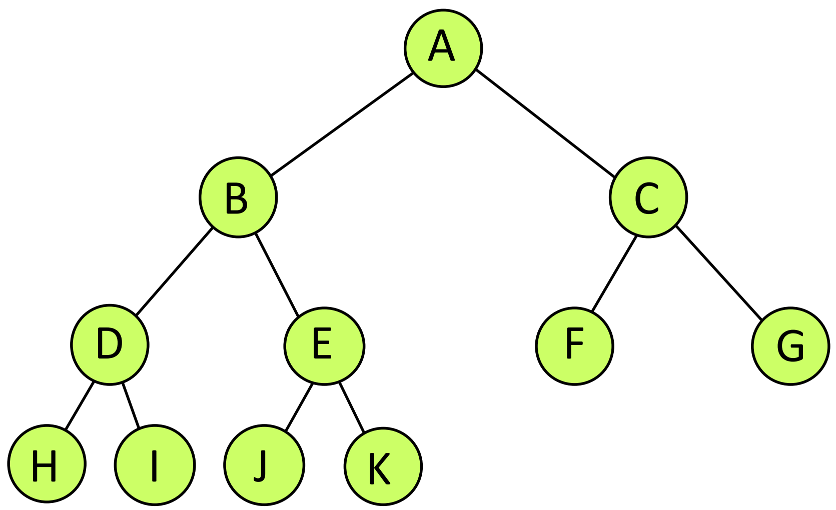
- **[정리] 포화이진트리는 완전이진트리이다.**

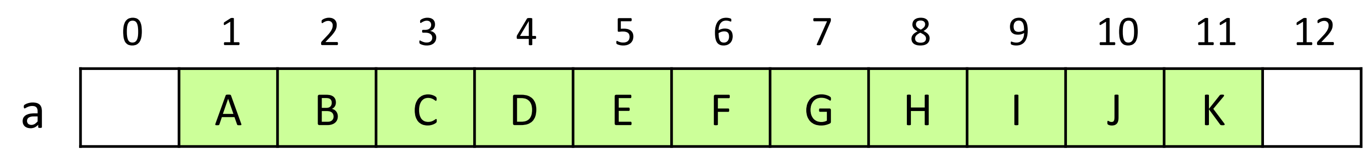
속성

- 레벨 k에 있는 최대 노드 수 = 2^{k-1} , k = 1, 2, 3, …
- 높이가 h인 포화이진트리에 있는 노드 수 = $2^h - 1$
- N개의 노드를 가진 완전이진트리의 높이 = $\lceil log_2(N+1) \rceil$



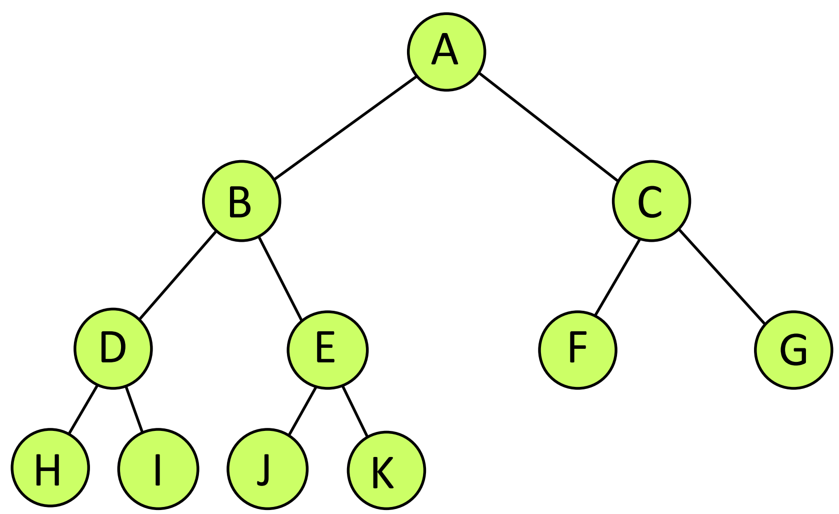
구현 (리스트, 배열)

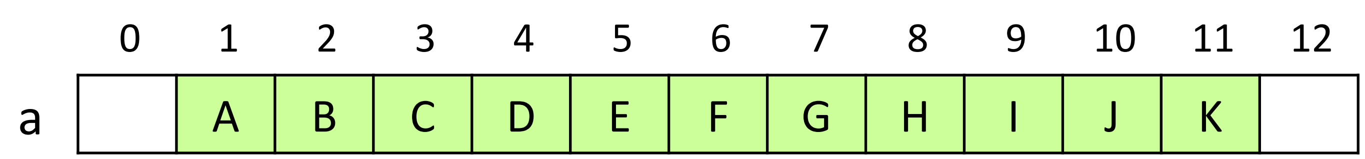




- 리스트에 저장하면 노드의 부모와 자식노드가 리스트의 어디에 저장되어 있는지를 다음과 같은 규칙을 통해 쉽게 알 수 있다.
  - 단, 트리에 N개의 노드가 있다고 가정

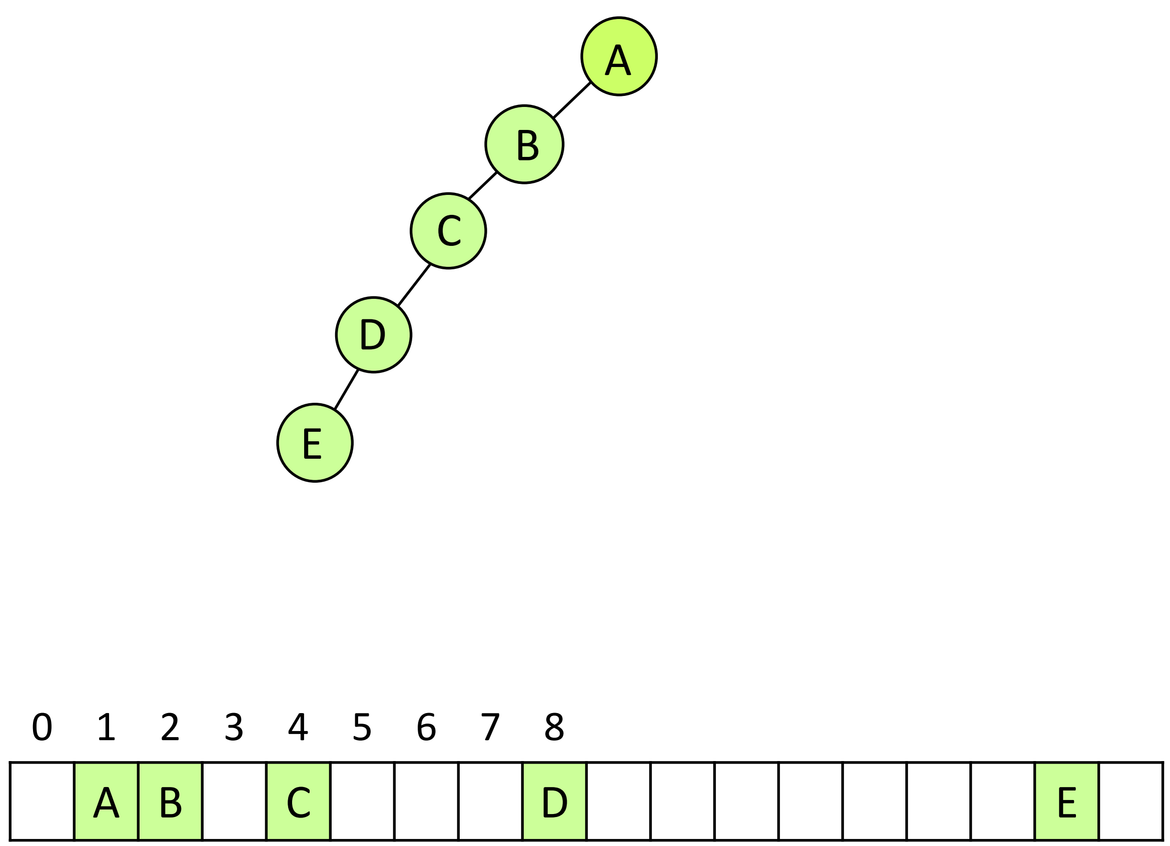
구현 (리스트, 배열)

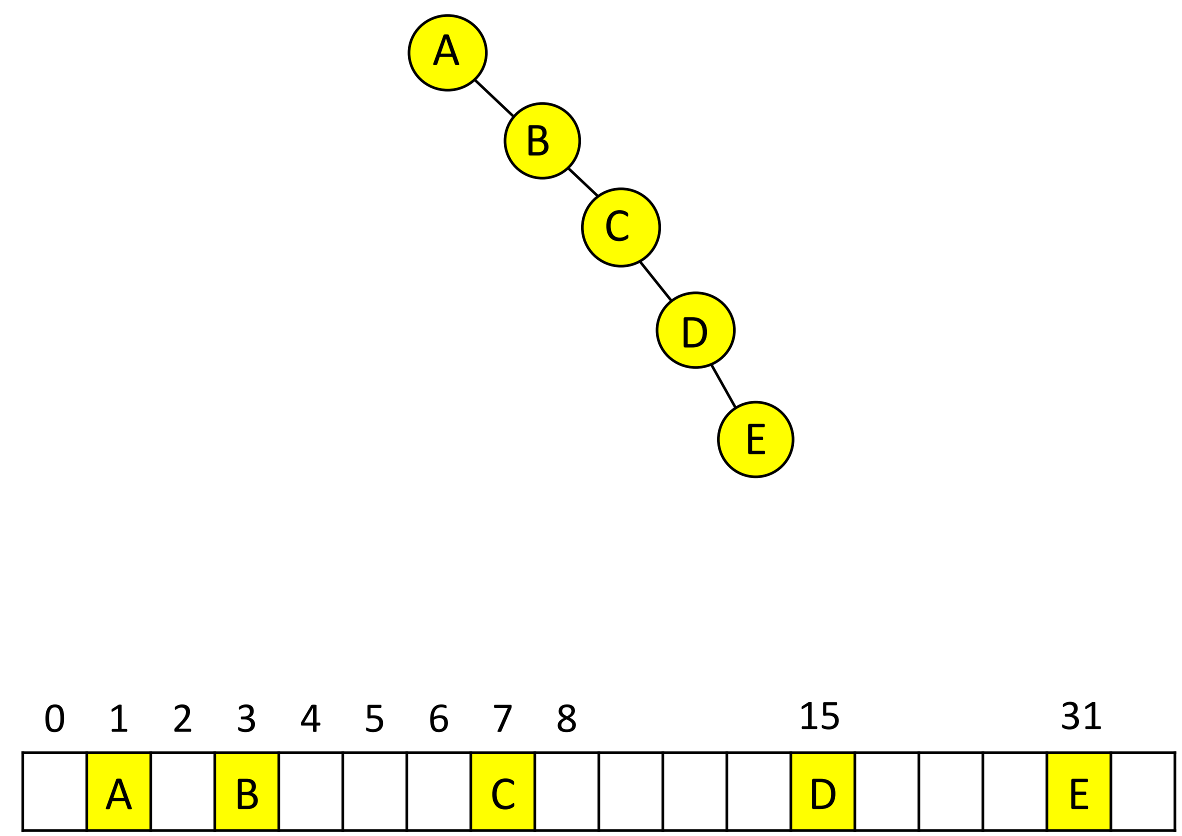




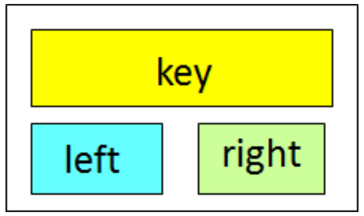
- a[i]의 부모는 a[i//2]에 있다. 단, i > 1이다. e.g.) E의 부모는 a[5//2] = a[2]
- a[i]의 왼쪽자식은 a[2i]에 있다. 단, 2i ≤ N이다. e.g.) E의 왼쪽 자식은 a[2x5] = a[10]
- a[i]의 오른쪽자식은 a[2i+1]에 있다. 단, 2i + 1 ≤ N이다. e.g.) E의 오른쪽 자식은 a[2x5+1] = a[11]

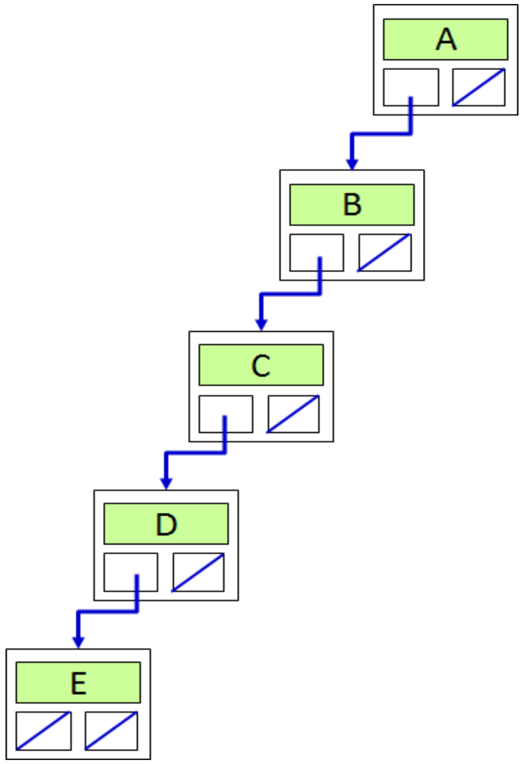
편향 이진트리 구현 (리스트, 배열)

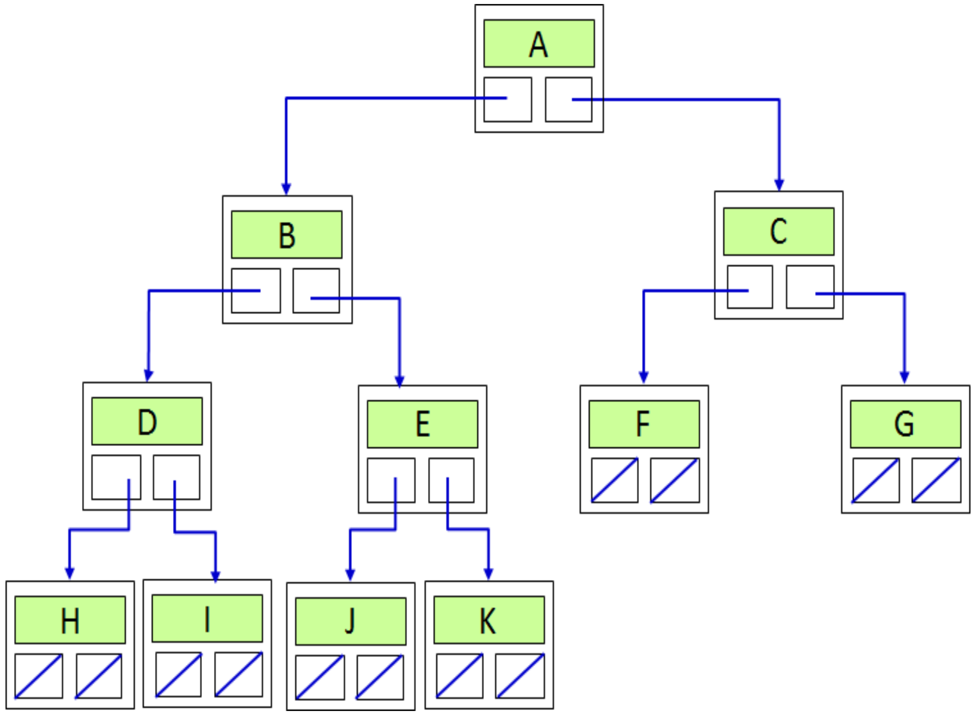




구현 (연결리스트)







트리

```python
class BinaryTree:
    class Node:
        def __init__(self, item, left=None, right=None):
            self.item  = item
            self.left  = left
            self.right = right

    def __init__(self): # 트리 생성자
        self.root = None
```

노드 생성자
항목과 왼쪽, 오른쪽 자식노드 레퍼런스

트리의 루트

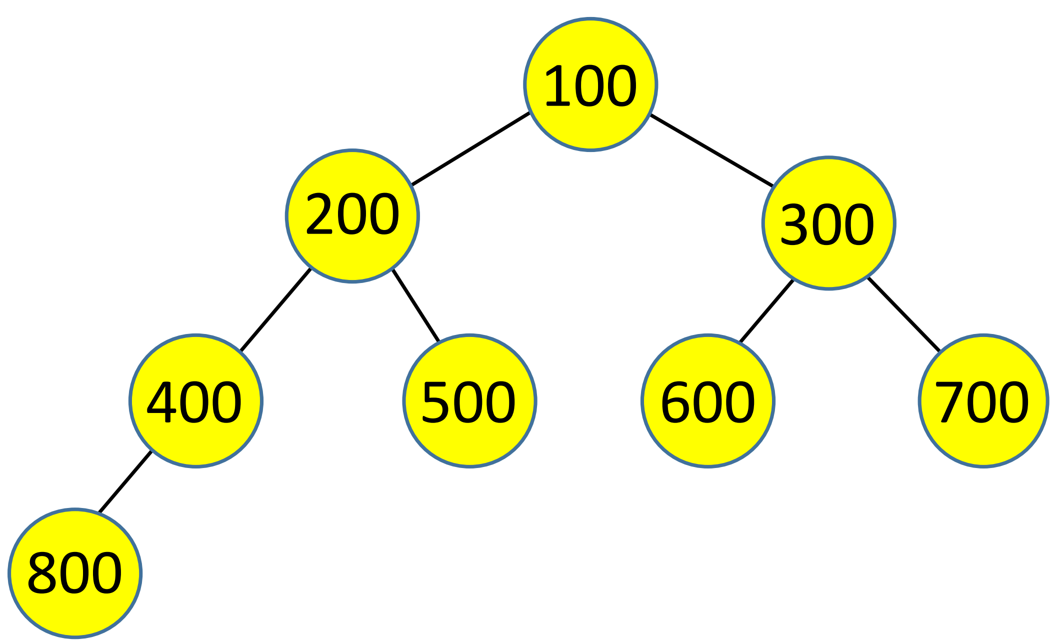
트리

```python
from binarytree import BinaryTree, Node
if __name__ == '__main__':
    t = BinaryTree()
    n1 = Node(100)
    n2 = Node(200)
    n3 = Node(300)
    n4 = Node(400)
    n5 = Node(500)
    n6 = Node(600)
    n7 = Node(700)
    n8 = Node(800)
    n1.left  = n2
    n1.right = n3
    n2.left  = n4
    n2.right = n5
    n3.left  = n6
    n3.right = n7
    n4.left  = n8
    t.root = n1
```

이진트리 객체 생성

8개의 노드 생성

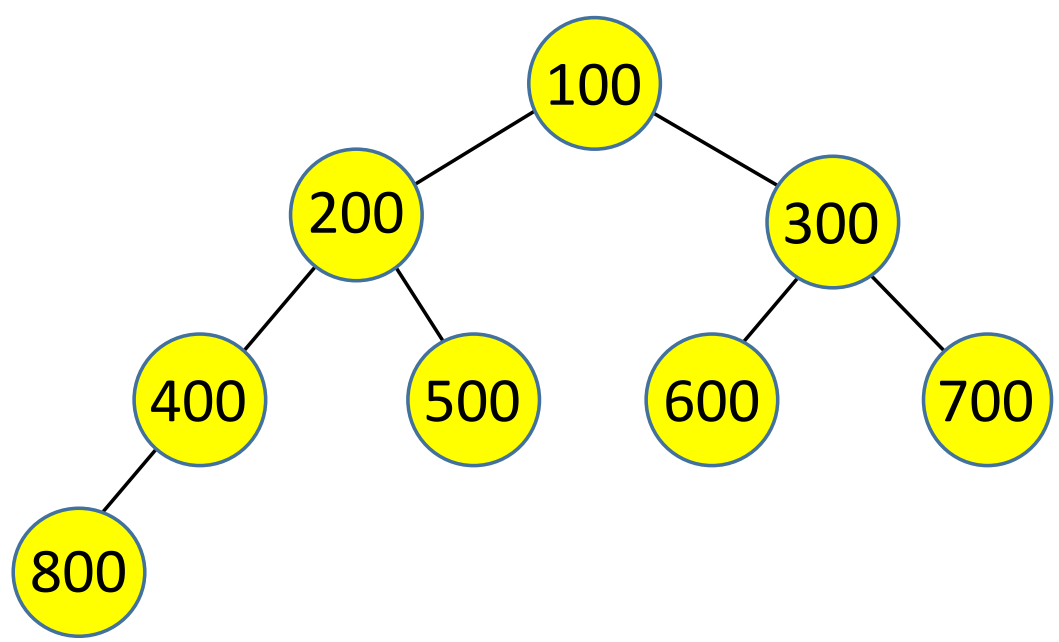
트리 만들기



트리 연산: 높이 계산

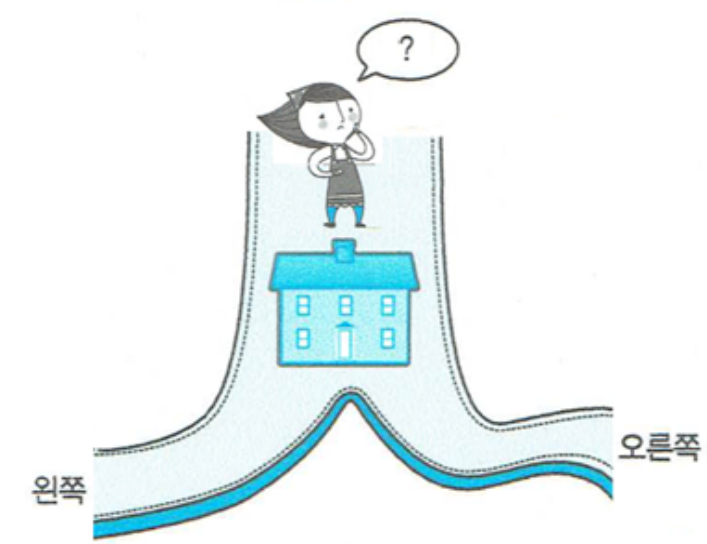
```python
def height(self, root): # 트리 높이 계산
    if root == None:
        return 0
    return max(self.height(root.left), self.height(root.right))+1
```

두 자식노드의 높이 중 큰 높이 + 1



트리연산: 순회

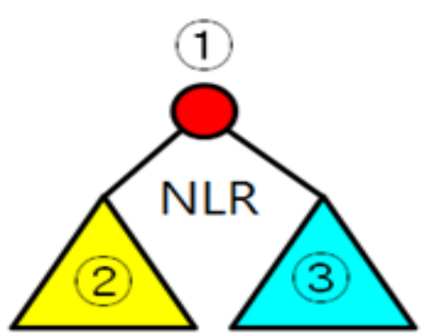
- 전위순회(Preorder Traversal)
- 중위순회(Inorder Traversal)
- 후위순회(Postorder Traversal)
- 레벨순회(Levelorder Traversal)



전위, 중위, 후위순회는 모두 루트노드로부터 동일한 순서로 이진트리의 노드들을 지나가는데, 특정 노드에 도착하자마자 그 노드를 방문하는지, 일단 지나치고 나중에 방문하는지에 따라 구분됨

전위순회 (NLR)

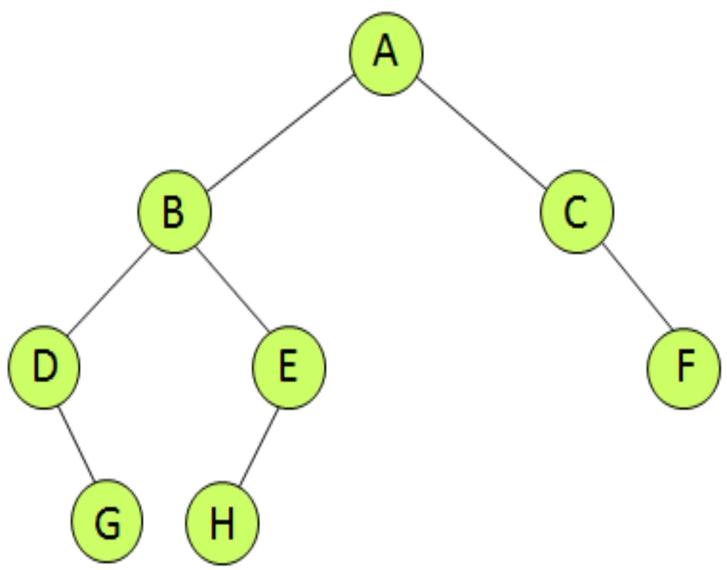
- 전위순회는 노드 n에 도착했을 때 n을 먼저 방문
- 그 다음에 n의 왼쪽 자식노드로 순회를 계속
- n의 왼쪽 서브트리의 모든 노드들을 방문한 후에는 n의 오른쪽 서브트리의 모든 후손 노드들을 방문
- 전위순회의 방문 규칙

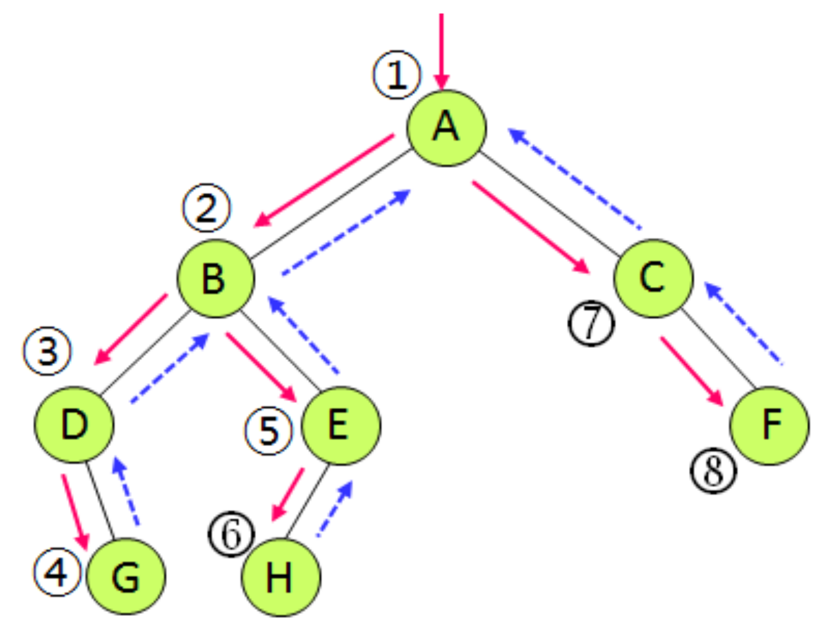



```python
def preorder(self, n): # 전위순회
    if n != None:
        print(str(n.item),' ', end='')
        if n.left:
            self.preorder(n.left)
        if n.right:
            self.preorder(n.right)
```

- 전위순회 순서를 NLR 으로 표현
- 여기서 N은 노드(Node)를 방문한다는 뜻이고, V는 Visit(방문)을 의미
- L은 왼쪽, R은 오른쪽 서브트리로 순회를 진행한다는 뜻

전위순회



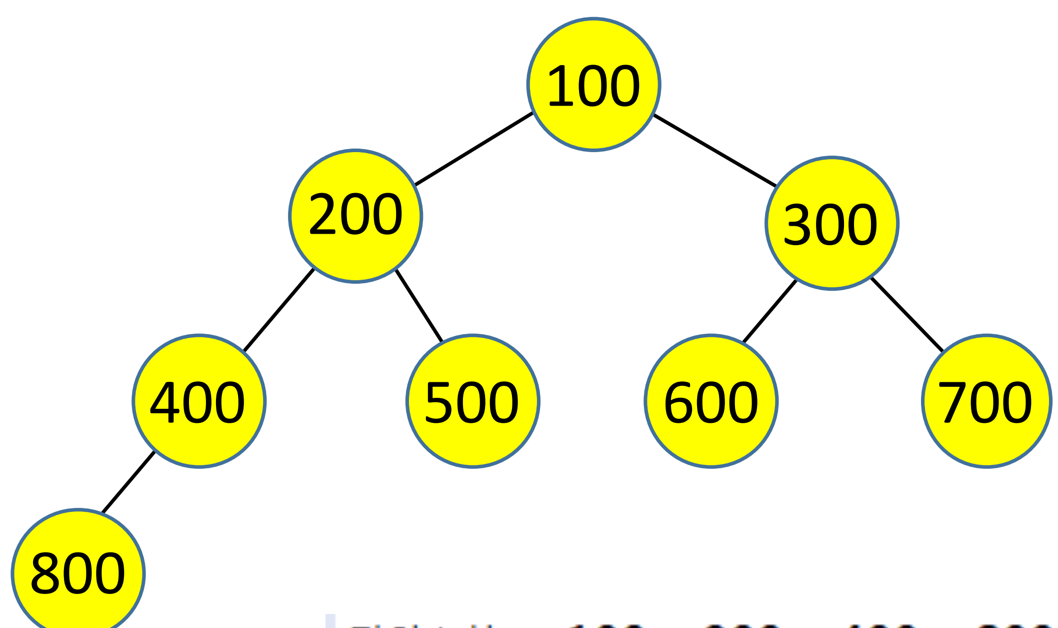


트리 연산: 전위순회

```
def preorder(self, n): # 전위순회
    if n != None:
        print(str(n.item),' ', end='')
        if n.left:
            self.preorder(n.left)
        if n.right:
            self.preorder(n.right)
```

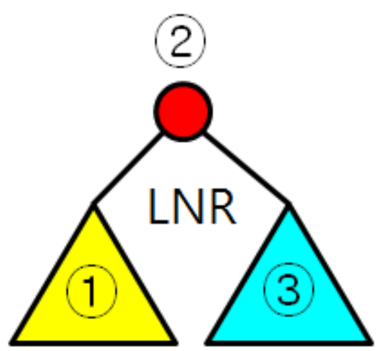
맨 먼저 노드 방문

왼쪽 서브트리 방문 후
오른쪽 서브트리 방문

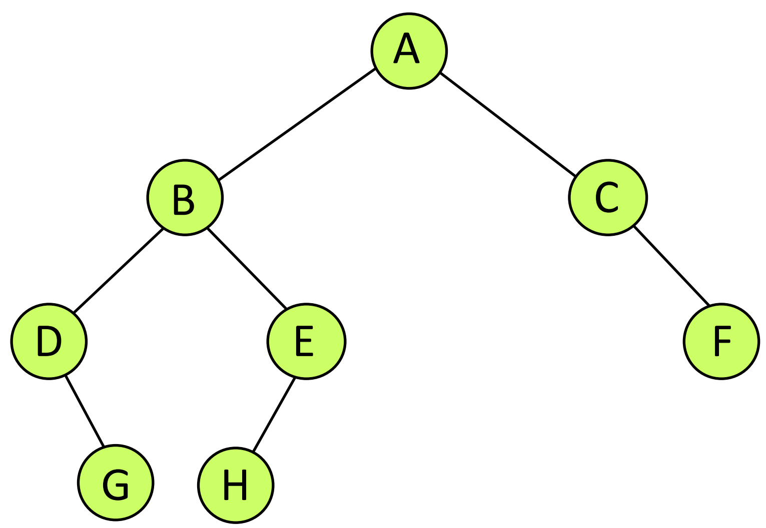


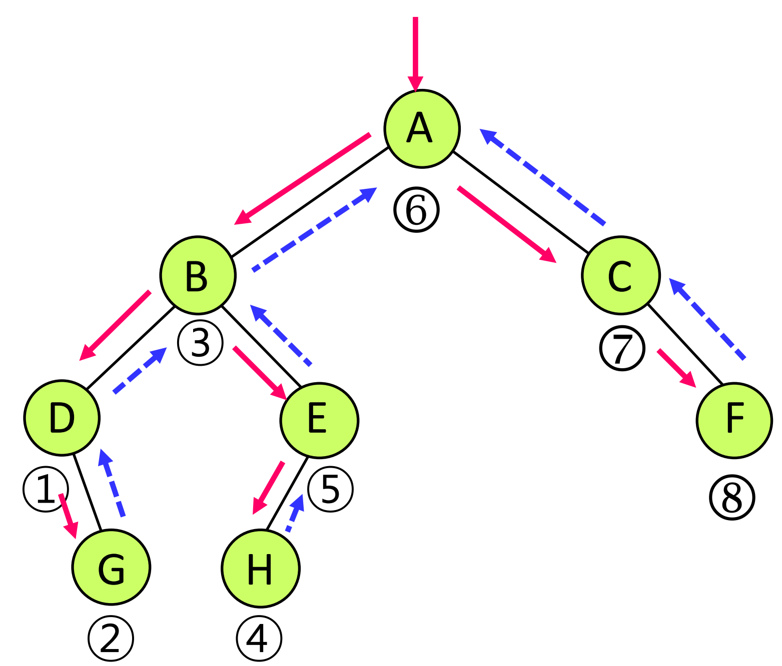
전위순회: 100 200 400 800 500 300 600 700

중위순회 (LNR)




```python
def inorder(self, n): # 중위순회
    if n != None:
        if n.left:
            self.inorder(n.left)
        print(str(n.item),' ', end='')
        if n.right:
            self.inorder(n.right)
```

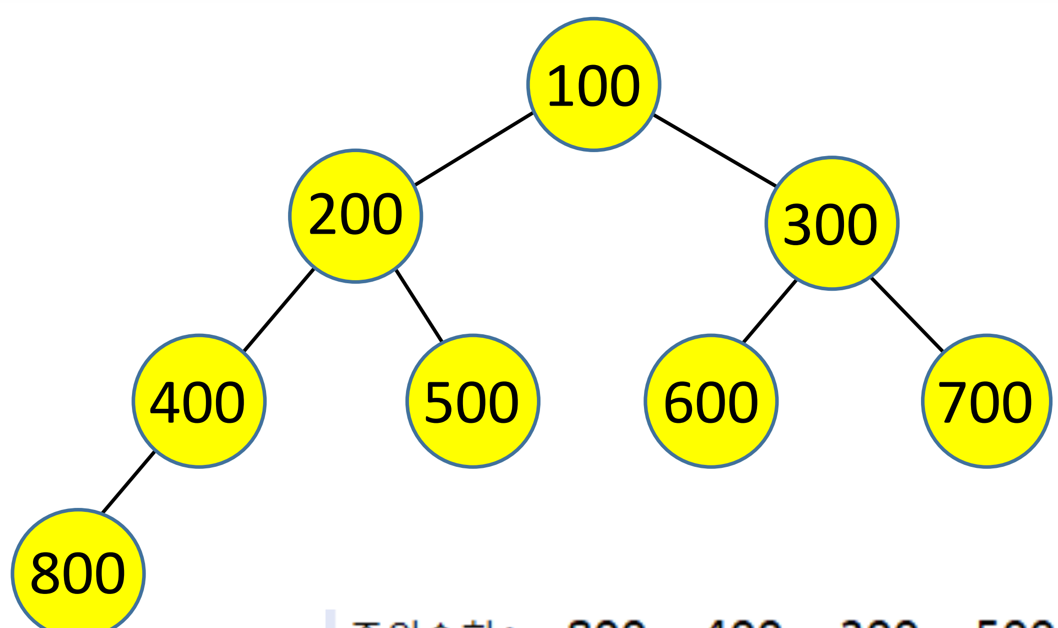




트리 연산: 중위순회

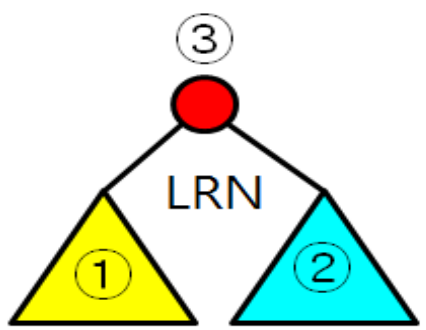
```python
    def inorder(self, n): # 중위순회
        if n != None:
            if n.left:
                self.inorder(n.left)
            print(str(n.item),' ', end='')
            if n.right:
                self.inorder(n.right)
```

왼쪽 서브트리 방문 후 노드 방문

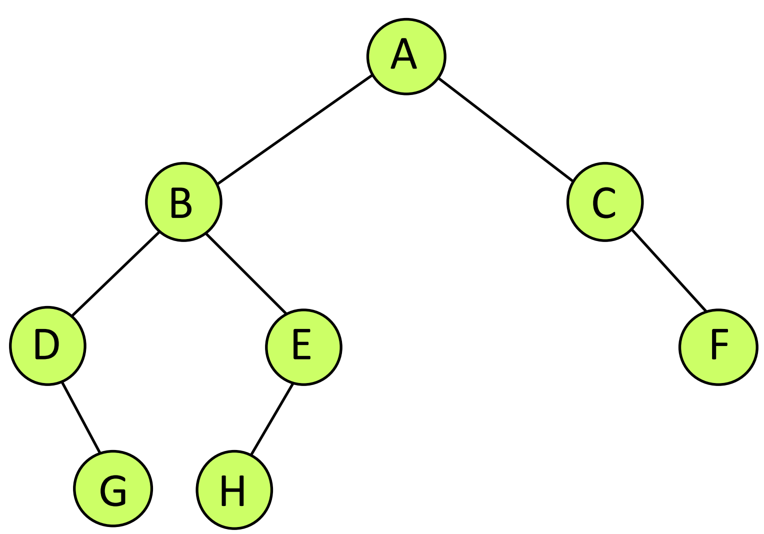


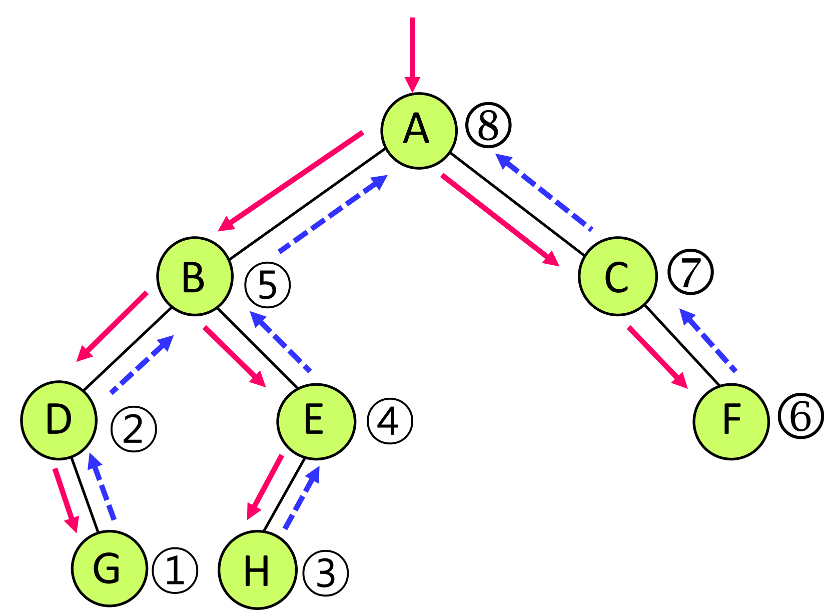

```
중위순회:  800  400  200  500  100  600  300  700
```

중위순회




```python
def postorder(self, n): # 후위순회
    if n != None:
        if n.left:
            self.postorder(n.left)
        if n.right:
            self.postorder(n.right)
        print(str(n.item),' ', end='')
```

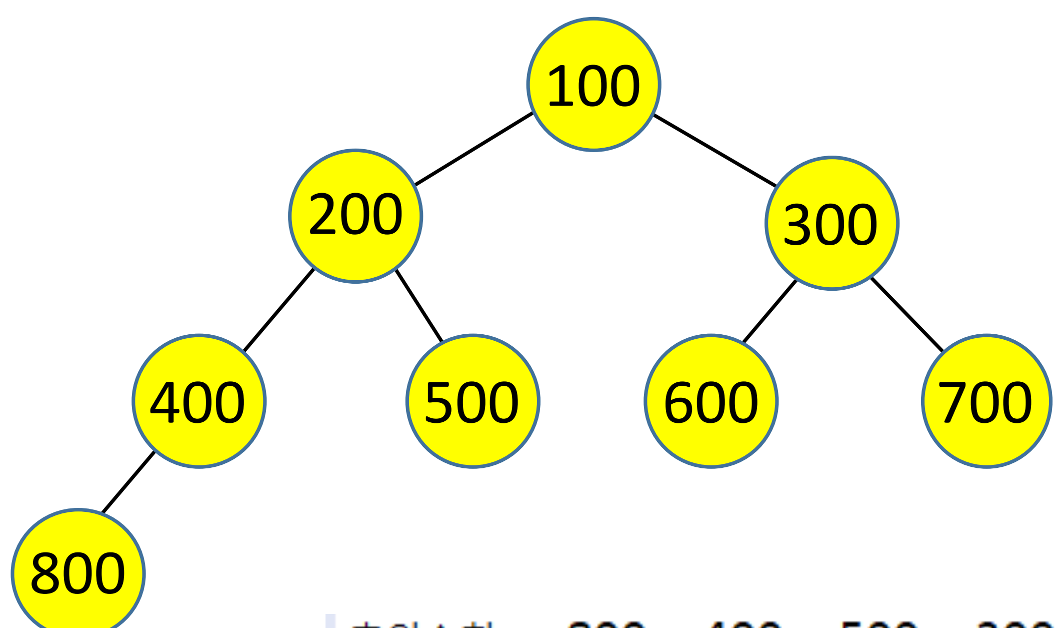




트리 연산: 후위순회

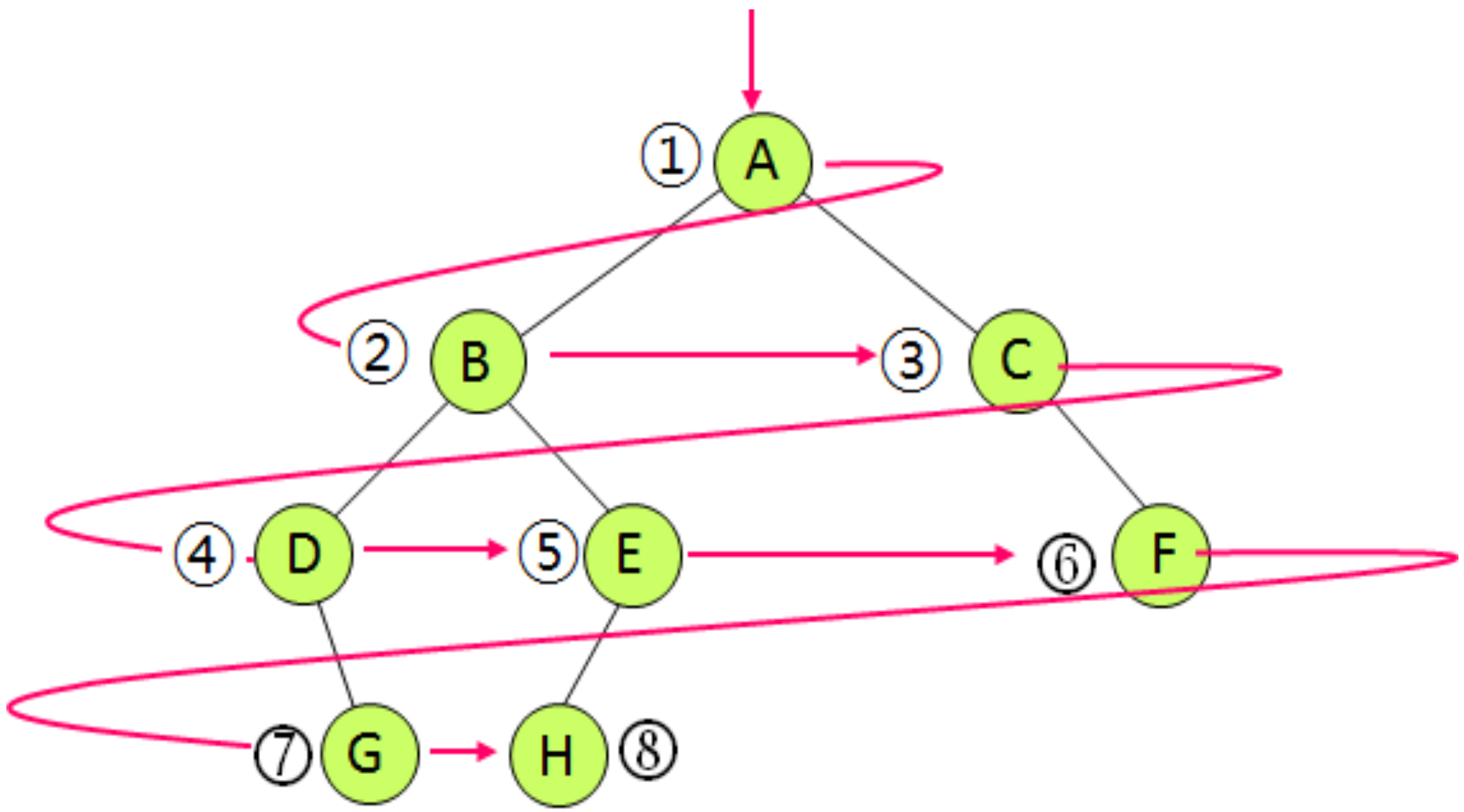
```python
    def postorder(self, n): # 후위순회
        if n != None:
            if n.left:
                self.postorder(n.left)
            if n.right:
                self.postorder(n.right)
            print(str(n.item),' ', end='')
```

왼쪽과 오른쪽 서브트리
모두 방문 후 노드 방문



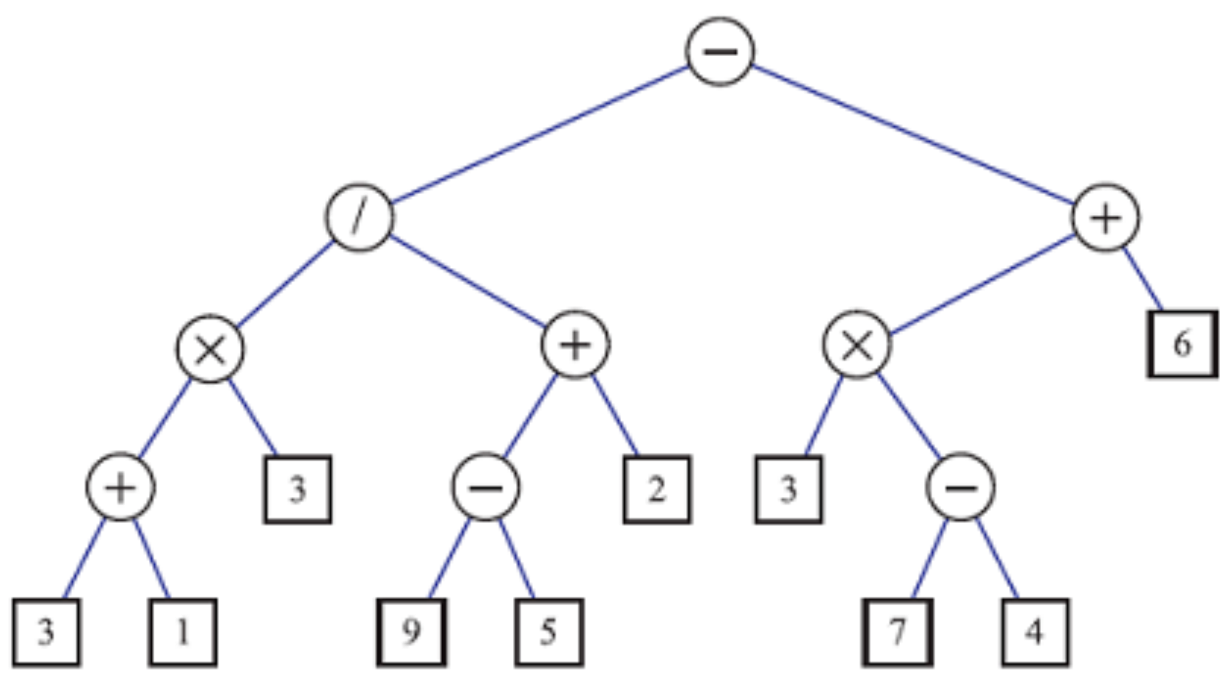

```
후위순회:  800  400  500  200  600  700  300  100
```

트리연산: 레벨순회



Exercise

- 수식을 표현한 다음 트리를 전위, 중위, 후위순회를 통해 전위, 중위, 후위표기법으로 나타내어 보아라.



트리 연산: 레벨순회

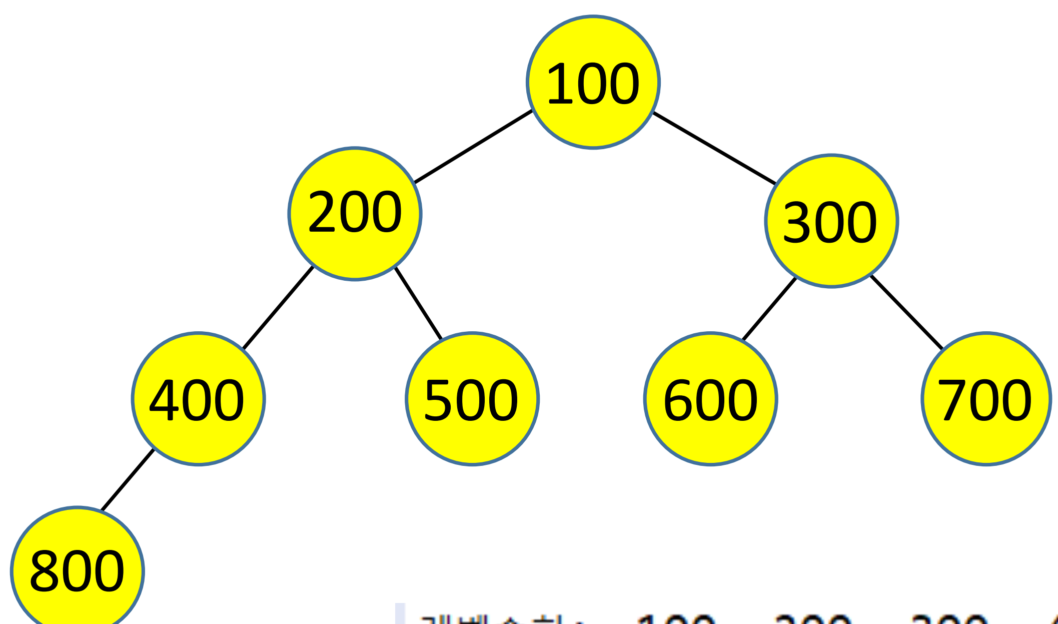
```python
def levelorder(self, root): # 레벨순회
    q = []
    q.append(root)
    while len(q) != 0:
        t = q.pop(0)
        print(str(t.item), ' ', end='')
        if t.left != None:
            q.append(t.left)
        if t.right != None:
            q.append(t.right)
```

리스트로 큐 자료구조 구현

큐에서 첫 항목 삭제

삭제된 노드 방문

왼쪽자식, 오른쪽자식
큐에 삽입




```
레벨순회:  100  200  300  400  500  600  700  800
```

트리연산: 총 노드수

- **트리의 노드 수**를 계산하는 것은 **트리의 아래에서 위로 각 자식의 후손노드 수를 합하며 올라가는 과정을 통해 수행되며,** 최종적으로 루트노드에서 총 합을 구함 (**후위순회**)

- **트리의 높이**도 아래에서 위로 두 자식을 각각 루트노드로 하는 서브트리의 높이를 비교하여 보다 큰 높이에 1을 더하는 것으로 자신의 높이를 계산하며, 최종적으로 루트노드의 높이가 트리의 높이가 됨 (후위순회방법)

- 2개의 이진트리를 비교하는 것은 다른 부분을 발견하는 즉시 비교 연산을 멈추기 위해 **전위순회 방법**을 사용

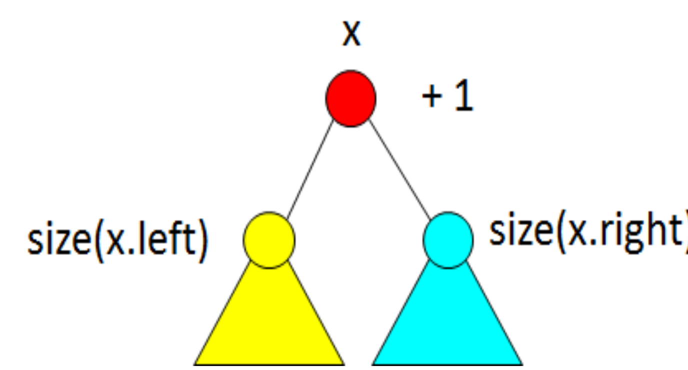
총 노드수 계산

- 트리의 노드 수 = 1 +

  (루트노드의 왼쪽 서브트리에 있는 노드 수) +

  (루트노드의 오른쪽 서브트리에 있는 노드 수)

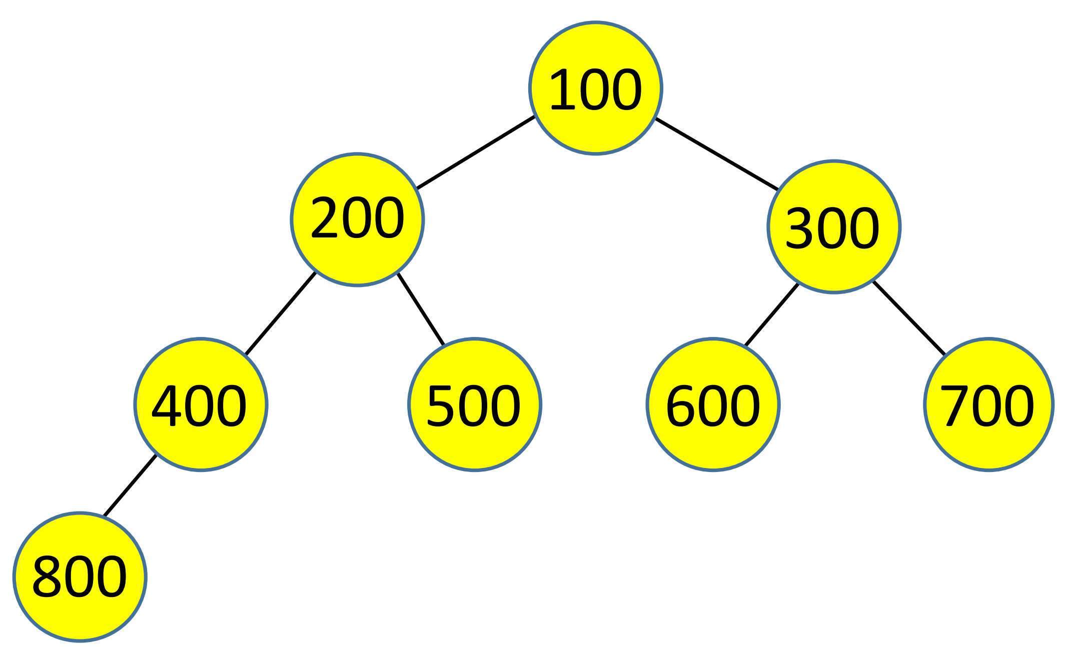
- 1은 루트노드 자신을 계산에 반영하는 것




```python
def size(self, root): # 트리 노드 수 계산
    if root is None:
        return 0
    else:
        return 1 + self.size(root.left) + self.size(root.right)
```

총노드수 계산

```python
def size(self, root): # 트리 노드 수 계산
    if root is None:
        return 0
    else:
        return 1 + self.size(root.left) + self.size(root.right)
```



수행시간

- 앞서 설명된 각 연산은 트리의 각 노드를 한 번씩만 방문하므로 O(N) 시간이 소요

기타 이진트리 연산

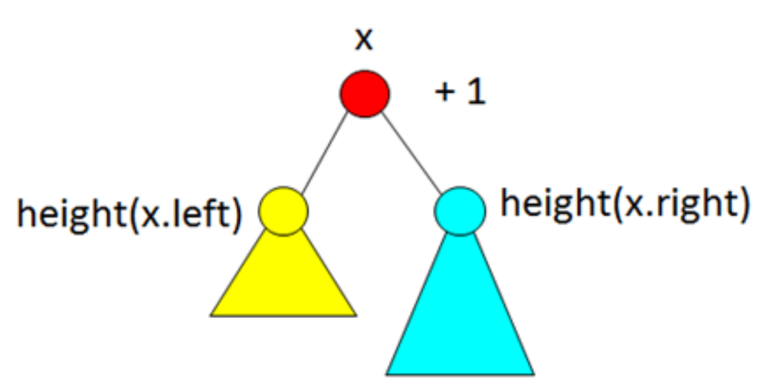
- 이진트리의 높이 계산
- isEqual(): 2개의 이진트리에 대한 동일성 검사

이진트리의 높이

- 트리의 높이

  = 1 + max (루트의 왼쪽 서브트리의 높이, 루트의 오른쪽 서브트리의 높이)

  - 1은 루트노드 자신을 계산에 반영
  - 왼쪽과 오른쪽 서브트리의 높이는 같은 재귀적 방식으로 계산



이진트리의 높이

- 트리의 높이

  = 1 + max (루트의 왼쪽 서브트리의 높이, 루트의 오른쪽 서브트리의 높이)

  - 1은 루트노드 자신을 계산에 반영
  - 왼쪽과 오른쪽 서브트리의 높이는 같은 재귀적 방식으로 계산

```
def height(self, root): # 트리 높이 계산
    if root == None:
        return 0
    return max(self.height(root.left), self.height(root.right))+1
```

Exercise

- 두 이진 트리가 주어지면, 두 트리가 동일한 트리인지 그 결과를 도출하라.
- [핵심 아이디어] 전위순회 과정에서 다른 점이 발견되는 순간 False를 리턴
    - is_equal(): 비교하려는 두 트리의 루트노드들을 인자로 전달하여 호출
    - 노드 n과 m 둘 중에 하나가 None인 경우
        - 만일 둘 다 None이면 True를 리턴하고
        - 한 쪽만 None이면 트리가 다른 것이므로 False를 리턴

쾌매진 이진트리
(Threaded Binary Tree)

- 이진트리의 기본 연산들은 레벨순회를 제외하고 모두 스택 자료구조를 사용: 메소드의 재귀호출은 시스템 스택을 사용하므로 스택 자료구조를 사용한 것으로 간주

- 스택에 사용되는 메모리 공간의 크기는 트리의 높이에 비례

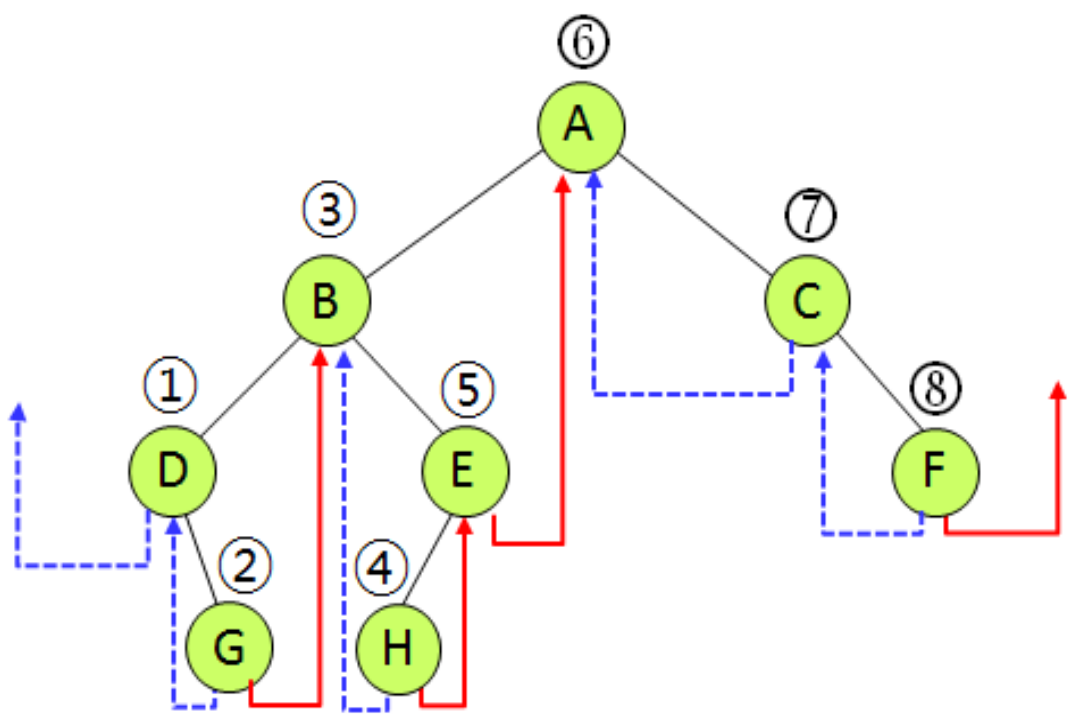
- 스택 없이 이진트리의 연산을 구현하는 2 가지 방법

1. Node 객체에 부모를 가리키는 레퍼런스를 추가로 선언하여 순회에 사용하는 방법

2. 노드의 None 레퍼런스들을 활용하는 것 (스레드 이진트리 (Threaded Binary Tree)

꿰매진 이진트리 (Threaded Binary Tree)

- **[정의]** 꿰매진 (쓰레드) 이진트리는 각 노드의 **오른쪽 non-레퍼런스를 다음 방문할 노드를 참조**하게 하고, 각 노드의 **왼쪽 non-레퍼런스를 직전 방문한 노드를 참조하게** 한 이진트리이다.

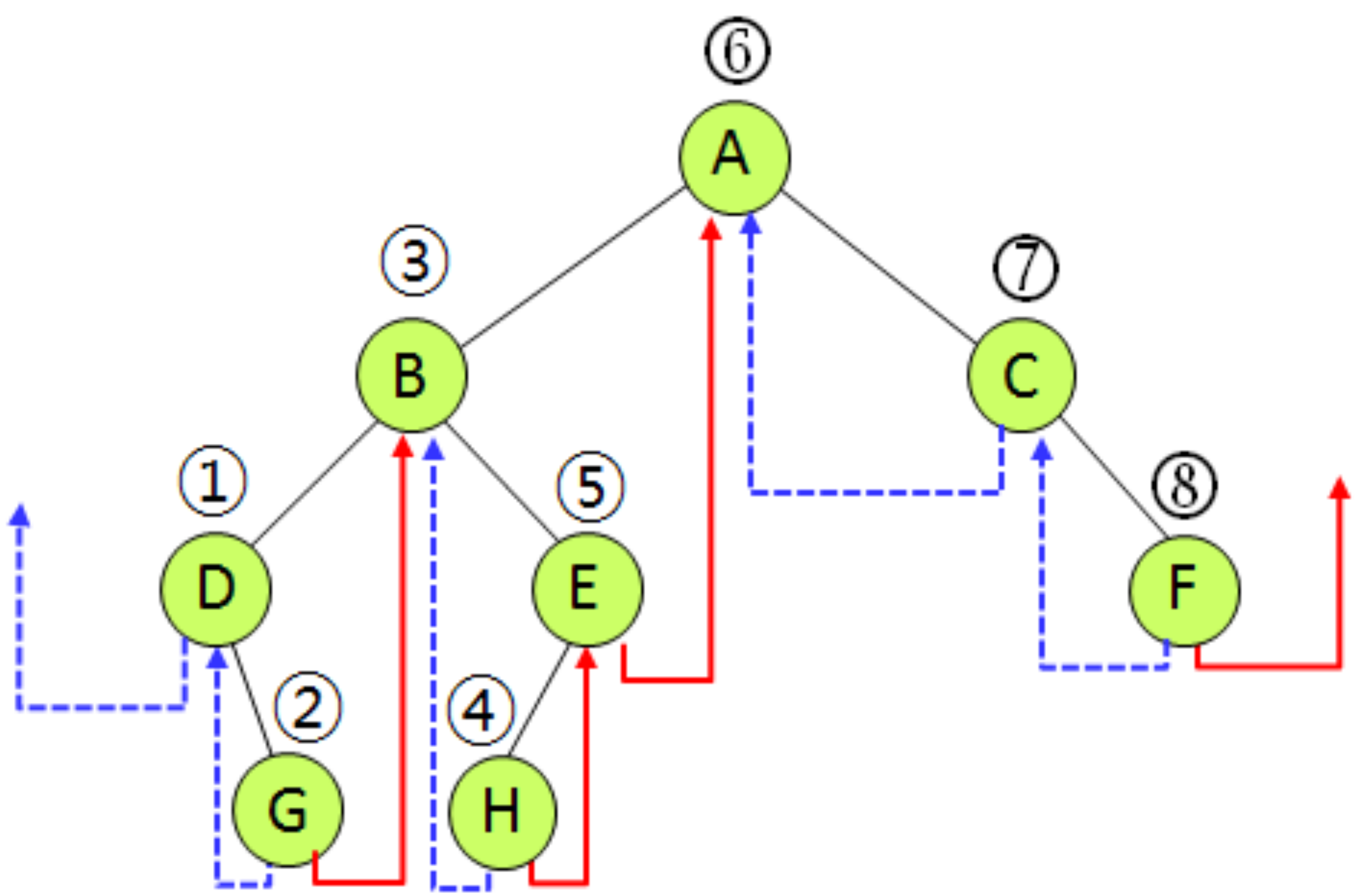


이진트리의 특성

- N개의 노드가 있는 이진트리의 **None 레퍼런스 필드** 수 = (N+1)
- 왜냐하면 각 노드마다 2개의 레퍼런스(left와 right)가 있으므로 총 2N개의 레퍼런스가 존재하고, 이 중에서 부모 자식을 연결하는 레퍼런스는 N-1개이기 때문
- 부모 자식을 연결하는 레퍼런스가 N-1개인 이유는 루트노드를 제외한 각 노드가 1개의 부모를 갖기 때문
- 따라서 (N+1) 개의 None 레퍼런스 필드의 남는 정보를 이용하여 스택을 대신할 수 있는 정보를 활용하게 함.

꿰매진 이진트리
(Threaded Binary Tree)

- 중위기반 꿰매진 이진트리



꿰매진 이진트리 (Threaded Binary Tree)

- 스레드 이진트리는 대부분의 경우 중위순회에 기반하여 구현되나, 전위순회이나 후위순회에 기반하여 스레드 트리를 구현할 수도 있음

- 스레드 이진트리는 스택을 사용하는 순회보다 빠르고 메모리 공간도 적게 차지한다는 장점을 갖지만 데이터의 삽입과 삭제가 잦은 경우 그 구현이 비교적 복잡한 편이므로 좋은 성능을 보여주지 못한다는 문제점

- Node 객체에 2개의 boolean 필드를 사용하여 레퍼런스가 스레드(다음 방문할 노드를 가리키는)로 사용되는 것인지 아니면 left나 right가 트리의 부모 자식 사이의 레퍼런스인지를 각각 True 와 False로 표시해주어야 함

이진힙 Binary Heap

- 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

우선순위큐

- 스택에 삽입되는 항목의 우선순위는 스택에 있는 모든 항목들의 우선순위보다 높음

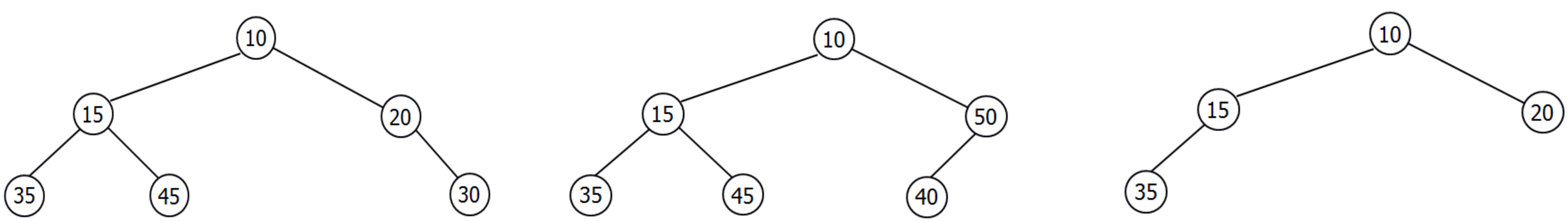
- 큐에 삽입되는 항목의 우선순위는 큐에 있는 모든 항목들의 우선수위보다 낮음

- 삽입되는 항목이 임의의 우선순위를 가지면 스택이나 큐는 새 항목이 삽입될 때마다 저장되어 있는 항목들을 우선순위에 따라 정렬해야 하는 문제점이 있음

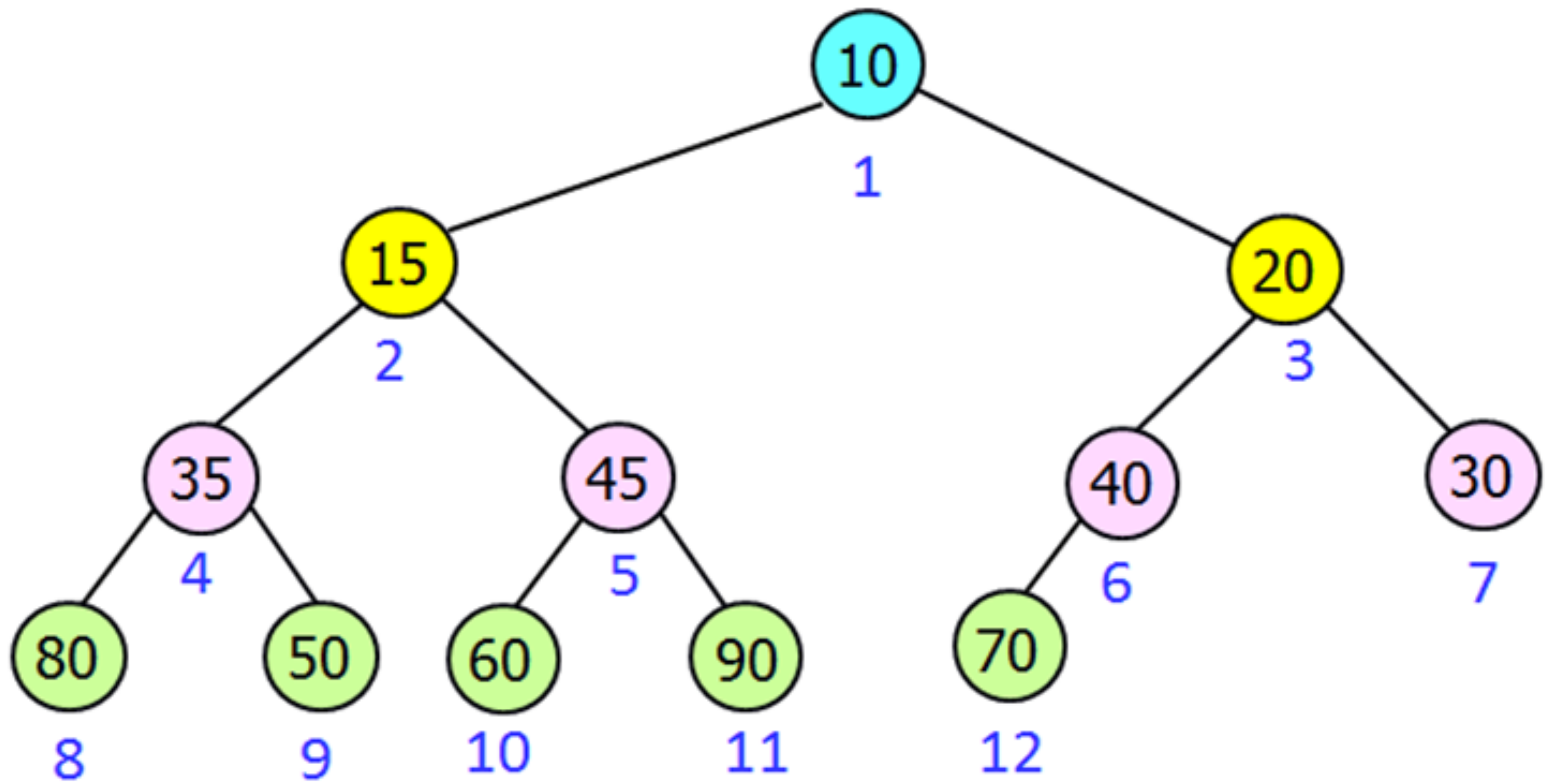
이진힙

- [정의] 이진힙(Binary Heap, 혹은 힙)은 완전이진트리로서 부모의 우선순위가 자식의 우선순위보다 높은 자료구조

어느 트리가 이진힙일까?



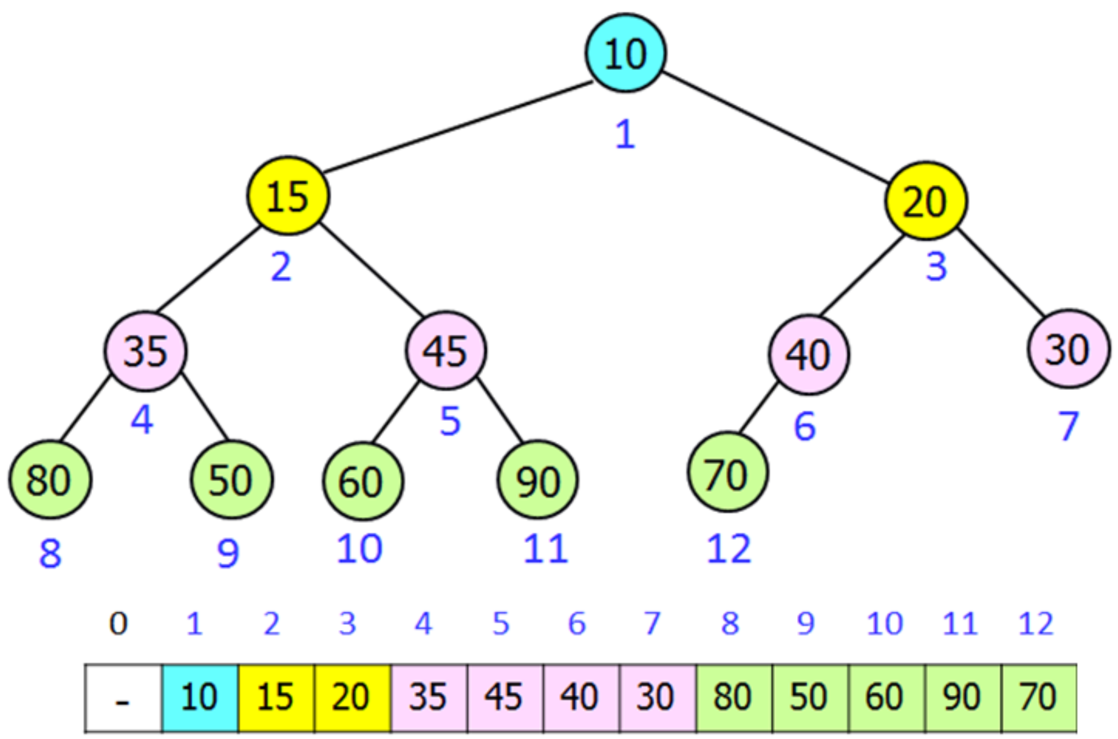
이진힙의 구현



0	1	2	3	4	5	6	7	8	9	10	11	12
-	10	15	20	35	45	40	30	80	50	60	90	70

힙

- 정리
  - a[i]의 자식은 a[2i]와 a[2i+1]에 있고,
  - a[j]의 부모는 a[j//2]에 있다, j > 1.



힙

- 이진힙의 종류:
  - 최소힙(Minimum Heap): 키 값이 작을수록 높은 우선순위
  - 최대힙(Maximum Heap): 키 값이 클수록 더 높은 우선순위
- 최소힙의 루트에는 항상 가장 작은 키가 저장됨
  - 부모에 저장된 키가 자식의 키보다 작다는 규칙 때문
  - 루트는 a[1]에 있으므로, O(1) 시간에 min 키를 가진 노드 접근

힙 연산들

```python
from binaryheap import BHeap
if __name__ == '__main__':
    a = [None] * 1
    a.append([90, 'watermelon'])
    a.append([80, 'pear'])
    a.append([70, 'melon'])
    a.append([50, 'lime'])
    a.append([60, 'mango'])
    a.append([20, 'cherry'])
    a.append([30, 'grape'])
    a.append([35, 'orange'])
    a.append([10, 'apricot'])
    a.append([15, 'banana'])
    a.append([45, 'lemon'])
    a.append([40, 'kiwi'])
    b = BHeap(a)
    print('힙 만들기 전:')
    b.print_heap()
    b.create_heap()
    print('최소힙:')
    b.print_heap()
    print('최솟값 삭제 후')
    print(b.delete_min())
    b.print_heap()
    b.insert([5,'apple'])
    print('5 삽입 후')
    b.print_heap()
```

12개 항목의 리스트 생성

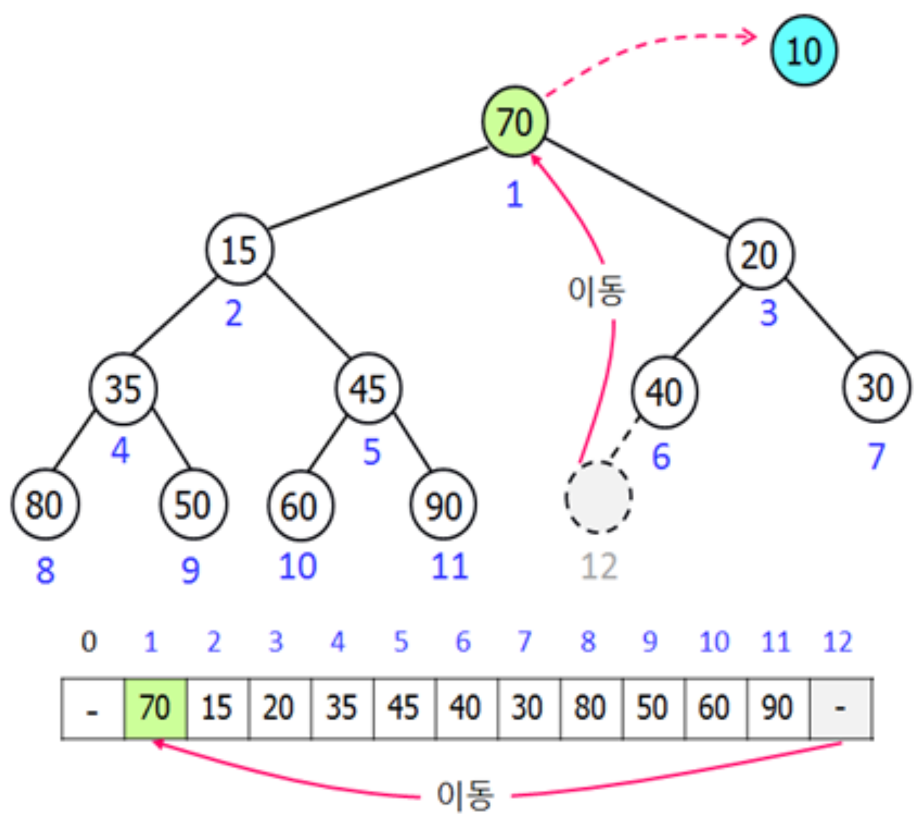
힙 객체 생성

힙 만들기, 삭제, 삽입 연산

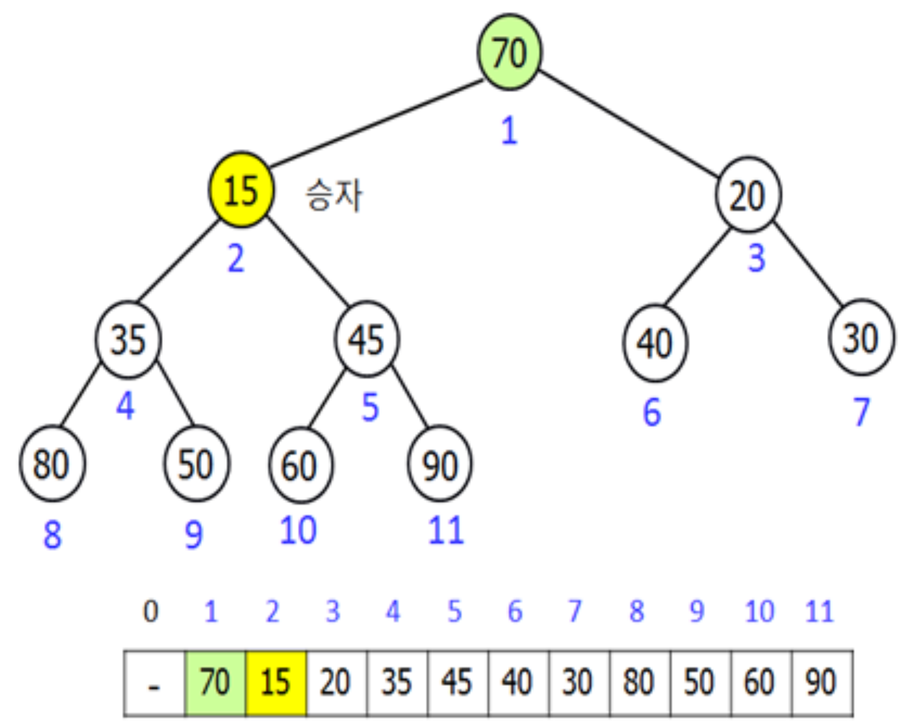
최소값 삭제 delete_min()

- [1] 힙의 가장 마지막 노드, 즉, 리스트의 가장 마지막 항목을 루트로 옮기고,
- [2] 힙 크기를 1 감소시킨다.
- [3] 루트로부터 자식들 중에서 작은 값을 가진 자식 (승자)과 키를 비교하여 힙속성이 만족될 때까지 키를 교환하며 이파리 방향으로 진행 (downheap())

최소값 삭제

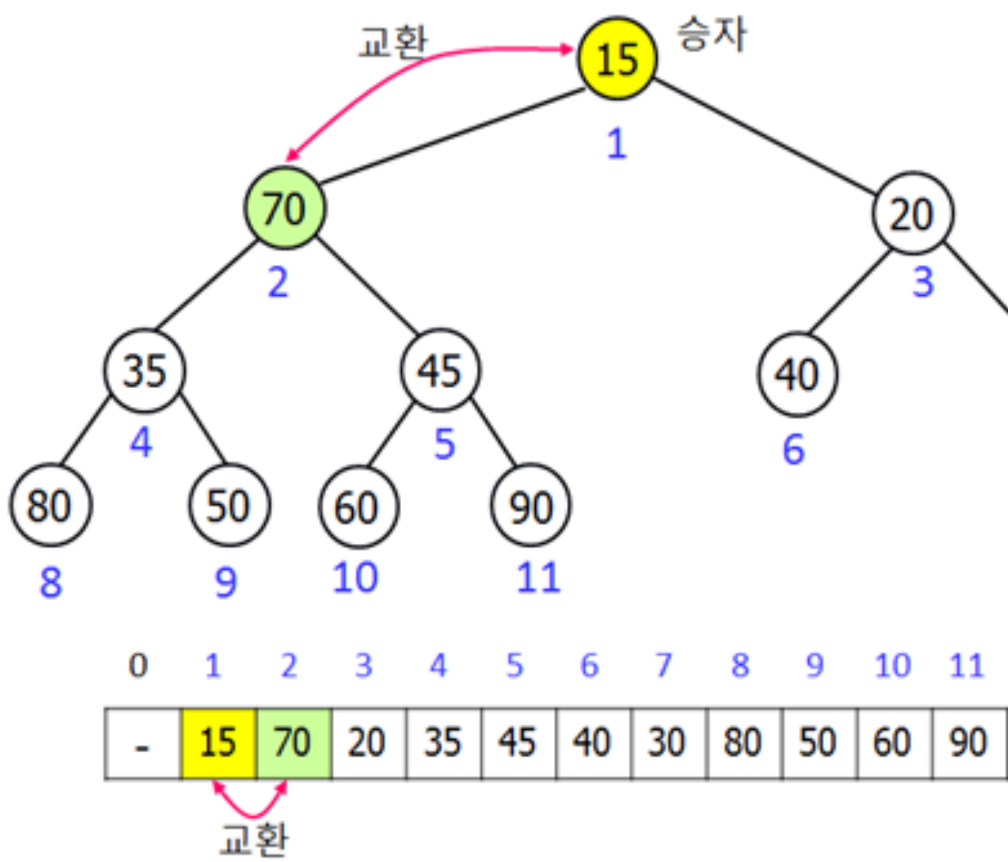


(a) 마지막 항목을 루트로 이동

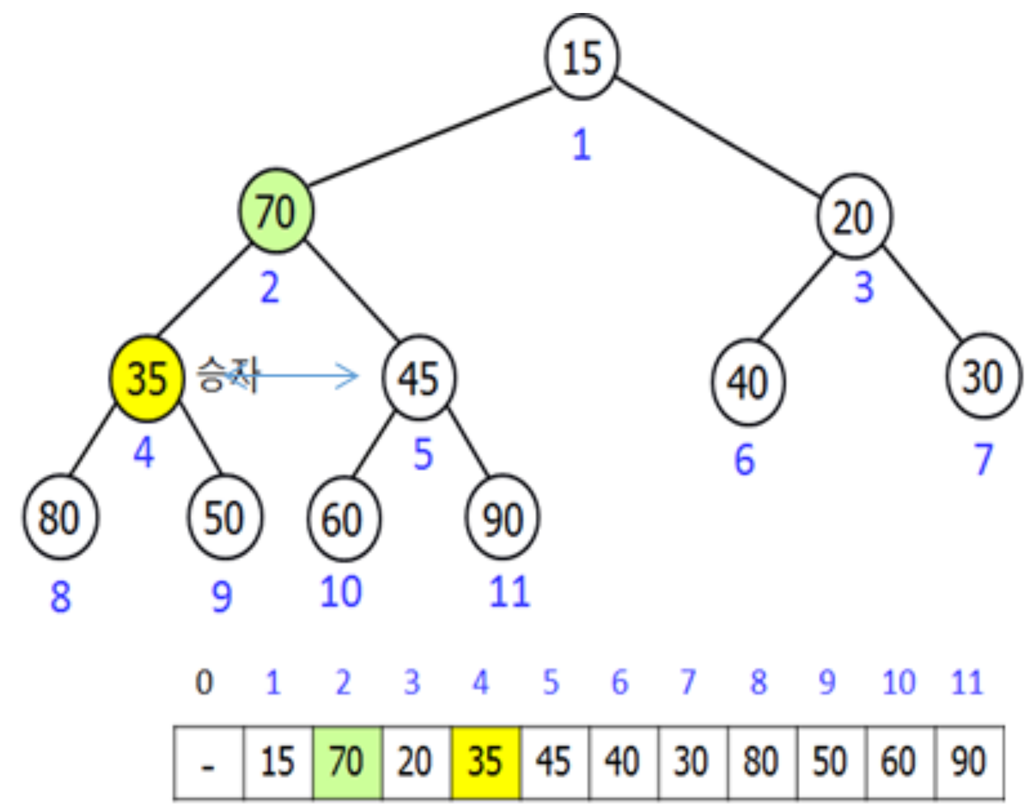


(b) 15와 20 중에 15가 승자

최소값 삭제

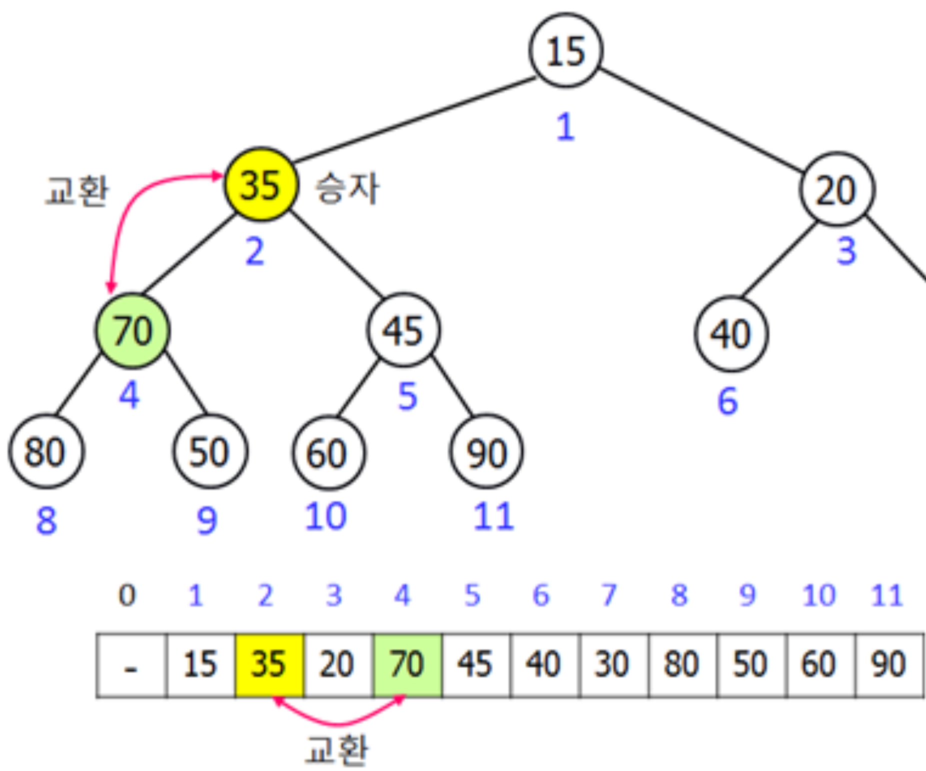


(c) 승자인 15와 루트를 교환

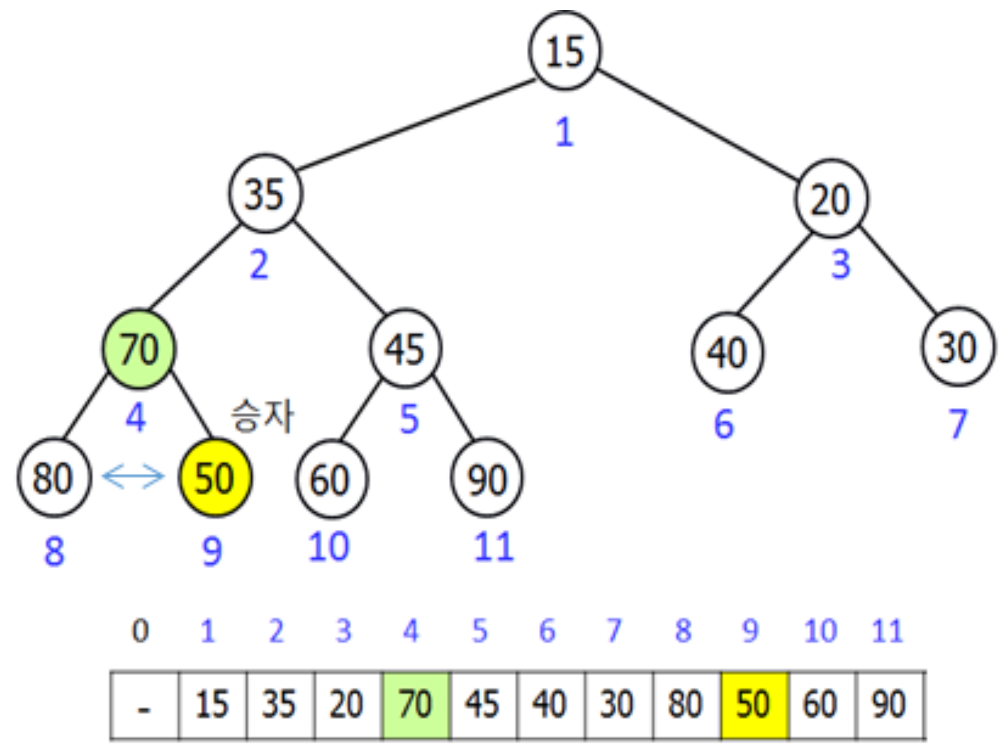


(d) 35와 45 중에 35가 승자

최소값 삭제



(e) 승자인 35와 70를 교환



(f) 80과 50 중에 50이 승자

최소값 삭제

delete_min()

```python
    def delete_min(self): # 최솟값 삭제
        if self.N == 0:
            print('힙이 비어 있음')
            return None
        minimum = self.a[1]
        self.a[1], self.a[-1] = self.a[-1], self.a[1]
        del self.a[-1]
        self.N -= 1
        self.downheap(1)
        return minimum
```

heapq.pop()과 동일함

a[1]과 a[N] 교환

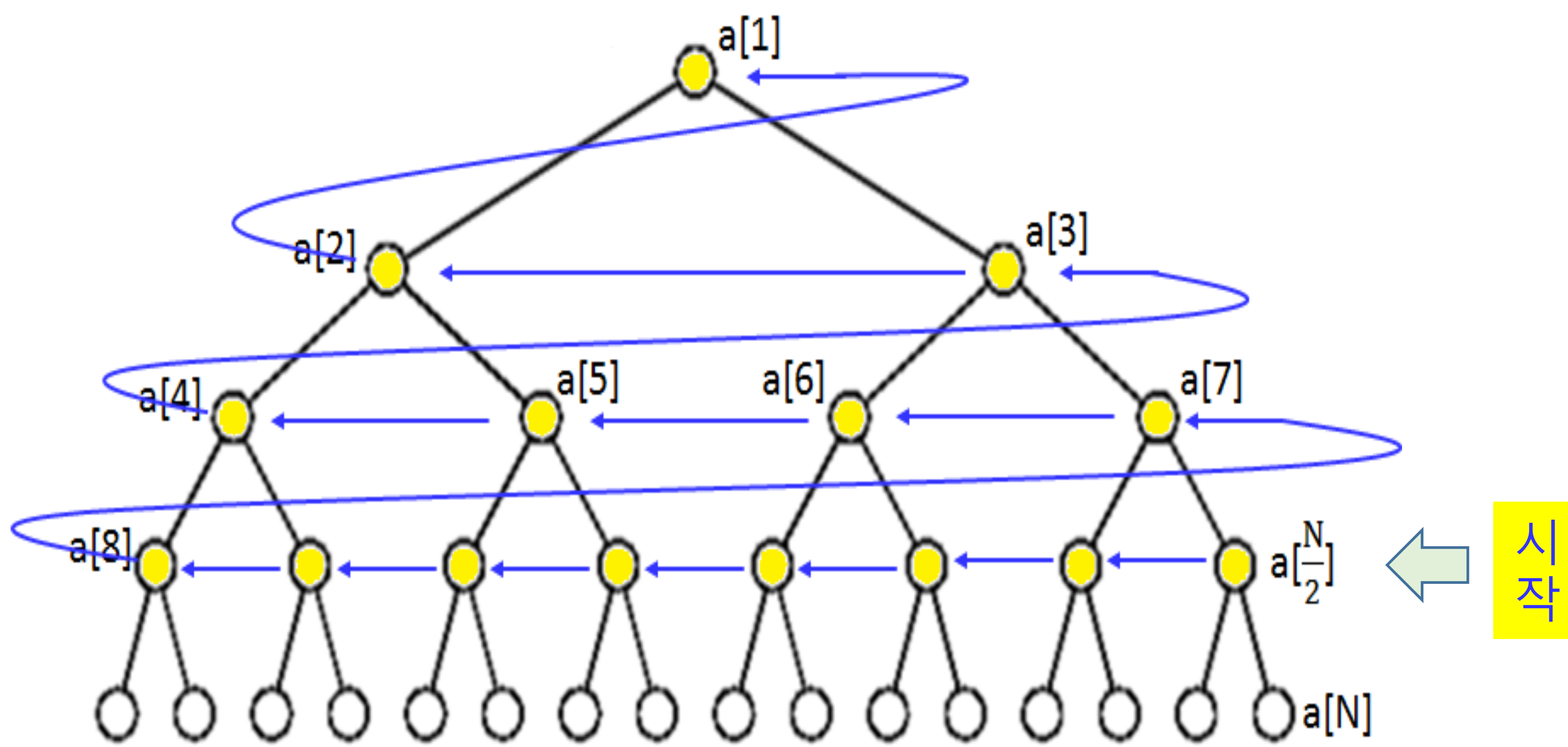
힙속성 회복시키기위해

맨 위에 가장 작은 것을 놓은 다음, 주어진 노드가 두 자식 노드보다 작을 때까지 아래로 내려보냄

힙 만들기
create_heap() + downheap()

- 상향식 힙만들기(Bottom-up Heap Construction)
- [핵심 아이디어]
    - 각 노드에 대해, 상향식 방식으로 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.

힙 만들기
create_heap() + downheap()



힙 만들기

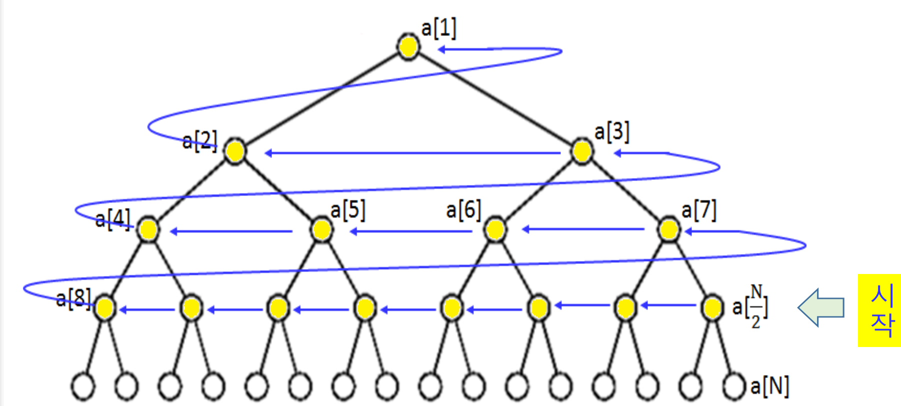
```python
class BHeap:
    def __init__(self, a):
        self.a = a
        self.N = len(a) - 1

    def create_heap(self): # 초기 힙 만들기
        for i in range(self.N//2, 0, -1):
            self.downheap(i)

```

이진힙 생성자
리스트 a
항목 수 N

heapq.heapify()와 동일함



주어진 노드가 두 자식 노드보다 작을 때까지 아래로 내려보냄

```python
    def downheap(self, i): # 힙 내려가며 힙속성 회복
        while 2*i <= self.N:
            k = 2*i
            if k < self.N and self.a[k][0] > self.a[k+1][0]:
                k += 1
            if self.a[i][0] < self.a[k][0]:
                break
            self.a[i], self.a[k] = self.a[k], self.a[i]
            i = k
```

왼쪽, 오른쪽자식 중에서 승자 결정

힙속성 만족하면 루프 나가기

자식 승자와 현재 노드 교환

자식 레벨로 이동

삽입
insert() + upheap()

- [1] 힙의 마지막 노드(즉, 리스트의 마지막 항목)의 바로 다음 비어 있는 원소에 새로운 항목을 저장

- [2] 루트 방향으로 올라가면서 부모의 키와 비교하여 힙속성이 만족될 때까지 노드를 교환

    - [2]의 과정은 이파리로부터 위로 올라가며 수행되므로 upheap 이라 부름

삽입

insert() + upheap()

```python
    def insert(self, key_value): # 삽입 연산
        self.N += 1
        self.a.append(key_value)
        self.upheap(self.N)
```

heapq.push()와 동일함

새 항목을 힙 마지막에 추가

힙속성 회복시키기위해

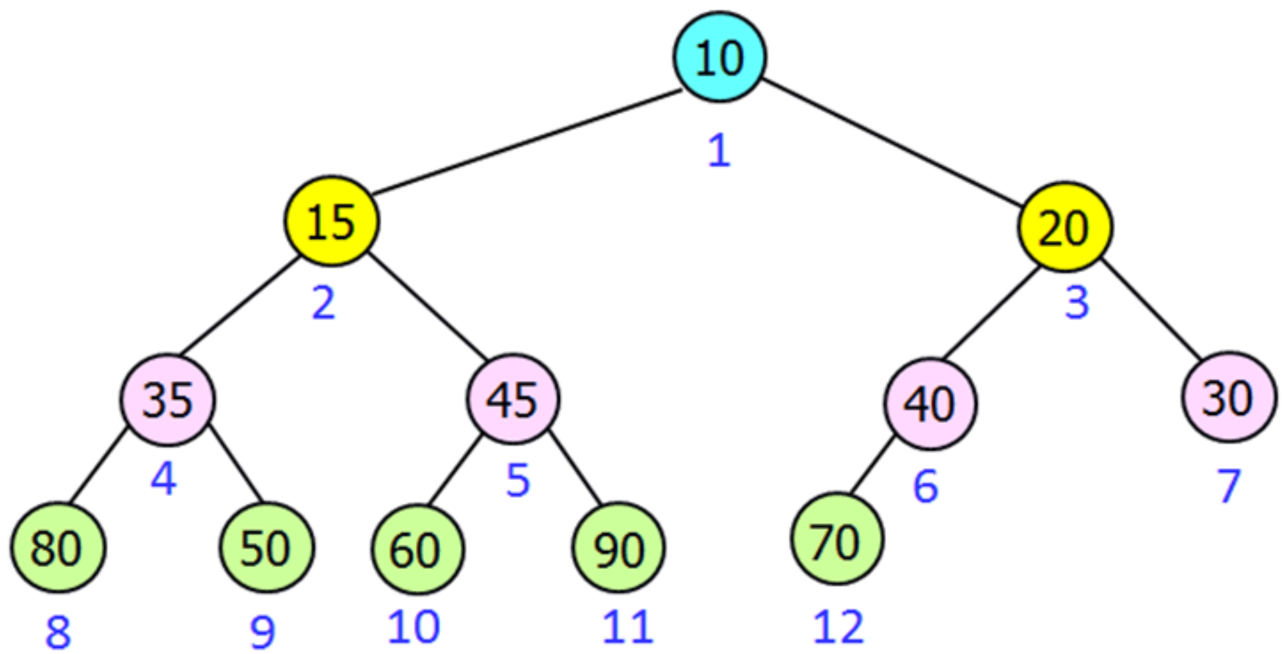
```python
    def upheap(self, j): # 힙 올라가며 힙속성 회복
        while j > 1 and self.a[j//2][0] > self.a[j][0]:
            self.a[j], self.a[j//2] = self.a[j//2], self.a[j]
            j = j//2

    def print_heap(self): # 힙 출력
        for i in range(1, self.N+1):
            print('[%2d' % self.a[i][0], self.a[i][1], ']', end='')
        print('\n힙 크기 = ', self.N)
```

부모와 자식 교환

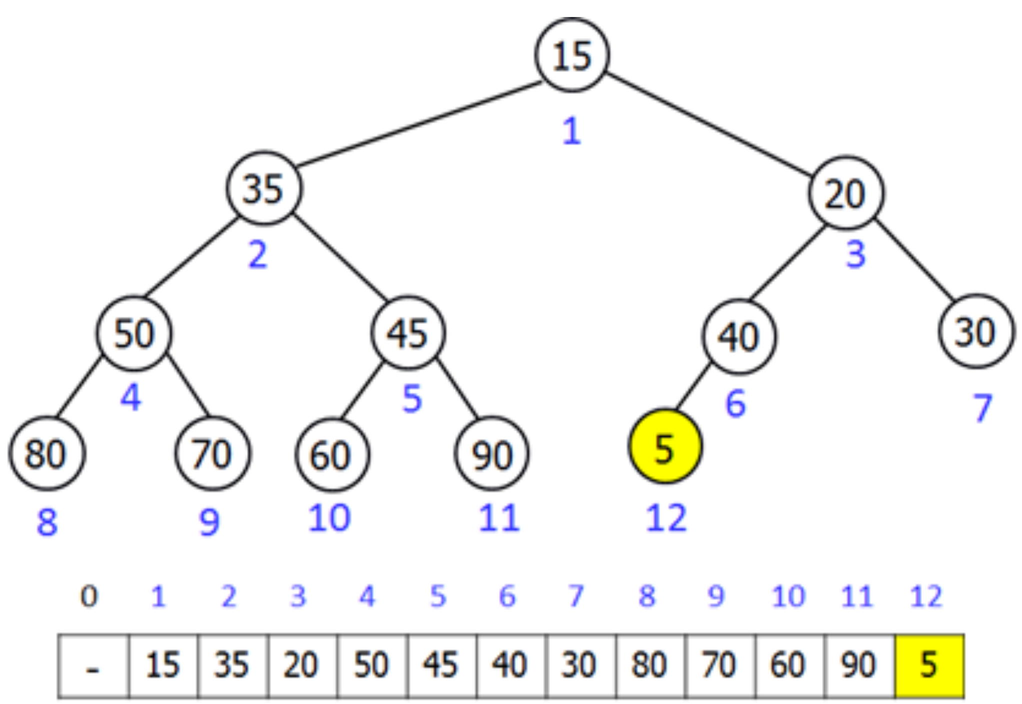
현재 노드가 한 층 올라감

삽입
insert() + upheap()

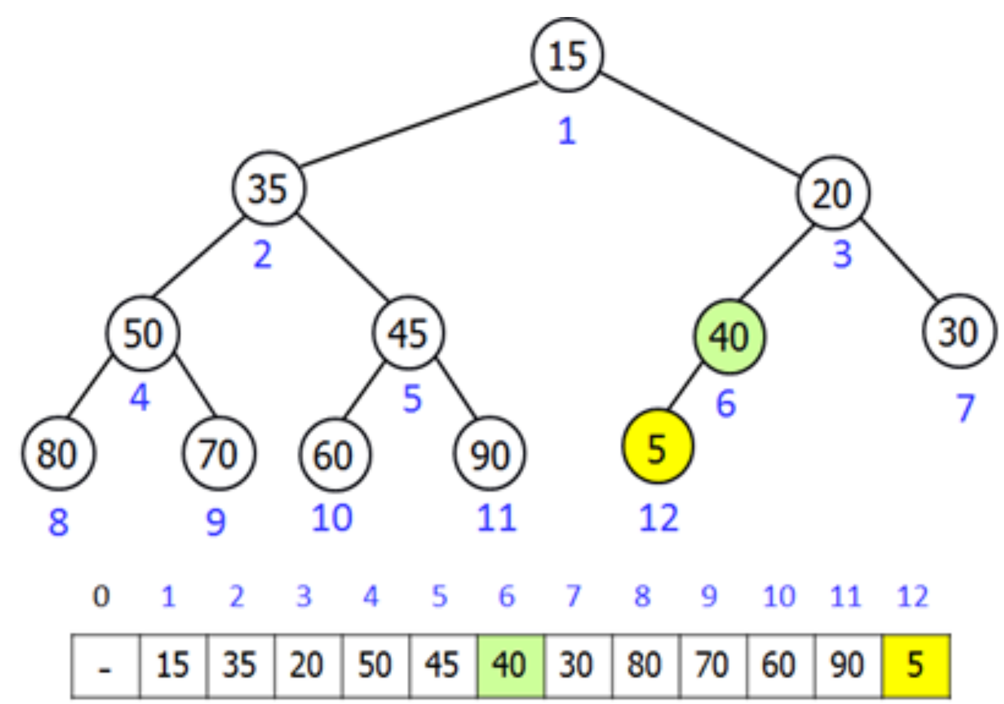


0	1	2	3	4	5	6	7	8	9	10	11	12
-	10	15	20	35	45	40	30	80	50	60	90	70

최소값 (5) 삽입

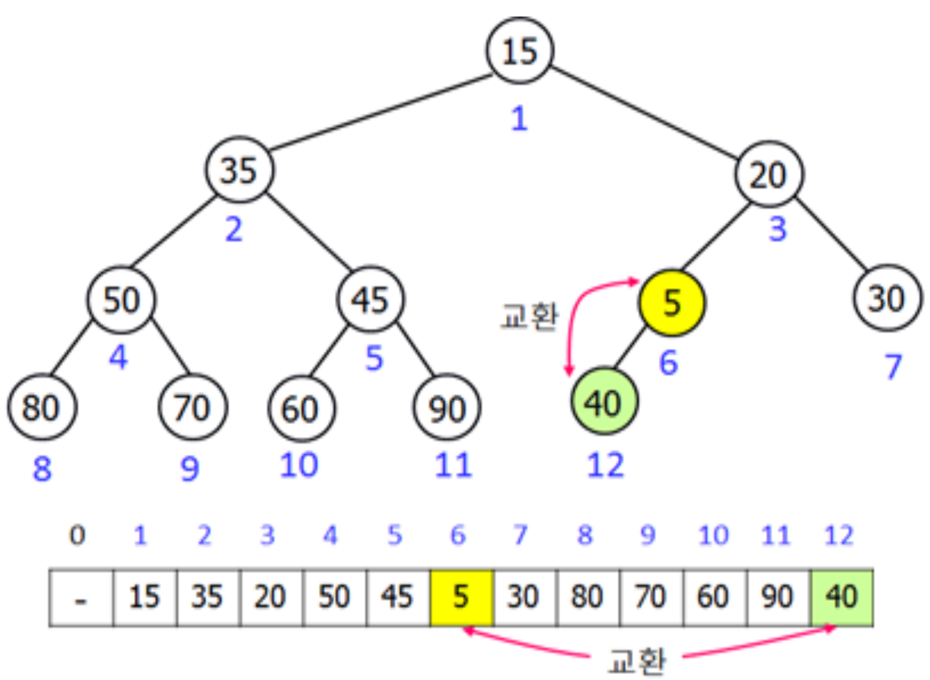


(a) 5를 마지막 항목(90) 다음에 저장

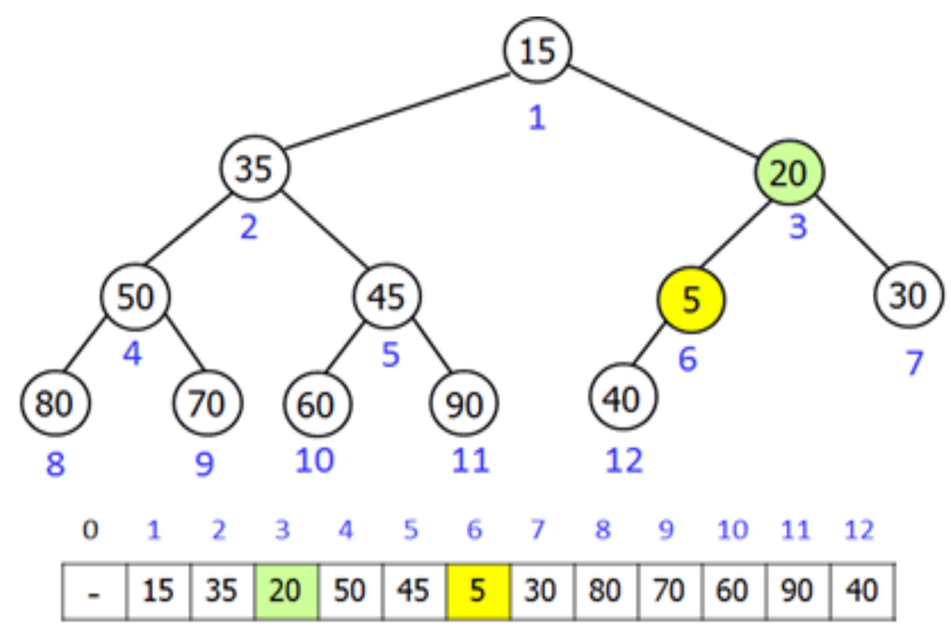


(b) a[12]의 5와 부모 a[6]의 40 비교

최소값 삽입

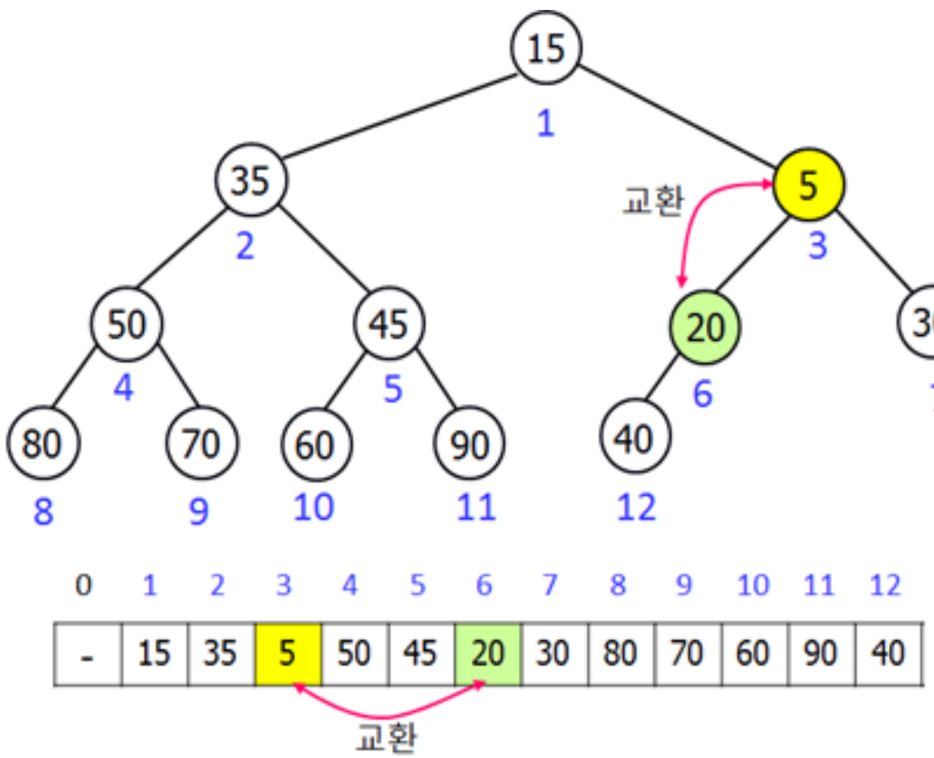


(c) 5와 40을 교환

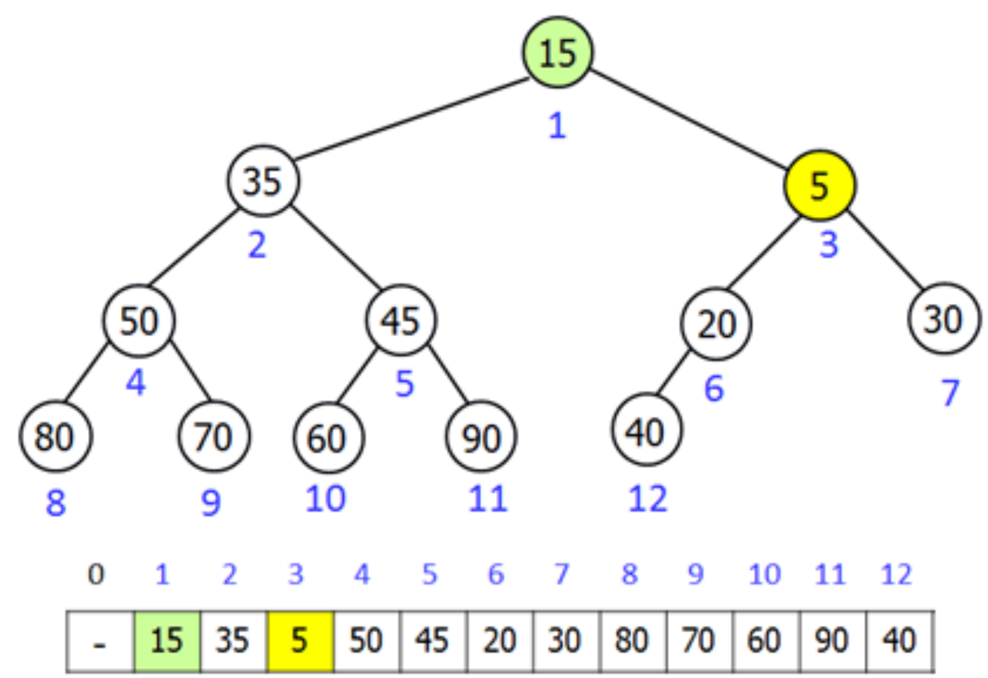


(d) a[6]의 5와 부모 a[3]의 20 비교

최소값 삽입

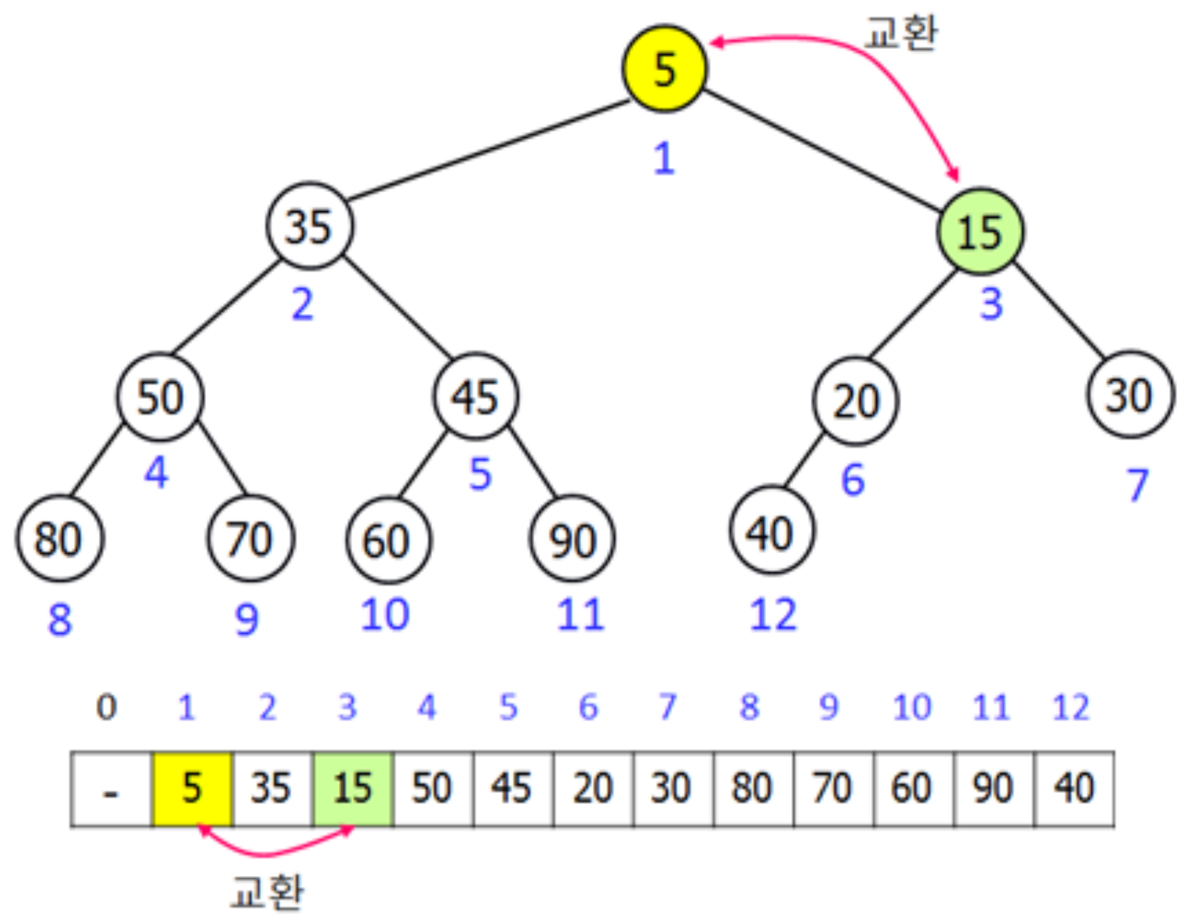


(e) 5와 20을 교환



(f) a[3]의 5와 부모 a[1]의 15 비교

최소값 삽입



(g) 5와 15를 교환

수행시간

- Insert 연산을 위한 upheap은 삽입된 노드로부터 최대 루트까지 올라가며 부모와 자식노드를 교환

- delete_min 연산에서는 힙의 마지막 노드를 루트로 이동한 후, downheap을 최하위 층의 노드까지 교환해야 하는 경우가 발생

- 힙에서 각 연산의 수행시간은 힙의 높이에 비례

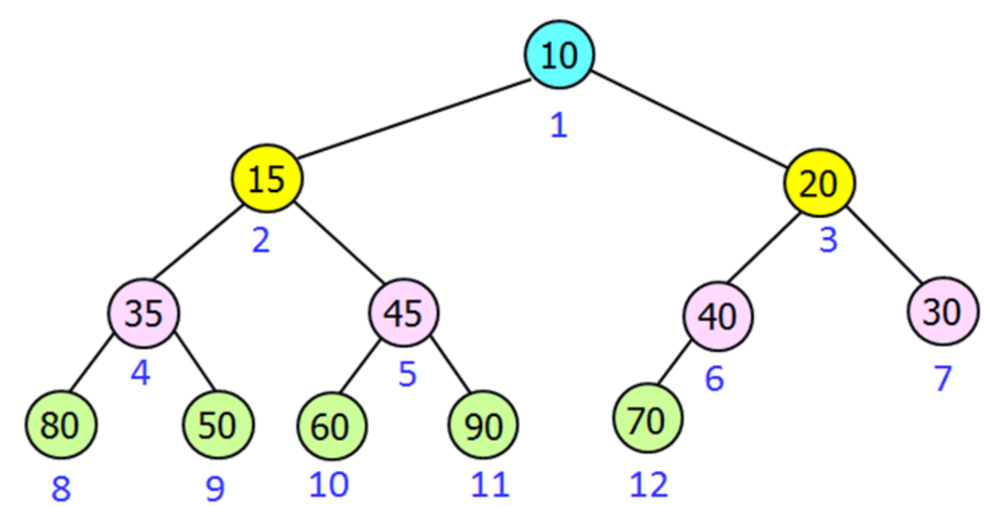
- 힙은 완전이진트리이므로 힙에 N개의 노드가 있으면 그 높이는 $\lceil \log(n+1) \rceil$

- 각 힙 연산의 수행시간은 O(logN)

힙 만들기 수행시간

- 노드 수가 N인 힙의 각 층에 있는 노드 수를 살펴보자. 단, 간단한 계산을 위하여 $N = 2^k - 1$로 가정, k는 양의 상수

- 최하위층 (h = 0)의 노드 수 = $\lceil N/2 \rceil$
  - (h=1)의 노드 수= $\lceil N/2^2 \rceil$
  - (h=2)의 노드 수 = $\lceil N/2^2 \rceil$
  - h층의 노드 수 = $\lceil N/2^{h+1} \rceil$



- 힙만들기는 h=1인 경우부터 시작하여 최상위층의 루트까지 각 노드에 대해 downheap을 수행

힙 만들기 수행시간

Last parents **...** **Root**

$$T(N) = 1 \cdot \frac{N}{2^2} + 2 \cdot \frac{N}{2^3} + 3 \cdot \frac{N}{2^4} + \cdots + (\log N - 1) \cdot \frac{N}{2^{\log N}}$$

$$\leq \sum_{h=1}^{\log N} h \cdot \frac{N}{2^{h+1}} = \frac{N}{2} \sum_{h=1}^{\log N} \frac{h}{2^h} \leq \frac{N}{2} \cdot 2, \quad \sum_{x=0}^{\infty} \frac{x}{2^x} = 2 \text{ 이므로}$$

$$= O(N)$$

파이썬의 우선순위큐 (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

Wrap-Up

- 트리는 계층적 자료구조로서 배열이나 연결리스트의 단점을 보완하는 자료구조

- 왼쪽자식-오른쪽형제 표현은 노드의 차수가 일정하지 않은 일반적인 트리를 구현하는 매우 효율적인 자료구조

- 포화이진트리는 각 내부노드가 2개의 자식 노드를 가지는 트리

- 완전이진트리는 마지막 레벨을 제외한 각 레벨이 노드들로 꽉 차있고, 마지막 레벨에는 노드들이 왼쪽부터 빠짐없이 채워진 트리이다.

- 포화이진트리는 완전이진트리이다.

Wrap-Up

- 이진트리의 순회 방법
    - 전위순회(NLR)
    - 중위순회(LNR)
    - 후위순회(LRN)
    - 레벨순회-레벨순회는 큐 자료구조를 사용해서 구현
- 이진트리 높이 계산과 노드 수의 계산에는 후위순회가 적합, 이진트리의 비교에는 전위순회가 적합

Wrap-Up

- 스택 없이 이진트리를 순회하기 위해 노드의 None 레퍼런스 대신 다음에 방문할 노드의 레퍼런스를 저장한 이진트리를 스레드 이진트리라고 함

- 이진트리의 높이 및 노드 수의 계산, 각 트리 순회, 동일성 검사는 트리의 모든 노드들을 방문해야 하므로 각각 O(N) 시간이 소요

- 우선순위큐는 가장 높은 우선순위를 가진 항목을 접근 또는 삭제하는 연산과 삽입 연산을 지원

- 이진힙은 완전이진트리로서 부모의 우선순위가 자식의 우선순위보다 높은 자료구조

- 파이썬은 우선순위큐를 위해 heapq 를 라이브러리로 제공

In next class

- Search Tree
    - Binary SEARCH Tree
    - AVL Tree
    - …