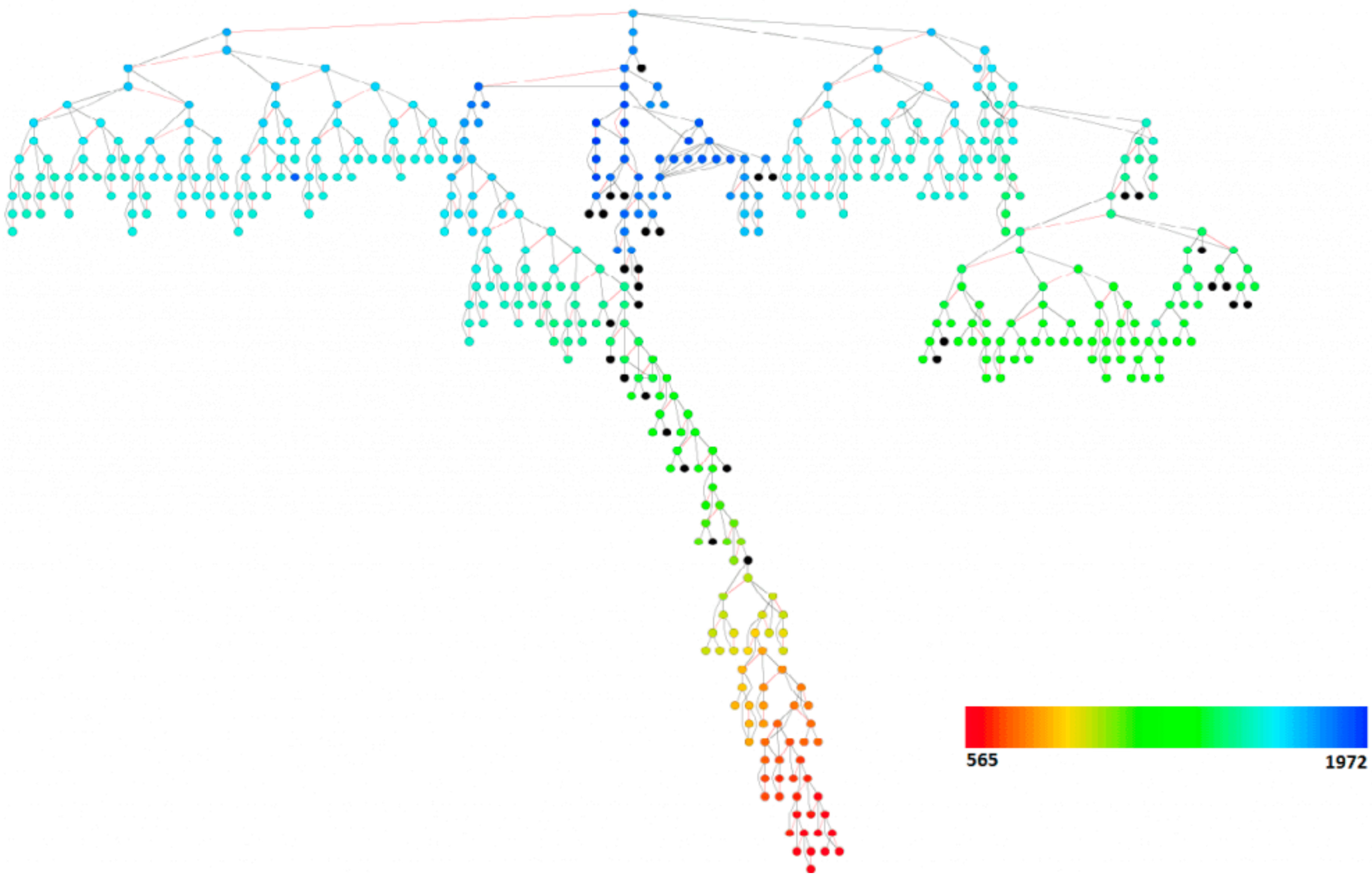


Tree

Jin Hyun Kim

Fall, 2019

Referenes

In this Topic

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

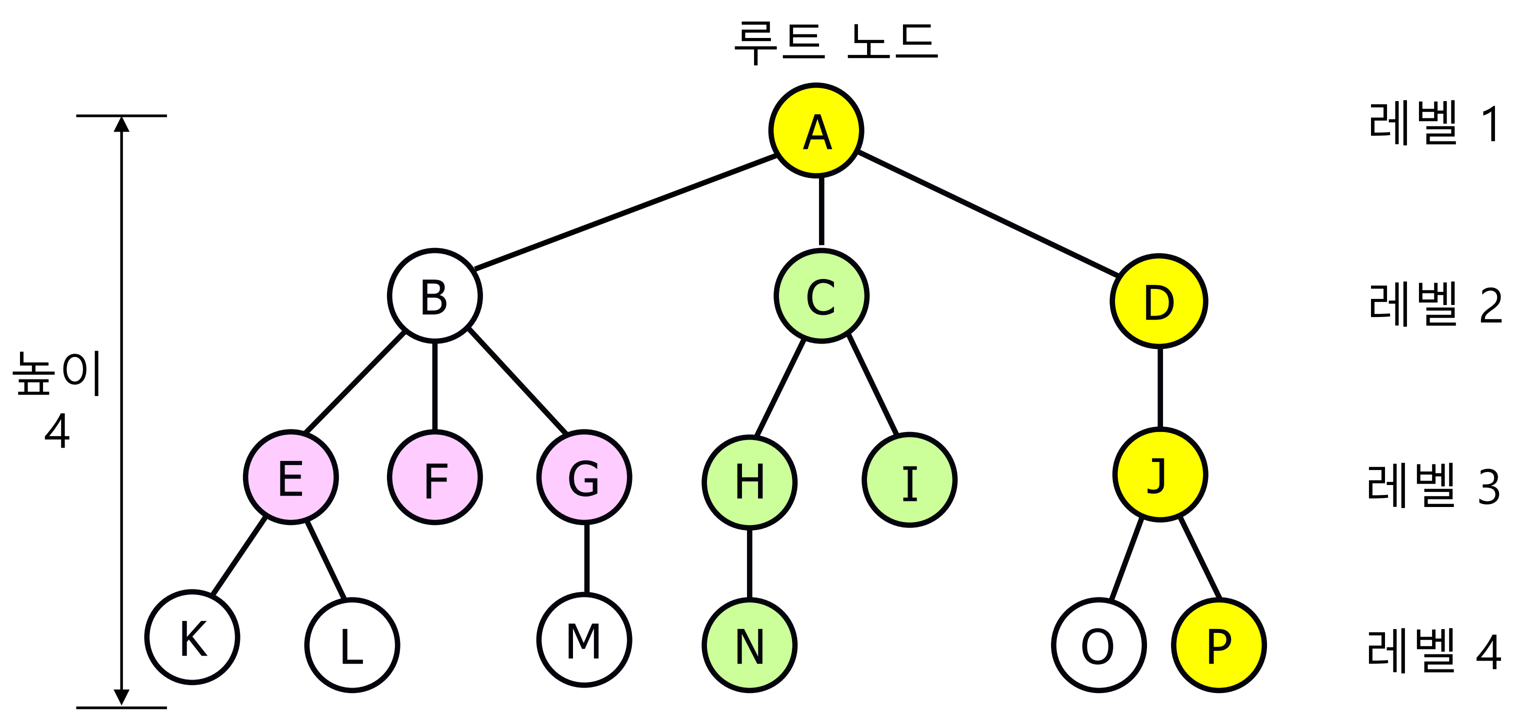
트리

- 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

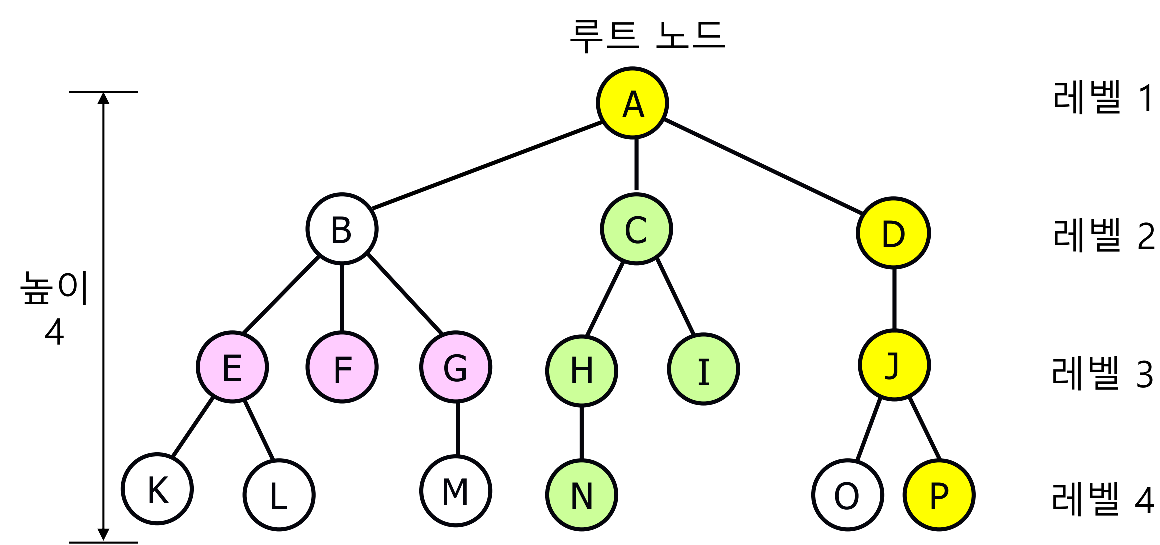
- 일반적인 트리의 정의

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



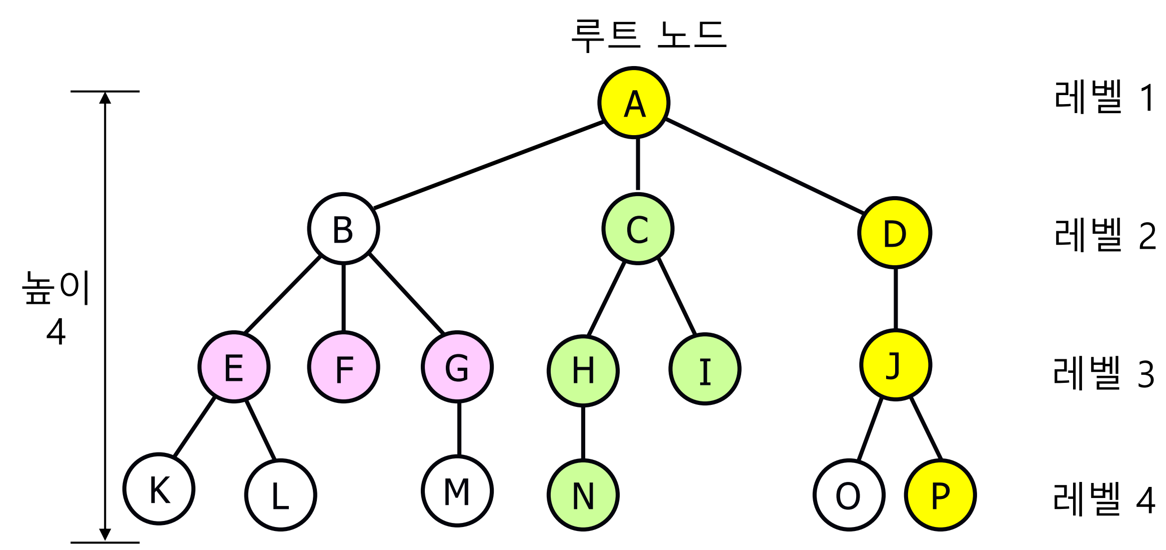
용어

- 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



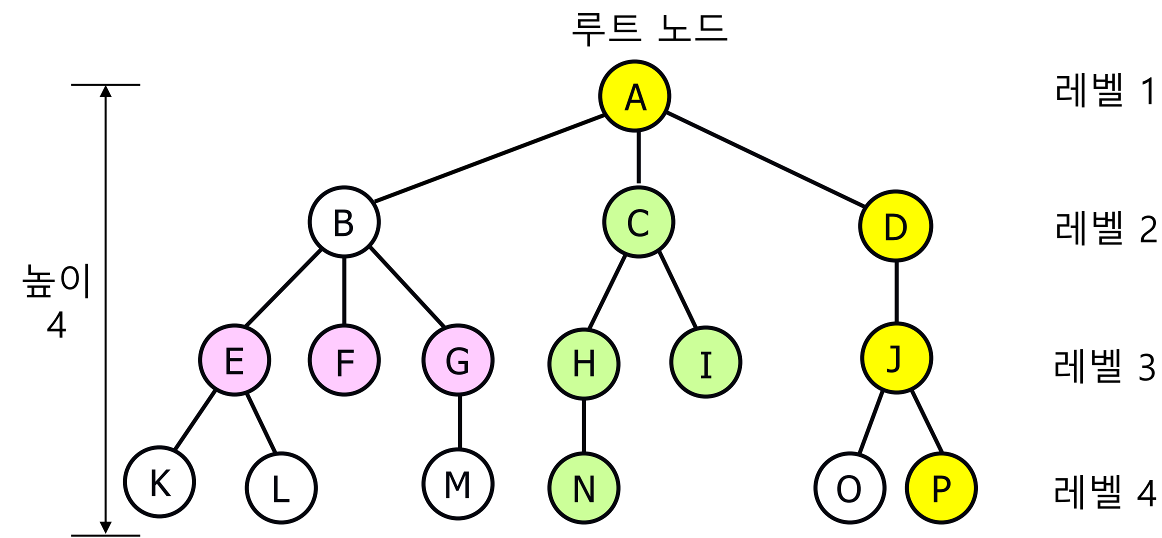
용어

- 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

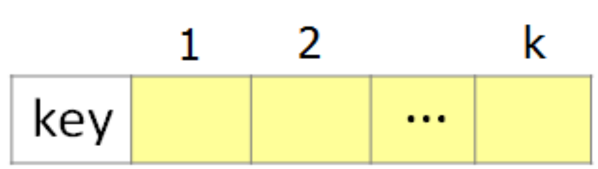


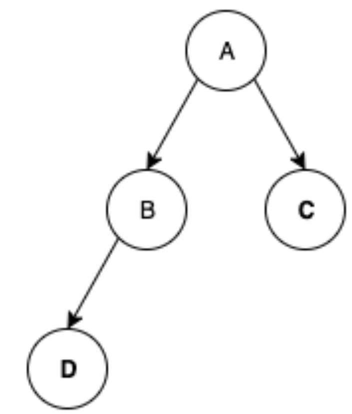
용어

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

특성

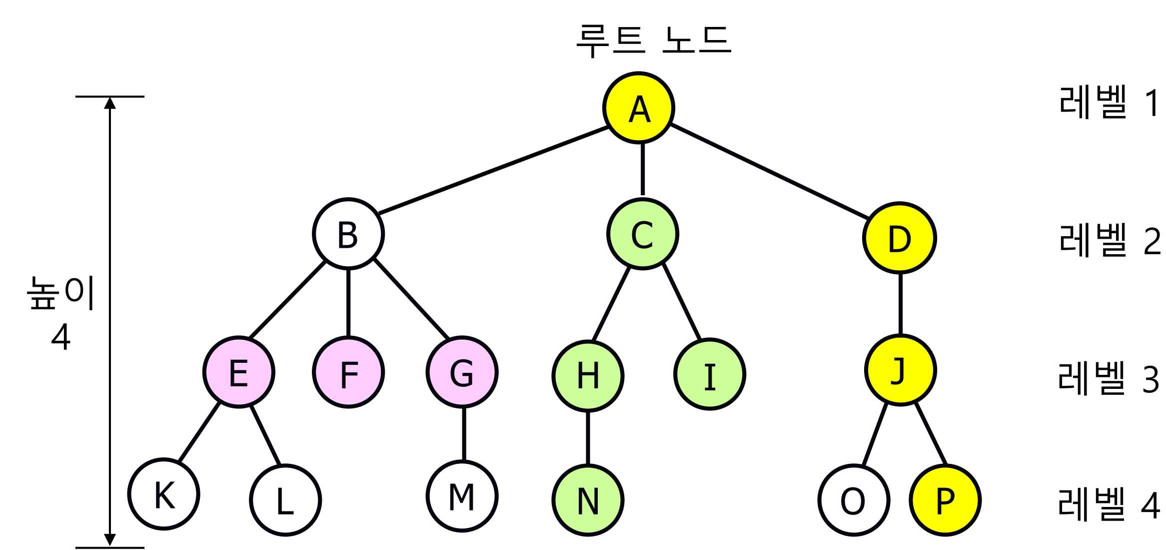
- 일반적인 트리를 메모리에 저장하려면 각 노드에 키와 자식 수만큼의 레퍼런스 저장 필요
    - 노드의 최대 차수가 k라면, k개의 레퍼런스 필드를 다음과 같이 선언해야





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

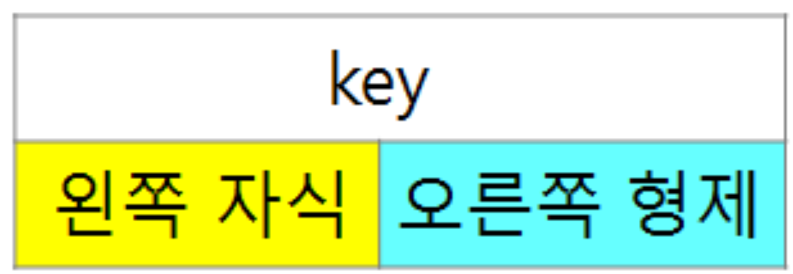
- N개의 노드가 있는 최대 차수가 k인 트리
    - None 레퍼런스 수 (자식으로 연결이 없는 연결의 수) = Nk – (N-1) = N(k-1) + 1
        - Nk = 총 레퍼런스의 수
        - **(N-1) = 트리에서 부모-자식을 연결하는 레퍼런스 수**



k = 3, N=16

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

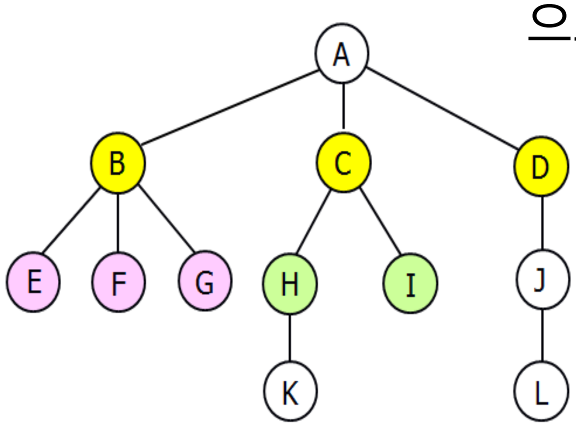
- 노드의 왼쪽 자식과 왼쪽 자식의 오른쪽 형제를 가리키는 2개의 레퍼런스만을 사용

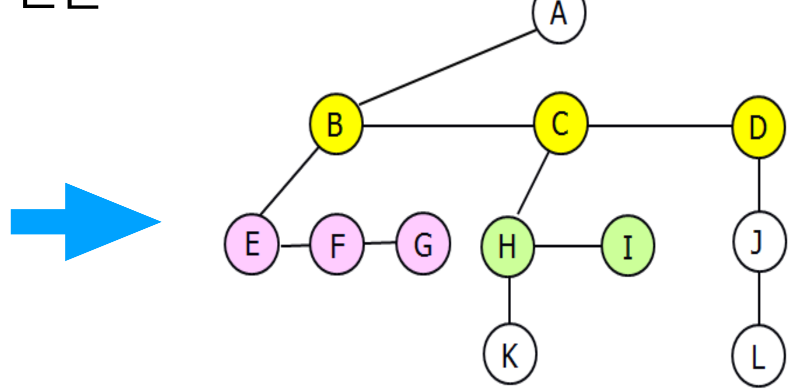


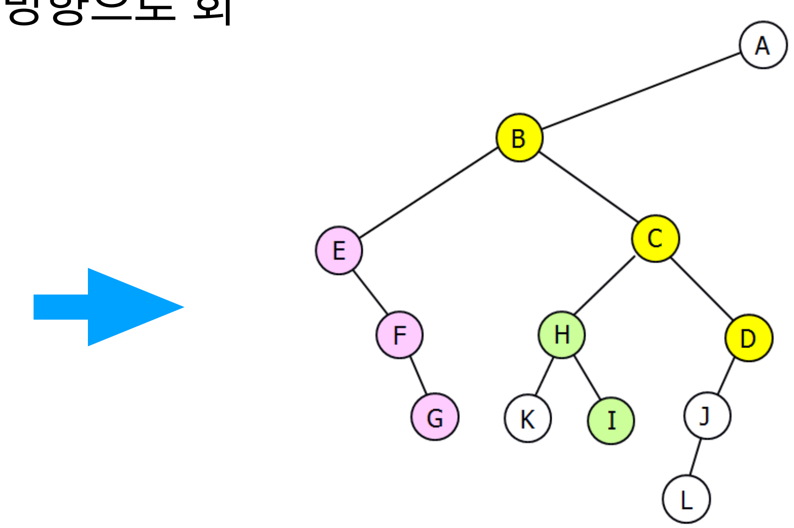
다중노드를 LCRS로

트리를 왼쪽자식-오른쪽형제 표현으로 변환

트리를 45° 시계 방향으로 회전





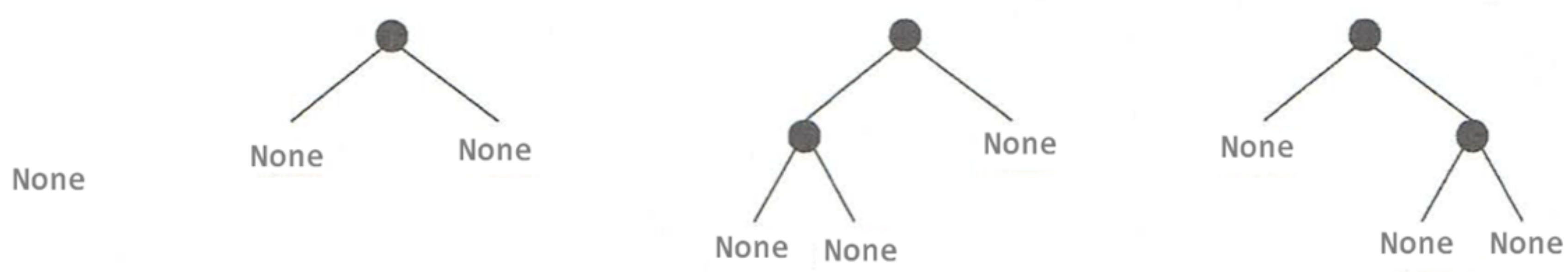


이진트리 (Binary Tree)

- 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) **(b)** **(c)** **(d)**

(a) empty 트리

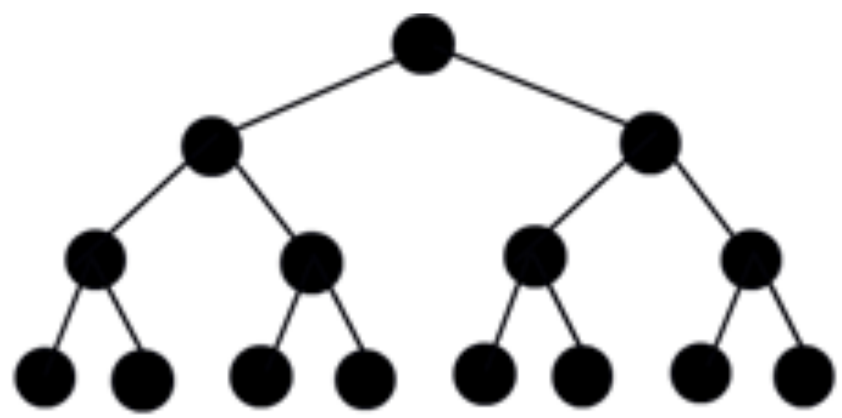
(b) 루트만 있는 이진트리

(c) 루트의 오른쪽 서브트리가 없는(empty) 이진트리

(d) 루트의 왼쪽 서브트리가 없는 이진트리

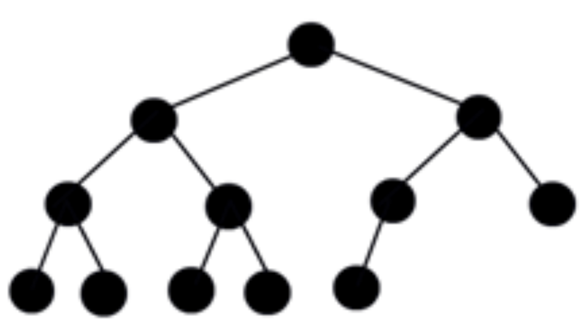
이진트리 (Binary Tree)

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

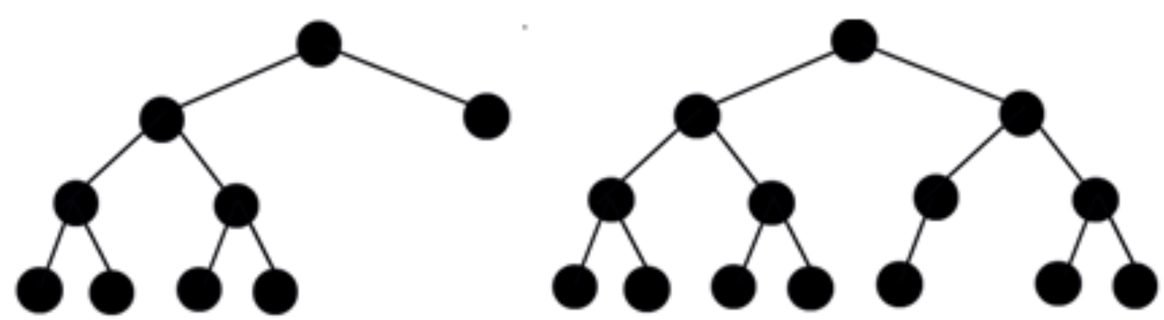


이진트리 (Binary Tree)

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



완전이진트리

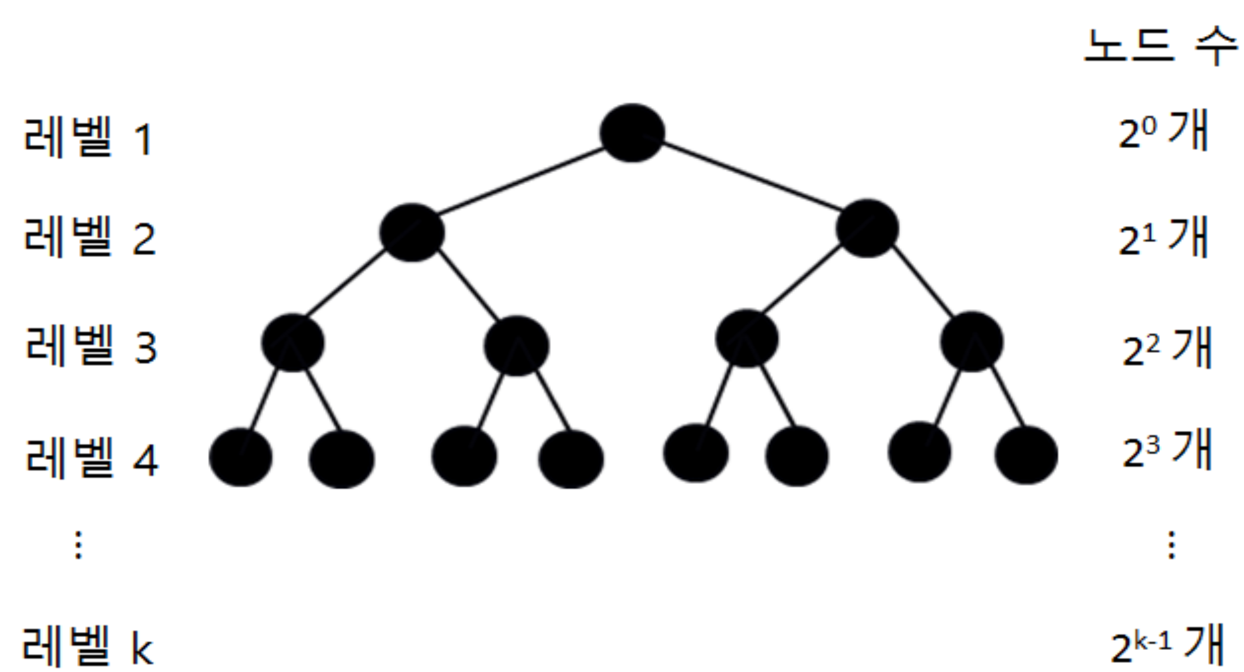


불완전이진트리

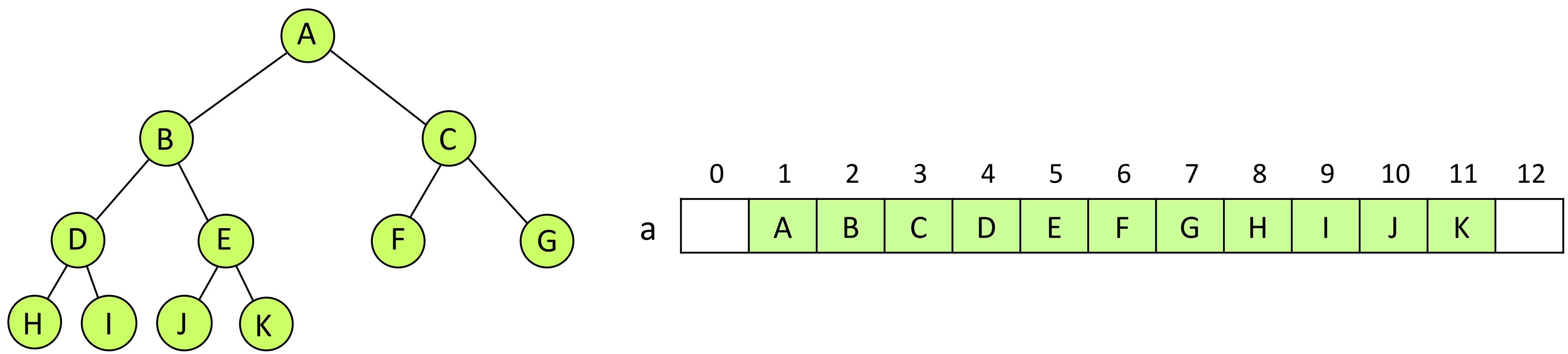
- **[정리] 포화이진트리는 완전이진트리이다.**

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 = $\lceil log_2(N+1) \rceil$

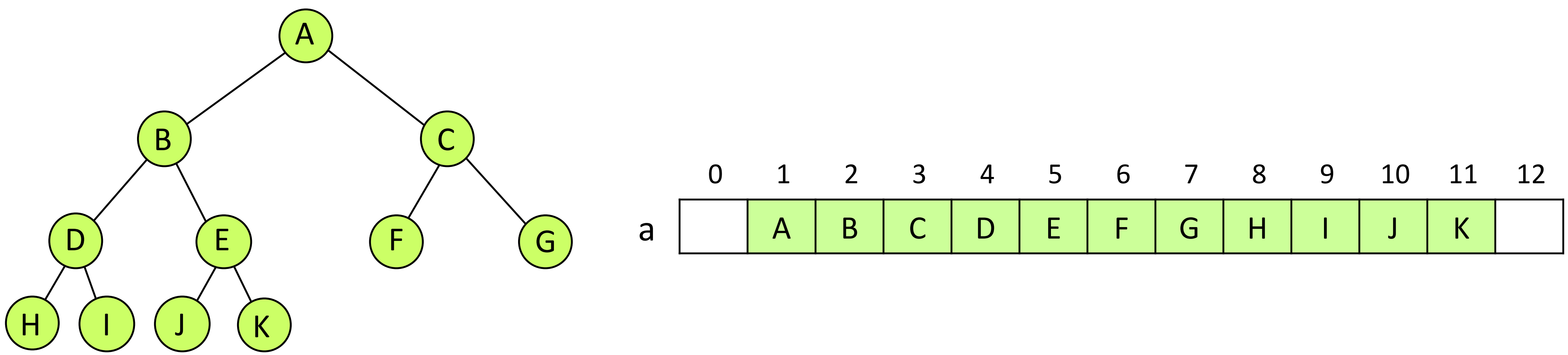


구현 (리스트, 배열)



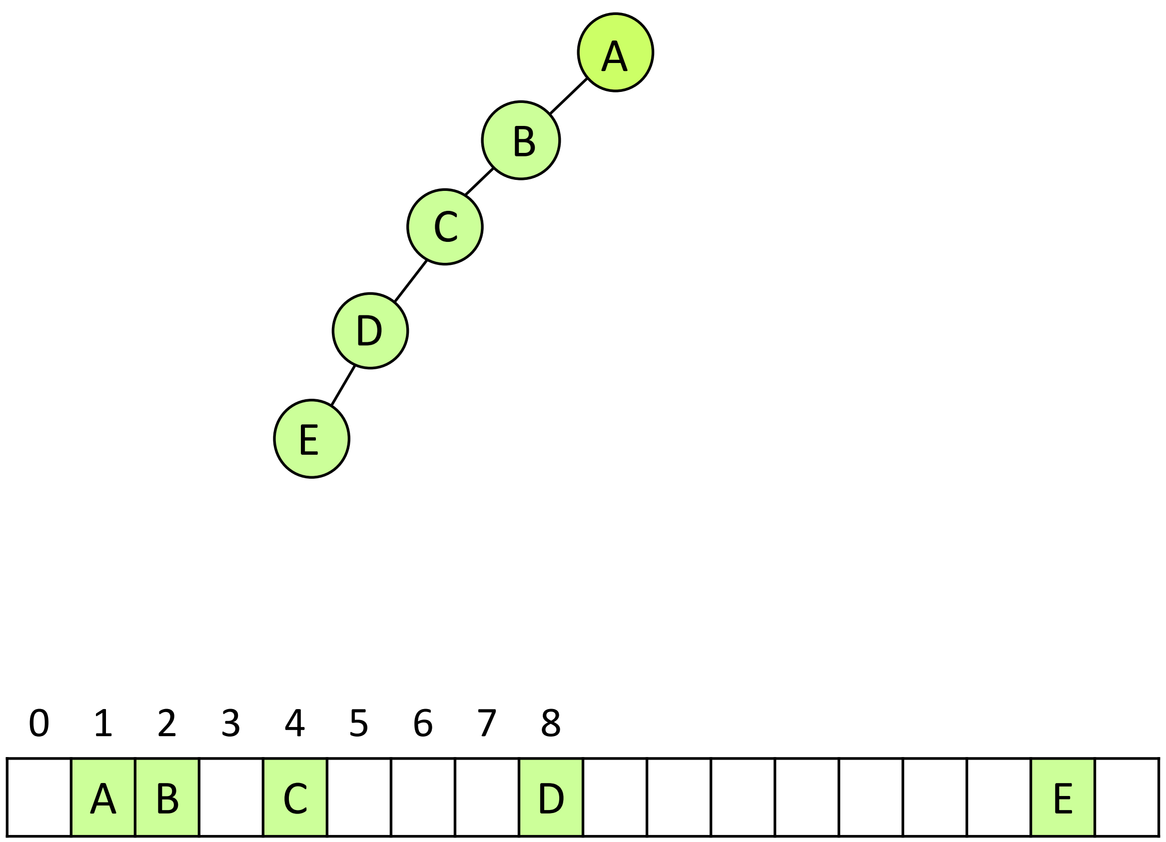
- 리스트에 저장하면 노드의 부모와 자식노드가 리스트의 어디에 저장되어 있는지를 다음과 같은 규칙을 통해 쉽게 알 수 있다.
  - 단, 트리에 N개의 노드가 있다고 가정

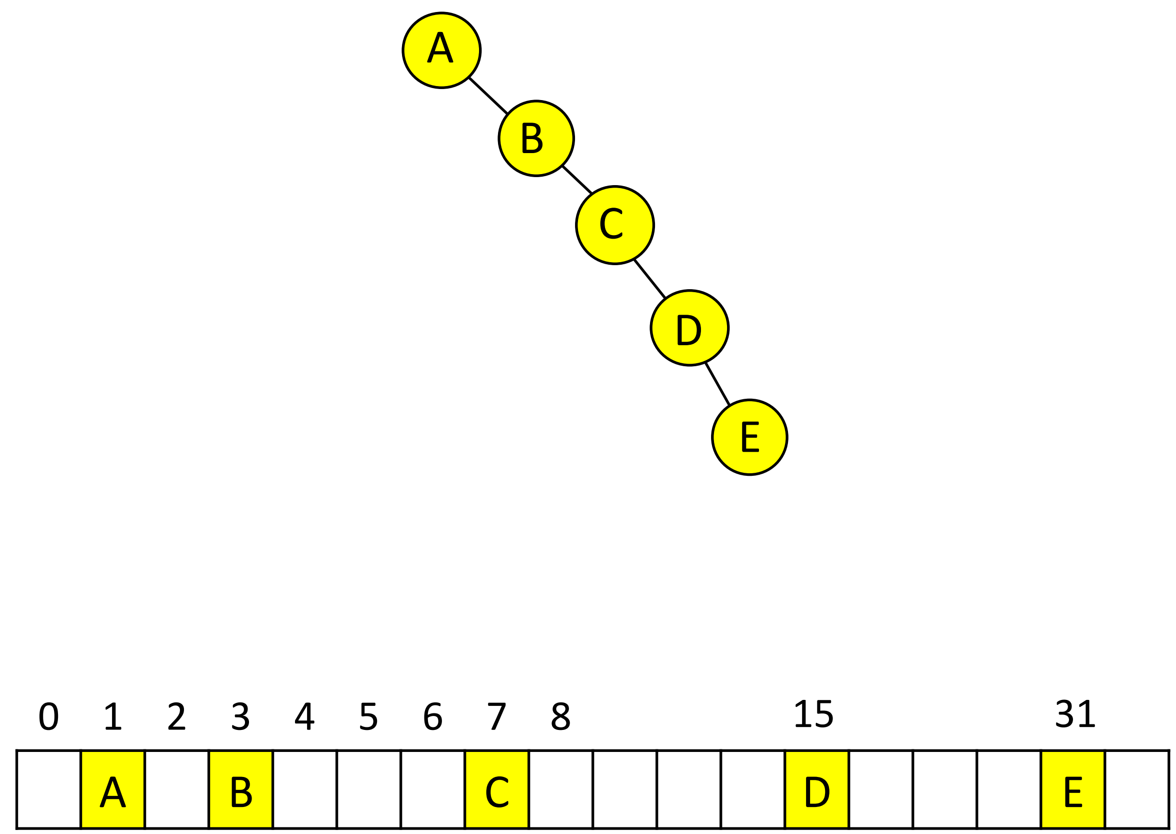
구현 (리스트, 배열)



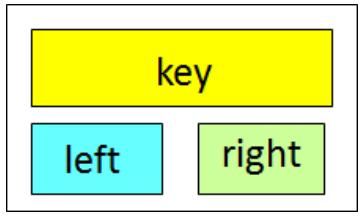
- a[i]의 부모는 a[i//2]에 있다. 단, i > 1이다. e.g.) E의 부모는 a[5//2] = a[2]
- a[i]의 왼쪽자식은 a[2i]에 있다. 단, 2i ≤ N이다. e.g.) E의 왼쪽 자식은 a[2x5] = a[10]
- a[i]의 오른쪽자식은 a[2i+1]에 있다. 단, 2i + 1 ≤ N이다. e.g.) E의 오른쪽 자식은 a[2x5+1] = a[11]

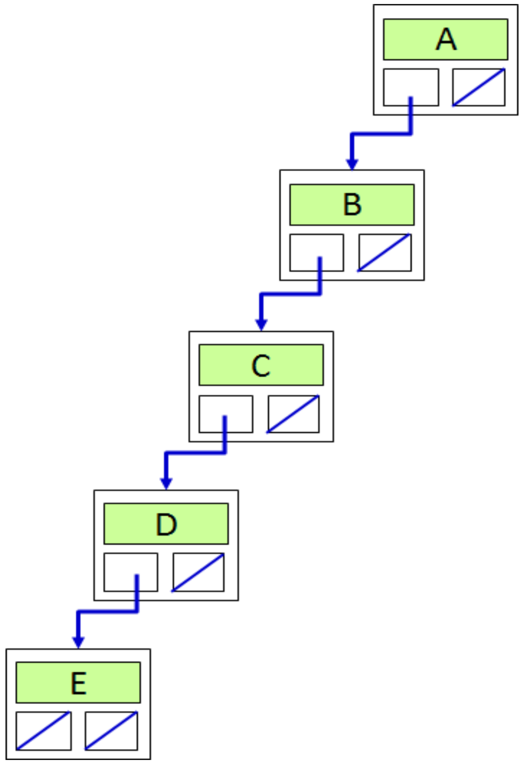
편향 이진트리 구현 (리스트, 배열)

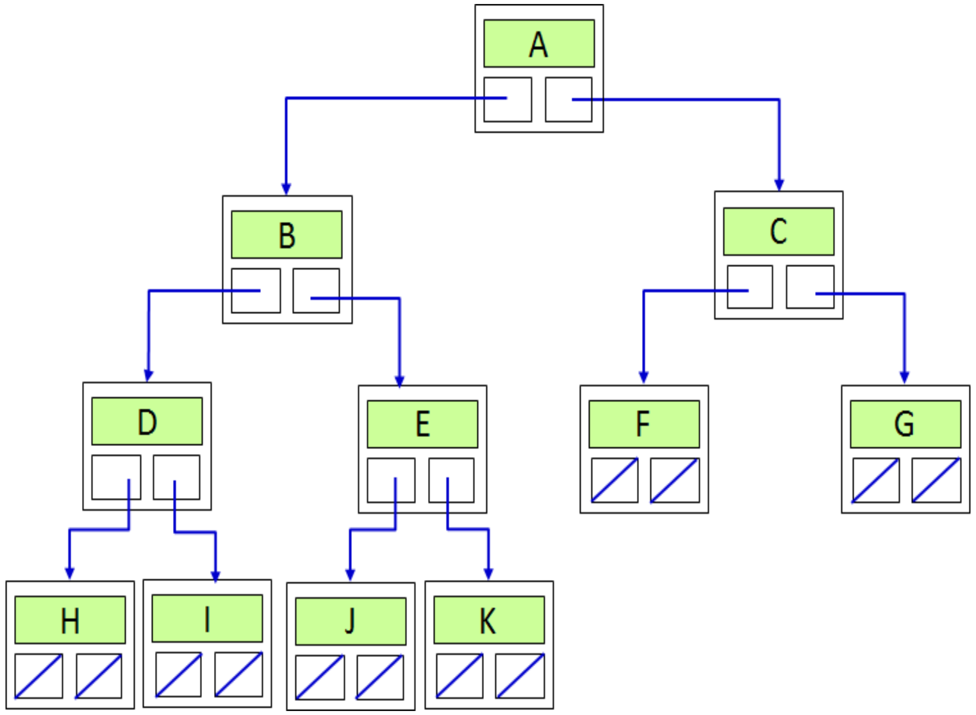




구현 (연결리스트)







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

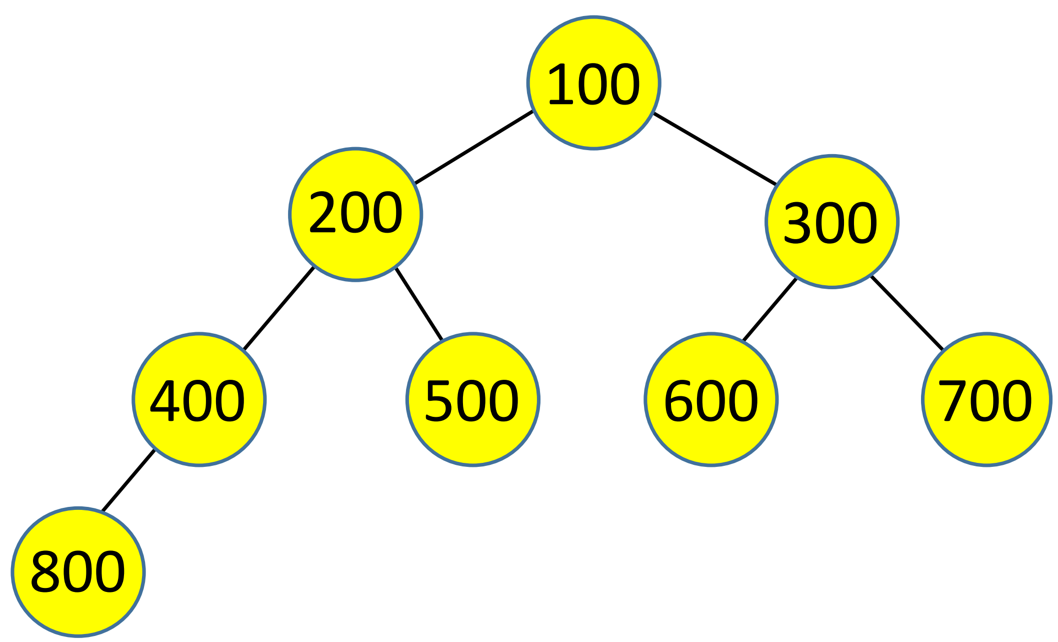
트리

```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

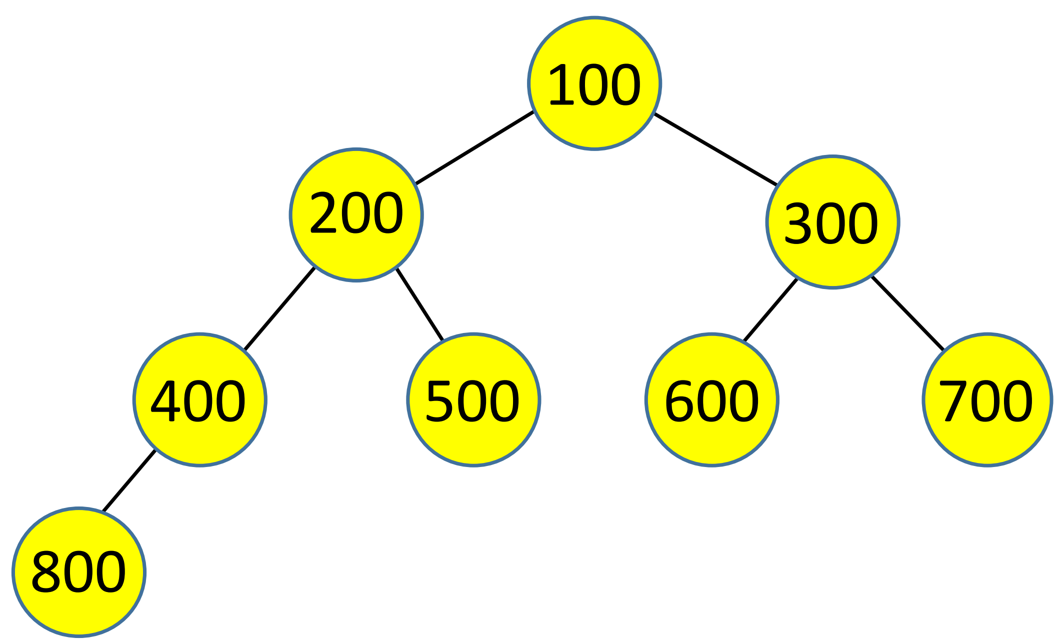
트리 만들기



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



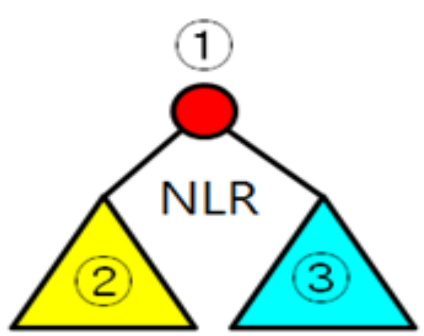
트리연산: 순회

- 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)

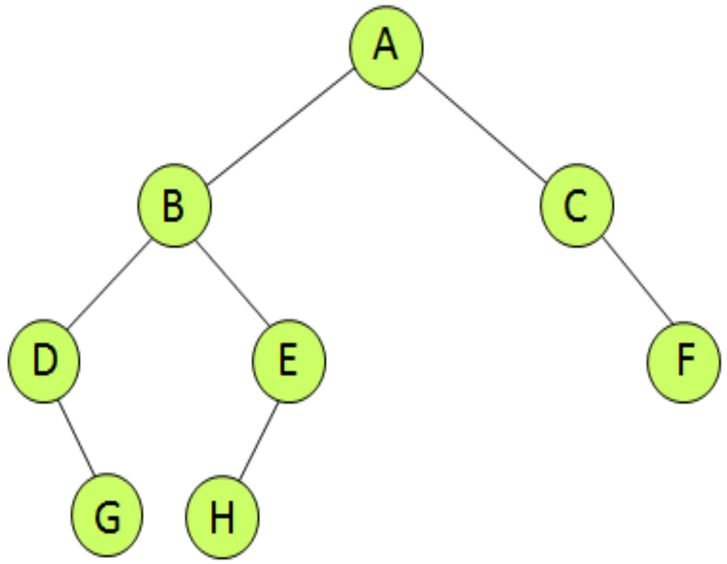
- 전위순회는 노드 n에 도착했을 때 n을 먼저 방문
- 그 다음에 n의 왼쪽 자식노드로 순회를 계속
- n의 왼쪽 서브트리의 모든 노드들을 방문한 후에는 n의 오른쪽 서브트리의 모든 후손 노드들을 방문
- 전위순회의 방문 규칙

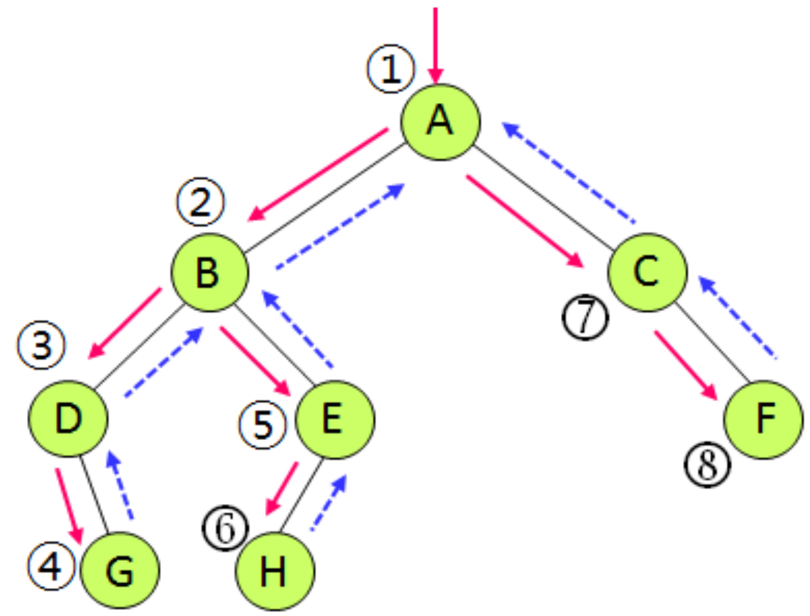



```python
def preorder(self, n): # 전위순회
    if n != None:
        print(str(n.item),' ', end='')
        if n.left:
            self.preorder(n.left)
        if n.right:
            self.preorder(n.right)
```

- 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



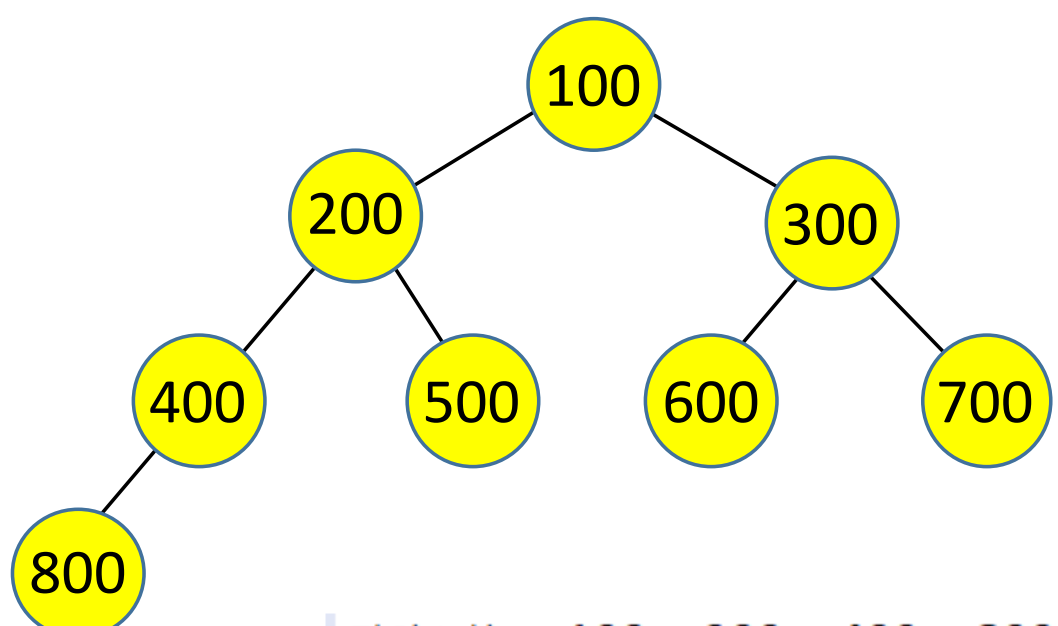


트리 연산: 전위순회

```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)
```

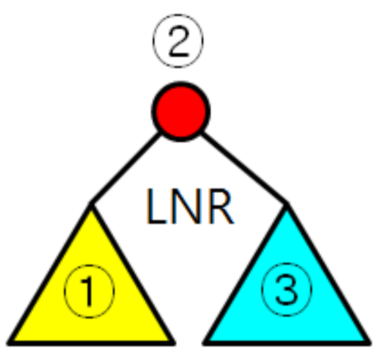
맨 먼저 노드 방문

왼쪽 서브트리 방문 후
오른쪽 서브트리 방문

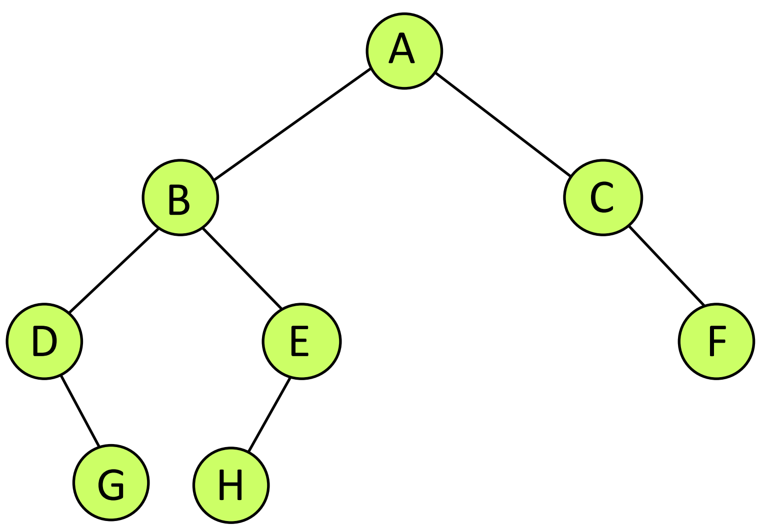


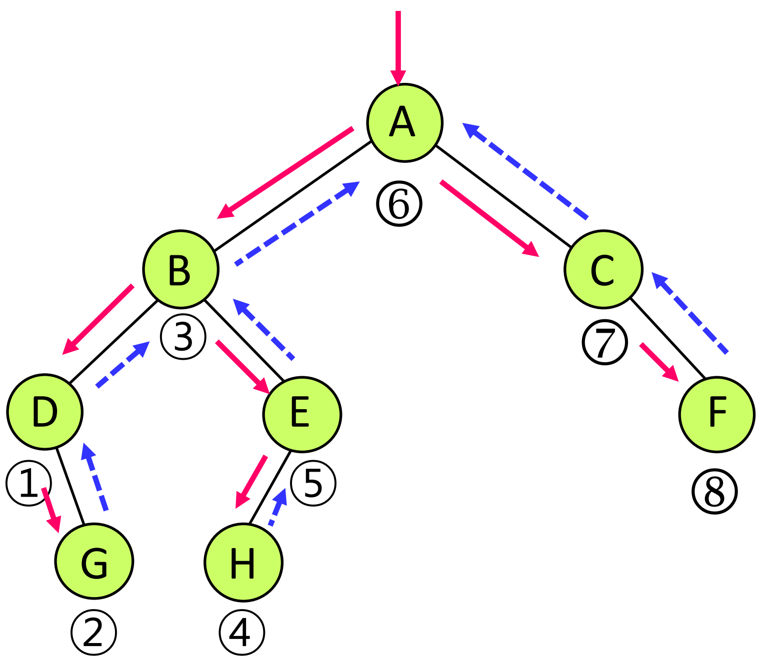
전위순회: 100 200 400 800 500 300 600 700

중위순회 (LNR)




```
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)
```

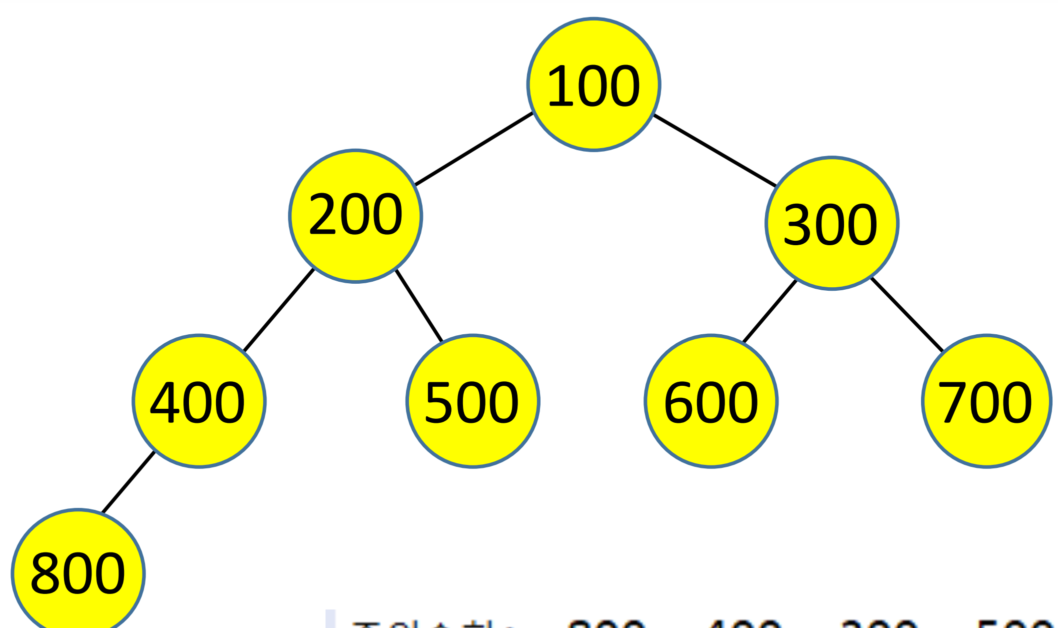




트리 연산: 중위순회

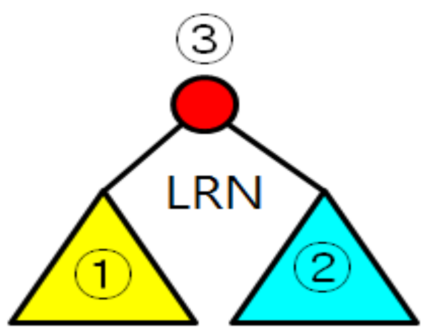
```python
    def inorder(self, n): # 중위순회
        if n != None:
            if n.left:
                self.inorder(n.left)
            print(str(n.item),' ', end='')
            if n.right:
                self.inorder(n.right)
```

왼쪽 서브트리 방문 후 노드 방문

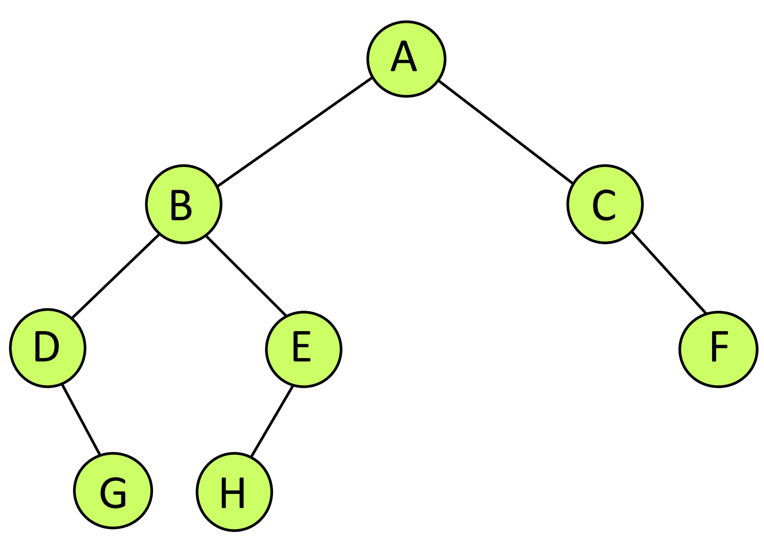


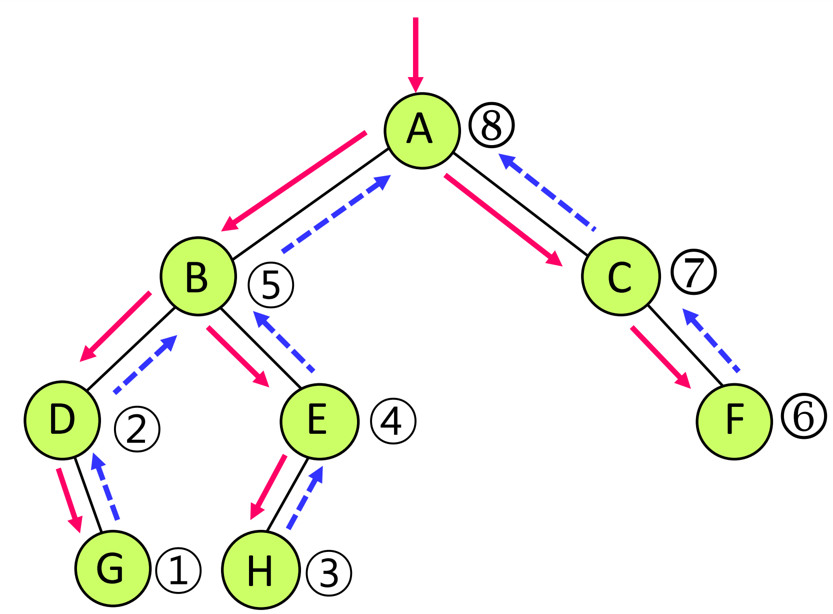
중위순회: 800 400 200 500 100 600 300 700

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='')
```

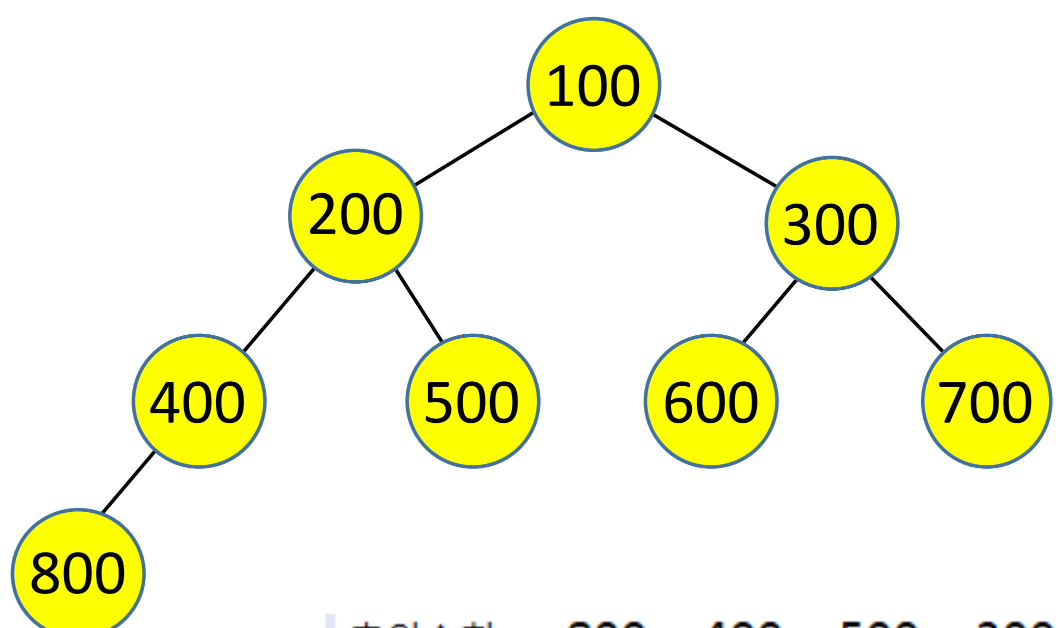




트리 연산: 후위순회

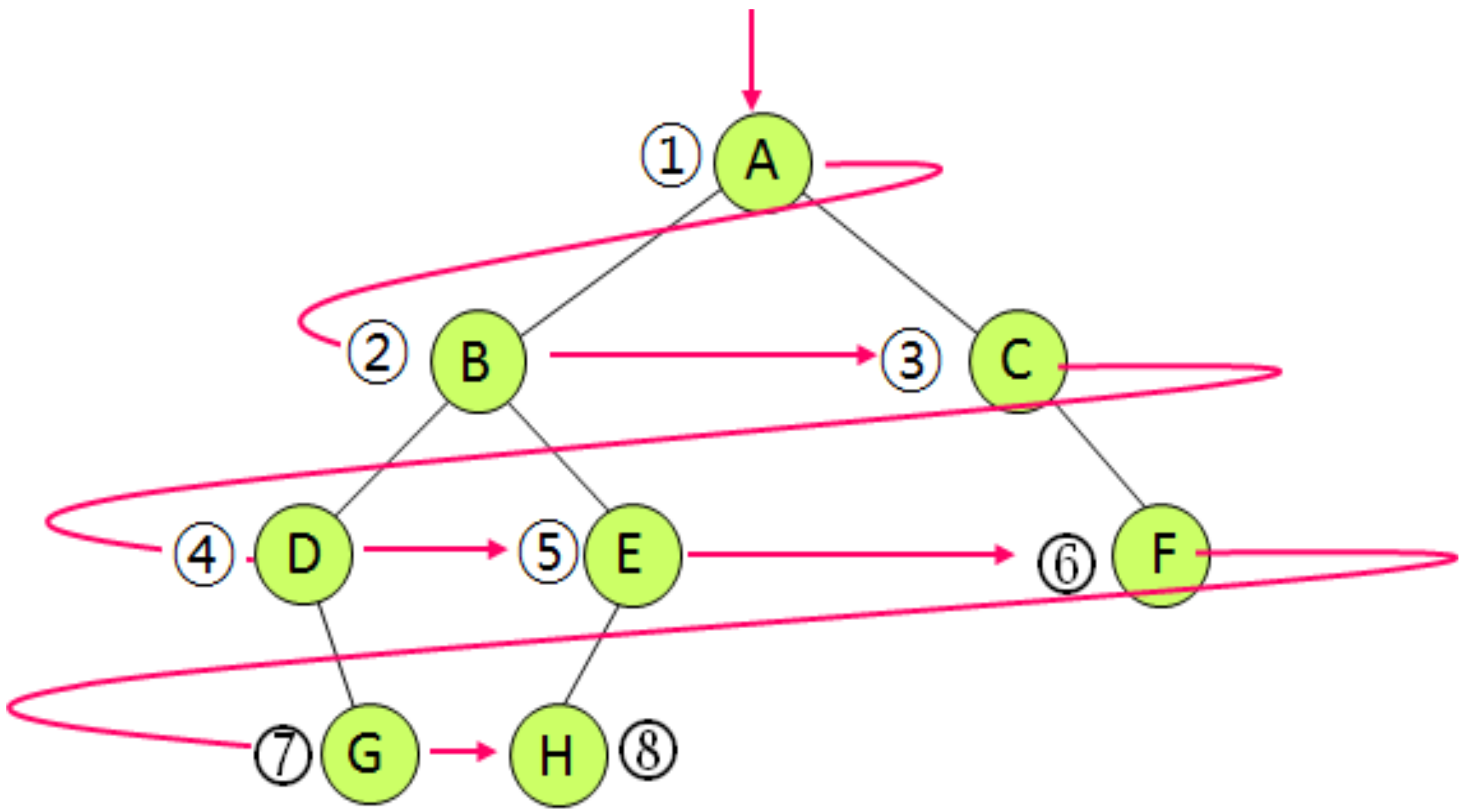
```python
    def postorder(self, n): # 후위순회
        if n != None:
            if n.left:
                self.postorder(n.left)
            if n.right:
                self.postorder(n.right)
            print(str(n.item),' ', end='')
```

왼쪽과 오른쪽 서브트리
모두 방문 후 노드 방문




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

트리연산: 레벨순회



트리 연산: 레벨순회

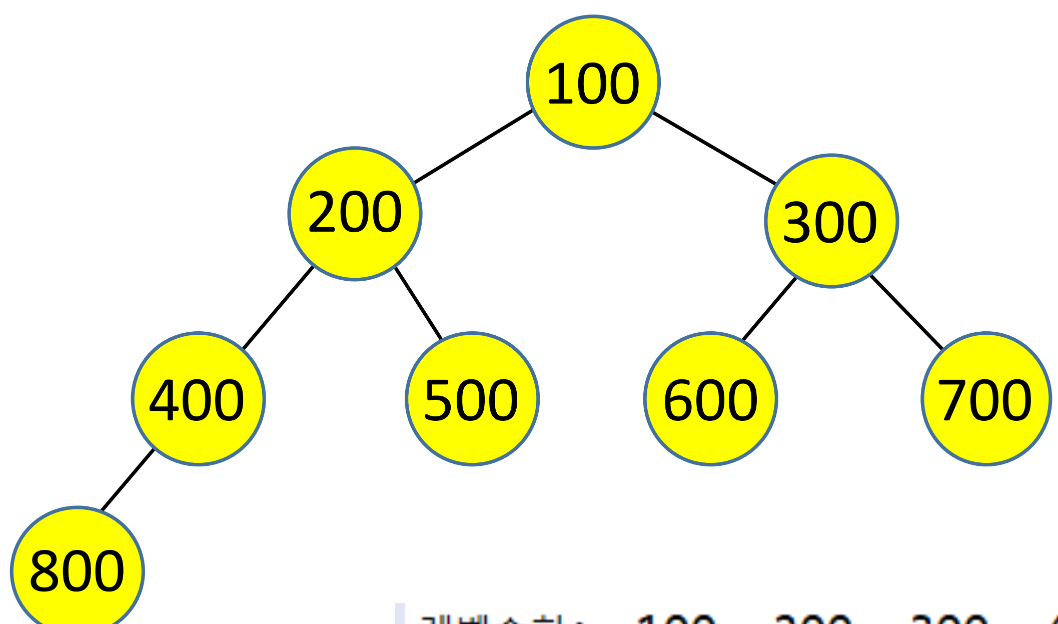
```python
    def levelorder(self, root): # 레벨순회
        q = []
        q.append(root)
        while len(q) != 0:
            t = q.pop(0)
            print(str(t.item), ' ', end='')
            if t.left != None:
                q.append(t.left)
            if t.right != None:
                q.append(t.right)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

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 수를 계산하는 것은 트리의 아래에서 위로 각 자식의 후손노드 수를 합하며 올라가는 과정을 통해 수행되며, 최종적으로 루트노드에서 총 합을 구함

- 트리의 높이도 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

- 2개의 이진트리를 비교하는 것은 다른 부분을 발견하는 즉시 비교 연산을 멈추기 위해 전위순회 방법을 사용aa

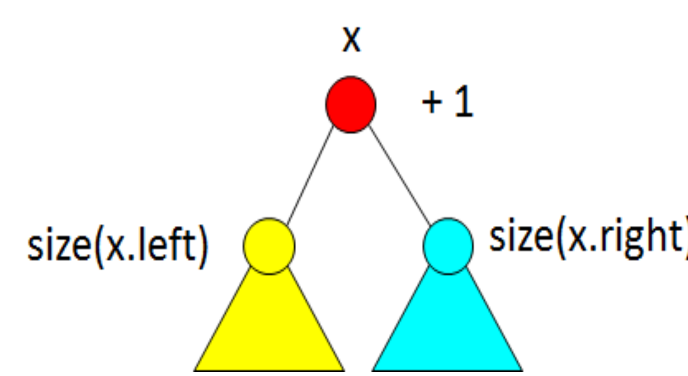
총 노드수 계산

- 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)
```

수행시간

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

기타 이진트리 연산

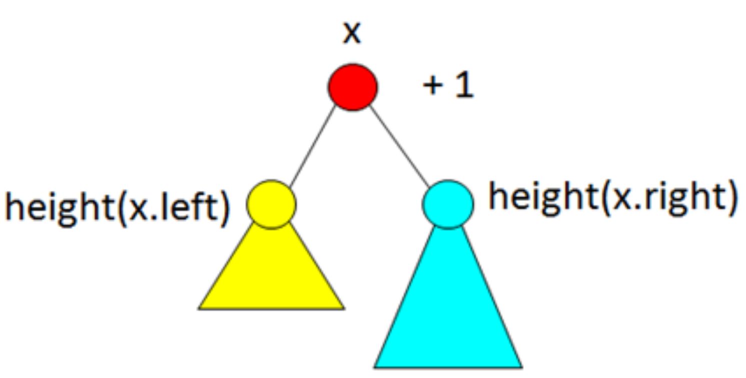
- 이진트리의 높이 계산
- isEqual(): 2개의 이진트리에 대한 동일성 검사

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

꿰매진 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

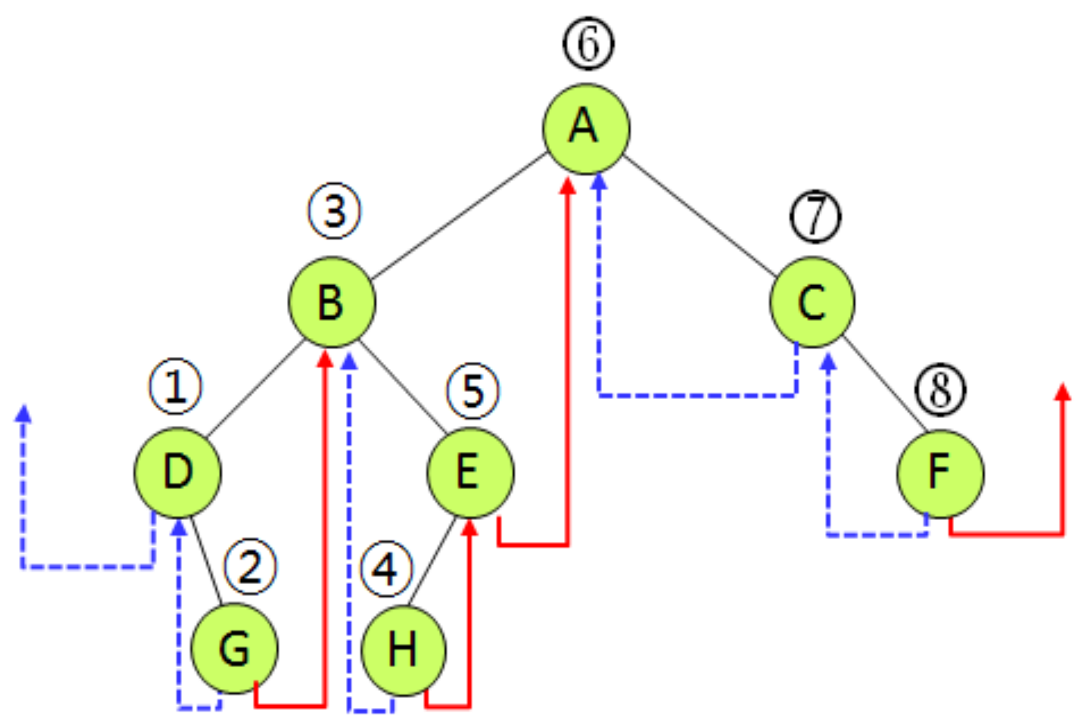
  - 스택 없이 이진트리의 연산을 구현하는 2 가지 방법

    [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 노들르 참조하게 하고, 각 노드의 왼쪽 non-레퍼런스를 직전 방문한 노드를 참조하게 한 이진트리이다.

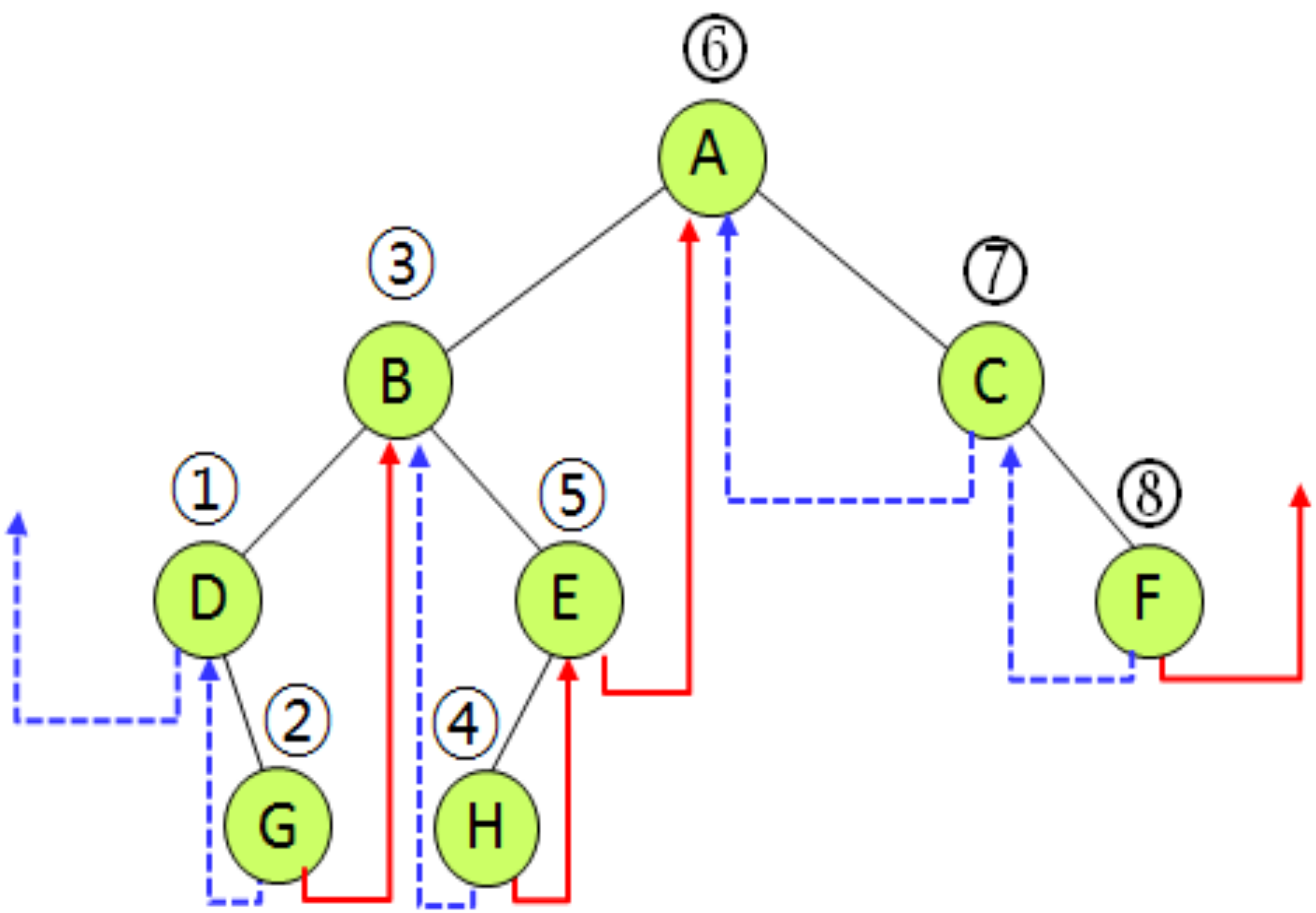


이진트리의 특성

- 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