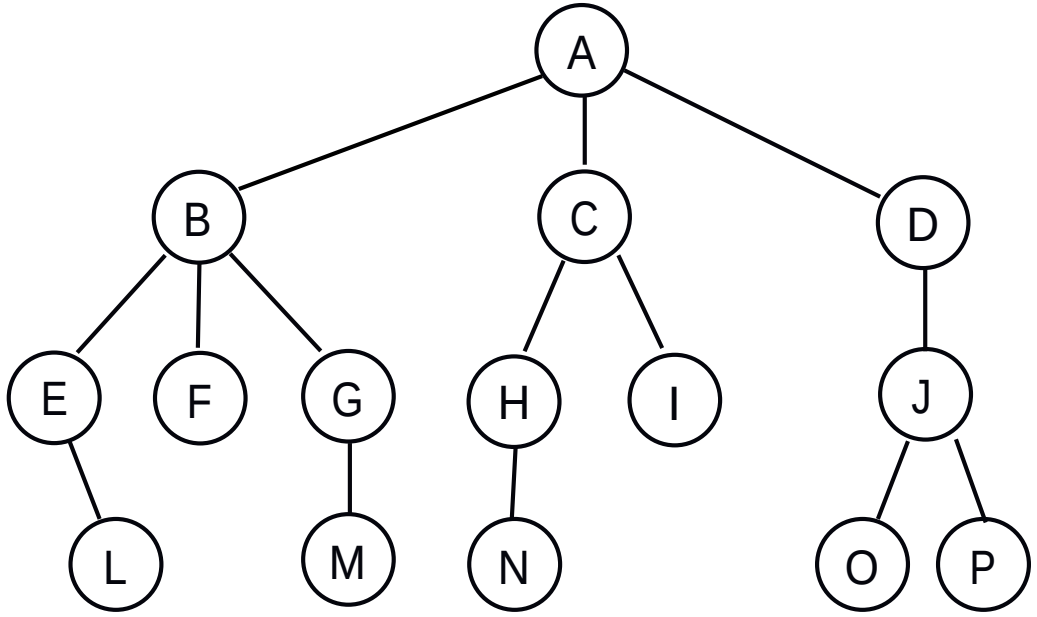
Tree (트리)

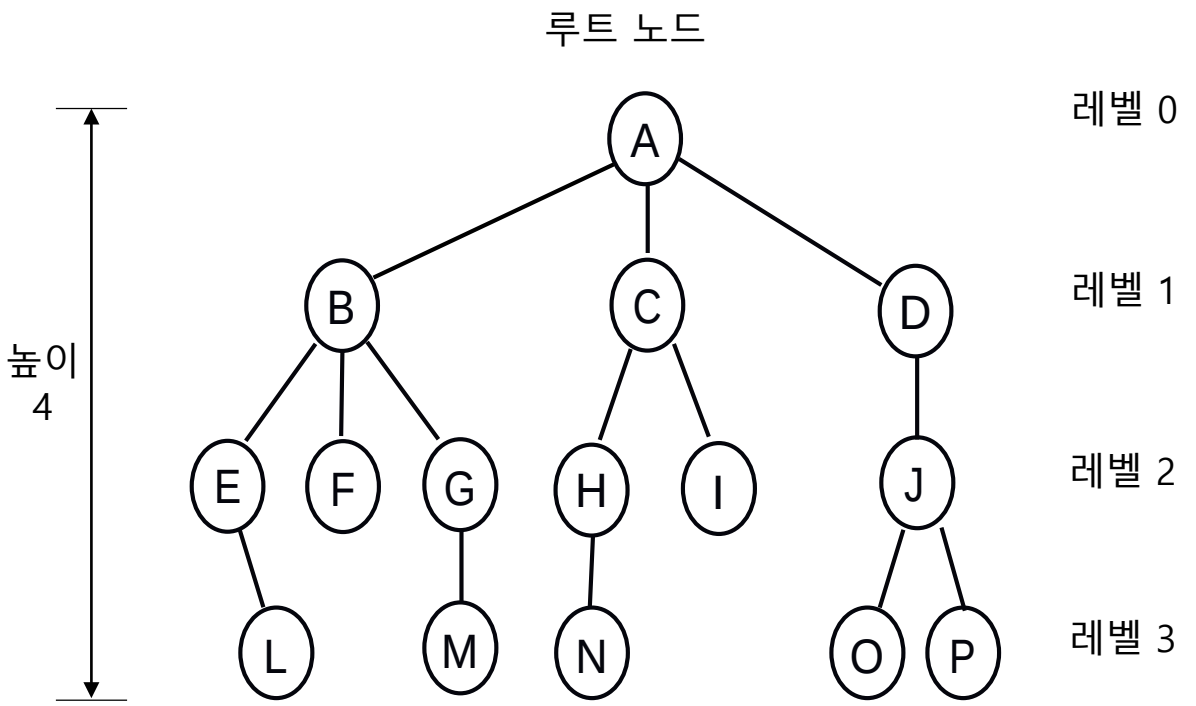
트리

- 연결 리스트, 스택, 큐 등은 모두 선형 구조
  - 구현 용이
  - 선형 탐색의 비효율성
- 트리
  - 계층적인 구조
  - 윈도우의 폴더 구조 (탐색기)
  - 조직도



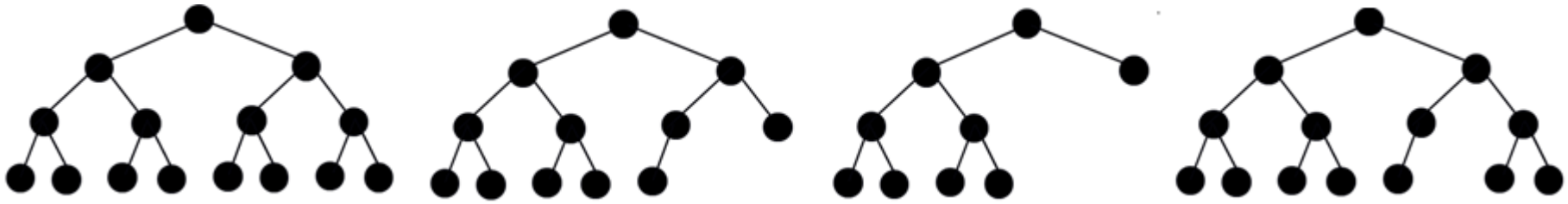
용어

- 루트(Root) – 트리의 최상위 노드
- 자식(Child) – 노드의 하위에 연결된 노드
- 부모(Parent) – 노드의 상위에 연결된 노드
- 차수(Degree) – 자식의 수
- Leaf – 자식이 없는 노드
- 레벨(Level) – 루트는 레벨 0, 아래 층으로 내려가며 레벨이 1씩 증가
  - 레벨은 깊이(Depth)와 동일
- 높이(Height) – 트리의 최대 레벨
- 키(Key) – 노드에 저장된 탐색에 사용되는 정보



이진 트리 (binary tree)

- 각 노드의 자식 수가 2 이하인 트리



(a) 포화이진트리 (b) 완전이진트리 (c) 불완전한 이진트리 (d) 불완전한 이진트리

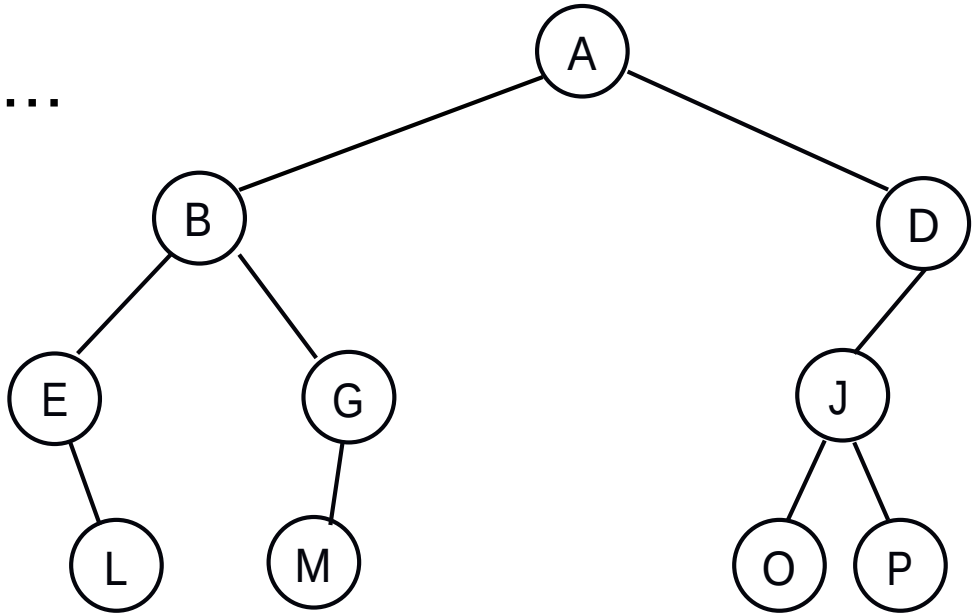
- 포화이진트리
  - 각 내부 노드(레벨이 높이보다 작은 노드)가 2개의 자식 노드를 가지는 트리
- 완전이진트리
  - 마지막 레벨을 제외한 트리가 포화이진트리이며, 마지막 레벨에는 노드들이 왼쪽부터 채워진 트리

이진트리 순회 (traversal)

- 전위순회 (preorder traversal)
- 중위순회 (inorder traversal)
- 후위순회 (postorder traversal)
- 레벨순회 (level-order traversal)

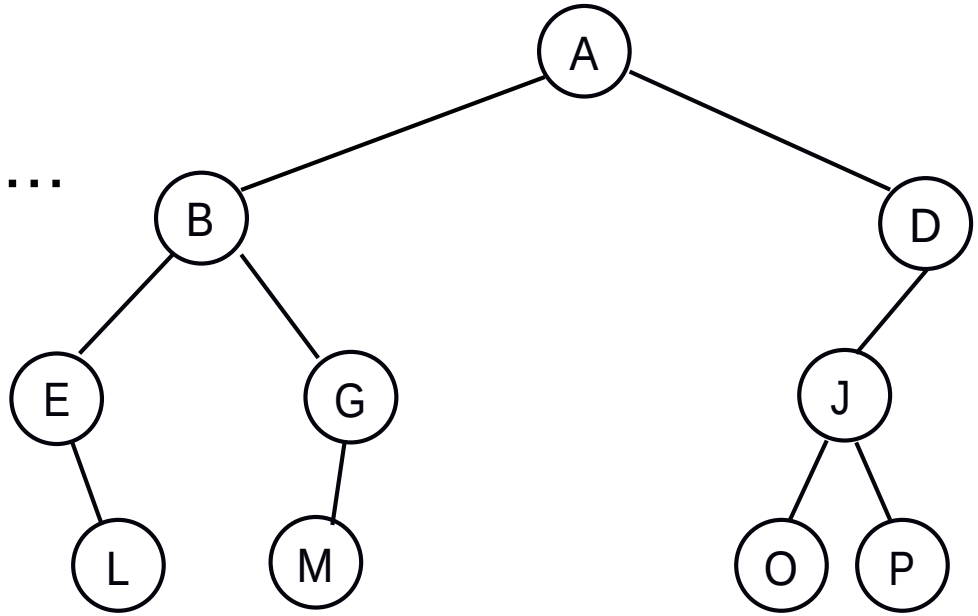
전위순회

- 노드 n에 도착하면 n을 먼저 탐색하고 n의 왼쪽 자식 노드 순회. 왼쪽 서브트리의 모든 노드를 순회한 후에는 n의 오른쪽 서브트리 순회
  - NLR(node, left, right) 순서
  - A -> B -> E -> L -> G -> M -> D …



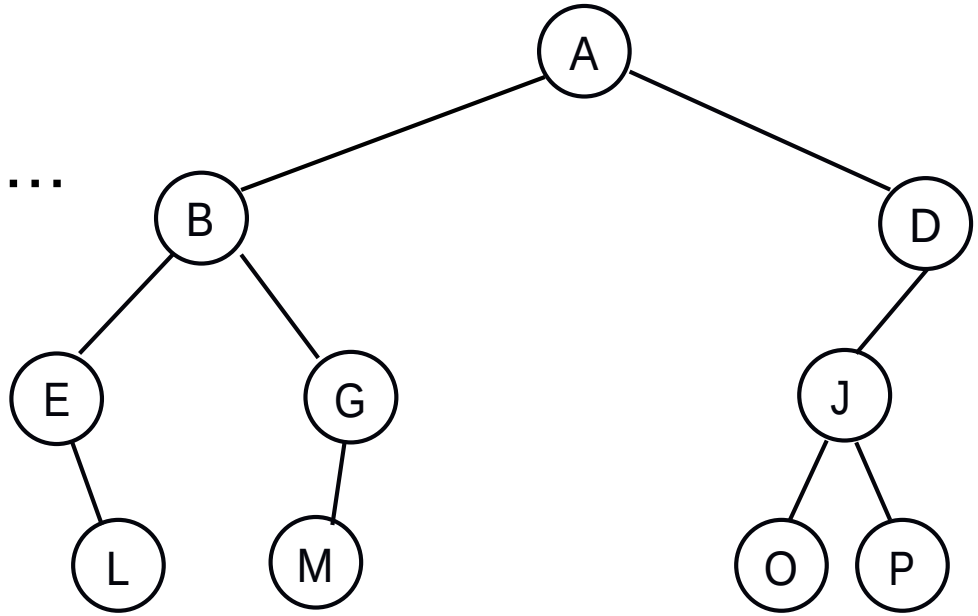
중위순회

- 노드 n에 도착하면 n의 탐색을 보류하고 먼저 n의 왼쪽 서브트리 순회. 왼쪽 서브트리의 모든 노드를 순회한 후에는 n을 탐색하고 이후 n의 오른쪽 서브트리 순회
  - LNR(left, node, right) 순서
  - E -> L -> B -> M -> G -> A -> D …



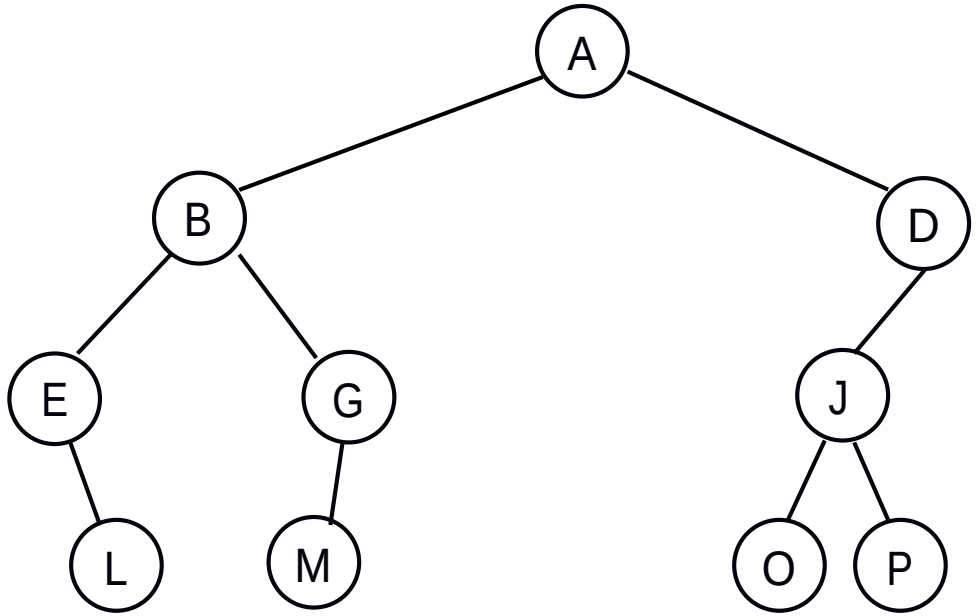
후위순회

- 노드 n에 도착하면 n의 탐색을 보류하고 먼저 n의 왼쪽 서브트리 순회. 왼쪽 서브트리의 모든 노드를 순회한 후에는 n의 오른쪽 서브트리 순회. 마지막으로 n 탐색
  - LRN(left, right, node) 순서
  - L -> E -> M -> G -> B -> O -> P …



레벨순회

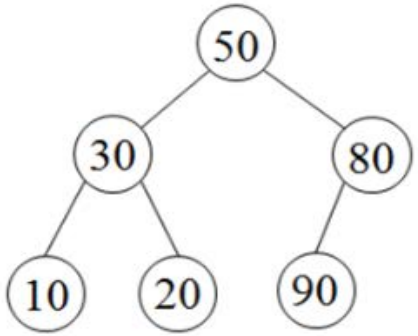
- 루트 노드부터 시작하여 좌에서 우 방향으로 노드 순회
  - A -> B -> D -> E -> G -> ···



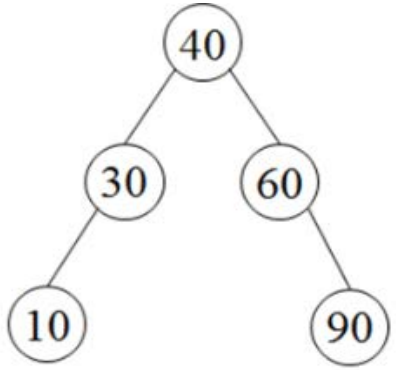
트리의 활용 — 이진탐색트리

이진탐색트리

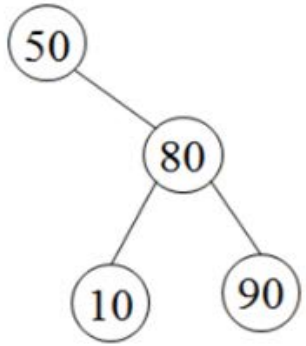
- 각 노드가 다음의 조건을 만족하는 이진트리
  - 노드 N의 키가 N의 왼쪽 서브트리의 키들보다 크고, N의 오른쪽 서브트리의 키들보다 작다.
- 예



X



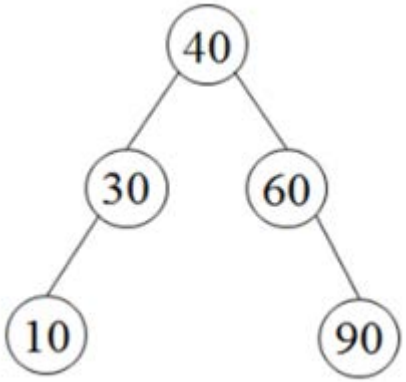
O



X

이진탐색트리 – 장점

- 탐색의 효율성
  - 최대값: 오른쪽 child를 계속 따라가면 leaf 노드가 최대값
  - 최소값: 왼쪽 child를 계속 따라가면 leaf 노드가 최소값
  - 특정값:
    - 루트노드부터 시작하여 노드의 키와 탐색 키를 비교
    - 탐색 키가 작으면 왼쪽 child에 대해 반복
    - 탐색 키가 크면 오른쪽 child에 대해 반복
  - 시간복잡도: O(h): h는 트리의 높이



이진탐색트리 – 구현 – 1단계 (노드 클래스)

- 노드 클래스의 정의
  - 이진 트리이므로 2개의 child node(left & right)를 갖는다.
  - 노드는 key와 value를 저장

```
class BSTNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
```

이진탐색트리 – 구현 – 2단계 (트리 클래스)

- 트리 클래스의 정의
  - 하나의 root node를 갖는다.
  - method: insert(), search(), get_min(), get_max()

```
class BST:
    def __init__(self):
        self.root = None
    def insert(self, key, value):
    def search(self, key):
    def get_min(self):
    def get_max(self):
```

이진탐색트리 – 구현 (insert())

- insert()의 구현
  - 루트노드가 None이면 루트 노트로 삽입하고 종료
  - 루트노드부터 시작하여 노드의 키와 탐색 키를 비교
    - 탐색 키가 작으면 왼쪽 child에 대해 반복
    - 탐색 키가 크면 오른쪽 child에 대해 반복
    - 키가 동일하면 충돌 또는 update 케이스. 실패로 처리할 수도 있고, value만 수정할 수도 있다.
  - 탐색이 실패로 끝난 위치에 (child가 존재하지 않음), left 또는 right child 노드로 삽입

이진탐색트리 – 구현 (insert())

```
def insert(self, key, value):
        if (self.root == None):
            node = BSTNode(key, value)
            self.root = node
            return True
```

이진탐색트리 – 구현 (insert())

```
target = self.root
while (target):
    if (target.key == key):
        # 중복.. value를 수정하도록 구현할 수도 있음
        return False
```

이진탐색트리 – 구현 (insert())

```
elif (target.key > key): # 왼쪽으로 진행
    if (target.left == None):
        # 삽입
        node = BSTNode(key, value)
        target.left = node
        return True
    target = target.left
```

이진탐색트리 – 구현 (insert())

```
elif (target.key < key): # 오른쪽으로 진행
    if (target.right == None):
        # 삽입
        node = BSTNode(key, value)
        target.right = node
        return True
    target = target.right
```

이진탐색트리 – 구현 (print())

- print()의 구현
  - 중위순회 방식으로 구현 (LNR)
    - 오름차순으로 정렬되어 출력
    - 재귀 방식으로 구현
      - print(left subtree)
      - node의 key 출력
      - print(right subtree)

이진탐색트리 – 구현 (print())

```
def print(self):
        self.doPrint(self, self.root)

def doPrint(self, node):
        if (node != None) :
            self.doPrint(node.left)
            print(node.key, endl=' ')
            self.doPrint(node.right)
```

이진탐색트리 – test1

```
myBST = BST()
myBST.insert(5, "a")
myBST.insert(7, "b")
myBST.insert(3, "c")
myBST.insert(1, "d")
myBST.insert(9, "e")
myBST.insert(15, "f")
myBST.print()
myBST.insert(8, "g")
myBST.print()
```

이진탐색트리 – 구현 (get_min())

- get_min()의 구현
  - 가장 왼쪽 leaf node : 반복해서 왼쪽 child를 따라가면 됨

```
def get_min(self):
    target = self.root
    while (target and target.left):
        target = target.left

    return target
```

이진탐색트리 – 구현 (get_max())

- get_max()의 구현
  - 가장 오른쪽 leaf node : 반복해서 오른쪽 child를 따라가면 됨

```
def get_max(self):
    target = self.root
    while (target and target.right):
        target = target.right

    return target
```

이진탐색트리 – test2

```
myBST = BST()
myBST.insert(5, "a")
myBST.insert(7, "b")
myBST.insert(3, "c")
myBST.insert(1, "d")
myBST.insert(9, "e")
myBST.insert(15, "f")
myBST.print()
print(myBST.get_max().key)
print(myBST.get_min().key)
```

이진탐색트리 – 구현 (search())

- search()의 구현
  - 루트노드부터 시작하여 노드의 키와 탐색 키를 비교
  - 탐색 키가 작으면 왼쪽 child에 대해 반복
  - 탐색 키가 크면 오른쪽 child에 대해 반복
  - 없는 경우도 있을 수 있음

이진탐색트리 – 구현 (search())

```
def search(self, key):
        target = self.root
        while (target):
            if (target.key == key):
                return target
            elif (target.key > key):
                target = target.left
            else:
                target = target.right
        return None
```

이진탐색트리 – test3

```
myBST = BST()
myBST.insert(5, "a")
myBST.insert(7, "b")
myBST.insert(3, "c")
myBST.insert(1, "d")
myBST.insert(9, "e")
myBST.insert(15, "f")
myBST.print()
print(myBST.search(9).value)
```

이진탐색트리 – 심화 구현 (print_level())

- print_level()
  - level 출력하는 함수
  - 루트에서 시작 (타겟)
  - 타겟 출력
  - 타겟의 자식 노드들 모두 출력
  - 타겟 수정하여 반복..
  - level 별로 출력되어야 함 (자식이 부모보다 먼저 출력되면 안 됨) => 큐 활용

이진탐색트리 – 심화 구현 (print_level())

- print_level()
    1. 루트를 큐에 삽입
    2. 큐가 비어 있지 않은 동안 반복
        1. 큐에서 노드 dequeue
        2. 노드 출력
        3. 노드의 자식 노드들을 큐에 삽입

이진탐색트리 – 심화 구현 (delete())

- delete()의 구현
  - 타겟이 leaf node인 경우
    - parent의 child link 수정
  - 타겟이 하나의 child를 갖는 경우
    - 타겟의 자리에 child와 parent를 연결 (parent 입장에서 손자가 자식으로 바뀌는 격)
  - 타겟이 두 개의 child를 갖는 경우
    - 타겟의 자리를 다른 후계자로 대체
    - 후계자는 타겟의 왼쪽 서브트리 중 키가 가장 큰 노드 또는 오른쪽 서브트리 중 키가 가장 작은 노드
    - 후계자 노드의 원래 위치는 삭제 (후계자의 원 parent의 child link 수정)

Appendix

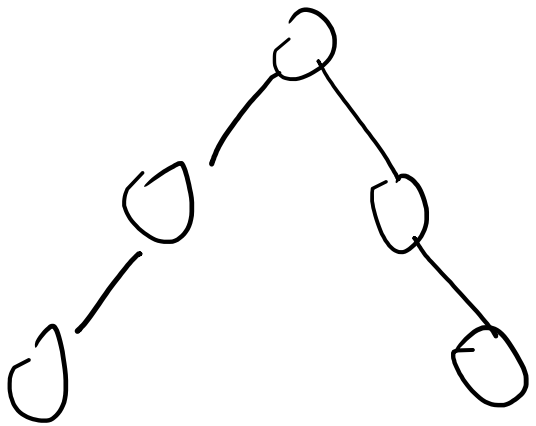
AVL 트리

- 이진탐색트리는 균형이 무너진 skewed tree가 될 수 있음.
  - 탐색 시간 증가..
- 이 문제를 해결하기 위해 나온 것이 AVL 트리
  - 균형이 깨지면 회전 연산을 통해 균형을 유지.
  - 모든 노드 n이 다음의 조건을 만족하는 이진탐색트리
    - n의 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 차이가 1 이하임
  - 삽입이나 삭제로 인해 높이 차이가 2가 되면 회전 연산 수행하여 균형 유지

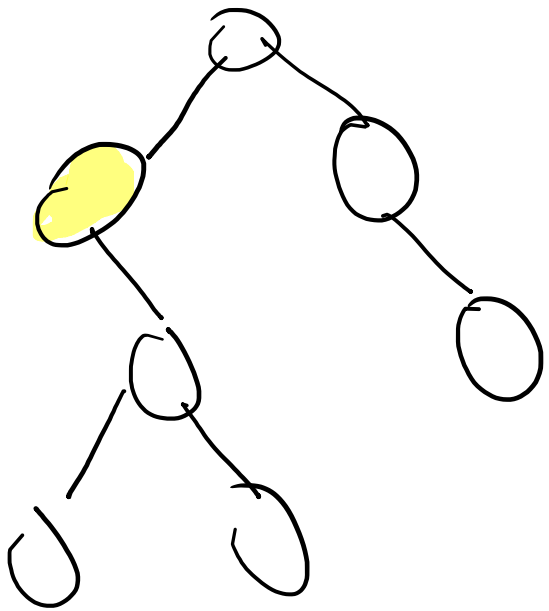
AVL 트리

- 어느 트리가 AVL 트리인가?

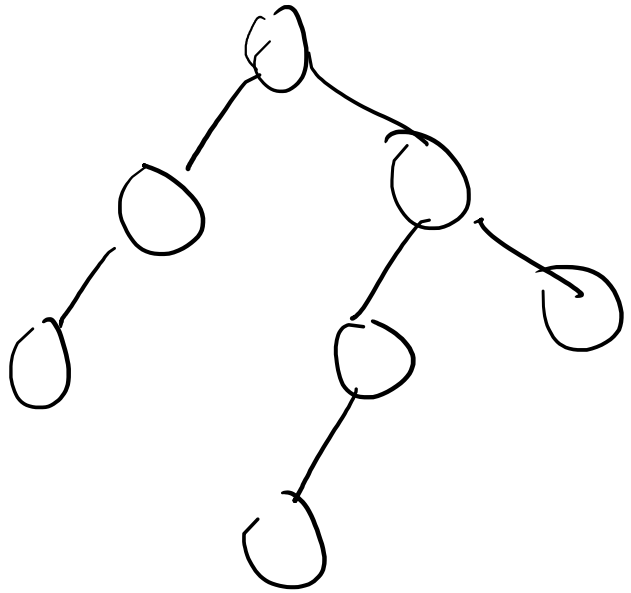
(a)



(b)



(c)

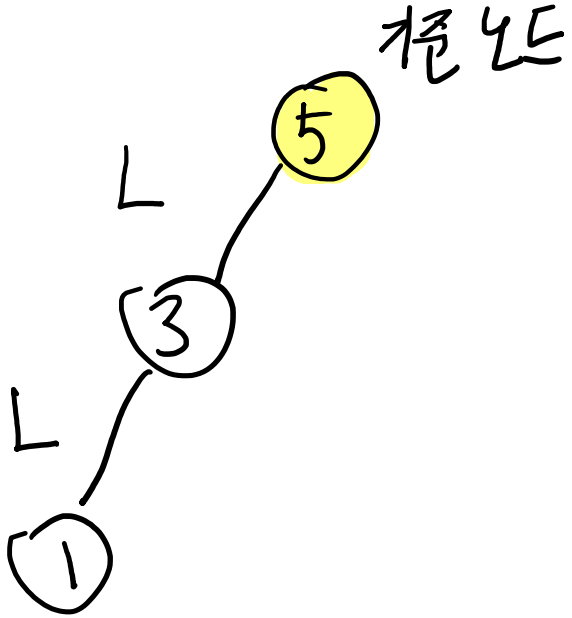


AVL 트리 – 회전 연산

- 회전 종류
  - LL 회전
  - RR 회전
  - LR 회전
  - RL 회전

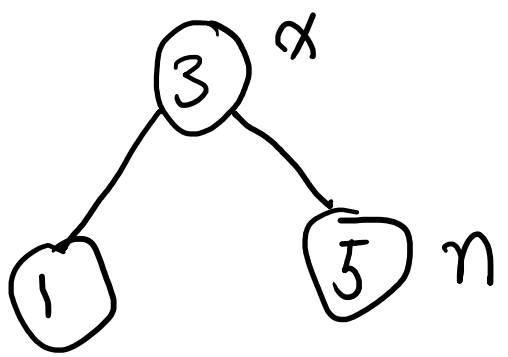
AVL 트리 – 회전 연산

- LL 회전



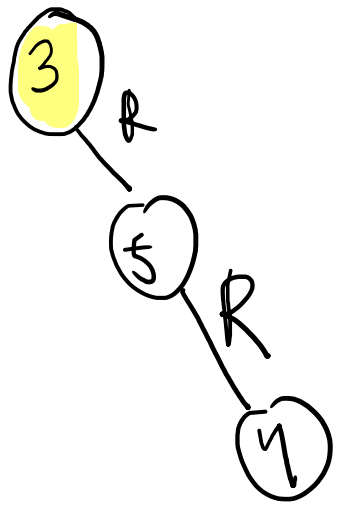
⇒

- 현 노드(∧)의 left child(x라고 하면)를 현 노드 자리로
- x의 right child는 n의 left child로 수정
- n이 x의 right child가 됨.



AVL 트리 – 회전 연산

- RR 회전



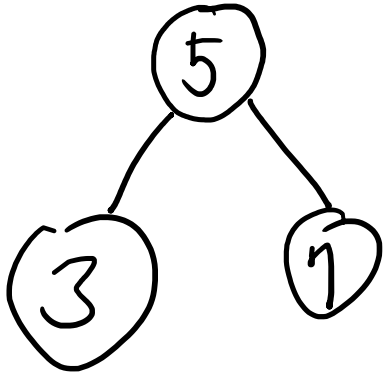
⇒

n x

① target node의 right child가 target node의 왼쪽으로

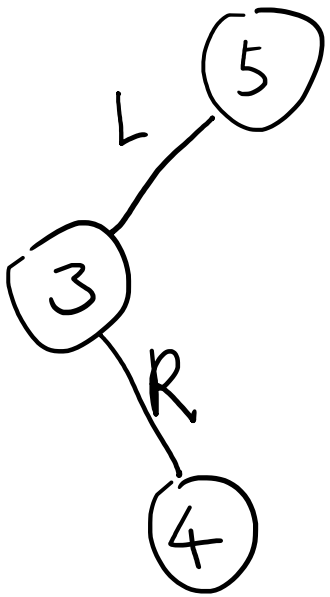
② n은 x의 left child가 됨.

③ x의 left child는 n의 right child로 수정.

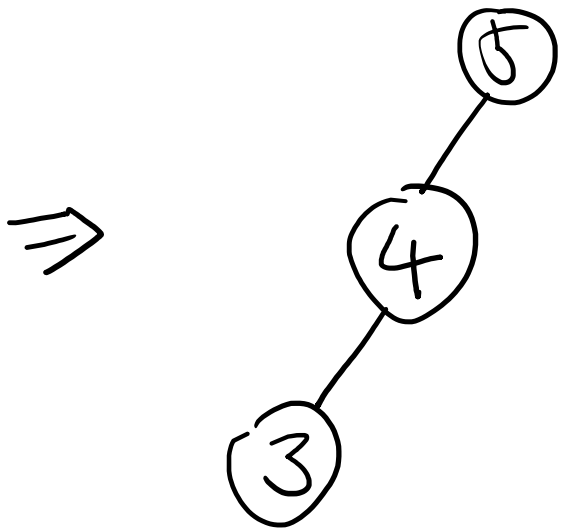


AVL 트리 – 회전 연산

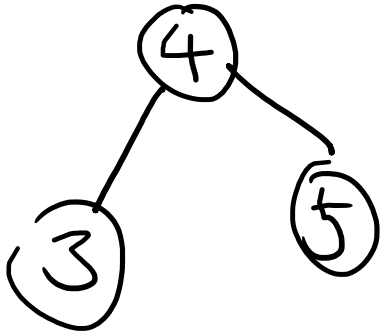
- LR 회전



① 먼저 target의 left child를 기준으로 RR 회전 수행

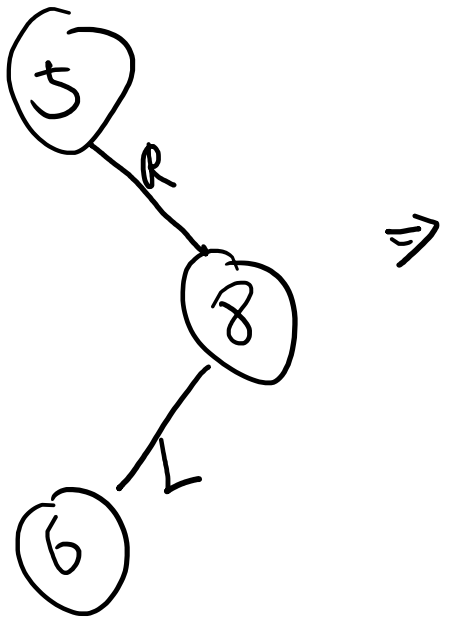


② target 기준으로 LL 회전 수행



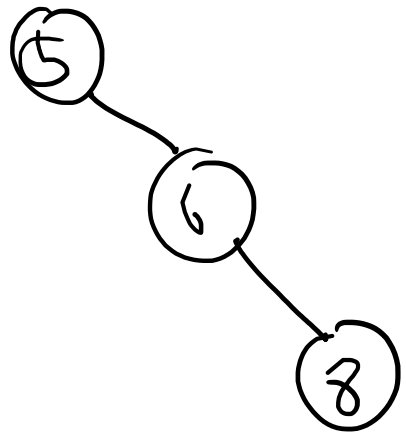
AVL 트리 – 회전 연산

- RL 회전



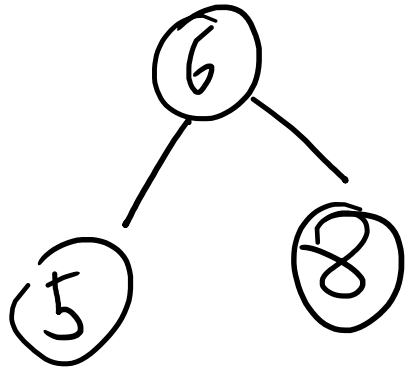
⇒

① target의 right child 기준으로 LL회전 수행.



⇒

② target 기준으로 RR 회전 수행.



AVL트리 – 구현 – 1단계 (노드 클래스)

- 노드 클래스의 정의
  - 이진 트리이므로 2개의 child node(left & right)를 갖는다.
  - 각 노드는 자신의 height를 저장 (균형이 깨졌는지 여부 확인을 위해..)
  - 노드는 key와 value를 저장

```
class AVLNode :
    def __init__(self, key, value, height, left=None, right=None):
        self.key = key
        self.value = value
        self.height = height
        self.left = left
        self.right = right
```

AVL트리 – 구현 – 2단계 (트리 클래스)

- 트리 클래스의 정의
  - 하나의 root node를 갖는다.
  - method: getHeight(), checkBalance(), doBalance(), rotateRight(), rotateLeft(), insert(), insertItem()

```
class AVL :
    def __init__(self):
        self.root = None

    def getHeight(self, n) :
        if (n == None) :
            return 0
        return n.height
```

AVL트리 – 구현 (rotateRight())

- rotateRight()의 구현

```
def rotateRight(self, n) :
    x = n.left
    n.left = x.right
    x.right = n
    n.height = max(self.getHeight(n.left), self.getHeight(n.right)) + 1
    x.height = max(self.getHeight(x.left), self.getHeight(x.right)) + 1
    return x
```

AVL트리 – 구현 (rotateLeft())

- rotateLeft()의 구현

```
def rotateLeft(self, n) :
    x = n.right
    n.right = x.left
    x.left = n
    n.height = max(self.getHeight(n.left), self.getHeight(n.right)) + 1
    x.height = max(self.getHeight(x.left), self.getHeight(x.right)) + 1
    return x
```

AVL트리 – 구현 (balance 관련 메서드)

- checkBalance(), doBalance()의 구현

```
def checkBalance(self, n) :
    return self.getHeight(n.left) - self.getHeight(n.right)

def doBalance(self, n) :
    if (self.checkBalance(n) > 1) :
        if (self.checkBalance(n.left) < 0) : # LR 회전
            n.left = self.rotateLeft(n.left)
        n = self.rotateRight(n)
    elif (self.checkBalance(n) < -1) :
        if (self.checkBalance(n.right) > 0) : # RL 회전
            n.right = self.rotateRight(n.right)
        n = self.rotateLeft(n)
    return n
```

AVL트리 – 구현 (insert 관련 메서드)

- insert(), insertItem()의 구현

```
def insert(self, key, value) :
    self.root = self.insertItem(self.root, key, value)

def insertItem(self, n, key, value):
    if n == None:
        return AVLNode(key, value, 1)

    if n.key > key:
        n.left = self.insertItem(n.left, key, value)
    elif n.key < key :
        n.right = self.insertItem(n.right, key, value)
    else:
        n.value = value
        return n

    n.height = max(self.getHeight(n.left), self.getHeight(n.right)) + 1
    return self.doBalance(n)
```

More Appendix

- 2-3 트리
- RB 트리
- B 트리