# 6장 인덱스 구조

**Part A**

# ❖ 인덱스(index)

- 특징
  - 파일의 레코드들에 대한 효율적 접근
  - <레코드 키 값, 레코드 주소(포인터)> 쌍
- 종류
  - 키에 따른 인덱스
    - 기본(primary) 인덱스 : 기본 키를 포함한 인덱스
    - 보조(secondary) 인덱스 : 기본 인덱스 이외의 인덱스
  - 파일 조직에 따른 인덱스
    - 집중(clustered) 인덱스 : 레코드의 물리적 순서가 어떤 인덱스의 엔트리 순서와 동일하게 유지하도록 구성된 인덱스 (단 하나의 인덱스만 가능)
    - 비집중(unclustered index) 인덱스 : 집중 형태가 아닌 인덱스 (여러 인덱스가 가능)
  - 데이터 범위에 따른 인덱스
    - 밀집(dense) 인덱스 : 데이터 레코드 하나에 대해 하나의 인덱스 엔트리가 만들어지는 인덱스. 역인덱스(inverted index)
    - 희소(sparse) 인덱스 : 데이타 파일의 레코드 그룹 또는 데이터 블록에 하나의 엔트리가 만들어지는 인덱스


L-B 258
Ncag STOLPE, Per
Värmlands natur och värmländskt arbetsliv.
Föredrag.
1934, 10.1 (Karlstad).
Stolpe, Per.
6'45
L-B 21000 - 23
L-B 1 - 999

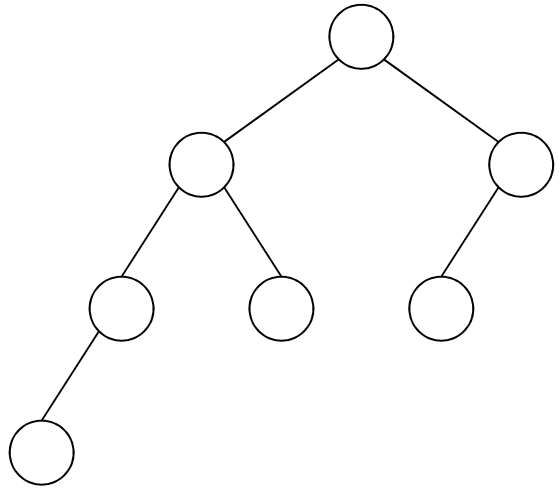
# ❖ 이원 탐색 트리 (binary search tree)

- 다음 트리의 차이점
  - 이진 트리 (binary tree)
  - 이원 탐색 트리 (binary search tree, BST)

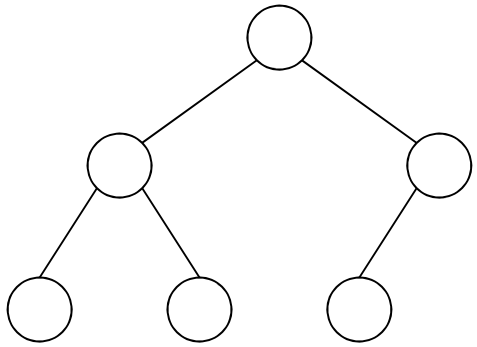
# ▶ 이진 트리

- 이진 트리**(Binary tree)**
  - 유한한 수의 노드를 가진 트리
  - 왼쪽 서브트리와 오른쪽 서브트리로 구성

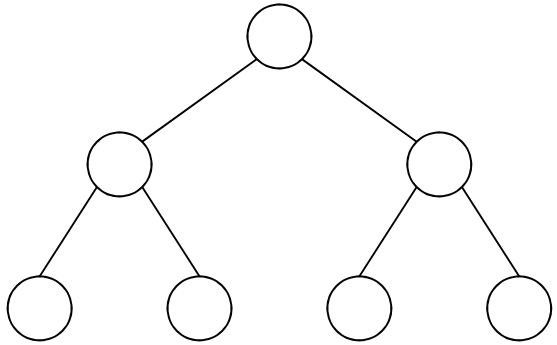
- 이진 트리의 특성
  - 레벨 i (1≤i) 에서의 최대 노드수: $2^{i-1}$
  - 높이 h (1≤h)인 이진 트리의 최대 노드수: $2^{h}-1$
  - 단말노드수 $n_0$, 차수가 2인 노드수 $n_2$에 대해: $n_0 = n_2+1$



◆ 완전 이진 트리 **(complete binary tree)**



◆ 포화 이진 트리 **(full binary tree)**



◆ 균형 트리 **(balanced tree)**

– 모든 리프 노드는 같은 레벨에 존재

# ▶ 이원 탐색 트리

◆ 이원 탐색 트리 **(Binary search tree, BST)**

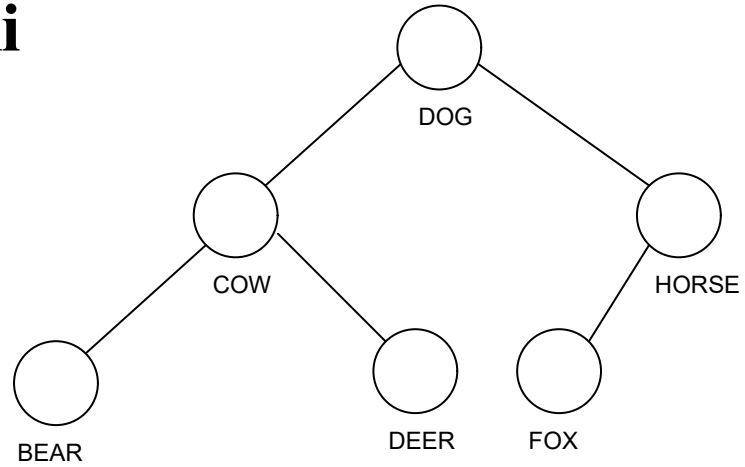
– 이진 트리

– 각 노드 Ni : 레코드 키 Ki와 이 키가 가지고 있는 레코드 포인터를 포함

◆ 노드 **Ni = (Ki, Ai)**

**① Nj ∈ RT(Ni) → Ki < Kj**

**② Nj ∈ LT(Ni) → Kj < Ki**




★ 키 값 $K_i$ : ($K_i$, $A_i$)를 의미, $A_i$ : 데이타 레코드의 주소

◆ **BST의 정의**

① 모든 노드 Ni는 상이한 키 값을 갖는다.

$Ni = (Ki, Ai)$

② 루트 노드 $N_i$의 왼쪽 서브트리($LT(N_i)$)에 있는 모든 노드의 키 값은 루트 노드의 키 값보다 작다.

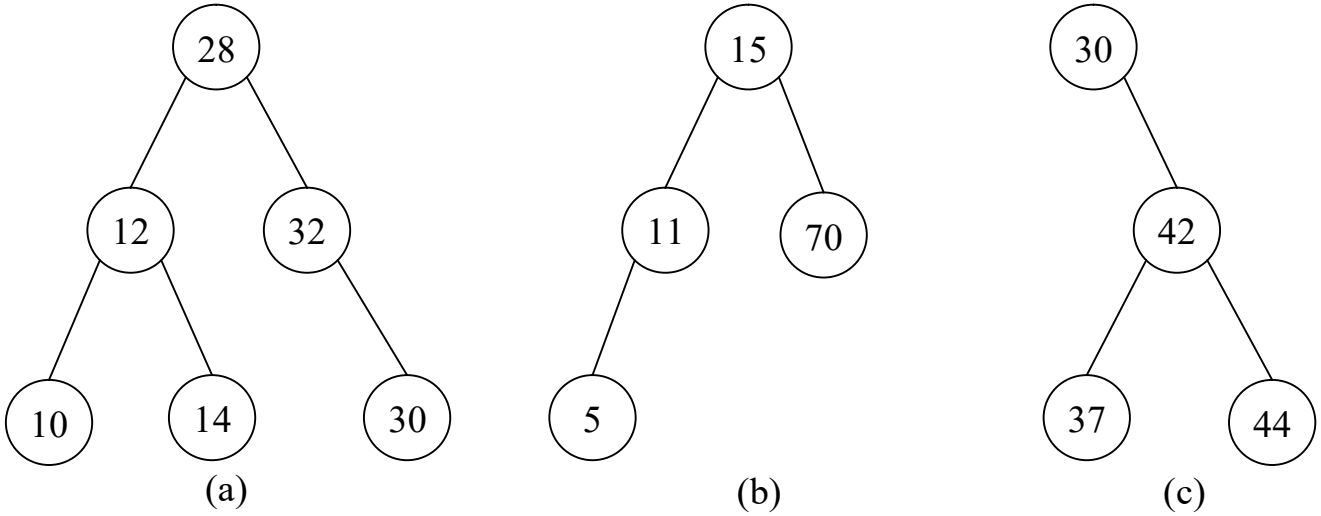
$Nj \in LT(Ni) \rightarrow Kj < Ki$

③ 루트 노드 $N_i$의 오른쪽 서브트리($RT(N_i)$)에 있는 모든 노드의 키 값은 루트 노드의 키 값보다 크다.

$Nj \in RT(Ni) \rightarrow Ki < Kj$

④ 왼쪽 서브트리와 오른쪽 서브트리는 모두 BST.

- 이진 트리와 이원 탐색 트리
  - 그림 (a): 이진 트리
    - 이원 탐색 트리의 특성 중 4번 항 위반
  - 그림 (b), (c): 이원 탐색 트리



# ▶ 이원 탐색 트리에서의 연산: 검색

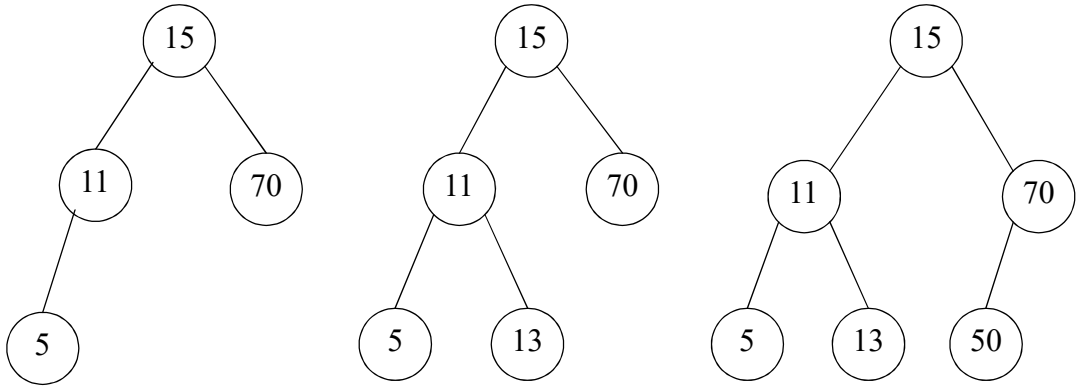
- 키값이 **K**인 노드 **Ni** 의 검색 과정
  1. 트리가 공백 : 검색은 노드를 찾지 못하고 실패
  2. $K=K_i$ : 노드 $N_i$가 원하는 노드
  3. $K<K_i$ : $N_i$의 왼쪽 서브트리를 검색
  4. $K>K_i$ : $N_i$의 오른쪽 서브트리를 검색

```
searchBST(B, s_key)          // B는 이원 탐색 트리, s_key는 검색 키 값
    p ← B;
    if (p = null) then     // 공백 이진 트리로 실패
        return null;
    if (p.key = s_key) then        // 검색 성공
        return p;
    if (p.key < s_key) then       //  오른쪽 서브트리 검색
        return searchBST(p.right, s_key);
    else
        return searchBST(p.left, s_key);    // 왼쪽 서브트리 검색
end searchBST()
```

# ▶ 이원 탐색 트리에서의 연산: 삽입

- **이원 탐색 트리에 키 값 K인 노드를 삽입**
  1. 트리가 공백 : K를 루트 노드로 삽입
  2. $K=K_i$ : 트리에 값은 키 값이 존재하므로 삽입을 거부
  3. $K<K_i$ : $N_i$의 왼쪽 서브트리로 이동하여 계속 탐색
  4. $K>K_i$ : $N_i$의 오른쪽 서브트리로 이동하여 계속 탐색

- **삽입 예 : 키 값 13, 50 삽입**




(a) 이원 탐색 트리 (b) 키값 13을 삽입 (c) 키값 50을 삽입

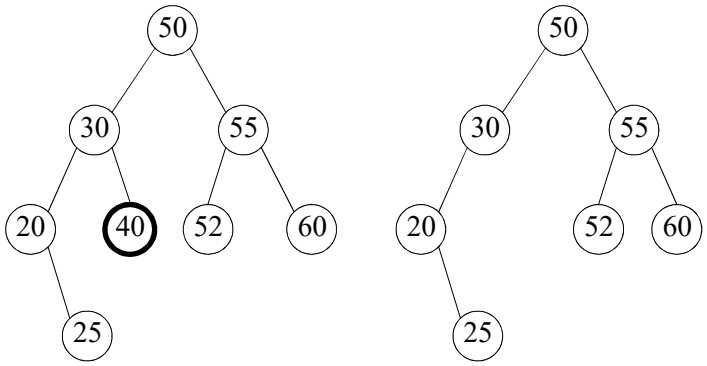
```
insertBST(B, new_key)   // B는 이원 탐색 트리, new_key는 삽입할 키 값
    p ← B;
    while (p ≠ null) do {  // 삽입하려는 키 값을 가진 노드가 이미 있는지 검사
       if (new_key = p.key) then return;
       q ← p;     // q는 p의 부모 노드를 지시
       if (new_key < p.key) then p ← p.left;
       else p ← p.right;
    }                                  // q ( 탐색이 실패로 종료하게 된 노드)를 찾아냄.
    newNode ← getNode();    // 삽입할 노드를 만듦
    newNode.key ← new_key;
    newNode.right ← null;
    newNode.left ← null;

    if (B = null) then B ← newNode;      // 공백 이원 탐색 트리인 경우
    else if (new_key < q.key) then
        q.left ← newNode;
    else
        q.right ← newNode;
    return;
end insertBST()
```

# ▶ 이원 탐색 트리에서의 연산: 삭제

- 삭제 노드가 가진 자식 수에 따른 삭제 연산
  1. 자식이 없는 리프 노드(차수가 0인 노드)의 삭제
     - 단순히 그 노드를 삭제
  2. 자식이 하나인 노드(차수가 1인 노드)의 삭제
     - 삭제되는 노드 자리에 자식 노드를 위치
  3. 자식이 둘인 노드(차수가 2인 노드)의 삭제
     - 삭제되는 노드 자리에 왼쪽 서브트리에서 제일 큰 키 값 또는 오른쪽 서브트리에서 제일 작은 키 값으로 대체 선택 (두 경우 중 높이가 높은 서브트리를 선택하는 것이 유리, 즉 균형 트리 유지)
     - 해당 서브트리에서 대체 노드를 삭제

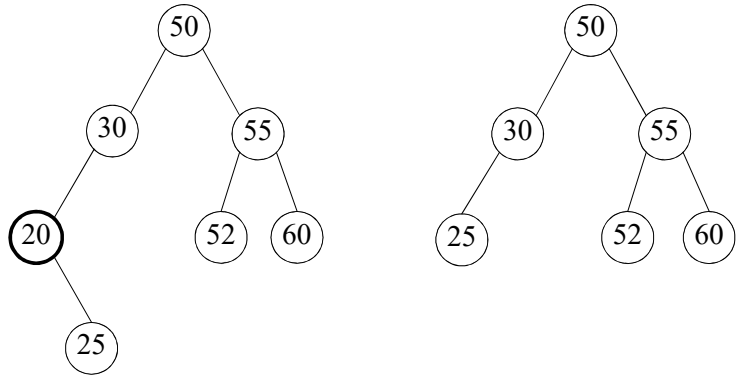
- **Trick: 삭제 표시로 삭제**
  - 물리적으로 즉각 삭제하지 않고, 삭제 표시만 해둠
  - 검색, 삽입 시 : 삭제된 노드의 키 값 이용
  - 정상적 키 값이 아니므로 특별히 취급

◆ 삭제 예 : 키 값 **40, 20, 50** 삭제



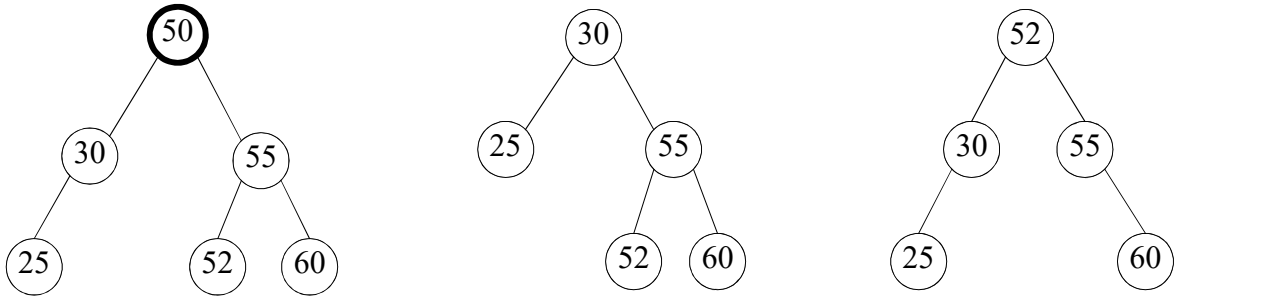

(a) 삭제전 (b) 삭제후

이원 탐색 트리에서 리프 노드(40)의 삭제




(a) 삭제전 (b) 삭제후

이원 탐색 트리에서 자식이 하나인 노드(20)의 삭제



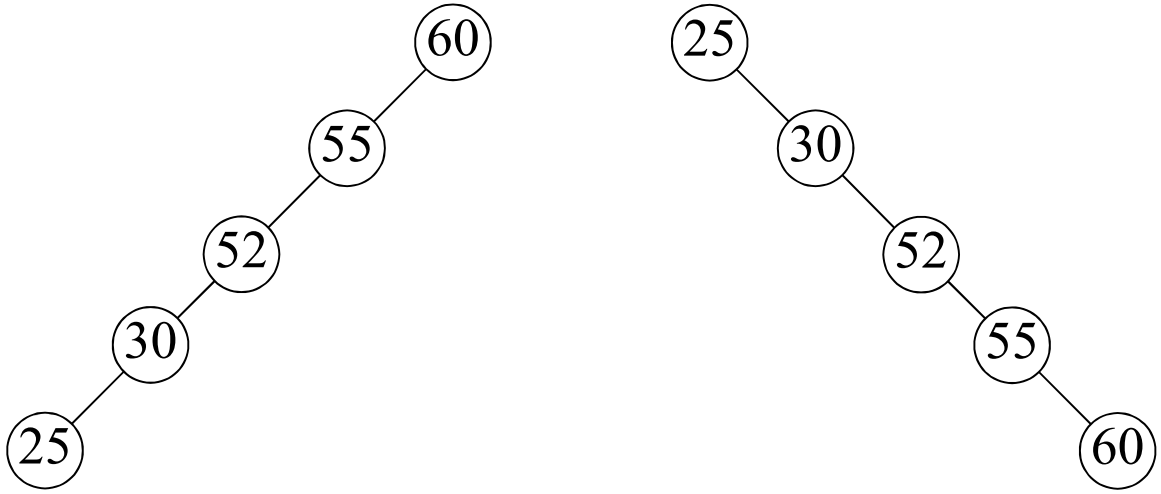

(a) 삭제전 (b) 왼쪽 서브트리의 최대 키 값(30)으로 대체 (c) 오른쪽 서브트리의 최소 키 값(52)으로 대체

이원 탐색 트리에서 자식이 둘인 노드(50)의 삭제

```
deleteBST(B, d_key)
   p ← node to be deleted;      // 주어진 키 d_key를 가진 노드
   parent ← parent node of p; // 삭제할 노드의 부모 노드
   if (p = null) then return;      // 삭제할 원소가 없음
   case {
      p.left=null and p.right=null :   // 삭제할 노드의 차수가 0인 경우 (리프 노드)
          if (parent.left = p) then parent.left ← null;
          else parent.right ← null;
      p.left=null or p.right = null :   // 삭제할 노드의 차수가 1인 경우
          if (p.left ≠ null) then { if (parent.left = p) then parent.left ← p.left;
              else parent.right ← p.left; }
          else { if (parent.left = p) then parent.left ← p.right;
                  else parent.right ← p.right; }
      p.left ≠ null and p.right ≠ null :   // 삭제할 노드의 차수가 2인 경우
          q←maxNode(p.left);   // 왼쪽 서브트리에서 최대 키 값의 노드를 탐색
          p.key ← q.key;
          deleteBST(p.left, q.key);
   }
end deleteBST()
```

# ▶ 이원 탐색 트리의 성능

- **편향 이원 탐색 트리(skewed binary search tree)**
  - 리프 노드의 탐색 시간은 최악
  - N개의 노드인 이원 탐색 트리에서 최악의 탐색 시간 : N번의 노드 탐색



- 이원 탐색 트리의 특징 (단점)
  - 삽입, 삭제 후 효율적 접근을 위한 균형 유지 부담
  - 작은 분기율(branching factor)에 따른 긴 탐색 경로와 검색 시간
    - 분기율=2 이므로, 각 노드는 많아야 두 개의 서브트리

- **BST의 높이**
  - N개의 노드를 갖는 높이 h인 BST의 최소 높이

$$N \le 2^h - 1$$

$$N + 1 \le 2^h$$

$$\lceil \log_2(N+1) \rceil \le h$$

  - BST의 높이 h의 범위

$$\lceil \log_2(N+1) \rceil \le h \le N$$

    - Best case : Complete binary tree
    - Worst case : Skewed binary tree

- 성능 개선 방향
  - 자주 접근되는 노드를 루트에 근접하도록 유지
    - 만일 미리 노드의 접근 빈도수를 알고, 삽입/삭제가 없다면, 최적의 이원 탐색 트리의 구축이 가능
    - 현실적이지 못함
  - 균형 트리(balanced tree)
    - 모든 노드에 대해 양쪽 서브트리의 노드수가 가능한 같게 만들어 트리의 최대 경로 길이를 최소화
    - 즉, 삽입과 삭제 후에는 트리가 균형을 유지하도록 다시 조정

# ❖ AVL 트리

- **완전 균형 이원 탐색 트리(Completely balanced BST)**
  - 모든 노드에 대해 양쪽 서브트리의 깊이가 가능한 같게 만듬
    - 트리의 최대 경로 길이를 최소화
  - 즉, 삽입과 삭제 후에는 트리가 균형을 유지하도록 전체 트리를 재균형 시킴
    - 삽입/삭제 연산 시간이 너무 커서 비현실적임.

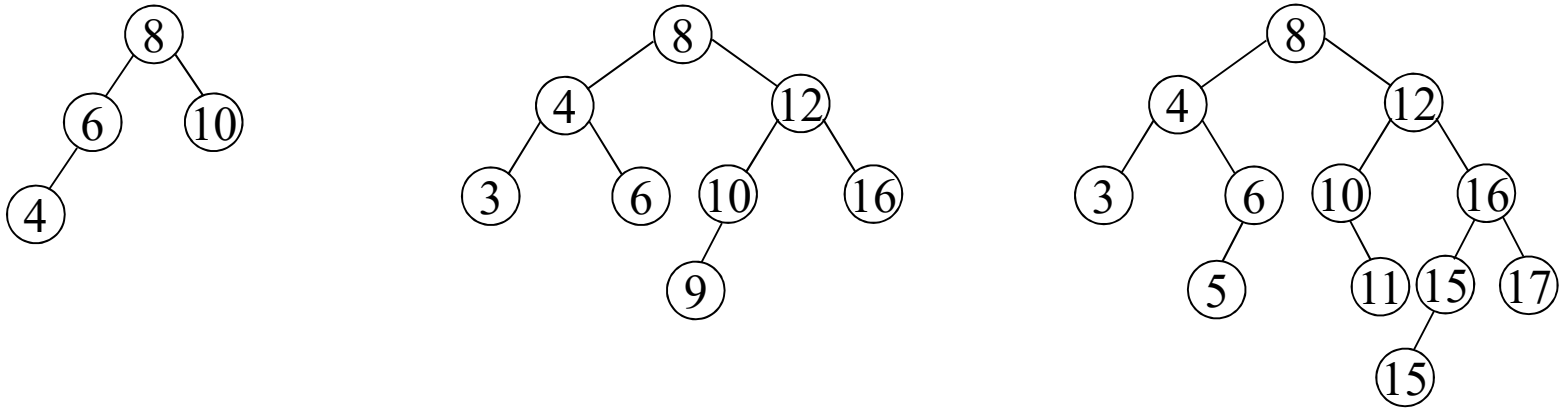
- **높이 균형 이원 탐색 트리(Height-balanced BST)**
  - 전체 트리를 재균형시키지 않고, 트리가 균형을 유지하도록 탐색 트리의 형태를 제어
    - 만일 삽입/삭제가 균형을 깨면, 트리를 부분적으로 재균형시킴
    - 회전(rotation)
    - 삽입/삭제 연산 시간이 짧음.
  - Adelson-Velskii와 Landis가 고안

- **AVL 트리 T의 정의**
  - 1. 공백이 아닌 이진 탐색 트리(BST)
  - 2. T의 모든 노드 Ni에 대해

$$\left| h(LT(Ni)) - h(RT(Ni)) \right| \leq 1 \quad Ni \in T$$

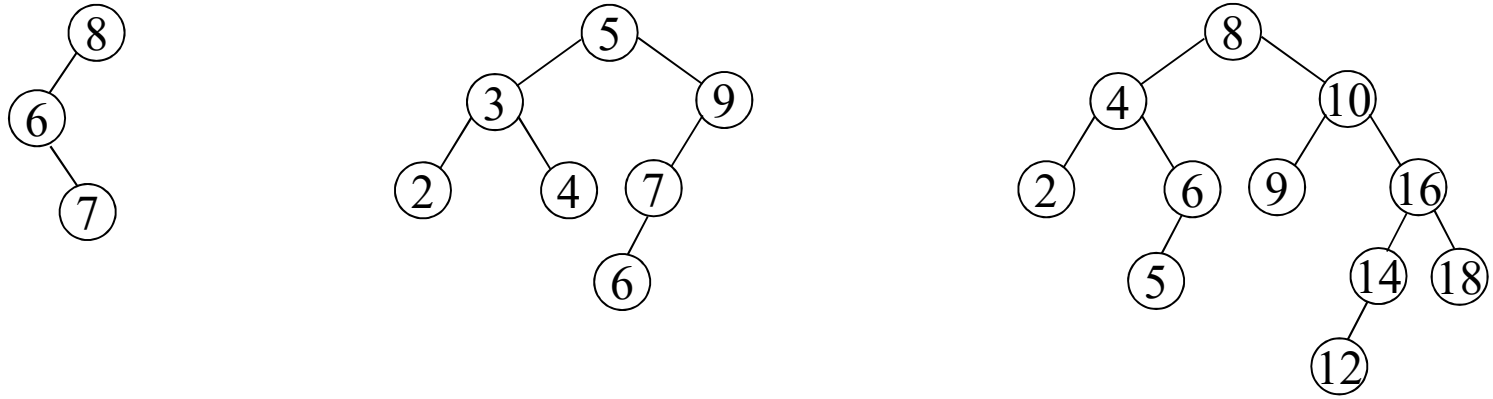
- **N개 노드 AVL트리에서 임의 접근시간 : $O(\log_2 N)$**
  - 실제로는 $O(1.4 \log_2 N)$임.

# ▶ AVL 트리, non-AVL 트리




(a) (b) (c)

AVL 트리




(a) (b) (c)

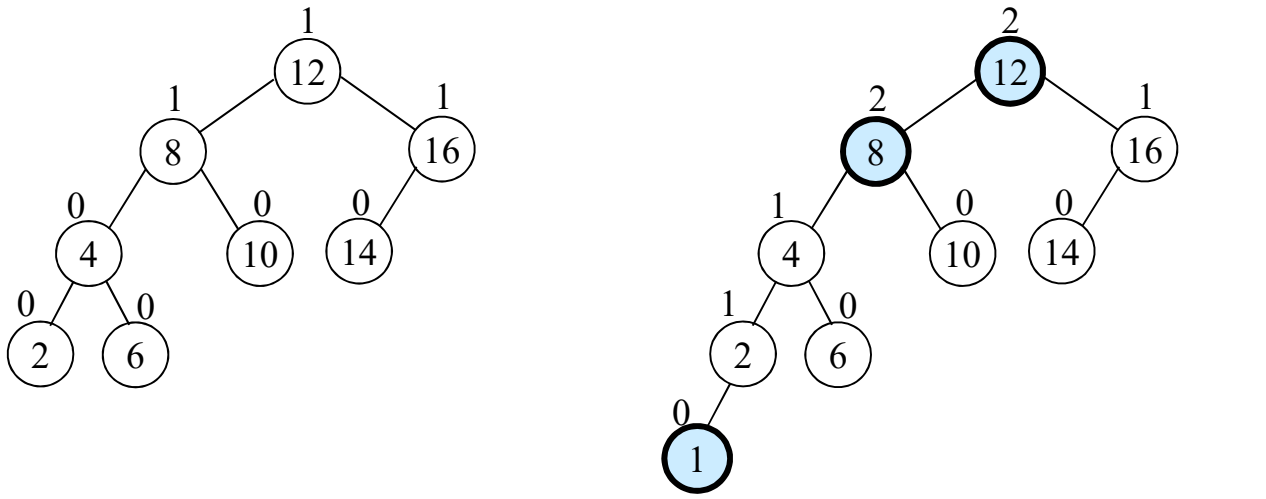
non-AVL 트리

# ▶ AVL 트리에서의 연산: 검색

- 검색
  - 일반 이진 탐색 트리의 검색 연산과 동일
  - 시간 복잡도 : $O(\log_2 N)$

# ▶ AVL 트리에서의 연산: 삽입

- 문제점
  - 삽입되는 위치에서 루트로의 경로에 있는 조상 노드들의 균형인수에 영향을 줄 수 있음
  - 따라서, 불균형이 탐지된 가장 가까운 조상 노드의 균형인수를 ±1 이하로 재균형 시켜야 함




(a) AVL 트리 (b) 원소 1의 삽입으로 non-AVL 트리로 변환

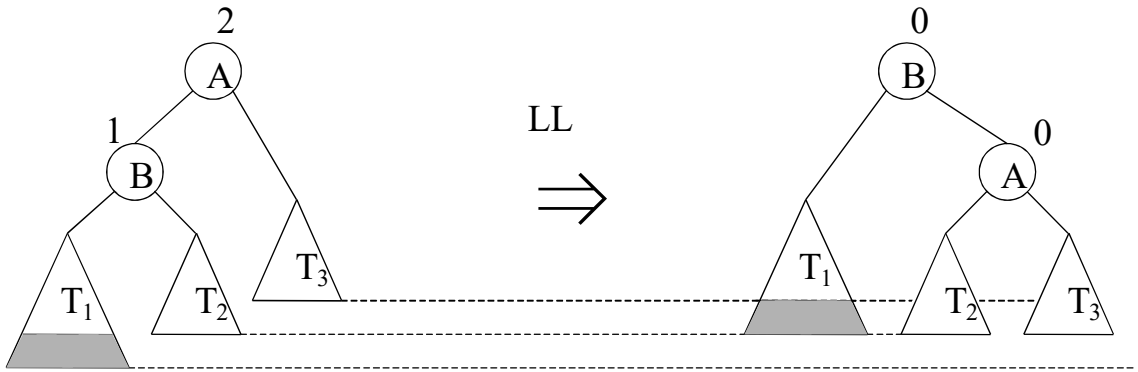
원소 삽입으로 인한 노드의 AVL 파괴 예

- 노드의 균형인수 **BF (balancing factor)**
  - BF = h(LT(Ni)) - h(RT(Ni))
- 노드 정의
  - 노드 y: 새로 삽입되는 노드
  - 노드 x: 노드의 BF가 ±2인 노드들 중 y와 가장 가까운 노드
- 삽입
  - 노드 x : 불균형으로 판명된 트리의 루트 노드
    - 노드 x의 두 서브트리 높이의 차가 2가 됨
    - 다음 4가지 경우 중 하나로 인해 발생함 (노드 y가 삽입되는 위치에 따라 회전의 이름을 결정)
    - "노드 y가 노드 x의 LL/RR/LR/RL"

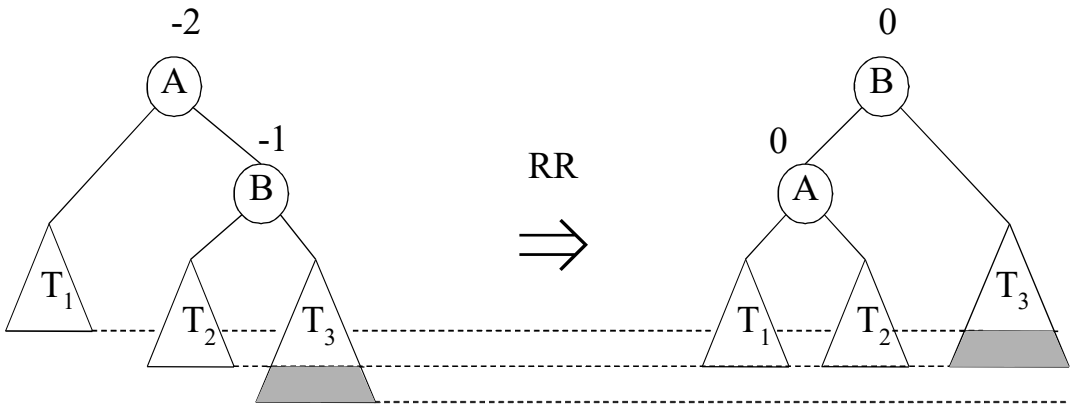
LL : x의 왼쪽 자식(L)의 왼쪽 서브트리 (L) 에 삽입
RR : x의 오른쪽 자식(R)의 오른쪽 서브트리 (R) 에 삽입
LR : x의 왼쪽 자식 (L) 의 오른쪽 서브트리 (R) 에 삽입
RL : x의 오른쪽 자식의 왼쪽 서브트리 (L) 에 삽입

# ▶ AVL 트리 삽입시 재균형 알고리즘

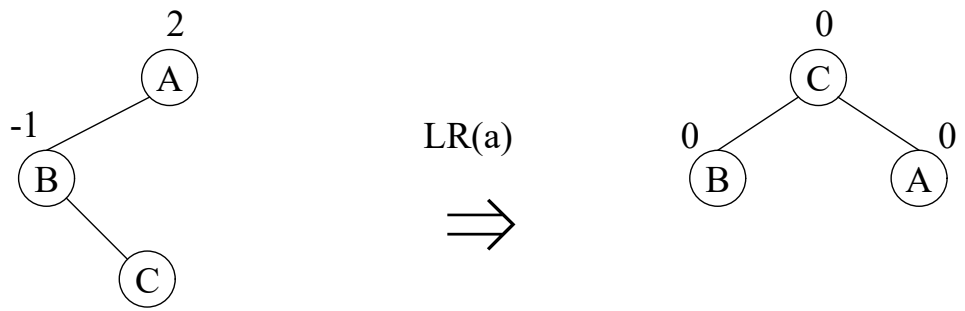
- **회전(rotation)**
  - 단순 회전(single rotation)
    - 한 번의 회전만 필요함: LL, RR
    - 탐색 순서를 유지 하면서 부모와 자식 원소의 위치를 교환
  - 이중 회전(double rotation)
    - 두 번의 회전이 필요함: LR, RL
    - 즉, LR : 부분 RR 회전 + LL 회전
      RL : 부분 LL 회전 + RR 회전



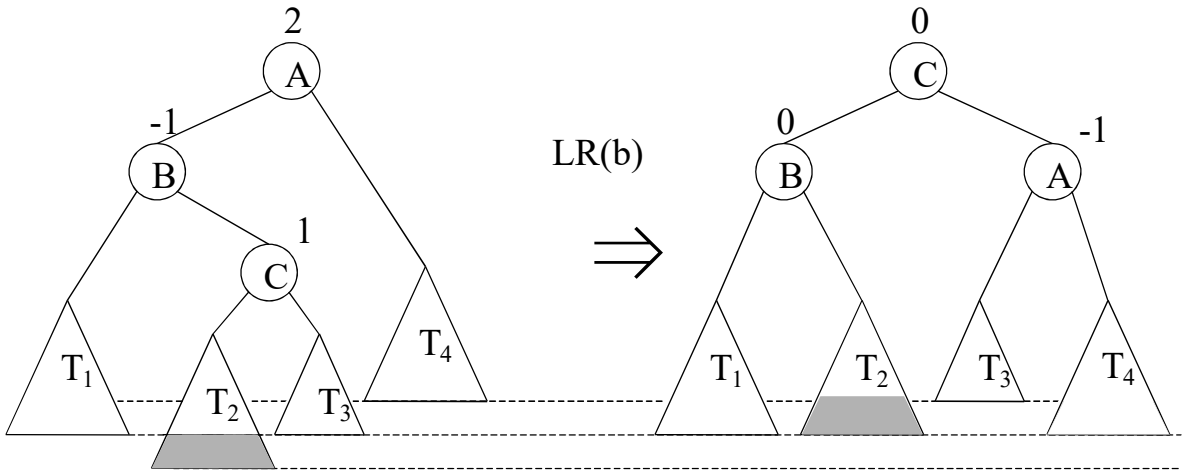

i) LL 회전



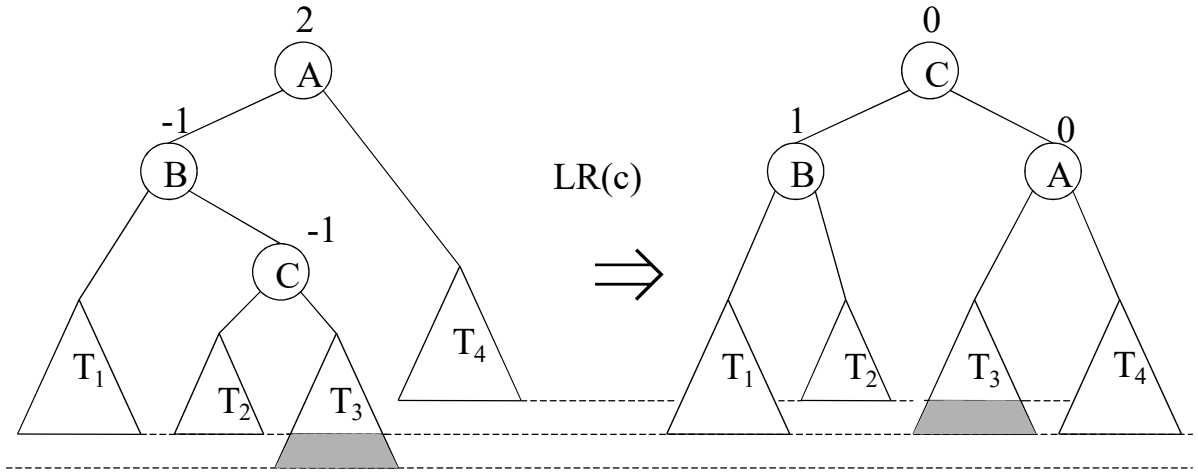

ii) RR 회전



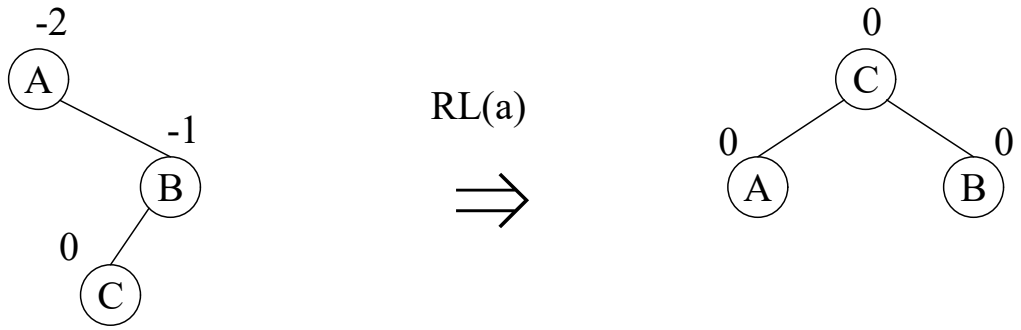

iii) LR(a) 회전



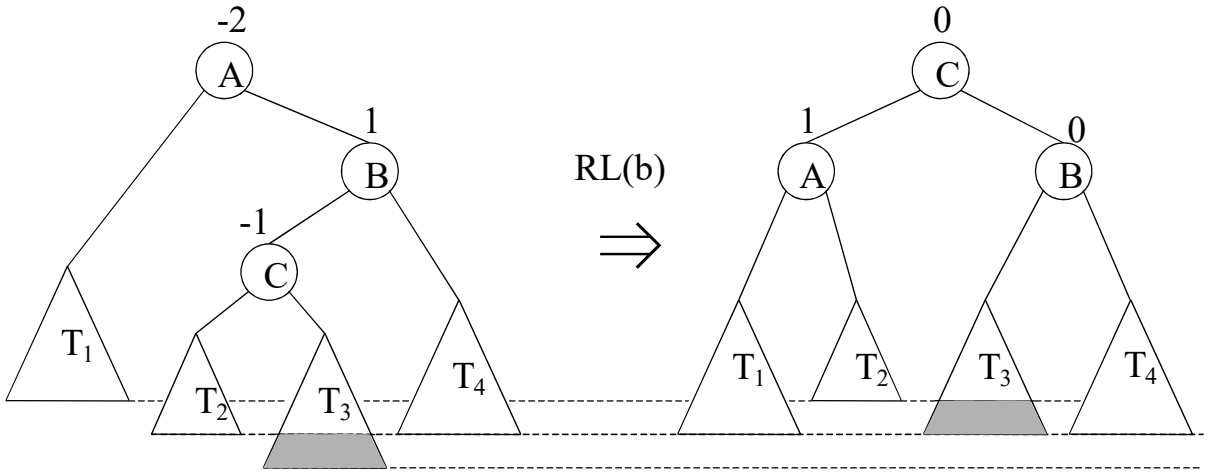

iv) LR(b) 회전



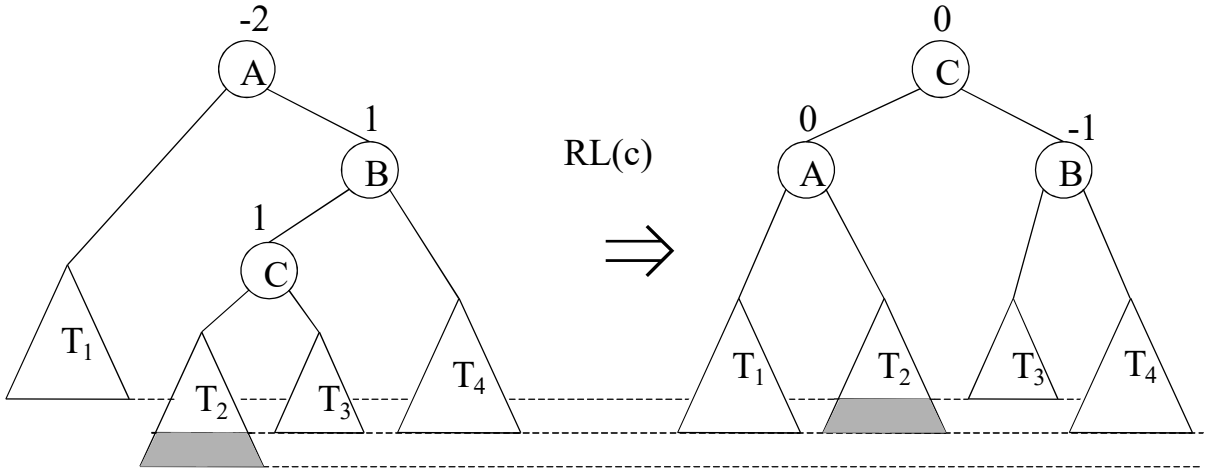

v) LR(c) 회전




vi) RL(a) 회전



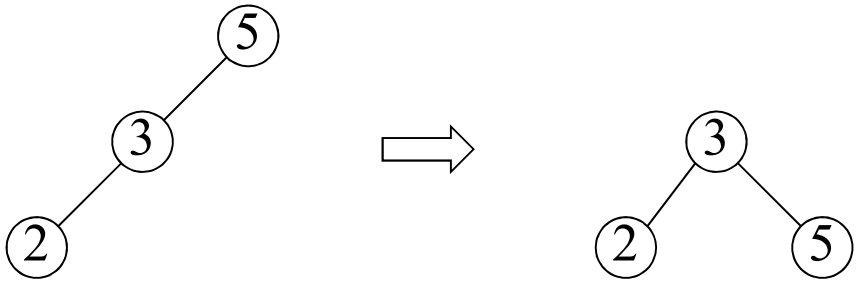

vii) RL(b) 회전



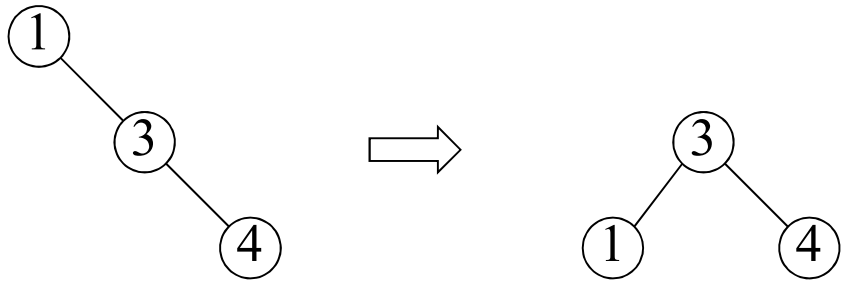

viii) RL(c) 회전

# ▶ 단순회전

◆ **LL 회전**

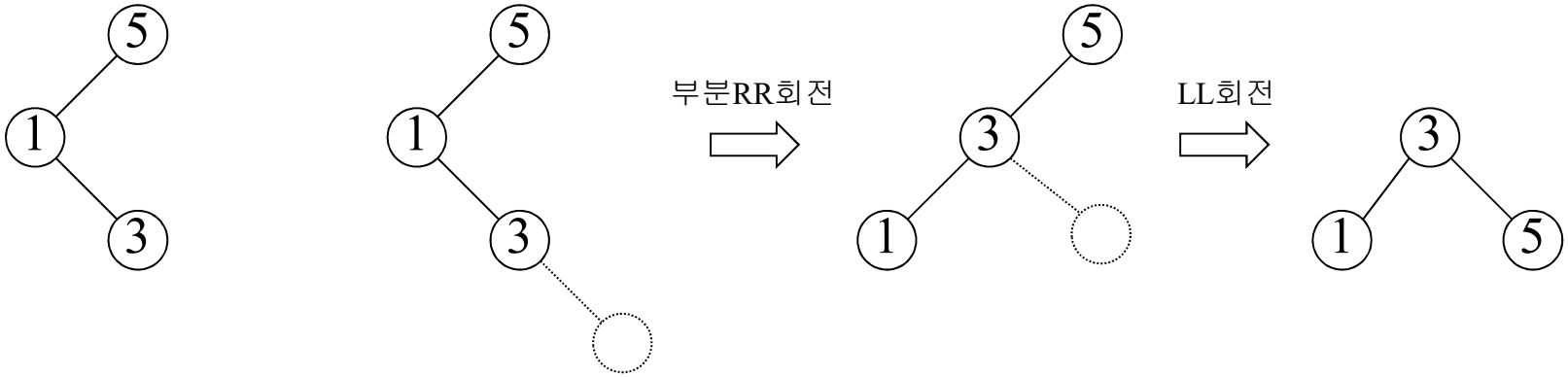



◆ **RR 회전**

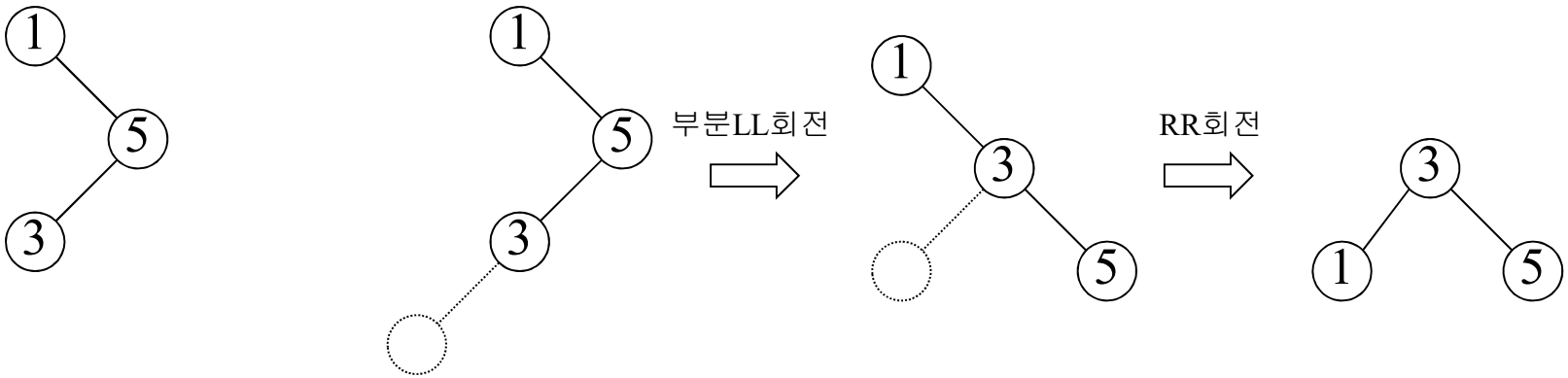


# ▶ 단순 회전을 이용한 이중 회전의 표현

## ◆ LR 회전

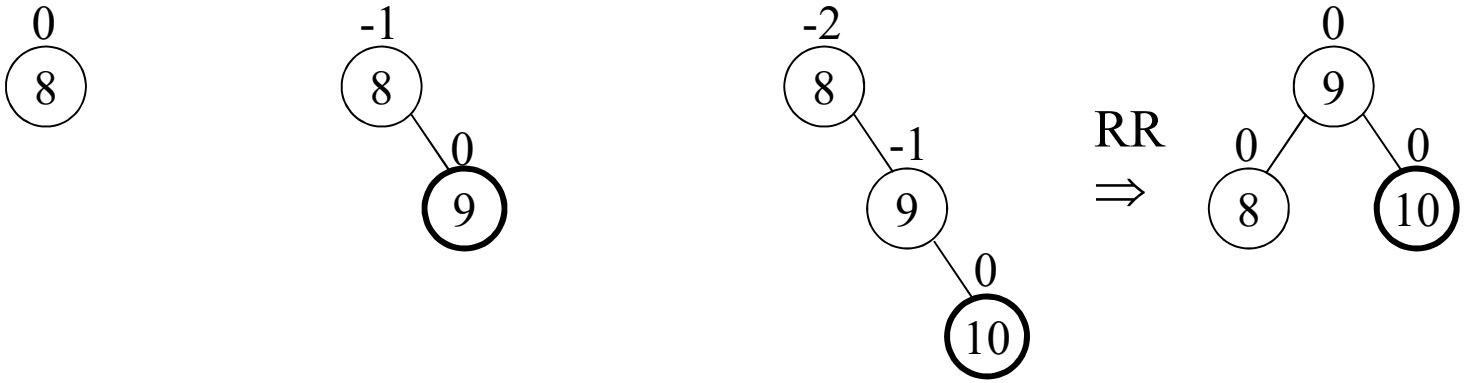



## ◆ RL 회전

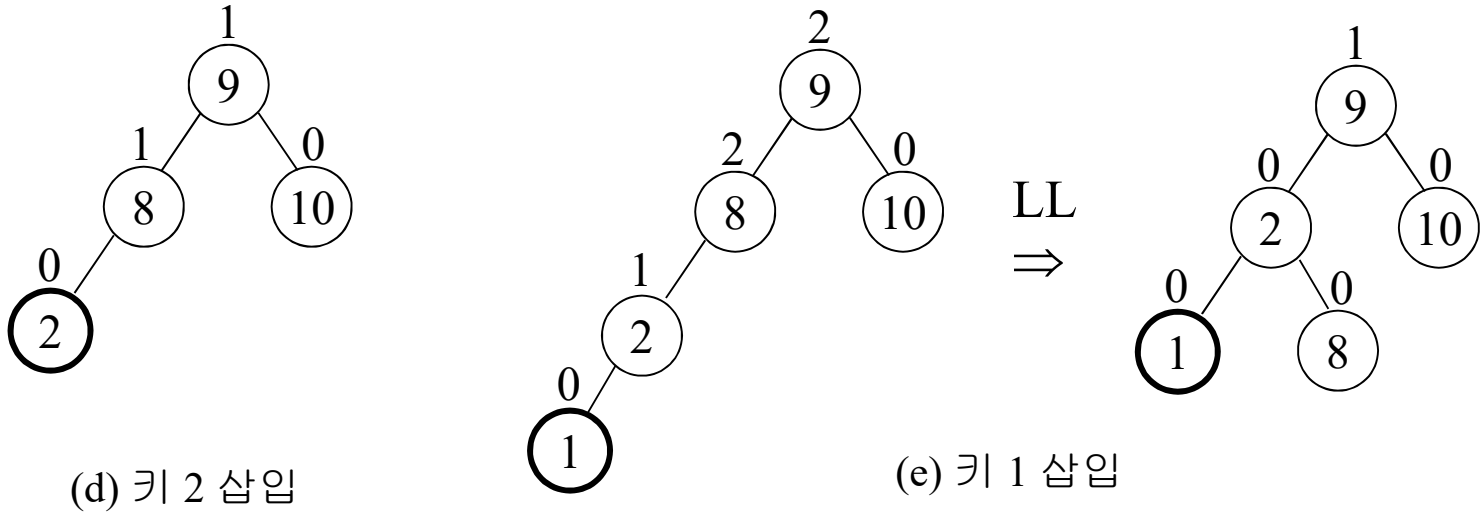


# ▶ AVL 트리 삽입 예제

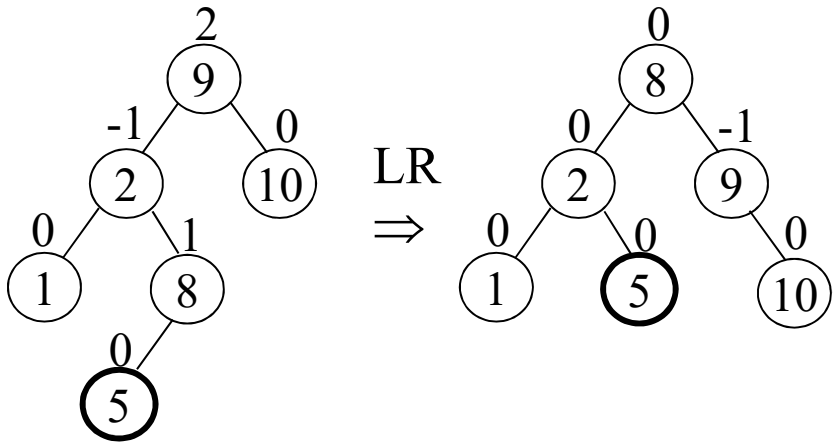
◆ 키 리스트 **(8,9,10,2,1,5,3,6,4,7,11,12)**를 차례대로 삽입하면서 **AVL** 트리를 구축하는 예



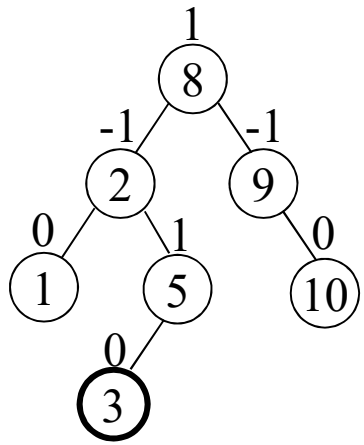

(a) 키 8 삽입 (b) 키 9 삽입 (c) 키 10 삽입



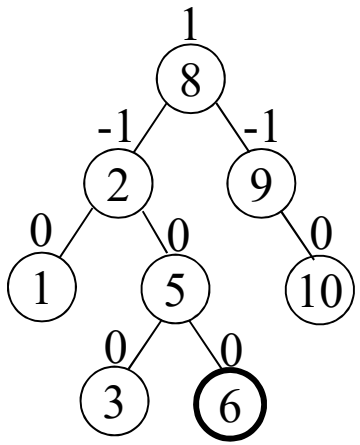

(d) 키 2 삽입 (e) 키 1 삽입



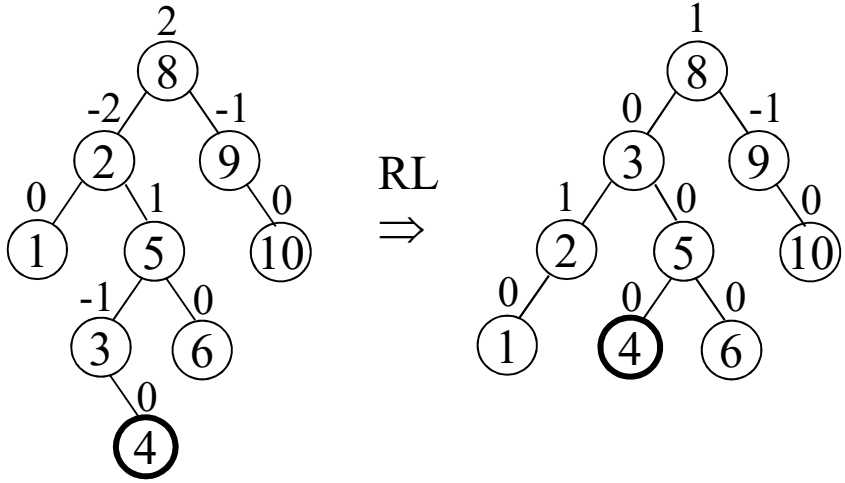

(f) 키 5 삽입



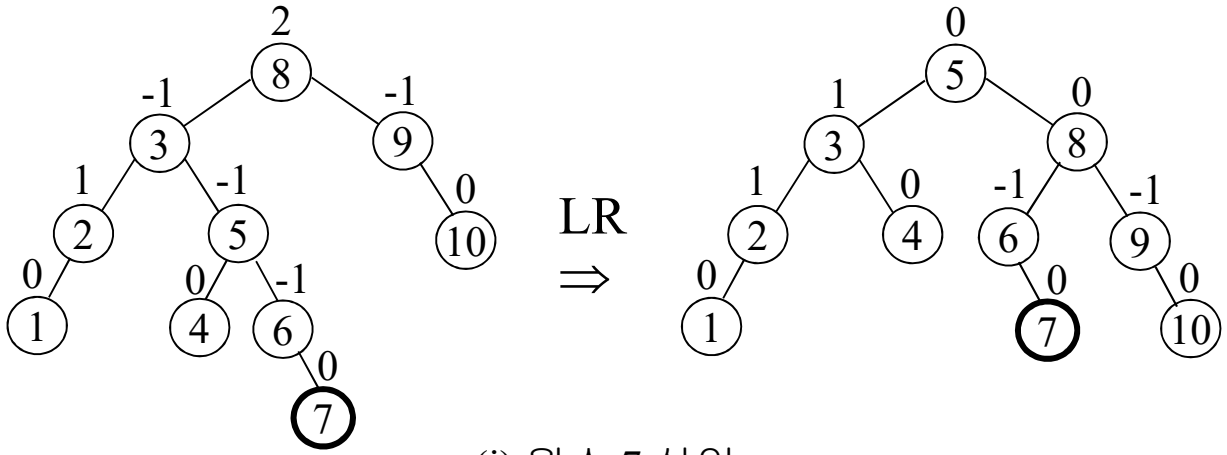

(g) 키 3 삽입



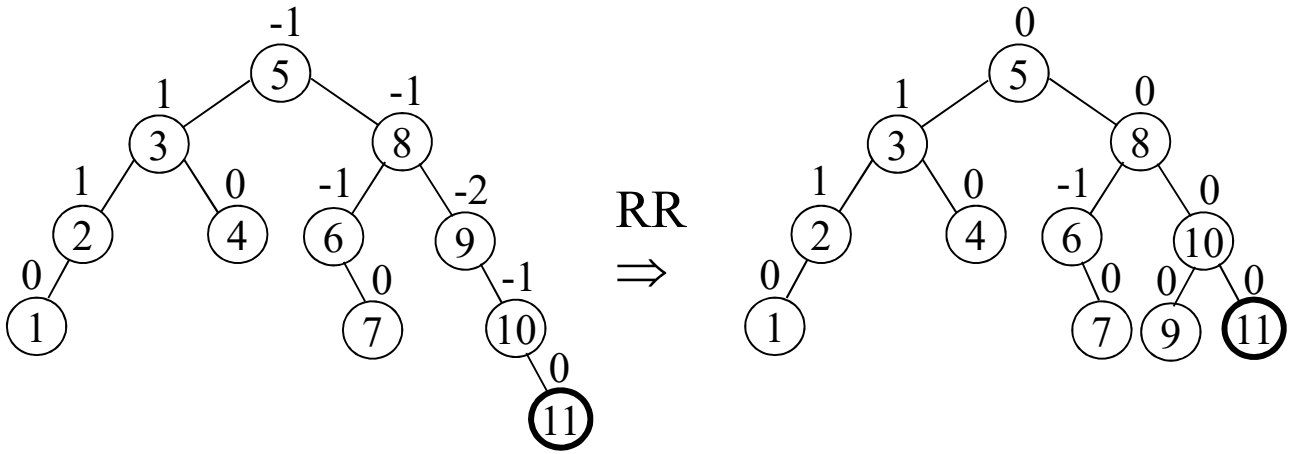

(h) 키 6 삽입



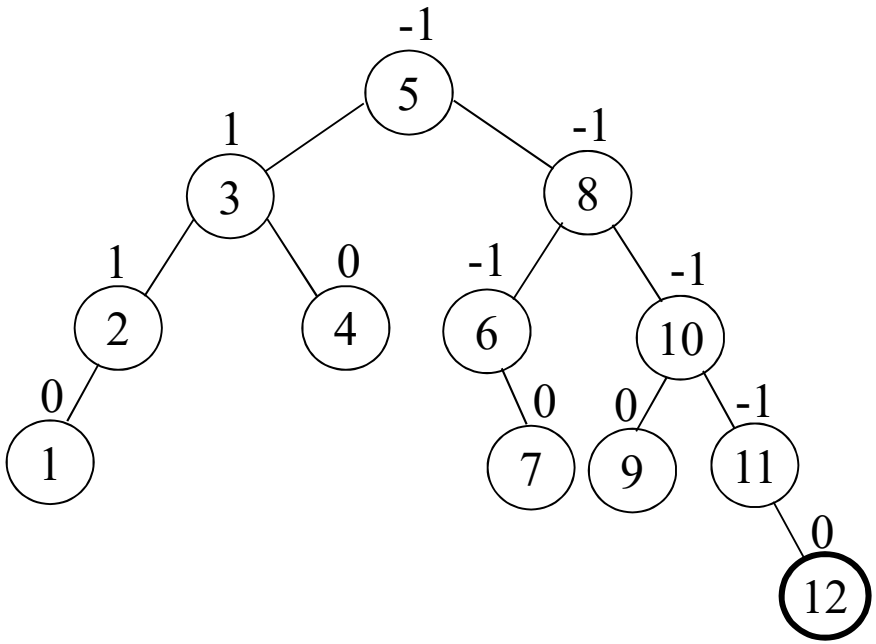

(i) 키 4 삽입




(j) 원소 7 삽입




(k) 원소 11 삽입




(l) 키 12 삽입

# ▶ AVL 트리의 성능

- **AVL 트리 = 높이 균형 BST (height-balanced BST)**

- **AVL 트리의 높이**
  - $\log_2(N+1) \le h \le 1.4404\log_2(N+2)-0.328$
  - N개의 노드를 가진 높이 균형 이원 탐색 트리(AVL 혹은 height-balanced BST)는 완전 균형 이원 탐색 트리(Balanced BST) 보다 약 45% 이상은 높아지지 않음

- **BST 계열 트리의 높이 비교**
  - BST: $\lceil \log_2(N+1) \rceil \le h \le N$
  - AVL 트리: $\lceil \log_2(N+1) \rceil \le h \le \lceil 1.44 * \log_2(N+2) \rceil$
  - 완전 균형 BST: $h = \lceil \log_2(N+1) \rceil$

- **AVL 트리 (높이 균형 BST) : 완전 균형 BST의 성능 비교**
  - 검색 성능
    - Balanced BST = $O(\log_2 N)$, AVL = $O(1.4 \log_2 N)$
    - $O(\log_2 N) \leq O(1.4 \log_2 N)$
    - AVL의 탐색 시간이 더 길다
  - 삽입/삭제 성능
    - AVL (Height-balanced BST): 부분 재균형만 수행됨
    - Balanced BST: 전체 트리의 재균형 필요

# ❖ m-원 탐색 트리(MST)

- **m-원 탐색 트리 (m-way search tree, MST)**
  - 이원 탐색 트리(BST) 보다 분기율을 높인다.
    - 즉, (m-1)개의 키, m개 서브트리
  - 분기율을 높여 트리의 높이 감소
    - 탐색 시간 감소
  - 삽입/삭제시 균형 유지의 어려움

- **BST 계열(BST, AVL)과 MST의 비교**
  - 장점 : 트리의 높이가 감소(특정 노드의 탐색시간 감소)
  - 단점 : 삽입, 삭제시 트리의 균형 유지를 위해 복잡한 연산 필요

# ▶ MST 노드의 구조

◆ **m-원 탐색 트리(MST)에서 노드의 구조**

$<n, p_0, <K_1, A_1>, P_1, <K_2, A_2>, P_2, \ldots, P_{n-1}, <K_n, A_n>, P_n>$

- n ($1 \le n \le m-1$): 한 노드 내의 키 값의 수
- $P_i$ ($0 \le i \le n$) : 서브트리에 대한 포인터
- Ki ($1 \le i \le n$) : 키 값
- Ai ($1 \le i \le n$) : 키 값으로 $K_i$를 가진 레코드에 대한 포인터

# ▶ MST의 정의

◆ **m-원 탐색 트리(MST)의 정의**

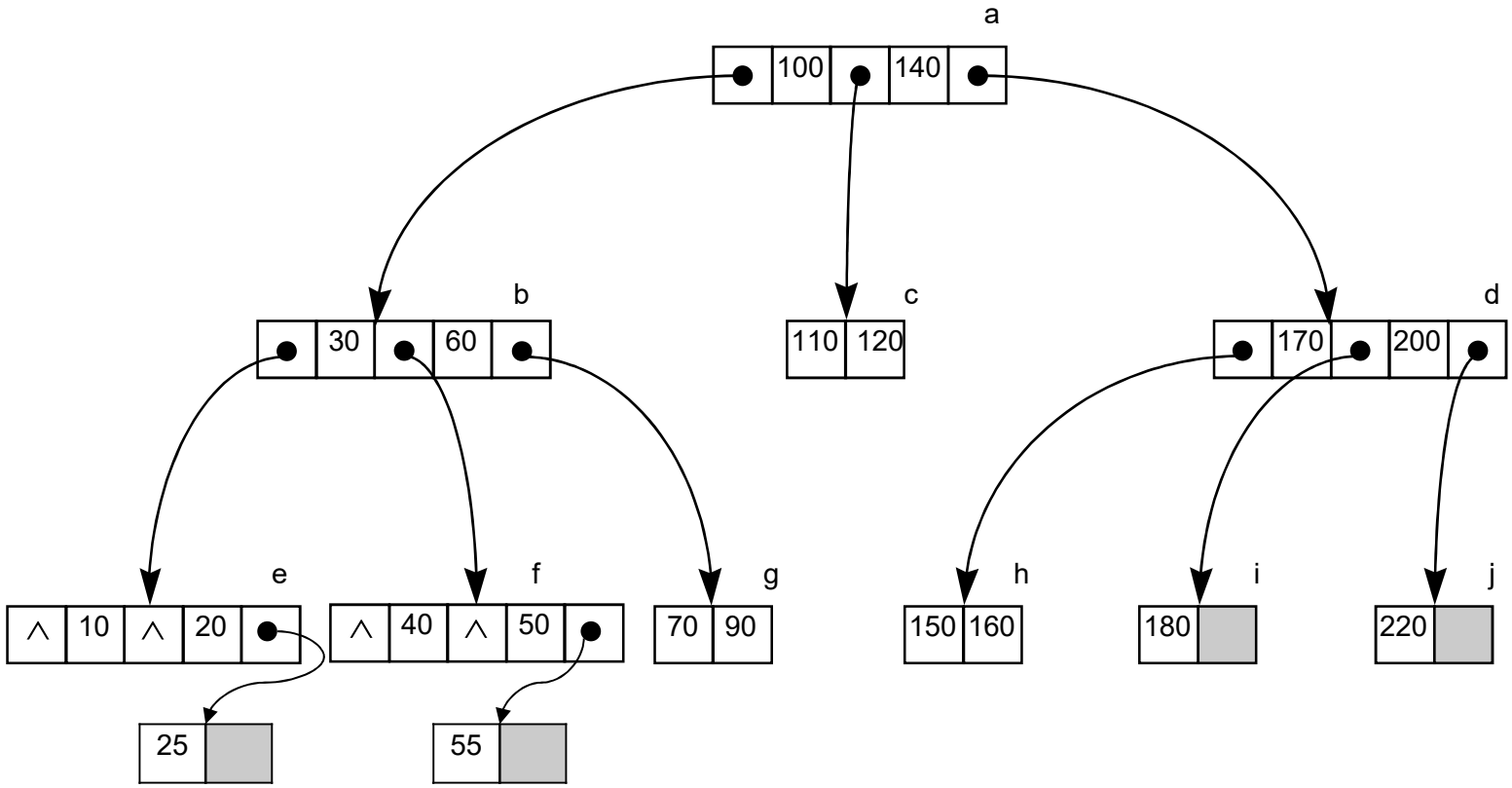
① 한 노드 내의 키 값: 키의 오름차순, 즉
$K_i < K_{i+1}$ , $1 \le i \le n-1$

② $P_i$ $(0 \le i \le n-1)$가 지시하는 서브트리 내의 키값 $< K_{i+1}$

③ $P_n$이 지시하는 서브트리 노드들의 키값 $> K_n$

④ $P_i$ $(0 \le i \le n)$ 가 지시하는 서브트리 : 위 성질을 만족하는 MST

# ▶ 3-원 탐색 트리




★ 키 값 $K_i$ : ($K_i$, $A_i$)를 의미, $A_i$ : 데이타 레코드의 주소

# ▶ MST의 검색

```
searchMST(T, key)  // m-원 탐색트리의 검색 알고리즘
        // key : 키의 값
        // x : 노드
        // T : 루트 노드
        // n : 노드에서의 키의 개수
        x ← T;
        do {
                i ← 1;
                n ← x.n;
                while( i <= n && key > x.Ki )
                        i ← i+1;
                if( i <= n && key = x.Ki)
                        then return Ai;    // 레코드의 주소를 반환
        } while( (x ← x.Pi-1) != null )
        return null;  // key와 일치하는 값이 트리에 없는 경우
end searchMST()
```

# ▶ BST와 MST

- **Binary search tree (BST)는 m-way search tree (MST)의 한 형태**
  - 2-원 탐색트리 (m=2)

- **AVL 트리와 2-원 탐색트리**
  - AVL 트리는 2-원 탐색트리
  - 그러나 그 역은 성립하지 않음.

# ▶ MST의 성능

- **m-원 MST의 높이 (N과 m이 주어졌을 때, h를 결정)**
  - 노드의 최대 개수: $m^0 + m^1 + \ldots + m^{h-1} = \frac{m^h - 1}{m - 1}$
    - 왜냐하면 등비수열 $ar^0 + ar^1 + \ldots + ar^{n-1} = \frac{a(r^n - 1)}{r - 1}$
  - 키의 최대 개수: 한 노드에 최대 (m-1) 개 키를 저장하므로

$$\frac{m^h - 1}{m - 1} * (m - 1) = m^h - 1$$

    - **4-원 탐색트리의 높이가 3이면, 최대 21개 노드에 최대 63개 키를 저장**
  - N개의 키를 갖는 m-원 MST의 최소 높이: $\lceil \log_m(N + 1) \rceil$
    - 왜냐하면 $N \le m^h - 1$
  - 따라서 MST의 높이: $\lceil \log_m(N + 1) \rceil \le h \le N$

(BST의 높이: $\lceil \log_2(N + 1) \rceil \le h \le N$ )

- **참고: N과 h가 주어졌을때, m을 결정**
  - N개의 키를 갖는 MST이 최대 높이 h를 유지하기 위해서는, m은 적어도 $(N+1)^{\frac{1}{h}}$ 이상이어야 한다.
    - 왜냐하면 $N \le m^h - 1$
    - **65535**개 키로 최대 높이가 **4**인 **MST**를 만들려면, **m**은 적어도 **16** 이상이어야 함.

- **N개의 키를 가진 m-원 탐색트리**
  - 최소 높이 $h = \lceil \log_m(N+1) \rceil$
  - 최소 탐색 시간 : $O(\log_m(N+1))$
    - **(예) m=2이면 : 이진 트리의 탐색시간**

- **m-원 탐색트리 탐색시간**
  - 탐색 경로의 길이(MST의 높이)에 비례
  - 각 레벨에서는 한 개의 노드만 탐색
  - 분기율(m)이 커지면, 트리의 높이가 낮아진다.

- **삽입, 삭제시 트리의 균형 유지**
  - 모든 내부 노드들이 m개의 서브트리를 유지할 필요는 없음
  - 그러나 균형을 유지하면, 트리의 높이가 낮아져 탐색 성능이 향상됨
  - 이를 위해서는 삽입/삭제를 위한 별도의 연산이 필요