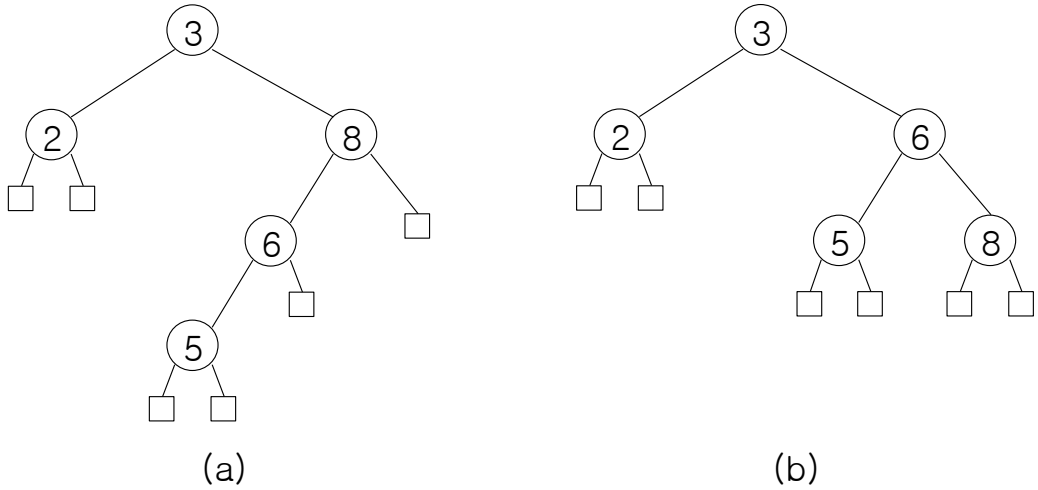
12장 균형 탐색 트리

순서

확장 이진 트리(1)

- **외부 노드(external node)**
  - 이진 트리 : n개의 노드, n+1개의 널 링크
  - 널 링크에 사각형 노드(외부 노드)를 붙이면 처리에 편리
  - 실패 노드(failure node)라고도 한다.
  - cf) 내부 노드(internal node) : 원래의 트리 노드
- **확장 이진 트리(extended binary tree)**
  - 외부 노드가 추가된 이진 트리



확장 이진 트리

확장 이진 트리(2)

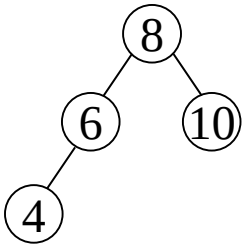
- 확장 이진 트리
  - 외부 경로 길이(external path length ; E)
    - 루트에서 각 외부 노드까지의 경로 길이를 모두 합한 것
  - 내부 경로 길이(internal path length ; I)
    - 루트에서부터 각 내부 노드까지의 경로 길이를 모두 합한 것
  - E = I + 2n(n : 내부 노드)
    - I가 최대 : 트리가 편향일 때, I가 최소 : 완전 이진 트리일 때
  - 성공적 탐색 평균 비용 : (I + n) / n
  - 모든 원소를 탐색하는 확률이 같을 때, 평균 내부 경로 길이 : $1.38n\log n \rightarrow O(n\log n)$
  - 실패한 탐색(삽입을 위한 탐색)의 평균 비용 : (E / (n + 1))
    - N+1 : 실패 노드 수

AVL 트리

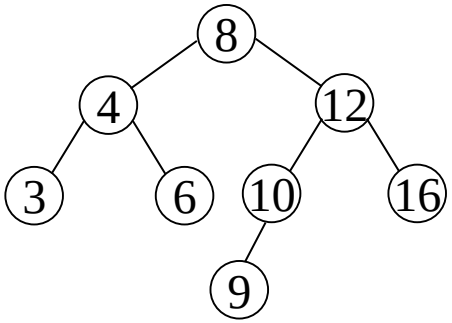
- **균형 이진 탐색 트리(height-balanced binary search tree)**
  - 원소의 탐색 시간 최적화 : O(logn)
  - 원소의 삽입과 삭제가 진행되면서 균형 유지에 비용이 많이 듬

- **AVL 트리**
  - Adelson-Velskii와 Landis에 의해 제안
  - 각 노드의 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 차이가 1 이하인 이진 탐색 트리.
    - 공백 서브트리의 높이는 -1로 정의
  - 모든 노드들이 AVL 성질을 만족하는 이진 탐색 트리
  - 균형인수(balance factor)
    - 왼쪽 서브트리의 높이 – 오른쪽 서브트리의 높이
    - 한 노드의 AVL 성질 만족 여부를 나타냄
    - 노드의 균형 인수가 ±1 이하이면 AVL 성질을 만족

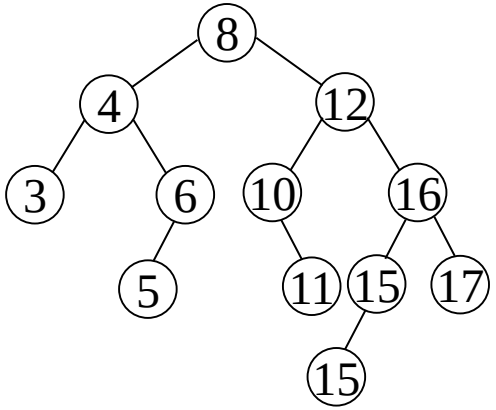
AVL 트리, non-AVL 트리



(a)

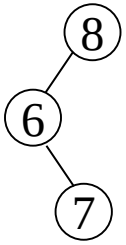


(b)

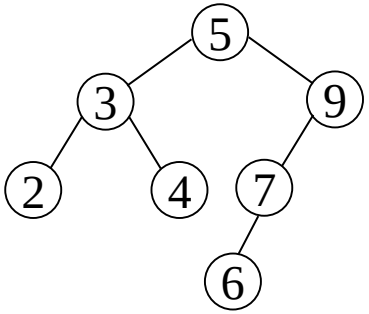


(c)

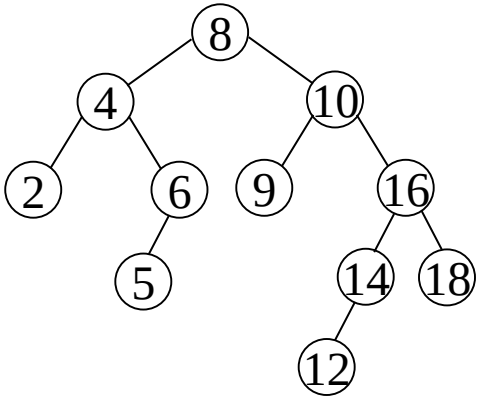
AVL 트리



(a)



(b)

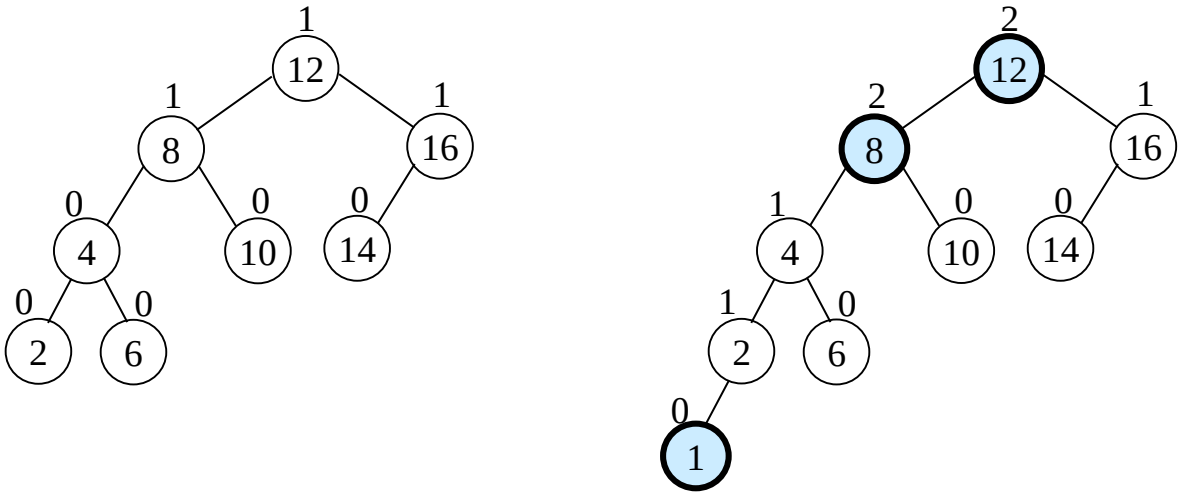


(c)

non-AVL 트리

AVL 트리에서의 검색과 삽입(1)

- 검색
  - 일반 이진 탐색 트리의 검색 연산과 동일
  - 시간 복잡도 : O(logn)
- 삽입
  - 삽입되는 위치에서 루트로의 경로에 있는 조상 노드들의 균형인수에 영향을 줄 수 있음
    - 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

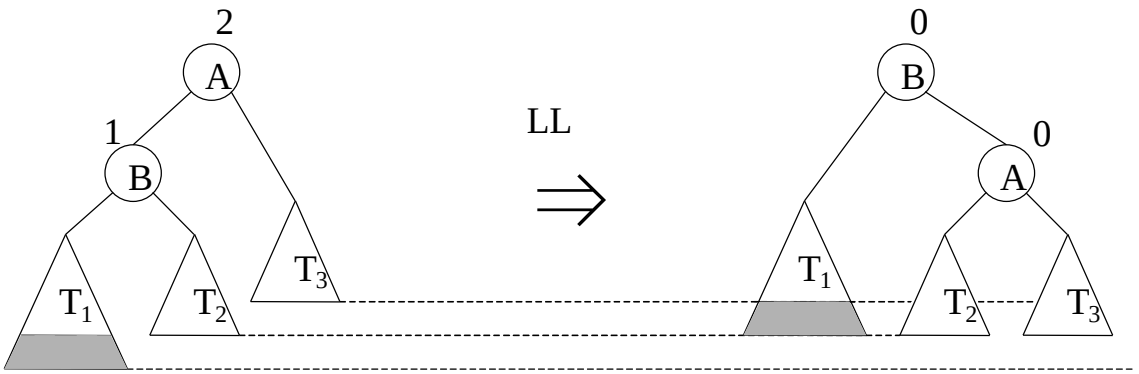
AVL 트리에서의 검색과 삽입(2)

- 삽입
  - x : 불균형으로 판명된 노드
    - x의 두 서브트리 높이의 차가 2가 됨
    - 다음 4가지 경우 중 하나로 인해 발생함

LL : x의 왼쪽 자식의 왼쪽 서브트리에 삽입
RR : x의 오른쪽 자식의 오른쪽 서브트리에 삽입
LR : x의 왼쪽 자식의 오른쪽 서브트리에 삽입
RL : x의 오른쪽 자식의 왼쪽 서브트리에 삽입

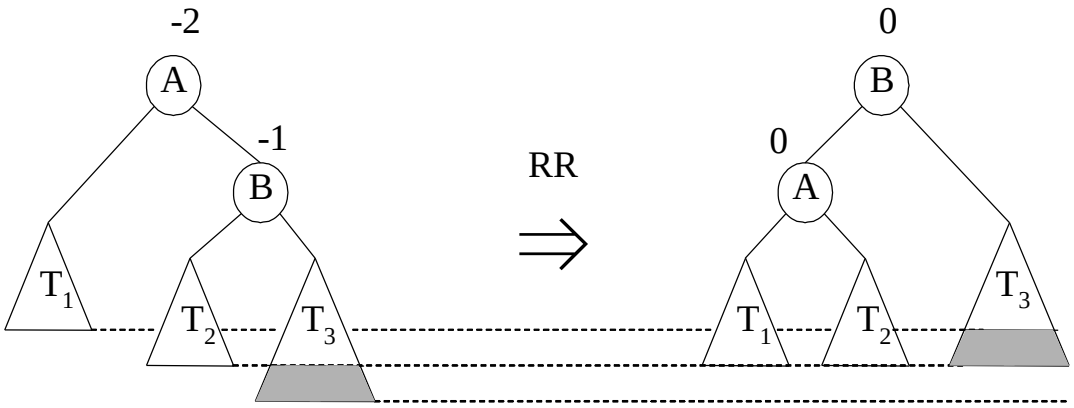
AVL 트리에서의 검색과 삽입 (3)

- **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

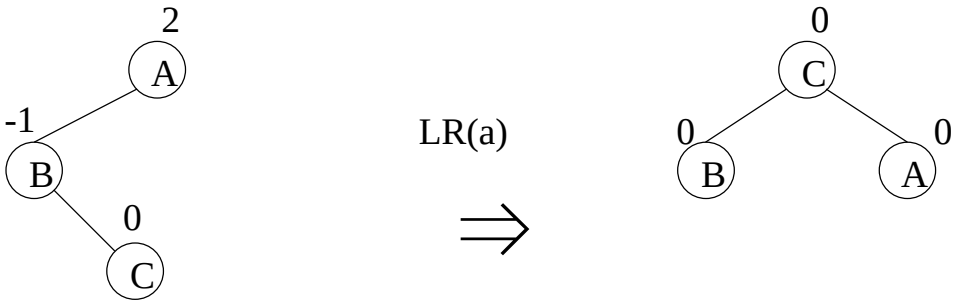


i) LL 회전

AVL 트리에서의 검색과 삽입 (4)

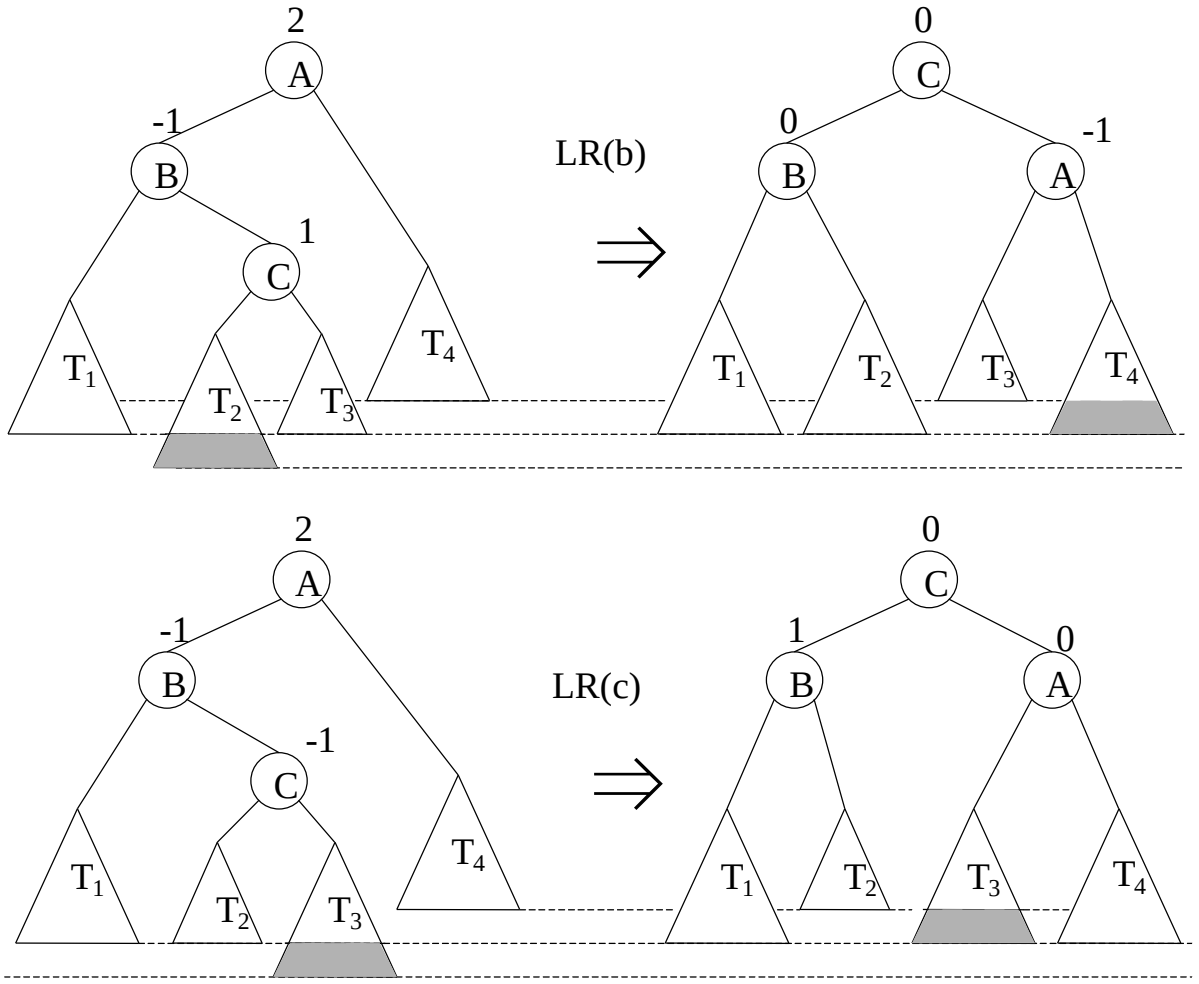


ii) RR 회전



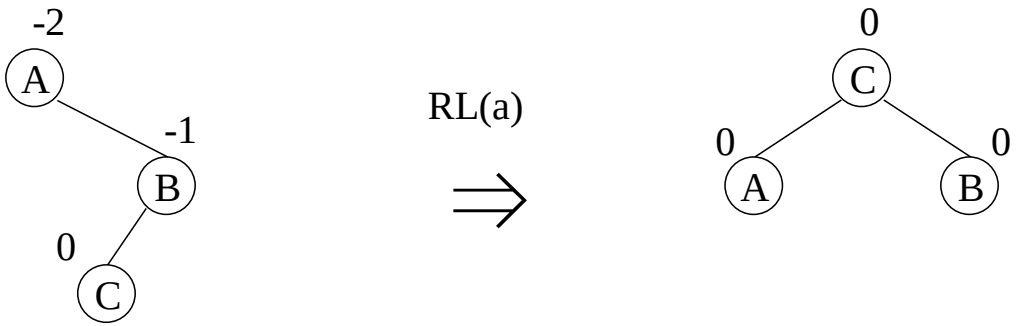
iii) LR(a) 회전

AVL 트리에서의 검색과 삽입(5)

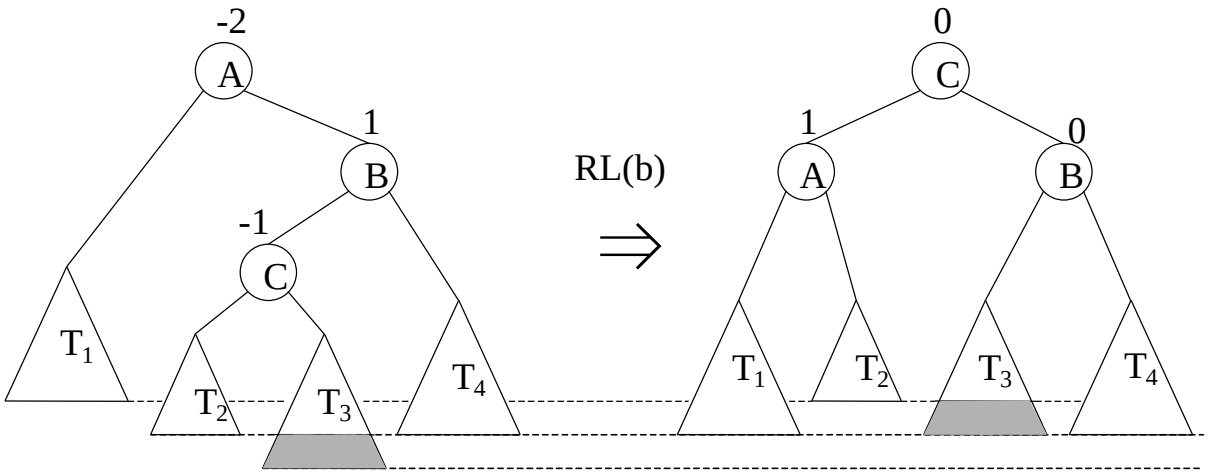


v) LR(c) 회전

AVL 트리에서의 검색과 삽입(6)

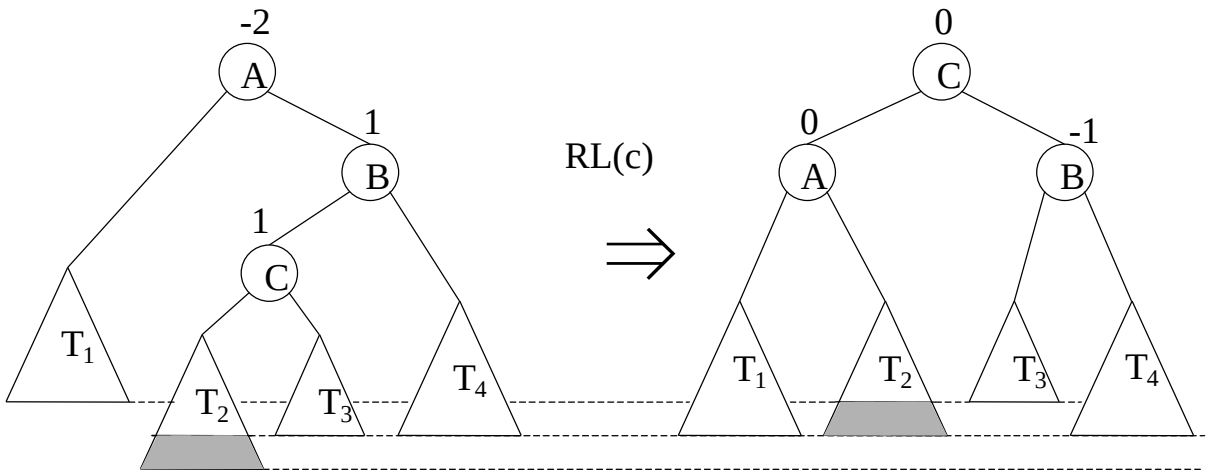


vi) RL(a) 회전



vii) RL(b) 회전

AVL 트리에서의 검색과 삽입(7)

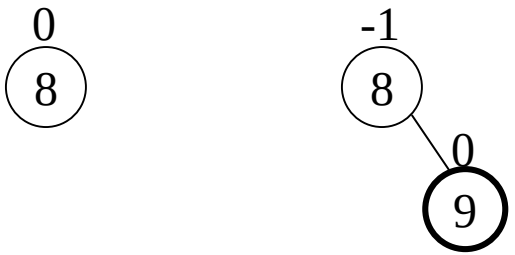


viii) RL(c) 회전

AVL 트리 회전

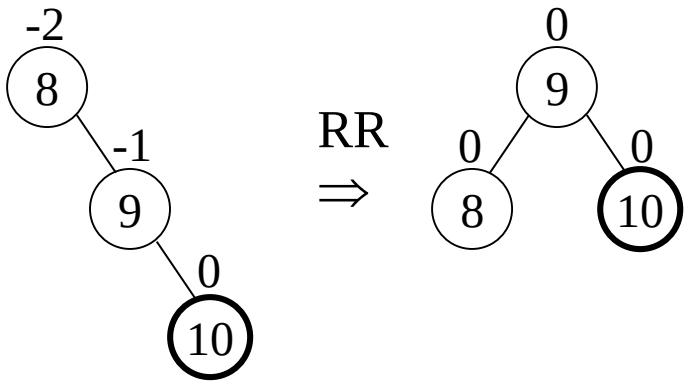
AVL 트리에서의 검색과 삽입 (8)

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

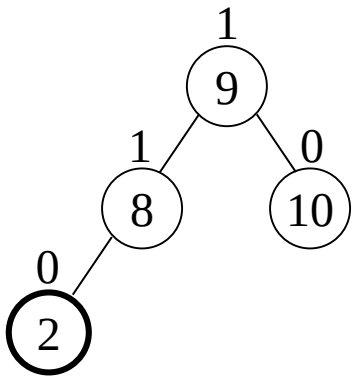


(a) 원소 8 삽입

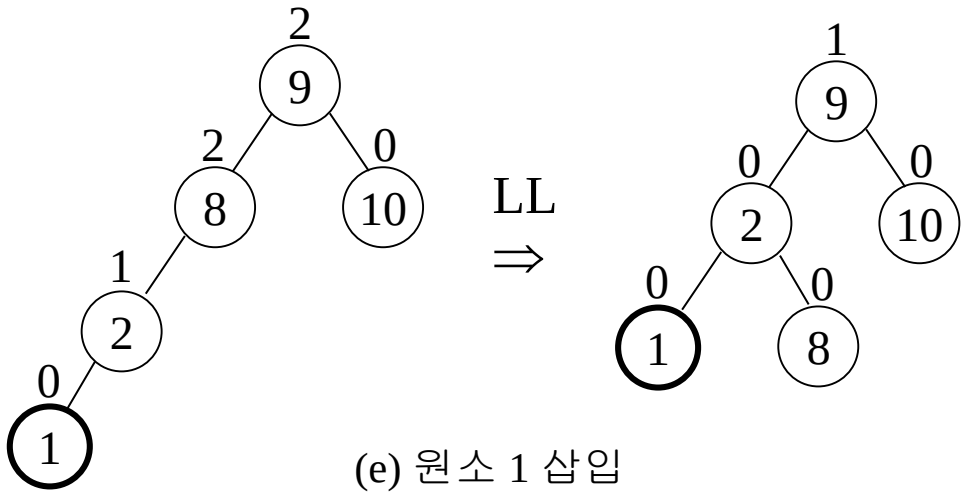
(b) 원소 9 삽입



(c) 원소 10 삽입

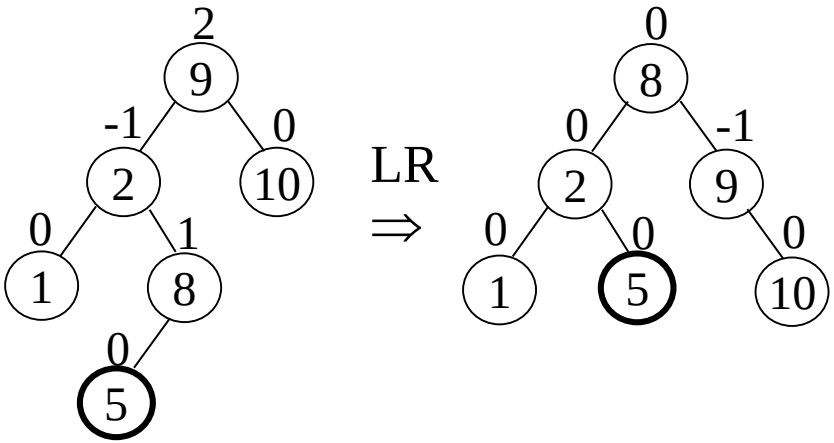


(d) 원소 2 삽입

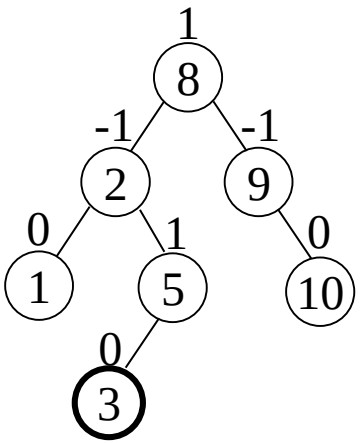


(e) 원소 1 삽입

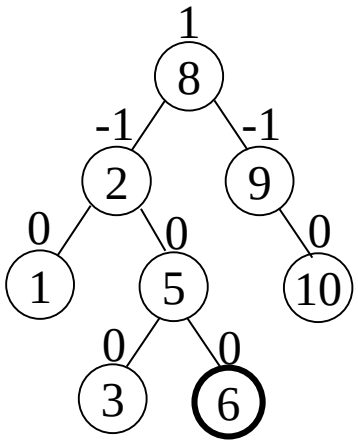
AVL 트리에서의 검색과 삽입 (9)



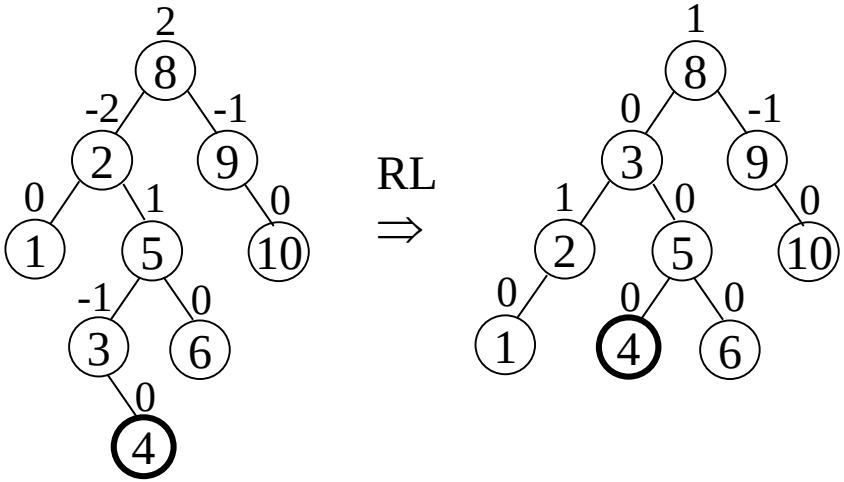
(f) 원소 5 삽입



(g) 원소 3 삽입

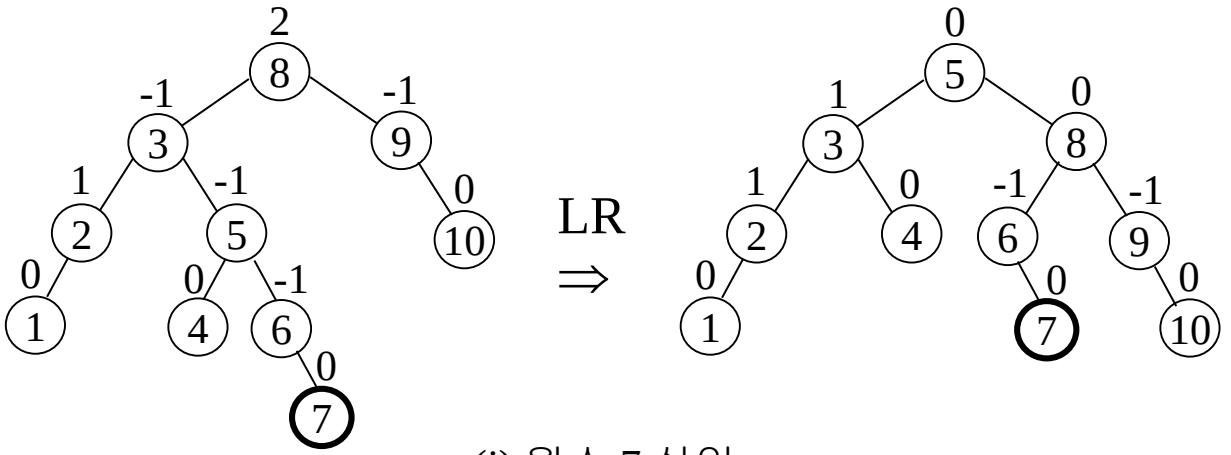


(h) 원소 6 삽입

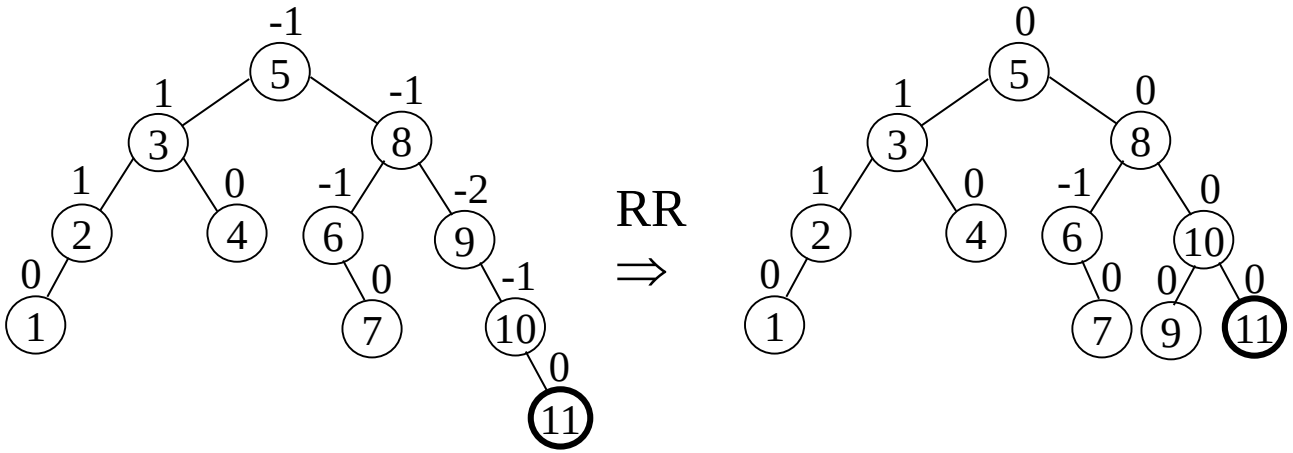


(i) 원소 4 삽입

AVL 트리에서의 검색과 삽입 (10)

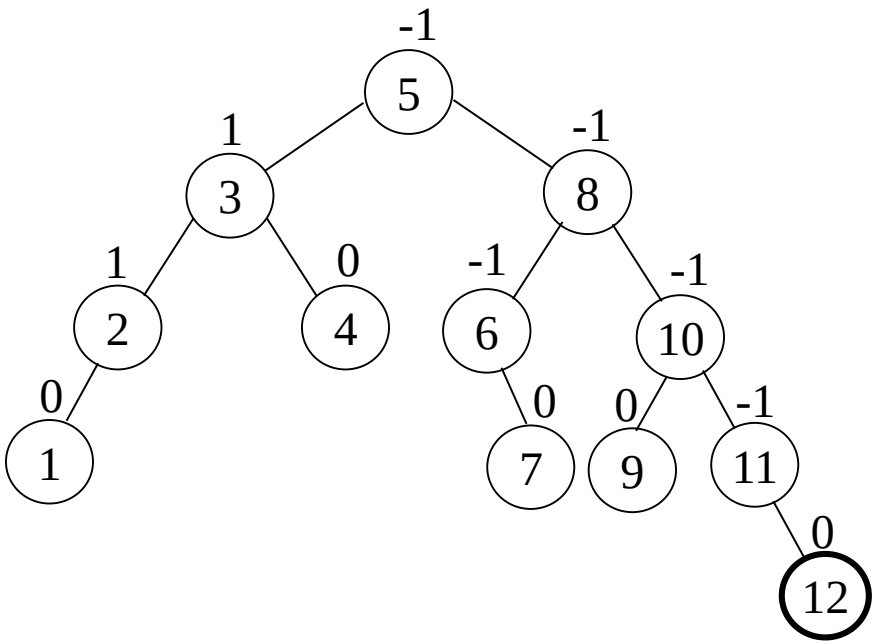


(j) 원소 7 삽입



(k) 원소 11 삽입

AVL 트리에서의 검색과 삽입 (11)



(l) 원소 12 삽입

스플레이 트리(1)

- **스플레이 트리(splay tree)**
  - 일련의 탐색, 삽입, 삭제 연산에 대한 종합적 시간이 효율적인 구조
  - 스플레이(splay)
    - 어떤 노드가 루트가 될 때까지 트리를 재구성하기 위한 일련의 회전(rotation)
  - 일반 이진 탐색 트리와 같이 탐색, 삽입, 삭제, 조인 수행
  - 각 연산이 수행된 뒤에 추가로 스플레이가 수행됨
    - 트리의 분할 연산 : 스플레이를 먼저 실행

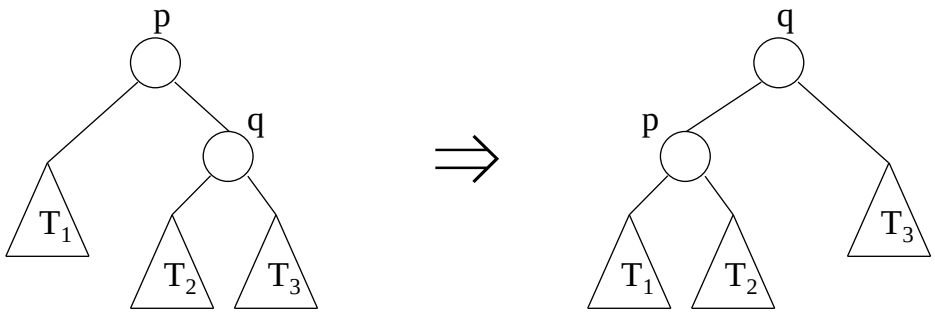
스플레이 트리(2)

- **스플레이 노드(splay node)의 결정**
  - 스플레이를 수행하는 노드는 연산에 따라 결정

1) 탐색 : 찾고자 하는 키값을 가진 노드
2) 삽입 : 새로 삽입한 노드
3) 삭제 : 삭제된 노드의 부모 노드. 이 노드가 루트인 경우에는 스플레이는 실행되지 않음
4) 3원 결합 : 스플레이는 실행되지 않음
5) 분할 : 트리에 있는 키 i에 대해 분할 시, i가 포함된 노드에 먼저 스플레이 수행한 후 트리를 분할

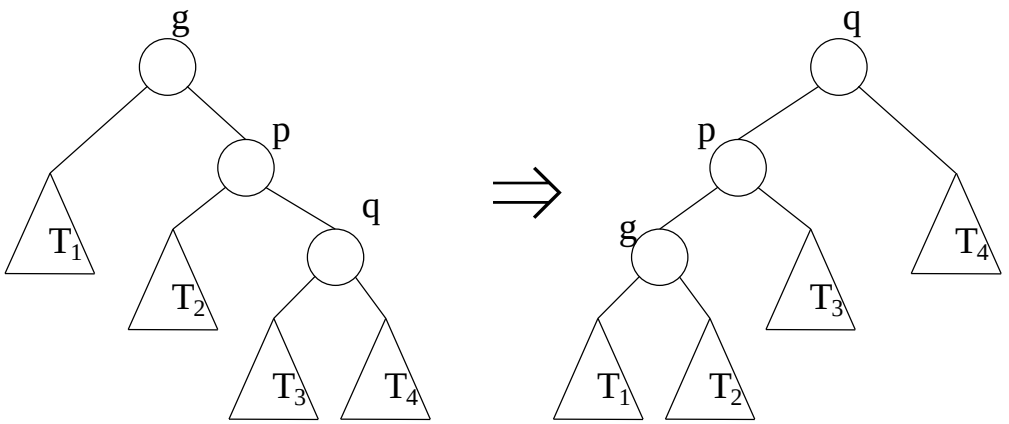
스플레이 트리(3)

- 스플레이 회전
  - 스플레이 노드(q)에서부터 루트까지의 경로를 따라가면서 수행
    1) q가 널이거나 루트이면 스플레이는 종료
    2) q가 부모 노드 p를 가지고 있으나 조부모 노드가 없음 : 회전을 수행하고 스플레이를 종료
    3) 만일 q가 부모 노드 p와 조부모 노드 g를 가지고 있음 : 회전은 LL(p는 g의 왼쪽 자식이고, q는 p의 왼쪽 자식), LR, RR, RL로 구분된다. 스플레이는 q의 새로운 위치에서 반복
  - 모든 회전은 q를 새로운 탐색 트리의 루트로 만듦
    - 분할은 루트에 대해 간단하게 수행

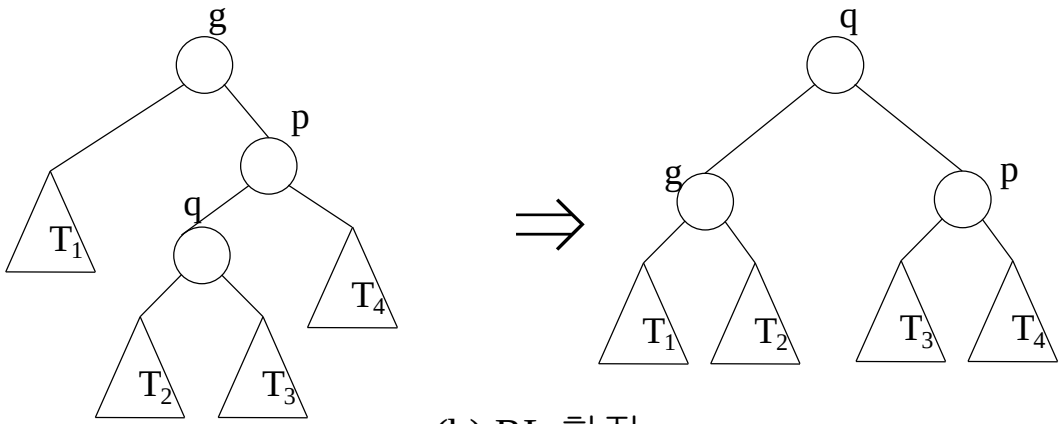


q가 오른쪽 자식이고 조부모가 없는 경우의 회전

스플레이 트리 (4)



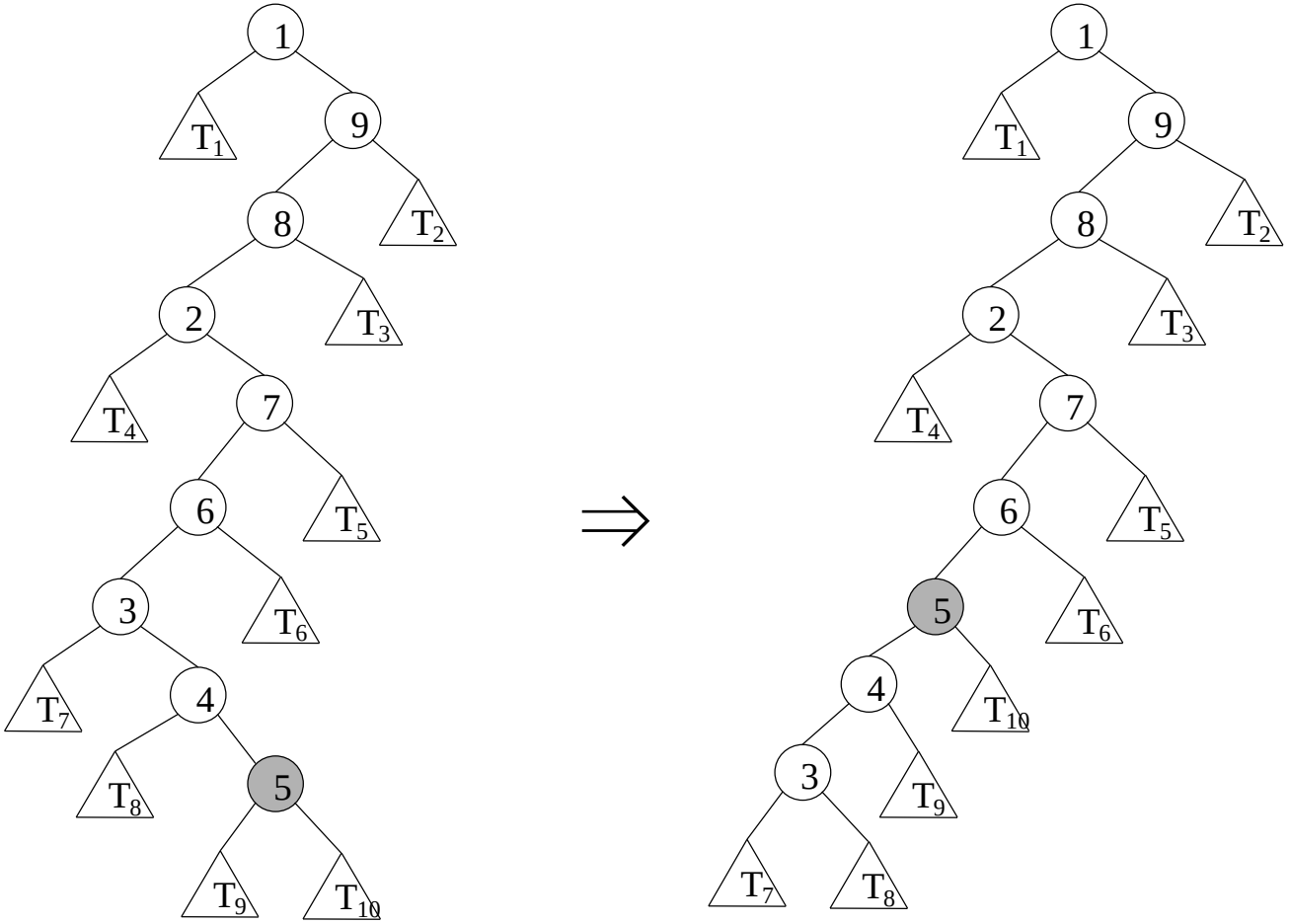
(a) RR 회전



(b) RL 회전

RR과 RL 회전

스플레이 트리(5)

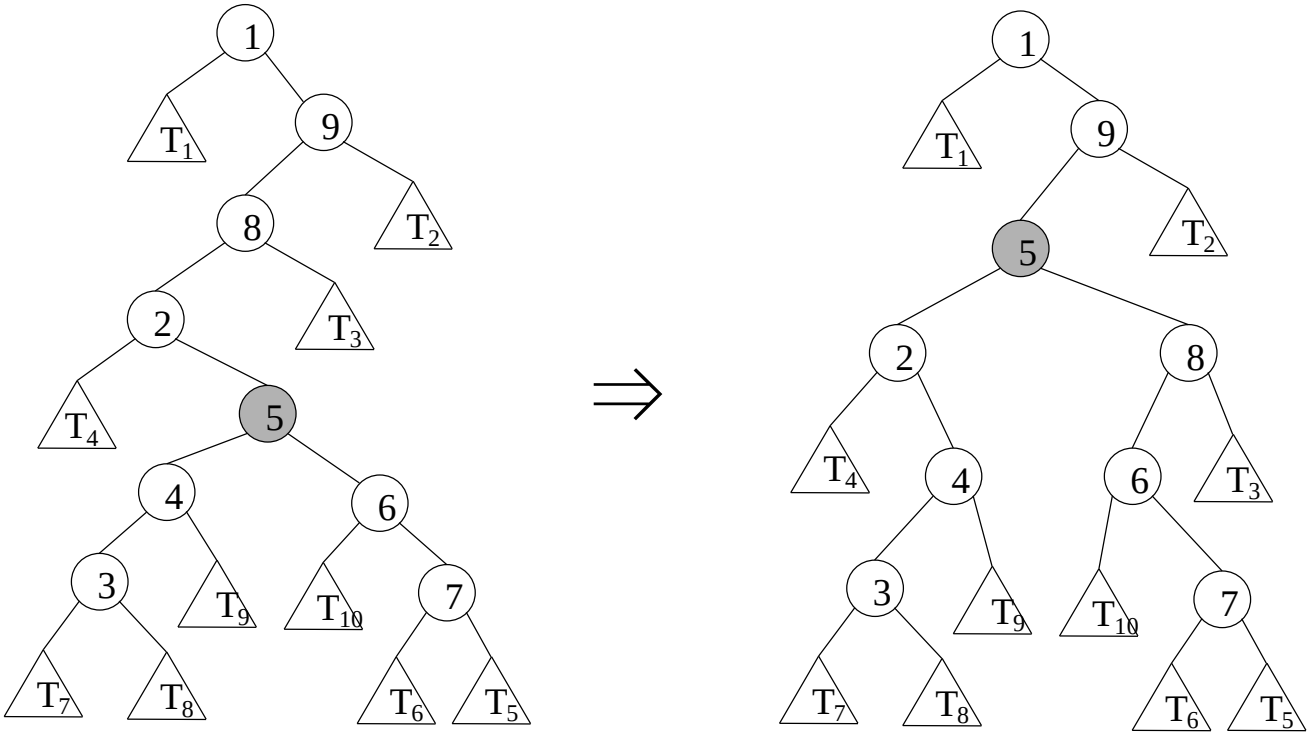


(a) 초기 탐색 트리

(b) RR 회전 이후

스플레이 노드(5)에서 시작하는 일련의 스플레이 회전

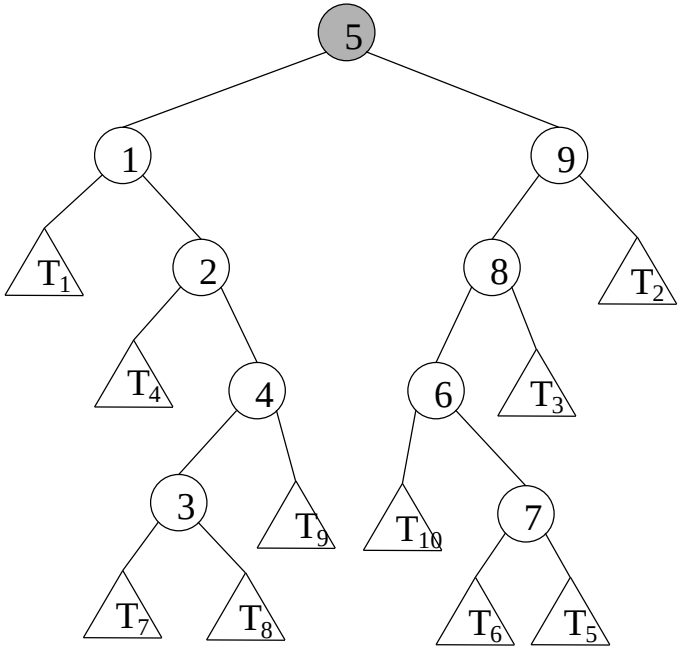
스플레이 트리(6)



(c) LL 회전 이후 (d) LR 회전 이후

스플레이 노드(5)에서 시작하는 일련의 스플레이 회전

스플레이 트리(7)

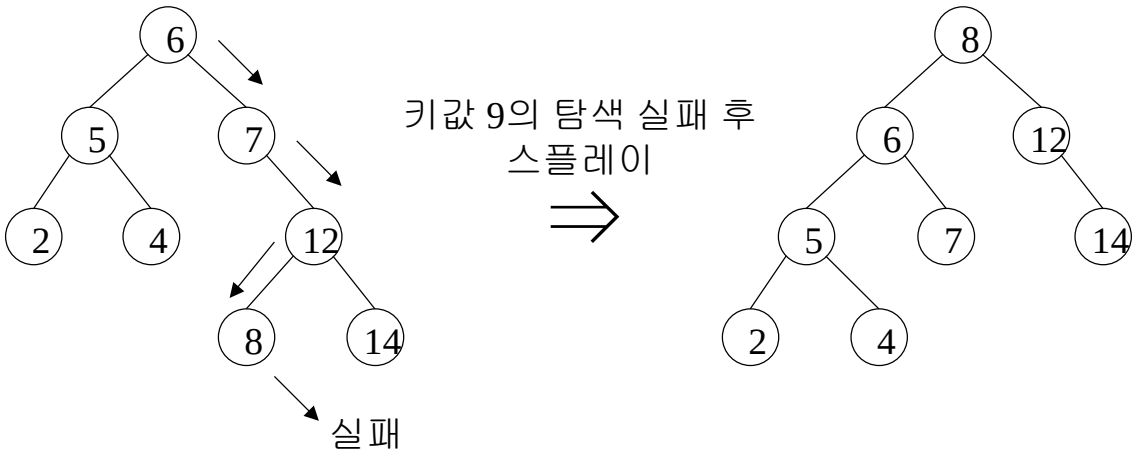


(e) RL 회전 이후

스플레이 노드(5)에서 시작하는 일련의 스플레이 회전

스플레이 트리(8)

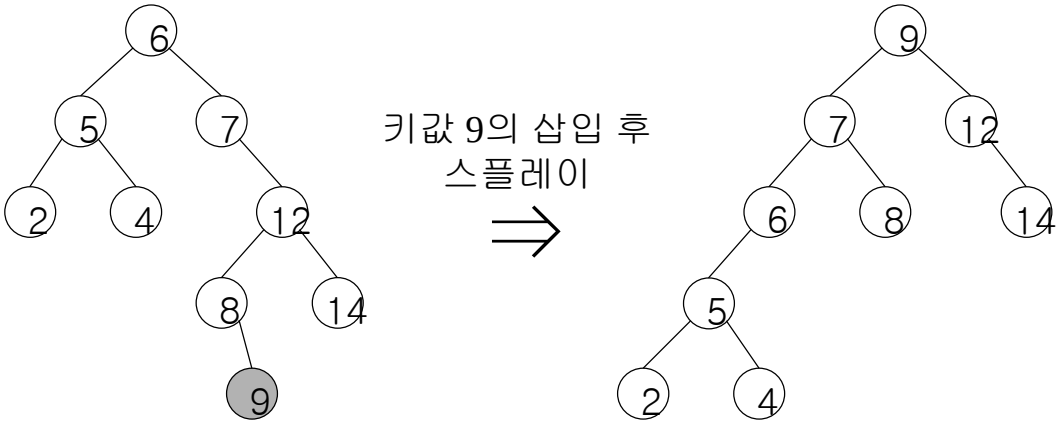
- **키값이 9인 원소를 탐색하는 연산**
  - 키값 9가 없으므로, 탐색 연산은 노드 9에서 실패로 끝남
  - 노드 8에서 스플레이 수행



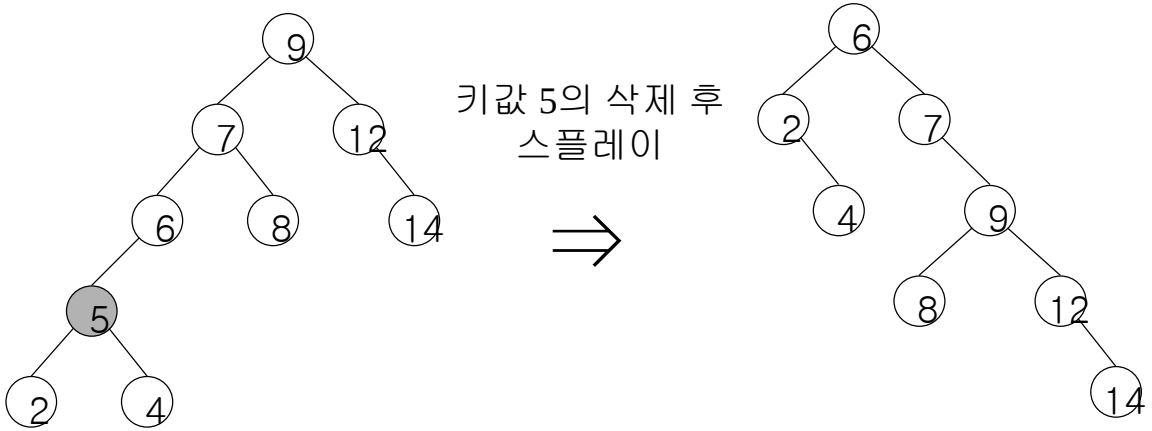
(a) 키값 9의 탐색 (b) 노드 8에서 스플레이 수행 결과

실패한 탐색 연산 뒤의 스플레이

스플레이 트리(9)



(a) 키값 9의 삽입 연산



(b) 키값 5의 삭제 연산

삽입과 삭제 연산 뒤의 스플레이

스플레이 트리 (10)

- 스플레이 트리의 시간 복잡도
  - 각 연산(탐색, 삽입, 삭제, 조인, 분할)은 O(*log*n) 상환 시간에 수행할 수 있음
  - 상환 시간(amortized time)
    - 일련의 연산 수행에서 시간이 많이 걸리는 연산의 시간을 적게 걸리는 연산에 전가시킨 뒤의 시간
    - 개개 연산의 최악의 경우에 걸리는 시간이 짧아짐
  - m번의 삽입, 삭제 연산을 수행 → O(m*log*n) 상환 시간

2-3 트리 (1)

- **2-3 트리**
  - 차수가 2 또는 3인 탐색 트리 구조
  - 삽입과 삭제 연산이 AVL 트리보다 간단
  - 시간 복잡도 : O(logn)
  - 2-노드 구조 :

left	key_1	middle		

  - 3-노드 구조 :

left	key_1	middle	key_2	right

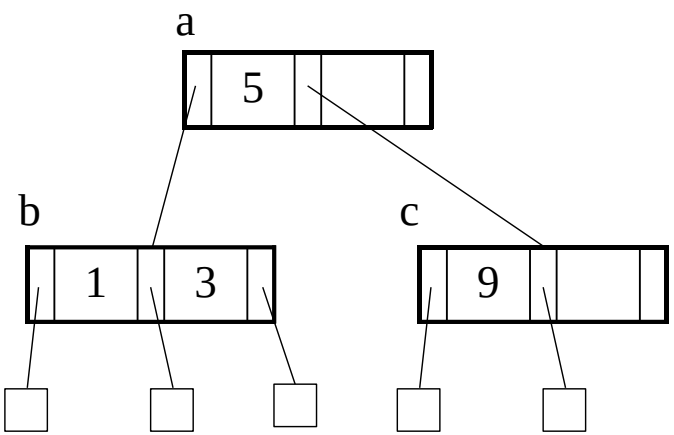
    - key1과 key2 : 키값
    - left, middle, right : 서브트리를 가리키는 링크

2-3 트리 (2)

- **2-3 트리의 성질**
  1) 각 노드는 2-노드 또는 3-노드이고, 2-노드는 하나의 키값을, 3-노드는 두 개의 키 값을 포함
  2) 2-노드 : left가 가리키는 서브트리의 키값 < key_1 < middle이 가리키는 서브트리의 키값
  3) 3-노드 : left가 가리키는 서브트리의 키값 < key_1 < middle이 가리키는 서브트리의 키값 < key_2 < right가 가리키는 서브트리의 키값
  4) 모든 외부 노드(external node) : 같은 레벨에 있음

2-3 트리 (3)

- **2-3 트리의 예**



  - a, c : 2-노드, b : 3-노드

- **높이가 h인 2-3 트리의 키수**
  - $2^{h+1}-1$과 $3^{h+1}-1$ 사이
- **n개의 키값을 가진 2-3 트리의 높이**
  - $\lceil \log_3(n+1)-1 \rceil$과 $\lceil \log_2(n+1)-1 \rceil$ 사이

2-3 트리에서의 탐색

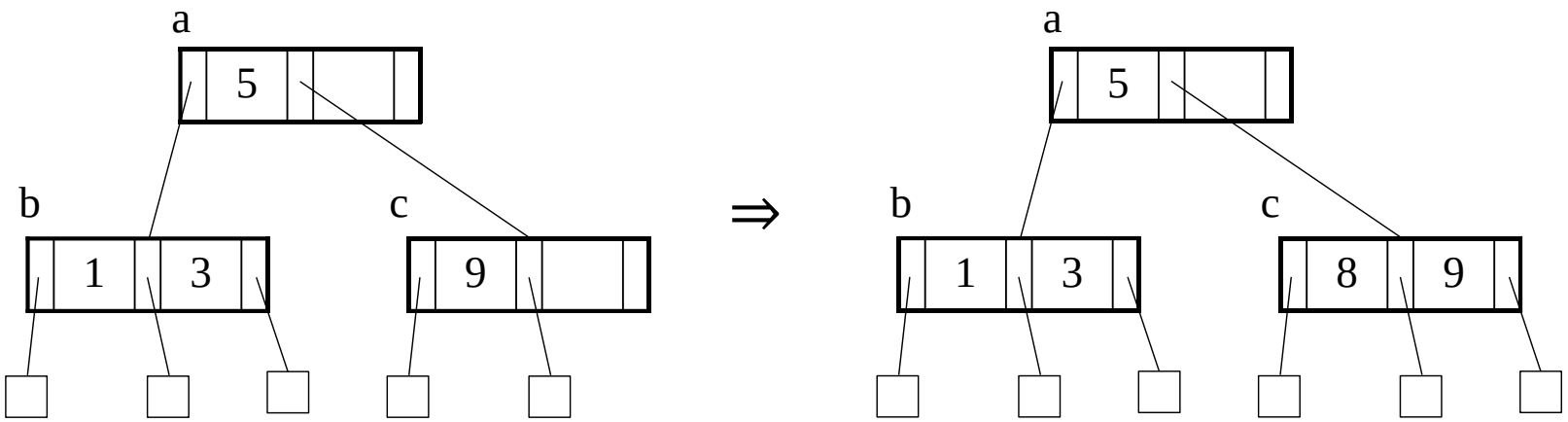
◆ 2-3 트리의 탐색 알고리즘

```
twoThreeSearch(x)
    for(p ← root; p; )          // root는 2-3 트리의 루트 노드
        switch (compare(p, x)) {
            case 1 : p ← p.left;     break;
            case 2 : p ← p.middle;     break;
            case 3 : p ← p.right;     break;
            case 4 : return p;     // x는 p의 키 중에 하나
        }
end twoThreeSearch()
```

- 함수 compare()는 주어진 키 x를 노드 p에 있는 키값들과 비교하여 그 결과에 따라 1,2,3 또는 4를 반환

2-3 트리에서의 삽입 (1)

- **2-노드에 삽입 (키 8을 삽입)**

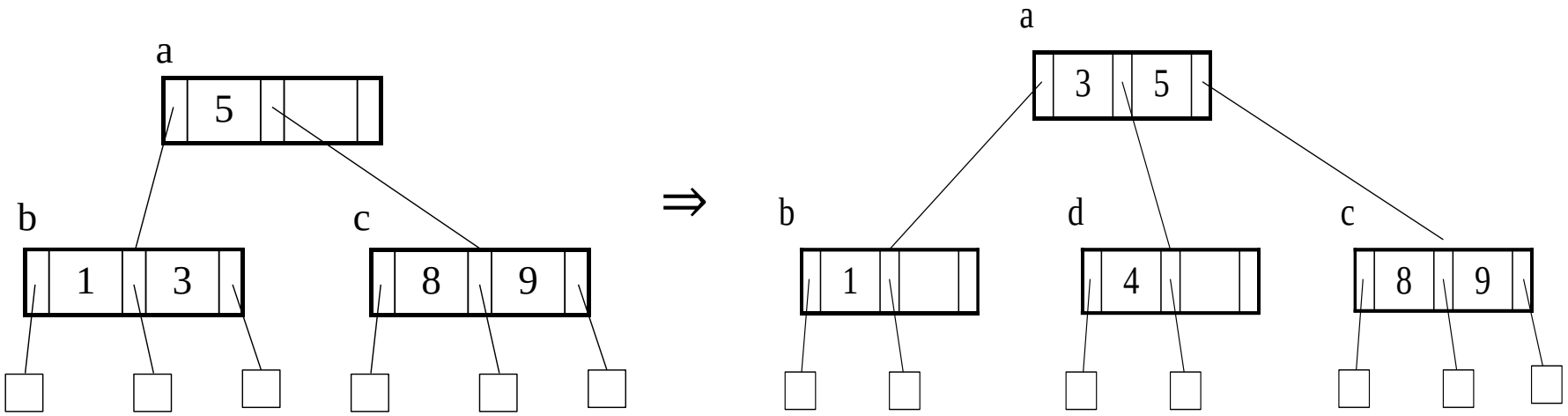


키 8의 삽입

- $key_1 < key_2$이므로 노드 c안에서 키값의 위치가 조정됨

2-3 트리에서의 삽입 (2)

- **3-노드에 삽입 (키 4를 삽입)**

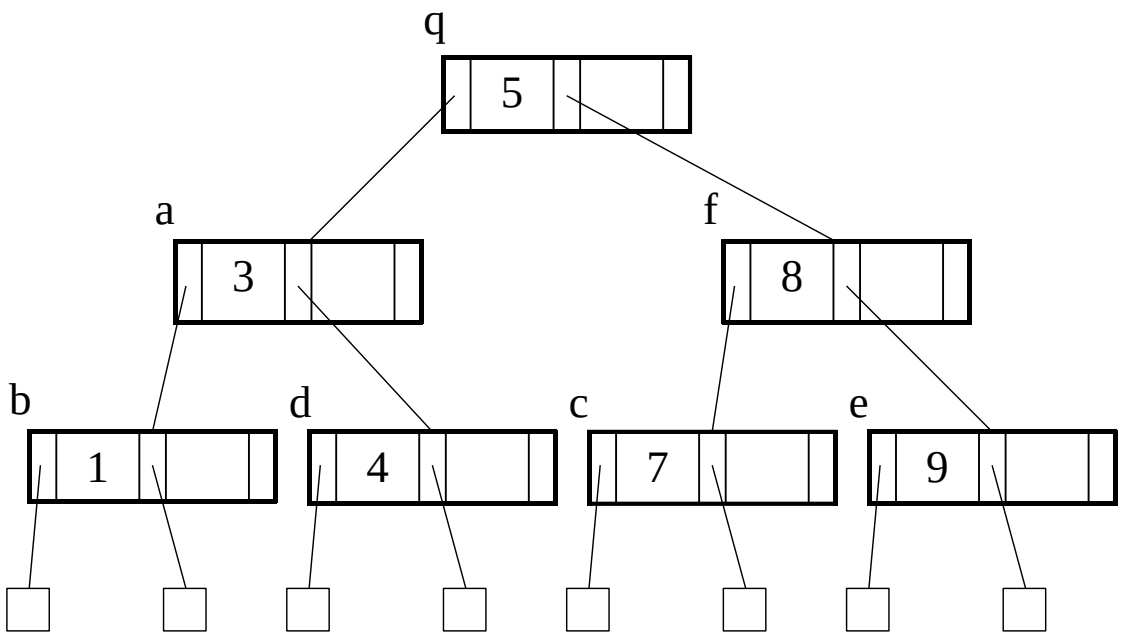


키 4의 삽입

  - 노드 b가 3-노드 → 새로운 노드 d가 필요
  - 노드 b의 두 키와 삽입하려는 키 중 가장 큰 키를 노드 d에 저장
  - 가장 작은 키는 b에 남기고, 중간 키값은 b의 부모 노드 a에 삽입
- **분할(split) : 3-노드의 키값을 새로운 노드와 재분배**

2-3 트리에서의 삽입 (3)

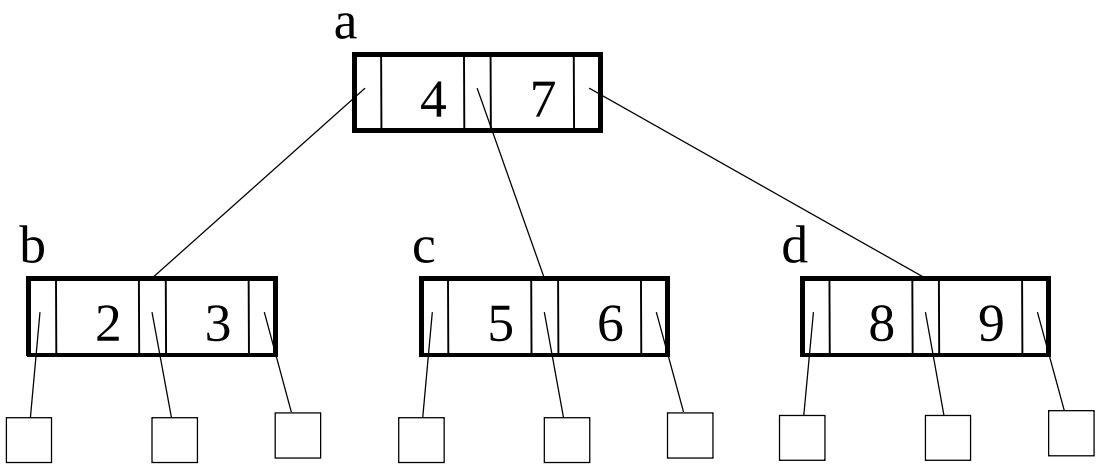
- 3-노드에 삽입 (키 7을 삽입)



키 7의 삽입

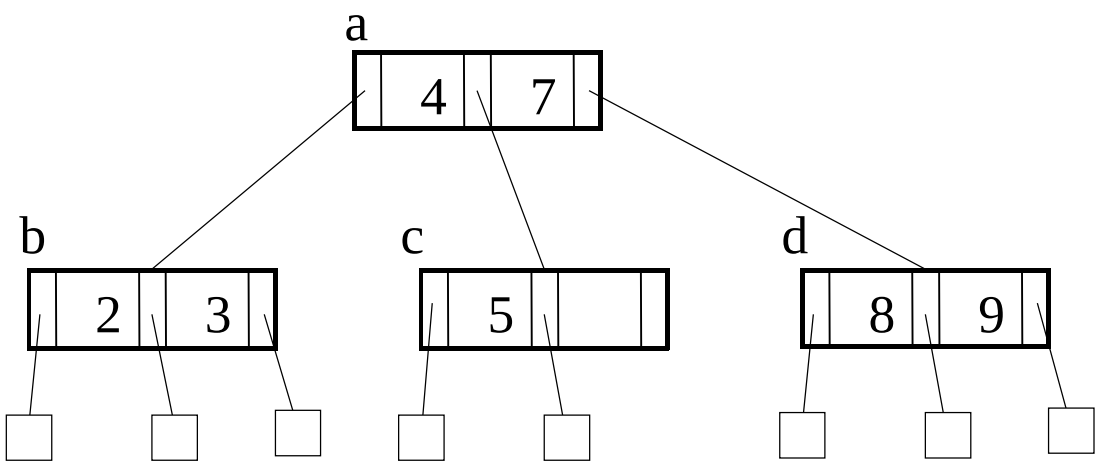
2-3 트리에서의 삭제 (1)

- 단말 노드의 삭제에 대해서만 고려
  - 단말 노드에 있지 않은 키를 삭제하는 경우 : 삭제된 키를 단말 노드에 있는 적절한 키로 대체
    - 단말 노드로부터의 삭제로 변환
    - 일반적으로 삭제될 키의 왼쪽 서브트리에서 가장 큰 키나 오른쪽 서브트리에서 가장 작은 키 중의 하나를 선택

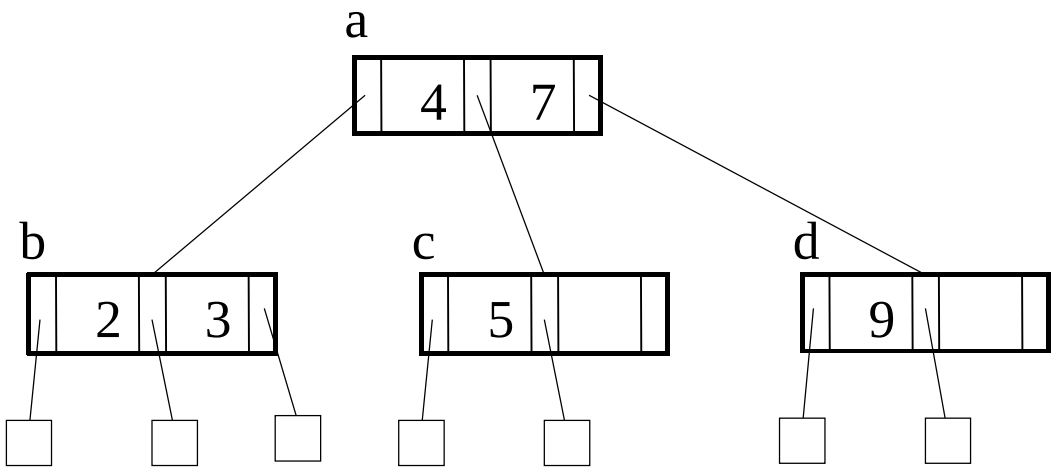


(a) 초기 2-3 트리

2-3 트리에서의 삭제 (2)

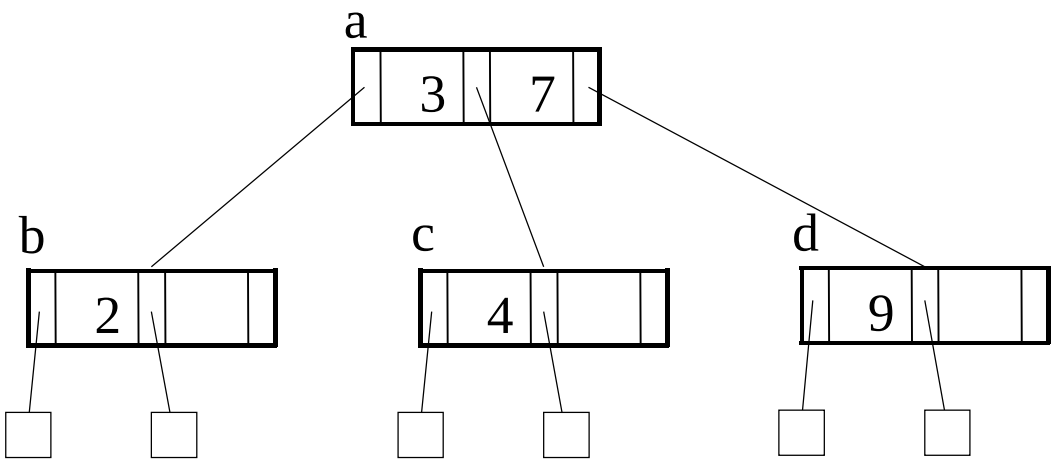


(b) 키 6의 삭제



(c) 키 8의 삭제

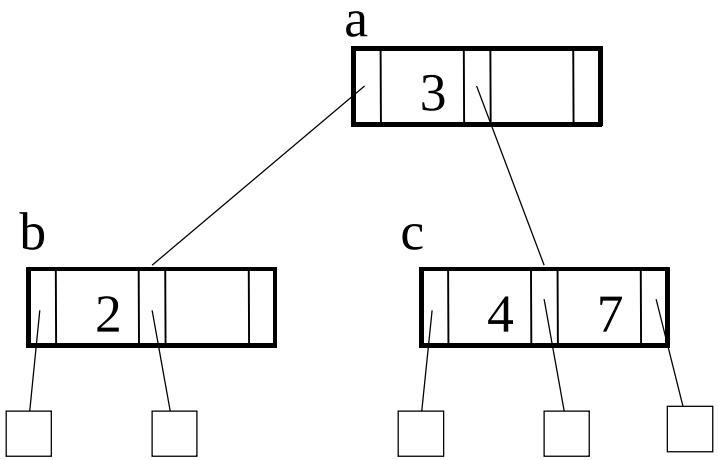
2-3 트리에서의 삭제 (3)



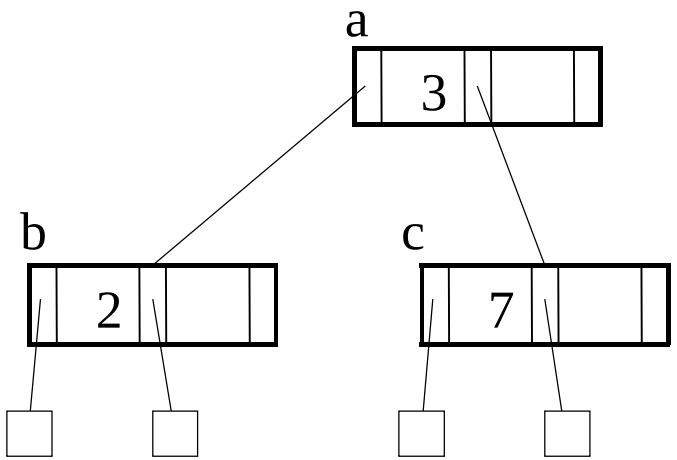
(d) 키 5의 삭제

- 회전(rotation)
  - 데이터를 이동하여 트리의 구조를 변형시키는 것

2-3 트리에서의 삭제(4)

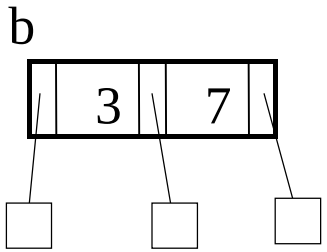


(e) 키 9의 삭제



(e) 키 4의 삭제

- 합병(merge)
  - 노드를 삭제하고 데이터를 결합시키는 연산



(g) 키 2의 삭제

2-3 트리에서의 삭제

2-3-4 트리 (1)

- **2-3-4 트리**
  - 트리의 노드가 2-노드, 3-노드 또는 4-노드로 구성된 트리
  - 2-3 트리가 확장된 트리

  - 2-노드 구조

left	key_1	leftMid				

  - 3-노드 구조

left	key_1	leftMid	key_2	rightMid		

  - 4-노드 구조

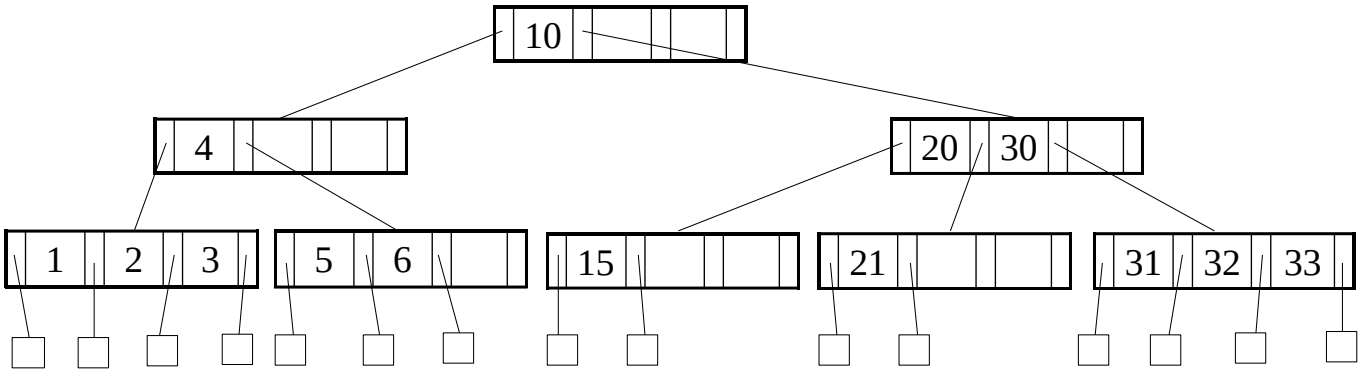
left	key_1	leftMid	key_2	rightMid	key_3	right

2-3-4 트리 (2)

- **2-3-4 트리의 성질**
  1) 각 노드는 2-노드, 3-노드 또는 3-노드이고, 2-노드는 하나의 키값, 3-노드는 두 개의 키 값. 4-노드는 3개의 키값을 포함
  2) 2-노드 : left가 가리키는 서브트리의 키값 < key_1 < leftMid가 가리키는 서브트리의 키값
  3) 3-노드 : left가 가리키는 서브트리의 키값 < key_1 < leftMid가 가리키는 서브트리의 키값 < key_2 < rightMid가 가리키는 서브트리의 키값
  4) 4-노드 : left가 가리키는 서브트리의 키값 < key_1 < leftMid가 가리키는 서브트리의 키값 < key_2 < rightMid가 가리키는 서브트리의 키값 < key_3 < right가 가리키는 서브트리의 키값
  5) 모든 외부 노드(external node) : 같은 레벨에 있음

2-3-4 트리 (3)

- **2-3-4 트리의 예**

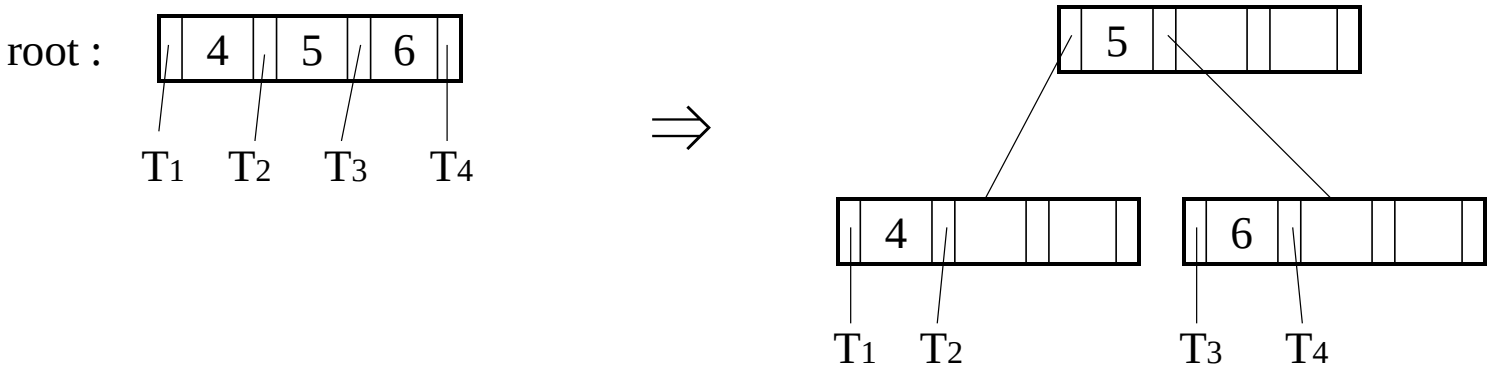


- **높이가 h인 2-3-4 트리의 키수**
  - $2^{h+1}-1$과 $4^{h+1}-1$ 사이
- **n개의 키값을 가진 2-3-4 트리의 높이**
  - $\lceil \log_4(n+1) \rceil - 1$ 과 $\lceil \log_2(n+1) \rceil - 1$ 사이
- **2-3-4 트리의 이점**
  - 루트에서 단말까지의 경로를 한번만 순회하면 됨
  - 2-3 트리는 삽입 또는 삭제를 위한 순회(루트→단말)와 분할과 합병의 영향으로 인한 순회(단말→루트)가 필요

2-3-4 트리에서의 삽입 (1)

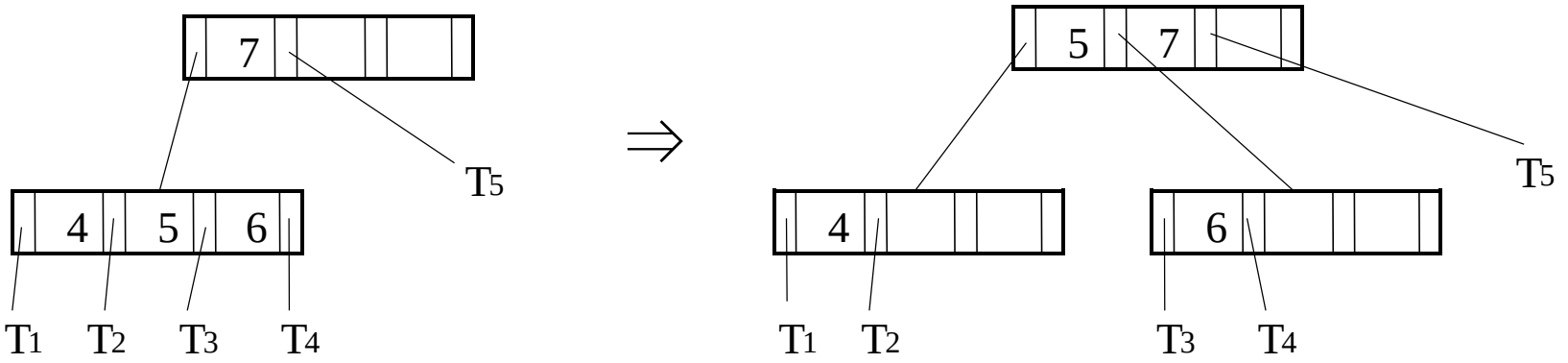
- **키를 삽입해야 할 단말 노드가 2-노드 또는 3-노드**
  - 간단하게 삽입만 하면 됨
- **키를 삽입해야 할 단말 노드가 4-노드**
  - 후진 분할(backward split)이 일어남
  - 후진 분할 연산을 방지하기 위하여 삽입 노드를 찾는 순회(루트->단말) 시에 4-노드를 만나면 미리 분할을 수행
    → 하향식 삽입(topdown insertion)
- **4-노드에 대한 분할의 경우**
  - 4-노드가 2-3-4 트리의 루트인 경우
  - 4-노드가 2-노드의 자식인 경우
    - 2-노드의 왼쪽 자식 · 왼쪽 중간 자식인 경우
  - 4-노드가 3-노드의 자식인 경우
    - 3-노드의 왼쪽 자식 · 왼쪽 중간 자식 · 오른쪽 자식인 경우

2-3-4 트리에서의 삽입 (2)

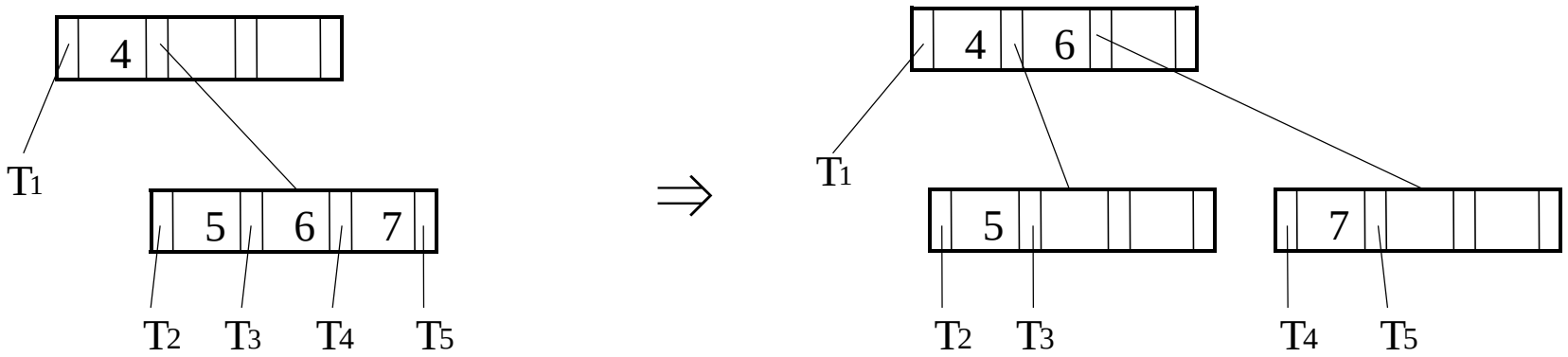


4-노드가 루트인 경우의 분할(Ti는 서브트리)

2-3-4 트리에서의 삽입(3)



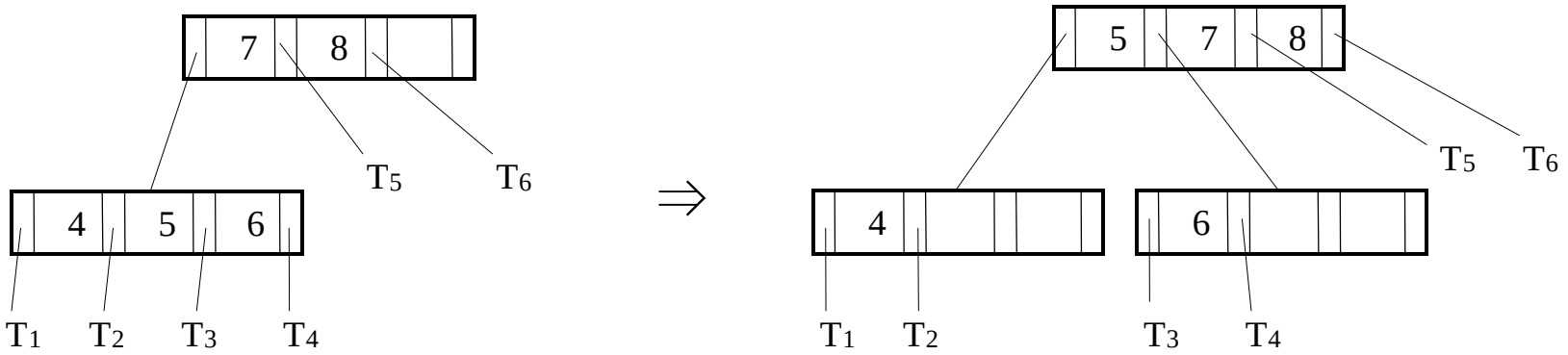
(a) 4-노드가 2-노드의 왼쪽 자식인 경우



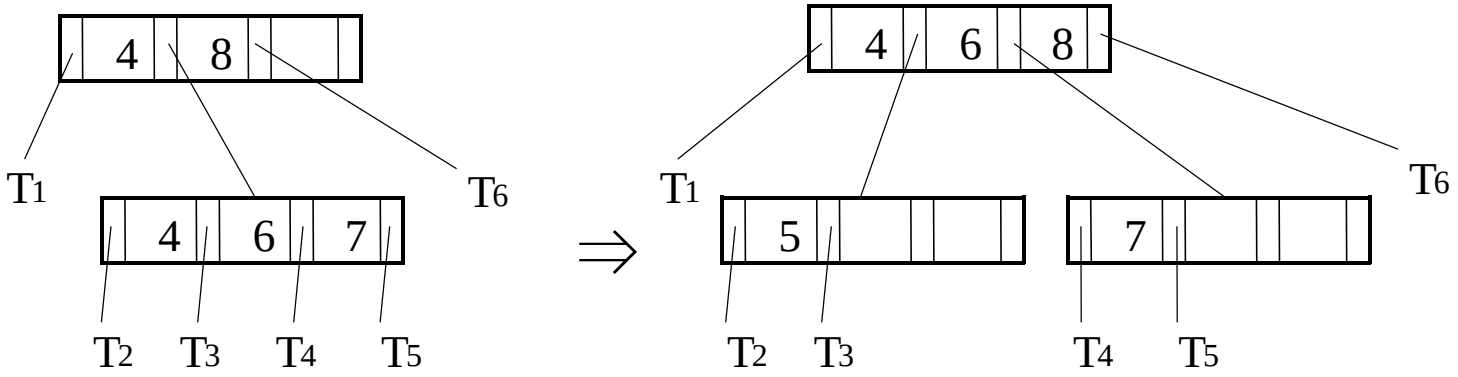
(b) 4-노드가 2-노드의 왼쪽 중간 자식인 경우

4-노드가 2-노드의 자식인 경우의 분할

2-3-4 트리에서의 삽입(4)



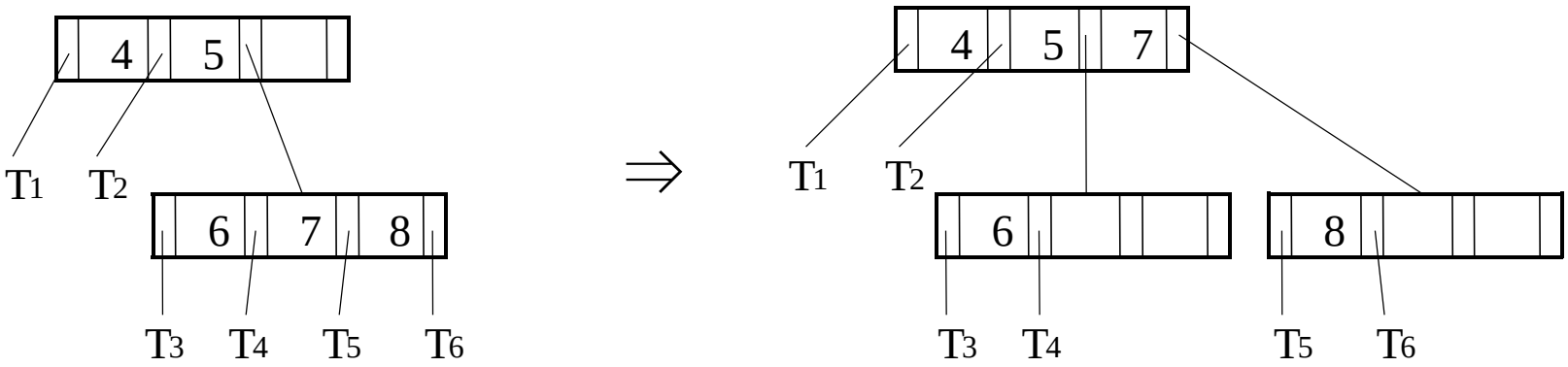
(a) 4-노드가 3-노드의 왼쪽 자식인 경우



(b) 4-노드가 3-노드의 왼쪽 중간 자식인 경우

4-노드가 3-노드의 자식인 경우의 분할

2-3-4 트리에서의 삽입(5)



(c) 4-노드가 3-노드의 오른쪽 자식인 경우

4-노드가 3-노드의 자식인 경우의 분할

2-3-4 트리에서의 삭제(1)

- 단말 노드의 키 삭제로 변환해서 수행

- 키를 삭제하려는 단말 노드가 3-노드 또는 4-노드
  - 트리의 재구성 작업이 필요 없음

- 키를 삭제하려는 단말 노드가 2-노드
  - 루트에서 목표 단말 노드로 내려가는 과정에서 미리 트리를 재구성

  → 하향식 삭제(topdown deletion)
    - 단말 노드에서 삭제를 수행한 뒤에 트리를 재구성할 필요가 없음
  - 탐색 과정에서 다음 노드로 이동할 때마다 그 노드가 3-노드 또는 4-노드가 되도록 변환

2-3-4 트리에서의 삭제(2)

- **p : 현재 탐색 노드, q : 다음 탐색 노드인 경우**
  - q는 p의 자식으로, 삭제할 키와 p에 있는 키들과의 관계

1) p가 단말 노드인 경우
   - 삭제할 키가 p에 있거나 트리에 없는 경우
   - 삭제가 실패하지 않는다고 가정하면 p가 루트인 경우에만 문제가 될 수 있음. 이 때에는 단순히 삭제한 뒤에 공백 트리가 됨

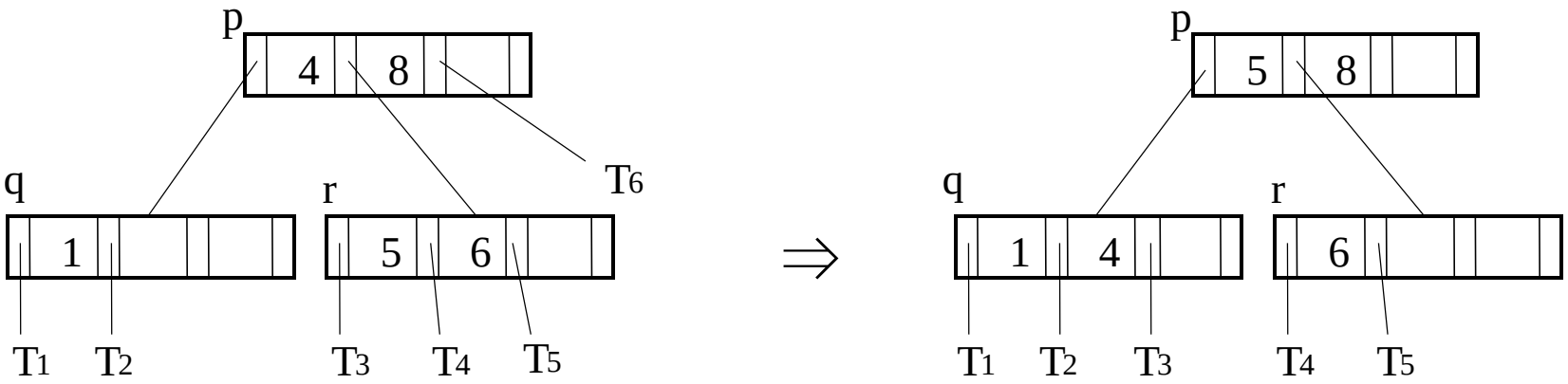
2) q가 2-노드가 아닌 경우
   - 탐색은 q 노드로 이동하고, 재구성 연산은 수행되지 않음

3) q와 가장 가까운 형제 노드 r도 2-노드인 경우
   - 노드 p가 2-노드 : p는 루트여야 하므로 4-노드 루트 형태로 합병, 트리 높이 하나 줄어듦
   - 노드 p가 3-노드, 4-노드 : 4-노드가 2-노드의 자식인 경우 분할과 4-노드가 3-노드 자식인 경우 분할의 역순으로 합병

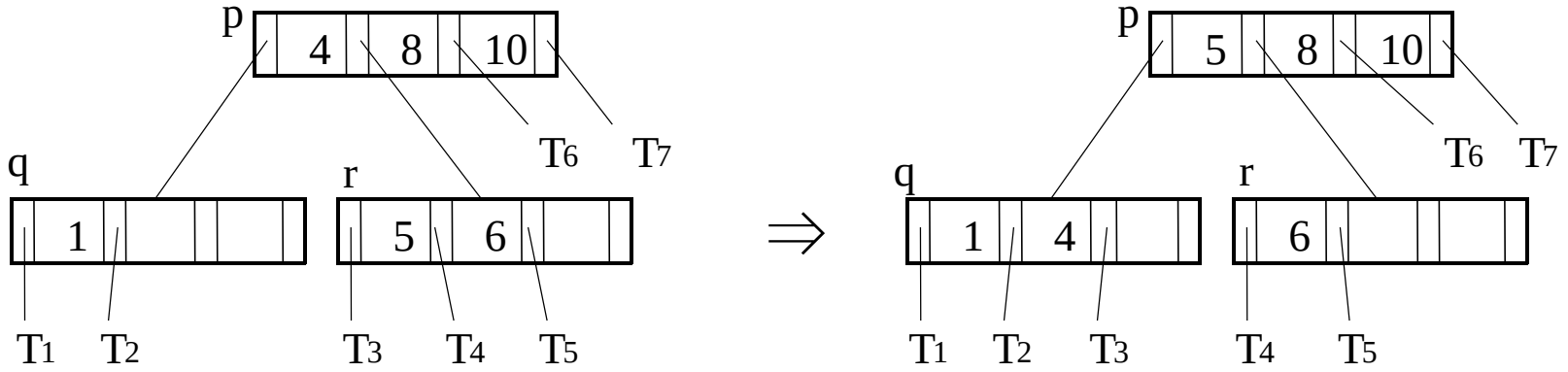
4) q가 2-노드, 가장 가까운 형제 노드 r이 3-노드인 경우
   - 다음 페이지 그림 참조

5) q가 2-노드, 가장 가까운 형제 노드 r이 4-노드인 경우
   - 노드 r이 3-노드인 경우와 유사

2-3-4 트리에서의 삭제(3)



(a) q가 3-노드의 왼쪽 자식인 경우



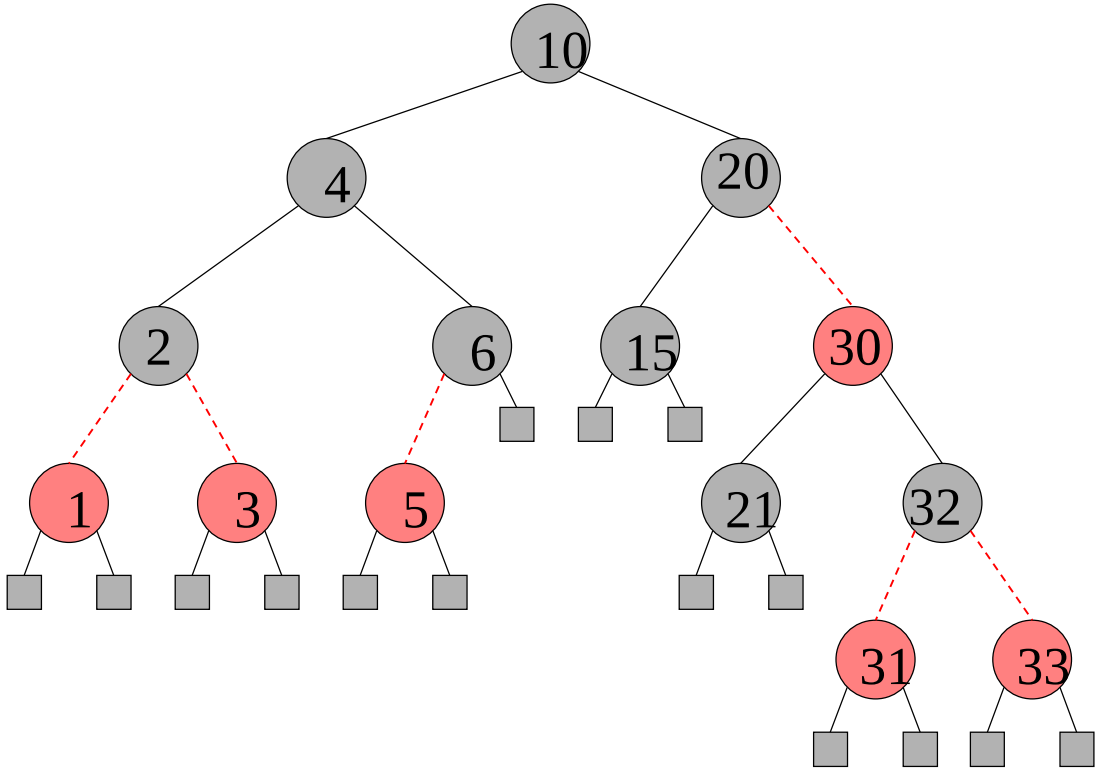
(b) q가 4-노드의 왼쪽 자식인 경우

가장 가까운 형제 노드(r)가 3-노드인 경우에 삭제 변환

레드-블랙 트리(1)

- **레드-블랙 트리(red-black tree)**
  - 노드 색깔이 레드나 블랙으로 된 이진 탐색 트리
  - 노드의 성질
    - N1. 루트나 외부 노드는 모두 블랙
    - N2. 루트에서 외부 노드까지 경로상에 2개 연속된 레드 노드 없음
    - N3. 루트에서 외부 노드까지 경로에 있는 블랙 노드 수 같음
  - 포인터의 성질
    - P1. 외부 노드를 연결하는 포인터는 블랙
    - P2. 루트에서 외부 노드까지 경로에 2개 연속된 레드 포인터 없음
    - P3. 루트에서 외부 노드까지 경로에 있는 블랙 포인터 수 같음
  - 포인터 색깔을 알면 그 노드 색깔을 알 수 있고, 노드 색깔을 알면 그 포인터 색깔을 알 수 있음

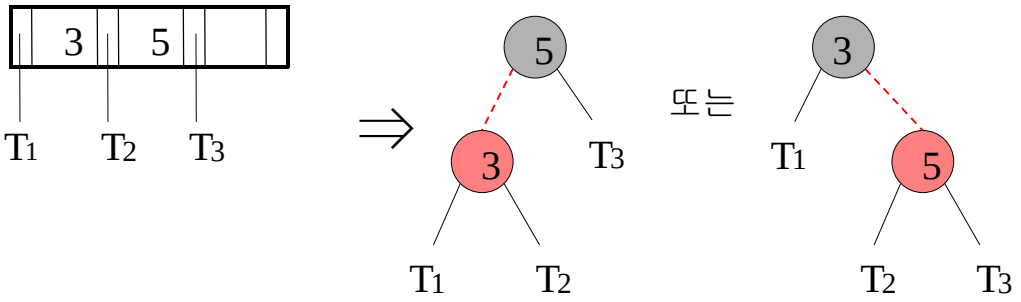
레드-블랙 트리(2)



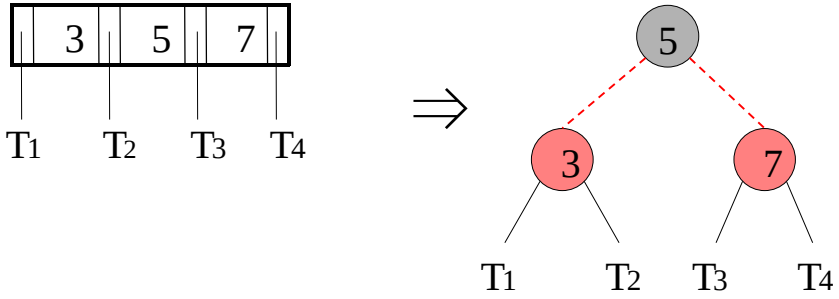
레드-블랙 트리 예(2-3-4 트리를 변환한 레드-블랙 트리)
(인쇄시 짙은 노드 : 레드 노드, 점선 : 레드 포인터)

레드-블랙 트리(3)

- **2-3-4트리를 레드-블랙 트리로 변환하는 방법**
  - 1) 2-노드는 그대로 레드-블랙 트리의 한 노드로 표현
  - 2) 3-노드는 2개의 노드와 하나의 레드 포인터로 표현
  - 3) 4-노드는 3개의 노드와 2개의 레드 포인터로 표현



3-노드를 2개의 레드-블랙 노드로 변환



4-노드를 3개의 레드-블랙 노드로 변환

레드-블랙 트리(4)

- 레드-블랙트리 복잡도
  - 외부 노드를 제외한 높이 : h
  - 내부 노드의 수 : n
  - 루트의 서열 : r

  - 트리의 노드 수 : $n \geq 2^r-1$
  - 트리의 높이 : $h \leq 2log(n+1)$
  - 탐색, 삽입, 삭제 연산의 시간복잡도 : $O(logn)$

레드-블랙 트리에서의 삽입(1)

- **레드-블랙 트리에서의 탐색**
  - 일반 이진 탐색 트리의 탐색 연산으로 수행
  - 탐색 시간 복잡도 : O(*log*n)

- **레드-블랙 트리에서의 삽입**
  - 일반 이진 트리에서 사용하는 연산과 비슷
  - 새로운 노드 삽입 시, 노드에 색깔을 지정하는 작업 추가
    - 트리가 공백 : 새로 삽입되는 노드는 루트가 되어 자연히 블랙
    - 트리가 공백이 아닌 경우, 블랙 노드가 추가되면 P3 위반
    - 트리가 공백이 아닌 경우, 레드 노드가 추가되면 P2 위반

레드-블랙 트리에서의 삽입(2)

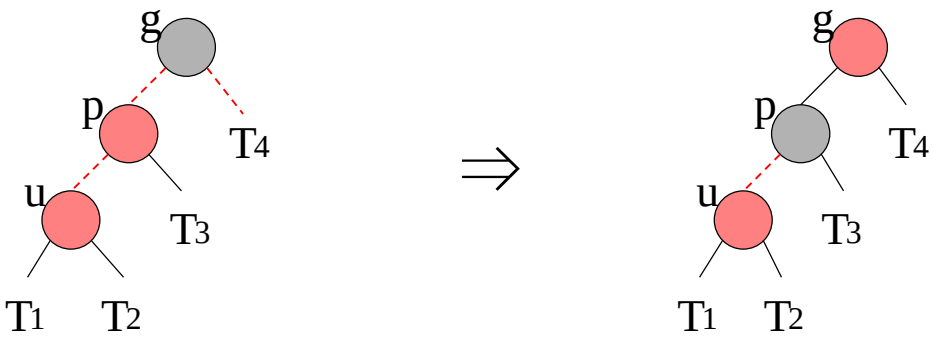
- **불균형(imbalance)**
  - 새로운 노드(u)가 레드이기 때문에 성질 P2를 위반한 경우
  - 노드 u, u의 부모 노드 p, u의 조부모 노드 g에 의한 유형

LLb : p는 g의 왼쪽 자식, u는 p의 왼쪽 자식, g의 오른쪽 자식이 블랙인 경우
LLr : p는 g의 왼쪽 자식, u는 p의 왼쪽 자식, g의 오른쪽 자식이 레드인 경우
LRb : p는 g의 왼쪽 자식, u는 p의 오른쪽 자식, g의 오른쪽 자식이 블랙인 경우
LRr : p는 g의 왼쪽 자식, u는 p의 오른쪽 자식, g의 오른쪽 자식이 레드인 경우

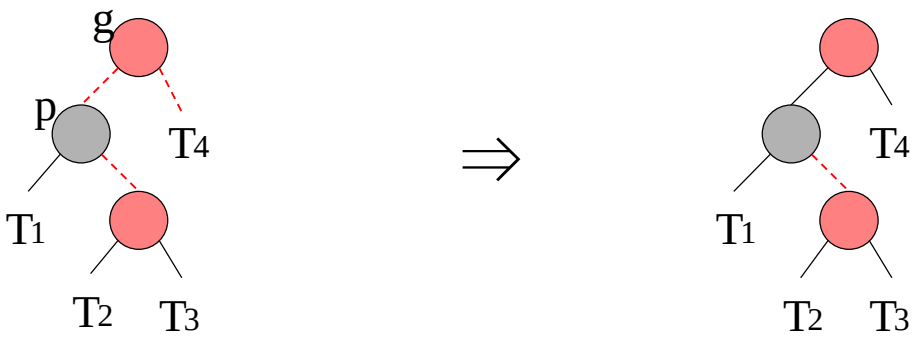
  - RRb, RRr, RLb, RLr도 위와 마찬가지

- **불균형의 처리**
  - XYr(X,Y는 L 또는 R) 타입 : 색깔만 변환시켜 처리
  - XYb 타입 : 색깔만 변환시키면 P2가 위반되므로, 회전을 시켜 처리

레드-블랙 트리에서의 삽입(3)



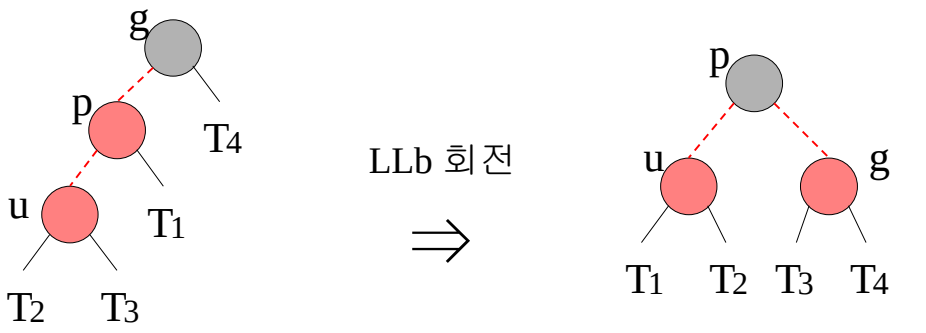
(a) LLr 불균형 처리



(b) LRr 불균형 처리

LLr과 LRr의 색깔 변환

레드-블랙 트리에서의 삽입(4)



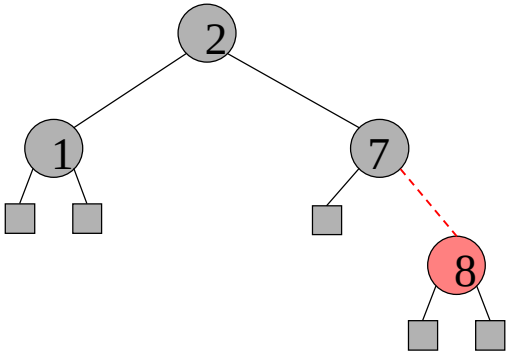
(a) LLb 불균형 (b) LLb 회전 후

(c) LRb 불균형 (d) LRb 회전 후

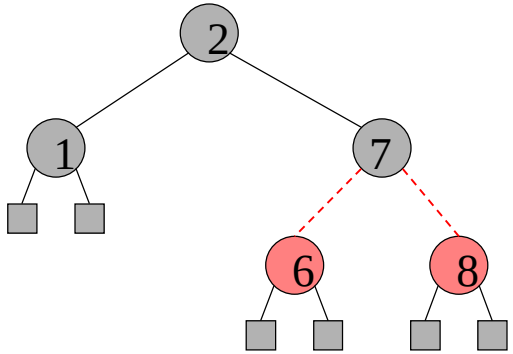
레드-블랙 트리의 삽입에 대한 LLb와 LRb 회전

레드-블랙 트리에서의 삽입(5)

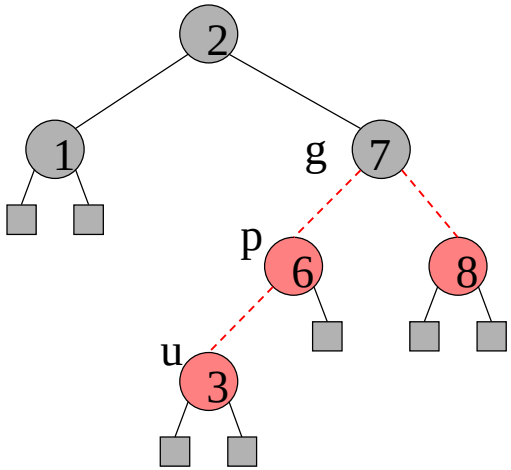
- 레드-블랙 트리에 삽입(6, 3, 5, 4) 예
  - 삽입에 따른 색깔 변환과 회전의 수행



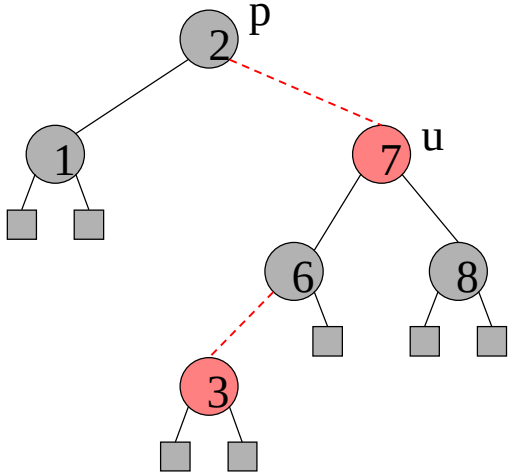
(a) 초기 레드-블랙 트리



(b) 키 6 삽입

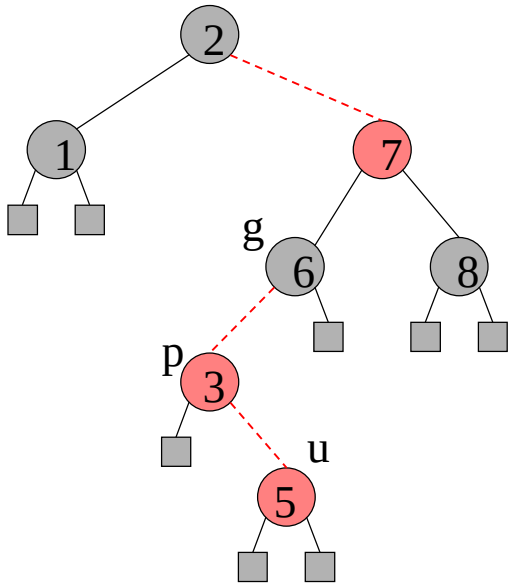


(c) 키 3 삽입

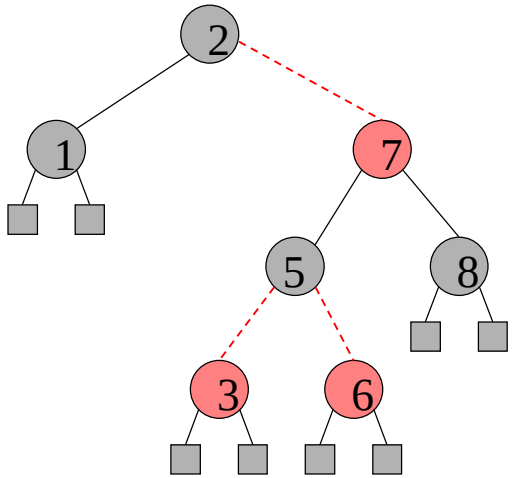


(d) LLr 색깔 변환

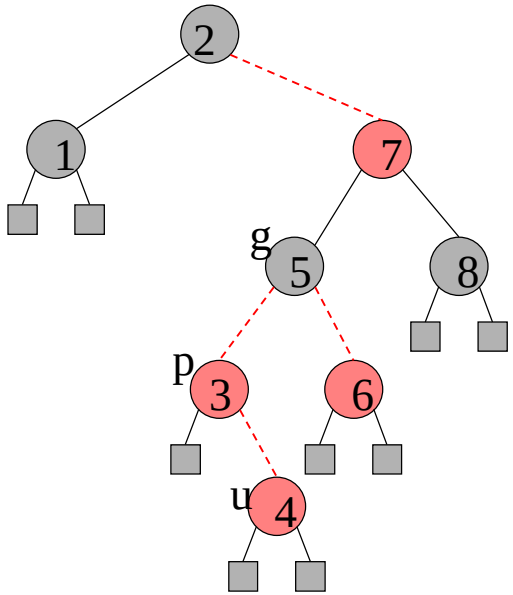
레드-블랙 트리에서의 삽입(6)



(e) 키 5 삽입

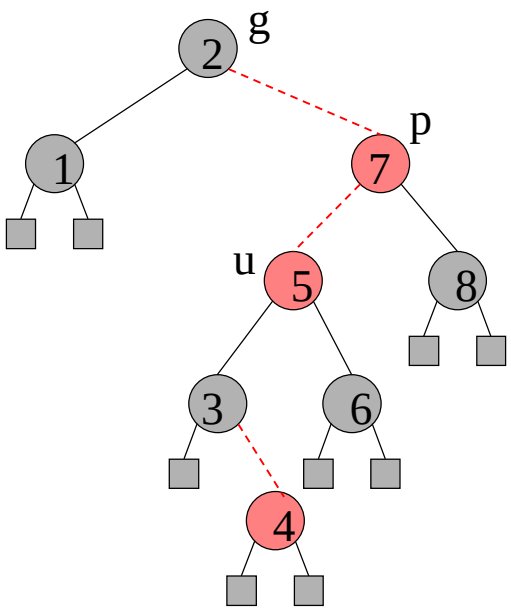


(f) LRb 회전

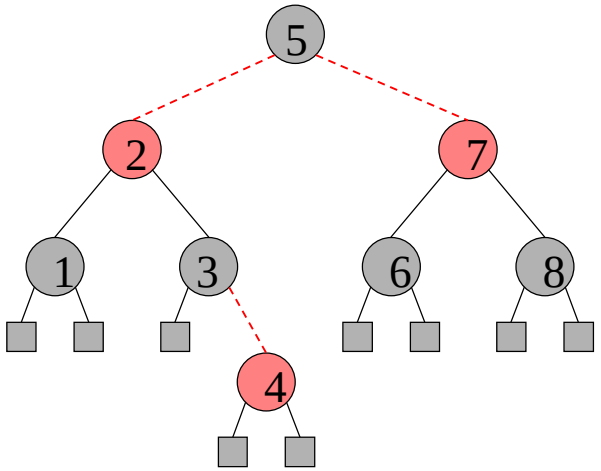


(g) 키 4 삽입

레드-블랙 트리에서의 삽입(7)



(h) LRr 색깔 변환



(i) RLb 회전

레드-블랙 트리에서의 삭제(1)

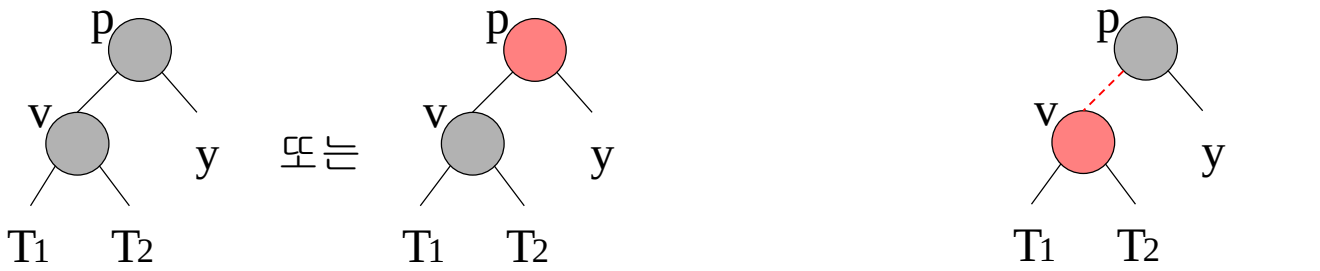
- **레드-블랙 트리에서의 삭제**
  - 일반 이진 탐색 트리에서와 같은 삭제 연산
  - 색깔 변환이나 단순 회전 수행 추가
  - 노드 색깔에 따른 삭제 처리
    - 레드 노드 삭제 : 경로의 블랙 노드 수에 영향을 주지 않음
    - 블랙 노드 삭제 : 경로의 블랙 노드 수를 감소시켜 P3 위반

- **불균형**
  - 삭제된 노드(y)가 블랙이기 때문에 성질 P3를 위반한 경우
  - 노드 y, y의 형제 v, y의 부모 p에 의한 유형
    - v가 블랙 노드 : Lb 또는 Rb 타입
    - v가 레드 노드 : Lr 또는 Rr 타입

레드-블랙 트리에서의 삭제(2)

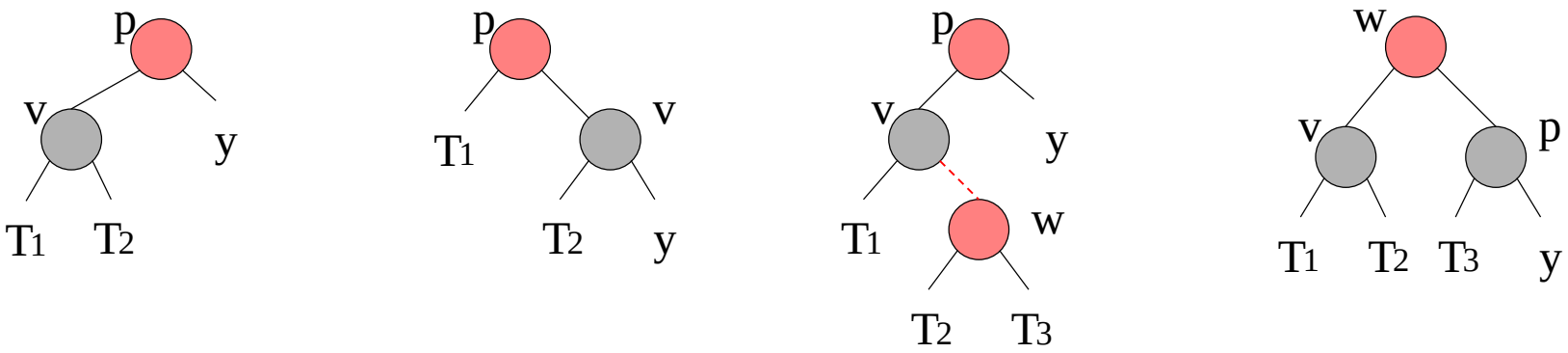
- 불균형의 처리
  - Rb, Lb 불균형: 서로 유사한 방법으로 처리.
    - Rb 불균형 : 삭제된 노드의 형제 노드 v의 레드 자식 수(0, 1, 2)에 따라 Rb0, Rb1, Rb2로 나누어 처리
    - Rb0 : 색깔 변환. p가 블랙인 경우에는 새로운 y로 하는 불균형 타입에 적절한 조치. p가 레드인 경우에는 색깔 변환만으로 종료
    - Rb1, Rb2 : 회전 시킴. Rb1은 다시 어느 쪽이 레드냐에 따라 2가지로 나뉨
  - Rr, Lr 불균형 : 서로 유사한 방법으로 처리.
    - Rr 불균형 : 형제 노드 v의 오른쪽 자식이 가지고 있는 레드 자식의 수(0, 1, 2)에 따라 Rr0, Rr1, Rr2로 나누어 처리
    - Rr1 : 어느 쪽 자식이 레드냐에 따라 2가지로 나뉨.
    - 모든 경우, 단순 회전으로 처리

레드-블랙 트리에서의 삭제(3)



(a) Rb0 불균형 (b) Rb0 색깔 변환

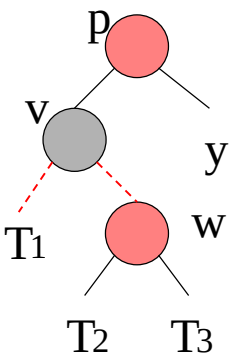
레드-블랙 삭제에 대한 Rb0 색깔 변환



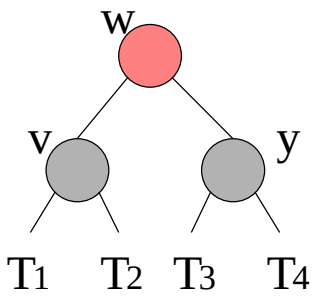
(a) Rb1(i) 불균형 (b) Rb1(i) 회전 (c) Rb1(ii) 불균형 (d) Rb1(ii) 회전

레드-블랙 삭제에 대한 Rb1과 Rb2 회전

레드-블랙 트리에서의 삭제(4)

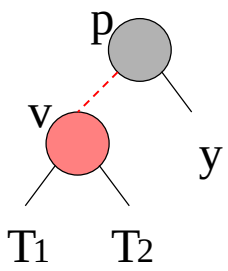


(e) Rb2 불균형

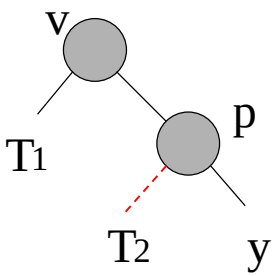


(f) Rb2 회전

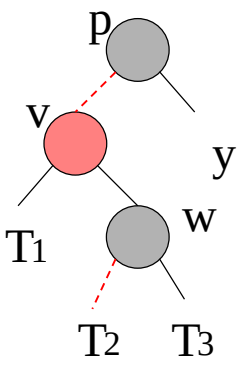
레드-블랙 삭제에 대한 Rb1과 Rb2 회전



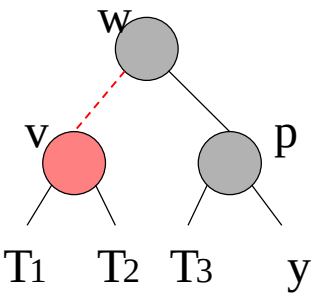
(a) Rr0 불균형



(b) Rr0 회전

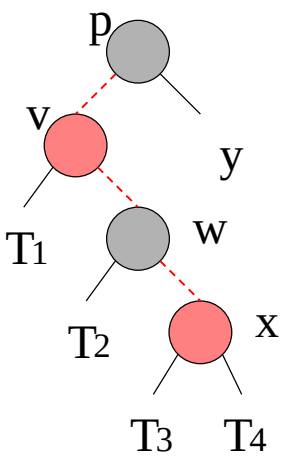


(c) Rr1(i) 불균형

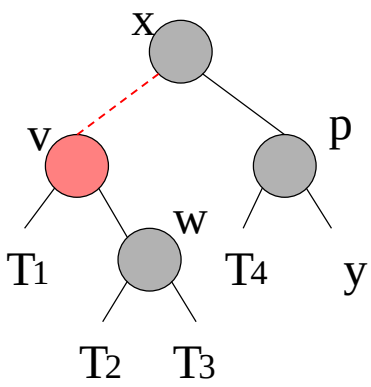


(d) Rb1(ii) 회전

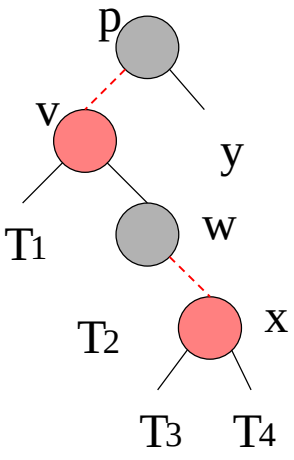
레드-블랙 트리에서의 삭제(5)



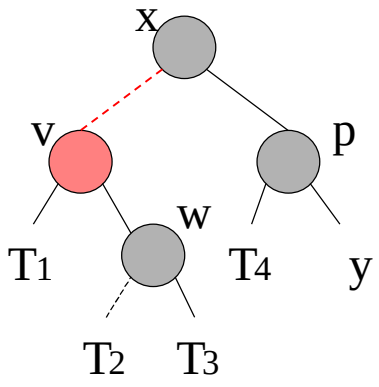
(e) Rb2 불균형



(f) Rr1(ii) 회전



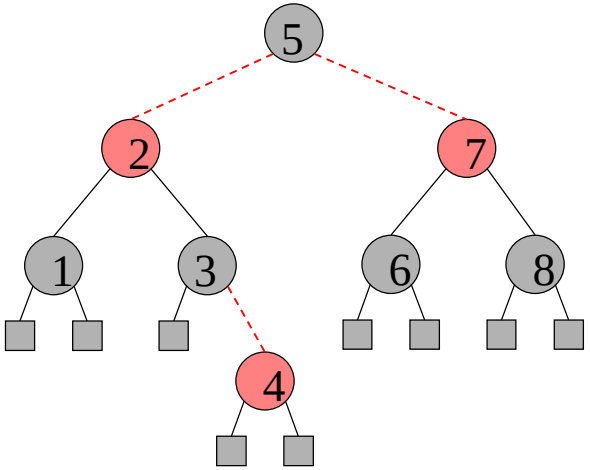
(g) Rr2 불균형



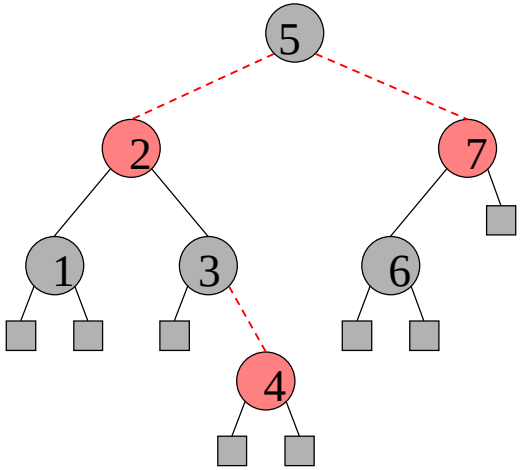
(h) Rr2 회전

레드-블랙 삭제에 대한 Rr0, Rr1, Rr2 회전

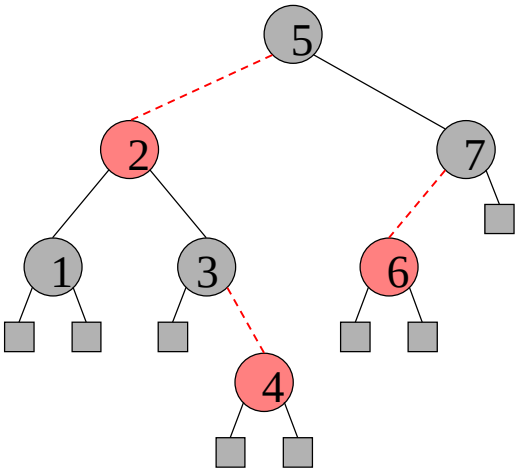
레드-블랙 트리에서의 삭제(6)



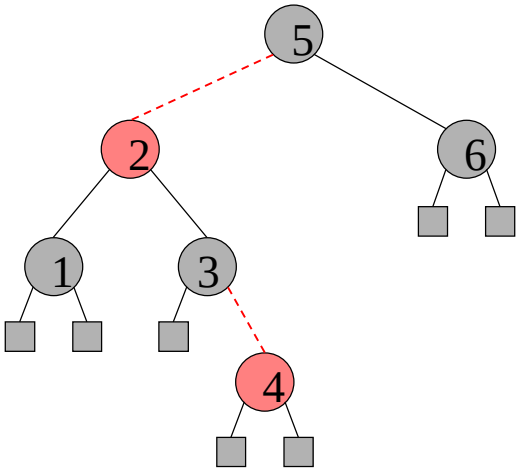
(a) 레드-블랙 트리



(b) 키 8 삭제

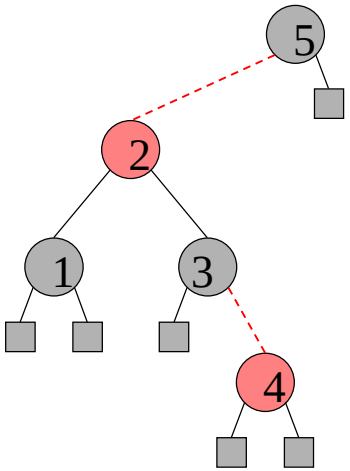


(c) Rb0 색깔 변환

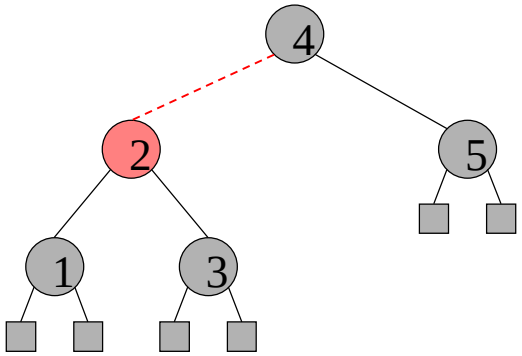


(d) 키 7 삭제

레드-블랙 트리에서의 삭제(7)



(e) 키 6 삭제



(f) Rr1(ii) 회전

레드-블랙 트리에서의 삭제 예