효율적인 이원 탐색 트리
(Efficient Binary Search Trees)

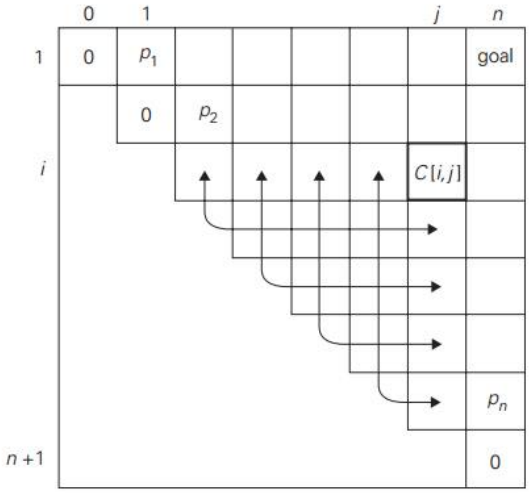
소프트웨어학과

이 의 종

최적 이원 탐색 트리
(Optimal Binary Search Trees)

In computer science, an optimal binary search tree (Optimal BST), sometimes called a weight-balanced binary tree, is a binary search tree which provides the smallest possible search time (or expected search time) for a given sequence of accesses (or access probabilities). Optimal BSTs are generally divided into two types: static and dynamic.

-- wikipedia



개요(Introduction)

- **Static Optimality Problem**
  - the tree cannot be modified after it has been constructed. In this case, there exists some particular layout of the nodes of the tree which provides the smallest expected search time for the given access probabilities.

10.1 Optimal Binary Search Tree
(Dynamic Programming Method 이용)

- **Dynamic Optimality Problem**
  - the tree can be modified at any time, typically by permitting tree rotations. The tree is considered to have a cursor starting at the root which it can move or use to perform modifications.

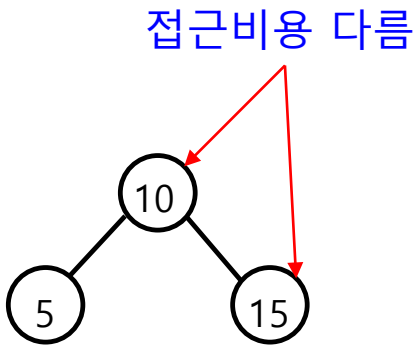
개요(Introduction)

- **정적 원소들의 집합에 대한 이원 탐색트리 구조**
  - 삽입이나 삭제는 하지 않고 탐색만 수행
  - 함수 iterSearch(프로그램 5.16 참고) 이용

```
element* iterSearch(treePointer tree, int key)
{
    while (tree) {
        if (key == tree->data) return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
    }
    return NULL;
}
```

  - while 루프가 탐색 비용을 결정

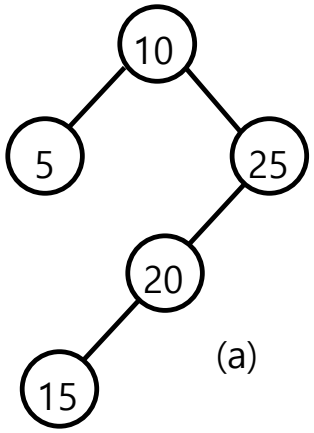
    → 노드의 레벨 수를 그 노드의 비용으로 이용



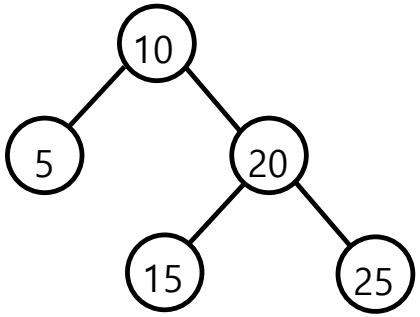
리스트(5,10,15)에서의 이원 탐색에 해당되는 이원 탐색 트리

예제(Example)

- **두 트리에 대해 최악과 평균 탐색시간 비교하면 (b)가 우수**



(a)



(b)

최악의 경우 탐색시간: (a) 4번, (b) 3번

성공적인 탐색에 필요한 평균: (a) 2.4번, (b) 2.2번

(1x1 + 2x2 + 1x3+1x4)/5 = 2.4

(1x1 + 2x2 + 2x3)/5 = 2.2

- **그러나, 각 노드에 접근 다음과 같은 확률정보를 반영 → (a) 우수**
  - 5, 10, 15, 20, 25가 각각 0.3, 0.3, 0.05, 0.05, 0.3의 확률

(a) 1.85 2x(0.3) + 1x(0.3) + 4x(0.05) + 3x(0.05) +2x(0.3) = 1.85

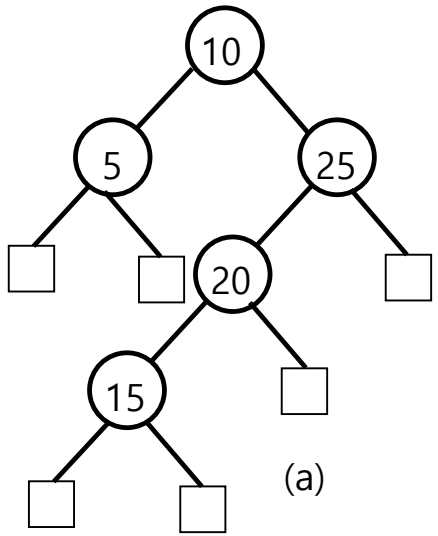
(b) 2.05 2x(0.3) + 1x(0.3) + 3x(0.05) + 2x(0.05) +3x(0.3) = 2.05

이원 탐색 트리 평가
(Cost Evaluation of Binary Search Trees)

– 특별한 사각형 노드(square node) 추가하면 유용

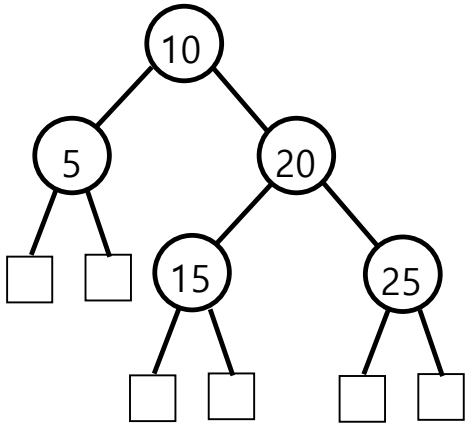
• 확장 이진 트리(extended binary tree)

external node
= fail node



(a)

확장 이진 트리



(b)

Why? → p.209

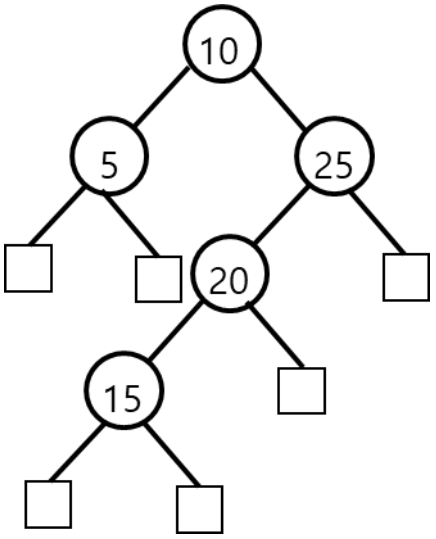
– 외부노드(실패노드) : n+1개의 사각노드

• 이원 탐색 트리에 없는 식별자 탐색 시 외부 노드에서 탐색이 끝남

– 내부노드 : 원래 노드

이원 탐색 트리 평가
(Cost Evaluation of Binary Search Trees)

- **외부경로길이(external path length)**
  - 루트로부터 각 외부 노드까지의 경로 길이의 합
- **내부경로길이(internal path length)**
  - 루트로부터 각 내부 노드까지의 경로 길이의 합



·내부경로길이(I) = 0+1+1+2+3 = 7

·외부경로길이(E) = 2+2+4+4+3+2 = 17

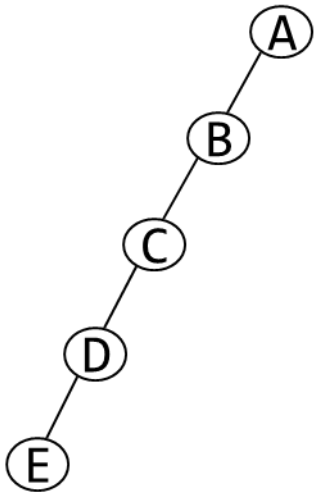
·I와 E의 관계(n개의 내부노드를 가질경우)

*E = I + 2n

E가 최대일 때 I도 최대

*internal node 하나 추가될 때 마다 external은 2만큼 증가

이원 탐색 트리 평가
(Cost Evaluation of Binary Search Trees)

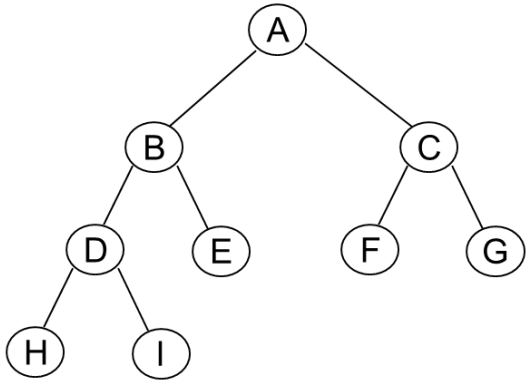


- I의 최소값
  - 최악의 경우: 트리가 편향될 때

$$I = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

  - 일반적인 경우:

$$0 + 2\times1 + 4\times2 + 8\times3 + \cdots\cdots +$$



  - 완전 이진 트리일 경우

$$I = \sum_{1\le i\le n} \lfloor \log_2 i \rfloor = O(n \log_2 n)$$

이원 탐색 트리 평가
(Cost Evaluation of Binary Search Trees)

- 정적 원소 집합을 이원탐색 트리로 표현
  - 탐색이 성공적일 때 비용

p_i : a_i를 탐색할 확률

$$\sum_{1 \le i \le n} p_i \times \text{level}(a_i)$$

  - 탐색이 실패했을 때 비용

탐색중인 원소가 E_i에 있을 확률 = q_i

$$\sum_{0 \le i \le n} q_i \times (\text{level}(\text{failure node } i) - 1)$$

  - 이원탐색트리의 총 비용

$$\sum_{1 \le i \le n} p_i \times \text{level}(a_i) + \sum_{0 \le i \le n} q_i \times (\text{level}(\text{failure node } i) - 1)$$

확률

$$\sum_{1 \le i \le n} p_i + \sum_{0 \le i \le n} q_i = 1$$

최소값 = 최적 이원탐색트리

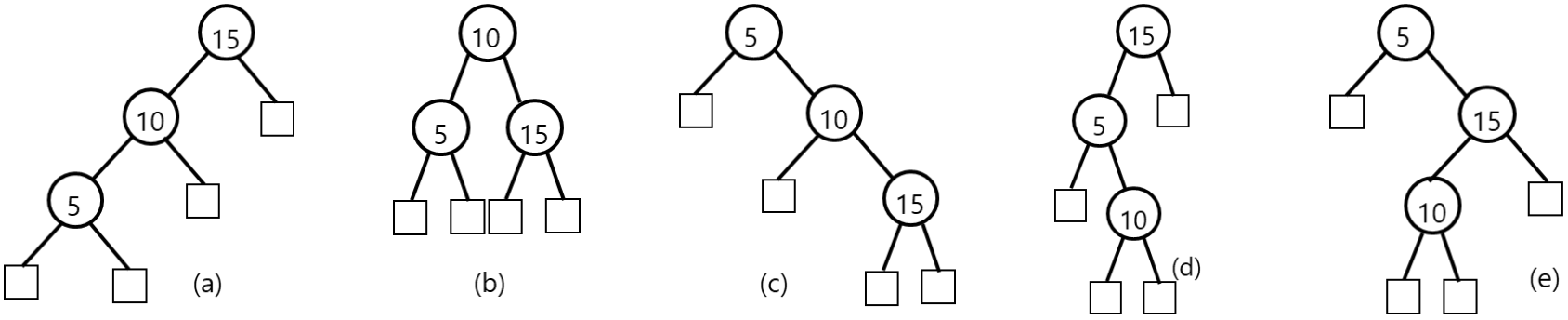
이원 탐색 트리 평가
(Cost Evaluation of Binary Search Trees)

- 예제: 키 집합 (a1, a2, a3)=(5, 10, 15)의 이원 탐색 트리

n개의 노드를 가진 서로 다른 이진 탐색 트리의 개수는?

catalan number

$$b_n = \frac{(2n)!}{n!\,(n+1)!}$$

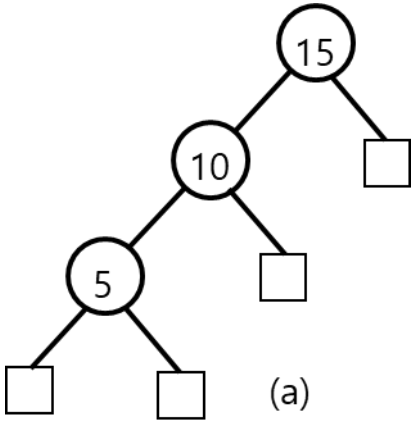


이원 탐색 트리 평가
(Cost Evaluation of Binary Search Trees)

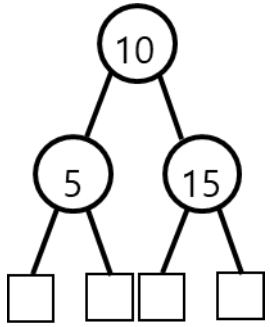
Static Optimality Problem

똑같은 확률 $p_i=q_i=1/7$일 경우

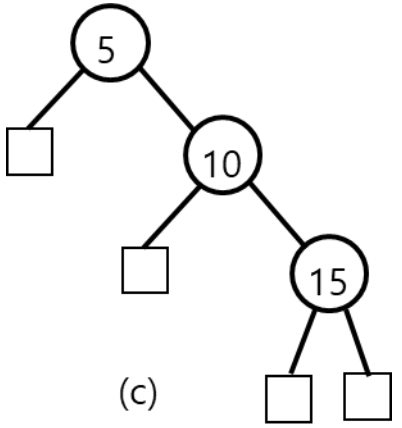
- 예제: 키 집합 (a1, a2, a3)=(5, 10, 15)의 이원 탐색 트리



1x1xp+1x1xq+2x1xp+2x1xq+
3x1xp+3x1xq+3x1xq = 15/7

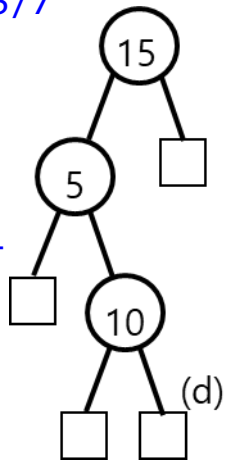


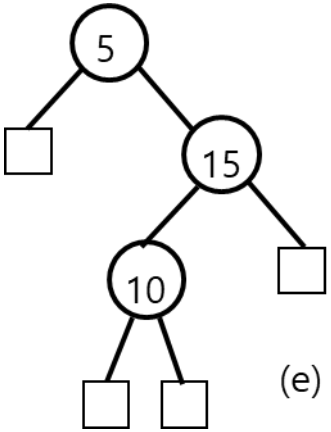
1x1xp+2x2xp+2x4xq = 13/7



1x1xp+1x1xq+2x1xp+2x1xq+
3x1xp+3x2xq = 15/7

1x1xp+1x1xq+2x1xp+2x1xq+
3x1xp+3x2xq = 15/7





1x1xp+1x1xq+2x1p+2x1xq+
3x1xp+3x2xq = 15/7

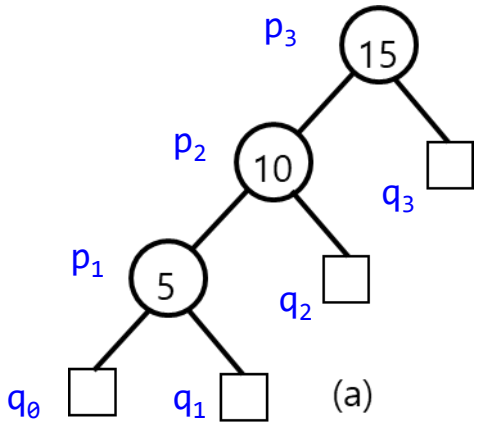
이원 탐색 트리 평가
(Cost Evaluation of Binary Search Trees)

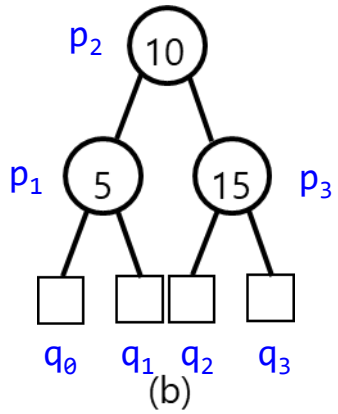
Static Optimality Problem

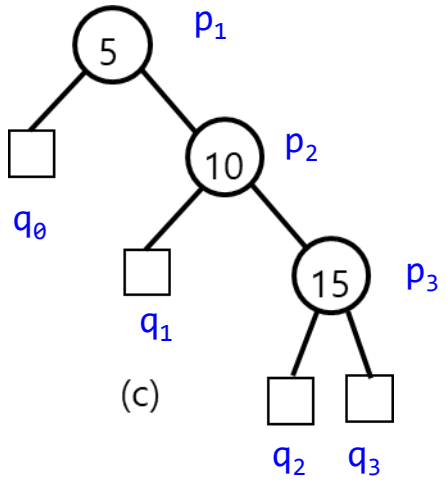
확률이 다른 경우

(a1, a2, a3)=(5, 10, 15)

p_1=0.5, p_2=0.1, p_3=0.05, q_0=0.15, q_1=0.1, q_2=0.05, q_3=0.05 확률을 갖는 경우



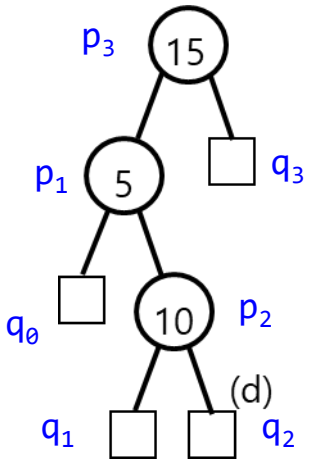


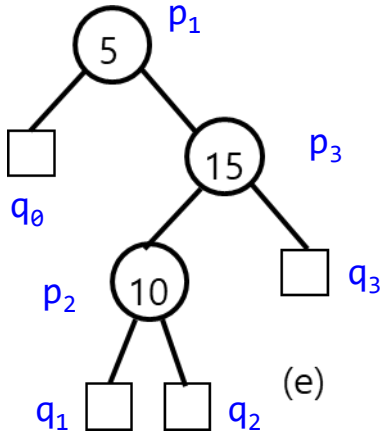


cost(b) = 1.9

cost(a) =
$1xp_3+1xq_3+2xp_2+2xq_2+$
$3xp_1+3xq_0+3xq_1$
= 0.05 + 0.05 +
2x0.1 + 2x0.05 +
3x0.5 + 3x0.15 +
3x0.1 = 2.65

cost(d) = 2.05





cost(c)=
$1xp_1+1xq_0+2xp_2+2xq_1+$
$3xp_3+3xq_2+3xq_3$
= 0.5 + 0.15 +
2x0.1 + 2x0.1 +
3x0.05 + 3x0.05 +
3x0.05 = **1.5**

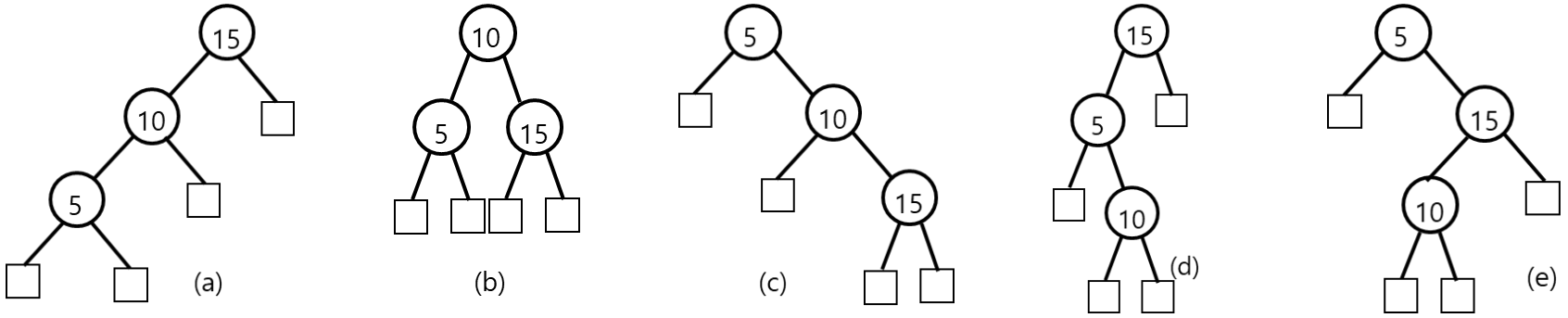
cost(e) = 1.6

이원 탐색 트리 평가
(Cost Evaluation of Binary Search Trees)

- **Time Complexity**

  n개의 노드를 가진 서로 다른 이진 트리의 개수는?

catalan number $\mathbf{b_n} = \frac{(2n)!}{n!(n+1)!} = 4^n/n^{3/2}$



$O(n(4^n/n^{3/2}))$ ← exponential time:
not reasonable time complexity

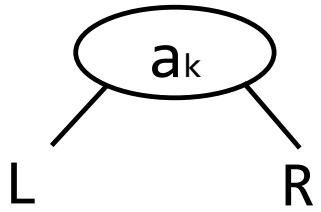
Dynamic Programming Approach 활용 →

Time Complexity를 Polynomial Time으로 Down

최적 이원 탐색 트리 찾기
(Determination of optimal binary search tree)

- $a_1 < a_2 < \ldots. < a_n$ n개의 키
- T_{ij} = 최적 이원 탐색 트리
  - 노드: $a_{i+1}, \ldots., a_j$, $i < j$
- w_{ij} = T_{ij}의 가중치, $w_{ii} = q_i$

$$w_{ij} = q_i + \sum_{k=i+1}^{j} (q_k + p_k)$$



최적 이원 탐색 트리 T_{ij}

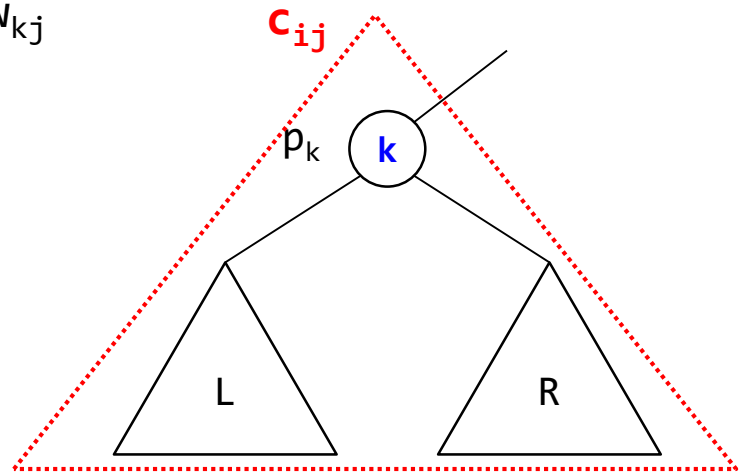
- c_{ij} = T_{ij}의 비용, $c_{ii} = 0$
- r_{ij} = T_{ij}의 root, $r_{ii} = 0$
- T_{ij}가 $a_{i+1}, \ldots., a_j$ 와 r_{ij}=k에 대한 최적 이원탐색트리 →
  - k는 $i < k \leq j$를 만족
- T_{0n} = $a_1, \ldots., a_n$ 노드를 갖는 최적 이원 탐색 트리
  - c_{0n} = 비용, r_{0n} = root, w_{0n}= 가중치

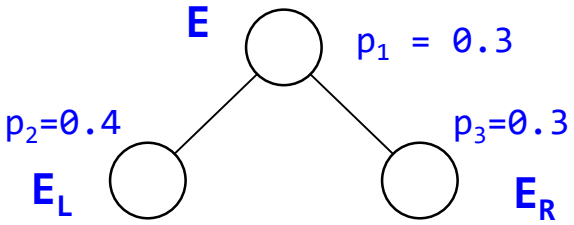
최적 이원 탐색 트리 찾기
(Determination of optimal binary search tree)

- $c_{ij} = p_k + cost(L) + cost(R) + weight(L) + weight(R)$

  $= p_k + c_{i,k-1} + c_{kj} + w_{i,k-1} + w_{kj}$

  $= w_{ij} + c_{i,k-1} + c_{kj}$





$cost(E) = 1x0.3 + 2x0.4 + 2x0.3$

$= 1x0.3 + (1+1)x0.4 + (1+1)x0.3$

$= p_1 + 1xp_2 + 1xp_3 + 1xp_2 + 1xp_3$

$= p_1 + cost(E_L) + cost(E_R) + weight(E_L) + weight(E_R)$

$= cost(E_L) + cost(E_R) + p_1 + weight(E_L) + weight(E_R)$

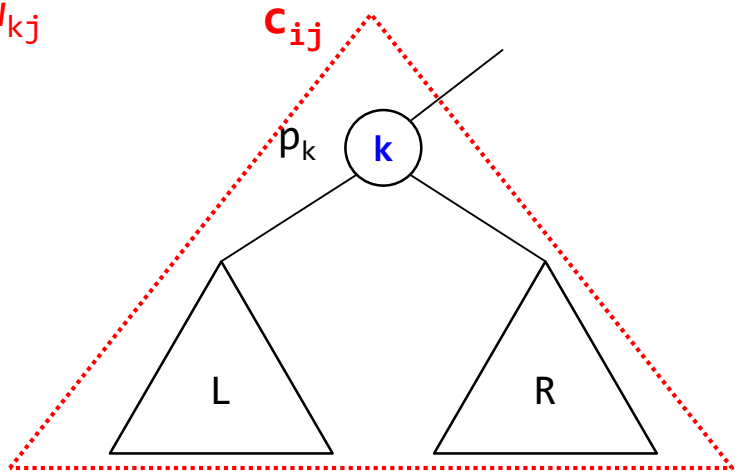
$= cost(E_L) + cost(E_R) + weight(E)$

최적 이원 탐색 트리 찾기
(Determination of optimal binary search tree)

- $c_{ij} = p_k + cost(L) + cost(R) + weight(L) + weight(R)$

  $= p_k + c_{i,k-1} + c_{kj} + w_{i,k-1} + w_{kj}$

  $= w_{ij} + c_{i,k-1} + c_{kj}$



$$w_{ij} + c_{i,k-1} + c_{kj} = \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{lj}\}$$

$$c_{i,k-1} + c_{kj} = \min_{i<l\le j}\{c_{i,l-1} + c_{lj}\}$$

최적 이원 탐색 트리 찾기

(Determination of optimal binary search tree)

- 예제: $n = 4$, $(a_1, a_2, a_3, a_4) = (10, 15, 20, 25)$
  - $(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$
  - $(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$
  - $w_{01} = p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 8$
  - $c_{01} = w_{01} + \min\{c_{00} + c_{11}\} = 8$
  - $r_{01} = 1$
  - $w_{12} = p_2 + w_{11} + w_{22} = p_2 + q_2 + w_{11} = 7$
  - $c_{12} = w_{12} + \min\{c_{11} + c_{22}\} = 7$
  - $r_{12} = 2$
  - $w_{23} = p_3 + w_{22} + w_{33} = p_3 + q_3 + w_{22} = 3$
  - $c_{23} = w_{23} + \min\{c_{22} + c_{33}\} = 3$
  - $r_{23} = 3$
  - $w_{34} = p_4 + w_{33} + w_{44} = p_4 + q_4 + w_{33} = 3$
  - $c_{34} = w_{34} + \min\{c_{33} + c_{44}\} = 3$
  - $r_{34} = 4$

p_i, q_i : original probability x 16

$$c_{ij} = w_{ij} + c_{i,k-1} + c_{kj}$$
$$= \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{lj}\}$$
$$= w_{ij} + \min_{i<l\le j}\{c_{i,l-1} + c_{lj}\}$$

$$w_{ij} = q_i + \sum_{k=i+1}^{j}(q_k + p_k)$$

$w_{ii} = q_i$

최적 이원 탐색 트리 찾기
(Determination of optimal binary search tree)

$(a_1, a_2, a_3, a_4) = (10, 15, 20, 25)$

$p_1 = 3,\ p_2 = 3,\ p_3 = 1,\ p_4 = 1$

$q_0 = 2,\ q_1 = 3,\ q_2 = 1,\ q_3 = 1,\ q_4 = 1$

$c_{ij} = p_k + c_{i,k-1} + c_{kj} + w_{i,k-1} + w_{kj}$

$= w_{ij} + c_{i,k-1} + c_{kj}$

	0	1	2	3	4
0	$w_{00} = 2(q_0)$ $c_{00} = 0$ $r_{00} = 0$	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$
1		$w_{11} = 3(q_1)$ $c_{11} = 0$ $r_{11} = 0$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$
2			$w_{22} = 1(q_2)$ $c_{22} = 0$ $r_{22} = 0$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$
3				$w_{33} = 1(q_3)$ $c_{33} = 0$ $r_{33} = 0$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$
4					$w_{44} = 1(q_4)$ $c_{44} = 0$ $r_{44} = 0$

$c[0,0] = w_{00}$

$c[0,1] = w_{01} + \min\{c[0,0] + c[1,1]\}$

$= w_{00} + w_{11} + p_1 + \min\{0+0\}$

$= 2 + 3 + 3 = 8$

$c[1,2] = w_{12} + \min\{c[1,1] + c[2,2]\}$

$= w_{11} + w_{22} + p_2 + \min\{0+0\}$

$= 3 + 1 + 3 = 7$

$c[2,3] = w_{23} + \min\{c[2,2] + c[3,3]\}$

$= w_{22} + w_{33} + p_3 + \min\{0+0\}$

$= 1 + 1 + 1 = 3$

$c[3,4] = w_{34} + \min\{c[3,3] + c[4,4]\}$

$= w_{33} + w_{44} + p_4 + \min\{0+0\}$

$= 1 + 1 + 1 = 3$

최적 이원 탐색 트리 찾기
(Determination of optimal binary search tree)

$(a_1, a_2, a_3, a_4) = (10, 15, 20, 25)$

$p_1 = 3,\ p_2 = 3,\ p_3 = 1,\ p_4 = 1$

$q_0 = 2,\ q_1 = 3,\ q_2 = 1,\ q_3 = 1,\ q_4 = 1$

	0	1	2	3	4
0	$w_{00} = 2(q_0)$ $c_{00} = 0$ $r_{00} = 0$	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$
1		$w_{11} = 3(q_1)$ $c_{11} = 0$ $r_{11} = 0$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$
2			$w_{22} = 1(q_2)$ $c_{22} = 0$ $r_{22} = 0$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$
3				$w_{33} = 1(q_3)$ $c_{33} = 0$ $r_{33} = 0$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$
4					$w_{44} = 1(q_4)$ $c_{44} = 0$ $r_{44} = 0$

$c[0,2] = w_{02} + \min\{c[0,0] + c[1,2],\ c[0,1] + c[2,2]\}$

$= w_{00} + w_{12} + p_1 + \min\{0+7,\ 8+0\}$

$= 2 + 7 + 3 + 7 = 19$

w_{02}

$w_{02} = w_{01} + w_{22} + p_2$

$= 8 + 1 + 3 = 12$

$c[1,3] = w_{13} + \min\{c[1,1] + c[2,3],\ c[1,2] + c[3,3]\}$

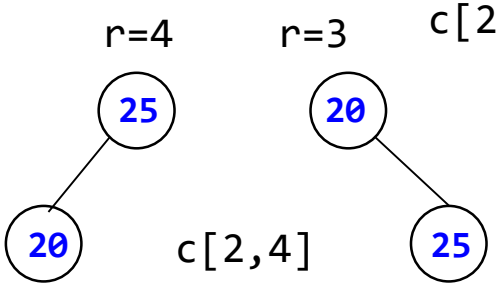
$= w_{11} + w_{23} + p_2 + \min\{0+3,\ 7+0\}$

$= 3 + 3 + 3 + 3 = 12$

$c[2,4] = w_{24} + \min\{c[2,2] + c[3,4],\ c[2,3] + c[4,4]\}$

$= w_{22} + w_{34} + p_3 + \min\{0+3,\ 3+0\}$

$= 1 + 3 + 1 + 3 = 8$



최적 이원 탐색 트리 찾기
(Determination of optimal binary search tree)

$(a_1, a_2, a_3, a_4) = (10, 15, 20, 25)$

$p_1 = 3,\ p_2 = 3,\ p_3 = 1,\ p_4 = 1$

$q_0 = 2,\ q_1 = 3,\ q_2 = 1,\ q_3 = 1,\ q_4 = 1$

	0	1	2	3	4
0	$w_{00} = 2(q_0)$ $c_{00} = 0$ $r_{00} = 0$	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$
1		$w_{11} = 3(q_1)$ $c_{11} = 0$ $r_{11} = 0$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$
2			$w_{22} = 1(q_2)$ $c_{22} = 0$ $r_{22} = 0$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$
3				$w_{33} = 1(q_3)$ $c_{33} = 0$ $r_{33} = 0$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$
4					$w_{44} = 1(q_4)$ $c_{44} = 0$ $r_{44} = 0$

$$c[0,3] = w_{03} + \min\{c[0,0] + c[1,3],\ c[0,1] + c[2,3],\ c[0,2] + c[3,3]\}$$
$$= w_{00} + w_{13} + p_1 + \min\{0+12,\ 8+3,\ 19+0\}$$
$$= 2 + 9 + 3 + 11 = 25$$

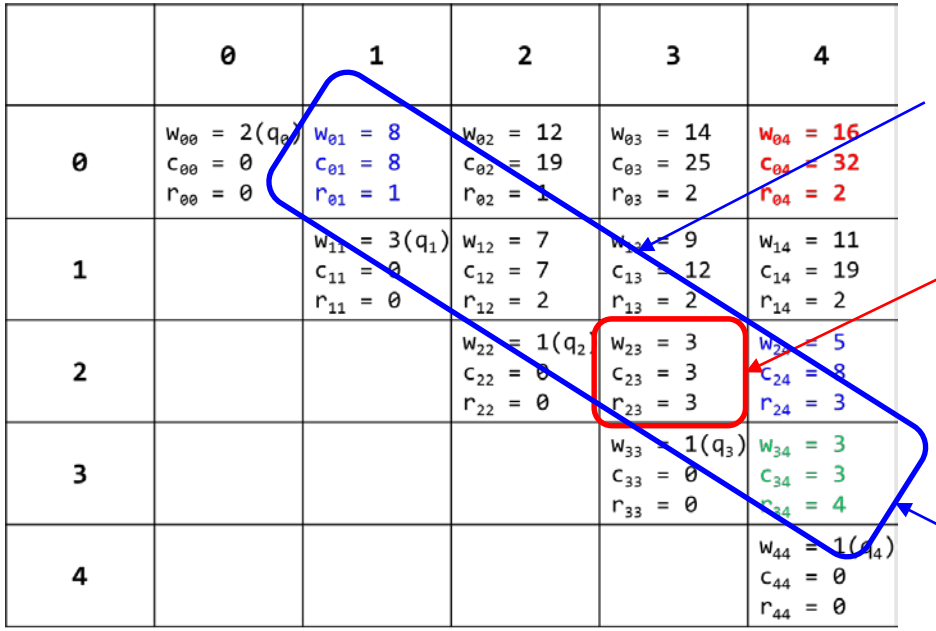
$$c[1,4] = w_{14} + \min\{c[1,1] + c[2,4],\ c[1,2] + c[3,4],\ c[1,3] + c[4,4]\}$$
$$= w_{11} + w_{24} + p_2 + \min\{0+8,\ 7+3,\ 12+0\}$$
$$= 3 + 5 + 3 + 8 = 19$$

$$c[0,4] = w_{04} + \min\{c[0,0] + c[1,4],\ c[0,1] + c[2,4],\ c[0,2] + c[3,4],\ c[0,3] + c[4,4]\}$$
$$= w_{00} + w_{14} + p_1 + \min\{0+19,\ 8+8,\ 19+3,\ 25+0\}$$
$$= 2 + 11 + 3 + 16 = 32$$

최적 이원 탐색 트리 찾기
(Determination of optimal binary search tree)

	0	1	2	3	4
0	$w_{00} = 2(q_0)$ $c_{00} = 0$ $r_{00} = 0$	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$
1		$w_{11} = 3(q_1)$ $c_{11} = 0$ $r_{11} = 0$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$
2			$w_{22} = 1(q_2)$ $c_{22} = 0$ $r_{22} = 0$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$
3				$w_{33} = 1(q_3)$ $c_{33} = 0$ $r_{33} = 0$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$
4					$w_{44} = 1(q_4)$ $c_{44} = 0$ $r_{44} = 0$

최적 이원 탐색 트리 찾기
(Determination of optimal binary search tree)



	0	1	2	3	4
0	$w_{00} = 2(q_0)$ $c_{00} = 0$ $r_{00} = 0$	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$
1		$w_{11} = 3(q_1)$ $c_{11} = 0$ $r_{11} = 0$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$
2			$w_{22} = 1(q_2)$ $c_{22} = 0$ $r_{22} = 0$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$
3				$w_{33} = 1(q_3)$ $c_{33} = 0$ $r_{33} = 0$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$
4					$w_{44} = 1(q_4)$ $c_{44} = 0$ $r_{44} = 0$

c_{ij}의 수 = n − m + 1, where m = j-i

c_{ij}의 min을 찾기 = m

```
c[0,3] = w03 + min{c[0,0] + c[1,3],
                   c[0,1] + c[2,3],
                   c[0,2] + c[3,3]}
```

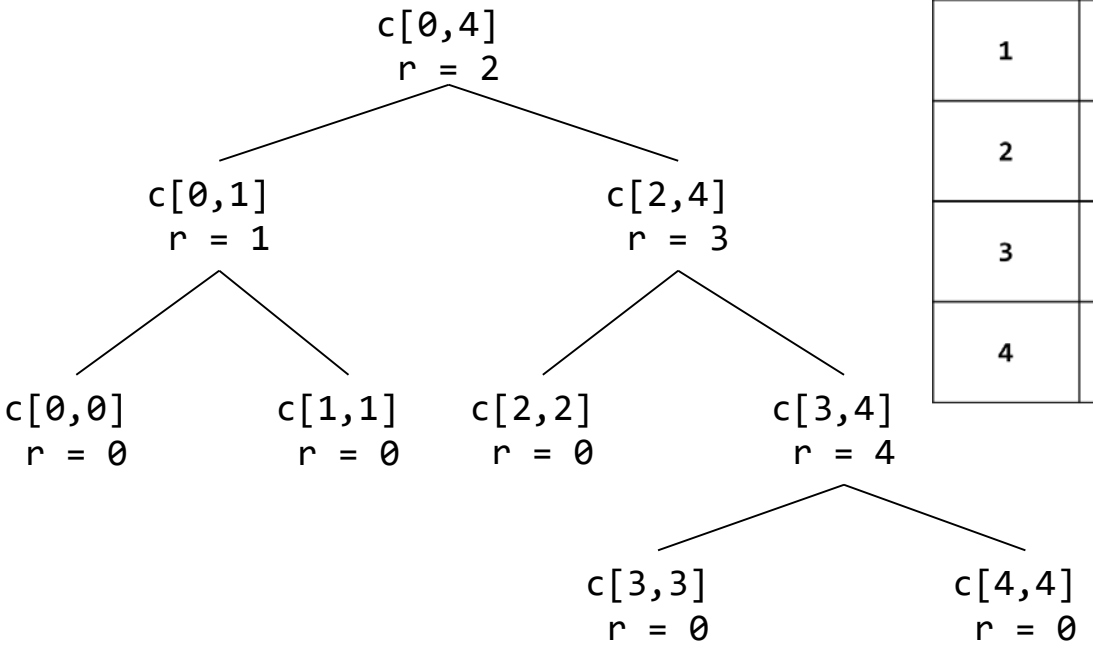
$$C_{ij} = w_{ij} + C_{i,k-1} + C_{kj}$$
$$= \min_{i<l\le j}\{w_{ij} + c_{i,l-1} + c_{lj}\}$$
$$= w_{ij} + \min_{i<l\le j}\{c_{i,l-1} + c_{lj}\}$$

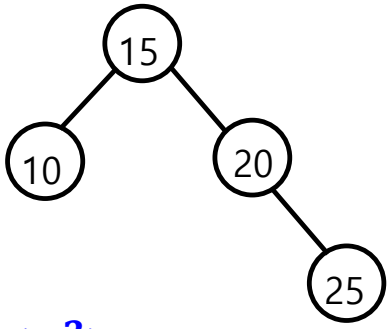
$O(m(n-m+1))$

$O(mn - m^2)$

최적 이원 탐색 트리 찾기
(Determination of optimal binary search tree)

	0	1	2	3	4
0	$w_{00} = 2(q_0)$ $c_{00} = 0$ $r_{00} = 0$	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$
1		$w_{11} = 3(q_1)$ $c_{11} = 0$ $r_{11} = 0$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$
2			$w_{22} = 1(q_2)$ $c_{22} = 0$ $r_{22} = 0$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$
3				$w_{33} = 1(q_3)$ $c_{33} = 0$ $r_{33} = 0$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$
4					$w_{44} = 1(q_4)$ $c_{44} = 0$ $r_{44} = 0$





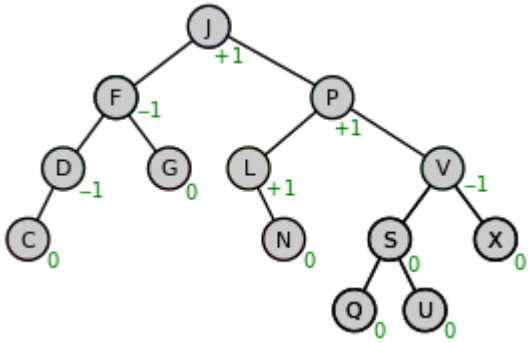
$$O(n(4^n/n^{3/2}))$$

$$\sum_{m=1}^{n}(nm - m^2) = O(n^3)$$

Georgy Maximovich Adelson-Velsky (8 January 1922 – 26 April 2014) was a Soviet and Israeli mathematician and computer scientist.

Evgenii Mikhailovich Landis (October 6, 1921 – December 12, 1997) was a Soviet mathematician who worked mainly on partial differential equations.

AVL 트리
(AVL Trees)

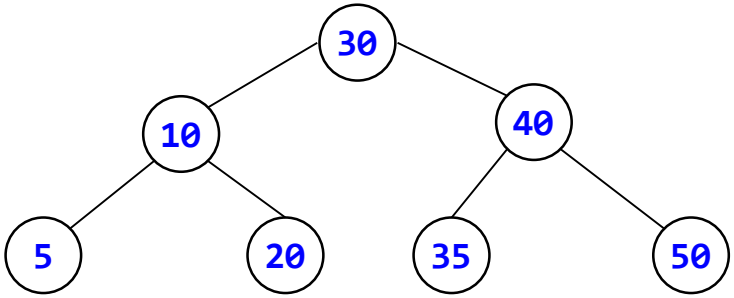


In computer science, an AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing binary search tree. It was the first such data structure to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

-- wikipedia

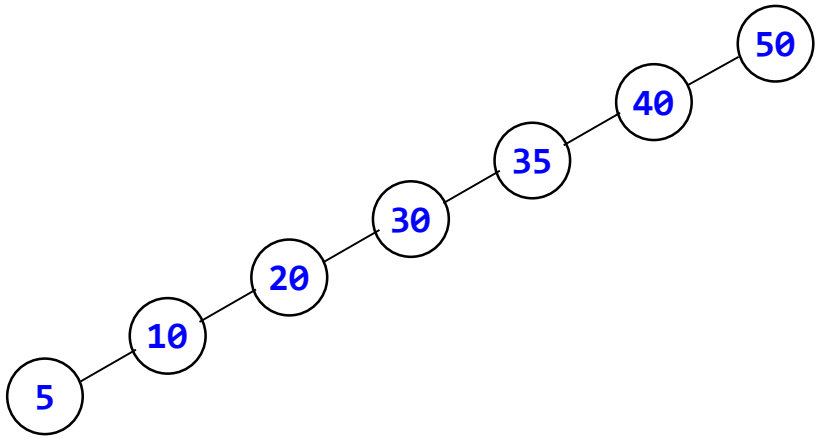
개념(Overview)

- **Binary Search Tree의 문제점(drawbacks)**
  - Key: 30, 40, 10, 50, 20, 5, 35



O(log n)

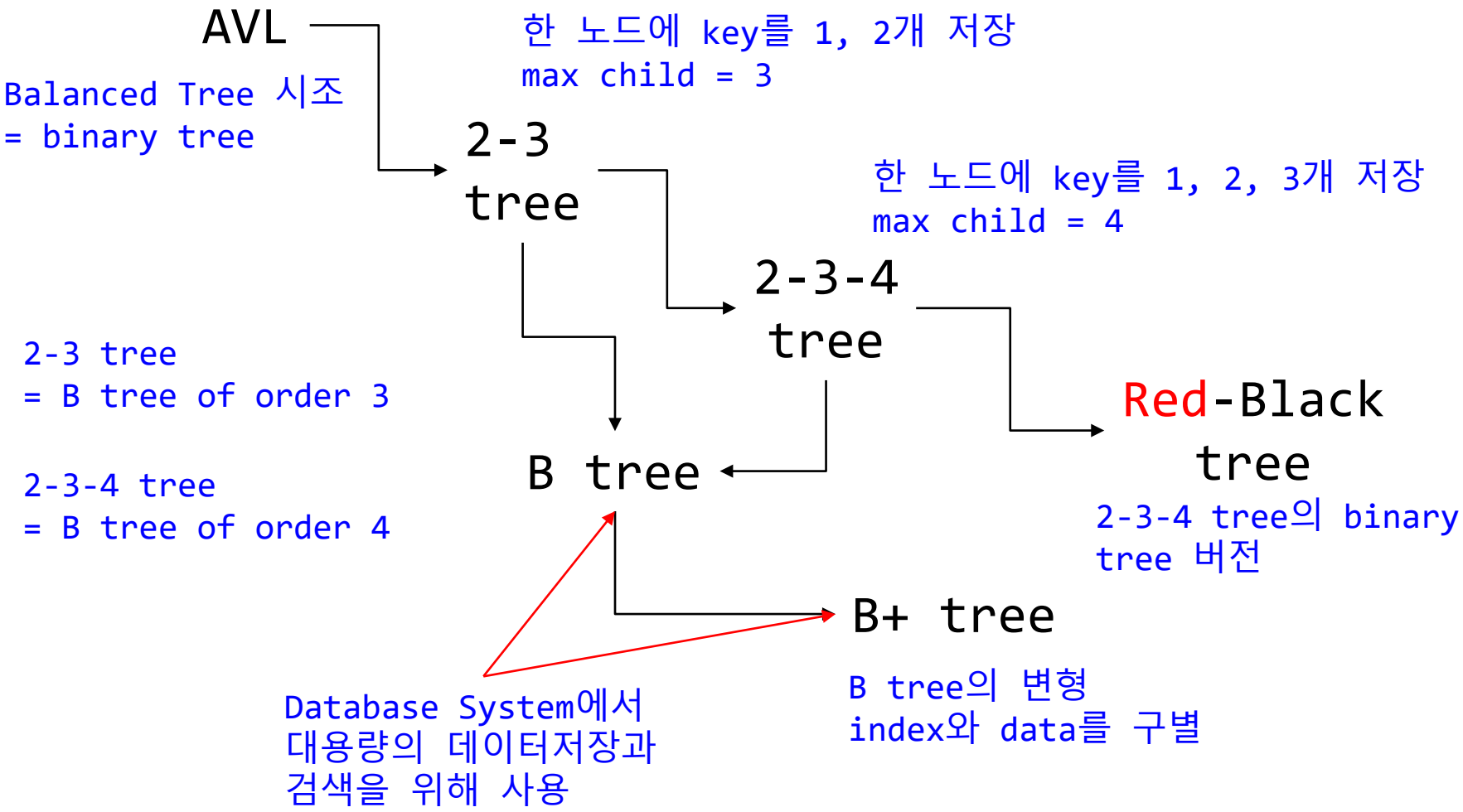
  - Key: 50, 40, 35, 30, 20, 10, 5 (same key different order)



O(n)

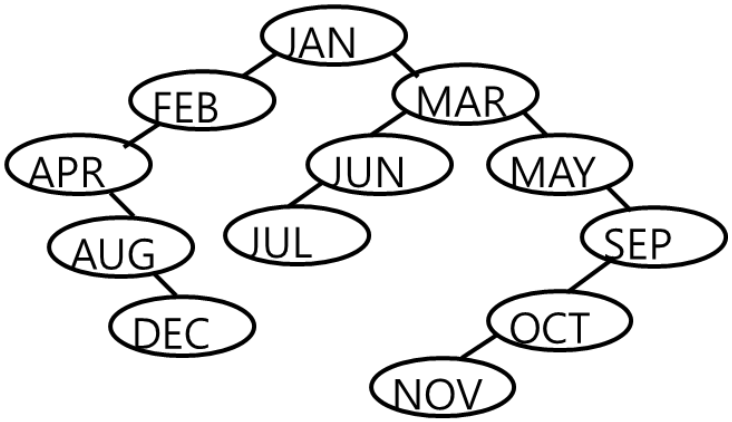
개념(Overview)

- **Balanced Binary Search Tree**

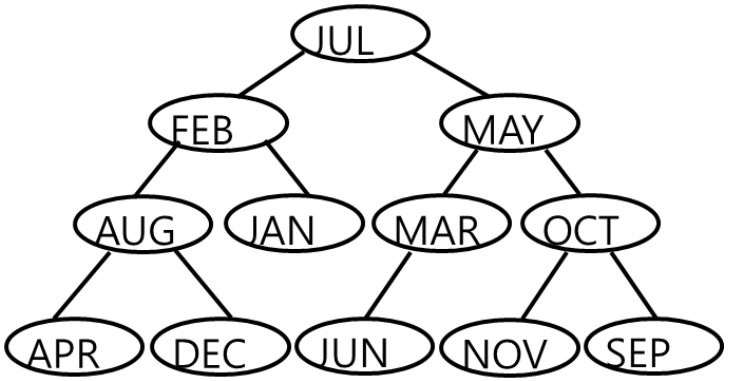


AVL Tree

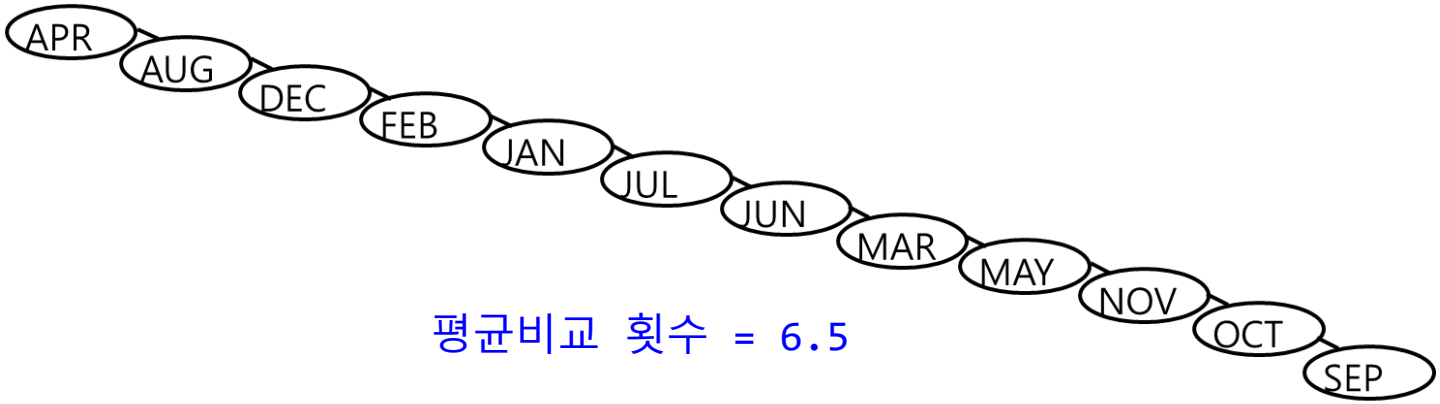
- 어떤 원소를 찾기 위한 최대 비교 횟수



평균비교 횟수 = 42/12 = 3.5



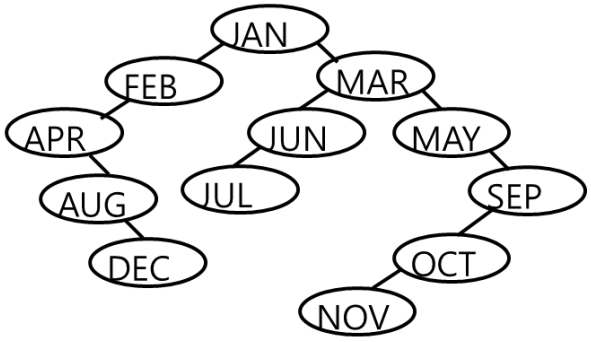
평균비교 횟수 = 37/12 ~ 3.1



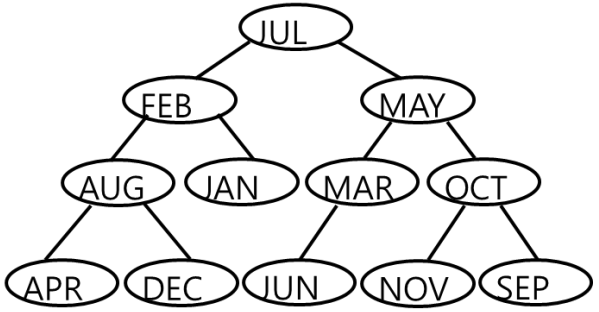
평균비교 횟수 = 6.5

AVL Tree 정의

- 공백 트리(empty tree)는 높이 균형을 이룬다
- T_L과 T_R을 가진 공백이 아닌 이진 트리라 할 때
  - T_L : 왼쪽 서브트리, T_R : 오른쪽 서브트리
  - $|h_L - h_R| \leq 1$
    - h_L: T_L의 높이, h_R: T_R의 높이
  - T_L과 T_R이 높이 균형을 이룬다



균형 잡히지 않은 트리



균형 잡힌 트리

AVL Tree 정의

- **균형인수(balance factor)**
  - BF(T) = h_L – h_R : 왼쪽, 오른쪽 서브트리 사이의 높이 차
  - AVL 트리의 어떠한 노드 T에 대해서도 BF(T)= -1, 0, 1

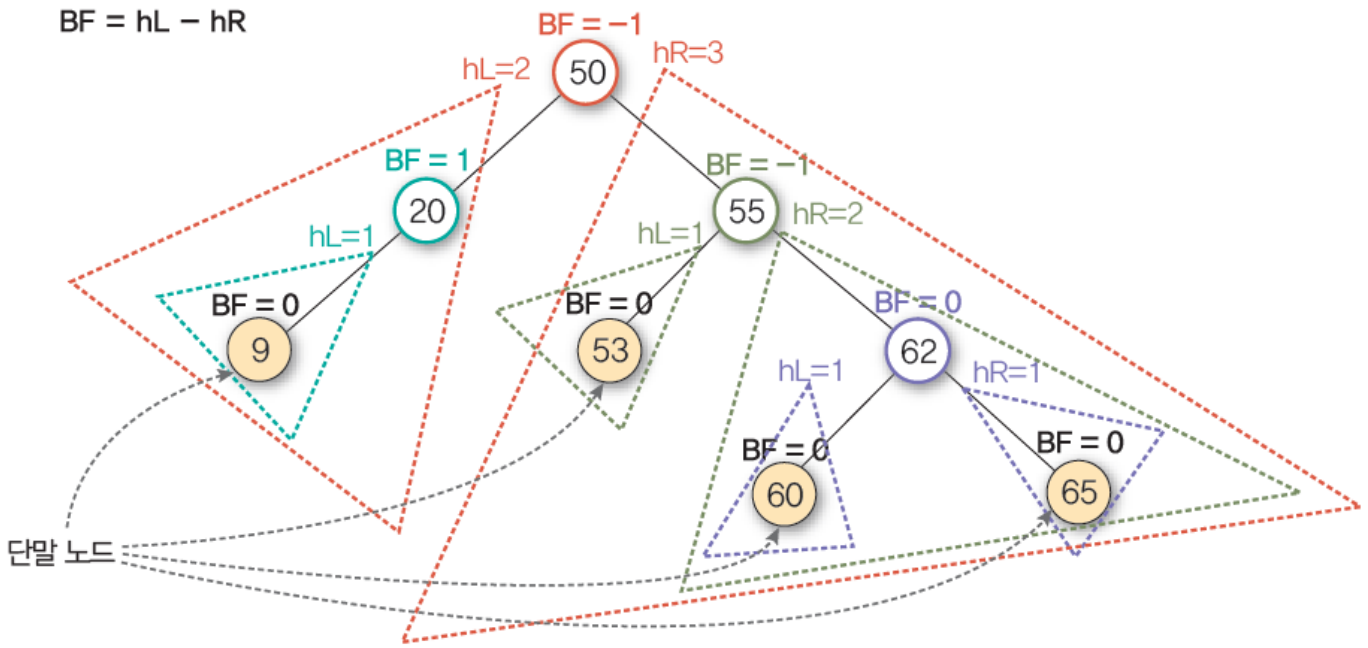
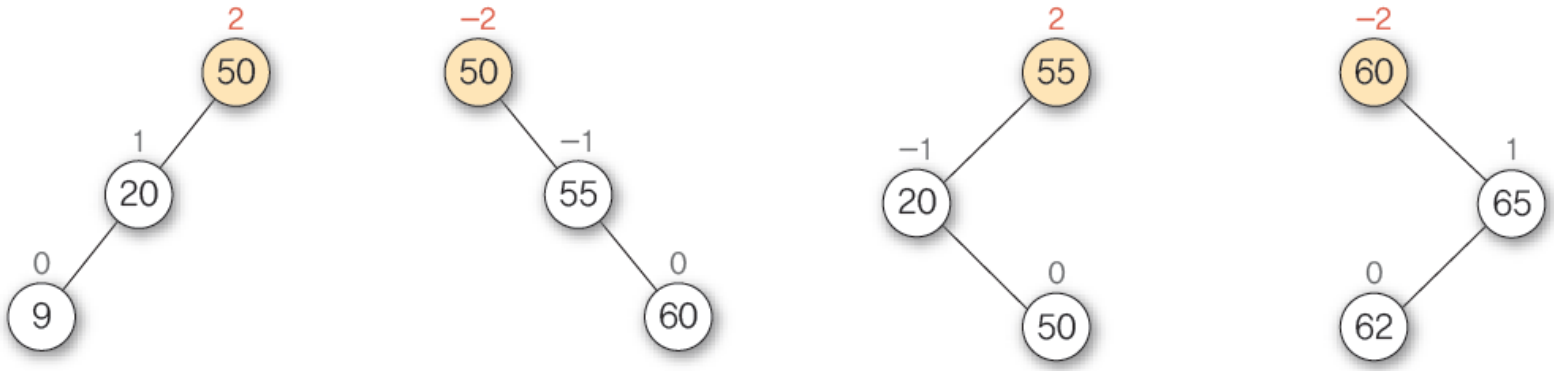


그림 7–45 AVL 트리의 균형 인수 구하기 예

AVL Tree 정의

- **균형인수가 ±2가 된 원인 (균형이 깨지는 원인)**
  - **LL**: 새 노드 Y는 A의 왼쪽 서브트리의 왼쪽 서브트리에 삽입
  - **RR**: Y는 A의 오른쪽 서브트리의 오른쪽 서브트리에 삽입
  - **LR**: Y는 A의 왼쪽 서브트리의 오른쪽 서브트리에 삽입
  - **RL**: Y는 A의 오른쪽 서브트리의 왼쪽 서비트리에 삽입

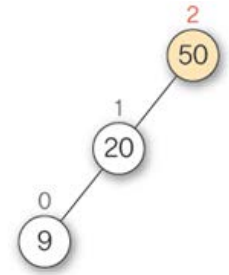


(a) LL유형 (b) RR 유형 (c) LR 유형 (d) RL 유형

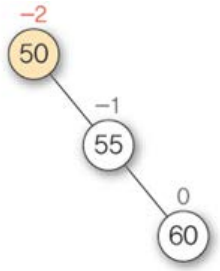
그림 7-47 비AVL 트리의 예

Rotations

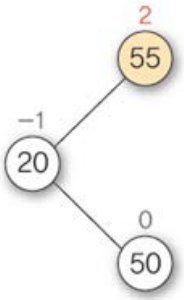
- **LL Rotation (Single Rotation)**
  - LL Line 상의 중앙 노드를 부모로 만듦
- **RR Rotation (Single Rotation)**
  - RR Line 상의 중앙 노드를 부모로 만듦
- **LR Rotation (Double Rotation)**
  - LL 유형을 만든 후 LL Rotation 적용
- **RL Rotation (Double Rotation)**
  - RR 유형을 만든 후 RR Rotation 적용



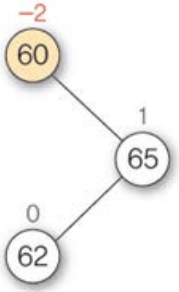
(a) LL유형



(b) RR 유형



(c) LR 유형



(d) RL 유형

Rotations

- **LL Rotation (Single Rotation)**
  - LL Line 상의 중앙 노드를 부모로 만듦

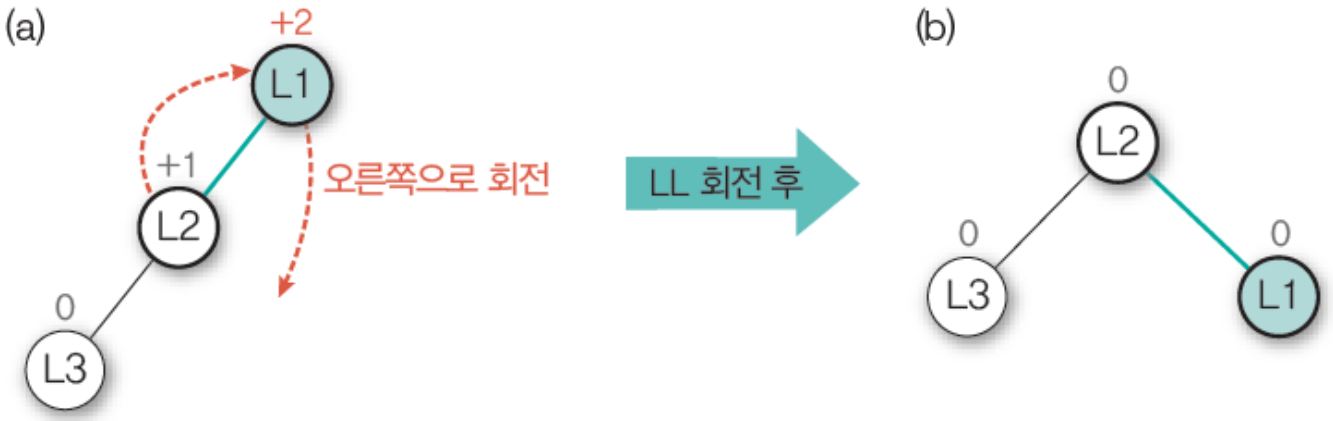


그림 7-48 LL 유형과 회전 예

Rotations

- **RR Rotation (Single Rotation)**
  - RR Line 상의 중앙 노드를 부모로 만듦

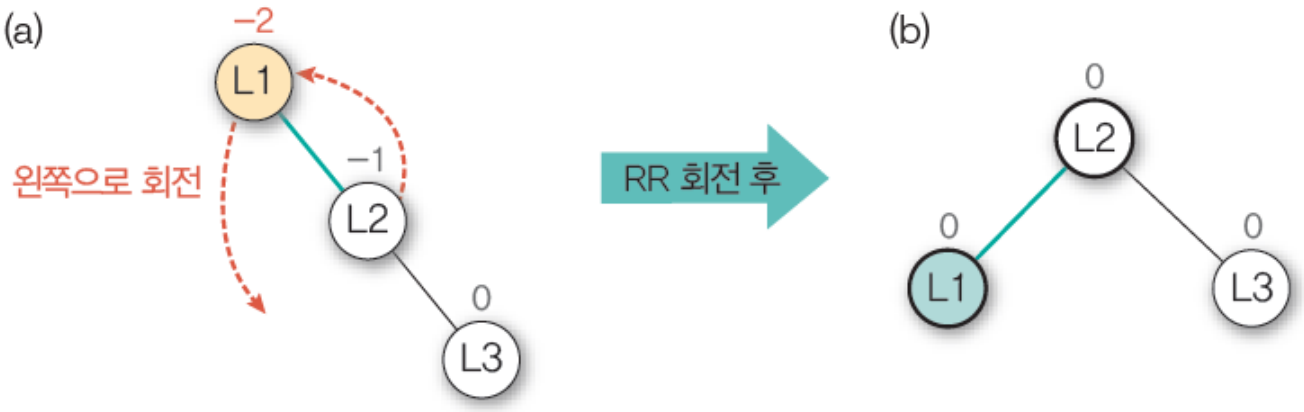


그림 7-49 RR 유형과 회전 예

Rotations

- **LR Rotation (Double Rotation)**
  - LL 유형을 만든 후 LL Rotation 적용

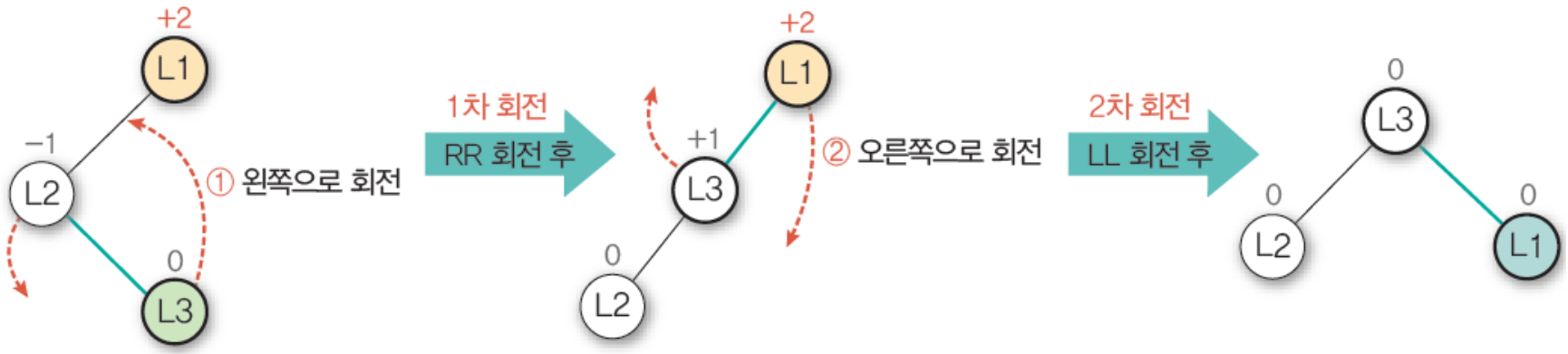


그림 7-50 LR 유형과 LR 회전 예

Rotations

- **RL Rotation (Double Rotation)**
  - RR 유형을 만든 후 RR Rotation 적용

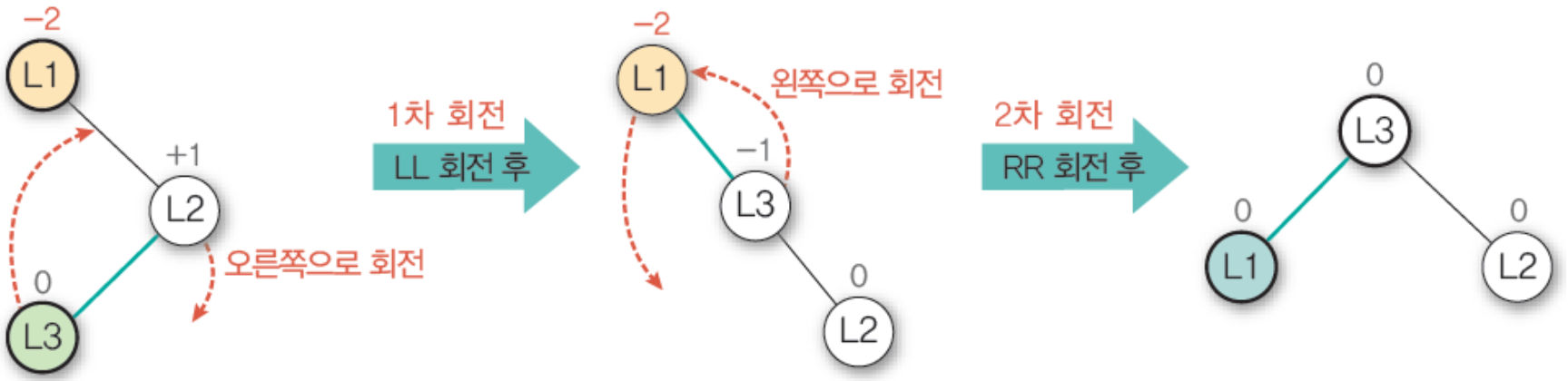
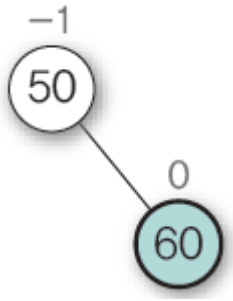
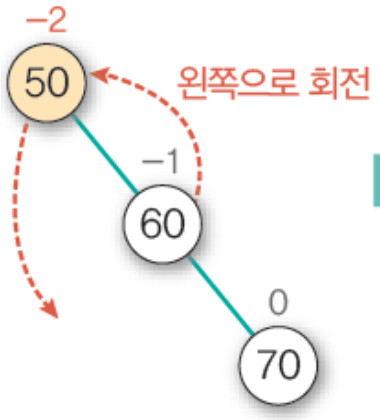


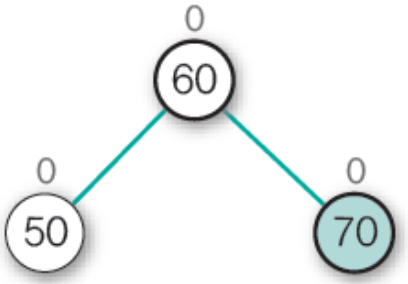
그림 7-51 RL 유형과 회전 예

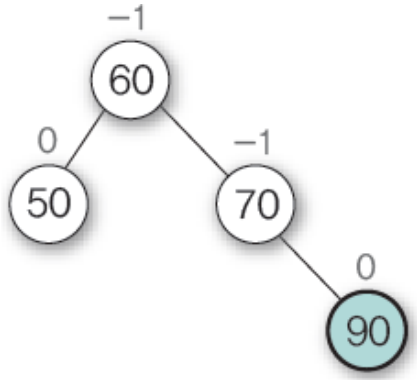
Example

insertion node

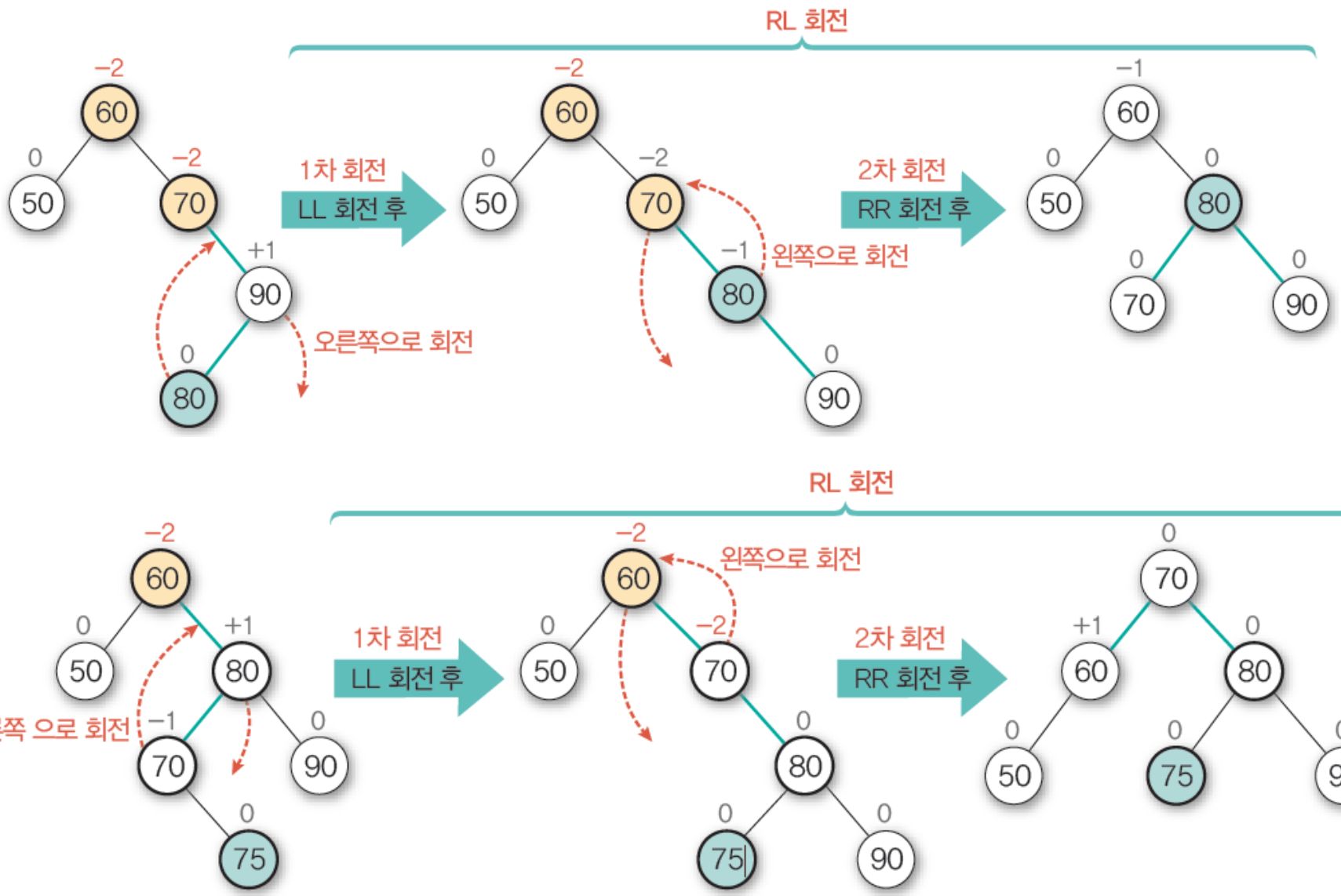




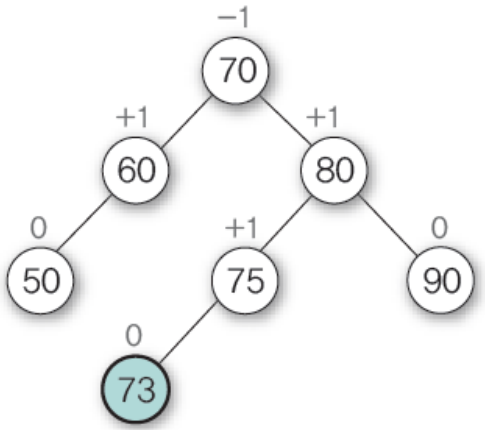


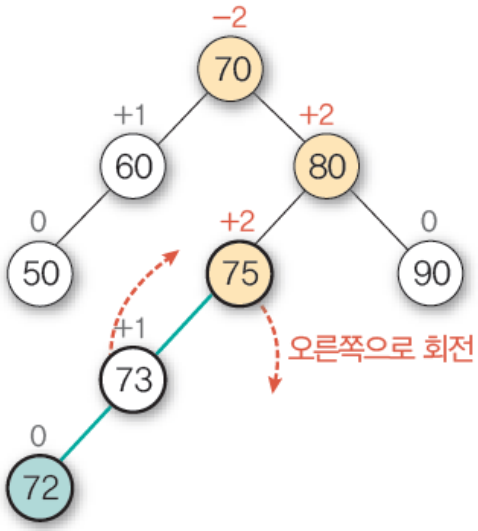


Example

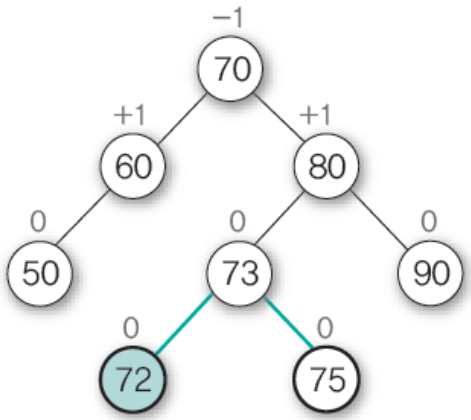

RL 회전
1차 회전
LL 회전 후
2차 회전
RR 회전 후
왼쪽으로 회전
오른쪽으로 회전
RL 회전
1차 회전
LL 회전 후
2차 회전
RR 회전 후
왼쪽으로 회전
오른쪽 으로 회전

Example

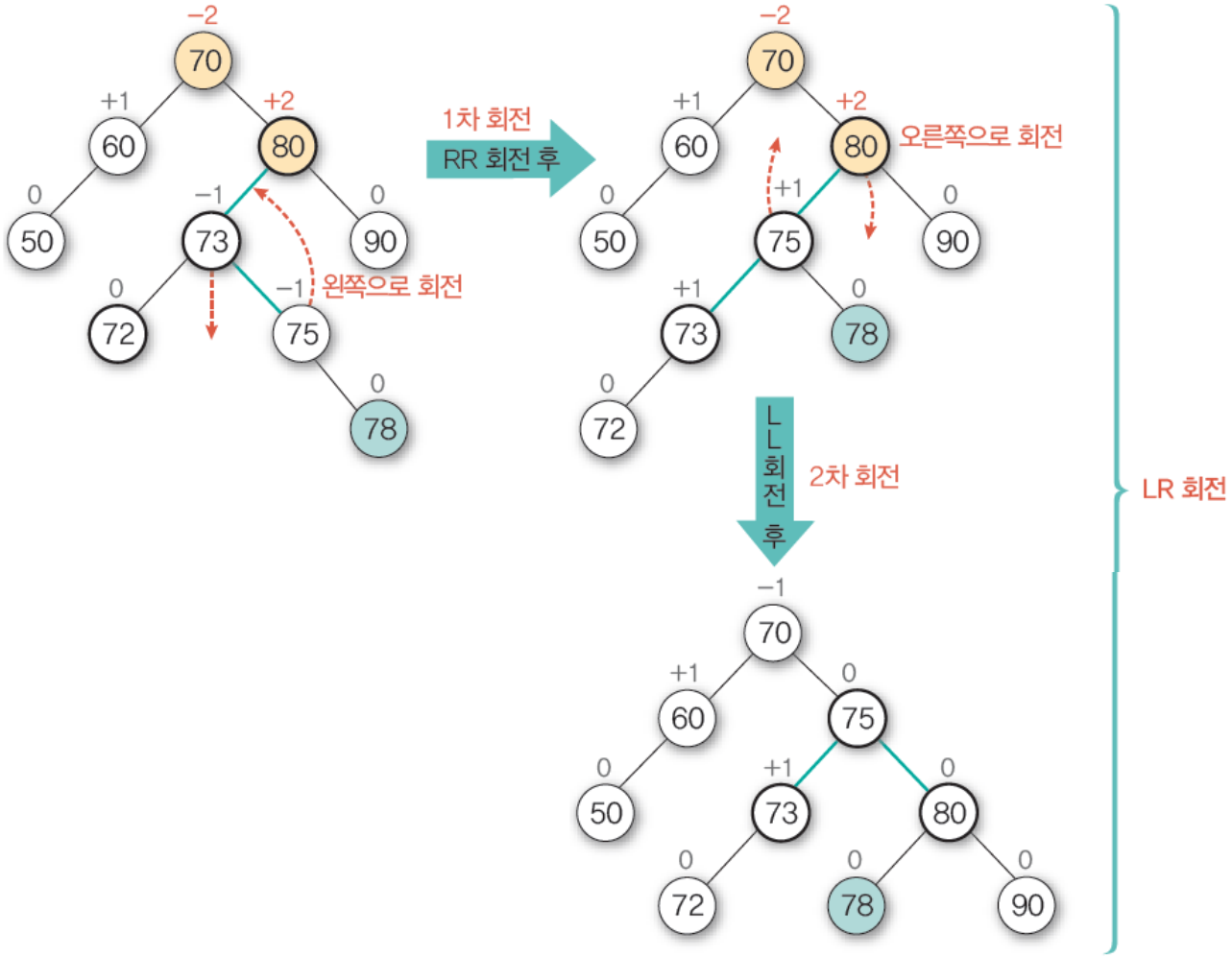

−1
70
+1
60
+1
80
0
50
+1
75
0
90
0
73


−2
70
+1
60
+2
80
0
50
+2
75
0
90
+1
73
오른쪽으로 회전
0
72

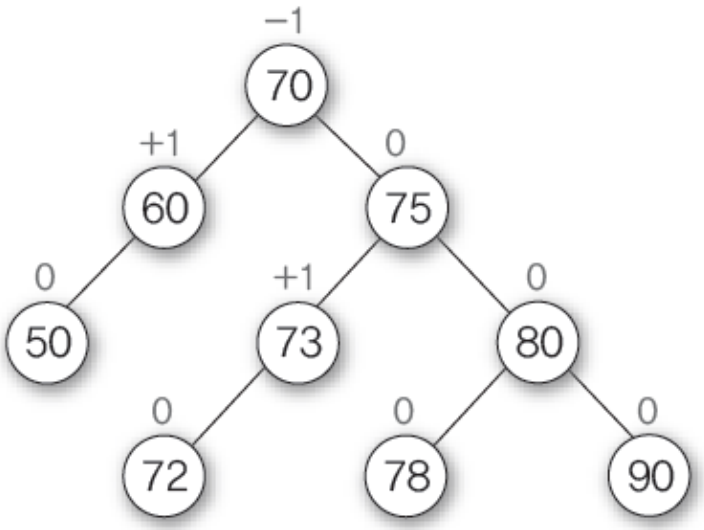

LL 회전 후


−1
70
+1
60
+1
80
0
50
0
73
0
90
0
72
0
75

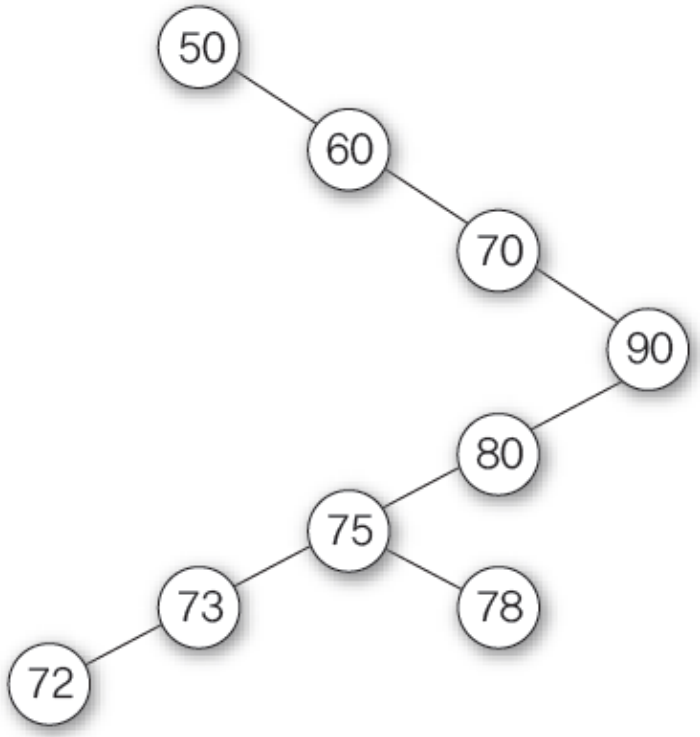
Example


−2
70
+1
60
+2
80
0
50
−1
73
0
90
0
72
−1
75
0
78
왼쪽으로 회전
1차 회전
RR 회전 후
−2
70
+1
60
+2
80
오른쪽으로 회전
0
50
+1
75
0
90
+1
73
0
78
0
72
LL 회전 후
2차 회전
LR 회전
−1
70
+1
60
0
75
0
50
+1
73
0
80
0
72
0
78
0
90

AVL vs. Binary Search Tree



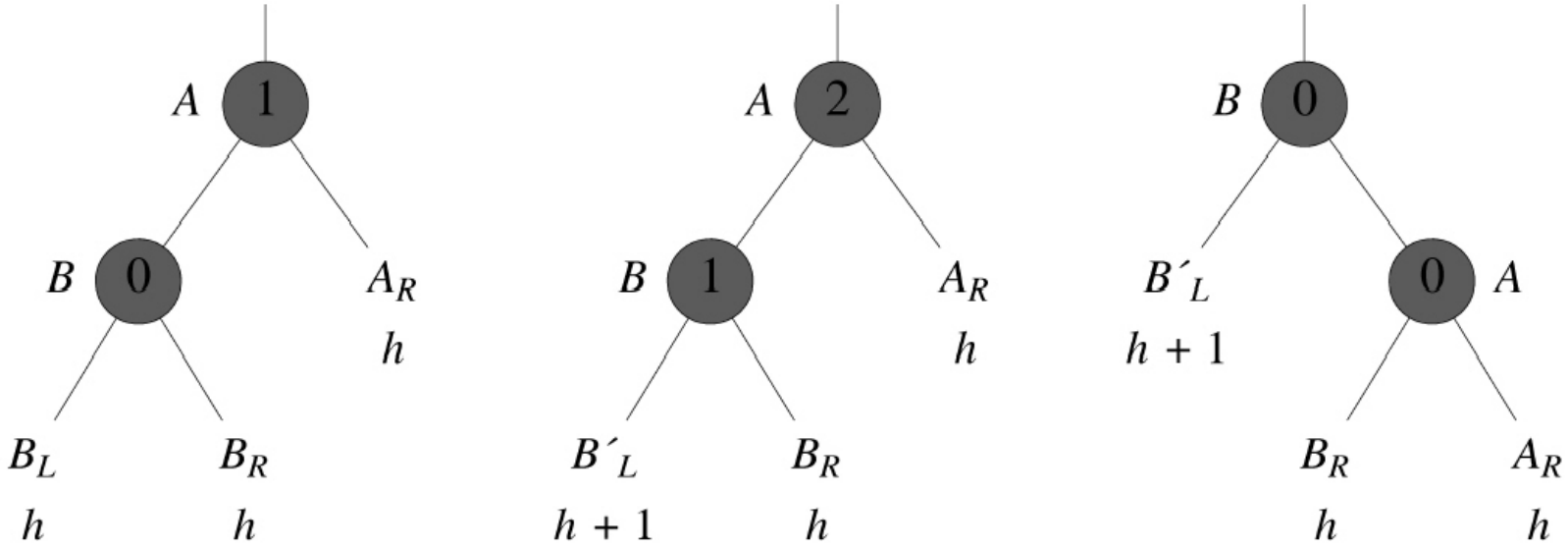
(a) AVL 트리



(b) 이진 탐색 트리

그림 7−52 같은 원소를 같은 순서로 삽입한 경우의 AVL 트리와 이진 탐색 트리 비교

(교재)LL 회전(LL Rotation)

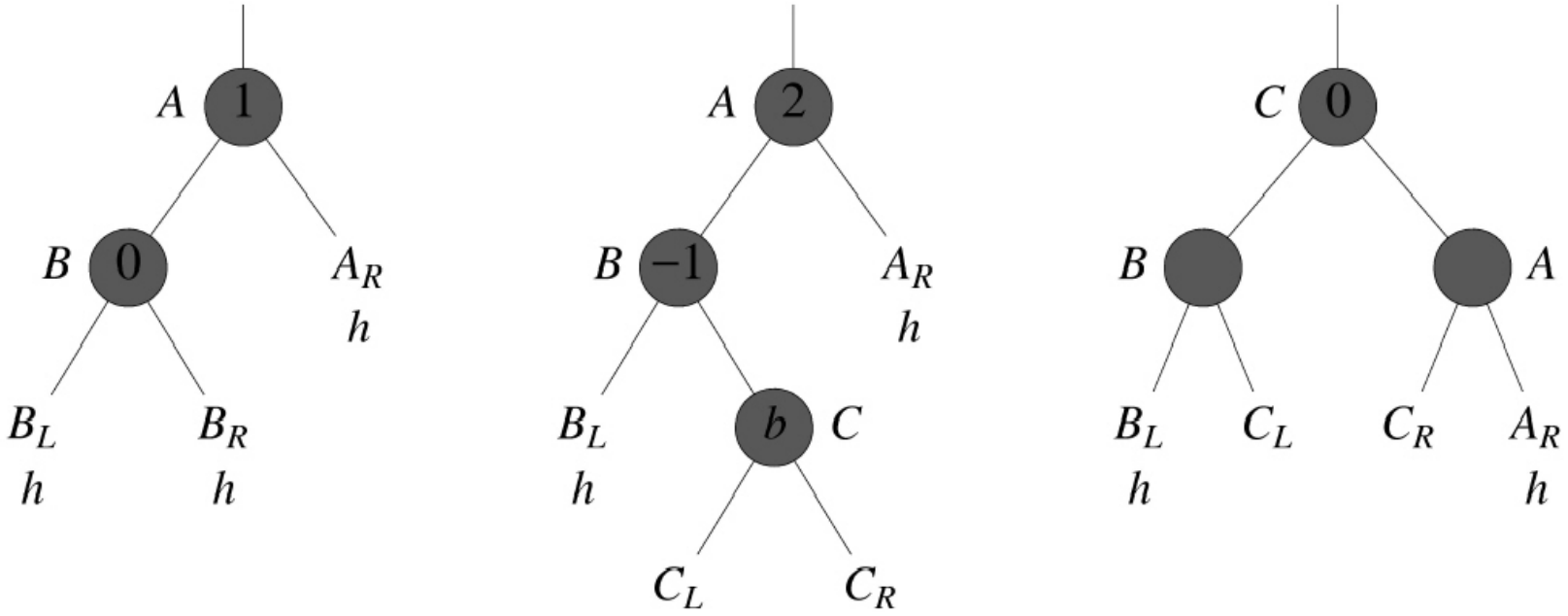


삽입 전 B_L에 삽인한 뒤 LL 회전 뒤

균형 인수(balance factor)는 노드 안에 있음

서브트리 높이는 서브트리 이름 밑에 있음

(교재)LR 회전(LR Rotation)



삽입 전 B_R에 삽입한 뒤 **LR 회전 뒤**

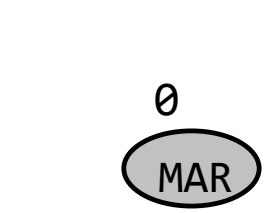
b = 0 ⇒ bf(B) = bf(A) = 0 (after rotation)

b = 1 ⇒ bf(B) = 0 and bf(A) = -1 (after rotation)

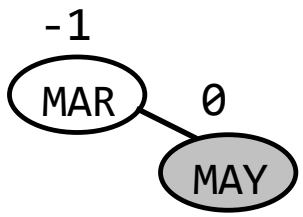
b = -1 ⇒ bf(B) = 1 and bf(A) = 0 (after rotation)

(교재)Example

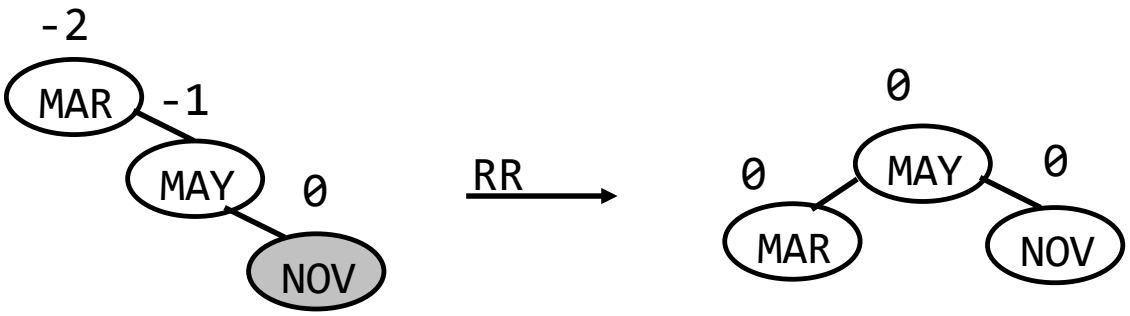
MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP



(a) 삽입 MARCH

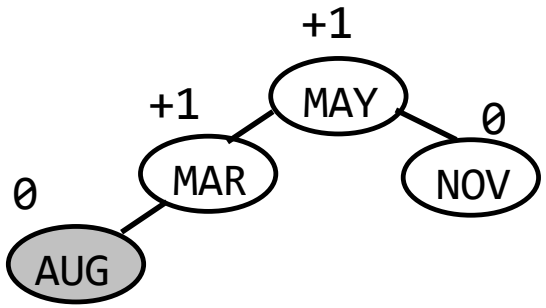


(b) 삽입 MAY

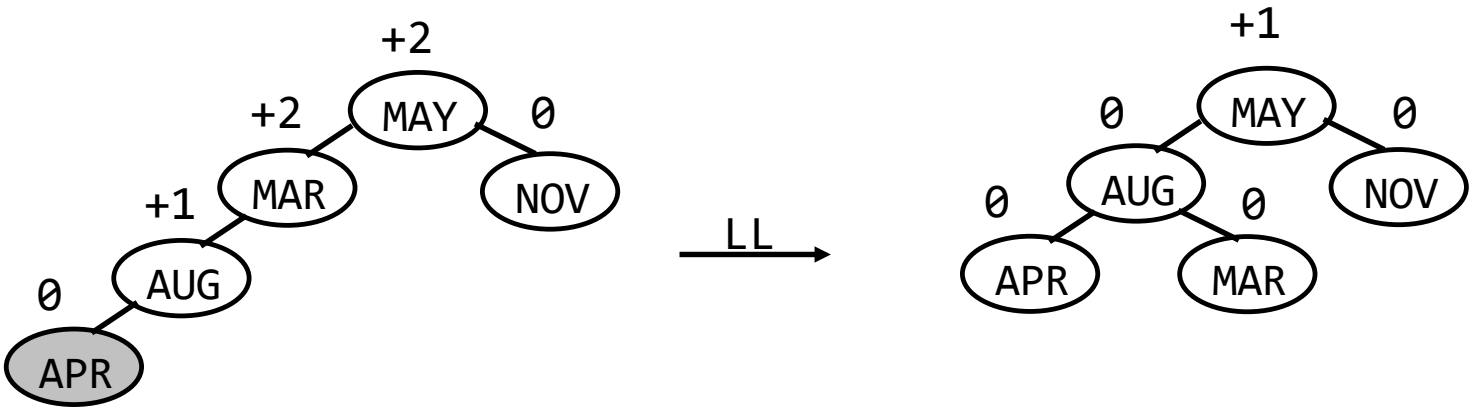


(c) 삽입 NOV

(교재)Example

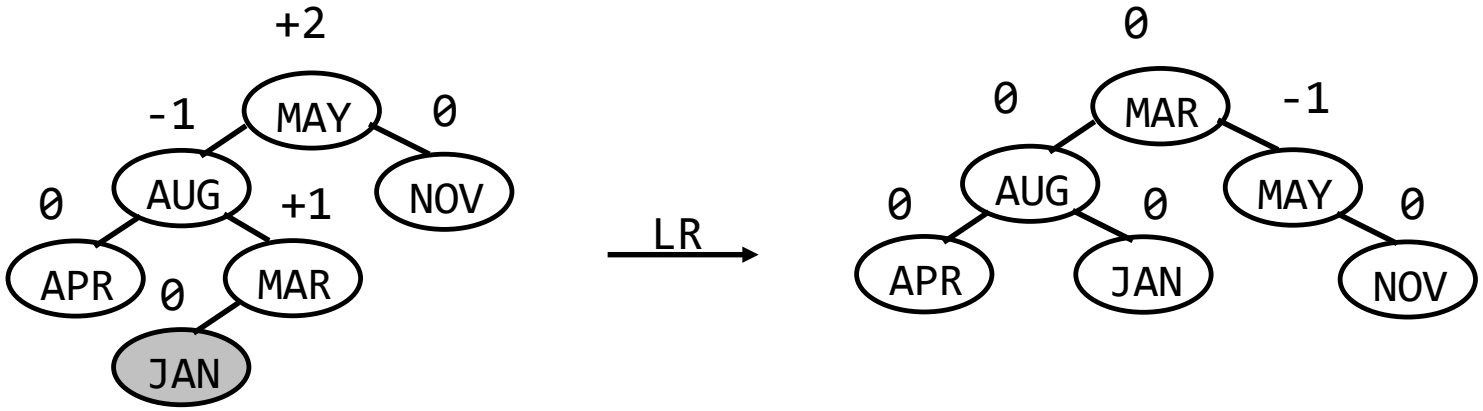


(d) 삽입 AUGUST

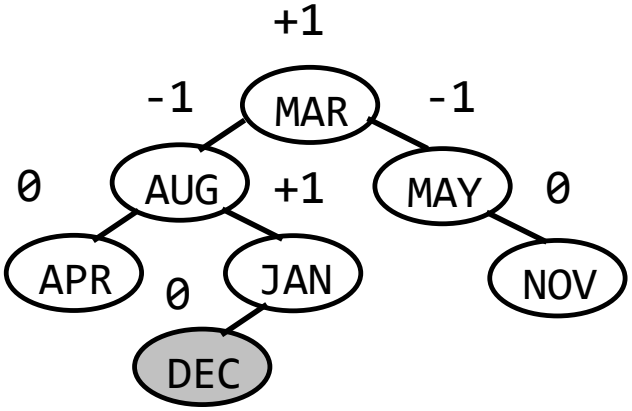


(e) 삽입 APRIL

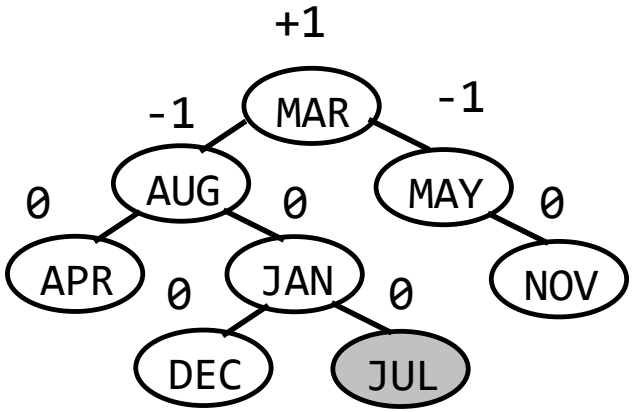
(교재)Example



(f) 삽입 JANUARY

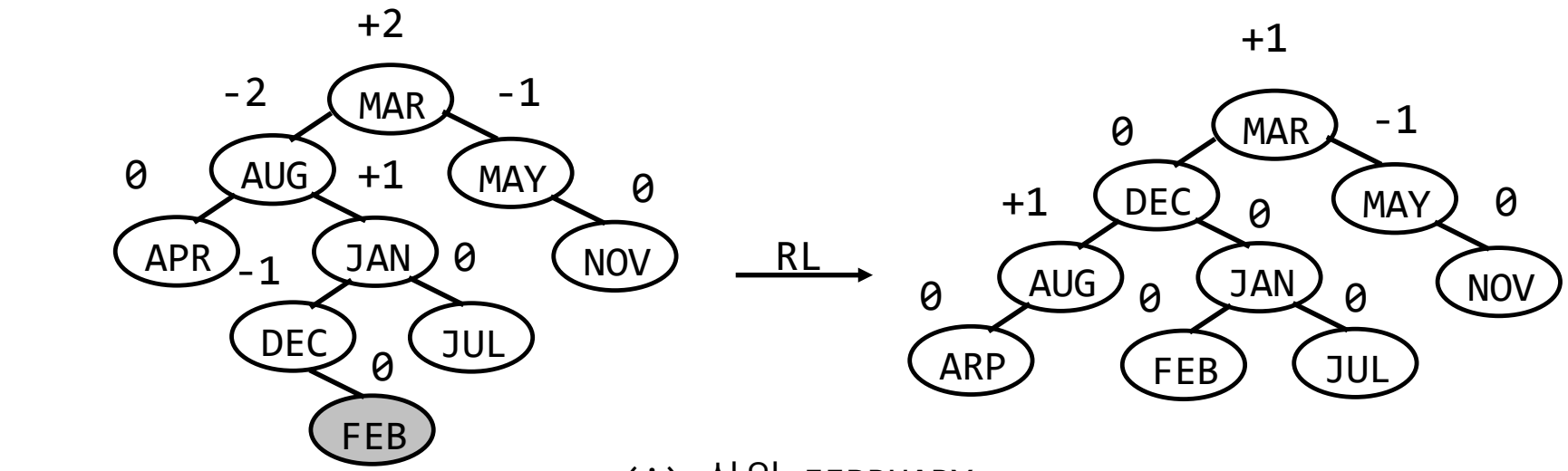


(g) 삽입 DECEMBER

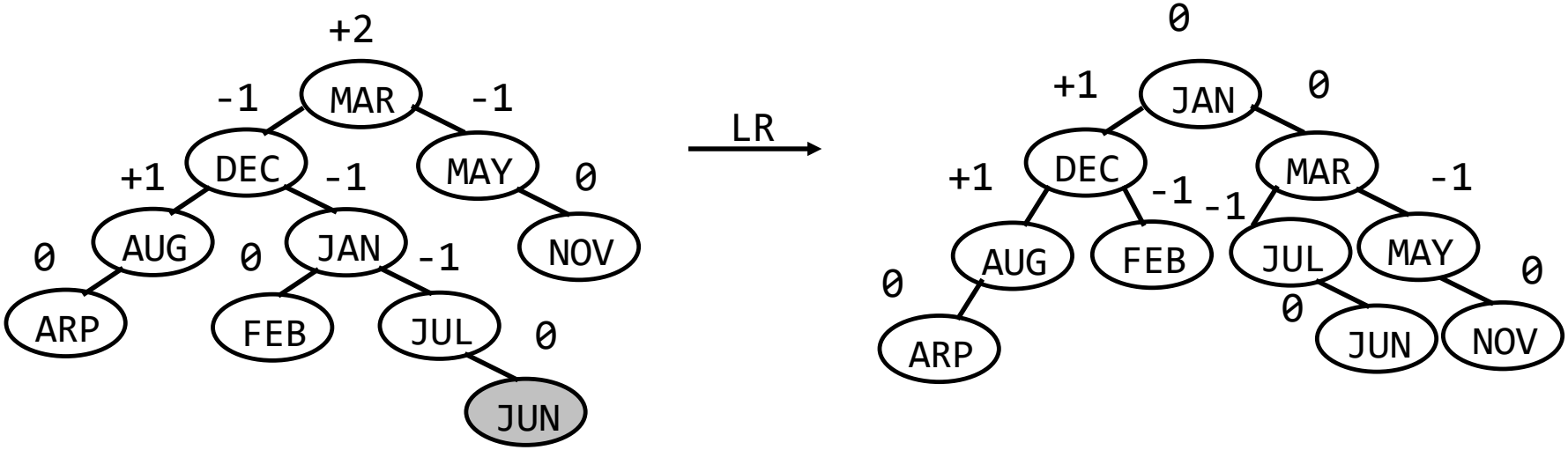


(h) 삽입 JULY

(교재)Example

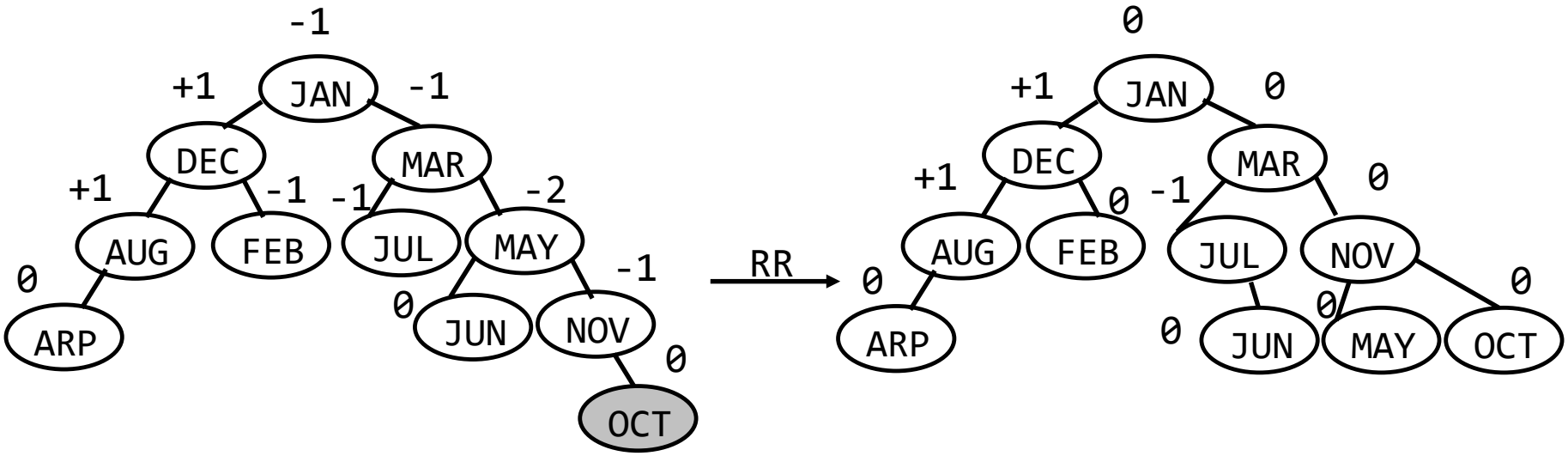


(i) 삽입 FEBRUARY

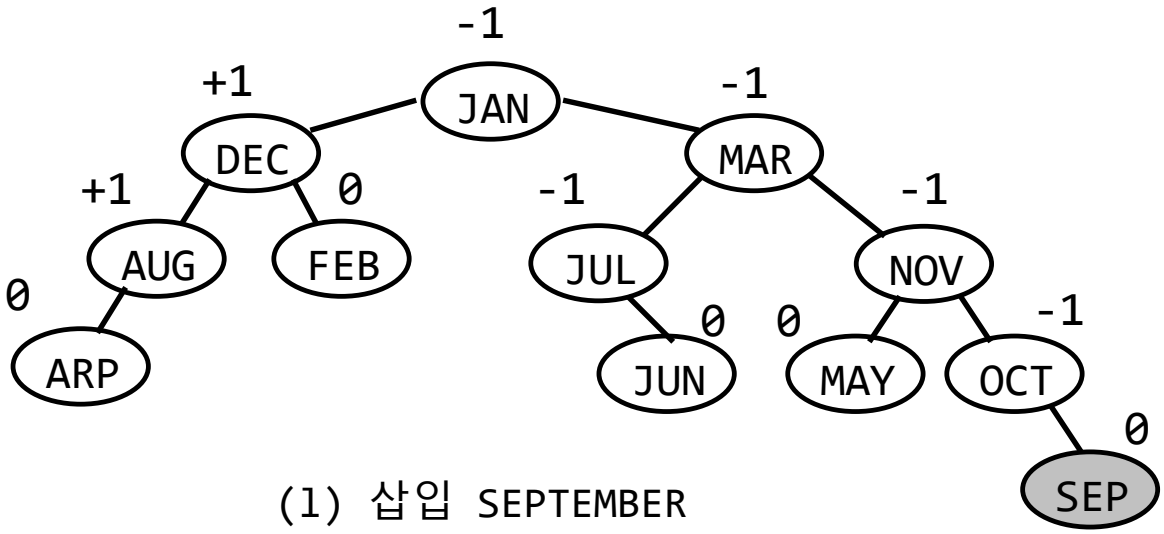


(j) 삽입 JUNE

(교재)Example



(k) 삽입 OCTOBER

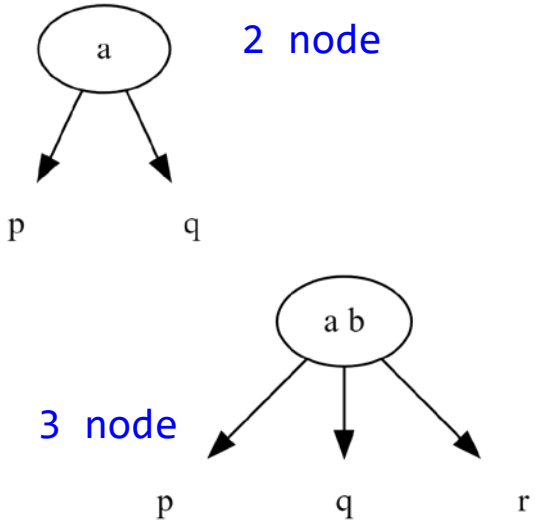


(l) 삽입 SEPTEMBER

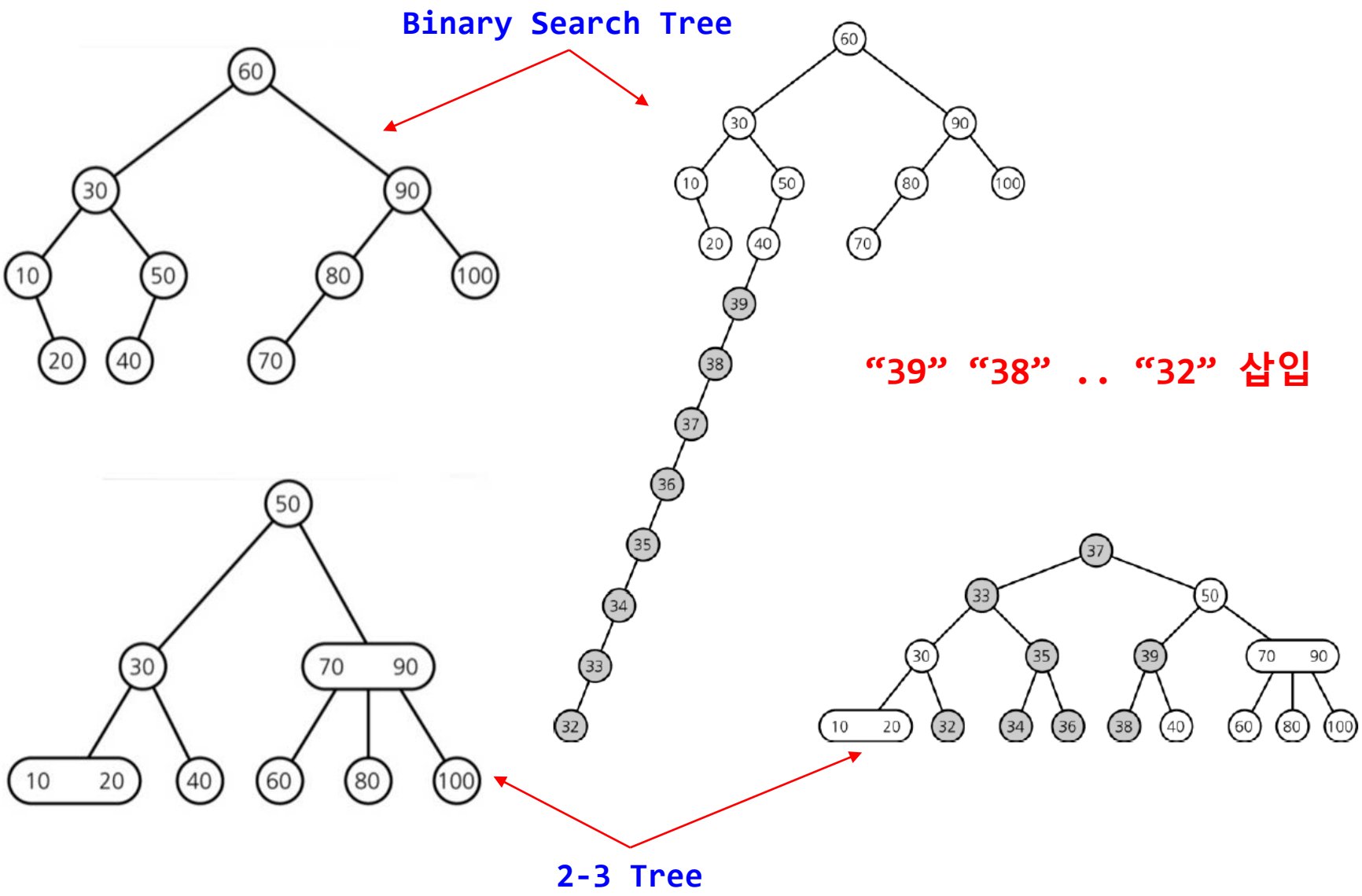
John Edward Hopcroft (born October 7, 1939) is an American theoretical computer scientist. His textbooks on theory of computation (also known as the Cinderella book) and data structures are regarded as standards in their fields. He is the IBM Professor of Engineering and Applied Mathematics in Computer Science at Cornell University.

2-3 트리
(2-3 Trees)

A 2–3 tree is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. According to Knuth, "a B-tree of order 3 is a 2-3 tree." 2−3 trees were invented by John Hopcroft in 1970. -- wikipedia

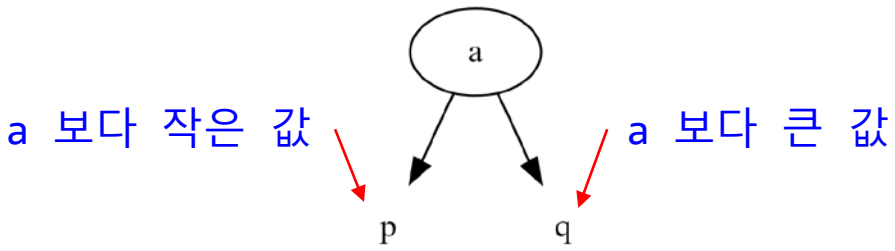


Binary Search Tree vs. 2-3 Tree

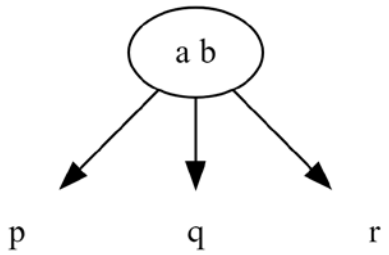


2-3 Tree Concept

- **Binary Tree의 확장**
  - 한 노드가 세 개까지의 자식을 가질 수 있는 트리



2-node



3-node

  - **2-node:**
    - Key 값은 하나, **자식은 2개**
  - **3-node:**
    - Key 값은 2개, **자식은 3개**

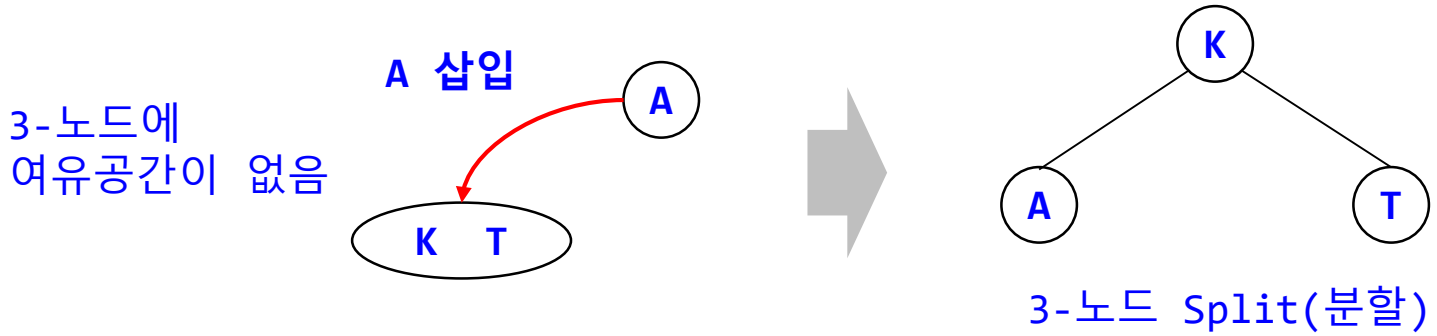
Important

2-node가 자식을 하나만 갖는 것은 허용이 안됨

3-node가 자식을 하나 또는 둘만 갖는 것은 허용이 안됨

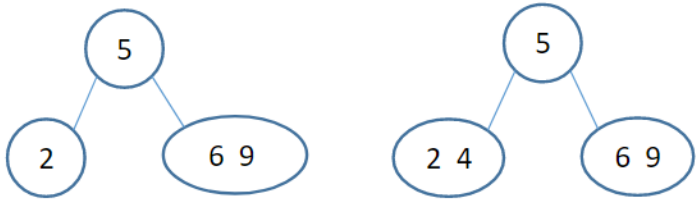
삽입(Insertion), 분할(Split)

- 데이터를 삽입할 때, 노드에 빈공간이 없을 경우 노드 Split

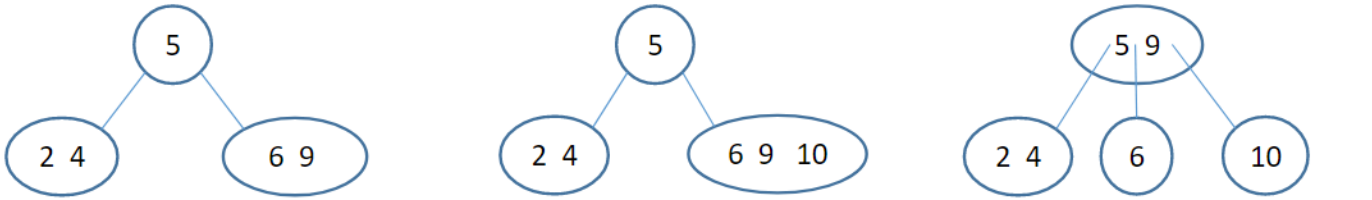


Split될 때 중앙 키가 부모로 이동

Insert in a 2-node :

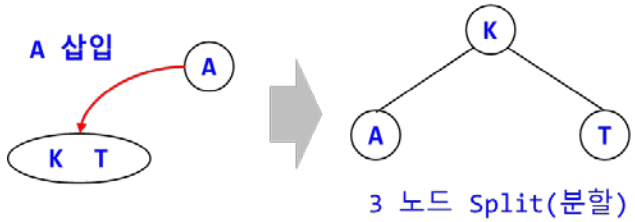


Insert in a 3-node (2 node parent) :

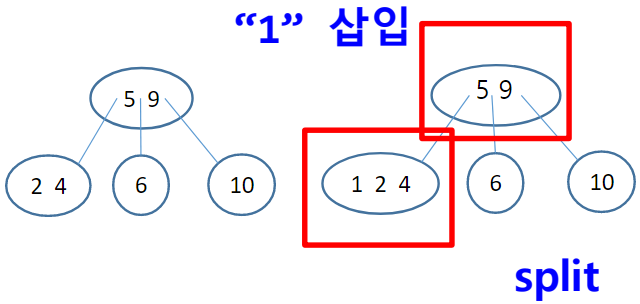


2-3 Tree 삽입 알고리즘

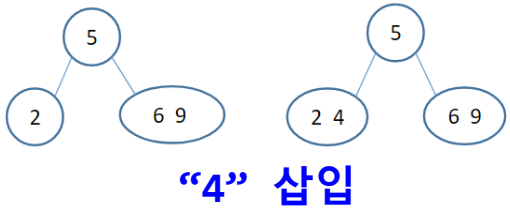
- **현재 노드가 외부 노드가 아닐 동안**
  - 삽입 키가 왼쪽 키보다 작으면 왼쪽 자식으로
  - 삽입 키가 오른쪽 키보다 작으면 중간 자식으로
  - 삽입 키가 오른쪽 키보다 크면 오른쪽 자식으로



- **현재 노드가 3-노드이면**
  - 현재 노드를 분할 한다.
  - 만약 그 부모 노드도 3-노드면 삽입경로를 거치면서 분할한다.

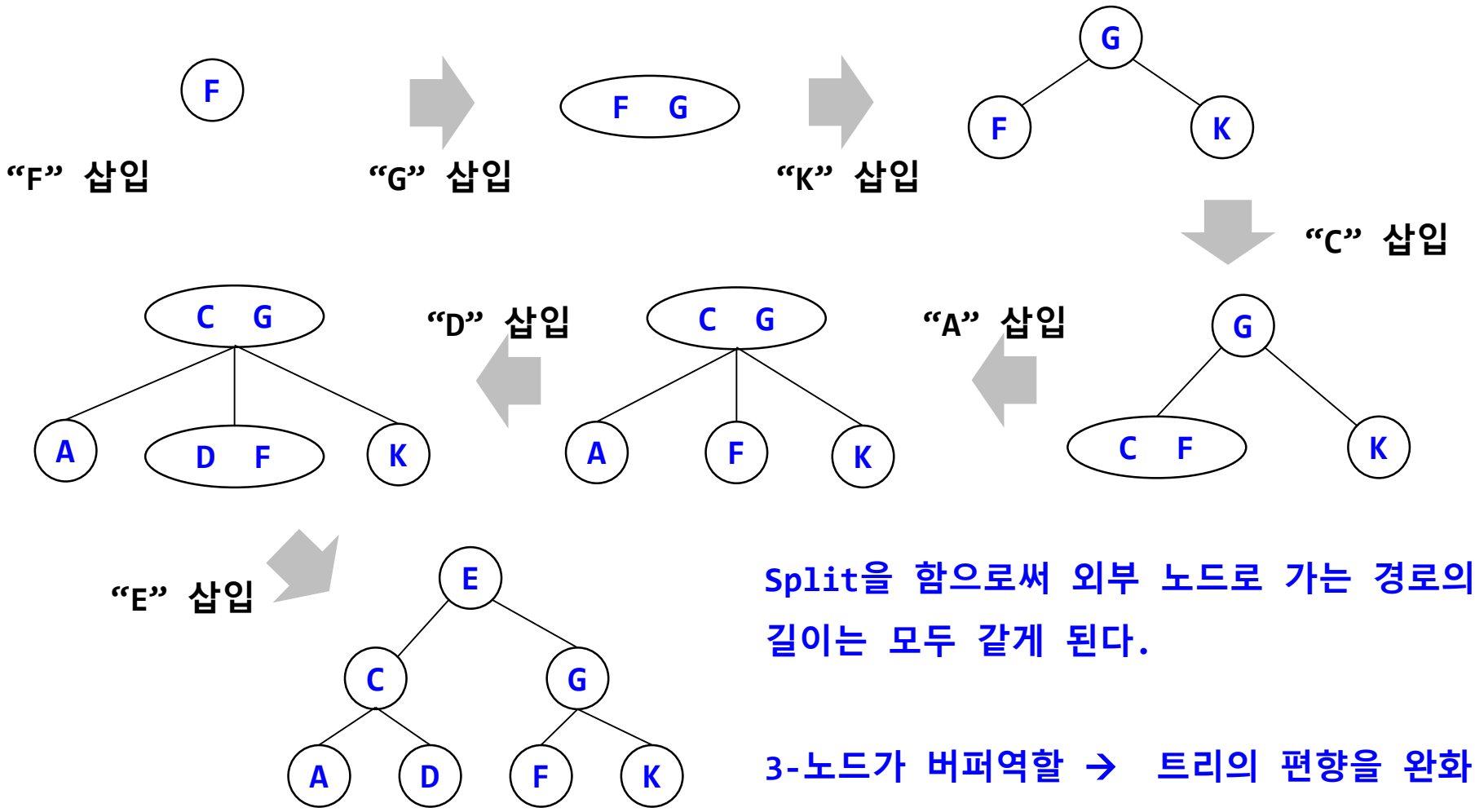


- **현재 노드가 2-노드이면 삽입키를 추가**



2-3 Tree Insertion Example

- F, G, K, C, A, D, E 순으로 2-3 Tree에 삽입

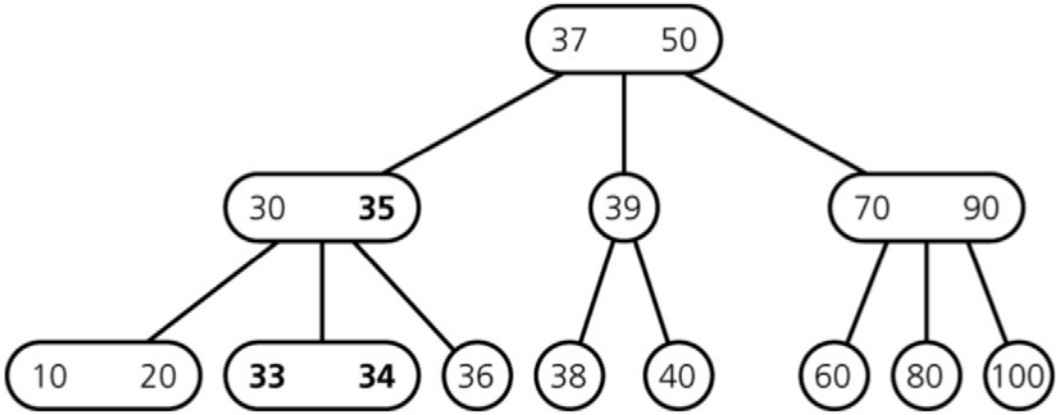


Split을 함으로써 외부 노드로 가는 경로의 길이는 모두 같게 된다.

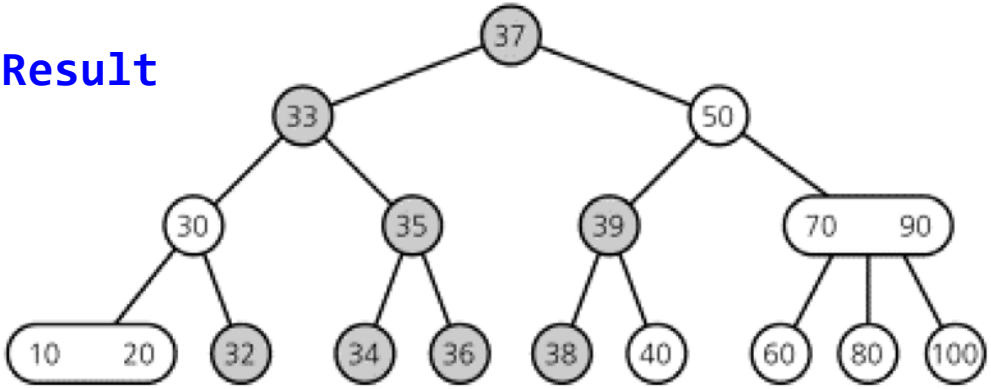
3-노드가 버퍼역할 → 트리의 편향을 완화

2-3 Tree Example

- "32"가 삽입될 때, 2-3 Tree는 어떻게 되는가?



Result



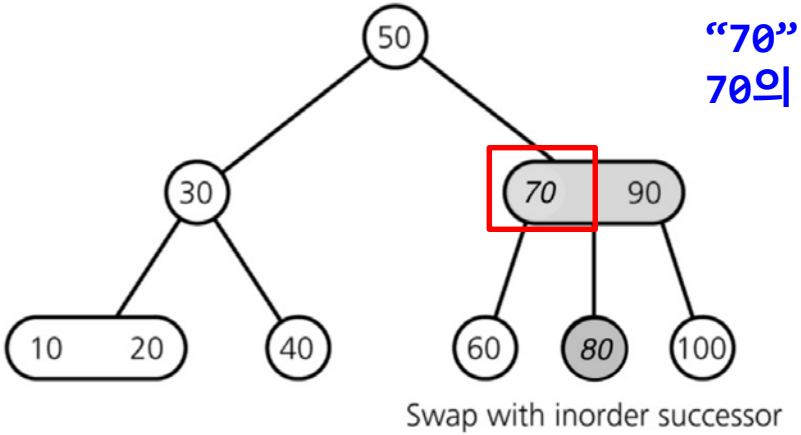
2-3 Tree 삭제 알고리즘

- **"삭제는 리프노드에서만 이루어짐"**

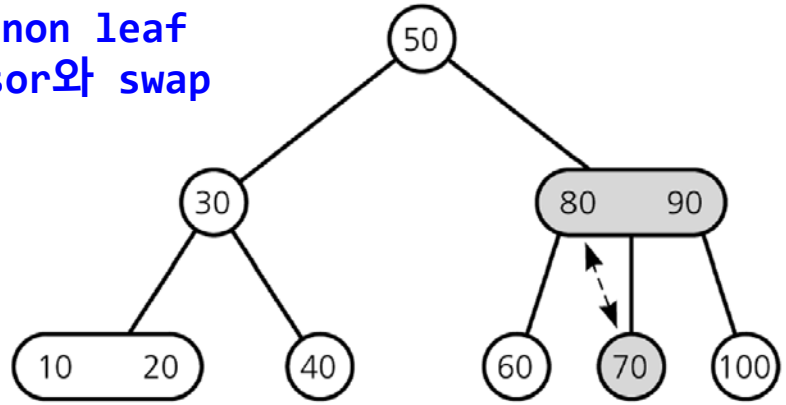
- **삭제할 아이템이 리프노드(leaf node)가 아니라면**
  - 삭제할 노드의 후속노드(successor)와 Swap한 후 노드 삭제

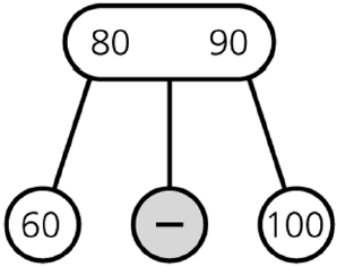
- **삭제할 아이템이 리프노드에 있다면**
  - 노드에 다른 아이템이 존재하면, 단순 삭제
  - 그렇지 않다면 형제노드(siblings)에서 아이템을 빌리고 삭제
  - 형제노드에서 빌릴 수 없다면 병합

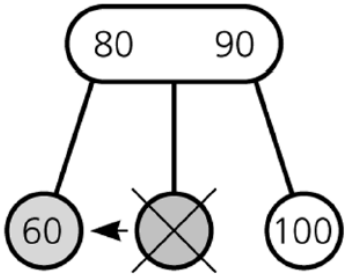
2-3 Tree Deletion Example



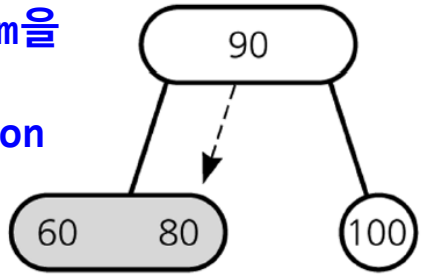
"70" 삭제 – non leaf
70의 successor와 swap

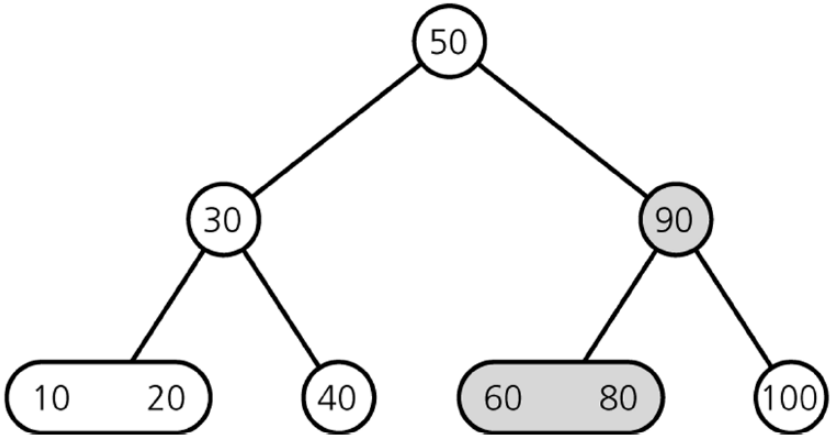






sibling에서 item을
빌릴 수 없다
→ Redistribution
(merge)

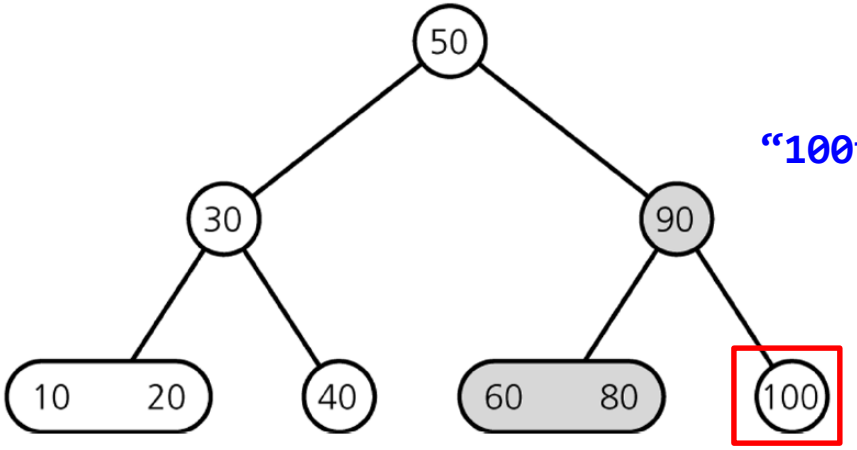




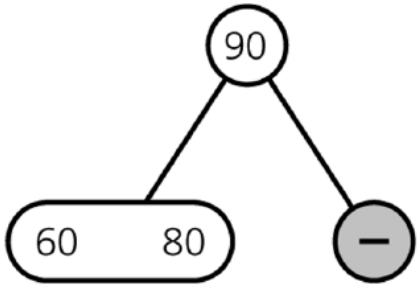
2-3 Tree Deletion Example

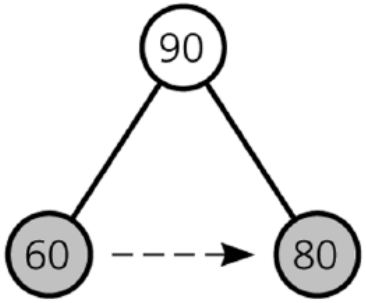
if leaf에서 삭제 가능?

else if sibling에서 차용 가능?

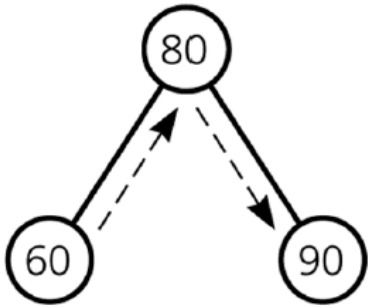


"100" 삭제

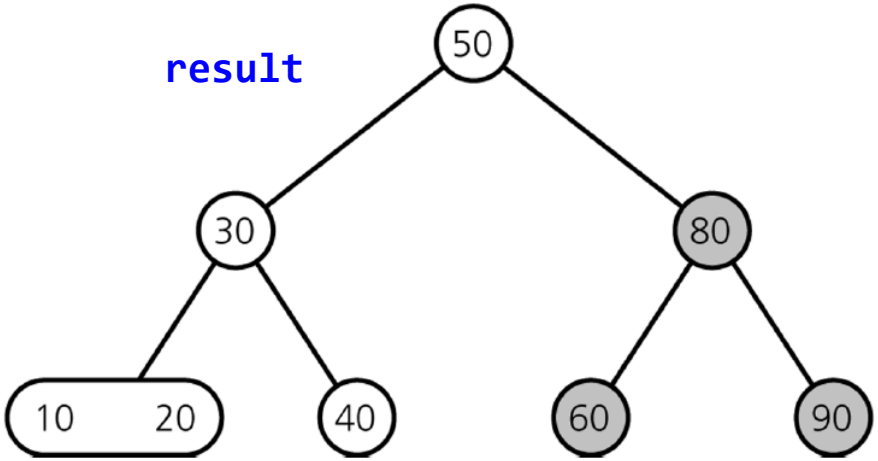




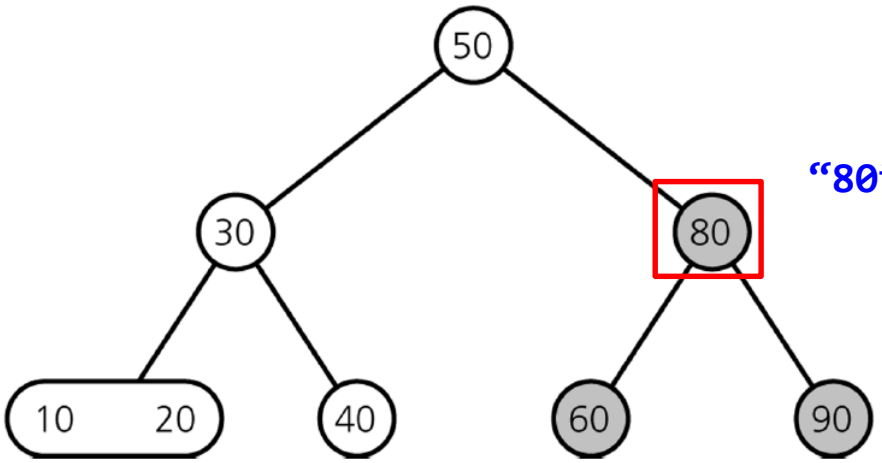
sibling에서 빌려옴



result

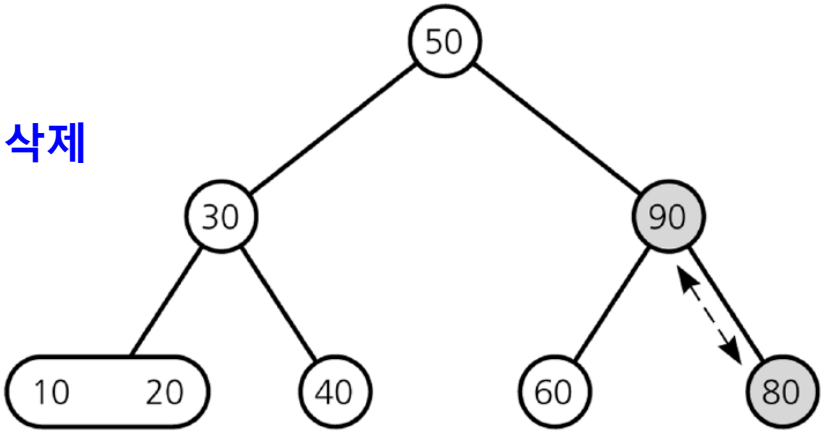


2-3 Tree Deletion Example

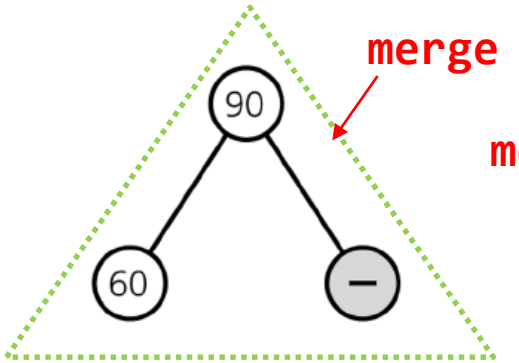


"80" 삭제

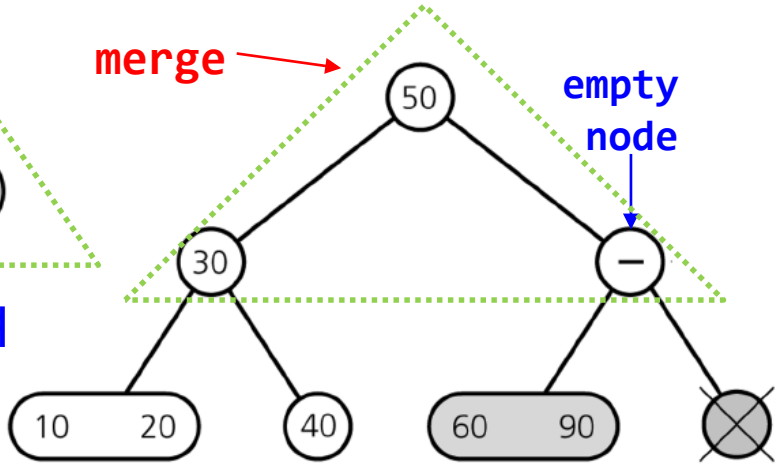
non leaf이면 successor와 swap



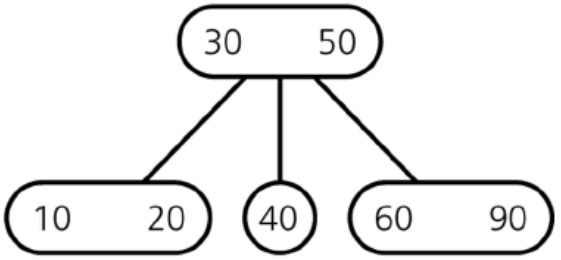
Swap with inorder successor



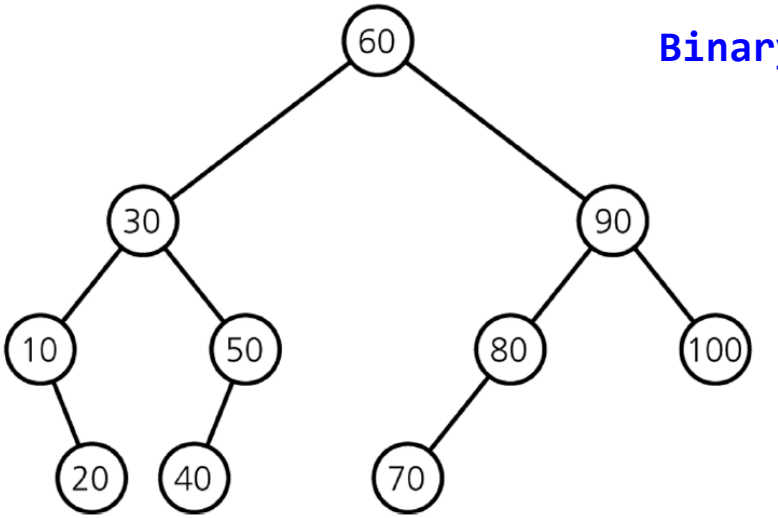
sibling에서 빌려 올 수 없음
→ merge



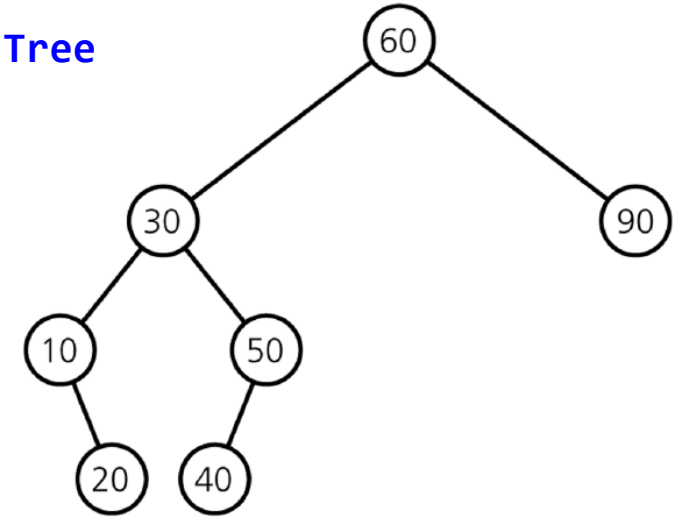
result



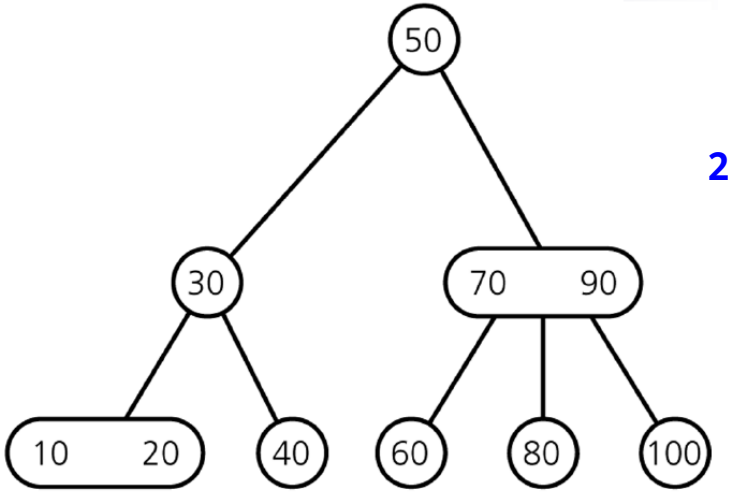
2-3 Tree Deletion Example



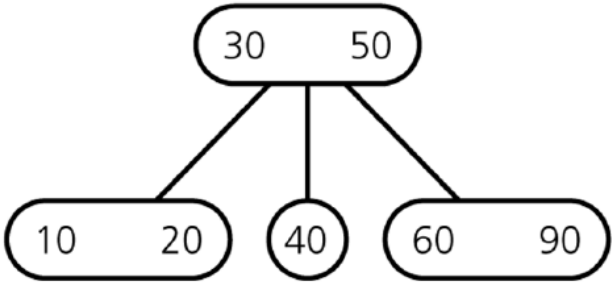
Binary Search Tree



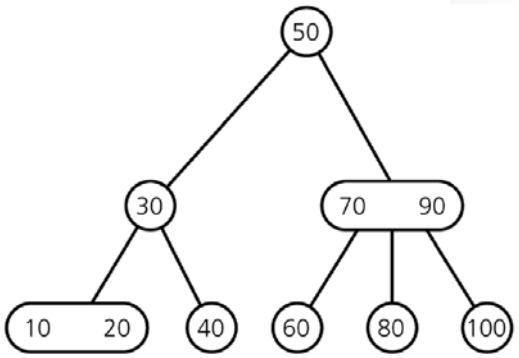
70, 100, 80
삭제



2-3 Tree

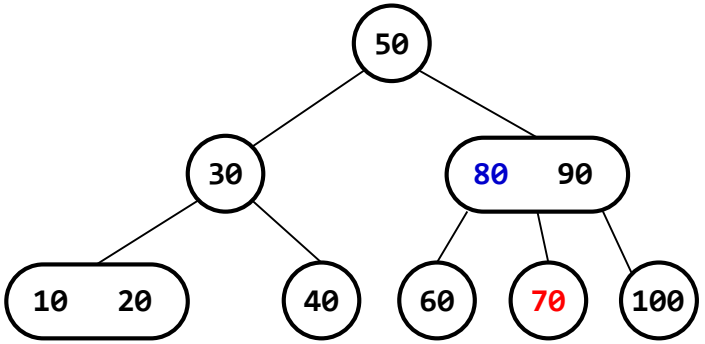


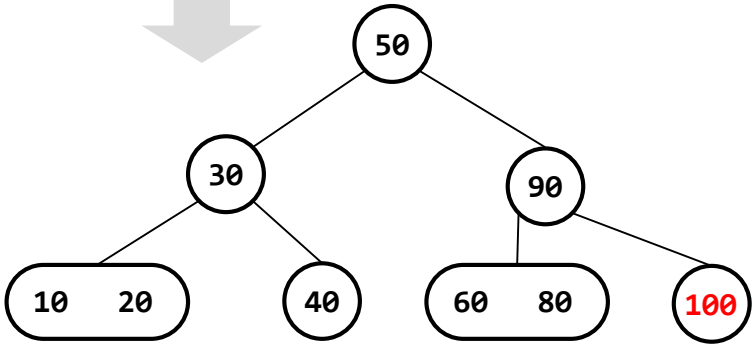
2-3 Tree Deletion Example

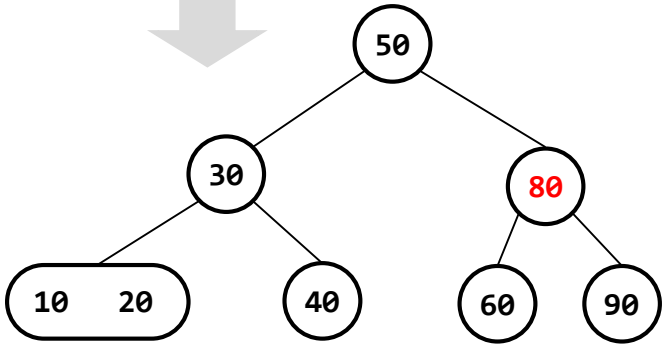


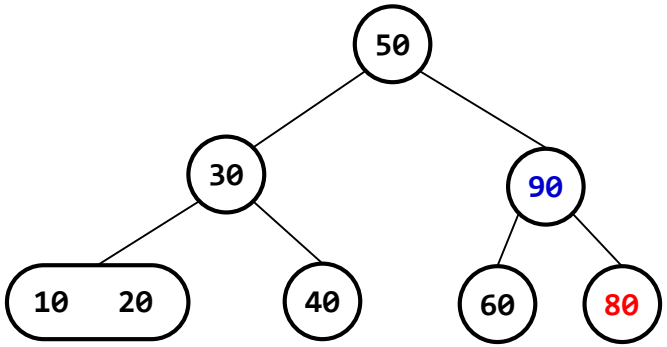
70, 100, 80
삭제

2-3 Tree

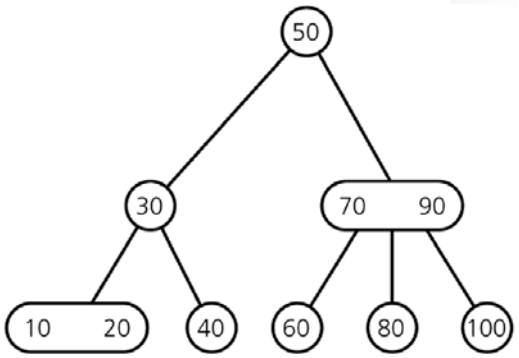






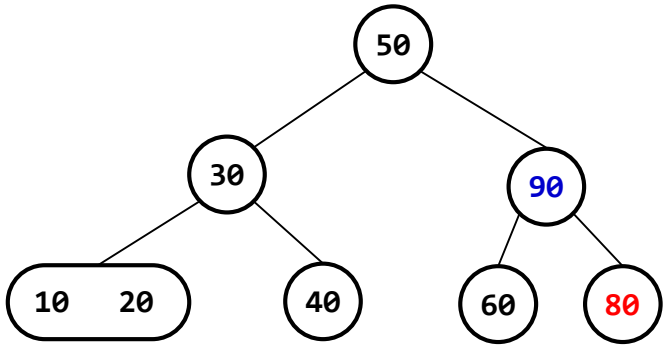


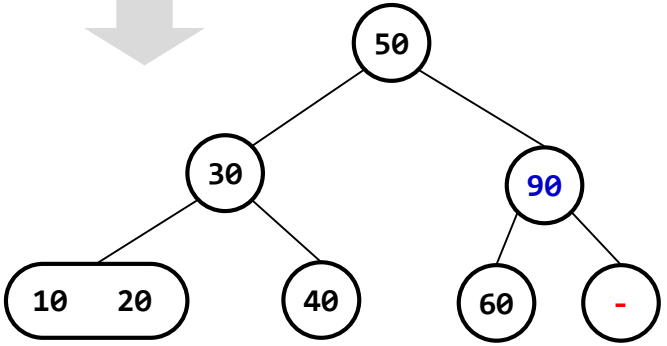
2-3 Tree Deletion Example

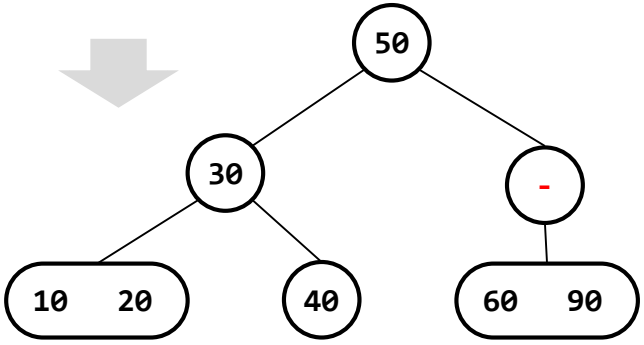


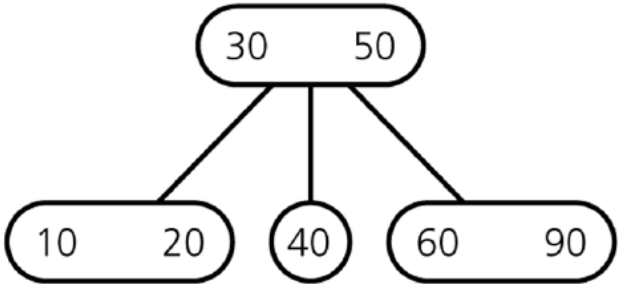
70, 100, 80
삭제

2-3 Tree







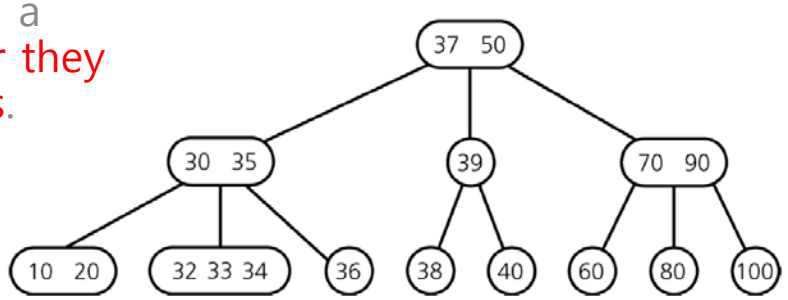


Rudolf Bayer (born 3 March 1939) is a German computer scientist.

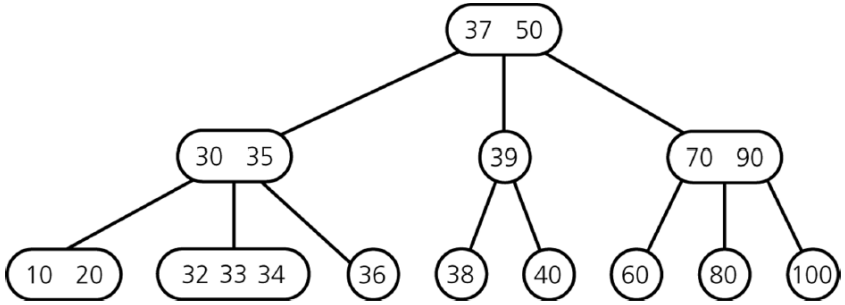
2-3-4 트리
(2-3-4 Trees)

In 1972, Rudolf Bayer invented a data structure that was a special order-4 case of a B-tree. These trees maintained all paths from root to leaf with the same number of nodes, creating perfectly balanced trees. However, they were not binary search trees. Bayer called them a "symmetric binary B-tree" in his paper and later they became popular as 2-3-4 trees or just 2-4 trees.

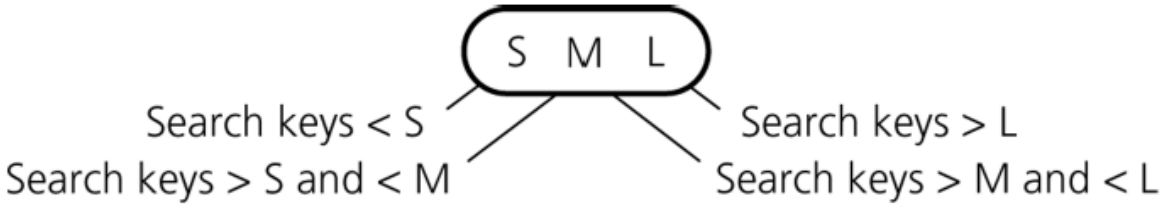
-- wikipedia



2-3-4 Tree Concept

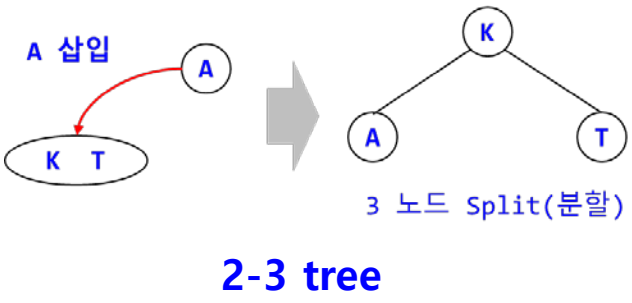


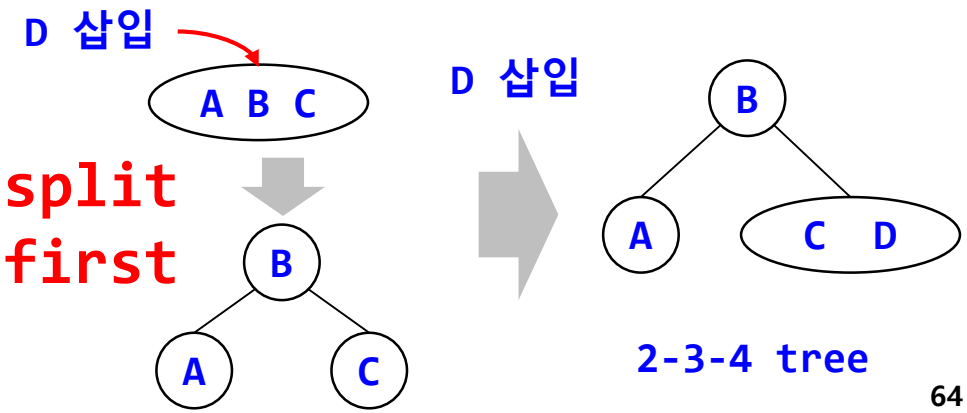
- **2–3–4 Tree = 2-4 Tree**
- **2-3-4 Tree = B tree of order 4**
  - 2-3 Tree = B tree of order 3
- **Red-Black Tree = 2-3-4 Tree의 Binary Search Tree version**
- **내부 노드(internal node)는 자식을 2개, 또는 3개, 또는 4개를 가짐**
  - **즉, 자식을 1개를 가질 수는 없음.**
- **Key 검색 = 2-3 Tree와 동일**



삽입(Insertion), 분할(Split)

- **2–3–4 Tree의 삽입 시 분할방식이 2-4 Tree와 다름**
  - 2-3 Tree:
    - 3-node의 두 개의 키와 새로 입력되는 키를 정렬
    - 중간 키가 부모, 작은 키는 부모의 왼편, 큰 키는 부모의 오른쪽에 위치
  - 2-3-4 Tree
    - 분할은 4-node에 새로운 Key가 삽입될 때 실시
    - 4-node의 중간 키를 부모로 올리고 작은 키를 부모의 왼쪽에 큰 키를 부모의 오른쪽에 둔 뒤에 새로운 키를 삽입



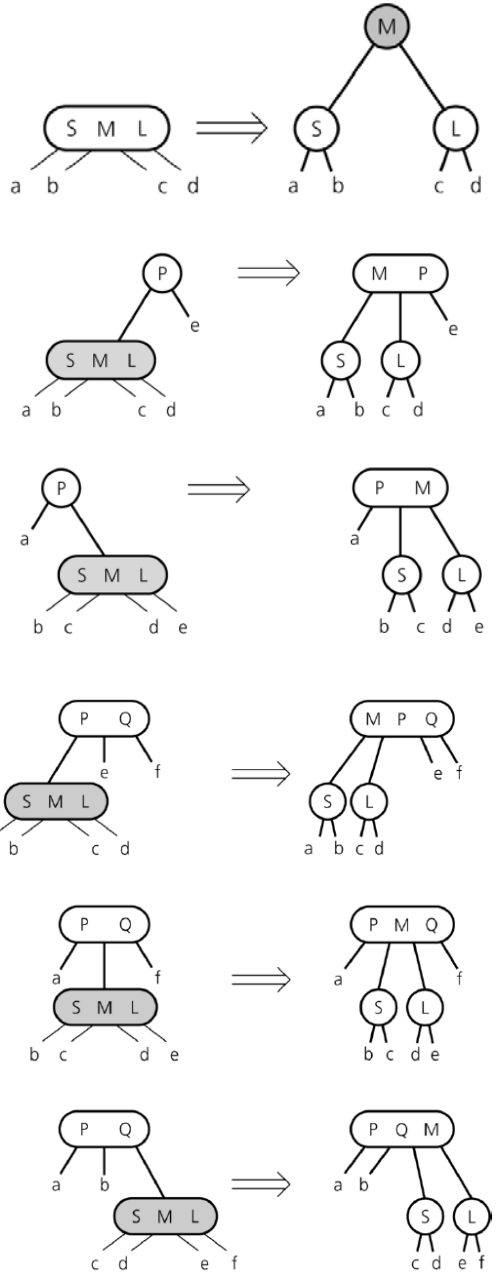


2-3-4 Tree 삽입 알고리즘

- **현재 노드가 외부 노드가 아닐 동안**
  - 현재 노드가 4-node이면 분할
  - 삽입 키가 첫 번째 키보다 작으면 첫 번째 자식으로
  - 삽입 키가 두 번째 키보다 작으면 두 번째 자식으로
  - 삽입 키가 세 번째 키보다 작으면 세 번째 자식으로
  - 삽입 키가 세 번째 키보다 크면 네 번째 자식으로

- **외부 노드가 4-node이면 분할**

- **외부 노드에 삽입 키를 삽입**



4-node가 split될 때
child link 정리 cases

삽입(Insertion), 분할(Split)

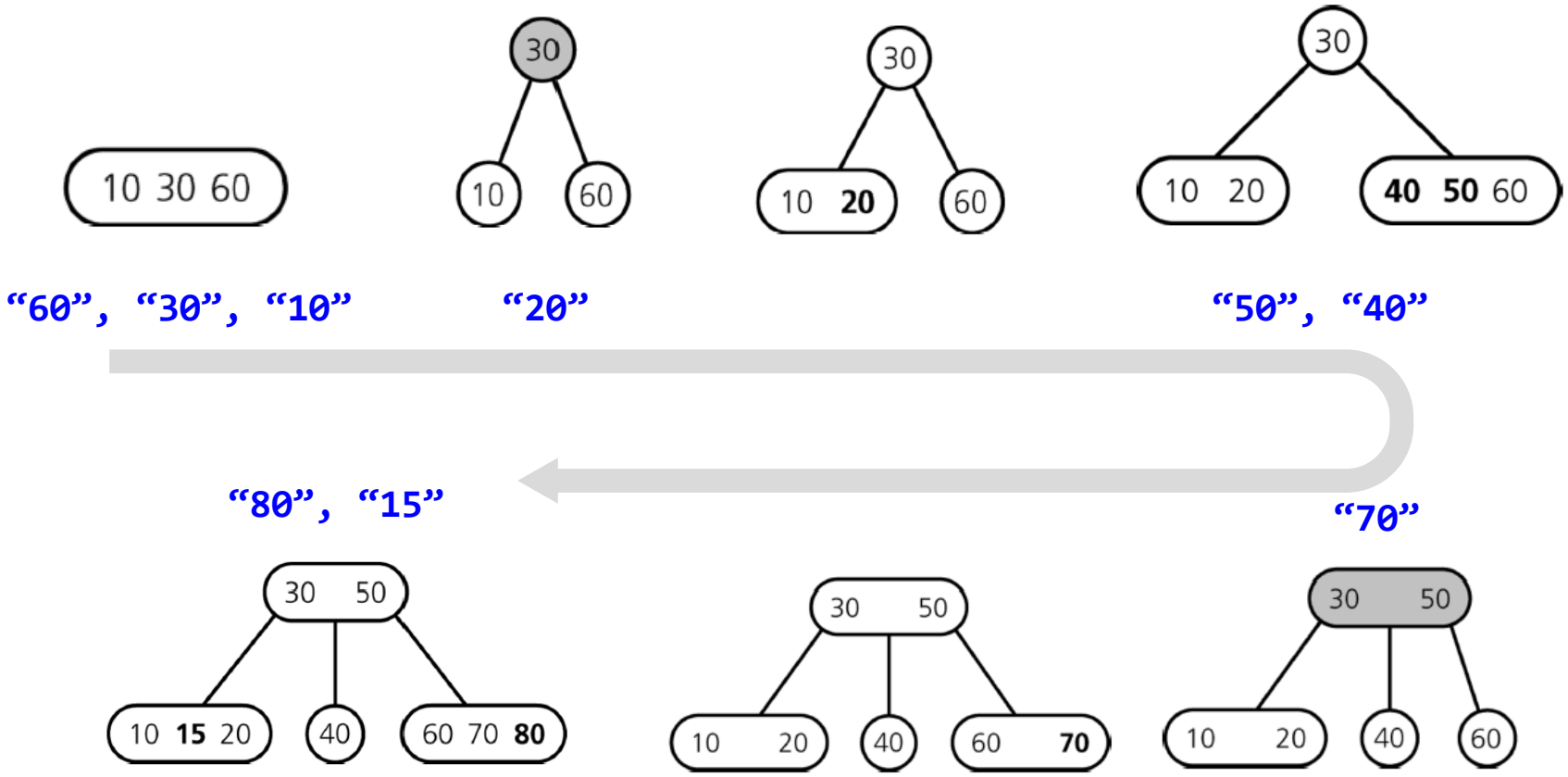
2-3-4 Tree의 노드분할 과정은

AVL Tree 처럼 삽입이나 삭제 후 전체적인 균형도를 조사할 필요가 없고,

2-3 Tree 처럼 삽입된 경로를 유지 해야 하는 불편이 없는 장점이 있다 (삽입 key가 분할의 대상이 되지 않는다.).

2-3-4 Tree Insertion Example

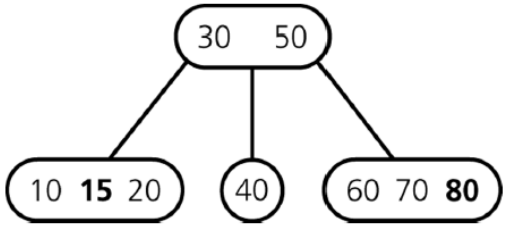
- **60, 30, 10, 20, 50, 40, 70, 80, 15, 90, 100**



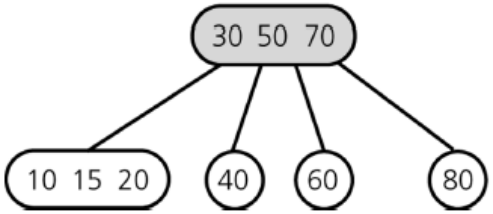
https://www.slideshare.net/sandpoonia/23-tree

2-3-4 Tree Insertion Example

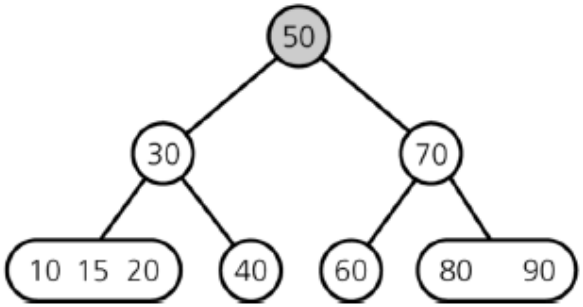
- **60, 30, 10, 20, 50, 40, 70, 80, 15, 90, 100**



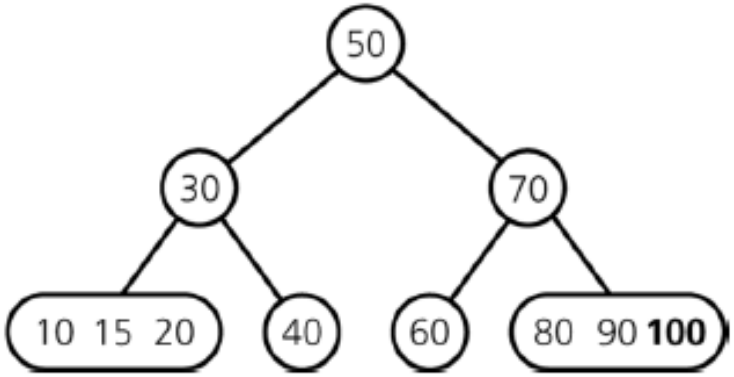
"80", "15"

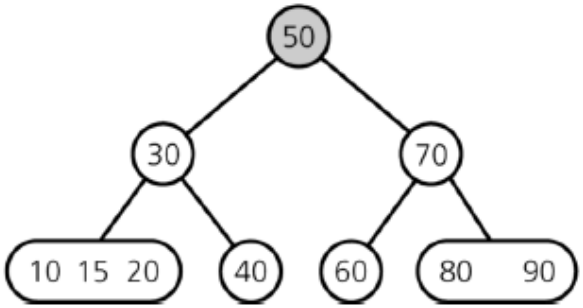


"90"



"100"



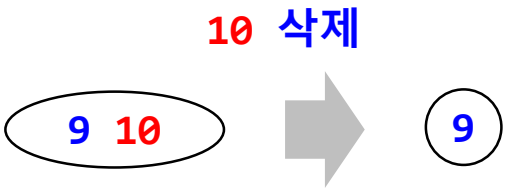


https://www.slideshare.net/sandpoonia/23-tree

2-3-4 Tree 삭제 알고리즘

- **"삭제는 리프노드에서만 이루어짐"**

- **Case 1: 3-node, 4-node leaf 일 때**
  - 단순 삭제



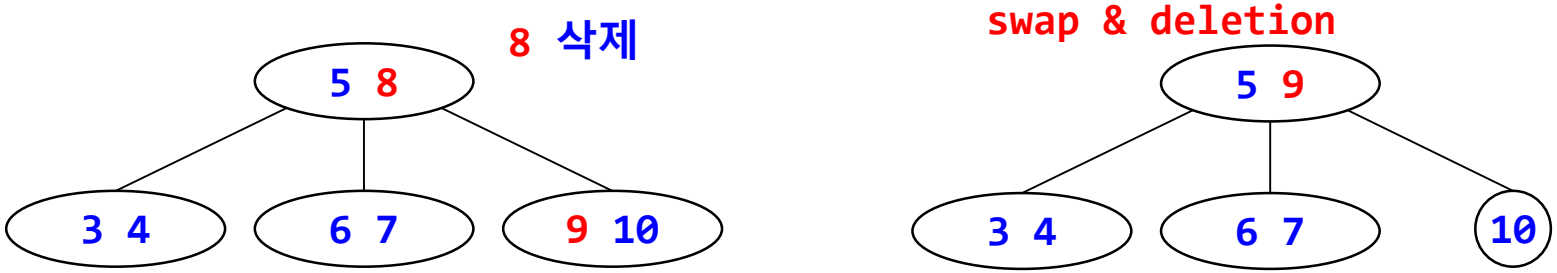
- **Case 2: non leaf node 일 때**
  - inorder successor key와 swap
  - leaf node가 3-node, 4-node → 단순 삭제

- **Case 3: 2-node leaf 일 때**
  - parent와 sibling이 모두 2-node가 아니면 borrow
  - sibling이 2-node → merge
  - parent가 2-node → merge

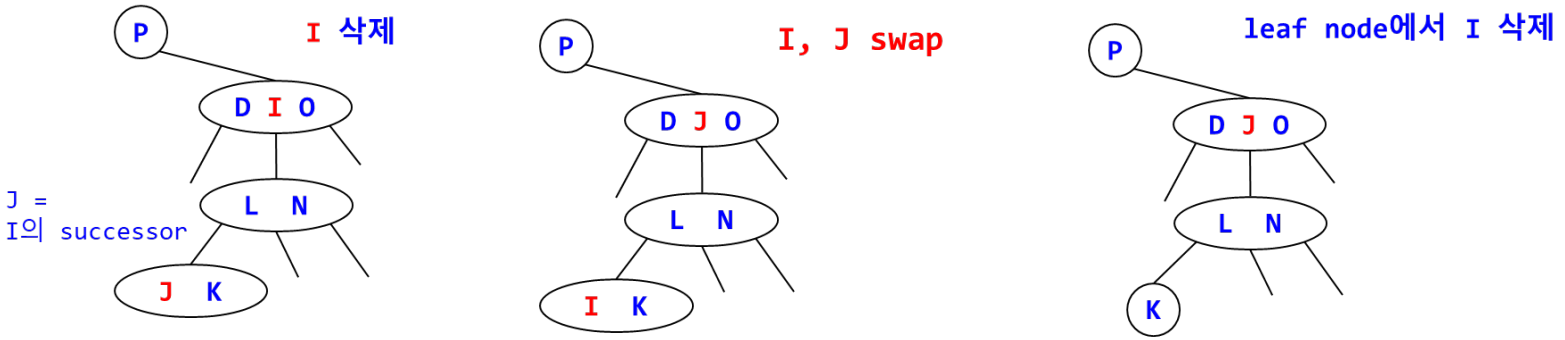
2-3-4 Tree 삭제 알고리즘

Case 2: non leaf node 일 때
inorder successor key와 swap
leaf node가 3-node, 4-node → 단순 삭제

example 1:



example 2:



2-3-4 Tree 삭제 알고리즘

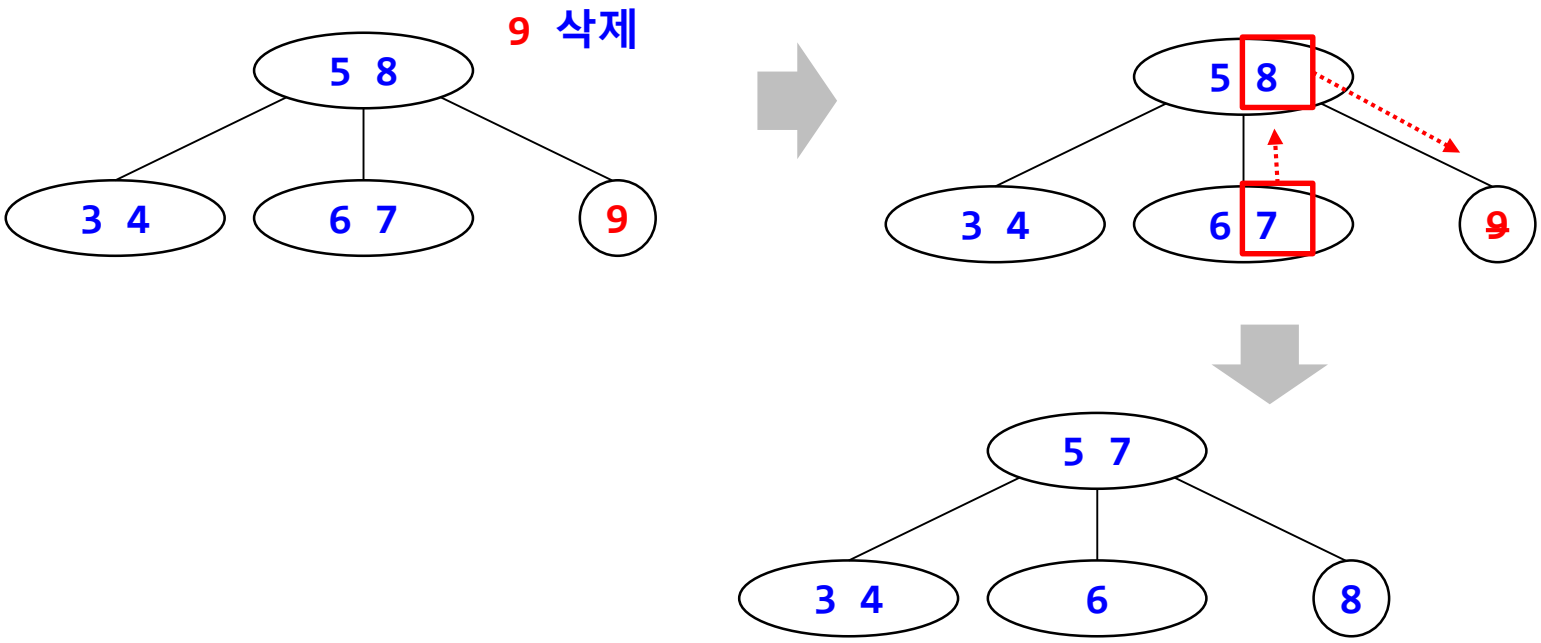
Case 3: 2-node leaf 일 때

parent와 sibling이 모두 2-node가 아니면 borrow

sibling이 2-node → merge

parent가 2-node → merge

sub case 1 - parent와 sibling이 모두 2-node가 아니면 borrow



2-3-4 Tree 삭제 알고리즘

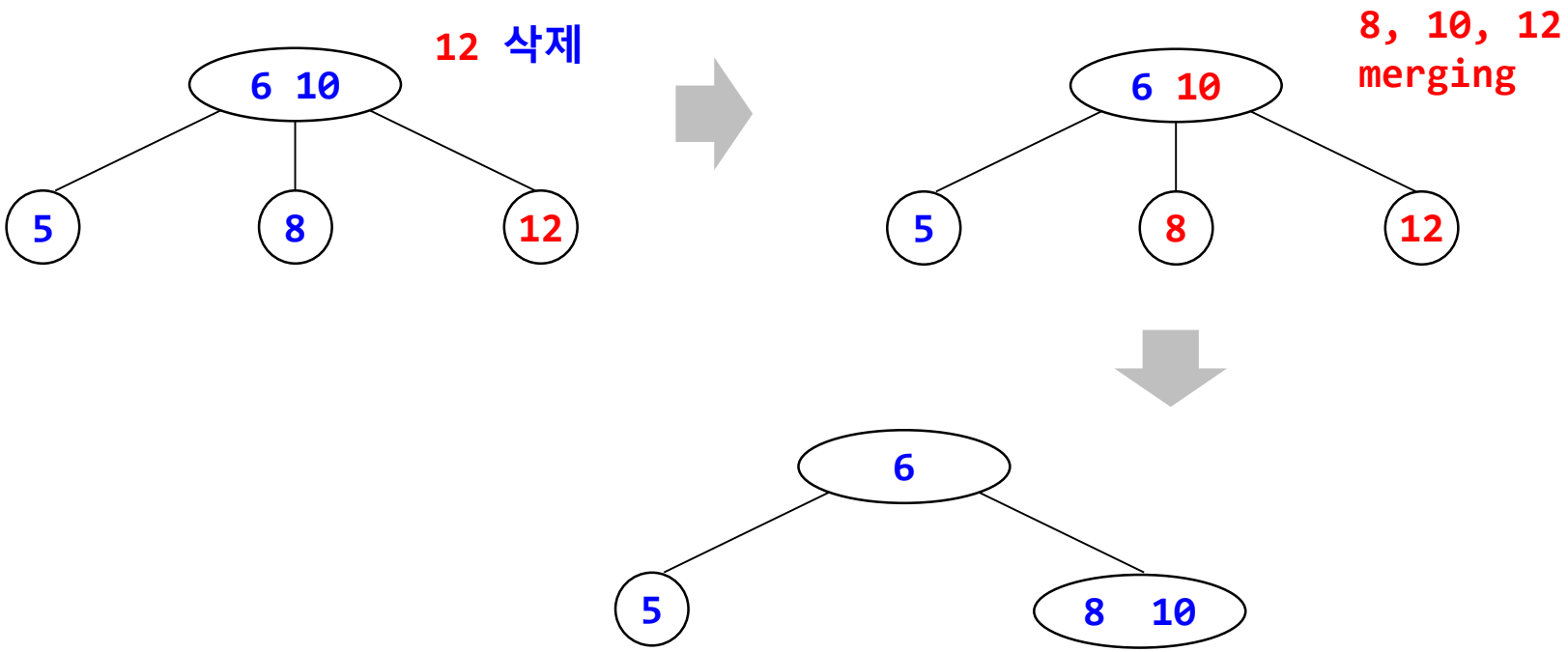
Case 3: 2-node leaf 일 때

parent와 sibling이 모두 2-node가 아니면 borrow

sibling이 2-node → merge

parent가 2-node → merge

sub case 2 - sibling이 2-node → merge



2-3-4 Tree 삭제 알고리즘

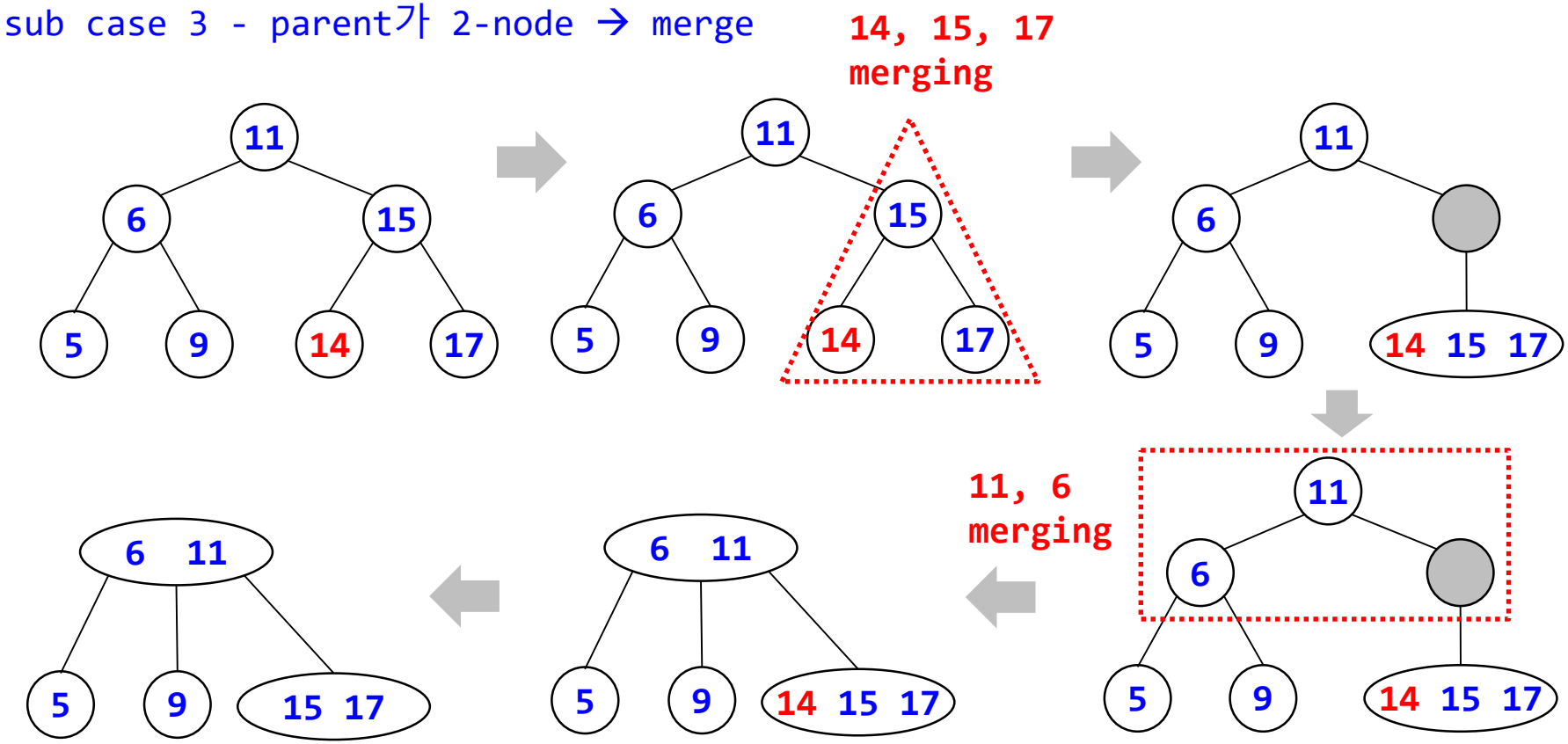
Case 3: 2-node leaf 일 때

parent와 sibling이 모두 2-node가 아니면 borrow

sibling이 2-node → merge

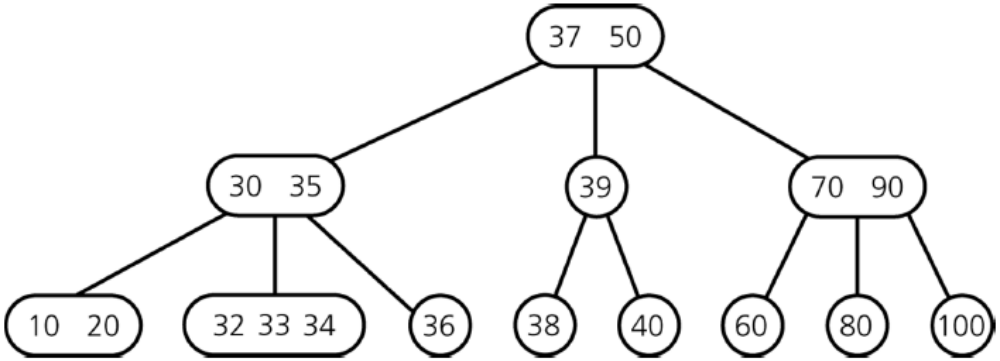
parent가 2-node → merge

sub case 3 - parent가 2-node → merge

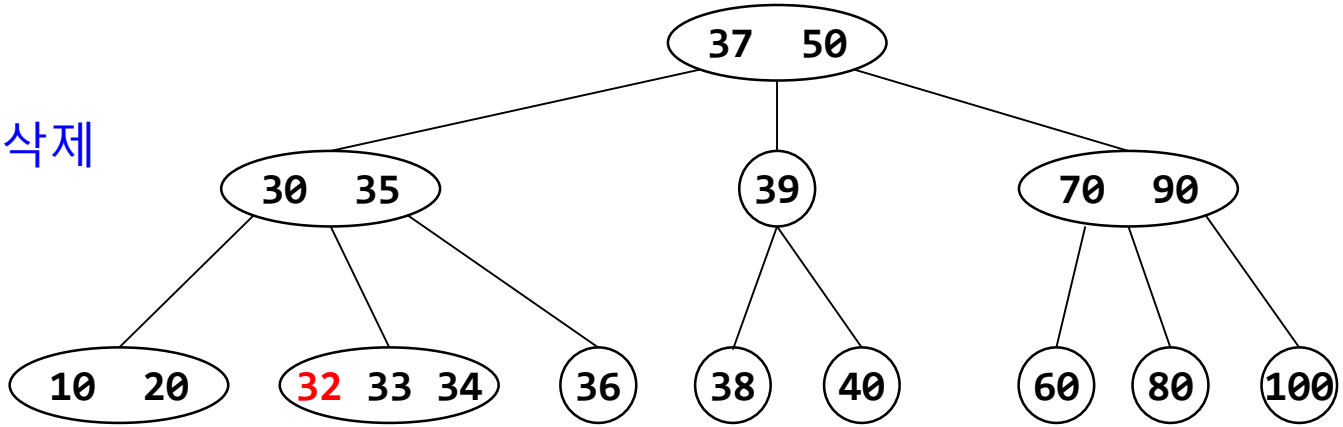


2-3-4 Tree Deletion Example

- **32, 35, 40, 38, 39, 37, 60**



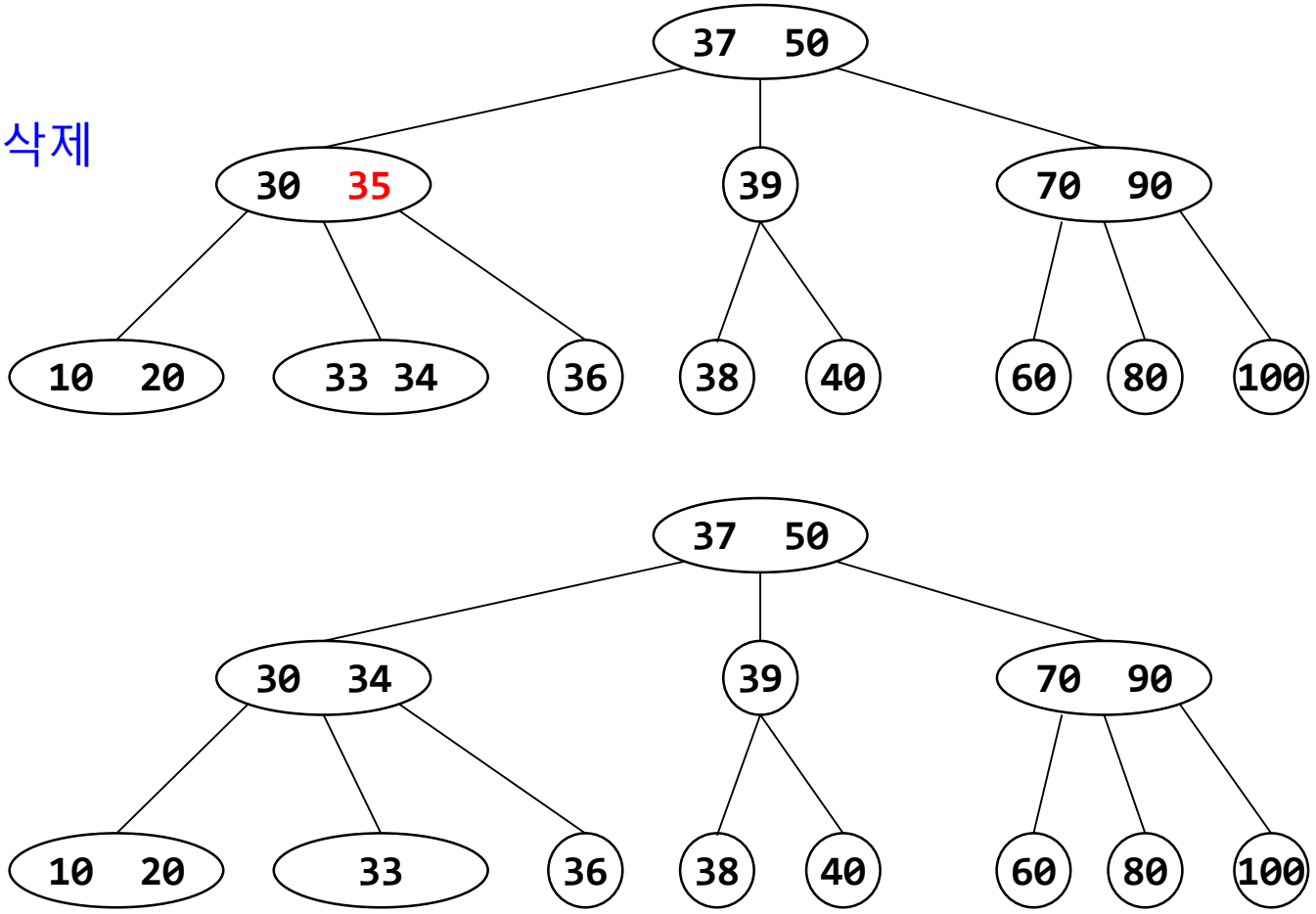
"32" 삭제



2-3-4 Tree Deletion Example

- **32, 35, 40, 38, 39, 37, 60**

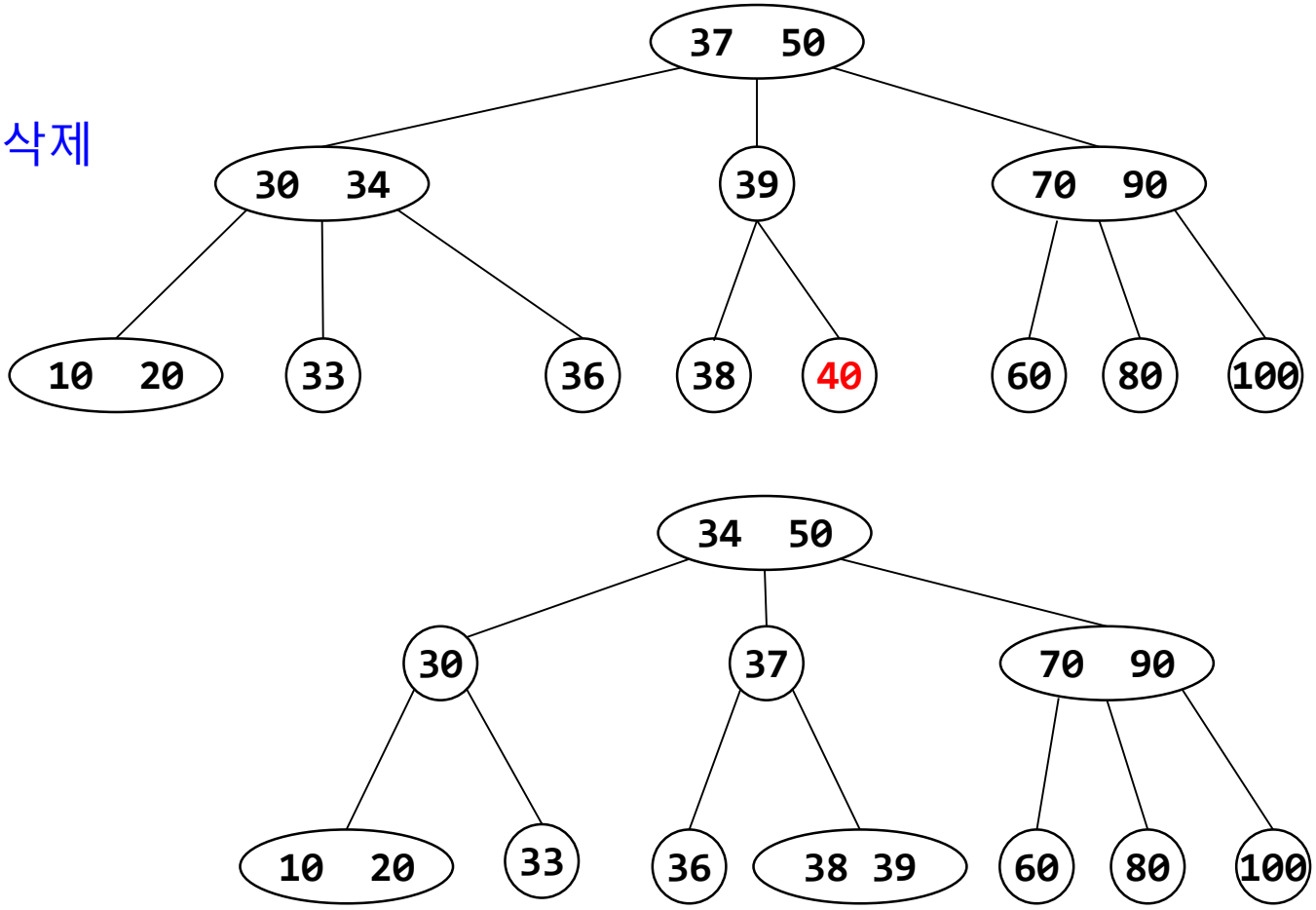
"35" 삭제



2-3-4 Tree Deletion Example

- **32, 35, 40, 38, 39, 37, 60**

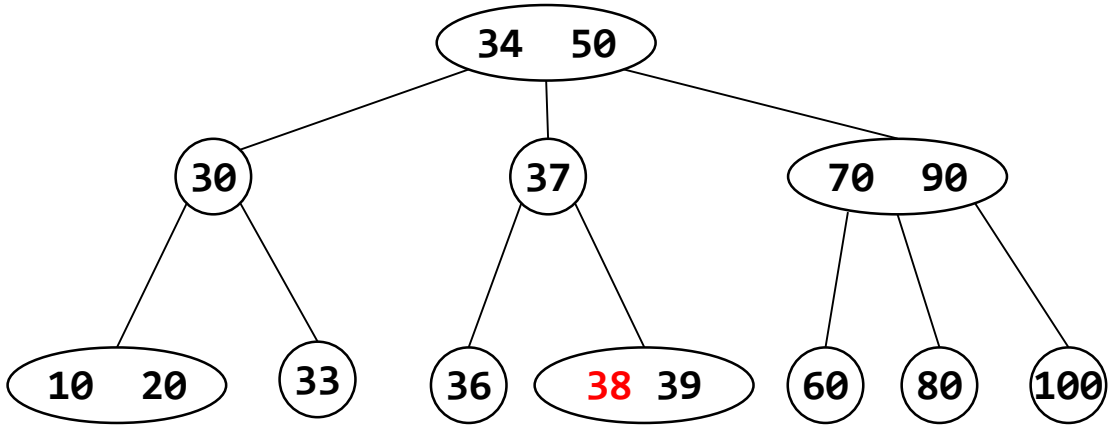
"40" 삭제

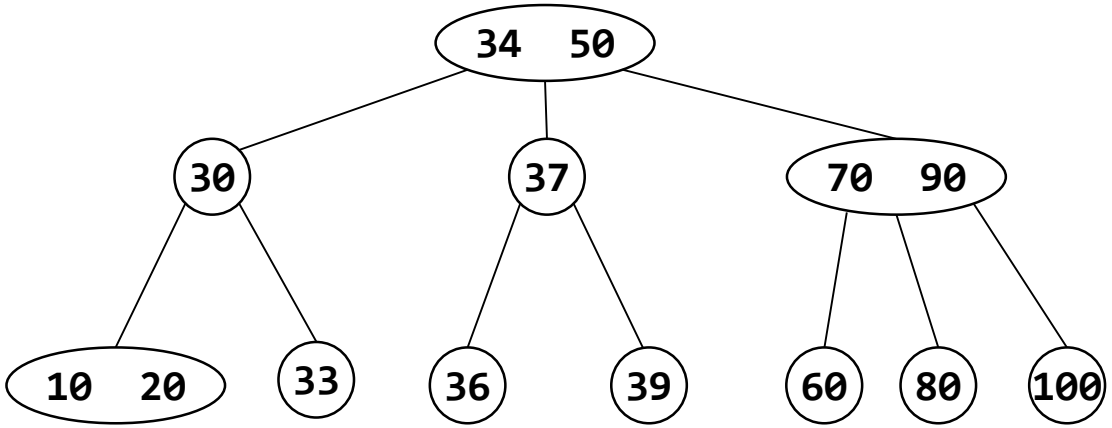


2-3-4 Tree Deletion Example

- **32, 35, 40, 38, 39, 37, 60**

"38" 삭제

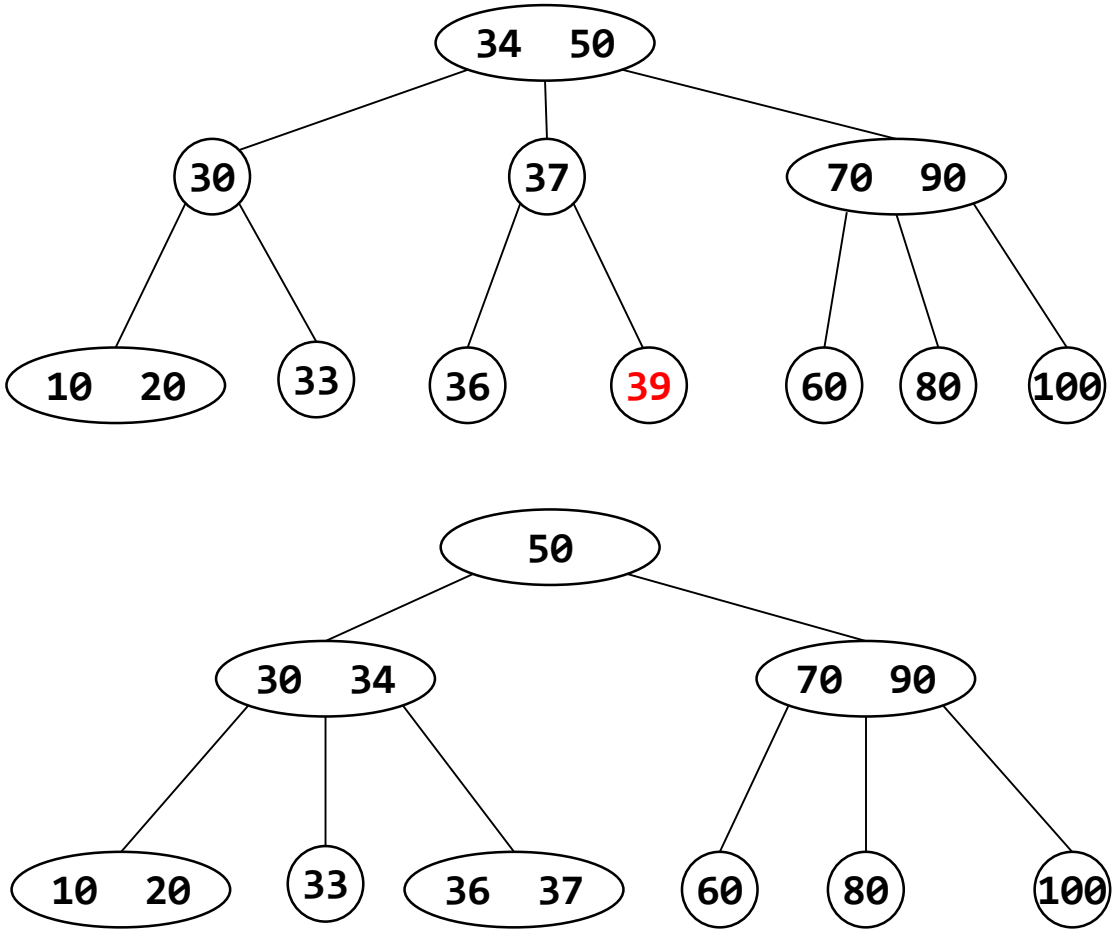




2-3-4 Tree Deletion Example

- **32, 35, 40, 38, 39, 37, 60**

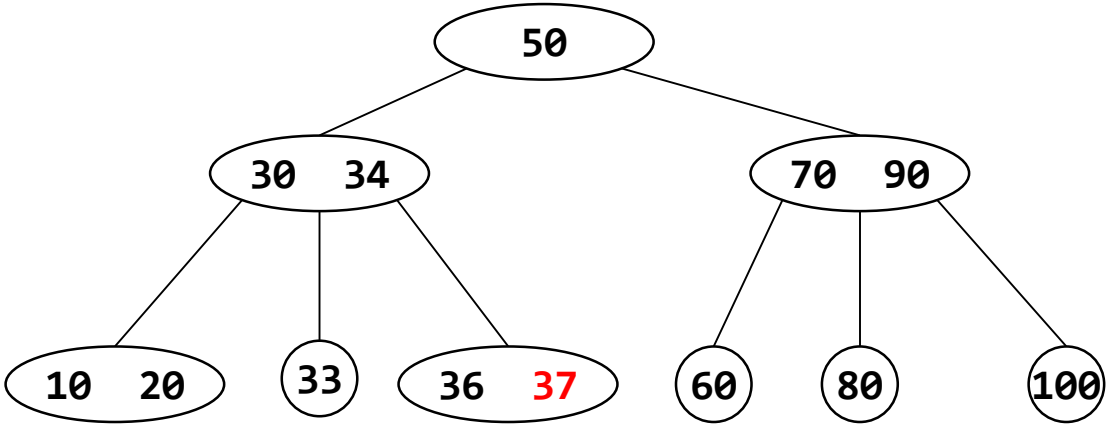
"39" 삭제

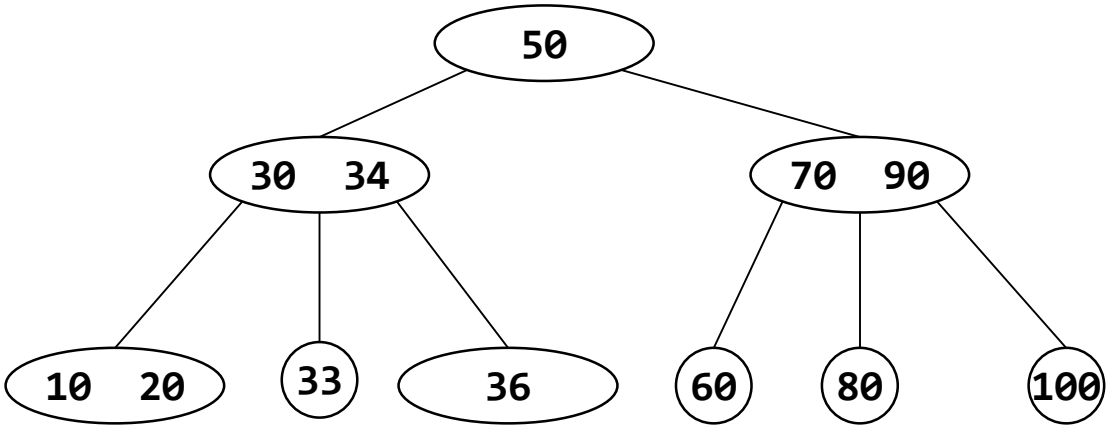


2-3-4 Tree Deletion Example

- **32, 35, 40, 38, 39, 37, 60**

"37" 삭제

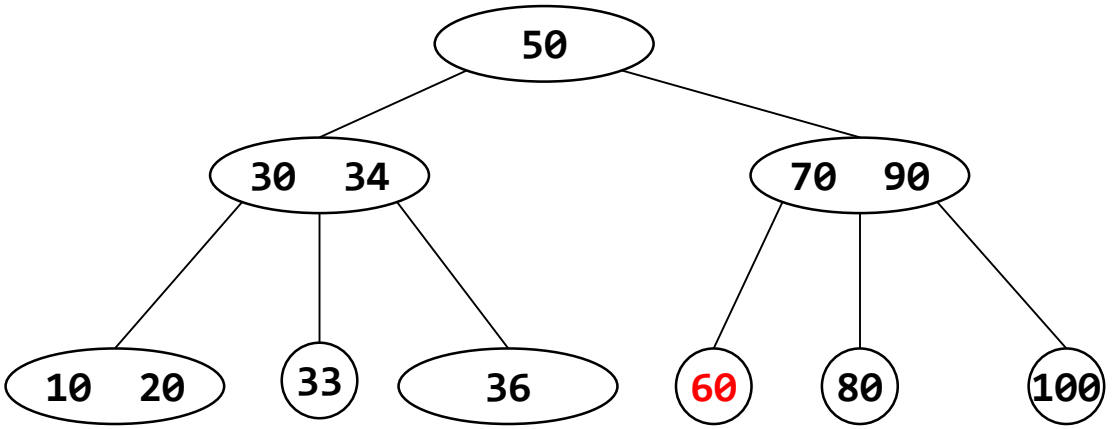


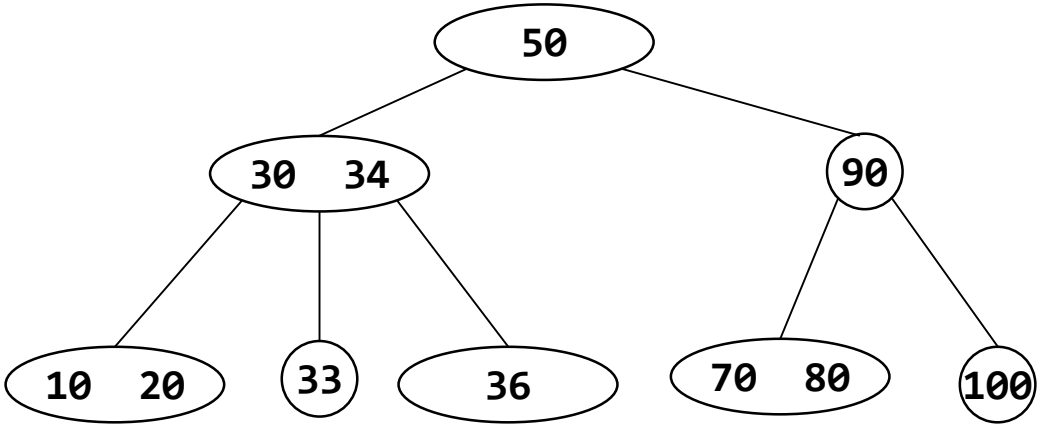


2-3-4 Tree Deletion Example

- **32, 35, 40, 38, 39, 37, 60**

"60" 삭제





2-3-4 Tree 장점 vs. 단점

- **장점**
  - AVL Tree 처럼 삽입, 삭제 후에 균형도를 맞출 필요가 없다.
  - 2-3 Tree 처럼 삽입된 경로를 유지해야 하는 불편이 없다.
    - 삽입노드가 분할의 대상이 되지 않는다.

- **단점**
  - 2-node, 3-node의 경우 메모리 낭비가 발생
  - 2-3-4 Tree는 내부 검색법(internal search, RAM 사용)으로 사용하기에는 부적절

단점을 극복하기 위한 방법

- 내부검색 → node 크기를 동일하게 한다 → red-black tree
- 외부검색 → B Tree*의 개념으로 활용

(*2-3-4 Tree = B Tree of order of 4)

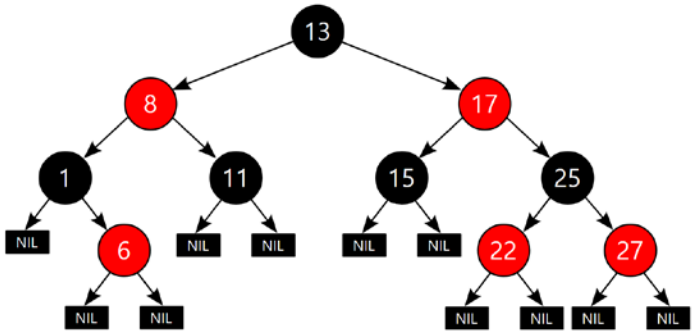

Leonidas John Guibas is the Paul Pigott Professor of Computer Science and Electrical Engineering at Stanford University

Robert Sedgewick (born December 20, 1946) is an American computer science professor at Princeton University

레드-블랙 트리
(Red-Black Trees)

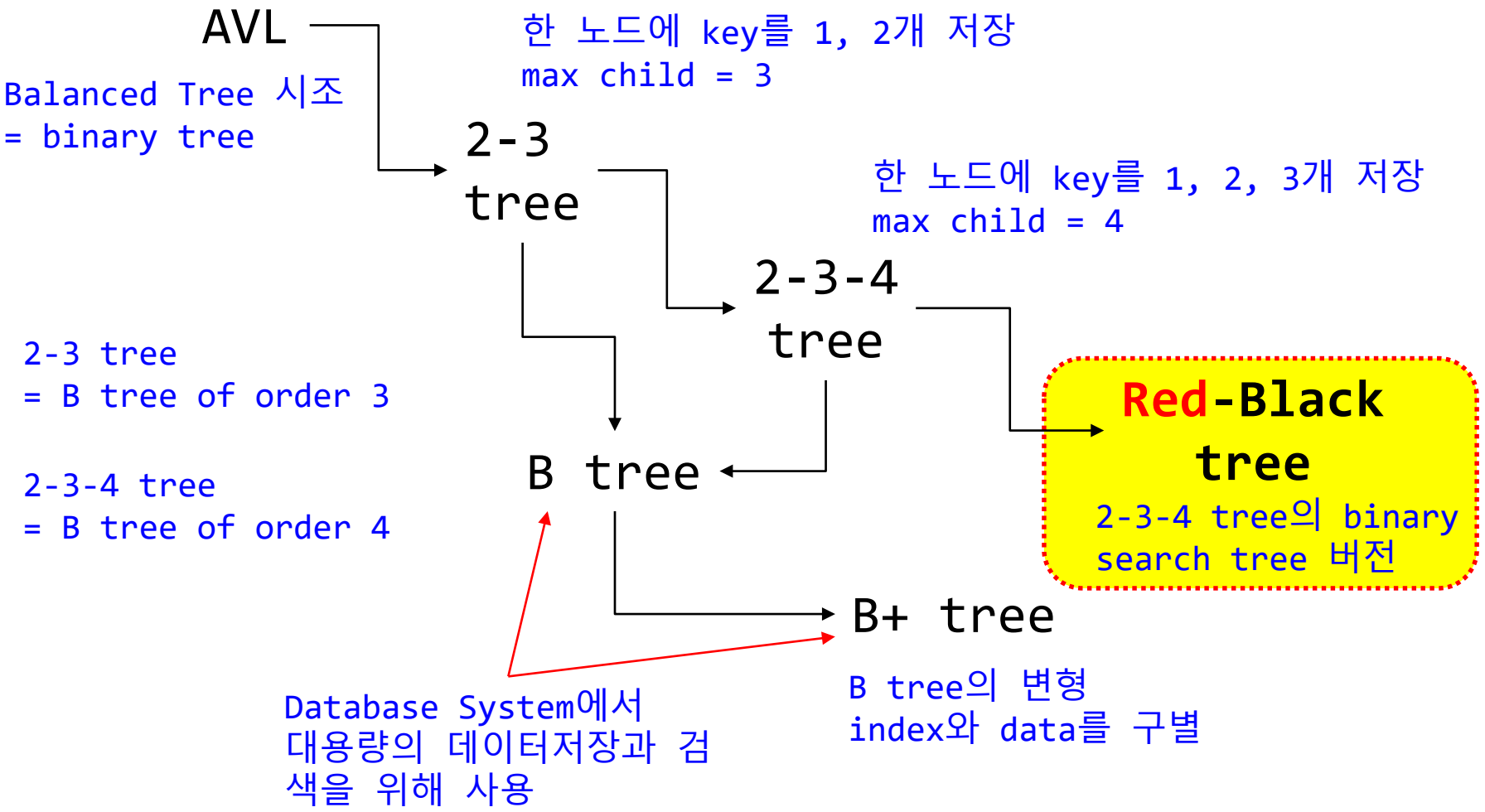
A red–black tree is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

--wikipedia



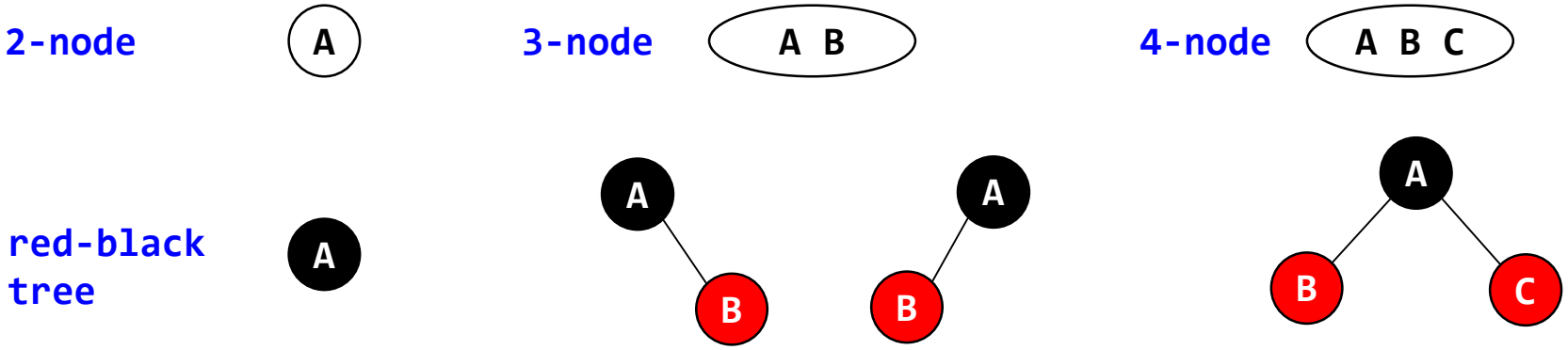
개념(Overview)

- **Balanced Binary Search Tree**

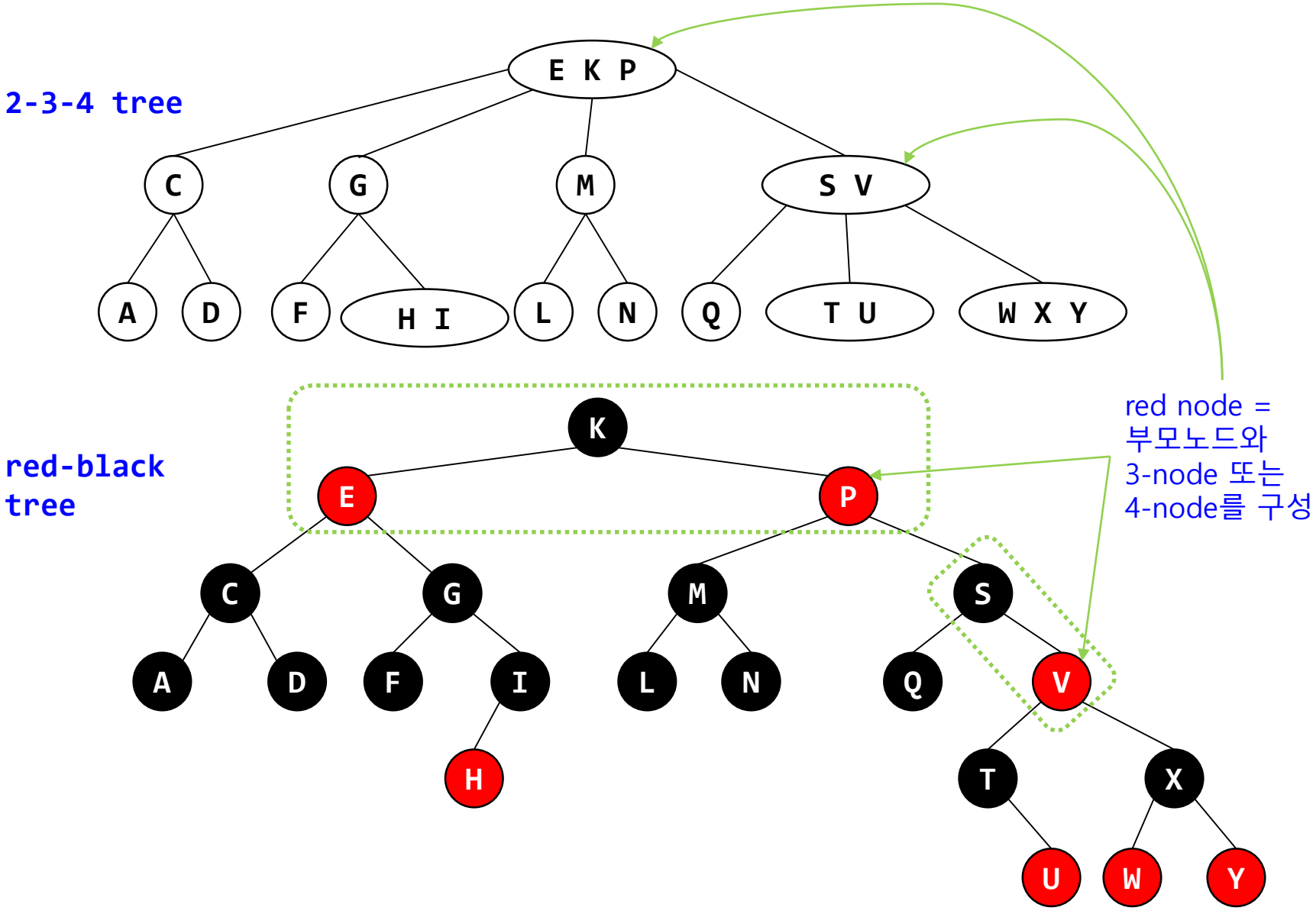


개요(Introduction)

- Red-Black Tree는 2-3-4 Tree의 기본 개념을 활용
- Red-Black Tree는 Binary Search Tree
- Node는 color(색)을 가짐 – red, black
- 2-3-4 node를 color로 구분



2-3-4 Tree ≡ Red-Black Tree

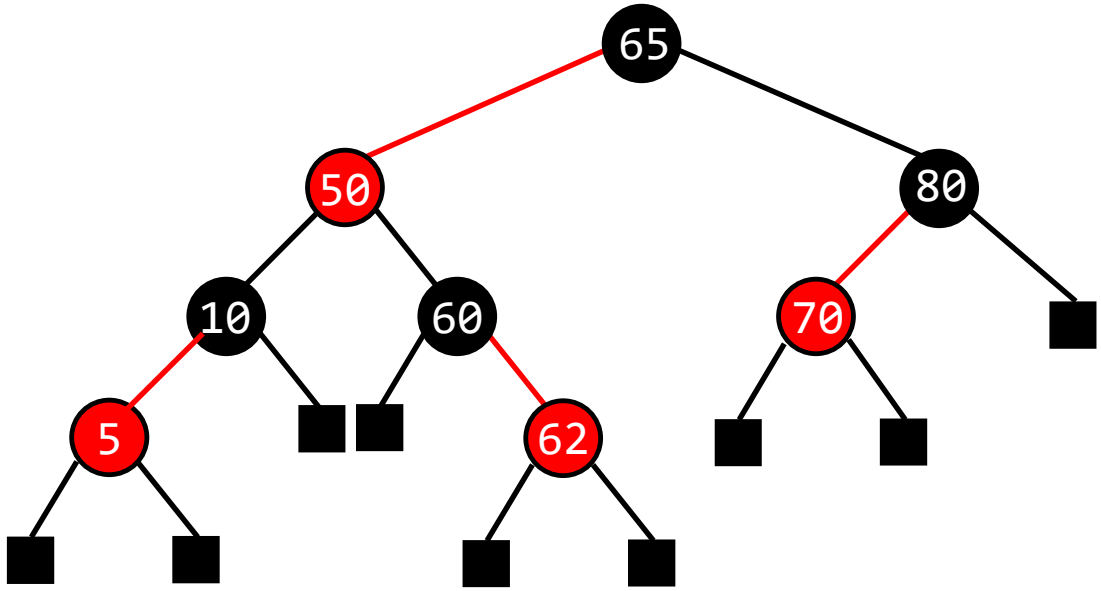


정의(Definition)

- RB1. 루트와 모든 외부 노드들은 컬러가 블랙이다.
- RB2. 루트에서 외부 노드로의 경로는 두 개의 연속적인 레드 노드를 가질 수 없다.
- RB3. 루트에서 외부 노드로의 모든 경로들에 있는 블랙 노드의 수는 동일하다.
- RB4. 삽입노드는 모두 레드노드이다. (추가사항)

red-black tree 이해를 위해
교재 link color에 대해서는 고려하지 말 것

정의(Definition)



블랙 포인터

레드 포인터

노드(레드) 삽입

→ red-black tree 규칙 위배

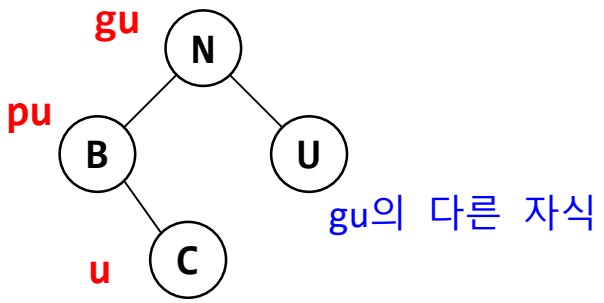
→ Rebalancing 작업 필요

Rotation, Color Flip

RB2 - 불균형 종류(Types of imbalance)

u: 새로운 노드

- LLb: pu는 gu의 왼쪽 자식, u는 pu의 왼쪽 자식, gu의 다른 자식이 블랙
- LLr: pu는 gu의 왼쪽 자식, u는 pu의 왼쪽 자식, gu의 다른 자식이 레드
- LRb: pu는 gu의 왼쪽 자식, u는 pu의 오른쪽 자식, gu의 다른 자식이 블랙
- LRr: pu는 gu의 왼쪽 자식, u=pu의 오른쪽 자식, gu의 다른 자식이 레드
- RRb: pu는 gu의 오른쪽 자식, u=pu의 오른쪽 자식, gu의 다른 자식이 블랙
- RRr: pu는 gu의 오른쪽 자식, u=pu의 오른쪽 자식, gu의 다른 자식이 레드
- RLb: pu는 gu의 오른쪽 자식, u=pu의 왼쪽 자식, gu의 다른 자식이 블랙
- RLr: pu는 gu의 오른쪽 자식, u=pu의 왼쪽 자식, gu의 다른 자식이 레드



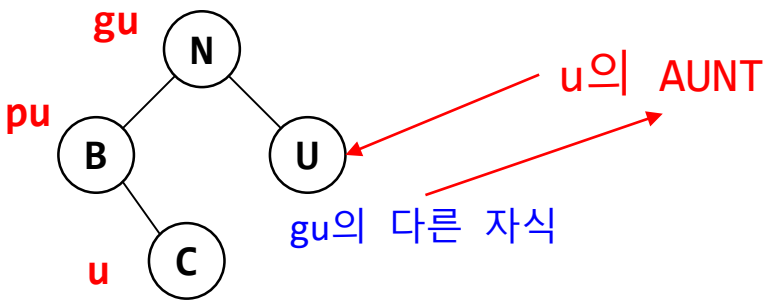
LRb or LRr

요약하면...

XYb type imbalance → rotation

XYr type imbalance → color flip

불균형 종류(Types of imbalance)



LRb or LRr

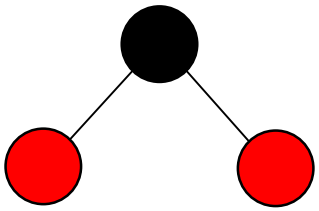
XYb type imbalance → rotation

BLACK AUNT → ROTATION (BAR)

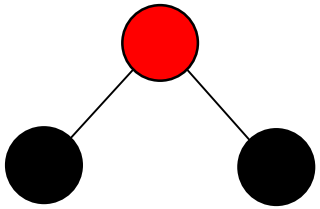
XYr type imbalance → color flip

RED AUNT → COLOR FLIP

AFTER ROTATION

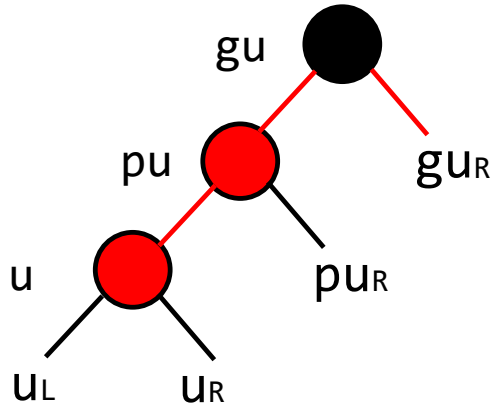


AFTER COLOR FLIP

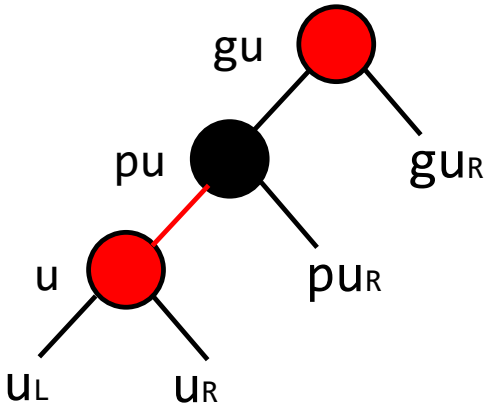


Color Flip

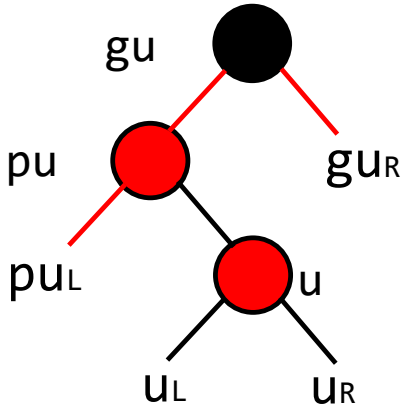
LLr 불균형



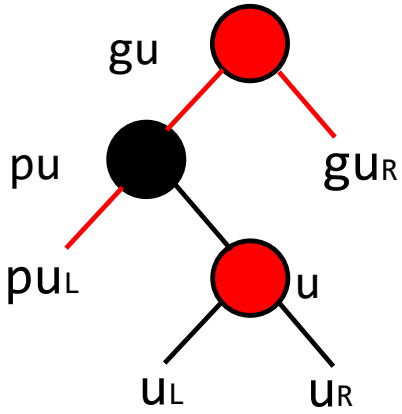
After Color Flip



LRr 불균형

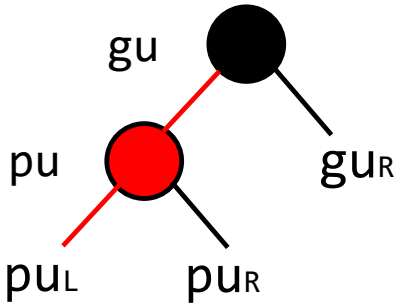


After Color Flip

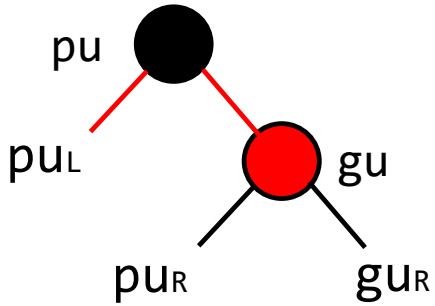


Rotation

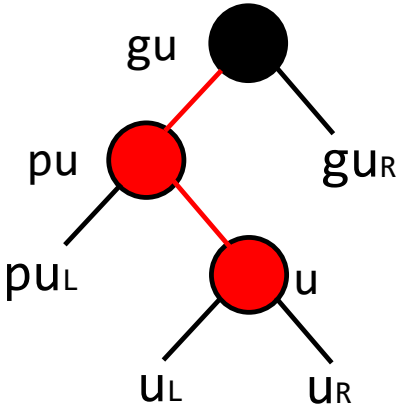
LLb 불균형



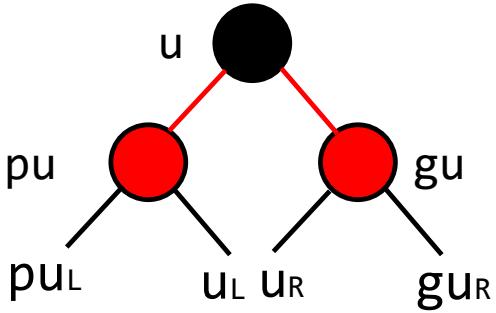
After Rotation



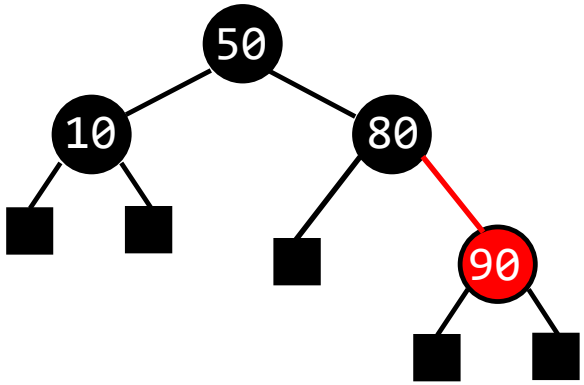
LRb 불균형



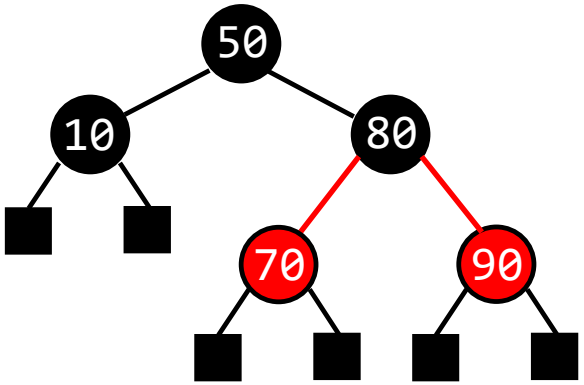
After Rotation



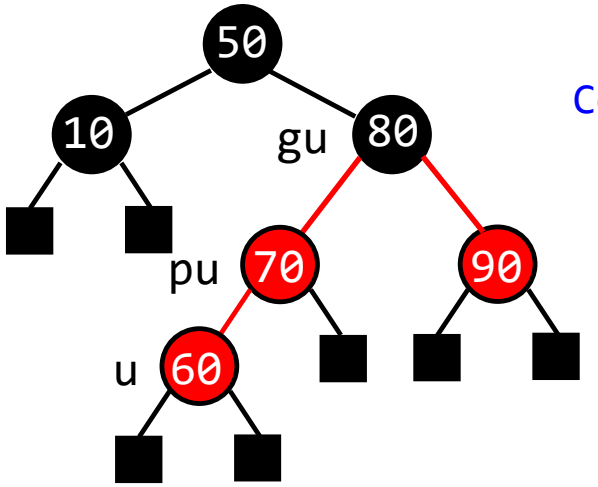
(교재)Example - 1



(a) 초기

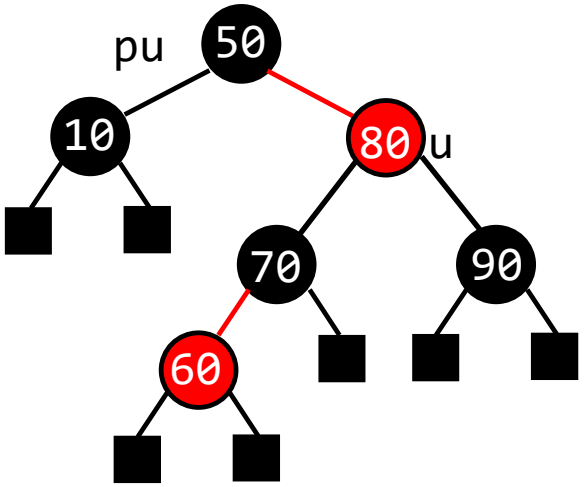


(b) 70을 삽입



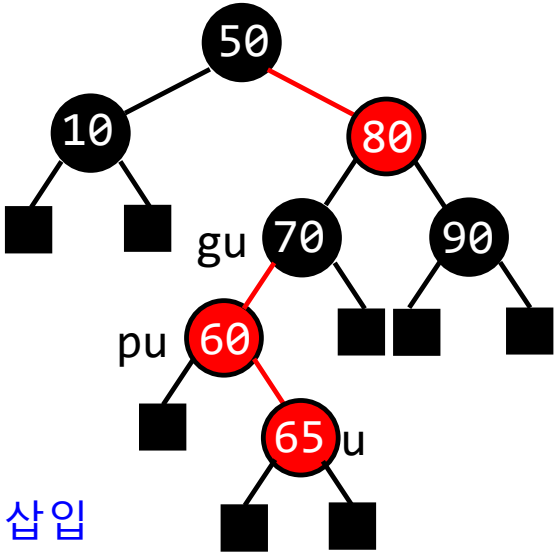
(c) 60을 삽입

Color Flip



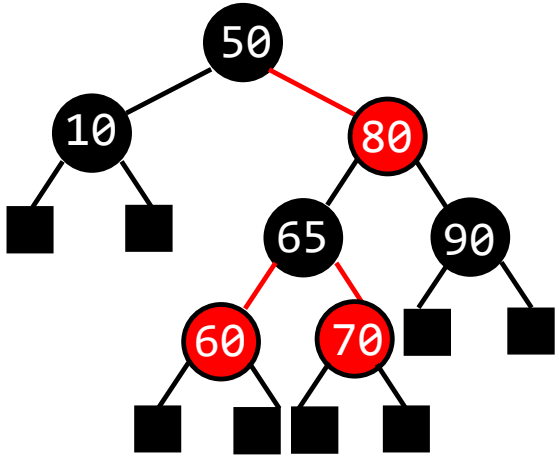
(d) LLr 컬러 변환

(교재)Example - 1

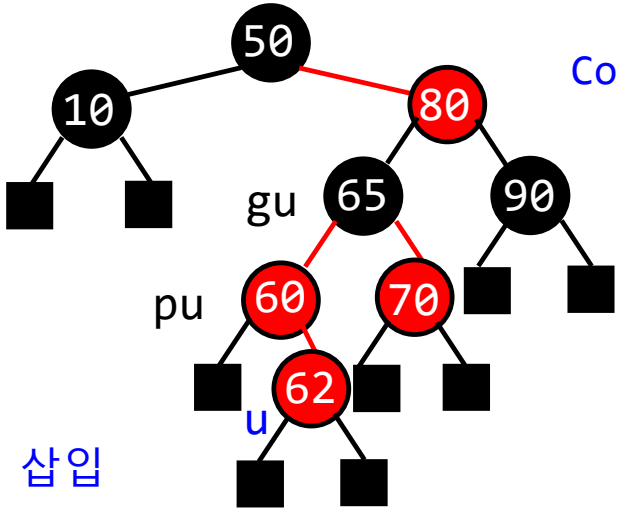


(e) 65 삽입

BAR Rotation

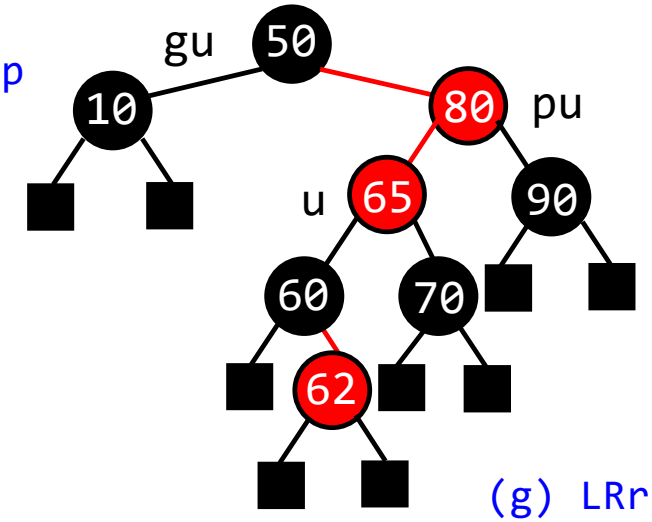


(f) LRb 회전



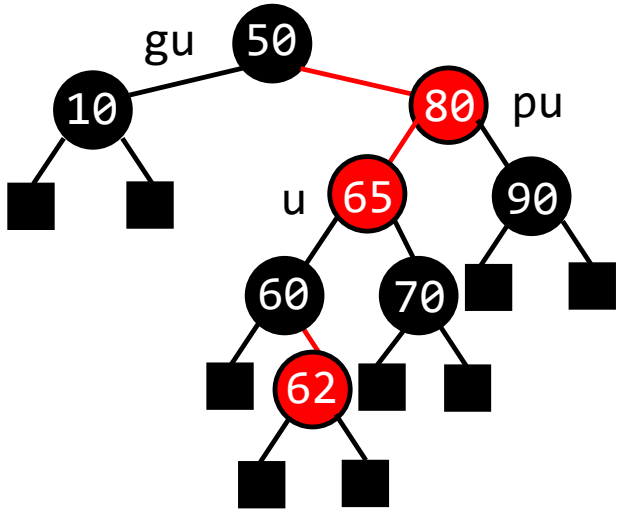
(g) 62 삽입

Color Flip

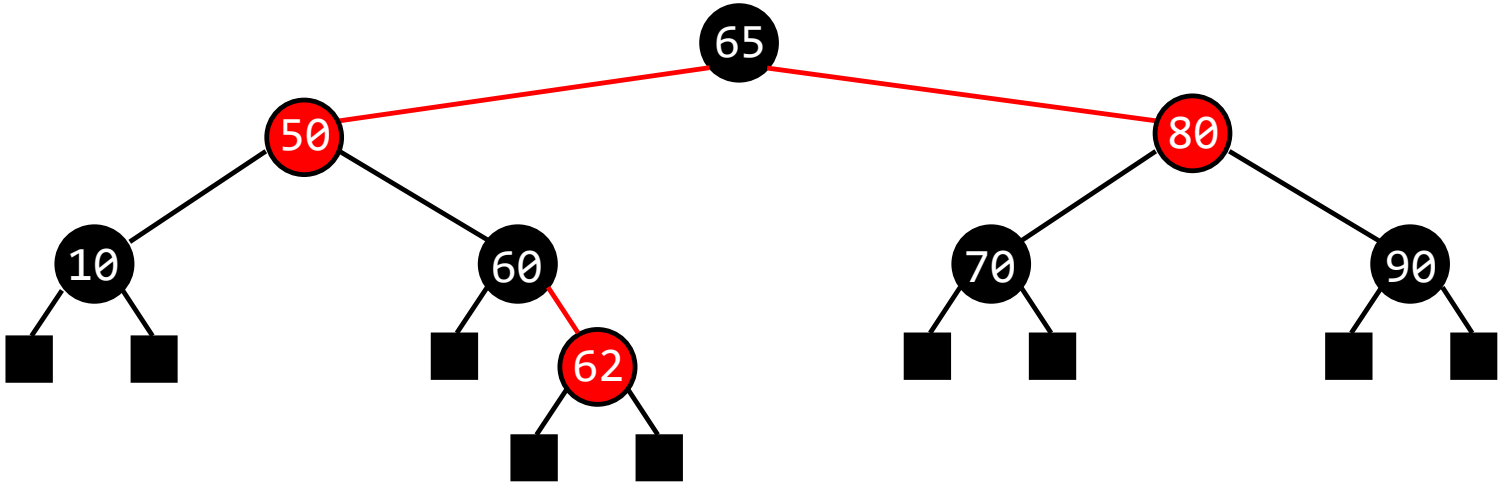


(g) LRr 컬러 변경

(교재)Example - 1

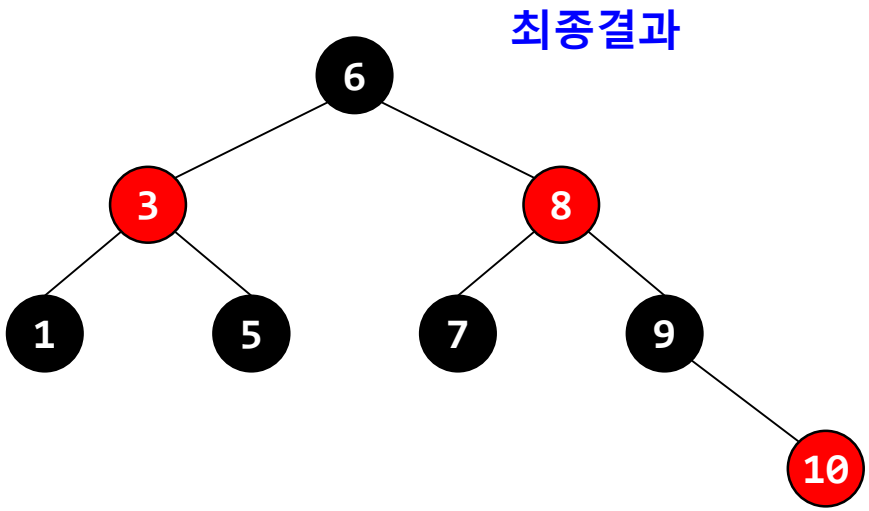


BAR
Rotation



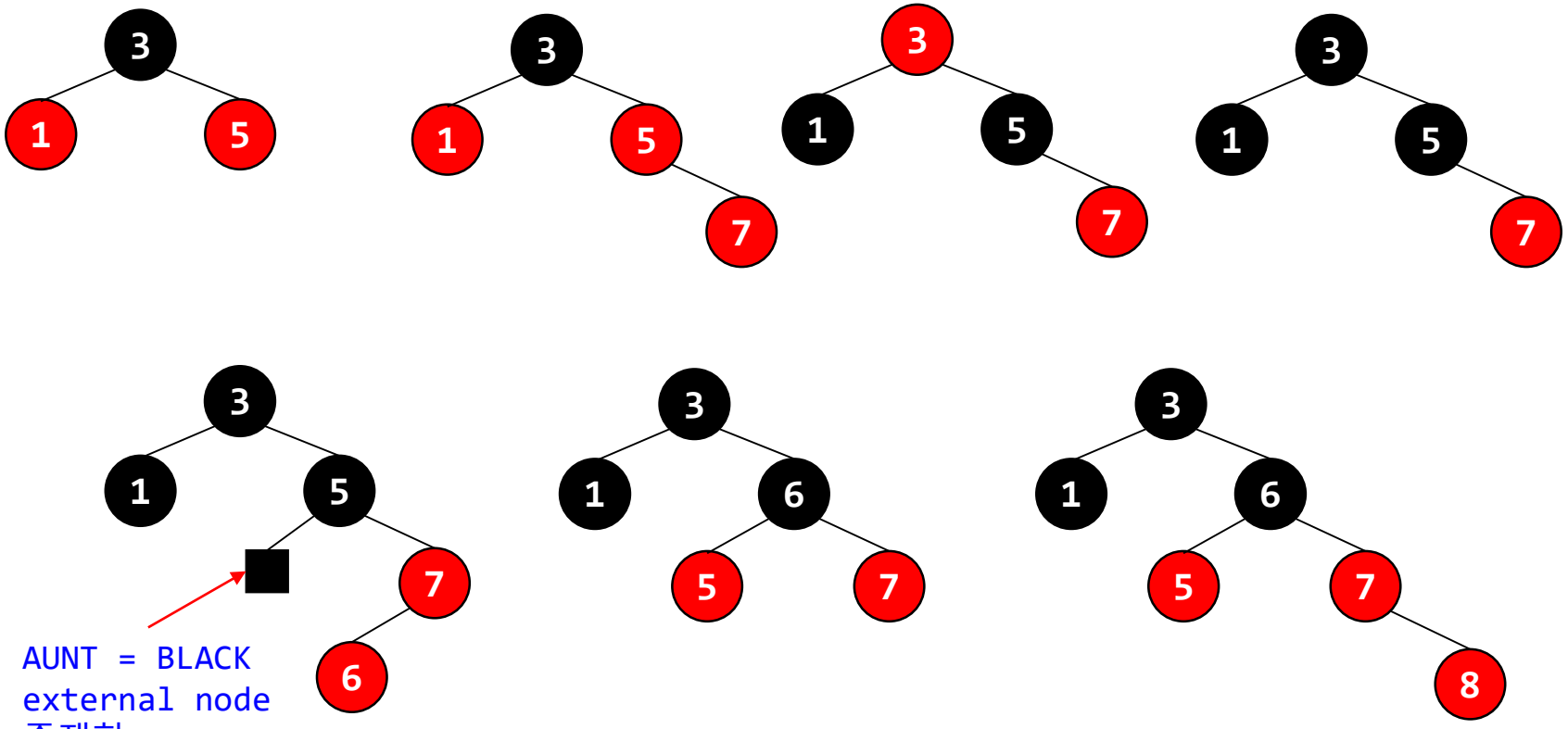
Example - 2

- 3, 1, 5, 7, 6, 8, 9, 10 순으로 삽입될 때 최종 red-black tree 구성 과정은?



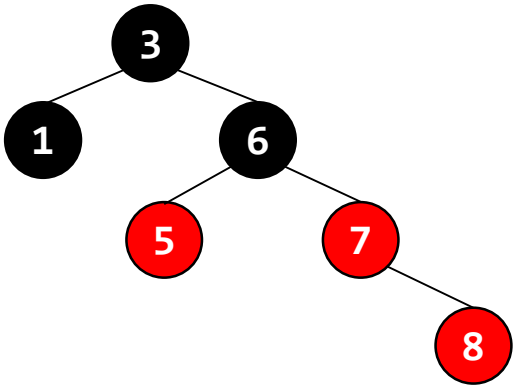
Example - 2

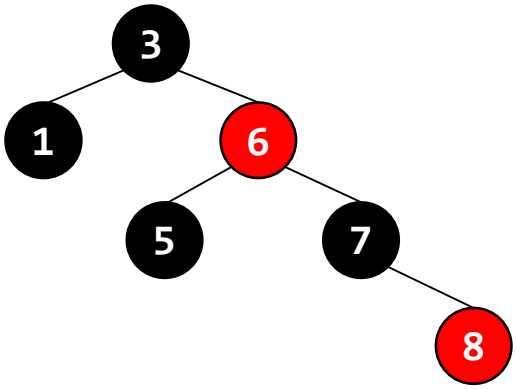
- 3, 1, 5, 7, 6, 8, 9, 10

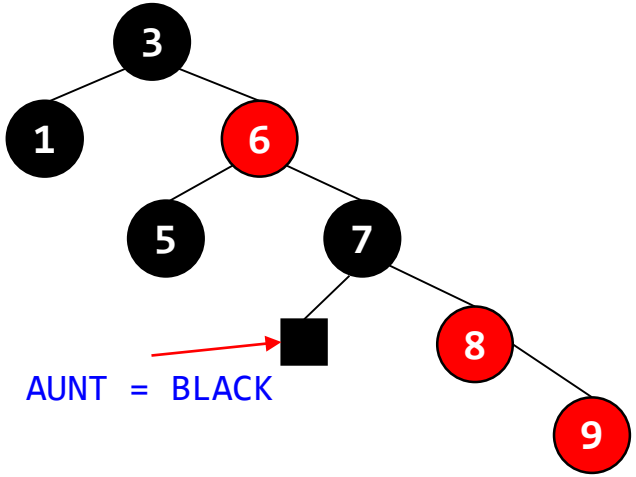


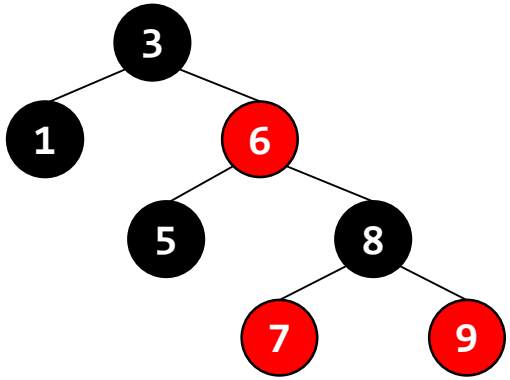
Example - 2

- 3, 1, 5, 7, 6, 8, 9, 10









Example - 2

- 3, 1, 5, 7, 6, 8, 9, 10