제5장 탐색 트리

BST, AVL트리

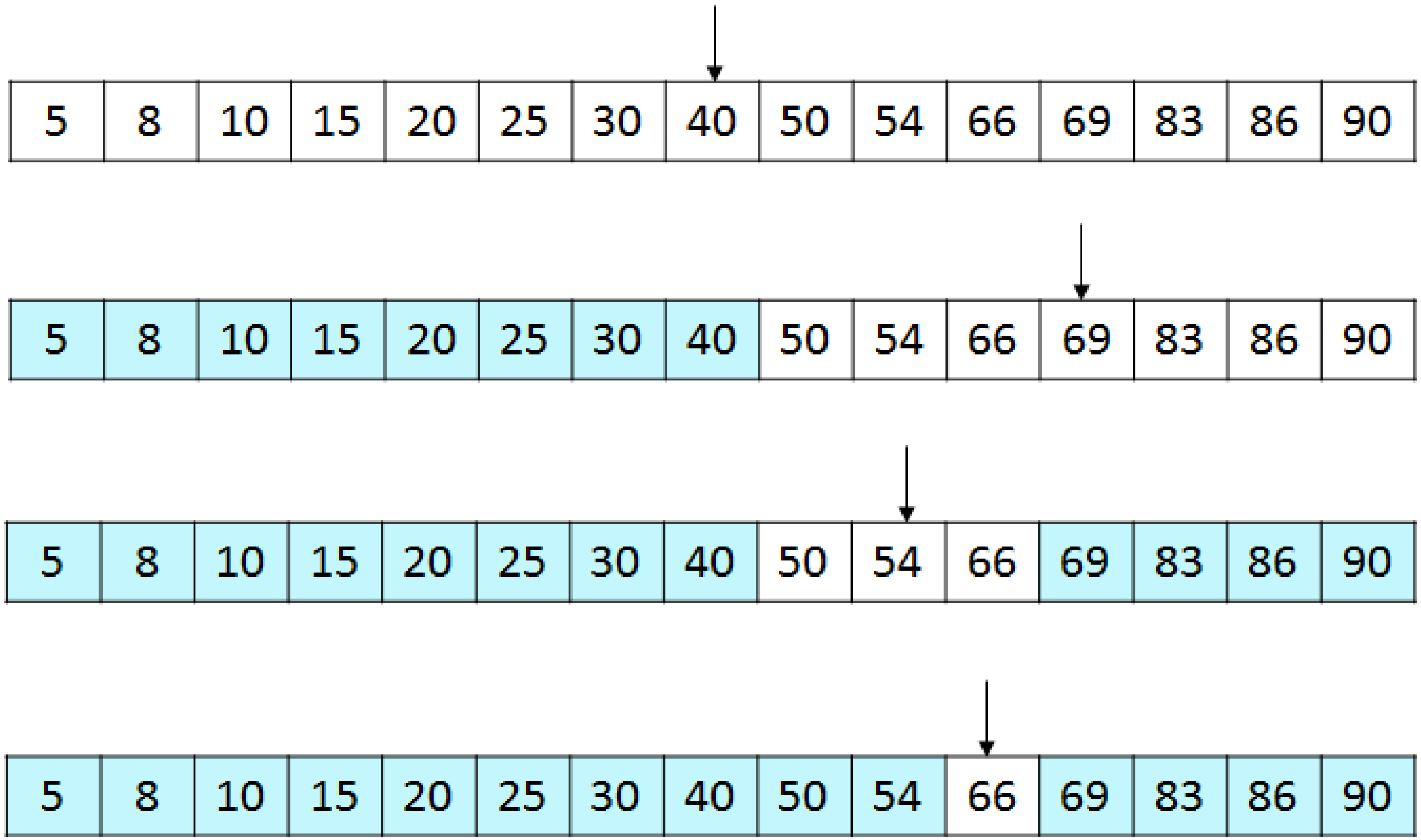
탐색트리

- **저장된 데이터에 대해 탐색, 삽입, 삭제, 갱신 등의 연산을 수행할 수 있는 자료구조**
- **배열이나 연결리스트는 각 연산을 수행하는데 O(N) 시간이 소요**
- **스택이나 큐는 특정 작업에 적합한 자료구조.**
- **5장에서는 리스트 자료구조의 수행시간을 향상시키기 위한 트리 형태의 다양한 사전 자료구조들을 소개**
  - 이진탐색트리, AVL트리, 2-3트리, 레드블랙트리, B-트리

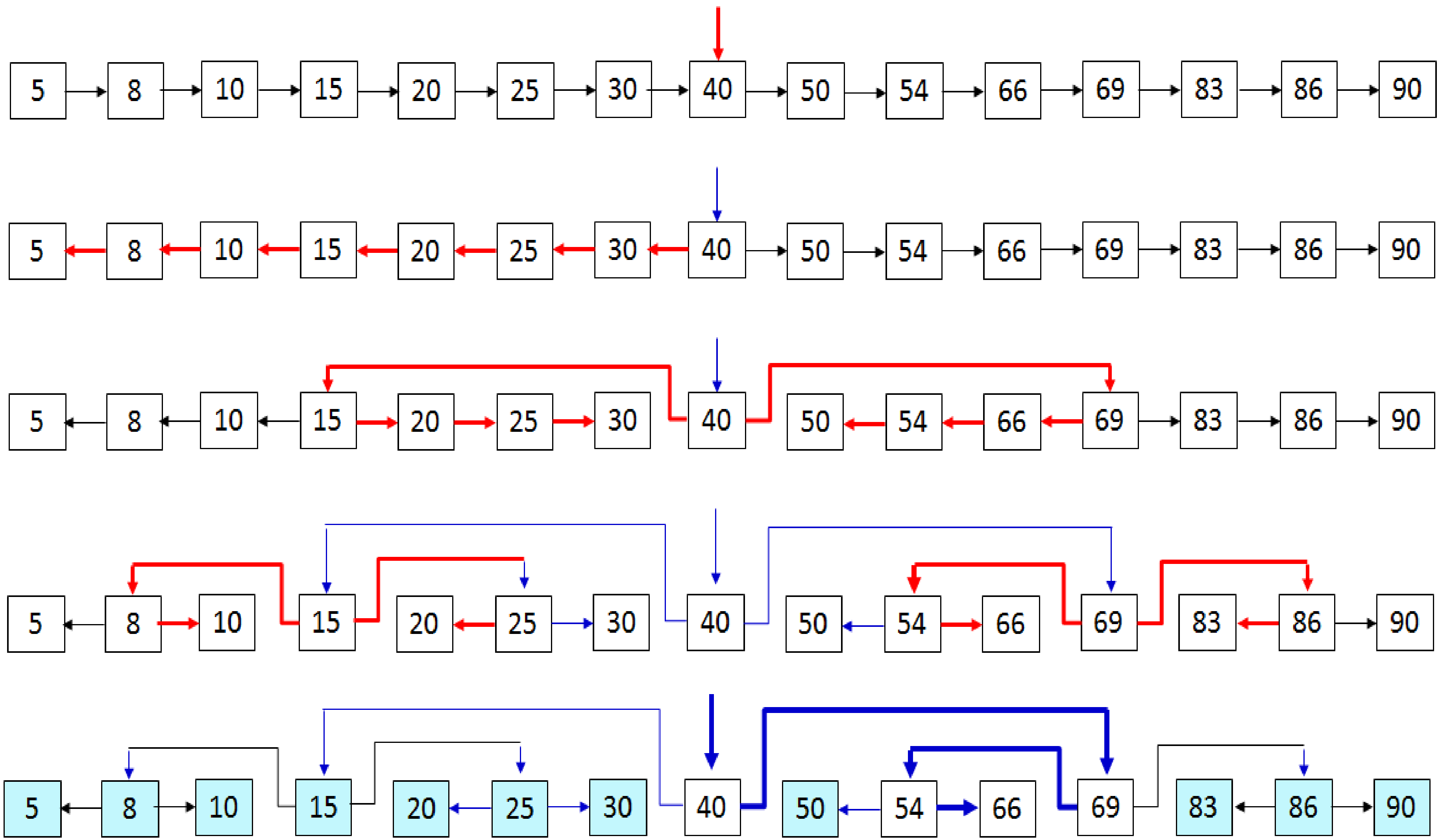
5.1 이진탐색트리

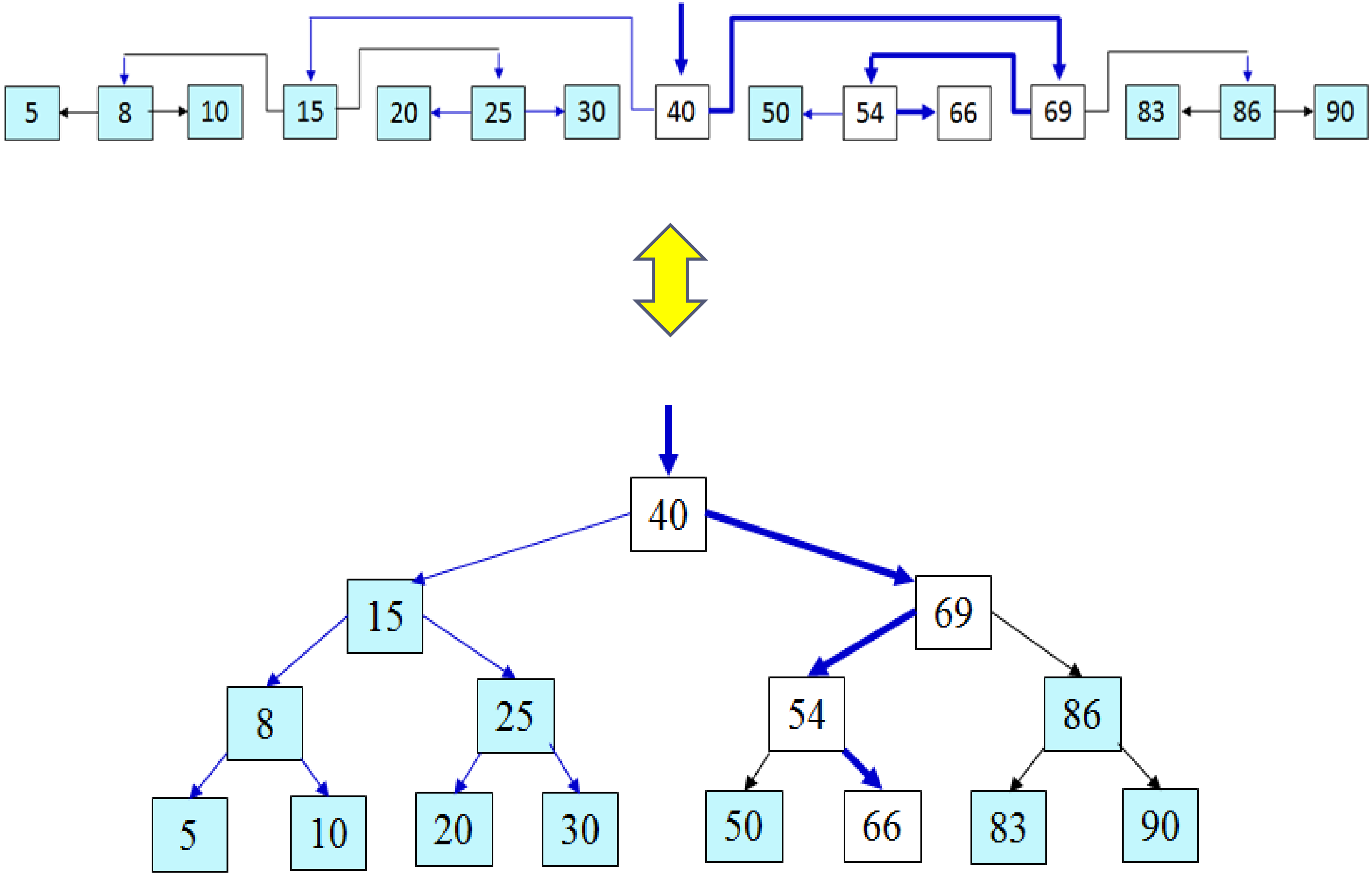
- **이진탐색트리(Binary Search Tree):**
  - 이진탐색(Binary Search)의 개념을 트리 형태의 구조에 접목한 자료구조
- **이진탐색:**
  - 정렬된 데이터의 중간에 위치한 항목을 기준으로 데이터를 두 부분으로 나누어 가며 특정 항목을 찾는 탐색방법

이진탐색으로 66을 찾는 과정

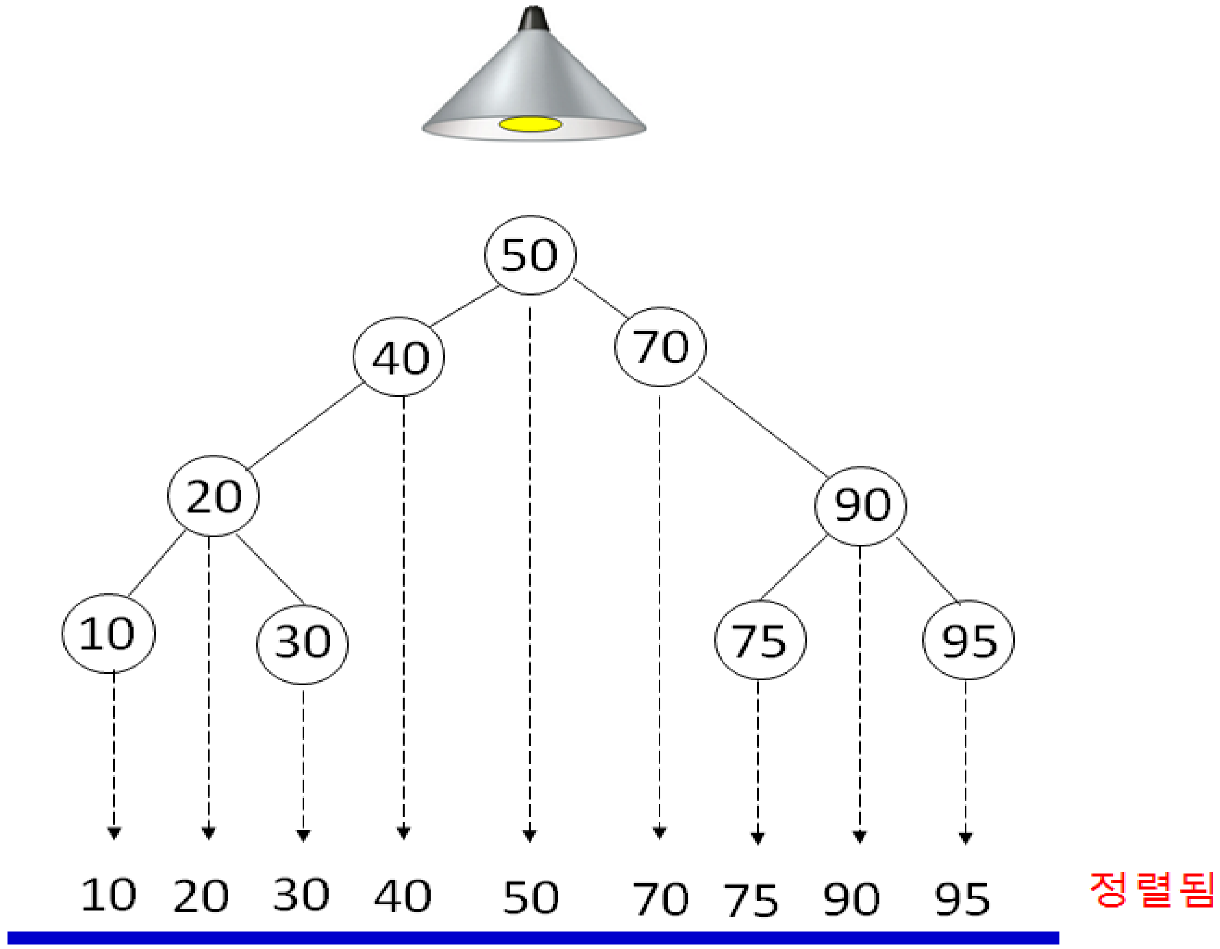


- 트리 형태의 자료구조에서 이진탐색을 수행하기 위해 1차원 배열을 단순연결리스트로 만든 후, 점진적으로 이진트리 형태로 변환해가는 과정

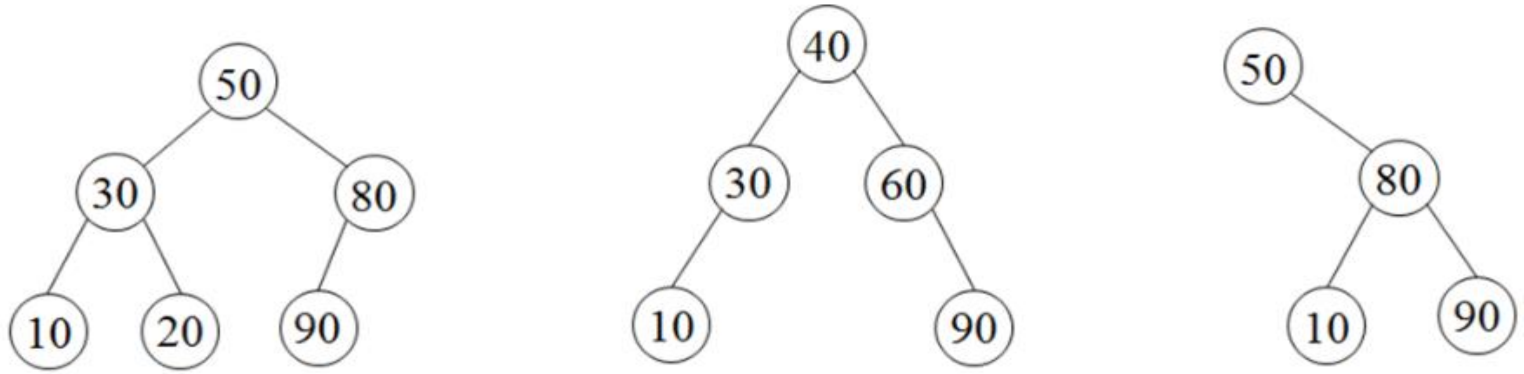



5
8
10
15
20
25
30
40
50
54
66
69
83
86
90

- 이진탐색트리의 특징 중의 하나는 트리를 중위순회 (Inorder Traversal)하면 정렬되어 출력



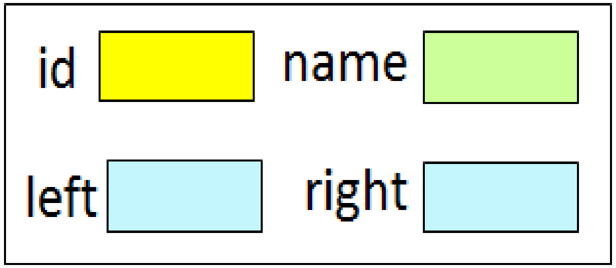
[정의] 이진탐색트리는 이진트리로서 각 노드가 다음의 조건을 만족한다.
- 각 노드 n의 키값이 n의 왼쪽 서브트리 노드들의 키값들보다 크고
- n의 오른쪽 서브트리에 있는 노드들의 키값들보다 작다
이를 이진탐색트리 조건이라 한다.



어느 트리가 이진탐색트리인가?

5.1.1 이진탐색트리 클래스

- 노드(Node) 클래스는 이진트리의 구현에 사용된 노드와 유사
- 노드 객체는 id(키), name(키에 관련된 정보), 왼쪽 자식과 오른쪽 자식을 각각 가리키기 위한 left와 right 필드를 가짐



```java
01 public class Node <Key extends Comparable<Key>, Value> {
02     private Key   id;
03     private Value name;
04     private Node  left, right;
05     public Node(Key newId, Value newName) { // 노드 생성자
06         id   = newId;
07         name = newName;
08         left = right = null;
09     }
10     // get과 set 메소드들
11     public Key getKey()      { return id; }
12     public Value getValue() { return name; }
13     public Node getLeft()   { return left; }
14     public Node getRight()  { return right;}
15     public void setKey(Key      newId)    { id    = newId; }
16     public void setValue(Value newName) { name  = newName; }
17     public void setLeft(Node   newLeft) { left  = newLeft; }
18     public void setRight(Node  newRight){ right = newRight;}
19 }
```

- Line 01: Key와 Value는 generic 타입이고, Key는 비교 연산을 위해 자바의 Comparable 인터페이스를 상속받음
- 키를 비교할 때 Comparable에 선언되어 있는 compareTo() 메소드를 사용하여 비교 연산을 수행
- Line 05~09: Node 클래스의 생성자
- Line 11~18: Node클래스의 get, set 메소드들

이진탐색트리를 위한 BST 클래스

```
01 public class BST<Key extends Comparable<Key>, Value>{
02     public Node root;
03     public Node getRoot() { return root; }
04     public BST(Key newId, Value newName) {  // BST 생성자
05         root = new Node(newId, newName);
06     }
       // get, put, min, deleteMin, delete
       // 메소드들 선언
   }
```

- Line 01: Key와 Value에 대한 부분은 Node 클래스와 동일
- Line 03: root를 리턴하는 getRoot() 메소드
- Line 04~06: BST 클래스의 생성자
- 이진탐색트리의 기본 연산에 대한 메소드들 선언

5.1.2 탐색 연산

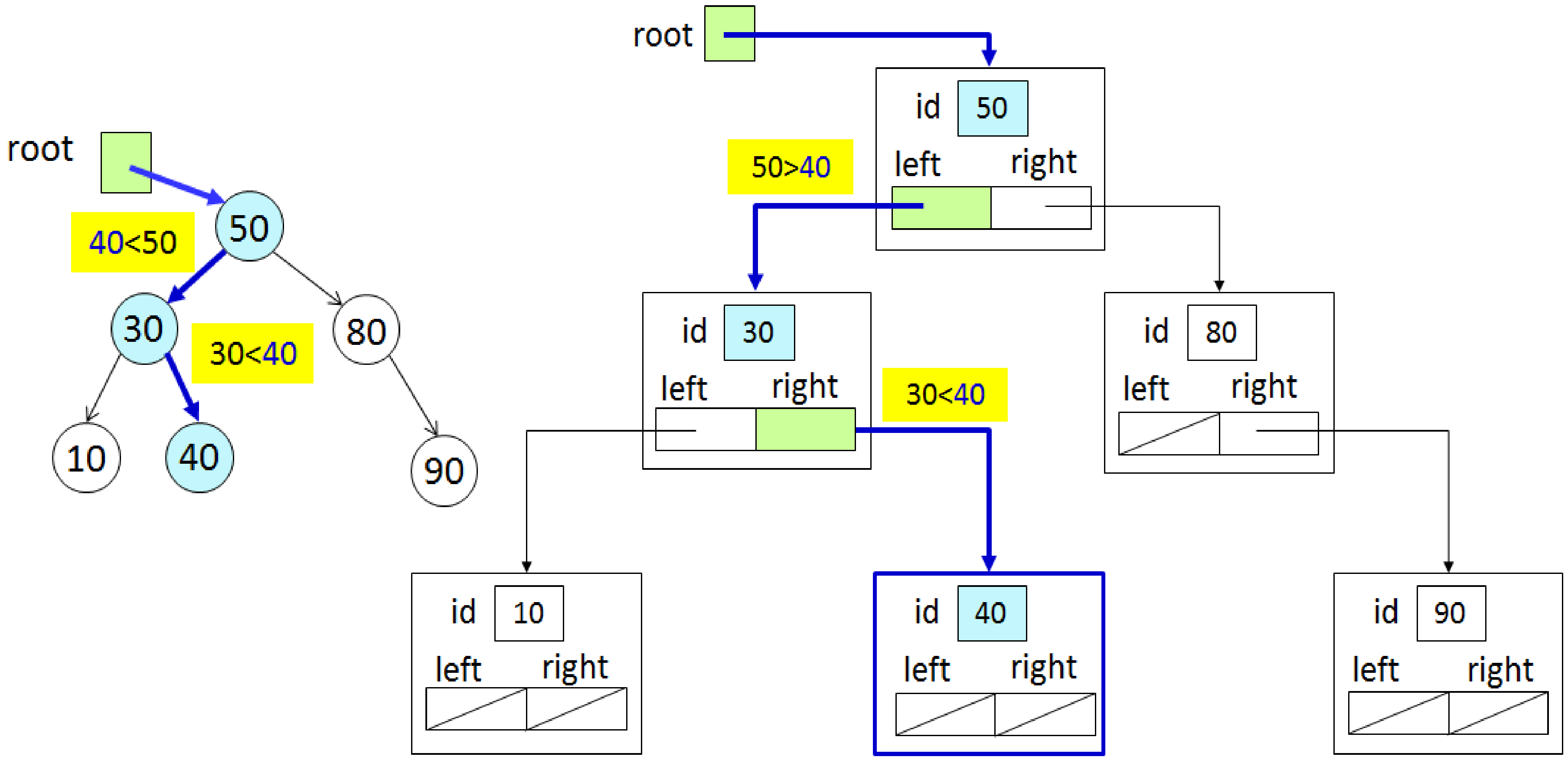
- 탐색하고자 하는 Key가 k라면, 루트노드의 id와 k를 비교하는 것으로 탐색을 시작
- id가 k 보다 작은 경우, 루트의 왼쪽 서브트리에서 k를 찾고, id가 k 보다 큰 경우에는 루트의 오른쪽 서브트리에서 k를 찾으며, id가 k와 같으면 탐색에 성공한 것이므로 해당 노드의 Value, 즉, name을 리턴
- 왼쪽이나 오른쪽 서브트리에서 k를 탐색하는 연산은 루트노드에서의 탐색 연산과 동일

탐색 연산

```
01 public Value get(Key k) { return get(root, k);}
02 public Value get(Node n, Key k) {
03     if (n == null)  return null;    // k를 발견 못함
04     int t = n.getKey().compareTo(k);
05     if (t > 0)      return get(n.getLeft(), k);  // if (k < 노드 n의 id) 왼쪽서브트리 탐색
06     else if (t < 0) return get(n.getRight(), k); // if (k > 노드 n의 id) 오른쪽서브트리 탐색
07     else            return (Value) n.getValue(); // k를 가진 노드 발견
08 }
```

- **get() 메소드는 오버로딩(overloading)으로 2단계로 구현**
  - Line 01: get() 메소드는 1 개의 매개변수인 Key k만을 인자로 갖고
  - Line 02: get() 메소드는 root와 k를 인자로 가짐
  - Line 03: 노드 n이 null인 경우 null을 리턴(탐색 실패)
  - Line 04: k와 노드의 id, 즉, n.getKey()를 비교한 결과에 따라서 line 05 나 06에서 get()을 재귀호출하거나 line 07에서 k를 가진 노드를 리턴

[예제] 40을 탐색하는 과정



C++에서의 탐색 - 반복 구조

```
template <class T, class K>  // Iterative version
T* BST<T, K>::get(const K& k)
{
  TreeNode<T, K> *curNode = root;
  while (curNode)
    if (k < (curNode->data).getKey())
      curNode = curNode->leftChild;
    else if (k > (curNode->data).getKey())
      curNode = curNode->rightChild;
    else return &curNode->data;
  }

  // no matching pair
  return 0;
}
```

```
template <class T, class K> class BST;

template <class T, class K>
class TreeNode {
friend class Tree<T>;
friend class BST<T, K>;
private:
  T data;
  TreeNode<T, K> *leftChild;
  TreeNode<T, K> *rightChild;
};
template <class T, class K>
class BST {
public:
  // Tree operations
   .
private:
  TreeNode<T, K> *root;
};
```

C++에서의 탐색 - 재귀 구조

```
template <class T, class K>  // Driver
T* BST<T, K>::get(const K& k)
{// Search the binary search tree (*this) for a pair with key k.
// If such a pair is found, return a pointer to this pair; otherwise, return 0.
  return get(root, k);
}

template <class T, class K>  // Workhorse
T* BST<T, K>::get(TreeNode<T, K>* p, const K& k)
{
  if (!p) return 0;
  if (k < (p->data).getKey()) return get(p->leftChild, k);
  if (k > (p->data).getKey()) return get(p->rightChild, k);
  return &p->data;
}
```

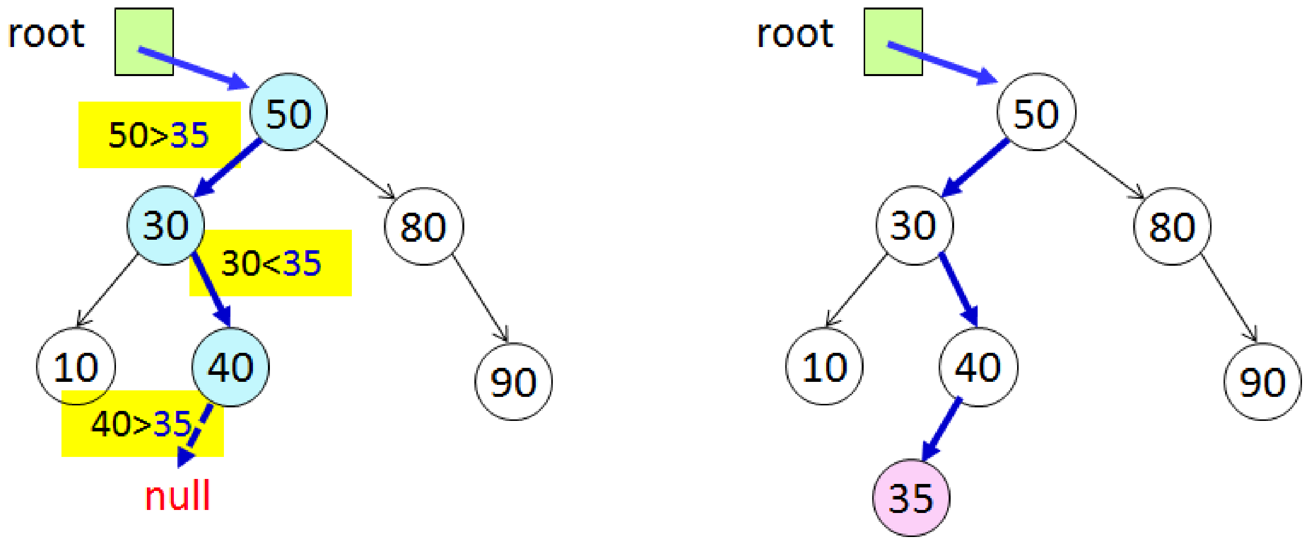
5.1.3 삽입 연산

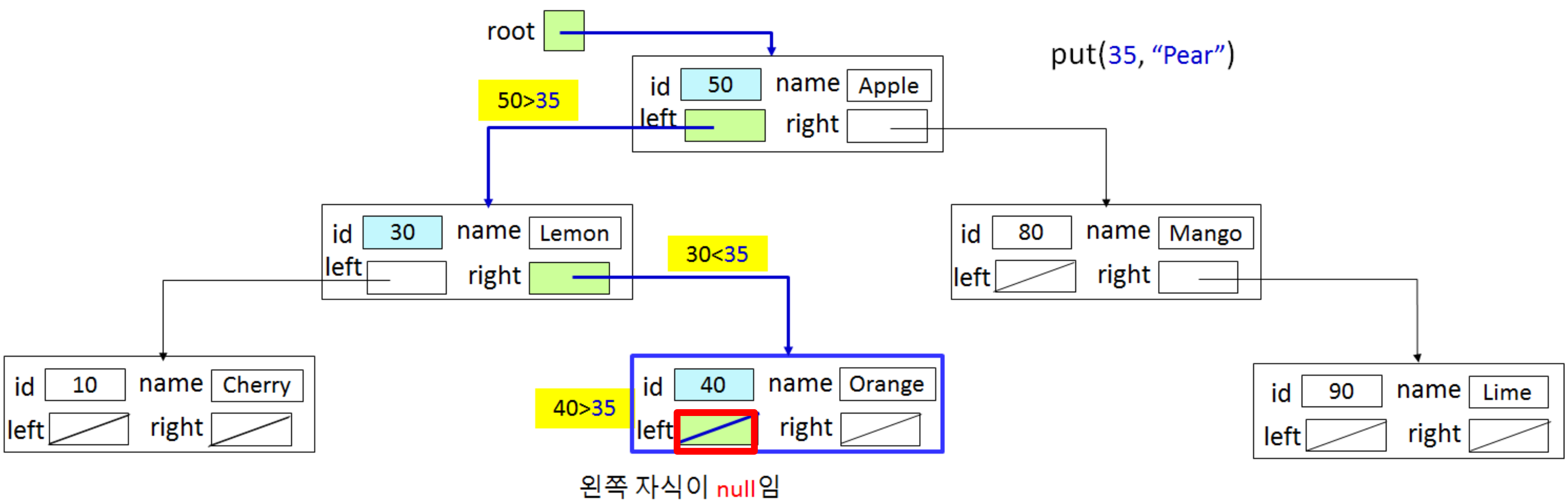
- **탐색 연산의 마지막에서 null이 반환되어야 할 상황에서 null을 반환하는 대신, 삽입하고자 하는 값을 갖는 새로운 노드를 생성하고 이 노드를 부모노드와 연결하면 삽입 연산이 완료**
  - 단, 이미 트리에 존재하는 id를 삽입한 경우, name을 갱신

```
01  public void put(Key k, Value v) { root = put(root, k, v);}
02  public Node put(Node n, Key k, Value v){
03      if (n == null)  return new Node(k, v);
04      int t = n.getKey().compareTo(k);
05      if (t > 0)      n.setLeft(put(n.getLeft(), k, v));   // if (k < 노드 n의 id) 왼쪽서브트리에 삽입
06      else if (t < 0) n.setRight(put(n.getRight(), k, v)); // if (k > 노드 n의 id) 오른쪽서브트리에 삽입
07      else n.setValue(v); // 노드 n의 name을 v로 갱신
08      return n;
09  }
```

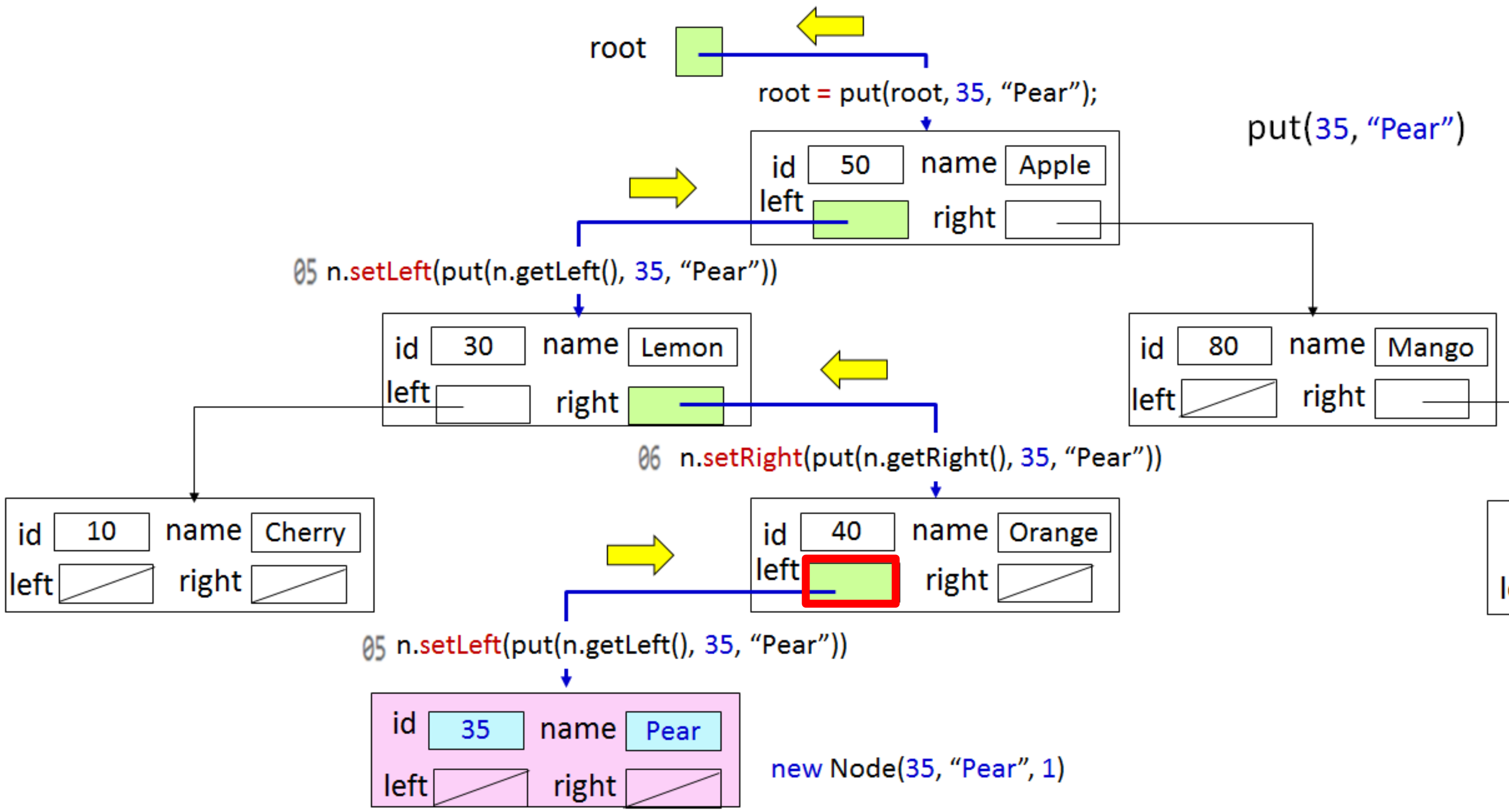
  - Line 01: 두 개의 매개변수 Key k와 Value v를 가지며, line 02의 put() 메소드를 호출
  - [주의] line 01에서 root가 put() 메소드가 리턴하는 Node를 가리키게 하는 것
  - Line 05, 06: setLeft()나 setRight()를 호출하여 put() 메소드가 리턴하는 Node를 연결

[예제] 35를 삽입하는 과정





35를 삽입할 장소를 탐색하는 과정



새 노드 삽입 후 루트노드로 거슬러 올라가며 재 연결하는 과정

C++에서의 insert - 반복 구조

```
template <class T, class K>
void BST<T,K>::insert(const T item, const K key)
{// Insert item with key into the binary search tree.
// search for key, parNode is par of curNode
   TreeNode<K,E> *curNode = root, *parNode = nullptr ;
   while (curNode) {
      parNode = curNode ;
      if (key < (curNode->data).getKey()) curNode = curNode->leftChild;
      else if (key > (curNode->data).getKey()) curNode = curNode->rightChild;
      else //  duplicate, update associated element
        { curNode->data = item ; return;}
   }

   // perform insertion
   curNode = new TreeNode<T, K> (item);
   if (root)  // tree not empty
      if (key < (parNode->data).getKey()) parNode->leftChild = curNode;
      else parNode->rightChild = curNode;
   else root = curNode ;
}
```

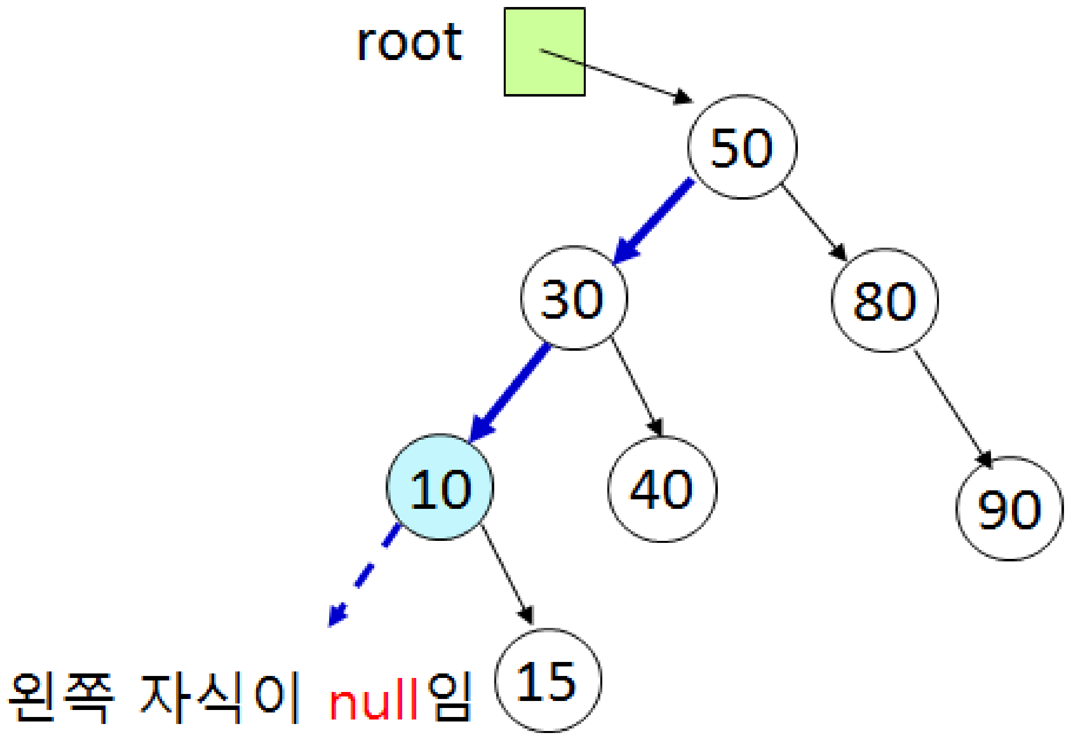
5.1.4 최솟값 찾기

- **최솟값은 루트노드로부터 왼쪽 자식노드를 따라 내려가며, null을 만났을 때 null 의 부모노드가 가진 id**
  - min() 메소드는 delete() 메소드에서 사용

```
1     public Key min() {
2         if (root == null) return null;
3         return (Key) min(root).getKey();}
4     private Node min(Node n) {
5         if (n.getLeft() == null)  return n;
6         return min(n.getLeft());
7     }
```

  - Line 01: min() 메소드와 line 05의min() 메소드로 구성
  - Line 05: min() 메소드는 인자로 전달받은 노드가 null이 아닌 한 계속 왼쪽 자식을 인자로 넘겨 min 메소드를 재귀호출하며(line 07),
  - 왼쪽 자식이 null이면 왼쪽 자식이 null인 노드(최솟값을 가진 노드)의 레퍼런스를 리턴(line 06).
  - Line 03: 리턴된 레퍼런스의 getKey()로 가져온 id를 최솟값으로 리턴

[예제]



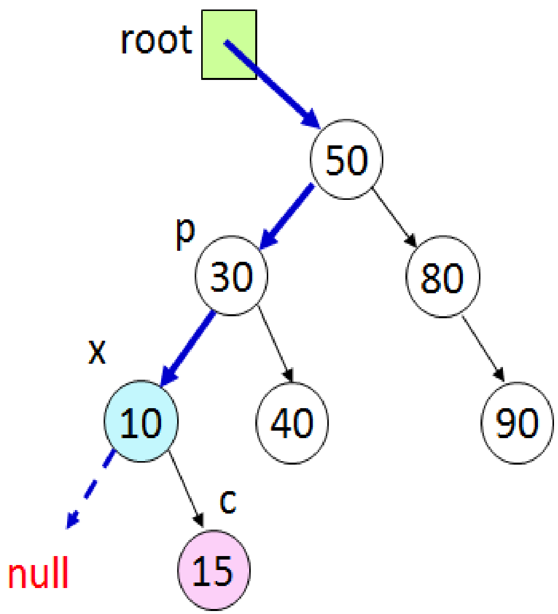
5.1.5 최솟값 삭제 연산

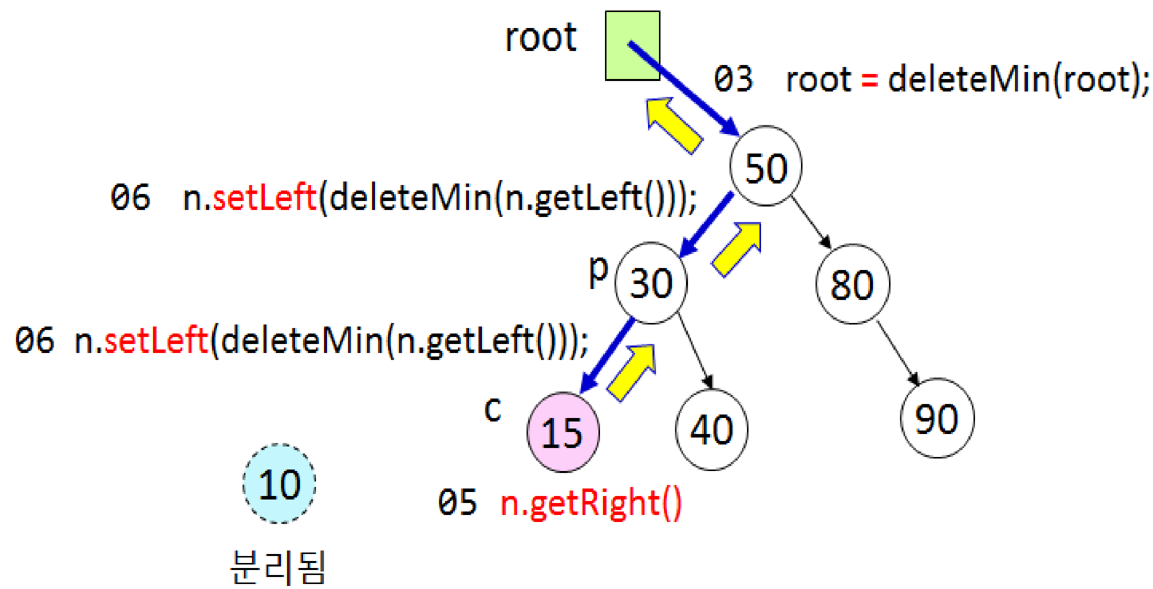
- **최솟값을 가진 노드를 삭제하는 것은 최솟값을 가진 노드 x를 찾아낸 뒤, x의 부모노드 p와 x의 오른쪽 자식노드 c를 연결**
  - 이 때 c 가 null이더라도 자식으로 연결
  - deleteMin() 메소드는 임의의 id를 가진 노드를 삭제하는 delete() 메소드에서 사용

```
01 public void deleteMin() {
02     if (root == null) System.out.println("empty 트리");
03     root = deleteMin(root); }
04 public Node deleteMin(Node n) {
05     if (n.getLeft() == null) return n.getRight(); // if (n의 왼쪽 자식==null) n의 오른쪽자식 리턴
06     n.setLeft(deleteMin(n.getLeft()));            // if (n의 왼쪽 자식≠null)  n의 왼쪽자식으로 재귀호출
07     return n;
08 }
```

  - 만일 트리가 empty라면 에러 메시지를 출력하고(line 02), 트리가 empty가 아닌 경우, line 04의 deleteMin() 메소드를 root를 인자로 하여 line 03에서 호출
  - 이후 루트노드로부터 왼쪽 자식으로 계속 내려가다가(line 06),
  - 왼쪽 자식이 null이면 현재 노드의 오른쪽 자식의 레퍼런스를 리턴 (line 05).
  - 그 후부터는 루트노드까지 거슬러 올라가며 부모와 자식을 line 06에서 다시 연결

[예제]


root
50
p
30
80
x
10
40
90
c
null
15

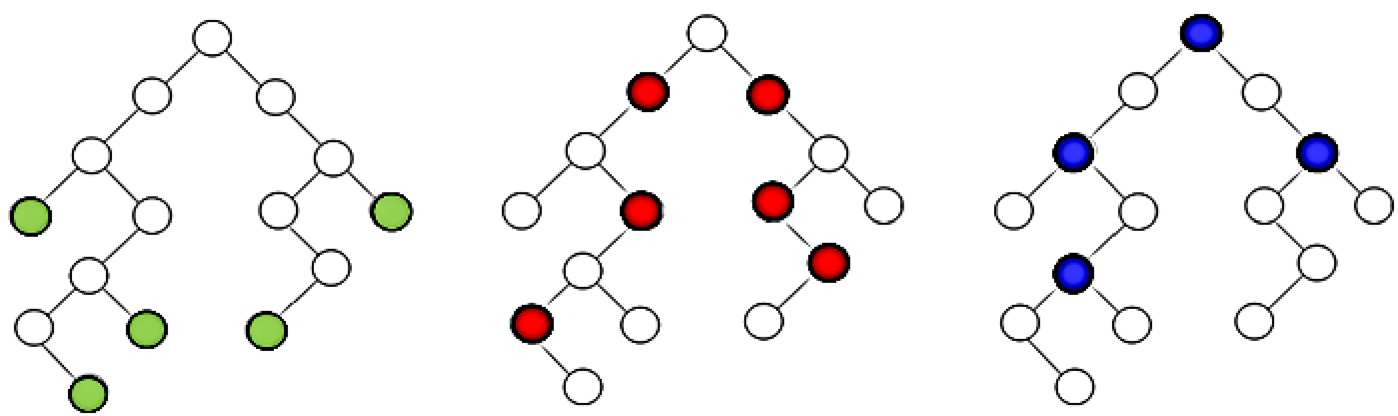

root
03 root = deleteMin(root);
50
06 n.setLeft(deleteMin(n.getLeft()));
p
30
80
06 n.setLeft(deleteMin(n.getLeft()));
c
15
40
90
10
분리됨
05 n.getRight()

5.1.6 삭제 연산

- 우선 삭제하고자 하는 노드를 찾은 후 이진탐색트리 조건을 만족하도록 삭제된 노드의 부모노드와 자식노드들을 연결해 주어야
- 삭제되는 노드가 자식이 없는 경우(case 0),
  자식이 하나인 경우(case 1),
  자식이 둘인 경우(case 2)로 나누어 delete연산을 수행

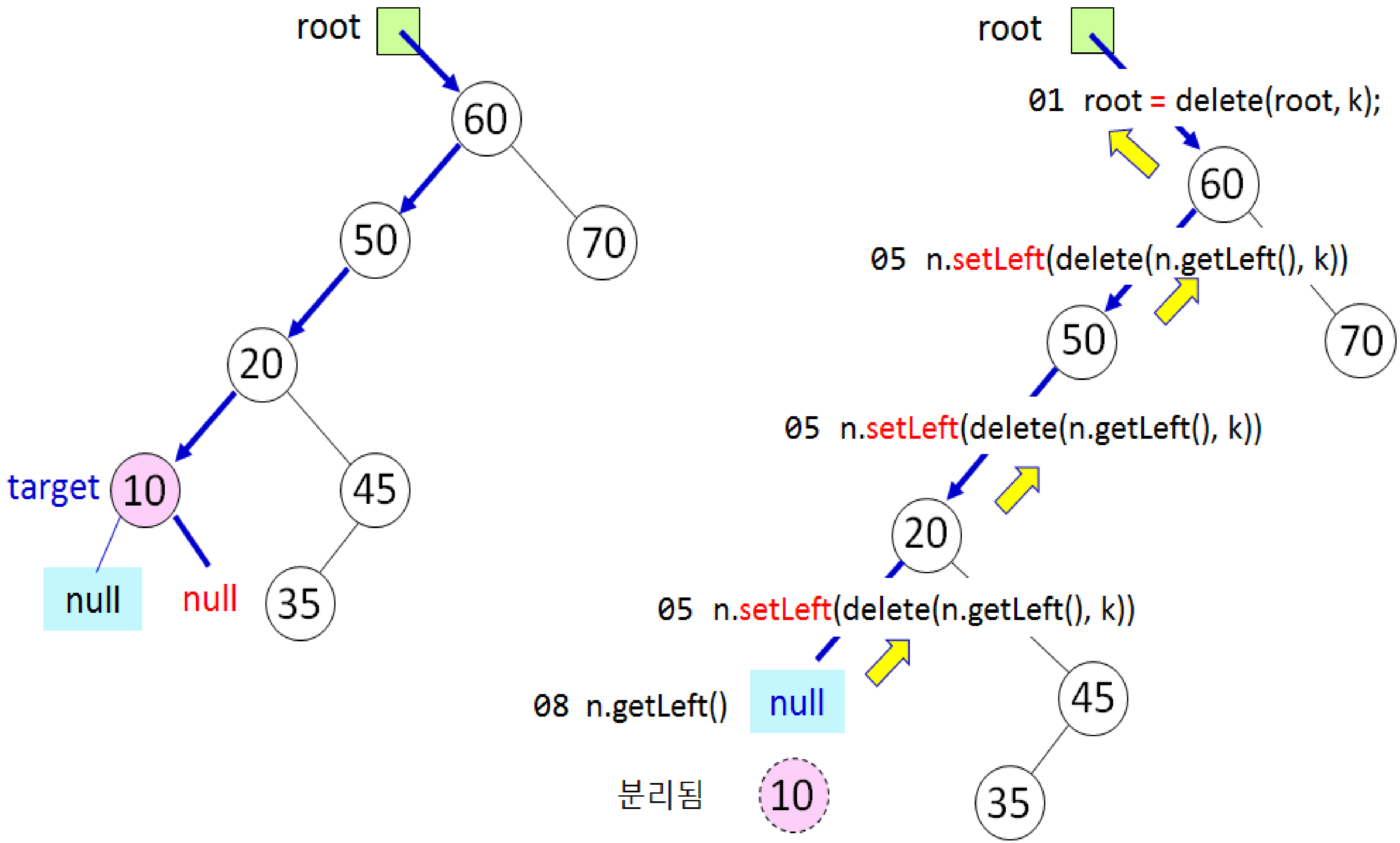
삭제 연산

- Case 0: 삭제해야 할 노드 x의 부모노드가 x를 가리키던 레퍼런스를 null로 만든다.
- Case 1: x가 한쪽 자식인 c만 가지고 있다면, x의 부모노드와 x의 자식노드 c를 직접 연결
- Case 2: x의 부모노드는 하나인데 x의 자식노드가 둘이므로 x의 자리에 중위순회하면서 x를 방문하기 직전 노드(Inorder Predecessor, 중위 선행자) 또는 직후에 방문되는 노드(Inorder Successor, 중위 후속자)를 이동

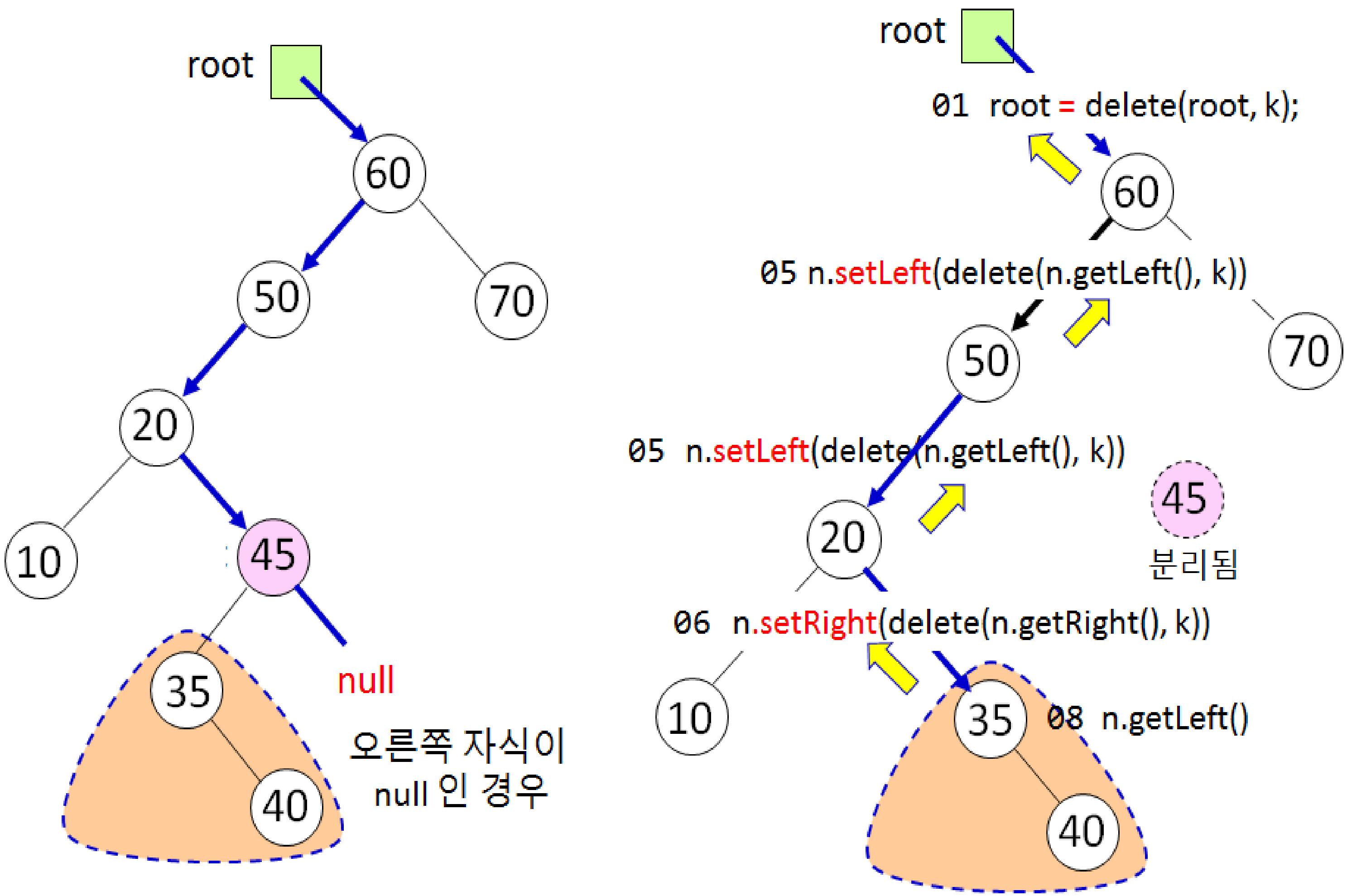


case 0 case 1 case 2

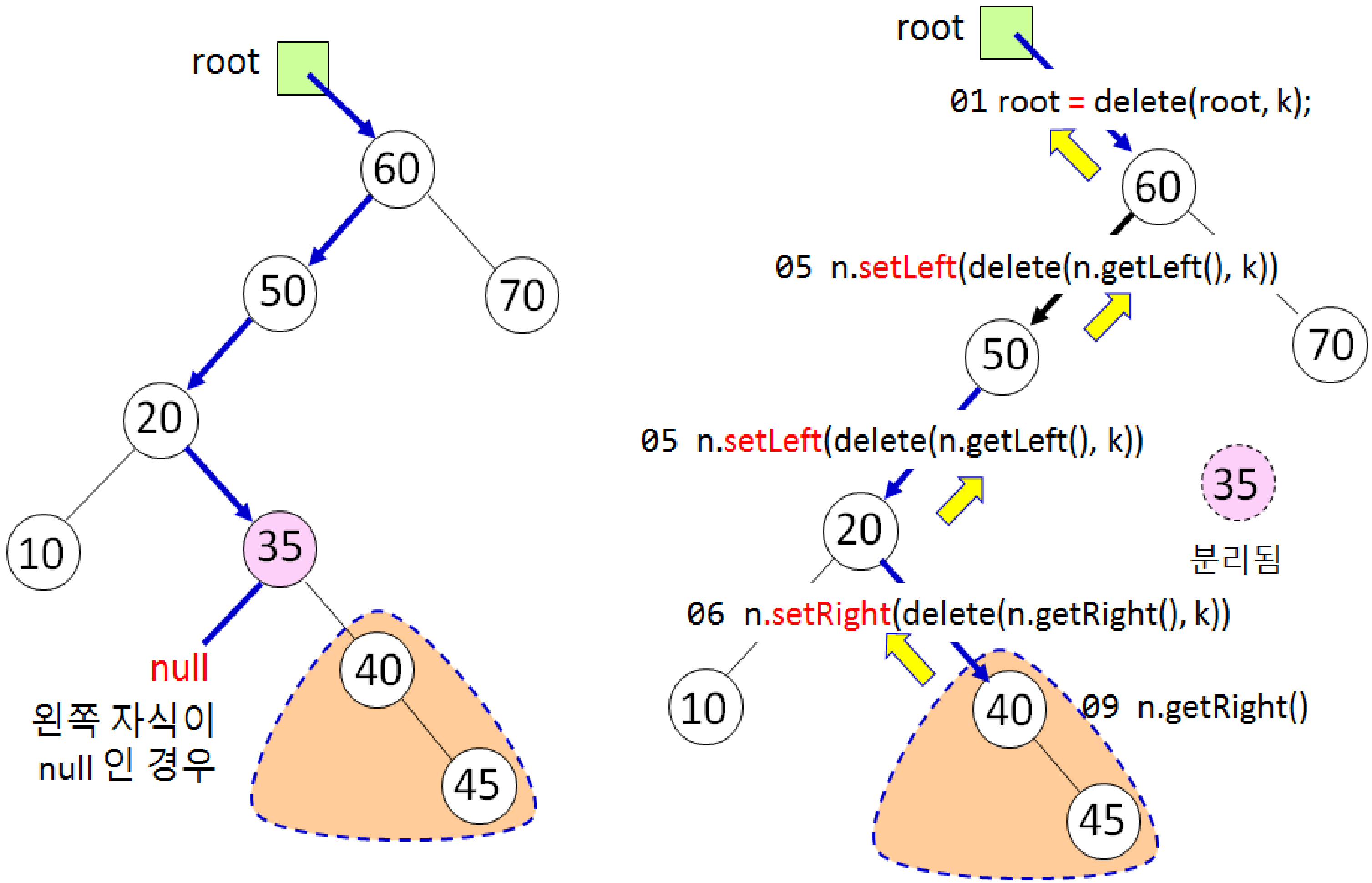
[예제 1] delete(10)이 수행되는 과정 (case 0)



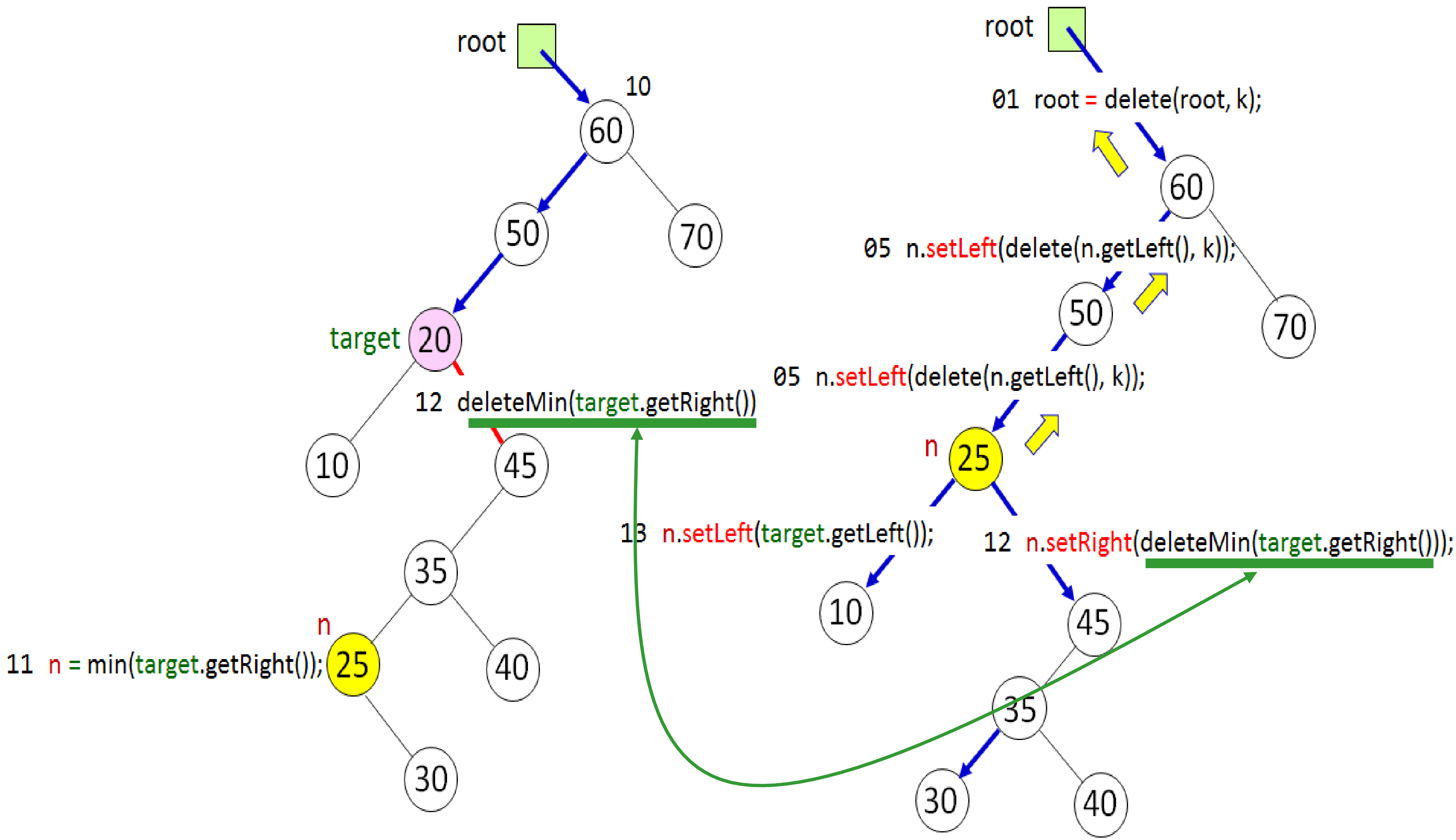
[예제 2] delete(45)가 수행되는 과정 (case 1, line 08)



[예제 3] delete(35)가 수행되는 과정 (case 1, line 09)



[예제 4] delete(20)이 수행되는 과정 (case 2)



```java
01 public void delete(Key k) { root = delete(root, k);}
02 public Node delete (Node n, Key k) {
03     if (n == null) return null;
04     int t = n.getKey().compareTo(k);
05     if (t > 0)       n.setLeft(delete(n.getLeft(), k));    // if (k < 노드 n의 id)  왼쪽 자식으로 이동
06     else if (t < 0) n.setRight(delete(n.getRight(), k)); // if (k > 노드 n의 id) 오른쪽 자식으로 이동
07     else {  // 삭제할 노드 발견
08         if (n.getRight() == null)  return n.getLeft();   // case 0, 1
09         if (n.getLeft()  == null)  return n.getRight(); // case 1
10         Node target = n;    // case 2 Line10-13
11         n = min(target.getRight()); // 삭제할 노드 자리로 옮겨올 노드 찾아서 n이 가리키게 함
12         n.setRight(deleteMin(target.getRight()));
13         n.setLeft(target.getLeft());
14     }
15     return n;
16 }
```

- **delete() 메소드에서 삭제할 노드를 탐색하는 과정은 line 05~06에서 재귀적으로 수행**
  - Line 08: 삭제되는 노드의 오른쪽 자식이 null인 경우를 처리하는데,
    - case 0: 두 자식 모두가 null이므로 line 08의 조건에 해당
    - case 1: 자식이 왼쪽 또는 오른쪽 둘 중에 한 개만 있으므로 오른쪽 자식이 없는 경우 역시 line 08의 조건에 해당
  - case 1에서 왼쪽 자식이 없는 경우는 line 09에서 처리

삭제 연산

- Case 2의 경우 삭제될 노드의 자리로 옮겨올 노드 n의 중위 후속자를min() 메소드를 호출하여 찾음(line 11).
- Line 12: deleteMin() 메소드를 호출하여 노드 n을 트리에서 분리시키고, n 의 부모와 n의 자식을 연결시킨 뒤, 계속해서 재 연결하며 거슬러 올라가며 최종적으로 삭제되는노드(target)의 오른쪽 자식노드의 레퍼런스를 리턴
- 리턴된 레퍼런스는 n.setRight()에 의해 n의 오른쪽 자식으로 연결 (line 12).
- Line 13: target의 왼쪽 자식을n.setLeft()를 이용해 노드n의 왼쪽 자식으로 만든다.

수행시간

- 이진탐색트리에서 탐색, 삽입, 삭제 연산은 공통적으로 루트노드에서 탐색을 시작하여 최악의 경우에 이파리노드까지 내려가고, 삽입과 삭제 연산은 다시 루트노드까지 거슬러 올라가야 함
- 트리를 1 층 내려갈 때는 재귀호출이 발생하고, 1 층을 올라갈 때는 setLeft() 또는 setRight() 메소드가 수행되는데, 이들 각각은 O(1) 시간 소요
- 연산들의 수행시간은 각각 트리의 높이(h)에 비례, O(h)

수행시간

- N개의 노드가 있는 이진탐색트리의 높이가 가장 낮은 경우는 완전이진트리 형태일 때이고, 가장 높은 경우는 편향이진트리
- 따라서 이진트리의 높이 h는 아래와 같이 표현

$$\lceil \log (N+1) \rceil \approx \log N \le h \le N$$

- Empty 이진탐색트리에 랜덤하게 선택된 N개의 키를 삽입한다고 가정했을 때, 트리의 높이는 약 1.39 log N

5.2 AVL트리

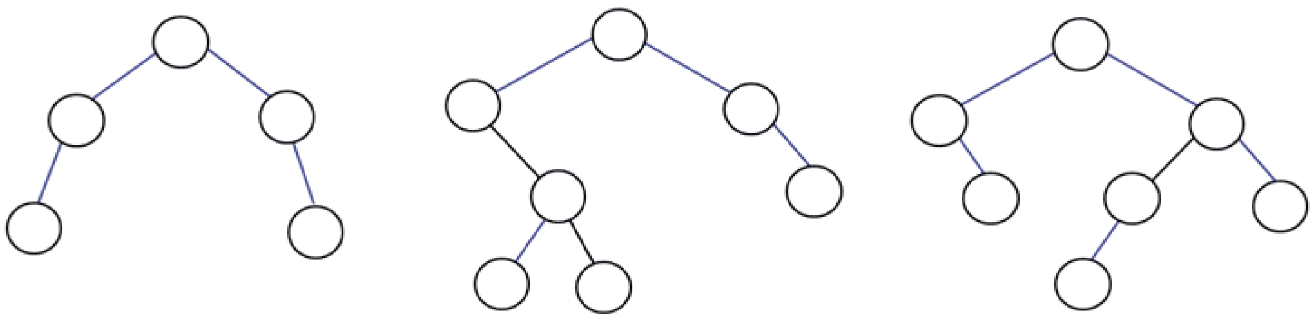
- AVL 트리는 트리가 한쪽으로 치우쳐 자라나는 현상을 방지하여 트리 높이의 균형(Balance)을 유지하는 이진탐색트리
- 균형(Balanced) 이진트리를 만들면 N개의 노드를 가진 트리의 높이가 O(logN)이 되어 탐색, 삽입, 삭제 연산 수행시간이 O(logN)으로 보장

[핵심 아이디어]

AVL트리는 삽입이나 삭제로 인해 균형이 깨지면 회전 연산을 통해 트리의 균형을 유지한다.

AVL트리

[정의] AVL트리는 임의의 노드 x에 대해 x의 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 차이가 1을 넘지 않는 이진탐색트리이다.



(a) (b) (c)

어느 트리가 AVL트리 형태를 갖추고 있나?

AVL트리

[정리] N개의 노드를 가진 AVL트리의 높이는 O(logN)이다.

- A(h)를 높이가 h인 AVL트리를 구성하는 최소의 노드 수로 정의하고 AVL트리의 왼쪽자식과 오른쪽자식이 AVL트리인 재귀적인 특성을 이용하여 A(h)와 피보나치 수열의 관련성을 통해 증명 가능

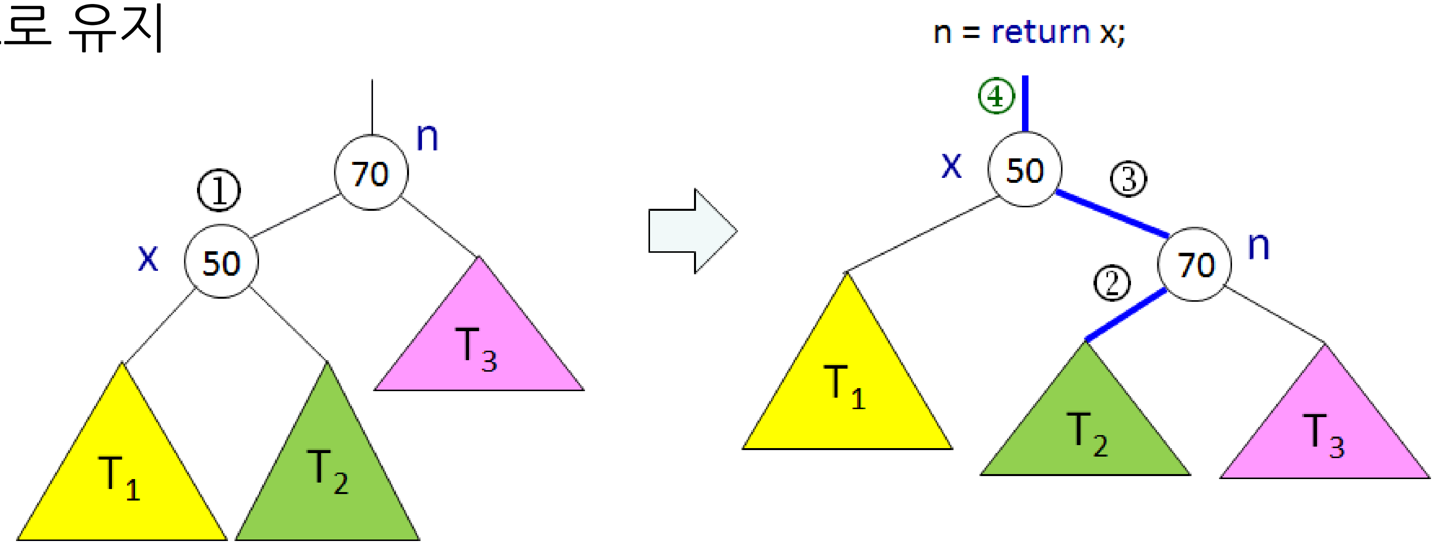
5.2.1 AVL트리의 회전연산

- **AVL트리에서 삽입 또는 삭제 연산을 수행할 때 트리의 균형을 유지하기 위해 LL-회전, RR-회전, LR-회전, RL-회전연산 사용**
- **회전연산은 2 개의 기본적인 연산으로 구현**
- **rotateRight(): 왼쪽 방향의 서브트리가 높아서 불균형이 발생할 때 서브트리를 오른쪽 방향으로 회전하기 위한 메소드**
  - 노드 n의 왼쪽 자식노드 x를 노드 n의 자리로 옮기고,
    노드 n을 노드 x의 오른쪽 자식노드로 만들며,
    이 과정에서 서브트리 T2가 노드 n의 왼쪽 서브트리로 이동

AVL트리의 회전연산

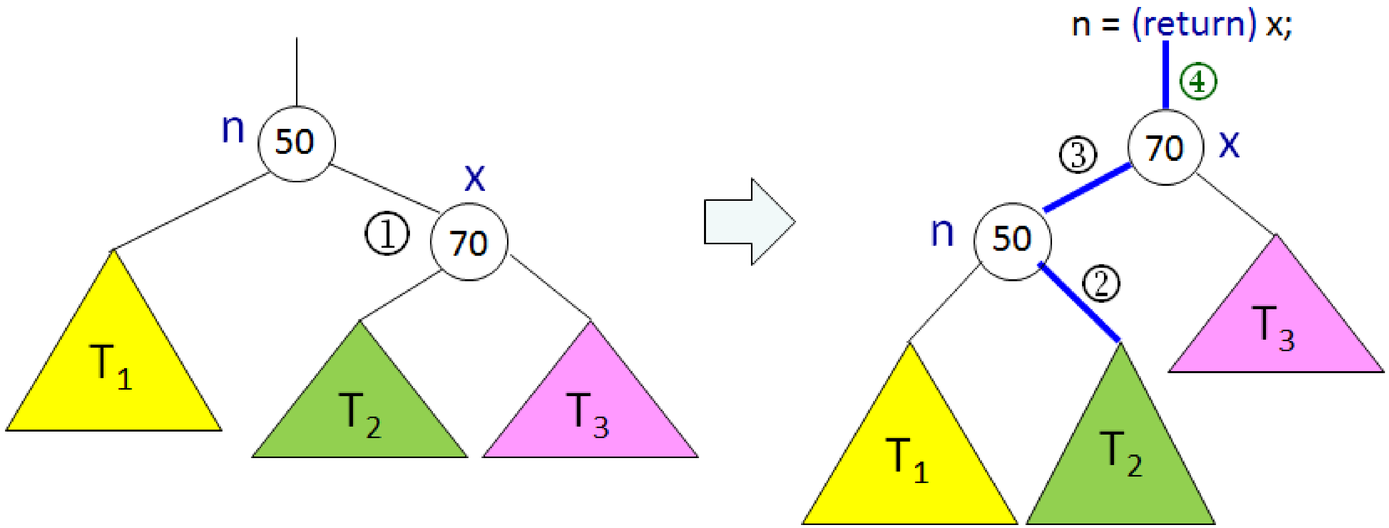
```
01 private Node rotateRight(Node n) {
02      Node x = n.left; ①
03      n.left = x.right; ②
04      x.right = n;      ③
05      n.height = tallerHeight(height(n.left), height(n.right)) + 1; // 높이 갱신
06      x.height = tallerHeight(height(x.left), height(x.right)) + 1; // 높이 갱신
07      return x; ④  // 회전 후 x가 n의 이전 자리로 이동되었으므로
08 }
```

- Line 02 ~ 04와 07에 각각 부여된 번호 순서에 따라 [그림 5-21]의 link 변경
- 마지막으로 line 07에서 노드 x의 레퍼런스를 리턴
- [주목해야 할 것] 서브트리들의 위치가 좌에서 우로 봤을 때, 항상 T1, T2, T3 순으로 유지



AVL트리의 회전연산

- **rotateLeft()는 오른쪽 방향의 서브트리가 높아서 불균형이 발생했을 때 왼쪽 방향으로 회전하기 위한 메소드**
  - 노드 n의 오른쪽 자식노드 x를 노드 n의 자리로 옮기고,
    노드 n을 노드 x의 왼쪽 자식노드로 만들며,
    이 과정에서 서브트리 T2가 노드 n의 오른쪽 서브트리로 이동



(a) 수행 전 (b) 회전 후

[그림 5-22] rotateLeft

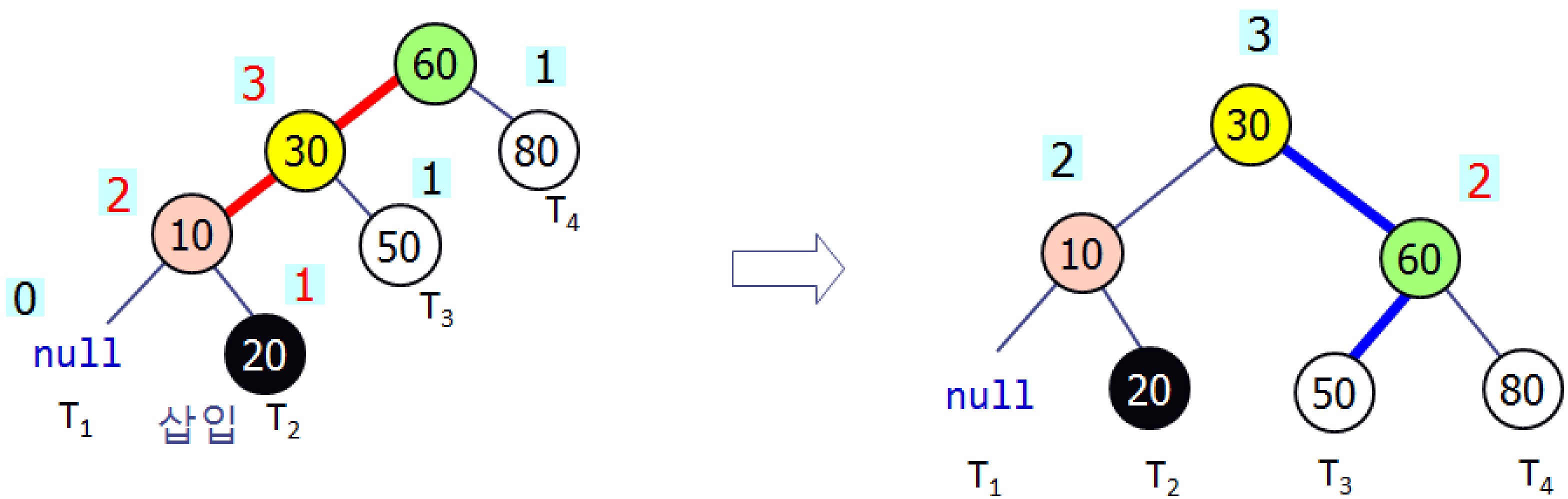
AVL트리의 회전연산

```
01 private Node rotateLeft(Node n) {
02     Node x = n.right; ①
03     n.right = x.left; ②
04     x.left = n;       ③
05     n.height = tallerHeight(height(n.left), height(n.right)) + 1; // 높이 갱신
06     x.height = tallerHeight(height(x.left), height(x.right)) + 1; // 높이 갱신
07     return x; ④ // 회전 후 x가 n의 이전 자리로 이동되었으므로
08 }
```

- Line 02 ~ 04와 07에 각각 부여된 번호 순서에 따라 link 변경
- 마지막으로 line 07에서 노드 x의 레퍼런스를 리턴

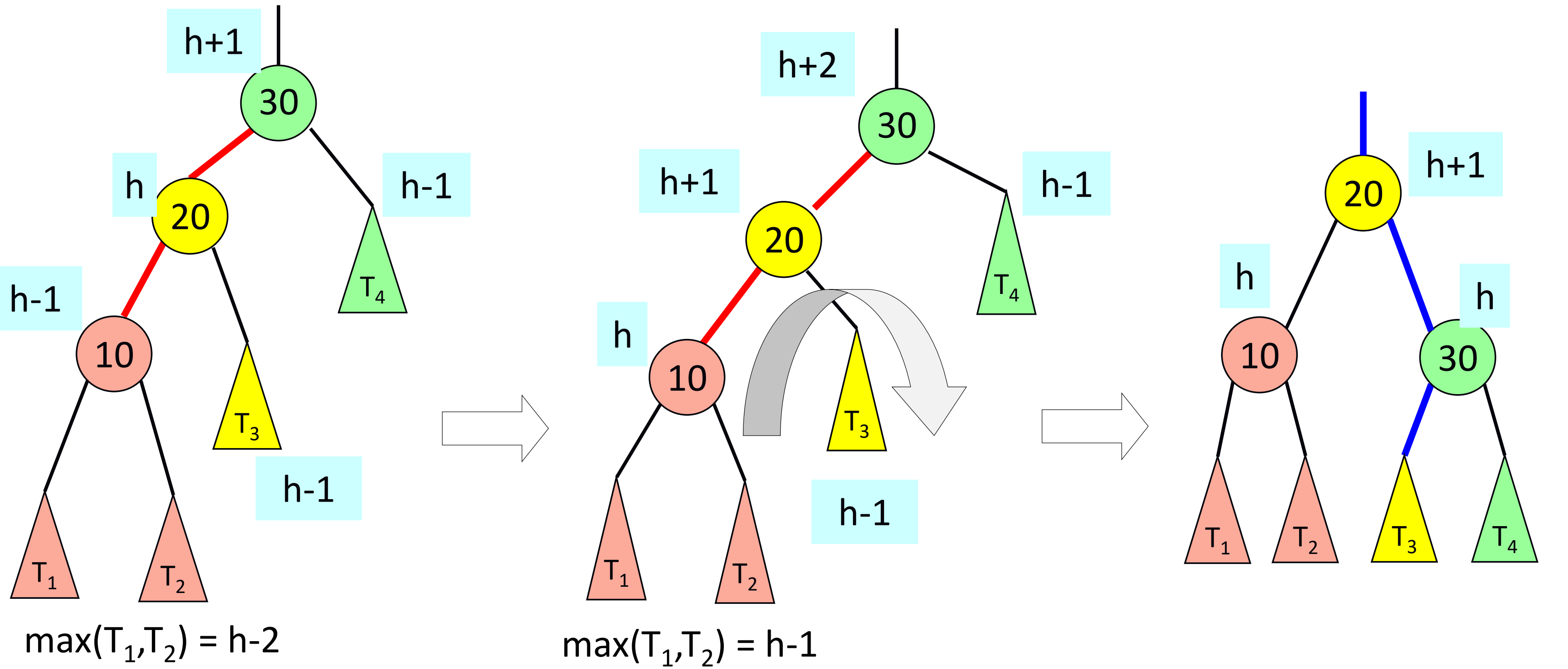
LL-회전

- **노드 10의 왼쪽 서브트리 T_1 또는 오른쪽 서브트리 T_2에 새 노드 삽입**
  - T_1 또는T_2의 높이 = h-1
  - 노드 60의 왼쪽과 오른쪽 서브트리의 높이 차이 = 2
    - 노드 60의 왼쪽(L) 서브트리의 왼쪽(L) 서브트리에 새로운 노드가 삽입되었기 때문



- **LL-회전은 rotateRight() 메소드를 사용**

LL 회전

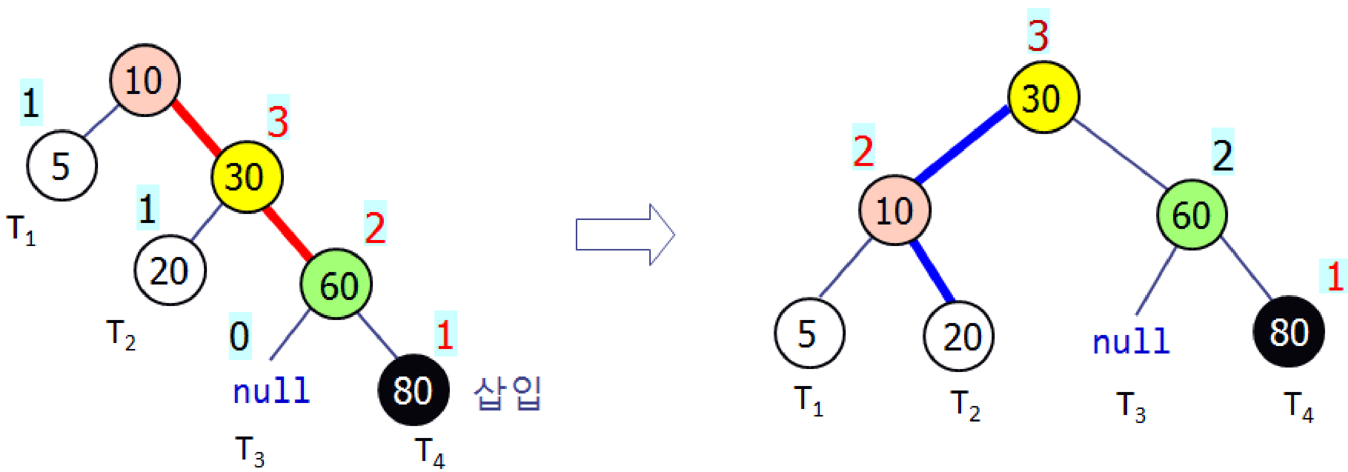


(a) T_1 또는T_2에 새 노드 삽입

(b) LL-회전 후

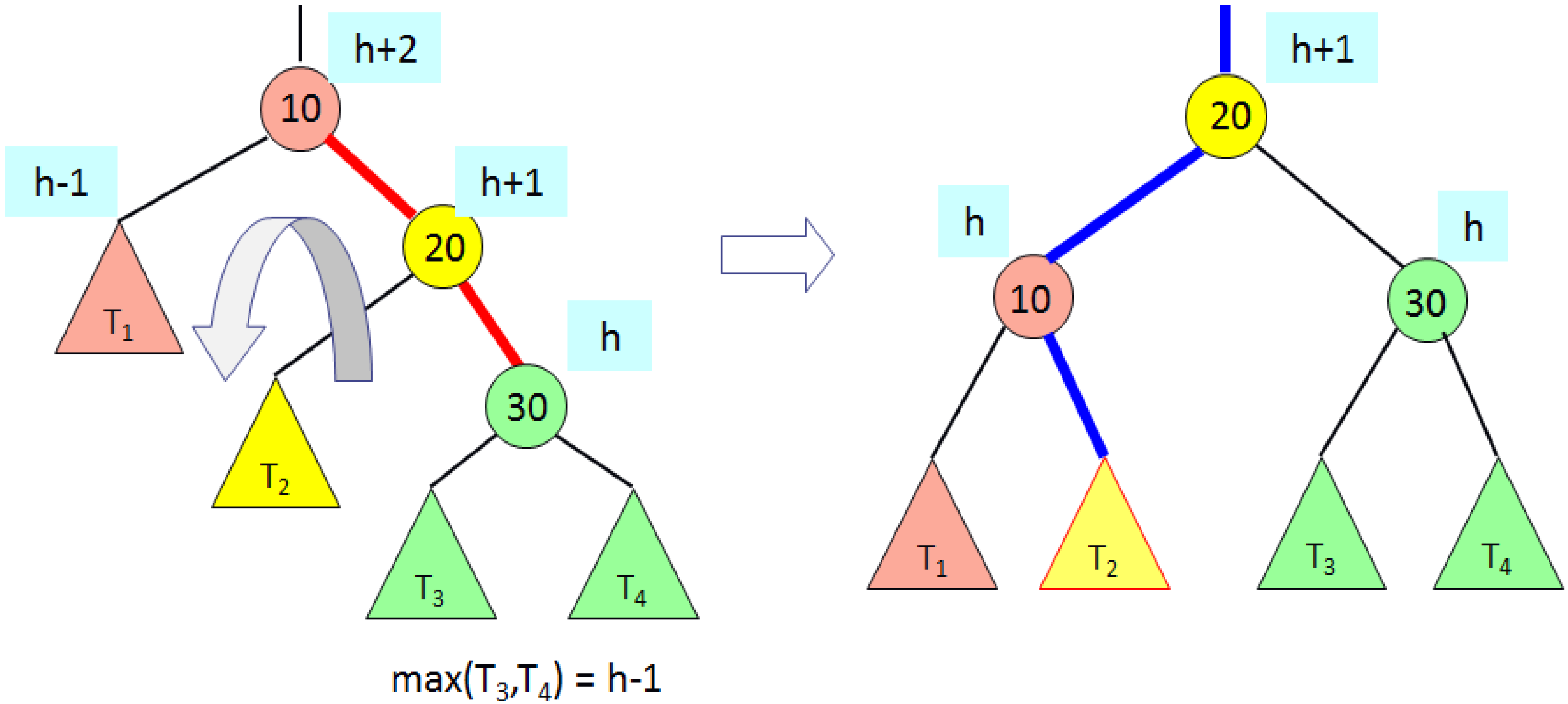
RR 회전

- **60의 왼쪽 서브트리(T_3) 또는 오른쪽 서브트리(T_4)에 새로운 노드 삽입**
  - T_3 또는T_4의 높이 = h-1
  - 노드 10의 왼쪽과 오른쪽 서브트리의 높이 차이 = 2
  - 노드 10의 오른쪽(R) 서브트리의 오른쪽(R) 서브트리에 새로운 노드가 삽입되었기 때문



- **RR-회전은 rotateLeft() 메소드를 사용**

RR 회전

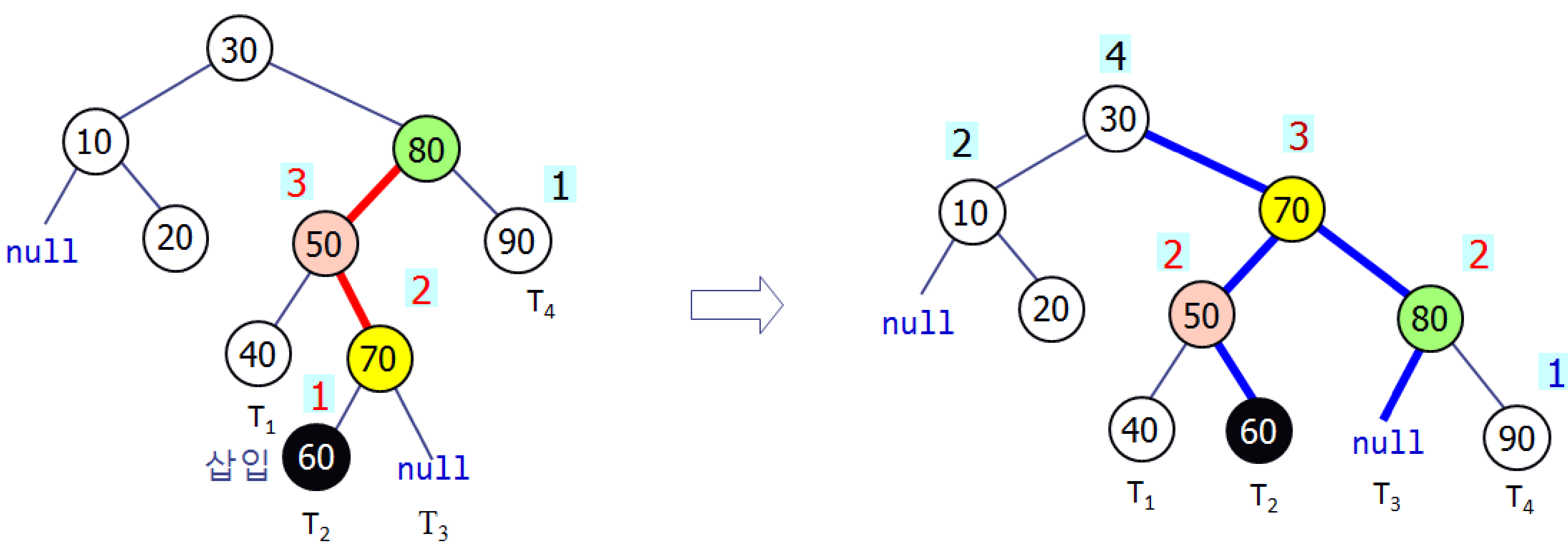


(a) T_3 또는T_4에 새 노드 삽입 (b) RR-회전 후

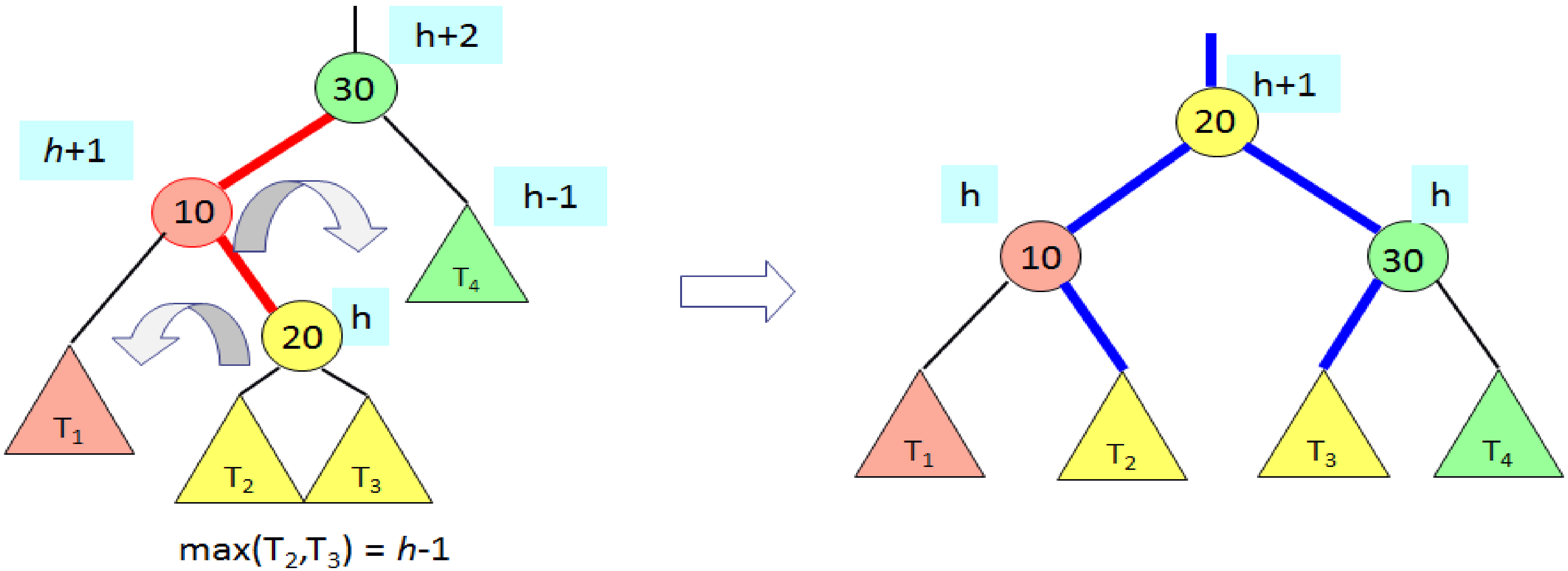
LR 회전

- (a) 70의 왼쪽 서브트리(T_2) 또는 오른쪽 서브트리(T_3)에 새로운 노드가 삽입 되어 T_2 또는T_3의 높이가 h-1이 됨에 따라, 80의 왼쪽과 오른쪽 서브트리의 높이 차이가 2가 된 상태
  - 80의 왼쪽(L) 서브트리의 오른쪽(R) 서브트리에서 새로운 노드가 삽입되었기 때문

- LR-회전은 rotateLeft(50)을 먼저 수행한 후 rotateRight(80)을 수행



LR 회전

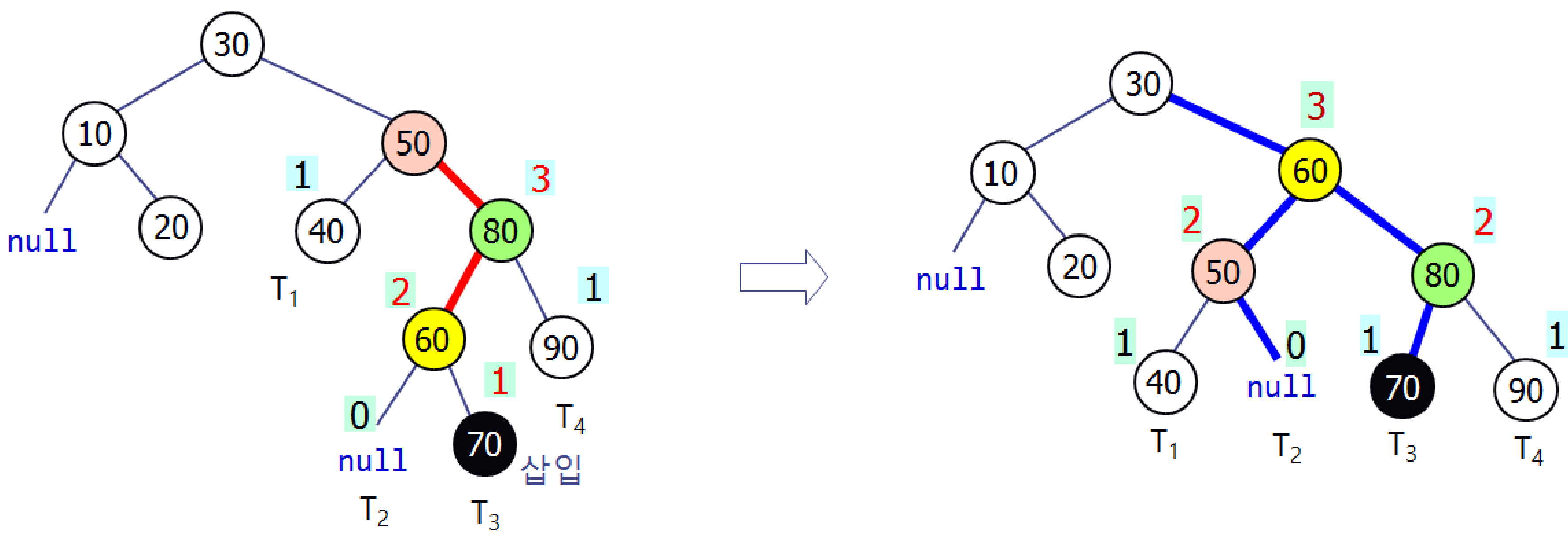


(a) T_2 또는T_3 에 새 노드 삽입

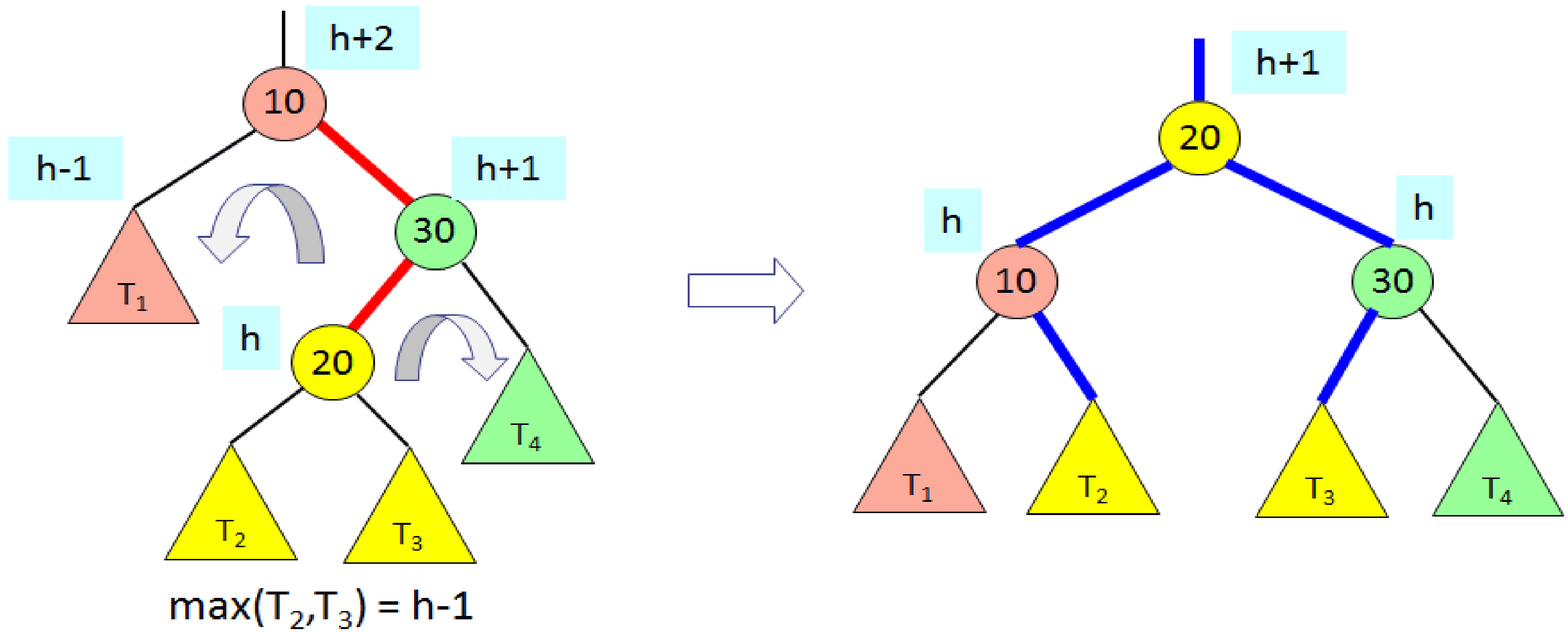
(b) LR-회전 후

RL회전

- (a) 60의 왼쪽 서브트리(T_2) 또는 오른쪽 서브트리(T_3)에 새로운 노드가 삽입 되어 T_2 또는T_3의 높이가 h-1이 되고 50의 왼쪽과 오른쪽 서브트리의 높이 차이가 2가 된 상태
  - 10의 오른쪽(R) 서브트리의 왼쪽(L) 서브트리에서 새로운 노드가 삽입되었기 때문
- RL-회전은 rotateRight(80)을 먼저 수행한 후 rotateLeft(50)을 수행



RL 회전



(a) T_2 또는T_3 에 새 노드 삽입

(b) RL-회전 후

4종류의 회전의 공통점

- **회전 후의 트리들이 모두 동일**
  - 각 그림(a)의 트리에서 10, 20, 30이 어디에 위치하든지,
    3개의 노드들 중에서 중간값을 가진 노드, 즉, 20이 위로 이동하면서
    10 과 30이 각각 20의 좌우 자식노드가 되기 때문
- **각 회전연산의 수행시간이 O(1)**
  - 각 그림(b)에서 변경된 노드 레퍼런스 수가 O(1) 개이기 때문

5.2.2 삽입 연산

- **AVL트리에서의 삽입은 2단계로 수행**
- **1 단계: 이진탐색트리의 삽입과 동일하게 새로운 노드 삽입**
- **2 단계: 새로 삽입한 노드로부터 루트로 거슬러 올라가며 각 노드의 서브트리 높이 차이를 갱신**
  - 이 때 가장 먼저 불균형이 발생한 노드를 발견하면, 이 노드를 기준으로 새 노드가 어디에 삽입되었는지에 따라 적절한 회전연산을 수행

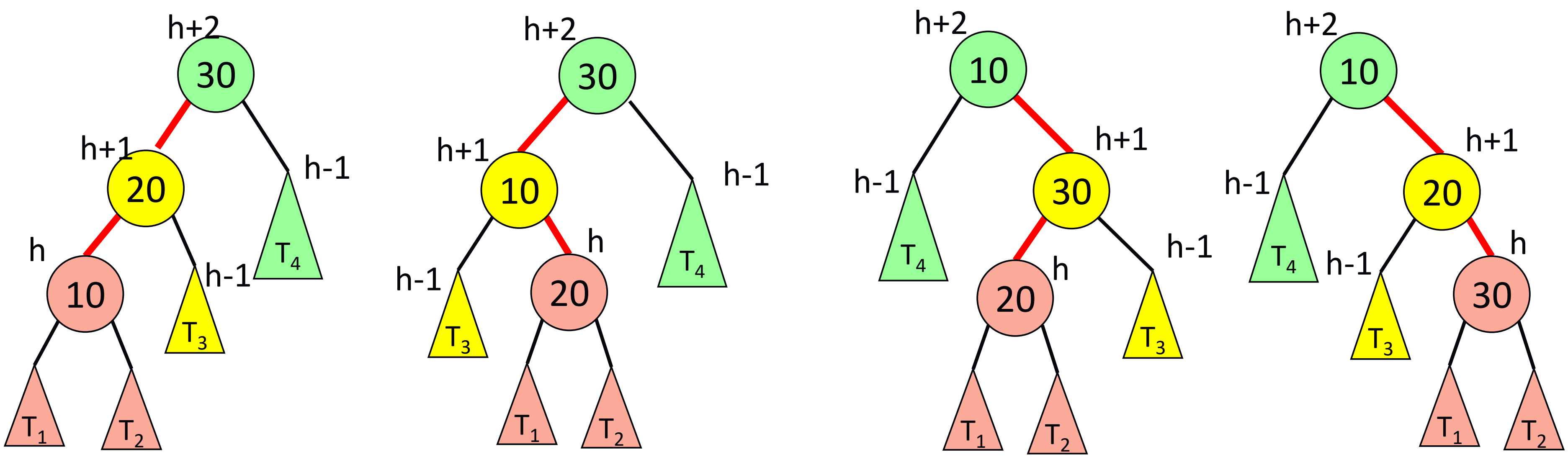
```java
public class Node {
    private Key id;
    private Value name;
    private int height;
    private Node left, right;
    public Node(Key newID, Value newName, int newHt) { // 생성자
        id = newID;
        name = newName;
        height = newHt;
        left = right = null;
    }
}
```

```java
01  public void put(Key k, Value v) {root = put(root, k, v);}
02  private Node put(Node n, Key k, Value v) {
03      if (n == null)  return new Node(k, v, 1);
04      int t = k.compareTo(n.id);
05      if (t < 0)        n.left  = put(n.left,  k, v);
06      else if (t > 0) n.right = put(n.right, k, v);
07      else {
08          n.name = v;   // k가 이미 트리에 있으므로 Value v만 갱신
09          return n;
10      }
11      n.height = tallerHeight(height(n.left), height(n.right)) + 1;
12      return balance(n); // 노드 n의 균형 점검 및 불균형을 바로 잡음
13  }
```

- **이진탐색트리의 put()과 거의 동일**
  - Line 11: 노드의 높이 계산
  - Line 12: balance() 메소드를 호출하여 불균형이 발생하였을 경우 적절한 회전연산을 수행 추가

- **Line 11: tallerHeight(int a, int b): a와 b 중에서 큰 값을 리턴**

```java
private Node balance(Node n) {
    if (bf(n) > 1) {  //노드 n의 왼쪽 서브트리가 높아서 불균형 발생
        if (bf(n.left) < 0) { //  노드 n의 왼쪽 자식노드의 오른쪽 서브트리가 높은 경우
            n.left = rotateLeft(n.left);
        }                                        // LR-회전
        n = rotateRight(n);   // LL-회전
    }
    else if (bf(n) < -1) {  //노드 n의 오른쪽 서브트리가 높아서 불균형 발생
        if (bf(n.right) > 0) {  //  노드 n의 오른쪽 자식노드의 왼쪽 서브트리가 높은 경우
            n.right = rotateRight(n.right);
        }                                          // RL-회전
        n = rotateLeft(n);  // RR-회전
    }
    return n;
}
```



- **balance() 메소드: 불균형 발생 시 회전연산으로 불균형 해소**
  - 현재 노드 n이 부모노드와 재 연결되기 바로 직전에 노드 n의 불균형 여부를 검사하고 적절한 회전연산으로 불균형을 해결
  - bf(n) > 1: 노드 n의 왼쪽 서브트리가 오른쪽 서브트리보다 높고,
    그 차이가 1보다 크므로 (line 02) 불균형 발생
    - bf(n.left)<0: n.left의 오른쪽 서브트리가 왼쪽보다 높아 불균형 발생됨(line 03)
    - Line 04: rotateLeft(n.left) 수행
    - Line 06에서 rotateRight(n) 수행 [LR-회전]
    - bf(n.left)가 음수가 아닌 경우: line 06에서 [LL-회전]
  - RR-회전과 RL-회전도 line 08~13에 따라 각각 수행되어 트리의 균형 유지
  - [참고] 현재 노드 n의 균형이 유지되어 있으면, if-와 else if- 문을 건너 뛰고 line 14에서 노드 n의 레퍼런스를 리턴

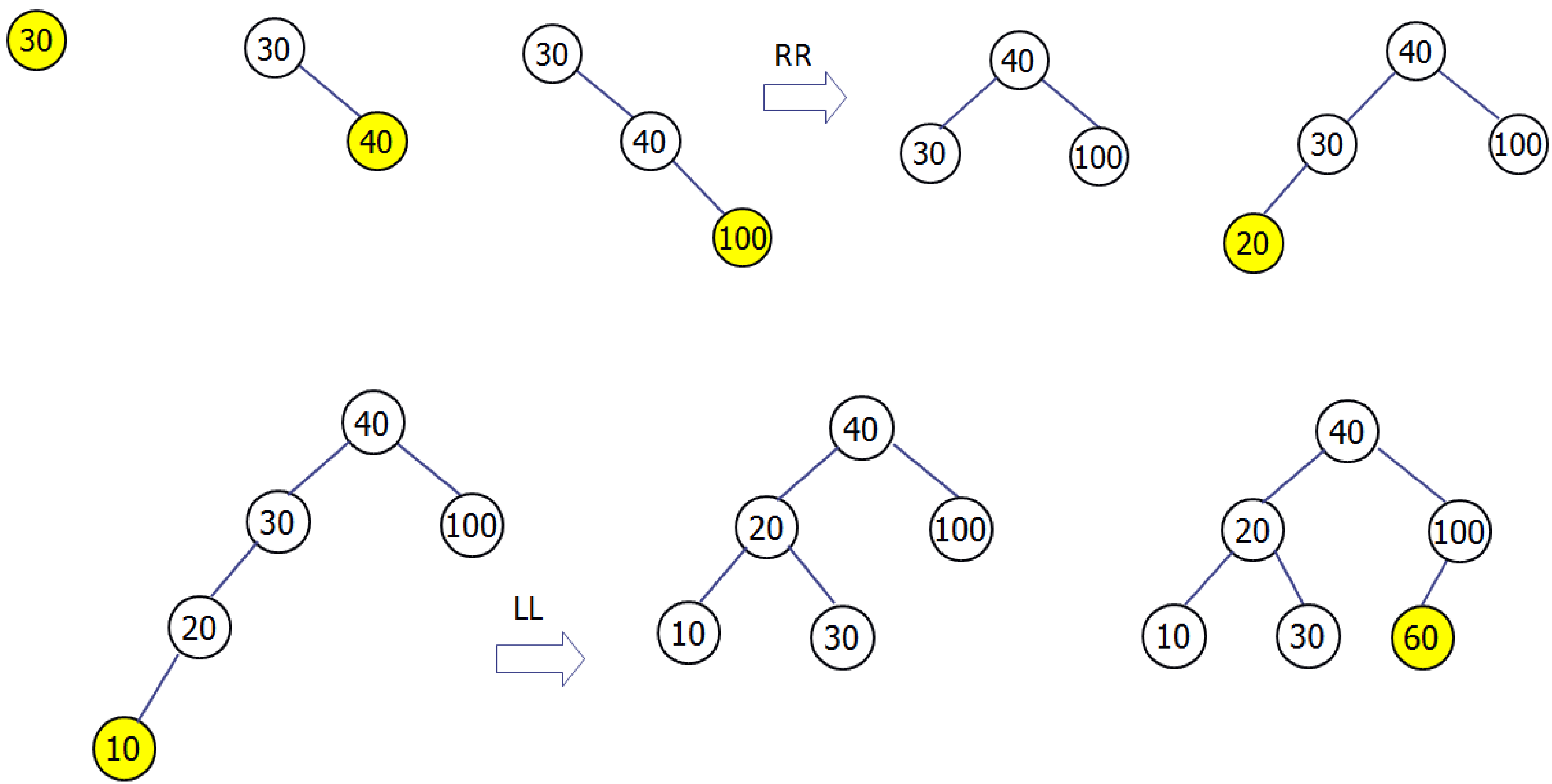
bf(n): (노드 n의 왼쪽 서브트리 높이) – (오른쪽 서브트리 높이) 리턴

```java
private int bf(Node n) {
    return height(n.left) - height(n.right);
}
```

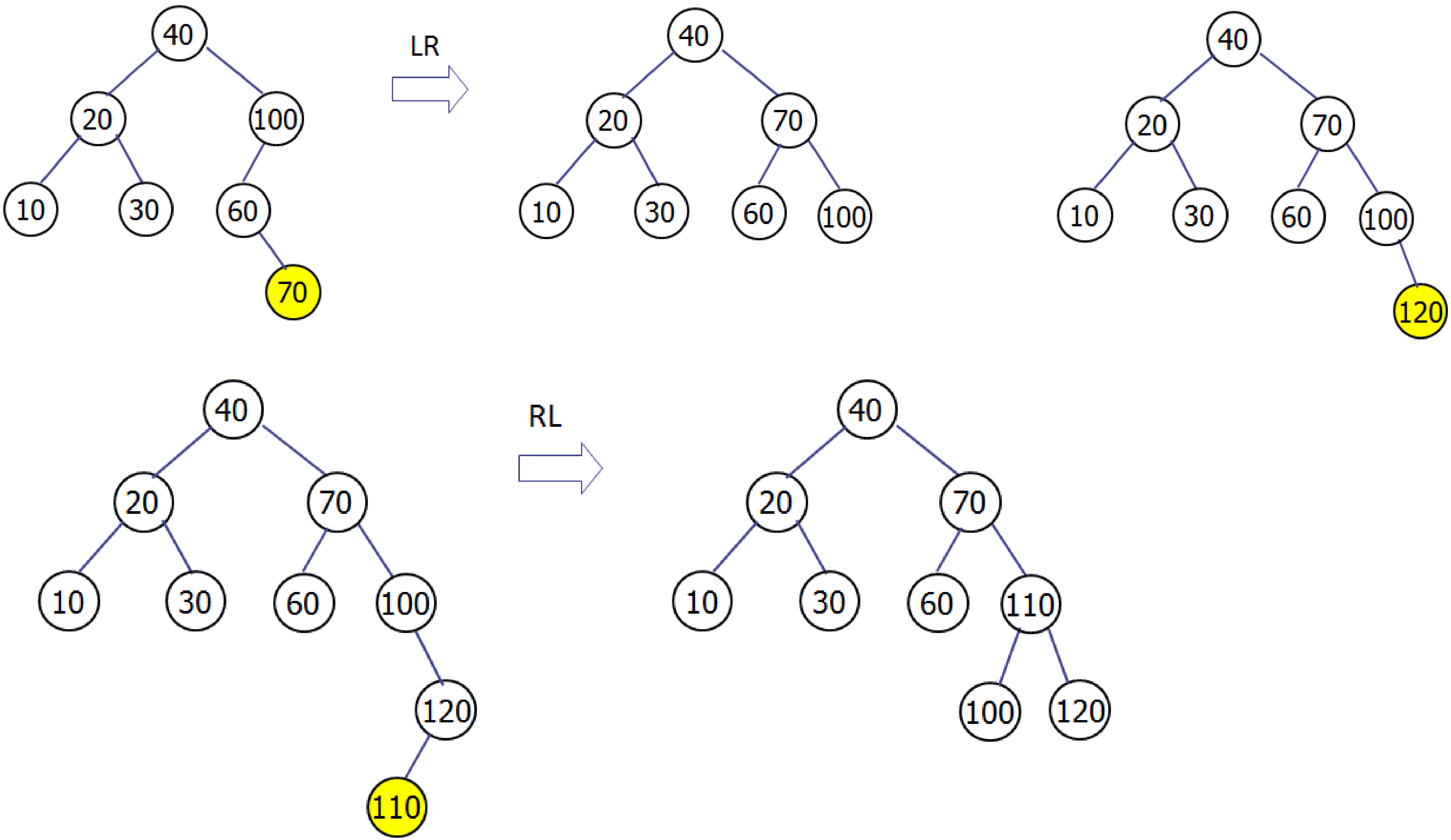
```java
private int height(Node n) {
    if (n == null) return 0;
    return n.height;
}
```

```java
private int tallerHeight(int x, int y){
    if (x>y) return x;
    else return y;
}
```

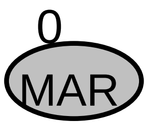
[예제] 30, 40, 100, 20, 10, 60, 70, 120, 110을 순차적으로 삽입



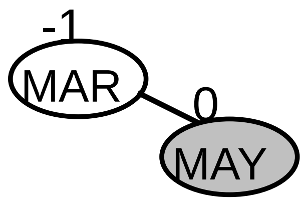
[예제] 30, 40, 100, 20, 10, 60, 70, 120, 110을 순차적으로 삽입



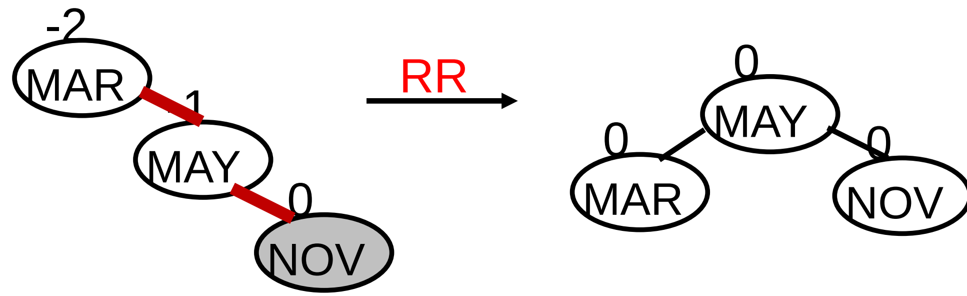
MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP 순 입력



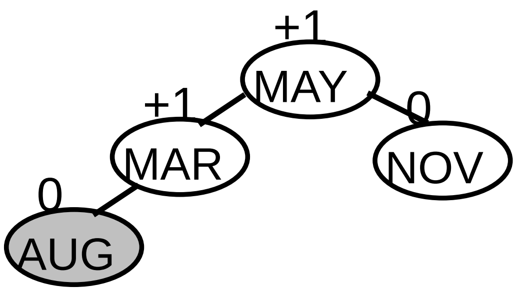
(a) 삽입 MARCH



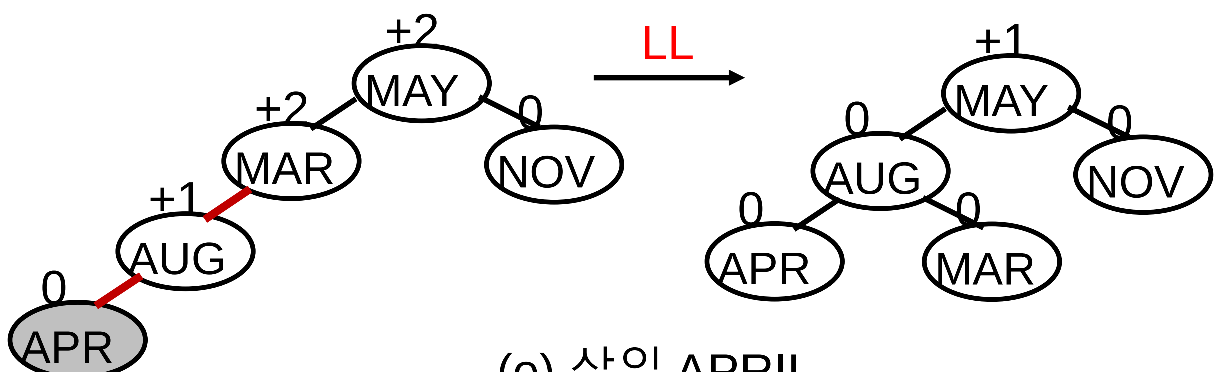
(b) 삽입 MAY



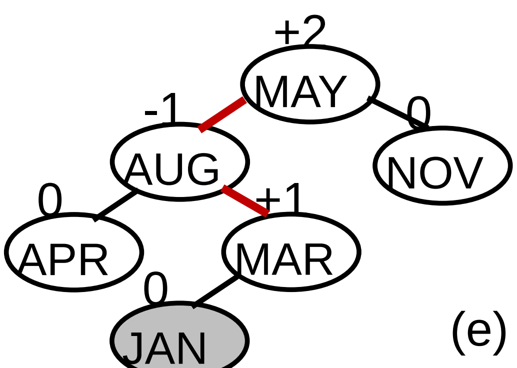
(c) 삽입 NOV

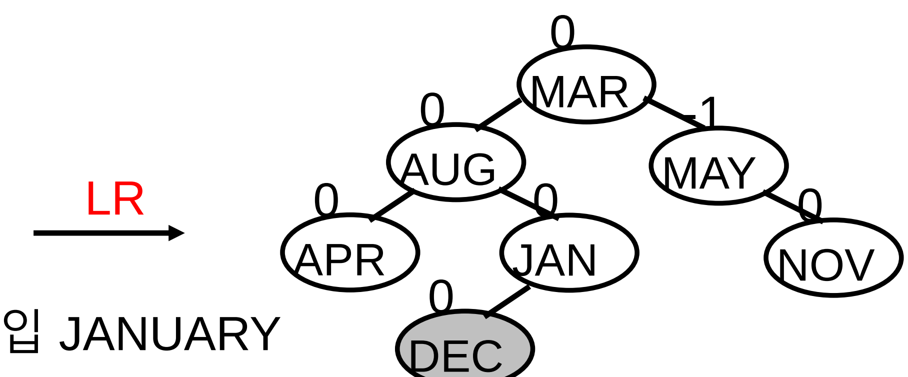


(d) 삽입 AUGUST



(e) 삽입 APRIL

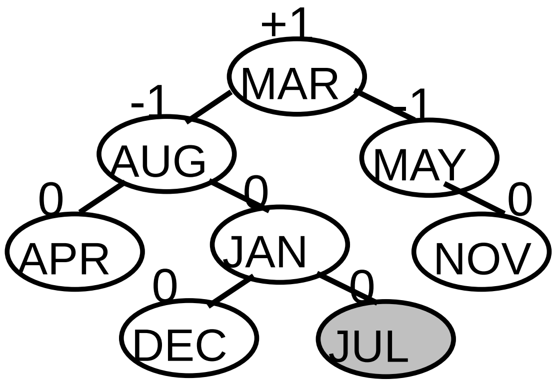




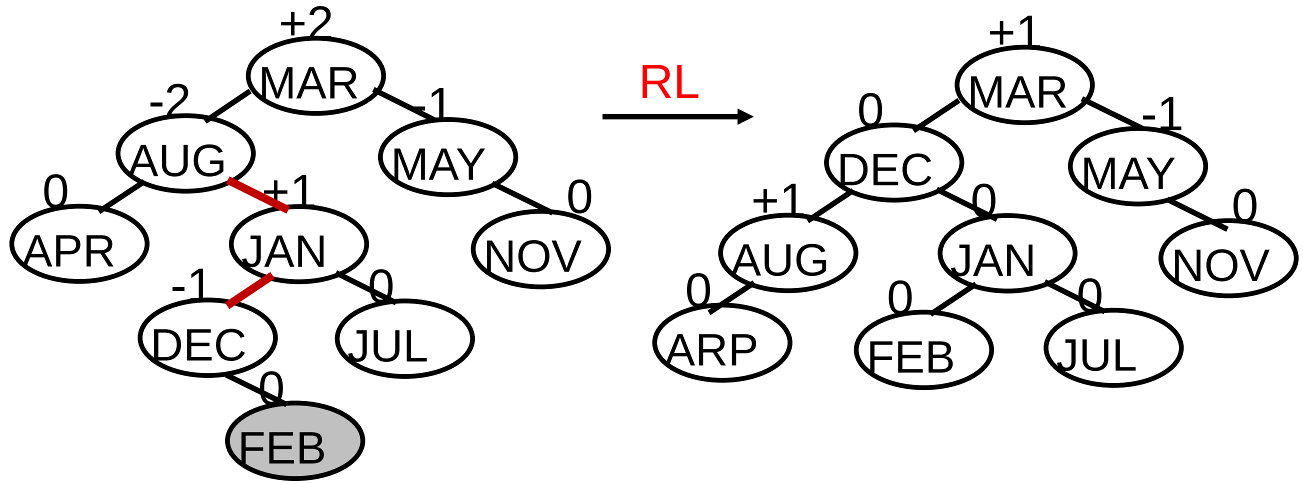
(e) 삽입 JANUARY

(g) 삽입 DECEMBER

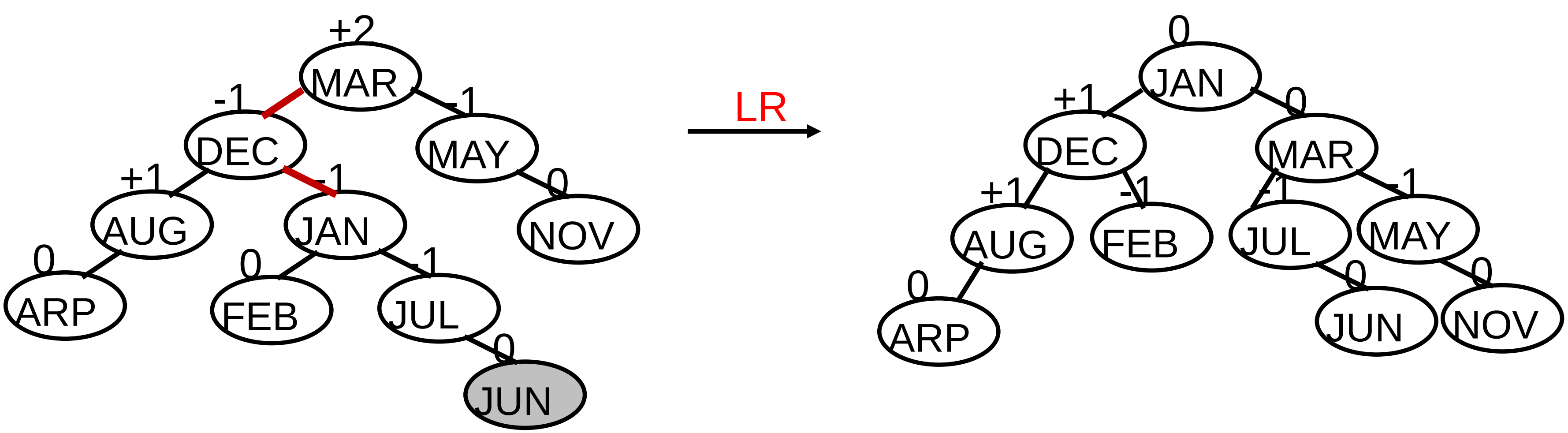
MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP 순 입력



(h) 삽입 JULY

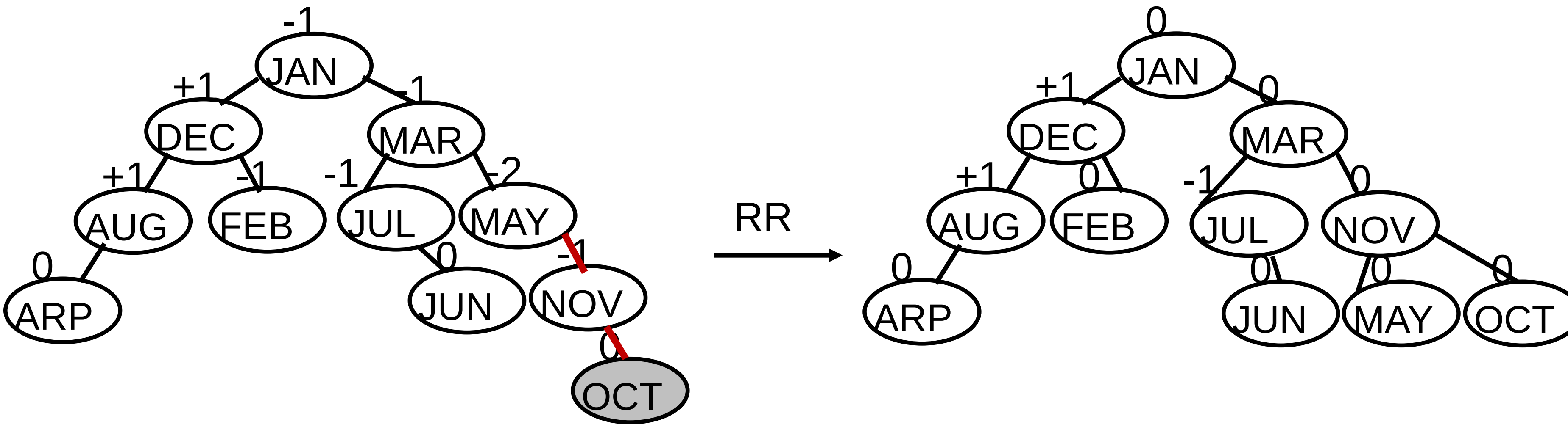


(i) 삽입 FEBRUARY

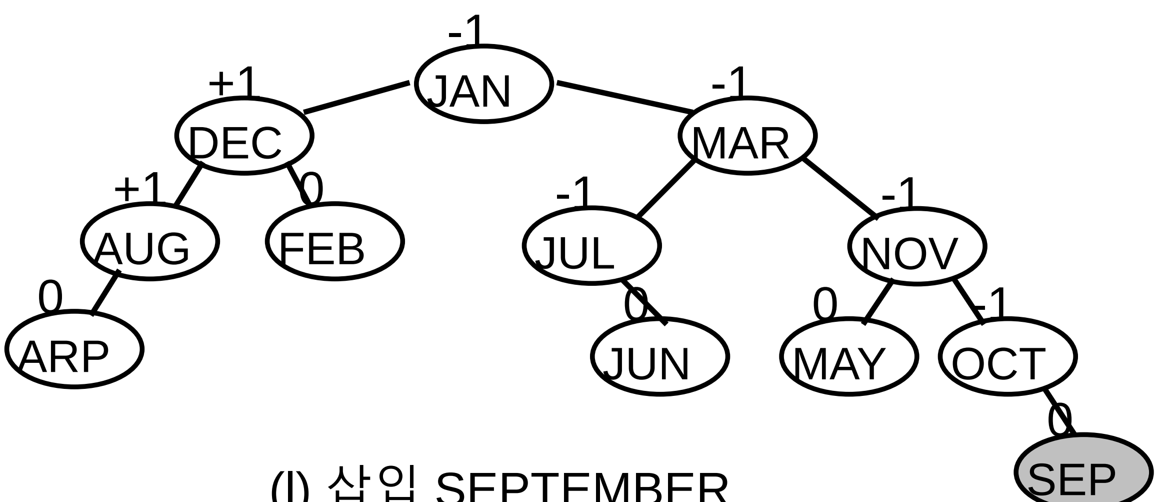


(j) 삽입 JUNE

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



(k) 삽입 OCTOBER

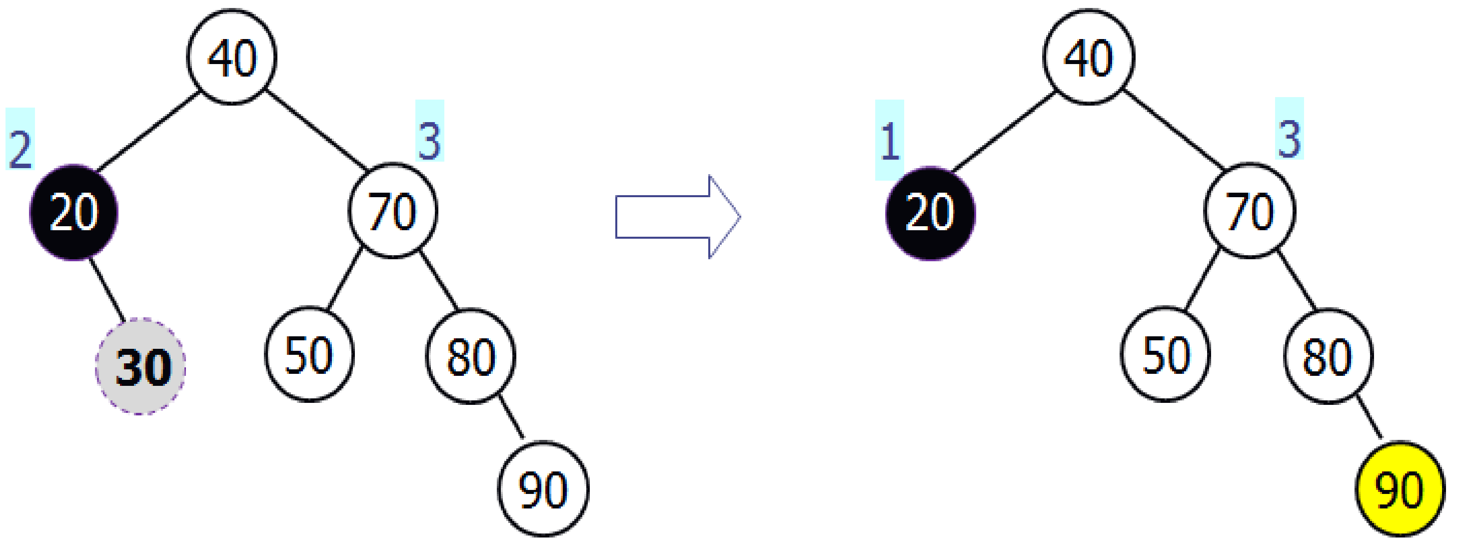


(l) 삽입 SEPTEMBER

5.2.3 삭제 연산

- **AVL트리에서의 삭제는 2단계로 진행**
- **[1단계] 이진탐색트리에서와 동일한 삭제 연산 수행**
- **[2단계] 삭제된 노드로부터 루트노드 방향으로 거슬러 올라가며 불균형이 발생한 경우 적절한 회전연산 수행**
  - 회전연산 수행 후에 부모노드에서 불균형이 발생할 수 있고, 이러한 일이 반복되어 루트노드에서 회전연산을 수행해야 하는 경우도 발생

30을 가진 노드의 삭제

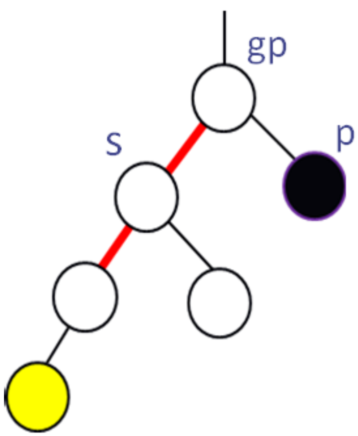


(a) 삭제 전 (b) 삭제 후 노드 40에서 불균형 발생

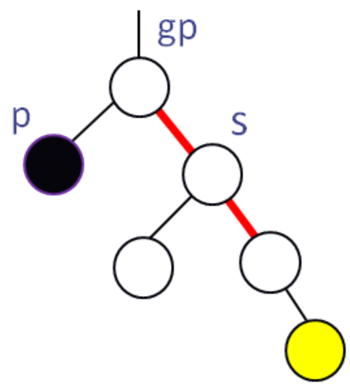
[핵심 아이디어]

삭제 후 불균형이 발생하면 반대쪽에 삽입이 이루어져 불균형이 발생한 것으로 취급하자

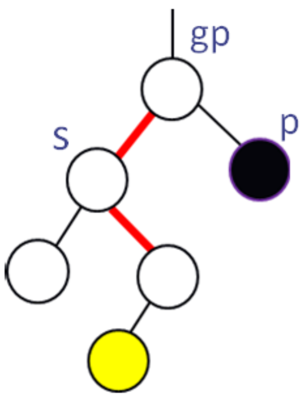
- 삭제된 노드의 부모노드 = p=, p의 부모노드 = gp, p의 형제노드= s라 하면, s의 왼쪽과 오른쪽 서브트리 중에서 높은 서브트리에 마치 새 노드가 삽입된 것으로 간주



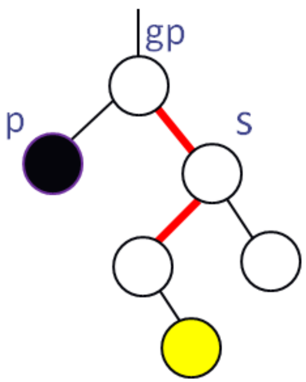
(a) LL-회전



(b) RR-회전

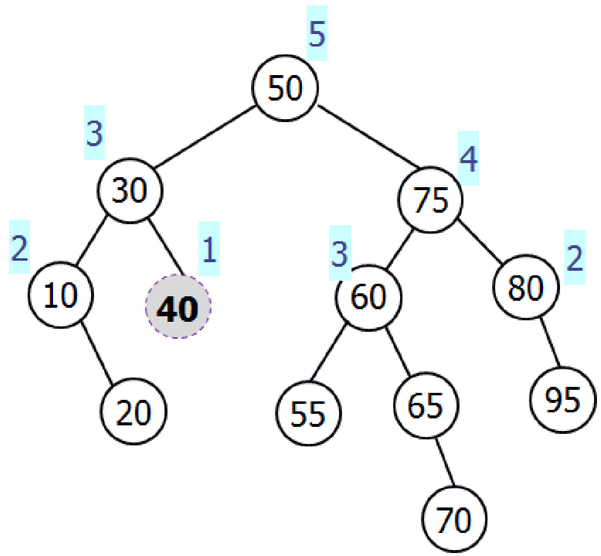


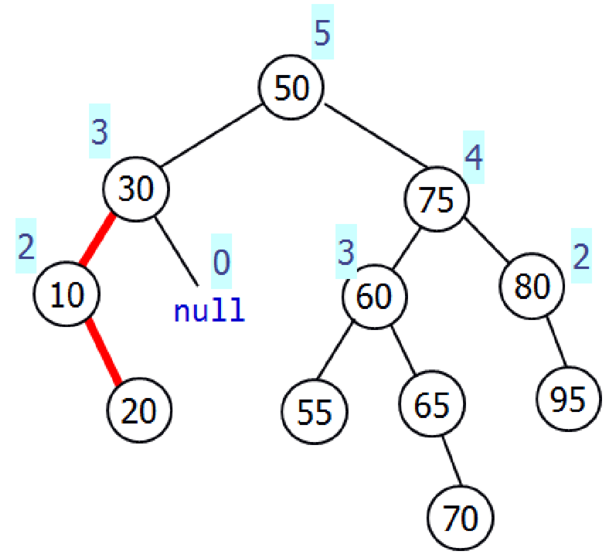
(c) LR-회전

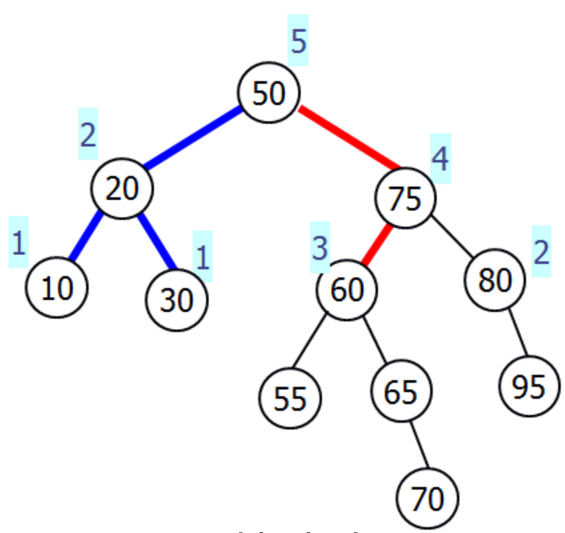


(d) RL-회전

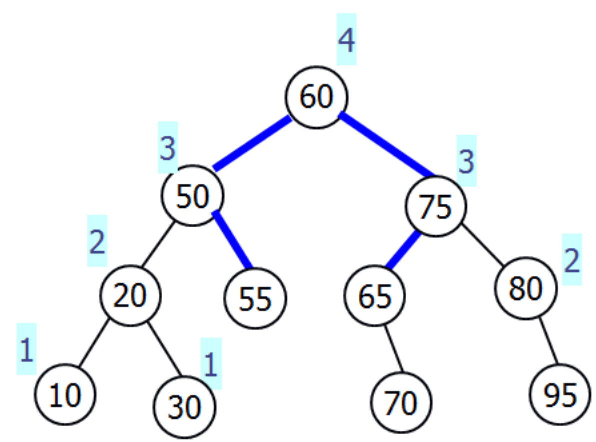
[예제] 40을 삭제







LR-회전 후



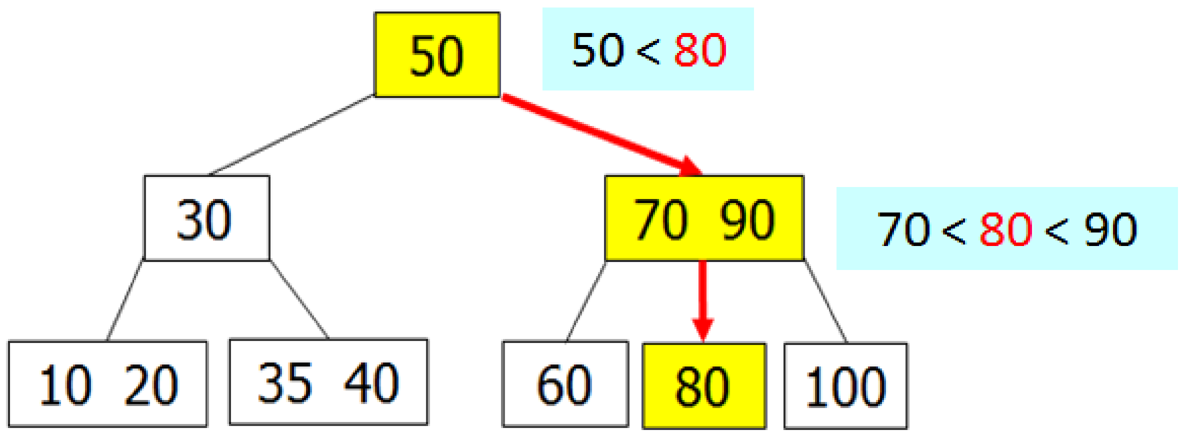
RL-회전 후

수행시간

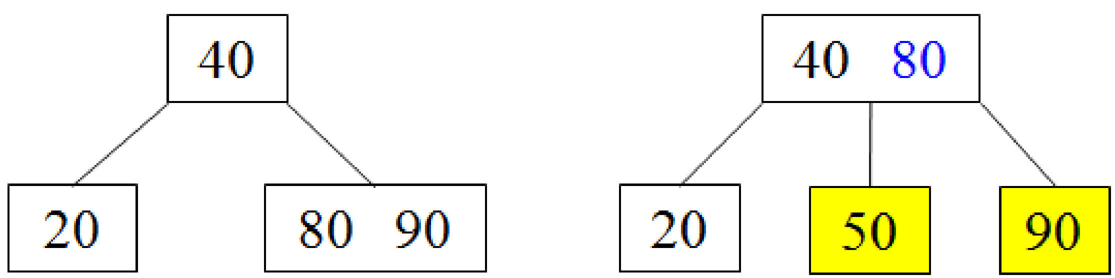
- AVL트리에서의 탐색, 삽입, 삭제 연산은 공통적으로 루트노드부터 탐색을 시작하여 최악의 경우에 이파리노드까지 내려가고, 삽입이나 삭제 연산은 다시 루트까지 거슬러 올라가야
- 트리를 1 층 내려갈 때는 재귀호출하며, 1 층을 올라갈 때 불균형이 발생하면 적절한 회전연산을 수행하는데, 이들 각각은 O(1) 시간 밖에 걸리지 않음
- 탐색, 삽입, 삭제 연산의 수행시간은 각각 AVL의 높이에 비례하므로 각 연산의 수행시간은 O(logN)
- 다양한 실험결과에 따르면, AVL트리는 거의 정렬된 데이터를 삽입한 후 랜덤 순서로 데이터를 탐색하는 경우 가장 좋은 성능을 보임
- 이진탐색트리는 랜덤 순서의 데이터를 삽입한 후에 랜덤 순서로 데이터를 탐색하는 경우 가장 좋은 성능을 보임

5.3 2-3트리

- **2-3트리는 내부노드의 차수가 2 또는 3인 균형 탐색트리**
  - 2-3트리는 루트로부터 각 이파리노드까지 경로의 길이가 같고, 모든 leaf node들이 동일한 층에 있는 완전한 균형트리



[예제]
50의 삽입

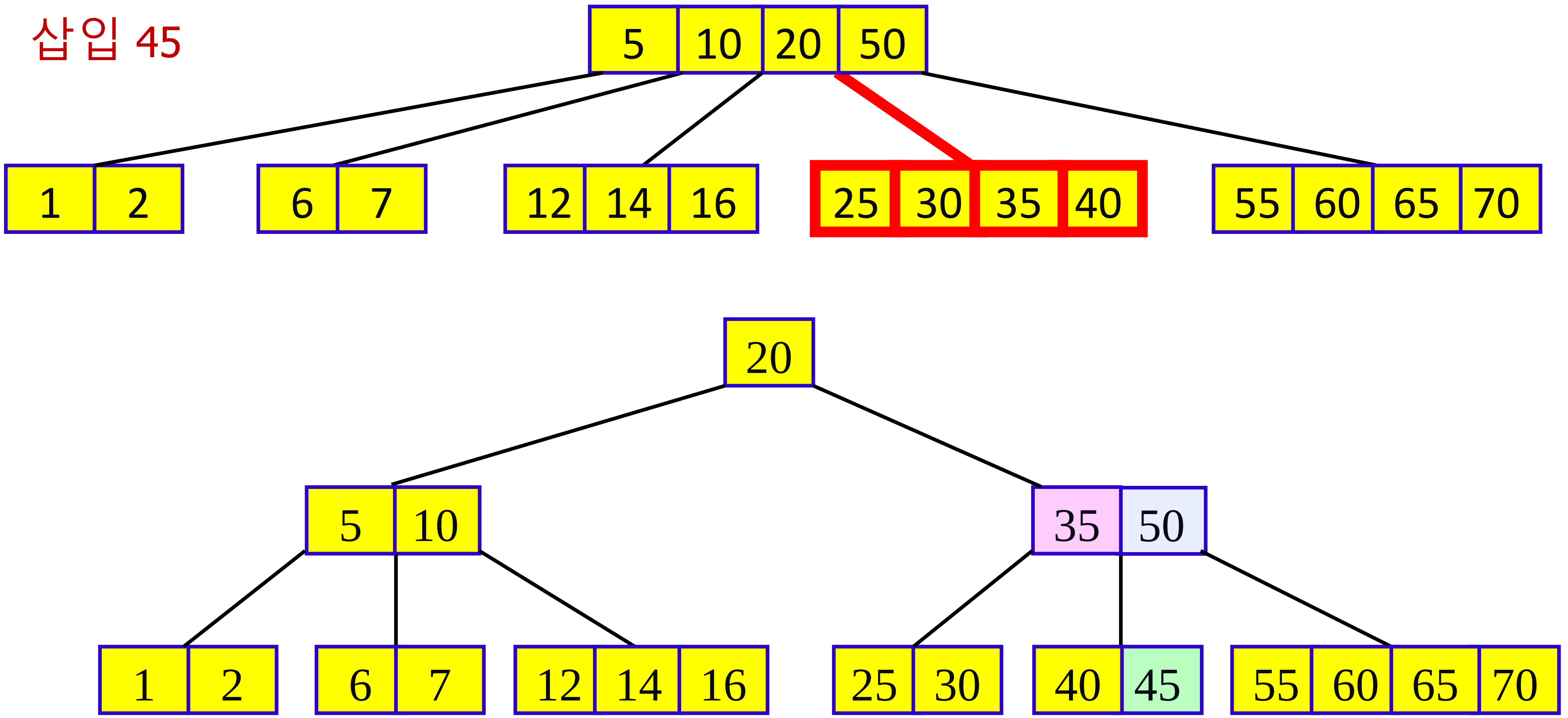


(a) 삽입 전　　(b) 삽입 후

5.5 B-트리

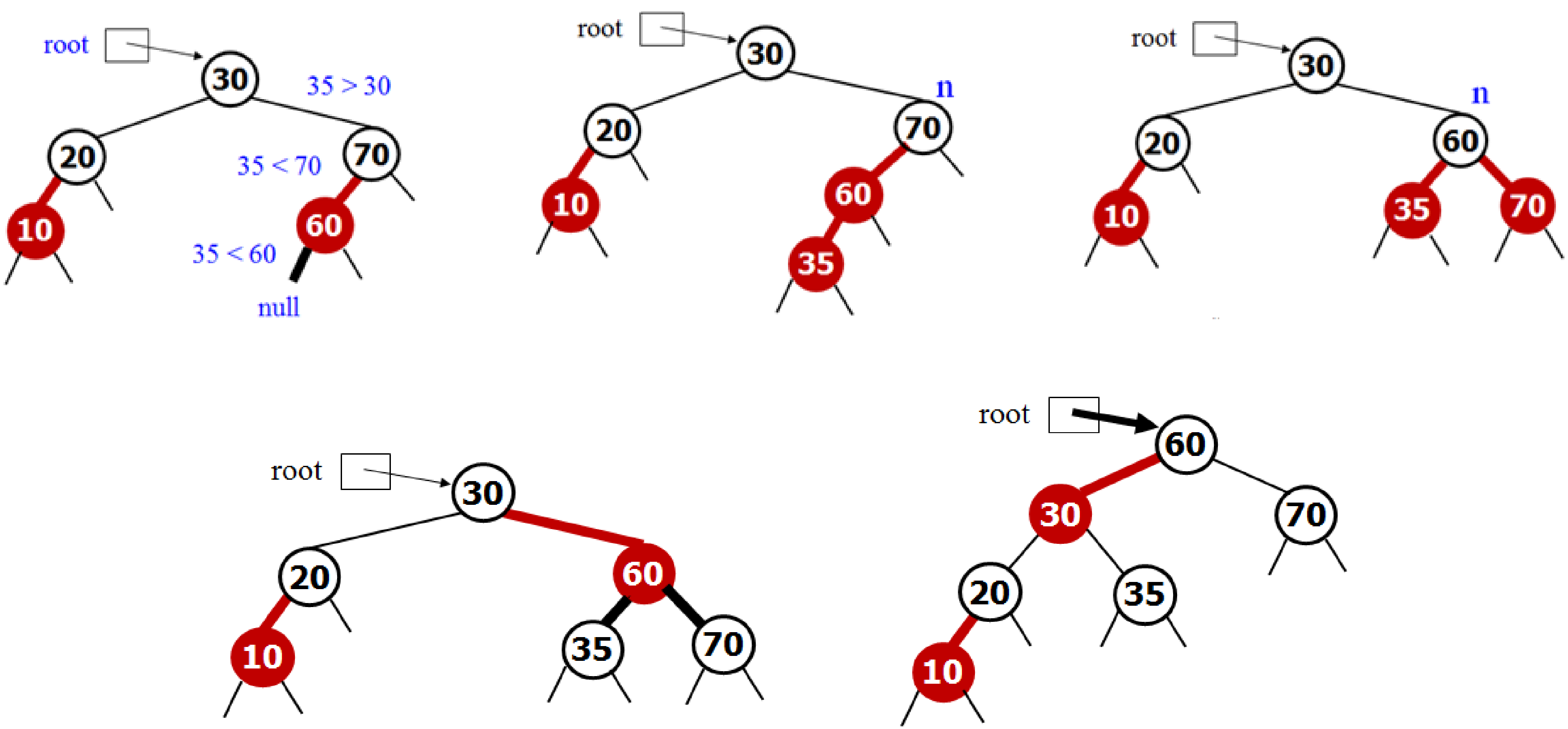
- **다수의 키를 가진 노드로 하여 트리의 탐색트리의 높이를 낮추는 구조**
  - 2-3트리는 B-트리의 일종으로 노드에 키가 2 개까지 있을 수 있는 트리
  - B-트리는 대용량의 데이터를 위해 고안되어 주로 데이터베이스에 사용

삽입 45



5.4 레드블랙트리

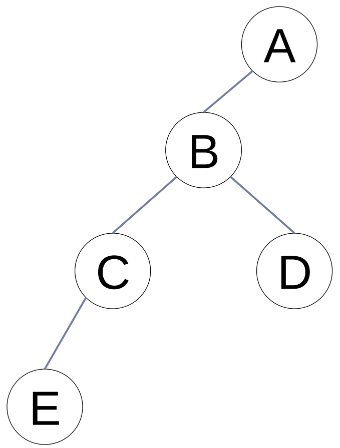
- 노드에 색을 부여하여 트리의 균형을 유지
- 탐색, 삽입, 삭제 연산의 수행시간이 각각 O(logN)을 넘지 않는 매우 효율적인 자료구조



요약

- 이진탐색트리는 이진탐색의 개념을 트리 형태의 구조에 접목시킨 자료구조
- 이진탐색트리의 각 노드 n의 키가 n의 왼쪽 서브트리의 키들보다 크고, n의 오른쪽 서브트리의 키들보다 작다.
- 이진탐색트리 탐색, 삽입, 삭제 연산의 수행시간은 각각 트리 높이에 비례
- AVL트리는 임의의 노드 x에 대해 노드 x의 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 차이가 1을 넘지 않는 이진탐색트리
- AVL트리는 트리가 한쪽으로 치우쳐 자라나는 것을 LL, LR, RR, RL-회전 연산들을 통해 균형을 유지
- AVL트리의 탐색, 삽입, 삭제 연산의 수행시간은 각각 O(logN)

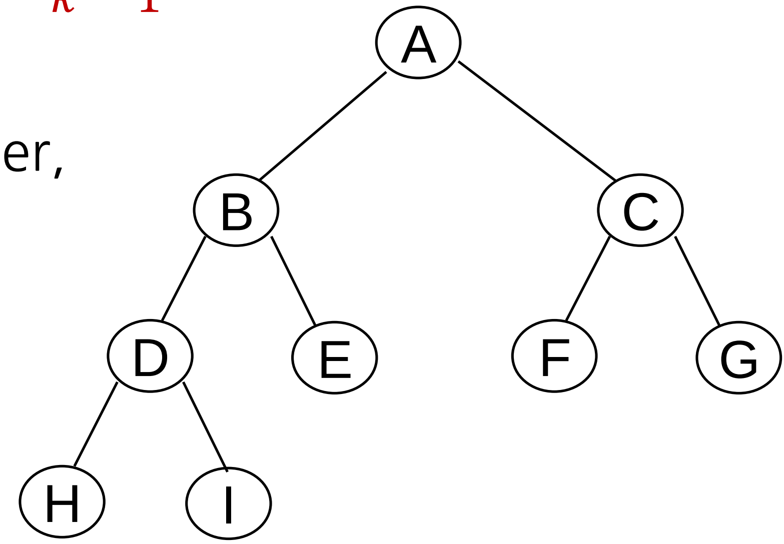
- **Problem1**
  - What are leaf nodes and non-leaf nodes?
  - What is the level of each node?



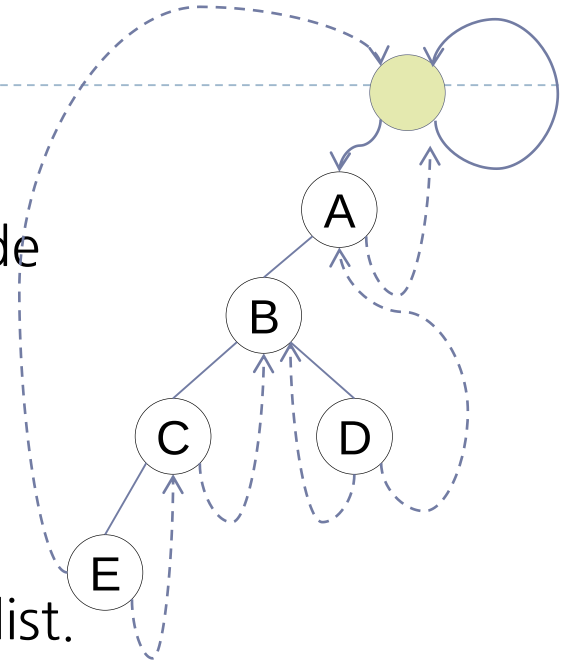
- **Problem2**
  - What is the maximum number of nodes in a k-ary tree of height h?

$$1 + k + k^2 + k^3 + \cdots + k^{h-1} = \frac{k^h - 1}{k - 1}$$

- **Problem3**
  - Write out the inorder, preorder, postorder, and level-order traversal for the two trees in this page.



- **Problem4**
  - Draw the threaded binary tree with a head node for the right tree.



- **Problem5**
  - Draw the binary search tree for a give integer list.
  - (5 7 9 4 1 2 3 6 8)