CSE 206 - 파일 처리론
(File Processing)

6 장. Index Structure

Contents

6.1. An overview of Index structure

Index : 색인

Left Image Source : https://www.betterweb.or.kr/blog/%EC%9B%B9%EA%B3%BC-%EC%9B%B9-%EA%B2%80%EC%83%89-%EC%9B%B9-%EA%B2%80%EC%83%89%EC%9D%98-%EC%9D%B4%ED%95%B4-%EC%83%89%EC%9D%B8/

Right Image source: https://ko.wikipedia.org/wiki/%ED%8C%8C%EC%9D%BC:%EA%B5%90%EB%B3%B4%EB%AC%B8%EA%B3%A0_%EB%8F%84%EC%84%9C_%EB%AC%B4%EC%9D%B8%EA%B2%80%EC%83%89%EB%8C%80.jpg

Index의 목적

- **필요한 것을 찾기 위해 필요한 것의 위치를 Key를 기반으로 체계적으로 저장해 놓은 정보.**
  - Index는 찾으려는 대상 (이하 타깃(target))에 대한 **위치 정보**가 있어야 한다.
  - 하나의 타깃을 찾기 위한 Index는 **여러 개 있을 수 있다.**
    - 여러 관점에서 타깃을 찾으려 할 수 있다.
      - 예) 책을 찾을 때, 타이틀, 저자, ISBN, 출판연도 등 여러 관점에서 검색할 수 있다.
  - Index의 Key는 **체계적으로 구성**되어 있어야 한다.
    - 무언가를 찾으려는 것이 목적인데 Index에서 타깃을 찾기 어려우면 Index로서의 역할을 하기 어렵다.

파일 구조에서의Index

- **목적**
  - 파일의 Record들에 대한 효율적 접근을 위하여 구성.
    - 필요한 Record를 빠르게 찾기 위해Index를 구성
      - Index가 없으면 풀 스캔(Full scan) 해서 찾아야 할 수도 있다.

- **특징**
  - <Key 값, Record 주소(포인터)> 쌍으로 구성
    - 최소한 Key 값과 Record 주소 (위치 정보)는 있어야 한다.

Index 분류

- **Key 값의 유형에 따른Index**
  - **Primary index :** Key 값이 **Primary Key**인Index
  - **Secondary index :** Primary Index 이외의Index

Primary key

Candidate key

학번	이름	나이	본적	성별	주민등록 번호
1243	홍길동	10	서울	남	yymmdd-1xxxxxx
1332	박영희	19	충청	여	yymmdd-2xxxxxx
1334	이기수	21	전라	남	yymmdd-1xxxxxx
1367	정미영	20	경상	여	yymmdd-2xxxxxx
1257	김철수	20	경기	남	yymmdd-1xxxxxx
1440	최미숙	21	강원	여	yymmdd-2xxxxxx

<학번이 Primary Key로 되어 있는 데이터>

Primary Key

- **Candidate Key:** 각각의 Record를 유일하게 식별(다른 Record와 구별)할 수 있는, 필드들의 부분집합
  - Candidate Key 값은 중복되어서는 안된다.
    - Record를 유일하게 식별하기 위해 필요한 속성
    - 예) 사람이 다른데 학번이 같은 학생들이 있으면 학번은 후보Key가 될 수 없다.

Candidate key

Candidate key

학번	이름	나이	본적	성별	주민등록 번호
1243	홍길동	10	서울	남	yymmdd-1xxxxxx
1332	박영희	19	충청	여	yymmdd-2xxxxxx
1334	이기수	21	전라	남	yymmdd-1xxxxxx
1367	정미영	20	경상	여	yymmdd-2xxxxxx
1257	김철수	20	경기	남	yymmdd-1xxxxxx
1440	최미숙	21	강원	여	yymmdd-2xxxxxx

Primary Key (Cont'd)

- **Candidate Key (이어서)**
  - 최소성을 만족시켜야 한다.
    - Record를 유일하게 식별하는데 필요한 최소한의 필드만 넣어야 한다.
      - 예) (학번, 이름), (주민등록번호, 학번) 쌍으로도 학생은 유일하게 구분할 수 있다.
      - 다만 이름은 없어도 구분 되므로 후보 Key가 될 수 있는 필드는 "학번", "주민등록번호"이다.
        - (학번, 이름), (주민등록번호, 학번) 쌍은 슈퍼 Key(Super key).
- **Primary Key:** Candidate Key 중에서 테이블 설계자가 선택한 Main Key.

Primary key

Candidate key

학번	이름	나이	본적	성별	주민등록 번호
1243	홍길동	10	서울	남	yymmdd-1xxxxxx
1332	박영희	19	충청	여	yymmdd-2xxxxxx
1334	이기수	21	전라	남	yymmdd-1xxxxxx
1367	정미영	20	경상	여	yymmdd-2xxxxxx
1257	김철수	20	경기	남	yymmdd-1xxxxxx
1440	최미숙	21	강원	여	yymmdd-2xxxxxx

<학번이 Primary Key로 되어 있는 데이터>

Index 분류

- **Key 값의 유형에 따른Index**
  - **Primary index :** Key 값이 **Primary Key**인Index
  - **Secondary index :** Primary Index 이외의Index

Index의 Key 값이 학번이면 Primary Index

Index의 Key 값이 나이이면 Secondary Index

Index의 Key 값이 주민등록 번호이면 Primary Index 아니면 Secondary Index?

Primary key

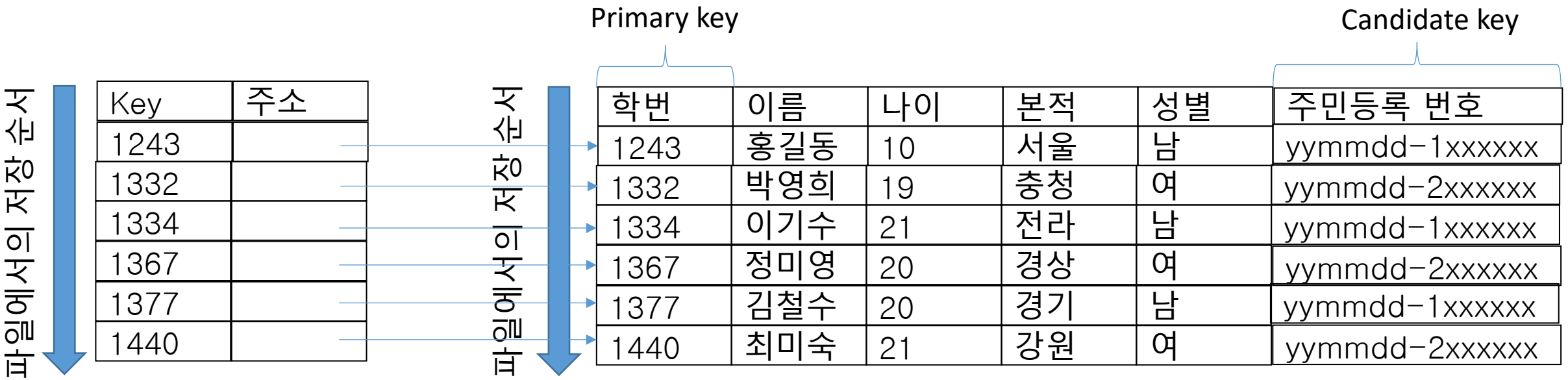
Candidate key

학번	이름	나이	본적	성별	주민등록 번호
1243	홍길동	10	서울	남	yymmdd-1xxxxxx
1332	박영희	19	충청	여	yymmdd-2xxxxxx
1334	이기수	21	전라	남	yymmdd-1xxxxxx
1367	정미영	20	경상	여	yymmdd-2xxxxxx
1257	김철수	20	경기	남	yymmdd-1xxxxxx
1440	최미숙	21	강원	여	yymmdd-2xxxxxx

<학번이 기본Key로 되어 있는 데이터>

Index 분류 (Cont'd)

- **파일 구조에 따른Index**
  - 집중Index (clustered index) : 파일의 데이터 Record들의 물리적 순서와Index 엔트리 순서가 동일한Index
    - 한 파일에 하나의 clustered index만 존재할 수 있음.
  - 비집중Index (unclustered index) : 집중 형태가 아닌Index



<Clustered Index 예>

Index 분류 (Cont'd)

- **데이터 범위에 따른Index**
  - **밀집Index(dense index)** : 데이터 Record 하나에 하나의Index 엔트리가 만들어지는Index. 역Index(inverted index)
  - **희소Index(sparse index)** : 데이터 Record 그룹 또는 데이터 블록에 하나의 엔트리가 만들어지는Index

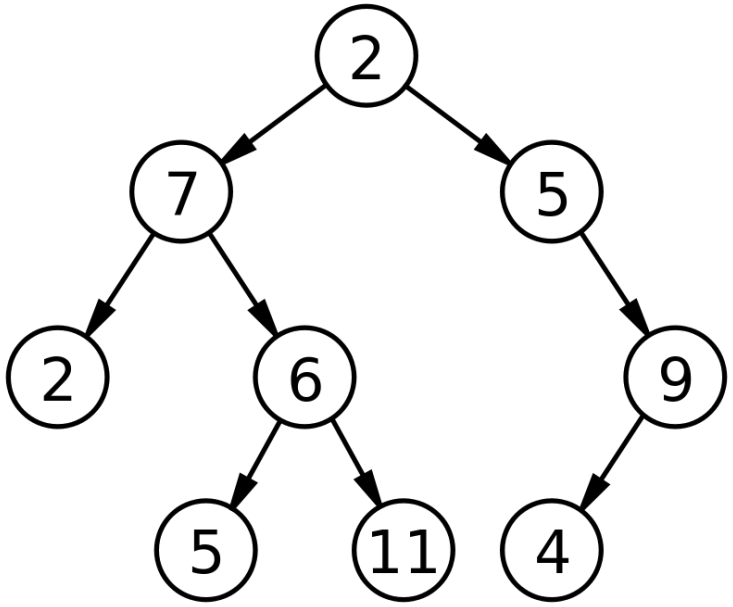
학번	이름	나이	본적	성별	주민등록 번호
1243	홍길동	10	서울	남	yymmdd-1xxxxxx
1332	박영희	19	충청	여	yymmdd-2xxxxxx
1334	이기수	21	전라	남	yymmdd-1xxxxxx
1367	정미영	20	경상	여	yymmdd-2xxxxxx
1377	김철수	20	경기	남	yymmdd-1xxxxxx
1440	최미숙	21	강원	여	yymmdd-2xxxxxx

<학번이 기본Key로 되어 있는 데이터>

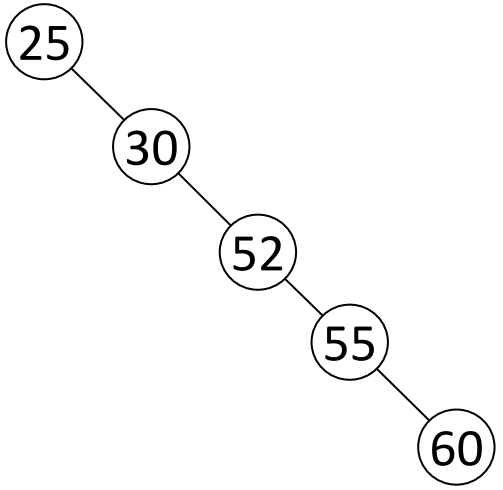
6.2. Binary search tree

Binary search tree (Binary Search Tree)

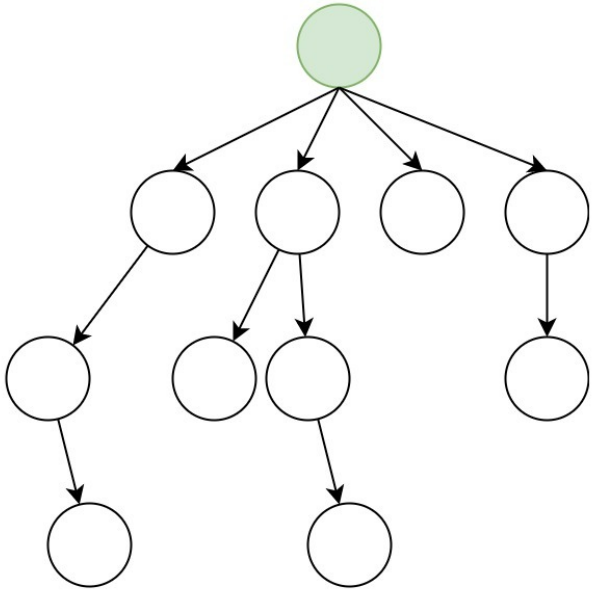
- **Index를 만드는 한가지 방법**
- **Binary tree (이진 트리)**
  - 유한한 수의 Node를 가진 트리.
  - 각각의 Node는 **최대 두개의 자식 Node**를 가진다.



Binary Tree



Binary Tree



Not a binary Tree

General Tree

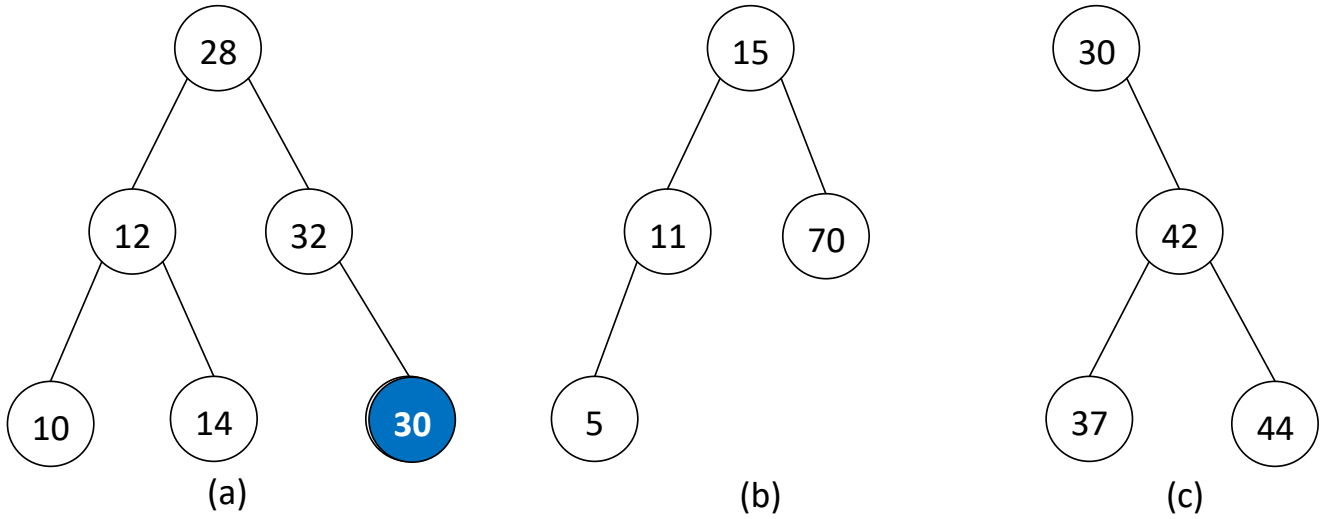
Image source:
https://www.geeksforgeeks.org/difference-between-general-tree-and-binary-tree/

Binary search tree

- **Binary search tree**
  - 1.이진 트리 이다
  - 2.각 Node N_i 는 Record Key K_i와 K_i 를 실제 가지고 있는 Record에 대한 포인터를 포함
  - 3.공백이 아닌 Binary Search Tree는 다음 성질을 만족한다
    - 모든 Node는 상이한 Key 값들을 갖는다.
    - 모든 Node N_i에 대해
      - **N_i의 왼쪽 서브 트리(Left(N_i))에 있는 모든 Node의 Key 값들은 N_i의 Key 값보다 작다.**
      - **N_i의 오른쪽 서브 트리(Right(N_i))에 있는 모든 Node의 Key 값들은 N_i의 Key 값보다 크다.**
    - 왼쪽 서브 트리와 오른쪽 서브 트리는 모두 Binary Search Tree이다.

Binary Tree VS. Binary Search Tree Example

- 그림 (a): Binary Search Tree가 아님
- 그림 (b), (c): Binary Search Tree



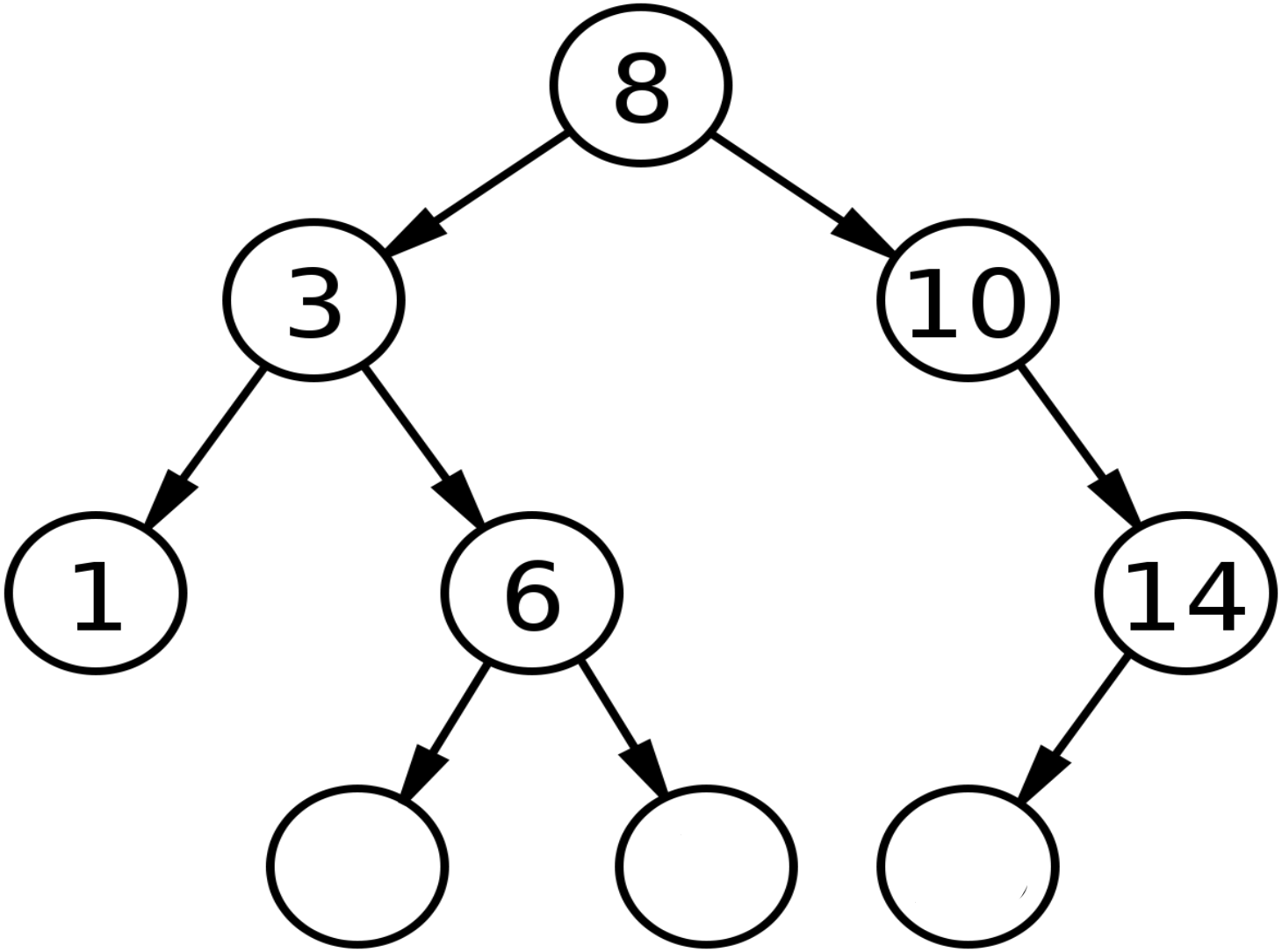
Binary Search Tree에서의 검색

- **Root Node가 N_i인 Binary Search Tree 에서 Key 값이 K인 Node를 검색**
  - **아이디어:**
    - 주어진 Node의 왼쪽 자손들은 해당 Node 보다 모두 Key 값이 작고, 오른쪽 자손들은 해당 Node 보다 Key 값이 크다.
    - 따라서 내가 찾으려는 Key 값이 어느 Node보다 작다면 왼쪽으로 크다면 오른쪽으로 간다.

Binary Search Tree에서의 검색 (Cont'd)

- **용어정리**
  - 어떤 Node i를 나타낼 때는 N_i, N_i 가 포함하는 Key 값을 나타낼 때는 K_i 로 표기한다.
- **검색 알고리즘**
  - If : N_i 가 존재하지 않는다 : 검색 실패
  - Else if : K = K_i : Node N_i 가 원하는 Node => 탐색 종료
  - Else if : K < K_i : N_i 의 왼쪽 서브 트리를 검색, 즉
    - N_i 의 왼쪽 자식을 Root Node로 하여 다시 검색 시작
      - N_i = N_i 의 왼쪽 자식;
  - Else if : K < K_i : N_i 의 오른쪽 서브 트리를 검색 즉
    - N_i 의 오른쪽 자식을 Root Node로 하여 다시 검색 시작
      - N_i = N_i 의 오른쪽 자식;

예제) 아래 Binary Search Tree에서 7이 있다면 그 위치는?



Binary Search Tree의 장점

- Root에서 모든 Leaf까지의 거리가 거의 비슷한, **균형 잡힌** Binary Search Tree (Balanced Binary Search Tree)가 만들어 진다면,
- 트리에 포함된 Key의 개수가 n 개일 때 탐색 비용 (Key 비교 횟수)은 Full scan 의 O(n)에 비해 훨씬 작은 O($\log_2 n$) 이다.

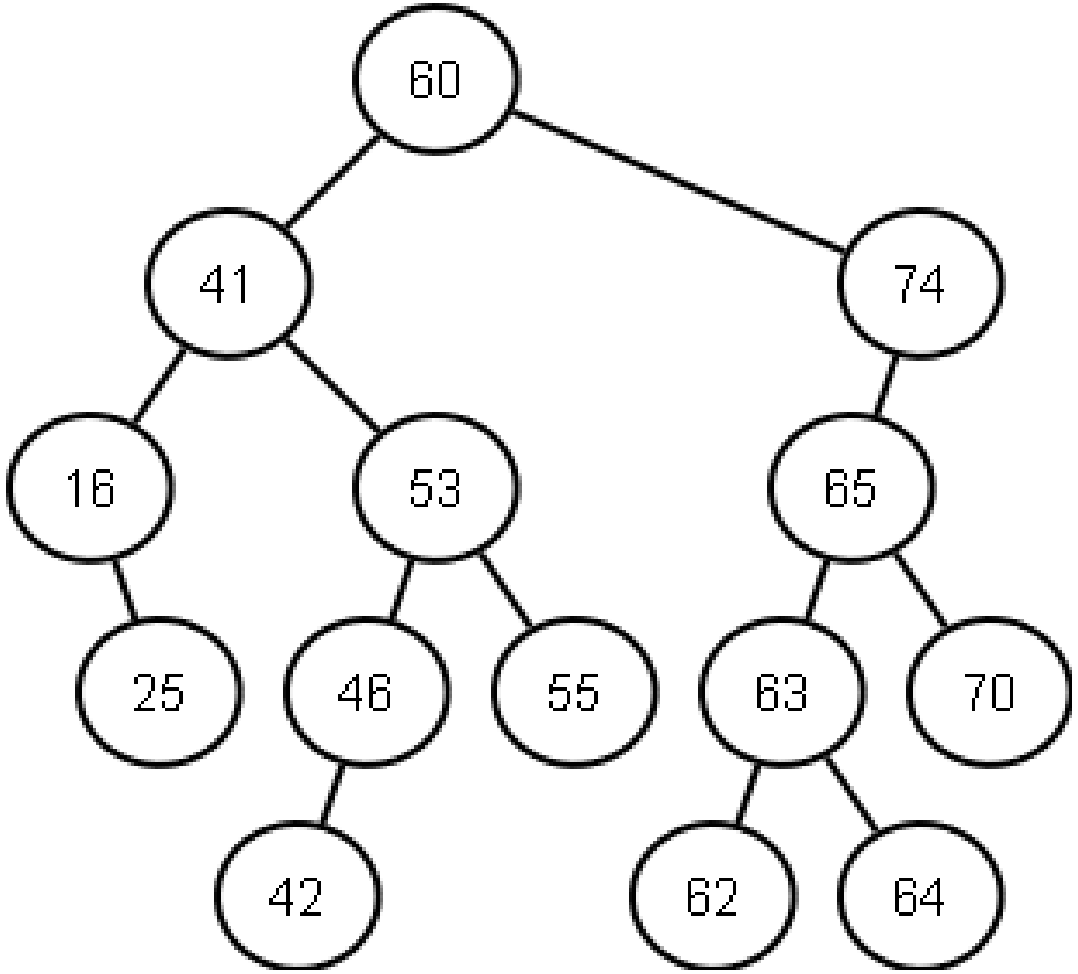
Binary Search Tree에 Key 삽입 (Node 삽입)

- Root Node가 N_i 인 Binary Search Tree에 Key 값이 K인 Node를 삽입
  - **아이디어**
    - **검색과 같은 요령으로 넣으려는 Key 값을 찾아가다가 Node가 없으면 거기가 해당 Key가 있을 곳이다.**

Binary Search Tree에 Key 삽입 (Node 삽입)(Cont'd)

- **용어정리**
  - 어떤 Node i를 나타낼 때는 N_i, N_i 가 포함하는 Key 값을 나타낼 때는 K_i 로 표기한다.
- **알고리즘**
  - If N_i 가 존재하지 않는다 : 해당 위치에 K_i = K 인 N_i 를 만들어 삽입
  - Else if K = K_i : 트리에 똑같은 Key 값이 이미 존재하므로 삽입을 거부
  - Else if K < K_i : N_i 의 왼쪽 서브 트리로 이동하여 계속 탐색. 즉,
    - N_i 의 왼쪽 자식을 Root Node로 하여 다시 탐색 시작
  - Else if K > K_i : N_i 의 오른쪽 서브 트리로 이동하여 계속 탐색. 즉,
    - N_i 의 오른쪽 자식을 Root Node로 하여 다시 탐색 시작

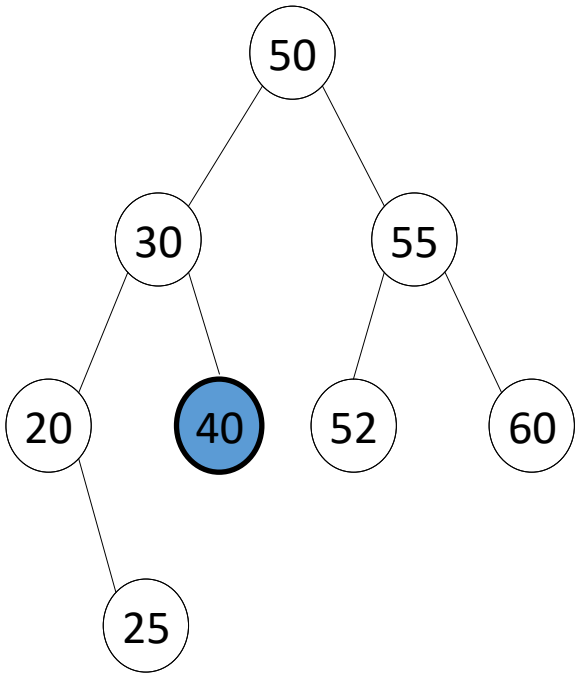
예제) 아래 Binary Search Tree에서 36를 추가 한다면 그 위치는?



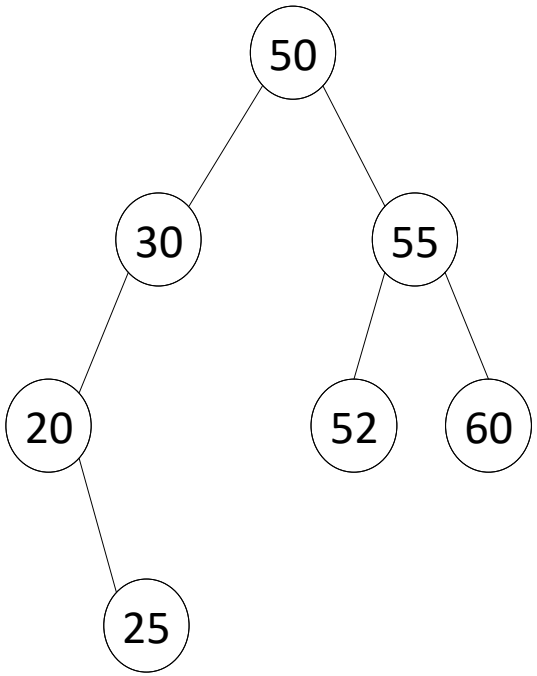
Binary Search Tree에서의 Key삭제 (Node 삭제)

- 삭제하려는 Node는 검색 알고리즘을 사용하여 찾아낸다.
- Node를 삭제한 뒤에도 트리는 계속 Binary Search Tree의 성질을 유지해야 함.
  - 따라서 삭제해야 하는 Node의 자식 상태에 따라 삭제 처리가 달라짐.
- 삭제하려는 Node의 자식 상태에 따른 분류
  - **1. 자식이 없다** = Leaf node.
  - **2. 자식이 하나만 있다** = 왼쪽 혹은 오른쪽 서브 트리 중 하나만 존재한다.
  - **3. 자식이 둘 있다** = 왼쪽과 오른쪽 서브 트리가 모두 존재한다.

Binary Search Tree에서 삭제 예제

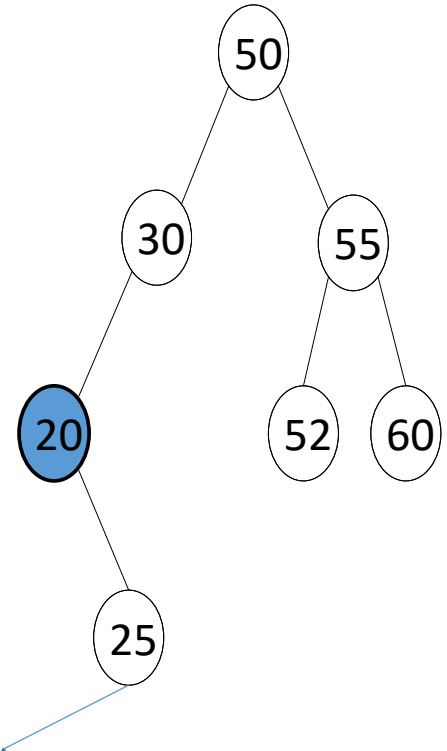


(a) 삭제 전

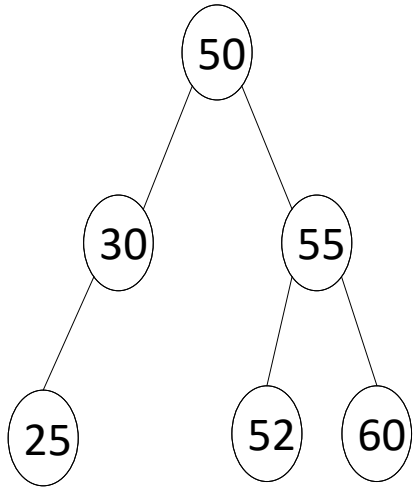


(b) 삭제 후

Binary Search Tree에서 Leaf Node(40)의 삭제



(a)삭제 전

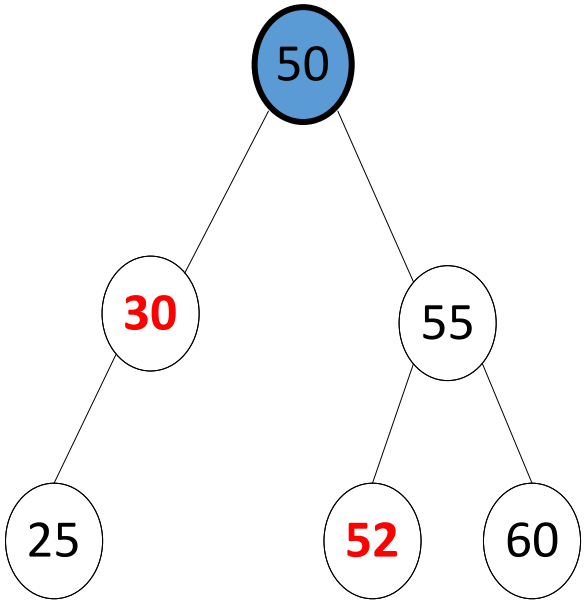


(b)삭제 후

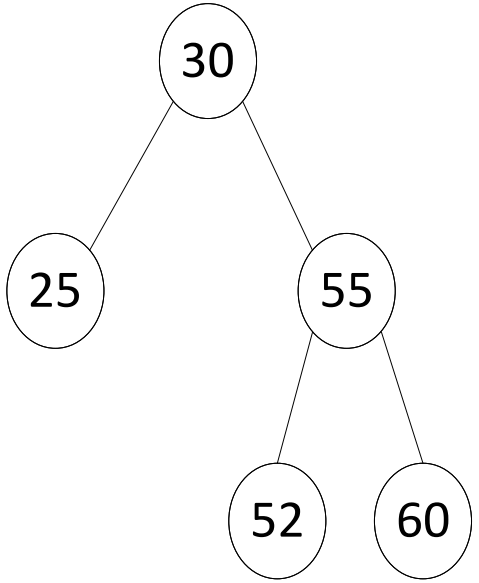
Binary Search Tree에서 자식이 하나인 Node(20)의 삭제
: 자식을 삭제한 Node에 붙인다.

Binary Search Tree에서 삭제 예제 (Cont'd)

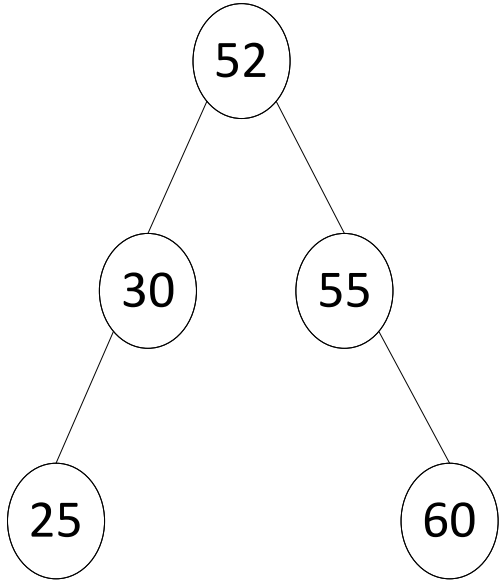
Binary Search Tree에서 자식이 둘인 Node(50)의 삭제



(a) 삭제 전



(b) 왼쪽서브트리의
최대Key 값(30)으로 대체



(c) 오른쪽서브트리의
최소Key 값(52)으로 대체

Binary Search Tree에서의 삭제

- **삭제 알고리즘**
  - 삭제 대상 Node를 N_d 라 할 경우
  - 1. N_d 가 자식이 없는 Leaf Node일 경우
    - 단순히 N_d 을 삭제
  - 2. N_d 가 자식이 하나일 경우
    - N_d 삭제하고 그 자리에 N_d의 자식 Node를 위치 시킨다.
  - 3. N_d 가 자식이 둘인 경우
    - N_d 삭제하고 그 자리에 왼쪽 서브 트리 에서 제일 큰 Key 값을 가지는 Node (**N_d 의 Key 값 K_d 보다 작은 값을 가지는 Node 중에서 제일 큰 Key 값을 가지는 Node)**
    - 또는 오른쪽 서브 트리에서 제일 작은 Key를 가지는 Node (**N_d 의 Key 값 K_d 보다 큰 값을 가지는 Node 중에서 제일 작은 Key 값을 가지는 Node**) 를 가져온다.

Binary Search Tree의 단점

- 편향 Binary Search Tree(skewed binary search tree)가 만들어 질 수 있다
- 편향 Binary Search Tree(skewed binary search tree)
  - 왼쪽이나 오른 쪽 어느 한쪽으로만 자식이 붙어있는 트리
  - Leaf Node의 탐색 시간은 최악
  - N개의 Node로 구성된 Binary Search Tree에서 최악의 탐색 시간 : N번의 Node 탐색

Binary Search Tree의 성능

- 성능
  - Binary Search Tree의 성능은 트리의 형태와 Node에 대한 접근 빈도에 의존
  - 우수한 성능을 위해서는
    1. 가장 자주 접근되는 Node는 Root에 가장 가깝게 유지
    2. Binary Search Tree를 균형 트리(balanced tree)로 유지
       - 모든 Node에 대해 양쪽 서브 트리의 Node 수가 가능한 똑같게 만들어 트리의 최대 경로 길이를 최소화
- Binary Search Tree의 단점
  - 삽입, 삭제 후 효율적 접근을 위한 균형 트리 유지 부담이 큼
  - 작은 분기율(branching factor)에 따른 긴 탐색 경로와 검색 시간
    - 분기율이 2: 각 Node는 많아야 두 개의 서브 트리를 가짐
    - n개의 Node를 갖는 트리의 최소 높이 : $\lfloor \log_2 n \rfloor$

6.3. AVL Tree

AVL Tree

- 1962년 러시아 수학자 Adelson-Velskii와 Landis가 고안한 높이 균형 이진 트리 (height-balanced binary tree)

- **Binary Tree의 높이:** Root에서 Leaf 까지의 경로 (Path) 중 가장 긴 경로의 길이

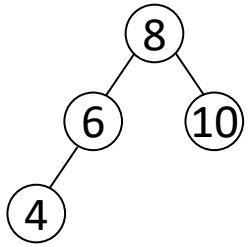
- **AVL 높이 균형 Binary Tree**
  - 삽입, 삭제, 검색 시간 : $O(\log N)$
  - 트리의 일부만 재균형시키면서 트리 전체가 균형을 계속 유지할 수 있도록 함.

AVL Tree의 정의

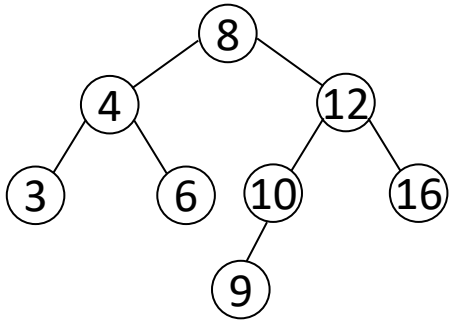
- **AVL Tree의 정의**
  - **각 Node의 왼쪽 Sub-Tree의 높이와 오른쪽 Sub-Tree 의 높이 차이가 1 이하**인 Binary Search Tree
    - $|\text{height}(\text{LeftSubTree}(N_i)) - \text{height}(\text{RightSubTree}(N_i))| \leq 1$, $N_i \in$ {Tree의 Nodes}
  - 공백 서브 트리의 높이 (= 서브 트리가 없다면): -1

- **Balance Factor (균형인수, BF)**
  - 왼쪽 Sub-Tree의 높이에서 오른쪽 Sub-Tree 의 높이를 뺀 수,
  - BF = $\text{height}(\text{LeftSubTree}(N_i)) - \text{height}(\text{LeftSubTree}(N_i))$
  - Node의 Balance Factor가
    - ±1 이하이면 이 Node는 **AVL 성질**(AVL property)을 만족하고
    - ±2 이상이면 AVL 성질을 만족하지 못한다고 말함.
  - AVL 트리의 **모든 Node는 Balance Factor 가 -1 이상 이고 1이하**이다.
    - 즉 AVL 트리의 모든 Node는 이 AVL 성질을 만족하고 있다.

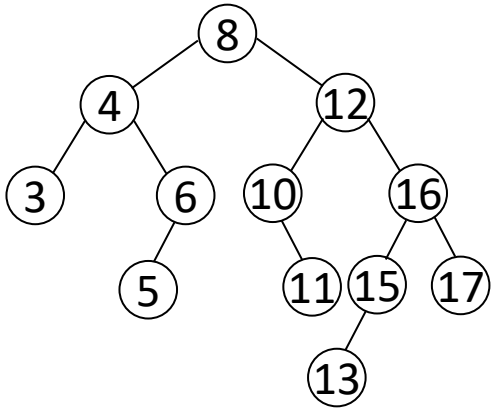
AVL Tree VS Non-AVL Tree



(a)



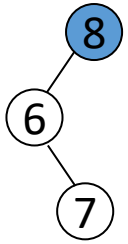
(b)



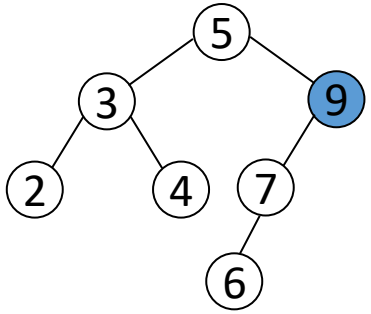
(c)

AVL 트리

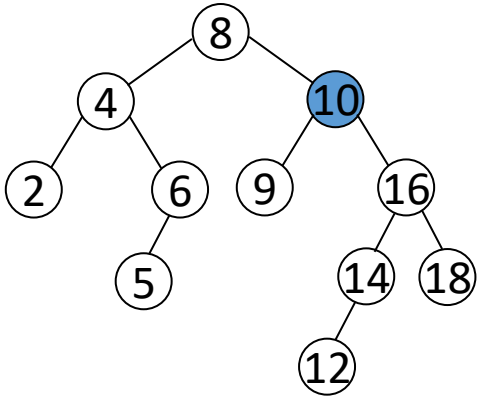
AVL 트리란? **모든 Node**의 왼쪽 서브 트리의 높이와 오른쪽 서브 트리의 높이 차이가 1 이하인 Binary Search Tree



(a)



(b)

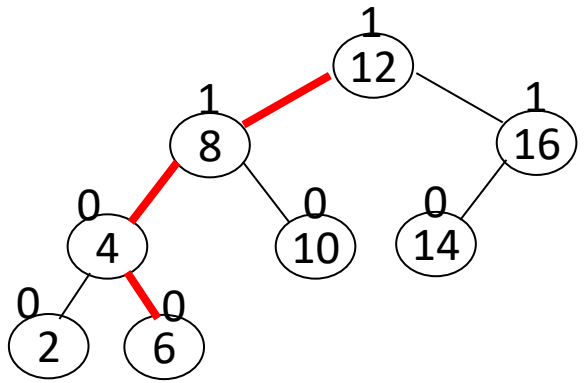


(c)

non-AVL 트리

AVL Tree 에서의 검색

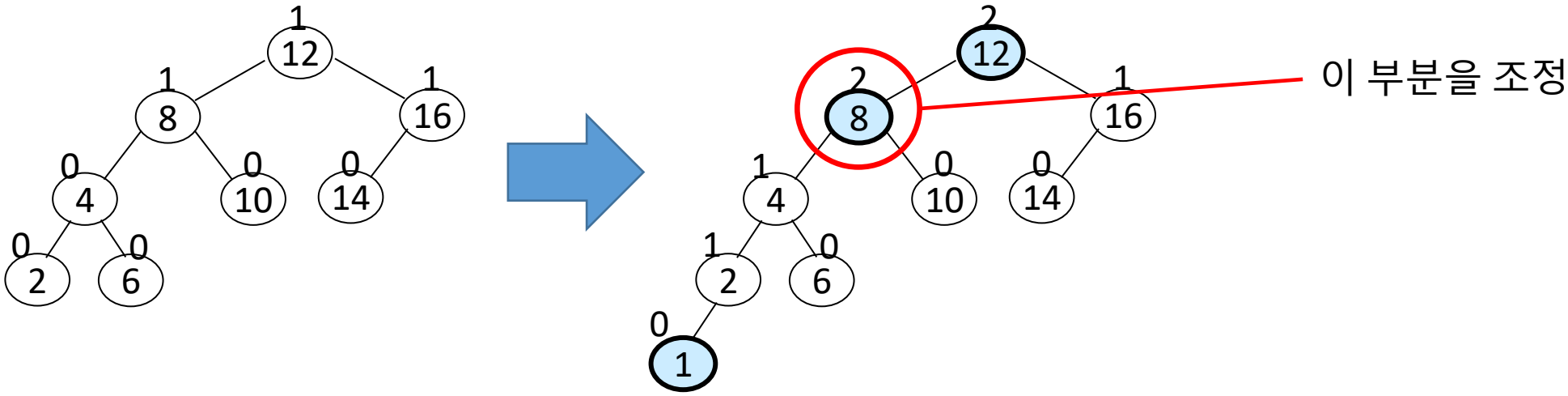
- **검색**
  - 일반 Binary Search Tree의 검색 방법과 동일
  - 시간 복잡도 : $O(\log_2 N)$



(a) AVL 트리 에서 6을 찾아가는 Path

AVL Tree 에서의 삽입

- **삽입**
  - Node가 삽입되면, 삽입되는 Node에서부터 Root까지의 경로에 있는 조상 Node들의 Balance Factor(bf)에 영향을 줄 수 있음
  - Node 삽입으로 AVL 트리가 non-AVL Tree가 되면 삽입된 Node와 가장 가까우면서 불균형이 된 조상 Node의 Balance Factor 의 절대값이 1 이하로 되게 Tree 구조를 조정해야 됨

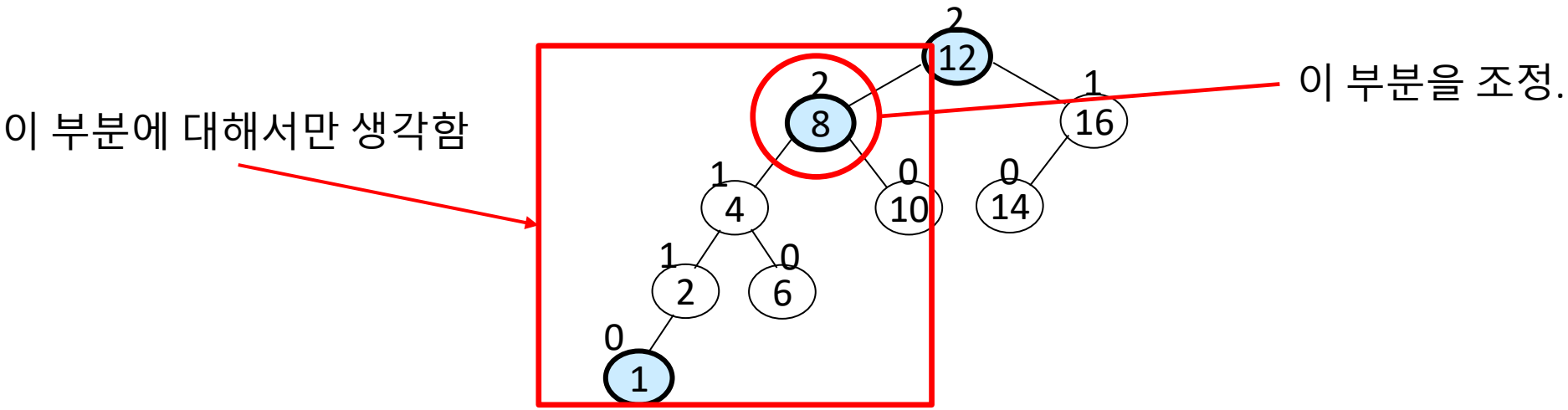


(a) AVL 트리 (b) 원소 1의 삽입으로 non-AVL 트리로 변환

원소 1의 삽입으로 Node 8의 AVL 성질 상실

AVL Tree에서의 삽입

- 왜 삽입된 Node와 가장 가까우면서 불균형이 된 조상 Node의 Balance Factor의 절대값이 1 이하가 되게 하는 것으로 AVL 트리가 되는가?
  - 더 먼 조상 Node는 안 고쳐도 되는가?



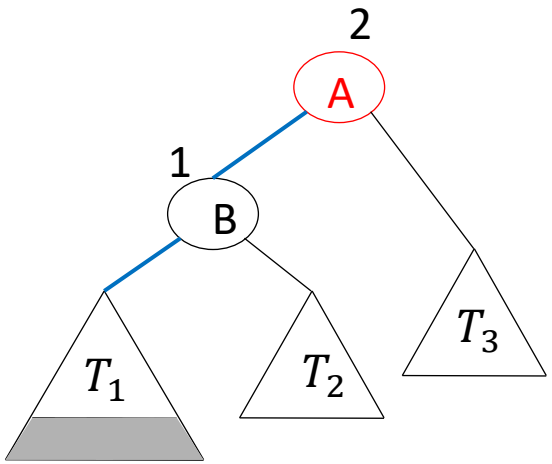
(b) 원소 1의 삽입으로 non-AVL 트리로 변환

원소 1의 삽입으로 Node 8이 AVL Tree성질 상실

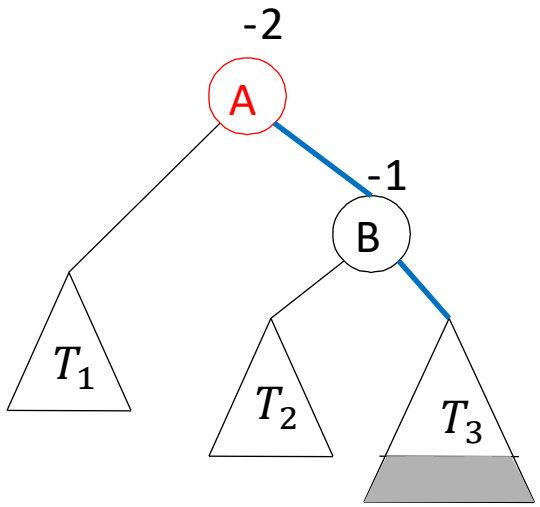
AVL Tree에서의 삽입

- Node 삽입으로 Balance Factor의 절대값이 2로 된 Node x가 출현
  - 다음 4가지 경우 중 하나로 인해 발생(**Node x는 Balance Factor의 절대값이 2**)

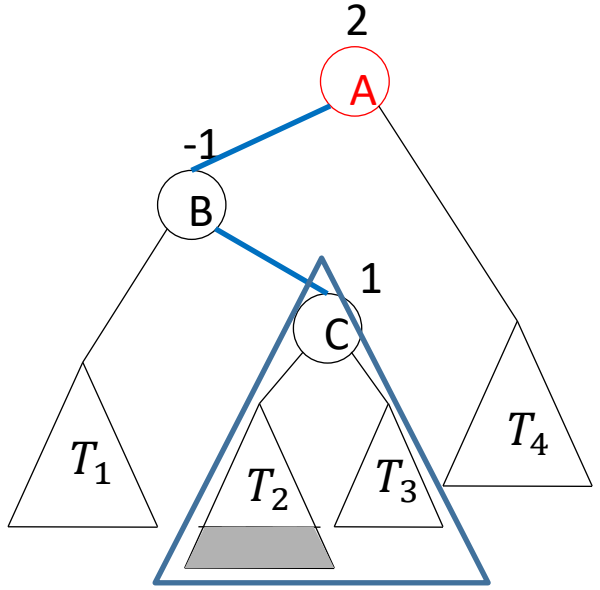
LL : x의 왼쪽 자식(L)의 왼쪽 서브 트리 (L) 에 삽입
RR : x의 오른쪽 자식(R)의 오른쪽 서브 트리 (R) 에 삽입
LR : x의 왼쪽 자식 (L) 의 오른쪽 서브 트리 (R) 에 삽입
RL : x의 오른쪽 자식(R)의 왼쪽 서브 트리 (L) 에 삽입



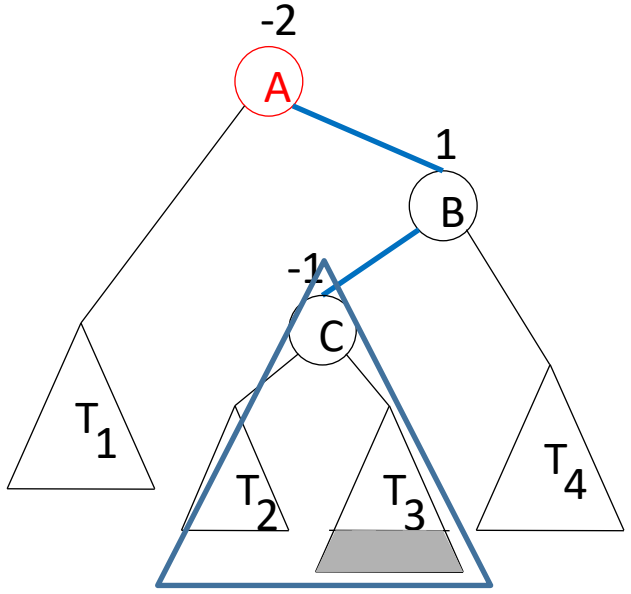
LL 삽입



RR 삽입



LR 삽입



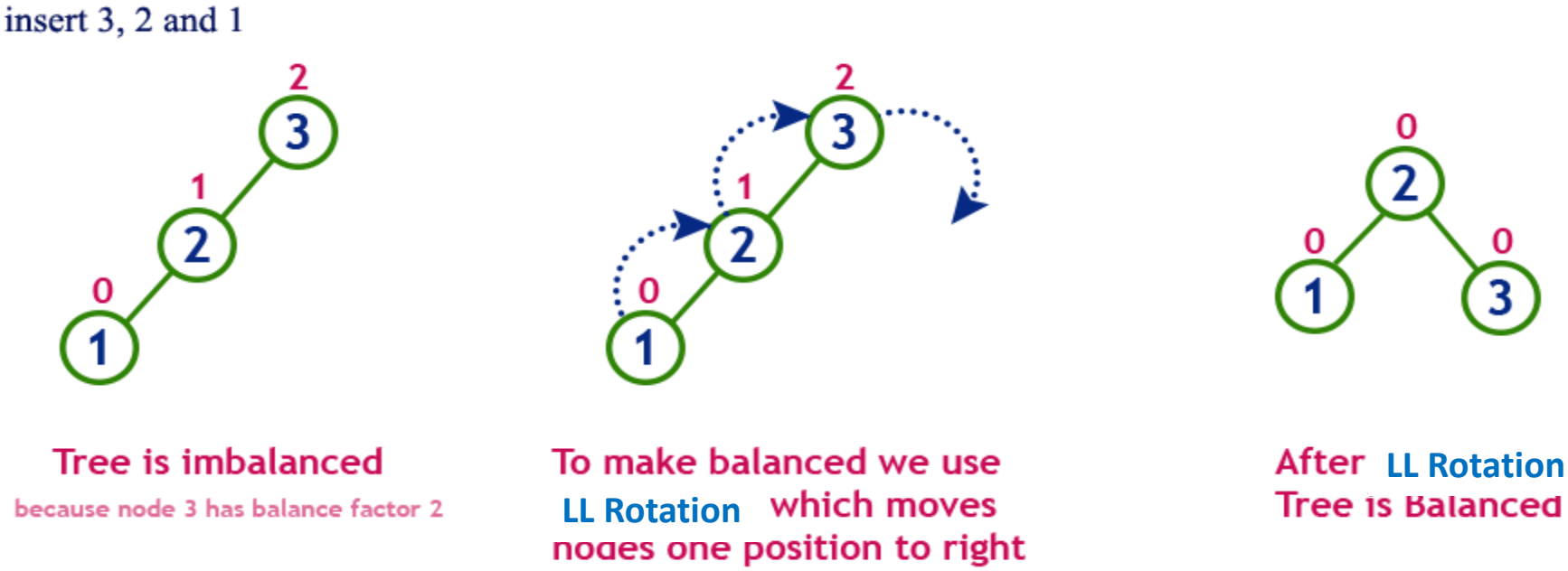
RL 삽입

AVL Tree에서의 삽입 : 회전 (Rotation)

- 회전(rotation): 불균형이 발생할 때 Tree 구조를 변경하여 균형을 잡아주는 것
  - **단일 회전(single rotation 혹은 단순 회전)**
    - 균형잡기 위해 한 번의 회전만 필요: LL, RR
    - 탐색 순서를 유지 하면서 부모와 자식 원소의 위치를 교환
  - **이중 회전(double rotation)**
    - 균형잡기 위해 두 번의 회전이 필요: LR, RL

AVL Tree에서의 삽입 : LL 회전 (Rotation)

- **LL Rotation case 1: 왼쪽 자식이 왼쪽 Sub-Tree만 가지고 있을 경우**

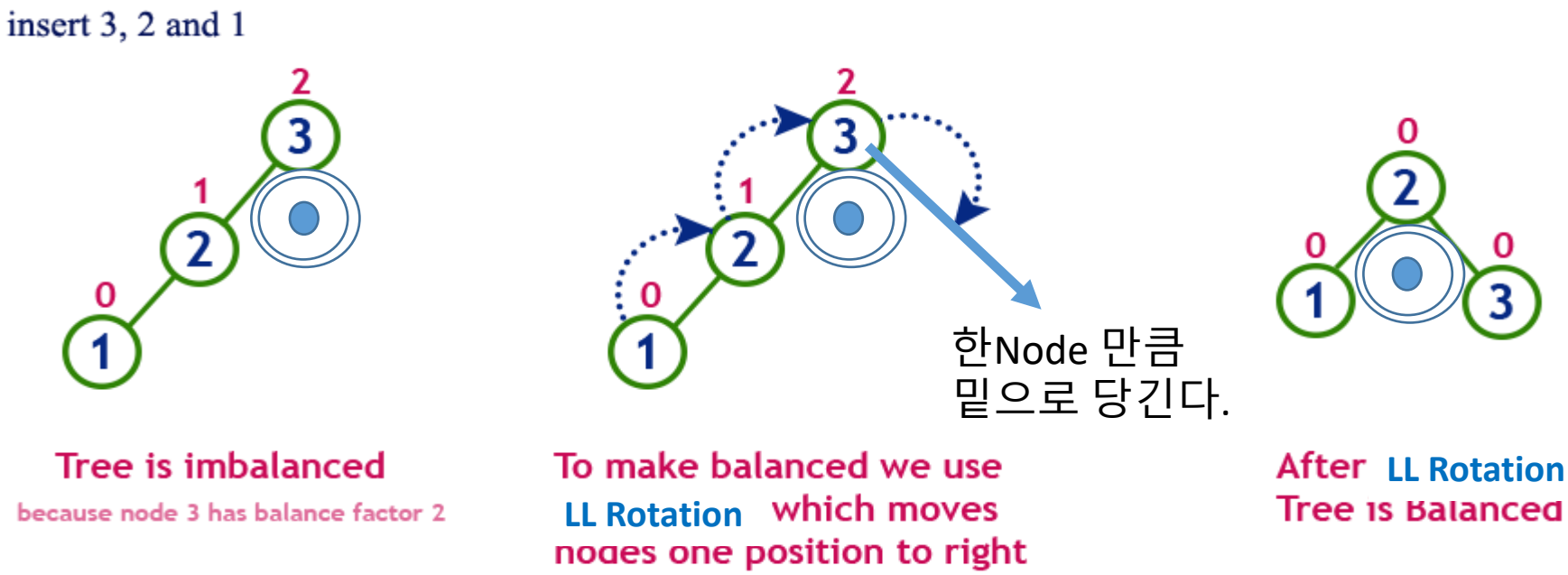


- 왼쪽 자식이 부모보다 작으므로 자식 (2)이 부모로 올라가고 원래 부모(3)가 올라간 자식의 오른쪽 자식이 되어도 Binary Search Tree의 정의를 만족한다.
- 이름과 실제 회전 방향을 다르다. (이름은 LL회전이나 실제 회전 방향은 오른쪽 (시계방향))

Image source : http://www.btechsmartclass.com/data_structures/avl-trees.html (2020.09 accessed)

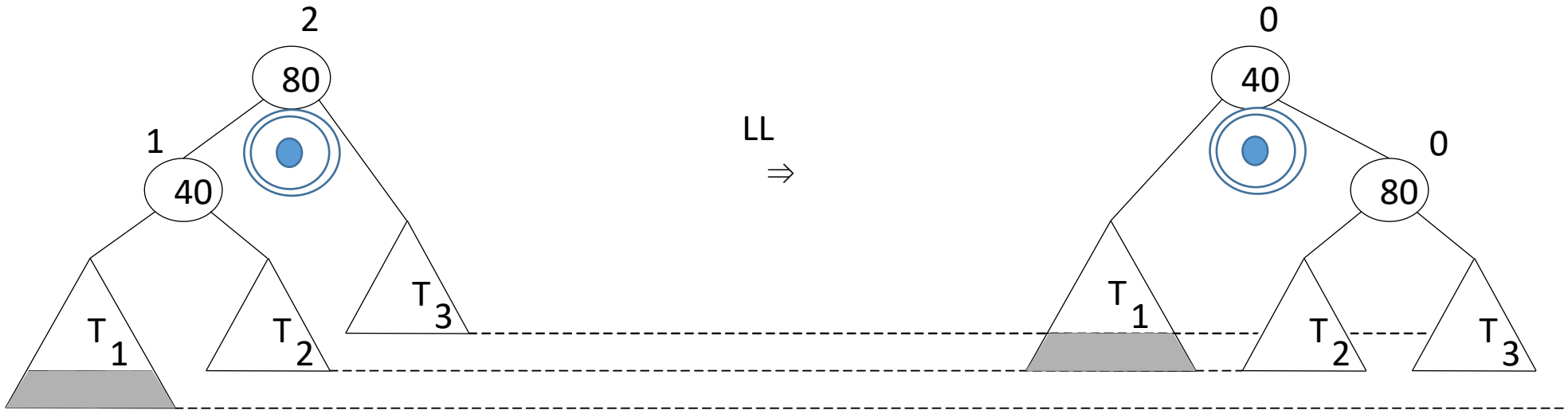
AVL Tree에서의 삽입 : LL 회전 (Rotation)

회전 개념: 각각의 Node가 끈(Edge)으로 연결되어 있다고 가정하고,
회전축이 되는 Node 밑에 도르레를 설치하고 한쪽 끝을 당기는 개념.



AVL Tree에서의 삽입 : LL 회전 (Rotation)

- **LL Rotation case 2 : 왼쪽 자식이 Sub-Tree를 양쪽 다 가질 때**



40이 부모가 되면 40은 T1, T2, 80 (원래 부모)를 자식으로 가지게 되어 Binary Search Tree 가 아니게 된다.

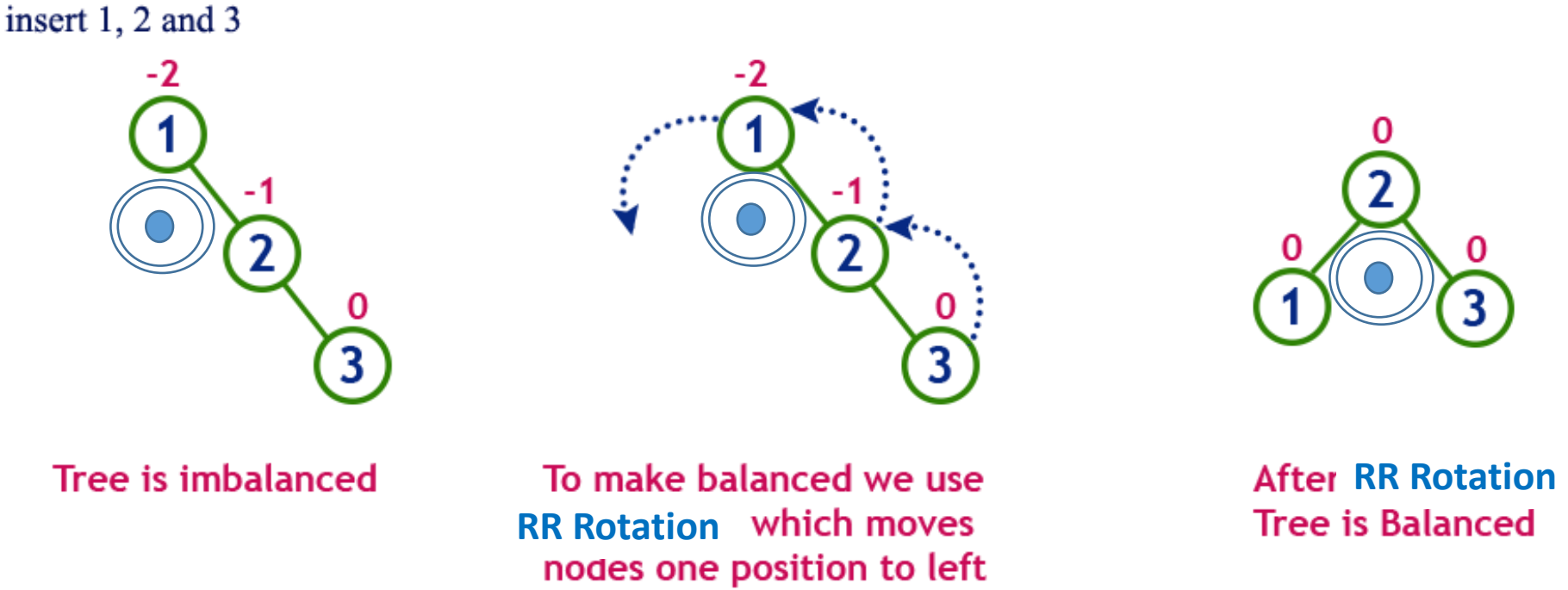
⇒ 40의 오른쪽 Subtree를 80의 왼쪽 Subtree로 만든다.

⇒ 40의 오른쪽 Subtree의 Node 값들은 40보다 크고, 80 보다 작기 때문에 80의 왼쪽 Subtree로 되어야 Binary Search Tree의 요건을 만족시킨다.

⇒ 40가 부모가 되는 순간 80의 왼쪽 Subtree는 존재하지 않기 때문에 붙일 곳에 이미 다른 Node가 존재하는 경우는 없다.

AVL Tree에서의 삽입 : RR 회전 (Rotation)

- **RR Rotation case 1 : 오른쪽 자식이 오른쪽 Sub-Tree만 가지고 있을 경우**

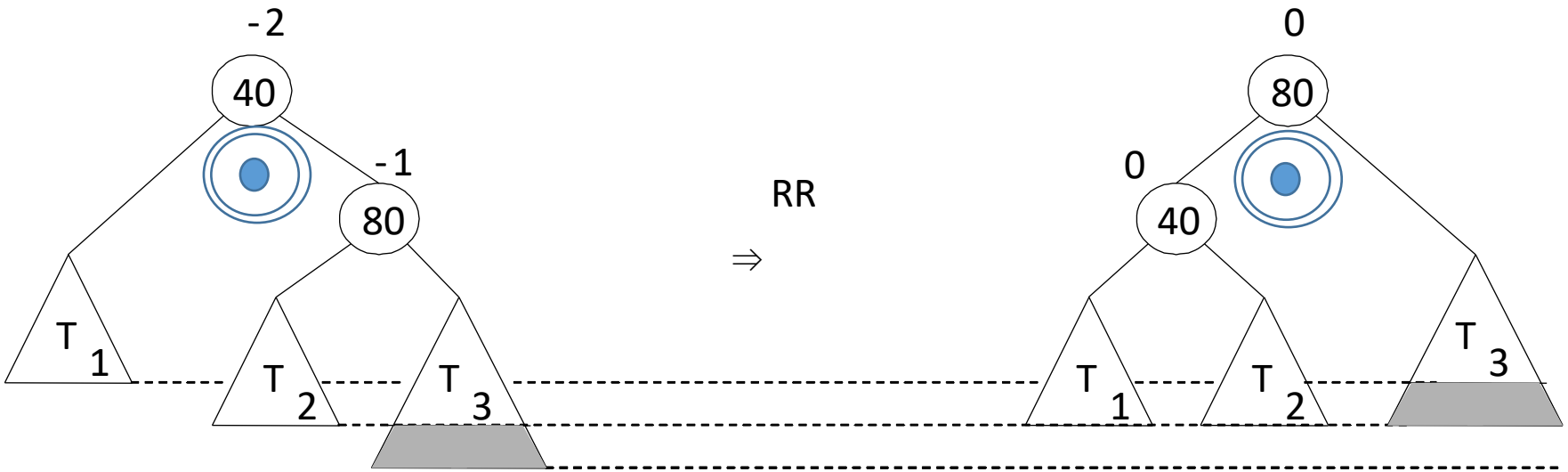


- 오른쪽 자식이 부모보다 크므로 자식 (2)이 부모로 올라가고 원래 부모(1)가 올라간 자식의 왼쪽 자식이 되어도 Binary Search Tree의 정의를 만족한다.
- 이름과 실제 회전 방향을 다르다. (이름은 RR회전이나 실제 회전 방향은 왼쪽 (반시계방향))

Image source : http://www.btechsmartclass.com/data_structures/avl-trees.html (2020.09 accessed)

AVL Tree에서의 삽입 : RR 회전 (Rotation)

- **RR Rotation case 2 : 오른쪽 자식이 Sub-Tree를 양쪽 다 가질 때**



80이 부모가 되면 80은 40 (원래 부모) , T2, T3, 를 자식으로 가지게 되어 Binary Search Tree 가 아니게 된다.

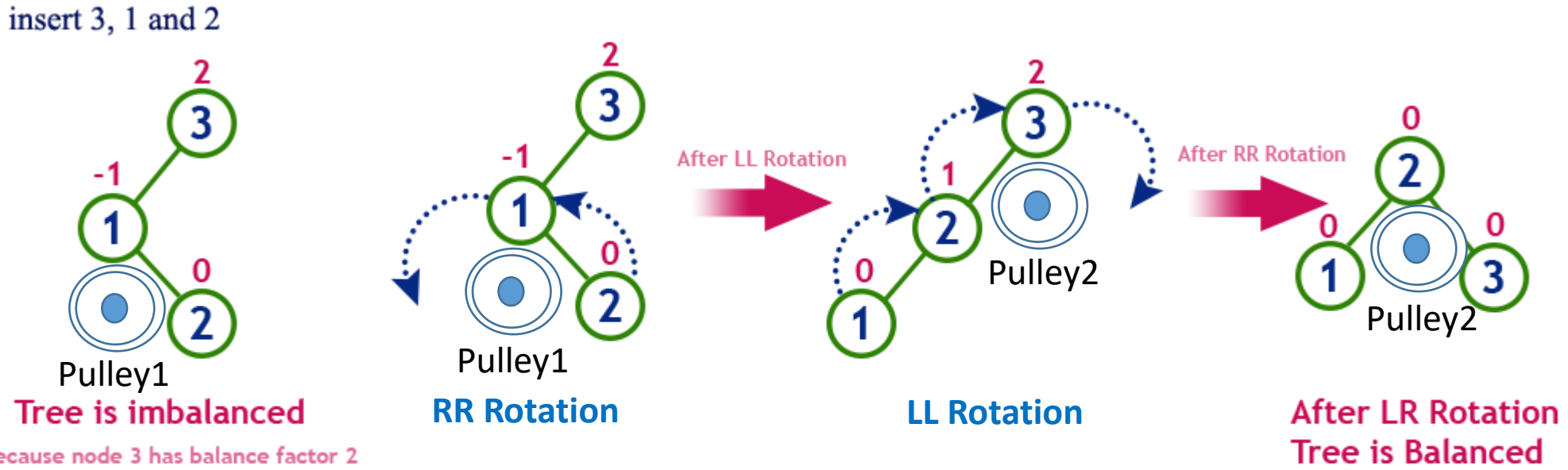
⇒ 80의 왼쪽 Subtree를 40의 오른쪽 Subtree로 만든다.

⇒ 80의 왼쪽 subtree의 Node 값들은 80보다 작고, 40보다 크기 때문에 40의 오른쪽 Subtree로 되어야 Binary Search Tree의 요건을 만족시킨다.

⇒ 80이 부모가 되는 순간 40의 오른쪽 Subtree는 존재하지 않기 때문에 붙일 곳에 이미 다른 Node가 존재하는 경우는 없다.

AVL Tree에서의 삽입 : LR 회전 (Rotation)

- **LR Rotation case 1 : 왼쪽 자식이 오른쪽 Sub-Tree만 가질 때**



Node 1 이 LL회전으로 부모가 되었다고 생각해 보자.
이렇게 되면 왼쪽 자식 Node 2가 부모인 Node 1 보다 커진다.
(Binary search tree가 아니게 됨)

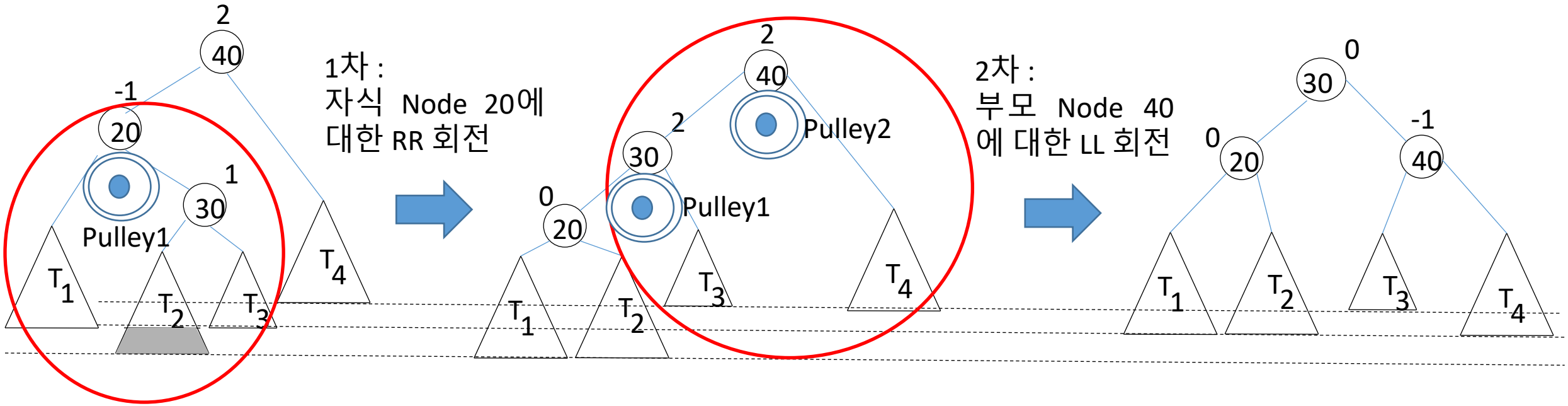
먼저 Node 1, Node 2에 대해 RR회전 시켜 Node 2를 부모, Node 1을 왼쪽 자식으로 만든다. (**자식에 대해 RR회전**)

이 상태에서 LL회전 시Key면 부모가 Node 2 왼쪽 자식이 Node1, 오른쪽 자식이 Node 3 이 되어 Binary search tree 가 되면서 BF 도 1 이하가 된다.
(**BF 깨어진 본인에 대해 LL회전**)

Image source : http://www.btechsmartclass.com/data_structures/avl-trees.html (2020.09 accessed)

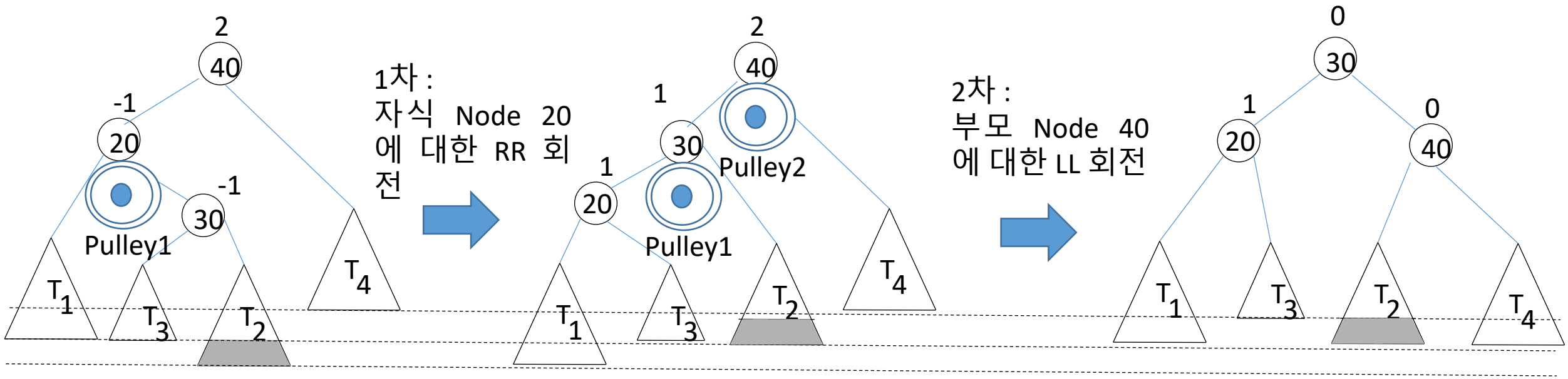
AVL Tree에서의 삽입 : LR 회전 (Rotation)

- **LR Rotation case 2 : 왼쪽 자식이 양쪽 Sub-Tree 모두 가지고 있는 상태에서 손자의 왼쪽 Sub-Tree에 새 데이터가 삽입되었을 때**



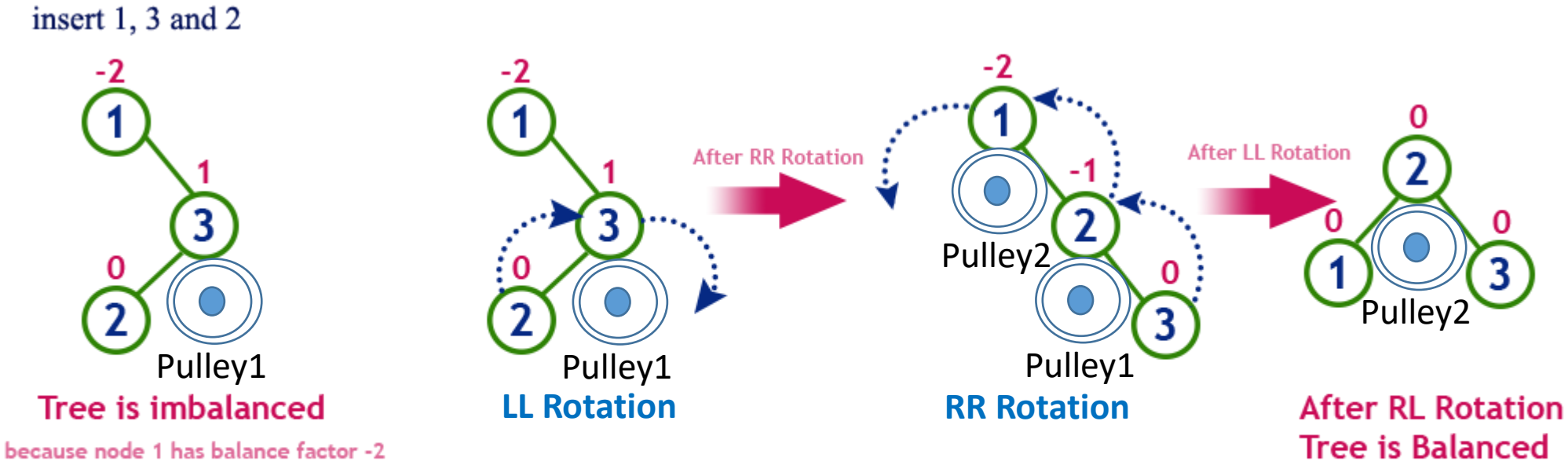
AVL Tree에서의 삽입 : LR 회전 (Rotation)

- **LR Rotation case 3 : 왼쪽 자식이 양쪽 Sub-Tree 모두 가지고 있는 상태에서 손자의 오른쪽 Sub-Tree에 새 데이터가 삽입되었을 때**



AVL Tree에서의 삽입 : RL 회전 (Rotation)

- **RL Rotation case 1 : 오른쪽 자식이 왼쪽 Sub-Tree만 가질 때**



Node 3 이 RR회전으로 부모가 되었다고 생각해 보자.
이렇게 되면 오른쪽 자식 Node 2가 부모인 Node 3 보다 커진다.
(Binary search tree가 아니게 됨)

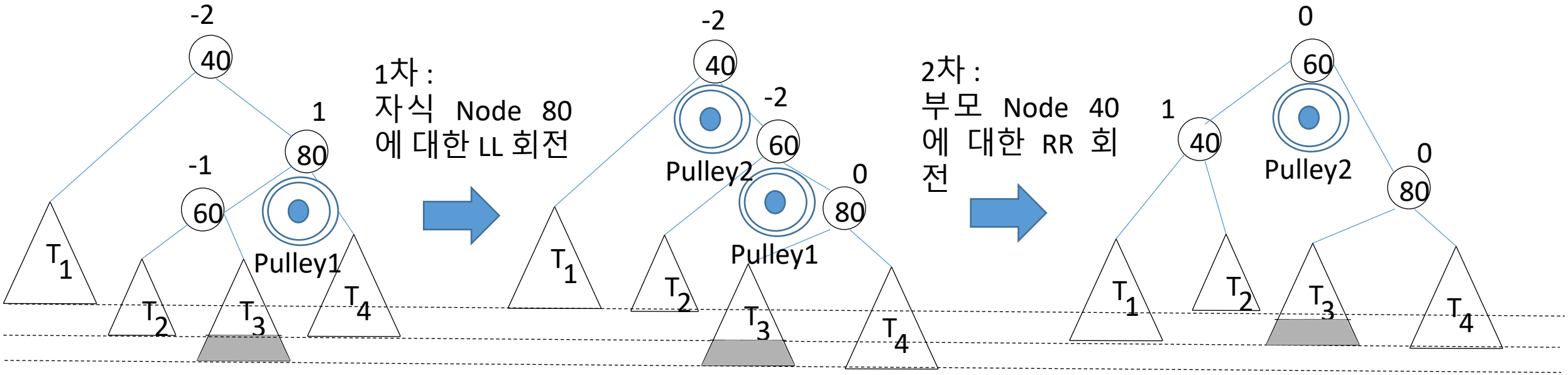
먼저 Node 2, Node 3에 대해 LL회전 시켜 Node 2를 부모, Node 3을 오른쪽 자식으로 만든다. (**자식에 대해 LL회전**)

이 상태에서 RR회전 시Key면 부모가 Node 2 왼쪽 자식이 Node1, 오른쪽 자식이 Node 3 이 되어 Binary search tree 가 되면서 BF 도 1 이하가 된다.
(**BF 깨어진 본인에 대해 RR회전**)

Image source : http://www.btechsmartclass.com/data_structures/avl-trees.html (2020.09 accessed)

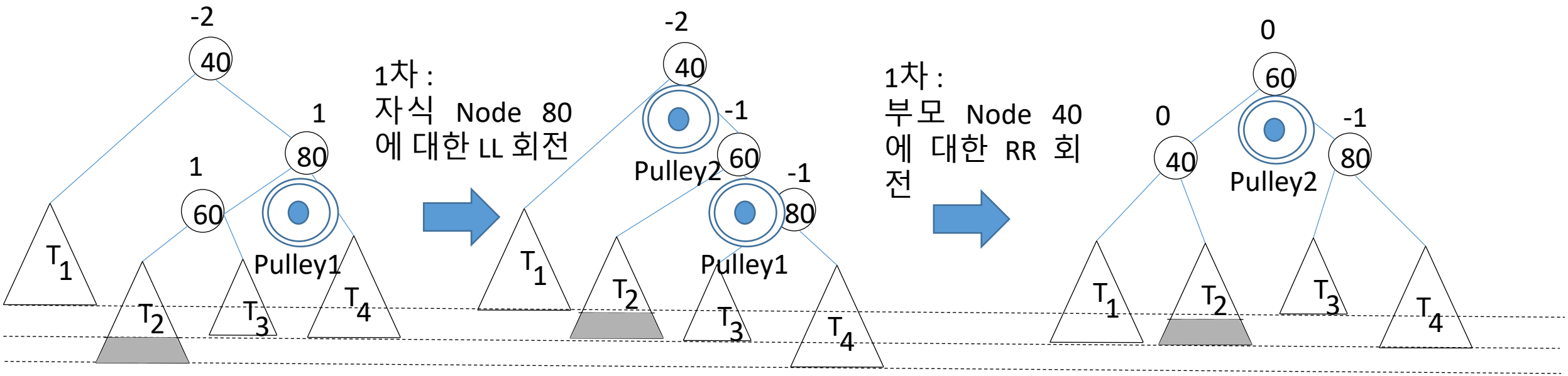
AVL Tree에서의 삽입 : RL 회전 (Rotation)

- **RL Rotation case 2 : 오른쪽 자식이 양쪽 Sub-Tree 모두 가지고 있는 상태에서 손자의 오른쪽 Sub-Tree에 새 데이터가 삽입되었을 때**



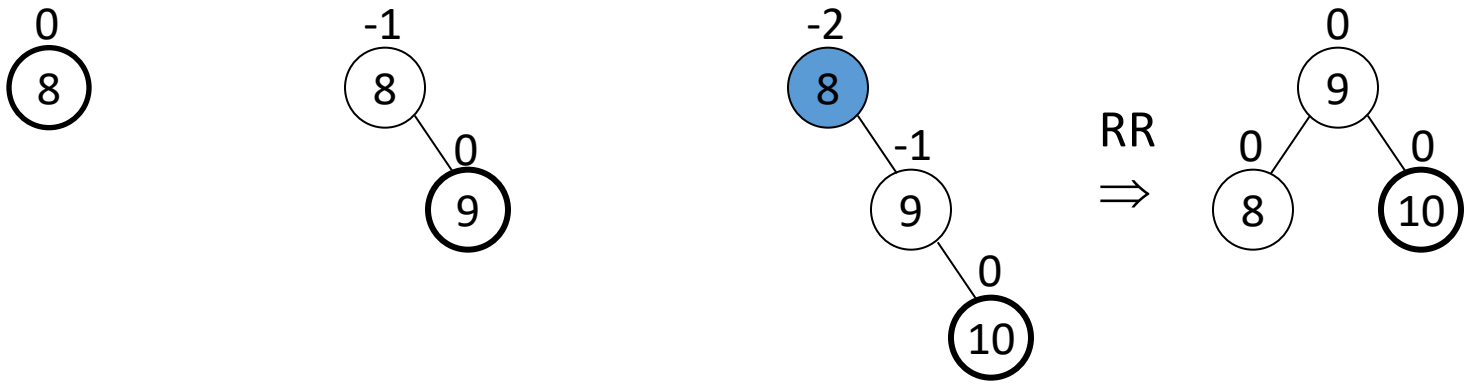
AVL Tree에서의 삽입 : RL 회전 (Rotation)

- **RL Rotation case 3 : 오른쪽 자식이 양쪽 Sub-Tree 모두 가지고 있는 상태에서 손자의 왼쪽 Sub-Tree에 새 데이터가 삽입되었을 때**

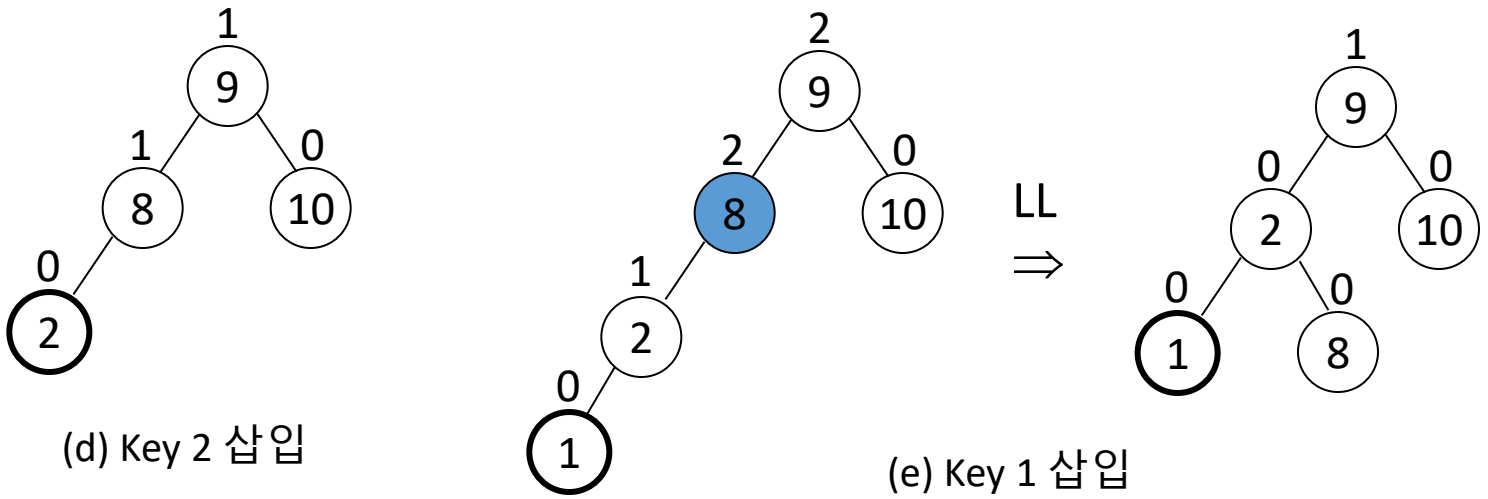


AVL Tree에서의 삽입 예

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

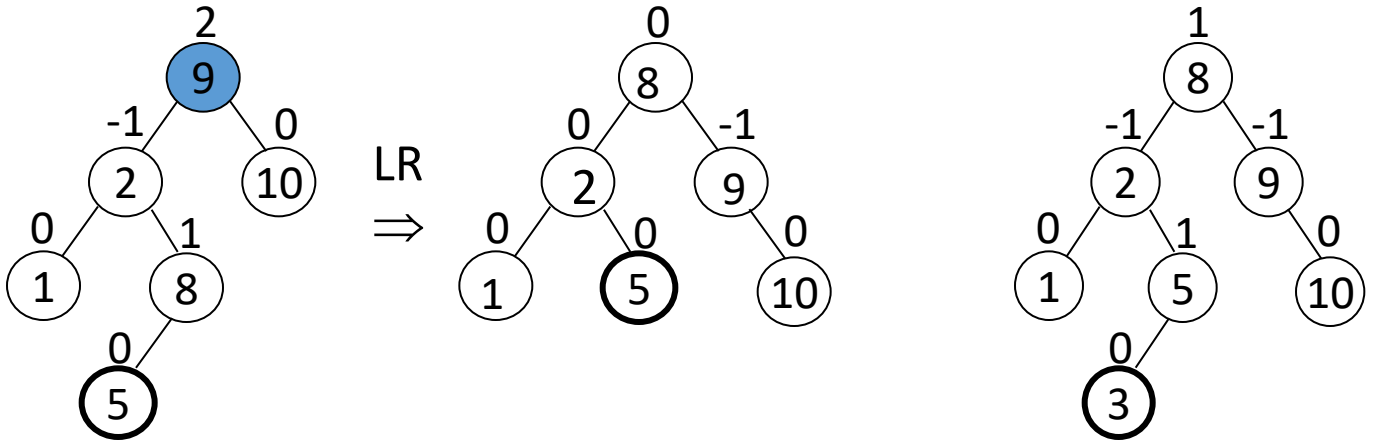


(a) Key 8 삽입 (b) Key 9 삽입 (c) Key 10 삽입



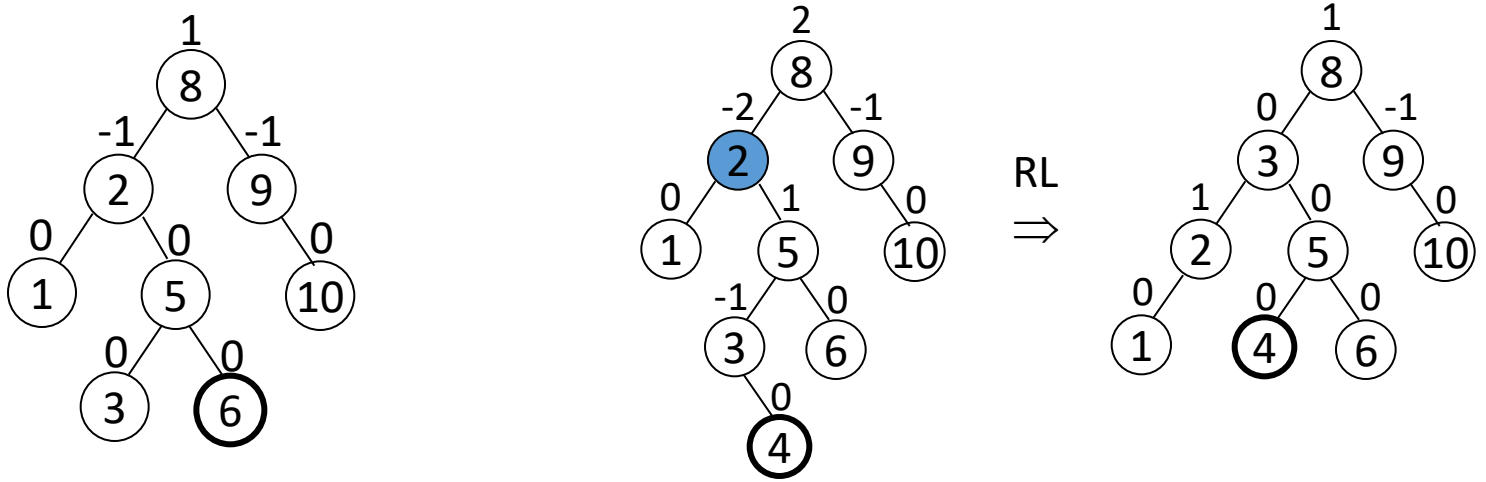
(d) Key 2 삽입 (e) Key 1 삽입

AVL Tree에서의 삽입 예 (Cont'd)



(f) Key 5 삽입

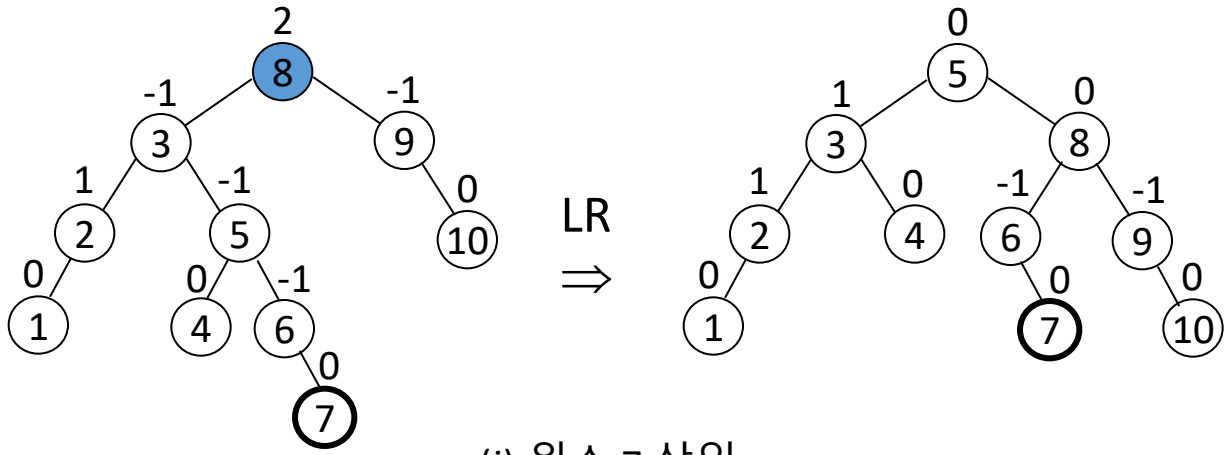
(g) Key 3 삽입



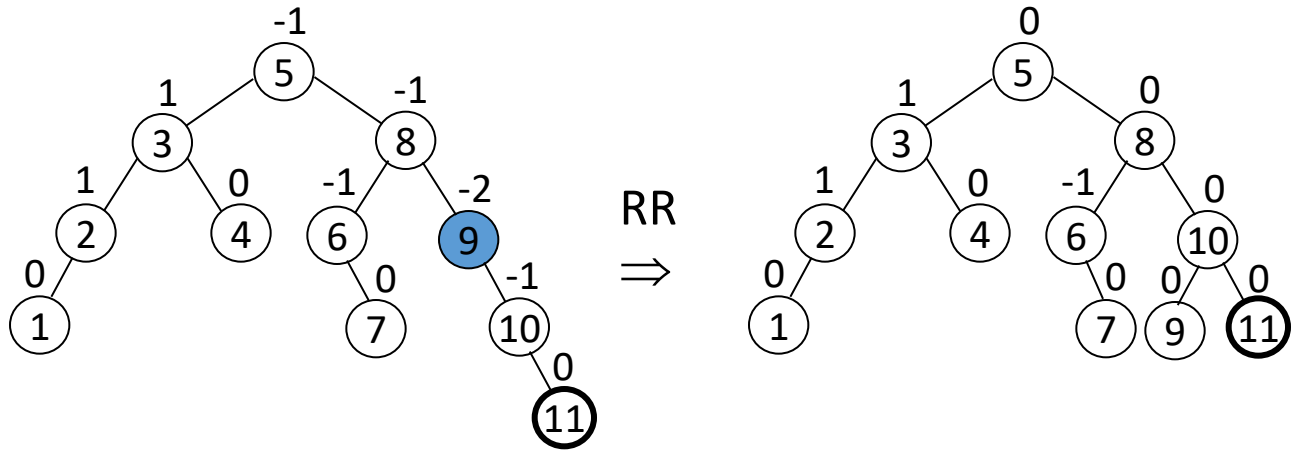
(h) Key 6 삽입

(i) Key 4 삽입

AVL Tree에서의 삽입 예 (Cont'd)

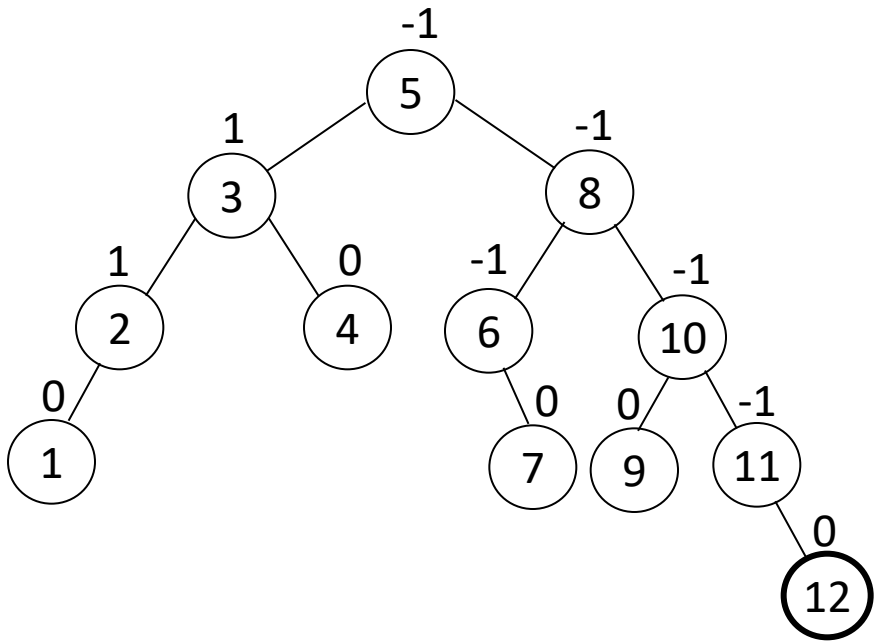


(j) 원소 7 삽입



(k) 원소 11 삽입

AVL Tree에서의 삽입 예 (Cont'd)



(l) Key 12 삽입

AVL Tree의 삭제

- **삭제 연산 알고리즘**
- **Phase 1. 일단 삭제 (Binary Search Tree와 같은 알고리즘으로 삭제)**
  - 삭제 대상 Node를 N_d 라 할 경우
  - 1. N_d 가 자식이 없는 Leaf Node일 경우
    - 단순히 N_d 을 삭제
  - 2. N_d 가 자식이 하나일 경우
    - N_d 삭제하고 그 자리에 N_d의 자식 Node를 위치 시킨다.
  - 3. N_d 가 자식이 둘인 경우
    - N_d 삭제하고 그 자리에 왼쪽 서브 트리 에서 제일 큰 Key 값을 가지는 Node (**N_d 의 Key 값 K_d 보다 작은 값을 가지는 Node 중에서 제일 큰 Key 값을 가지는 Node)**
    - 또는 오른쪽 서브 트리에서 제일 작은 Key를 가지는 Node (**N_d 의 Key 값 K_d 보다 큰 값을 가지는 Node 중에서 제일 작은 Key 값을 가지는 Node**) 를 가져온다.

AVL Tree의 삭제 (Cont'd)

- 삭제된 Node 위치 부터 시작해서 Root까지 가면서 처음으로 AVL 트리의 정의를 만족하지 않는 node를 찾는다 (BF 의 절대 값이 1 보다 큰 Node)
- 해당 Node에 대해 LL, RR, LR, RL중 하나의 연산을 수행한다.
  - 자식들과, 각 자식의 왼쪽, 오른쪽 subtree의 BF로 LL, RR, LR, RL을 판정한다.

AVL Tree의 높이

- **AVL Tree 의 높이**
  - N개의 Node를 가진 **AVL Tree** 는 완전 균형 이진 트리보다 45% 이상 높아지지 않음
  - $\log(N+1) \le h \le 1.4404\log(N+2)-0.328$

- **AVL 트리 VS 완전 균형 Binary Tree**
  - $O(1.4 \log N)$ 대 $O(\log N)$
  - AVL 트리의 탐색 시간이 더 길다
    - 이유 : 트리의 전체 재균형을 수행하지 않기 때문
  - 수 백만 개의 Node로 구성된 AVL 트리가 디스크에 저장된 상태에서의 Node 탐색은 많은 횟수의 디스크 접근을 요구한다. 그래서 m-원 탐색 트리를 고려하게 된다.