탐색

Data Structures and Algorithms

목차

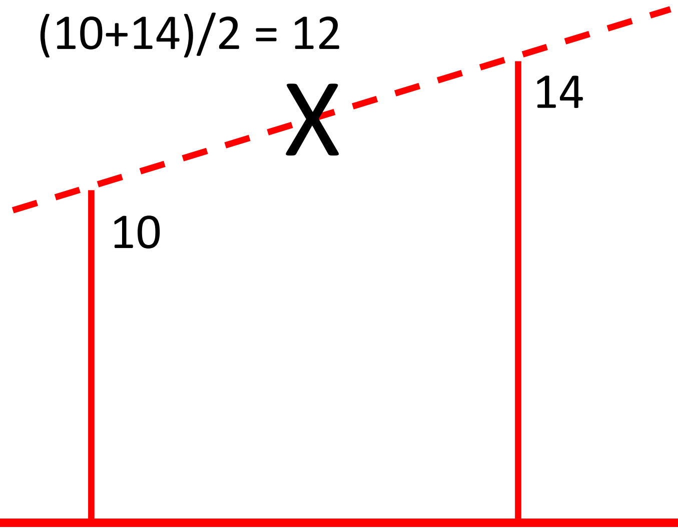
- 보간 탐색
- 이진탐색트리
- 균형 잡힌 이진 탐색 트리

탐색에서 중요한 것

- 두 가지 중요한 고민
  - 어떻게 저장할까?
  - 어떻게 찾을까?

- 트리 자료구조가 효율적인 탐색을 지원함

보간 탐색



보간: 주어진 구간의 값을 평균하여 추정하는 방법

보간 탐색

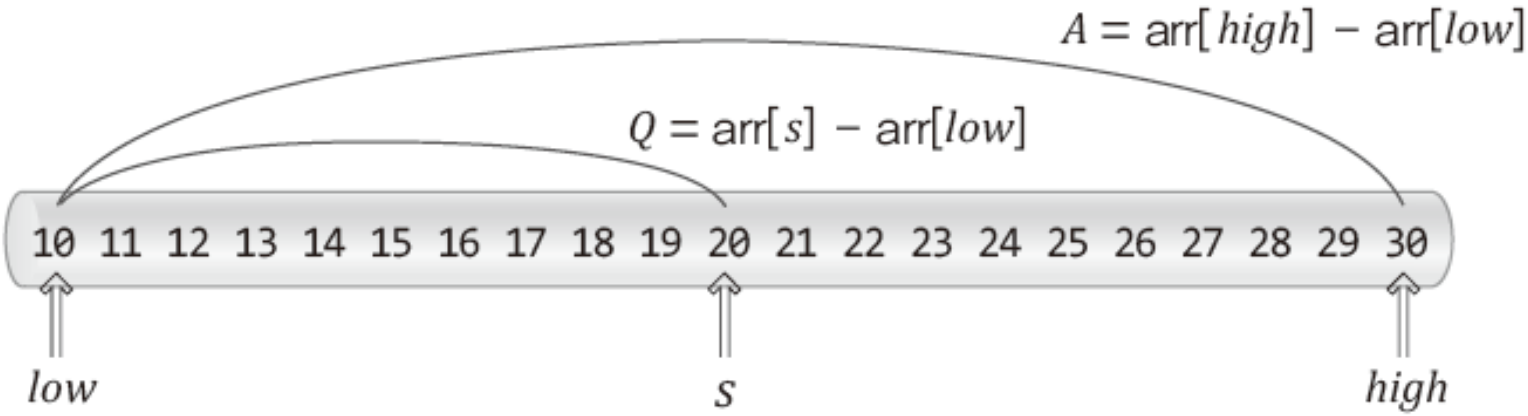
- 대상에 비례하여 탐색 위치 결정
  - 정렬된 값들을 대상으로 검색

- 확률적으로 한 번에 탐색 대상을 찾을 가능성이 있음

12 찾기

알고리즘 별 탐색 위치의 선택 방법

비례식



탐색 대상이 저장된 인덱스 값 임의의 S

탐색 위치의 인덱스 값 계산식

$$A : Q = (high\text{-}low) : (s\text{-}low)$$

$$s = \frac{Q}{A}(high\text{-}low) + low$$

$$s = \frac{x\text{-}arr[low]}{arr[high]\text{-}arr[low]}(high\text{-}low) + low$$

구조체 정의

- 실제 프로그램에서 탐색의 대상은 ‘데이터’가 아닌 ‘키(key)’
  - 편의상 데이터를 찾는 형태를 활용함

```
typedef Key int // 탐색 키 타입 정의

typedef Data double // 탐색 데이터 타입 정의

typedef struct item
{
    Key  searchKey;  // 탐색 키(search key)
    Data searchData; // 탐색 데이터(search data)
} Item;
```

구현

```
int search(int arr[], int first, int last, int target)
{
    int mid;
    // if(first > last)
    if(arr[first]>target || arr[last]<target)
        retrun -1;

     // mid = (first+last)/2;
     mid = ((double)(target-arr[first]) / (arr[last]-arr[first])
            * (last-first)) + first;

     if(arr[mid] == target)
         return mid;
     else if (target < arr[mid])
         return search(arr, first, mid-1, target);
     else
         return search(arr, mid+1, last, target);
}
```

$$s = \frac{x\text{-}arr[low]}{arr[high]\text{-}arr[low]}(high\text{-}low)+low$$

이진 탐색 트리

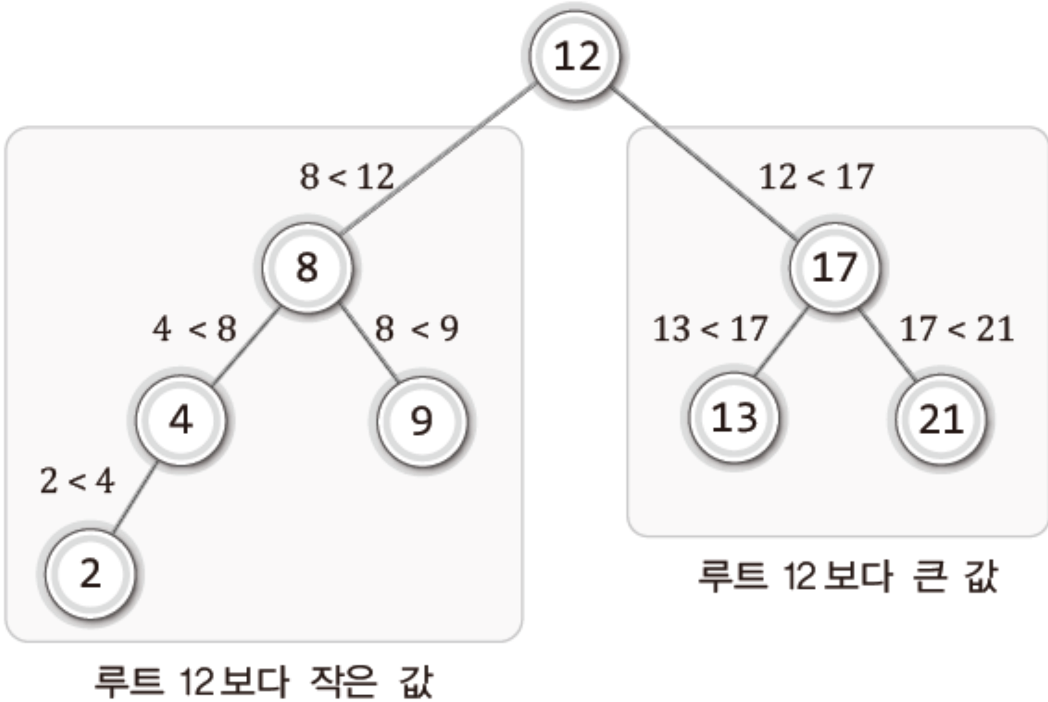
이진 탐색 트리

- 이진 탐색 트리 = 이진 트리 + 데이터의 저장 규칙

자료구조	위치 정보	데이터 수	탐색 노드 수
배열	有	1x10^9	1x10^9 (worst)
이진 탐색 트리	有	1x10^9	X < 30 (average)

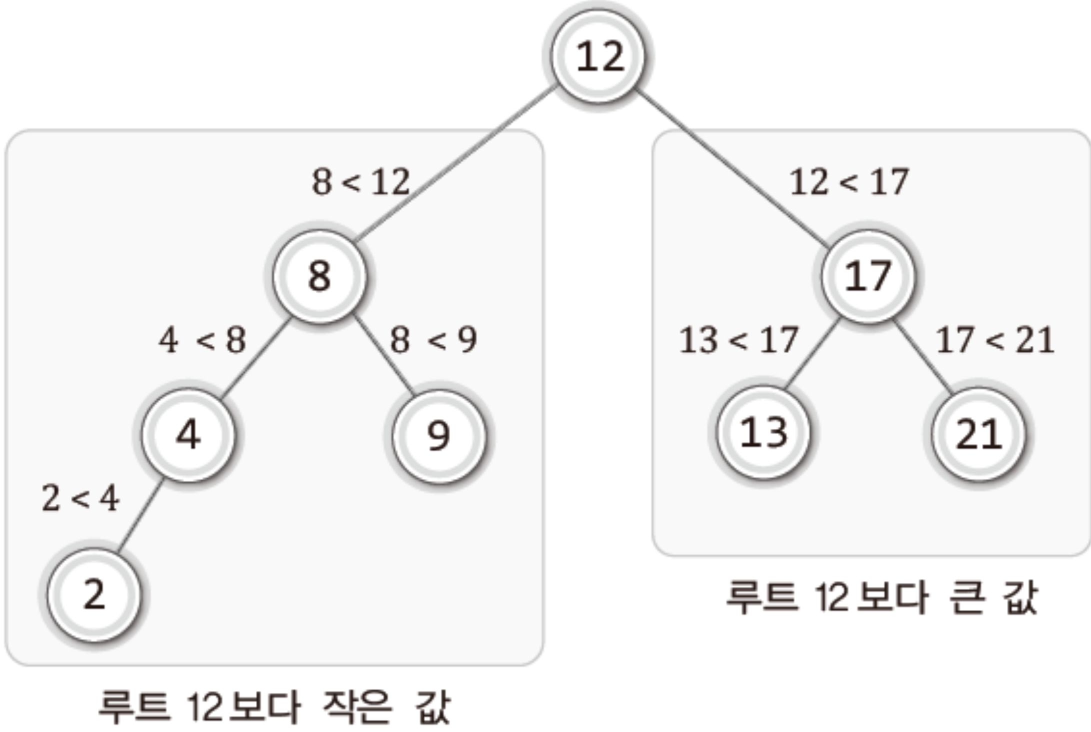
이진 탐색 트리

- 이진 탐색 트리의 노드에 저장된 키(key)는 유일!
- 루트 노드의 키 > 왼쪽 서브 트리를 구성하는 키
- 루트 노드의 키 < 오른쪽 서브 트리를 구성하는 키
- 왼쪽과 오른쪽 서브 트리도 이진 탐색 트리!



노드 추가: 예, 10 추가

- 새 노드의 추가 과정 = 탐색의 과정



구현: 기존 이진트리 코드 활용

BinarySearchTree.h 내용

```
#include "BinaryTree2.h"

typedef BTData BSTData;

// BST의 생성 및 초기화
void BSTMakeAndInit(BTreeNode ** pRoot);

// 노드에 저장된 데이터 반환
BSTData BSTGetNodeData(BTreeNode * bst);

// BST를 대상으로 데이터 저장(노드의 생성과정 포함)
void BSTInsert(BTreeNode ** pRoot, BSTData data);

// BST를 대상으로 데이터 탐색
BTreeNode * BSTSearch(BTreeNode * bst, BSTData target);
```

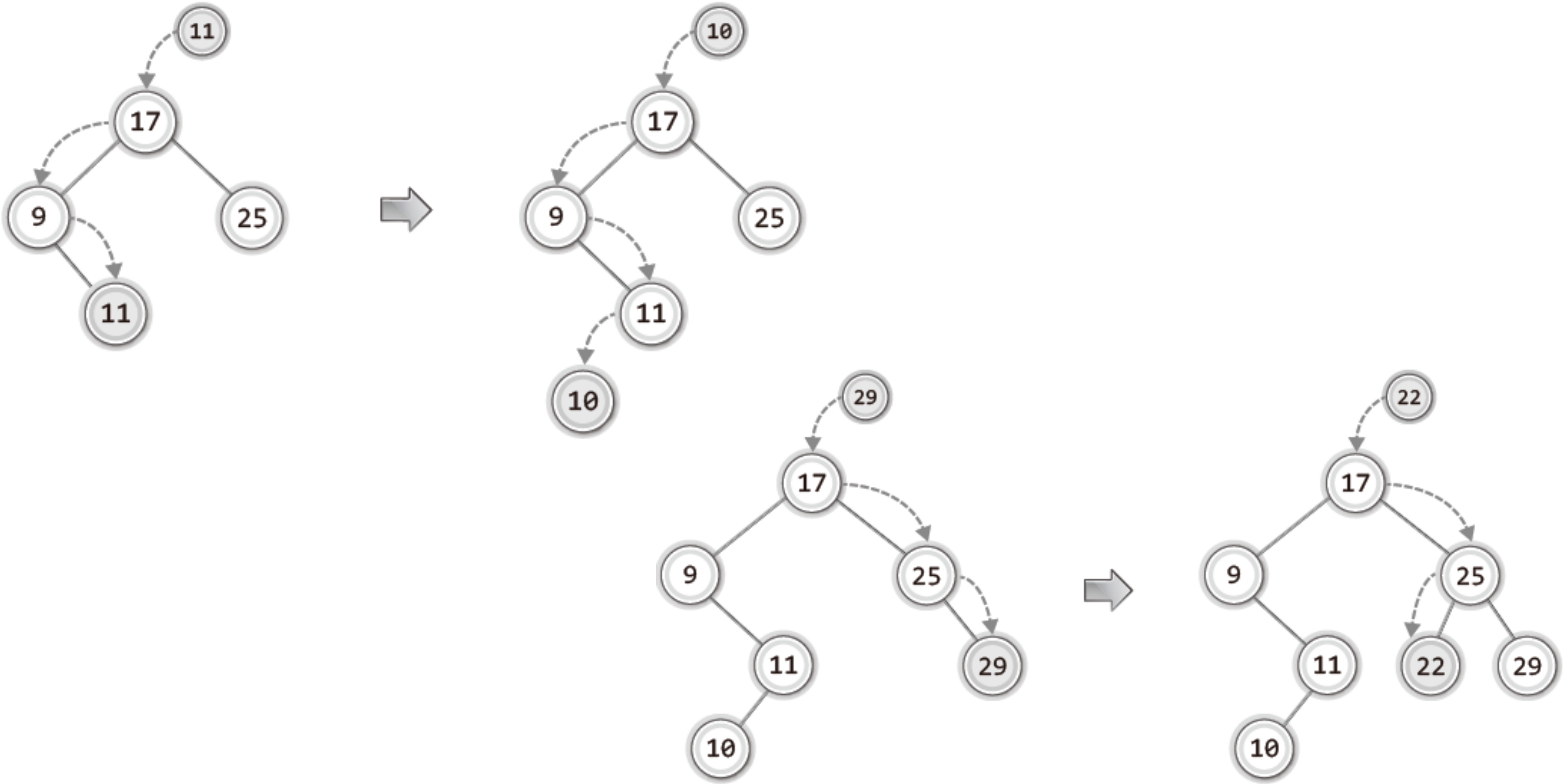
구현: 기존 이진트리 코드 활용

BinaryTree2.h 내용

```c
typedef int BTData;
typedef struct _bTreeNode
{
    BTData data;
    struct _bTreeNode * left;
    struct _bTreeNode * right;
} BTreeNode;

BTreeNode * MakeBTreeNode(void); // 노드를 동적으로 할당, 노드의 주소 값 반환
BTData GetData(BTreeNode * bt); // 저장된 노드의 데이터 반환
void SetData(BTreeNode * bt, BTData data); // 노드에 인자의 값을 저장

BTreeNode * GetLeftSubTree(BTreeNode * bt); // 노드의 왼쪽 자식 노드의 주소 반환
BTreeNode * GetRightSubTree(BTreeNode * bt); // 노드의 오른쪽 자식 노드의 주소 반환

//노드의 왼쪽/오른쪽 자식 노드 교체
void MakeLeftSubTree(BTreeNode * main, BTreeNode * sub);
void MakeRightSubTree(BTreeNode * main, BTreeNode * sub);
```

이진 탐색 트리: 삽입

- 삽입: 비교 대상이 없는 위치까지 탐색 후 저장
  - 11, 10, 29, 22 삽입



삽입 함수의 구현

```
void BSTInsert(BTreeNode ** pRoot, BSTData data)
{
    BTreeNode * pNode = NULL;    // parent node
    BTreeNode * cNode = *pRoot;    // current node
    BTreeNode * nNode = NULL;    // new node

    while(cNode != NULL){ // 새로운 노드가 추가될 위치를 찾는다.
        if(data == GetData(cNode)) return;    // 키의 중복을 허용하지 않음

        pNode = cNode;

        if(GetData(cNode) > data) cNode = GetLeftSubTree(cNode);
        else                      cNode = GetRightSubTree(cNode);
    }

    // pNode의 서브 노드에 추가할 새 노드의 생성
    nNode = MakeBTreeNode();    // 새 노드의 생성
    SetData(nNode, data);    // 새 노드에 데이터 저장

    // pNode의 서브 노드에 새 노드를 추가
    if(pNode != NULL)  {   // 새 노드가 루트 노드가 아니라면,
        if(data < GetData(pNode)) MakeLeftSubTree(pNode, nNode);
        else                      MakeRightSubTree(pNode, nNode);
    } else {   // 새 노드가 루트 노드라면,
        *pRoot = nNode;
    }
}
```

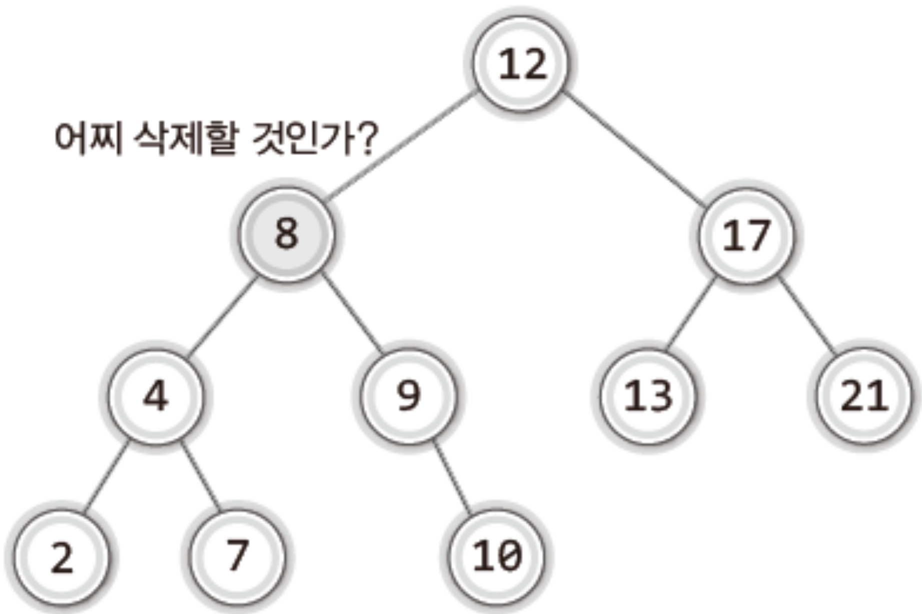
탐색 함수의 구현

```
BTreeNode * BSTSearch(BTreeNode * bst, BSTData target)
{
    BTreeNode * cNode = bst;    // current node
    BSTData cd;                 // current data

    while(cNode != NULL)
    {
        cd = GetData(cNode);

        if(target == cd)
            return cNode; // 탐색에 성공하면 해당 노드의 주소 값을 반환!
        else if(target < cd)
            cNode = GetLeftSubTree(cNode);
        else
            cNode = GetRightSubTree(cNode);
    }
    return NULL; // 탐색 대상이 없음
}
```

삭제 구현: 상황 별 삭제

- 삭제와 관련해서 고려해야 할 세 가지 상황
  1. 대상이 단말 노드인 경우
  2. 대상이 하나의 자식 노드를(하나의 서브 트리를) 갖는 경우
  3. 대상이 두 개의 자식 노드를(두 개의 서브 트리를) 갖는 경우

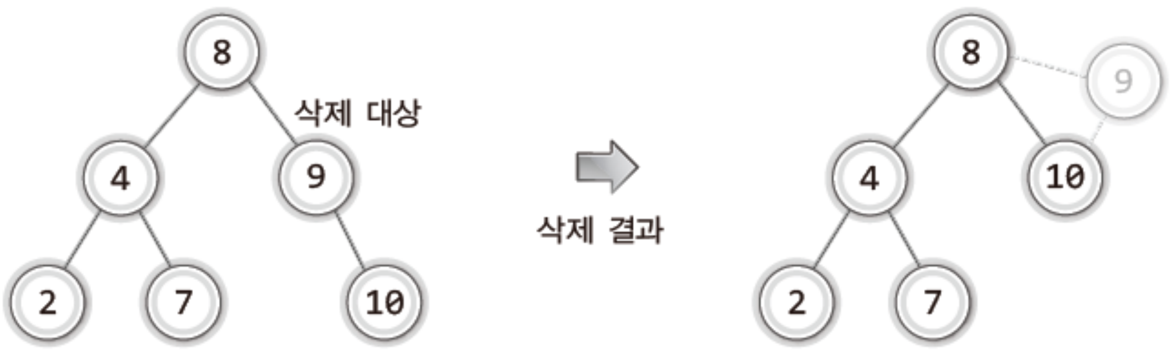
핵심 질문: 삭제로 인한 빈 자리를 어떻게 채워야 할까?



Case 1: 단말 노드 (Easiest case)


```
// dNode와 pNode는 각각 삭제할 노드와 이의 부모 노드를 가리키는 포인터 변수
if(GetLeftSubTree(dNode) == NULL && GetRightSubTree(dNode) == NULL)
{
    if(GetLeftSubTree(pNode) == dNode) // 왼쪽 자식 노드인 경우
        RemoveLeftSubTree(pNode);      // 왼쪽 자식 노드 삭제
    else                               // 오른쪽 자식 노드인 경우
        RemoveRightSubTree(pNode);     // 오른쪽 자식 노드 삭제
}
```

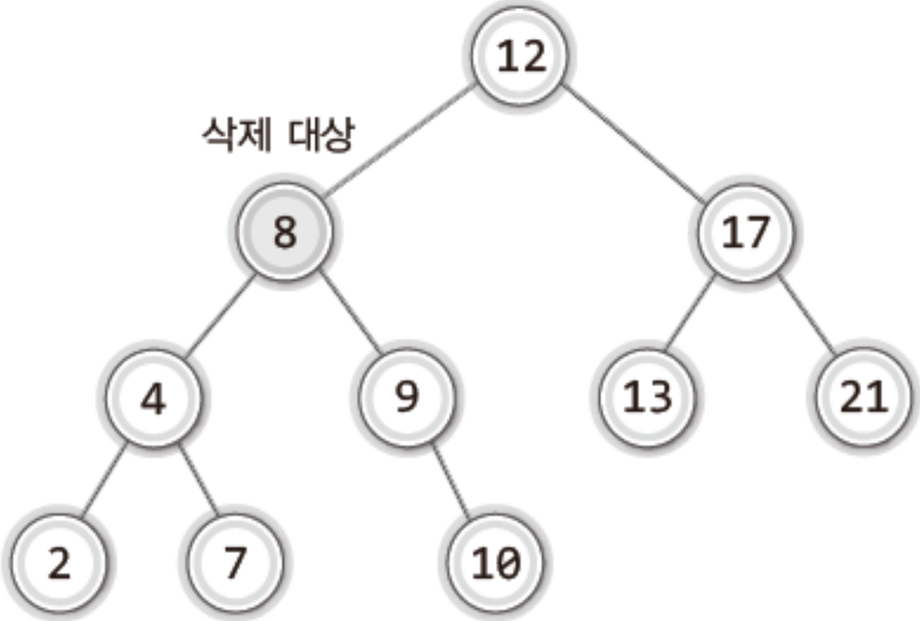
Case 2: 하나의 서브트리

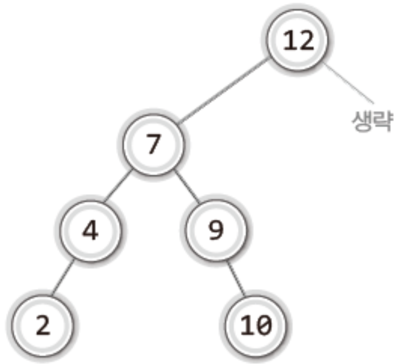



```
else if(GetLeftSubTree(dNode) == NULL || GetRightSubTree(dNode) == NULL)
{
    BTreeNode * dcNode;    // delete node의 자식 노드
    // 삭제 대상의 자식 노드를 찾기
    if(GetLeftSubTree(dNode) != NULL)     // 자식 노드가 왼쪽
        dcNode = GetLeftSubTree(dNode);
    else                                  // 자식 노드가 오른쪽
        dcNode = GetRightSubTree(dNode);

    if(GetLeftSubTree(pNode) == dNode).   // 삭제 대상이 왼쪽 자식 노드
        ChangeLeftSubTree(pNode, dcNode); // 왼쪽으로 연결
    else                                  // 삭제 대상이 오른쪽 자식 노드
        ChangeRightSubTree(pNode, dcNode);// 오른쪽으로 연결
}
```

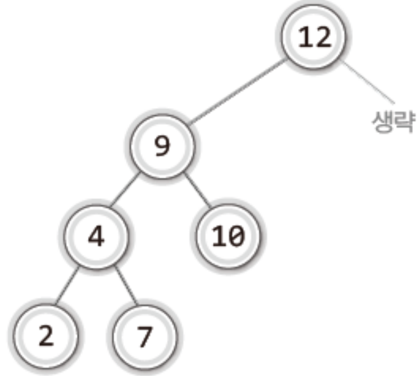
Case 3: 두 개의 서브트리 I

- 결정할 것: 삭제 대상 대신에 넣을 값
- 두 개의 선택지
  - 왼쪽 서브트리에서 가장 큰 값
  - 오른쪽 서브트리에서 가장 작은 값





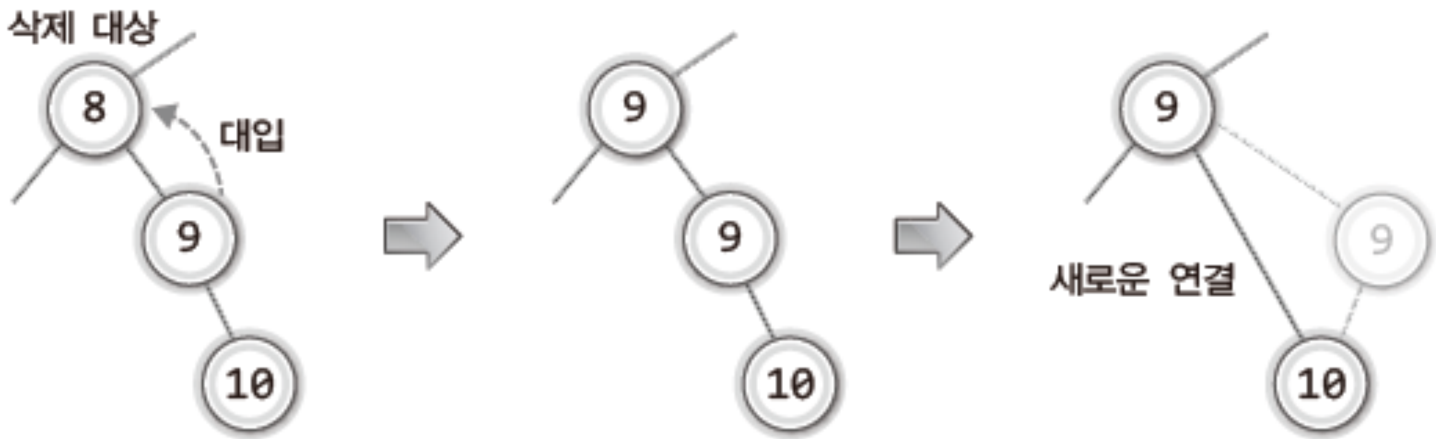
왼쪽



오른쪽

Case 3: 두 개의 서브트리 II

- 오른쪽에서 가장 작은 값으로 대체
  1. 삭제할 노드를 대체할 노드를 찾는다.
  2. 대체할 노드에 저장된 값을 삭제할 노드에 대입한다.
  3. 대체할 노드의 부모 노드와 자식 노드를 연결한다.



Case 3: 두 개의 서브트리 III

```
else {
    BTreeNode * mNode = GetRightSubTree(dNode);    // mininum node
    BTreeNode * mpNode = dNode;    // mininum node의 부모 노드
    int delData;

    while(GetLeftSubTree(mNode) != NULL) { // 삭제할 노드를 대체할 노드를 찾기
        mpNode = mNode;
        mNode = GetLeftSubTree(mNode);
    }
    // 대체할 노드에 저장된 값을 삭제할 노드에 대입
    delData = GetData(dNode);    // 대입 전 데이터 백업
    SetData(dNode, GetData(mNode));

    // 대체할 노드의 부모 노드와 자식 노드를 연결
    if(GetLeftSubTree(mpNode) == mNode) // 대체할 노드가 왼쪽 노드라면
        ChangeLeftSubTree(mpNode, GetRightSubTree(mNode)); //
    else                                // 대체할 노드가 오른쪽 노드라면
        ChangeRightSubTree(mpNode, GetRightSubTree(mNode));

    dNode = mNode;
    SetData(dNode, delData);    // 백업 데이터 복원
}
```

삭제 연산을 위한 추가 확장 함수들

```
// 왼쪽 자식 노드 제거, 제거된 노드의 주소 값이 반환
BTreeNode * RemoveLeftSubTree(BTreeNode * bt);


// 오른쪽 자식 노드 제거, 제거된 노드의 주소 값이 반환
BTreeNode * RemoveRightSubTree(BTreeNode * bt);


// 메모리 소멸을 수반하지 않고 main의 왼쪽 자식 노드를 변경
void ChangeLeftSubTree(BTreeNode * main, BTreeNode * sub);


// 메모리 소멸을 수반하지 않고 main의 오른쪽 자식 노드를 변경
void ChangeRightSubTree(BTreeNode * main, BTreeNode * sub);
```

확장 함수의 구현

```c
BTreeNode * RemoveLeftSubTree(BTreeNode * bt)
{
    BTreeNode * delNode;

    if(bt != NULL) {
        delNode = bt->left;
        bt->left = NULL;
    }
    return delNode;
}
```

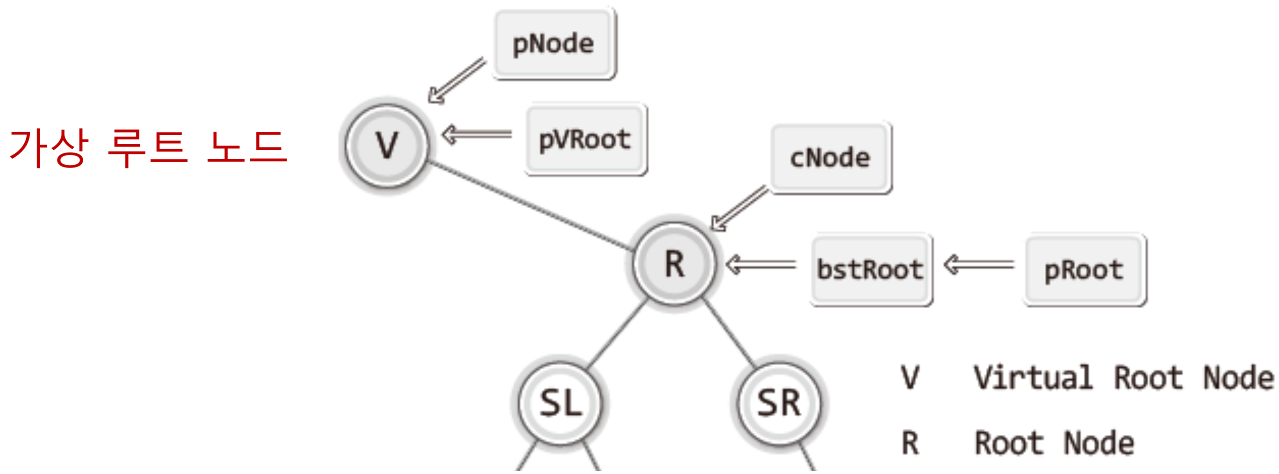
```c
void ChangeLeftSubTree(BTreeNode * main,
                       BTreeNode * sub)
{
    main->left = sub;
}
```

```c
BTreeNode * RemoveRightSubTree(BTreeNode * bt)
{
    BTreeNode * delNode;

    if(bt != NULL) {
        delNode = bt->right;
        bt->right = NULL;
    }
    return delNode;
}
```

```c
void ChangeRightSubTree(BTreeNode * main,
                        BTreeNode * sub)
{
    main->right = sub;
}
```

삭제 연산 (The Rest): 초기화

- 가상 루트 노드 생성 이유
  - 루트 노드 삭제의 경우를 나머지 삭제루틴과 같이 일반화하기 위해서




```
BTreeNode * BSTRemove(BTreeNode ** pRoot, BSTData target)
{
    // 삭제 대상이 루트 노드인 경우를 별도로 고려해야 한다.
    BTreeNode * pVRoot = MakeBTreeNode();    // 가상 루트 노드;

    BTreeNode * pNode = pVRoot;    // parent node
    BTreeNode * cNode = *pRoot;    // current node
    BTreeNode * dNode;    // delete node

    // 루트 노드를 pVRoot가 가리키는 노드의 오른쪽 서브 노드가 되게 한다.
    ChangeRightSubTree(pVRoot, *pRoot);
```

삭제 연산 (The Rest): 삭제 대상 찾기

- pNode가 cNode의 부모 노드를 가리켜야함
  - BSTSearch 함수 호출로 대신할 수 없음
- 이후 부터 case 1, 2, 3 수행

```
// 삭제 대상을 저장한 노드 탐색
while(cNode != NULL && GetData(cNode) != target) {
    pNode = cNode;

    if(target < GetData(cNode))
        cNode = GetLeftSubTree(cNode);
    else
        cNode = GetRightSubTree(cNode);
}

if(cNode == NULL)    // 삭제 대상이 존재하지 않는다면,
    return NULL;

dNode = cNode;    // 삭제 대상을 dNode가 가리키게 한다.
```

BSTSearch로 대체 불가

삭제 연산 (The Rest): 루트 노드 삭제

```
// 삭제된 노드가 루트 노드인 경우에 대한 처리
if(GetRightSubTree(pVRoot) != *pRoot)
    *pRoot = GetRightSubTree(pVRoot);

free(pVRoot); // 가상 루트 노드 삭제
return dNode; // 삭제 대상 반환
```

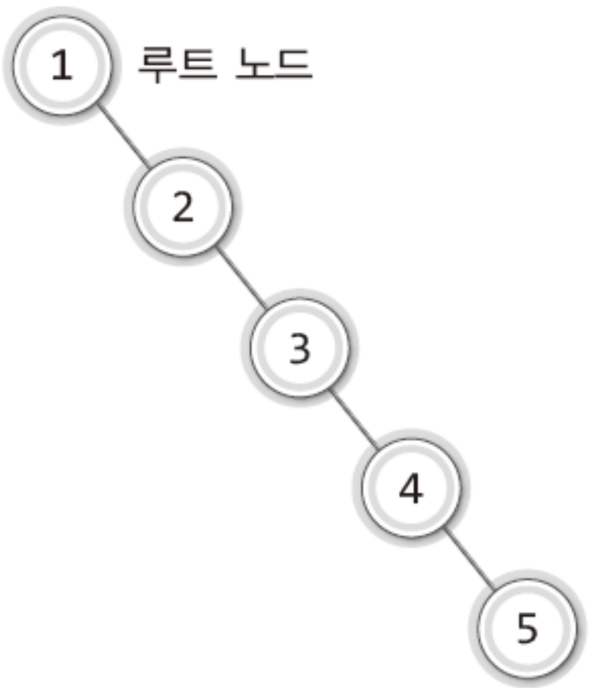
균형 잡힌 이진 탐색 트리

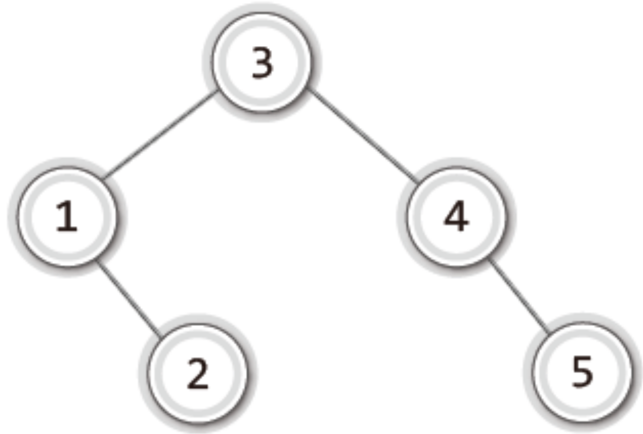
AVL 트리의 이해

이진 탐색 트리의 문제점

- 문제점
  - 이진 탐색 트리의 탐색 연산: O(log2n)
  - 균형이 깨진 경우: O(n)

- 예: 1부터 5까지 삽입한 경우   예: 3, 1, 2, 4 5순으로 삽입한 경우





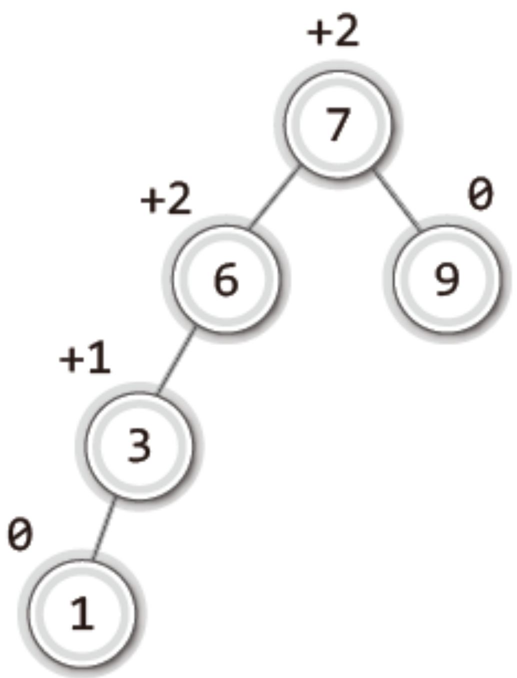
순서 변경으로 균형이 잡힘

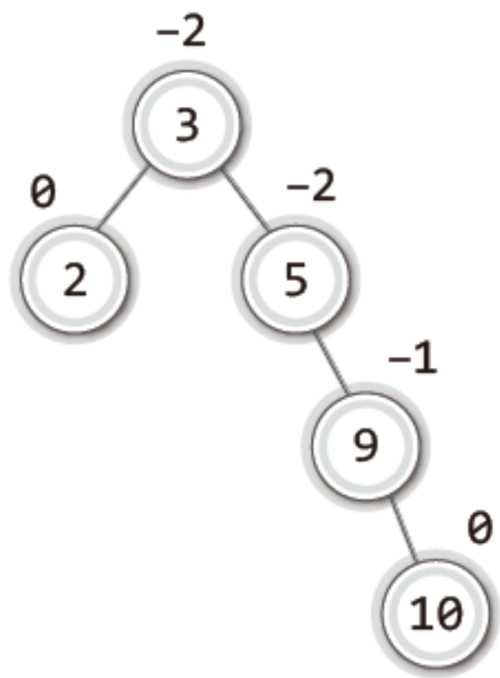
문제를 해결한 다른 트리

- 균형잡힌 트리
  - AVL 트리
  - 2-3-4 트리
  - Red-Black 트리
  - 등등…

AVL 트리: 균형 인수

- 균형 인수 = 왼쪽 서브트리의 높이 – 오른쪽 서브 트리의 높이
  Balance Factor
- 균형 잡기 규칙
  - 균형 인수의 절댓값이 2이상인 경우 rebalance





Case 1: LL 상태

- LL상태, 컨디션
- “노드의 **왼쪽(Left)**에 자식 노드가 하나 존재하고, 다시 **왼쪽(Left)**에 자식 노드가 또 하나 존재”

▶ [그림 12-4: LL회전의 방법과 그 결과]

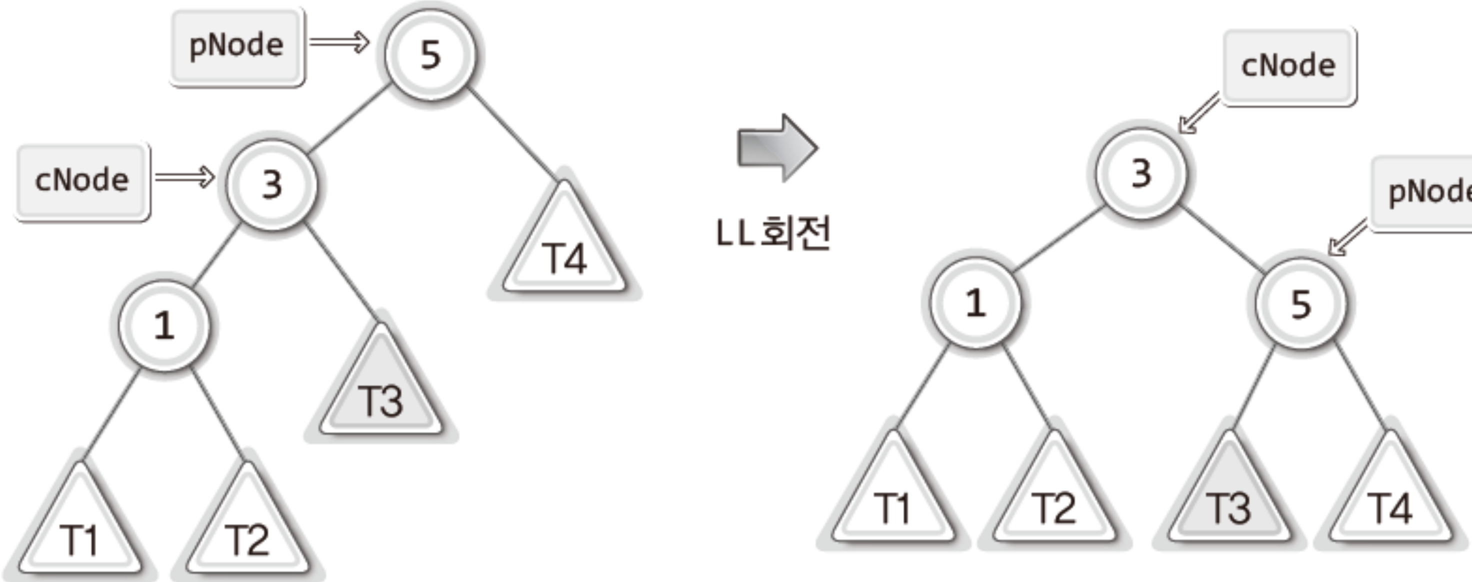
Case 1: LL 회전 - 솔루션

- 단순한 예


```
ChangeLeftSubTree(cNode, pNode);
```

Case 1: LL 회전 - 솔루션

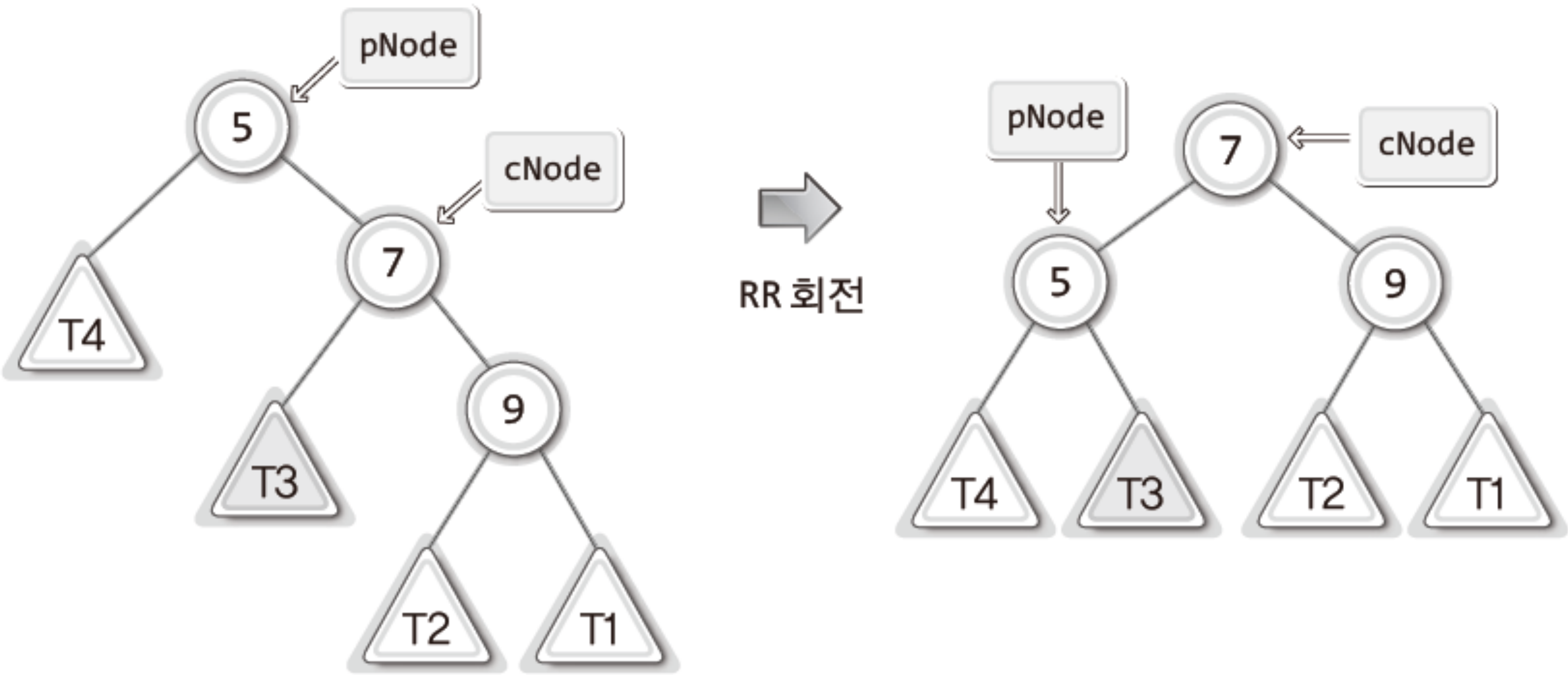
- 일반화




```
ChangeLeftSubTree(pNode, GetRightSubTree(cNode));
Change Right SubTree(cNode, pNode);
```

Case 1: RR회전

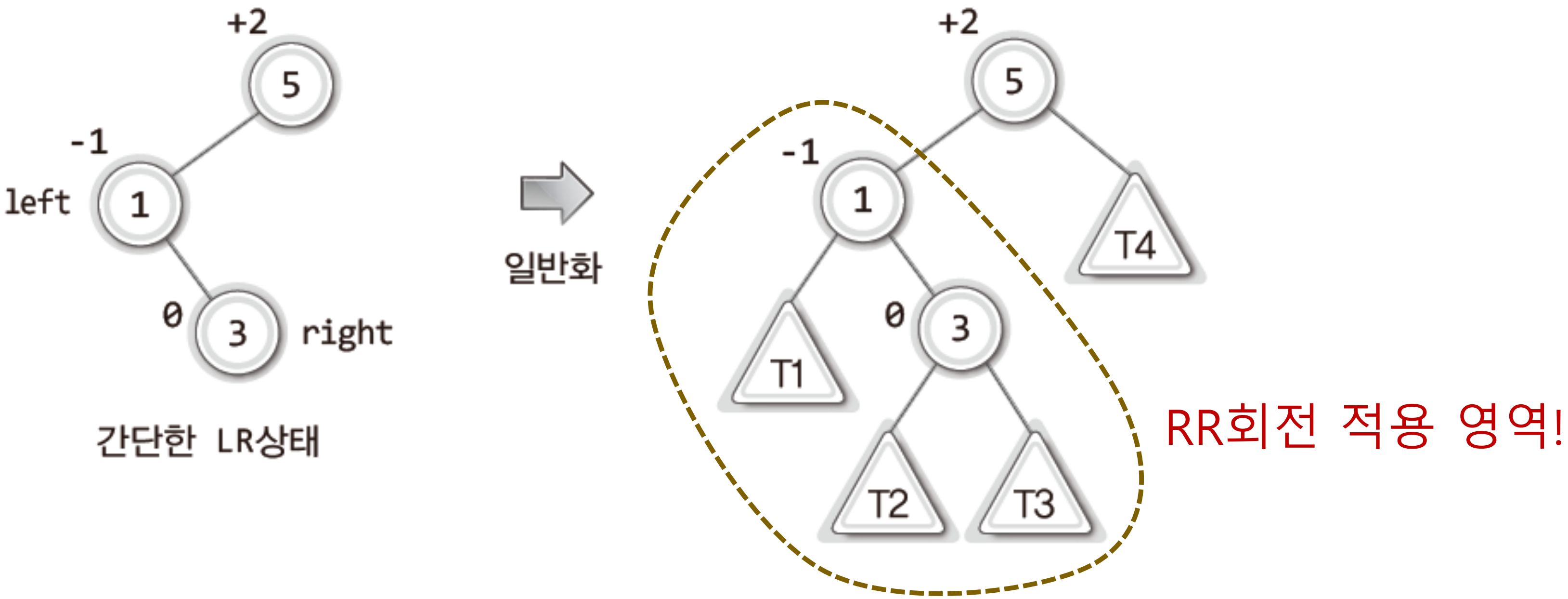
- LL회전과 RR회전은 같은 개념




```
ChangeRightSubTree(pNode, GetLeftSubTree(cNode));
ChangeLeftSubTree(cNode, pNode);
```

Case 2: LR 상태

- LL / RR상태처럼 한 번 회전으로 균형 잡을 수 없음
- LR 상태에서 LL / RR상태로 변환이 중요
  - 과정: LR상태 → RR회전 (부수 효과 활용) → LL상태



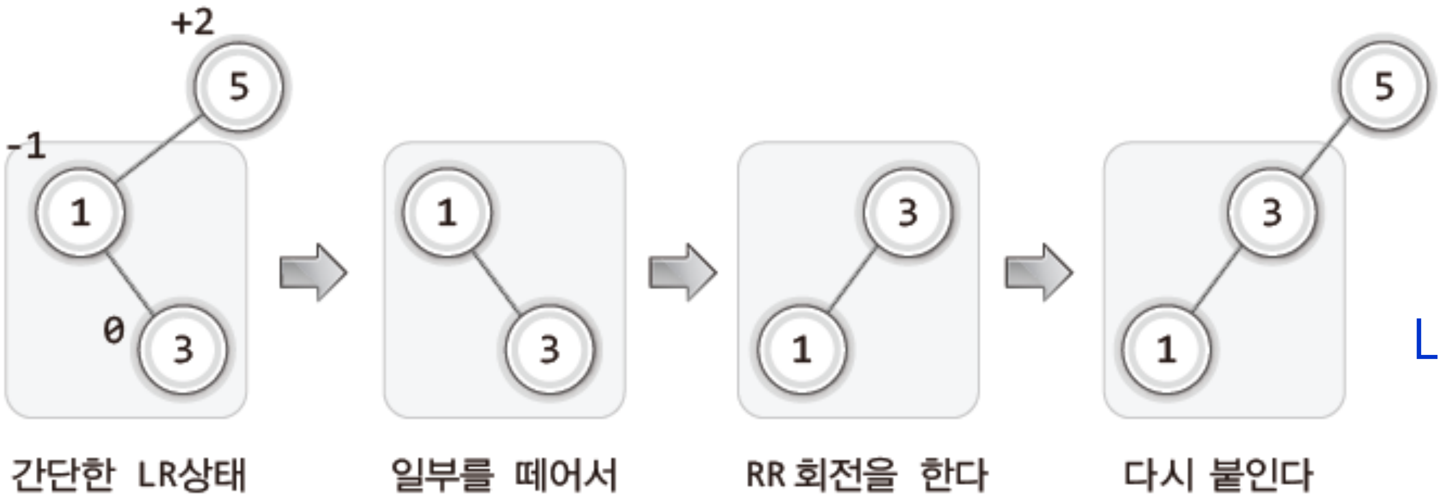
RR회전의 부수 효과: 부모 자식 관계 교환

일반적인 RR회전

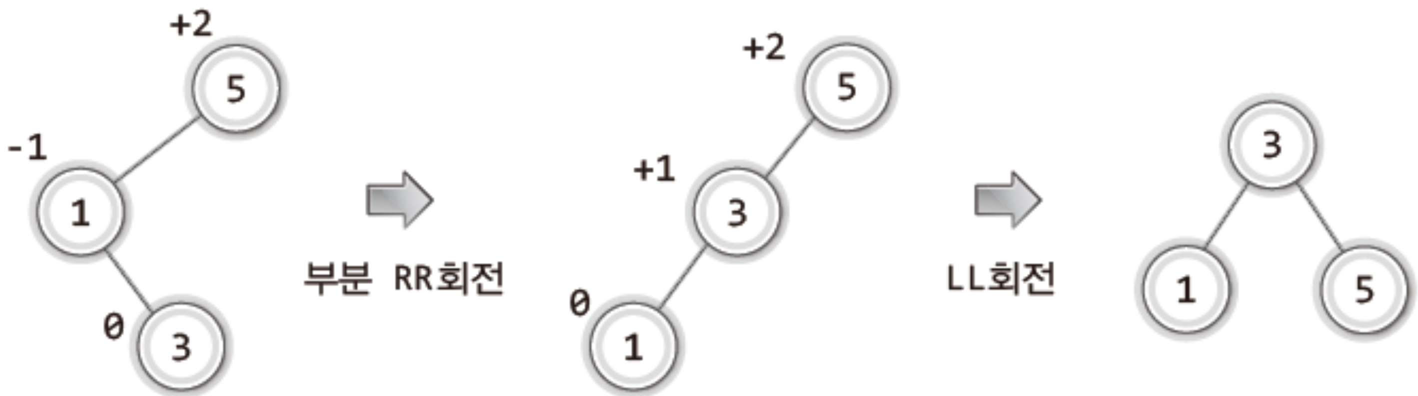
단말 노드가 NULL인 경우도
RR 회전 가능

부모 ←→ 자식 관계

Case 2: LR 회전



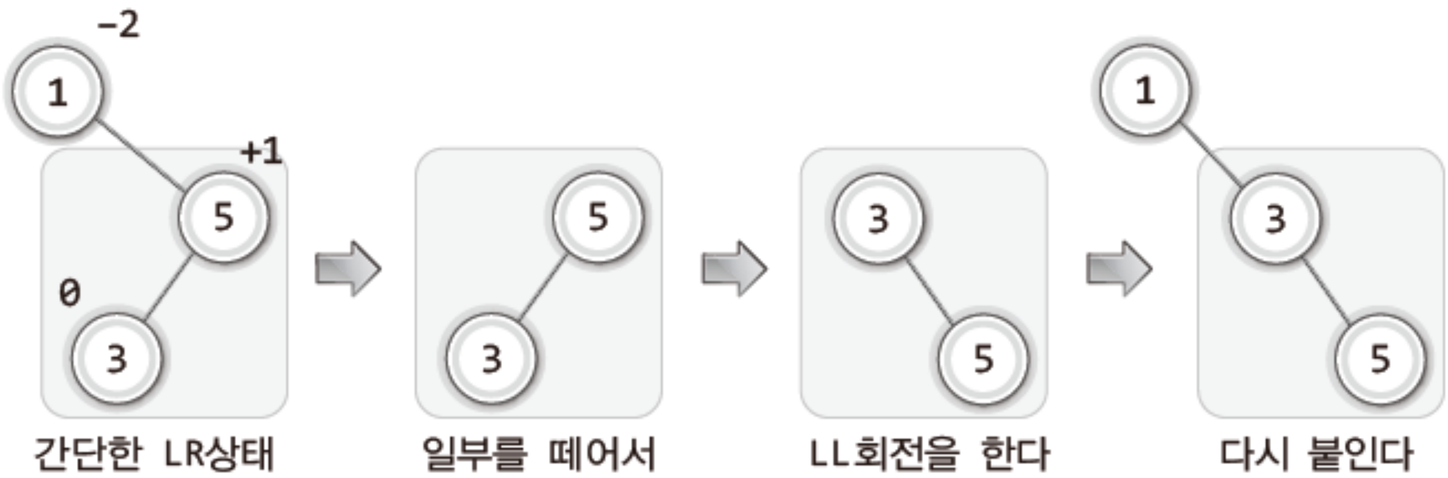
LR → LL 상태로



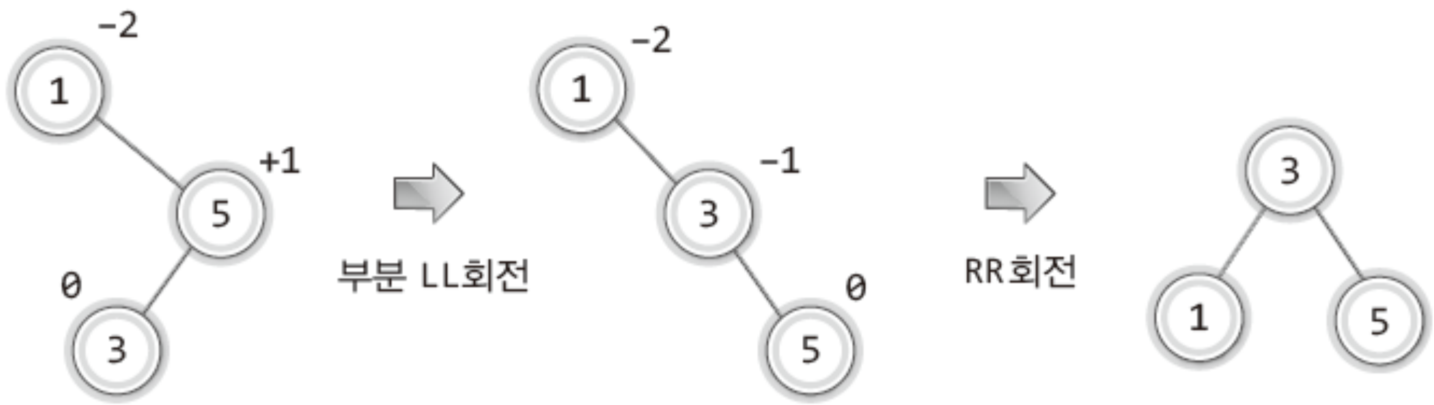
LR회전

Case 2: RL상태와 RL회전

- LR상태 LR회전과 RL상태 RL회전은 방향만 차이가 있음



RL → RR상태로



RL회전

균형 잡힌 이진 탐색 트리

AVL 트리 구현

구현 방법

- 기존 파일
  - BinaryTree3.h :이진 트리의 헤더파일
  - BinaryTree3.c :이진 트리 구성 함수
  - BinarySearchTree2.h : 이진 탐색 트리의 헤더파일
  - BinarySearchTree2.c : 이진 탐색 트리 구성 함수

- AVL 트리는 이진 탐색 트리의 일종
  - 이진 탐색 트리 기반으로 구현
- BinarySearchTree2.c에 리밸런싱 기능을 추가
  - 파일의 이름을 BinarySearchTree3.c로 변경
  - 새로 추가할 파일
    - AVLRebalance.h : 리밸런싱 관련 함수들의 선언
    - AVLRebalance.c : 리밸런싱 관련 함수들의 정의

구현의 핵심

- 확장할 함수들: 노드 삽입/삭제 시 균형이 깨짐
    - BSTInsert 함수 트리에 노드를 추가
    - BSTRemove 함수 트리에서 노드를 제거
- 확장의 형태

```
void BSTInsert(BTreeNode ** pRoot, BSTData data)
{
    …
    *pRoot = Rebalance(pRoot); // 노드 추가 후 리밸런싱
}

BTreeNode * BSTRemove(BTreeNode ** pRoot, BSTData target)
{
    …
    *pRoot = Rebalance(pRoot); // 노드 제거 후 리밸런싱!
    return dNode;
}
```

균형: 트리의 높이의 차

```
int GetHeightDiff(BTreeNode * bst) // 높이의 차를 반환
{
    int lsh;    // left sub tree height
    int rsh;    // right sub tree height

    if(bst == NULL)
        return 0;

    lsh = GetHeight(GetLeftSubTree(bst));
    rsh = GetHeight(GetRightSubTree(bst));

    return lsh - rsh;
}
```

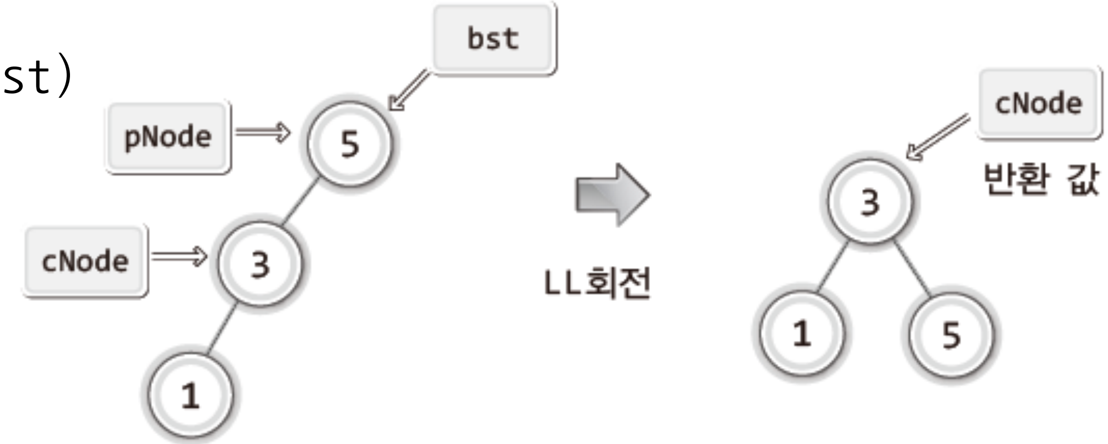
균형: 트리의 높이 계산

```
int GetHeight(BTreeNode * bst)  // 트리의 높이를 계산하여 반환
{
    int leftH; // left height
    int rightH; // right height

    if(bst == NULL)
        return 0;

    leftH = GetHeight(GetLeftSubTree(bst)); // 왼쪽 높이 계산

    rightH = GetHeight(GetRightSubTree(bst)); // 오른쪽 높이 계산

    if(leftH > rightH) // 큰 값의 높이를 반환한다.
        return leftH + 1;
    else
        return rightH + 1;
}
```

균형: LL/RR회전 (Case 1)

- 회전 후 parent node의 위치 바뀜
  새로운 parent node 위치 리턴
- LL->RR회전 Left-> Right로 수정

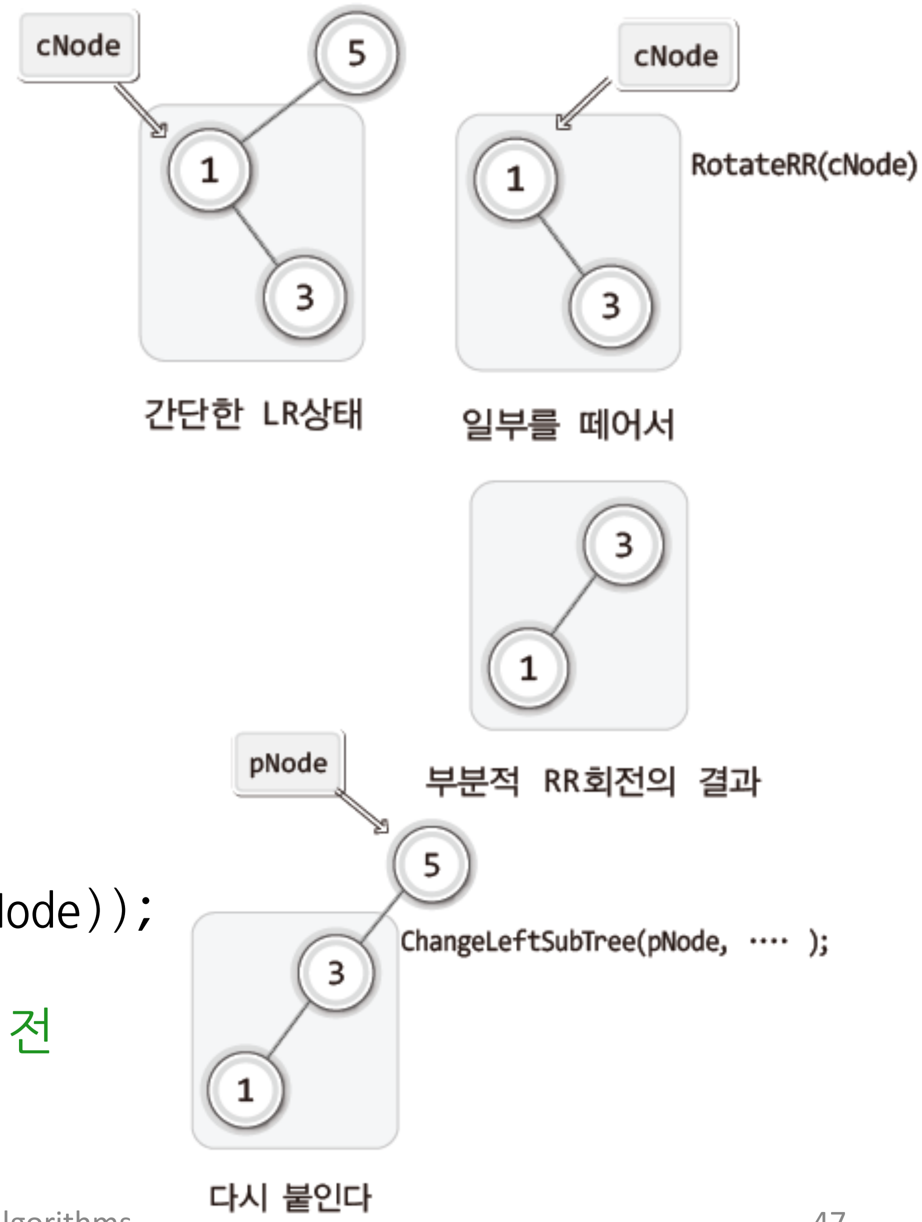
```
BTreeNode *RotateLL(BTreeNode *bst)
{
    BTreeNode *pNode;
    BTreeNode *cNode;

    // pNode와 cNode의
    // 회전을 위한 자리 잡기
    pNode = bst;
    cNode = GetLeftSubTree(pNode);

    // 회전
    ChangeLeftSubTree(pNode, GetRightSubTree(cNode));
    ChangeRightSubTree(cNode, pNode);

    return cNode; // 변경된 루트 노드 주소 값 반환
}
```



균형: LR/RL회전 (Case 2)

- 부분적 RR회전 후 LL회전을 진행
- RL회전은 Left->Right 변경

```
BTreeNode * RotateLR(BTreeNode * bst)
{
     BTreeNode * pNode;
     BTreeNode * cNode;

     // pNode와 cNode의
     // 회전을 위한 자리 잡기
     pNode = bst;
     cNode = GetLeftSubTree(pNode);

     // 부분적 RR 회전
     ChangeLeftSubTree(pNode, RotateRR(cNode));

     return RotateLL(pNode);      // LL 회전
}
```



균형: Rebalance

```
BTreeNode * Rebalance(BTreeNode ** pRoot)
{
    int hDiff = GetHeightDiff(*pRoot); // 균형 인수 계산

    if(hDiff > 1)       // 왼쪽 서브 트리 방향으로 높이가 2 이상 크다면
    {                   // 왼쪽으로 불균형: LL 또는 LR상태
        if(GetHeightDiff(GetLeftSubTree(*pRoot)) > 0)
            *pRoot = RotateLL(*pRoot);
        else
            *pRoot = RotateLR(*pRoot);
    }
    if(hDiff < -1)      // 오른쪽 서브 트리 방향으로 2 이상 크다면
    {                   // 오른쪽으로 불균형: RR 또는 RL상태
        if(GetHeightDiff(GetRightSubTree(*pRoot)) < 0)
            *pRoot = RotateRR(*pRoot);
        else
            *pRoot = RotateRL(*pRoot);
    }
    return *pRoot;
}
```

상태 구분

- GetHeightDiff(GetLeftSubTree(*pRoot)) > 0
  - LL 또는 LR 상태

간단한 LL상태 간단한 LR상태

- GetHeightDiff(GetLeftSubTree(*pRoot)) > 0
  - RR 또는 RL 상태

간단한 RR상태 간단한 RL상태