<BST & RBT Implement>

1. BST(binary search tree, 이진 탐색 트리)의 삽입, 삭제 연산

(1) 구현의 개요

[초기화 과정] 노드 구조체, 트리 구조체의 정의

bstree를 구성하고 있는 노드의 구조체를 bstreeNode로 정의하였습니다. 각각의 bstreeNode는 키 값, 자기 참조 구조체로 자신의 오른쪽 자식 노드와 왼쪽 자식 노드, 부모 노드를 필드값으로 가지고 있어야 합니다.

또한, 이진 탐색 트리의 구조체를 bstree로 정의했습니다. bstrree는 이진 탐색 트리의 기본적인 정보를 담고 있는 구조체로, 루트 노드의 정보를 가지고 있어야 합니다.

```c
/* BST node 구조체 정의 */
typedef struct bstreeNode {
    int key;
    struct  bstreeNode* left, * right, * p;
} bstreeNode;

typedef struct {
    bstreeNode* root;
}bstree;
```

[BST/bstree.h]

```c
// 새 노드 생성
bstreeNode* createNode(int k) {
    bstreeNode* newNode = (bstreeNode*)calloc(1, sizeof(bstreeNode));
    newNode->key = k;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->p = NULL; // 새 노드의 부모를 초기화
    return newNode;
}
```

[BST/bst.c]

키 값을 k로 매개변수로 받아 이진 탐색 트리의 노드를 새로 생성하여 반환하는 함수를 구현했습니다.

(2) BST 삽입, 삭제, 순회 연산 구현

[BST 중위 순회 연산 구현]

```c
// 중위 순회: L V R
void inOrderTreeWalk(bstreeNode* x) {
    if (x != NULL) {
        inOrderTreeWalk(x->left);
        printf("%d ", x->key);
        inOrderTreeWalk(x->right);
    }
}
```

[BST/bst.c]

이진 탐색 트리는 [1]각 노드의 키 값을 기준으로 중위 순회 시 오름차순으로 출력되는 성질을 가지고 있습니다. 따라서 강의 자료에 정의되어있던 중위 순회 의사코드를 구현했습니다.
이진 탐색 트리의 중위 순회는 (1)왼쪽 자식 노드를 루트 노드로 하는 서브 트리를 재귀적으로 호출하고, (2)루트 노드의 키 값을 출력한 후, (3)오른쪽 자식 노드를 루트 노드로 하는 서브 트리를 재귀적으로 호출합니다.

[BST 삽입 연산 구현]

```c
// 이진 탐색 트리에 노드 삽입
void treeInsert(bstree* T, bstreeNode* z) {
    bstreeNode* y = NULL;      // 초기화(root 노드의 부모 노드 = null)
    bstreeNode* x = T->root;   // x는 현재의 root 노드부터 시작

    while (x != NULL) {
        y = x;
        if (z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }

    z->p = y; // 새 노드의 부모를 설정

    if (y == NULL)
        T->root = z;
    else if (z->key < y->key)
        y->left = z;
    else
        y->right = z;
}
```

[BST/bst.c]

이진 탐색 트리의 삽입 연산은 키 값을 기준으로, *∀lkey(left subtree) ≤ key(root) ≤ ∀key(right subtree)* 라는 이진 탐색 트리의 특성에 맞추어, 삽입될 노드의 키 값으로 적절한 자리를 루트 노드에서부터 순차적으로 탐색한 후에 해당 자리에 적절히 삽입해줍니다.

---

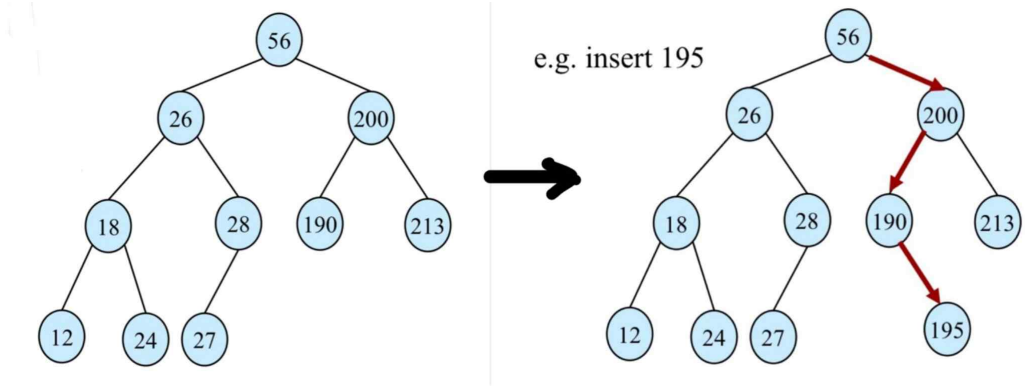
[1] recursively printed in monotonically increasing order

```c
bstree* BST = (bstree*)malloc(sizeof(bstree)); // bstree 구조체 초기화
BST->root = NULL;

// sample 노드 데이터 삽입
int bst_nodeList[] = { 56, 26, 18, 28, 190, 213, 200, 12, 24, 27 };

for (int i = 0; i < 10; i++) {
    treeInsert(BST, createNode(bst_nodeList[i]));
    inOrderTreeWalk(BST->root);
    printf("₩n");
}
// BST 출력
printf("초기 BST 중위 순회: ");
inOrderTreeWalk(BST->root);
printf("₩n");

printf("==============================₩n");
printf("BST insert 연산 진행₩n");
printf("==============================₩n");

// 키가 195인 노드 생성 후 삽입 연산 진행
int k1 = 195;
treeInsert(BST, createNode(195));

printf("%d 삽입 후 BST 중위 순회: ", k1);
inOrderTreeWalk(BST->root);
printf("₩n");
```

[bst/main.c]

main함수에서 bst.c에 구현한 treeInsert()함수를 이용하여 노드를 삽입하여 초기의 BST를 구축하였습니다.

후에는 treeInsert()함수를 이용하여 샘플 데이터인 키 값이 195인 노드를 생성 후 기존의 BST에 삽입하고, 중위 순회 연산을 이용하여 출력해보았습니다.



[출력 결과 화면]

```
56
26 56
18 26 56
18 26 28 56
18 26 28 56 190
18 26 28 56 190 213
18 26 28 56 190 200 213
12 18 26 28 56 190 200 213
12 18 24 26 28 56 190 200 213
12 18 24 26 27 28 56 190 200 213
초기 BST 중위 순회: 12 18 24 26 27 28 56 190 200 213
================================
BST insert 연산 진행
================================
195 삽입 후 BST 중위 순회: 12 18 24 26 27 28 56 190 195 200 213
```

중위 순회 시에 키 값을 기준으로 오름차순으로 출력되는 이진 탐색 트리의 특성에 의하여, 단계별로 키 값이 오름차순 정렬된 상태로 잘 출력되는 것을 확인할 수 있습니다.

[BST 삭제 연산 구현]

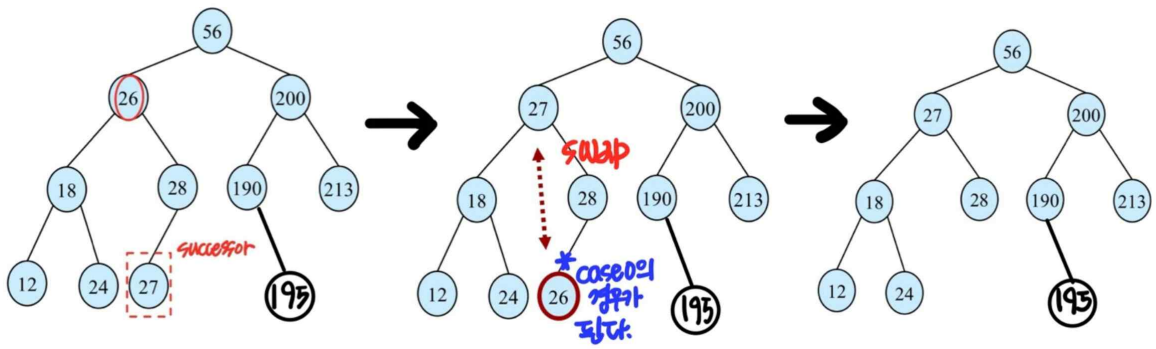
이진 탐색 트리의 삭제 연산은 삭제할 노드의 자식 노드의 개수에 따라 다음과 같은 의사코드가 짜여집니다.

case 0) 삭제할 노드의 자식 노드가 없는 경우
- 노드 삭제 진행

case 1) 삭제할 노드의 자식 노드 개수가 1개인 경우
- 삭제할 노드의 포인터가 해당 자식 노드를 가리키도록 만든 후 삭제 진행

case 2) 삭제할 노드의 자식 노드 개수가 2개인 경우
- 삭제할 노드의 successor을 찾는다.
- 해당 successor과 삭제할 노드의 자리를 뒤바꾼 후, case 0과 case 1로 만든다.

예를 들어, 키 값이 26인 노드를 삭제하는 연산을 생각해봅시다.
키 값이 26인 노드는 자식 노드의 개수가 2개인 case 2)에 해당합니다. 따라서, successor 노드(자신의 오른쪽 서브트리의 가장 작은 키 값을 가진 노드)인 27을 찾은 후 자리를 뒤바꾸어줍니다.
이후, 키 값이 26인 노드는 자식 노드가 없는 case 0)의 경우가 되므로 그대로 삭제 연산을 진행합니다.



```
bstreeNode* treeDelete(bstree* T, bstreeNode* z) {
    bstreeNode* y = NULL;
    /* Determine which node to splice out: either z
    if (z->left == NULL || z->right == NULL)
        y = z;
    else
        y = treeSuccessor(z);
    bstreeNode* x = NULL;

    /* set x to non-NIL child of x, or to NIL if y h
    // case 1) 한쪽 노드만 있는 경우
    if (y->left != NULL)
        x = y->left;
    else
        x = y->right;
    /* y is removed from tree by manipulating pointe
    if (x != NULL)
        x->p = y->p;

    if (y->p == NULL)
        T->root = x;
    else if (y == y->p->left)
        y->p->left = x;
    else
        y->p->right = x;

    if (y != z)
        z->key = y->key;

    free(y);
    return z;
}
```

[bst/bst.c]

```
printf("==============================₩n");
printf("BST delete 연산 진행₩n");
printf("==============================₩n");
// 키가 26인 노드 찾기
int k2 = 26;
bstreeNode* nodeToDelete = treeSearch(BST->root, k2);
if (nodeToDelete == NULL) {
    printf("키 값이 %d인 노드가 BST에 존재하지 않습니다.₩n", k2);
}
else {
    bstreeNode* deletedNode = NULL;
    deletedNode = treeDelete(BST, nodeToDelete);
    if (deletedNode == NULL) {
        printf("error caused when deleting the node₩n");
    }
    printf("키 값이 %d인 노드가 BST에서 삭제되었습니다.₩n", k2);
}

// 삭제 후 BST 출력
printf("Delete연산 후의 BST 중위 순회: ");
inOrderTreeWalk(BST->root);
printf("₩n");
```

[bst/main.c]

[출력 결과 화면]

```
==============================
BST delete 연산 진행
==============================
키 값이 26인 노드가 BST에서 삭제되었습니다.
Delete연산 후의 BST 중위 순회: 12 18 24 27 28 56 190 195 200 213
```

(3) 시간 복잡도 측면에서의 BST의 한계

이진 탐색 트리는 트리이므로 트리의 높이(h)에 비례하는 시간 복잡도를 가집니다. 균형 이진 트리일 때는 h=logn이므로 Θ(logn)의 시간 복잡도를 가지지만, 극단적인 나쁜 경우(편향 이진 트리일 때)에는 Θ(n)이라는 시간 복잡도를 가진다는 한계점이 있습니다.
상기 기술된 BST의 한계점을 극복하기 위해서는 트리가 '균형적'이어야 합니다. 편향되지 않는 상황인 '균형'을 맞추기 위한 특성을 가진 트리로, AVL 트리, B트리, 레드 블랙 트리가 있습니다.

이 중, 저는 레드 블랙 트리(RBT, red black tree)를 구현해보고자 합니다.

2. RBT(red-black tree, 레드 블랙 트리)의 삽입, 삭제 연산

(1) 레드 블랙 트리의 정의

레드 블랙 트리란, 이진 탐색 트리의 특성을 모두 가지되 트리를 구성하는 각 노드에 필드 값에 '색상'이라는 요소가 추가된 트리입니다.
레드 블랙 트리는 기본적으로 다음의 5가지의 특성을 만족시켜야 합니다:

1. 모든 노드는 레드 또는 블랙이다.
2. 루트 노드는 블랙이다.
3. 모든 센티넬 노드는 블랙이다.
4. 레드 노드는 그 자식 노드로 레드 노드를 가질 수 없다.
5. 모든 각 노드에 대하여, *해당 노드에서부터 센티넬 노드까지의 경로에서 발견되는 블랙 노드의 개수(이하 bh(x))* 는 모두 같다.(모든 노드 x의 bh(x)는 같다.)

```c
typedef enum NodeColor { RED, BLACK }Color;

typedef struct rbtreeNode {
    int key;
    Color color; // 색상 필드 추가
    struct rbtreeNode* left, * right, * p;
} rbtreeNode;

typedef struct {
    rbtreeNode* root;
    rbtreeNode* nil;
} rbtree;
```

[rbt/rbt.h]

레드 블랙 트리의 1번 특성[2] 을 반영하여, 레드 또는 블랙을 나타내는 색상을 열거형(enum)인 NodeColor로 정의하고, 레드 블랙 트리를 구성하는 노드의 구조체를 rbtreeNode로 정의하였습니다. 각 rbtreeNode는 bstreeNode에 추가된 필드인 Color 필드 정보값을 지닙니다.

레드 블랙 트리의 2번, 3번 특성[3]을 반영하여, rbtree 구조체를 정의하였습니다. rbtree는 레드 블랙 트리의 기본 정보값인 루트 노드와, 센티넬 노드(이하 nil)를 지니고 있습니다.

레드 블랙 트리의 5번 특성[4]에 의하여, 레드 블랙 트리는 균형을 맞춥니다. 균형을 맞추기 위한 연산으로는 노드를 기준으로 왼쪽으로 회전하는 leftRotate()연산, 오른쪽으로 회전하는 rightRotate()연산이 있습니다.

---

[2] 모든 노드는 레드 또는 블랙이다.

[3] 루트 노드, 센티넬 노드는 모두 블랙이다.

[4] 모든 노드 x의 bh(x)는 모두 같다.

(2) RBT의 회전, 삽입, 삭제 연산

[RBT 회전 연산]

왼쪽 회전 연산을 기준으로 설명드리겠습니다. x의 오른쪽 노드는 센티넬이 아닌 조건 하에 연산을 수행합니다:

x의 왼쪽 회전 연산은 x를 y의 왼쪽 자식 노드로 만들고, y의 왼쪽 서브트리를 x의 오른쪽 서브트리로 만드는 과정입니다.

오른쪽 회전 연산은 왼쪽 회전 연산의 코드에서 왼쪽, 오른쪽 방향만 바뀐 대칭적인 연산입니다.

```
/*      왼쪽 회전 연산      */
void leftRotate(rbtree* T, rbtreeNode* x) {
    rbtreeNode* y = x->right;
// y 설정: y는 x의 오른쪽 자식 노드
    x->right = y->left;
// y의 왼쪽 서브트리를 x의 오른쪽 서브트리로 만든다.

    if (y->left != T->nil)
// y의 왼쪽 자식 노드가 nil 노드가 아닌 내부 노드라면
        y->left->p = x;
// 부모로 x를 연결한다.

    y->p = x->p;
// x의 부모를 y로 연결한다.

    if (x->p == T->nil)
// CASE 1. 연결한 x의 부모가 nil 노드라면

// (레드블랙 트리 상에서의 루트)
        T->root = y;
// x가 루트인 경우, y가 새로운 루트가 된다.
    else if (x == x->p->left)
// CASE 2. x가 부모의 왼쪽 자식인 경우
        x->p->left = y;
    else
// CASE 3. x가 부모의 오른쪽 자식인 경우
        x->p->right = y;

    y->left = x;
// x를 y의 왼쪽 자식으로 만든다.
    x->p = y;
// x의 부모를 y로 설정한다.
}
```

```
/*      오른쪽 회전 연산      */
void rightRotate(rbtree* T, rbtreeNode* x) {
    rbtreeNode* y = x->left;
    x->left = y->right;

    if (y->right != T->nil)
        y->right->p = x;

    y->p = x->p;

    if (x->p == T->nil)
        T->root = y;
    else if (x == x->p->right)
        x->p->right = y;
    else
        x->p->left = y;

    y->right = x;
    x->p = y;
}
```

[삽입, 삽입 후 보정 연산]

RBT는 그 특성을 위반하지 않기 위해, 노드를 삽입하는 연산 수행 후에 트리의 구조를 조정하는 연산인 RB_Insert_FixUp을 호출하면서 삽입 연산을 수행합니다.

[rbt/rbt.c]

```
/*      레드 블랙 트리 삽입 연산     */
void RB_Insert(rbtree* T, rbtreeNode *z) {
    rbtreeNode* y = T->nil;
    rbtreeNode* x = T->root;
//x를 T의 루트 노드를 가리키는 포인터값으로 설정한다.

    /* 루트에서부터 쭉 내려오면서, BST 삽입 연산처럼 수행한다. */
    while (x != T->nil) {
        y = x;
        if (z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }

    z->p = y; //z의 부모를 설정한다.
    if (y == T->nil)
//CASE 1) z의 부모가 sentinel 노드라면(즉, z가 삽입될 자리가 루트)
        T->root = z;
    else if (z->key < y->key)
        y->left = z;
// CASE 2) z가 부모의 왼쪽 자식인 경우
    else
        y->right = z;
// CASE 3) z가 부모의 오른쪽 자식인 경우

    z->left = T->nil;
    z->right = T->nil;
    z->color = RED;
// 삽입되는 노드의 색깔은 항상 red로 설정
    RB_Insert_FixUp(T, z);
}
```

```
/* 레드-블랙 트리 삽입 후 보정 연산 */
void RB_Insert_FixUp(rbtree* T, rbtreeNode* z) {
    rbtreeNode* y;

    while (z->p != T->nil && z->p->color == RED) {
        if (z->p == z->p->p->left) {
 // 부모 노드가 조부모 노드의 왼쪽 자식이라면
            y = z->p->p->right;
// 삼촌 노드는 조부모 노드의 오른쪽 자식 노드
            /* CASE 1: 삼촌 노드 색이 레드인 경우 */
            if (y->color == RED) {
                z->p->color = BLACK;
// 부모 노드의 색깔을 블랙으로 만든다.
                y->color = BLACK;
// 삼촌 노드의 색깔도 블랙으로 만든다.
                z->p->p->color = RED;
// 조부모 노드의 색은 레드로 만든다.
                z = z->p->p;
// 두 칸 위로 거슬러 올라가 진행한다.
            }
            /* 삼촌 노드 색이 블랙인 경우 */
            else {
                if (z == z->p->right) {
// CASE 2. 삼촌 노드 색이 블랙이고, z가 오른쪽 자식인 경우
                    z = z->p;
// 부모 노드에 대해
                    leftRotate(T, z);
// 왼쪽 회전 연산 수행
                }
                // CASE 3. 삼촌 노드 색이 블랙이고, z가 오른쪽 자식인 경우
                z->p->color = BLACK;
// 부모노드의 색깔을 블랙으로 만들어준다.
                z->p->p->color = RED;
 // 조부모 노드 색깔을 레드로 만들어준다.
                rightRotate(T, z->p->p);
// 오른쪽으로 회전
            }
        }
        else {
// 부모 노드가 조부모 노드의 오른쪽 자식이라면
            y = z->p->p->left;
      // 삼촌 노드는 조부모 노드의 왼쪽 자식
```

```
    /* CASE 1: 삼촌 노드 색이 레드인 경우 */
            if (y->color == RED) {
                z->p->color = BLACK;
// 부모 노드의 색깔을 블랙으로 변경
                y->color = BLACK;
// 삼촌 노드의 색깔을 블랙으로 변경
                z->p->p->color = RED;
// 조부모 노드의 색깔을 레드로 변경
                z = z->p->p;
// 두 칸 위로 거슬러 올라가 진행한다.
            }
            /* 삼촌 노드 색이 블랙인 경우 */
            else {
                if (z == z->p->left) {
// CASE 2. 삼촌 노드의 색이 블랙이고, z가 왼쪽 자식인 경우
                    z = z->p;
// 부모 노드에 대해
                    rightRotate(T, z);
// 오른쪽 회전 연산 수행
                }
// CASE 3. 삼촌 노드 색이 블랙이고, z가 오른쪽 자식인 경우
                z->p->color = BLACK;
// 부모노드의 색깔을 블랙으로 만들어준다.
                z->p->p->color = RED;
// 조부모 노드 색깔을 레드로 만들어준다.
                leftRotate(T, z->p->p);
// 왼쪽 회전
            }
        }
    }
    T->root->color = BLACK;
// 루트 노드의 색깔은 항상 블랙
}
```

삽입 연산(RB_Insert)은 삽입되는 노드를 항상 레드로 설정하고, 기존의 RBT에 삽입합니다.
BST의 삽입 연산과 삽입하는 자리를 찾아가 자리를 정하여 삽입되는 알고리즘은 동일하지만, 노드를 삽입함으로써 레드 블랙 트리의 특성을 위반하는 상황이 발생할 수 있으므로, 삽입 후 보정 연산(RB_Insert_FixUp) 을 추가적으로 수행합니다.

삽입 후 보정 연산은 Re-Coloring, rotate 연산을 통하여 레드 블랙 트리의 특성을 유지하도록 합니다.

삽입 연산 시에는 '항상 삽입되는 노드는 레드 노드'라는 전제 때문에, RBT에서의 위 아래 레드 노드가 연속적으로 나타내는 문제가 발생할 수 있습니다.

1) 삽입되는 노드가 루트 노드일 때
    - RBT의 특성 2번 위반
2) 삽입되는 노드의 부모가 레드 노드일 때

・RBT의 특성 4번 위반

case 1) 삼촌 노드가 레드 노드일 때

・삼촌 노드와 부모 노드의 색깔을 블랙, 조부모 노드의 색깔을 레드로 바꾼다.

・조부모 노드에 대해서 보정 연산을 반복한다.

case 2) 삼촌 노드가 블랙 노드이고, 삽입되는 노드가 오른쪽 자식 노드일 때

・부모 노드를 기준으로 왼쪽 회전 연산(leftRotate)을 수행한다.

→ 삽입되는 노드가 왼쪽 자식 노드가 되고, 부모 노드와 자식 노드는 모두 레드

→ case 3)이 된다.

case 3) 삼촌 노드가 블랙 노드이고, 삽입되는 노드가 왼쪽 자식 노드일 때

・부모 노드를 블랙으로, 조부모 노드를 레드로 만든다.

・조부모 노드에 대해서 오른쪽 회전 연산(rightRotate)을 수행한다.

```c
rbtree* RBT = createRBTree(); // rbtree 구조체 초기화

RBT->nil = (rbtreeNode*)malloc(sizeof(rbtreeNode)); // nil 노드 초기화
RBT->nil->color = BLACK; // nil 노드는 항상 검정색
RBT->root = RBT->nil;

int rbt_nodeList[] = { 7, 3, 18, 10, 22, 8, 11, 26 };

for (int i = 0; i < 8; i++) {
    RB_Insert(RBT, createRBTreeNode(rbt_nodeList[i], RBT));
}

printf("₩n<원래의 RBT>₩n ");
inOrderRBTreeWalk(RBT->root, RBT);
```

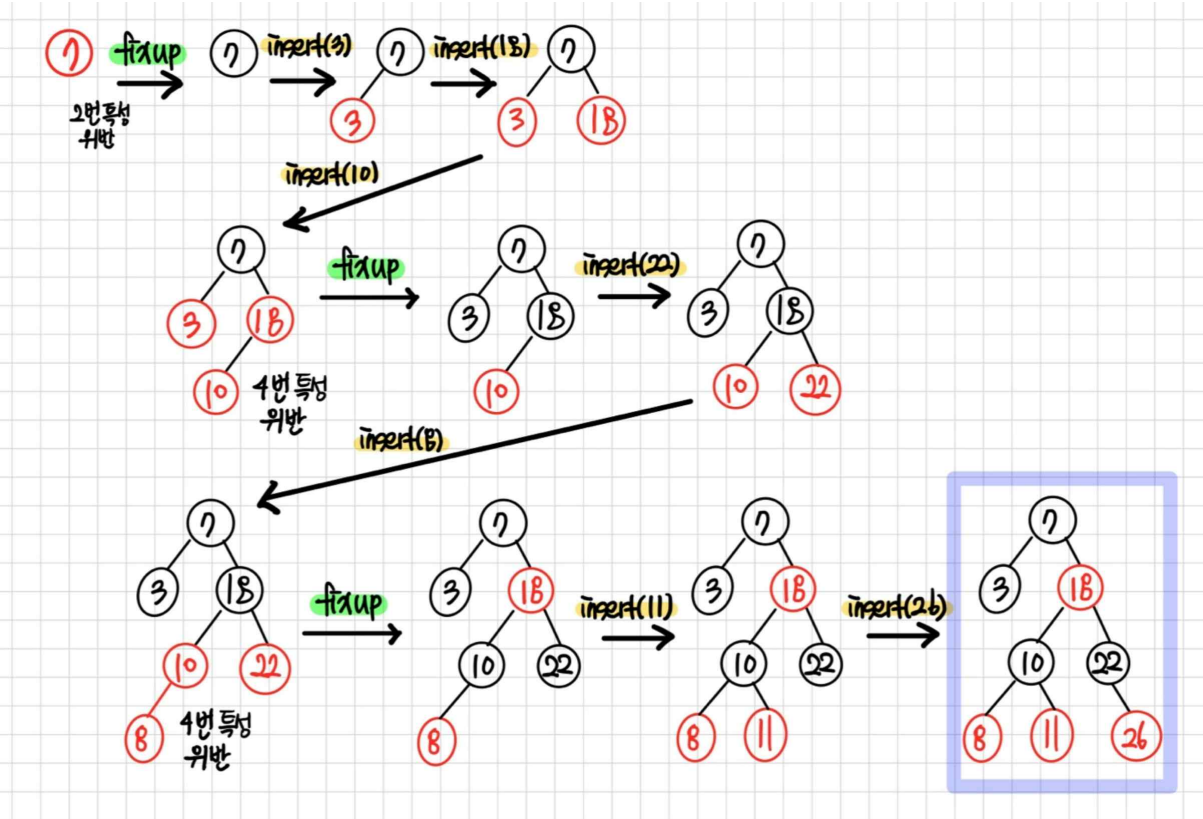
[rbt/main.c]

[출력 결과 화면]

```
<원래의 RBT>
 Node: 3, Color: BLACK, Parent: 7, LeftNode's key: nil, RightNode's key: nil
Node: 7, Color: BLACK, Parent: nil, LeftNode's key: 3, RightNode's key: 18
Node: 8, Color: RED, Parent: 10, LeftNode's key: nil, RightNode's key: nil
Node: 10, Color: BLACK, Parent: 18, LeftNode's key: 8, RightNode's key: 11
Node: 11, Color: RED, Parent: 10, LeftNode's key: nil, RightNode's key: nil
Node: 18, Color: RED, Parent: 7, LeftNode's key: 10, RightNode's key: 22
Node: 22, Color: BLACK, Parent: 18, LeftNode's key: nil, RightNode's key: 26
Node: 26, Color: RED, Parent: 22, LeftNode's key: nil, RightNode's key: nil
```

초기 레드 블랙 트리를 구축하기 위해, 레드 블랙 트리의 삽입 연산을 다음과 같은 샘플 데이터를 넣어 메인 함수로 테스트했습니다.

상기 코드가 작동하면서 초기 레드 블랙 트리가 구축되는 과정을 그림으로 그려 단계별로 표현해 보았습니다.



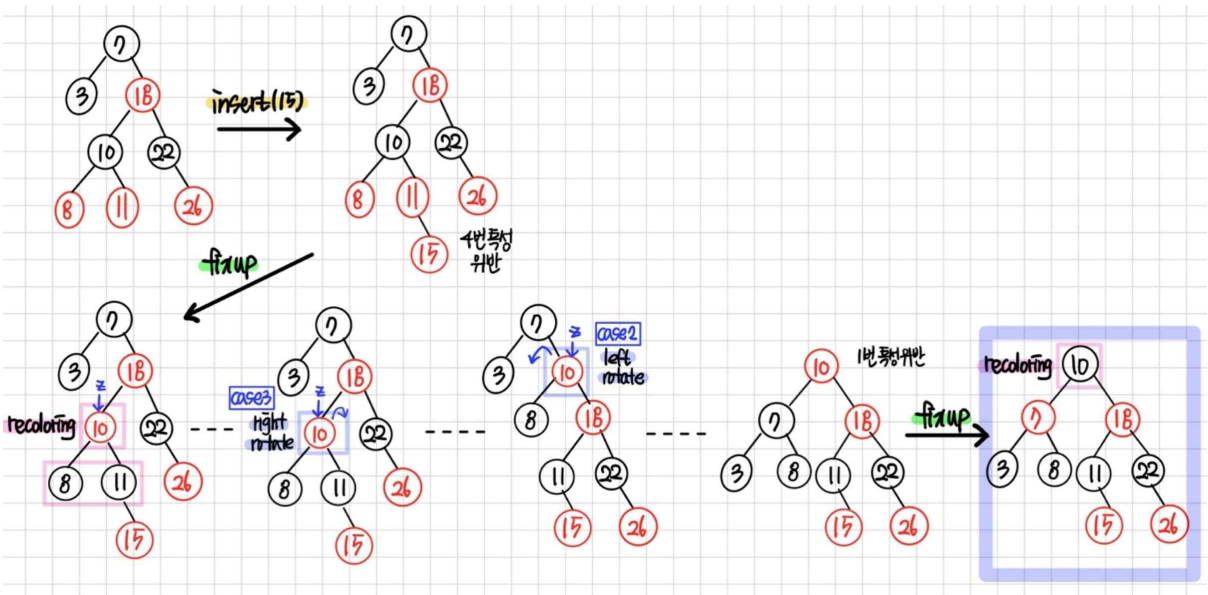
이제 구축된 초기의 레드 블랙 트리에 키 값이 15인 노드를 삽입해보겠습니다.

```
printf("==========================₩n");
// 키 값이 k1인 노드 삽입
int k1 = 15;
rbtreeNode* nodeToInsert = createRBTreeNode(k1, RBT);
RB_Insert(RBT, nodeToInsert);
printf("<키 값이 %d인 노드 삽입 후의 RBT> ₩n", k1);
inOrderRBTreeWalk(RBT->root, RBT);
```

[rbt/main.c]

[출력 결과 화면]

```
==========================
<키 값이 15인 노드 삽입 후의 RBT>
Node: 3, Color: BLACK, Parent: 7, LeftNode's key: nil, RightNode's key: nil
Node: 7, Color: RED, Parent: 10, LeftNode's key: 3, RightNode's key: 8
Node: 8, Color: BLACK, Parent: 7, LeftNode's key: nil, RightNode's key: nil
Node: 10, Color: BLACK, Parent: nil, LeftNode's key: 7, RightNode's key: 18
Node: 11, Color: BLACK, Parent: 18, LeftNode's key: nil, RightNode's key: 15
Node: 15, Color: RED, Parent: 11, LeftNode's key: nil, RightNode's key: nil
Node: 18, Color: RED, Parent: 10, LeftNode's key: 11, RightNode's key: 22
Node: 22, Color: BLACK, Parent: 18, LeftNode's key: nil, RightNode's key: 26
Node: 26, Color: RED, Parent: 22, LeftNode's key: nil, RightNode's key: nil
```



[삭제 연산]

우선, 강의자료의 수드 코드 레벨에서 분석해보았습니다.

```
RB-DELETE(T, z)
  y = z
  y-original-color = y.color
  if z.left == T.nil                          ← case1
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
  elseif z.right == T.nil                     ← case2
      x = z.left
      RB-TRANSPLANT(T, z, z.left)
  else y = TREE-MINIMUM(z.right)
      y-original-color = y.color
      x = y.right
      if y.p == z                             ← case3
          x.p = y
      else RB-TRANSPLANT(T, y, y.right)       ← case4
          y.right = z.right
          y.right.p = y
      RB-TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
      y.color = z.color
  if y-original-color == BLACK
      RB-DELETE-FIXUP(T, x)
```

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

- RB-Transplant

u: 대체될 노드 / v: u를 대체할 노드

u가 루트 노드라면, 루트 노드를 v로 설정

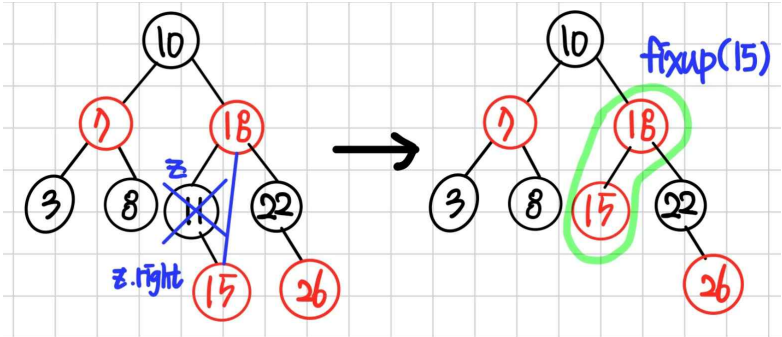
u가 부모 노드의 왼쪽 자식이면, u의 부모 노드의 왼쪽 자식 노드를 v로 설정

u가 부모 노드의 오른쪽 자식이면, u의 부모 노드의 오른쪽 자식 노드를 v로 설정

v의 부모 노드를 u의 부모 노드로 설정

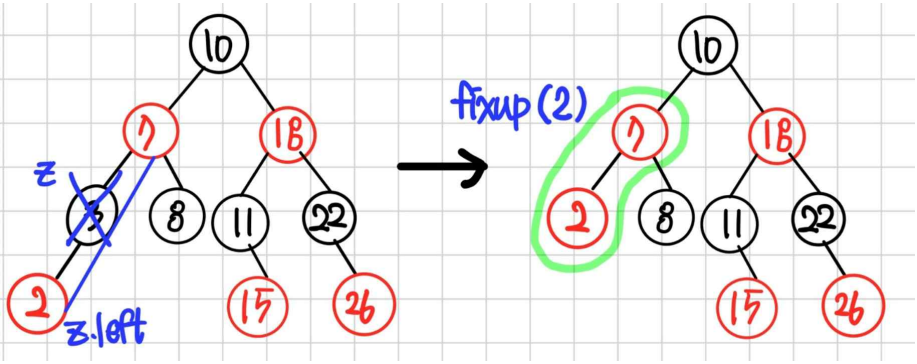
case 1) 삭제할 노드 z의 왼쪽 자식 노드가 없는 경우

- 삭제할 노드 z의 부모 노드와 삭제할 노드의 오른쪽 자식 노드를 연결한 후, 오른쪽 자식 노드에 대해서 보정 연산(fixUp) 수행



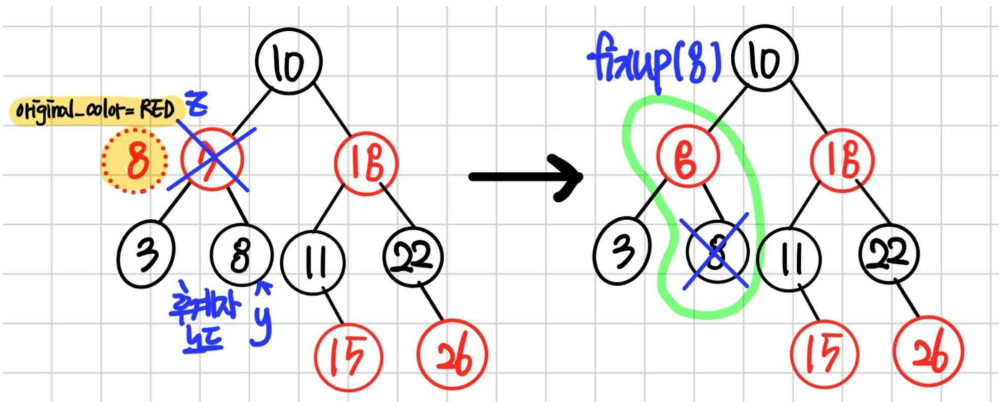
case 2) 삭제할 노드 z의 오른쪽 자식 노드가 없는 경우

- 삭제할 노드 z의 부모 노드를 삭제할 노드의 왼쪽 자식 노드와 연결한 후, 왼쪽 자식 노드에 대해서 보정 연산(fixUp) 수행



삭제할 노드 z가 양쪽에 자식 노드가 있다면,

case 3) 삭제할 노드 z의 자식 노드가 2개 있고, 후계자 노드가 z의 오른쪽 자식 노드가 아닐 때



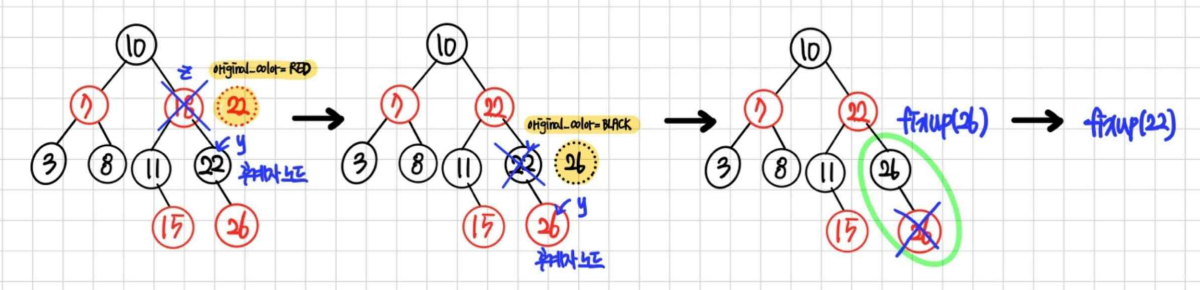
- 후계자 노드 찾기:
  - 삭제할 노드 z의 오른쪽 서브트리에서 가장 작은 값을 갖는 노드 y를 찾습니다.

- 후계자 노드의 부모 노드와 자식 노드 연결:
  - 후계자 노드 y의 부모 노드와 y의 오른쪽 자식을 연결합니다.

- 삭제할 노드 z위치에 후계자 노드 y대체:
  - 삭제할 노드 z의 부모 노드와 자식 노드들을 y와 연결합니다.
    y의 왼쪽 자식은 z의 왼쪽 자식이 되고, y의 오른쪽 자식은 z의 오른쪽 자식이 됩니다.

- 보정 연산 (fixUp) 수행
  - y가 원래 위치에서 이동했으므로, 이로 인한 트리의 불균형을 수정합니다.
    이는 y의 새로운 부모 노드에서부터 시작하여 루트까지 올라가면서, 필요한 보정 연산을 수행합니다.

case 4) 삭제할 노드 z의 자식 노드가 2개 있고, 후계자 노드가 z의 오른쪽 자식 노드일 때



- 후계자 노드 찾기:
  - 삭제할 노드 z의 오른쪽 서브트리에서 가장 작은 값을 갖는 노드 y를 찾습니다.

- 후계자 노드의 부모 노드와 자식 노드 연결:
  - 후계자 노드 y의 부모 노드와 y의 오른쪽 자식을 연결합니다.

- 삭제할 노드 z위치에 후계자 노드 y대체:
  - 삭제할 노드 z의 부모 노드와 자식 노드들을 y와 연결합니다.
    y의 왼쪽 자식은 z의 왼쪽 자식이 되고, y의 오른쪽 자식은 z의 오른쪽 자식이 됩니다.

- 보정 연산 (fixUp) 수행
  - y가 원래 위치에서 이동했으므로, 이로 인한 트리의 불균형을 수정합니다.
    이는 y의 새로운 부모 노드에서부터 시작하여 루트까지 올라가면서, 필요한 보정 연산을 수행합니다.

[삭제 후 보정 연산]

삭제 연산에서 후계자 노드 y가 레드인 경우,
블랙 높이인 bh(x)는 모두 유지되므로 특성 5를 위반하지 않고, 인접한 레드 노드를 가지는 경우가 발생할 수 없으므로 특성 4를 위반하지 않습니다. 또한, 후계자 노드가 루트 노드인 경우는 발생하지 않으므로 특성 2 또한 보장됩니다.

삭제 연산에서 후계자 노드 y가 블랙인 경우, 레드 블랙 트리의 특성 위반이 될 수 있습니다. 따라서 삭제 후 보정 연산(RB_Delete_FixUp)으로 레드 블랙 트리의 특성을 보존해야 합니다.

- 삭제할 노드 x가 루트이거나, 레드 노드라면 x를 블랙으로 recolor하고 리턴
- 삭제할 노드 x가 루트가 아니고, 블랙 노드라면 삭제할 노드가 부모의 왼쪽/오른쪽 자식인지에 따라 4개의 case로 분기됩니다.

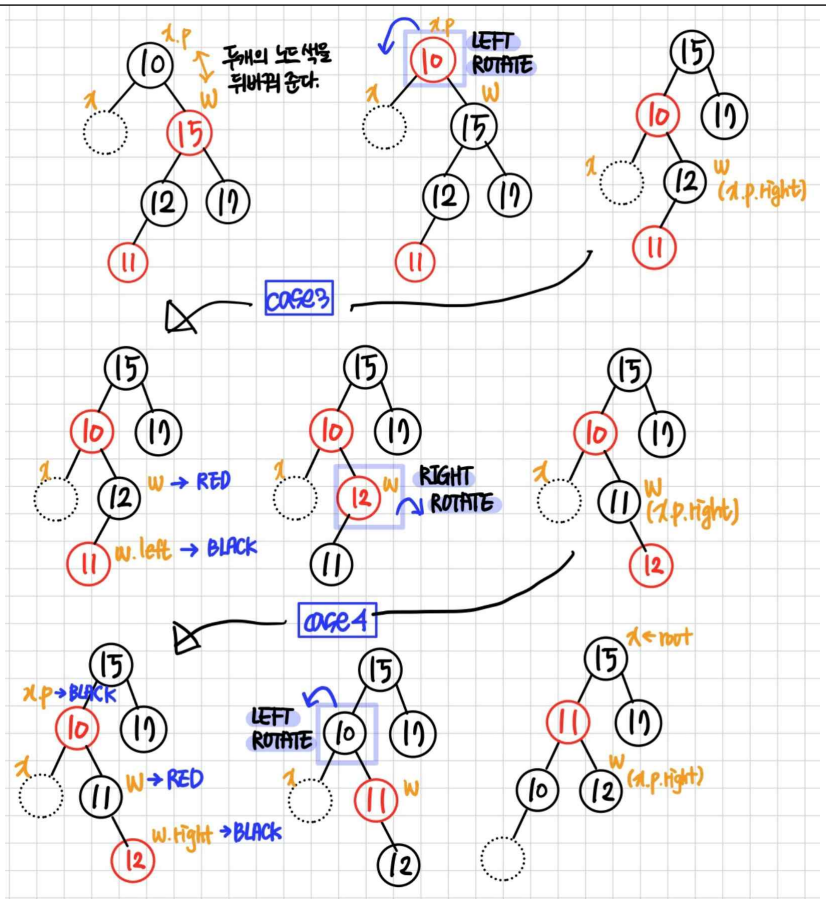
```
RB-DELETE-FIXUP(T, x)
  while x ≠ T.root and x.color == BLACK
      if x == x.p.left
          w = x.p.right
          if w.color == RED
              w.color = BLACK                          // case 1
              x.p.color = RED                          // case 1
              LEFT-ROTATE(T, x.p)                      // case 1
              w = x.p.right                            // case 1
          if w.left.color == BLACK and w.right.color == BLACK
              w.color = RED                            // case 2
              x = x.p                                  // case 2
          else if w.right.color == BLACK
                  w.left.color = BLACK                 // case 3
                  w.color = RED                        // case 3
                  RIGHT-ROTATE(T, w)                   // case 3
                  w = x.p.right                        // case 3
              w.color = x.p.color                      // case 4
              x.p.color = BLACK                        // case 4
              w.right.color = BLACK                    // case 4
              LEFT-ROTATE(T, x.p)                      // case 4
              x = T.root                               // case 4
      else (same as then clause with "right" and "left" exchanged)
  x.color = BLACK
```

case 1) x의 형제 노드 w가 레드 노드일 때(w의 자식 노드 2개는 모두 블랙)

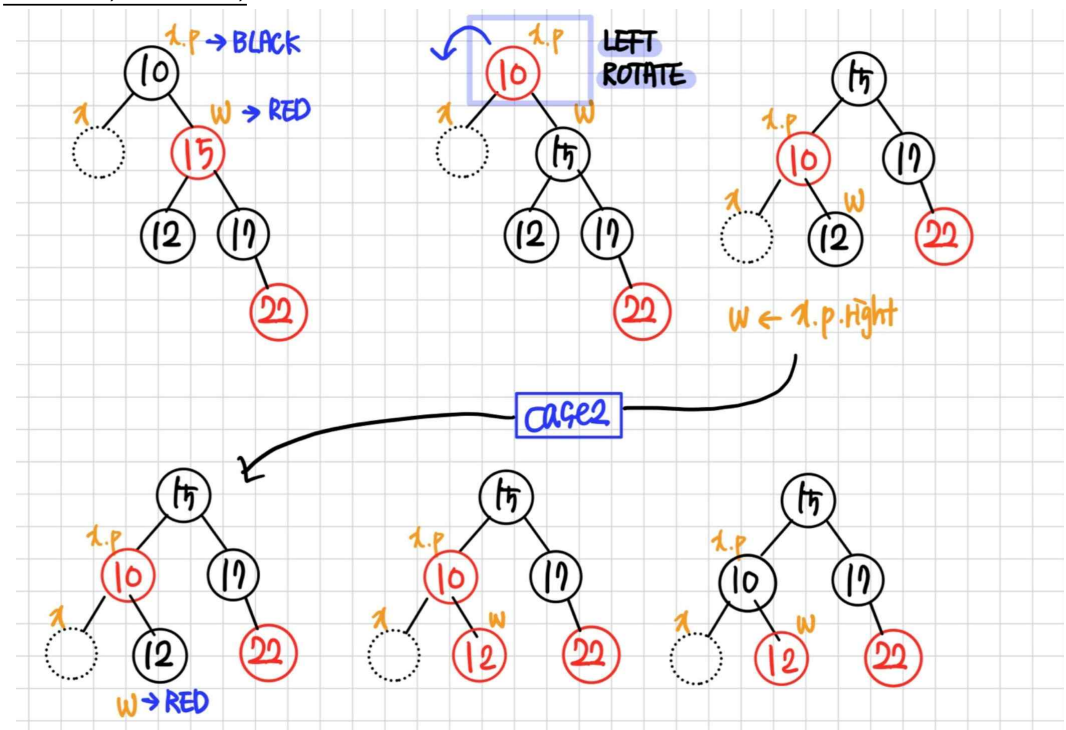
- 형제 노드 w를 블랙으로, 부모 노드를 레드로 바꾼다.(부모와 컬러를 바꾼다.)
- 부모 노드를 기준으로 왼쪽 회전 연산(leftRotate)를 수행한다.
- 경우가 바뀌면서 w가 재설정된다.(부모 노드의 오른쪽 자식 노드)

  → leftRotate로 인해 case 2), case 3), case 4)중 하나로 뒤바뀐다.

※ case 1) → case 3) → case 4)



※ case 1) → case 2)



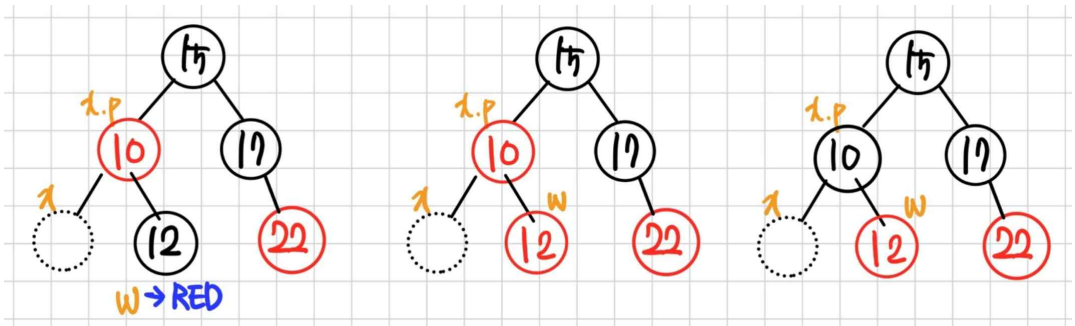
case 2) x의 형제 노드 w가 블랙 노드이고, w의 자식 노드 2개가 다 블랙인 경우

- 형제 노드 w를 레드로 바꿔준다.
- w의 기존 색상이었던 블랙을 부모 노드의 색상으로 만들어주고, 부모 노드로 x를 재설

정한다.

• 부모의 색상이 레드였다면, 블랙이 된 부모 노드는 레드-블랙(red-black) 트리의 특성을 만족하므로 종료한다.

• 부모의 색상이 블랙이었다면, 블랙이 된 부모 노드는 블랙-블랙(doubly black)이 되며 case 1, case 2, case 3, case 4 중 하나로 뒤바뀐다. 경우가 바뀌며 해당 부모 노드가 새로운 w가 된다.



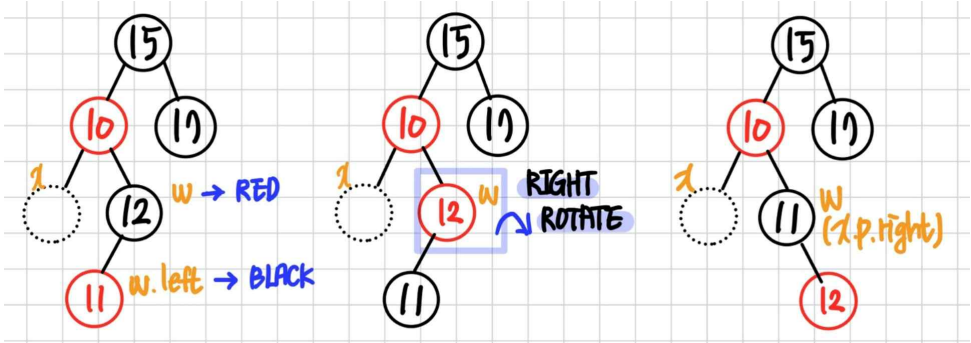
case 3) x의 형제 노드 w가 블랙 노드이고, w의 왼쪽 자식 노드는 레드/w의 오른쪽 자식 노드는 블랙일 때

• 형제 노드 w의 왼쪽 자식을 블랙, w 본인은 레드로 만들어준다.

• w를 기준으로 오른쪽 회전 연산(rightRotate)를 해준다.

• 경우가 바뀌면서, w가 재설정된다.(부모 노드의 오른쪽 자식 노드)

→ rightRotate로 인해 case 4)로 바뀐다.

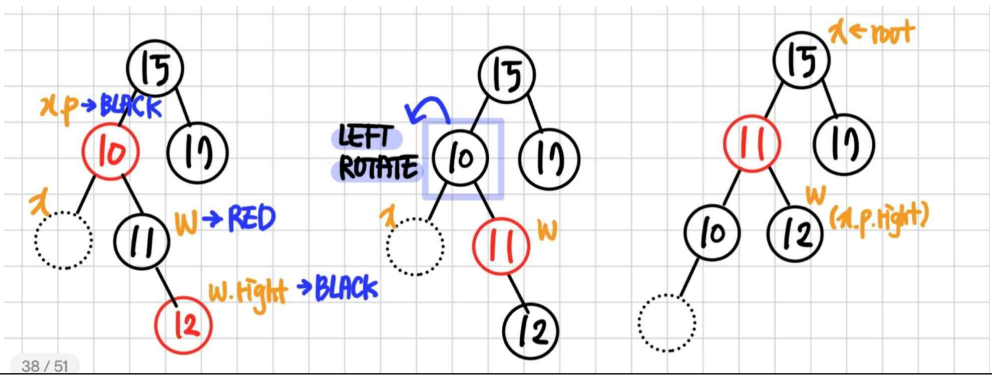


case 4) x의 형제 노드 w가 블랙 노드이고, w의 오른쪽 자식 노드가 레드일 때

• 형제 노드 w를 부모 노드의 컬러로 바꾼 후, 부모 노드와 w의 오른쪽 자식 노드의 컬러를 블랙으로 바꾼다.

• w의 부모 노드를 기준으로 왼쪽 회전 연산(leftRotate)를 해준다.

• x가 루트가 되면서 반복문이 종료된다.



```
void RB_Transplant(rbtree* T, rbtreeNode* u,
rbtreeNode* v) {
    if (u->p == T->nil) {
// 삭제될 노드가 root 인 경우
        T->root = v;
    }
    else if (u == u->p->left) {
        u->p->left = v;
    }
    else {
        u->p->right = v;
    }
    v->p = u->p;
}
```

```
void RB_Delete(rbtree* T, rbtreeNode* z) {
    rbtreeNode* y = z;
    rbtreeNode* x;
    Color y_original_color = y->color;

    if (z->left == T->nil) {
        x = z->right;
        RB_Transplant(T, z, z->right);
    }
    else if (z->right == T->nil) {
        x = z->left;
        RB_Transplant(T, z, z->left);
    }
    else {
        y = rbtreeMin(T, z->right);
        y_original_color = y->color;
        x = y->right;
        if (y->p == z) {
             x->p = y;
        }
        else {
            RB_Transplant(T, y, y->right);
            y->right = z->right;
            y->right->p = y;
        }
        RB_Transplant(T, z, y);
        y->left = z->left;
        y->left->p = y;
        y->color = z->color;
    }
    free(z);
    if (y_original_color == BLACK)
        RB_Delete_FixUp(T, x);
}
```

```
/*      RBT 삭제 후 보정 연산    */
void RB_Delete_FixUp(rbtree* T, rbtreeNode* x) {
    while (x != T->nil && x != T->root && x->color == BLACK) {
        if (x == x->p->left) {              // 삭제될 x가 왼쪽 자식인 경우
            rbtreeNode* w = x->p->right;   // w는 x의 형제 노드라고 가정하자
            /* CASE 1: x의 형제 w가 레드인 경우 */
            if (w->color == RED) {
                w->color = BLACK;           // 형제 노드의 색깔을 블랙으로 바꾼다.
                x->p->color = RED;          // 부모 노드의 색깔을 레드로 바꾼다.
                leftRotate(T, x->p);        // 부모 노드에 대해 왼쪽 회전 연산 수행
                w = x->p->right;            // 새로운 형제를 설정한다.
            }

            /* CASE 2: x의 형재 w가 블랙이고, w의 두 자식 노드도 모두 블랙인 경우 */
            if (w->left->color == BLACK && w->right->color == BLACK) {
                w->color = RED;             // 형제 노드의 색깔을 레드로 바꾼다.
                x = x->p;                   // x를 x의 부모 노드로 설정한다.
            }
            else {
```

```
                /* CASE 3: x의 형제 w가 블랙이고, w의 왼쪽 자식 노드는 레드, 오른쪽 자식 노
드는 블랙인 경우 */
                if (w->right->color == BLACK) {
                    w->left->color = BLACK;      // w의 왼쪽 자식 노드 색깔을 블랙으로 바꾼
다.
                    w->color = RED;             // w의 색깔을 레드로 바꾼다.
                    rightRotate(T, w);          // w에 대해 오른쪽 회전 연산 수행
                    w = x->p->right;              // 이제 형제 노드는 x의 부모 노드의 오른쪽
자식 노드
                }

                /* CASE 4: x의 형제 w가 BLACK이고 w의 오른쪽 자식이 RED인 경우 */
                w->color = x->p->color;         // 형제 노드를 부모의 색으로 바꾼다.
                x->p->color = BLACK;            // 부모 노드를 BLACK으로 바꾼다.
                w->right->color = BLACK;         // 형제의 오른쪽 자식을 BLACK으로 바꾼다.
                leftRotate(T, x->p);            // 부모 노드에 대해 좌회전 수행
                x = T->root;
            }
        }
        else {                                  // 삭제될 x가 오른쪽 자식인 경우
            rbtreeNode* w = x->p->left;     // 형제 노드 w는 왼쪽 자식 노드

            /* CASE 1: x의 형제 w가 레드인 경우 */
            if (w->color == RED) {
                w->color = BLACK;           // 형제 노드의 색깔을 블랙으로 만들어준다.
                x->p->color = RED;          // 부모 노드의 색깔을 레드로 만든다.
                rightRotate(T, x->p);       // 부모 노드에 대해 오른쪽 회전 연산 수행
                w = x->p->left;              // 설정된 새로운 형제는 부모 노드의 왼쪽 자식 노
드
            }

            /* CASE 2: x의 형제 w가 블랙이고, w의 두 자식 노드도 모두 블랙인 경우 */
            if (w->right->color == BLACK && w->left->color == BLACK) {
                w->color = RED;
                x = x->p;
            }
            else {
                /* CASE 3: x의 형제 w가 블랙이고, w의 오른쪽 자식 노드는 레드, 왼쪽 자식 노
드는 블랙인 경우 */
                if (w->left->color == BLACK) {
                    w->right->color = BLACK;
                    w->color = RED;
                    leftRotate(T, w);
                    w = x->p->left;
                }

                /* CASE 4: x의 형제 w가 블랙이고, w의 왼쪽 자식이 레드인 경우 */
                w->color = x->p->color;
                x->p->color = BLACK;
                w->left->color = BLACK;
                rightRotate(T, x->p);
                x = T->root;
            }
        }
    }
```

```
    x->color = BLACK;
}
```