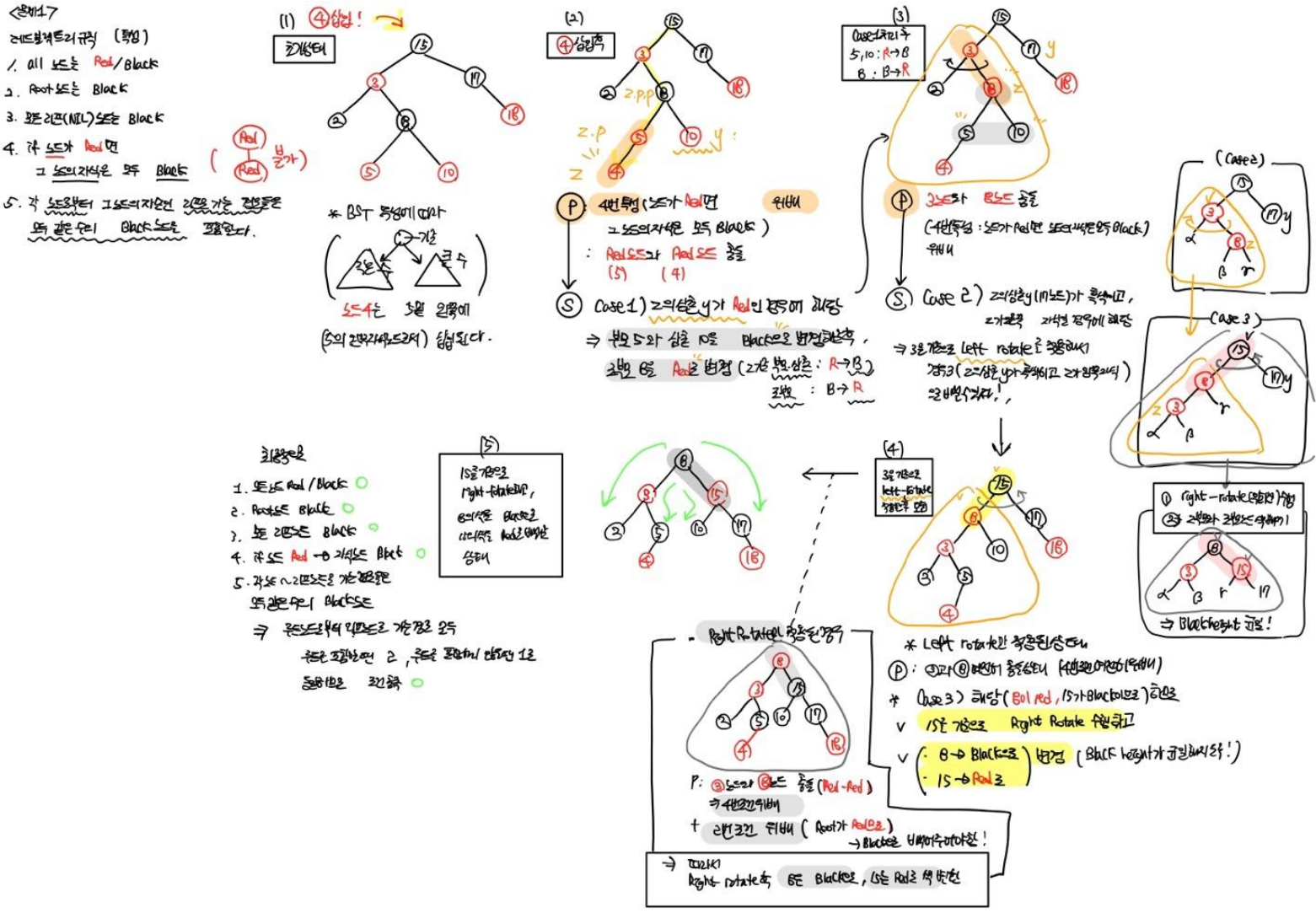
알고리즘 과제 보고서 8

2022110151 이주연

문제 1. 레드 블랙 트리에 4를 삽입할 때, 레드블랙 특성을 유지하기 위해 거치는 단계를 그림으로 표현하기

- 트리 노드의 색이나 위치가 변할 때 단계별 그림 및 변하는 이유 설명



레드블랙 트리에 새로운 노드를 삽입할 때, 삽입되는 노드는 Red 색상인데 이때, 5가지 특성 중 2번 "루트는 흑색이다" 특성과 4번 "노드가 적색이면 그 노드의 자식은 모두 흑색이다"라는 2가지 특성을 조심해야 합니다.

먼저 초기 상태(그림1)에서 4(R) 노드가 삽입되면 BST 특성에 따라 루트 노드를 기준으로 루트 값보다 작을 경우 왼쪽 서브 트리로, 루트 값보다 클 경우 오른쪽 서브 트리로 이동해 삽입되게 됩니다. 4의 경우 15보다 작고, 3보다 크며, 8보다 작고, 5보다 작으므로 5의 왼쪽 자식 노드로 삽입됩니다.

4번 노드가 삽입된 후(그림2) 4번 노드와 5번 노드 모두 적색 노드로 4번 특성

2022110151 이주연

(노드가 적색이면 그 노드의 자식은 모두 흑색이여야 한다)을 위배하게 됩니다. 4번 특성을 위배할 때 3가지의 케이스를 구분해 이에 따라 다른 조치를 취할 수 있는데, 그림 2의 경우 4를 z로 볼 때 4의 삼촌인 10이 적색인 경우 (case1)이기 때문에, z(4)를 기준으로 부모,삼촌 노드인 5와 10을 흑색으로, 조부모 노드를 적색으로 바꿉니다.

그리하여 그림3 모습이 되면, 이번에는 3과 8이 둘다 적색 노드로 충돌해 4번 특성을 또 위반하게 됩니다. 이 경우, 8(z)를 기준으로 삼촌 노드 17이 흑색이고, 8이 오른쪽 자식에 해당하므로 case2가 됩니다. Case2는 left-rotate를 해서 case3 모습으로 변환시킨 뒤, right-rotate를 하고 z의 부모와 조부모 노드의 색을 바꾸는 방식을 통해 최종적으로 레드블랙 트리의 특성을 유지할 수 있습니다.

따라서 3을 기준으로 left-rotate를 하게 되면, 그림4 형태로 나타나게 됩니다. 그림4는 case2(z의 삼촌 y가 흑색이고 z가 오른쪽 자식)에서 case3(z의 삼촌 y가 흑색이고 z가 왼쪽 자식일 경우)로 변환된 형태로, 3과 8이 둘다 적색으로 위배된 4번 조건을 충족시키 위해서는 right방향으로 다시 rotate해 처리해야 합니다.

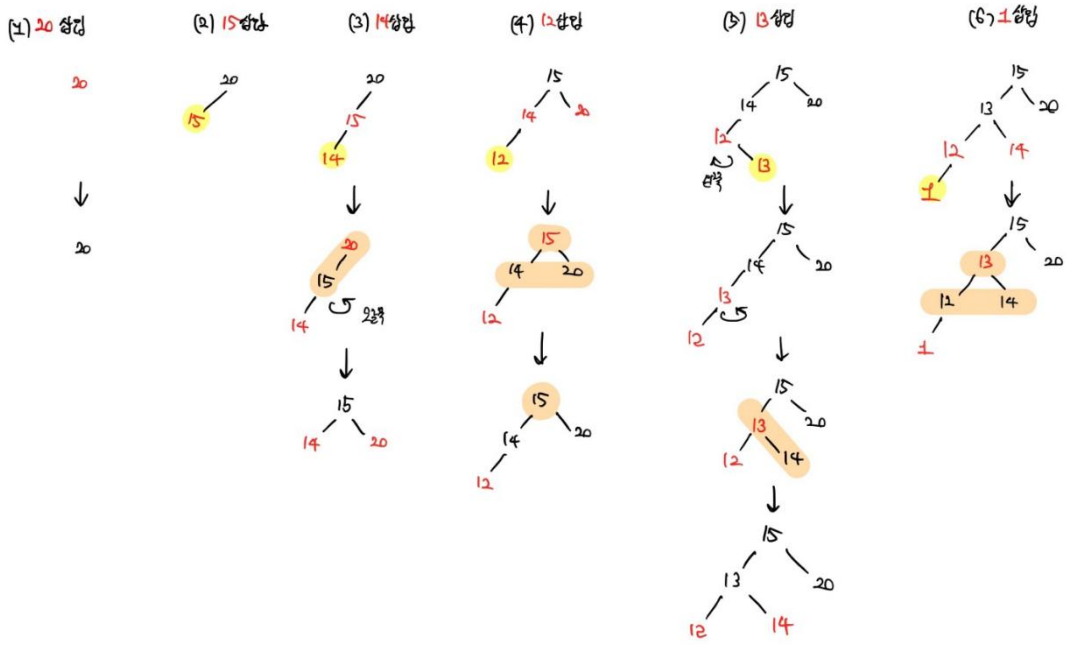
Right-rotate를 적용하면 사진과 같이 8이 루트 노드로 가면서 2번 특성 "root 노드는 흑색이어야 한다"를 위배하게 되므로 z부모와 조부모 노드들의 색을 바꿔(8: red-> black, 15: black->red) 최종적으로 그림5가 도출됩니다.

그림 5를 점검해 보았을 때 1~4번 특성 모두 충족하고, 루트노드로부터 리프 노드로 가는 모든 경로 black-height도 동일해 5번 특성까지 모두 충족하므로, 이와 같이 결과 그림을 그렸습니다.

알고리즘 과제 보고서 8

2022110151 이주연

문제 2. 빈 레드블랙 트리에 키 20, 15, 14, 12, 13, 1을 차례대로 삽입했을 때 만들어지는 레드블랙 트리의 모양을 보이기 – 삽입 시 변경되는 트리 모양 표현 및 이유 설명



레드블랙 트리의 black-height 유지를 위해 삽입 노드는 초기값이 빨간색입니다. 먼저 20®가 삽입되면, 2번 특성이 위배되기 때문에 루트 색을 검정색으로 바꿔줍니다. 그 후 15가 삽입되는데, 위배된 특성이 없으므로 14를 또 삽입해줍니다.

14가 삽입된 후 14와 그의 부모 노드인 15 모두 적색으로 4번 특성에 위배됩니다. 이 경우 부모인 15의 색을 검정색으로, 조부모인 20의 색을 빨간색으로 바꾼 후, 오른쪽 회전(right rotation)을 통해 4번 특성과 2번 특성을 모두 회복해줍니다.

다음으로 12가 14의 왼쪽 자식으로 삽입되면, 4번 특성에 또다시 위배됩니다. 이 경우 삼촌(20)이 Red로 case1에 해당해 부모(14)와 삼촌(20)을 검정색으로 바꿔준 후, 조부모인 15의 색을 적색으로 바꿔줍니다. 다만 이경우 2번 특성을 위배하므로 root를 검정색으로 바꿔줍니다.

13은 12의 오른쪽 자식으로 삽입되고, 삼촌 또한 검정색이므로 case2에 해당됩니다. 따라서 12를 기준으로 왼쪽으로 회전시켜 case3로 바꿔준 후, 오른쪽으로 회전시켜 12-13-14가 균형을 유지하도록 설정해줍니다. 그후 부모와 조부모의 색을 바꿔 4번 특성을 회복해줍니다.

1번을 12의 왼쪽 자식에 들어가게 되는데, 이때 삼촌의 색이 빨간색이므로 case1에 해당돼 부모,삼촌, 조부모의 색을 바꿔줍니다. 루트노드에서 리프노드로 가는 흑색 경로가 모두 동일하고 레드블랙 트리의 모든 특성을 충족하므로 최종적으로 레드블랙트리가 완성됩니다.

알고리즘 과제 보고서 8

2022110151 이주연

문제 3. 레드블랙 트리 삽입 구현

- 문제 2에서 삽입한 키들과 각 **삽입 과정에서 변경되는 트리의 모양 출력**

1) 20 삽입

```
>>>>>>>>>>>>>>>>>[ 삽입할 노드: 20 ]<<<<<<<<<<<<<<<<<

------------------ 20(R) 삽입 후 -----------------

└── 20(R)
-----------------------------------------------------
RB 특성 복구(fixup) 시작! - 노드 20(R)

-> 규칙 2: 루트 노드의 색상을 BLACK으로 바꿔주겠습니다.

>>>>>>>>>>[ 20 삽입이 완료되었습니다! ]<<<<<<<<<<

------------------ 20 삽입 후 최종 트리 -----------------

└── 20(B)
-----------------------------------------------------
```

문제 2번에서 그린 바와 같이 빈 트리에 20®가 삽입되고, root노드는 검정색이어야 한다는 2번 특성을 위배해 이를 검정색으로 바꿔준 결과가 나타났습니다.

2) 15 삽입

```
>>>>>>>>>>>>>>>>>[ 삽입할 노드: 15 ]<<<<<<<<<<<<<<<<<

------------------ 15(R) 삽입 후 -----------------

└── 20(B)
     └── 15(R)
-----------------------------------------------------
RB 특성 복구(fixup) 시작! - 노드 15(R)

-> 규칙 2: 루트 노드의 색상을 BLACK으로 바꿔주겠습니다.

>>>>>>>>>>[ 15 삽입이 완료되었습니다! ]<<<<<<<<<<

------------------ 15 삽입 후 최종 트리 -----------------

└── 20(B)
     └── 15(R)
-----------------------------------------------------
```

15®는 20(B)의 왼쪽 자식으로 삽입됩니다. 이 경우에는 위배된 특성은 없습니다.

알고리즘 과제 보고서 8

2022110151 이주연

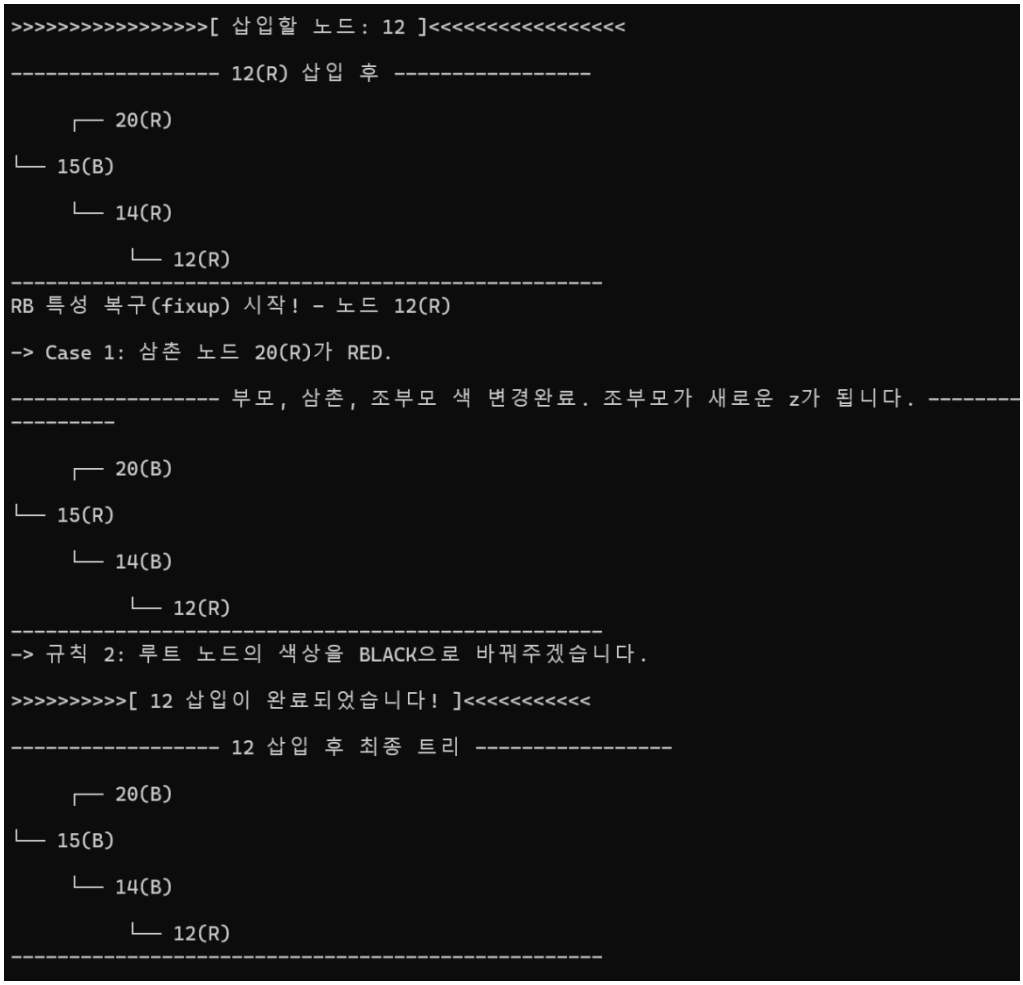
3) 14 삽입

```
>>>>>>>>>>>>>>>>[ 삽입할 노드: 14 ]<<<<<<<<<<<<<<<<
------------------ 14(R) 삽입 후 -----------------
└── 20(B)
    └── 15(R)
        └── 14(R)
--------------------------------------------------
RB 특성 복구(fixup) 시작! - 노드 14(R)
-> Case 3 : 부모와 조부의 색을 바꾼 후, rightRotate를 진행하겠습니다.
------------------ 부모와 조부모 색 변경 완료 -----------------
└── 20(R)
    └── 15(B)
        └── 14(R)
--------------------------------------------------
-> 노드 20(R)에 대해 Right Rotate 진행합니다.
------------------ Right Rotate 후 -----------------
    ┌── 20(R)
└── 15(B)
    └── 14(R)
--------------------------------------------------
-> 규칙 2: 루트 노드의 색상을 BLACK으로 바꿔주겠습니다.
>>>>>>>>>>[ 14 삽입이 완료되었습니다! ]<<<<<<<<<<
------------------ 14 삽입 후 최종 트리 -----------------
    ┌── 20(R)
└── 15(B)
    └── 14(R)
--------------------------------------------------
```

14는 15의 왼쪽 자식으로 삽입됩니다. 이때 4번 특성(Red노드의 모든 자식은 Black)을 위반하므로 반복문에 들어가 특성 복구 과정이 진행됩니다. 이 경우는 Case 3에 해당해 부모와 조부 색을 바꾼 후, right rotate가 진행됐습니다. 그림으로 그릴 때에는 rotate를 먼저 한 후 색을 바꿨지만, 코드 상에서는 먼저 색을 바꾼 후 rotate함수를 호출해 진행하는 방식이 더욱 편리하기 때문에 순서를 바꿔 진행했습니다.

2022110151 이주연

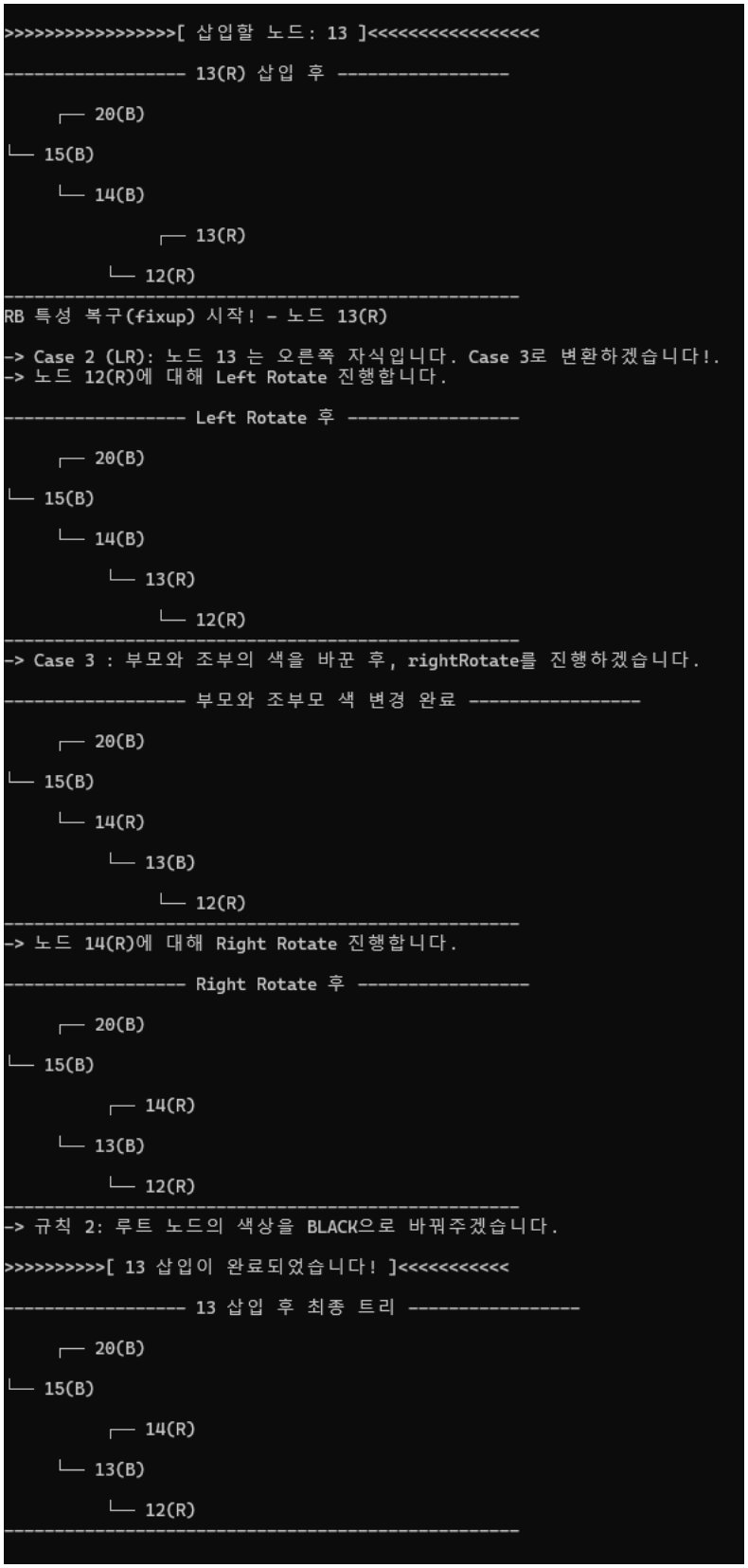
4) 12 삽입



12는 14의 왼쪽 자식으로 삽입됩니다. 이때에도 4번 특성을 위반하므로 특성 복구 과정에 들어가 케이스를 분석합니다. 삼촌노드가 red므로 case1에 해당하고, 부모,삼촌, 조부모의 색을 바꿔줍니다. 그러나 이 경우, 루트 노드가 빨간색으로 2번 특성에 위배되므로 루트 노드의 색상을 검정색으로 또다시 바꿔줍니다.

2022110151 이주연

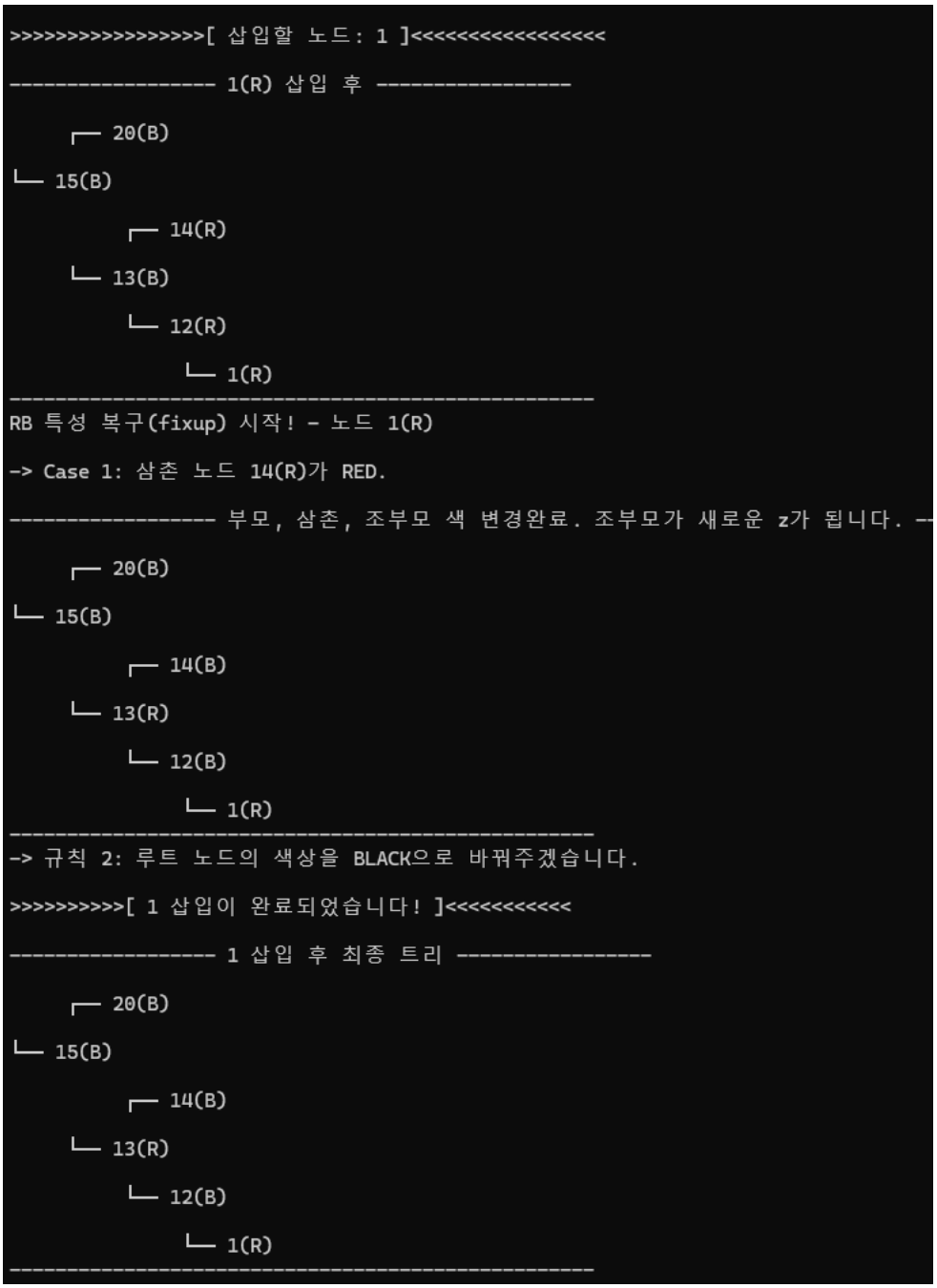
5) 13 삽입



13은 12의 오른쪽 자식으로 삽입됩니다. 이때에도 4번 특성을 위반하므로 특성 복구

를 시작해, 삼촌 노드가 검정색이고 13은 오른쪽 자식인 case2이므로, 이를 left rotate를 통해 case3번으로 바꿔줍니다. 그 후 부모와 조부모의 색을 변경한 후 rightRotate를 진행해 트리가 균형잡히도록 설정해주었습니다.

6) 1 삽입



1은 12의 왼쪽 자식으로 들어가고 4번 특성을 또다시 위배하게 됩니다. (R-R) 이 경우 삼촌 노드가 Red 이므로 단순히 부모, 삼촌 조부모의 색을 바꿔주면 됩니다.

알고리즘 과제 보고서 8

2022110151 이주연

7) 최종 레드 블랙 트리 모습

```
============= <최종 레드-블랙 트리 구조> =============

------------------ 레드블랙 트리 최종 구조 ---------------

      ┌── 20(B)
└── 15(B)
          ┌── 14(B)
      └── 13(R)
          └── 12(B)
                  └── 1(R)
-----------------------------------------------------
```

따라서 레드 블랙 트리는 다음과 같이 출력되는 모습을 볼 수 있었습니다.

- 알고리즘 간단 설명

```
// 2022110151 이주연
// 3. 레드블랙 트리 구현
//  [ 함수 구성 ]
//1. 데이터 구조 정의 (1)노드 2)레드블랙 트리 구조체 정의)
//2. 노드 & 트리 생성 함수
//3. 트리 그림으로 출력하는  함수
//4. 회전 함수 ( left rotate, right rotate)
//5. 레드 블랙 특성 복구 함수 (fix-up)
//6. 노드 삽입 (Insertion)
//7. 메모리 해제 (Free Memory)
//8. 메인 함수 (Main)
```

레드 블랙 트리는 크게 8가지 함수로 구현되어 있습니다. 그 중 핵심 함수인 회전 함수와 레드블랙 특성 복구 함수는 다음과 같습니다.

```
// 4. 회전 함수 : 1) 왼쪽(left)으로 회전.
// e.g. B노드 기준 왼쪽으로 회전 (left Rotate)    leftRotate(T,x)
//          x                        y
//        /  \                     /  \
//       a    y          =>       x    r
//           /  \                 /  \
//          b    r               a    b
void leftRotate(RedBlackTree *tree, Node *x) {
    // : 노드x를 기준으로 트리 T를 왼쪽으로 회전
    // Left Rotating 중이라는 메시지 출력
    char colorRB;
    if (x->color == RED){
        colorRB = 'R';
    } else {
        colorRB = 'B';
    }
    printf("-> 노드 %d(%c)에 대해 Left Rotate 진행합니다. \n", x->key, colorRB);

    // 1) 회전할 노드 X의 오른쪽 자식을 y로 설정
    Node *y = x->right;

    // 2) y의 왼쪽 서브트리(b)를 -> x의 오른쪽 서브트리(b)로 옮겨주기
    x->right = y->left;

    // 3) y의 왼쪽 자식이 NIL이 아니면 부모를 x로 설정하기
    if (y->left != tree->NIL) {
        y->left->parent = x;
    }
    // 4) y의 부모를  x의 부모로 설정하기
    y->parent = x->parent;
    //   a) 만약 x가 트리였으면 -> y가 새로운 트리가 되고,
    if (x->parent == tree->NIL) {
        tree->root = y;
    }
    //   b) x가 부모의 왼쪽 자식이었으면 -> y를 부모의 왼쪽 자식으로 설정하고,
    else if (x == x->parent->left) {
        x->parent->left = y;
    }
    //   c) x가 부모의 오른쪽 자식이었으면 -> y를 부모의 오른쪽 자식으로 설정하기
    else {
        x->parent->right = y;
    }
    // 5) y의 왼쪽 자식으로 x를 이동시키고,
    y->left = x;
    //    x의 부모를 y로 설정해주기
    x->parent = y;
    // left rotate 완료 후 트리 모습 출력
    displayTree(tree, "Left Rotate 후 ");
}
```

```
// 4. 회전 함수 : 2) 오른쪽(Right)으로  회전  rightRotate(T,y)
// e.g.       y                          x
//          /  \                       /  \
//         x    r          =>         a    y
//        /  \                            /  \
//       a    b                          b    r
void rightRotate(RedBlackTree *tree, Node *y) {
    // 트리할 노드를 입력받아 이를 기준으로 오른쪽 회전
    // Right Rotate 중이라는 메시지 출력
    char colorRB;
    if (y->color == RED){
        colorRB = 'R';
    } else {
        colorRB = 'B';
    }
    printf("-> 노드 %d(%c)에 대해 Right Rotate 진행합니다.\n", y->key, colorRB);

    // 1) 회전할 노드 y의 왼쪽 자식을 x로 설정
    Node *x = y->left;

    // 2) x의 오른쪽 서브트리(b)를 -> y의 왼쪽 서브트리(b)로 먼저 옮기기
    y->left = x->right;

    // 3) x의 오른쪽 자식이 NIL이 아니면, -> 부모를 y로 설정
    if (x->right != tree->NIL) {
        x->right->parent = y;
    }
    // 4) x의 부모를 y로 설정해주기
    x->parent = y->parent;
    //   a) 만약 y가 트리였으면 -> x를 새로운 트리로 설정하고,
    if (y->parent == tree->NIL) {
        tree->root = x;
    }
    //   b) 만약 y가 부모의 오른쪽 자식이었으면-> x를 부모의 오른쪽 자식으로 설정하고,
    else if (y == y->parent->right) {
        y->parent->right = x;
    }
    //   c) 만약 y가 부모의 왼쪽 자식이었으면 -> x를 부모의 왼쪽 자식으로 설정하기
    else {
        y->parent->left = x;
    }
    // 5) y를 x의 오른쪽 자식으로 이동시키고,
    x->right = y;
    //    y의 부모를 x로 설정해주기
    y->parent = x;

    // right-rotate후 트리 모습 출력하기
    displayTree(tree, "Right Rotate 후 ");
}
```

알고리즘 과제 보고서 8

2022110151 이주연

Left-rotate나 right-rotate 함수 구조상으로 같기 때문에 left rotate기준으로 말씀드리겠습니다. 트리와 노드 x를 입력받아 회전할 노드 x의 오른쪽 자식을 y로 설정해줍니다. 그후 로테이션 전 y의 왼쪽 서브트리(b)를 x의 오른쪽 서브트리(b)로 옮겨줍니다. Y의 왼쪽 자식이 NIL이 아니면 부모를 X로 설정하고, Y의 부모를 X의 부모로 설정한 후, 만약 X가 루트였다면 Y가 새로운 루트가 되도록, X가 부모의 왼쪽 자식이었다면 Y또한 부모의 왼쪽 자식으로, X가 부모의 오른쪽 자식이었다면 Y또한 부모의 오른쪽 자식으로 설정해줍니다. 그후 Y의 왼쪽 자식으로 X를 이동시킨 후 X의 부모를 Y로 설정해주었습니다. LEFT ROTATE가 완료된 후 트리 모습이 출력되도록 설정했습니다.

```
// 5. 레드 블랙 특성을 복구하는 함수 (fix-up)  : 위배된 특성을 -> 다시 복구하는 역할
void rbFixup(RedBlackTree *tree, Node *z) {
// tree : 수정할 레드블랙 트리
// z: 새롭게 삽입된 노드
// y: 삼촌 노드를 가리키는 포인터
    Node *y;
    // rbFixup (특성 복구)를 하는 노드 알려주는 메시지
    char colorRB;
    if (z->color == RED){
        colorRB = 'R';
    } else {
        colorRB = 'B';
    }
    printf("RB 특성 복구(fixup) 시작! - 노드 %d(%c) \n\n", z->key, colorRB);

    // 부모가 red일 때 반복  (4번 특성을 위배했을 때 진행)
    // (삽입된 노드도 red, 노드의 부모도 red로 레드블랙 트리 규칙이 위반되었을 때 처리 과정)
    while (z->parent->color == RED) {
        // 만약 부모(z.p)가 조부모(z.p.p)의 왼쪽(L) 자식일 때
        //              z.p.p (G)
        //              /    \
        //         z.p(여기)  y
        //           / \
        //          ?   ?
        if (z->parent == z->parent->parent->left) {
            y = z->parent->parent->right; // 삼촌 노드를 설정해준다.
            // Case 1 : 삼촌(y)이 Red인 경우
            //               z.p.p (black)       ->          z.p.p(Red)
            //              /     \           p&u:black       /     \
            //           z.p(red) y(RED)    G(p.p):red      z.p(B) y(B)
            //           /                                  /
            //          z                                  z
            if (y->color == RED) {
                char colorRB;
                if (y->color == RED){
                    colorRB = 'R';
                } else {
                    colorRB = 'B';
                }
                printf("-> Case 1: 삼촌 노드 %d(%c)가 RED.\n", y->key, colorRB);
                z->parent->color = BLACK; // 부모를 Black으로 바꾸고
                y->color = BLACK;         // 삼촌도 Black으로 바꾸고
                z->parent->parent->color = RED; // 조부모는 Red로 바꾼다.

                //문구 및 트리 형태 출력
                displayTree(tree, "부모, 삼촌, 조부모 색 변경완료. 조부모가 새로운 z가 됩니다.");
                // 조부모 (Grandp즉, z.p.p) 를 새로운 z로 설정
                z = z->parent->parent;
```

알고리즘 과제 보고서 8

2022110151 이주연

```
} else {
    // 삼촌이 Black인 경우 및
    // Case 2 :  z가 오른쪽(r)자식일 때 ->  (Left rotate) 진행 (해서 case3로 변환)
    //          z.p.p (black)       ->          z.p.p(black)
    //         /    \             왼쪽 회전        /  \
    //     z.p(red) y(BLACK)       (LR)       z.p(red) y(BLACK)
    //          \                                /
    //          z(red) <-이경우                z (왼쪽으로 회전)
    if (z == z->parent->right) {
        printf("-> Case 2 (LR): 노드 %d 는 오른쪽 자식입니다. Case 3로 변환하겠습니다!.\n", z->key);
        z = z->parent;
        leftRotate(tree, z); // 왼쪽 회전 진행
    }
    // Case 3: z가 부모의 왼쪽 자식 (LL Case, 또는 Case 2에서 변환됨)
    //          z.p.p (black)     ->              z.p.p (red)     ->         z.p (black)
    //         /    \           z.p: black          /  \     오른쪽으로 회전 /  \
    //     z.p(red) y(BLACK)     z.p.p: red     z.p(black) y(BLACK)       z(red)  z.p.p(red)
    //      /                  로 색 바꾼 후      /                                  \
    //     z(red) <-이경우                     z(red)                               y (black)
     printf("-> Case 3 : 부모와 조부의 색을 바꾼 후, rightRotate를 진행하겠습니다.\n");
    z->parent->color = BLACK;              // 부모&조부모 색 바꾼 후
    z->parent->parent->color = RED;
    displayTree(tree, "부모와 조부모 색 변경 완료");
    rightRotate(tree, z->parent->parent); // right rotate 진행
}
```

FIXup 레드블랙 특성을 복구하는 함수는 case 1,2,3에 따라 다르게 처리해 특성이 복구되도록 설정했습니다. 먼저 부모가 조부모의 왼쪽 자식인지/ 오른쪽 자식에 따라 구분했고, case1 만약 삼촌이 red인 경우에는 부모, 삼촌, 조부모의 색을 바꾸도록 설정해줬습니다. 만약 삼촌이 black인 경우에는 또 경우를 세분화해 z가 오른쪽 자식일 때를 case2로 표현했고, 왼쪽 자식일 때 case3로 표현했습니다. Case2의 경우 왼쪽 로테이트를 통해 case3로 변환될 수 있게 하였고, case3의 경우 부모와 조부모의 색을 바꾼 후 rightrotate를 진행해 트리가 모든 특성을 유지한채 균형 잡히도록 설정했습니다.