♣

9주차 [과제]

이름 : 유재영

학번 : 20181650

제출일 : 2023.05.08

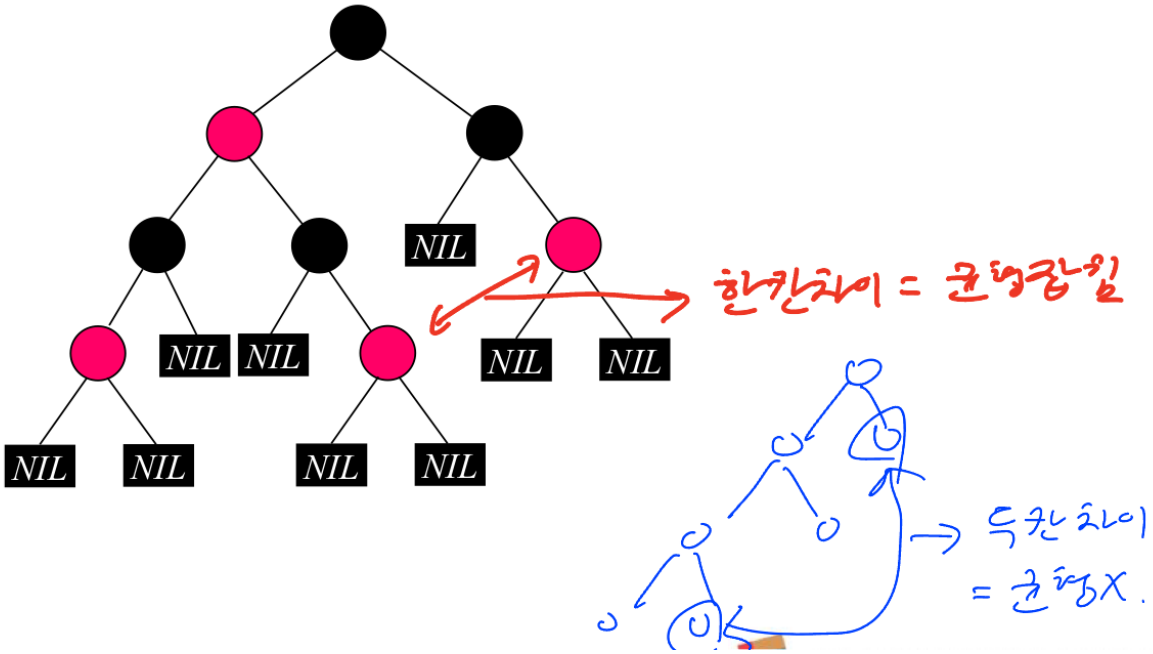
담당교수 : 최해철 교수님

과제. RBT 생성/검색/출력 프로그램 구현 및 분석서

Red Black Tree 분석 진행

RBT는 각 노드에 레드 or 블랙 색을 칠하고, 새로운 노드의 **추가/삭제 시 RBT 트리의 특성을 고려하여 진행**한다.

이를 통해, 균형 잡힌 트리를 만들게 되어 검색-삭제-삽입의 수행 시간을 $logn$ 으로 만든다.



여기서 **"균형이 잡혔다"**는 것은 리프 노드 간의 차이가 **두칸 이상 차이 나지 않는 것**을 말한다.

RBT는 BST의 모든 노드에 블랙 or 레드의 색을 칠하되 다음의 특성을 만족 해야한다.

1. **루트는 블랙이다.**
2. **모든 리프는 블랙이다.**
   → **여기서 리프는 맨 아래 노드가 아닌, NULL 노드를 리프라고 생각한다.**
   → **리프 노드는 NULL이며, NULL은 다 블랙이다.**
3. **노드가 레드이면 그 노드의 자식은 반드시 블랙이다. (R - R 불가능)**

4. **루트 노드에서 임의의 리프 노드에 이르는 경로에서 만나는 블랙 노드의 수는 모두 같다.**

위 특성을 만족시키면서 RBT를 제작하는데 이때 활용되는 방법이 **RBT의 회전**이다.

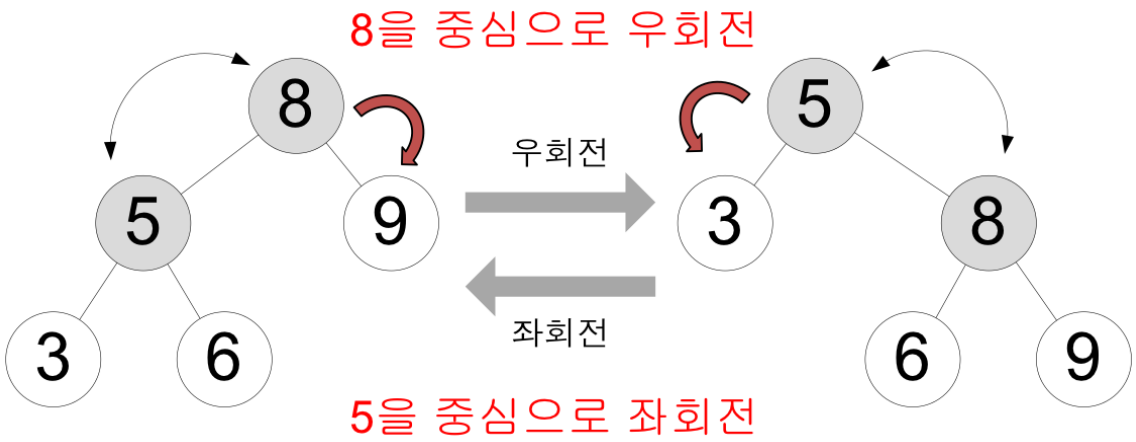
회전(Rotation) → 트리를 변경시키는 방법

- BST의 특성을 유지하면서 변경하는 것

- 부모-자식 노드의 위치를 서로 바꾸는 연산으로 서브트리를 대상으로 한다.

- 우회전(시계 방향) : 왼쪽 자식 노드의 오른쪽 자식 노드를 부모 노드의 왼쪽 자식 노드로 연결한다.

- 좌회전(반시계 방향) : 오른쪽 자식노드의 왼쪽 자식 노드를 부모 노드의 오른쪽 자식 노드로 연결한다.

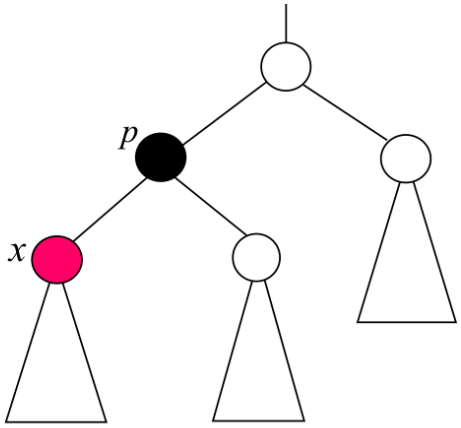


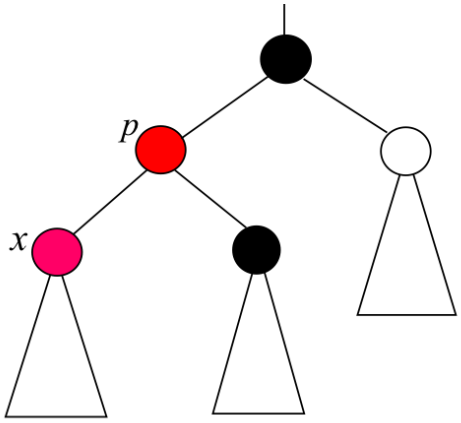
RBT에서의 삽입

RBT에서의 삽입은 BST에서의 삽입과 같다만, 삽입 후 삽입된 노드를 레드로 칠한다. (이 노드를 x라 가정)

x의 부모 노드 p의 색상에 따라 결과가 달라진다.

- p의 색상이 **블랙**이면 아무 문제 없다.
- p의 색상이 **레드**이면 레드 블랙 특성 3번이 깨지게 된다.



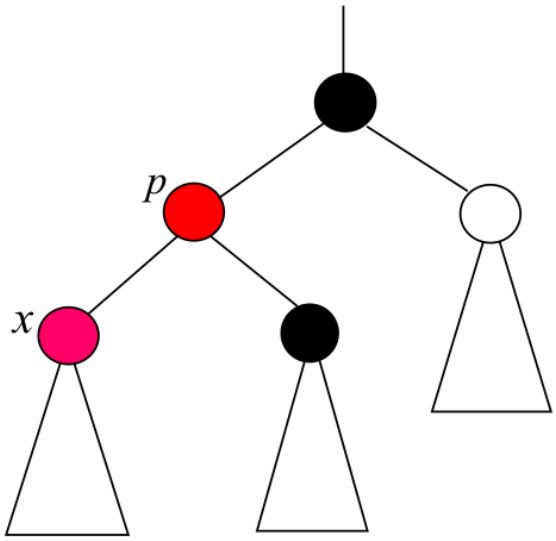


따라서, p의 색상이 **레드**인 경우만 고려하면 된다.

p의 부모 노드를 p^2라 할때, 아래의 과정에서는 p^2의 왼쪽 자식 p 에 대해서만 설명을 진행

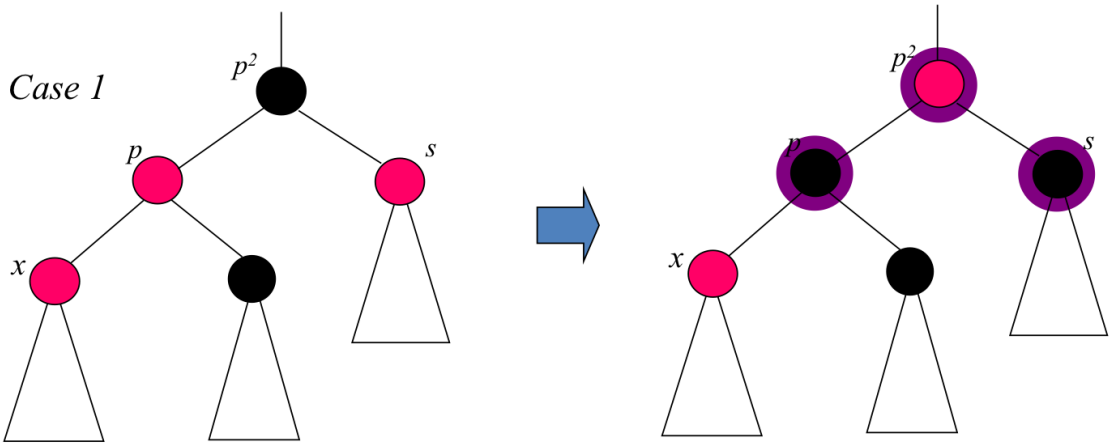
→ 오른쪽 자식인 경우 **완전히 대칭에 해당한다.**

- p^2와 x의 형제 노드는 반드시 블랙이다.
- p의 형제 노드 s의 색상에 따라 두 가지로 나뉘게 됨
  - Case 1: s가 **레드**
  - Case 2: s가 **블랙**
    - Case 2-1: x가 p의 오른쪽 자식
    - Case 2-2: x가 p의 왼쪽 자식



Case 1: s가 레드

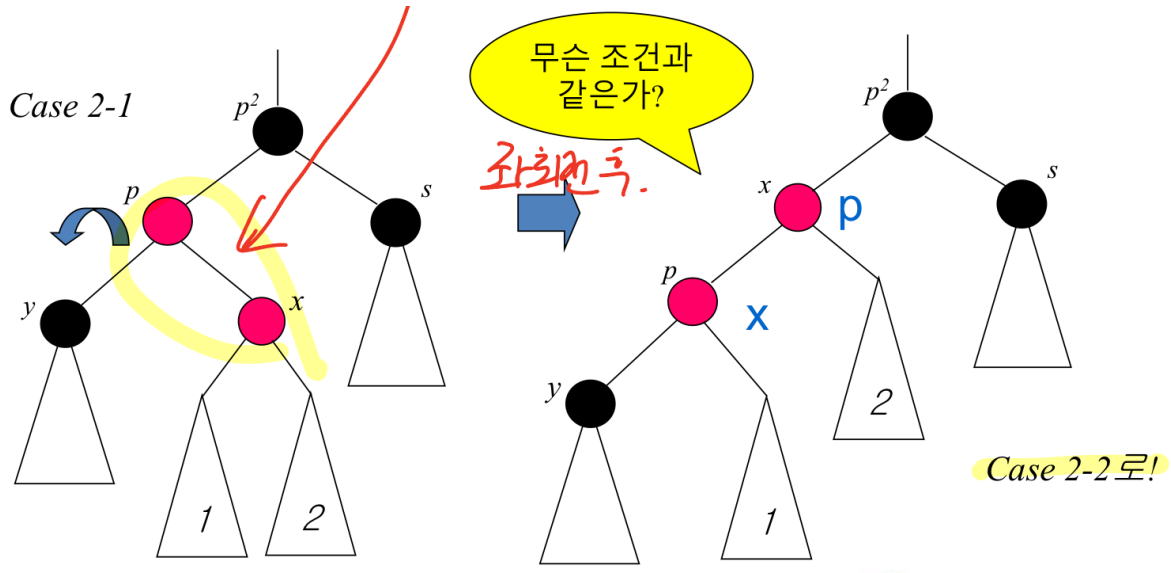
- p와 s의 색상을 **블랙**으로 변경
- p^2의 색상을 **레드**로 변경



- p^2에서 방금과 같은 문제가 발생할 수 있음 (p^2의 부모 노드가 레드인 경우)
- 해당 경우는 **재귀적 접근**으로 해결을 진행한다.

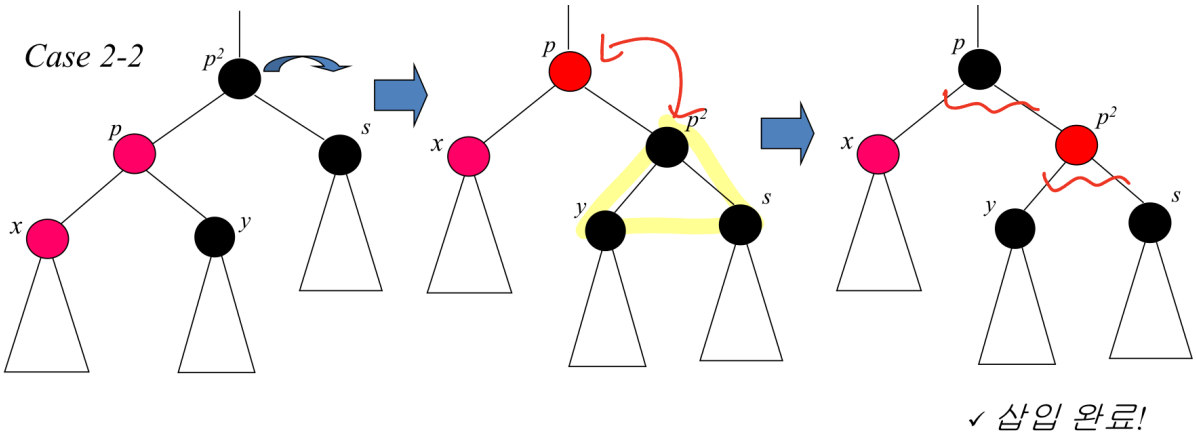
Case 2-1: s가 블랙이고, x가 p의 오른쪽 자식

- p를 좌회전 진행
- 여전히 레드 블랙 특성 3번을 위반하므로, Case 2-2 조건으로 이동



Case 2-2: s가 블랙이고, x가 p의 왼쪽 자식

- p^2를 중심으로 우회전 진행
- p와 p^2의 색상을 맞바꾼다.



RBT에서의 삭제

- 기본적으로 이진검색트리에서의 삭제 방법을 따라 노드 삭제 진행된다.

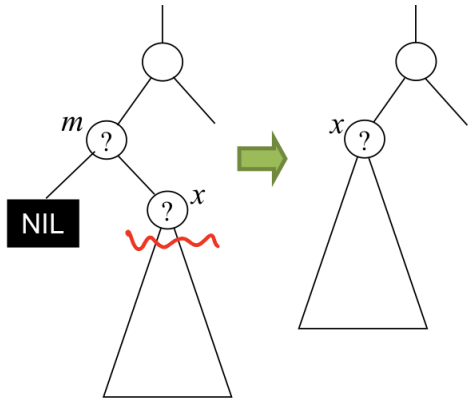
- BST에서의 삭제는 자식이 없는 경우, 자식이 1개인 경우, 자식이 2개인 경우에 따라 다르게 삭제 진행한다.

- 삭제 간에 발생하는 변화는 삭제되는 노드의 자리를 채우기 위해 변화가 생긴다는 것이다.

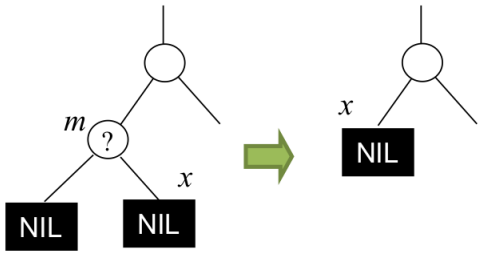
- 이때, RBT **특성 4번의 깨지게 되면 안된다.**

- 정리해보면,

  - 문제를 삭제 노드의 자식이 없거나 1개만을 가진 노드로 제한 (자식이 2개인 경우도 1개의 경우와 동일)
    - **자식이 2개인 경우에도 직후노드에 대해 처리하는 과정으로 접근하게 되어 자식이 1개인 경우로 생각**
  - 삭제하려는 노드 m의 (최대 1개의) 자식을 x로 설정
  - 만약 자식이 없으면 x는 NIL 노드이다. → **자식의 유무를 구분지을 필요가 없음 = 자식이 1개 있다고 가정**
  - m은 자신의 부모 노드의 왼쪽 혹은 오른쪽 자식 → 두 경우는 완전히 대칭이므로 왼쪽인 경우만 고려



자식이 1개인 경우

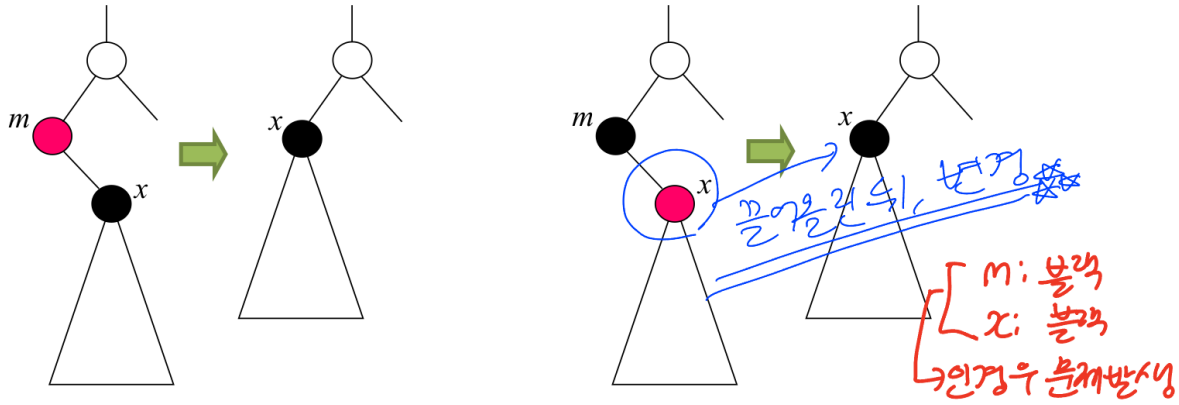


NULL=>c로 생각.

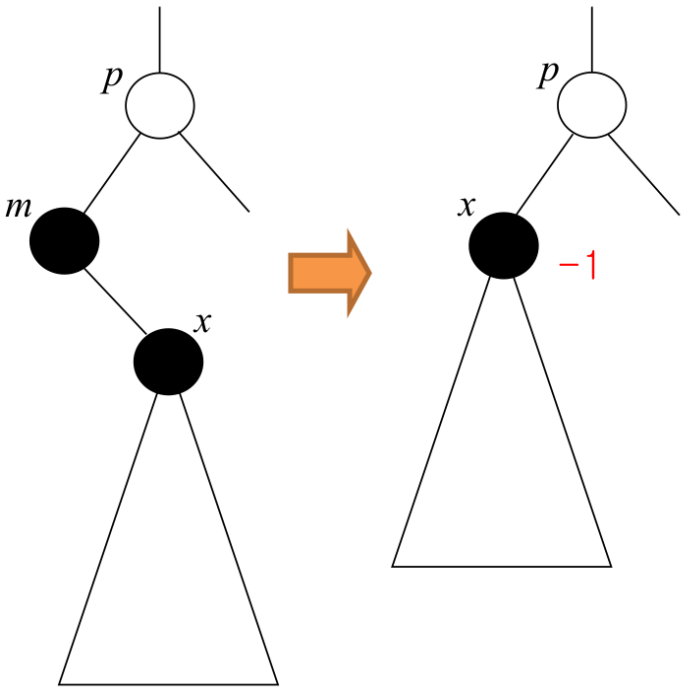
자식이 없는 경우

- 위 내용을 요약해보면

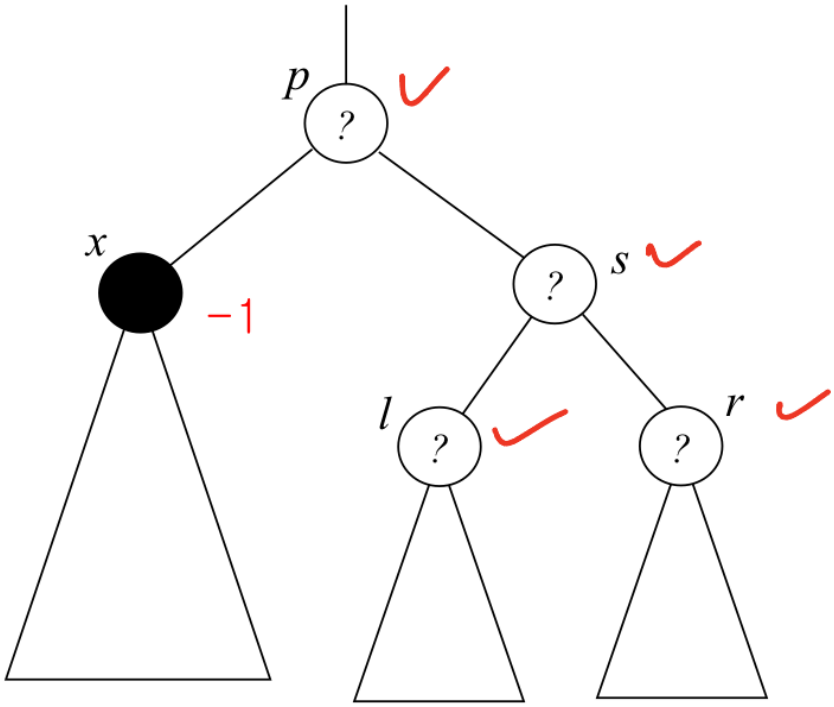
  - 삭제 노드의 자식이 1개인 노드로 제한
  - 삭제 노드가 레드이면 아무 문제 없음
  - 삭제 노드가 블랙이라도 (유일한) 자식이 레드라면 문제 없음
    - 삭제 후 x의 색상을 블랙으로 변경
    - 위 두 경우는 m의 부모가 무엇이든 관계 없음



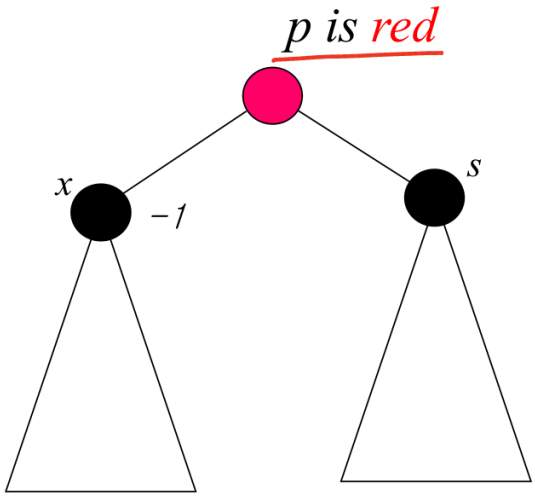
문제 발생 : 삭제 노드와 자식 노드가 모두 블랙인 경우, RBT 특성 4번 위반



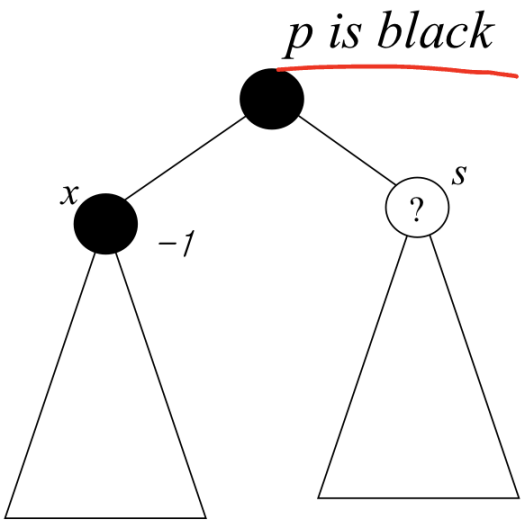
- x옆의 -1 은 루트에서 x를 통해 리프에 이르는 경로에서 블랙 노드의 수가 하나 모자람을 의미



- x의 주변 상황에 따라 처리 방법이 달라짐
- p의 색상에 따라 1차적으로 구분

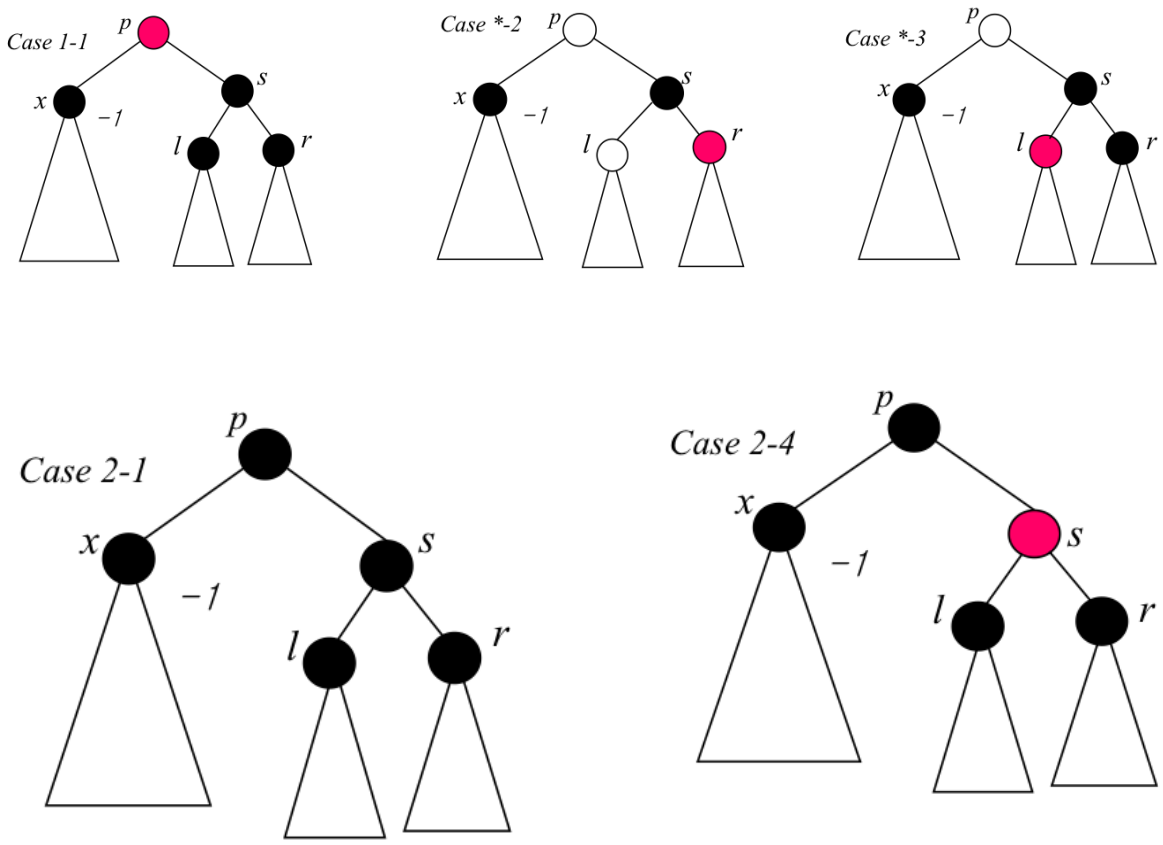


p가 레드이면 s는 반드시 블랙



p가 블랙이면 s는 블랙 or 레드

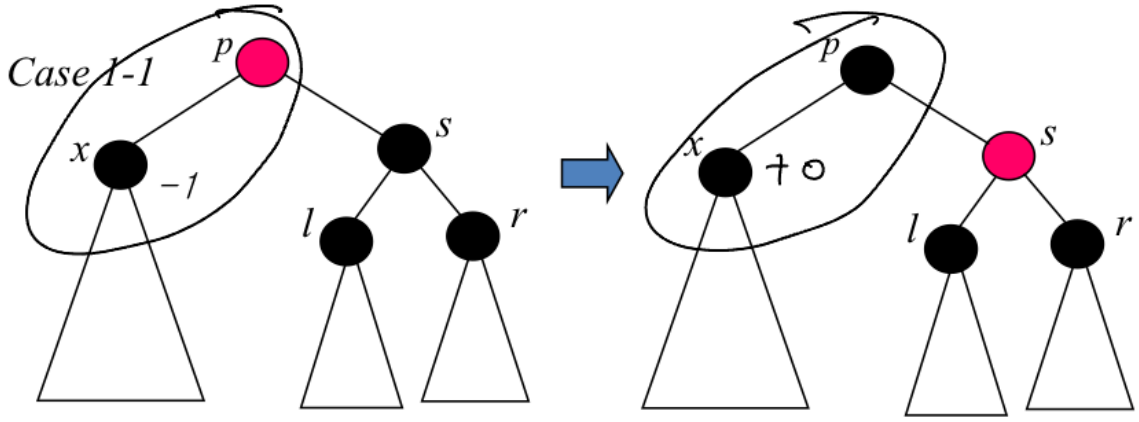
총 5가지 경우로 정리 가능



빈 곳의 색상은 레드 or 블랙 둘다 가능

Case 1-1: x는 블랙, p가 레드(s는 반드시 블랙), l과 r이 모두 블랙

→ p와 s의 색상을 맞바꿈. 즉, p를 블랙으로, s를 레드로 변경



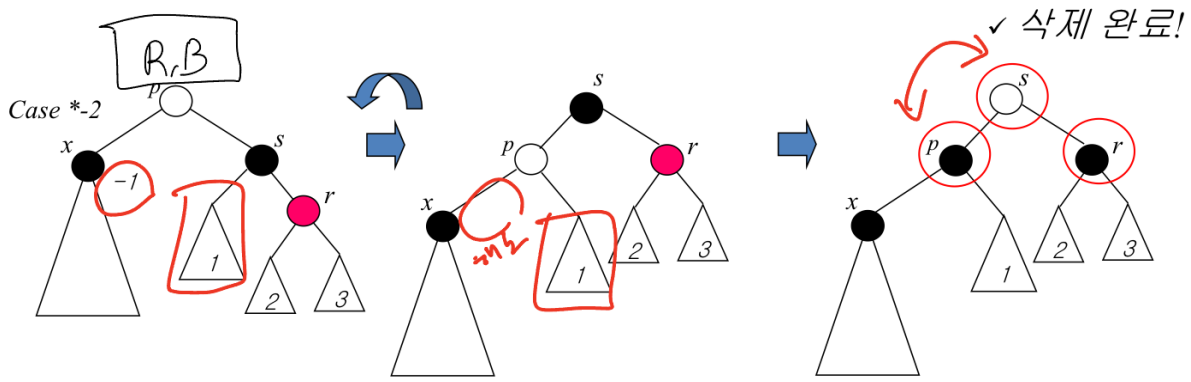
- x에 이르는 경로 상에 블랙이 하나 추가되었으므로 x에 이르는 경로에서 블랙 노드가 하나 모자란 것이 해소

- 루트로부터 s를 지나는 경로상의 블랙 노드의 수에는 변화 없음

Case *-2: x이 블랙, p가 레드 or 블랙, s가 블랙, l이 레드 or 블랙, r이 레드

→ p를 중심으로 좌회전 (시계 반대 방향)

→ p와 s의 색상 바꾸기. r의 색상을 블랙으로 변경



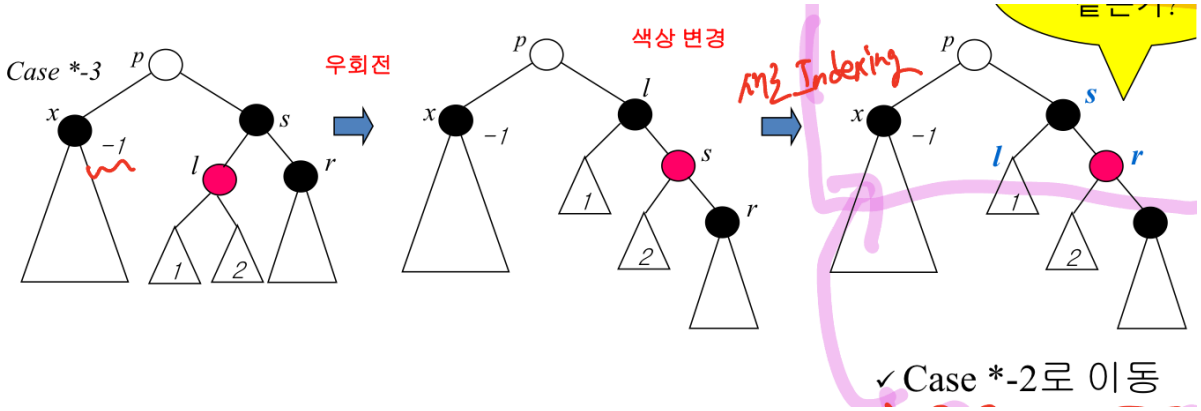
- x에 이르는 경로 상에 블랙이 하나 추가되었으므로, x에 이르는 경로에서 블랙 노드가 하나 모자라던 것이 해소

- 1, 2, 3으로 표시된 서브트리들은 루트로부터 지나가는 경로상에 있는 블랙 노드의 수에 변화가 없음

Case *-3: x이 블랙, p가 레드 or 블랙, s가 블랙, l이 레드, r이 블랙

→ s를 중심으로 우회전(시계 방향), l과 s의 색상을 바꿈

→ Case *-2로 이동



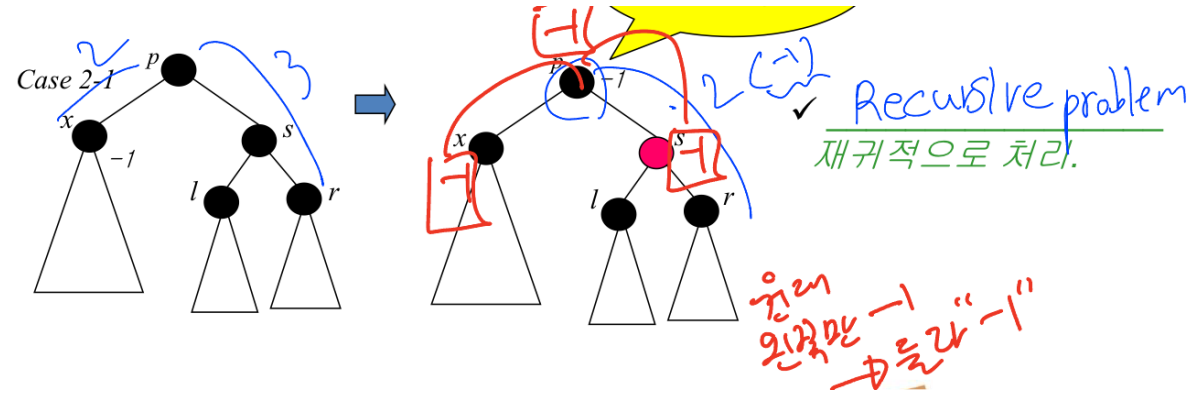
- Case *-2: x이 블랙, p가 레드 or 블랙, s가 블랙, l이 레드 or 블랙, r이 레드

Case 2-1: x이 블랙, p가 블랙, s가 블랙, l이 블랙, r이 블랙

→ s를 블랙에서 레드로 변경

→ s를 지나는 경로에서 블랙 노드의 개수가 하나 모자라게 되어, p를 지나는 경로에서 블랙 노드가 하나 모자람

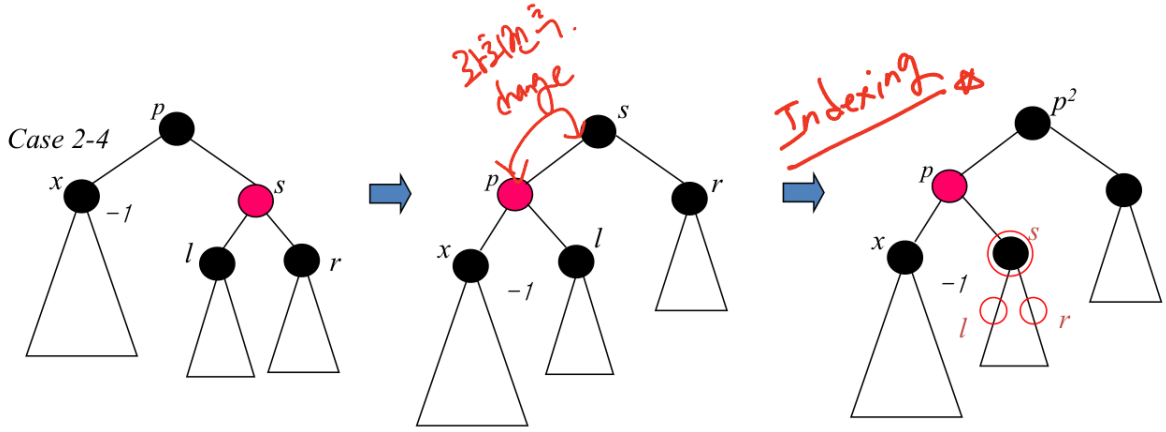
→ p를 **문제 발생 노드로 하여**(x로 생각하고) 재귀적으로 다시 시작



Case 2-4: x이 블랙, p가 블랙, s가 레드, l이 블랙, r이 블랙

→ p를 중심으로 좌회전(시계 반대 방향), p와 s의 색상을 바꿈

→ x의 부모가 레드가 되었으므로, 색상의 조합을 따져서 Case 1-1, 1-2, 1-3으로 이동



- l과 r을 경유하는 경로와 관련해서는 문제가 없음

→ 위 내용들을 바탕으로 프로그램 구현 진행

RBT 프로그램 구현

`RedBlackTree.h`

RedBlackTree.h

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   typedef int ElementType;
5
6   typedef struct tagRBTNode {
7       struct tagRBTNode *Parent;
8       struct tagRBTNode *Left;
9       struct tagRBTNode *Right;
10
11      enum {
12          RED, BLACK
13      } Color;
14
15      ElementType Data;
16  } RBTNode;
```

- 헤더파일에는 구조체 정의를 진행한다.

- 4행에서는 int형 변수를 ElementType으로 `typedef`를 진행한다.

- 6행부터는 구조체를 정의하며, 각각 부모노드, 왼쪽 자식노드, 오른쪽 자식노드의 주소값을 가진다.

- 11행에서는 내부 노드 색상을 열거형인 `enum 타입`을 사용해 구현하며, **RED**, **BLACK**으로 정의한다.

- 15행은 노드에 적히게되는 Data가 들어가는 부분을 선언해준다.

`RedBlackTree.c`

RedBlackTree.c

```c
#include "RedBlackTree.h"

extern RBTNode *Nil; // 선언 되어있는 외부 변수를 사용하겠다는 의미

RBTNode *RBT_CreateNode(ElementType NewData) {
    RBTNode *NewNode = (RBTNode *) malloc( size: sizeof(RBTNode));
    NewNode->Parent = NULL;
    NewNode->Left = NULL;
    NewNode->Right = NULL;
    NewNode->Data = NewData;
    NewNode->Color = BLACK;

    return NewNode;
}

void RBT_DestroyNode(RBTNode *Node) {
    free(Node);
}
```

- 3행에서는 선언되어 있는 외부 변수 Nil을 사용하겠다는 의미이다. (`Test_RedBlackTree.c`에 정의됨)

- 5행은 노드를 생성하는 부분으로 6행에서 동적 할당을 진행한다.

- 7행에서 10행은 노드 내부에 구현되어 있는 내용들을 초기화 한다.

- 11행은 노드의 색상은 RBT의 특성에 따라, 리프노드(Nil)에 맞는 블랙으로 색상을 설정한다.

- 13행에서는 생성된 노드의 주소값을 반환한다.

- 16행은 노드를 삭제하는 과정으로, 동적할당된 노드를 해제한다.

RedBlackTree.c

```
20  void RBT_DestroyTree(RBTNode *Tree) {
21      if (Tree->Right != Nil)
22          RBT_DestroyTree( Tree: Tree->Right);
23
24      if (Tree->Left != Nil)
25          RBT_DestroyTree( Tree: Tree->Left);
26
27      Tree->Left = Nil;
28      Tree->Right = Nil;
29
30      RBT_DestroyNode( Node: Tree);
31  }
32
33  RBTNode *RBT_SearchNode(RBTNode *Tree, ElementType Target) {
34      if (Tree == Nil)
35          return NULL;
36
37      if (Tree->Data > Target)
38          return RBT_SearchNode( Tree: Tree->Left, Target);
39      else if (Tree->Data < Target)
40          return RBT_SearchNode( Tree: Tree->Right, Target);
41      else
42          return Tree;
43  }
```

- 20행은 트리를 제거하는 과정으로, 21행과 24행의 if문은 리프노드(Nil)가 아닌 경우 해당 노드의 서브트리가 존재할 수 있으므로, 먼저 서브트리를 해제 진행하는 과정이다.

- 27,28행에서는 제거된 루트 노드의 자식 노드들을 리프노드로 변경하는 과정이다.

- 이후 30행에서 트리의 동적할당 해제를 진행한다.

- 33행에서는 노드를 검색하는 메서드이다. 만약 트리의 값이 Nil이면 찾으려는 값이 없으므로 NULL을 반환한다.

- 37행은 찾으려는 Target의 값이 트리의 값보다 작은 경우, 트리의 왼쪽 노드를 대상으로 다시 검색을 진행한다.

- 39행은 찾으려는 Target의 값이 트리의 값보다 큰 경우, 트리의 오른쪽 노드를 대상으로 다시 검색을 진행한다.

- 만약 둘다 해당되지 않으면 찾으려는 Target값을 찾은것이므로, 트리의 주소를 리턴한다.

RedBlackTree.c

```
45  RBTNode *RBT_SearchMinNode(RBTNode *Tree) {
46      if (Tree == Nil)
47          return Nil;
48
49      if (Tree->Left == Nil)
50          return Tree;
51      else
52          return RBT_SearchMinNode( Tree: Tree->Left);
53  }
54
55  void RBT_InsertNode(RBTNode **Tree, RBTNode *NewNode) {
56      RBT_InsertNodeHelper(Tree, NewNode);
57
58      NewNode->Color = RED;
59      NewNode->Left = Nil;
60      NewNode->Right = Nil;
61
62      RBT_RebuildAfterInsert(Tree, NewNode);
63  }
```

- 45행은 가장 작은 값을 가지는 MinNode를 찾는 과정이다.

- 49행에서 만약 Tree의 왼쪽 노드가 Nil이면, 해당 트리가 가장 작은 값을 가지는 노드이므로 트리의 값을 반환.

- 아닌 경우 52행을 통해 재귀호출을 통해 계속 왼쪽 밑으로 내려간다.

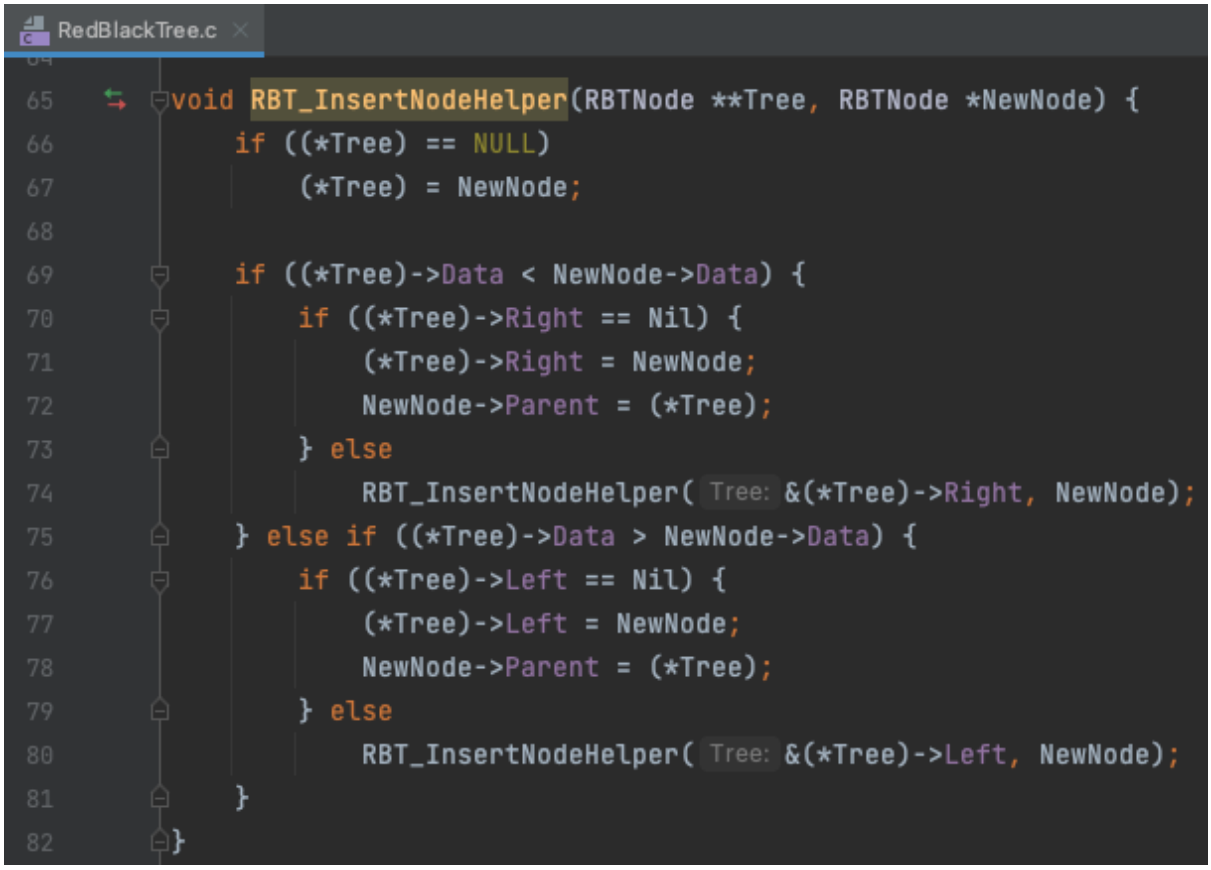
- **위 메서드의 경우 삭제되는 노드를 대체할 대상을 찾는 삭제과정에서 활용되는 메서드이다.**

- 55행은 노드를 삽입하는 메서드로, **트리의 주소를 이중포인터로 받는다.**

- 56행의 `RBT_InsertNodeHelper()`는 노드를 삽입하는 과정에서 이중포인터로 받은 트리의 주소에 직접 접근할때 활용되는 메소드로 아래에서 자세하게 설명을 진행한다.

- 58~60행은 삽입된 노드는 Color을 **RED**로 (삽입 조건), 노드의 양쪽 자식을 Nil로 초기화를 진행

- 62행은 삽입된 노드를 RBT 특성에 맞추기 위해 전체적으로 리빌딩을 진행하는 것이며, 해당 메서드의 상세한 설명은 아래에서 진행한다.



- 65행 `InsertNodeHelper()`는 노드를 추가하는 과정에서 활용되는 메소드이다.

- 66행은 만약 트리의 주소값이 NULL인 경우, 처음 노드가 추가되는 과정 or 재귀 호출을 통해 해당 자리에 노드를 추가하는 과정으로 진행되며, `(*Tree)`의 노드로 설정한다. → `(*Tree) = NewNode;`

- 69행과 75행은 Data 값을 보고 NewNode→Data와 판단한 뒤, 추가하는 과정이다.

- 69행은 NewNode→Data가, (*Tree)→Data보다 큰 경우로 이때, 트리의 오른쪽이 Nil인 경우 직접 연결한다.

- 만약 오른쪽이 Nil이 아닌 경우, `&(*Tree)→Right`로 `RBT_InsertNodeHelper()`재귀 호출을 진행해 처리

- 75행은 NewNode→Data가, (*Tree)→Data보다 작은 경우로 이때, 트리의 왼쪽이 Nil인 경우 직접 연결한다.

- 만약 왼쪽이 Nil이 아닌 경우, `&(*Tree)→Left`로 `RBT_InsertNodeHelper()`재귀 호출을 진행해 처리

RedBlackTree.c

```
84  void RBT_RotateRight(RBTNode **Root, RBTNode *Parent) {
85      RBTNode *LeftChild = Parent->Left;
86
87      Parent->Left = LeftChild->Right;
88
89      if (LeftChild->Right != Nil)
90          LeftChild->Right->Parent = Parent;
91
92      LeftChild->Parent = Parent->Parent;
93
94      if (Parent->Parent == NULL)
95          (*Root) = LeftChild;
96      else {
97          if (Parent == Parent->Parent->Left)
98              Parent->Parent->Left = LeftChild;
99          else
100             Parent->Parent->Right = LeftChild;
101     }
102
103     LeftChild->Right = Parent;
104     Parent->Parent = LeftChild;
105 }
```

- 84행은 우회전을 구현하는 과정이다.

- 85행에서는 우회전을 위해 `Parent→Left`을 LeftChild에 저장한다.

- 87행에서는 `Parent→Left`에 `Parent→Left→Right` 값 즉, `LeftChild→Right`를 연결한다.

- 89행에서는 LeftChild→Right가 Nil 값이 아니면, LeftChild의 오른쪽 자식의 부모 주소값을 연결되는 부모의 값으로 변경하는 과정이다. 즉, 87행에서 한 작업을 이어서 옮겨가는 LeftChild→Right 자식 입장에서도 진행하는 것이다.

- 92행의 과정은, 우회전을 진행하면 **기존 부모의 왼쪽 자식이 부모의 자리를 차지하는 것**이므로, LeftChild의 Parent 자리를 Parent→Parent로 대체(연결)한다.

- 94행에서는 만약 Parent→Parent의 값이 NULL인 경우, Parent가 루트노드임을 의미하므로, `*Root`의 주소 값을 LeftChild 값으로 변경한다.

- 아닌 경우, 96행부터는 부모가 부모의 부모(조상)의 왼쪽 자식이였는지, 오른쪽 자식이였는지를 판단하여 연결 해주는 과정이다. **위 과정을 거쳐야만, 우회전 이후 정상적으로 모든 노드가 연결되어 있다.**

- 103행에서는 우회전을 진행하였으므로, LeftChild→Right에 Parent 값을 연결하며

- 104행에서는 최종적으로 부모의 부모를 LeftChild 값으로 변경한다.

> ❗ 104행이 맨 마지막으로 진행되는 이유는, 96~101행에서 우선적으로 부모의 부모에 대해 작업을 진행해야하므로, 해당 정보가 덮어씌워지면 안된다.

RedBlackTree.c

```
107  void RBT_RotateLeft(RBTNode **Root, RBTNode *Parent) {
108      RBTNode *RightChild = Parent->Right;
109
110      Parent->Right = RightChild->Left;
111
112      if (RightChild->Left != Nil)
113          RightChild->Left->Parent = Parent;
114
115      RightChild->Parent = Parent->Parent;
116
117      if (Parent->Parent == NULL)
118          (*Root) = RightChild;
119      else {
120          if (Parent == Parent->Parent->Left)
121              Parent->Parent->Left = RightChild;
122          else
123              Parent->Parent->Right = RightChild;
124      }
125
126      RightChild->Left = Parent;
127      Parent->Parent = RightChild;
128  }
```

- 107행은 좌회전을 구현하는 과정이다.

- 108행에서는 좌회전을 위해 `Parent→Left`를 RightChild에 저장한다.

- 110행에서는 부모의 오른쪽에 부모의 오른쪽 자식의 왼쪽 값(`RightChild→Left`)을 저장한다.

- 112행에서는 부모의 오른쪽 자식의 왼쪽 값이 Nil이 아닌 경우, 그 부모의 값을 옮겨간 부모의 값으로 변경하는 과정이다. 이로써 변경된 노드의 부모 자식 관계가 연결되는 것이다. (위 89행과 같은 과정)

- 115행에서는 좌회전 진행간, RightChild가 부모의 자리를 차지하는 것이므로, RightChild 부모의 주소 값을 기존 부모의 부모 값으로 변경하는 과정이다.

- 117행은 만약, 기존 부모의 부모의 값이 NULL인 경우, Parent가 루트노드였음을 의미하므로 (*Root)의 값을 RightChild 값으로 변경한다.

- 120행부터는 부모의 부모(조상)의 입장에서 자식의 값이 변경된 것이므로, 알맞은 자리의 값을 변경하기위해 어떠한 자리(왼쪽 자식인지 오른쪽 자식인지)를 찾기위해 진행하는 과정이다.

- `if(Parent == Parent→Parent→Left)`를 통해서 비교를 진행하며, 올바른 자리의 주소값에 RightChild를 넣는다.

- 126행에서는 좌회전을 진행했으므로 최종적으로 RightChild→Left 값을 Parent로 변경하는 과정이다.

- 127행은 최종적으로 Parent의 Parent 값을 RightChild로 변경한다.

RedBlackTree.c

```c
void RBT_RebuildAfterInsert(RBTNode **Root, RBTNode *X) {
    while (X != (*Root) && X->Parent->Color == RED) {
        if (X->Parent == X->Parent->Parent->Left) {
            RBTNode *Uncle = X->Parent->Parent->Right;
            if (Uncle->Color == RED) {
                X->Parent->Color = BLACK;
                Uncle->Color = BLACK;
                X->Parent->Parent->Color = RED;

                X = X->Parent->Parent;
            } else {
                if (X == X->Parent->Right) {
                    X = X->Parent;
                    RBT_RotateLeft(Root, Parent: X);
                }

                X->Parent->Color = BLACK;
                X->Parent->Parent->Color = RED;

                RBT_RotateRight(Root, X->Parent->Parent);
            }
        } else {
            RBTNode *Uncle = X->Parent->Parent->Left;
            if (Uncle->Color == RED) {
                X->Parent->Color = BLACK;
                Uncle->Color = BLACK;
                X->Parent->Parent->Color = RED;
                X = X->Parent->Parent;
            } else {
                if (X == X->Parent->Left) {
                    X = X->Parent;
                    RBT_RotateRight(Root, Parent: X);
                }

                X->Parent->Color = BLACK;
                X->Parent->Parent->Color = RED;
                RBT_RotateLeft(Root, X->Parent->Parent);
            }
        }
    }

    (*Root)->Color = BLACK;
}
```

- 130행은 노드를 삽입한 뒤, RBT 특성에 맞는지 체크하며, 바르지 않은 경우 처리를 하는 메소드이다.

- 먼저 while문을 통해 특성에 맞을때 까지, 계속 진행을 하게되며 그 조건은

1. X의 주소값이 루트랑 같지 않으며,
2. X의 부모의 색상이 RED인 동안 진행한다.

→ 부모 노드가 블랙, 최종적으로 루트가 블랙이면 문제 없음

→ **루트 노드이지만, 루트 노드의 색상이 레드인 경우는 RBT의 특성을 만족하지 못한 것이므로 while문이 계속 진행된다.**

- 초반에 설명하였듯이 삽입 한 대상의 부모노드가 블랙이면 아무 문제가 없지만, 레드이면 특성 3번. 노드가 레드이면 그 노드의 자식은 반드시 블랙이다 깨지게된다. 따라서 해당 부분을 만족할 때 까지 진행한다.

- 130행에서 파라미터로 받는게 되는 값은 `RBT_InsertNode()`를 참고하면 Tree의 값과 NewNode을 파라미터로 받는다.

- 먼저 132행에 의해, 만약 X의 부모의 주소값이 X→부모의 부모(조상)의 왼쪽 자식 값과 같으면 조상의 오른쪽 값을 삼촌으로 취급하며(132행), 반대의 경우 조상의 왼쪽 값을 삼촌으로 취급한다(152행).

- 134행에서 삼촌의 색상이 **RED**인 경우에 대해 진행한다. 이때, X의 부모의 색상과, 삼촌의 색상을 블랙(135,136행)으로 바꾸며, X의 부모의 부모, 즉 조상의 색상을 **RED**(137행)로 변경한다.

- 이때, 발생하는 문제점은 단 하나로, 바로 조상님의 부모의 색상이 RED인 경우 **레드-레드의 형태**를 가지는 경우다. 따라서, X의 값을 조상의 값으로 변경한 다음 재귀호출(while문 반복)을 진행한다. (139행)

- 만약 삼촌의 색상이 **블랙**인 경우, 먼저 X의 값이 X의 부모의 오른쪽 값에 해당하면(141행), 부모를 기준으로 좌회전을 진행하기 위해, X의 값을 부모의 값으로 변경한 뒤(142행), 좌회전을 진행한다(143행)

- 위 과정은 Case 2-1에 해당한다. / Case 2-1: 삼촌이 블랙이고, X가 부모의 오른쪽 자식

- 이때, X의 부모 노드와, X의 조상의 색상을 서로 바꿔주는데, **각 노드의 색상은 RBT 특성에 의해 레드&블랙으로 고정되어 있으므로**, X의 부모 노드 색상을 블랙으로(146행), X의 조상 노드의 색상을 **레드**로 변경(147행)한다.

- 이후 우회전(149행)을 진행하며 위 과정은 Case 2-2에 해당한다. / Case 2-2: 삼촌이 블랙이고, X가 부모의 왼쪽 자식

- 152행은 왼쪽 값을 삼촌으로 취급하는 과정에 이어지는 것으로, 만약 삼촌의 색상이 **RED**인 경우 위에서 진행한 방법과 같이, X의 부모노드의 색상을 블랙으로, 삼촌의 색상을 블랙

으로, 조상의 색상을 **RED**로 변경한다. 마찬가지로, **레드-레드 형태**를 가지는 경우에 대해 처리를 위해 X의 값을 조상으로 변경을 진행한다.

- 아래의 내용은 위와 마찬가지로, 삼촌이 색상이 블랙인 경우에, 색상을 변경하는 과정으로 먼저 Case 2-1로 먼저 처리를 진행하고 Case 2-2로 이어진다.

- 이후 166행에서 좌회전을 진행하여 마무리 한다.

- 171행에서는 while문을 탈출한 뒤, 마무리 작업으로 Root의 색상을 블랙으로 바꿔주어 마무리한다.

- 171행 과정의 이유는 **RBT의 루트 노드의 색상은 블랙이어야 한다는 특성을 만족하기 위한 작업**이다.

RedBlackTree.c

```
175 RBTNode *RBT_RemoveNode(RBTNode **Root, ElementType Data) {
176     RBTNode *Removed = NULL;
177     RBTNode *Successor = NULL;
178     RBTNode *Target = RBT_SearchNode( Tree: (*Root),  Target: Data);
179
180     if (Target == NULL)
181         return NULL;
182
183     if (Target->Left == Nil || Target->Right == Nil) {
184         Removed = Target;
185     } else {
186         Removed = RBT_SearchMinNode( Tree: Target->Right);
187         Target->Data = Removed->Data;
188     }
189
190     if (Removed->Left != Nil)
191         Successor = Removed->Left;
192     else
193         Successor = Removed->Right;
194
195     Successor->Parent = Removed->Parent;
196
197     if (Removed->Parent == NULL)
198         (*Root) = Successor;
199     else {
200         if (Removed == Removed->Parent->Left)
201             Removed->Parent->Left = Successor;
202         else
203             Removed->Parent->Right = Successor;
204     }
205
206     if (Removed->Color == BLACK)
207         RBT_RebuildAfterRemove(Root,  X: Successor);
208
209     return Removed;
210 }
```

- 175행은 노드를 삭제하는 과정이다.
- 176행은 삭제할 노드를 담는 공간이며, 178행은 후임노드를 담는 공간이다.

- 178행은 삭제할 노드를 `RBT_SearchNode()`를 통해서 찾은 후 Target에 담는다.

- 180행에서는 삭제하려는 노드가 없는 경우에 해당하므로 NULL을 리턴한다.

- 183행에서 고려하는 케이스는 위에서 정리한 것과 같이, 한쪽 자식이라도 Nil 값인 경우와 둘다 Nil 값이 아닌 경우의 두 케이스, 즉 자식이 하나 혹은 없는 인 경우와 자식이 둘다 있는 경우 두 가지로 나뉘어 지는 것이다.

- 그에 따라서, 186행은 삭제되는 노드의 직후 노드를 `RBT_SearchMinNode()`로 찾는 과정으로 찾은 직후노드의 값, 즉 Removed→Data의 값을 Target→Data의 값 대신 대체한다(187행).

- 187행의 작업 이후 삭제되는 대상노드 Target은 186행에서 설정한, 직후 노드가 된다.

- **직후노드의 자식은 0개 혹은 오른쪽 자식 1개를 가질 수 밖에 없으므로, 아래 삭제 과정에서는 자식이 두개인 경우에 대해서 배제하고 진행하게 되며, 자식이 0개 혹은 1개 인 경우에 대해 고려하여 삭제를 진행하게 된다.**

- 190~193행은 후임 노드 Successor 공간에 삭제되는 노드의 후임자를 담는 과정이다. Removed→Left가 Nil이 아니면, Nil을 담고, 반대의 경우는 Removed→Right을 담는다.

- 195행에서는 후임노드의 부모 주소를 삭제되는 노드의 부모 주소로 바꿔준다.

- 197행에서 만약, 삭제되는 노드의 부모노드가 NULL인 경우, 삭제되는 노드가 루트노드에 해당하므로 (*Root)의 값을 Successor로 직접 변경해준다.(198행)

- 200행은 삭제노드 Removed의 값이 Removed의 부모의 왼쪽에 달려있었던 경우에 해당하면, 201행에 의해 해당 자리에 Successor, 후임자를 대체한다.

- 203행은 반대의 경우, 부모의 오른쪽에 달려있었던 경우에 해당하므로, 해당자리에 후임자로 대체한다.

- 206행에서는 만약 삭제되는 노드의 색상이 블랙인 경우, RBT 특성 4번을 깨트리게 되므로, `RBT_RebuildAfterRemove()`를 진행한다. 상세한 내용은 아래에서 설명한다.

- 이후 동적 할당을 해제하기 위해 제거되는 노드의 주소값을 반환한다(209행). 해당 처리 과정은 main문에서 진행된다.

RedBlackTree.c

```
212  void RBT_RebuildAfterRemove(RBTNode **Root, RBTNode *Successor) {
213      RBTNode *Sibling = NULL;
214
215      while (Successor->Parent != NULL && Successor->Color == BLACK) {
216          if (Successor == Successor->Parent->Left) {
217              Sibling = Successor->Parent->Right;
218
219              if (Sibling->Color == RED) {
220                  Sibling->Color = BLACK;
221                  Successor->Parent->Color = RED;
222                  RBT_RotateLeft(Root, Successor->Parent);
223                  Sibling = Successor->Parent->Right;
224              } else {
225                  if (Sibling->Left->Color == BLACK && Sibling->Right->Color == BLACK) {
226                      Sibling->Color = RED;
227                      Successor = Successor->Parent;
228                  } else {
229                      if (Sibling->Left->Color == RED) {
230                          Sibling->Left->Color = BLACK;
231                          Sibling->Color = RED;
232
233                          RBT_RotateRight(Root,  Parent: Sibling);
234                          Sibling = Successor->Parent->Right;
235                      }
236                      Sibling->Color = Successor->Parent->Color;
237                      Successor->Parent->Color = BLACK;
238                      Sibling->Right->Color = BLACK;;
239                      RBT_RotateLeft(Root, Successor->Parent);
240                      Successor = (*Root);
241                  }
242              }
```

- 212행은 삭제된 후 삭제된 노드의 색상이 블랙인 경우에 진행하게되는 메소드이다. 내용이 길어 파트를 두개로 나누어 설명을 진행한다.

- 215행 while문을 통해 반복 진행하며, 그 기간은 후임자(Successor)의 부모가 NULL이 아니고, 후임자의 색상이 블랙일 때까지 진행한다.

- 먼저 216행을 통해 두가지의 경우로 분리한다. 후임자가 부모의 왼쪽 자식인 경우(216행) or 오른쪽 자식인 경우(243행)이다. 따라서 형제를 의미하는 Sibling에 후임자가 왼쪽 자식이면 오른쪽 주소를, 후임자가 오른쪽 자식이면 왼쪽 주소를 넣어준다.

- 먼저 가장 간단한 경우인, 형제가 붉은 색인경우에 대해서 처리한다.(219행)

- 이때는, 부모를 기준으로 좌회전을 진행하며 형제의 색상을 블랙으로(220행), 부모의 색상을 레드로(221행) 처리한다. 해당 과정은 **삽입 후 처리 과정에서와 마찬가지로, 색상은 고정되어있으므로 Left전에 처리해도 무방하다.**

- 제시된 코드에서는 223행과, 249행에 Sibling의 값을 초기화 하는 과정이 있는데, 해당 부분은 삭제해도 무방한것 같다. 이유는 해당 과정에서 초기화가 일어나더라도, **217행에서 초기화를 다시 진행하기 때문에** 삭제해도 무방한듯 하다. 해당 부분은 교수님께 질문드릴 예정이다.

- 이후 먼저, 225행에서는 형제노드의 양쪽 자식 노드가 블랙인 경우의 Case 2-1에 대해 처리를 진행한다.

- 해당 부분은 형제의 색상을 레드로 변경한 뒤(226행), 부모노드 기준으로 다시 재귀 호출을 진행하기 위해 형제 노드의 값을 부모의 값으로 변경하여, while문이 다시 실행되도록 한다.(227행).

- 위 케이스는 Case2-1에서 얘기하고 있는 양쪽에 리프노드로 가는 동안 블랙의 숫자가 1개 부족한 현상에 대해서 **부모 노드의 입장으로써** 형제의 색상을 블랙으로 바꾸면서, 부모노드 입장에서 공통적으로 리프노드로 가는 동안 블랙 노드의 숫자 1개가 부족한 현상을 만든 뒤 위 문제 사항(블랙 노드 숫자 부족)을 처리하게 진행하는 과정인 것이다.

- 228행, 225행의 반례에 대해 처리를 진행하며,

- 229행은 형제의 왼쪽 자식이 RED인 경우에 대해 먼저 처리를 진행한다. 위 과정은 Case *-3에 해당한다.

- 형제의 왼쪽 자식의 색상을 블랙으로 바꾸고, 형제의 색상을 RED로 변경한다.

- 이후, 우회전을 진행한 뒤, Case *-2를 진행하기위해 Sibling의 값을 Successor→Parent→Right의 값으로 변경해준다.

- 이어서 236행은 Case *-2에 해당하는 과정을 진행하며, 먼저 Sibling의 색상을 Successor의 부모의 색상으로 변경해준다. 이때의 Sibling은 바로 위에서 대입을 진행한, Successor→Parent→Right의 주소 값이므로, 서로 색상을 교환하는 과정인데, 이때, Successor→Parent→Right의 색상은 R,B 둘 다 가능하므로, 색상을 지정하는 것이 아닌 해당하는 색상으로 변경하는 작업을 진행한다.

- 237~238행은 Case *-2에서 고정적으로 변경되는 색상자리를 블랙으로 변경하는 과정이다.

- 이후 좌회전을 진행하며, Successor의 값을 (*Root)로 초기화를 진행한다.

RedBlackTree.c

```
243         } else {
244             Sibling = Successor->Parent->Left;
245 
246             if (Sibling->Color == RED) {
247                 Sibling->Color = BLACK;
248                 Successor->Parent->Color = RED;
249                 RBT_RotateRight(Root, Successor->Parent);
250                 Sibling = Successor->Parent->Left;
251             } else {
252                 if (Sibling->Right->Color == BLACK && Sibling->Left->Color == BLACK) {
253                     Sibling->Color = RED;
254                     Successor = Successor->Parent;
255                 } else {
256                     if (Sibling->Right->Color == RED) {
257                         Sibling->Right->Color = BLACK;
258                         Sibling->Color = RED;
259 
260                         RBT_RotateLeft(Root, Parent: Sibling);
261                         Sibling = Successor->Parent->Left;
262                     }
263 
264                     Sibling->Color = Successor->Parent->Color;
265                     Successor->Parent->Color = BLACK;
266                     Sibling->Left->Color = BLACK;
267                     RBT_RotateRight(Root, Successor->Parent);
268                     Successor = (*Root);
269                 }
270             }
271         }
272     }
273     Successor->Color = BLACK;
274 }
```

- 위 243줄~272줄의 내용은 위에서 해석한 부분의 내용을 Left→Rigth로만 변경해서 해석하면 된다.

- 273줄에서 최종적으로 Successor == 루트의 색상을 블랙으로 변경한다.

```
275 void RBT_PrintTree(RBTNode *Node, int Depth, int BlackCount) {
276     int i = 0;
277     char c = 'X';
278     int v = -1;
279     char cnt[100];
280
281     if (Node == NULL || Node == Nil)
282         return;
283
284     if (Node->Color == BLACK)
285         BlackCount++;
286
287     if (Node->Parent != NULL) {
288         v = Node->Parent->Data;
289
290         if (Node->Parent->Left == Node)
291             c = 'L';
292         else
293             c = 'R';
294     }
295
296     if (Node->Left == Nil && Node->Right == Nil)
297         sprintf(cnt, "--------- %d", BlackCount);
298     else
299         sprintf(cnt, "");
300
301     for (i = 0; i < Depth; i++) {
302         printf(" ");
303     }
304
305     printf("%d %s [%c,%d] %s\n", Node->Data, (Node->Color == RED) ? "RED" : "BLACK", c, v, cnt);
306
307     RBT_PrintTree( Node: Node->Left,  Depth: Depth + 1, BlackCount);
308     RBT_PrintTree( Node: Node->Right,  Depth: Depth + 1, BlackCount);
309 }
```

- 275행은 RBT를 출력하는 과정이다. RBT를 출력하는 과정은 재귀 호출을 통해 진행된다.

- 281행에서는 노드가 NULL이거나, Nil인 경우 출력을 생략한다.

- 284행에서는 노드의 색상이 블랙인 경우 BlackCound를 증가시킨다.

- main문에서 처음 PrintTree의 출발은 `RBT_PrintTree(Tree, 0, 0)`이다.

- 287~294행을 통해 왼쪽 노드인 경우 'L'을 오른쪽 노드인 경우 'R'의 값을 c에 초기화한다. 아닐 경우, 루트 노드에 해당하므로 해당 값은 초기 값인 'X'로 출력된다.

- 296~299행에서는 노드 왼쪽 오른쪽 자식이 Nil인 경우, 해당 노드까지 지나온 블랙 노드의 개수를 출력하며, 아닌 경우 출력을 따로 진행하지 않는다.

- 301행은 깊이에 따라 공백을 주어서 출력하는 노드의 깊이 차이를 발생 시킨다.

- 305행에서는 순서대로, 데이터 값, 해당 노드의 색깔, 왼쪽 자식인지 오른쪽 자식인지, 부모 노드의 값, 296~299행에서 처리한 cnt의 값을 출력한다.

- 307~308행은 재귀 호출을 진행한다.

`Test_RedBlackTree.c`

Test_RedBlackTree.c

```c
#include "RedBlackTree.h"

RBTNode *Nil;

int main(void) {
    RBTNode *Tree = NULL;
    RBTNode *Node = NULL;

    Nil = RBT_CreateNode( NewData: 0);
    Nil->Color = BLACK;

    while (1) {
        int cmd = 0;
        int param = 0;
        char buffer[10];

        printf("Enter command number :\n");
        printf("(1) Create a Node, (2) Remove a Node, (3) Search a Node\n");
        printf("(4) Display Tree (5) quit\n");
        printf("command number:");

        fgets(buffer, sizeof(buffer) - 1, stdin);
        sscanf(buffer, "%d", &cmd);

        if (cmd < 1 || cmd > 5) {
            printf("Invalid command number.\n");
            continue;
        } else if (cmd == 4) {
            RBT_PrintTree( Node: Tree,  Depth: 0,  BlackCount: 0);
            printf("\n");
            continue;
        } else if (cmd == 5)
            break;

        printf("Enter parameter (1~200) :\n");

```

```
37          fgets(buffer, sizeof(buffer) - 1, stdin);
38          sscanf(buffer, "%d", &param);
39
40          if (param < 1 || param > 200) {
41              printf("Invalid parameter.%d\n", param);
42              continue;
43          }
44
45          switch (cmd) {
46              case 1:
47                  RBT_InsertNode(&Tree,  NewNode: RBT_CreateNode( NewData: param));
48                  break;
49              case 2:
50                  Node = RBT_RemoveNode(&Tree,  Target: param);
51
52                  if (Node == NULL)
53                      printf("Not found node to delete: %d\n", param);
54                  else
55                      RBT_DestroyNode(Node);
56                  break;
57              case 3:
58                  Node = RBT_SearchNode(Tree,  Target: param);
59
60                  if (Node == NULL)
61                      printf("Not found node:%d\n", param);
62                  else
63                      printf("Found Node: %d(Color:%s)\n",
64                             Node->Data, (Node->Color == RED) ? "RED" : "BLACK");
65
66                  break;
67          }
68
69          printf("\n");
70      }
71
72      RBT_DestroyTree(Tree);
73      return 0;
74  }
```

- 위 코드는 메인 문으로 먼저 9~10행 에서 Nil의 값을 초기화 한다.

- 이후 12행부터는 while문을 돌면서 진행을 하며, 25행에서는 입력한 값이 1~5사이가 아니면 오류 메시지를 보내고 다시 출력한다.

- 29행은 4번을 고른 상황으로, Display Tree를 진행하는 것으로 `RBT_PrintTree(Tree, 0, 0)` 메소드를 실행하며, 재귀호출로써 RBT를 출력한다.

- 45행 스위치 문에서는 1~3번 케이스에 대해 각각 실행하는 것이다.

- 1번 케이스의 경우, 노드를 추가하는 과정

- 2번 케이스의 경우, 노드를 제거하는 과정

- 3번 케이스의 경우 노드를 찾는 과정이며, 노드를 찾은 경우 해당 색상에 따라 RED, BLACK을 출력한다.

- 마지막으로 while문을 5번 break;를 통해서 나오게 되면 RBT_DestroyTree(Tree)를 진행한다.

위 진행 과정에서 확인할 수 있는 오류는 두가지였다.

RedBlackTree.c

```c
void RBT_InsertNodeHelper(RBTNode **Tree, RBTNode *NewNode) {
    if ((*Tree) == NULL)
        (*Tree) = NewNode;

    if ((*Tree)->Data < NewNode->Data) {
        if ((*Tree)->Right == Nil) {
            (*Tree)->Right = NewNode;
            NewNode->Parent = (*Tree);
        } else
            RBT_InsertNodeHelper( Tree: &(*Tree)->Right, NewNode);
    } else if ((*Tree)->Data > NewNode->Data) {
        if ((*Tree)->Left == Nil) {
            (*Tree)->Left = NewNode;
            NewNode->Parent = (*Tree);
        } else
            RBT_InsertNodeHelper( Tree: &(*Tree)->Left, NewNode);
    }
}
```

→ 같은 데이터를 넣으면 오류 발생!!

- 첫 번째는, 노드를 삽입하는 과정인데, 해당 부분에서 같은 값에 대한 처리가 따로되어 있지 않아, 만약 같은 값을 넣게 되면 오류가 발생하게 된다.

- 오류가 발생하지 않기위해서는 같은 값을 처리해주면 될 것 같은데, BST에서도 마찬가지로 해당 처리 방법은 작성하는 사용자 나름대로의 규칙을 가지고 같은 값에 대해서 처해주면 되므로, = 연산자만 추가하면 에러가 해결된다.

- 두 번째 오류는 마지막 메인문에서 break;로 while문을 탈출하는 부분이다.

```c
} else if (cmd == 5)
    break;
```

- 그 이유는, 바로 아래의 부분 때문인데,

```
RBT_DestroyTree(Tree);
return 0;
```

- while문을 탈출하면 바로 만나게 되는 부분이 바로 RBT_DestroyTree(Tree)이다. 해당 부분을 접근하는 과정에서, 만약 최초에 프로그램을 시작한 뒤에, Tree 생성 없이 바로 5번을 입력해 while문을 탈출하게 되면, 비정상적인 종료가 일어나게 된다. 따라서 해당 부분을 처리하기 위해서는 다음과 같이 코드를 수정하면 된다.

수정 전

```
/Users/yujaeyeong/Developments/algorithm/univ-algorithm/230502/cmake-bui
Enter command number :
(1) Create a Node, (2) Remove a Node, (3) Search a Node
(4) Display Tree (5) quit
command number:5

Process finished with exit code 139 (interrupted by signal 11: SIGSEGV)
```

수정 후

```
if (Tree == NULL) {
    return 0;
} else {
    RBT_DestroyTree(Tree);
}
return 0;
```

```
/Users/yujaeyeong/Developments/algorithm/univ-algorithm/23
Enter command number :
(1) Create a Node, (2) Remove a Node, (3) Search a Node
(4) Display Tree (5) quit
command number:5

Process finished with exit code 0
```

- 바로 5번을 입력하여도 정상적으로 종료된다.

실행 결과