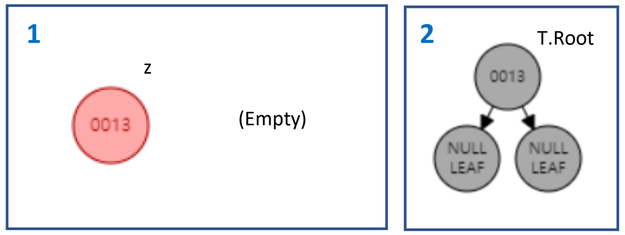
Assignment 9

Name: DONGWOOK LEE

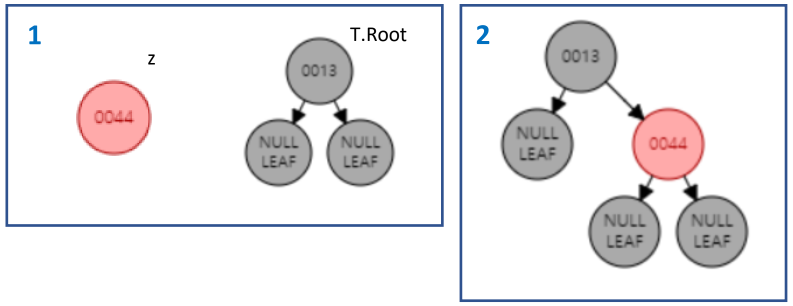
Problem 9.1 *Understanding Red Black Trees*

(a) Insert {13, 44, 37, 7, 22, 16} into an empty RB Tree in order



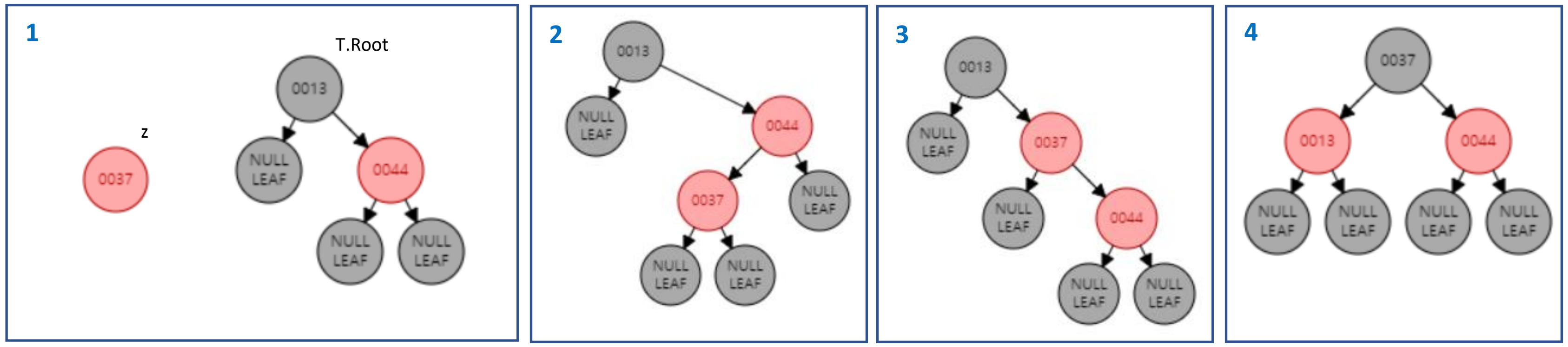
1. Insert 13 into an empty tree T.
   Violates RooB
   ➔ T.root.color = black;

2. Complete



1. Insert 44 into the tree.
   Compare keys of z and T.Root. ➔ z.key > T.Root.key
   Right child of T.Root is NULL, replace with z

2. Complete

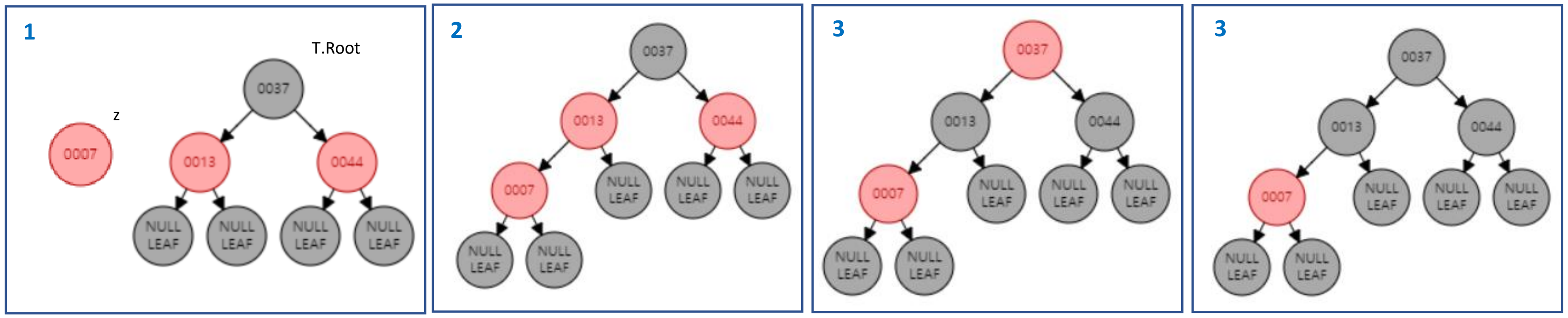


1. Insert 37 into the tree.
   Compare keys of z and T.Root. ➔ z.key > T.Root.key
   Compare keys of z and T.root.right ➔ z.key < T.root.right.key

2. T.Root.right.left is NULL, replace with z. Violates BredB
   z = z.p.left & z.p = z.p.p.right & z.p.p.left = black (Symmetric to Case 2)
   ➔ Right rotation (T, z.p)

3. z = z.p.right & z.p = z.p.p.right & z.p.p.left = black (Symmetric to Case 3)
   ➔ Left rotation (T, z.p.p)

4. Complete

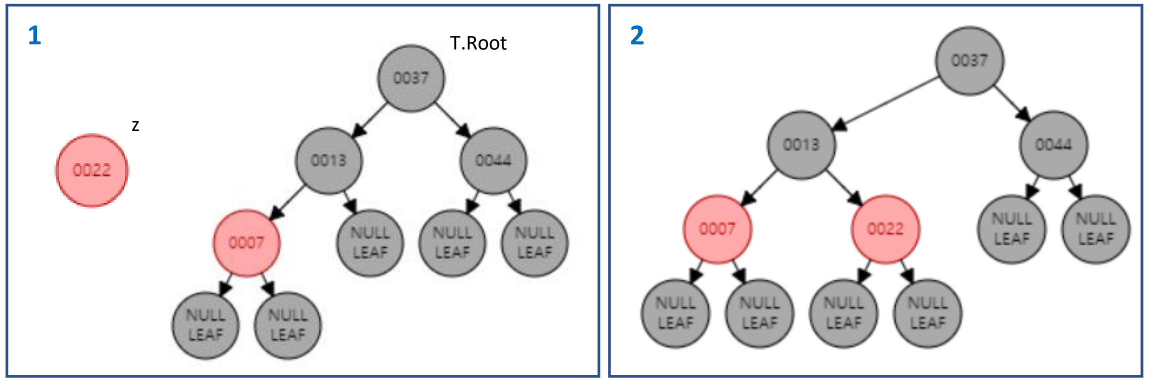


1. Insert 7 into the tree.
   Compare keys of z and T.root. ➔ z.key < T.root.key
   Compare keys of z and T.root.left ➔ z.key < T.root.left.key

2. T.root.left.left is NULL, replace with z. Violates BredB
   z.p = z.p.p.left & z.p.p.right = red (Case 1)
   ➔ z.p.color = black; z.p.p.right.color = black; z.p.p.color = red;

3. Violates RooB
   ➔ T.root.color = black;

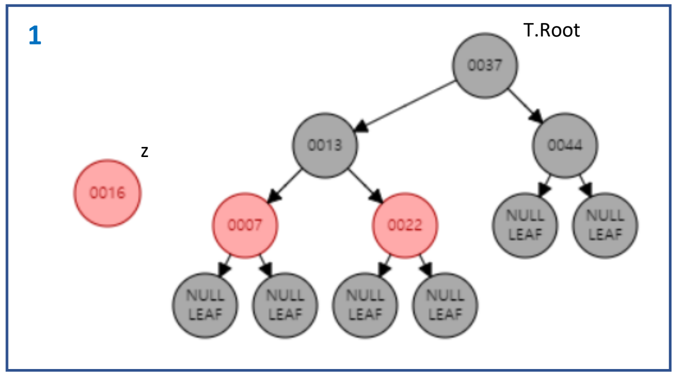
4. Complete



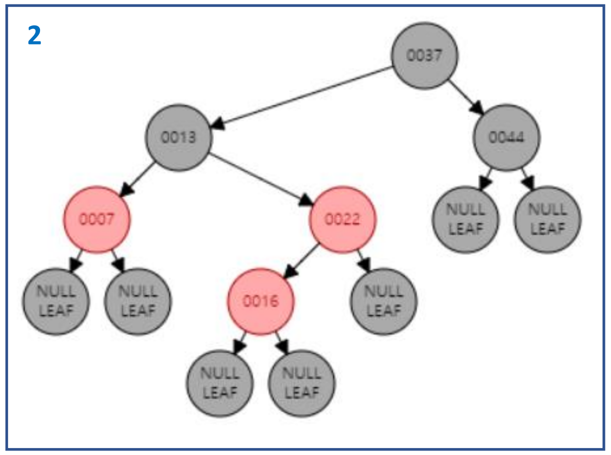
1. Insert 22 into the tree.
   Compare keys of z and T.root. ➔ z.key < T.root.key
   Compare keys of z and T.root.left ➔ z.key > T.root.left.key

2. T.root.left.right is NULL, replace with z.
   Complete

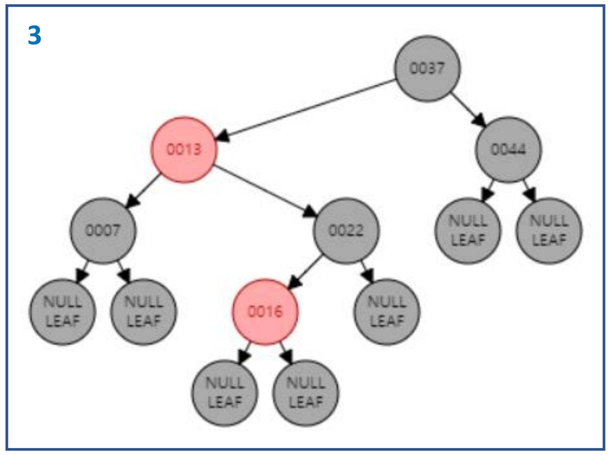
1



2



3

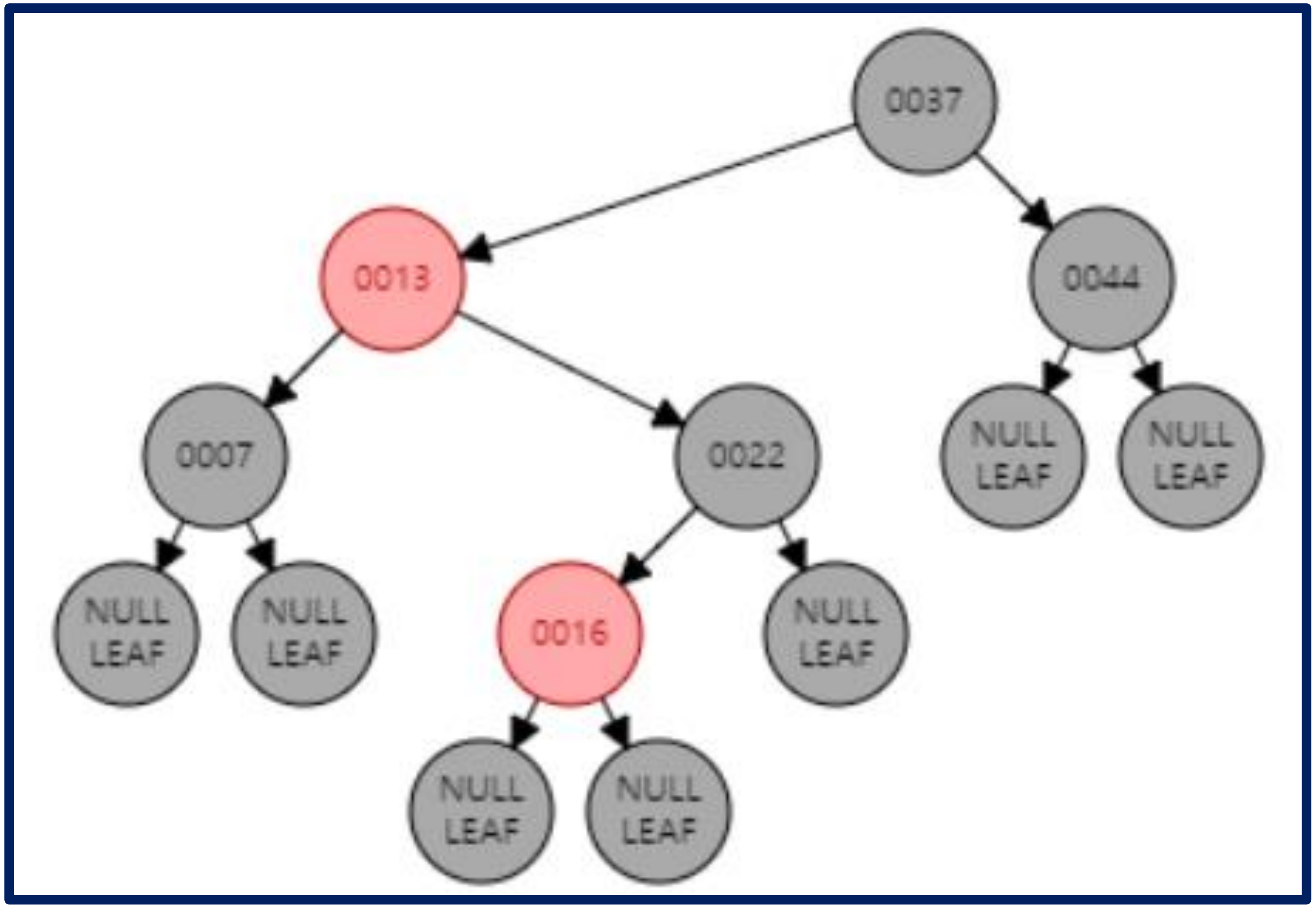


1. Insert 16 into the tree.
   Compare keys of z and T.root. → z.key < T.root.key
   Compare keys of z and T.root.left → z.key > T.root.left.key
   Compare keys of z and T.root.left.right → z.key < T.root.left.right.key

2. T.root.left.right.left is NULL, replace with z. Violates BredB
   z.p = z.p.p.right & z.p.p.left = red (Symmetric to Case 1)
   → z.p.color = black; z.p.p.left.color = black; z.p.p.color = red;

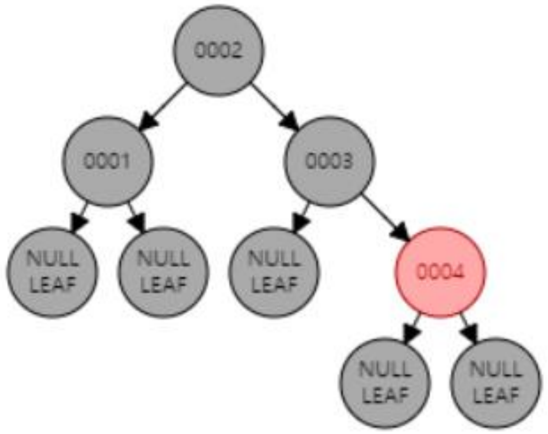
3. Complete

Result:

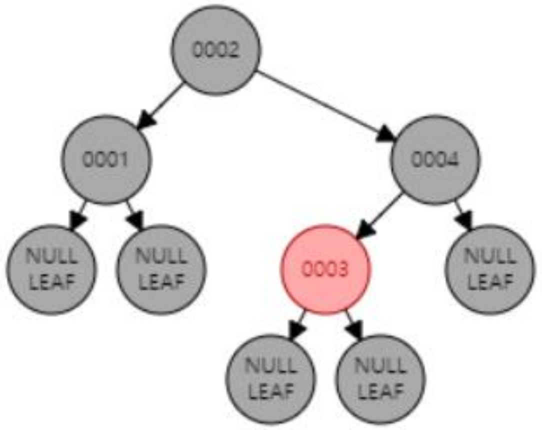


(b) Make all valid RB Tree with the elements {1, 2, 3, 4}.

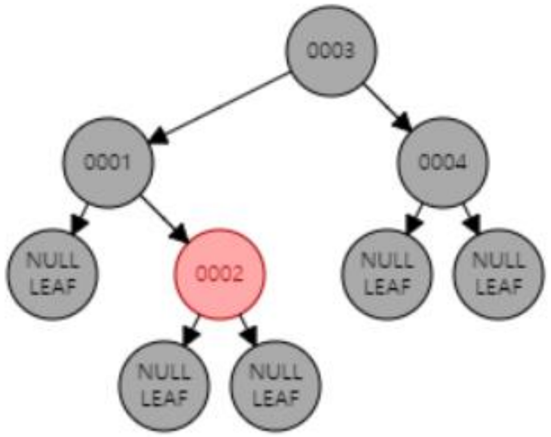
1 ➔ 2 ➔ 3 ➔ 4
1 ➔ 3 ➔ 2 ➔ 4
2 ➔ 1 ➔ 3 ➔ 4
2 ➔ 3 ➔ 1 ➔ 4
3 ➔ 1 ➔ 2 ➔ 4
3 ➔ 2 ➔ 1 ➔ 4



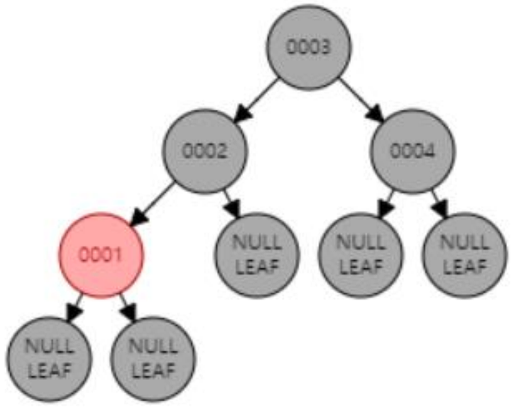
1 ➔ 2 ➔ 4 ➔ 3
1 ➔ 4 ➔ 2 ➔ 3
2 ➔ 1 ➔ 4 ➔ 3
2 ➔ 4 ➔ 1 ➔ 3
4 ➔ 1 ➔ 2 ➔ 3
4 ➔ 2 ➔ 1 ➔ 3



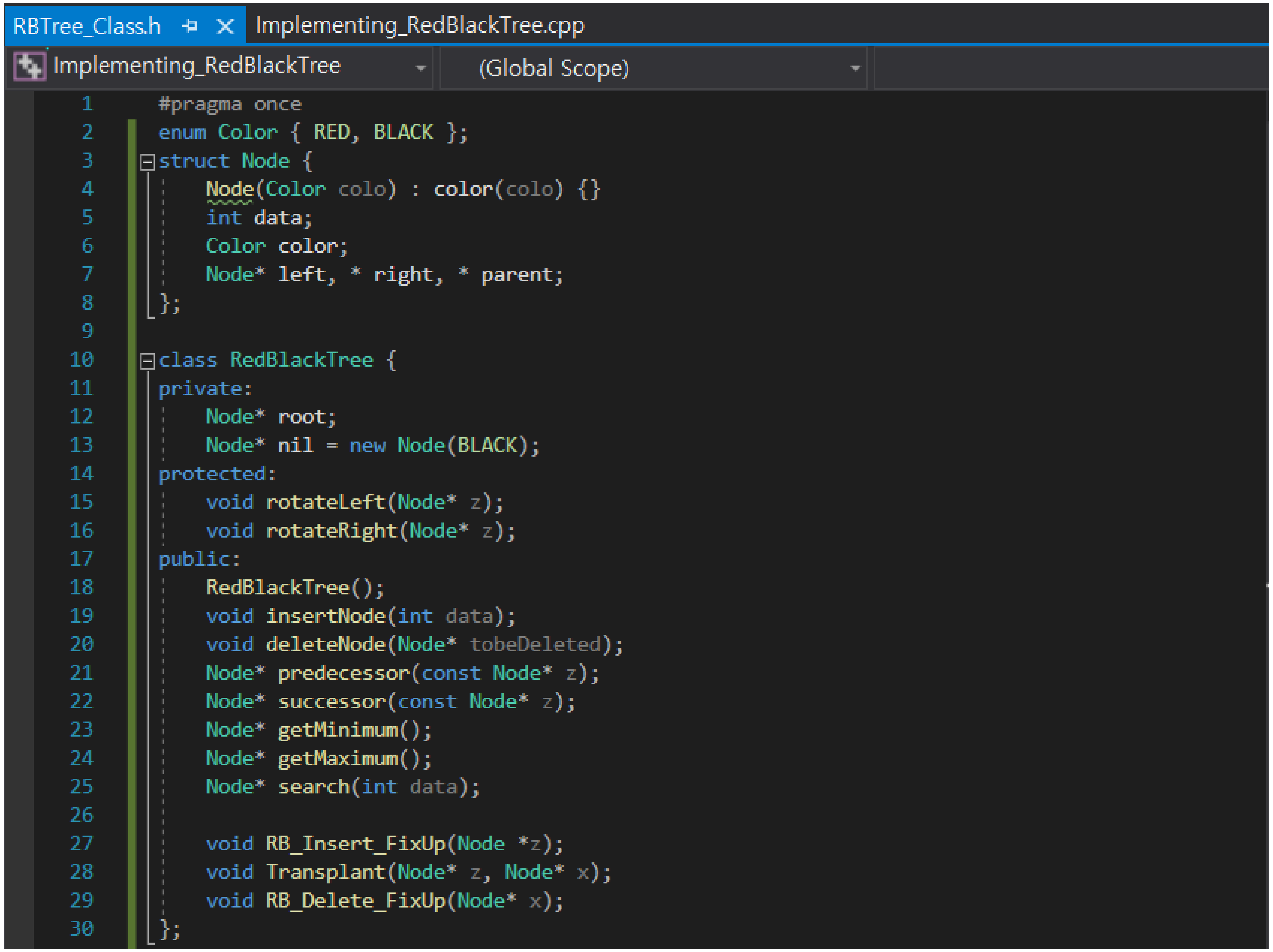
1 ➔ 3 ➔ 4 ➔ 2
1 ➔ 4 ➔ 3 ➔ 2
3 ➔ 1 ➔ 4 ➔ 2
3 ➔ 4 ➔ 1 ➔ 2
4 ➔ 1 ➔ 3 ➔ 2
4 ➔ 3 ➔ 1 ➔ 2



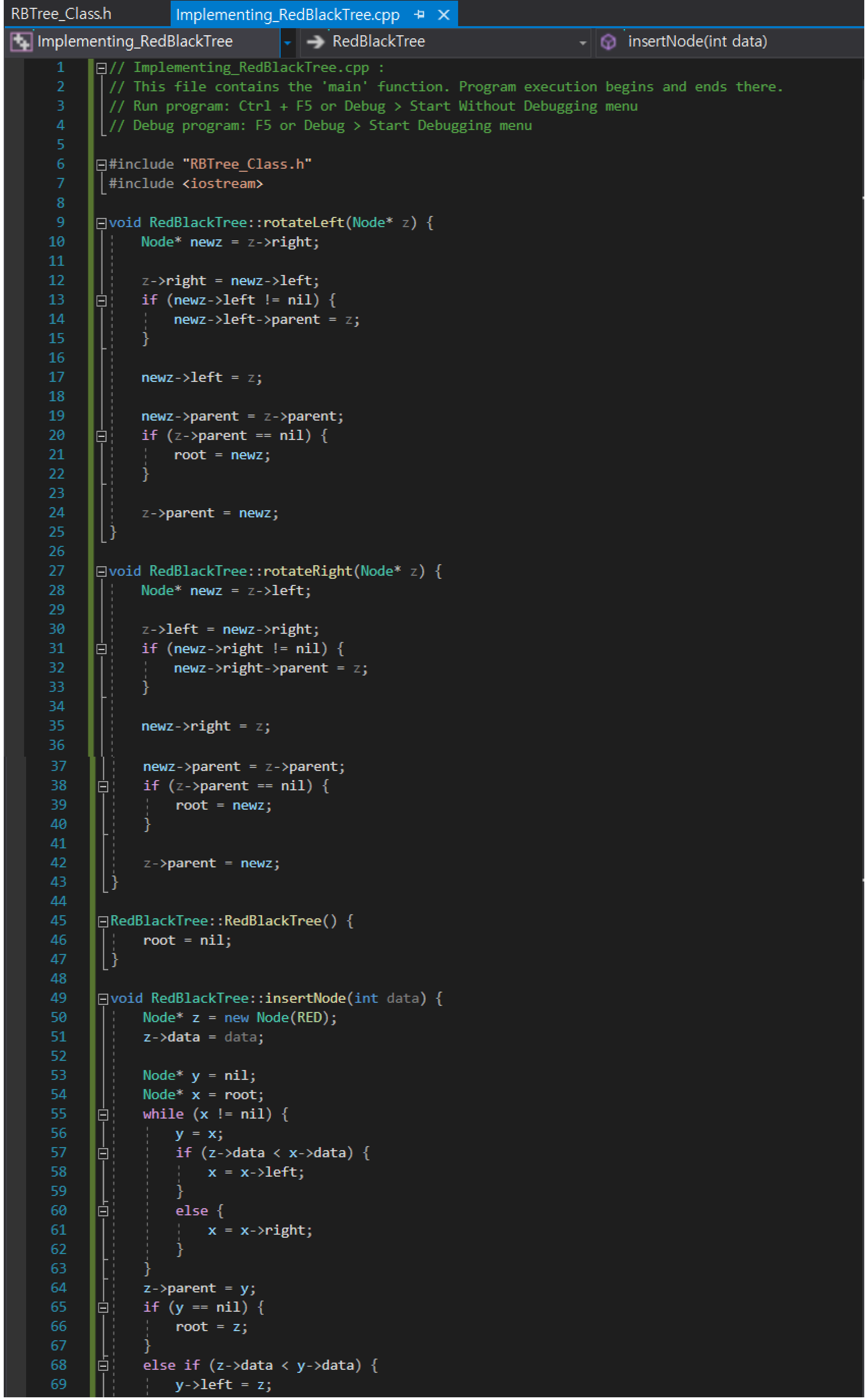
2 ➔ 4 ➔ 3 ➔ 1
2 ➔ 3 ➔ 4 ➔ 1
3 ➔ 2 ➔ 4 ➔ 1
3 ➔ 4 ➔ 2 ➔ 1
4 ➔ 2 ➔ 3 ➔ 1
4 ➔ 3 ➔ 2 ➔ 1



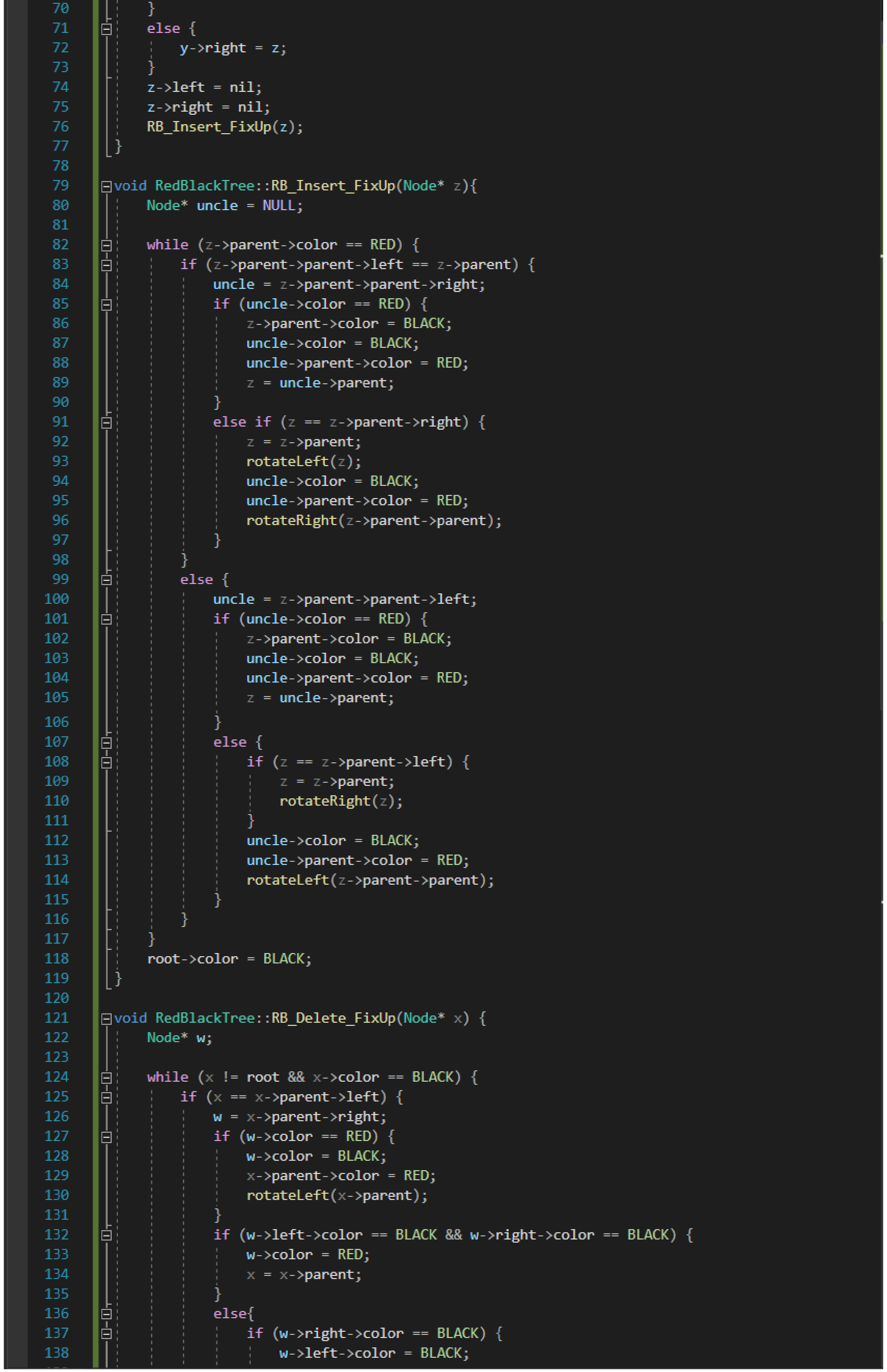
Problem 9.2 *Implementing Red Black Trees*



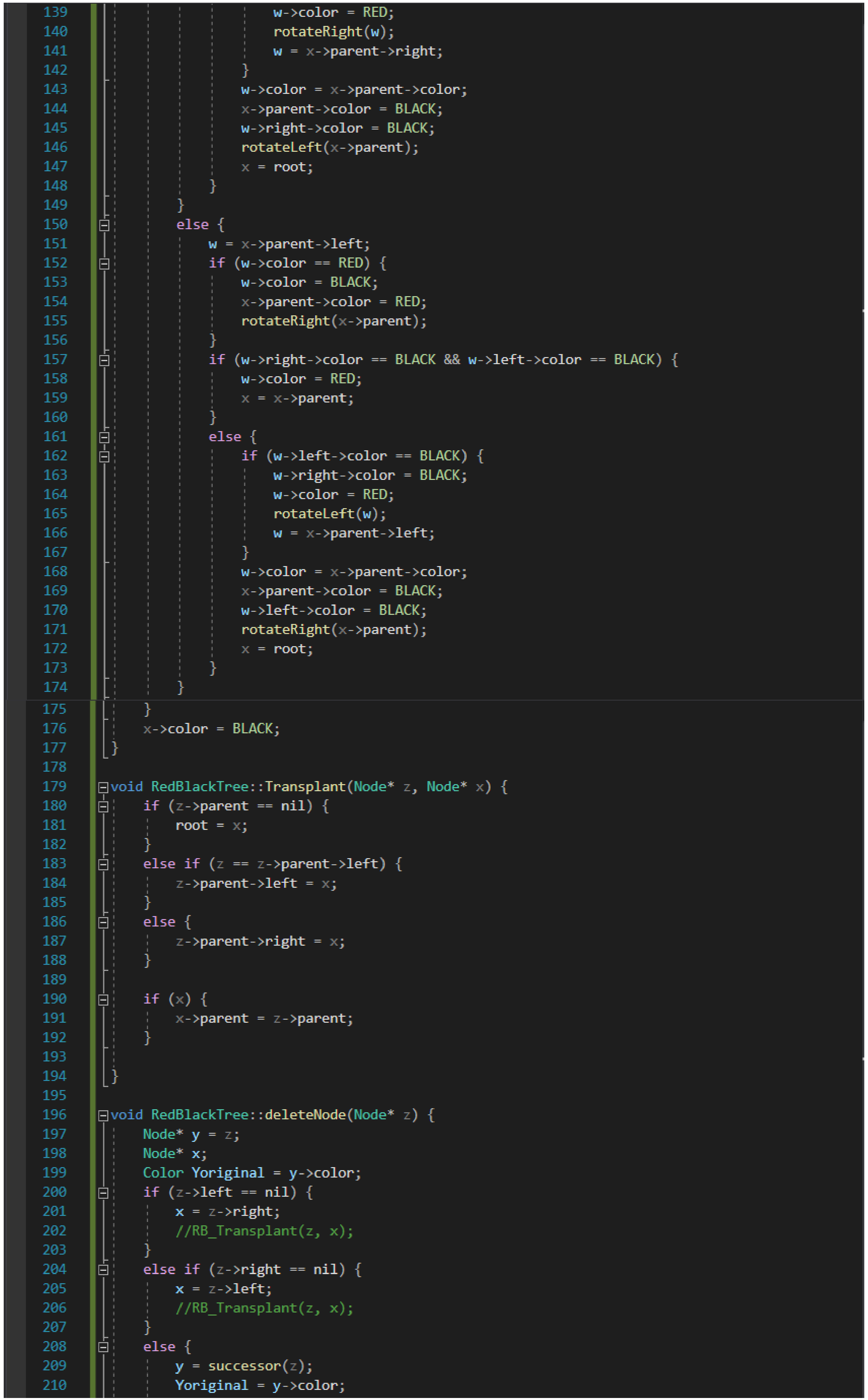
```cpp
#pragma once
enum Color { RED, BLACK };
struct Node {
    Node(Color colo) : color(colo) {}
    int data;
    Color color;
    Node* left, * right, * parent;
};

class RedBlackTree {
private:
    Node* root;
    Node* nil = new Node(BLACK);
protected:
    void rotateLeft(Node* z);
    void rotateRight(Node* z);
public:
    RedBlackTree();
    void insertNode(int data);
    void deleteNode(Node* tobeDeleted);
    Node* predecessor(const Node* z);
    Node* successor(const Node* z);
    Node* getMinimum();
    Node* getMaximum();
    Node* search(int data);

    void RB_Insert_FixUp(Node *z);
    void Transplant(Node* z, Node* x);
    void RB_Delete_FixUp(Node* x);
};
```



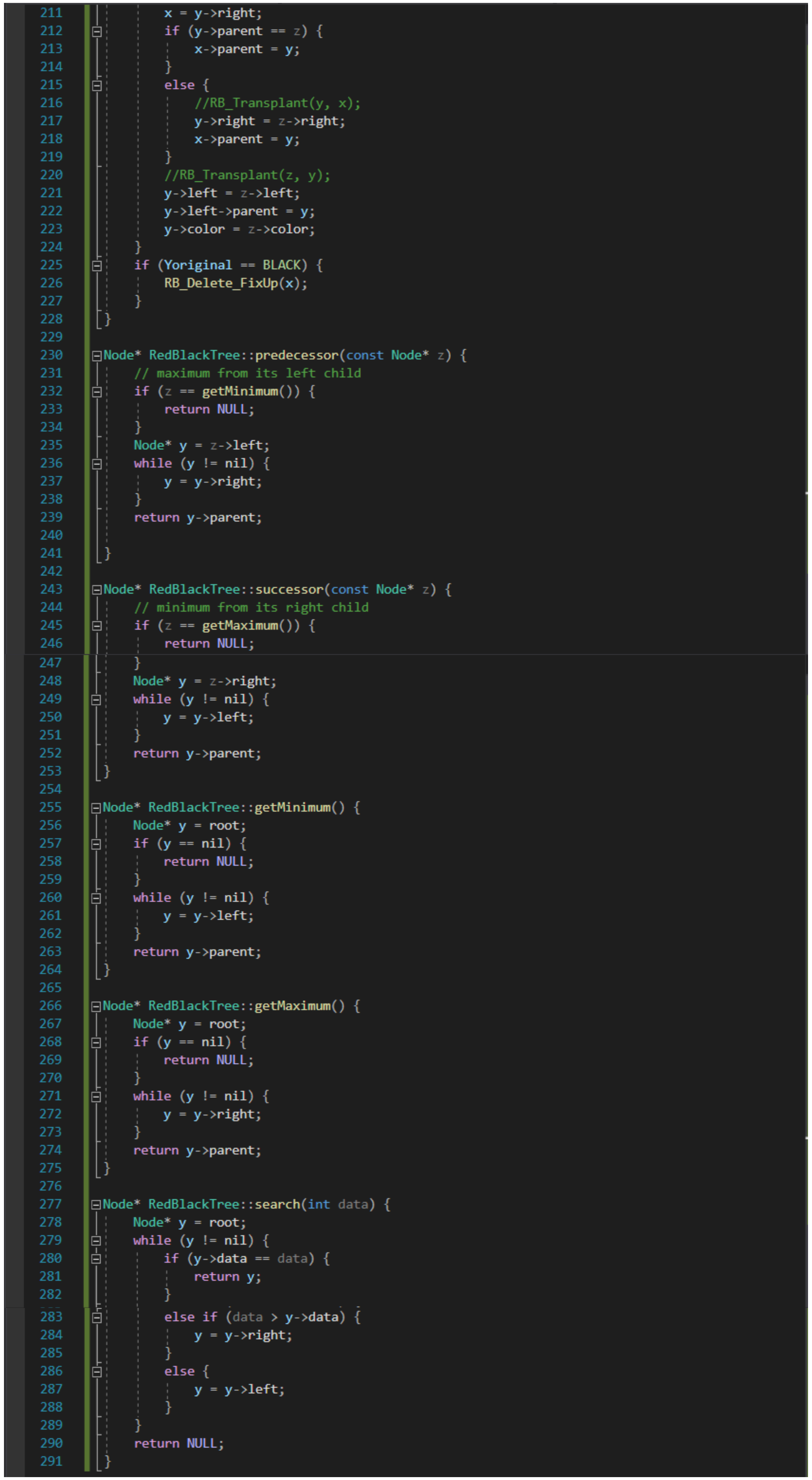
```
// Implementing_RedBlackTree.cpp :
// This file contains the 'main' function. Program execution begins and ends there.
// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

#include "RBTree_Class.h"
#include <iostream>

void RedBlackTree::rotateLeft(Node* z) {
    Node* newz = z->right;

    z->right = newz->left;
    if (newz->left != nil) {
        newz->left->parent = z;
    }

    newz->left = z;

    newz->parent = z->parent;
    if (z->parent == nil) {
        root = newz;
    }

    z->parent = newz;
}

void RedBlackTree::rotateRight(Node* z) {
    Node* newz = z->left;

    z->left = newz->right;
    if (newz->right != nil) {
        newz->right->parent = z;
    }

    newz->right = z;

    newz->parent = z->parent;
    if (z->parent == nil) {
        root = newz;
    }

    z->parent = newz;
}

RedBlackTree::RedBlackTree() {
    root = nil;
}

void RedBlackTree::insertNode(int data) {
    Node* z = new Node(RED);
    z->data = data;

    Node* y = nil;
    Node* x = root;
    while (x != nil) {
        y = x;
        if (z->data < x->data) {
            x = x->left;
        }
        else {
            x = x->right;
        }
    }
    z->parent = y;
    if (y == nil) {
        root = z;
    }
    else if (z->data < y->data) {
        y->left = z;
```



```
    }
    else {
        y->right = z;
    }
    z->left = nil;
    z->right = nil;
    RB_Insert_FixUp(z);
}

void RedBlackTree::RB_Insert_FixUp(Node* z){
    Node* uncle = NULL;

    while (z->parent->color == RED) {
        if (z->parent->parent->left == z->parent) {
            uncle = z->parent->parent->right;
            if (uncle->color == RED) {
                z->parent->color = BLACK;
                uncle->color = BLACK;
                uncle->parent->color = RED;
                z = uncle->parent;
            }
            else if (z == z->parent->right) {
                z = z->parent;
                rotateLeft(z);
                uncle->color = BLACK;
                uncle->parent->color = RED;
                rotateRight(z->parent->parent);
            }
        }
        else {
            uncle = z->parent->parent->left;
            if (uncle->color == RED) {
                z->parent->color = BLACK;
                uncle->color = BLACK;
                uncle->parent->color = RED;
                z = uncle->parent;

            }
            else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rotateRight(z);
                }
                uncle->color = BLACK;
                uncle->parent->color = RED;
                rotateLeft(z->parent->parent);
            }
        }
    }
    root->color = BLACK;
}

void RedBlackTree::RB_Delete_FixUp(Node* x) {
    Node* w;

    while (x != root && x->color == BLACK) {
        if (x == x->parent->left) {
            w = x->parent->right;
            if (w->color == RED) {
                w->color = BLACK;
                x->parent->color = RED;
                rotateLeft(x->parent);
            }
            if (w->left->color == BLACK && w->right->color == BLACK) {
                w->color = RED;
                x = x->parent;
            }
            else{
                if (w->right->color == BLACK) {
                    w->left->color = BLACK;
```



```
                    w->color = RED;
                    rotateRight(w);
                    w = x->parent->right;
                }
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->right->color = BLACK;
                rotateLeft(x->parent);
                x = root;
            }
        }
        else {
            w = x->parent->left;
            if (w->color == RED) {
                w->color = BLACK;
                x->parent->color = RED;
                rotateRight(x->parent);
            }
            if (w->right->color == BLACK && w->left->color == BLACK) {
                w->color = RED;
                x = x->parent;
            }
            else {
                if (w->left->color == BLACK) {
                    w->right->color = BLACK;
                    w->color = RED;
                    rotateLeft(w);
                    w = x->parent->left;
                }
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->left->color = BLACK;
                rotateRight(x->parent);
                x = root;
            }
        }
    }
    x->color = BLACK;
}

void RedBlackTree::Transplant(Node* z, Node* x) {
    if (z->parent == nil) {
        root = x;
    }
    else if (z == z->parent->left) {
        z->parent->left = x;
    }
    else {
        z->parent->right = x;
    }

    if (x) {
        x->parent = z->parent;
    }

}

void RedBlackTree::deleteNode(Node* z) {
    Node* y = z;
    Node* x;
    Color Yoriginal = y->color;
    if (z->left == nil) {
        x = z->right;
        //RB_Transplant(z, x);
    }
    else if (z->right == nil) {
        x = z->left;
        //RB_Transplant(z, x);
    }
    else {
        y = successor(z);
        Yoriginal = y->color;
```



```
        x = y->right;
        if (y->parent == z) {
            x->parent = y;
        }
        else {
            //RB_Transplant(y, x);
            y->right = z->right;
            x->parent = y;
        }
        //RB_Transplant(z, y);
        y->left = z->left;
        y->left->parent = y;
        y->color = z->color;
    }
    if (Yoriginal == BLACK) {
        RB_Delete_FixUp(x);
    }
}

Node* RedBlackTree::predecessor(const Node* z) {
    // maximum from its left child
    if (z == getMinimum()) {
        return NULL;
    }
    Node* y = z->left;
    while (y != nil) {
        y = y->right;
    }
    return y->parent;

}

Node* RedBlackTree::successor(const Node* z) {
    // minimum from its right child
    if (z == getMaximum()) {
        return NULL;
    }
    Node* y = z->right;
    while (y != nil) {
        y = y->left;
    }
    return y->parent;
}

Node* RedBlackTree::getMinimum() {
    Node* y = root;
    if (y == nil) {
        return NULL;
    }
    while (y != nil) {
        y = y->left;
    }
    return y->parent;
}

Node* RedBlackTree::getMaximum() {
    Node* y = root;
    if (y == nil) {
        return NULL;
    }
    while (y != nil) {
        y = y->right;
    }
    return y->parent;
}

Node* RedBlackTree::search(int data) {
    Node* y = root;
    while (y != nil) {
        if (y->data == data) {
            return y;
        }
        else if (data > y->data) {
            y = y->right;
        }
        else {
            y = y->left;
        }
    }
    return NULL;
}
```