```
#include <stdio.h>
#include <stdlib.h>

enum Color { RED, BLACK };

struct Node {
int data;
enum Color color;
struct Node *left, *right, *parent;
};

// Utility function to create a new node
struct Node* createNode(int data) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
                newNode->color = RED; // New nodes are  inserted as RED
newNode->left = newNode->right = newNode->parent = NULL;
return newNode;
}

// Perform an in-order traversal of the Red-Black Tree
void inorderTraversal(struct Node* root) {
if (root == NULL) return;

inorderTraversal(root->left);
printf("%d ", root->data);
inorderTraversal(root->right);
}

// Left rotation
```

```
struct Node* rotateLeft(struct Node* root, struct Node* pt) {
struct Node* pt_right = pt->right;

pt->right = pt_right->left;

if (pt->right != NULL)
pt->right->parent = pt;

pt_right->parent = pt->parent;

if (pt->parent == NULL)
root = pt_right;
else if (pt == pt->parent->left)
pt->parent->left = pt_right;
else
pt->parent->right = pt_right;

pt_right->left = pt;
pt->parent = pt_right;

return root;
}

// Right rotation
struct Node* rotateRight(struct Node* root, struct Node* pt) {
struct Node* pt_left = pt->left;

pt->left = pt_left->right;

if (pt->left != NULL)
pt->left->parent = pt;
```

```
pt_left->parent = pt->parent;

if (pt->parent == NULL)
root = pt_left;
else if (pt == pt->parent->left)
pt->parent->left = pt_left;
else
pt->parent->right = pt_left;

pt_left->right = pt;
pt->parent = pt_left;

return root;
}

// Fix Red-Black Tree violations
struct Node* fixViolation(struct Node* root, struct Node* pt) {
struct Node* parent_pt = NULL;
struct Node* grand_parent_pt = NULL;

while ((pt != root) && (pt->color != BLACK) && (pt->parent->color == RED)) {
parent_pt = pt->parent;
grand_parent_pt = pt->parent->parent;

// Case A: Parent is left child of grandparent
if (parent_pt == grand_parent_pt->left) {
struct Node* uncle_pt = grand_parent_pt->right;

// Case 1: The uncle is RED, recoloring
if (uncle_pt != NULL && uncle_pt->color == RED) {
```

```
grand_parent_pt->color = RED;
parent_pt->color = BLACK;
uncle_pt->color = BLACK;
pt = grand_parent_pt;
} else {
// Case 2: pt is right child of its parent, left-rotation needed
if (pt == parent_pt->right) {
root = rotateLeft(root, parent_pt);
pt = parent_pt;
parent_pt = pt->parent;
}

// Case 3: pt is left child, right-rotation needed
root = rotateRight(root, grand_parent_pt);
enum Color temp = parent_pt->color;
parent_pt->color = grand_parent_pt->color;
grand_parent_pt->color = temp;
pt = parent_pt;
}
}
// Case B: Parent is right child of grandparent
else {
struct Node* uncle_pt = grand_parent_pt->left;

// Case 1: The uncle is RED, recoloring
if (uncle_pt != NULL && uncle_pt->color == RED) {
grand_parent_pt->color = RED;
parent_pt->color = BLACK;
uncle_pt->color = BLACK;
pt = grand_parent_pt;
} else {
```

```
// Case 2: pt is left child of its parent, right-rotation needed
if (pt == parent_pt->left) {
root = rotateRight(root, parent_pt);
pt = parent_pt;
parent_pt = pt->parent;
}

// Case 3: pt is right child, left-rotation needed
root = rotateLeft(root, grand_parent_pt);
enum Color temp = parent_pt->color;
parent_pt->color = grand_parent_pt->color;
grand_parent_pt->color = temp;
pt = parent_pt;
}
}
}

root->color = BLACK;
return root;
}

// Insert a new node into the Red-Black Tree
struct Node* insert(struct Node* root, struct Node* pt) {
// Binary search tree insertion
if (root == NULL)
return pt;

if (pt->data < root->data) {
root->left = insert(root->left, pt);
root->left->parent = root;
} else if (pt->data > root->data) {
```

```
root->right = insert(root->right, pt);
root->right->parent = root;
}

return root;
}

// Main function to insert a new node and fix Red-Black Tree violations
struct Node* insertRBTree(struct Node* root, int data) {
struct Node* pt = createNode(data);
root = insert(root, pt);
root = fixViolation(root, pt);
return root;
}

int main() {
struct Node* root = NULL;

// Insert nodes
root = insertRBTree(root, 10);
root = insertRBTree(root, 20);
root = insertRBTree(root, 30);
root = insertRBTree(root, 15);
root = insertRBTree(root, 25);

// In-order traversal of the tree
printf("In-order traversal of the Red-Black Tree:\n");
inorderTraversal(root);

return 0;
}
```