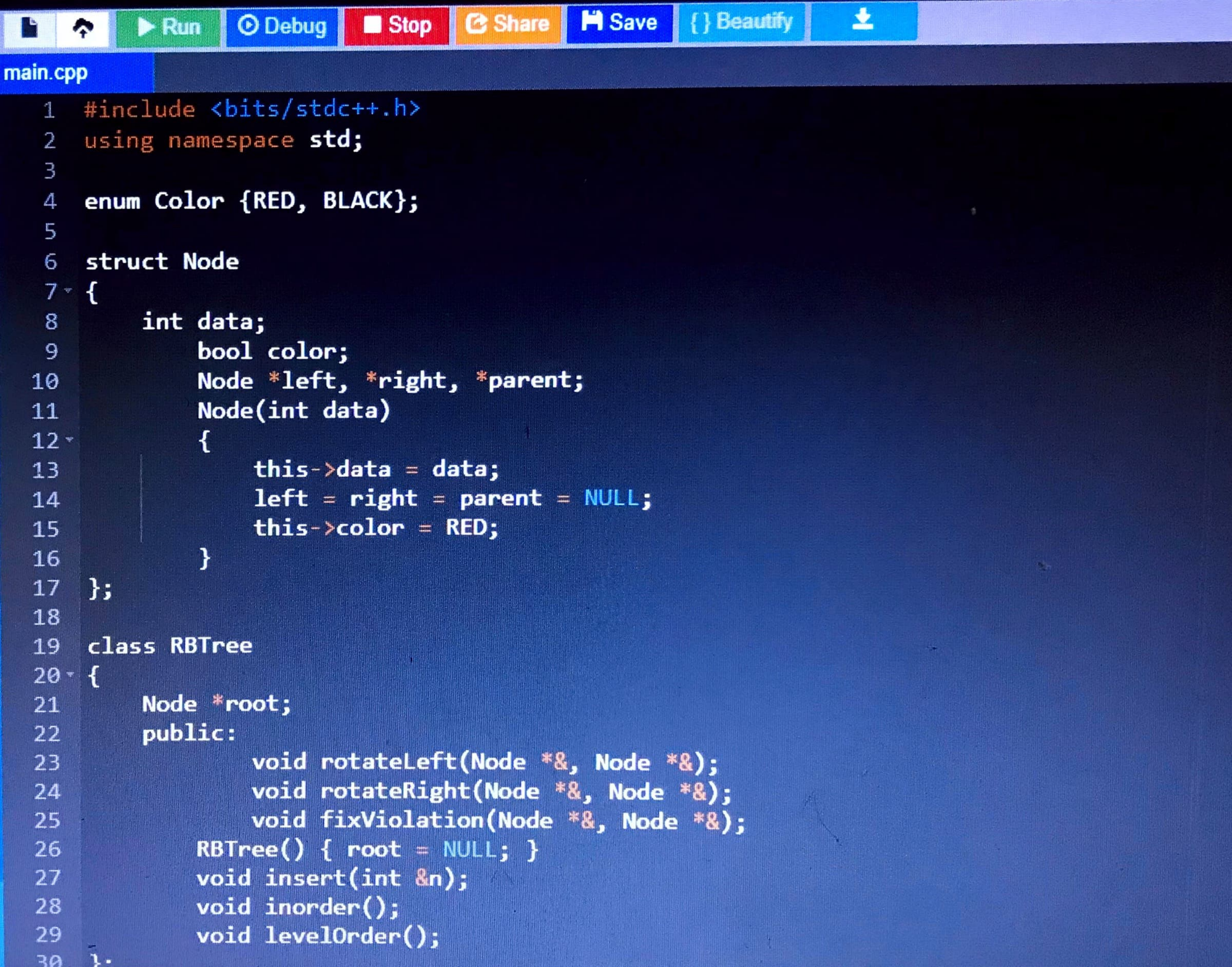
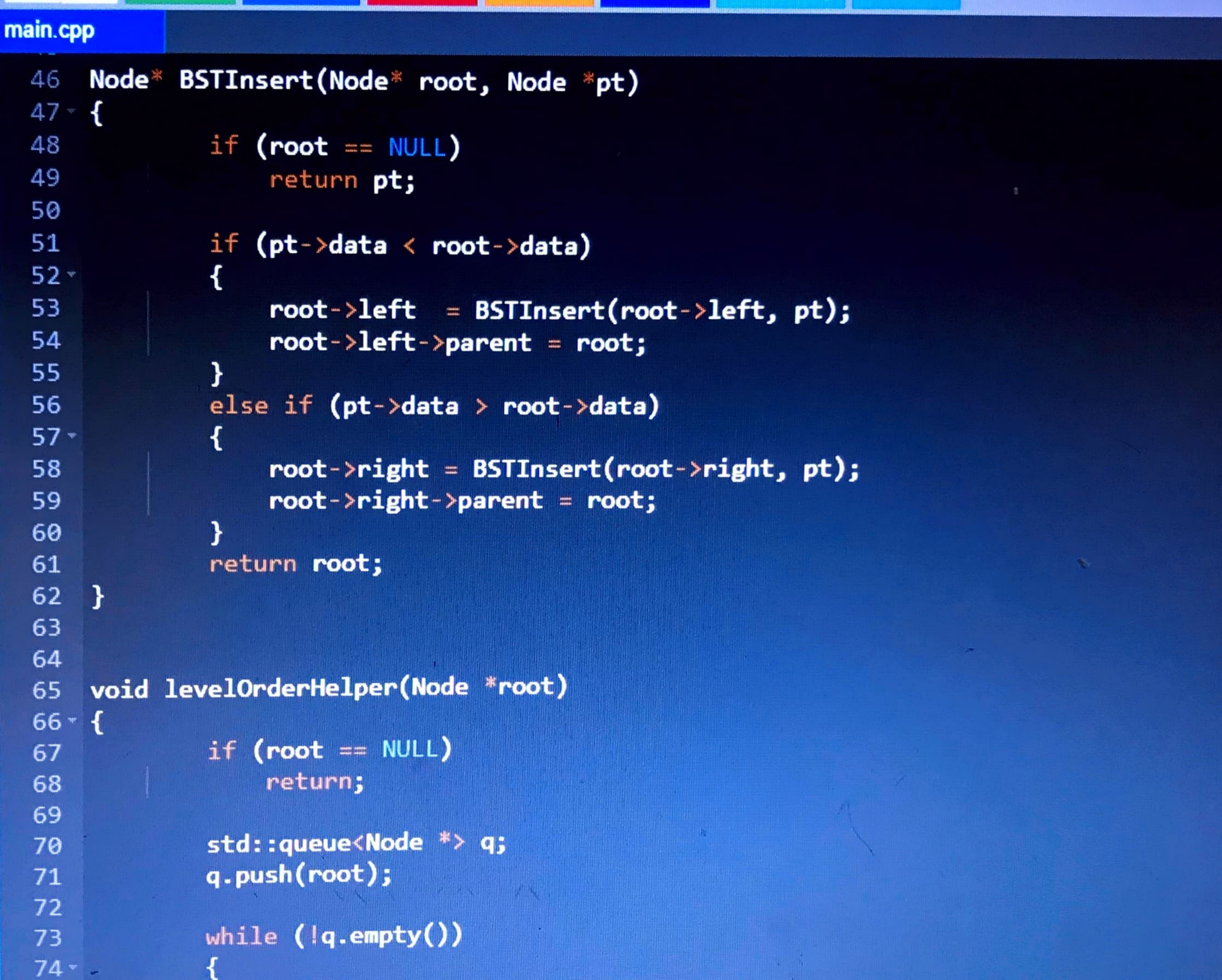
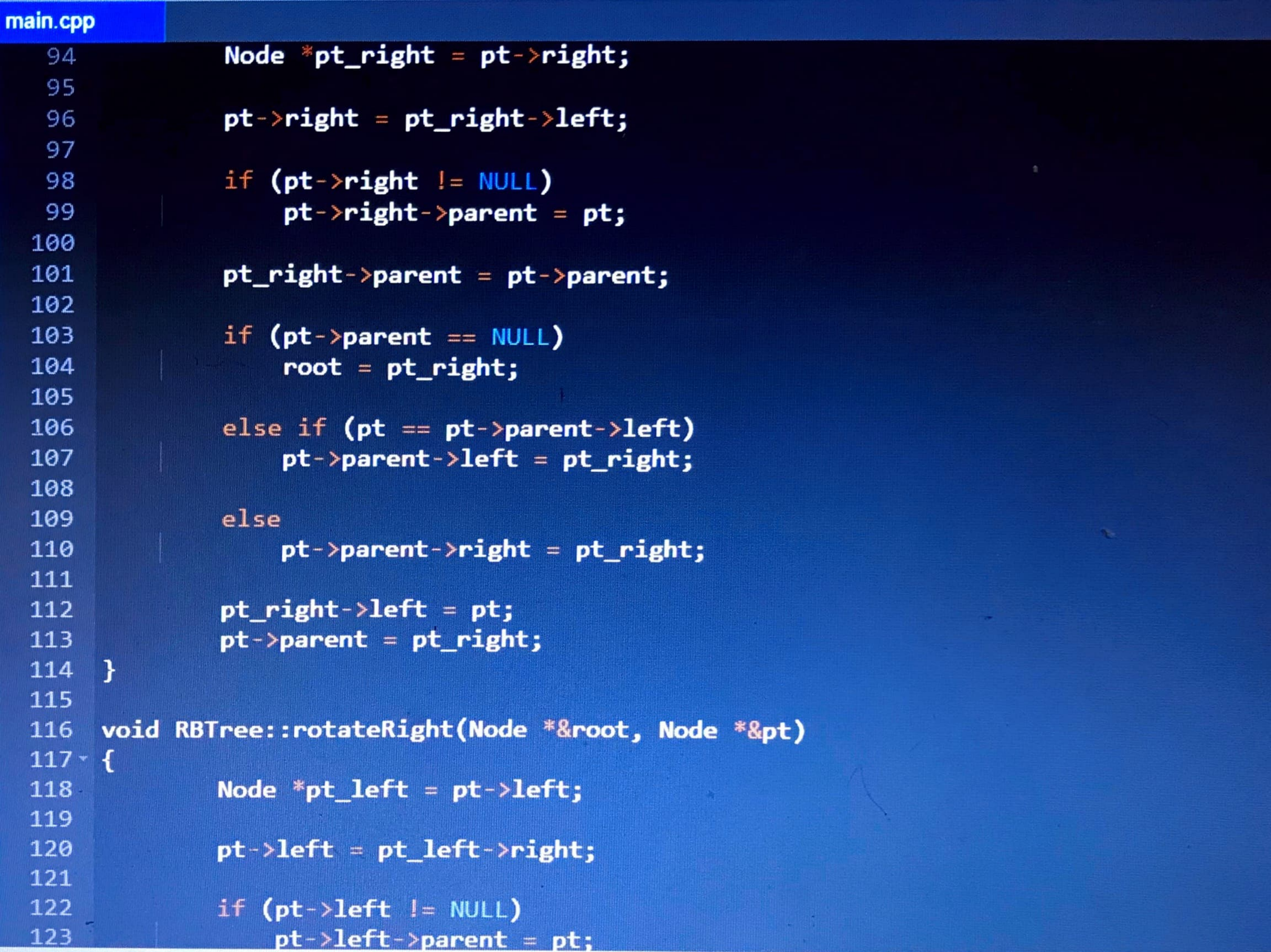


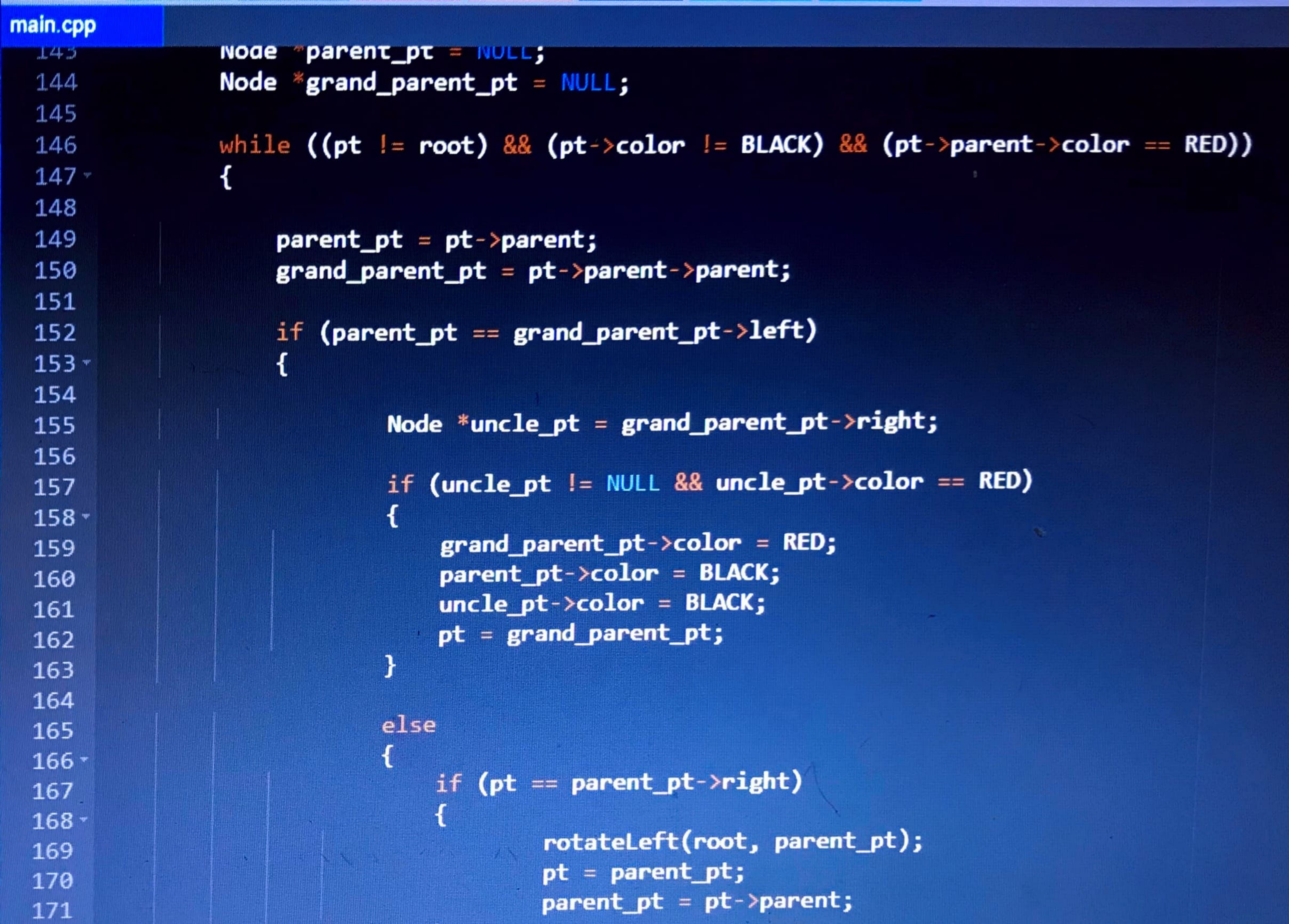
```cpp
#include <bits/stdc++.h>
using namespace std;

enum Color {RED, BLACK};

struct Node
{
    int data;
        bool color;
        Node *left, *right, *parent;
        Node(int data)
        {
            this->data = data;
            left = right = parent = NULL;
            this->color = RED;
        }
};

class RBTree
{
    Node *root;
    public:
            void rotateLeft(Node *&, Node *&);
            void rotateRight(Node *&, Node *&);
            void fixViolation(Node *&, Node *&);
        RBTree() { root = NULL; }
        void insert(int &n);
        void inorder();
        void levelOrder();
};
```



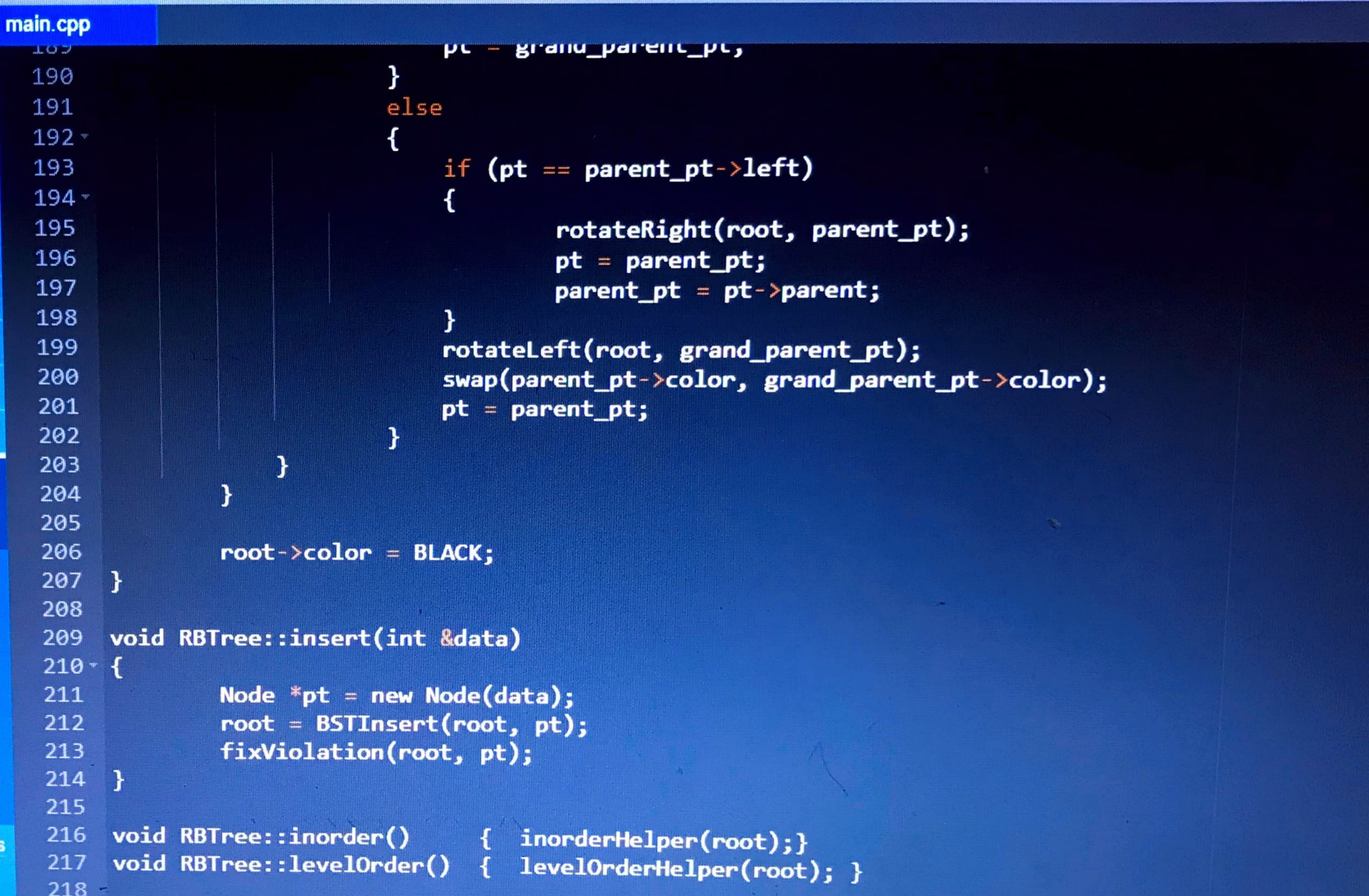
```
Node* BSTInsert(Node* root, Node *pt)
{
        if (root == NULL)
            return pt;

        if (pt->data < root->data)
        {
            root->left  = BSTInsert(root->left, pt);
            root->left->parent = root;
        }
        else if (pt->data > root->data)
        {
            root->right = BSTInsert(root->right, pt);
            root->right->parent = root;
        }
        return root;
}


void levelOrderHelper(Node *root)
{
        if (root == NULL)
            return;

        std::queue<Node *> q;
        q.push(root);

        while (!q.empty())
        {
```



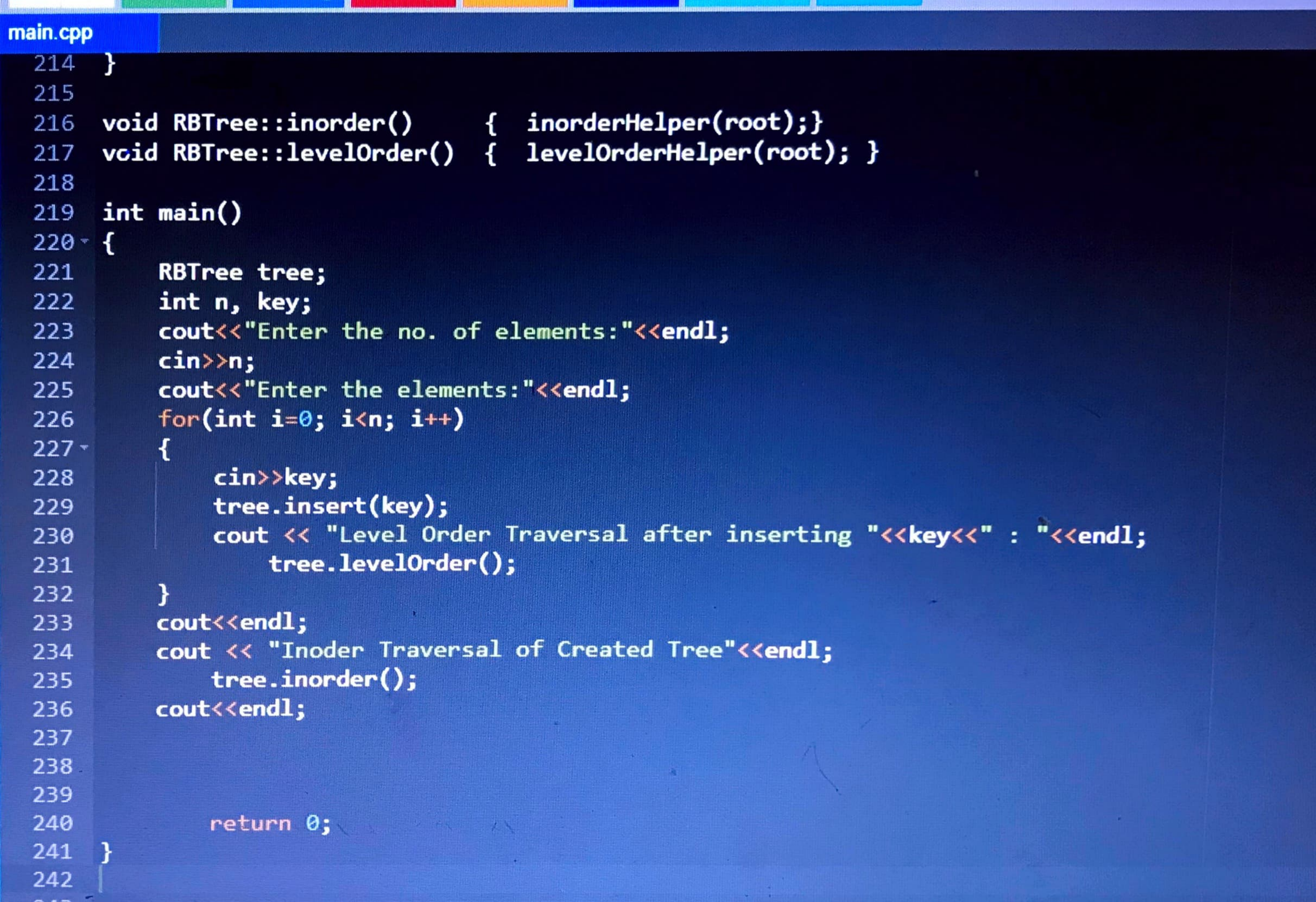
```cpp
        Node *pt_right = pt->right;

        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;
}

void RBTree::rotateRight(Node *&root, Node *&pt)
{
        Node *pt_left = pt->left;

        pt->left = pt_left->right;

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



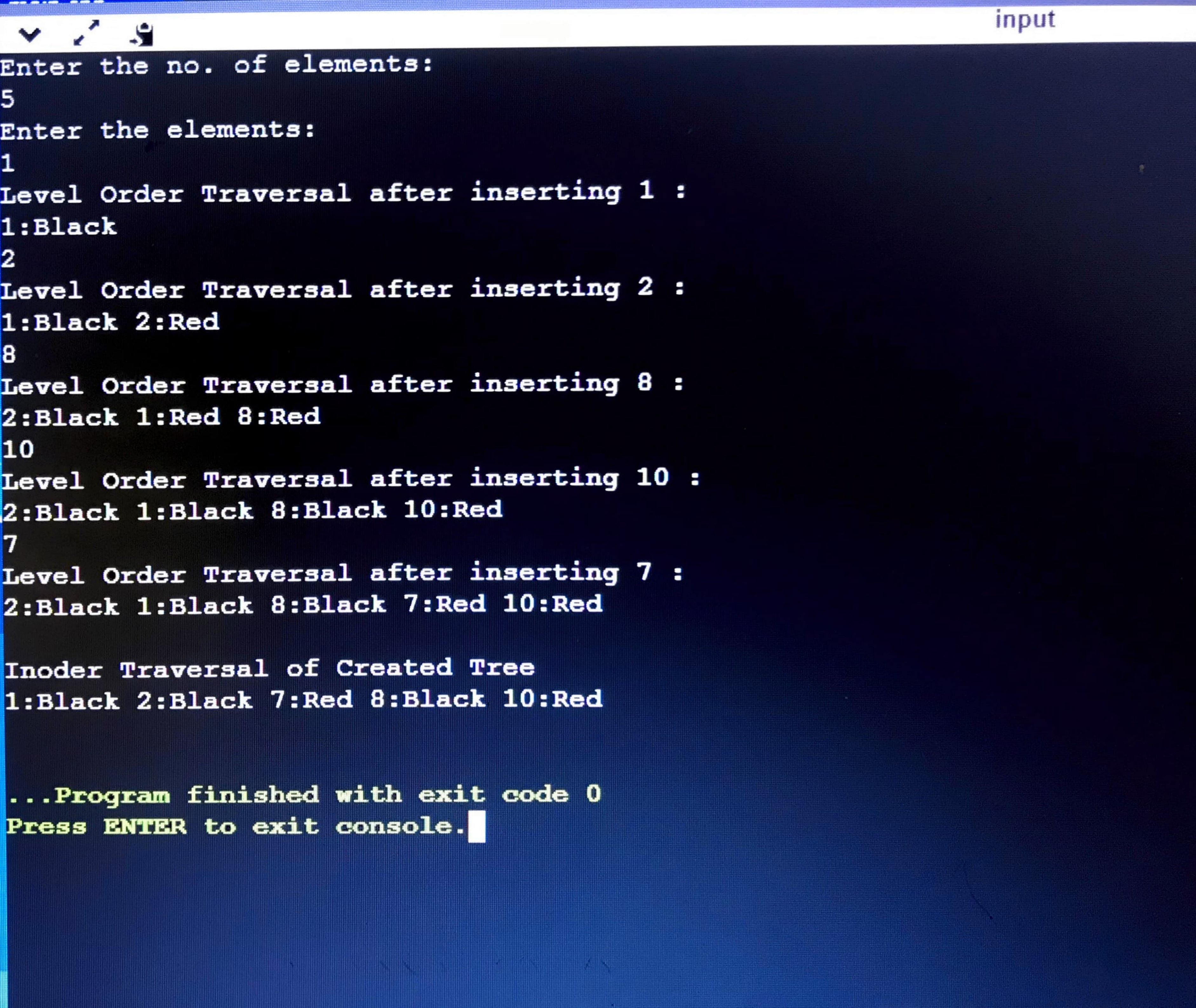
```cpp
    Node *parent_pt = NULL;
    Node *grand_parent_pt = NULL;

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

        parent_pt = pt->parent;
        grand_parent_pt = pt->parent->parent;

        if (parent_pt == grand_parent_pt->left)
        {

                Node *uncle_pt = grand_parent_pt->right;

                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
                {
                    if (pt == parent_pt->right)
                    {
                           rotateLeft(root, parent_pt);
                           pt = parent_pt;
                           parent_pt = pt->parent;
```



```
                    pt = grand_parent_pt;
                }
                else
                {
                    if (pt == parent_pt->left)
                    {
                        rotateRight(root, parent_pt);
                        pt = parent_pt;
                        parent_pt = pt->parent;
                    }
                    rotateLeft(root, grand_parent_pt);
                    swap(parent_pt->color, grand_parent_pt->color);
                    pt = parent_pt;
                }
            }
        }

        root->color = BLACK;
}

void RBTree::insert(int &data)
{
        Node *pt = new Node(data);
        root = BSTInsert(root, pt);
        fixViolation(root, pt);
}

void RBTree::inorder()     {  inorderHelper(root);}
void RBTree::levelOrder()  {  levelOrderHelper(root); }
```



```cpp
}

void RBTree::inorder()      {  inorderHelper(root);}
void RBTree::levelOrder()  {  levelOrderHelper(root); }

int main()
{
    RBTree tree;
    int n, key;
    cout<<"Enter the no. of elements:"<<endl;
    cin>>n;
    cout<<"Enter the elements:"<<endl;
    for(int i=0; i<n; i++)
    {
        cin>>key;
        tree.insert(key);
        cout << "Level Order Traversal after inserting "<<key<<" : "<<endl;
            tree.levelOrder();
    }
    cout<<endl;
    cout << "Inoder Traversal of Created Tree"<<endl;
        tree.inorder();
    cout<<endl;



        return 0;
}

```



```
Enter the no. of elements:
5
Enter the elements:
1
Level Order Traversal after inserting 1 :
1:Black
2
Level Order Traversal after inserting 2 :
1:Black 2:Red
8
Level Order Traversal after inserting 8 :
2:Black 1:Red 8:Red
10
Level Order Traversal after inserting 10 :
2:Black 1:Black 8:Black 10:Red
7
Level Order Traversal after inserting 7 :
2:Black 1:Black 8:Black 7:Red 10:Red

Inoder Traversal of Created Tree
1:Black 2:Black 7:Red 8:Black 10:Red


...Program finished with exit code 0
Press ENTER to exit console.
```