Run | Debug | Stop | Share | Save | { } Beautify

main.cpp

```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();
};
```

main.cpp

```cpp
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())
        {
```

main.cpp

```
94          Node *pt_right = pt->right;
95
96          pt->right = pt_right->left;
97
98          if (pt->right != NULL)
99              pt->right->parent = pt;
100
101         pt_right->parent = pt->parent;
102
103         if (pt->parent == NULL)
104             root = pt_right;
105
106         else if (pt == pt->parent->left)
107             pt->parent->left = pt_right;
108
109         else
110             pt->parent->right = pt_right;
111
112         pt_right->left = pt;
113         pt->parent = pt_right;
114 }
115
116 void RBTree::rotateRight(Node *&root, Node *&pt)
117 {
118         Node *pt_left = pt->left;
119
120         pt->left = pt_left->right;
121
122         if (pt->left != NULL)
123             pt->left->parent = pt;
```

main.cpp

```
143         Node *parent_pt = NULL;
144         Node *grand_parent_pt = NULL;
145
146         while ((pt != root) && (pt->color != BLACK) && (pt->parent->color == RED))
147         {
148
149             parent_pt = pt->parent;
150             grand_parent_pt = pt->parent->parent;
151
152             if (parent_pt == grand_parent_pt->left)
153             {
154
155                 Node *uncle_pt = grand_parent_pt->right;
156
157                 if (uncle_pt != NULL && uncle_pt->color == RED)
158                 {
159                     grand_parent_pt->color = RED;
160                     parent_pt->color = BLACK;
161                     uncle_pt->color = BLACK;
162                     pt = grand_parent_pt;
163                 }
164
165                 else
166                 {
167                     if (pt == parent_pt->right)
168                     {
169                         rotateLeft(root, parent_pt);
170                         pt = parent_pt;
171                         parent_pt = pt->parent;
```

main.cpp

```
189                     pt = grand_parent_pt;
190                 }
191                 else
192                 {
193                     if (pt == parent_pt->left)
194                     {
195                         rotateRight(root, parent_pt);
196                         pt = parent_pt;
197                         parent_pt = pt->parent;
198                     }
199                     rotateLeft(root, grand_parent_pt);
200                     swap(parent_pt->color, grand_parent_pt->color);
201                     pt = parent_pt;
202                 }
203             }
204         }
205
206         root->color = BLACK;
207 }
208
209 void RBTree::insert(int &data)
210 {
211         Node *pt = new Node(data);
212         root = BSTInsert(root, pt);
213         fixViolation(root, pt);
214 }
215
216 void RBTree::inorder()     {  inorderHelper(root);}
217 void RBTree::levelOrder()  {  levelOrderHelper(root); }
218
```

main.cpp

```
214 }
215
216 void RBTree::inorder()     {  inorderHelper(root);}
217 void RBTree::levelOrder()  {  levelOrderHelper(root); }
218
219 int main()
220 {
221     RBTree tree;
222     int n, key;
223     cout<<"Enter the no. of elements:"<<endl;
224     cin>>n;
225     cout<<"Enter the elements:"<<endl;
226     for(int i=0; i<n; i++)
227     {
228         cin>>key;
229         tree.insert(key);
230         cout << "Level Order Traversal after inserting "<<key<<" : "<<endl;
231             tree.levelOrder();
232     }
233     cout<<endl;
234     cout << "Inoder Traversal of Created Tree"<<endl;
235         tree.inorder();
236     cout<<endl;
237
238
239
240         return 0;
241 }
242
```

input

```
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.
```