**PROGRAM 6)** Write a program to implement insertion operation on a red black tree.

CODE:

#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();

};

void inorderHelper(Node \*root)

{

     if (root == NULL)

         return;

     inorderHelper(root->left);

     cout << root->data << ":";

if(root->color == 1)

cout<<"Black ";

else

cout<<"Red ";

     inorderHelper(root->right);

}

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())

     {

         Node \*temp = q.front();

         cout << temp->data << ":";

if(temp->color == 1)

cout<<"Black ";

else

cout<<"Red ";

         q.pop();

         if (temp->left != NULL)

             q.push(temp->left);

         if (temp->right != NULL)

             q.push(temp->right);

     }

cout<<endl;

}

void RBTree::rotateLeft(Node \*&root, Node \*&pt)

{

     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;

     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;

}

void RBTree::fixViolation(Node \*&root, Node \*&pt)

{

     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;

                 }

                 rotateRight(root, grand\_parent\_pt);

                 swap(parent\_pt->color,grand\_parent\_pt->color);

                 pt = parent\_pt;

             }

         }

         else

         {

             Node \*uncle\_pt = grand\_parent\_pt->left;

             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->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); }

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 << "Inorder Traversal of Created Tree"<<endl;

     tree.inorder();

cout<<endl;

     return 0;

}

OUTPUT:

