**Jayesh Kavedia**

**60004190053**

**TE COMPS A**

**Implementation of Advanced Data Structure for Red-Black Tree Insertion Rules**

**Theory**

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions.

Algorithm for Insertion of a Node

Let x be the newly inserted node.

1. Perform standard BST insertion and make the colour of newly inserted node as RED.
2. If x is the root, change the colour of x as BLACK.
3. Do the following if the color of x’s parent is not BLACK and x is not the root.
   1. If x’s uncle is RED
      1. Change the colour of parent and uncle as BLACK.
      2. Colour of a grandparent as RED.
      3. Change x = x’s grandparent, repeat steps 2 and 3 for new x.
   2. If x’s uncle is BLACK, then there can be four configurations for x, x’s parent (p) and x’s grandparent (g)
      1. Left Left Case (p is left child of g and x is left child of p) : Perform Right Rotation and Swap Color
      2. Left Right Case (p is left child of g and x is the right child of p) : Perform RL Rotation and Swap Color
      3. Right Right Case (Mirror of case i) : Perform Left Rotation and Swap Color
      4. Right Left Case (Mirror of case ii) : Perform LR Rotation and Swap Color

**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  
{  
private:  
    Node \*root;  
protected:  
    void rotateLeft(Node \*&, Node \*&);  
    void rotateRight(Node \*&, Node \*&);  
    void fixViolation(Node \*&, Node \*&);  
public:  
    RBTree() { root = NULL; }  
    void insert(const int &n);  
    void inorder();  
    void levelOrder();  
    int blackdepth();  
};  
  
int blackdepthHelper(Node \*root){  
    Node\* ptr = root;  
    int count = 0;  
    while(ptr!=NULL){  
        if (ptr->color ==1){  
            count++;  
        }  
        ptr= ptr->left;  
    }  
    return count;  
}  
   
void inorderHelper(Node \*root)  
{  
    if (root == NULL)  
        return;  
   
    inorderHelper(root->left);  
    cout << root->data << "  "<< root->color << endl;  
    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 << "  "<< temp->color << endl;  
        q.pop();  
   
        if (temp->left != NULL)  
            q.push(temp->left);  
   
        if (temp->right != NULL)  
            q.push(temp->right);  
    }  
}  
   
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(const 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 RBTree::blackdepth()  {  blackdepthHelper(root);  }  
   
int main()  
{  
    RBTree tree;  
   
    tree.insert(8);  
    tree.insert(18);  
    tree.insert(5);  
    tree.insert(15);  
    tree.insert(17);  
    tree.insert(25);  
    tree.insert(40);  
    tree.insert(80);  
      
    cout << "1 -> Black Node 0 -> Red Node\n\n";  
   
    cout << "Inorder Traversal of Created Tree\n";  
    tree.inorder();  
   
    cout << "\n\nLevel Order Traversal of Created Tree\n";  
    tree.levelOrder();  
      
    int bd = tree.blackdepth();  
    cout<< "\n\nBlack depth of RB Tree excluding NULL node " << bd;  
    cout<< "\n\nBlack depth of RB Tree including NULL node " << bd+1;  
   
    return 0;  
}

**Output**

1 -> Black Node 0 -> Red Node  
  
Inorder Traversal of Created Tree  
5  1  
8  0  
15  1  
17  1  
18  1  
25  0  
40  1  
80  0  
  
  
Level Order Traversal of Created Tree  
17  1  
8  0  
25  0  
5  1  
15  1  
18  1  
40  1  
80  0  
  
  
Black depth of RB Tree excluding NULL node 2  
  
Black depth of RB Tree including NULL node 3

**Observations**

1. Insertion follows BST insertion property along with a fixup method to balance the tree in case of any property violation.
2. In case of Red Node followed by Red Node there are three possible cases to be handled.
3. Red-black trees offer logarithmic average and worst-case time complexity for insertion.
4. Rebalancing has an average time complexity of O(1) and worst-case complexity of O(log n).

**Conclusion**

Red Black Tree is a self-balanced tree similar to BST with one extra bit of storage for the color value. While Inserting a node in RB Tree the node must be inserted such that no violation of RB Tree Property takes place. The Average Case Time Complexity for RB Tree Insertion is O (log n).

**References**

1. [https://www.baeldung.com/cs/red-black-trees#:~:text=5.-,Complexity,to%20bulk%20and%20parallel%20operations.](https://www.baeldung.com/cs/red-black-trees%23:~:text=5.-,Complexity,to%20bulk%20and%20parallel%20operations.)
2. <https://www.ijsrd.com/articles/IJSRDV6I90249.pdf>
3. <https://www.cs.umanitoba.ca/~hacamero/Research/RBTreesKim.pdf>