Red-Black Tree

Red-Black tree is a self-balancing binary search tree.
An additional bit is used to denote the colour of node.
Colouring is done to balance the height of tree

Two tools are used for balancing
1 Recoloring
2 Rotation

First priority is given to recoloring, if it doesn't work rotation is employed

Insertion

1 Insert a node similarly as in Binary tree and assign red colour to it.
2 If node is root node, change it to black colour
3 Else, check the colour of parent node.
 1 If parent's node is black, don't change the colour
 2 If parent node is red, check the colour of uncle node
  a. If uncle has red colour, change colour of parent and uncle to black and grandfather to red and repeat the same process

b. Uncle has black colour then there are 4 possible cases.

1. Leftleft case
2. Left Right Case
3. Right Right Case
4. Right Left case

~~Struct~~

```
class Node
{
    int data;
    bool color;
    Node* leftChild, rightChild, parent;
};

class RBTree
{
    Node* root;
    void rotateLeft(Node*, Node*);
    void rotateRight(Node*, Node*);
    void fixViolation(Node*, Node*);
    void insert(const int &n);
    void inorder();
};

Node* BSTInsert(Node* root, Node* pt)
{
    // Function to insert a new node in BST manner
}
```

```
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_righ-> left = pt;
    pt->parent = pt_right;
}

void RBTree :: rotateRight (Node *& root, Node *&pt)
{
    //Similar to rotateLeft, with leftChild pointer
}

void RBTree ::fixViolation (Node*& root, Node *&pt)
{
    Node*parent_pt = NULL;
    Node *grand_parent_pt = NULL;
```

```
while((pt != root) && (pt->color != BLACK) && ~~(pt->p~~
        (pt->parent->colorr == RED))
{
    parent_pt = pt->parent;
    grand_parent_pt = pt->parent->parent;
        // Case A, parent of pt is left child of grandparent of pt
    if (parent_pt == grand_parent_pt->left)
    {
        Node *uncle_pt = grand_parent_pt->right;

        // Case : 1, uncle of pt is red
        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; Left rotation
            if (pt == parent_pt->right)
            {
                rotateLeft (root, parent_pt);
                pt = parent_pt;
                parent_pt = pt->parent;
            }
            // Case 3, pt is left child, right rotation
            rotateRight (root, grand_parent_pt);
            swap (parent_pt-> color, grand_parent_pt->color);
            pt=parent_pt;
        }
    }
    // Case B, parent of pt is right child of
     grand parent of pt
    else
    {
        Node *uncle_pt= grand_parent_pt->left;
        // Case 1
        ~~if(~~
        // Case 2
        // Case 3
      }
    }
  }
  root->color=BLACK;
  } // End of function

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