Red Black Trees

Implementation of Insertion in Red Black Trees

```
struct Node
{   int data;
    bool color; // 0 → Red   1 → black
    Node *left, *right, *parent;

    Node(int data)
    { this → data = data;
      left = right = parent = NULL;
       this → color = 0
    }
};
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);
      void inorder()
};
```

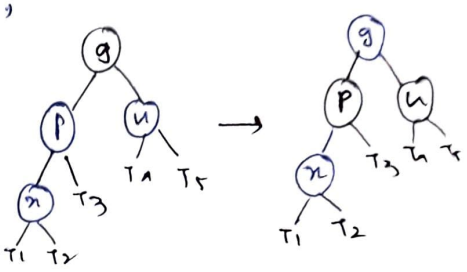
```
Node * BSTInsert (Node * root, Node * pt)
{
    if (root == NULL) return pt;
    if (pt -> data < root -> data)
    {
        root -> left = BSTInsert (root -> left, pt);
        root -> left -> parent = root;
    }
    else if (pt -> data > root -> data)
    {
        root -> right = BSTInsert (root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

void 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;
}
```

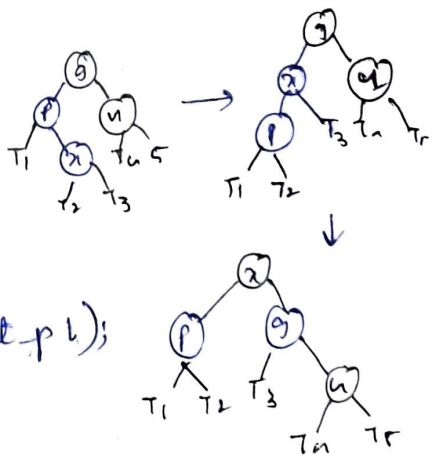
```
void RBTree rotateRight (Node
{
    Node *pt left pt *left
    pt *left - pt left *right
    if (pt *left != NULL)
        pt *left *parent - pt
    pt_left *parent pt *parent
    if (pt *parent == NULL)
        root - pt - left;
    else if ( pt == pt *parent *left)
        pt *parent *left - pt left;
    else
        pt *parent *right - pt - left
    pt_left *right = pt;
    pt *parent - pt-left;
}
```

```
void RBTree::insert (const int data)
{
    Node *pt - new Node (data);
    root - BST Insert (root, pt);
    fix Violation (root, pt);
}
```

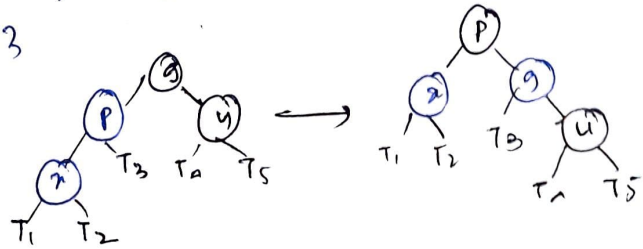
```
void RBTree:: fixViolation(Node *root, Node *pt)
{
    Node *parent_pt = NULL;
    Node *grand_parent_pt = NULL;
    // parent is left child of grandparent
    while ((parent_pt != grand_parent_pt->left)
    {
        Node *uncle_pt = grand_parent_pt->right;
        // uncle red.
        if (uncle_pt != NULL &&
            uncle_pt->color == RED)
        {
            grand_parent_pt->color = 0;
            parent_pt->color = 1;
            uncle_pt->color = 1;
            pt = grand_parent_pt;
        }
        else
        { // pt is right child of parent.
            if (pt == parent_pt->right)
            {
                rotateLeft(root, parent_pt);
                pt = parent_pt;
                parent_pt = pt->parent;
            }
            // pt is left child of parent
            rotateRight(root, grand_parent_pt);
            swap(parent_pt->color,
                 grand_parent_pt->color);
            pt = parent_pt;
        }
    }
```



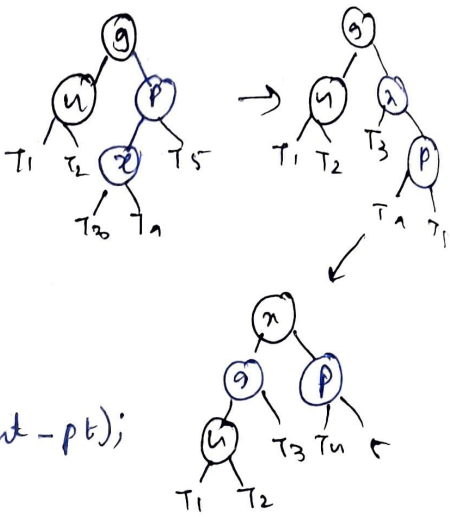
left Right.



left left.



```
else // parent of pt is a left child
     of grand parent of pt
{ Node * uncle_pt = grand_parent_pt->left;
  // uncle of pt is red.
  if((uncle_pt != NULL) &&
     (uncle_pt->color == 0))
  { grand_parent_pt->color = 0;
    parent_pt->color = 1;
    uncle_pt->color = 1;
    pt = grand_parent_pt;
  }
  else
  { // pt -> leftchild of parent
    if (pt == parent_pt->left)
    { rotateRight(root, parent_pt);
      pt = parent_pt;
      parent_pt = pt->parent;
    }
    // pt -> rightchild of parent
    rotateLeft(root, grand_parent_pt);
    Swap(parent_pt->color,
         grand_parent_pt->color);
    pt = parent_pt;
  }
}
root->color = 1;
}
```



Right Right case