PRANAV JAGADEESH IBM18CS071 5B BATCH 4

PROGRAM 6 RED BLACK TREES WRITE-UP

```
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()
};
void inorderHelper(Node *root)
{ if (root <=> NULL)
    return
  inorderHelper(root -> left)
  print root -> data
  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()
    print temp -> data
    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->right
  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 main()
{ RBTree tree
  int n <- 0, m <- 0
  print nodes to be inserted
  n <- value
  print Enter nodes
  for (int i <- 0, i<n, i++)
    { m <- value
      tree.insert(m)
    }
  print Inorder Traversal
      tree.inorder()
  print Level Order Traversal
      tree.levelOrder()
      return 0
}
```