Hemanth Kumar V.C
1BM18CS037
Sec 'A'
Date: 18/11/2020

ADS LAB

1) Red black tree

```
class RBTree
{
    private:
        Node *root;
        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);
    cout << root->data << " ";
    inorderHelper(root->right);
}
Node* BST insert(Node *root, Node *pt)
{
    if (root == NULL)
        return pt;
```

Scanned with CamScanner

```
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 levelorder Helper ( node *root)
{
    if ( root == NULL)
       return;
    std::queue <Node *> q;
      q.push(root);
    while( !q.empty())
    {
      node *temp = q.front();
       cout << temp->data << " ";
         q.pop();
      if (temp->left != NULL)
         q.push(temp->left);
      if ( temp->right != NULL)
         q.push(temp->right);
    }
}
```

Scanned with CamScanner

IBM18CS037
Sec A

```
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;
    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;
```

Scanned with CamScanner

```
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->~~
                (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->left)
```

Scanned with CamScanner

```
{
    rotate Right(root, parent_pt);
        pt = parent_pt;
        parent_pt = pt->parent;
    }
    rotate left(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 = BST insert(root, pt);
    fix violation(root, pt);
}
void RBTree::inorder()
{
    inorder Helper(root);
}
void RBTree::levelorder()
{
    levelorder Helper(root);
}
```

Scanned with CamScanner