```
//====================================================================
========
// Name        : DASL_Assignment 1 .cpp
// Author      : Shrikrushna S Zirape
// Version     :
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//====================================================================
========

#include<iostream>

#define max 20
using namespace std;

class Node{
     int data;
     Node *lchild, *rchild;

public:
     Node(){
          lchild = NULL;
          rchild = NULL;
          data = 0;
     }
     Node(int k){
          lchild = NULL;
          rchild = NULL;
          data = k;
     }

     friend class Queue;
     friend class Stack;
     friend class BT;
};

class Queue{
     int front,rear;
    int count;
     Node *item[max];

public:
     Queue(){
          front = -1;
          rear = -1;
         count = 0;

     }

     bool isEmpty();
     bool isFull();
     void push(Node *ptr);
     Node* pop();
     void printQueue();
     int sizeOfQueue();

};

bool Queue::isEmpty(){
     if(front == -1 || front > rear){
          return true;
     }
     else return false;
}

bool Queue::isFull(){
     if(rear == max -1){
          return true;
     }
     else return false;
}

void Queue::push(Node *ptr){
     if(this->isFull()){
          cout<<"Queue is Full \n";
          return;
     }
     else if(front == -1){
          front++;
         count ++;
          rear ++;
          item[rear] = ptr;
     }
     else{
          rear ++;
         count ++;
```

```
          item[rear]=ptr;
     }
}

Node* Queue::pop(){

     if(isEmpty()){
          cout<<"Nothing to show";
          return NULL;
     }
     Node *temp = new Node();
     temp = item[front];
     front ++;
    count --;

     return temp;
}

void Queue::printQueue(){
     if(isEmpty()){
          cout<<"Empty!! nothing to show\n";
          return;
     }
     for(int i=front; i<=rear; i++){
          cout<<item[i]->data<<" ";
     }

}

int Queue::sizeOfQueue(){
     return count;
}

class Stack{
     int top;
     Node* items[max];
public:
     Stack(){
          top = -1;
     }
     bool isEmpty();
     void push(Node * ptr);
     Node *pop();
```

```
};

bool Stack::isEmpty(){
     return (top < 0);
}

void Stack:: push(Node* ptr){
     if(top >=(max-1)){
          cout<<"Stack Overflow";
     }
     items[++top]=ptr;

}

Node* Stack::pop(){
     if(isEmpty()){
          return NULL;
     }

     return items[top--];

}

class BT{
    Node * root;

    public:
    BT(){
        root = NULL;
    }
     BT(Node *ptr){
         root = ptr;
     }
    Node * getRoot();
    void createTreeNonRecursivly();
     void createTreeRecursivly();
    Node *createNodeRecursivly();
     void inorderTraversalRecursivly(Node *);
     void preorderTraversalRecursivly(Node *);
     void postorderTraversalRecursivly(Node *);
     void inorderTraversalIterativly();
     void preorderTraversalIterativly();
     void postorderTraversalIterativly();
    int heightOfBinaryTreeRecursivly(Node *);
    int heightOfBinaryTreeIterativly();
    void mirrorTreeRecursivly(Node *ptr);
    void mirrorTreeIterativly();
     void operator = (BT & t1);
    Node *copyTreeRecursivly(Node *ptr);
    int leafNodeRecursivly(Node *ptr);
    int leafNodeIterativly();
    int internalCountRecursivly(Node *ptr);
    int internalCountIterativly();
     void printLevelRecursive(Node *ptr, int level);
     void printLevelwiseRecursive();
     void printLevelwiseIterative();
     void deleteTreeRecursivly(Node *ptr);
     void deleteTreeIterativly();




};

Node* BT::getRoot(){
    return root;
}

void BT::createTreeRecursivly(){
     root = createNodeRecursivly();
}

void BT::createTreeNonRecursivly(){
     Queue q;
     cout<<"Enter the Root Data :- ";
     int n;
     cin>>n;
     Node *t = new Node(n);
     if(root ==NULL){
          root =t;
          q.push(t);
     }
     while(!q.isEmpty()){
          Node* t2 = q.pop();
          cout<<"press 1 if has left node else 0 ";
          int k1;
```

```
        cin>>k1;
        if(k1 ==1){
            cout<<"Enter the left data :- ";
            int ldata;
            cin>>ldata;
            Node *left = new Node(ldata);
            t2->lchild = left;
            q.push(t2->lchild);
        }
        cout<<"press 1 if has right node else 0 ";
        int k2;
        cin>>k2;
        if(k2 ==1){
            cout<<"Enter the right data :- ";
            int rdata;
            cin>>rdata;
            Node *right = new Node(rdata);
            t2->rchild = right;
            q.push(t2->rchild);
        }
    }
}

Node * BT::createNodeRecursivly(){
    int data;
    cout<<"Enter the Data else enter the -1 ";
    cin>>data;
    if(data == -1){
        return NULL;
    }
    Node *t = new Node(data);
    cout<<"Enter the left child data of "<<t->data <<" ";
    t->lchild=createNodeRecursivly();
    cout<<"Enter the right child data of "<<t->data <<" ";
    t->rchild=createNodeRecursivly();
    return t;
}

void BT::inorderTraversalRecursivly(Node* t){
    if(t!=NULL){
        inorderTraversalRecursivly(t->lchild);
        cout<<t->data << " -> ";
        inorderTraversalRecursivly(t->rchild);
    }
}

void BT::preorderTraversalRecursivly(Node *t){
    if(t!=NULL){
        cout<<t->data<< " -> ";
        preorderTraversalRecursivly(t->lchild);
        preorderTraversalRecursivly(t->rchild);
    }
}

void BT::postorderTraversalRecursivly(Node *t){
    if(t!=NULL){
        postorderTraversalRecursivly(t->lchild);
        postorderTraversalRecursivly(t->rchild);
        cout<<t->data<<" -> ";
    }
}

void BT::inorderTraversalIterativly(){
    Stack s;
    Node *curr = root;
    while(curr !=NULL || s.isEmpty() == false){
        while(curr!=NULL){
            s.push(curr);
            curr = curr->lchild;
        }
        curr = s.pop();
        cout<<curr->data<<" -> ";
        curr = curr->rchild;
    }
}

void BT::preorderTraversalIterativly(){
    if(root == NULL){
        return;
    }
    Stack s;
    s.push(root);
    while(!s.isEmpty()){
        Node *node = s.pop();
        cout<<node->data<<" -> ";
        if(node->rchild){
            s.push(node->rchild);
        }
        if(node->lchild){
            s.push(node->lchild);
        }
    }
}

void BT::postorderTraversalIterativly(){
    if(root == NULL){
        return;
    }
    else{
        Stack s1, s2;
        s1.push(root);
        Node *node;
        while(!s1.isEmpty()){
            node=s1.pop();
            s2.push(node);

            if(node->lchild){
                s1.push(node->lchild);
            }
            if(node->rchild){
                s1.push(node->rchild);
            }
        }
        while(!s2.isEmpty()){
            node = s2.pop();
            cout<<node->data<<" -> ";
        }

    }
}

int BT::heightOfBinaryTreeRecursivly(Node *ptr){
    if(ptr == NULL) return 0;
    int right, left;
    right = heightOfBinaryTreeRecursivly(ptr->rchild);
    left = heightOfBinaryTreeRecursivly(ptr->lchild);
    if(right > left) return (right +1);
    return (left +1);
```

```
}

int BT::heightOfBinaryTreeIterativly(){
    if(root == NULL) return 0;
     int height = 0;
     Queue q;
     q.push(root);
     while(true){
          int noOfNodes = q.sizeOfQueue();
          if(noOfNodes ==0){
               return height;
          }
          height ++;
          while(noOfNodes > 0){
          Node *node = q.pop();
          if(node ->lchild != NULL)
         {q.push(node->lchild);}
          if(node->rchild !=NULL)
         {q.push(node->rchild);}
          noOfNodes--;
          }
     }
}

void BT::mirrorTreeRecursivly(Node *ptr){
     if(ptr->lchild == NULL || ptr->rchild == NULL){
          return;
     }
     Node *t = ptr->lchild;
     ptr->lchild = ptr->rchild;
     ptr->rchild = t;

     if (ptr->lchild){
          mirrorTreeRecursivly(ptr->lchild);
     }
     if(ptr->rchild){
          mirrorTreeRecursivly(ptr->rchild);
     }
}

void BT::mirrorTreeIterativly(){
    if(root == NULL){
        return;
    }
    Queue q;
    q.push(root);
    while( ! q.isEmpty()){
        Node *node = q.pop();
        swap(node->lchild, node->rchild);
        if (node ->lchild){
            q.push(node->lchild);
        }
        if(node ->rchild){
            q.push(node->rchild);
        }
    }

}

void BT::operator = (BT &t1){
     root = copyTreeRecursivly(t1.root);
}

Node * BT::copyTreeRecursivly(Node* ptr){
     Node *copyNode = NULL;
     if(ptr){
         copyNode = new Node(ptr->data);
         copyNode->lchild = copyTreeRecursivly(ptr->lchild);
         copyNode->rchild = copyTreeRecursivly(ptr->rchild);
     }
     return copyNode;
}

int BT::leafNodeRecursivly(Node *ptr){
    if(ptr == NULL){
        return  0;
    }
    if(ptr->lchild == NULL && ptr->rchild == NULL){
        return 1;
    }
    else{
        return (leafNodeRecursivly(ptr->lchild) +
leafNodeRecursivly(ptr->rchild));
    }
}
```

```
int BT::leafNodeIterativly(){
    if(root == NULL){
        return 0;
    }
    Queue q;
    int count = 0;
    q.push(root);
    while(!q.isEmpty()){
        Node *node = q.pop();
        if(node->lchild != NULL){
            q.push(node->lchild);
        }
        if(node->rchild != NULL){
            q.push(node->rchild);
        }
        if(node->lchild == NULL && node->rchild == NULL){
            count ++;
        }
    }
        return count;


}

int BT::internalCountIterativly(){
    if(root == NULL){
        return 0;
    }
   Queue q;
    int count = 0;
    q.push(root);
    while (!q.isEmpty())
    {
        struct Node *temp = q.pop();

        if (temp->lchild && temp->rchild)
            count++;

        if (temp->lchild != NULL)
            q.push(temp->lchild);
        if (temp->rchild != NULL)
            q.push(temp->rchild);
    }
    return count;
}

int BT::internalCountRecursivly(Node *ptr){
    if (ptr == NULL)
       return 0;

    int res = 0;
    if  (root->lchild && root->rchild)
       res++;

    res += (internalCountRecursivly(root->lchild) +
            internalCountRecursivly(root->rchild));
    return res;
}

void BT::printLevelwiseRecursive(){
     int h = heightOfBinaryTreeRecursivly(root);
     cout<<"\n Printitng tree levelwise :- \n";
     for(int i=1; i<=h; i++){
          printLevelRecursive(root, i);
     }
}

void BT::printLevelRecursive(Node *ptr, int level){
     if(ptr == NULL){
          return;
     }
     if (level == 1){
          cout<<ptr->data<<" -> ";
     }
     else if(level >1){
          printLevelRecursive(ptr->lchild, level -1);
          printLevelRecursive(ptr->rchild, level -1);
     }
}

void BT::printLevelwiseIterative(){
     cout<<"\nPrinting Levelwise Iterative :- \n";
     if(root == NULL){
          return;
     }
     Queue q;
    q.push(root);
    while(!q.isEmpty()){
        Node *node = q.pop();
        cout<<node->data<<" -> ";
        if(node->lchild != NULL){
            q.push(node->lchild);
        }
        if(node->rchild != NULL){
            q.push(node->rchild);
        }
    }
}

void BT::deleteTreeRecursivly(Node *ptr){
    if(ptr == NULL){
        return;
    }
    deleteTreeRecursivly(ptr->lchild);
    deleteTreeRecursivly(ptr->rchild);
    delete ptr;
    ptr = NULL;
}

void BT::deleteTreeIterativly(){
    if(root == NULL){
        return;
    }
    Queue q;
    q.push(root);
    while(!q.isEmpty()){
        Node *front;
        front = q.pop();
        if(front->lchild){
            q.push(front->lchild);
        }
        if(front->rchild){
            q.push(front->rchild);
        }
        delete front;
    }
    root = NULL;
}
```

```
int mainMenu(){
     int ch;
     cout<<"\n-----------------------Menu------------------------\n";
     cout<<"\n1.    Recursive Operation";
     cout<<"\n2.    Iterative Operation";
     cout<<"\n0.    End the Prgoram";
     cout<<"\nEnter the Correct Option:- ";
     cin>>ch;
     cout<<"\n----------------------------------------------------\n";
     return ch;
}

int RecMenu(){
     int k;
     cout<<"\n ------------------Recursive Menu-----------------";
     cout<<"\n1.   Inorder Traversal";
     cout<<"\n2.   Preorder Traversal";
     cout<<"\n3.   Postorder Traversal";
     cout<<"\n4.   Mirror The Tree";
     cout<<"\n5.   Height of the Tree";
     cout<<"\n6.   Copy the Tree";
     cout<<"\n7.   No of Nodes in the tree";
     cout<<"\n8.   Erase all nodes in tree";
     cout<<"\n0.   Return to Main Menu";
     cout<<"\n Enter the Correct Option :- ";
     cin>>k;
     cout<<"\n---------------------------------------------------\n";
     return k;
}

int NonRecMenu(){
     int l;
     cout<<"\n ------------------Iterative Menu-----------------";
     cout<<"\n1.   Inorder Traversal";
     cout<<"\n2.   Preorder Traversal";
     cout<<"\n3.   Postorder Traversal";
     cout<<"\n4.   Mirror The Tree";
     cout<<"\n5.   Height of the Tree";
     cout<<"\n6.   Copy the Tree";
     cout<<"\n7.   No of Nodes in the tree";
     cout<<"\n8.   Erase all nodes in tree";
     cout<<"\n0.   Return to Main Menu";
     cout<<"\n Enter the Correct Option :- ";
     cin>>l;
     cout<<"\n----------------------------------------------------\n";
     return l;
}

int main(){
    BT b,c,d,e;
     int ch;
     do{
          ch = mainMenu();
          switch (ch)
          {
          case 1:
          {


               int cr;
               b.createTreeRecursivly();
               Node *ptr1;
               ptr1= b.getRoot();
               do{
               cr = RecMenu();
               switch (cr)
               {
               case 1:
                    cout<<"\nInorder Traversal:- ";
                    b.inorderTraversalRecursivly(ptr1);
                    break;
               case 2:
                    cout<<"\nPreorder Traversal:- ";
                    b.preorderTraversalRecursivly(ptr1);
                    break;
               case 3:
                    cout<<"\nPostorder Traversal:- ";
                    b.postorderTraversalRecursivly(ptr1);
                    break;
               case 4:
                    cout<<"\nMirroring the Tree :-";
                    b.mirrorTreeRecursivly(ptr1);
                    break;
               case 5:
                    cout<<"\n Height of the tree :- ";
                    cout<<b.heightOfBinaryTreeRecursivly(ptr1);
```

```
                break;
            case 6:
                cout<<"\n Copying the tree ";
                d=b;
                cout<<"\nFirst Tree :- ";
                b.inorderTraversalIterativly();
                cout<<"\nSecond Tree :- ";
                d.inorderTraversalIterativly();
                break;
            case 7:
                cout<<"\n Node count:- ";
                int full, internal, leaf;
                internal = b.internalCountRecursivly(ptr1);
                leaf = b.leafNodeRecursivly(ptr1);
                full = internal+leaf;
                cout<<"\n Total Nodes :- "<<full;
                cout<<"\n Internal Nodes :- "<<internal;
                cout<<"\n Leaf Nodes :- "<<leaf;
                break;
            case 8:
                cout<<"\n Deleting the tree";
                b.deleteTreeRecursivly(ptr1);
                break;
            case 0:
                cout<<"\n Returning to the main menu";
                break;
            default:
                cout<<"\n Wrong Option Selected";
                break;

            }
            }while(cr!=0);
            break;
        }
        case 2:
        {
            int cnr;
            c.createTreeNonRecursivly();
            Node *ptr2 = c.getRoot();
            do{
            cnr = NonRecMenu();
            switch (cnr)
            {
```

```
case 1:
    cout<<"\n Inorder Traversal";
    c.inorderTraversalIterativly();
    break;
case 2:
    cout<<"\n Preorder Traversal";
    c.preorderTraversalIterativly();
    break;
case 3:
    cout<<"\n Postorder Traversal";
    c.postorderTraversalIterativly();
    break;
case 4:
    cout<<"\nmirroring the Tree:- ";
    c.mirrorTreeIterativly();
    break;
case 5:
    cout<<"\n Height of the tree:-";
    int k;
    k= c.heightOfBinaryTreeIterativly();
    cout<<k;
    break;
case 6:
    cout<<"\n Copying the tree";
    e=c;
    cout<<"\nFirst Tree :- ";
    c.inorderTraversalIterativly();
    cout<<"\nSecond Tree :- ";
    e.inorderTraversalIterativly();
    break;
case 7:
    cout<<"\n Node Count";
    int x,y,z;
    x=c.internalCountIterativly();
    y=c.leafNodeIterativly();
    z=x +y;
    cout<<"\nTotal :- "<<z;
    cout<<"\ninearnal  :- "<<x;
    cout<<"\nleaf :- "<<y;
    break;
case 8:
    c.deleteTreeIterativly();
    cout<<"\nDeleted the Tree";
```

```
                break;
            case 0:
                cout<<"\nReturning to the main menu";
                break;
            default:
                cout<<"\nincorrect Option";
                break;

            }
            }while(cnr!=0);
            break;
        }
        case 0:{
            break;
        }
        default:
        {
            cout<<"\nWrong Option ";
            break;
        }
        }
    }while(ch!=0);
    return 0;
}
```