DSAL : ASSIGNMENT-1

Roll No .: 21441

Code:

```
//======================================================================
========
// Name        : DSAL_assn1_21334.cpp
// Author      : Shantanu Patankar
// Version     :
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//======================================================================
========

#include <iostream>

using namespace std;

class Node
{
private:
      int data;
      Node* lchild;
      Node* rchild;

public:
      Node()
      {
            data = 0;
            lchild = NULL;
            rchild = NULL;
      }
      Node(int n)
      {
            data = n;
            lchild = NULL;
            rchild = NULL;
      }
      friend class Binarytree;
};

class Queue
{
```

```
private:
    Node* a[100];
    int n = 100,front = -1,rear = -1;

public:
    void enqueue(Node* b);
    Node* dequeue();
    bool empty();
    int size();
};

void Queue::enqueue(Node* b)
    {
        if(rear == n-1)
        {
            cout<<"Queue Overflow!"<<endl;
        }
        else
        {
            if(front == -1)
            {
                front = 0;
                rear = 0;
            }
            rear++;
            a[rear] = b;
        }
    }

Node* Queue::dequeue()
    {
        if(front == -1 || front>rear)
        {
            cout<<"Queue Underflow!"<<endl;
            return 0;
        }
        else
        {
            front++;
            return(a[front]);
        }
    }

bool Queue::empty()
    {
        if(rear-front == 0)
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }

int Queue::size()
{
    return(rear-front);
}

class Stack
{
private:
    Node* stack[100];int n=100,top=-1;
public:
    void push(Node*);
    Node* pop();
    bool isempty();
};

void Stack::push(Node* temp)
{
    if(top>=(n-1))
    {
        cout<<"Stack Overflow!"<<endl;
    }
    else
    {
        top++;
        stack[top] = temp;
    }
}

Node* Stack::pop()
{
    if(top<=-1)
    {
        cout<<"Stack Empty!"<<endl;
        return NULL;
    }
    else
    {
            top--;
            return(stack[top+1]);
      }
}

bool Stack::isempty()
{
      if(top==-1)
            return 1;
      else
            return 0;
}

class Binarytree
{
private:
      Node* root;
      int total_nodes = 0;

public:
      Binarytree()
      {
            root = NULL;
      }

      void create_nonrec();
      Node* getroot(Node*);
      void setroot(Node*);
      Node* create_rec();
      void inorder_nonrec();
      void inorder_rec(Node*);
      void preorder_nonrec();
      void preorder_rec(Node*);
      void postorder_nonrec();
      void postorder_rec(Node*);
      void levelwisetraver();               //bfs
      void shiftrole(Node*);
      int max(int,int);
      int height_rec(Node*);
      int height_nonrec();
      void operator=(Binarytree& );
      Node* copytree(Node*);
      void delallnode(Node*);
      int leavescount(Node*);
      int totalnodes(Node*);
```

```
};

void Binarytree::create_nonrec()
    {
        Queue q;
        if(root == NULL)
        {
            int x;
            cout<<"Enter value to be added at root: ";cin>>x;
            //root = new Node(x);
            root = new Node(x);
            q.enqueue(root);
        }

            while(!q.empty())
            {
                Node* parent = q.dequeue();
                cout<<"Current node data: "<<parent->data<<endl;
                char l,r;
                cout<<"Does the node have left child?
(Y/N)";cin>>l;
                if(l == 'Y' || l == 'y')
                {
                    int ld;
                    cout<<"Enter value for left child:
";cin>>ld;
                    Node* newnode1 = new Node(ld);
                    parent->lchild = newnode1;
                    q.enqueue(newnode1);
                }
                cout<<"Does the node have right child?
(Y/N)";cin>>r;
                if(r == 'Y' || r == 'y')
                {
                    int rd;
                    cout<<"Enter value for right child:
";cin>>rd;
                    Node* newnode2 = new Node(rd);
                    parent->rchild = newnode2;
                    q.enqueue(newnode2);
                }
            }

        return;
    }
```

```
Node* Binarytree::getroot(Node* temp)
{
    temp = root;
    return temp;
}

void Binarytree::setroot(Node* temp)
{
    root = temp;
    return;
}

Node* Binarytree::create_rec()
{
    int x;
    cout<<"Enter data or -1 for no data: ";cin>>x;
    if(x==-1)
    {
        return NULL;
    }
    Node* p = new Node(x);
    cout<<"Current data is "<<x<<endl;
    cout<<"Enter the left child data"<<endl;
    p->lchild = create_rec();
    cout<<"Current data is "<<x<<endl;
    cout<<"Enter the right child data"<<endl;
    p->rchild = create_rec();
    return p;
}

void Binarytree::inorder_nonrec()
{
    Stack s;
    Node* current = root;
    do
    {
        while(current!=NULL)
        {
            s.push(current);
            current = current->lchild;
        }
        Node* temp = s.pop();
        cout<<temp->data<<" ";
        current = temp->rchild;
    }
    while(current!=NULL || !s.isempty());
    cout<<endl;
}

void Binarytree::inorder_rec(Node* t)
{
    if(t==NULL)
        return;
    {
        inorder_rec(t->lchild);
        cout<<t->data<<" ";
        inorder_rec(t->rchild);
    }


}

void Binarytree::preorder_nonrec()
{
    Stack s;
    Node* current = root;
    s.push(current);
    do
    {
        Node* temp = s.pop();
        cout<<temp->data<<" ";
        if(temp->rchild != NULL)
            s.push(temp->rchild);
        if(temp->lchild != NULL)
            s.push(temp->lchild);
    }
    while(!s.isempty());

    cout<<endl;
}

void Binarytree::preorder_rec(Node* t)
{
    if(t==NULL)
        return;
    {
        cout<<t->data<<" ";
        preorder_rec(t->lchild);
        preorder_rec(t->rchild);
    }
}
```

```
void Binarytree::postorder_nonrec()
{
    Stack s1,s2;
    s1.push(root);
    while(!s1.isempty())
    {
        Node* temp = s1.pop();
        s2.push(temp);
        if(temp->lchild != NULL)
            s1.push(temp->lchild);
        if(temp->rchild != NULL)
            s1.push(temp->rchild);
    }
    while(!s2.isempty())
    {
        Node* t = s2.pop();
        cout<<t->data<<" ";
    }
    cout<<endl;
}

void Binarytree::postorder_rec(Node* t)
{
    if(t==NULL)
        return;
    {
        postorder_rec(t->lchild);
        postorder_rec(t->rchild);
        cout<<t->data<<" ";
    }
}

void Binarytree::levelwisetraver()          // bfs
{
    Queue q1;
    if(!root)
        return;
    q1.enqueue(root);
    while(!q1.empty())
    {
        Node* temp = q1.dequeue();
        cout<<temp->data<<" ";
        if(temp->lchild)
            q1.enqueue(temp->lchild);
        if(temp->rchild)
```

```
                  q1.enqueue(temp->rchild);
      }
      cout<<endl;
}

void Binarytree::shiftrole(Node* t)
{
      if(t!=NULL)
      {
      if(t->lchild==NULL && t->rchild==NULL)
      {
            // no switching
      }
      else if(t->rchild==NULL)
      {
            t->rchild = t->lchild;
            t->lchild = NULL;
      }
      else if(t->lchild==NULL)
      {
            t->lchild = t->rchild;
            t->rchild = NULL;
      }
      else
      {
            Node* temp = t->rchild;
            t->rchild = t->lchild;
            t->lchild = temp;
      }
      shiftrole(t->lchild);
      shiftrole(t->rchild);
      }
}

int Binarytree::max(int a,int b)
{
      if(a>=b)
      {
            return a;
      }
      else
      {
            return b;
      }
}
```

```
int Binarytree::height_rec(Node* temp)
{
    if(temp==NULL)
    {
        return -1;
    }
    else
    {
        return(1 + max(height_rec(temp->lchild),height_rec(temp-
>rchild)));
    }
}

int Binarytree::height_nonrec()
{
    Queue q1;
    q1.enqueue(root);
    int h = -1;
    while(!q1.empty())
    {
        int c = q1.size();
        if(c==0)
        {
            return h;
        }
        else
        {
            h++;
        }
        while(c>0)
        {
            Node* temp = q1.dequeue();
            if(temp->lchild != NULL)
                q1.enqueue(temp->lchild);
            if(temp->rchild != NULL)
                q1.enqueue(temp->rchild);
            c--;
        }
    }
    return h;
}

void Binarytree::operator =(Binarytree &t1)
{
    root = copytree(t1.root);
}
```

```
Node* Binarytree::copytree(Node* rt)
{
    Node* temp = NULL;
    if(rt)
    {
        temp = new Node(rt->data);
        temp->lchild = copytree(rt->lchild);
        temp->rchild = copytree(rt->rchild);
    }
    return temp;
}

void Binarytree::delallnode(Node* t)
{
    if(t!=NULL)
    {
        delallnode(t->lchild);
        delallnode(t->rchild);
        delete(t);
    }
}

int Binarytree::totalnodes(Node* t)
{
    if(t != NULL)
    {
        totalnodes(t->lchild);
        total_nodes++;
        totalnodes(t->rchild);
    }
    return(total_nodes);
}

int Binarytree::leavescount(Node* t)
{
    if(t==NULL)
    {
        return 0;
    }
    else if(t->lchild==NULL && t->rchild==NULL)
    {
        return 1;
    }
    else
    {
        return(leavescount(t->lchild)+leavescount(t->rchild));
    }
}

int type()
{
    int x;
    cout<<"Enter 0 to use recursive method and 1 to use non-recursive
method :";cin>>x;
    return x;
}

int main() {
    Binarytree obj1;
    while(true){
    cout<<"Enter 0 to end program"<<endl;
    cout<<"Enter 1 to create tree"<<endl;
    cout<<"Enter 2 for inorder traversal"<<endl;
    cout<<"Enter 3 for reorder traversal"<<endl;
    cout<<"Enter 4 for postorder traversal"<<endl;
    cout<<"Enter 5 for level wise traversal"<<endl;
    cout<<"Enter 6 for finding height"<<endl;
    cout<<"Enter 7 for finding mirror image"<<endl;
    cout<<"Enter 8 for copying the tree"<<endl;
    cout<<"Enter 9 for finding leaves"<<endl;
    cout<<"Enter 10 to find internal nodes"<<endl;
    cout<<"Enter 11 for deleting all nodes"<<endl;

    int c;cout<<"Enter your choice: ";cin>>c;
    if(c==0)
    {
        cout<<"Thank you!"<<endl;
        break;
    }
    else if(c==1)
    {
        int x = type();
        if(x==0)
            obj1.setroot(obj1.create_rec());
        else if(x==1)
            obj1.create_nonrec();
        else
            cout<<"Invalid option"<<endl;
    }
    else if(c==2)
    {
```

```
    int x = type();
    if(x==0)
    {
        Node* temp;
        obj1.inorder_rec(obj1.getroot(temp));
        cout<<endl;
    }
    else if(x==1)
        obj1.inorder_nonrec();
    else
        cout<<"Invalid option"<<endl;
}
else if(c==3)
{
    int x = type();
    if(x==0)
    {
        Node* temp;
        obj1.preorder_rec(obj1.getroot(temp));cout<<endl;
    }
    else if(x==1)
        obj1.preorder_nonrec();
    else
        cout<<"Invalid option"<<endl;
}
else if(c==4)
{
    int x = type();
    if(x==0)
    {
        Node* temp;
        obj1.postorder_rec(obj1.getroot(temp));
        cout<<endl;
    }
    else if(x==1)
        obj1.postorder_nonrec();
    else
        cout<<"Invalid option"<<endl;
}
else if(c==5)
{
    obj1.levelwisetraver();
}
else if(c==6)
{
    int x = type();
```

```
        if(x==0)
        {
            Node* temp;
            cout<<"Height is:
"<<obj1.height_rec(obj1.getroot(temp))<<endl;
        }
        else if(x==1)
            cout<<"Height is "<<obj1.height_nonrec()<<endl;
        else
            cout<<"Invalid option"<<endl;
    }
    else if(c==7)
    {
        obj1.levelwisetraver();
        Node* temp;
        obj1.shiftrole(obj1.getroot(temp));
        obj1.levelwisetraver();
    }
    else if(c==8)
    {
        Binarytree obj2;
        obj2 = obj1;
        obj2.levelwisetraver();
    }
    else if(c==9)
    {
        Node* temp;
        cout<<"Number of leaves are:
"<<obj1.leavescount(obj1.getroot(temp))<<endl;
    }
    else if(c==10)
    {
        Node* temp;
        cout<<"Internal nodes are:
"<<obj1.totalnodes(obj1.getroot(temp))-
obj1.leavescount(obj1.getroot(temp))<<endl;
    }
    else if(c==11)
        {
            Node* temp;
            obj1.delallnode(obj1.getroot(temp));
            cout<<"All nodes deleted successfully!"<<endl;
        }
    else
    {
        cout<<"Enter valid choice!"<<endl;
    }
    }
    return 0;
}
```

Output:

```
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 1
Enter 0 to use recursive method and 1 to use non-recursive method :1
Enter value to be added at root: 10
Current node data: 10
Does the node have left child? (Y/N)y
Enter value for left child: 20
Does the node have right child? (Y/N)y
Enter value for right child: 30
Current node data: 20
Does the node have left child? (Y/N)y
Enter value for left child: 40
Does the node have right child? (Y/N)y
Enter value for right child: 50
Current node data: 30
Does the node have left child? (Y/N)n
Does the node have right child? (Y/N)y
Enter value for right child: 60
Current node data: 40
Does the node have left child? (Y/N)n
Does the node have right child? (Y/N)n
Current node data: 50
Does the node have left child? (Y/N)n
Does the node have right child? (Y/N)n
Current node data: 60
Does the node have left child? (Y/N)n
Does the node have right child? (Y/N)n
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 2
Enter 0 to use recursive method and 1 to use non-recursive method :1
40 20 50 10 30 60
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 3
Enter 0 to use recursive method and 1 to use non-recursive method :1
10 20 40 50 30 60
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 4
Enter 0 to use recursive method and 1 to use non-recursive method :0
40 50 20 60 30 10
```

```
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 5
10 20 30 40 50 60
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 6
Enter 0 to use recursive method and 1 to use non-recursive method :1
Height is 2
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 7
10 20 30 40 50 60
10 30 20 60 50 40
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 8
10 30 20 60 50 40
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 9
Number of leaves are: 3
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 10
Internal nodes are: 3
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 11
All nodes deleted successfully!
Enter 0 to end program
Enter 1 to create tree
Enter 2 for inorder traversal
Enter 3 for reorder traversal
Enter 4 for postorder traversal
Enter 5 for level wise traversal
Enter 6 for finding height
Enter 7 for finding mirror image
Enter 8 for copying the tree
Enter 9 for finding leaves
Enter 10 to find internal nodes
Enter 11 for deleting all nodes
Enter your choice: 0
Thank you!
```