/\* ======================================================================

Assignment NO. 05

Title **: Implement bianry Search tree As ADT and Perform different**

**operation**.

Author **: MOMIN SOHAIL**

Roll no : **2074**

Class : SE

Division : I10

Batch : H10

==============================================================\*/

***/\*.h file for Linked List for Stack i.e. List.h \*/***

#ifndef LIST\_H\_

#define LIST\_H\_

#include <iostream>

struct node

{

int data;

node \*Rchild=NULL;

node \*Lchild=NULL;

};

class List

{

public :

node \*StackData[20];

int top;

List()

{

top=-1;

}

void insert(node \*Tnode);

node \*del();

bool isempty();

};

#endif /\* LIST\_H\_ \*/

=======================================================================

***/\*.cpp file for Linked List operation i.e.List.cpp***

#include<iostream>

#include "List.h"

using namespace std;

void List::insert(node \*Tnode)

{

StackData[++top]=Tnode;

}

node\* List::del()

{

return StackData[top--];

}

bool List::isempty()

{

if(top==-1)

return 1;

return 0;

}

***/\* header file for stack opeartion i.e. Stack.h \*/***

#ifndef STACK\_H\_

#define STACK\_H\_

#include "List.h"

class Stack

{

public:

List li;

void push(node \*Tnode);

node \*pop();

bool isEmpty();

};

#endif /\* STACK\_H\_ \*/

=======================================================================

***/\* .cpp file for stack operation implementation i.e. Stack.cpp***

#include<iostream>

#include "Stack.h"

using namespace std;

void Stack::push(node \*Tnode)

{

li.insert(Tnode);

}

node \*Stack::pop()

{

return li.del();

}

bool Stack::isEmpty()

{

return li.isempty();

}

***/\* .cpp file for Queue and its Operation i.e. Queue.cpp\*/***

#include "List.h"

class queue //CLASS QUEUE

{

node \*arr[20];

int r,f;

public:

queue()

{

f=r=-1;

}

int empty()

{

if(f==-1 && r==-1)

return 1;

return 0;

}

int size()

{

int c=1,t;

if(empty()==1)

c=0;

t=f;

while(t!=r)

{

t++;

c++;

}

return c;

}

void enqueue(node \*x)

{

if(f==-1)

{

f=r=0;

}

else

{

r++;

}

arr[r]=x;

}

node\* dequeue()

{

node \*x;

if(f!=-1)

{

x=arr[f];

if(f==r)

f=r=-1;

else

f++;

}

return x;

}

void init()

{

f=r=-1;

}

};

======================================================================

***/\*Header file for Binary search tree i.e. BST.h \*/***

#ifndef BST\_H\_

#define BST\_H\_

#include "List.h"

#include "Queue.cpp"

#include "Stack.h"

#include <iostream>

class BST

{

public :

node \*Root,\*pre;

queue Q;

Stack S;

BST()

{

Root=NULL;

pre=NULL;

}

void create();

bool insert(int x);

node \*Rec\_search(node \*Root,int key);

void Non\_Rec\_search(int key);

void Tra\_Levelorder();

void Recu\_Tra();

void Rec\_Inorder(node \*T);

void Rec\_Preorder(node \*T);

void Rec\_Postorder(node \*T);

void Non\_Recu\_Tra();

void Non\_Rec\_Inorder(node \*T);

void Non\_Rec\_Preorder(node \*T);

void Non\_Rec\_Postorder(node \*T);

int Non\_Rec\_Height();

void Rec\_Mirror(node \*Root);

void Itra\_Mirror();

node \*Delete(int x,node \*T);

node \*min(node \*T);

int MaxEle();

};

#endif /\* BST\_H\_ \*/

***/\* main C++ program for binary Search Tree with all Operation implementation\*/***

#include <iostream>

using namespace std;

#include "BST.h"

#include "List.h"

void BST::create() //function to create BST

{

int i,n,d;

cout<<"\nHow many Nodes you have :";

cin>>n;

for(i=1;i<=n;i++)

{

cout<<"\nEnter "<<i<<" Node Data";

cin>>d;

if(insert(d)==0)

i--;

}

cout<<"\nBST Created";

}

bool BST::insert(int x)//to insert one node in BST

{

node \*temp=NULL,\*p=NULL,\*q=NULL;

temp=new node;

temp->data=x;

if(Root==NULL)//when Root Node

{

Root=new node;

Root=temp;

return 1;

}

else

{

q=p=Root;

while(p!=NULL) //to find correct place for new node

{

q=p;

if(x>p->data)

p=p->Rchild;

else if(x<p->data)

p=p->Lchild;

else

{

cout<<"\nDublication not Allowed";

return 0;

}

}

if(temp->data>q->data)//insert at Right

{

q->Rchild=temp;

return 1;

}

else //insert at Left

{

q->Lchild=temp;

return 1;

}

}

return 0;

}

node \*BST::Rec\_search(node \*Root,int key)//search key in BST recursively

{

node \*temp=NULL;

if(Root==NULL)

{

cout<<"\nBST is Empty";

return 0;

}

else

{

temp=Root;

if(temp->data==key)//when found

{

cout<<"\nGiven Element FOUND in BST";

return temp;

}

else if(key<temp->data)

{

if(temp->Lchild==NULL)

{

cout<<"\nGiven Element NOT Found in BST";

return temp;

}

else

temp=Rec\_search(temp->Lchild,key);//recursive Call in Left sub Tree

}

else

{

if(temp->Rchild==NULL)

{

cout<<"\nGiven Element NOT Found in BST";

return temp;

}

temp=Rec\_search(temp->Rchild,key);//recursive call in Right sub Tree

}

}

return temp;

}

void BST::Non\_Rec\_search(int key)//search given key in BST Non\_Recursively

{

node \*temp;

if(Root==NULL)

{

cout<<"\nBST is Empty";

return ;

}

else

{

temp=Root;

while(temp!=NULL)//traversing to search key

{

if(key==temp->data)

{

cout<<"\nGiven Element FOUND in BST";

return ;

}

else if(temp->data>key)

temp=temp->Lchild;

else

temp=temp->Rchild;

}

cout<<"\n Given Element NOT Found";

}

}

void BST::Tra\_Levelorder()//function for level order search

{

node \*temp=NULL;

if(Root==NULL)

{

cout<<"\nBST Is Empty";

return ;

}

else

{

cout<<"\nLevel Order Traversal : ";

temp=Root;

Q.enqueue(temp);

while(Q.empty()==0)

{

temp=Q.dequeue();

cout<<temp->data<<"\t";

if(temp->Lchild!=NULL)

Q.enqueue(temp->Lchild);

if(temp->Rchild!=NULL)

Q.enqueue(temp->Rchild);

}

}

}

void BST::Recu\_Tra()//recursive traversaL FOR PREFIX

{

cout<<"\nRecursive Inorder Traverse:";

Rec\_Inorder(Root);

cout<<"\nRecursive Preorder Traverse:";

Rec\_Preorder(Root);

cout<<"\nRecursive Postorder Traverse:";

Rec\_Postorder(Root);

}

//traversal inorder ,post order and Preorder recursively

void BST:: Rec\_Inorder(node \*T)

{

if(T!=NULL)

{

Rec\_Inorder(T->Lchild);

cout<<"\t"<<T->data;

Rec\_Inorder(T->Rchild);

}

}

void BST::Rec\_Preorder(node \*T)

{

if(T!=NULL)

{

cout<<"\t"<<T->data;

Rec\_Preorder(T->Lchild);

Rec\_Preorder(T->Rchild);

}

}

void BST:: Rec\_Postorder(node \*T)

{

if(T!=NULL)

{

Rec\_Postorder(T->Lchild);

Rec\_Postorder(T->Rchild);

cout<<"\t"<<T->data;

}

}

void BST::Non\_Recu\_Tra()//non recursive traversal post fix exp

{

cout<<"\nNon\_Recursive Inorder Traverse:";

Non\_Rec\_Inorder(Root);

cout<<"\nNon\_Recursive Preorder Traverse:";

Non\_Rec\_Preorder(Root);

cout<<"\nNon\_Recursive Postorder Traverse:";

Non\_Rec\_Postorder(Root);

}

void BST:: Non\_Rec\_Postorder(node \*T)

{ node \*previous;

previous=T;

if(T==NULL)

{

cout<<"\nTree is Empty";

return;

}

S.push(T);

while(S.isEmpty()==0)

{

T=S.pop();

if(T->Rchild==NULL && T->Lchild==NULL)

{

cout<<"\t"<<T->data;

previous=T;

}

else

{

if(T->Rchild==previous || T->Lchild==previous)

{

previous=T;

cout<<"\t"<<previous->data;

}

else

{

S.push(T);

if(T->Rchild!=NULL)

S.push(T->Rchild);

if(T->Lchild!=NULL)

S.push(T->Lchild);

}

}

}

}

void BST:: Non\_Rec\_Inorder(node \*T)

{

if(T==NULL)

{

cout<<"\nTree is Empty";

return ;

}

else

{

while(T!=NULL)

{

S.push(T);

T=T->Lchild;

}

while(S.isEmpty()==0)

{

T=S.pop();

cout<<"\t"<<T->data;

T=T->Rchild;

while(T!=NULL)

{

S.push(T);

T=T->Lchild;

}

}

}

}

void BST:: Non\_Rec\_Preorder(node \*T)

{

if(T==NULL)

{

cout<<"\nTreeis Empty";

return ;

}

else

{

while(T!=NULL)

{

cout<<"\t"<<T->data;

S.push(T);

T=T->Lchild;

}

while(S.isEmpty()==0)

{

T=S.pop();

T=T->Rchild;

while(T!=NULL)

{

cout<<"\t"<<T->data;

S.push(T);

T=T->Lchild;

}

}

}

}

int BST::Non\_Rec\_Height()//to find height of BST

{

int h=0,node\_count=0;

node \*temp;

if(Root==NULL)

{

cout<<"\nEmpty BST";

return 0;

}

else

{

temp=Root;

Q.init();

Q.enqueue(temp);

while(1)

{

node\_count=Q.size();//node count from Queue

if(node\_count==0)

return h;

h++;

while(node\_count>0)

{

temp=Q.dequeue();

if(temp->Lchild!=NULL)

Q.enqueue(temp->Lchild);

if(temp->Rchild!=NULL)

Q.enqueue(temp->Rchild);

node\_count--;

}

}

}

return h;

}

void BST::Rec\_Mirror(node \*Root) //make Mirror Image of BST Recusively

{

node \*temp=NULL;

if(Root)

{

temp=Root->Rchild;

Root->Rchild=Root->Lchild;

Root->Lchild=temp;

Rec\_Mirror(Root->Lchild);//recursive call

Rec\_Mirror(Root->Rchild);

}

}

void BST:: Itra\_Mirror()//making Mirror Image non Recursively

{

if(Root==NULL)

{

cout<<"\nEmpty BST";

return ;

}

else

{

Q.init(); //create Empty Queue

node \*temp,\*temp2;

temp=Root;

Q.enqueue(temp);

while(Q.empty()==0)

{

temp=Q.dequeue();

if(temp->Lchild==NULL && temp->Rchild==NULL)

continue;

if(temp->Lchild!=NULL && temp->Rchild!=NULL)

{

temp2=temp->Lchild;

temp->Lchild=temp->Rchild;

temp->Rchild=temp2;

Q.enqueue(temp->Lchild);

Q.enqueue(temp->Rchild);

}

else if(temp->Lchild==NULL)

{

temp->Lchild=temp->Rchild;

temp->Rchild=NULL;

Q.enqueue(temp->Lchild);

}

else

{

temp->Rchild=temp->Lchild;

temp->Lchild=NULL;

Q.enqueue(temp->Rchild);

}

}

cout<<"\nMirror of BST DONE Iteratively";

}

}

node \*BST::min(node \*T)

{

node \*temp;

temp=T;

if(temp->Lchild==NULL)

{

// pre=T;

return temp;

}

else

{

pre=temp;

return min(temp->Lchild);

}

}

node \* BST::Delete(int x,node \*root)//to delete node from BST

{

node \*temp=NULL,\*T=NULL,\*p=NULL;

int del;

p=T=root;

if(T==NULL)

{

cout<<"Empty tree";

}

else

{

while(T!=NULL && T->data!=x)//searching for deleting element

{

if(x<T->data)

{

p=T;

T=T->Lchild;

}

else if(x>T->data)

{

p=T;

T=T->Rchild;

}

}

if(T==NULL)//when not found

{

cout<<"\nGiven Element NOT FOUND";

return temp;

}

if(p==T && T->Lchild==NULL && T->Rchild==NULL )//when Root Node deletion having no child

{

Root=NULL;

cout<<"\nBSt Become Empty ";

return T;

}

if(T->Lchild!=NULL && T->Rchild!=NULL)//deletion when Two child

{

pre=T;

temp=min(T->Lchild);

del=T->data;

T->data=temp->data;

temp->data=del;

if(pre->Lchild==temp)

{

pre->Lchild=NULL;

}

else if(pre->Rchild!=NULL)

{

pre=pre->Rchild;

}

return temp;

}

else

{

temp=T;

if(T->Lchild==NULL && T->Rchild!=NULL)//deletion when there is Right one Child

{

del=T->data;

T->data=T->Rchild->data;

temp=T->Rchild;

temp->data=del;

T->Rchild=NULL;

}

else if(T->Rchild==NULL && T->Lchild!=NULL)//Deletion when there is Left one Child

{

del=T->data;

T->data=T->Lchild->data;

temp=T->Lchild;

temp->data=del;

T->Lchild=NULL;

}

else //deletion when no child node

if(p->Rchild==temp)

{

temp=p->Rchild;

p->Rchild=NULL;

}

else

{

temp=p->Lchild;

p->Lchild=NULL;

}

}

}

return temp;

}

int BST::MaxEle()

{

int G;

node \*temp=NULL;

if(Root==NULL)

{

cout<<"\nBST Is Empty";

return 0;

}

else

{

cout<<"\nLevel Order Traversal : ";

temp=Root;

G=temp->data;

Q.enqueue(temp);

while(Q.empty()==0)

{

temp=Q.dequeue();

if(G<temp->data)

G=temp->data;

if(temp->Lchild!=NULL)

Q.enqueue(temp->Lchild);

if(temp->Rchild!=NULL)

Q.enqueue(temp->Rchild);

}

return G;

}

return G;

}

int main() {

BST bst;

int key,de;

node \*d;

int c;

while(c<12)

{

cout<<"\n\n1.Create BST \n2.Search (Recursive) \n3.Search (Non\_Recursive) \n4.Level Order Traversal";

cout<<"\n5.Recursive Traversal \n6.Non\_Recursive Traversal";

cout<<"\n7.Height of BST(Non\_Recursive)\n8.Mirror BST (Recursive) \n9.Mirror BST (Iterative) \n10.Max Element of BST";

cout<<"\n11.Delete \n12.Exit";

cout<<"\nEnter Your Choice :";

cin>>c;

switch(c)

{

case 1:

bst.create();

break;

case 2:

cout<<"\nEnter Element to search :";

cin>>key;

bst.Rec\_search(bst.Root,key);

break;

case 3:

cout<<"\nEnter Element to search :";

cin>>key;

bst.Non\_Rec\_search(key);

break;

case 4:

bst.Tra\_Levelorder();

break;

case 5:

bst.Recu\_Tra();

break;

case 6:

bst.Non\_Recu\_Tra();

break;

case 7:

cout<<"\nHeight of BST: "<<bst.Non\_Rec\_Height();

break;

case 8:

cout<<"\nBefor Mirroring BST Level Order Traversing";

bst.Tra\_Levelorder();

bst.Rec\_Mirror(bst.Root);

cout<<"\nMirroring DONE Recursively....";

cout<<"\nAfter Mirroring BST Level Order Traversal";

bst.Tra\_Levelorder();

break;

case 9:

cout<<"\nBefor Mirroring BST Level Order Traversing";

bst.Tra\_Levelorder();

bst.Itra\_Mirror();

cout<<"\nAfter Mirroring BST Level Order Traverse";

bst.Tra\_Levelorder();

break;

case 10:

cout<<"\nThe LARGEST Element of BST is : "<<bst.MaxEle();

break;

case 11:

cout<<"\nEnter Element for Delete :";

cin>>de;

d=bst.Delete(de,bst.Root);

cout<<"\nElement deleted : "<<d->data;

break;

case 12:

return 0;

break;

}

}

return 0;

}

+=+===+++++====+======++++====+=++++===++++====+++++====+=============

***OutPut of Program***

=========================

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :1

How many Nodes you have :5

Enter 1 Node Data55

Enter 2 Node Data44

Enter 3 Node Data66

Enter 4 Node Data40

Enter 5 Node Data60

BST Created

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :2

Enter Element to search :55

Given Element FOUND in BST

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :3

Enter Element to search :666

Given Element NOT Found

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :4

Level Order Traversal : 55 44 66 40 60

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :5

Recursive Inorder Traverse: 40 44 55 60 66

Recursive Preorder Traverse: 55 44 40 66 60

Recursive Postorder Traverse: 40 44 60 66 55

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :6

Non\_Recursive Inorder Traverse: 40 44 55 60 66

Non\_Recursive Preorder Traverse: 55 44 40 66 60

Non\_Recursive Postorder Traverse: 40 44 60 66 55

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :7

Height of BST: 3

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :8

Befor Mirroring BST Level Order Traversing

Level Order Traversal : 55 44 66 40 60

Mirroring DONE Recursively....

After Mirroring BST Level Order Traversal

Level Order Traversal : 55 66 44 60 40

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :9

Befor Mirroring BST Level Order Traversing

Level Order Traversal : 55 66 44 60 40

Mirror of BST DONE Iteratively

After Mirroring BST Level Order Traverse

Level Order Traversal : 55 44 66 40 60

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :10

Level Order Traversal :

The LARGEST Element of BST is : 66

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :11

Enter Element for Delete :60

Element deleted : 60

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :11

Enter Element for Delete :55

Element deleted : 55

1.Create BST

2.Search (Recursive)

3.Search (Non\_Recursive)

4.Level Order Traversal

5.Recursive Traversal

6.Non\_Recursive Traversal

7.Height of BST(Non\_Recursive)

8.Mirror BST (Recursive)

9.Mirror BST (Iterative)

10.Max Element of BST

11.Delete

12.Exit

Enter Your Choice :12