AIM:-

Assignment 11

Create a binary search tree( BST)

a)Perform recursive inorder , preorder and postorder traversals.

b) For BST, Perform nonrecursive inorder , preorder and postorder traversal.

OBJECTIVE:-

Traversing a tree means visiting every node in the tree. You might for instance want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Theory :-

1. Uses of Inorder In case of binary search trees (BST), Inorder traversal gives nodes in non- decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.
2. Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.
3. Postorder traversal is used to delete the tree.To delete a tree we must traverse all the nodes of the tree and delete them one by one,because before deleting the parent node we should delete its children nodes first.

Postorder traversal is also useful to get the postfix expression of an expression tree.

# include <iostream>

# include <cstdlib>

using namespace std;

struct node //NODE

{

int info;

struct node \*left;

struct node \*right;

}\*root;

class BST //Class

{

public:

void insert(node \*, node \*);

void preorder(node \*);

void inorder(node \*);

void postorder(node \*);

BST()

{

root = NULL; //Constructor

}

};

int main() //MAIN

{

int choice, num;

BST bst; //object of class

node \*temp;

while (1)

{

cout<<"-----------------"<<endl;

cout<<"Operations on BST"<<endl;

cout<<"-----------------"<<endl;

cout<<"1.Insert Elements "<<endl;

cout<<"2.Inorder Traversal"<<endl;

cout<<"3.Preorder Traversal"<<endl;

cout<<"4.Postorder Traversal"<<endl;

cout<<"5.Quit"<<endl;

cout<<"Enter your choice : ";

cin>>choice;

switch(choice)

{

case 1:

temp = new node;

cout<<"Enter the number to be inserted : ";

cin>>temp->info;

bst.insert(root, temp);

break;

case 2:

cout<<"Inorder Traversal of BST:"<<endl;

bst.inorder(root);

cout<<endl;

break;

case 3:

cout<<"Preorder Traversal of BST:"<<endl;

bst.preorder(root);

cout<<endl;

break;

case 4:

cout<<"Postorder Traversal of BST:"<<endl;

bst.postorder(root);

cout<<endl;

break;

case 5:

exit(1);

default:

cout<<"Wrong choice"<<endl;

}

}

}

/\*

\* Inserting Element into the Tree

\*/

void BST::insert(node \*tree, node \*newnode)

{

if (root == NULL)

{

root = new node;

root->info = newnode->info;

root->left = NULL;

root->right = NULL;

cout<<"Root Node is Added"<<endl;

return;

}

if (tree->info == newnode->info) //NO Duplication

{

cout<<"Element already in the tree"<<endl;

return;

}

if (tree->info > newnode->info)

{

if (tree->left != NULL)

{

insert(tree->left, newnode); //Call recursively left

}

else

{

tree->left = newnode;

(tree->left)->left = NULL;

(tree->left)->right = NULL;

cout<<"Node Added To Left"<<endl;

return;

}

}

else

{

if (tree->right != NULL)

{

insert(tree->right, newnode); //Call recursively right

}

else

{

tree->right = newnode;

(tree->right)->left = NULL;

(tree->right)->right = NULL;

cout<<"Node Added To Right"<<endl;

return;

}

}

}

/\*

\* Pre Order Traversal

\*/

void BST::preorder(node \*ptr)

{

if (root == NULL)

{

cout<<"Tree is empty"<<endl;

return;

}

if (ptr != NULL)

{

cout<<ptr->info<<" ";

preorder(ptr->left);

preorder(ptr->right);

}

}

/\*

\* In Order Traversal

\*/

void BST::inorder(node \*ptr)

{

if (root == NULL)

{

cout<<"Tree is empty"<<endl;

return;

}

if (ptr != NULL)

{

inorder(ptr->left);

cout<<ptr->info<<" ";

inorder(ptr->right);

}

}

/\*

\* Postorder Traversal

\*/

void BST::postorder(node \*ptr)

{

if (root == NULL)

{

cout<<"Tree is empty"<<endl;

return;

}

if (ptr != NULL)

{

postorder(ptr->left);

postorder(ptr->right);

cout<<ptr->info<<" ";

}

}



