ASSIGNMENT 1

Aim:

Construct an expression tree from postfix expression and perform recursive Inorder, Preorder and Post order traversals

Objective:

1. Understand the concept of expression tree and binary tree.

2. Understand the recursive traversal of an expression tree

Theory:

1. Definition of an expression tree with diagram.

Algebraic expressions such as a/b + (c-d) e The terminal nodes (leaves) of an expression tree are the variables or constants in the expression (a, b, c, d, and e). The non-terminal nodes of an expression tree are the operators (+, -, , and ). Notice that the parentheses which appear in Equation do not appear in the tree. Nevertheless, the tree representation has captured the intent of the parentheses since the subtraction is lower in the tree than the multiplication.

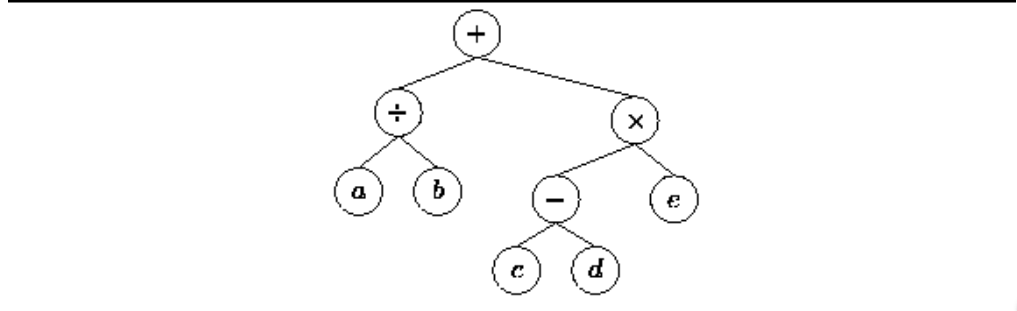


Figure: Tree representing the expression a/b+(c-d)e

1. Show the different type of traversals with example

To traverse a non-empty binary tree in preorder,

1. Visit the root.

2. Traverse the left subtree.

3. Traverse the right subtree.

To traverse a non-empty binary tree in inorder:

1. Traverse the left subtree.

2. Visit the root.

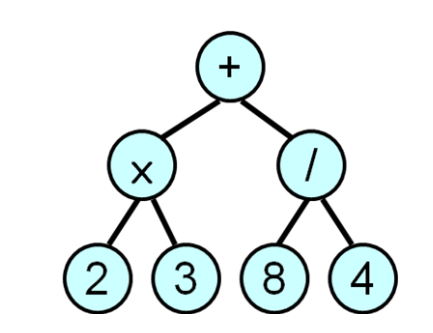
3. Traverse the right subtree.

To traverse a non-empty binary tree in postorder,

1. Traverse the left subtree.

2. Traverse the right subtree.

3. Visit the root.



Pre-order (prefix) + × 2 3 / 8 4

In-order (infix) 2 × 3 + 8 / 4

Post-order (postfix) 2 3 × 8 4 / +

ALGORITHM:

Define a class for Binary Tree (Information, Left Pointer & Right Pointer)

* **Create Expression Tree:**

CreateTree() Root& Node pointer variable of type structure. Stack is an pointer array of type class. String is character array which contains postfix expression. Top is a variable of type node. t,tn,l,r of type treenode.

Step 1: Top = NULL,

Step 2: Do Steps 3,4,5 While String[I] != NULL

Step 3: Create Node of type of class

Step 4: node->data=String[I];

Step 5:

if(( (String[I]>=65 )&& (String[I]<=90) )||( (String[I]>=97) && (String[I]<=122 ) ) )

If top==NULL then, node->nxt=top; top=node;

Else top=node

Else

l=pop()

r=pop()

tn->lptr=l

tn->rptr=r

push(tn)

step 6: displaying traversals

for inorder traversal, displayinorder(root)

for preorder traversal, displaypreorder(root)

for postorder traversal, displaypostorder(root)

step 7: end

* **Inorder Traversal Recursive algorithm :**

Tree is pointer of type class.

displayinorder (root)

Step 1: If root!= NULL

Step 2: displayinorder ( root ->lptr)

Step 3: Print root -> expression

Step 4: displayinorder ( root -> rptr)

* **Postorder Traversal Recursive algorithm:**

Tree is pointer of type class.

displaypostorder (root)

Step 1: If root!= NULL

Step 2: displaypostorder ( root -> lptr)

Step 3: displaypostorder ( root -> rptr)

Step 4: Print root -> expression

* **Preorder Traversal Recursive algorithm:**

Tree is pointer of type class.

displaypreorder (root)

Step 1: If root!= NULL

Step 2: Print root ->expression

Step 3: displaypreorder ( root -> lptr)

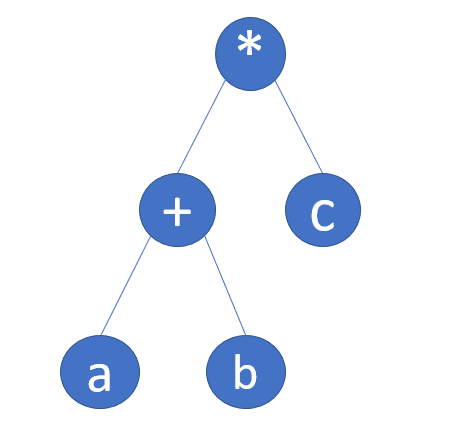
Step 4: displaypreorder ( root ->rptr )

**INPUT:**

Postfix Expression: a b + c \*

**OUTPUT:**

Display result of each operation with error checking.

**Expression tree:**

**OUTPUT:**

Inorder traversal :  a+b\*c

Preoreder traversal : \*a+bc

Postorder traversal :  a+bc\*

CPP SOURCE CODE:

#include<iostream>

using namespace std;

// treenode class containing data as well as create() function for creating a tree-node

class treenode{

public:

char expression;

treenode \*lptr;

treenode \*rptr;

treenode()

{

lptr=rptr=NULL;

}

treenode \*create(char c,treenode \*l,treenode \*r)

{

treenode \*newnode;

newnode=new treenode;

newnode->expression=c;

newnode->lptr=l;

newnode->rptr=r;

return newnode;

}

};

// node() function for having data which we want the node of a stack to contain

class node{

public:

treenode \*tptr;

node \*nxt;

};

// stack() function containing push(),pop() and display() functions

class stack{

public:

node \*top;

stack()

{

top=NULL;

}

void push(treenode \*a){

node \*b;

b=new node;

b->tptr=a;

b->nxt=NULL;

if(top)

{

b->nxt=top;

top=b; }

else{

top=b;

}

}

treenode \*pop()

{

node \*a;

a=top;

top=top->nxt;

treenode \*tn=a->tptr;

delete a;

return (tn);

}

void display()

{

cout<<"\n The inorder traversal of entered postfix expression : ";

displayinorder(top->tptr);

cout<<"\n The pre-order traversal of entered postfix expression : ";

displaypreorder(top->tptr);

cout<<"\n The post-order traversal of entered postfix expression : ";

displaypostorder(top->tptr);

cout<<"\n\n";

}

void displayinorder(treenode \*root)

{

if(root)

{

displayinorder(root->lptr);

cout<<root->expression;

displayinorder(root->rptr);

}

}

void displaypreorder(treenode \*root)

{

if(root)

{

cout<<root->expression;

displayinorder(root->lptr);

displayinorder(root->rptr);

}

}

void displaypostorder(treenode \*root)

{

if(root)

{

displayinorder(root->lptr);

displayinorder(root->rptr);

cout<<root->expression;

}

}

};

int main()

{

char e[50];

// we'll enter a postfix expression having maximum length of 50 characters

cout<<"\n enter the postfix expression : ";

cin>>e;

stack s;

treenode t,\*tn,\*l,\*r;

for(int i=0;e[i]!='\0';i++)

{

// checking if the entered character is an alphabet (operand)

if(((e[i]>=65)&&(e[i]<=90))||((e[i]>=97)&&(e[i]<=122)))

{

tn=t.create(e[i],NULL,NULL);

s.push(tn);

}

else

{

// checking if the entered character is an operator

r=s.pop();

l=s.pop();

tn=t.create(e[i],l,r);

s.push(tn);

}

}

s.display();

return 0;

}

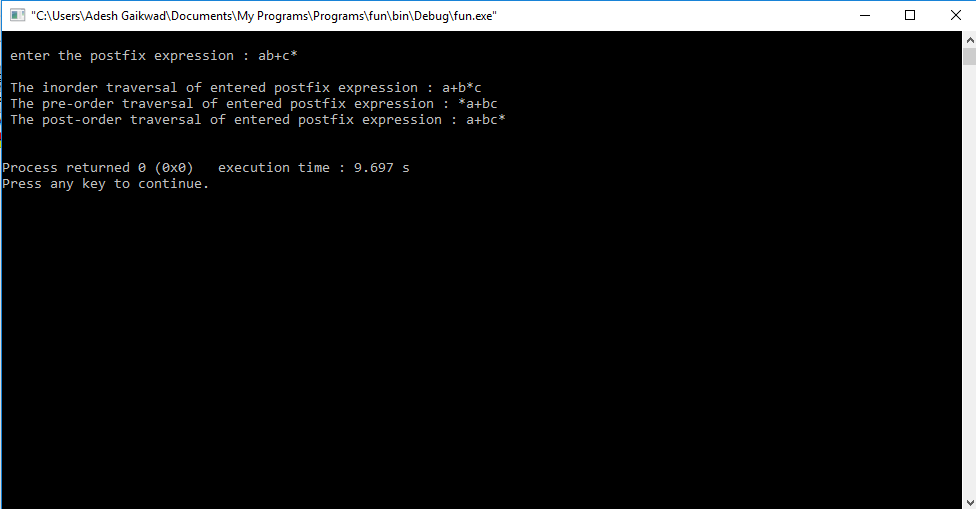
**OUTPUT:**

enter the postfix expression : ab+c\*

 The inorder traversal of entered postfix expression : a+b\*c

 The pre-order traversal of entered postfix expression : \*a+bc

 The post-order traversal of entered postfix expression : a+bc\*



**CONCLUSION:**

Through this assignment, we learnt and performed how to do recursive inorder, preorder and postorder traversal of an expression tree.