**ADVANCED DATA STRUCTURES**

**GROUP A**

**ASSIGNMENT 4: Expression Tree.**

**BATCH B1**

**YEAR : 2017-18**

**COLLEGE : VIIT**

**DATE OF COMPLETION : 30/03/2018**

**Aim:**

For given expression eg. a-b\*c- d/e+f construct inorder sequence and traverse it using postorder traversal (non recursive and recursive).

**Observation:**

To create an expression tree using BST and traverse it in post-order.

**Theory:**

**Deﬁnition:**

The leaves of a binary expression tree are operands, such as constants or variable names, and the other nodes contain operators. These particular trees happen to be binary, because all of the operations are binary, and although this is the simplest case, it is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the unary minus operator. An expression tree, T, can be evaluated by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees.

**Traversal :**

An algebraic expression can be produced from a binary expression tree by recursively producing a parenthesized left expression, then printing out the operator at the root, and ﬁnally recursively producing a parenthesized right expression. This general strategy (left, node, right) is known as an in-order traversal. An alternate traversal strategy is to recursively print out the left subtree, the right subtree, and then the operator. This traversal strategy is generally known as post-order traversal. A third strategy is to print out the operator ﬁrst and then recursively print out the left and right subtree.

These three standard depth-ﬁrst traversals are representations of the three diﬀerent expression formats: inﬁx, postﬁx, and preﬁx.

An inﬁx expression is produced by the inorder traversal, a postﬁx expression is produced by the post-order traversal, and a preﬁx expression is produced by the pre-order traversal.

**Algorithms:**

1. void create(char \*e)

(a) start.

(b) check for alphabet or opening bracket if alphabet is present then push on ans array.

(c) check for alphanumeric char if present then check its priority.

(d) if priority of incoming operator is greater than priority of operator in stack then push it on stack

(e) else pop all the operator into ans array until priority of incoming operator becomes less. (f) stop.

2. int priority(char x)

(a) start.

(b) return type integer and returns priority of char.

(c) arguments char \*e.

(d) stop.

3. void exp read()

(a) start.

(b) read the ans array.

(c) if alphabet comes then pop two times.

(d) else if alphanumeric char comes then put one alphabet left and one on right and push this on stack.

(e) stop.

4. void postﬁx(node \*parent)

(a) Start

(b) if parent is NULL, return

(c) recursively travel parent->left

(d) recursively travel parent->right

(e) print parent and copy in array arr

5. void inﬁx(node \*parent)

(a) Start

(b) if parent is NULL, return

(c) recursively travel parent->left

(d) print parent and copy in array arr

(e) recursively travel parent->right

**Code:**

#include<iostream>

#include<time .h>

#include<ctype .h>

#include<stack>

using namespace std ;

char ex [2 0] ;

char ans [20] ;

class expression tree

{

public : stack<char>st ;

void conversion ( char ∗e );

int priority ( char x );

};

int expression tree : : priority ( char x)

{

switch (x)

{

case ’+ ’:

return 1;

case ’−’:

return 1;

case ’/ ’:

return 2;

case ’∗ ’:

return 2;

default :

return 0;

}

}

void expression tree : : conversion ( char ∗e)

{

char op ;

int count=0;

while (∗e)

{

i f ( isalpha (∗e ))

{

ans [ count++]=∗e ;

e++;

}

i f (∗e==’(’)

{

ans [ count++]=∗e ;

e++;

}

i f (∗e==’)’)

{

op=st . top ();

// st . pop ();

while (op!= ’( ’)

{

ans [ count++]=op;

op=st . top ();

st . pop ();

}

e++;

}

i f (∗e==’+’ || ∗e==’−’ || ∗e==’/’ || ∗e==’∗’ )

{

i f ( st . empty ())

{

st . push(∗e );

e++;

}

else { op=st . top ();

i f ( priority (∗e)>priority (op))

{

st . push(∗e );

e++;

}

else

{

while ( priority (op)>=priority (∗e ))

{

ans [ count++]=st . top ();

st . pop ();

i f (! st . empty ())

{

op=st . top ();

}

else

{

break ;

}

}

st . push(∗e );

e++;

}

}

}

}

while (! st . empty ())

{

ans [ count++]=st . top ();

st . pop ();

}

int k = 0;

while (k<count )

{

cout<<ans [ k ] ;

k++;

}

cout<<endl ;

}

class tree

{

typedef struct node

{

char data ;

node ∗l e f t ;

node ∗right ;

}node ;

public :

node ∗root ,∗New;

stack<node∗>ST;

int count=−1;

void exp read ();

void infix (node∗);

void postfix (node∗ parent );

};

void tree : : exp read ()

{

node ∗x,∗y ;

while (ans[++count ])

{

New=new node ;

New−>l e f t=NULL;

New−>right=NULL;

New−>data=ans [ count ] ;

i f ( isalnum (New−>data ))

{

ST. push(New);

}

else

{

x=ST. top ();

ST. pop ();

y=ST. top ();

ST. pop ();

New−>right=x ;

New−>l e f t=y ;

ST. push(New);

}

}

root=New;

}

void tree : : infix (node∗ parent )

{

i f ( parent == NULL) return ;

infix ( parent−>l e f t );

cout<<parent−>data ;

infix ( parent−>right );

}

void tree : : postfix (node∗ parent )

{

i f ( parent == NULL) return ;

postfix ( parent−>l e f t );

postfix ( parent−>right ); cout<<parent−>data ;

}

int main()

{ expression tree ET;

tree treeob ;

cout<<”enter the infix expression”<<endl ;

cin>>ex ;

ET. conversion (ex );

treeob . exp read ();

treeob . infix ( treeob . root );

cout<<endl ; treeob . postfix ( treeob . root );

cout<<”\n time ”<<t<<endl ; return 0; }

Input And Output:

Enter expression : a+b∗c+d

Postfix : abc∗+d+

Infix Tree : a+b∗c+d

Postfix tree : abc∗+d+

Enter expression : a+b−c∗d/e

Postfix : ab+cd∗e/−

Infix Tree : a+b−c∗d/e

Postfix tree : ab+cd∗e/−

**Conclusion:**

We learned how to create an expression tree using BST and then traverse it in post-order.