**PUNE INSTITUTE OF COMPUTER TECHNOLOGY DHANKAWADI, PUNE**

**Data Structures And Algorithms(DSA)**

**Assignment No. 04**

**Title : Expression Tree**

**SE-IT-10**  **ACADEMIC YEAR :- 2020-2021**

**Name :- Diptesh Ravindra Varule Roll No :- 23277**

**Source Code :**

//============================================================================

// Name : dsa\_assignment4.cpp

// Author : Diptesh Varule

// Version : Updating…..

// Copyright : The Cartel

// Description : Hello World in C++, Ansi-style

//============================================================================

**Source Code:**

#include<iostream>

#include<string.h>

#include<string>

#include<cmath>

using namespace std;

template<typename T>

//class Node

class Node

{

public:

//Data members of class Node

T data;

Node<T>\* next;

//Parameterized constructor of class Node

Node(T data);

};

template<typename T>

//class Stack

class Stack

{

//Private data members

Node<T>\* Top; //pointer to the top element of stack

int size; //variable for size of the stack

//int maxSize; //variable for maximum size of stack

public:

Stack(); //default constructor

//~Stack(); //destructor

bool isEmpty(); //method to check if the stack is empty

void push(T element); //method to push new element in stack

T pop(); //method to delete top element of stack

void display(); //method to display complete stack

T peep(); //method to display only the top element of stack

};

class InfixToPostfix

{

public:

bool isOperator(char c);//method to check if the character is an operator

int associativity(char c);//method to check associativity

int precedence(char c);//method to check precedence

string infixToPostfix(string infix);//method to convert infix to postfix

bool isValid(string infix);//method to check validity of input expression

string infixToPrefix(string infix);//method to convert infix to prefix

string reverseArray(string expression);//method to reverse a string

};

template<typename T>

//method of bool return type to check if stack is empty

bool Stack<T>::isEmpty()

{

return (size==0);

}

template<typename T>

//method to push new data

void Stack<T>::push(T element)

{

Node<T>\* newNode = new Node<T>(element);

newNode->next=Top;

Top=newNode;

size++; //incrementing size variable

return;

}

template<typename T>

//method to delete the top element of stack

T Stack<T>::pop()

{

Node<T>\* temp=Top;

Top=Top->next;

temp->next=NULL;

T del = temp->data; //storing the element to be deleted

delete temp;

size--; //decrementing size variable on delete

return del;

}

template<typename T>

//method to display the complete stack

void Stack<T>::display()

{

InfixToPostfix object;

string displayStr;

//checking isEmpty condition

if(isEmpty())

{

cout<<"Empty.";

return;

}

Node<T>\* temp=Top;

while(temp!=NULL)

{

displayStr+=temp->data;

temp=temp->next;

}

string reversed = object.reverseArray(displayStr);

cout<<reversed;

}

template<typename T>

//method to display only top element in the stack

T Stack<T>::peep()

{

return Top->data;

}

template<typename T>

//parameterized constructor of class Node

Node<T>::Node(T data)

{

this->data=data;

next=NULL;

}

template<typename T>

//default constructor of class Stack

Stack<T>::Stack()

{

Top=NULL;

size=0;

//maxSize = 0;

}

/\*template<typename T>

//destructor of class Stack

Stack<T>::~Stack()

{

Node<T>\* temp=Top;

while(Top!=NULL)

{

temp=Top;

Top=Top->next;

delete(temp);

}

}\*/

bool InfixToPostfix::isOperator(char c)

{

if(c=='+' || c=='\*' || c=='/' || c=='^' || c=='-')

{

return true;

}

return false;

}

//Method to check associativity of operators

int InfixToPostfix::associativity(char c)

{

if(c=='^')

{

return 2;

}

return 1;

}

//Method to check precedence of operators

int InfixToPostfix::precedence(char c)

{

if(c=='^')

{

return 3;

}

else if(c=='\*' || c=='/')

{

return 2;

}

else if(c=='+' || c=='-')

{

return 1;

}

else

{

return -1;

}

}

//Method to convert Infix to postfix

string InfixToPostfix::infixToPostfix(string infix)

{

Stack<char> s;

string postfix;

int i=0;

//Start of while loop

while(i<infix.length())

{

char token = infix[i];

if(isalnum(token))

{

postfix+=token;

}

if(token=='(')

{

s.push(token);

}

if(token==')')

{

char out = s.pop();

while(out!='(')

{

postfix+=out;

out=s.pop();

}

}

if(isOperator(token))

{

if(s.isEmpty())

{

s.push(token);

}

else

{

if(s.peep()=='(')

{

s.push(token);

}

else

{

if(precedence(token)<precedence(s.peep()))

{

while(!(s.isEmpty()))

{

postfix+=s.pop();

}

continue;

}

if(precedence(token)==precedence(s.peep()))

{

if(associativity(token)==1)

{

postfix+=s.pop();

s.push(token);

}

if(associativity(token)==2)

{

s.push(token);

}

}

else

{

s.push(token);

}

}

}

}

i++;

}//End of while loop

while(!(s.isEmpty()))

{

postfix+=s.pop();

}

return postfix;

}

//Method to reverse a string

string InfixToPostfix::reverseArray(string expression)

{

int sizeOfArray=expression.length();

string reversed;

int i;

for(i=sizeOfArray-1 ; i>=0 ; i--)

{

reversed+=expression[i];

}

return reversed;

}

//Method to convert infix to prefix

string InfixToPostfix::infixToPrefix(string infix)

{

int sizeOfArray=infix.length();

//calling reverseArray() Method

string ptrArray=reverseArray(infix);

for(int i=0 ; i<sizeOfArray ; i++)

{

if(ptrArray[i]=='(')

{

ptrArray[i]=')';

i++;

}

else if(ptrArray[i]==')')

{

ptrArray[i]='(';

i++;

}

}

//calling infixToPostfix method

string postfixExp = infixToPostfix(ptrArray);

string prefix=reverseArray(postfixExp);

return prefix;

}

//Method to check validity of the input expression

bool InfixToPostfix::isValid(string infix)

{

int sizeOfArray=infix.length();

int count1=0,count2=0;

for(int i=0 ; i<sizeOfArray ; i++)

{

if(isOperator(infix[i]))

{

count1++;

}

if(isalnum(infix[i]))

{

count2++;

}

}

if(count2!=(count1+1))

{

return false;

}

int count=0 , count3=0;

for(int i=0 ; i<sizeOfArray ; i++)

{

if(infix[i]=='(')

{

count++;

}

if(infix[i]==')')

{

count3++;

}

}

if(count!=count3)

{

return false;

}

for(int i=0 ; i<sizeOfArray-1 ; i++)

{

if((isOperator(infix[i]) && isOperator(infix[i+1])) || (isalnum(infix[i]) && isalnum(infix[i+1])))

{

return false;

}

}

if(isOperator(infix[0]) || isOperator(infix[sizeOfArray-1]))

{

return false;

}

return true;

}

class TNode

{

public:

char data;

TNode\* leftNode;

TNode\* rightNode;

TNode();

};

class ExpressionTree

{

public:

TNode\* getNode(char data); //method to return node

TNode\* createNode(string Infix); //method to create tree

void recursiveInorder(TNode\* root); //method to traverse binary tree in Inorder recursively

void recursivePreorder(TNode\* root); //method to traverse binary tree in Preorder recursively

void recursivePostorder(TNode\* root); //method to traverse binary tree in Postorder recursively

void nonRecursiveInorder(TNode\* root);

void nonRecursivePreorder(TNode\* root);

void nonRecursivePostorder(TNode\* root);

};

TNode\* ExpressionTree::getNode(char data)

{

TNode\* temp = new TNode;

temp->leftNode=temp->rightNode=NULL;

temp->data=data;

return temp;

}

//method to create tree

TNode\* ExpressionTree::createNode(string infix)

{

Stack<TNode\*> stack;

InfixToPostfix post;

TNode\* temp;

string postfix = post.infixToPostfix(infix);

for(int i=0 ; i<postfix.length() ; i++)

{

if(!(post.isOperator(postfix[i])))

{

temp = getNode(postfix[i]);

stack.push(temp);

}

else

{

temp = getNode(postfix[i]);

temp->rightNode = stack.pop();

temp->leftNode = stack.pop();

stack.push(temp);

}

}

temp = stack.pop();

return temp;

}

//method to traverse binary tree in Inorder recursively

void ExpressionTree::recursiveInorder(TNode\* root)

{

if(root==NULL)

{

return;

}

recursiveInorder(root->leftNode);

cout<<root->data<<" ";

recursiveInorder(root->rightNode);

}

//method to traverse binary tree in Postorder recursively

void ExpressionTree::recursivePostorder(TNode\* root)

{

if(root==NULL)

{

return;

}

recursivePostorder(root->leftNode);

recursivePostorder(root->rightNode);

cout<<root->data<<" ";

}

//method to traverse binary tree in Preorder recursively

void ExpressionTree::recursivePreorder(TNode\* root)

{

if(root==NULL)

{

return;

}

cout<<root->data<<" ";

recursivePreorder(root->leftNode);

recursivePreorder(root->rightNode);

}

void ExpressionTree::nonRecursiveInorder(TNode\* root)

{

TNode\* currentNode=root;

Stack<TNode\*> stack;

do

{

while(currentNode!=NULL)

{

stack.push(currentNode);

currentNode = currentNode->leftNode;

}

currentNode = stack.pop();

cout<<currentNode->data<<" ";

currentNode = currentNode->rightNode;

}while(currentNode!=NULL || !(stack.isEmpty()));

}

void ExpressionTree::nonRecursivePostorder(TNode\* root)

{

Stack<TNode\*> stack1;

Stack<TNode\*> stack2;

stack1.push(root);

TNode\* currentNode;

do

{

currentNode = stack1.peep();

stack1.pop();

stack2.push(currentNode);

if(currentNode->leftNode != NULL)

{

stack1.push(currentNode->leftNode);

}

if(currentNode->rightNode != NULL)

{

stack1.push(currentNode->rightNode);

}

}while(!(stack1.isEmpty()));

while(!(stack2.isEmpty()))

{

currentNode = stack2.peep();

stack2.pop();

cout<<currentNode->data<<" ";

}

}

void ExpressionTree::nonRecursivePreorder(TNode\* root)

{

Stack<TNode\*> stack;

stack.push(root);

do

{

TNode\* currentNode = stack.peep();

cout<<currentNode->data<<" ";

stack.pop();

if(currentNode->rightNode != NULL)

{

stack.push(currentNode->rightNode);

}

if(currentNode->leftNode != NULL)

{

stack.push(currentNode->leftNode);

}

}while(!(stack.isEmpty()));

}

//default constructor of TNode class

TNode::TNode()

{

this->data = data;

leftNode=NULL;

rightNode=NULL;

}

int main() {

int choice;

InfixToPostfix object;

int ch;

//Menu driven

do

{

cout<<"\n Choose\n1. Recursive Traversal \n2. Non-Recursive Traversal \n 3.Exit"<<endl;

cout<<"Enter your choice: "<<endl;

cin>>ch;

switch(ch)

{

case 1:

{

do

{

cout<<"\n 1. In-Order Traversal \n 2. Post-Order Traversal \n"

" 3. Pre-Order Traversal \n 4. Exit"<<endl;

cin>>choice;

switch(choice)

{

case 1:

{

string infix;

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

if(object.isValid(infix))

{

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Recursive In-Order Traversal of tree: "<<endl;

tree.recursiveInorder(root);

}

else

{

while(!(object.isValid(infix)))

{

cout<<"The infix expression is invalid!!!"<<endl;

cout<<"Re-Enter Infix Expression: "<<endl;

cin>>infix;

}

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Recursive In-Order Traversal of tree: "<<endl;

tree.recursiveInorder(root);

}

break;

}

case 2:

{

string infix;

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

if(object.isValid(infix))

{

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Recursive Post-Order Traversal of tree: "<<endl;

tree.recursivePostorder(root);

}

else

{

while(!(object.isValid(infix)))

{

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

}

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Recursive Post-Order Traversal of tree: "<<endl;

tree.recursivePostorder(root);

}

break;

}

case 3:

{

string infix;

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

if(object.isValid(infix))

{

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Recursive Pre-Order Traversal of tree: "<<endl;

tree.recursivePreorder(root);

}

else

{

while(!(object.isValid(infix)))

{

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

}

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Recursive Pre-Order Traversal of tree: "<<endl;

tree.recursivePreorder(root);

}

break;

}

case 4:

{

cout<<"Loop exited successfully :)"<<endl;

break;

}

}

}while(choice!=4);

break;

}

case 2:

{

do

{

cout<<"\n 1. In-Order Traversal \n 2. Post-Order Traversal \n"

" 3. Pre-Order Traversal \n 4. Exit"<<endl;

cin>>choice;

switch(choice)

{

case 1:

{

string infix;

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

if(object.isValid(infix))

{

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Non-Recursive In-Order Traversal of tree: "<<endl;

tree.nonRecursiveInorder(root);

}

else

{

while(!(object.isValid(infix)))

{

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

}

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Non-Recursive In-Order Traversal of tree: "<<endl;

tree.recursiveInorder(root);

}

break;

}

case 2:

{

string infix;

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

if(object.isValid(infix))

{

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Non-Recursive Post-Order Traversal of tree: "<<endl;

tree.nonRecursivePostorder(root);

}

else

{

while(!(object.isValid(infix)))

{

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

}

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Non-Recursive Post-Order Traversal of tree: "<<endl;

tree.recursivePostorder(root);

}

break;

}

case 3:

{

string infix;

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

if(object.isValid(infix))

{

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Non-Recursive Pre-Order Traversal of tree: "<<endl;

tree.nonRecursivePreorder(root);

}

else

{

while(!(object.isValid(infix)))

{

cout<<"Enter Infix Expression: "<<endl;

cin>>infix;

}

ExpressionTree tree;

TNode\* root = tree.createNode(infix);

cout<<"Non-Recursive Pre-Order Traversal of tree: "<<endl;

tree.recursivePreorder(root);

}

break;

}

case 4:

{

cout<<"Loop exited successfully :)"<<endl;

break;

}

}

}while(choice!=4);

break;

}

case 3:

{

cout<<"Application exited successfully :)"<<endl;

break;

}

}

}while(ch!=3);

return 0;

}

**Output :**

Choose

1. Recursive Traversal

2. Non-Recursive Traversal

3.Exit

Enter your choice:

1

1. In-Order Traversal

2. Post-Order Traversal

3. Pre-Order Traversal

4. Exit

1

Enter Infix Expression:

((a+b)\*d)/c-e

Recursive In-Order Traversal of tree:

a + b \* d / c - e

1. In-Order Traversal

2. Post-Order Traversal

3. Pre-Order Traversal

4. Exit

2

Enter Infix Expression:

((a+b)\*d)/c-e

Recursive Post-Order Traversal of tree:

a b + d \* c / e -

1. In-Order Traversal

2. Post-Order Traversal

3. Pre-Order Traversal

4. Exit

3

Enter Infix Expression:

((a+b)\*d)/c-e

Recursive Pre-Order Traversal of tree:

- / \* + a b d c e

1. In-Order Traversal

2. Post-Order Traversal

3. Pre-Order Traversal

4. Exit

4

Loop exited successfully :)

Choose

1. Recursive Traversal

2. Non-Recursive Traversal

3.Exit

Enter your choice:

2

1. In-Order Traversal

2. Post-Order Traversal

3. Pre-Order Traversal

4. Exit

1

Enter Infix Expression:

((a+b)\*d)/c-e

Non-Recursive In-Order Traversal of tree:

a + b \* d / c - e

1. In-Order Traversal

2. Post-Order Traversal

3. Pre-Order Traversal

4. Exit

2

Enter Infix Expression:

((a+b)\*d)/c-e

Non-Recursive Post-Order Traversal of tree:

a b + d \* c / e -

1. In-Order Traversal

2. Post-Order Traversal

3. Pre-Order Traversal

4. Exit

3

Enter Infix Expression:

((a+b)\*d)/c-e

Non-Recursive Pre-Order Traversal of tree:

- / \* + a b d c e

1. In-Order Traversal

2. Post-Order Traversal

3. Pre-Order Traversal

4. Exit

4

Loop exited successfully :)

Choose

1. Recursive Traversal

2. Non-Recursive Traversal

3.Exit

Enter your choice:

3

Application exited successfully :)