Bansilal Ramnath Agarwal Charitable Trust's
Vishwakarma Institute of Information Technology
Department of
Artificial Intelligence and Data Science

Student Name: Siddhesh Dilip Khairnar		
Class: S.Y.	Division: B	Roll No: 272028
Semester: IV	Academic Year: 2022 - 23	
Subject Name & Code: Advanced Data Structures: ADUA22202		
Title of Assignment: Write a program to construct an expression tree from postfix/prefix expression and perform In-order, pre-order and post-order traversals.		
Date of Performance: 1/02/2023	Date of Submission: 10/02/2023	

PAGE NO.:
DATE: / /

Assignment no 2

Name: Siddhesh Dilip Khairnar
Division: B Roll no: 272028
PRN no: 22110398

Aim: Construct expression tree from Postfix/Prefix expression and Perform recursive and non recursive In order, Pre-order and Post order traversals.

Background Information: →

* **Expression tree:** An expression tree is a binary tree that is used to represent expression. The leaf nodes of the expression tree are operand and the other nodes are operator. The structure of the expression tree is determined by the order of the operator and operand in the original expression.

* **Postfix to expression tree:** The postfix expression is converted to an expression tree by traversing the expression from left to right. for each operand, a new nodes is created in the tree and for each operator, two nodes are popped from the stack and a new node is created with the operator as it data and the two popped nodes as it left and right children.

* **Prefix to expression tree:** The prefix expression is converted to an expression tree by traversing the expression from ~~left~~ right to ~~right~~ left. for each operand, a new node is created in the tree and for each operator, two nodes are popped from the stack and a new node is created with the operator as it data and the two popped nodes as its left and right children

PAGE NO. :
DATE : / /

* Expression Tree Algorithm (from Postfix expression):

1. Initialize an empty stack
2. Traverse the postfix expression from left to right
3. If the current character is an operand, create a new node with the operand as it data and push it onto the stack.
4. If the current character is an operator, pop two nodes from the stack, create a new node with the operator as it data, and make the two popped nodes its left and right children.
5. Repeat steps 3 and 4 until the end of the expression is reached
6. The final node remaining on the stack is the root of the expression tree.

* Expression Tree Algorithm (From Prefix Expression):

1. Initialize an ~~empty~~ empty stack
2. Traverse the prefix expression from right to left.
3. If the current character is an operand, create a new node with the operand as it data and push it onto the stack.
4. If the current character is an operator, pop two node from the stack, create a new node with the operator as it data, and make the two popped nodes its left and right children.
5. Repeat Step 3 and 4 until the end of the expression is reached.
6. The final node remaining on the stack is the root of the expression tree

* In-order Traversal (Recursive)

1. If current node is not null:

A. Traverse the left sub tree by calling the in order traversal function recursively on the left child of the current node.

B. Visit the current node and print it data.

C. Traverse the right sub tree by calling the in order traversal function

PAGE NO.:
DATE: / /

on the right child of the current node.

* In Order Traversal (Non-recursive):

1. Initialize a stack and push the root node onto the stack
2. Repeat the following step until the stack is empty.
   A. pop the top node from the stack and visit it left child.
   B. If the left child is not null, push it onto the stack.
   C. If the left child is null, visit the node and it right child
   d. If the right child is not null, push it into stack.

* Pre-Order Traversal (Recursive):

1. If the current node is not null:
   a. visit the current node and print it data
   b. Traverse the left subtree by calling the pre order traversal function recursively on the left child of the ~~of~~ the current node.
   c. Traverse the right subtree by calling the pre-order traversal function recursively on the right child of the current node.

* post Order Traversal (Recursive):

1. If the current node is not null:
   a. Traverse the subtree by calling the post order traversal function recursively on the left child of the current node.
   b. Traverse the right subtree by calling the post order traversal function recursively.

13/02/23

CODE AND OUTPUT

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

class Node
{
public:
    Node *left;
    Node *right;
    char value;
    Node(char value)
    {
        this->value = value;
        this->left = NULL;
        this->right = NULL;
    }
};

bool is_operand(char c)
{
    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
    {
        return true;
    }
    return false;
}

Node *build_tree(std::string expression, bool is_postfix)
{
    std::stack<Node *> stack;
    for (int i = 0; i < expression.length(); i++)
    {
        char c = expression[i];
        if (is_operand(c))
        {
            stack.push(new Node(c));
        }
        else
        {
            Node *node = new Node(c);
            if (is_postfix)
            {
                node->right = stack.top();
                stack.pop();
                node->left = stack.top();
```

```
                stack.pop();
            }
            else
            {
                node->left = stack.top();
                stack.pop();
                node->right = stack.top();
                stack.pop();
            }
            stack.push(node);
        }
    }
    return stack.top();
}

void in_order_traversal(Node *root)
{
    if (root != NULL)
    {
        in_order_traversal(root->left);
        std::cout << root->value << " ";
        in_order_traversal(root->right);
    }
}

void pre_order_traversal(Node *root)
{
    if (root != NULL)
    {
        cout << root->value << " ";
        pre_order_traversal(root->left);
        pre_order_traversal(root->right);
    }
}

void post_order_traversal(Node *root)
{
    if (root != NULL)
    {
        post_order_traversal(root->left);
        post_order_traversal(root->right);
        cout << root->value << " ";
    }
}
int main()
{
    string expression = "ab+ef*g*-";
    Node *root = build_tree(expression, false);
```

```
    cout << "In-order: ";
    in_order_traversal(root);
    cout << endl;
    cout << "Pre-order: ";
    pre_order_traversal(root);
    cout << endl;
    cout << "Post-order: ";
    post_order_traversal(root);
    cout << endl;
    return 0;
}
```

```
PS C:\Program language\C++> cd "c:\Program language\C++\" ; if ($?) { g++ ads2.cpp -o ads2 } ; if ($?)
{ .\ads2 }
In-order: g * f * e - b + a
Pre-order: - * g * f e + b a
Post-order: g f e * * b a + -
PS C:\Program language\C++>
```

Conclusion: Successfully implemented program for expression tree from postfix/prefix expression and perform In-order, pre-order and post-order traversals.