Name	Shubhan Singh
UID no.	2022300118
Experiment No.	7

Program 1	
PROBLEM STATEMENT :	-> *Create an expression tree from a postorder traversal* -> *Write a function to evaluate a given expression tree*
ALGORITHM:	1. Include necessary header files and define structures and functions: • Include a custom stack implementation (**stack.c**) for handling the expression tree. • Define the **isOperator** function to check if a character is an operator. • Define functions for creating expression tree nodes, performing operations, and displaying the tree. 2. Define the **create_node** function: • This function creates a new expression tree node. • It takes two parameters: **op_type** (OPERAND or OPERATOR) and **data** (operand value or operator character). • Allocate memory for the new node using **malloc**. • Initialize the **type** and **data** fields of the node. • Set the left and right child pointers to NULL. • Return the newly created node. 3. Define the **wrapper_create_node** function: • This function takes a character as input and determines the **op_type** and **data** based on whether the character is an operator or operand. • Calls **create_node** with the determined **op_type** and **data** and returns the new node.

4. Define the **create_ET_from_postfix** function:
    - This function creates an expression tree from a postfix expression.
    - Initialize a stack to store intermediate nodes.
    - Start iterating through the postfix expression from right to left.
    - For each character in the expression:
        - If the current node's right child is NULL:
            - If the character is not an operator:
                - Create new nodes for the right and left children.
                - If the left child is an operator, make it the current node.
                - Otherwise, pop a node from the stack and set it as the current node.
            - If the character is an operand and the right child is NULL:
                - Set the character as the right child and push the current node to the stack.
                - Make the right child the new current node.
        - If the right child is not NULL (i.e., left child is NULL):
            - If the character is not an operator:
                - Create a new node for the left child.
                - Pop a node from the stack and set it as the current node.
            - If the character is an operand:
                - Create a new node for the left child and set it as the current node.
        - Display the stack state after each character is processed.
    - Return the root of the expression tree.
5. Define the **evaluate_ET** function:
    - This function recursively evaluates the expression tree.
    - If the current node is a leaf node (both left and right are NULL), return its operand value.
    - Otherwise, get the operator from the current node, and recursively evaluate the left and right subtrees.
    - Use the **perform_operation** function to perform the operation and return the result.
6. In the **main** function:

	• Read the postfix expression from the user. • Create the expression tree using **create_ET_from_postfix**. • Display the inorder traversal of the tree using **displaytree**. • Evaluate the expression tree using **evaluate_ET** and display the result.
PROGRAM:	

```c
/*
 * File: expression_tree_postorder.c
 * Author: Siddhartha Chandra
 * Email: siddhartha_chandra@spit.ac.in
 * Created: October 17, 2023
 * Description: Create an expression tree from a postorder traversal.
 *  Additionally, add function to evaluate a given expression tree
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include "stack.c"
//The program assumes that the given input is a valid postfix expression, error handling has not been done. The
//input string does not contain any spaces, so it hac been assumed that operands are always single digit
//integers.

// Function to check if a character is an operator
bool isOperator(char c) {
```

```
    return (c == '+' || c == '-' || c == '*' || c == '/');//This function
determines whether the given character is
    //an operator or not
}

void displaytree(ExprTreeNode* root){//function for displaying the inorder
traversal of the tree
    if(root==NULL){return;}
    displaytree(root->left);
    if(root->type==OPERAND){
        printf("%.2f ",root->data.operand);
    }
    else{
        printf("%c ",root->data.operation);
    }
    displaytree(root->right);
}

// Function to perform an operation on 2 operands
float perform_operation(char op, float left, float right){//function to return
value of an operation
    switch (op) {
            case '+':
                return left + right;
            case '-':
                return left - right;
```

```
            case '*':
                return left * right;
            case '/':
                if (right != 0) {
                    return left / right;
                } else {
                    fprintf(stderr, "Error: Division by zero\n");
                    exit(EXIT_FAILURE);
                }
            default:
                fprintf(stderr, "Error: Unknown operator %c\n", op);
                exit(EXIT_FAILURE);
        }
}


// TODO: To be completed
ExprTreeNode *create_node(OpType op_type, Data data){//function to create a new node and return it
    ExprTreeNode* newnode=malloc(sizeof(ExprTreeNode));//allocates memory for the node using malloc
    if(op_type==1)//sets the "type" field of the node
        newnode->type=OPERAND;
    else
        newnode->type=OPERATOR;
    newnode->data=data;
```

```
    newnode->left=newnode->right=NULL;//initialises the pointers to the left
and right subtrees to NULL
    return newnode;
}

ExprTreeNode* wrapper_create_node(char c){//The newnode function requires two
parameters, optype and data, this
//function takes a character, determines the value of optype and data and
passes it to the create node function
//to create a new node.
    OpType type;
    Data dat;
    if(isOperator(c)){
        type=OPERATOR;
        dat.operation=c;
    }
    else{
        type=OPERAND;
        dat.operand=(c-'0');
    }
    ExprTreeNode* retval=create_node(type,dat);
    return retval;
}

// TODO: To be completed
// NOTE: Use the stack 'display' in this function to display stack state right
```

```
after a given character in the
//expression has been processed.
// Do this for all characters of the expression string
ExprTreeNode *create_ET_from_postfix(char *postfix_expression){
// This functions iterates through the postfix expression right to left. A
postfix expression is just the
// postorder traversal of the expression tree the expression. thus, the last
element of the postfix expression
// is the node, the one after that the right child and so on, until we
encounter our first operand, since we
// know that operands must be leaves, once we encounter an operand, we insert
the next character as the left
// child. if the left child is an operator, we then set it as our current node
and follow the same steps of
// inserting new elements to the right. whever we encounter a left leaf, we pop
our stack of previously visited
//  nodes and set it as our current node. we push every node into the stack
after assigning it a left child,
//  though nodes with 2 children are not pushed, as they won't be able to
accomodate another left child, so the
//   do not need to be visited once again.

    int index=strlen(postfix_expression)-1;
    Stack* nodestack=initialize_stack(index+1);
    ExprTreeNode* root=wrapper_create_node(postfix_expression[index]);
    index--;
```

```
    ExprTreeNode* current=root;
    while(index>=0){
        if(current->right==NULL){
            if(!isOperator(postfix_expression[index])){//if right child is null
and next charater is not an
            //operator, attach the next 2 elements as right and left children
and if the left child is an
            //operator, make it the current node.
                current->right=wrapper_create_node(postfix_expression[index]);
                current->left=wrapper_create_node(postfix_expression[index-1]);
                index-=2;
                if(isOperator(postfix_expression[index+1])){
                    current=current->left;
                }
                else{
                    //if the left child too is an operand, pop the stack and
make it the current node
                    if(!isEmpty(nodestack)){
                        current=pop(nodestack);
                    }
                    else{
                        return root;
                    }
                }
            }
            else{//if next element is an operand and right child is null,
```

```
attach next element as right child,
            // push the current element,and make the right child the new
current element.
                current->right=wrapper_create_node(postfix_expression[index]);
                push(nodestack,current);
                current=current->right;
                index--;
            }
        }
        else{//if right child is not null, i.e. left child is null(the program
has been written such that cases
        //where both children are null are managed within the if else
statements itself).
            if(!isOperator(postfix_expression[index])){
                current->left=wrapper_create_node(postfix_expression[index]);
                index--;
                if(!isEmpty(nodestack)){
                    current=pop(nodestack);
                }
                else{
                    return root;
                }
            }
            else{
                current->left=wrapper_create_node(postfix_expression[index]);
                index--;
```

```
                current=current->left;
            }
        }
        display(nodestack);
    }

    return root;
}


// TODO: To be completed
float evaluate_ET(ExprTreeNode* root){//this function evaluates the value of
the expression tree, the method used is
// taken from the ppt provided on moodle.
    if(root->left==root->right){
        return root->data.operand;
    }
    else{
        char op=root->data.operation;
        float left=evaluate_ET(root->left);
        float right=evaluate_ET(root->right);
        return perform_operation(op,left,right);
    }
}

int main(){
```

```
    printf("Enter the postfix expression:\n");
    char str[101];
    fgets(str,101,stdin);//reads the string until it encounters a '\n',
including the '\n'
    str[strlen(str)-1]='\0';//replace the '\n' with a '\0', since we do not
need a newline character at the end of
    //our string.
    ExprTreeNode* root=create_ET_from_postfix(str);
    printf("The inrorder traversal of the tree is:\n");
    displaytree(root);
    printf("\nThe evaluation of the tree is:\n%.2f\n",evaluate_ET(root));
}
```

RESULT:

```
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sa
tute Of Technology\C programs\data structures\" ; if ($
Enter the postfix expression:
53+
The inrorder traversal of the tree is:
5.00 + 3.00
The evaluation of the tree is:
8.00
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sa
```

```
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans S
tute Of Technology\C programs\data structures\" ; if (
Enter the postfix expression:
123/-4567***+
|  < + >  | <-- top
-----------

|  < * >  | <-- top
|  < + >  |
-----------

|  < * >  | <-- top
|  < * >  |
|  < + >  |
-----------

Popped element is: *

|  < * >  | <-- top
|  < + >  |
-----------

Popped element is: *

|  < + >  | <-- top
-----------

Popped element is: +

Nothing to display

Nothing to display

|  < - >  | <-- top
-----------

Popped element is: -

Nothing to display

The inrorder traversal of the tree is:
1.00 - 2.00 / 3.00 + 4.00 * 5.00 * 6.00 * 7.00
The evaluation of the tree is:
840.33
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans S
```

THEORY:	Expression trees, also known as parse trees or abstract syntax trees (ASTs), are a fundamental data structure used in computer science and programming to represent mathematical expressions, programming language expressions, and more. They play a crucial role in various applications, including compilers, interpreters, symbolic mathematics, and computer algebra systems. Here are some key points to note about expression trees: 1. **Hierarchical Structure:** Expression trees are hierarchical data structures that represent expressions in a tree-like format. Each node in the tree corresponds to an operator or an operand in the expression. The structure of the tree reflects the order of operations in the expression. 2. **Nodes:** The nodes in an expression tree can be of two types: • Operand Nodes: These represent the values or variables in the expression. Operand nodes have no children and are typically leaf nodes in the tree. • Operator Nodes: These represent the operators (e.g., +, -, *, /) in the expression. Operator nodes have one or more child nodes, which are themselves expressions. 3. **Infix, Prefix, and Postfix Notations:** Expression trees can be constructed from infix, prefix, or postfix notations. Infix notation is the traditional human-readable format, while prefix and postfix notations are more suitable for machine parsing and evaluation. 4. **Order of Operations:** The structure of an expression tree follows the order of operations defined by the expression. This ensures that operations are performed in the correct sequence. For example, in the expression "3 + 4 * 5," the multiplication operation is given higher precedence, so the expression tree reflects this with the multiplication as a parent node and addition as its child. 5. **Evaluation:** Expression trees can be evaluated to compute the result of the expression. Evaluation typically involves recursively traversing the tree, starting from the root and working down to the leaves, performing the operations as specified by the operator nodes. 6. **Parsing and Compilation:** Expression trees are used in the parsing and compilation process of programming languages. When a program is compiled, expressions are converted into expression trees, which are then used to generate machine code or intermediate representations. 7. **Symbolic Mathematics:** In symbolic mathematics and computer algebra systems, expression trees are used to represent complex mathematical expressions with variables, constants, and functions. These trees

	can be manipulated symbolically to simplify, differentiate, or integrate expressions. 8. **Visualization:** Expression trees are often visualized for debugging and educational purposes. They provide a clear and structured representation of the expression, making it easier to understand and analyse complex expressions. 9. **Optimizations:** Expression trees can be optimized to reduce redundancy and improve performance. Common subexpressions can be identified and computed only once to save computational resources. 10. **Error Detection:** Expression trees can be useful for error detection in expressions. They can help identify syntax errors, such as unbalanced parentheses, and semantic errors, like division by zero.
CONCLUSION:	➔ Expression trees are useful in parsing and evaluating expression. ➔ They can also be used to convert between prefix,postfix and infix notations, as these are just preorder, postorder and inorder traversals of the expression tree.