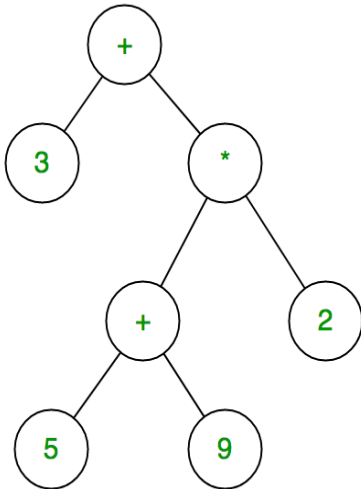


Name	Maureen Miranda
UID no.	2022300060
Experiment No.	7

AIM:	To implement expression trees
-------------	-------------------------------

Program 1

PROBLEM STATEMENT :	-> Create an expression tree from a preorder traversal -> Write a function to evaluate a given expression tree
----------------------------	---

THEORY:	<p>The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for $3 + ((5+9)*2)$ would be:</p> <p>Expression tree</p> <p>Inorder traversal of expression tree produces infix version of given postfix expression (same with postorder traversal it gives postfix expression)</p> <div style="text-align: center;">  </div> <p>For an expression tree we use a stack. We loop through the expression looking for every character.</p> <p>If a character is an operand push that into the stack If a character is an operator pop two values from the stack make them its children</p>
----------------	---

child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

Examples:

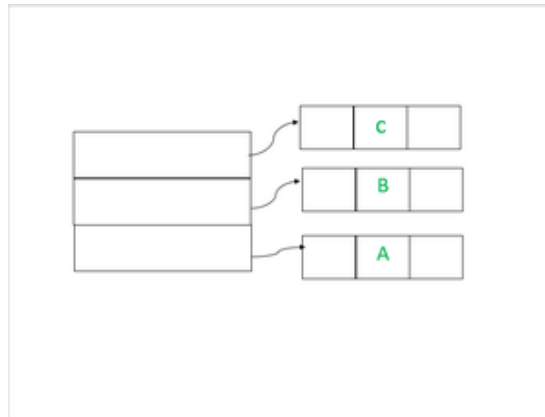
Input: A B C*+ D/

Output: A + B * C / D

The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.

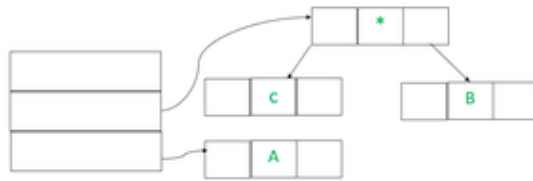
In the Next step, an operator '*' will going read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack

In the Next step, an operator '+' will read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



Similarly, as above cases first we push 'D' into the stack and then in the last step first, will read '/' and then as previous step topmost element will pop out and then will be right subtree of root '/' and other nodes will be right subtree.

Final Constructed Expression Tree is:



Algorithm:

1)Creating Expression Tree Node (create_node function):

- The create_node function takes two parameters: op_type (OPERATOR or OPERAND) and data.
- Allocate memory for a new expression tree node (et) using malloc.
- Set the type field of the newly created node (et) to the provided op_type.
- Set the data field of the node to the provided data.
- Initialize the left and right child pointers of the node to NULL.
- Return the created node.

2)Constructing Expression Tree from Prefix Expression (create_ET_from_prefix function):

- The create_ET_from_prefix function takes a prefix expression as input.
- Determine the length of the prefix expression and initialize a stack data structure to hold expression tree nodes.
- Iterate through the characters of the prefix expression from right to left:
 - For each character, create a new expression tree node (node) with the type set to OPERAND and the operand value initially set to 0.
 - Check if the current character is an operator:

- If it's an operator, change the type of the node to OPERATOR and set the operation field of the node to the current character.
- Pop the top two nodes from the stack and set them as the left and right children of the node.
- If the current character is not an operator (i.e., it's an operand):
 - Set the type of the node to OPERAND.
 - Calculate the operand value by subtracting the character '0' (ASCII value) from the current character and set it as the operand value.
 - Push the node onto the stack.
- After processing each character, call the display function to show the current state of the stack.
- After processing all characters, the stack should contain the root of the expression tree. Pop the root node from the stack and return it as the final expression tree.

3)Preorder Traversal (preorder function):

- The preorder function is a recursive function that performs a preorder traversal of the expression tree.
- It takes the current root node as a parameter.
- If the root node is NULL, return (base case).
- Check the type of the root node:
 - If it's an OPERAND, print the operand value with two decimal places.
 - If it's an OPERATOR, print the operator character.
- Recursively call the preorder function for the left and right children of the root node.

4)Evaluating Expression Tree (evaluate_ET function):

- The evaluate_ET function is a recursive function to evaluate the expression tree.

- It takes the current root node as a parameter.
- If the root node is of type OPERAND:
 - Return the operand value (data.operand).
- If the root node is of type OPERATOR:
 - Recursively evaluate the left and right children using the evaluate_ET function.
 - Perform the corresponding operation (addition, subtraction, multiplication, or division) based on the data.operation field of the root node.
 - Return the result of the operation.

Written solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string.h>
#include "stack.h"

ExprTreeNode* create_node(ExprType op_type, char* data)
{
    ExprTreeNode* et = (ExprTreeNode*) malloc(sizeof(ExprTreeNode));
    et->type = op_type;
    et->data = data;
    et->left = NULL;
    et->right = NULL;
    return et;
}

ExprTreeNode* create_ET_from_prefix(char* prefix_expression)
{
    int length = strlen(prefix_expression);
    Stack* stack = initialise_stack(length);

    for(int i = length-1; i >= 0; i--)
    {
        char c = prefix_expression[i];
        if(isalpha(c))
        {
            char* operand = malloc(sizeof(char)*10);
            strcpy(operand, c);
            ExprTreeNode* node = create_node(OPERAND, operand);
            stack->push(node);
        }
        else
        {
            ExprTreeNode* node = create_node(op_type, c);
            ExprTreeNode* right = stack->pop();
            ExprTreeNode* left = stack->pop();
            node->left = left;
            node->right = right;
            stack->push(node);
        }
    }

    return stack->pop();
}
```

```

if (!isOperator(c))
{
    node->type = OPERATOR;
    node->data.operator = c;
    node->left = pop(stack);
    node->right = pop(stack);
}
else if (!isOperand(c))
{
    node->type = OPERAND;
    node->data.operand = c;
    push(stack, node);
    display(stack);
}
return pop(stack);
}

void * float evaluate_ET (ExprTree * root)
{
    if (root->type == OPERAND)
        return root->data.operand;

    else {
        float left_value = evaluate(root->left);
        float right_value = evaluate(root->right);
        return perform_operation(root->data.operator,
                                left_value, right_value);
    }
}

```

PROGRAM:

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

```

```

typedef union {
    char operation;
    float operand;
} Data;

typedef enum { OPERATOR, OPERAND } OpType;

typedef struct ExprTreeNode{
    OpType type;
    Data data;
    struct ExprTreeNode *left;
    struct ExprTreeNode *right;
} ExprTreeNode;

// complete this function
ExprTreeNode *create_node(OpType op_type, Data data);
// complete this function
ExprTreeNode *create_ET_from_prefix(char* prefix_expression);
// complete this function
float evaluate_ET(ExprTreeNode* exp_tree);

typedef struct Stack
{
    int top;
    unsigned size;
    ExprTreeNode** array;
} Stack;

void display(Stack* stack);
Stack* initialize_stack(unsigned size);
int isEmpty(Stack* stack);
void push(Stack *stack, ExprTreeNode *item);
void peek(Stack* stack);
ExprTreeNode *pop(Stack* stack);

// Build a Stack Abstract Data structure and perform operations on it
// Operations:

```

```

// 1. Push
// 2. Pop
// 3. Peek
// 4. isEmpty
// 5. isFull
// 6. display

// 0 -> display
// Credit: containerized styling => [Manan Kher | SE Comps - A | Batch -
2026]
void display(Stack* stack){
    char displayed_val[10];
    if (stack->top == -1)
    {
        printf("Nothing to display\n\n");
        return;
    }

    for (int i=stack->top; i>=0; i--){
        ExprTreeNode* element = stack->array[i];
        if (element->type == OPERAND)
            sprintf(displayed_val, "%.2f ", element->data.operand);
        else
            sprintf(displayed_val, "< %c >", element->data.operation);

        if (i == stack->top)
            printf("| %s | <-- top\n", displayed_val);
        else
            printf("| %s | \n", displayed_val);
    }
    printf("-----\n\n");
}

// 1 -> Initialize
Stack* initialize_stack(unsigned size){
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->size = size;
    stack->top = -1;
    stack->array = (ExprTreeNode **)malloc(stack->size *

```



```

sizeof(ExprTreeNode *));
    return stack;
}

// 2 -> isFull
int isEmpty(struct Stack* stack){
    return stack->top == -1;
}

// 3 -> isFull
int isFull(Stack* stack){
    return stack->top == stack->size - 1;
}

// 4 -> push
void push(Stack *stack, ExprTreeNode *item){
    if (isFull(stack)){
        printf("Error: Stack is already full!\n");
    }
    else{
        stack->top += 1;
        stack->array[stack->top] = item;
    }
}

// 5 -> peek
void peek(Stack* stack){
    char top_elem[10];
    if (isEmpty(stack)) {
        printf("Error: Stack is empty!\n");
    }
    else{
        ExprTreeNode *element = stack->array[stack->top];
        printf("Top element is: ");
        if (element->type == OPERAND)
            sprintf(top_elem, "%f", element->data.operand);
        else
            sprintf(top_elem, "%c", element->data.operation);
    }
}

```

```

        printf("%s", top_elem);
        printf("\n\n");
    }

}

// 6 -> pop
ExprTreeNode *pop(Stack* stack){
    char top_elem[10];
    if (isEmpty(stack)) {
        printf("Error: Stack is empty!\n");
        return NULL;
    }
    else {
        ExprTreeNode *element = stack->array[stack->top];
        // printf("Popped element is: ");

        //if (element->type == OPERAND)
            //sprintf(top_elem, "%.2f", element->data.operand);
        //else
            //sprintf(top_elem, "%c", element->data.operation);

        // printf("%s", top_elem);
        // printf("\n\n");
        stack->top -= 1;
        return element;
    }
}

// Function to check if a character is an operator
bool isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

// Function to perform an operation on 2 operands
float perform_operation(char op, float left, float right){
    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){
ExprTreeNode *et = (ExprTreeNode *)malloc(sizeof(ExprTreeNode ));
et->type=op_type;
et->data=data;
et->left=NULL;
et->right=NULL;
return et;

}

ExprTreeNode *create_ET_from_prefix(char *prefix_expression)
{
    int length = strlen(prefix_expression);
    Stack *stack = initialize_stack(length);

    for (int i = length - 1; i >= 0; i--)
    {
        char c = prefix_expression[i];

```

```

Data d;
d.operand = 0;
ExprTreeNode *node = create_node(OPERAND, d);

if (isOperator(c))
{
    node->type = OPERATOR;
    node->data.operation = c;
    node->left = pop(stack);
    node->right = pop(stack);
}
else if (!isOperator(c))
{
    node->type = OPERAND;
    node->data.operand = c - '0';
}

push(stack, node);
display(stack);
}
return pop(stack);
}
// 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

void preorder(ExprTreeNode* root)
{
    if (root == NULL)
        return;

    if (root->type == OPERAND)
        printf("%.2f\n", root->data.operand);
    else if (root->type == OPERATOR)
        printf("%c\n", root->data.operation);

    preorder(root->left);
    preorder(root->right);
}

```

	<pre>// TODO: To be completed float evaluate_ET(ExprTreeNode* root){ if (root->type == OPERAND) { return root->data.operand; } else { float left_value = evaluate_ET(root->left); float right_value = evaluate_ET(root->right); return perform_operation(root->data.operation, left_value, right_value); } }</pre>
RESULT:	<p>Input prefix expression: *+2319</p> <p>Expression tree building</p>

```

| 9.00 | <-- top
-----

| 1.00 | <-- top
| 9.00 |
-----

| 3.00 | <-- top
| 1.00 |
| 9.00 |
-----

| 2.00 | <-- top
| 3.00 |
| 1.00 |
| 9.00 |
-----

| < + > | <-- top
| 1.00 |
| 9.00 |
-----

| < - > | <-- top
| 9.00 |
-----

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

```

Preorder traversal of expression tree

```

-----
*
-
+
2.00
3.00
1.00
9.00

```

Evaluation of expression tree

```

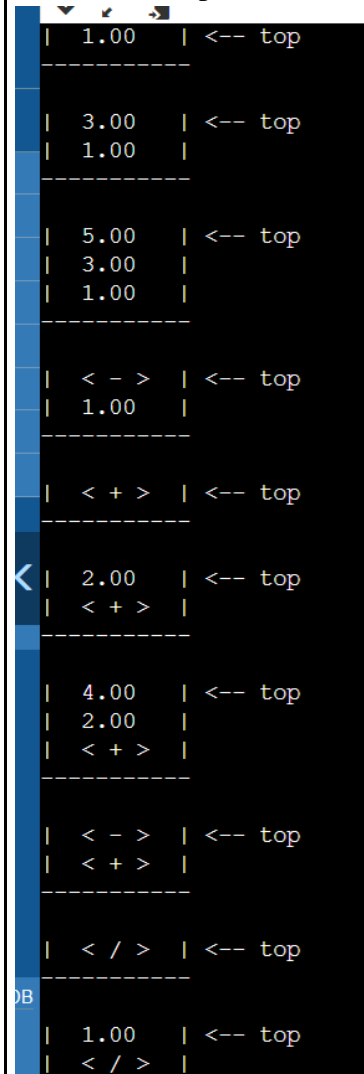
9.00
answer = 36.000000
C++

```

Input expression:

*+2-21/-42+-531

Creation of expression tree:



```

| 1.00 | <-- top
| < / > |
-----

| 2.00 | <-- top
| 1.00 |
| < / > |
-----

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

| 2.00 | <-- top
| < - > |
| < / > |
-----

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

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

```

Prefix traversal of expression tree

```

*
+
< 2.00
-
2.00
1.00
/
-
4.00
2.00
+
-
5.00
3.00
1.00

```

Evaluation of expression tree:

```

answer = 2.000000

```

conclusion

We have successfully implemented expression trees and learnt how to create ,traverse and evaluate an expression tree