NAME :- Manish Shashikant Jadhav
UID :- 2023301005.
BRANCH :- Comps -B. **BRANCH:** B.
EXPERIMENT 7: Implementation of Expression Tree.
SUBJECT :- DS (DATA STRUCTURES).

CODE :-

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

// 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);
    }
}

ExprTreeNode *create_node(OpType op_type, Data data) {
```

```
    ExprTreeNode *treenode = (ExprTreeNode
*)malloc(sizeof(ExprTreeNode));
    treenode->type = op_type;
    treenode->left = NULL;
    treenode->right = NULL;
    treenode->data = data;
    return treenode;
}

ExprTreeNode *create_ET_from_prefix(char *prefix_expression) {
    ExprTreeNode *root = NULL;
    int length = strlen(prefix_expression);
    printf("> %d\n", length);
    Stack *stack = initialize_stack(length);
    Data data;
    for (int i = length - 1; i >= 0; i--) {
        if (isOperator(prefix_expression[i])) {
            data.operation = prefix_expression[i];
            ExprTreeNode *a = pop(stack);
            ExprTreeNode *b = pop(stack);
            ExprTreeNode *c = create_node(OPERATOR, data);
            c->left = a;
            c->right = b;
            push(stack, c);
        } else {
            data.operand = prefix_expression[i] - '0';
            ExprTreeNode *c = create_node(OPERAND, data);
            push(stack, c);
        }
        display(stack);
    }
    root = pop(stack);
    free(stack);
    return root;
}

void InOrderDisplay(ExprTreeNode *root) {
    if (root != NULL) {
        InOrderDisplay(root->left);
        if (isOperator(root->data.operation)) {
```

```
            printf("%c ", root->data.operation);
        } else {
            printf("%.2f ", root->data.operand);
        }
        InOrderDisplay(root->right);
    }
}

void PrefixDisplay(ExprTreeNode *root) {
    if (root != NULL) {
        if (isOperator(root->data.operation)) {
            printf("%c ", root->data.operation);
        } else {
            printf("%.2f ", root->data.operand);
        }
        PrefixDisplay(root->left);
        PrefixDisplay(root->right);
    }
}

float evaluate_ET(ExprTreeNode *root) {
    if (root->left != NULL || root->right != NULL) {
        float ans = 0;
        float left = evaluate_ET(root->left);
        float right = evaluate_ET(root->right);
        char op = root->data.operation;
        if (isOperator(op)) {
            ans = perform_operation(op, left, right);
        }
        return ans;
    } else {
        return root->data.operand;
    }
}

void main() {
    char exp[] = "* + 3 4 2";
    ExprTreeNode *root = create_ET_from_prefix(exp);
    InOrderDisplay(root);
    printf("\n");
```

```
    PrefixDisplay(root);
    printf("\n\n> ANS: %.3f", evaluate_ET(root));
}
```

Output:

```
'--stderr=Microsoft-MIEngine-Error-tiljwmgl.ffw' '--pid=Microsoft-MIEngine-Pid-oxxnz3c2.x1l' '--dbgExe=C:\Program Fi
les (x86)\mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
> 9
|  2.00   | <-- top
-----------

|  -16.00   | <-- top
|  2.00   |
-----------

|  4.00   | <-- top
|  -16.00   |
|  2.00   |
-----------

|  -16.00   | <-- top
|  4.00   |
|  -16.00   |
|  2.00   |
-----------

|  3.00   | <-- top
|  -16.00   |
|  4.00   |
|  -16.00   |
|  2.00   |
-----------

|  -16.00   | <-- top
|  3.00   |
|  -16.00   |
|  4.00   |
|  -16.00   |
|  2.00   |
-----------
```

```
Popped element is: -16.00

Popped element is: 3.00

|  < + >  | <-- top
|  -16.00  |
|  4.00  |
|  -16.00  |
|  2.00  |
-----------

|  -16.00  | <-- top
|  < + >  |
|  -16.00  |
|  4.00  |
|  -16.00  |
|  2.00  |
-----------

Popped element is: -16.00

Popped element is: +

|  < * >  | <-- top
|  -16.00  |
|  4.00  |
|  -16.00  |
|  2.00  |
-----------

Popped element is: *

-16.00 * -16.00 + 3.00
* -16.00 + -16.00 3.00

> ANS: 208.000
PS D:\Manish\DS SPIT>
```

Algorithm:

1.Design the expression tree node structure with type, data that is either an operator or an operamd, and left and right children.

2. Incorporate a function that verifies whether a character is an operator called ‘`isOperator’’. It should return true for ‘+’, ‘-’, ‘*’, and ‘/’.

3. Develop a method that would carry out an operation utilizing two operands called the `perform_operation` method. It requires one operator and two operands, whose results are calculated by this process.

4. Develop a “create_node” function that adds another tree node. It receives a type and data, and initializes the left and right children to NULL, and allocates memory for the node.

5. Write a `create_ET_from_prefix` function that would parse a given prefix expression and generate an expression tree. In this regard, it accepts a prefix expression as an input and yields to the production of the root of the expression tree.

6. Set up an empty stack to hold the elements of the expression tree.

7. Iterate through the characters of the prefix expression from right to left:

a. If the current character is an operator, pop the top two nodes from the stack, create a new node with the operator, set its left and right children to the popped nodes, and push the new node back onto the stack.

b. If the current character is an operand, create a new node with the operand, and push it onto the stack.

8. At the end of the process all the items in the stack will vanish until only the root item remains. The first is pop it and push it as the trunk.

9. Release the space used by the stack.

10. Formulate functions that will show the expression tree under various arrangements like the in-order display (`InOrderDisplay`) and the prefix display (`PrefixDisplay`). The functions must be reusable and recursively traverse the trees printing operators and operands wherever needed.

11. Make an examine function for the expression tree called `evaluate_ET`. It is a recursive evaluation of the tree that is performed on operators and operands with the specified operations and gives the final result.

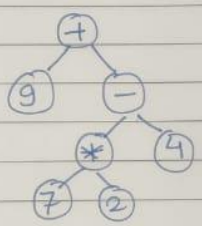
12. In the `main` function:
    a. For example, `char exp[“* + 3 4 2”]`.
    b. Use `create_ET_from_prefix` function to create tree of expressions.
    c. PrefixDisplay and InOrderDisplay for calling the tree using each order.
    d. Finally, call `evaluate_ET` to evaluate the value and finally print the output.

Experiment No. 7

* Aim:- To create a expression tree from a preorder traversal.

* Theory:-

Expression trees are the binary trees that are used to express various expressions like infix, postfix and prefix. Internal nodes of this tree consist of operators like +, -, *, /, ^, ^ and external nodes contains the operands. Operands can be of any datatype like character, integer, double, etc.



1. If character is operand push that into stack.
2. If character is operator pop two values from the stack make them its child and push the current node again.

Conclusion:

Hence, by completing this experiment I came to know about implement an expression tree.