BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai – 400058-India
Department of Computer Engineering

Name	Debjit Ghosal
UID no.	2023300065
Experiment No.	6

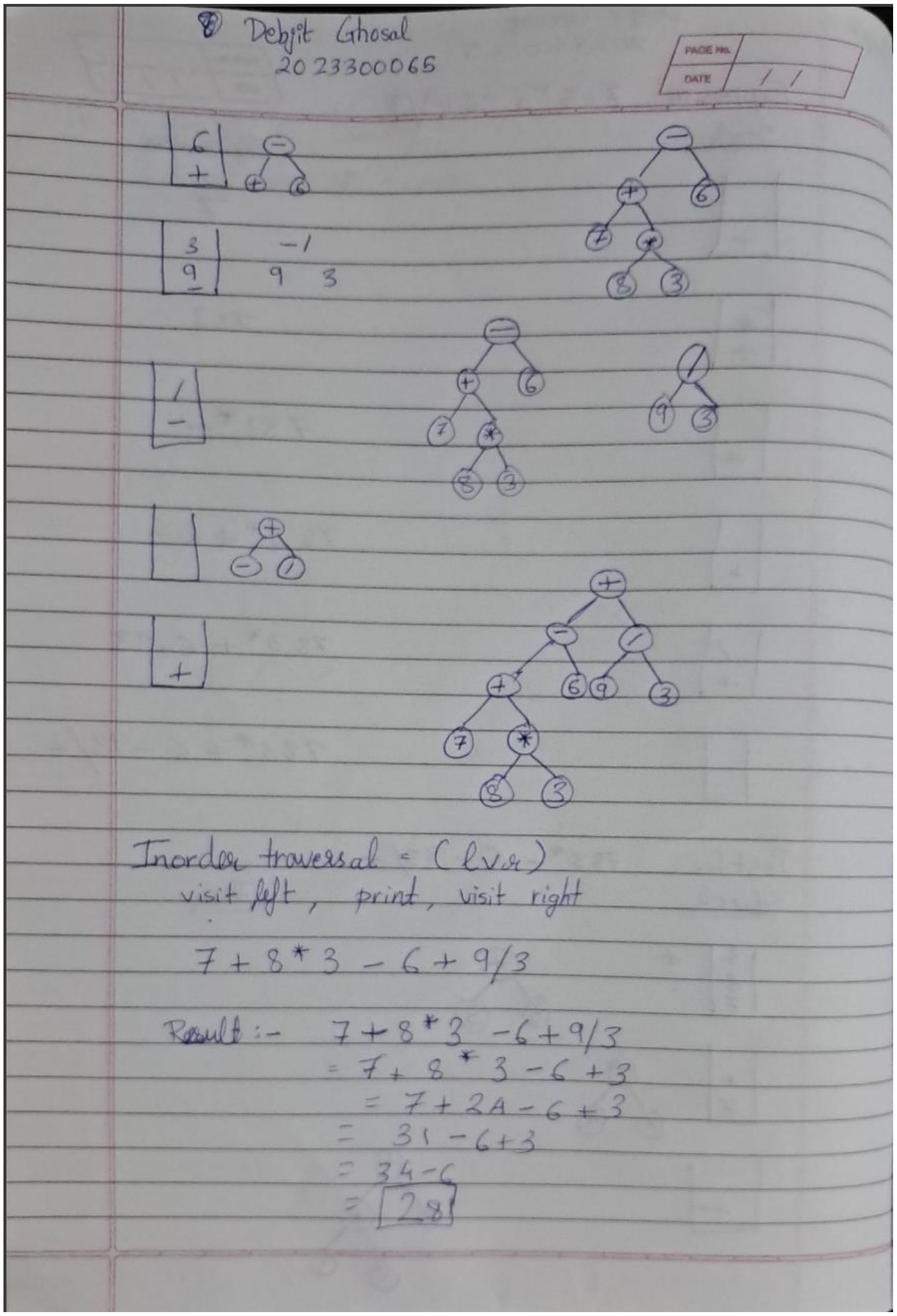
AIM:	**Create an expression tree from a given postorder traversal and perform the evaluation.**
THEORY:	An **Expression Tree** is a binary tree used to represent arithmetic expressions. Each node in the tree can either be an operator (e.g., +, -, *, /) or an operand (e.g., constants or variables like 3, x, y). The leaves of the tree are operands, and the internal nodes are operators. **Structure of an Expression Tree** 1. **Leaf Nodes**: Represent operands, such as constants or variables. For example, 3, 4, x, or y. 2. **Internal Nodes**: Represent operators, such as +, -, *, or /. *Expression trees can be beneficial for:* • Evaluation of the expression. • Generating correct compiler code to compute the expression's value at execution time. • Performing symbolic mathematical operations (such as differentiation) on the expression. Important Properties of an Expression Tree: →***Depth***: The number of levels in the tree affects the complexity of evaluating the expression. → ***Height***: The longest path from the root to a leaf; it determines the balance of the tree. → ***Subtree***: Any tree formed by a node and its descendants is a subtree, which can represent a sub-expression. Expression trees are precious for their ability to represent complex expressions in a structured and hierarchical manner. They offer several advantages. Some of them are: 1. **Simplification of Expressions**: It allows for easy manipulation of expressions. For

	example, an expression can be simplified by performing operations on the tree structure. 2. **Evaluation**: Expression trees can systematically evaluate mathematical expressions by traversing the tree. Each subtree represents a sub-expression that can be computed individually. 3. **Code Generation and Optimization**: Compilers use expression trees to generate optimized code and perform transformations on the code. 4. **Symbolic Computation**: They are used in symbolic computation to differentiate, integrate, or perform algebraic manipulations. 5. **Parsing and Conversion**: It allows easy conversion between infix, prefix, and postfix notations, making them useful for calculators and parsers. Applications of Expression tree are: 1) Compilers and interpreters use expression trees to parse and evaluate expressions in source code. Compilers and interpreters use expression trees to parse and evaluate expressions in source code. 2) Expression trees are used in query optimization to represent and evaluate SQL queries. The database query optimizer uses the expression tree to find an efficient execution plan by reordering operations, minimizing I/O operations, and using indexes effectively. 3) Expression trees are used in software that performs symbolic mathematics, such as computer algebra systems. The tree structure allows for easy manipulation of algebraic expressions, enabling operations like simplification, differentiation, and integration. 4) Expression trees are used to represent and evaluate decision trees and mathematical expressions in symbolic regression. Helps in representing and evolving complex mathematical expressions, which are useful in symbolic AI and machine learning models. 5) Expression trees are used in calculators and parsers to evaluate user-entered mathematical expressions. Ensures the correct order of operations and efficient evaluation of mathematical expressions.
ALGORITHM:	1. **Convert Infix to Postfix**: - Input: An infix expression (e.g., \`(3+5)*2\`). - Output: A postfix expression (e.g., \`35+2*\`). **Algorithm**: - Initialize an empty stack and an empty postfix array.

	- For each character in the infix expression: - If the character is an operand, add it to the postfix array. - If the character is \`(\`, push it onto the stack. - If the character is \`)\`, pop from the stack and add to the postfi array until \`(\` is encountered. - If the character is an operator: - While there is an operator at the top of the stack with higher or equal precedence, pop it and add to the postfix array. - Push the current operator onto the stack. - Pop all remaining operators in the stack and add them to the postfix array. - Return the postfix expression. 2. **Construct Expression Tree**: - Initialize an empty stack. - For each character in the postfix expression: - If the character is an operand: - Create a new node with the operand and push it onto the stack. - If the character is an operator: - Create a new node with the operator. - Pop two nodes from the stack: the first node popped is the right child, and the second node is the left child. - Set the left and right children of the operator node. - Push the operator node back onto the stack. - The remaining node on the stack is the root of the expression tree. 3. **Evaluate Expression Tree**: - If the node is a leaf, return its value. - Recursively evaluate the left and right children. - Operate the current node (\`+\`, \`-\`, \`*\`, or \`/\`) using the values obtained from the left and right children. - Return the result. 4. **Inorder Traversal**: - Recursively traverse the left subtree. - Print the value of the current node. - Recursively traverse the right subtree. **Overall Algorithm**: 1. Read the input infix expression from the user.

	2. Convert the infix expression to postfix using `infixToPostfix()` function. 3. Construct an expression tree from the postfix expression using the constructExpressionTree() function. 4. Display the inorder traversal of the tree using the `inorderTraversal()` function. 5. Evaluate the expression tree using `evaluateExpression()` function and print the result.

PROBLEM-SOLVING ON THE CONCEPT:

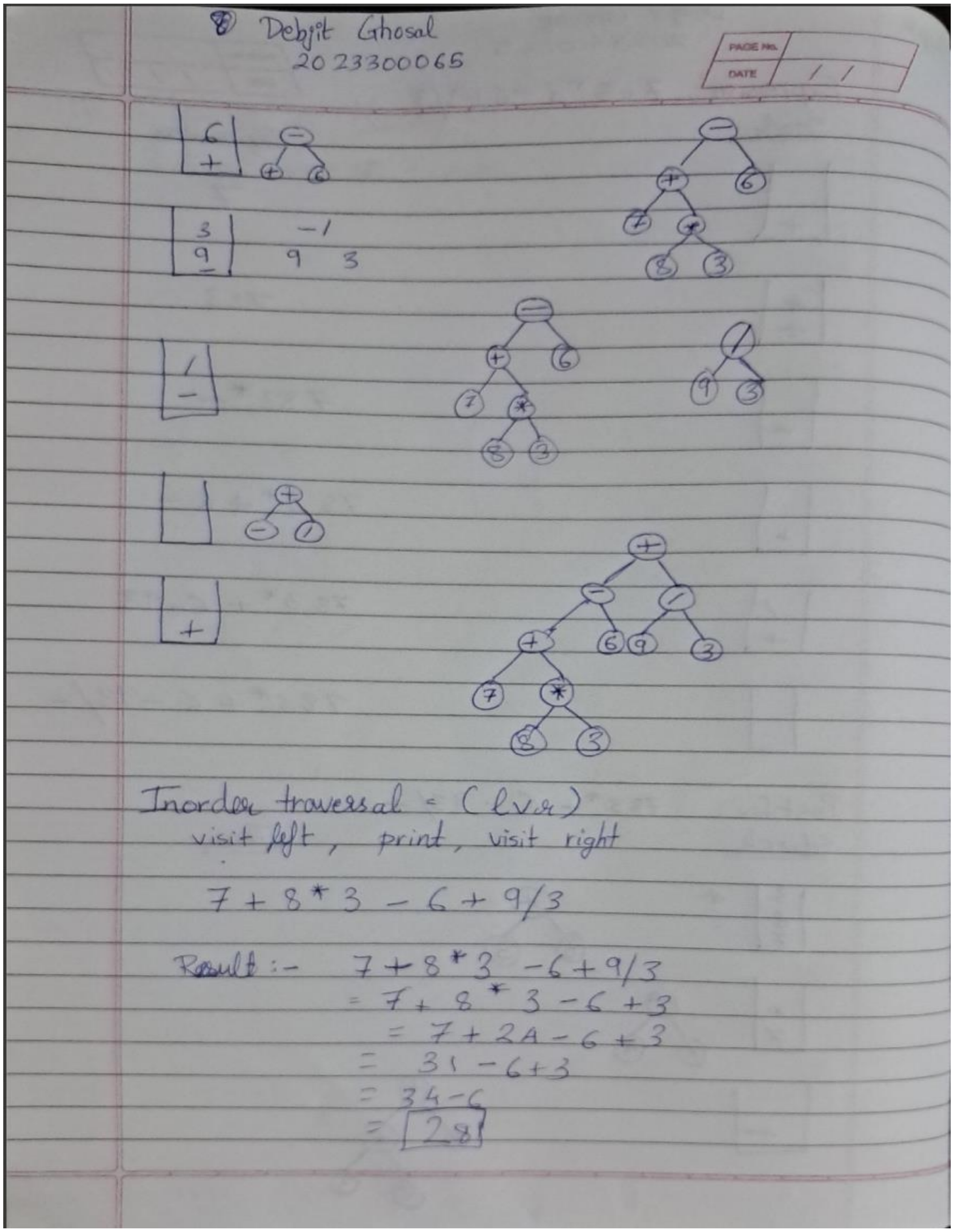

6
+
−
+
6
3
9
−
−
/
9
3
−
+
6
7
*
8
3
/
−
−
+
6
7
*
8
3
/
9
3
+
−
/
+
+
−
6
9
3
7
*
8
3

Debjit Ghosal
2023300065

Inorder traversal = (lvr)
visit left, print, visit right

7 + 8 * 3 − 6 + 9/3

Result :- 7 + 8 * 3 − 6 + 9/3
= 7 + 8 * 3 − 6 + 3
= 7 + 24 − 6 + 3
= 31 − 6 + 3
= 34 − 6
= $\boxed{28}$



8 Debjit Ghosal
2023300065

PAGE NO.
DATE / /

Inorder traversal = (lvr)
visit left, print, visit right

7 + 8 * 3 − 6 + 9/3

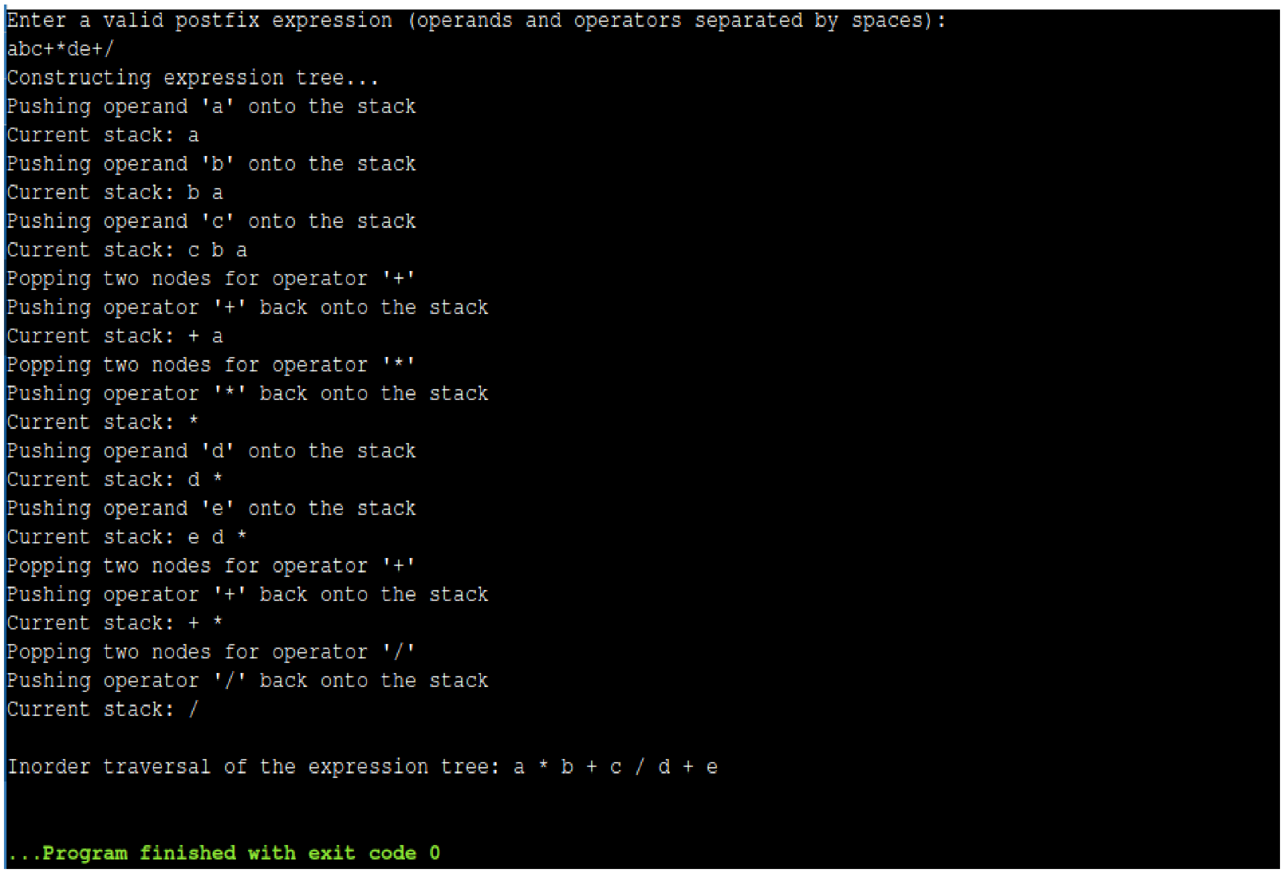
Result :- 7 + 8 * 3 − 6 + 9/3
= 7 + 8 * 3 − 6 + 3
= 7 + 24 − 6 + 3
= 31 − 6 + 3
= 34 − 6
= $\boxed{28}$

CODE:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

struct Node {
    char value;
    struct Node *left, *right, *next;
};

struct Node* createNode(char value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->value = value;
    newNode->left = newNode->right = newNode->next = NULL;
    return newNode;
}

int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

int precedence(char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return 0;
}

void infixToPostfix(char infix[], char postfix[]) {
    char stack[100];
    int top = -1, k = 0;

    for (int i = 0; infix[i] != '\0'; i++) {
        if (isalnum(infix[i])) {
            postfix[k++] = infix[i];
        }
        else if (infix[i] == '(') {
            stack[++top] = infix[i];
        }
        else if (infix[i] == ')') {
            while (top != -1 && stack[top] != '(') {
                postfix[k++] = stack[top--];
            }
            top--;
        }
        else if (isOperator(infix[i])) {
            while (top != -1 && precedence(stack[top]) >= precedence(infix[i])) {
                postfix[k++] = stack[top--];
            }
            stack[++top] = infix[i];
        }
    }
    while (top != -1) {
        postfix[k++] = stack[top--];
    }

    postfix[k] = '\0';
}
struct Node* push(struct Node *stack, struct Node *node) {
    node->next = stack;
    return node;
}
struct Node* pop(struct Node **stack) {
    if (*stack == NULL) return NULL;
    struct Node *temp = *stack;
    *stack = (*stack)->next;
    return temp;
}
void displayStack(struct Node *stack) {
    printf("Current stack: ");
    struct Node *current = stack;
    while (current != NULL) {
        printf("%c ", current->value);
        current = current->next;
    }
    printf("\n");
}
struct Node* constructExpressionTree(char postfix[], int length) {
    struct Node *stack = NULL;

    for (int i = 0; i < length; i++) {
        char symbol = postfix[i];
        if (symbol == ' ') continue;

        struct Node *node = createNode(symbol);

        if (!isOperator(symbol)) {
            printf("Pushing operand '%c' onto the stack\n", symbol);
            stack = push(stack, node);
        } else {
            printf("Popping two nodes for operator '%c'\n", symbol);
            node->right = pop(&stack);
            node->left = pop(&stack);
            printf("Pushing operator '%c' back onto the stack\n", symbol);
            stack = push(stack, node);
        }

        displayStack(stack);
    }
    return stack;
}

int evaluateExpression(struct Node *root) {
    if (!root)
        return 0;

    if (!root->left && !root->right)
        return root->value - '0';

    int leftValue = evaluateExpression(root->left);
    int rightValue = evaluateExpression(root->right);

    switch (root->value) {
        case '+': return leftValue + rightValue;
        case '-': return leftValue - rightValue;
        case '*': return leftValue * rightValue;
        case '/': return leftValue / rightValue;
    }

    return 0;
}

void inorderTraversal(struct Node *root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%c ", root->value);
        inorderTraversal(root->right);
    }
}

int main() {
    char infix[100], postfix[100];

    printf("Enter a valid infix expression:\n");
    fgets(infix, 100, stdin);

    int len = strlen(infix);
    if (infix[len - 1] == '\n') {
        infix[len - 1] = '\0';
    }
    infixToPostfix(infix, postfix);
    printf("Converted Postfix Expression: %s\n", postfix);

    printf("Constructing expression tree...\n");
    struct Node *root = constructExpressionTree(postfix, strlen(postfix));

    printf("\nInorder traversal of the expression tree: ");
    inorderTraversal(root);
    printf("\n");

    int result = evaluateExpression(root);
    printf("The result of the expression is: %d\n", result);

    return 0;
}
```

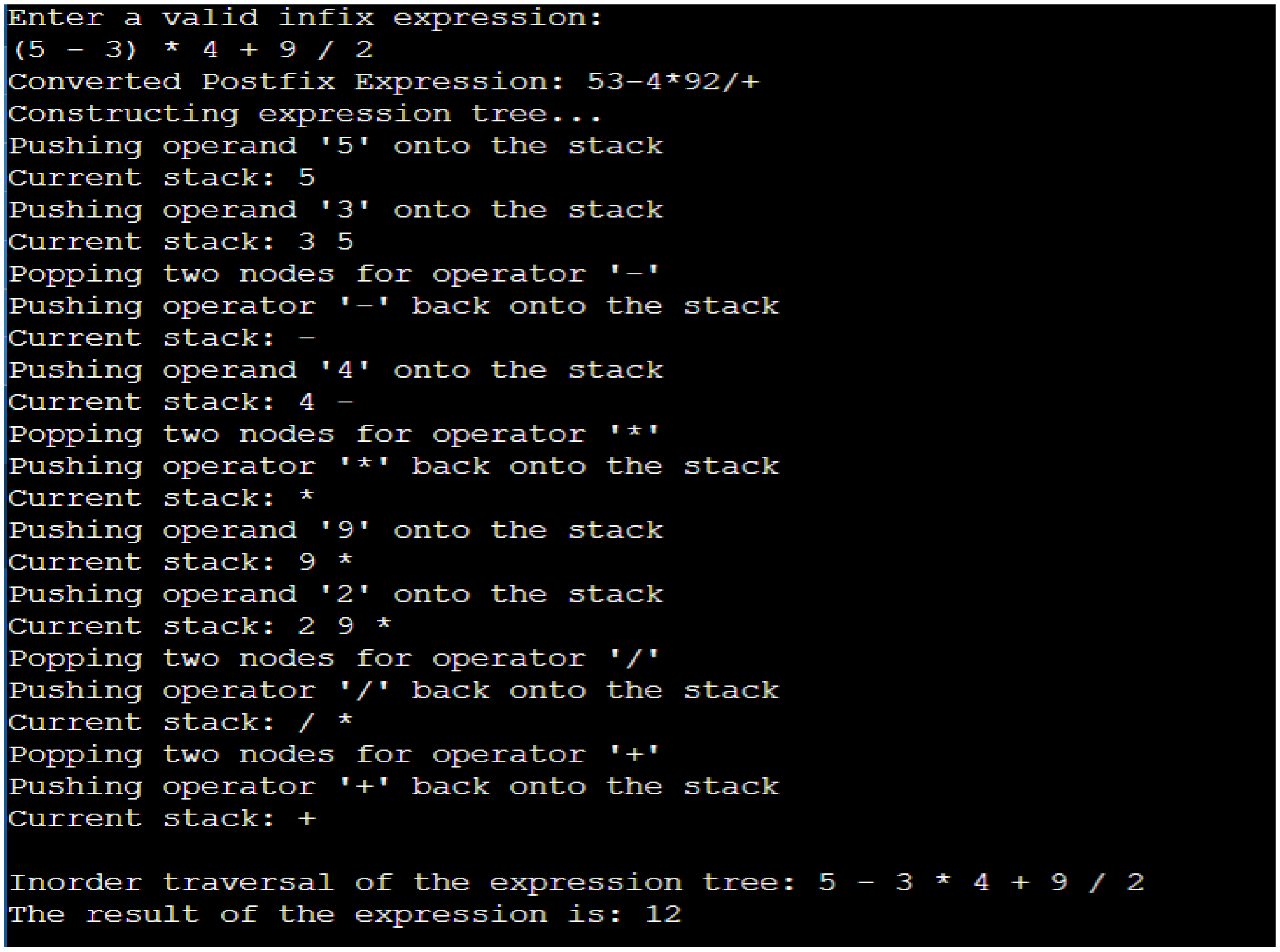
OUTPUT SCREENSHOT:

Output as of Lecture:



```
Enter a valid postfix expression (operands and operators separated by spaces):
abc+*de+/
Constructing expression tree...
Pushing operand 'a' onto the stack
Current stack: a
Pushing operand 'b' onto the stack
Current stack: b a
Pushing operand 'c' onto the stack
Current stack: c b a
Popping two nodes for operator '+'
Pushing operator '+' back onto the stack
Current stack: + a
Popping two nodes for operator '*'
Pushing operator '*' back onto the stack
Current stack: *
Pushing operand 'd' onto the stack
Current stack: d *
Pushing operand 'e' onto the stack
Current stack: e d *
Popping two nodes for operator '+'
Pushing operator '+' back onto the stack
Current stack: + *
Popping two nodes for operator '/'
Pushing operator '/' back onto the stack
Current stack: /

Inorder traversal of the expression tree: a * b + c / d + e


...Program finished with exit code 0
```

Complete Output as told by the instructor for final submission:



```
Enter a valid infix expression:
(5 - 3) * 4 + 9 / 2
Converted Postfix Expression: 53-4*92/+
Constructing expression tree...
Pushing operand '5' onto the stack
Current stack: 5
Pushing operand '3' onto the stack
Current stack: 3 5
Popping two nodes for operator '-'
Pushing operator '-' back onto the stack
Current stack: -
Pushing operand '4' onto the stack
Current stack: 4 -
Popping two nodes for operator '*'
Pushing operator '*' back onto the stack
Current stack: *
Pushing operand '9' onto the stack
Current stack: 9 *
Pushing operand '2' onto the stack
Current stack: 2 9 *
Popping two nodes for operator '/'
Pushing operator '/' back onto the stack
Current stack: / *
Popping two nodes for operator '+'
Pushing operator '+' back onto the stack
Current stack: +

Inorder traversal of the expression tree: 5 - 3 * 4 + 9 / 2
The result of the expression is: 12
```

CONCLUSION:	I have significantly deepened my understanding of data structures, particularly stacks and trees. I learned to convert infix expressions to postfix notation, enhancing my grasp of operator precedence. Implementing a stack using a linked list reinforced the LIFO principle and the importance of dynamic data structures. Constructing an expression tree from the postfix expression clarified relationships between components, making evaluation easier. Evaluating the tree recursively improved my recursion skills, and error handling taught me to manage unexpected input. Overall, after performing this experiment my experience has solidified my foundation in data structures and motivated me to explore more advanced algorithms in the future.