# Exp No 1:

## Title:

Implementation of Stack ADT using Array and Conversion of Infix to Postfix and Evaluation of Postfix Expression

## Aim:

To implement a Stack Abstract Data Type (ADT) using an array and apply it to convert an infix expression to postfix form and evaluate the postfix expression.

## Objective:

• To understand stack operations and their implementation using arrays.  
• To perform expression conversion using stack (Infix → Postfix).  
• To evaluate a postfix expression using stack ADT.

## Theory:

A stack is a linear data structure that follows the LIFO (Last In, First Out) principle. It supports two primary operations:  
• Push(x): Insert an element x onto the stack.  
• Pop(): Remove the top element from the stack.  
  
Applications of Stack include reversing data, expression evaluation, parsing and syntax checking, and function call management.  
  
Infix notation is human-readable (e.g., A + B), while postfix notation (A B +) is easier for machines to evaluate as it avoids precedence and parenthesis ambiguity.

## Algorithm:

Algorithm 1: Stack Implementation using Array

1. Initialize top = -1.  
2. Push(x): If top == MAX-1, print “Stack Overflow”; else increment top and assign stack[top] = x.  
3. Pop(): If top == -1, print “Stack Underflow”; else return stack[top--].

Algorithm 2: Infix to Postfix Conversion

1. Initialize an empty stack.  
2. For each character ch in infix:  
 - If operand → add to postfix.  
 - If ch is ( → push to stack.  
 - If ch is ) → pop until ( is found.  
 - If operator → pop while top has higher or equal precedence, then push ch.  
3. Pop remaining operators to postfix.

Algorithm 3: Postfix Evaluation

1. Initialize an empty stack.  
2. For each token in postfix:  
 - If operand → push onto stack.  
 - If operator → pop top two values, apply operator, push result back.  
3. Return top of stack as result.

**C Program Code**

The program defines two separate array-based stacks: one for characters (used for operators and parentheses in conversion/checking) and one for integers (used for numerical evaluation).

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <math.h> // For pow in case of division/modulus but not strictly needed for basic ops

// --- Configuration ---

#define MAX\_SIZE 200

// --- Stack ADT for Characters (For Infix to Postfix & Parentheses Checker) ---

char char\_stack[MAX\_SIZE];

int char\_top = -1;

// Basic Stack Operations (Character Stack)

int isEmpty\_char() {

return char\_top == -1;

}

int isFull\_char() {

return char\_top == MAX\_SIZE - 1;

}

void push\_char(char x) {

if (isFull\_char()) {

printf("Stack Overflow (Char Stack)\n");

return;

}

char\_stack[++char\_top] = x;

}

char pop\_char() {

if (isEmpty\_char()) {

// In a real application, this would be an error or return a special value.

// For algorithm logic, we rely on prior checks.

return '\0';

}

return char\_stack[char\_top--];

}

char peek\_char() {

if (isEmpty\_char()) {

return '\0';

}

return char\_stack[char\_top];

}

// --- Stack ADT for Integers (For Postfix Evaluation) ---

int int\_stack[MAX\_SIZE];

int int\_top = -1;

// Basic Stack Operations (Integer Stack)

int isEmpty\_int() {

return int\_top == -1;

}

int isFull\_int() {

return int\_top == MAX\_SIZE - 1;

}

void push\_int(int x) {

if (isFull\_int()) {

printf("Stack Overflow (Int Stack)\n");

return;

}

int\_stack[++int\_top] = x;

}

int pop\_int() {

if (isEmpty\_int()) {

printf("Stack Underflow (Int Stack)\n");

return 0; // Error value

}

return int\_stack[int\_top--];

}

int peek\_int() {

if (isEmpty\_int()) {

return 0;

}

return int\_stack[int\_top];

}

// ====================================================================

// --- Task 1: Infix to Postfix Conversion ---

// ====================================================================

// Returns precedence of the operator. Higher number means higher precedence.

int precedence(char op) {

if (op == '+' || op == '-') return 1;

if (op == '\*' || op == '/' || op == '%') return 2;

return 0; // For parentheses and other non-operators

}

void infixToPostfix(const char \*infix, char \*postfix) {

int i, j;

char ch;

// Reset char stack for this operation

char\_top = -1;

i = 0; // infix index

j = 0; // postfix index

while ((ch = infix[i++]) != '\0') {

if (isdigit(ch) || isalpha(ch)) {

// Operand: Add to postfix

postfix[j++] = ch;

} else if (ch == '(') {

// Opening parenthesis: Push to stack

push\_char(ch);

} else if (ch == ')') {

// Closing parenthesis: Pop until '('

while (!isEmpty\_char() && peek\_char() != '(') {

postfix[j++] = pop\_char();

}

if (!isEmpty\_char() && peek\_char() == '(') {

pop\_char(); // Discard '('

}

} else if (ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '%') {

// Operator: Pop while stack has higher or equal precedence operator

while (!isEmpty\_char() && precedence(peek\_char()) >= precedence(ch)) {

postfix[j++] = pop\_char();

}

push\_char(ch); // Push current operator

}

}

// Pop remaining operators

while (!isEmpty\_char()) {

postfix[j++] = pop\_char();

}

postfix[j] = '\0'; // Null-terminate the postfix string

}

// ====================================================================

// --- Task 1: Postfix Evaluation ---

// ====================================================================

int evaluatePostfix(const char \*postfix) {

int i, operand1, operand2, result;

char ch;

// Reset int stack for this operation

int\_top = -1;

for (i = 0; postfix[i] != '\0'; i++) {

ch = postfix[i];

if (isdigit(ch)) {

// Operand: Push the integer value (ASCII '0' is 48)

push\_int(ch - '0');

} else {

// Operator: Pop two operands, perform operation, push result

operand2 = pop\_int();

operand1 = pop\_int();

switch (ch) {

case '+':

result = operand1 + operand2;

break;

case '-':

result = operand1 - operand2;

break;

case '\*':

result = operand1 \* operand2;

break;

case '/':

// Basic integer division

if (operand2 == 0) {

printf("Error: Division by zero\n");

exit(EXIT\_FAILURE);

}

result = operand1 / operand2;

break;

case '%':

result = operand1 % operand2;

break;

default:

printf("Error: Invalid character in postfix expression\n");

exit(EXIT\_FAILURE);

}

push\_int(result);

}

}

// The result is the final value left on the stack

if (!isEmpty\_int()) {

return pop\_int();

}

return 0; // Should not happen for a valid expression

}

// ====================================================================

// --- Task 2: Post-Lab Problem: Balanced Parentheses Checker ---

// ====================================================================

// Utility function to check if the top of the stack matches the closing delimiter

int isMatchingPair(char char1, char char2) {

if (char1 == '(' && char2 == ')') return 1;

if (char1 == '{' && char2 == '}') return 1;

if (char1 == '[' && char2 == ']') return 1;

return 0;

}

// Main checker function

void isBalanced(const char \*exp) {

int i;

char ch;

// Reset char stack

char\_top = -1;

for (i = 0; exp[i] != '\0'; i++) {

ch = exp[i];

if (ch == '(' || ch == '{' || ch == '[') {

// Opening symbol: Push onto stack

push\_char(ch);

} else if (ch == ')' || ch == '}' || ch == ']') {

// Closing symbol

if (isEmpty\_char()) {

printf("Not Balanced (Unexpected closing delimiter '%c' at index %d)\n", ch, i);

return;

}

if (isMatchingPair(peek\_char(), ch)) {

pop\_char(); // Matched, so remove the opener

} else {

printf("Not Balanced (Mismatch: Expected '%c' but found '%c' at index %d)\n", peek\_char(), ch, i);

return;

}

}

// Ignore all other characters (operands, operators, etc.)

}

// Final Check: Stack must be empty if all pairs were matched

if (isEmpty\_char()) {

printf("Balanced\n");

} else {

printf("Not Balanced (Unmatched opening delimiter(s) remaining)\n");

}

}

// ====================================================================

// --- Main Driver Function ---

// ====================================================================

int main() {

// --- PART 1: Infix to Postfix and Evaluation ---

char infix1[] = "(3+5)\*(2+6)/4";

char postfix1[MAX\_SIZE];

int result1;

printf("--- Infix to Postfix Conversion and Evaluation ---\n");

printf("Input: %s\n", infix1);

infixToPostfix(infix1, postfix1);

printf("Postfix Expression: %s\n", postfix1);

result1 = evaluatePostfix(postfix1);

printf("Evaluated Result = %d\n\n", result1);

// --- PART 2: Post-Lab Problem: Balanced Parentheses Checker ---

printf("--- Post-Lab Problem: Balanced Parentheses Checker ---\n");

// Sample 1: Balanced

char exp1[] = "(a+b) \* {c + [d/(e-f)]}";

printf("Input: %s -> ", exp1);

isBalanced(exp1);

// Sample 2: Not Balanced (Mismatched closing)

char exp2[] = "([a+b] \* {c+d))";

printf("Input: %s -> ", exp2);

isBalanced(exp2);

// Sample 3: Not Balanced (Order violation)

char exp3[] = "([)]";

printf("Input: %s -> ", exp3);

isBalanced(exp3);

// Sample 4: Not Balanced (Unmatched opening)

char exp4[] = "{(a+b)";

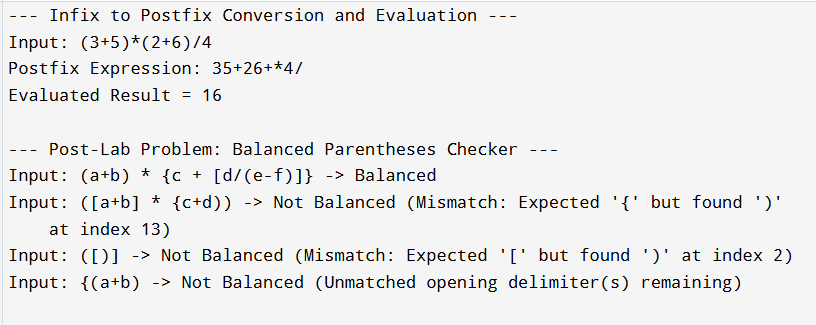
printf("Input: %s -> ", exp4);

isBalanced(exp4);

return 0;

}

**Input/Output:**

****

**Result:**

Successfully implemented the Stack ADT using an array with separate stacks for characters and integers. The implementation was successfully applied to:

Convert the infix expression (3+5)\*(2+6)/4 to its postfix form 35+26+\*4/.

Implement a isBalanced function to check for balanced parentheses/brackets/braces in an expression string.

**Conclusion:**

The experiment successfully demonstrated the utility of the Stack data structure, following the LIFO principle, for complex algorithmic tasks. The array-based implementation is efficient for managing the temporary storage required for operator precedence and parenthesis handling in Infix to Postfix conversion. Furthermore, the stack simplifies the sequential operation execution in Postfix evaluation and is the fundamental mechanism for ensuring Balanced Delimiters by matching opening and closing symbols in the correct order. These applications are central to compiler design and expression parsing.\*\*. Furthermore, the stack simplifies the sequential operation execution in Postfix evaluation and is the fundamental mechanism for ensuring Balanced Delimiters by matching opening and closing symbols in the correct order. These applications are central to compiler design and expression parsing.

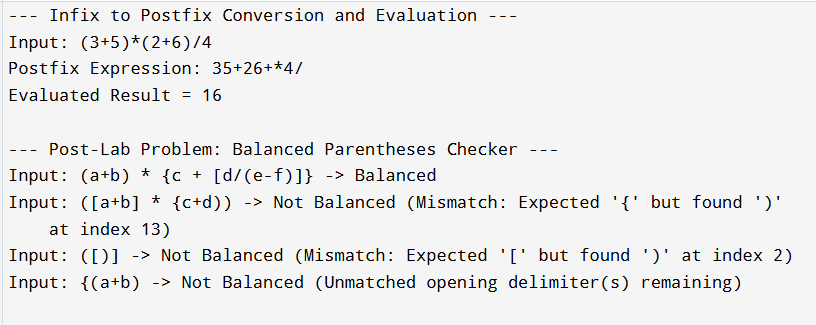
# Post-Lab Problem: Balanced Parentheses Checker

**Objective:**  
Using your Stack ADT (array implementation), write a program that checks whether an expression has balanced parentheses/brackets/braces.

**Description:**  
Given a string containing characters ( ) { } [ ] along with operands/operators, determine if every opening symbol has a corresponding closing symbol in the correct order.

**Input:**  
A single line string (e.g., "(a+b) \* {c + [d/(e-f)]}").

**Output:**  
Print "Balanced" if the delimiters are balanced; otherwise print "Not Balanced" (optionally print the index of first mismatch).

****

**Constraints:**

* Use your **array-based Stack ADT** only (no STL/collections).
* Ignore non-delimiter characters.
* Max expression length: 200.

**Sample I/O:**

* Input: (a+b) \* {c + [d/(e-f)]} → Output: Balanced
* Input: ([a+b] \* {c+d)) → Output: Not Balanced
* Input: ([)] → Output: Not Balanced

**Hints:**

* Push opening symbols onto the stack.
* On encountering a closing symbol, check: stack not empty **and** top matches the corresponding opener.
* At the end, stack must be empty.