# 

# EXPERIMENT NO: 2

**Date of Performance: Date of Submission :**

**AIM**: Convert an Infix expression to Postfix expression using stack ADT.

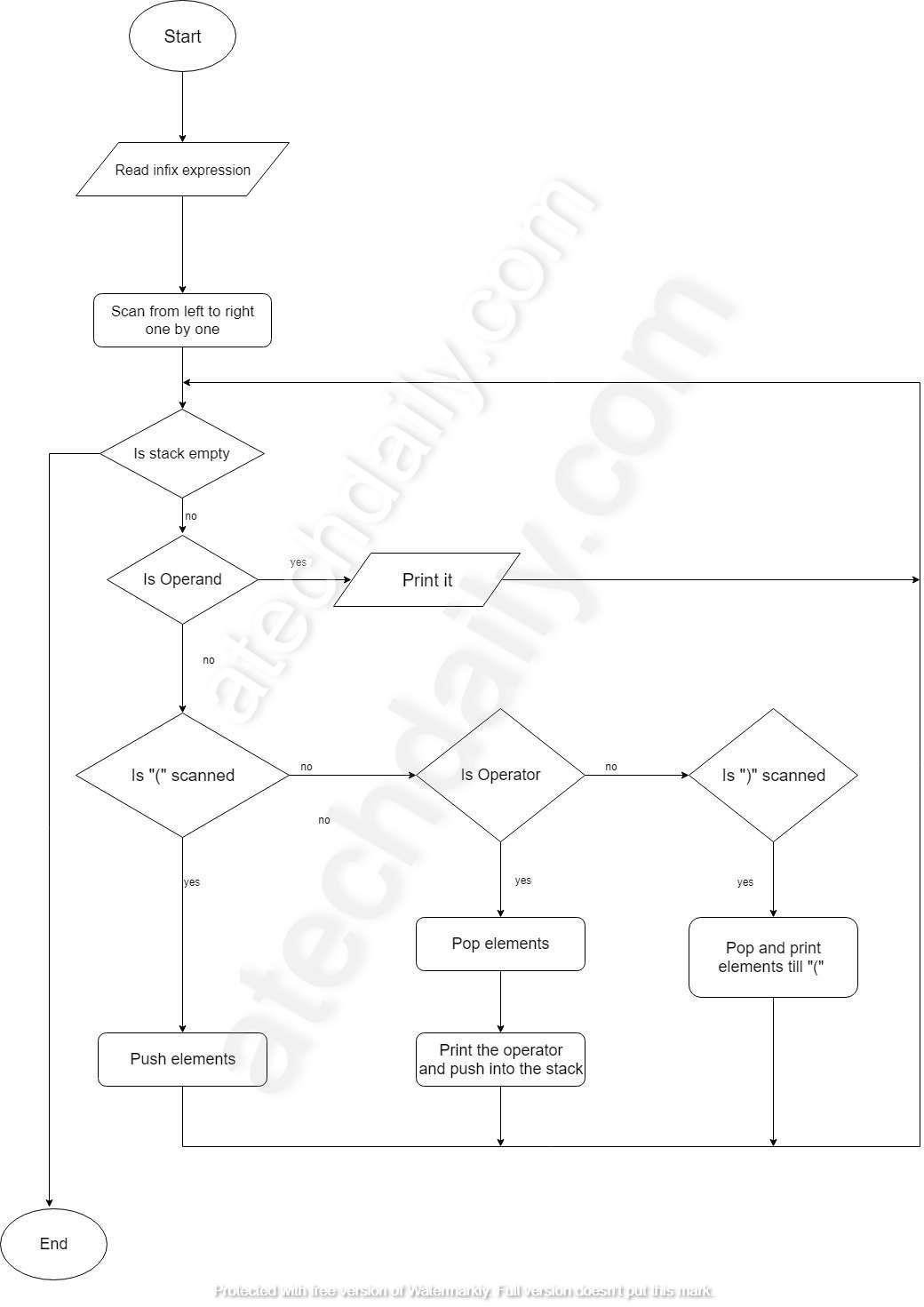
# THEORY:

In high level programming languages, we use arithmetic expression in its infix form. An expression in infix form contains operators in between operands on which it operates. Parentheses also appear in infix expressions to specify the order of evaluation. During compilation, the compiler converts the infix expression to postfix for easy evaluation, since a postfix expression does not contain any parenthesis. Also, a postfix expression can be evaluated easily by using a stack. Infix expressions are readable and solvable by humans because of easily distinguishable order of operators, but compiler doesn’t have integrated order of operators.

Postfix notation has the following virtues:

* No parenthesis.
* The priority of the operations is no longer relevant.
* Enables easy evaluation (evaluated by making a left to right scan, stacking the operands.) An infix expression can be manually converted to its post-fix form by following these steps:
* Step1: Fully parenthesis the expression.
* Step2: Move all operators so that they replace their corresponding right parenthesis.
* Step3: Delete all parentheses.

# FLOWCHART :



**ALGORITHMS**:

|  |  |  |
| --- | --- | --- |
| Step 1 | : | Add “)”to the end of the infix expression |
| Step 2 | : | Push “(“on to the stack |
| Step 3 | : | Repeat until each character in the infix notation is scanned IF a “(“ is encountered, push it on the stack  IF an operand (whether a digit or a character) is encountered, add it postfix expression.  IF a)is encountered, then   1. Repeatedly pop from stack and add it to the postfix expression until a “(“ is encountered. 2. Discard the “(“ . That is, remove the “(“ from stack and do not add it to the postfix expression   IF an operator is encountered, then   1. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than 2. Push the operator to the stack   [END OF IF] |
| Step 4 | : | Repeatedly pop from the stack and add it to the postfix expression until the  stack is empty |
| Step 5 | : | EXIT |

# PROGRAM:

Write a program to convert infix expression to postfix expression using stack.

# SOURCE CODE:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 100

int precedence(char operator) {

if (operator == '+' || operator == '-')

return 1;

if (operator == '\*' || operator == '/')

return 2;

return 0;

}

void infixToPostfix(const char\* infix) {

char postfix[MAX\_SIZE];

char stack[MAX\_SIZE];

int top = -1;

int j = 0;

for (int i = 0; infix[i]; i++) {

char c = infix[i];

if (isalnum(c)) {

postfix[j++] = c;

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

stack[++top] = c;

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

while (top >= 0 && stack[top] != '(') {

postfix[j++] = stack[top--];

}

if (top >= 0 && stack[top] == '(') {

top--; // Pop the '(' from the stack

}

} else {

while (top >= 0 && precedence(c) <= precedence(stack[top])) {

postfix[j++] = stack[top--];

}

stack[++top] = c;

}

}

while (top >= 0) {

postfix[j++] = stack[top--];

}

postfix[j] = '\0';

printf("Postfix expression: %s\n", postfix);

}

int main() {

printf("66 Sumit Kolhe \n");

char infix[MAX\_SIZE];

printf("Enter an infix expression: ");

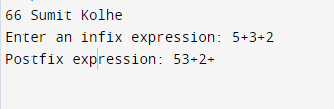
scanf("%s", infix);

infixToPostfix(infix);

return 0;

}

**OUTPUT :**



# Conclusion / Outcome: The experiment aimed to understand and implement the conversion of an infix expression to a postfix expression using the Stack Abstract Data Type (ADT). The process involved scanning an infix expression from left to right, and based on certain rules, generating the equivalent postfix expression. The stack was utilized to store operators temporarily and ensure the correct order of operations.

# Marks &Signature:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **R1 (4 Marks)** | **R2 (4 Marks)** | **R3 (4 Marks)** | **R4 (3 Marks)** | **Total**  **(15 Marks)** | **Signature** |
|  |  |  |  |  |  |

**EXPERIMENT NO: 3**

**Date of Performance: Date of Submission:**

**AIM**: To Evaluate Postfix Expression using Stack ADT.

# THEORY:

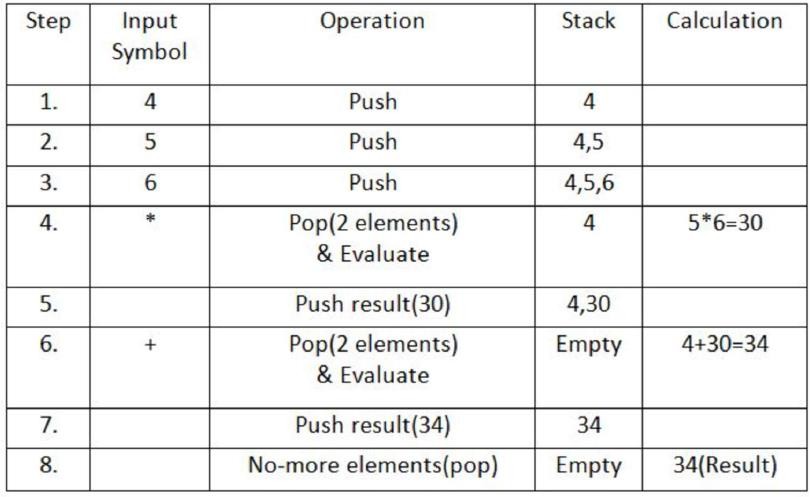
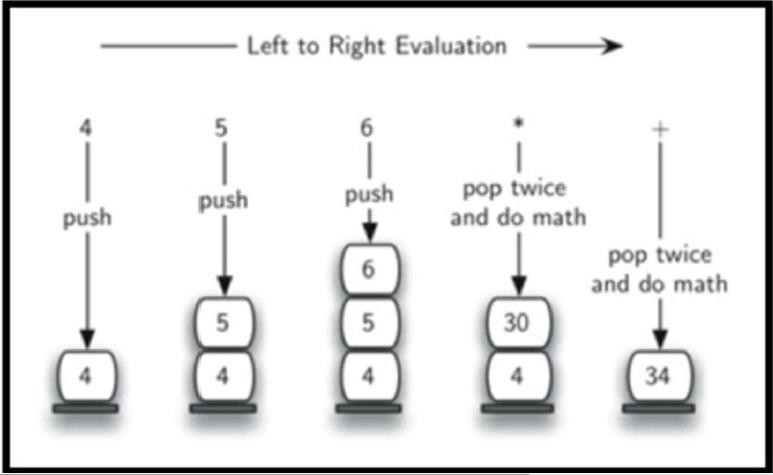
An infix expression in a High Level Language program is converted into its postfix form on its compilation time, since the evaluation of a postfix expression is much simpler than direct evaluation of an infix expression. The postfix expression is evaluated using Stack. Following is the method for evaluation postfix expressions:

1. Create a stack to store operands.
2. Scan the given expression and do following for every scanned element.
   1. If the element is a number, push it into the stack.
   2. If the element is an operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack.
3. When the expression is ended, the number in the stack is the final answer.

# ALGORITHM:

1. Add ) to postfix expression.
2. Read postfix expression Left to Right until ) encountered
3. If operand is encountered, push it onto Stack [End If]
4. If operator is encountered, Pop two elements
   1. A -> Top element
   2. B-> Next to Top element
   3. Evaluate B operator A
   4. Push B operator A onto Stack
5. Set result = pop
6. END

# EXAMPLE:- Expression: 456\*+



|  |
| --- |
|  |
|  |

**PROGRAM**:

Write a program to evaluate postfix expression using stack ADT.

**Source Code**:

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

// Structure for the stack

struct Stack {

int top;

int\* array;

};

// Function to create a new stack

struct Stack\* createStack(int capacity) {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->top = -1;

stack->array = (int\*)malloc(capacity \* sizeof(int));

return stack;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to push an element onto the stack

void push(struct Stack\* stack, int item) {

stack->array[++stack->top] = item;

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (!isEmpty(stack))

return stack->array[stack->top--];

return -1; // Return -1 to indicate an empty stack (error)

}

// Function to evaluate a postfix expression

int evaluatePostfix(char\* postfix) {

struct Stack\* stack = createStack(strlen(postfix));

int i;

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

char c = postfix[i];

if (isdigit(c)) {

push(stack, c - '0'); // Convert char to integer and push to stack

} else {

int operand2 = pop(stack);

int operand1 = pop(stack);

switch (c) {

case '+':

push(stack, operand1 + operand2);

break;

case '-':

push(stack, operand1 - operand2);

break;

case '\*':

push(stack, operand1 \* operand2);

break;

case '/':

push(stack, operand1 / operand2);

break;

}

}

}

return pop(stack);

}

int main() {

char postfix[100];

printf("66 Sumit Kolhe \n");

printf("Enter a postfix expression: ");

scanf("%s", postfix);

int result = evaluatePostfix(postfix);

printf("Result: %d\n", result);

return 0;

}

**OUTPUT**:

# C:\Users\AIDS01\Documents\Lightshot\Screenshot_2.png

# 

**Conclusion / Outcome**: The experiment focused on implementing a program to evaluate postfix expressions using the Stack Abstract Data Type (ADT). The goal was to comprehend the mechanics of postfix expression evaluation and the integral role of stacks in the process.

**Marks &Signature:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **R1 (4 Marks)** | **R2 (4 Marks)** | **R3 (4 Marks)** | **R4 (3 Marks)** | **Total**  **(15 Marks)** | **Signature** |
|  |  |  |  |  |  |