Aim: Convert an Infix expression to Postfix expression using stack ADT.

Theory:

Infix expressions are those where operators are placed between operands (e.g., A + B). While these expressions are intuitive for human readers, they can be ambiguous for machines due to the precedence of operators and the need for parentheses to dictate order of operations. Postfix notation (or Reverse Polish Notation) eliminates this ambiguity by placing operators after their operands (e.g., AB+).

Procedure:

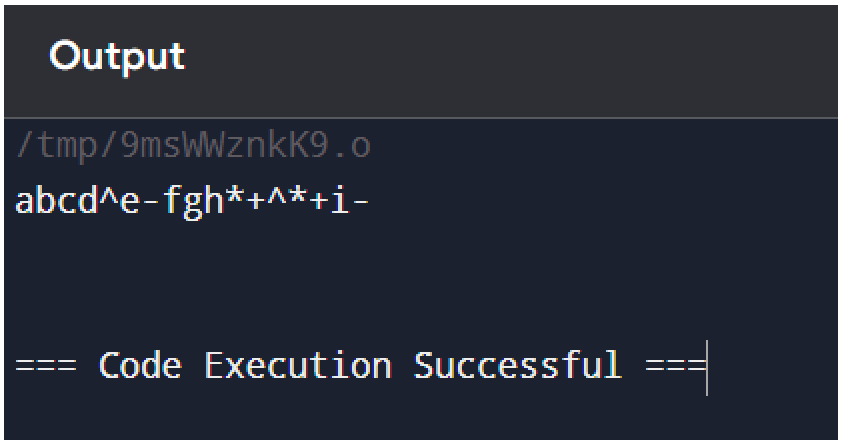
1. **Initialize**: Create an empty stack and an empty output string for the postfix expression.
2. **Scan the Infix Expression**:
    - Read the infix expression from left to right.
3. **Handle Operands**:
    - If the scanned character is an operand (e.g., a variable like A, B, C), add it directly to the output string.
4. **Handle Operators**:
    - If the scanned character is an operator:
        - While there is an operator at the top of the stack with greater or equal precedence, pop operators from the stack to the output.
        - Push the current operator onto the stack.
5. **Handle Parentheses**:
    - If the scanned character is a left parenthesis `(`, push it onto the stack.
    - If it is a right parenthesis `)`, pop from the stack to the output until a left parenthesis is at the top of the stack. Discard both parentheses.
6. **End of Expression**:
    - After scanning all characters, pop any remaining operators from the stack to the output.
7. **Output Result**: The output string now contains the postfix expression.

Code:

```
#include <stdio.h>
 #include <stdlib.h>
 #include <ctype.h>
 #include <string.h>
int prec(char c) {
if (c == '^')
 return 3;
 else if (c == '/' || c == '*')
 return 2;
 else if (c == '+' || c == '-')
 return 1;
 else
 return-1;
 }
 // Function to perform infix to postfix conversion
 void infixToPostfix(char* exp) {
 int len = strlen(exp);
 char result[len + 1];
 char stack[len];
 int j = 0;
int top =-1;
 for (int i = 0; i < len; i++) {
 char c = exp[i];
 // If the scanned character is
 // an operand, add it to the output string.
 if (isalnum(c))
 result[j++] = c;
 // If the scanned character is
 // an ‘(‘, push it to the stack.
 else if (c == '(')
 stack[++top] = '(';
 // If the scanned character is an ‘)’,
// pop and add to the output string from the stack
 // until an ‘(‘ is encountered.
 else if (c == ')') {
 while (top !=-1 && stack[top] != '(') {
 result[j++] = stack[top--];
 }
 top--;
 }
 // If an operator is scanned
 else {
 while (top !=-1 && (prec(c) < prec(stack[top]) ||
 prec(c) == prec(stack[top]))) {
 result[j++] = stack[top--];
 }
 stack[++top] = c;
}
 }
 // Pop all the remaining elements from the stack
 while (top !=-1) {
```

```
result[j++] = stack[top--];
}
result[j] = '\0';
printf("%s\n", result);
}
int main() {
char exp[] = "a+b*(c^d-e)^(f+g*h)-i";
infixToPostfix(exp);
return 0;
}
```

Output:



Output

```
/tmp/9msWWznkK9.o
abcd^e-fgh*+^*+i-


=== Code Execution Successful ===
```

Conclusion: This experiment explores the conversion of infix expressions to postfix expressions using a stack ADT. It reveals how stacks are essential for parsing and evaluating mathematical expressions, providing a foundation for compiler design and interpreters.