**EXPERIMENT NO. 1**

| **Objective(s):**  To implementing postfix evaluation using a stack is to evaluate arithmetic expressions written in postfix notation efficiently. |
| --- |
| **Outcome:**  Utilize a stack data structure to manage operands and intermediate results during evaluation. |
| **Problem Statement:**  Implement Postfix evaluation using Stack. |
| **Background Study:**   1. **Postfix Notation (Reverse Polish Notation, RPN)**   Postfix notation is a way of writing arithmetic expressions in which every operator follows all of its operands. This eliminates the need for parentheses to denote the order of operations (operator precedence). For example:   * **Infix Notation**: (3 + 4) \* 5 * **Postfix Notation**: 3 4 + 5 \*  1. **Why Use Postfix Notation?**   Postfix notation is easier to evaluate with a computer program because:   * It removes the need for parentheses and rules about operator precedence. * It is unambiguous and easier to parse. * Evaluation can be done using a stack-based algorithm, which is efficient and straightforward.  1. **Stack Data Structure**   A stack is a Last In First Out (LIFO) data structure, meaning that the last element added to the stack is the first one to be removed. This makes it particularly useful for evaluating postfix expressions, where operands are pushed onto the stack as they appear, and operators cause the operands to be popped, evaluated, and then their result pushed back onto the stack.   1. **Evaluation Process**   The evaluation process for postfix expressions involves:   * Iterating through each token (operand or operator) in the expression. * Pushing operands onto the stack. * When encountering an operator, popping the necessary number of operands from the stack, performing the operation, and pushing the result back onto the stack. * At the end of the expression, the stack should contain exactly one operand, which is the result of the expression.  1. **Operators and Operands**   In postfix notation, operands are numbers, and operators are symbols that represent arithmetic operations (e.g.,+, -, \*, /). Each operator takes a certain number of operands (e.g., + and - take two operands, while \* and / take  two operands).   1. **Error Handling**   It's essential to handle errors such as:   * Invalid tokens in the input expression. * Insufficient operands for an operator. * Extra operands left after evaluating the expression |

| **Algorithm (Student Work Area):**  **Algorithm for Infix to Postfix Conversion:**  Input: Infix expression infix[] Output: Postfix expression postfix[]   1. Initialize an empty stack stack[] and an empty string postfix[]. 2. Iterate through each character ch in the infix expression infix[]:  * If ch is an operand (number or variable), append it to postfix[]. * If ch is an opening parenthesis '(', push it onto stack[]. * If ch is a closing parenthesis ')':   + **Pop from stack**[] to postfix[] until an opening parenthesis '(' is encountered.   + **Pop the opening parenthesis** from stack[]. * If ch is an operator:   + **Pop from stack[] to postfix[]** until an operator with lower or equal precedence than ch is at the top of stack[].   + **Push ch onto stack[]**.  1. Pop all operators from stack[] to postfix[]. 2. Return postfix[] as the result.   **Algorithm for Evaluating a Postfix Expression**:  Input: Postfix expression postfix[] Output: Integer result of the expression   1. **Initialize an empty stack** stack[]. 2. **Iterate through each character** ch in the postfix expression postfix[]:  * If ch is an operand (number), **push its integer value onto stack[].** * If ch is an operator:   + **Pop the top two elements b and a from stack**[].   + Compute result as a ch b using the operator ch.   + **Push result onto stack**[].  1. **The result of the expression** is the only element left in stack[]. 2. **Return the result** |
| --- |

| **Code:** |
| --- |
| **OUTPUT :** |