**EXPERIMENT NO. 2**

| **Objective(s):**  To Implementing Infix to Postfix Expression Conversion using Stack |
| --- |
| **Outcome:**  A postfix expression obtained from converting an infix expression using a stack-based algorithm. |
| **Problem Statement:**  Implement Infix to Postfix Expression Conversion using Stack |
| **Background Study:**  Converting an infix expression to a postfix (or Reverse Polish Notation, RPN) expression involves rearranging the expression so that each operator follows all of its operands. This is useful for evaluating mathematical expressions efficiently.   1. **Infix Notation**: This is the conventional way of writing mathematical expressions, where operators are placed between operands. For example, a + b or a \* (b + c). 2. **Postfix Notation**: Also known as Reverse Polish Notation (RPN), operators are placed after their operands. For example, a b + or a b c + \*.   **Conversion (Using a Stack):**  To convert an infix expression to postfix, we typically use a stack to keep track of operators and manage the order of operations.   * Initialize an empty stack and an empty output list (or string). * Scan the infix expression from left to right.   + If the token is an operand (like a variable or a number), append it to the output list.   + If the token is an operator, pop operators (if there are any) from the stack to the output list until we find an operator of lower precedence (or with equal precedence, if left-associative).   + Push the current operator onto the stack. * When the input expression has been completely scanned, pop all the operators from the stack to the output list |

| **Algorithm (Student Work Area):**  **Step 1: Initialization**   * + stack (array of characters)   + top (index of the top of the stack, initialized to -1)   + max\_size (maximum size of the stack, input by the user)   **Step 2: Define helper functions**   * + isFull(): Check if the stack is full.   + isEmpty(): Check if the stack is empty.   + push(char value): Push a character onto the stack.   + pop(): Pop and return the top character from the stack.   + peek(): Return the top character from the stack without popping it.   + isOperator(char ch): Check if a character is an operator (+, -, \*, /, ^).   + precedence(char ch): Return the precedence of an operator (+, -, \*, ^).   **Step 3: Function to convert infix to postfix**   * + Initialize variables i and j for traversing the infix and postfix expressions respectively.   + Allocate memory for postfix array (char\* postfix) to store the postfix expression.   + While there are characters in the infix expression:     - If the character is an operand (digit or letter), add it to the postfix.     - If the character is (, push it onto the stack. If the character is ), pop from the stack to postfix until ( is encountered on the stack. Pop and discard (.     - If the character is an operator (+, -, \*, /):       * Pop operators from the stack to postfix until an operator with lower precedence or ( is encountered.       * Push the current operator onto the stack.   + Pop all remaining operators from the stack to postfix.   + Terminate postfix with a null character (\0) |
| --- |

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