**Experiment No. – 3**

Name- Veeransh Shah

Reg. Id- 221070063

**Aim:**

**Write a Program for Prefix Evaluation, Parenthesis Matching, Prefix to Postfix.**

**Theory:**

**1.Prefix Evaluation:**

In prefix notation, operators appear before the operands.

For example:

* **Prefix expression**: + 3 4
* **Equivalent infix expression**: 3 + 4

To Evaluate a Prefix Expression :

* Start from the Rightmost character.
* If it’s an operand (a number), push it onto the stack.
* If it’s an operator, pop the top two operands from the stack, apply the operator, and push the result back onto the stack.
* First popped is Second Operand, and Second Popped is First Operand.
* Repeat until all characters are processed.
* The final result is the value left on the stack.

Prefix notation eliminates the need for parentheses and ensures unambiguous expression evaluation. It’s used in programming languages, calculators, and expression evaluators .

**2.Parenthesis Matching:**

The idea is to put all the opening brackets in the stack. Whenever you hit a closing bracket, search if the top of the stack is the opening bracket of the same nature. If this holds then pop the stack and continue the iteration. In the end if the stack is empty, it means all brackets are balanced or well-formed. Otherwise, they are not balanced.

Time Complexity: O(N)

Space Complexity: O (N)

**3.Prefix to Postfix:**

A prefix expression is one where the operator appears in the expression before the operands.

A postfix expression is one where the operator appears in the expression after the operands. Read the Prefix expression in reverse order (from right to left).

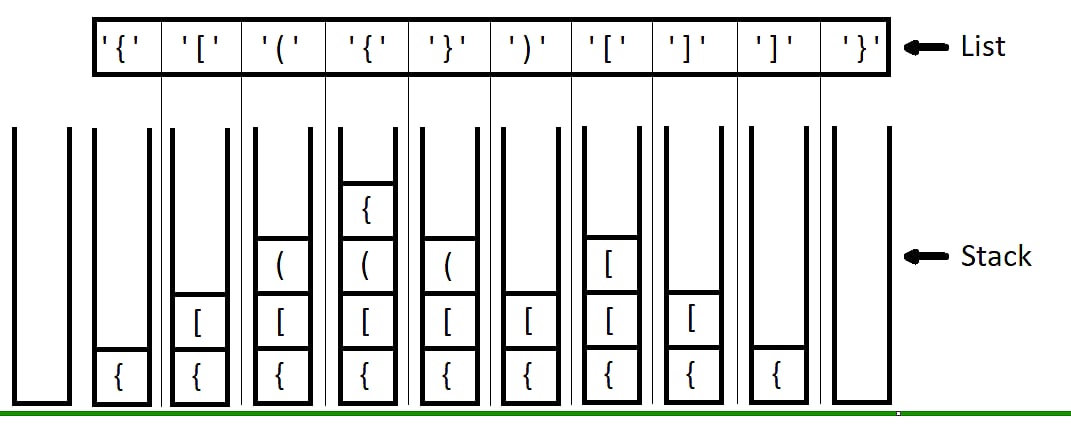
1. If the symbol is an operand, then push it onto the Stack.
2. If the symbol is an operator, then pop two operands from the Stack. Create a string by concatenating the two operands and the operator after them. The string will be of the form operand1 + operand2 + operator. Push the resultant string back to Stack.
3. Repeat the above steps until the end of the Prefix expression.

**Example :**

1. **Prefix Evaluation**

Consider the prefix expression: \* + 3 4 5

1. Start with an empty stack.
2. Read the characters from right to left
   * 5: Operand, push onto the stack.
   * 4: Operand, push onto the stack.
   * 3: Operand, push onto the stack.
   * +: Operator, pop 4 and 3, compute 4 + 3 = 7, and push 7 back.
   * \*: Operator, pop 5 and 7, compute 5 \* 7 = 35, and push 35 back.
3. The final result is 35.
   * + 1. **Parenthesis Matching**



**Algorithm:**

1. Start
2. **Function isBalancedExp(exp):**
   * Create an empty stack **stk**.
   * For each character **ch** in the input expression **exp**:
     + If **ch** is an opening parenthesis (**'('**, **'{'**, or **'['**), push it onto the stack.
     + If **ch** is a closing parenthesis (**')'**, **'}'**, or **']'**):
       - If the stack is empty, return **false** (unbalanced).
       - Pop the top element **x** from the stack.
       - If **x** is not a matching opening parenthesis for **ch**, return **false** (unbalanced).
   * If the stack is empty after processing the entire expression, return **true** (balanced); otherwise, return **false**.

**Algorithm for Prefix Expression Evaluation:**

**Data Structures:**

* **stack<double> operandStack**: Stack to store operands.

1. **Function evaluateprefix(prefixexp):**
   * Create an empty stack **operandStack**.
   * For each character **ch** in the reversed prefix expression **prefixexp**:
     + If **ch** is a digit, push its numeric value onto **operandStack**.
     + If **ch** is an operator (**'+'**, **'-'**, **'\*'**, **'/'**):
       - Pop the top two operands **o1** and **o2** from **operandStack**.
       - Perform the operation based on **ch** and push the result back onto **operandStack**.
       - If **ch** is not a valid operator, output "Invalid Expression" and return **-1**.
   * After processing the entire prefix expression, the result is at the top of **operandStack**.
   * Return the result.

**Main Function main():**

1. **Main Execution:**
   * Create a string **expression** containing parentheses.
   * Check if the expression has balanced parentheses using **isBalancedExp** and output the result.
   * Create a string **prefixexp** containing a prefix expression.
   * Evaluate the prefix expression using **evaluateprefix** and output the result.

**Time Complexity:**

* **Balanced Parentheses Check (isBalancedExp):**
  + The time complexity is O(n), where n is the length of the input expression.
* **Prefix Expression Evaluation (evaluateprefix):**
  + The time complexity is O(n), where n is the length of the input prefix expression.

**Space Complexity:**

* **Balanced Parentheses Check (isBalancedExp):**
  + The space complexity is O(n), where n is the length of the input expression (stack usage).
* **Prefix Expression Evaluation (evaluateprefix):**
  + The space complexity is O(n), where n is the length of the input prefix expression (stack usage).

Top of Form

**Conclusion:**

1. **Prefix Evaluation**: Prefix and Postfix expressions can be evaluated faster than an infix expression. This is because we don’t need to process any brackets or follow operator precedence rule.
2. **Parenthesis Matching**: The stack data structure comes in handy for determining whether or not the syntax has balanced parentheses.
3. **Prefix to Postfix Conversion**: To convert a prefix to postfix expression, we use a stack to hold the operands. Whenever an operator is found, we pop two operands from the stack and push a new operand. The final element at the top of the stack will be our postfix expression.

**Screenshot:**

****