Problem on Stacks

Lesson Plan

PW SKILLS

Infix,Prefix,Postfix

Infix expressions are mathematical expressions where operators are placed between operands, and infix evaluation involves calculating the result of such expressions while considering operator precedence and parentheses.

Let's break down infix evaluation with and without brackets:

Infix Evaluation (Without Brackets):
For infix evaluation without brackets, you need to consider operator precedence to correctly evaluate the expression. You can use the algorithm known as the shunting-yard algorithm to convert infix expressions to postfix (or Reverse Polish Notation, RPN) and then evaluate the postfix expression.

Here's an example of infix evaluation without brackets using the shunting-yard algorithm:

Expression: `3 + 4 * 2 / (1 - 5)^2`

1. Convert infix to postfix: `3 4 2 * 1 5 - 2 ^ / +`
2. Evaluate the postfix expression using a stack-based algorithm or by iterating through the postfix expression.

Infix Evaluation (With Brackets):
When dealing with infix expressions that include brackets, you need to respect the precedence of operations within brackets and evaluate them first.
Expression: `(3 + 4) * 2 / (1 - 5)^2`

1. Start from left to right and identify the innermost brackets.
2. Evaluate expressions within the innermost brackets first.
3. Replace the evaluated expressions with their results in the original expression.
4. Repeat the process until there are no more brackets.

Steps:
- `(3 + 4) * 2 / (1 - 5)^2`
- `7 * 2 / (1 - 5)^2`
- `14 / (1 - 5)^2`
- `14 / 16`
- Result: `0.875`

Evaluating Expressions with Brackets Recursively:
The process involves recursively evaluating subexpressions within brackets until there are no more brackets in the expression.

This evaluation process maintains proper operator precedence by prioritizing operations within brackets, gradually simplifying the expression until the final result is obtained.

Prefix Notation (Polish Notation):
In prefix notation, the operator precedes its operands. For example, the infix expression (5 + 3) * 2 in prefix notation becomes * + 5 3 2.

Postfix Notation (Reverse Polish Notation):
In postfix notation, the operator follows its operands. Using the same infix expression (5 + 3) * 2, the postfix notation is 5 3 + 2 *.

Q. Given an infix expression, the task is to convert it to a prefix expression.

Input: A * B + C / D
Output: + * A B/ C D

Input: (A – B/C) * (A/K-L)
Output: *-A/BC-/AKL

Explanation: To convert an infix expression to a prefix expression, we can use the stack data structure. The idea is as follows:

Step 1: Reverse the infix expression. Note while reversing each '(' will become ')' and each ')' becomes '('.
Step 2: Convert the reversed infix expression to "nearly" postfix expression.
While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.
Step 3: Reverse the postfix expression.

The stack is used to convert infix expression to postfix form.

```
class InfixToPrefixConverter:
    @staticmethod
    def is_operator(c):
        return not c.isalpha() and not c.isdigit()

    @staticmethod
    def get_priority(c):
        if c == '-' or c == '+':
            return 1
        elif c == '*' or c == '/':
            return 2
    elif c == '^':
            return 3
        return 0

    @staticmethod
    def infix_to_postfix(infix):
        infix = '(' + infix + ')'
        l = len(infix)
        char_stack = []
        output = []
    for i in range(l):
            if infix[i].isalpha() or infix[i].isdigit():
                output.append(infix[i])
            elif infix[i] == '(':
                char_stack.append('(')
            elif infix[i] == ')':
                while char_stack[-1] != '(':
                    output.append(char_stack.pop())
                char_stack.pop()  # Remove '(' from the
stack
            else:
```

```
  while (len(char_stack) > 0 and
InfixToPrefixConverter.is_operator(char_stack[-1]) and
                       InfixToPrefixConverter.get_priority(
infix[i]) <=
InfixToPrefixConverter.get_priority(char_stack[-1])):
                    output.append(char_stack.pop())
                char_stack.append(infix[i])

        while len(char_stack) > 0:
            output.append(char_stack.pop())

        return ''.join(output)

    @staticmethod
    def infix_to_prefix(infix):
        # Reverse String and replace ( with ) and vice
versa
        # Get Postfix
        # Reverse Postfix
        l = len(infix)

        # Reverse infix
        reversed_infix = infix[::-1]
        reversed_infix = reversed_infix.replace('(',
'temp').replace(')', '(').replace('temp', ')')

        prefix =
InfixToPrefixConverter.infix_to_postfix(reversed_infix)
 # Reverse postfix
        reversed_prefix = prefix[::-1]

        return reversed_prefix


# Driver code
if __name__ == '__main__':
    s = "x+y*z/w+u"
    print(InfixToPrefixConverter.infix_to_prefix(s))
```

Write a program to convert an Infix expression to Postfix form.
Input: A + B * C + D
Output: ABC*+D+

Input: ((A + B) – C * (D / E)) + F
Output: AB+CDE/*-F+

Idea: To convert infix expression to postfix expression, use the stack data structure. Scan the infix expression from left to right. Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.

Code

```
class InfixToPostfixConverter:
    @staticmethod
    def prec(c):
        if c == '^':
            return 3
        elif c == '/' or c == '*':
            return 2
        elif c == '+' or c == '-':
            return 1
        else:
            return -1

    @staticmethod
    def infix_to_postfix(s):
        st = []
        result = []

        for i in range(len(s)):
  c = s[i]

            # If the scanned character is an operand, add
it to the output string.
            if c.isalnum():
                result.append(c)

            # If the scanned character is an '(', push it
to the stack.
            elif c == '(':
                st.append('(')

            # If the scanned character is an ')', pop and
add to the output string
            # from the stack until an '(' is encountered.
            elif c == ')':
                while st and st[-1] != '(':
                    result.append(st.pop())
                st.pop()  # Remove '(' from the stack

            # If an operator is scanned
            else:
                while st and
InfixToPostfixConverter.prec(c) <=
InfixToPostfixConverter.prec(st[-1]):
                    result.append(st.pop())
                st.append(c)

        # Pop all the remaining elements from the stack
        while st:
            result.append(st.pop())
    return ''.join(result)

# Driver code
if __name__ == '__main__':
    exp = "a+b*(c^d-e)^(f+g*h)-i"
    print(InfixToPostfixConverter.infix_to_postfix(exp))
```

Q. Given a postfix expression, the task is to evaluate the postfix expression.

Input: str = “2 3 1 * + 9 -“
Output: -4

Explanation: If the expression is converted into an infix expression, it will be 2 + (3 * 1) – 9 = 5 – 9 = -4.

Input: str = “100 200 + 2 / 5 * 7 +”
Output: 757

Idea: Iterate the expression from left to right and keep on storing the operands into a stack. Once an operator is received, pop the two topmost elements and evaluate them and push the result in the stack again

Code

```
from typing import List

class PostfixEvaluator:
    @staticmethod
    def evaluate_postfix(exp: str) -> int:
        # Create a stack to store operands
        st = []

        # Scan all characters one by one
        for char in exp:
            # If the scanned character is an operand
(number here),
            # push it to the stack.
            if char.isdigit():
                st.append(int(char))

            # If the scanned character is an operator,
            # pop two elements from the stack and apply the
operator.
            else:
                val1 = st.pop()
                val2 = st.pop()
                if char == '+':
                    st.append(val2 + val1)
                elif char == '-':
                    st.append(val2 - val1)
                elif char == '*':
                    st.append(val2 * val1)
                elif char == '/':
                    st.append(val2 // val1)  # Use integer
division for division
```

Q. Given a postfix expression, the task is to evaluate the postfix expression.

Input: str = “2 3 1 * + 9 -“
Output: -4

Explanation: If the expression is converted into an infix expression, it will be 2 + (3 * 1) – 9 = 5 – 9 = -4.

Input: str = “100 200 + 2 / 5 * 7 +”
Output: 757

Idea: Iterate the expression from left to right and keep on storing the operands into a stack. Once an operator is received, pop the two topmost elements and evaluate them and push the result in the stack again

Code

```
from typing import List

class PostfixEvaluator:
    @staticmethod
    def evaluate_postfix(exp: str) -> int:
        # Create a stack to store operands
        st = []

        # Scan all characters one by one
        for char in exp:
            # If the scanned character is an operand
(number here),
            # push it to the stack.
            if char.isdigit():
                st.append(int(char))

            # If the scanned character is an operator,
            # pop two elements from the stack and apply the
operator.
            else:
                val1 = st.pop()
                val2 = st.pop()
                if char == '+':
                    st.append(val2 + val1)
                elif char == '-':
                    st.append(val2 - val1)
                elif char == '*':
                    st.append(val2 * val1)
                elif char == '/':
                    st.append(val2 // val1)  # Use integer
division for division
  # The final result will be on the top of the stack
        return st.pop()

# Driver code
if __name__ == '__main__':
    exp = "231*+9-"
    print("Result:",
PostfixEvaluator.evaluate_postfix(exp))
```

Q. Prefix Evaluation

Input : -+8/632
Output : 8

Input : -+7*45+20
Output : 25

Algorithm: EVALUATE_PREFIX(STRING)

Step 1: Put a pointer P at the end of the end
Step 2: If character at P is an operand push it to Stack
Step 3: If the character at P is an operator pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack
Step 4: Decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression.
Step 5: The Result is stored at the top of the Stack, return it
Step 6: End

Code

```
from typing import List

class PrefixEvaluator:
    @staticmethod
    def evaluate_prefix(exprsn: str) → float:
        stack = []

        # Iterate over the expression from right to left
        j = len(exprsn) - 1
        while j ⩾ 0:
            # Skip spaces
            if exprsn[j] == ' ':
                j -= 1
                continue
   # If the character is a digit, extract the entire number
            if exprsn[j].isdigit():
                num = 0
                while j ⩾ 0 and exprsn[j].isdigit():
                    num = num * 10 + int(exprsn[j])
    j -= 1
                j += 1
                stack.append(num)

            # If the character is an operator, perform the
operation
            else:
```

```
    o1 = stack.pop()
                o2 = stack.pop()
                if exprsn[j] == '+':
                    stack.append(o1 + o2)
                elif exprsn[j] == '-':
                    stack.append(o1 - o2)
                elif exprsn[j] == '*':
                    stack.append(o1 * o2)
                elif exprsn[j] == '/':
                    stack.append(o1 / o2)

            j -= 1

        # The result will be the only element left in the
stack
        return stack.pop()

# Driver code
if __name__ == '__main__':
    exprsn = "+ 5 * 4 7"
    print("Result:",
PrefixEvaluator.evaluate_prefix(exprsn))
```

Q. Given a Prefix expression, convert it into a Infix expression.

Input : Prefix : *+AB-CD
Output : Infix : ((A+B)*(C-D))

Input : Prefix : *-A/BC-/AKL
Output : Infix : ((A-(B/C))*((A/K)-L))

Algorithm for Prefix to Infix:

Read the Prefix expression in reverse order (from right to left)
If the symbol is an operand, then push it onto the Stack
If the symbol is an operator, then pop two operands from the Stack
Create a string by concatenating the two operands and the operator between them.
string = (operand1 + operator + operand2)
And push the resultant string back to Stack
Repeat the above steps until the end of Prefix expression.
At the end stack will have only 1 string i.e resultant string

Code

```
from typing import List

class PrefixToInfixConverter:
    @staticmethod
    def is_operator(x: str) → bool:
        """
        Function to check if character is an operator or
not.
        Operators supported: '+', '-', '/', '*', '^', '%'
        """
        return x in {'+', '-', '/', '*', '^', '%'}

    @staticmethod
    def pre_to_infix(pre_exp: str) → str:
        """
        Convert prefix expression to infix expression.
        """
        stack = []
   # Read from right to left
        for i in range(len(pre_exp)-1, -1, -1):
            # Check if the symbol is an operator
            if
PrefixToInfixConverter.is_operator(pre_exp[i]):
                # Pop two operands from the stack
                op1 = stack.pop()
                op2 = stack.pop()

                # Concatenate the operands and operator in
infix form
                temp = f"({op1}{pre_exp[i]}{op2})"

                # Push the infix string back to the stack
                stack.append(temp)
            else:
                # Push the operand to the stack
                stack.append(pre_exp[i])

        # Stack now contains the final infix expression
        return stack.pop()

# Driver code
if __name__ == '__main__':
pre_exp = "*-A/BC-/AKL"
    print("Infix:",
PrefixToInfixConverter.pre_to_infix(pre_exp))
```

Q. Given a Prefix expression, convert it into a Postfix expression.

Input : Prefix : *+AB-CD
Output : Postfix : AB+CD-*

Explanation : Prefix to Infix : (A+B) * (C-D)
Infix to Postfix : AB+CD-*

Input : Prefix : *-A/BC-/AKL
Output : Postfix : ABC/-AK/L-*
Explanation : Prefix to Infix : (A-(B/C))*((A/K)-L)
Infix to Postfix : ABC/-AK/L-*

Algorithm: Read the Prefix expression in reverse order (from right to left)
If the symbol is an operand, then push it onto the Stack
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.
string = operand1 + operand2 + operator
And push the resultant string back to Stack
Repeat the above steps until end of Prefix expression.

Code

```
from typing import List

class PrefixToPostfixConverter:
    @staticmethod
    def is_operator(x: str) -> bool:
        """
        Function to check if character is an operator or
not.
        Operators supported: '+', '-', '/', '*'
        """
        return x in {'+', '-', '/', '*'}

    @staticmethod
    def pre_to_post(pre_exp: str) -> str:
        """
        Convert prefix expression to postfix expression.
        """
        stack = []

        # Read from right to left
 for i in range(len(pre_exp)-1, -1, -1):
            # Check if the symbol is an operator
            if
PrefixToPostfixConverter.is_operator(pre_exp[i]):
                # Pop two operands from the stack
                op1 = stack.pop()
                op2 = stack.pop()
```

```
  # Concatenate the operands and operator in postfix form
                temp = op1 + op2 + pre_exp[i]

                # Push the postfix string back to the stack
                stack.append(temp)
            else:
                # Push the operand to the stack
                stack.append(pre_exp[i])

        # Stack now contains the final postfix expression
        return stack.pop()

# Driver code
if __name__ == '__main__':
    pre_exp = "*-A/BC-/AKL"
    print("Postfix:",
PrefixToPostfixConverter.pre_to_post(pre_exp))
```

Q. Postfix to Infix

Input : abJava
Output : (a + (b + c))

Input : ab*c+
Output : ((a*b)+c)

Algorithm

1.While there are input symbol left
…1.1 Read the next symbol from the input.
2.If the symbol is an operand
…2.1 Push it onto the stack.

3.Otherwise,
…3.1 the symbol is an operator.
…3.2 Pop the top 2 values from the stack.
…3.3 Put the operator, with the values as arguments and form a string.
…3.4 Push the resulted string back to stack.

4.If there is only one value in the stack
…4.1 That value in the stack is the desired infix string.
Below is the implementation of above approach:

Code

```
from typing import List

class PostfixToInfixConverter:
    @staticmethod
    def is_operand(x: str) -> bool:
        """
        Function to check if character is an operand (a-z
or A-Z).
        """
        return x.isalpha()

    @staticmethod
    def get_infix(exp: str) -> str:
        """
        Get infix expression for a given postfix
expression.
        """
        stack = []

        for char in exp:
            # Push operands
            if PostfixToInfixConverter.is_operand(char):
                stack.append(char)
            else:
                # Assume input is a valid postfix and
expect an operator
                op1 = stack.pop()
                op2 = stack.pop()
                stack.append(f"({op2}{char}{op1})")

        # There must be a single element in stack now which
is the required infix
        return stack.pop()

# Driver code
if __name__ == '__main__':
    exp = "ab*c+"
    print("Infix:", PostfixToInfixConverter.get_infix(exp))
```

Q. Postfix to Prefix Conversion
Input : Postfix : AB+CD-*
Output : Prefix : *+AB-CD

Explanation : Postfix to Infix : (A+B) * (C-D)
Infix to Prefix : *+AB-CD

Algorithm for Postfix to Prefix:

Read the Postfix expression from left to right
If the symbol is an operand, then push it onto the Stack
If the symbol is an operator, then pop two operands from the Stack
Create a string by concatenating the two operands and the operator before them.
string = operator + operand2 + operand1
And push the resultant string back to Stack
Repeat the above steps until end of Postfix expression.

Code

```
from typing import List

class PostfixToPrefixConverter:
    @staticmethod
    def is_operator(x: str) -> bool:
        """
        Function to check if character is an operator (+,
-, *, /).
        """
        return x in {'+', '-', '*', '/'}

    @staticmethod
    def post_to_pre(post_exp: str) -> str:
        """
        Convert postfix to prefix expression.
        """
        stack = []

        for char in post_exp:
            # Check if symbol is an operator
            if PostfixToPrefixConverter.is_operator(char):
                # Pop two operands from stack
                op1 = stack.pop()
                op2 = stack.pop()
  # Concatenate the operands and operator
                temp = char + op2 + op1

                # Push string temp back to stack
                stack.append(temp)
            else:
                # If symbol is an operand, push it to the
stack
                stack.append(char)

        # The stack now contains the prefix expression
        return ''.join(stack)

# Driver code
if __name__ == '__main__':
    post_exp = "ABC/-AK/L-*"
    print("Prefix:",
PostfixToPrefixConverter.post_to_pre(post_exp))
```