S Abhishek AM.EN.U4CSE19147

Data Structures

1. Implement a program which calculates the value of the postfix expression.

```
class post_fix:
    def __init__(self):
        self.stack = []
        self.top = -1

    def pop(self):
        if self.top == -1:
            return
        else:
            self.top -= 1
            return self.stack.pop()

    def push(self, i):
        self.top += 1
        self.stack.append(i)

    def postfix(self, ab):
        for i in ab:

            try:
                self.push(int(i))

            except ValueError:
                a = self.pop()
                b = self.pop()

                if i == '+':
```

```
                result = int(b) + int(a)  # old val <operator> recent value

            elif i == '-':
                result = int(b) - int(a)

            elif i == '*':
                result = int(b) * int(a)

            elif i == '%':
                result = int(b) % int(a)

            elif i == '/':
                result = int(b) / int(a)

            elif i == '**':
                result = int(b) ** int(a)

            self.push(result)

        return int(self.pop())


input_string = input("Enter the Expression : ")

array = input_string.split(' ')
obj = post_fix()
print(obj.postfix(array))
```

```
Enter the Expression : 1 2 3 * + 4 -
Result : 3
```

```
Enter the Expression : 2 3 * 15 5 / + 10 -
Result : -1
```

```
Enter the Expression : 10 2 * 8 4 / +
Result : 22
```

2. Infix to postfix conversion

```
class stack:
    def __init__(self):
        self.item = []

    def push(self, it):
        self.item.append(it)

    def peek(self):
        if self.isempty():
            return 0
        return self.item[-1]

    def pop(self):
        if self.isempty():
            return 0
        return self.item.pop()

    def isempty(self):
        if self.item == []:
            return True
        else:
            return False

    def display(self):
        if self.isempty():
            return
        temps = stack()
        while not self.isempty():
            x = self.peek()
```

```
            print("~", x)
            temps.push(x)
            self.pop()
        while not temps.isempty():
            x = temps.peek()
            self.push(x)
            temps.pop()

    def check(self, i):
        precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '%': 2, '^': 3}
        if self.peek() == '(':
            return False
        a = precedence[i]
        b = precedence[self.peek()]
        if a <= b:
            return True
        else:
            return False

    def Postfix(self, exp):
        output = ""

        for i in exp:

            if i.isalpha():  # check if operand add to output
                output = output + i
                output = output + " "

            elif i == " ":
                continue
            # If the character is an '(', push it to stack
            elif i == '(':
                self.push(i)

            elif i == ')':  # if ')' pop till '('
                while self.isempty() != True and self.peek() != '(':
                    n = self.pop()
```

```
                output = output + n
                output = output + " "
            if self.isempty() != True and self.peek() != '(':
                return -1
            else:
                x = self.pop()
        else:
            while self.isempty() != True and self.check(i):
                c = self.pop()
                output = output + c
                output = output + " "
            self.push(i)

    # pop all the operator from the stack
    while not self.isempty():
        result = self.pop()
        output = output + result
        output = output + " "
    print("The Postfix Expression is {}".format(output))
    self.display()


stack = stack()
inp_str = input("Enter the Infix Expression : ")
stack.Postfix(inp_str)
```

```
Enter the Infix Expression : A + B - C
The Postfix Expression is A B + C -
```

```
Enter the Infix Expression : A + B * C
The Postfix Expression is A B C * +
```

3. Suppose you have a stack of capacity, l. You keep performing push operations until you fill the stack. Then perform amortized expansion of 5 units. Implement this! For k push operations, calculate the runtime.

```
class Stack:
    def __init__(self, size):
        self.stack = []
        self.size = size

    def Push(self, element):
        if len(self.stack) != self.size:
            self.stack.append(element)
        else:
            self.stack, self.size = expansion(self)
            self.stack.append(element)

    def IsEmpty(self):
        return len(self.stack) == 0

    def Pop(self):
        if self.IsEmpty():
            print("Empty Stack")
        else:
            return self.stack.pop()


def Display(Stack1):
    print("The Stack is : ",end=" ")
    print(Stack1.stack)


def expansion(st):
    s = Stack(st.size + 5)
    for i in range(st.size):
        s.Push(st.stack.pop(0))
```

```
    return s.stack, s.size


n = int(input("Enter Stack Capacity : "))
stack = Stack(n)
p = int(input("Enter the Number of Elements to be Pushed : "))
f = 0
for i in range(p):
    if i > n - 1 and f == 0:
        f = 1
        print("Stack Expanded to the capacity of {}".format(stack.size + 5))
    ele = int(input("Enter the Element {} : ".format(i + 1)))
    stack.Push(ele)
Display(stack)
```

```
Enter Stack Capacity : 6
Enter the Number of Elements to be Pushed : 10
Enter the Element 1 : 1
Enter the Element 2 : 2
Enter the Element 3 : 3
Enter the Element 4 : 4
Enter the Element 5 : 5
Enter the Element 6 : 6
Stack Expanded to the capacity of 11
Enter the Element 7 : 7
Enter the Element 8 : 8
Enter the Element 9 : 9
Enter the Element 10 : 10
The Stack is :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
Enter Stack Capacity : 3
Enter the Number of Elements to be Pushed : 5
Enter the Element 1 : 1
Enter the Element 2 : 2
Enter the Element 3 : 3
Stack Expanded to the capacity of 8
Enter the Element 4 : 4
Enter the Element 5 : 5
The Stack is :  [1, 2, 3, 4, 5]
```