Problem: https://takeuforward.org/data-structure/implement-stack-using-array/
Solve:

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

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

    def Size(self):
        return len(self.stack)

    def Push(self, val):
        self.stack.append(val)

    def Pop(self):
        if self.Is_empty():
            print("It's empty!!!")
        else:
            return self.stack.pop()

    def Peek(self):
        if self.Is_empty():
            print("Stack is empty!")
            return None
        return self.stack[-1]

    def Print(self):
        while not self.Is_empty():
            print(self.Peek(), end=" ")
            self.Pop()
stk1 = Stack()
stk1.Push(3)
stk1.Push(2)
stk1.Push(1)
stk1.Pop()
print(stk1.Size())   # -> 2
stk1.Print()         # -> 2 3
```

Problem: https://takeuforward.org/data-structure/implement-queue-using-array/
Solve:

```
class Queue:
    def __init__(self):
        self.queue = []

    def Is_empty(self):
        return len(self.queue) == 0

    def Size(self):
        return len(self.queue)

    def Enqueue(self, val):
        self.queue.append(val)

    def Dequeue(self):
        if self.Is_empty():
            print("Queue is empty!")
        else:
            return self.queue.pop(0)

    def Front(self):
        if self.Is_empty():
            print("Queue is empty!")
            return None
        return self.queue[0]

    def Print(self):
        while not self.Is_empty():
            print(self.Front(), end=" ")
            self.Dequeue()
q1 = Queue()
q1.Enqueue(3)
q1.Enqueue(2)
q1.Enqueue(1)
q1.Dequeue()
print(q1.Size())    # -> 2
q1.Print()          # -> 2 1
```

Problem : https://takeuforward.org/data-structure/implement-stack-using-single-queue/
Solve:

```
from queue import LifoQueue
stack = LifoQueue()
stack.put(10)
stack.put(20)

print("Popped:", stack.get())  # 20
print("Is Empty?", stack.empty())  # False
print("Size:", stack.qsize())  # 1
while not stack.empty():
    print("Popped:", stack.get())
```

Problem : https://takeuforward.org/data-structure/implement-stack-using-linked-list/
Solve:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
        self.top = None
    def Is_empty(self):
        return self.top is None
    def Size(self):
        if(self.Is_empty()):
            return 0
        cnt = 0
        temp = self.top
        while(temp is not None):
            cnt += 1
            temp = temp.next
        return cnt
    def Push(self,Val):
        new_node = Node(Val)
        if(self.Is_empty()):
            self.top = new_node
        else:
```

```
            new_node.next = self.top
            self.top = new_node

    def Pop(self):
        if(self.Is_empty()):
            print("It's empty!!!")
        else:
            temp = self.top
            self.top = self.top.next
            return temp.data
    def Peek(self):    # get top
        if(self.Is_empty()):
            print("Stack is empty!")
            return None
        return self.top.data
    def Print(self):
        while( not self.Is_empty()):
            print(self.Peek(),end=" ")
            self.Pop()

stk1 = Stack()
stk1.Push(3)
stk1.Push(2)
stk1.Push(1)
stk1.Pop()
print(stk1.Size())  #-> 2
stk1.Print()        #-> 2 3
```

Problem : https://takeuforward.org/data-structure/implement-queue-using-linked-list/
Solve:

```
class Node :
    def __init__(self,Val):
        self.data = Val
        self.next = None

class Queue :
    def __init__(self):
        self.rear = None
        self.front = None
```

```
    def Is_empty(self):
        return self.front is None

    def enqueue(self, Val):
        new_node = Node(Val)
        if self.Is_empty():
            self.front = self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node

    def dequeue(self):
        if self.Is_empty():
            print("Queue empty")
            return None
        else:
            dequeued_Val = self.front.data
            self.front = self.front.next
            if self.front is None:
                self.rear = None

        return dequeued_Val

    def get_front(self):
        if self.Is_empty():
            print("Queue is empty!!")
            return
        return self.front.data

    def get_rear(self):
        if self.Is_empty():
            print("Queue is empty!!")
            return
        return self.rear.data
q = Queue()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
print(f"Front element: {q.get_front()}")     # Front element: 10
```

```
print(f"Rear element: {q.get_rear()}")       # Rear element:  30
print(f"Dequeued element: {q.dequeue()}")    # Dequeued element: 10
print(f"Front element after dequeue: {q.get_front()}")  # Front element
after dequeue: 20
```

Problem: https://takeuforward.org/data-structure/check-for-balanced-parentheses/
Solve:

```
def check_balanced(tx):
    stack = []
    pairs = {')': '(', ']': '[', '}': '{', '>': '<'}
    for i in tx :
        if i in '(<{[' :
            stack.append(i)
        elif i in ')]}>':
            if not stack or stack[-1] != pairs[i]:
                return False
            stack.pop()
    return not stack

tx = "<{[()()]}>"
print(check_balanced(tx))
```

Problem: https://takeuforward.org/data-structure/infix-to-postfix/
Solve:

```
def Priority(c):
    if c == '+' or c == '-':
        return 1
    elif c == '*' or c == '/' or c == '%':
        return 2
    elif c == '^':
        return 3
    else:
        return 0

def from_infix_to_postfix(exp):
    stack = []
    res = ""
    for i in range(len(exp)):
```

```
        if exp[i] == " ":
            continue
        elif exp[i].isdigit() or exp[i].isalpha():
            res += exp[i]
        elif exp[i] == '(':
            stack.append(exp[i])
        elif exp[i] == ')':
            while stack and stack[-1] != '(':
                res += stack.pop()
            stack.pop()  # Remove '('
        else:
            while stack and Priority(exp[i]) <= Priority(stack[-1]):
                res += stack.pop()
            stack.append(exp[i])

    while stack:
        res += stack.pop()
    return res
exp = "3 + 5 * (2 - 8)"
print(from_infix_to_postfix(exp))  # 3528-*+
```

Problem: Postfix to Infix

Solve:

```
def from_postfix_to_infix(exp):
    stack = []
    for char in exp:
        if char.isdigit() or char.isalpha():
            stack.append(char)
        else:  # Operator
            op2 = stack.pop()  # Right operand
            op1 = stack.pop()  # Left operand
            new_expr = f"({op1}{char}{op2})"
            stack.append(new_expr)

    return stack[-1]

postfix_exp = "3528-*+"
print(from_postfix_to_infix(postfix_exp))  # ((3+(5*(2-8))))
```