```python
#1. Convert a Postfix Expression to Infix
def postfix_to_infix(expression):
    stack = []
    for char in expression:
        if char.isalnum():  # If the character is an operand
            stack.append(char)
        else:  # Operator
            op2 = stack.pop()
            op1 = stack.pop()
            stack.append(f"({op1}{char}{op2})")
    return stack[-1]

postfix = "ab+c*"
print("Postfix:", postfix)
print("Infix:", postfix_to_infix(postfix))
```

```
Postfix: ab+c*
Infix: ((a+b)*c)
```

```python
#2. Convert a Postfix Expression to Prefix
def postfix_to_prefix(expression):
    stack = []
    for char in expression:
        if char.isalnum():
            stack.append(char)
        else:  # Operator
            op2 = stack.pop()
            op1 = stack.pop()
            stack.append(f"{char}{op1}{op2}")
    return stack[-1]


postfix = "ab+c*"
print("Postfix:", postfix)
print("Prefix:", postfix_to_prefix(postfix))
```

```
Postfix: ab+c*
Prefix: *+abc
```

```python
#3. Convert a Prefix Expression to Postfix
def prefix_to_postfix(expression):
    stack = []
    for char in reversed(expression):
        if char.isalnum():
            stack.append(char)
        else:  # Operator
            op1 = stack.pop()
            op2 = stack.pop()
            stack.append(f"{op1}{op2}{char}")
    return stack[-1]
```

```
prefix = "*+abc"
print("Prefix:", prefix)
print("Postfix:", prefix_to_postfix(prefix))
```

```
Prefix: *+abc
Postfix: ab+c*
```

```
#4. Implement Multiple Stacks Using a Single List
class MultipleStacks:
    def __init__(self, total_size, num_stacks):
        self.arr = [None] * total_size
        self.tops = [-1] * num_stacks
        self.stack_size = total_size // num_stacks
        self.num_stacks = num_stacks

    def push(self, stack_num, value):
        top_index = self.tops[stack_num] + 1
        if top_index < self.stack_size:
            self.arr[stack_num * self.stack_size + top_index] = value
            self.tops[stack_num] += 1
        else:
            print(f"Stack {stack_num} is full!")

    def pop(self, stack_num):
        if self.tops[stack_num] == -1:
            print(f"Stack {stack_num} is empty!")
            return None
        top_index = self.tops[stack_num]
        value = self.arr[stack_num * self.stack_size + top_index]
        self.arr[stack_num * self.stack_size + top_index] = None
        self.tops[stack_num] -= 1
        return value

    def display(self):
        for i in range(self.num_stacks):
            start = i * self.stack_size
            end = start + self.stack_size
            print(f"Stack {i}: {self.arr[start:end]}")

stacks = MultipleStacks(12, 3)
stacks.push(0, 1)
stacks.push(0, 2)
stacks.push(1, 10)
stacks.push(2, 20)
stacks.display()
print("Popped from Stack 0:", stacks.pop(0))
stacks.display()
```

```
Stack 0: [1, 2, None, None]
Stack 1: [10, None, None, None]
Stack 2: [20, None, None, None]
Popped from Stack 0: 2
Stack 0: [1, None, None, None]
Stack 1: [10, None, None, None]
Stack 2: [20, None, None, None]
```