Course 3: 基本数据结构

一、线性表（线性数据结构）

线性表是一种逻辑结构，描述了元素按线性顺序排列的规则。

在**存储方式**上，常见的线性表存储方式有数组和链表：

数组（array）是一种连续存储结构，将线性表的元素按照一定的顺序依次存储在内存中的连续地址空间上，需要预先分配一定的内存空间。由于可以直接计算元素的内存地址，访问元素的时间复杂度为O（1）；但由于插入和删除元素时需要移动其他元素来保持连续存储的特性，因此插入和删除元素的时间复杂度较高。

链表（linked list）是一种存储结构，是线性表的链性存储方式，其通过节点的相互链接来实现元素的存储。每个节点包含元素本身以及指向下一个节点的指针。链表的插入和删除操作只需要调整节点的指针，时间复杂度为O（1）；访问元素时必须从头节点开始遍历链表，平均复杂度为O（n）。python中没有内置链表类，内置数组类即为列表。

选择使用数组还是链表取决于具体问题的需求和限制。一般而言，频繁进行随机访问操作时，数组更优；频繁插入和删除元素时，链表更优。

在python中，**列表list**更接近于数组的存储结构

在**基本数据结构类型**中，栈stack、队列queue往往通过数组实现。双端队列可以使用循环数组来实现，也可以使用双链表来实现。

栈的顺序原则是LIFO（last-in-first-out）：后进先出。使用列表模拟栈时，时间复杂度为O（1）

队列的顺序原则是FIFO（first-in-first-out）：先进先出.使用列表模拟栈时，在删除元素时，时间复杂度为O（n）

双端队列可看作栈和队列的结合体。在加入或删除元素时，左端和右端都可以进行操作。

二、抽象数据结构——栈（stack）

1.栈的相关操作 (栈类可以用列表模拟实现)

Stack()：创造一个空栈

name.push(item)：向栈顶加入一个新的元素

name.pop()：移除栈顶元素，同时返回该元素值

name.isEmpty()：判断栈是否为空

name.size()：返回栈中元素数量

name.peek(): 返回栈顶元素，但不删除

用python中内置的list模拟实现

```
class Stack:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self,item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
#实际编程使用stack可直接调用list
```

2.匹配括号问题

```
#returns a boolean result as to whether the string of parentheses is balanced
def par_checker(symbol_string):
    s = [] # Stack()
    balanced = True
    index = 0
    while index < len(symbol_string) and balanced:
        symbol = symbol_string[index]
        if symbol == "(":
            s.append(symbol) # push(symbol)
        else:
            #if s.is_empty():
            if not s:
                balanced = False
            else:
                s.pop()
        index = index + 1

    #if balanced and s.is_empty():
    if balanced and not s:
        return True
    else:
        return False
```

更广义上的括号匹配问题：()[]{}不能交叉，且单一种类要相互配对

```
def par_checker(symbol_string):
    s = [] # Stack()
    balanced = True
    index = 0
    while index < len(symbol_string) and balanced:
```

```
            symbol = symbol_string[index]
            if symbol in "([{":
                s.append(symbol) # push(symbol)
            else:
                top = s.pop()
                if not matches(top, symbol):
                    balanced = False
            index += 1
            #if balanced and s.is_empty():
            if balanced and not s:
                return True
            else:
                return False

    def matches(open, close):
        opens = "([{"
        closes = ")]}"
        return opens.index(open) == closes.index(close)
```

3.进制转换问题（先除开的余数最后输出）

考虑十进制转为base进制问题（程序中base<=16）

```
    def base_converter(dec_num, base):
        digits = "0123456789ABCDEF"

        rem_stack = [] # Stack()

        while dec_num > 0:
            rem = dec_num % base
            #rem_stack.push(rem)
            rem_stack.append(rem)
            dec_num = dec_num // base

        new_string = ""
        #while not rem_stack.is_empty():
        while rem_stack:
            new_string = new_string + digits[rem_stack.pop()]

        return new_string
```

4. 中序、前序和后序表达式

中序表达式（中缀Infix expression）：正常的数学运算表达式，括号表示高优先级，内部括号优先级最高。

前序表达式（前缀Prefix expression）：先出操作符，再出操作数。因此，在看前序表达式时，我们要先看排在最后的操作符，将其与之后紧邻的两个操作数作运算，将相应数值反回对应位置，再迭代运算。

后序表达式（后缀Postfix expression）：先出运算数，再出运算符。因此，在看后序表达式时，我们要先看排在最前面的操作符，将其与之前紧邻的两个操作数作运算，将相应数值返回对应位置，再迭代运算。

相较中序表达式，前序和后序表达式不存在括号及乘除的高优先级关系，计算机在解读上更为高效。

（1）通用的中缀转后缀算法

[1] 创造一个空栈opstack用来存储操作符，再创造一个空列表postfixlist用于最后的输出

[2] 将input的中缀表达式转化为token列表

[3] 扫描token：

—— 若token是操作数，直接append到output list

—— 若token是左括号，直接push到操作stack中

—— 若token是右括号，在找到第一个左括号前，依次弹出操作符并append到输出端，直至找到并删除左括号

—— 若token是操作符，首先弹出栈顶优先级高于或等于该操作符的token并append到输出端，然后将其push到栈顶

[4] 当input expression完全扫描过后，依次从栈顶弹出剩余的操作符到output中，最后从前到后输出列表即可

Note：这种算法也被称为Shunting Yard（调度场）算法，由荷兰计算机科学家Edsger Dijkstra（1960）提出

```
def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = [] # Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            #对于正常的算法，有可能遇到浮点数或者多于一位的数字，此时需要修改程序
            #考虑通过循环得到总的num，再进行操作
            postfixList.append(token)
        elif token == '(':
            #opStack.push(token)
            opStack.append(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            #while (not opStack.is_empty()) and (prec[opStack.peek()] >= prec[token]):
            while opStack and (prec[opStack[-1]] >= prec[token]):
                postfixList.append(opStack.pop())
            #opStack.push(token)
            opStack.append(token)
```

```
    #while not opStack.is_empty():
    while opStack:
        postfixList.append(opStack.pop())
    return " ".join(postfixList)
```

（2）后缀表达式求值

后续表达式的计算可以通过使用一个栈来完成，按照以下步骤：

[1] 从左到右扫描后序表达式

[2] 遇到数字时，直接将其压入栈中

[3] 遇到运算符时，从栈中弹出两个数字，先弹出右操作数，再弹出左操作数。将相应数字进行运算：左-运算符-右，将结果压入栈中

[4] 当表达式扫描完毕时，栈顶的数字就是表达式的结果

```
def evaluate_postfix(expression):
    stack = []
    tokens = expression.split()

    for token in tokens:
        if token in '+-*/':
            # 弹出栈顶的两个元素
            right_operand = stack.pop()
            left_operand = stack.pop()
            # 执行运算
            if token == '+':
                stack.append(left_operand + right_operand)
            elif token == '-':
                stack.append(left_operand - right_operand)
            elif token == '*':
                stack.append(left_operand * right_operand)
            elif token == '/':
                stack.append(left_operand / right_operand)
        else:
            # 将操作数转换为浮点数后入栈
            stack.append(float(token))

    # 栈顶元素就是表达式的结果
    return stack[0]
```

将中缀表达式转换为完全括号表达式，亦可以通过后缀表达式的基础完成

```
def infix_to_postfix(infix_expr):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '(': 0}

    postfix_expr = ''
```

```
    stack = []

    for char in infix_expr:
        if char.isdigit():
            postfix_expr += char
        elif char == '(':
            stack.append(char)
        elif char == ')':
            top_token = stack.pop()
            while top_token != '(':
                postfix_expr += top_token
                top_token = stack.pop()
        else:
            while stack and precedence[stack[-1]] >= precedence[char]:
                postfix_expr += stack.pop()
            stack.append(char)

    while stack:
        postfix_expr += stack.pop()

    return postfix_expr

def postfix_to_infix(postfix_expr):
    stack = []

    for char in postfix_expr:
        if char.isdigit():
            stack.append(char)
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            expression = "(" + operand1 + char + operand2 + ")"
            stack.append(expression)

    return stack.pop()

infix_expr = "1+2*3-4"
postfix_expr = infix_to_postfix(infix_expr)
complete_expr = postfix_to_infix(postfix_expr)
print(complete_expr)  # Output: "((1+(2*3))-4)"
```

4. 经典八皇后（用栈迭代可以实现）

问题描述：国际象棋的棋盘有8*8的方格，皇后可以在横、竖、斜线上不限步数地吃掉其他棋子。现在考虑如何将8个皇后放在棋盘上，使之互相不受攻击。

一个简单的表达解的方式是定义一个解a的字符串映射: a = b1b2...b8，其中bi表示第i行皇后所处的列数

串的比较：将串视为整数，整数小的串在前

八皇后问题可以用dfs（深度优先搜索算法）回溯实现，也可以通过stack迭代实现

（1）dfs回溯算法1

```
def solve_n_queens(n):
    solutions = []  # 存储所有解决方案的列表
    queens = [-1] * n  # 存储每一行皇后所在的列数

    def backtrack(row):
        if row == n:  # 找到一个合法解决方案
            solutions.append(queens.copy())
        else:
            for col in range(n):
                if is_valid(row, col):  # 检查当前位置是否合法
                    queens[row] = col  # 在当前行放置皇后
                    backtrack(row + 1)  # 递归处理下一行
                    queens[row] = -1  # 回溯，撤销当前行的选择

    def is_valid(row, col):
        for r in range(row):
            if queens[r] == col or abs(row - r) == abs(col - queens[r]):
                return False
        return True

    backtrack(0)  # 从第一行开始回溯

    return solutions


# 获取第 b 个皇后串
def get_queen_string(b):
    solutions = solve_n_queens(8)
    if b > len(solutions):
        return None
    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string


test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

（2）dfs回溯算法2

```
def is_safe(board, row, col):
    # 检查当前位置是否安全
    # 检查同一列是否有皇后
    for i in range(row):
        if board[i] == col:
            return False
    # 检查左上方是否有皇后
    i = row - 1
    j = col - 1
    while i >= 0 and j >= 0:
```

```
        if board[i] == j:
            return False
        i -= 1
        j -= 1
    # 检查右上方是否有皇后
    i = row - 1
    j = col + 1
    while i >= 0 and j < 8:
        if board[i] == j:
            return False
        i -= 1
        j += 1
    return True

def queen_dfs(board, row):
    if row == 8:
        # 找到第b个解，将解存储到result列表中
        ans.append(''.join([str(x+1) for x in board]))
        return
    for col in range(8):
        if is_safe(board, row, col):
            # 当前位置安全，放置皇后
            board[row] = col
            # 继续递归放置下一行的皇后
            queen_dfs(board, row + 1)
            # 回溯，撤销当前位置的皇后
            board[row] = 0

ans = []
queen_dfs([None]*8, 0)
#print(ans)
for _ in range(int(input())):
    print(ans[int(input()) - 1])
```

（3）stack算法1

```
def queen_stack(n):
    stack = []  # 用于保存状态的栈
    solutions = [] # 存储所有解决方案的列表

    stack.append((0, []))  # 初始状态为第一行，所有列都未放置皇后,栈中的元素是 (row, queens) 的元组

    while stack:
        row, cols = stack.pop() # 从栈中取出当前处理的行数和已放置的皇后位置
        if row == n:    # 找到一个合法解决方案
            solutions.append(cols)
        else:
            for col in range(n):
                if is_valid(row, col, cols): # 检查当前位置是否合法
                    stack.append((row + 1, cols + [col]))

    return solutions

def is_valid(row, col, queens):
```

```
    for r in range(row):
        if queens[r] == col or abs(row - r) == abs(col - queens[r]):
            return False
    return True


# 获取第 b 个皇后串
def get_queen_string(b):
    solutions = queen_stack(8)
    if b > len(solutions):
        return None
    b = len(solutions) + 1 - b

    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string

test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

（4）stack算法2

```
def solve_n_queens(n):
    stack = []  # 用于保存状态的栈
    solutions = []  # 存储所有解决方案的列表

    stack.append((0, [-1] * n))  # 初始状态为第一行，所有列都未放置皇后

    while stack:
        row, queens = stack.pop()

        if row == n:  # 找到一个合法解决方案
            solutions.append(queens.copy())
        else:
            for col in range(n):
                if is_valid(row, col, queens):  # 检查当前位置是否合法
                    new_queens = queens.copy()
                    new_queens[row] = col  # 在当前行放置皇后
                    stack.append((row + 1, new_queens))  # 推进到下一行

    return solutions


def is_valid(row, col, queens):
    for r in range(row):
        if queens[r] == col or abs(row - r) == abs(col - queens[r]):
            return False
    return True


# 获取第 b 个皇后串
def get_queen_string(b):
```

```
    solutions = solve_n_queens(8)
    if b > len(solutions):
        return None
    b = len(solutions) + 1 - b

    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string


test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

二、抽象数据结构——队列（queue）

1.队列的相关操作 (队列可以用列表模拟实现，但存在时间复杂度问题)

Queue(): 创造一个新的空队列

name.enqueue(item): 在队尾加入一个新的元素item

name.dequeue(): 返回并删除对头的元素

name.isEmpty(): 判断队列是否为空

name.size(): 返回队列的元素个数

用python中内置的list模拟实现：

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

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item) #enqueue will be O(n)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
q = Queue()
```

2. 约瑟夫问题

问题描述：有n个人围成一圈，给定一个正整数m。从1号开始顺时针报数，数到m的人退出，再从下一个人从1开始顺时针报数，直到圈内只剩下一个人。

(1) 采用最传统的list建造queue处理问题

```
# 先使用pop从列表中取出，如果不符合要求再append回列表，相当于构成了一个圈
def hot_potato(name_list, num):
    queue = []
    for name in name_list:
        queue.append(name)

    while len(queue) > 1:
        for i in range(num):
            queue.append(queue.pop(0))  # O(N)
        queue.pop(0)                                    # O(N)
    return queue.pop(0)                                 # O(N)


while True:
    n, m = map(int, input().split())
    if {n,m} == {0}:
        break
    monkey = [i for i in range(1, n+1)]
    print(hot_potato(monkey, m-1))
```

(2) 采用python内置deque类处理问题

```
from collections import deque

# 先使用pop从列表中取出，如果不符合要求再append回列表，相当于构成了一个圈
def hot_potato(name_list, num):
    queue = deque()
    for name in name_list:
        queue.append(name)

    while len(queue) > 1:
        for i in range(num):
            queue.append(queue.popleft()) # O(1)
        queue.popleft()
    return queue.popleft()


while True:
    n, m = map(int, input().split())
    if {n,m} == {0}:
        break
    monkey = [i for i in range(1, n+1)]
    print(hot_potato(monkey, m-1))
```

3. 模拟器打印机

（1）问题描述

考虑实验室中的一台计算机——在一小时内，实验室里有10个学生，在一个小时内最多打印两次，且打印的页数从1到20不等（本质是页数+次数两个随机数）

实验室的打印机面临打印速度与打印质量的权衡取舍。我们希望使得学生打印等待的时间尽可能小，打印质量尽可能高（打印速度尽可能慢）

（2）问题抽象

[1] 创建一个打印任务队列。每一个任务到来时都会有一个时间戳。一开始，队列是空的。

[2] 针对每一秒（current second），执行以下操作：

——是否有新创建的打印任务？如果是，以currentsecond作为其时间戳并将该任务加入队列中

——如果打印机空闲，且有正在等待执行的任务，则：

————从队列中取出第一个任务提交给打印机

————用currentsecond减去该任务的时间戳，计算等待时间

————将该任务的等待时间存入一个列表，以准备后续计算平均等待时间

————根据该任务的页数，计算执行时间

——打印机进行一秒的打印，同时若有任务，从该任务执行时间中减去一秒

——若打印任务执行完毕（任务需要的时间小于等于0），则说明打印机回到空闲状态

[3] 当模拟完成之后，计算队列中剩余任务数量以及平均等待时间

（3）python实现

创建三个类，printer、task和printqueue，分别模拟打印机、打印任务和队列

print类需要检查当前是否有待完成的任务，用来确认打印机的工作状态，并且其工作所需的时间可以通过要打印的页数来计算

task类代表单个打印任务，随机数生成器会随机提供页数（random模块中的randrange可供生成随机整数）

```
import random

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

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)
```

```
    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)


class Printer:
    def __init__(self, ppm):
        self.pagerate = ppm
        self.currentTask = None
        self.timeRemaining = 0

    def tick(self):
        if self.currentTask != None:
            self.timeRemaining = self.timeRemaining - 1
            if self.timeRemaining <= 0:
                self.currentTask = None

    def busy(self):
        if self.currentTask != None:
            return True
        else:
            return False

    def startNext(self, newtask):
        self.currentTask = newtask
        self.timeRemaining = newtask.getPages() * 60 / self.pagerate


class Task:
    def __init__(self, time):
        self.timestamp = time
        self.pages = random.randrange(1, 21)

    def getStamp(self):
        return self.timestamp

    def getPages(self):
        return self.pages

    def waitTime(self, currenttime):
        return currenttime - self.timestamp


def simulation(numSeconds, pagesPerMinute):
    labprinter = Printer(pagesPerMinute)
    printQueue = Queue()
    waitingtimes = []

    for currentSecond in range(numSeconds):

        if newPrintTask():
            task = Task(currentSecond)
            printQueue.enqueue(task)
```

```
            if (not labprinter.busy()) and (not printQueue.is_empty()):
                nexttask = printQueue.dequeue()
                waitingtimes.append(nexttask.waitTime(currentSecond))
                labprinter.startNext(nexttask)

            labprinter.tick()

        averageWait = sum(waitingtimes) / len(waitingtimes)
        print("Average Wait %6.2f secs %3d tasks remaining." % (averageWait, printQueue.size()))


    def newPrintTask():
        num = random.randrange(1, 181)
        if num == 180:
            return True
        else:
            return False


    for i in range(10):
        simulation(3600, 10) # 设置总时间和打印机每分钟打印多少页
```

三、抽象数据结构——双端队列（deque：double-ended queue）

与栈和队列不同的是，双端队列的限制很少，有一前一后两端，元素在其中保持自己的位置。双端队列对在哪一端添加和移除元素没有任何限制。新元素既可以被添加到前端，也可以被添加到后段。同理，已有的元素也能从任意一端移除。

1.双端队列的实现

(1) 双端队列的具体操作

Deque():建立空双端队列

name.addfront(item):从对头加入元素

name.addrear(item):从队尾加入元素

name.removefront():从对头删除元素并返回

name.removerear():从队尾删除元素并返回

name.isempty():确认双端队列是否为空

name.size():确认双端队列的元素数量

(2) 用python内置list类实现双端队列

```
    class Deque:
        def __init__(self):
```

```
        self.items = []

    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0, item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)
#在该定义类中，对头为列表右端，队尾为列表左端。insert与pop(0)操作均为O（n）
```

(3) python中的内置deque类

实际上，python中有双端队列类，stack和queue也可以应用双端队列

deque是python的collections模块中的一个类，它提供了一个可变序列，支持从两端进行快速的添加和删除元素操作

deque在两端插入和删除操作的时间复杂度为O（1），要优于列表在左端插入和删除操作的时间复杂度，故而更适合频繁地在两端进行插入和删除操作的场景

```
from collections import deque
d = deque([1,2,3,4])
d.append(5)
d.appendleft(0)
firstone = d.pop()
lastone = d.popleft()
#此外还有extend,index,insert,copy,count等函数
```

2.回文字符串问题

判断一个字符串是否完全对称可以采用deque方法

操作上，先将字符串转换为列表形式，分别从首尾进行pop，判断是否一致，直至deque中剩余1个或0个元素

四、抽象数据结构——链表（Linked list）

不同于前三种数据结构类型以数组作为线性表的存储方式，链表从数据存储方式上以节点+指针的链式存储方式构建数据结构。其由一系列节点组成，每个节点包含一个数据元素和一个指向下一个节点（或前一个节点）的指

针

列表list指的是数据项能够维持相对位置的数据集，其可以在任意位置插入或者删除元素

在python内置的list中，对于列表的实现采用的是顺序存储，即所谓数组；而链表则是另外一种实现列表的方式

1. 链表的基本内容

(1) 链表的节点

[1] 数据元素（数据项）：节点存储的实际数据，可以是任何数据类型

[2] 指针（引用）：该指针指向链表中的下一个节点（或前一个节点）。它们用于建立节点之间的连接关系，从而形成链表的结构

(2) 根据指针的类型和连接方式对于链表的分类

[1] 单向链表（单链表）：每个节点只有一个指向下一节点的指针。链表的头部指针指向第一个节点，而最后一个节点的指针为空（指向None）

[2] 双向链表：每个节点有两个指针，一个指向前一个节点，一个指向后一个节点。双向链表可以从头部或尾部开始遍历，并且可以在任意位置插入或删除节点

[3] 循环链表：最后一个节点的指针指向链表的头部，形成一个环形结构。循环链表可以从任意节点开始遍历，并且可以无限地循环下去。

(3)链表与数组的比较

链表相对于数组的一个重要特点是，链表的大小可以动态地增长或缩小，而不需要预先定义固定的大小。这使得链表在需要频繁插入和删除元素的场景中更加灵活（尤其是在操作头部或尾部节点时）。

然而，链表的访问和搜索操作相对较慢，因为往往需要遍历整个链表才能找到目标节点

(4) python中内置list的对象指针&动态数组性质

注：链表的插入和删除元素时间复杂度还要看在哪个位置插入或者删除

2. 单向链表的实现

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

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, value): #在链表末尾插入元素
        new_node = Node(value)
        if self.head is None:
```

```
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def delete(self, value):
        if self.head is None:
            return

        if self.head.value == value:
            self.head = self.head.next
        else:
            current = self.head
            while current.next:
                if current.next.value == value:
                    current.next = current.next.next
                    break
                current = current.next

    def display(self):
        current = self.head
        while current:
            print(current.value, end=" ")
            current = current.next
        print() #换行操作

# 使用示例
linked_list = LinkedList()
linked_list.insert(1)
linked_list.insert(2)
linked_list.insert(3)
linked_list.display()  # 输出: 1 2 3
linked_list.delete(2)
linked_list.display()  # 输出: 1 3
```

3. 双向链表的实现

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

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def insert_before(self, node, new_node):
        if node is None:  # 如果链表为空，将新节点设置为头部和尾部
            self.head = new_node
            self.tail = new_node
```

```
            else:
                new_node.next = node
                new_node.prev = node.prev
                if node.prev is not None:
                    node.prev.next = new_node
                else:  # 如果在头部插入新节点，更新头部指针
                    self.head = new_node
                node.prev = new_node

        def display_forward(self):
            current = self.head
            while current is not None:
                print(current.value, end=" ")
                current = current.next
            print()

        def display_backward(self):
            current = self.tail
            while current is not None:
                print(current.value, end=" ")
                current = current.prev
            print()

    # 使用示例
    linked_list = DoublyLinkedList()

    # 创建节点
    node1 = Node(1)
    node2 = Node(2)
    node3 = Node(3)

    # 将节点插入链表
    linked_list.insert_before(None, node1)  # 在空链表中插入节点1
    linked_list.insert_before(node1, node2)  # 在节点1前插入节点2
    linked_list.insert_before(node1, node3)  # 在节点1前插入节点3

    # 显示链表内容
    linked_list.display_forward()  # 输出：3 2 1
    linked_list.display_backward()  # 输出：1 2 3
```

五、笔试问题

[1] 当表元素有序排序进行二分检索时，相较链式存储，应采用顺序存储形式（数据结构的两种主要存储方式：顺序、链式）

[2] 顺序表插入算法的平均时间复杂度为O(n)

[3] 链表：所需空间与线性表长度成正比；插入和删除不需要移动元素；不必事先估计存储空间（对于数组，则要事先估计分配多少连续的存储空间）。但不可以随机访问任意元素。

[4] 在广度优先搜索算法中，一般用队列作为辅助数据结构

[5] 环形队列：设队头标号是F，队尾标号是R，环总长度N。删除一个元素，对头+1，增加一个元素，队尾+1，到N就取模。队列总长度（R-F）%N

[6] 数据结构 = 逻辑结构（线性、非线性）+存储结构（顺序、链式、索引、散列）

逻辑结构：数据元素间抽象化的相互关系；常用的线性结构包括线性表、栈、队列、串等，非线性结构包括二维数组、树、图等

存储结构（物理结构）：在计算机存储器中的存储形式。

顺序存储结构（逻辑上相邻结点存储在物理位置上相邻的存储单元中；相邻数据元素的存放地址也相邻）

链式存储结构

索引存储结构：除建立存储结点信息外，还建立附加的索引表来标识结点的地址

散列存储：根据结点的关键字直接计算出该结点的存储地址（哈希存储）