Day01 - 数据结构回顾

数据结构分类

- **线性结构 - N个数据元素的有限序列**

```
【1】顺序表
    1.1> 把'线性表中的节点'按逻辑次序依次存放在一组'地址连续的存储单元里'
    1.2> 用顺序存储方法存储的线性表称为顺序表

【2】链表
    2.1> 定义:物理存储单元上非连续的存储结构，数据元素的逻辑顺序是通过链表中的指针链接实现的
    2.2> 特点:每个结点包括两个部分：存储数据元素的数据域、存储下一个结点地址的指针域
    2.3> 分类
         2.3.1> 单链表
         2.3.2> 单向循环链表
         2.3.3> 双链表
         2.3.4> 双向循环链表
    2.4> '用python实现单链表'

【3】栈stack
    3.1> 定义:栈是后进先出的线性表,只在表尾('栈顶')进行删除和插入操作('即入栈和出栈')
    3.2> 特点:
         3.2.1> LIFO(后进先出Last in first out)
         3.2.2> 只能在栈顶进行插入和删除
    3.3> '用python实现栈'

【4】队列
    4.1> 定义:队列是限定只能在表的一端进行插入,在表的另一端进行删除的线性表
    4.2> 特点:队尾进行插入,队头进行删除
    4.3> '用python实现队列'
```

- **非线性结构 - 一个结点元素可能有多个直接前驱和多个直接后继**

```
【1】集合
 1.1> 特点：  集合中任何两个数据元素之间都没有逻辑关系,组织形式松散

【2】树形结构
 2.1> 特点：树形结构具有分支、层次特性,其形态有点象自然界中的树
 2.2> 几个定义
      2.2.1> 树根          '没有父节点的节点'
      2.2.2> 节点的度       '一个节点的子树的个数'
      2.2.3> 节点的层次    '从根开始定义起，根为第1层'
      2.2.4> 树的深度       '树中节点的最大层次'
 2.3> 二叉树特点
      2.3.1> n个节点的有限集合
```

```
        2.3.2> 由根节点即左子树和右子树组成
        2.3.3> 严格区分左孩子和右孩子
    2.4> 二叉树的遍历
        2.4.1> 广度遍历 - 一层一层遍历，如何实现? --可利用队列
        2.4.2> 深度遍历
                1) 前序遍历 ： 根、左、右
                2) 中序遍历 ： 左、根、右
                3) 后序遍历 ： 左、右、根
    2.5> '用Python实现二叉树'


【3】图状结构
    3.1> 特点：图状结构中的结点按逻辑关系互相缠绕,任何两个结点都可以邻接
```

练习题1 - 栈和队列

- **题目描述+试题解析**

```
【1】题目描述
    用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。队列中的元素为 int 类型

【2】试题解析
    1、队列特点：先进先出（时刻铭记保证先进先出）
    2、栈 A 用来做入队列，栈 B 用来做出队列，当栈 B 为空时，栈 A 全部出栈到栈 B，栈B 再出栈(即出队列)
    3、精细讲解
      stack_a：入队列（队尾）
      stack_b：出队列（队头）

      stack_a队尾：[1,2,3]
      stack_b队头：[]
         stack_b.append(stack_a.pop()) # [3,2,1]
         stack_b.pop()  # 1
```

- **代码实现**

```python
"""
题目:用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。队列中的元素为 int 类型
"""

class Solution:
    def __init__(self):
        # 创建两个栈空间
        self.stack_a = []
        self.stack_b = []

    def push(self,val):
        """入队列:栈a用来入队列"""
        self.stack_a.append(val)

    def pop(self):
        """出队列:栈b用来出队列"""
```

```
        if self.stack_b:
            return self.stack_b.pop()
        else:
            while self.stack_a:
                self.stack_b.append(self.stack_a.pop())
            return self.stack_b.pop()

if __name__ == '__main__':
    q = Solution()
    q.stack_a = [1,2,3]
    q.stack_b = []
    print(q.pop())
```

练习题2 - 链表

- **题目描述+试题解析**

```
【1】题目描述
    输入一个链表，按链表值从尾到头的顺序返回一个 ArrayList

【2】试题解析
    将链表的每个值取出来然后存放到一个列表 ArrayList 中
    解题思路1：将链表中从头节点开始依次取出节点元素，append到array_list中，并进行最终反转
    解题思路2：将链表中从头节点开始依次取出节点元素，insert到array_list中的第1个位置
```

- **代码实现 - 方法1**

```
"""
输入一个链表，按链表值从尾到头的顺序返回一个 ArrayList
"""

class Node:
    """链表节点类"""
    def __init__(self,x):
        self.val = x
        self.next = None

class Solution:
    # 返回从尾部到头部的序列，node为头节点
    def get_list_from_tail_to_head(self,node):
        array_list = []
        while node:
            array_list.insert(0,node.val)
            node = node.next

        return array_list

if __name__ == '__main__':
    s = Solution()
    head = Node(100)
    head.next = Node(200)
    head.next.next = Node(300)
```

```
    print(s.get_list_from_tail_to_head(head))
```

- **代码实现 - 方法2**

```
"""
输入一个链表，按链表值从尾到头的顺序返回一个 ArrayList
"""

class Node:
    """链表节点类"""
    def __init__(self,x):
        self.val = x
        self.next = None

class Solution:
    # 返回从尾部到头部的序列，node为头节点
    def get_list_from_tail_to_head(self,node):
        array_list = []
        while node:
            array_list.append(node.val)
            node = node.next
        # 将最终列表进行反转,无返回值,直接改变列表
        array_list.reverse()

        return array_list

if __name__ == '__main__':
    s = Solution()
    head = Node(100)
    head.next = Node(200)
    head.next.next = Node(300)
    print(s.get_list_from_tail_to_head(head))
```