一、实验目的

掌握线性表相关概念及其实际问题中的运用。

二、实验内容

1. 将两个各有 n 个元素的有序表归并成一个有序表，其最少的比较次数是多少？给出计算过程。
2. 访问单链表中当前结点的后继和前驱的时间复杂度分别是多少？并给出计算过程。
3. 已知长度为 n 的线性表 A 采用顺序存储结构，请写一个时间复杂度为 O（n）、空间复杂度为 O（1）的算法，该算法可删除线性表中所有值为 item 的数据元素。给出详细步骤和结果。

三、实验环境

mac 操作系统， Visual studio 程序集成环境。

四、实验步骤及说明

1. 将两个各有 n 个元素的有序表归并成一个有序表，其最少的比较次数是多少？给出计算过程。
   计算过程：**归并的思想**

- 设置两个指针，分别指向两个表的开头。
- 比较两个指针所指的元素，把较小的元素放入结果表中，并将该指针向后移动一位。
- 重复上述过程，直到其中一个表的所有元素都被取完。
- 最后，将另一个表中剩余的元素直接复制到结果表末尾（**这部分不需要比较**）。
  理解归并的本质就很简单了，通过双指针逐个比较两个有序表的元素，将较小者放入结果中。
  代码及结果如下：

其实这个代码就是对应着我上面讲的归并的计算过程 我还特意挑了一个重复的5来验证下

2. 访问单链表中当前结点的后继和前驱的时间复杂度分别是多少？并给出计算过程。
   计算过程：
   首先明确单链表是由一个个“结点”组成的线性结构。
   单链表中，每个结点仅包含指向其后继结点的指针，因此如果已知当前结点的指针，可直接通过 `p->next` 在常数时间内访问其后继结点，时间复杂度为 **O(1)**。

代码以及结果如下：

```python
class ListNode:
    def __init__(self, val=0):
        self.val = val      # 数据
        self.next = None    # 指向后继结点

# 创建链表: 10 -> 20 -> 30 -> None
node1 = ListNode(10)
node2 = ListNode(20)
node3 = ListNode(30)
node1.next = node2
node2.next = node3


current = node2

print(f"当前结点的值: {current.val}")

if current.next is not None:
    successor = current.next
    print(f"后继结点的值: {successor.val}")
```

```
当前结点的值: 20
后继结点的值: 30
```

然而，单链表没有指向前驱结点的指针，要找到当前结点的前驱，必须从链表头开始顺序遍历，逐个检查每个结点的后继是否为当前结点。在最坏情况下（当前结点为尾结点），需要遍历 n−1 个结点，因此时间复杂度为 **O(n)**。

代码及其结果如下：

```python
class ListNode:
    def __init__(self, val=0):
        self.val = val
        self.next = None

def find_forward(head, target):
    # 如果 target 是头结点，没有前驱
    if head == target:
        return None

    current = head
    # 遍历链表，直到 current.next 是 target
    while current is not None and current.next != target:
        current = current.next

    return current  # 如果没找到，current 会是 None

# 重新创建同样的链表：10 -> 20 -> 30 -> None
node1 = ListNode(10)
node2 = ListNode(20)
node3 = ListNode(30)
node1.next = node2
node2.next = node3

head = node1          # 链表头
current = node2       # 当前结点（值为20）

print(f"当前结点的值：{current.val}")

# 查找前驱（必须从头开始遍历！）
predecessor = find_forward(head, current)

if predecessor is not None:
    print(f"前驱结点的值：{predecessor.val}")
```

```
当前结点的值：20
前驱结点的值：10
```

3. 已知长度为 n 的线性表 A 采用顺序存储结构，请写一个时间复杂度为 O（n）、空间复杂度为 O（1）的算法，该算法可删除线性表中所有值为 item 的数据元素。给出详细步骤和结果。
   这个算法用的是双指针：

- 用两个指针：
  - `i`（**快指针**）：遍历整个数组，检查每个元素。
  - `k`（**慢指针**）：指向“新数组”的下一个位置（即保留元素应放的位置）。
- 遍历过程：
  - 如果 `A[i] != item` → 说明这个元素要保留 → 把它放到 `A[k]`，然后 `k++`
  - 如果 `A[i] == item` → 跳过（不保留，`k` 不动）
- 遍历结束后，`k` 就是**删除后的实际长度**
  代码及其结果如下：

```python
def remove_item(A, item):

    k = 0  # 慢指针
    for i in range(len(A)):
        if A[i] != item:
            A[k] = A[i]
            k += 1
    return k

# 测试
A = [1, 2, 3, 2, 4]
item = 2
print("原列表:", A)

new_len = remove_item(A, item)

print("删除", item, "后的有效部分:", A[:new_len])
print("新长度:", new_len)
```

```
原列表: [1, 2, 3, 2, 4]
删除 2 后的有效部分: [1, 3, 4]
新长度: 3
```

五、实验总结

通过本次实验，我深入理解了线性表在不同存储结构下的操作特点及其时间与空间效率。
按道理用C++标准更好 但是我发现用python写的更快 代码更简洁
首先，在**归并两个有序表**的问题中，我了解了归并算法的基本思想，并通过构造了一个特殊的例子来验证了**最少比较次数为 n** 的结论。

其次，在**单链表结点访问**的分析中，我明确了单链表“单向性”的本质：**后继可直接访问（O(1)）**，而**前驱必须从头遍历（O(n)）**。通过编写 Python 模拟代码，我直观地体会到指针结构对操作效率的影响，也理解了为何在需要频繁访问前驱的场景中应选用双向链表。

最后，在**顺序表原地删除元素**的算法设计中，我学会了使用**双指针技巧**在 O(n) 时间和 O(1) 空间内完成删除操作。该方法避免了频繁移动元素或创建新数组，极大提升了效率，体现了“原地操作”在算法优化中的重要价值。

总的来说，本次实验不仅巩固了我对线性表基本操作的理解，也培养了我分析算法时间/空间复杂度的能力，为后续学习更复杂的数据结构和算法打下了坚实基础。