第3章 线 性 表

线性表是最简单也是最基本的一种线性数据结构。在程序中，如果需要将一组（通常是同为某一种类型）数据元素作为有序序列进行整体管理和使用时，则往往创建线性表这种数据结构。因此，一个线性表不仅是某类元素的一个集合，而且记录着元素之间的一种顺序关系。它通常有两种存储表示方法：顺序表和链表，它的主要基本操作是插入、删除和查找等。线性表在实际程序中应用非常广泛，还往往被用作更复杂数据结构的实现基础。在Python语言中，内置的列表类型list可以支持这种类型的操作需求，常用来作为线性表的顺序实现方式。本章将学习线性表的基本概念及其两种存储表示方式、链表的变形及一些应用实例。

3.1 线性表的概念

3.1.1 基本术语和概念

线性表是 n 个（$n \geqslant 0$，$n=0$ 称为空表）数据元素的集合：表中各个数据元素具有相同特性，表中相邻的数据元素之间存在“一对一的序偶”关系。通常记为

$$(e_1, e_2, \cdots, e_{i-1}, e_i, e_{i+1}, \cdots, e_{n-1}, e_n)$$

其中，e_{i-1} 领先于 e_i，称 e_{i-1} 是 e_i 的直接前驱元素，e_i 是 e_{i-1} 的直接后继元素。在非空线性表中，存在着唯一的一个表首元素和唯一的一个表尾元素。除表首元素之外，表中的其他元素均有且仅有一个直接前驱元素；除表尾元素之外，表中的其他元素均有且仅有一个直接后继元素。

例如，某个集合中只有4个整数1、2、3、4，而且1与2之间、2与3之间、3与4之间都存在一对一的顺序关系，那么就说这个集合是一个线性表。

可以表示为

$$E1=\{1,2,3,4\}$$
$$R1=\{(1,2),(2,3),(3,4)\}$$
$$L1=(E1,R1)$$

该线性表实例如图 3.1 所示，在这个例子中，2 是 3 的直接前驱，3 是 2 的直接后继。对于表首元素 1 来说，只有直接后继元素 2；而尾元素 4 则只有直接前驱元素 3。

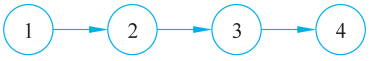

图 3.1　线性表的实例

3.1.2　线性表的操作

作为一种包含元素的数据结构，线性表通常都会支持一些基本操作，例如，如何创建和销毁一个线性表、如何判断表是否为空、如何没有重复和没有遗漏地访问表中的每一个元素、如何往表中插入或者从表中删除一个元素等。在很多实际问题的求解中，如果用到了线性表，对表的操作往往能转换成一个或者多个基本操作的组合或者升华，从而完成程序中的各类应用要求。

从作用性质来看，线性表结构的基本操作可以分为以下三类。

（1）构造操作：用于构造出该数据结构的一个新实例。

initList()：表构造操作，创建一个新表。

destroyList()：表销毁操作，销毁已存在的一个表。

（2）访问操作：用于从已有该数据结构的实例中提取某些信息，但不创建新结构实例，也不修改被操作的结构实例。

is_empty()：判断是否为一个空表。

len()：获得表的长度。

search(elem)：查找元素 elem 在表中出现的位置，不出现时返回－1。

getelem(i)：取出表中第 i 个元素，i 不合法时返回 None。

traverse()：依次访问表中的每个元素。

（3）变动操作：用于修改已有的该数据结构实例。

prepend(elem)：将元素 elem 加入表中作为第一个元素。

append(elem)：将元素 elem 加入表中作为最后一个元素。

insert(elem,i)：将 elem 加入表中作为第 i 个元素，其他元素的顺序不变。

delFirst()：删除表中的首元素。

delLast()：删除表中的尾元素。

del(i)：删除表中第 i 个元素。

forall(op)：对表中的每个元素执行操作 op。

上述各个操作的定义仅对抽象的线性表而言，定义中尚未涉及线性表的存储结构以及实现这些操作所用的编程语言，不同的编程语言也可能会影响需要实现的基本操作集合，例如，Python 能自动回收不用的对象，因此不需要专门实现销毁结构的操作。目前，利用这些基本操作可以避开技术细节完成研究算法、分析问题等工作。

从支持操作类型的角度看，线性表结构又可以分为以下两类。

（1）不变数据结构：该结构只支持构造操作和访问操作，不支持变动操作。创建之后，结构和存储的元素信息都不改变，所有得到该类结构的操作都是创建新的结构实例。例如，Python 中的 tuple 和 frozenset 类型。

（2）变动数据结构：该结构支持变动操作。在创建之后的存续期间，其结构和所保存的信息都可能变化。例如，Python 中的 list、dict、set 等类型。

3.1.3 线性表的实现基础

对于 3.1.2 节中介绍的线性表需要实现的基本操作，都要依托于线性表的具体存储结构加以实现，通常情况下，基于计算机内存中存储数据的特点以及各种重要操作实现的效率，提出了以下两种线性表的存储表示方法。

（1）顺序存储表示：将表中元素顺序地存放在一大块连续的存储区里，采用这种存储结构的线性表称为“顺序表”。

（2）链式存储表示：将表元素存放在通过链接构造起来的一系列存储块里，采用这种存储结构的线性表称为“链表”。

3.2 顺序表

3.2.1 顺序表的定义

在计算机中表示线性表的最简单直观的方法是将表中的数据元素按顺序存放在一片足够大的地址连续的存储单元里，即从首元素开始将线性表中的数据元素一个挨着一个地存放在某个存储区域中，这种存储方式称为线性表的顺序存储表示。相应地，把采用这种存储结构的线性表称为顺序线性表，简称为顺序表。

顺序表中元素之间的逻辑顺序关系与元素在存储区里的物理位置保持一致，因此只要是一个表里保存的元素类型相同，则采用顺序表结构可以方便地实现元素随机存取，即表中任何元素的位置计算非常简单，可以在 $O(1)$ 时间内完成随机访问操作。

设有一个顺序表对象，存储在一片连续的内存区域中，该存储区的起始地址为 L_0，存储在 L_0 处的元素为 e_0，即 $\mathrm{Loc}(e_0)=L_0$，若表中每一个元素需要用的存储单元个数为 a，则表中第 i 个元素 e_i 的地址计算公式为（i 从 0 开始）

$$\mathrm{Loc}(e_i)=L_0+i\times a$$

顺序表的元素存储布局情况如图 3.2 所示。

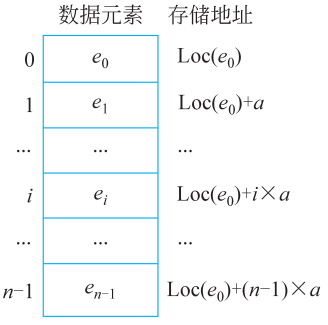

图 3.2 顺序表的元素存储布局示意

如果表元素大小不统一，则若按照上述方式将表元素依序存放在一个存储区域中，则无法使用以上方式来计算元素地址，这时可以采取以下策略完成顺序结构的定义，即将实际元素的引用链接保存在一个顺序表中。由于每个链接所需的存储大小相同，则还是可以通过上述公式，计算出

各个索引的元素链接的存放位置，对链接做一次间接访问，即可得到实际元素的数据了，具体实现方法就不展开详述了。

3.2.2 顺序表的基本实现

建立了线性表之后，一项重要的操作是可以往其中动态地加入或者删除元素，也就是说，在一个表存续期间，其长度可能会发生变化。那么在建立表时应采用多大的连续存储区则是一个必须要考量的问题，因为对于顺序存储，存储块一旦分配，就有了大小确定的元素容量。若要考虑变动的表，就需要区分存储区容量和表中元素个数这两个概念。因此，可以增设两个变量 max 和 n，分别记录元素存储区的容量，以及当前表中实际的元素个数。在发生插入或删除等表元素变化的操作时，可同步更新 n 的值，这就是顺序表的一般结构，如图 3.3 所示。

在以上结构中，采用了列表作为顺序表存储对象，max 表示表的容量，n 记录表中实际元素的个数。在此基础之上，线性表的一些基本操作则很容易实现，本节只讨论顺序表的几个主要操作的实现算法。

1. 创建和访问操作

创建空表：即构造一个空的顺序表。需分配一块元素存储区域，记录表的容量并将元素计数值设置为 0，如图 3.4 中是一个容量为 8 的空表，在创建表的同时，应将表信息域的 max 和 n 设置好，保证顺序表的合法性。

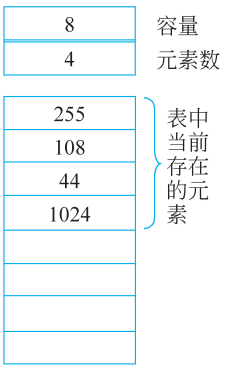

图 3.3 顺序表的一般结构示例

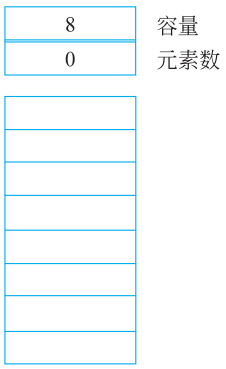

图 3.4 新建的空顺序表

简单判断操作：即判断表是否为空或者是否为满的操作。因为顺序表结构拥有记录表的容量和元素计数值的信息域 max 和 n，所以判断表空和表满的操作都很简单，表空即判断 n 是否为 0，表满即判断 n 是否等于 max，两个操作的时间复杂度均为 $O(1)$。

访问指定索引元素操作：即给定索引 i，访问顺序表第 i 个元素。通常首先需要进行 i 值的合法性判断，即 $0 \leqslant i \leqslant n-1$。若超出该范围则是非法访问，若索引合法，则可利用 3.2.1 节中的地址计算公式，由内定位置得到元素的值，前文已经分析了，该操作不依赖于表的大小，时间复杂度为 $O(1)$。

查找元素操作：即在顺序表中查找其值与给定的值相等的数据元素，并返回其在元素表中出现的索引，该操作又称为检索。通常会从表中的第一个元素开始，依次和所给的值进行比较，一旦找到一个与该值相等的数据元素，则返回它在表中的“位序”（即索引）。如果直

到表内所有元素都比较完也没有找到相等元素，则返回一个特殊值(如“－1”)。有时候也会要求查找在表中某一指定位置 k 之后是否存在所查找的对象，可只需要从 $k+1$ 位置的元素开始比较即可，查找元素操作的时间复杂度为 $O(n)$。

2. 变动操作

删除元素：即从已有顺序表中删除指定索引位置的元素。这种删除操作通常都会要求实现保序定位删除，即不仅需要保证删除操作结束后，剩余元素在内存中仍然是连续存放的，而且还需要保持剩余元素的相对顺序。因此，一个可行的实现办法就是从待删元素的下一位置起，逐个顺序地将元素上移覆盖。如图 3.5 所示的这个顺序表，开始有 5 个元素，计划要删除位置 2 处的 108，则可以将位置 3 上的 44 上移到位置 2，将位置 4 上的 1024 上移到位置 3，最后将元素计数变量 n 减 1，有效元素更新为 4 个，对应存放在位置 0 到位置 3，而位置 4 上的元素就不是表中的有效元素了，通过这种方式实现顺序表的删除操作。这种删除操作实现的时间主要花费在元素的挪动上面。对于元素个数为 n 的表，删除表首元素需要挪动 $n-1$ 个元素，是效率最低的；而删除表尾元素则无须挪动元素，是效率最高的。假设在任一位置上删除元素的概率相同，则删除顺序表中的某一元素平均需要挪动约一半的元素，因此，该操作的时间复杂度为 $O(n)$。

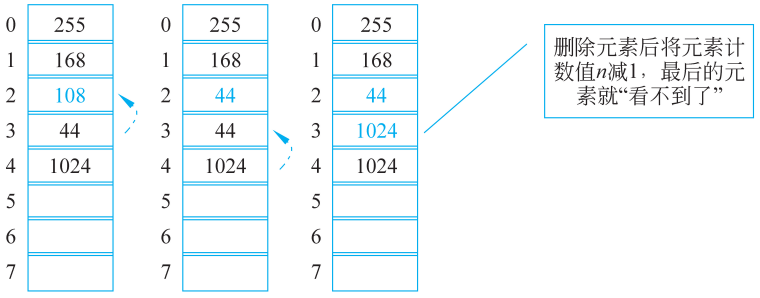

图 3.5　顺序表删除元素操作

插入元素：即在已有顺序表中指定索引位置插入一个元素。这种插入操作通常也都会要求实现保序定位插入，即不仅需要保证操作后所有的元素是连续的，而且原来元素的相对顺序不变。只是在具体实现插入时为了避免错误覆盖，需要逆序，即从最后一个元素起到原来插入位置上的元素，从后往前，逐个往后挪一位，从而腾出位置给待插入的新元素。如图 3.6 所示，以在目前这个元素个数为 4 的表中，在位置 2 处插入元素 108 为例，在插入元素之前，需要先把位置 2 上的 44 往后挪到位置 3 处，但是，位置 3 上有元素 1024，所以需要先把 1024 往后挪，这里恰好 1024 就是表中最后一个元素，因此，可以把它直接往后挪动一个位置，位置 3 上的 1024 后移之后，位置 3 就相当于可以被覆盖了，这时就可以把 44 后移到位置 3 上，此时，位置 2 这个指定插入位置就准备就绪了。将待插入的元素 108 放进去即可。插入新元素后，需将元素计数变量 n 加 1。对于元素个数为 n 的表，在表首插入元素需要挪动 n 个元素，是效率最低的；而在表尾插入元素则无须挪动元素，是效率最高的。假设在任一位置上插入元素的概率相同，则在顺序表中插入某一元素平均也需要挪动约一半的元素，因此，该操作的时间复杂度也为 $O(n)$。

对于插入操作需要注意的一点是：在插入元素前需要检查表是不是已经满了。如果

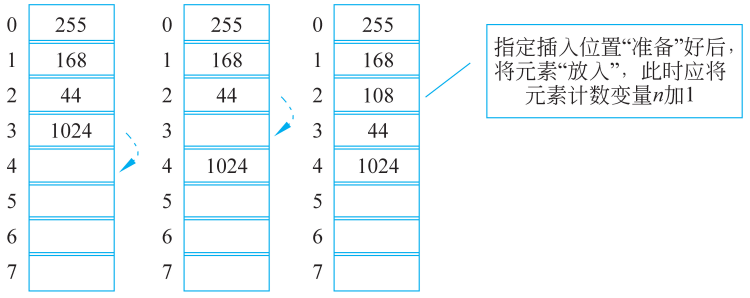

图 3.6　顺序表插入元素操作

是，插入前就需要更换一片更大的存储空间，然后把已有的元素一个个都“腾挪”过去，这也是需要消耗时间的。由于 Python 中的列表类型在内存中占据的即是一个地址连续的存储区域，因此可以直接采用 Python 的列表来描述顺序表中数据元素的存储区域。Python 的列表类型采用一种等比扩容的策略，如图 3.7 所示，比如最初分配的表容量是 4，第一次满时扩到 8，第二次满时扩到 16，以此类推，容量呈指数增长。在这种策略下，可以保证尾端操作的平均复杂度仍为 $O(1)$。

图 3.7　Python 列表的空间分配策略

由于 Python 列表不需要预先分配大小，所以通常情况下直接定义一个列表结构，并采用列表的相关函数和方法就可以完成顺序表的一些基本操作，必要时也可额外设置变量 max 表示表的最大范围。这里就不给出在 Python 里定义顺序表的实际代码了。

3.2.3　顺序表例题

例 3.1　（力扣 27）移除元素。

【题目描述】

给定一个可能包含重复元素的无序列表 nums 和一个确定的数值 val，设计算法实现原地移除数组中所有数值等于给定值 val 的元素，并返回移除后数组的新长度 k。算法空间复杂度要求为 $O(1)$。

例如，当 nums＝[3,2,2,3]，val＝3 时，应返回 2；而当 nums＝[0,1,2,2,3,0,4,2]，val＝2 时，应返回 5。

【解题思路】

由于题目明确要求算法空间复杂度为 $O(1)$，则不能采取新建顺序表，将原表中不等于 val 的元素添加到新表中的策略，只能采取原表操作，需遍历顺序表的同时删除与 val 相等的元素。若此时选择从前往后遍历，当遇到同 val 相等的元素删除时，后边的元素会自动覆盖到被删除元素的位置上，此时循环会略过这个前移的元素往下走，造成遍历的遗漏。因此，应采用逆序遍历来实现移除过程。

【参考代码】

```
def removeElement(self, nums: List[int], val: int) -> int:
    for i in nums[::-1]:
```

```
            if i == val:
                nums.remove(i)
        return len(nums)
```

例 3.2 跑道上的小朋友。

【题目描述】

在一条总长为 m 米的直线跑道上，每隔 1 米就站着一个小朋友。这条跑道可以被想象成一条数轴，其中，数轴的起点位于 0，终点位于 m；数轴上的每个整数值点，即 0，1，2，…，m，都对应着一个小朋友的位置。现在，由于学校活动需要，部分跑道区域将被划为比赛区域。这些比赛区域可以通过它们在数轴上的起始和终止位置来界定。每个区域的起始和终止位置都是整数，并且不同的区域可能会有重叠部分。这些区域可以通过一个包含多组起始和终止位置的列表 ls 来标识。设计算法计算在移除所有比赛区域内的小朋友（包括区域边界上的小朋友）之后，跑道上剩余的小朋友总数。

例如，当 $m=400$，ls=[(25,75),(50,150),(158,358)]时，应返回 74。

【解题思路】

定义一个恰好含有 $m+1$ 个元素的顺序表 flag 表示每个位置上有无小朋友，flag[i]为 1 表示 i 号位置有小朋友，否则表示无小朋友。则让小朋友离开该点的操作，仅需将 flag 中对应的元素变为 0。最后统计 flag 中 1 的个数（对 flag 列表求和）即可。

【参考代码】

```
def childCount(m, ls):
    flag = [1] * (m + 1)
    for begin, end in ls:
        flag[begin : end + 1] = [0] * (end - begin + 1)
    return sum(flag)
```

3.3 单链表

3.3.1 单链表的定义

3.2 节介绍的顺序表是使用一组连续的存储单元来存放线性表中的元素，元素间的顺序关联是由元素在物理存储器中的相对位置来自然满足的。本节介绍的单链表数据结构同样是一种线性表，但它是链式存储结构，可以用一组地址任意的存储单元来存放线性表中的数据元素，在这种存储机制下，需要为每个元素附加链接来指示其直接后继的存储位置，即用链接显式地表示元素之间的顺序关联。单链表的元素存储布局情况如图 3.8 所示。

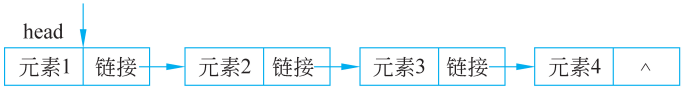

图 3.8 单链表的元素存储布局

单链表有一个至关重要的表头变量（也常称为表头指针），它记录了单链表中第一个结点的位置信息，通过它就可以顺着链找到每一个结点。对于表尾元素，虽没有直接后继，但

为了保持一致性，也给它添加一个链接，在 Python 中它的值就设为 None，图示时用^表示。通常，对于一个已经建好的单链表，只需要记住其表头变量就可以了，因此也常用表头变量来代表一个单链表。

单链表的优点是不需要大片连续的存储区域，内存动态管理灵活，在指定结点后插入结点或者删除指定结点的直接后继时效率很高，复杂度是 $O(1)$。但是，单链表在查找或访问指定索引值的结点时，需要从表头一个一个循链接遍历下去，因此效率比较低，复杂度为 $O(n)$。

3.3.2 单链表的基本实现

单链表是由一个个数据结点构成的。每个结点含有两个信息域，一个表示元素本身，称为元素域；另一个表示直接后继的存储位置，称为链接域。结点构成如图 3.9 所示。

图 3.9 单链表结点构成

Python 并没有提供现成的单链表类型，要想使用单链表，需要自定义相应的类。由于单链表是由一个个结点链接而成的，所以，首先需要定义表示结点的类型。这里使用 class 保留字定义一个名为 ListNode 的类来表示单链表中的结点。它包含两个属性：val 和 next，分别表示结点的元素值和直接后继链接，其定义如下。

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

ListNode 类里只包含一个初始化方法，用于实现给实例对象的 val 和 next 两个属性赋值。该方法的最后一个形参命名为 next_，是为了避免与 Python 标准函数 next 重名，这也符合 Python 的命名惯例。

定义了结点类之后，即可通过以下逐一建立结点的方法生成一个简单的单链表。该链表有 4 个结点，结点元素值从头到尾依次为 1、2、3、4，表头变量用 head 来表示。

```
node4 = ListNode(4)
node3 = ListNode(3, node4)
node2 = ListNode(2, node3)
node1 = ListNode(1, node2)
head = node1
```

在上面的代码中，每个结点的链接域存放的是它的直接后继的对象名。在建立一个结点之前，如果该结点的后继结点已存在，则该结点的链接域的设置就会十分简单，因此，逆序创建链表是一个很自然的选择。上述代码的执行过程如图 3.10 所示。首先实例化一个 ListNode 类的对象 node4 表示结点 4，该结点的链接域 next_属性采用默认值 None，对象名 node4 中就包含该结点在内存中的地址信息，因此可以将该对象名作为结点 3 的链接域来创建结点 3，以此类推，创建了结点 2 和结点 1，而结点 1 的对象名 node1 也对应着链表首结点的位置，因此，将其赋给表头变量 head 来表示整个单链表。

如果需要创建一个含有更多个结点的单链表，也可以采用类似的方式，并且可以使用循环来简化相似的代码。例如，要想创建结点元素值依次为 1～n 的单链表 head，可以使用以下代码来实现逆序创建过程。

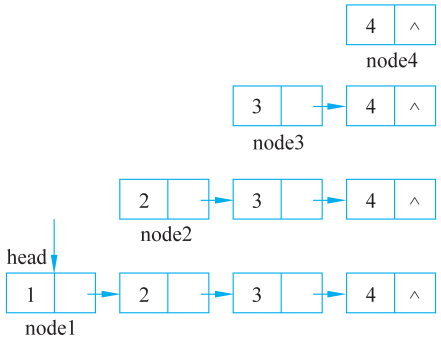

图 3.10　单链表的逆序创建

```
p = None
for i in range(n, 0, -1):
    p = ListNode(i, p)
head = p
```

也可以采用顺序方式创建结点元素值依次为 1～n 的单链表，过程比逆序创建略微复杂，需要设置一个辅助变量 p，用于记录当前创建的结点位置，实现代码如下。

```
head = ListNode(1)
p = head
for i in range(2, n + 1):
    p.next = ListNode(i)
    p = p.next
```

这段代码的执行流程如图 3.11 所示。首先创建结点 1，让表头变量 head 指向它，结点 1 的链接域为空，同步创建变量 p 让其指向结点 1；接着创建第二个结点，并设置 p.next 的值为新建立的结点 2，即让第一个结点链接上了第二个结点。然后，通过执行 $p=p$.next，让变量 p 指向当前创建完毕的第二个结点。以此类推，让结点顺序被创建，逐个被链接，形成最终的单链表。

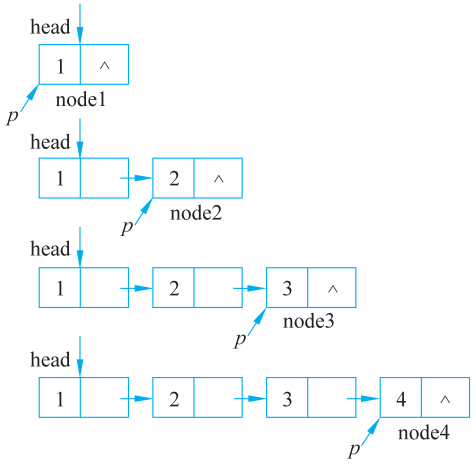

图 3.11　单链表的顺序创建

单链表建立之后，若要实现从头到尾顺序遍历单链，则也可以借助一个 p 指针，从链表头开始，依次后移，通过循环的方式实现。代码如下，在该遍历过程中实现了以每行一个元素的方式输出表中各元素的值。

```
p = head
while p:
    print(p.val)
    p = p.next
```

3.3.3 单链表基本操作的实现

本节讨论单链表的几个主要操作的实现算法。

1. 创建和访问操作

创建链表：从 3.3.2 节中得知要掌握一个单链表，需要且仅需要掌握该表的表头变量。因此，可以按如下方式定义一个单链表类 LList。

```
class LList:
    def __init__(self):
        self._head = None
```

LList 类只包含一个名为_head 的属性，表示表头变量，在 LList 类的初始化函数里，它被直接设置为空链接(None)。因此，可以通过诸如 mlist＝LList()的方式来实例化一个空链表 mlist。

但若想实现根据传入的列表来构建一个单链表，让单链表中结点自表头起依次存放列表中自表头起的每一个元素，则需要修改 LList 类的初始化函数，为其增加一个形参 ls 用以接收传入的列表。同时，可以采用逆序方式或者顺序方式根据 ls 中的元素值逐个创建 ListNode 型的结点，从而建立单链表，并将表头变量赋给 self._head 即可。其代码实现参考代码清单 3.1 中的魔术方法__init__。

判断是否为空表：判断是否为空链表只需要判断表头变量的值是否为 None 即可。其代码实现参考代码清单 3.1 中的实例方法 is_empty。该操作的时间复杂度为 $O(1)$。而对于单链表，是不需要判断表是否已满的，因为结点可以存放在任意位置上，不存在表满一说，除非所有存储空间用完。

求表长：在顺序表中，求表长的时间复杂度是 $O(1)$，因为表中设置了专门的元素个数计数变量。但求单链表的表长需要从表头开始，顺着链表一个结点一个结点地数，直到最后一个结点被数完为止。其代码实现参考代码清单 3.1 中的魔术方法__len__。该方法的时间复杂度为 $O(n)$。

将单链表转换字符串：在实际应用中，常常需要将表中的元素转换为某种形式的字符串，如“[1,2,3,4]”这种列表形式的字符串。这也需要从表头开始，顺着链表一个结点一个结点地将其元素值以要求的字符串形式拼接在一起，直到最后一个结点被拼完为止。其代码实现参考代码清单 3.1 中的魔术方法__str__。该方法的时间复杂度为 $O(n)$。

2. 变动操作

定位插入元素：定位插入元素操作即要求在表中第 i 个结点之后插入一个新元素。在

链表中加入新元素时，并不需要移动已有的元素，只需要建立一个新结点，然后根据位置要求以修改链接的方式将新结点插入链表即可。定位插入元素操作的代码实现参考代码清单 3.1 中的实例方法 insert。但针对不同位置的操作复杂度可能不同。

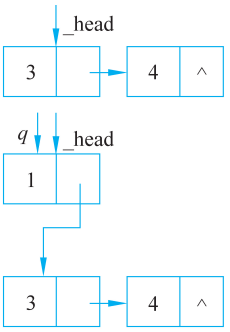

图 3.12 在单链表表头插入元素

假定传入的 i 值合法，也就是它的值大于或等于 0 并且小于或等于表长。当 i 等于 0 时，表示插到表头结点之前，成为新的表头结点。例如，目前表中有 3、4 两个结点，要想在表头插入结点 q，则可以通过将 q 的后继链上表头结点，然后将表头变量更新为 q 即可，实现过程如图 3.12 所示。

而对于一般的情况，即 i 大于 0 且小于或等于表长时，则需要通过从表头结点开始计数，来找到第 i 个结点。这里的计数类似于求表长的操作，不同之处在于 count 的初值设为 1，表示表头结点是第 1 个结点。设 p 初始指向头结点，循环条件 count$<i$ 则表示还没有数到第 i 个结点时就继续循环；p 指针不断后移，当循环停止时，p 所指的就是第 i 个结点。之后，将 q 插到 p 之后，也就是将 p 的后继变成 q 的后继，而 q 本身成为 p 的后继即可，实现过程如图 3.13 所示。

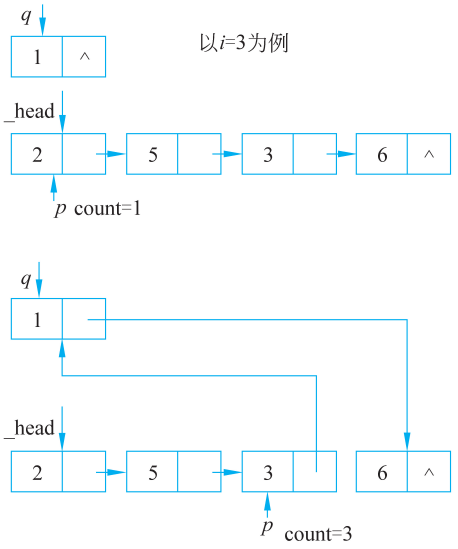

图 3.13 在单链表第 i（i 大于 0 且小于或等于表长）个位置插入元素

从以上分析可以得出，对于单链表的插入操作，在表头插入最快，其时间复杂度为 $O(1)$，在表尾插入最慢，在任意位置处插入时，平均需要寻找约一半的结点，其时间复杂度为 $O(n)$。

借助以上定位插入元素的方法，可便捷地使用单链表类逆序创建一个元素依次为 1～n 的单链表，实现代码如下。

```
mlist = LList()                    #实例化一个空表 mlist
n = int(input())                   #n 个元素，值依次为 1,2,3,…,n
for i in range(n, 0, -1):          #逆序，在表头插入元素
    mlist.insert(0, i)
```

定位删除元素：定位删除元素操作即要求删除表中第 i 个元素对应的结点，并返回该元素值。该操作的前提是：表非空，且 i 是 1 和表长之间的任意值。同样，在链表中删除元素时，也不需要移动已有的元素，只需要定位到待删除结点的前驱结点，然后重置其后继结点为待删除结点的后继结点即可。定位删除元素操作的代码实现参考代码清单 3.1 中的实例方法 delete。针对不同位置的操作复杂度也不同。

假定传入的 i 值合法，当 i 等于 1 时，表示要删除当前的表头元素，则只需取出表头元素值，并将表头变量后移一位即可，实现过程如图 3.14 所示。

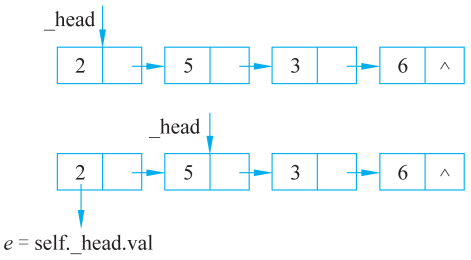

图 3.14　删除单链表表头结点

而对于一般的情况，即 i 大于 1 且小于或等于表长时，则先需要通过从 1 开始的计数来找到第 $i-1$ 个结点，并令 p 指向该结点；然后，通过设置 $p.next=p.next.next$ 的方式实现将第 i 个结点删除的目的。以删除这个单链表的第 4 个结点为例，其实现过程如图 3.15 所示。

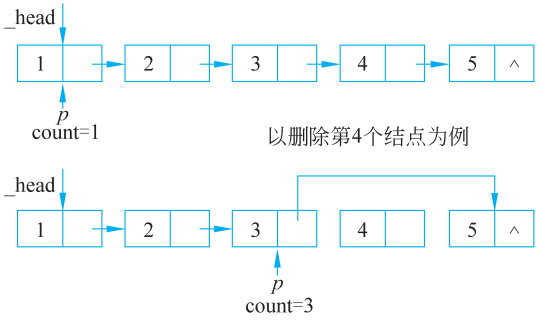

图 3.15　删除单链表第 i（i 大于 1 且小于或等于表长）个结点

从以上分析可以得出，对于单链表的删除操作，仍然是在表头删除最快，其时间复杂度为 $O(1)$，在表尾删除最慢，在任意位置处删除元素，平均也需要寻找约一半的结点，其时间复杂度也为 $O(n)$。

代码清单 3.1　定义单链表 LList 类

```
#定义单链表结点类
class ListNode:
    def __init__(self, val, next_=None):
        self.val = val
        self.next = next_
```

```
class LList:
    def __init__(self, ls=None):          #根据传入的列表 ls 逆序创建单链表
        p = None
        for item in ls[::-1]:
            p = ListNode(item, p)
        self._head = p

    def is_empty(self):                   #用于判断表是否空
        return self._head is None

    def __len__(self):                    #计算表长并返回
        p = self._head
        count = 0
        while p is not None:
            count += 1
            p = p.next
        return count

    def __str__(self):                    #将单链表转换为列表形式的字符串
        p = self._head
        s = ""
        while p is not None:
            s += str(p.val) + ","
            p = p.next
        return "[" + s[:-1] + "]"

    def insert(self, i, val):             #在表中第 i 个结点后插入元素值为 val 的新结点
        q = ListNode(val)
        if i == 0:
            q.next = self._head
            self._head = q
            return
        p = self._head
        count = 1
        while count < i:
            p = p.next
            count += 1
        q.next = p.next
        p.next = q

    def delete(self, i):                  #删除表中的第 i 个结点,并将其元素值返回
        if i == 1:
            e = self._head.val
            self._head = self._head.next
            return e
        p = self._head
        count = 1
        while count < i - 1:
            p = p.next
            count += 1
        e = p.next.val
        p.next = p.next.next
        return e
```

3.3.4 单链表例题

例 3.3 （力扣 206）反转链表。

【题目描述】

给定单链表的头结点 head，设计算法反转链表，并返回反转后的链表。注意，空链表反转后仍为空链表；本题链表中结点的数目范围是[0，5000]，−5000≤Node.val≤5000。

例如，图 3.16 中显示了一个链表反转前后的形态。

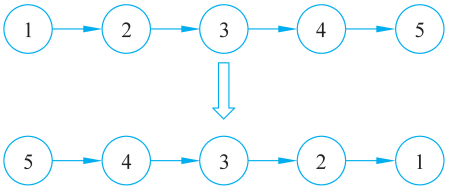

图 3.16 反转链表示例

【解题思路】

本题采用头部删除加头部插入策略，则无须创建额外空间，算法空间复杂度为 $O(1)$，实现过程如下。

（1）定义变量 p，用于记录结果链表的表头。初始时设置 p 为 None。

（2）遍历单链表 head，循环执行以下操作直至 head 为 None。

① 定义变量 q，用于指向当前待操作结点，每次循环开始都指向当前表头结点即可。

② 从单链表 head 中删除表头结点（即 head＝head.next，让表头变量 head 顺链后移一个结点即可），注意，此时表头结点仅剩变量 q 来指向。

③ 将 q 所指结点插到结果链表 p 的表头结点之前（即 q.next＝p），q 就成为结果链表新的表头。

④ 更新 p，让它始终指向结果链表的表头结点（即 $p=q$）。

链表反转过程如图 3.17 所示，每次从原链表中删除首结点，再将其插到结果链表的表头，从而实现链表的逆序。

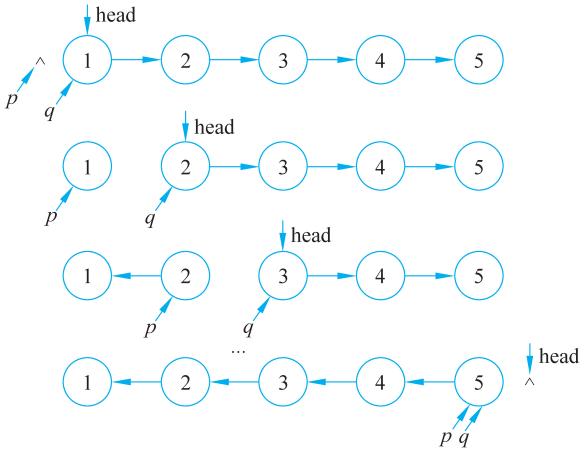

图 3.17 链表反转过程

【参考代码】

```
def reverseList(self, head: ListNode) -> ListNode:
    p = None
    while head is not None:
        q = head
        head = head.next
        q.next = p
        p = q
    return p
```

例 3.4 （力扣 LCR 136）删除链表的结点。

【题目描述】

给定单向链表的头指针 head 和一个要删除的结点的值 val，设计算法删除该结点。返回删除后的链表的头结点（题目保证链表中结点的值互不相同，且 val 一定存在于链表中）。

例如，图 3.18 显示了在一个单链表中删除值为 5 的结点前、后的形态。

【解题思路】

本题采用首先定位结点再修改引用的策略，实现过程为：遍历单链表 head，直到某一结点的元素域和所给 val 值相等，即可定位目标结点 cur。设结点 cur 的前驱结点为 pre，后继结点为 cur.next，则执行 pre.next＝cur.next，即可实现删除 cur 结点，删除过程如图 3.19 所示。

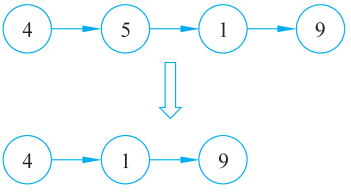

图 3.18　删除链表的结点示例

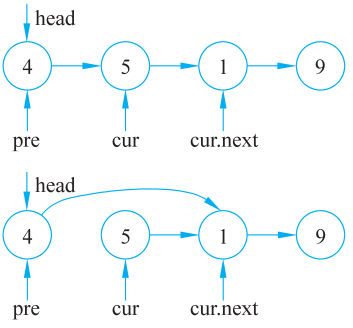

图 3.19　链表中删除给定值结点

可以通过从头开始遍历、逐个结点地查看其元素值是否为 val 的方式找到结点 cur，该操作时间复杂度为 $O(n)$。还有个关键问题是：如何确定 cur 的直接前驱结点 pre。因为在单链表中，可以通过结点的链接域以 $O(1)$ 的复杂度定位其直接后继，但是并无链接域以 $O(1)$ 的复杂度定位其直接前驱。若想找到其直接前驱，一个自然的思路也是从头开始遍历，逐个结点地查看其后继是否为指定结点 cur。显然，可以在查找结点 cur 的同时，亦步亦趋地记录下 cur 的前驱 pre。即开始时初始化 pre＝None 以及 cur＝head，然后，当每次 cur 需要顺链后移（即 cur＝cur.next）时，先让 pre＝cur，然后 cur 再后移。这样当找到了 cur 结点的时候，也记录下了其直接前驱 pre。

本题算法中这种以亦步亦趋的方式同时记录了当前结点及其前驱结点，并同步沿着链表向后边探索边移动的方法，称为“蠕动”，是解决单链表指定元素查找和删除的常用方法。

【参考代码】

```
def deleteNode(self, head: ListNode, val: int) -> ListNode:
    cur = head
    pre = None
    while cur.val != val:
        pre = cur
        cur = cur.next
    if not pre:
        head = head.next
    else:
        pre.next = cur.next
    return head
```

例 3.5 （力扣 24）两两交换链表中的结点。

【题目描述】

给定一个链表 head，设计算法两两交换其中相邻的结点，并返回交换后链表的头结点。注意：必须在不修改结点内部的值的情况下完成本题（即只能进行结点交换）。

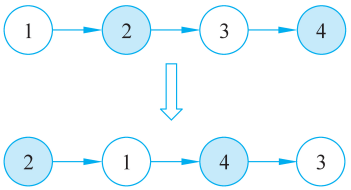

图 3.20 两两交换链表中的结点示例

例如，图 3.20 显示了两两交换一个单链表中的结点前、后的形态。

【解题思路】

可以从前往后依次对相邻的每对结点进行交换处理，为了让第一对结点也和后面的处理模式相同，可以在 head 前先设置一个虚拟的头结点 dummy_head，另设置一个 p 表示当前到达的结点，初始时 p＝dummy_head，每次需要交换 p 后面的两个结点。如果 p 的后面没有结点或者只有一个结点，则没有更多的结点需要交换，因此结束交换。否则，获得 p 后面的两个结点分别记作 a 和 b，通过更新结点的指针关系实现两两交换结点，交换过程如图 3.21 所示。

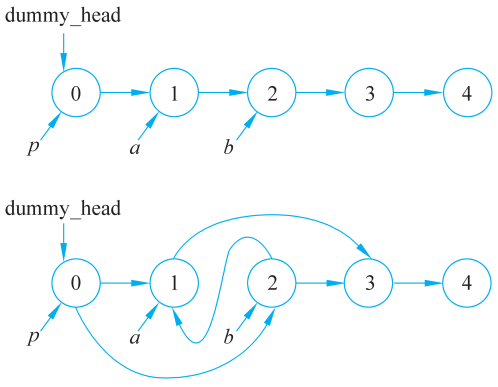

图 3.21 单次交换结点 p 后的两个结点 a、b 的过程

【参考代码】

```
def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:
    dummy_head = ListNode(0, head) #在头 head 前增加一个虚拟的头结点
    p = dummy_head
    while p.next and p.next.next:
```

```
        a = p.next                  #a 表示目前正在交换的结点对中的第一个结点
        b = p.next.next             #b 表示目前正在交换的结点对中的第二个结点
        a.next = b.next             #第一个结点先抓住第二个结点的后继
        b.next = a                  #然后第二个结点抓住第一个结点
        p.next = b                  #与前面连上
        p = a                       #准备处理下一对结点
    return dummy_head.next
```

例 3.6 （力扣 19）删除链表的倒数第 N 个结点。

【题目描述】

删除链表的倒数第 n 个结点，并且返回链表的头结点。链表长度范围为[1，30]，0≤Node.val≤100。

例如，图 3.22 显示了在一个单链表中删除 n 为 2 的结点（即倒数第 2 个结点）前、后的形态。

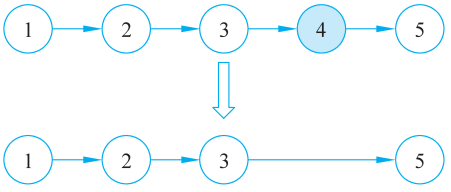

图 3.22 删除倒数第 2 个结点示例

【解题思路】

本题的核心问题是找到链表倒数第 n 个结点的前驱结点，一种容易想到的方法是，首先从头结点开始对链表进行一次遍历，得到链表的长度 L，随后再从头结点开始对链表进行一次遍历，当遍历到第 $L-n+1$ 个结点时，它就是需要删除的结点。

若想提高效率，也可以在不预先求出链表的长度的情形下通过一趟扫描解决本题，可以采用“快慢指针”的方法，用两个指针 fast 和 slow 同时对链表进行遍历，并且 fast 比 slow 超前 n 个结点，当 fast 遍历到链表的末尾时，slow 就恰好处于倒数第 n 个结点。而对于删除结点操作，关键是要找到待删除结点的前驱（即倒数第 $n+1$ 个结点），因此应设法让 fast 比 slow 超前 $n+1$ 个结点。同时，为了避免对“要删除的结点恰好是表头结点 head”这种特殊情况进行单独处理，可以先设置一个虚拟的头结点，指向该结点的指针为 dummy_head。快指针从 head 出发，走 n 步，接着让慢指针从 dummy_head 出发，这样，快指针比慢指针领先 $n+1$ 个结点。再令快慢指针一起同步移动，当快指针走到链表的末尾（即指向 None）时，慢指针指向的就是倒数第 n 个结点的前驱结点，如图 3.23 所示。

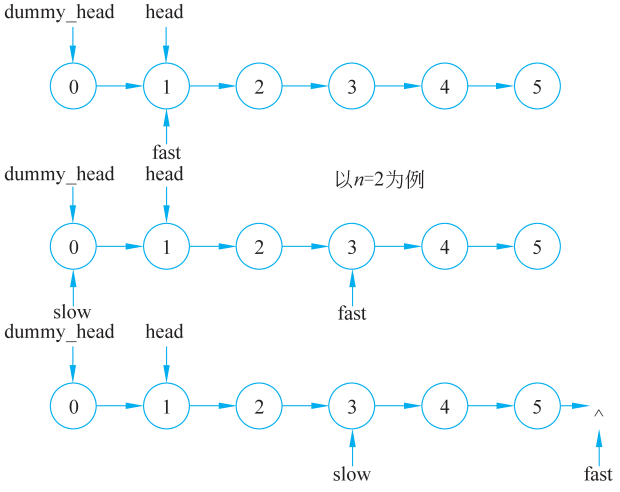

图 3.23 快慢指针法定位待删除结点前驱

【参考代码】

```
def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
    dummy_head = ListNode(0, head)
    fast, slow = head, dummy_head
    for i in range(n):
        fast = fast.next
    while fast:
        fast = fast.next
        slow = slow.next
    slow.next = slow.next.next
    return dummy_head.next
```

3.4 链表的变形与操作

观察 3.3 节介绍的单链表结构的线性表发现，由于有表头变量直接标记着头结点，所以在表头进行加入和删除等操作的时间复杂度均为 $O(1)$，但是，想要在单链表的尾端加入或删除元素效率却很低，因为需要先从头开始以顺链逐个结点遍历的方式来定位尾结点或其前驱，然后才能链接新结点或删除尾结点。在实际应用中，如果经常需要在表的两端进行一些变动操作，则可以通过一些链表的变形来提高操作的效率。

3.4.1 带尾结点引用的单链表

希望能够快速地定位到表尾，最直观的处理方式是可以直接给表对象增加一个对表尾结点的引用，这样在尾端加入元素，也能做到 $O(1)$ 的复杂度，如图 3.24 所示。

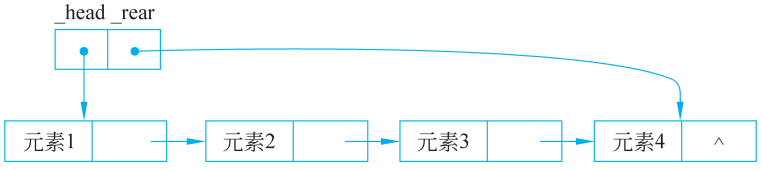

图 3.24 带有尾结点引用的单链表

定义带有首尾结点的单链表，可以通过继承 3.3 节中定义的单链表并在初始化操作中增加一个尾结点的设置来实现，也可以直接重新定义一个新的单链表类 LList1，并在初始化操作中分别设置首尾结点即可，具体定义方法如下。

```
class LList1:
    def __init__(self):
        self._head = None
        self._rear = None
```

链表的这一新的设计与之前单链表的结构近似，这种结构变化对非变动操作影响不大，一般只影响到表的变动操作，以下重点讨论带尾结点引用的单链表的首端和尾端的插入、删除操作的实现算法。

首端插入：在链表非空时，前端插入不影响_rear，但在表为空表时，要考虑给_rear 赋值，如图 3.25 所示。

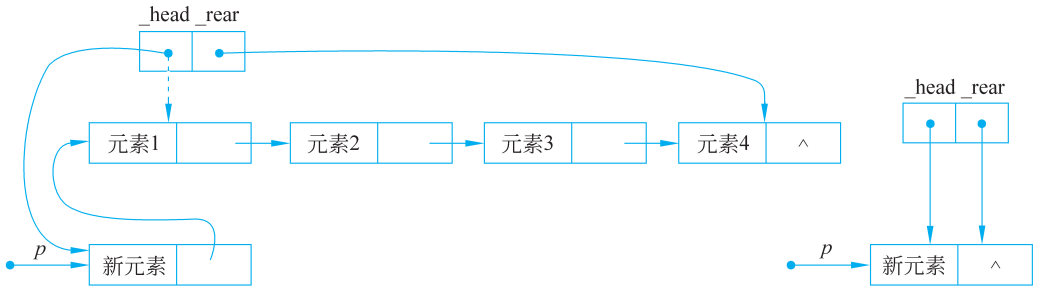

图 3.25　带有尾结点引用的单链表的首端插入

该首端插入元素操作时间复杂度为 $O(1)$，完整代码如下。

```
def prepend(self, val):
    if self._head is None:
        self._head = ListNode(val)
        self._rear = self._head
    else:
        self._head = ListNode(val, self._head)
```

尾端插入：在链表非空时，将尾结点的后继指向新结点，并更新_rear 指向新的尾结点，如图 3.26 所示。但在表为空表时，仍然要考虑给_rear 赋值，应使用和前端插入一样的操作。

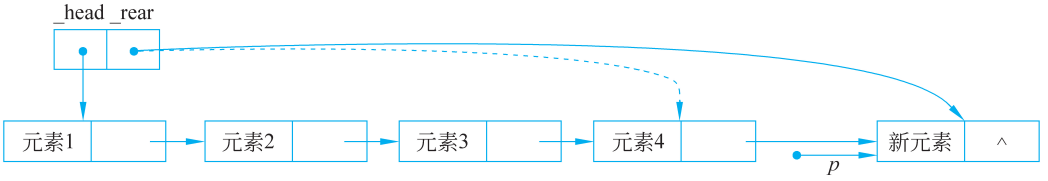

图 3.26　带有尾结点引用的单链表的尾端插入

该尾端插入元素操作时间复杂度也为 $O(1)$，完整代码如下。

```
def append(self, val):
    if self._head is None:
        self._head = ListNode(val)
        self._rear = self._head
    else:
        self._rear.next = ListNode(val)
        self._rear = self._rear.next
```

首端删除：假定链表非空，首端删除很简单，只需取出表头元素，并重置表头指针即可，如图 3.27 所示。

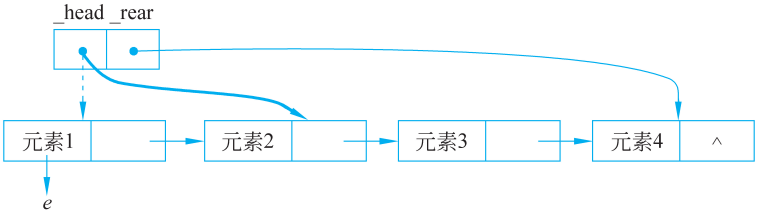

图 3.27　带有尾结点引用的单链表的首端删除

该首端删除元素操作时间复杂度为 $O(1)$，完整代码如下。

```
def pop(self):
    e = self._head.val
    self._head = self._head.next
    return e
```

尾端删除：假定链表非空，尾端删除相对比较复杂，需要先记录下表尾结点的元素值，还需要找到倒数第二个结点，也就是表尾结点的直接前驱，并将它设为新的表尾，这种处理需要表中至少有两个结点。当表中只有一个结点时，删除结点就相当于把表变为空表，而判断是否为空表只需要看表头变量的值是否为 None，所以此时将_head 置为 None 即可；而当原表不止一个结点的时候，就需要寻找表尾的前驱。依旧只能从表头开始，一个个结点进行检查，直至某个结点的后继是_rear 为止，如图 3.28 所示。

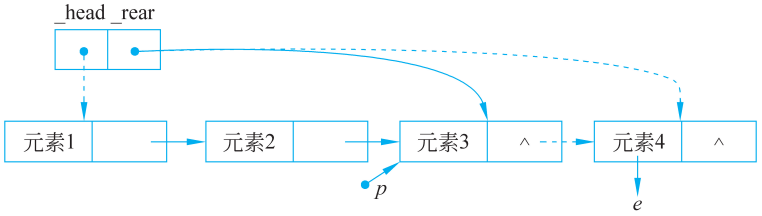

图 3.28　带有尾结点引用的单链表的尾端删除

该尾端删除元素的操作由于要从头遍历查找表尾的前驱，因此时间复杂度为 $O(n)$，完整代码如下。

```
def pop_last(self):
    e = self._rear.val
    if self._head.next is None:  #表中只有一个结点
        self._head = None
    else:
        p = self._head
        while p.next is not self._rear:
            p = p.next
        self._rear = p
    return e
```

3.4.2　循环单链表

单链表还有一种常见的变形设计也能够实现表头、表尾的快速定位，即循环单链表，简称循环链表，只需要将原来单链表中表尾结点的 next 域由 None 改为指向表的第一个结点即可，如图 3.29 所示。

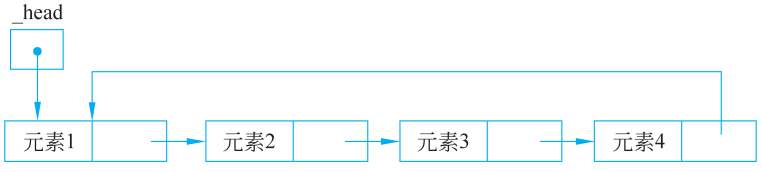

图 3.29　标识头结点的循环单链表