第3章

线 性 表

在程序中经常要对一组同类型的数据元素进行整体管理和使用。例如电话号码、学生成绩、商品记录等，最简单、有效的方法是将它们放在一个线性表中。线性表是最基本的数据结构，它不仅有着广泛的应用，也是其他数据结构的基础。本章介绍普通线性表，第4章和第5章将分别介绍两种特殊的线性表——栈和队列。


视频讲解

3.1 线性表的基本概念

线性表简称表，是由n(n≥0)个数据元素构成的有限序列，其中n称为线性表的表长。当n=0时，表长为0，表中没有元素，称为空线性表，简称空表；当n>0时，线性表是非空表，并记为$L=(a_0, a_1, a_2, \cdots, a_i, \cdots, a_{n-1})$，其中每个元素有一个固定的位序号，如元素$a_0$的位序号是0，$a_i$的位序号是i。

例如，线性表A=(5, 3, 2, 9)中包含了4个整数，这4个元素的位序依次为0、1、2、3；图书管理系统中的图书清单也是一个线性表，其中每个数据元素是确定的一本书，每本书在图书清单中有一个确定的位序。又如，一个字符串是由字符构成的线性表，一篇文章是由单词构成的线性表，一个菜谱是由操作指令构成的线性表，一个文件是由磁盘上的数据块构成的线性表。可见线性表中元素之间的次序非常重要，如果打乱，这些表就毫无意义。因此，必须强调线性表中的数据元素之间存在前驱和后继的关系，除了首元素a_0以外，每个元素都有一个直接前驱；除了尾元素a_{n-1}以外，每个元素都有一个直接后继。元素之间的关系为一对一的线性关系，线性表的逻辑结构是线性结构，如图3.1所示。

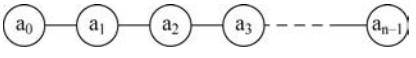


图3.1 线性表的逻辑结构示意图

如果线性表中的元素值随着位序的增大递增或递减，则该线性表称为**有序表**；如果元素值的大小和位序之间没有特定关系，则该线性表称为**无序表**。本书主要讨论更普通的没有明确是否有序的线性表，而将有序表看成普通线性表的一个特例。

可将线性表定义为一个二元组：List=(D, R)，其中 D 是数据元素的有限集合，R 是 D 上关系的有限集合，则 $D=\{a_i \mid 0\leqslant i<n,\ n\geqslant 0\}$，$R=\{<a_i,\ a_{i+1}> \mid 0\leqslant i<n-1\}$。

线性表具有以下特点。

(1) 有穷性：线性表中的元素个数是有限的。

(2) 同构性：一般来说，线性表中的所有元素具有相同的性质，即具有相同的类型。如果数据元素的性质不同，通常不具有实际应用意义。

(3) 不同类型元素构成的线性表，例如一个整数线性表和一个图书清单，虽然应用场合不同，但其元素之间的逻辑关系和基本操作是相同的。

3.2 线性表的抽象数据类型

抽象数据类型从数据结构的逻辑结构及可对其进行的基本操作两个方面进行定义。

T 类型元素构成的线性表是由 T 类型元素构成的有限序列，并且具有以下基本操作：

(1) 创建一个空线性表(__init__)。

(2) 判断线性表是否为空(empty)。

(3) 求出线性表的长度(__len__)。

(4) 将线性表清空(clear)。

(5) 在指定位置插入一个元素(insert)。

(6) 删除指定位置的元素(remove)。

(7) 获取指定位置的元素(retrieve)。

(8) 用指定元素替换线性表中指定位置处的元素(replace)。

(9) 判断指定元素 item 在线性表中是否存在(contains)。

(10) 对线性表中的每个元素进行遍历(traverse)。

在以上 ADT 描述中，线性表元素的逻辑关系可以从“序列”这个关键描述中得到，即元素之间具有次序关系；线性表的基本操作则定义了 10 种，可以根据实际情况增加或减少。从定义的操作来看，线性表具有以下特性：

(1) 线性表是动态的结构，可以进行元素的插入或删除，长度可以变化。

(2) 线性表的插入、删除、读/写等主要操作基于位序进行。

在 Python 的内置数据类型中，列表(list)、元组(tuple)和字符串(str)元素之间的关系都是线性关系，它们的逻辑结构都属于线性表；从数据元素类型来看，Python 列表和元组中的数据元素类型允许各不相同，而 Python 字符串限定了表中的数据元素为单个字符；从基本操作角度来看，只有 Python 列表实现了线性表 ADT 中的全部方法，而 Python 元组不可以改变，不能进行修改、添加、删除元素等操作，只能按位序访问；

Python 字符串的基本操作则主要是字符串的特有操作,因此 Python 元组和字符串与上述 ADT 并不一致。

下面借助 Python 语言的 abc 模块定义抽象类 AbstractList,用于描述线性表的抽象数据类型。

```
from abc import ABCMeta, abstractmethod
class AbstractList(metaclass = ABCMeta):
    """抽象表类,metaclass = ABCMeta 表示 AbstractList 类为 ABCMeta 的子类"""

    @abstractmethod
    def __init__(self):
        """初始化线性表"""

    @abstractmethod
    def empty(self):
        """判断表是否为空"""

    @abstractmethod
    def __len__(self):
        """返回表中元素的个数"""

    @abstractmethod
    def clear(self):
        """清空表"""

    @abstractmethod
    def insert(self, i, item):
        """在表中的 i 号位置插入元素 item"""

    @abstractmethod
    def remove(self, i):
        """删除 i 号位置的元素"""

    @abstractmethod
    def retrieve(self, i):
        """获取 i 号位置的元素"""

    @abstractmethod
    def replace(self, i, item):
        """用 item 替换表中 i 号位置的元素"""

    @abstractmethod
    def contains(self, item):
        """判断表中是否包含元素 item"""
```

```
    @abstractmethod
    def traverse(self):
        """输出表中的所有元素"""
```

在抽象数据类型定义中并没有规定其在计算机中的具体实现，但要真正让线性表在程序中发挥作用，必须对线性表进行存储，需要定义实现以上抽象类的普通类。在第 2 章中提到数据结构有两类最常见的存储结构——顺序存储结构和链式存储结构，以下分别介绍。

3.3 线性表的顺序存储及实现


视频讲解

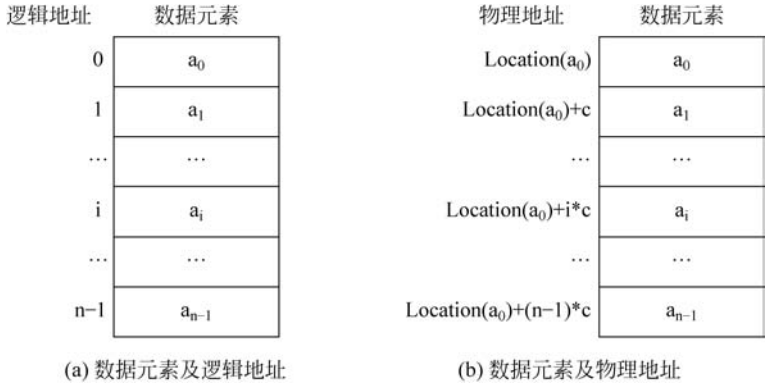
3.3.1 线性表顺序存储的基本方法

线性表的顺序存储方案是将表中的所有元素按照逻辑顺序依次存储在一块连续的存储空间中。表中的首元素存入存储区的开始位置，其余元素依次顺序存放，因此具有前驱、后继关系的两个元素，其内存映像也是相邻的，即元素之间物理位置的相邻直接反映它们逻辑关系的前后。线性表的顺序存储结构简称**顺序表**。

根据高级语言存储对象方法的不同，线性表的顺序存储结构可分为两类，即元素内置的顺序表和元素外置的顺序表。

1. 元素内置的顺序表

如果元素直接存储在连续存储区里，称为元素内置的顺序表。绝大多数高级语言提供元素内置的顺序存储方式，例如 C、C++ 和 Java 语言中的数组及 Python 语言中的字符串都采用此类存储方式，如图 3.2 所示。



(a) 数据元素及逻辑地址　　(b) 数据元素及物理地址

图 3.2　元素内置的顺序表

在该方案下，由于线性表的每个元素类型相同，所需存储量相同，所以顺序表中任一元素的位置都可直接计算出来，元素 a_i 的地址的计算公式为：

$$Location(a_i) = Location(a_0) + c * i$$

其中，c 是一个元素的存储量。

2. 元素外置的顺序表

如果连续存储区中存储的是每个线性表元素的地址,元素对象存放在该地址指示的内存单元中,则称为元素外置的顺序表。Python 语言的列表和元组的存储即基于此存储方式,如图 3.3 所示。

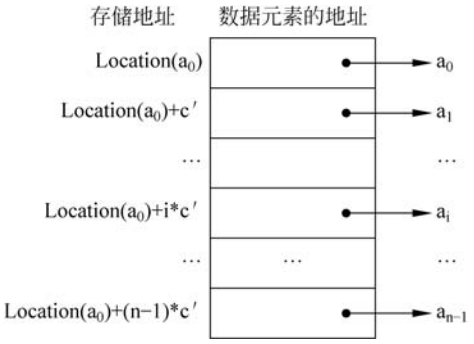


图 3.3 元素外置的顺序表

此时元素 a_i 的地址存放位置的计算公式为:

$$\text{Location}(a_i) = \text{Location}(a_0) + c' * i$$

其中,c'是一个地址所占的存储量,$\text{Location}(a_i)$位置存储的是对象 a_i 的引用(地址),而不是对象本身。

在 Python 语言中,对象的引用即对象的存储地址,对应于 2.1.1 节中提到的指针这个概念,而在 C 和 C++语言中可直接用指针类型的变量来存储对象的地址。在本书以后的章节中统一用术语“指针”来描述对象的地址。

因此,不管是元素内置的顺序表还是外置的顺序表,根据元素位序号可以直接定位到元素的地址,对元素的存取操作可以在 O(1)时间内完成,故称顺序表具有随机存取(random access)或直接存取的特性。

3.3.2 Python 列表的内部实现

Python 列表是一种基于元素外置存储的顺序表,图 3.4 是 Python 列表的存储示意图。一个列表对象包含引用计数(ob_refcnt)、类型(ob_type)、列表所能容纳的元素个数(allocated)、变长对象的当前长度(ob_size)以及列表元素容器的首指针(ob_item)等信息。列表元素容器是一块连续的内存块,依次存储指向列表中各个数据元素的指针,因此列表元素容器是元素外置的顺序结构。因为每个列表元素也是一个包含类型等信息的完整结构,所以一个 Python 列表中各个元素的类型可以不同。

3.1 节和 3.2 节定义的线性表可以直接用 Python 语言的列表来进行顺序存储。例如 3.1 节所述的线性表 A=(5, 3, 2, 9),在 Python 语言中可以直接将其表示为一个列表,如 alst=[5, 3, 2, 9]。

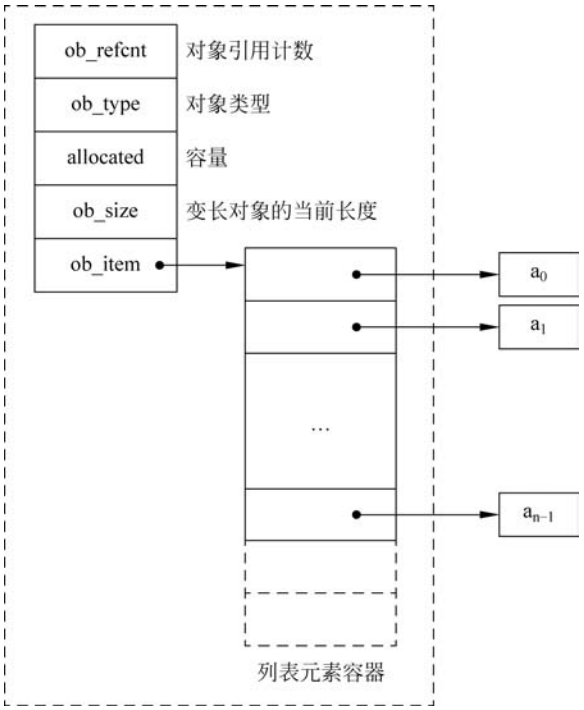


图 3.4 Python 列表的存储示意图

接下来通过一个例子分析列表元素增加时 Python 列表的空间递增机制。下列代码为初始为空的列表 lst,循环 500 次依次在 lst 的尾部添加一个元素。若添加元素后发现表的容量改变,则输出当前列表的长度、列表所占的字节数、列表元素容器的当前容量和增量等信息。

```
import sys
lst = []
empty_size = b = sys.getsizeof(lst)
count = 0
print("列表长度 %4d,  总占用字节数 %4d" % (0, b))
for k in range(500):
    lst.append(None)
    a = len(lst)
    old_b = b
    b = sys.getsizeof(lst)
    if b != old_b:
        print("列表长度 %4d,  总占用字节数 %4d, "
              " 表元素容器大小 %4d,  增加字节数:%4d"
              % (a, b, (b - empty_size) / 8, (b - old_b) / 8))
        count += 1
print("扩容总次数:", count)
```

在 Python 3.8 的 64 位字长环境下执行以上代码,输出如下:

```
列表长度    0,  总占用字节数   56
列表长度    1,  总占用字节数   88,  表元素容器大小    4,  增加大小:   4
列表长度    5,  总占用字节数  120,  表元素容器大小    8,  增加大小:   4
列表长度    9,  总占用字节数  184,  表元素容器大小   16,  增加大小:   8
列表长度   17,  总占用字节数  256,  表元素容器大小   25,  增加大小:   9
列表长度   26,  总占用字节数  336,  表元素容器大小   35,  增加大小:  10
列表长度   36,  总占用字节数  424,  表元素容器大小   46,  增加大小:  11
列表长度   47,  总占用字节数  520,  表元素容器大小   58,  增加大小:  12
列表长度   59,  总占用字节数  632,  表元素容器大小   72,  增加大小:  14
列表长度   73,  总占用字节数  760,  表元素容器大小   88,  增加大小:  16
列表长度   89,  总占用字节数  904,  表元素容器大小  106,  增加大小:  18
列表长度  107,  总占用字节数 1064,  表元素容器大小  126,  增加大小:  20
列表长度  127,  总占用字节数 1240,  表元素容器大小  148,  增加大小:  22
列表长度  149,  总占用字节数 1440,  表元素容器大小  173,  增加大小:  25
列表长度  174,  总占用字节数 1664,  表元素容器大小  201,  增加大小:  28
列表长度  202,  总占用字节数 1920,  表元素容器大小  233,  增加大小:  32
列表长度  234,  总占用字节数 2208,  表元素容器大小  269,  增加大小:  36
列表长度  270,  总占用字节数 2528,  表元素容器大小  309,  增加大小:  40
列表长度  310,  总占用字节数 2888,  表元素容器大小  354,  增加大小:  45
列表长度  355,  总占用字节数 3296,  表元素容器大小  405,  增加大小:  51
列表长度  406,  总占用字节数 3752,  表元素容器大小  462,  增加大小:  57
列表长度  463,  总占用字节数 4264,  表元素容器大小  526,  增加大小:  64
扩容总次数: 21
```

分析运行结果可以得出结论:这里的总占用字节数不包括列表中每个元素对象所占的空间,即只包含图 3.4 虚线框中的两部分——列表对象头部和列表元素容器部分。

(1) 空表占 56 字节,即只包含列表对象头部。

(2) 当添加第 1 个元素时,列表元素容器获得 4 个连续空间,可容纳 4 个对象的地址,每个地址为 8 字节,因此增加 32 字节,总空间为 88 字节,这些空间接下来依次存放第 2、3、4 个元素的地址。

(3) 当添加第 5 个元素时,列表元素容器的容量从 4 翻倍为 8,即增加 32 字节,总空间为 120 字节,这些空间接下来依次存放第 6、7、8 个元素的地址。

(4) 当添加第 9 个元素时,列表元素容器的容量从 8 翻倍为 16。

(5) 当添加第 17 个元素时,列表元素容器的容量从 16 扩大至 25。

(6) 当添加第 26 个元素时,列表元素容器的容量从 25 扩大至 35。

(7) 当添加第 36 个元素时,列表元素容器的容量从 35 扩大至 46,以此类推。

也就是说,列表元素容器的空间是动态增长的,若当前空间不够,则需要进行扩容。那扩容到多大呢?一般有两种策略,即增量策略和翻倍策略。增量策略是对当前容量增加一个数值,而翻倍策略是将当前容量乘以 2。从以上实验来看,Python 列表在空间较小时空间增长采用翻倍机制,而在空间较大时采用增量策略,即依次增加 9、10、11、12、14、16、18、…个空间。运行结果的最后一行表明,从空表开始生成长度为 500 的表共需要

21 次空间扩容。若从空表开始生成长度为 100000 的表，则需要 65 次空间扩容。因此，扩容操作的次数随着表长 n 的增长非常缓慢地增长，当 n 很大时，将扩容时间均摊到每个 append 操作中，所花费的时间可以忽略，故每个 append 操作的摊销时间复杂度仍为 O(1)。

接下来讨论如何自定义顺序表类，并实现线性表抽象数据类型中的所有操作。

3.3.3 基于 Python 列表的实现

假设定义一个自定义类 PythonList，它使用 Python 的列表 list 存储线性表的数据，并封装 AbstractList 类中的所有操作。PythonList 类拥有 3.2 节中 AbstractList 抽象类的所有性质和方法，将它定义为 AbstractList 类的派生类，并实现其全部的方法。参考代码如下：

```
from AbstractList import AbstractList
class PythonList(AbstractList):
    def __init__(self):
        self._entry = []

    def __len__(self):
        return len(self._entry)

    def empty(self):
        return not self._entry

    def clear(self):
        self._entry = []

    def insert(self, i, item):
        self._entry.insert(i, item)

    def remove(self, i):
        self._entry.pop(i)

    def retrieve(self,i):
        return self._entry[i]

    def replace(self,i, item):
        self._entry[i] = item

    def contains(self, item):
        return item in self._entry

    def traverse(self):
        print(self._entry)
```

PythonList 类中的每个方法分别仅包含一个调用 list 内置操作的语句，注意这些方法的时间复杂度并不都是 O(1)，因为有些 list 的内置操作并不是原操作。例如 contains 方法，它调用的 in 方法不是原操作，需要进行元素的重复比较，最坏情况下需比较 n 次，

因此时间复杂度为 O(n)。更多 list 内置操作的时间复杂度可以参考 2.3.2 节的表 2.6。


视频讲解

3.3.4 基于底层 C 数组的实现

接下来利用 ctypes 模块提供的底层 C 数组实现线性表的顺序存储，定义一个自定义类 DynamicArrayList 来模拟一个顺序表(ctypes 模块的介绍见 1.3.2 节)。底层 C 数组对应于内存中的一块容量确定的连续空间，线性表元素依次直接存放在该数组中，即以元素内置方式顺序存储。DynamicArrayList 类的定义框架如下：

```
import ctypes
from AbstractList import AbstractList
class DynamicArrayList(AbstractList):
    def __init__(self):
    def empty(self):
    def __len__(self):
    def clear(self):
    def insert(self, i, item):
    def remove(self, i):
    def retrieve(self, i):
    def replace(self, i, item):
    def contains(self, item):
    def append(self, item):
    def traverse(self):
    def __str__(self):
    def _make_array(self, cap):
    def _resize(self, cap):
```

在类中除了包含 3.2 节所描述的线性表抽象数据类型定义中的方法以外，还包含了一些其他方法，例如_make_array(cap)、_resize(cap)、append(item)和__str__()等。其中，_make_array(cap)方法的功能是返回一个容量为 cap 的数组。由于底层 C 数组的容量是固定的，在对线性表不断插入之后数组空间很可能耗尽而无法插入，为此采用空间动态增长的方式，即一旦空间用完就向内存重新申请一块更大的空间。_resize(cap)方法的功能是将当前数组空间扩容至 cap。_make_array(cap)和_resize(cap)是供类的其他方法调用的保护方法。append(item)和__str__()是为了方便在表尾插入和输出元素而增加的方法，在具体实现时读者可以自行取舍。

为了做到数据封装和信息隐藏，定义类时在实例属性变量前加了下画线，用于标明是受保护(protected)的变量，原则上外部类不允许直接访问它；在类中还有部分以小写命名且加前导下画线的保护方法。为简化文字描述，在本书中对保护属性的变量或方法，其前导下画线只在程序代码中严格保留，在不发生歧义的情况下，文字描述中将略去前后下画线。

假设用一个当前容量为 capacity 的 entry 数组存储线性表的元素，并用 cur_len 记录当前线性表的长度，则线性表元素占用了 entry 数组中从 0 至 cur_len-1 的位置，如图 3.5 所示。

图 3.5 底层 C 数组实现的顺序表

接下来依次介绍 DynamicArrayList 类的各常用方法的具体实现。

1. 初始化空表的方法

DynamicArrayList 类初始化空表的方法 init 需要对 3 个成员变量 capacity、entry 和 cur_len 分别赋值，在 init 方法中这 3 个实例变量前都加了前导下画线。

初始容量 capacity 可由用户设定；entry 数组则是通过调用保护方法 make_array() 获得的一个容量为 capacity 的数组；初始化空表时，表长 cur_len 为 0。初始化后顺序表的状态如图 3.6 所示。

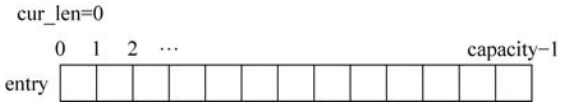


图 3.6 空顺序表

算法如下：

```
def __init__(self, cap = 0):
    """初始化一个空表"""
    super().__init__()
    self._cur_len = 0                                   #线性表元素个数的计数
    self._capacity = cap                                #当前数组容量
    self._entry = self._make_array(self._capacity)      #存放所有表元素的数组
```

2. 生成一个容量固定的底层 C 数组

```
def _make_array(self, cap):
    """保护方法,返回一个容量为 cap 的 py_object 数组"""
    return (cap * ctypes.py_object)()
```

3. 判别线性表是否为空

```
def empty(self):
    return self._cur_len == 0
```

4. 求线性表的长度

```
def __len__(self):
    """返回线性表中元素的个数"""
    return self._cur_len
```

5. 清空线性表

```
def clear(self):
    self._capacity = 0
    self._cur_len = 0
```

在 empty、__len__和 clear 这 3 个算法中都只含有原操作语句且不含有循环,算法的时间复杂度都为常量阶 O(1)。

6. 将元素 item 添加到线性表的尾部

如果 entry 数组还有空余空间,即 cur_len 小于 capacity,只需将 item 放在 cur_len 位置,并且 cur_len 增 1。如果当前数组空间已满,即 cur_len 等于 capacity,再插入元素会发生**上溢出**(overflow)。为防止上溢出,调用 resize 保护方法对线性表容量进行扩充,这里采用翻倍策略,即将数组容量乘以 2。

```
def append(self, item):
    """将元素 item 添加到线性表的尾部"""
    if self._cur_len == self._capacity:          #如果线性表的空间已用完
        if self._capacity == 0:
            cap = 4
        else:
            cap = 2 * self._capacity
        self._resize(cap)                        #给线性表扩一倍空间
    self._entry[self._cur_len] = item            #将 item 存储到表尾位置
    self._cur_len += 1                           #表长增 1
```

7. 数组的扩容

为了将线性表容量扩至 cap,首先调用 make_array 方法生成容量为 cap 的数组 temp,然后将数组 entry 中的元素复制到 temp 中,最后启用新数组 temp 存放表元素并将 capacity 调整为 cap。

```
def _resize(self, cap):                          #保护方法
    """将数组空间扩至 cap"""
    temp = self._make_array(cap)                 #生成新的更大的数组 temp
    for k in range(self._cur_len):               #将原线性表中的元素复制到新数组 temp 中
        temp[k] = self._entry[k]
    del self._entry
    self._entry = temp                           #启用新数组 temp 存放线性表元素
    self._capacity = cap                         #当前线性表的容量为 cap
```

resize 方法中的循环语句依次复制线性表中的所有元素,算法的时间复杂度为 O(n)。append、insert 等方法在插入元素时若遇到 entry 数组的容量不够时,需调用

resize 方法。可以证明，在数组大小以倍数扩大时，n 次 append 操作的总运行时间为 O(n)，每个 append 操作的摊销时间为 O(1)。因此，如果空间管理得当，保证 resize 方法不被频繁调用，append 算法的时间复杂度可以达到 O(1)。

8. 将元素 item 插入表的 i 号位置

为了将 item 插入表的 i 号位置，需要将表尾 cur_len－1 号至 i 号位置的每个元素向后移一个位置，然后在空出来的 i 号位置上存放 item，再将 cur_len 加 1，如图 3.7 所示。

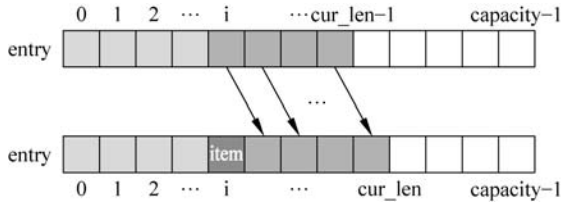


图 3.7　顺序表下的插入操作

另外，当 i 的值不合法，即不满足 0≤i≤self._cur_len 时，算法抛出异常；当数组容量用完时，则调用 resize 方法将数组空间扩容一倍。算法如下：

```
def insert(self, i, item):
    """将元素 item 插入表的 i 号位置"""
    if not 0 <= i <= self._cur_len:
        raise IndexError("插入位置不合法")
    if self._cur_len == self._capacity:          # 如果线性表的空间已用完
        if self._capacity == 0:
            cap = 4
        else:
            cap = 2 * self._capacity
        self._resize(cap)                        # 给线性表扩容一倍空间
    for j in range(self._cur_len, i, -1):        # 线性表尾部至 i 号位置的所有元素后移
        self._entry[j] = self._entry[j - 1]
    self._entry[i] = item                        # 将新元素 item 放在 i 号位置
    self._cur_len += 1                           # 表长增 1
```

在上述插入算法中，关键操作是元素的向后移动。当 i＝0 时，发生最坏情况，此时元素的移动次数为 n；当 i＝n 时，发生最好情况，此时元素的移动次数为 0。因此，insert 算法最坏情况下的时间复杂度为 O(n)，最好情况下的时间复杂度为 O(1)。

接下来计算平均情况下的时间复杂度。由于元素 item 的插入位置 i 可以是 0～n 号的任一位置，假设插入在 i 号位置的概率为 p_i，此时元素的移动次数为 c_i，一般情况下假设插入在任一位置的概率相同，即 $p_i=\frac{1}{n+1}$，而 $c_i=n-i$，所以平均情况下的移动次数为：

$$\sum_{i=0}^{n} p_i c_i = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{n}{2}$$

因此,insert 算法平均情况下的时间复杂度也为 O(n)。

9. 删除 i 号位置的元素

为返回被删除元素,先将 i 号位置的元素暂存在 item 中,然后将从 i+1 号至表尾 cur_len-1 号的每个元素向前移一个位置,再将 cur_len 减 1,最后返回 item,如图 3.8 所示。

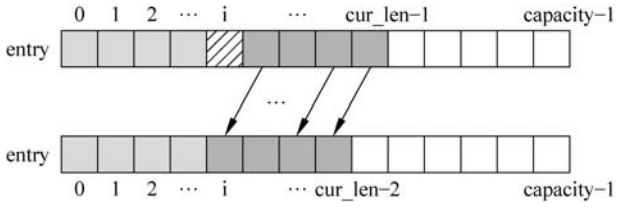


图 3.8 顺序表下的删除操作

另外,在算法开始处首先检查表是否为空,如果为空,则发生下溢出;接着检查 i 的值是否合法,当不满足 0≤i<self._cur_len 时算法抛出异常。

```
def remove(self, i):
    if self.empty():
        raise Exception("underflow")
    if not 0 <= i < self._cur_len:
         raise IndexError("删除位置不合法")
    item = self._entry[i]
    for j in range(i, self._cur_len - 1):              #将找到位置之后的元素后移
        self._entry[j] = self._entry[j + 1]
    self._cur_len -= 1                                 #表长减 1
    return item                                        #返回被删除元素
```

在上述删除算法中,关键操作是元素的向前移动。当 i=0 时,发生最坏情况,此时元素的移动次数为 n-1;当 i=n-1 时,发生最好情况,此时元素的移动次数为 0 次。因此,remove 算法最坏情况下的时间复杂度为 O(n),最好情况下的时间复杂度为 O(1)。

接下来计算平均情况下的时间复杂度。由于删除位置 i 可以是 0 至 n-1 号的任一位置,假设在 i 号位置删除的概率为 p_i,此时元素移动的次数为 c_i,一般情况下在任一位置删除的概率相同,即 $p_i=\frac{1}{n}$,而 $c_i=n-i-1$,所以平均情况下的移动次数为:

$$\sum_{i=0}^{n-1} p_i c_i = \frac{1}{n}\sum_{i=0}^{n}(n-i-1) = \frac{n-1}{2}$$

因此,删除算法平均情况下的时间复杂度也为 O(n)。

10. 读取 i 号元素

此处的方法 retrieve 也可用 Python 中的特殊方法名__getitem__,以方便使用者可以用[ ]操作直接访问表中的 i 号元素,在本书中与抽象类型中的方法一致,方法名为 retrieve。

```
def retrieve(self, i):
    if not 0 <= i < self._cur_len:
        raise IndexError("元素读取位置不合法")
    return self._entry[i]
```

11. 将 item 值写入表的 i 号位置

元素的写入方法与读取方法 retrieve 基本相似。

```
def replace(self, i, item):
    if not 0 <= i < self._cur_len:
        raise IndexError("元素写入位置不合法")
    self._entry[i] = item
```

从上述第 10、11 这两个算法可以看出，根据位序 i 可以直接在 entry 数组中读/写元素，读/写算法的时间复杂度都为常量阶 O(1)，再次说明顺序存储结构具有随机存取的特性。

12. 判断指定元素 item 在表中是否存在

将 item 依次与 entry 数组中的每个元素进行比较，在最坏情况下 item 与表中的所有元素比较一次，contains 算法的时间复杂度为 O(n)。

```
def contains(self, item):
    for i in range(self._cur_len):
        if self._entry[i] == item:
            return True
    return False
```

13. 将线性表转换成字符串

定义特殊方法__str__，从而方便使用者调用 print 方法输出 DynamicArrayList 的对象，即依次输出线性表中的所有元素。该算法可以作为 traverse 操作的一种替代实现，其时间复杂度为 O(n)。

```
def __str__(self):
    """将线性表转换成字符串，用于输出线性表中的所有元素"""
    elements = ''.join(str(self._entry[c]) for c in range(self._cur_len))
    return elements
```

可以看到，Python 语言的列表和 C 语言的数组都可以用来实现线性表的顺序存储，并且其基本操作的实现和时间性能也大体一致。在具体存储和使用时，前者更加方便、灵活，但后者空间更加紧凑。Python 列表的本质就相当于一个指针数组，在后续章节中，凡是用列表实现的各个数据结构都可以用 C 语言数组或 array 模块中的数组

来实现。


视频讲解

3.4 线性表的链式存储及实现

顺序表利用连续的存储空间存储数据元素，通过元素物理位置的前后来反映元素之间的逻辑关系；而线性表的链式实现不需要使用连续内存空间，表中的数据元素可以存储在任意可用空间中。在链式结构中，元素在内存中的映像称为**结点**，结点包含元素值和指针域两大部分，其中，指针域用于记录其后继结点或前驱结点的地址。可见，链式结构通过显式的指针域来表明元素的前驱、后继关系。

线性表的链式实现结构简称为**链表**，根据结点中指针域的数目及尾部结点指针的定义，链表可以分为**单链表**、**双向链表**、**循环链表**等。最常见的链表形式是单链表。

3.4.1 单链表

单链表的每个结点包含两个域，结点结构如图 3.9 所示，其中 entry 域存储元素，next 域存储后继结点的指针。在 Python 语言中，entry 域存储的是元素的指针，而在其他语言(例如 C++)中，entry 域存储的是元素值。

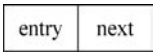


图 3.9 单链表结点结构

图 3.10 和图 3.11 为线性表(a_0，a_1，a_2，…，a_i，…，a_{n-1})对应的单链表结构示意图。其中，图 3.10 为 Python 语言中的单链表结构，图 3.11 为 C++等语言中的单链表结构。为简单起见，后续单链表的图示将统一简化为如图 3.11 所示的结构。在单链表中，每个结点的地址存储在前驱结点的指针域中，因此必须通过首元素结点(简称**首结点**)指针(设为 head)，依次顺序访问表中的元素。链表中的最后一个结点称为**尾结点**，尾结点指针域中的“Λ”表示不指向任何结点，即空指针，对应 Python 中的 None。当线性表为空表时，对应的单链表为空，即 head 为 None。

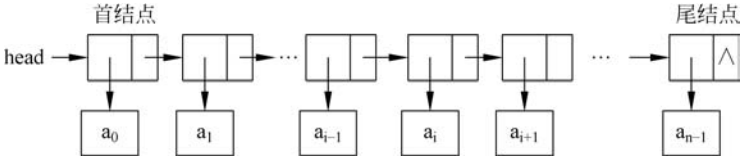


图 3.10 Python 语言中单链表的存储

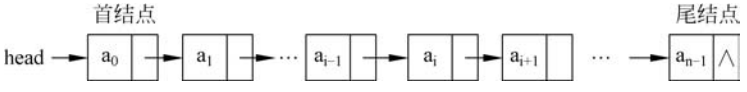


图 3.11 不带头结点的单链表示意图

为使操作更简单、方便，通常在单链表首结点前附加一个**表头结点**(简称**头结点**)，即为**带头结点的单链表**。在图 3.12 中，图(a)为非空线性表的单链表表示，图(b)为空线性表的单链表表示。通过指向头结点的指针(简称**头指针**)head 对整个链表进行操作，整个

链表即由头指针 head 代表。由于 head 指针始终指向头结点，它永不为空，所以空线性表和非空线性表的处理得到了统一。

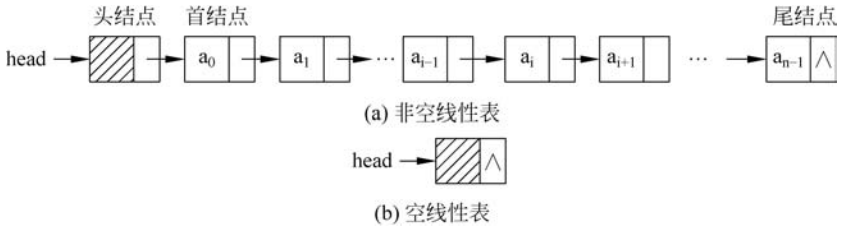


图 3.12 带头结点的单链表

首先定义结点类 Node 表示单链表中的一个结点。Node 类的初始化方法完成 entry 域和 next 域的赋值。由于链表类的各方法需要通过指针频繁地访问链表中的各个结点，所以将 entry 和 next 定义为公有属性，在定义时没有加前导下画线。

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

然后用 Python 语言实现一个带头结点的单链表。假设用自定义类 LinkedList 表示单链表，类中的各主要方法对应于 3.2 节所描述的抽象数据类型定义中的各个操作。LinkedList 类的定义框架如下：

```
from AbstractList import AbstractList
from Node import Node
class LinkedList(AbstractList):
    def __init__(self):
    def empty(self):
    def __len__(self):
    def clear(self):
    def insert(self, i, item):
    def remove(self, i):
    def retrieve(self, i):
    def replace(self, i, item):
    def contains(self, item):
    def traverse(self):
    def get_head(self):
```

接下来介绍 LinkedList 类中部分常用方法的具体实现。

1. 初始化空表的方法

对于一个带头结点的单链表，由于用 head 指针标识整个单链表，所以单链表类的数

据成员只有一个 head 变量。当初始化一个空表,产生如图 3.12(b)所示的单链表,即生成一个头结点,并由 head 指针指示。

```
def __init__(self):
    self._head = Node(None)
```

在__init__算法中只包含一个原操作语句,时间复杂度为 O(1)。

2. 判别线性表是否为空

```
def empty(self):
    return self._head.next is None
```

只需判别表头结点的指针域是否为空即可,时间复杂度为 O(1)。

3. 求线性表的长度

在单链表类定义中没有记录线性表元素个数的变量,无法直接获得表的长度,但可以通过活动指针从 head 之后开始移动并进行同步计数来间接求得。

在下列代码中,p 从首元素结点开始,count 为计数器,初始为 0,当 p 不为 None 时 count 增 1,p 往后移动,直到 p 为空为止。此时 count 即为表长。这是单链表的常见顺序操作模式,活动指针从头至尾移动一遍,时间复杂度为 O(n)。

```
def __len__(self):
    p = self._head.next
    count = 0
    while p:
        count += 1
        p = p.next
    return count
```

如果需要经常获取线性表的长度,可在类中增加一个变量记录表长,并在插入、删除等算法中对该变量进行维护,这样__len__算法的时间效率可提高至 O(1)。

4. 清空列表

```
def clear(self):
    p = self._head.next
    self._head.next = None
    while p:
        q = p
        p = p.next
        del q
```

此处将所有的结点依次进行人工回收，时间复杂度为 O(n)。由于 Python 语言中内存自动管理的机制，也可将算法改写为如下简单形式，仅将头结点的指针域设为 None。

```
def clear(self):
    self._head.next = None
```

算法的时间复杂度为 O(1)。Python 的垃圾回收器将自动回收引用计数为 0 的结点。

5. 读取 i 号元素

在单链表中只能通过 head 指针顺序向后依次访问每个元素，为定位 i 号元素，需要活动指针从 head 之后的首元素结点开始移动，并需要计数器从 0 开始同步计数，直至遇到第 i 号结点。因此，链表下元素存取的方式称为顺序存取，它不具有顺序表下随机存取的特点。

设活动指针 p 从 0 号结点开始，count 计数器初始为 0，如图 3.13 所示。

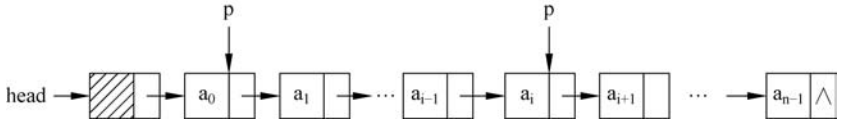


图 3.13　元素读取操作

当 p 不为 None 且 count＜i 时，p 往后移动，count 增 1，直至 p 为 None 或 count 为 i；若 p 不为 None，即 count＝i，则 p 为 i 号元素结点；若 p 为 None，说明 i 太大，不存在 i 号结点。另外，在算法开始处检查 i 是否小于 0。在最坏情况下，p 从头至尾移动一遍，时间复杂度为 O(n)。

```
def retrieve(self, i):
    if i < 0:
        raise IndexError("元素读取位置不合法,i 小于 0")
    p = self._head.next
    count = 0
    while p and count < i:
        p = p.next
        count += 1
    if p:
        return p.entry
    else:
        raise IndexError("元素读取位置不合法,i 太大,不存在 i 号元素")
```

6. 将元素 item 插入表的 i 号位置

将新结点插入表的 i 号位置，即插入在 i－1 号结点之后，要完成插入，必须有一个指针，假设为 previous，指向 i－1 号结点，如图 3.14 所示。

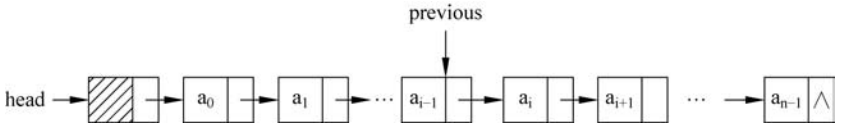


图 3.14　插入操作之前

previous 指向 i－1 号结点，接着生成一个值为 item 的新结点，然后将新结点 new_node 插入 previous 结点之后。图 3.15 给出了结点插入操作的示意，具体如下：

```
new_node = Node(item)
new_node.next = previous.next
previous.next = new_node
```

将 previous 定位至 i－1 号结点的方法与 retrieve 方法中的定位方法基本一致，只不过定位的结点换成了 i－1 号位置。

```
previous = self._head
count = -1
while previous and count < i - 1:
    previous = previous.next
    count += 1
```

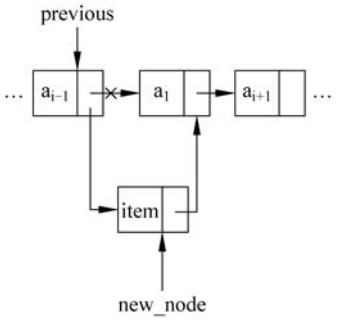


图 3.15　结点插入的操作

注意在本算法中 previous 和 count 的初始值从头结点开始，而不是从首结点开始。这样可以在 i＝0，即 item 插入为 0 号元素时也无须进行特殊处理，而直接将新结点插入在头结点之后。从这个算法可以看到头结点的作用之一就是可以使得对首元素的处理与对其他位置元素的处理一致，从而使算法得到简化。另外，如果上述循环结束时 previous 为空，说明 i－1 号结点不存在，不能在 i 号位置插入。

在单链表的插入算法中，关键操作是指针的向后移动，对于成功的插入，当 i＝0 时发生最好情况，时间复杂度为 O(1)；当在表尾插入时发生最坏情况，此时指针的移动次数为 n。i＞n 时为插入失败的最坏情况，此时指针的移动次数为 n＋1。因此，insert 算法的时间复杂度为 O(n)。具体实现算法如下：

```
def insert(self, i, item):
    if i < 0:
        raise IndexError("插入位置不合法,i 值小于 0")
    previous = self._head
    count = -1
    while previous and count < i - 1:
        previous = previous.next
        count += 1
    if previous is None:
        raise IndexError("插入位置不合法,i 太大")
    new_node = Node(item)
```

```
        new_node.next = previous.next
        previous.next = new_node
```

7. 删除 i 号位置的元素

如图 3.16 所示，在完成具体删除之前先将 current 指针定位在被删的 i 号结点上，从链表中删除 current 结点需要将其前驱结点的指针域与其后继结点相连接，因此另设指针 previous 定位至 i－1 号结点。

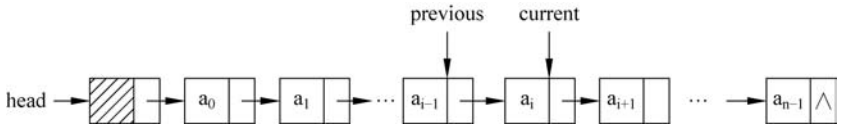


图 3.16 删除操作之前

接着 previous 结点的指针域指向 current 所指结点的后继结点。图 3.17 给出了结点删除操作的示意，具体如下：

```
previous.next = current.next
del current
```

将 previous 定位至 i－1 号结点的方法与 insert 方法中的定位方法一致。若 previous 最后为 None，说明定位 i－1 号结点失败，否则 current = previous.next；若 current 为 None，说明 i 号结点不存在。这两种情况都不能删除 i 号位置结点。具体实现算法如下：

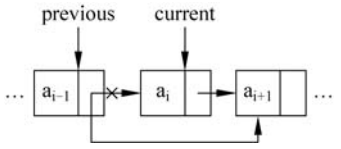


图 3.17 结点删除操作

```
def remove(self, i):
    if i < 0:
        raise IndexError("删除位置不合法,i 值小于 0")
    previous = self._head
    j = -1
    while previous and j < i - 1:
        previous = previous.next
        j += 1
    if previous is None:
        raise IndexError("删除位置不合法,不存在 i-1 号元素")
    current = previous.next
    if current is None:
        raise IndexError("删除位置不合法,不存在 i 号元素")
    previous.next = current.next
    item = current.data
    del current
    return item
```

单链表的删除算法与插入算法类似,关键操作是指针的向后移动,当 i=0 时发生最好情况,时间复杂度为 O(1);当删除表尾结点或 i 太大删除失败时发生最坏情况。remove 算法的时间复杂度为 O(n)。

从插入和删除算法的实现过程可知,链表是一种动态的存储结构。在插入元素时,向系统申请一个结点空间并加入链表;在删除元素时,在链表中删除对应结点并将结点空间归还给系统。由于结点空间动态分配和回收,链表实现不需要事先申请空间,不需要担心上溢出。在程序运行过程中,链表的规模可随时发生动态变化。

8. 获取头结点

有时候外部代码需要获得链表的头结点,所以增加以下 get_head 方法:

```
def get_head(self):
    return self._head
```

在实现单链表下的各基本操作时可以发现,对单链表任意结点的访问都必须从头结点或首结点开始顺序地向后操作,访问效率较低。一种可选的改进方法是在链表类定义中添加 current 指针指示最近访问的结点,同时增设 current_position 记录该结点的位序号。这样,当重复访问 current 结点或访问比 current_position 位序号大的结点时,不必再从 head 开始定位,而可以直接从 current 结点开始,结点访问的效率得到提高。这种方法只在按从前往后的次序对表结点进行访问时才能提高效率。


视频讲解

3.4.2 循环链表

如果将单链表的最后一个结点的指针域指向键表开始位置,就构成了**循环链表**。在具体实现时,也可像单链表一样,设计成带头结点或不带头结点。

图 3.18 是带头结点的单向循环链表示意图,图(a)和图(b)分别表示非空线性表和空线性表。

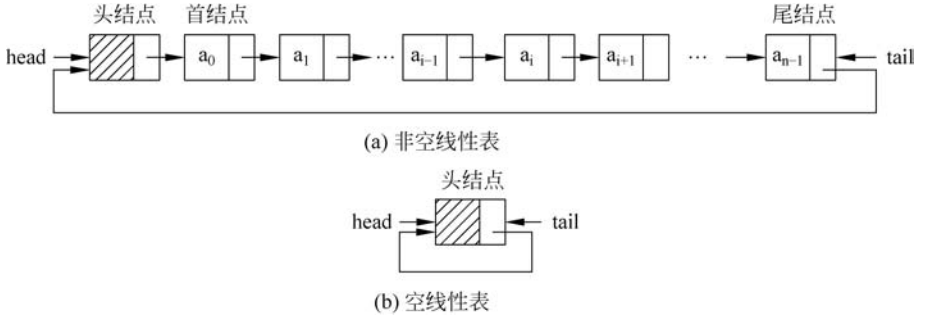


图 3.18 带头结点的单向循环链表

单向循环链表的结点与单链表结点的结构一致,结点类的定义可以直接共用。循环链表下各基本操作的实现方法也与普通单链表基本一致,二者的主要差别如下:

(1) 判别活动指针 p 是否到达表尾的条件不同。在循环链表中,p 到达表尾时

p.next＝head；而在单链表中，p 到达表尾时 p.next＝None。

（2）在循环链表中可设头指针 head，也可仅设尾指针 tail 标识一个链表，而在单链表中必须设头指针标识链表。

循环链表的特点如下：

（1）从任一结点出发都可访问到表中的所有结点。

（2）在用头指针表示的单循环链表中，首结点定位操作的时间性能是 O(1)，尾结点定位操作的时间性能是 O(n)。

（3）在用尾指针表示的单循环链表中，首结点和尾结点的定位都只需 O(1)的时间性能。

3.4.3　双向链表

如果每个结点不仅存储后继结点的指针，还存储前驱结点的指针，这样就形成了**双向链表**。双向链表的结点结构如图 3.19 所示。其中，entry 和 next 的含义跟单链表结点一致，而 prior 指针指向当前结点的前驱结点。

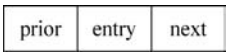


图 3.19　双向链表的结点结构

双向链表可以带头结点或不带头结点，可以是循环链表或不是循环链表。图 3.20 为带头结点的双向非循环链表；图 3.21 为带头结点的双向循环链表。

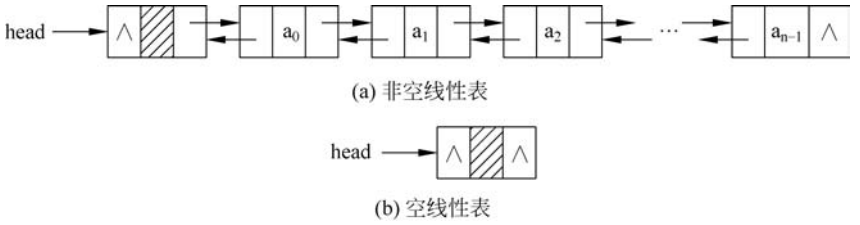


图 3.20　带头结点的双向非循环链表

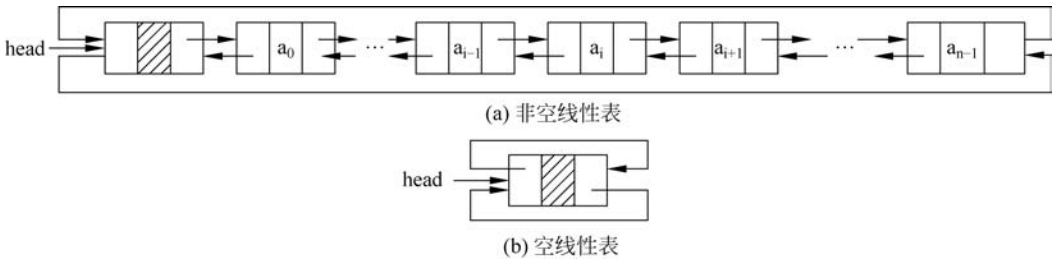


图 3.21　带头结点的双向循环链表

接下来分别介绍双向链表结点类和双向循环链表类的定义和实现。

1. 双向链表结点类

假设 DuNode 类表示双向链表中的一个结点。DuNode 类的初始化方法完成 entry 域、prior 域和 next 域的赋值。

```
class DuNode:
    def __init__(self, entry, prior = None, next = None):
        self.entry = entry
        self.prior = prior
        self.next = next
```

2. 双向循环链表类

1) 双向循环链表类及初始化方法

假设 DuLinkedList 类表示双向链表。初始化一个空线性表,即生成如图 3.21(b)所示的双链表结构,也即生成一个头结点,该结点由 head 指针指示,且该结点的前驱和后继指针域都指向自身。DuLinkedList 类只有一个数据成员 head。

```
from DuNode import DuNode
class DuLinkedList:
    def __init__(self):
        self._head = DuNode(None)
        self._head.next = self._head
        self._head.prior = self._head
```

2) 双向循环链表下的插入算法

双向循环链表下的结点插入算法与单链表下的结点插入算法的思想类似。将 previous 指针定位在 i−1 号结点,following 指针定位在 i 号结点,生成新结点 new_node 后完成插入。图 3.22 是插入操作的示意图,插入操作语句如下:

```
new_node.next = following
previous.next = new_node
new_node.prior = previous
following.prior = new_node
```

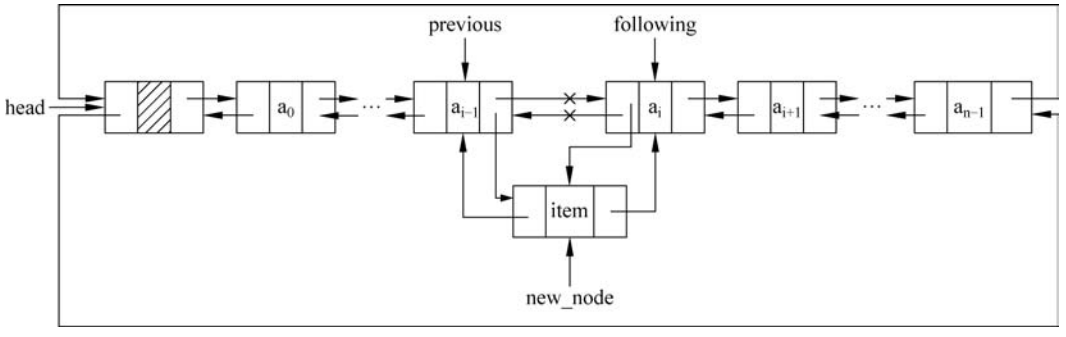


图 3.22 双向循环链表下的插入操作

双向循环链表下定位 previous 指针的方法也与单链表类似,而 following 就是 previous 的后继。由于是循环链表,所以将 previous.next!=self._head 作为是否已经遍

历整个表的条件。另外,算法还应处理参数不合法的情况,例如 i 小于 0 或 i 过大的情况,还要注意确保对 i 为 0 以及线性表为空表等边界情况的正确处理。插入算法的完整实现代码如下:

```
def insert(self, i, item):
    if i < 0:
        raise IndexError("插入位置不合法,i 值小于 0")
    previous = self._head
    count = -1
    while previous.next != self._head and count < i - 1:
        previous = previous.next
        count += 1
    following = previous.next
    if count == i-1:
        new_node = DuNode(item,previous,following)
        previous.next = new_node
        following.prior = new_node
    else:
        raise IndexError("插入位置不合法,i 值太大")
```

3. 双向链表的特点

与单链表相比,双向链表主要有以下特点:

(1) 可以根据实际需求不设头指针或尾指针,即去除类定义中的 head 指针,而设一个 current 指针指示最近访问结点,同时设 current_position 记录该结点的位序号。当需要定位 i 号结点时,活动指针总是从 current 开始,根据 current_position 与 i 的大小关系确定移动方向和次数。

(2) 在插入或删除结点时,需同时修改前驱和后继两个方向的指针。

(3) 设 current 指针指向双向链表中任意一个存在前驱和后继的结点,则该结点为其后继结点的前驱,也为其前驱结点的后继,即 current = current. next. prior = current. prior. next。因此,在做插入或删除操作时不必定位插入或删除位置的前驱结点,而可以直接定位当前位置结点。

在本节介绍的各种链表中,链表每个结点的数据域都只存放一个元素,在实际应用中,可能会在一个结点中存放多个数据元素,这时的结点称为块(block),这样的链表结构被称为块链表(简称块链)结构。

3.5 顺序表与链表实现小结

3.5.1 顺序表与链表的比较

1. 基本操作的时间复杂度

前面给出了线性表的两种截然不同的存储方案——顺序表与链表。表 3.1 列出了顺

序表和链表下各基本操作的时间复杂度,以方便读者进行对比。

表 3.1 顺序表和链表下各基本操作的时间复杂度

序　号	方　法	顺 序 表	链　表
1	__init__	O(1)	O(1)
2	empty	O(1)	O(1)
3	__len__	O(1)	O(n)
4	clear	O(1)	O(n)/ O(1) *
5	append	O(1)	O(n)
6	insert	O(n)	O(n)
7	remove	O(n)	O(n)
8	retrieve	O(1)	O(n)
9	replace	O(1)	O(n)
10	contains	O(n)	O(n)

* 如果结点由 clear 算法人工回收,时间复杂度为 O(n);如果结点由垃圾回收器自动回收,则时间复杂度为 O(1)。

2. 优缺点和适用场合

表 3.2 总结了顺序表和链表的优缺点和适用场合。

表 3.2 顺序表和链表的优缺点与适用场合

类别	优　点	缺　点	适 用 场 合
顺序表	(1) 程序设计简单; (2) 元素的物理位置反映逻辑关系,可实现随机存取,根据位序的读/写时间效率为 O(1); (3) 存储密度为 1	(1) 必须事先确定初始表长; (2) 插入、删除会带来元素的移动; (3) 多次插入后初始空间耗尽,造成溢出或需要空间扩容	(1) 表长能事先确定; (2) 元素个体较小; (3) 很少在非尾部位置插入和删除; (4) 经常需要根据位序进行读/写
链表	(1) 存储空间动态分配,不需事先申请空间; (2) 不需要担心溢出; (3) 插入、删除只引起指针的变化	(1) 不能做到随机存取,根据位序读/写效率为 O(n); (2) 链域也占空间,使存储密度降低,必定小于 1; (3) 由于涉及指针操作,程序设计的复杂性增大	(1) 元素个体较大; (2) 不能事先确定表长; (3) 很少需要根据位序进行读/写; (4) 经常需要做插入、删除和元素重排等

在此定义结点的**存储密度**。

$$\text{存储密度}=\frac{\text{数据元素本身所占的存储量}}{\text{该数据元素的存储映像实际所占的存储量}}$$

以元素内置的顺序表为例,顺序表中元素的存贮映像(不含指针域),因此存储密度为 1,而链表的存储密度必定小于 1。在 Python 实现的单链表中,每个结点存放一个数据元素对象的指针(引用)和一个后继结点的指针,因此存储密度的值为 1/2。在链式结构下,

去除表头结点等的影响，存储空间的利用率与存储密度的值相同；而在顺序结构下，由于分配的空间中可以有空位置，所以虽然存储密度为1，但存储空间的利用率通常不是100%。

顺序表有一个初始容量分配的问题。如果初始容量分配很大，可能会造成浪费；如果容量分配太小，且频繁地进行插入操作，一些简单的实现方案会造成溢出，即使如3.3.4节中采用了动态调整策略，频繁地进行空间调整也会使算法效率变低。

链表是一种动态的存储结构。由于结点空间动态分配和回收，链表的实现不需要事先申请空间，不需要担心上溢出。在程序运行过程中，链表的规模可随时发生动态变化。

总的来说，如经常需要根据位序进行读/写，应选用顺序表，因为顺序表最大的优点就是随机存取；如经常进行插入和删除，则应选用链表，因为链表具有动态存储的特性，插入和删除只需修改指针，不需移动元素。

3.5.2 各种链表实现的比较

3.4节中介绍了线性表的多种链式实现，读者在选择使用何种方案时应从有利于基本操作的实现，有利于提高基本操作的时间和空间效率等方面来考虑。表3.3列出了各种链表结构的特点和适用场合。

表3.3 各种链表结构的特点和适用场合

类 别	特点和适用场合
不加头结点的单链表	0号位置的插入、删除等操作需要额外操作，适合递归处理
加了头结点的单链表	可以使0号位置的操作与其他位置的操作一致，对空表与非空表的操作一致，使算法得到简化，被广泛使用
循环链表	可以方便地从尾结点走到头结点，方便循环往复地操作
双向链表	存储密度更低，在需要两个方向的操作时适用

总之，关于如何选择线性表的存储方案，着重考虑两方面的情况：一是充分利用计算机内存的特点对表中元素和元素之间的关系进行存储；二是充分考虑一些重要操作的效率。通常，对表进行的最频繁的操作有访问元素、插入元素、删除元素等，因此希望这些操作的时间效率尽可能高。

3.5.3 自顶向下的数据结构实现

如同算法自顶向下的细化过程，也可以对算法所操作的数据进行自顶向下、从抽象到具体的逐层细化。首先确定问题研究对象的数学概念和抽象数据类型，然后逐渐确定更多的细节，直到最终可以将数据结构实现为高级语言的类。尽管不同的问题需要不同数量的细化步骤，而且这些步骤之间的界限有时会模糊，但通常可以采用5个细化步骤。

(1) 数学概念层：确定研究对象的数学模型，例如一个序列(sequence)。

(2) 抽象数据类型层：确定数据之间的关系以及需要哪些概念性操作，但是不需要确定数据实际如何存储或操作如何执行。例如，明确当前的操作对象是一个线性表(list)。

(3) 数据结构层：指定足够的细节，分析各操作的行为，例如主要做查找操作还是做

增删操作,并根据所求解的问题做出适当的选择。例如,选定顺序存储,将数据存储在数组中。

(4) 实现层:确定如何在计算机内存中表示数据结构;确定算法实现的细节,例如用底层 C 数组实现顺序表。

(5) 应用层:实现特定应用程序所需的所有细节,例如求两个线性表的相同元素、约瑟夫环问题等(见 3.6 节)。

图 3.23 给出了自顶向下对数据进行细化的 5 个层次。其中,前 3 个层次常称为概念层,因为在这些层次上更关心问题的解决,而不是编程;中间两个层次称为算法层,因为它们涉及数据表示以及对数据进行操作的具体方法;最后两个层次则与具体程序设计有关,因此称为程序设计层。

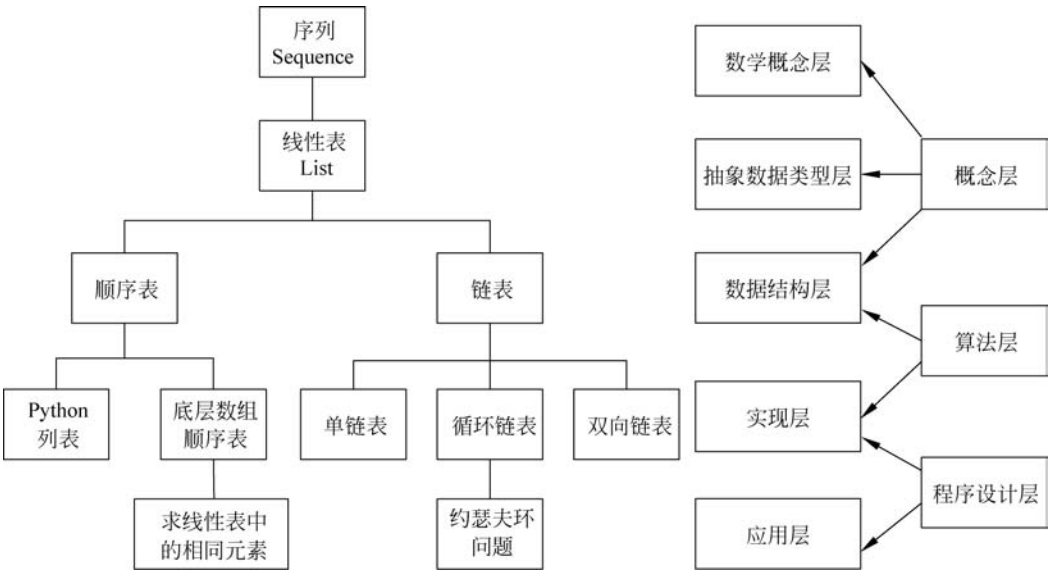


图 3.23 自顶向下的数据结构层次

在用 Python 实现数据结构时,任务是从抽象概念开始,逐步对其进行细化,最终类的方法对应于 ADT 的操作,类的数据成员对应于该数据结构的存储结构,这样就得到了该数据结构的 Python 实现。

3.5.4 算法设计的基本步骤

根据对线性表的插入和删除等算法的分析和实现,算法设计的基本步骤可总结如下:

(1) 确定算法的详细功能,包括确定函数的入口参数、出口参数和返回值。入口参数是为完成此功能需从外界获取得到的信息,出口参数是除了返回值之外向外界传递信息时用的参数。在 Python 中当可变对象为入口参数时,它在函数中的变化会影响实参对象,即自然成为出口参数。

(2) 分析一般情况下算法的实现步骤,通常可借助图示。

(3) 写出一般情况下算法的主体执行部分。

(4) 检查入口参数的合法性。

(5) 检查特殊情况。

(6) 分析算法的性能及可能的改进方法,分析算法的适用场合。

3.6 线性表的应用

3.6.1 求两个线性表的相同元素

在实际应用中,经常要对两个线性表中的数据进行合并,求其中相同或不同元素等操作。以下分别在无序表和有序表结构下,以求两个表的相同元素为例,讨论线性表的使用方法。

1. 无序线性表下的实现

假设线性表为无序表,例如 A=(7, 2, 1, 9),B=(3, 6, 7, 2, 5),A 和 B 中的相同元素存放在无序表 C 中,则 C=(7, 2)。

求 A 和 B 中相同元素的算法可以描述为:对于 A 中的每个元素 A_i,检查它在 B 中是否存在,即 A_i 与 B 中的元素依次比较,如果存在,则加入 C 中。对应算法如下:

```
def intersect(la, lb):
    m = len(la)
    n = len(lb)
    lc = DynamicArrayList()
    for i in range(m):
        x = la.retrieve(i)
        if lb.contains(x):
            lc.append(x)
    return lc
```

上述 intersect 算法在 DynamicArrayList 存储结构下进行测试,但很显然该算法的正确性不依赖于具体的存储方案,即对于线性表的不同存储结构都是有效的。

A 中的每个元素都要与 B 中的每个元素进行一次比较,设 A 的长度为 m,B 的长度为 n,则比较次数为 m * n 次,算法的关键操作即为比较操作,因此理论上该算法能达到的最好性能为 O(m * n)。

如果 la、lb 和 lc 采用顺序结构存储,retrieve 方法的时间性能为 O(1),contains 方法为 O(n),append 方法为 O(1),外层循环 m 次,因此算法的时间复杂度为 O(m * n)。

如果 la、lb 和 lc 采用链式结构存储,retrieve 方法的时间性能为 O(m),contains 方法为 O(n),append 方法为 O(min(m,n)),外层循环 m 次,因此算法的时间复杂度为 O(m * (m+n)),算法的效率更差。请读者自行考虑如何修改链表的定义和 intersect 算法,使得本算法在链式结构下的时间性能也能达到 O(m * n)。

2. 有序线性表下的实现

如果线性表为有序表,在上例中即 A=(1, 2, 7, 9),B=(2, 3, 5, 6, 7),A 和 B 中

的相同元素存放在有序表C中,则C=(2, 7)。在对两个有序表进行合并等运算时经常采用双下标法求解。

求解两个有序表中相同元素的算法步骤如下:

(1) 设两个下标变量i和j分别指示A和B的当前位置,初值都为0。

(2) 当i小于A表的长度且j小于B表的长度时执行循环:将A_i与B_j进行比较,如果$A_i<B_j$,说明A_i不在C中,i加1;如果$A_i>B_j$,说明B_j不在C中,j加1;如果$A_i=B_j$,将A_i加入C的末尾,i加1,j加1。

循环退出时,至少有一个表的全部元素已经检查完毕,如果另一个表还未到达表尾,则它剩下的元素也不会是相同元素,算法结束。

根据以上算法步骤,设计与线性表实现无关的算法,算法的完整代码如下:

```
def intersect(la, lb):
    m = len(la)
    n = len(lb)
    lc = DynamicArrayList()
    i = 0
    j = 0
    k = 0
    while i < m and j < n:
        item_a = la.retrieve(i)
        item_b = lb.retrieve(j)
        if item_a < item_b:
            i += 1
        elif item_a > item_b:
            j += 1
        else:
            lc.insert(k, item_a)
            k += 1
            i += 1
            j += 1
    return lc
```

上述算法通过一对元素item_a和item_b的比较,可以排除掉一个元素或在C中加入一个元素。由于两个表的元素总数为m+n,所以最多比较m+n-1对元素。在线性表存储结构选择合理时,算法的时间复杂度可达到O(m+n)。例如,在顺序结构下,retrieve算法的性能为O(1),insert算法在表尾位置插入时性能为O(1),整个算法的性能即为O(m+n)。

由此可见,相比于无序表,在有序表下求两个表的相同元素的算法的效率更高。当然,读者应该注意到,生成一个有序表的时间代价高于生成一个无序表的时间代价,但在对有序表的后续操作中得到了补偿。

3.6.2 约瑟夫环问题

设有n个人围坐一圈,从1开始顺序编号。现在从第1个人开始报数,报到第

m(m>0)的人退出。然后继续进行 1～m 的报数，直至所有人退出，最后一个退出的人是优胜者。依次输出出列人员的编号。该问题即著名的约瑟夫环问题。

1. 基于 Python 内置 list 的实现

假设用列表 people 存储所有人员，例如 n=10 时，people=[1，2，3，4，5，6，7，8，9，10]。在找出应该退出的人之后，将其对应编号从表里删除。在计算过程中表越来越短，用 num 表示表的长度，每退出一人，删除表中的对应元素，长度 num 减 1，至表长度为 0 时工作结束。假设 i 为本轮报数人员的开始位置，则该轮报数出列人员的位置以及下一轮报数的开始位置都为(i+m-1) % num，重复报数 n 轮，即可完成报数。基于上述思路的算法如下：

```
def josephus(n, m):
    people = list(range(1, n+1))
    i = 0
    for num in range(n, 0, -1):
        i = (i + m-1) % num
        print(people.pop(i), end="")
        if num > 1:
            print(",", end="")
```

如 n=10，m=3，以上算法输出的出列人员编号为 3，6，9，2，7，1，8，5，10，4。虽然这个简单的循环计数算法很容易理解，并且似乎是一个线性时间算法，但其实不然，因为 Python 列表非尾部位置的 pop 操作的时间效率为线性阶，整个算法的时间复杂度为 $O(n^2)$。

2. 基于单向循环链表的实现

单向循环链表可以很好地模拟围坐一圈的人，顺序地报数则相当于指针在循环链表中沿 next 链域向后移动，一个人退出则相当于删除相应结点。在删除某结点之后，接下来仍沿着原方向继续报数。因此可以用单向循环链表的操作来求解约瑟夫环问题。为方便快速地从尾部到达首结点，该链表不应设置表头结点。例如，编号为 1～10 的一圈人，可用如图 3.24 所示的单链表进行模拟。

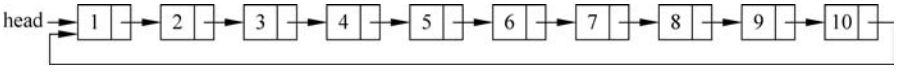


图 3.24　用不带头结点的单向循环链表表示约瑟夫环

单向循环链表类的定义和实现可以参考 3.4.1 节的单链表类，接下来为单向循环链表类添加两个方法。

1) 建立结点值依次为 1 至 n 的不带头结点的单向循环链表

```
def create_cll(self, n):
    self._head = p = n_node = Node(1)
```

```
        for i in range(2, n+1):
            n_node = Node(i)
            p.next = n_node
            p = p.next
        n_node.next = self._head
```

将每个人的编号1到n依次存储在该链表从头至尾的结点中。从空表开始在尾部逐个加入结点,最后注意将链表的尾结点指针连接到首结点。该算法的时间复杂度为O(n)。

2) 单向循环链表下的循环报数

对上述create_cll方法所建立的单向循环链表进行1～m的循环报数,并逐个删除需退出的结点,直至链表为空。假设编号为1到n的人以1～m进行循环报数,则算法如下:

```
def josephus(self, n, m):
    self.create_cll(n)                    #创建单向循环链表
    p = self._head                        #p定位在不带头结点的循环链表的首结点
    q = p
    count = n                             #count是表的长度
    while q.next != p:
        q = q.next                        #q定位在该链表的末尾,即q.next是p结点
    while count != 0:
        num = m % count
        #如果报的数很大,为减少循环,将报数的范围缩小到小于count
        if num == 0:
            num = count                   #如果num为0,说明报的数应为count
        while num > 1:                    #循环报数
            q = q.next
            p = p.next
            num -= 1
        print(p.entry, end="")            #输出将要删除的结点的值
        if count > 1:
            print(",", end="")
        q.next = p.next                   #删除p结点
        del p
        count -= 1
        if count == 0:
            break
        p = q.next                        #恢复p的位置,继续进行下一轮报数
```

当10个人以1～3进行报数,即调用josephus(10, 3)算法,输出的出列人员编号为3, 6, 9, 2, 7, 1, 8, 5, 10, 4。虽然用单向循环链表实现的方法比用list实现的方法复杂,但算法的效率较高。现分析其时间复杂度,算法外层循环n次,每次循环删除一个结点,在删除结点前指针p和q分别移动m次,因此算法的时间复杂度为O(n * m),如果m≪n,则算法的效率可达到O(n)。

3.7 线性表算法举例

线性表是使用最广泛的一种数据结构，线性表下的算法设计尤为重要。接下来通过一些例子介绍顺序表、单链表以及与存储结构无关的线性表算法的基本设计方法，将顺序表、单链表下的算法作为类的方法来设计，与存储结构无关的线性表算法则作为调用类的外部函数来设计。

3.7.1 顺序表下的算法

【例 3.1】 为底层动态数组实现的顺序表类添加一个方法，删除第 i 号位置开始的 k 个元素。

首先画出底层数组示意图，如图 3.25 所示。

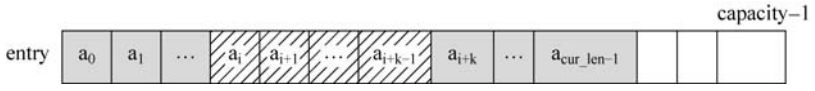


图 3.25 顺序表元素连续删除示意图

为了删除从 a_i 开始的连续 k 个元素，需将从 a_{i+k} 位置开始直到最后一个位置的所有元素依次往前移动 k 个位置，可用循环：

```
for j in range(i + k, self._cur_len):
    self._entry[j - k] = self._entry[j]
```

然后元素个数的计数变量减去 k，即：

```
self._cur_len -= k
```

在算法开始处加上参数合法性的检查，要能够删除 k 个元素，i+k−1 号元素要存在，即必须 i+k−1≤cur_len−1，所以 i<0 或 k≤0 或者 i+k>cur_len 都属于不合法情况。

算法的完整代码如下：

```
def remove_k(self, i, k):
    if i < 0 or k <= 0 or i + k > self._cur_len:
        raise IndexError("参数不合法")
    for j in range(i + k, self._cur_len):
        self._entry[j - k] = self._entry[j]
    self._cur_len -= k
```

【例 3.2】 假设顺序表中存储了若干个整数，设计时间性能和空间性能尽可能高效的算法，将表中小于或等于 x 的元素都放在列表的前端，将大于 x 的元素都放在列表的后

端。例如,线性表为(3, 2, 1, −2, −4, 9),x=0,则经过算法处理后,负数在前,正数在后,而这些数的顺序可以是随意的。

一个比较简单的方法是从左到右扫描表中的每个元素。将已经处理的元素分为两部分,第一部分元素小于或等于 x,第二部分元素都大于 x。第二部分第一个元素的下标为 first_large。对于正在处理的元素 entry[i],若 entry[i]>x,则 i 加 1,否则将 entry[i]与 entry[first_large]交换,并且 first_large 加 1。算法的完整代码如下:

```
def adjust(self, x):
    first_large = 0
    for i in range(0, self._cur_len):
        if self._entry[i] <= x:
            self._entry[first_large], self._entry[i] = self._entry[i], self._entry[first_large]
            first_large = first_large + 1
```


视频讲解

3.7.2 带头结点单链表下的算法

【例 3.3】 为带头结点的单链表类添加一个方法,利用原表空间将单链表中的所有元素进行逆置,即将线性表(a_0, a_1, …, a_i, …, a_{n-1})逆置为(a_{n-1}, a_{n-2}, …, a_i, …, a_0)。

为了在链表下完成逆置,可以采用头插法,把每个元素结点依次插入表的最前面,即插入表头结点之后。

首先将原表分成两部分,活动指针 p 初始时指向首结点,并且将表头结点的链域赋值为空指针,如图 3.26 所示。

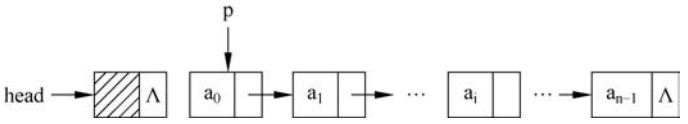


图 3.26 单链表逆置 1

接着将 p 及之后的每个结点依次插入 head 所指的结点之后,所有结点插入的方法是一致的。假设在某个时刻已经将 a_0 至 a_{i-1} 的每个结点依次插入 head 之后,这时表的状态如图 3.27 所示。

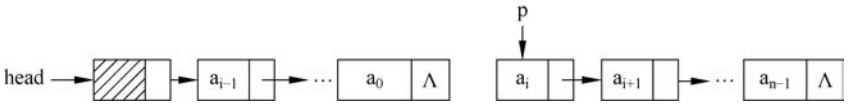


图 3.27 单链表逆置 2

为了看得更清楚,把 p 结点画在 head 附近,如图 3.28 所示。

将 p 所指结点插入 head 之后的代码如下,代码的执行效果如图 3.29 所示。

```
p.next = self._head.next
self._head.next = p
```

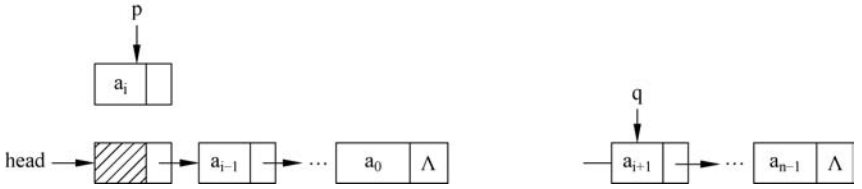


图 3.28　单链表逆置 3

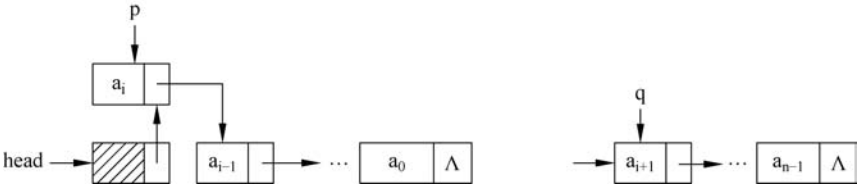


图 3.29　单链表逆置 4

当 p 所指结点插入完成之后，接着需要处理元素 a_{i+1} 这个结点，由于此时它已经不是 p 的后继，所以必须在插入 p 之前用另一个指针 q 指向 a_{i+1} 这个结点。逆置算法的完整代码如下：

```
def reverse(self):
    p = self._head.next
    self._head.next = None
    while p:
        q = p.next                  #q 指针指向 p 的下一个结点
        p.next = self._head.next    #将 p 插入为首结点
        self._head.next = p
        p = q                       #p 指向下一个待插入结点
```

由于将每个元素结点依次插入在表头结点之后，本算法的时间复杂度为 O(n)。

【例 3.4】 为单链表类设计特殊方法__lt__，判断当前线性表是否小于另一个给定线性表 other。设线性表 A 为(a_0，a_1，a_2，…，a_i，…，a_{n-1})，线性表 B 为(b_0，b_1，b_2，…，b_j，…，b_{m-1})。如果存在一个 k(k≥0)使得 $a_i=b_i$(i=0，1，…，k−1)且 $a_k<b_k$，或者 n<m 且对任意 i=0，1，…，n−1 都有 $a_i=b_i$，则称 A 小于 B。例如(1，2，3，4)小于(1，3)，(1，2)<(1，2，5)，空表<任何非空表。

根据题目中给出的比较两个线性表大小的方法得出以下算法步骤：

(1) 在 A、B 两个表中设活动指针，假设分别为 p 和 q，初始指向首元素结点。

(2) 当 p 和 q 都非空时循环执行：如果 p 结点的值小于 q 结点的值，返回 True；如果 p 结点的值大于 q 结点的值，返回 False；如果 p 结点的值等于 q 结点的值，p 和 q 都往后移动一个结点。

(3) 循环(2)已退出，说明 p 和 q 至少有一个为空。如果 q 为空，若 p 非空，说明 B 表小于 A 表；若 p 空，说明两个表相等，因此返回结果都为 False；否则，即 p 为空而 q 非空，说明 A 表小于 B 表，返回结果为 True。

```
def __lt__(self, other):
    p = self._head.next
    q = other.get_head().next
    while p and q:
        if p.entry < q.entry:
            return True
        if p.entry > q.entry:
            return False
        p = p.next
        q = q.next
    if q is None:                    #q为空,说明A比B长(p不空)或等长(p空)
        return False
    return True                      #p为空,q非空,说明A比B短
```

本算法的时间复杂度为O(min(m,n))。

【例3.5】 假设OrderedList类为LinkedList类的继承类,即以带头结点的单链表实现有序线性表,定义框架如下:

```
from Node import Node
from LinkedList import LinkedList
class OrderedList(LinkedList):
    def add(self, item):
    def insert(self, position, item):
    def replace(self, position, item):
```

实现OrderedList类的add方法,将值为item的结点插入合适的位置,使得插入后的表仍保持有序。例如,原线性表为(1, 2, 5, 9, 16),插入item=8,则插入后的表为(1, 2, 5, 8, 9, 16)。

根据题目中给出的例子,可画出如图3.30所示的示意图。为了插入值为8的结点,需要一个指向值为5的结点的指针,设为p,那么结点的插入就能迎刃而解,相应语句为:

```
newnode = Node(item, p.next)
p.next = newnode
```

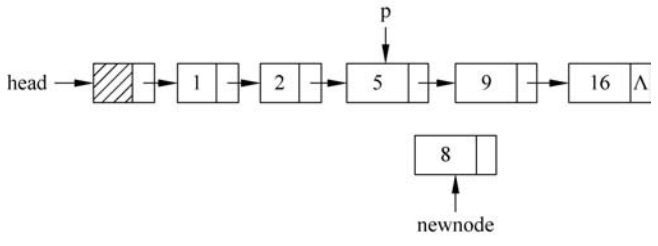


图3.30 有序表下的插入

如何定位p呢?唯一的办法是从head开始寻找,即初始时p=self._head。如果p的下一个结点存在,且下一结点的值小于或等于item,p指针后移,否则p指针停止移动。

根据以上分析可得到如下算法：

```
def add(self, item):
    p = self._head
    while p.next and p.next.entry <= item:
        p = p.next
    newnode = Node(item, p.next)
    p.next = newnode
```

如果由于 p.next 为空而结束 while 循环，说明 p 已到达链表的尾部，item 的值大于原表中的所有结点或原表为空，新结点插入在表尾。该算法的时间复杂度为 O(n)。

3.7.3 与线性表具体实现无关的算法

如果并不关注或不知道当前线性表采用何种存储方案，只知道这个线性表拥有 AbstractList 类的所有性质和方法，这时可调用线性表类提供的方法进行算法设计。例如，3.6.1 节介绍的求两个线性表相同元素的算法都与具体实现无关。

【例 3.6】 调用线性表类提供的方法对线性表进行逆置。

对线性表逆置可以采用首尾交换法，即将线性表首尾对应位置的元素进行交换。算法的完整代码如下：

```
def reverse(alist):
    i = 0
    j = len(alist) - 1
    while i < j:
        item_a = alist.retrieve(i)
        item_b = alist.retrieve(j)
        alist.replace(i, item_b)
        alist.replace(j, item_a)
        i += 1
        j -= 1
```

调用 retrieve 方法获得表的 i 号元素 item_a 和 j 号元素 item_b；然后用 replace 方法将 item_b 替换到 i 号位置，将 item_a 替换到 j 号位置。

上述算法适用于线性表的任意存储结构。如果存储结构为顺序表，由于 retrieve 和 replace 的性能都为 O(1)，所以 reverse 算法的时间复杂度为 O(n)。如果存储结构为链表，由于 retrieve 和 replace 的性能都为 O(n)，reverse 算法的时间复杂度为 $O(n^2)$。

对线性表的逆置还可以采用头插法，即将线性表中从 0 号位置开始的每个元素依次插入在表的最前端，即 0 号位置上。算法的完整代码如下：

```
def reverse(alist):
    for i in range(0, len(alist)):
        item = alist.remove(i)
        alist.insert(0, item)
```

上述算法适用于线性表的任意存储结构。如果存储结构是顺序表,由于循环中调用的 remove 和 insert 方法的时间性能都为 O(n),所以 reverse 算法的时间复杂度为 $O(n^2)$。如果存储结构为链表,remove 和 0 号位置的 insert 方法的时间性能分别为 O(n) 和 O(1),而 reverse 算法的时间复杂度也为 $O(n^2)$。在顺序表下的删除和插入操作需要移动元素,并且在 0 号位置的 insert 操作为最坏情况。在链表下的删除和插入操作需要移动指针,但在链表下 0 号位置的插入属于最好情况,因此在链表结构下使用头插法的效率好于顺序表结构。

在 3.7.2 节的例 3.3 已介绍了单链表结构下用头插法逆置线性表的具体算法,该算法的时间复杂度为 O(n)。因此,调用类的方法进行操作可能降低算法的效率。

3.8 上机实验

3.8.1 线性表的顺序表实现

(1) 参考 3.3.4 节,用底层 C 数组实现无序顺序表类。

(2) 定义一个顺序表类的派生类——有序顺序表类,用于存放递增有序表,并为该有序顺序表类增加按值插入元素、按位置插入元素和替换元素等方法。可参考如下类定义框架:

```
from DynamicArrayList import DynamicArrayList
class OrderedArrayList(DynamicArrayList):
    def add(self, item):
    def insert(self, position, item):
    def replace(self, position, item):
```

(3) 设计一个测试程序,测试所设计的两个类是否正确。

要求:测试程序可以测试类中所有方法的正确性。用户界面友好,程序每次执行可以循环接受多次命令,直至用户按“Q”或“q”键退出程序。

3.8.2 线性表的单链表实现

参考 3.4.1 节的内容,实现单链表类并进行正确性测试。

3.8.3 线性表的双向非循环链表实现

参考 3.4.3 节描述的双向循环链表,实现双向非循环链表类并进行正确性测试。

3.8.4 消费支出项目管理

琳琳上大学后将她每天的消费支出项一行一项写在她的一个日记文件中,她的每个支出项目记录了支出日期、支出项目和金额。3 个月之后,她想统计一下她的消费行为信息,请设计程序帮助她完成相应的查询和统计。

程序功能包括：

(1) 从文本文件读入所有 n 项支出项目，并依次输出所有支出项目。

(2) 求出这 n 个支出项目中的最小、最大和总消费。

(3) 按照日期找出某一天的所有消费。

(4) 按照日期和支出项目找出该项目的消费。

(5) 按照项目找出该支出项目的所有消费，例如要求给出在“鞋子”这一项上的总消费。

(6) 按照支出项消费递减的顺序输出每项的对应总消费。

3.8.5 每日快递

快递员小丰每日负责高新区中 n 个居民小区的快递派送任务。按公司规定，他应该根据固定路线每天上午和下午各进行一次派送。在派送过程中，他还应接受小区内已经预定寄出的快递。若某小区的派件数和收件数均为 0，则他不需要前往该小区。每日派送结束后，公司会对每个快递员去过的小区数目、派件数和收件数进行统计。

输入：

(1) 第 1 行为居民小区数 n。

(2) 第 2 行包含 n 个数字，对应于当天上午 n 个小区的派件数。

(3) 第 3 行包含 n 个数字，对应于当天上午 n 个小区的收件数。

(4) 第 4 行包含 n 个数字，对应于当天下午 n 个小区的派件数。

(5) 第 5 行包含 n 个数字，对应于当天下午 n 个小区的收件数。

输出：快递员一天去过的小区数目、总的派件数和收件数。

3.8.6 扑克牌整理

原有 n 副扑克牌，但因时间久远均已张数不全。现把它们合在一起，看是否能凑成 m(m＜n)副完整的扑克牌(不考虑大王和小王)。

(1) 输入：输入数据来自文本文件，文件的第 1 行是一个数字 n(1≤n≤20)，表示原有 n 副牌。从第 2 行起，每 4 行用于描述一副牌的不同花色(固定为黑、红、梅、方的顺序)的现有张数(各行的第一个数)和牌号 i(1≤i≤13)(各行的其余数字，无序)。

(2) 输出：输出的第 1 行是所能拼凑的扑克牌套数。接着按花色输出第 j(1≤j≤n)副扑克牌中剩下的扑克，且对每个花色按牌点大小顺序输出。

(3) 采用带头结点的单链表。

本章习题

一、选择题

1. 在单链表中增加一个头结点的目的是_____。

A. 使单链表至少有一个结点

B. 标识表结点中首结点的位置

C. 方便运算的实现

D. 说明单链表是线性表的链式存储

2. 设 p 为指向单向循环链表上某内部结点的指针,则查找 p 指向结点的直接前驱结点时_____。

A. 查找时间为 O(n)　　B. 查找时间为 O(1)

C. 指针 p 移动的次数约为 n/2　　D. 找不到

3. 能在 O(1)时间内访问线性表的第 i 号元素的结构是_____。

A. 顺序表　　B. 单向非循环链表

C. 单向循环链表　　D. 双向循环链表

4. 对于线性表,在顺序存储结构和链式存储结构中查找第 k 号元素,其时间复杂度_____。

A. 都是 O(1)　　B. 都是 O(k)

C. 分别是 O(1)和 O(k)　　D. 分别是 O(k)和 O(1)

5. 单链表只设头指针,将长度为 n 的单链表链接在长度为 m 的单链表之后的算法的时间复杂度为_____。

A. O(1)　　B. O(n)　　C. O(m)　　D. O(m+n)

6. 设某顺序表中第一个元素的起始存储地址为 a,每个元素的长度为 b,则第 c 个元素的起始存储地址是_____。(a、b、c 均为非负整数)

A. a+b*c−b　　B. a+b*c

C. a+b+c　　D. a+b*c−c

7. (多选)单链表的特点包括_____。

A. 顺序存取

B. 在插入、删除元素时不需要移动表中的元素

C. 在插入、删除元素时需要修改指针

D. 随机存取

8. (多选)顺序表的特点包括_____。

A. 随机存取

B. 在插入、删除元素时需要移动表中的元素

C. 顺序存取

D. 在插入、删除元素时不需要移动表中的元素

二、判断题

(　　)1. 顺序表结构适宜于进行顺序存取,而链表适宜于进行随机存取。

(　　)2. 线性表的插入、删除总是伴随着大量数据的移动。

(　　)3. 顺序存储结构要求连续的存储区域,在存储管理上不够灵活,因此不常用。

(　　)4. 在一个设有表头指针和表尾指针的单链表中,删除其最后一个元素的操作的时间性能与链表的长度无关。

(　　)5. 在链表的每个结点中都恰好包含一个指针。

(　　)6. 顺序存储结构的优点是存储密度大,且插入、删除运算的效率高。

(　　)7. 线性表在顺序存储时,逻辑上相邻的元素在存储的物理位置上未必相邻。

(　　)8. 链表的删除算法很简单,因为在删除链表中的某个结点后,计算机会自动将后续各个单元向前移动。

(　　)9. 逆置单链表适合用头插法,逆置顺序表适合用首尾交换法。

三、填空题

1. 在长度为 n 的顺序表中删除 i 号数据元素,需要移动表中的_____个元素。(0≤i<n)

2. 对于一个长度为 n 的顺序表,将值为 x 的元素插入在表中的 i 号位置,需向后移动的元素个数为 _____,i 的合法范围为_____。

3. 在顺序表中,按位序访问任一元素的时间复杂度均为_____,因此顺序表也称为_____的数据结构。

4. 顺序表中逻辑次序相邻的元素的物理位置_____相邻。单链表中逻辑次序相邻的元素的物理位置_____相邻。(填写:“一定”或“不一定”)

5. 在单链表中,除了首元素结点外,任一结点的存储位置由_____指示。

6. 在 n 个结点的单链表中要删除已知结点 p,需找到它的_____,其时间复杂度为_____。

7. 链式存储的特点是利用_____来表示数据元素之间的逻辑关系。

8. 在单链表中,指针 p 所指结点有后继结点的条件是_____。

9. 对于一个具有 n 个结点的单链表,在已知的结点 p 后插入一个新结点的时间复杂度为_____。

10. 在 n 个结点的单循环链表中,若仅设头指针,则访问首结点和尾结点的时间复杂度分别为_____、_____;若仅设尾指针,则访问首结点和尾结点的时间复杂度分别为_____、_____。

11. 在 n 个结点的双向非循环链表中,若仅设尾指针,则访问首结点和尾结点的时间复杂度分别为_____、_____。

12. 当线性表中的元素总数基本稳定,且很少进行插入和删除操作,但要求以最快的速度按序号存取线性表的元素时,应采用_____存储结构。

13. 给定两个有序顺序表 A 和 B,设 A 表的长度为 m、B 表的长度为 n,将两个有序表合并成有序表 C,在最坏情况下需进行_____次比较。

14. 以下算法的时间复杂度分别是多少? 请用大 O 记号表示。

(1) 将长度为 n 的顺序表置成空表:_____。

(2) 将长度分别为 m 和 n 的递增有序单链表 A 和 B 合并成递减有序单链表 C(假设可以通过头指针直接访问链表的结点):_____。

四、应用题

1. 说明顺序表和链表的优缺点和适用场合。

2. 线性表的顺序存储结构具有3个弱点：其一，在进行插入或删除操作时需移动元素；其二，由于难以估计，必须预先分配较大的空间，往往使存储空间不能得到充分利用；其三，表的容量较难扩充。线性表的链式存储结构是否一定能够克服上述3个弱点，试讨论之。

3. 线性表有两类存储结构，一是顺序表，二是链表。如果有n个线性表并存，并且在处理过程中各表的长度会动态变化，同时线性表的总数也会不断改变。在此情况下，应选用哪种存储结构？为什么？

4. 在线性表的以下链式存储中，若链表的头结点的地址未知，仅已知p指针指向的结点，能否从中删除该结点？为什么？

(1) 单链表。

(2) 双链表。

(3) 单向循环链表。

五、算法题

要求将算法1～6设计为3.3.4节定义的顺序表类DynamicArrayList的方法。

1. 设计算法，在顺序表中删除所有奇数位序的元素。

2. 设计算法，将顺序表中存放的元素循环左移p位，即L从(a_0, a_1, a_2, …, a_p, …, a_{n-1})变为(a_p, a_{p+1}, …, a_{n-1}, a_0, …, a_{p-1})。

3. 设计算法，将整数顺序表中的所有元素划分为两部分，其中前面部分的元素都小于或等于x，后面部分的元素都大于x。

4. 设计尽可能高效的算法，删除顺序表中所有值为x的元素。

5. 在某顺序表中，设有一个整数在该表中的出现次数为奇数，其余整数的出现次数均为偶数。设计尽可能高效的算法，寻找出现次数为奇数的整数。例如，在(1, 2, 5, 2, 5, 1, 5)中，出现次数为奇数的整数为5。

6. 设计算法，将有序顺序表中position位置的元素的值替换为item，要求替换后的表仍保持有序。

要求将算法7～13设计为3.4.1节定义的带头结点单链表类LinkedList的方法。

7. 设计算法，定位单链表中间位置的结点。例如，线性表为(1, 2, 3, 4, 5)，则中间位置的元素为3；线性表为(1, 2, 3, 4)，则中间位置的元素为2。

8. 设计算法，判断非空单链表是否递增有序。

9. 设计算法，删除单链表中的所有冗余元素，并返回删除的元素个数。例如，原线性表为(7, 2, 1, 7, 2, 3, 6, 3, 5)，去除冗余之后的表为(7, 2, 1, 3, 6, 5)。

10. 设计算法，找出单链表中最后一个满足n % k=0的结点，n表示从首结点开始的结点个数(未知)，k是给定的整型常数。

11. 设计算法，将给定的单链表中所有值为偶数的结点放在值为奇数的结点前面。

12. 设计算法，将一个整数的质因数进行分解并按递减顺序生成一个有序单链表。例如输入2100，则生成的单链表中的元素为(7，5，5，3，2，2)。

13. 设计算法，在有序单链表中插入值为x的元素，并保持表的有序性。

在算法14～19中，la、lb、lc为各个单链表的头指针，直接用头指针表示和操作单链表。

14. 假设链表la有两种可能的状态：它或者有尾部(蛇)，或者最后一个结点的指针域指向链表前面的某个结点(蜗牛)。给出一个算法，判断给定的链表la是蛇还是蜗牛。

15. la和lb分别为两个不带头结点的单链表，设计算法，从la中删除自i号元素起的len个结点，并将这len个结点插入lb中的i号结点之前。

16. 设计算法，将一个单链表la分裂成奇、偶两个单链表，偶数位序的结点留在原表，奇数位序的结点形成一个新表lb。

17. 设计算法，将两个递增有序单链表la和lb合并成一个递减有序表。

18. 设计算法，求两个有序单链表la和lb表示的集合的差集lc，lc也为有序单链表。

19. 设计算法，求两个有序单链表la和lb表示的集合的交集lc，lc也为有序单链表。

在算法20中，la为双向链表的头指针，直接用头指针表示和操作双向链表。

20. 设有一个双向链表la，每个结点中除了有prior、data和next域外，还有一个访问频度域freq，在链表被使用之前，freq域初始化为0。每当链表进行一次locate(la，x)运算后，令值为x的结点中的freq域增1，并调整表中结点的次序，使其按访问频度的递减序列排列，以便使频繁访问的结点总是靠近表头。设计满足上述要求的locate(la，x)算法。

算法21可直接使用Python的列表设计。

21. 假设用一个线性表记录某地区每年的平均气温，以便研究全球气候变暖的趋势。研究者发现，气温的变化是有一定规律的。例如，线性表L=(3，4，6，4，5，7，5)记录了连续7年的平均气温，从第一年开始，按照+1、+2、−2的规律变化，然后又按照+1、+2、−2的规律变化，称这个表中存在一个长度为3的温度变化环。又如，(1，10，13，45，48，57，60，92，95，104，107)存在一个长度为4的环+9，+3，+32，+3。设计算法求线性表中这个环的长度。