第 3 章 线性表

线性表是一个由元素构成的序列，该序列有唯一的首元素和尾元素，除了首元素外，每个元素都有唯一的前驱元素，除了尾元素外，每个元素都有唯一的后继元素，因此线性表中的元素是有先后关系的。

线性表中的元素数据类型相同，即每个元素所占的空间必须相同。

实践中，适合用线性表存放数据的情形有很多，如 26 个英文字母的字母表、一个班级全部学生的个人信息表、参加会议的人员名单等。

根据在内存中存储方式的不同，线性表可以分为顺序表和链表两种。

3.1 顺 序 表

顺序表是元素在内存中连续存放的线性表。**顺序表的最根本和最有用的性质，是每个元素都有唯一序号，且根据序号访问（包括读取和修改）元素的时间复杂度是 $O(1)$ 的。**序号通常也称为元素的“下标”。首元素的下标是 0（规定为 1 也可以），下标为 i 的元素的前驱元素如果存在的话，下标是 $i-1$，后继元素如果存在的话，下标是 $i+1$。下标为 i 的元素，就称为第 i 个元素。

各种程序程序设计语言，如 C/C++、Java 等中的数组，都是顺序表。Python 中的列表（list）也是顺序表。**顺序表中的元素类型是一样的。**虽然 Python 的列表可以包含不同类型的元素，比如一个列表中可以有字符串、整数、元组等不同类型的元素，但那只是表面现象。本质上 Python 列表中的元素都是指针，它们的类型都一样，占用存储空间也一样大，只是这些指针可以指向不同类型的数据而已。

一般来说，顺序表应当支持表 3.1 中的操作。

表 3.1 顺序表支持的操作

序号	操　作	含　义	时间复杂度
1	init(n)	生成一个含有 n 个元素的顺序表，元素值随机	$O(1)$
2	init(a_0,a_1,…,a_n)	生成元素为 a_0,a_1,…,a_n 的顺序表	$O(n)$
3	length()	求表中元素个数	$O(1)$
4	append(x)	在表的尾部添加一个元素 x	$O(1)$

续表

序号	操　作	含　义	时间复杂度
5	pop()	删除表尾元素	$O(1)$
6	get(i)	返回下标为 i 的元素	$O(1)$
7	set(i,x)	将下标为 i 的元素设置为 x	$O(1)$
8	find(x)	查找元素 x 在表中的位置	$O(n)$
9	insert(i,x)	在下标为 i 处插入元素 x	$O(n)$
10	remove(i)	删除下标为 i 的元素	$O(n)$

init(n)：分配一片能够放下 n 个元素的内存空间，不需要对这片空间进行初始化。这片内存空间里原来的内容是什么，分配完还是什么，因此 n 个元素的初始值是随机无意义的。当然，分配空间本身是需要时间的，其快慢取决于操作系统，姑且认为是 $O(1)$的。

init(a_0,a_1,…,a_n)：和 init 操作相比，需要将存放 $n+1$ 个元素的内存空间初始化为 a_0,a_1,…,a_n，因此复杂度为 $O(n)$。

length()：求顺序表元素个数。该操作应在 $O(1)$时间内完成，专门维护一个记录顺序表元素个数的变量即可做到这一点。

append(x)：在顺序表尾部添加元素。这是一件比较复杂的事情。如果为顺序表分配的内存空间刚好容纳原有的全部 n 个元素，则直接将新元素添加到末尾元素的后面是不行的，因为末尾元素后面的内存空间有可能并非空闲，而是已经另有他用，如果鲁莽地往这部分空间写入数据，可能导致不可预料的错误。一种解决办法是重新分配一块足以容纳 $n+1$ 个元素的连续的内存空间，将原来的 n 个元素复制过来，然后再写入新的元素，当然还要释放原来的空间。这样做需要用 $O(n)$的时间来复制元素，append(x)的复杂度就是 $O(n)$。为了避免每次执行 append(x)都要复制元素，可以预先为顺序表多分配一些空间，这样需要执行 append(x)时，绝大多数情况下就可以直接将 x 写入原末尾元素的后面。待多余空间用完了，再次执行 append(x)，才需要重新分配一片更大的空间并执行复制元素的操作。

按照预先多分配空间的思想，可以引入顺序表的“容量”这个概念。顺序表的容量，是指不需要重新分配空间就能容纳的最大元素个数。空顺序表生成时，容量可以为 0，但是第一次执行 append(x)操作时，就可以多分配存储空间，例如，使其容量变为 4(当然也可以是其他数目)，这样接下来的 3 次 append(x)操作就不需要重新分配空间。当元素个数到达容量时，再执行 append(x)操作就需要重新分配空间并进行元素的复制。重新分配空间时，新的容量自然不能只是原容量加 1，而应该增加得更多。

一种方案是每次重新分配空间时，新容量总是等于旧容量加 k，k 是固定值。对空表执行 append(x)操作时即分配容量 k。由于元素个数每达到 k 的倍数加 1 时就需要重新分配存储空间并进行元素的复制，因此可以算出，执行 $m\times k$ 次 append(x)操作，元素个数从 0 增长到 $m\times k$ 的过程中，总共复制过的元素个数是：

$$k+2k+3k+\cdots+(m-1)k=k(1+2+3+\cdots+(m-1))=\frac{km(m-1)}{2}$$

因此在元素总数 $n=m\times k$ 的情况下，平均每次 append(x)操作，需要复制 $\frac{m-1}{2}$ 个元

素，$\frac{m-1}{2}=\frac{k(m-1)}{2k}=\frac{n-k}{2k}=\frac{n}{2k}-\frac{1}{2}$，由于 k 是常数，所以 append(x)的复杂度是 $O(n)$。这说明扩容时新容量总是等于旧容量加 k 的办法意义不大。

还有一种方案，扩容时，新容量总是旧容量的 k 倍。k 是大于 1 的固定值，可以取 1.2、1.5、2 等。假定第一次分配空间时，容量为 1。由于元素个数每达到 k 的幂(向上取整)加 1 时就需要重新分配存储空间并进行元素的复制，因此可以算出，执行 k^m 次 append(x)操作，元素个数达到 k^m 个时，总共复制过的元素个数是：

$$1+k+k^2+\cdots+k^{m-1}=\frac{k^m-1}{k-1}$$

因此在元素总数 $n=k^m$ 的情况下，平均每次 append(x)操作，需要复制的元素个数是：

$$\frac{(k^m-1)}{k^m(k-1)}=\frac{n-1}{n(k-1)}<\frac{1}{k-1}$$

因 k 是常数，所以 append(x)操作的平均复杂度是 $O(1)$的。一般来说，k 取 1.2 左右既不会太浪费空间，又能兼顾效率。

在上面的推导过程中，k^m 不是整数时，向上取整即可，不影响结论的正确性。

pop()：append(x)都能做到 $O(1)$，删除表尾元素做到 $O(1)$当然也没有问题，一般情况下，只需要将元素总个数的计数值减 1 即可。在表中空闲单元超过一定程度(如超过容量的一半)的时候，可以为顺序表重新分配更小的容量，将原有元素复制过去。Python 的 list 就会有类似的做法。

get(i)和 set(i,x)：假设顺序表在内存的起始地址是 s，且每个元素占用的内存空间为 m 字节，则第 i 个元素的内存地址就是 $s+i\times m$——这里的下标 i 从 0 开始算。由于用 $O(1)$的时间就能找到第 i 个元素的地址，所以读写第 i 个元素的时间，就是 $O(1)$的，和顺序表中元素的个数无关。

find(x)：要在顺序表中查找元素 x，没有什么好办法，只能从头看到尾。如果表中有 x，可能出现在头一个，也可能出现在最后一个，对一个有 n 个元素的顺序表，平均要看 $(n+1)/2$ 个元素才能找到 x。如果表中不包含 x，则要看完 n 个元素才能得到这个结论。不管哪种情况，复杂度都是 $O(n)$。如果找到 x，则返回 x 第一次出现的下标；如果找不到 x，可以返回 -1。find()还可以有功能更强的版本，如 find(x,i)表示从下标 i 处开始寻找 x。

如果顺序表中元素是排好序的，则用二分查找的办法，可以使得查找的复杂度变为 $O(\log(n))$。但那是另一回事，因为顺序表的特性并不包括元素有序。

insert(i,x)：在下标 i 处插入元素 x，会导致原下标 i 处及其后面的元素都要往后移动一个元素的位置(实际上就是把元素复制到后面的位置)。对原本有 n 个元素的顺序表来说，要移动的元素个数是 $n-i$，平均是 $(n+1)/2$ 个，所以 insert(i,x)的复杂度是 $O(n)$。当然，如果 insert 操作导致元素个数超过了顺序表的容量，还需要重新分配空间。

remove(i)：删除下标 i 处的元素，会导致原下标 $i+1$ 及后面的元素都要往前移动。类似于 insert(i,x)，复杂度也是 $O(n)$。

总之，在顺序表中进行查找，以及在中间插入、删除元素，都没有办法做到低于 $O(n)$的复杂度。

不同语言中的顺序表，支持的操作不一样，但是**所有语言的顺序表，都支持 get(i)和 set**

(i,x)这两个在$O(1)$时间内根据下标读写元素的根本操作，这个操作，也叫“随机访问”。

Python 中的顺序表 list 是个类，支持上述的各种操作（init(n)除外），只不过操作的名称和用法形式不同。C 语言中的顺序表就是数组，大小在初始化的时候就固定了且不能更改，因此只支持 length()、两种 init 操作以及 get(i)和 set(i,x)操作——init 操作就是定义数组，get(i)和 set(i,x)操作通过“[ ]”和下标进行。如果想在 C 语言中使用支持上述全部操作的顺序表，就得基于数组和动态内存分配自己写一个。在 C++ 语言中，STL 中的 vector 就是支持上述各种操作的顺序表。Java 语言中支持各种操作的顺序表则是 ArrayList。

3.2 链　　表

顺序表的劣势，是在中间插入和删除元素较慢（$O(n)$复杂度）。另外，在一些极端情况下，元素太多以至于找不到足够大的连续内存来存放它们，此时就无法使用顺序表。采用链接结构的链表能够较好地解决上述问题。

链接结构是一类元素在内存中不必连续存放，可以随意分布，元素之间通过指针链接起来的数据结构。在链接结构中，一个元素内部除了存放数据，还有一个或多个指针，指向其他元素。“指向”的准确含义是“指出其他元素的内存地址”。链接结构中的元素，往往称为“结点”。链接结构可以用来实现二叉树和树等数据结构。**用链接结构实现的线性表，叫做链表。**

链表有单链表、双链表、循环链表等多种形式。它们的共同特点如下。

（1）结点在内存中不需要连续存放。

（2）每个结点中都包含一个指向其后继结点的指针，不妨称其为 next 指针。表尾结点的 next 指针设为 None 或做另外特殊处理。

（3）表中结点可以动态增减。增删结点时，不需要复制结点或移动结点。

（4）如果已经得到了指向结点 p 的指针，则删除 p 的后继结点，或者在 p 后面插入新结点，复杂度都是 $O(1)$的。

链表不支持随机访问。要访问表中第 i 个元素，只能从表首元素开始，顺着 next 指针链一步步往后走，直到走到第 i 个元素。所以访问第 i 个元素这样的操作复杂度是 $O(n)$的。

3.2.1 单链表

每个结点中只包含一个指针，该指针指向后继结点，这样的链表称为单链表。图 3.1 是一个单链表的例子：

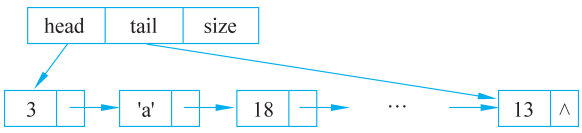

图 3.1　单链表

head、tail 和 size 是三个变量，head 指向表首结点 3，tail 指向表尾结点 13，size 记录表

中结点个数。单链表的结点由数据和 next 指针构成。在图 3.1 中 next 指针就是两个结点之间的箭头。只要有了指向表首结点的指针 head，就可以从表首结点出发，顺着 next 指针链找到全部结点。表尾结点的 next 指针设置为 None(在图中用“^”表示)。结点中的数据，类型是相同的。在 Python 语言中，数据表面上看可以不同，比如有的是整数，有的是字符串，有的是对象，但是归根到结点中的数据只是一个指针，只是这个指针可以指向不同类型的数据而已。

对于有 head 和 tail 指针的单链表，删除表首结点，或者在表首结点前面插入结点，复杂度是 $O(1)$的。在表尾后面添加结点，复杂度也是 $O(1)$的。但是要删除表尾结点，复杂度是 $O(n)$的，因为需要先从表首结点出发顺着 next 指针链找到表尾的前驱结点。

单链表可以实现为下面的 LinkList 类。

```
#prg0260.py
1.  class LinkList:
2.      class Node:
3.          def __init__(self, data, next=None):
4.              self.data, self.next = data, next
5.      def __init__(self):
6.          self.head = self.tail = None
7.          self.size = 0
```

Node 是内部类，用于表示链表的结点。data 是数据，next 是指向后继结点的指针。

链表对象初始化为一个空链表，head 和 tail 都是 None，size 值为 0。

遍历并输出一个单链表全部内容的方法如下。

```
8.      def printList(self):
9.          ptr = self.head
10.         while ptr is not None:
11.             print(ptr.data, end=",")
12.             ptr = ptr.next
```

如果已经定位到了结点 p，在结点 p 后面插入新结点 nd 的过程如图 3.2～图 3.4 所示。

(1) 执行 nd = Node(data,None)，新建结点 nd。

(2) 执行 nd.next = p.next，如图 3.3 所示。

(3) 执行 p.next = nd，完成插入，如图 3.4 所示。

相应的方法如下。

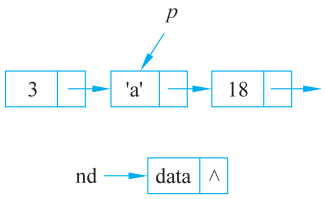

图 3.2 链表插入新结点步骤(1)

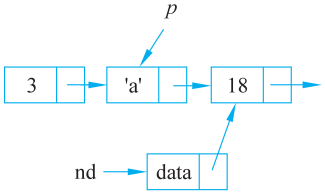

图 3.3 链表插入新结点步骤(2)

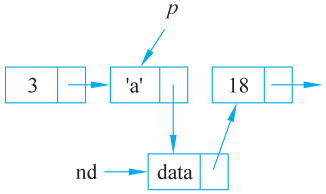

图 3.4 链表插入新结点步骤(3)

```
13.     def insert(self,p,data):                #在结点 p 后面插入元素
14.         nd = LinkList.Node(data,None)
15.         if self.tail is p:                  #新增的结点是新表尾
16.             self.tail = nd
17.         nd.next = p.next
18.         p.next = nd
19.         self.size += 1
```

如果已经定位到了结点 p，删除 p 的后继结点的过程如图 3.5 和图 3.6 所示。

(1) 初始状态，将要删除'a'结点，如图 3.5 所示。

(2) 执行 p.next = p.next.next，完成删除，如图 3.6 所示。

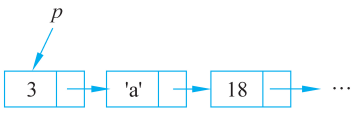

图 3.5 链表删除结点步骤(1)

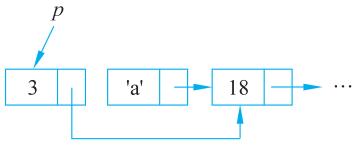

图 3.6 链表删除结点步骤(2)

相应的方法为：

```
20.     def delete(self,p):             #删除 p 后面的结点，调用时要确保 p 后面有结点
21.         if self.tail is p.next:     #要删除的结点是表尾结点
22.             self.tail = p
23.         p.next = p.next.next
24.         self.size -= 1
```

定位到了结点 p，要删除结点 p 是困难的，因为必须找到 p 的前驱。而找前驱，就需要从 head 开始往后找，这样复杂度就不是 $O(1)$的了。

在 C/C++ 语言中，从链表中删除结点 p 后，还需要写一行代码释放结点 p 占用的空间，但是在 Python 和 Java 中，不用操心这一点。等到 p 被重新赋值，或者 p 是局部变量且包含它的函数返回，结点 p 的空间就会被释放。

在链表前端插入一个元素的函数如下。

```
25.     def pushFront(self,data):               #在链表前端插入一个元素 data
26.         nd = LinkList.Node(data, self.head)
27.         self.head = nd
28.         self.size += 1
29.         if self.tail is None:
30.             self.tail = nd
```

第 29 行：判断一个变量 p 是不是 None，最好写“p is None”，“p is not None”，而不要写“p == None”，“p ! = None”。因为 p 有可能是个对象，其所属的类，有可能会重写了__eq__ 函数，那样的话用“==”或“! =”就很可能导致 runtime error。

删除链表前端元素的方法为：

```
31.     def popFront(self):
32.         if self.head is None:
```

```
            raise Exception("Popping front for Empty link list.")
        else:
            self.head = self.head.next
            self.size -= 1
            if self.size == 0:
                self.head = self.tail = None
```

在链表尾部添加元素的函数是：

```
    def pushBack(self,data):
        if self.size == 0:
            self.pushFront(data)
        else:
            self.insert(self.tail,data)
```

清空链表的函数是：

```
    def clear(self):
        self.head = self.tail = None
        self.size = 0
```

Python 会自动回收原链表结点的空间。

还可以为 LinkList 类添加__iter__方法和__next__方法，使其可以用 for 循环遍历：

```
    def __iter__(self):
        self.ptr = self.head
        return self
    def __next__(self):
        if self.ptr is None:
            raise StopIteration()          #引发异常
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data
    def index(self,i):                     #返回第 i 个元素
        n = 0
        ptr = self.head
        while n < i:
            ptr = ptr.next
            n += 1
        return ptr

linkLst = LinkList()
linkLst.pushFront(0)
linkLst.pushFront(1)
for i in range(2,5):
    linkLst.pushBack(i)
for x in linkLst:                          #>>1,0,2,3,4
    print(x,end=",")
```

从工程实践上来说，上面的 LinkList 类不完备且有缺陷，因为没有实现“隐藏”，类内部的成员变量都暴露在外，很容易因对成员变量的误操作导致整个链表崩溃。如何在用类实现一个数据结构时，进行“隐藏”以保护数据结构不被不小心破坏，可参看 3.2.3 节“双链表”。

为了避免处理在表首尾增删、空表等特殊情况，简化编程，在实现单链表时，常常设置一个空闲的头结点，即使是空表，也包含该头结点。这样的链表称为“带头结点的链表”。如图 3.7 和图 3.8 所示，图中数据部分为阴影的结点就是头结点。

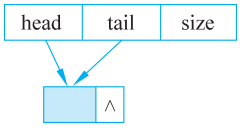

图 3.7　带头结点的空单链表

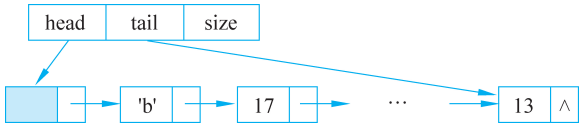

图 3.8　带头结点的非空单链表

前面特意用“表首结点”来称呼链表中最靠前的存有有效数据的结点，以和这里的“头结点”进行区分。“头结点”特指不包含有效数据的冗余结点。

带头结点的单链表的构造函数应如下编写。

```
class LinkList:
    def __init__(self):
        self.head = self.tail = LinkList.Node(None,None)
        self.size = 0
```

3.2.2　循环单链表

在单链表中，若将表尾结点的 next 指针由 None 改为指向表首结点，next 指针链就形成了循环，这样的单链表称为循环单链表。在循环单链表中，可以只设置表尾指针 tail，因为 tail.next 即是表首。只设置表首指针 head 则不好，因为要找到表尾需要遍历整个链表。图 3.9 和图 3.10 则是带头结点的循环单链表。

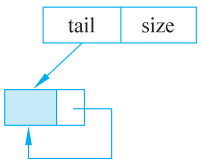

图 3.9　带头结点的空单循环链表

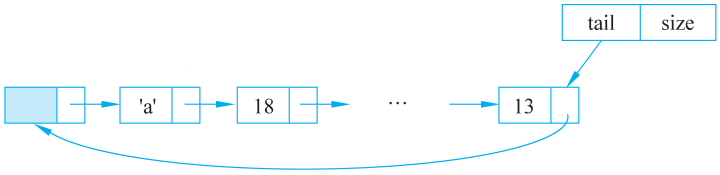

图 3.10　带头结点的非空单循环链表

循环单链表在表首或表尾增加元素，以及删除表首元素，复杂度都是 $O(1)$的。

3.2.3　双链表

在单链表中要找结点的前驱，只能从表首开始往后找，复杂度是 $O(n)$的。为了消除这一不便，引入了双链表，也叫双向链表。双链表中每个结点不但有指向后继的指针 next，还有指向前驱的指针 prev。一个带头结点的双链表如图 3.11 所示，头结点的 prev 为 None，尾结点的 next 为 None。

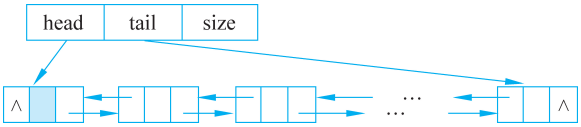

图 3.11　带头结点的双链表

如果已经定位到了结点 p，在结点 p 后面插入新结点 nd 的过程如下。

(1) 执行 nd = Node(data,None,None)新建结点 nd，如图 3.12 所示。

(2) 执行 nd.prev, nd.next = p, p.next，如图 3.13 所示。

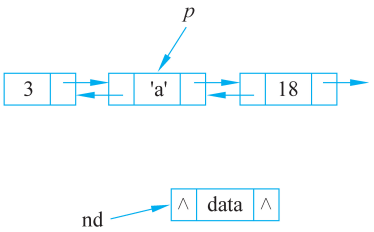

图 3.12　双链表插入结点结点步骤(1)

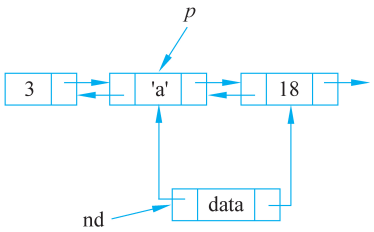

图 3.13　双链表插入结点结点步骤(2)

(3) 若 p.next 不为 None，则执行 p.next.prev = nd，如图 3.14 所示。

(4) 执行 p.next = nd，插入完成，如图 3.15 所示。

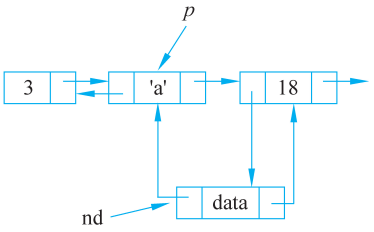

图 3.14　双链表插入结点步骤(3)

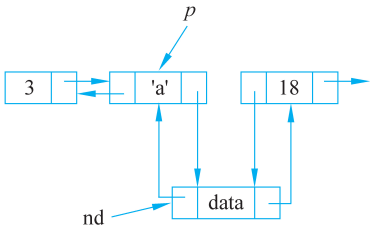

图 3.15　双链表插入结点步骤(4)

注意，上面的步骤(3)和(4)的次序是一定不能颠倒的。

如果已经定位到了结点 p，删除结点 p 的过程如下。

(1) 初始状态，将要删除'a'结点，如图 3.16 所示。

(2) 执行 p.prev.next = p.next，如图 3.17 所示。

(3) 若 p.next 不为 None，则执行 p.next.prev = p.prev，完成删除，如图 3.18 所示。

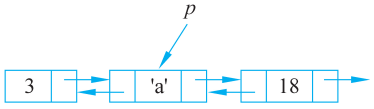

图 3.16　双链表删除结点步骤(1)

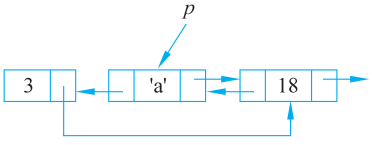

图 3.17　双链表删除结点步骤(2)

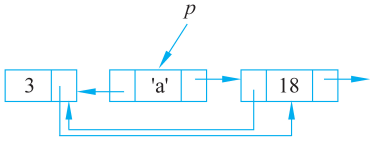

图 3.18　双链表删除结点步骤(3)

下面这个带头结点的双链表的实现，模仿 C++ 语言 STL 中的 list 进行了封装和隐藏。使用者只能通过 DoubleLinkList 类中不以下画线打头的函数来使用链表，不会破坏链表内部的结构。这部分内容的重点是封装和隐藏，而非链表操作，对计算机专业读者也属于较高

要求，非计算机专业的读者可以跳过。

在这个实现中，要访问链表元素，必须通过一个“迭代器”，即 DoubleLinkList 类的内部类_Iterator 的对象来进行，“迭代器”包含指向链表元素的指针，因此也可以说迭代器指向链表元素。迭代器的 getData()和 setData()方法用来访问列表元素的数据部分。对迭代器 i，next(i)会返回 i 指向的元素的后继的迭代器。DoubleLinkList 类的 begin()函数返回指向第一个元素的迭代器，由它开始就可以访问整个链表。

受篇幅所限，这个实现并不完备，没有提供由后往前遍历链表的手段，读者可以自行研究加上。

```
#prg0270.py
class DoubleLinkList:
    class _Node:
        def __init__(self, data, prev=None, next=None):
            self.data, self.prev, self.next = data, prev, next
    class _Iterator:
        def __init__(self,p):
            self.ptr = p
        def getData(self):
            return self.ptr.data
        def setData(self,data):
            self.ptr.data = data
        def __next__(self):
            self.ptr = self.ptr.next
            if self.ptr is None:
                return None
            else:
                return DoubleLinkList._Iterator(self.ptr)
        def prev(self):
            self.ptr = self.ptr.prev
            return DoubleLinkList._Iterator(self.ptr)

    def __init__(self):
        self._head = self._tail = \
            DoubleLinkList._Node(None,None,None)
        self._size = 0
    def _insert(self,p,data):
        nd = DoubleLinkList._Node(data,p,p.next)
        if self._tail is p:                    #新增的结点是新表尾
            self._tail = nd
        if p.next:
            p.next.prev = nd
        p.next = nd
        self._size += 1
    def _delete(self,p):                       #删除结点 p
        if self._size == 0 or p is self._head:
            raise Exception("Illegal deleting.")
```

```
        else:
            p.prev.next = p.next
            if p.next:                          #如果 p 有后继
                p.next.prev = p.prev
            if self._tail is p:
                self._tail = p.prev
            self._size -= 1
    def clear(self):
        self._tail = self._head
        self._head.next = self._head.prev = None
        self._size = 0
    def begin(self):
        return DoubleLinkList._Iterator(self._head.next)
    def end(self):
        return None
    def insert(self,i,data):                    #在迭代器 i 指向的结点后面插入元素
        self._insert(i.ptr,data)
    def delete(self, i):                        #删除迭代器 i 指向的结点
        self._delete(i.ptr)
    def pushFront(self,data):                   #在链表前端插入一个元素
        self._insert(self._head,data)
    def popFront(self):
        self._delete(self._head.next)
    def pushBack(self,data):
        self._insert(self._tail,data)
    def popBack(self):
        self._delete(self._tail)
    def __iter__(self):
        self.ptr = self._head.next
        return self
    def __next__(self):
        if self.ptr is None:
            raise StopIteration()               #引发异常
        else:
            data = self.ptr.data
            self.ptr = self.ptr.next
            return data
    def find(self,val):             #查找元素 val,找到返回迭代器,找不到返回 None
        ptr = self._head.next
        while ptr is not None:
            if ptr.data == val:
                return DoubleLinkList._Iterator(ptr)
            ptr = ptr.next
        return self.end()
    def printList(self):
        ptr = self._head.next
        while ptr is not None:
            print(ptr.data,end=",")
            ptr = ptr.next

```

```
88. linkLst = DoubleLinkList()
89. for i in range(5):
90.     linkLst.pushBack(i)
91. i = linkLst.begin()
92. while i != linkLst.end():                    #>>0,1,2,3,4
93.     print(i.getData(),end = ",")
94.     i = next(i)
95. print()
96. i = linkLst.find(3)
97. i.setData(300)
98. linkLst.printList()                          #>>0,1,2,300,4
99. print()
100.linkLst.insert(i,6000)                       #在 i 后面插入 6000
101.linkLst.printList()                          #>>0,1,2,300,6000,4
102.print()
103.linkLst.delete(i)
104.linkLst.printList()                          #>>0,1,2,6000,4
```

3.2.4 静态链表

可以将链表存储于顺序表中，这样的链表称为静态链表。顺序表中的每个元素，除了存放数据的部分，还包含一个 next 指针，实际上是一个整数，表示链表中下一个元素在顺序表中的下标。顺序表下标为 0 的元素不存放数据，其 next 指针指明了链表第一个元素的下标。链表最后一个元素的 next 指针为 None。图 3.19 展示了一个静态链表，链表中的元素按顺序是 78,81,92,49,52。

下标	0	1	2	3	4	5	6	7	8	9
data		52	81	92		78		49		
next	5	None	3	7		2		1		

图 3.19 静态链表

在静态链表中插入元素比较简单，将新元素添加到顺序表末尾，然后修改一些指针即可。删除元素，只能让被删除的元素所占的单元空着，并不回收其内存空间。例如图 3.19 中下标为 4、6 的单元，就可能是在删除元素后留下的空白单元。删除操作会造成一些空间浪费。可以在顺序表空白单元的比例超过某个阈值的时候重新分配一个顺序表，将原表中的内容复制过去。

一般提到链表的时候指的都不是静态链表，本书也是如此。

3.3 顺序表和链表的选择

由于顺序表支持随机访问而链表不支持，所以顺序表的使用场景远远多于链表。一些老的数据结构教材，会说在最大元素个数不能确定或者为了节约空间，不希望总按最大可能元素个数来开设顺序表或元素需要动态增减等场合，应该使用链表。这个结论早已过时。现在除了 C 语言，各种常用的语言如 C++、Java、Python 等都提供了能够很方便动态添加元

素的顺序表。而且说到节约空间，在 Java 和 Python 这类所有变量都是指针的语言中，相比顺序表的冗余空间，链表元素中的 next 指针其实浪费了更多的空间，所以从空间的角度来说，链表相比顺序表基本没有优势。

列表的元素在内存中是连续存放的，而链表的元素却不是，这会导致在现实计算机系统中前者访问效率远好于后者。现代计算机的 CPU 中都有 cache(高速缓存)，其访问速度比内存快数倍到数十倍。当 CPU 访问某内存单元时，会将该内存单元附近的一片连续内存区域的内容读取到 cache(这叫内存预取)，下次再访问同一区域的内容时，就只需要从 cache 读取数据，不必再访问速度较慢的内存。甚至对内存的写入，也可以只暂时写入 cache，在适当时候才写入内存。因此，遍历同样多元素的顺序表和链表，前者速度比后者快数倍甚至数十倍。此外，链表结点的空间回收是逐个结点进行的，需要花 $O(n)$ 的时间，而顺序表的空间是连续的，回收一般只需要 $O(1)$ 时间。总体来说，在绝大多数应用场景中，链表相比顺序表，无论是编程效率、时间效率还是空间效率，都处于劣势。

与顺序表相比，链表的真正优点是在表中间插入和删除元素的复杂度是 $O(1)$ 的，但前提是已经找到了要增删元素的位置。对于常见的要在第 i 个元素处插入元素，或删除第 i 个元素、删除等于 x 的元素这样的操作，在链表中首先要找到第 i 个元素或找到等于 x 的元素(或它们的前驱元素)，这也要花 $O(n)$ 的时间，因此和顺序表相比没有优势。

只有在线性表的中间某个位置附近频繁地做增删操作，链表相比顺序表的优势才能体现出来，因为只需要花一次 $O(n)$ 时间找到这个位置，以后多次增删就都是 $O(1)$ 的。但是现实中需要这样做的场景不多，绝大多数场景都是在表头部或尾部进行增删，如栈和队列这样的数据结构。即便需要 $O(1)$ 时间在线性表头部进行增删，也可以使用基于顺序表实现的循环双向队列，还是没有必要用链表(有些语言的双向队列内部实现结合了顺序表和链表)。

在某些特殊的场合，可以考虑使用静态链表，有可能同时得到链表插入删除快和连续内存访问快的优势。

总之，在软件开发的实践中，只要不是用 C 语言，哪怕是用 C++ 语言，需要使用链表的情况都很少。在操作系统这样的用 C/C++ 语言实现的底层软件系统中，以及一些语言自带的库中，链表的应用才会多些。

链表能和顺序表取得相同地位的场景，大概就只有数据结构考试的笔试了——因为链表方面容易出题。甚至在 OpenJudge 上的机考都很难考链表，因为不能用顺序表而必须用链表才能在时限内通过的考题，设计起来比较麻烦而且题目往往显得不自然，所以现成的题目也很少。

小　　结

线性表分为顺序表和链表两种。前者元素在内存中连续存放，支持随机访问操作($O(1)$时间访问第 i 个元素)，后者元素在内存中不连续存放，不支持随机访问操作。

顺序表在尾部增删元素的复杂度是 $O(1)$，在中间增删元素的复杂度是 $O(n)$。链表在确定增删位置的前提下，增删元素复杂度是 $O(1)$。

顺序表需要预先多分配空间，才能支持 $O(1)$ 时间的尾部添加元素。预分配的策略是每次重新分配空间时，分配的容量是元素个数的 k 倍(k 是大于 1 的固定值)。

为链表设置头结点可以提高编程实现的方便性。

由于 CPU 有 cache，所以遍历同样多元素的顺序表和链表，前者要快得多。

习　　题

1. 顺序表中下标为 0 的元素的地址是 x，每个元素占 4B，则下标为 20 的元素的地址是(地址以 B 为单位)(　　)。

A. x+80　　B. x+20　　C. x+84　　D. 无法确定

2. 以下 4 种操作，有几种的时间复杂度是 $O(1)$的？(　　)

(1) 在顺序表尾部添加一个元素

(2) 找到链表的第 i 个元素

(3) 求链表元素个数

(4) 在链表头部插入一个元素

A. 1　　B. 2　　C. 3　　D. 4

3. 链表不具有的特点是(　　)。

A. 可随机访问任意元素

B. 插入和删除不需要移动元素

C. 不必事先预分配存储空间

D. 所需空间与链表长度成正比(静态链表除外)

以下为编程题。本书的编程案例和习题均可在配套网站北京大学在线程序评测平台 OpenJudge 的“程序设计实习 MOOC”小组中与书名相同的题集中进行提交。每道题都有编号，如 P0010、P0020。

1. 合并有序序列(P0010)：给定两个从小到大排好序的整数序列，长度分别为 m 和 n，要求用 $O(m+n)$时间将其合并为一个新的有序序列。

2. 删除链表元素(P0020)：为 3.2.1 节中的单链表程序 prg0260.py 中的 LinkList 类添加一个方法 remove(self,data)，用以删除值为 data 的元素。如果有多个，只删除最前面的那个。

3. 约瑟夫问题(P0030)：用链表模拟解决猴子选大王问题：n 个猴子编号 1～n，以顺时针顺序排成一个圆圈。从 1 号猴子开始顺时针方向不停报数，1 号猴子报 1，2 号猴子报 2，…，报到 m 的猴子出圈，下一个猴子再从 1 开始报。最后剩下的那只猴子是大王。求大王的编号。

4. 颠倒链表(P0040)：写一个函数，将一个单链表颠倒过来。要求额外空间复杂度为 $O(1)$，即原链表以外的存储空间是 $O(1)$的。

★5. 寻找链表交叉点(P0050)：两个单链表，从某个结点开始，后面所有的结点都是两者共享的(类似于字母 Y 形状)。请设计 $O(n)$算法求两者共享的第一个结点(n 是两者中间较长的链表的长度)。

★★6. 链表寻环(P0060)：请设计时间复杂度 $O(n)$的算法判断一个单链表是否有环。要求额外空间复杂度 $O(1)$。有环的意思，就是最后一个元素的 next 指针，指向了链表中的某个元素。