目录

湖南工商大学 算法导论 课程教案

授课题目（教学章、节或主题） **课时安排: 2学时**

第八讲：基本数据结构 **授课时间 :第八周周一第1、2节**

教学内容（包括基本内容、重点、难点）：

基本内容： （1）基本数据结构的概念：栈、队列、列表、链表；

（2）栈的实现；

（3）队列的实现;

（4）列表与链表的实现.

教学重点、难点：重点为基本数据结构的原理、基本数据结构的程序实现与应用

教学媒体的选择：本讲使用大数据分析软件Jupyter教学，Jupyter集课件、Python程序运行、HTML网页制作、Pdf文档生成、Latex文档编译于一身，是算法导论课程教学的最佳选择。

板书设计：黑板分为上下两块，第一块基本定义，推导证明以及例子放在第二块。第一块 整个课堂不擦洗，以便学生随时看到算法流程图以及基本算法理论等内容。

课程过程设计： （1）讲解基本算法理论； （2）举例说明； （3）程序设计与编译； （4）对本课堂进行总 结、讨论； （5）布置作业与实验报告

一. 基本数据结构简介

1. 目标

- 理解抽象数据类型的栈，队列，deque 和列表。
- 能够使用 Python 列表实现 ADT 堆栈，队列和 deque。
- 了解基本线性数据结构实现的性能。
- 了解前缀，中缀和后缀表达式格式。
- 使用栈来实现后缀表达式。
- 使用栈将表达式从中缀转换为后缀。
- 使用队列进行基本时序仿真。
- 能够识别问题中栈，队列和 deques 数据结构的适当使用。
- 能够使用节点和引用将抽象数据类型列表实现为链表。
- 能够比较我们的链表实现与 Python 的列表实现的性能。

2. 什么是线性数据结构

我们从四个简单但重要的概念开始研究数据结构。栈，队列，deques, 列表是一类数据的容器，它们数据项之间的顺序由添加或删除的顺序决定。一旦一个数据项被添加，它相对于前后元素一直保持该位置不变。诸如此类的数据结构被称为线性数据结构。

线性数据结构有两端，有时被称为左右，某些情况被称为前后。你也可以称为顶部和底部，名字都不重要。将两个线性数据结构区分开的方法是添加和移除项的方式，特别是添加和移除项的位置。例如一些结构允许从一端添加项，另一些允许从另一端移除项。

这些变种的形式产生了计算机科学最有用的数据结构。他们出现在各种算法中，并可以用于解决很多重要的问题。

一. 栈

1. 栈的基本概念

栈（有时称为“后进先出栈”）是一个项的有序集合，其中添加移除新项总发生在同一端。这一端通常称为“顶部”。与顶部对应的端称为“底部”。

栈的底部很重要，因为在栈中靠近底部的项是存储时间最长的。最近添加的项是最先会被移除的。这种排序原则有时被称为 LIFO，后进先出。它基于在集合内的时间长度做排序。较新的项靠近顶部，较旧的项靠近底部。

栈的例子很常见。几乎所有的自助餐厅都有一堆托盘或盘子，你从顶部拿一个，就会有一个新的托盘给下一个客人。想象桌上有一堆书(Figure 1), 只有顶部的那本书封面可见，要看到其他书的封面，只有先移除他们上面的书。Figure 2 展示了另一个栈，包含了很多 Python 对象。

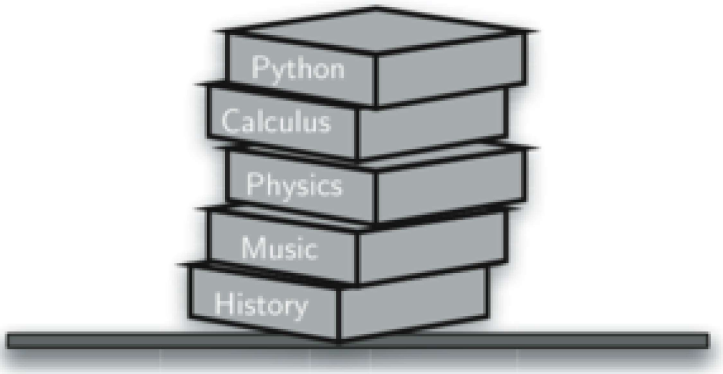

Figure 1

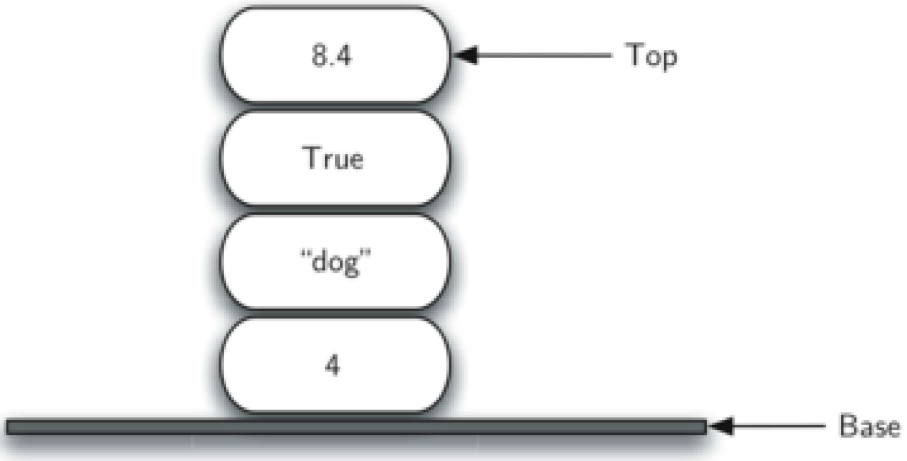

Figure 2

和栈相关的最有用的想法之一来自对它的观察。假设从一个干净的桌面开始，现在把书一本本叠起来，你在构造一个栈。考虑下移除一本书会发生什么。移除的顺序跟刚刚被放置的顺序相反。栈之所以重要是因为它能反转项的顺序。插入跟删除顺序相反，Figure 3 展示了 Python 数据对象创建和删除的过程，注意观察他们的顺序。

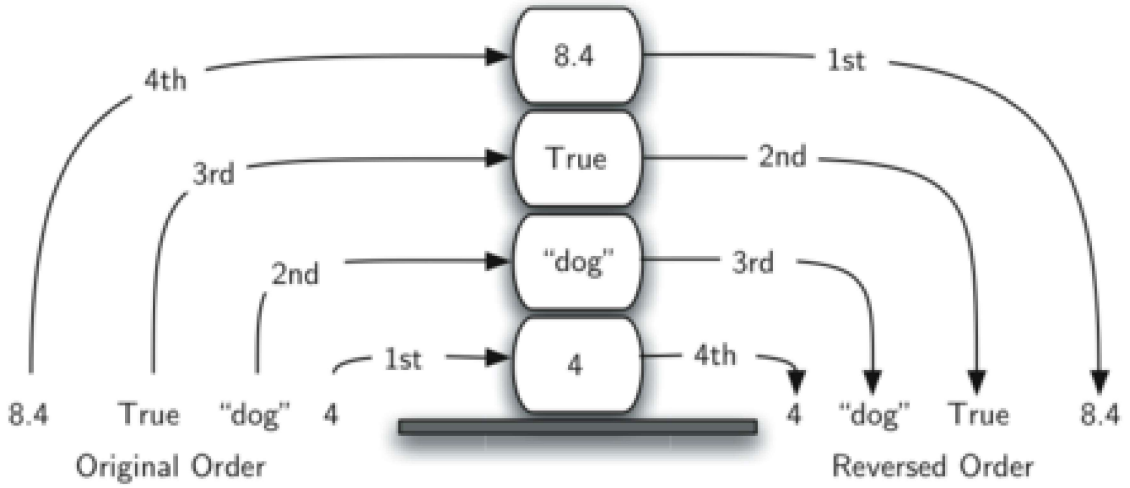

Figure 3

想想这种反转的属性，你可以想到使用计算机的时候所碰到的例子。例如，每个 web 浏览器都有一个返回按钮。当你浏览网页时，这些网页被放置在一个栈中（实际是网页的网址）。你现在查看的网页在顶部，你第一个查看的网页在底部。如果按'返回'按钮，将按相反的顺序浏览刚才的页面。

2. 栈的抽象数据类型

栈的抽象数据类型由以下结构和操作定义。如上所述，栈被构造为项的有序集合，其中项被添加和从末端移除的位置称为"顶部"。栈是有序的 LIFO 。栈操作如下。

- Stack() 创建一个空的新栈。 它不需要参数，并返回一个空栈。
- push(item)将一个新项添加到栈的顶部。它需要 item 做参数并不返回任何内容。
- pop() 从栈中删除顶部项。它不需要参数并返回 item 。栈被修改。
- peek() 从栈返回顶部项，但不会删除它。不需要参数。 不修改栈。
- isEmpty() 测试栈是否为空。不需要参数，并返回布尔值。
- size() 返回栈中的 item 数量。不需要参数，并返回一个整数。

例如，s 是已经创建的空栈，Table1 展示了栈操作序列的结果。栈中，顶部项列在最右边。

Stack Operation	Stack Contents	Return Value
s.isEmpty()	[]	True
s.push(4)	[4]	
s.push('dog')	[4,'dog']	
s.peek()	[4,'dog']	'dog'
s.push(True)	[4,'dog',True]	
s.size()	[4,'dog',True]	3
s.isEmpty()	[4,'dog',True]	False
s.push(8.4)	[4,'dog',True,8.4]	
s.pop()	[4,'dog',True]	8.4
s.pop()	[4,'dog']	True
s.size()	[4,'dog']	2

Table1

3. Python实现栈

现在我们已经将栈清楚地定义了抽象数据类型，我们将把注意力转向使用 Python 实现栈。回想一下，当我们给抽象数据类型一个物理实现时，我们将实现称为数据结构。

正如我们在第1章中所描述的，在 Python 中，与任何面向对象编程语言一样，抽象数据类型（如栈）的选择的实现是创建一个新类。栈操作实现为类的方法。此外，为了实现作为元素集合的栈，使用由 Python 提供的原语集合的能力是有意义的。 我们将使用列表作为底层实现。

回想一下，Python 中的列表类提供了有序集合机制和一组方法。例如，如果我们有列表 [2,5,3,6,7,4]，我们只需要确定列表的哪一端将被认为是栈的顶部。一旦确定，可以使用诸如 append 和 pop 的列表方法来实现操作。

以下栈实现（ActiveCode 1）假定列表的结尾将保存栈的顶部元素。随着栈增长（push 操作），新项将被添加到列表的末尾。 pop 也操作列表末尾的元素。

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

ActiveCode 1

记住我们只定义类的实现，我们需要创建一个栈，然后使用它。ActiveCode 2 展示了我们通过实例化 Stack 类执行 Table 1中的操作。注意，Stack 类的定义是从 pythonds 模块导入的。

> **Note** pythonds 模块包含本书中讨论的所有数据结构的实现。它根据以下部分构造：基本数据类型，树和图。 该模块可以从 pythonworks.org (http://www.pythonworks.org/pythonds)下载。

```
from pythonds.basic.stack import Stack

s=Stack()

print(s.isEmpty())
s.push(4)
s.push('dog')
print(s.peek())
s.push(True)
print(s.size())
print(s.isEmpty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

ActiveCode 2

三. 队列

1. 队列的基本概念

队列是项的有序结合，其中添加新项的一端称为队尾，移除项的一端称为队首。当一个元素从队尾进入队列时，一直向队首移动，直到它成为下一个需要移除的元素为止。

最近添加的元素必须在队尾等待。集合中存活时间最长的元素在队首，这种排序成为 FIFO，先进先出，也被成为先到先得。

队列的最简单的例子是我们平时不时会参与的列。排队等待电影，在杂货店的收营台等待，在自助餐厅排队等待（这样我们可以弹出托盘栈）。行为良好的线或队列是有限制的，因为它只有一条路，只有一条出路。不能插队，也不能离开。你只有等待了一定的时间才能到前面。Figure 1 展示了一个简单的 Python 对象队列。

Figure 1

计算机科学也有常见的队列示例。我们的计算机实验室有 30 台计算机与一台打印机联网。当学生想要打印时，他们的打印任务与正在等待的所有其他打印任务“一致”。第一个进入的任务是先完成。如果你是最后一个，你必须等待你前面的所有其他任务打印。我们将在后面更详细地探讨这个有趣的例子。

除了打印队列，操作系统使用多个不同的队列来控制计算机内的进程。下一步做什么的调度通常基于尽可能快地执行程序和尽可能多的服务用户的排队算法。此外，当我们敲击键盘时，有时屏幕上出现的字符会延迟。这是由于计算机在那一刻做其他工作。按键的内容被放置在类似队列的缓冲器中，使得它们最终可以以正确的顺序显示在屏幕上。

2.队列抽象数据类型

队列抽象数据类型由以下结构和操作定义。如上所述，队列被构造为在队尾添加项的有序集合，并且从队首移除。队列保持 FIFO 排序属性。 队列操作如下。

- Queue() 创建一个空的新队列。 它不需要参数，并返回一个空队列。
- enqueue(item) 将新项添加到队尾。 它需要 item 作为参数，并不返回任何内容。
- dequeue() 从队首移除项。它不需要参数并返回 item。 队列被修改。
- isEmpty() 查看队列是否为空。它不需要参数，并返回布尔值。
- size() 返回队列中的项数。它不需要参数，并返回一个整数。

作为示例，我们假设 q 是已经创建并且当前为空的队列，则 Table 1 展示了队列操作序列的结果。右边表示队首。 4 是第一个入队的项，因此它 dequeue 返回的第一个项。

Queue Operation	Queue Contents	Return Value
`q.isEmpty()`	`[]`	`True`
`q.enqueue(4)`	`[4]`	
`q.enqueue('dog')`	`['dog',4]`	
`q.enqueue(True)`	`[True,'dog',4]`	
`q.size()`	`[True,'dog',4]`	`3`
`q.isEmpty()`	`[True,'dog',4]`	`False`
`q.enqueue(8.4)`	`[8.4,True,'dog',4]`	
`q.dequeue()`	`[8.4,True,'dog']`	`4`
`q.dequeue()`	`[8.4,True]`	`'dog'`
`q.size()`	`[8.4,True]`	`2`

Table 1

3.Python实现队列

我们为了实现队列抽象数据类型创建一个新类。和前面一样，我们将使用列表集合来作为构建队列的内部表示。

我们需要确定列表的哪一端作为队首，哪一端作为队尾。Listing 1 所示的实现假定队尾在列表中的位置为 0。这允许我们使用列表上的插入函数向队尾添加新元素。弹出操作可用于删除队首的元素（列表的最后一个元素）。回想一下，这也意味着入队为

$$O(n)$$

，出队为

$$O(1)$$

。

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

Listing 1

进一步的操作这个队列产生如下结果：

```
>>> q.size()
3
>>> q.isEmpty()
False
>>> q.enqueue(8.4)
>>> q.dequeue()
4
>>> q.dequeue()
'dog'
>>> q.size()
2
```

四. Deque（双端队列）

1. Deque的基本概念

deque（也称为双端队列）是与队列类似的项的有序集合。它有两个端部，首部和尾部，并且项在集合中保持不变。deque 不同的地方是添加和删除项是非限制性的。可以在前面或后面添加新项。同样，可以从任一端移除现有项。在某种意义上，这种混合线性结构提供了单个数据结构中的栈和队列的所有能力。 Figure 1 展示了一个 Python 数据对象的 deque 。

要注意，即使 deque 可以拥有栈和队列的许多特性，它不需要由那些数据结构强制的 LIFO 和 FIFO 排序。这取决于你如何持续添加和删除操作。

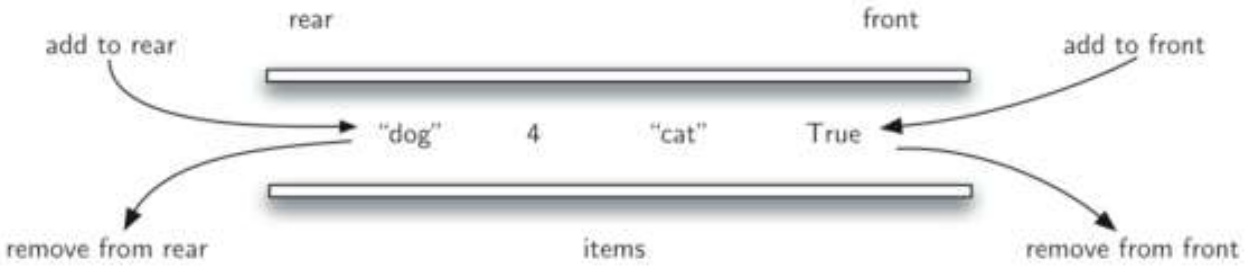

Figure 1

2. Deque抽象数据类型

deque 抽象数据类型由以下结构和操作定义。如上所述，deque 被构造为项的有序集合，其中项从首部或尾部的任一端添加和移除。下面给出了 deque 操作。

- Deque() 创建一个空的新 deque。它不需要参数，并返回空的 deque。
- addFront(item) 将一个新项添加到 deque 的首部。它需要 item 参数 并不返回任何内容。
- addRear(item) 将一个新项添加到 deque 的尾部。它需要 item 参数并不返回任何内容。
- removeFront() 从 deque 中删除首项。它不需要参数并返回 item。deque 被修改。
- removeRear() 从 deque 中删除尾项。它不需要参数并返回 item。deque 被修改。
- isEmpty() 测试 deque 是否为空。它不需要参数，并返回布尔值。
- size() 返回 deque 中的项数。它不需要参数，并返回一个整数。

例如，我们假设 d 是已经创建并且当前为空的 deque，则 Table 1 展示了一系列 deque 操作的结果。注意，首部的内容列在右边。在将 item 移入和移出时，跟踪前面和后面是非常重要的，因为可能会有点混乱。

Deque Operation	Deque Contents	Return Value
d.isEmpty()	[]	True
d.addRear(4)	[4]	
d.addRear('dog')	['dog',4,]	
d.addFront('cat')	['dog',4,'cat']	
d.addFront(True)	['dog',4,'cat',True]	
d.size()	['dog',4,'cat',True]	4
d.isEmpty()	['dog',4,'cat',True]	False
d.addRear(8.4)	[8.4,'dog',4,'cat',True]	
d.removeRear()	['dog',4,'cat',True]	8.4
d.removeFront()	['dog',4,'cat']	True

Table 1

3. Python实现Deque

正如我们在前面的部分中所做的，我们将为抽象数据类型 deque 的实现创建一个新类。同样，Python 列表将提供一组非常好的方法来构建 deque 的细节。我们的实现（Listing 1）将假定 deque 的尾部在列表中的位置为 0。

```
class Deque:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0,item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)
```

Listing 1

在 removeFront 中，我们使用 pop 方法从列表中删除最后一个元素。 但是，在removeRear中，pop(0)方法必须删除列表的第一个元素。同样，我们需要在 addRear 中使用insert方法（第12行），因为 append 方法在列表的末尾添加一个新元素。

你可以看到许多与栈和队列中描述的 Python 代码相似之处。你也可能观察到，在这个实现中，从前面添加和删除项是

$$O(1)$$

，而从后面添加和删除是

$$O(n)$$

。 考虑到添加和删除项是出现的常见操作，这是可预期的。 同样，重要的是要确定我们知道在实现中前后都分配在哪里。

五. 列表与链表

1. 列表的基本概念

在对基本数据结构的讨论中，我们使用 Python 列表来实现所呈现的抽象数据类型。列表是一个强大但简单的收集机制，为程序员提供了各种各样的操作。然而，不是所有的编程语言都包括列表集合。在这些情况下，列表的概念必须由程序员实现。

列表是项的集合，其中每个项保持相对于其他项的相对位置。更具体地，我们将这种类型的列表称为无序列表。我们可以将列表视为具有第一项，第二项，第三项等等。我们还可以引用列表的开头（第一个项）或列表的结尾（最后一个项）。为了简单起见，我们假设列表不能包含重复项。

例如，整数 $54, 26, 93, 17, 77$ 和 31 的集合可以表示考试分数的简单无序列表。请注意，我们将它们用逗号分隔，这是列表结构的常用方式。当然，Python 会显示这个列表为 $[54, 26, 93, 17, 77, 31]$。

2. 无序列表抽象数据类型

如上所述，无序列表的结构是项的集合，其中每个项保持相对于其他项的相对位置。下面给出了一些可能的无序列表操作。

- List() 创建一个新的空列表。它不需要参数，并返回一个空列表。
- add(item) 向列表中添加一个新项。它需要 item 作为参数，并不返回任何内容。假定该 item 不在列表中。
- remove(item) 从列表中删除该项。它需要 item 作为参数并修改列表。假设项存在于列表中。
- search(item) 搜索列表中的项目。它需要 item 作为参数，并返回一个布尔值。
- isEmpty() 检查列表是否为空。它不需要参数，并返回布尔值。
- size（）返回列表中的项数。它不需要参数，并返回一个整数。
- append(item) 将一个新项添加到列表的末尾，使其成为集合中的最后一项。它需要 item 作为参数，并不返回任何内容。假定该项不在列表中。
- index(item) 返回项在列表中的位置。它需要 item 作为参数并返回索引。假定该项在列表中。
- insert(pos，item) 在位置 pos 处向列表中添加一个新项。它需要 item 作为参数并不返回任何内容。假设该项不在列表中，并且有足够的现有项使其有 pos 的位置。
- pop() 删除并返回列表中的最后一个项。假设该列表至少有一个项。
- pop(pos) 删除并返回位置 pos 处的项。它需要 pos 作为参数并返回项。假定该项在列表中。

3. 实现无序列表：链表

为了实现无序列表，我们将构造通常所知的链表。回想一下，我们需要确保我们可以保持项的相对定位。然而，没有要求我们维持在连续存储器中的定位。例如，考虑 Figure 1 中所示的项的集合。看来这些值已被随机放置。如果我们可以在每个项中保持一些明确的信息，即下一个项的位置（参见 Figure 2），则每个项的相对位置可以通过简单地从一个项到下一个项的链接来表示。

Figure 1

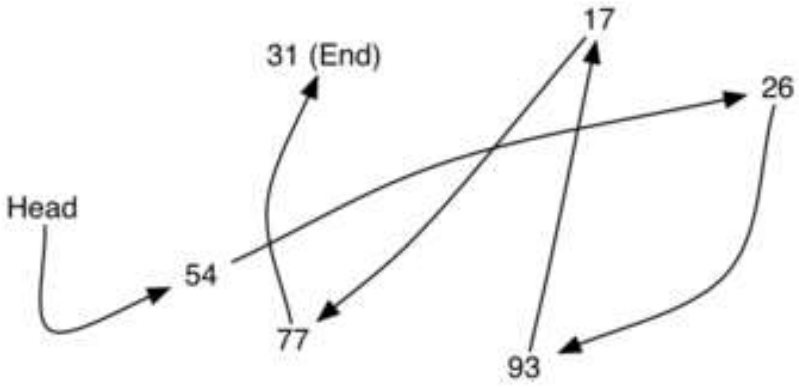

Figure 2

要注意，必须明确地指定链表的第一项的位置。一旦我们知道第一个项在哪里，第一个项目可以告诉我们第二个是什么，等等。外部引用通常被称为链表的头。类似地，最后一个项需要知道没有下一个项。

Node 类

链表实现的基本构造块是节点。每个节点对象必须至少保存两个信息。首先，节点必须包含列表项本身。我们将这个称为节点的数据字段。此外，每个节点必须保存对下一个节点的引用。Listing 1 展示了 Python 实现。要构造一个节点，需要提供该节点的初始数据值。下面的赋值语句将产生一个包含值 93 的节点对象（见 Figure 3）。应该注意，我们通常会如 Figure 4 所示表示一个节点对象。Node 类还包括访问，修改数据和访问下一个引用的常用方法。

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

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self,newdata):
        self.data = newdata

    def setNext(self,newnext):
        self.next = newnext
```

Listing 1

我们创建一个 Node 对象

```
>>> temp = Node(93)
>>> temp.getData()
93
```

Python 引用值 None 将在 Node 类和链表本身发挥重要作用。引用 None 代表没有下一个节点。请注意在构造函数中，最初创建的节点 next 被设置为 None。有时这被称为 接地节点 ，因此我们使用标准接地符号表示对 None 的引用。将 None 显式的分配给初始下一个引用值是个好主意。

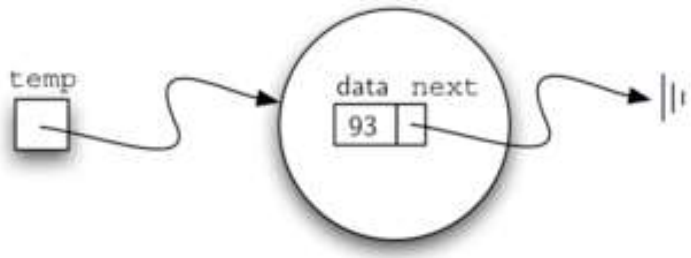

Figure 3

Figure 4

Unordered List 类

如上所述，无序列表将从一组节点构建，每个节点通过显式引用链接到下一个节点。只要我们知道在哪里找到第一个节点（包含第一个项），之后的每个项可以通过连续跟随下一个链接找到。考虑到这一点，UnorderedList 类必须保持对第一个节点的引用。Listing 2 显示了构造函数。注意，每个链表对象将维护对链表头部的单个引用。

```
class UnorderedList:

    def __init__(self):
        self.head = None
```

Listing 2

我们构建一个空的链表。赋值语句

```
>>> mylist = UnorderedList()
```

创建如 Figure 5 所示的链表。正如我们在 Node 类中讨论的，特殊引用 None 将再次用于表示链表的头部不引用任何内容。最终，先前给出的示例列表如 Figure 6 所示的链接列表表示。链表的头指代列表的第一项的第一节点。反过来，该节点保存对下一个节点（下一个项）的引用，等等。重要的是注意链表类本身不包含任何节点对象。相反，它只包含对链接结构中第一个节点的单个引用。

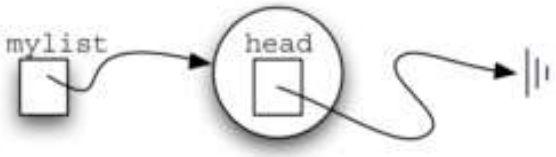

Figure 5

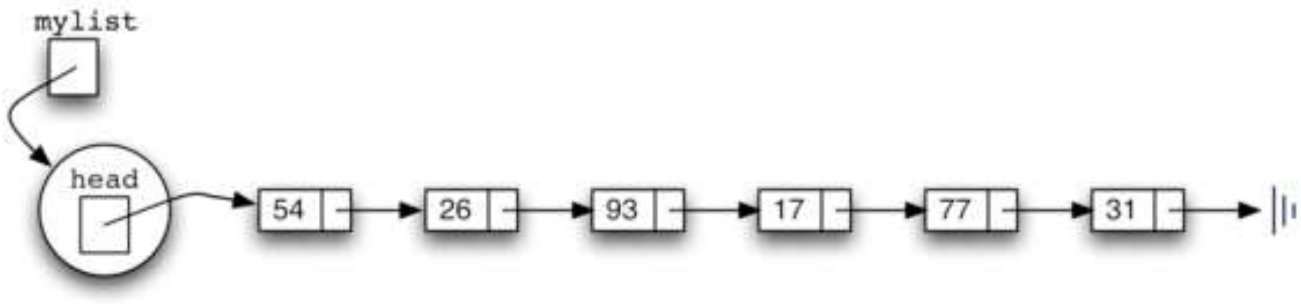

Figure 6

Listing 3 中所示的 isEmpty 方法只是检查链表头是否是 None 的引用。 布尔表达式 self.head == None 的结果只有在链表中没有节点时才为真。由于新链表为空，因此构造函数和空检查必须彼此一致。这显示了使用引用 None 来表示链接结构的 end 的优点。在 Python 中，None 可以与任何引用进行比较。如果它们都指向相同的对象，则两个引用是相等的。我们将在其他方法中经常使用它。

```
def isEmpty(self):
    return self.head == None
```

Listing 3

那么，我们如何将项加入我们的链表？我们需要实现 add 方法。然而，在我们做这一点之前，我们需要解决在链表中哪个位置放置新项的重要问题。由于该链表是无序的，所以新项相对于已经在列表中的其他项的特定位置并不重要。 新项可以在任何位置。考虑到这一点，将新项放在最简单的位置是有意义的。

回想一下，链表结构只为我们提供了一个入口点，即链表的头部。所有其他节点只能通过访问第一个节点，然后跟随下一个链接到达。这意味着添加新节点的最简单的地方就在链表的头部。 换句话说，我们将新项作为链表的第一项，现有项将需要链接到这个新项后。

Figure 6 展示了链表调用多次 add 函数的操作

```
>>> mylist.add(31)
>>> mylist.add(77)
>>> mylist.add(17)
>>> mylist.add(93)
>>> mylist.add(26)
>>> mylist.add(54)
```

Figure 6

因为 31 是添加到链表的第一个项，它最终将是链表中的最后一个节点，因为每个其他项在其前面添加。此外，由于 54 是添加的最后一项，它将成为链表的第一个节点中的数据值。

add 方法如 Listing 4 所示。链表的每项必须驻留在节点对象中。第 2 行创建一个新节点并将该项作为其数据。现在我们必须通过将新节点链接到现有结构中来完成该过程。这需要两个步骤，如 Figure 7 所示。步骤1（行3）更改新节点的下一个引用以引用旧链表的第一个节点。现在，链表的其余部分已经正确地附加到新节点，我们可以修改链表的头以引用新节点。第 4 行中的赋值语句设置列表的头。

上述两个步骤的顺序非常重要。如果第 3 行和第 4 行的顺序颠倒，会发生什么？如果链表头部的修改首先发生，则结果可以在 Figure 8 中看到。由于 head 是链表节点的唯一外部引用，所有原始节点都将丢失并且不能再被访问。

```
def add(self,item):
    temp = Node(item)
    temp.setNext(self.head)
    self.head = temp
```

Listing 4

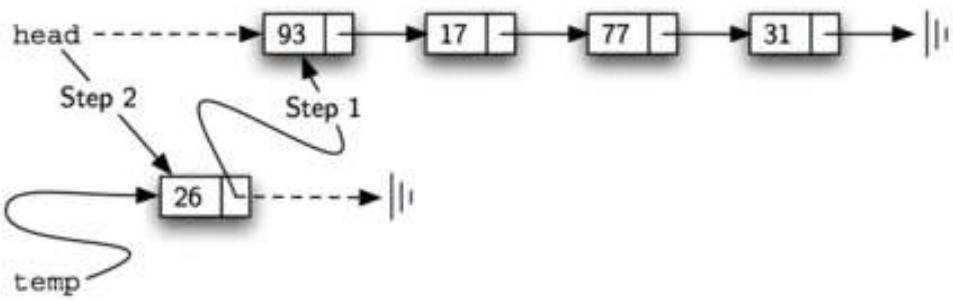

Figure 7

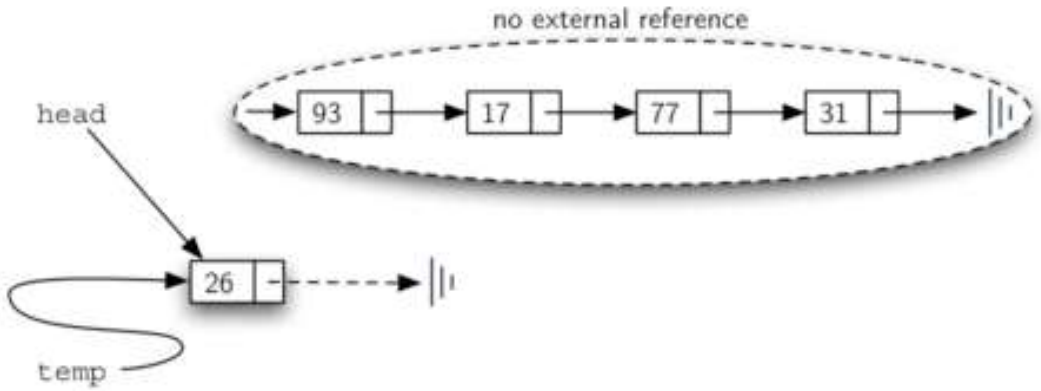

Figure 8

我们将实现的下面的方法 - `size`， `search` 和 `remove` - 都基于一种称为链表遍历的技术。遍历是指系统地访问每个节点的过程。为此，我们使用从链表中第一个节点开始的外部引用。当我们访问每个节点时，我们通过“遍历”下一个引用来移动到对下一个节点的引用。

要实现 `size` 方法，我们需要遍历链表并对节点数计数。Listing 5 展示了用于计算列表中节点数的 Python 代码。外部引用称为 `current`，并在第二行被初始化到链表的头部。开始的时候，我们没有看到任何节点，所以计数设置为 0 。第 4-6 行实际上实现了遍历。只要当前引用没到链表的结束位置（None），我们通过第 6 行中的赋值语句将当前元素移动到下一个节点。再次，将引用与 None 进行比较的能力是非常有用的。每当 `current` 移动到一个新的节点，我们加 1 以计数。最后， `count` 在迭代停止后返回。Figure 9 展示了处理这个链表的过程。

```
def size(self):
    current = self.head
    count = 0
    while current != None:
        count = count + 1
        current = current.getNext()

    return count
```

Listing 5

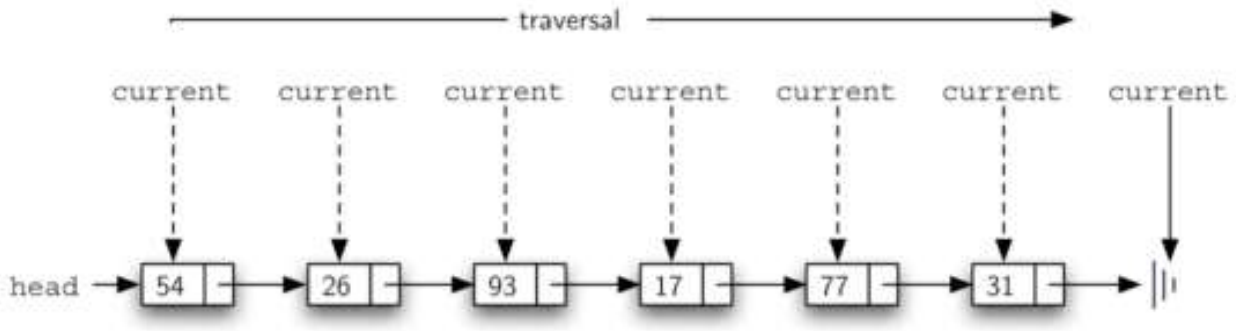

Figure 9

在链表中搜索也使用遍历技术。当我们访问链表中的每个节点时，我们将询问存储在其中的数据是否与我们正在寻找的项匹配。然而，在这种情况下，我们不必一直遍历到列表的末尾。事实上，如果我们到达链表的末尾，这意味着我们正在寻找的项不存在。此外，如果我们找到项，没有必要继续。

Listing 6 展示了搜索方法的实现。和在 `size` 方法中一样，遍历从列表的头部开始初始化（行 2）。我们还使用一个布尔变量叫 `found`，标记我们是否找到了正在寻找的项。因为我们还没有在遍历开始时找到该项， `found` 设置为 False（第3行）。第4行中的迭代考虑了上述两个条件。只要有更多的节点访问，而且我们没有找到正在寻找的项，我们就继续检查下一个节点。第 5 行检查数据项是否存在于当前节点中。如果存在， `found` 设置为 True。

```
    def search(self,item):
        current = self.head
        found = False
        while current != None and not found:
            if current.getData() == item:
                found = True
            else:
                current = current.getNext()

        return found
```

Listing 6

作为一个例子，试试调用 `search` 方法来查找 item 17

```
>>> mylist.search(17)
True
```

因为 17 在列表中，所以遍历过程需要移动到包含 17 的节点。此时， `found` 变量设置为 True，while 条件将失败，返回值。 这个过程可以在 Figure 10中看到。

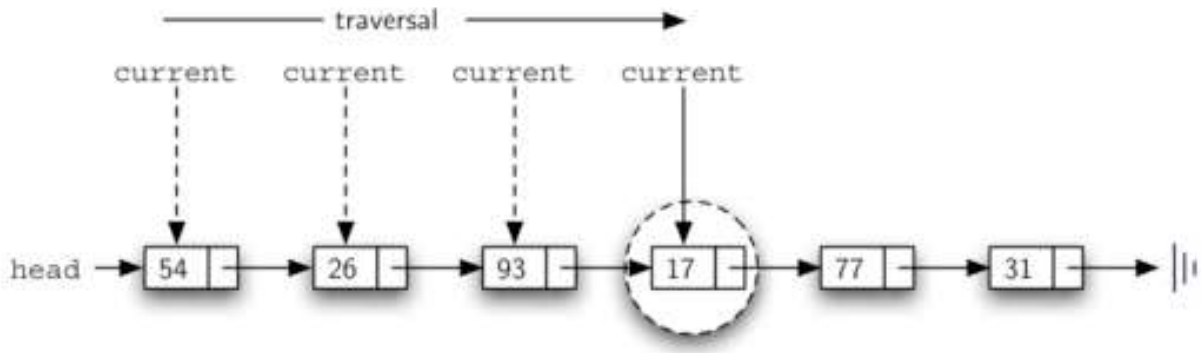

Figure 10

`remove` 方法需要两个逻辑步骤。首先，我们需要遍历列表寻找我们要删除的项。一旦我们找到该项（我们假设它存在），删除它。第一步非常类似于搜索。从设置到链表头部的外部引用开始，我们遍历链接，直到我们发现正在寻找的项。因为我们假设项存在，我们知道迭代将在 `current` 变为 `None` 之前停止。这意味着我们可以简单地使用 `found` 布尔值。

当 `found` 变为 True 时， `current` 将是对包含要删除的项的节点的引用。但是我们如何删除呢？一种方法是用标示该项目不再存在的某个标记来替换项目的值。这种方法的问题是节点数量将不再匹配项数量。最好通过删除整个节点来删除该项。

为了删除包含项的节点，我们需要修改上一个节点中的链接，以便它指向当前之后的节点。不幸的是，链表遍历没法回退。因为 current 指我们想要进行改变的节点之前的节点，所以进行修改太迟了。

这个困境的解决方案是在我们遍历链表时使用两个外部引用。 current 将像之前一样工作，标记遍历的当前位置。新的引用，我们叫 previous ，将总是传递 current 后面的一个节点 。这样，当 current 停止在要被去除的节点时， previous 将引用链表中用于修改的位置。

Listing 7 展示了完整的 remove 方法。第 2-3 行给这两个引用赋初始值。注意， current 在链表头处开始，和在其他遍历示例中一样。然而， previous 假定总是在 current 之后一个节点。因此，由于在 previous 之前没有节点，所以之前的值将为 None（见 Figure 11）。 found 的布尔变量将再次用于控制迭代。

在第 6-7 行中，我们检查存储在当前节点中的项是否是我们希望删除的项。如果是， found 设置为 True 。如果我们没有找到该项，则 previous 和 current 都必须向前移动一个节点。同样，这两个语句的顺序是至关重要的。 previous 必须先将一个节点移动到 current 的位置。此时，才可以移动 current 。这个过程通常被称为“英寸蠕动”，因为 previous 必须赶上 current ，然后 current 前进。Figure 12 展示了 previous 和 current 的移动，它们沿着链表向下移动，寻找包含值 17 的节点。

```
    def remove(self,item):
        current = self.head
        previous = None
        found = False
        while not found:
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()

        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
```

Listing 7

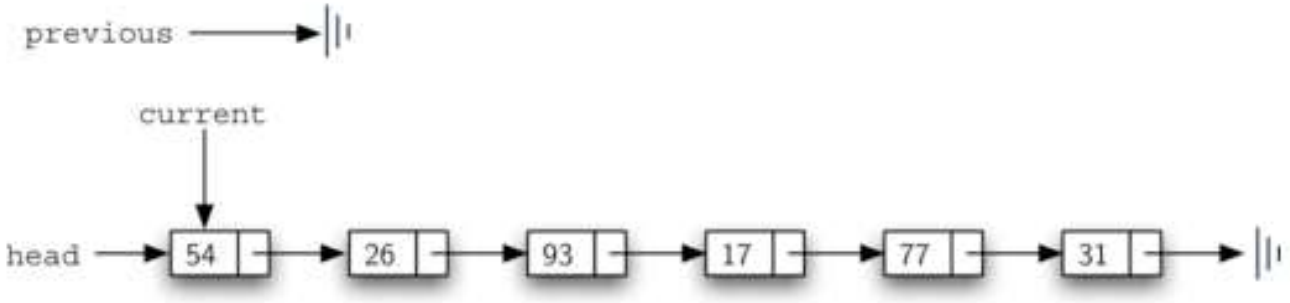

Figure 11

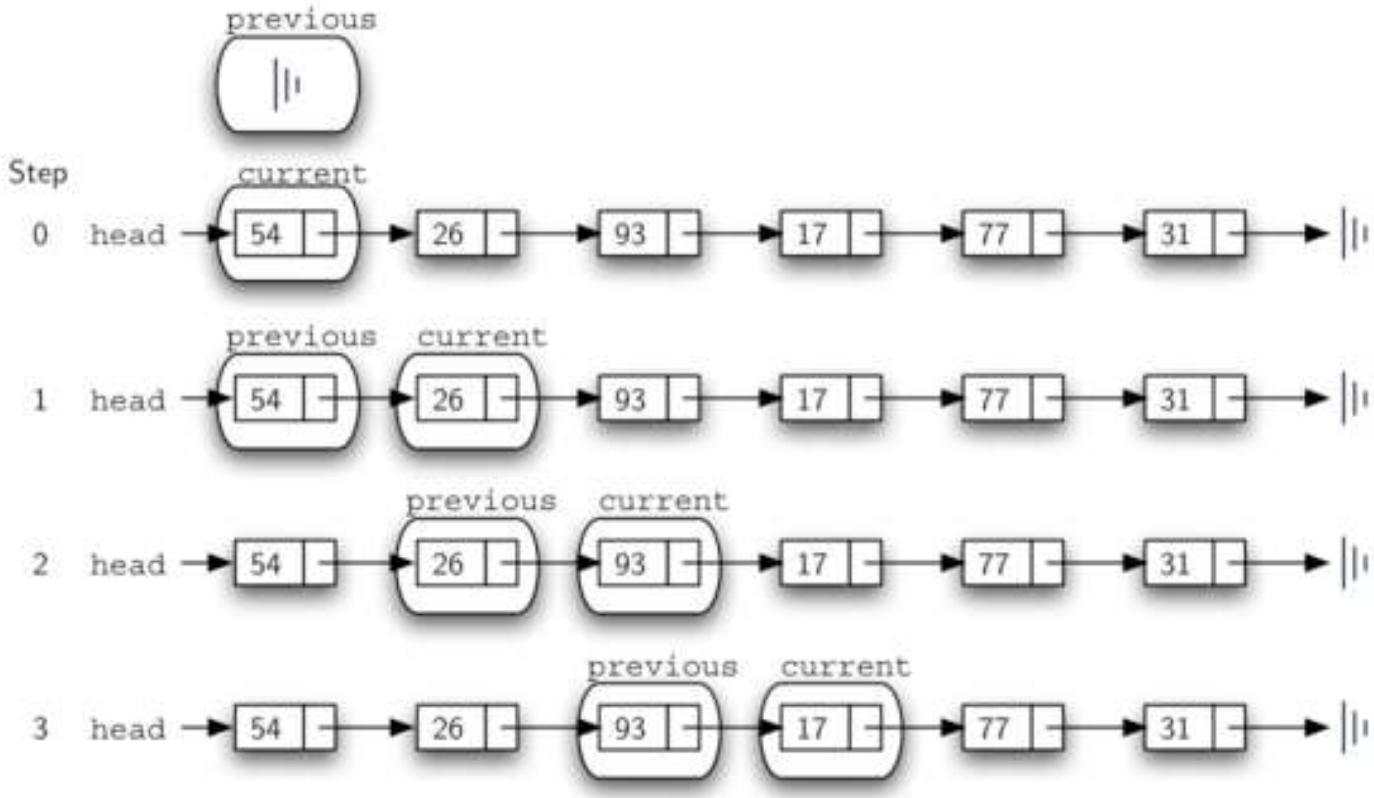

Figure 12

一旦 `remove` 的搜索步骤已经完成，我们需要从链表中删除该节点。 Figure 13 展示了要修改的链接。但是，有一个特殊情况需要解决。 如果要删除的项目恰好是链表中的第一个项，则 `current` 将引用链接列表中的第一个节点。这也意味着 `previous` 是 None。 我们先前说过，`previous` 是一个节点，它的下一个节点需要修改。在这种情况下，不是 `previous` ，而是链表的 `head` 需要改变（见 Figure 14）。

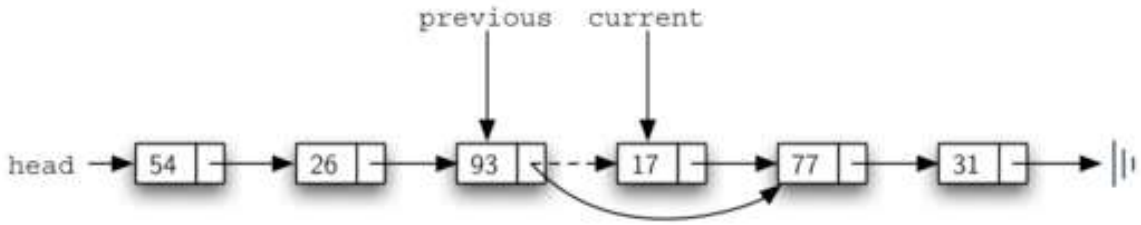

Figure 13

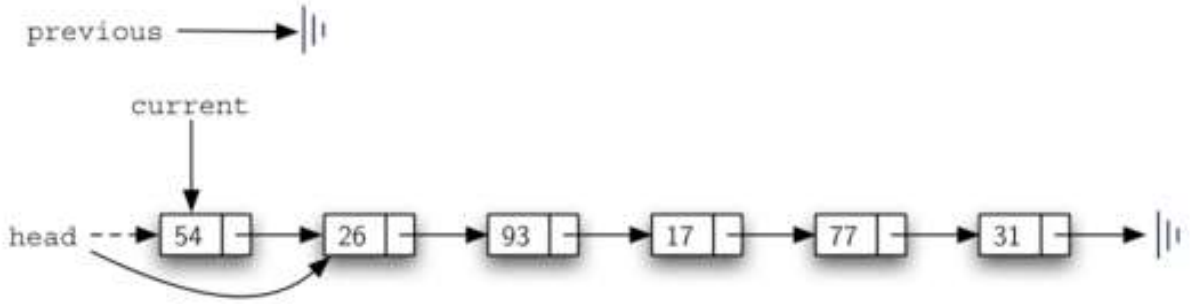

Figure 14

第 12 行检查是否处理上述的特殊情况。如果 `previous` 没有移动，当 `found` 的布尔变为 True 时，它仍是 None。 在这种情况下（行13），链表的 `head` 被修改以指代当前节点之后的节点，实际上是从链表中移除第一节点。 但是，如果 `previous` 不为 None，则要删除的节点位于链表结构的下方。 在这种情况下， `previous` 的引用为我们提供了下一个引用更改的节点。第 15 行使用之前的 `setNext` 方法完成删除。注意，在这两种情况下，引用更改的目标是 `current.getNext()` 。 经常出现的一个问题是，这里给出的两种情况是否也将处理要移除的项在链表的最后节点中的情况。我们留给你思考。

4. 有序列表抽象数据结构

我们现在将考虑一种称为有序列表的列表类型。例如，如果上面所示的整数列表是有序列表（升序），则它可以写为 `17,26,31,54,77和93` 。由于 17 是最小项，它占据第一位置。同样，由于 93 是最大的，它占据最后的位置。

有序列表的结构是项的集合，其中每个项保存基于项的一些潜在特性的相对位置。排序通常是升序或降序，并且我们假设列表项具有已经定义的有意义的比较运算。许多有序列表操作与无序列表的操作相同。

- OrderedList() 创建一个新的空列表。它不需要参数，并返回一个空列表。
- add(item) 向列表中添加一个新项。它需要 item 作为参数，并不返回任何内容。假定该 item 不在列表中。
- remove(item) 从列表中删除该项。它需要 item 作为参数并修改列表。假设项存在于列表中。
- search(item) 搜索列表中的项目。它需要 item 作为参数，并返回一个布尔值。
- isEmpty() 检查列表是否为空。它不需要参数，并返回布尔值。

- size（）返回列表中的项数。它不需要参数，并返回一个整数。
- index(item) 返回项在列表中的位置。它需要 item 作为参数并返回索引。假定该项在列表中。
- pop() 删除并返回列表中的最后一个项。假设该列表至少有一个项。
- pop(pos) 删除并返回位置 pos 处的项。它需要 pos 作为参数并返回项。假定该项在列表中。

5. 实现有序列表

为了实现有序列表，我们必须记住项的相对位置是基于一些潜在的特性。上面给出的整数的有序列表 17, 26, 31, 54, 77 和 93 可以由 Figure 15 所示的链接结构表示。节点和链接结构表示项的相对位置。

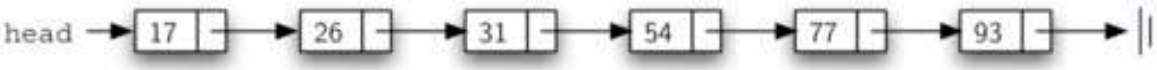

Figure 15

为了实现 OrderedList 类，我们将使用与前面看到的无序列表相同的技术。再次， head 的引用为 None 表示为空链表（参见 Listing 8）。

```
class OrderedList:
    def __init__(self):
        self.head = None
```

Listing 8

当我们考虑有序列表的操作时，我们应该注意， isEmpty 和 size 方法可以与无序列表一样实现，因为它们只处理链表中的节点数量，而不考虑实际项值。同样， remove 方法将正常工作，因为我们仍然需要找到该项，然后删除它。剩下的两个方法， search 和 add ，将需要一些修改。

搜索无序列表需要我们一次遍历一个节点，直到找到我们正在寻找的节点或者没找到节点（None）。事实证明，相同的方法在有序列表也有效。然而，在项不在链表中的情况下，我们可以利用该顺序来尽快停止搜索。

例如，Figure 16 展示了有序链表搜索值 45 。从链表的头部开始遍历，首先与 17 进行比较。由于 17 不是我们正在寻找的项，移动到下一个节点 26 。再次，这不是我们想要的，继续到 31 ，然后再到 54 。在这一点上，有一些不同。由于 54 不是我们正在寻找的项，我们以前的方法是继续向前迭代。然而，由于这是有序列表，一旦节点中的值变得大于我们正在搜索的项，搜索就可以停止并返回 False 。该项不可能存在于后面的链表中。

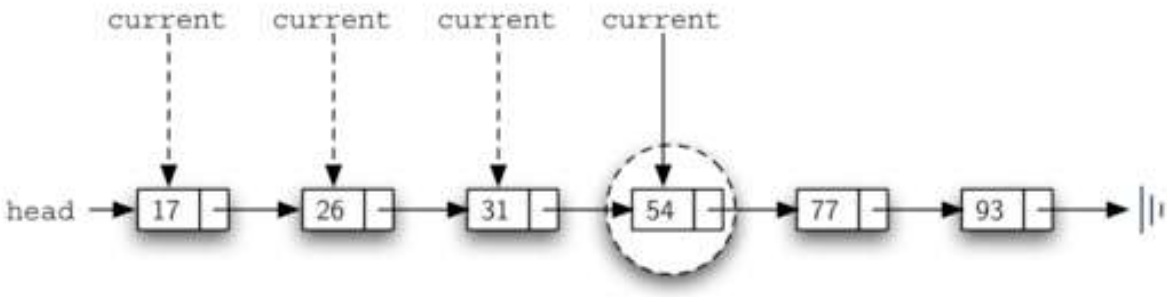

Figure 16

Listing 9 展示了完整的搜索方法。通过添加另一个布尔变量 `stop` 并将其初始化为 `False` （第4行），很容易合并上述新条件。 当 `stop` 是 `False` （不停止）时，我们可以继续在列表中前进（第5行）。如果发现任何节点包含大于我们正在寻找的项的数据，我们将 `stop` 设置为 `True` （第9-10行）。其余行与无序列表搜索相同。

```
def search(self,item):
    current = self.head
    found = False
    stop = False
    while current != None and not found and not stop:
        if current.getData() == item:
            found = True
        else:
            if current.getData() > item:
                stop = True
            else:
                current = current.getNext()

    return found
```

Listing 9

最重要的需要修改的方法是 `add` 。 回想一下，对于无序列表， `add` 方法可以简单地将新节点放置在链表的头部。 这是最简单的访问点。 不幸的是，这将不再适用于有序列表。需要在现有的有序列表中查找新项所属的特定位置。

假设我们有由 `17,26,54,77` 和 `93` 组成的有序列表，并且我们要添加值 `31` 。 `add` 方法必须确定新项属于 `26` 到 `54` 之间。Figure 17 展示了我们需要的设置。正如我们前面解释的，我们需要遍历链表，寻找添加新节点的地方。我们知道，当我们迭代完节点（ `current` 变为 None）或 `current` 节点的值变得大于我们希望添加的项时，我们就找到了该位置。在我们的例子中，看到值 `54` 我们停止迭代。

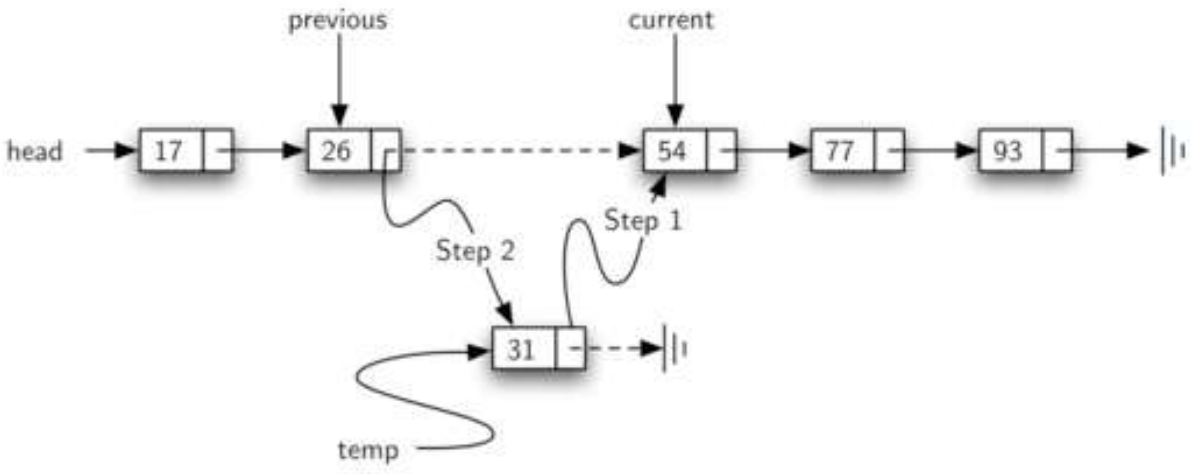

Figure 17

正如我们看到的无序列表，有必要有一个额外的引用，再次称为 previous ，因为 current 不会提供对修改的节点的访问。 Listing 10 展示了完整的 add 方法。 行 2-3 设置两个外部引用，行 9-10 允许 previous 每次通过迭代跟随 current 节点后面 。 条件（行5）允许迭代继续，只要有更多的节点，并且当前节点中的值不大于该项。 在任一种情况下，当迭代失败时，我们找到了新节点的位置。

该方法的其余部分完成 Figure17 所示的两步过程。一旦为该项创建了新节点，剩下的唯一问题是新节点是否将被添加在链表的开始处或某个中间位置。再次， previous == None （第13行）可以用来提供答案。

```
def add(self,item):
    current = self.head
    previous = None
    stop = False
    while current != None and not stop:
        if current.getData() > item:
            stop = True
        else:
            previous = current
            current = current.getNext()

    temp = Node(item)
    if previous == None:
        temp.setNext(self.head)
        self.head = temp
    else:
        temp.setNext(current)
        previous.setNext(temp)
```

Listing 10

链表分析

为了分析链表操作的复杂性，我们需要考虑它们是否需要遍历。考虑具有 n 个节点的链表。 `isEmpty` 方法是 $O(1)$，因为它需要一个步骤来检查头的引用为 `None` 。另一方面， `size` 将总是需要 n 个步骤，因为不从头到尾地移动没法知道有多少节点在链表中。因此，长度为 $O(n)$。将项添加到无序列表始终是 $O(1)$，因为我们只是将新节点放置在链表的头部。但是，搜索和删除，以及添加有序列表，都需要遍历过程。虽然平均他们可能只需要遍历节点的一半，这些方法都是 $O(n)$，因为在最坏的情况下，都将处理列表中的每个节点。

你可能还注意到此实现的性能与早前针对 Python 列表给出的实际性能不同。这表明链表不是 Python 列表的实现方式。 Python 列表的实际实现基于数组的概念。我们在第 8 章中更详细地讨论这个问题。

引用及参考：

[1]《Python数据结构与算法分析》布拉德利.米勒等著，人民邮电出版社，2019年9月.
[2]

课后练习

1. 写出栈、队列的完整Python代码或 C语言代码。

2. 写出列表和链表的完整Python代码或 C语言代码。

3. 举例说明基本数据结构。

讨论、思考题、作业：

参考资料（含参考书、文献等）： 算法导论. Thomas H. Cormen等，机械工业出版社，2017.

授课类型（请打√）： 理论课□ 讨论课□ 实验课□ 练习课□ 其他□

教学过程设计（请打√）： 复习□ 授新课□ 安排讨论□ 布置作业□

教学方式（请打√）： 讲授□ 讨论□ 示教□ 指导□ 其他□

教学资源（请打√）： 多媒体□ 模型□ 实物□ 挂图□ 音像□ 其他□

填表说明：1、每项页面大小可自行添减；2、教学内容与讨论、思考题、作业部分可合二为一。