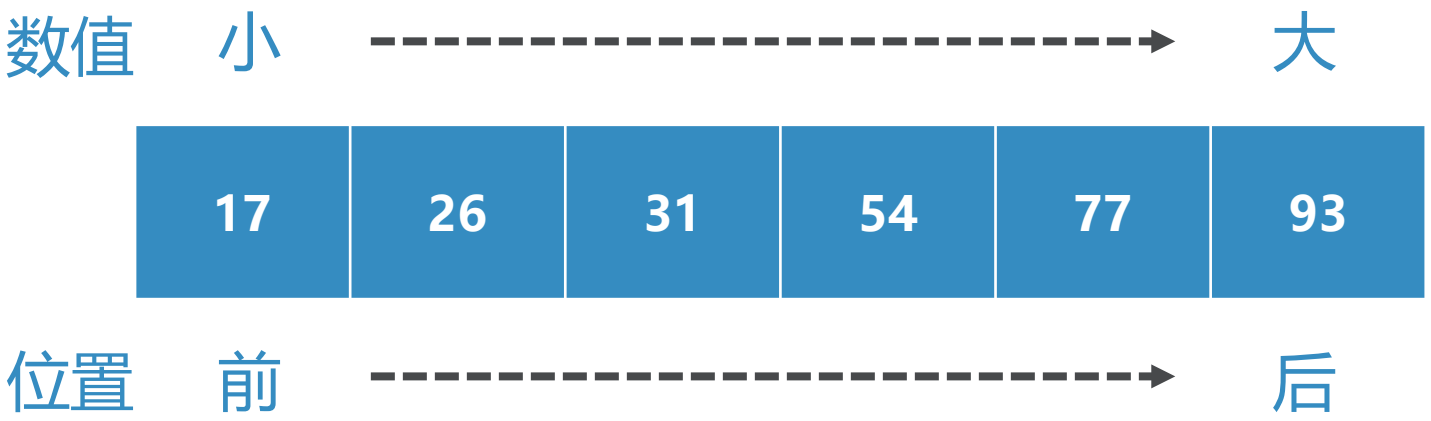
数据结构与算法（Python版）

有序表抽象数据类型及Python实现

陈斌 北京大学 gischen@pku.edu.cn

抽象数据类型：有序表OrderedList

- 有序表是一种数据项依照其某**可比性质**（如整数大小、字母表先后）来决定在列表中的**位置**
- 越“小”的数据项越靠近列表的头，越靠“前”



抽象数据类型：有序表OrderedList

- OrderedList所定义的操作如下：

  OrderedList(): 创建一个空的有序表

  add(item): 在表中添加一个数据项，并保持整体顺序，此项原不存在

  remove(item): 从有序表中移除一个数据项，此项应存在，有序表被修改

  search(item): 在有序表中查找数据项，返回是否存在

  isEmpty(): 是否空表

  size(): 返回表中数据项的个数

  index(item): 返回数据项在表中的位置，此项应存在

  pop(): 移除并返回有序表中最后一项，表中应至少存在一项

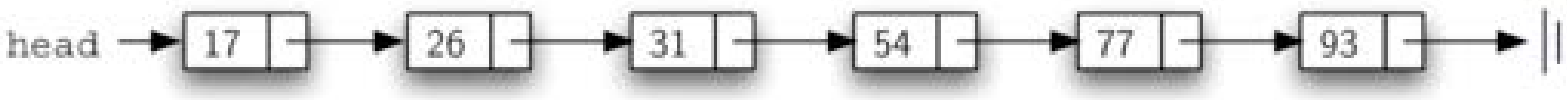
  pop(pos): 移除并返回有序表中指定位置的数据项，此位置应存在

有序表OrderedList实现

- 在实现有序表的时候，需要记住的是，数据项的相对位置，取决于它们之间的“大小”比较

  由于Python的扩展性，下面对数据项的讨论并不仅适用于整数，可适用于所有定义了__gt__方法（即'>'操作符）的数据类型

- 以整数数据项为例，(17, 26, 31, 54, 77, 93)的链表形式如图



有序表OrderedList实现

- 同样采用链表方法实现
- Node定义相同
- OrderedList也设置一个head来保存链表表头的引用

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

有序表OrderedList实现

- 对于isEmpty/size/remove这些方法，与节点的次序无关，所以其实现跟UnorderedList是一样的。
- search/add方法则需要有修改

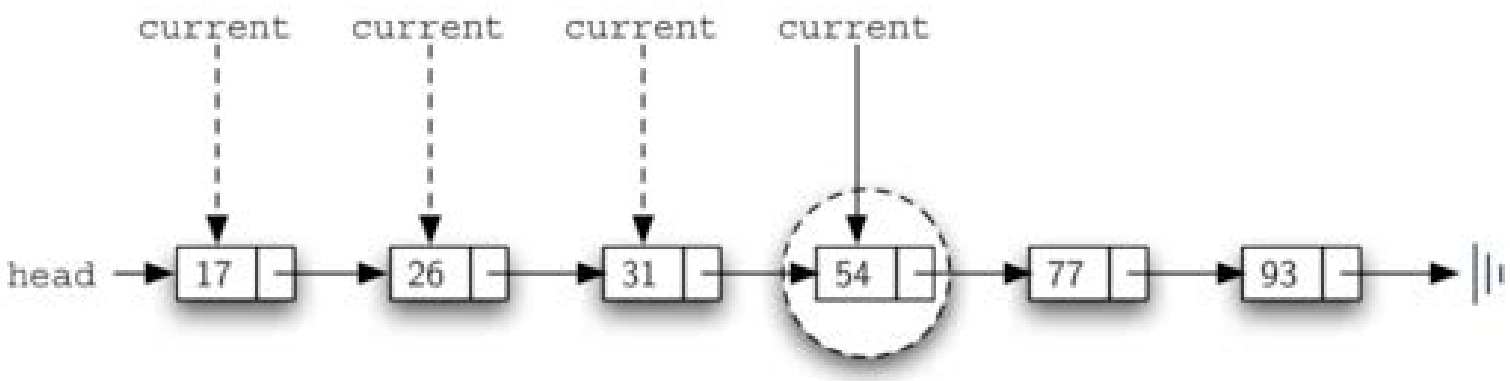
有序表实现：search方法

- 在**无序表**的search中，如果需要查找的数据项不存在，则会搜遍整个链表，直到表尾
- 对于**有序表**来说，可以利用链表节点有序排列的特性，来为search节省**不存在数据项**的查找时间

  一旦当前节点的数据项大于所要查找的数据项，则说明链表后面已经不可能再有要查找的数据项，可以直接返回False

有序表实现：search方法

- 如我们要在下图查找数据项45



有序表实现：search方法

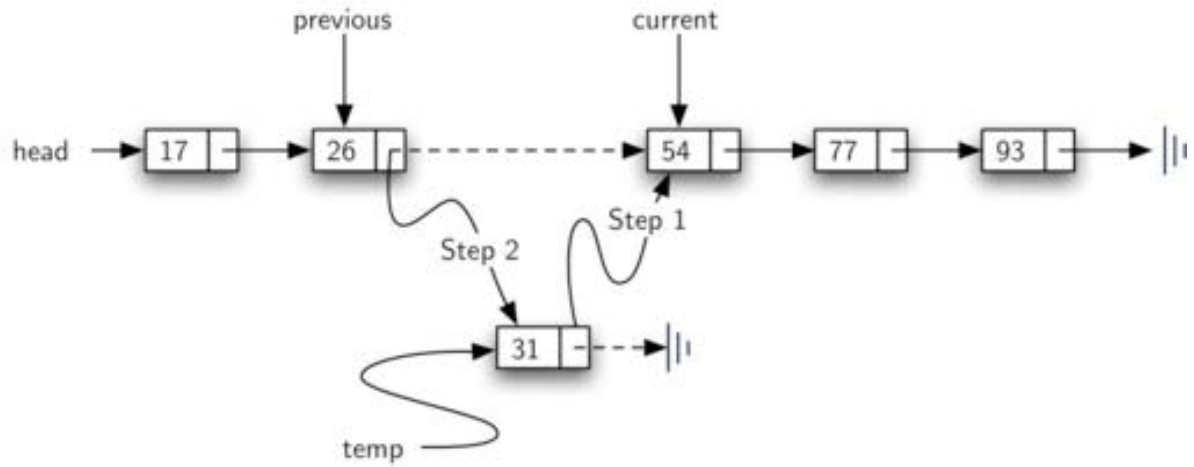

```
def search(self,item):
    current = self.head
    found = False
    stop = False
    while current != None and not found and not stop:
        if current.getData() == item:
            found = True
        else:
            if current.getData() > item:
                stop = True
            else: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

    return found
```

有序表实现：add方法

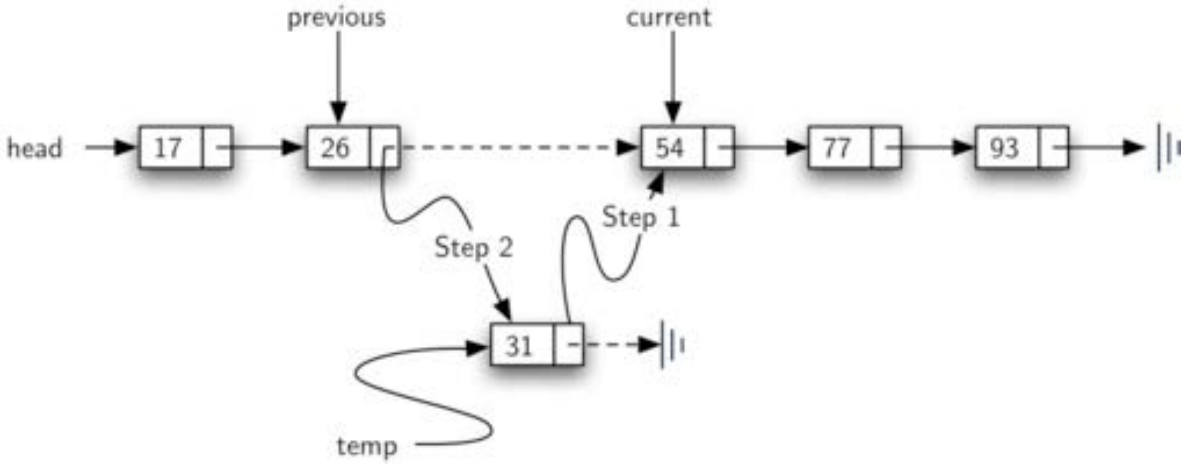
- 相比无序表，改变最大的方法是add，因为add方法必须**保证**加入的数据项添加在合适的位置，以**维护**整个链表的有序性

  比如在(17，26，54，77，93)的有序表中，加入数据项31，我们需要沿着链表，找到第一个比31大的数据项54，将31插入到54的前面



有序表实现：add方法

- 由于涉及到的插入位置是当前节点**之前**，而链表无法得到“前驱”节点的引用
- 所以要跟remove方法类似，引入一个previous的引用，跟随当前节点current
- 一旦找到首个比31大的数据项，previous就派上用场了



有序表OrderedList实现：add方法

发现插入位置

插入在表头

插入在表中

```
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)
```

链表实现的算法分析

- 对于链表复杂度的分析，主要是看相应的方法是否涉及到链表的**遍历**
- 对于一个包含节点数为n的链表

  isEmpty是O(1)，因为仅需要检查head是否为None

  size是O(n)，因为除了遍历到表尾，没有其它办法得知节点的数量

  search/remove以及有序表的add方法，则是O(n)，因为涉及到链表的遍历，按照概率其平均操作的次数是n/2

  无序表的add方法是O(1)，因为仅需要插入到表头

链表实现的算法分析

- 链表实现的List，跟Python内置的列表数据类型，在有些相同方法的实现上的时间复杂度不同
- 主要是因为Python内置的列表数据类型是基于**顺序存储**来实现的，并进行了优化。