

1 基本数据结构

1.1 栈

1.1.1 何谓栈

1.1.2 栈抽象数据类型

1.1.3 用Python实现栈

1.1.4 匹配括号

1.1.5 普通情况：匹配符号

1.1.6 将十进制数转换成二进制数

1.1.7 前序、中序和后序表达式

1.1.7.1 从中序到后序的通用转换法

1.1.7.2 计算后序表达式

1.2 队列

1.2.1 何谓队列

1.2.2 队列抽象数据类型

1.2.3 用Python实现队列

1.2.4 模拟：传土豆

1.3 双端队列

1.3.1 何谓双端队列

1.3.2 双端队列抽象数据类型

1.3.3 用Python实现双端队列

1.3.4 回文检测器

1.4 列表

1.4.1 无序列表抽象数据类型

1.4.2 实现无序列表：链表

1.4.3 有序列表抽象数据类型

1.4.4 实现有序列表

1.4.5 链表分析

1 基本数据结构

- 4种简单而强大的数据结构：栈、队列、双端队列和列表。它们都是有序的数据集合，其元素的顺序取决于添加顺序或移除顺序。
- **线性数据结构**：一旦某个元素被添加进来，它与前后元素的相对位置将保持不变。

1.1 栈

1.1.1 何谓栈

- 栈是有序集合，添加操作和移除操作总发生在同一端，即“顶端”，另一端则被称为“底端”。
- 最新添加的元素将被最先移除，这种排序原则被称作**LIFO**（last-in first-out），即后进先出。

1.1.2 栈抽象数据类型

1. `Stack()` 创建一个空栈。它不需要参数，且会返回一个空栈。
2. `push(item)` 将一个元素添加到栈的顶端。它需要一个参数 `item`，且无返回值。
3. `pop()` 将栈顶端的元素移除。它不需要参数，但会返回顶端的元素，并且修改栈的内容。
4. `peek()` 返回栈顶端的元素，但是并不移除该元素。它不需要参数，也不会修改栈的内容。
5. `isEmpty()` 检查栈是否为空。它不需要参数，且会返回一个布尔值。
6. `size()` 返回栈中元素的数目。它不需要参数，且会返回一个整数。

1.1.3 用Python实现栈

```
1 class Stack:
2     def __init__(self):
3         self.items = []
4
5     def isEmpty(self):
6         return self.items == []
7
8     def push(self, item):
9         return self.items.append(item)
10
11    def pop(self):
12        return self.items.pop()
13
14    def peek(self):
15        return self.items[-1]
16
17    def size(self):
18        return len(self.items)
```

```
1 >>> s = Stack() # 栈内容: []
2 >>> s.isEmpty() # 栈内容: []
3 True
4 >>> s.push(4) # 栈内容: [4]
5 >>> s.push('dog') # 栈内容: [4, 'dog']
6 >>> s.peek() # 栈内容: [4, 'dog']
7 'dog'
8 >>> s.push(True) # 栈内容: [4, 'dog', True]
9 >>> s.size() # 栈内容: [4, 'dog', True]
10 3
11 >>> s.isEmpty() # 栈内容: [4, 'dog', True]
12 False
13 >>> s.push(8.4) # 栈内容: [4, 'dog', True, 8.4]
14 >>> s.pop() # 栈内容: [4, 'dog', True]
15 8.4
16 >>> s.pop() # 栈内容: [4, 'dog']
17 True
18 >>> s.size() # 栈内容: [4, 'dog']
19 2
```

1.1.4 匹配括号

- 由一个空栈开始，从左往右依次处理括号。
- 如果遇到左括号，便通过push操作将其加入栈中，以此表示稍后需要有一个与之匹配的右括号。
- 反之，如果遇到右括号，就调用pop操作。
- 只要栈中的所有左括号都能遇到与之匹配的右括号，那么整个括号串就是匹配的；如果栈中有任何一个左括号找不到与之匹配的右括号，则括号串就是不匹配的。
- 在处理完匹配的括号串之后，栈应该是空的。

```
1 def parChecker(symbolString):
2     s = Stack()
3     balanced = True
4     index = 0
5     while index < len(symbolString) and balanced:
6         symbol = symbolString[index]
7         if symbol == '(':
8             s.push(symbol)
9         else:
10            if s.isEmpty():
11                balanced = False
12            else:
13                s.pop()
14
15        index += 1
16
17    if balanced and s.isEmpty():
18        return True
19    else:
20        return False
```

```
1 >>> parChecker('(()())')
2 True
3 >>> parChecker('((((())))')
4 True
5 >>> parChecker('((((())())')
6 True
7 >>> parChecker('((((((((()'))
8 False
9 >>> parChecker('()'))')
10 False
11 >>> parChecker('(()())()')
12 False
```

1.1.5 普通情况：匹配符号

- 要处理新类型的符号，可以轻松扩展括号匹配检测器。
- 每一个左符号都将被压入栈中，以待之后出现对应的右括号。
- 唯一的区别在于，当出现右符号时，必须检测其类型是否与栈顶的左符号类型相匹配。
- 如果两个符号不匹配，那么整个符号串也就不匹配。
- 同样，如果整个符号串处理完成并且栈是空的，那么就说明所有符号正确匹配。

```
1 def parChecker(symbolString):
2     s = Stack()
3
4     balanced = True
5     index = 0
6
7     while index < len(symbolString) and balanced:
8         symbol = symbolString[index]
9         if symbol in '([{':
10            s.push(symbol)
11        else:
12            if s.isEmpty():
13                balanced = False
14            else:
15                top = s.pop()
16                if not matches(top, symbol):
17                    balanced = False
18
19            index += 1
20
21    if balanced and s.isEmpty():
22        return True
23    else:
24        return False
25
26 def matches(left, right):
27     lefts = '(['{'
28     rights = ')]}'
29
30     return lefts.index(left) == rights.index(right)
```

```

1  >>> parChecker('{{([[])}())'})
2  True
3  >>> parChecker('[[{{{(())}}}]')
4  True
5  >>> parChecker('[] [] () {}')
6  True
7  >>> parChecker('([)]')
8  False
9  >>> parChecker('((([])))')
10 False
11 >>> parChecker('[{()}]')
12 False

```

1.1.6 将十进制数转换成二进制数

1. $233_{10} = 2 \times 10^2 + 3 \times 10^1 + 3 \times 10^0$
2. $11101001_2 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

- “除以2”算法用一个简单的循环不停地将十进制数除以2，并且记录余数。
- 第一次除以2的结果能够用于区分奇数和偶数。
- 如果是偶数，则余数为0，因此个位上的数字为0；如果是奇数，则余数为1，因此个位上的数字为1。
- 可以将要构建的二进制数看成一系列数字；计算出的第一个余数是最后一位。

```

1  def divideBy2(decNumber):
2      remstack = Stack()
3
4      while decNumber > 0:
5          rem = decNumber % 2
6          remstack.push(rem)
7          decNumber = decNumber // 2
8
9      binString = ''
10     while not remstack.isEmpty():
11         binString = binString + str(remstack.pop())
12
13     return binString

```

```

1  >>> divideBy2(233)
2  '11101001'

```

- 可以将divideBy2函数修改成接受一个十进制数以及希望转换的进制基数，“除以2”则变成“除以基数”。
- 当基数超过10时，不能再直接使用余数，这是因为余数本身就是两位的十进制数。
- 因此，需要创建一套数字来标示大于9的余数。

```

1 def baseConverter(decNumber, base):
2     digits = '0123456789ABCDEF'
3
4     remstack = Stack()
5
6     while decNumber > 0:
7         rem = decNumber % base
8         remstack.push(rem)
9         decNumber = decNumber // base
10
11     newString = ''
12     while not remstack.isEmpty():
13         newString = newString + digits[remstack.pop()]
14
15     return newString

```

```

1 >>> baseConverter(233, 8)
2 '351'
3 >>> baseConverter(233, 16)
4 'E9'

```

1.1.7 前序、中序和后序表达式

- 前序和后序表达式不需要括号，这两种表达式中的运算符所对应的操作数是明确的。
- 只有中序表达式需要额外的符号来消除歧义。
- 若要将任意复杂的中序表达式转换成前序表达式或后序表达式，可以先将其写作完全括号表达式，然后将括号内的运算符移到左括号处（前序表达式）或者右括号处（后序表达式）。

1.1.7.1 从中序到后序的通用转换法

当遇到左括号时，需要将其保存，以表示接下来会遇到高优先级的运算符；那个运算符需要等到对应的右括号出现才能确定其位置；当右括号出现时，便可以将运算符从栈中取出来。

1. 创建用于保存运算符的空栈 `opStack`，以及一个用于保存结果的空列表。
2. 使用字符串方法 `split` 将输入的中序表达式转换成一个列表。
3. 从左往右扫描这个标记列表。
 - 如果标记是操作数，将其添加到结果列表的末尾。
 - 如果标记是左括号，将其压入 `opStack` 栈中。
 - 如果标记是右括号，反复从 `opStack` 栈中移除元素，直到移除对应的左括号。将从栈中取出的每一个运算符都添加到结果列表的末尾。
 - 如果标记是运算符，将其压入 `opStack` 栈中。但是，在这之前，需要先从栈中取出优先级更高或相同的运算符，并将它们添加到结果列表的末尾。
4. 当处理完输入表达式以后，检查 `opStack`。将其中所有残留的运算符全部添加到结果列表的末尾。

```

1 import string
2

```

```

3  def infixToPostfix(infixexpr):
4      prec = {}
5      prec['*'] = 3
6      prec['/'] = 3
7      prec['+'] = 2
8      prec['-'] = 2
9      prec['('] = 1
10
11     opStack = Stack()
12     postfixList = []
13
14     tokenList = infixexpr.split()
15
16     for token in tokenList:
17         if token in string.ascii_uppercase:
18             postfixList.append(token)
19         elif token == '(':
20             opStack.push(token)
21         elif token == ')':
22             topToken = opStack.pop()
23             while topToken != '(':
24                 postfixList.append(topToken)
25                 topToken = opStack.pop()
26         else:
27             while (not opStack.isEmpty()) and (prec[opStack.peek()] >=
prec[token]):
28                 postfixList.append(opStack.pop())
29                 opStack.push(token)
30
31     while not opStack.isEmpty():
32         postfixList.append(opStack.pop())
33
34     return ' '.join(postfixList)

```

```

1  >>> infixToPostfix('( A + B ) * ( C + D )')
2  'A B + C D + *'
3  >>> infixToPostfix('( A + B ) * C')
4  'A B + C *'
5  >>> infixToPostfix('A + B * C')
6  'A B C * +'

```

1.1.7.2 计算后序表达式

当遇到一个运算符时，需要用离它最近的两个操作数来计算。

1. 创建空栈operandStack。
2. 使用字符串方法split将输入的后序表达式转换成一个列表。
3. 从左往右扫描这个标记列表。

- 如果标记是操作符，将其转换成整数并且压入operandStack栈中。
 - 如果标记是运算符，从operandStack栈中取出两个操作数。第一次取出右操作数，第二次取出左操作数。进行相应的算术运算，然后将运算结果压入operandStack栈中。
4. 当处理完输入表达式时，栈中的值就是结果。将其从栈中返回。

```
1 def postfixEval(postfixExpr):
2     operandStack = Stack()
3
4     tokenList = postfixExpr.split()
5
6     for token in tokenList:
7         if token in '0123456789':
8             operandStack.push(token)
9         else:
10            operand2 = operandStack.pop()
11            operand1 = operandStack.pop()
12            result = doMath(token, int(operand1), int(operand2))
13            operandStack.push(result)
14
15    return operandStack.pop()
16
17 def doMath(op, op1, op2):
18     if op == '*':
19         return op1 * op2
20     elif op == '/':
21         return op1 / op2
22     elif op == '+':
23         return op1 + op2
24     else:
25         return op1 - op2
```

```
1 >>> postfixEval('4 5 6 * +')
2 34
3 >>> postfixEval('7 8 + 3 2 + /')
4 3
```

1.2 队列

1.2.1 何谓队列

- 队列是有序集合，添加操作发生在“尾部”，移除操作则发生在“头部”。
- 最新添加的元素必须在队列的尾部等待，在队列中时间最长的元素则排在最前面，这种排序原则被称作**FIFO**（first-in first-out），即先进先出。

1.2.2 队列抽象数据类型

1. `Queue()` 创建一个空队列。它不需要参数，且会返回一个空队列。
2. `enqueue(item)` 在队列的尾部添加一个元素。它需要一个元素作为参数，不返回任何值。
3. `dequeue()` 从队列的头部移除一个元素。它不需要参数，且会返回一个元素，并修改队列的内容。
4. `isEmpty()` 检查队列是否为空。它不需要参数，且会返回一个布尔值。
5. `size()` 返回队列中元素的数目。它不需要参数，且会返回一个整数。

1.2.3 用Python实现队列

```
1 class Queue:
2     def __init__(self):
3         self.items = []
4
5     def isEmpty(self):
6         return self.items == []
7
8     def enqueue(self, item):
9         return self.items.insert(0, item)
10
11    def dequeue(self):
12        return self.items.pop()
13
14    def size(self):
15        return len(self.items)
```

```
1 >>> q = Queue() # 队列内容：[]
2 >>> q.isEmpty() # 队列内容：[]
3 True
4 >>> q.enqueue(4) # 队列内容：[4]
5 >>> q.enqueue('dog') # 队列内容：['dog', 4]
6 >>> q.enqueue(True) # 队列内容：[True, 'dog', 4]
7 >>> q.size() # 队列内容：[True, 'dog', 4]
8 3
9 >>> q.isEmpty() # 队列内容：[True, 'dog', 4]
10 False
11 >>> q.enqueue(8.4) # 队列内容：[8.4, True, 'dog', 4]
12 >>> q.dequeue() # 队列内容：[8.4, True, 'dog']
13 4
14 >>> q.dequeue() # 队列内容：[8.4, True]
15 'dog'
16 >>> q.size() # 队列内容：[8.4, True]
17 2
```

1.2.4 模拟：传土豆

在这个游戏中，孩子们围成一圈，并依次尽可能快地传递一个土豆。在某个时刻，大家停止传递，此时手里有土豆的孩子就得退出游戏。重复上述过程，直到只剩下一个孩子。

- 该程序接受一个名字列表和一个用于计数的常量num，并且返回最后一人的名字。
- 使用队列来模拟一个环，假设握着土豆的孩子位于队列的头部。
- 在模拟传土豆的过程中，程序将这个孩子的名字移出对垒，然后立刻将其插入队列的尾部。
- 随后，这个孩子会一直等待，直到再次到达队列的头部。
- 在出列和入列num次之后，此时位于队列头部的孩子出局，新一轮游戏开始。
- 如此反复，直到队列中只剩下一个名字（队列的大小为1）。

```
1 def hotPotato(namelist, num):
2     simqueue = Queue()
3     for name in namelist:
4         simqueue.enqueue(name)
5
6     while simqueue.size() > 1:
7         for i in range(num):
8             simqueue.enqueue(simqueue.dequeue())
9
10        simqueue.dequeue()
11
12    return simqueue.dequeue()
```

```
1 >>> hotPotato(['Bill', 'David', 'Susan', 'Jane', 'Kent', 'Brad'], 7)
2 'Susan'
```

1.3 双端队列

1.3.1 何谓双端队列

- 双端队列是与队列类似的有序集合。
- 与队列不同的是，双端队列对在哪一端添加和移除元素没有任何限制。
- 新元素既可以被添加到前端，也可以被添加到后端。
- 同理，已有的元素也能从任意一端移除。

1.3.2 双端队列抽象数据类型

1. `Deque()` 创建一个空的双端队列。它不需要参数，且会返回一个空的双端队列。
2. `addFront(item)` 将一个元素添加到双端队列的前端。它接受一个元素作为参数，没有返回值。
3. `addRear(item)` 将一个元素添加到双端队列的后端。它接受一个元素作为参数，没有返回值。
4. `removeFront()` 从双端队列的前端移除一个元素。它不需要参数，且会返回一个元素，并修改双端队列的内容。
5. `removeRear()` 从双端队列的后端移除一个元素。它不需要参数，且会返回一个元素，并修改双端队列的内容。

6. `isEmpty()`检查双端队列是否为空。它不需要参数，且会返回一个布尔值。
7. `size()`返回双端队列中元素的数目。它不需要参数，且会返回一个整数。

1.3.3 用Python实现双端队列

```
1 class Deque:
2     def __init__(self):
3         self.items = []
4
5     def isEmpty(self):
6         return self.items == []
7
8     def addFront(self, item):
9         self.items.append(item)
10
11    def addRear(self, item):
12        return self.items.insert(0, item)
13
14    def removeFront(self):
15        return self.items.pop()
16
17    def removeRear(self):
18        return self.items.pop(0)
19
20    def size(self):
21        return len(self.items)
```

```
1 >>> d = Deque() # 双端队列内容：[]
2 >>> d.isEmpty() # 双端队列内容：[]
3 True
4 >>> d.addRear(4) # 双端队列内容：[4]
5 >>> d.addRear('dog') # 双端队列内容：['dog', 4]
6 >>> d.addFront('cat') # 双端队列内容：['dog', 4, 'cat']
7 >>> d.addFront(True) # 双端队列内容：['dog', 4, 'cat', True]
8 >>> d.size() # 双端队列内容：['dog', 4, 'cat', True]
9 4
10 >>> d.isEmpty() # 双端队列内容：['dog', 4, 'cat', True]
11 False
12 >>> d.addRear(8.4) # 双端队列内容：[8.4, 'dog', 4, 'cat', True]
13 >>> d.removeRear() # 双端队列内容：['dog', 4, 'cat', True]
14 8.4
15 >>> d.removeFront() # 双端队列内容：['dog', 4, 'cat']
16 True
```

1.3.4 回文检测器

- 回文是指从前往后读和从后往前读都一样的字符串，例如radar、toot和madam。
- 按从左往右的顺序将字符串中的字符添加到双端队列的后端。
- 利用双端队列的双重性，其前端是字符串的第一个字符，后端是字符串的最后一个字符。
- 由可以从前后两端移除元素，因此我们能够比较两个元素，并且只有在二者相等时才继续。
- 如果一直匹配第一个和最后一个元素，最终会处理完所有的字符（如果字符数是偶数），或者剩下只有一个元素的双端队列（如果字符数是奇数）。
- 任何一种结果都表明输入字符串是回文。

```
1 def palchecker(aString):
2     chardeque = Deque()
3
4     for ch in aString:
5         chardeque.addRear(ch)
6
7     stillEqual = True
8
9     while chardeque.size() > 1 and stillEqual:
10         first = chardeque.removeFront()
11         last = chardeque.removeRear()
12         if first != last:
13             stillEqual = False
14
15     return stillEqual
```

```
1 >>> palchecker('lsdkjfskf')
2 False
3 >>> palchecker('toot')
4 True
```

1.4 列表

- 列表是元素的集合，其中每一个元素都有一个相对于其他元素的位置。
- 为简单起见，我们假设列表中没有重复元素。

1.4.1 无序列表抽象数据类型

1. `List()` 创建一个空列表。它不需要参数，且会返回一个空列表。
2. `add(item)` 假设元素 `item` 之前不在列表中，并向其中添加 `item`。它接受一个元素作为参数，无返回值。
3. `remove(item)` 假设元素 `item` 已经在列表中，并从其中移除 `item`。它接受一个元素作为参数，并且修改列表。
4. `search(item)` 在列表中搜索元素 `item`。它接受一个元素作为参数，并且返回布尔值。
5. `isEmpty()` 检查列表是否为空。它不需要参数，并且返回布尔值。
6. `length()` 返回列表中元素的个数。它不需要参数，并且返回一个整数。

7. `append(item)` 假设元素 `item` 之前不在列表中，并在列表的最后位置添加 `item`。它接受一个元素作为参数，无返回值。
8. `index(item)` 假设元素 `item` 已经在列表中，并返回该元素在列表中的位置。它接受一个元素作为参数，并且返回该元素的下标。
9. `insert(pos, item)` 假设元素 `item` 之前不在列表中，同时假设 `pos` 是合理的值，并在位置 `pos` 处添加元素 `item`。它接受两个参数，无返回值。
10. `pop()` 假设列表不为空，并移除列表中的最后一个元素。它不需要参数，且会返回一个元素。
11. `pop(pos)` 假设在指定位置 `pos` 存在元素，并移除该位置上的元素。它接受位置参数，且会返回一个元素。

1.4.2 实现无序列表：链表

- 无序列表需要维持元素之间的相对位置，但是并不需要再连续的内存空间中维护这些位置信息。
- 必须指明列表中第一个元素的位置。
- 一旦知道第一个元素的位置，就能根据其中的链接信息访问第二个元素，接着访问第三个元素，依次类推。
- 指向链表第一个元素的引用被称作头。
- 最后一个元素需要知道自己没有下一个元素。

1. Node类

- **节点 (node)** 是构建链表的基本数据结构。
- 每一个节点对象都必须持有至少两份信息：首先，节点必须包含列表元素，我们称之为节点的数据变量；其次，节点必须保存指向下一个节点的引用。

```
1 class Node:
2     def __init__(self, initdata):
3         self.data = initdata
4         self.next = None
5
6     def getData(self):
7         return self.data
8
9     def getNext(self):
10        return self.next
11
12    def setData(self, newdata):
13        self.data = newdata
14
15    def setNext(self, newnext):
16        self.next = newnext
```

```
1 >>> temp = Node(93)
2 >>> temp.getData()
3 93
```

2. UnorderedList类

- 每一个列表对象都保存了指向列表头部的引用。

```
1 class UnorderedList:
2     def __init__(self):
3         self.head = None
```

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

- isEmpty方法检查列表的头部是否为指向None的引用。

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

```
1 >>> mylist.isEmpty()
2 True
```

- 为了将元素添加到类表中，需要实现add方法。
- 由于本例中的列表是无序的，因此新元素相对于已有元素的位置并不重要。
- 添加新节点最简便的位置就是头部，或者说链表的起点。
- 我们把新元素作为列表的第一个元素，并且把已有的元素链接到该元素的后面。

```
1 def add(self, item):
2     temp = Node(item)
3     temp.setNext(self.head) # 将新节点的next引用指向当前列表中的第一个节点
4     self.head = temp # 修改列表的头节点，使其指向新创建的节点
```

```
1 >>> mylist.add(31)
2 >>> mylist.add(77)
3 >>> mylist.add(17)
4 >>> mylist.add(93)
5 >>> mylist.add(26)
6 >>> mylist.add(54)
```

- 接下来要实现的方法（length、search和remove）都基于链表遍历技术。
- 遍历是指系统地访问每一个节点，具体做法是用一个外部引用从列表的头节点开始访问。
- 随着访问每一个节点，我们将这个外部引用通过“遍历”下一个引用来指向下一个节点。

```

1 def length(self):
2     current = self.head
3     count = 0
4     while current != None:
5         count += 1
6         current = current.getNext()
7
8     return count

```

```

1 >>> mylist.isEmpty()
2 False
3 >>> mylist.length()
4 6

```

- 在无序列表中搜索一个值同样也会用到遍历技术。
- 事实上，如果遍历到列表的末尾，就意味着要找的元素不在列表中。
- 如果在遍历过程中找到所需的元素，就没有必要继续遍历了。

```

1 def search(self, item):
2     current = self.head
3     found = False
4     while current != None and not found:
5         if current.getData() == item:
6             found = True
7         else:
8             current = current.getNext()
9
10    return found

```

```

1 >>> mylist.search(17)
2 True
3 >>> mylist.search(66)
4 False

```

- 为了将包含元素的节点移除，需要将其前面的节点中的`next`引用指向`current`之后的节点。
- 然而，并没有反向遍历链表的方法。
- 这一困境的解决方法就是在遍历链表时使用两个外部引用。
- `current`与之前一样，标记在链表中的当前位置；新的引用`previous`总是指向`current`上一次访问的节点。

```

1 def remove(self, item):
2     current = self.head
3     previous = None

```

```

4         found = False
5         while not found:
6             if current.getData() == item:
7                 found = True
8             else:
9                 previous = current
10                current = current.getNext()
11
12        if previous == None:
13            self.head = current.getNext()
14        else:
15            previous.setNext(current.getNext())

```

```

1 >>> mylist.remove(26)
2 >>> mylist.length()
3 5

```

1.4.3 有序列表抽象数据类型

1. `OrderedList()` 创建一个空有序列表。它不需要参数，且会返回一个空列表。
2. `add(item)` 假设 `item` 之前不在列表中，并向其中添加 `item`，同时保持整个列表的顺序。它接受一个元素作为参数，无返回值。
3. `remove(item)` 假设 `item` 已经在列表中，并从其中移除 `item`。它接受一个元素作为参数，并且修改列表。
4. `search(item)` 在列表中搜索 `item`。它接受一个元素作为参数，并且返回布尔值。
5. `isEmpty()` 检查列表是否为空。它不需要参数，并且返回布尔值。
6. `length()` 返回列表中元素的个数。它不需要参数，并且返回一个整数。
7. `index(item)` 假设 `item` 已经在列表中，并返回该元素在列表中的位置。它接受一个元素作为参数，并且返回该元素的下标。
8. `pop()` 假设列表不为空，并移除列表中的最后一个元素。它不需要参数，且会返回一个元素。
9. `pop(pos)` 假设在指定位置 `pos` 存在元素，并移除该位置上的元素。它接受位置参数，且会返回一个元素。

1.4.4 实现有序列表

- 在实现有序列表时必须记住，元素的相对位置取决于它们的基本特征。
- `OrderedList` 类的构造方法与 `UnorderedList` 类的相同。

```

1 class OrderedList:
2     def __init__(self):
3         self.head = None

```

- 因为 `isEmpty` 和 `length` 仅与列表中的节点数目有关，而与实际的元素值无关，所以这两个方法在有序列表的实现与在无序列表中一样。

- 同理，由于仍然需要找到目标元素并且通过更改链接来移除节点，因此remove方法的实现也一样。
- 剩下的两个方法，search和add需要做一些修改。
- 如果目标元素不在列表中，可以利用元素有序排列这一特性尽早终止搜索。

```
1  def search(self, item):
2      current = self.head
3      found = False
4      stop = False
5      while current != None and not found and not stop:
6          if current.getData() == item:
7              found = True
8          else:
9              if current.getData() > item:
10                 stop = True
11             else:
12                 current = current.getNext()
13
14     return found
```

- 需要修改最多的是add方法，我们需要再已有链表中为新节点找到正确的插入位置。
- 当访问完所有节点（current是None）或者当前值大于要添加的元素时，就找到了插入位置。

```
1  def add(self, item):
2      current = self.head
3      previous = None
4      stop = False
5      while current != None and not stop:
6          if current.getData() > item:
7              stop = True
8          else:
9              previous = current
10             current = current.getNext()
11
12     temp = Node(item)
13     if previous == None:
14         temp.setNext(self.head)
15         self.head = temp
16     else:
17         temp.setNext(current)
18         previous.setNext = temp
```

1.4.5 链表分析

- 在分析链表操作的时间复杂度时，考虑其是否需要遍历列表。
- `isEmpty`方法的时间复杂度是 $O(1)$ 。
- `length`方法的时间复杂度是 $O(n)$ 。
- 无序列表的`add`方法的时间复杂度是 $O(1)$ 。
- 有序列表的`search`、`remove`和`add`方法的时间复杂度是 $O(n)$ 。