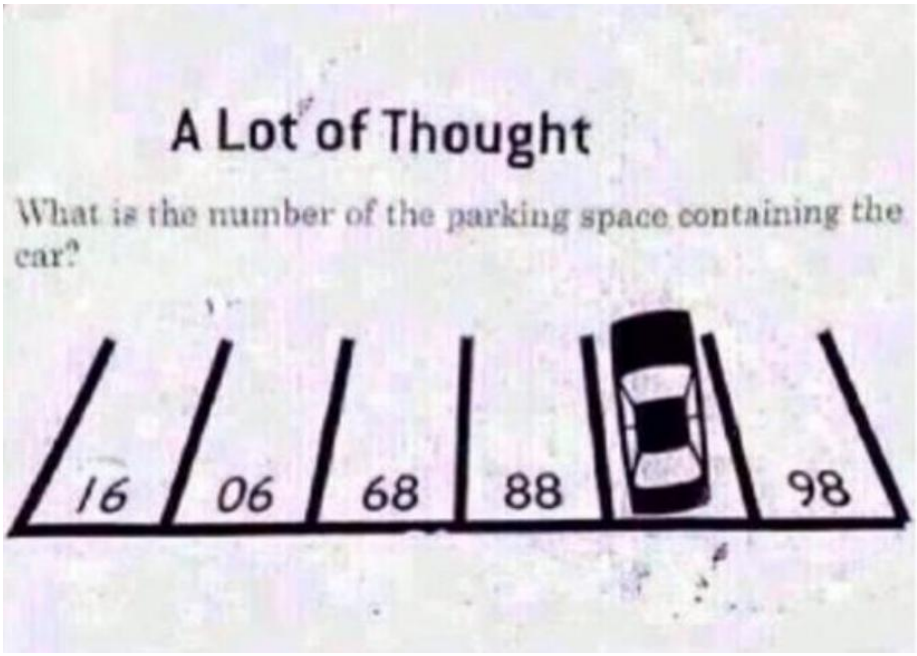
一、算法概述

1.什么是算法

思考1:1+2+...+1000=?

思考2：停车的是几号车位?



先来看下面一道题: 如果 a+b+c=1000，且 $a^2+b^2=c^2$（a,b,c 为自然数），如何求出所有a、b、c可能的组合?**Low B的方案:**

```
import time
start_time = time.time()
# 注意是三重循环
for a in range(0, 1001):
    for b in range(0, 1001):
        for c in range(0, 1001):
            if a**2 + b**2 == c**2 and a+b+c == 1000:
                print("a, b, c: %d, %d, %d" % (a, b, c))
end_time = time.time()
print("elapsed: %f" % (end_time - start_time))
print("complete!")
```

运行结果:

```
a, b, c: 0, 500, 500
a, b, c: 200, 375, 425
a, b, c: 375, 200, 425
a, b, c: 500, 0, 500
elapsed: 214.583347
complete!
```

运行时间是214秒

1）算法的概念

- 算法是计算机处理信息的本质，因为计算机程序本质上是一个算法来告诉计算机确切的步骤来执行一个指定的任务。一般地，当算法在处理信息时，会从输入设备或数据的存储地址读取数据，把结果写入输出设备或某个存储地址供以后再调用。

  算法是独立存在的一种解决问题的方法和思想。

  一个计算过程（算），解决问题的方法（法）

对于算法而言，实现的语言并不重要，重要的是思想。

算法可以有不同的语言描述实现版本（如C描述、C++描述、Python描述等），我们现在是在用Python语言进行描述实现。

2）算法的五大特性

- **输入**: 算法具有0个或多个输入
- **输出**: 算法至少有1个或多个输出
- **有穷性**: 算法在有限的步骤之后会自动结束而不会无限循环，并且每一个步骤可以在可接受的时间内完成
- **确定性**：算法中的每一步都有确定的含义，不会出现二义性
- **可行性**：算法的每一步都是可行的，也就是说每一步都能够执行有限的次数完成

No BB的方案:

```
import time

start_time = time.time()

# 注意是两重循环
for a in range(0, 1001):
    for b in range(0, 1001-a):
        c = 1000 - a - b
        if a**2 + b**2 == c**2:
            print("a, b, c: %d, %d, %d" % (a, b, c))

end_time = time.time()
print("elapsed: %f" % (end_time - start_time))
print("complete!")
```

运行结果:

```
a, b, c: 0, 500, 500
a, b, c: 200, 375, 425
a, b, c: 375, 200, 425
a, b, c: 500, 0, 500
elapsed: 0.686474
```

2. 算法的代价及其度量

1）执行时间反应算法效率

对于同一问题，我们给出了两种解决算法，在两种算法的实现中，我们对程序执行的时间进行了测算，发现两段程序执行的时间相差悬殊（214.583347秒相比于0.686474秒），由此我们可以得出结论：**实现算法程序的执行时间可以反应出算法的效率，即算法的优劣。**

2）单靠时间值绝对可信吗?

假设我们将第二次尝试的算法程序运行在一台配置古老性能低下的计算机中，情况会如何？很可能运行的时间并不会比在我们的电脑中运行算法一的214.583347秒快多少。 **单纯依靠运行的时间来比较算法的优劣并不一定是客观准确的!** 程序的运行离不开计算机环境（包括硬件和操作系统），这些客观原因会影响程序运行的速度并反应在程序的执行时间上。那么如何才能客观的评判一个算法的优劣呢?

3）时间复杂度与“大O记法”

我们假定计算机执行算法每一个基本操作的时间是固定的一个时间单位，那么有多少个基本操作就代表会花费多少时间单位。显然对于不同的机器环境而言，确切的单位时间是不同的，但是对于算法进行多少个基本操作（即花费多少时间单位）在规模数量级上却是相同的，由此可以忽略机器环境的影响而客观的反应算法的时间效率。 对于算法的时间效率，我们可以用“大O记法”来表示。

“大O记法”：对于单调的整数函数f，如果存在一个整数函数g和实常数c>0，使得对于充分大的n总有f(n)<=c*g(n)，就说函数g是f的一个渐近函数（忽略常数），记为f(n)=O(g(n))。也就是说，在趋向无穷的极限意义下，函数f的增长速度受到函数g的约束，亦即函数f与函数g的特征相似。

时间复杂度：假设存在函数g，使得算法A处理规模为n的问题示例所用时间为T(n)=O(g(n))，则称O(g(n))为算法A的渐近时间复杂度，简称时间复杂度，记为T(n)

4）如何理解“大O记法”

对于算法进行特别具体的细致分析虽然很好，但在实践中的实际价值有限。对于算法的时间性质和空间性质，最重要的是其数量级和趋势，这些是分析算法效率的主要部分。而计量算法基本操作数量的规模函数中那些常量因子可以忽略不计。例如，可以认为3n2和100n2属于同一个量级，如果两个算法处理同样规模实例的代价分别为这两个函数，就认为它们的效率“差不多”，都为n2级。

上面四组代码那组运行时间最短?

类比生活中的一些事件，估计时间：

1、眨一下眼	**几毫秒**
2、口算35+204	**几秒**
3、烧一壶水	**几分钟**
4、睡一觉	**几小时**
5、完成一个项目	**几天/几星期/几个月**
6、飞船从地球飞出太阳系	**几年**

1）print（“Hello World”）

2）
```
for i in range(n):
    print（“Hello World”）
```

3）
```
for i in range(n):
    for j  in range(n):
        print（“Hello World”）
```

4）
```
for i in range(n):
    for j in range(n):
        for k in range(n):
            print（“Hello World”）
```

```
print('Hello World')
```

$O(1)$

```
for i in range(n):
    print('Hello World')
```

$O(n)$

```
for i in range(n):
    for j in range(n):
        print('Hello World')
```

$O(n^2)$

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            print('Hello World')
```

$O(n^3)$

```
print('Hello World')

print('Hello Python')

print('Hello Algorithm')
```

$O(3)$ $O(1)$

```
for i in range(n):

        print('Hello World')
                for j in range(n):
                        print('Hello World')
```

$O(n^2+n)$ $O(n^2)$

```
for i in range(n):
    for j in range(i):
        print('Hello World')
```

$O(1/2n^2)$ $O(n^2)$

我们会说3个几毫秒？项目完成两个几天？同学们说5个工作日，那个是具体的描述，我们是具体的估计，我们运行代码时，你实际能看出它哪个快嘛？我们就用这个单位来估计，所以它还是右边的。同学们记住，我们除

了1以外，不会再遇到任何阿拉伯数字。

下面这个代码呢：

```
while n > 1:
    print(n)
    n = n // 2
```

$2^6=64$

$\log_2 64=6$

```
n=64 输出：

64

32

16

8

4

2
```

$O(\log_2 n)$

或

$O(\log n)$

读法：以2为底64的对数。默认以2为底，所以2省略。

5）最坏时间复杂度

分析算法时，存在几种可能的考虑：

- 算法完成工作最少需要多少基本操作，即最优时间复杂度
- 算法完成工作最多需要多少基本操作，即最坏时间复杂度
- 算法完成工作平均需要多少基本操作，即平均时间复杂度
- 对于最优时间复杂度，其价值不大，因为它没有提供什么有用信息，其反映的只是最乐观最理想的情况，没有参考价值。

对于最坏时间复杂度，提供了一种保证，表明算法在此种程度的基本操作中一定能完成工作。 对于平均时间复杂度，是对算法的一个全面评价，因此它完整全面的反映了这个算法的性质。但另一方面，这种衡量并没有保证，不是每个计算都能在这个基本操作内完成。而且，对于平均情况的计算，也会因为应用算法的实例分布可能并不均匀而难以计算。 **因此，我们主要关注算法的最坏情况，亦即最坏时间复杂度。**

6）时间复杂度的几条基本计算规则

1. 基本操作，即只有常数项，认为其时间复杂度为O(1)
2. 顺序结构，时间复杂度按加法进行计算
3. 循环结构，时间复杂度按乘法进行计算
4. 分支结构，时间复杂度取最大值
5. 判断一个算法的效率时，往往只需要关注操作数量的最高次项，其它次要项和常数项可以忽略

6. 在没有特殊说明时，我们所分析的算法的时间复杂度都是指最坏时间复杂度

7）空间复杂度

类似于时间复杂度的讨论，一个算法的空间复杂度S(n)定义为该算法所耗费的存储空间，它也是问题规模n的函数。

渐近空间复杂度也常常简称为空间复杂度。

空间复杂度(SpaceComplexity)是对一个算法在运行过程中临时占用存储空间大小的量度。

算法的时间复杂度和空间复杂度合称为算法的复杂度。

8）算法分析

Low B的方案算法核心部分

```
for a in range(0, 1001):
    for b in range(0, 1001):
        for c in range(0, 1001):
            if a**2 + b**2 == c**2 and a+b+c == 1000:
                print("a, b, c: %d, %d, %d" % (a, b, c))
```

时间复杂度：

T(n) = O(n*n*n) = O(n^3)

no BB的方案算法核心部分

```
for a in range(0, 1001):
    for b in range(0, 1001-a):
        c = 1000 - a - b
        if a**2 + b**2 == c**2:
            print("a, b, c: %d, %d, %d" % (a, b, c))
```

时间复杂度：

T(n) = O(n*n*(1+1)) = O(n*n) = O(n^2)

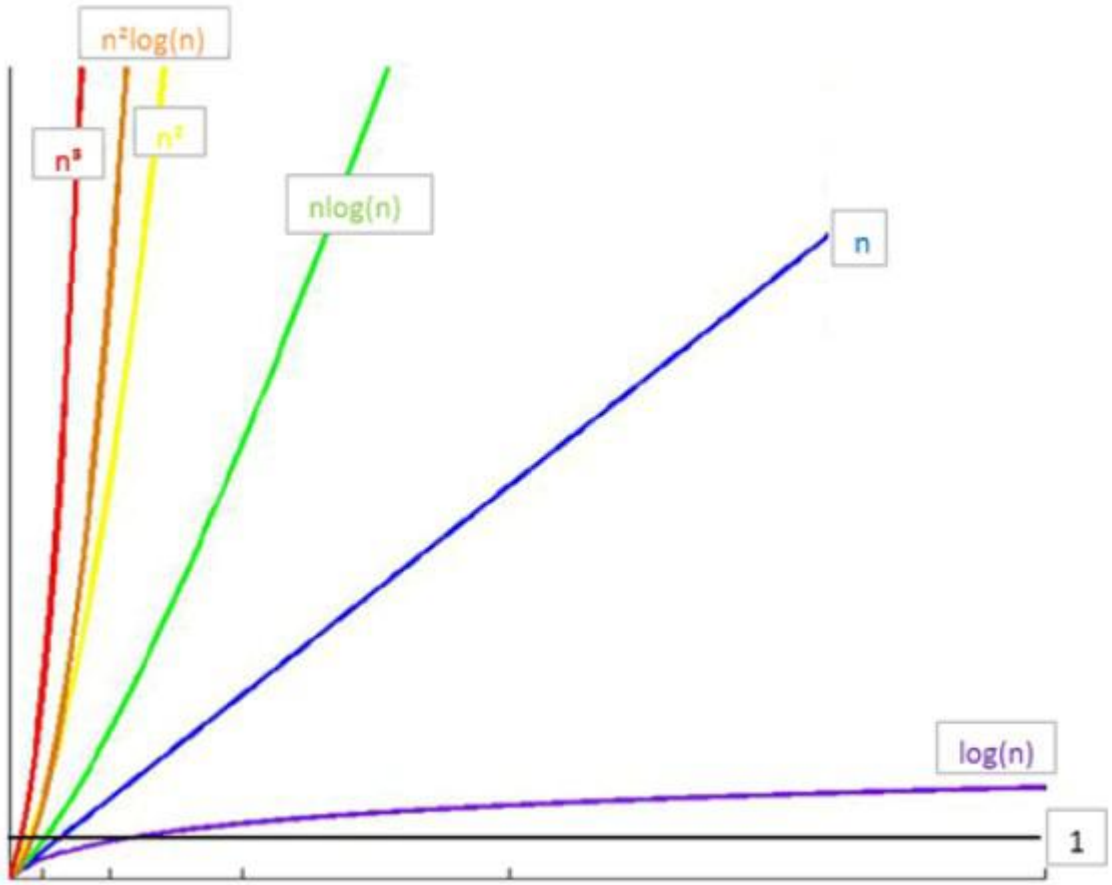
第二种算法要比第一种算法的时间复杂度好多了

3.常见时间复杂度

执行次数函数举例	阶	非正式术语
12	O(1)	常数阶
2n+3	O(n)	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
$5\log_2 n+20$	O(logn)	对数阶
$2n+3n\log_2 n+19$	O(nlogn)	nlogn阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

注意，经常将$\log_2 n$（以2为底的对数）简写成logn

常见复杂度之间的关系



所消耗时间从小到大为

$$O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

时间复杂度用来估计算法运行时间的一个式子（单位）

一般来说，时间复杂度高的算法比复杂度低的算法快？ （。。。）

4.Python内置类型性能分析

通常在一段程序的前后都用上time.time()，然后进行相减就可以得到一段程序的运行时间，我门可以用个装饰器弄一下,decrator.py

不过python提供了更强大的计时库：timeit

```
# timeit(函数名_字符串，运行环境_字符串，number=运行次数)
t = timeit('func()', 'from __main__ import func', number=1000)
```

list的操作测试:timeit_list.py

concat

append

comprehension

list range

pop操作测试 :timeit_pop.py

测试pop操作：从结果可以看出，pop最后一个元素的效率远远高于pop第一个元素

list 内置操作的时间复杂度

操作	时间复杂度
index[]	O(1)
index assingnment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,iterm)	O(n)
del	O(n)
iteration	O(n)
contains(in)	O(n)
set slice	O(n)
reverse	O(n)
sort	O(nlogn)

5.数据结构

我们如何用Python中的类型来保存一个班的学生信息？ 如果想要快速的通过学生姓名获取其信息呢？ 实际上当我们在思考这个问题的时候，我们已经用到了数据结构。列表和字典都可以存储一个班的学生信息，但是想要在列表中获取一名同学的信息时，就要遍历这个列表，其时间复杂度为O(n)，而使用字典存储时，可将学生姓名作为字典的键，学生信息作为值，进而查询时不需要遍历便可快速获取到学生信息，其时间复杂度为O(1)。

我们为了解决问题，需要将数据保存下来，然后根据数据的存储方式来设计算法实现进行处理，那么数据的存储方式不同就会导致需要不同的算法进行处理。我们希望算法解决问题的效率越快越好，于是我们就需要考虑数据究竟如何保存的问题，这就是数据结构。

在上面的问题中我们可以选择Python中的列表或字典来存储学生信息。列表和字典就是Python内建帮我们封装好的两种数据结构。

1）概念

数据是一个抽象的概念，将其进行分类后得到程序设计语言中的基本类型。如：int，float，char等。数据元素之间不是独立的，存在特定的关系，这些关系便是结构。数据结构指数据对象中数据元素之间的关系。

Python给我们提供了很多现成的数据结构类型，这些系统自己定义好的，不需要我们自己去定义的数据结构叫做Python的内置数据结构，比如列表、元组、字典。而有些数据组织方式，Python系统里面没有直接定义，需要我们自己去定义实现这些数据的组织方式，这些数据组织方式称之为Python的扩展数据结构，比如栈，队列等。

2）算法与数据结构的区别

数据结构只是静态的描述了数据元素之间的关系。

高效的程序需要在数据结构的基础上设计和选择算法。

程序 = 数据结构 + 算法

总结：算法是为了解决实际问题而设计的，数据结构是算法需要处理的问题载体

3）抽象数据类型(Abstract Data Type)

抽象数据类型(ADT)的含义是指一个数学模型以及定义在此数学模型上的一组操作。即把数据类型和数据类型上的运算捆在一起，进行封装。引入抽象数据类型的目的是把数据类型的表示和数据类型上运算的实现与这些数据类型和运算在程序中的引用隔开，使它们相互独立。

最常用的数据运算有五种：

- 插入
- 删除
- 修改
- 查找
- 排序

二、顺序表

在程序中，经常需要将一组（通常是同为某个类型的）数据元素作为整体管理和使用，需要创建这种元素组，用变量记录它们，传进传出函数等。一组数据中包含的元素个数可能发生变化（可以增加或删除元素）。

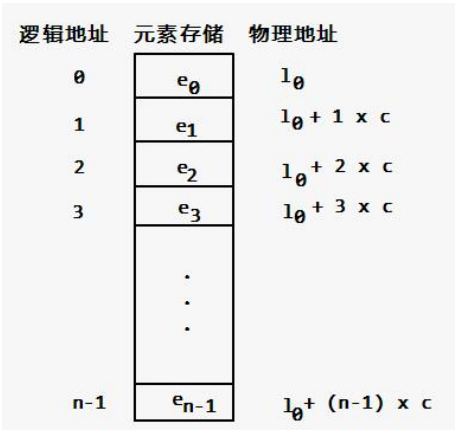
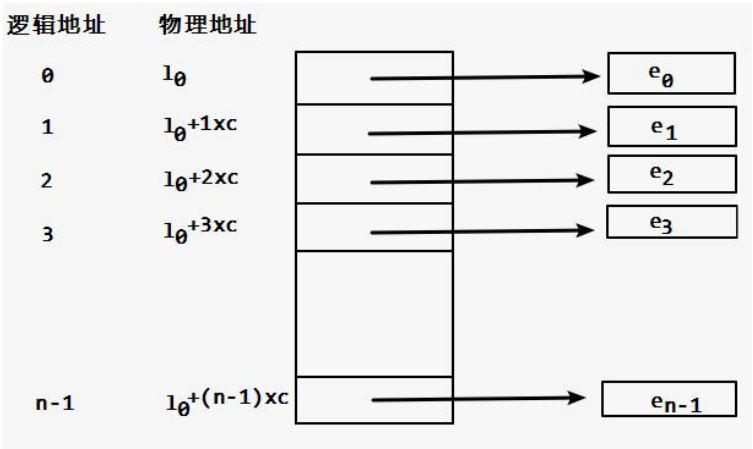
对于这种需求，最简单的解决方案便是将这样一组元素看成一个序列，用元素在序列里的位置和顺序，表示实际应用中的某种有意义的信息，或者表示数据之间的某种关系。

这样的一组序列元素的组织形式，我们可以将其抽象为线性表。一个线性表是某类元素的一个集合，还记录着元素之间的一种顺序关系。线性表是最基本的数据结构之一，在实际程序中应用非常广泛，它还经常被用作更复杂的数据结构的实现基础。

根据线性表的实际存储方式，分为两种实现模型：

- **顺序表**，将元素顺序地存放在一块连续的存储区里，元素间的顺序关系由它们的存储顺序自然表示。
- **链表**，将元素存放在通过链接构造起来的一系列存储块中。

1.顺序表的基本形式

a顺序表基本布局	b元素外置顺序表
	

图a表示的是顺序表的基本形式，数据元素本身连续存储，每个元素所占的存储单元大小固定相同，元素的下标是其逻辑地址，而元素存储的物理地址（实际内存地址）可以通过存储区的起始地址Loc(e0)加上逻辑地址（第i个元素）与存储单元大小（c）的乘积计算而得，即：

$$Loc(e_i) = Loc(e_0) + c * i$$

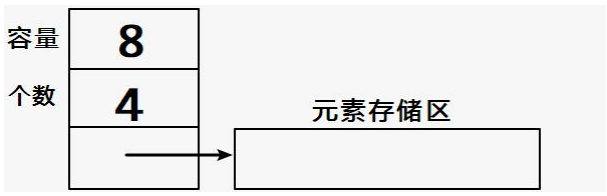
访问指定元素时无需从头遍历，通过计算便可获得对应地址，其时间复杂度为O(1)。 如果元素的大小不统一，则须采用图b的元素外置的形式，将实际数据元素另行存储，而顺序表中各单元位置保存对应元素的地址信息（即链接）。由于每个链接所需的存储量相同，通过上述公式，可以计算出元素链接的存储位置，而后顺着链接找到实际存储的数据元素。注意，图b中的c不再是数据元素的大小，而是存储一个链接地址所需的存储量，这个量通常很小。 图b这样的顺序表也被称为对实际数据的索引，这是最简单的索引结构。

2.顺序表的结构与实现

1) 顺序表的结构

一个顺序表的完整信息包括两个部分，一部分是表中的元素集合，另一部分是为实现正确操作而需记录的信息，即有关表的整体情况的信息，这部分信息主要包括元素存储区域的**容量**和当前表中的**元素个数**两项。

2) 顺序表的两种基本实现方式

a一体式结构	b分离式结构
	

- 一体式结构存储表信息的单元与元素存储区以连续的方式安排在一块存储区里，两部分数据的整体形成一个完整的顺序表对象。

- 一体式结构整体性强，易于管理。但是由于数据元素存储区域是表对象的一部分，顺序表创建后，元素存储区就固定了。
- 分离式结构表对象里只保存与整个表有关的信息（即容量和元素个数），实际数据元素存放在另一个独立的元素存储区里，通过链接与基本表对象关联。

3) 元素存储区替换

- 一体式结构由于顺序表信息区与数据区连续存储在一起，所以若想更换数据区，则只能整体搬迁，即整个顺序表对象（指存储顺序表的结构信息的区域）改变了。
- 分离式结构若想更换数据区，只需将表信息区中的数据区链接地址更新即可，而该顺序表对象不变。

3.python中的顺序表

Python中的list和tuple两种类型采用了顺序表的实现技术，具有前面讨论的顺序表的所有性质。 tuple是不可变类型，即不变的顺序表，因此不支持改变其内部状态的任何操作，而其他方面，则与list的性质类似。

list的基本实现原理 Python标准类型list就是一种元素个数可变的顺序表，可以加入和删除元素，并在各种操作中维持已有元素的顺序（即保序），而且还具有以下行为特征：

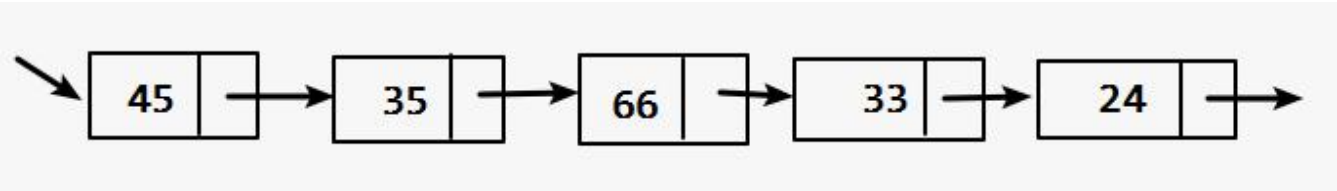
- 基于下标（位置）的高效元素访问和更新，时间复杂度应该是O(1)； 为满足该特征，应该采用顺序表技术，表中元素保存在一块连续的存储区中。
- 允许任意加入元素，而且在不断加入元素的过程中，表对象的标识（函数id得到的值）不变。 为满足该特征，就必须能更换元素存储区，并且为保证更换存储区时list对象的标识id不变，只能采用分离式实现技术。

在Python的官方实现中，list就是一种采用分离式技术实现的动态顺序表。这就是为什么用list.append(x) （或list.insert(len(list), x)，即尾部插入）比在指定位置插入元素效率高的原因。

三、链表

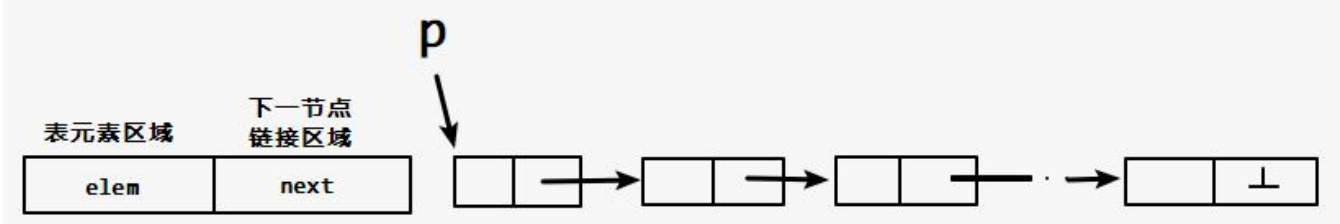
链表（Linked list） 是一种常见的基础数据结构，是一种线性表，但是不像顺序表一样连续存储数据，而是在每一个节点（数据存储单元）里存放下一个节点的位置信息（即地址）。

顺序表的构建需要预先知道数据大小来申请连续的存储空间，而在进行扩充时又需要进行数据的搬迁，所以使用起来并不是很灵活。 链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。



1. 单向链表

单向链表也叫单链表，是链表中最简单的一种形式，它的每个节点包含两个域，一个信息域（元素域）和一个链接域。这个链接指向链表中的下一个节点，而最后一个节点的链接域则指向一个空值。



- 表元素域 elem 用来存放具体的数据。
- 链接域 next 用来存放下一个节点的位置（python 中的标识）
- 变量 p 指向链表的头节点（首节点）的位置，从 p 出发能找到表中的任意节点。

节点实现

```
class SingleNode(object):
    """单链表的结点"""
    def __init__(self,item):
        # item 存放数据元素
        self.item = item
        # next 是下一个节点的标识
        self.next = None
```

1)单链表的操作

- is_empty() 链表是否为空
- length() 链表长度
- travel() 遍历整个链表
- add(item) 链表头部添加元素
- append(item) 链表尾部添加元素
- insert(pos, item) 指定位置添加元素
- remove(item) 删除节点
- search(item) 查找节点是否存在

2)单链表的实现

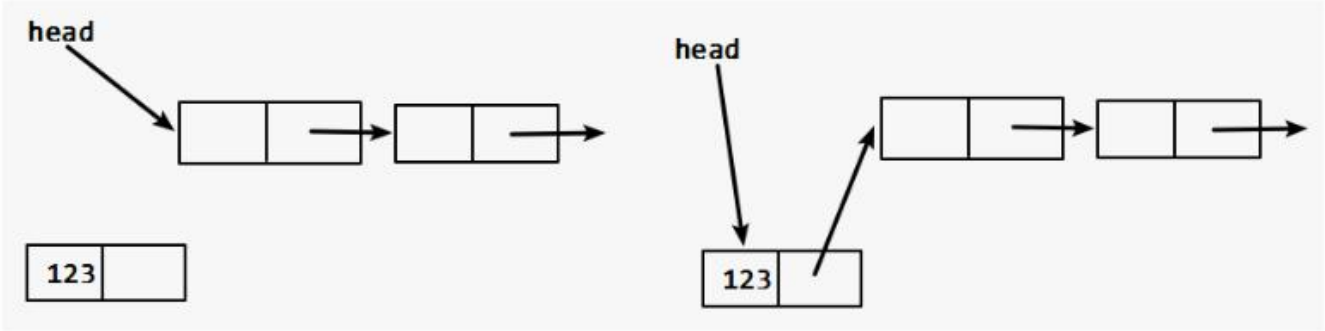
```
class SingleLinkList(object):
    """单链表"""
    def __init__(self):
        self.__head = None

    def is_empty(self):
        """判断链表是否为空"""
        return self.__head == None

    def length(self):
        """链表长度"""
        # cur 初始时指向头节点
        cur = self.__head
        count = 0
        # 尾节点指向 None，当未到达尾部时
        while cur != None:
            count += 1
            # 将 cur 后移一个节点
            cur = cur.next
        return count

    def travel(self):
        """遍历链表"""
        cur = self.__head
        while cur != None:
            print(cur.item)
            cur = cur.next
```

3)头部添加元素

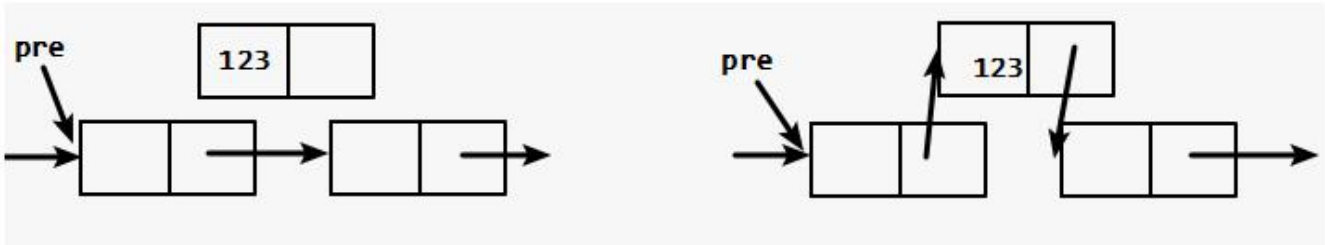



```
def add(self,item):
        '''头部添加元素'''
        #创建一个保存 item 的节点
        node = SingleNode(item)
        #将新节点的链接区域 next 指向头节点，即__head 指向的位置
        node.next = self.__head
        #将链表的__head 指向新节点
        self.__head = node
```

4)尾部添加元素

```
def append(self,item):
    '''尾部添加元素'''
    node = SingleNode(item)
    #先判断链表是否为空，若为空则将__head 指向新节点
    if self.is_empty():
        self.__head = node
    #若不为空，则找到尾部，将尾部节点的 next 指向新节点
    else:
        cur = self.__head
        while cur.next != None:
            cur = cur.next
        cur.next = node
```

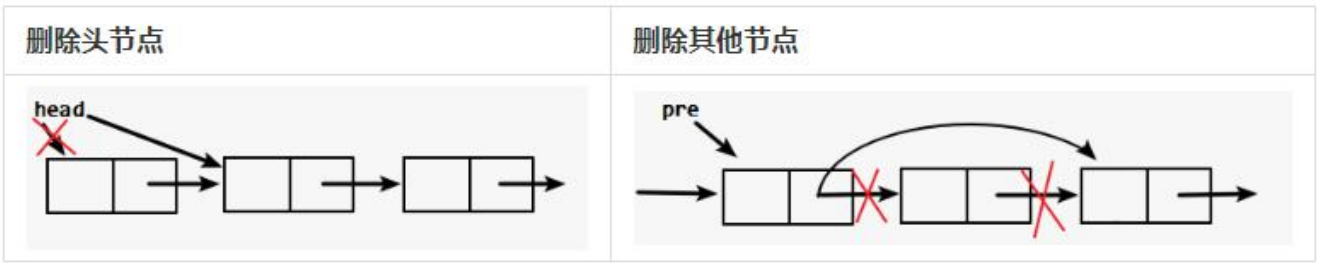
5)指定位置添加元素




```
def insert(self,pos,item):
    '''指定位置添加元素'''
    #若指定位置 pos 为第一个元素之前，则执行头部插入
    if pos <= 0:
        self.add(item)
    #若指定位置超过链表尾部，则执行尾部插入
    elif pos > self.length()-1:
        self.append(item)
```

```
    #找到指定位置
    else:
        node = SingleNode(item)
        count = 0
        #pre 用来指定位置 pos 的前一个位置
        pre = self.__head
        while count < pos -1:
            count += 1
            pre = pre.next
        #先将新节点的 node 的 next 指向插入位置节点(pre.next)
        node.next = pre.next
        #再将插入位置的前一个节点 pre 的 next 指向新的节点
        pre.next = node
```

6)删除节点




```
def remove(self,item):
    '''删除节点'''
    cur = self.__head
    pre = None
    while cur != None:
        #找到指定元素
        if cur.item == item:
            #如果第一个就是删除的节点(not None)
            if not pre:
                #将头指针指向头节点的后一个节点
                self.__head = cur.next
            else:
                #将删除位置前一个节点的 next 指向删除位置的后一个节点
                pre.next = cur.next
            break
        else:
            #继续后移节点
            pre = cur
            cur = cur.next
```

7)查找节点是否存在

```
def search(self,item):
    '''链表查找节点是否存在，并返回 True 或 False'''
    cur = self.__head
    while cur != None:
        if cur.item == item:
            return True
        cur = cur.next
    return False
```

8)测试

```
if __name__ == '__main__':
```

```
    L1 = SingleLinkList()
    L1.add(1)
    L1.add(5)
    L1.add(9)
    print('当前长度是%s'%L1.length())
    L1.travel()
    L1.append('haha')
    print('-------------')
    print('当前长度是%s' % L1.length())
    L1.travel()
    print(L1.search(5))
    L1.remove(1)
    print('-------------')
    print('当前长度是%s' % L1.length())
    L1.travel()
    print(L1.search(1))
    print('-------------')
    L1.insert(1,'heihei')
    print('当前长度是%s' % L1.length())
    L1.travel()



当前长度是 3
9
5
1
-------------
当前长度是 4
9
5
1
haha
True
-------------
当前长度是 3
9
5
haha
False
-------------
当前长度是 4
9
heihei
5
haha
```

2.链表与顺序表的对比

链表失去了顺序表随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大，但对存储空间的使用要相对灵活。 链表与顺序表的各种操作复杂度如下所示：

操作	顺序表	链表
访问元素	O(1)	O(n)

操作	顺序表	链表
在头部插入/删除	O(n)	O(1)
在尾部插入/删除	O(1)	O(n)
在中级插入/删除	O(n)	O(n)

链表和顺序表在插入和删除时进行的是完全不同的操作。链表的主要耗时操作是遍历查找，删除和插入操作本身的复杂度是 O(1)。顺序表查找很快，主要耗时的操作是拷贝覆盖。因为除了目标元素在尾部的特殊情况，顺序表进行插入和删除时需要对操作点之后的所有元素进行前后移位操作，只能通过拷贝和覆盖的方法进行