数据结构与算法

学习手册

作者：itcast

版本：v 2.3

文档编号：【20180206】

日期：2018年04月20日

摘 要

本文档说明了 数据结构与算法 使用规范相关的内容

历史变更记录

日期	当前版本号	作者	变更内容	备注

目录

第 1 章 基础

我们的这门课讲完后，大家就会发现通篇围绕了介绍了四个字：数据、成本。至于是怎么回事儿，我们接下来好好的学习。

1.1 为什么学？

我们从两个方面来学习：

特点，怎么学

特点是什么？

高：需要知识水平高,一定的知识储备

大：能做"大事"

上：公司的顶级技术人才的归宿

如何学好"它们"？

1、了解它们

数据结构和算法是从事程序开发工作的基本功

基本功，俩个字：累、慢

累：学习非常累，而且很枯燥

慢：见效慢

2、认识它们

现阶段：我们入职的一个高级"敲门砖"

远阶段：作为技术骨干的核心价值

3、搞定它们

升职加薪、迎娶白富美、走上人生巅峰

1.2 简介

我们主要从四个方面来介绍这块知识：

什么是数据结构、什么是算法、两者区别、抽象数据类型

什么是数据结构

存储、组织数据的方式

数据的种类有很多：字符串、整数、浮点、...

组织各种数据的方式：即数据元素之间的关系

列表、字典、元组、...

举例：

列表方式：

```
[老王,18,男]
```

数组方式：

```
{name:"老王",age:18,sex:"男"}
```

综合方式：

```
[{name:"老王",age:18,sex:"男"},{name:"老李",age:19,sex:"男"}]
{老王:{age:18,sex:"男"},老李:{age:19,sex:"男"}}
```

数据的结构有两种形式：

物理形式：顺序表、链表

逻辑形式：集合、线性、树形、图形

算法是什么？

为了实现业务目的各种分析方法和思路就是算法

公司的核心：数据

数据的存储：数据结构：

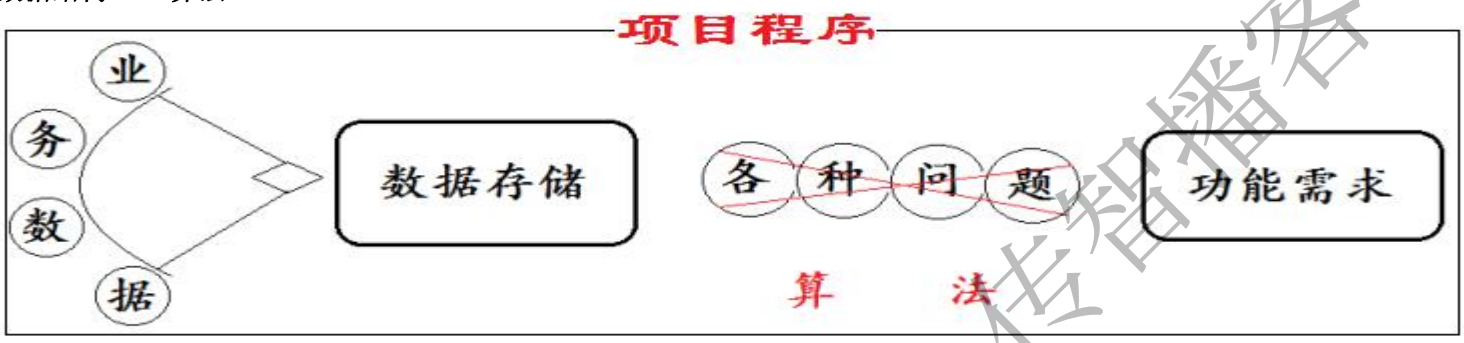
存储数据的目的：提供信息和解决问题

算法复杂度：

时间复杂度：代码执行的时间

空间复杂度：代码运行的空间

数据结构 vs 算法



业务数据：

用户访问业务时候，产生的信息内容

数据结构：

静态的描述了数据元素之间的关系

算法：

解决各种实际问题的方法和思路

数据结构 + 算法 = 程序

案例举例：天天生鲜、ihome

我们一般说的数据不是干巴巴的字母数字，而是在某种场景下来对这些数据的含义进行分析等操作，数据一旦有了场景意义：

"一" 在不同场景的声音和含义。

纯粹的数据加上场景，他们就有了新的名称：ADT

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类型以及定义在此数学类型上的一组操作。即把数据类型和数据类型上的运算捆在一起，进行封装。

举例一：

数据类型-人	多个人，陈浩南、山鸡、大天二、大飞...
数据运算-关系	彼此间的团队联系
抽象数据类型=类型+运算=人+关系	洪兴

举例二：

游戏按钮“空格”：人物A(数据类型) + 打子弹(动作)

引入抽象数据类型的目的：

是把数据类型的表示和运算的实现与这两者在程序中的引用隔开，使它们相互独立。

简单来说：就是把数据带入场景，让他们“有意义”

常见数据运算类型：

插入、删除、修改、查找、排序，即所谓的“增删改查”

第 2 章 快速入门

快速入门这部分我们从三个方面来学习：

入门案例、算法复杂度、性能分析模块详解

2.1 入门案例

入门案例的目的：

一个日常的查询示例，多种解决方法，进而了解：同样是做一件事情，为什么有些人待遇就高的不得了。

案例：

如果 a+b+c=1000，且 a^2+b^2=c^2（a,b,c 为自然数），求出所有a、b、c可能的组合？

2.1.1 解决方法一

思路分析：

笨方法：

一个一个的猜　　穷举法或者枚举法

代码实现

```
import time

start_time = time.time()

# 注意是三重循环
for a in range(1001):
    for b in range(1001):
        for c in range(1001):
            if 1000 == a + b + c and a**2 + b**2 == c**2:
                print("a,b,c: %d,%d,%d" % (a,b,c))
```

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

观察执行时间： 305.590478

思路总结：**算法特性**

1、输入：算法具有0个或多个输入
2、输出：算法至少有1个或多个输出
3、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
4、确定性：算法中的每一步都有确定的含义，不会出现二义性
5、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

一句话：

量入出题题能解，行有意步步可为。

2.1.2 解决方法二

我们知道a、b、c他们三者是有一个关系的，我就可以不用对c进行遍历了，直接把c列成一个条件即可

代码实现

```
import time


start_time = time.time()


# 注意是两重循环
for a in range(1001):
    for b in range(1001):
        # 根据条件直接获取 c 的值
        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()
cost = end_time - start_time
print("elapsed: %f" % (cost))
print("complete!")
```

观察执行时间： 2.464141

2.1.3 方法思考

问题现象：

两次解决思路的执行时间：305.590478：2.464141 相差了将近 124倍，

问：

为什么不一样？
哪个方法好，时间慢的方法就好么？

答：

1、因为不一样，所以不一样。

2、不一定

计算依赖于服务器环境，环境不唯一，所以用时间来评判，不准确。

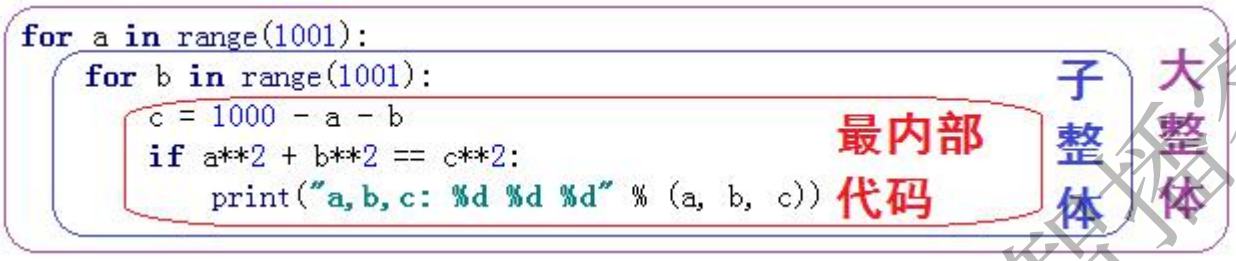
问： 如何综合评判两种方法？

分析：至少把代码运行的环境因素抛离出去，就剩下代码本身，代码本身使我们解决思路的实现，也就是算法

那么我们应该怎么评判算法呢？：

答：

代码量太大，没办法整体分析，我们就拆，拆到我们能接收的方式来分析：



我们对于每一个循环，他们都是执行的最内部代码，所以我们只需要分析好最内部代码的时间花费，整体代码的分析无非就是乘以循环数量就可以了，这个循环数量我们成为"步骤数量"。

而最内部代码的执行时间，我们成为单位时间

所以评判一个算法是否优秀，我们应该从两个方面来思考方法的优劣：

步骤数量和单位时间

问：步骤数量和单位时间的关系？

步骤数量 * 单步执行时间 == 代码执行总时间

2.1.4 步骤数量

步骤数量算什么因素？

示例：

从1数到100，从1数到1000，从1数到10000000万

数量越大，那么解决问题的步骤越多，其实归根结底就是：处理问题的**规模**问题。

规模再大，也要从最基础的数量来解决，我们首先来说一下数量的计算。

步骤数量：

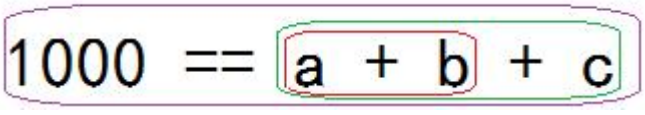
就是一段代码执行的步骤总量。

方法一代码总数量计算：

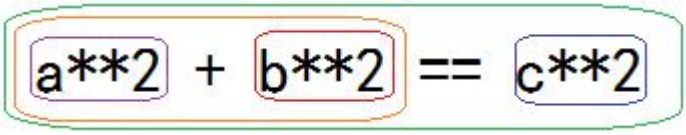
```
for a in range(1001):                                    # 1001次
    for b in range(1001):                                # 1001次
        for c in range(1001):                            # 1001次
            if 1000 == a + b + c and a**2 + b**2 == c**2:
                print("a,b,c: %d,%d,%d" % (a,b,c))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

分析：每个循环的基本单位是：if的语句 + print的语句

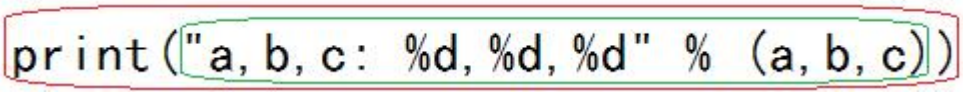
1000 == a + b + c 为：3次



a**2 + b**2 == c**2 为：5次



print("a,b,c: %d,%d,%d" % (a,b,c))为：2次



所以每个循环的步骤次数是：3+5+2=10次

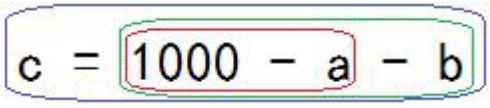
代码步骤总数量是：1001*1001*1001*10 次

方法二代码总数量计算：

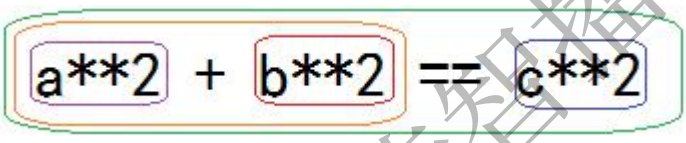
```
for a in range(1001):                                    # 1001次
    for b in range(1001):                                # 1001次
        c = 1000 - a - b
        if a**2 + b**2 == c**2:
            print("a,b,c: %d,%d,%d" % (a,b,c))
```

分析：每个循环的基本单位是：c计算 + if的语句 + print的语句

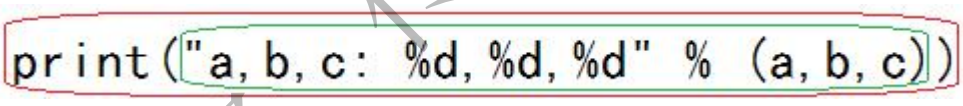
c = 1000 - a - b 为：3次



a**2 + b**2 == c**2 为：5次



print("a,b,c: %d,%d,%d" % (a,b,c))为：2次



所以每个循环的步骤次数是：3+5+2=10次

代码步骤总数量是：1001*1001*10 次

规律总结：

以方法二为例：通过规模的不同，来计算时间总量T

总量为1000：

T = 1001*1001*10 次

总量为2000：

T = 2001*2001*10 次

总量为3000：

T = 3001*3001*10 次

...

总量为n：

T = n*n*10 次

我们就将上面的算数，总结为一个表达式：

T(n) = n*n*10

我们把这个表达式称为：时间复杂度

2.2 算法复杂度

算法复杂度的目的：分析代码执行的时间成本。

我们从五个方面来介绍算法复杂度：

时间复杂度、时间复杂度分类、时间复杂度计算规则、空间复杂度、实践

2.2.1 时间复杂度

什么是时间复杂度？

做一件事情需要花费多少时间

花费时间举例：

眨眼、口算1+1、找个女朋友

通过上面的总结出来的表达式，我们知道，表达式中具体的数字10其实处于一个次要的关系，其实我们在总结规律的时候，其实就是将所有和主干无关的条件都抛开，就分析主干，在我们这里其实本质上是T和n的关系。

如果将次要关系都慢慢的省略掉的话，就会逐渐演变成一个渐变的函数O(g(n))，我们一般把这种渐进的表示方法，称为：大o记法

渐变函数(规律函数)：其实就是所谓的“g(n)长大后我就成了你O(g(n))”

他的重点是：规律趋势

所以，我们可以将这个表达式，最终演变成一个T和n的表达式：

假设存在算法A函数g(n)=n*n*10，总结规律时候，把无关紧要的条件10去除掉，就变成了一个渐进的算法A函数O(g(n))，所以最终的时间T(n)=O(g(n))

我们一般称O(g(n))为算法A的渐近时间复杂度，简称时间复杂度，标记为T(n)

2.2.2 时间复杂度分类

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

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。

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。

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。

我们知道有一个“木桶原理”，所以我们把问题最坏的一个条件作为处理该问题的最低标准，所以我们平常所说的时间复杂度，其实说的是"最坏时间复杂度"。

2.2.3 基本计算规则

时间复杂度的6条基本计算规则

1、基本操作，即只有常数项

简单来说：没有数量规模，就执行一次

时间复杂度：O(1)

2、顺序结构，时间复杂度按加法进行计算

简单来说：一步一步的执行下去，类似上面的方法二

```
c = 1000 - a - b
if a**2 + b**2 == c**2:
    print("a,b,c: %d,%d,%d" % (a,b,c))
```

1 2

时间复杂度：O(n)

3、循环结构，时间复杂度按乘法进行计算

简单循环：就是批量执行多次，类似于上面的方法一：

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

n n n

时间复杂度：$O(n^3)$

递归循环：重复同样的步骤的重复次数

时间复杂度O(logn)

4、分支结构，时间复杂度取最大值

简单来说：就是多分支if语句，找一个时间最长的作为标准的时间

```
if XXX:
  print("...")
elif XXX:
  print("...")
else:
  print("...")
```

5 11 4

最终以最长的11为最终的花费时间

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

2.2.4 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。

其实简单来说：

空间复杂度在代码上的表现形式就是：我用一行代码表示还是两行代码表示，例如方法一和方法二：

方法一：2

```
if 1000 == a + b + c and a**2 + b**2 == c**2:
    print("a,b,c: %d,%d,%d" % (a,b,c))
```

方法二：3

```
c = 1000 - a - b
if a**2 + b**2 == c**2:
    print("a,b,c: %d,%d,%d" % (a,b,c))
```

在例子中：方法1就是方法2 的渐进趋势函数

注意：

一般情况下，代码量少的情况下，不需要考虑空间复杂度，而代码量非常大的时候，才会考虑到空间复杂度。

空间复杂度其实就是用空间换时间。

简单来说：同样一件事情，一个人干和多个人干，通过人数量增加换取时间的降低

算法的时间复杂度和空间复杂度合称为算法的复杂度。

2.2.5 算法复杂度实践

方法一：

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

分析：

主干：1001*1001*1001*10

表达式：n*n*n*10

算法复杂度：

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

方法二：

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

分析：

主干：1001*1001*10

表达式：n*n*10

算数复杂度：

$T(n) = O(n*n) = O(n^2)$

因为 $O(n^2) < O(n^3)$，所以方法二的算法效率高一点。

2.2.6 常见时间复杂度

常见函数样式

执行次数函数举例	阶	非正式术语

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
$5log2^n+20$	O(logn)	对数阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶

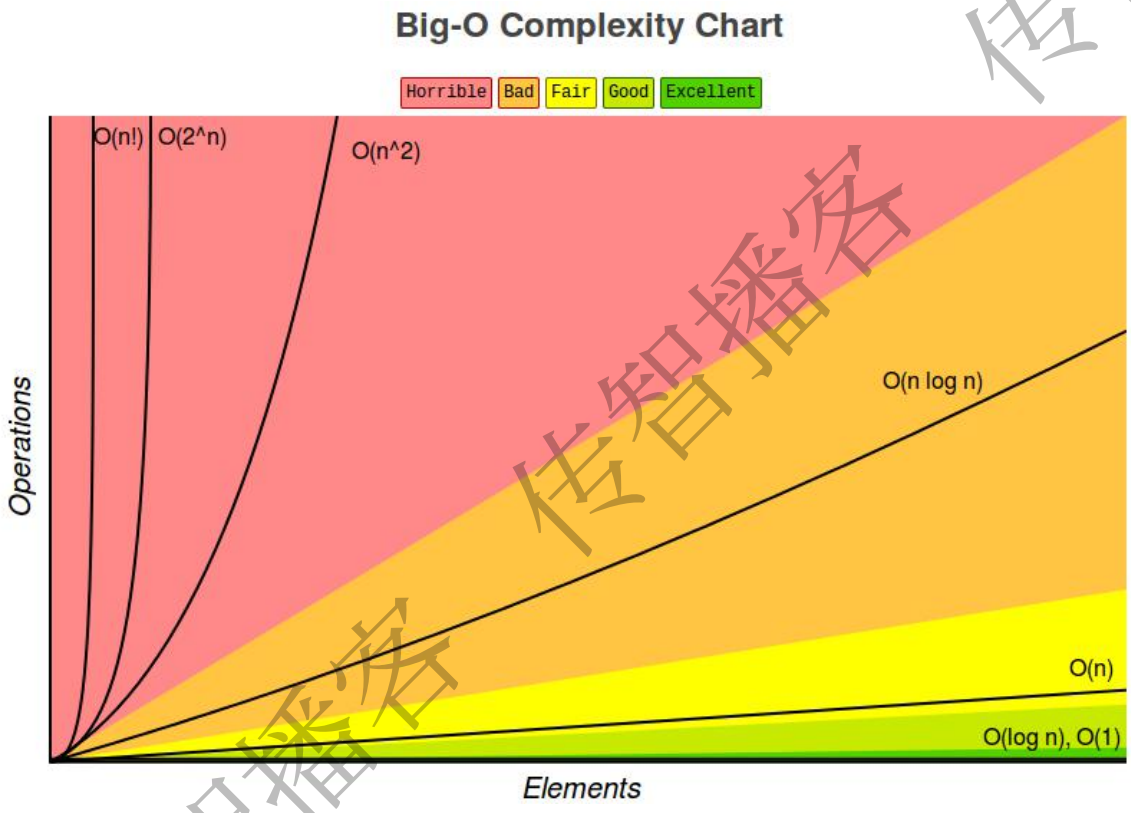
注意：

$2^m = n$，知道n的值，求m，

$m=log2^n$ 而我们经常将$log2^n$（以2为底的对数）简写成logn

我们一般在递归循环的时候使用到这个

常见时间复杂度之间的关系



所消耗的时间从小到大

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

简单来说：数字最小，n越多，值越大

时间复杂度越小，效率越高，时间越短

练习： 时间复杂度练习（ 参考算法的效率规则判断 ）

$O(5)$

$O(2n + 1)$

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

$O(3n^3 +1)$

2.3 性能分析模型详解

这一部分：我们介绍一个python代码性能测试的模块timeit，然后通过一些简单的代码，对这个代码进行性能测试。

2.3.1 timeit模块简介

timeit模块

timeit模块可以用来测试一小段Python代码的执行速度。

核心代码1介绍

```
class timeit.Timer(stmt='pass', setup='pass', timer=<timer function>)
```

代码详解：

Timer是测量小段代码执行速度的类。

stmt参数是要测试的代码语句（statment）

例子：

测试某个函数，函数代码用"func()"表示

setup参数是运行代码时需要的设置；

例子：

测试当前文档的func模块，使用 "from __main__ import func"

timer参数是一个定时器函数，与操作系统平台有关，一般省略。

示例：

```
timer1 = timeit.Timer("T1()","from __main__ import T1")
```

核心代码2介绍

```
timeit.Timer.timeit(number=1000000)
```

代码详解：

Timer类中测试语句执行速度的对象方法。

number参数是测试代码时的测试次数，默认为1000000次。

timeit方法返回执行代码的平均耗时，它是一个float类型的秒数。

简写方法：测试代码.timeit(1000000)

示例：

```
timer1.timeit(number=1000) 简写 timer1.timeit(1000)
```

2.3.2 列表性能测试实践

我们对列表的四个大类进行测试：

列表内置属性	list.append()
列表拼接	list1 + list2
列表推导式	[x for x in range(10)]
列表属性转换	list(range(10))

列表内置属性性能测试：

代码演示：

```
import timeit
def T1():
    li = []
    for i in range(1000):
        li.append(i)
```

```
timer1 = timeit.Timer("T1()","from __main__ import T1")
print("内置属性: %f seconds" % timer1.timeit(number=1000))
```

执行效果:

```
内置属性: 0.234965 seconds
```

列表拼接性能测试:

代码演示:

```
import timeit
def T2():
    li = []
    for i in range(1000):
        li = li + [i]

timer2 = timeit.Timer("T2()","from __main__ import T2")
print("列表拼接: %f seconds" % timer2.timeit(number=1000))
```

执行效果:

```
列表拼接: 4.817975 seconds
```

列表推导式性能测试:

代码演示:

```
import timeit
def T3():
    li = [i for i in range(1000)]

timer3 = timeit.Timer("T3()","from __main__ import T3")
print("列表推导: %f seconds" % timer3.timeit(number=1000))
```

执行效果:

```
列表推导: 0.085915 seconds
```

列表属性转换性能测试:

代码演示:

```
import timeit
def T4():
    li = list(range(1000))

timer4 = timeit.Timer("T4()","from __main__ import T4")
print("列表拼接: %f seconds" % timer4.timeit(number=1000))
```

执行效果:

```
属性转换: 0.042624 seconds
```

第 3 章 数据结构 进阶

这一部分，我们会从六个方向来介绍数据结构：

线性表、顺序表、链表、栈、队列、树

3.1 线性表

3.1.1 线性表简介

什么是线性表?

线性表：顾名思义：就是将一大堆同类型的数据，按照某种关系，依次排列出来，就形成了一条线。

特点：数据之间，只有前后关系。

举例：

100 200 300 400 。。。 n

线性表是最基本，也使用最广的数据结构之一，常被用作更复杂的数据结构的实现基础。

线性表分类：

根据线性表的实际存储方式，分为两种常见的模型：

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。

一句话：谁先来？ 或者 排排队吃果果

链表：

将元素存放在通过链接构造起来的一系列存储块中。也就是说，这种表不但有自己的数据信息，也有下一个数据结点的地址信息

链表有如下分类：单向链表、双向链表、单向循环链表

一句话：我先，谁在我后面？ 或者 吃套餐

3.2 顺序表

这一节，我们从五个方面来学习顺序表：

顺序表形式、顺序表结构、顺序表实践、顺序表常见操作、python顺序表。

3.2.1 顺序表形式

顺序表两种形式：

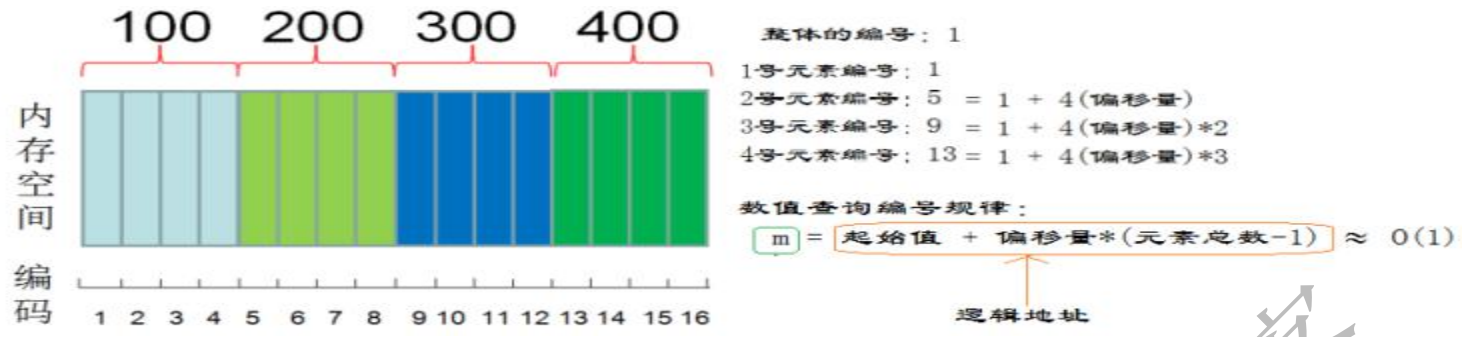
基本布局 + 元素外置

基本布局存储同类数据

元素外置存储异类数据

a) 基本布局：

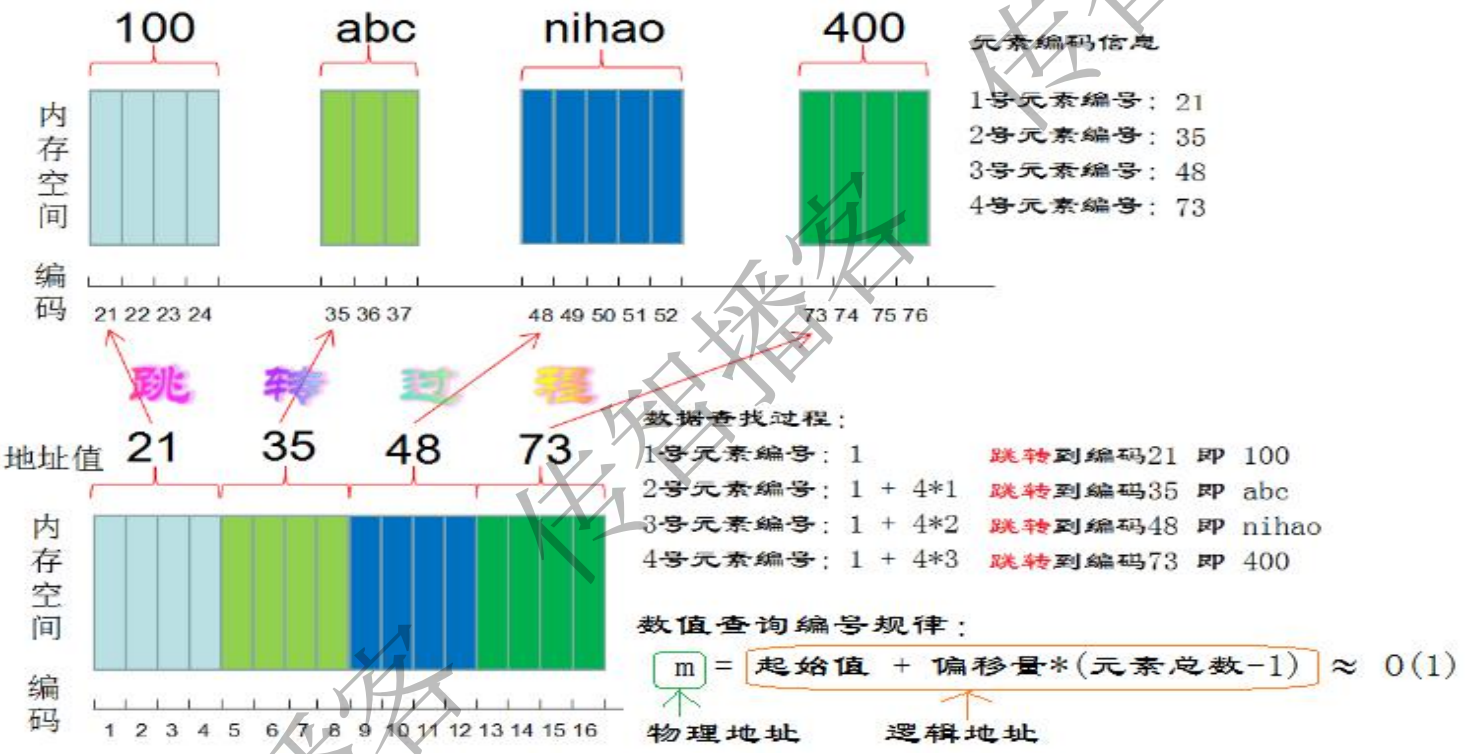
简单来说，就是存储多个同类型的数据，每个数据占用一个固定存储空间，多个数据组合在一块的时候，物理地址就会连续起来



b） 元素外置

元素外置，适用于存储不同类型的数据A-D，因为不同类型数据的占用存储空间不一样，不能按照基本布局的方式来存储。

通过比较发现，虽然他们数据占用的空间不一致，但是他们的逻辑地址编号都是同一的数据类型，所以存储他们的时候，可以单独申请一块连续空间，来存储A-D数据的逻辑地址。

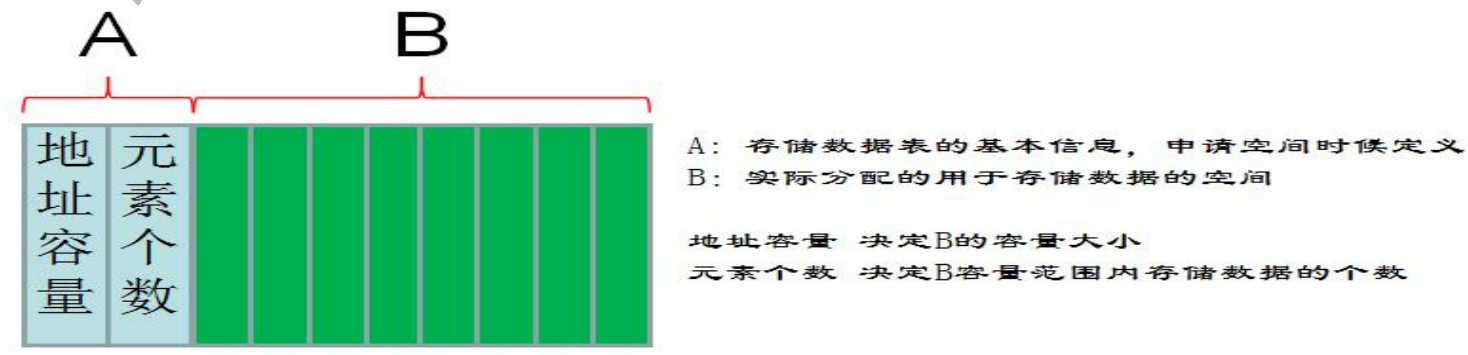


图b这样的顺序表，因为顺序表B内存储的是上面顺序表A的逻辑地址数值，通过存储的逻辑地址，找到真正存储数据的地址，所以顺序表B中的数值也被称为对实际数据的索引，这是最简单的索引结构。

3.2.2 顺序表结构

现在我们介绍一下顺序表的结构

顺序表结构



通过上图的介绍，我们可以知道顺序表由两部分组成：基本信息+存储元素。
因为顺序表有两中表现形式，所以就出现了两种表现形式：

基本布局：一体式结构
元素外置：分离式结构

整合式　或　一体式

一体式结构：

存储表信息与元素存储区以连续的方式安排在一块存储区里，形成一个完整的顺序表对象。

一体式特点：

因为整体，所以易于管理，顺序表创建后，元素存储区就固定了。

分离式

分离式结构：

顺序表对象里保存基本信息(地址容量+元素个数)和元素存储空间的一个逻辑地址(A)，通过逻辑地址(A)找到真正的数据存储地址。

分离式特点：

因为分离，所以灵活，元素的存储空间可以灵活调整

3.2.3 顺序表实践

我们在设定数据表的时候，会首先申请我要使用的存储空间。而存储空间中的数据不是一成不变的，接下来我们来介绍一下元素存储内容变化时候顺序表的变化。

存储内容的变化，无非是少或多，少会造成存储空间的浪费，多的话，存储空间就会发生变化。我们接下来主要是对存储内容更改或者存储内容大于计划空间的这两种情况进行分析。

内容更改

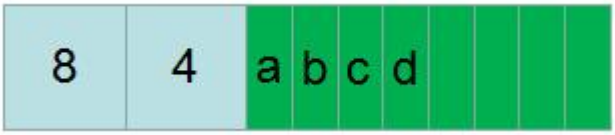
一体式：

因为一体式，基本信息和存储元素是一个整体的对象，而且一旦定义，就不会在更改了。

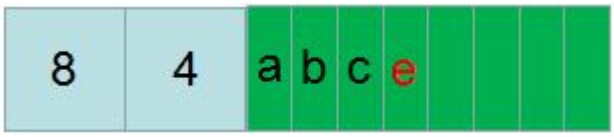
存储内容一旦大于计划容量，若要存储更多内容，只能整体搬迁。即整个顺序表对象（指存储顺序表的结构信息的区域）改变了

示例：

计划8个容量，存储4个值



更改最后一个数据的内容d为e



因为存储数据变化，所以顺序表对象就变动了

分离式：

因为分离式，基本信息和存储元素是两个独立的对象，他们彼此间使用基本信息后面的逻辑地址来连接，所以存储数据变动，只不过是存储的逻辑地址变动，而存储对象中的基本信息没有发生变动，所以，对于顺序表对象来说，没有发生变动

示例：

计划8个容量，存储4个值

更改最后一个数据的内容d为e

通过上面对于两种顺序表结构的内容更新实践分析，数据发生变动，我们推荐使用分离式顺序表结构，为什么？因为对象没有变动，一体式相当于重新开辟了一片空间，产生了一个新的顺序表对象，没有分离式省事简便。

所以接下来分析数据内容增加的时候，我们只针对分离式顺序表结构。

空间扩容

采用分离式结构的顺序表，可以在不改变表对象的前提下，将数据存储区更换为存储空间更大的区域，所有使用这个顺序表的地方都不必修改。只要程序的运行环境（计算机系统）还有空闲存储，这种表结构就不会因为数据存储区空间不足满了而导致操作无法进行。人们把采用这种技术实现的顺序表称为**动态顺序表**，因为其容量可以在使用中动态变化。

目前动态顺序表的扩容策略分为两大类：线性增长、倍数增长

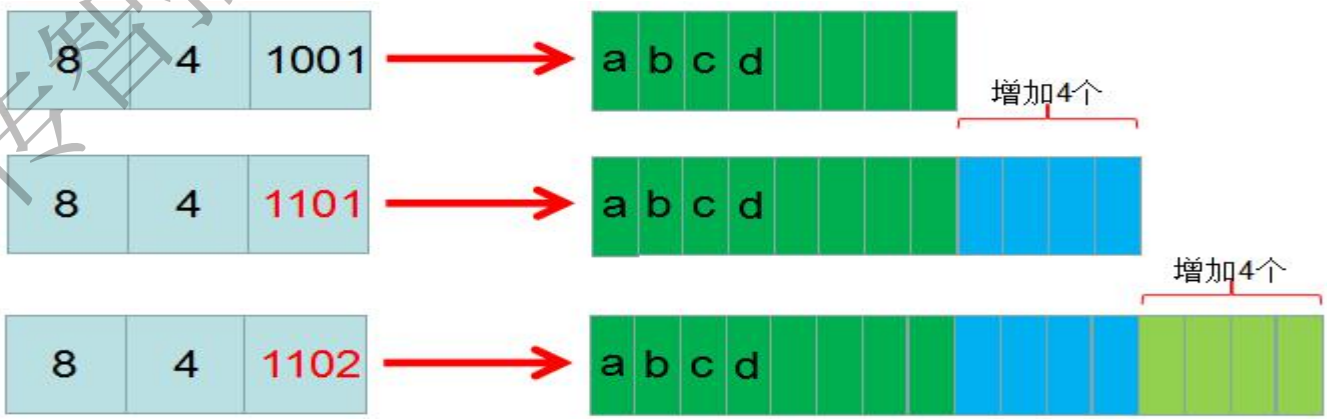
线性增长：

数据存储区容量一旦发现不足，每次扩充增加固定数目的存储区容量，如每次扩充增加4个元素位置。

特点：

确定性：因为增加数量有限制，所以节省空间，

不确定性：扩充操作可能频繁发生、随时发生。



倍数增长：

数据存储区容量一旦发现不足，每次扩充容量加倍，如：

当前容量8，容量不足时候，就扩充8，最终容量为16。

当前容量16，容量不足时候，就扩充16，最终容量为32。

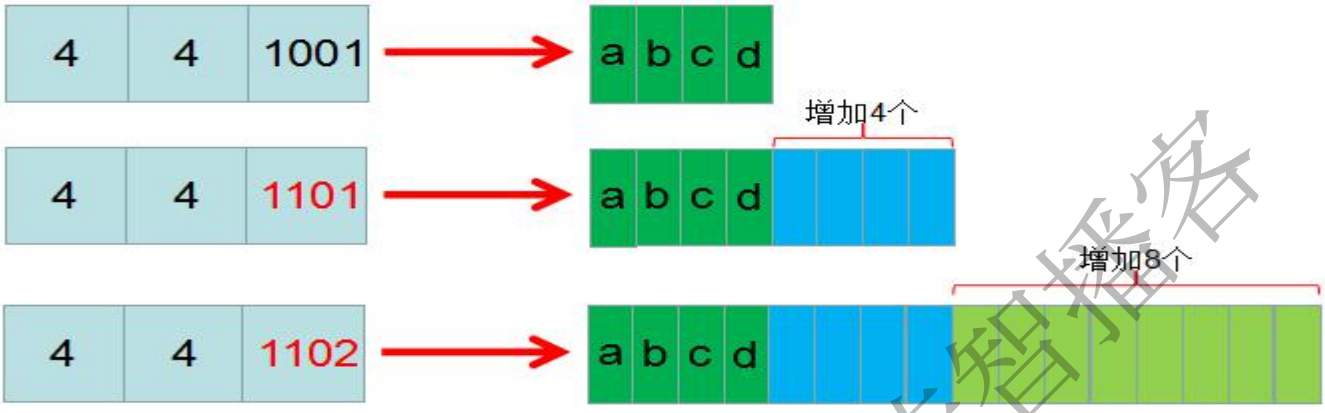
...

如此循环往复，每次扩充的容量都是当前的容量空间大小。

特点：

确定性：每次增加容量会很大，所以扩充操作的执行次数不会频繁发生

不确定性：容量扩充答，可能会造成空间资源浪费。



根据上面对于动态顺序表的容量增加分析，可以知道，倍数增长方式是以空间换时间,因为目前空间代价越来越低廉，所以我们工作中推荐使用倍数增长的方式来调整容量空间。

隐含意思：单一元素的值就是顺序表 -- 后续所有知识点都会用到这一个认知

3.2.4 顺序表常见操作

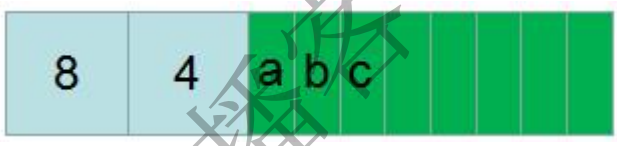
顺序表的操作，无非就是我们平常所说的“增删改查”，目前根据我们上面的数据表形式的介绍，我们学习了数据表的“查”操作，通过顺序表结构和实践，我们学习了数据表的“改”操作。所以这一节，我们**重点**学习顺序表的“增”操作和“删”操作

“增”操作

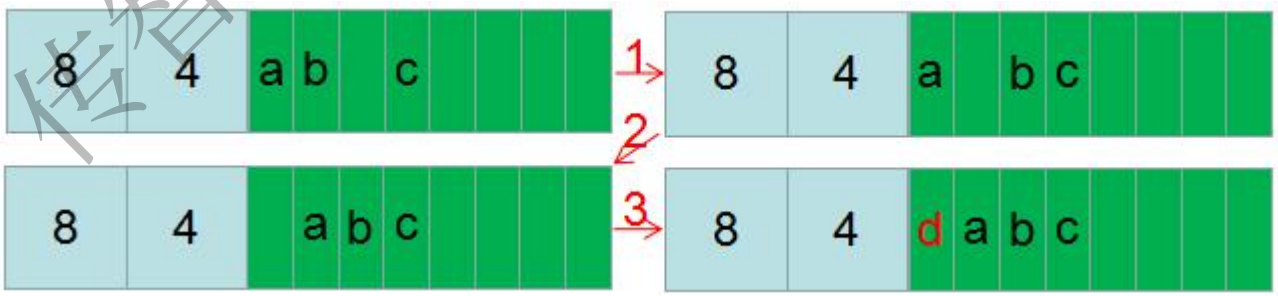
顺序表的“增”操作存在三种情况：插头，插尾，乱插

插头“增”：

增前：



增加数据过程：

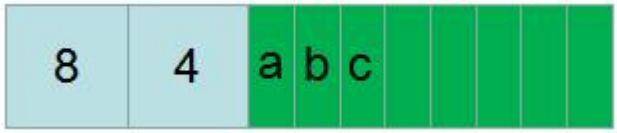


思考：

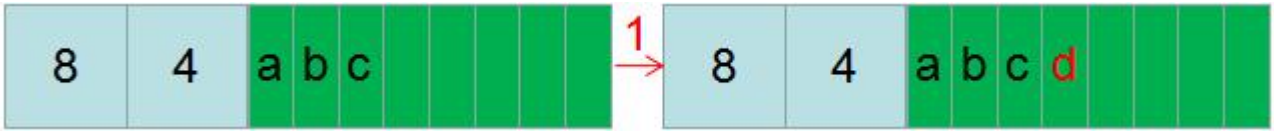
如果我要增加两个元素怎么办？

插尾“增”：

增前：

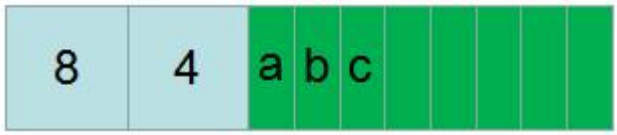


增加数据过程：

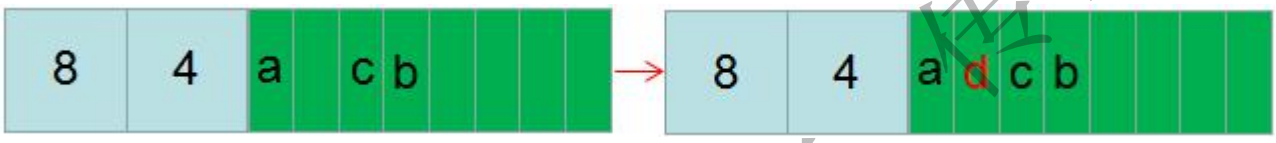


乱插“增”：

增前：



增加数据过程：



总结：

插头和插尾这两种方式，原来的数据顺序没有变化，所以我们称他们为“保序增加”

乱插这种方式，原来的数据顺序发生变化，所以我们称他们为“非保序增加”，而这种方法生产中，我们一般不用

时间复杂度：

对于插尾和乱插这两种情况，只需要一步就搞定，所以他们的时间复杂度为O(1)

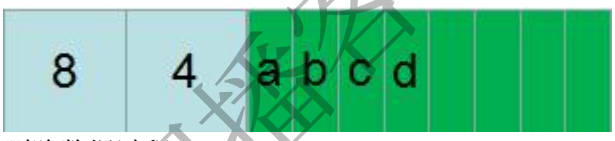
对于插头这种情况，有n个元素就需要n步，所以这种方法的时间复杂度为O(n)

“删”操作

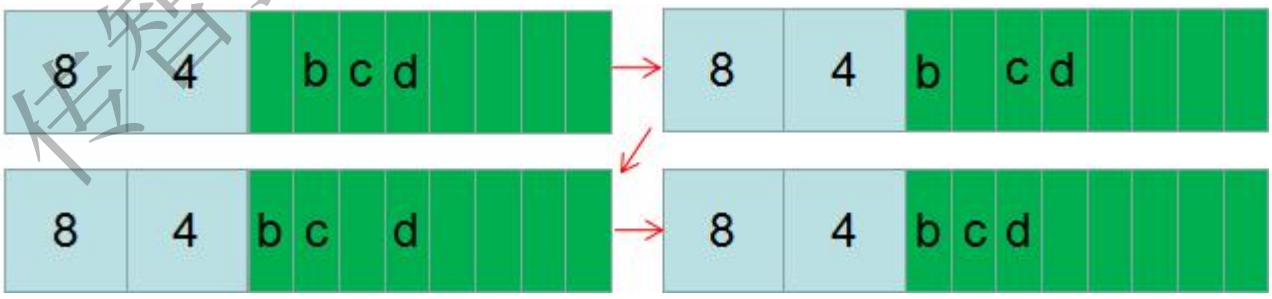
顺序表的“删”操作存在三种情况：减头，减尾，乱减

减头“删”

删前

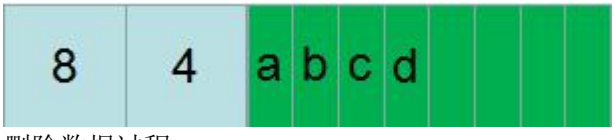


删除数据过程

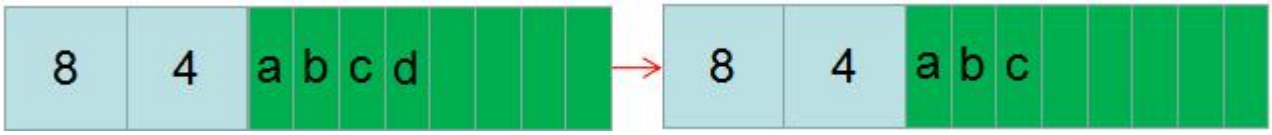


减尾“删”

删前：

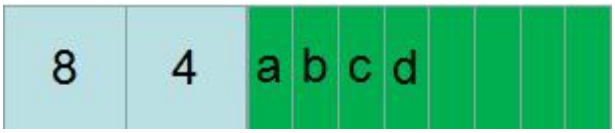


删除数据过程：

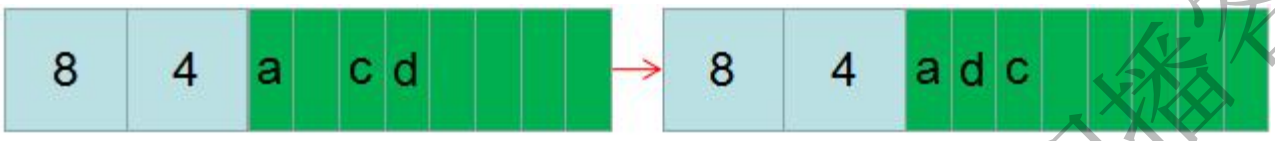


乱减“删”

删前



删除数据过程



总结：

减头和减尾这两种方式，原来的数据顺序没有变化，所以我们称他们为“保序删除”

乱减这种方式，原来的数据顺序发生变化，所以我们称他们为“非保序删除”，而这种方法生产中，我们一般不用

时间复杂度：

对于减尾和乱减这两种情况，只需要一步就搞定，所以他们的时间复杂度为O(1)

对于减头这种情况，有n个元素就需要n步，所以这种方法的时间复杂度为O(n)

3.3 单向链表

这一节我们从四个方面来学习单向链表相关的知识：

单向链表简介、python单向链表解析、单向链表实践、链表和顺序表

3.3.1 单向链表简介

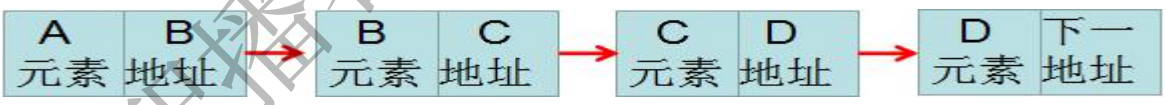
链表我们会从四个点去学习：

单向链表是什么、链表基本术语、单向链表特点、为什么用链表

单向链表是什么

单向链表（Linked list）也叫单链表，是链表中最简单的一种形式。

单向链表的基本单位叫“结点”，它包括两部分：元素存储区域(item)+下一元素地址(next)，多个结点组合起来就是一个单向链表



简单来说：单向链表就是"看星星一颗两颗三颗四颗连成线"

链表基本术语：

链表中的基本术语大致包括如下五个结点术语：

结点：元素+下一元素的地址 这两部分的组合。

例如：A元素 + 下一地址,即A结点，我们上个图中有4个结点：A结点、B结点、C结点、D结点

头结点：单向链表中的第一个结点

例如：链表中的第一个结点(A结点)

尾结点：单向链表中的最后一个结点

例如：链表中的最后一个结点(D结点)

前驱结点：链表中间的某个结点的前一个结点

例如：链表中的"C结点"的前驱结点就是"B结点"

后继结点：链表中间的某个结点的后一个结点

例如：链表中的"C结点"的后继结点就是"D结点"

单向链表特点：

只要找到头结点，那么整个链表就能找全

尾结点中的“下一地址”内容为空(None)

保存单向链表，只需要保存头结点地址即可

为什么用链表

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。

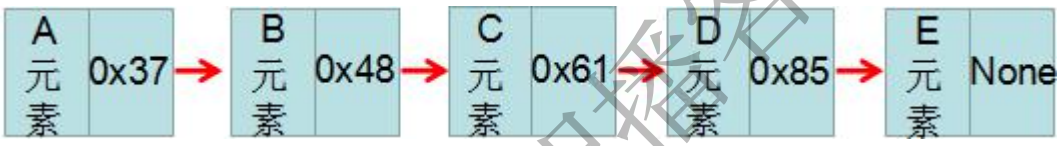
链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。

一句话：**链表灵活**

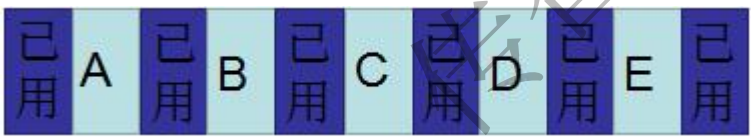
示例：一个存过数据的内存区域 [A,B,C,D,E]

在不连续的空间中存放一组数据，首先记住各个未使用空间的地址(浅蓝色)

借助于上面介绍的单向链表，构造一个单向链表



最终完成效果：



P = [A,B,C,D,E]

变量P指向链表的头节点（A）的位置，从P出发能找到表中的所有节点（B,C,D,E）。

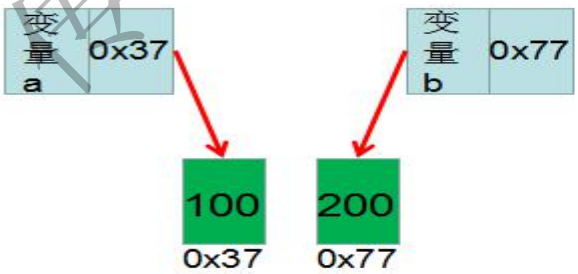
3.3.2 python 链表解析

接下来，从python的三个常见实践上来体验一下"链表"

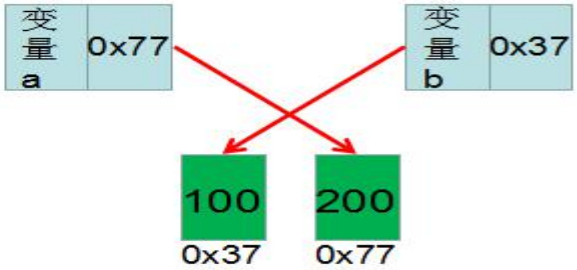
变量、函数、类

变量的本质：

变量 a = 100 和 b = 200 解析



变量替换：a,b = b,a

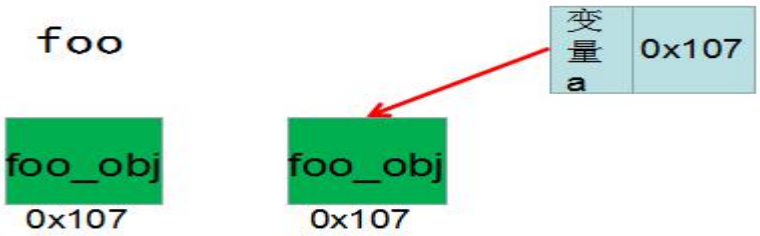


函数的本质

函数对象：

```
def foo():
    pass
```

实例 a = foo()

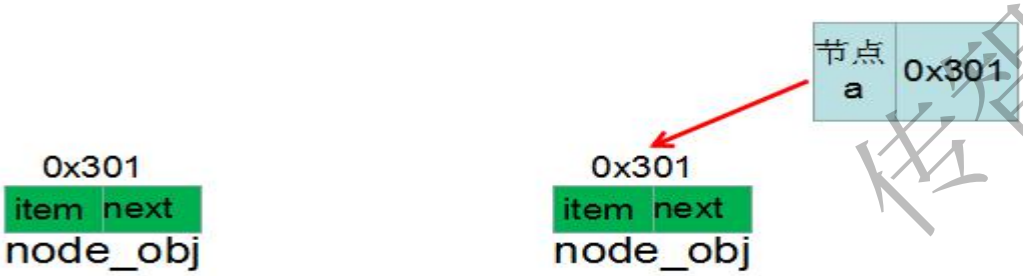


类的本质

类就是所有对象的共同属性，以我们的节点为例，所有的节点的结构都是一样的，所以我们就可以通过来构建一个类来描述节点的基本信息：

```
class Node(object):
    pass
```

类实例对象 node = Node()　　结点 a = node



链表类型获取结点中的具体内容：

A结点实例对象：A = node()　　获取A结点的内容：A.item

获取B结点的地址：A.next　　获取B结点的内容：A.next.item

根据上面的三种类型介绍，我们知道了，**等号 (=)** 其实**就是根据地址指向**下一个元素空间的**动作**

3.3.3 实践 之 基本属性

链表结点实践

关于链表结点，我们从两个方面来学习他的内容：

结点基本属性 和 基本属性值

我们知道，“结点”是链表中的一个基本单位，“结点”对象，有两个基本属性：内容(item)+链接地址(next)。

我们需要在“节点类”的构造方法(__init__)中的self中去定义他们，这样一旦我们构造一个结点，那么构造方法中的属性就会变成了结点对象的属性。

```
class BaseNode(object):
    def __init__(self):
        self.item
        self.next
```

通过我们上一节对于链表术语的介绍，我们知道类对象主要是用来存放数据的，存放的数据有以下特点：

1、结点的item属性的内容需要我们指定
2、创建的结点是一个单独的结点，所以它的next就是空
既然是结点的两个属性，那么我们可以使用self.item 和 self.next 来指定他们具体的内容

结点的代码实现：

```
class BaseNode(object):
    """单向链表的结点"""
    def __init__(self,item):
        # item 存放数据元素
        self.item = item
        # next 是下一个结点的地址
        self.next = None
```

验证代码

```
if __name__ == "__main__":
    node = BaseNode(100)
    print(node.item)
    print(node.next)
```

3.3.4 实践 之 操作分析

关于自定义链表的基本操作，我们从2个方面来说：
1、准备工作
2、哪些操作

1、准备工作

基本操作类：

我们创建一个SingleLinkList类，然后在这类中对BaseNode的基本属性进行各种各样的操作。

```
class SingleLinkList(object):
```

对于单向链表来说，只要获取到头结点，就相当于获取到了链表所有结点。如果要对链表进行操作，首先必须获取链表的头结点，所以在初始化链表的时候，必须有个方法自动找到头结点，一个构造方法实现它。

```
def __init__(self):
    self.head = ...
```

我们操作的对象有可能是一个空列表，那么self.head = None
我们操作的对象有可能是一个存在具体内容的列表，那么self.head = node
综合起来，传一个携带默认值的参数即可：

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

对于链表来说，最重要的就是头结点，因为重要，所以我们要保护起来不要别人看到，我们就可以用私有方法(__head)来实现

```
class SingleLinkList(object):
    def __init__(self,node=None):
        self.__head = node
```

2、哪些操作

根据我们之前对顺序表和list的学习，链表主要有以下几个方面的操作，也就是"类方法"

信息查看类：

链表是否为空：is_empty()

链表元素数量：length()

链表内容查看：travel()

链表内容搜索：search(item)

内容增加类：

插头增：add(item)

插尾增：append(item)

指定位置增：insert(pos,item)

内容删除类：

内容删除：remove(item)

3.3.5 实践 之 查看

判断是否链表为空

直接去获取链表头信息，如果头信息为空，那么肯定是空链表

```
    def is_empty(self):
        return self.__head is None
```

注释：

self.__head is None 就是一个判断表达式,正确返回True，错误返回False

测试：

```
if __name__ == "__main__":
    ll = SingleLinkList()
    print(ll.is_empty())
```

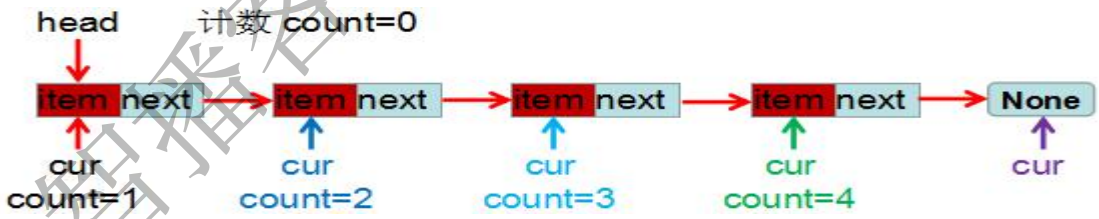
关键点：

头信息的内容判断

获取链表的长度

分析：

获取链表的长度，只要涉及到获取总数量，那么就要想到“循环遍历”+“计数器”



循环前：设定计数器count的初始值为0,定义一个表示当前结点的临时变量cur，指向链表的头结点

循环中：只要cur的内容不为空，而且cur不是尾结点，就后移cur到下一结点"cur = cur.next"，计数器就加1"count += 1"

循环后：如果cur的next属性值为None，cur所处的位置就是尾结点，这个时候的count就是链表的长度。

实践代码：

```
    def length(self):
        # 找到当前链表的头结点
        cur = self.__head
        # 设置计数器的初始值为 0
        count = 0
        # 查找尾结点
```

```
        while cur is not None:
            # 对计数进行递增
            count += 1
            # 移动当前的结点位置
            cur = cur.next
        # 输出最终计数
        return count
```

测试代码：

```
if __name__ == "__main__":
    ll = SingleLinkList()
    print(ll.length())
```

关键点：

遍历循环的终止条件：`while cur is not None`

计数器递增和结点移动：`count += 1`，`cur = cur.next`，

链表所有内容查看

类似于计数：

循环前：定义一个表示当前结点的临时变量cur，指向链表的头结点

循环中：只要cur的内容不为空，而且cur不是尾结点，打印当前结点的item，就后移cur到下一结点"cur = cur.next"

循环后：如果cur的next属性值为None，cur所处的位置就是尾结点，输出内容完毕。

代码实践

```
    def travel(self):
        # 找到当前链表的头信息
        cur = self.__head
        # 查找尾结点
        while cur is not None:
            # 输出每个结点的内容，设置分隔符为空格
            print(cur.item, end=" ")
            # 移动当前的结点位置
            cur = cur.next
        # 还原分隔符为换行符
        print("")
```

注释：

默认print的内容是以换行符为分隔符，因为我们要遍历内容，所以设置end=" "，这样所有内容在一行就显示出来，内容以空格为分隔符

最后的print表示，该实践结束后就恢复默认的分隔符

测试代码：

```
if __name__ == "__main__":
    ll = SingleLinkList()
    ll.travel()
```

关键点：

循环遍历的退出条件：`while cur is not None`

结点内容输出：`print(cur.item, end=" ")`

结点移动：`cur = cur.next`

搜索某个元素

类似我们的列表长度查看，但是增加了内容匹配的功能

循环前：定义一个表示当前结点的临时变量cur，指向链表的头结点

循环中：只要cur的内容不为空，而且cur不是尾结点，判断结点是否是搜索的item，是就返回True，不是的话就后移cur到下一结点

循环后：cur所处的位置就是尾结点，说明全部内容都没有匹配，返回没有匹配False。

实践代码

```
    def search(self,item):
        # 找到当前链表的头信息
        cur = self.__head
        # 判断当前结点不是尾结点
        while cur is not None:
            # 如果当前结点的内容就是我们要查找的，就返回 True
            if cur.item == item:
                return True
            # 移动当前结点位置到下一节点
            cur = cur.next
        # 如果整个循环都找不到我们要的内容，就返回 False
        return False
```

测试代码：

```
if __name__ == "__main__":
    ll = SingleLinkList()
    print(ll.search(5))
```

关键点：

循环遍历的退出条件：while cur is not None

内容的匹配：if cur.item == item

循环终止内容不匹配：return False

3.3.6 实践 之 增加

根据前面的分析，我们的增加存在三种方式：插头增、插尾增、指定位置增

插头增和插尾增涉及到两个结点的属性修改，而乱插增涉及到三个结点的属性修改

插头增

步骤：

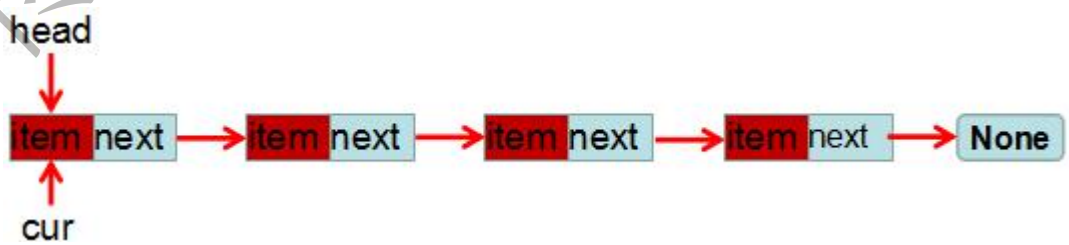
增加结点--找到位置--修改属性

根据我们之前分析的三步走：

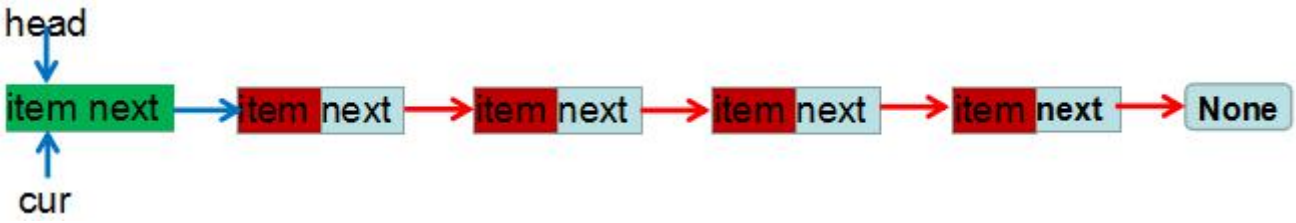
1、存储数据到一个新的结点中

```
node = BaseNode()
```

2、找到当前链表的头结点



3、将新的结点加入到头部，修改新结点的next为之前的头结点，并设定新的头结点为新结点



代码实践：

```
    def add(self, item):
        # 定义一个存储数据的新结点
        node = BaseNode(item)
        # 指定新结点的 next 属性为之前链表的头信息
        node.next = self.__head
        # 指定的当前链表的头结点为新结点
        self.__head = node
```

测试代码：

```
if __name__ == "__main__":
    ll = SingleLinkList()
    ll.add(1)
    print(ll.length())
    ll.add(2)
    print(ll.length())
    ll.travel()
```

关键点：

新结点的next指向旧的头结点：node.next = self.__head

链表头结点指向新结点：self.__head = node

插尾增

步骤：

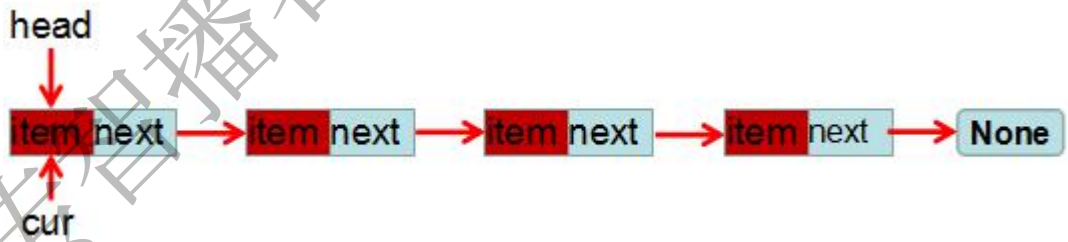
增加结点--找到位置--修改属性

我们需要在插尾增的基础上引入循环，找到尾结点。

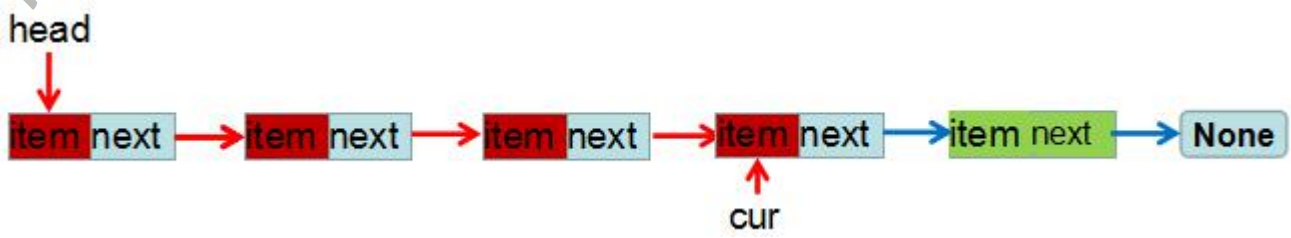
1、存储数据到一个新的结点中

node=BaseNode()

2、找到当前链表的头结点



3、找到尾结点后，定义尾结点的next指向新的node即可



代码实践：

```
    def append(self, item):
        # 定义一个存储数据的新结点
```

```
        node = BaseNode(item)
        # 如果当前列表为空，那么直接指定头信息为新结点即可
        if self.is_empty():
            self.__head = node
        # 如果当前列表不为空
        else:
            # 找到当前链表的头信息，然后找到尾结点
            cur = self.__head
            while cur.next is not None:
                cur = cur.next
            #  找到尾结点就退出循环，尾结点的 next 指向新结点即可
            cur.next = node
```

验证代码：

```
if __name__ == "__main__":
    ll = SingleLinkList()
    ll.add(1)
    ll.add(2)
    ll.append(5)
    print(ll.length())
    ll.travel()
```

关键点：

空链表增加结点，就是增加头结点：self.__head = node

查找尾结点：while cur.next is not None

尾结点next属性变化：cur.next = node

指定位置增

步骤：

增加结点--找到位置--判断位置--修改属性(两个结点)

对于指定位置增加，需要不但要引入循环，还要引入计数器，找到对应的插入位置

注意：

1、因为列表插入指定的位置是索引位置，而索引位置是从0开始的，所以我们在循环计数的时候，计数值要根据索引规律，就是(列表长度-1)

2、因为单向链表，只能从左向右找，所以我们在找插入位置的时候，一定要找到要插入位置的**前一个位置**

比如：我们要在下标3位置插入，我们应该将cur的位置移动到下标为2的位置。

3、关于首位置插入、末位置插入，我们可以根据指定的pos值来判断：

如果pos<=0,那么就直接调用add方法，

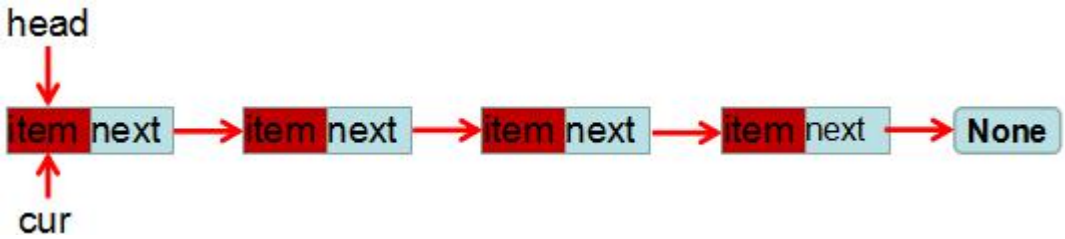
如果pos>=len(list_name),那么就调用append方法，在尾部增加元素即可。

我们**重点关心**指定位置插入，

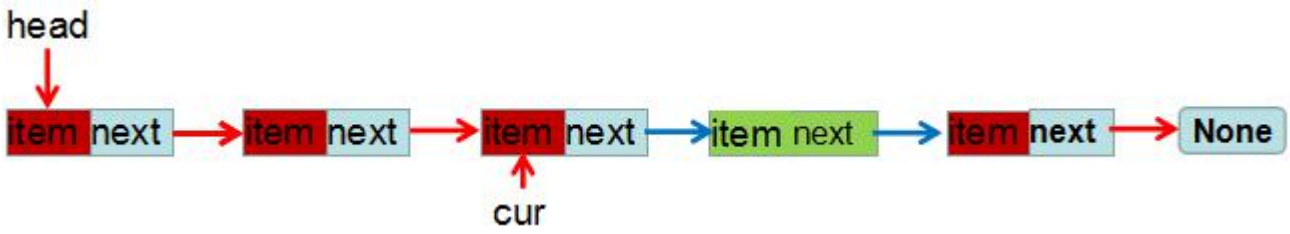
1、存储数据到一个新的结点中

node=BaseNode()

2、找到当前链表的头结点



3、找到要插入节点的前一个位置，然后修改当前结点的next值为新结点



代码实践：

```
    def insert(self, pos, item):
        # 定义一个存储数据的新结点
        node = BaseNode(item)
        # 头部添加内容
        if pos <= 0:
            self.add(item)
        # 在尾部添加元素
        elif pos >= self.length():
            self.append(item)
        # 在中间添加元素
        else:
            # 找到头信息，并且开始设置计数器初始值
            cur = self.__head
            count = 0
            # 找到要插入的位置的上一个位置
            while count < (pos - 1):
                count += 1
                cur = cur.next
            # 设置新结点的 next 属性为当前结点的下一个结点
            node.next = cur.next
            # 设置当前结点的 next 属性为新结点
            cur.next = node
```

测试代码：

```
if __name__ == "__main__":
    ll = SingleLinkList()
    ll.append(5)
    ll.travel()
    ll.insert(0,1)
    ll.travel()
    ll.insert(1,2)
    ll.travel()
    ll.insert(4, 6)
    ll.travel()
```

关键点：

指定位置插入，肯定包括头和尾，所以活用已有方法：`self.add(item)`、`self.append(item)`

插入位置的判断：`while count < (pos - 1)`

新结点的next指向cur的下一个结点：`node.next = cur.next`

cur结点的next指向新结点：`cur.next = node`

3.3.7 实践 之 删除

删除的操作就是指定元素操作，而元素所在的位置可能是头、中间、尾

删除的步骤：

判断链表是否为空，空则直接返回False，不空的话，继续下面步骤

判断内容是否是我们要删除的

是的话，判断内容的位置

头位置：直接切换当前链表的self.__head为下一个结点即可

尾位置：把要删除的结点B的上一个结点A的next属性指向B结点的next属性即可

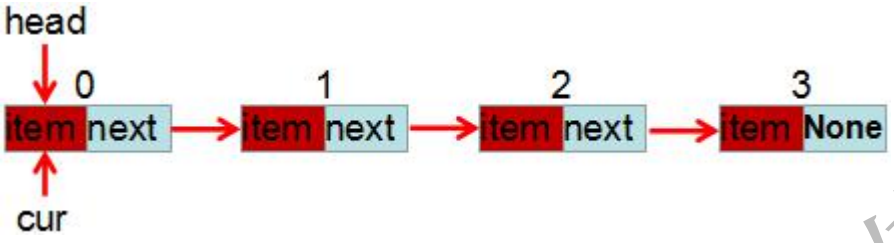
中间位置：把要删除的结点B的上一个结点A的next属性指向B结点的next属性即可

删除完成后，直接退出 return

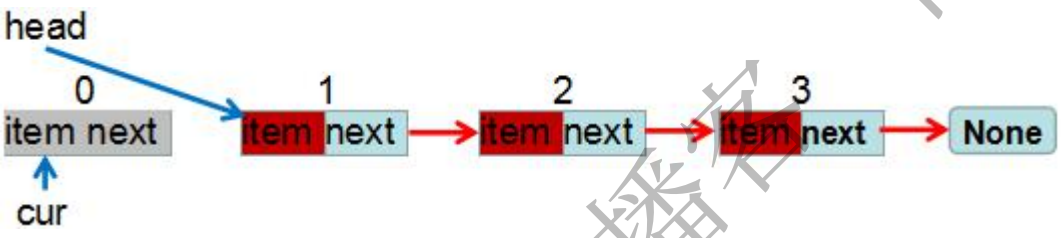
不是的话，直接返回False

异常情况：如果第一次找到的不是要删除的话，移动查找下一个

1、当前链表的情况



2、删除头情况



分析：

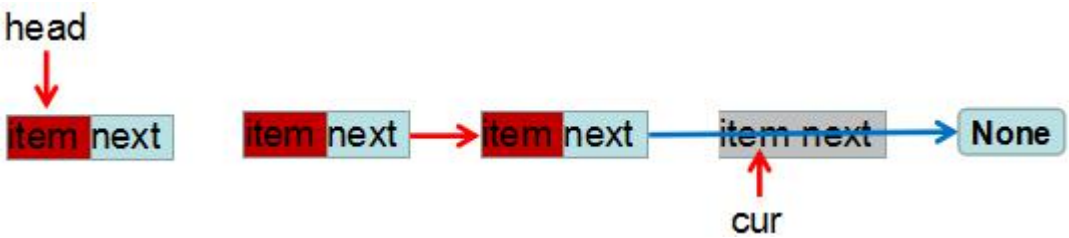
首先链表不为空的，然后再对当前结点的内容是否是要删除的，如果匹配到的话，接下来判断该内容的位置是否是头结点，是的话，在移动链表的头信息为当前结点cur的下一结点即可，然后退出循环

代码实现：

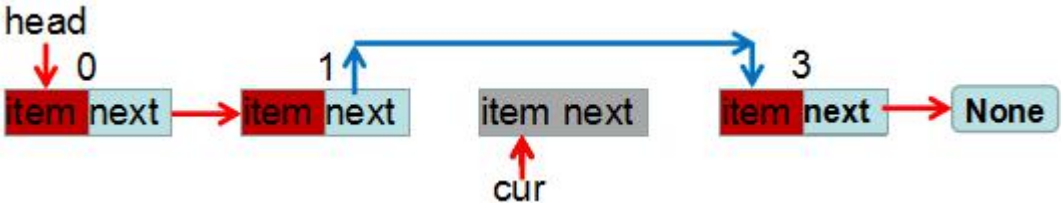
```
        # 找到当前链表的头信息
        cur = self.__head
        while cur is not None:
            # 如果当前结点的 item 就是要删除的内容
            if cur.item == item:
                # 如果当前结点 A 就是当前链表的头信息
                if cur == self.__head:
                    # 设置当前链表的头信息为当前结点 A 的下一个结点 B
                    self.__head = cur.next
                # 确定移除后，退出函数即可
                return
```

3、其他删除情况

删除尾



删除中间



其实删除尾结点也可以理解成删除中间节点，因为尾结点后面还有一个None

分析：

中间节点的删除就涉及到了三个结点之间的关系了，目前我们的标签只能表示两个结点的关系，所以我们就需要引入一个pre标签表示前一个结点，这样我们就要考虑两个方面：

1、对于头结点来说：pre就是一个空，

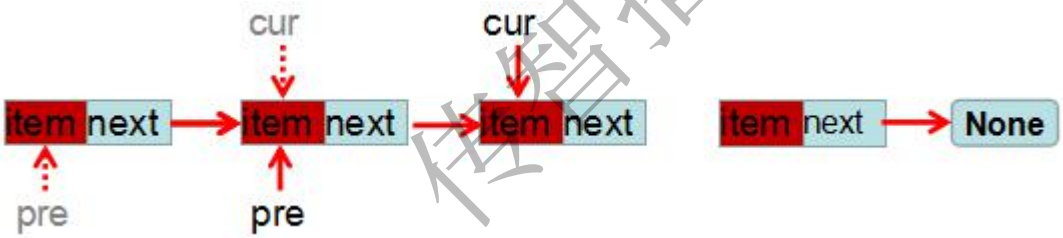
2、删除一个中间节点，那么就将1结点pre的next属性指向为3结点(即cur.next)即可

代码实现：

```
# 自定义一个当前结点的上一个结点 pre,默认值是 None
pre = None
# 当前链表不为空
while cur is not None:
...
   # 如果当前结点不是头结点
   else:
       # 将当前结点 B 的上一结点 A 的 next 属性设定为下一个结点 C
       pre.next = cur.next
```

4、其他情况

如果我们cur标签所在节点的内容不是我们要找的内容怎么办？



把pre标签移动到cur标签，然后把cur移动到cur.next标签即可，然后接着循环

代码实现：

```
# 如果当前结点的 item 就是要删除的内容
if cur.item == item:
    ...
# 如果不是要找的删除元素，移动 cur 的标签，继续进行判断
pre = cur
# 将当前结点 A 的 next 属性为下一结点 C
cur = cur.next
```

整体代码实践：

```
def remove(self,item):
    # 找到当前链表的头信息
    cur = self.__head
    # 自定义一个当前结点的上一个结点 pre,默认值是 None
    pre = None
    # 当前链表不为空
    while cur is not None:
        # 如果当前结点的 item 就是要删除的内容
        if cur.item == item:
```

```
                # 如果当前结点 A 就是当前链表的头信息
                if cur == self.__head:
                    # 设置当前链表的头信息为当前结点 A 的下一个结点 B
                    self.__head = cur.next
                # 如果当前结点不是头结点
                else:
                    # 将当前结点 B 的上一结点 A 的 next 属性设定为下一个结点 C
                    pre.next = cur.next
                # 确定移除后，退出函数即可
                return
            # 如果不是要找的删除元素，移动 cur 的标签，继续进行判断
            pre = cur
            # 将当前结点 A 的 next 属性为下一结点 C
            cur = cur.next
```

测试代码：

```
if __name__ == "__main__":
    ll = SingleLinkList()
    ll.append(1)
    ll.append(2)
    ll.append(3)
    ll.append(4)
    ll.append(5)
    ll.travel()
    ll.remove(1)
    ll.travel()
    ll.remove(5)
    ll.travel()
    ll.remove(3)
    ll.travel()
```

关键点：

头删除：

判断的顺序：当前链表不为空-->删除的内容匹配-->内容所在位置

头结点的指向：self.__head = cur.next

删除节点后，退出循环：return

尾删除：

单向链表只能前找后，当前结点的上一结点:pre = None

删除节点后，前一结点next属性变化：pre.next = cur.next

内容不匹配：

内容不匹配，当前节点和前一节点同时后移：pre = cur 、 cur = cur.next

3.3.8 链表 vs 顺序表

结构上：

链表失去了顺序表随机读取的优点，同时链表由于增加了下一结点的跳转地址，空间开销比较大，但对存储空间的使用要相对灵活。

成本上：

链表与顺序表的各种操作复杂度如下所示：

操作	链表	顺序表
访问元素	O(n)	O(1)
在头部插入/删除	O(1)	O(n)

在尾部插入/删除	O(n)	O(1)
在中间插入/删除	O(n)	O(n)

注意：

对于**最后一项**,虽然在表面看起来链表和顺序表的时间复杂度都是 O(n)，但是本质上是完全不同的操作。

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)，所以最终才是O(n)。

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，所以才是O(n)。

3.4 双向链表

双向链表我们从两个方面来学习：

双向链表简介和双向链表实践

3.4.1 双向链表简介

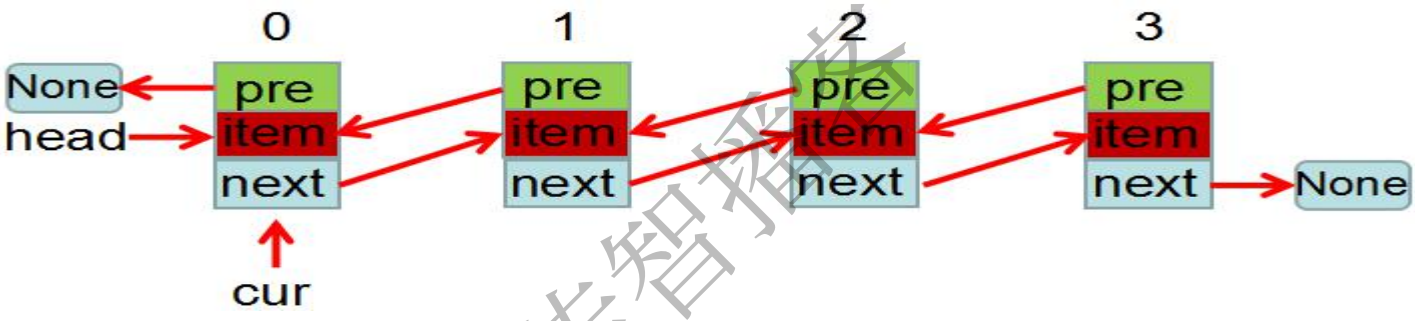
双向链表定义：

“双向链表”相对于“单向链表”是一种更复杂的链表。原因在于，双向链表的每个节点有两个链接地址：

next：指向下一个节点，当此节点为最后一个节点时，指向空值；

pre： 指向前一个节点，当此节点为第一个节点时，指向空值

双向链表示例：



3.4.2 实践 之 基本属性

根据我们对双向链表的结构学习以及单向链表的基本属性学习，我们知道双向链表的结点属性，无非就是多了一个pre的基本属性

代码实践：

```
class BaseNode(object):
    """双向链表的结点"""
    def __init__(self,item):
        # 定义上一个结点的地址
        self.pre = None
        # item 存放数据元素
        self.item = item
        # next 是下一个结点的地址
        self.next = None
```

验证代码

```
if __name__ == "__main__":
    node = BaseNode(100)
    print(node.pre)
    print(node.item)
    print(node.next)
```

关键点：

前驱结点的地址：self.pre = None

3.4.3 实践 之 操作分析

操作分析我们从两个方面来说：

1、准备工作

2、基本操作

准备工作

关于链表的基本操作，我们肯定需要知道当前链表的头信息，所以我们需要定义一个链表的头信息

代码实践：

```
class DoubleLinkList(object):
    # 定义操作的基本属性
    def __init__(self, node=None):
        # 确定当前链表的头信息
        self.__head = node
```

双链操作

我们知道双向链表只不多是在单向链表的基础上多了向前的连接信息，所以双向链表的操作和单向链接的操作基本一样。

所以双向链表的操作也分为三个大类：

信息查看类：

链表是否为空：is_empty()

链表元素数量：length()

链表内容查看：travel()

链表内容搜索：search(item)

内容增加类：

插头增：add(item)

插尾增：append(item)

指定位置增：insert(pos,item)

内容删除类：

内容删除： remove(item)

3.4.4 实践 之 增加

关于增加结点的操作，无非就是在链表的头部增加、指定位置增加、尾部增加三种样式。

插头增和插尾增涉及到两个结点的属性修改，而乱插增涉及到三个结点的属性修改

头部增加

头部增加的步骤跟单向链表的步骤一致：

增加结点 -- 找到位置 -- 更改属性

区别就在第三步：

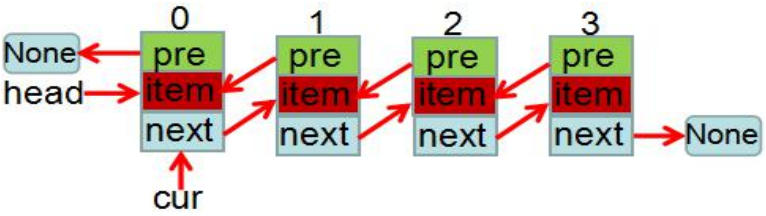
不管链表是否为空，头结点的标识指向新结点即可，但是一旦链表不为空，那么就需要调整老结点的pre属性为新结点了

根据我们之前分析的三步走：

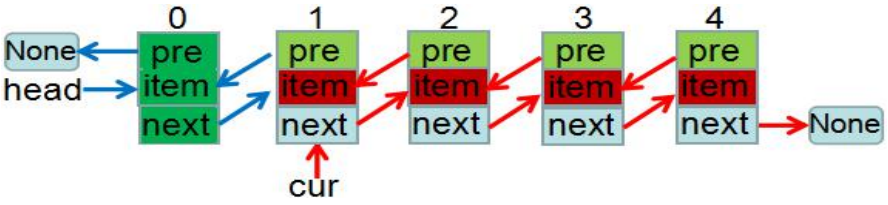
1、存储数据到一个新的结点中

node = BaseNode()

2、找到当前链表的头结点



3、将新的结点A加入到头部，修改新结点A的next为之前的头结点，设定新的头结点为新结点A，修改老结点B的pre为新结点A即可



实践代码：

```
    def add(self, item):
        # 定义一个存储数据的新结点
        node = BaseNode(item)
        # 指定新结点的 next 属性为之前链表的头信息
        node.next = self.__head
        # 指定的当前链表的头信息为新结点
        self.__head = node
        # 如果链表不为空
        if node.next:
            # 修改老结点的 pre 属性为新结点
            node.next.pre = node
```

测试代码：

```
if __name__ == "__main__":
    ll = DoubleLinkList()
    ll.add(1)
    ll.travel()
```

关键点：

新结点的next指向旧的头结点：node.next = self.__head

链表头结点指向新结点：self.__head = node

旧结点的pre属性变化：node.next.pre = node

尾部增加

尾部增加的步骤跟单向链表的步骤一致，

增加结点 -- 找到位置 -- 更改属性

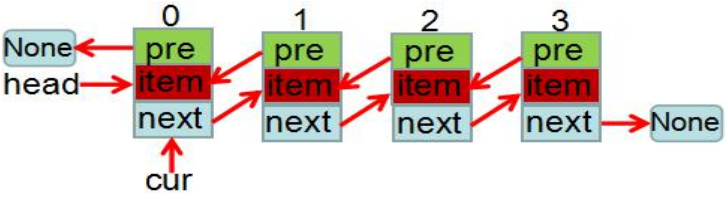
区别就在第三步修改的属性多了一个pre属性

根据我们之前分析的三步走：

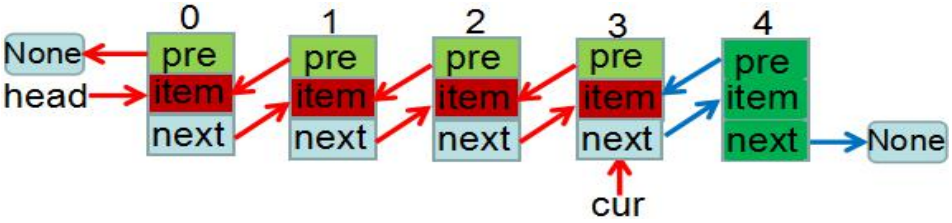
1、存储数据到一个新的结点中

```
node = BaseNode()
```

2、找到当前链表的头结点



3、将新的结点A加入到尾部，修改新结点A的pre为当前的结点B，设定当前结点B的next为新结点A



实践代码：

```
    def append(self, item):
        # 定义一个存储数据的新结点
        node = BaseNode(item)
        # 如果当前列表为空，那么直接指定头信息为新结点即可
        if self.is_empty():
            self.__head = node
        # 如果当前列表不为空
        else:
            # 找到当前链表的头信息，然后找到尾结点
            cur = self.__head
            while cur.next is not None:
                cur = cur.next
            # 指定新结点的 pre 属性为当前结点
            node.pre = cur
            # 指定当前结点的 next 属性为新结点
            cur.next = node
```

测试代码：

```
if __name__ == "__main__":
    ll = DoubleLinkList()
    ll.add(1)
    ll.append(7)
    ll.travel()
```

关键点：

空链表增加结点，就是增加头结点：self.__head = node

查找尾结点：while cur.next is not None

新结点的pre属性变化：node.pre = cur

尾结点next属性变化：cur.next = node

指定位置增加

指定位置增加的步骤跟单向链表的步骤一致：

增加结点--找到位置--判断位置--修改属性

区别就在第3和4步

对于指定位置增加，需要不但要引入循环，还要引入计数器，找到对应的插入位置

注意：

1、因为列表插入指定的位置是索引位置，而索引位置是从0开始的，所以我们在循环计数的时候，计数值

要根据索引规律，就是列表长度-1

2、因为双向链表，可以两方向查找，所以我们在找插入位置的时候，直接找到对应位置即可

比如：我们要在下标3位置插入，我们应该将cur的位置移动到下标为3的位置。

3、关于首位置插入、末位置插入，我们可以根据指定的pos值来判断：

如果pos<=0,那么就直接调用add方法，

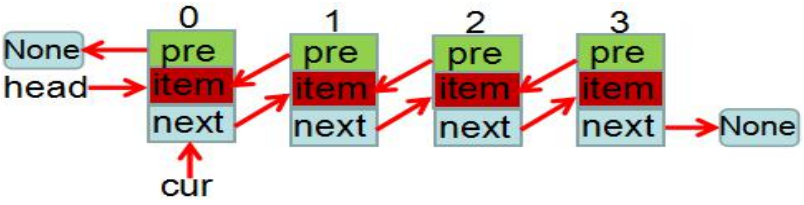
如果pos>=len(list_name),那么就调用append方法，在尾部增加元素即可。

根据我们之前分析的三步走：

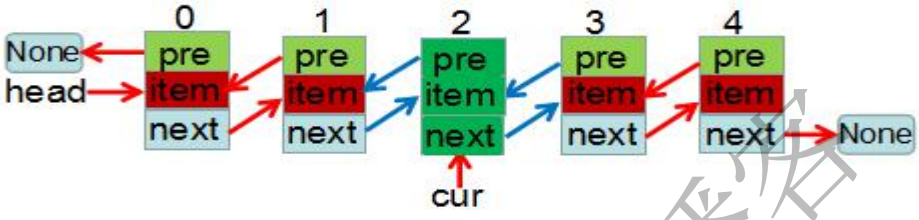
1、存储数据到一个新的结点中

```
node = BaseNode()
```

2、找到当前链表的头结点



3、将新的结点加入到下标为2的位置，修改新结点2的pre为结点1，next属性为结点3,同时修改结点3的pre为新结点2，修改结点1的next属性为新结点2



代码实践：

```
    def insert(self, pos, item):
        # 定义一个存储数据的新结点
        node = BaseNode(item)
        # 头部添加内容
        if pos <= 0:
            self.add(item)
        # 在尾部添加元素
        elif pos >= self.length()-1:
            self.append(item)
        # 在中间添加元素
        else:
            # 找到头信息，并且开始设置计数器初始值
            cur = self.__head
            count = 0
            # 找到要插入的位置
            while count < pos:
                count += 1
                cur = cur.next
            # 设置新结点的 next 属性为当前结点
            node.next = cur
            # 设置新结点的 pre 属性为当前结点之前的上一个结点
            node.pre = cur.pre
            # 设置当前结点的上一个结点的 next 属性为新结点
            cur.pre.next = node
```

```
            # 设置当前结点的 pre 为新结点
            cur.pre = node
```

测试代码：

```
if __name__ == "__main__":
    ll = DoubleLinkList()
    ll.add(1)
    ll.append(3)
    ll.insert(1,4)
    ll.travel()
```

关键点：

指定位置插入，肯定包括头和尾，所以活用已有方法：self.add(item)、self.append(item)

插入位置的判断：while count < pos

新结点的next指向cur：node.next = cur

新结点的next指向cur的上一个结点：node.pre = cur.pre

cur结点上一结点的next指向新结点：cur.pre.next = node

cur结点上一结点的pre指向新结点：cur.pre = node

3.4.5 实践 之 删除

关于增加结点的操作，无非就是在链表的头部删除、指定位置删除、尾部删除三种情况。

删除的步骤：

判断链表是否为空，空则直接返回False，不空的话，继续下面步骤

判断内容是否是我们要删除的

是的话，判断内容的位置

头位置：...

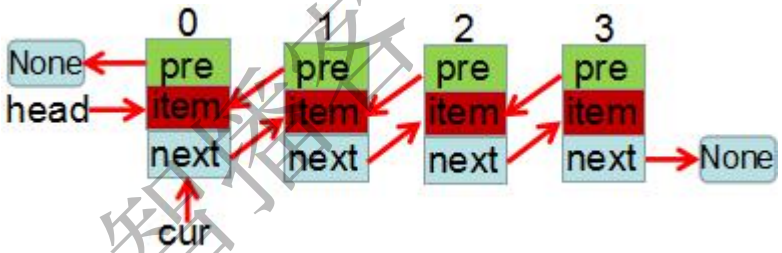
尾位置：...

中间位置：...

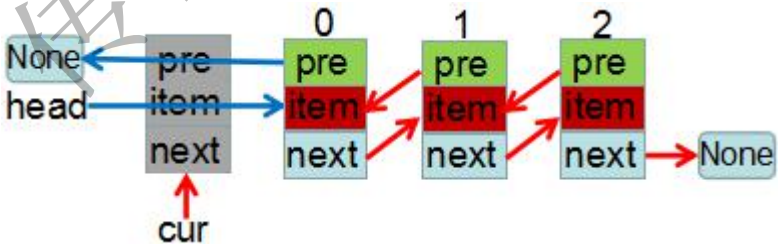
删除完成后，直接返回 return

不是的话，直接返回False

1、当前链表效果：



2、头部删除：



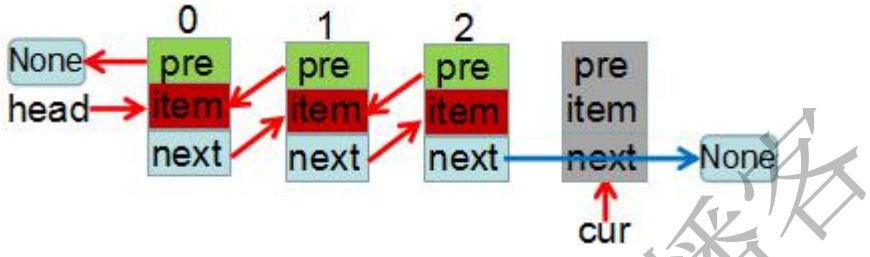
分析：

首先链表不为空的，然后再对当前结点的内容是否是要删除的，如果匹配到的话，接下来判断该内容的位置是否是头结点，是的话，在移动链表的头信息为当前结点cur的下一结点即可，如果cur.next结点存在的话，将cur下一结点的pre指向None，然后退出循环

代码实现：

```
    def remove(self,item):
        # 找到当前链表的头信息
        cur = self.__head
        # 遍历所有结点
        while cur is not None:
            # 如果当前结点的 item 就是要删除的内容
            if cur.item == item:
                # 如果当前结点 A 就是链表的首结点
                if cur == self.__head:
                    # 设置当前链表的头信息为当前结点 A 的下一个结点 B
                    self.__head = cur.next
                    # 链表不是只有一个结点，新的头结点的 pre 属性指向 None
                    if cur.next:
                        self.__head.pre = None
                    # 删除完毕后，就退出
                    return
```

3、尾部删除：



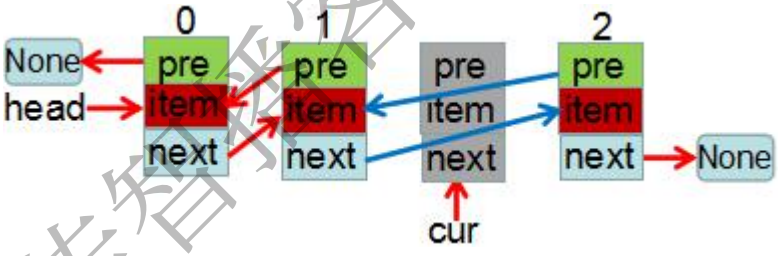
分析：

对于尾部删除来说，只需要将上一个结点的next属性定义为none就行了

代码实现：

```
                # 如果当前结点不是头结点
                else:
                    # 将当前结点 B 的上一结点 A 的 next 属性设定为下一个结点 C
                    cur.pre.next = cur.next
```

4、中间删除：



分析：

对于中间删除，需要定义前一个结点和后一个节点的关系：

1） 1结点的next属性定义为cur结点的下一节点2的内容

2） 2结点的pre属性定义为cur结点的上一结点1的内容

注意：后面一条想要执行，必须有个前提--2结点存在

代码实现：

```
        while cur is not None:
            ...
                if cur == self.__head:
                ...
                else:
```

```
                ...
                # 如果当前结点不是尾结点
                if cur.next:
                    cur.next.pre = cur.pre
```

5、其他情况

刚才我们分析的场景都是一下子就找到了我们要删除的内容，如果查找的内容不是我们要删除的内容，方法同单链表一样，只需要把当前结点cur移动到下一节点cur即可

代码实现

```
            # 将当前结点 A 的 next 属性为下一结点 C
            cur = cur.next
```

最终实践代码：

```
    def remove(self,item):
        # 找到当前链表的头信息
        cur = self.__head
        # 遍历所有结点
        while cur is not None:
            # 如果当前结点的 item 就是要删除的内容
            if cur.item == item:
                # 如果当前结点 A 就是链表的首结点
                if cur == self.__head:
                    # 设置当前链表的头信息为当前结点 A 的下一个结点 B
                    self.__head = cur.next
                    # 链表不是只有一个结点
                    if cur.next:
                        self.__head.pre = None
                # 如果当前结点不是头结点
                else:
                    # 将当前结点 B 的上一结点 A 的 next 属性设定为下一个结点 C
                    cur.pre.next = cur.next
                    # 如果当前结点不是尾结点
                    if cur.next:
                        cur.next.pre = cur.pre
                return
            # 将当前结点 A 的 next 属性为下一结点 C
            cur = cur.next
```

测试代码：

```
if __name__ == "__main__":
    ll = DoubleLinkList()
    ll.add(1)
    ll.append(3)
    ll.travel()
    ll.remove(3)
    ll.travel()
```

关键点：

头删除：

判断的顺序：当前链表不为空-->删除的内容匹配-->内容所在位置

头结点的指向：self.__head = cur.next

新头结点的pre指向：self.__head.pre = None

删除节点后，退出循环：return

尾删除：

删除节点后，前一结点next属性变化：cur.pre.next = cur.next

中间删除：

除了尾删除的一点，还有cur下一个节点的pre属性：cur.next.pre = cur.pre

内容不匹配：

当前节点后移即可：cur = cur.next

3.5 单向循环链表

单向循环链表我们从两个方面来学习：

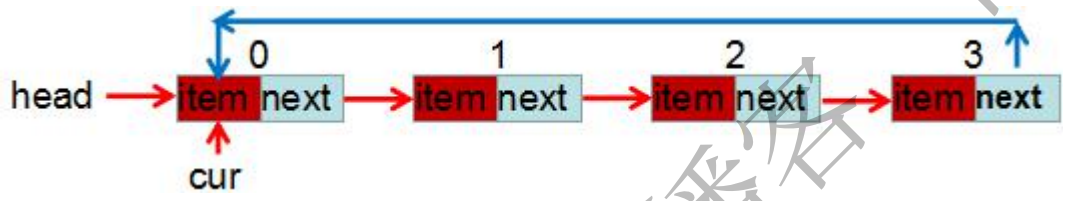
单向循环链表简介和单向循环链表的实践

3.5.1 单向循环链表简介

单向循环链表的定义

单链表的一个特殊形式就是单向循环链表，链表中最后一个节点的next不再为None，而是指向当前链表的头节点

单向循环链表的结构



3.5.2 实践 之 基本属性

因为单向循环链表是单向链表的一种特殊形式，所以他们的基本属性是一致的。

结点的代码实现：

```
class BaseNode(object):
    """单向链表的结点"""
    def __init__(self,item):
        # item 存放数据元素
        self.item = item
        # next 是下一个结点的地址
        self.next = None
```

验证代码

```
if __name__ == "__main__":
    node = BaseNode(100)
    print(node.item)
    print(node.next)
```

3.5.3 实践 之 操作分析

操作分析我们从两个方面来说：

1、准备工作

2、常见操作

准备工作

关于单向循环链表的基本操作，我们肯定需要知道当前链表的头信息，所以我们需要定义一个链表的头信息

代码实践：

```
class CycleSingleLinkList(object):
    # 定义操作的基本属性
    def __init__(self, node=None):
        # 确定当前链表的头信息
        self.__head = node
```

常见操作：

单向循环链表的操作也是三大类：

信息查看类：

链表是否为空：is_empty()

链表元素数量：length()

链表内容查看：travel()

链表内容搜索：search(item)

内容增加类：

插头增：add(item)

插尾增：append(item)

乱插增：insert(pos,item)

内容删除类：

内容删除： remove(item)

因为单向循环链表的最后一个结点的next指向了首结点的地址，所以只要涉及到循环遍历的方法，全部都需要单独设计。

3.5.4 实践 之 查看

判断链表是否为空

因为这个方法，主要是针对于首结点的内容是否为空来判断的，对于单向链表和单向循环链表没有任何区别，所以is_empty()不需要做任何变动

判断链表数量

判断条件：

单向链表来判断链表长度，是根据最后一个结点的next属性是否为None来判断的，但是对于单向循环链表来说，最后一个结点的next属性是当前链表的头结点，也就是我们常说的 self.__head。

计数器：

原来的单向链表的最后结点指向的是None，所以我们count计数从0开始没有问题，但是对于单向循环链表来说，最后一个结点的next属性指向的是头结点，所以判断的时候，就少算了一位，所以对于单向循环链表来说，计数的初始值应该是1.

空链表：

因为单向链表循环计数的时候，无论是否是空链表都能正常执行，但是对于单向循环链表来说，空链表就没有首结点内容，所以我们可以结合is_empty()函数来判断链表为空。

关键点：

我们的循环条件一旦满足，也就是找到了尾结点，但是尾结点并没有执行循环体内的语句。需要我们来单独操作它。

所以判断链表长度的方法内容如下：

代码实践：

```
    def length(self):
        # 如果当前链表是空链表
        if self.is_empty():
            return 0

        # 设置计数器的初始值为1
        count = 1
        # 找到当前链表的头位置
        cur = self.__head

        # 查找尾结点
        while cur.next is not self.__head:
            # 对计数进行递增
            count += 1
            # 移动当前的结点位置
            cur = cur.next
        # 输出最终计数
        return count
```

代码测试：

```
if __name__ == "__main__":
    ll = CycleSingleLinkList()
    print(ll.length())
    ll = CycleSingleLinkList(100)
    print(ll.length())
```

关键点：

遍历循环的终止条件：`while cur.next is not self.__head`

计数器递增和结点移动：`count += 1，cur = cur.next`

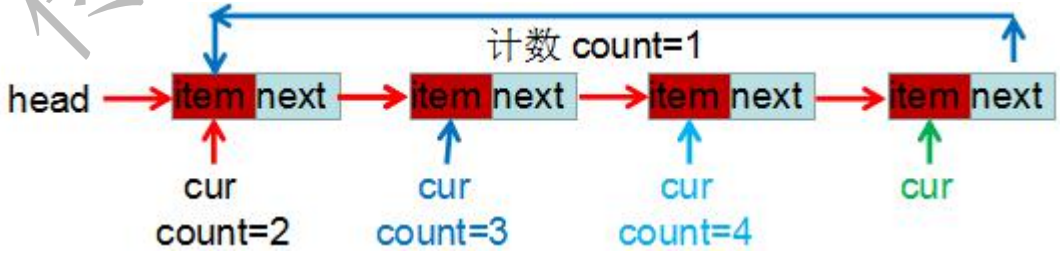
单向循环链表遍历内容

类似于上面显示链表元素数量的分析过程

1、循环条件的改变 None 改成了 self.__head

2、循环退出，将尾结点的item输出即可

3、当前链表是否为空



代码实践：

```
    def travel(self):
        # 如果当前链表是空链表
        if self.is_empty():
```

```
            print("")
            return

        # 找到当前链表的头信息
        cur = self.__head
        # 查找尾结点
        while cur.next is not self.__head:
            # 输出每个结点的内容，设置分隔符为空格
            print(cur.item, end=" ")
            # 移动当前的结点位置
            cur = cur.next
        #  从循环退出，cur 指向的是尾结点，所以我们之间打印 cur 的 item 信息即可
        print(cur.item)
```

测试代码：

```
if __name__ == "__main__":
    ll = CycleSingleLinkList()
    ll.travel()
```

关键点：

循环遍历的退出条件：`while cur.next is not self.__head`

结点内容输出：`print(cur.item, end=" ")`

结点移动：`cur = cur.next`

退出循环，输出尾结点的内容：`print(cur.item)`

信息查找情况

代码实践：

```
    def search(self,item):
        # 如果链表是空链表
        if self.is_empty():
            return False
        # 找到当前链表的头信息
        cur = self.__head
        # 判断当前结点不是尾结点
        while cur.next is not self.__head:
            # 如果当前结点的内容就是我们要查找的，就返回 True
            if cur.item == item:
                return True
            # 移动当前结点位置到下一节点
            cur = cur.next
        # 对尾结点进行判断
        if cur.item == item:
            return True
        # 如果整个循环都找不到我们要的内容，就返回 False
        return False
```

测试代码

```
if __name__ == "__main__":
    ll = SingleLinkList()
    print(ll.search(2))
    ll.add(2)
    ll.add(4)
```

```
        print(ll.search(2))
```

关键点：

循环遍历的退出条件：while cur is not self.__head

内容的匹配：if cur.item == item

循环终止内容不匹配：return False

3.5.5 实践 之 增加

单向循环链表增加内容的操作，还是头部增加、指定位置增加、尾部增加三种样式

关于单向循环链表的增加操作，虽然跟单向链表的增加类似，但是尾结点的next属性设置成了新的结点，那么我们在**将新结点增加到链表内部之前，必须要先找到尾结点**。

头部增加

步骤： 创建结点--找到位置--找尾结点--更改属性

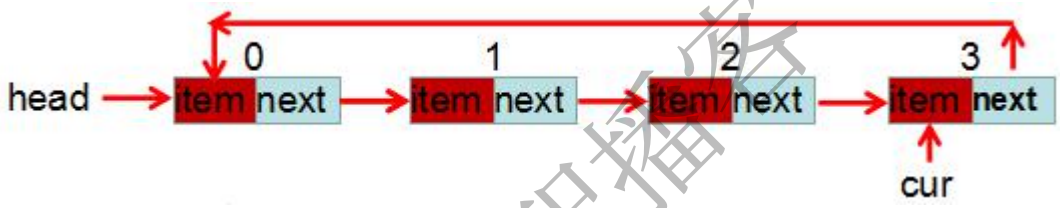
对于头部增加的操作，不但要考虑刚才我们说明的一点，还要判断是否存在空列表，因为空列表没有next的属性。

根据我们之前分析的三步走：

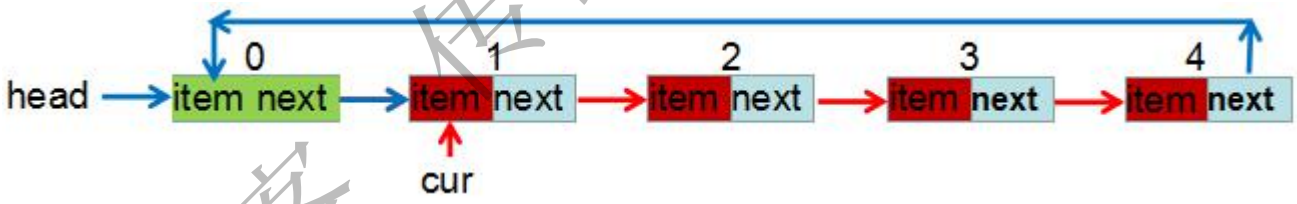
1、存储数据到一个新的结点中

node=BaseNode()

2、定位当前链表的头结点并且找到尾结点



3、将新的结点加入到头部，修改新结点A的next为之前的头结点B，并设定新的头结点为新结点A，修改尾结点N的next属性为新结点A



4、如果当前链表就是空链表，那么直接将头标签定义到新结点，然后把结点的next属性定义成头结点即可

代码实践：

```
        def add(self, item):
            # 定义一个存储数据的新结点
            node = BaseNode(item)
            # 对于空链表，我们需要单独来操作
            if self.is_empty():
                self.__head = node
                node.next = self.__head

            # 对于非空链表，我们要先定位首结点
            cur = self.__head
            # 查找尾结点，退出循环，表示 cur 处在尾结点
            while cur.next is not self.__head:
                cur = cur.next
            # 指定当前尾结点的 next 属性为新结点
```

```
            cur.next = node
            # 指定新结点的 next 属性为之前链表的头结点
            node.next = self.__head
            # 指定的当前链表的头结点为新结点
            self.__head = node
```

测试代码：

```
    if __name__ == "__main__":
        ll = CycleSingleLinkList()
        ll.add(1)
        ll.travel()
        ll.add(4)
        ll.travel()
```

关键点：

新结点的next指向旧的头结点：node.next = self.__head

链表头结点指向新结点：self.__head = node

非空链表的尾结点查找：while cur.next is not self.__head

尾结点的next属性变化：cur.next = node

尾部增加

对于尾部增加的效果，跟之前的类似：

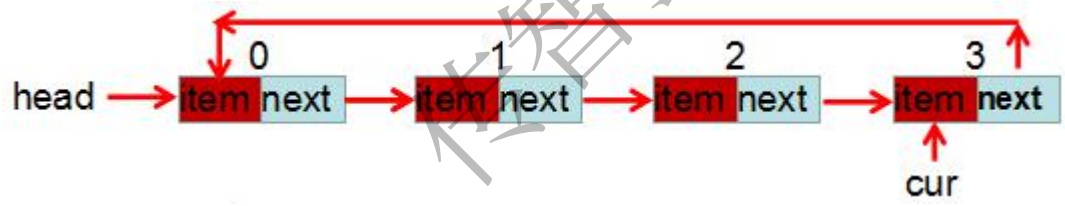
新增结点--定位首结点和尾结点--更改属性

根据我们之前分析的三步走：

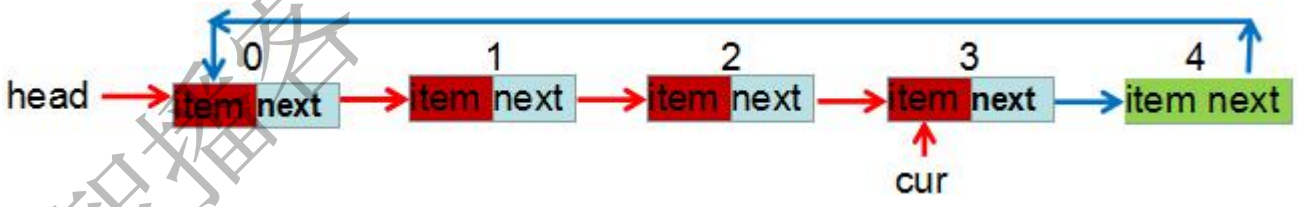
1、存储数据到一个新的结点中

node=BaseNode()

2、定位当前链表的头结点并且找到尾结点



3、将新的结点加入到尾部，设定尾结点N的next属性为新结点A，修改新结点A的next为当前的头结点B



代码实践：

```
    def append(self, item):
            # 定义一个存储数据的新结点
            node = BaseNode(item)
            # 如果当前列表为空，那么直接指定头信息为新结点即可
            if self.is_empty():
                self.__head = node
                node.next = self.__head
            # 如果当前列表不为空
            # 找到当前链表的头信息，然后找到尾结点
            cur = self.__head
            while cur.next is not self.__head:
```

```
            cur = cur.next
        #  找到尾结点就退出循环，尾结点的 next 指向新结点即可
        cur.next = node
        # 将新结点的 next 指向当前链表的首结点
        node.next = self.__head
```

代码测试：

```
if __name__ == "__main__":
    ll = CycleSingleLinkList()
    ll.add(1)
    ll.append(3)
    ll.travel()
```

关键点：

新结点的next指向旧的头结点：node.next = self.__head

非空链表的尾结点查找：while cur.next is not self.__head

尾结点的next属性变化：cur.next = node

指定位置增加

步骤：

增加结点--找到位置--判断位置--修改属性

对于指定位置增加，需要不但要引入循环，还要引入计数器，找到对应的插入位置

注意：

1、因为列表插入指定的位置是索引位置，而索引位置是从0开始的，所以我们在循环计数的时候，计数值要根据索引规律，就是列表长度-1

2、因为单向链表，只能从左向右找，所以我们在找插入位置的时候，一定要找到要插入位置的前一个位置

比如：我们要在下标3位置插入，我们应该将cur的位置移动到下标为2的位置。

3、关于首位置插入、末位置插入，我们可以根据指定的pos值来判断：

如果pos<=0,那么就直接调用add方法，

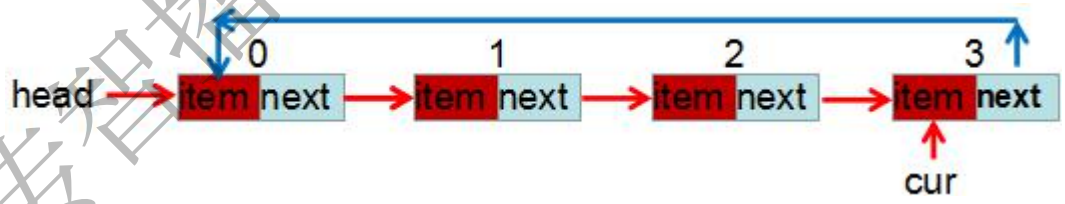
如果pos>=len(list_name),那么就调用append方法，在尾部增加元素即可。

根据我们之前分析的三步走：

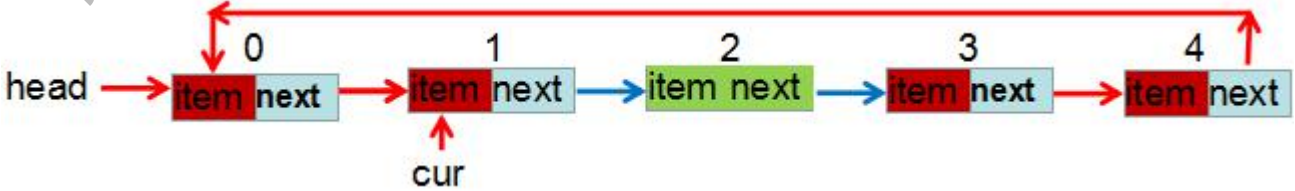
1、存储数据到一个新的结点中

node=BaseNode()

2、当前链表



3、将新的结点加入到尾部，设定尾结点N的next属性为新结点A，修改新结点A的next为当前的头结点B



关键点：

找位置，cur的位置必须在pos的前一个位置，原因就是链表结构的next属性

4、中间添加数据，肯定会遇到头添加和尾添加，那么我们之间调用之前的方法即可。

代码实践：

```
    def insert(self, pos, item):
        # 定义一个存储数据的新结点
        node = BaseNode(item)
        # 头部添加内容
        if pos <= 0:
            self.add(item)
        # 在尾部添加元素
        elif pos >= self.length():
            self.append(item)
        # 在中间添加元素
        else:
            # 找到头信息，并且开始设置计数器初始值
            cur = self.__head
            count = 0
            # 找到要插入的位置的上一个位置
            while count < (pos - 1):
                count += 1
                cur = cur.next
            # 设置新结点的 next 属性为当前结点的下一个结点
            node.next = cur.next
            # 设置当前结点的 next 属性为新结点
            cur.next = node
```

代码测试：

```
if __name__ == "__main__":
    ll = CycleSingleLinkList()
    ll.add(1)
    ll.append(3)
    ll.insert(1,6)
    ll.travel()
```

关键点：

指定位置插入，肯定包括头和尾，所以活用已有方法：self.add(item)、self.append(item)

插入位置的判断：while count < (pos - 1)

新结点的next指向cur的下一个结点：node.next = cur.next

cur结点的next指向新结点：cur.next = node

3.5.6 实践 之 删除

删除的操作就是指定元素操作，而元素的位置可能是头、中间、尾

删除的步骤：

判断链表是否为空，空则直接返回False，不空的话，继续下面步骤

判断内容是否是我们要删除的

是的话，判断内容的位置

头位置：直接切换当前链表的self.__head为下一个结点，修改末尾结点的next属性即可

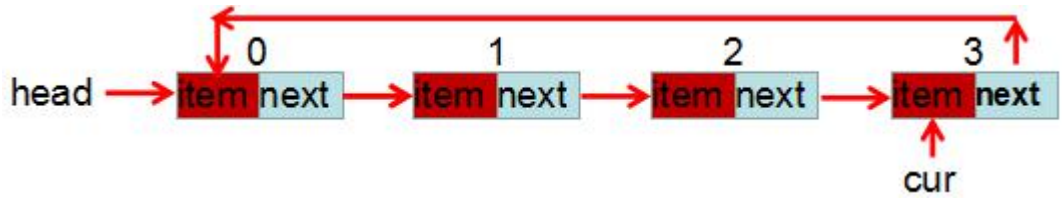
尾位置：把要删除的结点B的上一个结点A的next属性指向头结点即可

中间位置：把要删除的结点B的上一个结点A的next属性指向B结点的next属性即可

删除完成后，直接返回True

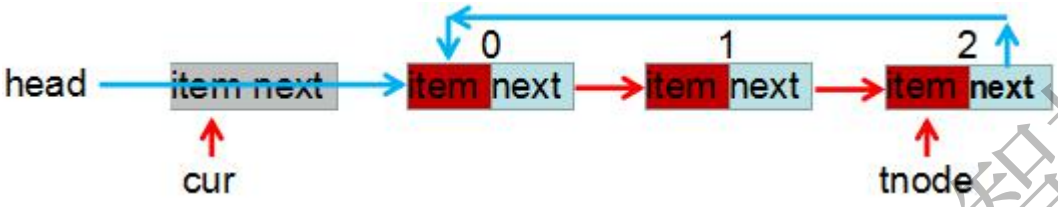
不是的话，直接返回False

1、当前链表的情况



2、删除头情况

删除头结点，本身跟单向链表的动作样，但是多了一个尾结点的next指定首结点的操作，我们之前只是用cur来确定当前结点，一旦cur指定了我们的头结点，那么尾结点我们就需要有一个专用的标识来指定。



分析：

无非就是将头结点标签指向cur.next即可，然后将尾结点的next指向新的头结点即可

所以需要先获取尾结点tnode，然后删除首结点，再设置尾结点的next属性

删除成功后，需要退出操作，return

代码实现：

1、确定头结点就是我们要删除的内容

```
def remove(self,item):
    # 找到当前链表的头信息
    cur = self.__head
    # 链表不为空情况下(除了尾结点之外)
    while cur.next is not self.__head:
        # 如果当前结点 cur 的 item 就是要删除的内容
        if cur.item == item:
            # 如果当前结点 cur 就是当前链表的头结点
            if cur == self.__head:
```

2、获取尾结点内容，尾结点的获取方法跟之前一样

```
                # 继续定位尾结点 tnode
                tnode = self.__head
                # 只要退出当前循环，那么 tnode 所处的下一个位置就是尾结点
                while tnode.next is not self.__head:
                    tnode = tnode.next
```

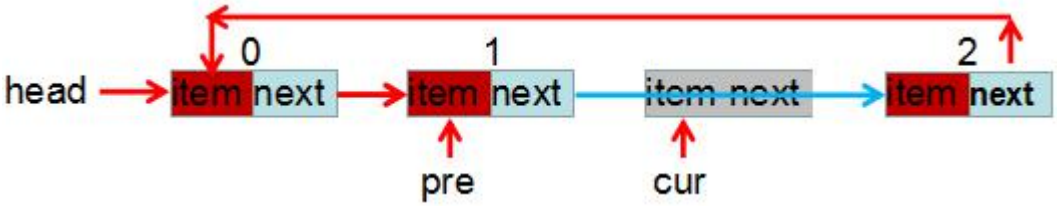
3、重新定义新头结点和尾结点的next属性

```
                # 设定新的头结点位置
                self.__head = cur.next
                # 重新指定尾结点的 next 属性为新的头结点
                tnode.next = self.__head
```

4、删除成功，那么退出当前操作

```
                # 删除完毕，退出操作
                return
```

4、删除中间情况



分析：

删除中间的情况，跟之前单向链表的情况是一样的，所以代码不需要做任何变动

当前cur结点的上一个结点pre

元素不匹配时候，pre和cur的移动

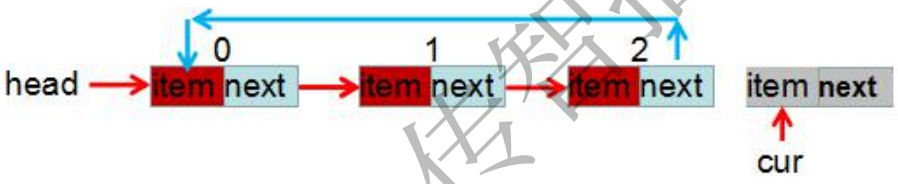
删除元素时候，pre结点的next属性修改

代码实现：

```
            # 如果当前结点 cur 不是头结点
            else:
                # 将当前结点 cur 的上一结点的 next 属性设定为 cur 的下一个结点 C
                pre.next = cur.next
            return
        # 如果不是要找的删除元素，将当前结点 cur 的上一结点作为当前结点
        pre = cur
        # 将当前结点 cur 的 next 属性为下一结点 C
        cur = cur.next
```

3、删除尾情况

情况一：多结点链表：



情况二：单节点链表

分析：

只要while的循环退出，那么相当于cur的标签就移动到了尾结点，那我就需要来判断这个尾结点的内容是不是我要删除的

代码实现：

```
        # 退出循环时候 cur 已是尾结点
        if cur.item == item:
            # 当 cur 是尾结点又是头结点，那么当前链表就是一个元素
            if cur == self.__head:
                self.__head = None
            # cur 不是唯一的结点，
            else:
                pre.next = self.__head
```

5、链表为空

使用已有的方法进行判断即可，如果为空就直接退出即可

代码实现：

```
        # 如果链表是空链表
        if self.is_empty():
            return
```

最终代码实践：

```
    def remove(self,item):
        # 如果链表是空链表
        if self.is_empty():
            return
        # 找到当前链表的头信息
        cur = self.__head
        pre = None
        # 链表不为空情况下
        while cur.next is not self.__head:
            # 如果当前结点 cur 的 item 就是要删除的内容
            if cur.item == item:
                # 如果当前结点 cur 就是当前链表的头结点
                if cur == self.__head:
                    tnode = self.__head
                    while tnode.next is not self.__head:
                        tnode = tnode.next
                    self.__head = cur.next
                    tnode.next = self.__head
                # 如果当前结点 cur 不是头结点
                else:
                    pre.next = cur.next
                return
            # 如果不是要找的删除元素，移动结点
            pre = cur
            cur = cur.next
        # cur 是尾结点情况
        if cur.item == item:
            # 链表只有一个元素
            if cur == self.__head:
                self.__head =None
            # 链表不止一个元素，
            else:
                pre.next = self.__head
```

测试代码：

```
if __name__ == "__main__":
    ll = CycleSingleLinkList()
    ll.add(1)
    ll.append(3)
    ll.insert(1,6)
    ll.remove(6)
    ll.travel()
```

关键点：

头部元素删除：

判断的顺序：当前链表不为空-->删除的内容匹配-->内容所在位置

头结点的指向：`self.__head = cur.next`

删除节点后，退出循环：`return`

中间元素删除：

当前结点cur的上一个结点：`pre = None`

前后两个结点的next属性：pre.next = cur.next

内容不匹配，cur节点和pre节点同时后移：pre = cur 、 cur = cur.next

尾部元素删除：

循环退出时候的cur就是尾结点：while cur.next is not self.__head

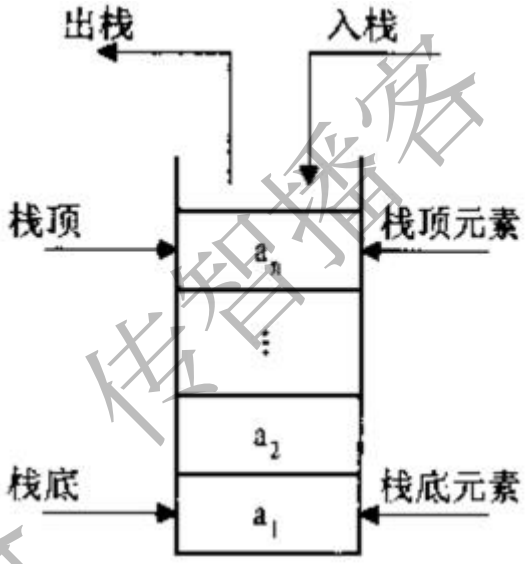
链表为一个结点：self.__head =None

链表不是一结点：pre.next = self.__head

链表为空：if self.is_empty()

3.6 栈(简单)

3.6.1 栈的简介



栈概念：

栈（stack），是一种数据项按序排列的数据结构，只能在一端对数据项进行插入和删除等操作，简单来说它就是一个数据的容器。

栈特点：

只能允许在容器的一端进行加入数据和输出数据的运算。

由于栈数据结构只允许在一端进行操作，因而按照后进先出（LIFO, Last In First Out）的原理运作。

栈基本术语：

栈顶、栈底、栈顶元素、栈底元素、入栈、出栈

3.6.2 栈的实践

栈本质上就是对我们之前学的线性表的一个功能封装。

可以用顺序表实现，也可以用链表实现。接下来就用列表的方式来演示一下栈的基本功能

栈的基本功能：

栈的创建：

Stack() 创建一个新的空栈

信息查看：

peek() 返回栈顶元素

is_empty() 判断栈是否为空

size() 返回栈的元素个数

栈的基本操作：

push(item) 添加一个新的元素item到栈顶

pop() 弹出栈顶元素

实践代码：

栈的基本实现

```
class Stack(object):
    # 栈的基本特性
    def __init__(self):
```

```
        self.__items = []
```

栈的基本信息查看

```
    # 判断栈是否为空
    def is_empty(self):
        return self.__items  == []
```

关键点：

对栈的基本属性进行空匹配：self.__items == []

```
    # 查看栈顶元素
    def peek(self):
        if self.is_empty():
            return False
        return self.__items[-1]
```

关键点：

获取列表的末尾元素，切片功能:self.__items[-1]

```
    # 返回栈的长度
    def size(self):
        return len(self.__items)
```

关键点：

使用列表的len功能自动获取:len(self.__items)

栈的基本动作

```
    # 添加元素到栈
    def push(self, item):
        self.__items.append(item)
```

关键点：

利用列表的追加功能实现：self.__items.append(item)

```
    # 出栈动作
    def pop(self):
        return self.__items.pop()
```

关键点：

利用列表的删除末尾元素功能实现：self.__items.pop()

3.7 队列(简单)

队列我们学习两个方面：

单向队列和双向队列

3.7.1 队列简介

队列 是一种数据项按序排列的数据结构，只能在一端对数据项进行插入和删除等操作

队列是一种**先进先出**的（First In First Out）的线性表，简称FIFO。允许插入的一端为队尾，允许删除的一端为队头。队列不允许在中间部位进行操作！

假设队列是q=（a1，a2，……，an），那么a1就是队头元素，而an是队尾元素。这样我们就可以删除时，总是从a1开始，而插入时，总是在队列最后。这也比较符合我们通常生活中的习惯，排在第一个的优先出列，最后来的当然排在队伍最后。

3.7.2 队列的实践

队列的实现

同栈一样，队列也可以用顺序表或者链表实现。

操作

队列的创建

Queue() 创建一个空的队列

队列信息查看

is_empty() 判断一个队列是否为空

size() 返回队列的大小

队列的基本操作

enqueue(item) 往队列中添加一个item元素

dequeue() 从队列头部删除一个元素

代码实践：

队列的基本实现：

```
class Queue(object):
    """队列的基本属性"""
    def __init__(self):
        self.__items = []
```

队列的基本信息查看

```
    # 判断队列是否为空
    def is_empty(self):
        return self.__items == []


    # 队列的长度获取
    def size(self):
        return len(self.__items)
```

队列的基本操作

```
    # 给队列添加元素
    def enqueue(self, item):
        self.__items.append(item)


    # 将队列中的元素弹出
    def dequeue(self):
        return self.__items.pop(0)
```

3.7.3 双端队列(了解)

双端队列

双端队列（deque，全名double-ended queue），是一种具有队列和栈的性质的数据结构。

双端队列中的元素可以从两端弹出，其限定插入和删除操作在表的两端进行。双端队列可以在队列任意一端入队和出队。

关键点：

两个不同方向的栈或队列，组合到一起了。

操作

队列的创建

Deque() 创建一个空的双端队列

队列信息查看

is_empty() 判断双端队列是否为空

size() 返回队列的大小

队列的基本操作

add_front(item) 从队头加入一个item元素

add_back(item) 从队尾加入一个item元素

remove_front() 从队头删除一个item元素

remove_back() 从队尾删除一个item元素

实践代码：

队列的基本实现：

```
class Deque(object):
    """双端队列"""
    def __init__(self):
        self.__items = []
```

队列的基本信息查看：

```
    def is_empty(self):
        """判断队列是否为空"""
        return self.__items == []


    def size(self):
        """返回队列大小"""
        return len(self.items)
```

队列的基本操作：

```
    def add_front(self, item):
        """在队头添加元素"""
        self.__items.insert(0,item)


    def add_rear(self, item):
        """在队尾添加元素"""
        self.__items.append(item)

    def remove_front(self):
        """从队头删除元素"""
        return self.__items.pop(0)


    def remove_rear(self):
        """从队尾删除元素"""
        return self.__items.pop()
```

3.8 树

这部分我们从三个方面来学习：树简介、二叉树简介、二叉树实践

3.8.1 树简介

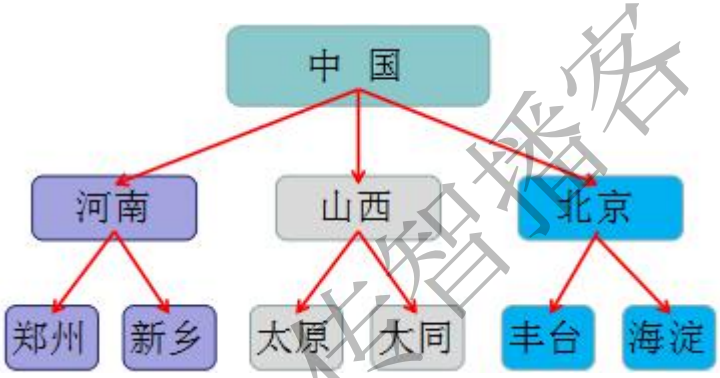
关于树的简介，我们从以下六个方面来介绍：

基本概念、基本特点、基本术语、树的种类、树的存储、使用场景

基本概念

树是一种数据结构，基本单位是节点。它是由n个节点组成一个具有**层次关系**的集合。因为看像一棵**倒挂的树**，也就是说它是根朝上，而叶朝下的。

特点：队列中的数据不仅仅有前后关系，还有层次关系，

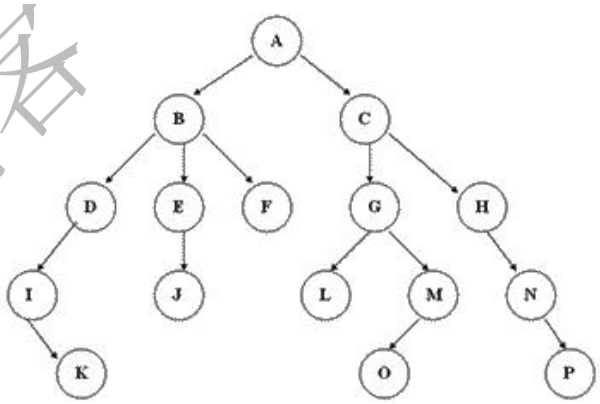


基本特点：

每个节点有零个或多个子节点；

没有父节点的节点称为根节点；

每一个非根节点有且只有一个父节点；



基本术语

父节点：含有子节点的节点；

子节点：有父节点的节点

兄弟节点：相同父节点的节点

叶节点或终端节点：没有子节点的结点；

节点的祖先：根节点到该节点之间所有父节点的集合；

节点的度：一个节点含有子节点的个数；

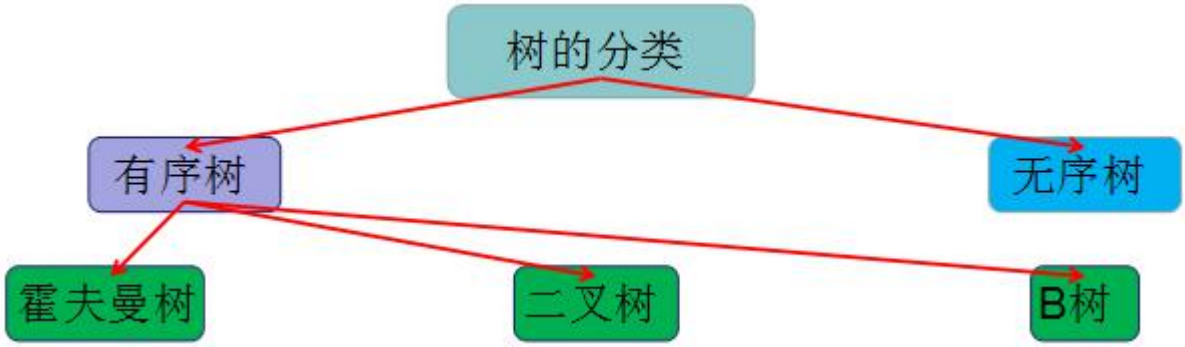
节点的层次：根节点到该节点所经历的结点的个数；

树的度：一棵树中，最大的节点的度称为树的度；

树的高度或深度：树中节点的最大层次；

森林：由m（m>=0）棵互不相交的树的集合称为森林；

树的种类



无序树：树中任意节点的子节点之间没有顺序关系，这种树称为无序树，也称为自由树；
有序树：树中任意节点的子节点之间有顺序关系，这种树称为有序树；
二叉树：每个节点最多含有两个子树(度小于2)的树称为二叉树；
...

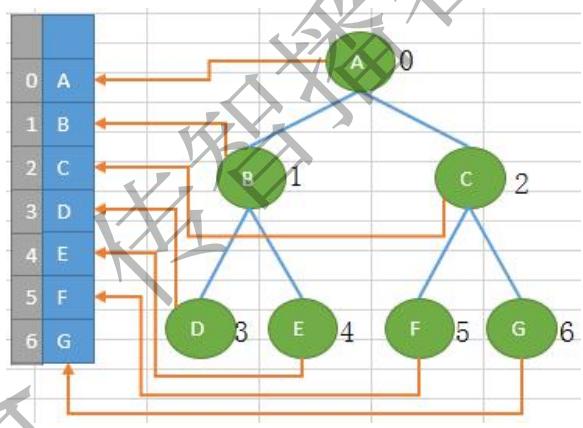
树的存储

在数据结构进阶中，我们学到了多种数据存储结构，最基础的就是：顺序表和链表，而树就自然而然的有两种存储方式：

顺序存储：

顺序存储常见的形式是列表

将二叉树的所有数据，从上到下分层，每层从左到右，依次编号，然后按照编号顺序把所有元素存储到一个存储空间中即可。



示例：

使用顺序表存储一组队列：[0,1,2,3,4,5,6]

特点：

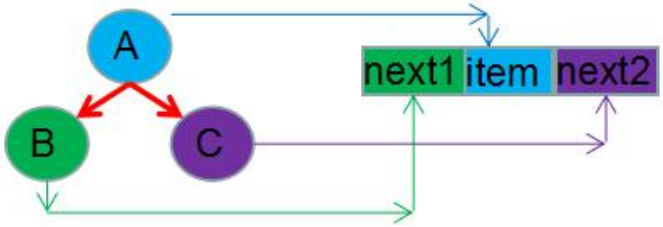
按照编号顺序存储，所以查询比较快
元素之间只有顺序，没有关系
所有元素存储到一个空间，所以整体占用空间大

父节点和子节点关系：

父节点位置 i,找子节点：
左子节点位置：2i + 1　　右子节点位置：2i + 2
左子节点位置 i，找父节点：
父节点位置：(i-1)/2
右子节点位置i，找父节点：
父节点位置：(i-2)/2

链式存储：

顺序表只是保存元素的前后关系，并没有前有的调用关系，而二叉树的重点就是关系，所以二叉树通常以链式存储



特点：

链表格式：存储数据+关系索引
存储二叉树无非就是多几个关系索引而已。

3.8.2 二叉树简介

简介

二叉树是每个节点最多有两个子树的树结构。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）

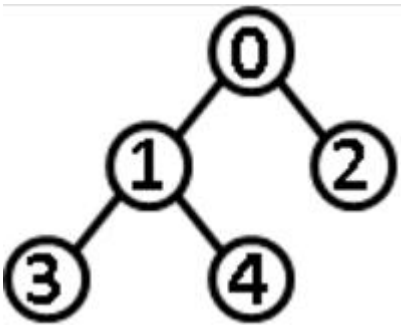
二叉树特性

对完全二叉树，若从上至下、从左至右编号：
起始根节点编号为0：
编号为i的结点，其左节点 2i+1，右节点 2i＋2,其父节点的编号(i-1)/2或(i-2)/2
起始根节点编号为1：

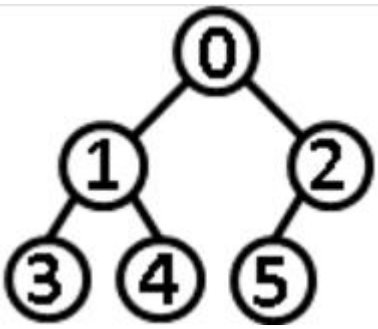
编号为i的结点，其左节点 2i，右节点 2i+1,其父节点的编号i/2或(i-1)/2

常见二叉树：

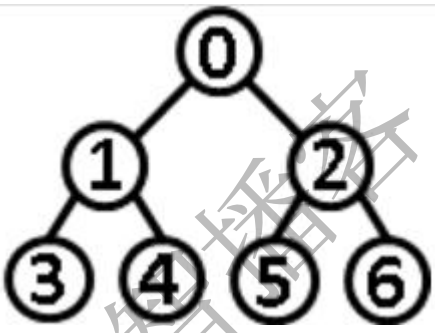
完美二叉树	除了叶子结点之外的每一个结点都有两个孩子，每一层都被完全填充。
完全二叉树	除了最后一层之外的其他每一层都被完全填充，并且所有结点都保持向左对齐。
完满二叉树	所有非叶子结点的度都是2，除了叶子结点之外的所有结点都有两个子结点。



完满二叉树



完全二叉树



完美二叉树

3.8.3 二叉树实践

我们用链表的方式来实践一下完美二叉树。

二叉树创建：

关于创建，我们主要创建两个内容：节点和树的结构

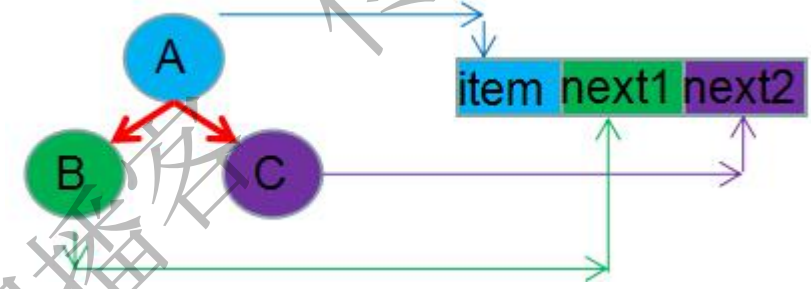
1、节点基本信息

根据我们对链表的理解，我们要保存的数据，至少有三个属性：

item：真正存储的数据

lsub：左子节点的索引

rsub：右子节点的索引



代码实现：

```
class Node(object):
    """ 定义节点的基本属性"""
    def __init__(self,item):
        # 节点存储的内容
        self.item = item
        # 左侧子节点的索引值
        self.lsub = None
        # 右侧子节点的索引值
        self.rsub = None
```

2、二叉树的结构

根据树的特点，我们要从下面两个方面来考虑：根节点+添加节点

2.1 根节点

二叉树必定有一个根节点，而且根节点可以为空(表示树不存在)

构建一个二叉树的时候，一定要让所有操作知道他的根节点是谁--基本属性

代码实现：

```
class Tree(object):
    # 定义树结构的基本属性：根节点
    def __init__(self, node=None):
        self.root = node
```

思考：

self.root 能否写成 self.__root?

不能，因为这个值是树的根，肯定要看

2.2 添加节点

添加节点我们会遇到两种情况：

没有任何数据的二叉树

存在数据的二叉树

2.2.1 对于没有任何数据的二叉树：

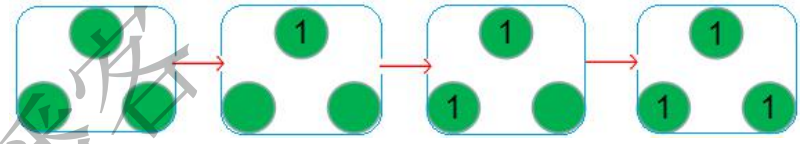
1)如果根节点没有数据，添加到根节点即可。

代码实现：

```
#定义一个添加节点的函数
def add(self, elem):
    # 定义要添加到树结构中的节点信息
    node = Node(elem)
    #如果树是空的，则对根节点赋值，对应的特点：从上到下
    if self.root == None:
        self.root = node
```

2.2.2 存在数据的二叉树

对于存在数据的二叉树，添加数据步骤如下：



1) 二叉树有数据，说明肯定有根节点或者父节点，那么我在给哪个父节点添加数据

2) 添加左侧子节点数据

3) 再添加右侧子节点数据

4) 接着按照 "父节点->左子节点->右子节点" 的节点添加顺序循环下去。

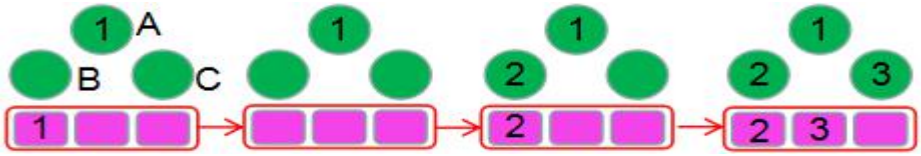
在有数据的二叉树中添加元素，需要对父节点的两个子节点分别进行空值判断

1> 前提，我要对哪个父节点添加子节点数据

二叉树和子二叉树们都是"一父两子"结构，那么添加数据的时候，是针对哪个父节点呢？

我们建一个空列表alist，把父节点A添加进去，即"待处理的父节点列表"

当要在A结点下添加数据，先把A结点从alist中弹出，说明我给A添加子节点了。



2> 左侧B节点空值判断

如果内容为空，则添加数据，因为该结点是下一层的父节点，同时追加到alist末尾

如果内容非空，对右子节点进程空值判断

3> 右侧C节点空值判断

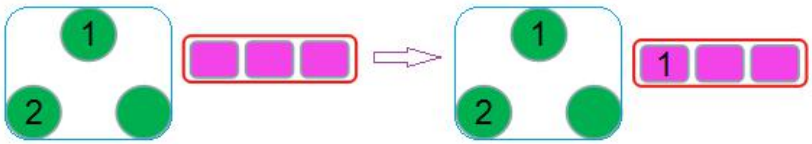
如果内容为空，则添加数据，因为该结点是下一层的父节点，同时追加到alist末尾

如果右子节点内非空，说明该层已满，只能在下一层添加数据，

将左B节点作为父节点，从alist中弹出B元素，代表要给B结点添加数据了，执行2-3操作即可

添加子节点相当于是在操作哪个父节点，所以搞一个空列表，没处理的放到队列，扔一个处理一个

1)待处理父节点队列



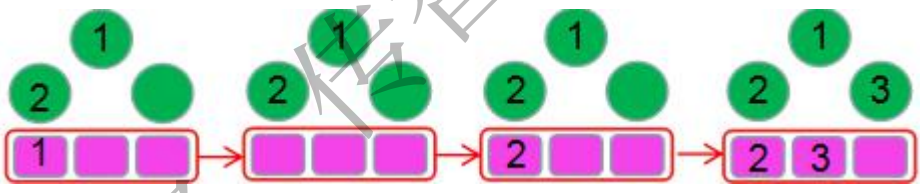
代码实现：

```
#定义一个添加节点的函数
def add(self, elem):
    ...
    # 如果根节点存在数据
    else:
        # 使用一个临时列表来存储我们要处理的元素：对应的特点-从左到右
        queue = []
        # 先把根节点放到我们要处理的临时队列中
        queue.append(self.root)
```

3)在A结点添加数据时，将alist队列里面的A元素弹出。

4)判断左子节点是否有数据，有的话，将其追加到alist队列末尾，对同级的右子节点进行空值判断

5)判断右子节点是否有数据，没有的话，就添加数据，同时将元素追加到alist末尾



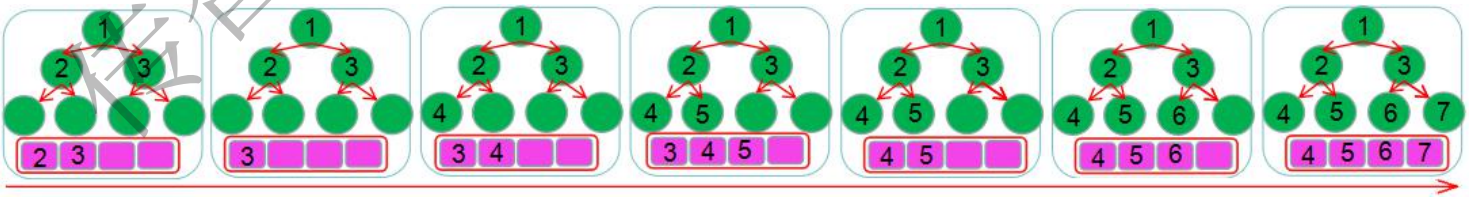
6) 第二层的2和3节点都有内容，所以再次添加数据的话，从alist中找待处理父节点,添加相应数据

7) 先给2节点添加数据，先把2的元素从alist中弹出，

8) 2节点的两个子节点都是空，添加数据至相应位置，同时元素追加到alist末尾

9) 同样方法给3结点添加元素，先弹出在加数据.

10)再添加数据的话，重复6-9步骤



代码实现：

```
#定义一个添加节点的函数
def add(self, elem):
    ...
else:
    ...
    # 只要待处理队列中有要处理的元素，那么就一直处理下去
```

```
            while queue:
                # 从队列的头部获取要处理的元素
                cur = queue.pop(0)
                # 如果要处理节点的左侧子节点为空
                if cur.lsub == None:
                    # 把接收的 item 放到左侧子节点位置
                    cur.lsub = node
                    # 添加完毕后，退出
                    return
                # 左侧节点有数据，把该数据追加到待处理父节点队列末尾
                else:
                    queue.append(cur.lsub)
                # 如果要处理节点的右侧子节点为空
                if cur.rsub == None:
                    # 把接收的 item 放到左侧子节点位置
                    cur.rsub = node
                    # 添加完毕后，退出
                    return
                # 右侧节点有数据，把该数据追加到待处理父节点队列末尾
                else:
                    queue.append(cur.lsub)
```

代码实践：

二叉树增加数据完整代码：

```
class Tree(object):
    """树结点类"""
    def __init__(self, root=None):
        self.root = root

    def add(self, elem):
        # 待增加树节点
        node = Node(elem)
        # 树空，元素给根
        if self.root == None:
            self.root = node
        else:
            # 待处理父节点队列
            queue = []
            queue.append(self.root)
            # 给父节点增加数据
            while len(queue) > 0:
                # 确定要操作的父节点
                cur = queue.pop(0)
                # 左侧子节点添加数据
                if not cur.lsub:
                    cur.lsub = node
                    return
                else:
                    queue.append(cur.lsub)
```

```
                # 右侧子节点添加数据
                if not cur.rsub:
                    cur.rsub = node
                    return
                else:
                    queue.append(cur.rsub)
```

测试代码：

关于怎么在树创建好后的查看方法，参考后续的二叉树查询知识

关键点：

1、空树的处理：　　if self.root == None:

2、不空树情况，待处理父节点处理：　　queue.append(self.root)

3、左右侧子节点处理：

空则增加数据：　　if not cur.lsub:

不空，则把数据追加到待处理队列　　queue.append(cur.lsub)

第 4 章 算法 进阶

算法进行部分我们从三个方面来学习：排序、搜索、树

每个具体的知识点我们都会根据六个角度来学习：

简介 -- 原理或图示 -- 实践分析 -- 代码实践 -- 时间复杂度 -- 大总结

4.1 排序

我们之前学习的是线性表，那么说到线性，我们就会想到一个词“顺序”，说到顺序，那么不可避免的就涉及到了“排序”

什么是排序？

把无序的队列变成有序的队列

特点：

输入：无序队列

输出：有序队列

应用场景：

各种排行榜 - 服不服排行榜

各种表格 - 座位表

临时列表 - 其他程序用的临时队列

...

4.1.1 排序算法简介

我们会从四个方面来介绍排序算法的简介：

定义、关键点、稳定性、常见种类

排序算法

排序算法是一种将一串无规律数据依照特定顺序进行排列的一种方法或思路。

排序算法关键点：

有序队列 ：有序区刚开始没有任何数据， 逐渐变多

无序队列 ：无序区刚开始存放所有数据， 逐渐变空

排序算法的稳定性

稳定性：

队列中有相同的元素，排序前后，这两个相同元素的顺序有没有发生变化。

没有发生变化，就表示算法有稳定性　　如果发生变化，就表示算法没有稳定性

常见排序算法：

基础： 冒泡、插入、选择　　中级： 快速

高级： 堆、归并　　其他： 基数、希尔、桶

4.1.2 冒泡排序

军训：

冒泡排序简介

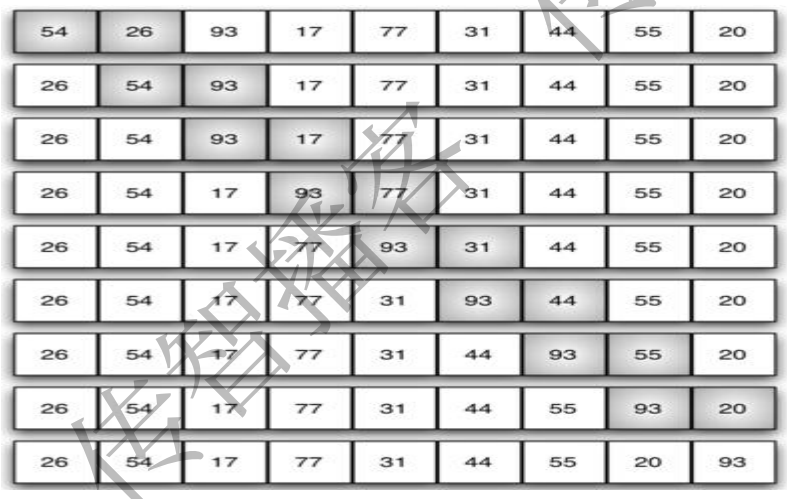
冒泡排序是一种简单的排序算法。

相邻的元素两两比较，升序的话：大的在右，小的在左，降序的话，反之。

经过数次比较循环，最终达到一个从小到大(升序)或者从大到小(降序)的有序序列

这个算法由于类似于气泡从水底冒出，所以叫“冒泡”排序。

冒泡排序原理



在整个冒泡排序过程中，有一个标识指向两个元素的最大值，当这个最大值移动的时候，标识也会随之移动，这就叫做：**过程跟踪**

举例：

alist[i]这个i一直跟着93

93和17比较的时候，i=2

93大于17，所以两个元素要换位置，但是i又要保证一直指向93，所以

i = i+1

保证93移动的过程中，

alist[i] = 93

特点：

元素替换：相邻元素

从小到大：

左比右大，数据先交换位置，大的和右侧的元素继续比较

左比右小，数据不交换位置，大的和右侧的元素继续比较

左右相等，数据不交换位置，大的和右侧的元素继续比较

比较次数：无序队列元素个数 - 1

冒泡次数：无序队列元素个数 - 1

冒泡次数和比较次数关系

冒泡次数：1 2 3 4 5 6 7 8
比较次数：8 7 6 5 4 3 2 1
示例：如果当前无序列表元素有n个，那么在整个冒泡排序过程中
比较次数的范围：从 n-1 逐渐降序到 1

过程跟踪：
比如说：alist[i] = 93,

冒泡排序的分析

根据我们对冒泡排序的原理分析，我们知道要完成一次完整冒泡排序需要考虑如下几个方面：

1、最基本元素比较 即“元素替换”
2、每一次冒泡排序，内层的元素比较次数 即“内层比较循环”
3、执行多少次冒泡排序 即“外层冒泡循环”
4、特殊情况 即“不替换情况”

冒泡排序例子：要实现一个从小到大的有序队列

1、元素替换

两个**相邻数字**比较大小，如何进行**大小替换**，大的放到小的后面？

数字比较后互换位置，无非就是变量的替换。

因为我们是以python的list来举例，那么就可以使用列表的下标来获取具体的内容。

alist列表：[54,26,93,17,77,31,44,55,20]
alist[0] = 54 alist[1] = 26

前面两个数量，先进行比较，然后进行互换，谁大谁在后面：

```
if alist[0] > alist[1]:
    alist[0],alist[1] = alist[1],alist[0]
```

总结规律：第 i 个变量和第 i+1个变量来替换：

```
if alist[i] > alist[i+1]:
    alist[i],alist[i+1] = alist[i+1],alist[i]
```

2、内层比较循环

这个知识点，我们从两个方面来分析：

比较次数、元素范围(下标)

2.1 比较次数

第一次冒泡排序时候，需要多少次**数值比较**？

第一次排序，由第1行数据的总数量来决定，比如说有9个数，那么我们就比较8次，并且确最大93的放在队列的最后，相当于有序队列多了1个元素，无序队列少了1个元素

第二次排序，我们只需要对未排序队列中的8个元素进行冒泡排序，那么8个元素需要比较7次，最终将最大的77从无序队列中移除，放到有序队列中93的前面。

...

根据上面分析，我们知道，每次冒泡排序元素比较的范围，都和需要未排序队列元素的数量n有关，这个比较数量就是当前无序队列的元素总个数减一：【n - 1】。

2.2 元素取值范围

元素的取值范围，无非就是列表下标的范围而已，下标肯定是从0开始，所以我们分析最大的取值即可。

如果假设无序队列有n个元素，list[i]和list[i+1]进行比较，那么这个i的取值范围是多少？

根据我们对下标的理解，最后元素下标的取值，肯定是n-1，对于list[i+1]来说，i+1最多就是n-1

那么i的取值范围就是0 ~ n-2，也就是：range(n-1)

综合分析，每一次冒泡排序代码效果就是：

代码实现：（可以测试）

```
# 获取当前列表的总数量
n = len(alist)
# 确定元素比较的范围
for i in range(n-1):
    # 元素比较替换操作
    if list[i] > list[i+1]:
        list[i],list[i+1] = list[i+1],list[i]
```

3、外层的冒泡循环

这个知识点，我们从两个方面来分析：

冒泡次数、冒泡次数和元素比较次数关系

3.1 冒泡次数

对于一个无序队列来说，需要进行多少次冒泡排序，整个队列才有序？

第二次排序，因为有一个数字已经放到有序队列了，所以无序队列就剩下8个数值了，元素比较 7 次

第三次排序，无序队列剩下7个数值，比较 6 次

...

第八次排序，无序队列剩下2个数值，比较 1 次

根据分析，冒泡排序次数(m)还是无序队列的元素数量(n)有关系，关系就是：

m = n - 1，那么冒泡排序次数的取值范围就是 range(n)

3.2 冒泡次数和元素比较次数关系

通过冒泡原理的分析，冒泡排序次数m和元素比较次数j之间有如下关系：

冒泡排序次数m：1 2 3 4 5 6 7 8 范围是 for m in range(n)

元素比较次数j：8 7 6 5 4 3 2 1 范围是 for j in range(n-1,0,-1)

因为我们的冒泡排序只关心次数，所以冒泡排序次数m也可以写成 for m in range(n-1,0,-1)，那么m和j效果就变成了：

冒泡排序次数m：8 7 6 5 4 3 2 1

元素比较次数j：8 7 6 5 4 3 2 1

问：为甚要关注j的排序呢？

因为j的值和元素比较的i是一一对应的

所以我们在获取冒泡排序次数m的时候，就可以直接把元素比较的次数j获取出来，所以冒泡循环和元素比较循环的关系就是：

代码实现：（可以测试）

```
# 获取列表元素的总数量
n = len(alist)
# 冒泡排序循环范围
for j in range(n-1,0,-1):
    # 内层的数据比较循环范围
    for i in range(j):
        ...
```

4、列表的特殊情况

如果列表本来就是已经排好序的列表，我还一遍遍的冒泡排序就没有意义了，那该怎么判断呢？

解决方法：

在数据比较之前先定义一个计数器初始值0，每比较替换一次，计数器就加1

如果排序完毕计数器还是0，说明整个序列就是有序队列了，所以直接终止当前循环 break 即可。

代码实现：

```
# 获取列表元素的总数量
n = len(alist)
for j in range(n-1,0,-1):
    # 开始比较前，定义计数器 count 的初始值为 0
    count = 0
    for i in range(j):
        if alist[i] > alist[i+1]:
            ...
            # 数据替换完毕，计数器加 1
            count += 1
    # 如果单次循环比较结束后，计数器还是 0 ,那么就表示列表已排序，终止当前循环
    if count == 0:
        break
```

冒泡排序实践

代码实践

```
def bubble_sort(alist):
    # 获取列表元素的总数量
    n = len(alist)
    # 冒泡排序循环范围
    for j in range(n - 1, 0, -1):
        # 开始比较前，定义计数器 count 的初始值为 0
        count = 0
        # 内层的数据比较循环范围
        for i in range(j):
            if alist[i] > alist[i + 1]:
                alist[i], alist[i + 1] = alist[i + 1], alist[i]
                # 数据替换完毕，计数器加 1
                count += 1
        # 如果计数器的值为 0，表示没有发生任何替换,那么就退出当前循环
        if count == 0:
            break
```

```
```

代码测试：

```
if __name__ == "__main__":
    li = [54,26,93,17,77,31,44,55,20]
    print(li)
    bubble_sort(li)
    print(li)
```

拓展：

生成有序列表：`list(range(10000))`

有序打乱成无序列表：`random.shuffle(list)`

时间复杂度

最优时间复杂度：O(n)

对于每一次冒泡排序的内部元素比较循环，都不进行替换，内部的比较循环时间复杂度是O(1)。

对于冒泡循环，和序列元素数量相关，所以冒泡循环的时间复杂度是O(n)

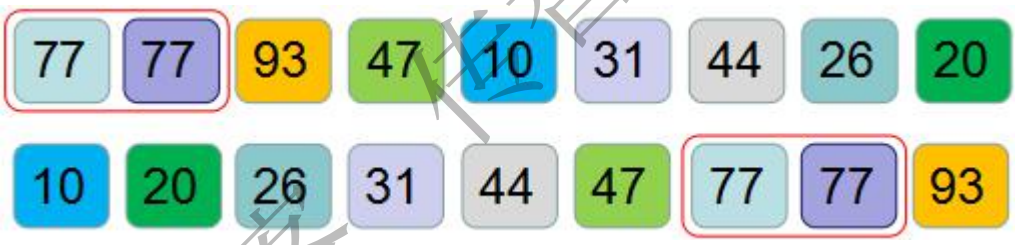
对于整体来说：最优时间复杂度就是 O(n)

最坏时间复杂度：$O(n^2)$

最坏不过每次冒泡循环的元素比较，需要全部进行比较替换，所以每一次元素比较循环都是O(n)

那么对于总体来说，内外都是O(n)，所以最坏的时间复杂度是$O(n^2)$

稳定性：稳定



对于图示得知，47进行冒泡排序，两个77的顺序没有做变动，所以是稳定。

拓展：

改那个地方，结果是降序？

`if alist[i] > alist[i + 1]` 代码中的 > 改为 <

4.1.3 选择排序

选择排序简介

选择排序（Selection sort）是一种简单直观的排序算法。

简单来说就是从无序队列里面挑选最小的元素，和无序队列头部元素替换(放到有序队列中)，最终全部元素形成一个有序的队列。

特点：

两个队列：有序队列，无序队列

元素替换

选择排序原理

首先在未排序序列中找到最小（大）元素，和无序队列的第一个元素替换位置，（即形成有序队列）
以此类推，直到所有元素全部进入有序队列，即排序完毕。

选择排序图示

原始序列：

选择排序-1：无序队列最小元素17，和无序队列第一个位置元素替换

选择排序-2：无序队列最小元素20和无序队列第一个位置元素替换

...

选择排序-8：无序队列最小元素77和无序队列第一个位置元素替换

注释：

红框表示已排序队列，其他为非排序队列

总结：

选择排序的主要特点与**元素替换**有关。

每次移动一个元素，就有一个元素放到有序队列，
n个元素的无序队列最多进行(n-1)次交换，就可以形成有序队列。
如果整个队列已排序，那么它不会被移动。

实践分析：

通过上面的原理解析，我们知道，要进行一次完整的选择排序，我们要考虑三个方面：

1、无序队列查找最小元素　　即“比较循环”
2、最小元素和无序队列第一个元素替换位置　　即“元素替换”
3、需要进行多少次替换，才能形成队列　　即“选择循环”

1、比较循环

如何从无序的队列中，寻找最小的数值？

所有元素两两比较定最小：所以需要有两个标签：mix标识最小元素，cur标识用于遍历所有元素。

注意：

过程跟踪：mix永远指向最小的，cur指向的元素负责对比。
mix 和 cur 标签的初始化地址是相邻的：
mix标签所在元素的下标是j，那么cur标签所在元素的下标是 j+1

1.1）将mix标签(下标是mix_index)指向无序队列第一个位置

1.2 ）cur指向mix的下一个元素，然后mix标签所在元素和后面cur标签所在元素进行比较，看谁小

注意：cur标签所在元素下标是i

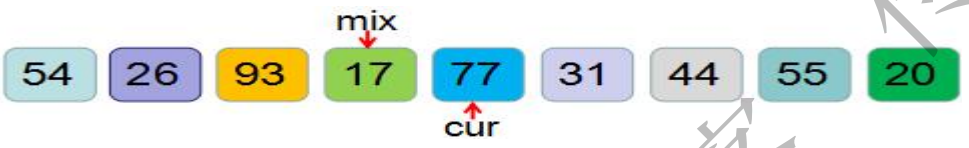
1.3 ）大小比较后，将mix的标签移动到最小元素的上面，cur标签后移一位。

代码实现：

```
# 定义 mix 标签初始值
min_index = 0
# 让下标为 min_index 和 i 的数值进行比较，谁小，那么就移动 min_index 标签到小的元素
if alist[i] < alist[min_index]:
    min_index = i
```

1.4 ） 继续比较mix和cur连个标签指定的值，如果mix比不过cur标签的值，那么cur的标签往后移

1.5） mix和cur标签所在元素，继续进行比较后，然后移动mix标签到最新的最小值上，cur标签后移一位



1.6 ） 如此循环操作，当cur标签移动到最后一位，发现mix标签没有动

综合分析：

上面的六步主要做了两件事情："mix标签移动" 和 "cur标签的后移"

如果当前无序队列有n个元素，那么整个过程当中，

1） mix标签只有在cur比mix元素小的时候才移动到小元素的位置

2） cur标签从【min_index+1】，一直移动到了无序列表的末尾元素位置【n-1】

3） 当cur标签元素最终都没有比mix标签所在元素小，那么就退出当前比较循环

当mix标签的下标min_index初始值是0，列表元素数量是n，而cur和mix标签的初始值是+1的关系。

所以cur标签移动的范围是1到n-1，那么i的范围就是：

1 ~ n-1 即 min_index+1 ~ n-1 也就是 range(min_index+1,n)

代码实现：

```
n = len(alist)
min_index = 0
# cur 标签元素下标移动范围
for i in range(min_index+1, n):
    if alist[i] < alist[min_index]:
        min_index = i
```

代码测试一下（打印最小元素）：

```
def xuanze(alist):
    n = len(alist)
    ...
    # 打印出最小元素
```

```
        print(alist[min_index])

    if __name__ == '__main__':
        li = [11,3,6,33,5,8,2,88]
        print(li)
        xuanze(li)
```

关键点：

1、mix标签初始位置：　min_index = 0

2、元素比较：　if alist[i] < alist[min_index]:

3、mix移动到小元素：　min_index = i

4、比较次数范围确定：　for i in range(min_index+1,n):

2、元素替换

元素替换，就是将获取到的无序队列中最小元素和第一个位置元素交换位置，而第一个位置就是mix标签初始化时候，min_index所在的位置，所以就把这两个元素替换即可

正常情况下：当第一次退出循环的时候，mix标签所在位置就是无序队列的最小值(下标是i)，那么就把这个值和队首的元素[下标是0]进行互换，队首的元素就是已排序列，其他的元素就是未排序列

但是，元素替换的前提就是，小元素和第一个位置的元素不是一个，也就是说：移动后的 min_index不能是它初始化时候的值

```
min_index != 0
```

然后，继续进行下一轮的比较循环即可，先初始化mix和cur标签，然后比较...（见1.1-1.6步骤）

代码实现

```
    n = len(alist)
    min_index = 0
    ...
    # 保证最新的min_index不在无序队列首位，那么就将它和无序队列的首个元素进行替换
    if min_index != 0:
        alist[0], alist[min_index] = alist[min_index], alist[0]
```

关键点：

替换前提是mix标签确实移动了： min_index != 0

元素交换位置：alist[0], alist[min_index] = alist[min_index], alist[0]

问题：0 只能用一次

3、选择循环

需要进行多少次元素替换，才最终形成一个新的有序队列，元素替换无非就是mix标签初始化地址的范围罢了，有几次初始化，就有几次替换，所以对于选择循环次数，我们只需要确定mix标签初始化值的范围即可。

我们先来回顾一下比较循环：

3.1 ） 在未排序列中，继续执行1-6这步操作，

3.2 ） 将这次最小的值，放到已排序列(即红色框)的队尾，对未排序列进行min和cur的标签标识

3.3 ） 多次执行2.1-2.2，最终形成一个完整的有序队列

通过上面的3.1-3.2的描述，我们确定了选择排序的次数和mix标签移动的范围是存在一定的关系的。

根据我们观察，每次选择排序，mix标签的初始位置j一直在移动，如果当前列表的长度是n，规律如下：

循环排序次数m： 1 2 3 4 5 6 7 8

mix初始位置j： 0 1 2 3 4 5 6 7 范围是 for j in range(n-1)

也就是说，mix标签的初始值位置是根据排序循环的次数而同步变化的，所以我们在获取mix初始位置j的时候，就可以直接从排序循环的范围中获取出来，所以排序循环和mix标签初始值关系如下：

代码实现：

```
# 当前列表的长度
n = len(alist)
# 定义外围排序循环次数
for j in range(n - 1):
    # 设定 min_index 的初始值为 j
    min_index = j
```

我们知道：cur的下标范围是 range(min_index+1,n)，所以代码同步变动为 range(j + 1, n)。

```
# cur 的下标移动范围，因为 j 是从 0 开始的，所以 i 的范围应该是 j+1~n
for i in range(j + 1, n):
    ...
```

相应的，无序列表最小元素移动替换时候使用到了mix的初始值，所以也要做相应的修改

```
# 如果 min_index 的元素不在队列首位，那么就将他和队列的首个元素进行替换进行交换
if min_index != j:
    alist[j], alist[min_index] = alist[min_index], alist[j]
```

最终代码实践

代码实践：

```
def selection_sort(alist):
    # 当前列表的长度
    n = len(alist)
    # 定义外围循环次数
    for j in range(n - 1):
        # 定义 min_index 的初始值为 j
        min_index = j
        # cur 的下标移动范围
        for i in range(j + 1, n):
            # 找到最小元素
            if alist[i] < alist[min_index]:
                min_index = i
        # mix 标签元素和无序队列首位置元素替换
        if min_index != j:
            alist[j], alist[min_index] = alist[min_index], alist[j]
```

关键点：

1、mix标签初始化：min_index = j

2、比较循环的范围：for i in range(j+1,n)

3、元素替换的条件：if min_index != j

4、排序次数范围的确定：for j in range(n-1)

时间复杂度

最优时间复杂度：O(n²)

对于无序队列选取最小元素时候，所有数值都进行了比较。所以内部的最优时间复杂度是O(n)

对于选择排序的外部循环，只和无序列表元素个数相关，元素个数是n个，,所以外部的最优时间复杂度是O(n)

所以整体来说，对于选择排序的时间复杂度就是O(n²)

最坏时间复杂度：O(n²)

因为最好的时间复杂度就是O(n²)了，所以最坏的时间复杂度还是O(n²)

稳定性：稳定

稳定性举例队列：

第一次替换：首个77和10进行替换

第二次替换：第二个77和20进行替换

可以看到：

原队列和第二次替换后的队列，两个77的位置已经替换了，所以对于选择队列来说，肯定不稳定。

思考：

改那个地方，结果是降序？

```
if alist[i] < alist[min_index]: 代码中的 < 改为 >
```

4.1.4 插入排序

插入排序简介

插入排序也是一种简单的排序算法。

简单来说，先定义一个有序队列，然后把无序队列中的第一个元素放到有序队列的合适位置，重复操作，直至形成一个完整的有序队列

生活实例：

打扑克

插入排序原理

1、构建有序序列

2、选择无序队列的第一个元素，先放在有序队列末尾，然后进行冒泡排序，放到指定的位置

3、循环2步，直到无序队列中所有元素全部进入有序队列的合适位置

特点：

插入vs冒泡：

插入排序的有序队列用到了冒泡算法，

区别：

升序情况下：

冒泡：有序队列在后，无序队列在前，

插入：有序队列在前，无序队列在后

插入vs选择：

选择是遍历未排序队列，将最小的元素移动到有序队列的末尾

插入是把无序队列第一个元素放到有序队列，通过使用冒泡算法，移动到合适的位置

插入排序示意图

1、原始序列：

2、将未排序队列的第一个54放到左侧的有序队列中

3、将未排序队列的第一个元素26，放到左侧有序队列的末尾

接着冒泡排序：26先和54来比，26小，然后和54互换位置，

4、将未排序队列的第一个元素93，放到左侧已排序队列的末尾，然后和前面的54比较，放到合适位置

5、将未排序队列的第一个元素17，放到已排序队列的末尾，经过如下比较，将17放到合适的位置。

注意：

17先和93来比，17小，然后和93互换位置，

17在和54来比，17小，然后和54互换位置，

17在和26来比，17小，然后和26互换位置

6、将未排序队列的第一个元素77，放到已排序队列的末尾，然后经过比较，放到合适位置

...

7、最终效果

总结：

1、刚开始有序队列为空，直接把无序队列的第一个元素放到有序队列里即可

2、如果有序队列有内容，那么把无序队列的第一个元素放到有序队列里，然后有序队列进行冒泡排序

3、需要进行多少次冒泡排序呢？无序队列有几个元素，就进行n-1次有序队列的冒泡排序

通过分析，整个过程包括如下内容：

1、有序队列中元素比较替换　　即"元素替换"

2、每次排序，有序队列中元素比较替换的次数　　即"比较循环"

3、需要进行多少次排序　　即"排序循环"

代码实践分析

这部分的知识点，我们从三个方面来学习：

元素替换、比较循环、排序循环

1、元素替换

将有序的队列中两个数值进行比较，小的放到前面

26是有序列表的最后一个元素(下标是j)，他要和前一个元素54(下标是j-1)进行比较

代码实现：

```
# 有序列表两个元素进行比较
if alist[j] < alist[j-1]:
    # 大小数值元素进行替换
    alist[j], alist[j-1] = alist[j-1], alist[j]
# 条件不满足，大小元素不替换
else:
    break
```

2、比较循环

有序队列中的最后元素需要比较多少次才能放到合适的位置？

经过分析：

有序队列中末尾新元素17(即无序队列的首元素)，假设初始下标是i，在冒泡排序过程中，位置变化如下：

alist[i]->alist[i-1]->alist[i-2]->alist[i-3]

也就是说，新元素的初始下标是i在排序过程中是一个降序的状态，它的取值范围是

i,i-1,i-2,...,1　即 range(i,0,-1)

问：为什么i最终会到1？

因为alist[i] 还要和前一个元素 alist[i-1]比较，而 i-1 >= 0，所以i最小值就是1

所以，元素替换时候下标j的范围应该是 range(i,0,-1)

代码实现：

```
# 比较循环次数的确定
for j in range(i,0,-1):
    # 元素替换条件判断
    if alist[j] < alist[j-1]:
        ...
```

3、排序循环

有序队列需要经过多少次元素的增加？

答：无序队列有多少个元素就往有序队列增加多少次元素。

通过上面的2分析，无序列表队列中的第一个元素的下标是i，如果无序队列的长度是n，那么i的取值范围是：

0,1,2,3,...,n-1　即 range(n)

而i每变化一次，都要进行一次有序队列元素的比较循环

所以i的取值范围是 range(n)

代码实现：

```
# 无序队列元素数量
n = len(alist)
# 有序队列循环的次数
for i in range(0,n):
...
```

代码实践

代码实践：

```
def insert_sort(alist):
    # 无序队列元素数量
    n = len(alist)
    # 有序队列循环的次数
    for i in range(0, n):
        # 比较循环次数的确定
        for j in range(i, 0, -1):
            # 假设 26 的下标是 j，那么 54 的下标为 j-1
            if alist[j] < alist[j - 1]:
                # 大小数值元素进行替换
                alist[j], alist[j - 1] = alist[j - 1], alist[j]
            # 否则的话，大小元素不替换
            else:
                break
```

关键点：

1、元素替换：if alist[j] < alist[j-1]

2、比较循环：for j in range(i,0,-1)

3、插入循环：for i in range(n):

时间复杂度

最优时间复杂度：O(n) （升序排列，序列已经处于升序状态）

如果有序队列里的元素正好是已排序的状态，不需要比较替换，那么内部的比较循环的时间复杂度就是O(1)

无序队列中有多少元素，就要进行多少次"比较循环",所以外部排序循环的时间复杂度是O(n)

所以整体来说：最优时间复杂度是O(n)

最坏时间复杂度：$O(n^2)$

如果内部比较循环，需要全部排序，则它的最坏时间复杂度就是O(n)

所以整体来说：最坏时间复杂度是$O(n^2)$

稳定性：稳定

通过示例可以得知，有序队列里面的元素顺序没有做任何变动，所以稳定

思考：

改那个地方，结果是降序？

if alist[j] < alist[j - 1]: 代码中的 < 改为 >

4.1.5 希尔排序

希尔排序简介

希尔排序(Shell Sort)是插入排序的一种。也称缩小增量排序，是插入排序算法的一种高效的**改进**版本。

希尔排序原理：

1、第1次希尔，两两分组，根据队列元素个数获取组内元素之间的偏移量

分组方式是：0-4、1-5、2-6、3-7

也就是说：i和i+4是一组，4称为下标偏移量

2、对组内元素间进行插入排序(即元素替换)

3、第2次希尔，四四一组，组内插入排序。即组内元素间偏移量是上一次标偏移量/2

分组方式：0-2-4-6、1-3-5-7

4、第3次希尔，八八一组，组内插入排序。偏移量同上

5、循环下去，直到所有元素为一组(即组内元素下标偏移量为1)。

一句话：

两两一组、四四一组、八八一组...，直到所有元素为一组，进行排序

特点：

下标增量分组，对小组元素进行插入排序

下标增量的特点：

第一次分组，gap=n/2 ，

从第二次分组，gap=gap/2，

最后一次分组gap=1

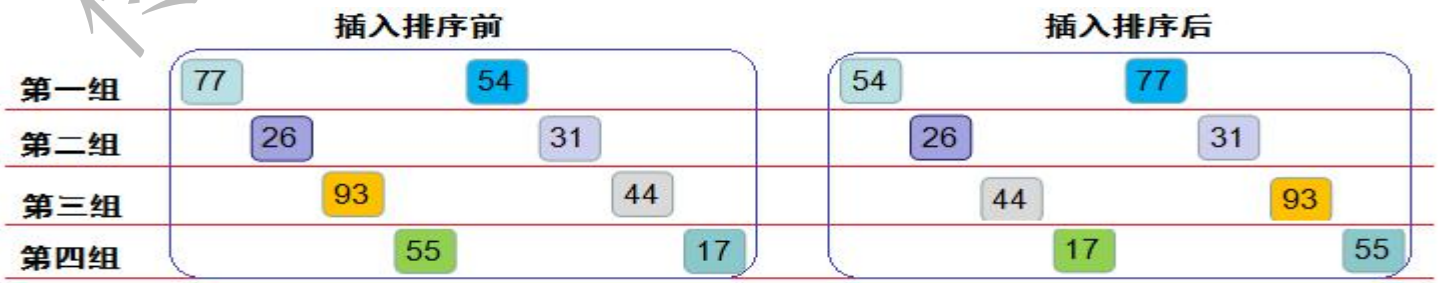
整个分组过程就是:递归

示意图

原始队列：

第一次分组和插入排序：

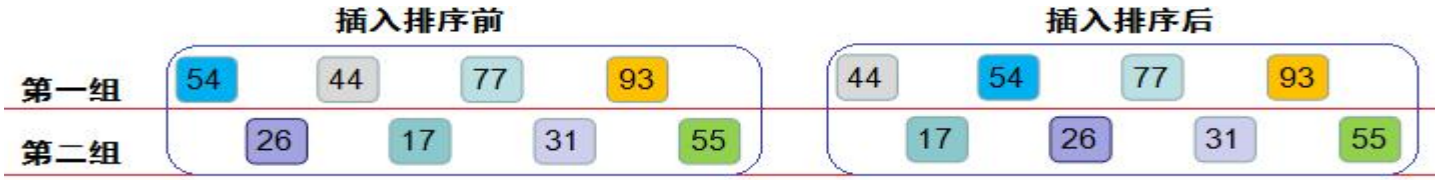
分组特点：下标+4 为一组



第一次分组和插入排序后效果：

第二次分组和插入排序：

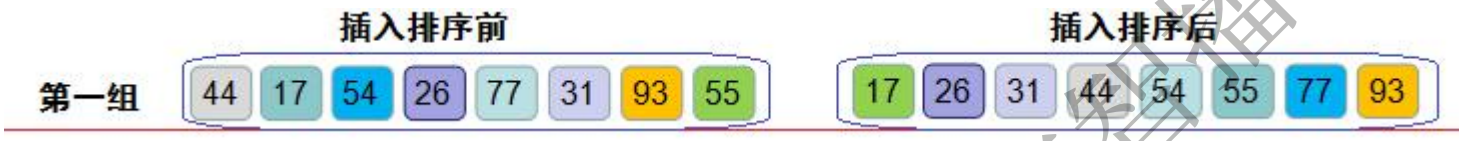
分组特点：下标+2 为一组



第二次分组和插入排序后效果：

第三次分组和插入排序：

分组特点：下标+1 为一组



最终排序效果：

整体感觉就是一个词：**折腾**

但是操作完毕后，我们隐隐的感觉到一个字，爽！！！

比如说：

插入排序：将队列末尾的17放到最前面，需要先进行7次循环，在第8次循环中，进行7次比较替换操作

希尔排序：将队列末尾的17放到最前面，需要先进行2次循环，在第3次循环中，进行1次比较替换操作

通过分析，整个希尔排序过程包括如下内容：

1、分组队列中元素比较替换　　即"元素替换"

2、每次分组后，同时有几组在进行比较　　即"比较次数"

3、需要进行多少次分组　　即"分组次数"

代码实践分析

这部分知识，我们从三个方面来学习这块的知识：

元素替换、比较次数、分组次数

1、 元素替换

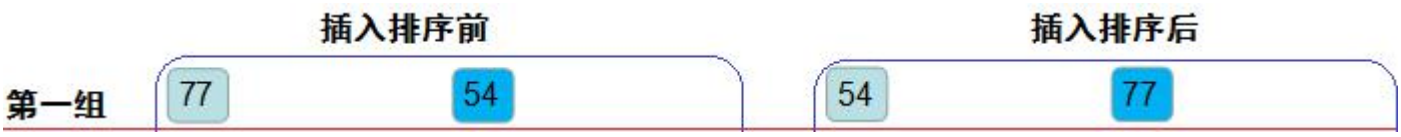
如果当前队列的长度是n，对于第一次分组，下标的偏移量gap是当前队列长度的一半，也就是n/2.所以gap的初始值是n/2

第一组的元素是54【下标是i】和77【下标是i-gap】，进行比较替换

思考点：

1.1、元素的替换

1.2、元素下标值



1.1 元素的替换

对于元素的替换，其实非常简单，只需要让两个结点的值进行大小匹配，谁小，谁在前面

54和77换位置，那么54的下标i也要和77的下标(i-gap)进行替换，即 i = i - gap，为什么？

因为插入排序的过程跟踪特性，保证了alist[i]永远要指向54元素

代码实现

```
# 组内大小元素进行替换操作
if alist[i] < alist[i-gap]:
    alist[i],alist[i-gap] = alist[i-gap],alist[i]
    # 修改 i 的属性重新指向老元素
    i = i - gap
# 否则的话，不进行替换
else:
    break
```

1.2 元素的下标：

关于元素的下标我们要考虑1个方面：

下标的范围值：

下标的范围必须大于0，所以只要下标 i-gap >= 0 ，即 while (i-gap) >= 0

```
# 对移动元素的下标进行条件判断
while (i-gap) >= 0:
    ...
```

2、比较次数

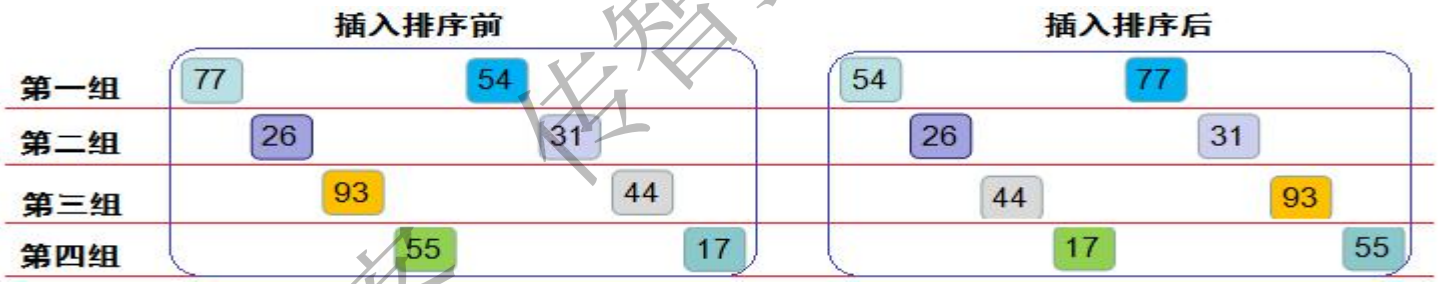
根据我们写的代码可以发现：

i = 4 ，表示对第一组进行插入排序

i = 5 ，表示对第二组进行插入排序

i = 6 ，表示对第三组进行插入排序

...



经分析，我们这一段代码表示序列中有多少分组进行了插入排序，也就是说i的范围该怎么确定呢？

首先，如果当前列表长度是n，所以 i < n-1

其次：i-gap >=0，所以 i >= gap

所以i的范围就是 gap ~ n-1 ，即：for i in range(gap,n)

代码实现：

```
# 获取列表的长度
n = len(alist)
# 指定 i 下标的取值范围
for i in range(gap,n):
    while (i-gap) >= 0:
      ...
```

3、分组次数

整个希尔排序需要进行几次分组？

在这个问题中，包含了两个问题：

3.1、怎么分组

通过我们对希尔排序的示意图的分析，可以知道，

第1次的分组是在队列长度的基础上进行了/2　　gap = 4

第2次的分组是在第1次分组的基础上进行了/2　　gap = 2

...

3.2、分几次组？

根据我们对之前对递归的理解，要保证最后一次/2是1，也就是说要保证偏移gap>=1

代码实现：

```
n = len(alist)
# 获取 gap 的偏移值
gap = n//2
# 只要 gap 在我们的合理范围内，就一直分组下去
while gap >= 1:
    # 执行分组中的插入排序
    ...
    # 每执行完毕一次分组内的插入排序，对 gap 进行/2 细分
    gap = gap//2
```

代码实践

代码实践：

```
def shell_sort(alist):
    # 获取列表的长度
    n = len(alist)
    # 获取 gap 的偏移值
    gap = n // 2
    # 只要 gap 在我们的合理范围内，就一直分组下去
    while gap >= 1:
        # 指定 i 下标的取值范围
        for i in range(gap, n):
            # 对移动元素的下标进行条件判断
            while (i - gap) >= 0:
                # 组内大小元素进行替换操作
                if alist[i] < alist[i - gap]:
                    alist[i], alist[i - gap] = alist[i - gap], alist[i]
                    # 更新迁移元素的下标值为最新值
                    i = i - gap
                # 否则的话，不进行替换
                else:
                    break
        # 每执行完毕一次分组内的插入排序，对 gap 进行/2 细分
        gap = gap // 2
```

关键点：

1、元素替换：

下标范围：while (i - gap) >= 0:

替换条件：if alist[i] < alist[i-gap]:

过程跟踪：i = i - gap

2、比较循环：

元素的范围：for i in range(gap,n):

3、递归分组循环

偏移量初始值：gap = n // 2

递归循环的退出条件：while gap >= 1

gap偏移量规律：gap = gap // 2

时间复杂度

最优时间复杂度：根据步长序列的不同而不同

最内部的元素进行插入排序，我们知道插入排序的最优时间复杂度是O(n)

外部分组遇到了递归拆分，那么我们知道递归拆分的时间复杂度是O(logn)。

整体来说：最优的时间复杂度是 O(nlogn)

然而分组/2是一种情况，如果分组直接分成最细的粒度，也就是说每一个元素都是一个组，所以外部的时间复杂度就变成了是O(n)

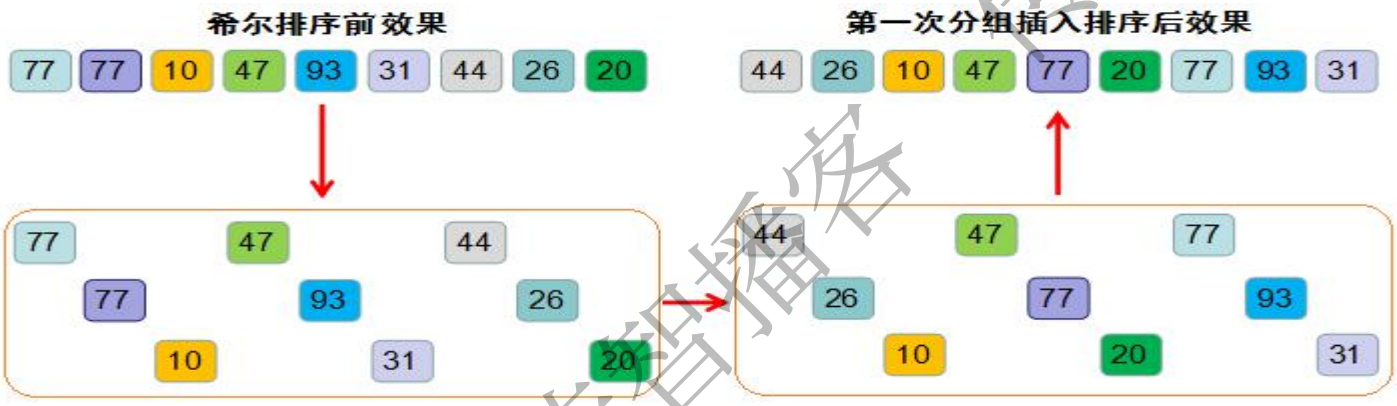
所以整体时间复杂度就变成了$O(n^2)$

结合两个方面：最优时间复杂度就是 O(nlogn)~$O(n^2)$

最坏时间复杂度：O(n2)

我们知道希尔排序的本质就是插入排序，所以最坏也就是插入排序的最坏时间复杂度了，也就是$O(n^2)$

稳定性：不稳定

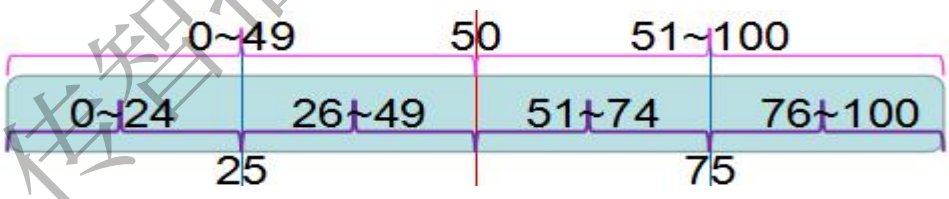


4.1.6 快速排序

快速排序简介

快速排序，又称划分交换排序，从无序队列中挑取一个元素，把无序队列分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

简单来说：挑元素、划分组、分组重复前两步

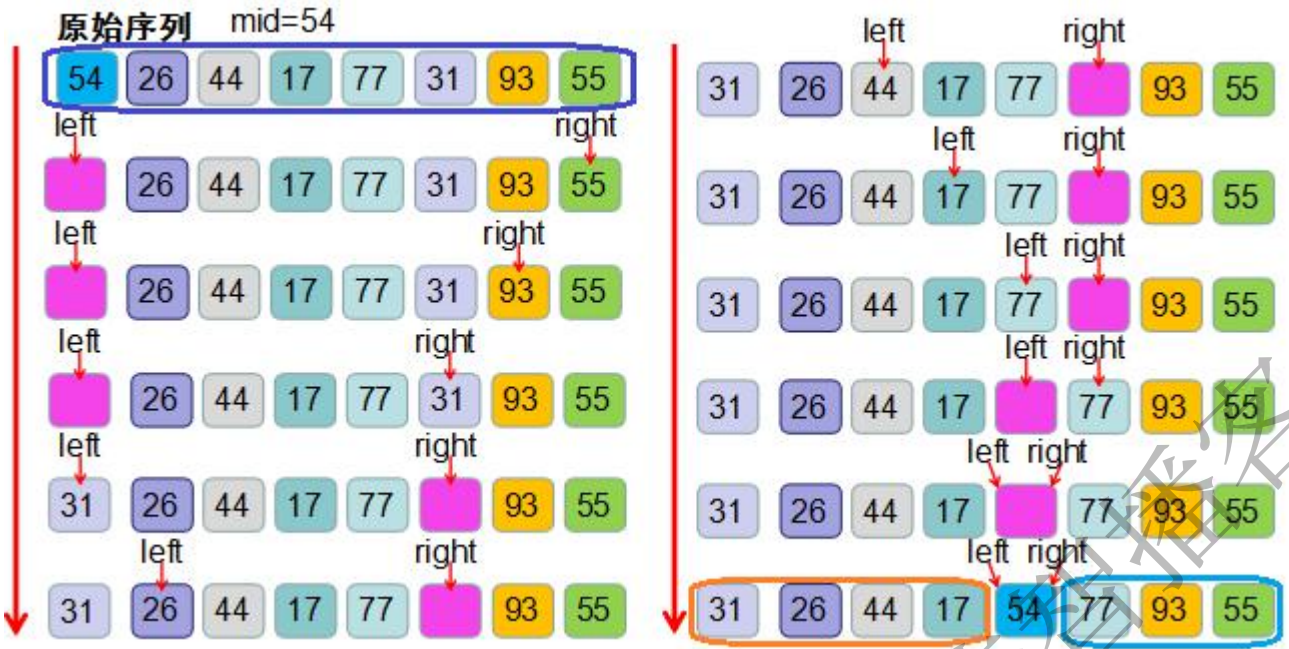


快速排序原理示意图

通过上面对快速排序的简介，我们知道了，快速排序主要包括以下两方面：

挑元素划分组、整体递归分组

挑元素划分组示意图：



特点：

1、因为是无序队列，所以位置可以随机挑

2、临时划分一个空间，存放我们挑选出来的中间元素

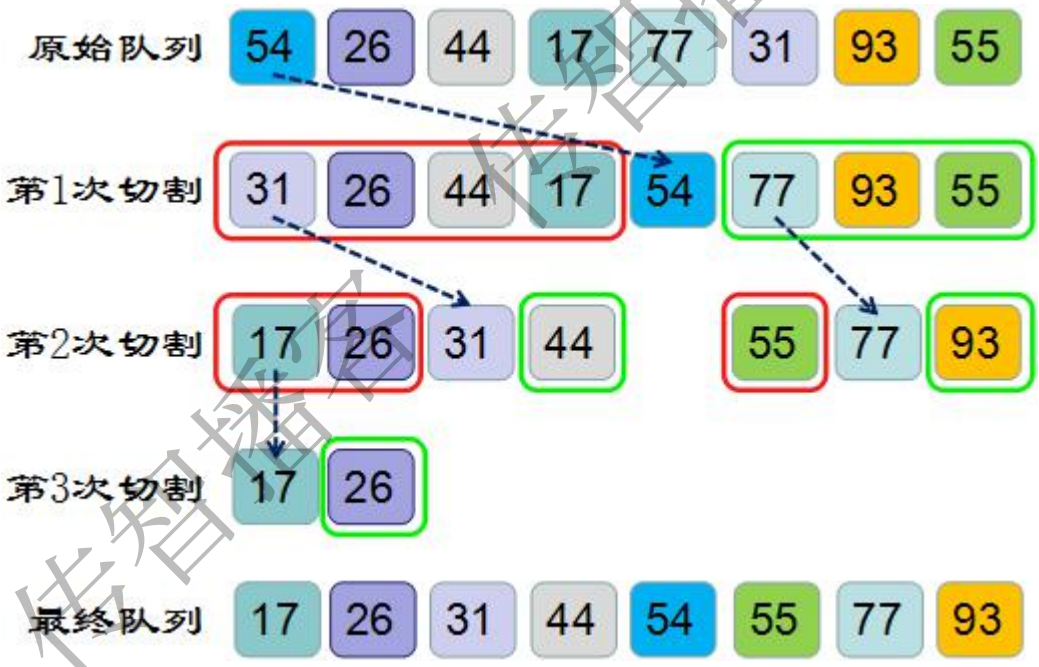
3、左标签位置空，移动右标签，反之一样

4、重复3，直到左右侧标签指向同一个位置，

5、把临时存放的中间元素，归位

一句话：左手右手一个慢动作，右手左手慢动作重播

整体划分示意图：



特点：

1、递归拆分

2、拆分到最后，所有小组内的元素个数都是1

一句话：递归拆分到不能再拆

代码实践分析

根据上面两个示意图的分析，我们要从两个大方面分析：

序列切割 和 递归拆分

1、序列切割

序列切割这个知识点，我们从四个方面分别介绍：

3个基本标签、右侧推进、左侧推进、停止推进(即元素归位)

1.1、3个基本标签

大小区域切割，至少涉及到三个标签：

mid：指定要切割的临时中间数字

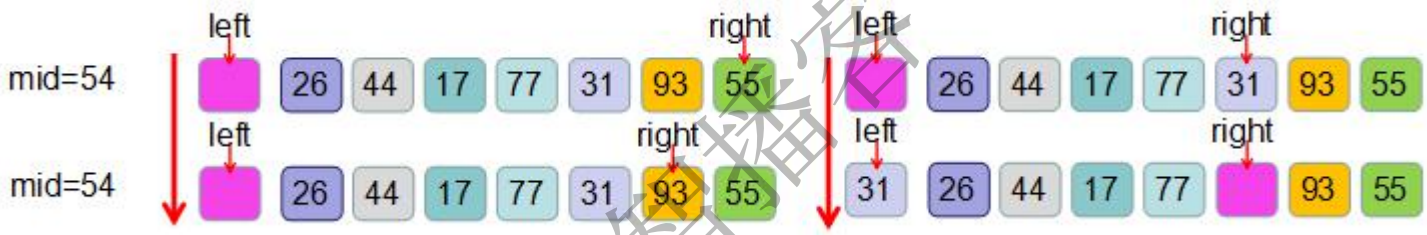
left：从队列左侧推进的标签

right：从队列右侧推进的标签

代码实现：

```
# 定义三个标签
mid = alist[0]
left = 0
right = len(alist)-1
```

1.2、右侧推进



前提：

1、left永远小于right

推进过程：

如果right标签元素比mid指定的元素大或者一样，那么right标签左移一位

如果right标签元素比mid指定元素小，那么left标签元素设置为right标签元素

代码实现：

```
# right 元素大于 mid 值，左移 right 标签
while right > left and alist[right] >= mid:
    right -= 1
# right 元素小于 mid 值，left 标签元素设置为 right 标签元素
alist[left] = alist[right]
```

1.3、左侧推进

前提：

1、left永远小于right

移动过程：

如果left标签元素比mid指定的元素小，那么left标签右移一位

如果left标签元素比mid指定元素大，那么把right标签所在的元素设置为left标签元素

代码实现：

```
# left 元素小于 mid 值，右移 left 标签
```

```
    while left < right and alist[left] < mid:
        left += 1
    # left 元素大于 mid 值，right 标签元素设置为 left 标签元素
    alist[right] = alist[left]
```

1.4、左右标签停止推进(即元素归位)

left和right标签有两种情况：

left < right 情况下，左推进完右推进，右推进完左推进，循环下去

left = right 情况下，左右推进完毕，连个标签都指向一个位置，这个位置就是无序队列的中间位置，即 mid元素最终的"归宿",接着把将mid值交给left或者right标签所在位置元素即可

代码实现：

```
    while left < right:
        # 左右推进代码
    # 退出循环，表示 left 和 right 标签合并到一起了
    alist[left] = mid
```

(可以验证一下)

2、递归拆分

这块我们从两个方面来学习：递归拆分，递归退出条件

2.1、递归拆分

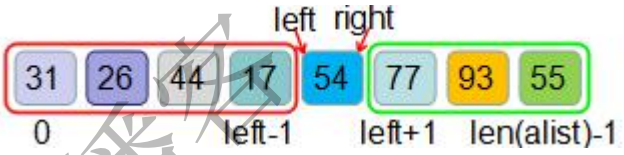
递归拆分就是在拆分后的小组上再次进行快速排序，根据我们对拆分的理解：

对无序队列拆分过程中，我们用两个标签left和right来指定了队列的两侧边界，那么要实现对子小组递归拆分，我们就需要考虑两个方面：

小组边界的确定 和 递归功能实现

2.1.1 小组边界的确定

获取小组边界其实是很简单的：



红色小组：

左侧边界start：0　右侧边界end：left-1

绿色小组：

左侧边界start：left+1　　右侧边界end：len(alist)-1

2.1.2 递归功能实现

说到递归，不能不提"函数自调用"，在函数内部调用自己，实现对内部小组再次拆分的目的。

结合刚才小组边界的确定，所以我们在调用拆分函数本身的时候，需要将小组的两个边界作为参数传输进去即可

综合上面两个分析，代码效果如下：

```
    # 增加两个参数，左边界 start，右边界 end
    def quick_sort(alist, start, end):
        # 因为 mid 指定的是传入列表的左边界元素
        mid = alist[start]
```

```
        left = start
        right = end
        # 进行左右推进操作
        ...

        # 对切割后左边的子部分进行快速排序
        quick_sort(alist, start, left-1)
        # 对切割后左边的子部分进行快速排序
        quick_sort(alist, left+1, end)
```

2.3 递归退出

根据我们对递归拆分的理解：只要列表元素大于一个，即左边界start小于右边界end，我们就拆分下去，一旦start=end，即只有一个元素的时候，就退出。

代码实现：

```
    def quick_sort(alist, start, end):
        # 递归退出条件
        if start < end:
            # 执行切分操作
```

代码实践

实践代码：

```
    def quick_sort(alist, start, end):
        # 定义递归条件
        if start < end:
            # 定义三个标签
            mid = alist[start]
            left = start
            right = end

            # 定义拆分条件
            while left < right:
                # 右推进
                while right > left and alist[right] >= mid:
                    right -= 1
                alist[left] = alist[right]
                # 左推进
                while left < right and alist[left] < mid:
                    left += 1
                alist[right] = alist[left]
            # 获取中间值
            alist[left] = mid

            # 对切割后左边的小组进行快速排序
            quick_sort(alist, start, left-1)
            # 对切割后左边的小组进行快速排序
            quick_sort(alist, left+1, end)
```

测试代码：

```
    if __name__ == "__main__":
        li = [54,26,93,17,77,31,44,77,20]
        print(li)
```

```
    quick_sort(li,0,len(li)-1)
    print(li)
```

关键点：

序列切割：

1、挑中间元素：mid = alist[start]

2、右推进：while right > left and alist[right] >= mid:

3、左推进：while left < right and alist[left] < mid:

4、推进循环：while left < right:

5、元素归位：alist[left] = mid

递归拆分：

1、小组边界确定：left = start、right = end

2、递归退出条件：if start < end:

3、函数自调用：quick_sort(alist, start, end)

时间复杂度

最优时间复杂度：O(nlogn)

对于每次快排，left和right的标签分别在左右两册数据全部都移动了一遍，相当于遍历了所有数据，那么时间复杂度是O(n)

因为涉及到了递归分组，所以他的时间复杂度是O(logn)

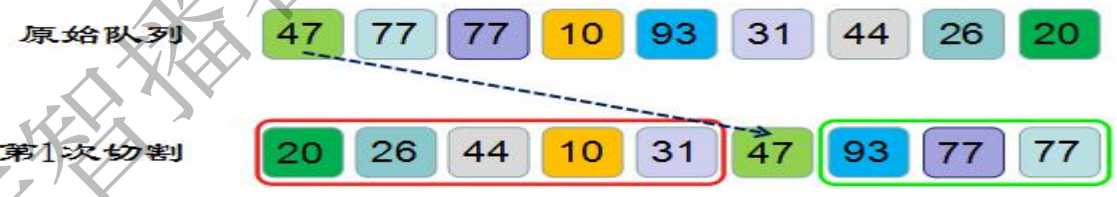
整体来说：最优的时间复杂度是 O(nlogn)

最坏时间复杂度：O(n2)

因为递归分组分组的条件不一定是一分，有可能每一次mid指定的都是最大或者最小，那么有多少个元素，我们就可能分多少次组，这种情况时间复杂度就是O(n)了

所以最坏的时间复杂度就是O(n^2)，那么最坏也不过如此了。

稳定性：不稳定



思考：

改哪个地方，结果是降序？

while right > left and alist[right] >= mid: 代码中的 >= 改为 <=

while left < right and alist[left] < mid 代码中的 < 改为 >

4.1.7 归并排序

归并排序简介

归并排序是采用分治法的一个非常典型的应用。

将无序队列拆分成两个小组，组内元素排序，然后组间元素逐个比较，把小元素依次放到新队列中。

汉朝　　魏蜀吴　　朝廷　　打仗　　晋

关键字：拆分、排序、组间、小、新队列
一句话：分组排序，合并新队列

归并排序原理：

归并排序分为两个阶段：

分组排序阶段：

1、将无序队列alist，拆分成两个小组A和B，
2、分别对两个小组进行同样的冒泡排序
3、用标签left和right，分别对小组A和B进行管理

合并新队列阶段：

4、两个标签所在的元素比较大小，
5、将小的元素放到一个新队列中，然后小元素所在的标签向右移
6、多次执行4和5，最终肯定有一个小组先为空
7、把不为空的小组元素，按顺序全部移到新队列的末尾
8、无序队列中的所有元素就在新队列中形成有序队列了

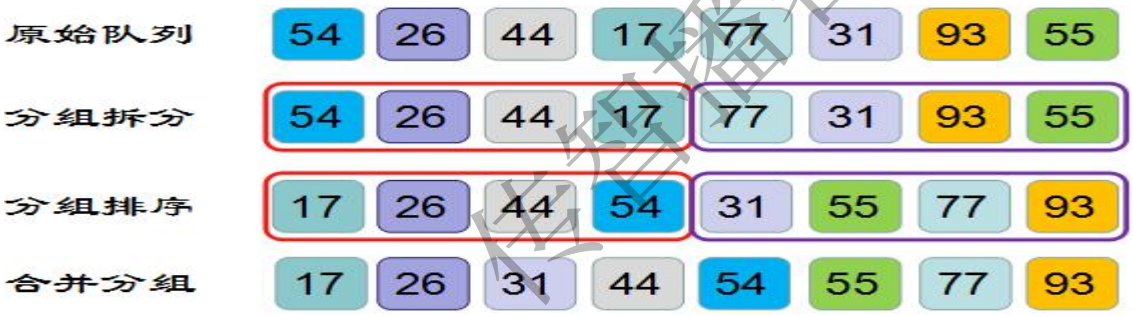
特点：

两个阶段：分组排序+合并
合并策略：组间比较，新增小，小移标

归并排序示意图

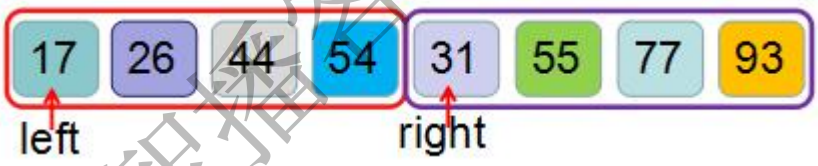
归并排序图示有两种情况：分两组合并排序、递归分组合并排序

归并排序情况一：分两组

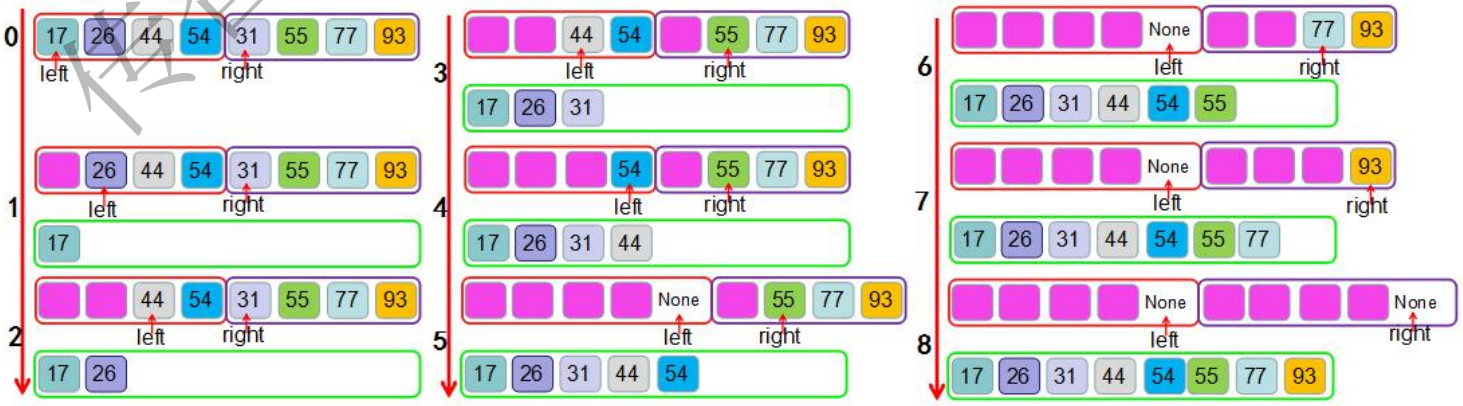


合并分组示意图：

0、分别给两个小组的首部元素添加标签left和right



具体的移动流程如下：



整个过程就是left和right两个标签的元素的对比

1、17和31比，17小，然后将17移动到新队列中，left标签右移一位，指向26

2、26和31比，26小，然后将26移动到新队列末尾，left标签右移一位，指向44
3、44和31比，31小，然后将31移动到新队列末尾，right标签右移一位，指向55
4、44和55比，44小，然后将44移动到新队列末尾，left标签右移一位，指向54
5、54和54比，54小，然后将54移动到新队列末尾，左侧队列为空，left标签指向None
6-8、右侧right标签位置的数据比None大，所以右侧全部数据按顺序移动到列表中。

归并排序情况二：递归分组合并排序

代码实践分析

根据我们对于并归排序的原理讲解，我们知道了整个并归排序分为两部分：分组实现+合并分组的排序策略

1、分组实现

这一部分我们主要考虑三个因素：

首次分组、递归分组、合并分组数据

1.1 首次分组

分组的话，需要考虑两种情况：正常分组和不能分组

正常分组：

将一个无序队列alist分为两部分，队列长度/2获取中间元素的地址mid，

然后通过切片的方式来实现分组

左分组 alist[mid:]，右分组 alist[:mid]

代码实现：

```
# 获取当前序列的长度
n = len(alist)
# 将当前的序列分成两部分,使用切片方式获取两部分内容
mid = n // 2
# 左半部分数据
left = alist[:mid]
# 右半部分数据
right = alist[mid:]
```

不能分组：

如果是个空队列或者队列只有一个元素，是不需要分组的，返回队列内容即可

代码实现：

```
# 准备工作
n = len(alist)
# 队列异常情况
if n <= 1:
    return alist
```

1.2 递归分组

递归分组无非就是在已分小组的基础上继续进行分组，即函数内部调用自身功能

代码实现：

```
def fen_zu(alist):
    # 获取当前序列的长度
    n = len(alist)
    # 将当前的序列分成两部分,使用切片方式获取两部分内容
    mid = n // 2
    # 左半部分数据
    zuo = fen_zu(alist[:mid])
    # 右半部分数据
    you = fen_zu(alist[mid:])
```

1.3 合并分组数据

分组后的数据是给合并数据用的，所以我们要将拆分后的两部分交给一个合并的函数去使用

代码实现：

```
def fen_zu(alist):
    # 对无序队列进行分组
    ...
    # 将分组后的数据交个一个合并数据的函数去处理
    return merge(zuo,you)
```

分组功能完整代码

```
def fen_zu(alist):
    # 准备工作
    n = len(alist)
    if n <= 1:
        return alist
    mid = n // 2
    # 两侧分组进行递归分组
    zuo = fen_zu(alist[:mid])
    you = fen_zu(alist[mid:])
    # 分组合并
    return merge(zuo, you)
```

关键点：

1、异常分组：if n <= 1:

2、递归分组：fen_zu(alist[:mid])

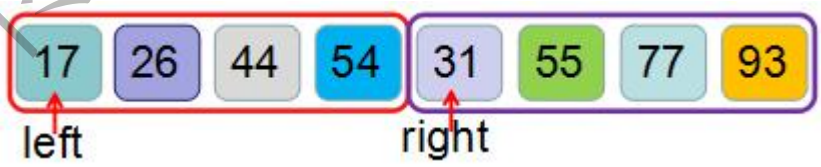
3、分组合并：merge(zuo, you)

2、合并分组的排序策略

关于归并排序，我们从两个方面来学习：

准备工作、空列表增加数据

2.1 准备工作



准备工作阶段我们要考虑三个因素：组标签、组长度、空队列

组标签：

管理分组我们定义两个标签l和r，分别是分组列表的首位下标值0

组长度：

组元素变化时候，两个标签肯定会移动，所以标签的移动范围必须确定，即获取组长度

空队列：

合并分组，需要有一个新队列，我们用空列表result来实现

代码实现：

```
# 定义标签 l 和 r 在两组的位置
l, r = 0, 0
# 定义一个空列表
result = []
# 获取两个分组的长度
zuo_len = len(zuo)
you_len = len(you)
# 指定标签的有效范围
while l < zuo_len and r < you_len:
```

2.2 移动相应数据到空列表

移动数据我们从两个方面来考虑：

1、空列表添加数据

2、一组空，另一组剩余元素按顺序一次性添加到空列表

1、空列表添加数据

比较的元素：

左侧列表left的l标签所在元素zuo[l]

右侧列表right的r标签所在元素you[r]

比较的效果：

小元素添加到空队列result，即result.appent(元素)

小元素标签右移一位：标签 += 1

两种情况：

左侧元素小：代码效果：

```
    # 判断两侧标签指定的数据大小
    if zuo[l] <= you[r]:
        # 将左侧小数据追加到新队列
        result.append(zuo[l])
        # left 标签右移一位
        l += 1
```

右侧元素小，代码效果：

```
    else:
        # 将右侧小数据追加到新队列
        result.append(you[r])
        # right 标签右移一位
        r += 1
```

3、一组空，另一组剩余元素按顺序添加到新队列

组元素为空，该组标签指向None，标签操作范围条件不成立，退出操作循环.就会出现两种情况：

如果zuo队列空，you队列剩余元素you[r:]

如果you队列空，zuo队列剩余元素zuo[l:]

把剩余元素添加到新队列result，因为都是列表，所以直接拼接即可。两种拼接情况：

左队列空，新队列添加右侧元素： result += you[r:]

右队列空，新队列添加左侧元素： result += zuo[l:]

最后返回新队列result内容

代码实现：

```
# 将左侧的剩余内容，一次性添加到 result 表中
result += zuo[l:]
# 将右侧的剩余内容，一次性添加到 result 表中
result += you[r:]
# 返回 result 表
return result
```

归并功能完整代码

```
def merge(zuo, you):
    # 准备工作
    l, r = 0, 0
    result = []
    # 获取分组的长度
    zuo_len = len(zuo)
    you_len = len(you)

    # 元素比较条件
    while l < zuo_len and r < you_len:
        # 左队列移动元素到新队列
        if zuo[l] <= you[r]:
            result.append(zuo[l])
            l += 1
        # 右队列移动元素到新队列
        else:
            result.append(you[r])
            r += 1

    # 剩余内容添加到 result 表中
    result += zuo[l:]
    result += you[r:]
    # 返回 result 表
    return result
```

测试代码：

```
if __name__ == "__main__":
    li = [54,26,93,17,77,31,44,77,20]
    print("处理前: %s" % li)
    sortlist = fen_zu(li)
    print("处理后: %s" % li)
    print("新列表: %s" % sortlist)
```

关键点：

1、数据比较条件：while l < zuo_len and r < you_len:

2、小元素移动：result.append(zuo[l])

3、小元素标签处理：l += 1

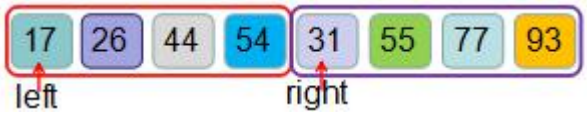
4、异常情况：result += zuo[l:]

5、最终效果：return result

时间复杂度

最优时间复杂度：O(nlogn)

对于每次合并，left和right的标签分别在左右两组数据全部都移动了一遍，相当于遍历了所有数据，那么时间复杂度是O(n)



对于分组来说，因为他是递归，所以他的时间复杂度是O(logn)

整体来说：最优的时间复杂度是 O(nlogn)

最坏时间复杂度：O(nlogn)

最优的时间复杂度已经到了O(nlogn)，那么最坏也不过如此

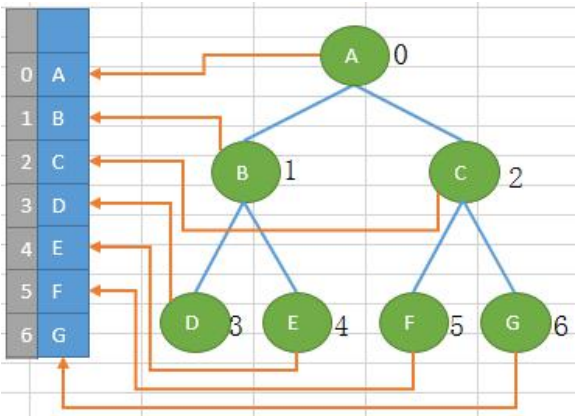
稳定性：稳定

思考：

改哪个地方，结果是降序？

if zuo[l] <= you[r]: 将 < 改为 > 即可

4.1.8 堆排序



堆简介

堆是采用顺序表存储的一种近似完全二叉树的结构。

父节点和子节点关系：

父节点位置 i，找子节点：

左子节点位置：2i + 1　　右子节点位置：2i + 2

堆分类：

列表 li = [A,B,C]，二叉树的根节点编号从0开始。

A节点的标号是i，左侧子节点B下标2i+1，右侧子节点c下标2i+2

大顶堆：任一节点都比其孩子节点大

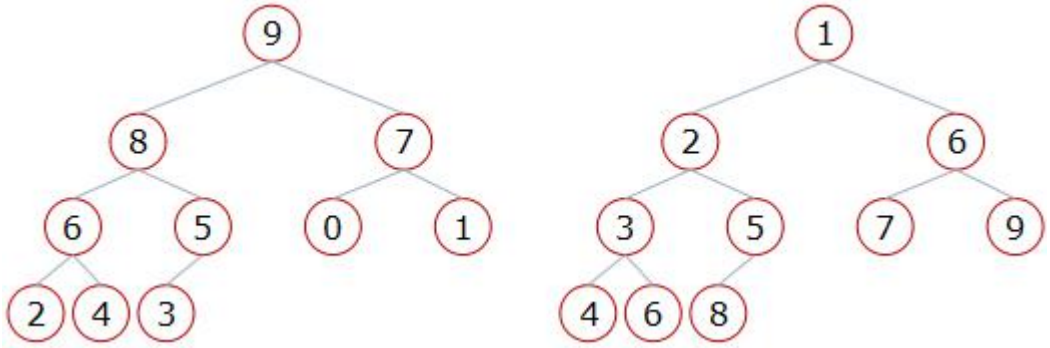
li[i] > li[2i+1] 且 li[i] > li[2i+2]

父　　左子　　父　　右子

小顶堆：任一节点都比其孩子节点小

li[i] < li[2i+1] 且 li[i] < li[2i+2]

父　　左子　　父　　右子



堆排序原理

它是指利用堆这种树结构所设计的一种排序算法。

简单来说，就是将无序列表先构造一个有特点的堆，然后利用列表的特点快速定位最大/小的元素，将其放到一个队列中。

1、根据完全二叉树结构，将无序队列构造成一个大顶堆，

2、将堆顶的根节点移走，与无序列表的末尾元素交换，此时末尾元素就是最大值，相当于进入了一个有序队列。

3、再将剩余的n-1个无序队列重新构造一个同样堆，

4、重复循环1-3步，最终将所有元素都移动到有序队列。

特点：

无序队列构建一个堆，堆顶和堆尾元素替换位置

重新构建堆，堆顶和堆尾元素替换位置，

...

简单来说：头尾替换，恢复堆后再继续

堆排序实现步骤

以大顶堆为例：

1.构建一个堆

从最后一个有子节点的节点开始构建，他的下标是[n/2 - 1]

2.堆的调整

移除堆顶元素之后，然后用队列中最后一个元素填补它的位置，自上向下进行调整：

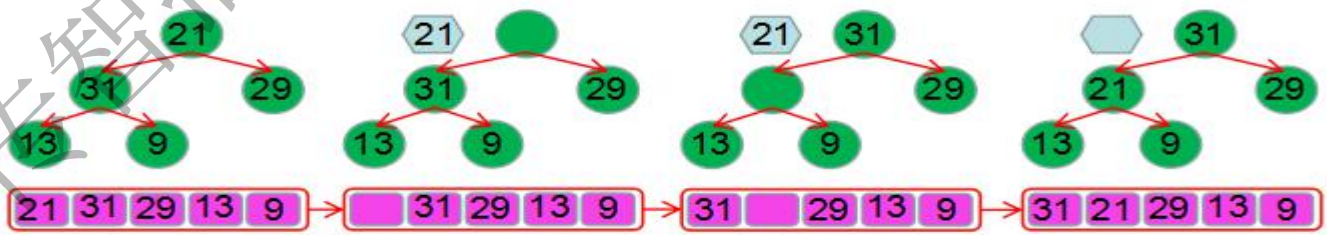
首先将临时堆顶元素和它的左右子结点进行比较，把最大的元素交换到堆顶；

然后顺着被破坏的路径一路调整下去，直至叶子结点，就得到新的堆。

所以构建一个堆，无非就是多个循环的堆调整

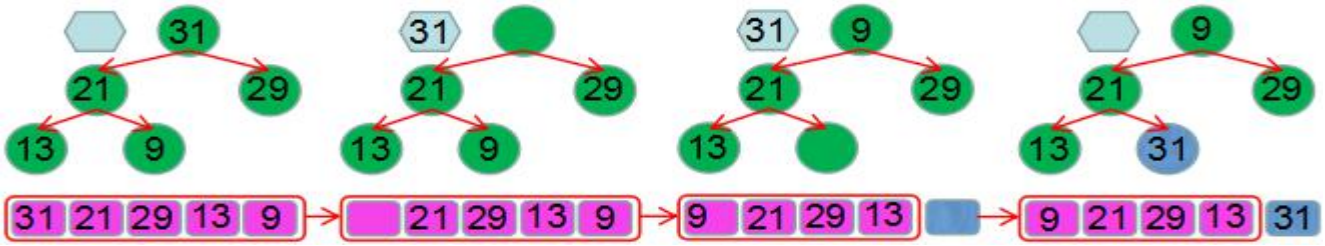
堆队列排序原理：

堆的调整

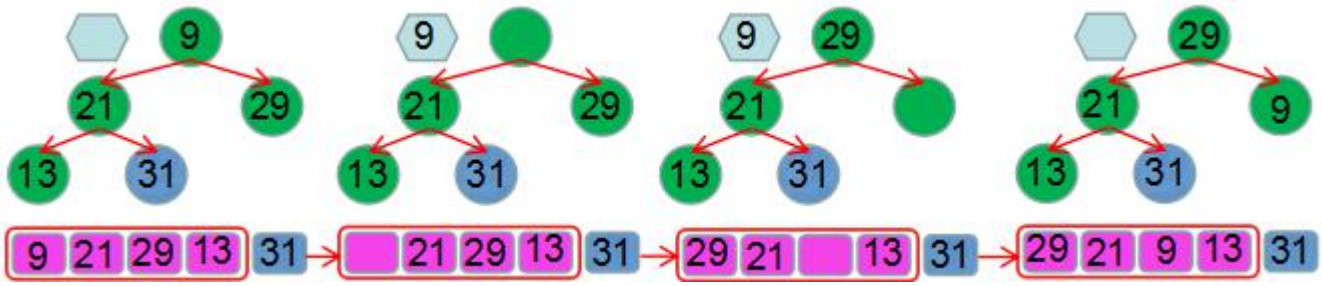


排序原理：

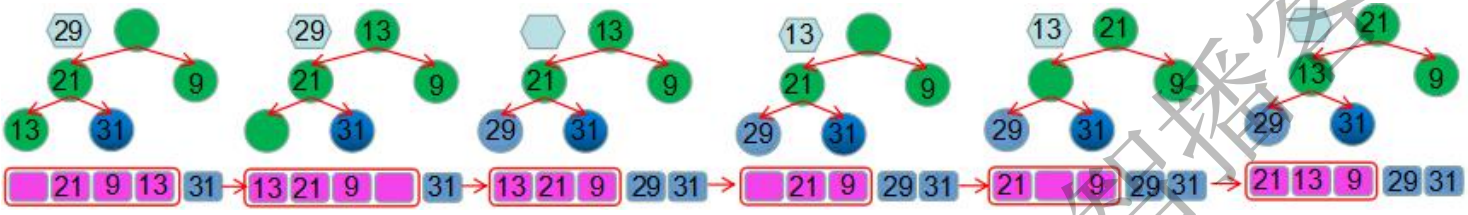
1、移除堆顶元素之后，然后用表中最后一个元素填补它的位置



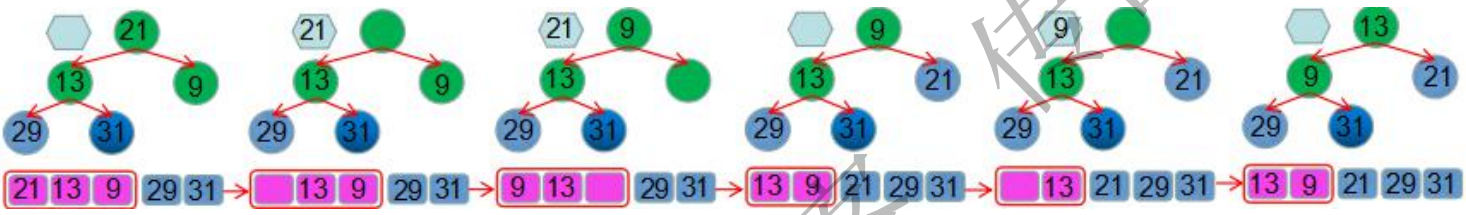
2、将临时堆顶元素和它的左右子结点进行比较，把最大的元素交换到堆顶



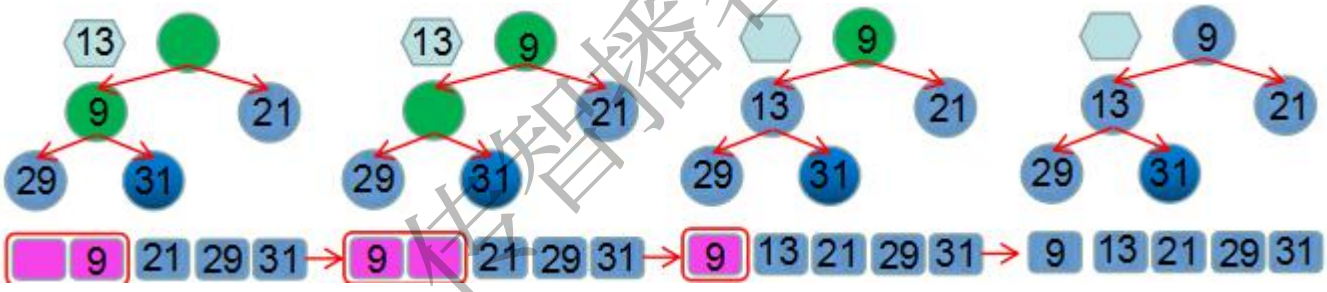
第二次循环1-2步



第三次循环1-2步



第四次循环1-2步



代码分析

根据我们的分析，堆排序需要三步：

堆的构建和堆顶元素排序、堆调整，

但是堆的调整和堆的构建其实是一个事情，就是形成一个完整堆结构，所以我们从两个方面来说

堆调整和堆顶元素排序

1、堆的调整：

关于堆的调整我们从以下两个方面来分析：

准备工作、堆排序

1.1、准备工作

1） 堆调整是将一个无序队列构建一个完整的堆，所以，要传入一个队列参数：data

2） 堆调整的时候，元素会临时移除，并最终回到队列，所以我们需要一个临时空间存放这个值

堆顶元素的确定：传入一个堆顶队列的下标low

堆顶元素的临时存放空间：tmp

3） 堆排序过程中，有序队列中的数据会慢慢增加，所以无序队列长度会变化，所以我们在进行堆调整的时候，要知道无序队列的范围，即无序队列中元素的最大下标值：high

简单来说：就是要确定三个内容：无序列表范围和空列表

代码实现：

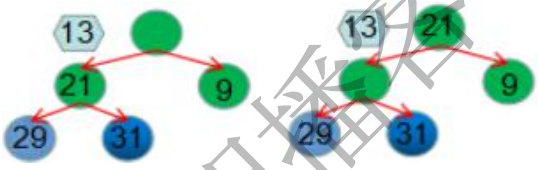
```
def sift(data,low,high)
    # 指定移除的堆顶位置元素下标为 i
    i = low
    # 将移除的堆顶元素存放到一个临时队列 tmp
    tmp = data[i]
```

关键点：

1、堆顶的标签最小

2、临时队列

1.2、选取新的堆顶节点



选取新的堆顶节点(下标i)，需要经过以下几个方面来确定：

1.2.1 选大的子节点

1.2.2 最大子节点跟移除的堆顶元素进行比较：

1.2.1 选取最大子节点

两个子节点比较，大子节点的下标为j，可以先假设左子节点下标数值j，

j的范围如何确定？。

如果说队列data的元素个数是n个，那么最后一个元素的下标high=n-1，所以，

j肯定在0 ~ high之间,即 j <= high

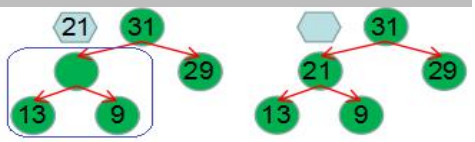
那么接下来无非就是 左结点data[j] 和 右结点data[j+1]比较了

根据我们在冒泡排序中说到的重点：过程跟踪,通过j标签永远找到大节点

最大结点的下标是j，所以左边结点小于右边结点的话，j =+ 1 就可以了

代码实现：

```
    # 假设左子节点是大节点，下标是 j
    j = 2 * i + 1
    # j 下标的操作范围
    while j <= high:
        # 两子节点进行比较
        if j + 1 <= high and data[j] < data[j+1]:
            # 过程跟踪，保证通过 j 找到最大元素
            j += 1
```



1.2.2 子节点和原堆顶节点比较

最大结点的元素是 data[j]，源堆顶元素data[i]

子节点元素 > 堆顶元素，

把子节点元素放到堆顶位置，对移动的后的空位置，重复1.2.1操作

子节点元素 < 堆顶元素，

子节点不动，移除的堆顶元素恢复原位，即终止操作break即可

代码实现：

```
        # 子节点比原堆顶节点元素大
        if data[j] > tmp:
            # 将子节点元素移动到堆顶位置
            data[i] = data[j]
            # 因为子节点位置空了，相当于堆顶节点移除了，又要重复 2.1 操作，所以需要更新 i 和 j 的值
            i = j
            j = 2 * i + 1
        # 如果最大的子节点小于移除的堆顶元素，终止该操作即可
        else:
```

```
            break
        data[i] = tmp
```

堆的调整最终代码：

```
# 堆的调整
def sift(data, low, high):
    i = low                                          # 堆顶节点的下标 i
    j = 2 * i + 1                                    # 上移节点标号 j，临时指向左侧子节点标号
    tmp = data[i]                                    # 把堆顶节点移动到临时队列，

    while j <= high:                                 # 左侧子节点小于堆的最大范围值
        if j + 1 <= high and data[j] < data[j+1]:    # 右侧节点元素如果大于左侧节点元素
            j += 1                                   # 上移节点标号 j 指向右侧节点
        if data[j] > tmp:                            # 如果上移节点标号的元素大于移除的堆顶元素
            data[i] = data[j]                        # 把上移节点元素移动到堆顶位置
            i = j                                    # 调整空位置节点标号 i 指向最新的空位置节点标号
            j = 2 * i +1                             # 上移节点标号 j，临时指向空位置节点的左侧子节点标号
        else:                                        # 如果子节点标号不在队列中，就退出操作
            break
    data[i] = tmp                                    # 设置堆顶节点为原来的内容
```

关键点：

1、父节点和子节点的关系：　`j = 2 * i + 1`

2、选择大节点：　`if j + 1 <= high and data[j] < data[j+1]:`

3、堆顶元素的确定：　`data[i] = data[j]` 或者 `data[i] = tmp`

2、堆顶元素排序

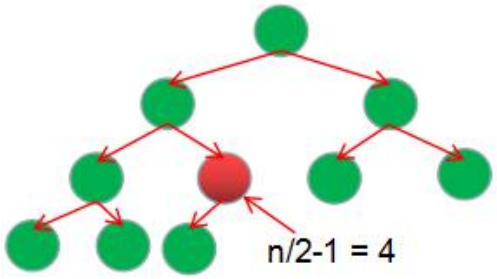
关于堆顶元素排序，我们从以下两个方面来分析：

堆的构造、堆顶元素输出到有序队列

2.1、堆的构造

我们要根据提供的无序队列，生成一个堆结构，

构造一个初始的堆结构，必须应该从最后一个包含子节点的父节点开始构造



这个父节点的地址是**[n/2 - 1]**

根据我们对堆的理解，从上到下，从左到右，他们的编号是依次递增的，所以[n/2 - 1]是最大的值，所以堆顶元素的位置变化如下：

[n/2 - 1] -> [n/2 - 2] -> ... -> 0,即 range(int(n/2)-1,-1,-1)

堆初始化的时候，传入的low和high是给下标用的，所以，在调用sift时候，high值应该为n-1

sift(data,i,n-1)

代码实现：

```
def heap_sort(data):
    # 获取当前列表的长度
    n = len(data)
    # 对所有父节点进行堆的调整，而且是降序排列
    for i in range(int(n/2) - 1, -1, -1):
        sift(data,i,n-1)
```

2.2 堆顶元素输出到有序队列

根据大顶堆排序的原理我们知道，每进行一次排序(元素替换+堆调整)

元素调整：即堆尾元素(下标i)和堆顶元素(下标0)替换

```
            data[0], data[i] = data[i], data[0]
```

堆调整：将无序队列调整为大顶堆

```
            sift(data, 0, i - 1)
```

堆顶元素就进入有序队列，无序队列元素就少1个。整个过程中，无序队列末尾元素的位置变化如下：

```
n-1 -> n-2 -> n-3 -> ...  -> 0 即 range(n-1,-1,-1)
```

最后返回最终的有序队列

```
return data
```

代码实现：

```
    # 指定最小元素的范围
    for i in range(n-1, -1, -1):
        # 队列中最大元素和最小元素进行替换
        data[0], data[i] = data[i], data[0]
        # 替换完毕后，重新调整堆结构，新的堆结构元素个数变成了 i-1 个
        sift(data, 0, i - 1)
    # 返回最终的有序队列
    return data
```

堆排序最终代码

```
# 构建堆
def heap_sort(data):
    n = len(data)                                   # 获取队列的长度
    # 构建初始堆结构
    for i in range(int(n/2) - 1, -1, -1):           # 调整的节点范围，从 n/2 - 1 开始，倒序到 0
        sift(data, i, n-1)                          # 调整当前结点堆序列
    #堆顶元素排序
    for i in range(n-1, -1, -1):                    # 遍历队列最小元素
        data[0], data[i] = data[i], data[0]         # 堆顶元素和堆最小元素进行替换
        sift(data, 0, i - 1)                        # 调整新队列
    # 返回排序后的队列
    return data
```

关键点：

1、无序队列构建初始堆

堆顶节点的范围：range(int(n/2) - 1, -1, -1)

2、排序过程中进行堆调整

堆顶和堆尾元素替换：data[0], data[i] = data[i], data[0]

剩余无序堆序列长度：range(n-1, -1, -1)

测试代码：

```
if __name__ == '__main__':
    a = [0, 2, 6, 98, 34, 5, 23, 11, 89, 100, 7]
    print("排序之前：%s" % a)

    c = heap_sort(a)
    print("排序之后：%s" % c)
```

时间复杂度

最优时间复杂度：O(nlogn)

对于每次堆顶元素退出，如果堆顶的元素放到队列的末尾，就和队列元素数量相关，所以时间复杂度就是O(n)

因为涉及到了递归，所以时间复杂度就是O(logn)

整体来说：最优的时间复杂度是 O(nlogn)

最坏时间复杂度：O(nlogn)

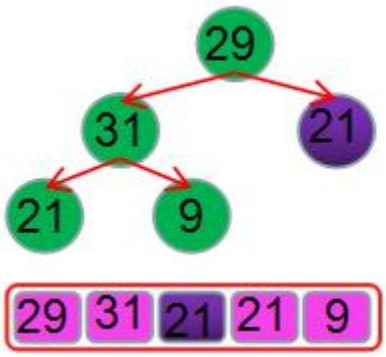
不管最好还是最坏都需要递归，所以循环的时间复杂度还是O(logn)

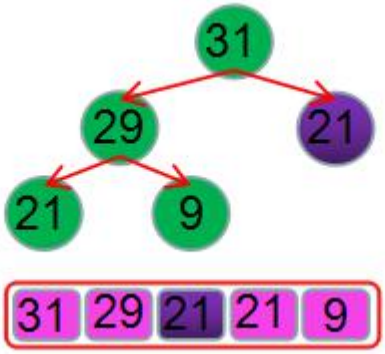
所以最坏也就最坏到每个具体的元素排序了，然而他们都和元素的数量有关，所以时间复杂度还是O(n)

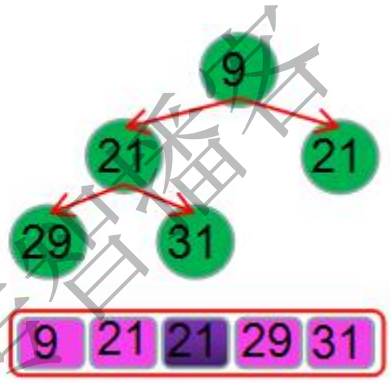
整体来说：最坏的时间复杂度是 O(nlogn)

稳定性：

不稳定，看图不解释







思考：小顶堆如果创建？

4.1.9 排序总结

排序总结中，我们从三个方面来学习：技术总结、成本总结、成本实践

技术总结：

以从小到大进行排序为例：

冒泡排序：在无序队列中选择**最小的移动到最左侧**，

选择排序：定一个有序队列，从无序队列中**选**择**最小**的元素**追加**到有序队列的末尾

插入排序：定一个有序队列，从无序队列中选择**第一**个元素，**插入**到到有序队列的**合适位置**

希尔排序：通过对无序队列进行**分组**，然后再采用**插入**的排序方法

快速排序：指定一个元素，将无序队列**拆分**为**大小**两部分，然后**层级递进**，最终实现有序队列

归并排序：是将无序队列**拆分**，然后小**组内排序**，组间元素比较后在**新队列**中进行排序

堆 排 序：顺序表方式构造堆，首尾替换调整堆

总结小诗：

冒小左移**选**追加，**插入**合适分**希尔**，**快速**两半**归**新列，顺表构造首尾**堆**

成本总结：

排序方法	时间复杂度			稳定性	代码复杂度
	最坏情况	平均情况	最好情况		
冒泡排序	O(n²)	O(n²)	O(n)	稳定	简单
选择排序	O(n²)	O(n²)	O(n²)	不稳定	简单
插入排序	O(n²)	O(n2)	O(n²)	稳定	简单
希尔排序	O(n²)	O(nlogn~n²)	O(nlogn~n²)	不稳定	中下等
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	不稳定	中等
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	稳定	中等
快速排序	O(n²)	O(nlogn)	O(nlogn)	不稳定	中下等

注意：

堆排序，我们在"树&二叉树"部分中讲解

成本实践：

结合我们之前对python性能分析模块timeit的学习，接下来，我们就在这个地方对所有的排序进行一次性能实践分析。

场景：

对一千个随机数字进行大小排序，然后分析孰优孰劣。

分析：

完成场景我们需要考虑以下几步：

1、生成一千个数字

2、将数字的顺序打乱

3、性能分析基准值实践

4、排序算法性能实践

5、排序总结

技术分析：

1、生成一千个数字，我们使用range方法搞定,然后使用list方法形成一个队列

代码：data = list(range(1000))

2、将数字的顺序打乱，我们结合random模块的shuffle方法实现队列的打散

代码：random.shuffle(data)

3、算法代码的性能分析实践

为了对同一个无序序列进行不同方法进行排序，而每次一次random.shuffle(li)都生成不一样的序列，所以我们采用“深拷贝”的方法对第一个random.shuffle(li)进行复制，然后对复制品进行排序分析。

代码：data1 = copy.deepcopy(data)

我们知道系统有一个默认的排序方法sort()，因为是底层c写的，他的排序速度是最快的，所以我们就以它的时间为基准值。然后按照同样的方法对其他算法进行测试。

测试代码：

```
if __name__ == '__main__':
    data = list(range(1000))
    random.shuffle(data)
    data1 = copy.deepcopy(data)

    # 测试系统默认的排序，因为是底层的 c 写的，所以非常快
    def test_sys():
        data1.sort()
        print(data1)

    # 性能数据输出
    sort1 = timeit.Timer("test_sys()", "from __main__ import test_sys")
    print("test_sys: %f seconds" % sort1.timeit(1))
```

4、排序算法性能实践

代码实践：

```
if __name__ == '__main__':
    # 生成散队列
    data = list(range(1000))
    random.shuffle(data)
    # 性能分析对象
    data1 = copy.deepcopy(data)
    data2 = copy.deepcopy(data)
    data3 = copy.deepcopy(data)
    data4 = copy.deepcopy(data)
```

```
data5 = copy.deepcopy(data)
data6 = copy.deepcopy(data)
data7 = copy.deepcopy(data)
data8 = copy.deepcopy(data)

# 测试基准值
def test_sys():
    data1.sort()
    print(data1)

# 测试冒泡排序
def test_bubble():
    bubble_sort(data2)
    print(data2)

# 测试选择排序
def test_xuanze():
    selection_sort(data3)
    print(data3)

# 测试插入排序
def test_charu():
    insert_sort(data4)
    print(data4)

# 测试希尔排序
def test_xier():
    shell_sort(data5)
    print(data5)

# 测试快速排序
def test_kuaisu():
    bubble_quick(data6,0,len(data6))
    print(data6)

# 测试归并排序
def test_guibing():
    fen_zu(data7)
    print(data7)

# 测试堆排序
def test_dui():
    fen_zu(data8)
    print(data8)

# 信息输出
sort1 = timeit.Timer("test_sys()", "from __main__ import test_sys")
print("系统排序: %f seconds" % sort1.timeit(1))
```

```
mp1 = timeit.Timer("test_bubble()", "from __main__ import test_bubble")
print("冒泡排序: %f seconds" % mp1.timeit(1))

xz1 = timeit.Timer("test_xuanze()", "from __main__ import test_xuanze")
print("选择排序: %f seconds" % xz1.timeit(1))

cr1 = timeit.Timer("test_charu()", "from __main__ import test_charu")
print("插入排序: %f seconds" % cr1.timeit(1))

xr1 = timeit.Timer("test_xier()", "from __main__ import test_xier")
print("希尔排序: %f seconds" % xr1.timeit(1))

ks1 = timeit.Timer("test_kuaisu()", "from __main__ import test_kuaisu")
print("快速排序: %f seconds" % ks1.timeit(1))

gb1 = timeit.Timer("test_guibing()", "from __main__ import test_guibing")
print("归并排序: %f seconds" % gb1.timeit(1))

dp1 = timeit.Timer("test_dui()", "from __main__ import test_dui")
print("堆 排 序: %f seconds" % dp1.timeit(1))
```

最终效果：

```
系统排序: 0.000325 seconds
冒泡排序: 0.194804 seconds
选择排序: 0.075775 seconds
插入排序: 0.138085 seconds
希尔排序: 0.008100 seconds
快速排序: 0.004462 seconds
归并排序: 0.006758 seconds
堆 排 序: 0.007604 seconds
```

经过分析，他们的执行时间如下：

系统排序 < 快速排序 < 归并排序 < 堆 排 序 < 希尔排序 < 选择排序 < 插入排序 < 冒泡排序

这也对应了我们在讲排序的时候，给他们的一个分类。

4.2 搜索

搜索这部分知识，我们从三个方面来介绍：

搜索简介、递归二分查找实践、顺序二分查找实践

4.2.1 搜索简介

简介

搜索是在队列中找到一个特定元素的算法过程。搜索的结果，只有两个：True(找到)或False(找不到)。

搜索的几种常见方法：

顺序查找、二分法查找、二叉树查找、哈希查找

顺序查找简介：

就是从头到尾或者从尾到头的遍历查找。

从我们开始学习Python的列表功能开始，就学习了顺序查找，在数据结构进阶部分，我们在顺序表、链表、栈及队列的学习中，对顺序查找的理解有更深了一步，所以这个方法我们就不再次专门学习了。

我们重点来学习二分查找。

4.2.2 二分查找

简介：

二分查找又称折半查找，

优点：

比较次数少，查找速度快，平均性能好；

缺点：

要求待查表为有序表，且插入删除困难。

原理：

场景：有序队列alist，升序排序。 [1,2,3,...,n]，要在alist中找到一个叫j的元素

1、找中间，比较

设置alist中间位置为mid，然后让mid元素和j元素进行比较，

如果两者相等，则查找成功；否则进行"二分比较"

2、二分比较

利用中间位置mid将alist分成两个小组A队列和B队列，

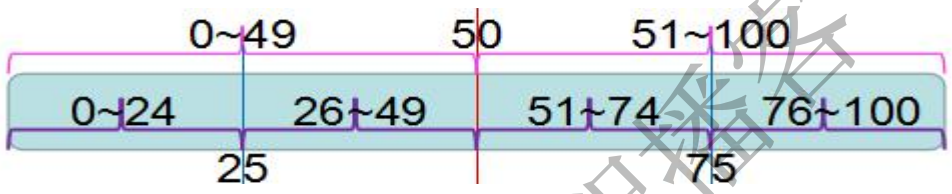
A队列元素都比mid值小，B队列元素比mid值大

然后将mid值和我们要查找的元素j进行比较，

j如果比mid小，那么去A队列找，反之去B队列找。

3、重复1-2过程，如果最终能找到，表示查找成功，否则的话，查找不成功

示意图



中间值：mid = 50

前半部分A：[:mid] 后半部分B:[mid+1:]

适用范围：

适用于不经常变动而查找频繁的有序列表。

二分查找有两种情况：

常见的递归二分查找、普通的遍历二分查找

4.2.3 递归二分实践

递归二分查找特点

函数本身就是对一个队列进行二分比较的功能，而对子队列进行二分比较，无非就是调用函数自身的一个效果。

代码分析

经过我们对递归二分查找的原理解析，我们知道要实现一个二分查找的话，必须从三方面考虑：

列表是否为空，列表中间元素匹配，递归二分查找

1、列表是否为空

如果列表为空，那么肯定找不到

代码实现：

```
def binary_search(alist, item):
    # 获取列表的长度
    n = len(alist)
    # 判断传入列表长度是否为 0
    if n == 0:
```

```
        return False
```

2、列表中间元素匹配

中间元素匹配，主要有两种情况：

匹配成功和匹配失败

匹配成功的话，直接返回True即可，

代码实现：

```
        # 列表从中间切开
        mid = n // 2
        # 判断中间值是否是我们想要的值，是的话返回 True
        if alist[mid] == item:
          return True
```

匹配不成功的话，判断元素和mid值的大小，分别去两个小组中查找

左侧小组：alist[:mid] 右侧小组：alist[mid+1:]

思考：为什么mid要+1

因为mid我们已经比较过了

代码实现：

```
        # 查找的元素小于队列中间值
        elif item < alist[mid]:
            去左侧 alist[:mid]查找元素
        # 查找的元素大于队列中间值
        else:
            去右侧 alist[mid:]查找元素
```

3、递归二分查找

递归二分查找，就是在我已分小组的基础上，再次进行二分查找，所以直接调用函数本身即可实现效果

左侧二分递归查找：binary_search(alist[:mid],item)

右侧二分递归查找：return binary_search(alist[mid+1:],item)

```
    def binary_search(alist, item):
        ...
        elif item < alist[mid]:
            return binary_search(alist[:mid],item)
        else:
            return binary_search(alist[mid+1:],item)
```

代码实践

实践代码：

```
    def binary_search(alist, item):
        # 准备工作
        n = len(alist)
        # 列表空判断
        if n == 0:
            return False
        mid = n // 2
        # 中间值匹配
        if alist[mid] == item:
            return True
        # 左侧二分查找
        elif item < alist[mid]:
```

```
            return binary_search(alist[:mid], item)
        # 右侧二分查找
        else:
            return binary_search(alist[mid + 1:], item)
```

测试代码：

```
    if __name__ == "__main__":
        testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42, ]
        print(binary_search(testlist, 3))
        print(binary_search(testlist, 13))
```

注意：测试的时候列表的格式一定要正确，如果返回查找不到的话，那就是列表格式的问题

4.2.4 普通二分实践(了解)

普通二分查找特点：

普通方式的二分查找获取中间位置的元素方法改变了：

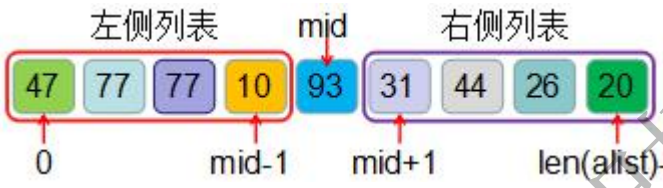
当我开始进行二分匹配的时候，直接将队列的首位分别用两个标签表示

简单来说：普通二分查找就是

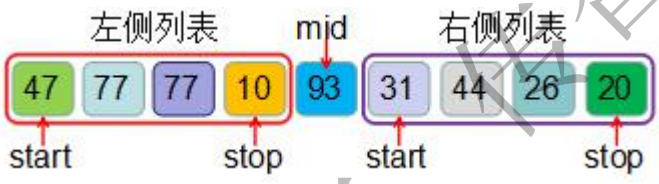
一次二分组，遍历子组找元素。

普通二分查找原理解析：

1、第一次二分后的效果：



2、这个时候，使用start和stop标签表示队列的两端



左侧队列：start = 0 ， stop = mid-1

右侧队列：start = mid+1 ， stop = n-1

3、如果当前的mid值和查找元素不匹配，找到相应的队列(即移动相应的标签即可)

如果去左侧队列查找元素，移动stop标签：stop = mid - 1

如果去右侧队列查找元素，移动start标签：start = mid + 1

4、通过(start+stop)/2的方式在子队列找到中间位置mid，再次进行二分匹配

5、然后一直循环下去，

实践分析：

通过上面的分析，我们知道，普通二分查找需要通过以下三个个方面来完成：

基本功能、子队列二分查找、二分遍历

1、基本功能

基本功能这块，主要包括以下几个方面：

队列的边界确定：

```
    start = 0
    stop = len(alist)-1
```

队列空值判断

```
    len(alist) == 0
```

队列不空，确定中间值mid，

mid = (start + stop) // 2

mid值和要查找的元素进行匹配

alist[mid] == item

代码实现：

```
# 获取列表的长度
n = len(alist)
# 队列空，直接返回 False
if n == 0:
    return False
# 队列不空，进行中间值判断
mid = (start + stop) // 2
if alist[mid] == item:
    return True
else:
```

2、子队列二分查找

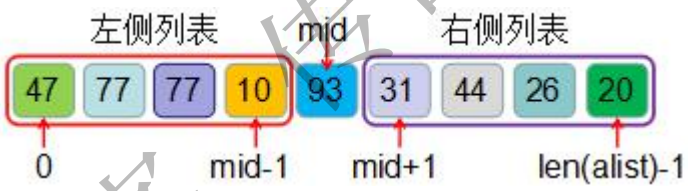
关于子列表的二分查找，无非就是进入左右侧哪个队列罢了：

进入左侧队列

start标签不用动，移动stop = mid - 1

进入右侧队列

stop标签不用动，移动start = mid + 1



代码实现：

```
# 如果我们匹配的内容在左侧列表
elif alist[mid] < item:
    start = mid + 1
# 如果我们匹配的内容在左侧列表
else:
    stop = mid - 1
```

3、递归遍历

需要对子列表进行再次的二分查找，那么必须有个前提：

列表至少有元素，即队列的两个标签必须满足 start <= stop，

如果start > stop的话，就表示空队列，没有找到元素，直接返回False即可

代码实现：

```
# 对循环条件进行判断
while start <= stop:
    # 执行匹配语句
# 循环条件不成立，直接返回错误
return False
```

代码实践：

实践代码：

```
def binary_search(alist, item):
    n = len(alist)
    # 列表空判断
    if n == 0:
        return False
    # 当前队列范围
    start = 0
    stop = len(alist) - 1
```

```
    # 递归遍历
    while start <= stop:
        # 获取中间值，然后进行匹配
        mid = (start + stop) // 2
        if alist[mid] == item:
            return True
        # 不匹配的话，移动队列的边界
        elif alist[mid] < item:
            start = mid + 1
        else:
            stop = mid - 1
    # 循环结束匹配失败，返回 False
    return  False
```

测试代码：

```
if __name__ == "__main__":
    testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42, ]
    print(binary_search(testlist, 3))
    print(binary_search(testlist, 13))
```

4.3 二叉树

这部分我们从

二叉树遍历、二叉树查询、二叉树反推

4.3.1 二叉树遍历

关于二叉树的遍历我们从以下五个方面来介绍：

简介、方法、广度优先、深度优先、三种深度

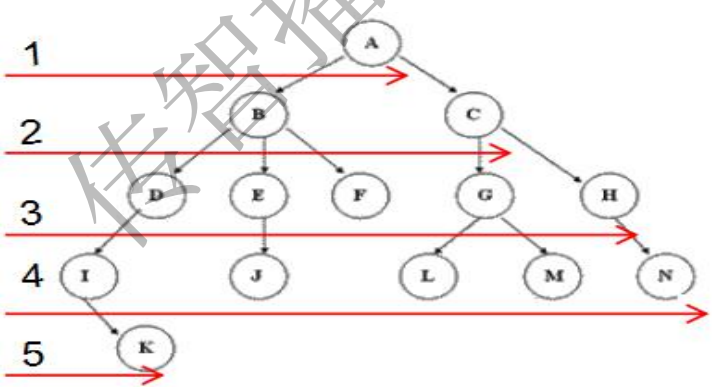
遍历简介

遍历是指对树中所有结点的信息的访问一次且仅访问一次。

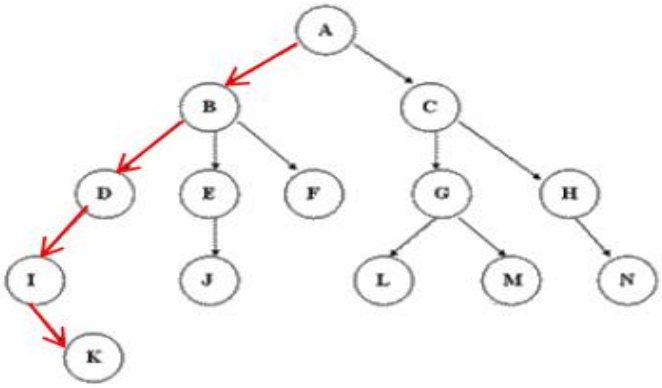
遍历方法

树有两种遍历模式是：**广度优先遍历**和**深度优先遍历，**

广度优先一般用队列，深度优先一般用递归。



广度优先



深度优先

广度优先遍历

特点：

查看数据：

从上到下，分层查看，每层从左向右依次查看，直至所有数据查看完毕

添加数据：

从上到下，分层添加，每层从左向右依次添加。

示例：

A B C D E F G H I J L M N K

深度优先遍历

特点：

首先递归方法看最深的分支元素，再看其他的节点元素。

三种深度遍历方法：

先序遍历（preorder），中序遍历（inorder）和后序遍历（postorder）。

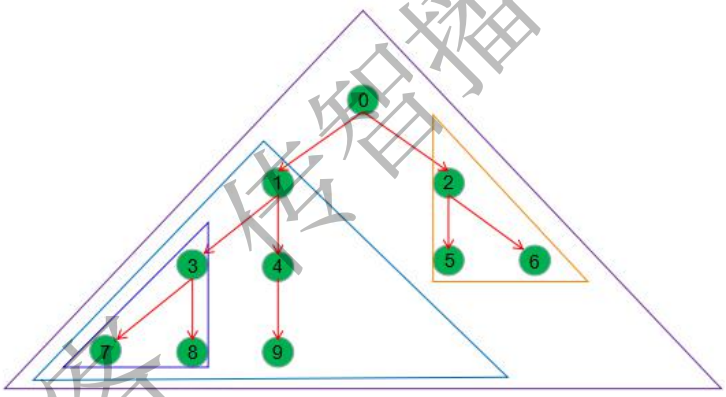
三种深度查看方法：

先序遍历：

访问顺序：根节点->左子树->右子树

示例：

0 1 3 7 8 4 9 2 5 6



中序遍历：

访问顺序：左子树->根节点->右子树

示例：

7 3 8 1 9 4 0 5 2 6

后序遍历：

访问顺序：左子树->右子树->根节点

示例：

7 8 3 9 4 1 5 6 2 0

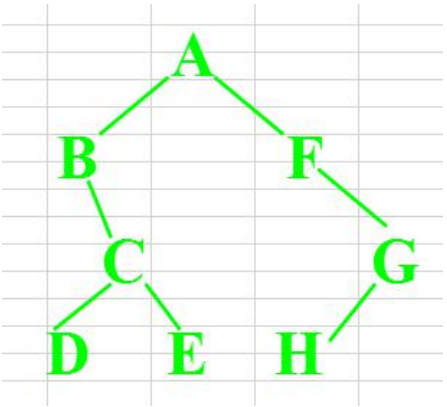
拓展：写出如下图的四种遍历方法：

层次：

先序：

中序：

后序：



4.3.2 查询实践(广度优先)

分析

在3.8.3一节二叉树实践，就是按照广度优先的方式来添加数据的，所以，我们使用广度优先的方式来查看的话，跟我们刚才写的代码就很类似。

查看二叉树分为两种情况：

二叉树没有数据

二叉树有数据

二叉树没有数据的情况：

二叉树有没有数据，判断根节点有没有数据即可，没有的话，直接退出就可以了

代码实现：

```
    def guangdu_travel(self):
        # 判断有没有根节点，没有的话直接退出即可
```

```
        if self.root == None:
            return
```

二叉树有数据的情况：

如果有数据，那么我要将所有数据输出，结合我们添加数据的方式，我们先创建一个临时队列，先把根的元素放进去，以便查看子节点元素

代码实现：

```
    def guangdu_travel(self):
        ...
        # 创建一个临时元素存放队列
        queue = []
        # 把根的元素放进去
        queue.append(self.root)
```

当待处理的父节点队列中有数据，说明就有父节点的信息没有查看。

代码实现：

```
    # 判断临时队列是否为空，如果非空，查看相应父节点的子节点
    while len(queue) > 0:
        ...
```

从待处理父节点队列中，按顺序处理每一个父节点：

根据结点的item属性，输出父节点内容

判断左右侧子节点是否为空，

左子节点不空，将其放入待处理父节点队列；空，判断右子节点，

右子节点不空，将其放入待处理父节点队列；空，同样方式处理临时队列中的下一个父节点

代码实现：

```
        # 从临时队列中获取父节点元素，并打印父节点信息
        node = queue.pop(0)
        print(node.item,end=" ")
        # 如果该节点存在左侧子节点元素，把该子节点数据放到临时队列中
        if node.lsub:
            queue.append(node.lsub)
        # 如果该节点存在右侧子节点元素，把该子节点数据放到临时队列中
        if node.rsub:
            queue.append(node.rsub)
```

广度优先查看代码：

```
class Tree(object):
    """树类"""
    def __init__(self, root=None):
        self.root = root
    ...

    def guangdu_travel(self):
        # 判断有没有根节点，没有的话直接退出即可
        if self.root == None:
            return

        # 创建一个临时元素存放队列，将根节点放入进去
        queue = []
```

```
        queue.append(self.root)

        # 判断临时队列不为空，表示还有未查看的父节点
        while len(queue) > 0:
            # 获取父节点并打印期信息
            node = queue.pop(0)
            print(node.item,end=" ")
            # 左侧节点不为空，将其放入待处理队列
            if node.lsub:
                queue.append(node.lsub)
            # 右侧节点不为空，将其放入待处理队列
            if node.rsub:
                queue.append(node.rsub)
        # 修复 print 功能
        print("")
```

测试代码：

```
if __name__ == '__main__':
    tree = Tree()
    tree.add(0)
    ...
    tree.add(9)
    tree.guangdu_travel()
```

关键点：

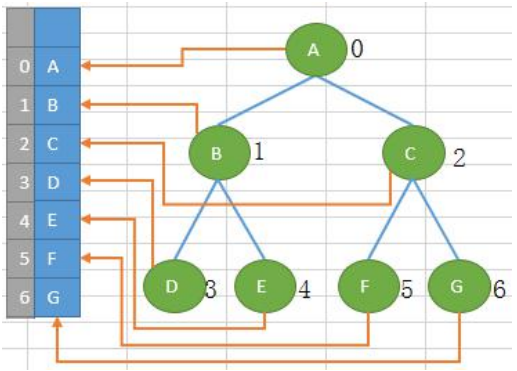
1、空树处理：　　if self.root == None:

2、树非空，待处理父节点队列　　queue.append(self.root)

3、左右侧子节点处理：

不空，将其放入待处理父节点队列：queue.append(node.lsub)

4.3.3 查询实践(深度优先)



根据我们对二叉树深度遍历的了解，这三种方法特点在于：

同：递归查询

异：查询的顺序不一样

先序遍历

遍历的基本功能就是打印出当前树的节点内容，先序遍历打印特点是：

根节点(父节点) --> 左侧子节点 --> 右侧子节点

所以先序遍历打印的第一个内容肯定是根节点元素

代码实现：

```
    # 打印根/父节点内容
    print(root.item, end=" ")
```

我们先序遍历的对象是某个具体的树，所以我们要考虑两个方面：

这个树对象是我们传递过来的一个对象

这个树对象不能为空

代码实现：

```
    def xianxu_travel(self,root):
        # 先对传入节点的内容是否为空进行判断，
```

```
        if root:
            # 打印根节点内容
            print(root.item, end=" ")
```

接着就是打印传入节点的左右侧两个子节点的内容了，我们知道了：

打印父节点A内容的功能我们已经实现了，

父节点的子节点是否还有下一层的子节点我们不确定

所以我们可以使用函数自调用的方法来实现递归打印所有节点的内容，注意前后打印的顺序

代码实现：

```
    def xianxu_travel(self,root):
        ...
            # 打印左侧节点内容
            self.xianxu_travel(root.lsub)
            # 打印右侧节点内容
            self.xianxu_travel(root.rsub)
```

最终实践代码：

实践代码：

```
    class Tree(object):
    ...
        def xianxu_travel(self,root):
            # 先对传入节点的内容是否为空进行判断，
            if root:
                # 打印根节点内容
                print(root.item, end=" ")
                # 打印左侧节点内容
                self.xianxu_travel(root.lsub)
                # 打印右侧节点内容
                self.xianxu_travel(root.rsub)
```

测试代码：

```
    if __name__ == '__main__':
        tree = Tree()
        for i in range(0,10):
            tree.add(i)
        tree.xianxu_travel(tree.root)
```

中序遍历

关于中序遍历，我们知道他的打印特点是：

左侧子节点 --> 根节点(父节点) --> 右侧子节点

其他的根先序遍历完全一致，所以实现中序遍历只需要更改一下打印顺序即可

实践代码：

```
    class Tree(object):
        """树类"""
        def __init__(self, root=None):
            self.root = root
    ...
        def zhongxu_travel(self,root):
            # 先对传入节点的内容是否为空进行判断，
```

```
        if root:
            # 打印左侧节点内容
            self.zhongxu_travel(root.lsub)
            # 打印根节点内容
            print(root.item, end=" ")
            # 打印右侧节点内容
            self.zhongxu_travel(root.rsub)
```

测试代码：

```
if __name__ == '__main__':
    tree = Tree()
    for i in range(0,10):
        tree.add(i)
    tree.zhongxu_travel(tree.root)
```

后序遍历

关于后序遍历，我们知道他的打印特点是：

根节点(父节点) --> 左侧子节点 --> 右侧子节点

其他的根先序遍历完全一致，所以实现后序遍历只需要更改一下打印顺序即可

实践代码：

```
class Tree(object):
    """树类"""
    def __init__(self, root=None):
        self.root = root
...
    def houxu_travel(self,root):
        # 先对传入节点的内容是否为空进行判断,
        if root:
            # 打印左侧节点内容
            self.houxu_travel(root.lsub)
            # 打印右侧节点内容
            self.houxu_travel(root.rsub)
            # 打印根节点内容
            print(root.item, end=" ")
```

测试代码：

```
if __name__ == '__main__':
    tree = Tree()
    for i in range(0,10):
        tree.add(i)
    tree.houxu_travel(tree.root)
```

4.3.4 二叉树反推(拓展)

这是一个拓展的内容：根据提供的三种深度排序中的两种排序，反推出来二叉树的图

比如：

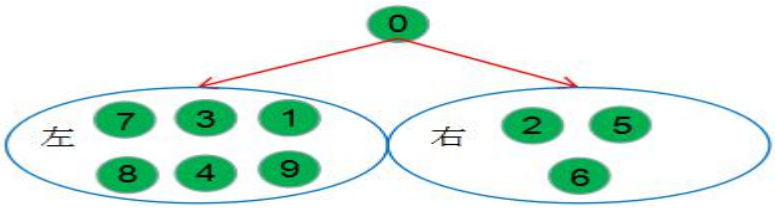
先序：0 1 3 7 8 4 9 2 5 6

中序：7 3 8 1 9 4 0 5 2 6

1、找根

先序的特点是第一个元素是根，中序的特点是根两侧分别是左右子树

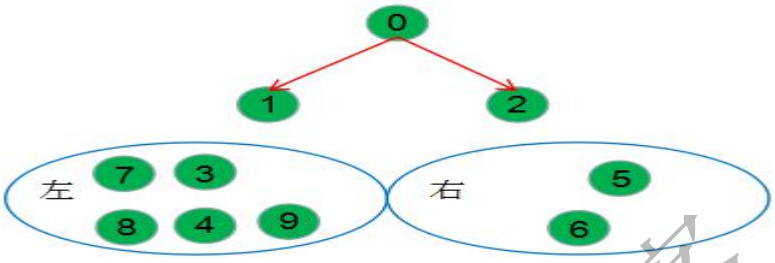
所以我们反推分界初始图：



2、找第二层子节点

根据中序的内容，我们确定了两个子树包含的内容，那么结合先序的特点，两个范围内首先出现的数字就是第一层的节点内容

所以左侧子树的根节点是1，右侧子树的根节点是2



3、找第三层数据

找到左侧子树的根节点是1，

那么结合中序的左侧子树内容：7 3 8 1 9 4，可以确定：左侧子树包括

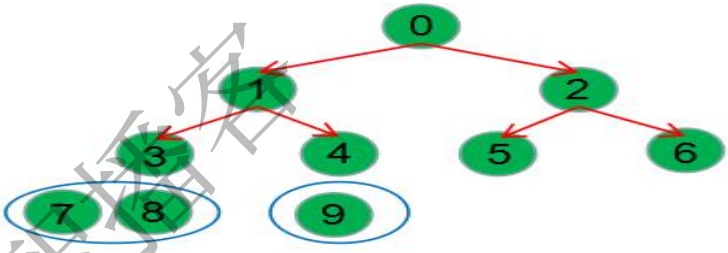
左部分：738　　右部分：94

结合先序的左侧子树内容：1 3 7 8 4 9，可以确定：左侧子树的1元素的两个子节点是3和9

找到右侧子树的根节点是2

结合中序的右侧子树内容：5 2 6　　结合先序的右侧子树内容：2 5 6

可以确定：2节点的左侧元素是5，右侧元素是6



4、找第四层数据

对于3结点来说：

结合中序的内容：7 3 8　　结合先序的内容：3 7 8

可以确定：3节点的左侧元素是7，右侧元素是8

对于9结点来说：

结合中序的内容：9 4　　结合先序的内容：4 9

可以确定：9节点的左侧元素是4

所以最终的二叉树图是：