数据结构和算法

ju7ran

目　录

1-为什么要学习数据结构和算法

你是不是觉得数据结构和算法，跟操作系统、计算机网络一样，是脱离实际工作的知识？可能除了面试，这辈子也用不着？

尽管计算机相关专业的同学在大学都学过这门课程，但是据我了解，很多程序员对数据结构和算法依旧一窍不通。还有一些人也只听说过数组、链表、快排这些最最基本的数据结构和算法，稍微复杂一点的就完全没概念。

当然，也有很多同学说，自己实际工作中根本用不到数据结构和算法。所以，就算不懂这块知识，只要代码、开发框架用得熟练，照样可以把代码写得'飞'起来。事实真的是这样吗？

今天我们就来详细聊一聊,为什么要学习数据结构和算法。

想要通关大厂面试，千万别让数据结构和算法拖了后腿
很多大公司，比如BAT、Google、Facebook，面试的时候都喜欢考算法、让人现场写代码。有些人虽然技术不错，但每次去面试都会'跪'在算法上，很是可惜。那你有没有想过，为什么这些大公司都喜欢考算法呢？

校招的时候，参加面试的学生通常没有实际项目经验，公司只能考察他们的基础知识是否牢固。社招就更不用说了，越是厉害的公司，越是注重考察数据结构与算法这类基础知识。相比短期能力， 他们更看中你的长期潜力。

你可能要说了，我不懂数据结构与算法，照样找到了好工作啊。那我是不是就不用学数据结构和算法呢？当然不是，你别忘了，我们学任何知识都是为了“用”的，是为了解决实际工作问题的，学习数据结构和算法自然也不例外。

业务开发工程师，你真的愿意做一辈子 CRUD boy 吗？
如果你是一名业务开发工程师，你可能要说，我整天就是做数据库 CRUD（增删改查），哪里用得到数据结构和算法啊？

是的，对于大部分业务开发来说，我们平时可能更多的是利用已经封装好的现成的接口、类库来堆砌、翻译业务逻辑，很少需要自己实现数据结构和算法。但是，不需要自己实现，并不代表什么都不需要了解。

如果不知道这些类库背后的原理，不懂得时间、空间复杂度分析，你如何能用好、用对它们？存储某个业务数据的时候，你如何知道应该用 ArrayList，还是 Linked List 呢？调用了某个函数之后，你又该如何评估代码的性能和资源的消耗呢？

作为业务开发，我们会用到各种框架、中间件和底层系统，比如 Spring、RPC 框架、消息中间件、Redis 等等。在这些基础框架中，一般都揉和了很多基础数据结构和算法的设计思想。

比如，我们常用的 Key-Value 数据库 Redis 中，里面的有序集合是用什么数据结构来实现的呢？为什么要用跳表来实现呢？为什么不用二叉树呢？

如果你能弄明白这些底层原理，你就能更好地使用它们。即便出现问题，也很容易就能定位。因此，掌握数据结构和算法，不管对于阅读框架源码，还是理解其背后的设计思想，都是非常有用的。

在平时的工作中，数据结构和算法的应用到处可见。我来举一个你非常熟悉的例子：如何实时地统计业务接口的 99% 响应时间？
你可能最先想到，每次查询时，从小到大排序所有的响应时间，如果总共有 1200 个数据，那第1188 个数据就是 99% 的响应时间。很显然，每次用这个方法查询的话都要排序，效率是非常低的。但是，如果你知道“堆”这个数据结构，用两个堆可以非常高效地解决这个问题。

基础架构研发工程师，写出达到开源水平的框架才是你的目标！
现在互联网上的技术文章、架构分享、开源项目满天飞，照猫画虎做一套基础框架并不难。我就拿RPC框架举例。

不同的公司、不同的人做出的 RPC 框架，架构设计思路都差不多，最后实现的功能也都差不多。但是有的人做出来的框架，Bug 很多、性能一般、扩展性也不好，只能在自己公司仅有的几个项目里面用一下。而有的人做的框架可以开源到 GitHub 上给很多人用，甚至被 Apache 收录。为什么会有这么大的差距呢？

我觉得，高手之间的竞争其实就在细节。这些细节包括：你用的算法是不是够优化，数据存取的效率是不是够高，内存是不是够节省等等。这些累积起来，决定了一个框架是不是优秀。所以，如果你还不懂数据结构和算法，没听说过大 O 复杂度分析，不知道怎么分析代码的时间复杂度和空间复杂度，那肯定说不过去了，赶紧来补一补吧！

对编程还有追求？不想被行业淘汰？那就不要只会写凑合能用的代码！
何为编程能力强？是代码的可读性好、健壮？还是扩展性好？我觉得没法列，也列不全。但是，在我看来，性能好坏起码是其中一个非常重要的评判标准。但是，如果你连代码的时间复杂度、空间复杂度都不知道怎么分析，怎么写出高性能的代码呢？

你可能会说，我在小公司工作，用户量很少，需要处理的数据量也很少，开发中不需要考虑那么多性能的问题，完成功能就可以，用什么数据结构和算法，差别根本不大。但是你真的想“十年如一 日”地做一样的工作吗？经常有人说，程序员 35 岁之后很容易陷入瓶颈，被行业淘汰，我觉得原因其实就在此。有的人写代码的时候，从来都不考虑非功能性的需求，只是完成功能，凑合能用就好；做事情的时候，也从来没有长远规划，只把眼前事情做好就满足了。

我曾经面试过很多大龄候选人，简历能写十几页，经历的项目有几十个，但是细看下来，每个项目都是重复地堆砌业务逻辑而已，完全没有难度递进，看不出有能力提升。久而久之，十年的积累可能跟一年的积累没有任何区别。这样的人，怎么不会被行业淘汰呢？

如果你在一家成熟的公司，或者 BAT 这样的大公司，面对的是千万级甚至亿级的用户，开发的是TB、PB 级别数据的处理系统。性能几乎是开发过程中时刻都要考虑的问题。一个简单的ArrayList、Linked List 的选择问题，就可能会产生成千上万倍的性能差别。这个时候，数据结构和算法的意义就完全凸显出来了。

其实，我觉得，数据结构和算法这个东西，如果你不去学，可能真的这辈子都用不到，也感受不到它的好。但是一旦掌握，你就会常常被它的强大威力所折服。之前你可能需要费很大劲儿来优化的代码，需要花很多心思来设计的架构，用了数据结构和算法之后，很容易就可以解决了。

2-如何抓住重点，系统高效地学习数据结构与算法？

你是否曾跟我一样，因为看不懂数据结构和算法，而一度怀疑是自己太笨？实际上，很多人在第一次接触这门课时，都会有这种感觉，觉得数据结构和算法很抽象，晦涩难懂，宛如天书。正是这个原因，让很多初学者对这门课望而却步。

我个人觉得，其实真正的原因是你没有找到好的学习方法，没有抓住学习的重点。实际上，数据结构和算法的东西并不多，常用的、基础的知识点更是屈指可数。只要掌握了正确的学习方法，学起来并没有看上去那么难，更不需要什么高智商、厚底子。

什么是数据结构？什么是算法？

大部分数据结构和算法教材，在开篇都会给这两个概念下一个明确的定义。但是，这些定义都很抽象，对理解这两个概念并没有实质性的帮助，反倒会让你陷入死抠定义的误区。毕竟，我们现在学习，并不是为了考试，所以，概念背得再牢，不会用也就没什么用。

虽然我们说没必要深挖严格的定义，但是这并不等于不需要理解概念。 下面我就从广义和狭义两个层面，来帮你理解数据结构与算法这两个概念。

从广义上讲，数据结构就是指一组数据的存储结构。算法就是操作数据的一组方法。

图书馆储藏书籍你肯定见过吧？为了方便查找，图书管理员一般会将书籍分门别类进行“存储”。按照一定规律编号，就是书籍这种“数据”的存储结构。

那我们如何来查找一本书呢？有很多种办法，你当然可以一本一本地找，也可以先根据书籍类别的编号，是人文，还是科学、计算机，来定位书架，然后再依次查找。笼统地说，这些查找方法都是算法。

那数据结构和算法有什么关系呢？为什么大部分书都把这两个东西放到一块儿来讲呢？

这是因为，数据结构和算法是相辅相成的。数据结构是为算法服务的，算法要作用在特定的数据结构之上。 因此，我们无法孤立数据结构来讲算法，也无法孤立算法来讲数据结构。

比如，因为数组具有随机访问的特点，常用的二分查找算法需要用数组来存储数据。但如果我们选择链表这种数据结构，二分查找算法就无法工作了，因为链表并不支持随机访问。

数据结构是静态的，它只是组织数据的一种方式。如果不在它的基础上操作、构建算法，孤立存在的数据结构就是没用的。

下面我们来看看，究竟该怎么学数据结构与算法。

学习这个需要什么基础？

看到数据结构和算法里的“算法”两个字，很多人就会联想到“数学”，觉得算法会涉及到很多深奥的数学知识。那我数学基础不是很好，学起来会不会很吃力啊？

数据结构和算法课程确实会涉及一些数学方面的推理、证明，尤其是在分析某个算法的时间、空间复杂度的时候，但是这个你完全不需要担心。

我会由浅入深，从概念到应用，一点一点给你解释清楚。你只要有高中数学水平，就完全可以学习。

3-算法引入

引入

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

代码实现

```
import time

start_time = time.time()

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

end_time = time.time()
print("times: %f" % (end_time - start_time))
print("end!")
```

算法的概念

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

算法是独立存在的一种解决问题的方法和思想。

对于算法而言，实现的语言并不重要，重要的是思想。

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

算法的五大特性

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

第二次尝试

```
import time

start_time = time.time()

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

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

算法效率衡量

执行时间反应算法效率

对于同一问题，我们给出了两种解决算法，在两种算法的实现中，我们对程序执行的时间进行了测算，发现两段程序执行的时间相差悬殊（214.583347秒相比于0.182897秒），由此我们可以得出结论：实现算法程序的执行时间可以反应出算法的效率，即算法的优劣。

单靠时间值绝对可信吗？

1.测试结果非常依赖测试环境

测试环境中硬件的不同会对测试结果有很大的影响。比如，我们拿同样一段代码，分别用 Intel Core i9 处理器和 Intel Core i3 处理器来运行，不用说，i9 处理器要比 i3 处理器执行的速度快很多。还有，比如原本在这台机器上 a 代码执行的速度比 b 代码要快，等我们换到另一台机器上时，可能会有截然相反的结果。

2.测试结果受数据规模的影响很大

对同一个排序算法，待排序数据的有序度不一样，排序的执行时间就会有很大的差别。极端情况下，如果数据已经是有序的，那排序算法不需要做任何操作，执行时间就会非常短。除此之外，如果测试数据规模太小，测试结果可能无法真实地反应算法的性能。

所以，我们需要一个不用具体的测试数据来测试，就可以粗略地估计算法的执行效率的方法

大 O 复杂度表示法

算法的执行效率，粗略地讲，就是算法代码执行的时间。但是，如何在不运行代码的情况下，用“肉眼”得到一段代码的执行时间呢？

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

通过这段代码执行时间的推导过程，我们可以得到一个非常重要的规律，那就是，所有代码的执行时间 T(n) 与每行代码的执行次数 n 成正比。

$$T(n) = O(f(n))$$

其中，T(n) 我们已经讲过了，它表示代码执行的时间；n 表示数据规模的大小；f(n) 表示每行代码执行的次数总和。因为这是一个公式，所以用 f(n) 来表示。公式中的 O，表示代码的执行时间 T(n) 与 f(n) 表达式成正比。

`T(n) = O(n^3*2)` 这就是大 O 时间复杂度表示法,大 O 时间复杂度实际上并不具体表示代码真正的执行时间，而是表示代码执行时间随数据规模增长的变化趋势，所以，也叫作渐进时间复杂度（asymptotic time complexity），简称时间复杂度。

当n很大时,你可以把它想象成 10000、100000。而公式中的低阶、常量、系数三部分并不左右增长趋势，所以都可以忽略。我们只需要记录一个最大量级就可以了

时间复杂度分析

1.只关注循环执行次数最多的一段代码

大 O 这种复杂度表示方法只是表示一种变化趋势。我们通常会忽略掉公式中的常量、低阶、系数，只需要记录一个最大阶的量级就可以了。所以，我们在分析一个算法、一段代码的时间复杂度的时候，也只关注循环执行次数最多的那一段代码就可以了。这段核心代码执行次数的 n 的量级，就是整段要分析代码的时间复杂度。

```
def cal(n):
    sum = 0
    i = 1
    for i in rang(n+1):
```

```
        sum += i
        i+=1
    return sum
```

其中第 2、3 行代码都是常量级的执行时间，与 n 的大小无关，所以对于复杂度并没有影响。循环执行次数最多的是第 4、5 行代码，所以这块代码要重点分析。前面我们也讲过，这两行代码被执行了 n 次，所以总的时间复杂度就是 O(n)。

2.加法法则：总复杂度等于量级最大的那段代码的复杂度

```
def cal(n):
    sum = 0
    i = 1

    for in range(100):
        sum += i

    for i in rang(n+1):
        sum += i
        i+=1

    for i in range(n+1):
        for j in range(n+1):
            pass
```

综合这三段代码的时间复杂度，我们取其中最大的量级。所以，整段代码的时间复杂度就为 O(n2)。也就是说：总的时间复杂度就等于量级最大的那段代码的时间复杂度

最坏时间复杂度

分析算法时，存在几种可能的考虑：

- 算法完成工作最少需要多少基本操作，即最优时间复杂度
- 算法完成工作最多需要多少基本操作，即最坏时间复杂度
- 算法完成工作平均需要多少基本操作，即平均时间复杂度

对于列表排序,要查找的变量 x 可能出现在数组的任意位置。如果数组中第一个元素正好是要查找的变量 x，那就不需要继续遍历剩下的 n-1 个数据了，那时间复杂度就是 O(1)。但如果数组中不存在变量 x，那我们就需要把整个数组都遍历一遍，时间复杂度就成了 O(n)。所以，不同的情况下，这段代码的时间复杂度是不一样的。
对于最优时间复杂度，其价值不大，因为它没有提供什么有用信息，其反映的只是最乐观最理想的情况，没有参考价值。
对于最坏时间复杂度，提供了一种保证，表明算法在此种程度的基本操作中一定能完成工作。
对于平均时间复杂度，是对算法的一个全面评价，因此它完整全面的反映了这个算法的性质。但另一方面，这种

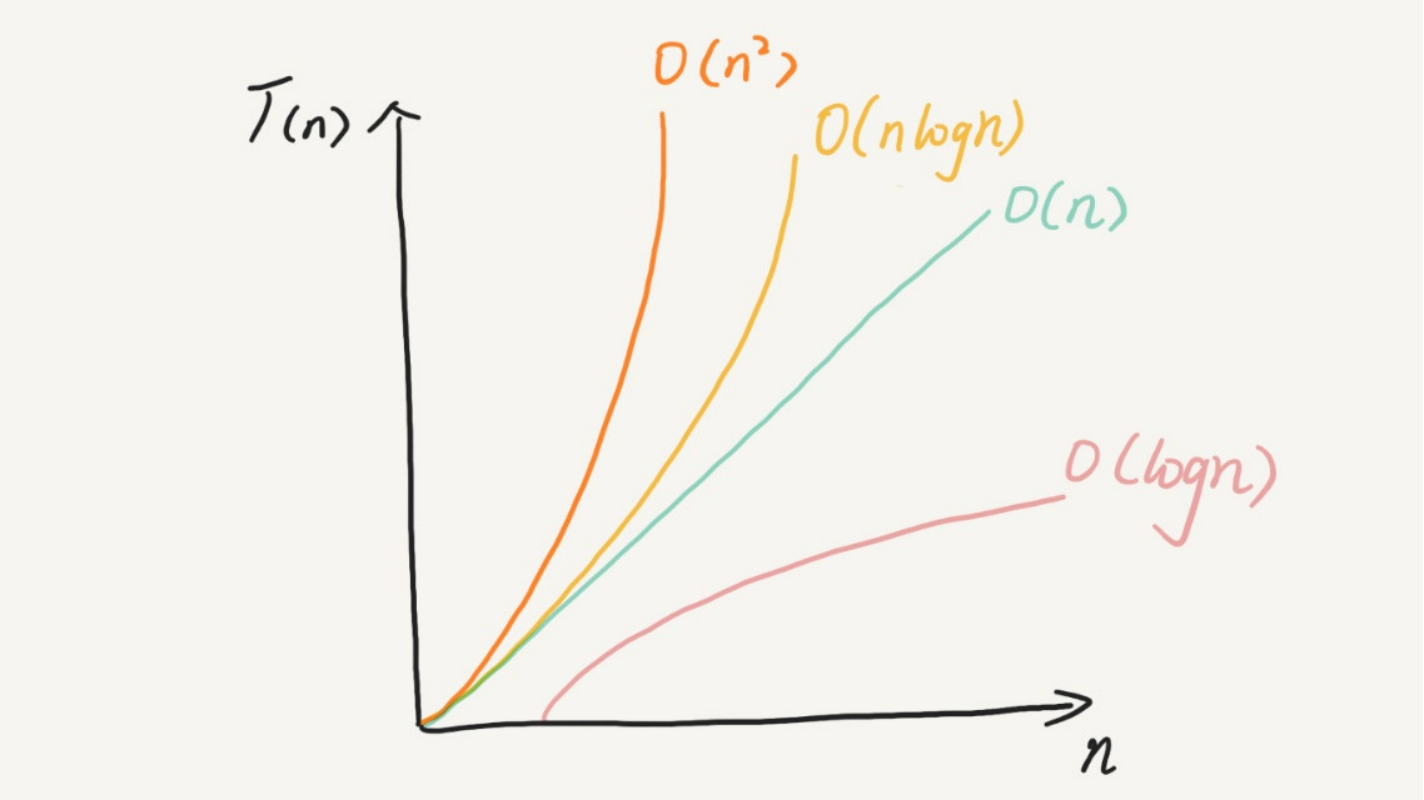
衡量并没有保证，不是每个计算都能在这个基本操作内完成。而且，对于平均情况的计算，也会因为应用算法的实例分布可能并不均匀而难以计算。

因此，我们主要关注算法的最坏情况，亦即最坏时间复杂度。

常见时间复杂度

常见的复杂度并不多，从低阶到高阶有：O(1)、O(logn)、O(n)、O(nlogn)、O(n2)。

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 + 3n + 1	O(n^2)	平方阶
log2n + 20	O(logn)	对数阶
2n+3nlog2n+19	O(nlogn)	nlogn阶
2^n	O(2^n)	指数阶



4-Python内置类型性能分析

代码执行时间测量模块

```
li = []
li.append()
li.insert()
```

timeit模块

timeit模块可以用来测试一小段Python代码的执行速度。

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


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）；

setup参数是运行代码时需要的设置；

timer参数是一个定时器函数，与平台有关。
```

timeit.Timer.timeit(number=1000000)

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。方法返回执行代码的平均耗时，一个float类型的秒数。

list的操作测试

```
def test1():
    l = []
    for i in range(1000):
        l = l + [i]
```

```
def test2():
    l = []
    for i in range(1000):
        l.append(i)

def test3():
    l = [i for i in range(1000)]

def test4():
    l = list(range(1000))

def test5():
    li = []
    for i in range(1000):
        li.insert(0, i)

from timeit import Timer

t1 = Timer("test1()", "from __main__ import test1")
print("add", t1.timeit(number=1000))
t2 = Timer("test2()", "from __main__ import test2")
print("append ",t2.timeit(number=1000))
t3 = Timer("test3()", "from __main__ import test3")
print("list derivation",t3.timeit(number=1000))
t4 = Timer("test4()", "from __main__ import test4")
print("list range ",t4.timeit(number=1000))
t5 = Timer("test5()", "from __main__ import test5")
print("list insert ",t5.timeit(number=1000))
```

5-数据结构

数据结构

我们如何用Python中的类型来保存一个班的学生信息？ 如果想要快速的通过学生姓名获取其信息呢？

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

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

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

概念

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

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

算法与数据结构的区别

数据结构只是静态的描述了数据元素之间的关系。
高效的程序需要在数据结构的基础上设计和选择算法。
程序 = 数据结构 + 算法
总结：算法是为了解决实际问题而设计的，数据结构是算法需要处理的问题载体

抽象数据类型(Abstract Data Type)

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

6-顺序表

在学习顺序表之前,让我们来思考几个问题

- 列表的下标为什么从零开始
- 为什么列表append比insert快
- 列表append之后,id值为什么不变,也可以说内存地址不变

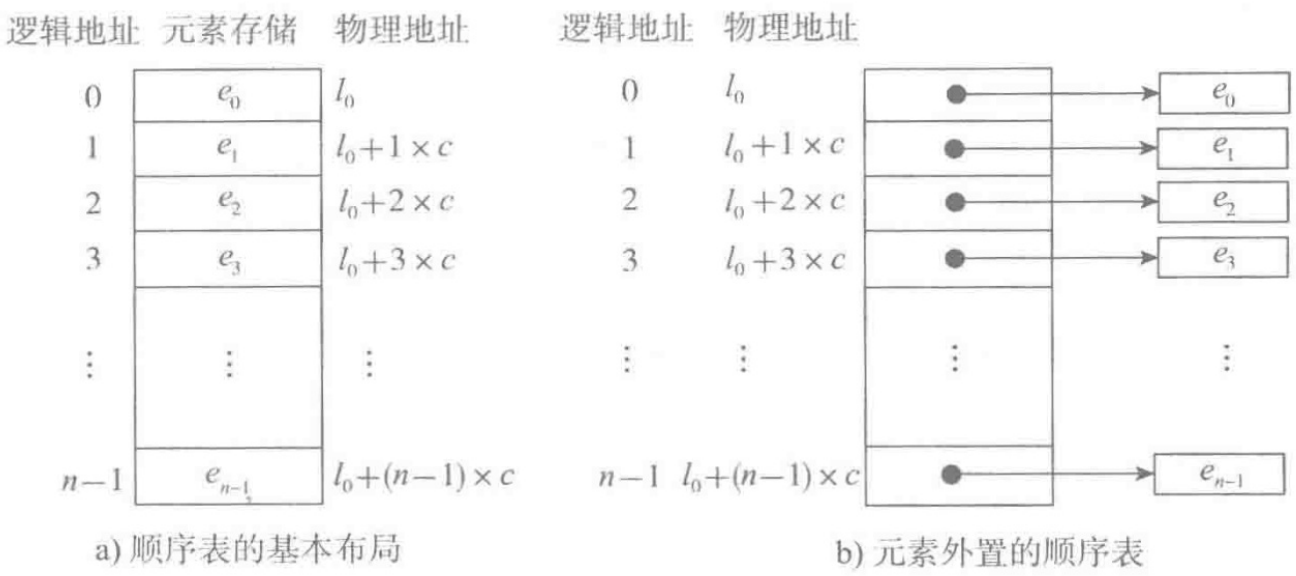
在程序中，经常需要将一组（通常是同为某个类型的）数据元素作为整体管理和使用，需要创建这种元素组，用变量记录它们，传进传出函数等。一组数据中包含的元素个数可能发生变化（可以增加或删除元素）。

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

这样的一组序列元素的组织形式，我们可以将其抽象为线性表。一个线性表是某类元素的一个集合，还记录着元素之间的一种顺序关系。线性表是最基本的数据结构之一，在实际程序中应用非常广泛，它还经常被用作更复杂的数据结构的实现基础。
根据线性表的实际存储方式，分为两种实现模型：

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

顺序表的基本形式



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

Loc(ei) = Loc(e0) + c*i

故，访问指定元素时无需从头遍历，通过计算便可获得对应地址，其时间复杂度为O(1)。

如果元素的大小不统一，则须采用图b的元素外置的形式，将实际数据元素另行存储，而顺序表中各单元位置保存对应元素的地址信息（即链接）。由于每个链接所需的存储量相同，通过上述公式，可以计算出元素链接的存储位置，而后顺着链接找到实际存储的数据元素。注意，图b中的c不再是数据元素的大小，而是存储一个链接地址所需的存储量，这个量通常很小。

图b这样的顺序表也被称为对实际数据的索引，这是最简单的索引结构。

数组要从0开始编号，而不是从1开始呢？

从数组存储的内存模型上来看，“下标”最确切的定义应该是“偏移（offset）”。前面也讲到，如果用 a 来表示数组的首地址，a[0] 就是偏移为 0 的位置，也就是首地址，a[k] 就表示偏移 k 个 type_size 的位置，所以计算 a[k] 的内存地址只需要用这个公式：

```
a[k]_address = base_address + k * type_size
```

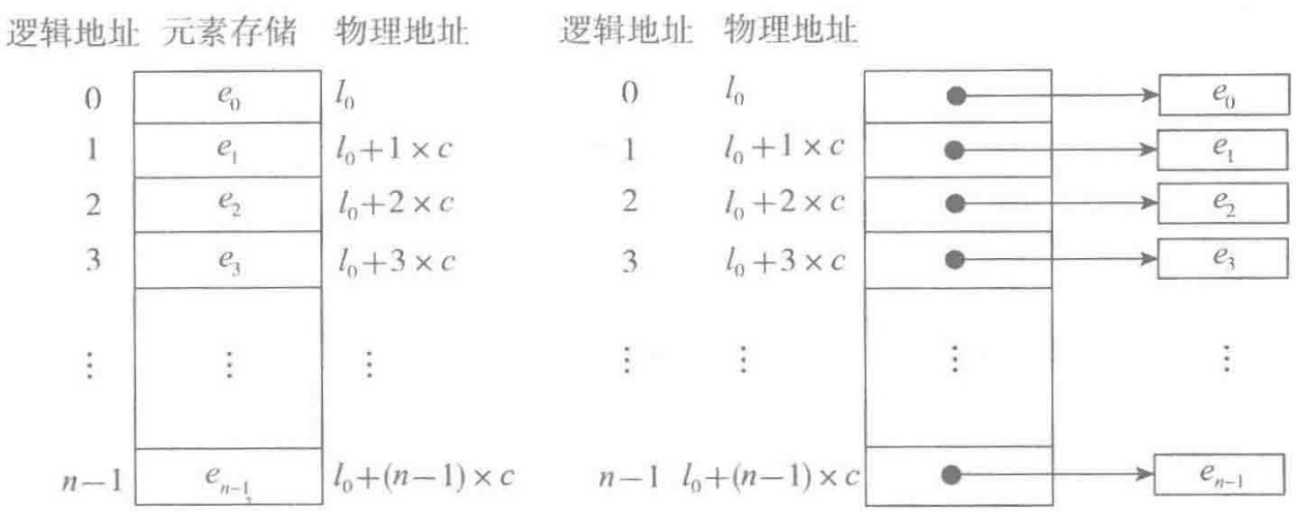
但是，如果数组从 1 开始计数，那我们计算数组元素 a[k] 的内存地址就会变为：

```
a[k]_address = base_address + (k-1)*type_size
```

对比两个公式，我们不难发现，从 1 开始编号，每次随机访问数组元素都多了一次减法运算，对于 CPU 来说，就是多了一次减法指令。数组作为非常基础的数据结构，通过下标随机访问数组元素又是其非常基础的编程操作，效率的优化就要尽可能做到极致。所以为了减少一次减法操作，数组选择了从 0 开始编号，而不是从 1 开

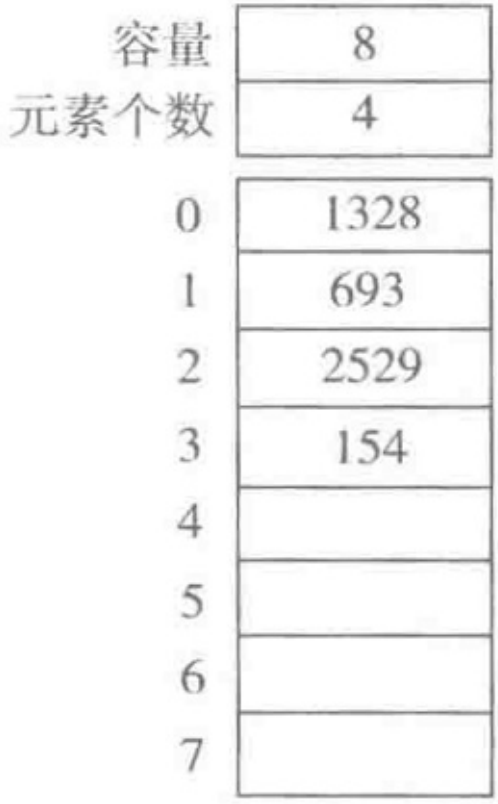
始。

基本顺序表与元素外围顺序表



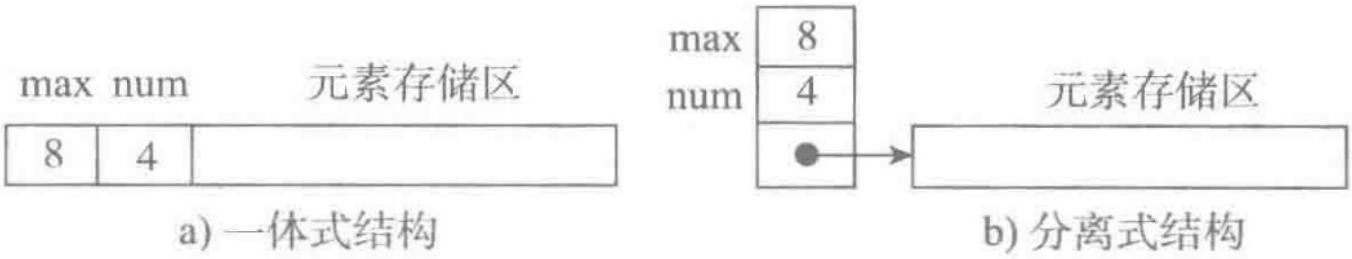
a) 顺序表的基本布局　　b) 元素外置的顺序表

顺序表的结构



一个顺序表的完整信息包括两部分，一部分是表中的元素集合，另一部分是为实现正确操作而需记录的信息，即有关表的整体情况的信息，这部分信息主要包括元素存储区的容量和当前表中已有的元素个数两项。

顺序表的两种基本实现方式



图a为一体式结构，存储表信息的单元与元素存储区以连续的方式安排在一块存储区里，两部分数据的整体形成一个完整的顺序表对象。

一体式结构整体性强，易于管理。但是由于数据元素存储区域是表对象的一部分，顺序表创建后，元素存储区就固定了。

图b为分离式结构，表对象里只保存与整个表有关的信息（即容量和元素个数），实际数据元素存放在另一个独立的元素存储区里，通过链接与基本表对象关联。

元素存储区替换

一体式结构由于顺序表信息区与数据区连续存储在一起，所以若想更换数据区，则只能整体搬迁，即整个顺序表对象（指存储顺序表的结构信息的区域）改变了。

分离式结构若想更换数据区，只需将表信息区中的数据区链接地址更新即可，而该顺序表对象不变。

元素存储区扩充

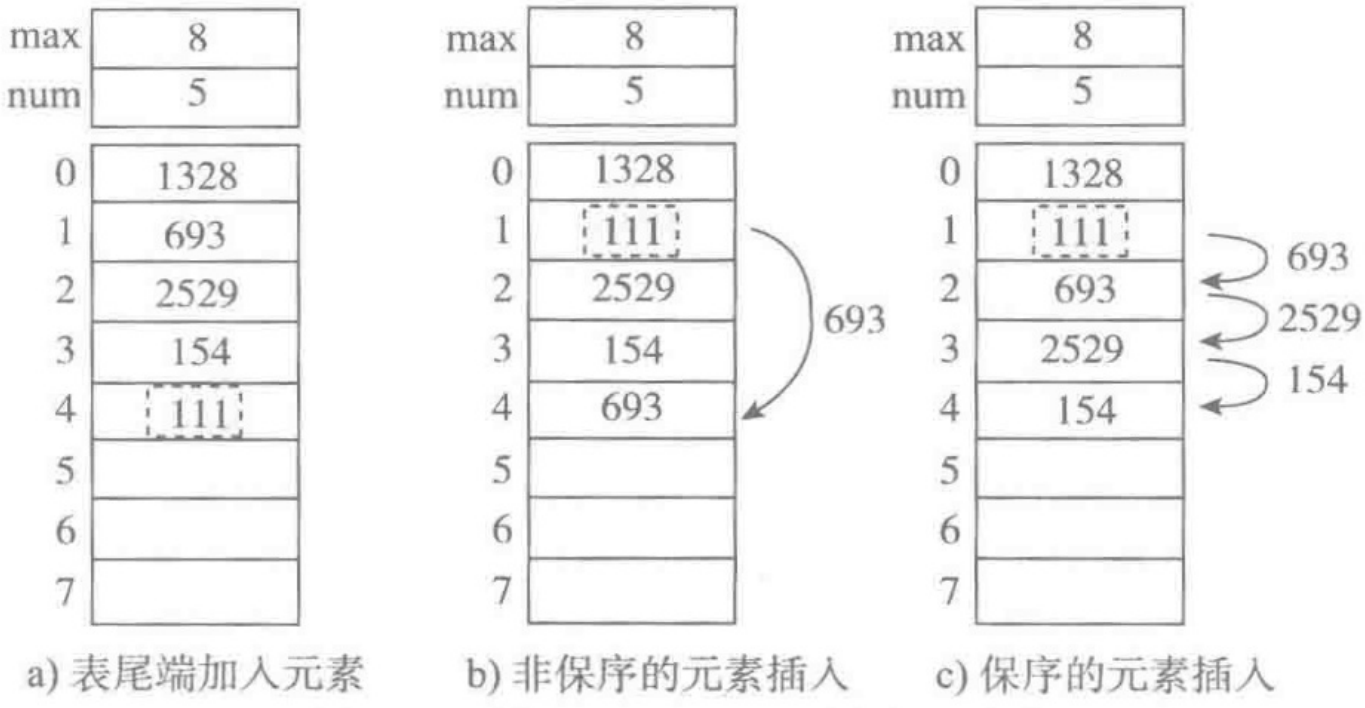
采用分离式结构的顺序表，若将数据区更换为存储空间更大的区域，则可以在不改变表对象的前提下对其数据存储区进行了扩充，所有使用这个表的地方都不必修改。只要程序的运行环境（计算机系统）还有空闲存储，这种表结构就不会因为满了而导致操作无法进行。人们把采用这种技术实现的顺序表称为动态顺序表，因为其容量可以在使用中动态变化。

扩充的两种策略

- 每次扩充增加固定数目的存储位置,如每次扩充增加10个元素位置,这种策略可称为线性增长。
  - 特点:节省空间，但是扩充操作频繁,操作次数多。
- 每次扩充容量加倍,如每次扩充增加一倍存储空间。
  - 特点:减少了扩充操作的执行次数,但可能会浪费空间资源。以空间换时间,推荐的方式。

顺序表的操作

增加元素

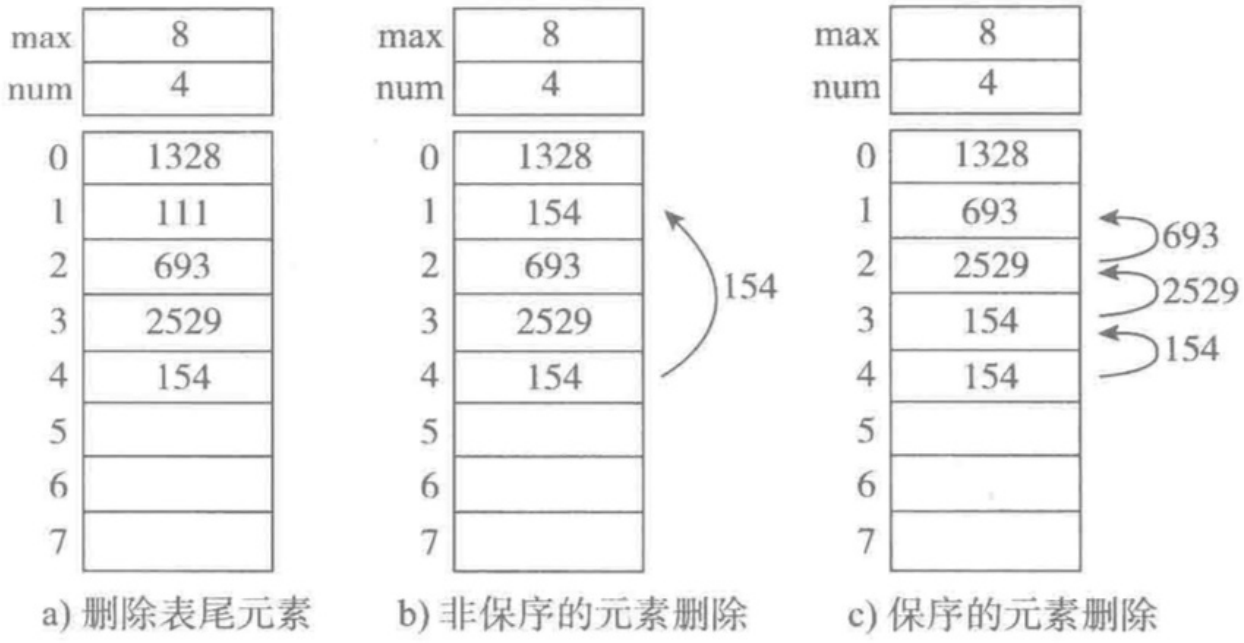


a. 尾端加入元素，时间复杂度为O(1)

b. 非保序的加入元素（不常见），时间复杂度为O(1)

c. 保序的元素加入，时间复杂度为O(n) # 保序是插入元素,原先的顺序不变

删除元素



a. 删除表尾元素，时间复杂度为O(1)

b. 非保序的元素删除（不常见），时间复杂度为O(1)

c. 保序的元素删除，时间复杂度为O(n)

Python中的顺序表

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

list的基本实现技术

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

- 基于下标（位置）的高效元素访问和更新，时间复杂度应该是O(1)；

为满足该特征，应该采用顺序表技术，表中元素保存在一块连续的存储区中。

- 允许任意加入元素，而且在不断加入元素的过程中，表对象的标识（函数id得到的值）不变。

为满足该特征，就必须能更换元素存储区，并且为保证更换存储区时list对象的标识id不变，只能采用分离式实现技术。
在Python的官方实现中，list就是一种采用分离式技术实现的动态顺序表。这就是为什么用list.append(x)（或list.insert(len(list), x)，即尾部插入）比在指定位置插入元素效率高的原因。
在Python的官方实现中，list实现采用了如下的策略：在建立空表（或者很小的表）时，系统分配一块能容纳8个元素的存储区；在执行插入操作（insert或append）时，如果元素存储区满就换一块4倍大的存储区。但如果此时的表已经很大（目前的阀值为50000），则改变策略，采用加一倍的方法。引入这种改变策略的方式，是为了避免出现过多空闲的存储位置。

7-链表

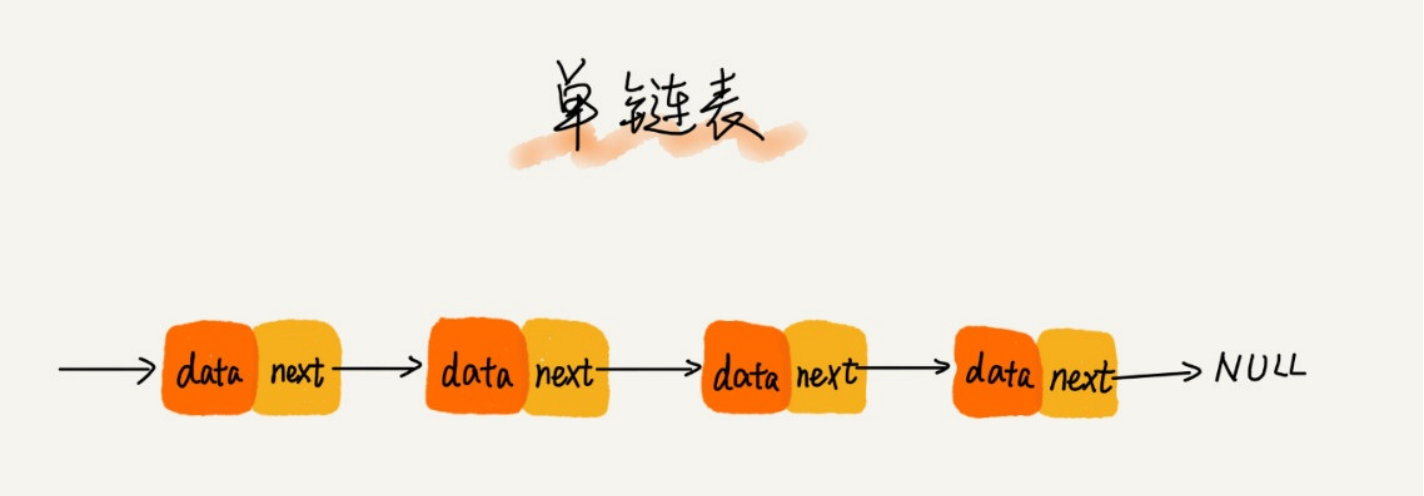
为什么需要链表

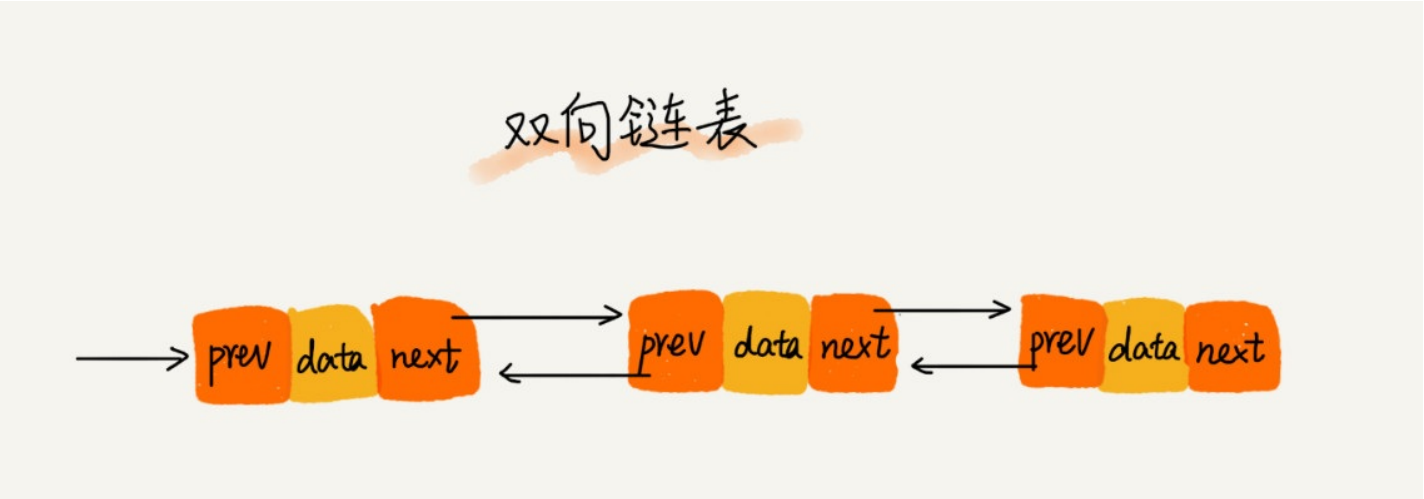
顺序表的构建需要预先知道数据大小来申请连续的存储空间，而在进行扩充时又需要进行数据的搬迁，所以使用起来并不是很灵活。

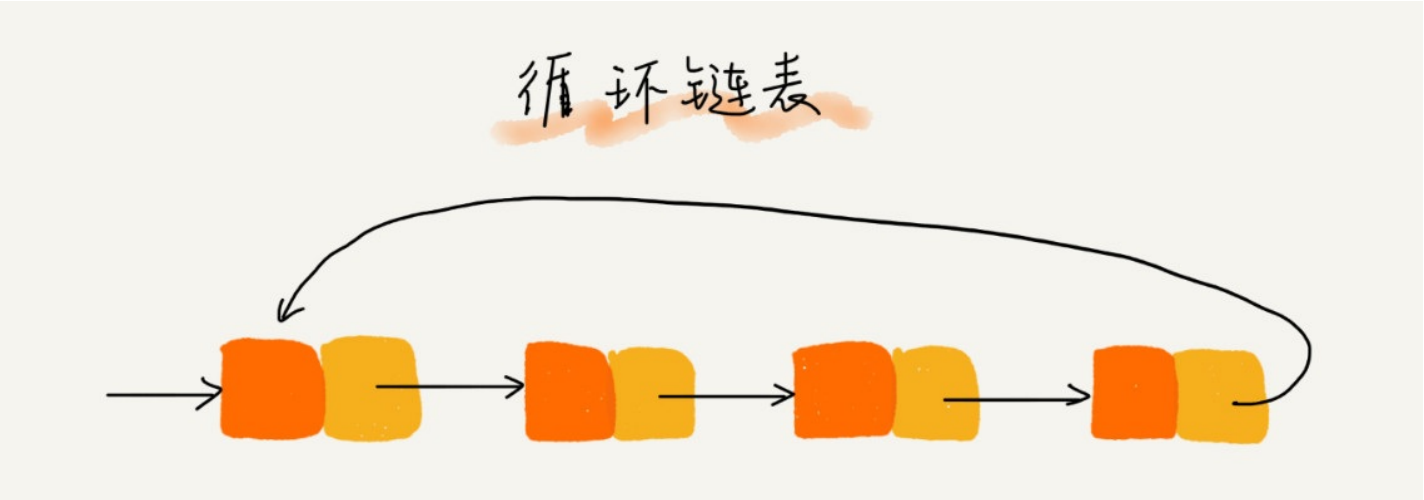
链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。

链表的定义

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



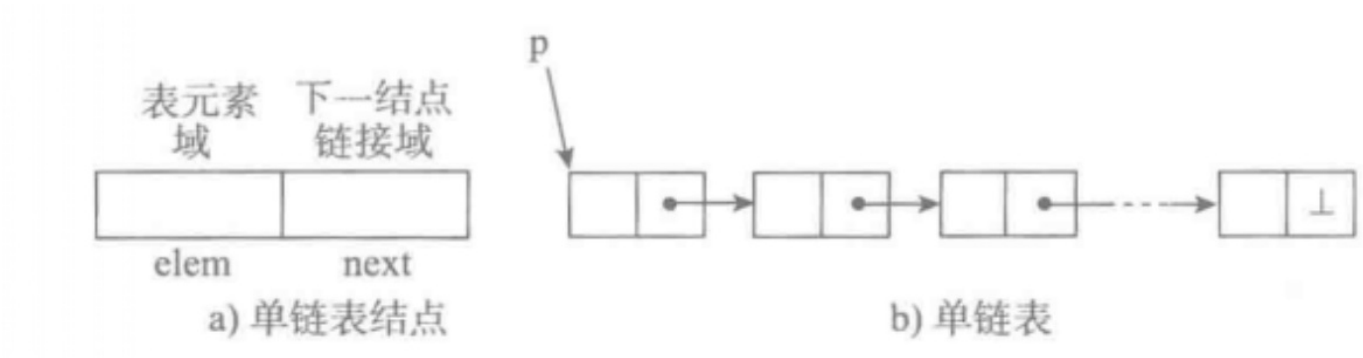

双向链表
prev
data
next
prev
data
next
prev
data
next


循环链表

7-1-单向链表

单向链表

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



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

节点实现

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

单链表的操作

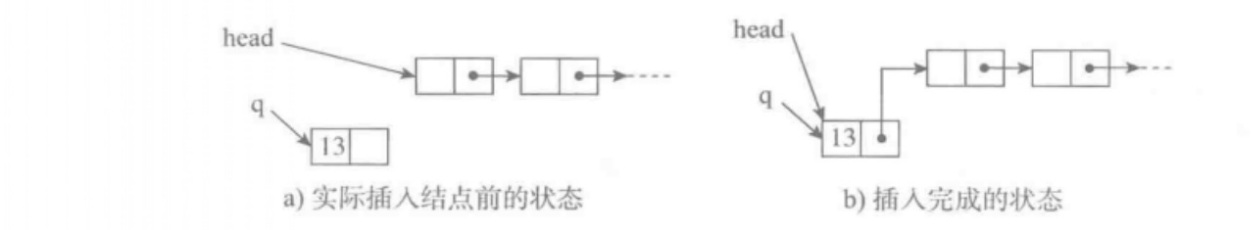
- is_empty() 链表是否为空
- length() 链表长度

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

单链表的实现

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

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

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

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

头部添加元素

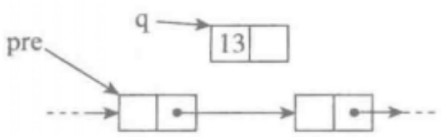


```
def add(self, item):
    """头部添加元素"""
    # 先创建一个保存item值的节点
    node = SingleNode(item)
    # 将新节点的链接域next指向头节点，即_head指向的位置
    node.next = self.__head
    # 将链表的头_head指向新节点
    self._head = node
```

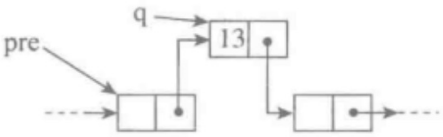
尾部添加元素

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

指定位置添加元素



a) 实际插入结点前的状态

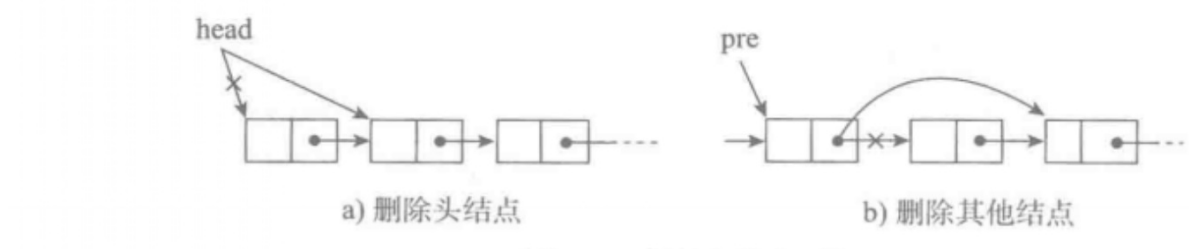


b) 插入完成的状态

```
def insert(self, pos, item):
    """指定位置添加元素"""
    # 若指定位置pos为第一个元素之前，则执行头部插入
    if pos <= 0:
        self.add(item)
    # 若指定位置超过链表尾部，则执行尾部插入
    elif pos > (self.length()-1):
        self.append(item)
    # 找到指定位置
    else:
        node = SingleNode(item)
        count = 0
        # pre用来指向指定位置pos的前一个位置pos-1，初始从头节点开始移动到指定位置
        pre = self._head
```

```
        while count < (pos-1):
            count += 1
            pre = pre.next
        # 先将新节点node的next指向插入位置的节点
        node.next = pre.next
        # 将插入位置的前一个节点的next指向新节点
        pre.next = node
```

删除节点




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

查找节点是否存在

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

链表与顺序表的对比

链表失去了顺序表随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大，但对存储空间的使用要相对灵活。

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)

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

写链表代码建议

- 理解指针或引用的含义
- 警惕指针丢失
- 重点留意边界条件处理
- 举例画图，辅助思考

如何实现LRU缓存淘汰算法?

缓存是一种提高数据读取性能的技术，在硬件设计、软件开发中都有着非常广泛的应用，比如常见的 CPU 缓存、数据库缓存、浏览器缓存等等。

缓存的大小有限，当缓存被用满时，哪些数据应该被清理出去，哪些数据应该被保留？这就需要缓存淘汰策略来决定。常见的策略有三种：先进先出策略 FIFO（First In，First Out）、最少使用策略 LFU（Least Frequently Used）、最近最少使用策略 LRU（Least Recently Used）。

这些策略你不用死记，我打个比方你很容易就明白了。假如说，你买了很多本技术书，但有一天你发现，这些书太多了，太占书房空间了，你要做个大扫除，扔掉一些书籍。那这个时候，你会选择扔掉哪些书呢？对应一下，你的选择标准是不是和上面的三种策略神似呢？

我的思路是这样的：我们维护一个有序单链表，越靠近链表尾部的结点是越早之前访问的。当有一个新的数据被访问时，我们从链表头开始顺序遍历链表。

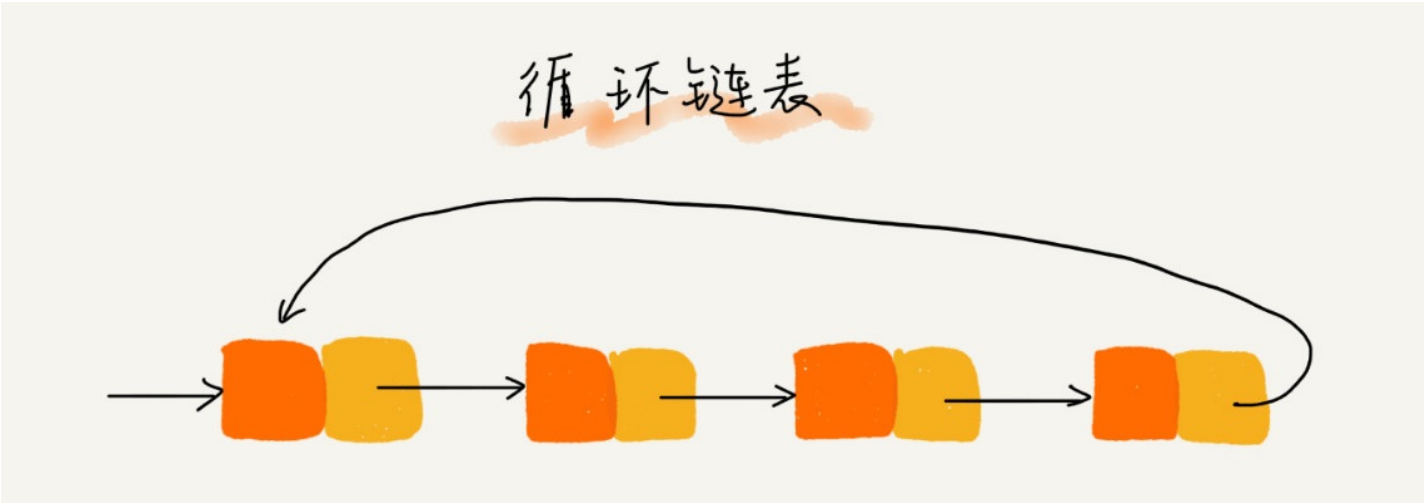
1. 如果此数据之前已经被缓存在链表中了，我们遍历得到这个数据对应的结点，并将其从原来的位置删除，然后再插入到链表的头部。
2. 如果此数据没有在缓存链表中，又可以分为两种情况：

- 如果此时缓存未满，则将此结点直接插入到链表的头部；
- 如果此时缓存已满，则链表尾结点删除，将新的数据结点插入链表的头部。

7-2-单向循环链表

循环链表

循环链表是一种特殊的单链表。实际上，循环链表也很简单。它跟单链表唯一的区别就在尾结点。我们知道，单链表的尾结点指针指向空地址，表示这就是最后的结点了。而循环链表的尾结点指针是指向链表的头结点。从我画的循环链表图中，你应该可以看出来，它像一个环一样首尾相连，所以叫作“循环”链表。



操作

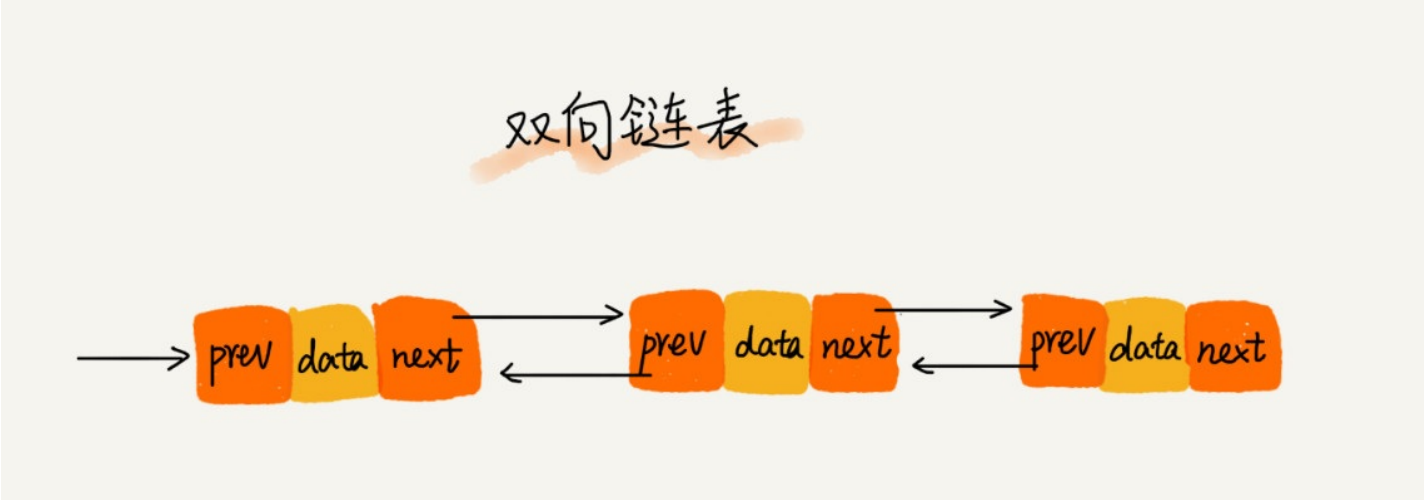
- is_empty() 判断链表是否为空
- length() 返回链表的长度
- travel() 遍历
- add(item) 在头部添加一个节点
- append(item) 在尾部添加一个节点
- insert(pos, item) 在指定位置pos添加节点
- remove(item) 删除一个节点
- search(item) 查找节点是否存在

7-3-双向链表

双向链表

接下来我们再来看一个稍微复杂的，在实际的软件开发中，也更加常用的链表结构：双向链表。

单向链表只有一个方向，结点只有一个后继指针 next 指向后面的结点。而双向链表，顾名思义，它支持两个方向，每个结点不止有一个后继指针 next 指向后面的结点，还有一个前驱指针 prev 指向前面的结点。



操作

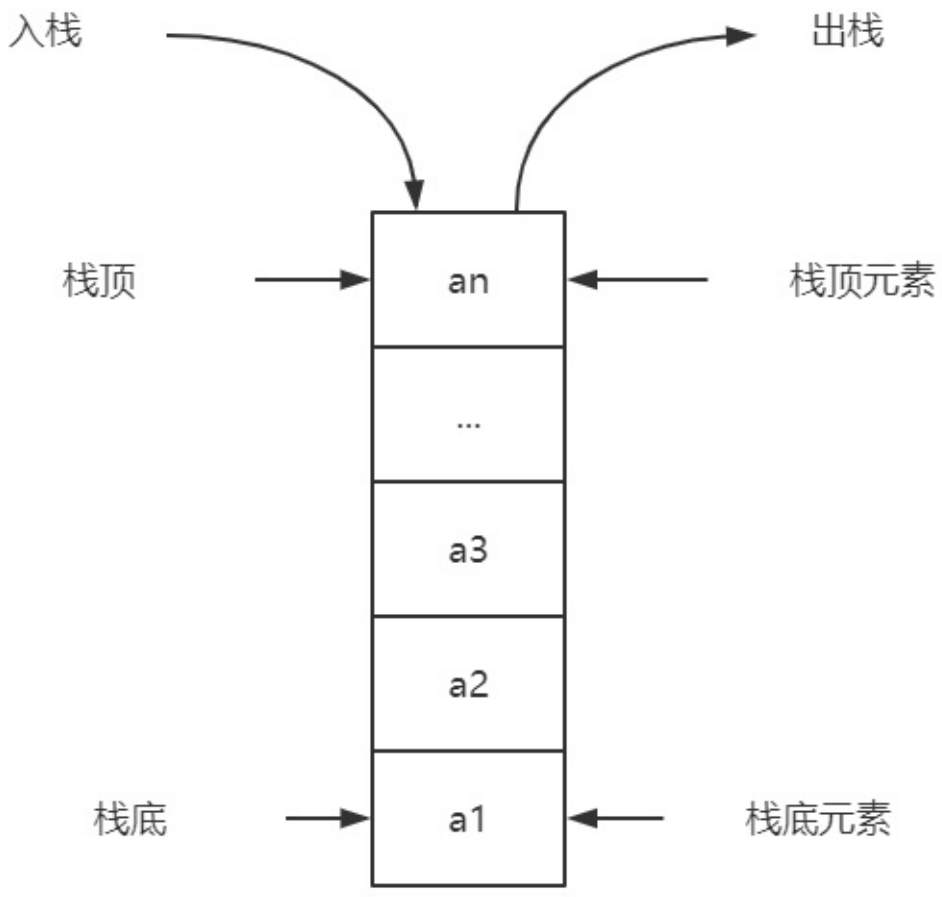
- is_empty() 链表是否为空
- length() 链表长度
- travel() 遍历链表
- add(item) 链表头部添加
- append(item) 链表尾部添加
- insert(pos, item) 指定位置添加
- remove(item) 删除节点
- search(item) 查找节点是否存在

8-栈与队列

栈与队列

如何理解“栈”？

关于“栈”，我有一个非常贴切的例子，就是一摞叠在一起的盘子。我们平时放盘子的时候，都是从下往上一个一个放；取的时候，我们也是从上往下一个一个地依次取，不能从中间任意抽出。后进者先出，先进者后出，这就是典型的“栈”结构。



从栈的操作特性上来看，栈是一种“操作受限”的线性表，只允许在一端插入和删除数据

大家有没有觉得，相比链表，栈带给我的只有限制，并没有任何优势。那我直接使用数组或者链表不就好了吗？为什么还要用这个“操作受限”的“栈”呢？

事实上，从功能上来说，链表确实可以替代栈，但你要知道，特定的数据结构是对特定场景的抽象，而且，数组或链表暴露了太多的操作接口，操作上的确灵活自由，但使用时就比较不可控，自然也就更容易出错。

当某个数据集合只涉及在一端插入和删除数据，并且满足后进先出、先进后出的特性，我们就应该首选“栈”这种数据结构。

如何实现一个“栈”？

从刚才栈的定义里，我们可以看出，栈主要包含两个操作，入栈和出栈，也就是在栈顶插入一个数据和从栈顶删除一个数据。理解了栈的定义之后，我们来看一看如何用代码实现一个栈。

实际上，栈既可以用顺序表来实现，也可以用链表来实现。用顺序表实现的栈，我们叫作顺序栈，用链表实现的栈，我们叫作链式栈。

栈的操作

- Stack() 创建一个新的空栈
- push(item) 添加一个新的元素item到栈顶
- pop() 弹出栈顶元素
- 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

案例

浏览器的前进、后退功能，我想你肯定很熟悉吧？

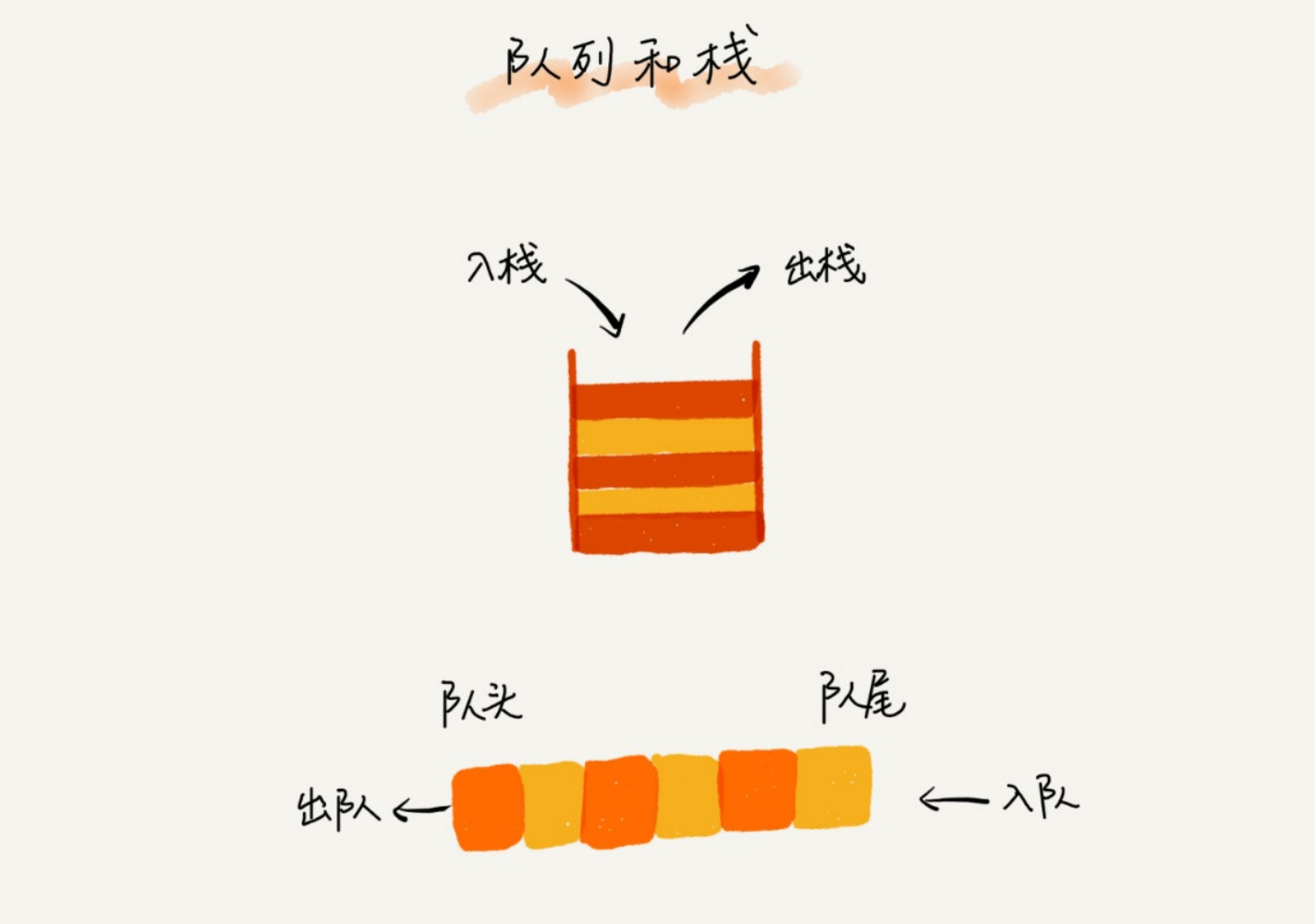
当你依次访问完一串页面 a-b-c 之后，点击浏览器的后退按钮，就可以查看之前浏览过的页面 b 和 a。当你后退到页面 a，点击前进按钮，就可以重新查看页面 b 和 c。但是，如果你后退到页面 b 后，点击了新的页面 d，那就无法再通过前进、后退功能查看页面 c 了。

假设你是 Chrome 浏览器的开发工程师，你会如何实现这个功能呢？

队列

队列这个概念非常好理解。你可以把它想象成排队买票，先来的先买，后来的人只能站末尾，不允许插队。先进者先出，这就是典型的“队列”。

我们知道，栈只支持两个基本操作：入栈 push()和出栈 pop()。队列跟栈非常相似，支持的操作也很有限，最基本的操作也是两个：入队 enqueue()，放一个数据到队列尾部；出队 dequeue()，从队列头部取一个元素。



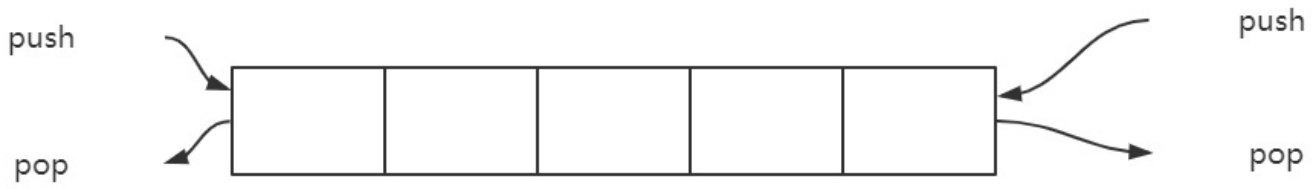
所以，队列跟栈一样，也是一种操作受限的线性表数据结构。
队列的概念很好理解，基本操作也很容易掌握。作为一种非常基础的数据结构，队列的应用也非常广泛，特别是一些具有某些额外特性的队列，比如循环队列、阻塞队列、并发队列。它们在很多偏底层系统、框架、中间件的开发中，起着关键性的作用。

队列的实现

- Queue() 创建一个空的队列
- 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
- is_empty() 判断一个队列是否为空
- size() 返回队列的大小

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。



操作

- Deque() 创建一个空的双端队列
- add_front(item) 从队头加入一个item元素
- add_rear(item) 从队尾加入一个item元素
- remove_front() 从队头删除一个item元素
- remove_rear() 从队尾删除一个item元素
- is_empty() 判断双端队列是否为空
- size() 返回队列的大小

阻塞队列

阻塞队列其实就是在队列基础上增加了阻塞操作。简单来说，就是在队列为空的时候，从队头取数据会被阻塞。因为此时还没有数据可取，直到队列中有了数据才能返回；如果队列已经满了，那么插入数据的操作就会被阻塞，直到队列中有空闲位置后再插入数据，然后再返回。

案例

我们知道，CPU 资源是有限的，任务的处理速度与线程个数并不是线性正相关。相反，过多的线程反而会导致 CPU 频繁切换，处理性能下降。所以，线程池的大小一般都是综合考虑要处理任务的特点和硬件环境，来事先设置的。

当我们向固定大小的线程池中请求一个线程时，如果线程池中没有空闲资源了，这个时候线程池如何处理这个请求？是拒绝请求还是排队请求？各种处理策略又是怎么实现的呢？

9-排序算法

排序算法

排序算法太多了，有很多可能你连名字都没听说过，比如猴子排序、睡眠排序、面条排序等。我只讲众多排序算法中的一小撮，也是最经典的、最常用的：冒泡排序、插入排序、选择排序、归并排序、快速排序、希尔排序

如何分析一个“排序算法”？

学习排序算法，我们除了学习它的算法原理、代码实现之外，更重要的是要学会如何评价、分析一个排序算法。那分析一个排序算法，要从哪几个方面入手呢？

1.排序算法的执行效率

对于排序算法执行效率的分析，我们一般会从这几个方面来衡量：

- 1.时间复杂度
- 2.比较次数和交换（或移动）次数

2. 排序算法的内存消耗

原地排序算法，就是特指空间复杂度是 O(1) 的排序算法

3.排序算法的稳定性

仅仅用执行效率和内存消耗来衡量排序算法的好坏是不够的。针对排序算法，我们还有一个重要的度量指标，稳定性。这个概念是说，如果待排序的序列中存在值相等的元素，经过排序之后，相等元素之间原有的先后顺序不变。

9-1-递归

如何理解“递归”

递归是一种应用非常广泛的算法（或者编程技巧）。之后我们要讲的很多数据结构和算法的编码实现都要用到递归，比如 DFS 深度优先搜索、前中后序二叉树遍历等等。所以，搞懂递归非常重要，否则，后面复杂一些的数据结构和算法学起来就会比较吃力。

递归需要满足的三个条件

刚刚这个例子是非常典型的递归，那究竟什么样的问题可以用递归来解决呢？我总结了三个条件，只要同时满足以下三个条件，就可以用递归来解决。

- 一个问题的解可以分解为几个子问题的解
- 这个问题与分解之后的子问题，除了数据规模不同，求解思路完全一样
- 存在递归终止条件

练习

- 用递归的方式输出l=['jack',('tom',23),'rose',(14,55,67)] 列表内的每一个元素

```
def dp(s):
  if isinstance(s,(int,str)):
    print(s)
  else:
    for item in s:
      dp(item)
l=['jack',('tom',23),'rose',(14,55,67)]
dp(l)
```

9-2-冒泡排序

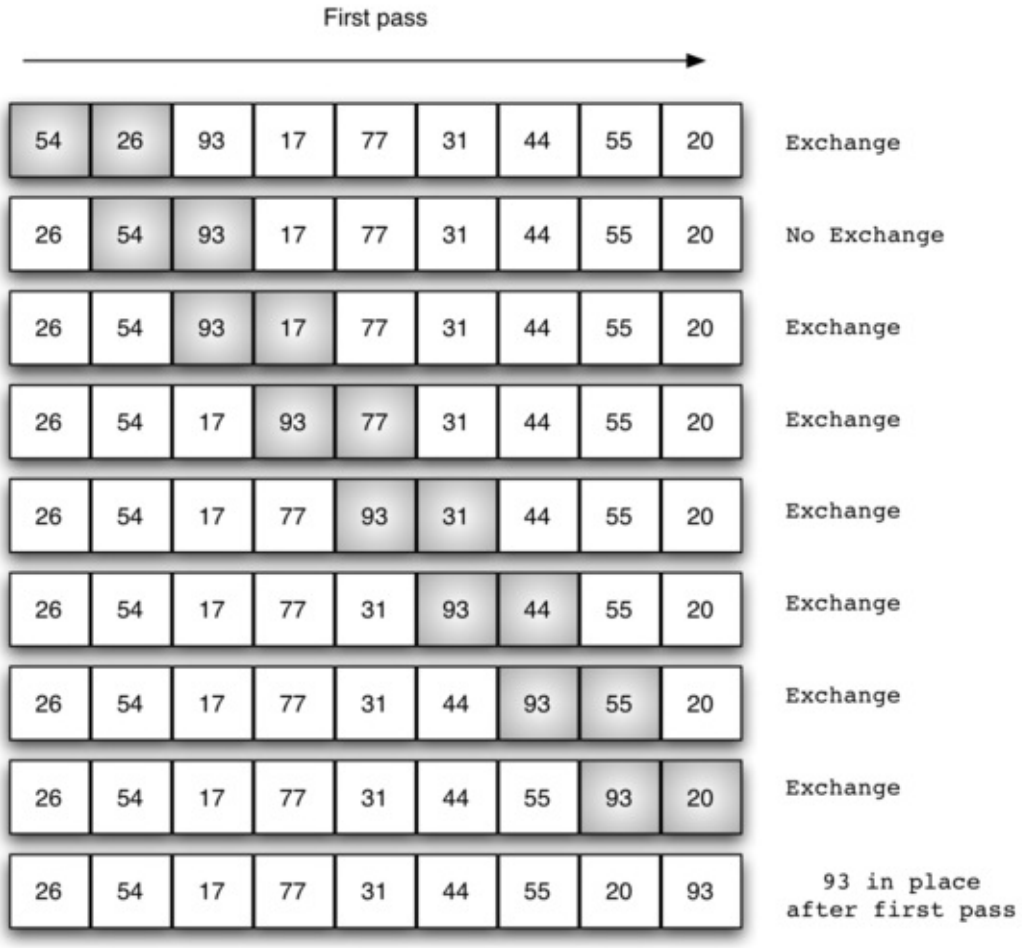
冒泡排序

冒泡排序（英语：Bubble Sort）是一种简单的排序算法。它重复地遍历要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。遍历数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

冒泡排序算法的运作如下：

- 比较相邻的元素。如果第一个比第二个大（升序），就交换他们两个。
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
- 针对所有的元素重复以上的步骤，除了最后一个。
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

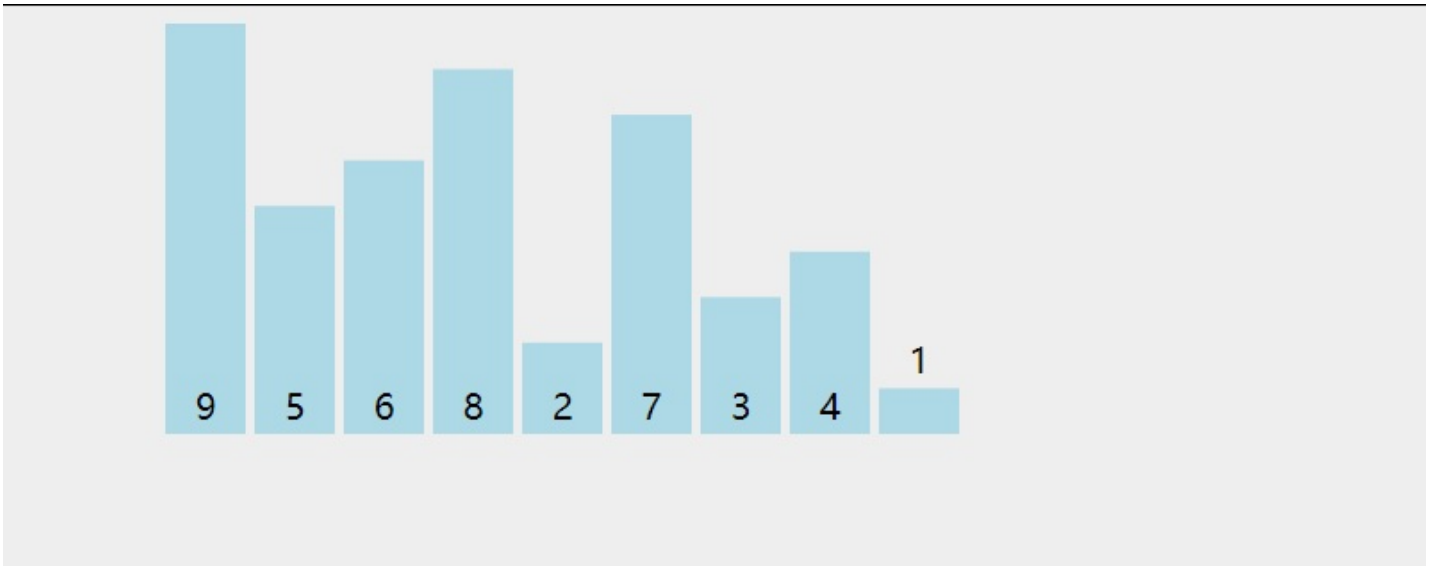
冒泡排序的分析



冒泡排序的实现

```
def bubble_sort(li):
    for j in range(len(li)-1,0,-1):
        # j表示每次遍历需要比较的次数，是逐渐减小的
        for i in range(j):
            if li[i] > li[i+1]:
                li[i], li[i+1] = li[i+1], li[i]

li = [54,26,93,17,77,31,44,55,20]
bubble_sort(li)
print(li)
```

冒泡排序的演示



- 冒泡排序是稳定的排序算法吗？

在冒泡排序中，只有交换才可以改变两个元素的前后顺序。为了保证冒泡排序算法的稳定性，当有相邻的两个元素大小相等的时候，我们不做交换，相同大小的数据在排序前后不会改变顺序，所以冒泡排序是稳定的排序算法。

- 冒泡排序的时间复杂度是多少？

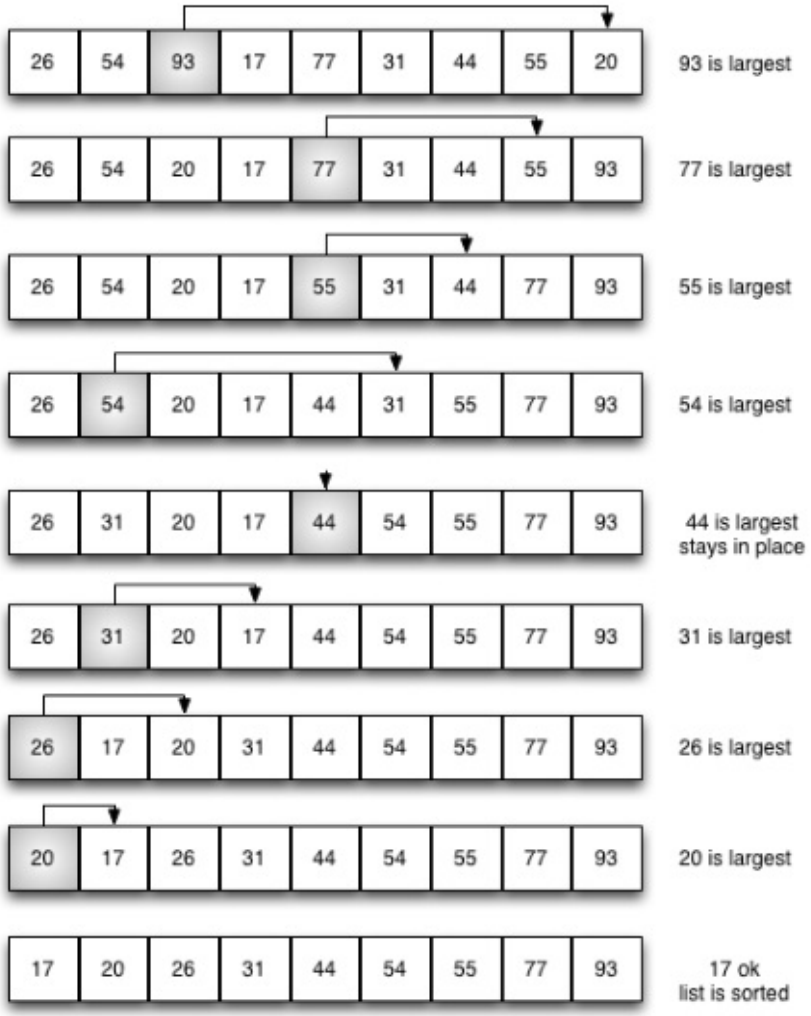
最好情况下，要排序的数据已经是有序的了，我们只需要进行一次冒泡操作，就可以结束了，所以最好情况时间复杂度是 O(n)。而最坏的情况是，要排序的数据刚好是倒序排列的，我们需要进行 n 次冒泡操作，所以最坏情况时间复杂度为 O(n2)。

9-3-选择排序

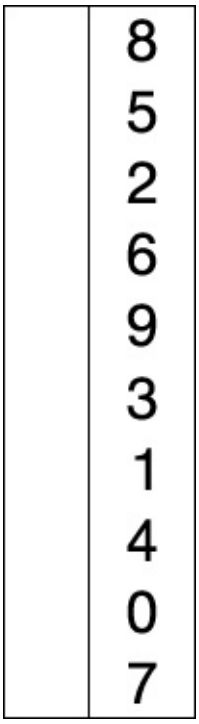
选择排序

选择排序（Selection sort）是一种简单直观的排序算法。它的工作原理如下。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。
选择排序的主要优点与数据移动有关。如果某个元素位于正确的最终位置上，则它不会被移动。选择排序每次交换一对元素，它们当中至少有一个将被移到其最终位置上，因此对n个元素的表进行排序总共进行至多n-1次交换。在所有的完全依靠交换去移动元素的排序方法中，选择排序属于非常好的一种。

选择排序分析



选择排序演示



选择排序的实现

```python
def selection_sort(li):
    """选择排序"""
    n = len(li)
    for i in range(n-1): # 0-7
        # 记录最小位置
        min_index = i
        for j in range(min_index, n):
            # print(min_index)
            if li[j] < li[min_index]:
                min_index = j
        li[i], li[min_index] = li[min_index], li[i]

selection_sort(li)
print(li)
```

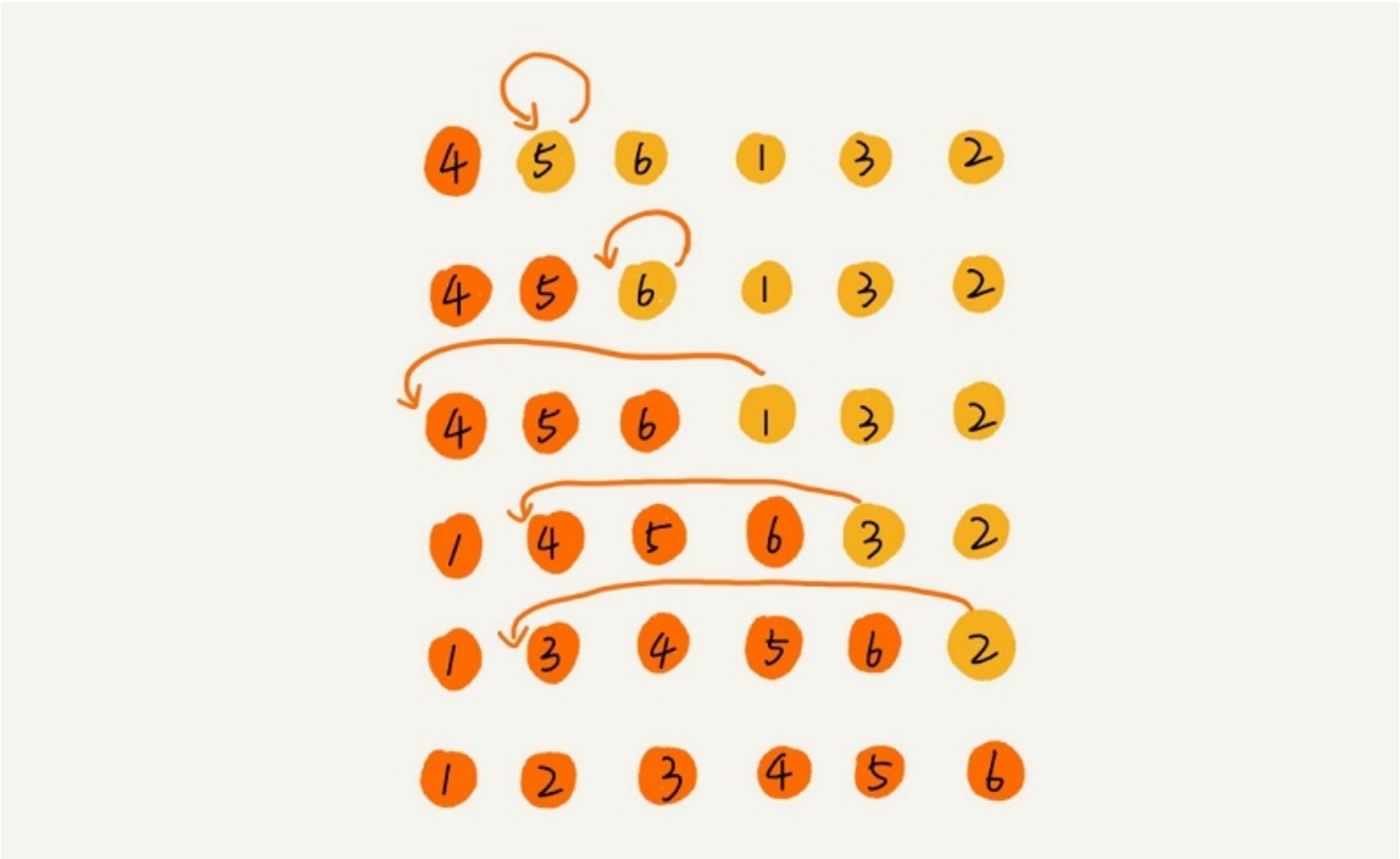
9-4-插入排序

插入排序

插入排序（英语：Insertion Sort）是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

首先，我们将数组中的数据分为两个区间，已排序区间和未排序区间。初始已排序区间只有一个元素，就是数组的第一个元素。插入算法的核心思想是取未排序区间中的元素，在已排序区间中找到合适的插入位置将其插入，并保证已排序区间数据一直有序。重复这个过程，直到未排序区间中元素为空，算法结束。

如图所示，要排序的数据是 4，5，6，1，3，2，其中左侧为已排序区间，右侧是未排序区间。



插入排序也包含两种操作，一种是元素的比较，一种是元素的移动。当我们需要将一个数据 a 插入到已排序区间

时，需要拿 a 与已排序区间的元素依次比较大小，找到合适的插入位置。找到插入点之后，我们还需要将插入点之后的元素顺序往后移动一位，这样才能腾出位置给元素 a 插入。

插入排序演示


```
def insert_sort(li):
    # 从第二个位置，即下标为1的元素开始向前插入
    for i in range(1, len(li)):
        # 从第i个元素开始向前比较，如果小于前一个元素，交换位置
        for j in range(i, 0, -1):
            if li[j] < li[j - 1]:
                li[j], li[j - 1] = li[j - 1], li[j]


li = [54, 26, 93, 17, 77, 31, 44, 55, 20]
insert_sort(li)
print(li)
```

9-5-希尔排序

希尔排序

希尔排序(Shell Sort)是插入排序的一种。也称缩小增量排序，是直接插入排序算法的一种更高效的改进版本。希尔排序是非稳定排序算法。该方法因DL．Shell于1959年提出而得名。 希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

希尔排序过程

希尔排序的基本思想是：将数组列在一个表中并对列分别进行插入排序，重复这过程，不过每次用更长的列（步长更长了，列数更少了）来进行。最后整个表就只有一列了。将数组转换至表是为了更好地理解这算法，算法本身还是使用数组进行排序。
例如，假设有这样一组数[13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10]，如果我们以步长为5开始进行排序，我们可以通过将这列表放在有5列的表中来更好地描述算法，这样他们就应该看起来是这样(竖着的元素是步长组成)：

```
13 14 94 33 82
25 59 94 65 23
45 27 73 25 39
10
```

然后我们对每列进行排序：

```
10 14 73 25 23
13 27 94 33 39
25 59 94 65 82
45
```

将上述四行数字，依序接在一起时我们得到：[10 14 73 25 23 13 27 94 33 39 25 59 94 65 82 45]。这时10

已经移至正确位置了，然后再以3为步长进行排序：

```
10 14 73
25 23 13
27 94 33
39 25 59
94 65 82
45
```

排序之后变为：

```
10 14 13
25 23 33
27 25 59
39 65 73
45 94 82
94
```

希尔排序实现

```
def shell_sort(li):
    n = len(li)
    # 初始步长
    gap = n // 2
    while gap > 0:
        # 按步长进行插入排序 4-8
        for i in range(gap, n):
            # 插入排序
            while i >= gap and li[i - gap] > li[i]:
                li[i - gap], li[i] = li[i], li[i - gap]
                i -= gap
        # 得到新的步长
        gap = gap // 2

li = [54, 26, 93, 17, 77, 31, 44, 55, 20]
shell_sort(li)
print(li)
```

时间复杂度

- 最优时间复杂度：根据步长序列的不同而不同
- 最坏时间复杂度：O(n2)
- 稳定想：不稳定

9-6-快速排序

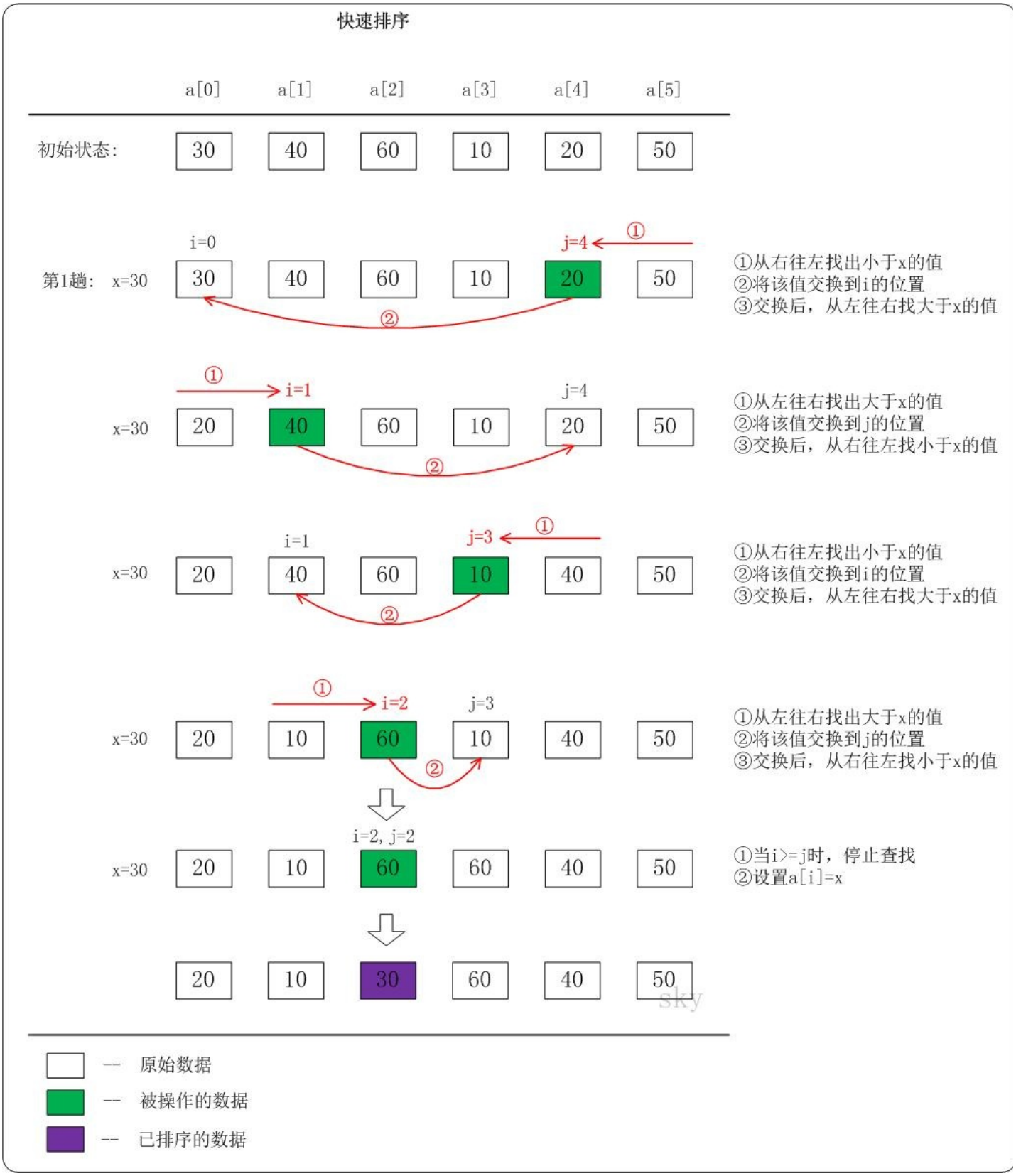
快速排序

快速排序（英语：Quicksort），又称划分交换排序（partition-exchange sort），通过一趟排序将要排序的数据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。
步骤为：

1. 从数列中挑出一个元素，称为"基准"（pivot），
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

递归的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递归下去，但是这个算法总会结束，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它最后的位置去。

快速排序的分析

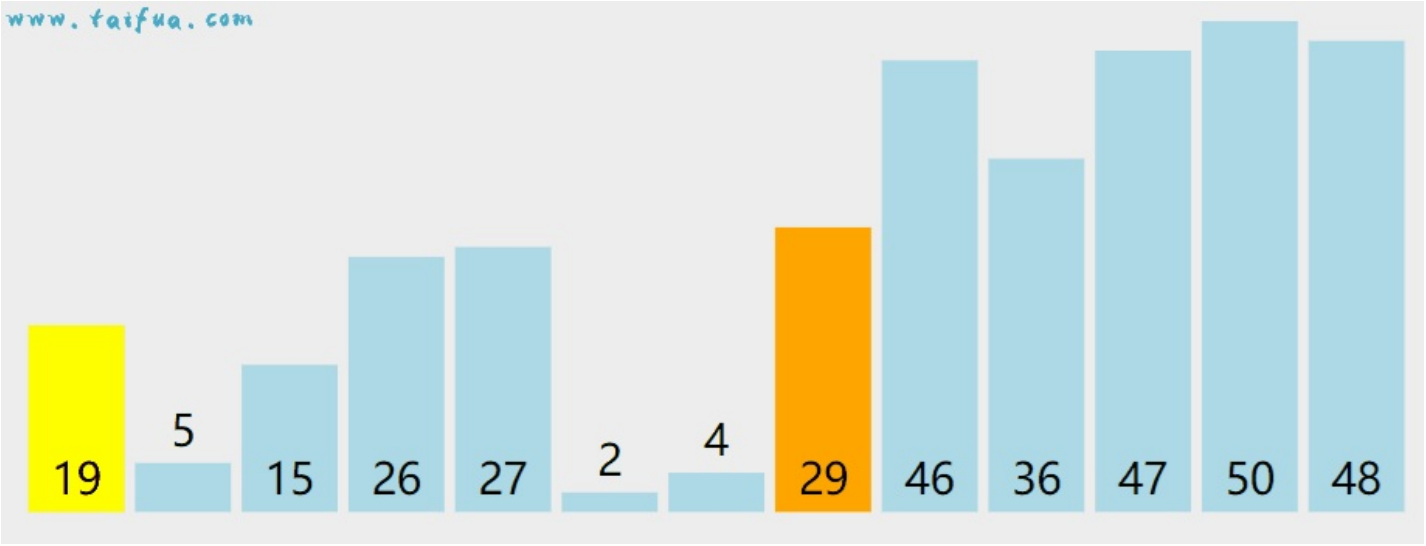


快速排序的实现

```
def quick_sort(alist, start, end):
    """快速排序"""

    # 递归的退出条件
    if start >= end:
        return
```

```
    # 设定起始元素为要寻找位置的基准元素
    mid = alist[start]

    # low为序列左边的由左向右移动的游标
    low = start

    # high为序列右边的由右向左移动的游标
    high = end

    while low < high:
        # 如果low与high未重合，high指向的元素不比基准元素小，则high向左移动
        while low < high and alist[high] >= mid:
            high -= 1
        # 将high指向的元素放到low的位置上
        alist[low] = alist[high]

        # 如果low与high未重合，low指向的元素比基准元素小，则low向右移动
        while low < high and alist[low] < mid:
            low += 1
        # 将low指向的元素放到high的位置上
        alist[high] = alist[low]


    # 退出循环后，low与high重合，此时所指位置为基准元素的正确位置
    # 将基准元素放到该位置
    alist[low] = mid

    # 对基准元素左边的子序列进行快速排序
    quick_sort(alist, start, low-1)

    # 对基准元素右边的子序列进行快速排序
    quick_sort(alist, low+1, end)


alist = [54,26,93,17,77,31,44,55,20]
quick_sort(alist,0,len(alist)-1)
print(alist)
```

快速排序的演示


www.taifua.com
19
5
15
26
27
2
4
29
46
36
47
50
48

9-7-归并排序

归并排序

归并排序是采用分治法的一个非常典型的应用。归并排序的思想就是先递归分解数组，再合并数组。

将数组分解最小之后，然后合并两个有序数组，基本思路是比较两个数组的最前面的数，谁小就先取谁，取了后相应的指针就往后移一位。然后再比较，直至一个数组为空，最后把另一个数组的剩余部分复制过来即可。

归并排序的分析

归并排序的实现

```
def merge_sort(li):
    # 如果列表长度小于1,不在继续拆分
    if len(li) <= 1:
        return li
    # 二分分解
    mid_index = len(li) // 2
    left = merge_sort(li[:mid_index])
    right = merge_sort(li[mid_index:])
    # 合并
    return merge(left,right)

def merge(left, right):
    '''合并操作，将两个有序数组left[]和right[]合并成一个大的有序数组'''
```

```
    # left与right的下标指针
    l_index, r_index = 0, 0
    result = []
    while l_index < len(left) and r_index < len(right):
        if left[l_index] < right[r_index]:
            result.append(left[l_index])
            l_index += 1
        else:
            result.append(right[r_index])
            r_index += 1

    result += left[l_index:]
    result += right[r_index:]
    return result

alist = [54,26,93,17,77,31,44,55,20]
sorted_alist = merge_sort(alist)
print(sorted_alist)
```

9-8-常见排序算法效率比较

常见排序算法效率比较

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	O(1)	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	O(1)	稳定
希尔排序	$O(n \log n)$~ $O(n^2)$	$O(n^{1.3})$	$O(n^2)$	O(1)	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	O(1)	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$~ $O(n)$	不稳定

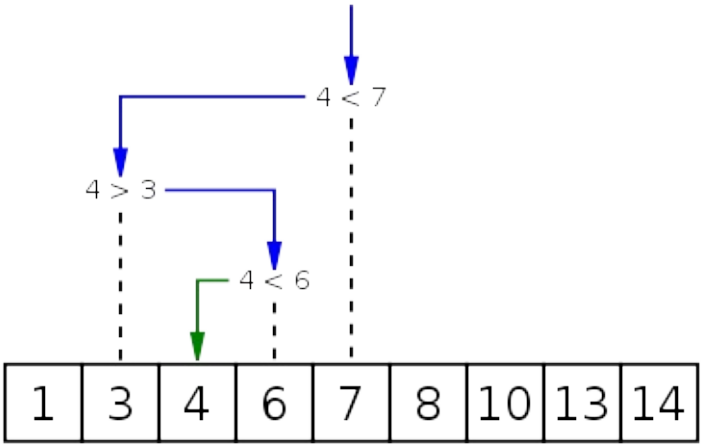
9-9-搜索

搜索

搜索是在一个项目集合中找到一个特定项目的算法过程。搜索通常的答案是真的或假的，因为该项目是否存在。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

二分查找

二分查找又称折半查找，优点是比较次数少，查找速度快，平均性能好；其缺点是要求待查表为有序表，且插入删除困难。因此，折半查找方法适用于不经常变动而查找频繁的有序列表。首先，假设表中元素是按升序排列，将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表，如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。



二分法查找实现

（非递归实现）

```
def binary_search(li, item):
    first = 0
    last = len(li) - 1
```

```
    while first <= last:
        midpoint = (first + last) // 2
        if li[midpoint] == item:
            return True
        elif item < li[midpoint]:
            last = midpoint - 1
        else:
            first = midpoint + 1

    return False


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

（递归实现）

```
def binary_search(li, item):
    if len(li) == 0:
        return False

    else:
        midpoint = len(li) // 2
        if midpoint > 0:
            if li[midpoint] == item: # li[4] == 17 13 != 17
                return True
            else:
                if item < li[midpoint]:   # 17 < 13
                    return binary_search(li[:midpoint], item)
                else:
                    return binary_search(li[midpoint+1:], item)
        else:
            return False

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

问题

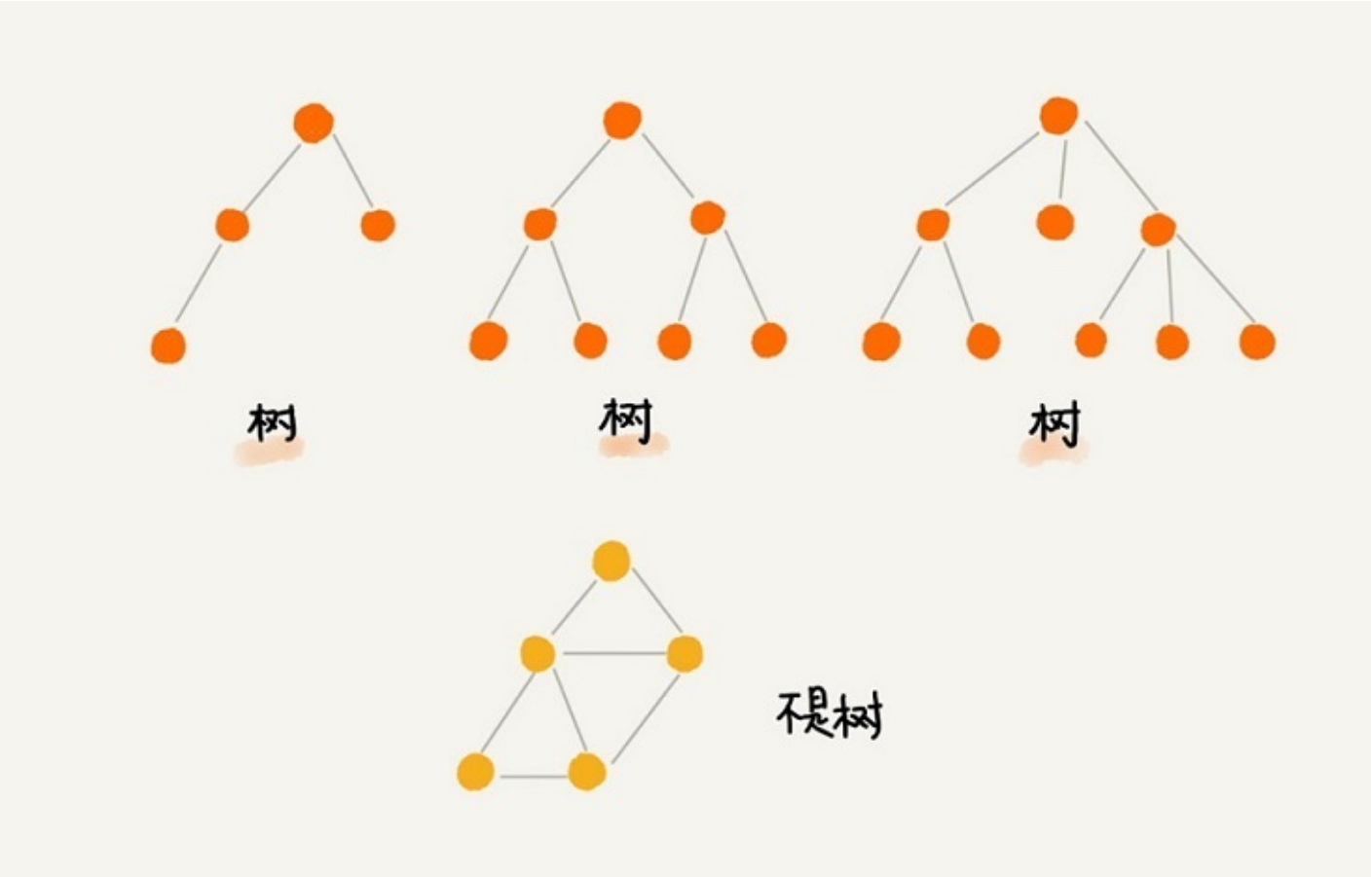
假设我们有 1000 万个整数数据，每个数据占 8 个字节，如何设计数据结构和算法，快速判断某个整数是否出现在这 1000 万数据中？ 我们希望这个功能不要占用太多的内存空间，最多不要超过 100MB，你会怎么做呢？

10-树

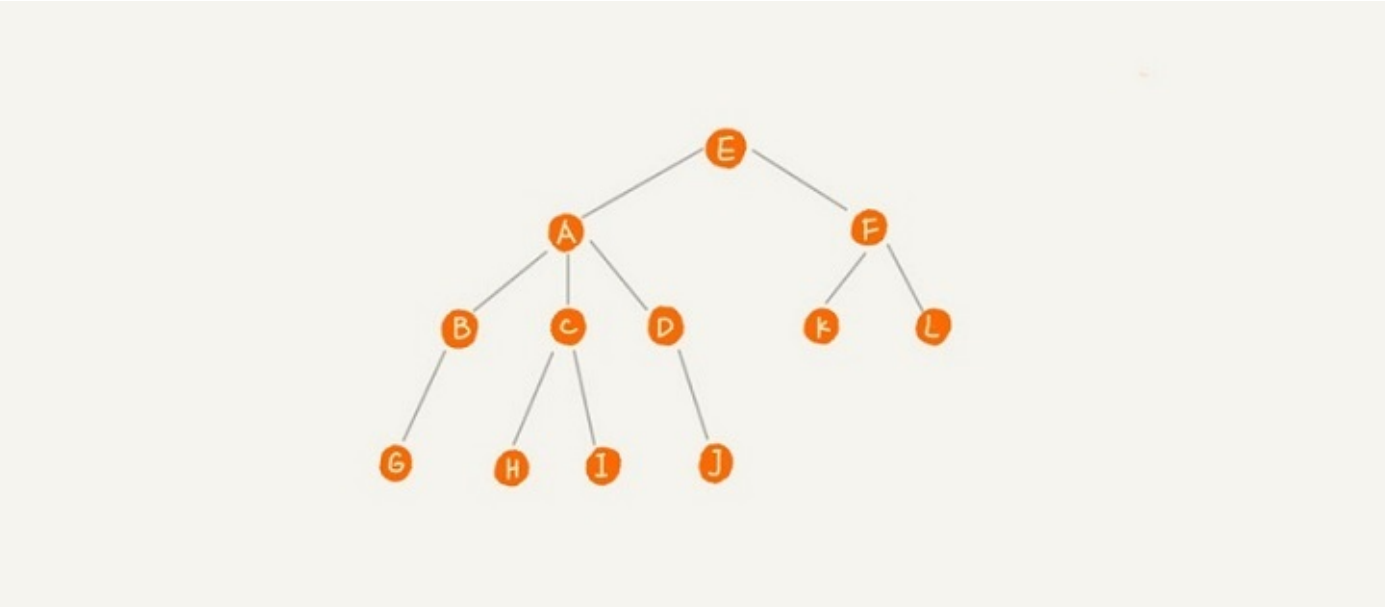
树的概念

树（英语：tree）是一种抽象数据类型（ADT）或是实作这种抽象数据类型的数据结构，用来模拟具有树状结构性质的数据集合。它是由n（n>=1）个有限节点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

- 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；
- 除了根节点外，每个子节点可以分为多个不相交的子树；



你有没有发现，“树”这种数据结构真的很像我们现实生活中的“树”，这里面每个元素我们叫作“节点”；用来连线相邻节点之间的关系，我们叫作“父子关系”。

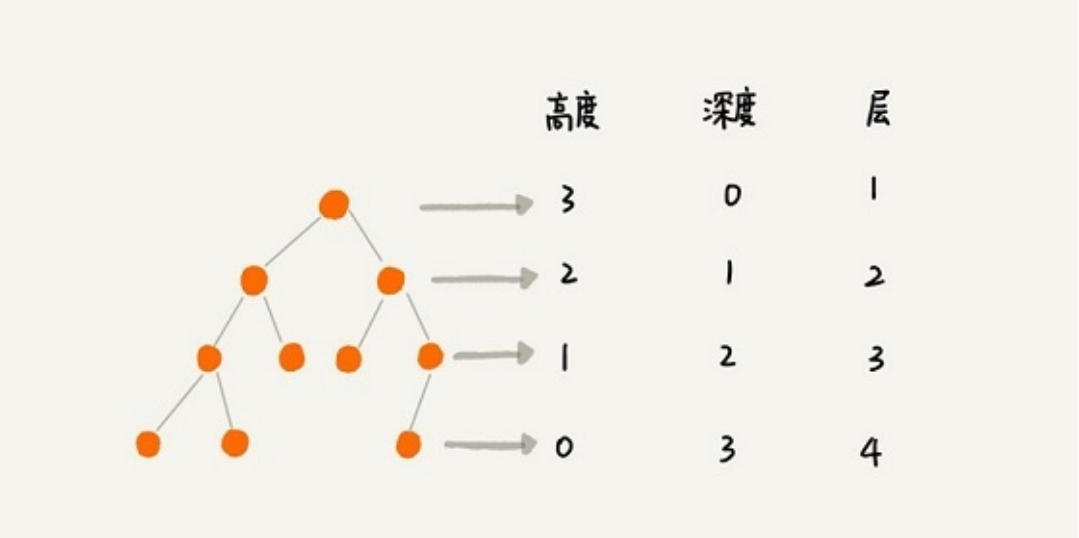


A 节点就是 B 节点的父节点，B 节点是 A 节点的子节点。B、C、D 这三个节点的父节点是同一个节点，所以它们之间互称为兄弟节点。我们把没有父节点的节点叫作根节点，也就是图中的节点 E。我们把没有子节点的节点叫作叶子节点或者叶节点，比如图中的 G、H、I、J、K、L 都是叶子节点。

除此之外，关于“树”，还有三个比较相似的概念：高度（Height）、深度（Depth）、层（Level）。它们的定义是这样的

- 节点的高度:节点到叶子节点的最长路径
- 节点的深度:根节点到这个节点所经历的边的个数
- 节点的层数:节点的深度+1
- 树的高度:根节点的高度

这三个概念的定义比较容易混淆，描述起来也比较空洞。我举个例子说明一下，你一看应该就能明白。

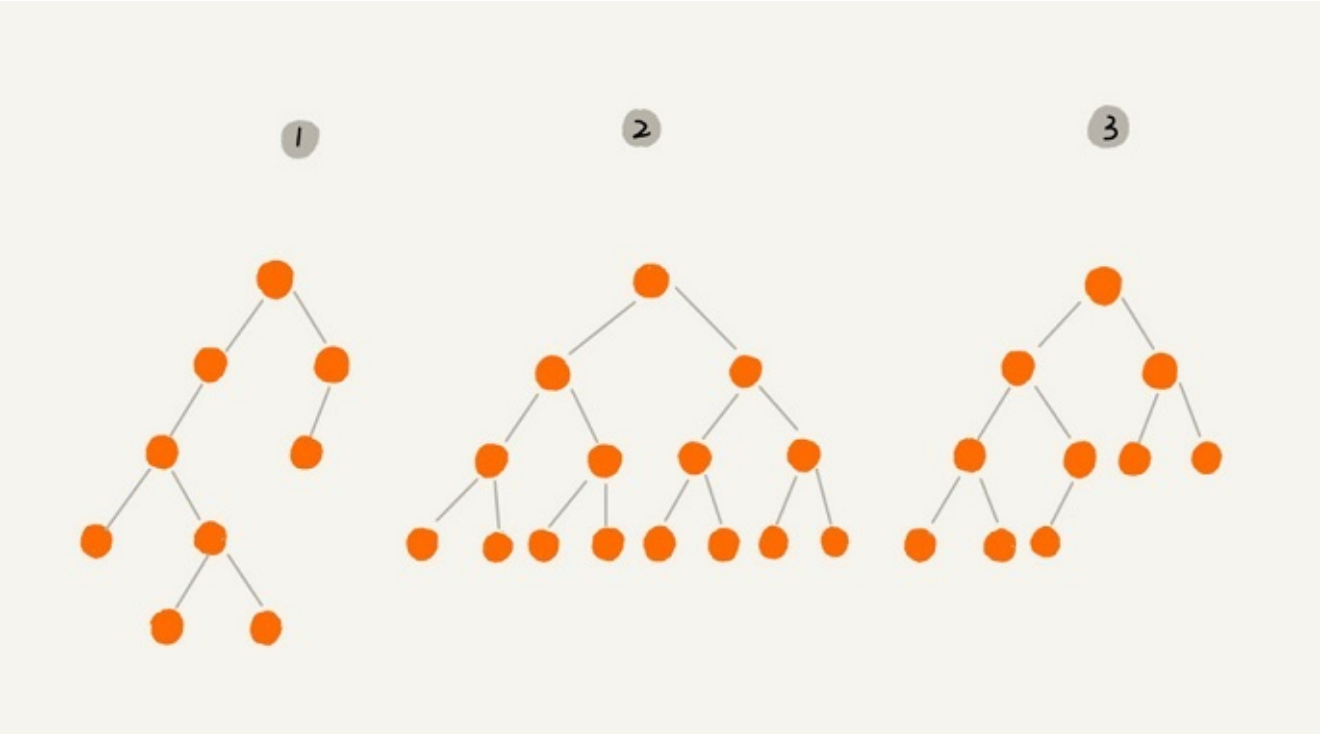


树的种类

- 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的树称为二叉树；
    - 完全二叉树：对于一颗二叉树，假设其深度为d(d>1)。除了第d层外，其它各层的节点数目均已达最大值，且第d层所有节点从左向右连续地紧密排列，这样的二叉树被称为完全二叉树，其中满二叉树的定义是所有叶节点都在最底层的完全二叉树;
    - 平衡二叉树（AVL树）：当且仅当任何节点的两棵子树的高度差不大于1的二叉树；
    - 排序二叉树（二叉查找树（英语：Binary Search Tree），也称二叉搜索树、有序二叉树）；
  - 霍夫曼树（用于信息编码）：带权路径最短的二叉树称为哈夫曼树或最优二叉树；
  - B树：一种对读写操作进行优化的自平衡的二叉查找树，能够保持数据有序，拥有多余两个子树。

二叉树

二叉树，顾名思义，每个节点最多有两个“叉”，也就是两个子节点，分别是左子节点和右子节点。不过，二叉树并不要求每个节点都有两个子节点，有的节点只有左子节点，有的节点只有右子节点。



这个图里面，有两个比较特殊的二叉树，分别是编号 2 和编号 3 这两个。

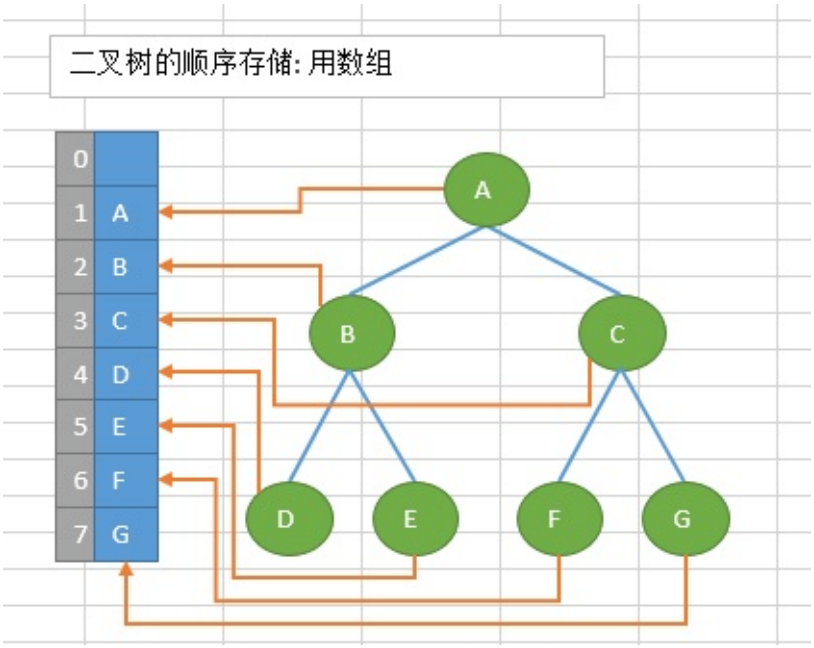
其中，编号 2 的二叉树中，叶子节点全都在最底层，除了叶子节点之外，每个节点都有左右两个子节点，这种二叉树就叫作满二叉树。

编号 3 的二叉树中，叶子节点都在最底下两层，最后一层的叶子节点都靠左排列，并且除了最后一层，其他层的节点个数都要达到最大，这种二叉树叫作完全二叉树。

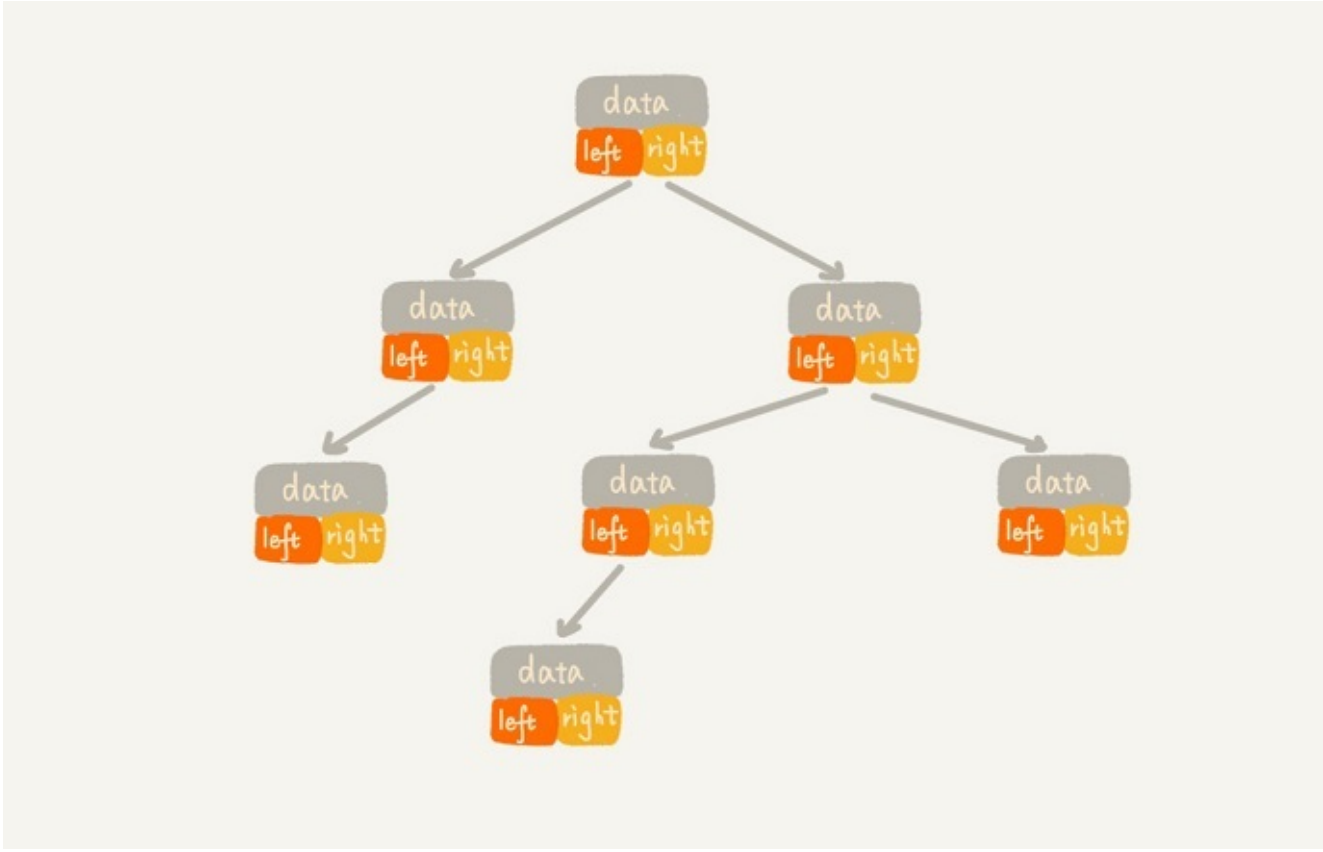
二叉排序树:https://baike.baidu.com/item/二叉排序树/10905079?fr=aladdin

树的存储与表示

顺序存储：将数据结构存储在固定的数组中，然在遍历速度上有一定的优势，但因所占空间比较大，是非主流二叉树。二叉树通常以链式存储。



链式存储：



从图中你应该可以很清楚地看到，每个节点有三个字段，其中一个存储数据，另外两个是指向左右子节点的指针。我们只要拎住根节点，就可以通过左右子节点的指针，把整棵树都串起来。这种存储方式我们比较常用。大部分二叉树代码都是通过这种结构来实现的。

常见的一些树的应用场景

1. xml，html等，那么编写这些东西的解析器的时候，不可避免用到树
2. 路由协议就是使用了树的算法
3. mysql数据库索引

4. 文件系统的目录结构
5. 所以很多经典的AI算法其实都是树搜索，此外机器学习中的decision tree也是树结构

10-1-二叉树的实现与遍历

二叉树的节点表示以及树的创建

通过使用Node类中定义三个属性，分别为item本身的值，还有litem和ritem

```
class Node(object):
    def __init__(self, item, litem=None, ritem=None):
        self.item = item
        self.litem = litem
        self.ritem = ritem
```

树的创建,创建一个树的类，并给一个root根节点，一开始为空，随后添加节点

```
class Tree(object):
    """树类"""

    def __init__(self, root=None):
        self.root = root

    def add(self, elem):
        """为树添加节点"""
        node = Node(elem)
        # 如果树是空的，则对根节点赋值
        if self.root == None:
            self.root = node
        else:
            queue = []
            queue.append(self.root)
            # 对已有的节点进行层次遍历
            while queue:
                # 弹出队列的第一个元素
                cur = queue.pop(0)
                if cur.litem == None:
                    cur.litem = node
```

```
                return
            elif cur.ritem == None:
                cur.ritem = node
                return
            else:
                # 如果左右子树都不为空，加入队列继续判断
                queue.append(cur.litem)
                queue.append(cur.ritem)
```

二叉树的遍历

树的遍历是树的一种重要的运算。所谓遍历是指对树中所有结点的信息的访问，即依次对树中每个结点访问一次且仅访问一次，我们把这种对所有节点的访问称为遍历（traversal）。那么树的两种重要的遍历模式是深度优先遍历和广度优先遍历,深度优先一般用递归，广度优先一般用队列。一般情况下能用递归实现的算法大部分也能用堆栈来实现。

深度优先遍历

对于一颗二叉树，深度优先搜索(Depth First Search)是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。

那么深度遍历有重要的三种方法。这三种方式常被用于访问树的节点，它们之间的不同在于访问每个节点的次序不同。这三种遍历分别叫做先序遍历（preorder），中序遍历（inorder）和后序遍历（postorder）。我们来给出它们的详细定义，然后举例看看它们的应用。

- 先序遍历 在先序遍历中，我们先访问根节点，然后递归使用先序遍历访问左子树，再递归使用先序遍历访问右子树

  根节点->左子树->右子树

```
def preorder(self, node):
    """递归实现先序遍历"""
    if node == None:
        return
    print(node.item, end=" ")
    self.preorder(node.litem)
    self.preorder(node.ritem)
```

- 中序遍历 在中序遍历中，我们递归使用中序遍历访问左子树，然后访问根节点，最后再递归使用中序遍历访问右子树

  左子树->根节点->右子树

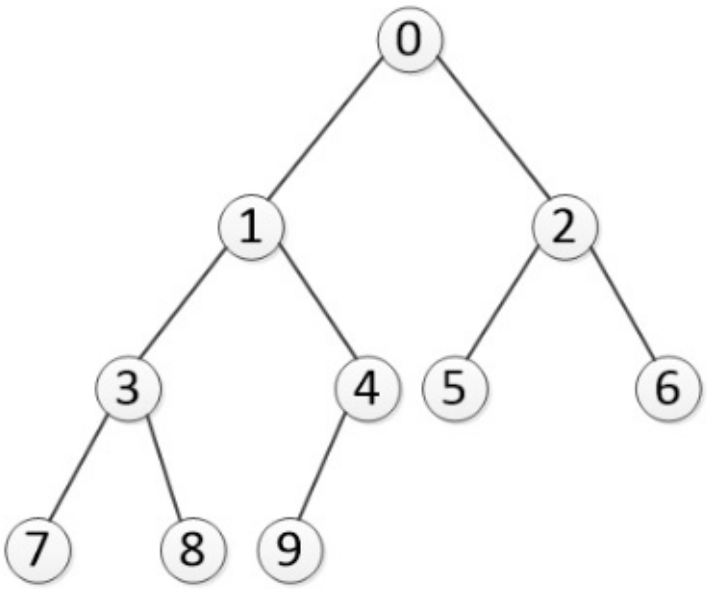
```
def inorder(self, node):
    """递归实中序遍历"""
```

```
    if node == None:
        return
    self.inorder(node.litem)
    print(node.item, end=" ")
    self.inorder(node.ritem)
```

- 后序遍历 在后序遍历中，我们先递归使用后序遍历访问左子树和右子树，最后访问根节点

  左子树->右子树->根节点

```
def postorder(self, node):
    """递归实现后序遍历"""
    if node == None:
        return
    self.postorder(node.litem)
    self.postorder(node.ritem)
    print(node.item, end=" ")
```



层次遍历：0 1 2 3 4 5 6 7 8 9

先序遍历：0 1 3 7 8 4 9 2 5 6

中序遍历：7 3 8 1 9 4 0 5 2 6

后序遍历：7 8 3 9 4 1 5 6 2 0

广度优先遍历(层次遍历)

从树的root开始，从上到下从从左到右遍历整个树的节点

```
def breadth_travel(self):
    """利用队列实现树的层次遍历"""
    if self.root == None:
        return
    queue = []
    queue.append(self.root)
    while queue:
        node = queue.pop(0)
        print(node.item)
```

```
        if node.litem != None:
            queue.append(node.litem)
        if node.ritem != None:
            queue.append(node.ritem)
```

思考：哪两种遍历方式能够唯一的确定一颗树？？？