Python Data Structure and Algorithms

Fan er

目录

序 言

作者介绍

- 范儿
- f18811010711@gmail.com
- 我的博客 @范儿
- 某大学大二学生，学校课程学习python。
- 必须强调，女朋友比代码重要，以后一定要多陪陪女友。

本书介绍

- 新手入门，高手退避。
- 熟悉python项目。
- 目前基于python 2，作者正在升级到3
- 本人能力有限。如有错误，欢迎读者反馈。
- 本书开源，读者可以随意修改、发布，但必须保留本页。

本书目标

- 掌握python。
- 基础算法知识。
- 望读者手敲每一个代码，并思考其中逻辑。
- 后续有升级版，敬请关注。

引 子

- 算法
- 算法效率的衡量
- python内置类型性能分析
- 数据结构

在这一章里，我将要介绍算法、算法效率的衡量、python内置类型的性能分析以及数据结构的一些概念，为以后的学习奠定基础。

第一节 第一个程序

在开始之前，我用一个例子引入。

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**?**

对于这个问题你有什么自己的想法吗？你可以自己试试，我将在后面一节里给出答案。

第二节 第一次尝试

你有写你自己的吗？我写了下面的代码。

```
import time

start_time = time.time()

# 注意是三重循环
for a in range(0, 1001):
    for b in range(0, 1001):
        for c in range(0, 1001):
            if a**2 + b**2 == c**2 and a+b+c == 1000:
                print("a, b, c: %d, %d, %d" % (a, b, c))

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

就像你看到的，我使用time这个模块记录了整个程序运行的时间，便于后面的分析。程序运行结果：

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

注意运行时间，居然有214.58s!显然，这个结果是我们很难接受的，那么这个程序还可以优化吗？如果可以，怎么优化，这就是我们要思考的问题了。

第三节 算法的提出

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

第四节 第二次尝试

既然现在我们知道有算法这么一说，就尝试用算法实现，下面是举例：

```
import time

start_time = time.time()

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

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

这次和上次一样，依旧记录时间，这样的话就可以对比了，明显可以看出的是，我们少了一次循环，这会给程序带来什么影响呢？让我们看看：

```
a, b, c: 0, 500, 500
a, b, c: 200, 375, 425
a, b, c: 375, 200, 425
a, b, c: 500, 0, 500
elapsed: 0.182897
complete!
```

注意这次时间，只用了0.18s，几乎比第一次尝试缩短了百倍不止，读到这里，你应该感受到算法的魅力了。当然这里面涉及到算法效率的衡量，我将会在下面写到。

第五节 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？

假设我们将第二次尝试的算法程序运行在一台配置古老性能低下的计算机中，情况会如何？很可能运行的时间并不会比在我们的电脑中运行算法一的214.583347秒快多少。

单纯依靠运行的时间来比较算法的优劣并不一定是客观准确的！

程序的运行离不开计算机环境（包括硬件和操作系统），这些客观原因会影响程序运行的速度并反应在程序的执行时间上。那么如何才能客观的评判一个算法的优劣呢？

时间复杂度与“大O记法”

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

对于算法的时间效率，我们可以用“大O记法”来表示。

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

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

如何理解“大O记法”

对于算法进行特别具体的细致分析虽然很好，但在实践中的实际价值有限。对于算法的时间性质和空间性质，最重要的是其数量级和趋势，这些是分析算法效率的主要部分。而计量算法基本操作数量的规模函数中那些常量因子可以忽略不计。例如，可以认为3n^2和100n^2属于同一个量级，如果两个算法处理同样规模实例的代价分别为这两个函数，就认为它们的效率“差不多”，都为n^2级。

最坏时间复杂度

分析算法时，存在几种可能的考虑：

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

对于最优时间复杂度，其价值不大，因为它没有提供什么有用信息，其反映的只是最乐观最理想的情况，没有参考价值。

对于最坏时间复杂度，提供了一种保证，表明算法在此种程度的基本操作中一定能完成工作。

对于平均时间复杂度，是对算法的一个全面评价，因此它完整全面的反映了这个算法的性质。但另一方面，这种衡量并没有保证，不是每个计算都能在这个基本操作内完成。而且，对于平均情况的计算，也会因为应用算法的实例分布可能并不均匀而难以计算。

因此，我们主要关注算法的最坏情况，亦即最坏时间复杂度。

时间复杂度的几条基本计算规则

- 基本操作，即只有常数项，认为其时间复杂度为O(1)

- 顺序结构，时间复杂度按加法进行计算
- 循环结构，时间复杂度按乘法进行计算
- 分支结构，时间复杂度取最大值
- 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
- 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

第六节 算法分析

第一次尝试的算法核心部分

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

时间复杂度:

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

第二次尝试的算法核心部分

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

时间复杂度:

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

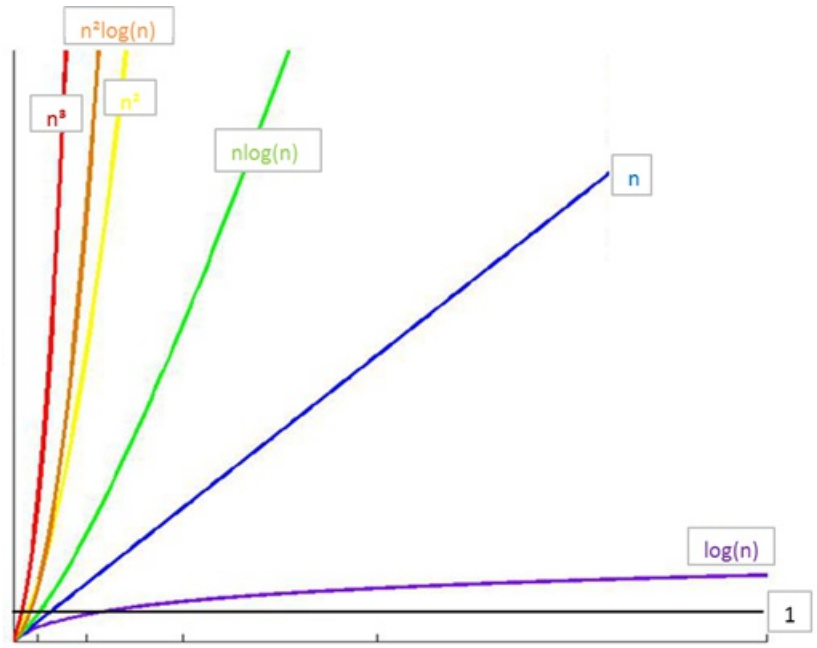
由此可见，我们尝试的第二种算法要比第一种算法的时间复杂度好多的。

第七节 常见时间复杂度

执行次数函数举例	阶	非正式术语
12	O(1)	常数阶
2n+3	O(n)	线性阶
3n^2+2n+1	O(n^2)	平方阶
5log2n+20	O(logn)	对数阶
2n+3nlog2n+19	O(nlogn)	nlogn阶
6n^3+2n^2+3n+4	O(n^3)	立方阶
2^n	O(2^n)	指数阶

注意，经常将**log2n**（以**2**为底的对数）简写成**logn**

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(n^n)

> 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(n²+ n + 1)
>
> O(3n³+1)

第一章 顺序表

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

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

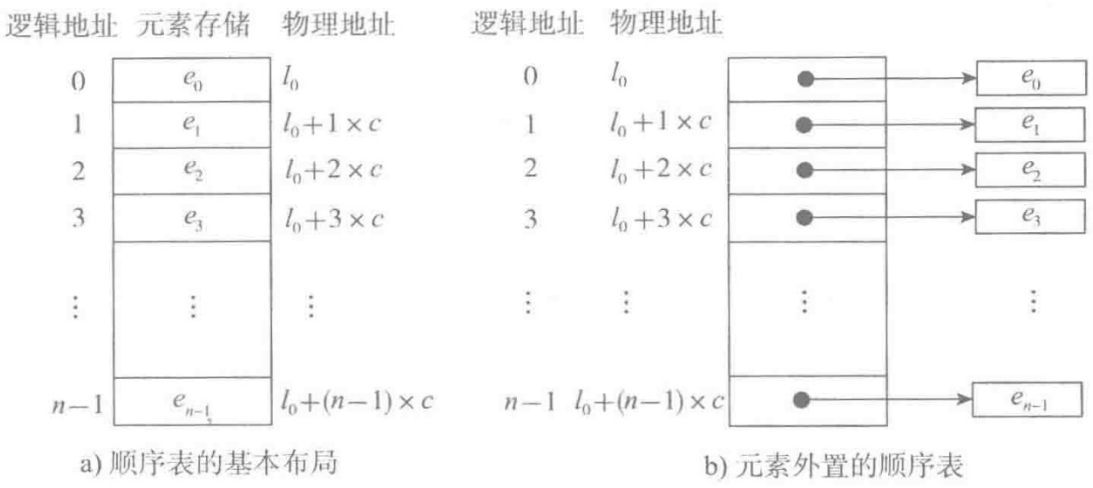
这样的一组序列元素的组织形式，我们可以将其抽象为线性表。一个线性表是某类元素的一个集合，还记录着元素之间的一种顺序关系。线性表是最基本的数据结构之一，在实际程序中应用非常广泛，它还经常被用作更复杂的数据结构的实现基础。

根据线性表的实际存储方式，分为两种实现模型：

顺序表：将元素顺序地存放在一块连续的存储区里，元素间的顺序关系由它们的存储顺序自然表示。

链表将元素存放在通过链接构造起来的一系列存储块中。

第一节 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

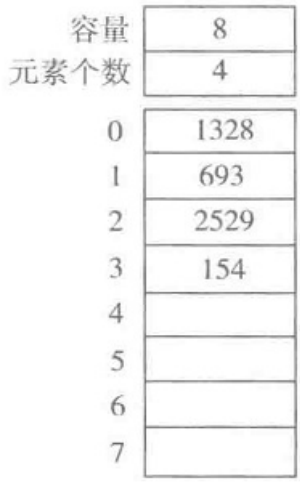
故，访问指定元素时无需从头遍历，通过计算便可获得对应地址，其时间复杂度为O(1)。

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。

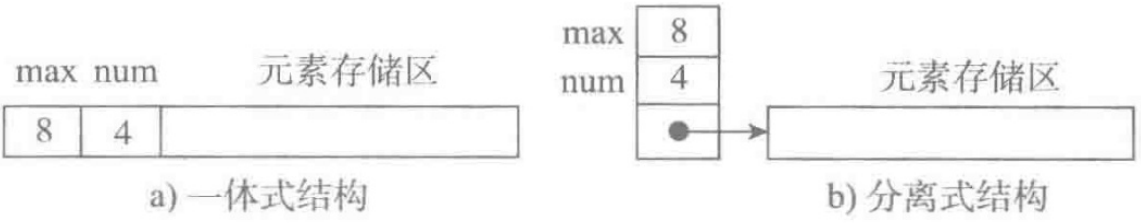
第二节 顺序表的结构与实现

顺序表的结构与实现



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。

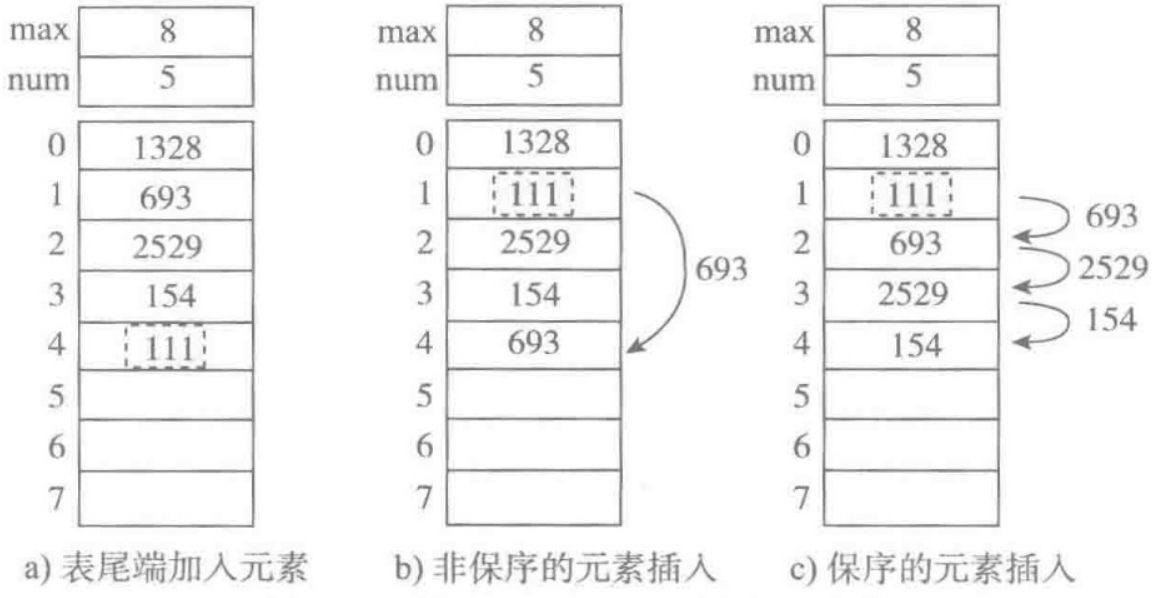
- 每次扩充容量加倍，如每次扩充增加一倍存储空间。

> 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。

第三节 顺序表的操作

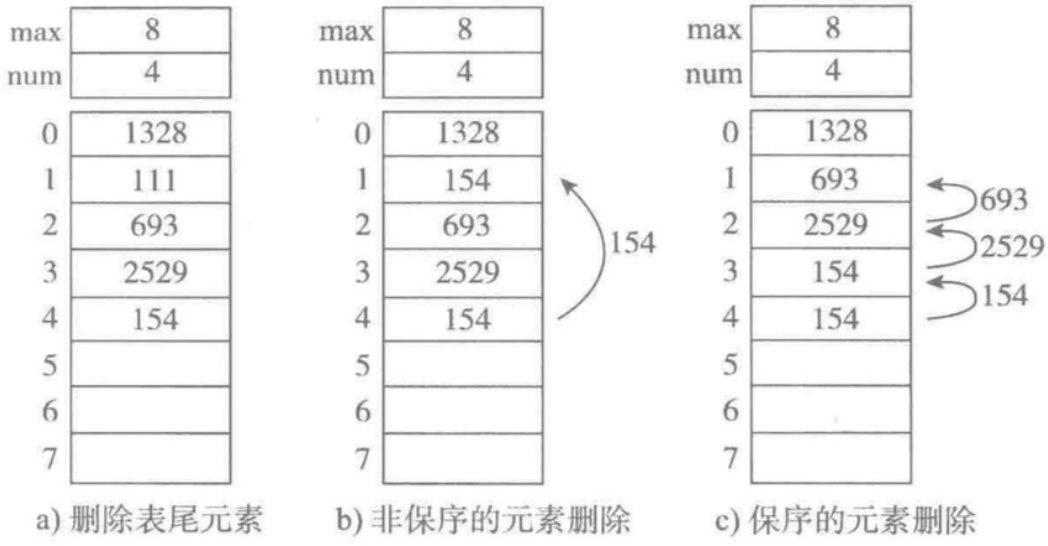
增加元素

如图所示，为顺序表增加新元素111的三种方式:



- 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)

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)

第四节 **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。

第二章 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。

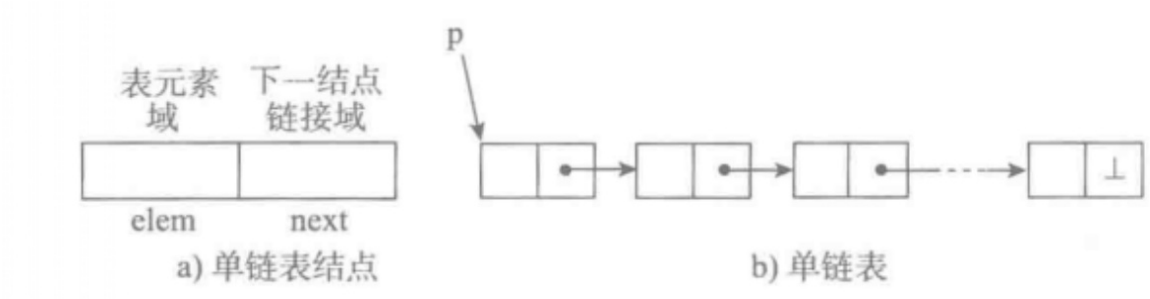
链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。

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）。

第一节 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

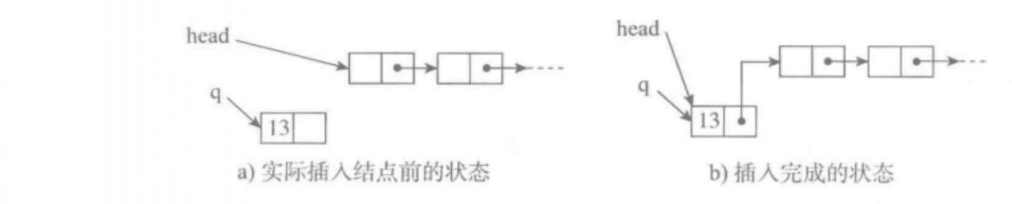
- is_empty() 链表是否为空
- length() 链表长度
- travel() 遍历整个链表
- add(item) 链表头部添加元素
- append(item) 链表尾部添加元素
- insert(pos, item) 指定位置添加元素
- remove(item) 删除节点
- search(item) 查找节点是否存在

单链表的实现

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

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

    def length(self):
        """链表长度"""
        # cur初始时指向头节点
        cur = self._head
```

```
        count = 0
        # 尾节点指向None，当未到达尾部时
        while cur != None:
            count += 1
            # 将cur后移一个节点
            cur = cur.next
        return count

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

头部添加元素

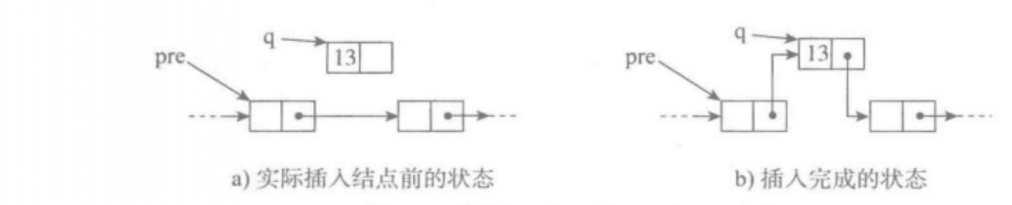



```
    def add(self, item):
        """头部添加元素"""
        # 先创建一个保存item值的节点
        node = SingleNode(item)
        # 将新节点的链接域next指向头节点，即_head指向的位置
        node.next = self._head
        # 将链表的头_head指向新节点
        self._head = node
```

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
```

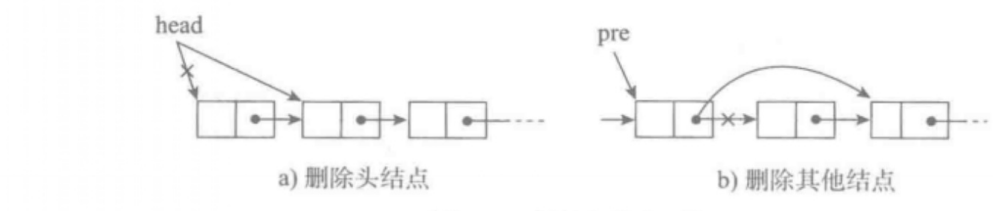
指定位置添加元素




```
    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
```

测试

```
if __name__ == "__main__":
    ll = SingleLinkList()
    ll.add(1)
    ll.add(2)
    ll.append(3)
    ll.insert(2, 4)
    print "length:",ll.length()
    ll.travel()
    print ll.search(3)
    print ll.search(5)
    ll.remove(1)
    print "length:",ll.length()
    ll.travel()
```

链表与顺序表的对比

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

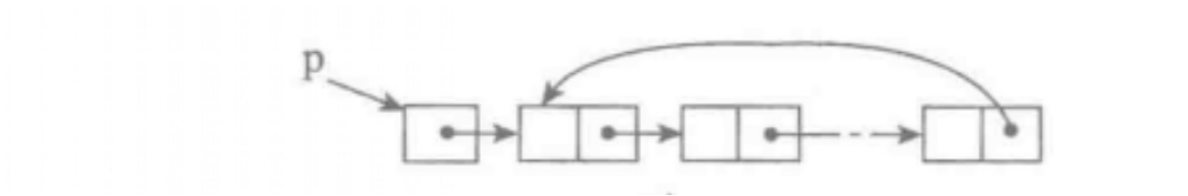
链表与顺序表的各种操作复杂度如下所示：

操作	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。

第二节 单向循环链表

单链表的一个变形是单向循环链表，链表中最后一个节点的next域不再为None，而是指向链表的头节点。



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

代码实现

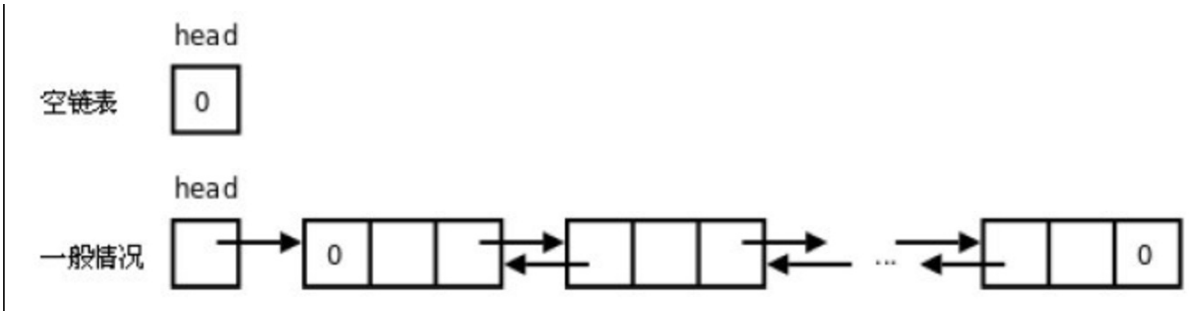
```
class Node(object):
    """节点"""
    def __init__(self, item):
        self.item = item
        self.next = None


class SinCycLinkedlist(object):
    """单向循环链表"""
    def __init__(self):
        self._head = None

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

    def length(self):
        """返回链表的长度"""
        # 如果链表为空，返回长度0
        if self.is_empty():
            return 0
        count = 1
        cur = self._head
        while cur.next != self._head:
            count += 1
            cur = cur.next
        return count

    def travel(self):
        """遍历链表"""
        if self.is_empty():
            return
        cur = self._head
        print cur.item,
        while cur.next != self._head:
            cur = cur.next
            print cur.item,
        print ""
```

```
def add(self, item):
    """头部添加节点"""
    node = Node(item)
    if self.is_empty():
        self._head = node
        node.next = self._head
    else:
        #添加的节点指向_head
        node.next = self._head
        # 移到链表尾部，将尾部节点的next指向node
        cur = self._head
        while cur.next != self._head:
            cur = cur.next
        cur.next = node
        #_head指向添加node的
        self._head = node

def append(self, item):
    """尾部添加节点"""
    node = Node(item)
    if self.is_empty():
        self._head = node
        node.next = self._head
    else:
        # 移到链表尾部
        cur = self._head
        while cur.next != self._head:
            cur = cur.next
        # 将尾节点指向node
        cur.next = node
        # 将node指向头节点_head
        node.next = self._head

def insert(self, pos, item):
    """在指定位置添加节点"""
    if pos <= 0:
        self.add(item)
    elif pos > (self.length()-1):
        self.append(item)
    else:
        node = Node(item)
        cur = self._head
        count = 0
        # 移动到指定位置的前一个位置
        while count < (pos-1):
            count += 1
            cur = cur.next
        node.next = cur.next
        cur.next = node

def remove(self, item):
    """删除一个节点"""
    # 若链表为空，则直接返回
    if self.is_empty():
        return
    # 将cur指向头节点
    cur = self._head
    pre = None
    # 若头节点的元素就是要查找的元素item
    if cur.item == item:
        # 如果链表不止一个节点
        if cur.next != self._head:
            # 先找到尾节点，将尾节点的next指向第二个节点
            while cur.next != self._head:
                cur = cur.next
            # cur指向了尾节点
            cur.next = self._head.next
            self._head = self._head.next
        else:
            # 链表只有一个节点
            self._head = None
    else:
        pre = self._head
        # 第一个节点不是要删除的
        while cur.next != self._head:
            # 找到了要删除的元素
            if cur.item == item:
                # 删除
                pre.next = cur.next
                return
            else:
                pre = cur
```

```
                    cur = cur.next
            # cur 指向尾节点
            if cur.item == item:
                # 尾部删除
                pre.next = cur.next

    def search(self, item):
        """查找节点是否存在"""
        if self.is_empty():
            return False
        cur = self._head
        if cur.item == item:
            return True
        while cur.next != self._head:
            cur = cur.next
            if cur.item == item:
                return True
        return False

if __name__ == "__main__":
    ll = SinCycLinkedlist()
    ll.add(1)
    ll.add(2)
    ll.append(3)
    ll.insert(2, 4)
    ll.insert(4, 5)
    ll.insert(0, 6)
    print "length:",ll.length()
    ll.travel()
    print ll.search(3)
    print ll.search(7)
    ll.remove(1)
    print "length:",ll.length()
    ll.travel()
```

第三节 双向链表

一种更复杂的链表是“双向链表”或“双面链表”。每个节点有两个链接：一个指向前一个节点，当此节点为第一个节点时，指向空值；而另一个指向下一个节点，当此节点为最后一个节点时，指向空值。



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

代码实现

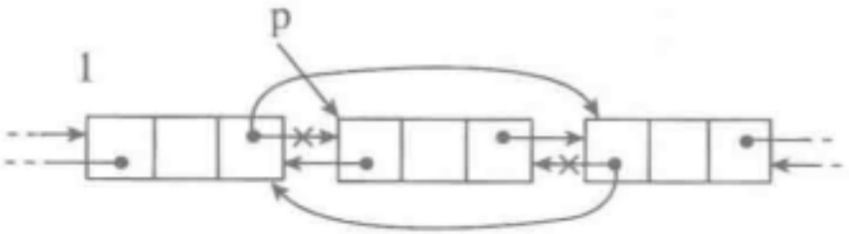
```
class Node(object):
    """双向链表节点"""
    def __init__(self, item):
        self.item = item
        self.next = None
        self.prev = None


class DLinkList(object):
    """双向链表"""
    def __init__(self):
        self._head = None

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

    def length(self):
        """返回链表的长度"""
        cur = self._head
        count = 0
        while cur != None:
            count += 1
            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 ""
```

```
    def add(self, item):
        """头部插入元素"""
        node = Node(item)
        if self.is_empty():
            # 如果是空链表，将_head指向node
            self._head = node
        else:
            # 将node的next指向_head的头节点
            node.next = self._head
            # 将_head的头节点的prev指向node
            self._head.prev = node
            # 将_head 指向node
            self._head = node

    def append(self, item):
        """尾部插入元素"""
        node = Node(item)
        if self.is_empty():
            # 如果是空链表，将_head指向node
            self._head = node
        else:
            # 移动到链表尾部
            cur = self._head
            while cur.next != None:
                cur = cur.next
            # 将尾节点cur的next指向node
            cur.next = node
            # 将node的prev指向cur
            node.prev = cur



    def search(self, item):
        """查找元素是否存在"""
        cur = self._head
        while cur != None:
            if cur.item == item:
                return True
            cur = cur.next
        return False
```

指定位置插入节点

```
    def insert(self, pos, item):
        """在指定位置添加节点"""
        if pos <= 0:
            self.add(item)
        elif pos > (self.length()-1):
            self.append(item)
        else:
            node = Node(item)
            cur = self._head
            count = 0
            # 移动到指定位置的前一个位置
            while count < (pos-1):
                count += 1
                cur = cur.next
            # 将node的prev指向cur
            node.prev = cur
            # 将node的next指向cur的下一个节点
            node.next = cur.next
            # 将cur的下一个节点的prev指向node
            cur.next.prev = node
            # 将cur的next指向node
            cur.next = node
```

删除元素




```
def remove(self, item):
        """删除元素"""
        if self.is_empty():
            return
        else:
            cur = self._head
            if cur.item == item:
                # 如果首节点的元素即是要删除的元素
                if cur.next == None:
                    # 如果链表只有这一个节点
                    self._head = None
                else:
                    # 将第二个节点的prev设置为None
                    cur.next.prev = None
                    # 将_head指向第二个节点
                    self._head = cur.next
                return
            while cur != None:
                if cur.item == item:
                    # 将cur的前一个节点的next指向cur的后一个节点
                    cur.prev.next = cur.next
                    # 将cur的后一个节点的prev指向cur的前一个节点
                    cur.next.prev = cur.prev
                    break
                cur = cur.next
```

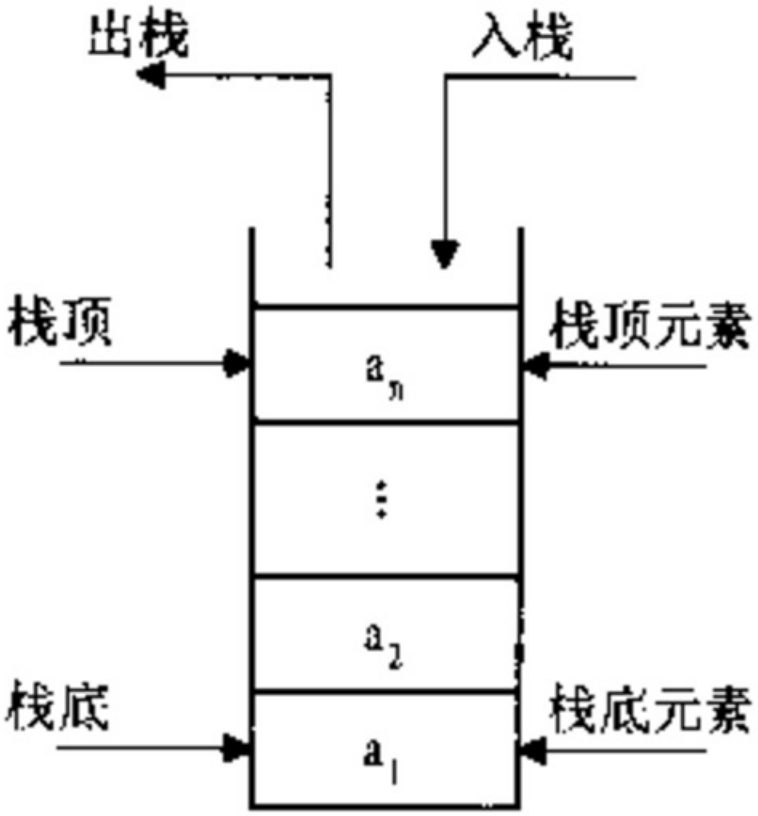
测试

```
if __name__ == "__main__":
    ll = DLinkList()
    ll.add(1)
    ll.add(2)
    ll.append(3)
    ll.insert(2, 4)
    ll.insert(4, 5)
    ll.insert(0, 6)
    print "length:",ll.length()
    ll.travel()
    print ll.search(3)
    print ll.search(4)
    ll.remove(1)
    print "length:",ll.length()
    ll.travel()
```

第三章 栈

栈（stack），有些地方称为堆栈，是一种容器，可存入数据元素、访问元素、删除元素，它的特点在于只能允许在容器的一端（称为栈顶端指标，英语：top）进行加入数据（英语：push）和输出数据（英语：pop）的运算。没有了位置概念，保证任何时候可以访问、删除的元素都是此前最后存入的那个元素，确定了一种默认的访问顺序。

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。



栈结构实现

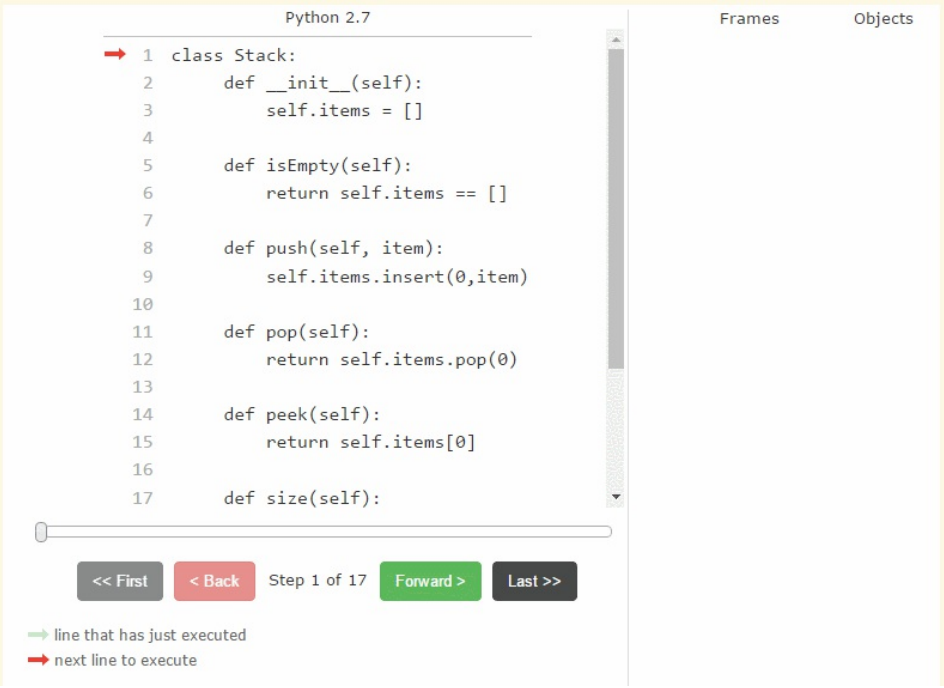
栈可以用顺序表实现，也可以用链表实现。

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

```
class Stack(object):
    """栈"""
    def __init__(self):
         self.items = []

    def is_empty(self):
        """判断是否为空"""
        return self.items == []

    def push(self, item):
        """加入元素"""
        self.items.append(item)

    def pop(self):
        """弹出元素"""
        return self.items.pop()

    def peek(self):
        """返回栈顶元素"""
        return self.items[len(self.items)-1]

    def size(self):
        """返回栈的大小"""
        return len(self.items)

if __name__ == "__main__":
    stack = Stack()
    stack.push("hello")
    stack.push("world")
    stack.push("itcast")
    print stack.size()
    print stack.peek()
    print stack.pop()
    print stack.pop()
    print stack.pop()
```

执行过程如下：



第四章 队列

队列（queue）是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。

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。

第一节 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

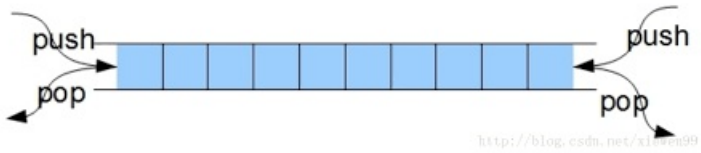
- Queue() 创建一个空的队列
- enqueue(item) 往队列中添加一个item元素
- dequeue() 从队列头部删除一个元素
- is_empty() 判断一个队列是否为空
- size() 返回队列的大小

```
class Queue(object):
    """队列"""
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        """进队列"""
        self.items.insert(0,item)

    def dequeue(self):
        """出队列"""
        return self.items.pop()

    def size(self):
        """返回大小"""
        return len(self.items)

if __name__ == "__main__":
    q = Queue()
    q.enqueue("hello")
    q.enqueue("world")
    q.enqueue("itcast")
    print q.size()
    print q.dequeue()
    print q.dequeue()
    print q.dequeue()
```

第二节 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

实现

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

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

    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()

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


if __name__ == "__main__":
    deque = Deque()
    deque.add_front(1)
    deque.add_front(2)
    deque.add_rear(3)
    deque.add_rear(4)
    print deque.size()
    print deque.remove_front()
    print deque.remove_front()
```

```
print deque.remove_rear()
print deque.remove_rear()
```

第五章 排序与搜索

排序与搜索

排序算法（英语：Sorting algorithm）是一种能将一串数据依照特定顺序进行排列的一种算法。

排序算法的稳定性

稳定性：稳定排序算法会让原本有相等键值的纪录维持相对次序。也就是如果一个排序算法是稳定的，当有两个相等键值的纪录R和S，且在原本的列表中R出现在S之前，在排序过的列表中R也将会是在S之前。

当相等的元素是无法分辨的，比如像是整数，稳定性并不是一个问题。然而，假设以下的数对将要以他们的第一个数字来排序。

```
(4, 1) (3, 1) (3, 7)（5, 6）
```

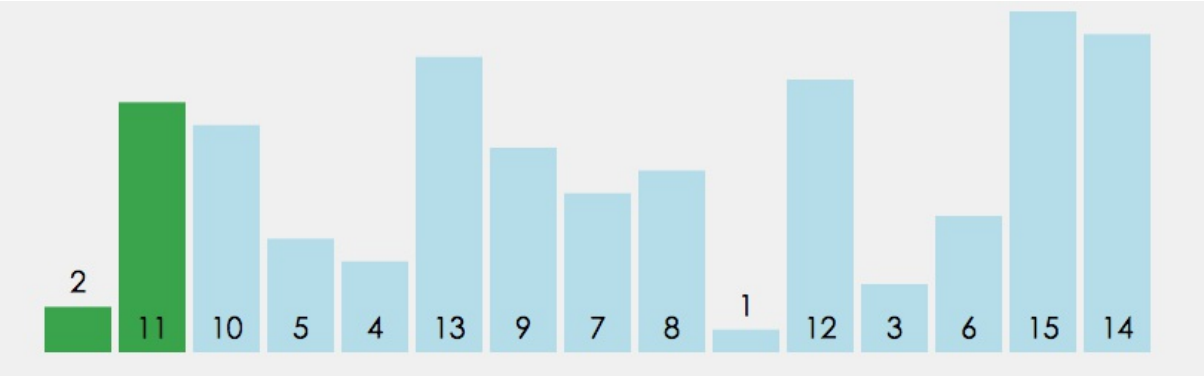
在这个状况下，有可能产生两种不同的结果，一个是让相等键值的纪录维持相对的次序，而另外一个则没有：

```
(3, 1) (3, 7) (4, 1) (5, 6) （维持次序）
(3, 7) (3, 1) (4, 1) (5, 6) （次序被改变）
```

不稳定排序算法可能会在相等的键值中改变纪录的相对次序，但是稳定排序算法从来不会如此。不稳定排序算法可以被特别地实现为稳定。作这件事情的一个方式是人工扩充键值的比较，如此在其他方面相同键值的两个对象间之比较，（比如上面的比较中加入第二个标准：第二个键值的大小）就会被决定使用在原先数据次序中的条目，当作一个同分决赛。然而，要记住这种次序通常牵涉到额外的空间负担。

第一节 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

我们需要进行n-1次冒泡过程，每次对应的比较次数如下图所示：

Pass	Comparisons
1	$n-1$
2	$n-2$
3	$n-3$
...	...
$n-1$	1

```
def bubble_sort(alist):
    for j in range(len(alist)-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 alist[i] > alist[i+1]:
                alist[i], alist[i+1] = alist[i+1], alist[i]

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

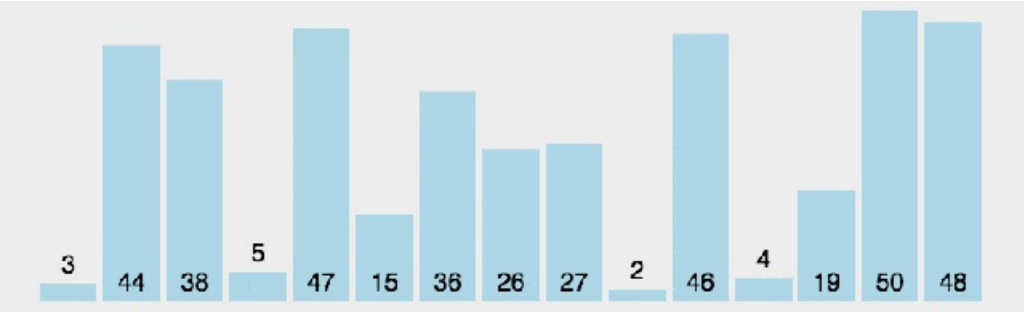
时间复杂度

- 最优时间复杂度：O(n)（表示遍历一次发现没有任何可以交换的元素，排序结束。）
- 最坏时间复杂度：O(n^2)
- 稳定性：稳定

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。



选择排序实现

```
def selection_sort(alist):
    n = len(alist)
    # 需要进行n-1次选择操作
    for i in range(n-1):
        # 记录最小位置
        min_index = i
        # 从i+1位置到末尾选择出最小数据
        for j in range(i+1, n):
            if alist[j] < alist[min_index]:
                min_index = j
        # 如果选择出的数据不在正确位置，进行交换
        if min_index != i:
            alist[i], alist[min_index] = alist[min_index], alist[i]

alist = [54,226,93,17,77,31,44,55,20]
selection_sort(alist)
print(alist)
```

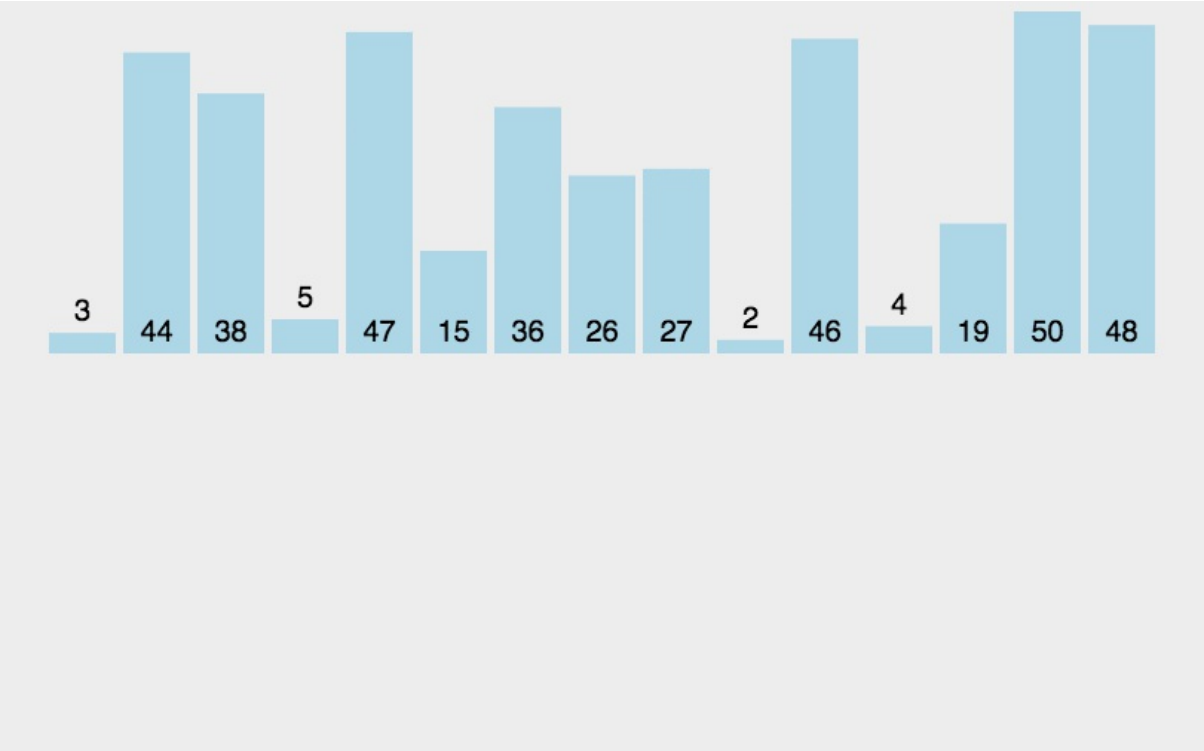
时间复杂度

- 最优时间复杂度：O(n^2)
- 最坏时间复杂度：O(n^2)
- 稳定性：不稳定（考虑升序每次选择最大的情况）

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。



插入排序实现

```
def insert_sort(alist):
    # 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(alist)):
        # 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 alist[j] < alist[j-1]:
                alist[j], alist[j-1] = alist[j-1], alist[j]

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

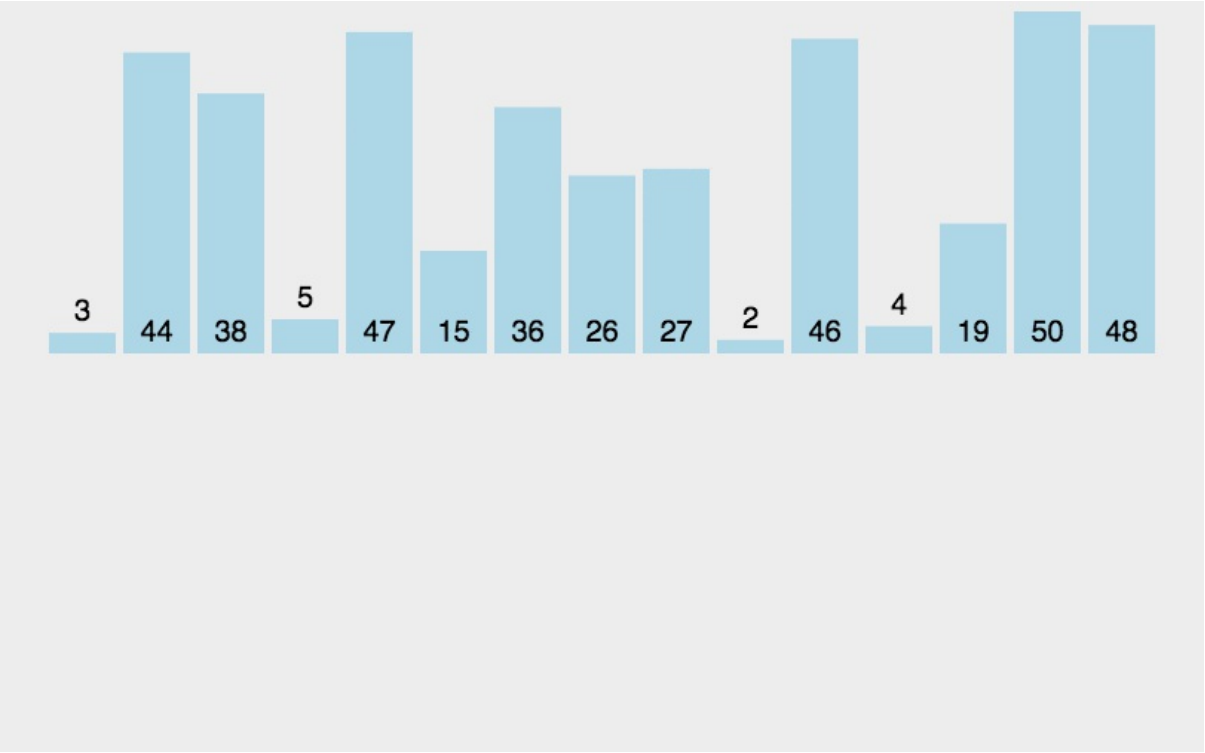
时间复杂度

- 最优时间复杂度：O(n)（升序排列，序列已经处于升序状态）
- 最坏时间复杂度：O(n^2)
- 稳定性：稳定

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。



快速排序运作

- 从数列中挑出一个元素，称为"基准"（pivot），
- 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。
- 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)
```

时间复杂度

- 最优时间复杂度：O(nlogn)
- 最坏时间复杂度：O(n2)
- 稳定性：不稳定

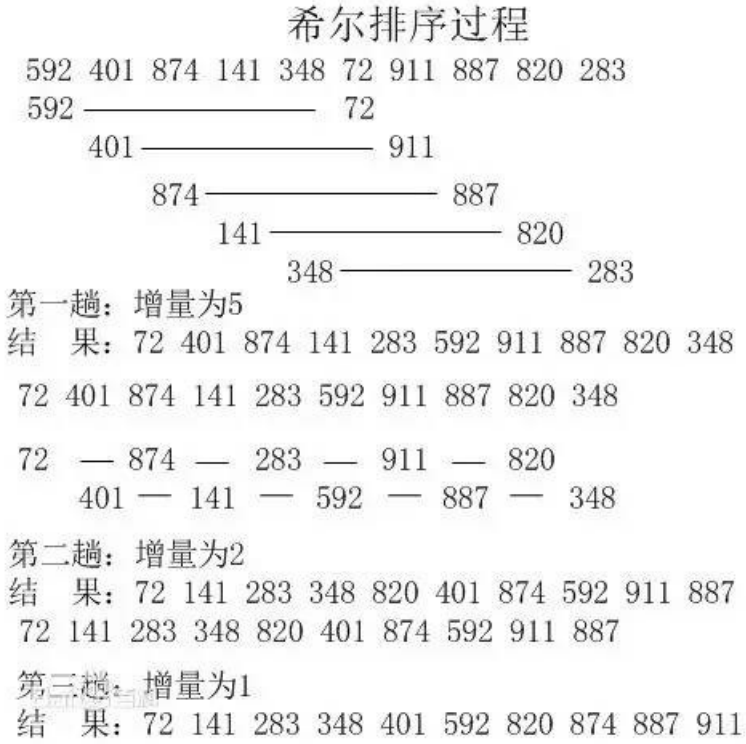
从一开始快速排序平均需要花费O(n log n)时间的描述并不明显。但是不难观察到的是分区运算，数组的元素都会在每次循环中走访过一次，使用O(n)的时间。在使用结合（concatenation）的版本中，这项运算也是O(n)。

在最好的情况，每次我们运行一次分区，我们会把一个数列分为两个几近相等的片段。这个意思就是每次递归调用处理一半大小的数列。因此，在到达大小为一的数列前，我们只要作log n次嵌套的调用。这个意思就是调用树的深度是O(log n)。但是在同一层次结构的两个程序调用中，不会处理到原来数列的相同部分；因此，程序调用的每一层次结构总共全部仅需要O(n)的时间（每个调用有某些共同的额外耗费，但是因为在每一层次结构仅仅只有O(n)个调用，这些被归纳在O(n)系数中）。结果是这个算法仅需使用O(n log n)时间。

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
```

最后以1步长进行排序（此时就是简单的插入排序了）。

希尔排序的实现

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

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

时间复杂度

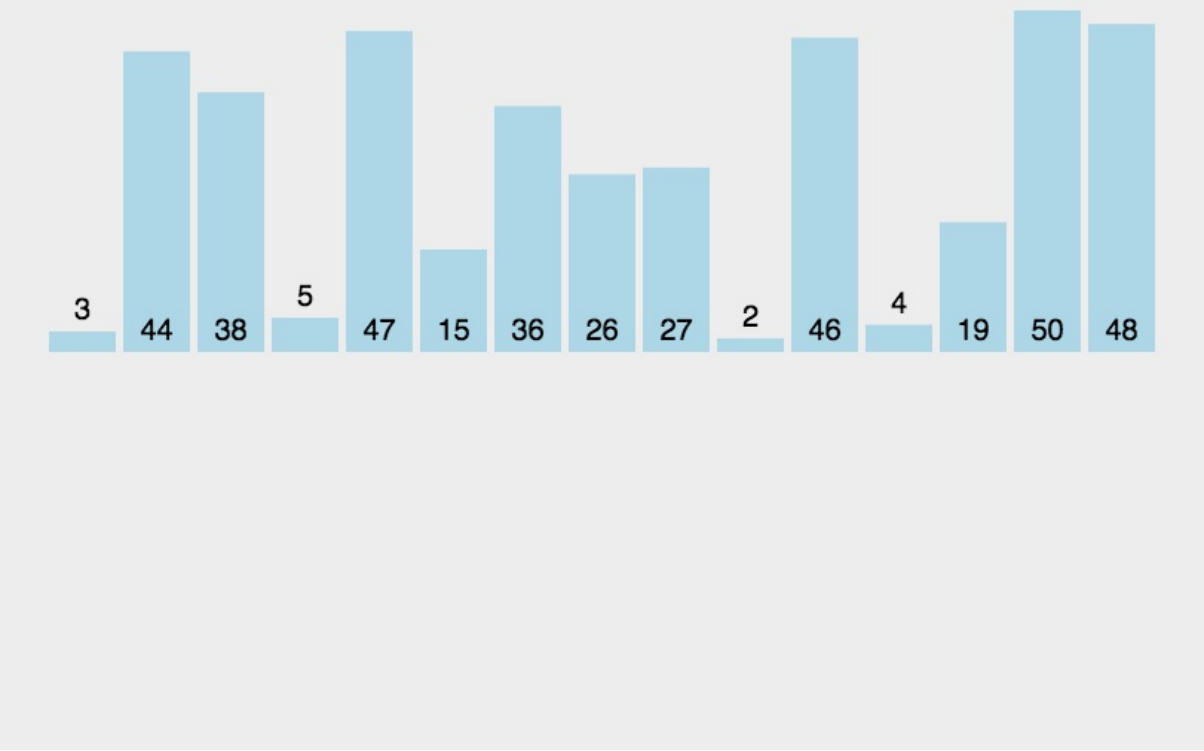
- 最优时间复杂度：根据步长序列的不同而不同
- 最坏时间复杂度：O(n2)
- 稳定想：不稳定

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。



归并排序实现

```
def merge_sort(alist):
    if len(alist) <= 1:
        return alist
    # 二分分解
    num = len(alist)/2
    left = merge_sort(alist[:num])
    right = merge_sort(alist[num:])
    # 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, r = 0, 0
    result = []
    while l<len(left) and r<len(right):
        if left[l] < right[r]:
            result.append(left[l])
            l += 1
        else:
            result.append(right[r])
            r += 1
    result += left[l:]
    result += right[r:]
    return result

alist = [54,26,93,17,77,31,44,55,20]
```

```
sorted_alist = mergeSort(alist)
print(sorted_alist)
```

时间复杂度

- 最优时间复杂度：O(nlogn)
- 最坏时间复杂度：O(nlogn)
- 稳定性：稳定

常见排序算法效率分析

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

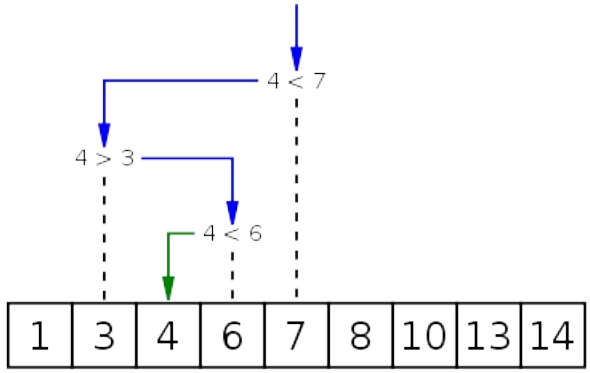
搜索

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

二分法查找

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。



二分查找实现

非递归实现

```
def binary_search(alist, item):
      first = 0
      last = len(alist)-1
      while first<=last:
          midpoint = (first + last)/2
          if alist[midpoint] == item:
              return True
          elif item < alist[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(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)//2
        if alist[midpoint]==item:
          return True
        else:
          if item<alist[midpoint]:
            return binary_search(alist[:midpoint],item)
          else:
            return binary_search(alist[midpoint+1:],item)
```

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

时间复杂度

- 最优时间复杂度：O(1)
- 最坏时间复杂度：O(logn)

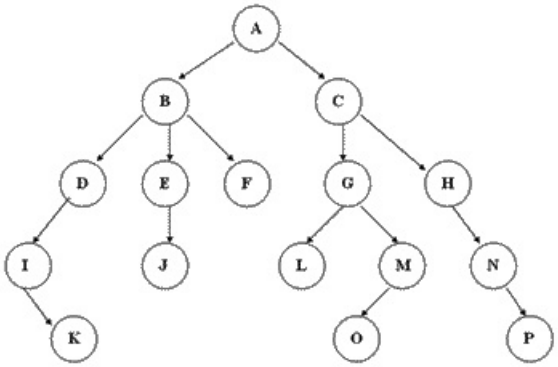
树

树与树算法

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；



树的术语

- 节点的度：一个节点含有的子树的个数称为该节点的度。
- 树的度：一棵树中，最大的节点的度称为树的度。
- 叶节点或终端节点：度为零的节点。
- 父亲节点或父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点。
- 孩子节点或子节点：一个节点含有的子树的根节点称为该节点的子节点。
- 兄弟节点：具有相同父节点的节点互称为兄弟节点。
- 节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推。
- 树的高度或深度：树中节点的最大层次。
- 堂兄弟节点：父节点在同一层的节点互为堂兄弟。
- 节点的祖先：从根到该节点所经分支上的所有节点。
- 子孙：以某节点为根的子树中任一节点都称为该节点的子孙。
- 森林：由m（m>=0）棵互不相交的树的集合称为森林。

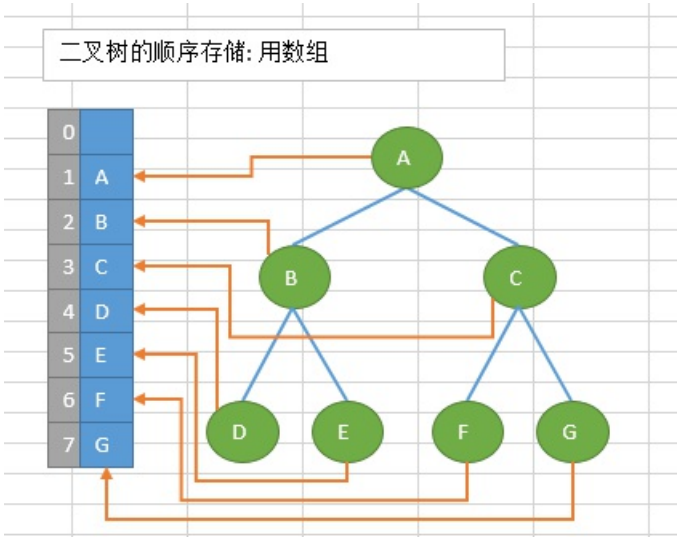
树的种类

- 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。

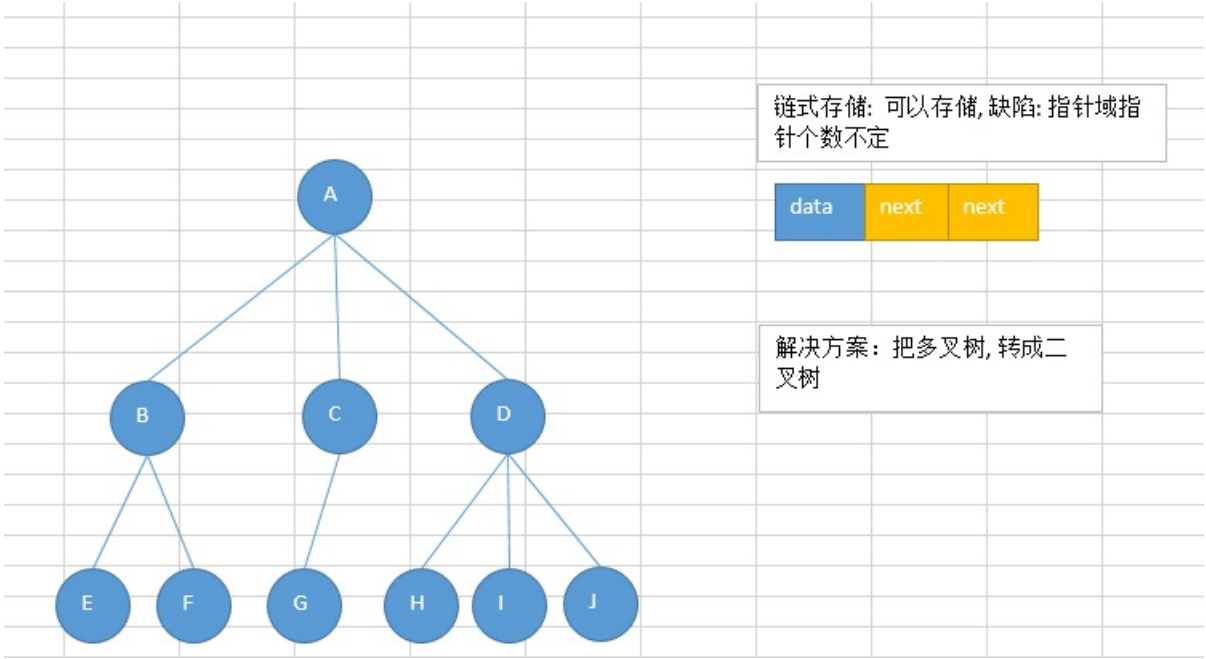
树的存储与表示

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。



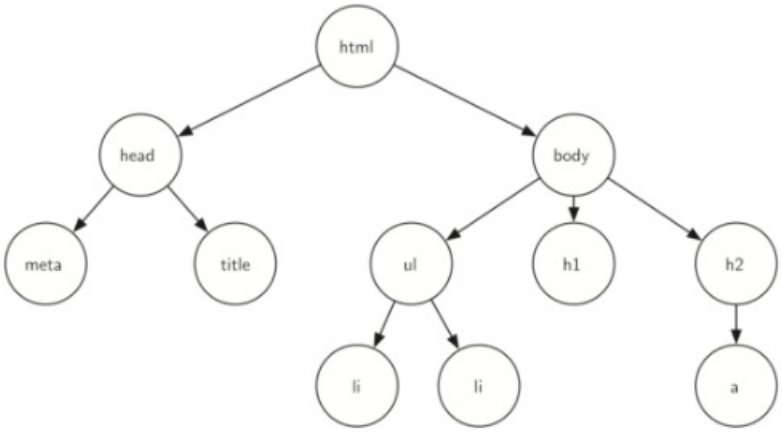
链式存储



由于对节点的个数无法掌握，常见树的存储表示都转换成二叉树进行处理，子节点个数最多为2。

常见的一些树的应用场景

- 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。
- 路由协议就是使用了树的算法。
- mysql数据库索引。
- 文件系统的目录结构。
- 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。



二叉树

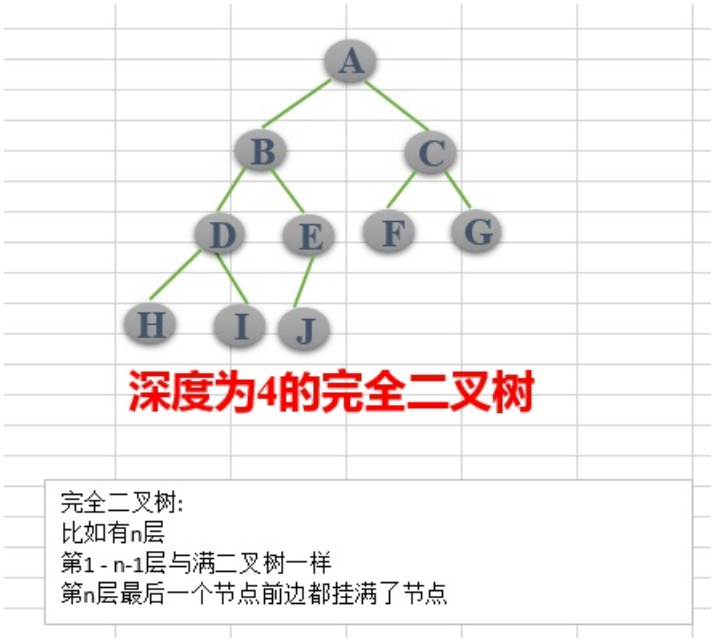
二叉树的基本概念

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）

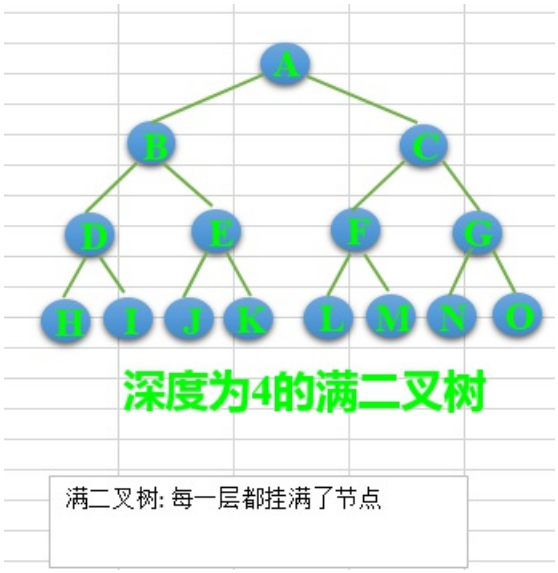
二叉树的性质(特性)

- 性质**1**: 在二叉树的第i层上至多有2^(i-1)个结点（i>0）。
- 性质**2**: 深度为k的二叉树至多有2^k - 1个结点（k>0）。
- 性质**3**: 对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，则N0=N2+1。
- 性质**4**:具有n个结点的完全二叉树的深度必为 log2(n+1)。
- 性质**5**:对完全二叉树，若从上至下、从左至右编号，则编号为i 的结点，其左孩子编号必为2i，其右孩子编号必为2i＋1；其双亲的编号必为i/2（i＝1 时为根,除外）。

完全二叉树——若设二叉树的高度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第h层有叶子结点，并且叶子结点都是从左到右依次排布，这就是完全二叉树。



满二叉树——除了叶结点外每一个结点都有左右子叶且叶子结点都处在最底层的二叉树。



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elem本身的值，还有lchild左孩子和rchild右孩子。

```
class Node(object):
    """节点类"""
    def __init__(self, elem=-1, lchild=None, rchild=None):
        self.elem = elem
        self.lchild = lchild
        self.rchild = rchild
```

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

二叉树的遍历

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

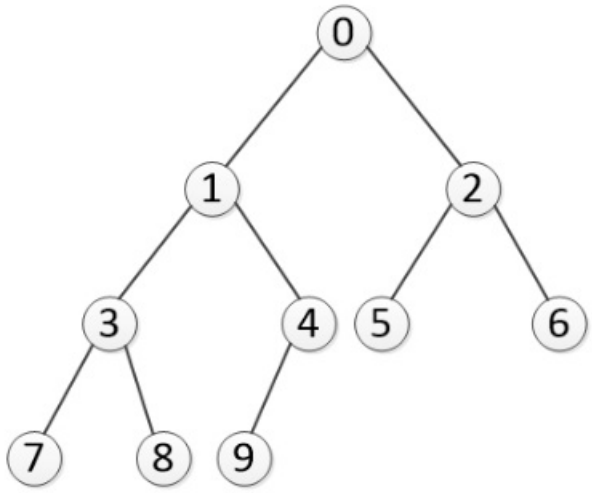
```
def preorder(self, root):
      """递归实现先序遍历"""
      if root == None:
          return
      print root.elem
      self.preorder(root.lchild)
      self.preorder(root.rchild)
```

- 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, root):
      """递归实现中序遍历"""
      if root == None:
          return
      self.inorder(root.lchild)
      print root.elem
      self.inorder(root.rchild)
```

- 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, root):
      """递归实现后续遍历"""
      if root == None:
          return
      self.postorder(root.lchild)
      self.postorder(root.rchild)
      print root.elem
```

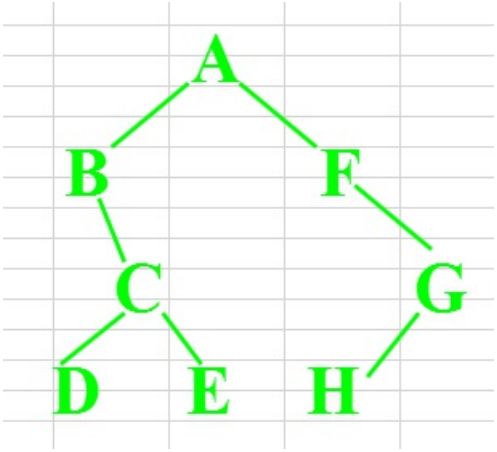


层次遍历：0 1 2 3 4 5 6 7 8 9

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

> 练习： 按照如图树的结构写出三种遍历的顺序:
>
> 
>
>
> 结果:
>
> 先序:a b c d e f g h
>
> 中序:b d c e a f h g
>
> 后序:d e c b h g f a
>
> 思考：哪两种遍历方式能够唯一的确定一颗树？？？

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, root):
        """利用队列实现树的层次遍历"""
        if root == None:
            return
        queue = []
        queue.append(root)
        while queue:
            node = queue.pop(0)
            print node.elem,
            if node.lchild != None:
                queue.append(node.lchild)
            if node.rchild != None:
                queue.append(node.rchild)
```