[笔记][黑马 Python 24 之数据结构与算法]

Python

1. 引入概念

1.1 算法引入

如果写代码是打仗，程序员就是将军，数据结构和算法就是**兵法**。

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?

思路：枚举法

```
"""
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?
"""

import time

start_time = time.time()
for a in range(1001):
    for b in range(1001):
        for c in range(1001):
            if a + b + c == 1000 and a ** 2 + b ** 2 == c ** 2:
                print('a, b, c: %d, %d, %d' % (a, b, c))
end_time = time.time()
print('times: %d' % (end_time - start_time))
print('finished')
"""
a, b, c: 0, 500, 500
a, b, c: 200, 375, 425
a, b, c: 375, 200, 425
a, b, c: 500, 0, 500
times: 132
finished
"""
```

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。

算法的特征

- 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

1.2 时间复杂度与大 O 表示法

第二次尝试

```
"""
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?
"""

import time

start_time = time.time()

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))

end_time = time.time()

print('times: %d' % (end_time - start_time))
print('finished')
"""
a, b, c: 0, 500, 500
a, b, c: 200, 375, 425
a, b, c: 375, 200, 425
a, b, c: 500, 0, 500
```

```
    times: 1
    finished
    """
```

算法效率衡量
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?
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。

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。

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 `3*n^2和100*n^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。

1.3 最坏时间复杂度与计算规则

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**

第一次尝试的时间复杂度为 `T(n) = O(n*n*n) = O(n3)`
第二次尝试的时间复杂度为 `T(n) = O(n*n*(1+1)) = O(n*n) = O(n2)`
我们尝试的第二种算法要比第一种算法的时间复杂度好得多。

1.4 常见时间复杂度与大小关系

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
$5log_2n+20$	O(logn)	对数阶
$2n+3nlog_2n+19$	O(nlogn)	nlogn阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

注意，经常将log_2n（以2为底的对数）简写成logn

> 忽略 常数项，忽略 次要项。

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

1.5 代码执行时间测量模块 timeit

`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=<built-in function perf_counter>, globals=None)`

- `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；
- `globals` 如果指定了这个参数，代码会在该参数指定的命名空间中运行，而不是在 `timeit` 的命名空间中运行

`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**总耗时**，一个 `float` 类型的秒数。

1.6 Python 列表类型不同操作的时间效率

`list` 操作测试：

```
from timeit import Timer


def test1():
    """使用加号拼接列表。"""
    l = []
    for i in range(1000):
        l = l + [i]

def test2():
    """使用 append 追加列表。"""
    l = []
    for i in range(1000):
        l.append(i)

def test3():
    """使用列表推导式。"""
    l = [i for i in range(1000)]

def test4():
    """使用 list 转换可迭代对象。"""
    # Python 2 中想要得到可迭代的 range
    # 需要使用 xrange
    l = list(range(1000))


t1 = Timer('test1()', globals=globals())
t2 = Timer('test2()', globals=globals())
t3 = Timer('test3()', globals=globals())
t4 = Timer('test4()', globals=globals())

print('使用加号拼接列表：%s' % t1.timeit(number = 1000))
print('使用 append 追加列表：%s' % t1.timeit(number =
1000))
print('使用列表推导式：%s' % t1.timeit(number = 1000))
```

```
print('使用 list 转换可迭代对象：%s' % t1.timeit(number = 1000))

"""
使用加号拼接列表：1.8383459586043422
使用 append 追加列表：1.959418708009967
使用列表推导式：1.7276078443271952
使用 list 转换可迭代对象：1.6721926026145377
"""
```

也可以使用 `from __main__ import test1`

> 【注】
> 更方便的方法：

```
from timeit import timeit
t = timeit('test1()', number=1000, globals=globals())
print(t)
```

在头部 `insert(0, i)` 比在尾部添加 `append(i)` **慢很多**：

```
def test5():
    """在末尾添加元素。"""
    li = []
    for i in range(1000):
        li.append(i)


def test6():
    """在头部添加元素。"""
    li = []
    for i in range(1000):
        li.insert(0, i)


print('在末尾 append：%s' % timeit('test5()', number=1000, globals=globals()))
```

```
print('在头部 insert：%s' % timeit('test6()', number=1000,
 globals=globals()))
"""
在末尾 append：0.08858736782733523
在头部 insert：0.5678873078543223
"""
```

头部取元素 `pop(0)` 也比尾部取元素 `pop()` 慢很多：

```
my_li = list(range(10000))
def test7():
    """在头部取元素。"""
    head = my_li.pop(0)
print('在头部取元素：%s' % timeit('test7()', number=1000, g
lobals=globals()))

my_li = list(range(10000))
def test8():
    """在尾部取元素。"""
    tail = my_li.pop()
print('在尾部取元素：%s' % timeit('test8()', number=1000, g
lobals=globals()))

"""
在头部取元素：0.0030223006789800166
在尾部取元素：0.0001981251294065344 8
"""
```

1.7 Python 列表与字典操作的时间复杂度

list内置操作的时间复杂度

Operation	Big-O Efficiency
indexx[]	$O(1)$
index assignment	$O(1)$
append	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
insert(i,item)	$O(n)$
del operator	$O(n)$
iteration	$O(n)$
contains (in)	$O(n)$
get slice [x:y]	$O(k)$
del slice	$O(n)$
set slice	$O(n+k)$
reverse	$O(n)$
concatenate	$O(k)$
sort	$O(n \log n)$
multiply	$O(nk)$

Table 2.2: Big-O Efficiency of Python List Operators

`set slice` 指的是切片赋值。

dict内置操作的时间复杂度

Operation	Big-O Efficiency
copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$

Table 2.3: Big-O Efficiency of Python Dictionary Operations

1.8 数据结构引入

补充上节：
尽量少用加号 `+`。
多用 `append`，`extend` 和 `+=`。

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。**

> 【注】数据结构就是数据的组织方式。

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。**

> 有一种面向对象的感觉。

引入抽象数据类型的**目的**是把数据类型的表示和数据类型上运算的实现与这些数据类型和运算在程序中的引用隔开，使它们相互独立。

最常用的数据运算有五种：

- 插入
- 删除
- 修改
- 查找
- 排序

---

2. 线性表（顺序表和链表）

2.1 内存、类型本质、连续存储

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。

根据线性表的实际存储方式，分为两种实现模型：

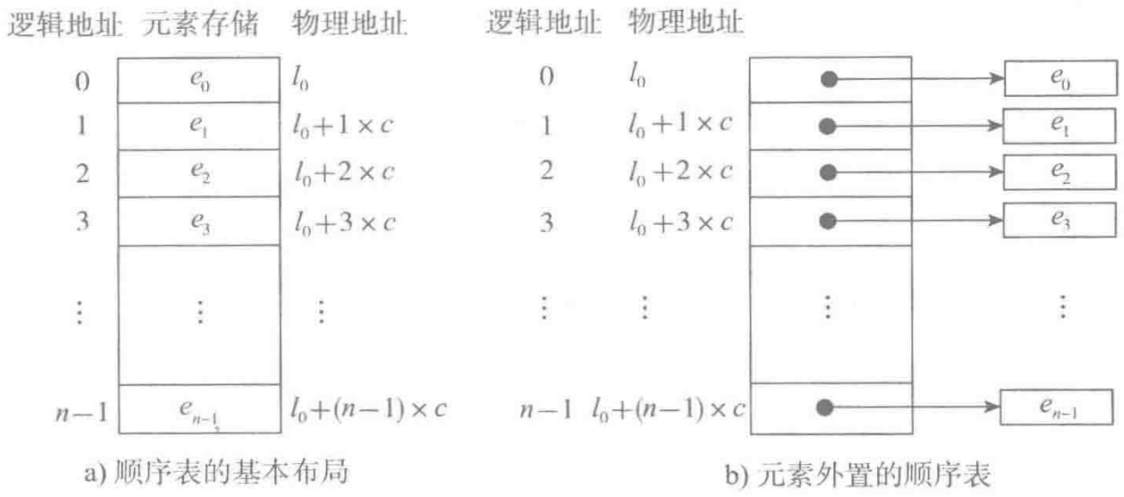
- 顺序表，将元素顺序地存放在一块连续的存储区里，元素间的顺序关系由它们的存储顺序自然表示。
- 链表，将元素存放在通过链接构造起来的一系列存储块中。

内存中的一个字节是一个存储单元，有单独的地址。

2.2 基本顺序表与元素外置顺序表

`32` 位系统，一个整型占用 `4` 个字节，`32` 个比特位。
操作系统标识计算机内存的时候，最小寻址单位是按字节，而不是按位。一个字节就代表了一个地址单元。

顺序表的基本形式



a) 顺序表的基本布局　　b) 元素外置的顺序表

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

```
Loc(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)`。

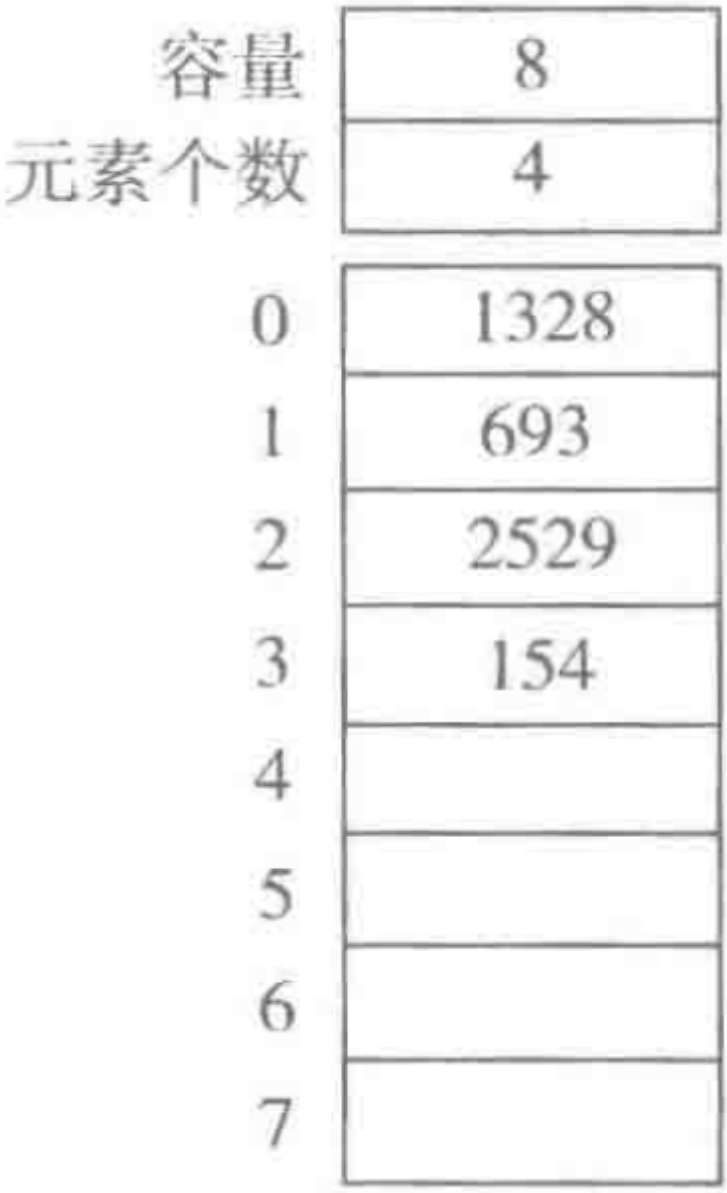
如果元素的大小不统一，则须采用图 b 的 元素外置的形式，将实际数据元素另行存储，而**顺序表中各单元位置保存对应元素的地址信息**（即链接）。由于每个链接所需的存储量相同，通过上述公式，可以计算出元素链接的存储位置，而后顺着链接找到实际存储的数据元素。注意，图 b 中的 c 不再是数据元素的大小，而是**存储一个链接地址所需的存储量，这个量通常很小。**

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。

> 下标从零开始算，因为下标实际上是**内存地址的偏移量**。

2.3 顺序表的一体式结构与分离式结构

顺序表的结构



一个顺序表的完整信息包括两部分：

- 表中的元素集合（**数据区**）
- 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（**表头信息**）
  - 元素存储区的 容量
  - 已有的 元素个数

顺序表的两种基本实现方式

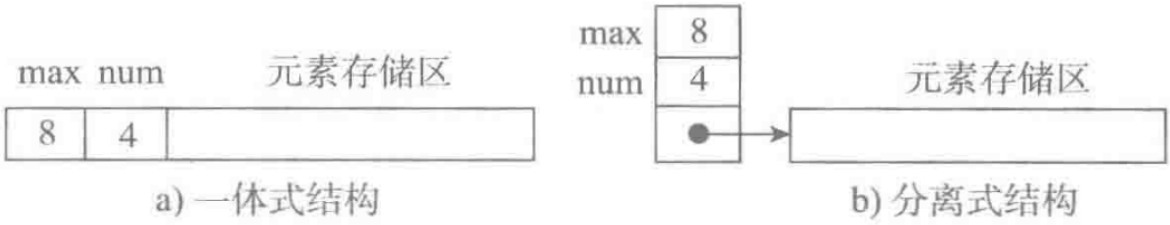


图 a 为 一体式结构，存储表信息的单元与元素存储区以连续的方式安排在一块存储区里，两部分数据的整体形成一个完整的顺序表对象。
一体式结构整体性强，易于管理。但是由于数据元素存储区域是表对象的一部分，顺序表创建后，元素存储区就固定了。

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。

2.4 顺序表数据区替换与扩充

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。

> 对象变了意思就是**变量名字指向的内存地址变了，因为表头信息变了**。

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

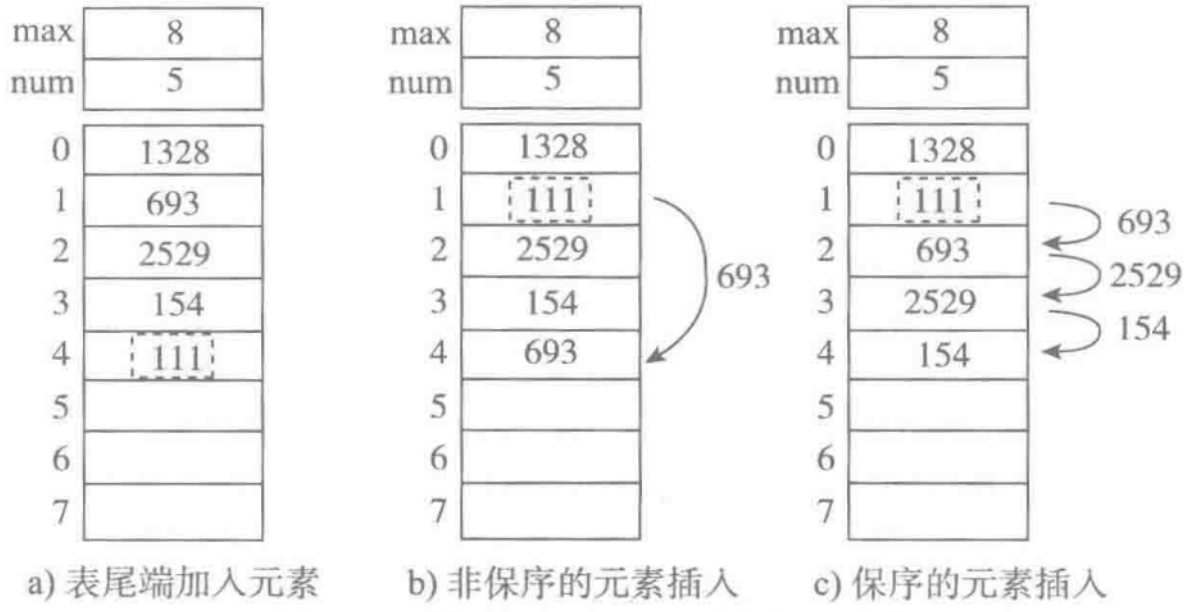
- 每次扩充增加固定数目的存储位置，如每次扩充增加 `10` 个元素位置，这种策略可称为**线性增长**。
- 特点：节省空间，但是扩充操作频繁，操作次数多。
- 每次扩充容量加倍，如每次扩充增加一倍存储空间。
- 特点：减少了扩充操作的执行次数，但可能会浪费空间资源。以 空间换时间，推荐的方式。

> 支持存储区扩充的顺序表被称为 动态数据表。

2.5 顺序表的操作以及 Python 列表的实现

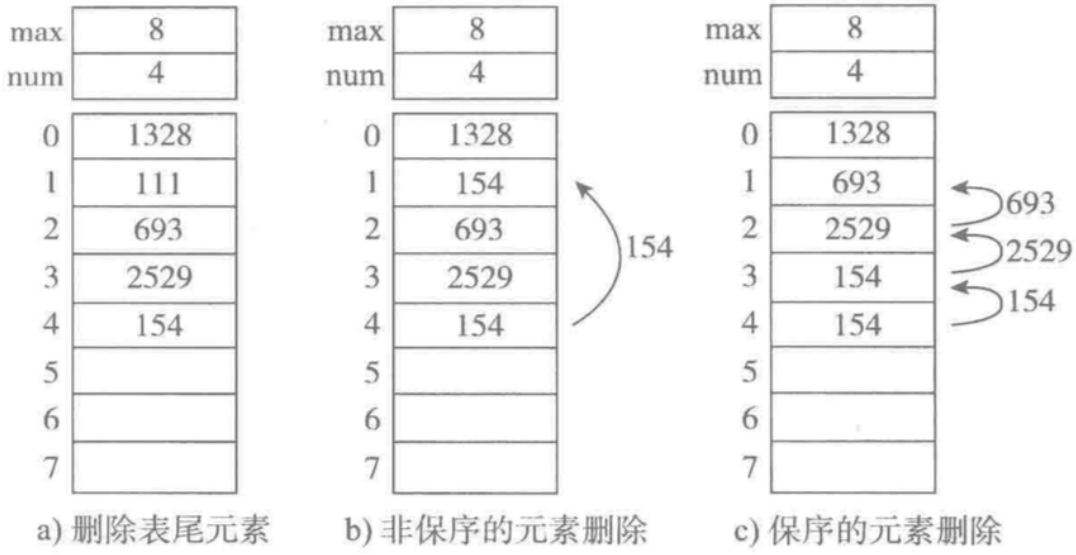
增加元素

如图所示，为顺序表增加新元素 `111` 的三种方式



- 尾端加入元素，时间复杂度为 `O(1)`
- 非保序 的加入元素（不常见），时间复杂度为 `O(1)`
- 保序 的元素加入，时间复杂度为 `O(n)`

删除元素



- 删除表尾元素，时间复杂度为 `O(1)`
- 非保序的元素删除（不常见），时间复杂度为 `O(1)`
- 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。
- 元素外置
- 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。

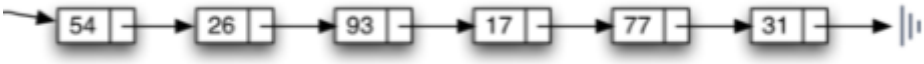
---

3. 链表

3.1 链表的提出

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。

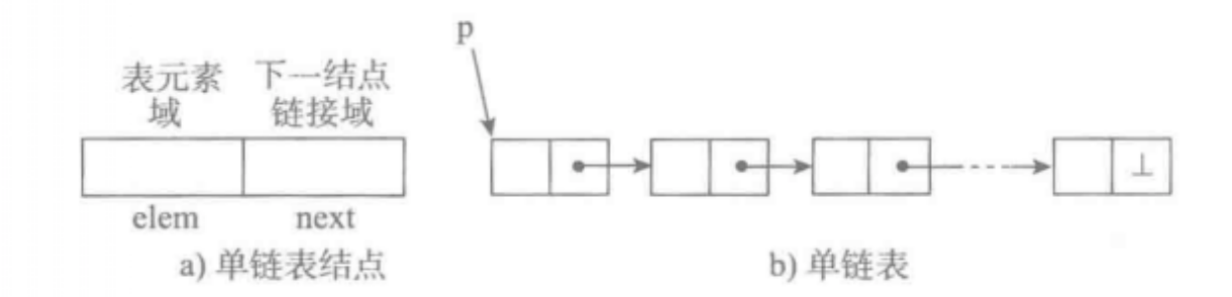
链表的定义
链表（`Linked list`）是一种常见的基础数据结构，是一种线性表，但是不像顺序表一样连续存储数据，而是在每一个节点（数据存储单元）里存放下一个节点的位置信息（即地址）。



链表的一个单元叫做**节点**，不能叫做**元素**。
因为该 节点 由 数据区 和 链接区 组成，也可以叫做 指针区。

3.2 单链表的 ADT 模型

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。

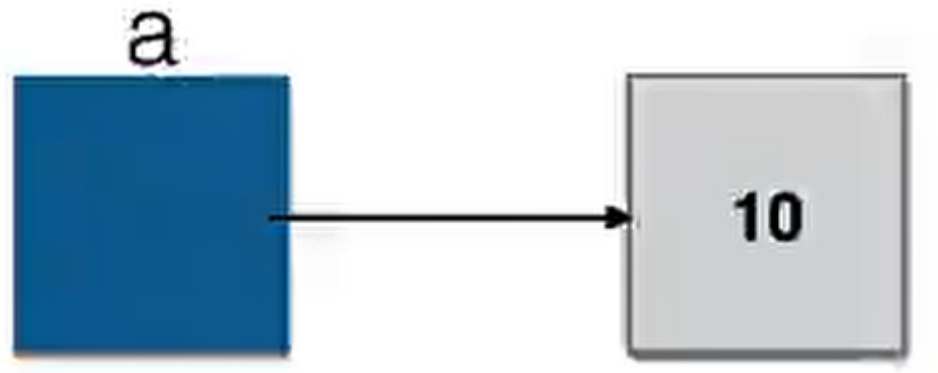


> 补充：一竖一横代表指向空，尾节点的链接是指向空的。

第一个节点叫做 头节点。
最后一个节点叫做 尾节点。

3.3 Python 中变量标识的本质

在别的编程语言中，变量名 `a` 就是数据所在地的内存地址的别名。
然而在 `Python` 里面，变量名 `a` 找到的内存空间里面，存放的是指向数据的指针。




```
a = 10
b = 20
a, b = b, a
```

最后一句相当于 `a, b = 20, 10`
赋值其实是改变了变量的引用地址。

`Python` 中的变量仅仅是一个名字，它真实维护的是一个地址，地址指向的东西不同，`a` 能代表的东西也就不同。

3.4 单链表及节点的定义代码

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

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


class SinglyLinkedList:
    """单链表"""
    def __init__(self, node=None):
        self._head = node

    def is_empty(self):
        """链表是否为空"""
        pass

    def length(self):
        """链表长度"""
        pass

    def travel(self):
```

```
        """遍历整个链表"""
        pass

    def add(self, item):
        """链表头部添加元素"""
        pass

    def append(self, item):
        """链表尾部添加元素"""
        pass

    def insert(self, pos, item):
        """指定位置添加元素"""
        pass

    def remove(self, item):
        """删除节点"""

    def search(self, item):
        """查找节点是否存在"""
        pass
```

3.5 单链表的判空、长度、遍历与尾部添加结点的代码实现

```
class Node:
    """节点"""

    def __init__(self, elem):
        self.elem = elem
        self.next = None


class SinglyLinkedList:
    """单链表"""
```

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

    def is_empty(self):
        """链表是否为空"""
        # return True if self._head is None else False
        # 推荐写法:
        return self._head is None

    def length(self):
        """链表长度"""
        # cur 游标，用来移动遍历节点
        cur = self._head
        # count 记录数量
        count = 0
        while cur:
            count += 1
            cur = cur.next
        return count

    def travel(self):
        """遍历整个链表"""
        cur = self._head
        while cur:
            print(cur.elem)
            cur = cur.next

    def add(self, item):
        """链表头部添加元素"""
        pass

    def append(self, item):
        """链表尾部添加元素"""
        # 注意这个 item 不是节点对象，而是数据
        # 所以要新建一个节点
        node = Node(item)
        cur = self._head
        if self.is_empty():
            self._head = node
            return
```

```
            while cur.next:
                cur = cur.next
            cur.next = node

    def insert(self, pos, item):
        """指定位置添加元素"""
        pass

    def remove(self, item):
        """删除节点"""

    def search(self, item):
        """查找节点是否存在"""
        pass


if __name__ == '__main__':
    sll = SinglyLinkedList()
    print(sll.is_empty())
    print(sll.length())

    print('-' * 20)
    sll.append(1)
    print(sll.is_empty())
    print(sll.length())

    print('-' * 20)
    sll.append(2)
    sll.append(3)
    sll.append(4)
    sll.append(5)
    sll.append(6)
    sll.travel()

"""
True
0
--------------------
False
1
--------------------
```

```
1
2
3
4
5
6
"""
```

3.6 单链表尾部添加和在指定位置添加

在 `Python 2` 中关闭换行
`print something,`
在 `Python 3` 中关闭换行
`print(something, end=' ')`

私有变量应该用 `__` 前导。
在链表尾部添加元素：尾插法。
在链表头部添加元素：头插法。

换行：

- `print()` 换一行
- `print('')` 换一行
- `print('\n')` **换两行!**

```
class Node:
    """节点"""

    def __init__(self, elem):
        self.elem = elem
        self.next = None


class SinglyLinkedList:
    """单链表"""
```

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

    def is_empty(self):
        """链表是否为空"""
        # return True if self.__head is None else False
        # 推荐写法:
        return self.__head is None

    def length(self):
        """链表长度"""
        # cur 游标，用来移动遍历节点
        cur = self.__head
        # count 记录数量
        count = 0
        while cur:
            count += 1
            cur = cur.next
        return count

    def travel(self):
        """遍历整个链表"""
        cur = self.__head
        while cur:
            print(cur.elem)
            cur = cur.next

    def add(self, item):
        """链表头部添加元素，头插法。"""
        node = Node(item)
        self.__head, node.next = node, self.__head
        # 如果采用下面这种写法，要注意顺序！
        # node.next = self.__head
        # self.__head = node

    def append(self, item):
        """链表尾部添加元素，尾插法。"""
        # 注意这个 item 不是节点对象，而是数据
        # 所以要新建一个节点
        node = Node(item)
```

```
            cur = self.__head
            if self.is_empty():
                self.__head = node
                return
            while cur.next:
                cur = cur.next
            cur.next = node

        def insert(self, pos, item):
            """指定位置添加元素
            :param pos 从 0 开始
            """
            if pos <= 0:
                self.add(item)
            elif pos >= self.length():
                self.append(item)
            else:
                pre = self.__head
                count = 0
                while count < pos - 1:
                    count += 1
                    pre = pre.next
                # 当循环退出后，pre 指向 pos - 1 位置
                node = Node(item)
                node.next = pre.next
                pre.next = node

        def remove(self, item):
            """删除节点"""

        def search(self, item):
            """查找节点是否存在"""
            pass


    if __name__ == '__main__':
        sll = SinglyLinkedList()
        print(sll.is_empty())
        print(sll.length())

        print('-' * 20)
```

```
    sll.append(1)
    print(sll.is_empty())
    print(sll.length())

    print('-' * 20)
    sll.append(2)
    sll.add(8)
    sll.append(3)
    sll.append(4)
    sll.append(5)
    sll.append(6)
    # 8 1 2 3 4 5 6
    sll.insert(-1, 9)  # 9 8 1 2 3 4 5 6
    sll.travel()
    print('-' * 20)
    sll.insert(2, 100)  # 9 8 100 1 2 3 4 5 6
    sll.travel()
    print('-' * 20)
    sll.insert(10, 200)  # 9 8 100 1 2 3 4 5 6 200
    sll.travel()

"""
True
0
--------------------
False
1
--------------------
9
8
1
2
3
4
5
6
--------------------
9
8
100
1
```

```
2
3
4
5
6
--------------------
9
8
100
1
2
3
4
5
6
200
"""
```

3.7 单链表查找和删除元素

后继节点：某个节点的下一个节点叫做后继节点。

```
class Node:
    """节点"""

    def __init__(self, elem):
        self.elem = elem
        self.next = None


class SinglyLinkedList:
    """单链表"""

    def __init__(self, node=None):
        self.__head = node

    def is_empty(self):
```

```
        """链表是否为空"""
        # return True if self.__head is None else False
        # 推荐写法:
        return self.__head is None

    def length(self):
        """链表长度"""
        # cur 游标，用来移动遍历节点
        cur = self.__head
        # count 记录数量
        count = 0
        while cur:
            count += 1
            cur = cur.next
        return count

    def travel(self):
        """遍历整个链表"""
        cur = self.__head
        while cur:
            print(cur.elem, end=' ')
            cur = cur.next
        print()

    def add(self, item):
        """链表头部添加元素，头插法。"""
        node = Node(item)
        self.__head, node.next = node, self.__head
        # 如果采用下面这种写法，要注意顺序！
        # node.next = self.__head
        # self.__head = node

    def append(self, item):
        """链表尾部添加元素，尾插法。"""
        # 注意这个 item 不是节点对象，而是数据
        # 所以要新建一个节点
        node = Node(item)
        cur = self.__head
        if self.is_empty():
            self.__head = node
            return
```

```
        while cur.next:
            cur = cur.next
        cur.next = node

    def insert(self, pos, item):
        """指定位置添加元素
        :param pos 从 0 开始
        """
        if pos <= 0:
            self.add(item)
        elif pos >= self.length():
            self.append(item)
        else:
            pre = self.__head
            count = 0
            while count < pos - 1:
                count += 1
                pre = pre.next
            # 当循环退出后，pre 指向 pos - 1 位置
            node = Node(item)
            node.next = pre.next
            pre.next = node

    def remove(self, item):
        """删除节点"""
        cur = self.__head
        pre = None
        while cur:
            if cur.elem == item:
                # 先判断此节点是否是头节点
                # 两种方式
                # pre 是 None 或者 cur 是 self.__head
                if cur == self.__head:
                    self.__head = cur.next
                else:
                    pre.next = cur.next
                break
            else:
                pre = cur
                cur = cur.next
```

```
    def search(self, item):
        """查找节点是否存在"""
        cur = self.__head
        while cur:
            if cur.elem == item:
                return True
            else:
                cur = cur.next
        return False

if __name__ == '__main__':
    sll = SinglyLinkedList()
    print(sll.is_empty())
    print(sll.length())

    sll.append(1)
    print(sll.is_empty())
    print(sll.length())

    sll.append(2)
    sll.add(8)
    sll.append(3)
    sll.append(4)
    sll.append(5)
    sll.append(6)
    # 8 1 2 3 4 5 6
    sll.insert(-1, 9)  # 9 8 1 2 3 4 5 6
    sll.travel()

    sll.insert(2, 100)  # 9 8 100 1 2 3 4 5 6
    sll.travel()

    sll.insert(10, 200)  # 9 8 100 1 2 3 4 5 6 200
    sll.travel()

    sll.remove(100)
    sll.travel()

    sll.remove(9)
    sll.travel()
```

```
    sll.remove(200)
    sll.travel()

"""
True
0
False
1
9 8 1 2 3 4 5 6
9 8 100 1 2 3 4 5 6
9 8 100 1 2 3 4 5 6 200
9 8 1 2 3 4 5 6 200
8 1 2 3 4 5 6 200
8 1 2 3 4 5 6
"""
```

3.8 单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。

> 【注】
> 元素外置的顺序表也需要存储指针啊！

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)

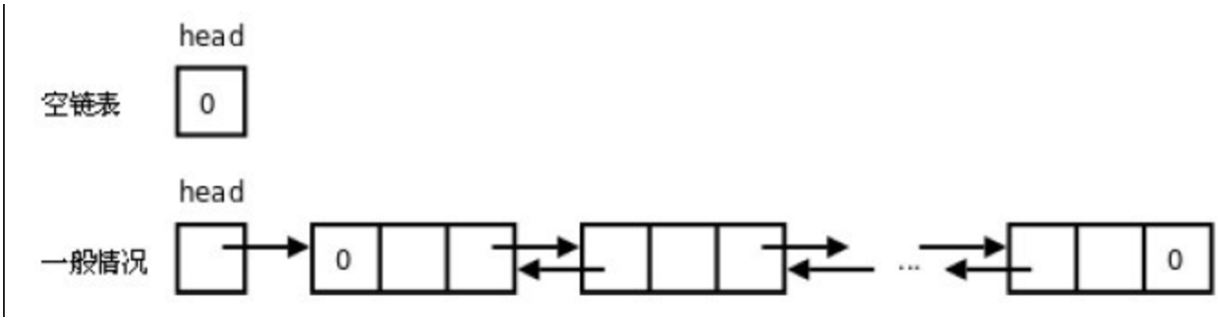
注意
虽然表面看起来复杂度都是 `O(n)`，但是链表和顺序表在插入和删除时进行的是完全不同的操作。

- 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。

3.9 双向链表及添加元素

对于一个节点，有数据区、后继区和前驱区。
后继节点、前驱节点。

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

> 【注】
> 循环退出时，就是循环条件刚好不符合的时候！

```python
from hm004_单链表ADT import SinglyLinkedList

class Node:
    """节点"""
    def __init__(self, elem):
        self.elem = elem
        self.prev = None
        self.next = None


class DoublyLinkedList(SinglyLinkedList):
    """双链表"""
    def __init__(self, node=None):
        self.__head = node


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

    def length(self):
        """链表长度"""
        cur = self.__head
        count = 0
        while cur:
            count += 1
            cur = cur.next
        return count

    def travel(self):
        """遍历整个链表"""
        cur = self.__head
        while cur:
            print(cur.elem, end =' ')
            cur = cur.next
        print()

    def add(self, item):
        """头部添加元素"""
        node = Node(item)
        self.__head.prev = node
```

```
            node.next = self.__head
            self.__head = node

        def append(self, item):
            """尾部添加元素"""
            node = Node(item)
            if self.is_empty():
                self.__head = node
            else:
                cur = self.__head
                while cur.next:
                    cur = cur.next
                cur.next = node
                node.prev = cur

        def insert(self, pos, item):
            """指定位置添加节点。
            :param pos 从 0 开始
            """
            if pos <= 0:
                self.add(item)
            elif pos >= self.length():
                self.append(item)
            else:
                cur = self.__head
                count = 0
                while count < pos:
                    count += 1
                    cur = cur.next
                cur.prev.next = node
                node.prev = cur.prev
                node.next = cur
                cur.prev = node
```

3.10 双向链表删除元素

```
from hm004_单链表ADT import SinglyLinkedList
```

```python
class Node:
    """节点"""

    def __init__(self, elem):
        self.elem = elem
        self.prev = None
        self.next = None


class DoublyLinkedList(SinglyLinkedList):
    """双链表"""

    def __init__(self, node=None):
        self.__head = node

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

    def length(self):
        """链表长度"""
        cur = self.__head
        count = 0
        while cur:
            count += 1
            cur = cur.next
        return count

    def travel(self):
        """遍历整个链表"""
        cur = self.__head
        while cur:
            print(cur.elem, end=' ')
            cur = cur.next
        print()

    def add(self, item):
        """头部添加元素"""
        node = Node(item)
```

```
        self.__head.prev = node
        node.next = self.__head
        self.__head = node

    def append(self, item):
        """尾部添加元素"""
        node = Node(item)
        if self.is_empty():
            self.__head = node
        else:
            cur = self.__head
            while cur.next:
                cur = cur.next
            cur.next = node
            node.prev = cur

    def insert(self, pos, item):
        """指定位置添加节点。
        :param pos 从 0 开始
        """
        if pos <= 0:
            self.add(item)
        elif pos >= self.length():
            self.append(item)
        else:
            cur = self.__head
            count = 0
            while count < pos:
                count += 1
                cur = cur.next
            cur.prev.next = node
            node.prev = cur.prev
            node.next = cur
            cur.prev = node

    def search(self, item):
        cur = self.__head
        while cur:
            if cur.elem == item:
                return True
            else:
```

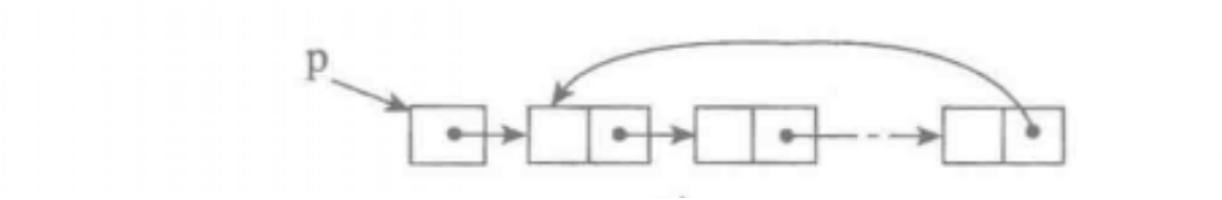
```
                cur = cur.next
        return False

    def remove(self, item):
        cur = self.__head
        while cur:
            if cur.elem == item:
                if cur == self.__head:
                    self.__head = cur.next
                    if cur.next:
                        # 判断链表是否只有一个节点
                        cur.next.prev = None
                else:
                    cur.prev.next = cur.next
                    if cur.next:
                        # 判断是否是最后一个节点
                        cur.next.prev = cur.prev
                break
            else:
                cur = cur.next


if __name__ == '__main__':
    dll = SinglyLinkedList()
    print(dll.is_empty())
    print(dll.length())

    dll.append(1)
    print(dll.is_empty())
    print(dll.length())

    dll.append(2)
    dll.add(8)
    dll.append(3)
    dll.append(4)
    dll.append(5)
    dll.append(6)
    # 8 1 2 3 4 5 6
    dll.insert(-1, 9)  # 9 8 1 2 3 4 5 6
    dll.travel()
```

```
    dll.insert(2, 100)  # 9 8 100 1 2 3 4 5 6
    dll.travel()

    dll.insert(10, 200)  # 9 8 100 1 2 3 4 5 6 200
    dll.travel()

    dll.remove(100)
    dll.travel()

    dll.remove(9)
    dll.travel()

    dll.remove(200)
    dll.travel()

"""
True
0
False
1
9 8 1 2 3 4 5 6
9 8 100 1 2 3 4 5 6
9 8 100 1 2 3 4 5 6 200
9 8 1 2 3 4 5 6 200
8 1 2 3 4 5 6 200
8 1 2 3 4 5 6
"""
```

3.11 单向循环链表遍历和求长度

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

```
class Node:
    def __init__(self, elem):
        self.elem = elem
        self.prev = self.next = None


class SinglyCircularLinkedList:
    def __init__(self, node=None):
        self.__head = node
        if node:
            node.next = node

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

    def length(self):
        if self.is_empty():
            return 0
        cur = self.__head
        count = 1
        while cur.next is not self.__head:
            cur = cur.next
            count += 1
        return count
```

```
    def travel(self):
        if self.is_empty():
            return
        cur = self.__head
        while cur.next is not self.__head:
            print(cur.elem, end=' ')
            cur = cur.next
        # 退出循环时，cur 指向尾节点，但是尾节点未打印
        print(cur.elem)
```

3.12 单向循环链表添加元素

略，后面写完整代码

3.13 单向循环链表删除元素

> 【注】
> 个人体会：**根据循环条件，判断退出循环时变量的状态。**

略，后面写完整代码

3.14 单向循环链表删除元素复习及链表扩展

```
class Node:
    def __init__(self, elem):
        self.elem = elem
        self.next = None
```

```python
class SinglyCircularLinkedList:
    def __init__(self, node=None):
        self.__head = node
        if node:
            node.next = node

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

    def length(self):
        if self.is_empty():
            return 0
        cur = self.__head
        count = 1
        while cur.next is not self.__head:
            cur = cur.next
            count += 1
        return count

    def travel(self):
        if self.is_empty():
            return
        cur = self.__head
        while cur.next is not self.__head:
            print(cur.elem, end=' ')
            cur = cur.next
        # 退出循环时，cur 指向尾节点，但是尾节点未打印
        print(cur.elem)

    def add(self, item):
        """头插法。"""
        node = Node(item)
        cur = self.__head
        if self.is_empty():
            self.__head = node
            node.next = node
        else:
            while cur.next is not self.__head:
                cur = cur.next
            cur.next = node
            node.next = self.__head
```

```
            self.__head = node

    def append(self, item):
        """尾插法。"""
        node = Node(item)
        cur = self.__head
        if self.is_empty():
            self.__head = node
            node.next = node
        else:
            while cur.next is not self.__head:
                cur = cur.next
            node.next = self.__head
            cur.next = node

    def insert(self, pos, item):
        count = 0
        pre = self.__head
        if pos <= 0:
            self.add(item)
        elif pos >= self.length():
            self.append(item)
        else:
            while count < pos - 1:
                count += 1
                pre = pre.next
            node.next = pre.next
            pre.next = node

    def remove(self, item):
        """删除节点。"""
        # 判空
        if self.is_empty():
            return

        cur = self.__head
        pre = None
        while cur.next is not self.__head:
            if cur.elem == item:
                if cur is self.__head:
                    # 头节点
```

```
                    # 找尾节点
                    tail = self.__head
                    while tail.next is not self.__head:
                        tail = tail.next
                    self.__head = cur.next
                    tail.next = self.__head
                else:
                    # 中间节点
                    pre.next = cur.next
                return
            else:
                pre = cur
                cur = cur.next
        # 退出循环，cur 指向尾节点
        # 或者只有一个节点
        if cur.elem == item:
            if cur is self.__head:
                # 只有一个节点
                self.__head = None
            else:
                pre.next = cur.next


    def search(self, item):
        cur = self.__head
        if self.is_empty():
            return False
        while cur.elem != item:
            if cur.next is self.__head:
                return False
            cur = cur.next
        return True
```

扩展：双向循环列表

---

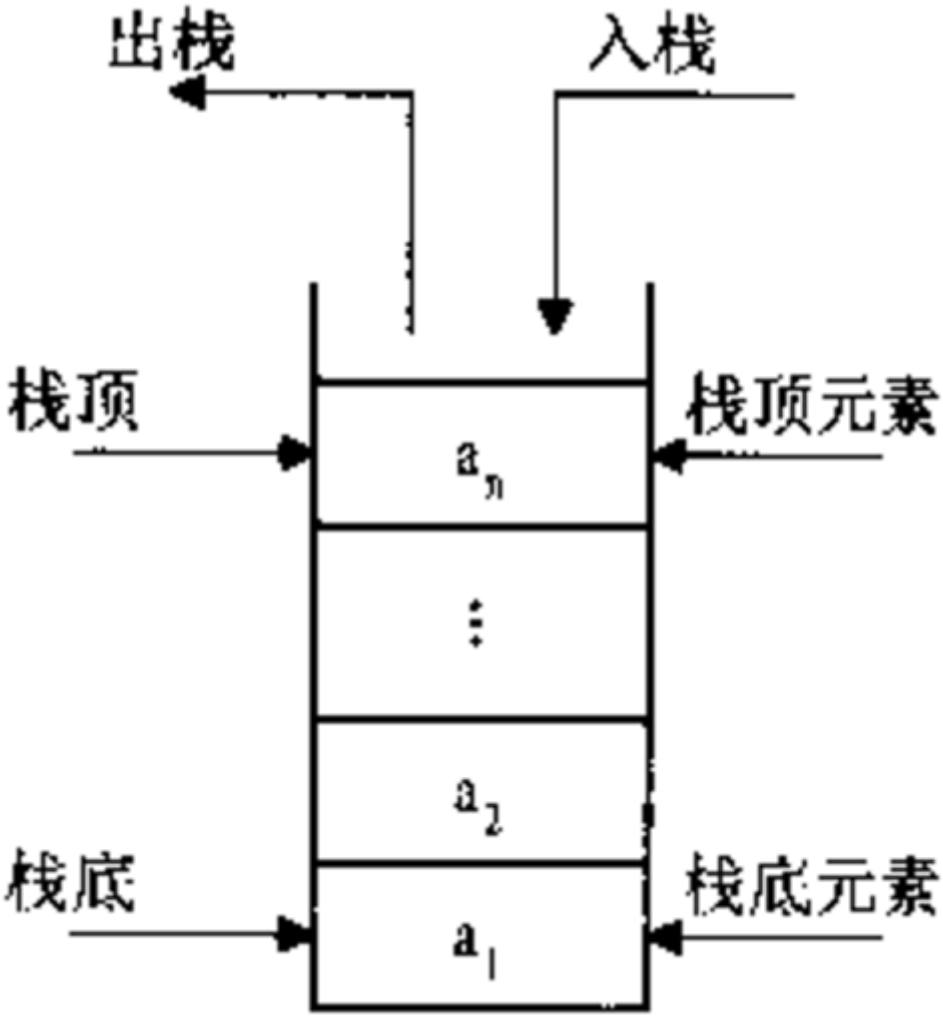
4. 栈与队列

4.1 栈与队列的概念

栈的概念

> 栈就可以用线性数据结构（之前说的顺序表、链表）来实现。

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。
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。

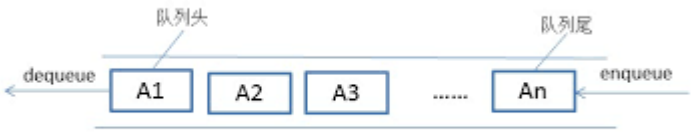


> 在学习栈和队列的时候，不要关心它底层的物理实现。
> 只要关心这两种数据结构的特性、支持什么样的操作、这些操作有什么特点就可以了。
> 栈这种数据结构，描述的是数据怎么操作，而顺序表描述的是数据怎么存放。

栈可以应用在表达式求值。
队列可以应用在树，后面会讲。

队列的概念
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。



4.2 栈的实现

栈可以用顺序表实现，也可以用链表实现。

压栈、入栈、push。
栈底，栈顶。

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

栈的实现

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

    def push(self, item):
        """压栈"""
        # 尾部作为栈顶，因为对于顺序表，尾部存取更快。
        # 如果使用单链表，应该把头部当做栈顶。
        self.__list.append(item)

    def pop(self):
        """弹栈"""
        return self.__list.pop()

    def peek(self):
        """查看栈顶元素"""
        if self.__list:
            return self.__list[-1]
        else:
            return None

    def is_empty(self):
        """判断栈是否为空"""
        # return not len(self.__list)  # 我的写法
        # return not self.__list  # 推荐写法
        return self.__list == []

    def size(self):
        """返回栈的元素个数"""
        return len(self.__list)

if __name__ == '__main__':
    s = Stack()
    s.push(1)
    s.push(2)
    s.push(3)
```

```
    s.push(4)
    print(s.pop())
    print(s.pop())
    print(s.pop())
    print(s.pop())
```

4.3 队列与双端队列的实现

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:
    """队列"""

    def __init__(self):
        # 队头为 0
        # 队尾为 -1
        self.__list = []

    def enqueue(self, item):
        """入队"""
        self.__list.append(item)

    def dequeue(self):
        """出队"""
        return self.__list.pop(0)

    def is_empty(self):
        """判空"""
        return not self.__list
```

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


if __name__ == '__main__':
    q = Queue()
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)
    q.enqueue(4)
    print(q.dequeue())
    print(q.dequeue())
    print(q.dequeue())
    print(q.dequeue())

"""
1
2
3
4
"""
```

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。

双端队列可以在队列任意一端入队和出队。

> 只考虑双端队列的一端的情况下，它具有栈的特性。

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

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

    def add_front(self, item):
        # 列表的头就是双端队列的头
        self.__list.insert(0, item)

    def add_rear(self, item):
        self.__list.append(item)

    def remove_front(self):
        return self.__list.pop(0)

    def remove_rear(self):
        return self.__list.pop()

    def is_empty(self):
        return not self.__list

    def size(self):
        return len(self.__list)

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())

"""
4
2
1
4
3
"""
```

5. 排序与搜索

5.1 排序算法的稳定性

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。

稳定性：稳定排序算法会让相等的记录维持原来的次序。

比如要从小到大排序：

```
(4, 1), (3, 1), (3, 7), (5, 6)
```

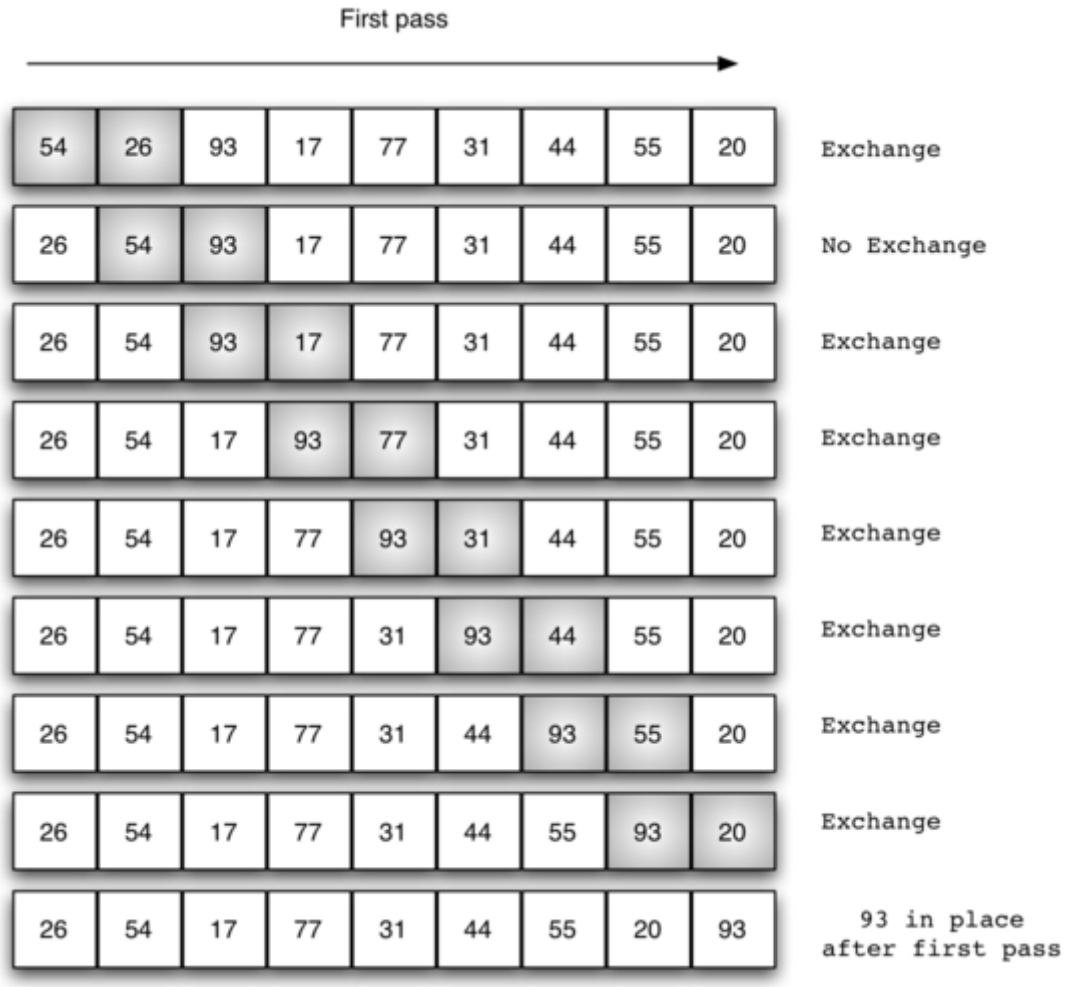
在这个状况下，有可能产生两种不同的结果，一个是让相等键值的纪录维持相对的次序，而另外一个则没有：

```
# 维持次序
(3, 1), (3, 7), (4, 1), (5, 6)

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

不稳定排序算法可能会改变相等键值的次序，但是稳定排序算法不会。
不稳定排序算法可以被实现为稳定排序。
一个方法是增加键值的比较，可以在其他方面把键值相同的两个对象进行比较。
比如在上面的比较中加入第二个标准：第二个键值的大小。
相当于一次同分决赛。
然而，要记住这种次序通常牵涉到额外的空间负担。

5.2 冒泡排序

冒泡排序，英语：`Bubble Sort`。



> 【注】
> 个人理解：每次冒泡过程，都把最大的数固定到了最后面，一次固定一个数。

> 冒泡到了最后一位。
> 所以要固定 `n - 1` 次，因为最后一个数不用再进行冒泡过程。

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

> 【注】
> 左右两边相加为 `n`！

个人理解：一个是**冒泡过程**（大循环、外层循环），第二个是**比较过程**（小循环、内层循环）
要知道：你要冒多少次泡？每次冒泡要比较多少次？

```python
# 外层循环控制要走多少次
# 内层循环控制从头走到尾

def bubble_sort(li):
    n = len(li)
    # 冒泡 n - 1 次
    for i in range(n - 1):
        # 每次比较 n - i 次
        for j in range(n - i - 1):
            if li[j] > li[j + 1]:
                li[j], li[j + 1] = li[j + 1], li[j]

if __name__ == '__main__':
    li = [1, 4, 2, 8, 5, 7, 3]
    print(li)
    print('-' * 20)
    bubble_sort(li)
    print(li)
```

```
"""
[1, 4, 2, 8, 5, 7, 3]
--------------------
[1, 2, 3, 4, 5, 7, 8]
"""
```

还有一种写法

```
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)
```

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(n2)`
稳定性：稳定

优化

```
# 外层循环控制要走多少次
# 内层循环控制从头走到尾

def bubble_sort(li):
    n = len(li)
    # 冒泡 n - 1 次
    for i in range(n - 1):
        count = 0
        # 每次比较 n - i 次
        for j in range(n - i - 1):
            if li[j] > li[j + 1]:
```

```
                li[j], li[j + 1] = li[j + 1], li[j]
                count += 1
        # 如果某次冒泡过程中没有发生交换，则全部有序
        # 直接返回
        if count == 0:
            return

if __name__ == '__main__':
    li = [1, 4, 2, 8, 5, 7, 3]
    print(li)
    print('-' * 20)
    bubble_sort(li)
    print(li)
"""
[1, 4, 2, 8, 5, 7, 3]
--------------------
[1, 2, 3, 4, 5, 7, 8]
"""
```

5.3 选择排序算法及实现

选择排序：选择最小的，交换到最前面，以此类推。
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。

```
def selection_sort(li):
    n = len(li)
    for i in range(n - 1):
        min_i = i
        for j in range(i, n):
            if li[j] < li[min_i]:
                min_i = j
        if li[i] > li[min_i]:
            li[i], li[min_i] = li[min_i], li[i]
```

```
if __name__ == '__main__':
    li = [1, 4, 2, 8, 5, 7, 3]
    print(li)
    print('-' * 20)
    selection_sort(li)
    print(li)
"""
[1, 4, 2, 8, 5, 7, 3]
--------------------
[1, 2, 3, 4, 5, 7, 8]
"""
```

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

> 来自百度知道的例子：`5, 8, 5, 2, 9`

5.4 插入算法

插入排序（英语：`Insertion Sort`）
个人理解：扫描无序序列，插入到有序序列。插入其实是采用逐个比较交换的方式换过去。

5.5 插入排序1

略，代码后面写

5.6 插入排序2

```
def insertion_sort(li):
    n = len(li)
    # 需要选 n - 1 个元素
    for i in range(1, n):
        j = i
        while j > 0:
            if li[j] < li[j - 1]:
                li[j - 1], li[j] = li[j], li[j - 1]
            else:
                break
            j -= 1


if __name__ == '__main__':
    li = [1, 4, 2, 8, 5, 7, 3]
    print(li)
    print('-' * 20)
    insertion_sort(li)
    print(li)
"""
[1, 4, 2, 8, 5, 7, 3]
--------------------
[1, 2, 3, 4, 5, 7, 8]
"""
```

还有一种写法

```
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]
            else:  # 我添加的优化！
                break
```

```
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(n2)`
- 稳定性：稳定

5.7 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)：
`gap = 5`

```
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：

`gap = 3`

```
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）。

5.8 希尔排序实现

```python
def shell_sort(li):
    """希尔排序"""
    n = len(li)
    gap = n // 2
```

```
    while gap:  # 我的写法，比较简洁
        # 控制步长的缩短
        for j in range(gap, n):
            # 插入算法，与普通插入算法的区别就是 gap 步长
            i = j
            while i > 0:
                if li[i] < li[i - gap]:
                    li[i], li[i - gap] = li[i - gap], li
[i]
                else:
                    break
                i -= gap
        # 缩短 gap 步长
        gap = gap // 2

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

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)`
稳定性：不稳定

5.9 快速排序

略，后面写代码

5.10 快速排序实现 1

略

5.11 快速排序实现 2

注意：相等的数据出现在一边。
比如拿 `54` 作为基准值 `pivot`，那么它们要出现在一边。

```
def quick_sort(li, first, last):
    """快速排序"""
    if first >= last:
        return

    pivot = li[first]
    low = first
    high = last

    while low < high:
        # high 左移
        while low < high and li[high] >= pivot:
            high -= 1
        li[low] = li[high]

        # low 右移
        while low < high and li[low] < pivot:
            low += 1
        li[high] = li[low]

    # 从循环退出时，low == high
    li[low] = pivot
    quick_sort(li, first, low - 1)
    quick_sort(li, low + 1, last)


if __name__ == '__main__':
    li = [54, 26, 93, 17, 77, 31, 44, 55, 20]
    print(li)
    print('-' * 40)
    quick_sort(li, 0, len(li) - 1)
    print(li)

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

```
----------------------------------------
[17, 20, 26, 31, 44, 54, 55, 77, 93]
"""
```

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

> 个人理解：计算它的时间复杂度，就是要看每次递归中的时间复杂度，还有总共有多少层递归。如果基准值选的好，每次都是对半分，那么一共递归 `logn` 层，每层时间复杂度为 `n`，所以最优时间复杂度为 `O(nlogn)`。最坏情况下，原列表已经是升序排列，需要有 `n` 层递归，这样最坏时间复杂度为 `O(n2)`。

5.12 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。
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。

```
def merge_sort(li):
    """归并排序"""
    n = len(li)
    if n <= 1:
        return li
    mid = n // 2

    left_li = merge_sort(li[:mid])
    right_li = merge_sort(li[mid:])

    left_pointer, right_pointer = 0, 0
    result = []
```

```
    while left_pointer < len(left_li) and right_pointer < len(right_li):
        if left_li[left_pointer] <= right_li[right_pointer]:
            result.append(left_li[left_pointer])
            left_pointer += 1
        else:
            result.append(right_li[right_pointer])
            right_pointer += 1

    result += left_li[left_pointer:]
    result += right_li[right_pointer:]

    return result

if __name__ == '__main__':
    li = [54, 26, 93, 17, 77, 31, 44, 55, 20]
    print(li)
    print('-' * 40)
    li = merge_sort(li)
    print(li)

"""
[54, 26, 93, 17, 77, 31, 44, 55, 20]
----------------------------------------
[17, 20, 26, 31, 44, 54, 55, 77, 93]
"""
```

5.13 归并排序代码执行流程

5.14 归并排序时间复杂度及排序算法复杂度对比

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

每一级合并，需要的时间复杂度为 `n`，一共有 `logn` 级，所以时间复杂度为 `O(nlogn)`

归并算法返回新列表，有额外的空间开销。

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

重要：必须掌握 快速排序!

5.15 二分查找

搜索：在序列中找一个元素是否存在。

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。

```
def binary_search1(li, item):
    """递归版本：二分查找"""
    n = len(li)
    if n > 0:
        mid = n // 2
        if li[mid] == item:
```

```
            return True
        elif item < li[mid]:
            return binary_search1(li[:mid], item)
        else:
            return binary_search1(li[mid + 1:], item)
    else:
        return False


def binary_search2(li, item):
    """非递归版本：二分查找"""
    n = len(li)
    low = 0
    high = n - 1

    while low <= high:
        mid = (high + low) // 2
        if li[mid] == item:
            return True
        elif li[mid] > item:
            high = mid - 1
        elif li[mid] < item:
            low = mid + 1

    return False


if __name__ == '__main__':
    li = [1, 2, 3, 4, 5, 7, 8, 9]
    print(binary_search1(li, 6))
    print(binary_search1(li, 3))
    print(binary_search2(li, 6))
    print(binary_search2(li, 3))

"""
False
True
False
True
"""
```

5.16 二分查找时间复杂度

时间复杂度
最优时间复杂度：`O(1)`
最坏时间复杂度：`O(logn)`

> 比起简单查找，最坏时间复杂度有所改进。

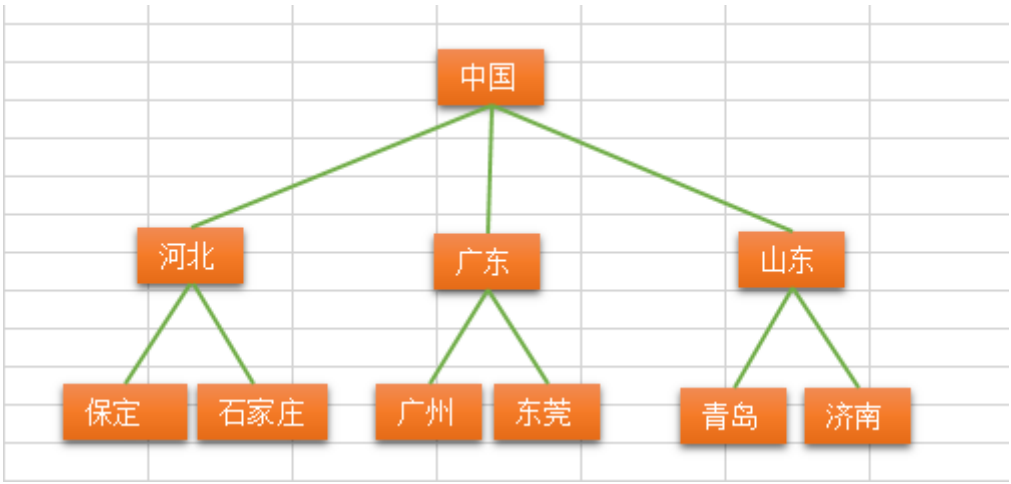
6. 树和树的算法

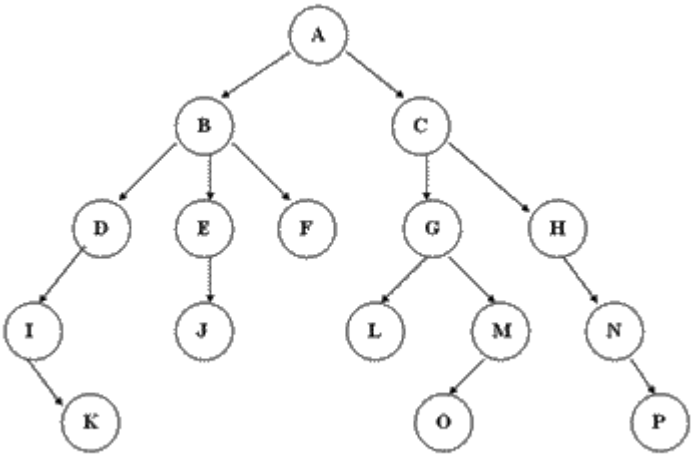
6.1 树的概念

树（英语：tree）是一种抽象数据类型（ADT）。
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。

树的特点

- 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**；

> 无序树没有任何研究价值

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；

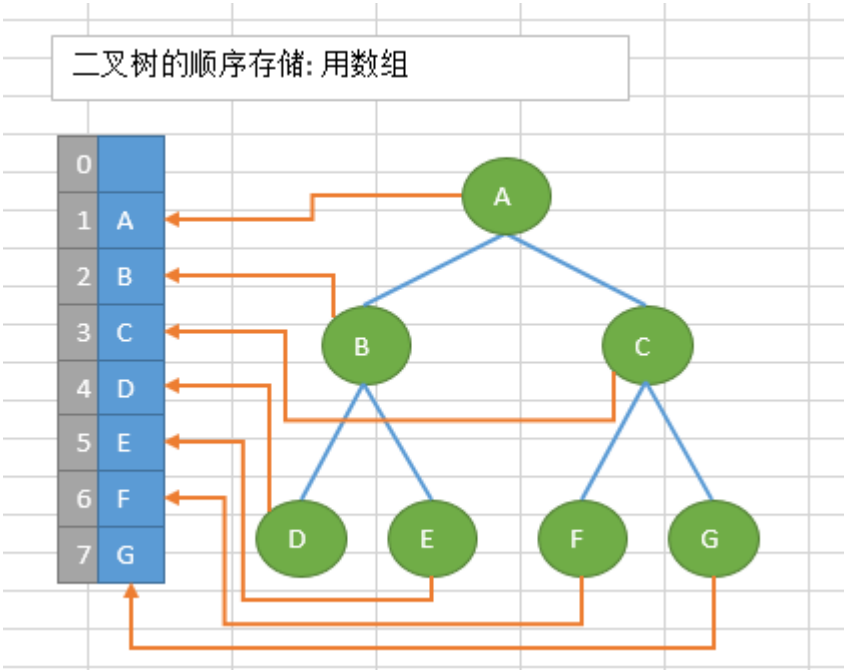
- **二叉树**：每个节点最多含有两个子树的树称为二叉树；

    - **完全二叉树**：对于一颗二叉树，假设其深度为 `h(h>1)`。除了第 `h` 层外，其它各层的节点数目均已达最大值，且第 `h` 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。
    - **满二叉树**：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）；左小右大

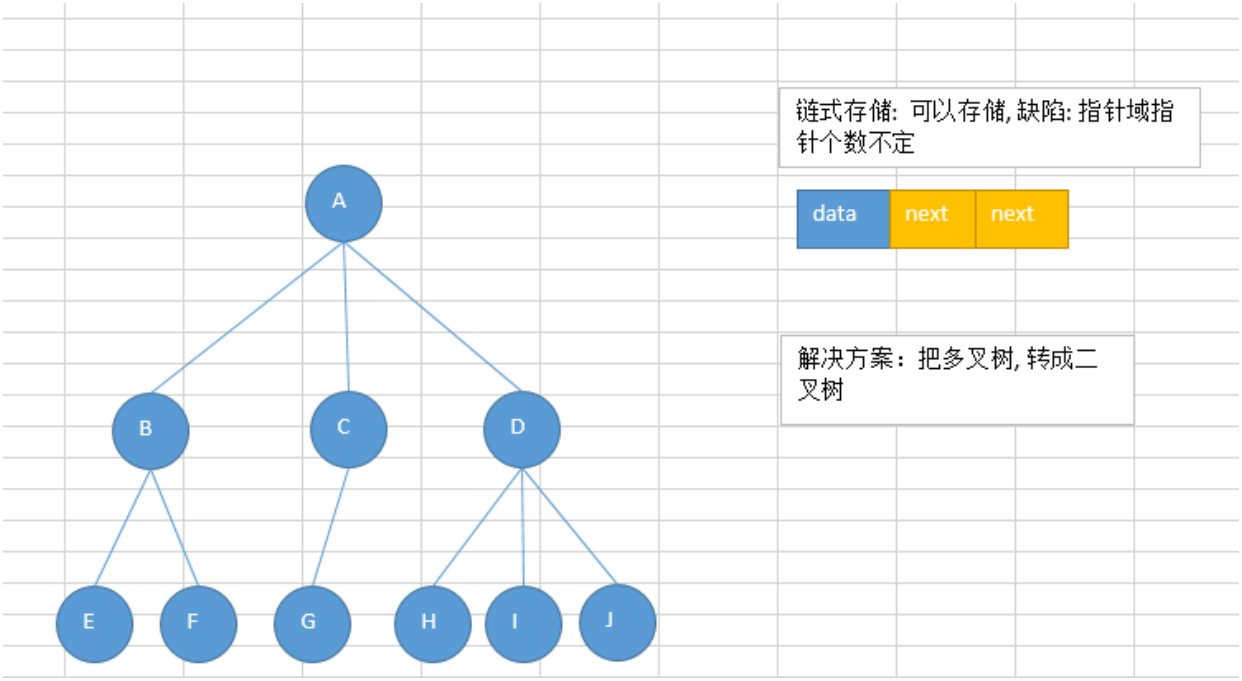
- **霍夫曼树**（用于信息编码）：带权路径最短的二叉树称为哈夫曼树或最优二叉树;
- `B` **树**：一种对读写操作进行优化的自平衡的二叉查找树，能够保持数据有序，拥有多余两个子树。

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。

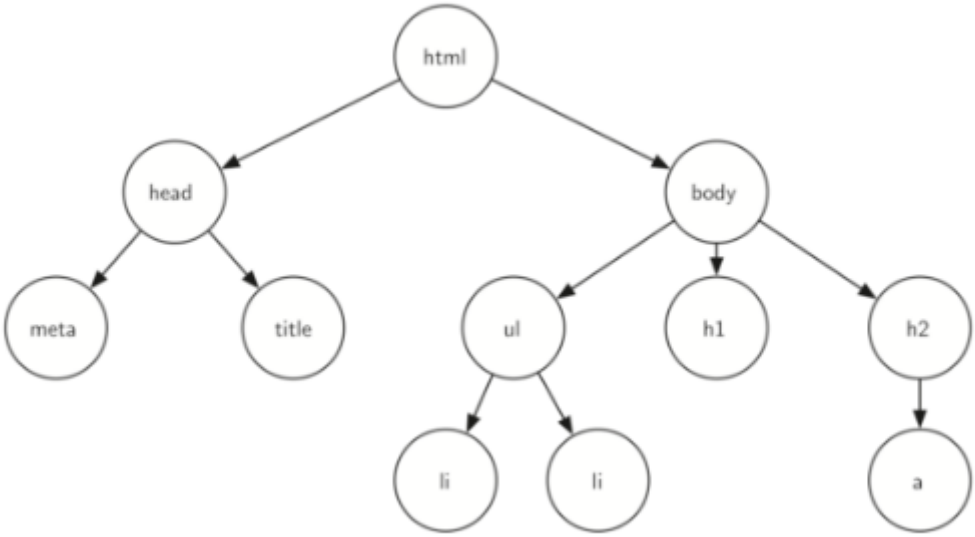


链式存储：



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



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

6.2 二叉树的概念

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）

6.3 二叉树的广度优先遍历

使用队列实现广度优先遍历

6.4 二叉树的实现

```
class Node:
    """节点"""

    def __init__(self, item):
        self.elem = item
        self.lchild = None
        self.rchild = None


class Tree:
    """二叉树"""
```

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

    def add(self, item):
        node = Node(item)
        if self.root is None:
            self.root = node
            return
        queue = [self.root]
        while queue:
            cur_node = queue.pop(0)
            if cur_node.lchild is None:
                cur_node.lchild = node
                return
            else:
                queue.append(cur_node.lchild)

            if cur_node.rchild is None:
                cur_node.rchild = node
                return
            else:
                queue.append(cur_node.rchild)
```

6.5 二叉树的先序、中序、后序遍历

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。

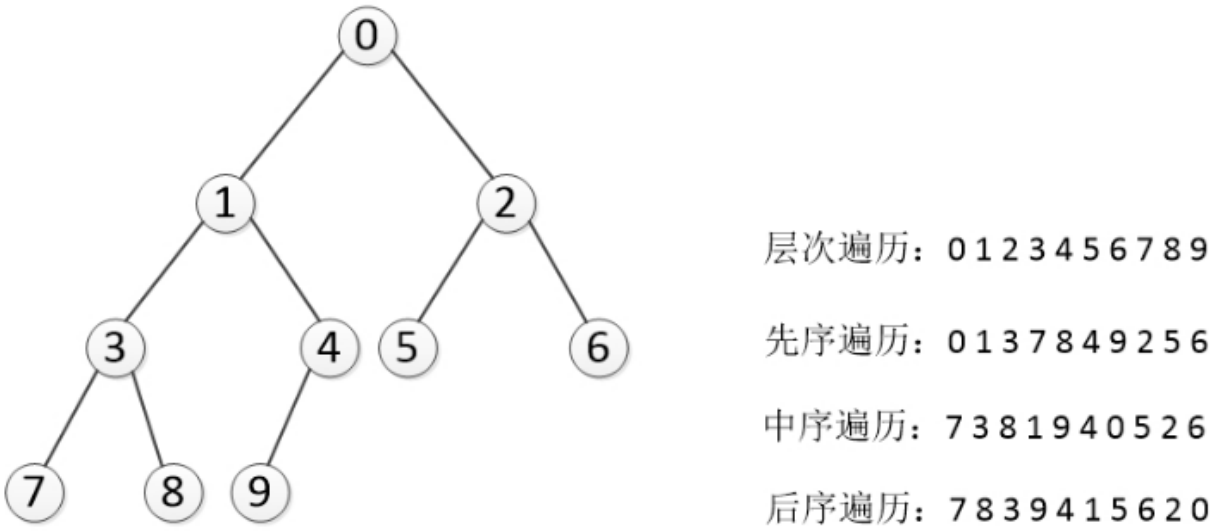
这三种方式常被用于访问树的节点，它们之间的不同在于访问每个节点的次序不同。
这三种遍历分别叫做**先序遍历**（`preorder`），**中序遍历**（`inorder`）和**后序遍历**（`postorder`）。

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`

> 先序指的是先根

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`

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`



> 一旦涉及到递归，代码通常比较简单。

```
class Node:
```

```
    """节点"""

    def __init__(self, item):
        self.elem = item
        self.lchild = None
        self.rchild = None


class Tree:
    """二叉树"""

    def __init__(self):
        self.root = None

    def add(self, item):
        node = Node(item)
        if self.root is None:
            self.root = node
            return
        queue = [self.root]
        while queue:
            cur_node = queue.pop(0)
            if cur_node.lchild is None:
                cur_node.lchild = node
                return
            else:
                queue.append(cur_node.lchild)

            if cur_node.rchild is None:
                cur_node.rchild = node
                return
            else:
                queue.append(cur_node.rchild)

    def breadth_travel(self):
        """广度遍历"""
        if self.root is None:
            return
        queue = [self.root]
        while queue:
            cur_node = queue.pop(0)
```

```
                print(cur_node.elem, end=' ')
                if cur_node.lchild is not None:
                    queue.append(cur_node.lchild)
                if cur_node.rchild is not None:
                    queue.append(cur_node.rchild)

    def preorder(self, node):
        """先序遍历"""
        if node is None:
            return
        print(node.elem, end=' ')
        self.preorder(node.lchild)
        self.preorder(node.rchild)

    def inorder(self, node):
        """中序遍历"""
        if node is None:
            return
        self.inorder(node.lchild)
        print(node.elem, end=' ')
        self.inorder(node.rchild)

    def postorder(self, node):
        """后序遍历"""
        if node is None:
            return
        self.postorder(node.lchild)
        self.postorder(node.rchild)
        print(node.elem, end=' ')


if __name__ == '__main__':
    tree = Tree()
    tree.add(0)
    tree.add(1)
    tree.add(2)
    tree.add(3)
    tree.add(4)
    tree.add(5)
    tree.add(6)
    tree.add(7)
```

```
    tree.add(8)
    tree.add(9)

    # 广度优先遍历
    tree.breadth_travel()
    print()
    print('-' * 20)

    # 深度优先（先序遍历）
    tree.preorder(tree.root)
    print()
    print('-' * 20)

    # 深度优先（中序遍历）
    tree.inorder(tree.root)
    print()
    print('-' * 20)

    # 深度优先（后序遍历）
    tree.postorder(tree.root)
    print()
    print('-' * 20)

"""
0 1 2 3 4 5 6 7 8 9
--------------------
0 1 3 7 8 4 9 2 5 6
--------------------
7 3 8 1 9 4 0 5 2 6
--------------------
7 8 3 9 4 1 5 6 2 0
--------------------
"""
```

6.6 二叉树由遍历确定一棵树

只要给一个中序，加上先序或者后序，就能画出整个树。
给出中序的原因是，这样就可以分开左和右了，否则无法分开左右。

完成于 `2018.12.14 14:55`