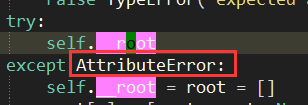
OrderedDict

1.判断一个成员是否存在的一种方法：



2.实现有序dict的思路：

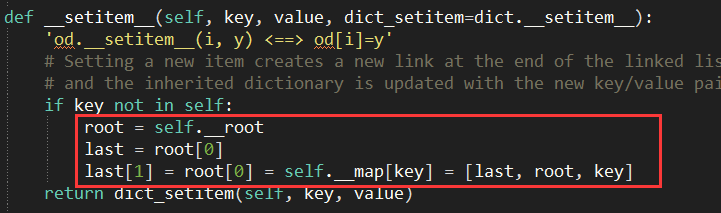
a) key->val 由继承的dict来实现；

b) 建立双向链表来插入key，这样就知道顺序，而且可以插入和删除；

c) 建立一个dict，映射key->链表位置，快速定位双向链表，查找为O（Logn）；

3.插入：

root



a) root = self.\_\_root

获取双向链表

b) last = root[0]

获取双向链表最后一个节点

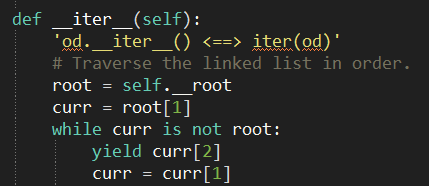
c) self.\_\_map[key] = [last, root, key]

创建一个新的节点，插在链表最后面

d) last[1] = root[0] = self.\_\_map[key]

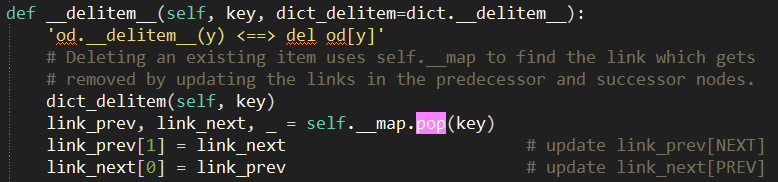
修改之前最后节点的前个位置为新节点；修改根节点之前是新节点。

4.遍历：



简单的链表遍历，没什么好看的

5.删除：



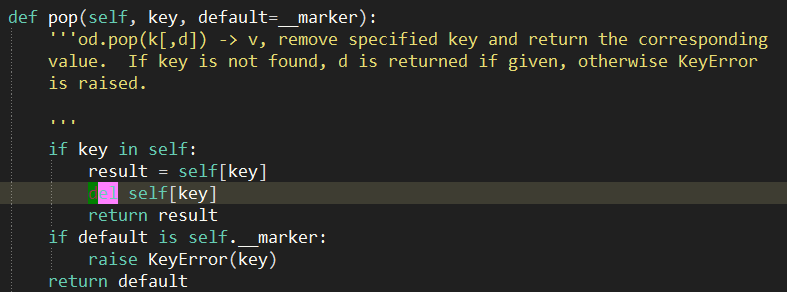
a) 删掉基类的val

b) 删掉链表节点

c) 修改前后节点

\* 就是有这个指定key的删除，所以才需要有一个self.\_\_map

6.pop:



需要注意的是del self[key]，其实调用的是上面介绍的\_\_delitem\_\_

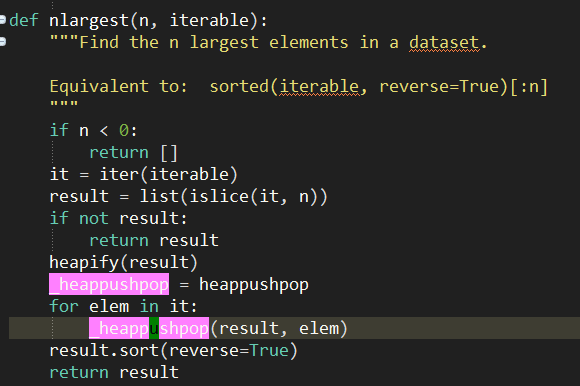
Counter

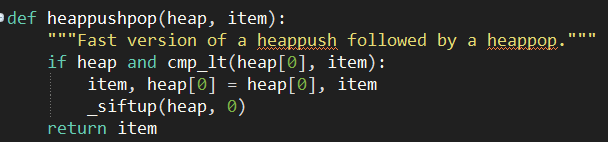
在获取有长度为m的列表中找最大的n个元素的算法：

a) 建立一个大小为n的最小堆；

b) 把剩下的（m-n）个元素执行push和pop的操作；

c) 最后剩下的堆就是结果了。





顺便重学堆算法（突然觉得好简单）：

1.首先，要知道堆算法有两个原则：

a) 完全二叉树

b) 父节点 >= 左 >= 右（最大堆）

2.因此，在操作的时候都要遵守：

a) 插入：

I) 完全二叉树：

因此，插入在最后面，就满足条件啦；

II) 大小关系：

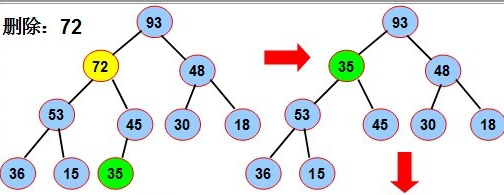
可以很容易就推导出，向上移动，肯定还是满足整体的大小关系的。

b) 删除：

I) 完全二叉树：

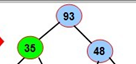
还是从最后一个节点考虑，把要删除的位置用最后节点补上；

II) 大小关系：



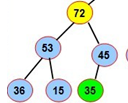
因为调换之后可能出现新节点是在中间的位置，这时可以这么去理解：

A) 上部分一个整体：



因此，按照上面解释插入的原理，如果能保证35这个位置的值是理想的 值，就可以向上调整。

B) 下部分看成一个整体：



可以很简单推导出，如果从根节点向下调整，是可以保证整体有效的。