第二周

1.栈（stack），队列（queue），优先队列(prority-queue)

Stack: 添加删除为O（1）,查询O（n）

Queue: 添加删除为O（1），查询O（n）

Deque: 双端队列 插入和删除都是O(1),查询O(n)

如何查询接口信息（API）empty,peek,pop,push,search

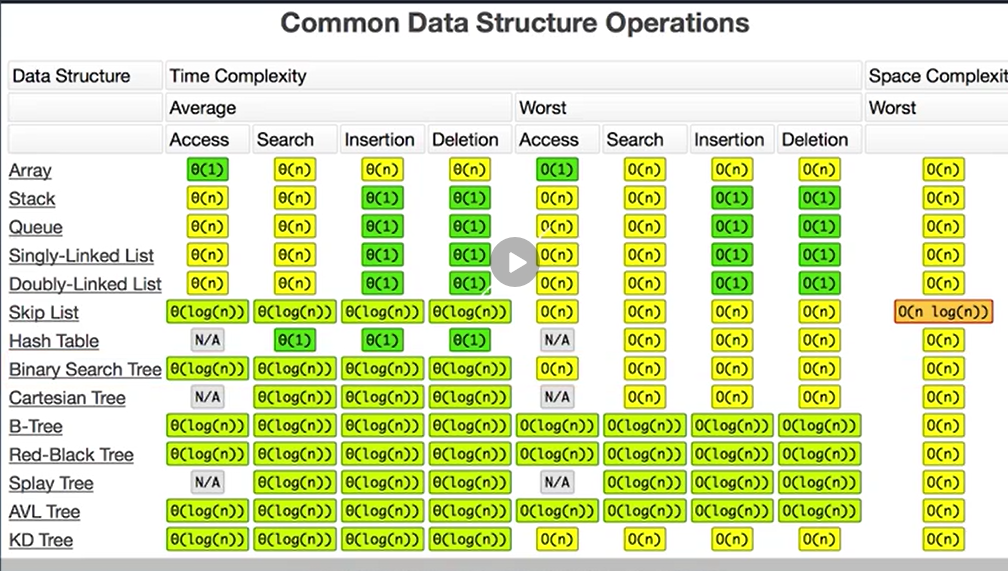
Deque

Priority Queue:按照优先级排序

插入操作：O（1），取出操作：O（logN）

源代码：python souce code download

作业：分析Queue和Priority Queue的源码



练习1：

1. 有效的括号
2. 暴力，不断replace匹配的括号
3. 栈的用法

2.柱状图最大的矩形

1.暴力O（n^3）

2..左边界和右边界O（n^2）

3.栈的方法O（n）

3.滑动窗口的最大值

1.暴力求解O(n\*k)

2.单调队列O（n+k）

2.哈希表，映射和集合（Hash Function）

哈希函数，主要包括（map和set）

哈希碰撞（Hash collison）

增删减除都是O（1）

练习2

242.有效的字母字母异位词

1.暴力.sorted O（nlogn）

2.hashcode，统计每个字符的频次。O(n)

1.两数之和

1.哈希表

3.Prityqueue-heaq queue algorith总结

从源代码可以看出来，PriorityQueue使用的就是heapq来实现的，所以可以认为两者算法本质上是一样的。当然PriorityQueue考虑到了线程安全的问题。Priorityqueue的API主要包括：

\_\_all\_\_ = ['heappush', 'heappop', 'heapify', 'heapreplace', 'merge', 'nlargest', 'nsmallest', 'heappushpop']

def heappush(heap, item):

把这个类放进堆里面，保证这个堆不变

def heappop(heap):

把这个类的最小的值推出堆，保证这个堆不变

def heapreplace(heap, item):

pop出最小的值，并且将新的item放进里面

def heappushpop(heap, item):

将项目推入堆中，然后弹出并从堆中返回最小的项目 。heappush() 与单独调用相比，组合操作的运行效率更高heappop()

def heapify(x):

将列表转变为堆，时间复杂度位为O(len(x))时间

def \_heappop\_max(heap)：  
 pop出最大的值，并且将新的item放进里面

def \_heapreplace\_max(heap, item):

pop出最大值，并且将最新的项放进堆里面

def \_heapify\_max(x):

把列表转换成最大堆，时间复杂度O（len(x)）

def merge(\*iterables, key=None, reverse=False):

将多个排序的输入合并到一个排序的输出中（例如，合并多个日志文件中带有时间戳的条目）。返回 排序后的值的[迭代器](https://docs.python.org/2/glossary.html#term-iterator)。

def nsmallest(n, iterable, key=None):

返回堆中最小的n项

def nlargest(n, iterable, key=None):

返回最大的n项

1. python queue 总结

该queue模块实现了多生产者，多消费者队列。当必须在多个线程之间安全地交换信息时，它在线程编程中特别有用。Queue此模块中的类实现所有必需的锁定语义。该模块实现三种类型的队列，它们的区别仅在于检索条目的顺序不同。在FIFO 队列中，首先检索到添加的第一个任务。在 LIFO队列中，最近添加的条目是第一个检索到的条目（操作类似于堆栈）。使用优先级队列，条目将保持排序（使用heapq模块），并且最先检索值最低的条目。主要包括queue.Queue;queue.LifoQueue;queue.PriorityQueue.

def qsize(self):

返回这个队列的实际大小

def empty(self):

判断是否为空

def full(self):

判断队列是否满了

def put(self, item, block=True, timeout=None):

将项目放入队列

def get(self, block=True, timeout=None):

移除一个队列，并且将此项返回

def put\_nowait(self, item):

将项目放入队列而不会堵塞

def get\_nowait(self):

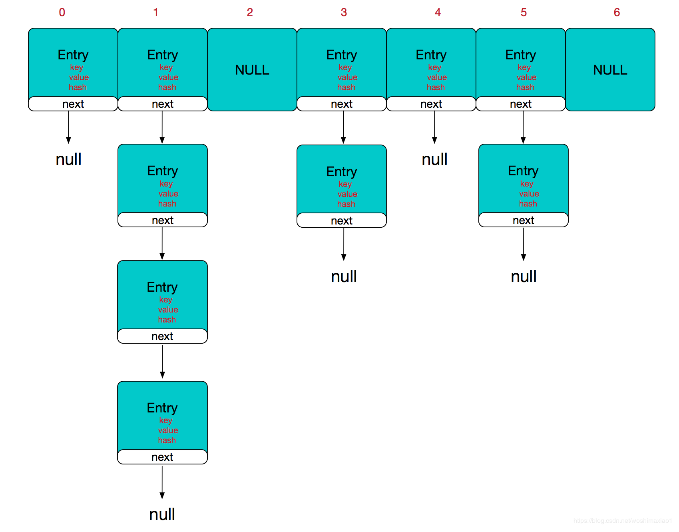
移除并且返回一项而不会堵塞

1. hashmap总结

如果两个不同的元素，通过哈希函数得出的实际存储地址相同怎么办？也就是说，当我们对某个元素进行哈希运算，得到一个存储地址，然后要进行插入的时候，发现已经被其他元素占用了，其实这就是所谓的哈希冲突，也叫哈希碰撞。前面我们提到过，哈希函数的设计至关重要，好的哈希函数会尽可能地保证 计算简单和散列地址分布均匀,但是，我们需要清楚的是，数组是一块连续的固定长度的内存空间，再好的哈希函数也不能保证得到的存储地址绝对不发生冲突。那么哈希冲突如何解决呢？哈希冲突的解决方案有多种:开放定址法（发生冲突，继续寻找下一块未被占用的存储地址），再散列函数法，链地址法，而HashMap即是采用了链地址法，也就是数组+链表的方式

HashMap的主干是一个Entry数组。Entry是HashMap的基本组成单元，每一个Entry包含一个key-value键值对。（其实所谓Map其实就是保存了两个对象之间的映射关系的一种集合）

HashMap的总体结构



简单来说，**HashMap由数组+链表组成的**，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，如果定位到的数组位置不含链表（当前entry的next指向null）,那么查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，**HashMap中的链表出现越少，性能才会越好。**

Hashmap put方法逻辑图

