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

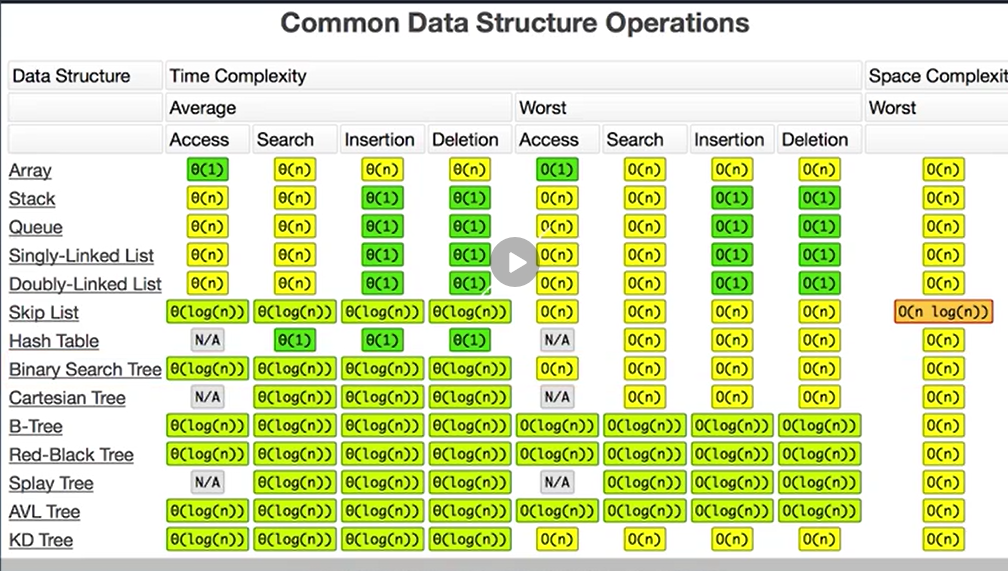
Deque

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):

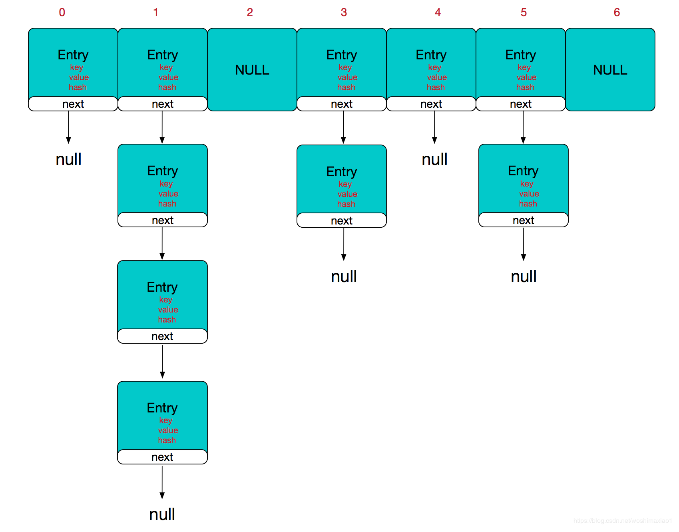
移除并且返回一项而不会堵塞

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

第三周

1. 树

跳表：升维

树和图的差别是有没有环

Python

# Definition for a binary tree node.

# class TreeNode(object):

#     def \_\_init\_\_(self, x):

#         self.val = x

#         self.left = None

#         self.right = None

C++

1. /\*\*
2. \* Definition for a binary tree node.
3. \* struct TreeNode {
4. \*     int val;
5. \*     TreeNode \*left;
6. \*     TreeNode \*right;
7. \*     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
8. \* };
9. \*/
10. JAVA
11. /\*\*
12. \* Definition for a binary tree node.
13. \* public class TreeNode {
14. \*     int val;
15. \*     TreeNode left;
16. \*     TreeNode right;
17. \*     TreeNode(int x) { val = x; }
18. \* }
19. \*/

2.二叉树搜索树（二叉排序树，有序二叉树）

1. 左子树上所有结点的值均小于它的根节点
2. 右子树所有结点的值均大于它的根节点的值
3. 左右子树也是二叉查找树

中序遍历;升序排列

1. 查询 log(n)
2. 插入新结点（logn）
3. 删除（logn）

思考：树的面试题为什么一般是递归

利用递归的方法来执行树方面的操作，没有容易循环的结构，是因为树的特性决定的，运用递归的方法更加简单。

3.堆

Heap:可以迅速找到一堆数中的最大或者最小值的数据结构。

将根节点最大的堆叫大顶堆或者大根堆，根节点最小的堆叫小顶堆或者小根堆

常见的堆有二叉堆，斐波那契堆

假设是大顶堆，则常见操作（API）：

Find\_max O（1）

Delete-max O(logN)

Insert(create):O(logN)orO(1)

二叉堆性质：

通过完全二叉树实现

二叉堆它满足下列性质：

是一棵完全树

树中任意节点的值大于等于其子节点

怎么实现：

1. 通过数组实现，2.假设第一个元素为0，左孩子索引2\*i+，右孩子索引2\*i+2

父节点floor（（i-1）/2）

插入元素O（logn）：1.新元素插到尾部2.依次向上调整这个堆的结构O（logn）

删除元素：1.将堆尾元素替换到顶部2.依次从根部向下调整整个结构

二叉堆不是最优的实现（优先队列）

Python的大顶堆怎么实现：

1. 图

图的属性和分类

有点有边

表示方法：邻接矩阵和邻接表

无向无权图和有向无权图和有向有权图

算法：DFS和BFS

DFS：visited=set()#和树中的DFS的区别

BFS：visited

Def BFS(graph,start,end):

Queue=[]

Queue.append([start])

Visited=set()

While queue:

Node=queue.pop()

Visited.add(node)

Process(node)

Nodes=generate\_related\_nodes(node)

Queue.push(nodes)

Visited=set()

Def dfs(node,visited):

If node in visited:

Return

Visited.add(node)

For next\_node in node.children():

If not next\_node in visited:

Dfs(next\_node,visited)

1. 递归

代码模板：利用递归的方法：找到截止条件；当前层逻辑；到下一层

**Python 代码模板**

**def** **recursion**(level, param1, param2, ...):

# recursion terminator

**if** level > MAX\_LEVEL:

process\_result

**return**

# process logic in current level

process(level, data...)

# drill down

**self**.recursion(level + 1, p1, ...)

# reverse the current level status if needed

1. 分治，回溯：找到子问题然后组合

1.递归状态树

Python 分治代码模板

终止条件；当前条件；drilldown,merege;reverse states

# recursion terminator

if problem is None:

print\_result

return

# prepare data

data = prepare\_data(problem)

**subproblems** = split\_problem(problem, data)

# conquer **subproblems**

**subresult1** = **self.divide\_conquer(subproblems[0],** p1, ...)

**subresult2** = **self.divide\_conquer(subproblems[1],** p1, ...)

**subresult3** = **self.divide\_conquer(subproblems[2],** p1, ...)

…

# process **and** generate the final result

result = process\_result(**subresult1, subresult2, subresult3,** …)

# **revert** the current level states