HashHeap 和普通heap区别：是否需要remove操作

普通heap可以在O（logn）时间复杂度内插入，弹出极值元素

但是不能在这样的时间复杂度内删除任意元素。所以在维护堆的同时维护一个hashmap，

hash 记录的是每个值在堆里面的位置。

hash中key是堆内元素的值，value 是该值在堆的位置。

所以可以在O（1）时间复杂度内定位到堆中待删除元素

为了处理相同数字的情况，堆中每个节点要维护count值

- [x] ~~trap rain water 见九章算法强化1.a~~
- [x] ~~trap rain waterii 见九章算法强化1.a~~
- [ ] **Building Outline**

两个对象也可以通过a,b=b,a这样进行交换

TreeMap里面的实现是平衡二叉树：

avl

splay

红黑树

Sliding Window Median

基本思想

mid的左边是一个大顶堆，mid的右边是一个小顶堆。

```
class HashHeap:

    def __init__(self):
        self.heap = []
        self.hash = {}
        self.totals = 0

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

    def empty(self):
        return True if self.size() == 0 else False

    def top(self):
        return self.heap[0][0]

    def add(self, item):
        self.totals += 1
        if item in self.hash:
            pos = self.hash.get(item)
            self.heap[pos][1] += 1
            return
        self.heap.append([item, 1])
        self.hash[item] = self.size() - 1
        self._siftup(self.size() - 1)

    def pop(self):
        assert self.size() > 0
        self.totals -= 1
        if self.heap[0][1] > 1:
```

```
            self.heap[0][1] -= 1
            return self.heap[0][0]
        self._swap(0, self.size() - 1)
        del self.hash[self.heap[-1][0]]
        p = self.heap.pop(-1)
        self._siftdown(0)
        return p[0]

    def remove(self, value):
        pos = self.hash.get(value)

        if pos != -1:
            self.totals -= 1
            if self.heap[pos][1] > 1:
                self.heap[pos][1] -= 1
                return
            self._swap(pos, self.size() - 1)
            del self.hash[self.heap[-1][0]]
            self.heap.pop(-1)
            if pos >= self.size():
                return
            if pos == 0:
                self._siftdown(pos)
            else:
                father = (pos - 1) // 2
                if self.heap[pos] < self.heap[father]:
                    self._siftup(pos)
                else:
                    self._siftdown(pos)

    def _swap(self, i1, i2):
        self.heap[i1], self.heap[i2] = self.heap[i2], self.heap[i1]
        self.hash[self.heap[i2][0]] = i2
        self.hash[self.heap[i1][0]] = i1

    def _siftup(self, index):
        while index > 0:
            father = (index - 1) // 2
            if self.heap[father] < self.heap[index]:
                break
            self._swap(father, index)
            index = father

    def _siftdown(self, index):
        while True:
            lchild, rchild = index * 2 + 1, index * 2 + 2
            if lchild >= self.size():
                return
            minimum = lchild
            if rchild < self.size():
                minimum = lchild if self.heap[lchild] < self.heap[rchild] else rchild
            if self.heap[minimum] > self.heap[index]:
```

```
                return
            self._swap(minimum, index)
            index = minimum


class Solution:
    """
    @param nums: A list of integers
    @param k: An integer
    @return: The median of the element inside the window at each moving
    """

    def medianSlidingWindow(self, nums, k):
        # write your code here
        if not nums or k > len(nums):
            return []
        if k == 1:
            return nums
        leftMaxHeap = HashHeap()
        rightMinHeap = HashHeap()
        ans = []
        mid = nums[0]

        def addTo(i, mid):
            if nums[i] < mid:
                leftMaxHeap.add(-nums[i])
                if leftMaxHeap.totals > rightMinHeap.totals:
                    rightMinHeap.add(mid)
                    mid = -leftMaxHeap.pop()
            else:
                rightMinHeap.add(nums[i])
                if rightMinHeap.totals > leftMaxHeap.totals + 1:
                    leftMaxHeap.add(-mid)
                    mid = rightMinHeap.pop()
            return mid

        def removeFrom(i, mid):
            if nums[i] == mid:
                if rightMinHeap.totals == leftMaxHeap.totals:
                    mid = -leftMaxHeap.pop()
                else:
                    mid = rightMinHeap.pop()
            elif -nums[i] in leftMaxHeap.hash:
                leftMaxHeap.remove(-nums[i])
                if leftMaxHeap.totals < rightMinHeap.totals - 1:
                    leftMaxHeap.add(-mid)
                    mid = rightMinHeap.pop()
            elif nums[i] in rightMinHeap.hash:
                rightMinHeap.remove(nums[i])
                if rightMinHeap.totals < leftMaxHeap.totals:
                    rightMinHeap.add(mid)
                    mid = -leftMaxHeap.pop()
```

```
            return mid

        for i in range(1, k):
            mid = addTo(i, mid)
        ans.append(mid)
        for i in range(k, len(nums)):
            mid = removeFrom(i - k, mid)
            mid = addTo(i, mid)
            ans.append(mid)
        return ans


```

Sliding Window Maximum

递减的单调栈

在左边删除，在右边加入，在左边取元素

dq存放下标

当即将进来的元素比最上面元素大，删掉最上面元素（dq始终保持下标所指的数字递减存放）

起始放k-1个，主循环里面要先加入一个元素，然后放入答案，然后再删除一个元素

```
def maxSlidingWindow(nums, k):
    # write your code here
        if not nums or k > len(nums):
            return []
        if k == 1:
            return nums
        ans = []
        def push(dq, nums, i):
            while dq and nums[dq[-1]] < nums[i]:
                dq.pop()
            dq.append(i)
        from collections import deque
        dq = deque()
        for i in range(k-1):
            push(dq, nums, i)
        for i in range(k-1, len(nums)):
            push(dq,nums,i)
            ans.append(nums[dq[0]])
            if dq[0] == i-k+1:
                dq.popleft()
        return ans
```