一.堆的基本介绍

1）为什么我们需要堆

数据结构的选择与算法实现息息相关，比如要按升序输出一系列数值，利用数组实现或利用二叉搜索树实现其复杂度和算法实现完全不同，数据结构也直接影响着算法复杂度。

在操作系统的CPU调度中，经常涉及在就绪队列中增加一个作业，以及从就绪队列中选择一个作业占用CPU；之前在某个汉服体验馆，一直有新客户到来，商家会根据会员等级选择挑选汉服的客户。由此可见，不管是生活中还是计算机系统的实现都离不开两个重要的操作：**插入**，**找出最大值**（也可以理解为权重最高，按值排序其实就是将值作为权重）

先考虑一下我们之前常用的数据结构，在线性表中，查找最大元素要遍历整个表，在排好序的线性表中，为保证插入元素不破坏线性表的属性，需要遍历表使元素插入到合适的位置，优先队列可以兼顾到插入和找最值，堆就是一种优先队列，接下来我们会介绍在堆上这些操作的算法复杂度。

2）什么是堆

- 堆是一个**完全二叉树**，并且满足堆的性质
- 堆的性质：
  - 每一个父节点的key都大于等于子节点的key（大根堆）
  - 或者：每一个父节点的key都小于等于子节点的key（小根堆）
  - 递归定义：递归的思想对于树非常重要
    - 父节点的key大于等于子节点的key，并且以子节点为根节点的树也是大根堆。

以下讨论以大根堆为例。

3）堆的表示

根据完全二叉树的性质，T的根节点存储在H[1]中。
假设T的节点x存储在H[j]中，如果它有左子节点，这个子节点存储在H[2j]中；如果它也有右子节点，这个子节点存储在 H[2j+1]中
元素H[j]的父节点存储在H[j/2]（下取整）中

4）堆的特性

沿着每条从根到叶子的路径，元素的键值以非升序排列。

5）运算

堆的基本运算实现思想是解决与堆有关算法题的关键

在每种基本操作后附带了基于c++STL的实现

①两个基本操作

这两个基本操作是堆的运算实现的基础

😈sift-up

假定对于某个i>1, H[i]变成了键值大于它父节点键值的元素，这样就违反了堆的特性，因此这种数据结构就不再是堆了。如要修复堆的特性，需要用称为Sift-up的运算把新的数据项上移到在二叉树中适合它的位置上，这样堆的属性就修复了。
sift-up 运算沿着从H[i]到根节点的惟一一条路径，把H[i]移到适合它的位置上。　在沿着路径的每一步上，都将H[i]键值和它父节点的键值H[i/2]相比较。

```
void siftup(vector<int> &heap,int i){
    if(i==1) return;
    while(i>1&&heap[i]>=heap[i/2]){
        swap(heap[i], heap[i/2]);
        i = i/2;
    }
}
```

😈sift-down

假定对于i<=n/2,，存储在H[i]中元素的键值变成小于H[2i]和H[2i+1]中的最大值(如果H[2i+1]存在的话)，这样就违反了堆的特性，树就不再表示一个堆。
如要修复堆的特性，需要用Sift-down运算使H[i]“渗”到二叉树中适合它的位置上，沿着这条路径的每一步，都把H[i]的键值和存储在它子节点(如果存在)中的两个键值里最大的那个相比较

```
void siftdown(vector<int> &heap,int i){
    unsigned int n = unsigned(heap.size()-1);
    int maxindex;
    while(2*i<=n){
        if(2*i==n) maxindex = 2*i;
        else maxindex = heap[2*i]>heap[2*i+1]?2*i:2*i+1;

        if(heap[maxindex]>=heap[i]) {
            swap(heap[maxindex],heap[i]);
            i = maxindex;
        }
        else return;
    }
}
```

②插入

插入操作基于sift-up操作，可以先将元素插入到堆的尾部，然后进行上移操作进行调整。

```
void insertHeap(vector<int> &heap,int a){
    heap.push_back(a);
    siftup(heap,int(heap.size())-1);
}
// STL
vector.push_back(11);
push_heap(vector.begin()+1,vector.end());
```

时间复杂度=O(logn)

③删除

删除操作是基于sift-up的sift-down操作，先将待删除的元素i和最后一个元素n进行互换，删除最后一个元素n，然后对i位置的元素进行sift-up或sift-down操作

```
void deleteHeap(vector<int> &heap,int i){
    swap(heap[i],heap[heap.size()-1]);
    heap.pop_back();
    if(i!=1&&heap[i]>=heap[i/2]) siftup(heap, i);
    else siftdown(heap, i);
}
```

时间复杂度=O(logn)

④删除最大元素

删除最大元素相当于调用`deleteHeap(heap,1)`

```
void deleteHeapMax(vector<int> &heap){
    deleteHeap(heap,1);
}
// STL
pop_heap(vector.begin()+1,vector.end());
vector.pop_back();
// 将最大元素移到末尾，需要自行删除
```

时间复杂度=O(logn)

⑤创建堆

创建堆的思想同样基于sift-up和sift-down两个基本操作

问题描述：给定一个数组，构造一个堆

😈思想一：从一个空堆开始，不断插入数组中的元素

```
vector<int> makeHeap1(int a[],int n){
    vector<int> heap;
    heap.push_back(0);  // 也可以不占用heap[0]的位置，但是这样便于角标计算
    for(int i=0;i<n;i++){
        insertHeap(heap, a[i]);
    }
    return heap;
}
```

> ❓这里有一个疑问：这种构造方法的时间复杂度是多少?
>
> 最开始我的想法是对于第k个元素，插入操作的时间复杂度是`O(logk)`，那求和后为

$$O(log(\frac{n(n+1)}{2})) = O(logn + log(n+1) - log2)$$

那时间复杂度不就是O(logn)吗？这里犯了一个错误，就是log(n+1)不能归于log(n)，正确做法应该对上述和式求上界和下界。对于logk求和的推导如下：

$$\begin{aligned} 1.\sum_{k=1}^{n} logk &= logn + \sum_{k=1}^{n-1} logk \\ &<= logn + \int_{1}^{n} logxdx \\ &= logn + nlogn - nloge + loge \end{aligned}$$

$$\begin{aligned} 2.\sum_{k=1}^{n} logk &= \sum_{k=2}^{n} logk \\ &>= \int_{1}^{n} logxdx \\ &= nlogn - nloge + loge \end{aligned}$$

综上，时间复杂度为O(nlogn).

😈将数组看做一个未调整好的堆

这里用到了树的递归思想，如果一个结点的左子树和右子树都是堆，那么只需要对该节点进行sift-down调整即可得到堆。

n为最后一个结点，那么n/2为最后一个有孩子的父节点，在`n/2+1`至`n`之间的结点就都是叶子结点，每个叶子结点可以看作是一个堆，从结点`n/2`开始，其左子树和右子树都是堆，将该结点看做一个根节点，进行sift-down操作，依次对`n/2-1`，`n/2-2`结点进行调整，即可得到一个大根堆。

```
void makeHeap2(vector<int> &heap){
    int n = int(heap.size())-1;
    for(int j=n/2;j>=1;j--){
        siftdown(heap, j);
    }
}
```

❓算法分析：

树高H=logn，对于第i层的元素，移动次数最多为H-i，第i层需要移动的点最多有2^i个，移动总次数的上界为

$$\sum_{i=0}^{H-1}(H-i)2^i = \sum_{m=1}^{H} m2^{H-m} = 2^H \sum_{m=1}^{H} m2^{-m} <= 2n$$

在每次循环中最多有两次比较，所以元素比较次数上界为$4n$

当数组本身就是大根堆时，进入$sift-down$函数进行两次比较（至少一次循环），最小比较次数为$2*n/2=n$

时间复杂度O(n),空间复杂度O(1)

😈STL(包含头文件algorithm)

```
make_heap(vector.begin()+1,vector.end());  // 大根堆
make_heap(heap.begin()+1, heap.end(), greater<>());  // 小根堆
```

这里的vector[0]是无用的空间，使用+1是便于角标计算

函数原型

```
make_heap(<#_RandomAccessIterator __first#>, <#_RandomAccessIterator __last#>,
<#_Compare __comp#>)
```

cmp默认为`>`

⑥堆排序

先将待排序数组变成堆，然后对最大值元素与最后一个元素互换并进行sift-down操作

```
// STL
sort_heap(vector.begin()+1,vector.end());
```

注意：使用相关函数时第三个参数要保持一致

二.习题

1.数组中第k个最大元素

https://leetcode-cn.com/problems/kth-largest-element-in-an-array/

- 思路一：排序
  - 快排：从大到小排序，当遇到对应下标为k-1的元素时停止递归

```
class Solution {
public:
    bool flag=true;
    int value;
    int index;
    int findKthLargest(vector<int>& nums, int k) {
       // 快速排序
       index=k;
        quickSort(nums,0,nums.size()-1);
        return value;
    }
    void quickSort(vector<int> &nums,int begin,int end){
        if(flag&&begin<=end){
            int temp=nums[begin];
            int i=begin;
            int j=end;
```

```
                while(i<j){
                    while(nums[j]<=temp&&j>i) j--;
                    nums[i]=nums[j];
                    while(nums[i]>temp&&i<j) i++;
                    nums[j]=nums[i];
                }
                nums[j] = temp;
                if(j==index-1){
                    flag=false;
                    value=nums[j];
                    return;
                }
                quickSort(nums,begin,j-1);
                quickSort(nums,j+1,end);
            }
        }
};
```

执行结果： 通过 显示详情 › 添加

执行用时： 44 ms ，在所有 C++ 提交中击败了 22.89% 的用户

内存消耗： 9.8 MB ，在所有 C++ 提交中击败了 40.56% 的用户

通过测试用例： 32 / 32

看了下题解，可以再优化一下，排到的位置与k比较，比k大的话就只对左边部分递归，比k小的话就只对右边部分递归

- 堆排序：当移动到第k项时停止排序，思想和选择排序有点相近，但是时间复杂度更低

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        // 创建大根堆
        make_heap(nums.begin(),nums.end());
        // 执行k-1次下移操作
        for(int i=0;i<k-1;i++){
            pop_heap(nums.begin(),nums.end()-i);
        }
        return nums[0];
    }
};
```

执行结果： 通过 显示详情 › 添加备注

执行用时： 0 ms ，在所有 C++ 提交中击败了 100.00% 的用户

内存消耗： 9.7 MB ，在所有 C++ 提交中击败了 88.17% 的用户

通过测试用例： 32 / 32

2.合并k个升序链表

https://leetcode-cn.com/problems/merge-k-sorted-lists/

基本思路：归并排序的思想，但由于是链表，无需给格外的空间。将结点都插入第一个元素最小的链表，给该链表建立一个遍历的指针，每次将最小的数值插入该指针之后的位置，数组存放其余链表下一个待比较的结点。（因为除了要输入的链表，其余链表是否保留原状无意义）

进一步思考：在找n个元素的最小值时，指针会指向找到的最小值，然后该最小值所在链表的遍历指针向后移1，其实不需要全部重新比较，只需要比较这个新元素和其余n-1个元素的大小即可，这样可以避免一直遍历整个序列，这就让我们想到了堆，需要找最小值，并且会持续给排序好的序列插入少量新元素。

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
 struct Node{
     int val;
     int index;
 };
bool cmp(Node a,Node b){
        return a.val>b.val;
}
class Solution {
public:
    // bool cmp(Node a,Node b){
    //     return a.val<b.val;
    // }
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        vector<Node> heap;
        ListNode* ptr=nullptr;
        ListNode* temp=nullptr;
        int minIndex;
        for(int i=0;i<lists.size();i++){
```

```
            if(lists[i]!=nullptr){
                Node temp={lists[i]->val,i};
                heap.push_back(temp);
            }
        }
        if(heap.size()==0) return nullptr;
        // 建立小根堆
        make_heap(heap.begin(),heap.end(),cmp);
        // 1-先确定待插入的链表
        ptr=lists[heap[0].index];
        minIndex=heap[0].index;
        // 2-删除元素
        pop_heap(heap.begin(),heap.end(),cmp);
        heap.pop_back();
        if(ptr->next!=nullptr){
            Node temp={ptr->next->val,minIndex};
            heap.push_back(temp);
            push_heap(heap.begin(),heap.end(),cmp);
        }
        while(heap.size()!=0){
            int tempindex=heap[0].index;
            if(tempindex==minIndex){
                temp=ptr->next;
            }else{
                temp=lists[tempindex];
                lists[tempindex]=temp->next;
                temp->next=ptr->next;
                ptr->next=temp;
            }
            if(ptr->next!=nullptr) ptr=ptr->next;
            pop_heap(heap.begin(),heap.end(),cmp);
            heap.pop_back();
            if(tempindex!=minIndex&&lists[tempindex]!=nullptr){
                Node temp={lists[tempindex]->val,tempindex};
                heap.push_back(temp);
                push_heap(heap.begin(),heap.end(),cmp);
            }else if(tempindex==minIndex&&ptr->next!=nullptr){
                Node temp={ptr->next->val,minIndex};
                heap.push_back(temp);
                push_heap(heap.begin(),heap.end(),cmp);
            }
        }
        return lists[minIndex];
    }
};
```

思路没错，代码不太优美，得考虑很多特殊情况

执行结果： 通过 显示详情 › 添加备注

执行用时： 24 ms ，在所有 C++ 提交中击败了 61.28% 的用户

内存消耗： 13 MB ，在所有 C++ 提交中击败了 60.16% 的用户

通过测试用例： 133 / 133

3.滑动窗口最大值问题

https://leetcode-cn.com/problems/sliding-window-maximum/

思路与上一题思路很接近，每次减少一个，再增加一个，使用大根堆很好实现，每次移动时将最大的元素保存。

增加的时候很好增加，但是删除的时候比较困难，可以先不删除，如果最大值不在滑动窗口范围内，就删除最大值，直到最大值在滑动窗口范围内。

```
struct Node{
    int val;
    int index;
};
bool cmp(Node a,Node b){
        return a.val<b.val;
}
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> ans;
        vector<Node> heap;
        for(int i=0;i<k;i++){
            Node temp={nums[i],i};
            heap.push_back(temp);
        }
        make_heap(heap.begin(),heap.end(),cmp);
        ans.push_back(heap[0].val);
        for(int i=k;i<nums.size();i++){
            Node temp={nums[i],i};
            heap.push_back(temp);
            push_heap(heap.begin(),heap.end(),cmp);
            while(heap[0].index<i-k+1){
                pop_heap(heap.begin(),heap.end(),cmp);
                heap.pop_back();
            }
            ans.push_back(heap[0].val);
        }
        return ans;
    }
};
```

4.数据流中第K大元素

https://leetcode-cn.com/problems/kth-largest-element-in-a-stream/

思路和1.一致，这里就很明显要用堆了，动态插入新元素以及求最大值。

如果每次插入元素后重构堆，再仿照之前的思路求第k大元素，有些浪费前一次已经做过的调整，基于一个第k大元素已经在根部的堆（见1.）其实插入的元素只需要和A[n-k+2]元素进行比较，如果比A[n-k+2]小，直接插入到vector[n-k+2]的位置然后调整堆，那不是会覆盖了原数据吗，可以将暂时最大的前k-1个元素单独存储在一个vector中；如果比A[n-k+2]大，则说明该元素应该属于前k-1个元素的vector中，挤掉这k个元素中最小的值，因此可以将前k-1个元素建立小根堆（起始时这k-1个元素是升序序列，本身就是一个小根堆），移除根部元素，然后将根部元素插回原数组。(m代表add操作次数)

时间复杂度O(n+(k+m)logn+mlogk)

空间复杂度O(k)

写到这里的时候突然想到，剩下的n-k的元素实际上没有存在的意义了，第k大的元素只可能是原数组根部或者新插入的元素，插入元素时直接和原数组堆根部进行比较，如果比其小，直接舍弃，输出根部键值，如果比其大，再插入k-1个元素的堆，堆中退下来的最小值直接赋给原数组堆的根部，这样优化时间复杂度为O(n+klogn+mlogk)

```
class KthLargest {
public:
    vector<int> kth;
    vector<int> mynums;
    int myk;
    bool firstInsert=true;
    KthLargest(int k, vector<int>& nums) {
        make_heap(nums.begin(),nums.end());
        // 找到第k大元素
        for(int i=0;i<k-1;i++){
            pop_heap(nums.begin(),nums.end()-i);
        }
        for(int i=k-2;i>=0;i--){
            kth.push_back(nums[nums.size()-1-i]);
        }
        mynums=nums;
        myk=k;
    }

    int add(int val) {
        if(mynums.size()==0){
            mynums.push_back(val);
        }else if(val>mynums[0]){
            kth.push_back(val);
            push_heap(kth.begin(),kth.end(),greater<>());
            pop_heap(kth.begin(),kth.end(),greater<>());
            mynums[0]=kth[myk-1];
            kth.pop_back();
        }else if(firstInsert&&val<=mynums[0]&&mynums.size()<myk){
            mynums[0]=val;
```

```
        }
        firstInsert=false;
        return mynums[0];
    }
};

/**
 * Your KthLargest object will be instantiated and called as such:
 * KthLargest* obj = new KthLargest(k, nums);
 * int param_1 = obj->add(val);
 */
```

优化了一下代码：(只维护kth堆)

```
class KthLargest {
public:
    vector<int> kth;
    int myk;
    KthLargest(int k, vector<int>& nums) {
        make_heap(nums.begin(),nums.end());
        // 找到第k大元素
        int i=0;
        while(i<k&&i<nums.size()) {
            pop_heap(nums.begin(),nums.end()-i);
            i++;
        }
        i=k-1;
        while(i>=0){
            if(i>=nums.size()){
                i--;
                continue;
            }
            kth.push_back(nums[nums.size()-1-i]);
            i--;
        }
        myk=k;
    }

    int add(int val) {
        if(kth.size()==myk-1||val>kth[0]){
            kth.push_back(val);
            push_heap(kth.begin(),kth.end(),greater<>());
            if(kth.size()>myk){
                pop_heap(kth.begin(),kth.end(),greater<>());
                kth.pop_back();
            }
        }
        return kth[0];
    }
```

```
};

/**
 * Your KthLargest object will be instantiated and called as such:
 * KthLargest* obj = new KthLargest(k, nums);
 * int param_1 = obj->add(val);
 */
```