合并果子

题目描述

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过n-1次合并之后，就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。

例如有3种果子，数目依次为1，2，9。可以先将 1、2堆合并，新堆数目为3，耗费体力为3。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为12，耗费体力为 12。所以多多总共耗费体力=3+12=15。可以证明15为最小的体力耗费值。

输入

第一行是一个整数n（1 <= n <= 10000），表示果子的种类数。

第二行包含n个整数，用空格分隔，第i个整数a_i（1 <= ai <= 20000）是第i种果子的数目。

输出

只包含一个整数，也就是最小的体力耗费值。输入数据保证这个值小于2^{31}。

分析

该如何选择每次合并的果子?

贪心 (每次找到最小的两个合并)

时间复杂度: 合并n次;每次找最小O(n)

$O(n^2)$ **n<=10000**

超时

什么地方还能优化?

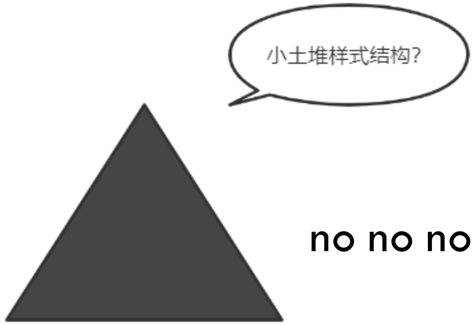
总共合并**n**次不能改变;
找最小值能够优化到**O(logn)**?

堆

堆(HEAP)

QYC

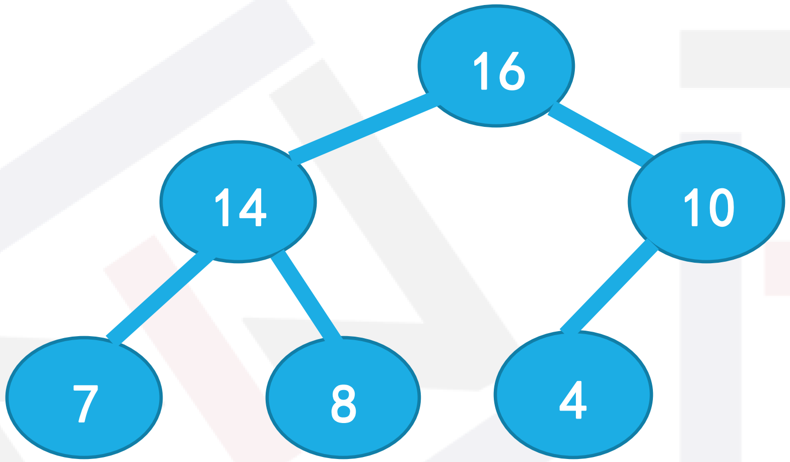
什么是堆?



- 堆其实是一棵**完全二叉树**。
- 堆中某个节点的值总是**不大于或不小于**其**父节点**的值;
  - **大根堆:** 所有结点的值不大于父亲结点。最大值为根结点。
  - **小根堆:** 所有结点的值不小于父亲结点。最小值为根结点。

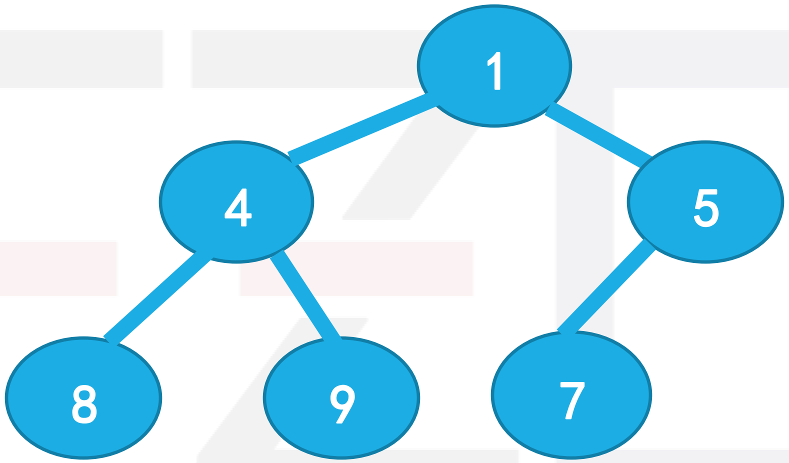
堆

最大值



大根堆

最小值



小根堆

如何存储?

由于是一棵完全二叉树

数组 a[N]

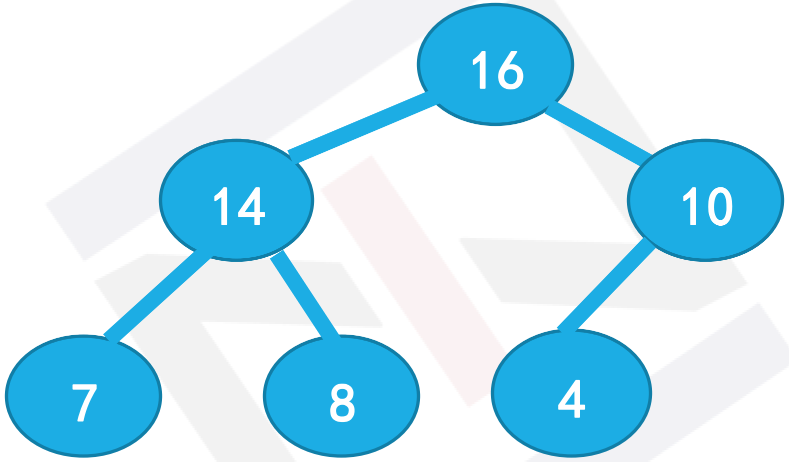
已知结点：i

父亲结点：i/2

左孩子：2*i

右孩子：2*i+1

堆的数组存储



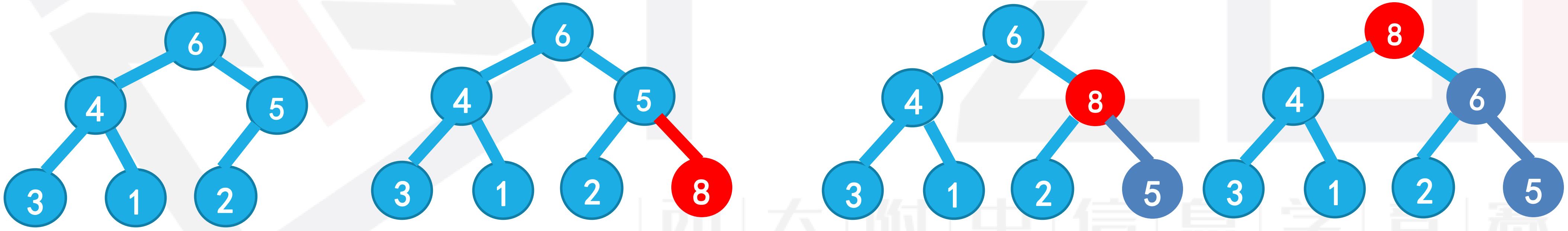
大根堆

1	2	3	4	5	6
16	14	10	7	8	4

堆的操作-添加

put操作(添加元素)

往大顶堆添加元素8



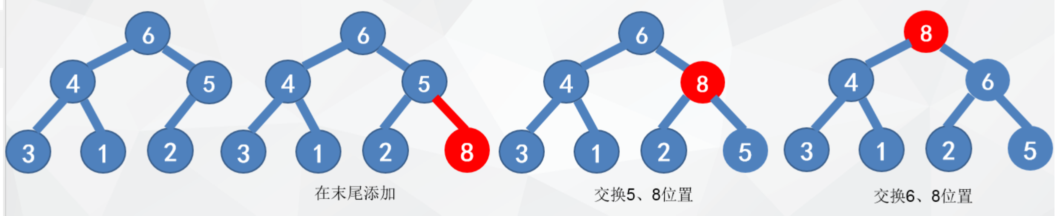
在末尾添加　　交换5、8位置　　交换6、8位置

堆的添加操作时间复杂度：

O(logn)

堆的操作-添加

put操作(添加元素)



大根堆put()：

算法：

1. 在**堆尾**加入一个元素，并把这个结点置为当前结点。
2. 比较当前结点与父亲结点的大小

   如果当前结点**大于**父亲结点，则**交换**它们的值，并把父亲结点置为当前结点，继续执行第2步。

   如果当前结点小于等于父结点。结束。

堆的操作-添加代码

核心：

1. 从下往上调整

2. 结束条件：

   到达根结点或父亲结点优先级大于它

如果要建立一个堆，怎么办？

执行n次添加，即可建立一个大根堆。

```
for(n次){
    cin>>x;
    put(x);
}
```

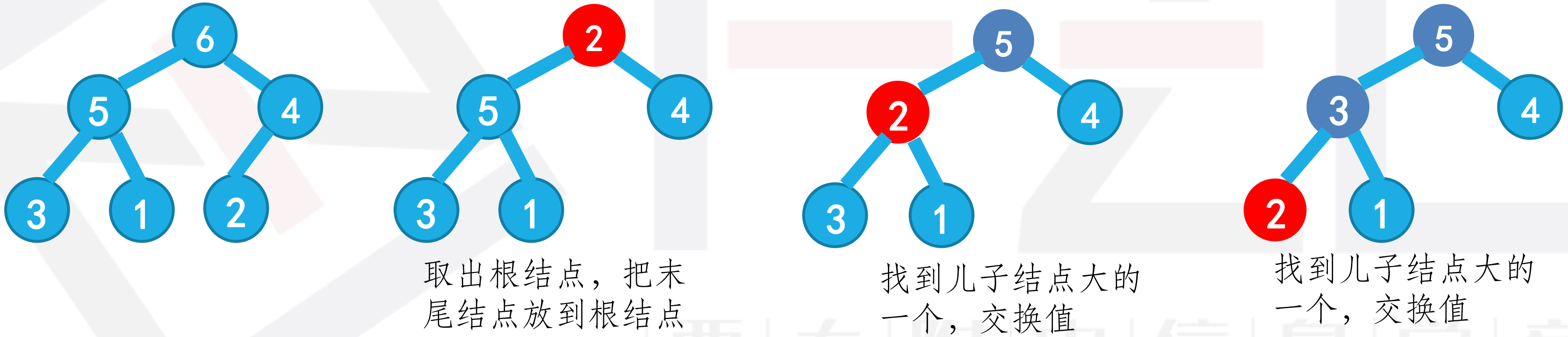
heap[]数组表示堆

```
void put(int d){
int now,father;
heap[++heap_size]=d;   //添加到堆尾
now=heap_size;         //当前结点设为堆尾
while(now>1) {  //当前结点不是根
     father=now/2;
     if(heap[now]>=heap[father]) break;
     swap(heap[now],heap[father]);
     now=father;
   }
}
```

堆的操作-删除

get操作(删除元素)：

只能删除根结点



堆的删除操作时间复杂度：

O(logn)

堆的操作-删除

大根堆get()：

算法：

1. 取出堆的**根节点**的值。

2. 把堆的**最后一个结点**放到根的位置，把根覆盖掉，把堆**长度-1**。

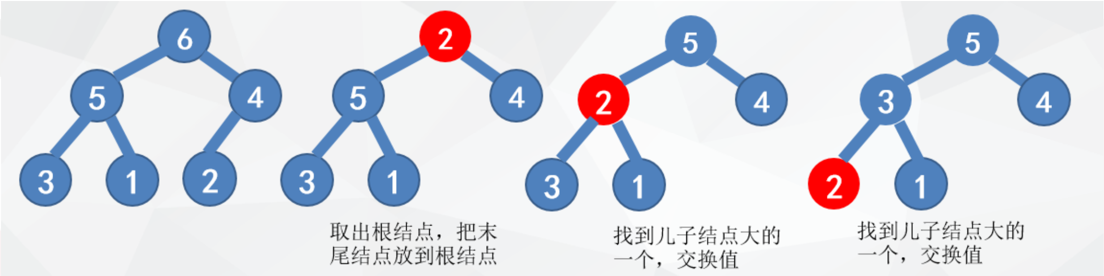
3. 把根节点置为当前结点。

4. 如果当前结点无孩子，结束；否则，将两个孩子**值大的置为当前结点**。

5. 比较当前结点与父亲结点大小：

   如果小于等于父亲结点，结束。

   如果大于父亲结点，交换父亲孩子结点，将孩子结点作为当前结点执行第四步。



堆的操作-删除代码

核心：

1. 从上往下调整
2. 结束条件：

   到达叶子结点或两个孩子结点优先级小于它

```
int get(){
    int now,father,res;
    res=heap[1];            //取出根
    heap[1]=heap[heap_size--];//堆尾放到根的位置
    now=1;          //当前结点设为根
    while(now*2<=heap_size){
        next=now*2;
        if(next+1<=heap_size&&heap[next+1]>heap[next])
            next++;    //选择大的儿子结点
        if(heap[now]>=heap[next]) break;
        swap(heap[now],heap[next]);
        now=next;

    }
    return res;
}
```

堆的应用：堆排序

n个数，排序，从小到大输出。

方法：

1. 建小根堆
2. 输出最小（根结点），调整堆，反复这个过程直到堆为空；

堆排代码核心代码

```
for(i:1~n){ //建堆
    cin>>x;
    put(x);
}

for(i:1~n){ //取数
    t=get();
    cout<<t<<" ";
}
```

= 有序

算一下时间复杂度：

n个数调整，每次调整O(logn)

堆排的时间复杂度为O(nlogn)

但是堆排常数比快排高，理论上没有快排快

再回到合并果子一题

合并果子

解决方法：

1. 建堆 $O(n\log n)$

2. 两次取出最小数（根）。 $O(\log n)+O(\log n)$

3. 将两个最小数之和添加进堆 $O(\log n)$

4. 重复2，3步，直到堆大小size=1。

时间复杂度：$O(n\log n)$

合并果子代码(手写堆版)

```
#include <bits/stdc++.h>
using namespace std;
int n, heap[20002], size;
void put(int t)
{
    int p, p2;
    heap[++size] = t;
    p = size;
    while (p > 1) {
        p2 = p / 2;
        if (heap[p] >= heap[p2])
            return;
        swap(heap[p], heap[p2]);
        p = p2;
    }
}
int get()
{
    int p = 1, ne, ans;
    ans = heap[1];
    heap[1] = heap[size--];
    while (p * 2 <= size) {
        ne = p * 2;
        if (ne < size && heap[ne + 1] < heap[ne])
            ne++;
        if (heap[p] <= heap[ne])
            return ans;
        swap(heap[p], heap[ne]);
        p = ne;
    }
    return ans;
}
```

```
int main(){
    int ans = 0;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        int t;
        cin >> t;
        put(t);
    }
    size = n;
    while (size > 1) {
        int x = get(), y = get();
        ans += (x + y);
        put(x + y);
    }
    cout << ans;
    return 0;
}
```

拓展阅读：合并果子-双队列法

双队列：

队列A:存放未合并的值

队列B:存放合并的新值

1. 对队列A排序,取出前两个合并后加入队列B

2. 比较队列A和队列B的队头，选择一个最小值x出队，继续比较队列A和队列B的队头，选择一个最小值y出队，将x+y入队B。

3. 重复2过程。

STL-优先队列

优先队列

在队列的基础上，添加了内部的一个排序，它本质是一个堆实现的

基础用法：

```
priority_queue<数据类型 > 名字
priority_queue<int > q;    //默认降序排列
```

默认是一个大根堆

进阶用法：

```
升序，小根堆：priority_queue<int,vector<int>,greater<int> > q1;
降序，大根堆：priority_queue<int,vector<int>,less<int> > q2;
```

和队列基本操作相同：

top() 访问队头元素
empty() 队列是否为空
size() 返回队列内元素个数
push(x) 插入元素x到队尾 (并排序)
pop() 弹出队头元素

了解更多：https://www.cnblogs.com/huashanqingzhu/p/11040390.html

结构体-重载运算符

```
struct node{
    int no,dis;
    bool operator < (const node& other) const {  //运算符重载
        if(dis!=other.dis)  //排序条件，根据实际书写这只是举例
            return dis<other.dis;
        return no<other.no;
    }
};
priority_queue<node> q;
```

结构体如何排序?

合并果子代码(STL版)

```
//大根堆版
priority_queue<int> q;
for(i=1;i<=n;++i){
      scanf("%d",&x);
      q.push(-x);
   }
   while(n--){
      t=0;
      t-=q.top();q.pop();
      t-=q.top();q.pop();
      ans+=t;
      q.push(-t);   //取反放入
   }
   printf("%d",ans);
```

```
//小根堆版
priority_queue<int,vector<int>,greater<int> > q;
int n,a,ans=0;
scanf("%d",&n);
for(int i=;i<=n;i++){
    scanf("%d",&a);
    q.push(a);
}

while(q.size()>){
    int x=q.top();q.pop();
    int y=q.top();q.pop();
    ans+=x+y;
    q.push(x+y);
}
printf("%d\n",ans);
return ;
```

序列合并

题目描述
有两个长度都是N的序列A和B，在A和B中各取一个数相加可以得到 N^2个和，求这 N^2个和中最小的N个。
输入输出格式
输入格式:
第一行一个正整数N;
第二行N个整数 A_i，满足 $A_i \le A_i+1$ 且 $Ai \le 10^9$；
第三行N个整数 B_i，满足 $B_i \le B_i+1$ 且 $Bi \le 10^9$.
【数据规模】
对于50%的数据中，满足1<=N<=1000;
对于100%的数据中，满足1<=N<=100000。
输出格式:
输出仅一行，包含N个整数，从小到大输出这N个最小的和，相邻数字之间用空格隔开。
输入输出样例
输入样例:
3
2 6 6
1 4 8
输出样例:
3 6 7

分析

输入：

3

2 6 6

1 4 8

输出：

3 6 7

序列A	2	6	6
序列B	1	4	8

N^2个和

	A[1]	A[2]	A[3]
B[1]	3	7	7
B[2]	6	10	10
B[3]	10	14	14

分析

每一行的最小值是它们

合成的N^2个数：

$$
\begin{array}{l}
\boxed{\begin{array}{l} A[1]+B[1] \\ A[2]+B[2] \\ \cdots\cdots \\ A[N]+B[1] \end{array}}
\end{array}
$$

A[1]+B[1] <= A[1]+B[2] <= … <= A[1]+B[N]
A[2]+B[1] <= A[2]+B[2] <= … <= A[2]+B[N]
……
A[N]+B[1] <= A[N]+B[2] <= … <= A[N]+B[N]

1. 先将a[i]+b[1]都入堆
2. 每出堆一个a[i]+b[j]，就让它的a[i]+b[j+1]入堆，这样可以保证入堆的数保持单调
3. 由于入堆需要知道i,j和入堆的值val三个信息，所以可以使用结构体

时间复杂度：O(NlogN)

序列合并-代码

```
#include<bits/stdc++.h>
using namespace std;
const int N=100000+5;

int n, a[N], b[N], ans[N];

struct number{
    int id1, id2, val;
    bool operator < (const number &a) const{  //运算符重载
        return val > a.val;
    }
};

priority_queue <number> h;

int main(){
    ios::sync_with_stdio(false);  //关闭iostream的输入 输出缓存 ，使cin、cout与scanf和printf效率无差别
    cin >> n;
    for(int i=1;i<=n;i++) cin >> a[i];
    for(int i=1;i<=n;i++) cin >> b[i];
    for(int i=1;i<=n;i++)  h.push((number){ i, 1, a[i]+b[1] });  //数据打包成一个结构体，先让a[n]+b[1]都入堆
    for(int i=1;i<=n;i++){
        number top = h.top(); h.pop();
        ans[i] = top.val;
        h.push((number){ top.id1, top.id2+1, a[top.id1]+b[top.id2+1] });
    }
    for(int i=1;i<=n;i++) cout << ans[i] << ' '; cout << endl;
    return 0;
}
```

查询第K大/小

给定N个数字，求其前i个元素中第K大/小的那个元素。

如果使用数组实现的话，我们直接输入数组下标为K的元素即可。
但我们使用的是堆，它的实现方式——优先队列是不支持任意点访问的，那么我们就无法进行单点查询。

对顶堆解决朴素堆不支持单点查询的问题

对顶堆

对顶堆

对顶堆1　　对顶堆2

如果把堆中的大根堆想成一个上宽下窄的三角形，把小根堆想成一个上窄下宽的三角形

问题：能否想到如何维护这两个堆，求第k大？

求第k大的方法：
我们把大根堆的元素个数限制成K个，前K个元素入队之后，每个元素在入队之前先与堆顶元素比较，如果比堆顶元素大，就加入小根堆，如果没有的话，把大根堆的堆顶弹出，将新元素加入大根堆。这样就维护出一个对顶堆。

求第K大

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 30005;
priority_queue<ll,vector<ll> > q1;  //大根堆
priority_queue<ll, vector<ll>, greater<ll> > q2;  //小根堆
int main()
{
	int n,k;
	k=5; //第k大
	cin>>n;
	for(int i=1;i<=k;i++){  //前k个数入大根堆
		int x;
		cin>>x;
		q1.push(x);
	}
	cout<<"当前第k大: "<<q1.top()<<endl;
	for(int i=1;i<=n-k;i++){
		int x;
		cin>>x;
		if(x<q1.top()){  //如果x小于大根堆根节点
			q1.pop();   //弹出节点
			q1.push(x);  //x入堆
			cout<<"当前第k大: "<<q1.top()<<endl;
		}
		else{  //否则进入小根堆
			q2.push(x);
		}
	}
}
```

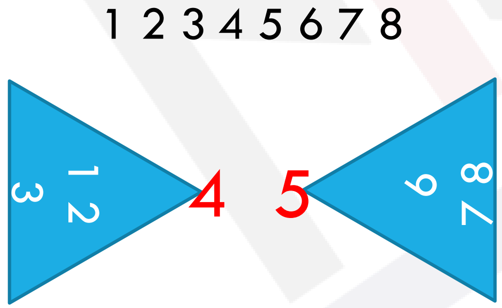
求中位数问题

对顶堆还可以用于解决其他“第K大/小”的变形问题：比如求前i个元素的中位数

思考一下怎么求?

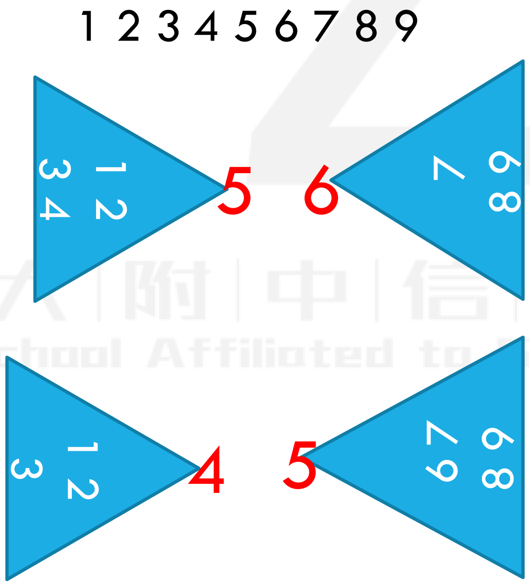
求中位数问题

1.当前总元素量为奇数时只要保证 大顶堆元素比小顶堆元素多一 则中位数为大顶堆堆顶
同理 只要保证 小顶堆元素比大顶堆元素多一 则中位数为小顶堆堆顶
2.当前总元素量为偶数时只要保证大顶堆元素等于小顶堆 则中位数为（大顶堆顶 + 小顶堆顶） / 2.0



大根堆 小根堆

偶数大小相等



奇数任意一个堆比另一个堆大1

例题： Running median

任意位置删除

问题：删除堆中k个任意的值

可删除堆解决朴素堆不支持任意删除的问题。

可删除堆

可删堆

原理也比较简单：我们建一个和原始堆一样的临时堆，如果要删除哪个元素，就把哪个元素压入临时堆，然后待此元素和正常堆的堆顶元素相同时（即两个堆顶一样），就同时弹出

临时堆类似于一个信息库，标记了哪些数字需要删除，
而且临时堆和原始堆方向一致，所以不会出现把后面待删元素提前删除的情况

形象化的比喻：警察拥有一个通缉犯人脸库，对于过安检的人一一比对，比对成功就拷走~~(删除)~~

伪代码

```
 for(k次) {  //k个删除的数入临时堆
       cin>>x;
       q2.push(x);
 }

for(n次) {  //k个删除的数入堆
  if(q1.top()==q2.top()){q1.pop(),q2.pop();}
  else{
    做其他的事情...
  }
}
```

总结

朴素堆、对顶堆和可删除堆一起构成了堆这种数据结构的大部分内容。

对于能用堆维护和解决的一些问题，大体都能用这三种方式解决。

堆的难点，或者推广到数据结构，并不在于理解和掌握，而在于活学活用。

THANKS