LeetCode 1. 两数之和

1. 两数之和

难度 简单 7848

给定一个整数数组 `nums` 和一个目标值 `target`，请你在该数组中找出和为目标值的那 **两个** 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

示例:

```
给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

(哈希表) $O(n)$

使用C++中的哈希表—— `unordered_map<int, int> hash` .

循环一遍 $nums$ 数组，在每步循环中我们做两件事：

1. 判断 $target - nums[i]$ 是否在哈希表中；
2. 将 $nums[i]$ 插入哈希表中；

解释：由于数据保证有且仅有一组解，假设是 $[i, j](i < j)$，则我们循环到 j 时，$nums[i]$一定在哈希表中，且有 $nums[i] + nums[j] == target$，所以我们一定可以找到解。
时间复杂度：由于只扫描一遍，且哈希表的插入和查询操作的复杂度是 $O(1)$，所以总时间复杂度是 $O(n)$.

```cpp
/*
暴力：两重循环

优化：开一个Hash表，hash表中是否存在target - nums[i]
    将nums[i]插入hash表
*/
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target)
    {
        vector<int> res;
        unordered_map<int,int> hash;
        for (int i = 0; i < nums.size(); i ++ )
        {
            int another = target - nums[i];
            if (hash.count(another))//是否存在
            {
                res = vector<int>({hash[another], i});
                break;
            }
            hash[nums[i]] = i;
```

```
        }
        return res;
    }
};
**********************************************************
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int,int> hash;
        for(int i = 0;i < nums.size();i ++)
        {
            if(hash.count(target - nums[i])) return {hash[target - nums[i]],i};
            hash[nums[i]] = i;
        }
        return {-1,-1};
    }
};
```

LeetCode 187. 重复的DNA序列

187. 重复的DNA序列

难度 中等 76

所有 DNA 都由一系列缩写为 A，C，G 和 T 的核苷酸组成，例如："ACGAATTCCG"。在研究 DNA 时，识别 DNA 中的重复序列有时会对研究非常有帮助。

编写一个函数来查找 DNA 分子中所有出现超过一次的 10 个字母长的序列（子串）。

示例：

输入： s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
输出： ["AAAAACCCCC", "CCCCCAAAAA"]

```
/*
用哈希表记录所有长度是10的子串的个数。
从前往后扫描，当子串出现第二次时，将其记录在答案中。
1.插入一个字符串
2.统计字符串出现的次数
*/
class Solution {
public:
    vector<string> findRepeatedDnaSequences(string s) {
        vector<string> res;
        unordered_map<string, int> S;
        for (int i = 0; i + 10 <= s.size(); i ++ )
        {
            string str = s.substr(i, 10);
            if (S[str] == 1) res.push_back(str);
            S[str] ++ ;
```

```
        }
        sort(res.begin(), res.end());
        return res;
    }
};
*****************************************************************************
***
class Solution {
public:
    vector<string> findRepeatedDnaSequences(string s) {
        unordered_map<string,int> hash;
        vector<string> res;

        for(int i = 0; i + 10 <= s.size();i ++)
        {
            string str = s.substr(i,10);
            hash[str] ++;
            if(hash[str] == 2) res.push_back(str);
        }
        return res;
    }
};
```

LeetCode 706. 设计Hash表

706. 设计哈希映射

难度 简单 35

不使用任何内建的哈希表库设计一个哈希映射

具体地说，你的设计应该包含以下的功能

- `put(key, value)`：向哈希映射中插入(键,值)的数值对。如果键对应的值已经存在，更新这个值。
- `get(key)`：返回给定的键所对应的值，如果映射中不包含这个键，返回-1。
- `remove(key)`：如果映射中存在这个键，删除这个数值对。

示例:

```
MyHashMap hashMap = new MyHashMap();
hashMap.put(1, 1);
hashMap.put(2, 2);
hashMap.get(1);            // 返回 1
hashMap.get(3);            // 返回 -1 (未找到)
hashMap.put(2, 1);         // 更新已有的值
hashMap.get(2);            // 返回 1
hashMap.remove(2);         // 删除键为2的数据
hashMap.get(2);            // 返回 -1 (未找到)
```

注意:

- 所有的值都在 `[1, 1000000]` 的范围内。
- 操作的总数目在 `[1, 10000]` 范围内。
- 不要使用内建的哈希库。

算法1

(拉链法) $O(1)$

哈希表的基本思想都是先开一个大数组，然后用某种哈希函数将key映射到数组的下标空间。
不同算法的区别在于如何处理下标冲突，即当两个不同的key被映射到同一下标时，该怎么办。

一般有两种方式处理冲突：拉链法和开放寻址法。
首先我们来介绍拉链法。它的思想很简单，在哈希表中的每个位置上，用一个链表来存储所有映射到该位置的元素。

- 对于 `put(key, value)` 操作，我们先求出key的哈希值，然后遍历该位置上的链表，如果链表中包含key，则更新其对应的value；如果链表中不包含key，则直接将（key，value）插入该链表中。
- 对于 `get(key)` 操作，求出key对应的哈希值后，遍历该位置上的链表，如果key在链表中，则返回其对应的value，否则返回-1。
- 对于 `remove(key)`，求出key的哈希值后，遍历该位置上的链表，如果key在链表中，则将其删除。

时间复杂度分析：最坏情况下，所有key的哈希值都相同，且key互不相同，则所有操作的时间复杂度都是 $O(n)$。但最坏情况很难达到，每个操作的期望时间复杂度是 $O(1)$。
空间复杂度分析：一般情况下，初始的大数组开到总数据量的两到三倍大小即可，且所有链表的总长度是 $O(n)$ 级别的，所以总空间复杂度是 $O(n)$。

```
/*
```

```
*/
class MyHashMap {
public:
    /** Initialize your data structure here. */
    const static int N = 20011;

    vector<list<pair<int,int>>> hash;

    MyHashMap() {
        hash = vector<list<pair<int,int>>>(N);
    }

    list<pair<int,int>>::iterator find(int key)
    {
        int t = key % N;
        auto it = hash[t].begin();
        for (; it != hash[t].end(); it ++ )
            if (it->first == key)
                break;
        return it;
    }

    /** value will always be non-negative. */
    void put(int key, int value) {
        int t = key % N;
        auto it = find(key);
        if (it == hash[t].end())
            hash[t].push_back(make_pair(key, value));
        else
            it->second = value;
    }

    /** Returns the value to which the specified key is mapped, or -1 if this
map contains no mapping for the key */
    int get(int key) {
        auto it = find(key);
        if (it == hash[key % N].end())
            return -1;
        return it->second;
    }

    /** Removes the mapping of the specified value key if this map contains a
mapping for the key */
    void remove(int key) {
        int t = key % N;
        auto it = find(key);
        if (it != hash[t].end())
            hash[t].erase(it);
    }
};

/**
 * Your MyHashMap object will be instantiated and called as such:
 * MyHashMap obj = new MyHashMap();
 * obj.put(key,value);
 * int param_2 = obj.get(key);
 * obj.remove(key);
 */
```

算法2

(开放寻址法) $O(1)$

开放寻址法的基本思想是这样的：如果当前位置已经被占，则顺次查看下一个位置，直到找到一个空位置为止。

- 对于 `put(key, value)` 操作，求出key的哈希值后，顺次往后找，直到找到key或者找到一个空位置为止，然后将key放到该位置上，同时更新相应的value。
- 对于 `get(key)` 操作，求出key的哈希值后，顺次往后找，直到找到key或者-1为止（注意空位置有两种：-1和-2，这里找到-1才会停止），如果找到了key，则返回对应的value。
- 对于 `remove(key)` 操作，求出key的哈希值后，顺次往后找，直到找到key或者-1为止，如果找到了key，则将该位置的key改为-2，表示该数已被删除。

注意：当我们把一个key删除后，不能将其改成-1，而应该打上另一种标记。否则一个连续的链会从该位置断开，导致后面的数查询不到。

时间复杂度分析：最坏情况下，所有key的哈希值都相同，且key互不相同，则所有操作的时间复杂度都是 $O(n)$。但实际应用中最坏情况难以遇到，每种操作的期望时间复杂度是 $O(1)$。
空间复杂度分析：一般来说，初始大数组开到总数据量的两到三倍，就可以得到比较好的运行效率，空间复杂度是 $O(n)$。

```
class MyHashMap {
public:
    /** Initialize your data structure here. */
    const static int N = 20011;
    int hash_key[N], hash_value[N];

    MyHashMap() {
        memset(hash_key, -1, sizeof hash_key);
    }

    int find(int key)
    {
        int t = key % N;
        while (hash_key[t] != key && hash_key[t] != -1)
        {
            if ( ++t == N) t = 0;
        }
        return t;
    }

    /** value will always be non-negative. */
    void put(int key, int value) {
        int t = find(key);
        hash_key[t] = key;
        hash_value[t] = value;
    }

    /** Returns the value to which the specified key is mapped, or -1 if this
map contains no mapping for the key */
    int get(int key) {
        int t = find(key);
        if (hash_key[t] == -1) return -1;
        return hash_value[t];
    }
```

```
    /** Removes the mapping of the specified value key if this map contains a
mapping for the key */
    void remove(int key) {
        int t = find(key);
        if (hash_key[t] != -1)
            hash_key[t] = -2;
    }
};

/**
 * Your MyHashMap object will be instantiated and called as such:
 * MyHashMap obj = new MyHashMap();
 * obj.put(key,value);
 * int param_2 = obj.get(key);
 * obj.remove(key);
 */
```

LeetCode 652.寻找重复的子树

652. 寻找重复的子树

难度 中等 81

给定一棵二叉树，返回所有重复的子树。对于同一类的重复子树，你只需要返回其中任意一棵的根结点即可。

两棵树重复是指它们具有相同的结构以及相同的结点值。

示例 1：

```
        1
      / \
     2   3
    /   / \
   4   2   4
      /
     4
```

下面是两个重复的子树：

```
      2
     /
    4
```

和

```
    4
```

因此，你需要以列表的形式返回上述重复子树的根结点。

算法

(深度优先遍历，哈希表) $O(n^2)$

1. 使用 `unordered_map` 记录每个子树经过哈希后的数量，哈希方法可以用最简单的前序遍历，即 `根,左子树,右子树` 的方式递归构造。 `逗号` 和每个叶子结点下的 `空结点` 的位置需要保留。
2. 若发现当前子树在哈希表第二次出现，则将该结点记入答案列表。

时间复杂度

- 每个结点仅遍历一次， `unordered_map` 单次操作的时间复杂度为 $O(1)$。
- 但遍历结点中，可能要拷贝当前字符串到答案，拷贝的时间复杂度为 $O(n)$，故总时间复杂度为 $O(n^2)$。

```
/*
先把一棵二叉树中序遍历成字符串
把结果的字符串hash成整数
再用一个hash求相同的
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    string dfs(TreeNode *r, unordered_map<string, int> &hash, vector<TreeNode*>
&ans) {
        if (r == NULL)
            return "";
        string cur = "";
        cur += to_string(r -> val) + ",";
        cur += dfs(r -> left, hash, ans) + ",";
        cur += dfs(r -> right, hash, ans);
        hash[cur]++;
        if (hash[cur] == 2)
            ans.push_back(r);
        return cur;
    }
    vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
        unordered_map<string, int> hash;
        vector<TreeNode*> ans;
        dfs(root, hash, ans);
        return ans;
    }
};
**********************************************************************************
***
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
```

```
 */
class Solution {
public:

    int cnt = 0;//第几次遇到的字符串
    unordered_map<string,int> hash;
    unordered_map<int,int> count;
    vector<TreeNode*> ans;

    string dfs(TreeNode* root)
    {
        if(!root) return to_string(hash["#"]);

        auto left = dfs(root->left);
        auto right = dfs(root->right);
        string tree = to_string(root->val) + ',' + left + ',' + right;
        if(!hash.count(tree)) hash[tree] = ++ cnt;
        int t = hash[tree];//字符串hash之后的值
        count[t] ++;
        if(count[t] == 2) ans.push_back(root);

        return to_string(t);
    }
    vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
        hash["#"] = ++ cnt; //空字符串用#代替
        dfs(root);
        return ans;
    }
};
```

LeetCode 560. 和为K的子数组

560. 和为K的子数组

难度 中等 234

给定一个整数数组和一个整数 **k**，你需要找到该数组中和为 **k** 的连续的子数组的个数。

示例 1：

```
输入:nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。
```

说明：

1. 数组的长度为 [1, 20,000]。
2. 数组中元素的范围是 [-1000, 1000] ，且整数 **k** 的范围是 [-1e7, 1e7]。

算法

(前缀和，哈希表) $O(n)$

1. 对原数组求前缀和后，一个和为 **k** 的子数组即为一对前缀和的差值为 **k** 的位置。
2. 遍历前缀和数组，用 `unordered_map` 哈希表记录每个前缀和出现的次数。特别地，初始时前缀和为 0 需要被额外记录一次。
3. 遍历过程中，对于当前前缀和 tot，累计之前 tot - k 前缀和出现的次数累积到答案即可。

时间复杂度

- 每个位置仅遍历一次，哈希表单次操作的时间复杂度为 $O(1)$，故总时间复杂度为 $O(n)$。

```
/*
前缀和加Hash

暴力：枚举以i为终点，前面有多少个数等于s[i] - k
前缀和加Hash：前缀和算，Hash查
*/
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<int, int> hash;
        int tot = 0, ans = 0;
        hash[0] = 1;
        for (auto x : nums) {
            tot += x;
            ans += hash[tot - k];
            hash[tot]++;
        }
        return ans;
    }
};
**********************************************************************
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<int,int> hash;
        hash[0] = 1;
        int res = 0;

        for(int i = 0,sum = 0;i < nums.size();i ++)
        {
            sum += nums[i];
            res += hash[sum - k];
            hash[sum] ++;
        }
        return res;
    }
};
```

LeetCode 547. 朋友圈

547. 朋友圈

难度 中等 197

班上有 **N** 名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。所谓的朋友圈，是指所有朋友的集合。

给定一个 **N * N** 的矩阵 **M**，表示班级中学生之间的朋友关系。如果M[i][j] = 1，表示已知第 i 个和 j 个学生**互为**朋友关系，否则为不知道。你必须输出所有学生中的已知的朋友圈总数。

示例 1:

```
输入：
[[1,1,0],
 [1,1,0],
 [0,0,1]]
输出：2
说明：已知学生0和学生1互为朋友，他们在一个朋友圈。
第2个学生自己在一个朋友圈。所以返回2。
```

示例 2:

```
输入：
[[1,1,0],
 [1,1,1],
 [0,1,1]]
输出：1
说明：已知学生0和学生1互为朋友，学生1和学生2互为朋友，所以学生0和学生2也是朋友，所以他们三个在一个朋友圈，返回1。
```

注意:

1. N 在[1,200]的范围内。
2. 对于所有学生，有M[i][i] = 1。
3. 如果有M[i][j] = 1，则有M[j][i] = 1。

算法

(并查集) $O(n^2)$

1. 此题为并查集的入门题。
2. 基础的并查集能解决的一类问题是不断将两个元素所在集合合并，并随时询问两个元素是否在同一集合。
3. 定义数组 f(i) 表示 i 元素所在集合的根结点（代表元素）。初始时，所有元素所在集合的根结点就是自身。
4. 合并时，直接将两个集合的根结点合并，即修改 f 数组。
5. 查询时，不断通过判断 i 是否等于 f(i) 的操作，若不相等则递归判断 f(f(i))，直到 i == f(i) 为止。
6. 但以上做法会在一条链的情况下单次查询的时间复杂度退化至线性，故可以采用 路径压缩 优化，将复杂度降到近似常数。读者可以自行查阅相关资料。
7. 对于此题，最后只需检查有多少个元素为一个集合的根结点即可。

```
/*
并查集作用o(1)
```

```
1.合并两个集合
2.判断连个点是否在同一集合中

*/
class Solution {
public:
    int n;
    vector<int> f;//父节点

    int find(int x) {
        return x == f[x] ? x : f[x] = find(f[x]);
    }
    int findCircleNum(vector<vector<int>>& M) {
        n = M.size();
        f = vector<int>(n);
        for (int i = 0; i < n; i++)
            f[i] = i;
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)//无向图，遍历一半就可以
                if (M[i][j] == 1) {
                    int fx = find(i), fy = find(j);
                    if (fx != fy)
                        f[fx] = fy;
                }

        int ans = 0;
        for (int i = 0; i < n; i++)
            if (i == find(i))
                ans++;
        return ans;
    }
};
```

LeetCode 684. 冗余连接

684. 冗余连接

难度 中等 84

在本问题中, 树指的是一个连通且无环的**无向**图。

输入一个图，该图由一个有着N个节点 (节点值不重复1, 2, ..., N) 的树及一条附加的边构成。附加的边的两个顶点包含在1到N中间，这条附加的边不属于树中已存在的边。

结果图是一个以 边 组成的二维数组。每一个 边 的元素是一对 `[u, v]` ，满足 `u < v`，表示连接顶点 `u` 和 `v` 的**无向**图的边。

返回一条可以删去的边，使得结果图是一个有着N个节点的树。如果有多个答案，则返回二维数组中最后出现的边。答案边 `[u, v]` 应满足相同的格式 `u < v`。

示例 1:

```
输入: [[1,2], [1,3], [2,3]]
输出: [2,3]
解释: 给定的无向图为:
  1
 / \
2 - 3
```

示例 2:

```
输入: [[1,2], [2,3], [3,4], [1,4], [1,5]]
输出: [1,4]
解释: 给定的无向图为:
5 - 1 - 2
    |   |
    4 - 3
```

注意:

- 输入的二维数组大小在 3 到 1000。
- 二维数组中的整数在1到N之间，其中N是输入数组的大小。

```
/*
出现环的条件是某条边，边的两个端点原本就是连通的，那么加上了这条边以后就产生了环，因此我们在加入每条边的时候需要判断一下边的两个端点本身是不是连通的即可。

如果连通且不在一个集合之中就合并集合，否则这边就是要求的边
*/
class UnionFind{
public:
    vector<int>father;
    UnionFind(int num){//num表示元素的个数
        for(int i = 0; i < num; i++){
            father.push_back(i);//箭头指向自己
        }
    }
```

```
    int Find(int n){
        //递归
        if(father[n] == n)
            return n;
        father[n] = Find(father[n]);//路径压缩版本
        return father[n];
    }
    bool Union(int a, int b){//返回a和b是否本身在一个集合里
        int fa = Find(a);
        int fb = Find(b);
        bool res = fa==fb;
        father[fb] = fa;
        return res;
    }
};

class Solution {
public:
    vector<int> findRedundantConnection(vector<vector<int>>& edges) {
        int N = edges.size();
        UnionFind UF(N+1);
        vector<int>res(2, 0);
        for(int i = 0; i < edges.size(); i++){
            int u = edges[i][0];
            int v = edges[i][1];
            if(UF.Union(u, v)){//冗余
                res[0] = u;
                res[1] = v;
            }
        }
        return res;
    }
};
**************************************************************************
class Solution {
public:
    vector<int> p;

    int find(int x)
    {
        if(p[x] != x) p[x] = find(p[x]);
        return p[x];
    }
    vector<int> findRedundantConnection(vector<vector<int>>& edges) {
        int n = edges.size();

        for(int i = 0;i <= n;i ++) p.push_back(i);

        for(auto e : edges)
        {
            int a = e[0],b = e[1];
            if(find(a) == find(b)) return {a,b};
            p[find(a)] = find(b);
        }
        return {-1,-1};
    }
};
```

LeetCode 692.前K个高频单词

692. 前K个高频单词

难度 中等 71

给一非空的单词列表，返回前 *k* 个出现次数最多的单词。

返回的答案应该按单词出现频率由高到低排序。如果不同的单词有相同出现频率，按字母顺序排序。

示例 1：

```
输入：["i", "love", "leetcode", "i", "love", "coding"], k = 2
输出：["i", "love"]
解析："i" 和 "love" 为出现次数最多的两个单词，均为2次。
    注意，按字母顺序 "i" 在 "love" 之前。
```

示例 2：

```
输入：["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4
输出：["the", "is", "sunny", "day"]
解析："the", "is", "sunny" 和 "day" 是出现次数最多的四个单词，
    出现次数依次为 4, 3, 2 和 1 次。
```

注意：

1. 假定 *k* 总为有效值， 1 ≤ *k* ≤ 集合元素数。
2. 输入的单词均由小写字母组成。

扩展练习：

1. 尝试以 *O*(*n* log *k*) 时间复杂度和 *O*(*n*) 空间复杂度解决。

算法

(堆) $O(nlogk)$

题目要求$O(nlogk)$复杂度，所以考虑用堆(优先队列)来做。

1. 先用一个哈希表(unordered_map)来统计每个单词出现次数
2. 遍历一次哈希表的单词及其频率，把频率的相反数作为堆的排序依据塞进堆中。注意C++默认大顶堆序，我们希望堆顶是k个结果中频率最小的(充当边界线),所以用频率相反数作为排序依据。
3. 将堆中元素逆序输出

```
/*
找出出现次数最多的k个单词
```

```
用小根堆来委会出现次数最多的k个单词
*/
class Solution {
public:

    vector<string> topKFrequent(vector<string>& words, int k) {
        unordered_map<string, int> hash;

        typedef pair<int, string> PIS;
        priority_queue<PIS> heap;
        for (auto word : words) hash[word] ++ ;

        for (auto item : hash)
        {
            PIS t(-item.second, item.first);
            if (heap.size() == k && t < heap.top()) heap.pop();
            if (heap.size() < k) heap.push(t);
        }

        vector<string> res(k);
        for (int i = k - 1; i >= 0; i -- )
        {
            res[i] = heap.top().second;
            heap.pop();
        }

        return res;
    }
};
**********************************************************************
class Solution {
public:
    vector<string> topKFrequent(vector<string>& words, int k) {
        unordered_map<string,int> hash;
        typedef pair<int,string> PIS;
        priority_queue<PIS> heap;

        for(auto word : words) hash[word] ++;

        for(auto item : hash)
        {
            PIS t(-item.second,item.first);
            heap.push(t);
            if(heap.size() > k) heap.pop();
        }

        vector<string> res(k);
        for(int i = k - 1;i >= 0;i --)
        {
            res[i] = heap.top().second;
            heap.pop();
        }
        return res;
    }
};
```

LeetCode 295.数据流的中位数

295. 数据流的中位数

难度 困难 133

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是 (2 + 3) / 2 = 2.5

设计一个支持以下两种操作的数据结构：

- void addNum(int num) - 从数据流中添加一个整数到数据结构中。
- double findMedian() - 返回目前所有元素的中位数。

示例：

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

进阶：

1. 如果数据流中所有整数都在 0 到 100 范围内，你将如何优化你的算法?
2. 如果数据流中 99% 的整数都在 0 到 100 范围内，你将如何优化你的算法?

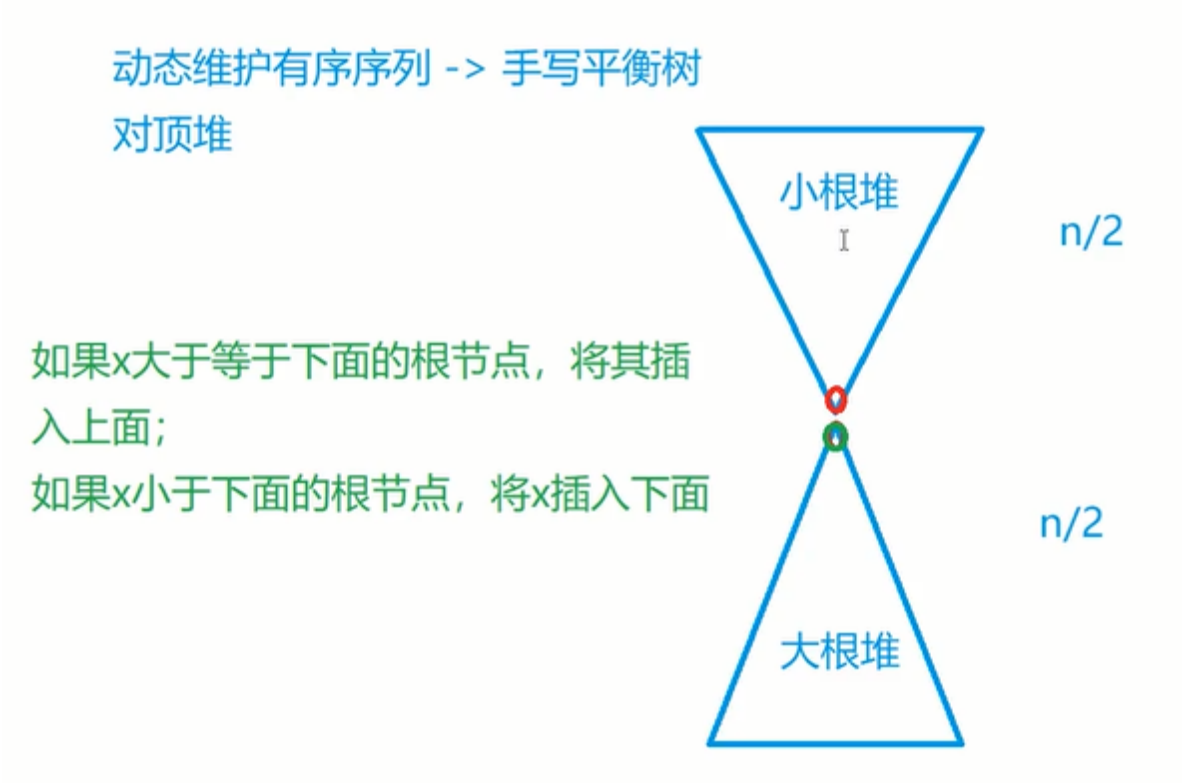
算法

(双堆) $O(n \log n)$

1. 建立一个大根堆，一个小根堆。大根堆存储小于当前中位数，小根堆存储大于等于当前中位数。且小根堆的大小永远都比大根堆大1或相等。
2. 根据上述定义，我们每次可以通过小根堆的堆顶或者两个堆的堆顶元素的平均数求出中位数。
3. 维护时，如果新加入的元素大于等于当前的中位数，则加入小根堆；否则加入大根堆。然后如果发现两个堆的大小关系不满足上述要求，则可以通过弹出一个堆的元素放到另一个堆中。

时间复杂度

- 每次维护堆的时间为$O(\log n)$，取出中位数的时间为$O(1)$。故总时间复杂度为$O(n \log n)$。




```cpp
//维护下面的比上面的最多少一个！！！
class MedianFinder {
public:
    /** initialize your data structure here. */
    priority_queue<int> smaller;
    priority_queue<int, vector<int>, greater<int>> larger;
    MedianFinder() {

    }

    void addNum(int num) {
        if (smaller.empty() || num <= smaller.top())
            smaller.push(num);
        else
            larger.push(num);

        if (smaller.size() == larger.size() + 2) {
            int top = smaller.top();
            smaller.pop();
            larger.push(top);
        }
        else if (larger.size() == smaller.size() + 1) {
            int top = larger.top();
            larger.pop();
            smaller.push(top);
        }
    }

    double findMedian() {
        if (smaller.size() == larger.size())
            return (smaller.top() + larger.top()) / 2.0;

        return smaller.top();
    }
};
```

```
/**
 * Your MedianFinder object will be instantiated and called as such:
 * MedianFinder obj = new MedianFinder();
 * obj.addNum(num);
 * double param_2 = obj.findMedian();
 */
*****************************************************************************
class MedianFinder {
public:
    /** initialize your data structure here. */
    priority_queue<int> down;//下面的
    priority_queue<int,vector<int>,greater<int>> up;//上面的
    MedianFinder() {

    }

    void addNum(int num) {
        if(down.empty() || num >= down.top()) up.push(num);
        else
        {
            down.push(num);//每次要往下面添加时，先添加，再往上拿一个，拿下面最大的
            up.push(down.top());
            down.pop();
        }
        if(up.size() > down.size() + 1)//上面多了，往下拿最小的
        {
            down.push(up.top());
            up.pop();
        }
    }

    double findMedian() {
        if(down.size() + up.size() & 1) return up.top();//总数是奇数
        else return (down.top() + up.top()) / 2.0;//总数是偶数
    }
};

/**
 * Your MedianFinder object will be instantiated and called as such:
 * MedianFinder* obj = new MedianFinder();
 * obj->addNum(num);
 * double param_2 = obj->findMedian();
 */
```

LeetCode 352. 将数据流变为多个不相交区间

352. 将数据流变为多个不相交区间

难度 困难 21

给定一个非负整数的数据流输入 a_1，a_2，…，a_n，…，将到目前为止看到的数字总结为不相交的区间列表。

例如，假设数据流中的整数为 1，3，7，2，6，…，每次的总结为：

```
[1, 1]
[1, 1], [3, 3]
[1, 1], [3, 3], [7, 7]
[1, 3], [7, 7]
[1, 3], [6, 7]
```

进阶：
如果有很多合并，并且与数据流的大小相比，不相交区间的数量很小，该怎么办?

提示：
特别感谢 @yunhong 提供了本问题和其测试用例。

算法

(平衡树) addNum: $O(logn)$, getIntervals: $O(n)$

我们用 `map<int,int> L, R` 来动态维护所有区间，`L` 可以从区间右端点索引到左端点，`R` 可以从区间左端点索引到右端点。
举例来说，假设有个区间是 `[x, y]`，则 `L[y] = x` 并且 `R[x] = y`。

对于 `addNum(val)` 操作：

1. 我们先判断 `val` 是否已经包含在某个区间中。具体来说，先用 `upper_bound(val)` 找出最后一个左端点小于等于 `val` 的区间，然后判断该区间的右端点是否大于等于 `val`。
2. 如果 `val` 不包含在任何区间中，则分别判断 `val-1` 和 `val+1` 是否存在：
   - 如果都存在，则合并左右区间，合并方式：`R[L[val - 1]] = R[val + 1]`，`L[R[val + 1]] = L[val - 1]`，然后将 `R[val+1]` 和 `L[val-1]` 删除；
   - 如果只有一边存在，则将 `val` 插入该区间中；
   - 如果都不存在，则将 `val` 表示成一个单独的区间；

对于 `getIntervals()` 操作，直接输出所有区间即可。

时间复杂度分析：对于 `addNum` 操作，只涉及常数次平衡树的增删改查操作，所以时间复杂度是 $O(logn)$。对于 `getIntervals` 操作，需要将整棵平衡树遍历一遍，所以时间复杂度是 $O(n)$。

```
/**
 * Definition for an interval.
 * struct Interval {
 *     int start;
 *     int end;
 *     Interval() : start(0), end(0) {}
 *     Interval(int s, int e) : start(s), end(e) {}
 * };
 */
class SummaryRanges {
public:
```

```
    /** Initialize your data structure here. */
    map<int,int>L, R;
    SummaryRanges() {

    }

    void addNum(int val) {
        if (R.size())
        {
            auto it = R.upper_bound(val);
            if (it != R.begin())
            {
                -- it;
                if (it->second >= val) return;
            }
        }
        int right = R.count(val + 1), left = L.count(val - 1);
        if (left && right)
        {
            R[L[val - 1]] = R[val + 1];
            L[R[val + 1]] = L[val - 1];
            R.erase(val + 1), L.erase(val - 1);
        }
        else if (right)
        {
            L[R[val + 1]] = val;
            R[val] = R[val + 1];
            R.erase(val + 1);
        }
        else if (left)
        {
            R[L[val - 1]] = val;
            L[val] = L[val - 1];
            L.erase(val - 1);
        }
        else
        {
            R[val] = val;
            L[val] = val;
        }
    }

    vector<Interval> getIntervals() {
        vector<Interval> res;
        for (auto &p : R) res.push_back(Interval(p.first, p.second));
        return res;
    }
};

/**
 * Your SummaryRanges object will be instantiated and called as such:
 * SummaryRanges obj = new SummaryRanges();
 * obj.addNum(val);
 * vector<Interval> param_2 = obj.getIntervals();
 */
```