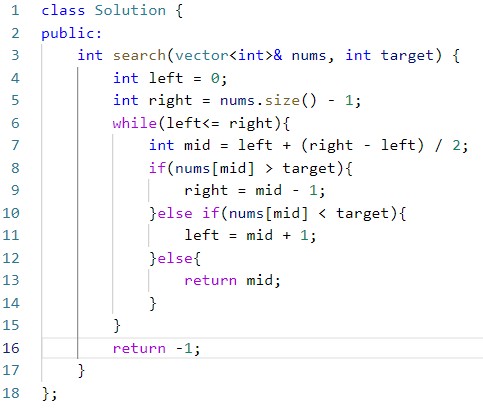
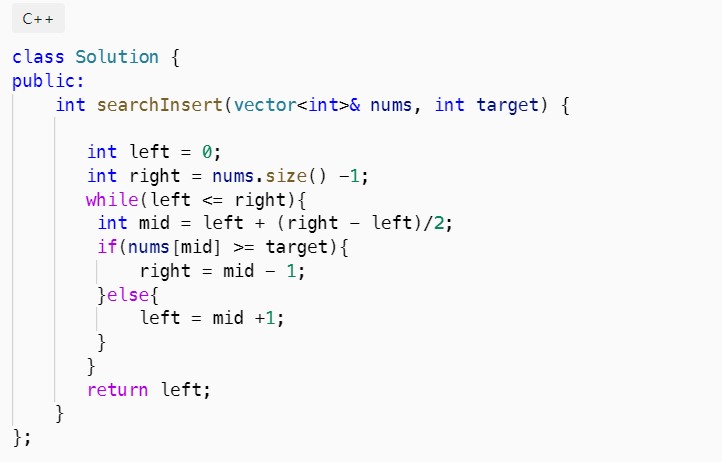
1. 704：二分查找





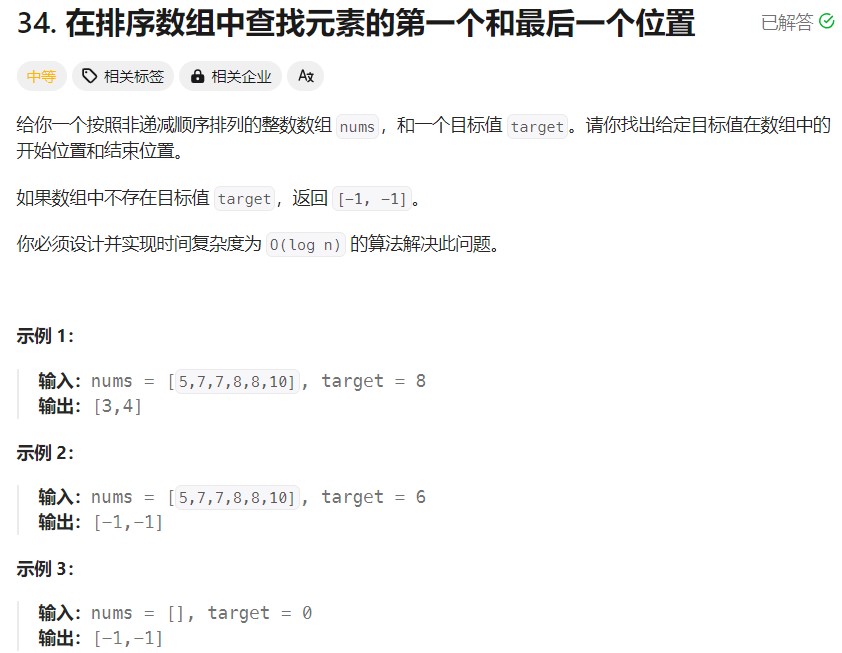
1. 35：搜索插入位置





想象一个for循环，两个指针，一个从左边开始往右，一个从右边开始往左，当while中left<=right的时候就是for循环中两个指针指向了中间位置，就像上面在nums[mid] >= target的时候right = mid – 1;也就是说，如果要返回right那就要返回right + 1。如果是left那就可以直接返回，因为left已经往右边移动一个位置了

1. 34：在排序数组中查找元素的第一个和最后一个位置



下面是用二分法做的，也可以用暴力法的思路，就算左右边界都用for循环遍历数组，找到第一个nums[i]==target的时候就算左边界，右边界可以从右边往左遍历

class Solution {

public:

    vector<int> searchRange(vector<int>& nums, int target) {

        vector<int> res{-1, -1};

        res[0] = left(nums, target);

        res[1] = right(nums, target);

        return res;

    }

    int left(vector<int>& nums, int target){

        int left = 0;

        int right = nums.size() -1;

        int ans = -1;

        while(left<=right){

            int mid = left + (right - left) / 2;

            if(nums[mid]>=target){

                if(nums[mid] == target) ans = mid;

                right = mid - 1;

            }else{

                left = mid + 1;

            }

        }

        return ans;

    }

    int right(vector<int>& nums, int target){

        int left = 0;

        int right = nums.size() -1;

        int ans = -1;

        while(left<=right){

            int mid = left + (right - left) / 2;

            if(nums[mid]<=target){

                if(nums[mid] == target) ans = mid;

                left = mid + 1;

            }else{

                right = mid - 1;

            }

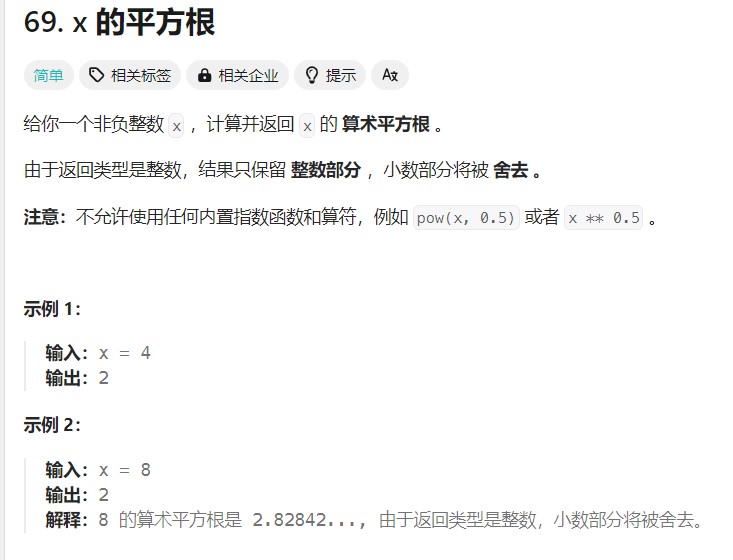
        }

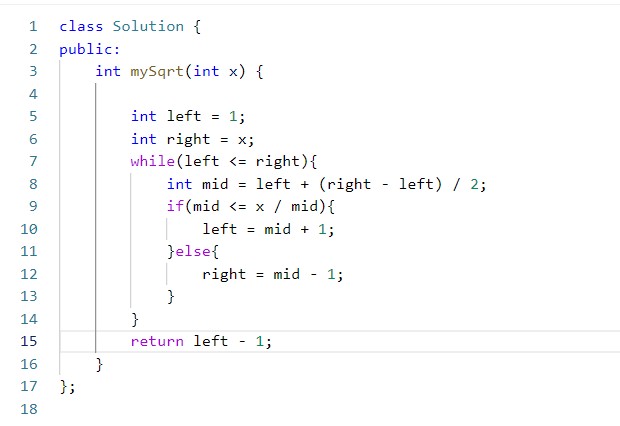
        return ans;

    }

};

1. 69题





这里为什么是返回left-1呢，是因为mid <= x / mid的时候left = mid + 1;所以退出循环的时候left \* left 会大于target，所以返回left – 1;，也可以返回right

1. 367：有效的完全平方数



class Solution {

public:

    bool isPerfectSquare(int num) {

        if(num == 1) return true;

        int left = 1;

        int right = num;

        while(left<=right){

            int mid = left + (right - left) / 2;

            if(mid <= num / mid){

                left = mid + 1;

            }else{

                right = mid - 1;

            }

        }

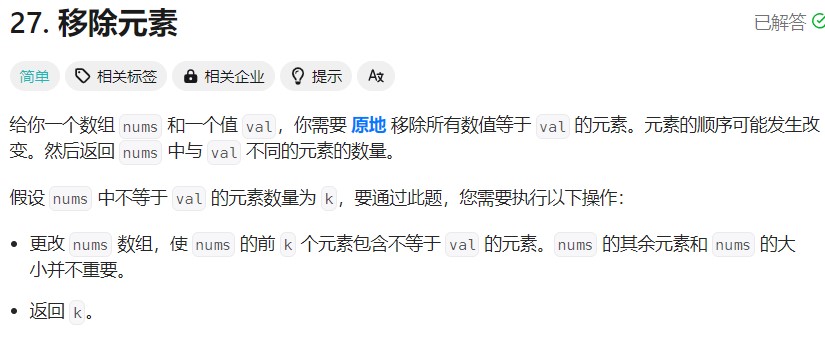
        return (right \* right) == num;

    }

};

就像上面找边界一样，循环中如果mid\*mid > num，那么right = mid – 1；所以退出循环的时候right \* right 是<= num的，所以返回return (right \* right) == num;

1. 27：移除元素



class Solution {

public:

    int removeElement(vector<int>& nums, int val) {

        int slow = 0;

        for(int fast = 0; fast < nums.size(); ++fast){

            if(nums[fast]!=val){

                nums[slow++] = nums[fast];

            }

        }

        return slow;

    }

};

直接快慢指针秒了

1. 26：删除有序数组中的重复项

class Solution {

public:

    int removeDuplicates(vector<int>& nums) {

        int slow = 0;

        nums[slow++] = nums[0];

        for(int fast = 1; fast < nums.size(); fast++){

            if(nums[slow - 1] != nums[fast]){

                nums[slow++] = nums[fast];

            }

        }

        return slow;

    }

};

1. 283：移动零

class Solution {

public:

    void moveZeroes(vector<int>& nums) {

        int count = 0;

        int n = nums.size();

        //count就是不为0的数量

        for(int i = 0; i < n; i++){

            if(nums[i]!=0){

                nums[count++] = nums[i];

            }

        }

        for(int i = count; i < n; i++){

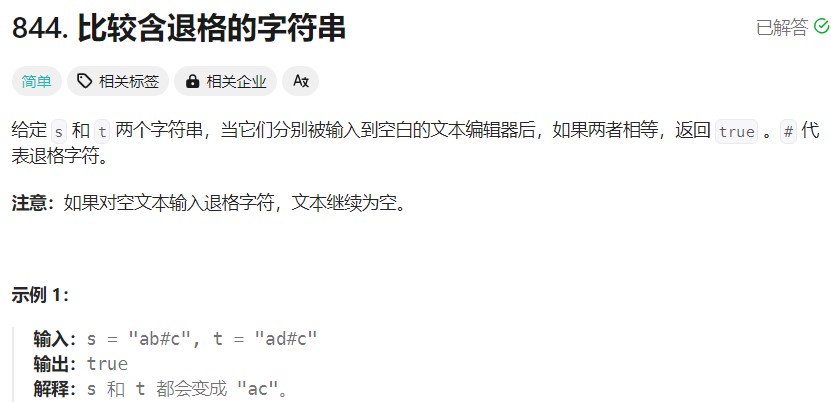
            nums[count++] = 0;

        }

    }

};

1. 844：比较含退格的字符串



class Solution {

public:

    bool backspaceCompare(string s, string t) {

        return outString(s) == outString(t);

    }

    string outString(const string& s){

        string res = "";

        for(int i = 0; i < s.size(); i++)

        {

            if(s[i] != '#'){

                res.push\_back(s[i]);

            }else if(!res.empty()){

                res.pop\_back();

            }

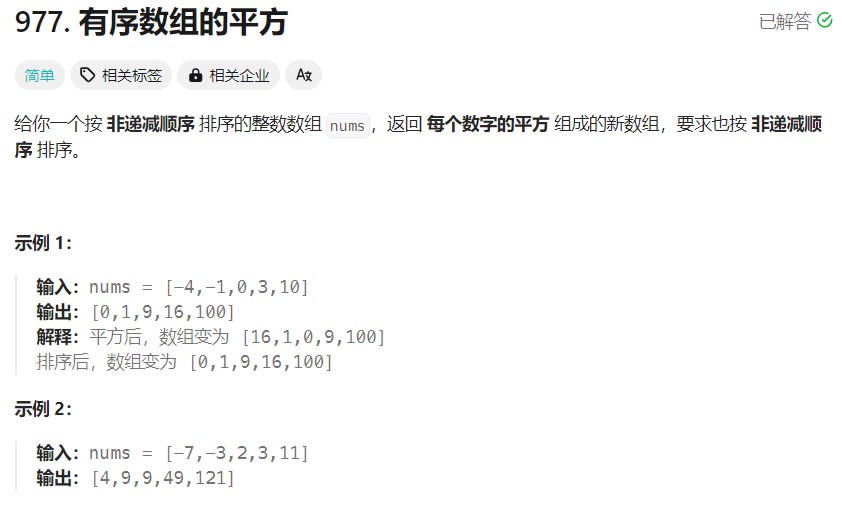
        }

        return res;

    }

};

1. 977：有序数组的平方



class Solution {

public:

    vector<int> sortedSquares(vector<int>& nums) {

        int size = nums.size();

        vector<int> res;

        res.resize(size);

        int left = 0;

        int right = size - 1;

        size--;

        while(left<=right){

            if(nums[left] \* nums[left] >= nums[right] \* nums[right]){

                res[size--] = nums[left] \* nums[left];

                left++;

            }else{

                res[size--] = nums[right] \* nums[right];

                right--;

            }

        }

        return res;

    }

};

1. 209：长度最小的子数组



class Solution {

public:

    int minSubArrayLen(int target, vector<int>& nums) {

        int sum = 0;

        int count = 0;

        int ans = INT\_MAX;

        for(int i = 0, j = 0; j < nums.size(); j++){

            sum += nums[j];

            while(sum >= target){

                count = j - i + 1;

                ans = count < ans ? count : ans;

                sum -= nums[i];

                i++;

            }

        }

        return ans != INT\_MAX ? ans : 0;

        //暴力 无法完全通过测试用例18/21

        // int res = INT\_MAX;

        // int length = 0;

        // for(int i = 0; i< nums.size(); i++){

        //     int sum = 0;

        //     for(int j = i; j < nums.size(); j++){

        //         sum += nums[j];

        //         length = j - i + 1;

        //         if(sum >= target) {

        //             res = min(length,res);

        //             break;

        //         }

        //     }

        // }

        // if(res == INT\_MAX)  return 0;

        // return res;

    }

};

要注意提供的第三个测试用例，因为可能就数组所有元素加起来都不大于target，所以在最后要判断ans != INT\_MAX ? ans : 0;

1. 904：水果成篮



class Solution {

public:

    int totalFruit(vector<int>& fruits) {

        unordered\_map<int, int> m\_map;

        int ans = 0;

        int count = 0;

        for(int i=0, j=0; j < fruits.size(); j++){

            m\_map[fruits[j]]++;

            while(m\_map.size() > 2){

                int tmp = fruits[i];

                if(--m\_map[tmp] == 0) m\_map.erase(tmp);

                i++;

            }

            count = j - i + 1;

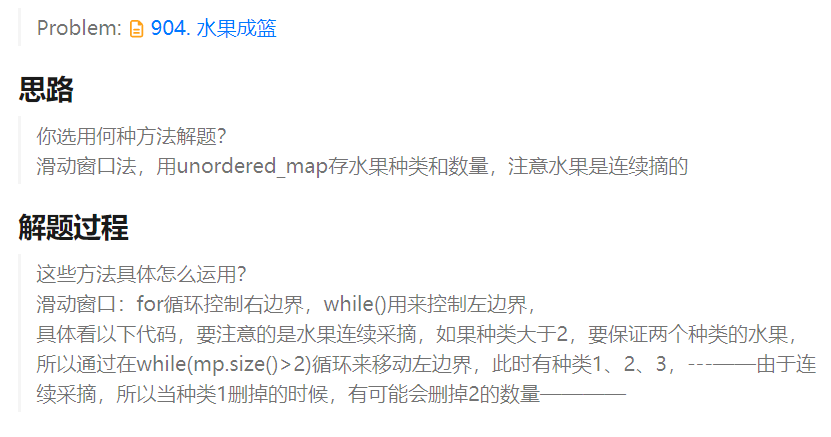
            ans = count < ans ? ans : count;

        }

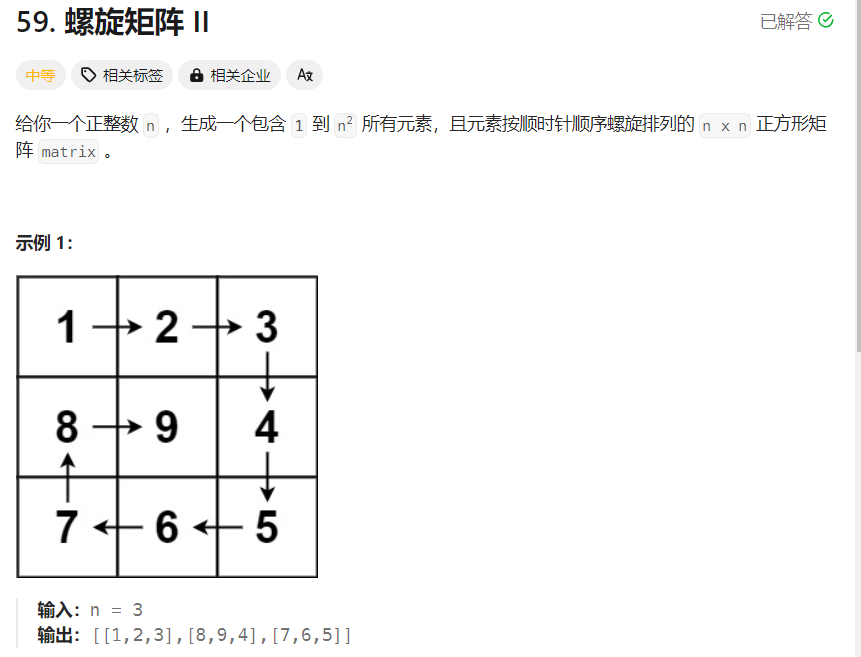
        return ans;

    }

};

思路：

1. 螺旋矩阵2 这个题要注意必须给res进行初始化！



class Solution {

public:

    vector<vector<int>> generateMatrix(int n) {

        vector<vector<int>> res(n, vector<int>(n, 0));

        int up = 0, down = n - 1, left = 0, right = n -1;

        int count = 1;

        while(true){

            for(int i = left; i <= right; i++){

                res[up][i] = count++;

            }

            up++;

            if(up > down) break;

            for(int i = up; i <= down; i++){

                res[i][right]= count++;

            }

            right--;

            if(right < left)break;

            for(int i =right; i>=left; i--){

                res[down][i] =count++;

            }

            down--;

            if(down < up)break;

            for(int i = down; i >= up; i--){

                res[i][left]=count++;

            }

            left++;

            if(left > right)break;

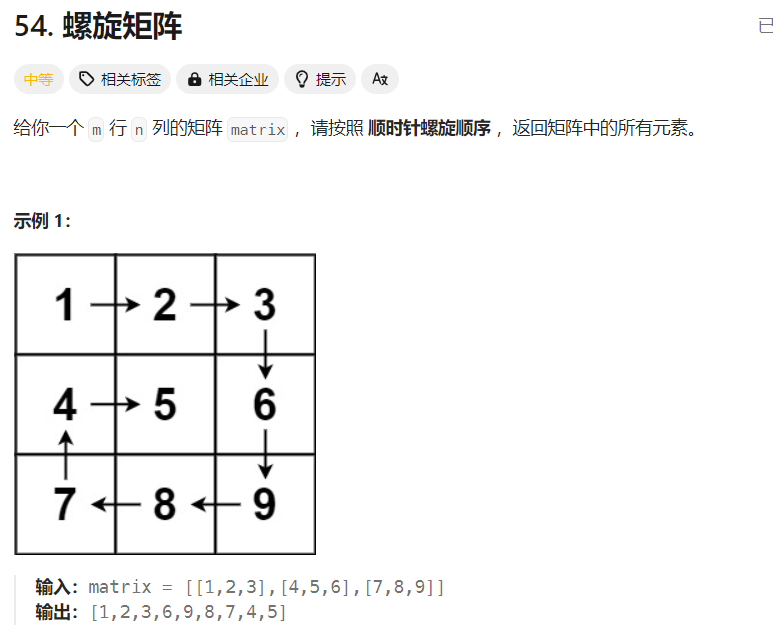
        }

        return res;

    }

};

1. 54：螺旋矩阵 注意行和列是怎么求出来的



class Solution {

public:

    vector<int> spiralOrder(vector<vector<int>>& matrix) {

        int m = matrix.size();

        int n = matrix[0].size();

        int up = 0, down = m-1, left = 0, right=n-1;

        vector<int> res;

        while(true){

            for(int i = left; i <= right; i++)  res.push\_back(matrix[up][i]);

            up++;

            if(up > down)break;

            for(int i = up; i <=down; i++)  res.push\_back(matrix[i][right]);

            right--;

            if(right < left)break;

            for(int i = right; i>=left; i--)    res.push\_back(matrix[down][i]);

            down--;

            if(down < up)break;

            for(int i = down; i>=up; i--)   res.push\_back(matrix[i][left]);

            left++;

            if(left > right)break;

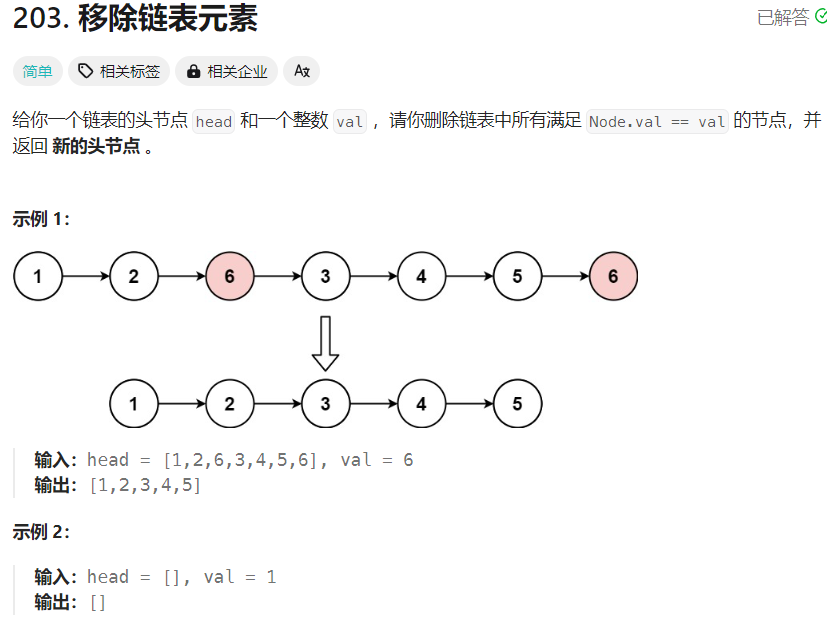
        }

        return res;

    }

};

1. 203：移除链表元素



class Solution {

public:

    ListNode\* removeElements(ListNode\* head, int val) {

        if(head == nullptr) return head;

        ListNode\* dummyHead = new ListNode(0);

        dummyHead->next = head;

        ListNode\* cur = dummyHead;

        while(cur->next != nullptr){

            if(cur->next->val == val){

                ListNode\* tmp = cur->next;

                cur->next = cur->next->next;

                delete tmp;

            }else{

                cur = cur->next;

            }

        }

        return dummyHead->next;

    }

};

1. 设计链表



class MyLinkedList {

    struct ListedNode {

        ListedNode(int i) : val(i), next(nullptr) {}

        int val;

        ListedNode\* next;

    };

public:

    //下标是从0开始的

    MyLinkedList() : size\_(0), dummyHead(new ListedNode(0)){}

    int get(int index){

        if(index > size\_ -1  || index < 0)   return -1;

        ListedNode\* cur = dummyHead->next;

        while(index--){

            cur = cur->next;

        }

        return cur->val;

    }

    void addAtHead(int val){

        ListedNode\* newNode = new ListedNode(val);

        newNode->next = dummyHead->next;

        dummyHead->next = newNode;

        size\_++;

    }

    void addAtTail(int val){

        ListedNode\* newNode = new ListedNode(val);

        ListedNode\* cur = dummyHead;

        int count = size\_;

        while(count--){

            cur = cur->next;

        }

        cur->next = newNode;

        size\_++;

    }

    void addAtIndex(int index, int val){

        if(index > size\_ ) return;

        ListedNode\* newNode = new ListedNode(val);

        ListedNode\* cur = dummyHead;

        while(index--){

            cur = cur->next;

        }

        newNode->next = cur->next;

        cur->next = newNode;

        size\_++;

    }

    void deleteAtIndex(int index){

        if(index > size\_ - 1)   return;

        ListedNode\* cur = dummyHead;

        while(index--){

            cur = cur->next;

        }

        ListedNode\* tmp = cur->next;

        cur->next = cur->next->next;

        delete tmp;

        size\_--;

    }

private:

    int size\_;

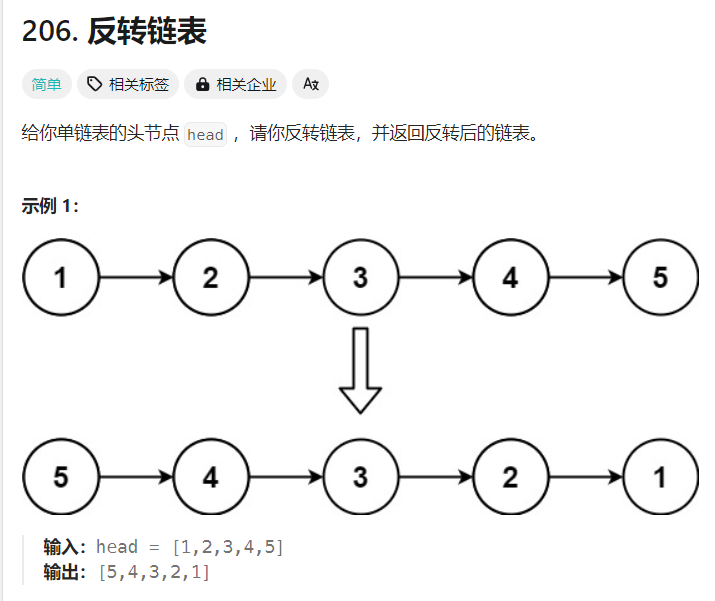
    ListedNode\* dummyHead;

};

注意

在这里最容易弄错的就是，下标是从0开始的，不是从1开始，我错在认为是1开始的下标，同时在addAtIndex的时候要注意判断条件是if(index > size\_ ) return;而不是if(index > size\_ - 1) return;这是因为有一个插入末尾的情况。

1. 206 翻转链表



class Solution {

public:

    ListNode\* reverseList(ListNode\* head) {

        ListNode\* pre = nullptr;

        ListNode\* cur = head;

        ListNode\* tmp;

        while(cur){

            tmp = cur->next;

            cur->next = pre;

            pre = cur;

            cur = tmp;

        }

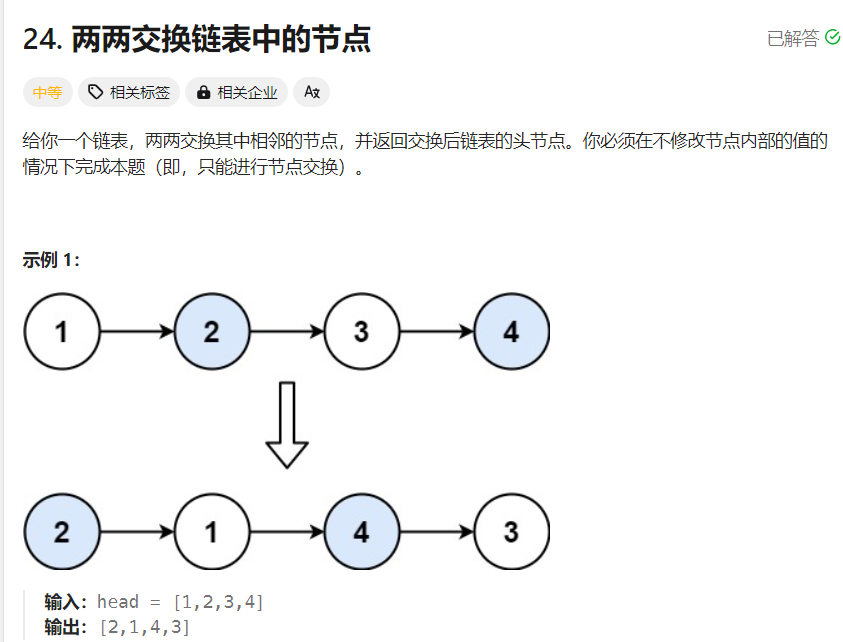
        return pre;

    }

};

易错点是tmp要在循环内进行初始化而不是声明的时候

1. 24：两两交换链表节点



class Solution {

public:

    ListNode\* swapPairs(ListNode\* head) {

        ListNode\* dummyHead = new ListNode(0);

        dummyHead->next = head;

        ListNode\* cur = dummyHead;

        while(cur->next && cur->next->next){

            ListNode\* tmp = cur->next;

            ListNode\* tmp1 = cur->next->next->next;

            cur->next = cur->next->next;

            cur->next->next = tmp;

            cur->next->next->next = tmp1;

            cur = cur->next->next;

        }

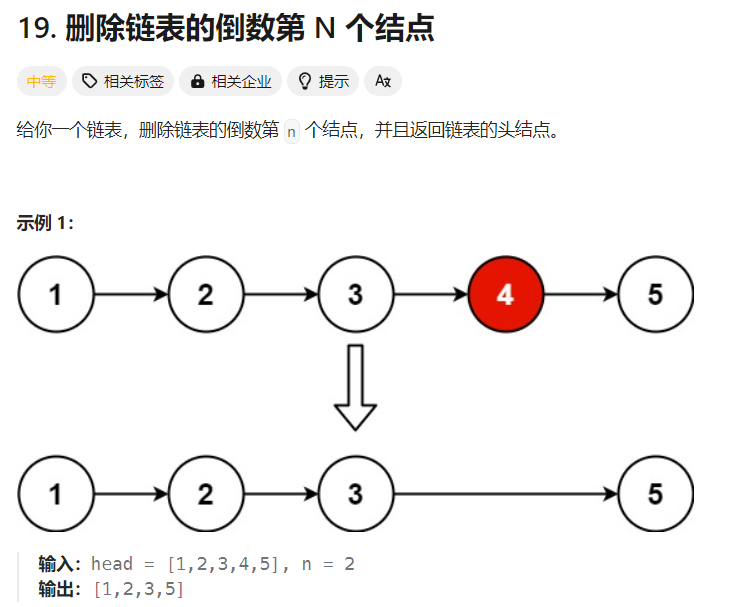
        return dummyHead->next;

    }

};

要注意的是：循环的条件和tmp、tmp1存的地址

1. 19删除链表的倒数第N个节点



class Solution {

public:

    ListNode\* removeNthFromEnd(ListNode\* head, int n) {

        ListNode\* dummyHead = new ListNode(0);

        dummyHead->next = head;

        ListNode\* cur = dummyHead;

        int count = 0;

        while(cur->next){

            count++;

            cur = cur->next;

        }

        int m = count - n;

        cur = dummyHead;

        while(m--){

            cur = cur->next;

        }

        ListNode\* tmp = cur->next;

        cur->next = cur->next->next;

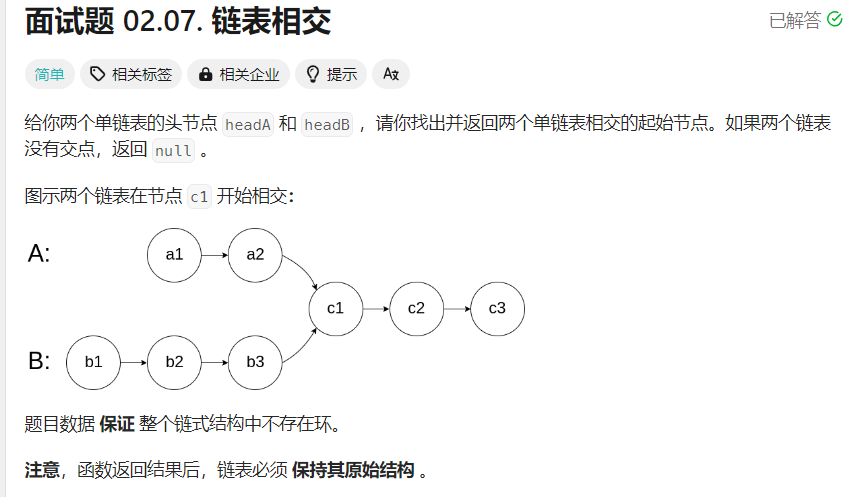
        delete tmp;

        return dummyHead->next;

    }

};

1. 链表相交



class Solution {

public:

    ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

        unordered\_set<ListNode\*> m\_set;

        ListNode\* curA = headA;

        while(curA){

            m\_set.insert(curA);

            curA = curA->next;

        }

        ListNode\* curB = headB;

        while(curB){

            if(m\_set.find(curB)!=m\_set.end())   return curB;

            curB = curB->next;

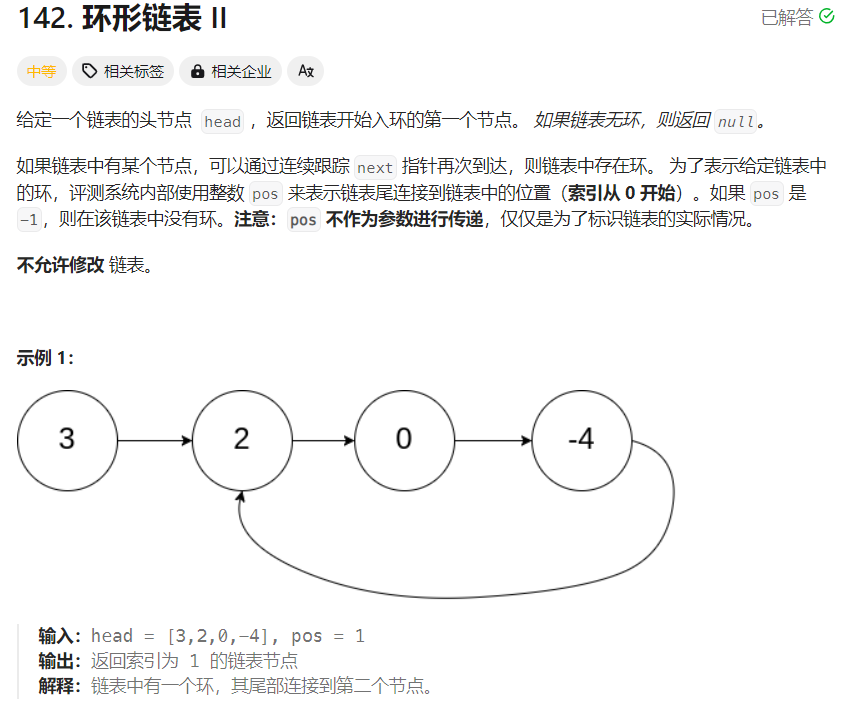
        }

        return nullptr;

    }

};

1. 142：链表相交2



class Solution {

public:

    ListNode \*detectCycle(ListNode \*head) {

        unordered\_set<ListNode\*> m\_set;

        ListNode\* cur = head;

        while(cur){

            if(m\_set.find(cur) != m\_set.end()) return cur;

            m\_set.insert(cur);

            cur = cur->next;

        }

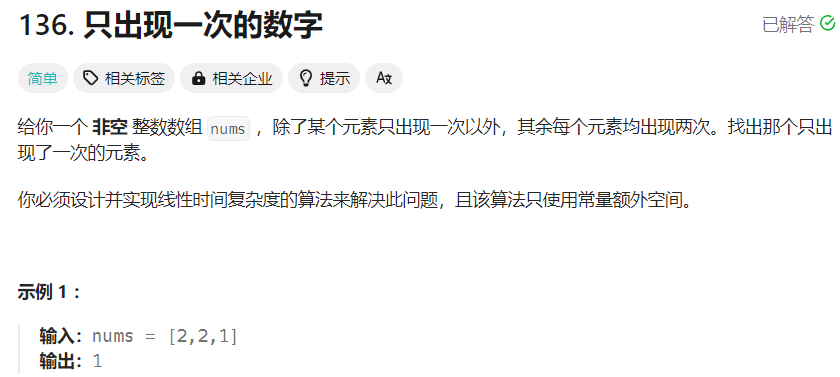
        return nullptr;

    }

};

注意别忘了让cur=cur->next;

1. 136：只出现一次的数字



class Solution {

public:

    int singleNumber(vector<int>& nums) {

        int res = 0;

        for(int n : nums){

            res ^= n;

        }

        return res;

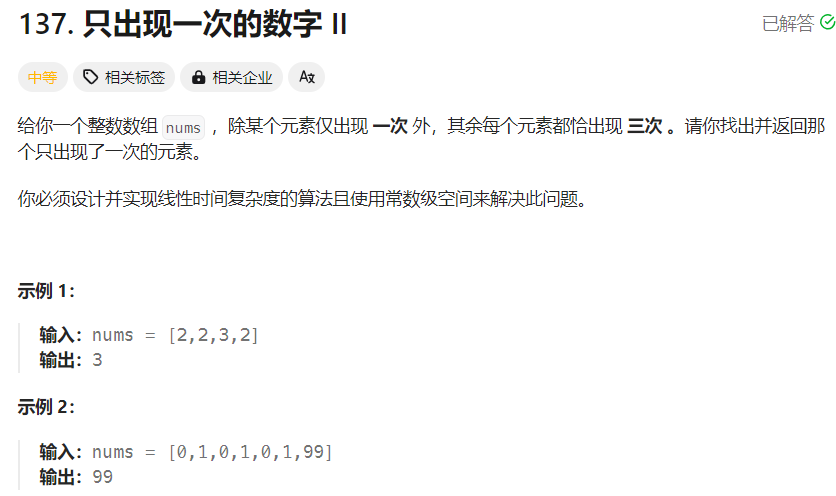
    }

};

这里用的异或操作，a ^ 0 = a; a ^ a = 0; a ^ b = c; c ^ b = a;

所以res ^ n会留下只出现一次的数字，因为其他都是两次，所以a ^ a = 0;

1. 只出现一次的数字2



class Solution {

public:

    int singleNumber(vector<int> &nums) {

        int res = 0;

        for(int i = 0; i < 32; i++){

            int count = 0;

            for(int n : nums){

                count += n >> i & 1;

            }

            if(count % 3) res = res | 1 << i;

        }

        return res;

    }

};

把每个元素转换成二进制，由于同一个数字要么有3个要么就是一个，所以转成二进制后每一位上的1%3要么为0要么为1，可以用左移和右移来记录

for(int n : nums){

count += n >> i & 1;

} 以上是算每个数字在第i位上的1的数量，n>>i是把对应第i位的数字移动到最低位，然后 & 1，1是最低位的，1 & 1为1，1&0为0，按照这个策略计算数量。 if(count % 3) res = res | 1 << i; 这里是让1 << i，也就是让1 左移回到原来的位置，然后和res进行或操作

1. 有效的字母异位词



class Solution {

public:

    bool isAnagram(string s, string t) {

        int sSize = s.size();

        int tSize = t.size();

        if(sSize != tSize)  return false;

        unordered\_map<char, int> m\_map;

        for(char ch = 'a'; ch <= 'z'; ch++){

            m\_map[ch]=0;

        }

        for(int i = 0; i < sSize; i++){

            m\_map[s[i]]++;

        }

        for(int i = 0; i < tSize; i++){

            m\_map[t[i]]--;

        }

        for(auto it : m\_map){

            if(it.second != 0)  return false;

        }

        return true;

    }

};

思路：用m\_map获取a 到 z 的键值对，然后对s进行++，对t进行--，在进行对应处理

1. 383：赎金信



class Solution {

public:

    bool canConstruct(string ransomNote, string magazine) {

        int rSize = ransomNote.size();

        int mSize = magazine.size();

        if(rSize > mSize) return false;

        unordered\_map<char, int> m\_map;

        for(char ch = 'a'; ch <= 'z'; ch++){

            m\_map[ch] = 0;

        }

        for(int i = 0; i < rSize; i++){

            m\_map[ransomNote[i]]++;

        }

        for(int i = 0; i< mSize; i++){

            m\_map[magazine[i]]--;

        }

        for(auto it : m\_map){

            if(it.second > 0) return false;

        }

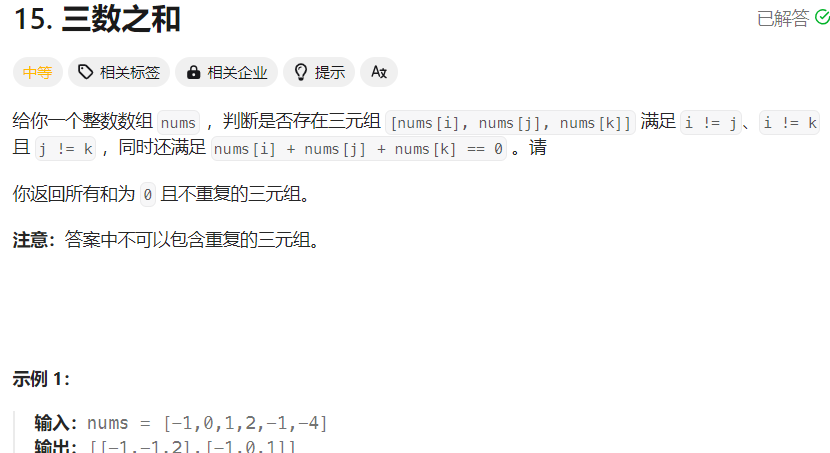
        return true;

    }

};

别搞错返回条件了，一开始if(rSize > mSize) return false;写成if(rSize < mSize) return false;

1. 15:三数之和



class Solution {

public:

    vector<vector<int>> threeSum(vector<int>& nums) {

        //先确定一个，然后用左右指针确定后面两个

        sort(nums.begin(), nums.end());

        vector<vector<int>> res;

        for(int i = 0; i < nums.size(); i++){

            if(i > 0 && nums[i] == nums[i-1]) continue;

            int tmp = 0 - nums[i];

            for(int left = i + 1, right = nums.size() - 1; left < right;){

                if(tmp == nums[left] + nums[right]){

                    res.push\_back({nums[i], nums[left], nums[right]});

                    while(left < right && nums[left] == nums[left+1])left++;

                    left++;

                    while(left < right && nums[right] == nums[right-1])right--;

                    right--;

                }else if(tmp < nums[left] + nums[right]){

                    right--;

                }else{

                    left++;

                }

            }

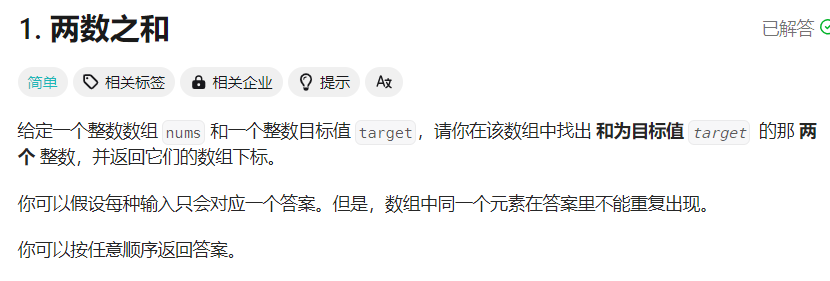
        }

        return res;

    }

};

1. 1：两数之和



class Solution {

public:

    vector<int> twoSum(vector<int>& nums, int target) {

        unordered\_map<int,int> m\_map;

        for(int i = 0; i < nums.size(); i++){

            if(m\_map.find(target - nums[i]) != m\_map.end()){

                return {i, m\_map[target-nums[i]]};

            }else{

                m\_map.insert(pair<int,int>(nums[i],i));

            }

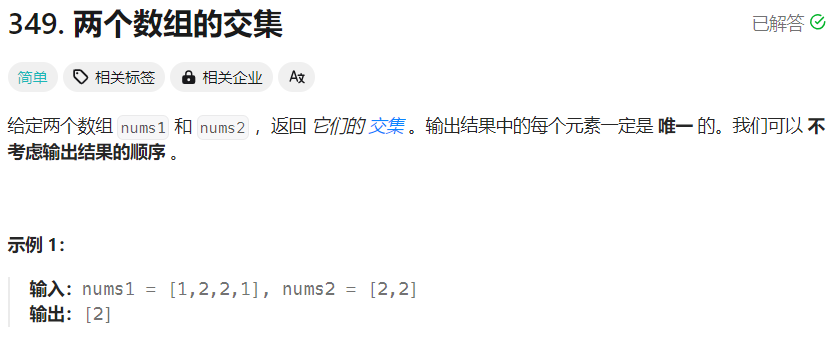
        }

        return {-1, -1};

    }

};

1. 349：两个数组的交集



class Solution {

public:

    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {

        unordered\_set<int> m\_set(nums1.begin(), nums1.end());

        unordered\_set<int> tmp(nums2.begin(), nums2.end());

        vector<int> res;

        for(int n : tmp){

            if(m\_set.find(n)!=m\_set.end()){

                res.push\_back(n);

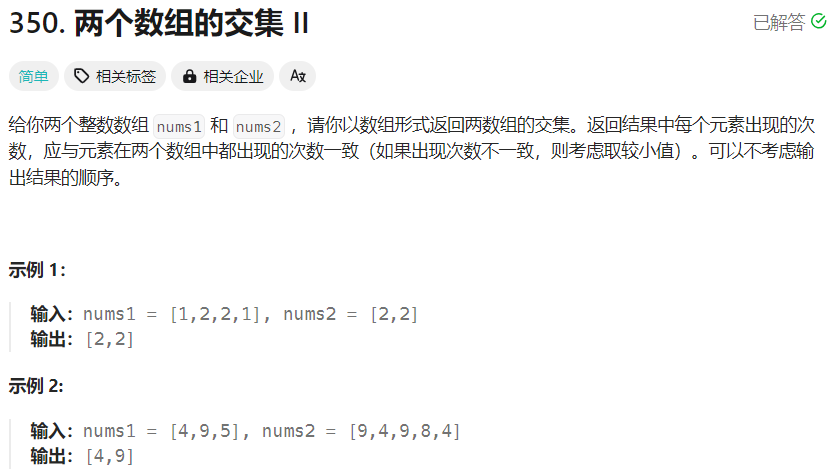
            }

        }

        return res;

    }

};

1. 350：

class Solution {

public:

    vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {

        int size1 = nums1.size();

        int size2 = nums2.size();

        vector<int> res;

        if(size1 < size2) return intersect(nums2, nums1);

        unordered\_map<int ,int> m\_map;

        for(int n : nums2)  m\_map[n]++;

        for(int i = 0; i < nums1.size(); i++){

            if(m\_map.find(nums1[i])!=m\_map.end() && m\_map[nums1[i]]>0){

                res.push\_back(nums1[i]);

                m\_map[nums1[i]]--;

            }

        }

        return res;

    }

};

找到size更小的那个容器，然后用m\_map保存值和出现次数，之后去另一个数组找对应出现的值，同时要进行对应的--操作，注意对应值得数量要>0才行

1. 202：快乐数



class Solution {

public:

    bool isHappy(int n) {

        unordered\_set<int> m\_set;

        while(true){

            int tmp = getSum(n);

            if(tmp == 1){

                return true;

            }else if(m\_set.find(tmp)!=m\_set.end()){

                return false;

            }else{

                m\_set.insert(tmp);

                n=tmp;

            }

        }

        return false;

    }

    int getSum(int n){

        int sum = 0;

        while(n){

            int tmp = n % 10;

            sum += tmp \* tmp;

            n/=10;

        }

        return sum;

    }

};

1. 18：四数之和



class Solution {

public:

    vector<vector<int>> fourSum(vector<int>& nums, int target) {

        vector<vector<int>> res;

        sort(nums.begin(), nums.end());

        for(int i = 0; i < nums.size(); i++){

            if(i > 0 && nums[i] == nums[i-1])    continue;

            for(int j = i + 1; j < nums.size(); j++){

                if(j > i+1 && nums[j] == nums[j-1]) continue;

                int left = j + 1;

                int right = nums.size() - 1;

                long tmp = (long)target - nums[i] - nums[j];

                while(left<right){

                    if(tmp == nums[left] + nums[right]){

                        res.push\_back({nums[i], nums[j], nums[left], nums[right]});

                        while(left < right && nums[left]==nums[left+1]) left++;

                        left++;

                        while(left < right && nums[right] == nums[right-1]) right--;

                        right--;

                    }else if(tmp < nums[left] + nums[right]){

                        right--;

                    }else{

                        left++;

                    }

                }

            }

        }

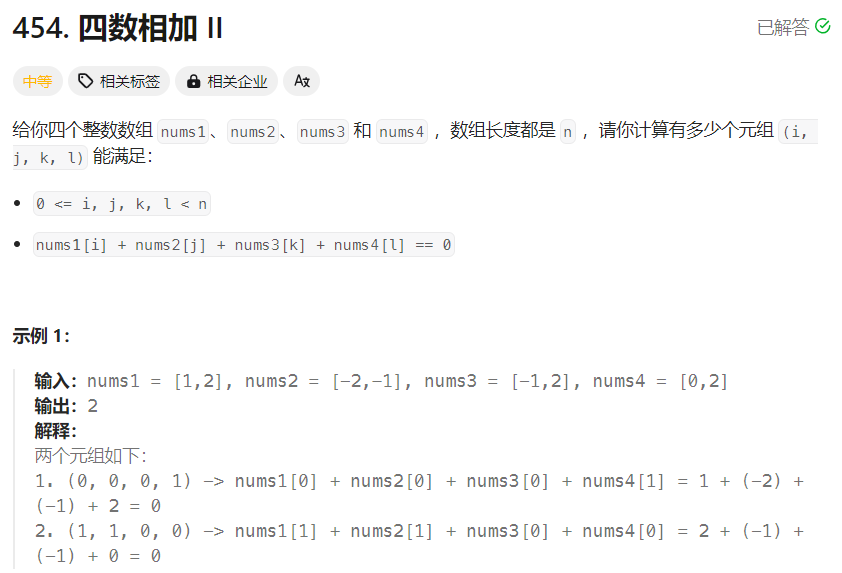
        return res;

    }

};

思路：先用sort去给数组进行排序，这点很重要，然后确定第一个和第二个数字，再确定后面的两个数字

1. 454：四数相加2



class Solution {

public:

    int fourSumCount(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3, vector<int>& nums4) {

        //先把两个数组中所有数字的和放到unordered\_map里，key为和,value为同一个数字出现的次数

        //然后在剩下两个数组中找有多少和m\_map中的键相加为0的值，然后count加上key的次数

        unordered\_map<int,int> m\_map;

        int count = 0;

        for(int n : nums1){

            for(int k : nums2){

                m\_map[n+k]++;

            }

        }

        for(int n : nums3){

            for(int k : nums4){

                if(m\_map.find(0 - n - k) != m\_map.end()){

                    count += m\_map[0 - n - k];

                }

            }

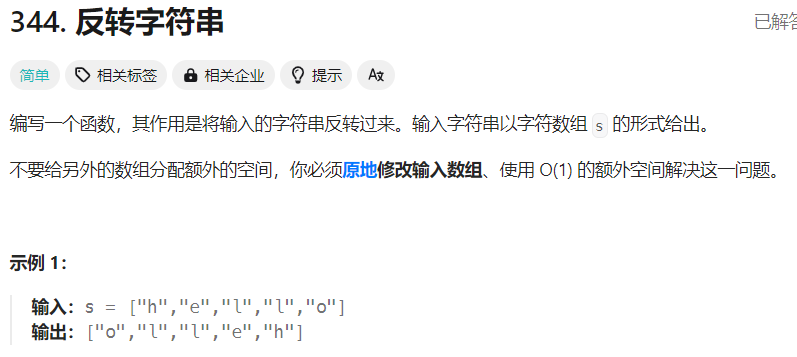
        }

        return count;

    }

};

1. 344：翻转字符串



class Solution {

public:

    void reverseString(vector<char>& s) {

        for(int slow = 0, fast = s.size()-1; slow < fast; slow++, fast--){

            s[slow] = s[slow] ^ s[fast];

            s[fast] = s[slow] ^ s[fast];

            s[slow] = s[slow] ^ s[fast];

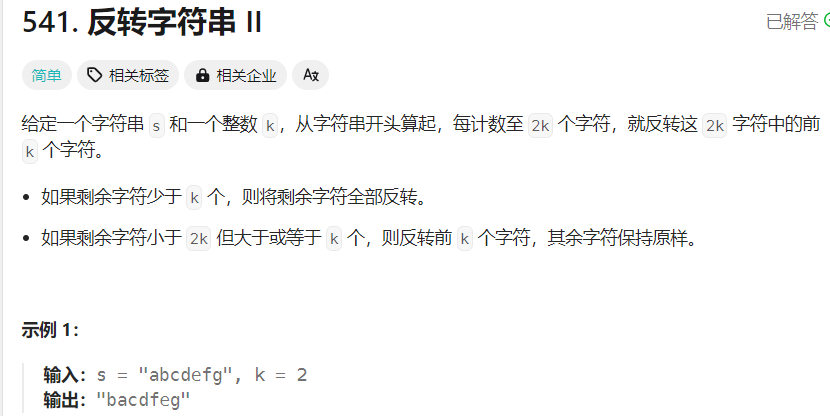
        }

    }

};

可以使用双指针，也可以使用异或操作

1. 翻转字符串2



class Solution {

public:

    string reverseStr(string s, int k) {

        int size = s.size();

        for(int i = 0; i < size; i += 2 \* k){

            if(size - i >= 2 \* k){

                reverse(s.begin() + i, s.begin() + i+k);

            }else if(size - i >= k){

                reverse(s.begin()+i, s.begin() + i+k);

            }else{

                reverse(s.begin() + i, s.begin() + size);

            }

        }

        return s;

    }

};

第二种解法：

class Solution {

public:

string reverseStr(string s, int k) {

int length = s.size();

for(int i = 0; i < length; i += 2\*k){

reverse(s.begin()+i, s.begin() + min(i+k, length));

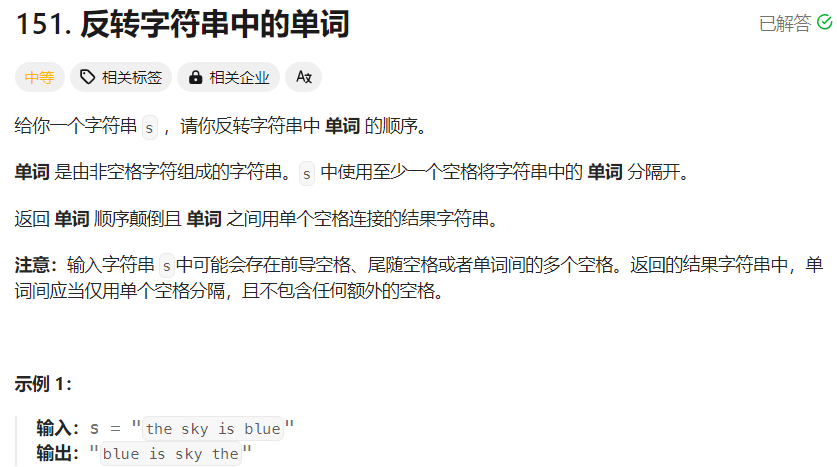
}

return s;

}

};

1. 151：反转字符串中的单词



第一种解法：

class Solution {

public:

    string reverseWords(string s) {

        stack<string> m\_str;

        string res = "";

        for(int i=0; i < s.size(); i++){

            if(s[i] != ' '){

                res += s[i];

            }else if(res != ""){

                m\_str.push(res);

                res = "";

            }

        }

        if(res != "")

            m\_str.push(res);

        string m\_res = "";

        while(m\_str.size() != 0){

            m\_res += m\_str.top();

            m\_str.pop();

            m\_res.push\_back(' ');

        }

        m\_res.pop\_back();

        return m\_res;

    }

};

第二种解法：

class Solution {

public:

string reverseWords(string s) {

vector<string> res;

string ss;

string tmp = "";

//把单词一个一个装到tmp，然后放到res中

for(int i = 0; i < s.size(); i++){

if(s[i]!= ' '){

tmp += s[i];

}else if(tmp != ""){

res.push\_back(tmp);

tmp = "";//要将tmp置空

}

}

if(tmp != "") res.push\_back(tmp);//要注意最后一个单词后面没有空格，所以循环后单独加到res

for(int i = res.size()-1; i >=0; i--){//注意是 i>=0，如果是i>0就会少第一个单词

ss += res[i];

//加空格

if(i != 0){

ss += ' ';

}

}

return ss;

}

};

1. 28：找出字符串第一个匹配项的下标



class Solution {

public:

    int strStr(string haystack, string needle) {

        int hSize = haystack.size();

        int nSize = needle.size();

        if(hSize < nSize)   return -1;

        for(int i = 0; i <= hSize - nSize; i++){

            string tmp = "";

            for(int j = i; j < i + nSize; j++){

                tmp += haystack[j];

            }

            if(tmp == needle)return i;

        }

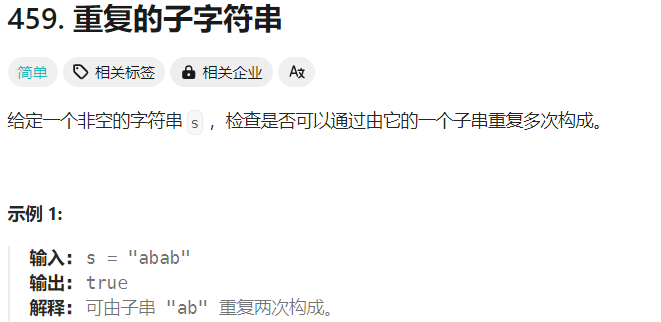
        return -1;

    }

};

这里主要出错的就是边界问题，I 和 ｊ容易弄错边界

1. 459：重复的子字符串



class Solution {

public:

    bool repeatedSubstringPattern(string s) {

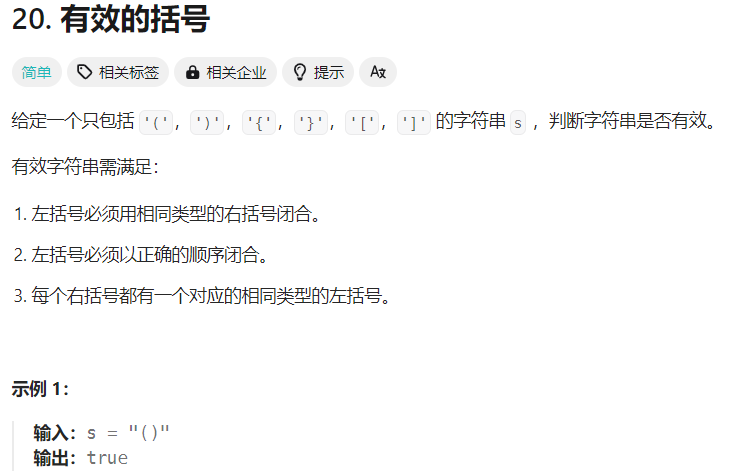
        return (s + s).find(s, 1) != s.size();

    }

};

思路：如果是重复字符串，那么(s + s)也是重复字符串，所以find(s, 1)得到的下标就不会为ｓ的开始，也就是s.size()

1. 20：有效的括号



class Solution {

public:

    bool isValid(string s) {

        stack<char> m\_stack;

        for(int i = 0; i < s.size(); i++){

            if(s[i] == '(') m\_stack.push(')');

            else if(s[i] == '[') m\_stack.push(']');

            else if(s[i] == '{') m\_stack.push('}');

            else if(m\_stack.empty() || s[i] != m\_stack.top())   return false;

            else{

                m\_stack.pop();

            }

        }

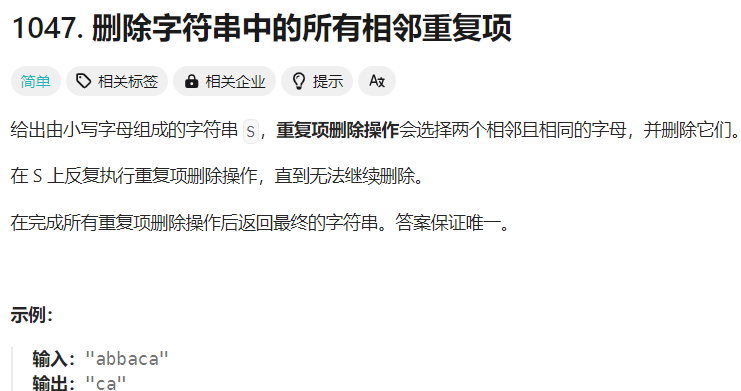
        return m\_stack.empty();

    }

};

注意返回的是m\_stack.empty()

1. １０４７：删除字符串中的所有相邻重复项



class Solution {

public:

    string removeDuplicates(string s) {

        stack<char> m\_stack;

        string res ="";

        for(int i = 0; i < s.size(); i++){

            if(m\_stack.empty()){

                m\_stack.push(s[i]);

            }else if(!m\_stack.empty() && m\_stack.top() == s[i]){

                m\_stack.pop();

            }else{

                m\_stack.push(s[i]);

            }

        }

        while(m\_stack.size() != 0){

            res.push\_back(m\_stack.top());

            m\_stack.pop();

        }

        reverse(res.begin(), res.end());

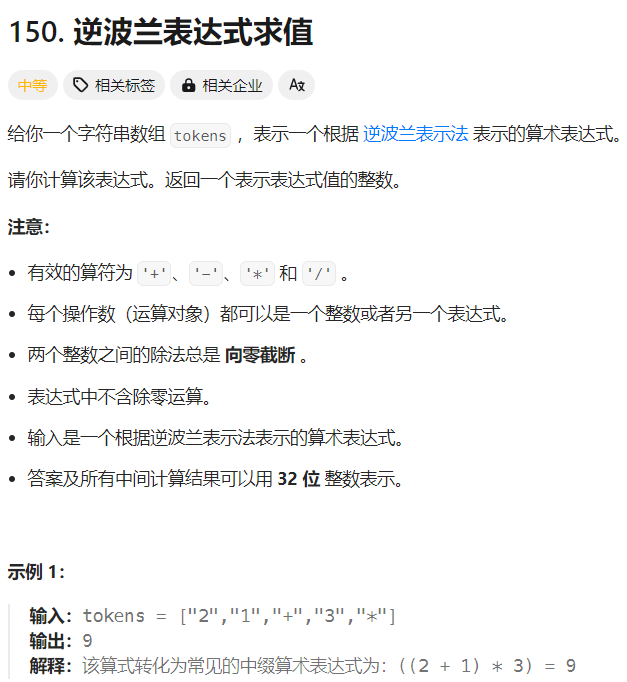
        return res;

    }

};

主要用栈来解决

1. 150：逆波兰表达式求值



class Solution {

public:

    int evalRPN(vector<string>& tokens) {

        stack<int> ss;

        for(int i = 0; i < tokens.size(); i++){

            int left=0,right=0, sum=0;

            if(tokens[i] == "+"){

                right = ss.top();

                ss.pop();

                left = ss.top();

                ss.pop();

                sum = left + right;

                ss.push(sum);

            }else if(tokens[i] == "-"){

                right = ss.top();

                ss.pop();

                left = ss.top();

                ss.pop();

                sum = left - right;

                ss.push(sum);

            }else if(tokens[i] == "\*"){

                right = ss.top();

                ss.pop();

                left = ss.top();

                ss.pop();

                sum = left \* right;

                ss.push(sum);

            }else if(tokens[i] == "/"){

                right = ss.top();

                ss.pop();

                left = ss.top();

                ss.pop();

                sum = left / right;

                ss.push(sum);

            }else{

                ss.push(stoi(tokens[i]));

            }

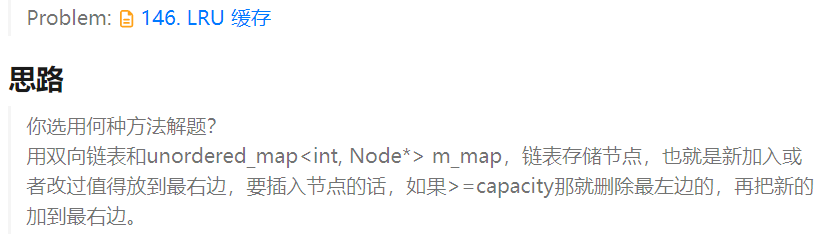
        }

        return ss.top();

    }

};

1. 146:LRU缓存



class LRUCache {

public:

    struct Node {

        int key;

        int val;

        Node\* pre;

        Node\* next;

        Node() : key(0), val(0), pre(nullptr), next(nullptr) {}

        Node(int k, int v) : key(k), val(v), pre(nullptr), next(nullptr) {}

    };

    int capacity;

    std::unordered\_map<int, Node\*> m\_map;

    Node\* head;

    Node\* tail;

    LRUCache(int capacity) {

        this->capacity = capacity;

        head = new Node();

        tail = new Node();

        head->next = tail;

        tail->pre = head;

    }

    int get(int key) {

        if (m\_map.find(key) != m\_map.end()) {

            Node\* node = m\_map[key];

            moveToTail(node);

            return node->val;

        }

        return -1;

    }

    void put(int key, int value) {

        if (m\_map.find(key) != m\_map.end()) {

            Node\* node = m\_map[key];

            node->val = value;

            moveToTail(node);

        } else {

            if (m\_map.size() >= capacity) {

                removeHead();

            }

            Node\* node = new Node(key, value);

            m\_map[key] = node;

            addToTail(node);

        }

    }

    void moveToTail(Node\* node) {

        // Remove node from current position

        node->pre->next = node->next;

        node->next->pre = node->pre;

        // Insert node at the tail

        node->next = tail;

        node->pre = tail->pre;

        tail->pre->next = node;

        tail->pre = node;

    }

    void removeHead() {

        Node\* node = head->next;

        head->next = node->next;

        node->next->pre = head;

        m\_map.erase(node->key);

        delete node;

    }

    void addToTail(Node\* node) {

        node->next = tail;

        node->pre = tail->pre;

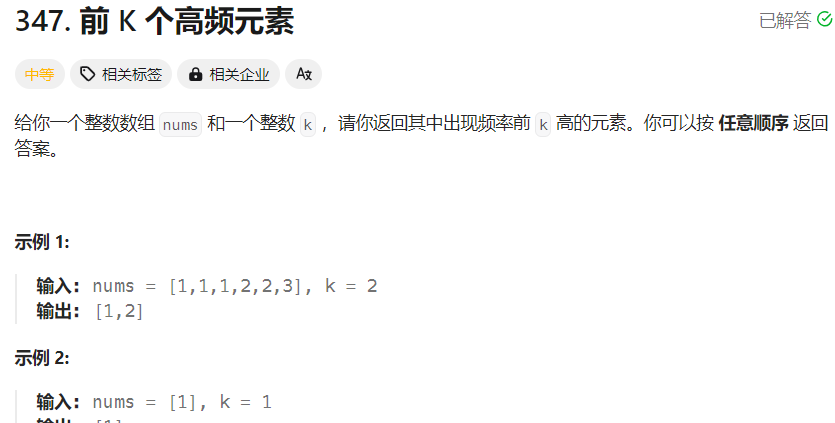
        tail->pre->next = node;

        tail->pre = node;

    }

};

1. 347：前K个高频元素



class Solution {

public:

    class small{

        public:

        bool operator()(const pair<int,int>& l, const pair<int,int>& r){

            return l.second > r.second;//优先级低的在堆顶，优先出队列

        }

    };

    vector<int> topKFrequent(vector<int>& nums, int k) {

        unordered\_map<int,int> m\_map;

        for(int n : nums){

            m\_map[n]++;

        }

        priority\_queue<pair<int,int>, vector<pair<int,int>>, small> m\_que;

        for(auto it = m\_map.begin(); it != m\_map.end(); ++it){

            m\_que.push(pair<int,int>(it->first, it->second));

            if(m\_que.size() > k){

                m\_que.pop();

            }

        }

        vector<int> res;

        for(int i = 0; i < k; i++){

            res.push\_back(m\_que.top().first);

            m\_que.pop();

        }

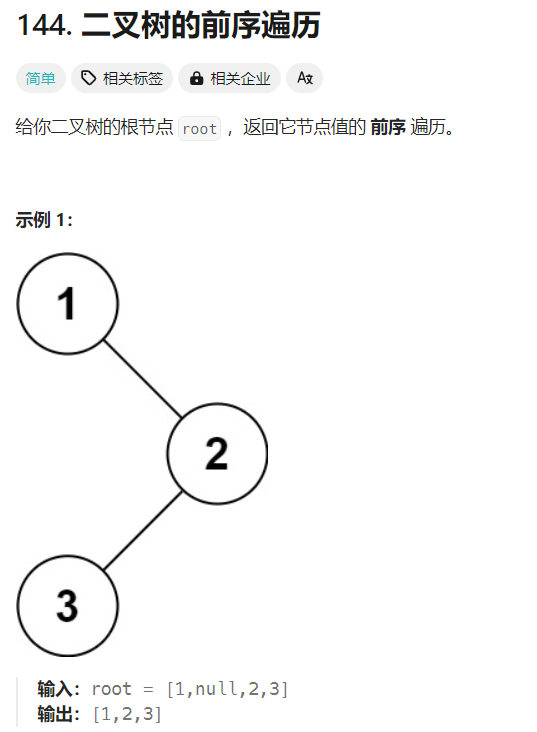
        return res;

    }

};

要注意小顶堆的设置，也就是仿函数

1. 144：二叉树的前序遍历



class Solution {

    void treeTraversal(TreeNode\* root, vector<int>& res){

        if(root == nullptr) return;

        //中

        res.push\_back(root->val);

        //左

        treeTraversal(root->left, res);

        //右

        treeTraversal(root->right, res);

    }

public:

    vector<int> preorderTraversal(TreeNode\* root) {

        vector<int> res;

        //treeTraversal(root, res);//递归法

        //迭代法  前序遍历是中左右，所以一开始根节点放进去之后处理要让右节点先入栈，这样才能保证中左右吗，不然就是中右左了

        stack<TreeNode\*> m\_stack;

        if(root == nullptr) return res;

        m\_stack.push(root);

        while(m\_stack.size()!=0){

            TreeNode\* tmp = m\_stack.top();

            res.push\_back(tmp->val);

            m\_stack.pop();

            if(tmp->right != nullptr)   m\_stack.push(tmp->right);

            if(tmp->left != nullptr)    m\_stack.push(tmp->left);

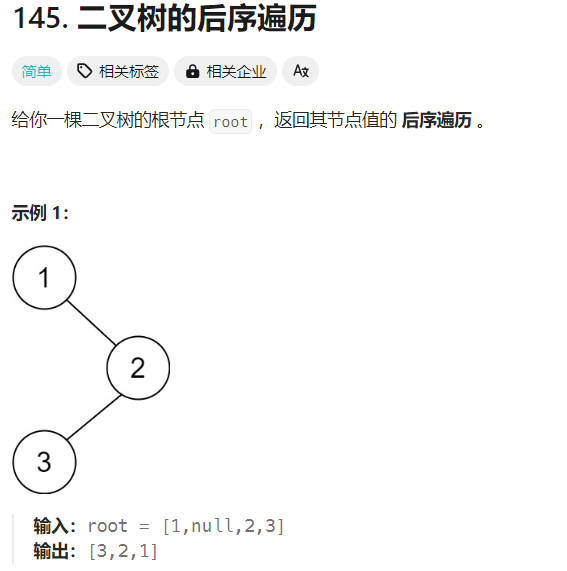
        }

        return res;

    }

};

1. 145：二叉树的后序遍历



class Solution {

    void treeTraversal(TreeNode\* root, vector<int>& res){

        if(root == nullptr) return;

        //左

        treeTraversal(root->left, res);

        //右

        treeTraversal(root->right, res);

        //中

        res.push\_back(root->val);

    }

public:

    vector<int> postorderTraversal(TreeNode\* root) {

        vector<int> res;

        //treeTraversal(root, res);//递归法

        //以下是迭代法 前序是中左右，后序是左右中，所以可以先中右左，存到res后，逆序排列res，就是左右中

        if(root == NULL) return res;

        stack<TreeNode\*> m\_stack;

        m\_stack.push(root);

        while(!m\_stack.empty()){

            TreeNode\* node = m\_stack.top();

            m\_stack.pop();

            res.push\_back(node->val);

            //中右左,所以先放左再放右

            if(node->left)  m\_stack.push(node->left);

            if(node->right) m\_stack.push(node->right);

        }

        //最后要翻转数组，不然就是中右左

        reverse(res.begin(), res.end());

        return res;

    }

};

1. 94：二叉树的中序遍历



class Solution {

public:

    void treeTraversal(TreeNode\* root, vector<int>& res){

        if(root == nullptr) return;

        //左

        treeTraversal(root->left, res);

        //中

        res.push\_back(root->val);

        //右

        treeTraversal(root->right, res);

    }

    vector<int> inorderTraversal(TreeNode\* root) {

        vector<int> res;

        //treeTraversal(root, res);//递归法

        //迭代法，中序是 左中右

        stack<TreeNode\*> m\_stack;

        TreeNode\* cur = root;

        while(cur != NULL || !m\_stack.empty()){

            if(cur != NULL){

                m\_stack.push(cur);

                cur = cur->left;//左

            }else{

                cur = m\_stack.top();

                m\_stack.pop();

                res.push\_back(cur->val);//中

                cur = cur->right;

            }

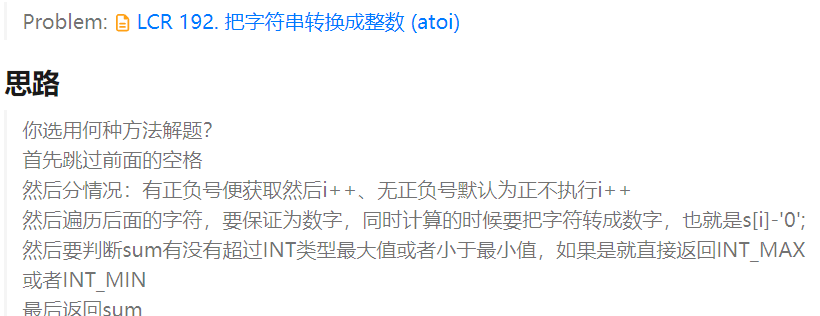
        }

        return res;

    }

};

1. 192：将字符串转成整数



class Solution {

public:

    int myAtoi(string str) {

        int size = str.size();

        int i = 0;

        while(str[i]==' ') i++;

        int flag = 1;

        if(str[i] == '+'){

            i++;

        }else if(str[i] == '-'){

            flag = -1;

            i++;

        }

        long sum = 0;

        for( ; i < size; i++){

            if(str[i] < '0' || str[i] > '9'){

                return sum;

            }

            sum = sum \* 10 + flag \* (str[i] - '0');

            if(sum >= INT\_MAX && flag == 1) return INT\_MAX;

            if(sum < INT\_MIN && flag == -1) return INT\_MIN;

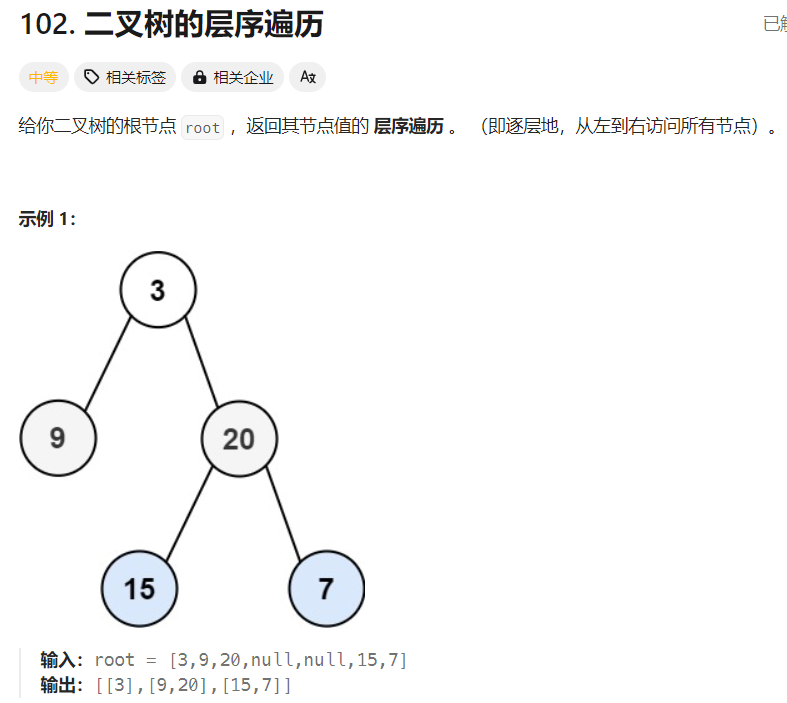
        }

        return sum;

    }

};

1. 102：二叉树的层序遍历



class Solution {

public:

    vector<vector<int>> levelOrder(TreeNode\* root) {

        vector<vector<int>> res;

        if(root == nullptr) return res;

        queue<TreeNode\*> m\_que;

        m\_que.push(root);

        while(!m\_que.empty()){

            vector<int> m\_res;

            int size = m\_que.size();

            for(int i = 0; i < size; i++){

                TreeNode\* tmp = m\_que.front();

                m\_que.pop();

                m\_res.push\_back(tmp->val);

                if(tmp->left != nullptr)    m\_que.push(tmp->left);

                if(tmp->right != nullptr)   m\_que.push(tmp->right);

            }

            res.push\_back(m\_res);

        }

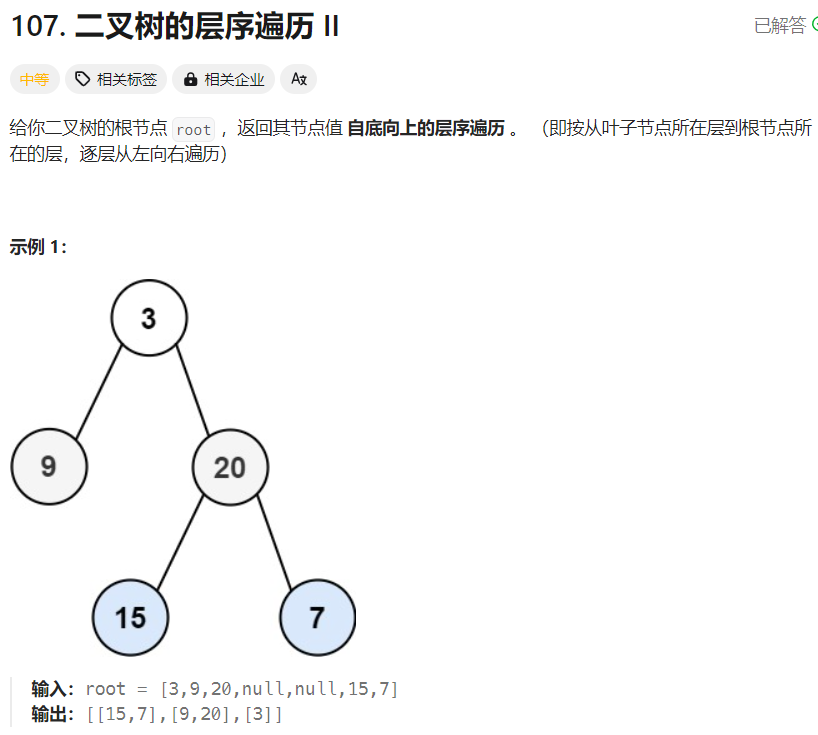
        return res;

    }

};

注意，递归去进行前、中、后序遍历，递归可以做的栈也可以做，所以也可以使用栈去进行迭代遍历，进行前、中、后序遍历

1. 107：二叉树的层序遍历2



class Solution {

public:

    vector<vector<int>> levelOrderBottom(TreeNode\* root) {

        vector<vector<int>> res;

        queue<TreeNode\*> m\_que;

        if(root == nullptr) return res;

        m\_que.push(root);

        while(m\_que.size()!=0){

            vector<int> m\_res;

            int size = m\_que.size();

            for(int i = 0; i < size; i++){

                TreeNode\* tmp = m\_que.front();

                m\_que.pop();

                m\_res.push\_back(tmp->val);

                if(tmp->left != nullptr)    m\_que.push(tmp->left);

                if(tmp->right != nullptr)   m\_que.push(tmp->right);

            }

            res.push\_back(m\_res);

        }

        reverse(res.begin(), res.end());

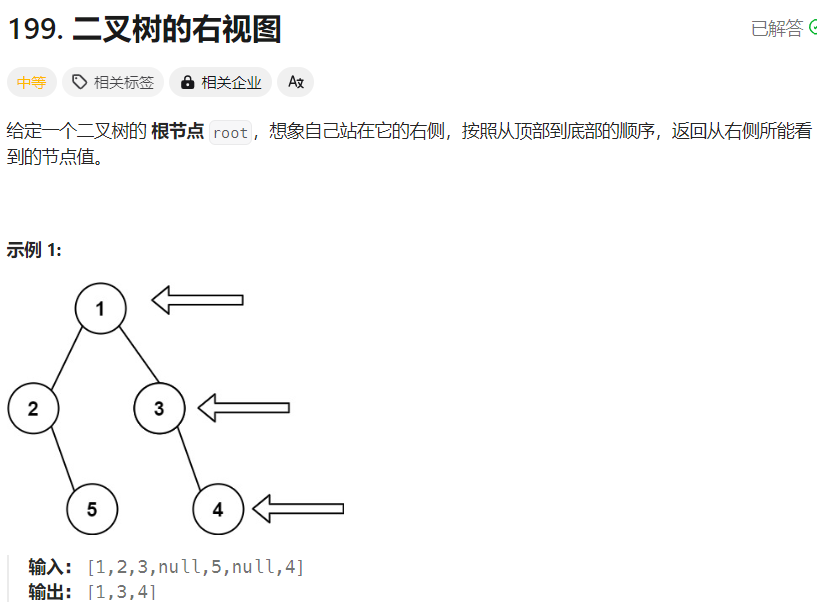
        return res;

    }

};

思路就是，层序遍历后逆序一下

1. 199:二叉树的右视图



class Solution {

public:

    vector<int> rightSideView(TreeNode\* root) {

        vector<int> res;

        if(root == nullptr) return res;

        queue<TreeNode\*> m\_que;

        m\_que.push(root);

        while(m\_que.size() != 0){

            int size = m\_que.size();

            for(int i = 0; i < size; i++){

                TreeNode\* tmp = m\_que.front();

                m\_que.pop();

                if(i == size - 1){

                    res.push\_back(tmp->val);

                }

                if(tmp->left != nullptr) m\_que.push(tmp->left);

                if(tmp->right != nullptr)   m\_que.push(tmp->right);

            }

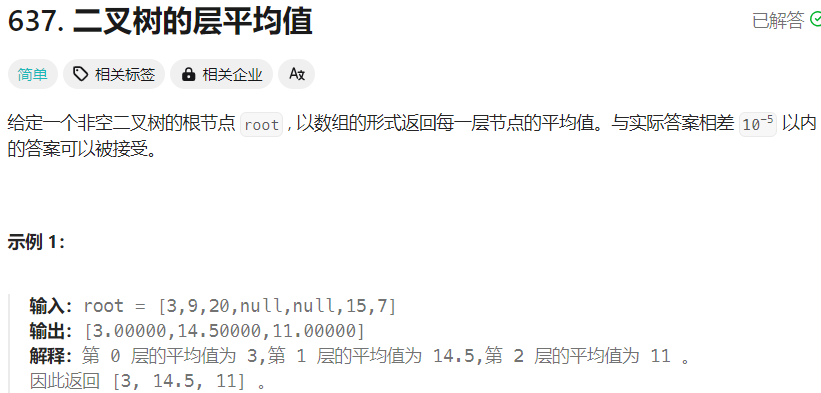
        }

        return res;

    }

};

1. 637：二叉树的层平均值



class Solution {

public:

    vector<double> averageOfLevels(TreeNode\* root) {

        vector<double> res;

        if(root == nullptr) return res;

        queue<TreeNode\*> m\_que;

        m\_que.push(root);

        while(m\_que.size() != 0){

            int size = m\_que.size();

            double sum = 0;

            for(int i = 0; i < size; i++){

                TreeNode\* tmp = m\_que.front();

                m\_que.pop();

                sum += tmp->val;

                if(tmp->left != nullptr)    m\_que.push(tmp->left);

                if(tmp->right != nullptr)   m\_que.push(tmp->right);

            }

            res.push\_back(sum / size);

        }

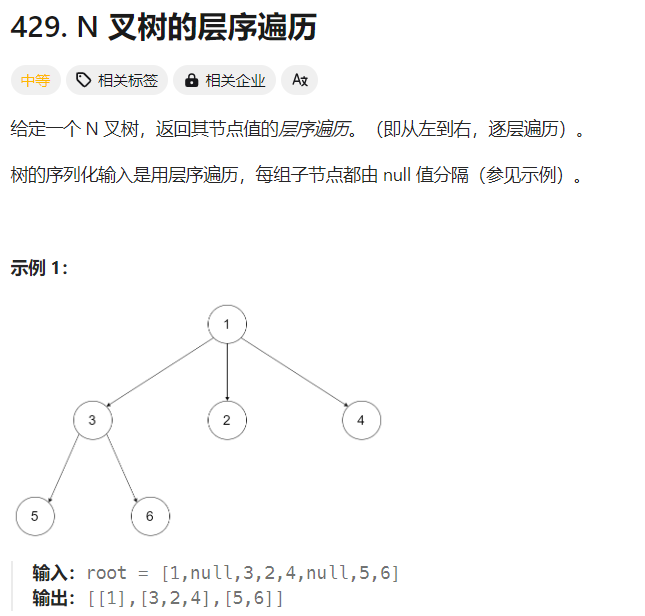
        return res;

    }

};

注意：要记得pop()

1. 429N叉树的层序遍历



class Solution {

public:

    vector<vector<int>> levelOrder(Node\* root) {

        vector<vector<int>> res;

        if(root == nullptr) return res;

        queue<Node\*> m\_que;

        m\_que.push(root);

        while(m\_que.size() != 0){

            int size = m\_que.size();

            vector<int> m\_res;

            for(int i = 0; i < size; i++){

                Node\* tmp = m\_que.front();

                m\_que.pop();

                m\_res.push\_back(tmp->val);

                for(Node\* n : tmp->children){

                    m\_que.push(n);

                }

            }

            res.push\_back(m\_res);

        }

        return res;

    }

};

注意：这里和二叉树层序遍历的思路类似，只是需要在加入节点的时候把Node里面的children都放到队列，可以用for直接遍历

1. 515：在每个树中找最大值



class Solution {

public:

    vector<int> largestValues(TreeNode\* root) {

        vector<int> res;

        if(root == nullptr) return res;

        queue<TreeNode\*> m\_que;

        m\_que.push(root);

        while(m\_que.size()!=0){

            int size = m\_que.size();

            int max = INT\_MIN;

            for(int i = 0; i < size; i++){

                TreeNode\* tmp = m\_que.front();

                m\_que.pop();

                max = max > tmp->val ? max : tmp->val;

                if(tmp->left != nullptr)    m\_que.push(tmp->left);

                if(tmp->right != nullptr)   m\_que.push(tmp->right);

            }

            res.push\_back(max);

        }

        return res;

    }

};

1. 116：填充每个节点的下一个右侧节点指针



class Solution {

public:

    Node\* connect(Node\* root) {

        if(root == nullptr)  return root;

        Node\* pre = nullptr;

        Node\* cur = nullptr;

        queue<Node\*> m\_que;

        m\_que.push(root);

        while(m\_que.size() != 0){

            int size = m\_que.size();

            for(int i = 0; i < size; i++){

                if(i == 0){

                    pre = m\_que.front();

                    m\_que.pop();

                }else{

                    cur = m\_que.front();

                    m\_que.pop();

                    pre->next = cur;

                    pre = cur;

                }

                if(pre->left != nullptr) m\_que.push(pre->left);

                if(pre->right != nullptr) m\_que.push(pre->right);

            }

            pre->next = nullptr;

        }

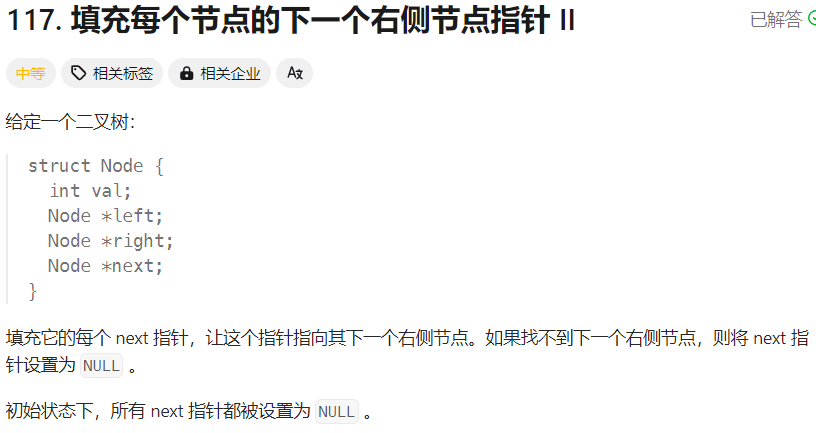
        return root;

    }

};

注意：这里要用到层序遍历和快慢指针的思想，每一层都用快慢指针来设置next的指向

1. 117：填充每个节点的下一个右侧节点指针2



class Solution {

public:

    Node\* connect(Node\* root) {

        if(root == nullptr) return root;

        Node\* pre = nullptr;

        Node\* cur = nullptr;

        queue<Node\*> m\_que;

        m\_que.push(root);

        while(!m\_que.empty()){

            int size = m\_que.size();

            for(int i = 0; i < size; i++){

                if(i == 0){

                    pre = m\_que.front();

                    m\_que.pop();

                }else{

                    cur = m\_que.front();

                    m\_que.pop();

                    pre->next = cur;

                    pre = cur;

                }

                if(pre->left != nullptr) m\_que.push(pre->left);

                if(pre->right != nullptr) m\_que.push(pre->right);

            }

            pre->next = nullptr;

        }

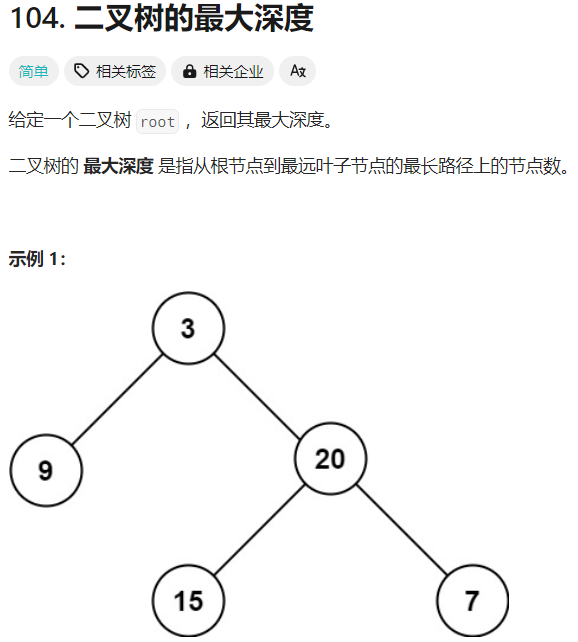
        return root;

    }

};

注意这题和上面那题思路一样，都是层序遍历加快慢指针，上面那题是完美二叉树，这题是普通的二叉树，不过都是一样的代码。

1. 104：二叉树的最大深度



class Solution {

public:

    int maxDepth(TreeNode\* root) {

        if(root == nullptr) return 0;

        int count = 0;

        queue<TreeNode\*> m\_que;

        m\_que.push(root);

        while(m\_que.size() != 0){

            int size = m\_que.size();

            count++;

            for(int i = 0; i < size; i++){

                TreeNode\* tmp = m\_que.front();

                m\_que.pop();

                if(tmp->left != nullptr) m\_que.push(tmp->left);

                if(tmp->right != nullptr) m\_que.push(tmp->right);

            }

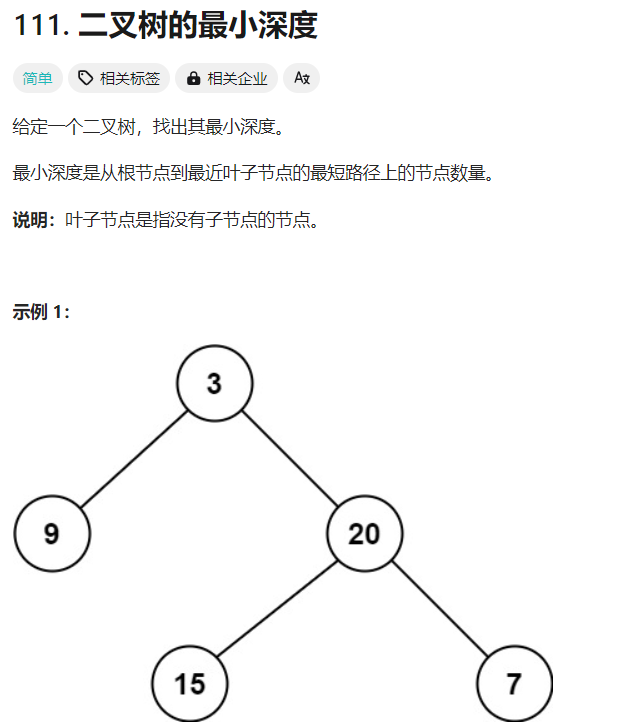
        }

        return count;

    }

};

1. 111：二叉树的最小深度



class Solution {

public:

    int minDepth(TreeNode\* root) {

        if(root == nullptr)return 0;

        int count =0;

        queue<TreeNode\*> m\_que;

        m\_que.push(root);

        while(m\_que.size()!=0){

            count++;

            int size = m\_que.size();

            for(int i = 0; i <size;i++){

                TreeNode\* cur = m\_que.front();

                m\_que.pop();

                if(cur->left==nullptr && cur->right == nullptr) return count;

                if(cur->left!=nullptr)m\_que.push(cur->left);

                if(cur->right!=nullptr)m\_que.push(cur->right);

            }

        }

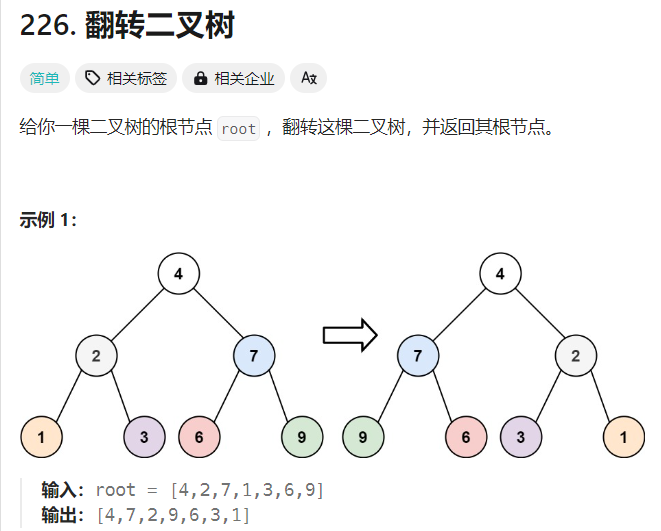
        return count;

    }

};

注意：有一个节点的左右节点都是空那就是没子树了

1. 226：反转二叉树



class Solution {

public:

    void reverseTree(TreeNode\* cur){

        if(cur == nullptr)return;

        swap(cur->left, cur->right);

        reverseTree(cur->left);

        reverseTree(cur->right);

    }

    TreeNode\* invertTree(TreeNode\* root) {

        if(root == nullptr)return root;

        //reverseTree(root);//递归法

        queue<TreeNode\*> m\_que;

        m\_que.push(root);

        while(m\_que.size() != 0){

            int size= m\_que.size();

            for(int i = 0; i < size; i++){

                TreeNode\* tmp = m\_que.front();

                m\_que.pop();

                swap(tmp->left, tmp->right);

                if(tmp->left != nullptr) m\_que.push(tmp->left);

                if(tmp->right != nullptr) m\_que.push(tmp->right);

            }

        }

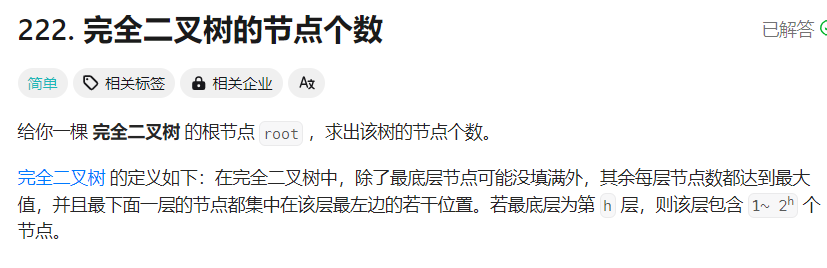
        return root;

    }

};

注意：这里有两种方法：递归和层序，主要思想就是交换每个节点的左右指针的值

1. 完全二叉树的节点个数



class Solution {

public:

    int countNodes(TreeNode\* root) {

        int count = 0;

        if(root == nullptr) return count;

        queue<TreeNode\*> m\_que;

        m\_que.push(root);

        while(m\_que.size()!=0){

            int size = m\_que.size();

            for(int i = 0; i < size; i++){

                TreeNode\* tmp = m\_que.front();

                m\_que.pop();

                count++;

                if(tmp->left != nullptr) m\_que.push(tmp->left);

                if(tmp->right != nullptr) m\_que.push(tmp->right);

            }

        }

        return count;

    }

};

思路：层序遍历，不管你是二叉树还是完全二叉树，没遍历一个节点就count++

1. 110平衡二叉树



class Solution {

public:

    int getCount(TreeNode\* node){

        int count = 0;

        if(node == nullptr) return 0;

        queue<TreeNode\*> m\_Lque;

        m\_Lque.push(node);

        while(m\_Lque.size()!=0){

            int size = m\_Lque.size();

            count++;

            for(int i = 0; i < size; i++){

                TreeNode\* cur = m\_Lque.front();

                m\_Lque.pop();

                if(cur->left != nullptr) m\_Lque.push(cur->left);

                if(cur->right != nullptr) m\_Lque.push(cur->right);

            }

        }

        return count;

    }

    bool isBalanced(TreeNode\* root) {

        int lCount = 0;

        int rCount = 0;

        if(root == nullptr) return true;

        //计算左子树深度

        lCount = getCount(root->left);

        //计算右子树深度

        rCount = getCount(root->right);

        return abs(lCount - rCount) <= 1 && isBalanced(root->left) && isBalanced(root->right);

    }

};

注意看返回条件，这可不是只判断abs(lCount - rCount) <= 1。因为平衡二叉树的定义是每个节点的左右子树的高度差不大于1，所以如果只是abs(lCount - rCount) <= 1的话就只看了根节点，这是错的，只能通过百分之九十的测试用例。正确的是return abs(lCount - rCount) <= 1 && isBalanced(root->left) && isBalanced(root->right);