剑指offer

1 栈与队列

（1）用两个栈实现队列

> 用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 `appendTail` 和 `deleteHead` ，分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素， `deleteHead` 操作返回 -1)
>
> 输入：
>
> ```
> ["CQueue","appendTail","deleteHead","deleteHead","deleteHead"]
> [[],[3],[],[],[]]
> ```
>
> 输出：
>
> ```
> [null,null,3,-1,-1]
> ```
>
> - `1 <= values <= 10000`
> - 最多会对 `appendTail、deleteHead` 进行 `10000` 次调用

思路：用两个栈可以模拟队列的实现，栈s1用来存储队列中的元素，辅助栈s2当需要弹出队头元素时，将s1中的内容全部pop到s2，最后s2的栈顶元素就是队头元素，再讲s2中的元素pop回s1，`deleteHead` 的时间复杂度O(n)

```
class CQueue {
private:
    stack<int> s1, s2;
public:
    CQueue() {

    }

    void appendTail(int value) {
        s1.push(value);
    }

    int deleteHead() {
        if (s1.empty())
            return -1;
        while (s1.size()){
            s2.push(s1.top());
            s1.pop();
        }
        int temp = s2.top();
        s2.pop();
        while (s2.size()){
            s1.push(s2.top());
            s2.pop();
        }
        return temp;
```

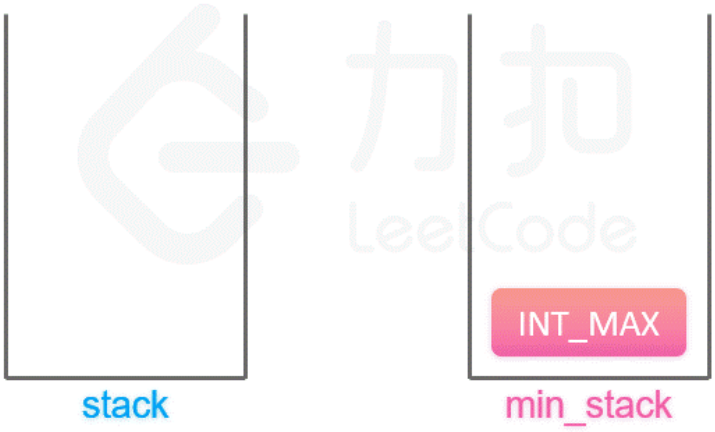
```
    }
};
```

(2) 包含min函数的栈

> 定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。
>
> **示例:**
>
> ```
> MinStack minStack = new MinStack();
> minStack.push(-2);
> minStack.push(0);
> minStack.push(-3);
> minStack.min();   --> 返回 -3.
> minStack.pop();
> minStack.top();      --> 返回 0.
> minStack.min();   --> 返回 -2.
> ```
>
> **提示:**
>
> 1. 各函数的调用总次数不超过 20000 次

1）、使用两个辅助栈

栈`st`存储栈中的元素，`min_st`存储每次入栈之后元素的最小值，当栈`st`中的元素pop出栈时，同时将`min_st`中的元素pop



在min_st初始化时，将INT_MAX入栈，就能保证min_st中至少存在一个元素，不用判断栈是否为空

```
class MinStack {
    stack<int> st, min_st;
public:
    /** initialize your data structure here. */
    MinStack() {
        min_st.push(INT_MAX);
```

```
    }

    void push(int x) {
        st.push(x);
        min_st.push( x <= min_st.top() ? x :  min_st.top());
    }

    void pop() {
        st.pop();
        min_st.pop();
    }

    int top() {
        return st.top();
    }

    int min() {
        return min_st.top();
    }
};
```

2）只实用一个栈

只使用一个栈，对于栈中的元素而言，只有当前待入栈的元素`val < min`时，才需要更新栈中最小的元素。但如何在最小的元素出栈之后，以O(1)的时间来找到出栈后的最小元素？

solution using one stack

当`val <= min`时，min元素会被更新，我们应该还保存之前的min，这样当最小的元素出栈后，可以以O(1)的时间来找到最小的元素。

详细通俗的思路分析，多解法

```
入栈 3，存入 3 - 3 = 0
|   |   min = 3
|   |
|_0_|
stack

入栈 5，存入 5 - 3 = 2
|   |   min = 3
| 2 |
|_0_|
stack

入栈 2，因为出现了更小的数，所以我们会存入一个负数，这里很关键
也就是存入  2 - 3 = -1，并且更新 min = 2
对于之前的 min 值 3，我们只需要用更新后的 min - 栈顶元素 -1 就可以得到
| -1|   min = 2
| 5 |
|_3_|
stack

入栈 6，存入  6 - 2 = 4
| 4 |   min = 2
| -1|
```

```
| 5 |
|_3_|
stack

出栈，返回的值就是栈顶元素 4 加上 min，就是 6
|   |   min = 2
| -1|
| 5 |
|_3_|
stack

出栈，此时栈顶元素是负数，说明之前对 min 值进行了更新。
入栈元素 - min = 栈顶元素，入栈元素其实就是当前的 min 值 2
所以更新前的 min 就等于入栈元素 2 - 栈顶元素(-1) = 3
|   | min = 3
| 5 |
|_3_|
stack
```

代码

```
class MinStack {
    stack<int> st;
    int MIN = INT_MAX;
public:
    /** initialize your data structure here. */
    MinStack() {
    }

    void push(int x) {
        if (x <= MIN){
            st.push(MIN);
            MIN = x;
        }
        st.push(x);
    }

    void pop() {
        if (MIN == st.top()){
            st.pop();
            MIN = st.top();
        }
        st.pop();
    }

    int top() {
        return st.top();
    }

    int min() {
        return MIN;
    }
};
```

2 链表（简单）

（1）从尾到头打印链表

> 输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。
>
> **示例 1：**
>
> ```
> 输入：head = [1,3,2]
> 输出：[2,3,1]
> ```
>
> **限制：**
>
> ```
> 0 <= 链表长度 <= 10000
> ```

1）、利用reverse函数，将结果进行反转

```
class Solution {
public:
    vector<int> reversePrint(ListNode* head) {
        vector<int> res;
        while (head){
            res.push_back(head->val);
            head = head->next;
        }
        reverse(res.begin(), res.end());
        return res;
    }
};
```

2）利用递归的特性

```
class Solution {
public:
    //使用递归
    vector<int> res;
    vector<int> reversePrint(ListNode* head) {
        dfs(head);
        return res;
    }

    void dfs(ListNode *head){
        if (!head)  return ;
        dfs(head->next);
        res.push_back(head->val);
    }
};
```

(2) 反转链表

> 定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。
>
> **示例:**
>
> ```
> 输入: 1->2->3->4->5->NULL
> 输出: 5->4->3->2->1->NULL
> ```
>
> **限制:**
>
> ```
> 0 <= 节点个数 <= 5000
> ```

1）尾插法建立单链表

```
class Solution {
public:
    // 1、使用头插法反转单链表
    // 2、递归
    ListNode* reverseList(ListNode* head) {
        ListNode* dummy = new ListNode(-1), *p;
        while (head){
            p = head->next;
            head->next = dummy->next, dummy->next = head;
            head = p;
        }
        return dummy->next;
    }
};
```

时间复杂度O(n)，空间复杂度O(1)

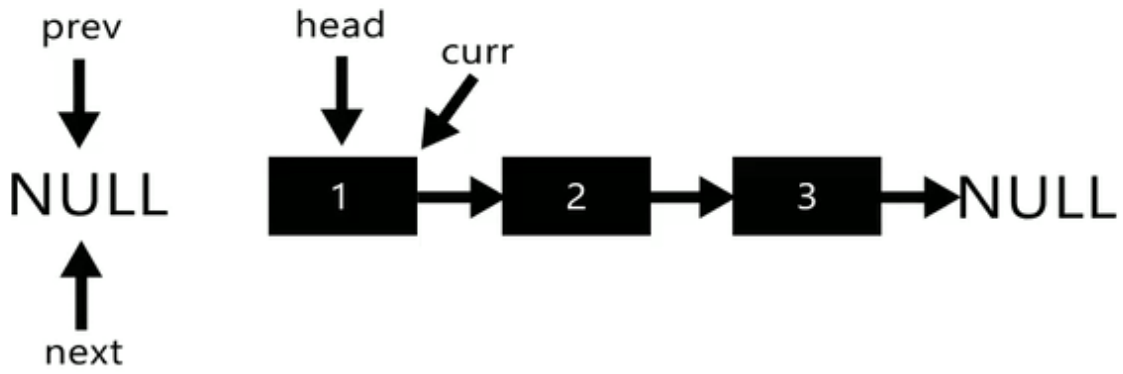
2）递归

注意递归边界以及递归的方式

```
class Solution {
public:
    // 1、使用头插法反转单链表
    // 2、递归
    ListNode* reverseList(ListNode* head) {
        return dfs(NULL, head);
    }

    ListNode* dfs(ListNode* pre, ListNode* cur){
        if (!cur)
            return pre;
        ListNode* res = dfs(cur, cur->next);    //cur反转之后的 cur <- res
        cur->next = pre;
        return res;
    }
};
```

3）反转指针




```
while (current != NULL)
      {
          next  = current->next;
          current->next = prev;
          prev = current;
          current = next;
      }
      *head_ref = prev;
```

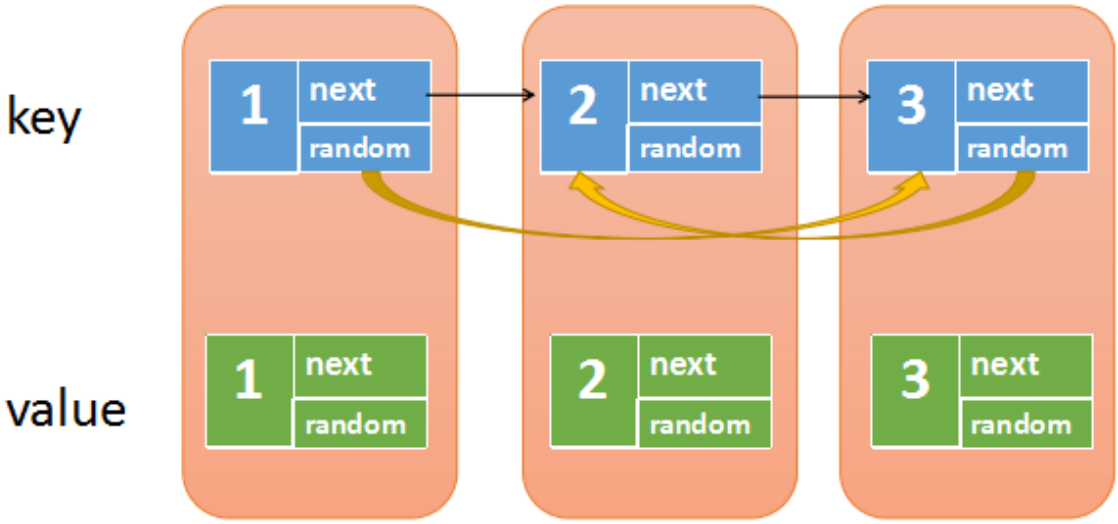
```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* pre = NULL, *next;
        while (head){
            next = head->next;  //保存head的后继
            head->next = pre;
            pre = head;
            head = next;
        }
        return pre;
    }
};
```

(3) 复杂链表的复制

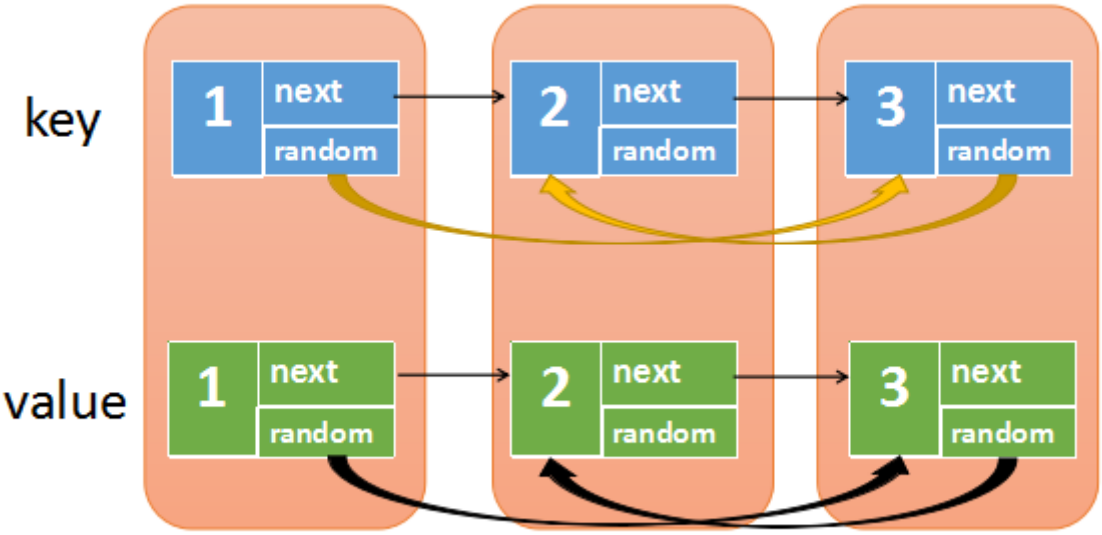
1）使用哈希表

时间复杂度O(N)，空间复杂度O(N)

首先创建一个哈希表，再遍历原链表，遍历的同时再不断创建新节点
我们将原节点作为**key**，新节点作为**value**放入哈希表中



第二步我们再遍历原链表，这次我们要将新链表的next和random指针给设置上



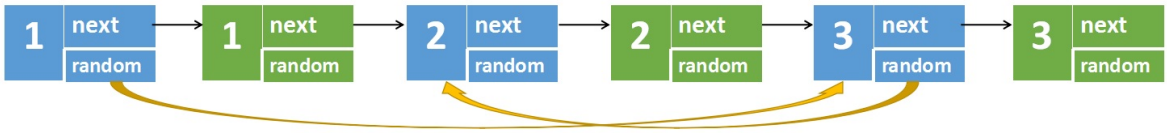

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        unordered_map<Node*, Node*> mp;
        Node* cur = head;
        while (cur){
            mp[cur] = new Node(cur->val);
            cur = cur->next;
        }
        cur = head;
        while (cur){
```

```
            mp[cur]->next = mp[cur->next];
            mp[cur]->random = mp[cur->random];
            cur = cur->next;
        }
        return mp[head];
    }
};
```

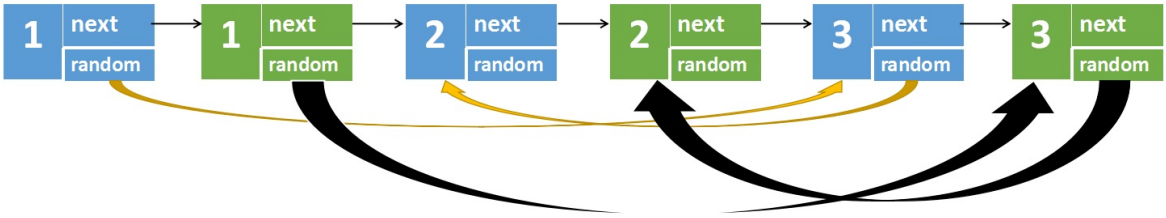
2)

[两种实现+图解 138. 复制带随机指针的链表]

第一步，根据遍历到的原节点创建对应的新节点，每个新创建的节点是在原节点后面，比如下图中原节点**1**不再指向原原节点**2**，而是指向新节点**1**



第二步是最关键的一步，用来设置新链表的随机指针

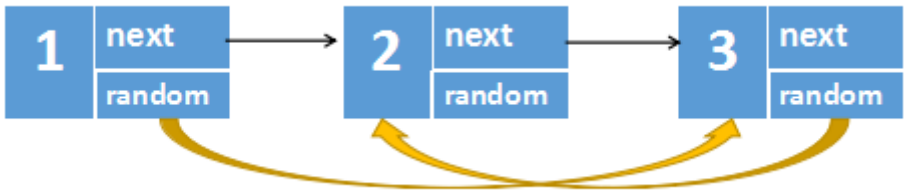


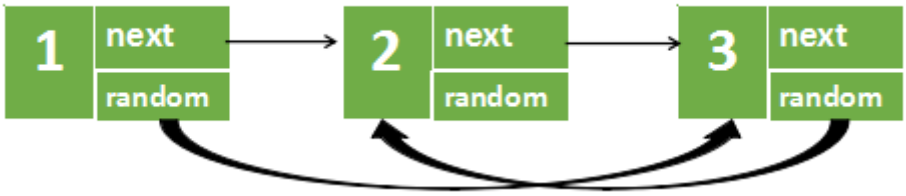
上图中，我们可以观察到这么一个规律

- 原节点1的随机指针指向原节点3，新节点1的随机指针指向的是原节点3的next
- 原节点3的随机指针指向原节点2，新节点3的随机指针指向的是原节点2的next

也就是，原节点`i`的随机指针(如果有的话)，指向的是原节点`j`
那么新节点`i`的随机指针，指向的是原节点`j`的next

第三步就简单了，只要将两个链表分离开，再返回新链表就可以





时间复杂度O(N)，空间复杂度O(1)

```
class Solution {
public:
```

```
    Node* copyRandomList(Node* head) {
        Node* dummy = new Node(-1), *p = head, *r;
        //1、创建一个新的链表
        while (p){
            Node* newNode = new Node(p->val);
            newNode->next = p->next;
            p->next = newNode;
            p = newNode->next;
        }

        //2、复制random
        p = head;
        while (p){
            if (p->random)  //核心
                p->next->random = p->random->next;
            p = p->next->next;
        }

        p = head, r = dummy;
        while (p){
            r->next = p->next;
            r = p->next;
            p->next = r->next;
            p = p->next;
        }

        return dummy->next;
    }
};
```

3 字符串（简单）

（1）替换空格

> 请实现一个函数，把字符串 `s` 中的每个空格替换成"%20"。
>
> **示例 1：**
>
> ```
> 输入：s = "We are happy."
> 输出："We%20are%20happy."
> ```

1）边扫描边替换

```
class Solution {
public:
    string replaceSpace(string s) {
        string res = "";
        for (char ch : s){
            if (ch != ' ')
                res += ch;
            else    res += "%20";
        }
        return res;
    }
};
```

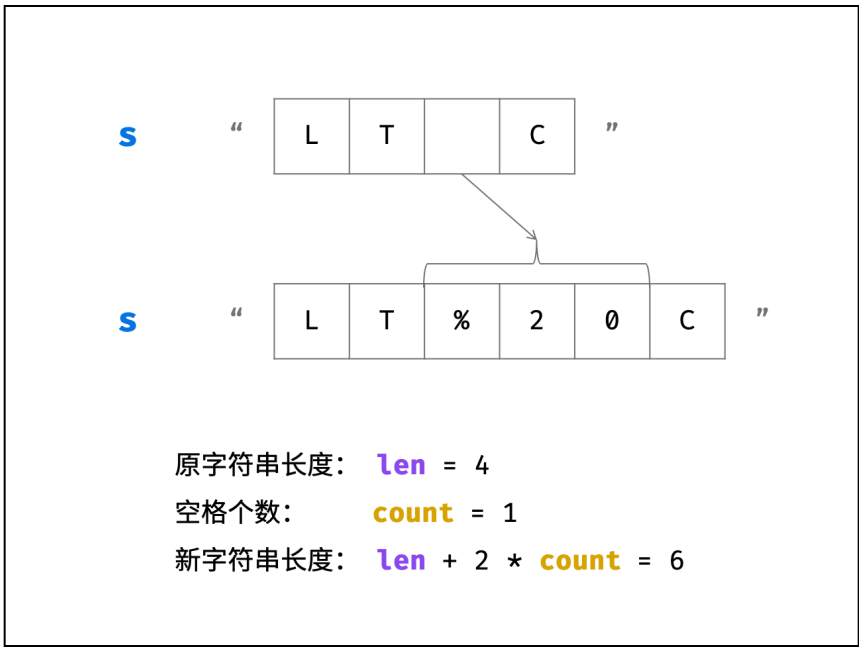
2）利用函数直接在s中进行替换

```
string& replace (size_t pos, size_t len, const string& str);
```

- `pos`：要替换的字符串的起始位置。
- `len`：要替换的字符串的长度。
- `str`：要插入的新字符串。

```
class Solution {
public:
    string replaceSpace(string s) {
        while (s.find(" ") != string::npos){
            s.replace(s.find(" "), 1, "%20");
        }
        return s;
    }
};
```

3）修改s的长度，进行替换



```
class Solution {
public:
    string replaceSpace(string s) {
        int cnt = 0, n = s.size();
        for (char ch : s){
            if (ch == ' ')  cnt++;
        }
        s.resize(n+2*cnt);  //字符串扩展
        for (int i = n-1, j = s.size()-1; i>= 0 && i != j; ){
            if (s[i] != ' ')
                s[j--] = s[i--];
            else{
                s[j] = '0', s[j-1] = '2', s[j-2] = '%';
                j -= 3, i--;
            }
        }
        return s;
    }
};
```

(2) 左旋转字符串

> 字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。
>
> **示例 1:**
>
> ```
> 输入: s = "abcdefg", k = 2
> 输出: "cdefgab"
> ```
>
> **示例 2:**
>
> ```
> 输入: s = "lrloseumgh", k = 6
> 输出: "umghlrlose"
> ```

1）字符串拼接

时间复杂度O(N)，空间复杂度O(1)

```
class Solution {
public:
    string reverseLeftWords(string s, int n) {
        string res = "";
        for (int i = n; i < s.size(); i++)
            res += s[i];
        for (int i = 0; i < n; i++)
            res += s[i];
        return res;
    }
};
```

2）两次翻转

`abcdefg, k = 2`

将前k个字符和剩下的字符翻转，`bagfedc`

再将整个字符串翻转`cdefgab`

```
class Solution {
public:
    string reverseLeftWords(string s, int n) {
        reverse(s.begin(), s.begin() + n);
        reverse(s.begin() + n, s.end());
        reverse(s.begin(), s.end());
        return s;
    }
};
```

4 查找算法（简单）

(1) 数组中重复的数字

> 找出数组中重复的数字。
> 在一个长度为 n 的数组 nums 里的所有数字都在 0～n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。
>
> **示例 1：**
>
> ```
> 输入：
> [2, 3, 1, 0, 2, 5, 3]
> 输出：2 或 3
> ```
>
> **限制：**
>
> ```
> 2 <= n <= 100000
> ```

1）使用哈希表

利用`hashTable[MAXN]`来统计每个数字出现的次数，当`nums[i]`的出现次数>1，直接将`nums[i]`返回

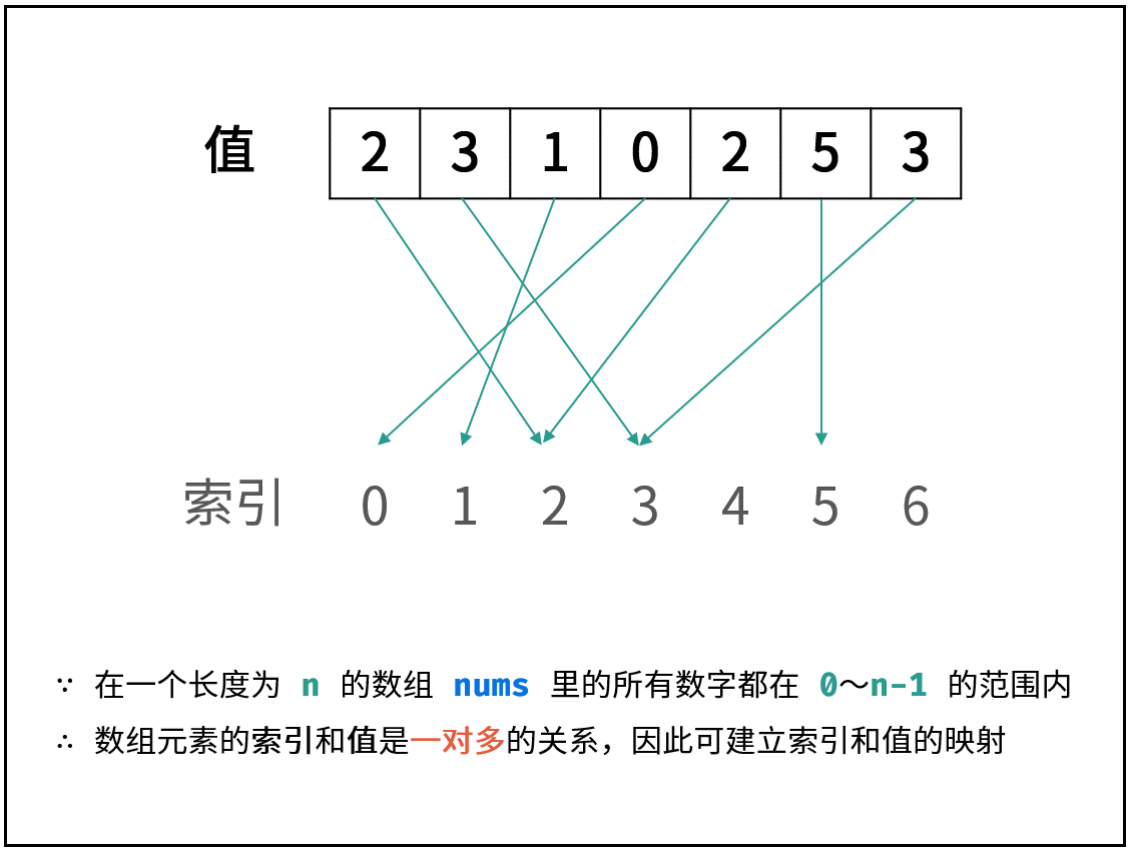
时间复杂度$O(N)$，空间复杂度$O(N)$

```
class Solution {
public:
    int findRepeatNumber(vector<int>& nums) {
        int hashTable[100010] = {0};
        for (int i = 0; i < nums.size(); i++){
            if (hashTable[nums[i]])
                return nums[i];
            hashTable[nums[i]]++;
        }
        return -1;
    }
};
```

2）充分挖掘题目信息

题解

可以注意到`nums`中的数字都在`[0，n-1]`之间，刚好和下标相对应



遍历中，第一次遇到数字 x 时，将其交换至索引 x 处；而当第二次遇到数字 `x` 时，一定有 `nums[x]=x` ，此时即可得到一组重复数字。

算法流程：

1. 遍历数组`nums` ，设索引初始值为 `i=0`：
    1. **若** `nums[i]=i` ：说明此数字已在对应索引位置，无需交换，因此跳过；
    2. **若** `nums[nums[i]]=nums[i]` ： 代表索引`nums[i]`处和索引 `i`处的元素值都为 `nums[i]`，即找到一组重复值，返回此值 `nums[i]`；
    3. **否则：** 交换索引为 `i` 和 `nums[i]`的元素值，将此数字交换至对应索引位置。

       `swap(nums[i], nums[nums[i]])`能够将索引为`i`上的`nums[i]`元素交换到索引为`nums[i]`的位置上
2. 若遍历完毕尚未返回，则返回 −1 。

复杂度分析：

- **时间复杂度 O(N)：** 遍历数组使用 O(N)O(N)*O*(*N*) ，每轮遍历的判断和交换操作使用 O(1) 。
- **空间复杂度 O(1)：** 使用常数复杂度的额外空间。

```
数组                  [2, 3, 1, 0, 2, 5, 3]
i = 0时，nums[0] != 0，判断nums[nums[i]] == nums[i]，现在不等于，将元素进行交换
                    [1, 3, 2, 0, 2, 5, 3]
对于i=0而言，当前位置上的元素仍不满足nums[i] = i，继续判断是否nums[i]位置上的元素等于i位置上的元素
                    [3, 1, 2, 0, 2, 5, 3]
下一步               [0, 1, 2, 3, 2, 5, 3]
每次swap(nums[i], nums[nums[i]])能够将索引为i上的nums[i]元素交换到索引为nums[i]的位置上
i++,
当i=4时，nums[nums[4]] == nums[4]时，返回nums[i]
```

简而言之，因为数组中元素都在`[0, n-1]`

那么我们通过交换，使得`nums[i] = i`，但数组中存在重复的元素必定使得有多个元素对应同一个下标

对于索引为`i`上的`nums[i]`元素

如果`i == nums[i]`，那么`i++`，元素已经在对应的位置上

否则，就将索引为`i`和`nums[i]`位置上的元素作比较（因为不满足`i == nums[i]`，那么对于索引为`i`位置上的元素`nums[i]`的位置应该在`nums[nums[i]]`），如果两者相等说明已经找到了重复元素

再将索引为`i`和`nums[i]`位置上的元素进行交换（每次交换之后都可以使索引为`nums[i]`的元素放在正确的位置`nums[nums[i]]`上）

explanation:

> 这个原地交换法就相当于分配工作，每个索引代表一个工作岗位，每个岗位必须专业对口，既0索引必须0元素才能上岗。而我们的目的就是找出溢出的人才，既0索引岗位有多个0元素竞争。
>
> 我们先从0索引岗位开始遍历，首先我们看0索引是不是已经专业对口了，如果已经专业对口既`nums[0]=0`，那我们就跳过0岗位看1岗位。如果0索引没有专业对口，那么我们看现在0索引上的人才调整到他对应的岗位上，比如num[0]=2，那我们就把2这个元素挪到他对应的岗位上既num[2]，这个时候有两种情况:
>
> 1、num[2]岗位上已经有专业对口的人才了，既num[2]=2，这就说明刚刚那个在num[0]上的2是溢出的人才，我们直接将其返回即可。
>
> 2、num[2]上的不是专业对口的人才，那我们将num[0]上的元素和num[2]上的元素交换，这样num[2]就找到专业对口的人才了。
>
> 之后重复这个过程直到帮num[0]找到专业对口的人才，然后以此类推帮num[1]找人才、帮num[2]找人才，直到找到溢出的人才。

代码

```cpp
class Solution {
public:
    int findRepeatNumber(vector<int>& nums) {
        for (int i = 0; i < nums.size(); ){
            if (nums[i] == i){
                i++;
                continue;
            }
            if (nums[nums[i]] == nums[i])
                return nums[i];
            swap(nums[i], nums[nums[i]]);
```

```
        }
        return -1;
    }
};
```

(2) 在排序数组中查找数字 I

1）二分+库函数

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        return upper_bound(nums.begin(), nums.end(), target) -
lower_bound(nums.begin(), nums.end(), target);
    }
};
```

2）手写二分

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        return lower_bound(nums, target+1) - lower_bound(nums, target);
    }

    int lower_bound(const vector<int> &nums, int target){
        //寻找数组中>=target的第一个位置
        int low = 0, high = nums.size();
        while (low < high){
            int mid = (low + high) / 2;
            if (nums[mid] >= target)    high = mid;
            else    low = mid+1;
        }
        return low;
    }
};
```

(3) 0～n-1中缺失的数字 ❤

1）哈希表

2）利用数组中的元素递增并且缺失的是最小未出现的元素

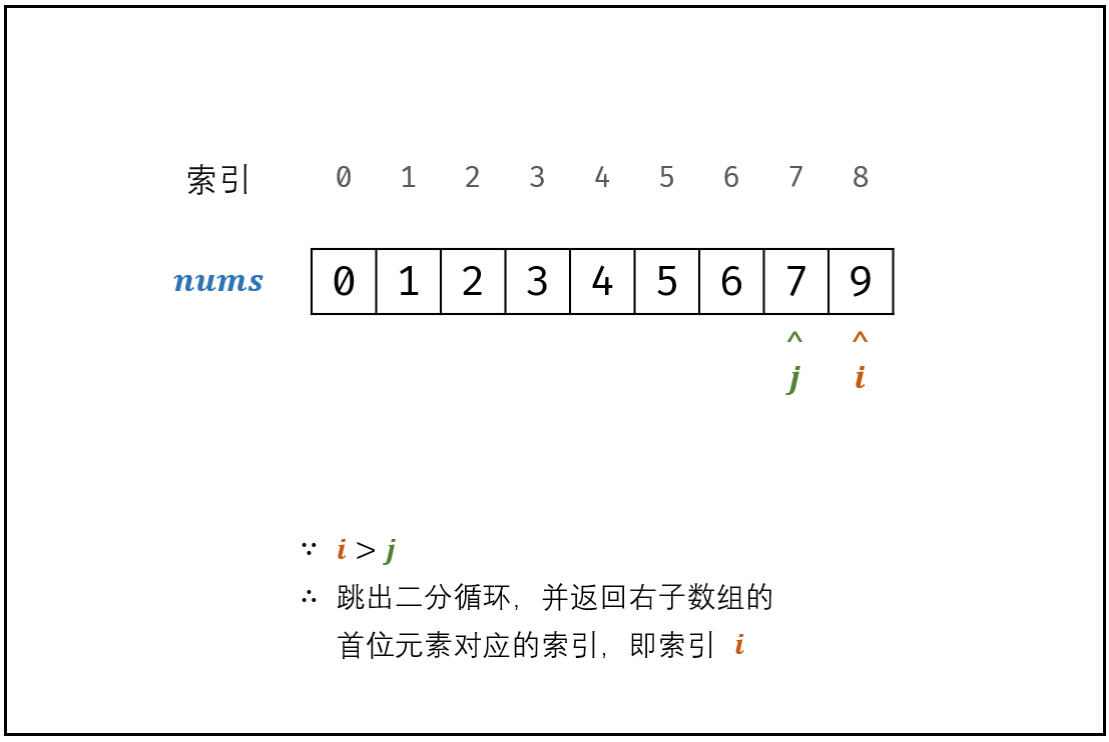
```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int i;
        for(i = 0; i < nums.size(); i++) {
            if(nums[i] != i) {
                return i;
            }
        }
        return i;
    }
};
```

3）二分查找

对于有序数组而言，使用二分查找可以到达$O(logN)$的时间

算法解析：

1. **初始化：** 左边界`i=0`，右边界`j=len(nums)-1`；代表闭区间`[i,j]`。
2. 循环二分： 当`i≤j`时循环，（即当闭区间`[i,j]`为空时跳出）；
    1. 计算中点`m=(i+j)/2`；
    2. 若`nums[m]=m`，则“右子数组的首位元素”一定在闭区间`[m+1,j]`中，因此执行`i=m+1`
    3. 若`nums[m]≠m`，则“左子数组的末位元素”一定在闭区间`[i,m-1]`中（缺失了元素，必定达不到m），因此执行 j=m−1
3. **返回值：** 跳出时，变量`i`和`j`分别指向“右子数组的首位元素”和“左子数组的末位元素”。因此返回`i`即可。




```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        //二分查找：未缺失元素的性质nums[i] == i
        int left = 0, right = nums.size()-1;
        while (left <= right){
            int mid = (left + right) >> 1;
```

```
            if (nums[mid] == mid)
                left = mid+1;
            else
                right = mid-1;
        }
        return left;
    }
};
```

5 查找算法（中等）

（1）二维数组中的查找 ❤

> 在一个 n * m 的二维数组中，每一行都按照从左到右 **非递减** 的顺序排序，每一列都按照从上到下 **非递减** 的顺序排序。请完成一个高效的函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。
>
> **示例:**
>
> 现有矩阵 matrix 如下：
>
> ```
> [ [1,   4,  7, 11, 15],
> [2,   5,  8, 12, 19],
> [3,   6,  9, 16, 22],
> [10, 13, 14, 17, 24],
> [18, 21, 23, 26, 30]]
> ```
>
> 给定 target = `5`，返回 `true`。
>
> 给定 target = `20`，返回 `false`。
>
> **限制:**
>
> ```
> 0 <= n <= 1000
> 0 <= m <= 1000
> ```

1）遍历矩阵

时间复杂度$O(n*m)$，空间复杂度$O(1)$

2）二分

矩阵每一行的元素都是递增有序的

时间复杂度$O(n*log_m)$，空间复杂度$O(1)$

```
class Solution {
public:
    bool findNumberIn2DArray(vector<vector<int>>& matrix, int target) {
        if (matrix.size() == 0) return false;
        int m = matrix.size(), n = matrix[0].size();
        for (vector<int> row : matrix){
            int j = lower_bound(row.begin(), row.end(), target) - row.begin();
            if (j < n && row[j] == target)
                return true;
        }
        return false;
    }
};
```

3）将矩阵看成二叉树

从左下角(m-1, 0)或者右上角(0, n-1)出发，将矩阵看成一棵二叉树

充分利用矩阵每一行有序、每一列有序

时间复杂度$O(n+m)$，空间复杂度$O(1)$

```
class Solution {
public:
    bool findNumberIn2DArray(vector<vector<int>>& matrix, int target) {
        if (matrix.size() == 0) return false;
        int m = matrix.size(), n = matrix[0].size();
        int row = m-1, col = 0;
        while (row >= 0 && col < n){
            if (matrix[row][col] == target)
                return true;
            else if (matrix[row][col] > target)
                row--;
            else
                col++;
        }
        return false;
    }
};
```

(2) 旋转数组的最小数字 ♡

> 把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。
>
> 给你一个可能存在 **重复** 元素值的数组 `numbers` ，它原来是一个升序排列的数组，并按上述情形进行了一次旋转。请返回旋转数组的**最小元素**。例如，数组 `[3,4,5,1,2]` 为 `[1,2,3,4,5]` 的一次旋转，该数组的最小值为 1。
>
> 注意，数组 `[a[0], a[1], a[2], ..., a[n-1]]` 旋转一次 的结果为数组 `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]` 。
>
> **示例 1：**

```
输入：numbers = [3,4,5,1,2]
输出：1
```

示例 2：

```
输入：numbers = [2,2,2,0,1]
输出：0
```

提示：

- `n == numbers.length`
- `1 <= n <= 5000`
- `-5000 <= numbers[i] <= 5000`
- `numbers` 原来是一个升序排序的数组，并进行了 `1` 至 `n` 次旋转

1）遍历

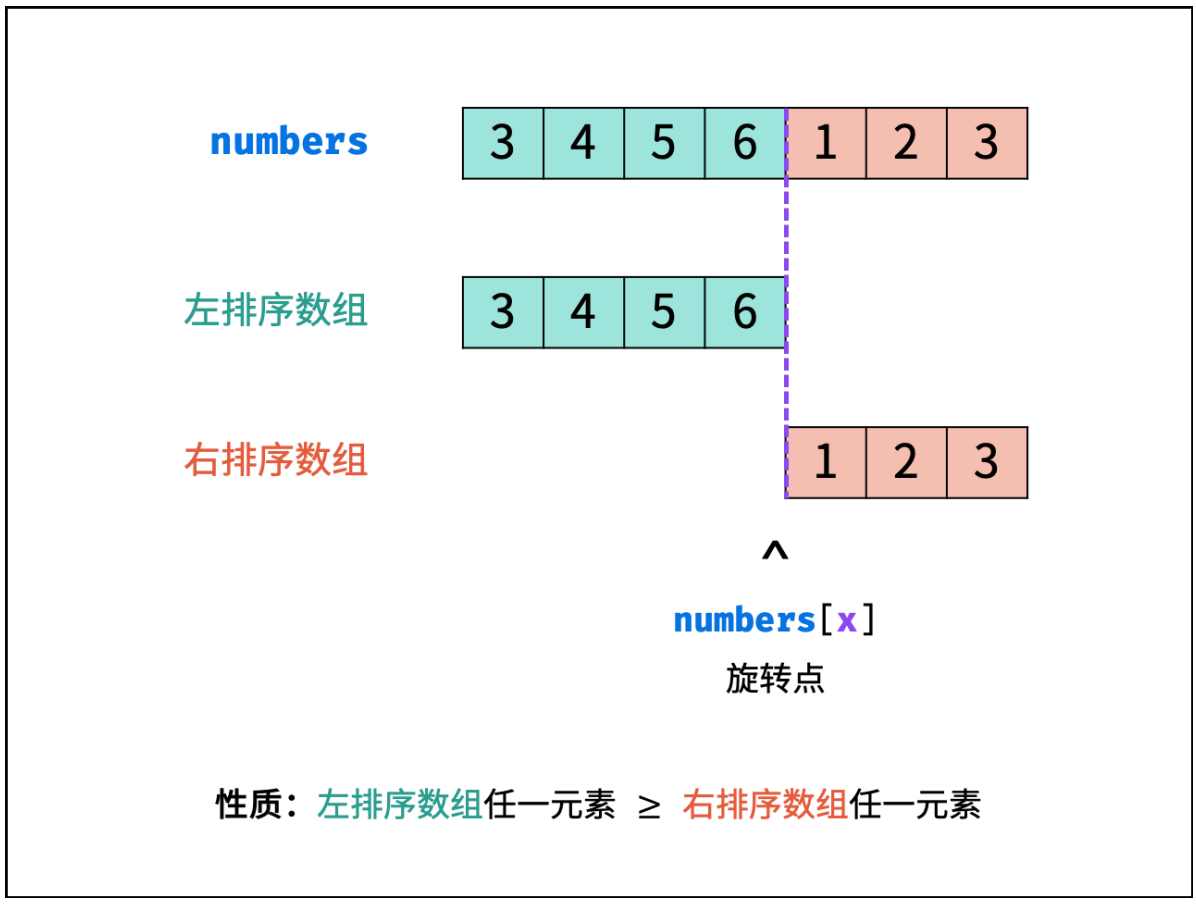
时间复杂度$O(n)$，空间复杂度$O(1)$

```
class Solution {
public:
    int minArray(vector<int>& numbers) {
        return *min_element(numbers.begin(), numbers.end());
    }
};
```

2）二分

题解

如下图所示，寻找旋转数组的最小元素即为寻找 右排序数组 的首个元素 `nums[x]` ，称 x 为 旋转点 。



排序数组的查找问题首先考虑使用 二分法 解决，其可将 遍历法 的 线性级别 时间复杂度降低至 对数级别 。

算法流程：
初始化： 声明 `i, j` 双指针分别指向 `nums` 数组左右两端；
循环二分： 设 `m=(i+j)/2` 为每次二分的中点（ 因此恒有 `i ≤ m < j` ），可分为以下三种情况：
当 `nums[m]>nums[j]` 时： `m` 一定在 左排序数组 中，即旋转点 `x` 一定在 `[m + 1, j]` 闭区间内，因此执行 `i=m+1`；
当 `nums[m]<nums[j]` 时： `m` 一定在 右排序数组 中，即旋转点 `x` 一定在 `[i,m]` 闭区间内，因此执行 `j=m`；
当 `nums[m]=nums[j]` 时： 无法判断 `m` 在哪个排序数组中，即无法判断旋转点 `x` 在 `[i,m]` 还是 `[m+1,j]` 区间中。

解决方案： 执行 `j=j-1` 缩小判断范围，分析见下文。
返回值： 当 `i=j` 时跳出二分循环，并返回 旋转点的值 `nums[i]` 即可。

为什么是nums[m]和右端点nums[j]比较:

如果`nums[m]`和`左端点nums[i]`比较，那么可能无法区分最小值的位置，例如

```
1 2 3 4 5
2 3 4 5 1
```

对于上述两种情况，都有`nums[mid] > nums[i]`，但是上面一种情况的最小值在m之前，而下面一种情况最小值在m之后

如果`nums[m]`和`有端点nums[j]`比较

- 对`1 2 3 4 5`而言，有`nums[m] < nums[j]`，那么`[m...j]`之间必定是递增有序的，最小值必定在`[i...m]`之间，所以可以修改区间右端点`j = m`

- 对`2 3 4 5 1`而言，有`nums[m] > nums[j]`，原数组是一个递增有序的序列，但是现在中间的值>右端点的值，即**出现了断层**，旋转点必定在`[m+1...j]`之间，所以可以修改区间左端点`i = m + 1`

```
class Solution {
public:
    int minArray(vector<int>& numbers) {
        int n = numbers.size(), left = 0, right = n-1;
        if (numbers.size() == 1)
            return numbers[0];
        while (right > left){
            int mid = left + (right - left) / 2;
            if (numbers[mid] > numbers[right])
                left = mid + 1;
            else if (numbers[mid] < numbers[right])
                right = mid;
            else
                right--;
        }
        return numbers[left];
    }

};
```

(3) 第一个只出现一次的字符

在字符串 s 中找出第一个只出现一次的字符。如果没有，返回一个单空格。 s 只包含小写字母。

示例 1:

```
输入：s = "abaccdeff"
输出：'b'
```

示例 2:

```
输入：s = ""
输出：' '
```

限制：

```
0 <= s 的长度 <= 50000
```

1）哈希表

```
class Solution {
public:
    char firstUniqChar(string s) {
        int hastTable[26] = {0};
        for (char ch : s){
```

```
            hastTable[ch - 'a']++;
        }
        for (int i = 0; i < s.size(); i++){
            if (hastTable[s[i] - 'a'] == 1)
                return s[i];
        }
        return ' ';
    }
};
```

2)

6 搜索与回溯算法（简单）

(1) I. 从上到下打印二叉树

> 从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。
>
> 例如:
> 给定二叉树: `[3,9,20,null,null,15,7]`,
>
> ```
>  3
> / \
> 9  20
>  /  \
> 15   7
> ```
>
> 返回:
>
> ```
> [3,9,20,15,7]
> ```
>
> **提示:**
>
> 1. `节点总数 <= 1000`

利用广度优先遍历来得到每一层的结点值（需要注意根节点是否为空）

```
class Solution {
public:
    vector<int> levelOrder(TreeNode* root) {
        if (!root)  return {};
        queue<TreeNode*> q;
        TreeNode* p =root;
        vector<int> res;
        q.push(p);
        while (q.size()){
            p = q.front();
            q.pop();
            res.push_back(p->val);
            if (p->left)    q.push(p->left);
            if (p->right)   q.push(p->right);
        }
        return res;
```

```
    }
};
```

(2) II. 从上到下打印二叉树

> 从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行。
>
> 例如:
> 给定二叉树: `[3,9,20,null,null,15,7]`,
>
> ```
>  3
> / \
> 9  20
>  /  \
> 15   7
> ```
>
> 返回其层次遍历结果：
>
> ```
> [
> [3],
> [9,20],
> [15,7]
> ]
> ```
>
> **提示：**
>
> 1. `节点总数 <= 1000`

BFS+辅助变量r来判断是否到达了当前层的最后一个结点

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        if (!root)  return {};

        vector<int> layer;
        vector<vector<int>> res;

        queue<TreeNode*> q;
        TreeNode* p = root, *r = root;
        q.push(p);

        while (q.size()){
            p = q.front();
            q.pop();
            layer.push_back(p->val);
            if (p->left)    q.push(p->left);
            if (p->right)   q.push(p->right);

            if (p == r){        //当前节点p是当前层的最后一个结点
                res.push_back(layer);
                layer.clear();  //存储当前层的layer清空
                r = q.back();   //更新下一层的最后一个结点
            }
```

```
        }

        return res;
    }
};
```

(3) III. 从上到下打印二叉树

> 请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。
>
> 例如:
> 给定二叉树: `[3,9,20,null,null,15,7]`,
>
> ```
>  3
> / \
> 9  20
>  /  \
> 15   7
> ```
>
> 返回其层次遍历结果：
>
> ```
> [
> [3],
> [20,9],
> [15,7]
> ]
> ```
>
> **提示：**
>
> 1. `节点总数 <= 1000`

和上一题相比，需要在偶数层将每一层的元素反转（假设根节点位于第一层），这里在增加一个变量high用来表示当前节点的高度

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        if (!root)  return {};

        int high = 1;   //增加的表示当前层高度的变量high
        vector<int> layer;
        vector<vector<int>> res;

        queue<TreeNode*> q;
        TreeNode* p = root, *r = root;
        q.push(p);

        while (q.size()){
            p = q.front();
            q.pop();
            layer.push_back(p->val);
            if (p->left)    q.push(p->left);
            if (p->right)   q.push(p->right);
```

```
            if (p == r){
                if (high%2 == 0)//偶数层，需要进行翻转
                    reverse(layer.begin(), layer.end());
                res.push_back(layer);
                layer.clear();
                r = q.back();
                high++;
            }
        }

        return res;
    }
};
```

7 搜索与回溯算法（简单）

(1) 树的子结构 💗

> 输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)
>
> B是A的子结构， 即 A中有出现和B相同的结构和节点值。
>
> 例如:
> 给定的树 A:
>
> `3  4  5  1  2`
> 给定的树 B：
>
> `4  1`
> 返回 true，因为 B 与 A 的一个子树拥有相同的结构和节点值。
>
> **示例 1：**
>
> ```
> 输入：A = [1,2,3], B = [3,1]
> 输出：false
> ```
>
> **示例 2：**
>
> ```
> 输入：A = [3,4,5,1,2], B = [4,1]
> 输出：true
> ```

递归

```
class Solution {
public:
    bool isSubStructure(TreeNode* A, TreeNode* B) {
        if (!A || !B)
            return false;
        return dfs(A, B) || isSubStructure(A->left, B) || isSubStructure(A->right, B);
        //B是以A为根的子结构      B是以A的左子树为根的子结构      B是以A的右子树为根的子结构
    }

    bool dfs(TreeNode* A, TreeNode* B){
        if (!B) return true;
```

```
        if (!A) return false;
        return A->val == B->val && dfs(A->left, B->left) && dfs(A->right, B->right);
    }
};
```

(2) 二叉树的镜像

> 请完成一个函数，输入一个二叉树，该函数输出它的镜像。
>
> 例如输入：
>
> `4  2  7  1  3 6  9`
>
> 镜像输出：
>
> ```
> 4  7  2  9  6 3  1
> ```
>
> **示例 1：**
>
> ```
> 输入：root = [4,2,7,1,3,6,9]
> 输出：[4,7,2,9,6,3,1]
> ```
>
> **限制：**
>
> ```
> 0 <= 节点个数 <= 1000
> ```

交换二叉树的左右孩子

```
class Solution {
public:
    TreeNode* mirrorTree(TreeNode* root) {
        if (!root)  return root;
        // TreeNode* p = root->left, *q = root->right;
        root->left = mirrorTree(root->left);
        root->right = mirrorTree(root->right);
        swap(root->left, root->right);
        return root;
    }
};
```

(3) 对称的二叉树