目录

Introduction

leetcode 刷题及总结

从一个数组中找到特定组合

array常见题型及总结

从数组中寻找寻找特定组合

从数组中找到和为指定target的序列

这类题型往往是给定一个数组，要求从数组中找到k（k=1,2,3,...）个元素和为target的序列，求解可以是找到指定序列的下标id。

从给定数组中找到key

常见的题型，主要包括：

- 给定一个数组，查找key是否在数组中
- 给定循环递增数组，从数组中找到最小值，或者找给定的key是否存在于这个循环数组中。

循环数组最小值

153. Find Minimum in Rotated Sorted Array

- special case：整个数组是有序的，这时候只需要返回第一个元素即可 （本题不考虑空数组的情况）
- 除了special case，那么最小值一定在数组中间，这时利用二分来判断，如果nums[left] < numsp[mid]则此部分一定是有序的，最小值在另一部分。
- 这里没有采用二分当中的left = mid + 1和right = mid -1，是为了将最小值限制在数组中间。

```
int findMin(vector<int>& nums) {
  size_t left = 0,right = nums.size()-1,mid;
  if(nums.size() == 1) return nums[left];
  if(nums[left]  < nums[right]) return nums[left];
  while(left + 1 < right){
      mid = (left + right)/2;
      if(nums[mid] > nums[left]){ //left is sorted
subarray
          left = mid;
      }else right = mid;
  }
  return nums[right];
}
```

154. Find Minimum in Rotated Sorted Array II

- 对于有重复元素的循环数组，special case如[0,0,0,-1,0],[1,1,2,1,1]等类似的用例，是没办法通过二分的方式查找的，只能顺序查找，除了原始用例这样的，还有可能通过二分划分之后subarray也是这种情况,因此这部分代码放在二分划分当中
- special case：nums[0] < nums[size - 1]，这时候只需要返回第一个元素即可

```
int findMin(vector<int>& nums) {
      size_t left = 0,right = nums.size()-1,mid;
      if(nums[left] < nums[right]) return nums[left];
      while(left + 1 < right){
          if(nums[left] == nums[right]) return
find_bysquence(nums,left,right);
          mid = (left + right)/2;
          if(nums[mid] >= nums[left]){ //left is
sorted subarray
              left = mid;
          }else right = mid;
      }
      return nums[right];
  }
```

- 平均时间复杂度O(log N)，最差时间复杂度为O(N)，空间复杂度O(1)

循环数组找特定值

33.Search in Rotated Sorted Array

容易想到的办法是直接遍历数组判断是否存在相同的数值，时间复杂度为O(N),很明显没有利用好循环递增数组的特定。

- 思路一
  - 找循环数组最小值，划分为两个有序递增数组
  - 在两个递增序列中进行二分查找
- 思路二
  - 直接利用二分查找的思路去判断，如arr[left] <= arr[mid]，说明此部分一定是递增的，然后判定target是否在区间内，不在则更新查找区间为后半部分。

```
int search(vector<int>& nums, int target) {
    if(nums.size() == 0) return -1;

    int left = 0,right = nums.size()-1,mid;
    while(left <= right){
        mid = (left + right) / 2;
        if(nums[mid] == target) return mid;
        if(nums[left] <= nums[mid]){
            if(nums[left] <= target && target <=
nums[mid]) right = mid -1;
            else left = mid + 1;
        }else{
            if(nums[mid] <= target && target <=
nums[right]) left = mid + 1;
            else right = mid-1;
        }
    }

    return -1;
}
```

- 空间复杂度O(logN)，时间复杂度为O(1)
  81. Search in Rotated Sorted Array II
- 由于存在重复元素，因此不能使用上题的思路二，但是可以采用思路一
- special case:[0,1,2,3,0,0,0,0]，通过找到第二个有序数组的最小值的下标，将他们切分成两个有序数组即[0,1,2,3],[0,0,0,0],再分别二分

从数组中找N-sum

这类题目主要分为

- 2-sum
- 3-sum
- 4-sum
- k-sum / combine-sum
- k-close

题目大意基本都是给定一个数组，和一个sum,从数组中找到k个元素和为sum的组合（一般需要组合不重复）

2-sum

1. Two Sum
数组可以分为两种情况，一种是经过排序的，一种是无序的,主要解法分为以下几种

- 暴力解法，挑选好一个元素nums[i]后，遍历数组是是否存在nums[j] == sum-nums[i], 空间复杂度O(1)，时间复杂度O(N^2)
- hash_map，遍历数组的同时查寻sum-nums[i]是否在map中，如果不再将nums[i]插入到map中，空间复杂度O(N)，时间复杂度

map:O(logN),unordered_map:O(N)

- 先排序，遍历，每遍历一个元素的同时，对数组进行二分查找。时间复杂度O(nlogn)，空间复杂度O(1)
- 先排序，然后双指针start和end，如果nums[start] + nums[end] < sum 则++start ,nums[start] + nums[end] > sum 则--end，若相等则返回。时间复杂度为O(nlogn)，空间复杂度为O(1)

3-sum

3.Sum

- 暴力，时间复杂度O(n^3)
- 排序，3个指针移动，start,mid,end,其中 0<=start < mid < end <= nums.size()-1，时间复杂度O(n^2)
- 排序，然后双指针i,j,然后通过二分查找查找target - nums[i] - nums[j]，但是有一个缺点是导致元素的重复使用。时间复杂度为O(n^2)

4-sum

18. 4Sum

- 暴力，时间复杂度O(n^4)
- 排序，四指针或者分解成3-sum->2-sum，时间复杂度O(n^3)

```
class Solution {
public:
    vector<vector<int>> twoSum(vector<int>& nums,int
start, int target) {
        int end = nums.size()-1;
        vector<vector<int>> res;
        while(start < end){
            int sum = nums[start] + nums[end];
            if(sum < target)++start;
            else if(sum > target) --end;
            else{
                res.push_back(vector<int>
{nums[start],nums[end]});
                while(--end > start && nums[end] ==
nums[end+1]);
                while(end > ++start && nums[start] ==
nums[start-1]);
            }
        }
        return res;
    }
    vector<vector<int>> threeSum(vector<int>& nums,int
start,int target) {
        vector<vector<int>> res;
        int end = nums.size()-1;
        while(start + 2 <= end){
            vector<vector<int>> two =
twoSum(nums,start+1,target-nums[start]);
            for(int j=0;j<two.size();++j){

two[j].insert(two[j].begin(),nums[start]);
                res.push_back(two[j]);
            }
            while(++start < end &&
nums[start]==nums[start-1]);
        }
        return res;
    }

    vector<vector<int>> fourSum(vector<int>& nums, int
target) {
        int start = 0,end = nums.size()-1;
        sort(nums.begin(),nums.end());
        vector<vector<int>>res;
        while(start + 3 <= end){
            vector<vector<int>> three =
threeSum(nums,start+1,target-nums[start]);
            for(int i=0;i<three.size();++i){
```

```
three[i].insert(three[i].begin(),nums[start]);
                res.push_back(three[i]);
            }
            while(++start < end && nums[start] ==
nums[start-1]);
        }
        return res;
    }
};
```

k-sum / combine-sum

39. Combination 4-sum
40. Combination Sum II

对于此类题目的通用解法，采用简单的0-1背包+回溯剪枝

- 对数组进行排序
- 当前数字nums[i]分为两种情况：加入背包和放弃加入背包，若加入背包，背包数字数字数量n+=1，sum = old_sum + nums[i];
- 判断sum == target，k== n,如果条件成立则当前背包中的数字集合即为结果之一

数组位运算符专题

首先说明一下几个位运算符

- 按位与（&）运算符
  - 1 & 1 = 1
  - 1 & 0 = 0, 0 & 1 = 0
  - 0 & 0 = 0
- 按位或(｜)运算符
  - 1 | 1 = 1
  - 1 | 0 = 1, 0 | 1 = 1
  - 0 | 0 = 0
- 按位异或(^)运算符
  - 1 ^ 1 = 0
  - 1 ^ 0 = 1, 0 ^ 1 = 1
  - 0 ^ 0 = 0

数组异或的题型，主要包括

- 给定一个数组，数组中的元素满足：仅有一个元素出现次数为奇数次，其余全为偶数次
- 给定一个数组，数组中的元素满足：仅有两个元素出现次数为奇数次，其余全为偶数次
- 给定一个数组，数组中的元素满足：仅有三个元素出现次数为奇数次，其余全为偶数次
- 给定一个数组，数组中的元素满足：仅有一个元素出现1次，其余出现3次

然后题目要求为计算出出现次数为奇数次的元素，这类题目通用但是效率不是最优的解法为：
hash表暴力统计，普通的hash表统计适合给定元素范围的，如果没给定范围，可采用map+count，key为数组元素值，val为出现的次数，统计完毕后再进行一次遍历hash表或者map，得到出现奇数次的元素

- 时间复杂度，如果采用hash表，时间复杂度为O(N),map由于底层采用红黑树，查找复杂度为O(logN),故时间复杂度为O(N * logN)，空间复杂度为O(N)

仅有一个元素出现次数为奇数次，其余全为偶数次

136. Single Number

解法一

通过位运算符号易知:

- num ^ num = 0, num ^ 0 = num
- num | num = num, num | 0 = num
- num & num = num, num & 0 = 0

可以发现，对于出现偶数次的数字，异或后的结果为0，出现奇数次 N = even + 1，即可以转化为一个偶数+1个奇数。因此可以对数组中所有的元素进行异或运算，最终得到的结果就是出现奇数次的元素

```
int singleNumber(vector<int>& nums) {
    int res = 0;
    for(size_t idx = 0;idx < nums.size();++idx){
        res ^= nums[idx];
    }
    return res;
}
```

- 时间复杂度O(N)，空间复杂度为O(1)

仅有两个元素出现次数为奇数次，其余全为偶数次

260. Single Number III

显然，出现了两个出现次数为奇数的元素，使用直接异或得到的数值为两个数字的异或结果，而直接通过这个异或的结果显然是不能得到两个异或前的元素的，但是由于两个数值不一样，因此一定存在异或后的结果中某一位为1（假设为从右往左的第一位），显然，数组中这一位出现0，1的次数是奇数次,且两个出现奇数次的数分别分布在两个subarray中，在两个subarray中分别异或就能得到需要寻找的数值

解

- 通过异或的值从右往左的第一位1作为划分标准，将原数组中的所有值划分成两个subarray
- 在两个subarray中分别异或就能得到需要寻找的数值

```
class Solution {
public:
    vector<int> singleNumber(vector<int>& nums) {
        int x_or = 0, f_one_bit = 0;
        for(int i=0;i<nums.size();++i) x_or ^= nums[i];
        f_one_bit = x_or & (~(x_or - 1));
        vector<int> res(2,0);
        for(int i=0;i<nums.size();++i){
            if(nums[i] & f_one_bit) res[0] ^= nums[i];
            else res[1] ^= nums[i];
        }
        return res;
    }
};
```

- 空间复杂度O(N)，空间复杂度O(1)

540. Single Element in a Sorted Array

- 二分查找，根据mid奇偶的情况来判断
- 如果mid是奇数，那么判断前一个元素与自身是否相等，相等则出现一次的元素在后半部分，否则在前半部分
- 如果mid是偶数，那么判断后一个元素与自身是否相等，相等则出现一次的元素在后半部分，否则在前半部分

```
class Solution {
public:
    int singleNonDuplicate(vector<int>& nums) {
        if(nums.size() == 1) return nums[0];
        int left = 0,right = nums.size()-1,mid;
        while(left + 1 < right){
            mid = (left + right) / 2;
            if(mid & 0x1){
                if(nums[mid-1] == nums[mid]) left = mid;
                else right = mid;
            }else{
                if(nums[mid] == nums[mid+1]) left = mid;
                else right = mid;
            }
        }

        return nums[right + 1 >= nums.size() ? right :
                    (nums[right] == nums[right + 1] ?
left : right)];
    }
};
```

- 时间复杂度O(logN)，空间复杂度O(1)

仅有三个元素出现次数为奇数次，其余全为偶数次

对于有三个数出现了奇数次，如果直接全部异或所有的数值，得到的结果意义不大，有没有一种方式，求出三个数中的任意一个，然后转换为2个数出现次数为奇数的情况

解

对于这三个数的任意一位（bit），存在以下情况

- 均为0
- 0和1出现的次数，一个为1次，一个为两次
- 均为1

对于第二种情况，可以发现出现1次的case，就可以划分（按照当前bit是否为1来划分）为数组中某一个数出现了奇数次，其余的为偶数次的情况，剩下的两个异或结果不为零。因此不难推断解法：通过计数当前位为1和为0的数字出现的次数，对当位为1的进行异或结果为xor_A，对为0的进行异或结果为oxr_B，如果是第二种情况，那么出现出现偶数次bit的异或结果一定为非零，此时，返回另一个subarray的异或结果即是三个数中的其中一个

```
#define is_one(n, i) ((n) & 1 << (i))
const int len = 32;
int find_anyone(vector<int> &arr){
    for(int i = 0; i < len; i++){
        int tmpA = 0,tmpB = 0, countA = 0,countB = 0;
        for(int j = 0; j < arr.size(); j++){
            if(is_one(arr[j], i)){
                tmpA ^= arr[j];
                countA++;
            }else{
                tmpB ^= arr[j];
                countB++;
            }
        }
        if(countA & 0x1){ //如果出现奇数次
            if(0 == tmpB) continue; //这里说明三个数字的当前bit相同
            else return tmpA; //tmpB不为0说明为0和1出现的次数，一个为1次，一个为两次
        }else{
            if(0 == tmpA) continue;
            else return tmpB;
        }
    }
}
```

- 空间复杂度O(1),时间复杂度O(N)

仅有一个元素出现1次，其余出现3次

540. Single Element in a Sorted Array

除了一个元素外，其余的出现次数都是3次，那么除了这个元素外，其余的数值所有bit位为1的总和为3的倍数，如果剩下的那个元素当前位如果是1，那么所有数值当前bit出现1的次数为 3 * N + 1

解

- 对32位，分别统计出现1的次数count
- count余3后 |= 到res上，所算即所得

```
class Solution {
    const int len = 32;
public:
    int singleNumber(vector<int>& nums) {
        int res = 0;
        for(int i=0;i<len;++i){
            int count = 0;
            for(size_t j = 0;j < nums.size();++j)count +=
(nums[j] >> i & 0x1);
            res |= ((count % 3) << i);
        }
        return res;
    }
};
```

- 时间复杂度O(N)，空间复杂度O(1)

树的遍历

二叉树的遍历

二叉树树的遍历属于树的基本操作，主要分为：

- 先序遍历
- 中序遍历
- 后续遍历
- 层次遍历
- 其他

对于树的遍历一般存在两种通用解法，一是递归，二是利用栈来代替递归操作

二叉树先序遍历

144. Binary Tree Preorder Traversal.

- 递归
  - 访问节点内容
  - 递归访问左孩子
  - 递归访问右孩子
  - 递归结束条件：root == nullptr

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> res;
        pre_order(root,res);
        return res;
    }

    void pre_order(TreeNode * root,vector<int> &res){
        if(!root) return;
        res.push_back(root->val);
        pre_order(root->left,res);
        pre_order(root->right,res);
    }
};
```

- 栈
  - 1根结点入栈
  - 2若栈不为空，出栈访问；若栈为空结束
  - 3若出栈节点右孩子不为空，将右孩子入栈
  - 4若出栈节点左孩子不为空，将左孩子入栈
  - 5重复步骤2-4

```
vector<int> preorderTraversal(TreeNode* root) {
    vector<int> res;
    if(!root)return res;
    stack<TreeNode *> st;
    st.push(root);
    while(!st.empty()){
        TreeNode * node = st.top();
        st.pop();
        res.push_back(node->val);
        if(node->right)st.push(node->right);
        if(node->left)st.push(node->left);
    }
    return res;
}
```

二叉树中序遍历

94. Binary Tree Inorder Traversal

- 递归
  - 递归访问左孩子
  - 访问节点内容
  - 递归访问右孩子
  - 递归结束条件：root == nullptr
- 栈
  - 1当前节点若不为空则入栈，否则开始执行3
  - 2若当前节点左孩子不为空，将当前节点左孩子入栈，同时更新左孩子为当前节点，重复2
  - 3若栈不为空，出栈节点，并访问；若栈为空，结束
  - 4将出栈节点的右孩子作为当前节点
  - 5重复1-4

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        stack<TreeNode *> st;
        vector<int> res;
        push_left_util_null(root,st);
        while(!st.empty()){
            TreeNode * node = st.top();st.pop();
            res.push_back(node->val);
            node = node->right;
            push_left_until_null(node,st);
        }
        return res;
    }
    void push_left_until_null(TreeNode *
root,stack<TreeNode *> &st){
        while(root){
            st.push(root);
            root=root->left;
        }
    }
};
```

二叉树后序遍历

145. Binary Tree Postorder Traversal

- 递归
  - 递归访问左孩子
  - 递归访问右孩子
  - 访问节点内容
  - 递归结束条件：root == nullptr
- 栈 + reverse
  对于二叉树的先序遍历非递归代码，是先将根节点的右孩子入栈，然后再左孩子入栈。如果我们将根节点的左孩子先入栈，再入栈后孩子呢会发生什么，细心的同学会发现，交换入栈顺序后，得到的序列恰好是后序遍历的reverse，因此将遍历得到的序列最后进行一次reverse即可得到后序遍历.
  - 1根结点入栈
  - 2若栈不为空，出栈访问；若栈为空,执行6
  - 3若出栈节点右孩子不为空，将右孩子入栈
  - 4若出栈节点左孩子不为空，将左孩子入栈
  - 5重复步骤2-4
  - 6对得到的序列进行一次reverse操作

```
vector<int> postorderTraversal(TreeNode* root) {
vector<int> res;
if(!root) return res;
stack<TreeNode *> st;st.push(root);
while(!st.empty()){
  TreeNode * node = st.top();st.pop();
  res.push_back(node->val);
  if(node->left)st.push(node->left);
  if(node->right)st.push(node->right);
}
reverse(res.begin(),res.end());
return res;
}
```

- 栈 (no reverse)
  与先序遍历和中序遍历不同，这两种栈顶元素直接pop，因此不存在麻烦和困难的点，而后序遍历不同，放在栈顶的一般都是二叉树或者二叉子树的根节点，而根节点在后序遍历中恰恰又是最后访问的，如果根节点的左子树和右子树还未访问，那栈顶是不能出栈的，而若左右子树都已经访问，则可以出栈进行访问。现在的问题就是如何解决让程序知道当前节点的左右子树已经访问过或者没有被访问。最简单的方式是采用一个指针pre，标记上一次访问的节点，如果上一次访问的节点是当前栈顶节点的右孩子，说明当前节点的左右子树都已经被访问，进而出栈访问，同时更新pre. 不难总结得出步骤如下：
  - 1若当前节点不为空，入栈.
  - 2若当前节点左孩子不为空，将当前节点左孩子入栈，同时更新左孩子为当前节点，重复2
  - 3若栈不为空，得到栈顶节点（注意此时并没有出栈）
  - 4如果栈顶节点右孩子不为空，判断栈顶节点node右孩子是否为上一次访问的节点(node->right == pre)，如果等于，说明已经访问过了右子树，此时出栈栈顶元素并访问，更新pre为出栈节点；如果不相等，说明右子树未访问，更新当前节点为栈顶右孩子，回到1继续执行。
  - 5若栈顶右孩子为空，则栈顶元素可直接出栈并访问,更新pre为出栈节点
  - 6重复3-5

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> res;
        if(!root) return res;
        stack<TreeNode *> st;
        push_left_until_null(root,st);
        TreeNode * pre = nullptr;
        while(!st.empty()){
            TreeNode * node = st.top();
            if(node->right){
                if(node->right == pre){
                    res.push_back(node->val);
                    st.pop();
                    pre = node;
                }else {
                    node = node->right;
                    push_left_until_null(node,st);
                }
            }else {
                res.push_back(node->val);
                st.pop();
                pre = node;
            }
        }
        return res;
    }
    void push_left_until_null(TreeNode *
root,stack<TreeNode *> & st){
        while(root){
            st.push(root);
            root=root->left;
        }
    }

};
```

二叉树的层次遍历

- 用队列即可实现层次遍历
  - 1若root不为空，入队
  - 2若队列不为空，出队访问
  - 3若出队节点左孩子不为空，入队
  - 4若出队节点右孩子不为空，入队
  - 重复2-4

102. Binary Tree Level Order Traversal.
107. Binary Tree Level Order Traversal II
637. Average of Levels in Binary Tree
103. Binary Tree Zigzag Level Order Traversal

树（森林）的遍历

树，与二叉树相比，孩子节点更多，因此只定义三种遍历方式：

- 先序遍历
- 后序遍历
- 层次遍历 因为一个节点的孩子较多，因此对于中序无法定义

树的先序遍历

- 流程与二叉树先序遍历一致，不过由于结果不同，因此有些许差异
  - 先访问根节点
  - 访问根节点的第一个孩子为根节点的子树（仍按照先序的次序）
  - 访问根节点的第二个孩子为根节点的子树（仍按照先序的次序）
  - ...
  - 访问根节点的第最后一个孩子为根节点的子树（仍按照先序的次序）

树的后序遍历

- 流程与二叉树后序遍历一致，不过由于结果不同，因此有些许差异
  - 访问根节点的第一个孩子为根节点的子树（仍按照后序的次序）
  - 访问根节点的第二个孩子为根节点的子树（仍按照后序的次序）
  - ……
  - 访问根节点的第最后一个孩子为根节点的子树（仍按照后序的次序）
  - 访问根节点

这里重点说一下树的后序遍历的非递归方式

- 栈+reverse，这里不赘述，和二叉树的栈+reverse基本一致，除了些许差别
  - 1根结点入栈
  - 2若栈不为空，出栈访问；若栈为空,执行5
  - 3若出栈节点拥有孩子节点，将孩子依次(从左至右 node->children[0] - node->children[node->children.size()-1])入栈
  - 4重复步骤2-3
  - 5对得到的序列进行一次reverse操作
- 栈(no reverse)
  我们通过二叉树的非递归遍历可知，只要知道当前节点的最后一个孩子已经遍历，则说明此节点所有孩子树已经遍历完成，即可出栈遍历。由于树的孩子节点比较多，我们怎么能知道当前节点是父节点的第几个孩子呢？我们再入栈的时候打包入栈节点的标记，这个标记代表本节点的孩子树已经访问完k(1<=k<=node->children.size())个，若k == node->children.size()出栈访问，否则将node的第k个孩子入栈，同时更新node节点的标记k=k+1;
  - 1根节点入栈，k = 0

- 2若栈不为空，得到栈顶元素（注意此时不是出栈）node和k
- 3若k==node->children.size()说明node节点的孩子已经访问玩，出栈访问之，否则压栈node节点的第k+1个孩子以及其对应的标记k = 0
- 重复2-3

```
class Solution {
public:
    vector<int> postorder(Node* root) {
        stack<pair<Node*,int>> st;
        vector<int> res;
        if(root==NULL) return res;
        st.push(pair(root,0));
        while(!st.empty()){
            auto &[node,idx] = st.top();
            if(idx == node->children.size()){
                res.push_back(node->val);
                st.pop();
            }else st.push(pair(node->children[idx++],0));
        }
        return res;
    }
};
```

590. N-ary Tree Postorder Traversal

树的层次遍历

- 与二叉树的层次遍历一样，同样使用队列来完成层次遍历
  - 1若root不为空，入队
  - 2若队列不为空，出队访问
  - 3若出队节点孩子不为空，将所有孩子依次入队
  - 重复2-4

429. N-ary Tree Level Order Traversal

二叉搜索树

二叉搜索树，又名二叉排序树，它或是一颗空树，若非空满足以下条件：

- 若左子树不为空，则根节点值大于左子树所有节点的值
- 若左子树不为空，则根节点值小于右子树所有节点的值
- 若左右子树不为空，则左右子树也分别为二叉排序树

不难发现一个二叉搜索树的一个重要性质，中序遍历一颗二叉搜索树得到的是一个递增的数组，除此以外，二叉搜索树的不同遍历组合可以唯一确定一颗二叉树。他们分别是：

- 先序遍历+中序遍历
- 后序遍历+中序遍历

对于二叉搜索树出现的题目，主要归纳为

- BST与双向链表
- find_kth smallest
- 给定特定的遍历序列，重建BST/判断是否为BST遍历序列
- LCA（LCA跳转）

BST与双向链表

通过以上分析，BST中序遍历得到一个递增序列。而BST有的节点的指针域是空着的，这些空着的指针域若重新赋值为前驱（left指针域，指向前一个大于本节点值的节点）或者后继（right指针域，指向前一个大于本节点值的节点），即可得到一个双向链表

面试题36. 二叉搜索树与双向链表

- 定义一个pre节点初始为nullptr，用于记录上一次访问过的节点（当前节点的前驱）
- 利用递归或者栈的中序遍历 遍历BST，遍历的同时，修改pre和当前节点node的左右孩子域指针
- pre->right = node;node->left = pre;
- 更新pre = node

当遍历完毕后，得到的就是一个完整的双向链表，如果题目要求是一个循环双向链表，则需要通过root的左孩子一直往前遍历(if !root->left then root = root->left)得到root->left == nullptr位置，当前root为链表头节点，再将root和pre(遍历完BST后pre是BST最后的一个节点)链接起来：root->left = pre;pre->right = pre;

173. Binary Search Tree Iterator

- 转为为list，然后进行next和hasnext操作
  - 这种思路的时间复杂度集中在BST转换为list的过程中为O(N),空间复杂度为O(h)，h为BST树高
  - next操作O(1)，hasNext()操作O(1)
- 转化为list的思路，提前做了转化的工作，然而有的部分转化工作是没必要的（如只有很少的next和hasNext操作），那么可以将转化的过程放到next当中

来做

- 时间复杂度next(),最差O(n)，最好O(1)；hasNext() O(1)

```
class BSTIterator {
private:
    stack<TreeNode *>st;
    void push_left_util_null(TreeNode *
root,stack<TreeNode *> &st){
        while(root){
            st.push(root);
            root=root->left;
        }
    }
public:
    BSTIterator(TreeNode* root) {
        push_left_util_null(root,st);
    }
    /** @return the next smallest number */
    int next() {
        TreeNode * node = st.top();st.pop();
        if(node->right)push_left_util_null(node-
>right,st);
        return node->val;
    }
    /** @return whether we have a next smallest number */
    bool hasNext() {
        return !st.empty();
    }
};
```

230. Kth Smallest Element in a BST

最近公共祖先LCA(Lowest Common Ancestor)

定义

最近公共祖先简称 LCA（Lowest Common Ancestor）。两个节点的最近公共祖先，就是这两个点的公共祖先里面，离根最远的那个。 为了方便，我们记某点集S = {v1,v2,.....,vn}的最小公共祖先为LCA(v1,v2,...,vn)或者LCA(S)

- 性质
  - LCA(v) = v
  - u是v的祖先，则LCA(u,v) = u
  - 若u，v均不是对方的祖先，则u和v在LCA(u,v)的不同子树中

朴素

BST LCA

- BST的最近公共祖先是LCA比较特殊的一种情况，几种判定结果为：
  - 若node节点的值均大于p,q节点的值，则LCA(p,q)在node的左子树
  - 若node节点的值均小于p,q节点的值，则LCA(p,q)在node的右子树
  - 若node节点的值大于等于其中一个节点的值，且小于等于另一个值，则LCA(p,q) == node.
  - 时间复杂度O(h)，h为树高，时间复杂度O(1)

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode*
p, TreeNode* q){
    if(!root || !p || !q) return nullptr;
    if(p->val > q->val)swap(p,q);

    while(root){
        if(root->val < p->val) root = root->right;
        else if(root->val > q->val)root = root->left;
        else return root;
    }
    return nullptr;
}
```

普通二叉树

普通二叉树由于不具备BST的性质，那么最直接的方式是通过计算每个节点的祖先路径，然后对比两个路径得到最近公共祖先

- 时间复杂度，依次查询O(N)，N为树的节点数量

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root,
TreeNode* p, TreeNode* q) {
        vector<TreeNode *>p_ans,q_ans;
        get_ancestor_path(root,p_ans,p);
        get_ancestor_path(root,q_ans,q);
        int len = min(p_ans.size(),q_ans.size());
        TreeNode * res = nullptr;
        for(int i =0;i < len && p_ans[i] ==
q_ans[i];++i)res = p_ans[i];
        return res;
    }
    bool get_ancestor_path(TreeNode *
root,vector<TreeNode *> &path,  TreeNode * p){
        if(!root) return false;
        path.push_back(root);
        if(root == p) return true;
        bool res = get_ancestor_path(root->left,path,p);
        if(!res) res = get_ancestor_path(root-
>right,path,p);
        if(!res) path.pop_back();
        return res;
    }
};

class Solution{
public:
    TreeNode* lowestCommonAncestor(TreeNode* root,
TreeNode* p, TreeNode* q) {
        if(!root || !p || !q) return nullptr;
        if(root == p || root == q) return root;
        TreeNode * left = lowestCommonAncestor(root-
>left,p,q);
        TreeNode * right = lowestCommonAncestor(root-
>right,p,q);
        if(left && right) return root;
        else if(left) return left;
        else if(right) return right;
        else return nullptr;
    }
};
```

然而以上的算法对于少量的查询性能还能接受，如果进行批量查询，则将难以忍受龟速效率

Tarjan算法（DFS+并查集）