1、在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，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。

**class Solution {**

**public:**

**bool Find(int target, vector<vector<int> > array) {**

**int i=array.size()-1;**

**int j=0;**

**while((i>=0) && (j<array[0].size())){//用<=会发生内存溢出的错误，莫名其妙**

**if(array[i][j]==target) return true;**

**if(array[i][j]>target){i--;continue;}**

**if(array[i][j]<target){j++;continue;}**

**}**

**return false;**

**}**

**};**

**2.** 请实现一个函数，将一个字符串中的空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

**class Solution {**

**public:**

**void replaceSpace(char \*str,int length) {**

**//遍历一边字符串找出空格的数量**

**if(str==NULL||length<0)**

**return ;**

**int i=0;**

**int oldnumber=0;//记录以前的长度**

**int replacenumber=0;//记录空格的数量**

**while(str[i]!='\0')**

**{**

**oldnumber++;**

**if(str[i]==' ')**

**{**

**replacenumber++;**

**}**

**i++;**

**}**

**int newlength=oldnumber+replacenumber\*2;//插入后的长度**

**if(newlength>length)//如果计算后的长度大于总长度就无法插入**

**return ;**

**int pOldlength=oldnumber; //注意不要减一因为隐藏个‘\0’也要算里**

**int pNewlength=newlength;**

**while(pOldlength>=0&&pNewlength>pOldlength)//放字符**

**{**

**if(str[pOldlength]==' ') //碰到空格就替换**

**{**

**str[pNewlength--]='0';**

**str[pNewlength--]='2';**

**str[pNewlength--]='%';**

**}**

**else //不是空格就把pOldlength指向的字符装入pNewlength指向的位置**

**{**

**str[pNewlength--]=str[pOldlength];**

**}**

**pOldlength--; //不管是if还是elsr都要把pOldlength前移**

**}**

**}**

**};**

**3.** 输入一个链表，从尾到头打印链表每个节点的值。

**/\*\***

**\* struct ListNode {**

**\* int val;**

**\* struct ListNode \*next;**

**\* ListNode(int x) :**

**\* val(x), next(NULL) {**

**\* }**

**\* };**

**\*/**

**class Solution {**

**public:**

**vector<int> printListFromTailToHead(ListNode\* head) {**

**ListNode\* h;**

**vector<int> result;**

**stack<struct ListNode\*> nodes;//用栈来存储每个节点**

**while(head != NULL){**

**nodes.push(head);**

**head = head->next;**

**}**

**while(!nodes.empty()){**

**h = nodes.top();**

**result.push\_back(h->val);**

**nodes.pop();**

**}**

**return result;**

**}**

**};**

**或者:**

**struct ListNode {**

**int val;**

**struct ListNode \*next;**

**ListNode(int x) :**

**val(x), next(NULL) {**

**}**

**};**

**class Solution{**

**public:**

**vector<int> printListFromTailToHead(struct ListNode\* head){**

**vector<int> result;**

**struct ListNode\* pNode=head;**

**while(pNode!=NULL){**

**result.push\_back(pNode->val);**

**pNode=pNode->next;**

**}**

**reverse(result.begin(),result.end());//applying reverse()**

**return result;**

**}**

**};**

**4.** 输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

**/\*\***

**\* Definition for binary tree**

**\* struct TreeNode {**

**\* int val;**

**\* TreeNode \*left;**

**\* TreeNode \*right;**

**\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}**

**\* };**

**\*/**

**class Solution {**

**public:**

**TreeNode\* reConstructBinaryTree(vector<int> pre,vector<int> vin) {**

**int in\_size = vin.size();//获得序列的长度**

**if (in\_size == 0)**

**return NULL;**

**//分别存储先序序列的左子树，先序序列的右子树，中序序列的左子树，中序序列的右子树**

**vector<int> pre\_left, pre\_right, in\_left, in\_right;**

**int val = pre[0];//先序遍历第一个位置肯定是根节点node,取其值**

**//新建一个树结点，并传入结点值**

**TreeNode\* node = new TreeNode(val);//root node is the first element in pre**

**//p用于存储中序序列中根结点的位置**

**int p = 0;**

**for (p; p < in\_size; ++p){**

**if (vin[p] == val) //Find the root position in in**

**break; //找到即跳出for循环**

**}**

**for (int i = 0; i < in\_size; ++i){**

**if (i < p){**

**//建立中序序列的左子树和前序序列的左子树**

**in\_left.push\_back(vin[i]);//Construct the left pre and in**

**pre\_left.push\_back(pre[i + 1]);//前序第一个为根节点,+1从下一个开始记录**

**}**

**else if (i > p){**

**//建立中序序列的右子树和前序序列的左子树**

**in\_right.push\_back(vin[i]);//Construct the right pre and in**

**pre\_right.push\_back(pre[i]);**

**}**

**}**

**//取出前序和中序遍历根节点左边和右边的子树**

**//递归，再对其进行上述所有步骤，即再区分子树的左、右子子数，直到叶节点**

**node->left = reConstructBinaryTree(pre\_left, in\_left);**

**node->right = reConstructBinaryTree(pre\_right, in\_right);**

**return node;**

**}**

**};**

**5.** 用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。

**class Solution**

**{**

**//思路：用stack1来实现队列的push操作**

**//用stack2来实现队列的pop操作，当stack2为空时**

**//将stack1的数据全部压入stack2，等待队列的pop操作。**

**public:**

**void push(int node) {**

**stack1.push(node);**

**}**

**int pop() {**

**int result;**

**if(stack2.empty()){**

**while(!stack1.empty()){**

**stack2.push(stack1.top());**

**stack1.pop();**

**}**

**}**

**result = stack2.top();**

**stack2.pop();**

**//stack1的第一个元素出去了，恢复stack和stack2**

**while(!stack2.empty()){**

**stack1.push(stack2.top());**

**stack2.pop();**

**}**

**return result;**

**}**

**private:**

**stack<int> stack1;**

**stack<int> stack2;**

**};**

**6.** 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。 输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。 例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。 NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

**class Solution {**

**public:**

**int minNumberInRotateArray(vector<int> rotateArray) {**

**if(rotateArray.size()==0)**

**return 0;**

**int i;**

**for(i=0;i<rotateArray.size()-1;i++){**

**if(rotateArray[i+1] < rotateArray[i])**

**return rotateArray[i+1];**

**}**

**return 0;**

**}**

**};**

**7.** 大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项。

n<=39

**class Solution {**

**public:**

**int Fibonacci(int n) {**

**int f[40];**

**f[0]=0;**

**f[1]=1;**

**int i=0;**

**for(i=2;i<=n;i++){**

**f[i] = f[i-1]+f[i-2];**

**}**

**return f[n];**

**}**

**};**

**8.** 一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

**/\***

**可以考虑，小青蛙每一步跳跃只有两种选择：一是再跳一级阶梯到达第 i 级阶梯，此时小青蛙处于第 i-1 级阶梯；或者再跳两级阶梯到达第 i 级阶梯，此时小青蛙处于第 i-2 级阶梯。**

**于是，i 级阶梯的跳法总和依赖于前 i-1 级阶梯的跳法总数f(i-1)和前 i-2 级阶梯的跳法总数f(i-2)。因为只有两种可能性，所以，f(i)=f(i-1)+f(i-2);**

**依次类推，可以递归求出n级阶梯跳法之和。**

**\*/**

**class Solution {**

**public:**

**int jumpFloor(int number) {**

**int f[number];**

**f[0]=0;**

**f[1]=1;**

**f[2]=2;**

**int i=0;**

**for(i=3;i<=number;i++){**

**f[i] = f[i-1]+f[i-2];**

**}**

**return f[number];**

**}**

**};**

**9.** 一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

**class Solution {**

**public:**

**int jumpFloorII(int number) {**

**if (number <= 0)**

**return -1;**

**if (number == 1)**

**return 1;**

**return 2 \* jumpFloorII(number - 1);**

**}**

**};**

**10.** 我们可以用2\*1的小矩形横着或者竖着去覆盖更大的矩形。请问用n个2\*1的小矩形无重叠地覆盖一个2\*n的大矩形，总共有多少种方法？

**class Solution {**

**public:**

**int rectCover(int number) {**

**int f[1000000];**

**f[0]=0;**

**f[1]=1;**

**f[2]=2;**

**if(number>=3)**

**for(int i=3;i<=number;i++){**

**f[i] = f[i-1]+f[i-2];**

**}**

**return f[number];**

**}**

**};**

**11、**输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。

**public static int NumberOf1(int n) {**

**int count = 0;**

**while (n != 0) {**

**++count;**

**n = (n - 1) & n;//n&(n-1)表示将一个整数的二进制形式的最后一个1变为0.所以可以利用这个特性来计算1的数量。**

**}**

**return count;**

**}**

**12、**给定一个double类型的浮点数base和int类型的整数exponent。求base的exponent次方。

链接：<https://www.nowcoder.com/questionTerminal/1a834e5e3e1a4b7ba251417554e07c00>  
来源：牛客网

/\*剑指书中细节：

\*1.当底数为0且指数<0时

\*会出现对0求倒数的情况，需进行错误处理，设置一个全局变量；

\*2.判断底数是否等于0

\*由于base为double型，不能直接用==判断

\*3.优化求幂函数

\*当n为偶数，a^n =（a^n/2）\*（a^n/2）

\*当n为奇数，a^n = a^[(n-1)/2] \* a^[(n-1)/2] \* a

\*时间复杂度O(logn)

\*/

**13.** 输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

**class Solution {**

**public:**

**void reOrderArray(vector<int> &array) {**

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

**if(array[i]%2==0)**

**for(int j=i+1;j<array.size();j++)**

**if(array[j]%2!=0){//把j移动到i,i~j的往后面的顺移**

**int odd = array[j];**

**for(int k =j;k>=i+1;k--)**

**array[k]=array[k-1];**

**array[i]=odd;**

**break;**

**}**

**}**

**};**

**14.** 输入一个链表，输出该链表中倒数第k个结点。

**/\***

**struct ListNode {**

**int val;**

**struct ListNode \*next;**

**ListNode(int x) :**

**val(x), next(NULL) {**

**}**

**};\*/**

**class Solution {**

**public:**

**ListNode\* FindKthToTail(ListNode\* pListHead, unsigned int k) {**

**if(pListHead == NULL)**

**return NULL;**

**int a=0;**

**ListNode \*p = pListHead;**

**while(p!=NULL){**

**p = p->next;**

**a++;**

**}**

**if(a<k)**

**return NULL;**

**a=a-k;**

**p=pListHead;**

**while(a){**

**p=p->next;**

**a--;**

**}**

**return p;**

**}**

**};**

**15.** 输入一个链表，反转链表后，输出链表的所有元素。

**/\***

**struct ListNode {**

**int val;**

**struct ListNode \*next;**

**ListNode(int x) :**

**val(x), next(NULL) {**

**}**

**};\*/**

**class Solution {**

**public:**

**ListNode\* ReverseList(ListNode\* pHead) {**

**ListNode\* p,\*q;**

**vector<int>a;**

**p=pHead;**

**q=pHead;**

**while(p!=NULL){**

**a.push\_back(p->val);**

**p=p->next;**

**}**

**int i=a.size()-1;**

**while(i>=0){**

**q->val = a[i];**

**q=q->next;**

**i--;**

**if(i==0){**

**q->val = a[0];**

**q->next=NULL;**

**return pHead;**

**}**

**}**

**return pHead;**

**}**

**};**

**class Solution {**

**public:**

**ListNode\* ReverseList(ListNode\* pHead) {**

**if(pHead==NULL)**

**return NULL;**

**//head为当前节点，如果当前节点为空的话，那就什么也不做，直接返回null；**

**ListNode \*pre = NULL;**

**ListNode \*next = NULL;**

**//当前节点是head，pre为当前节点的前一节点，next为当前节点的下一节点**

**//需要pre和next的目的是让当前节点从pre->head->next1->next2变成pre<-head next1->next2**

**//即pre让节点可以反转所指方向，但反转之后如果不用next节点保存next1节点的话，此单链表就此断开了**

**//所以需要用到pre和next两个节点**

**//1->2->3->4->5**

**//1<-2<-3 4->5**

**while(pHead!=NULL){**

**//做循环，如果当前节点不为空的话，始终执行此循环，此循环的目的就是让当前节点从指向next到指向pre**

**//如此就可以做到反转链表的效果**

**//先用next保存head的下一个节点的信息，保证单链表不会因为失去head节点的原next节点而就此断裂**

**next = pHead->next;**

**//保存完next，就可以让head从指向next变成指向pre了，代码如下**

**pHead->next = pre;**

**//head指向pre后，就继续依次反转下一个节点**

**//让pre，head，next依次向后移动一个节点，继续下一次的指针反转**

**pre = pHead;**

**pHead = next;**

**}**

**//如果head为null的时候，pre就为最后一个节点了，但是链表已经反转完毕，pre就是反转后链表的第一个节点**

**//直接输出pre就是我们想要得到的反转后的链表**

**return pre;**

**}**

**};**

**16.** 输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

**/\***

**struct ListNode {**

**int val;**

**struct ListNode \*next;**

**ListNode(int x) :**

**val(x), next(NULL) {**

**}**

**};\*/**

**class Solution {**

**public:**

**ListNode\* Merge(ListNode\* pHead1, ListNode\* pHead2)**

**{**

**if(!pHead1)**

**return pHead2;**

**if(!pHead2)**

**return pHead1;**

**ListNode\* Head;**

**ListNode\* p;**

**//取较小值作头结点**

**if(pHead1->val<=pHead2->val){**

**Head=pHead1;**

**pHead1=pHead1->next;**

**}**

**else{**

**Head=pHead2;**

**pHead2=pHead2->next;**

**}**

**//开始遍历合并**

**p=Head;                                                   //p为合并后的链表的工作指针**

**while(pHead1&&pHead2){                       //当有一个链表到结尾时，循环结束**

**if(pHead1->val<=pHead2->val){          //如果链表1的结点小于链表2的结点**

**p->next=pHead1;                            //取这个结点加入合并链表**

**pHead1=pHead1->next;                 //链表1后移一位**

**p=p->next;                                      //工作指针后移一位**

**}**

**else{                                               //否则取链表2的结点**

**p->next=pHead2;**

**pHead2=pHead2->next;**

**p=p->next;**

**}**

**}**

**if(pHead1 == NULL)           //链表1遍历完了**

**p->next = pHead2;         //如果链表2也遍历完了，则pHead2=NULL**

**if(pHead2 == NULL)            //链表2遍历完了**

**p->next = pHead1;         ///如果链表1也遍历完了，则pHead1=NULL**

**return Head;**

**}**

**};**

**17.** 输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）

**/\***

**struct TreeNode {**

**int val;**

**struct TreeNode \*left;**

**struct TreeNode \*right;**

**TreeNode(int x) :**

**val(x), left(NULL), right(NULL) {**

**}**

**};\*/**

**class Solution {**

**public:**

**bool HasSubtree(TreeNode\* pRoot1, TreeNode\* pRoot2)**

**{**

**if(pRoot1==NULL || pRoot2==NULL)**

**return false;**

**bool result = false;**

**if(pRoot1->val == pRoot2->val)**

**result = isSubtree(pRoot1,pRoot2);**

**if (result == false)**

**result = HasSubtree(pRoot1->left,pRoot2);**

**if (result == false)**

**result = HasSubtree(pRoot1->right,pRoot2);**

**return result;**

**}**

**bool isSubtree(TreeNode\* tree1, TreeNode\* tree2){**

**if(tree1 == NULL &&tree2 == NULL) return true;**

**else if(tree1 != NULL &&tree2 == NULL) return true;**

**else if(tree1 == NULL &&tree2 != NULL) return false;**

**else{**

**if(tree1->val != tree2->val)**

**return false;**

**return isSubtree(tree1->left, tree2->left) && isSubtree(tree1->right, tree2->right);**

**}**

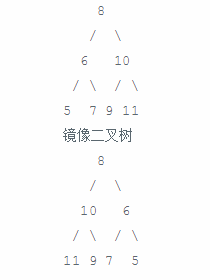
**}**

**};**

**18.** 操作给定的二叉树，将其变换为源二叉树的镜像。

**输入描述:**

二叉树的镜像定义：源二叉树



**/\***

**struct TreeNode {**

**int val;**

**struct TreeNode \*left;**

**struct TreeNode \*right;**

**TreeNode(int x) :**

**val(x), left(NULL), right(NULL) {**

**}**

**};\*/**

**//步骤：1.交换根结点的左右子结点；2.将左右子结点看作根结点进行1的操作**

**class Solution {**

**public:**

**void Mirror(TreeNode \*pRoot) {**

**//根节点为空或者没有左右子结点的情况，函数返回**

**if((pRoot==NULL)||(pRoot->left==NULL && pRoot->right==NULL))**

**return;**

**TreeNode \*p;**

**p=pRoot->left;**

**pRoot->left=pRoot->right;**

**pRoot->right = p;**

**if(pRoot->left)**

**Mirror(pRoot->left);**

**if(pRoot->right)**

**Mirror(pRoot->right);**

**}**

**};**

**19.** 输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下矩阵： 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 则依次打印出数字1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.

class Solution {

public:

vector<int> printMatrix(vector<vector<int> > matrix) {

int row1 = 0, row2 = matrix.size();//row2行数

int col1 = 0, col2 = matrix[0].size();//col2列数

vector<int> result;

while (true)

{

//从左到右打印

for (int i = col1; i < col2; i++)

result.push\_back(matrix[row1][i]);

row1++;

if (row1 >= row2)break; //循环终止条件

//从上往下打印

for (int i = row1; i < row2; i++)

result.push\_back(matrix[i][col2-1]);

col2--;

if (col1 >= col2)break;

//从右往左打印

for (int i = col2-1; i >= col1; i--)

result.push\_back(matrix[row2-1][i]);

row2--;

if (row1 >= row2)break;

//从下往上打印

for (int i = row2-1; i >= row1; i--)

result.push\_back(matrix[i][col1]);

col1++;

if (col1 >= col2)break;

}

return result;

}

};

**20.** 定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的min函数。

**class Solution {**

**public:**

**void push(int value) {**

**stack1.push(value);**

**}**

**void pop() {**

**stack1.pop();**

**}**

**int top() {**

**return stack1.top();**

**}**

**int min() {**

**int min = stack1.top();**

**while(!stack1.empty()){**

**if(min>stack1.top())**

**min=stack1.top();**

**stack2.push(stack1.top());**

**stack1.pop();**

**}**

**while(!stack2.empty()){**

**stack1.push(stack2.top());**

**stack2.pop();**

**}**

**return min;**

**}**

**private:**

**stack<int>stack1;**

**stack<int>stack2;**

**};**

**21.** 输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列1,2,3,4,5是某栈的压入顺序，序列4，5,3,2,1是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的）

**class Solution {**

**public:**

**bool IsPopOrder(vector<int> pushV,vector<int> popV) {**

**if(pushV.size() == 0) return false;**

**vector<int> stack;**

**for(int i = 0,j = 0 ;i < pushV.size();){**

**stack.push\_back(pushV[i++]);**

**while(j < popV.size() && stack.back() == popV[j]){**

**stack.pop\_back();**

**j++;**

**}**

**}**

**return stack.empty();**

**}**

**};**

**22.** 从上往下打印出二叉树的每个节点，同层节点从左至右打印。

/\*

struct TreeNode {

int val;

struct TreeNode \*left;

struct TreeNode \*right;

TreeNode(int x) :

val(x), left(NULL), right(NULL) {

}

};\*/

class Solution {

public:

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

vector<int>result;

queue<TreeNode\*>Q;

if(root)

Q.push(root);

while(!Q.empty()){

TreeNode\* p =Q.front();

Q.pop();

result.push\_back(p->val);

if(p->left)

Q.push(p->left);

if(p->right)

Q.push(p->right);

}

return result;

}

};

23. 输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出Yes,否则输出No。假设输入的数组的任意两个数字都互不相同。

bool verifySquenceOfBST(int squence[], int length)  
{  
      if(squence == NULL || length <= 0)  
            return false;  
  
      // root of a BST is at the end of post order traversal squence  
      int root = squence[length - 1];  
  
      // the nodes in left sub-tree are less than the root  
      int i = 0;  
      for(; i < length - 1; ++ i)  
      {  
            if(squence[i] > root)  
                  break;  
      }  
  
      // the nodes in the right sub-tree are greater than the root  
      int j = i;  
      for(; j < length - 1; ++ j)  
      {  
            if(squence[j] < root)  
                  return false;  
      }  
  
      // verify whether the left sub-tree is a BST  
      bool left = true;  
      if(i > 0)  
            left = verifySquenceOfBST(squence, i);  
  
      // verify whether the right sub-tree is a BST  
      bool right = true;  
      if(i < length - 1)  
            right = verifySquenceOfBST(squence + i, length - i - 1);  
  
      return (left && right);  
}

24. 输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。

* 递归先序遍历树， 把结点加入路径。
* 若该结点是叶子结点则比较当前路径和是否等于期待和。
* 弹出结点，每一轮递归返回到父结点时，当前路径也应该回退一个结点

**/\***

**struct TreeNode {**

**int val;**

**struct TreeNode \*left;**

**struct TreeNode \*right;**

**TreeNode(int x) :**

**val(x), left(NULL), right(NULL) {**

**}**

**};\*/**

**class Solution {**

**public:**

**int sum = 0;**

**vector<int> path;**

**vector <vector<int>> paths;**

**int flag = 0;**

**vector<vector<int> > FindPath(TreeNode\* root, int expectNumber) {**

**paths = {};**

**if (root != NULL)**

**FindOnePath(root, expectNumber);**

**return paths;**

**}**

**void FindOnePath(TreeNode\* &root, int expectNumber) {**

**if (root == NULL){**

**if(flag==0){**

**sum = 0;**

**for (auto it = path.begin(); it != path.end(); ++it)**

**sum += \*it;**

**if (sum == expectNumber)**

**paths.push\_back(path);**

**flag = 1;**

**}**

**else**

**flag = 0;**

**return;**

**}**

**else{**

**path.push\_back(root->val);**

**FindOnePath(root->left, expectNumber);**

**FindOnePath(root->right, expectNumber);**

**path.pop\_back();**

**}**

**}**

**};**

**class Solution {**

**public:**

**vector<vector<int>> result;**

**vector<int>temp;**

**vector<vector<int> > FindPath(TreeNode\* root,int expectNumber) {**

**if(root != NULL)**

**findOnePath(root,expectNumber);**

**return result;**

**}**

**void findOnePath(TreeNode\* root,int expectNumber){**

**temp.push\_back(root->val);**

**if(expectNumber - root->val==0 && root->right ==NULL && root->left == NULL)**

**result.push\_back(temp);**

**if(root->left != NULL)findOnePath(root->left,expectNumber-root->val);**

**if(root->right != NULL)findOnePath(root->right,expectNumber-root->val);**

**temp.pop\_back();//**深度遍历完一条路径后要回退

**}**

**};**

**25.** 输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

从头遍历链表，对每一个节点复制一个，插在它后边；接着遍历这个链表，那么原链表中节点的特殊指针若为空，则复制链表对应节点的特殊指针也为空，若源链表中结点的特殊指针不为空，那复制链表中对应节点的特殊指针为原链表中结点特殊指针的->next。

**/\***

**struct RandomListNode {**

**int label;**

**struct RandomListNode \*next, \*random;**

**RandomListNode(int x) :**

**label(x), next(NULL), random(NULL) {**

**}**

**};**

**\*/**

**class Solution {**

**public:**

**RandomListNode\* Clone(RandomListNode\* pHead)**

**{**

**//第一步：复制next单链表，第二步：复制random单链表，第三步：拆分**

**if(pHead == NULL)**

**return NULL;**

**RandomListNode \*pNode;**

**pNode = pHead;**

**RandomListNode \*nextNode = pNode->next;**

**//复制next单链表**

**while(pNode){**

**RandomListNode \*temp = new RandomListNode(pNode->label);**

**pNode->next = temp;**

**temp->next = nextNode;**

**pNode = temp->next;**

**if (pNode){**

**nextNode = pNode->next;//这里是判断链表本身只有一个结点时和遍历到最后时两种情况**

**}**

**}**

**//复制random单链表**

**pNode = pHead;**

**while(pNode){**

**if(pNode->random != NULL)**

**pNode->next->random = pNode->random->next;**

**else**

**pNode->next->random = NULL;**

**pNode = pNode->next->next;**

**}**

**//拆分**

**pNode = pHead;**

**RandomListNode \*clonepHead = pNode->next;**

**RandomListNode \*clonepNode = clonepHead;**

**while(pNode){**

**pNode->next = clonepNode->next;**

**pNode = pNode->next;**

**if(pNode){**

**clonepNode->next = pNode->next;**

**clonepNode = clonepNode->next;**

**}**

**}**

**return clonepHead;**

**}**

**};**

**也可以先复制单链表，再考虑特殊指针：每找一次特殊指针，从头遍历原链表一次（因为特殊指针有可能指向当前节点的前面节点），通过比较顺序指针和特殊指针是否相等（为什么不比较值呢？因为链表中没说值不可以重复），记录顺序指针走的步数，同理在赋值链表中，走这么多步，找到谁，就把谁赋值给当前节点的特殊指针。时间复杂度为O（n2）。**

**26.** 输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

链接：<https://www.nowcoder.com/questionTerminal/947f6eb80d944a84850b0538bf0ec3a5>  
来源：牛客网

方法一：非递归版

解题思路：

1.核心是中序遍历的非递归算法。

2.修改当前遍历节点与前一遍历节点的指针指向。

    import java.util.Stack;

    public TreeNode ConvertBSTToBiList(TreeNode root) {

        if(root==null)

            return null;

        Stack<TreeNode> stack = new Stack<TreeNode>();

        TreeNode p = root;

        TreeNode pre = null;// 用于保存中序遍历序列的上一节点

        boolean isFirst = true;

        while(p!=null||!stack.isEmpty()){

            while(p!=null){

                stack.push(p);

                p = p.left;

            }

            p = stack.pop();

            if(isFirst){

                root = p;// 将中序遍历序列中的第一个节点记为root

                pre = root;

                isFirst = false;

            }else{

                pre.right = p;

                p.left = pre;

                pre = p;

            }

            p = p.right;

        }

        return root;

    }

方法二：递归版

解题思路：

1.将左子树构造成双链表，并返回链表头节点。

2.定位至左子树双链表最后一个节点。

3.如果左子树链表不为空的话，将当前root追加到左子树链表。

4.将右子树构造成双链表，并返回链表头节点。

5.如果右子树链表不为空的话，将该链表追加到root节点之后。

6.根据左子树链表是否为空确定返回的节点。

    public TreeNode Convert(TreeNode root) {

        if(root==null)

            return null;

        if(root.left==null&&root.right==null)

            return root;

        // 1.将左子树构造成双链表，并返回链表头节点

        TreeNode left = Convert(root.left);

        TreeNode p = left;

        // 2.定位至左子树双链表最后一个节点

        while(p!=null&&p.right!=null){

            p = p.right;

        }

        // 3.如果左子树链表不为空的话，将当前root追加到左子树链表

        if(left!=null){

            p.right = root;

            root.left = p;

        }

        // 4.将右子树构造成双链表，并返回链表头节点

        TreeNode right = Convert(root.right);

        // 5.如果右子树链表不为空的话，将该链表追加到root节点之后

        if(right!=null){

            right.left = root;

            root.right = right;

        }

        return left!=null?left:root;

    }

方法三：改进递归版

解题思路：

思路与方法二中的递归版一致，仅对第2点中的定位作了修改，新增一个全局变量记录左子树的最后一个节点。

    // 记录子树链表的最后一个节点，终结点只可能为只含左子树的非叶节点与叶节点

    protected TreeNode leftLast = null;

    public TreeNode Convert(TreeNode root) {

        if(root==null)

            return null;

        if(root.left==null&&root.right==null){

            leftLast = root;// 最后的一个节点可能为最右侧的叶节点

            return root;

        }

        // 1.将左子树构造成双链表，并返回链表头节点

        TreeNode left = Convert(root.left);

        // 3.如果左子树链表不为空的话，将当前root追加到左子树链表

        if(left!=null){

            leftLast.right = root;

            root.left = leftLast;

        }

        leftLast = root;// 当根节点只含左子树时，则该根节点为最后一个节点

        // 4.将右子树构造成双链表，并返回链表头节点

        TreeNode right = Convert(root.right);

        // 5.如果右子树链表不为空的话，将该链表追加到root节点之后

        if(right!=null){

            right.left = root;

            root.right = right;

        }

        return left!=null?left:root;

    }

## End