#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

#include <unordered\_map>

using namespace std;

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

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

};

class Solution {

public:

// 合并两个树， 位置相同的地方值相加

TreeNode\* mergeTrees(TreeNode\* t1, TreeNode\* t2) {

if (!t1) return t2;

if (!t2) return t1;

t1->val = t1->val + t2->val;

t1->left = mergeTrees(t1->left, t2->left);

t1->right = mergeTrees(t1->right, t2->right);

return t1;

}

//返回树每层的均值

void sumOfLevels(TreeNode\* root, int level, vector<double>& sum, vector<double>& num) {

if (!root) return;

if (level >= sum.size())

{

sum.push\_back(0);

num.push\_back(0);

}

sum[level] += root->val;

num[level]++;

sumOfLevels(root->left, level + 1, sum, num);

sumOfLevels(root->right, level + 1, sum, num);

}

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

vector<double> sum;

vector<double> num;

int level = 0;

sumOfLevels(root, 0, sum, num);

vector<double> res(sum.size(), 0.0);

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

res[i] = sum[i] / num[i] \* 1.0;

return res;

}

//求树的最大深度

int maxDepth(TreeNode\* root) {

if (!root) return 0;

return 1 + max(maxDepth(root->left), maxDepth(root->right));

}

//翻转二叉树

TreeNode\* invertTree(TreeNode\* root) {

if (!root) return root;

TreeNode\* temp = root->left;

root->left = invertTree(root->right);

root->right = invertTree(temp);

return root;

}

//是否存在两个数的和为目标值，二叉搜索树

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

if (!root) return;

inorder(root->left, res);

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

inorder(root->right, res);

}

bool findTarget(TreeNode\* root, int k) {

vector<int> res;

inorder(root, res);

int flag = 0;

for (auto i = res.end() - 1; i >= res.begin(); i--) {

int err = k - \*i;

for (auto j = res.begin(); j < i; j++) {

if (\*j == err)

return true;

if (\*j > err)

break;

}

}

return false;

}

//树转数组

string tree2str(TreeNode\* t) {

string res;

if (!t) return res;

res += to\_string(t->val);

if (!t->left && !t->right) return res;

if (!t->right) {

res += "(" + tree2str(t->left) + ")";

return res;

}

res += "(" + tree2str(t->left) + ")";

res += "(" + tree2str(t->right) + ")";

return res;

}

//二叉搜索树， 从后求和

int convertBST\_sum = 0;

TreeNode\* convertBST(TreeNode\* root) {

if (!root) return root;

convertBST(root->right);

convertBST\_sum += root->val;

root->val = convertBST\_sum;

convertBST(root->left);

return root;

}

//求左叶子节点的和

int sumOfLeftLeaves(TreeNode\* root) {

if (!root) return 0;

if (!root->left) return sumOfLeftLeaves(root->right);

if (!root->left->left && !root->left->right)

return root->left->val + sumOfLeftLeaves(root->right);

return sumOfLeftLeaves(root->left) + sumOfLeftLeaves(root->right);

}

//所有子树和的差 的和

int sum(TreeNode\* root, int err\_sum) {

if (!root) return 0;

int l = sum(root->left, err\_sum);

int r = sum(root->right, err\_sum);

err\_sum += abs(l - r);

return l+r+root->val;

}

int findTilt(TreeNode\* root) {

int err\_sum = 0;

sum(root, err\_sum);

return err\_sum;

}

//判断两棵树是否相同

bool isSameTree(TreeNode\* p, TreeNode\* q) {

if (!p && !q) return true;

if (!p || !q) return false;

if (p->val == q->val)

return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);

return false;

}

int diameterOfBinaryTree(TreeNode\* root) {

}

//数组变平衡搜索树

template<typename iterator>

TreeNode\* sortedArrayToBST(iterator begin, iterator end) {

if (begin >= end) return nullptr;

auto mid = begin + (end - begin) / 2;

TreeNode\* root = new TreeNode(\*mid);

root->left = sortedArrayToBST(begin, mid);

root->right = sortedArrayToBST(mid + 1, end);

return root;

}

TreeNode\* sortedArrayToBST(vector<int>& nums) {

return sortedArrayToBST(nums.begin(), nums.end());

}

//判断一棵树是否为另一颗的子树

bool isSubtree(TreeNode\* s, TreeNode\* t) {

if (!s) return t == nullptr;

if (isSameTree(s, t)) return true;

if (isSubtree(s->left, t)) return true;

if (isSubtree(s->right, t)) return true;

return false;

}

//广度遍历

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

if (!root) return;

if (level >= res.size())

res.push\_back(vector<int>{});

res[level].push\_back(root->val);

levelOrderBottom(root->left, level + 1, res);

levelOrderBottom(root->right, level + 1, res);

}

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

vector<vector<int>> res;

levelOrderBottom(root, 0 , res);

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

return res;

}

//路径和

void pathSum(TreeNode\* root, int sum, int num) {

if (!root) return;

}

int pathSum(TreeNode\* root, int sum) {

}

//两个节点的最小公共祖先, BST

TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

if (!root) return root;

if (!p || q == root) return q;

if (!q || p == root) return p;

int left, right;

if (p->val < q->val) {

left = p->val;

right = q->val;

}

else {

left = q->val;

right = p->val;

}

if (left < root->val && root->val < right)

return root;

if (right < root->val)

return lowestCommonAncestor(root->left, p, q);

if (left > root->val)

return lowestCommonAncestor(root->right, p, q);

}

//判断是否为中心对称树

bool isSymmetric(TreeNode\* l, TreeNode\* r) {

if (!l) return r == nullptr;

if (!r) return false;

if (l->val != r->val) return false;

return isSymmetric(l->left, r->right) && isSymmetric(l->right, r->left);

}

bool isSymmetric(TreeNode\* root) {

if (!root) return true;

return isSymmetric(root->left, root->right);

}

//遍历树的路径

void binaryTreePaths(TreeNode\* root, vector<string>& res, vector<string>& path) {

if (!root) return;

if (!root->left && !root->right) {

string temp;

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

temp += (path[i] + "->");

temp += to\_string(root->val);

res.push\_back(temp);

}

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

binaryTreePaths(root->left, res, path);

binaryTreePaths(root->right, res, path);

path.pop\_back();

}

vector<string> binaryTreePaths(TreeNode\* root) {

vector<string> res;

vector<string> path;

binaryTreePaths(root, res, path);

return res;

}

//判读是否为平衡树

int isbalanced(TreeNode\* root) {

if (!root) return 0;

int h\_left = isbalanced(root->left);

int h\_right = isbalanced(root->right);

if (h\_left > -1 && h\_right > -1 && abs(h\_left - h\_right) <= 1)

return max(h\_left, h\_right) + 1;

else

return -1;

}

bool isBalanced(TreeNode\* root) {

return isbalanced(root) > -1;

}

//是否有路径和为目标

bool hasPathSum(TreeNode\* root, int sum) {

if (!root) return false;

if (!root->left && !root->right)

return root->val == sum;

if (hasPathSum(root->left, sum - root->val))

return true;

return hasPathSum(root->right, sum - root->val);

}

//树的最短路径

int minDepth(TreeNode\* root) {

if (!root) return 0;

if (!root->right) return minDepth(root->left) + 1;

if (!root->left) return minDepth(root->right) + 1;

return min(minDepth(root->left), minDepth(root->right)) + 1;

}

};

#include <iostream>

#include <vector>

using namespace std;

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) :val(x), next(NULL) {}

};

class Solution\_list {

public:

//翻转链表

ListNode\* reverseList(ListNode\* head) {

if (!head || !head->next) return head;

ListNode dummy(-1);

dummy.next = head;

ListNode\* pre = head;

ListNode\* now = head->next;

while (now) {

pre->next = now->next;

now->next = dummy.next;

dummy.next = now;

now = pre->next;

}

return dummy.next;

}

//删除节点

void deleteNode(ListNode\* node) {

\*(node) = \*(node->next);

}

//从排序链表中删除相同元素

ListNode\* deleteDuplicates(ListNode\* head) {

if (!head || !head->next) return head;

ListNode dummy(-1);

dummy.next = head;

ListNode\* pre = &dummy;

ListNode\* now = head;

while (now) {

if (now->next && now->val == now->next->val) {

ListNode\* del = now;

pre->next = now->next;

now = pre->next;

delete del;

}

else {

pre = now;

now = now->next;

}

}

return dummy.next;

}

//合并排序链表, 升序

ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) {

if (!l1) return l2;

if (!l2) return l1;

if (l1->val <= l2->val) {

l1->next = mergeTwoLists(l1->next, l2);

return l1;

}

l2->next = mergeTwoLists(l1, l2->next);

return l2;

}

//检测链表是否有环

bool hasCycle(ListNode \*head) {

ListNode\* slow = head;

ListNode\* fast = head;

while (fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

if (slow == fast)

return true;

}

return false;

}

//判断链表是否是回文

#include <vector>

bool isPalindrome(ListNode\* head) {

if (!head || !head->next) return true;

ListNode\* slow = head;

ListNode\* fast = head;

vector<ListNode\*> left;

while (fast && fast->next) {

fast = fast->next->next;

left.push\_back(slow);

slow = slow->next;

}

if (fast) slow = slow->next;

while (slow) {

if (slow->val == left.back()->val) {

slow = slow->next;

left.pop\_back();

}

else

return false;

}

return true;

}

//删除链表指定元素

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

if (!head) return head;

ListNode dummy(-1);

dummy.next = head;

ListNode\* pre = &dummy;

ListNode\* now = head;

while (now) {

if (now->val == val) {

ListNode\* del = now;

pre->next = now->next;

now = pre->next;

delete del;

}

else {

pre = pre->next;

now = now->next;

}

}

return dummy.next;

}

//找到两个链表的起始交叉点

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

if (!headA || !headB) return nullptr;

int lenA = 0, lenB = 0;

ListNode\* A = headA, \*B = headB;

while (A) {

A = A->next;

lenA++;

}

while (B) {

B = B->next;

lenB++;

}

int err = lenA - lenB;

A = headA, B = headB;

while (err < 0) {

B = B->next;

err++;

}

while (err > 0) {

A = A->next;

err--;

}

while (A && B) {

if (A == B)

return A;

else {

A = A->next;

B = B->next;

}

}

return nullptr;

}

};

//归并

template<typename iterator, typename type>

void merge(iterator begin, iterator mid, iterator end) {

vector<type> num2;

copy(mid, end, back\_inserter(num2));

auto lr = mid, rr = num2.end(), r = end ;

while (lr > begin && rr > num2.begin()) {

\*(--r) = \*(lr-1) >= \*(rr-1) ? \*(--lr) : \*(--rr);

}

while (rr > num2.begin())

\*(--r) = \*(--rr);

}

template<typename iterator>

void merge\_sort(iterator begin, iterator end) {

if (begin >= end - 1) return;

auto mid = begin + (end - begin) / 2;

merge\_sort(begin, mid);

merge\_sort(mid, end);

merge<vector<int>::iterator, int>(begin, mid, end);

}

//快排

template<typename iterator>

void quick\_sort(iterator left, iterator right) {

if (left < right) {

auto l = left, r = right;

auto mid\_val = \*left;

while (l < r) {

while (l < r && \*r >= mid\_val) r--;

if (l < r) \*(l++) = \*r;

while (l < r && \*l < mid\_val) l++;

if (l < r) \*(r--) = \*l;

}

\*l = mid\_val;

if(l > left)

quick\_sort(left, l - 1);

if(right > l)

quick\_sort(l + 1, right);

}

}