二叉树

jask

09/30/2024

102 二叉树层序遍历

```
class Solution {
    vector<vector<int>> level_order;
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        std::list<TreeNode*> list;
        if(root!=nullptr)
        list.push_back(root);
        while(list.size()){
            int size=list.size();
            vector<int> vec;
            for(int i=0;i<size;i++){
                auto node=list.front();
                list.pop_front();
                vec.push_back(node->val);
                if(node->left) list.push_back(node->left);
                if(node->right) list.push_back(node->right);
            }
            level_order.emplace_back(vec);
        }
        return level_order;
    }
};
```

199 二叉树的右视图

给定一个二叉树的根节点 `root`，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值

```
class Solution {
 public:
  vector<int> rightSideView(TreeNode *root) {
      if(root==nullptr) return {};
    vector<vector<int>> levelOrder;
    queue<TreeNode *> que;
    que.push(root);
    while (!que.empty()) {
      int n = que.size();
      levelOrder.push_back(vector<int>());
      for (int i = 0; i < n; i++) {
        auto node = que.front();
        que.pop();
        levelOrder.back().push_back(node->val);
        if (node->left) que.push(node->left);
        if (node->right) que.push(node->right);
      }
    }
    vector<int> ans;
    for (auto i : levelOrder) {
```

```
      ans.push_back(i.back());
    }
    return ans;
  }
};
```

637 二叉树的层平均值

```
class Solution {
public:
    vector<double> averageOfLevels(TreeNode* root) {
        vector<double> ans;
        std::queue<TreeNode*> que;
        if(root!=nullptr) que.push(root);
        while(que.size()){
            int size=que.size();
            double level_sum=0;
            for(int i=0;i<size;i++){
                auto node=que.front();
                que.pop();
                level_sum+=node->val;
                if(node->left) que.push(node->left);
                if(node->right) que.push(node->right);
            }
            ans.push_back((double)(level_sum/size));
        }
        return ans;
    }
};
```

429 N 叉树的层序遍历

```
class Solution {
public:
    vector<vector<int>> levelOrder(Node* root) {
        vector<vector<int>> level_order;
        std::queue<Node*> que;
        if(root) que.push(root);
        while(que.size()){
            int n=que.size();
            vector<int> level;
            for(int i=0;i<n;i++){
                auto node=que.front();
                que.pop();
                level.push_back(node->val);
                for(auto each: node->children){
                    que.push(each);
                }
            }
            level_order.emplace_back(std::move(level));
        }
        return level_order;
    }
};
```

116 填充每一个节点的下一个右侧节点指针

给定一个完美二叉树，其所有叶子节点都在同一层，每个父节点都有两个子节点。二叉树定义如下：

```
struct Node { int val; Node left; Node right; Node *next; }
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。

```
class Solution {
 public:
  Node* connect(Node* root) {
    if (root == nullptr) return root;
    queue<Node*> Q;
    Q.push(root);
    while (!Q.empty()) {
      int size = Q.size();
      for (int i = 0; i < size; i++) {
        auto node = Q.front();
        Q.pop();
        if (i < size - 1) {  // 连接
          node->next = Q.front();
        }
        if (node->left) Q.push(node->left);
        if (node->right) Q.push(node->right);
      }
    }
    return root;
  }
};
```

117 填充下一个右侧节点 2

给定一个二叉树：

`struct Node { int val; Node left; Node right; Node *next; }`

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL 。

初始状态下，所有 next 指针都被设置为 NULL 。

```
class Solution {
 public:
  Node* connect(Node* root) {
    if (root == nullptr) return root;
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
      int n = q.size();
      Node* last = nullptr;
      for (int i = 1; i <= n; i++) {
        auto f = q.front();
        q.pop();
        if (f->left) q.push(f->left);
        if (f->right) q.push(f->right);
        if (i != 1) last->next = f;
        last = f;
      }
    }
    return root;
  }
};
```

对称二叉树

```
class Solution {
    bool checker(TreeNode* left,TreeNode *right){
        if(!left&&!right) return true;
```

```
        if(left&&right)
        return left->val==right->val&&checker(left->left,right->right)&&checker(left->right, right->left);
        return false;
    }
public:
    bool isSymmetric(TreeNode* root) {
        return checker(root,root);
    }
};
```

272 最接近的二叉搜索树值 2

给定二叉搜索树的根 `root` 、一个目标值 `target` 和一个整数 `k` ，返回 BST 中最接近目标的 `k` 个值。你可以按任意顺序返回答案。题目保证该二叉搜索树中只会存在一种 `k` 个值集合最接近 `target`

思路：由于二叉树中序便利结果有序，就应该先获得中序便利的序列再用双指针法获得结果。

```
class Solution {
public:
    vector<int> closestKValues(TreeNode* root, double target, int k) {
        vector<int> traverse;
        stack<TreeNode*> nodes;
        int closestIndex=-1;
        if(root!=nullptr) nodes.push(root);
        while(!nodes.empty()){
            auto node=nodes.top();
            if(node!=nullptr){
                nodes.pop();
                if(node->right) nodes.push(node->right);//添加右节点
                nodes.push(node);//添加中节点
                nodes.push(nullptr);//中节点访问过但是没有处理
                if(node->left) nodes.push(node->left);
            }else{
                nodes.pop();//弹出空节点
                node=nodes.top();//重新取出栈中元素
                nodes.pop();
                if(node->val<=target){
                    closestIndex=traverse.size();
                }
                traverse.emplace_back(node->val);
            }
        }
        vector<int> result;
        int n=traverse.size();
        int index1=closestIndex,index2=closestIndex+1;
        while(k>0&&index1>=0&&index2<n){
            if(target-traverse.at(index1)<traverse.at(index2)-target){
                result.push_back(traverse.at(index1));
                index1--;
            }else{
                result.push_back(traverse.at(index2));
                index2++;
            }
            k--;
        }
        while(k>0&&index1>=0){
            result.push_back(traverse.at(index1--));
            k--;
        }
        while(k>0&&index2<n){
            result.push_back(traverse.at(index2++));
            k--;
```

```
        }
        return result;
    }
};
```

298 二叉树最长连续序列

给你一棵指定的二叉树的根节点 `root` ，请你计算其中最长连续序列路径的长度。

最长连续序列路径是依次递增 `1` 的路径。该路径，可以是从某个初始节点到树中任意节点，通过「父 - 子」关系连接而产生的任意路径。且必须从父节点到子节点，反过来是不可以的。

```
class Solution {
public:
    int dfs(TreeNode* node,TreeNode* Parent,int length){
        if(node==nullptr) return length;
        length=(Parent!=nullptr&&node->val==(Parent->val+1))?length+1: 1;
        return max(length,max(dfs(node->left,node,length),dfs(node->right,node,length)));
    }
    int longestConsecutive(TreeNode* root) {
        return dfs(root,nullptr,0);
    }
};
```

404 左叶子之和

```
class Solution {
public:
    int sumOfLeftLeaves(TreeNode* root) {
        if(root==nullptr) return 0;
        int midValue=0;
        if(root->left&&root->left->left==nullptr&&root->left->right==nullptr){
            midValue=root->left->val;
        }
        return midValue+sumOfLeftLeaves(root->left)+sumOfLeftLeaves(root->right);
    }
};
```

513 找树左下角的值

给定一个二叉树的根节点 `root`，请找出该二叉树的最底层最左边节点的值。

假设二叉树中至少有一个节点。

```
class Solution {
public:
    int findBottomLeftValue(TreeNode* root) {
        TreeNode *last;
        queue<TreeNode*> queue;
        if(root) queue.push(root);
        while(queue.size()){
            last=queue.front();
            queue.pop();
            if(last->right) queue.push(last->right);
            if(last->left) queue.push(last->left);
        }
        return last->val;
    }
};
```

112 路径总和

给你二叉树的根节点 `root` 和一个表示目标和的整数 `targetSum` 。判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和 `targetSum` 。如果存在，返回 `true` ；否则，返回 `false` 。

叶子节点是指没有子节点的节点。

```
class Solution {

    bool traversal(TreeNode* cur,int count){
        if(!cur->left&&!cur->right) return count==0;
        if(cur->left){
            count-=cur->left->val;
            if(traversal(cur->left, count)) return true;
            count+=cur->left->val;
        }
        if(cur->right){
            count-=cur->right->val;
            if(traversal(cur->right, count)) return true;
            count+=cur->right->val;
        }
        return false;
    }
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if(root==nullptr) return false;
        return traversal(root,targetSum-root->val);
    }
};
```

113 路径总和

给你二叉树的根节点 `root` 和一个整数目标和 `targetSum` ，找出所有从根节点到叶子节点路径总和等于给定目标和的路径。

叶子节点是指没有子节点的节点。

```
class Solution {
 public:
  vector<vector<int>> ret;
  vector<int> path;
  void dfs(TreeNode *root, int targetSum) {
    if (root == nullptr) {
      return;
    }
    path.emplace_back(root->val);
    targetSum -= root->val;
    if (root->left == nullptr && root->right == nullptr && targetSum == 0) {
      ret.push_back(path);
    }
    dfs(root->left, targetSum);
    dfs(root->right, targetSum);
    path.pop_back();
  }
  vector<vector<int>> pathSum(TreeNode *root, int targetSum) {
    dfs(root, targetSum);
    return ret;
  }
};
```

106 从中序与后序遍历序列构造二叉树

给定两个整数数组 `inorder` 和 `postorder` ，其中 `inorder` 是二叉树的中序遍历，`postorder` 是同一棵树的后序遍历，请你构造并返回这颗二叉树。

```
class Solution {
public:
    map<int, int> midorder_index_map;
    int post_index = 0;
    TreeNode* helper(int left, int right, vector<int>& mid, vector<int>& post) {
        if (left > right)
            return nullptr;
        int root_val = post[post_index];
        auto root = new TreeNode(root_val);
        int root_index = midorder_index_map[root_val];
        post_index--;
        root->right = helper(root_index + 1, right, mid, post);
        root->left = helper(left, root_index - 1, mid, post);

        return root;
    }
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        post_index = postorder.size() - 1;
        int idx = 0;
        for (auto& k : inorder) {
            midorder_index_map[k] = idx++;
        }
        return helper(0, inorder.size() - 1, inorder, postorder);
    }
};
```

105 从中序和前序遍历建树

```
class Solution {
public:
vector<int> pre,mid;
map<int,int> memo;
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        pre=std::move(preorder);
        mid=std::move(inorder);
        for(int i=0;i<mid.size();i++){
            memo[mid[i]]=i;
        }
        return build_tree_help(0,pre.size()-1,0,mid.size()-1);
    }
    TreeNode* build_tree_help(int pre_start,int pre_end,int mid_start,int mid_end){
        if(pre_start>pre_end||mid_start>mid_end){
            return nullptr;
        }
        int root_val=pre[pre_start];
        int root_pos=memo[root_val];
        TreeNode* root=new TreeNode(root_val);
        root->left=build_tree_help(pre_start+1,pre_start+root_pos-mid_start,mid_start,root_pos-1);
        root->right=build_tree_help(pre_start+root_pos-mid_start+1,pre_end,root_pos+1,mid_end);
        return root;
    }
};
```

654 最大二叉树

给定一个不重复的整数数组 `nums` 。最大二叉树可以用下面的算法从 `nums` 递归地构建：

创建一个根节点，其值为 `nums` 中的最大值。

递归地在最大值左边的子数组前缀上构建左子树。

递归地在最大值右边的子数组后缀上构建右子树。

返回 `nums` 构建的最大二叉树。

```
class Solution {
    unordered_map<int,int> index;
public:
    TreeNode* builder(vector<int>& nums, int start, int end) {
        if(start>end) return nullptr;
        auto max_num=*std::max_element(nums.begin()+start,nums.begin()+end+1);
        auto max_index=index[max_num];
        auto node=new TreeNode(max_num);
        node->left=builder(nums,start,max_index-1);
        node->right=builder(nums,max_index+1,end);
        return node;
    }
    TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
        for(int i=0;i<nums.size();i++){
            index[nums[i]]=i;
        }
        return builder(nums,0,nums.size()-1);
    }
};
```

236 二叉树的最近公共祖先

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

701 二叉搜索树插入操作

给定二叉搜索树（BST）的根节点 `root` 和要插入树中的值 `value` ，将值插入二叉搜索树。返回插入后二叉搜索树的根节点。输入数据保证，新值和原始二叉搜索树中的任意节点值都不同。

注意，可能存在多种有效的插入方式，只要树在插入后仍保持为二叉搜索树即可。你可以返回任意有效的结果。

```
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
       if(root==nullptr){
        auto node=new TreeNode(val);
        return node;
       }
       if(root->val>val) root->left=insertIntoBST(root->left,val);
       if(root->val<val) root->right=insertIntoBST(root->right,val);
       return root;
```

```
    }
};
```

450 删除二叉搜索树中的节点

给定一个二叉搜索树的根节点 `root` 和一个值 `key`，删除二叉搜索树中的 `key` 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

首先找到需要删除的节点；
如果找到了，删除它。

```
class Solution{
public:
    TreeNode* deleteNode(TreeNode* root, int key) {
        if(root==nullptr) return nullptr;
        if(key<root->val) root->left=deleteNode(root->left,key);
        else if(key>root->val) root->right=deleteNode(root->right,key);
        else{
            if(!root->left)  return root->right;
            if(!root->right) return root->left;
            TreeNode *node=root->right;
            while(node->left) node=node->left;
            node->left=root->left;
            root=root->right;
        }
        return root;
    }
};
```

669 修建二叉搜索树

给你二叉搜索树的根节点 `root` ，同时给定最小边界 `low` 和最大边界 `high`。通过修剪二叉搜索树，使得所有节点的值在 `[low, high]` 中。修剪树不应该改变保留在树中的元素的相对结构 (即，如果没有被移除，原有的父代子代关系都应当保留)。可以证明，存在唯一的答案。

所以结果应当返回修剪好的二叉搜索树的新的根节点。注意，根节点可能会根据给定的边界发生改变。

```
class Solution {
public:
    TreeNode* trimBST(TreeNode* root, int low, int high) {
        if(root==nullptr) return root;
        if(root->val<low){
            return trimBST(root->right,low,high);
        }else if(root->val>high){
            return trimBST(root->left,low,high);
        }
        root->left=trimBST(root->left, low, high);
        root->right=trimBST(root->right, low, high);
        return root;
    }
};
```

538 把二叉搜索树转换为累加树

给出二叉搜索树的根节点，该树的节点值各不相同，请你将其转换为累加树（*Greater Sum Tree*），使每个节点 `node` 的新值等于原树中大于或等于 `node.val` 的值之和。

提醒一下，二叉搜索树满足下列约束条件：

节点的左子树仅包含键小于节点键的节点。

节点的右子树仅包含键大于节点键的节点。

左右子树也必须是二叉搜索树。

需要注意到中序便利是有序的，所有本题是反向中序便利

```
class Solution {
    int pre=0;
    void traversal(TreeNode *cur){
        if(cur==nullptr) return ;
        traversal(cur->right);
        cur->val+=pre;
        pre=cur->val;
        traversal(cur->left);
    }
public:
    TreeNode* convertBST(TreeNode* root) {
        traversal(root);
        return root;
    }
};
```