This repository Search | Pull requests Issues Gist

huyilong / **OOP-Design**

Unwatch 1 | Unstar 1 | Fork 0

Code | Issues 0 | Pull requests 0 | Wiki | Pulse | Graphs | Settings

Branch: master | **OOP-Design / ******TreeSummaryII******.java** | Find file | Copy path

huyilong SUMMARY - ALLLL **TreeSummaryII - New Problems** Problems — 43332e0 a minute ago

1 contributor

707 lines (613 sloc) | 23 KB — Raw | Blame | History

```
public class Solution {
    public TreeNode invertTree(TreeNode root) {
        if(root!=null){
            helper(root);

        }
        return root;
    }

    private void helper(TreeNode root){
        if(root == null){
            return;
        }

        TreeNode temp = root.left;
        root.left = root.right;
        root.right = temp;

        if(root.left!=null){
            helper(root.left);
        }

        if(root.right!=null){
            helper(root.right);
        }
    }

}

Root To Leaf Binary Tree Paths
时间 O(b^(h+1)-1) 空间 O(h) 递归栈空间 对于二叉树b=2
public class Solution {

    List<String> res = new ArrayList<String>();

    public List<String> binaryTreePaths(TreeNode root) {
        if(root != null)
            findPaths(root,String.valueOf(root.val));
        return res;
    }

    private void findPaths(TreeNode n, String path){
        if(n.left == null && n.right == null)
            res.add(path);
        if(n.left != null)
            findPaths(n.left, path+"->"+n.left.val);
        if(n.right != null)
            findPaths(n.right, path+"->"+n.right.val);
    }
}
-------------------all path recorded-----------------------------
public class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
    if(root==null)
        return res;
    ArrayList<Integer> item = new ArrayList<Integer>();
    item.add(root.val);
    helper(root,sum-root.val,item,res);
    return res;
}
private void helper(TreeNode root, int sum, List<Integer> item, List<List<Integer>> res)
{
    if(root == null)
        return;
    //make sure it is leaf node and the target is reached !!!!!
    if(  (root.left==null && root.right==null)   && sum==0)//root.val == sum )//sum==0)
    {
        res.add(new ArrayList<Integer>(item));
        return;
    }
    if(root.left!=null)
    {
        item.add(root.left.val);
        helper(root.left,sum-root.left.val,item,res);
        item.remove(item.size()-1);
    }
    if(root.right!=null)
    {
        item.add(root.right.val);
        helper(root.right,sum-root.right.val,item,res);
        item.remove(item.size()-1);
    }
}
}
---------------------------easy one------------------------
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root == null){
            return false;
        }

        if(root.val == sum && root.left == null && root.right == null){
            return true;
        }

        return hasPathSum(root.left, sum-root.val) || hasPathSum(root.right, sum-root.val);
    }
}


Node to Node Binary Tree Path
给定一棵二叉树的根节点和两个任意节点，返回这两个节点之间的最短路径
复杂度
时间 O(h) 空间 O(h) 递归栈空间
思路
两个节点之间的最短路径一定会经过两个节点的最小公共祖先，所以我们可以用LCA的解法。
不同于LCA的是，我们返回不只是标记，而要返回从目标结点递归回当前节点的路径。
当遇到最小公共祖先的时候便合并路径。需要注意的是，我们要单独处理目标节点自身是最小公共祖先的情况。

public LinkedList<TreeNode> helper(TreeNode n, TreeNode p, TreeNode q){
    if(n == null){
        return null;
    }

    LinkedList<TreeNode> left = helper(n.left, p, q);
    LinkedList<TreeNode> right = helper(n.right, p, q);

    // 当左右都为空时
    if(left == null && right == null){
        // 如果当前节点是目标节点，开启一条新路径
        if(n == p || n == q){
            LinkedList l = new LinkedList<TreeNode>();
            l.add(n);
            return l;
        } else {
        // 否则标记为空
            return null;
        }
    // 如果左右节点都不为空，说明是最小公共祖先节点，合并两条路径
    } else if(left != null && right != null){
        finalPath.addAll(left);
        finalPath.add(n);
        Collections.reverse(right);
        finalPath.addAll(right);
        return left;
    // 如果当前节点是目标结点，且某一个子树不为空时，说明最小公共祖先是节点自身
    } else if (left != null){
        left.add(n);
        if(n == p || n == q){
            finalPath.addAll(left);
        }
        return left;
    } else {
        right.add(n);
        if(n == p || n == q){
            finalPath.addAll(right);
        }
        return right;
    }
}
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root == null){
            return null;
        }
        if(root == p || root == q){
            return root;
        }
        TreeNode l = lowestCommonAncestor(root.left, p,q);
        TreeNode r = lowestCommonAncestor(root.right, p,q);

        if(l!=null && r!=null){
            //the nodes were found on the two sides of the root
            return root;
        }
        return r != null ? r:l;
    }
}

Closest Binary Search Tree Value
Given a non-empty binary search tree and a target value, find the value in the
BST that is closest to the target.
Note:
Given target value is a floating point.
You are guaranteed to have only one unique value in the BST that is closest to the target.
[思路]
closest必然在查找路径上.
public class Solution {
    public int closestValue(TreeNode root, double target) {

        int closest = root.val;
        double min = Double.MAX_VALUE;

        while(root!=null) {
            if( Math.abs(root.val - target) < min  ) {
                min = Math.abs(root.val - target);
                closest = root.val;
            }

            if(target < root.val) {
                root = root.left;
            } else if(target > root.val) {
                root = root.right;
            } else {
                return root.val;
            }
        }

        return closest;
    }
}

Closest Binary Search Tree Value II
Total Accepted: 984 Total Submissions: 3704 Difficulty: Hard
Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.
Note:
Given target value is a floating point.
You may assume k is always valid, that is: k ≤ total nodes.
You are guaranteed to have only one unique set of k values in the BST that are closest to the target.
Follow up:
Assume that the BST is balanced, could you solve it in less than O(n) runtime (where n = total nodes)?
Hint:
Consider implement these two helper functions:
getPredecessor(N), which returns the next smaller node to N.
getSuccessor(N), which returns the next larger node to N.
Try to assume that each node has a parent pointer, it makes the problem much easier.
Without parent pointer we just need to keep track of the path from the root to the current node using a stack.
You would need two stacks to track the path in finding predecessor and successor node separately.
[思路]
prefix traverse. 同时维护一个大小为k的 max heap. 注意根据bst的性质,在diff 大于 maxHeap时, 可以只遍历一边的子树.

```

```
public class Solution {

    public List<Integer> closestKValues(TreeNode root, double target, int k) {
        PriorityQueue<Double> maxHeap = new PriorityQueue<Double>(k, new Comparator<Double>() {
            @Override
            public int compare(Double x, Double y) {
                return (int)(y-x);
            }
        });
        Set<Integer> set = new HashSet<Integer>();

        rec(root, target, k, maxHeap, set);

        return new ArrayList<Integer>(set);
    }

    private void rec(TreeNode root, double target, int k, PriorityQueue<Double> maxHeap, Set<Integer> set) {
        if(root==null) return;
        double diff = Math.abs(root.val-target);
        if(maxHeap.size()<k) {
            maxHeap.offer(diff);
            set.add(root.val);
        } else if( diff < maxHeap.peek() ) {
            double x = maxHeap.poll();
            if(! set.remove((int)(target+x))) set.remove((int)(target-x));
            maxHeap.offer(diff);
            set.add(root.val);
        } else {
            if(root.val > target) rec(root.left, target, k, maxHeap,set);
            else rec(root.right, target, k, maxHeap, set);
            return;
        }
        rec(root.left, target, k, maxHeap, set);
        rec(root.right, target, k, maxHeap, set);
    }
}


Binary Tree Longest Consecutive Sequence
Given a binary tree, find the length of the longest consecutive sequence path.
The path refers to any sequence of nodes from some starting node to any node
in the tree along the parent-child connections. The longest consecutive path
need to be from parent to child (cannot be the reverse).
For example,
   1
    \
     3
    / \
   2   4
        \
         5
Longest consecutive sequence path is 3-4-5, so return 3.
   2
    \
     3
    /
   2
  /
 1
Longest consecutive sequence path is 2-3,not3-2-1, so return 2.

public class Solution {
    int max = 1;

    public int longestConsecutive(TreeNode root) {
        if(root==null) return 0;
        helper(root, 1);
        return max;
    }

    private void helper(TreeNode n, int c) {
        if(n.left!=null) {
            if(n.val+1 == n.left.val) {
                helper(n.left, c+1);
                max = Math.max(max, c+1);
            }else{
                helper(n.left, 1);
            }
        }

        if(n.right!=null) {

            if(n.val+1 == n.right.val) {
                helper(n.right, c+1);
                max = Math.max(max, c+1);
            }else{
                helper(n.right, 1);
            }
        }
    }
}

Binary Tree Maximum Path Sum
Given a binary tree, find the maximum path sum.
The path may start and end at any node in the tree.
For example:
Given the below binary tree,
       1
      / \
     2   3
Return 6.
key-points: globe variable record the max value of local branch.
at the end, in root node compare max value cross root node with maxmum local
branch which may not cross root node.

public class Solution {
    int globe = Integer.MIN_VALUE;
// null, {1}, {-1}, {0} , {1,-2,-3}, {-1,#,2,-3,0} {1,#,2,3,#,4,5,6}
    public int maxPathSum(TreeNode root) {
        // Start typing your Java solution below
        // DO NOT write main() function

        //input check
        globe = Integer.MIN_VALUE;

        int passRoot = maxRec(root);

        return globe>passRoot ? globe : passRoot;  //Math.max(globe, passRoot) instead.
    }

    private int maxRec(TreeNode root){
        if(root==null) return 0;

        int l = maxRec(root.left);
        int r = maxRec(root.right);

        int local = root.val;
        if(l>0) local += l;
        if(r>0) local += r;

        globe = globe>local ? globe : local;

        return Math.max( root.val, Math.max( root.val+l, root.val+r) );
    }
}

Given a binary tree, count the number of uni-value subtrees.

A Uni-value subtree means all nodes of the subtree have the same value.

For example:
Given binary tree,
              5
             / \
            1   5
           / \   \
          5   5   5
return 4.

public class Solution {
    public int countUnivalSubtrees(TreeNode root) {
        univalٍ(root);
        return count;
    }

    private boolean unival(TreeNode root) {
        if(root == null)
            return true;
        if(root.left ==null && root.right == null) {
            count++;
            return true;
        }
        boolean left = unival(root.left);
        boolean right = unival(root.right);
        if(left && right && (root.left == null ||
                    root.left.val == root.val) && (root.right == null ||
                        root.right.val == root.val)) {
            count++;
            return true;
        }
        return false;
    }

    private int count = 0;
}


 Graph Valid Tree
 Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether the

For example:

Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], returntrue.

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], returnfalse.

Hint:

Given n = 5 andedges = [[0, 1], [1, 2], [3, 4]], what should your return? Is this case a valid tree?
According to the definition of tree on Wikipedia: “a tree is an undirected graph in which any two vertices are connected byexactly one path
Note: you can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus wi

public class Solution {
    public boolean validTree(int n, int[][] edges) {
        int[] root = new int[n];
        for(int i = 0; i < n; i++)
            root[i] = i;
        for(int i = 0; i < edges.length; i++) {
            int root1 = find(root, edges[i][0]);
            int root2 = find(root, edges[i][1]);
            if(root1 == root2)
                return false;
            root[root2] = root1;
        }
        return edges.length == n - 1;
    }

    private int find(int[] root, int e) {
        if(root[e] == e)
            return e;
        else
            return find(root, root[e]);
    }
}

Given an array of numbers, verify whether it is the correct preorder
traversal sequence of a binary search tree.
You may assume each number in the sequence is unique.

Follow up:
Could you do it using only constant space complexity?

先复习一下BST，给定一个节点，其左子树的所有节点都小于该节点，右子树的所有节点都大于该节点；
preorder序列是指在遍历该BST的时候，先记录根节点，再遍历左子树，然后遍历右子树；
所以一个preorder序列有这样一个特点，左子树的序列必定都在右子树的序列之前；
并且左子树的序列必定都小于根节点，右子树的序列都大于根节点；

根据上面的特点很容易通过递归的方式完成：

如果序列只有一个元素，那么肯定是正确的，对应只有一个节点的树；

如果多于一个元素，以当前节点为根节点；并从当前节点向后遍历，直到大于根节点的节点出现（或者到尾巴），
那么根节点之后，该大节点之前的，是左子树；该大节点及之后的组成右子树；递归判断左右子树即可；

那么什么时候一个序列肯定不是一个preorder序列呢？前面得到的右子树，如果在其中出现了比根节点还小的数，
么就可以直接返回false了；

public boolean verifyPreorder(int[] preorder) {
    return verifyPreorder(preorder, 0, preorder.length);
}

```

```
public boolean verifyPreorder(int[] seq, int start, int end) {
    if (start + 1 >= end) {
        return true;
    }

    int root = seq[start];

    int i = start + 1;
    while (i < end && seq[i] < root) {
        i++;
    }

    if (i < end) {
        int j = i;
        while (j < end && seq[j] > root) {
            j++;
        }
        if (j < end) {
            return false;
        }

        return verifyPreorder(seq, start + 1, i) && verifyPreorder(seq, i, end);
    } else {
        return verifyPreorder(seq, start + 1, end);
    }
}

Kth Smallest Element in a BST
Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.

Note:
You may assume k is always valid, 1 ≤ k ≤ BSTs total elements.

Follow up:
What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently?
How would you optimize the kthSmallest routine?
public class Solution {
    public int kthSmallest(TreeNode root, int k) {
         //if it is a binary search tree then the left child is less than the middle one and then less than the right one
        //this is same with the bst iterator
        Stack<TreeNode> s = new Stack<>();
        //we never want to change the position of tree root
        //because it is similar to the head and end of the linkedlist if we change if
        //we lost the whole information of the tree
        TreeNode p= root;
        //we need to push the root into the stack to drive the following while(!s.isEmpty())
        s.push(p);
        int res = 0;
        while(!s.isEmpty()){
            if(p != null){
                s.push(p);
                p=p.left;
            }else{
                TreeNode t = s.pop();
                //at least here we could decrease the number of min we found
                //by each time we pop up the node
                k--;
                if(k==0){
                    res = t.val;
                   // return t.val;
                }
                //here once we pop up a node we need to push the left subtree into
                //the stack --- > iterator
                p = t.right;

            }
        }
        return res;
    }
}

Second Largest Element in an Array
 static int secondHighest(int... nums) {
    int high1 = Integer.MIN_VALUE;
    int high2 = Integer.MIN_VALUE;
    for (int num : nums) {
      if (num > high1) {
        high2 = high1;
        high1 = num;
      } else if (num > high2) {
        high2 = num;
      }
    }
    return high2;
 }

Kth Largest Element in an Array - similar to o(n) //which is for quik selecting
public class Solution {
    public int findKthLargest(int[] nums, int k) {
1. Pick an element within current segment
   and call it the pivot

2. Count elements that are smaller and
   elements that are larger than the pivot

3. If number of elements smaller than the pivot
   is larger than K, then move those elements
   to the beginning of the array and run
   the algorithm recursively only on that part of the array.  -- our objects are limited to this range

4. Otherwise, if number of elements smaller than the pivot
   plus number of elements       equal to the pivot is larger
   than K, then Kth element is equal to pivot
   so just return the pivot and finish.

5. Otherwise, move all elements larger than the pivot
   to the beginning of the array and run the algorithm
   recursively only on that part of the array.
//here to simplify we just select the last element in the array to be the pivot
            if(k<1 || nums == null){
                return 0;
            }
            return getKth(nums.length-k+1, nums, 0, nums.length-1);
    }
    public int getKth(int k, int[] nums, int l, int h){
        int pivot = nums[h];//let the pivot be the last element in the array
        int left = l;//l and h are head and end we cannot move them
        int right = h;
        while(left <=right){
            while(nums[left] <pivot ){
                left++;
            }
            while(nums[right]>pivot){
                right--;
            }

            //here we
            if(left < right){
                int temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;
            }
        }
        int temp = nums[h];//we need to put the pivot in place -- in the current middle which is left
        nums[h] = nums[left];//left is left in the while
        nums[left] = temp;//we place the pivot in the right place

        if(k == left + 1){
            //here we find the kth largest
            return pivot;
        }else if(k<left+1){
            //the result is existing in the left side of the array
            return getKth(k, nums, l, left-1);
        }else{
            //the result is in the right side of the array
            return getKth(k, nums, left+1, h);
        }
    }
}

Find the Celebrity
Total Accepted: 1126 Total Submissions: 3603 Difficulty: Medium
Suppose you are at a party with n people (labeled from 0 ton - 1) and among them,
here may exist one celebrity. The definition of a celebrity is that all the othern - 1 people know him/her but he/she does not know any of
Now you want to find out who the celebrity is or verify that there is not one.
The only thing you are allowed to do is to ask questions like: Hi, A. Do you know B? to get
information of whether A knows B. You need to find out the celebrity (or verify there is not one)
by asking as few questions as possible (in the asymptotic sense).
You are given a helper function bool knows(a, b) which tells you whether A knows B.
Implement a functionint findCelebrity(n), your function should minimize the number of calls toknows.
```

```
Note: There will be exactly one celebrity if he/she is in the party.
Return the celebritys label if there is a celebrity in the party. If there is no celebrity, return-1.
[思路]
当 a -> b时，可以推出， a不可能是celebrity，b被人知道的数目+1... 用bitmap记录.
[CODE]
/* The knows API is defined in the parent class Relation.
      boolean knows(int a, int b); */
public class Solution extends Relation {
    public int findCelebrity(int n) {
        int[] bitmap = new int[n];
        for(int i=0; i<n; i++) {
            for(int j=0; j<n; j++) {
                if(i==j) continue;

                if(bitmap[j]>=0) {
                    if( knows(i, j) ) {
                        bitmap[i] = -1;
                        bitmap[j]++;
                    } else {
                        bitmap[j] = -1;
                    }
                }
            }
        }
        for(int i=0; i<n; i++) {
            if(bitmap[i] == n-1) {
                for(int j=0; j<n; j++) {
                    if(i==j) continue;
                    if(knows(i, j)) return -1;
                }
                return i;
            }
        }

        return -1;
    }
}
Zigzag Iterator
Given two 1d vectors, implement an iterator to return their elements alternately.
For example, given two 1d vectors:
v1 = [1, 2]
v2 = [3, 4, 5, 6]
By calling next repeatedly until hasNext returns false, the order of elements
returned by next should be: [1, 3, 2, 4, 5, 6].
Follow up: What if you are given k 1d vectors? How well can your code be extended
o such cases?
Clarification for the follow up question - Update (2015-09-18):
The "Zigzag" order is not clearly defined and is ambiguous for k > 2 cases.
If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic".
For example, given the following input:
[1,2,3]
[4,5,6,7]
[8,9]
It should return [1,4,8,2,5,9,3,6,7].
[思路]
iterator都放到一个list里，用一个count循环，
public class ZigzagIterator {
    List<Iterator<Integer> > iters = new ArrayList<Iterator<Integer> >();

    int count = 0;

    public ZigzagIterator(List<Integer> v1, List<Integer> v2) {
        if( !v1.isEmpty() ) iters.add(v1.iterator());
        if( !v2.isEmpty() ) iters.add(v2.iterator());
    }

    public int next() {
        int x = iters.get(count).next();
        if(!iters.get(count).hasNext()) iters.remove(count);
        else count++;

        if(iters.size()!=0) count %= iters.size();
        return x;
    }

    public boolean hasNext() {
        return !iters.isEmpty();
    }
}
```