TalkGo算法之美第三期

第二周线上讨论会

主持人：小林

第二周--周一题目--路径和计算

129. 求根到叶子节点数字之和

给定一个二叉树，它的每个结点都存放一个 `0-9` 的数字，每条从根到叶子节点的路径都代表一个数字。

例如，从根到叶子节点路径 `1->2->3` 代表数字 `123`。

计算从根到叶子节点生成的所有数字之和。

说明: 叶子节点是指没有子节点的节点。

示例 1:

```
输入: [1,2,3]
    1
   / \
  2   3
输出: 25
解释:
从根到叶子节点路径 1->2 代表数字 12.
从根到叶子节点路径 1->3 代表数字 13.
因此，数字总和 = 12 + 13 = 25.
```

第二周--周一题目--路径和计算

129. 求根到叶子节点数字之和

思路：深度优先搜索或者广度优先搜索。定义一个num参数，用于保存遍历过程中产生的结果。公式为：num(本节点)=num(父节点)*10+本节点的Val值。

```go
func sumNumbers(root *TreeNode) int {
    if root == nil {                        // 根为nil，返回0
        return 0
    }
    res := 0
    var helper func(num int, node *TreeNode)
    helper = func(num int, node *TreeNode) {
        num = num*10 + node.Val
        // 叶子节点，把num加到最终的结果中去。
        if node.Left == nil && node.Right == nil {
            res += num
            return
        }
        if node.Left != nil {
            helper(num, node.Left)
        }
        if node.Right != nil {
            helper(num, node.Right)
        }
    }
    helper(0, root)
    return res
}
```

第二周--周一题目--路径和计算

124. 二叉树中的最大路径和

给定一个**非空**二叉树，返回其最大路径和。

本题中，路径被定义为一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列。该路径**至少包含一个**节点，且不一定经过根节点。

示例 1:

```
输入: [1,2,3]

       1
      / \
     2   3

输出: 6
```

示例 2:

```
输入: [-10,9,20,null,null,15,7]

   -10
   / \
  9  20
    /  \
   15   7

输出: 42
```

第二周--周一题目--路径和计算

124. 二叉树中的最大路径和

思路：递归。获得左子树和右子树的最大单边路径和。然后将左右子树的单边路径和与当前节点的值加起来，并判断是否大于目前最大的路径和。

```go
func maxPathSum(root *TreeNode) int {
    if root == nil {
        return 0
    }
    res := math.MinInt64
    var helper func(node *TreeNode) int
    helper = func(node *TreeNode) int {
        if node == nil {
            return 0
        }
        leftMax := helper(node.Left)
        rightMax := helper(node.Right)
        res = max(res, node.Val + leftMax + rightMax)
        // 如果返回值小于0，会导致加起来的最大路径变小，故取0
        return max(max(node.Val+leftMax, node.Val+rightMax), 0)
    }
    helper(root)
    return res
}

func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}
```

第二周--周一题目--路径和计算

437. 路径总和 III

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过1000个节点，且节点数值范围是 [-1000000,1000000] 的整数。

示例：

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

返回 3。和等于 8 的路径有:

1.  5 -> 3
2.  5 -> 2 -> 1
3.  -3 -> 11
```

第二周--周一题目--路径和计算

437. 路径总和 III

思路：使用两个递归。一个计算包含当前根节点，并向下递归。一个调用上面递归前提下，求不包含当前根节点，只看左右子树有多少中结果。

```
func pathSum(root *TreeNode, sum int) int {
    if root == nil {
        return 0
    }
    // 对左节点和右节点进行pathSum递归，对当前节点使用calPath递归
    return pathSum(root.Left, sum) + pathSum(root.Right, sum) + calPath(root, sum)
}
// 辅助函数，用于计算从当前根节点出发路径和等于给定值的路径数量
func calPath(node *TreeNode, sum int) (n int) {
    if node == nil {
        return
    }
    //
    sum -= node.Val
    if sum == 0 {
        n = 1
    }
    n = n + calPath(node.Left, sum) + calPath(node.Right, sum)
    return
}
```

第二周--周三题目--二叉搜索树

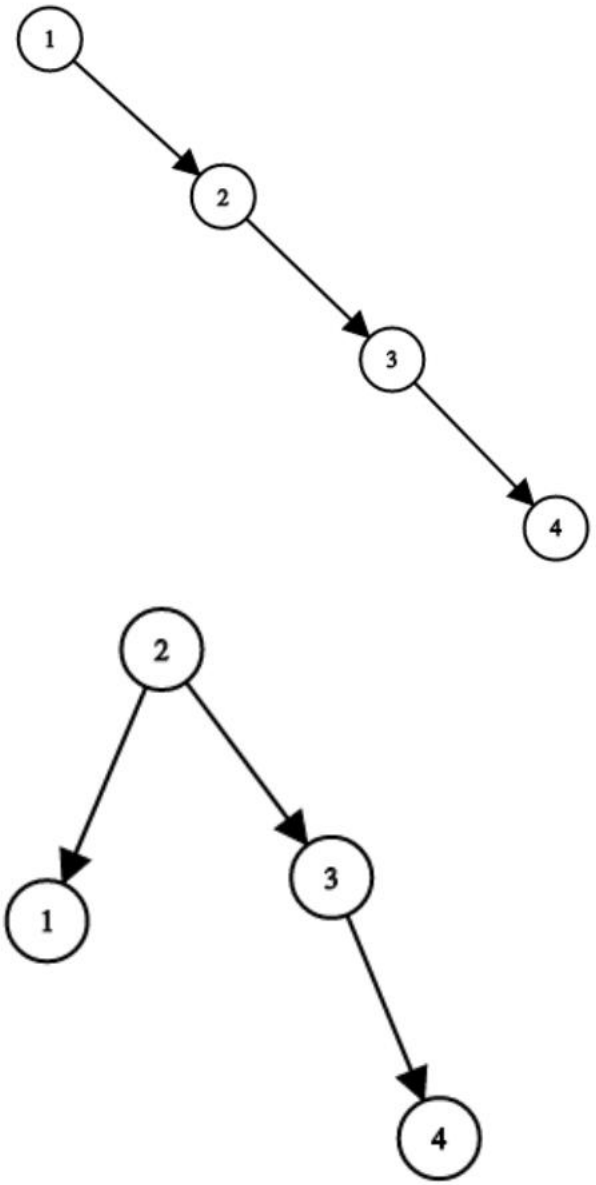
1382. 将二叉搜索树变平衡

给你一棵二叉搜索树，请你返回一棵 **平衡后** 的二叉搜索树，新生成的树应该与原来的树有着相同的节点值。

如果一棵二叉搜索树中，每个节点的两棵子树高度差不超过 1 ，我们就称这棵二叉搜索树是 **平衡的** 。

如果有多种构造方法，请你返回任意一种。

示例：



第二周--周三题目--二叉搜索树

1382. 将二叉搜索树变平衡

思路：中序遍历一遍生成升序数组，然后利用数组构造一棵新的二叉树

```go
func balanceBST(root *TreeNode) *TreeNode {
    arr := []int{}
    inOrder(root, &arr)
    return construct(arr)
}

// 使用中序变量得到按升序排列的切片
func inOrder(root *TreeNode, arr *[]int) {
    if root == nil {
        return
    }
    inOrder(root.Left, arr)
    *arr = append(*arr, root.Val)
    inOrder(root.Right, arr)
}

// 参考第一周周一的108题，通过切片来构建二叉树
func construct(arr []int) *TreeNode {
    if len(arr) == 0 {
        return nil
    }
    mid := len(arr)/2
    node := &TreeNode{arr[mid], nil, nil}
    node.Left = construct(arr[:mid])
    node.Right = construct(arr[mid+1:])
    return node
}
```

第二周--周三题目--二叉搜索树

面试题 04.05. 合法二叉搜索树

实现一个函数，检查一棵二叉树是否为二叉搜索树。

示例 1:

```
输入:
    2
   / \
  1   3
输出: true
```

示例 2:

```
输入:
    5
   / \
  1   4
     / \
    3   6
输出: false
解释: 输入为: [5,1,4,null,null,3,6]。
     根节点的值为 5 ，但是其右子节点值为 4 。
```

第二周--周三题目--二叉搜索树

面试题 04.05. 合法二叉搜索树

思路：使用中序遍历的非递归形式，定义一个pre变量，用以保存遍历的前一个结点的值，遍历过程中，只需比较当前结点和前一个结点的值，如果出现前一个结点大于或等于当前结点，那么该树就不是合法的搜索二叉树。

```go
func isValidBST(root *TreeNode) bool {
    if root == nil {
        return true
    }
    pre, stack := math.MinInt64, []*TreeNode{}
    ptr := root
    // 中序遍历非递归形式
    for ptr != nil || len(stack) > 0 {
        if ptr != nil {
            stack = append(stack, ptr)
            ptr = ptr.Left
        } else {
            top := stack[len(stack)-1]
            // 判断前一个节点是否小于当前节点
            if top.Val > pre {
                pre = top.Val
            } else {
                return false
            }
            stack = stack[:len(stack)-1]
            ptr = top.Right
        }
    }
    return true
}
```

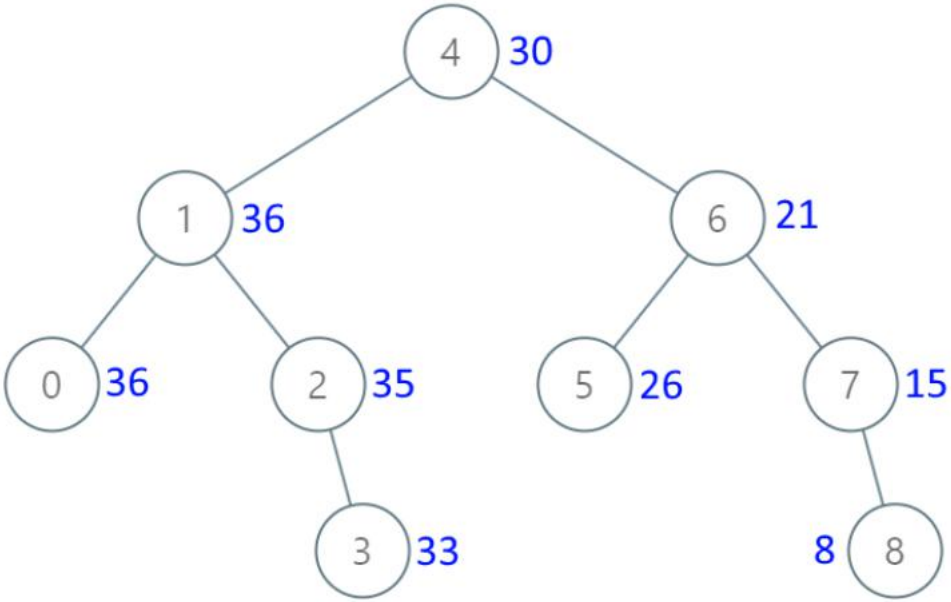
第二周--周三题目--二叉搜索树

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。



第二周--周三题目--二叉搜索树

538. 把二叉搜索树转换为累加树

思路：中序遍历的递归形式，不过这里和中序变量稍微不同的是，先遍历右边子树。

```go
func convertBST(root *TreeNode) *TreeNode {
    num := 0
    var back func(r *TreeNode)
    // 辅助函数
    helper = func(r *TreeNode) {
        if r == nil {
            return
        }
        helper(r.Right) // 先变量右边
        r.Val += num
        num = r.Val
        helper(r.Left)  // 再遍历左边
    }
    helper(root)
    return root
}
```

第二周--周五题目--巧用遍历

199. 二叉树的右视图

给定一棵二叉树，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。

示例:

```
输入: [1,2,3,null,5,null,4]
输出: [1, 3, 4]
解释:

   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
```

第二周--周五题目--巧用遍历

199. 二叉树的右视图

方法一：使用层次遍历，将每一层所维护的队列的最后一个元素取出来，按顺序合并到一个切片

```
func rightSideView(root *TreeNode) (res []int) {
    if root == nil {
        return nil
    }
   queue := []*TreeNode{root}
   for len(queue) > 0 {
       n := len(queue)
       // 将队列最后一个元素取出
       res = append(res, queue[n-1].Val)
       for i:=0; i<n; i++ {
           if queue[0].Left != nil {
               queue = append(queue, queue[0].Left)
           }
           if queue[0].Right != nil {
               queue = append(queue, queue[0].Right)
           }
           queue = queue[1:]
       }
   }
   return
}
```

第二周--周五题目--巧用遍历

199. 二叉树的右视图

方法二：深度优先搜索，不过从题意来看（右视图）这里每个结点需要先遍历右孩子，并且定义一个curMaxDepth用于保存当前已经遍历到的最大深度，一旦有某个结点得深度超过curMaxDepth，说明该结点是从右边看过去，该层能看到的第一个结点，故取出到答案切片，并更新curMaxDepth

```go
func rightSideView(root *TreeNode) (res []int) {
    // 用于保存当前已经遍历到的最大深度
    curMaxDepth := 0
    var helper func(node *TreeNode, depth int)
    helper = func(node *TreeNode, depth int) {
        if node == nil {
            return
        }
        // 发现深度大于之前所保存的深度
        if depth > curMaxDepth {
            // 取出结果
            res = append(res, node.Val)
            // 当前深度
            curMaxDepth = depth
        }
        helper(node.Right, depth+1)
        helper(node.Left, depth+1)
    }
    helper(root, 1)
    return
}
```

第二周--周五题目--巧用遍历

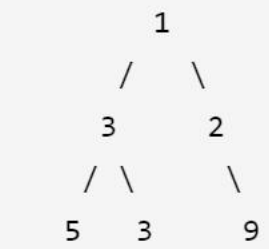
662. 二叉树最大宽度

给定一个二叉树，编写一个函数来获取这个树的最大宽度。树的宽度是所有层中的最大宽度。这个二叉树与**满二叉树（full binary tree）**结构相同，但一些节点为空。

每一层的宽度被定义为两个端点（该层最左和最右的非空节点，两端点间的`null`节点也计入长度）之间的长度。

示例 1:

输入：



输出：4
解释：最大值出现在树的第 3 层，宽度为 4 (5,3,null,9)。

第二周--周五题目--巧用遍历

662. 二叉树最大宽度

思路：层次遍历。然后需要给每一个结点打上标签，表示该节点是该层的第几个结点，然后将该层最后出现的结点和最早出现结点的标签相减+1就是该层的宽度。

```
type QueueNode struct {
     Index int                // 标签。用于保存结点在该层是属于第几个位置
     Node *TreeNode
 }
func widthOfBinaryTree(root *TreeNode) (res int) {
    if root == nil {
        return
    }
    queue := []QueueNode{{1, root}}
    for len(queue) > 0 {
        n := len(queue)
        // 判断该层宽度是否大于已知最大宽度
        if width := (queue[n-1].Index-queue[0].Index) + 1; width > res {
            res = width
        }
        for i:=0; i<n; i++ {
            if queue[0].Node.Left != nil {
                queue = append(queue, QueueNode{queue[0].Index*2-1, queue[0].Node.Left})
            }
            if queue[0].Node.Right != nil {
                queue = append(queue, QueueNode{queue[0].Index*2, queue[0].Node.Right})
            }
            queue = queue[1:]
        }
    }
    return
}
```