Siyao's Blog

【九章算法基础班】二叉树与分治法

2017-11-29 | 算法, 九章算法 | 浏览 次

11,065 | 53

数组：内存空间连续，支持下标访问，访问时间复杂度$O(1)$

链表：内存空间不联系，不支持下表访问，访问时间复杂度$O(n)$

1. 树形分析法求解时间复杂度：

$$T(n) = T(n/2) + O(1) -> O(logn)$$

$$T(n) = T(n/2) + O(n) -> O(n)$$

$$T(n) = 2T(n/2) + O(1) -> O(n)$$

```
T(n) = 2T(n/2) + O(1) -> O(n)
O(1)：一次拆分所需时间
 --                                  n
  |                ↙        ↘       ···············O(1)拆分所需时间
  |              n/2                 n/2
log(n)      ↙       ↘          ↙       ↘        ···············O(2)
  |       n/4       n/4       n/4       n/4
  |     ↙   ↘     ↙   ↘     ↙   ↘     ↙   ↘      ···············O(4)
  |    n/8  n/8  n/8  n/8  n/8  n/8  n/8  n/8
  |                  ···············                                  ···············O(n)
  | n/n n/n     n/n             ·············   n/n     n/n     n/n
 --
 O(1+2+4+....+n) = O(2n-1) = O(n)
 由此如果T(n) = 2T(n/2) + O(1)，则时间复杂度为O(n)
```

$$T(n) = 2T(n/2) + O(n) -> O(n * logn)$$

```
T(n) = 2T(n/2) + O(n) -> O(n)
O(n)：一次拆分所需时间
 --                                  n
  |                ↙        ↘       ·············O(n)拆分所需时间
  |              n/2                 n/2
log(n)      ↙       ↘          ↙       ↘        ·············O(n/2)*2=O(n)
  |       n/4       n/4       n/4       n/4
  |     ↙   ↘     ↙   ↘     ↙   ↘     ↙   ↘      ·············O(n/4)*4=O(n)
  |    n/8  n/8  n/8  n/8  n/8  n/8  n/8  n/8
  |                  ···············                                  ·············O(n/n)*n=O(n)
  | n/n n/n     n/n             ·············   n/n     n/n     n/n
 --
 O(n+n+...+n)*logn = O(nlogn)
 由此如果T(n) = 2T(n/2) + O(n)，则时间复杂度为O(nlogn)
```

2. 二叉树的遍历

树的遍历三种方式：

```
     1
    ↙ ↘
  2     3
 ↙ ↘
4   5
- 前序遍历（根左右）：1->2->4->5->3
- 中序遍历（左根右）：4->2->5->1->3
- 后序遍历（左右根）：4->5->3->2->1
```


2.1.前序遍历Binary Tree Preorder Traversal

题目

> Given a binary tree, return the *preorder* traversal of its nodes' values.
>
> For example:
>
> Given binary tree `[1,null,2,3]`,
>
> ```
> >    1
> >     \
> >      2
> >     /
> >    3
> >
> >
> ```
>
> >
>
> return `[1,2,3]`.

分析

1. 递归法

   首先遍历根节点，然后对其左节点做前序遍历，对其右节点做前序遍历。

   递归三要素：

   - 定义：要做什么事情，这道题就是先遍历父亲节点，然后左节点、右节点
   - 拆分：差分成同样的问题，但规模变小，本题就是拆成左子树和右子树，对左子树和右字数分别做前序遍历
   - 结束条件：遇到空节点停止。

```
public List<Integer> preorderTraversal(TreeNode root) {
  ArrayList<Integer> result = new ArrayList();
  traverse(root,result);
  return result;
}
public void traverse(TreeNode root,ArrayList<Integer> result){
  if(root==null){
    return;
  }
  else{
    result.add(root.val);
    traverse(root.left,result);
    traverse(root.right,result);
  }
}
```

2. 分治法

   分而治之，先分开求结果，再合并

   先得到左子树的结果，再得到右子树的结果，然后将左子、右子树、root结果合并得到最终结果。

   通常来说，分治法的函数是有返回值的。

   分治法三要素：

   - 定义：要做什么事情
   - 拆分与合并问题：
   - 结束条件。

```
public ArrayList<Integer> divide(TreeNode root){
  //结束条件
```

```
        if(root==null){
          ArrayList<Integer> res = new ArrayList<Integer>();
          return res;
        }

        //拆分问题，获取子问题结果
        ArrayList<Integer> leftres = divide(root.left);
        ArrayList<Integer> righttres = divide(root.right);

        //合并子问题
        ArrayList<Integer> result = new ArrayList<>();
        result.add(root.val);
        result.addAll(leftres);
        result.addAll(righttres);

        //返回结果
        return result;
    }
```

3. 非递归方法

阅读理解并背诵

利用stack实现树的前序遍历，每次弹出栈顶元素，先后压入其右孩子和左孩子。

```
public class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        Stack<TreeNode> stack = new Stack<TreeNode>();
        List<Integer> preorder = new ArrayList<Integer>();

        if (root == null) {
            return preorder;
        }
        //根节点入栈
        stack.push(root);
        while (!stack.empty()) {
            TreeNode node = stack.pop();
            preorder.add(node.val);
            //右孩子入栈
            if (node.right != null) {
                stack.push(node.right);
            }
          //左孩子入栈
            if (node.left != null) {
                stack.push(node.left);
            }
        }
        return preorder;
    }
}
```

2.2. 中序遍历Binary Tree Inorder Traversal

题目

> Given a binary tree, return the *inorder* traversal of its nodes' values.
>
> For example:
>
> Given binary tree `[1,null,2,3]`,
>
> ```
> >    1
> >     \
> >      2
> >     /
> >    3
> >
> >
> ```
>
> return `[1,3,2]`.

代码

```
//递归法
    public void helper(TreeNode root,ArrayList<Integer> result){
        //终止条件
        if(root == null){
            return;
        }

        //拆分问题
        helper(root.left,result);
        result.add(root.val);
        helper(root.right,result);

    }
    public List<Integer> inorderTraversal(TreeNode root) {
        ArrayList<Integer> result = new ArrayList();
        helper(root,result);
        return result;
    }

//分治法
    public ArrayList<Integer> inorderTraversal(TreeNode root) {
        //终止条件
        if(root == null){
            return new ArrayList<>();
        }
        ArrayList<Integer> result = new ArrayList<>();
        //拆分
        ArrayList<Integer> leftRes = inorderTraversal(root.left);
        ArrayList<Integer> rightRes = inorderTraversal(root.right);

        //合并
        result.addAll(leftRes);
        result.add(root.val);
        result.addAll(rightRes);

        //返回结果
        return result;
    }

//非递归方法，很重要！！！！！！！！！！！！！！
public ArrayList<Integer> inorderTraversal(TreeNode root) {
        Stack<TreeNode> stack = new Stack<TreeNode>();
        ArrayList<Integer> result = new ArrayList<Integer>();
        TreeNode curt = root;
        while (curt != null || !stack.empty()) {
            while (curt != null) {
                stack.add(curt);
                curt = curt.left;
            }
            curt = stack.pop();
            result.add(curt.val);
            curt = curt.right;
        }
        return result;
    }
```

2.3.后序遍历Binary Tree Postorder Traversal

题目

> Given a binary tree, return the *postorder* traversal of its nodes' values.
>
> For example:
>
> Given binary tree `{1,#,2,3}`,
>
> ```
> >    1
> >     \
> >      2
> >     /
> >    3
> >
> ```

>

>

return [3,2,1].

代码

```
//递归法
public void helper(TreeNode root,ArrayList<Integer> result){
  //终止条件
  if(root == null){
    return;
  }

  //拆分问题
  helper(root.left,result);
  helper(root.right,result);
  result.add(root.val);

}
public List<Integer> postorderTraversal(TreeNode root) {
  ArrayList<Integer> result = new ArrayList();
  helper(root,result);
  return result;
}

//分治法
public ArrayList<Integer> postorderTraversal(TreeNode root) {
  if(root == null){
    return new ArrayList<>();
  }

  ArrayList<Integer> result = new ArrayList<>();
  //拆分
  ArrayList<Integer> leftRes = postorderTraversal(root.left);
  ArrayList<Integer> rightRes = postorderTraversal(root.right);
  //合并
  result.addAll(leftRes);
  result.addAll(rightRes);
  result.add(root.val);
  return result;
}
```

2.4.分治法相比于遍历法的优点：

1. 无需全局变量存储结果，无需helper函数
2. 可并行

2.5.二叉树三种遍历的非递归实现

整体思路

三种遍历的非递归解决思路核心思想是一致的：

1. 将二叉树分为“左”（包括一路向左，经过的所有实际左+根）、“右”（包括实际的右）两种节点
2. 使用同样的顺序将“左”节点入栈
3. 在合适的时机转向（转向后，“右”节点即成为“左”节点）、访问节点、或出栈

比如{1,2,3}，当cur位于节点1时，1、2属于“左”节点，3属于“右”节点。DFS的非递归实现本质上是在协调入栈、出栈和访问，三种操作的顺序。上述统一使得我们不再需要关注入栈顺序，仅需要关注出栈和访问（第3点），随着更详细的分析，你将更加体会到这种简化带来的好处。

将对节点的访问定义为 `results.add(node.val);`，分析如下：

前序遍历

前序遍历的顺序是：根->左->右，按照上面提到的思路，可以简化为左->右

从root节点开始访问，依次向下访问左节点(cur指向当前节点)，此时立即将这些“左”节点输出到结果中，同时把他们压入栈，便于后续访问其右节点：

```
while (cur != null) {
    results.add(cur.val);
    stack.push(cur);
    cur = cur.left;
}
```

上面循环结束意味着我们已经访问过所有的“左”节点，现在需要将这些节点出栈，转到其“右”节点，此时右节点也变成了“左”节点，需要对其进行上面的处理。

```
if (!stack.empty()) {
    cur = stack.pop();
    // 转向
    cur = cur.right;
}
```

完整代码：

```
public List<Integer> nonRecursion(TreeNode root){
        TreeNode cur = root;
        Stack<TreeNode> stack = new Stack<>();
        ArrayList<Integer> results = new ArrayList<>();

        while(cur != null || !stack.empty()){//停止条件：栈和cur都为空
            while (cur != null) {
                results.add(cur.val);//左节点加入结果集
                stack.push(cur);//左节点入栈
                cur = cur.left;
            }
            if(!stack.empty()){
                cur = stack.pop();
                cur = cur.right;
            }
        }
        return results;
    }
```

中序遍历

先序与中序的区别只在于对“左”节点的处理上，前序遍历是先访问实际根，再访问左节点，而中序是先访问实际左节点，再访问实际根节点，所以需要将中序改为出栈时才访问这个节点的值。

```
while (cur != null) {
  stack.push(cur);//左节点入栈
  cur = cur.left;
}
if(!stack.empty()){
  cur = stack.pop();
  results.add(cur.val);//左节点加入结果集
  cur = cur.right;
}
```

完整代码

```
public List<Integer> nonRecursion(TreeNode root){
        TreeNode cur = root;
        Stack<TreeNode> stack = new Stack<>();
        ArrayList<Integer> results = new ArrayList<>();

        while(cur != null || !stack.empty()){//停止条件：栈和cur都为空
            while (cur != null) {
                stack.push(cur);//左节点入栈
                cur = cur.left;
            }
            if(!stack.empty()){
                cur = stack.pop();
                results.add(cur.val);//左节点加入结果集
                cur = cur.right;
            }
        }
        return results;
    }
```

后序遍历

后序遍历的实际访问顺序是：左右根

入栈顺序不变，需考虑转向和出栈时机。

对于实际的根，需要保证先后访问了左子树、右子树之后，才能访问根。实际的右节点、左节点、根节点都会成为"左"节点入栈，所以我们只需要**在出栈之前，将该节点视作实际的根节点，并检查其右子树是否已被访问**即可。如果不存在右子树，或右子树已被访问了，那么可以访问根节点，出栈，并不需要转向；如果还没有访问，就转向，使其"右"节点成为"左"节点，等着它先被访问之后，再来访问根节点。

```
public class BinaryTreePostorderTraversal {
    class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;
        TreeNode(int x) { val = x; }
    }
    class TreeNodeWithFlag {
        TreeNode node;
        boolean flag;
        TreeNodeWithFlag(TreeNode node,boolean flag) {
            this.flag = flag;
            this.node = node;
        }
    }

    public List<Integer> postorderTraversal(TreeNode root) {
        TreeNode cur = root;
        Stack<TreeNodeWithFlag> stack = new Stack<>();
        ArrayList<Integer> results = new ArrayList<>();

        while(cur != null || !stack.empty()){//停止条件：栈空或cur空
            while (cur != null) {
                stack.push(new TreeNodeWithFlag(cur,false));//左节点入栈，标记为右子树未访问
                cur = cur.left;//继续访问左子树
            }
            if(!stack.empty()){
                if(stack.peek().flag){//右子树已经处理过
                    results.add(stack.peek().node.val);//左节点加入结果集
                    stack.pop();//弹出
                    cur = null;//左右根节点都已处理过，不转向，继续弹栈
                }
                else{//右子树没有处理过
                    stack.peek().flag = true;//标记为已处理
                    cur = stack.peek().node.right;
                }
            }
        }
        return results;
    }

   //自己写的一个版本
       public List<Integer> postorderTraversal(TreeNode root) {
         List<Integer> res = new ArrayList<>();
         Stack<TreeNodeWithL> stack = new Stack<>();
         TreeNode node = root;
         while(!stack.isEmpty() || node != null){
             while(node != null){
                 stack.push(new TreeNodeWithL(node,false));
                 node = node.left;
             }//跳出時node==null
             TreeNodeWithL temp = stack.pop();
             if(temp.flag){
                 res.add(temp.node.val);
                 node = null;//左右根节点都已处理过，不转向，继续弹栈
             }
             else{
                 temp.flag = true;
                 stack.push(temp);
                 node = temp.node.right;
             }
         }
         return res;
     }
}
```


3.相关习题

Maximum Depth of Binary Tree

题目

> Given a binary tree, find its maximum depth.
>
> The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

给定一个数，求出树的最大深度

代码

```
public int maxDepth(TreeNode root) {
  //终止条件
  if(root == null){
    return 0;
  }

  //拆分成求解左子树和右子树的最大深度
  int leftDepth = maxDepth(root.left);
  int rihtDepth = maxDepth(root.right);

  //合并，根节点的最大深度=max(左子树最大深度，右子树最大深度)+1
  int res = Math.max(leftDepth,rihtDepth)+1;
  //返回结果
  return res;
}
```

Balanced Binary Tree

题目

> Given a binary tree, determine if it is height-balanced.
>
> For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

判断二叉树是否是平衡二叉树：

1. 左右子树平衡
2. 左子树和右子树的高度差不超过1

分析

考虑某一个时间成立或者不成立需要考虑多个因素的时候，需要定义一个`class ResultType`把这两个值包进去，存储中间结果。

```
        //定义resultType
        class ResultType{
        boolean isBalanced;//是否平衡
        int maxDepth;//最大深度
        ResultType(boolean isBalanced,int maxDepth){
            this.isBalanced = isBalanced;
            this.maxDepth = maxDepth;
        }
    }
    //
    public ResultType helper(TreeNode root){
        //结束条件
        if (root == null) {
            return new ResultType(true,0);
        }
        //拆分成分别计算左子树的右子树信息
        ResultType leftRes = helper(root.left);
        ResultType rightRes = helper(root.right);

        //如果左右有一个不是平衡树
        if(!leftRes.isBalanced || !rightRes.isBalanced){
            return new ResultType(false,-1);
        }
        //都是平衡树，但深度差>1
```

```
            if(Math.abs(leftRes.maxDepth-rightRes.maxDepth) >1){
                return new ResultType(false,-1);
            }
            //都是平衡树，深度差<=1，node节点最大深度=max(左、右子树深度)+1
            return new ResultType(true,Math.max(rightRes.maxDepth,leftRes.maxDepth)+1);
        }

        public boolean isBalanced(TreeNode root) {
            return helper(root).isBalanced;//返回根节点是否是平衡。
        }
```

方法二：不引入额外的结构，利用高度判断，当不平衡时，置高度为-1

```
 public int height(TreeNode root){
      if(root == null){
          return 0;
      }
      int leftH = height(root.left);
      int rightH = height(root.right);
      if(leftH == -1 || rightH == -1 || Math.abs(leftH - rightH) > 1){
          return -1;
      }
      else{
          return Math.max(leftH,rightH)+1;
      }
  }

  public boolean isBalanced(TreeNode root) {
      return height(root)!=-1;
  }
```

Lowest Common Ancestor of a Binary Tree

题目

> Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
>
> According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”
>
> ```
> >          _______3______
> >         /              \
> >     ___5__          ___1__
> >    /      \        /      \
> >    6      _2       0       8
> >           /  \
> >          7   4
> >
> >
> ```
>
> >
>
> For example, the lowest common ancestor (LCA) of nodes `5` and `1` is `3`. Another example is LCA of nodes `5` and `4` is `5`, since a node can be a descendant of itself according to the LCA definition.

给一棵二叉树和二叉树上的两个点，返回其最近公共祖先

分析

```
如果二叉树中存储了父亲节点，则可以从两个点出发往上寻找至root:
比如5和1:
5:[5,3]
1:[1,3]
得到路径之后从后向前遍历，3,3一样，5,1不一样了，所以最近公共祖先是3

再比如5和4:
5:[5,3]
4:[4,2,5,3]
从后向前遍历，发现3,5之后不一样了，所以公共祖先是5

如果没有存储父亲节的信息，给定root节点和两个点n1,n2:
```



```
          _______3______
         /              \
     ___5__          ___1__
    /      \        /      \
    6      _2       0       8
           /  \
          7   4
n1和n2的分布情况有以下几种：
1.其中有一个是root -> 返回root
2.全在左子树 -> 返回左子树root
3.全在右子树 -> 返回右子树root
4.一个在左子树、一个在右子树 -> 返回root
5.这两个点不在这棵树里 -> 返回null

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root == null){
            return null;
        }
        if(root == p || root == q){//其中有一个是根节点
            return root;
        }
        //拆分,分别在左、右子树中寻找LCA
        TreeNode left = lowestCommonAncestor(root.left,p,q);
        TreeNode right = lowestCommonAncestor(root.right,p,q);

        //合并
        if(left != null && right != null){//一个在左子树，一个在右子树，则LCA为root
            return root;
        }
        else if(left != null){//两个节点都在左子树
            return left;
        }
        else if(right != null){//两个节点都在右子树
            return right;
        }
        else //左右子树的LCA都是null,都没有这俩节点
            return null;
    }
```

Binary Tree Maximum Path Sum II

题目

给定一个二叉树，从根节点root出发，求最大路径和，可以在任一点结束

分析

```
如果是求从root到leaf的最大路径，就用分治法，从上到下，每个节点的最大路径是其左子树和右子树的最大路径的最大值:
root.val + Math.max(maxleft,maxright)
如果二叉树上的节点值有负数，那么最大路径就有可能不到leaf就结束了，所以在计算节点最大路径时，如果其左右子树最大路径的最大值为负数，则该节点到leaf的最大路径长度应该为0:
root.val + Math.max(0,Math.max(maxleft,maxright))
```

代码

```
public int maxPathSum2(TreeNode root){
  if(root == null){
    return 0;
  }
  int maxleft = maxPathSum2(root.left);
  int maxright = maxPathSum2(root.right);

  //root->leaf
  //return root.val + Math.max(maxleft,maxright);

  //root->any
  return root.val + Math.max(0,Math.max(maxleft,maxright));
}
```

之前的题目复杂度基本都是O(n)，分析：

一共有多少个点*每个点上的时间复杂度* = *n* O(1) = O(n)

Binary Tree Maximum Path Sum

题目

> Given a binary tree, find the maximum path sum.
>
> For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain **at least one node** and does not need to go through the root.
>
> For example:
> Given the below binary tree,
>
> ```
>        1
>       / \
>      2   3
> ```
>
> Return 6.

分析

跟LCA问题思考方式类似，考虑某一点root的最大路径的位置可能有如下三种情况：
1.都在左子树中(root.left:any->any)
2.都在右子树中(root.right:any->any)
3.跨过root节点左右子树中都有(root:any->any)
对三种情况取个最大，就是该root节点的最大路径长度

对于第三种情况，可以分为三个子问题：
A:从左子节点出发的最大路径长度（root.left->any）
root
B:从右子节点出发的最大路径长度（root.right->any）
对三个子问题求和，就是跨过此root节点的最大路径长度

这里需要同时计算any->any和root->any，所以需要定义一个ResultType存储每个节点的any->any和root->any

代码

```
class ResultType{
    int root2any;
    int any2any;
    ResultType(int root2any,int any2any){
        this.any2any = any2any;
        this.root2any = root2any;
    }
}

public ResultType helper(TreeNode root){
    //题目要求至少要包含一个node，所以此时不满足条件，应返回负无穷
    if(root == null){
        return new ResultType(Integer.MIN_VALUE,Integer.MIN_VALUE);
    }

    //divide
    ResultType leftRes = helper(root.left);
    ResultType rightRes = helper(root.right);

    //conquer
    //root到左边或右边
    int root2any = Math.max(0,Math.max(leftRes.root2any,rightRes.root2any)) + root.val;

    //包含完全在左边、完全在右边和跨过root三种情况
    //完全在左边和完全在右边的情况
    int any2any = Math.max(leftRes.any2any,rightRes.any2any);

    //跨过root,分别在两边的情况
    any2any = Math.max(any2any,
            Math.max(0,leftRes.root2any)
            + Math.max(0,rightRes.root2any)
            + root.val
            );
    return new ResultType(root2any,any2any);
}

public int maxPathSum(TreeNode root){
    return helper(root).any2any;
}
```

方法二：

```
public class BinaryTreeMaximumPathSum {
    int maxValue;

    //从某root向下到任意一个节点的最大值，包括本身
    public int maxPathDown(TreeNode root){
        if(root == null){
            return 0;
        }
        int left = Math.max(0,maxPathDown(root.left));
        int right = Math.max(0,maxPathDown(root.right));
        maxValue = Math.max(maxValue,left+right+root.val);//更新最大值
        return Math.max(left,right)+root.val;//从当前节点到任意一个节点的路径和最大值
    }

    public int maxPathSum(TreeNode root) {
        maxValue = Integer.MIN_VALUE;
        maxPathDown(root);
        return maxValue;
    }
}
```

Longest Univalue Path

> Given a binary tree, find the length of the longest path where each node in the path has the same value. This path may or may not pass through the root.
>
> **Note:** The length of path between two nodes is represented by the number of edges between them.
>
> **Example 1:**
>
> Input:
>
> ```
>              5
>             / \
>            4   5
>           / \   \
>          1   1   5
> ```

>

> Output:
>
> ```
> 2
> ```

>

> **Example 2:**
>
> Input:
>
> ```
>              1
>             / \
>            4   5
>           / \   \
>          4   4   5
> ```

>

> Output:
>
> ```
> > 2
> >
> ```

求给定二叉树上某一个相同的值的最大路径长度

针对每一个点：

1. 计算通过它本身的相同值最大路径长度，更新全局最大长度
2. 返回以它本身为根节点，向下的最大深度

```
class Solution {
    int maxLen;
  //从root点出发和自己相同值的最长路径长度
    public int helper(TreeNode root){
        if(root == null){
            return 0;
        }
        int left = helper(root.left);
        int right = helper(root.right);
        int len = 1;//记录过当前节点的最大路径长度
        int res = 1;//记录以当前节点为起点向下的最大路径长度
        if(root.left != null && root.val == root.left.val){
            len+=left;
            res = Math.max(res,left+1);
        }
        if(root.right != null && root.val == root.right.val){
            len+=right;
            res = Math.max(res,right+1);
        }
        maxLen = Math.max(maxLen,len);
        return res;
    }

    public int longestUnivaluePath(TreeNode root) {
        if(root == null){
            return 0;
        }
        maxLen = 1;
        helper(root);
        return maxLen-1;
    }
}
```

Diameter of Binary Tree

> Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the **longest** path between any two nodes in a tree. This path may or may not pass through the root.
>
> **Example:**
> Given a binary tree
>
> ```
> >           1
> >          / \
> >         2   3
> >        / \
> >       4   5
> >
> >
> ```

>

> Return **3**, which is the length of the path [4,2,1,3] or [5,2,1,3].
>
> **Note:** The length of path between two nodes is represented by the number of edges between them.

求二叉树中的最长路径，不一定过root点

思路：

求过每一个点的最长路径，取全局最大值

过某一个点的最长路径=左子树的最大高度+右子树的最大高度

因此需要分治法计算左右子树的最大高度，求和

计算某一个节点的最大高度需要计算其左子树的右子树的最大高度，然后取最大值+1

两个函数整合成一个，同时更新全局最大路径长度和节点高度，代码如下：

代码：

```
class Solution {
    int max = 0;
    //计算到bottom最长距离
    public int longest(TreeNode root){
        if(root == null){
            return 0;
        }
        int left = longest(root.left);//左最长
        int right = longest(root.right);//右最长
        max = Math.max(max,left+right);
        return Math.max(left,right)+1;
    }
    public int diameterOfBinaryTree(TreeNode root) {
        longest(root);
        return max;
    }
}
```

Convert BST to Greater Tree

> Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the original key plus sum of all keys greater than the original key in BST.
>
> **Example:**
>
> ```
> > Input: The root of a Binary Search Tree like this:
> >              5
> >            /   \
> >           2     13
> >
> > Output: The root of a Greater Tree like this:
> >             18
> >            /   \
> >          20     13
> >
> ```

思路：

按照右根左的顺序遍历二叉树，记录遍历过的节点的值的和，依次修改遍历的节点。

代码：

```
class Solution {
    int sum = 0;
    public TreeNode convertBST(TreeNode root) {
        //右根左遍历求和
        if(root == null){
            return root;
        }
        root.right = convertBST(root.right);
        sum += root.val;
        root.val = sum;
        root.left = convertBST(root.left);
        return root;
```

```
        }
    }
```

Subtree of Another Tree

> Given two non-empty binary trees **s** and **t**, check whether tree **t** has exactly the same structure and node values with a subtree of **s**. A subtree of **s** is a tree consists of a node in **s** and all of this node's descendants. The tree **s** could also be considered as a subtree of itself.
>
> **Example 1:**
> Given tree s:
>
> ```
>      3
>     / \
>    4   5
>   / \
>  1   2
>
>
> ```
>
> >
>
> Given tree t:
>
> ```
>    4
>   / \
>  1   2
>
>
> ```
>
> >
>
> Return
>
> true
>
> , because t has the same structure and node values with a subtree of s.
>
> **Example 2:**
> Given tree s:
>
> ```
>      3
>     / \
>    4   5
>   / \
>  1   2
>     /
>    0
>
>
> ```
>
> >
>
> Given tree t:
>
> ```
>    4
>   / \
>  1   2
>
>
> ```
>
> >
>
> Return false

判断t是否是s的子树

代码：

```
//从树s的s节点开始的子树，是否和t完全一样
public boolean isSame(TreeNode s, TreeNode t){
    if(s == null && t == null){
        return true;
    }
    if(s == null || t == null){
        return false;
    }
    if(s.val == t.val){
        return isSame(s.left,t.left) && isSame(s.right,t.right);
    }
    //遇到不相等的点直接返回false
    else{
        return false;
    }
}
public boolean isSubtree(TreeNode s, TreeNode t) {
    if(s == null){return false;}
    //如果从s点开始和t完全一样，返回true
    if(isSame(s,t)){
        return true;
    }
    //否则考察左右节点
    else {
        return isSubtree(s.left,t) || isSubtree(s.right,t);
    }
}
```

Boundary of Binary Tree

> Given a binary tree, return the values of its boundary in **anti-clockwise** direction starting from root. Boundary includes left boundary, leaves, and right boundary in order without duplicate nodes.
>
> **Left boundary** is defined as the path from root to the **left-most** node. **Right boundary** is defined as the path from root to the **right-most**node. If the root doesn't have left subtree or right subtree, then the root itself is left boundary or right boundary. Note this definition only applies to the input binary tree, and not applies to any subtrees.
>
> The **left-most** node is defined as a **leaf** node you could reach when you always firstly travel to the left subtree if exists. If not, travel to the right subtree. Repeat until you reach a leaf node.
>
> The **right-most** node is also defined by the same way with left and right exchanged.
>
> **Example 1**
>
> ```
> Input:
>   1
>    \
>     2
>    / \
>   3   4
>
> Ouput:
> [1, 3, 4, 2]
>
> Explanation:
> The root doesn't have left subtree, so the root itself is left boundary.
> The leaves are node 3 and 4.
> The right boundary are node 1,2,4. Note the anti-clockwise direction means you should output reversed right boundary.
> So order them in anti-clockwise without duplicates and we have [1,3,4,2].
>
>
> ```
>
> >
>
> **Example 2**
>
> ```
> Input:
>     ____1_____
>    /          \
>   2            3
> ```

```
>  / \          /
> 4   5        6
>    / \      / \
>   7   8    9  10
>
> Ouput:
> [1,2,4,7,8,9,10,6,3]
>
> Explanation:
> The left boundary are node 1,2,4. (4 is the left-most node according to definition)
> The leaves are node 4,7,8,9,10.
> The right boundary are node 1,3,6,10. (10 is the right-most node).
> So order them in anti-clockwise without duplicate nodes we have [1,2,4,7,8,9,10,6,3].
>
```

思路：

分三部分：

1. 找到leftmost节点，过程中将左边界节点加入res
2. 找到叶子节点，加入res
3. 找到rightmost节点，过程中将左边界节点加入res

代码：

```
class Solution {
    public TreeNode leftMost(TreeNode root,List<Integer> left){
        if(root.left == null && root.right == null){//找到left-most点
            return root;
        }
        if(root.left != null){
            left.add(root.left.val);
            return leftMost( root.left,left);
        }
        else {
            left.add(root.right.val);
            return leftMost( root.right,left);
        }
    }

    public TreeNode rightMost(TreeNode root,List<Integer> right){
        if(root.left == null && root.right == null){//找到right-most点
            return root;
        }
        if(root.right != null){
            right.add(0,root.right.val);
            //System.out.println(root.right.val);
            return rightMost( root.right,right);
        }
        else {
            right.add(0,root.left.val);
            return rightMost( root.left,right);
        }
    }
    public void getLeaf(TreeNode root,List<Integer> leaf,TreeNode leftmost,TreeNode rightmost){
        //叶子节点且不是左右轮廓
        if(root.right == null && root.left == null && root != leftmost && root != rightmost){
            leaf.add(root.val);
        }
        if(root.left != null){
            getLeaf(root.left,leaf,leftmost,rightmost);
        }
        if(root.right != null){
            getLeaf(root.right,leaf,leftmost,rightmost);
        }
    }

    public List<Integer> boundaryOfBinaryTree(TreeNode root) {
        List<Integer> left = new ArrayList<>();
        List<Integer> right = new ArrayList<>();
        List<Integer> leaf = new ArrayList<>();
        if(root == null){
            return left;
        }
        left.add(root.val);
        TreeNode leftmost = root;
        TreeNode rightmost = root;
```



```
        //如果有左孩子，加入左轮廓
        if(root.left != null){
            leftmost = leftMost(root,left);
        }
        //如果有右孩子，加入右轮廓
        if(root.right != null){
            rightmost = rightMost(root,right);
        }
        getLeaf(root,leaf,leftmost,rightmost);
        // System.out.println(left);
        // System.out.println(leaf);
        // System.out.println(right);
        left.addAll(leaf);
        left.addAll(right);
        return left;
    }
}
```

4.二叉搜索树(BST)

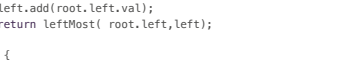
二叉搜索树的基本性质

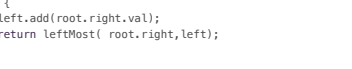
1. 从定义出发

   左子树都比跟节点小

   右子树都比根节点大

   遇到重复的元素，可以自行定义放在左子树还是右子树

   因为BST的定义是比根节点小（包括相等）的要么都在左边，要么都在右边

2. 从效果出发

   BST的中序遍历是升序序列

   例：

   利用BST的这个性质，可以做排序：

   比如给定一个无序序列[2,1,4,3,5]

   可以构造一个BST，然后再中序遍历，输出序列就是有序序列了

   因此，**BST又叫排序二叉树**

   相关练习题：

   Binary Search Tree 的insert、remove等

3. 性质

   - 如果一棵二叉树的中序遍历不是升序，则一定不是BST
   - 如果一棵二叉树的中序遍历是升序，也未必是BST

     比如：

```
    1
  ↙   ↘
 1     1
```

中序遍历得到`[1,1,1]`，是升序序列，但是不是BST

- 存在重复元素时，要么都在左子树要么都在右子树，不可以两边都有

Validate Binary Search Tree

题目

> Given a binary tree, determine if it is a valid binary search tree (BST).
>
> Assume a BST is defined as follows:
>
> - The left subtree of a node contains only nodes with keys **less than** the node's key.
> - The right subtree of a node contains only nodes with keys **greater than** the node's key.
> - Both the left and right subtrees must also be binary search trees.
>
> **Example 1:**
>
> ```
> >     2
> >    / \
> >   1   3
> >
> >
> ```
>
> >
>
> Binary tree
>
> ```
> > [2,1,3]
> >
> ```
>
> >
>
> , return true.
>
> **Example 2:**
>
> ```
> >     1
> >    / \
> >   2   3
> >
> >
> ```
>
> >
>
> Binary tree
>
> ```
> > [1,2,3]
> >
> ```
>
> >
>
> , return false.

分析

给定一个二叉树，判断是否是二叉搜索树，两种思路:

1. 根据二叉搜索树的性质，中序遍历是升序序列，可以对给定二叉树进行中序遍历，输出序列如果是升序序列则是二叉搜索树，可以利用二叉树的非递归中序遍历，每次弹栈时跟前一个元素进行比较，如果小于等于前一个元素就直接返回false。
2. 分治法递归判断，需要存储节点以下是否是BST，以及节点以下部分的最大最小值，和root节点进行比较，是否满足左子树所有元素都小于root,右子树所有节点都大于root。

代码

```
//分治法
class ResultType{
        boolean isValid;
        long min;
        long max;
        ResultType(boolean isValid,long min,long max){
            this.isValid = isValid;
            this.max = max;
            this.min = min;
        }
    }
    public ResultType helper(TreeNode root){
        if(root == null){
            return new ResultType(true,Long.MAX_VALUE,Long.MIN_VALUE);
        }
        //拆分
        ResultType leftValid = helper(root.left);
        ResultType rightValid = helper(root.right);

        //合并
        if(leftValid.isValid && rightValid.isValid){//左右子树都是BST
            if(leftValid.max < root.val && rightValid.min > root.val){//左边元素都比root小，右边元素都比root大
                ResultType res = new ResultType(true,Math.min(leftValid.min,root.val),Math.max(rightValid.max,root.val));
                return res;
            }
            return new ResultType(false,Math.min(leftValid.min,root.val),Math.max(rightValid.max,root.val));
        }
        return new ResultType(false,Long.MIN_VALUE,Long.MAX_VALUE);
    }

    public boolean isValidBST(TreeNode root) {
        return helper(root).isValid;
    }

//非递归中序遍历
    public boolean isValidBST2(TreeNode root){
        int last = Integer.MIN_VALUE;
        TreeNode cur = root;
        Stack<TreeNode> stack = new Stack<>();
        if (root==null){
            return true;
        }
        while(cur != null ||!stack.empty()){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }
            if(!stack.empty()){
                cur = stack.pop();
                if(cur.val <= last){
                    return false;
                }
                cur = cur.right;
            }
        }
        return true;
    }
```

Binary Search Tree Iterator

题目

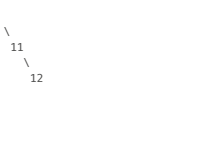
> Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.
>
> Calling `next()` will return the next smallest number in the BST.
>
> **Note:** `next()` and `hasNext()` should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.

设计实现一个带有下列属性的二叉查找树的迭代器：

1. 元素按照递增的顺序被访问（比如中序遍历）
2. next()和hasNext()的询问操作要求**均摊**时间复杂度是O(1)

分析

对于下列二叉查找树，使用迭代器进行中序遍历的结果为 [3, 6, 7, 8, 9, 10, 11, 12]



- 本题相当于考察了BST的非递归中序遍历
- 需要maintain一个stack，首先从root开始push入栈直到最左节点

  初始stack为：

```
10, 6, 3
```

- 在遍历过程中，如果某个节点存在右儿子，则继续从右儿子开始push入栈直到其最左节点

  result = 3, 6

  因为6有右儿子，所以6被pop出去之后，从6为root开始push入栈直到最左节点，然后stack为：

```
10，  9，  8，  7
```

代码

```
class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode(int x) {
            val = x;
        }
    }
public class BSTIterator {
        TreeNode root;
        Stack<TreeNode> stack = new Stack<>();

                //初始化，将root全部left入栈
        public BSTIterator(TreeNode root) {
            this.root = root;
            while(root != null){
                stack.push(root);
                root = root.left;
            }
        }

        /** @return whether we have a next smallest number */
        public boolean hasNext() {
            return !stack.empty();
        }

        /** @return the next smallest number */
        public int next() {
            TreeNode temp = stack.pop();
            int val = temp.val;
            temp = temp.right;
                        //如果弹出节点有右节点，右节点视作左节点，全部左孩子入栈
            while(temp != null){
                stack.push(temp);
                temp = temp.left;
            }
            return val;//返回当前节点值
```



```
        }
    }
```

Inorder Successor in BST

题目

> Given a binary search tree and a node in it, find the in-order successor of that node in the BST.
>
> **Note**: If the given node has no in-order successor in the tree, return `null`.

给定一个二叉搜索树和一个节点p，返回p的后继节点（中序遍历时和p相邻比p大的节点）

分析

1. BST的stack实现，弹栈遇到p节点时标记下，下一次弹栈的节点即是所求
2. 递归实现：
   - 如果p的值小于root，则在左子树中寻找后继，若没有找到，则root就是p的后继
   - 如果p的值大于root，则在右子树中寻找后继节点

代码

```
//非递归
public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
  Stack<TreeNode> stack = new Stack<>();
  TreeNode cur = root;
  if(root == null){
    return null;
  }
  boolean flag = false;
  while(cur != null || !stack.empty()){
    while(cur != null){
      stack.push(cur);
      cur = cur.left;
    }
    if(!stack.empty()){
      cur = stack.pop();
      if(flag == true){
        return cur;
      }
      if(cur == p){
        flag = true;
      }
      cur = cur.right;
    }
  }
  return null;
}
//递归
 public TreeNode inorderSuccessor2(TreeNode root, TreeNode p){
        if(root == null || p == null){
            return null;
        }
        //后级节点在左子树或root节点中
        if(p.val < root.val){
            //后继节点在以左孩子为root的子树中的位置
            TreeNode left = inorderSuccessor2(root.left,p);
            //没有刚好比p大一个的，则root就是p的后继节点
                if(left == null){
                return  root;
            }
                //在左子树中找到了p的后继
            else {
                return left;
            }
        }
        //后继节点在右子树中
        else{
            return inorderSuccessor2(root.right,p);
        }
    }
```

Search Range in Binary Search Tree

题目

> Given two values k1 and k2 (where k1 < k2) and a root pointer to a Binary Search Tree. Find all the keys of tree in range k1 to k2. i.e. print all x such that k1<=x<=k2 and x is a key of given BST. Return all the keys in ascending order.
>
> Have you met this question in a real interview?
>
> Yes
>
> Example
>
> If k1 = `10` and k2 = `22`, then your function should return `[12, 20, 22]`.
>
> ```
> >     20
> >    /  \
> >   8   22
> >  / \
> > 4   12
> >
> ```

给定一个二叉搜索树和一个区间，要求升序输出二叉搜索树中在给定区间内的节点

分析

中序遍历BST，将符合条件的结果输出即可

代码

```
public List<Integer> searchRange(TreeNode root, int k1, int k2) {
    List<Integer> results = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
    TreeNode cur = root;
    if(root == null){
        return results;
    }
    while(cur != null || !stack.empty()){
        while(cur != null){
            stack.push(cur);
            cur = cur.left;
        }
        if(!stack.empty()){
            cur = stack.pop();
            if(cur.val <= k2 && cur.val >= k1) {
                results.add(cur.val);
            }
            cur = cur.right;
        }
    }
    return results;
}
```

Insert Node in Binary Search Tree

题目

> Given a binary search tree and a new tree node, insert the node into the tree. You should keep the tree still be a valid binary search tree.
>
> **Notice**
>
> You can assume there is no duplicate values in this tree + node.
>
> Example
>
> Given binary search tree as follow, after Insert node 6, the tree should be:


> ```
> >  /2\           /2\
> > 1   4   -->   1   4
> >    /             / \
> >   3             3   6
> >
> ```

给定一个二叉搜索树和一个新节点，插入新节点，保持二叉树仍然是二叉搜索树

分析

递归实现，判断node的值和root节点的大小关系，返回值是当前root节点：

1. node小于root，要插在左子树，root.left = insertNode(root.left,node);
2. node大于root，要插在右子树，root.right= insertNode(root.right,node);

代码

```
public TreeNode insertNode(TreeNode root, TreeNode node) {
        // 找到插入位置，插入node节点
        if(root == null){
            return node;
        }
        //在左子树
        if(node.val < root.val){
            root.left = insertNode(root.left,node);
        }
        //在右子树
        else{
            root.right = insertNode(root.right,node);
        }
        return root;
    }
```

Delete Node in a BST

题目

> Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.
>
> Basically, the deletion can be divided into two stages:
>
> 1. Search for a node to remove.
> 2. If the node is found, delete the node.
>
> **Note:** Time complexity should be O(height of tree).
>
> **Example:**
>
> ```
> > root = [5,3,6,2,4,null,7]
> > key = 3
> >
> >     5
> >    / \
> >   3   6
> >  / \   \
> > 2   4   7
> >
> > Given key to delete is 3. So we find the node with value 3 and delete it.
> >
> > One valid answer is [5,4,6,2,null,null,7], shown in the following BST.
> >
> >     5
> >    / \
> >   4   6
> >  /     \
> > 2       7
> >
> > Another valid answer is [5,2,6,null,4,null,7].
> >
> >     5
> >    / \
> ```

```
>   2   6
>    \   \
>     4   7
>
```

分析

首先要找到要删除的节点，然后用它左子树的最大值（或右子树的最小值）的值取代要删除节点的值，再将左子树的最大值（或右子树的最小值）节点删除。

代码

```
//查找以root为跟的树的最大节点，root非空
public TreeNode findMax(TreeNode root){
  while(root.right!=null){
    root= root.right;
  }
  return root;
}

//查找并删除节点
public TreeNode deleteNode(TreeNode root, int key) {
  if(root == null){
    return  null;
  }
  //在左子树
  if(root.val > key){
    root.left = deleteNode(root.left,key);
  }
  //在右子树
  else if(root.val < key){
    root.right = deleteNode(root.right,key);
  }
  //要删除的就是root
  else{
    if(root.left == null){//如果左节点是空的
      return root.right;//返回右节点
    }
    else if (root.right == null){//左节点非空，右节点为空
      return root.left;//返回左节点
    }
    //左右节点都非空，用左子树最大节点取代当前节点
    root.val = findMax(root.left).val;
    //删除左子树最大值节点
    root.left = deleteNode(root.left,root.val);
  }
  return root;
}
```

Count of Smaller Numbers After Self

题目

> You are given an integer array *nums* and you have to return a new *counts* array. The *counts* array has the property where `counts[i]` is the number of smaller elements to the right of `nums[i]`.
>
> **Example:**
>
> ```
> > Given nums = [5, 2, 6, 1]
> >
> > To the right of 5 there are 2 smaller elements (2 and 1).
> > To the right of 2 there is only 1 smaller element (1).
> > To the right of 6 there is 1 smaller element (1).
> > To the right of 1 there is 0 smaller element.
> >
> >
> ```
>
> >

> Return the array `[2, 1, 1, 0]`.

思路

1. baseline：插入排序+二分优化

   开一个新的数组，从后向前遍历原数组，将数字插入新数组的合适位置，该位置就是原数组中在其右侧小于其的数字个数。插入过程采用二分查找。

2. BST

   构建一个BST，节点记录当前出现的比自己小的元素个数，对于每个节点分三种情况讨论：

   1. val < root.val，递归向root.left插入节点，同时root的smallerNum++
   2. val == root.val，递归向root.right插入节点，其smallerNum = root.smallerNum+insert(root.right,val)
   3. val > root.val，递归向root.right插入节点，其smallerNum = root.smallerNum+1+insert(root.right,val)

代码

```
class Solution {
    class Node{
        int val;
        int num;//记录比val小的所有数字数目
        Node left;
        Node right;
        Node(int val,int num){
            this.val = val;
            this.num = num;
        }
    }
    //构建BST
    public int insert(Node root,int val){
        if(val == root.val){
            if(root.right == null){
                root.right = new Node(val,0);
                return root.num;
            }
            else {
                return insert(root.right,val) + root.num;
            }
        }
        if(val > root.val){
            if(root.right == null){
                root.right = new Node(val,0);
                return root.num + 1;
            }
            else{
                return insert(root.right,val) + root.num + 1;
            }
        }
        else {
            root.num++;
            if(root.left == null){
                root.left = new Node(val,0);
                return 0;
            }
            return insert(root.left,val);
        }
    }


    public List<Integer> countSmaller(int[] nums) {
        List<Integer> res = new ArrayList<>();
        if(nums.length == 0){
            return res;
        }
        Node root = new Node(nums[nums.length-1],0);
        res.add(0);
        for(int i = nums.length-2;i >= 0;i--){
            res.add(0,insert(root,nums[i]));
        }
        return res;
    }
}
```

有时间可以再写一写这道题，还不是很好

Unique Binary Search Trees

题目：

> Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1...*n*?
>
> For example,
>
> Given *n* = 3, there are a total of 5 unique BST's.
>
> ```
> >    1         3     3      2      1
> >     \       /     /      / \      \
> >      3     2     1      1   3      2
> >     /     /       \                 \
> >    2     1         2                 3
> >
> ```

给定数组n，返回1-n共n个节点一共可以构成多少个独一无二的BST

分析：

DP

遍历1-n个节点，一次选取一个k作为root,则其左边有1...k-1共k-1个节点，右边有k+1...n共n-k个节点，因此数量是两边BST数量的乘积，则这种情况共有dp[k-1]*dp[n-k]种不同的BST

因此dp[n] = dp[0]dp[n-1] + dp[1]dp[n-2]+...+dp[n-1]dp[0]

代码：

```
public int numTrees(int n) {
        int[] dp = new int[n+1];
        dp[0] = 1;
        dp[1] = 1;
        for(int i = 2; i <= n;i ++){
            for(int j = 0;j < i;j++){
                dp[i] += dp[j] * dp[i-j-1];
            }
        }
        return dp[n];
    }
```

Unique Binary Search Trees II

题目

> Given an integer *n*, generate all structurally unique **BST's** (binary search trees) that store values 1...*n*.
>
> For example,
>
> Given *n* = 3, your program should return all 5 unique BST's shown below.
>
> ```
> >    1         3     3      2      1
> >     \       /     /      / \      \
> >      3     2     1      1   3      2
> >     /     /       \                 \
> >    2     1         2                 3
> >
> ```

给定数组n，返回1-n共n个节点构成的所有独一无二的BST

分析

遍历每一个节点k，以该节点为root的BST可以由其左、右子树所有可能的情况构成

代码

```
public List<TreeNode> create(int start,int end){
    List<TreeNode> results = new LinkedList<>();
    if(start > end){
        results.add(null);
        return results;
    }
    for(int i = start; i <= end;i++){
        List<TreeNode> left = create(start,i-1);//计算左子树所有可能情况
        List<TreeNode> right = create(i+1,end);//计算右子树所有可能情况
      //遍历左右子树所有情况，以当前节点为root，构造BST加入当前结果集
        for(int j = 0; j < left.size();j++){
            for(int k = 0; k < right.size() ; k++){
                TreeNode root = new TreeNode(i);
                root.left = left.get(j);
                root.right = right.get(k);
                results.add(root);
            }
        }
    }
    return results;//返回当前层所有BST
}

public List<TreeNode> generateTrees(int n) {
    if(n==0){
        return new ArrayList<>();
    }
    return create(1,n);
}
```


5.二叉树的层序遍历

Binary Tree Level Order Traversal

题目

> Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).
>
> For example:
>
> Given binary tree `[3,9,20,null,null,15,7]`,
>
> ```
> >     3
> >    / \
> >   9  20
> >     /  \
> >    15   7
> >
> >
> ```

>

> return its level order traversal as:
>
> ```
> > [
> >   [3],
> >   [9,20],
> >   [15,7]
> > ]
> >
> ```

分析

广度优先搜索，利用 `Queue` ，先进先出

```
-----|--------|-------
<- 3 | 9   20 | 15   7
-----|--------|-------
```

代码

```
public List<List<Integer>> levelOrder(TreeNode root) {
  List<List<Integer>> results = new ArrayList<>();
  Queue<TreeNode> queue = new LinkedList<>();
```

```
    if(root == null){
      return results;
    }

    queue.add(root);
    while(!queue.isEmpty()){
      List<Integer> res = new ArrayList<>();
      int size = queue.size();
      for(int i = 0 ; i < size ;i++){
        TreeNode temp = queue.poll();
        res.add(temp.val);
        if(temp.left != null){
          queue.add(temp.left);
        }
        if(temp.right != null){
          queue.add(temp.right);
        }
      }
      results.add(res);
    }
    return results;
  }
```

宽度优先搜索最常用的数据结构是队列和hash表，但是在二叉树的问题中不会用到hash表，只会用到Queue队列

Binary Tree Level Order Traversal II

题目

> Given a binary tree, return the *bottom-up level order* traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).
>
> For example:
>
> Given binary tree `[3,9,20,null,null,15,7]`,
>
> ```
> >     3
> >    / \
> >   9  20
> >     /  \
> >    15   7
> >
> >
> ```
>
> >
>
> return its bottom-up level order traversal as:
>
> ```
> > [
> >   [15,7],
> >   [9,20],
> >   [3]
> > ]
> >
> ```

分析

跟上一题的差别是层与层之间的顺序是反的，每层元素的顺序不变，所以只需要插入最终的结果集的时候反序插入，所以可以利用LinkedList，每次将每层的结果在前面插入。

代码

```
 LinkedList<List<Integer>> results = new LinkedList<>();
 Queue<TreeNode> queue = new LinkedList<>();

 if(root == null){
   return results;
 }

 queue.add(root);
 while(!queue.isEmpty()){
   List<Integer> res = new ArrayList<>();
   int size = queue.size();
```



```
   for(int i = 0 ; i < size ;i++){
     TreeNode temp = queue.poll();
     res.add(temp.val);
     if(temp.left != null){
       queue.add(temp.left);
     }
     if(temp.right != null){
       queue.add(temp.right);
     }
   }
   results.addFirst(res);
 }
 return results;
```

Binary Tree Zigzag Level Order Traversal

题目

> Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).
>
> For example:
>
> Given binary tree `[3,9,20,null,null,15,7]`,
>
> ```
> >     3
> >    / \
> >   9  20
> >     /  \
> >    15   7
> >
> >
> ```
>
> >
>
> return its zigzag level order traversal as:
>
> ```
> > [
> >   [3],
> >   [20,9],
> >   [15,7]
> > ]
> >
> ```

分析

这道题是每一层交替正反序输出，层与层之间的顺序不变，所以result可以使用ArratList，但是每一层的子res要是用LinkedList，额外立flag取定是从前插入还是从后插入。

代码

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
  List<List<Integer>> results = new ArrayList<>();
  Queue<TreeNode> queue = new LinkedList<>();

  if(root == null){
    return results;
  }

  queue.add(root);
  int flag = 0;

  while(!queue.isEmpty()){
    LinkedList<Integer> res = new LinkedList<>();
    int size = queue.size();
    for(int i = 0 ; i < size ;i++){
      TreeNode temp = queue.poll();
      if(flag%2 == 1){
        res.addFirst(temp.val);
      }
      else{
        res.addLast(temp.val);
      }
```

```
            if(temp.left != null){
              queue.add(temp.left);
            }
            if(temp.right != null){
              queue.add(temp.right);
            }
        }
        flag++;
        results.add(res);
    }
    return results;
}
```

Serialize and Deserialize Binary Tree

题目

> Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.
>
> Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.
>
> For example, you may serialize the following tree
>
> ```
> >     1
> >    / \
> >   2   3
> >      / \
> >     4   5
> >
> >
> ```
>
> >
>
> as `"[1,2,3,null,null,4,5]"`, just the same as how LeetCode OJ serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

思路

序列化so easy，用queue层序遍历即可

反序列化也要用queue，将root节点放入队列，然后数组中的前两个元素是其左右孩子，依次加入队列。。。。

代码

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Codec {
    public String serialize(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        StringBuilder res = new StringBuilder();
        if(root == null){
            return res.toString();
        }
        queue.add(root);
        while(!queue.isEmpty()){
            TreeNode temp = queue.poll();
            if(temp != null){
                res.append(temp.val+",");
            }
            else {
                res.append("null,");
                continue;
```



```
            }
            queue.add(temp.left);
            queue.add(temp.right);
        }
        res.subSequence(0,res.length()-1);
        return res.toString();
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if(data.length() == 0){
            return null;
        }
        String[] nums = data.split(",");
        Queue<TreeNode> queue = new LinkedList<>();
        TreeNode root = new TreeNode(Integer.parseInt(nums[0]));
        queue.add(root);
        for(int i = 1; i < nums.length;i++){
            TreeNode parent = queue.poll();
            if(!nums[i].equals("null")){
                parent.left = new TreeNode(Integer.parseInt(nums[i]));
                queue.add(parent.left);
            }
            if(!nums[++i].equals("null")){
                parent.right = new TreeNode(Integer.parseInt(nums[i]));
                queue.add(parent.right);
            }
        }
        return root;
    }
}

// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.deserialize(codec.serialize(root));
```

6.二叉树的深度优先遍历

DFS是搜索算法的一种。它沿着树的深度遍历树的节点，尽可能深的搜索树的分支。

当节点v的所有边都已被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。

如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。

如上图的例子，DFS访问数组为：ABDECFG。

实现方式

分析一下，在遍历了根结点后，就开始遍历左子树，最后才是右子树。

因此可以借助堆栈的数据结构，由于堆栈是后进先出的顺序，由此可以先将右子树压栈，然后再对左子树压栈，

这样一来，左子树结点就存在了栈顶上，因此某结点的左子树能在它的右子树遍历之前被遍历。

递归

思路比较简单，就是从root开始，先将root值加入结果集，然后先对其做左节点递归调用做DFS，然后是对右节点DFS。当遇到空节点时，返回上层。

```
public class BinaryTreeDFS {
    class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode(int val) {
            this.val = val;
        }
    }

    public void DFSRecurtionHelper(TreeNode root,List<Integer> results){
        //遇到空节点，返回
        if (root == null){
```

```
            return;
        }
        //root放入results,递归处理左右节点
        results.add(root.val);
        DFSRecurtion(root.left);
        DFSRecurtion(root.right);
    }

    public List<Integer> DFSRecurtion(TreeNode root){
        List<Integer> results = new ArrayList<>();
        DFSRecurtionHelper(root,results);
        return results;
    }
}
```

非递归（栈）

因此可以借助堆栈的数据结构，由于堆栈是后进先出的顺序，由此可以先将右子树压栈，然后再对左子树压栈，这样一来，左子树结点就存在了栈顶上，因此某结点的左子树能在它的右子树遍历之前被遍历。

```
public List<Integer> DFSwithStack(TreeNode root){
    Stack<TreeNode> stack = new Stack<>();
    List<Integer> results = new ArrayList<>();

    stack.push(root);
    while (!stack.empty()){
        TreeNode temp = stack.pop();
        results.add(temp.val);
        if(temp.right != null){
            stack.push(temp.right);
        }
        if(temp.left != null){
            stack.push(temp.left);
        }
    }
    return results;
}
```

7.完全二叉树

Count Complete Tree Nodes

给定一棵完全二叉树，求树中的节点个数

完全二叉树只有最后一层的节点可能是不满的，所以对于树中的任意一个节点来说，其左右子节点的高度只有两种情况：

1. 高度相等，说明该节点左子树是一个满二叉树，其节点个数为2^h-1，仍需对其右子树递归求解节点个数
2. 右节点高度比左节点高度小1，说明右节点是满二叉树，节点个数为$2^{h-1}-1$,仍需对其左子树递归求解节点个数

这里的高度计算的是到最左侧叶子节点的高度。

```
public class CountCompleteTreeNodes {
    //计算树高，root非空
    public int height(TreeNode root){
        if(root == null){
            return 0;
        }
        int height = 1;
        while(root.left != null){
            height++;
            root = root.left;
        }
        return height;
    }

    public int countNodes(TreeNode root) {
        if(root == null){
            return 0;
        }
        int count = 1;
        int leftheight = height(root.left);
        int rightheight = height(root.right);
        //如果左右节点高度一致
```



```
        if(leftheight == rightheight){
            count = count + (1<<leftheight) - 1;
            count += countNodes(root.right);
        }
        else {
            count = count + (1<<rightheight) - 1;
            count += countNodes(root.left);
        }
        return count;
    }
}
```

总结

1. 递归是深度优先搜索（DFS）的一种实现形式
   - 递归也可以用非递归方式实现
2. 二叉树上的递归
   - 遍历法
   - 分治法
3. 二叉搜索树
   - 性质：中序遍历是升序序列
   - 功能：O(h)的时间复杂度查找、删除、插入，h为BST高度
4. 二叉树上的宽度优先遍历
   - 利用队列实现宽度优先搜索
   - 如何实现分层遍历
5. 必背程序
   - 二叉树的前序、中序遍历的非递归实现
   - 二叉树的层序遍历
6. 二叉树的深度优先遍历DFS
   1. 递归实现
   2. 非递归实现（stack）

参考

猴子007的博客

二叉树

Bid Optimizing and Inventory Scoring in Targeted Online论文笔记 WAND算法