树

1.树的遍历

2.实现一个函数，判断二叉树是否对称。（如果一个二叉树和它的镜像一样，则它是对称的）

解题思路：定义一种针对前序遍历的对称遍历规则，先遍历右结点，在遍历左结点，然后判断前序遍历序列和对称遍历序列是否相同（但序列中需要包含null指针，即遍历到null时依然加入序列）

3.从上到下，从左到右分行打印二叉树

解题思路：利用队列保存将要打印的节点数，但需要设置两个变量，一个用来保存当前层中还没有打印的节点数，另一个变量用来表示下一层节点的数目，通过判断当前层中没有打印节点数是否为0从而换行。

4.之字形打印二叉树

解题思路：利用两个栈，一个栈用来保存奇数层节点（先保存右子结点再保存左子结点），一个栈用来保存偶数层结点（先保存左子结点再保存右子结点），设计一个标志位表示当前打印的是奇数层还是偶数层，每次根据标志位和相应层的栈是否为空来判断当前层是否打印结束进而进行换行打印。

5.二叉搜索树的后序遍历序列（判定给定数组是否是某二叉搜索树的后序遍历结果）

解题思路：递归解决，首先最后一个树为根节点值，从数组开始部分遍历小于它的都为左子树，大于它的都为右子树（但是在数组中遍历后半部分如果存在小于根节点值的则返回false）,如果数组存在左子树（即i>start），递归调用判读左子树，如果存在右子树（即i<end-1）,递归判断右子树，最后返回二者相与值

6.二叉树树中和为某一值的路径

解题思路：递归寻找，当前序遍历到某一节点时判断其是否是叶子节点，如果为叶子节点且路径中节点值的和等于当前整数，则将其打印出来，如果不是的话递归寻找它的左子结点和右子结点，如果子节点访问完毕则在当前路径上删除该节点。

7.序列化二叉树

解题思路：可以根据前序遍历来序列化二叉树，在碰到null指针时，这些null指针序列化为一个特殊的字符如“$”,另外节点的数值之间要用一个,隔开，例如“1,2,4,$,$,$,3,5,$,$,6,$,$”

反序列化时，读取的第一个节点为根节点，然后再递归的去处理左子树和右子树

8.二叉树的深度

解题思路：深度为左右子树深度大的值加一

```
public static int treeDep(Node head){
        if(head==null)
            return 0;
        int lH=treeDep(head.left);
        int rH=treeDep(head.right);
        return (lH>rH)?(lH+1):(rH+1);
    }
```

2.二叉树的下一个结点（给定一棵二叉树和其中一结点，找出中序遍历序列下一个结点，该二叉树的定义结构中存在parent指针）

解题思路：如果该结点存在右子树，下一结点就是它右子树的最左子结点

如果该结点不存在右子树，且是它父节点的左子结点，则下一结点是它父节点

如果该结点不存在右子树，且是它的右子结点，则就沿着父结点的指针一直向上遍历，直到找到某一结点是它父结点的左子结点，如果存在，下一结点是 其父结点，如果不存在，则无下一结点

```
public static Node getNext(Node node){
        if(node==null)
            return null;
        if(node.right!=null){
            Node cur=node.right;
            if(cur.left!=null)
                cur=cur.left;
            return cur;
        }else{
            Node par=node.parent;
            while(par!=null&&par.left!=node){
                node=par;
                par=par.parent;
            }
            return par;
        }
    }
```

3.找到两个结点的最近公共祖先

解题思路：后序遍历二叉树，遍历到某节点，如果它等于null或者o1、o2，则返回cur

否则返回左子树和右子树，若二者都为null,返回null,若二者都不为null，说明o1、o2在cur相遇，若一个为null，一个不为null,返回不为null的node

```
public static Node lowestAncestor(Node head,Node o1,Node o2){
        if(head==null||head==o1||head==o2)
            return head;
        Node left=lowestAncestor(head.left,o1,o2);
        Node right=lowestAncestor(head.right,o1,o2);
        if(left!=null&&right!=null)
            return head;
        return left!=null?left:right;
    }
```

4.树的子结构（输入两棵二叉树A和B，判断B是不是A的子结构）

解题思路：先在A中遍历到根B根结点相等的值，然后再判断A中子树结构与B是否相同（递归判断根结点，左子树和右子树是否对应相等），如果不相同，再依次递归调用左子树和右子树

```
public static boolean isSubtree(Node head1,Node head2){
        boolean result=false;
        if(head1!=null&&head2!=null){
            if(head1.value==head2.value)
                result=isEqual(head1,head2);
            if(!result)
                result=isSubtree(head1.left, head2);
            if(!result)
                result=isSubtree(head1.right, head2);
```

```
        }
        return result;
    }
    public static boolean isEqual(Node head1,Node
head2){
        if(head1==null)
            return false;
        if(head2==null)
            return true;
        if(head1.value!=head2.value)
            return false;
        return isEqual(head1.left,
head2.left)&&isEqual(head1.right, head2.right);
    }
```

5.判断一棵二叉树是否对称

解题思路：定义一种遍历规则：先序遍历的对称遍历规则（先遍历右子结点，再遍历左子结点），然后判断先序遍历所得结果与对称遍历结果是否相同，如若相同，则为对称二叉树

```
public static boolean isSym(Node head){
        return isSym(head,head);
    }
    public static boolean isSym(Node head1,Node
head2){
        if(head1==null&&head2==null)
```

```
        return true;
    if(head1==null||head2==null)
        return false;
    if(head1.value!=head2.value)
        return false;
    return isSym(head1.left,
head2.right)&&isSym(head1.right, head2.left);
}
```

二叉搜索树：

1）插入：

思路：如果当前结点是null，则创建新结点返回。如果插入结点比当前结点值大，则插入其右孩子结点中。如果插入结点比当前结点值小，则插入其左孩子结点中。 **复杂度**: 平均 O(logn) 最坏O(n)

实现：

```
1. public Tree insert(Tree root, int val) {

if (root == null) {
return new Tree(val);
}
if (val == root.val) {
return root;
} else if (val > root.val) {
root.right = insert(root.right, val);
} else {
root.left = insert(root.left, val);
}
return root;
```

}

2）查找：

思想:查找方式有点类型二分查找方法，知识这里采用的树结构进行存储。首先与根结点进行判断:

如果当前结点为null，则直接放回Null。

如果当前结点值与查找值相同则返回当前结点.

如果当前结点值小于查找值，则递归地到当前结点的右孩子查找

如果当前结点值大于查找值，则递归地到当前结点的左孩子查找。 **时间复杂度**:平均O(logn) 最坏O(n)

1. public Tree search(Tree root, int val) {

```
if(root == null) {
return null;
}
if(root.val == val) {
return root;
} else if(root.val > val) {
return search(root.left, val);
} else {
return search(root.right, val);
}
}
```

3）查找最小值

思想:根据二叉搜索树的特点，最小结点都是在最左结点上或者如果根结点无左孩子便是其本身.

```
public Tree min(Tree root) {
if(root == null) {
return null;
}
if (root.left != null) {
return min(root.left);
```

```
}
return root;
}
```

4）查找最大值

```
public Tree max(Tree root) {
if(root == null) {
return null;
}
if (root.right != null) {
return max(root.right);
}
return root;
}
```

5）删除某个结点

思想:要删除一个结点首先需要找到该结点的位置，采用上面的查找方式进行查找。找到结点后就是删除的问题的，可以按照下面的策略进行删除。如果一个结点无做孩子和右孩子，那么就可以直接删除，如果只存在一个孩子结点，则用孩子结点替换。如果存在两个孩子结点，那么可以用其左孩子最大的结点或右孩子最小结点替换，并删除最左孩子结点或最右孩子结点.

```
public Tree delete(Tree root, int val) {
if(root == null) {
return null;
}
if(root.val == val) {
if(root.left == null && root.right == null) {
return null;
} else if(root.left!= null && root.right != null) {
Tree leftBig = max(root.left);
root.val = leftBig.val;
root.left = delete(root.left, leftBig.val);
} else if(root.left != null){
return root.left;
} else {
```

```
return root.right;
}
} else if(root.val < val) {
root.right = delete(root.right, val);
} else {
root.left = delete(root.left, val);
}
return root;
}
```

6）删除最小结点

思想:找到根结点最左结点，如果其不存在右孩子则直接删除，否则用右孩子替换最左结点。需要注意的是根结点可能为null和不存在左孩子的情况。

```
public Tree deleteMin(Tree root) {
if(root == null) {
return null;
}
if(root.left != null) {
root.left = deleteMin(root.left);
} else if (root.right != null) {
return root.right;
}
return null;
}
```

7）删除最大结点

思路: 与删除最小结点类型，根据二叉搜索树的特性，最大结点是根结点的最右孩子。所以只要找到最右孩子结点，其存在左结点的话就用左结点替换否则直接删除.

```
public Tree deleteMax(Tree root) {
if(root == null) {
return null;
}
if(root.right != null) {
root.right = deleteMax(root.right);
} else if(root.left != null) {
```

```
return root.left;
}
return null;
}
```

8）根据后序数组重建搜索二叉树

a)给定整形数组，没有重复值，判断是否是搜索二叉树后序遍历的结果

找到数组中最后一个元素为根节点，第一个大于根节点的到之后为右子树，剩下前面的为左子树，然后再分别判断左子树和右子树满不满足搜索二叉树的条件，返回左子树和右子树的判断结果

```
public static boolean isPost(int[] arr,int start,int end){
    if(arr==null||arr.length==0){
        return false;
    }
    if(start==end)
        return true;
    int root=arr[end];
    int i=start;
    for(;i<end;i++){
        if(arr[i]>root)
            break;
    }
    int j=i;
    for(;j<end;j++){
        if(arr[j]<root)
            return false;
    }
```

```
        boolean left=true;
        //存在左子树
        if(i>0)
            left=isPost(arr,start,i-1);
        boolean right=true;
        //存在右子树
        if(i<end)
            right=isPost(arr, i, end-1);
        return (left&&right);


    }
```

b)通过搜索二叉树的后序遍历结果重构二叉树

```
public static Node posToBST(int[] posArr,int start,int
end){
        if(start>end)
            return null;
        BST T=new BST();
        BST.Node head=T.new Node(posArr[end]);
        int i=start;
        for(;i<end;i++){
            if(posArr[i]>posArr[end])
                break;
        }
```

```
        if(i>0)
            head.left=posToBST(posArr,start,i-1);
        if(i<end)
            head.right=posToBST(posArr, i, end-1);
        return head;

    }
```

c)判断是否是搜索二叉树

改写树的中序遍历，只是在遍历过程中判断是否满足条件

1）判断树是否是平衡二叉树

判断左子树是否是平衡二叉树，如果不是则退出，遍历过程中记录最深层次lh

判断右子树是否是平衡二叉树，如果不是则退出，遍历过程中记录最深层次rh

最后比较lh和rh的值判断整棵树是否是平衡二叉树

```
//判断是否是平衡二叉树(flag用来存储判断结果)
    public static boolean isBalance(Node head){
        boolean[] res=new boolean[1];
        res[0]=true;
        getHight(head, 1, res);
        return res[0];
    }
    public static int getHight(Node head,int
level,boolean[] flag){
        if(head==null){
            return level;
```

```
        }
        int lH=getHight(head.left, level+1, flag);
        //判断是否左子树是否为平衡二叉树
        if(!flag[0])
            return level;
        int rH=getHight(head.right, level+1, flag);
        //判断右子树是否是平衡二叉树
        if(!flag[0])
            return level;
        if(Math.abs(lH-rH)>1)
            flag[0]=false;
        return Math.max(lH, rH);

    }
```

链表

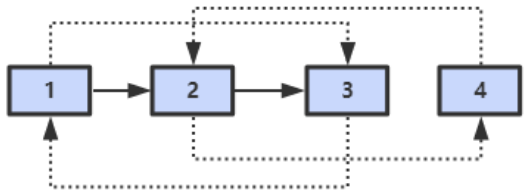
1.判断两个链表是否相交

解题思路：一种是hash计数法，保存第一个链表的所有节点的地址值，然后遍历第二个链表的每个节点的地址值判断其是否hash表中出现。

另一种是分别走到两个链表的尾部判断其是否相等，相等即相交。

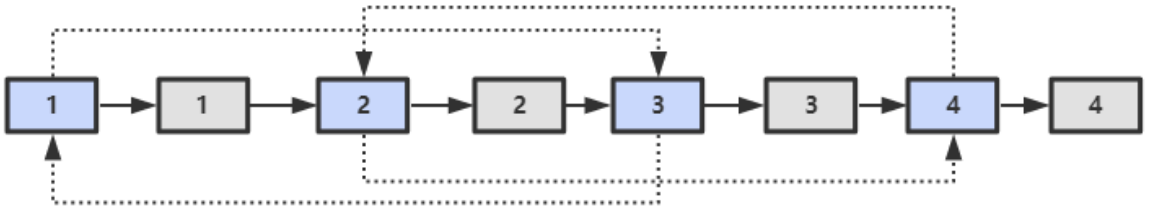
2.复杂链表的复制

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的head。

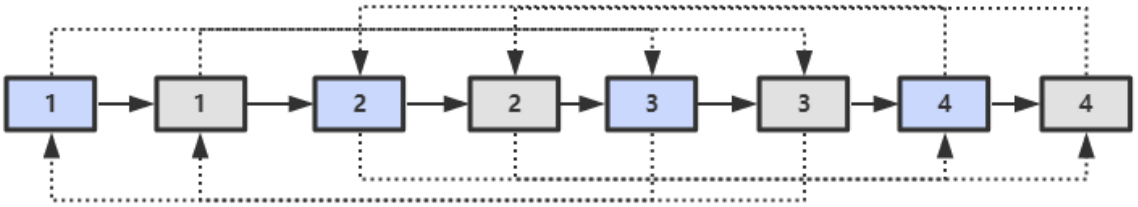


解题思路：

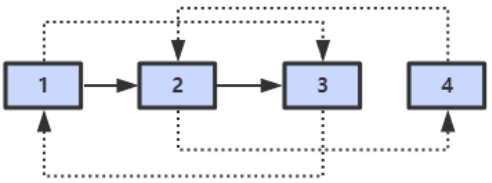
第一步，在每个节点的后面插入复制的节点。

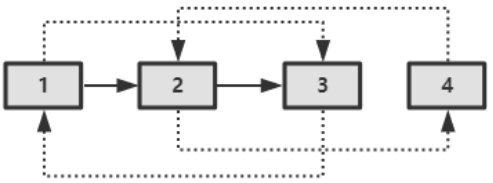


第二步，对复制节点的 random 链接进行赋值。



第三步，拆分。





3.二叉搜索树与双向链表

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

解题思路：left指针指向为指向前一个节点的前驱指正，right指针指向后一节点的后继指针，中序遍历整个二叉树，每次保存当前链表中最后一个节点作为前驱然后进行连接，最后通过最后一个节点找到链表头节点进行返回

数组：

1.在数组中找到出现次数超过一半的数(时间复杂度为O(n)，空间复杂度为O(1))

解题思路：设定候选值及其出现的次数，当其次数为0时设定新的候选值，核心思想是一次在数组中删除两个数，不停的删除，剩下的最后一个候选值就可能是超过一半的数，然后再遍历数组判断这个值的出现次数是否超过一半，超过就输出，没有超过的话就返回。

进阶问题是：在数组中找到出现次数超过N/K的数（时间复杂度为O(n*k)，空间复杂度为O(K)）

解题思路：核心思想是一次在数组中删除k个不同数，不停的删除，直到剩下数的种类不足k时就停止删除，如果存在一个数在数组中出现的次数大于N/K,那么这个数一定会被剩下来。

设定k个候选值的hashmap，如果当前数为map中的key，则将相应候选值的次数加1，如果不存在判断现在是否到达k个不同数（即map的大小为k-1）,到达的话就将相应候选值的value值都减一并且删除-1后value值为0的候选值，如果没有到达的话就把当前值放入做为候选值，数组遍历完成后针对map中所有候选值key，判断其出现次数是否大于n/k，大于就输出。