一。递归遍历

1.前中后序遍历

```
struct node {
    int data;
    struct node* left;
    struct node* right;
};
```

```
1.先序遍历
void preOrder(struct node* root)
{
    if (root == NULL)
        return;
    printf("%d ", data);
    preOrder(root->left);
    preOrder(root->right);
}

 2.中序遍历
 void inOrder(struct node* root)
 {
     if (root == NULL)
         return;
     inOrder(root->left);
    printf("%d ", data);
     inOrder(root->right);
 }
 3.后序遍历
 void postOrder(struct node* root)
 {
     if (root == NULL)
         return;
     postOrder(root->left);
     postOrder(root->right);
     printf("%d ", data);
 }
```

2.层次遍历

```
//层次遍历 C++版本，要借助队列
```

```
void levelorder(BTNode *root){
 if(root!=NULL){
 queue<BTNode*> q;//队列
 q.push(root);
 BTNode *p;
 while(!q.empty()){
 p=q.front();
 q.pop();
  printf("%c ",p->data);
  if(p->lchild!=NULL)
  q.push(p->lchild);
  if(p->rchild!=NULL)
  q.push(p->rchild);
  }
  }
 }
```

二。相关算法

1.递归统计叶子结点

```
int leaf_count(Tree T)
{
 if(T == NULL)
 return 0;
 if(T->lchild == NULL &&T->rchild == NULL)//叶子结点
 return 1;
 //递归统计左子树叶子结点和右子树叶子节点的个数
 return leaf_count(T->lchild)+leaf_count(T->rchild);
}
```

2.递归统计双分支结点

```
int branch_count(Tree T)
{
 if(T == NULL)
 return 0;

 if(T->lchild !=NULL &&T->rchild !=NULL)//双分支结点
 return 1;
 //递归统计左子树和右子树双分支结点的个数
 return branch_count(T->lchild)+branch_count(T->rchild);
 }

```

```
12
```

3.求二叉树的最大深度

```
int maxDeath(TreeNode node){
    if(node==NULL){
        return 0;
    }
    int left = maxDeath(node->left);
    int right = maxDeath(node->right);
    return Math.max(left,right) + 1;
}
```

4.求二叉树的最小深度

```
 int getMinDepth(TreeNode root){
        if(root == NULL){
            return 0;
        }
        return getMin(root);
    }
 int getMin(TreeNode root){
 if(root == NULL){
            return Integer.MAX_VALUE;
         }
         if(root.left == NULL&&root.right == NULL){
             return 1;
         }
         return Math.min(getMin(root.left),getMin(root.right)) + 1;
     }
```

5.求二叉树中节点的个数

```
int numOfTreeNode(TreeNode root){
        if(root == NULL){
            return 0;
        }
        int left = numOfTreeNode(root.left);
        int right = numOfTreeNode(root.right);
        return left + right + 1;
}
```

6.求二叉树中叶子节点的个数

```
int numsOfNoChildNode(TreeNode root){
 if(root == NULL){
 return 0;
```

```
}
if(root.left==NULL&&root.right==NULL){
return 1;
}
return numsOfNodeTreeNode(root.left)+numsOfNodeTreeNode(root.right);
}
```

7.求二叉树中第k层节点的个数

```
int numsOfkLevelTreeNode(TreeNode root,int k){
if(root == NULL || k<1 ){
return 0;
}
if(k==1){
return 1;
}
int numsLeft = numsOfkLevelTreeNode(root.left,k-1);
int numsRight = numsOfkLevelTreeNode(root.right,k-1);
 return numsLeft + numsRight;
}
```

8.判断二叉树是否是平衡二叉树

```
boolean isBalanced(TreeNode node){
        return maxDeath2(node)!=-1;
    }
    int maxDeath2(TreeNode node){
        if(node == NULL){
            return 0;
        }
        int left = maxDeath2(node.left);
        int right = maxDeath2(node.right);
         if(left==-1||right==-1||Math.abs(left-right)>1){
             return -1;
         }
         return Math.max(left, right) + 1;
}
```

9.两个二叉树是否完全相同

```
boolean isSameTreeNode(TreeNode t1,TreeNode t2){
        if(t1==NULL&&t2==NULL){
            return true;
        }
```

```
        else if(t1==NULL||t2==NULL){
            return false;
        }
        if(t1.val != t2.val){
            return false;
         }
         boolean left = isSameTreeNode(t1.left,t2.left);
         boolean right = isSameTreeNode(t1.right,t2.right);
         return left&&right;
    }
```

10.两个二叉树是否互为镜像

```
boolean isMirror(TreeNode t1,TreeNode t2){
        if(t1==NULL&&t2==NULL){
            return true;
        }
        if(t1==NULL||t2==NULL){
            return false;
        }
        if(t1.val != t2.val){
            return false;
         }
         return isMirror(t1.left,t2.right)&&isMirror(t1.right,t2.left);
}
```

11.翻转二叉树or镜像二叉树

```
TreeNode mirrorTreeNode(TreeNode root){
        if(root == NULL){
            return NULL;
        }
        TreeNode left = mirrorTreeNode(root.left);
        TreeNode right = mirrorTreeNode(root.right);
        root.left = right;
        root.right = left;
        return root;
}
```

12.求两个二叉树的最低公共祖先节点

```
 TreeNode getLastCommonParent(TreeNode root,TreeNode t1,TreeNode t2){
        if(findNode(root.left,t1)){
```

```
            if(findNode(root.right,t2)){
                return root;
            }else{
                return getLastCommonParent(root.left,t1,t2);
            }
        }else{
            if(findNode(root.left,t2)){
                 return root;
             }else{
                 return getLastCommonParent(root.right,t1,t2)
             }
         }
    }
    // 查找节点node是否在当前 二叉树中
    boolean findNode(TreeNode root,TreeNode node){
        if(root == NULL|| node == NULL){
            return false;
        }
        if(root == node){
            return true;
        }
        boolean found = findNode(root.left,node);
        if(!found){
            found = findNode(root.right,node);
        }
        return found;
    }
```

13.在二叉树中插入节点

```
reeNode insertNode(TreeNode root,TreeNode node){
 if(root == node){
 return node;
 }
 TreeNode tmp = new TreeNode();
 tmp = root;
 TreeNode last = NULL;
 while(tmp!=NULL){
 last = tmp;
  if(tmp.val>node.val){
  tmp = tmp.left;
  }else{
```

```
 tmp = tmp.right;
 }
 }
 if(last!=NULL){
 if(last.val>node.val){
 last.left = node;
 }else{
 last.right = node;
 }
 }
 return root;
}
```

三。选看

14.判断二叉树是否是合法的二叉查找树(BST)

原文链接：https://www.cnblogs.com/Angel-Demon/p/10244420.html

> 一棵BST定义为：
> 节点的左子树中的值要严格小于该节点的值。
> 节点的右子树中的值要严格大于该节点的值。
> 左右子树也必须是二叉查找树。
> 一个节点的树也是二叉查找树。

```
struct BinaryTree {
 int value;
 BinaryTree* lson;
 BinaryTree* rson;
};
int maxOf(BinaryTree* root) {
 if (root->rson) {
 return maxOf(root->rson);
 }
 return root->value;
}
int minOf(BinaryTree* root) {
 if (root->lson) {
 return maxOf(root->lson);
 }
 return root->value;
}
```

```
bool check(BinaryTree* root) {
 if (root == NULL) {
 return true;
 }
 // 如果左子树的最大值小于当前节点值并且右子树的最小值大于当前节点的值并且左右子树都是二叉搜索树，则返回true，否则返回false;
 if (root->lson != NULL && maxOf(root->lson) >= root->value) {
 return false;
 }
 if (root->rson != NULL && minOf(root->rson) <= root->value) {
 return false;
 }
 return check(root->lson) && check(root->rson);
}
```

15.给出 n，问由 1...n 为节点组成的不同的二叉查找树有多少种？（不同的二叉树）

原文链接：https://blog.csdn.net/weixin_40673608/article/details/86533230

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

16.二叉树内两个节点的最长距离

原文链接：https://blog.csdn.net/liuyi1207164339/article/details/50898902

二叉树中两个节点的最长距离可能有三种情况：

1.左子树的最大深度+右子树的最大深度为二叉树的最长距离

2.左子树中的最长距离即为二叉树的最长距离

3.右子树种的最长距离即为二叉树的最长距离

因此，递归求解即可

```
struct NODE
{
```

```
 NODE* pLeft; // 左子树
 NODE* pRight; // 右子树
 int nMaxLeft; // 左子树中的最长距离
 int nMaxRight; // 右子树中的最长距离
 char chValue; // 该节点的值
};

int nMaxLen = 0;

// 寻找树中最长的两段距离
void FindMaxLen(NODE* pRoot)
{
 // 遍历到叶子节点，返回
 if (pRoot == NULL)
 {
 return;
 }

 // 如果左子树为空，那么该节点的左边最长距离为0
 if (pRoot->pLeft == NULL)
 {
 pRoot->nMaxLeft = 0;
 }

 // 如果右子树为空，那么该节点的右边最长距离为0
 if (pRoot->pRight == NULL)
 {
 pRoot->nMaxRight = 0;
 }

 // 如果左子树不为空，递归寻找左子树最长距离
 if (pRoot->pLeft != NULL)
 {
 FindMaxLen(pRoot->pLeft);
 }

 // 如果右子树不为空，递归寻找右子树最长距离
 if (pRoot->pRight != NULL)
 {
 FindMaxLen(pRoot->pRight);
 }
```

```
44
45   // 计算左子树最长节点距离
46   if (pRoot->pLeft != NULL)
47   {
48   pRoot->nMaxLeft = ((pRoot->pLeft->nMaxLeft > pRoot->pLeft->nMaxRight) ?
pRoot->pLeft->nMaxLeft : pRoot->pLeft->nMaxRight) + 1;
49   }
50
51   // 计算右子树最长节点距离
52   if (pRoot->pRight != NULL)
53   {
54   pRoot->nMaxRight = ((pRoot->pRight->nMaxLeft > pRoot->pRight-
>nMaxRight) ? pRoot->pRight->nMaxLeft : pRoot->pRight->nMaxRight)+1;
55   }
56
57   // 更新最长距离
58   if (pRoot->nMaxLeft + pRoot->nMaxRight > nMaxLen)
59   {
60   nMaxLen = pRoot->nMaxLeft + pRoot->nMaxRight;
61   }
62 }
63
```

17.判断二叉树是否是完全二叉树

//原文链接：https://blog.csdn.net/qq_40938077/article/details/80471997

```
1   struct node
2 {
3   node left;
4   node right;
5   int value;
6 };
7 bool isCBT(node head)//判断以head为头节点的二叉树是否为完全二叉树
8 {
9   if(head==null)
10   return true;
11   bool leaf=false;//leaf变量用来标记一个状态是否发生（只要当前节点的左孩子和右
孩子都为空或者左孩子不为空，右孩子为空时，这个状态就发生，只要发生了这个状态，以后
访问到的节点必须都是叶节点）
12   queue < node >q;//通过队列q实现二叉树的层次遍历，通过层次遍历来判断是否为完全
二叉树
13   q.push(head);//加入头节点
```

```
14   while (!!q.empty())
15   {
16   node p=q.front();
17   q.pop();
18   if((leaf&&(p.left!=null||p.right!=null))||
(p.left==null&&p.right!=null))//这些判断条件是所有的不满足是完全二叉树的条件。条
件一（第二个||前面的条件）：上述的状态已经发生，但是当前访问到的节点不是叶节点（有
左孩子或者右孩子）。条件二：当前节点有右孩子，没有左孩子
19   return false ;
20   if(p.left!=null)//左孩子不为空，加入到队列中去
21   q.push(p.left);
22   if(p.right!=null)//右孩子不为空，加入到队列中去
23   q.push(p.right);
24   if((p.left!=null&&p.right==null)||(p.left==null&&p.right==null))//这个if
语句就是判断状态是否要发生
25   leaf=true;
26   }
27   return true;
28  }
```