@[TOC]

树算法总结（红皮书+王道）

遍历算法

先序遍历

从上往下找，也适用于一些算法

递归算法：

```
void PreOrder(tree root){
    if(root!=NULL){
        printf("%d ",root->data);
        PreOrder(root->left);
        PreOrder(root->right);
    }
}
```

非递归算法1：（类似于深度的非递归）

```
void PreOrder_2(tree root){
    tree stack[maxsize];
    int top = -1;
    tree p = root;

    while(p || top != -1){
        if(p){
            printf("%d ", p->data);      //一直遍历左子树
            stack[++top] =  p;
            p = p->left;
        }
        else{                            //左子树不存在遍历右子树
            p = stack[top--];
            p = p->right;
        }
    }
}
```

非递归算法2:(类似于层次的非递归)

```
void PreOrder_3(tree root){
    tree stack[maxsize];
    int top = -1;
    tree p = root;
    stack[++top] = p;
    while(p || top != -1){
        p = stack[top--];
        printf("%d ", p->data);
        if(p->right) stack[++top] = p->right;        //先访问右子树，入栈，出栈的顺序就是左子树、然后右子树
        if(p->left)  stack[++top] = p->left;
    }
}
```

应用1 删除以x为根的子树，不包含节点本身（从上往下找）

思路： 从上往下找，找到节点值为x的，然后删除其左右子树

代码：

```
void Release(tree root){//释放以root为根的所有节点
    if(root){
        Release(root->left);
        Release(root->right);
        free(root);
    }
}

void Del_X(tree root, int x){//先序递归遍历
    if(root == NULL) return;
    if(root->data == x){
            Release(root->left);              //删除其节点的子树
            Release(root->right);
            root->left = NULL;           //不要忘了加后面的指针为空
            root->right = NULL;
            return;
        }
    else{
        Del_X(root->left,x);
        Del_X(root->right,x);
    }
}
```

应用2 判断二叉树是否相似，相似是指T1,T2都是空的二叉树或者都只有一个节点（从上往下比较）

思路： 利用先序遍历的递归实现，从上往下比较

代码：

```
int IsSimilar(tree t1, tree t2){
    if(t1==NULL && t2 == NULL) return 1;            //都不空，相似
    else if(t1==NULL || t2 == NULL) return 0;       //有一个为空，不相似
    else{
        int left =  IsSimilar(t1->left,t2->left);
        int right = IsSimilar(t1->right,t2->right);
        return left && right;
    }
}
```

应用3　判断二叉树是否对称（自上而下判断）

思路：利用先序遍历的递归实现，从上往下比较

代码：

```
int IsSymmetry(tree t1, tree t2){
    if(t1==NULL && t2 == NULL) return true;             //都为空，对称
    else if(t1==NULL || t2==NULL) return false;         //一个为空，不对称
    else if(t1->data != t2->data) return false;         //都不空，但是值不同，不对称
    else{                                               //左右节点都不为空，继续递归判
断对称
        return  IsSymmetry(t1->left,t2->right) && IsSymmetry(t1->right, t2-
>left);
    }       //这里需要注意对称是指最左和最右对称，里面一层的左右子树对称，注意递归的里面的写法
}

int JudgeSymmetry(tree t1){
    if(t1==NULL) return 1;
    return IsSymmetry(t1->left, t1->right);
}
```

应用4 判断二叉树是否存在平衡节点（自上而下判断）

若二叉树中的某个节点其左子树所有节点之和与右子树节点之和相等，称为平衡节点

思路：两个递归，一个计算节点值之和，一个判断是否有平衡节点

代码：

```
int ChildSum(tree root){        //计算孩子节点和
    if(root == NULL) return 0;
    return ChildSum(root->left)+ChildSum(root->right)+root->data;
}
int IsHaveAVLNode(tree root){   //判断是否存在平衡节点
    if(root==NULL) return false;        //如果为空，返回0
    if(root->left != NULL && root->right!= NULL){       //返回true的条件是，左右子树
不为空且结点值相同
        if(ChildSum(root->left) == ChildSum(root->right))
            return true;
    }
    return IsHaveAVLNode(root->left) || IsHaveAVLNode(root->right);
}
```

应用5 利用二叉树序列构造树（题目所给的条件就是自上而下构建）

一颗以r为根的二叉树可以用以下规则表示成一个由0,1,2组成的字符序列，称之为"二叉树序列S"：
S=0，如果r没有子结点
S=1S1，如果r有一个子结点，S1为其子树的二叉树序列
S=2S1S2，如果r有两个子结点，S1和S2分别表示其两个子树的二叉树序列。

思路：题目所给的条件就是自上而下构建

代码：

```
tree PreCreateTree(int A[], int n){
    if(n == 0) return 0;
    struct tnode *root = (struct tnode *)malloc(sizeof(struct tnode));
    tnode *p = root;
    tree stack[maxsize];
    int top = -1, i = 0;

    while(p != NULL || top != -1){
        if(p != NULL){
            if(A[i] == 0){      //一直创建p节点的左右孩子，直到p为空
                p->left = NULL;
                p->right = NULL;
            }
            else if(A[i] == 1){
                struct tnode *temp = (struct tnode *)malloc(sizeof(struct
tnode));
                p->left = temp;
                p->right = NULL;
            }
            else{
                struct tnode *temp1 = (struct tnode *)malloc(sizeof(struct
tnode));
                struct tnode *temp2 = (struct tnode *)malloc(sizeof(struct
tnode));
                p->left = temp1;
                p->right = temp2;
            }
            stack[++top] = p;       //入栈并访问左孩子
            p = p->left;
            i++;
        }
        else{
            p = stack[top--];           //当前p为空时，说明其根节点已经被访问过，现在访问
其右子树节点
            p = p->right;
        }
    }
    return root;
}
```

中序遍历

递归算法

```
void InOrder(tree root){
    if(root!=NULL){
        InOrder(root->left);
        printf("%d ",root->data);
        InOrder(root->right);
    }
}
```

非递归算法

```
void InOrder_2(tree root){
    tree stack[maxsize];
    int top = -1;
    tree p = root;
    while(p || top != -1){
        if(p){
            stack[++top] = p;                    //左子树入栈，先不访问
            p = p->left;
        }
        else{
            p = stack[top--];                    //出栈后，再访问，访问完转右子树
            printf("%d ", p->data);
            p = p->right;
        }
    }
}
```

应用1 给定的表达式树，转化为等价的中缀表达式输出

思路： 中序遍历的结果就是中缀表达式，注意加括号

```
void BtreeToExp(tree bt, int deep){
    if(bt == NULL) return;
    else if(bt ->left == NULL && bt->right == NULL){
        printf("%d",bt->data);
    }
    else{
        if(deep>1) printf("(");
        BtreeToExp(bt->left, deep+1);
        printf("%d",bt->data);
        BtreeToExp(bt->right,deep+1);
        if(deep>1) printf(")");
    }
}
```

后序遍历

很重要，因为它是从先访问了左右子树然后根节点往上遍历的一个典型代表，可以运用到很多方面

递归算法

```
void PostOrder(tree root){
    if(root!=NULL){
        PostOrder(root->left);
        PostOrder(root->right);
        printf("%d ",root->data);
    }
}
```

非递归算法1

```
void PostOrder_2(tree root){
    tree stack[maxsize];
    int top = -1;
    tree p = root,r;                                //添加标志位r，判断右子树是否访问

    while(p || top!=-1){
        if(p){
            stack[++top] = p;                       //左子树入栈
            p = p->left;
        }
        else{
            p = stack[top];                         //先取栈顶，不出栈
            if(p->right && p->right != r){
                p = p->right;                       //右子树未被访问，那先访问右子树
            }
            else{                                   //访问过了，那就出栈
                top--;
                printf("%d ",p->data);              //访问
                r = p;                              //标记前一个节点
                p = NULL;                           //已经访问过，置空
            }
        }
    }
}
```

非递归算法2（基于先序遍历）

思路：先序遍历：根左右 ===> 先访问右子树，再访问左子树，得到 根右左 ===> 入栈，再出栈，得到左右根（后序遍历结果）

```
void PostOrder_3(tree root){          //先序遍历的改装，两个栈
    tree stack1[maxsize];             //栈1用来先序遍历的非递归遍历
    tree stack2[maxsize];             //栈2保存后序遍历的结果
    int top1 = -1, top2 = -1;
    tree p = root;

    while(top1!=-1 || p){
        if(p){
```

```
            stack1[++top1] = p;
            stack2[++top2] = p;
            p = p->right;           //先访问右子树
        }
        else{
            p = stack1[top1--];
            p = p->left;            //再访问左子树
        }
    }
    while(top2!=-1){
        printf("%d ", stack2[top2--]->data);
    }
}
```

应用1 计算双分支节点个数（从下往上找）

思路： 递归遍历，如果当前节点为双分支节点，返回左子树双分支节点和右子树双分支节点个数+1，否则不加1

代码：

```
int DsonNodes(tree root){
    if(root == NULL) return 0;
    if(root->left != NULL && root->right != NULL){//双分支节点
        return DsonNodes(root->left)+DsonNodes(root->right)+1;
    }
    else{//不是双分支节点
        return DsonNodes(root->left)+DsonNodes(root->right);
    }
}
```

应用2 交换所有左右子树 （从下往上交换）

思路： 递归遍历，到下一层为根节点的时候，交换左右子树

代码：

```
void SwapTree(tree root){
    if(root){           //从下往上交换
        SwapTree(root->left);
        SwapTree(root->right);
        tree t = root->left;
        root->left = root->right;
        root->left = t;
    }
}
```

应用3 利用中序和先序序列构建二叉树,二叉树节点值各不相同 （从下往上建立）

思路： 找到根节点位置，划分左右子树，然后递归建立左右子树，注意下标对应关系

代码：

```
tree PreInCreate(int pre[], int in[], int f1, int e1, int f2, int e2){
```

```
    //f1 e1 为先序的起始和终止位置 f2 e2 为中序的起始和终止位置
    tree root =  (struct tnode *)malloc(sizeof(struct tnode));
    root->data = pre[f1];
    int pos=f2;
    while(in[pos] != pre[f1]) pos++;             //找到根节点位置
    int llen = pos - f2;                         //左子树长度
    int rlen = e2 - pos;                         //右子树长度
    if(llen){                                    //左子树不为空，递归建立左子树
        root->left = PreInCreate(pre,in,f1+1,f1+llen,f2,f2+llen-1);
    }
    else{                                        //左子树为空
        root->left = NULL;
    }
    if(rlen){                                    //右子树不为空，递归建立左子树
        root->right = PreInCreate(pre,in,e1-rlen+1,e1,e2-rlen+1,e2);
    }
    else{                                        //右子树为空
        root->right = NULL;
    }
    return root;                                 //返回结果
}
```

应用4 查找节点值x的祖先节点

思路： 后序非递归遍历查找到x节点时，栈中保存的节点就是x的祖先节点

代码：

```
void SearchAncestor(tree root, int x){
    tree stack[maxsize];
    int top = -1;
    tree p = root,r=NULL;
    while(p || top != -1){
        if(p){
            stack[++top] = p;
            p = p->left;
        }
        else{
            p = stack[top];
            if(p->right && p->right != r){
                p = p->right;
            }
            else{
                if(p->data == x) break;      //找到了
                top--;
                r = p;
                p = NULL;
            }
        }
    }
    printf("%d的祖先节点有：",x);
    while(top!=-1){
        printf("%d ", stack[top--]->data);
    }
}
```

应用5 查找节点p和q的公共祖先

递归实现

思路：

主要是多种递归情况的考虑：
递归结束标志：
1、如果root为空，直接返回root
2、如果root为p或者q找到了其中一个节点的祖先节点，返回root

返回值：
1、左右子树均不为空，说明p和q分布在root的两侧且均找到了，返回root
2、左右子树其中一个为空，说明那个空的树既没有找到p，也没有找到q，所以不用继续找了，直接返回另外一个值

代码：

```
void SearchPQAncestor(tree root, tree p, tree q){
    if(!root || root == p || root == q) return root;     //为空，或找到一个都返回其本身
    tree left = SearchPQAncestor(root->left,p,q);
    tree right = SearchPQAncestor(root->right,p,q);
    if(left == NULL) return right;                       //左子树没找到，右子树继续找
    if(right == NULL) return left;                       //右子树没找到，左子树找
    return root;                                         //左右子树都找到了，那说明肯定找到了
}
```

非递归实现

思路：后序非递归遍历

代码：

```
tree SearchPQAncestor_2(tree root, tree p, tree q){
    tree stack1[maxsize];
    tree stack2[maxsize];
    int top1 = -1, top2 = -1;
    tree m = root,r;
    while(top1 != -1 || m){
        if(m){
            stack1[++top1] = m;
            m = m->left;
        }
        else{
            m = stack1[top1];
            if(m->right && m->right != r){       //未被访问过，转为右子树，继续访问
                m = m->right;
            }
            else{//栈中元素已经访问过，出栈
                //这个是基于p在q的左侧的前提下，有缺陷，暂时先这样了，可能找的不是最近的公共祖先节点
                if(stack1[top1] == p){                   //如果栈顶为p，那么栈中元素全部为p的祖先
                    for(int i = 0; i <= top1; i++){
                        stack2[i] = stack1[i];           //栈2保存p的祖先节点
```

```
                    }
                    top2 = top1;
                }
                if(stack2[top1] == q){                    //如果栈2中找到了q，那么栈2中全部元素为q的祖先，因为栈2是由栈1过来的，找到，那么必然有pq的公共祖先在栈中
                    for(int i = top2; i >= 0; i--){
                        for(int j = top1; j >= 0; j--){
                            if(stack2[i] == stack1[j]){
                                return stack2[i];
                            }
                        }
                    }
                }
                top1--;                                   //继续找
                r = m;
                m = NULL;
            }
        }
    }
}
```

应用6 将二叉树的叶节点从左往右连成一个单链表，表头指针为head，链接是用叶节点的右指针域存放单链表指针

思路： 后序递归算法改写，递归出口如果是叶子节点，接链，不是就继续遍历，最后返回head

代码：

```
tree p = NULL,head;
tree PreOrderLeafList(tree root){
    if(root->left == NULL && root->right == NULL){//是叶子节点
        if(p == NULL){      //如果是第一个节点
            head = root;
            p = head;
        }
        else{
            p->right = root;    //尾插法
            p = p->right;

        }
    }
    PreOrderLeafList(root->left);
    PreOrderLeafList(root->right);
    return head;                //后序递归遍历实现
}
```

应用7 一个根为t的节点，后序遍历找到这颗树的最后一个节点

思路： 后序遍历的最后一个叶子结点，就是最后一个节点

代码：

```
void FindLastNode(tree root, tree &last){
    if(root){
        FindLastNode(root->left,last);
        FindLastNode(root->right,last);
        if(root->left==NULL && root->right==NULL){
            last = root;
        }
    }
}
```

层次遍历

基础算法

思路： 队列，不断的出队，入队访问

代码：

```
void BFS_Tree(tree root){
    tree queue[maxsize];
    int front = -1, rear = -1;
    tree q;
    queue[++rear] = root;
    while(rear != front){
        q = queue[++front];
        printf("%d ",q->data);
        if(q->left) queue[++rear] = q->left;
        if(q->right) queue[++rear] = q->right;
    }
}
```

应用1 从下而上，从左而右的层次遍历

二叉树自下而上，从右往左的层次遍历算法 这个比较简单，直接层次遍历入栈，然后出栈便得到了
变形： 二叉树自下而上，自左而右的层次遍历算法

思路： **按层遍历**，是利用两个栈，**每一层的节点入栈后，然后出栈入另外一个栈2**，栈2最后出栈便是最后的结果

代码：

```
void InvertLevel_2(tree root){
    tree stack1[maxsize];           //栈1保存每一层的节点
    tree stack2[maxsize];           //栈2保存最后的结果
    int top1 = -1, top2  = -1;
    tree queue[maxsize];            //队列用于层次遍历
    int front = -1, rear = -1;

    tree p = root;
    queue[++rear] = p;
    while(rear != front){
        int n = rear - front;           //每一层的个数
        for(int i = 0; i < n; i++){     //先访问每一层的节点
```

```
            p = queue[++front];
            stack1[++top1] = p;          //每一层的节点入栈1
            if(p->left) queue[++rear] = p->left;
            if(p->right) queue[++rear] = p->right;
        }
        while(top1!=-1){                 //栈2得到从右到左的每一层节点个数，出栈便是从左到右
            stack2[++top2] = stack1[top1--];
        }
    }
    while(top2!=-1){
        printf("%d ",stack2[top2--]->data);
    }
}
```

应用2 层次遍历判断是否是完全二叉树

思路： 层次遍历，无需判断左子树还是右子树是否为空，直接入队列，然后不断出队，出队元素如果是空，继续判断后序节点是否为空，出现了不为空，就不是完全二叉树

代码：

```
bool IsComlepteTree(tree root){
    tree queue[maxsize];
    int front = -1, rear = -1;
    tree p = root;
    queue[++rear] = p;
    while(rear != front){
        p = queue[++front];
        if(p){           //不为空，则将左右节点入队列
            queue[++rear] = p->left;
            queue[++rear] = p->right;
        }
        else{            //为空，判断后序节点
            while(rear!=front){
                p = queue[++front];
                if(p){  //存在不为空的节点，说明不是完全二叉树
                    return 0;
                }
            }
        }
    }
    return 1;
}
```

应用3 层次遍历求树的宽度

```
int WidthTree(tree root){//按层的层比遍历，统计每层节点个数
    tree queue[maxsize];
    int front = -1, rear = -1;
    tree p = root;
    int maxwidth = 0,width;
    queue[++rear] = p;
    while(rear != front){
        int n = rear - front;
```

```
        if(n > maxwidth) maxwidth = n;
        for(int i =0; i < n; i++){
            p = queue[++front];
            if(p->left) queue[++rear] = p->left;
            if(p->right) queue[++rear] = p->right;
        }
    }
    return maxwidth;
}
```

应用4 统计二叉树各层独生叶节点（既是叶节点又无兄弟节点的数目）

要求：编写函数LeafBrotherInEachLevel(root)，输出以root为根的二叉树**各层**的独生叶节点的数目

思路： 这里需要注意是输出各层的节点个数，而不只是单纯的统计。因为是各层，所以很容易想到树的**层次遍历算法**
具体操作：

1. 层边遍历采用**每一层每一层的遍历**，先统计每层的个数，然后直接一层遍历，而不是单个节点进行遍历
2. 在判断了左孩子为空右孩子不为空的时候，还得**继续判断左孩子的左孩子，与左孩子的右孩子都为空**，才能确定这是一个独生叶节点

代码：

```
void LeafBrotherInEachLevel(tree root){
    //层次遍历
    if(root == NULL){
        printf("0 层 0 个独生节点\n");
        return;
    }
    if(root->left== NULL && root->right == NULL){
        printf("0 层 1 个独生节点\n");
        return;
    }
    printf("0 层 0 个独生节点\n");

    tree queue[maxsize];
    int front = -1, rear = -1;
    int level = 1;
    queue[++rear] = root;

    while(front != rear){
        int n = rear - front;   //表示该层的节点个数

        int count = 0;          //当前层独生节点个数
        for(int i = 0; i < n; i++){
            tnode *p = queue[++front];
            if(p->left != NULL && p->right == NULL){//左孩子为独生节点
                if(p->left->left == NULL && p->left->right == NULL)
                    count++;
            }
            if(p->left == NULL && p->right != NULL){//右孩子为独生节点
                if(p->right->left == NULL && p->right->right == NULL)
                    count++;
```

```
            }
            if(p->left != NULL){//左右孩子入队列
                queue[++rear] = p->left;
            }
            if(p->right != NULL){//左右孩子入队列
                queue[++rear] = p->right;
            }
        }
        if(rear != front){
            printf("%d 层 %d 个独生节点\n",level++, count);
        }
    }
}
```

求树高

递归求树高

代码：

```
int HighTree(tree root){//后序遍历得到树高
    if(root == NULL) return 0;
    int l = HighTree(root->left);
    int r = HighTree(root->right);
    return l > r ? l+1 : r+1;
}
```

非递归求树高

思路：按层遍历的层次遍历

代码：

```
int HighTree_2(tree root){
    tree queue[maxsize];
    int front = -1, rear = -1;
    tree p = root;
    int level = 0;
    queue[++rear] = p;
    while(rear != front){
        int n = rear - front;               //每层节点个数
        level++;
        for(int i =0; i < n; i++){          //遍历
            p = queue[++front];
            if(p->left) queue[++rear] = p->left;
            if(p->right) queue[++rear] = p->right;
        }
    }
    return level;
}
```

应用1 判断是否是平衡二叉树

思路：求树高的过程中，判断是否为完全二叉树，赋值为-1保存结果

```
int AVLHigh(tree root){
    if(root == NULL) return 0;
    int l = HighTree(root->left);
    int r = HighTree(root->right);
    if(l == -1 || r == -1 || abs(l-r)>1){   //不是完全二叉树
        return -1;
    }
    return l > r ? l+1 : r+1;
}

bool Is_AVL(tree root){
    if(AVLHigh(root)>=0) return true;
    else return false;
}
```

二叉查找树

应用1 判断一棵树是否为二叉查找树

思路： 利用二叉树的中序遍历为一个递增序列，全局变量保存前一个节点，利用中序递归遍历，进行比较

代码：

```
int pre1 = -10000;
bool IsBSTree(tree root){
    //利用二叉排序树的中序遍历为一个递增序列
    int b1,b2;
    if(root == NULL){
        return 1;
    }
    else{
        int l = IsBSTree(root->left);            //判断左子树
        if(l == 0 || pre1 >= root->data){
            return 0;                            //如果左子树为空，或者左子树的值大于当前值，不满足二叉树排序树性质
        }
        pre1 = root->data;                       //保存中序遍历当前结点值，用于后序比较
        int r = IsBSTree(root->right);       //判断右子树
        return r;
    }
}
```

应用2 编写算法实现删除二叉查找数中的指定节点

使得该节点删除后，树仍然满足二叉查找数的性质，并画出删除节点p之后的结果

思路:
算法实现
在二叉排序树的查找递归中实现，先实现两个函数，找到直接前驱与后继
如果删除的节点为叶子节点，直接删除，不影响二叉排序树的性质
如果删除的节点存在右子树，直接找右子树的最左下节点进行替换（直接后继）
如果删除的节点无右子树，直接找左子树的最右下节点进行替换（直接前驱）

代码:

```
int NextSuccessor(tree root){        //最左下节点 后继节点
    while(root->left != NULL){
        root = root->left;
    }
    return root->data;
}

int PreSuccessor(tree root){    //最右下节点 前驱节点
    while(root->right){
        root = root->right;
    }
    return root->data;
}


tree DelFindTreeNode(tree root, int v){//递归实现
    if(root == NULL) return NULL;
    else if(root->data > v) root->left  = DelFindTreeNode(root->left,v);
    else if(root->data < v) root->right = DelFindTreeNode(root->right,v);
    else{
        if(root->left == NULL && root->right == NULL)   //叶子节点
            root = NULL;
        else if(root->right != NULL){   //存在右子树
            root->data  = NextSuccessor(root->right);
            root->right = DelFindTreeNode(root->right, root->data);
        }
        else{                           //存在左子树
            root->data  = NextSuccessor(root->left);
            root->left  = DelFindTreeNode(root->left, root->data);
        }
    }
    return root;
}
```

应用3 二叉查找树，关键字均不同，编写非递归算法，按递减次序打印所有左子树非空，右子树为空的节点的关键字

思路: 二叉排序树的性质：**中序遍历下是递增序列**，因为是要求递减，而且只需要判断右子树为空的情况，那么只需要对中序遍历做一个变形，**从先访问左子树变为先访问右子树即可得到递减序列**

代码:

```
//非递归中序遍历的实现
void OutDFindTree(tree root){
    int top = -1;
    tree stack[maxsize];
    tnode *p = root;
    while(p != NULL || top != -1){
        if(p){
            stack[++top] = p;
            p = p->right;                //这里先访问右子树
        }
        else{
            p = stack[top--];
            if(p->left != NULL && p->right == NULL){
                printf("%d ", p->data);
            }
            p = p->left;             //然后再访问左子树
        }
    }
}
```

应用4 求二叉树第k小元素，关键字（n>=k>=1），要求时间复杂度不超过logn

思路:

- 注意到题目已经给了每个树节点多了一个孩子节点的个数
- 这里只需要判断三种情况
  - 左子树个数刚好是k-1个，那么根节点就是要找的节点
  - 左子树个数大于k个，那么继续在左子树找第k小元素
  - 左子树个数小于k-1个，那么需要在右子树找，第 k -（左子树+根节点个数）大的节点

代码:

```
int Search_ksmall(node *root, int k){
    if(root == NULL) return 0;               //递归出口
    int leftnum = root->left->n;             //得到以root为根节点的左子树孩子节点个数
    if(k == leftnum + 1) return root->data; //如果左子树个数为k-1，直接返回根节点
    else if(leftnum >= k) return Search_ksmall(root->left,k); //左子树个数大于k，继续在左子树中找
    else return Search_ksmall(root->right, k-leftnum-1);      //在右子树中找


}//算法的时间复杂度为log2n
```

```
## 其他算法

### 1、找按顺序存储的完全二叉树，节点为i和j的公共祖先

**思路：** 子节点与父节点关系 i = i/2;为父节点下标，注意下标是以1开头

**代码：**
```

```c
int Common_Ancestor(int a[], int i, int j){//数组下标从1开始存储
    if(a[i] && a[j]){
        while(i != j){
            if(i > j){
                i = i/2;
            }
            else{
                j = j/2;
            }
        }
    }
    printf("%d ",a[i]);
    return a[i];
}
```

2、一个满二叉树中所有节点均不同，已知先序遍历pre，求后序遍历

思路： 根据满二叉树性质找到左右子树的划分，然后根据先序的第一个节点是后序的最后一个节点递归求后序遍历

注意下标之间的对应关系

- l1+1:pre左子树除根节点外的起始点
- l1+half:half是左右子树长度，l1+half到达左子树最后一个节点
- l2：由于post根节点存储在最后，所以左子树起始节点为l2
- l2+half-1:l2+half指向了右子树第一个节点，-1指向左子树最后一个节点

代码：

```c
void PreToPost(int pre[],int low1,int high1,int post[],int low2,int high2){
    int half;
    if(high1>=low1){
        post[high2] = pre[low1];
        half = (high1-low1)/2;
        PreToPost(pre,low1+1,low1+half,post,low2,low2+half-1);//转换左子树
        PreToPost(pre,low1+half+1,high1,post,low2+half,high2-1);//转换右子树
    }
}
```

3、假定用两个一维数组L[1:n] 和R[1:n] 作为有n个节点二叉树的存储结构 L[i]和R[i]分别表示节点i的左孩子和右孩子，0表示为空，试写一个算法表示u是否为节点v的子孙

代码：

```
bool IsChild(int L[], int R[], int v, int u){
    if(v==0) return 0;          //递归出口
    if(L[v]==u || R[v]==v) return 1; //是其孩子节点
    return IsChild(L,R,L[v],u)&& IsChild(L,R,R[v],u);
}
```