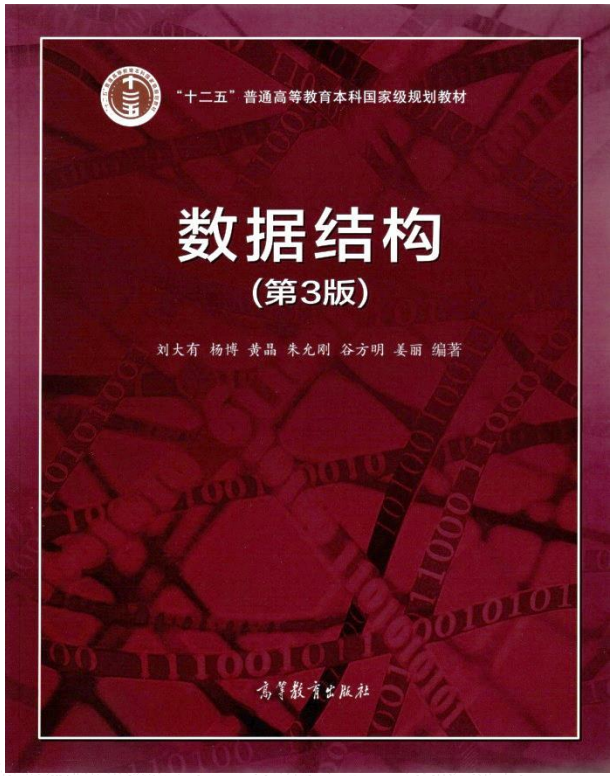


二叉树的存储和操作

- 二叉树的存储结构
- 二叉树遍历的递归算法
- 二叉树遍历的非递归算法
- 二叉树的重建和计数
- 二叉树其他操作

数据之法
结构之美
算法之道

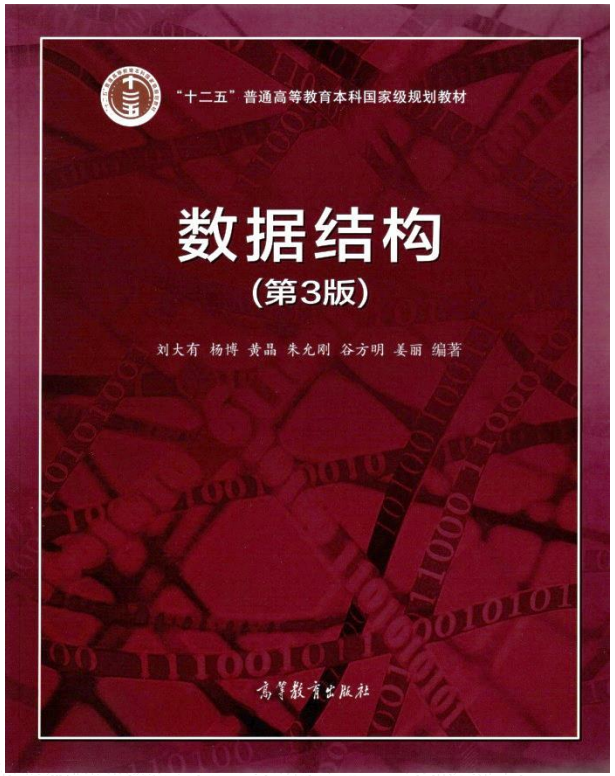
zhuyungang@jlu.edu.cn

我平均每周写5个新程序。诗人必须写诗，而我必须写计算机程序。

——Donald Knuth

2020年接受采访（时年82岁）

图灵奖获得者、斯坦福大学教授



二叉树的存储和操作

- **二叉树的存储结构**
- 二叉树遍历的递归算法
- 二叉树遍历的非递归算法
- 二叉树的重建和计数
- 二叉树其他操作

数据之法
结构之美
算法之道

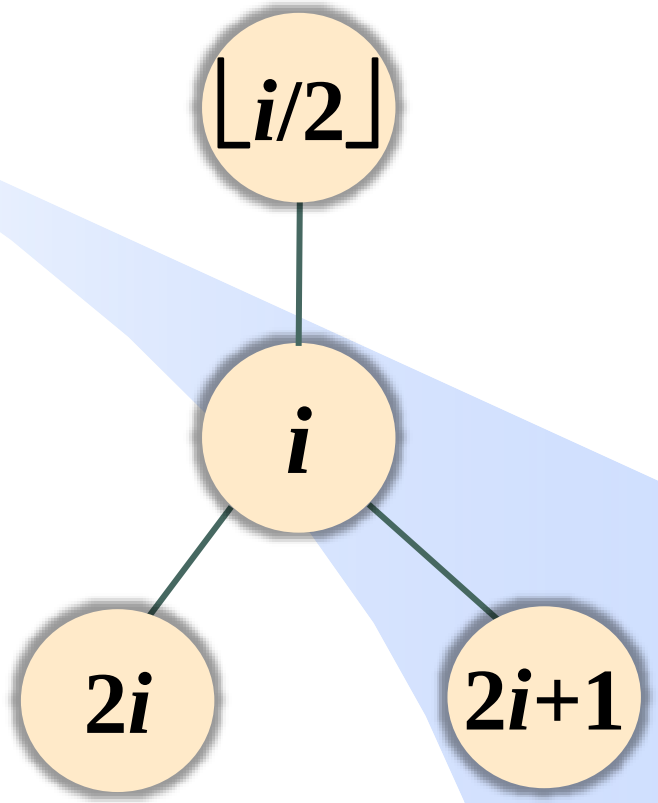
zhuyungang@jlu.edu.cn

二叉树顺序存储

- 要存储一棵二叉树，必须存储其所有结点的数据信息、左孩子和右孩子地址，既可用顺序结构存储，也可用链接结构存储。
- 二叉树的顺序存储是指将二叉树中所有结点存放在一块地址连续的存储空间中，同时反映出二叉树中结点间的逻辑关系。

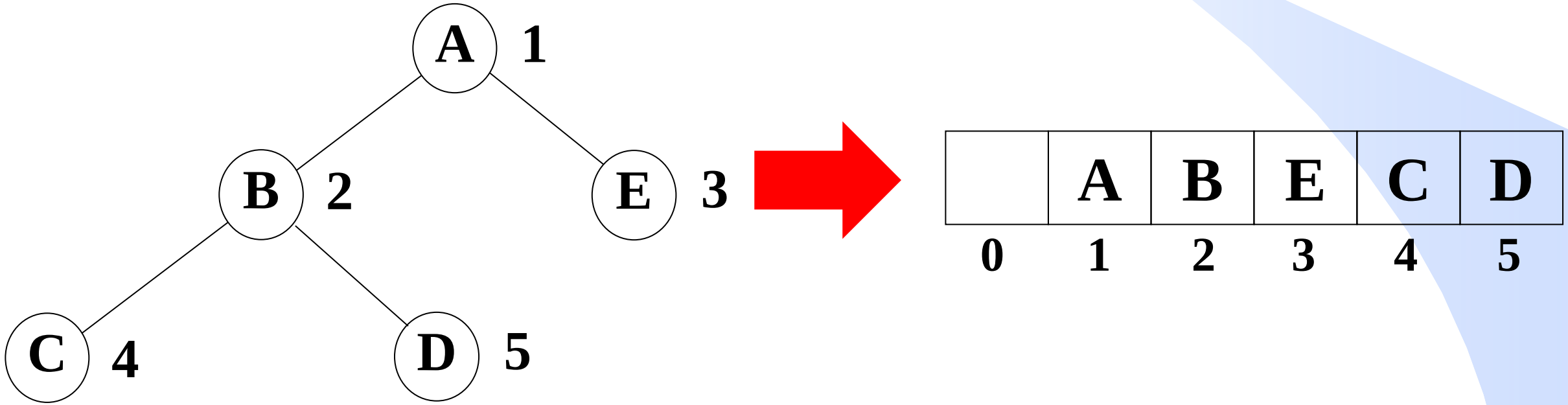
二叉树顺序存储

- 回顾：对于完全二叉树，可按层次顺序对结点编号，结点的编号恰好反映了结点间的逻辑关系。
- 借鉴上述思想，利用一维数组*T*存储二叉树，把编号作为数组下标，根结点存放在*T*[1]位置 。
- 结点*T*[*i*]的左孩子（若存在）存放在*T*[2*i*]处，而*T*[*i*]的右孩子（若存在）存放在*T*[2*i*+1]处。



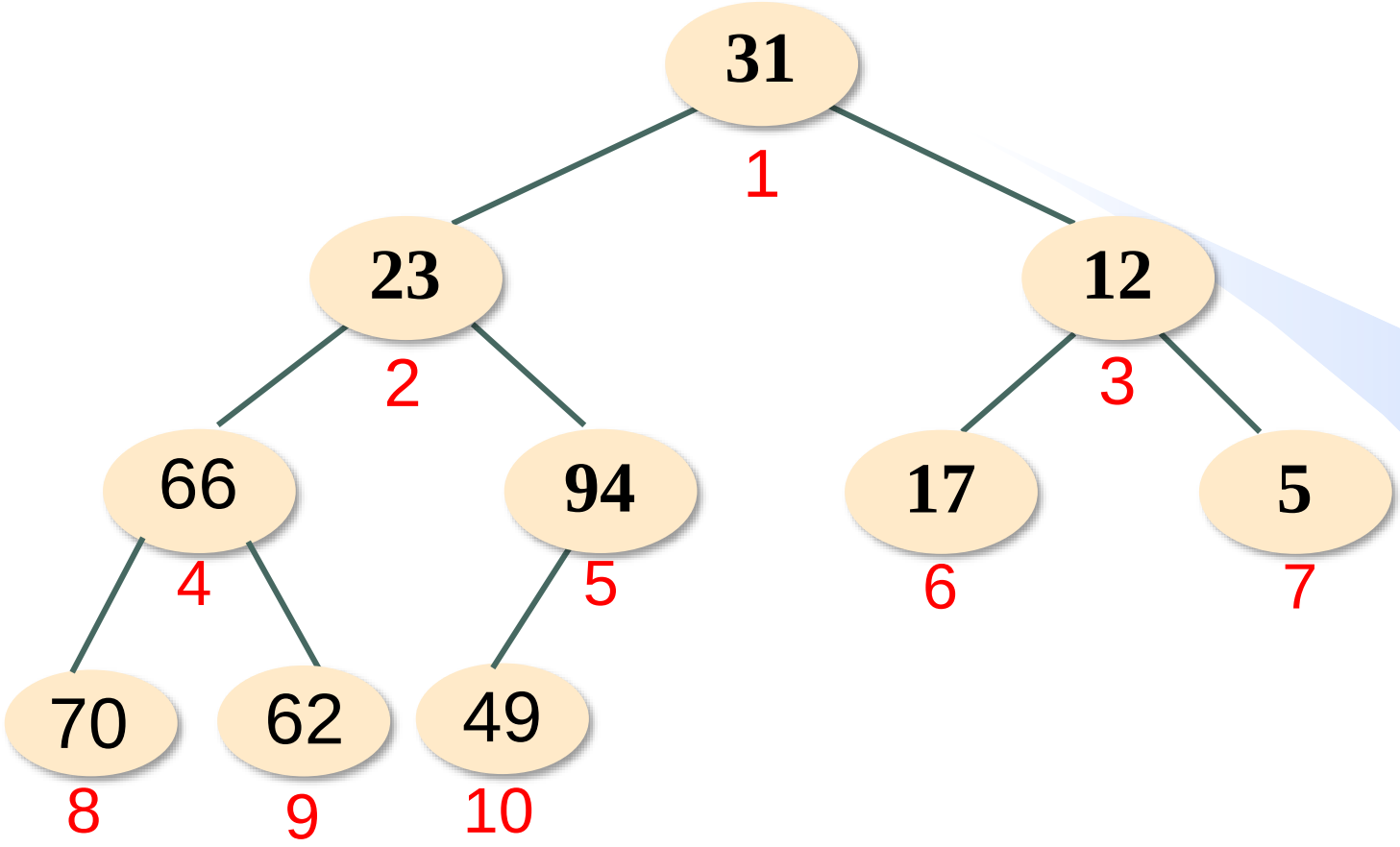
二叉树顺序存储

若一个结点的下标是*i*，则其左孩子（若存在）存放在下标2*i*处，右孩子（若存在）存放在2*i*+1处



二叉树顺序存储结构

二叉树顺序存储

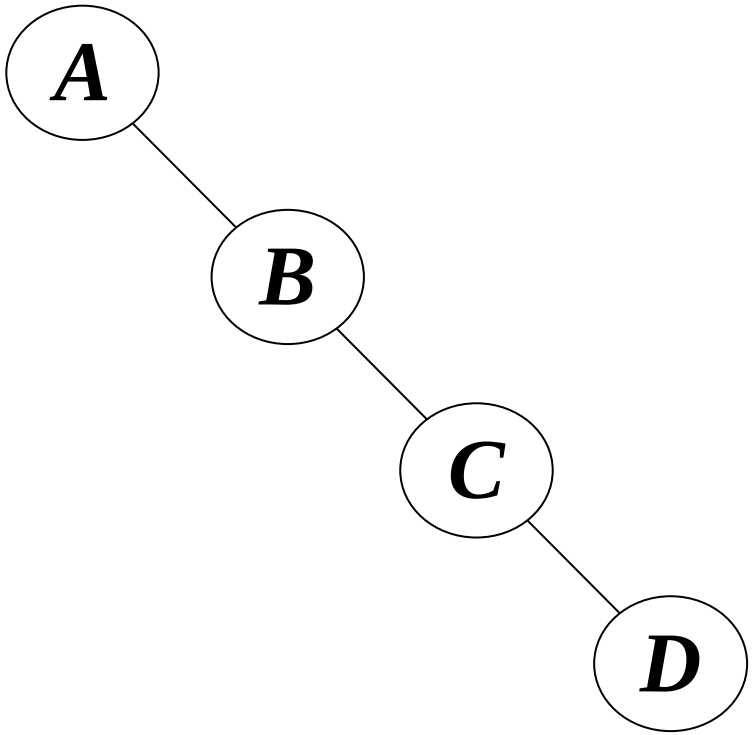


T		31	23	12	66	94	17	5	70	62	49
	0	1	2	3	4	5	6	7	8	9	10

二叉树顺序存储

- 这种顺序存储方式是完全二叉树最简单、最节省空间的存储方式。它实际上只存储了结点数据域之值，而未存储其左孩子和右孩子地址，通过下标的计算可找到一个结点的子结点和父结点。
- 非常适合于完全二叉树。但是，应用到非完全二叉树时，将造成空间浪费。

非完全二叉树的顺序存储



	A	B	C	D
0	1	2	3	4

无法表达父子结点间的逻辑关系

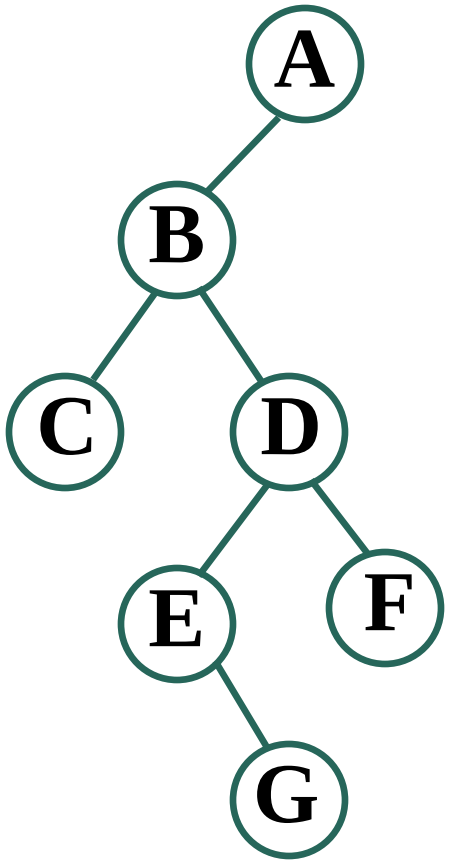
T		A		B				C							D
		1		3				7							15

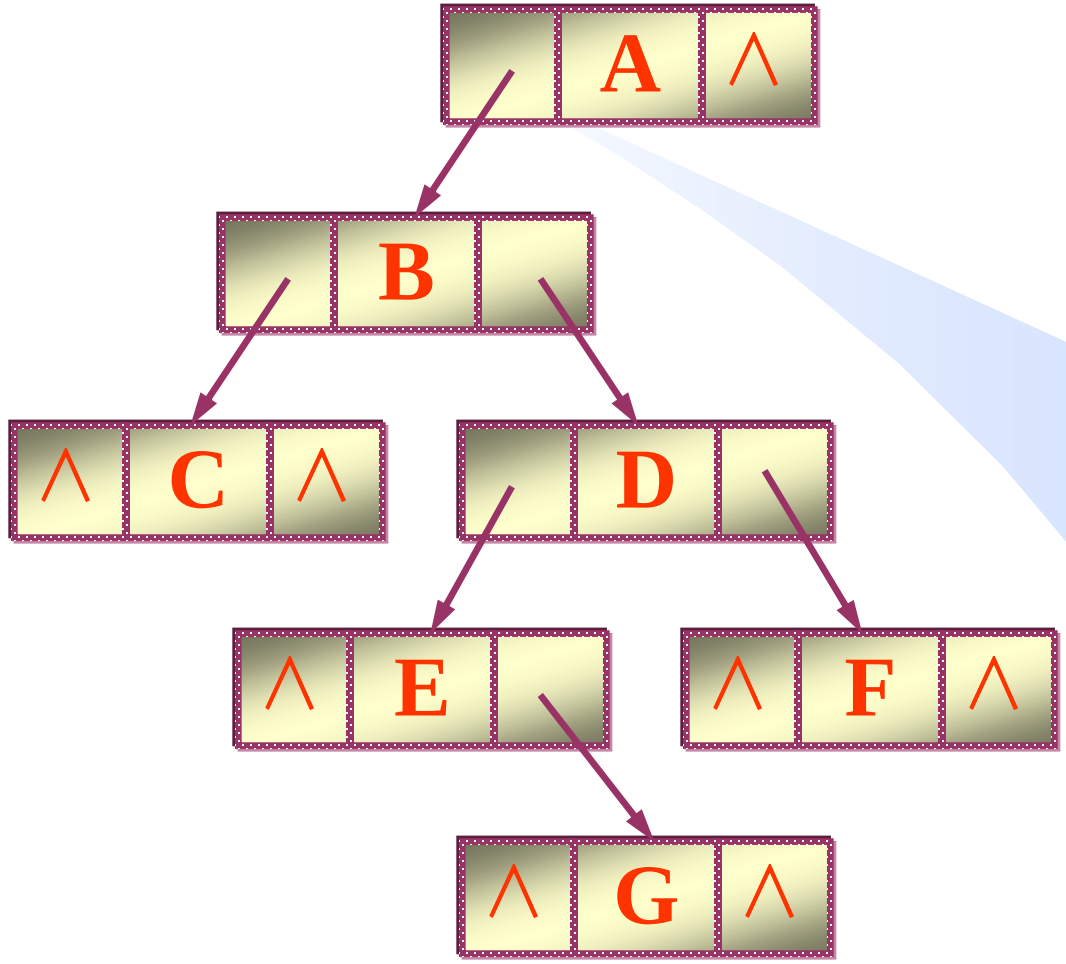
二叉树链接存储——二叉链表

- 各结点被随机存放在内存空间中，结点间的关系用指针表达。
- 二叉树的结点结构：二叉树结点应包含三个域——数据域*data*、指针域*left*（称为左指针）和指针域*right*（称为右指针），其中左、右指针分别指向该结点的左、右子结点。


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

二叉树链接存储





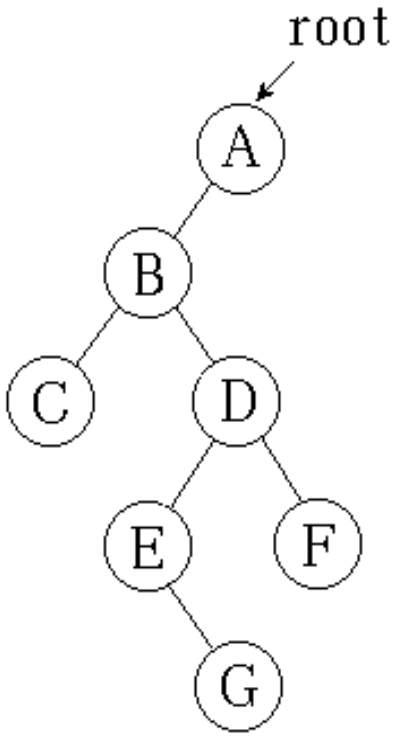
二叉树链接存储

- 左子结点 = 左孩子
- 右子结点 = 右孩子
- **ADL：Left (t)、Data (t)、Right (t)**

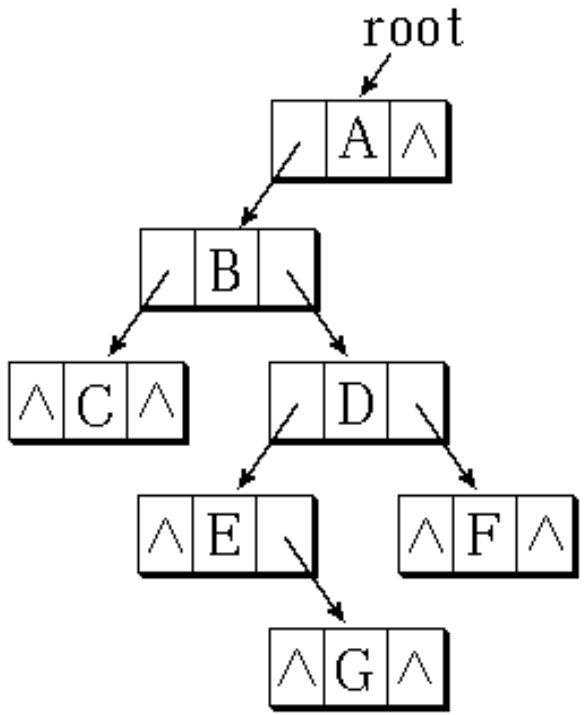
二叉树链接存储——三叉链表

Left	*Data*	*Parent*	*Right*

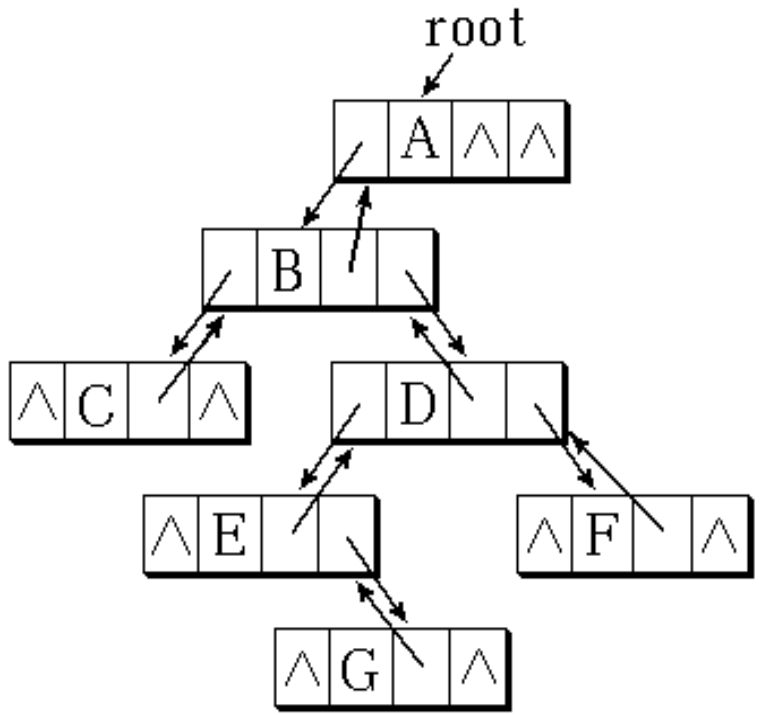
Parent指针指向父结点



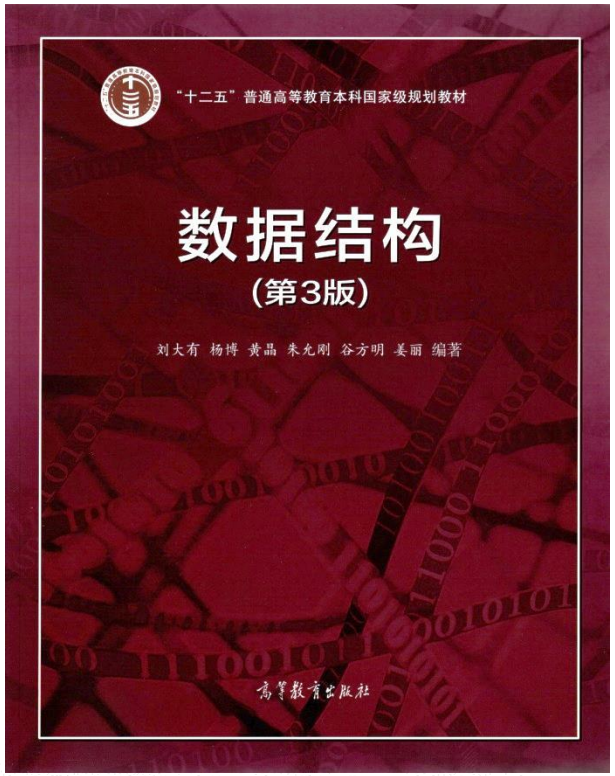
(a) 二叉树



(b) 二叉链表



(c) 三叉链表



二叉树的存储和操作

- 二叉树的存储结构
- **二叉树遍历及其递归算法**
- 遍历的非递归算法
- 二叉树的重建和计数
- 二叉树其他操作

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

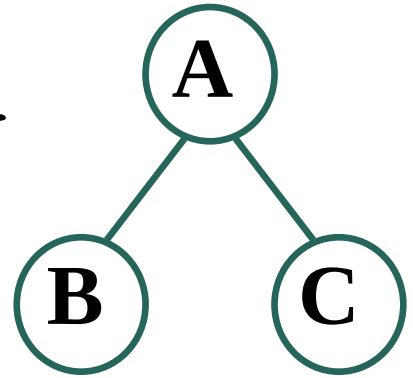
二叉树的遍历

二叉树的遍历：按照一定**次序**访问二叉树中所有结点，并且每个结点仅被**访问一次**的过程。

当二叉树为空则什么都不做；否则遍历分三步进行：

遍历方法 / 步骤	先根遍历 （先/前序遍历）
步骤一	访问根结点
步骤二	先根遍历左子树
步骤三	先根遍历右子树

二叉树

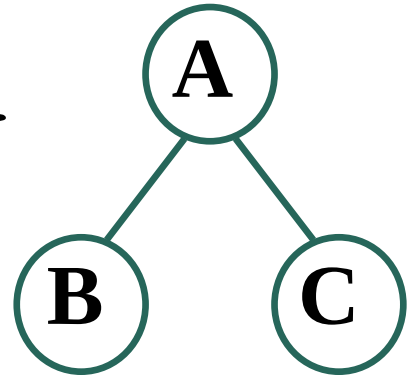


遍历方式 { 先根遍历：ABC

当二叉树为空则什么都不做；否则遍历分三步进行：

遍历方法 / 步骤	先根遍历（先/前序遍历）	中根遍历（中序遍历）
步骤一	访问根结点	中根遍历左子树
步骤二	先根遍历左子树	访问根结点
步骤三	先根遍历右子树	中根遍历右子树

二叉树

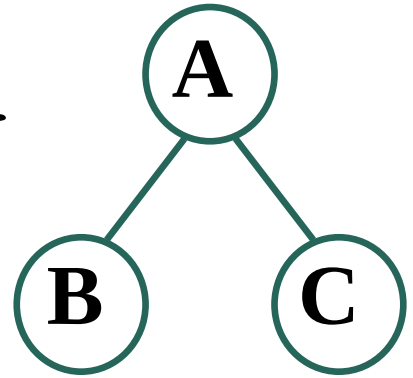


遍历方式
- 先根遍历：ABC
- 中根遍历：BAC

当二叉树为空则什么都不做；否则遍历分三步进行：

遍历方法 / 步骤	先根遍历（先/前序遍历）	中根遍历（中序遍历）	后根遍历（后序遍历）
步骤一	访问根结点	中根遍历左子树	后根遍历左子树
步骤二	先根遍历左子树	访问根结点	后根遍历右子树
步骤三	先根遍历右子树	中根遍历右子树	访问根结点

二叉树



遍历方式
- 先根遍历：ABC
- 中根遍历：BAC
- 后根遍历：BCA

先根（中根、后根）遍历二叉树 T，得到 T 之结点的一个序列，称为 T 的先根（中根、后根）序列。

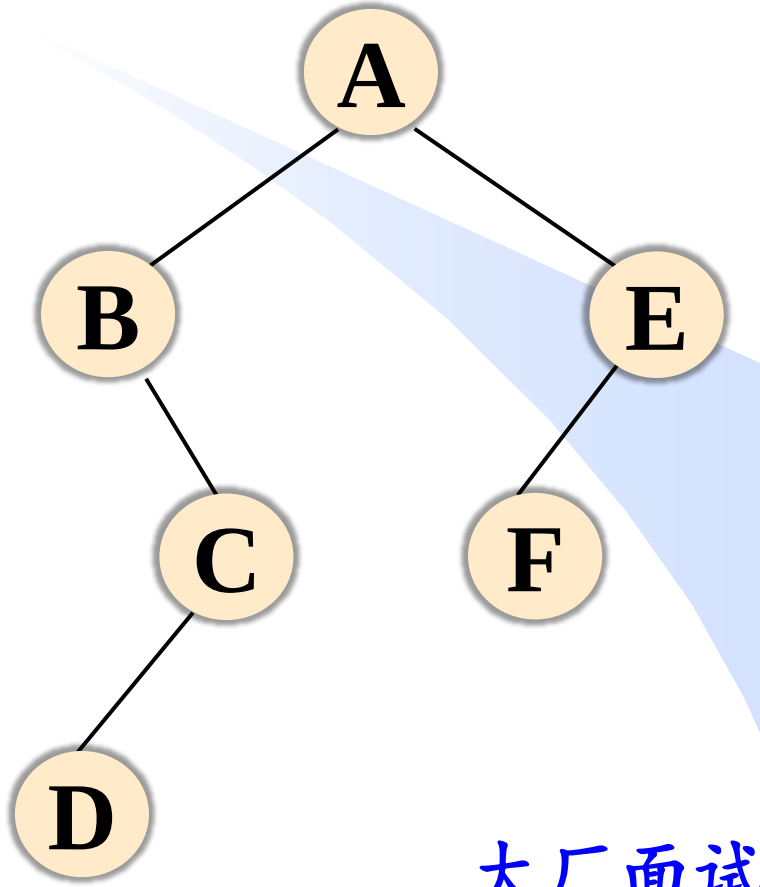
先根遍历 (Preorder Traversal，前/先序遍历)

先根遍历二叉树算法的框架:

- 若二叉树为空，则空操作;
- 否则
  - ✓访问根结点；
  - ✓先根遍历左子树；
  - ✓先根遍历右子树。

遍历结果

A B C D E F



大厂面试题

LeetCode144

二叉树递归先根遍历的算法

```
void Preorder (TreeNode* t){
     if(t == NULL) return;
     visit(t->data);
     Preorder(t->left);
     Preorder(t->right);
}
```

时间复杂度O(*n*)
空间复杂度O(*h*)
*n*为二叉树结点数
*h*为二叉树高度

```
void visit (int data){
     printf("%d ",data);
}
```

二叉树递归先根遍历的算法

```
void Preorder (TreeNode* t){
    if(t == NULL) return;
    visit(t->data);
    Preorder(t->left);
    Preorder(t->right);
}
```

分治法
分而治之

大事化小
小事化了

大问题
访问整棵树
的所有结点

- 子问题1
  访问根结点
- 子问题2
  访问左子树的结点
- 子问题3
  访问右子树的结点

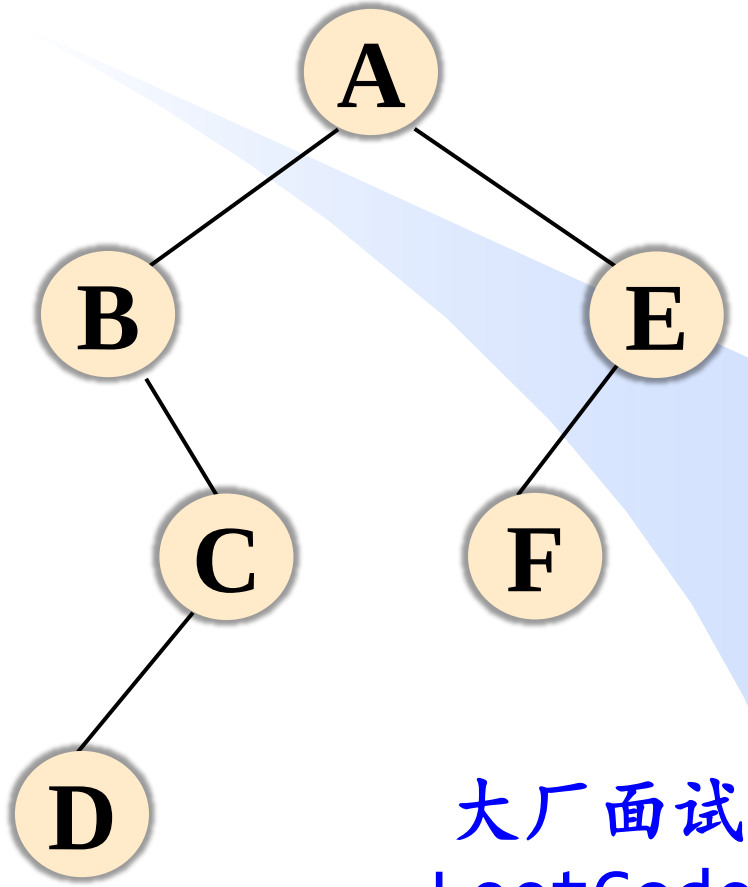
分治法	子问题相互独立 不重叠	自顶向下 递归实现
动态规划	子问题间往往具有重叠性，可将子问题的解存入表中，以后再遇到相同的子问题直接查表	自底向上 递推实现

中根遍历 (Inorder Traversal，中序遍历)

中根遍历二叉树算法的框架：

- 若二叉树为空，则空操作；
- 否则
  - ✓中根遍历左子树；
  - ✓访问根结点；
  - ✓中根遍历右子树。

遍历结果　**B D C A F E**



大厂面试题

LeetCode94

二叉树中根遍历的递归算法

```
void Inorder(TreeNode* t){
    if (t == NULL) return;
    Inorder(t->left);
    visit(t->data);
    Inorder(t->right);
}
```

时间复杂度O(*n*)
空间复杂度O(*h*)
*n*为二叉树结点数
*h*为二叉树高度

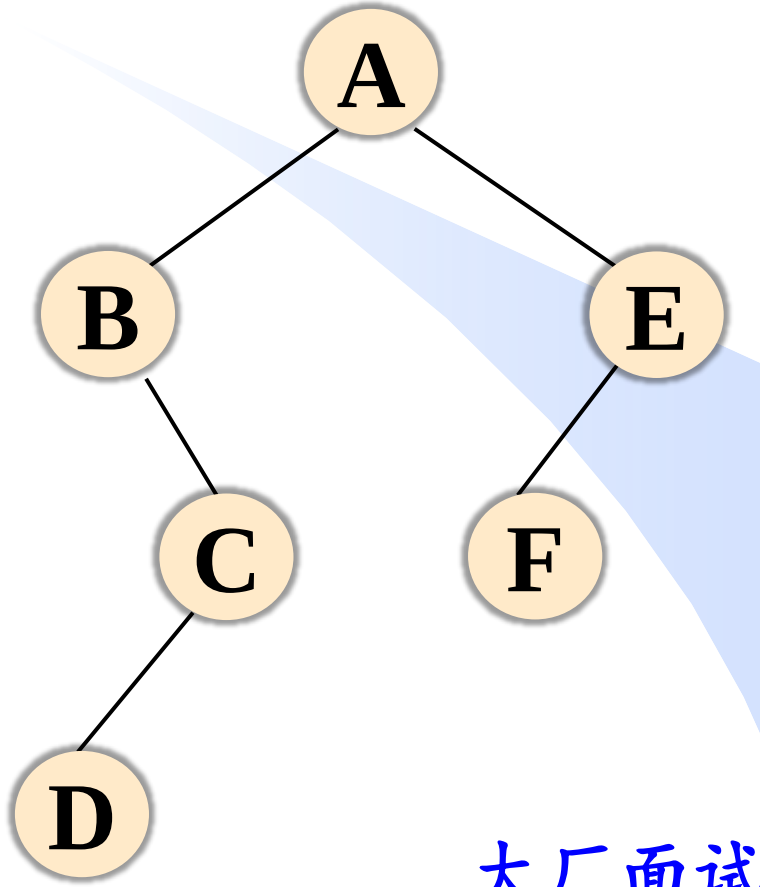
```
void visit (int data){
    printf("%d ",data);
}
```

后根遍历 (Postorder Traversal，后序遍历)

后根遍历二叉树算法的框架:

- 若二叉树为空，则空操作；
- 否则
  - ✓后根遍历左子树；
  - ✓后根遍历右子树；
  - ✓访问根结点。

遍历结果 **D C B F E A**



大厂面试题

LeetCode145

二叉树后根遍历的递归算法

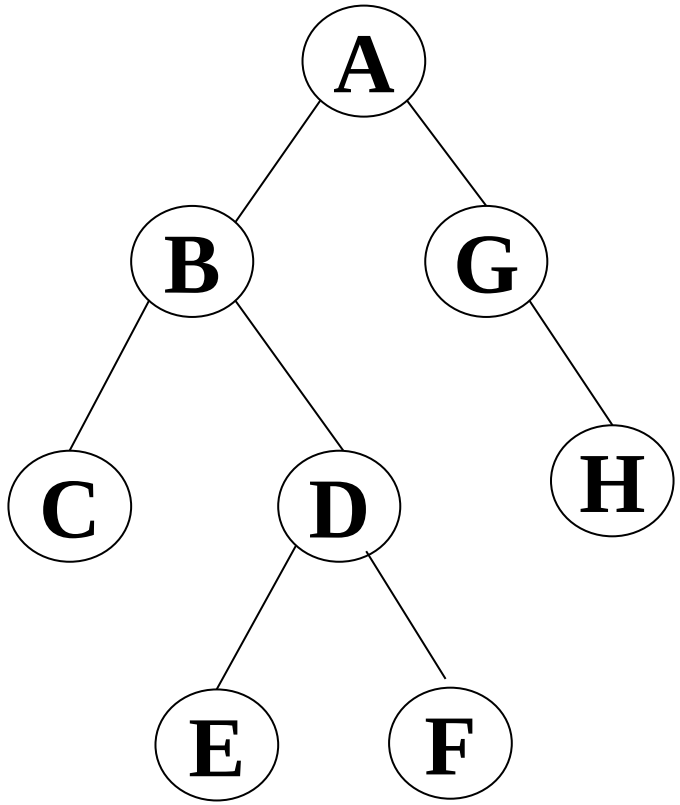
```
void Postorder(TreeNode* t){
    if (t == NULL) return;
    Postorder(t->left);
    Postorder(t->right);
    visit(t->data);
}
```

时间复杂度O(*n*)
空间复杂度O(*h*)
*n*为二叉树结点数
*h*为二叉树高度

```
void visit (int data){
    printf("%d ", data);
}
```

课下练习

下面二叉树的先根序列为_____，中根序列为_____
_____，后根序列为_____。



练习

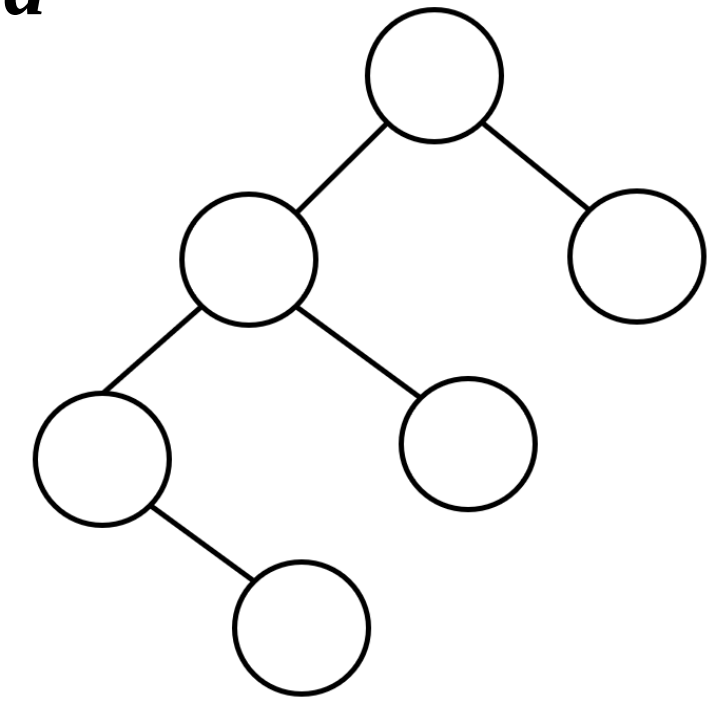
已知一棵二叉树的树形如下，若其后根序列为*f, d, b, e, c, a*，则其先根序列为__*A*__. 【2023年考研题全国卷】

A. *a e d f b c*

B. *a c e b d f*

C. *c a b e f d*

D. *d f e b a c*



练习

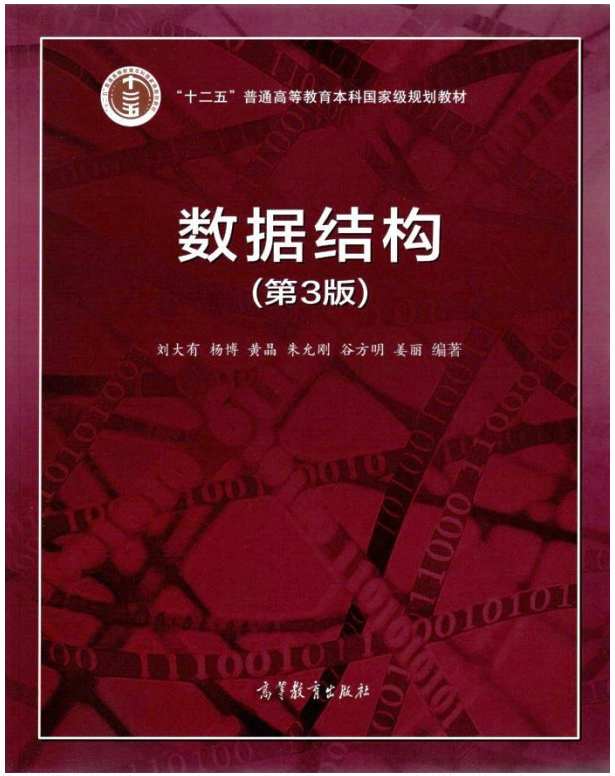
要使一棵非空二叉树的先根序列与中根序列相同，其所有非叶结点须满足的条件是(　*B*　)【2017年考研题全国卷】

A.只有左子树　　B.只有右子树

C.结点的度均为1　　D.结点的度均为2

先根序列：根 左 右

中根序列：左 根 右



二叉树的存储和操作

- 二叉树的存储结构
- 二叉树遍历的递归算法
- **遍历的非递归算法**
- 二叉树的重建和计数
- 二叉树其他操作

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

为什么研究二叉树遍历的非递归算法？

- 当二叉树高度很高时，递归算法可能因递归深度过深导致系统栈溢出
- 有利于更深刻理解二叉树遍历的过程

非递归先根遍历算法

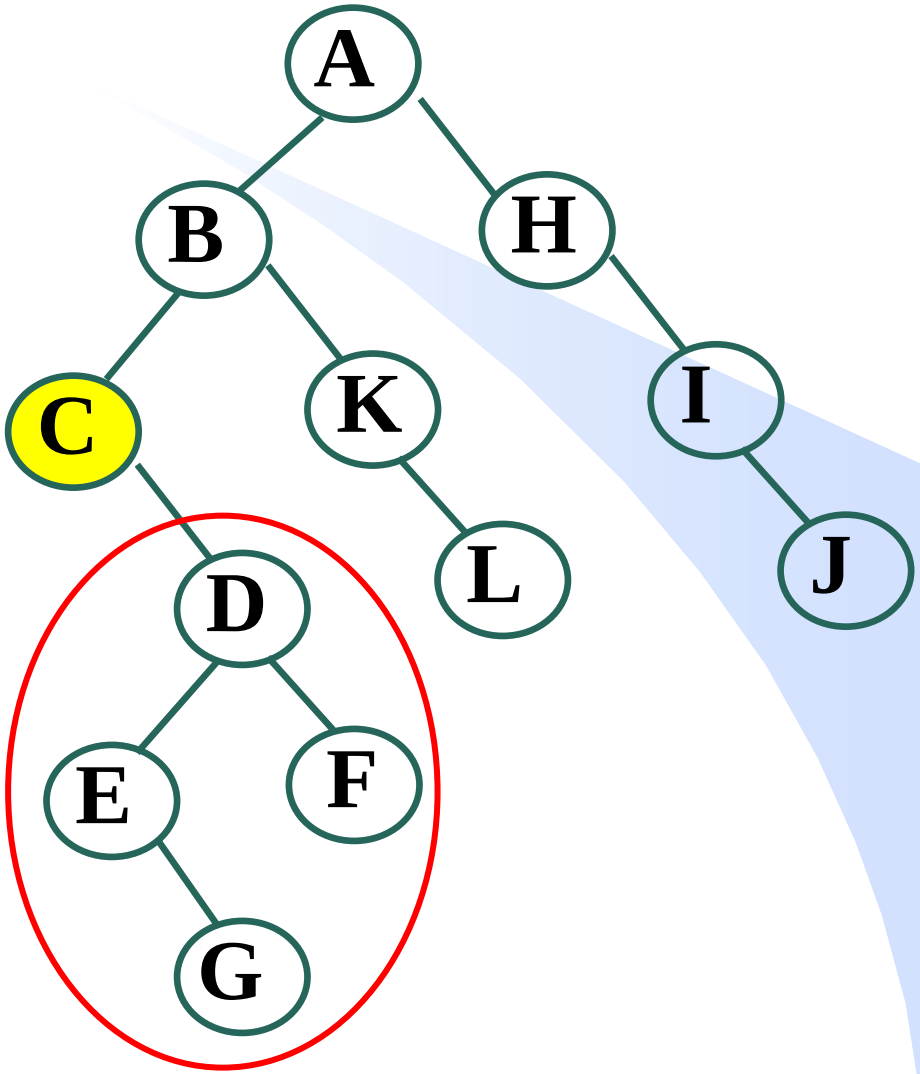
先根遍历过程：

① 从根结点开始自上而下沿着左侧分支访问结点。

② 自下而上依次访问沿途各结点的右子树。

不同右子树的遍历：

- 相互独立
- 自成一个子任务

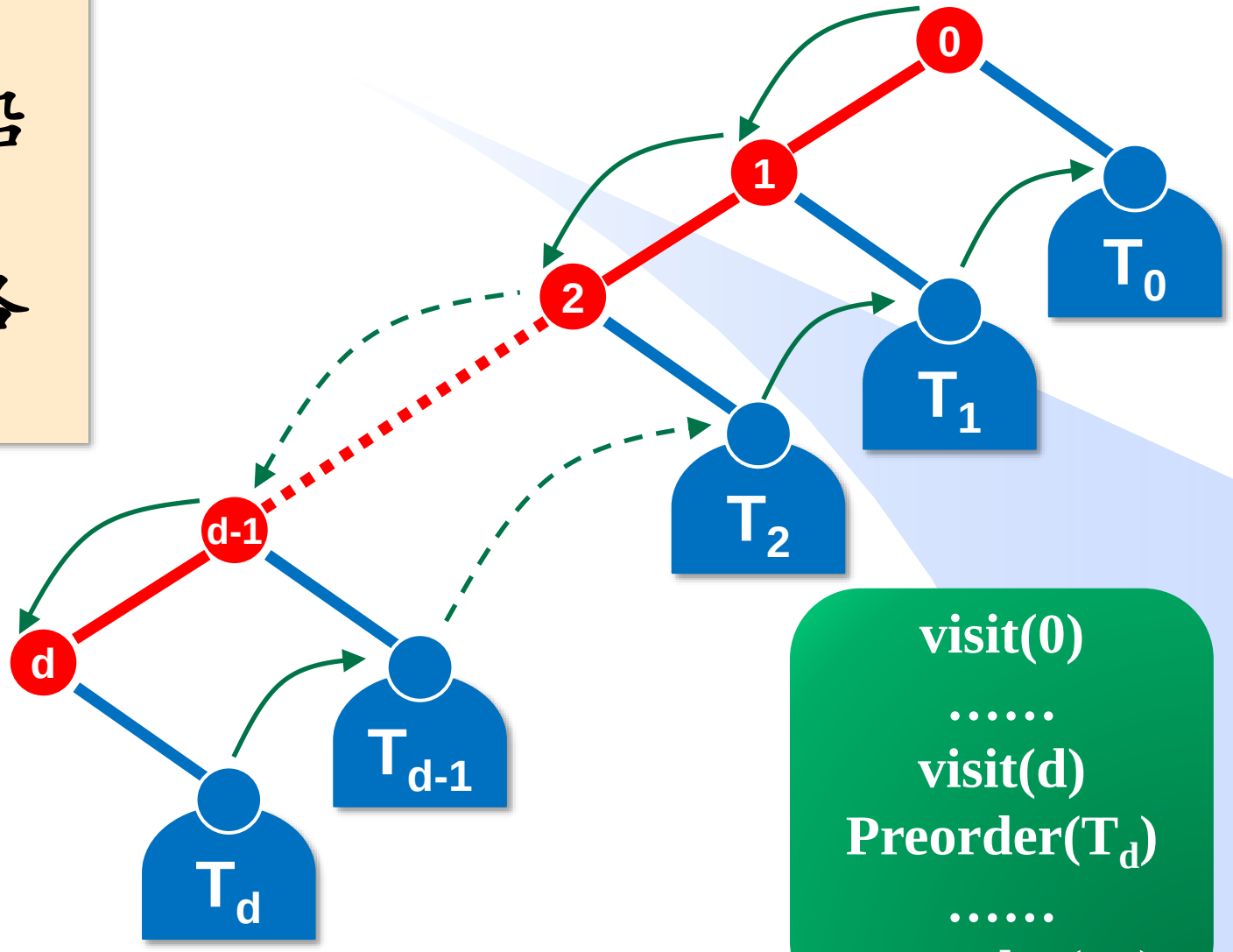


非递归先根遍历算法

先根遍历过程：

① 从根结点开始自上而下沿着左侧分支访问结点。

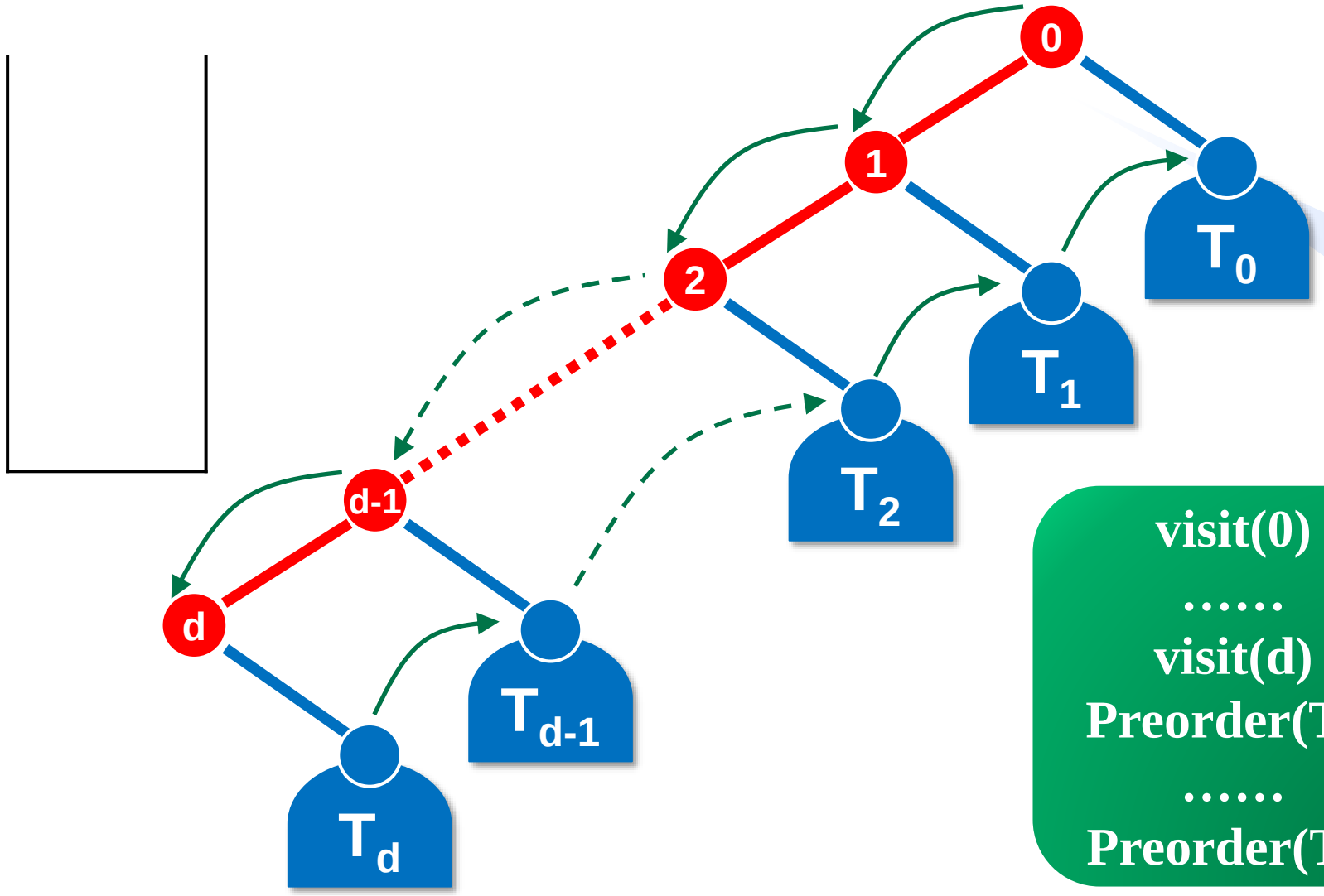
② 自下而上依次访问沿途各结点的右子树。



不同右子树的遍历：

- 相互独立
- 自成一个子任务

非递归先根遍历算法



策略：从根结点开始自上而下沿着左分支访问结点，并把该结点压栈。之后再自下而上弹栈，访问沿途结点的右子树。

visit(0)

……

visit(d)

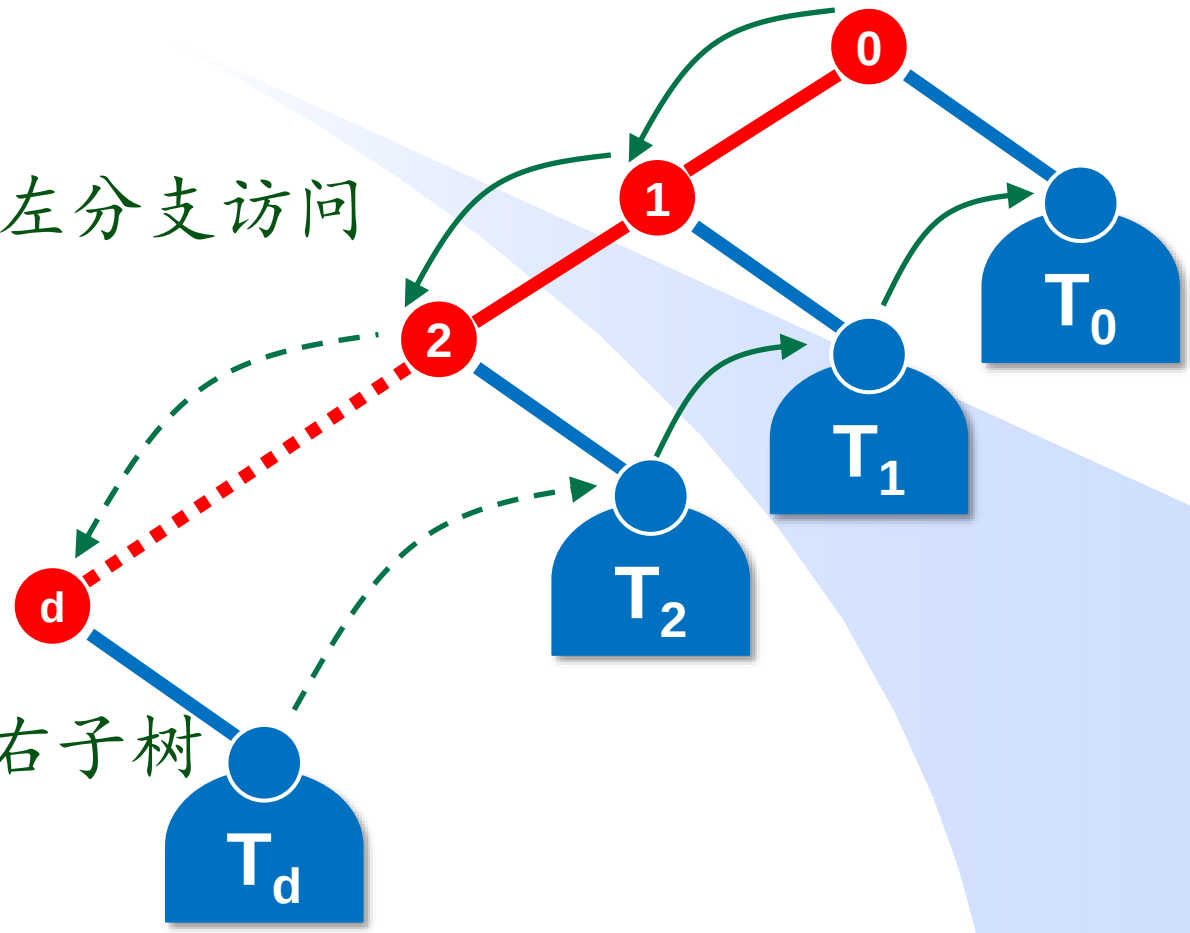
Preorder(T_d)

……

Preorder(T_0)

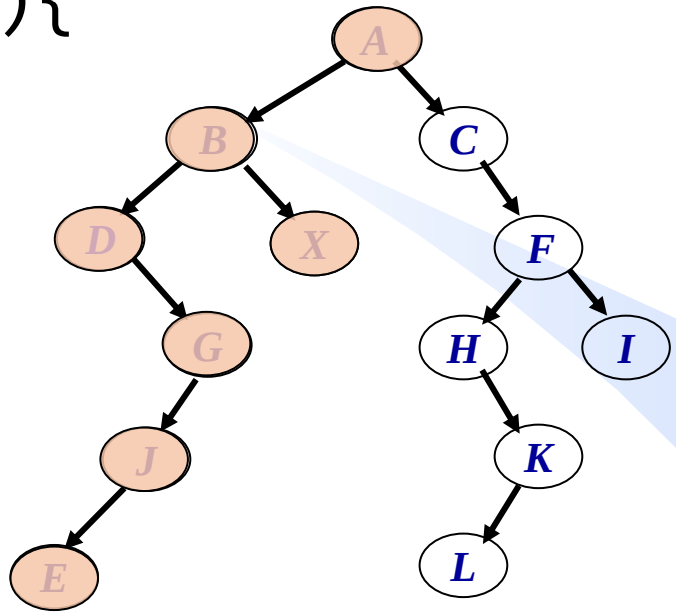
非递归先根遍历算法

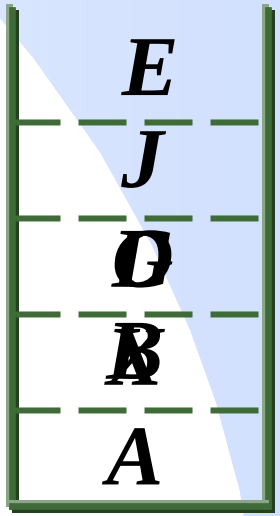
```
void NonRecPreOrder(TreeNode* t){
    Stack S; TreeNode * p = t;
    while(true) {
        while(p!=NULL){//自上而下沿左分支访问
            visit(p->data);
            S.PUSH(p);
            p=p->left;
        }
        if(S.IsEmpty()) return;
        p=S.POP(); //自下而上访问各右子树
        p=p->right;
    }
}
```



非递归先根遍历算法运行实例

```
void NonRecPreOrder(TreeNode *t){
    Stack S; TreeNode *p = t;
    while(true) {
        while(p != NULL){
            visit(p->data);
            S.PUSH(p);
            p=p->left;
        }
        if(S.IsEmpty()) return;
        p = S.POP();  //自下而上访问各右子树
        p = p->right;
    }
}
```

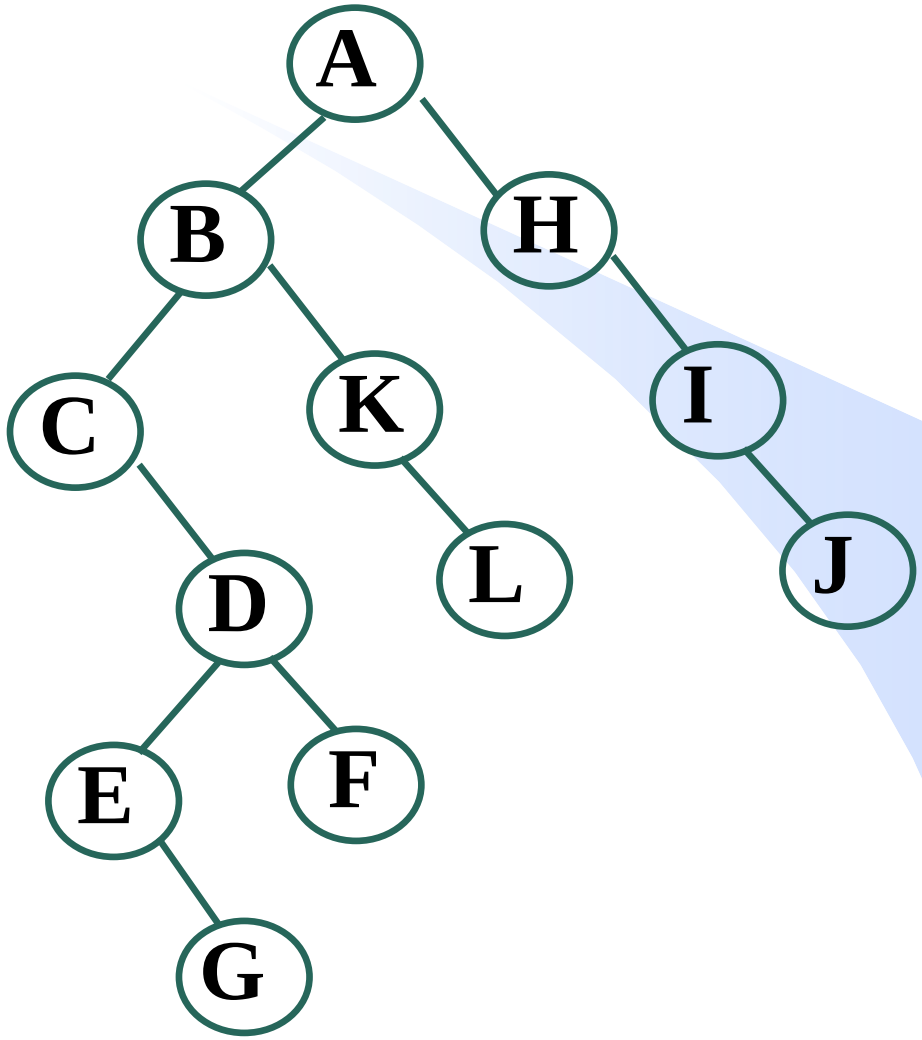




非递归中根遍历算法

中根序列第一个结点

从根结点出发，沿左分支下行，直到最深的结点（没有左孩子的结点），该结点是中根序列第一个结点



非递归中根遍历算法

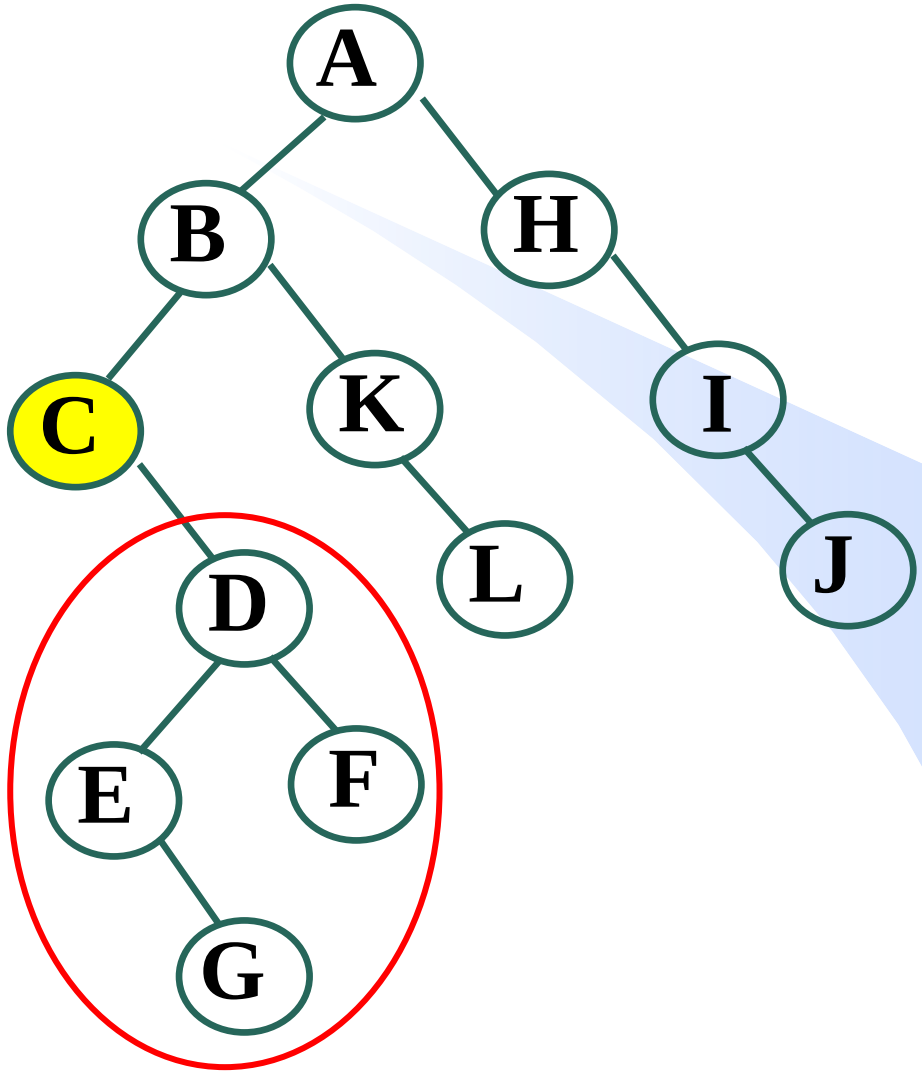
中根遍历过程：

① 从根结点出发沿左分支下行，直到最深的结点（无左孩子）。

② 沿着左侧通道，自下而上依次访问沿途各结点及其右子树。

不同右子树的遍历：

- 相互独立
- 自成一个子任务



非递归中根遍历算法

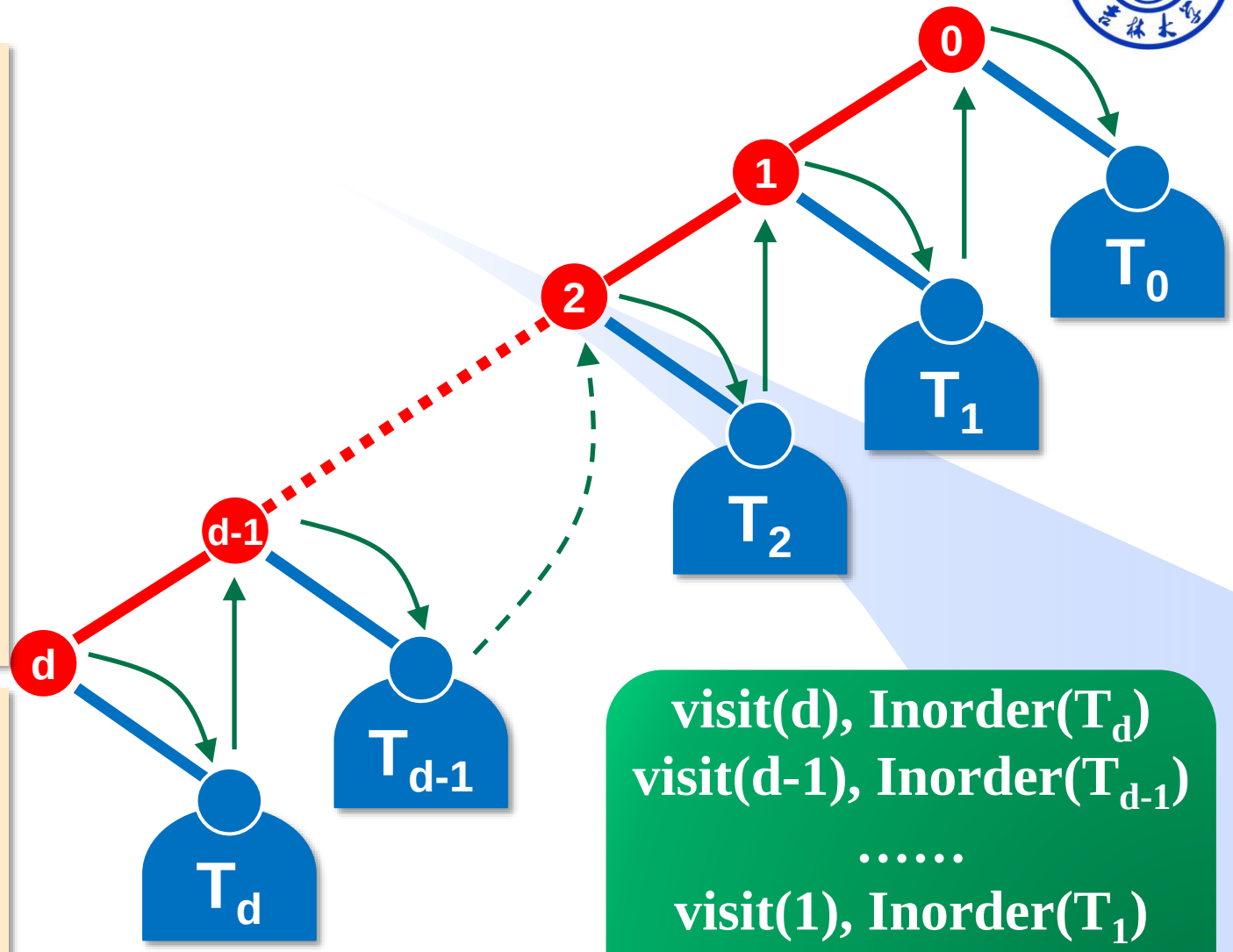
中根遍历过程：

① 从根结点出发沿左分支下行，直到最深的结点（无左孩子）。

② 沿着左侧通道，自下而上依次访问沿途各结点及其右子树。

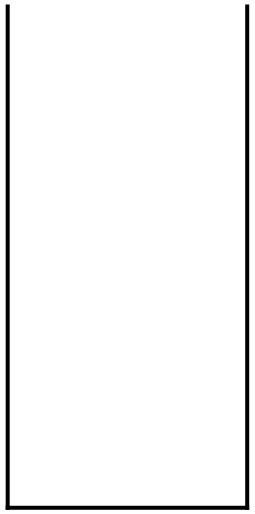
不同右子树的遍历：

- 相互独立
- 自成一个子任务

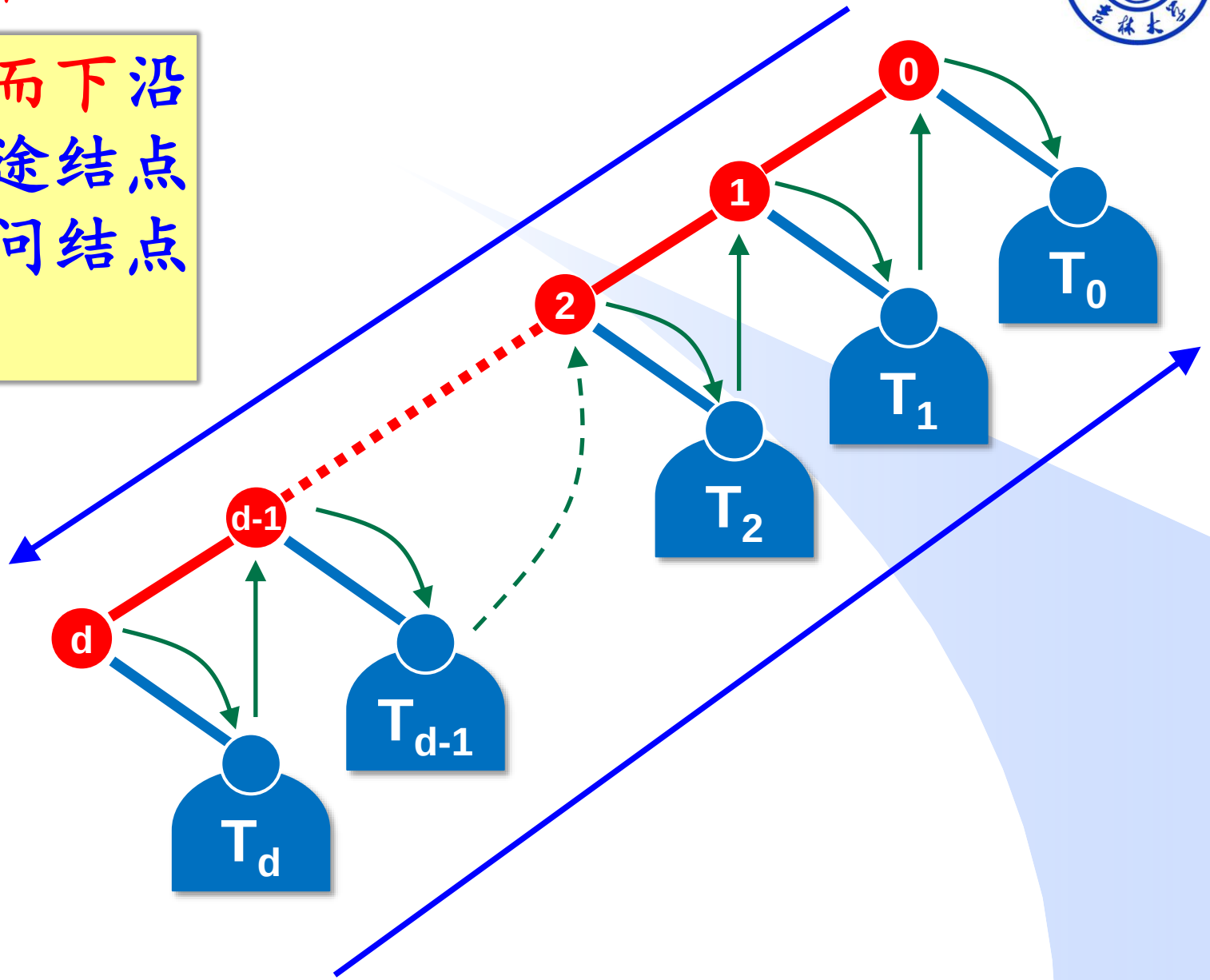


非递归中根遍历算法

策略：从根结点开始自上而下沿着左侧分支下行，并把沿途结点压栈。自下而上弹栈，访问结点，访问其右子树

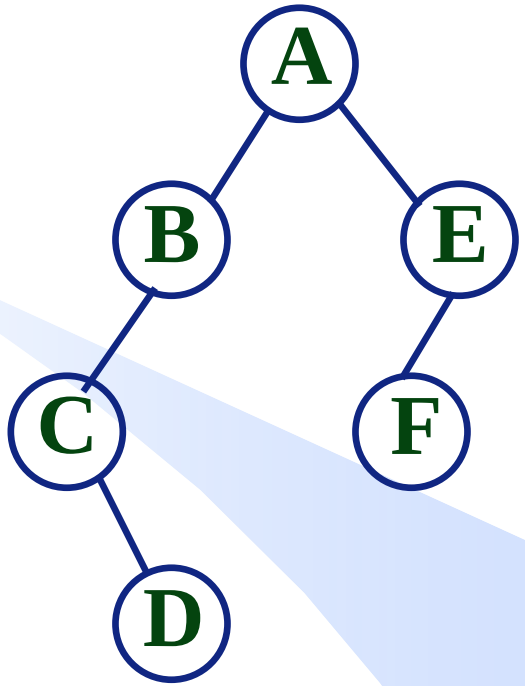


用栈存放沿途遇到的结点



非递归中根遍历算法

```
void NonRecInOrder(TreeNode *t){
  Stack S; TreeNode *p = t;
  while (true) {
     while (p != NULL) {//沿左分支下行
           S.PUSH(p);
           p=p->left;
     }
     if (S.IsEmpty()) return;
     p=S.POP(); //自下而上访问结点及右子树
     visit(p->data);
     p=p->right;
  }
}
```



非递归中根遍历算法——正确性证明

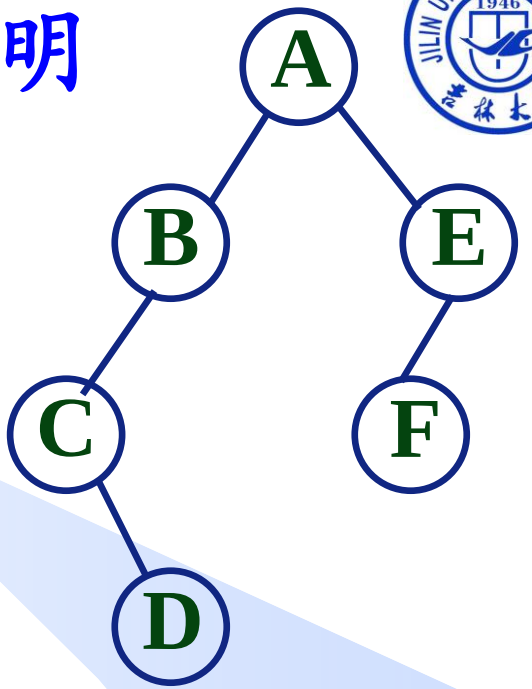
```
void NonRecInOrder(TreeNode *t){
  Stack S; TreeNode *p = t;
  while (true) {
     while (p != NULL) {
           S.PUSH(p);
           p=p->left;
     }
     if (S.IsEmpty()) return;
     p=S.POP();
     visit(p->data);
     p=p->right;
  }
}
```

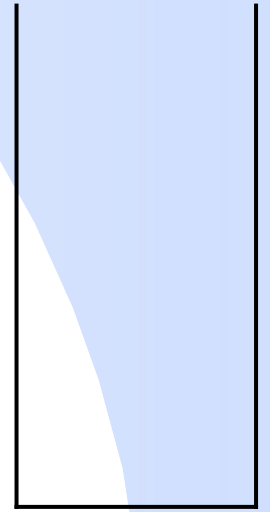
归纳法：设n为p指向的二叉树的结点个数。

① n=0时成立

② 假设<n时算法正确

③ 往证等于n时亦正确





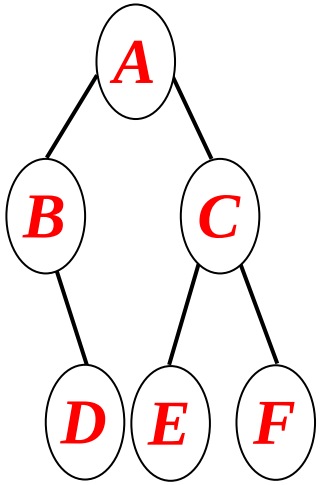
算法正确：

✓能够按中根遍历的顺序访问各结点

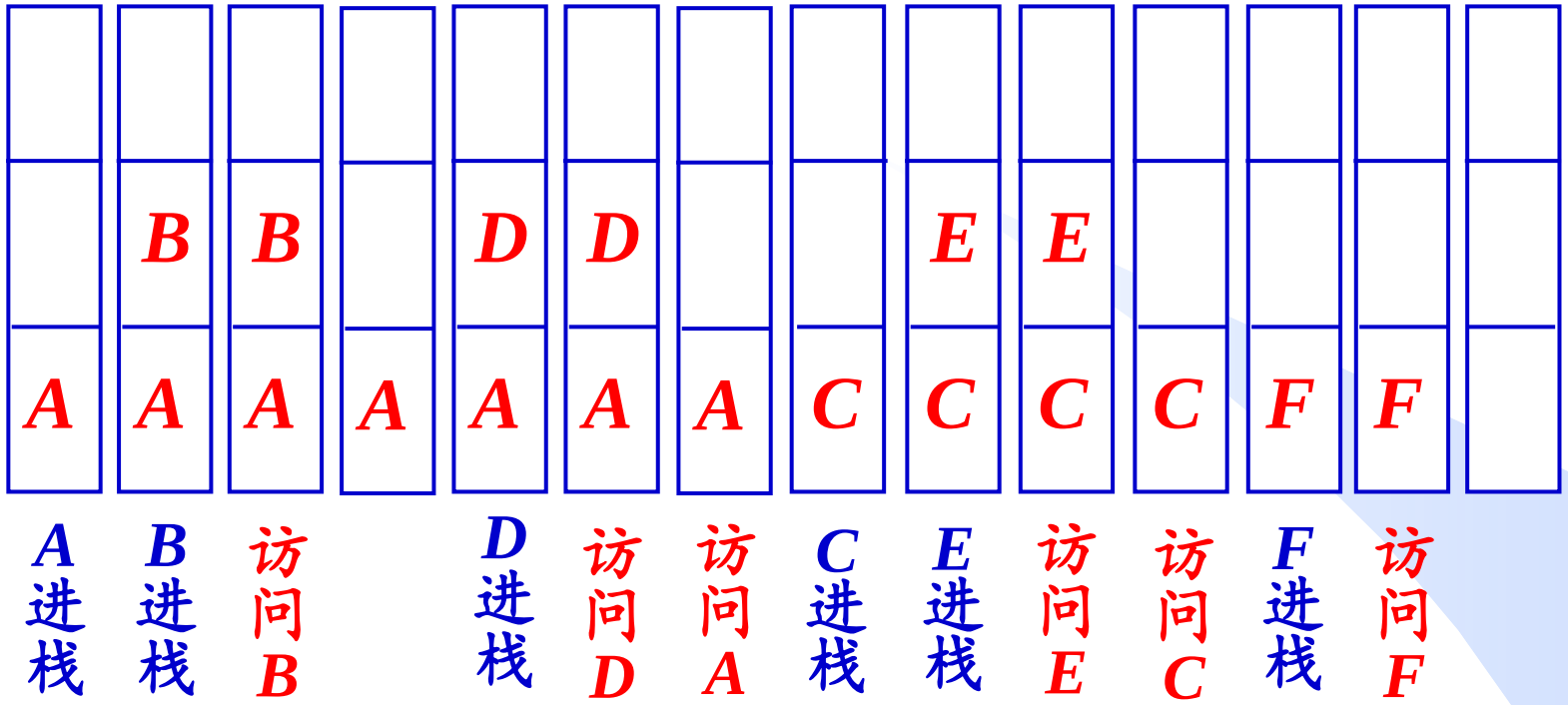
✓遍历结束后站内不留存树中结点

✓遍历结束后算法执行到此处

运行实例：留做作业



(a) 二叉树



中根遍历(a)中二叉树，栈内容变化过程

非递归先根遍历

```
void NonRecPreOrder(TreeNode* t){
    Stack S; TreeNode *p = t;
    while(true) {
        while(p != NULL){
            visit(p->data);
            S.PUSH(p);
            p=p->left;
        }
        if(S.IsEmpty()) return;
        p=S.POP();
        p=p->right;
    }
}
```

非递归中根遍历

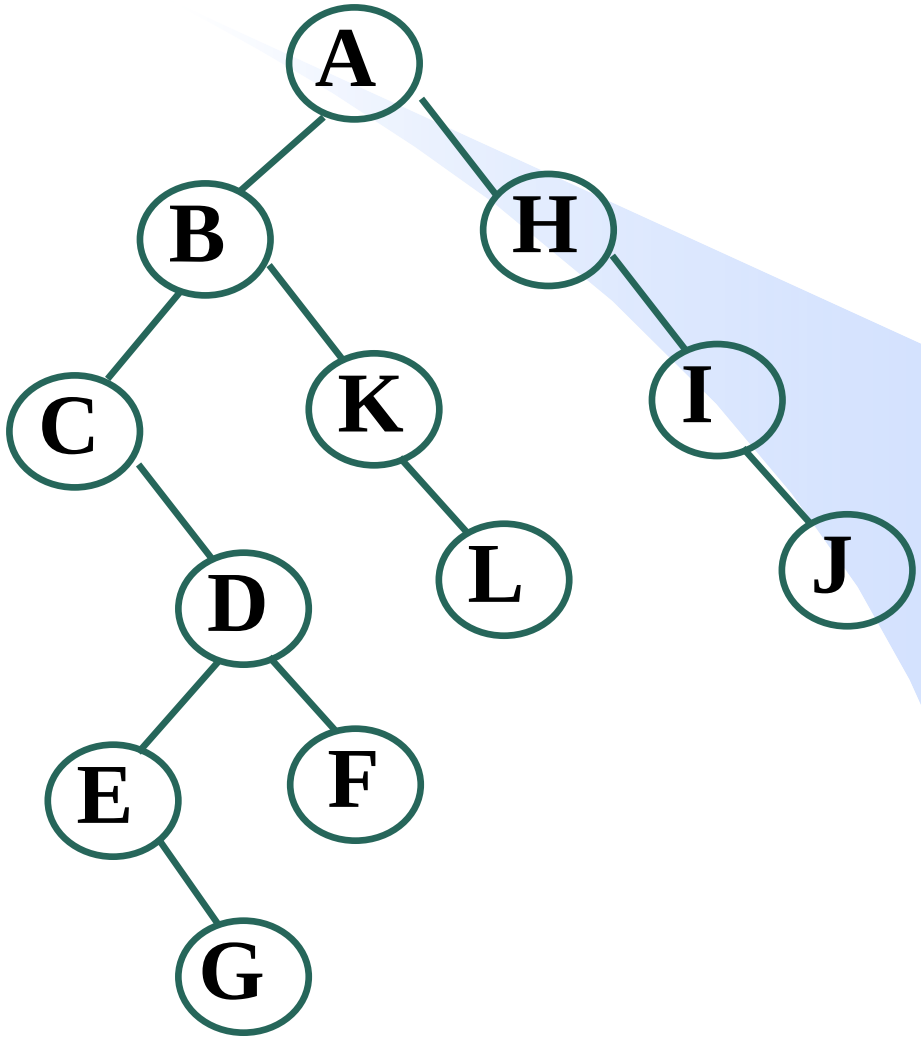
```
void NonRecInOrder(TreeNode *t){
   Stack S; TreeNode *p = t ;
   while (true) {
      while (p != NULL) {
         S.PUSH(p);
         p=p->left;
      }
      if(S.IsEmpty()) return;
      p=S.POP();
      visit(p->data);
      p=p->right;
   }
}
```

先根遍历的进栈序列=中根遍历的进栈序列

先根遍历的出栈序列=中根遍历的出栈序列

非递归后根遍历算法

回顾中根遍历：从根结点出发，沿左分支下行，直到最深的结点



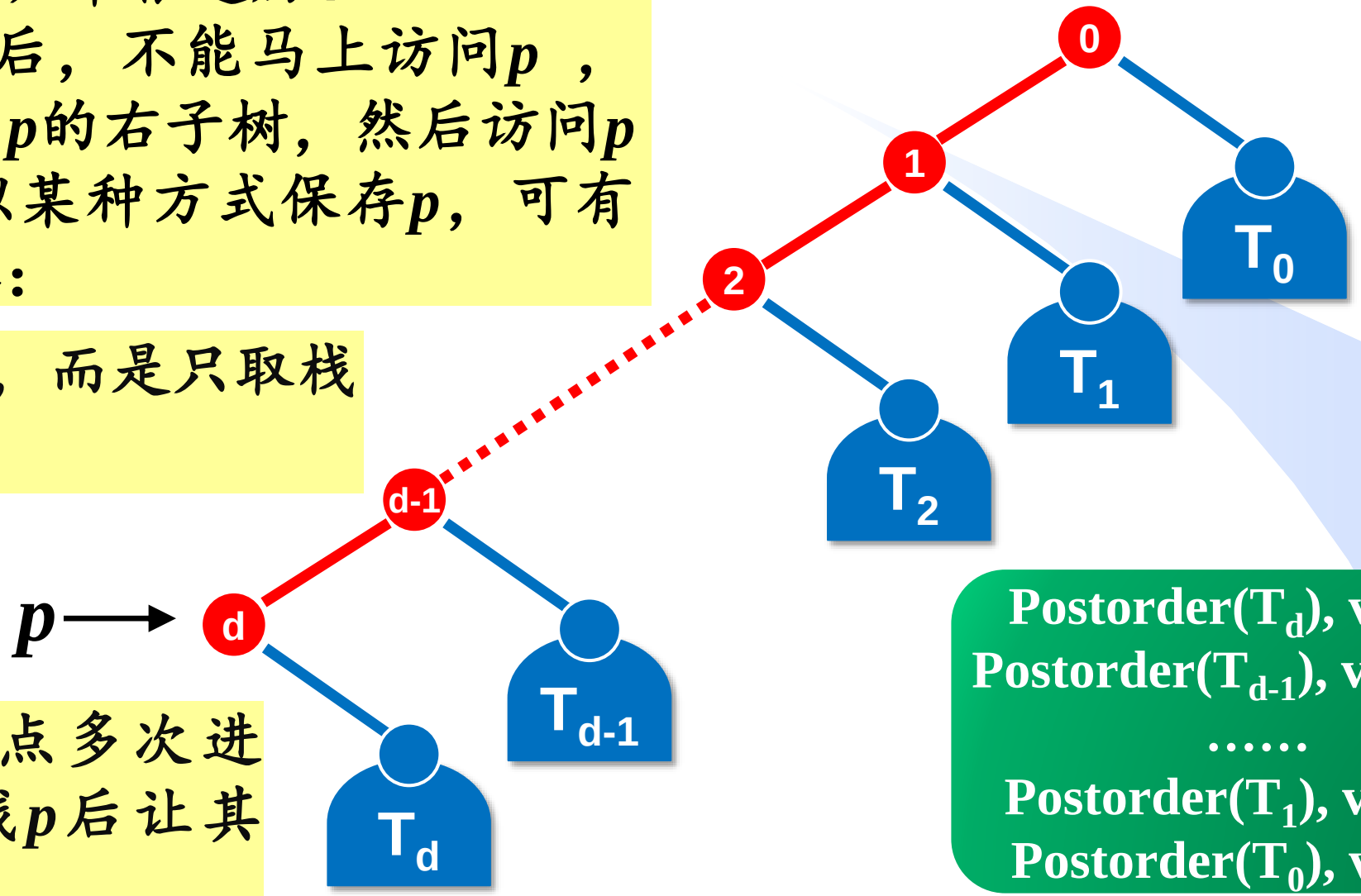
非递归后根遍历算法

可否仿造非递归中根遍历？

问题：弹栈p后，不能马上访问p，而是先要访问p的右子树，然后访问p。所以需要以某种方式保存p，可有如下两种策略：

策略1 不弹栈p，而是只取栈顶元素值。

策略2 允许结点多次进出栈，即弹栈p后让其马上再进栈。

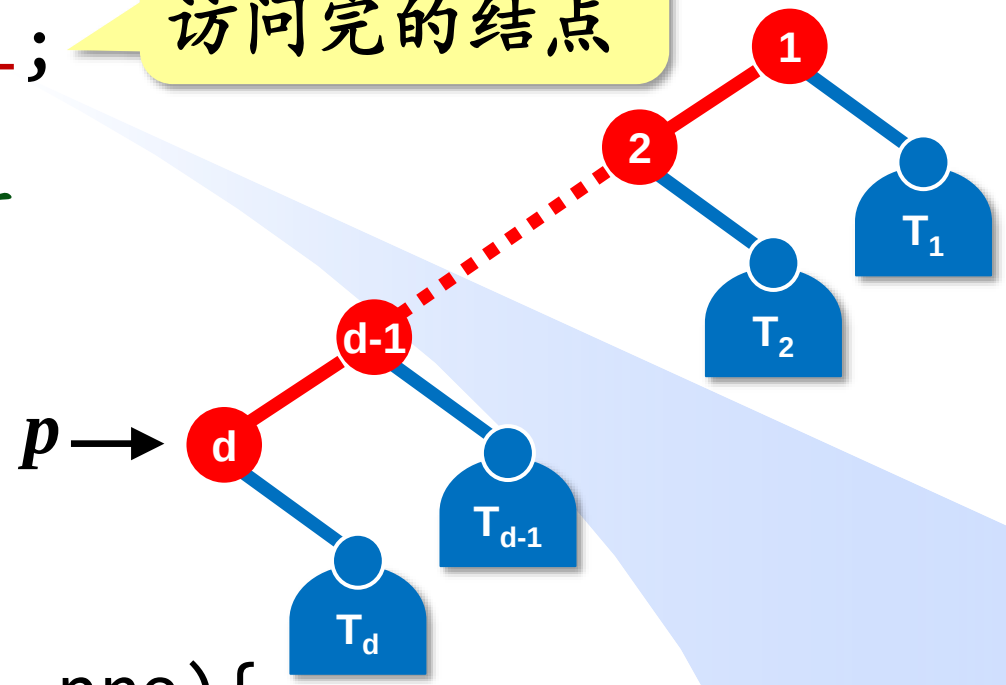


Postorder(T_d), visit(d)
Postorder(T_{d-1}), visit(d-1)
……
Postorder(T_1), visit(1),
Postorder(T_0), visit(0)

非递归后根遍历算法（版本1）

```
void NonRecPostOrder(TreeNode* t){
    Stack S; TreeNode *p=t, *pre=NULL;
    while (true){
        while(p!=NULL){  //沿左分支下行
            S.Push(p);
            p=p->left;
        }
        if(S.IsEmpty()) return;
        p=S.Peek();
        if(p->right==NULL || p->right==pre){
            p=S.Pop(); visit(p->data); pre=p;  p=NULL;
        }else
            p=p->right;
    }
}
```

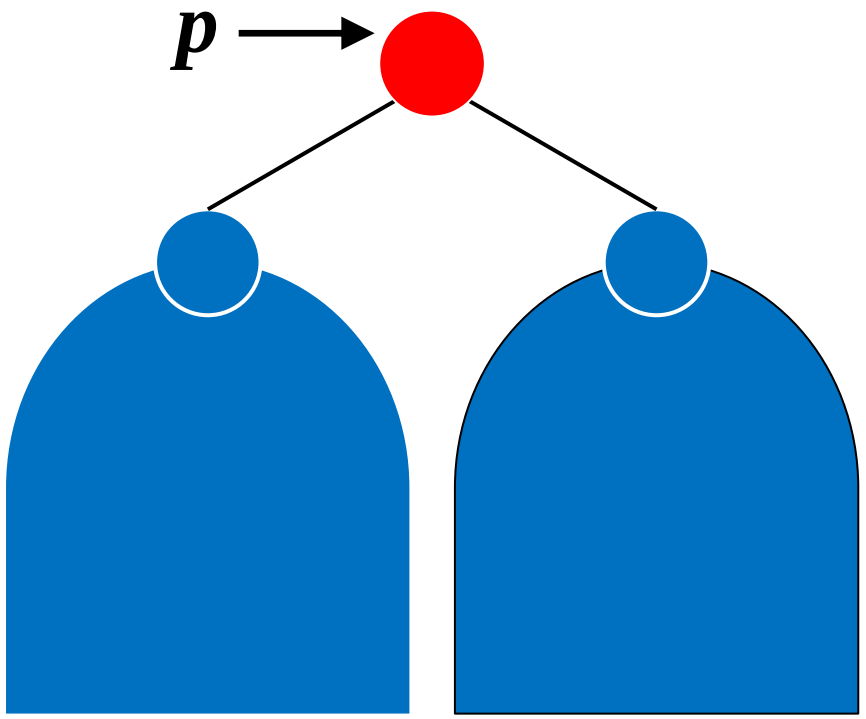
*pre*是上一步刚访问完的结点

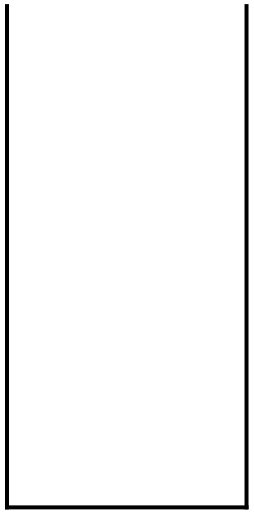


*p*无右子树或*p*的右子树刚访问完，此时应访问*p*

非递归后根遍历算法（版本2）

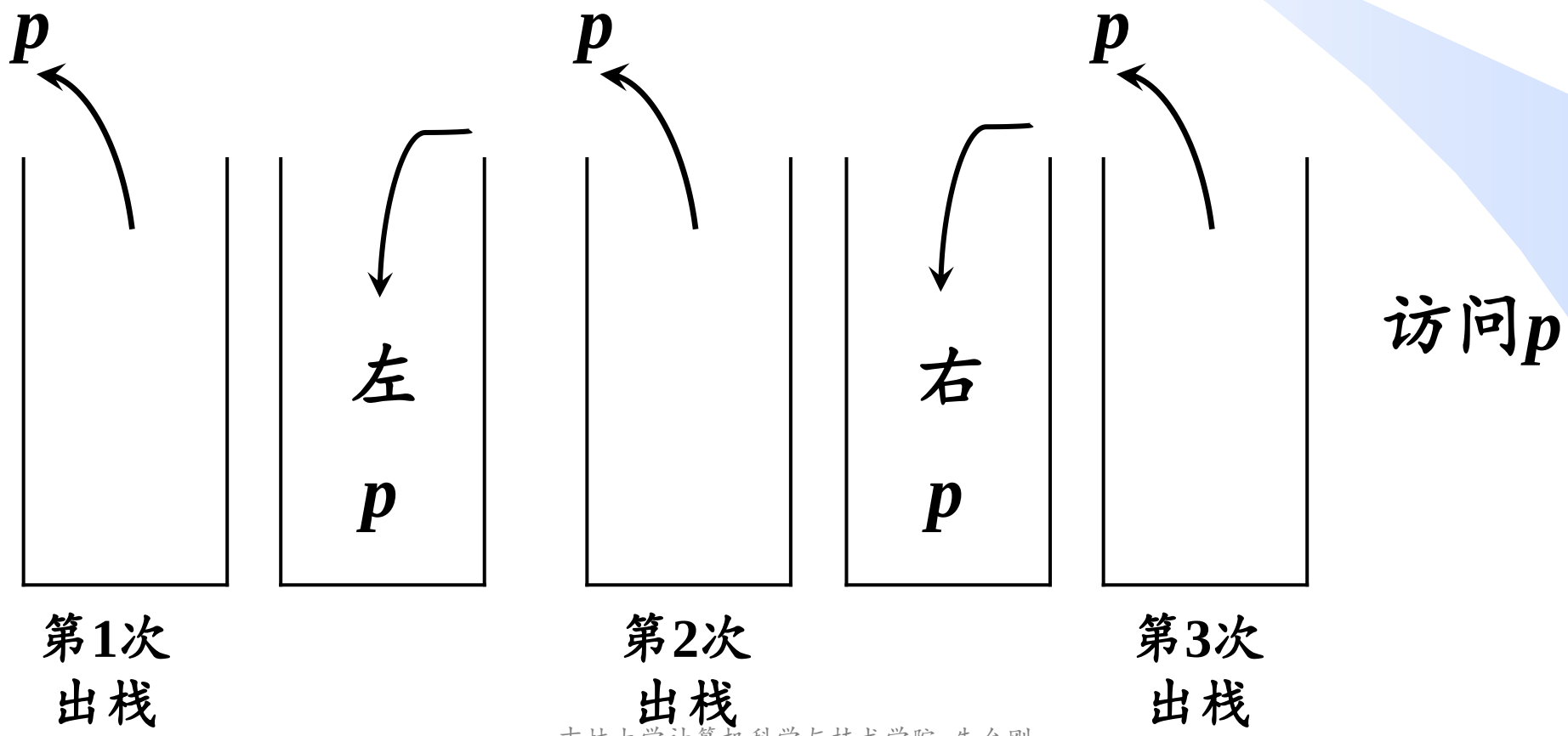
允许结点多次进出栈，栈元素增加关于进栈/出栈次数的信息。





非递归后根遍历算法（版本2）

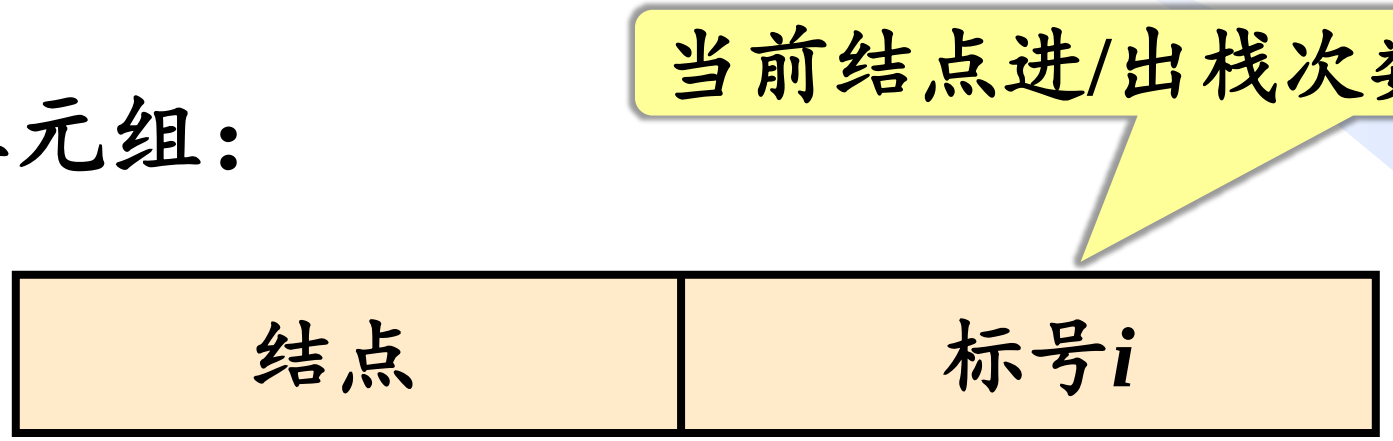
允许结点多次进出栈，栈元素增加关于进栈/出栈次数的信息。



非递归后根遍历算法（版本2）

允许结点多次进出栈，栈元素增加关于进/出栈次数的信息。

栈元素为二元组：



出栈时：

$i=1$（第1次出栈）：没有访问结点的任何子树，准备遍历其左子树；

$i=2$（第2次出栈）：遍历完左子树，准备遍历其右子树；

$i=3$（第3次出栈）：遍历完右子树，准备访问该结点。

非递归后根遍历算法（版本2）

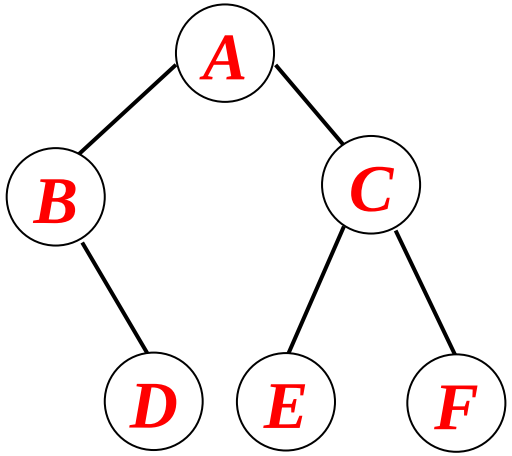
```
const int MaxSize = 1e5+10;
class Stack{
public:
    void Push(TreeNode *p, int i){cnt[++top]=i; node[top]=p;}
    void Pop(TreeNode *&p, int &i){p=node[top]; i=cnt[top--];}
    bool Empty() { return top == -1; }
    bool Full() { return top == MaxSize - 1; }
private:
     TreeNode* node[MaxSize];
     int cnt[MaxSize];
     int top = -1;
};
```

非递归后根遍历算法（版本2）

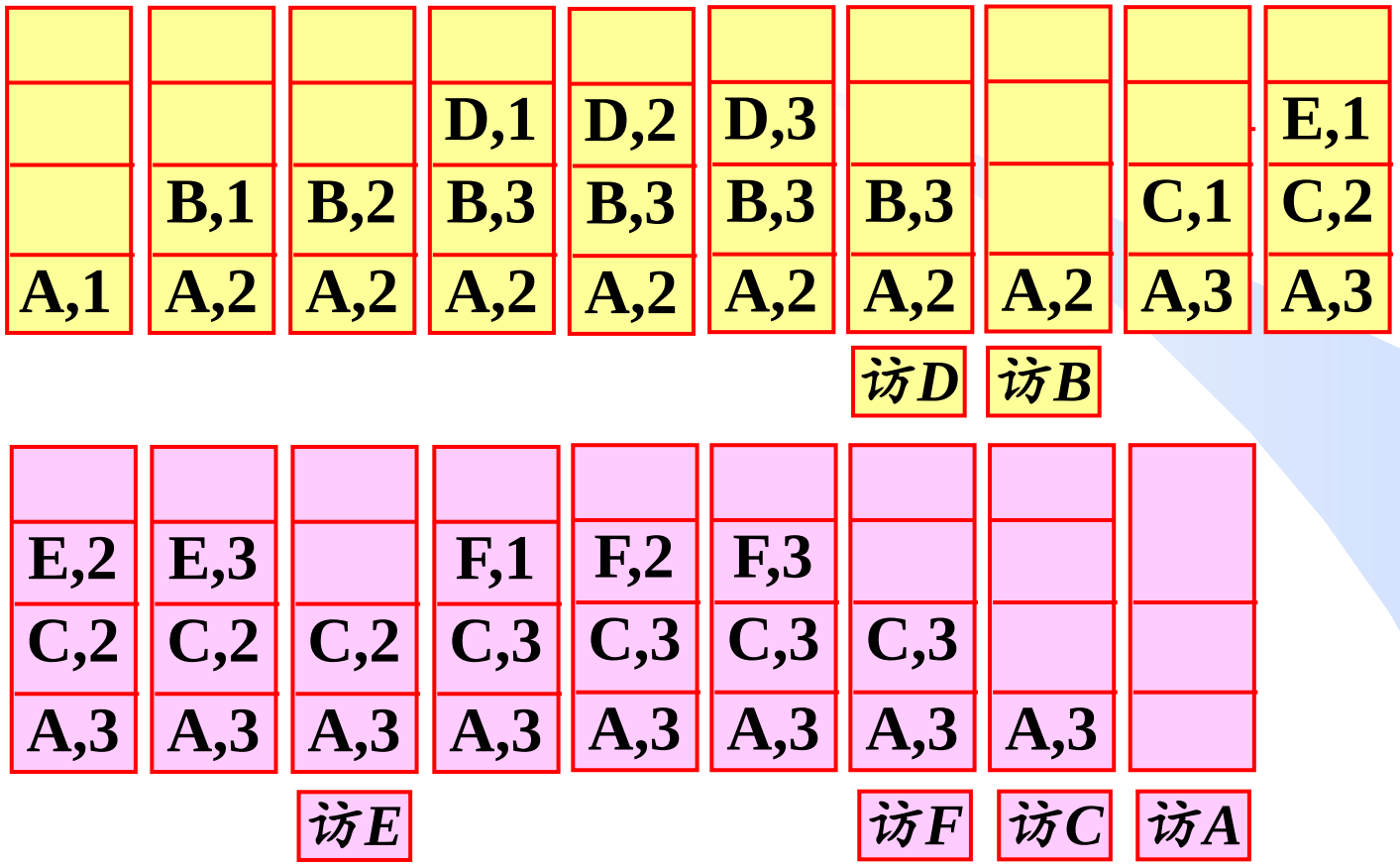
```
void NonRecPostorder2(TreeNode* t) {
    Stack S;
    if(t!=NULL) S.Push(t,1);
    while(!S.Empty()){
        TreeNode *p; int i;
        S.Pop(p,i);
        if(i==1){S.Push(p,2); if(p->left!=NULL) S.Push(p->left,1);}
        if(i==2){S.Push(p,3); if(p->right!=NULL) S.Push(p->right,1);}
        if(i==3) visit(p->data);
    }
}
```

$i = 1$: 准备遍历其左子树;
$i = 2$: 遍历完左子树, 准备遍历其右子树;
$i = 3$: 遍历完右子树，准备访问该结点。

运行实例：留做作业



非递归后根遍历算法（版本2）对上图二叉树进行后根遍历，栈的变化

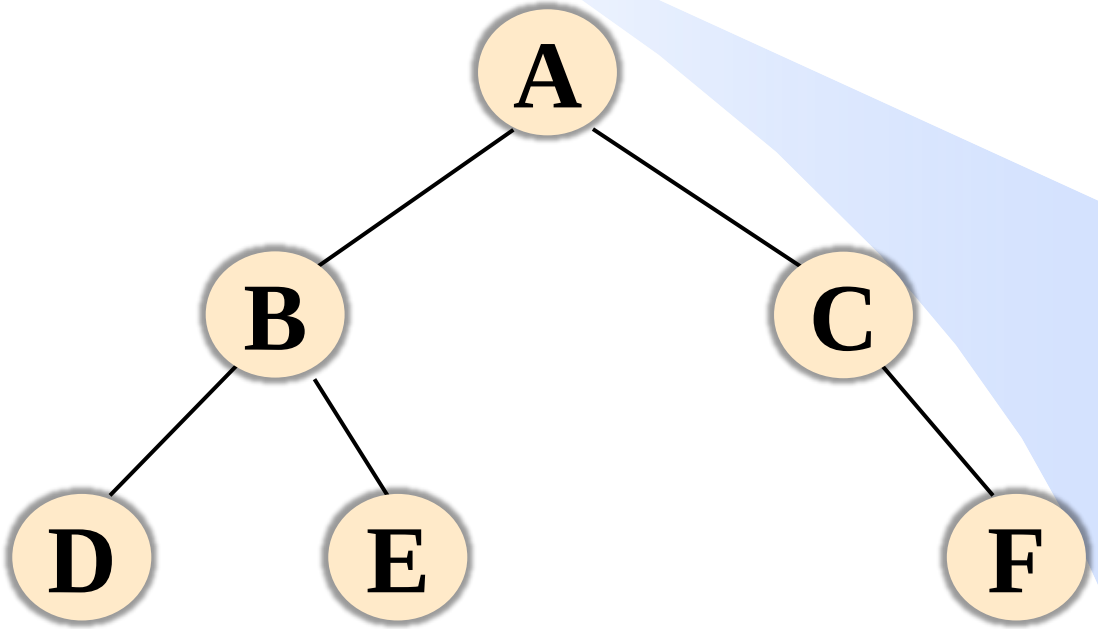


层次遍历——定义

按层数由小到大，同层由左向右的次序访问结点。

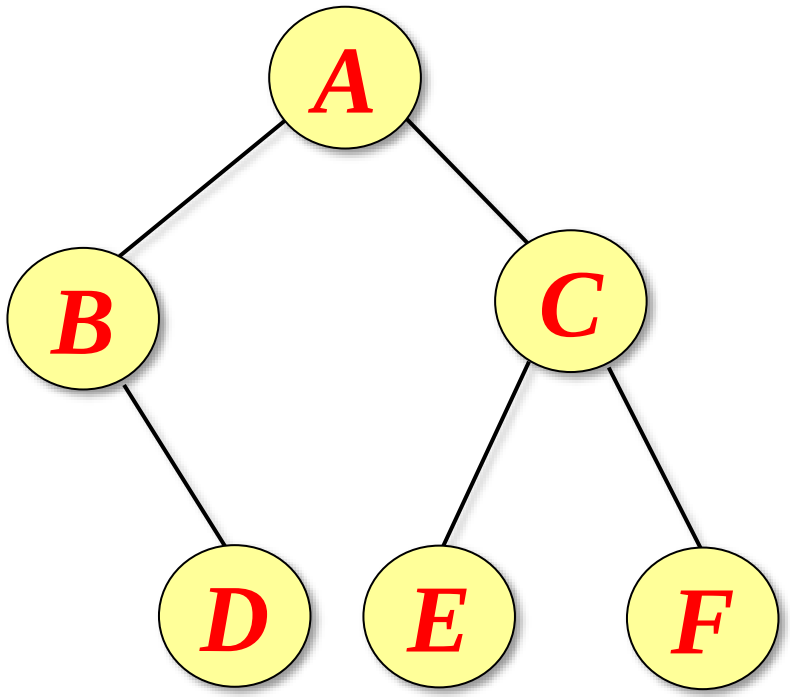
遍历结果:

A B C D E F



层次遍历——实现

- 通过观察发现，在第i层上若结点x在结点y的左边，则x一定在y之前被访问。
- 并且，在第$i+1$层上，x的子结点一定在y的子结点之前被访问。
- 用一个队列来实现。



A入队

A出队，B、C入队

B出队，D入队

C出队，E、F入队

D出队

E出队

F出队

A			
B	C		
C	D		
D	E	F	
E	F		
F			

出队即访问

层次遍历——实现

二叉树层次遍历算法需要一个辅助队列，具体方法如下：

- 根结点入队。
- 重复本步骤直至队为空：
  出队一个结点并访问；
  若其有左孩子，则将其左孩子入队；
  若其有右孩子，则将其右孩子入队。

二叉树层次遍历——实现

```
void LevelOrder(TreeNode *t){
  Queue Q;
  if(t!=NULL) Q.Enque(t);
  while(!Q.Empty()){
      TreeNode* p=Q.Dequeue();
      visit(p->data);
      if(p->left!=NULL) Q.Enque(p->left);
      if(p->right!=NULL) Q.Enque(p->right);
  }
}
```

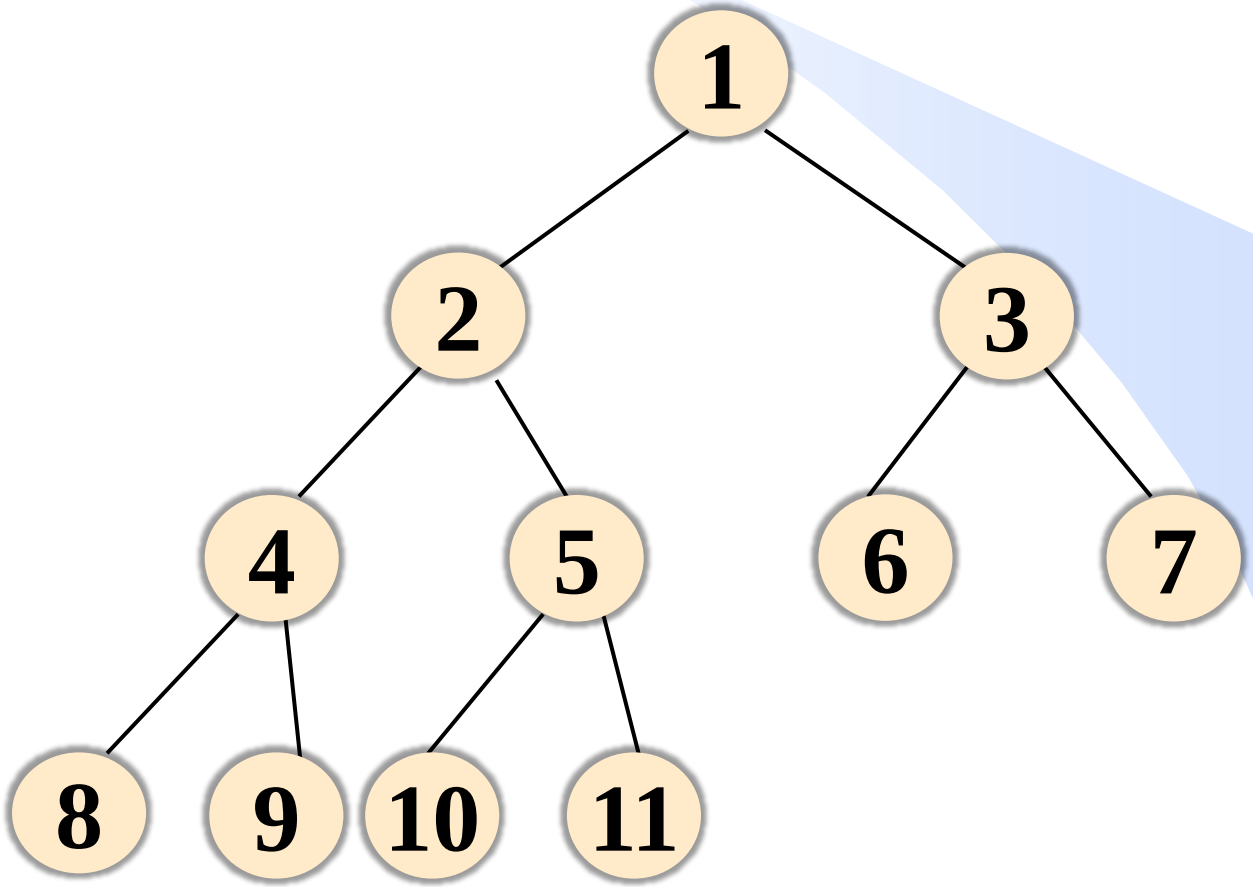
辅助队列的规模

若对由2021个结点构成的完全二叉树进行层次遍历，辅助队列的容量至少需要多大（即同时保存在队列中的结点的最大个数）。【清华大学、吉林大学期末考试题】

辅助队列的规模

若对由*n*个结点构成的完全二叉树进行层次遍历，辅助队列的容量至少需要多大（即同时保存在队列中的结点的最大个数）。

最大规模：$\lceil n/2 \rceil$

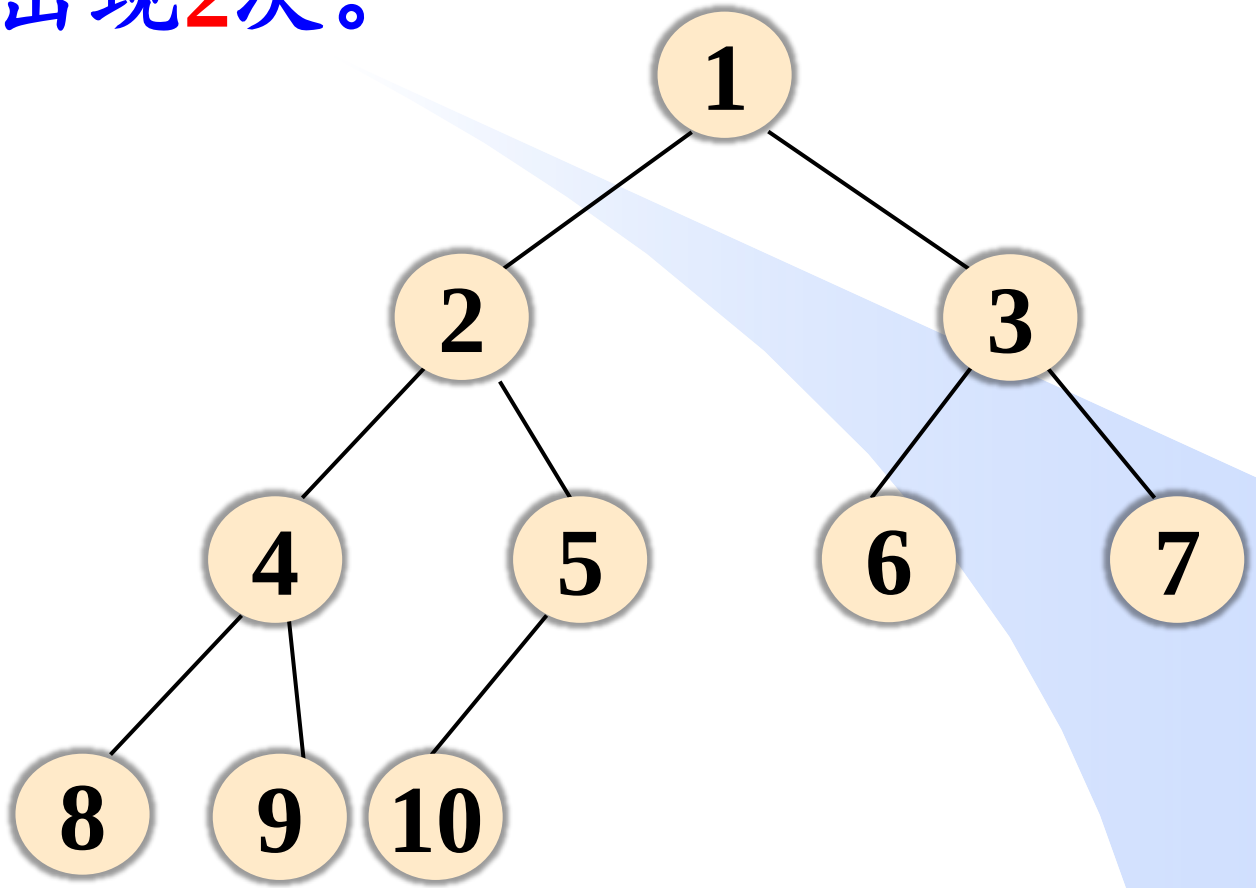


辅助队列的规模

课下思考：最大规模可能出现2次。

5	6	7	8	9

6	7	8	9	10



课下思考

判断题：

对叶结点数量为2018的二叉树进行层次遍历，辅助队列的最大规模（同时保存在队列中的结点个数）绝对不超过2018.【清华大学考研题】

统计二叉树每层结点信息

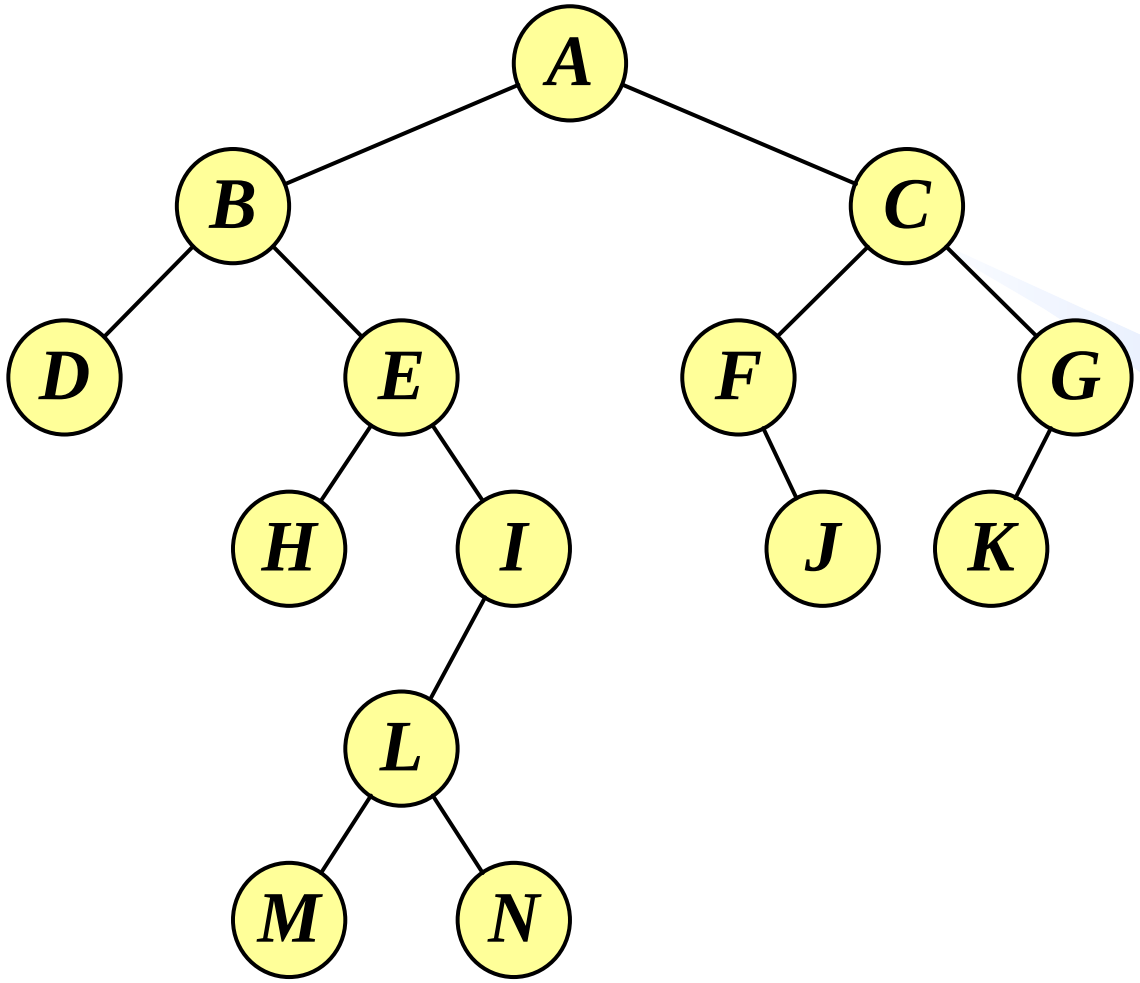
统计一棵二叉树中每层叶结点的数目【吉林大学上机考试题】。

➢需要识别每层的结束，每层结束后，统计该层信息。

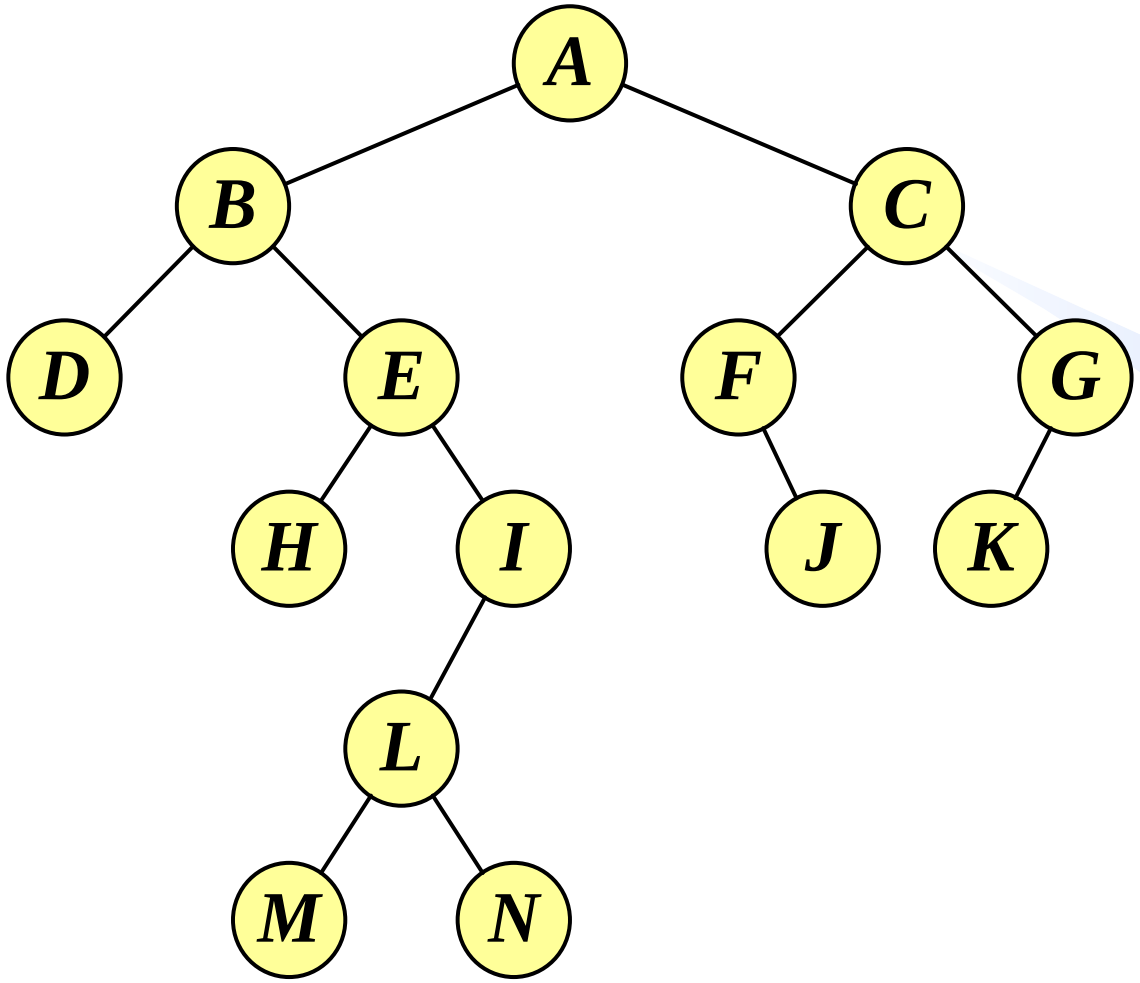
方案1:

✓可以设置一个不同于队列中其他结点的特殊结点（比如空结点NULL）来表示每层的结束。

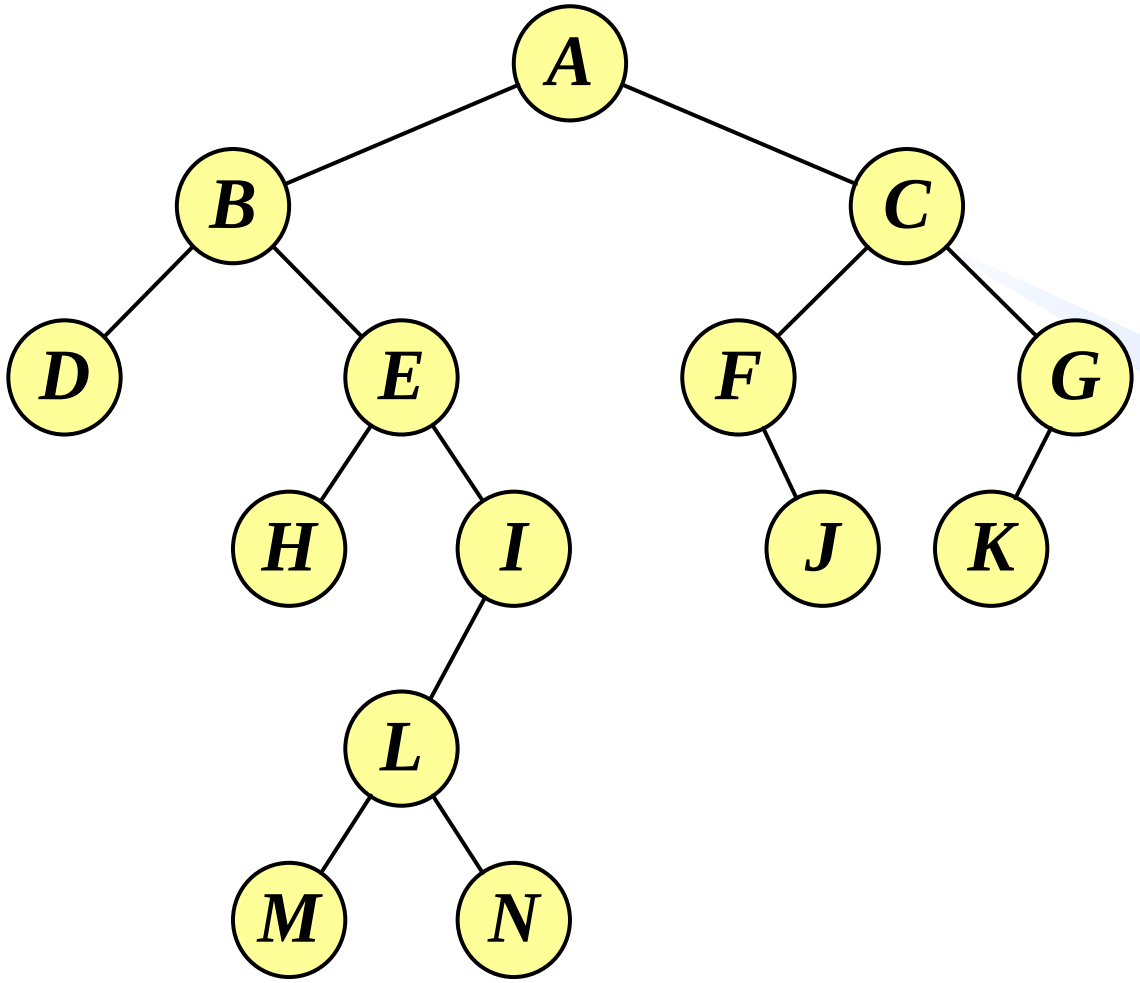
✓遍历第0层时，先将根结点入队，再将NULL入队。在遍历过程中，当NULL出队时，表示已经遍历完本层，将NULL再入队，此时NULL即为下一层的结尾。



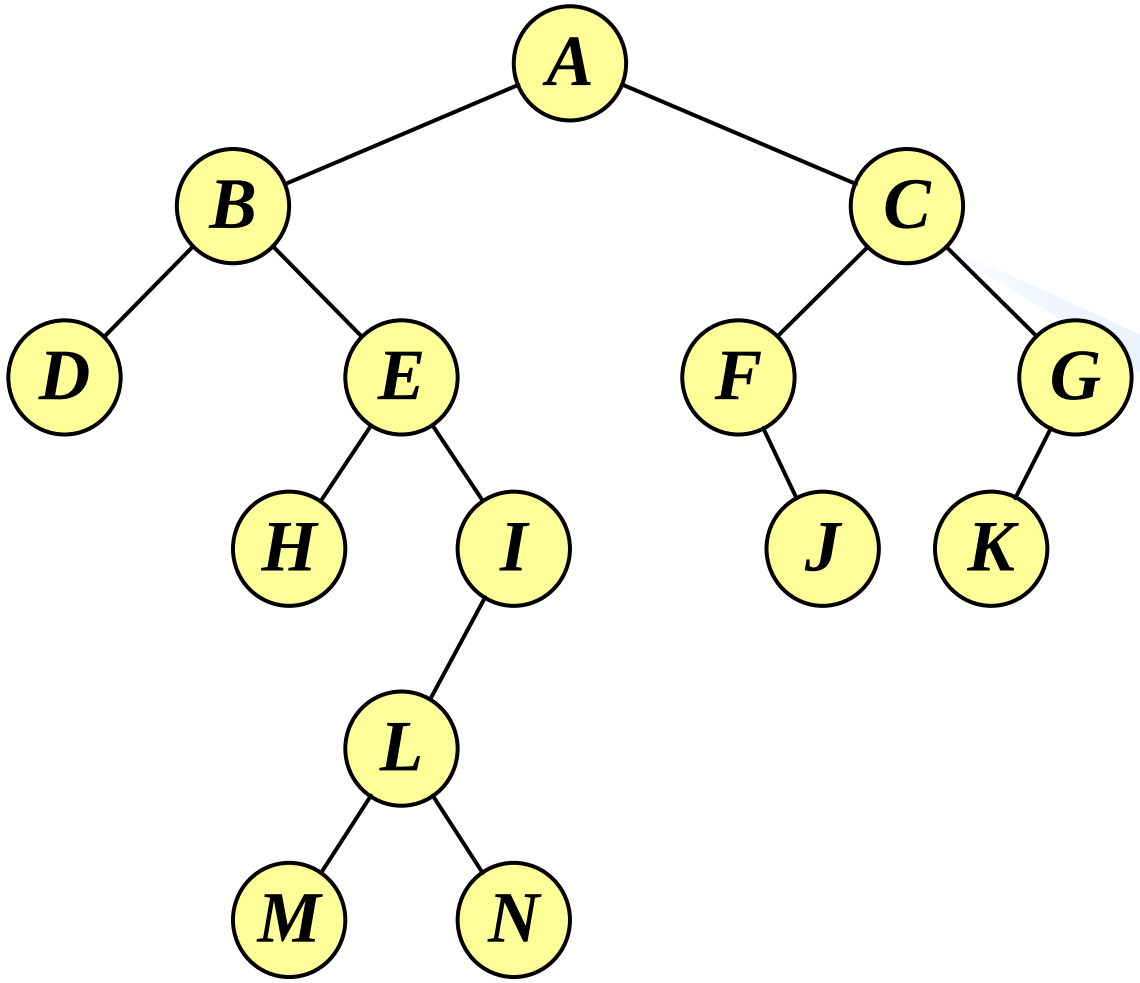
A	Λ



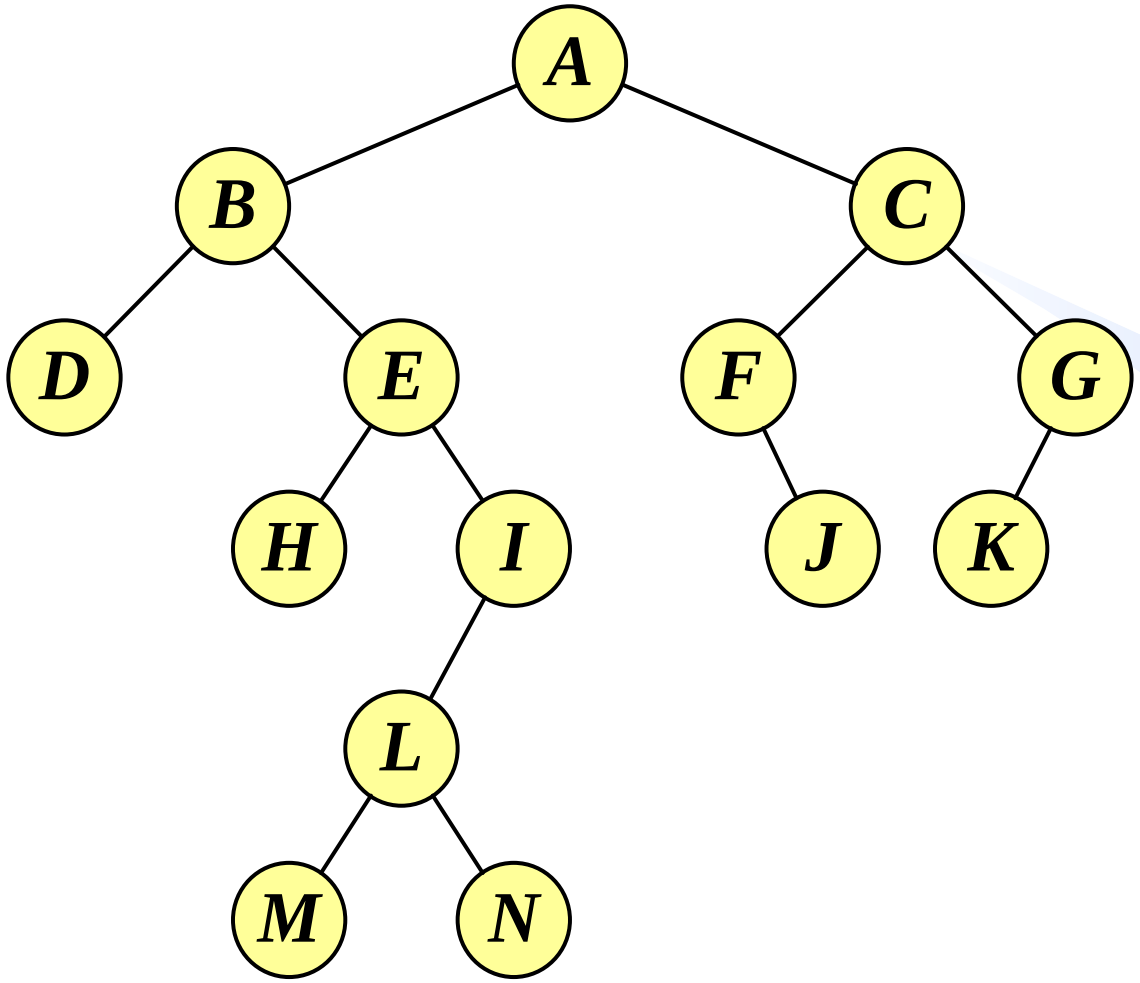
Λ	B	C



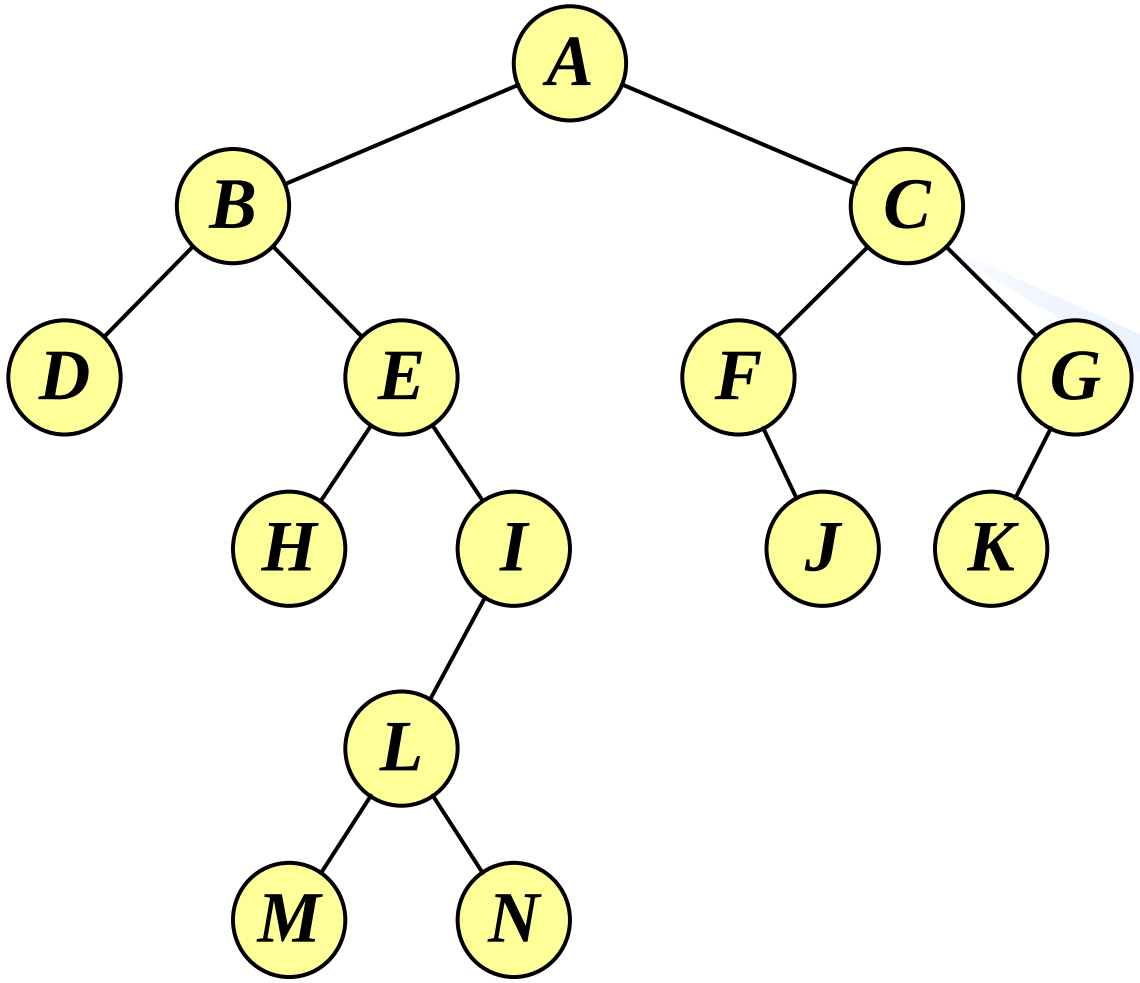
B	**C**	Λ



C　　Λ　　D　　E



Λ	D	E	F	G



D	E	F	G	Λ

```
int LeafInEachLevel (TreeNode *root, int leafnum[]){
    if(root==NULL) return -1;
    Queue q ; int level=0;
    q.Enqueue(root);
    q.Enqueue(NULL); //NULL入队
    while (!q.Empty()){
        TreeNode * p=q.Dequeue(); //出队一个结点
        if(p==NULL) {                    //本层结束
            level++;
            if(!q.Empty()) q.Enqueue(NULL);
        }else{
            if(p->left==NULL && p->right==NULL) leafnum[level]++;
            if(p->left!=NULL) q.Enqueue(p->left);
            if(p->right!=NULL) q.Enqueue(p->right);
        }
    }
    return level-1; //返回二叉树的高度
};
```

***leafnum[i]*表示第*i*层的叶结点数。调用时*leafnum*数组各元素值应初始化为0。**

统计二叉树每层结点信息

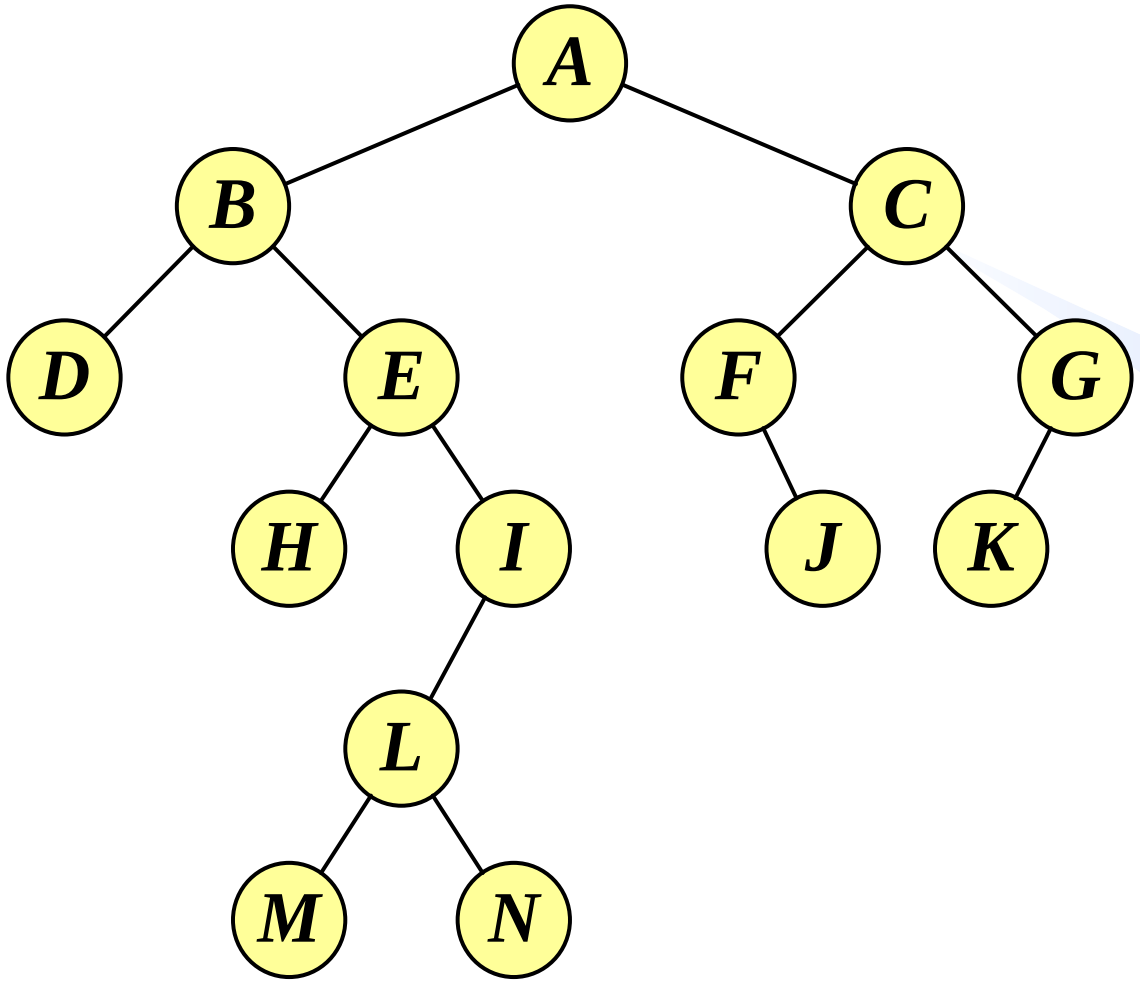
统计一棵二叉树中每层叶结点的数目【吉林大学上机考试题】。

➢需要识别每层的结束，每层结束后，统计该层信息。

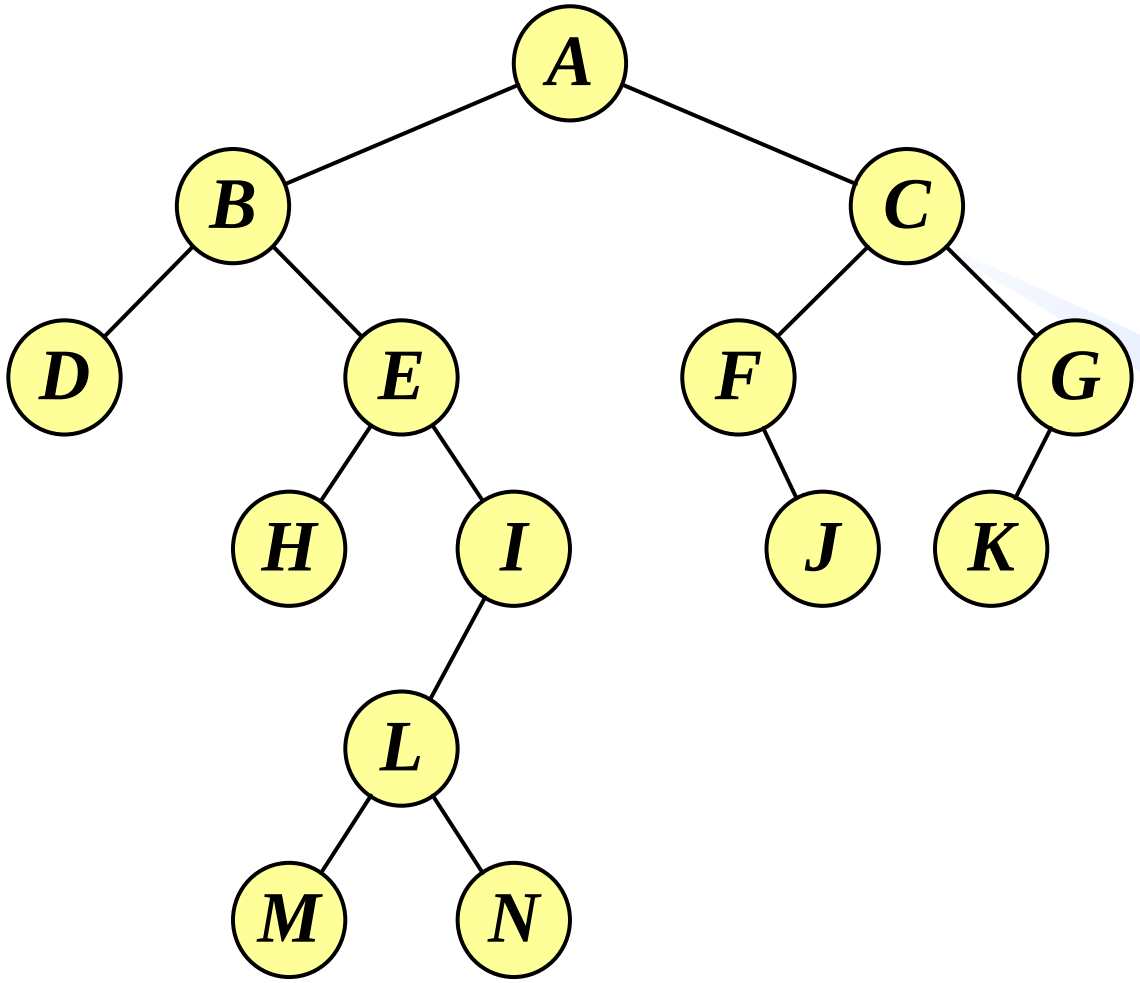
方案2:

✓每层结点全部出队后，队列中的元素即为下一层的全部结点

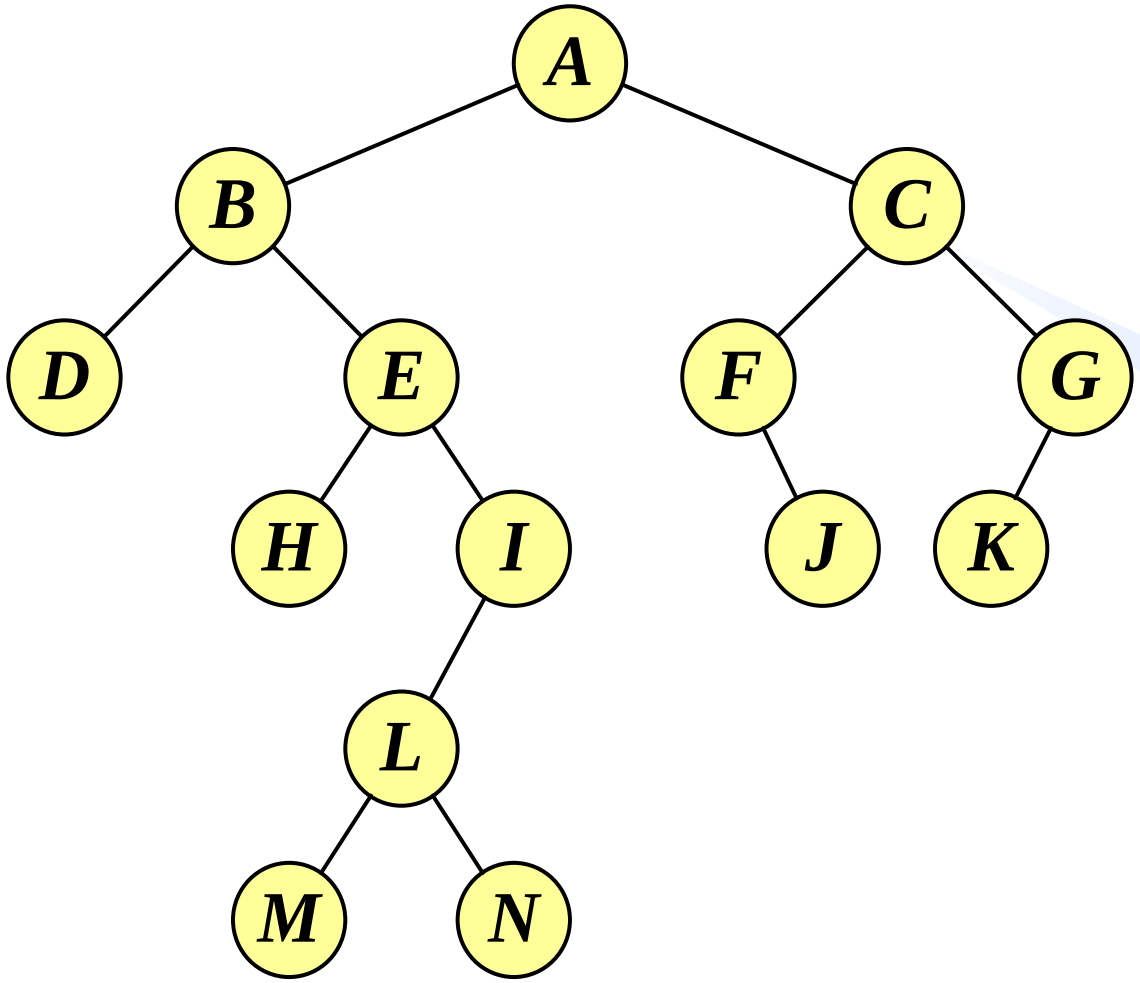
✓出队时，在while循环里面做个for循环，根据队列中元素个数，把同层结点连续出队。



A



B　　C



D　　E　　F　　G

```
int LeafInEachLevel (TreeNode *root, int leafnum[]){
    if(root==NULL) return -1;
    Queue q ; int level=0;
    q.Enqueue(root);
    while (!q.Empty()){
        int size=q.size(); //size为本层结点数
        for(int i=0; i<size; i++){ //连续出队并访问本层结点
            TreeNode * p=q.Dequeue();
            if(p->left==NULL && p->right==NULL)leafnum[level]++;
            if(p->left!=NULL) q.Enqueue(p->left);
            if(p->right!=NULL) q.Enqueue(p->right);
        }
        level++;    //本层结束，准备访问下一层
    }
    return level-1; //返回二叉树的高度
};
```

***leafnum[i]*表示第*i*层的叶结点数。调用时*leafnum*数组各元素值应初始化为0**

统计二叉树每层结点信息

统计一棵二叉树中每层叶结点的数目【吉林大学上机考试题】。

➢需要识别每层的结束，每层结束后，统计该层信息。

方案3：

✓采用先根遍历，用变量*k*标识递归深度，每进入一层递归，*k*加1，递归深度*k*其实就是当前结点所在的层数.

✓用数组*leafnum*[]记录各层叶节点数目，每遍历到一个叶结点时，*leafnum*[*k*]++，表示第*k*层叶节点数目加1.

✓当遍历完成后，*leafnum*[*i*]数组就存储了第*i*层叶节点数目。

```
void LeafInEachLevel(TreeNode *root,int k,int leafnum[],int &h){
    //计算每层叶结点数存入leafnum[]数组,h为二叉树高度即最大层数
    //k为递归深度，每往深递归一层k加1，k亦为当前访问结点所在层数
    if(root==NULL) return; //空树高度为-1
    if(k>h) h=k; //若当前层数超过最大树高，则更新树高
    if(p->left==NULL && p->right==NULL) leafnum[k]++;
    LeafInEachLevel(t->left, k+1, leafnum, h);
    LeafInEachLevel(t->right, k+1, leafnum, h);
}
```

初始调用

```
int h=-1;  //h为遍历过程中遇到的最大层数，初值≤0
LeafInEachLevel(root, 0, leafnum, h)
```

//leafnum[i]表示第i层的叶结点数，初始调用前数组各元素值初始化为0

总结

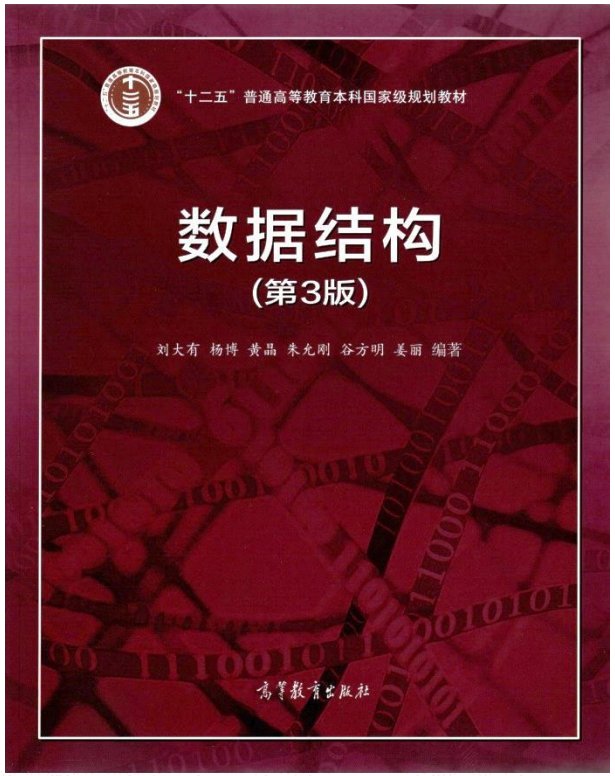
先根遍历

中根遍历

后根遍历

} 深度优先搜索

层次遍历 —— 广度优先搜索



二叉树的存储和操作

- 二叉树的存储结构
- 二叉树遍历及其递归算法
- 遍历的非递归算法
- **二叉树的重建和计数**
- 二叉树其他操作

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

二叉树的重建

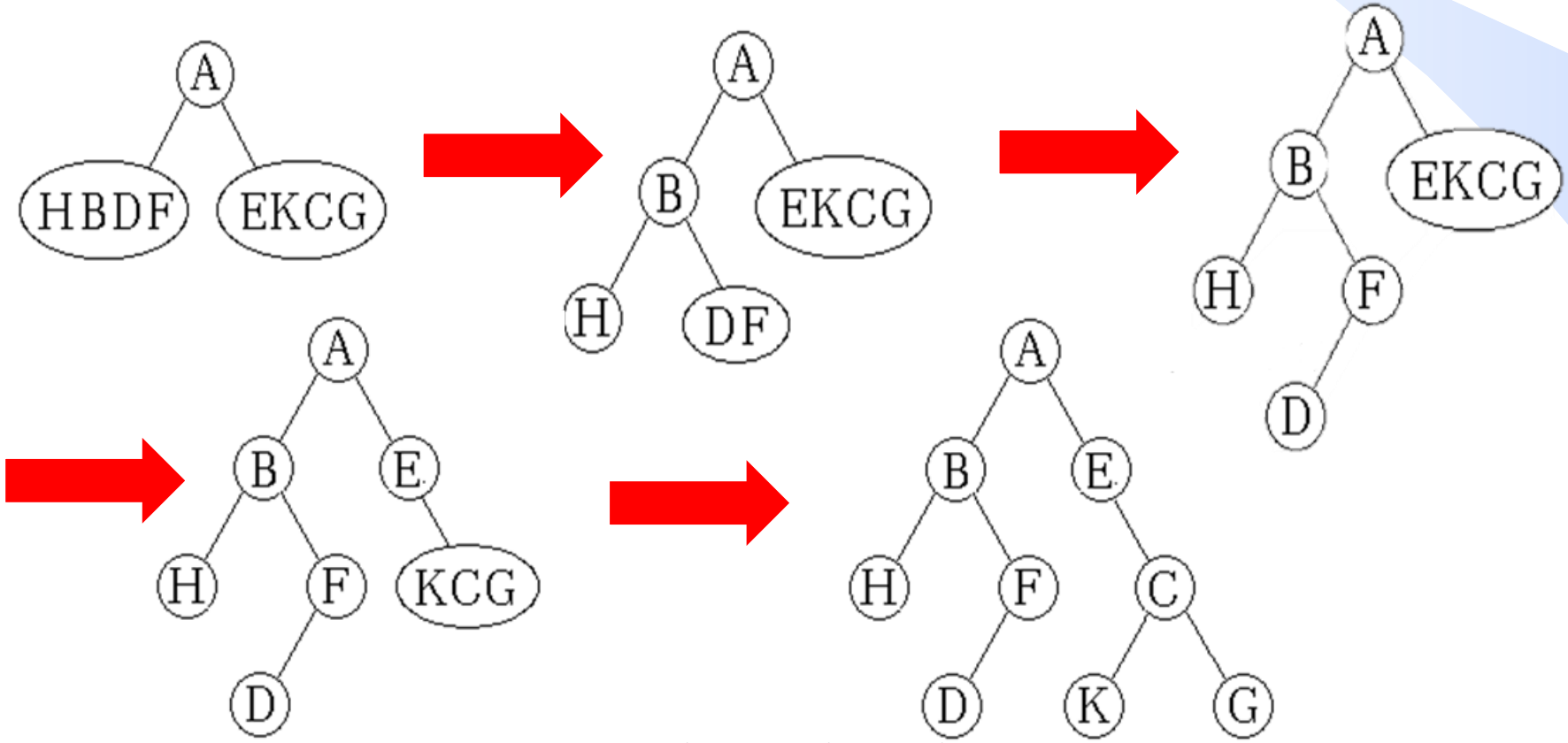
由先根序列和中根序列可否唯一确定一棵二叉树？

[例] 先根序列 A B H F D E C K G

中根序列 H B D F A E K C G

通过先根序列确定子树的根

通过中根序列确定左右子树

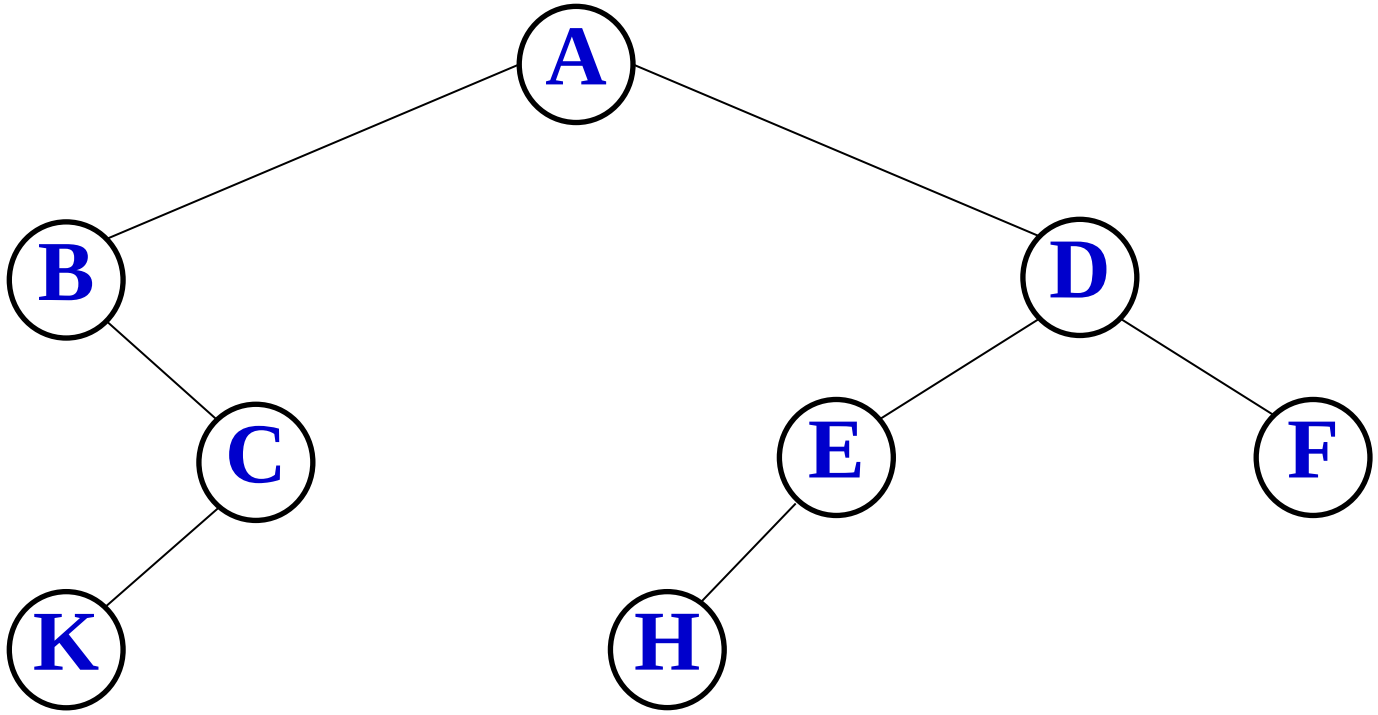


课下练习

由以下先根序列和中根序列确定一棵二叉树

先根序列　A B C K D E H F

中根序列　B K C A H E D F



课下练习

已知一棵二叉树的先序和中序遍历序列如下：

先序序列：A B C D E F G H I J

中序序列：C B A E F D I H J G

则其后序遍历序列为_____【阿里笔试题】

A. C B D E A G I H J F

B. C B D A E G I H J F

C. C E D B I J H G F A

D. C E D B I H J G F A

E. C B F E I J H G D A

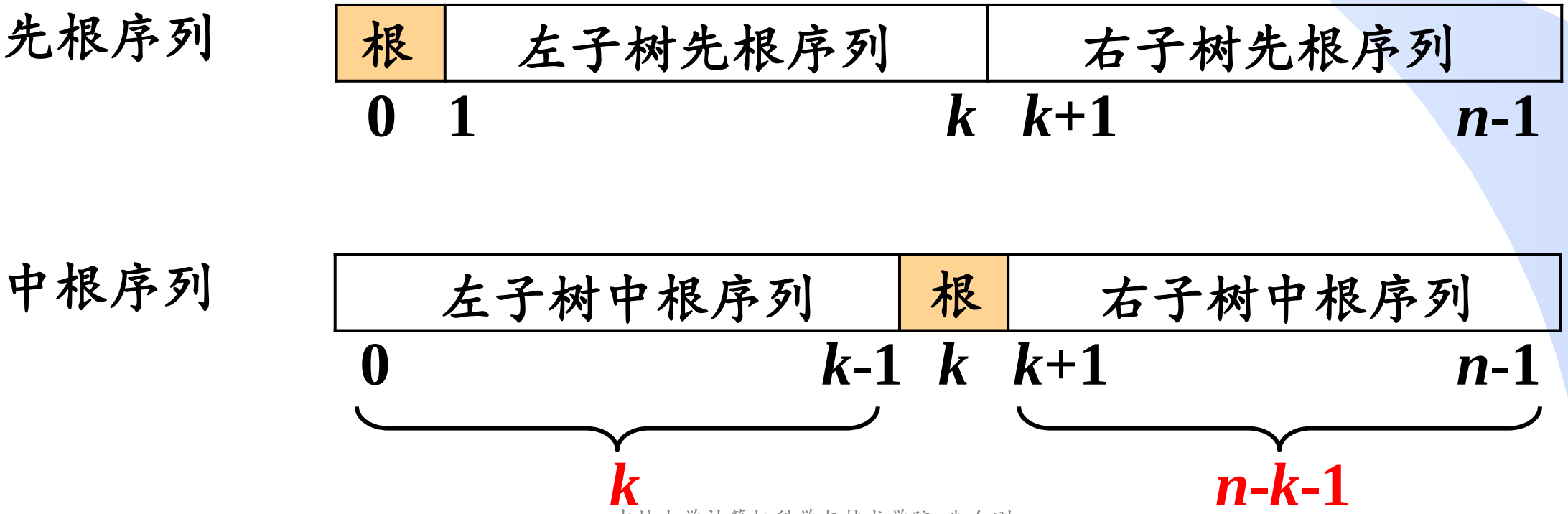
F. C B F E I H J G D A

二叉树的重建——编程实现

给定二叉树的先根序列和中根序列，编写程序构建二叉树。

【大厂面试题、华中科技大学考研复试机试题、OpenJudgeP0570、LeetCode105】

```
TreeNode* buildTree(char *preorder, char *inorder, int n)
```



二叉树的重建——编程实现

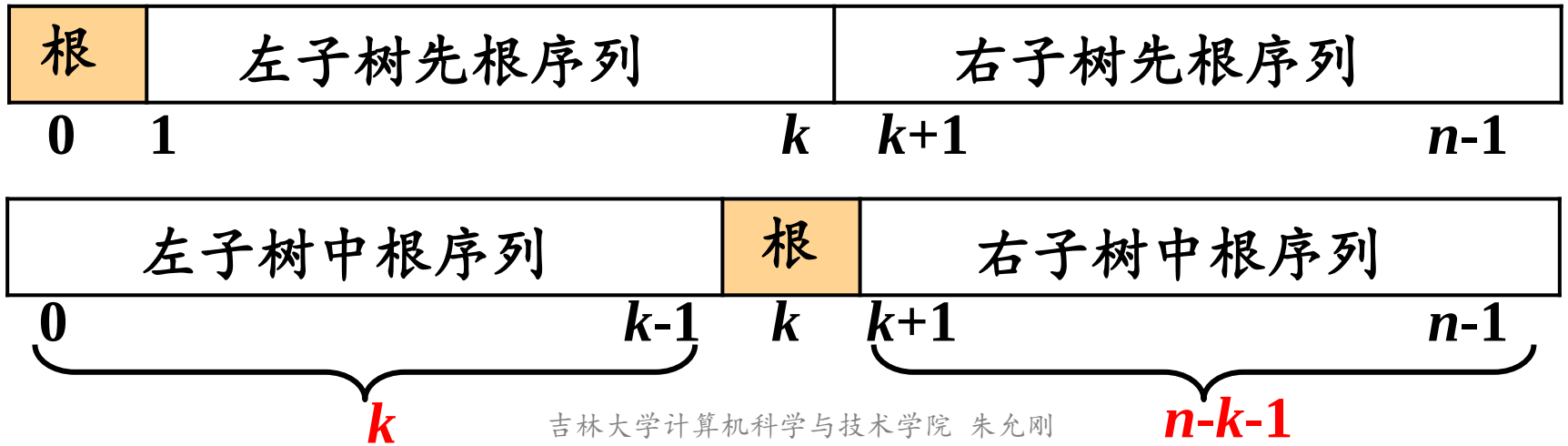
```
TreeNode* buildTree(char *preorder, char *inorder, int n) {
    if (n <= 0) return NULL;
    char rootval = preorder[0]; //先根序列的根
    TreeNode *root = new TreeNode;
    root->data = rootval;
    int k = find(inorder, n, rootval); //在中根序列找根
```

```
int find(char *inorder, int size, char val) {
    for(int i = 0; i<size; i++)
        if(inorder[i] == val) return i;
    return -1;
}
```

二叉树的重建——编程实现

```
TreeNode* buildTree(char *preorder, char *inorder, int n) {
    if (n <= 0) return NULL;
    char rootval = preorder[0]; //先根序列的根
    TreeNode *root = new TreeNode;
    root->data = rootval;
    int k = find(inorder, n, rootval); //在中根序列找根
    root->left=buildTree(&preorder[1], &inorder[0], k);
    root->right=buildTree(&preorder[k+1],&inorder[k+1],n-k-1);
    return root;
}
```

最好/平均时间复杂度O(*n*log*n*)
最坏时间复杂度O(n^2)

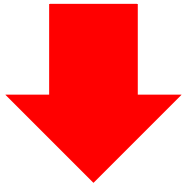


二叉树的计数

*n*个结点的二叉树有多少种的形态？

- 对二叉树的*n*个结点进行编号，不妨按先根序列进行编号1...*n*。

- 因中根序列和先根序列能唯一确定一棵二叉树，故二叉树有多少个可能得中根序列，就能确定多少棵不同的二叉树。



二叉树先根序列为1...*n*时，有多少种可能的中根序列

非递归先根遍历

```
void NonRecPreOrder(TreeNode* t){
    Stack S; TreeNode *p = t;
    while(true) {
        while(p != NULL){
            visit(p->data);
            S.PUSH(p);
            p=p->left;
        }
        if(S.IsEmpty()) return;
        p=S.POP();
        p=p->right;
    }
}
```

非递归中根遍历

```
void NonRecInOrder(TreeNode *t){
   Stack S; TreeNode *p = t ;
   while (true) {
      while (p != NULL) {
         S.PUSH(p);
         p=p->left;
      }
      if(S.IsEmpty()) return;
      p=S.POP();
      visit(p->data);
      p=p->right;
   }
}
```

先根遍历的进栈/出栈序列=中根遍历的进栈/出栈序列

看先根算法：结点进栈顺序就是先根访问的顺序，即进栈序列=先根序列

看中根算法：结点出栈顺序就是中根访问的顺序，即出栈序列=中根序列

二叉树先根序列为1...*n*时，有多少种可能的中根序列

*n*个结点的二叉树有多少种的形态？

二叉树先根序列为1...*n*时，有多少种可能的中根序列

对于进栈序列1...*n*，有多少种可能的合法出栈序列

$$\text{Catalan}(n) = \frac{1}{n+1} C_{2n}^{n}$$

练习

先根序列为a, b, c, d的不同二叉树的个数是（ B ）

【考研题全国卷】

A. 13　　B. 14

C. 15　　D. 16

$$\text{Catalan}(n) = \frac{1}{n+1} C_{2n}^{n}$$

练习

对于一个栈，若其入栈序列为1，2，3,…,*n*，其合法的出栈序列个数正好等于包含*n*个结点的二叉树的个数，且与不同形态的二叉树一一对应。请简要叙述一种从“入栈序列（固定为1，2，3,…,*n* ）/出栈序列”对应一种“二叉树形态”的方法，并举例。【浙江大学考研题】

入栈序列 = 先根序列

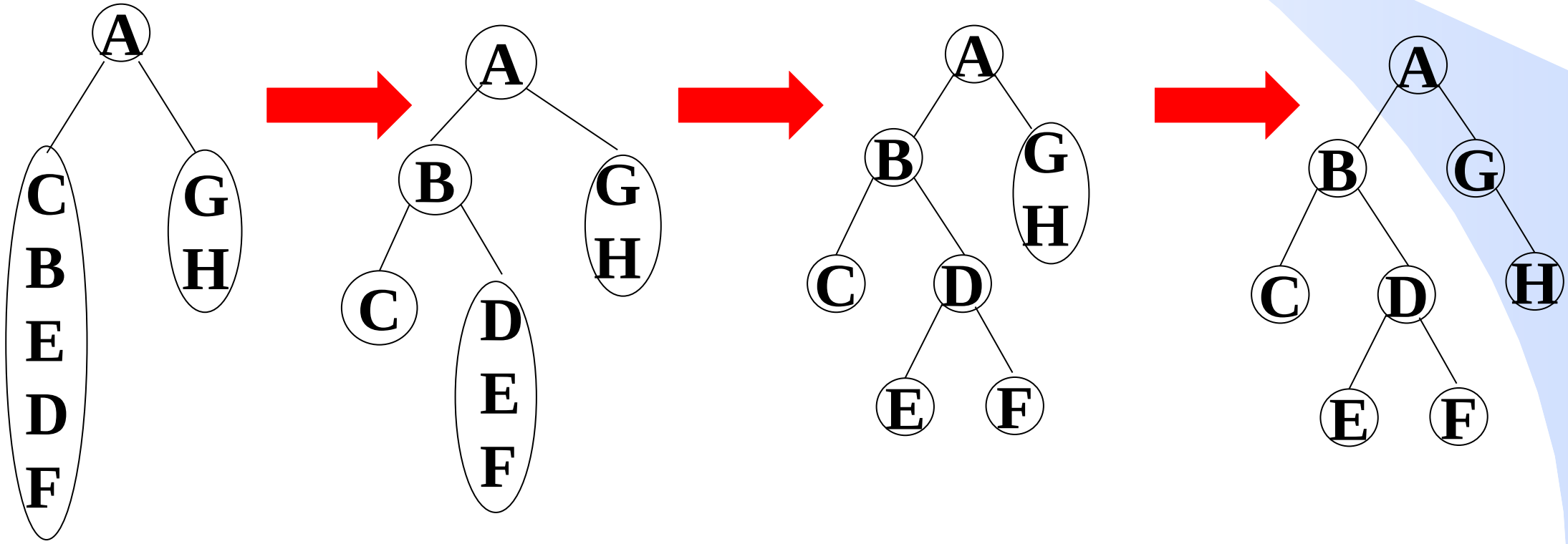
出栈序列 = 中根序列

二叉树的重建

由后根序列和中根序列是否可以唯一地确定一棵二叉树？

后根序列　C E F D B H G A

中根序列　C B E D F A G H



- 由先根序列和后根序列是否可以唯一地确定一棵二叉树？

- 先根序列：A B
  后根序列：B A

- 由层次遍历序列可否唯一确定一棵二叉树？

- 层次序列：AB

课下思考

- 利用完全二叉树的先根序列能否唯一确定一棵完全二叉树?
- 利用完全二叉树的中根序列能否唯一确定一棵完全二叉树?
- 利用完全二叉树的后根序列能否唯一确定一棵完全二叉树?
- 利用完全二叉树的层次序列能否唯一确定一棵完全二叉树?

由先根序列和层次遍历序列可否唯一确定一棵二叉树？

- 先根序列：AB
- 层次序列：AB

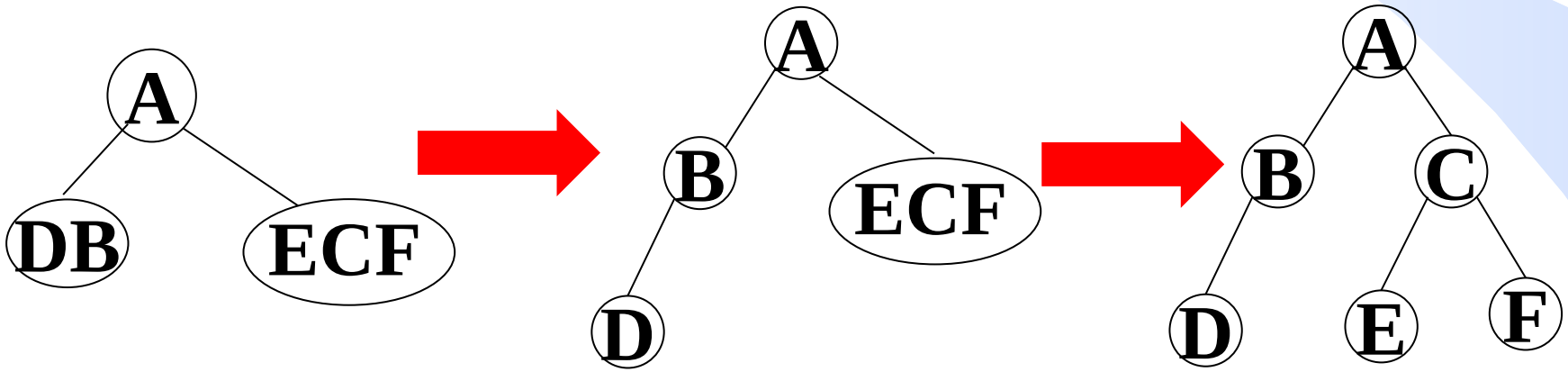
由后根序列和层次遍历序列可否唯一确定一棵二叉树？

- 后根序列：BA
- 层次序列：AB

由中根序列和层次遍历序列可否唯一确定一棵二叉树？【清华大学考研题、吉林大学18级期末考试题】

中根序列　D B A E C F

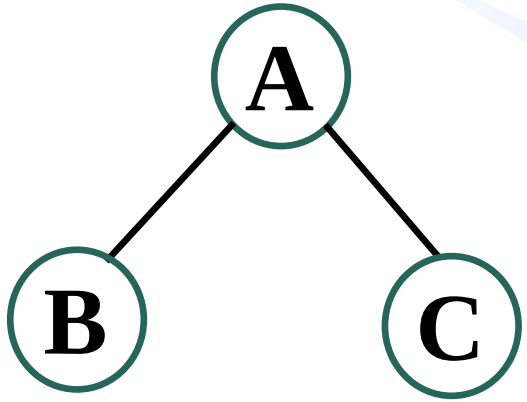
层次序列　A B C D E F



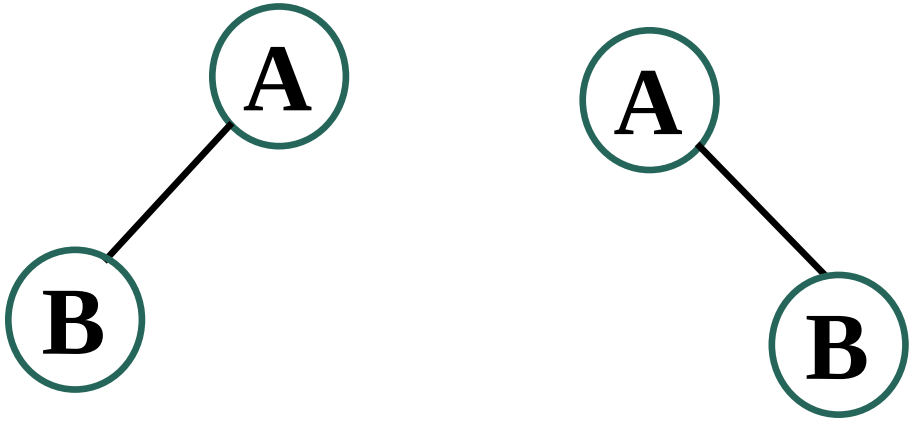
中根序列和任意一种遍历序列都可以唯一地确定一棵二叉树

由先根序列和后根序列可否唯一确定层次遍历序列？【清华大学考研题】

先根序列：A B C
后根序列：B C A

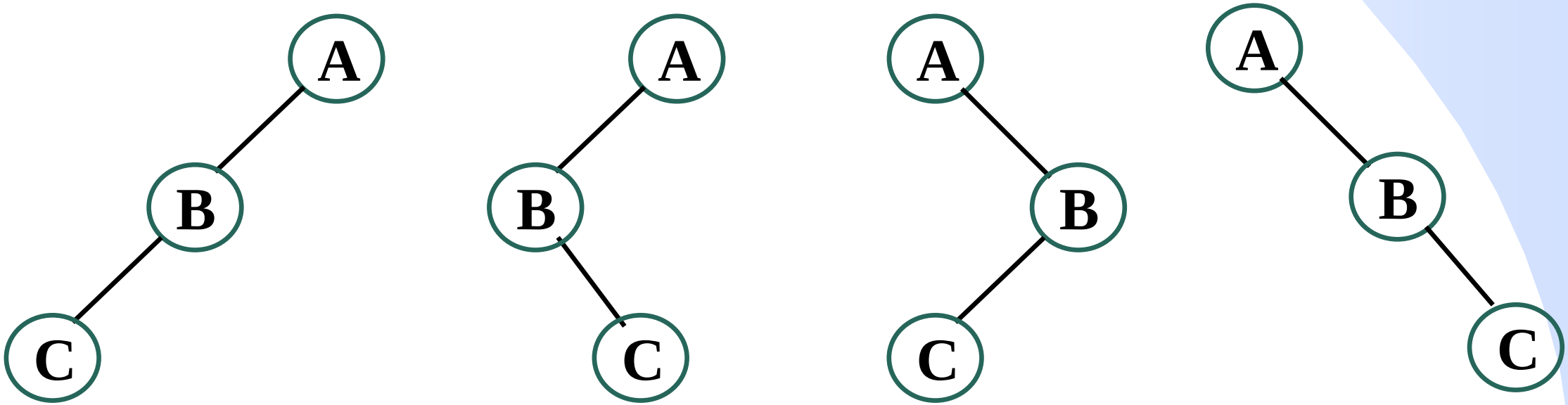


先根序列：A B
后根序列：B A



给定先根序列和后根序列，输出可能有多少种二叉树结构【洛谷P1229】

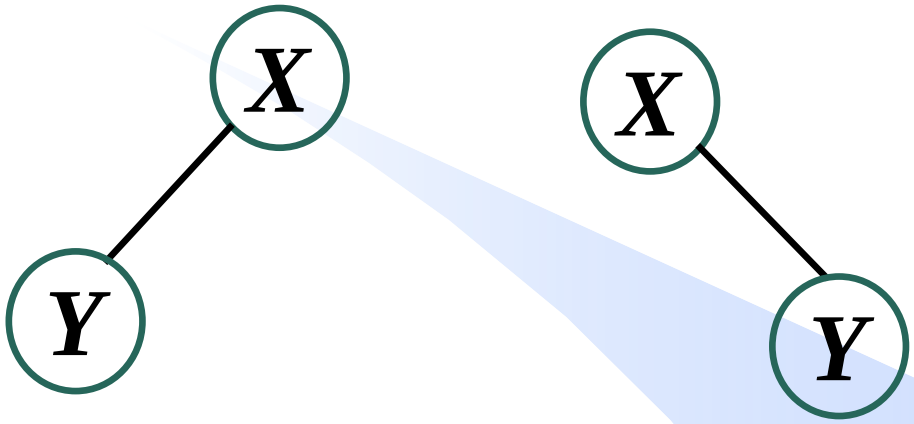
先根序列：A B C
后根序列：C B A



给定先根序列和后根序列，输出可能有多少种二叉树结构【洛谷P1229】

先根序列：…… *X* *Y*……

后根序列：…… *Y* *X* …

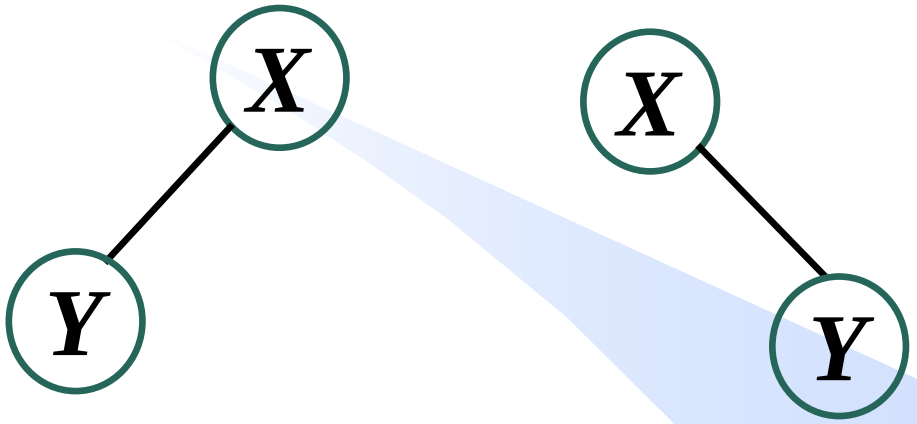


每多一个度为1的结点，二叉树结构翻倍

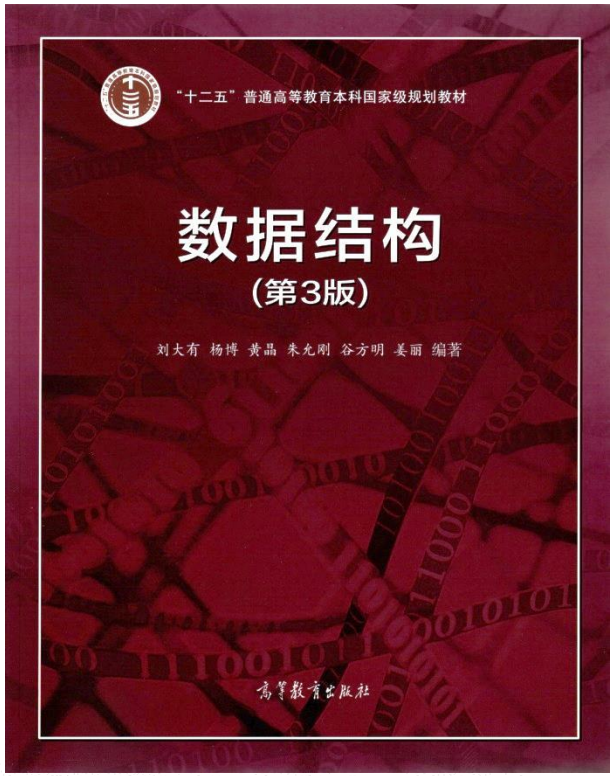
给定先根序列和后根序列，输出可能有多少种二叉树结构【洛谷P1229】

先根序列：……X Y……

后根序列：……Y X …




```
int TreeNum(char preorder[], char postorder[], int n){
    if(n==1) return 1;
    int ans=1;
    for(int i=0;i<n;i++)
        for(int j=1;j<n;j++)
            if(preorder[i]==postorder[j] && preorder[i+1]==postorder[j-1])
                ans*=2;
    return ans;
}
```



二叉树的存储和操作

- 二叉树的存储结构
- 二叉树遍历的递归算法
- 遍历的非递归算法
- 二叉树的重建和计数
- **二叉树其他操作**

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

在二叉树中搜索给定结点的父结点

```
TreeNode* Father(TreeNode *root, TreeNode *p){
    //在以root为根的二叉树中找p的父结点，返回指针
    if(root==NULL||p==root) return NULL; //根空或p为根
    if(root->left==p||root->right==p) //root即p的父结点
        return root;
    TreeNode *fa=Father(root->left,p);//在左子树中找p父
    if(fa!=NULL) return fa;
    return Father(root->right, p); //在root右子树中找p父
}
```

策略
分治法

实质
遍历

解决二叉树问题的一般框架

算法*f* (*root*)

可在此处处理根结点

递归处理左子树*f* (*root*->*left*).

可在此处处理根结点

递归处理右子树*f* (*root*->*right*).

可在此处处理根结点

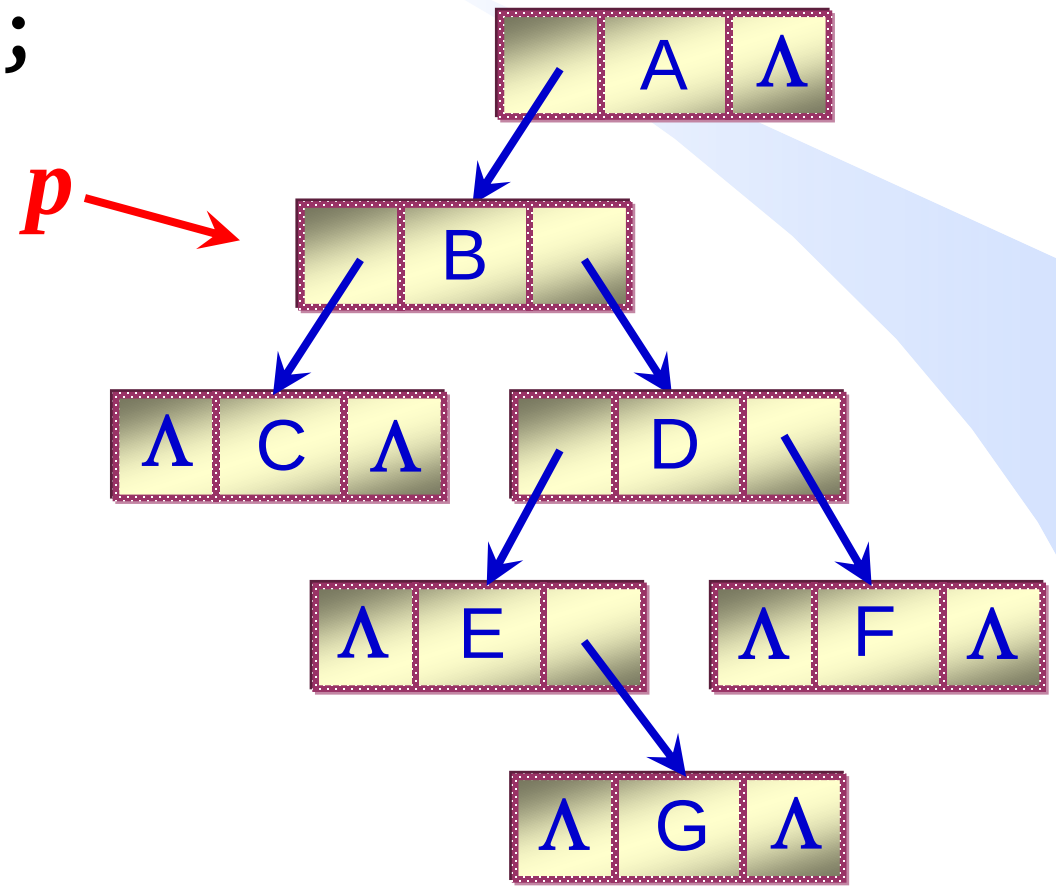
RETURN.

搜索二叉树中符合数据域条件的结点

```
TreeNode* Find(TreeNode *t, int item){  /*在以t为根的二叉树中找数据域为item的结点，返回指向该结点的指针*/
    if (t == NULL) return NULL;
    if (t->data == item) return t;  //t即为所求
    TreeNode* p=Find(t->left, item);//在t左子树中查找
    if(p!=NULL) return p;
    return Find(t->right, item); //在t右子树中查找
}
```

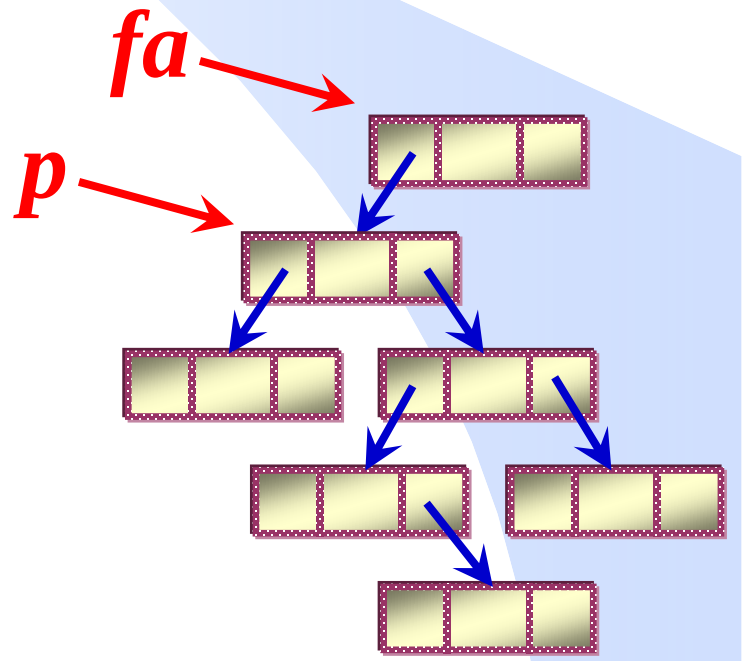
释放二叉树

```
void Del(TreeNode* p){ //释放p指向的子树所占空间
    if(p==NULL) return;
    Del(p->left);
    Del(p->right);
    delete p;
}
```



在以t为根的二叉树中删除p指向的子树

```
void DeleteSubTree(TreeNode *t, TreeNode *p ){
    if(p==NULL) return;
    if(p==t){ Del(t); t=NULL; return;} //p是根
    TreeNode* fa = Father(t, p); //找p的父结点fa
    //修改父结点的指针域
    if(fa->left==p) fa->left=NULL;
    if(fa->right==p) fa->right=NULL;
    Del(p);
}
```

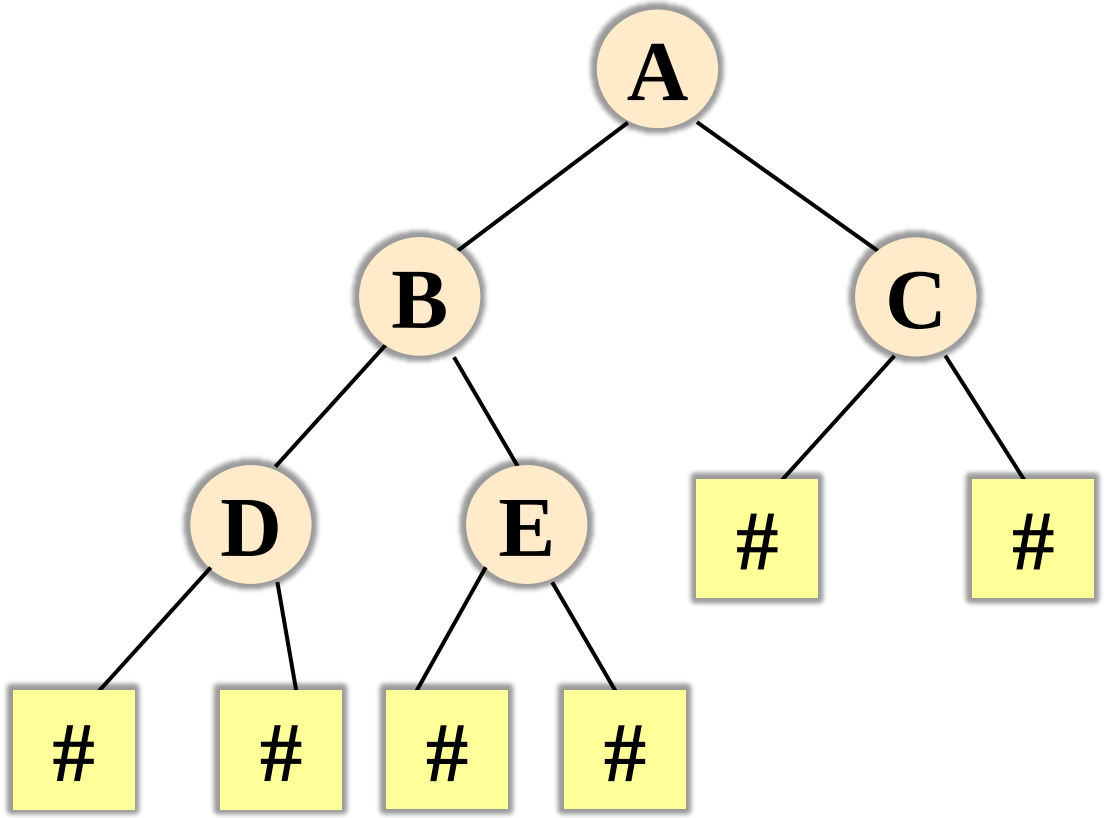


创建二叉树

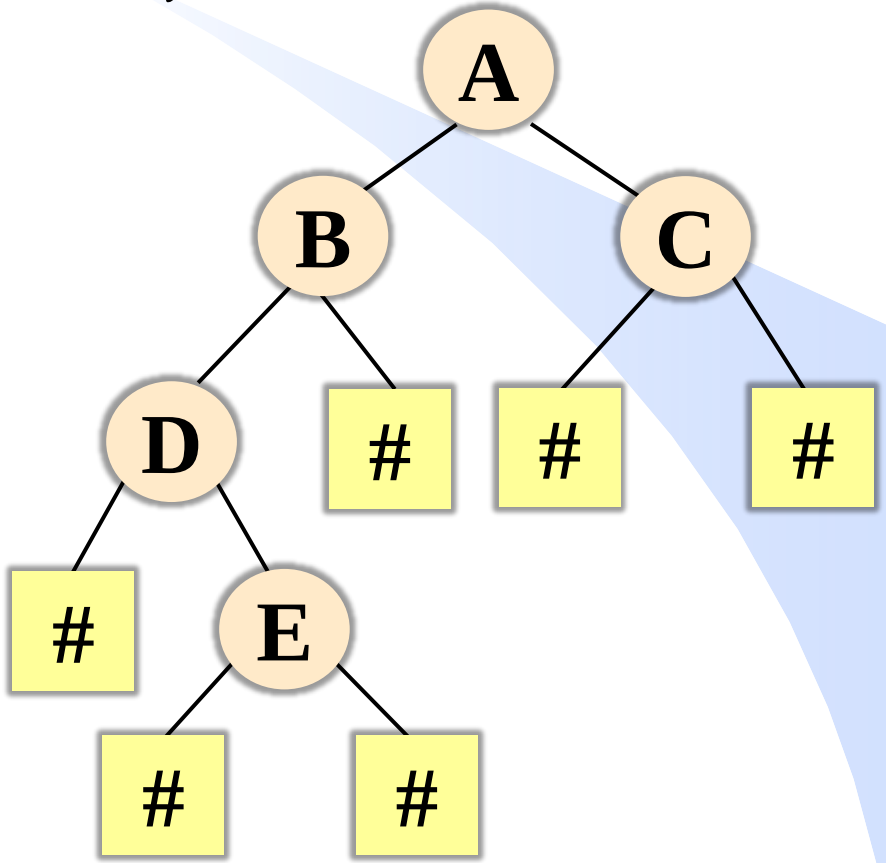
- 通过中根和某种遍历序列：如先根序列+中根序列。
- 通过一种遍历序列：先根序列？
- 先根序列不能唯一确定二叉树。因为在二叉树中，有的结点之左/右指针可能为空，这在先根序列中不能被体现，导致两棵不同的二叉树却可能有相同的先根序列。
- 在先根序列中加入特殊符号以示空指针位置，不妨用#表示空指针位置。

带空指针信息的先根序列——增强先根序列

下面两棵不同的二叉树有相同的先根序列*ABDEC*。在先根序列中加入特殊符号#以表示空指针位置后，先根序列则不同。



A B D # # E # # C # #



A B D # E # # # C # #

根据增强先根序列创建二叉树

根据带空指针信息的先根序列创建二叉树，其中空指针信息用#表示，如*ABD##E##C##*【清华大学考研复试机试】

```
char s[N]; int k=0;
TreeNode* CreateBinTree(char preorder[]){
    char ch=preorder[k++];
    if(ch=='#') return NULL;
    TreeNode *t = new TreeNode;
    t->data=ch;
    t->left=CreateBinTree(preorder);
    t->right=CreateBinTree(preorder);
    return t;
}
```

```
scanf("%s",s);
TreeNode *root=CreateBinTree(s);
```

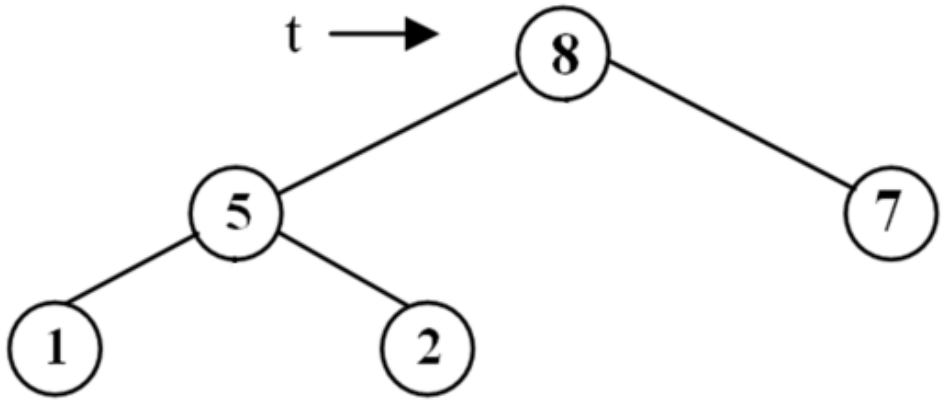
```
struct TreeNode{
    char data;
    TreeNode *left;
    TreeNode *right;
};
```

当读入#字符时，将其初始化为一个空指针，否则生成一个新结点

上机实验常见创建二叉树形式

已知一棵非空二叉树结点的数据域为不等于0的整数，输入为一组用空格间隔的整数，表示带空指针信息的二叉树先根序列，其中空指针信息用0表示，如8 5 1 0 0 2 0 0 7 0 0表示如下二叉树。

```
TreeNode* CreateBinTree(){
    int k;
    scanf("%d", &k);
    if(k==0) return NULL;
    TreeNode *t = new TreeNode;
    t->data = k;
    t->left = CreateBinTree();
    t->right = CreateBinTree();
    return t;
}
```

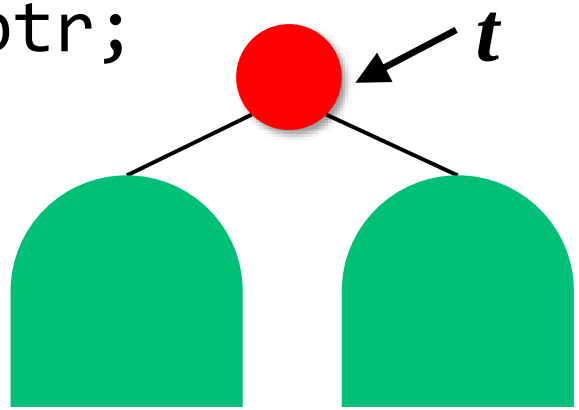


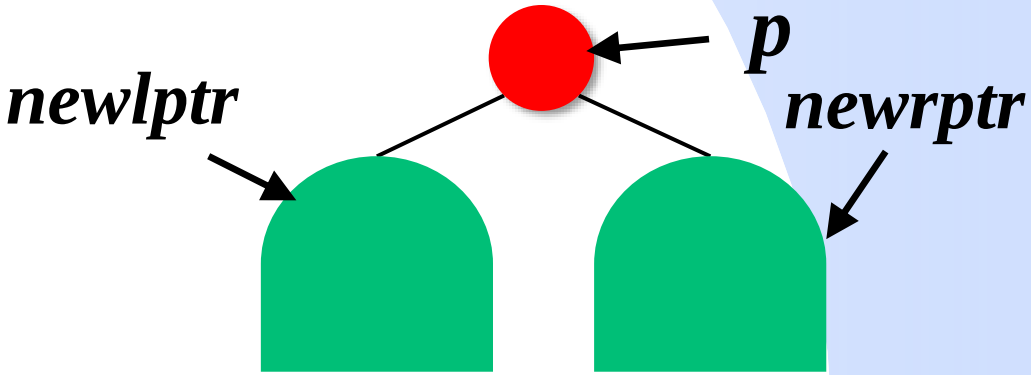

```
struct TreeNode{
    int data;
    TreeNode *left;
    TreeNode *right;
};
```

复制二叉树

先复制子树，再复制根结点，将父结点与子结点连接起来。

```
TreeNode* CopyTree(TreeNode* t){ //复制以t为根的二叉树
    if(t==NULL) return NULL;
    TreeNode* newlptr=CopyTree(t->left); //复制左子树
    TreeNode* newrptr=CopyTree(t->right);//复制右子树
    TreeNode* p=new TreeNode; //生成根结点
    p->data = t-data;
    p->left = newlptr;
    p->right = newrptr;
    return p;
}
```





计算二叉树结点个数

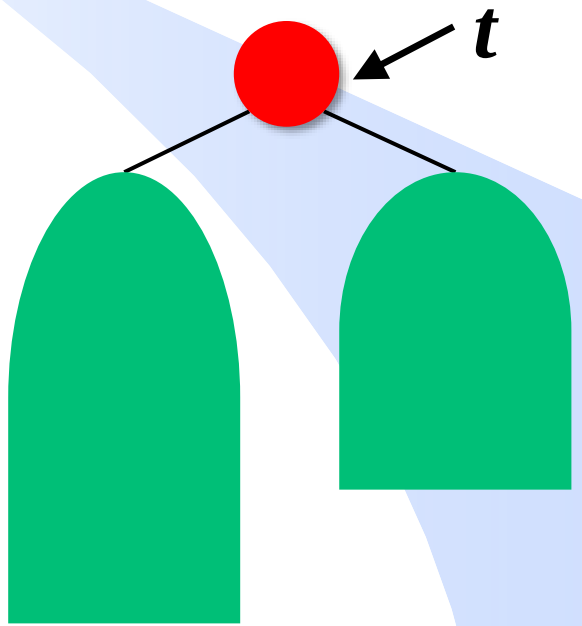
```
int Count(TreeNode* t){
    if(t==NULL) return 0;
    return Count(t->left)+Count(t->right)+1;
}
```

二叉树中结点总数 = 左子树结点总数+右子树结点总数+1（根结点）

计算二叉树高度

$$depth(t)=\begin{cases}-1 & 若t=NULL\\ \max\{depth(t->left),depth(t->right)\}+1 & 若t\neq NULL\end{cases}$$

```
int depth(TreeNode* t){
    if (t==NULL) return -1;
    int d1 = depth(t->left);
    int d2 = depth(t->right);
    return(d1>d2)? d1+1:d2+1;
}
```



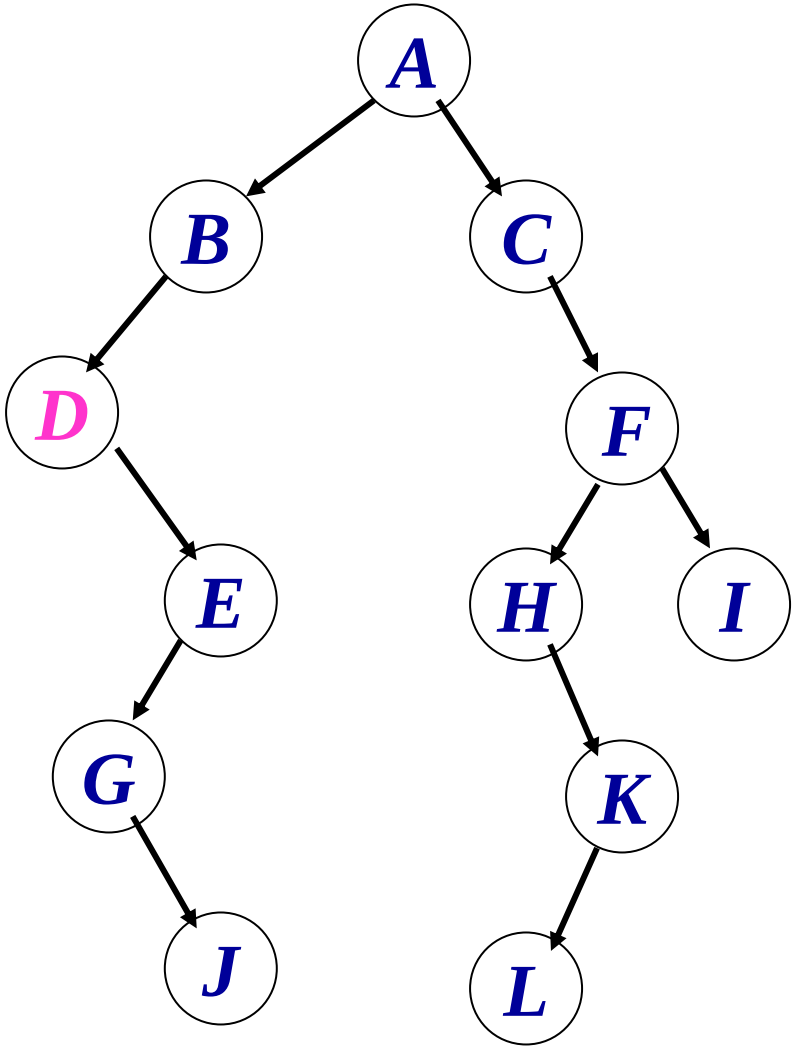
二叉树中、先、后根序列的首末结点

- 编写算法，找出二叉树中根序列的第一个结点，要求不使用递归、不使用栈。
- 编写算法，找出二叉树先根序列的最后一个结点，要求不使用递归、不使用栈。【上海交通大学、吉林大学考研题】
- 编写算法，找出二叉树后根序列的第一个结点，要求不使用递归、不使用栈。【哈尔滨工业大学期末考试题】

二叉树中、先、后根序列的首末结点

二叉树的根结点指针 t	中根序列	先根序列	后根序列
第一个结点			
最后一个结点			

二叉树中根序列的第一个结点




```
if(t==NULL)return NULL;
TreeNode* p=t;
while(p->left!=NULL)
    p=p->left;
return p;
```

时间复杂度O(*h*)
***h*为二叉树高度**

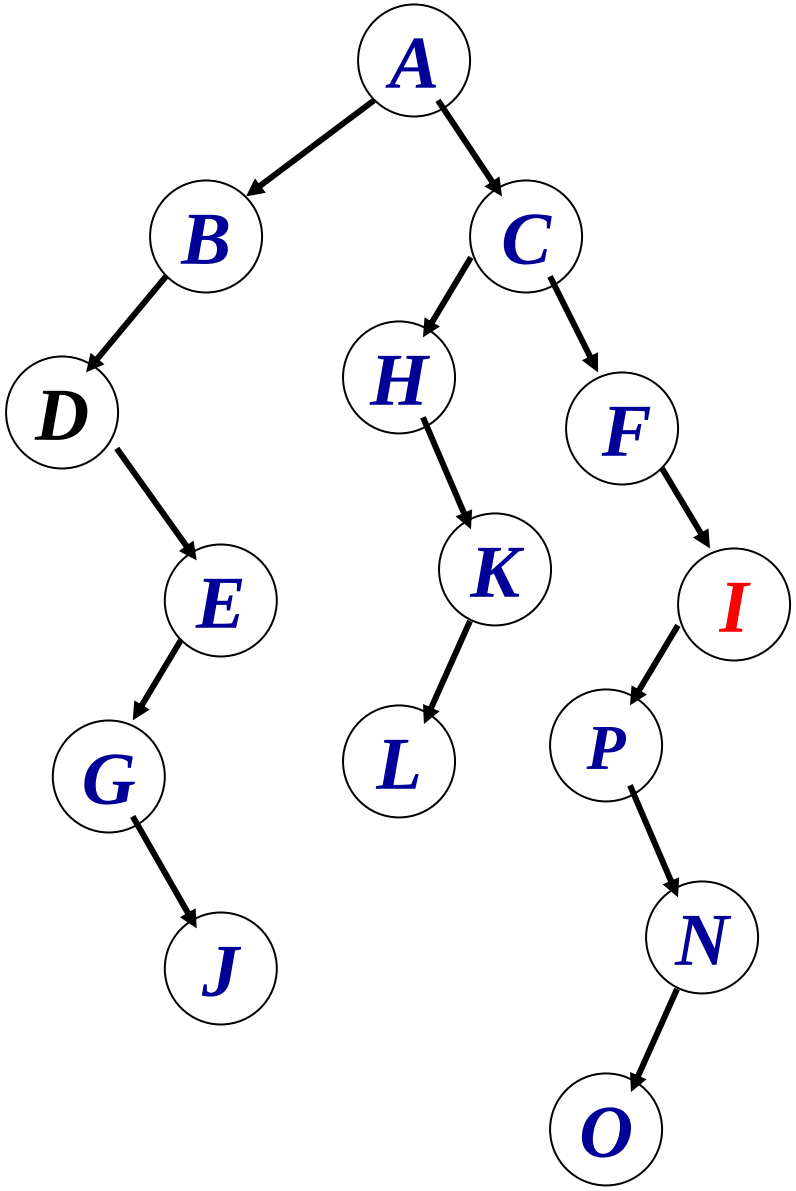
二叉树中、先、后根序列的首末结点

二叉树根指针 t	中根序列	先根序列	后根序列

第一个结点	`if(t==NULL)return NULL;`
	`TreeNode* p=t;`
	`while(p->left!=NULL)`
	`  p=p->left;`
	`return p;`		

最后一个结点			

二叉树中根序列的最后一个结点




```
if(t==NULL)return NULL;
TreeNode* p=t;
while(p->right!=NULL)
    p=p->right;
return p;
```

时间复杂度O(*h*)
***h*为二叉树高度**

二叉树中、先、后根序列的首末结点

二叉树根指针 t	中根序列	先根序列	后根序列

第一个结点	`if(t==NULL)return NULL;`
	`TreeNode* p=t;`
	`while(p->left!=NULL)`
	`  p=p->left;`
	`return p;`		

最后一个结点	`if(t==NULL)return NULL;`
	`TreeNode* p=t;`
	`while(p->right!=NULL)`
	`  p=p->right;`
	`return p;`		

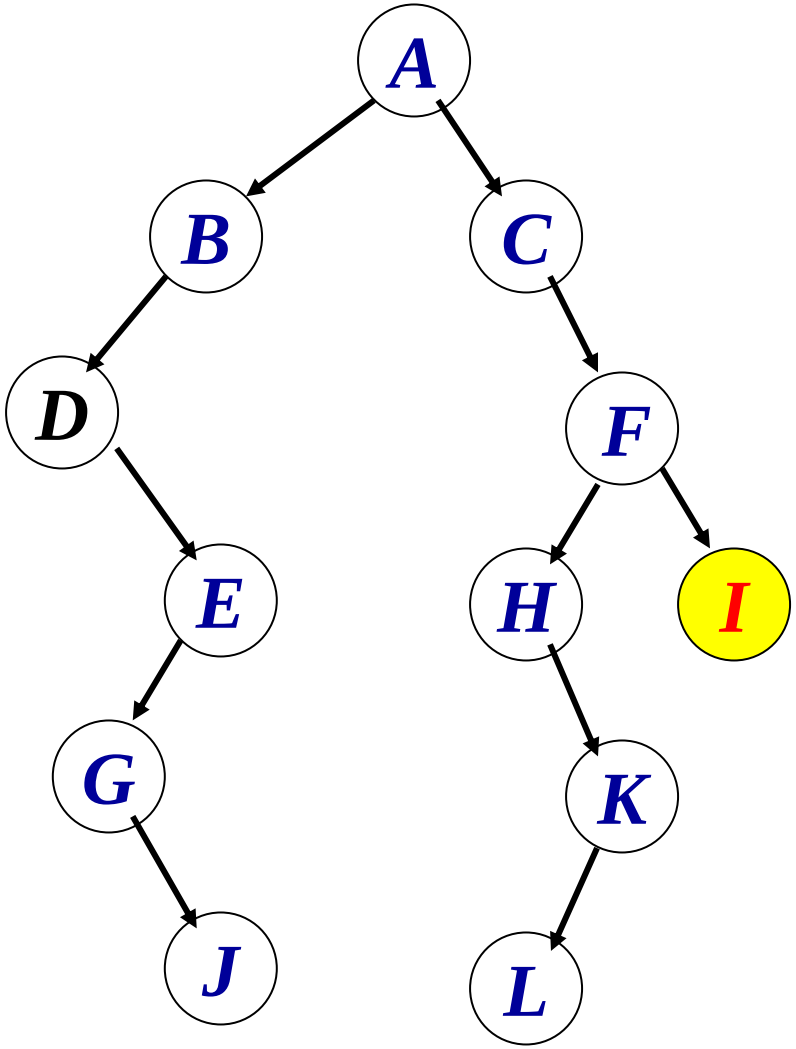
二叉树中、先、后根序列的首末结点

二叉树根指针 t	中根序列	先根序列	后根序列

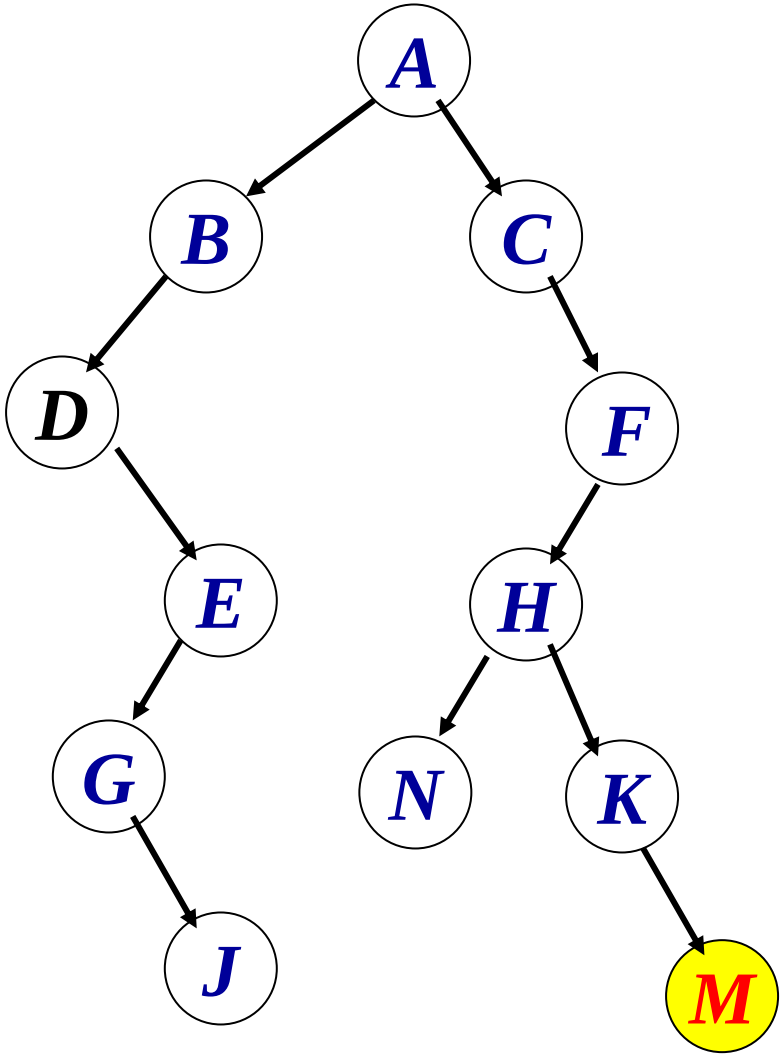
第一个结点	`if(t==NULL)return NULL;`
	`TreeNode* p=t;`
	`while(p->left!=NULL)`
	`  p=p->left;`
	`return p;`
		`return t;`	

最后一个结点	`if(t==NULL)return NULL;`
	`TreeNode* p=t;`
	`while(p->right!=NULL)`
	`  p=p->right;`
	`return p;`		

二叉树先根序列的最后一个结点

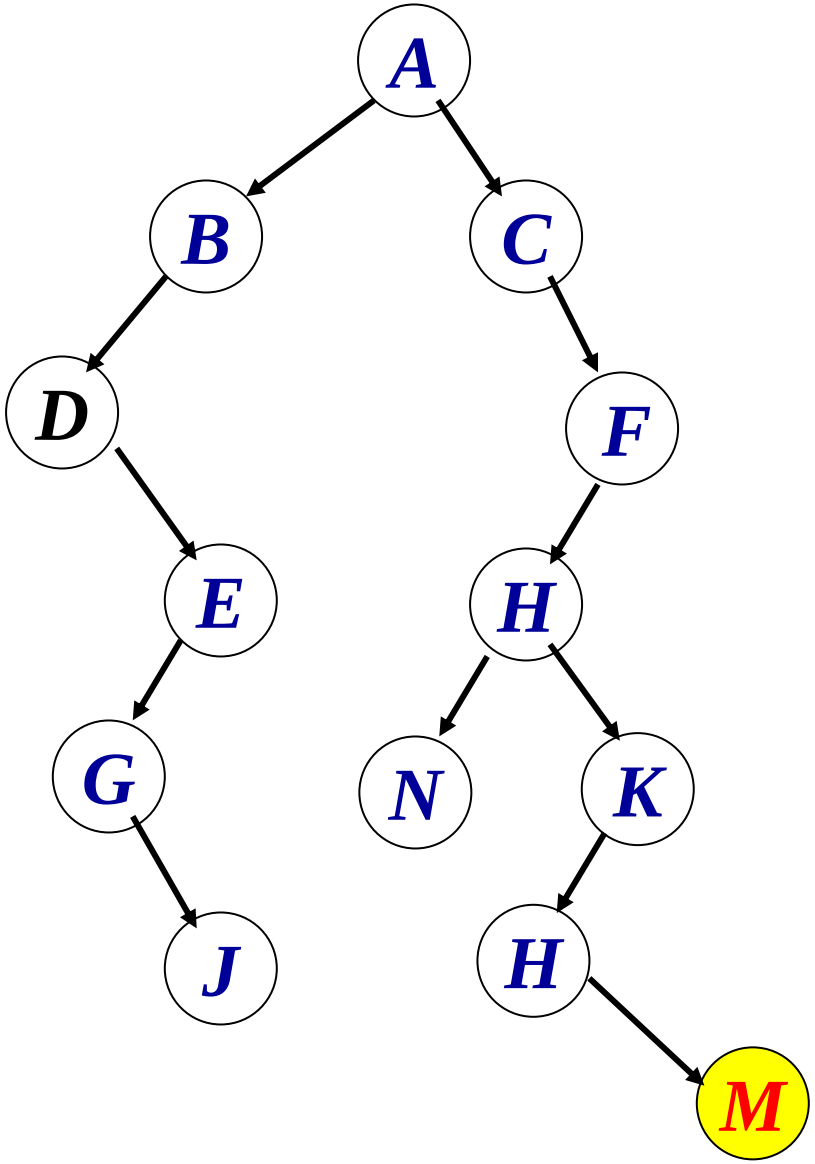


二叉树先根序列的最后一个结点



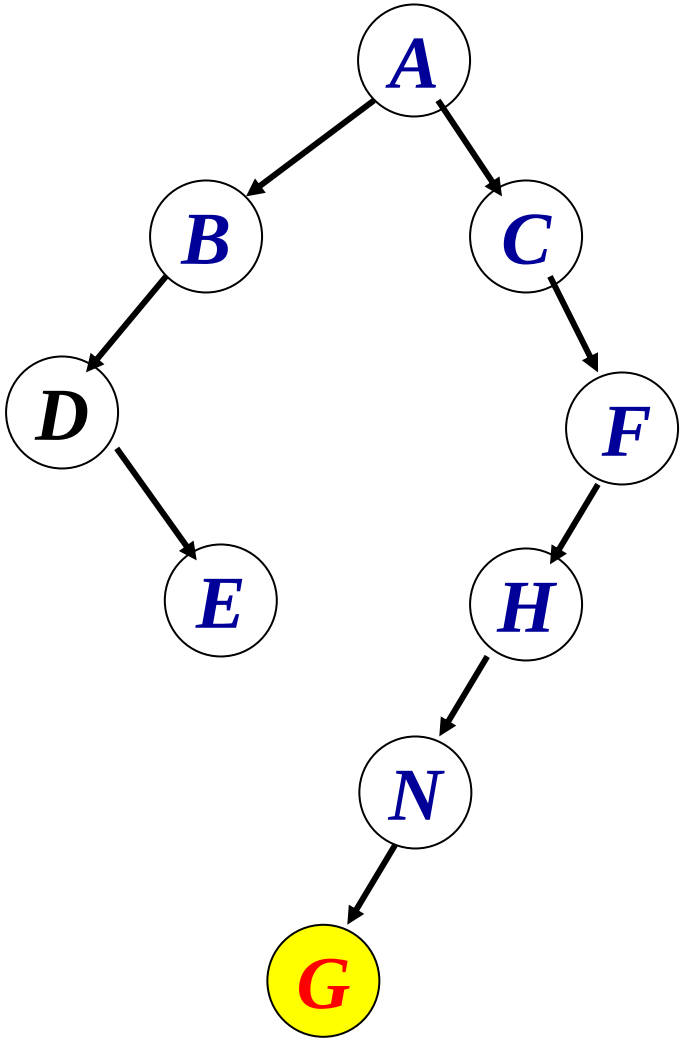
从根结点开始，沿右分支找第一个叶结点，若找不到则在最右边的结点的左子树沿右分支找叶结点，以此类推。

二叉树先根序列的最后一个结点



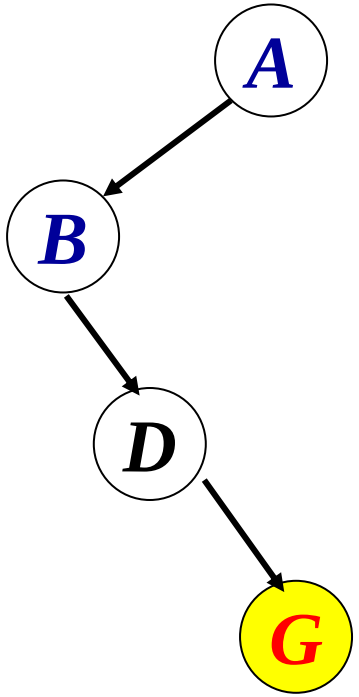
从根结点开始，沿右分支找第一个叶结点，若找不到则在最右边的结点的左子树沿右分支找叶结点，以此类推。

二叉树先根序列的最后一个结点



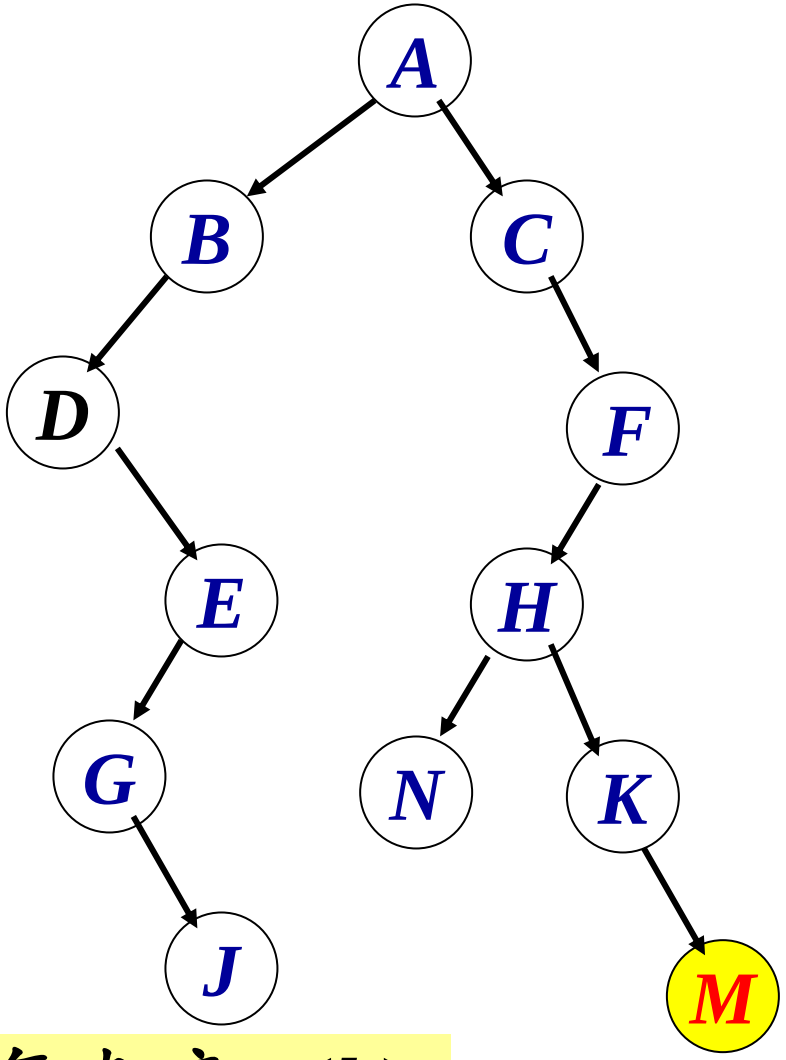
从根结点开始，沿右分支找第一个叶结点，若找不到则在最右边的结点的左子树沿右分支找叶结点，以此类推。

二叉树先根序列的最后一个结点



从根结点开始，沿右分支找第一个叶结点，若找不到则在最右边的结点的左子树沿右分支找叶结点，以此类推。

二叉树先根序列的最后一个结点



从根结点开始，沿右分支找第一个叶结点，若找不到则在最右边的结点的左子树沿右分支找叶结点，以此类推。

```
if(t==NULL) return NULL;
TreeNode* p=t;
while(p!=NULL){
  if(p->right!=NULL)
      p=p->right;
  else if(p->left!=NULL)
      p=p->left;
  else return p;
}
```

时间复杂度O(*h*)
***h*为二叉树高度**

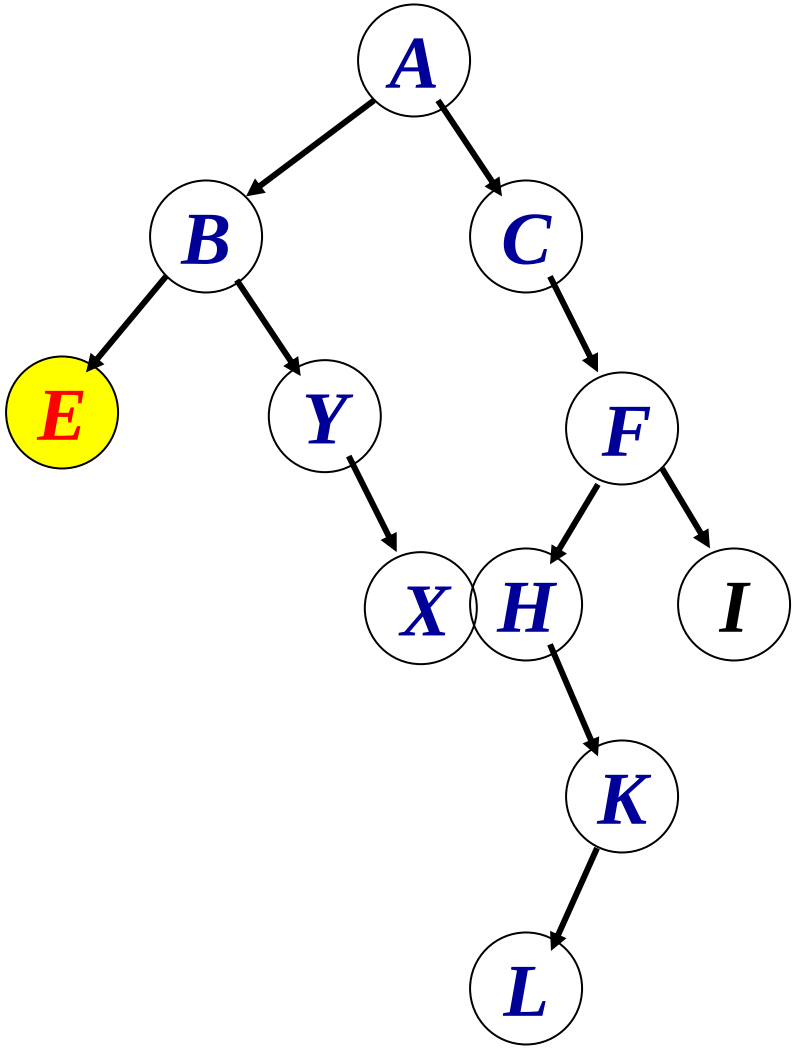
二叉树中、先、后根序列的首末结点

二叉树根指针 t	中根序列	先根序列	后根序列

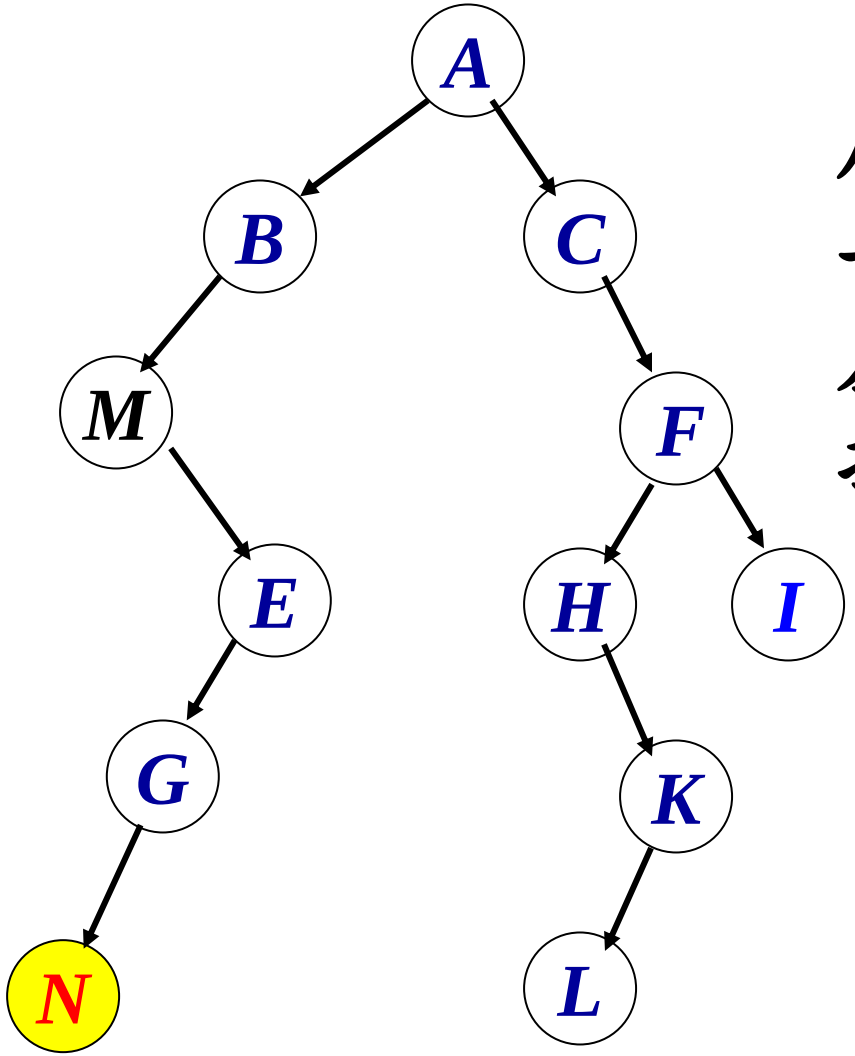
第一个结点	`if(t==NULL)return NULL;`
	`TreeNode* p=t;`
	`while(p->left!=NULL)`
	`  p=p->left;`
	`return p;`
		`return t;`	

最后一个结点	`if(t==NULL)return NULL;`
	`TreeNode* p=t;`
	`while(p->right!=NULL)`
	`  p=p->right;`
	`return p;`
		`if(t==NULL) return NULL;`
		`TreeNode* p=t;`
		`while(p!=NULL){`
		`  if(p->right!=NULL)`
		`  p=p->right;`
		`  else if(p->left!=NULL)`
		`  p=p->left;`
		`  else return p;`
		`}`	

二叉树后根序列的第一个结点

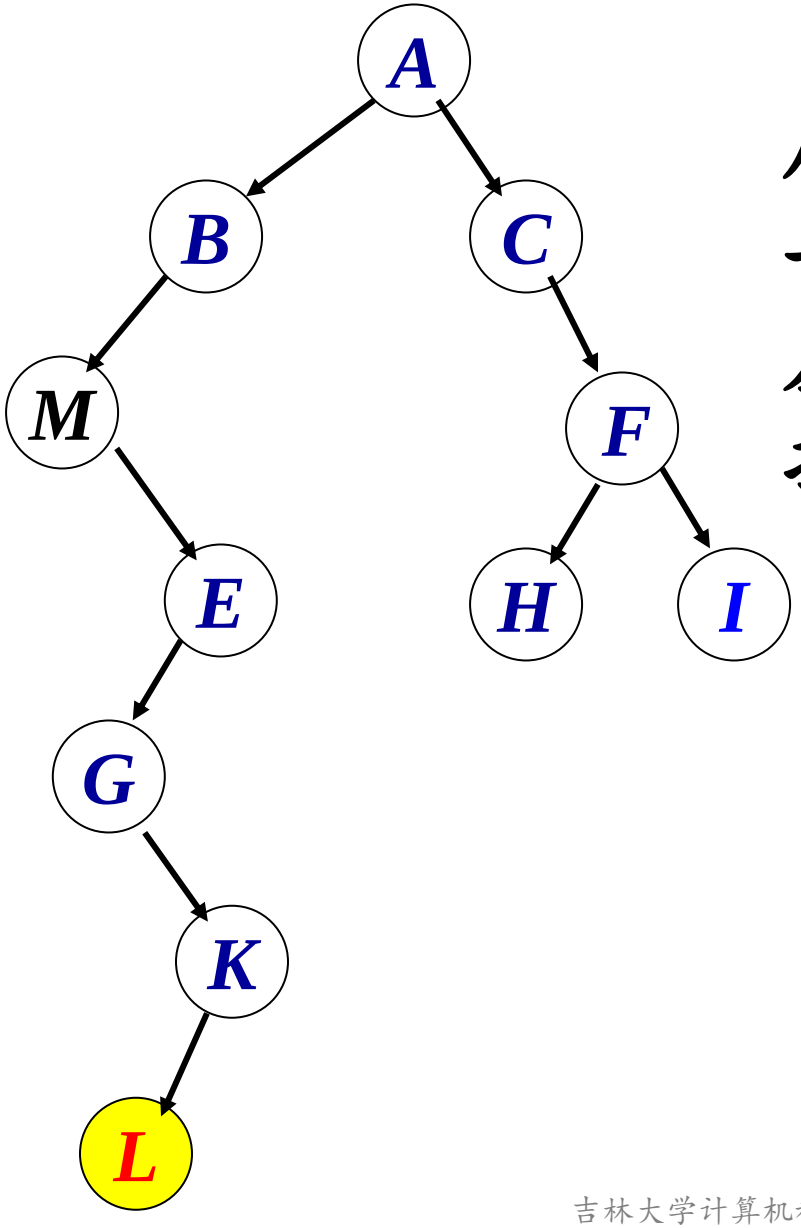


二叉树后根序列的第一个结点



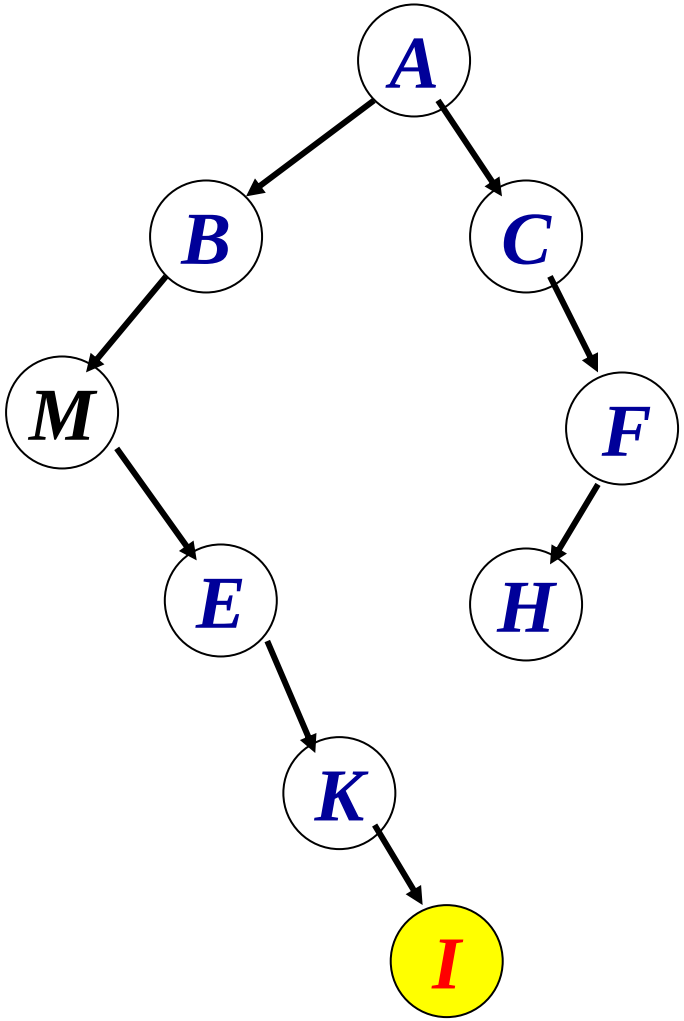
从根结点开始，沿左分支找第一个叶结点，若找不到则在最左边的结点的右子树沿左分支找叶结点，以此类推。

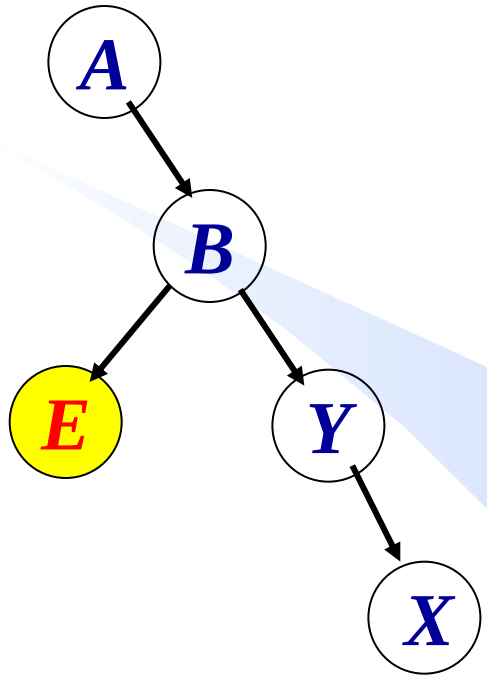
二叉树后根序列的第一个结点



从根结点开始，沿左分支找第一个叶结点，若找不到则在最左边的结点的右子树沿左分支找叶结点，以此类推。

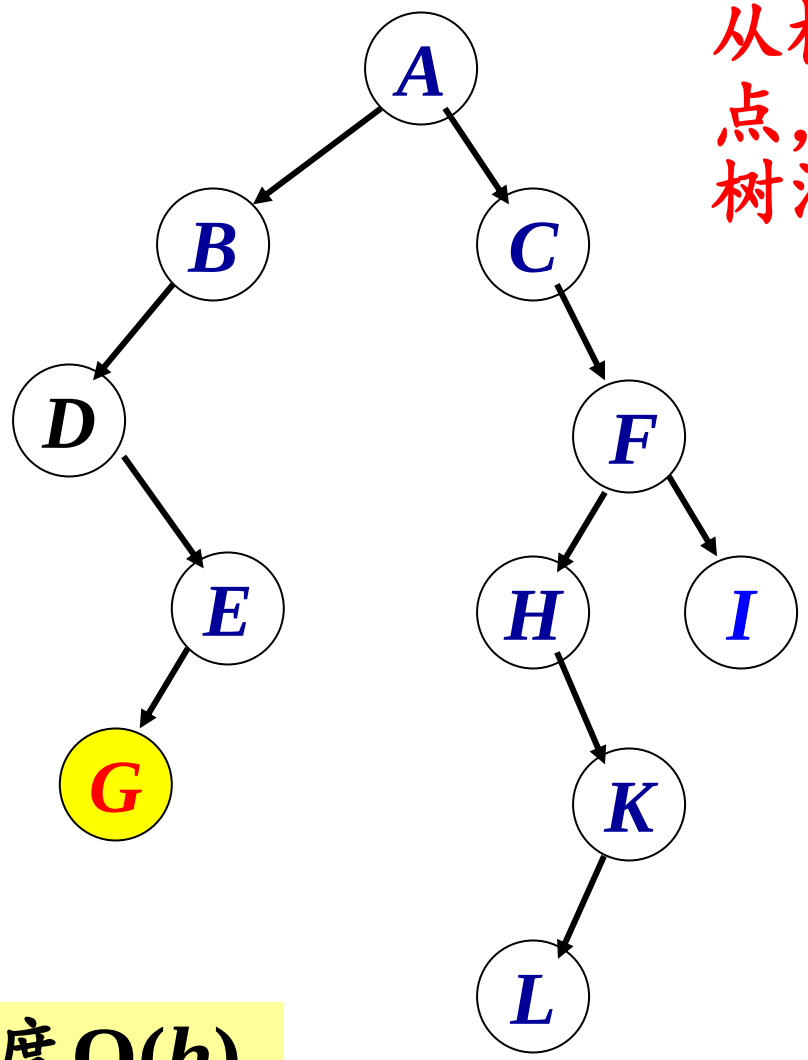
二叉树后根序列的第一个结点





从根结点开始，沿左分支找第一个叶结点，若找不到则在最左边的结点的右子树沿左分支找叶结点，以此类推。

二叉树后根序列的第一个结点



从根结点开始，沿左分支找第一个叶结点，若找不到则在最左边的结点的右子树沿左分支找叶结点，以此类推。

```
if(t==NULL) return NULL;
TreeNode* p=t;
while(p!=NULL){
   if(p->left!=NULL)
       p=p->left;
   else if(p->right!=NULL)
       p=p->right;
   else return p;
}
```

时间复杂度O(*h*)
***h*为二叉树高度**

二叉树中、先、后根序列的首末结点

根指针 t	中根序列	先根序列	后根序列

第一个结点	`if(t==NULL)return NULL;`
	`TreeNode* p=t;`
	`while(p->left!=NULL)`
	`  p=p->left;`
	`return p;`
		`return t;`	`if(t==NULL) return NULL;`
			`TreeNode* p=t;`
			`while(p!=NULL){`
			`  if(p->left!=NULL)`
			`  p=p->left;`
			`  else if(p->right!=NULL)`
			`  p=p->right;`
			`  else return p;`
			`}`

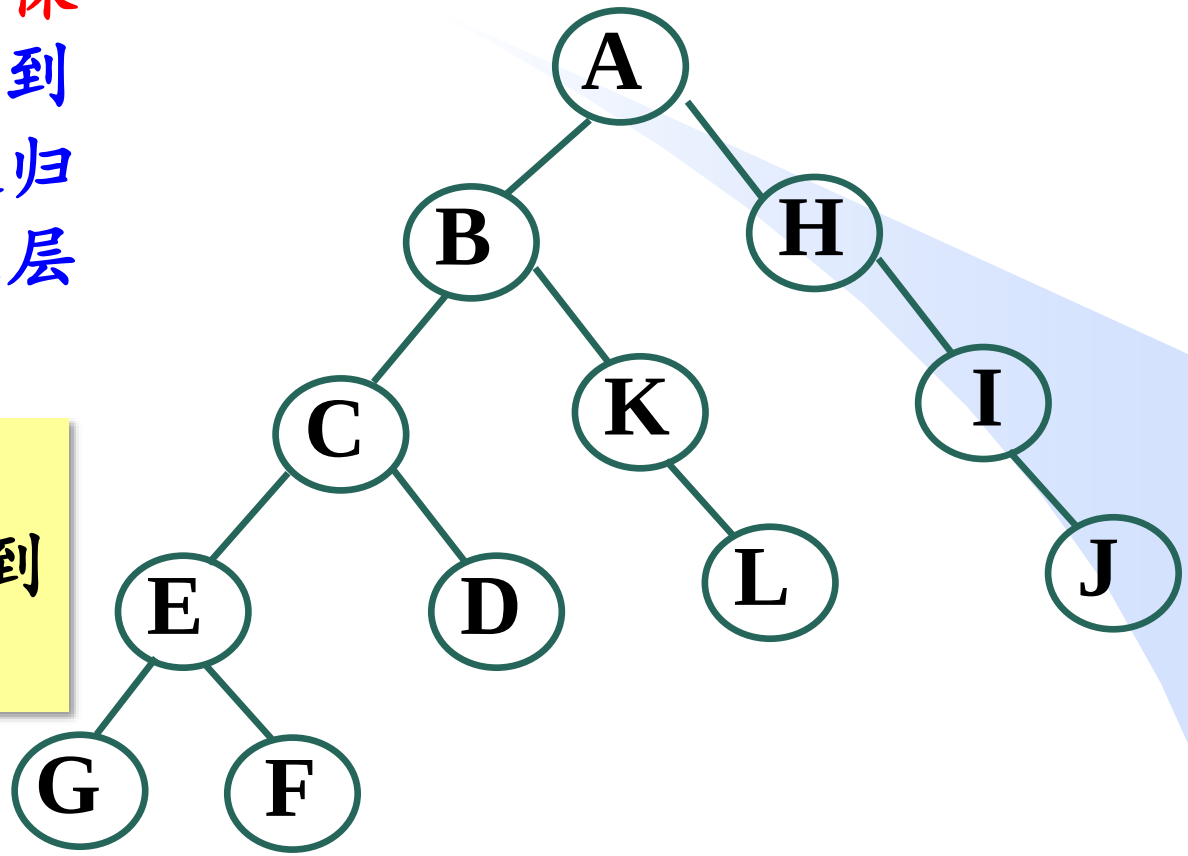
最后一个结点	`if(t==NULL)return NULL;`
	`TreeNode* p=t;`
	`while(p->right!=NULL)`
	`  p=p->right;`
	`return p;`
		`if(t==NULL) return NULL;`
		`TreeNode* p=t;`
		`while(p!=NULL){`
		`  if(p->right!=NULL)`
		`  p=p->right;`
		`  else if(p->left!=NULL)`
		`  p=p->left;`
		`  else return p;`
		`}`
			`return t;`

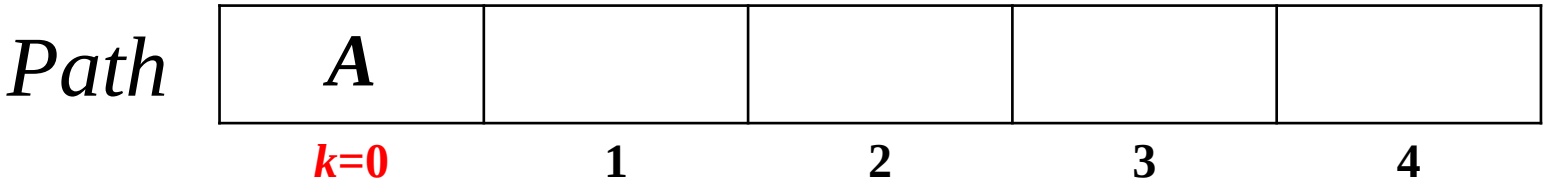
二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



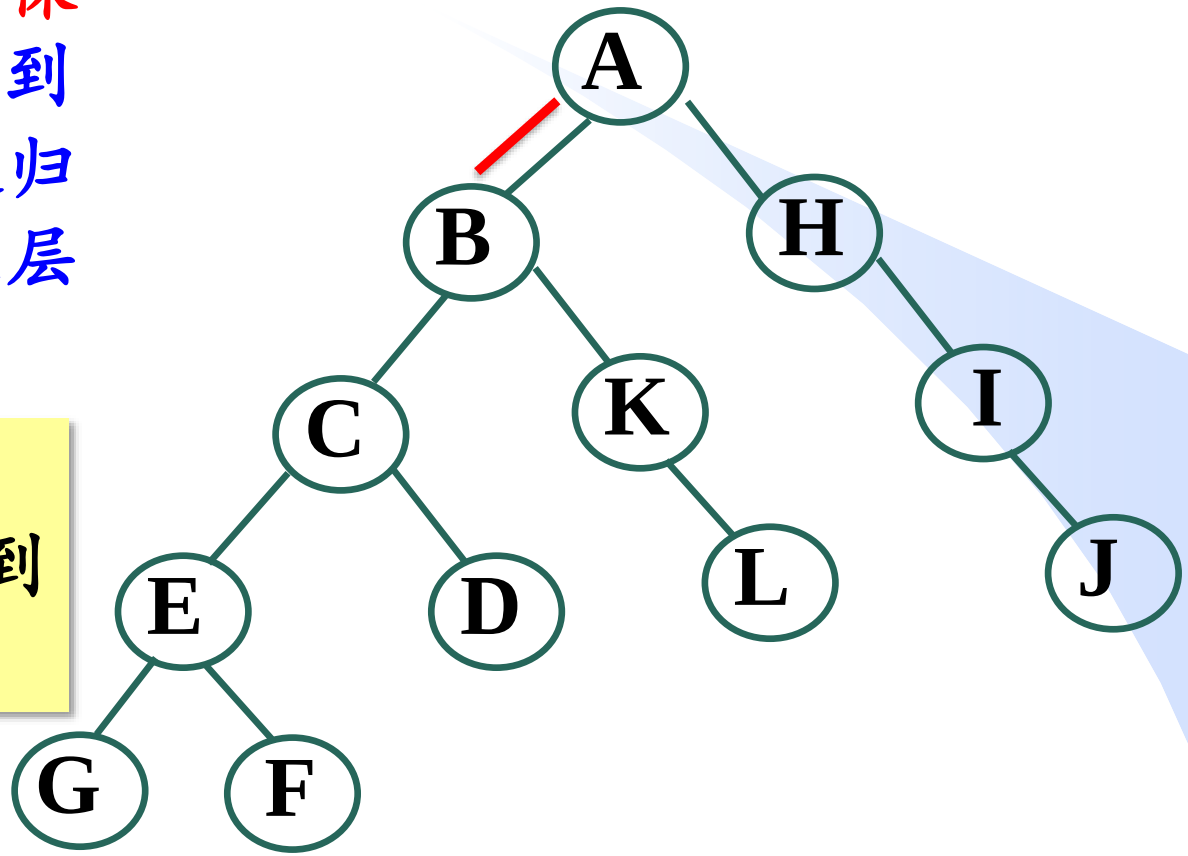


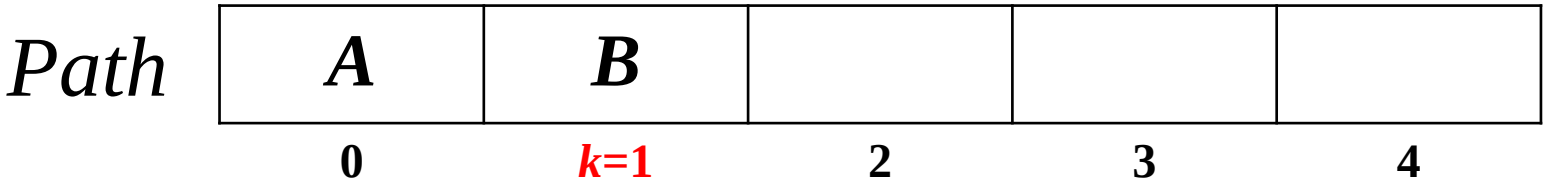
二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



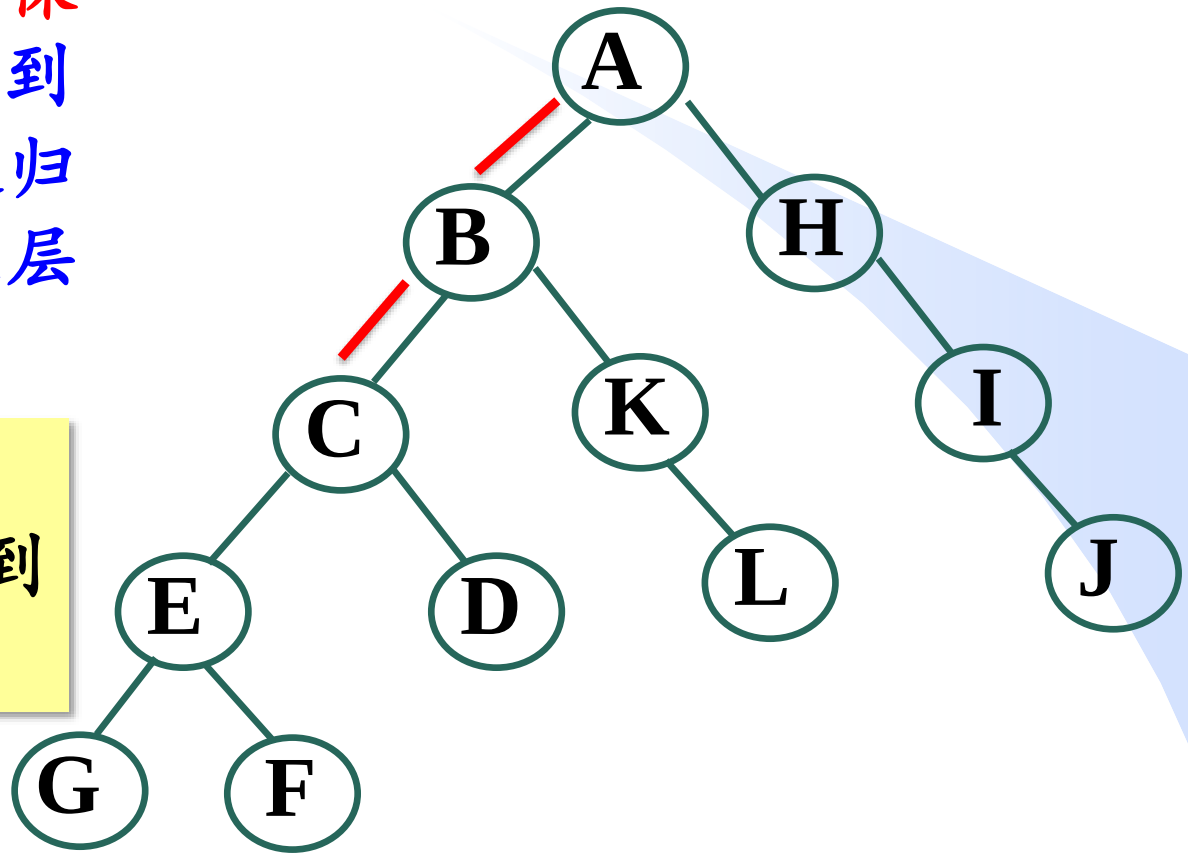


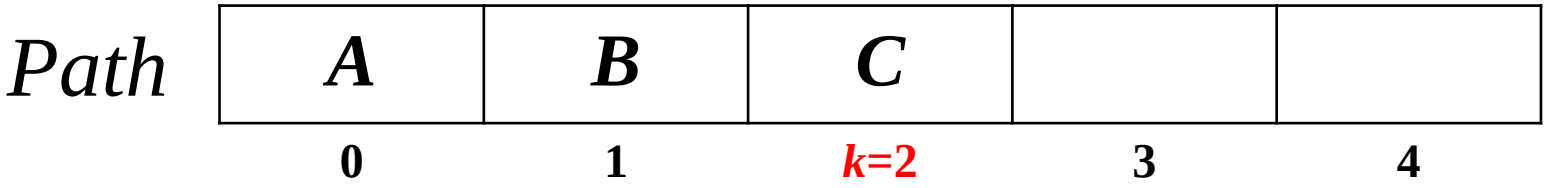
二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



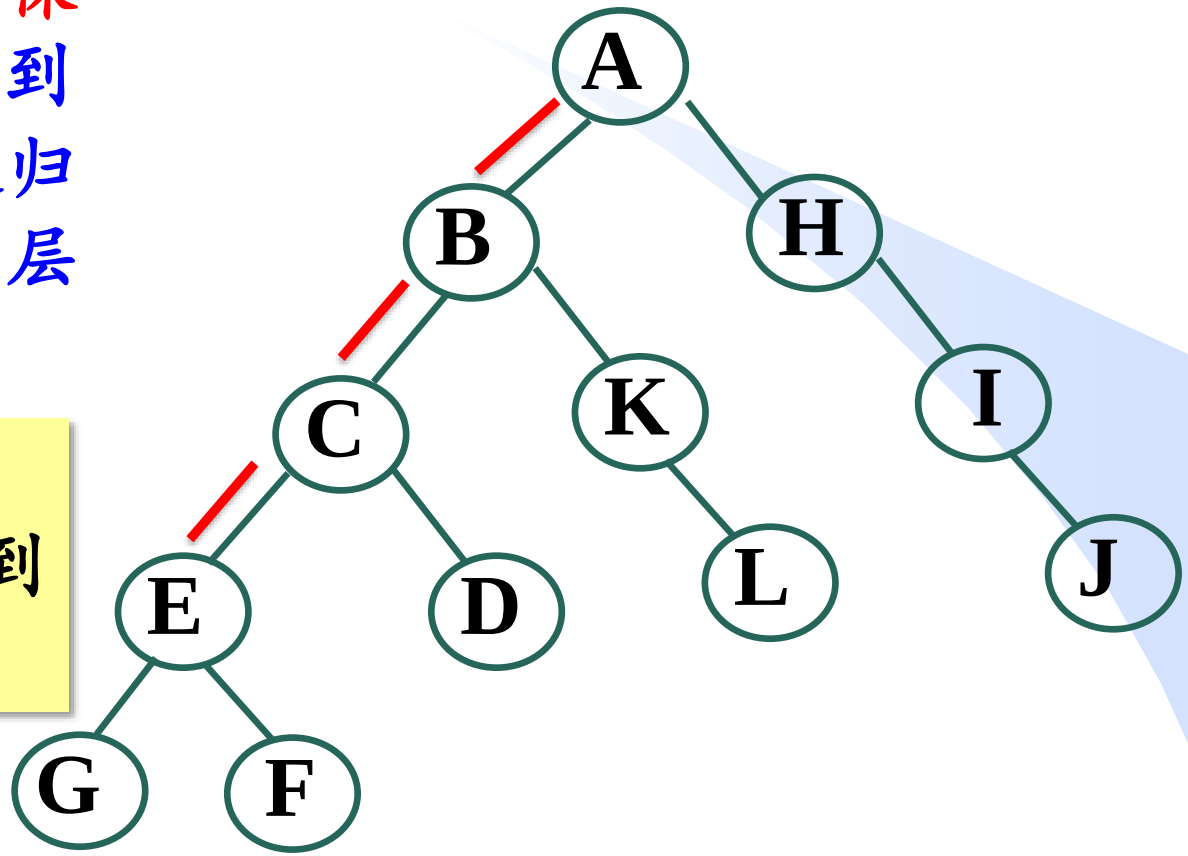


二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



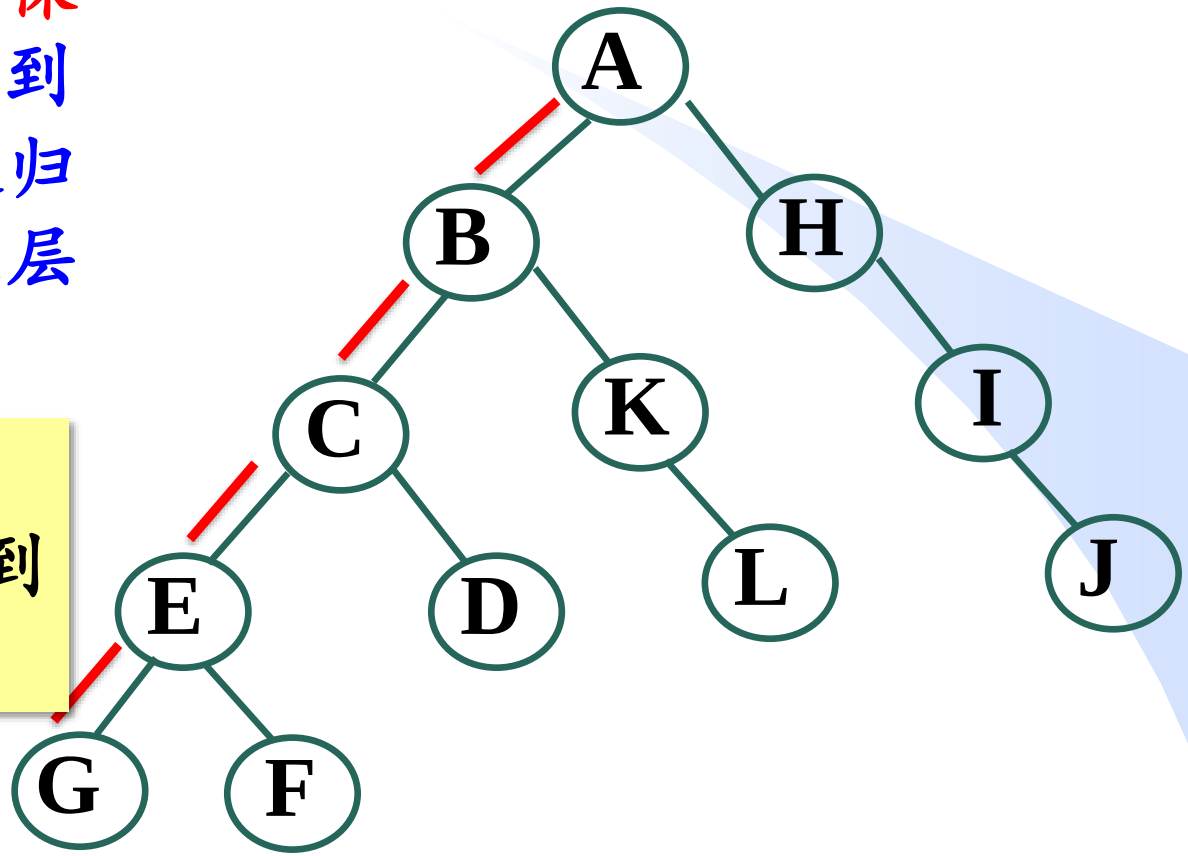
Path	*A*	*B*	*C*	*E*	
	0	1	2	*k*=3	4

二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



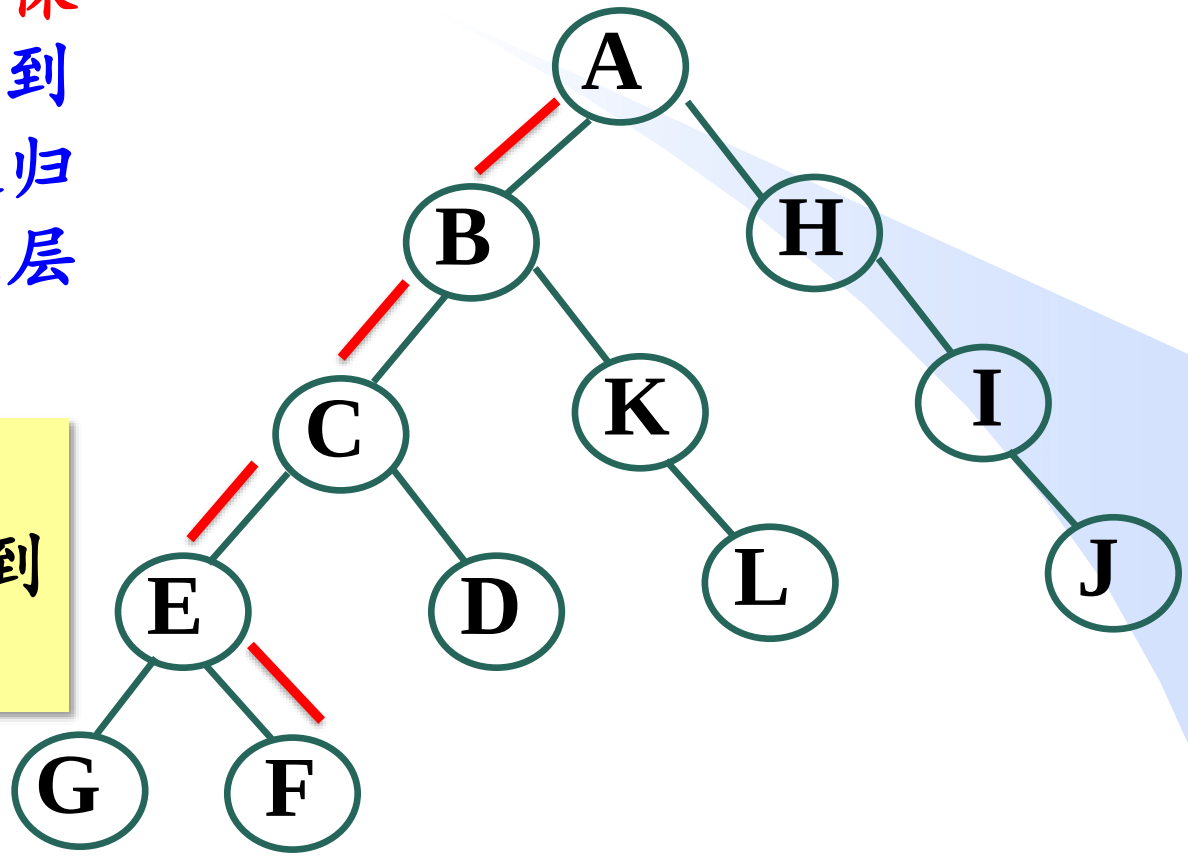
Path	*A*	*B*	*C*	*E*	*G*
	0	1	2	3	*k*=4

二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



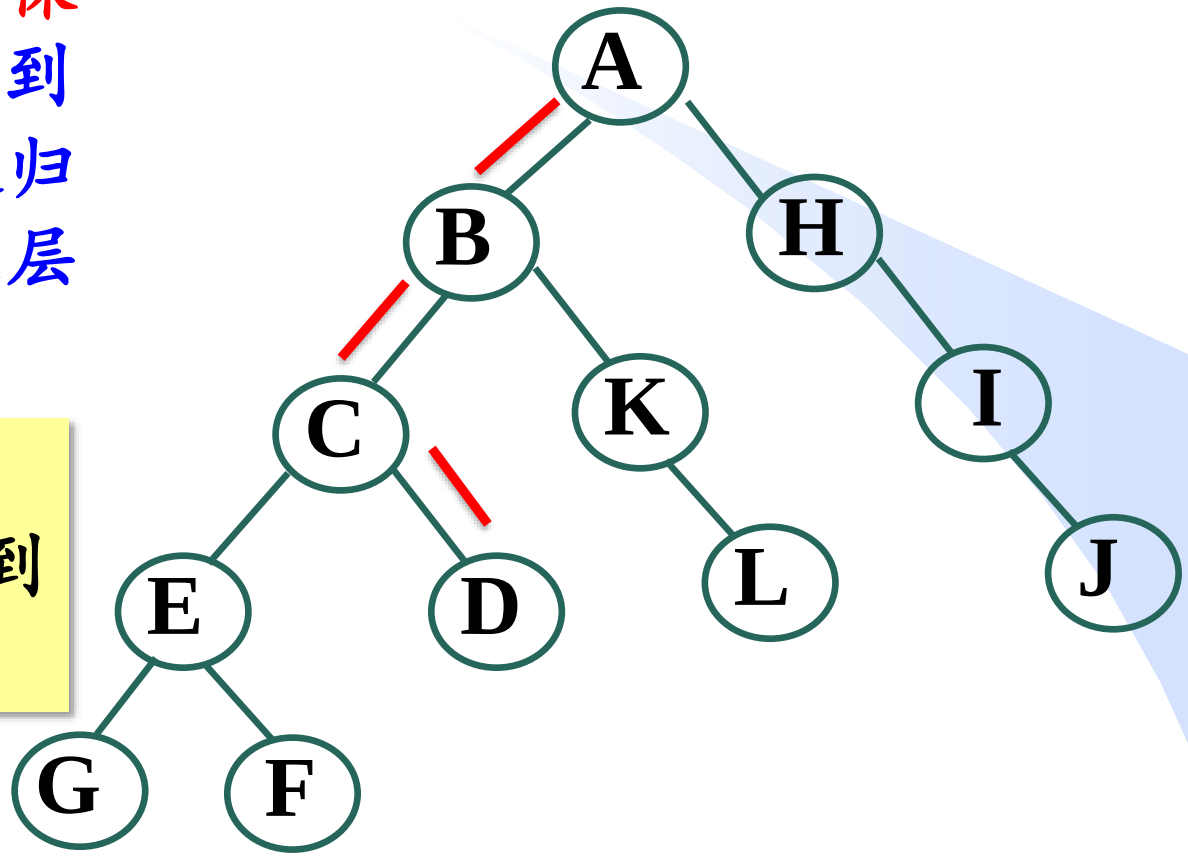
Path	*A*	*B*	*C*	*E*	*F*
	0	1	2	3	*k*=4

二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



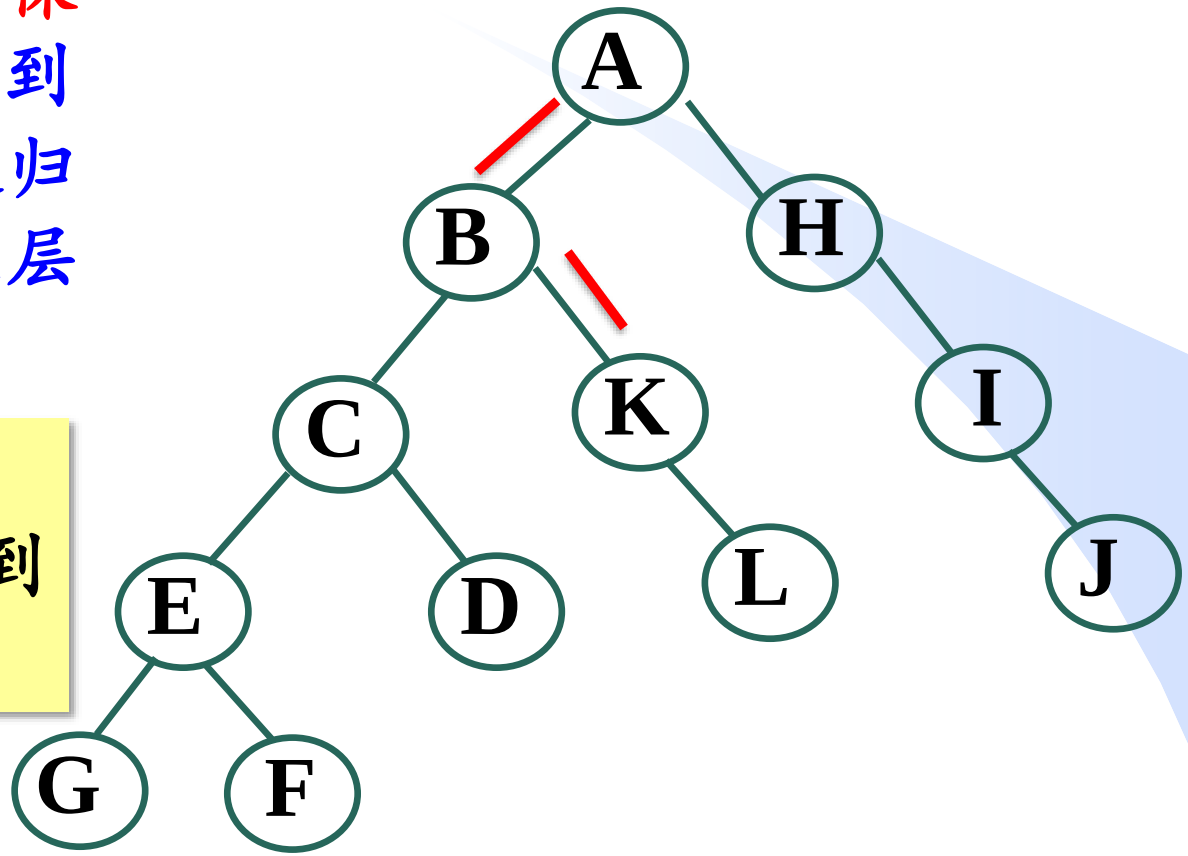
Path	*A*	*B*	*C*	*D*	*F*
	0	1	2	*k*=3	4

二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



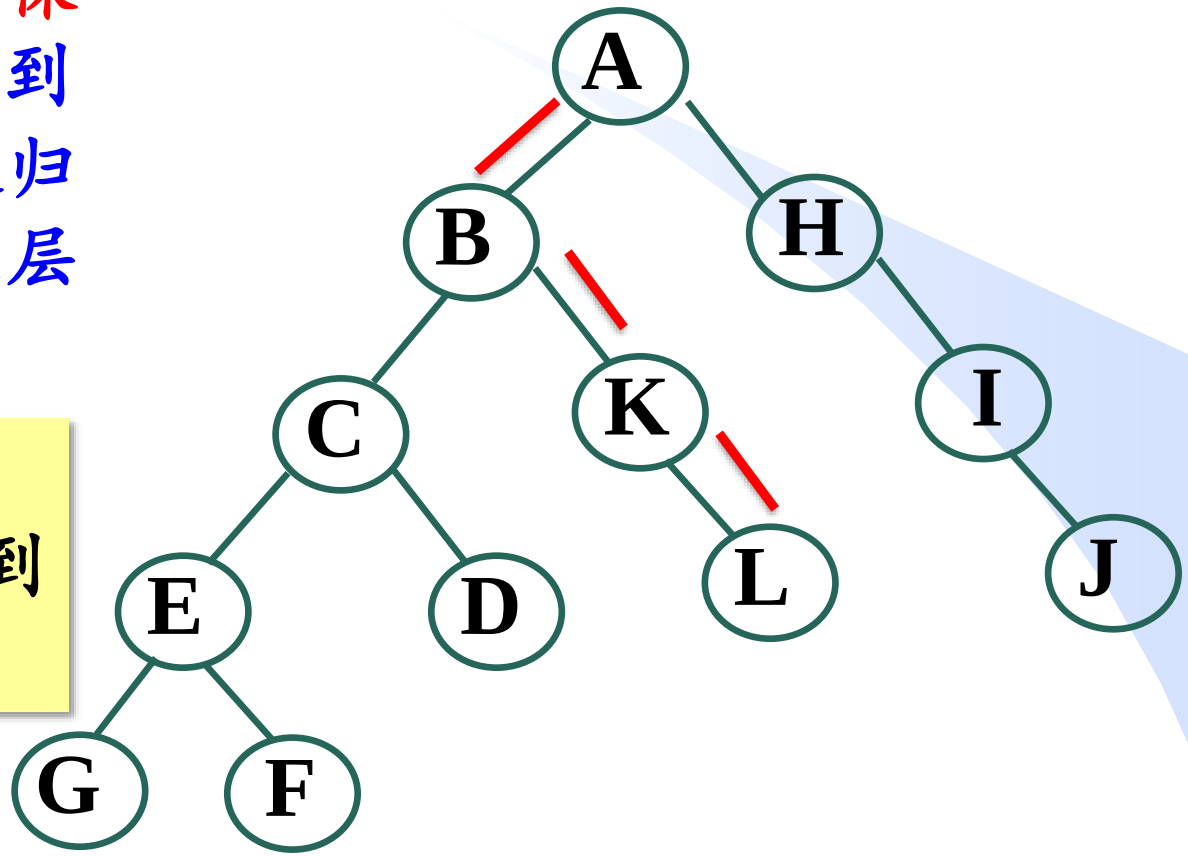
Path	*A*	*B*	*K*	*D*	*F*
	0	1	*k*=2	3	4

二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



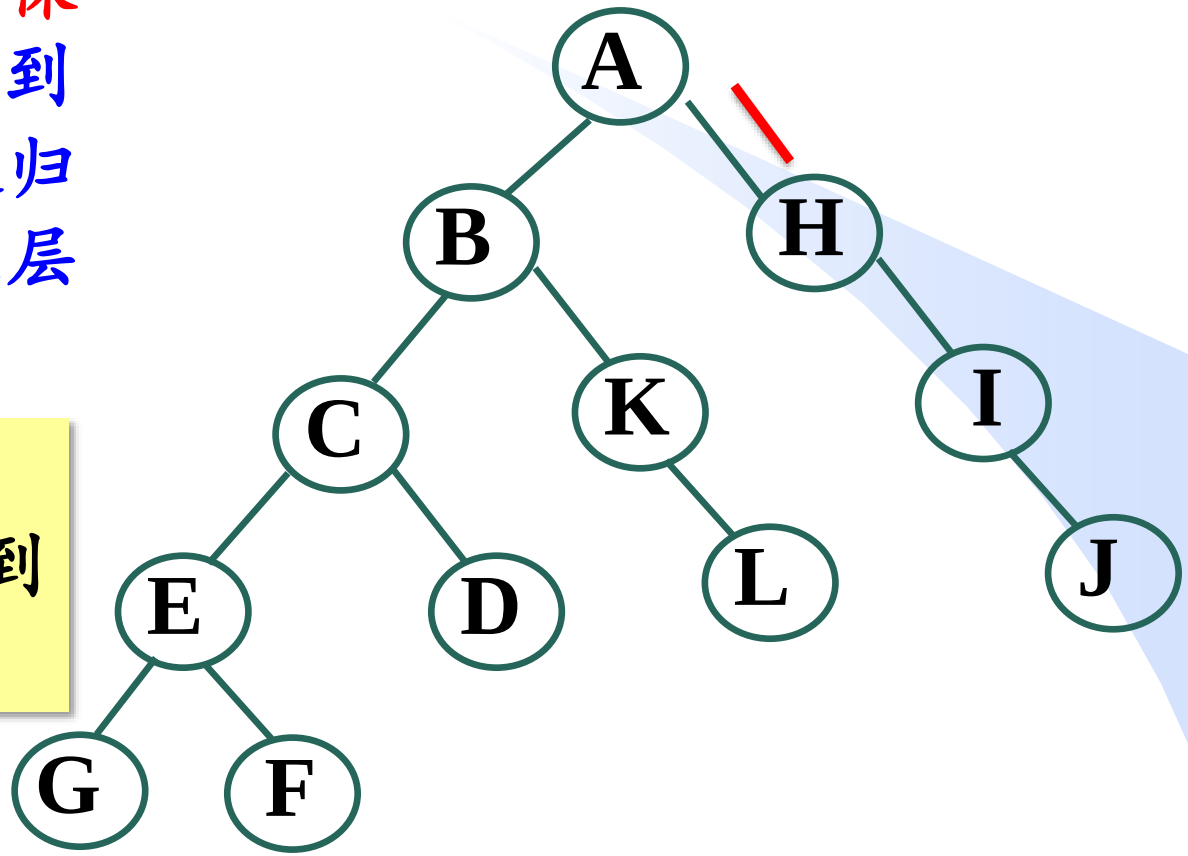
Path	*A*	*B*	*K*	*L*	*F*
	0	1	2	*k*=3	4

二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



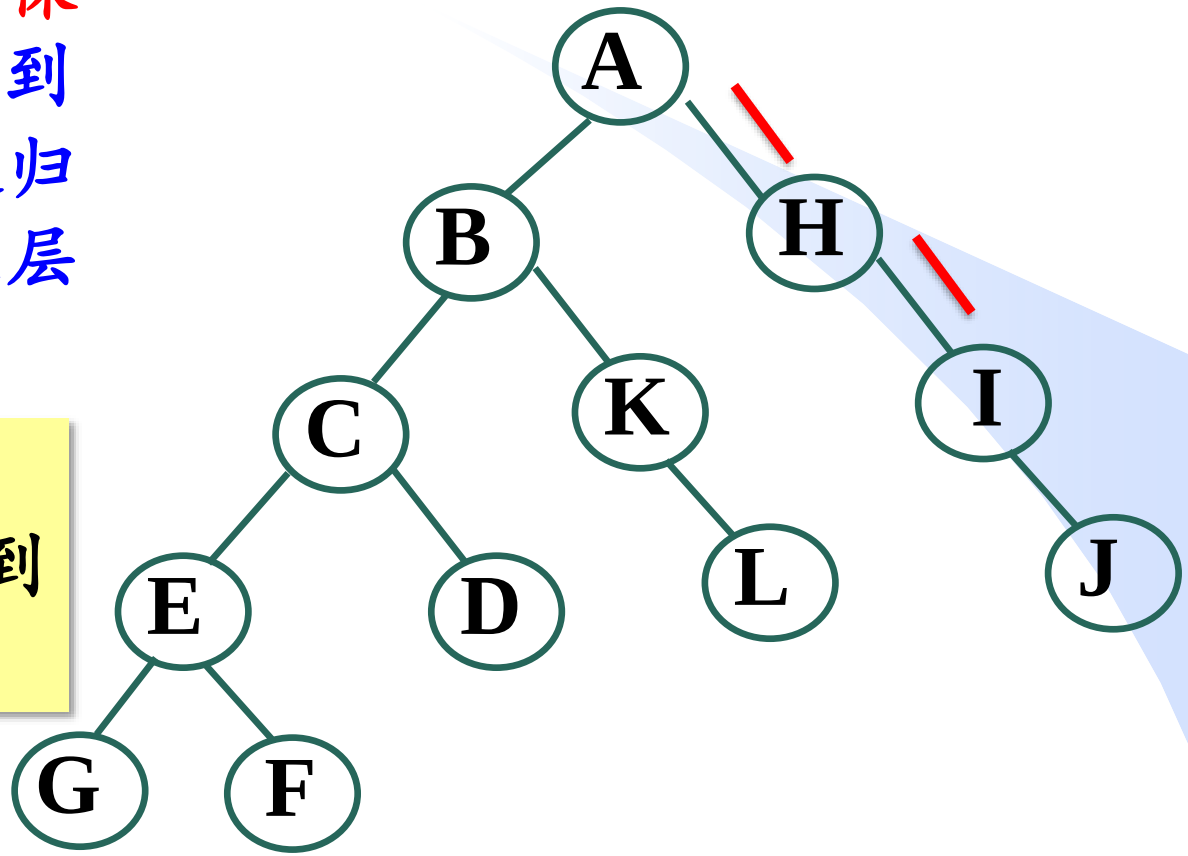
Path	*A*	*H*	*K*	*L*	*F*
	0	*k*=1	2	3	4

二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



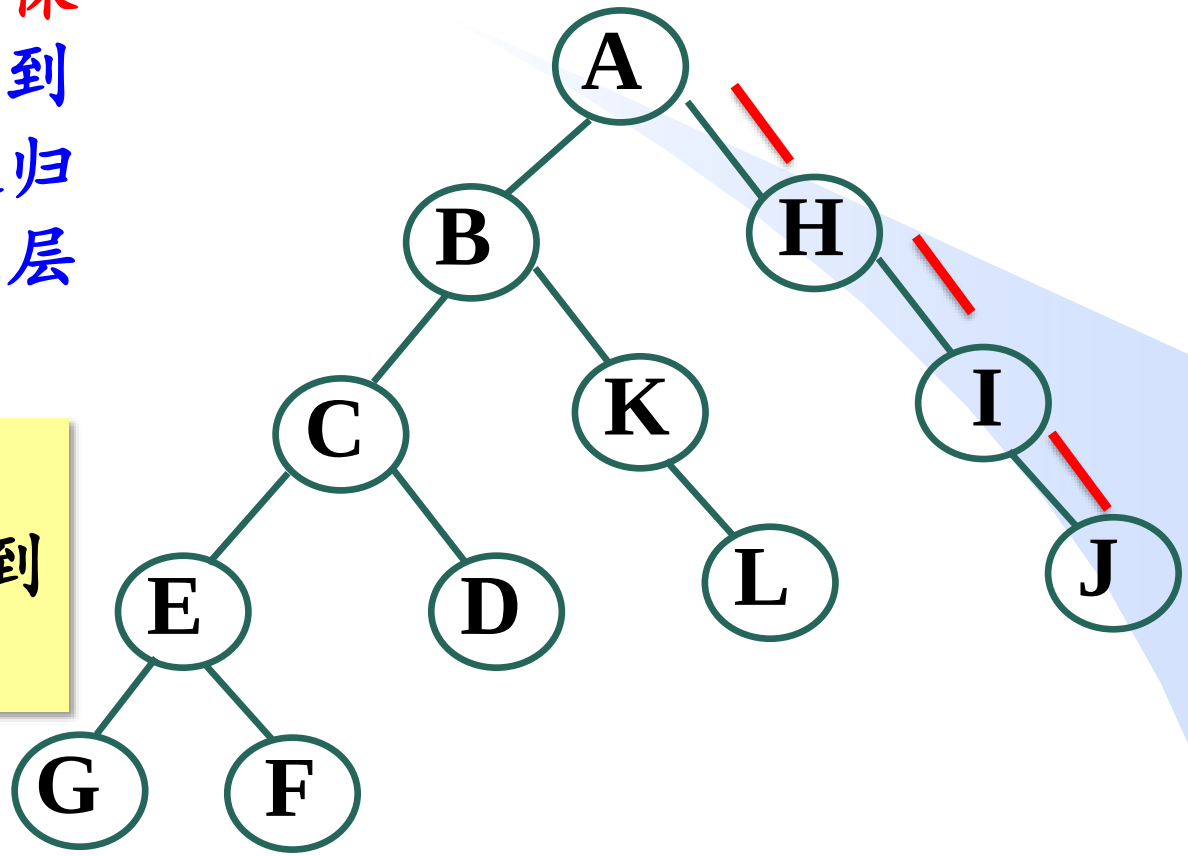
Path	*A*	*H*	*I*	*L*	*F*
	0	1	*k*=2	3	4

二叉树的路径

先根遍历，将沿途访问的结点保存在*Path*数组中，即得到从根到当前访问结点的一条路径。递归深度对应当前访问的结点所在层数，亦即当前路径的长度。

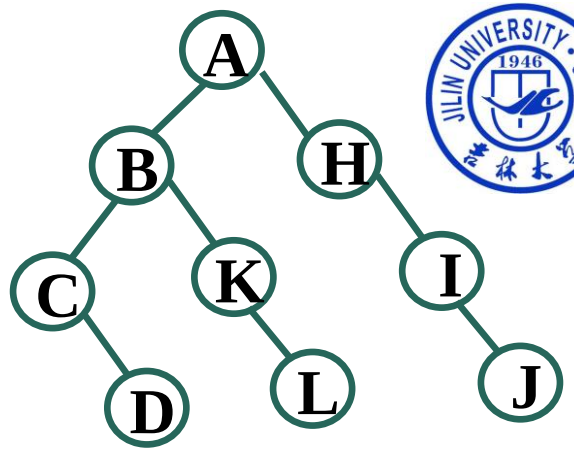
当前访问的结点存入*Path*[*k*]，则*Path*[0]…*Path*[*k*]即为从根到当前访问结点的路径。

*k*为先根遍历的递归深度



Path	*A*	*H*	*I*	*J*	*F*
	0	1	2	*k*=3	4

二叉树的路径




```
void findPath(TreeNode *t, int path[], int k){
    //输出从根到叶的所有路径，k为递归深度
    if(t==NULL) return;
    path[k] = t->data;
    if(t->left==NULL && t->right==NULL) { //找到一条路径
        for(int i=0; i<=k; i++)    //输出找到的路径
            printf("%d ", path[i]);
        printf("\n");
        return;
    }
    findPath(t->left, path, k+1);
    findPath(t->right, path, k+1);
}
```

先根遍历
回溯

初始调用
findPath(root,path,0)

二叉树的路径

```
bool findPath(TreeNode *t, int path[], int x, int k){
    //输出从根到数据值等于x的一条路径，k为递归深度
    if(t==NULL) return;
    path[k] = t->data;
    if(t->data == x) { //找到一条路径
        for(int i=0; i<=k; i++)    //输出找到的路径
            printf("%d ", path[i]);
        printf("\n");
        return true;
    }
    if(findPath(t->left, path, x, k+1)) return true;
    if(findPath(t->right, path, x, k+1)) return true;
    return false;
}
```

初始调用

findPath(root,path,x,0)