

# 树 Tree



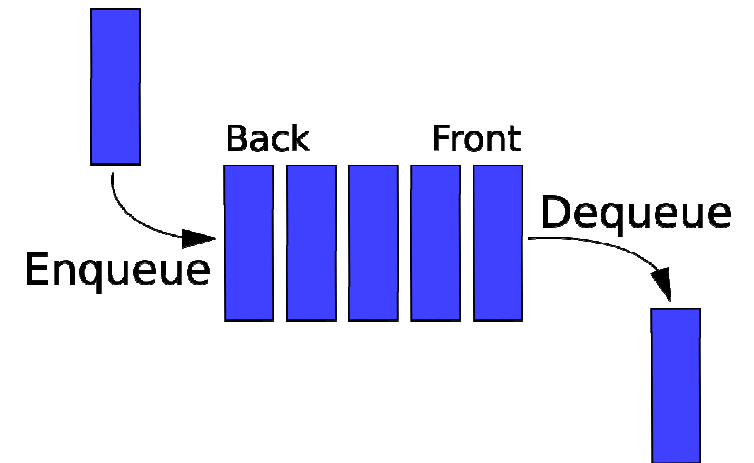
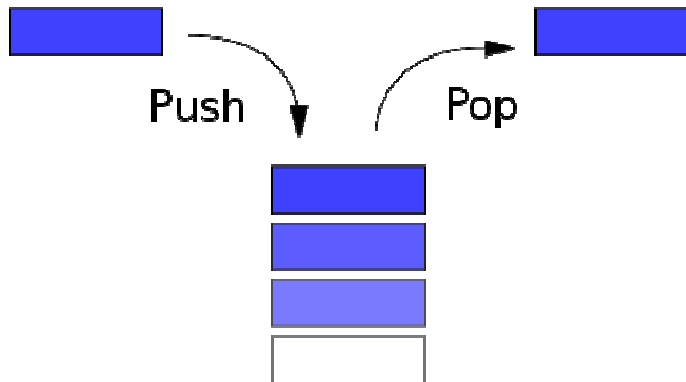
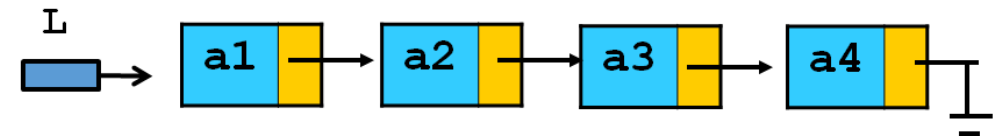
hwdong

博客: <https://xuepro.xcguan.net>

微博: 教小白精通编程

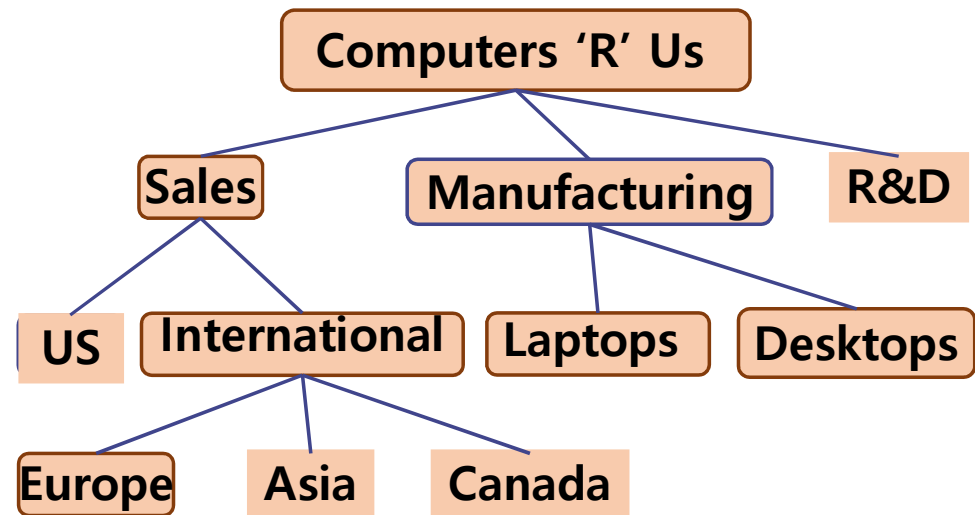
# 回顾：线性数据结构

每个元素最多只有一个“直接前驱元素”，最多只有一个“直接后续元素”



# 树的概念

- 树是一个层次结构的抽象模型
- 树由具有“父子关系”的数据元素构成
- 应用示例：
  - 组织结构
  - 文件系统
  - 决策树（机器学习）
  - 博弈树(蒙特卡罗树)



树中的数据元素也习惯称之为“结点”。

- 树的概念和类型定义
- 二叉树的概念和类型定义
- 二叉树的性质
- 二叉树的顺序表示
- 二叉树的链式表示
- 二叉树的遍历
- 哈夫曼编码
- 树和森林

# 树的概念

- 树的定义(Tree)

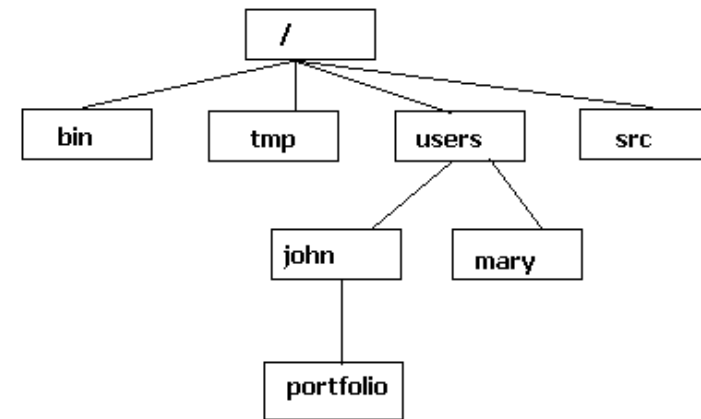
- 树是由 $n(n \geq 0)$ 个结点组成的有限集合

- 如果 $n=0$ ，称为空树

- 如果 $n>0$ ，则

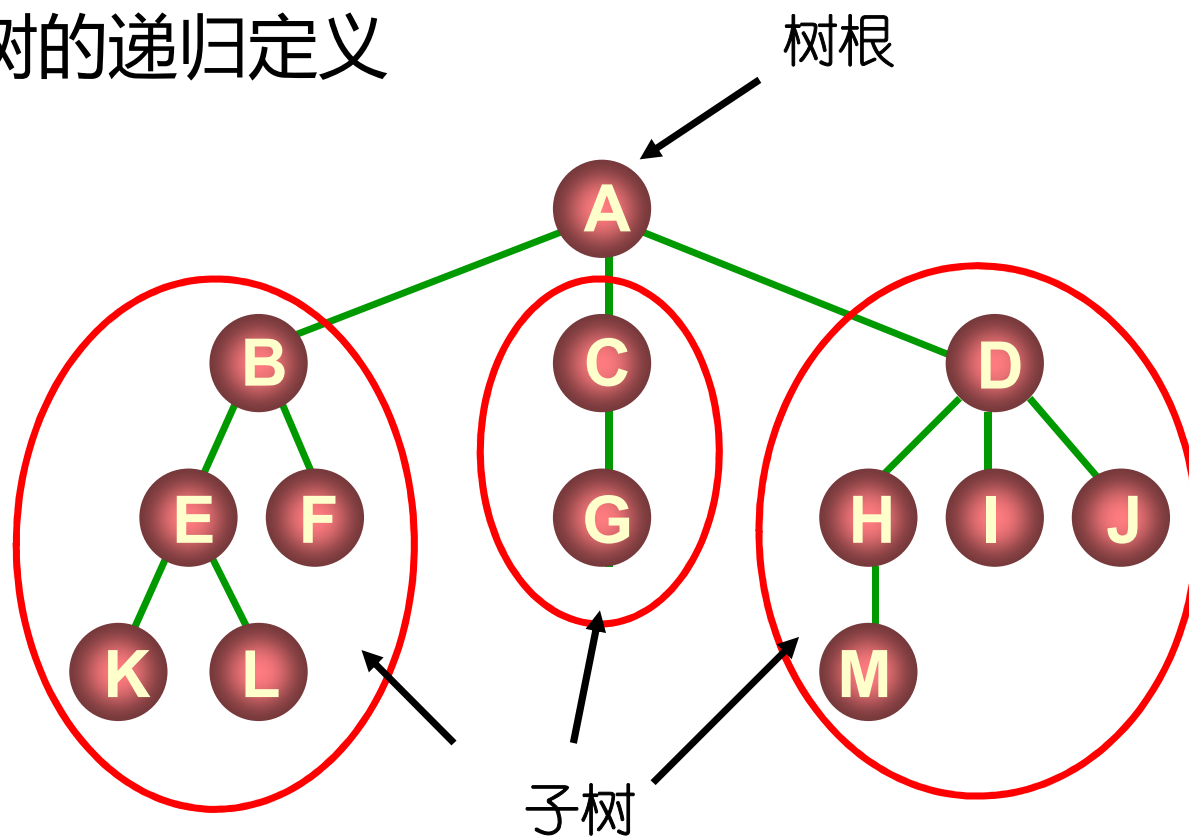
- 有一个特定的称之为根(root)的结点，它只有直接后继，但没有直接前驱

- 除根以外的其它结点划分为 $m(m \geq 0)$ 个互不相交的有限集合 $T_0, T_1, \dots, T_{m-1}$ ，每个集合又是一棵树，并且称之为根的子树



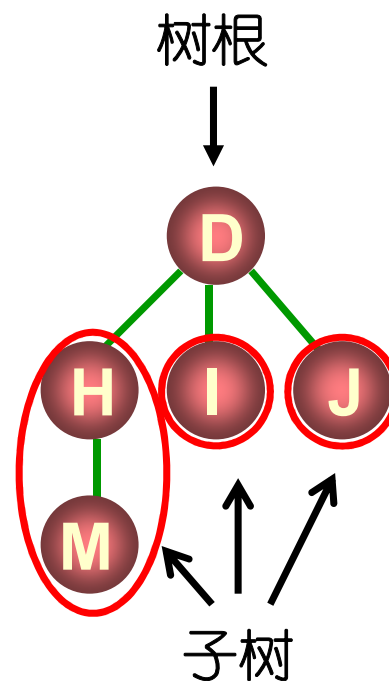
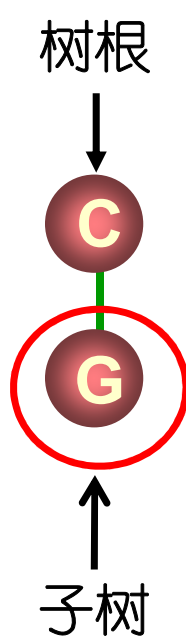
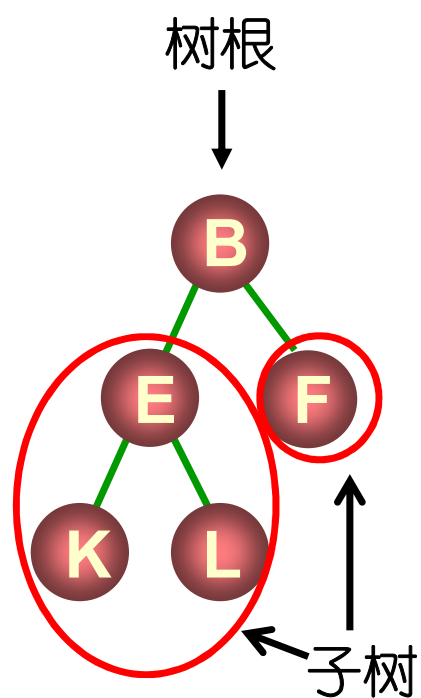
# 树的概念

- 树的递归定义



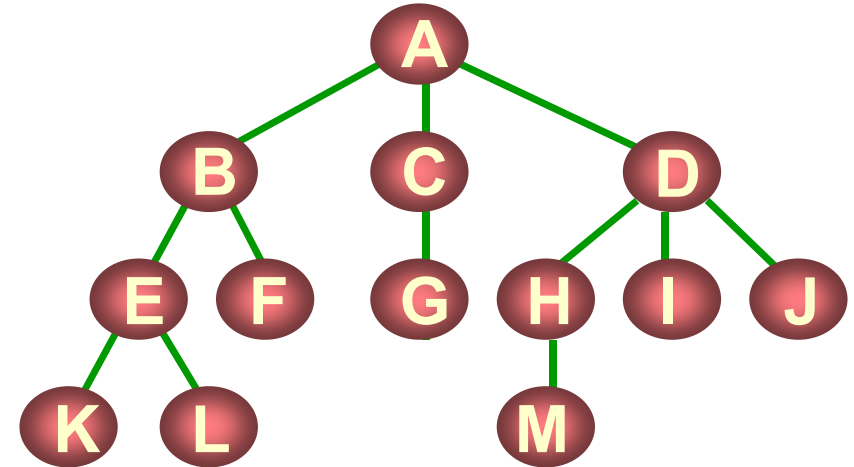
# 树的概念

- 树的递归定义



# 树的概念

- 树和线性结构的对比
  - 线性结构：一对一
  - 树结构：一对多

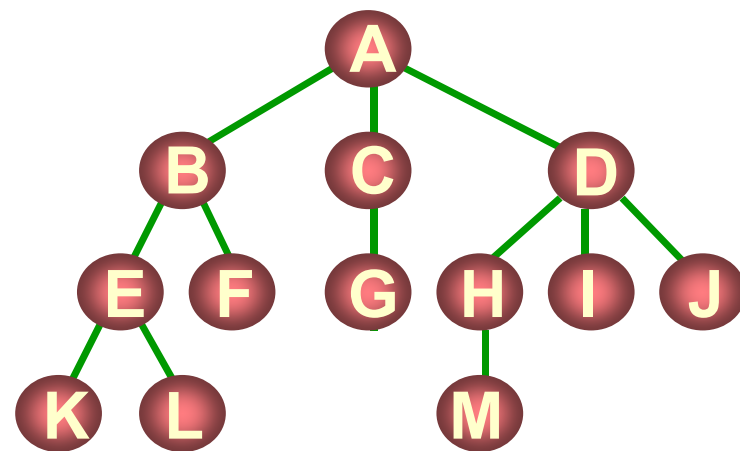


线性结构	树结构
第一个元素（无前驱）	根结点（无前驱）
最后一个元素（无后继）	多个叶子结点（无后继）
其它数据元素（一个前驱、一个后继）	树中的其它结点（一个前驱、多个后继）



# 树的概念

- **根**：唯一没有双亲的结点，其他结点都有唯一的双亲
- **结点的度**：子树的个数
- **树的度**：结点的度的最大值
- **分支结点**：度不为0的结点
- **叶结点**：度为0的结点
- **孩子**：某结点的子树的根
- **双亲**：该结点称为孩子的双亲
- **兄弟**：同一个双亲的孩子之间互为兄弟



# 树的概念

- 祖先：从根到该结点所经分支的所有结点

- 子孙：以某结点为根的子树中的任一结点

- 结点的深度：结点的祖先个数 + 1?

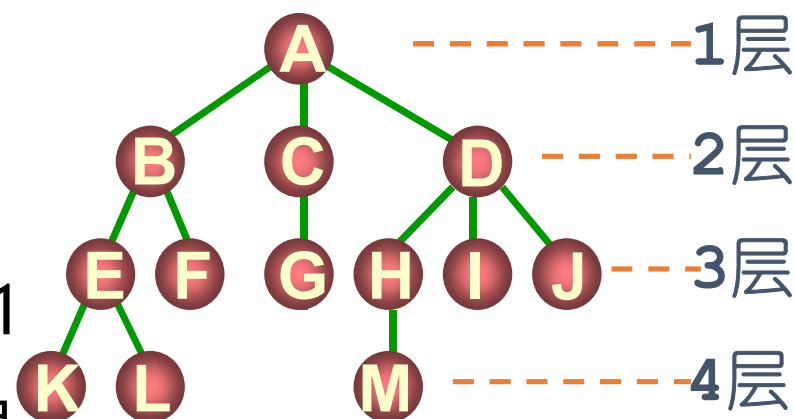
- 结点的高度：从结点到其子树叶子结点

经过的最大边数，叶子节点的高度为0?1

- 结点的层次：根为第一层，其孩子第二层...

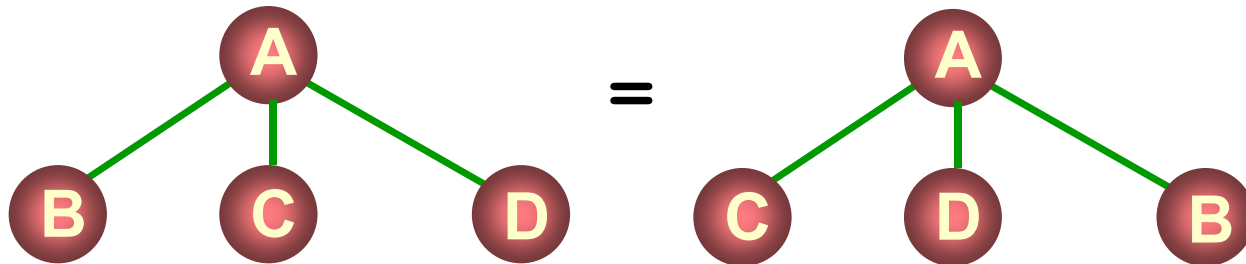
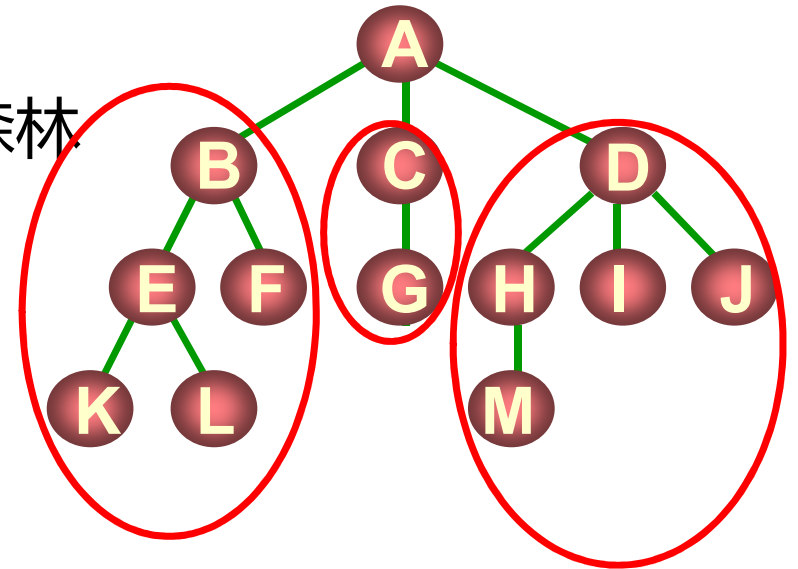
- 树的高度(深度)：树中结点的最大层次或结点的最大深度

- 堂兄弟：双亲在同一层的结点互为堂兄弟



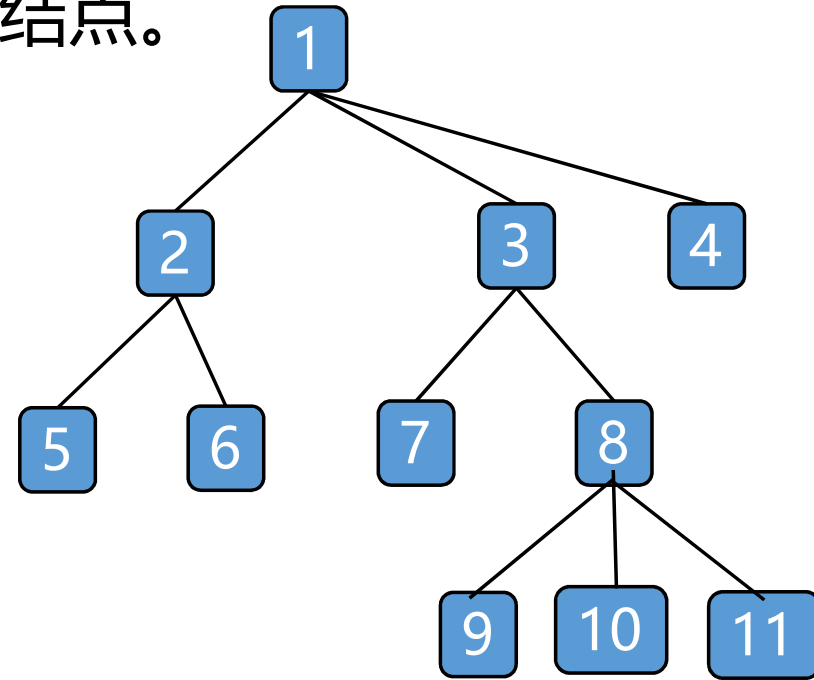
# 树的概念

- 森林：多个树的集合
- 子树森林：一个结点的所有子树构成的森林
- 有序树：每个结点的子树有次序之分
- 无序树：每个结点的子树无次序之分



## 练习：回答下列问题

- 将结点分类为根、其他内部结点、叶子结点。
- 画出结点3的子树森林
- 结点1,3,8的高度、深度是多少？
- 整个树的高度是多少？
- 结点1,3,8的祖先是哪些？
- 结点1,3,8的子孙是哪些？



# 树的类型定义

ADT Tree{

元素关系：父子关系(最多一个双亲可有多个孩子)

操作:

int **init** (T)    //初始化一颗树

Node **root**(T) //返回树的根结点

int **depth**(T) //返回树的高度

int **size**(T)    //返回树中结点个数

Node **parent**(T, Node p)    //返回结点p的双亲

Node[] **children**(T, p) //返回结点p的所有孩子

## 树的类型定义

Node[] **silbings** (parent, p) //返回结点p的所有兄弟

Node **find**(T, condition) //查找满足条件的结点

void **traverse**(T, visit(),way) //遍历树，用visit处理每个结点  
// way是遍历方法（深度、广度）

void **copy**(dstT, srcT) //复制树

void **compare**(dstT, srcT,way) //比较树,way是类型(相同、相似)  
visit()做什么用?  
1) 打印 2) 统计 3) ...

Tree **create**(definition); //从输入definition中创建树

## 树的类型定义

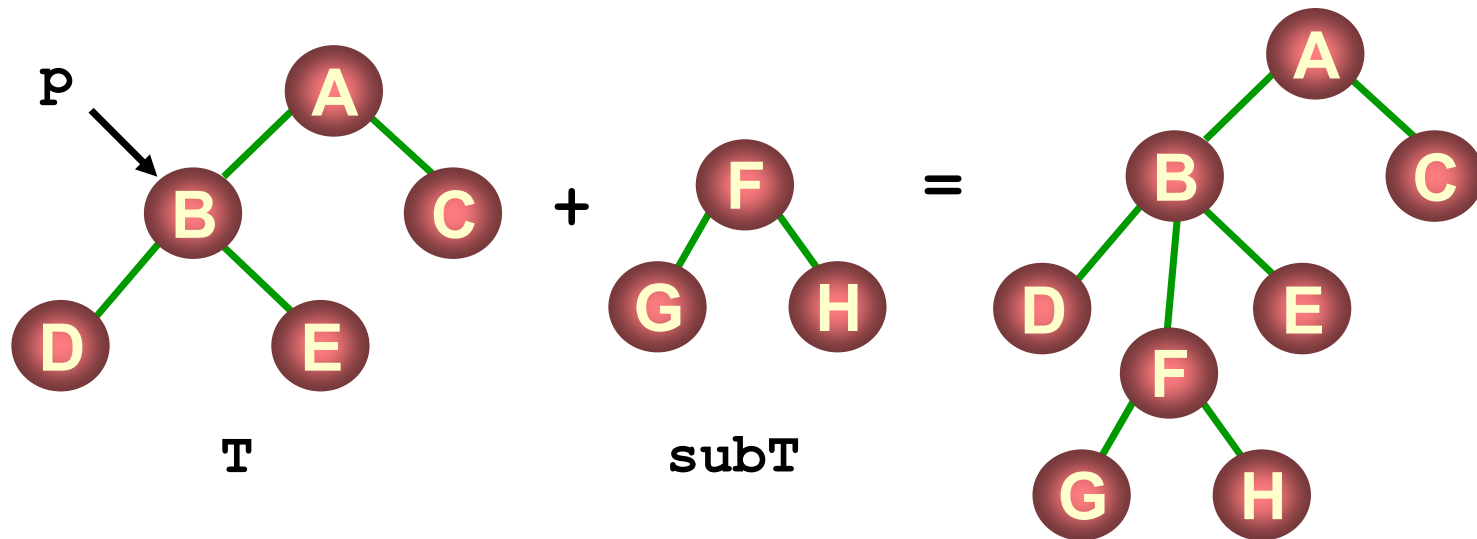
int **insert**(T, p, i, subT) //插入子树成为p的第i个子树

Tree **delete**(T, p, i) //删除子树的第i个子树，并返回该子树

void **insertNode**(T, p, i, e) //插入元素e成为子树的第i个孩子

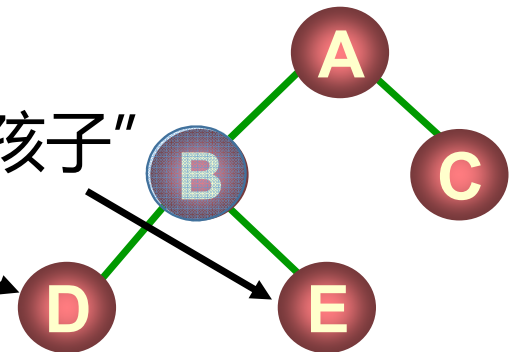
void **deleteNode**(T, p, i) //只删除子树的第i个孩子结点

}



# 二叉树的概念

- 每个结点最多有2棵子树（即度 $\leq 2$ ）的有序树  
一个结点的2个孩子分别称为“左孩子”和“右孩子”
- 为什么要引入二叉树？
  - 树太一般，子树的个数无限制，表示困难
  - 事实上很多问题最多只需要用二叉树表示
  - 树和森林可以转化为二叉树

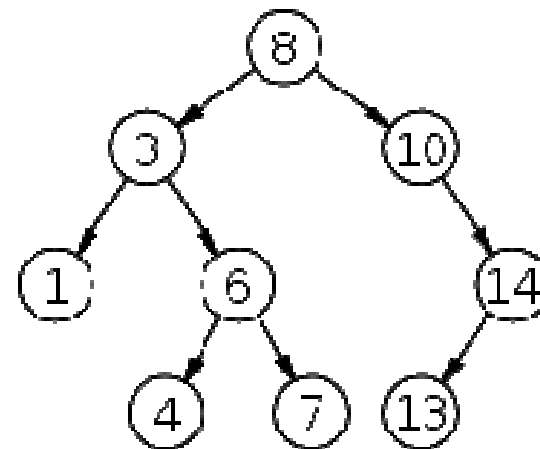




# 二叉树的概念

- 二叉排序树或二叉搜索树 (Binary Search Tree, 简称**BST**)
  - 每个结点包含一个唯一的关键字
  - 任何结点的左子树上的关键字  $<$  该结点的关键字
  - 任何结点的右子树上的关键字  $>$  该结点的关键字
  - 不允许存在相同关键字的结点

实际使用中二叉树几乎都是二叉排序树，因为这样才能比线性结构效率高，如搜索



# 二叉树的类型定义： 扩展树的类型定义

ADT Tree{

元素关系：父子关系，每个结点最多2个孩子

操作：

int **init** (T)    //初始化一颗二叉树

int **insertNode**(T, p, e, LR)    //插入数据元素

int **deleteNode**(T, P, e, LR)    //删除数据元素

int **insertTree**(T, p, subT, LR)    //插入子树

int **deleteTree**(T, p, subT, LR)    //删除子树

## 二叉树的类型定义

```
Node parent(T, p)    //得到结点p的双亲结点
Node LeftChild(T, p) //得到结点p的左孩子
Node RightChild(T, p) //得到结点p的右孩子
int hasChild(T, p)   //判断结点p有几个孩子
int LevelOrderTraverse(T, visit()) //层次遍历
int PreOrderTraverse(T, visit())   //先序遍历
int InOrderTraverse(T, visit())    //中序遍历
int PostOrderTraverse(T, visit())  //后序遍历
```

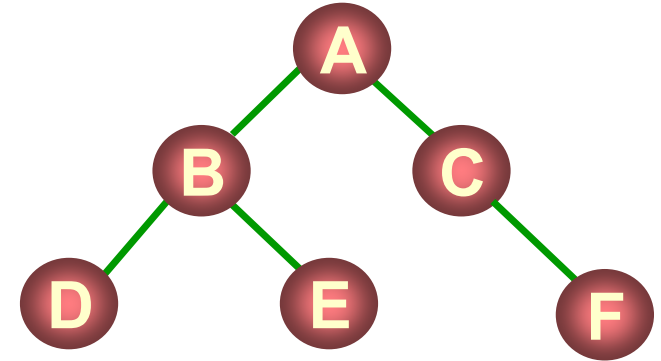
```
}
```

## int LevelOrderTraverse(T, visit()) //层次遍历

- **遍历**：按某种方式访问每个结点（元素）且每个结点只访问一次。
- 如打印出文件系统的所有文件夹（文件）、产生各种名单（家族族谱、组织会议、捐款名单）

Visit函数可以是打印/输出/统计，  
比如统计叶子结点个数

- 遍历就是将树型结构转化为线性结构



A B C D E F

# 二叉树的性质：性质1

- 在二叉树的第 $i$ 层最多有 $2^{i-1}$ 个结点 ( $i \geq 1$ )

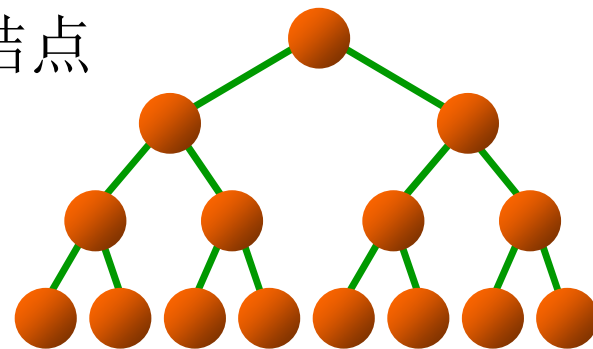
证明：(数学归纳法)

1) 当 $i=1$ 时，显然成立

2) 设当 $i=k$ 时成立，即第 $k$ 层最多 $2^{k-1}$ 个结点

当 $i=k+1$ 时，由于二叉树的每个结点最多有2个孩子，所以第 $k+1$ 层最多有 $2 * 2^{k-1} = 2^{(k+1)-1}$ 个结点

所以对于任意  $i$  ( $i \geq 1$ )，二叉树的第 $i$ 层最多有 $2^{i-1}$ 个结点



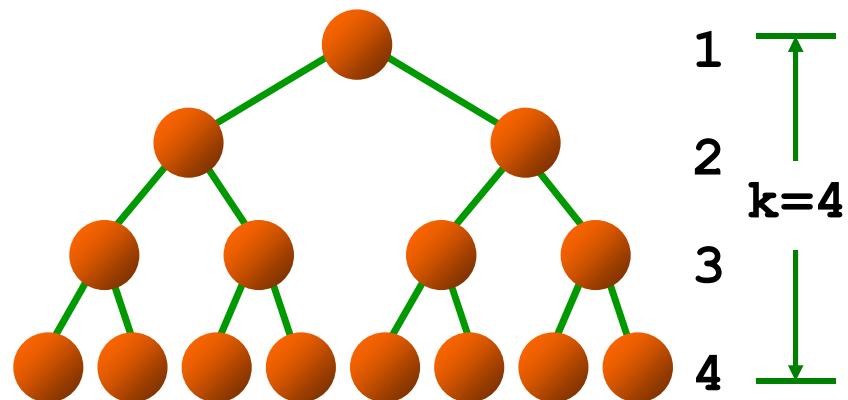
## 二叉树的性质: 性质2

- 深度为k的二叉树最多有 $2^k - 1$ 个结点 ( $k \geq 1$ )

证明:

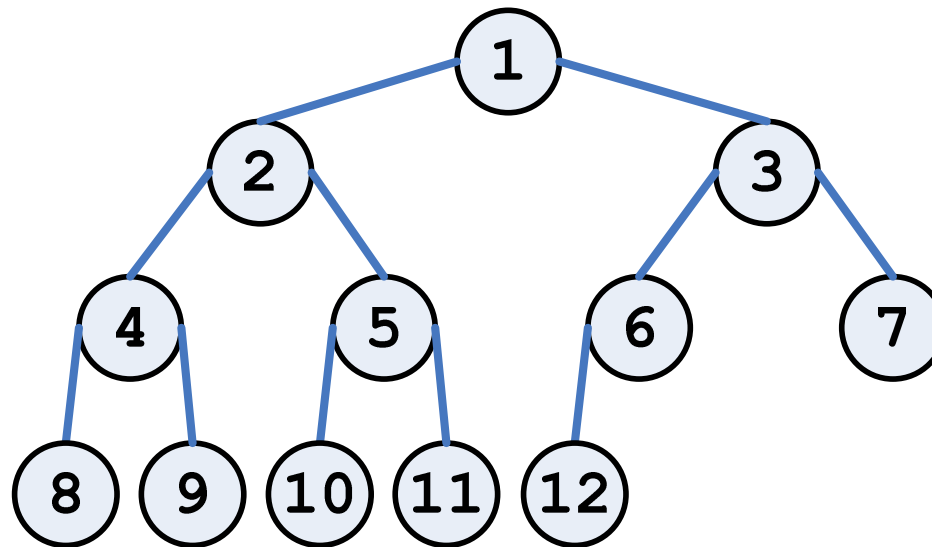
- 由性质1可知: 第i层最多有 $2^{i-1}$ 个结点
- 所以总的结点数最多为

$$\sum_{i=1}^k 2^{i-1} = 2^k - 1$$



## 二叉树的性质: 性质3

- 对任何一棵二叉树T,
- 若叶结点数(即度为0的结点数)为 $n_0$
- 度为2的结点数为 $n_2$
- 则 $n_0 = n_2 + 1$



• 方程1:  $n = n_0 + n_1 + n_2$

- 结点无外乎度为0、1、2三种情况

• 方程2:  $n = B + 1$

- 除了树根, 其余每个结点“上方”都有一个分支

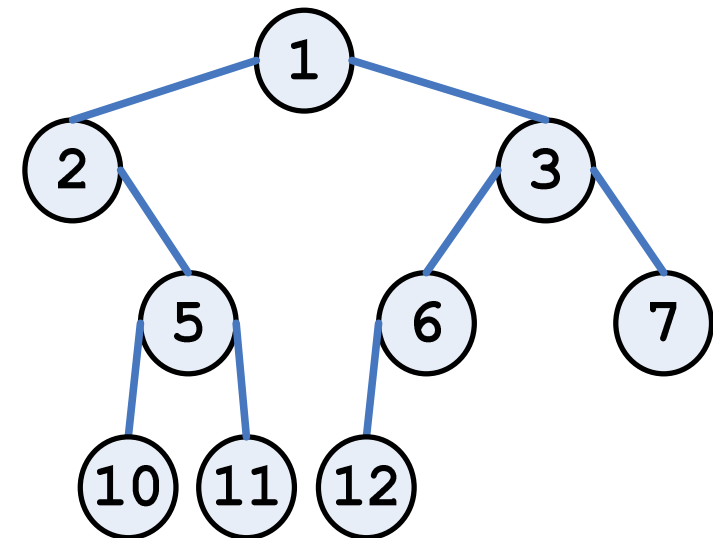
• 方程3:  $B = n_1 + 2n_2$

- 度为2的结点“下方”有2个分支

- 度为1的结点“下方”有1个分支

- 度为0的结点“下方”有0个分支

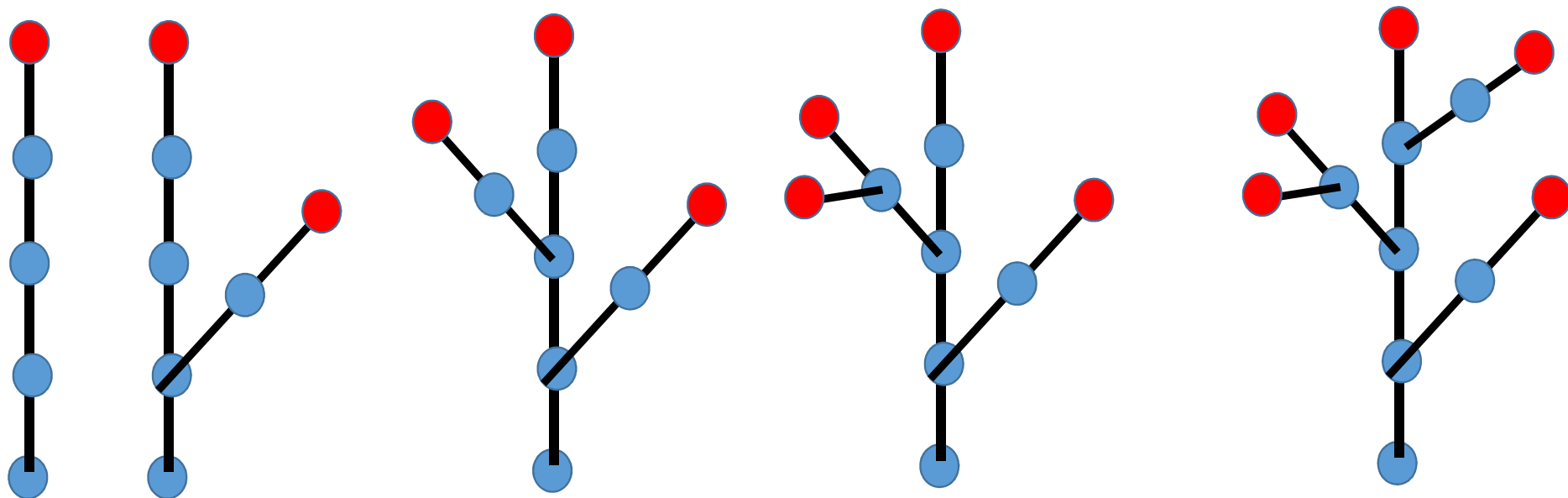
$$\begin{cases} \mathbf{n} = \mathbf{n}_0 + \mathbf{n}_1 + \mathbf{n}_2 \\ \mathbf{n} = \mathbf{B} + \mathbf{1} \\ \mathbf{B} = \mathbf{n}_1 + 2\mathbf{n}_2 \end{cases}$$





$$n_0 = n_2 + 1$$

- 也可以这么理解：假设没有度为2的结点，则只有1个度为0的结点，满足： $n_0 = n_2 + 1$
- 每增加1个度为2的，就增加1个度为0的结点，也就是公式两边同等增加。



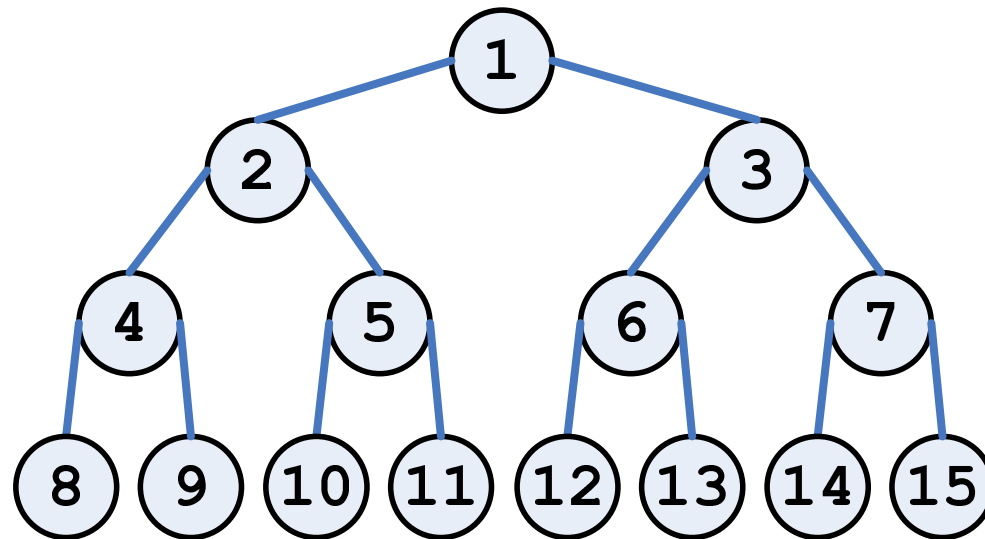
## 推广到一般的树

- 叶子结点和度为2以上结点数目这种关系可以推广到一般的树：

$$n_0 = (k-1)n_k + (k-2)n_{k-1} + \dots + 2n_3 + n_2 + 1$$

# 二叉树的性质

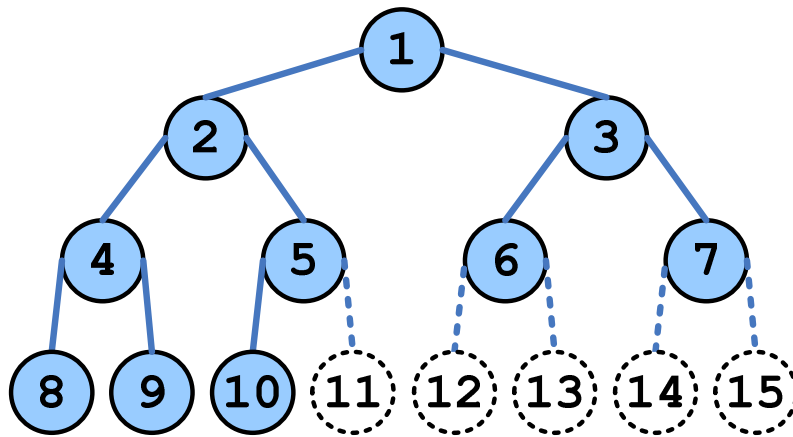
- 特殊形态的二叉树
  - 满二叉树（**Full Binary Tree**）
    - 深度为k，结点数为 $2^k-1$
    - 即结点数达到最大值



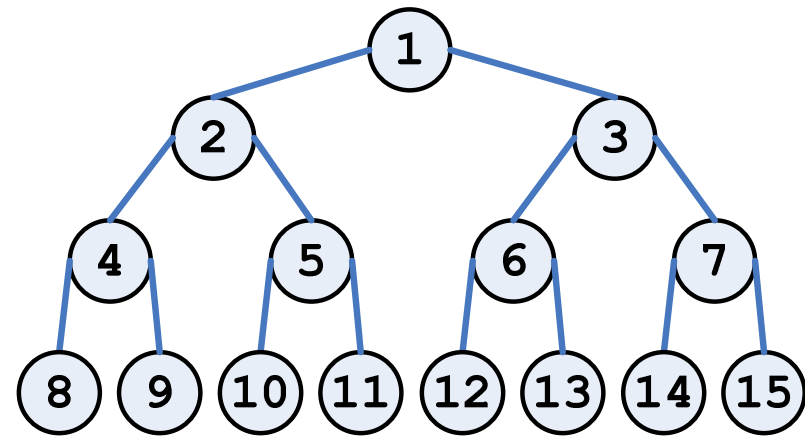
# 二叉树的性质

## 完全二叉树（Complete Binary Tree）

从上到下，从左到右对结点编号,该树的每一个结点的编号都与一个同深度的满二叉树的结点一一对应



完全二叉树

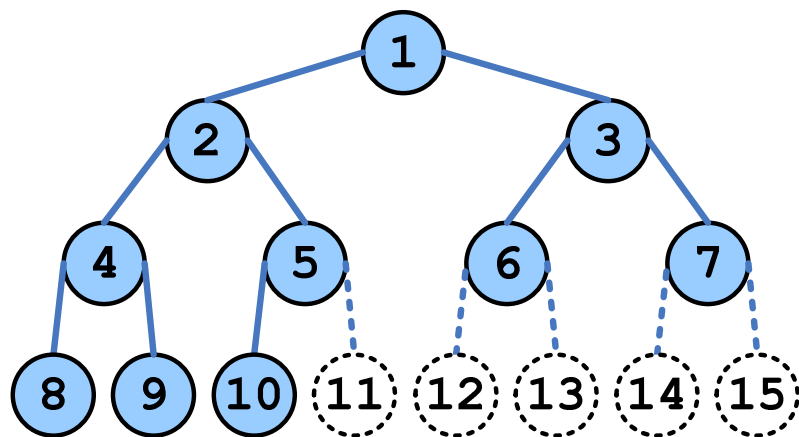


满二叉树

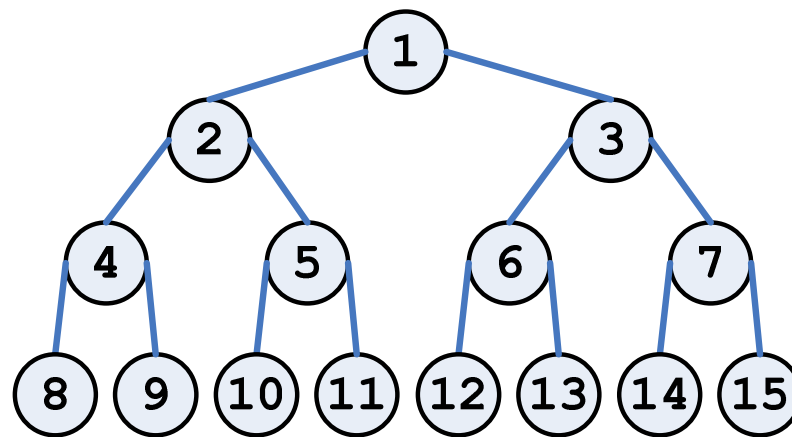
# 二叉树的性质

理解**1**：和满二叉树相比，就是最底层最右边连续缺少一些结点

理解**2**：除最后一层外都是满的，最后一层的结点一律靠左，中间没有间断



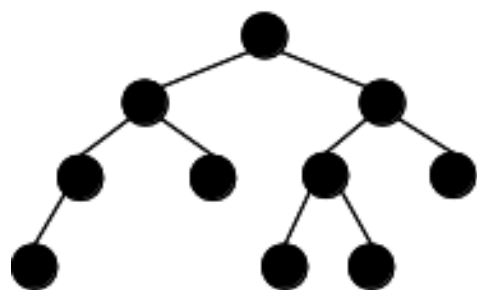
完全二叉树



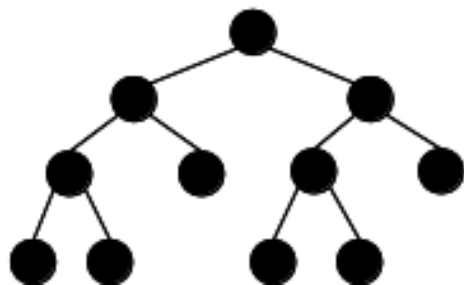
满二叉树

## 练习

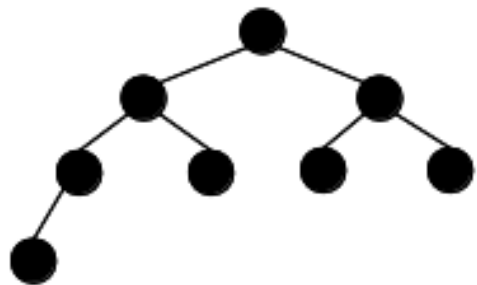
- 识别二叉树的类型（满二叉树、完全二叉树、非完全二叉树）



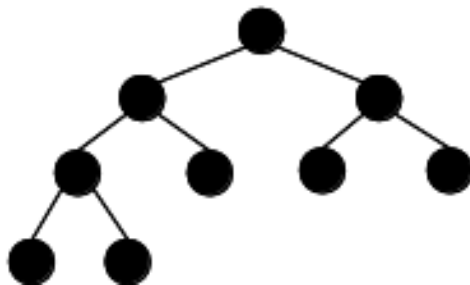
A



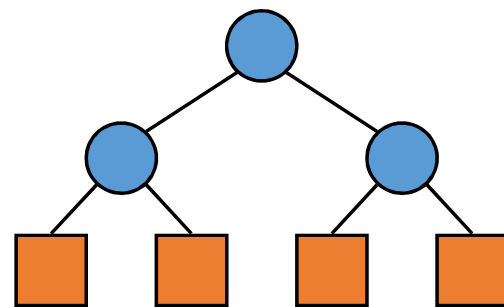
C



B



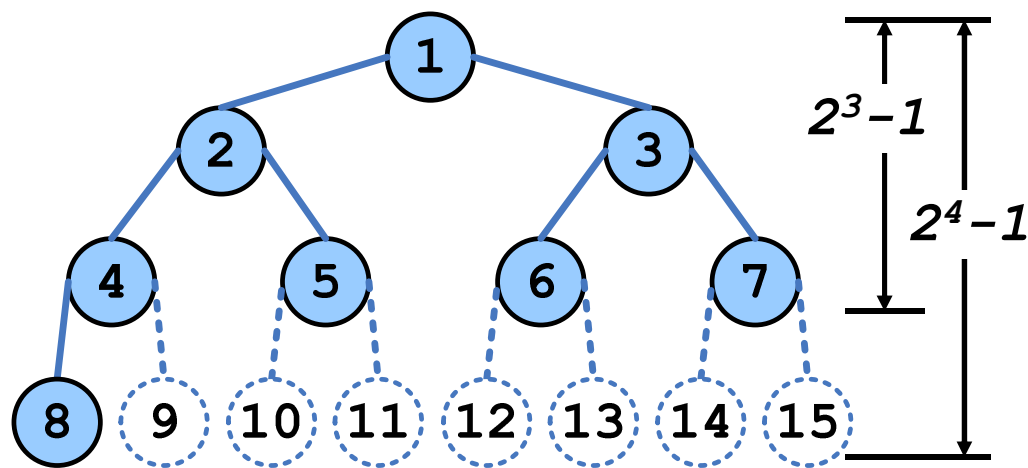
D



E

## 二叉树的性质：性质4

- 具有 $n$ 个结点的完全二叉树的深度为  $\lfloor \log_2 n \rfloor + 1$



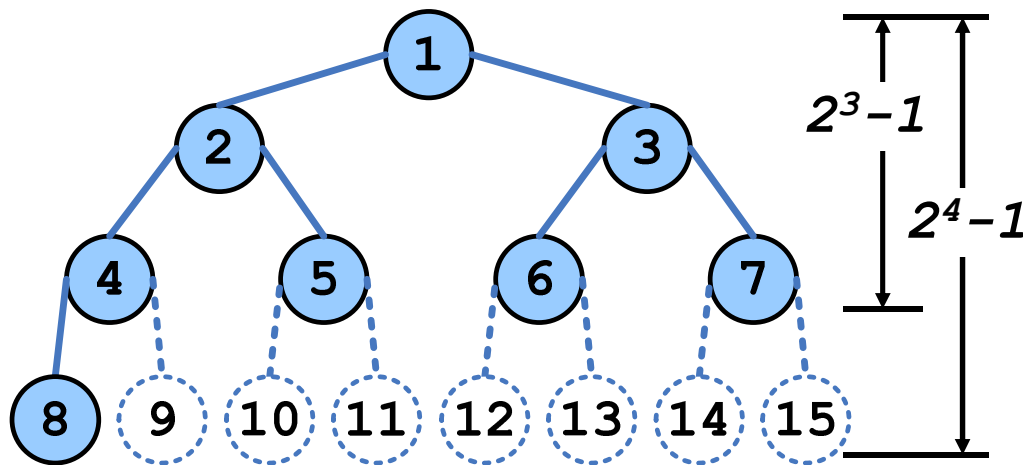
## 二叉树的性质：性质4

- 证明：

- 设深度为 $k$ ，则： $2^{k-1} \leq n < 2^k$

- 两边求对数： $k-1 \leq \log_2 n < k$

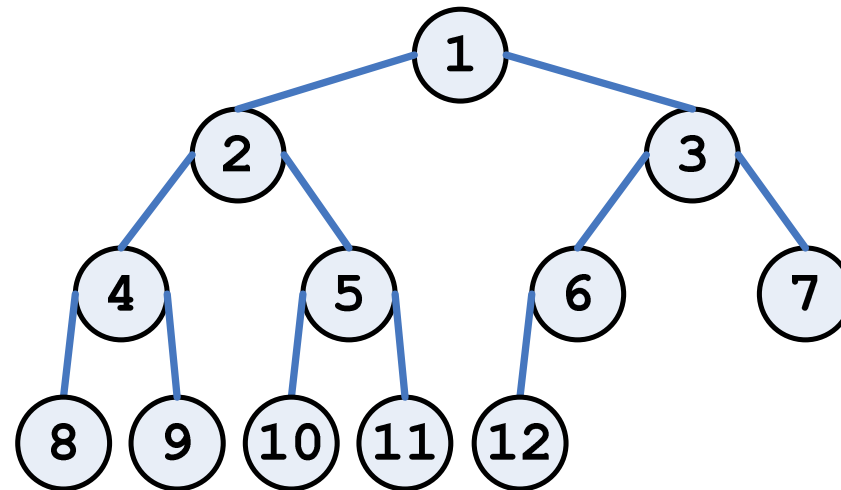
- 所以： $k = \lfloor \log_2 n \rfloor + 1$





## 二叉树的性质：性质5

- 若将一棵有 $n$ 个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号：
  - (1) 若 $i=1$ ，则结点 $i$ 是树根，无双亲
  - 若 $i>1$ ，则其双亲是节点  $\lfloor i/2 \rfloor$

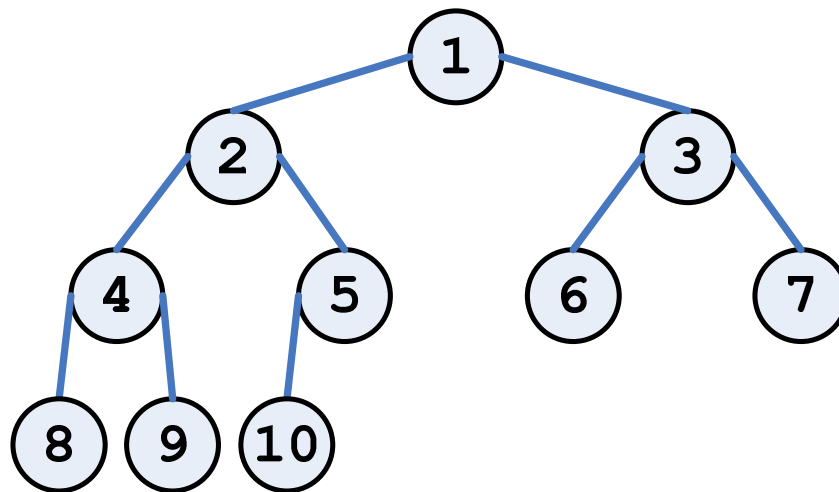


## 二叉树的性质：性质5

(2)若 $2i > n$ ，则结点 $i$ 无左孩子（即 $i$ 为叶结点）  
否则其左孩子为 $2i$

**理解：**

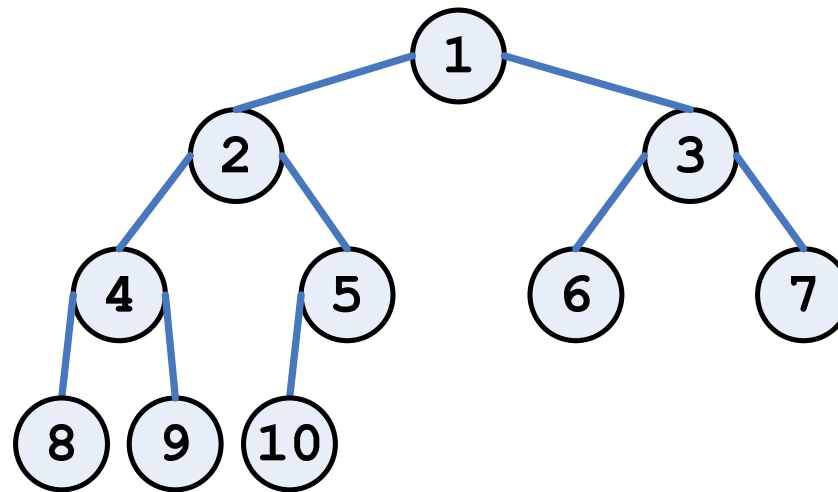
结点 $i$ 如果有左孩子的话，其编号应该为 $2i$   
如果 $2i > n$ ，则左孩子不存在



## 二叉树的性质：性质5

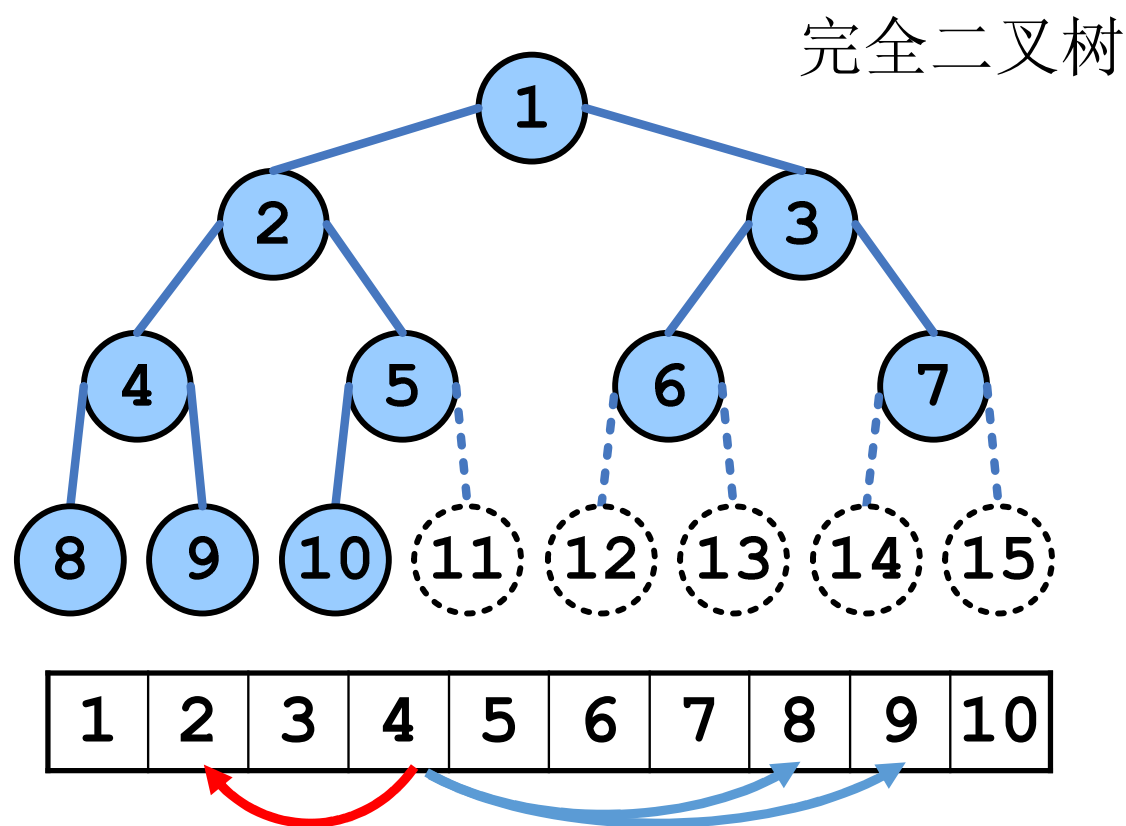
(3) 若 $2i+1 > n$ ，则结点 $i$ 无右孩子  
否则其右孩子为 $2i+1$

由(2)(3)可以推导出(1)



## 二叉树的顺序表示

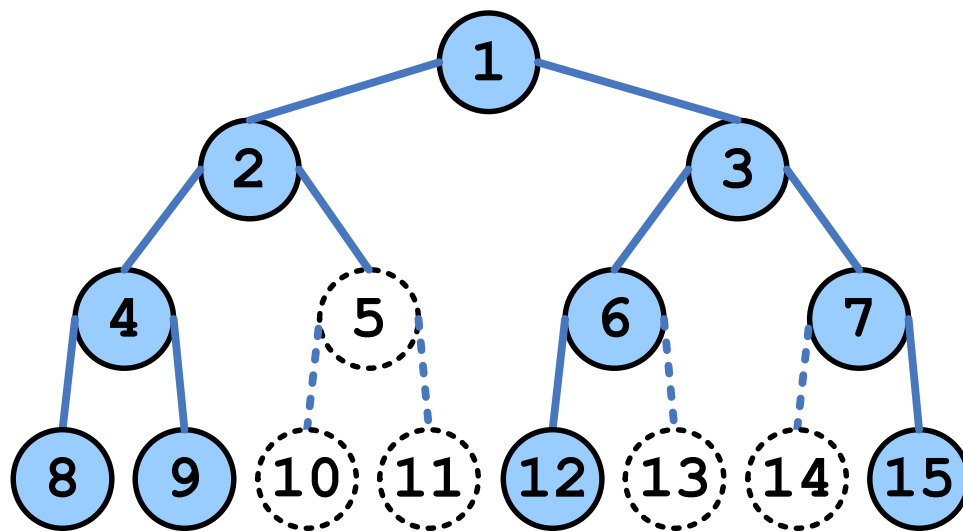
- 用一维数组存储二叉树的所有元素，按照满二叉树对应结点的编号确定存放位置。



## 二叉树的顺序表示

- 用一维数组存储二叉树的所有元素，按照满二叉树对应结点的编号确定存放位置。

一般二叉树



1	2	3	4	0	6	7	8	9	0	0	12	0	0	15
---	---	---	---	---	---	---	---	---	---	---	----	---	---	----

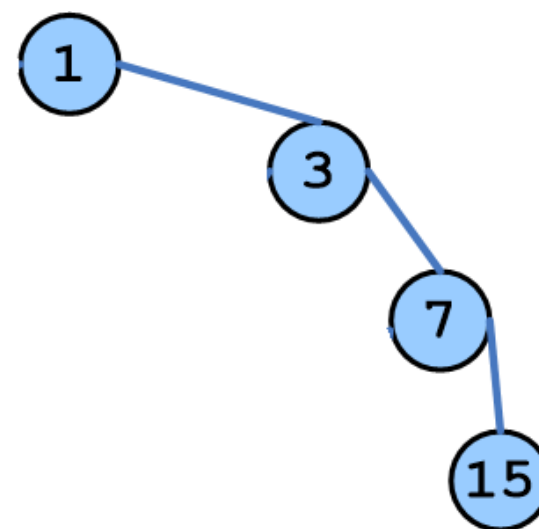
## 二叉树的顺序表示

- 深度为k的二叉树，最少只有k个结点

却需要 $2^k - 1$ 个存储单元

空间利用率:  $k / (2^k - 1)$

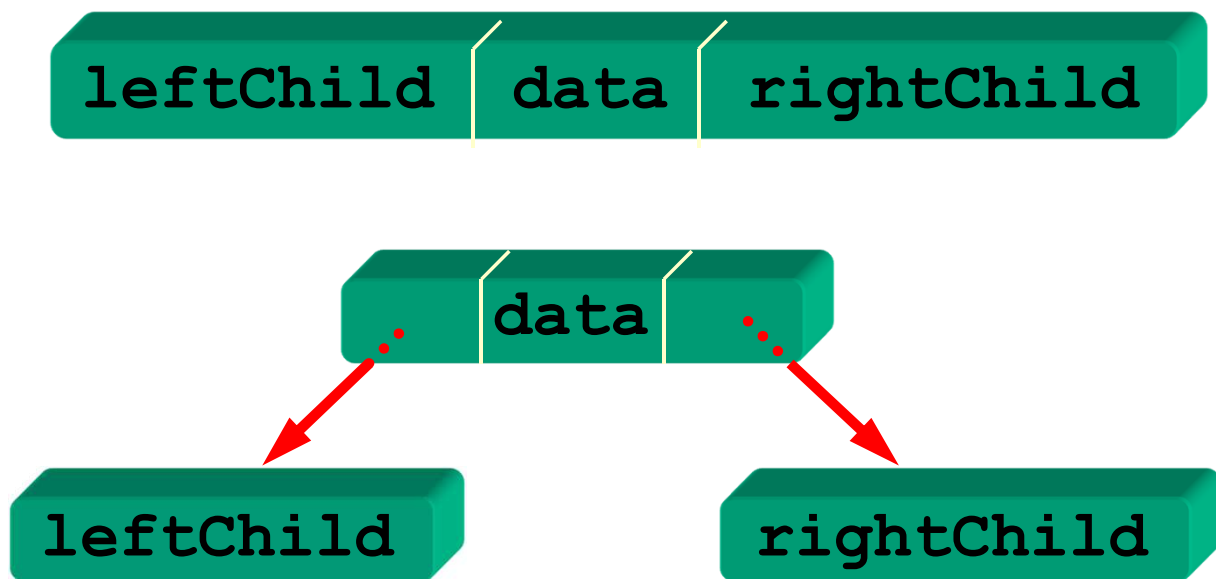
极端情形:单支树



1	0	3	0	0	0	7	0	0	0	0	0	0	0	15
---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

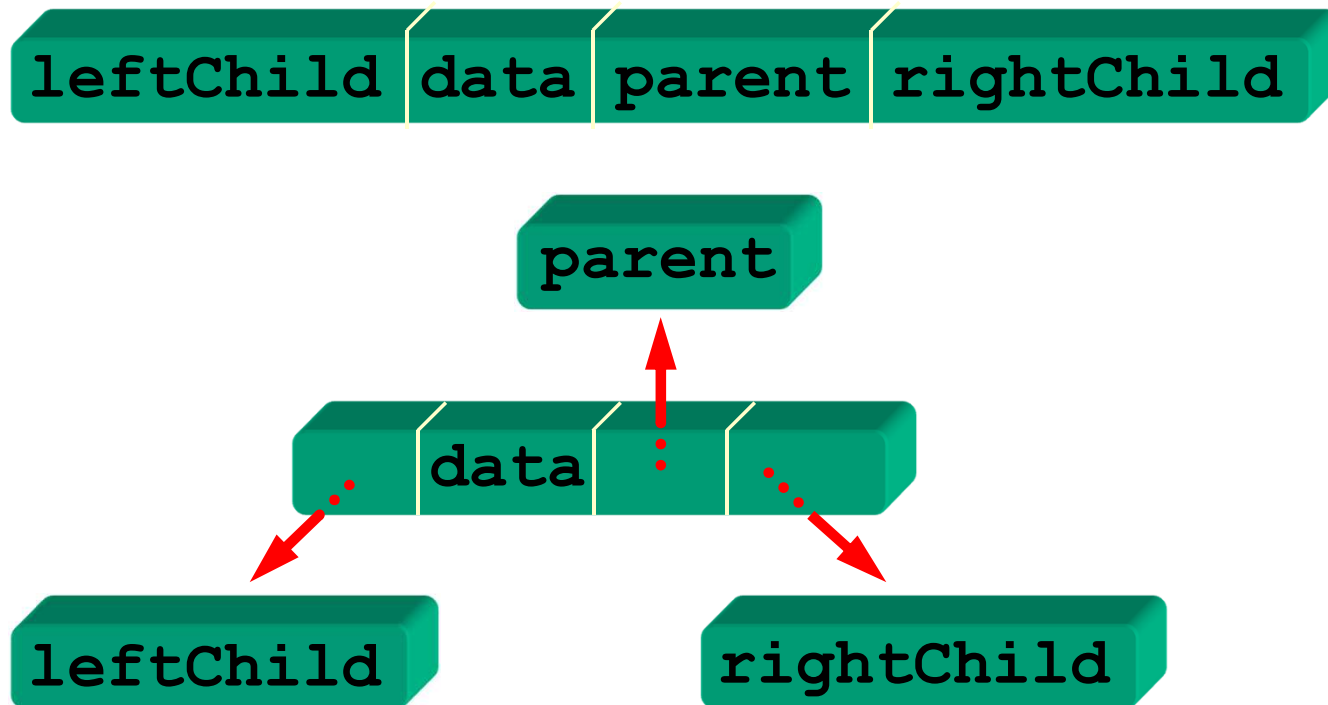
# 二叉树的链表表示

- 二叉链表：每个结点包括数据本身和、左右孩子指针



## 二叉树的链表表示

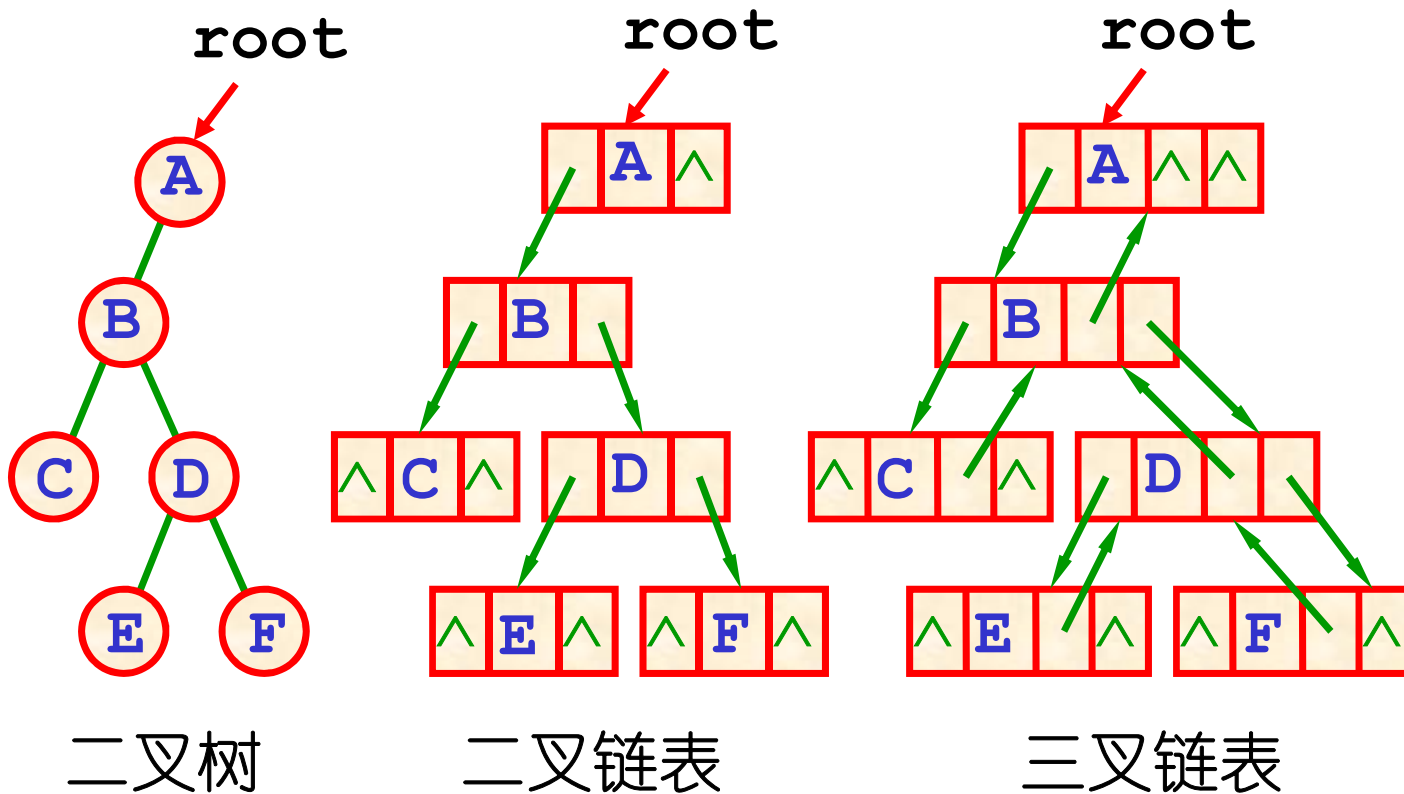
- 二叉链表：每个结点包括数据本身、左右孩子指针、双亲指针





# 二叉树的链表表示

- 例子



# 二叉树的链表表示

- 二叉链表的**结点**

二叉链表结点

```
typedef struct binode{  
    EType data;  
    struct binode *lchild, *rchild;  
} BiNode;  
typedef BiNode* BiTree;
```

三叉链表结点

```
typedef struct binode{  
    EType data;  
    struct binode *lchild, *rchild,  
    *parent;  
} BiNode;  
typedef BiNode* BiTree;
```

## 各种表示的比较

- 对一般二叉树，顺序表示空间浪费大。二叉链表比三叉链表和顺序表示法空间利用率高！

假如data占10个字节，指针为4个字节，则

二叉链表利用率： $10/18 = 5/9$

三叉链表利用率： $10/22 = 5/11$

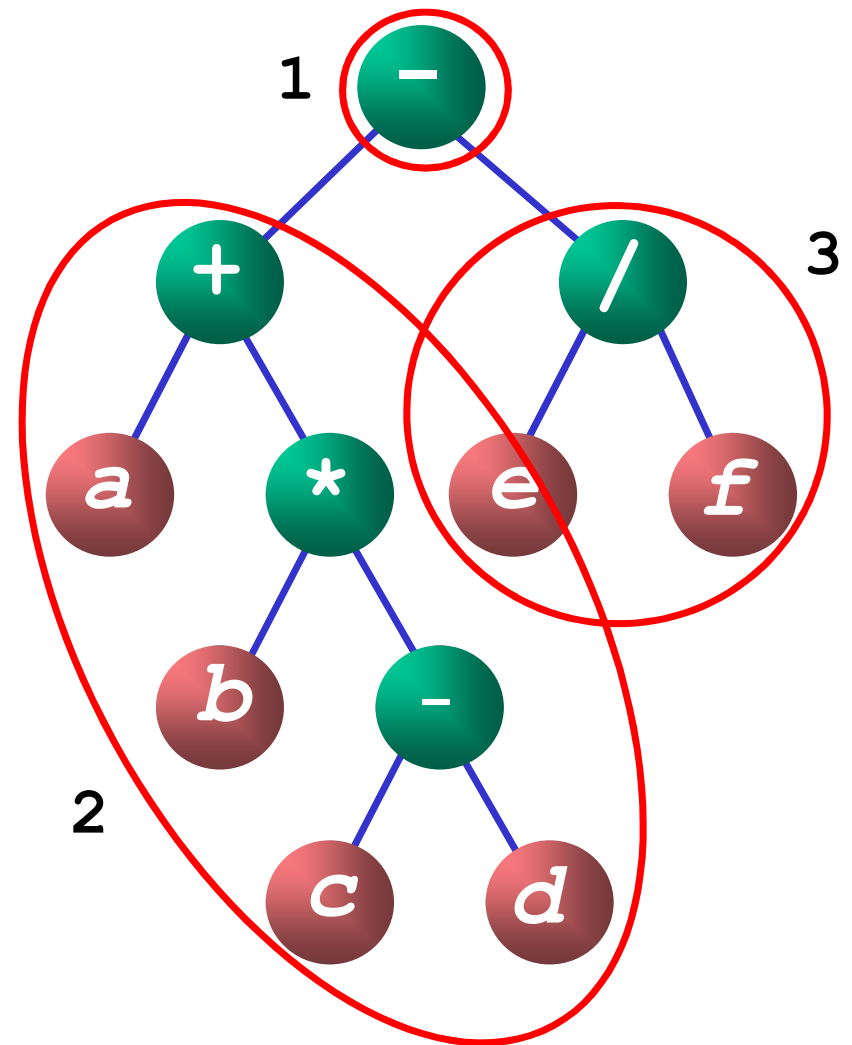
- 查询一个结点的双亲：二叉链表需要遍历整个树( $O(n)$ )，而顺序表示和三叉链表立即得到( $O(1)$ )。

# 二叉树的遍历

- 遍历
  - 按照某种搜索路径访问每个单元，且每个单元仅被访问一次
- 二叉树的遍历
  - 深度优先遍历
    - 先（前）序遍历
    - 中序遍历
    - 后序遍历
  - 广度优先遍历（层序）

## 二叉树的遍历：先序遍历

- 1 先访问树根 “-”
- 2 再访问 “-” 的左子树
- 3 再访问 “-” 的右子树



# 二叉树的遍历：先序遍历

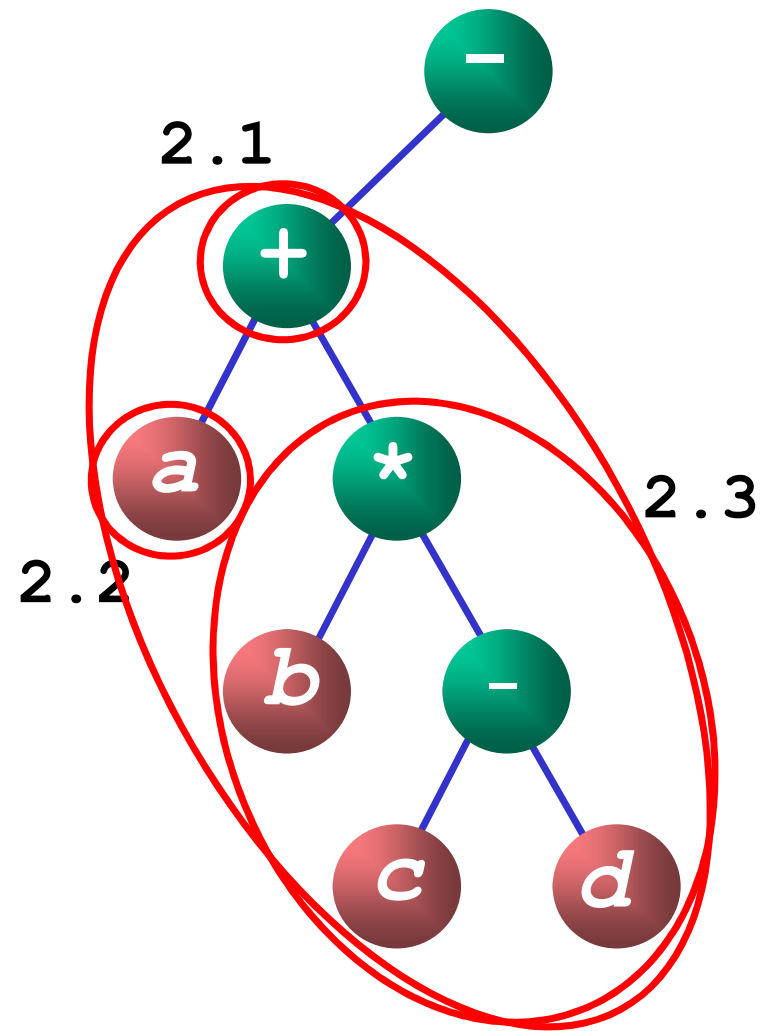
2 访问 “-” 的左子树

对于这棵左子树

2.1 先访问树根 “+”

2.2 再访问 “+” 的左子树

2.3 再访问 “+” 的右子树



## 二叉树的遍历：先序遍历

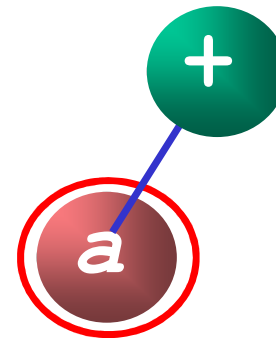
2.2 访问 “+” 的左子树

对于这棵左子树

2.2.1 先访问树根 “a”

2.2.2 再访问 “a” 的左子树（空）

2.2.3 再访问 “a” 的右子树（空）



## 二叉树的遍历：先序遍历

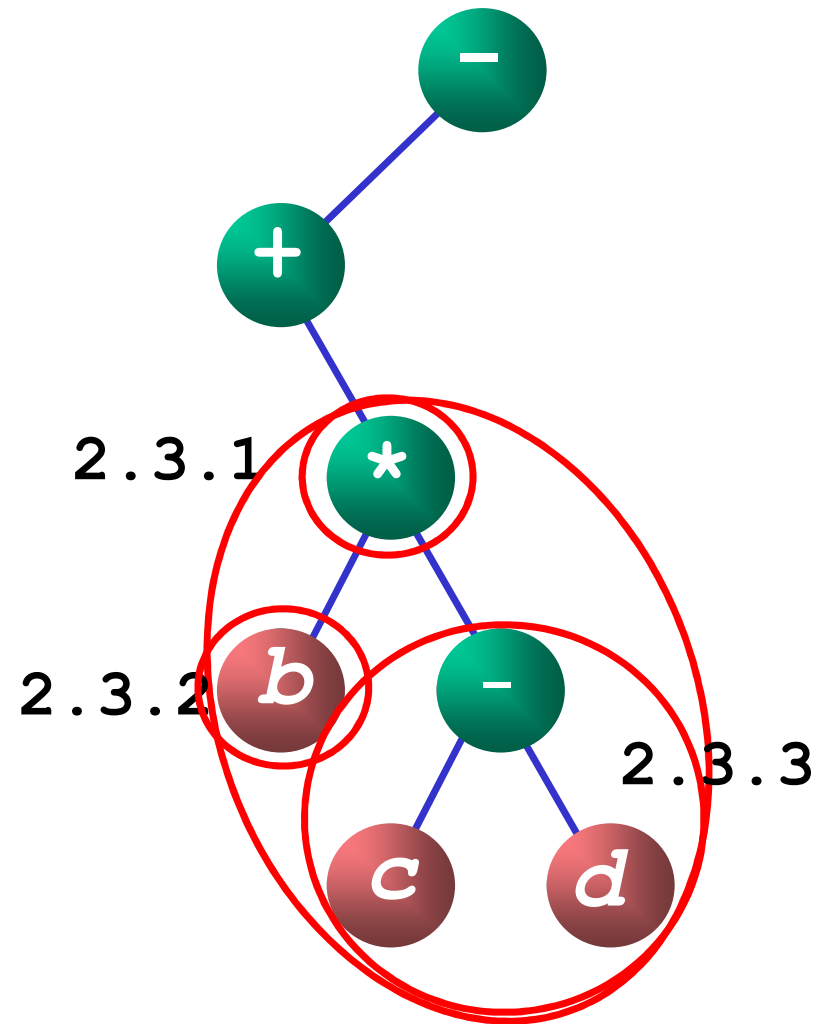
2.3 访问 “+” 的右子树

对于这棵右子树

2.3.1 先访问树根 “\*”

2.3.2 再访问 “\*” 的左子树

2.3.3 再访问 “\*” 的右子树

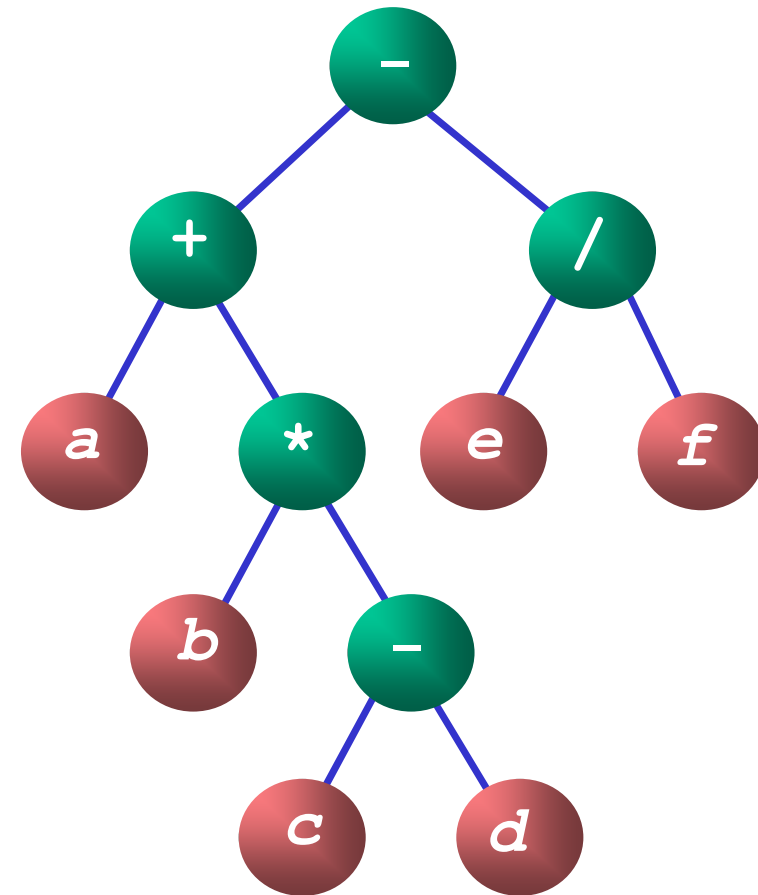




# 二叉树的遍历：先序遍历

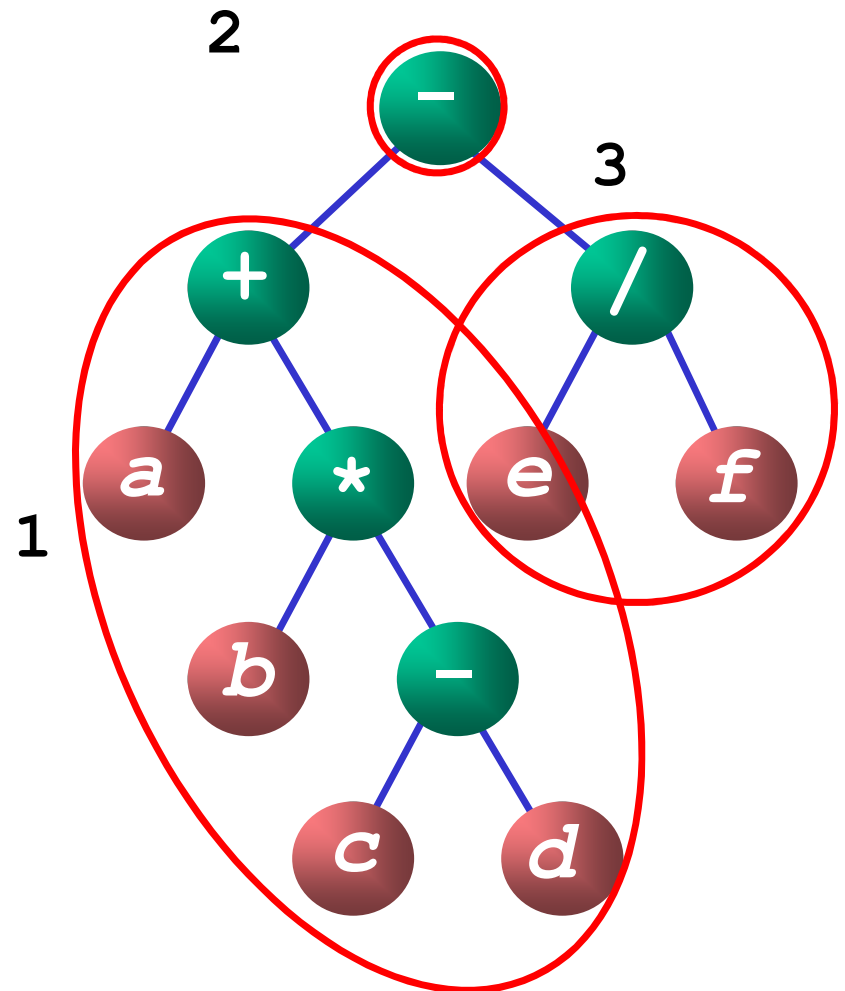
- 理解

- 如果是空树，直接结束
- 如果树非空
  - 先访问树根
  - 把左子树看成跟原来地位相同的另一棵树，用同样的方法去遍历它
  - 左子树遍历完以后再同样的方法去遍历右子树
- 右图的先序遍历结果为：  
**- + a \* b - c d / e f**



## 二叉树的遍历：中序遍历

- 1 先访问 “-” 的左子树
- 2 再访问树根 “-”
- 3 再访问 “-” 的右子树



# 二叉树的遍历：中序遍历

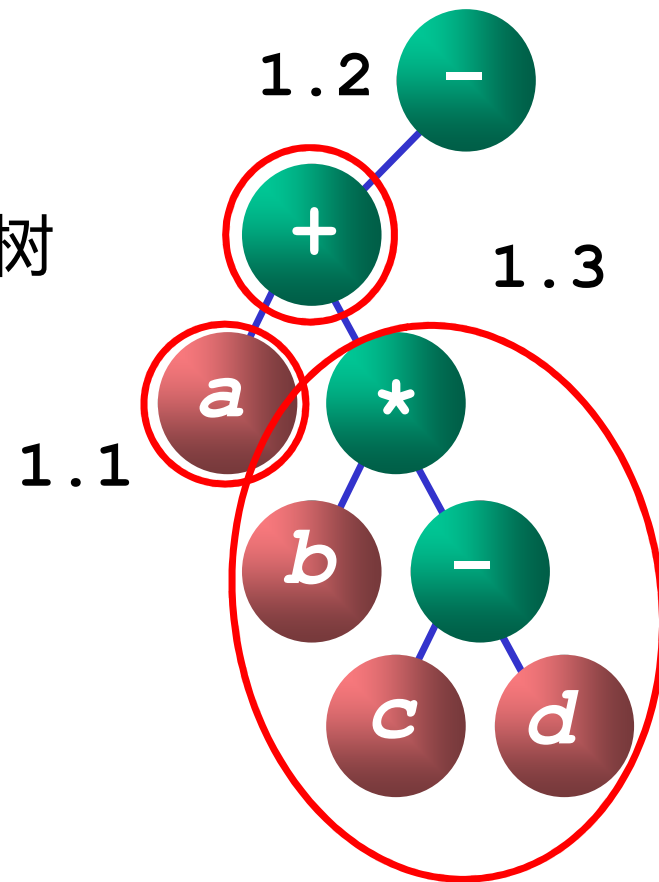
## 1 访问 “-” 的左子树

对于这棵左子树

1.1 先访问子树根 “+” 的左子树

1.2 再访问子树根 “+”

1.3 最后访问 “+” 的右子树



# 二叉树的遍历：中序遍历

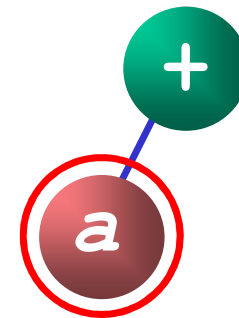
## 1.1 访问 “+” 的左子树

对于这棵左子树

1.1.1 先访问 “a” 的左子树（空）

1.1.2 再访问树根 “a”

1.1.3 最后访问 “a” 的右子树（空）



# 二叉树的遍历：中序遍历

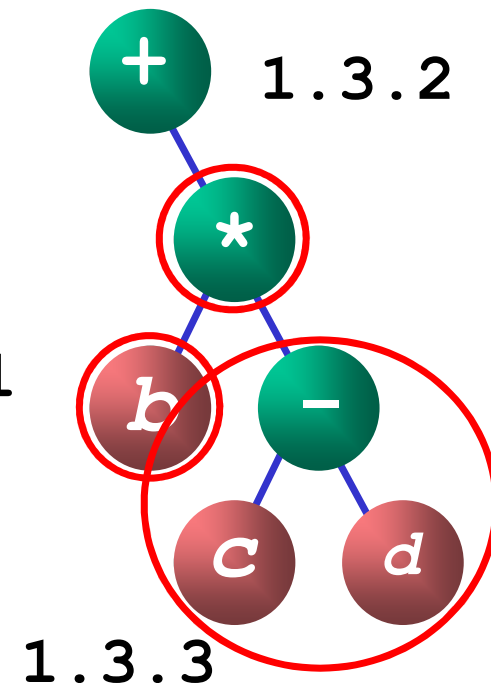
## 1.3 访问 “+” 的右子树

对于这棵右子树

1.3.1 先访问子树根 “\*” 的左子树

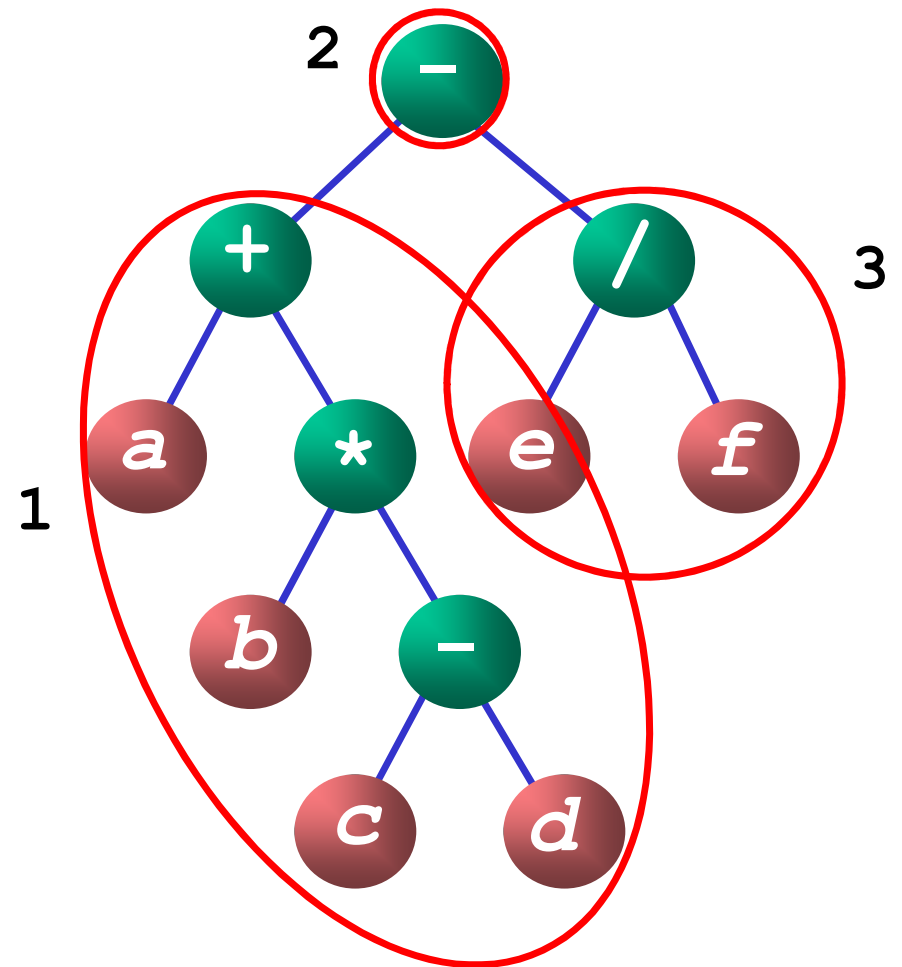
1.3.2 再访问子树根 “\*”

1.3.3 最后访问 “\*” 的右子树



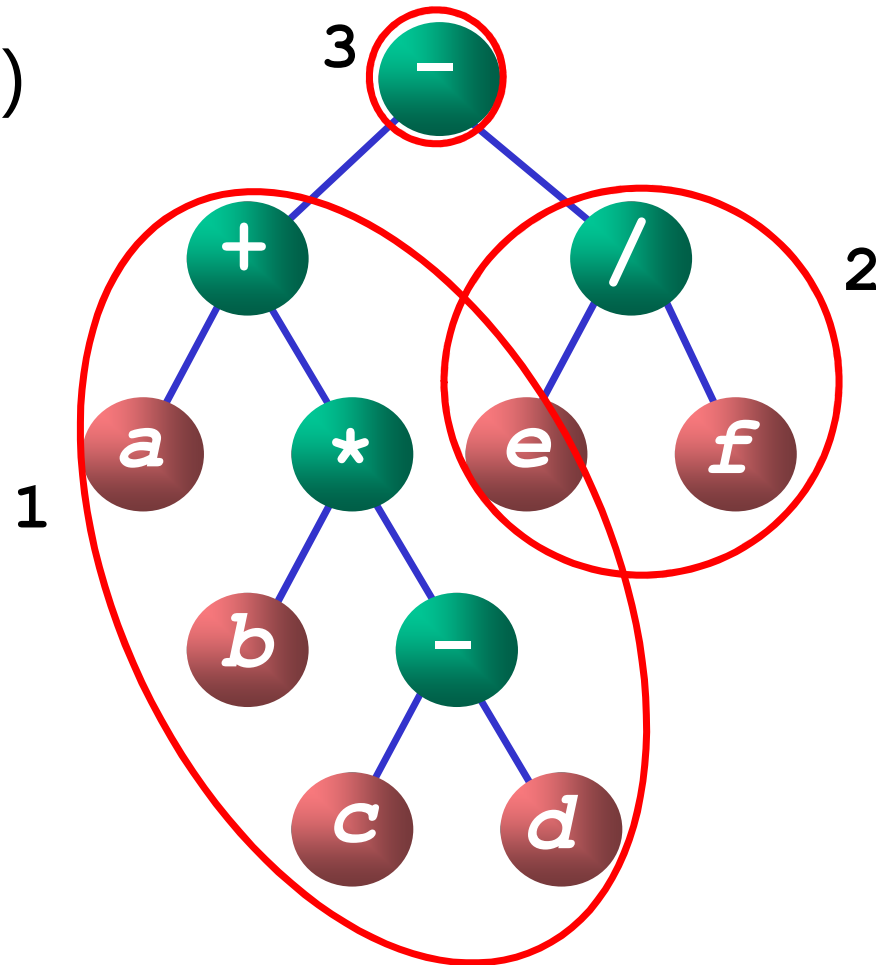
# 二叉树的遍历：中序遍历

- 理解
  - 原理和先序遍历相同
  - 区别在于递归的顺序：
    - 树根在左右两棵子树的中间被访问
- 右图中序遍历结果为：
  - $a + b * c - d - e / f$



## 二叉树的遍历：后序遍历

- 后序遍历 (Postorder Traversal)
  - 若二叉树为空, 则空操作
  - 否则
    - 先后序遍历左子树
    - 再后序遍历右子树
    - 最后访问根结点
  - 右图后序遍历结果为:
    - $a b c d - * + e f / -$

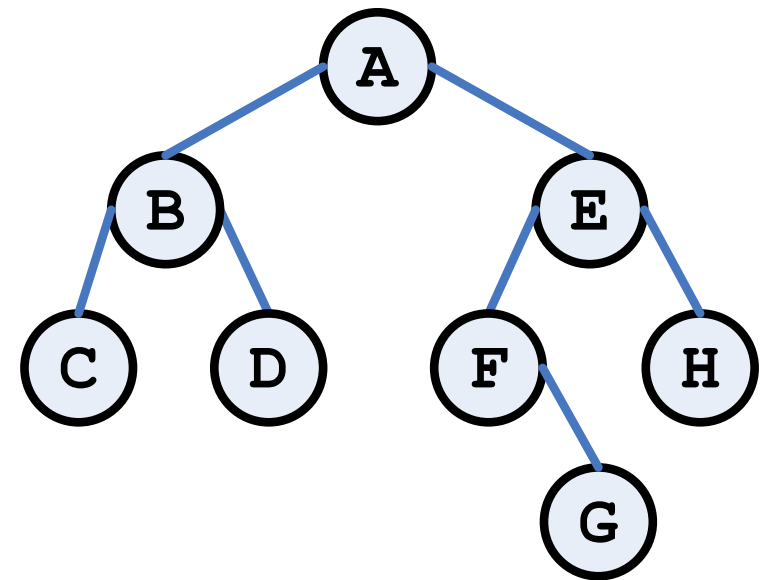


# 二叉树的遍历

- 练习

- 请写出下图先、中、后序遍历的结果

- 先序：ABCDEFGH
- 中序：CBDAFGEH
- 后序：CDBGFHEA





# 二叉树的遍历：递归算法

- 何时用递归？递归算法的适用情况
  - (1)问题本身直接用递归定义的:  $n! = n \cdot (n-1)!$
  - (2)问题的规律有递归的特点: 汉诺塔问题
- 二叉树及其遍历是用递归定义的
  - 用递归算法肯定可以解决
  - 如果不用递归呢？

# 二叉树的遍历：递归算法

- 编写递归程序的要点
  - (1)把部分看成整体
    - 把整体的解决分成若干部分
    - 每个部分的解决方法和整体相同
    - 解决整体时假设部分已经解决
  - (2)注意留递归出口

```
int fact(int n) {  
    if (n > 1)  
        return n*fact(n-1);  
  
    else return 1  
}
```

# 二叉树的遍历：先序遍历

- 先序遍历 (Preorder Traversal)
  - 若二叉树为空，则空操作
  - 否则
    - 先访问根结点
    - 再先序遍历左子树
    - 最后先序遍历右子树

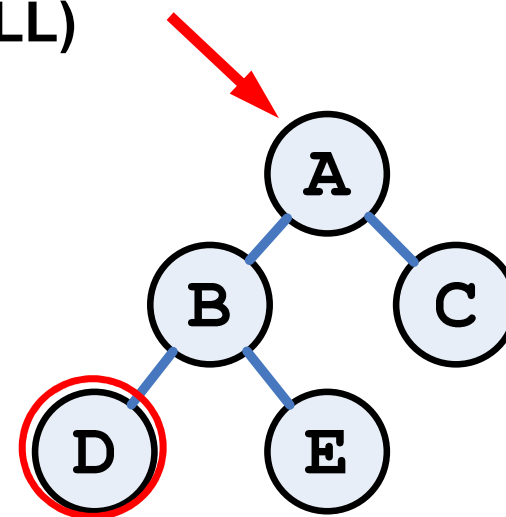
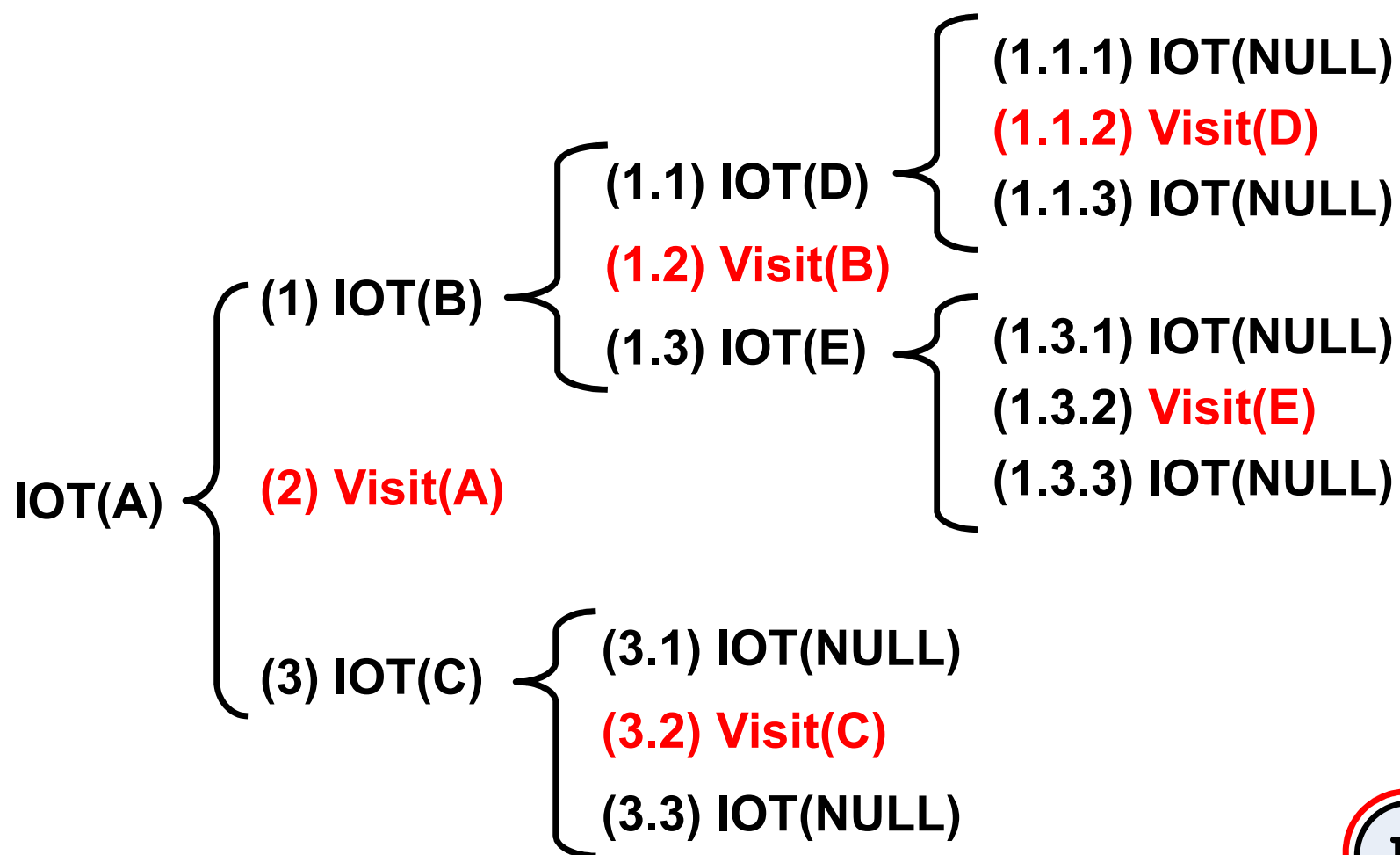
```
PreOT(Tree T) {  
    if(!T) return;  
    visit(T的根);  
    PreOT(T的左子树);  
    PreOT(T的右子树);  
}
```

# 二叉树的遍历：中序遍历

- 中序遍历 (Inorder Traversal)
  - 若二叉树为空，则空操作
  - 否则
    - 先中序遍历左子树
    - 再访问根结点
    - 最后中序遍历右子树

```
IOT (Tree T) {  
    if (!T) return;  
    IOT (T的左子树)  
    visit (T的根);  
    IOT (T的右子树)  
}
```

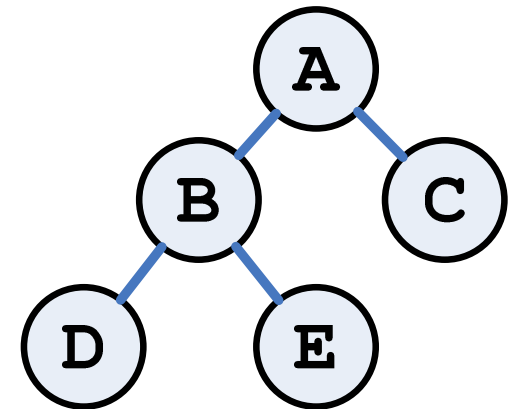
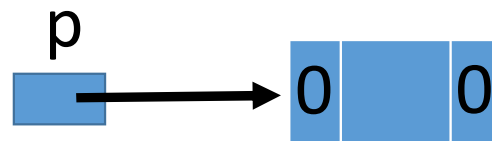
# 二叉树的遍历：中序遍历递归算法



## 练习

- 手工创建如图的二叉树并用遍历算法给出遍历的结果

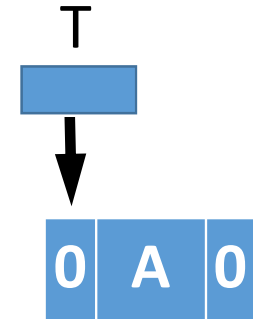
```
BiNode *newNode(){  
    BiNode *p = (BiNode *)malloc(sizeof(BiNode) );  
    if(!p) return 0;  
    p->lchild = p->rchild = 0;  
    return p;  
}
```



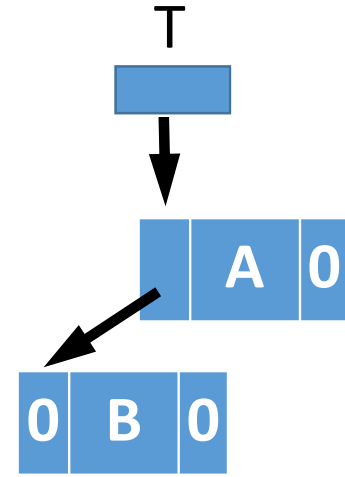
```
int main( ){
```

```
    BiNode *T = newNode();  T->data = 'A';
```

```
}
```



```
int main( ){  
    BiNode *T = newNode();  T->data = 'A';  
    T->lchild= newNode();  T->lchild->data = 'B';  
  
}
```





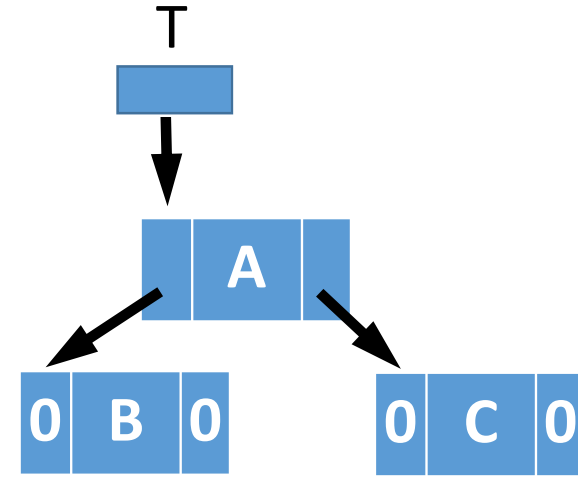
```
int main( ){
```

```
    BiNode *T = newNode();  T->data = 'A';
```

```
    T->lchild= newNode();  T->lchild->data = 'B';
```

```
    T->rchild= newNode();  T->rchild->data = 'C';
```

```
}
```



```
int main( ){
```

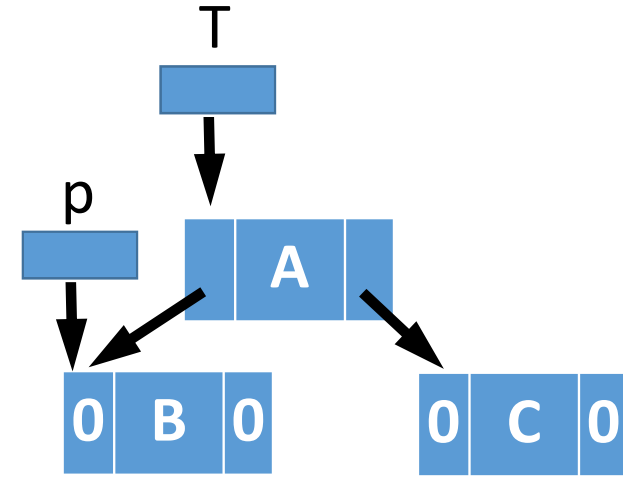
```
    BiNode *T = newNode();  T->data = 'A';
```

```
    T->lchild= newNode();  T->lchild->data = 'B';
```

```
    T->rchild= newNode();  T->rchild->data = 'C';
```

```
    BiNode * p = T->lchild;
```

```
}
```



```
int main( ){
```

```
    BiNode *T = newNode();  T->data = 'A';
```

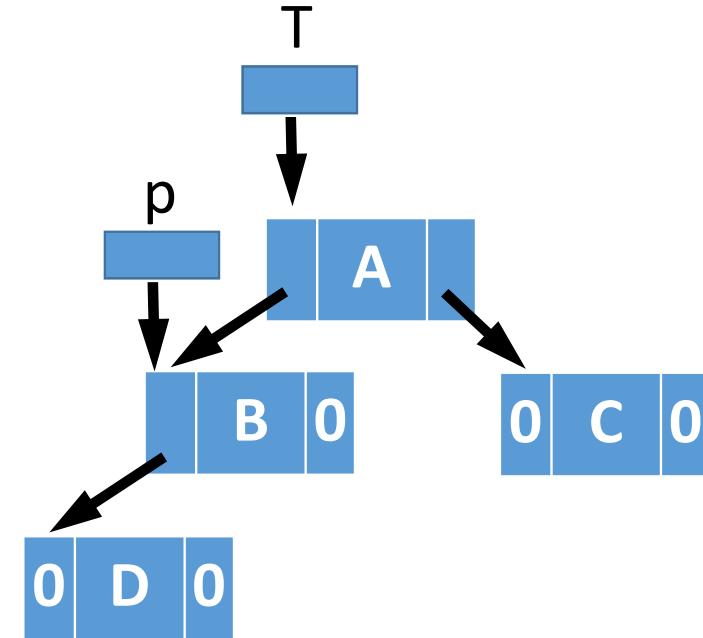
```
    T->lchild= newNode();  T->lchild->data = 'B';
```

```
    T->rchild= newNode();  T->rchild->data = 'C';
```

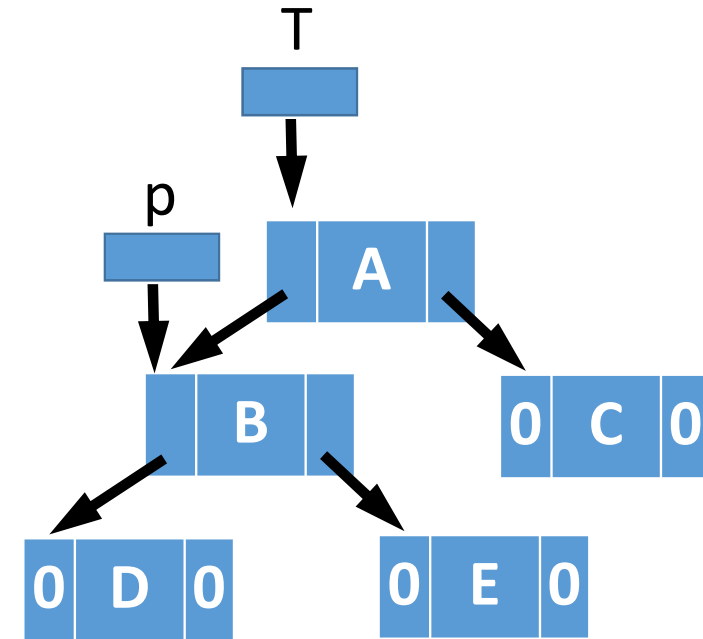
```
    p = T->lchild;
```

```
    p->lchild= newNode();  p->lchild->data = 'D';
```

```
}
```



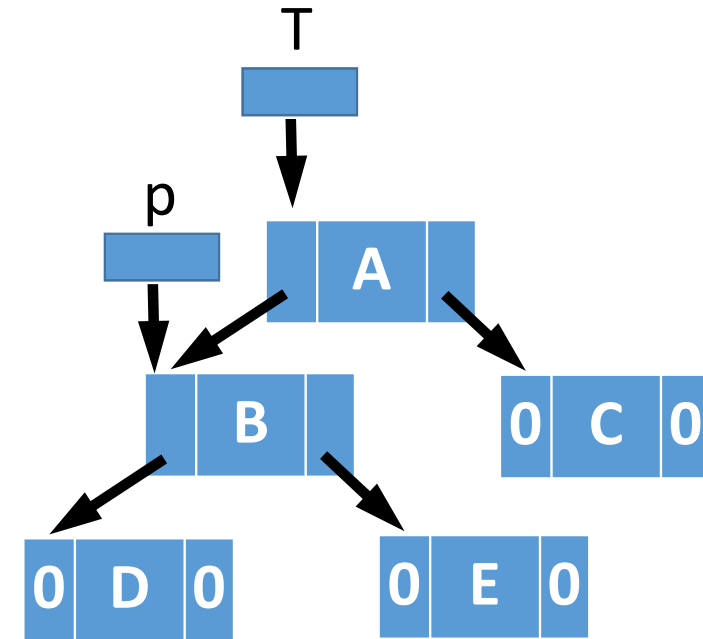
```
int main( ){  
    BiNode *T = newNode();  T->data = 'A';  
    T->lchild= newNode();  T->lchild->data = 'B';  
    T->rchild= newNode();  T->rchild->data = 'C';  
    p = T->lchild;  
    p->lchild= newNode();  p->lchild->data = 'D';  
    p->rchild= newNode();  p->rchild->data = 'E';  
  
}
```



```

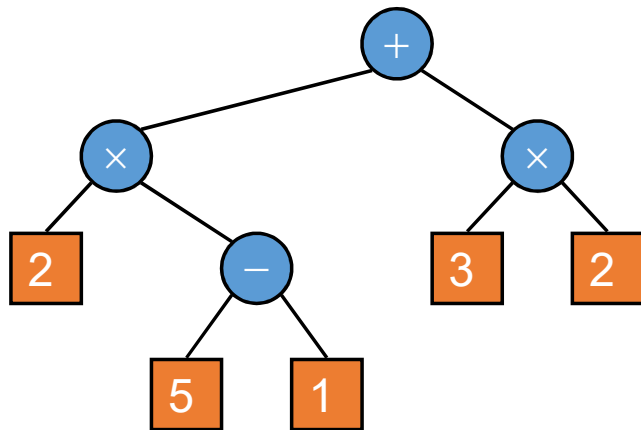
int main( ){
    BiNode *T = newNode();  T->data = 'A';
    T->lchild= newNode();  T->lchild->data = 'B';
    T->rchild= newNode();  T->rchild->data = 'C';
    p = T->lchild;
    p->lchild= newNode();  p->lchild->data = 'D';
    p->rchild= newNode();  p->rchild->data = 'E';
    IOT(T); /*输出中序遍历序列*/
    PreOT(T); /*输出先序遍历序列*/
    PostOT(T); /*输出先序遍历序列*/
}

```



# 打印算术表达式

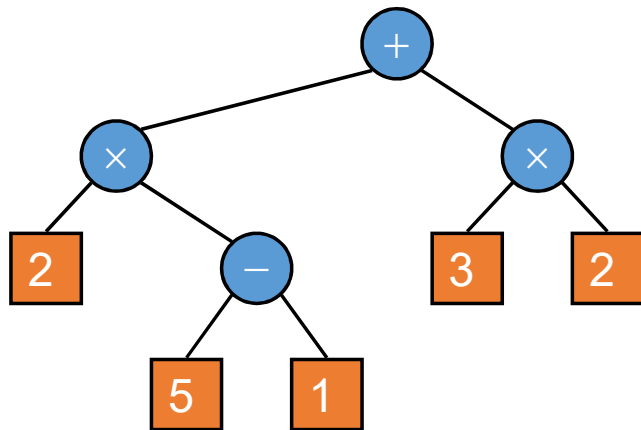
- 访问结点时打印运算符或运算数
- 访问左子树前打印 (
- 访问右子树后打印 )



$$((2 \times (5 - 1)) + (3 \times 2))$$

# 打印算术表达式

- 访问结点时打印运算符或运算数
- 访问左子树前打印 (
- 访问右子树后打印 )



$((2 \times (5 - 1)) + (3 \times 2))$

```
void printExpr(p)
    if(!p) return;
    if (isLeaf(p))
        print(p->data);
    else{
        printf("("); printExpr(p->lchild);
        print(p->data);
        printExpr(p->rchild);printf(")");
    }
}
```

# 二叉树的遍历：非递归算法

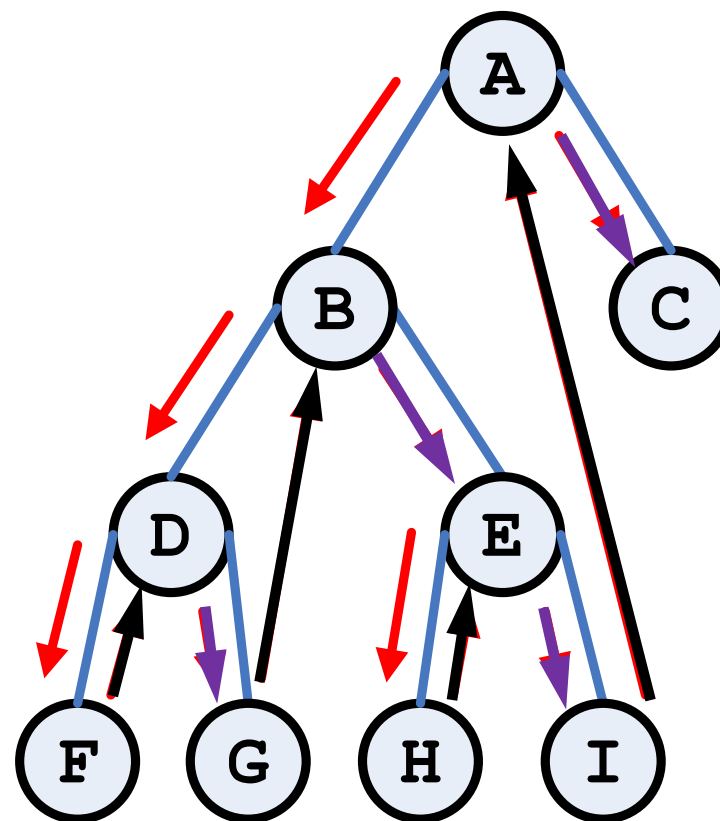
- 为什么要将递归算法转化为非递归算法?
  - 1) 程序堆栈的大小是有限制的，过深的递归可能会溢出程序堆栈
  - 2) 系统维护函数调用需要一定的开销，耗费一定时间



# 二叉树的遍历：非递归算法

- 回顾遍历的过程（以中序为例）

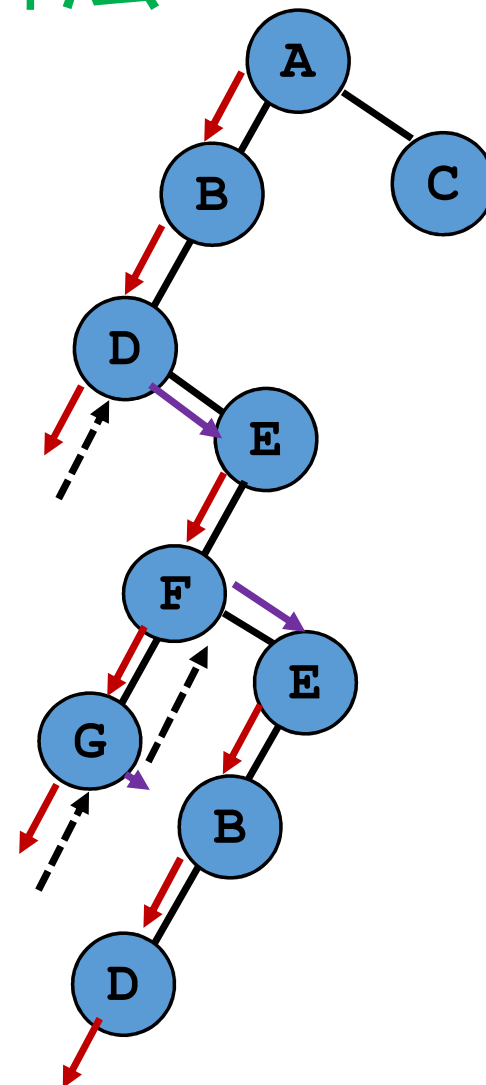
- 1 先走到最左
- 2 往回访问父结点
- 3 往右访问右子树
  - 3.1 走到最左
  - 3.2 回访父结点
  - 3.3 访问右子树
- ...



# 二叉树的遍历：非递归算法

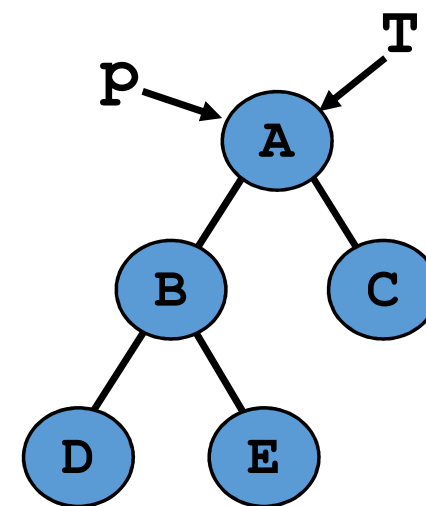
- 回顾遍历的过程（以中序为例）

- 1 先走到最左
- 2 往回访问父结点
- 3 往右访问右子树
  - 3.1 走到最左
  - 3.2 回访父结点
  - 3.3 访问右子树
- ...



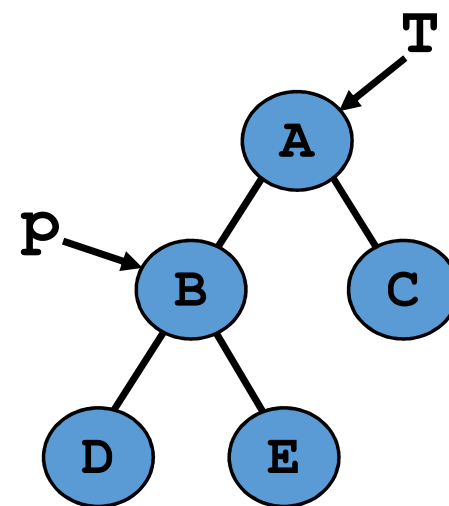
算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```



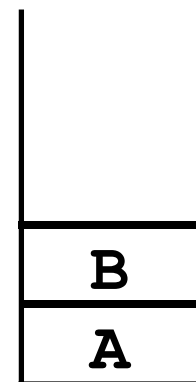
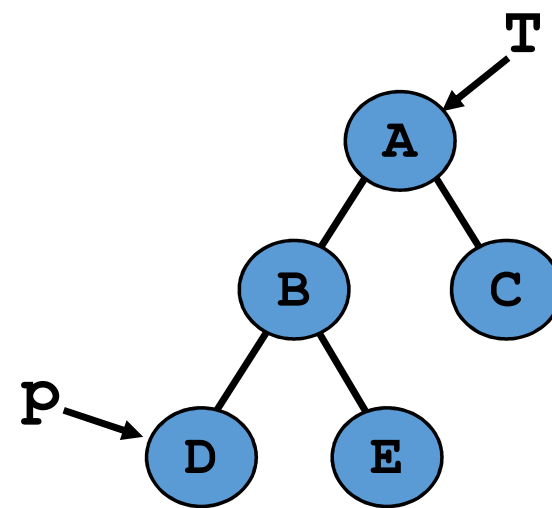
算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```



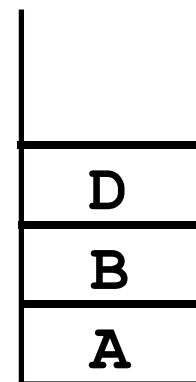
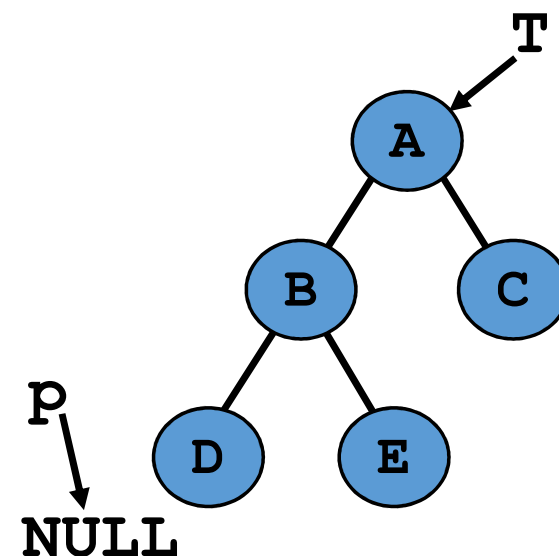
算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```



算法1

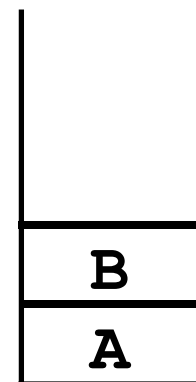
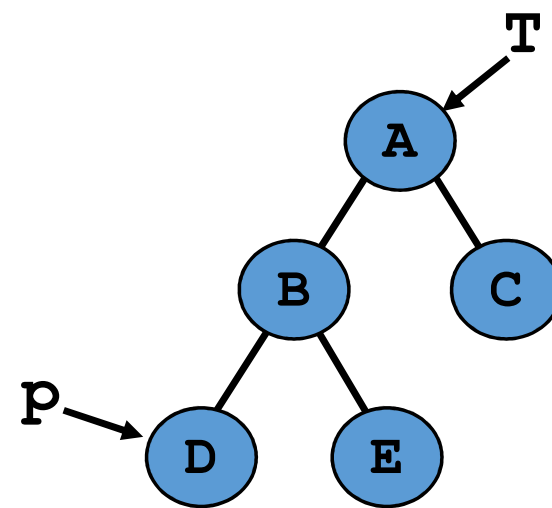
```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

D



# 算法1

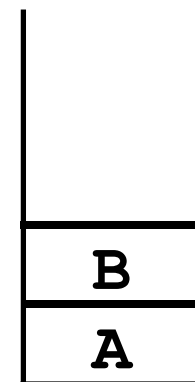
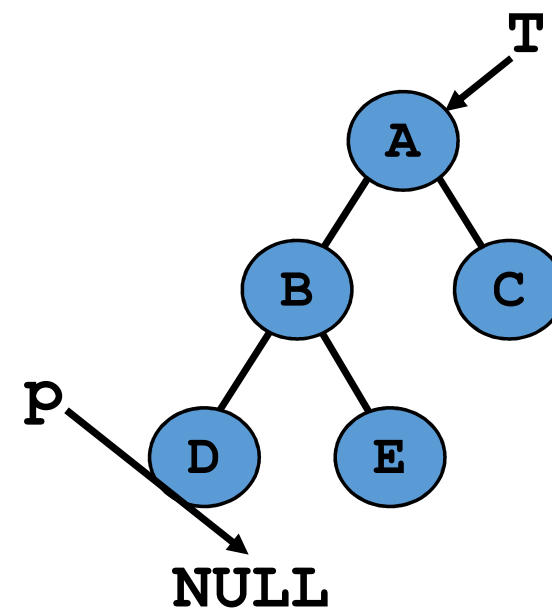
```

bool InOrderTraverse (BiNode* T) {
    InitStack(S); BiNode *p = T;
    do{
        while(p){ //向左走到头
            Push(S, p); p = p->lchild;
        }
        Pop(S, p); //栈顶的左子树访问完
        Visit(p->data) //栈顶出栈并访问之

        p = p->rchild; //再向右走
    }while(!StackEmpty(S));
    return true;
}

```

D

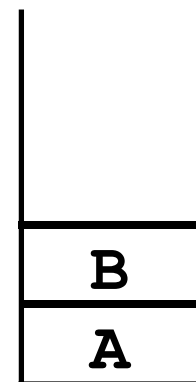
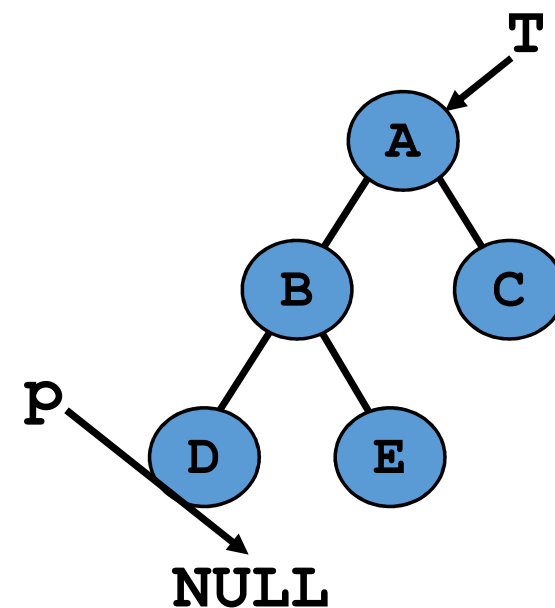




算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

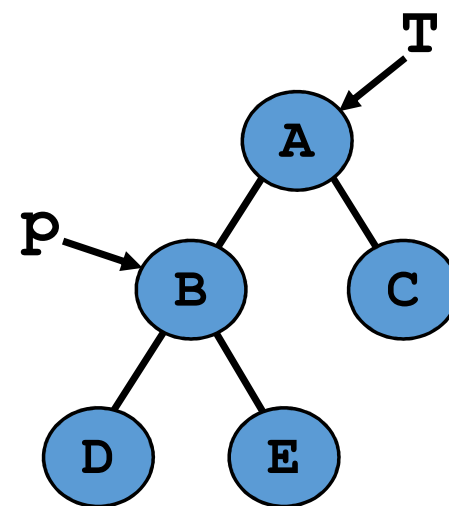
D



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

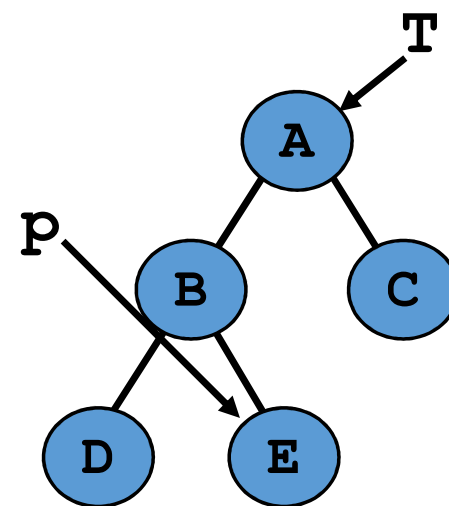
D B



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

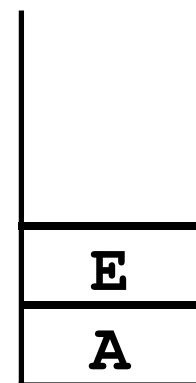
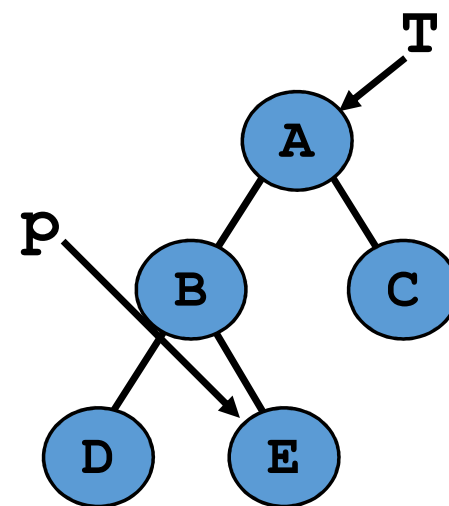
D B



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

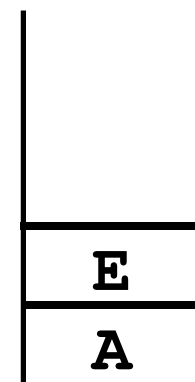
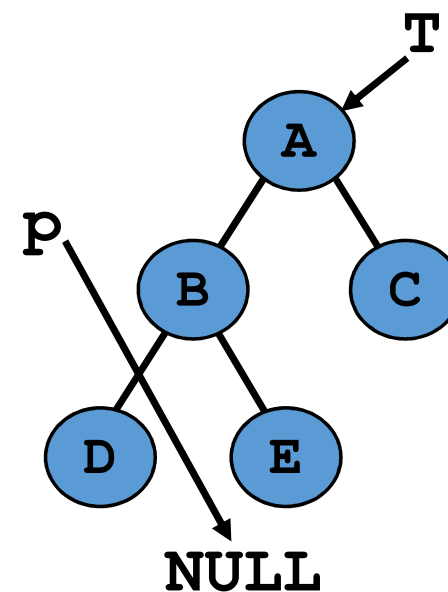
D B



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

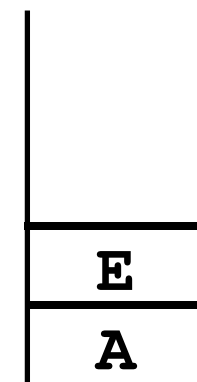
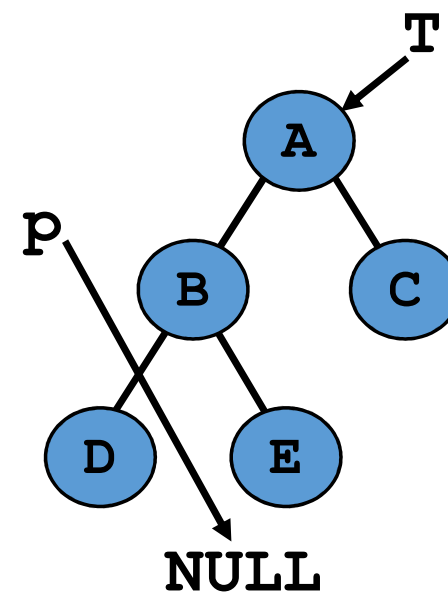
D B



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

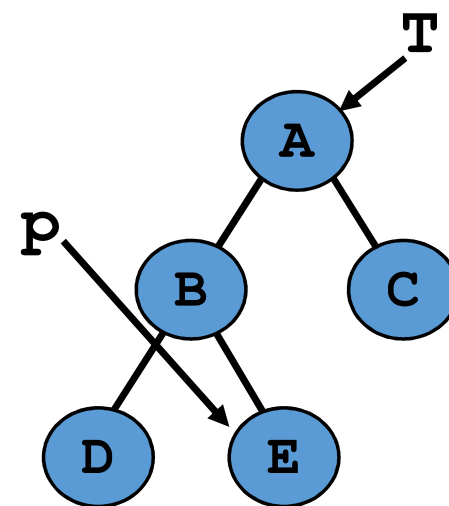
D B



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

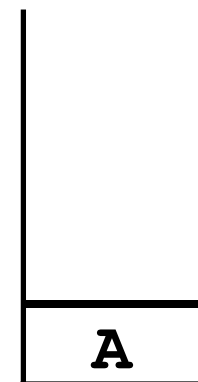
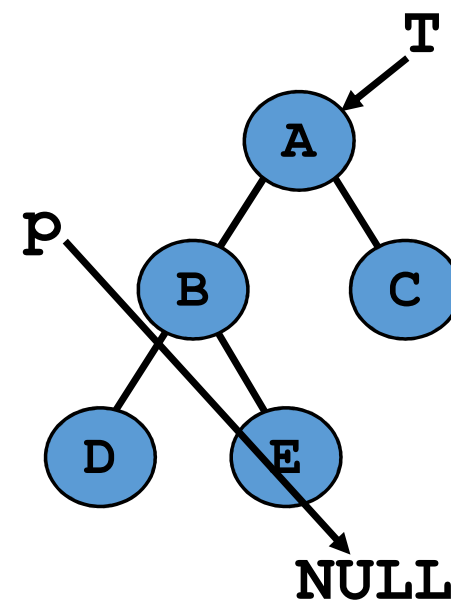
D B E



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

D B E

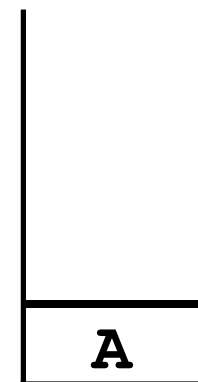
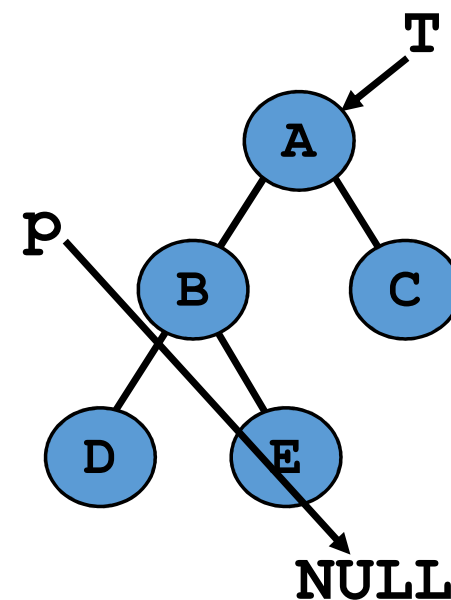




算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

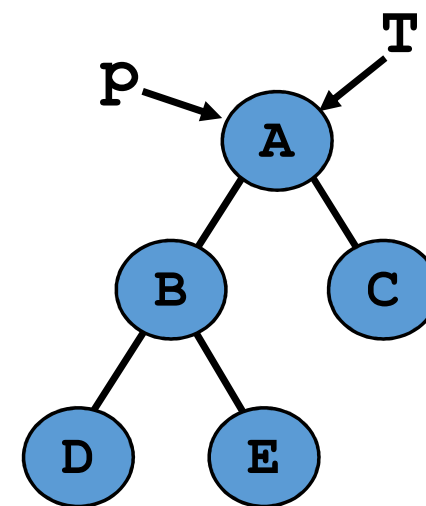
D B E



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

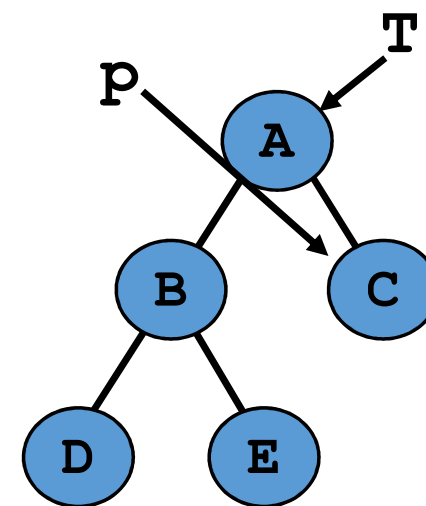
D B E A



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

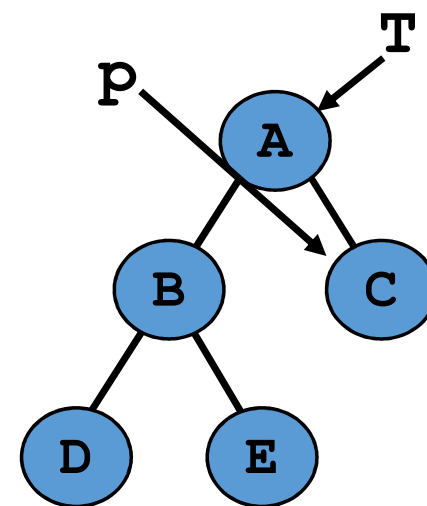
D B E A



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

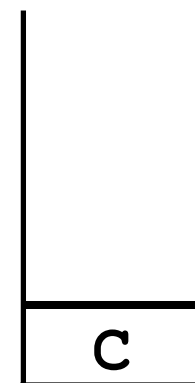
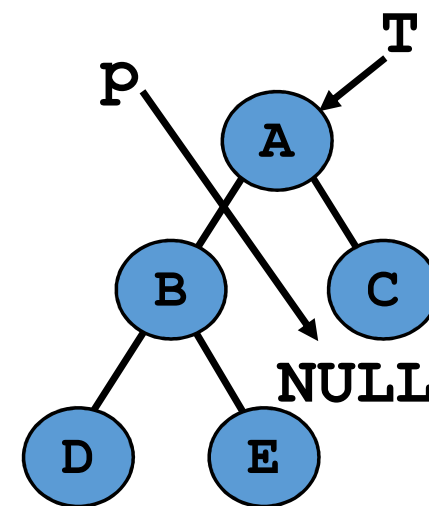
D B E A



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

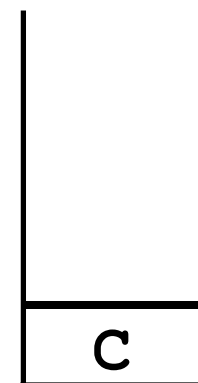
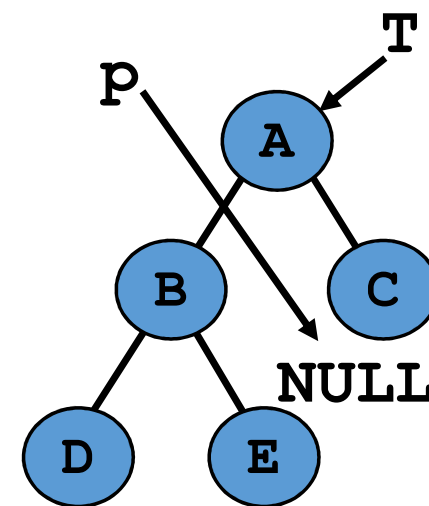
D B E A



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

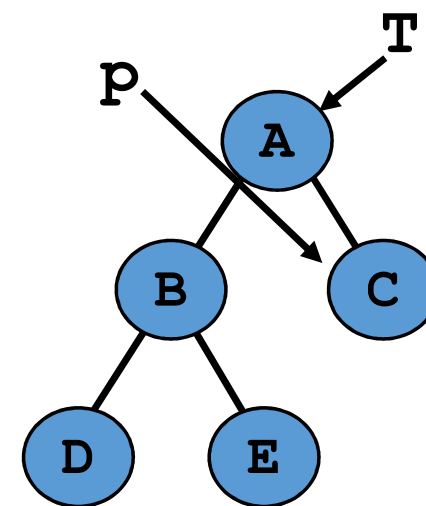
D B E A



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

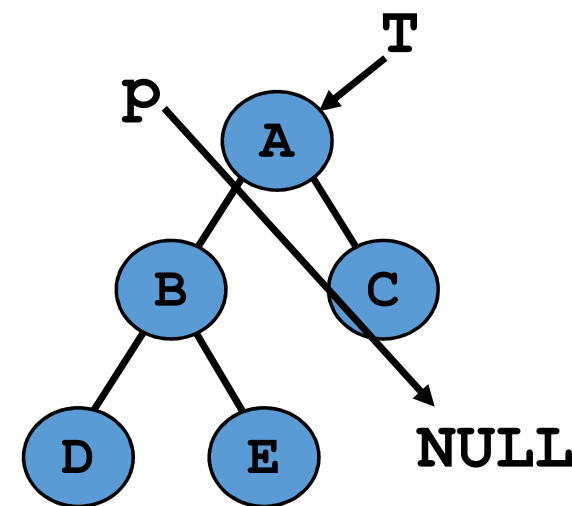
D B E A C



算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S));  
    return true;  
}
```

D B E A C

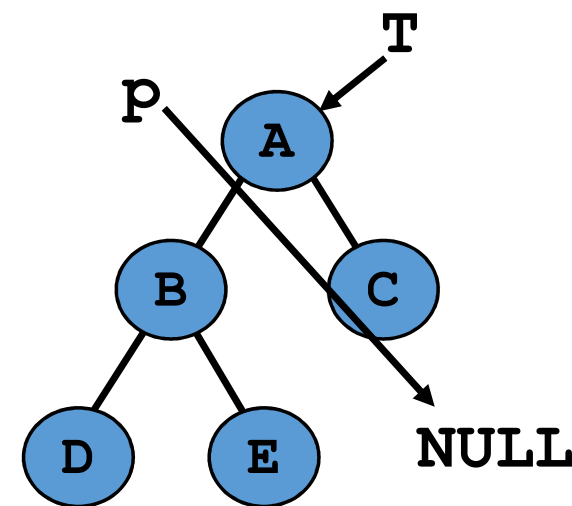




算法1

```
bool InOrderTraverse (BiNode* T) {  
    InitStack(S); BiNode *p = T;  
    do{  
        while(p){ //向左走到头  
            Push(S, p); p = p->lchild;  
        }  
        Pop(S, p); //栈顶的左子树访问完  
        Visit(p->data) //栈顶出栈并访问之  
  
        p = p->rchild; //再向右走  
    }while(!StackEmpty(S)) ;  
    return true;  
}
```

D B E A C



```

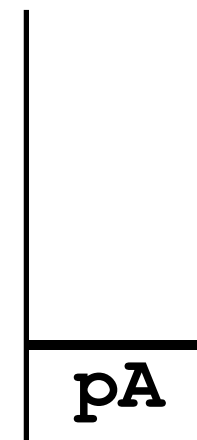
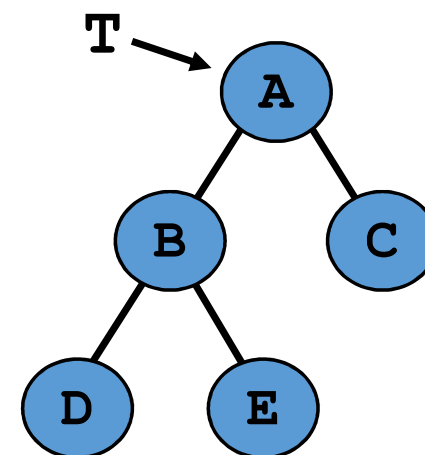
算法2 int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while (!StackEmpty(S)) {           //只要栈非空
        while (GetTop(S, p) && p)       //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                      //弹出多入栈的空结点
        if (!StackEmpty(S)) {          //如果栈非空
            Pop(S, p);                  //弹出一个元素
            if (!Visit(p->data))         //访问之
                return -1;
            Push(S, p->rchild);          //再向右走
        }
    }
    return 0;
}

```

```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while (!StackEmpty(S)) {             //只要栈非空
        while (GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if (!StackEmpty(S)) {            //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if (!Visit(p->data))         //访问之
                return -1;
            Push(S, p->rchild);          //再向右走
        }
    }
    return 0;
}

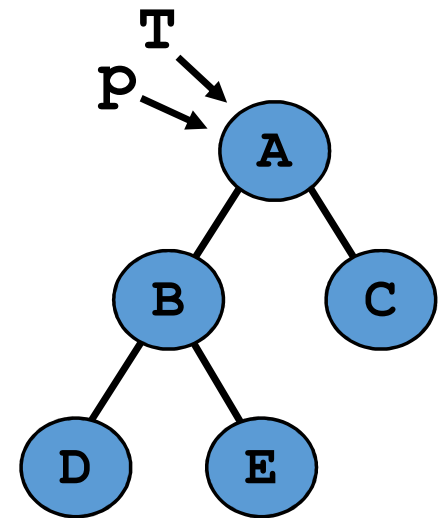
```



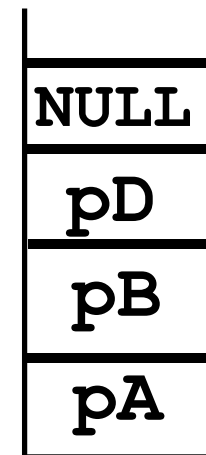
```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while (!StackEmpty(S)) {           //只要栈非空
        while (GetTop(S, p) && p)      //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                     //弹出多入栈的空结点
        if (!StackEmpty(S)) {         //如果栈非空
            Pop(S, p);                 //弹出一个元素
            if (!Visit(p->data))       //访问之
                return -1;
            Push(S, p->rchild);        //再向右走
        }
    }
    return 0;
}

```



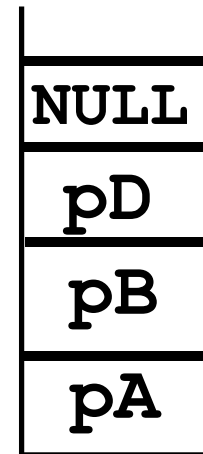
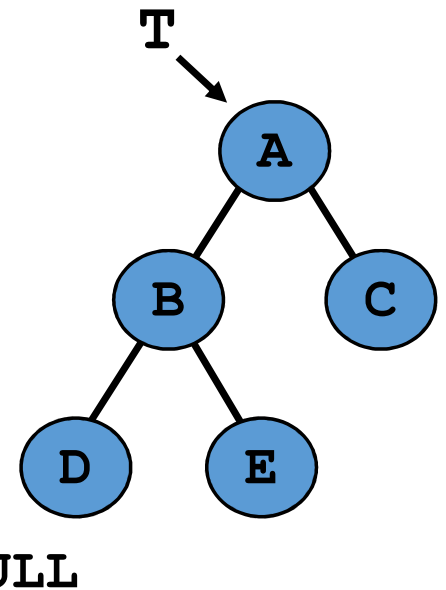
NULL



```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {             //只要栈非空
        while(GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点 p
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                     //弹出一个元素
            if(!Visit(p->data))             //访问之
                return -1;
            Push(S, p->rchild);             //再向右走
        }
    }
    return 0;
}

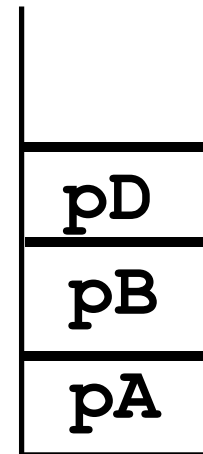
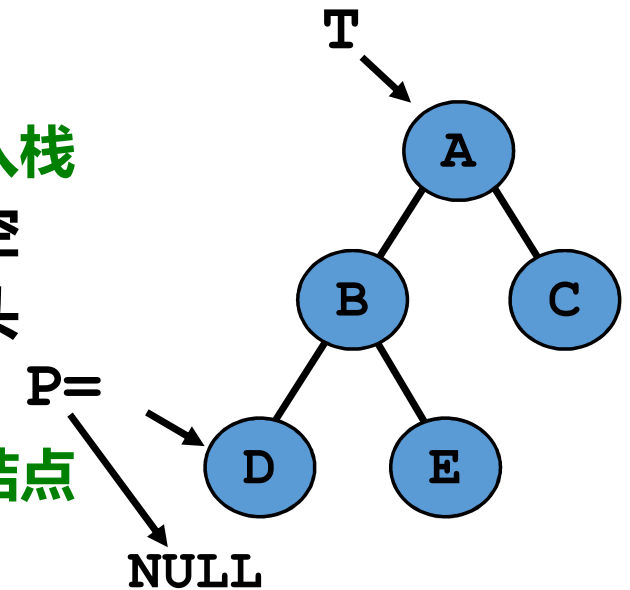
```



```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {             //只要栈非空
        while(GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {            //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);          //再向右走
        }
    }
    return 0;
}

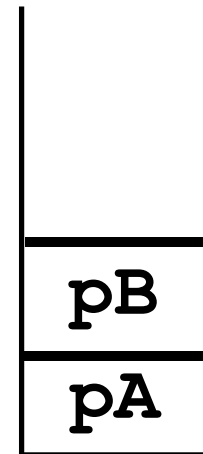
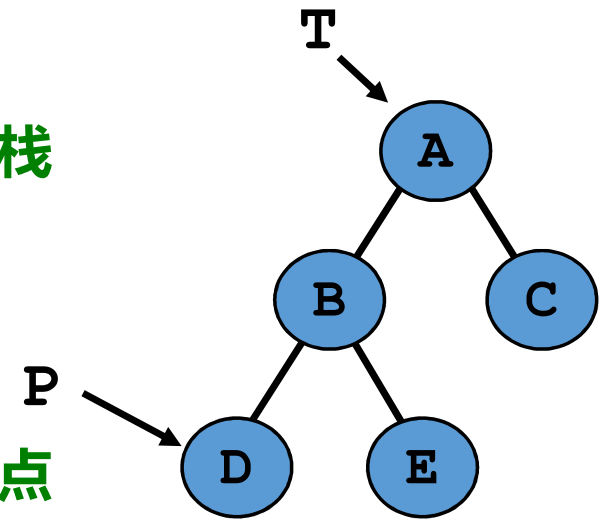
```



```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {             //只要栈非空
        while(GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {            //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

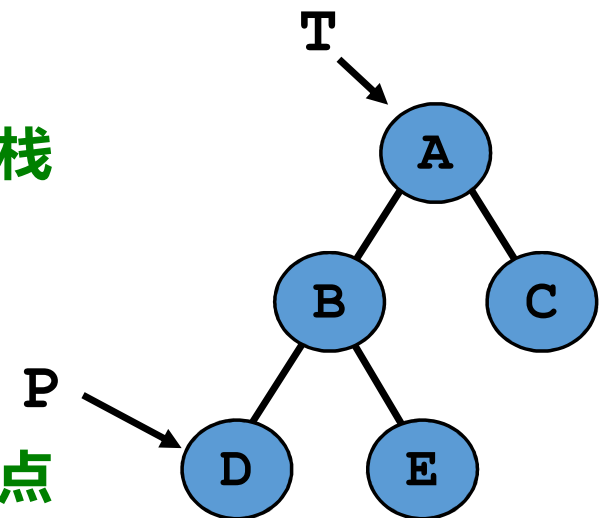
```



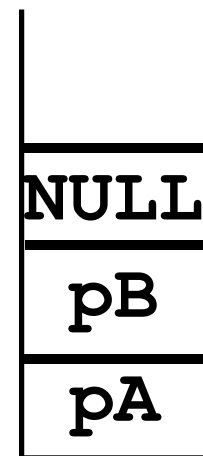
```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```



NULL



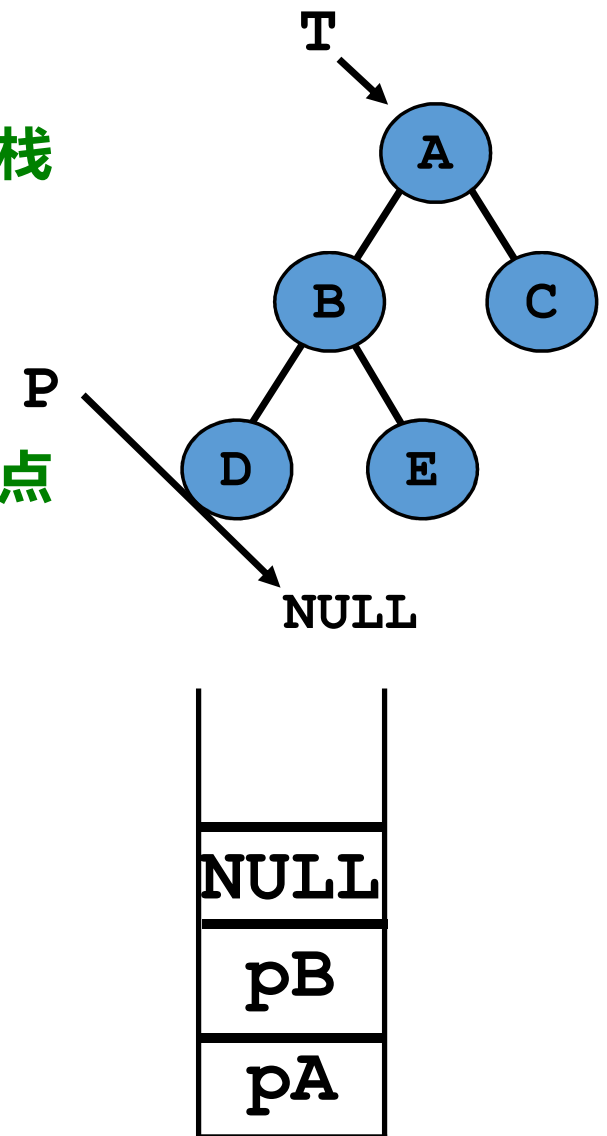
D



```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while (!StackEmpty(S)) {             //只要栈非空
        while (GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if (!StackEmpty(S)) {            //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if (!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

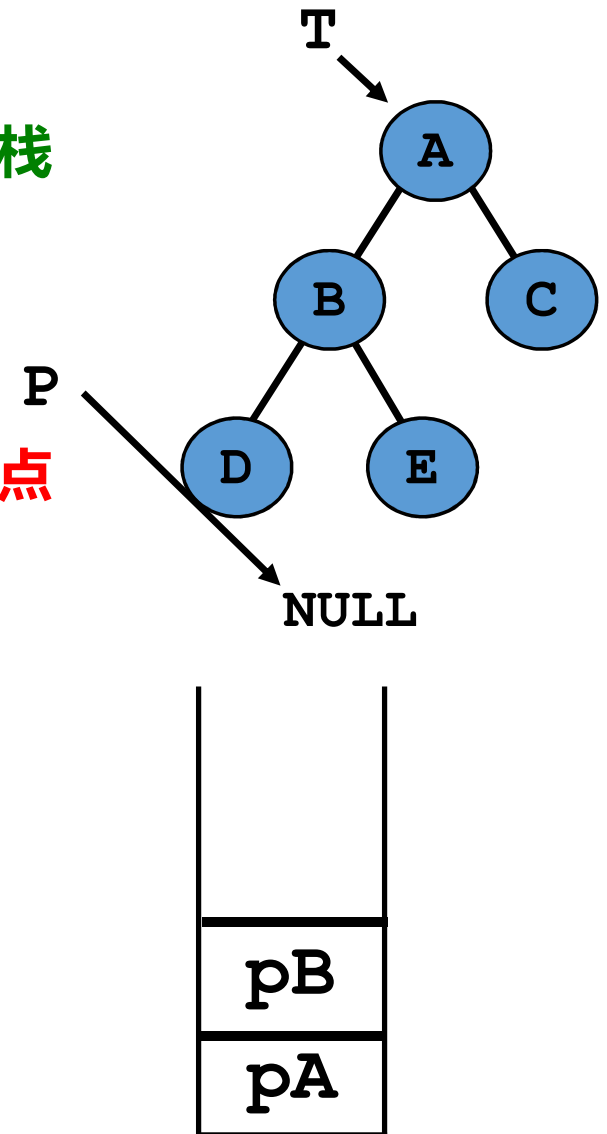
```



```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

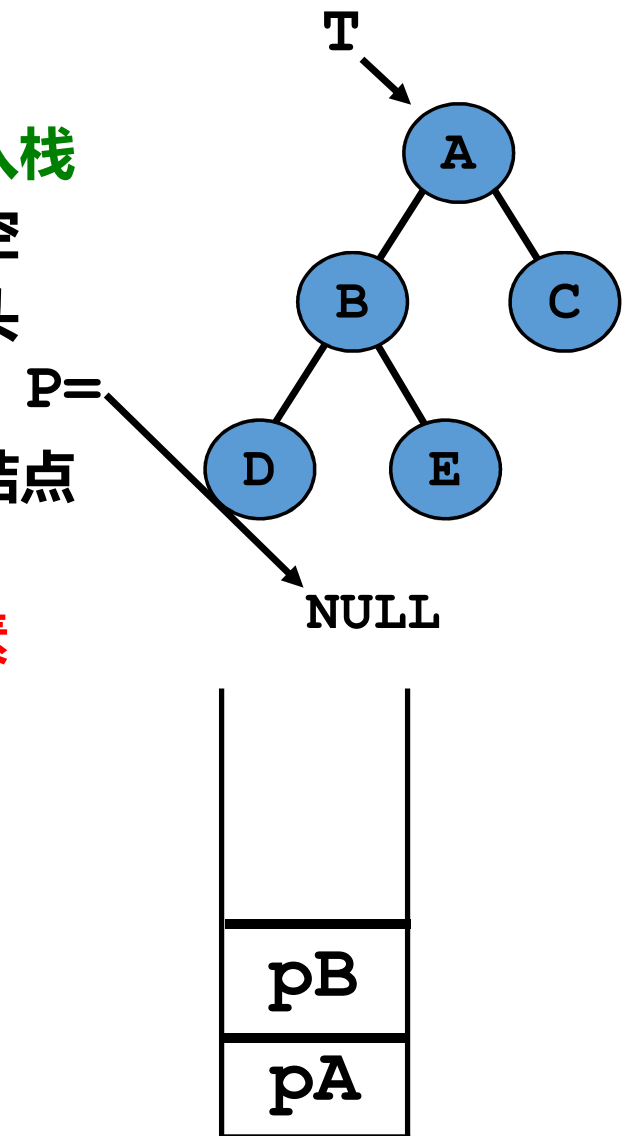
```



```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

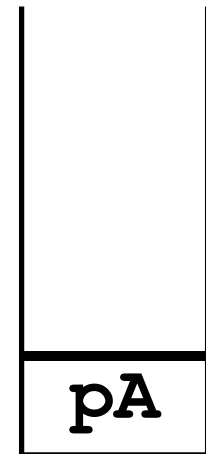
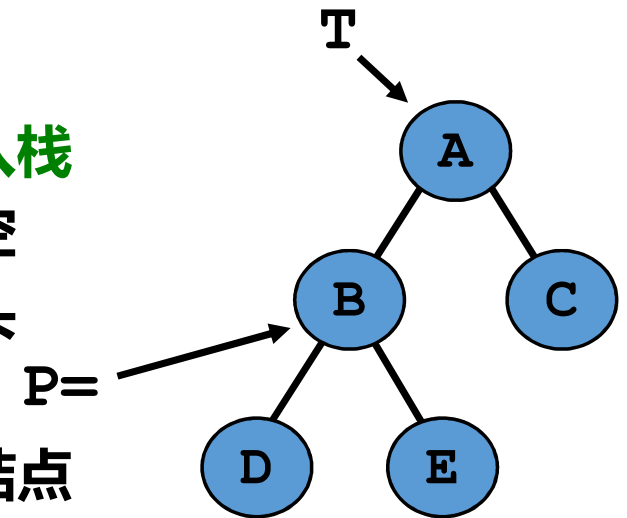


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {             //只要栈非空
        while(GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                      //弹出多入栈的空结点
        if(!StackEmpty(S)) {           //如果栈非空
            Pop(S, p);                  //弹出一个元素
            if(!Visit(p->data))         //访问之
                return -1;
            Push(S, p->rchild);         //再向右走
        }
    }
    return 0;
}

```

D   B

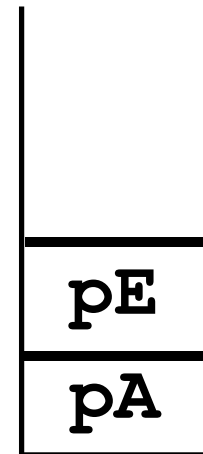
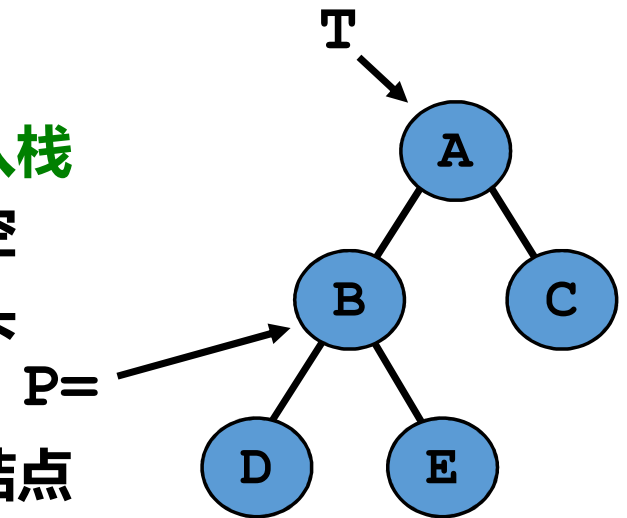


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D   B

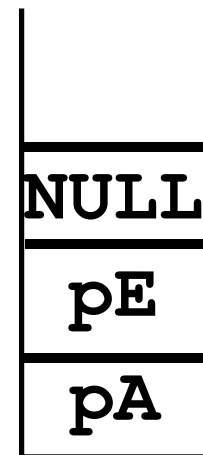
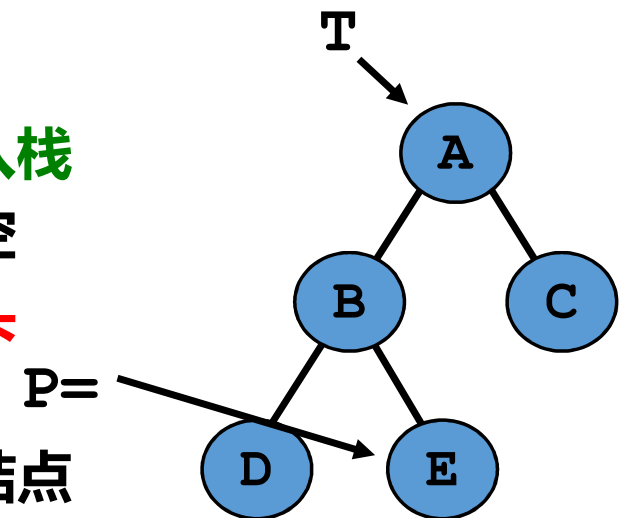


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while (!StackEmpty(S)) {             //只要栈非空
        while (GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if (!StackEmpty(S)) {            //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if (!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D   B

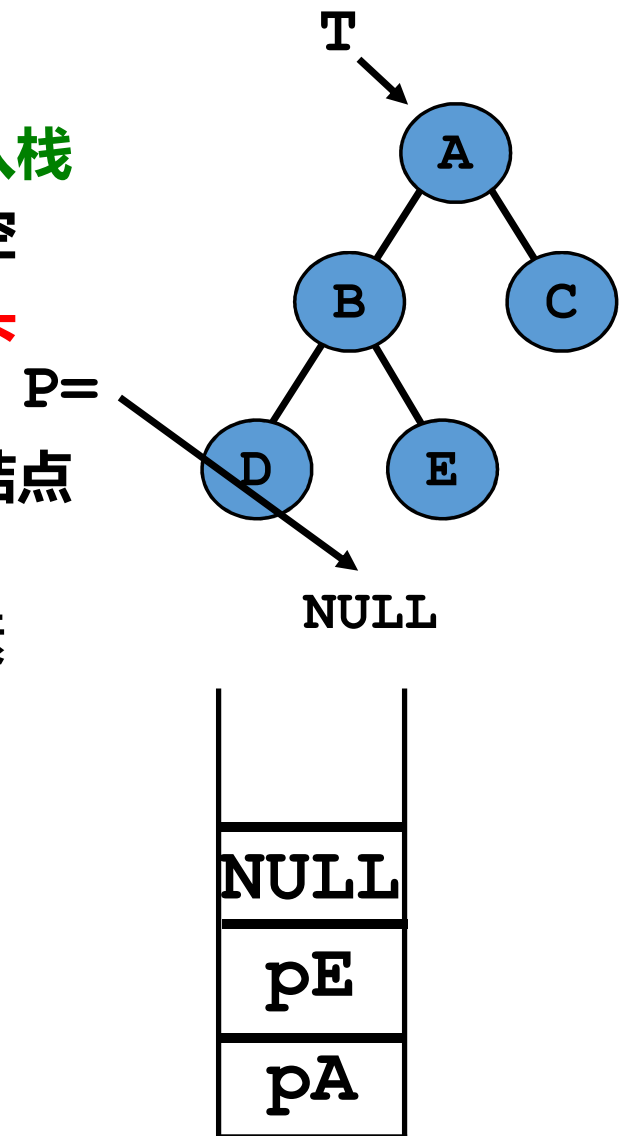


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while (!StackEmpty(S)) {             //只要栈非空
        while (GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if (!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if (!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D B

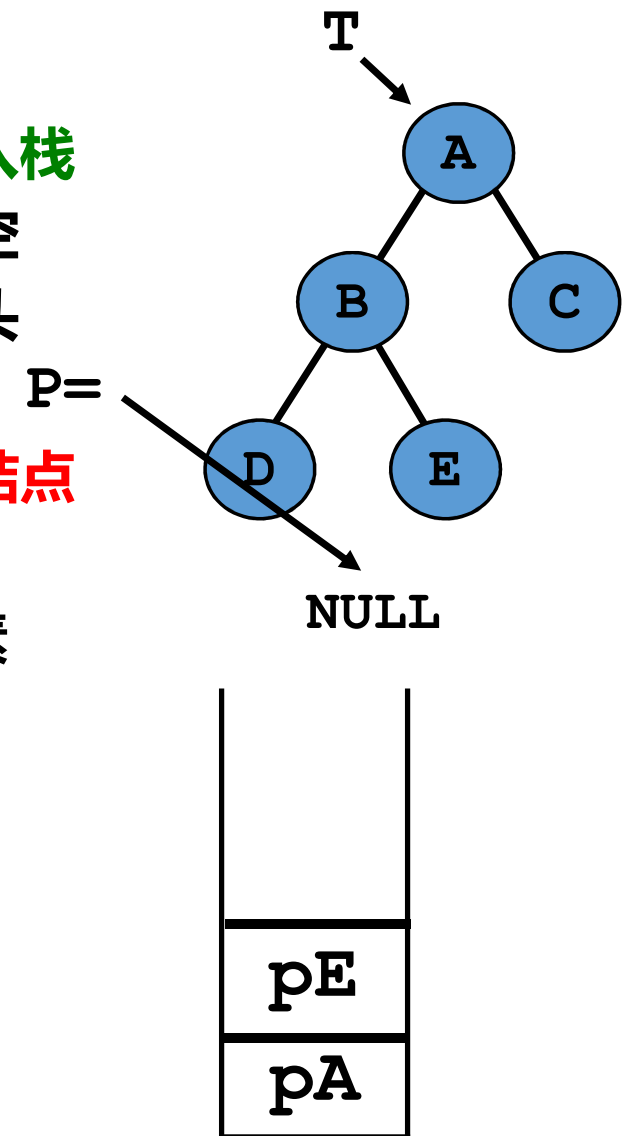


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {             //只要栈非空
        while(GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                      //弹出多入栈的空结点
        if(!StackEmpty(S)) {            //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))           //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D   B



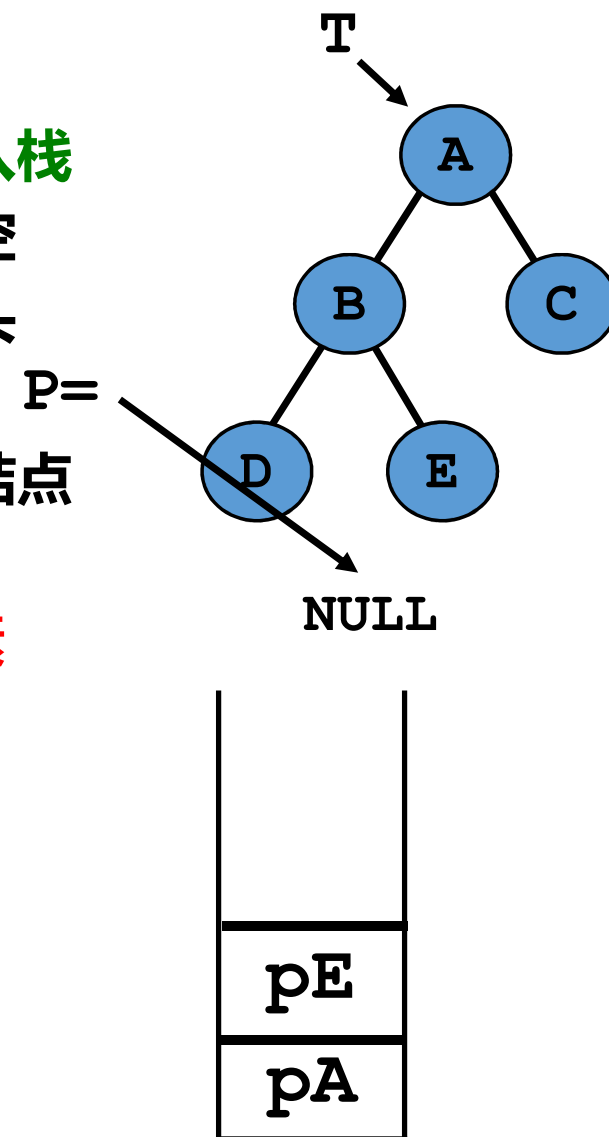


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)           //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                        //弹出多入栈的空结点
        if(!StackEmpty(S)) {              //如果栈非空
            Pop(S, p);                     //弹出一个元素
            if(!Visit(p->data))             //访问之
                return -1;
            Push(S, p->rchild);             //再向右走
        }
    }
    return 0;
}

```

D B

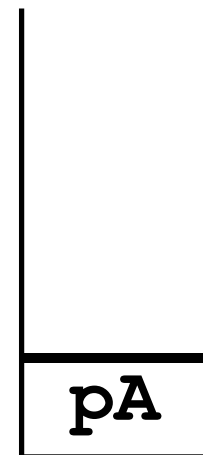
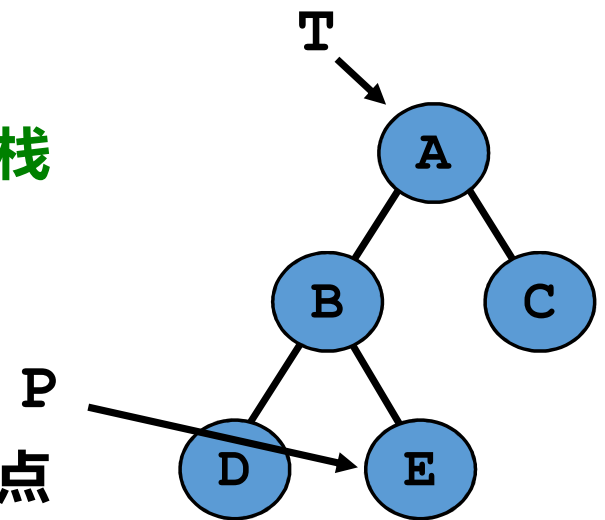


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {             //只要栈非空
        while(GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                     //弹出一个元素
            if(!Visit(p->data))            //访问之
                return -1;
            Push(S, p->rchild);             //再向右走
        }
    }
    return 0;
}

```

D   B   E

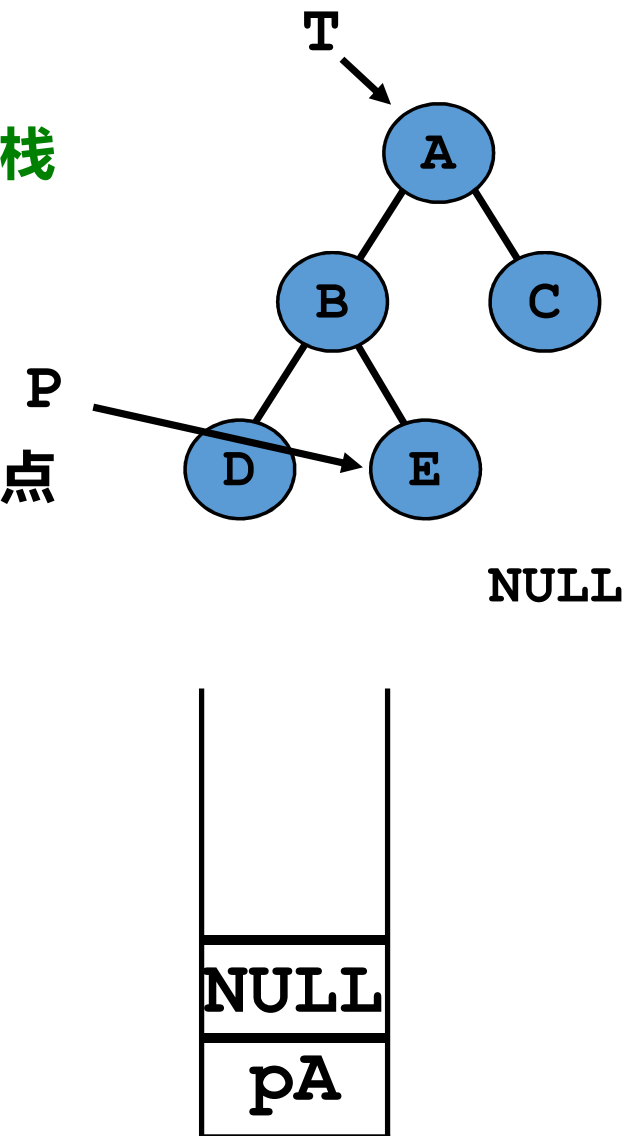


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D   B   E

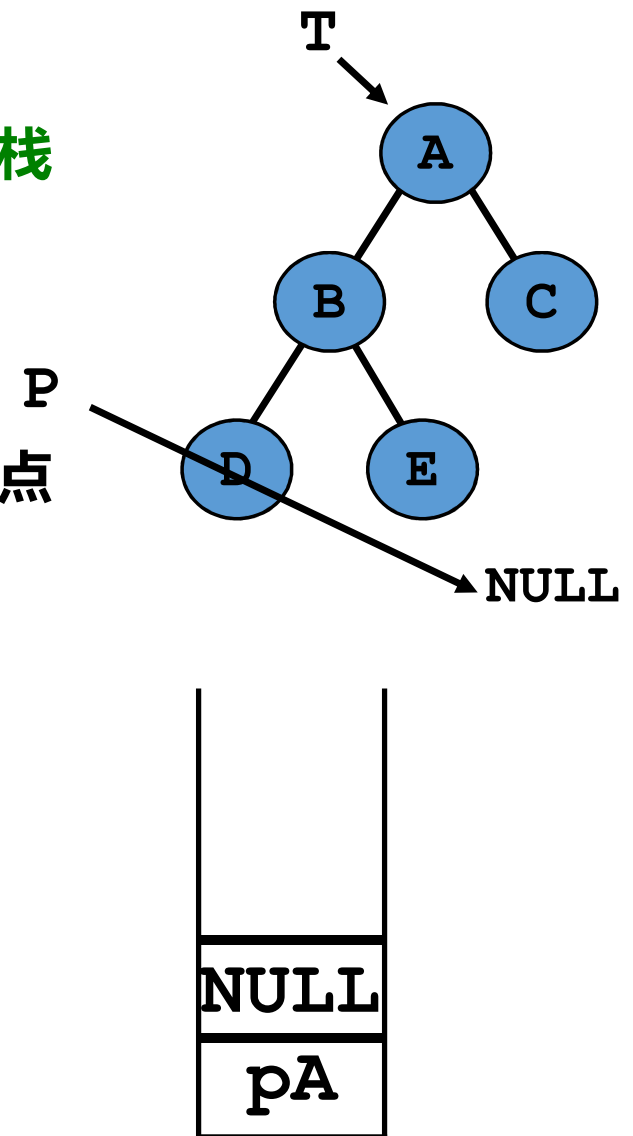


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))           //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D   B   E

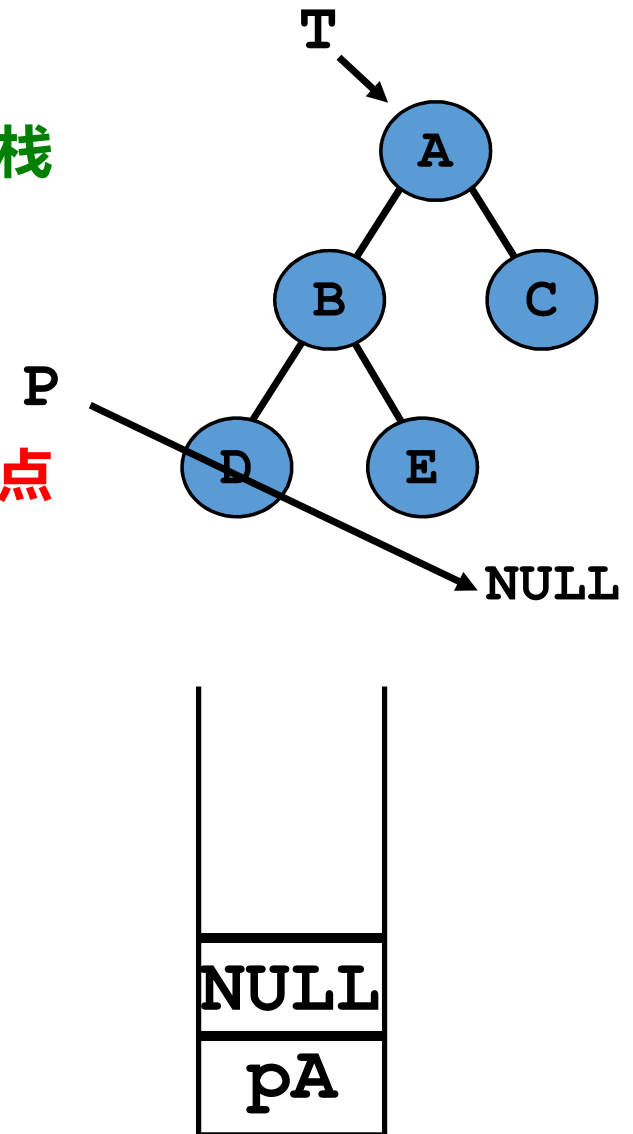


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);          //再向右走
        }
    }
    return 0;
}

```

D   B   E

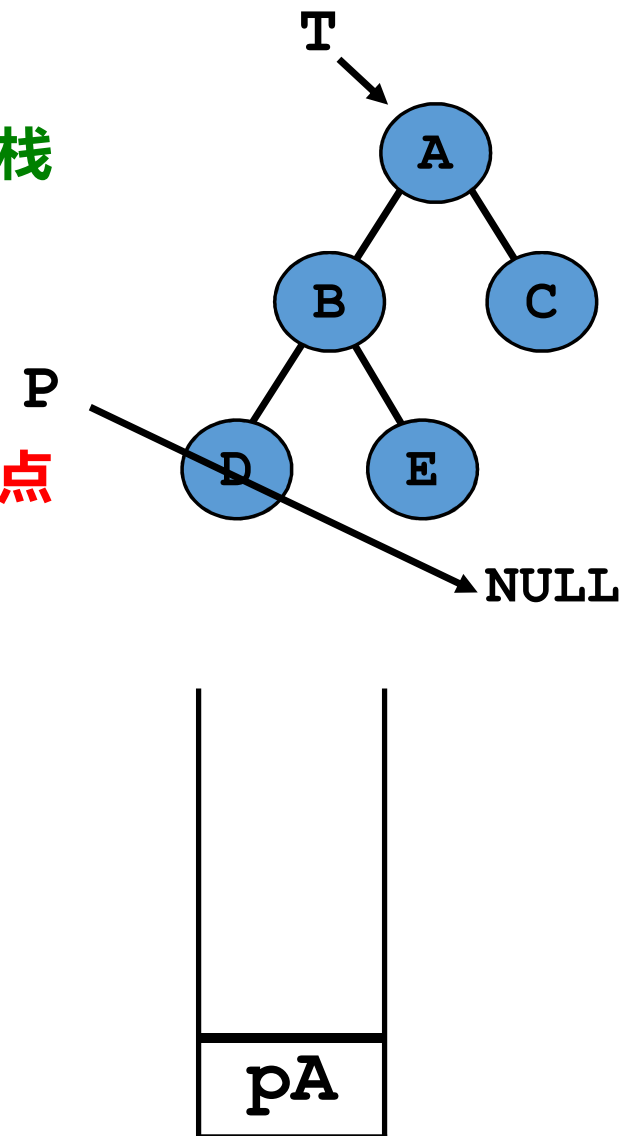


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                        //弹出多入栈的空结点
        if(!StackEmpty(S)) {              //如果栈非空
            Pop(S, p);                    //弹出一个元素
            if(!Visit(p->data))           //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D   B   E

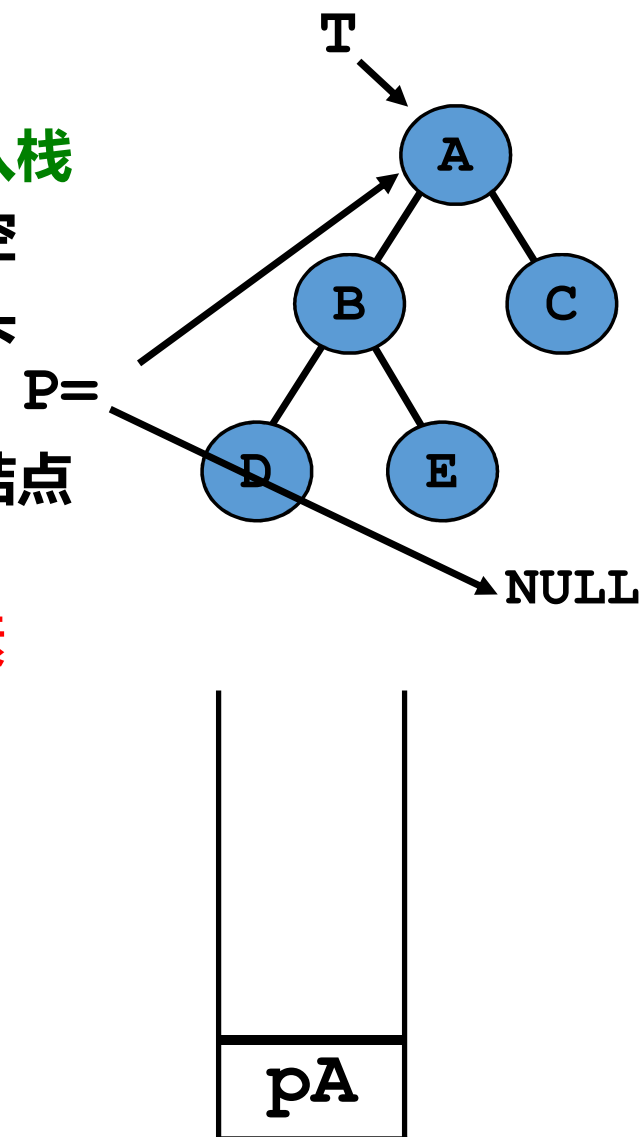


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                        //弹出多入栈的空结点
        if(!StackEmpty(S)) {              //如果栈非空
            Pop(S, p);                     //弹出一个元素
            if(!Visit(p->data))             //访问之
                return -1;
            Push(S, p->rchild);             //再向右走
        }
    }
    return 0;
}

```

D   B   E

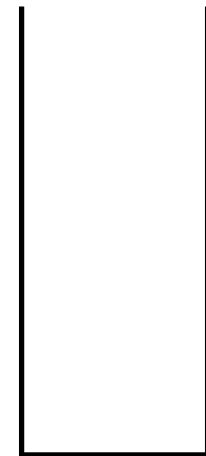
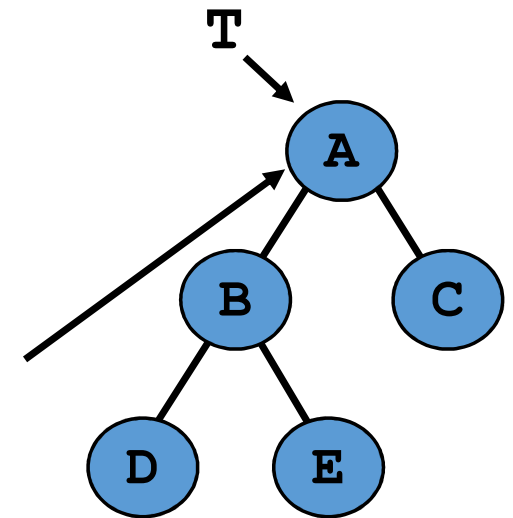


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);           P
        Pop(S, p);                        //弹出多入栈的空结点
        if(!StackEmpty(S)) {              //如果栈非空
            Pop(S, p);                     //弹出一个元素
            if(!Visit(p->data))             //访问之
                return -1;
            Push(S, p->rchild);             //再向右走
        }
    }
    return 0;
}

```

D   B   E   A



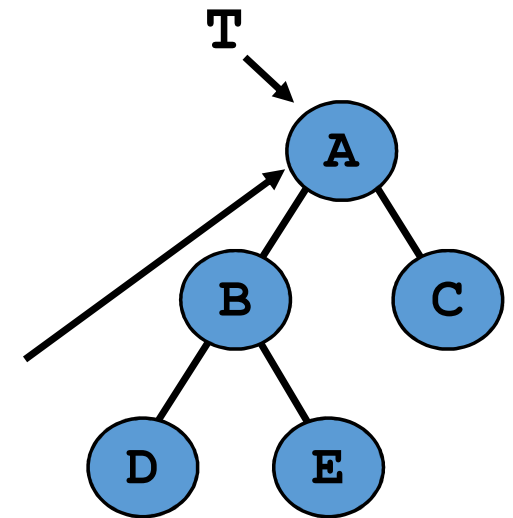


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {             //只要栈非空
        while(GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);          P
        Pop(S, p);                      //弹出多入栈的空结点
        if(!StackEmpty(S)) {           //如果栈非空
            Pop(S, p);                  //弹出一个元素
            if(!Visit(p->data)          //访问之
                return -1;
            Push(S, p->rchild);          //再向右走
        }
    }
    return 0;
}

```

D   B   E   A

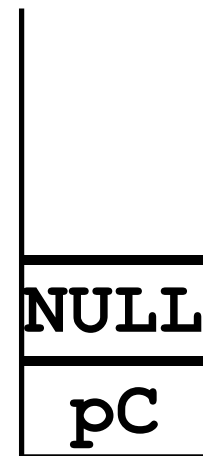
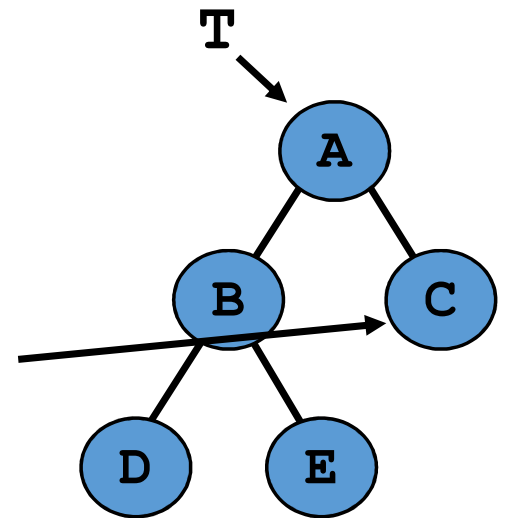


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while (!StackEmpty(S)) {             //只要栈非空
        while (GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);           P
        Pop(S, p);                        //弹出多入栈的空结点
        if (!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                     //弹出一个元素
            if (!Visit(p->data))           //访问之
                return -1;
            Push(S, p->rchild);             //再向右走
        }
    }
    return 0;
}

```

D   B   E   A

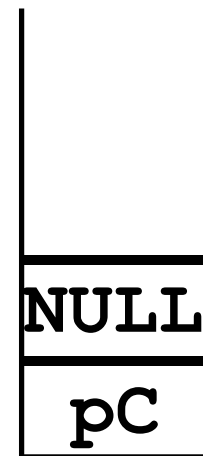
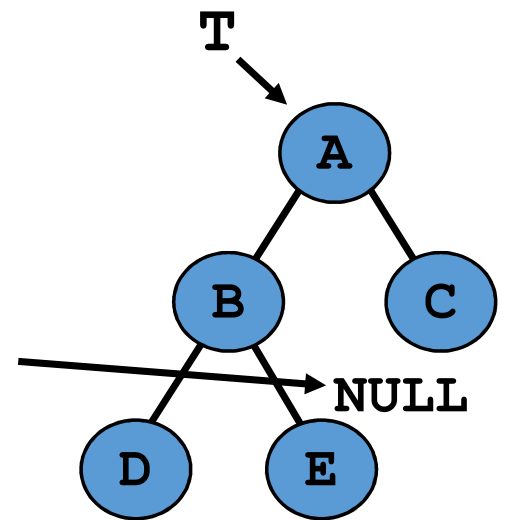


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while (!StackEmpty(S)) {             //只要栈非空
        while (GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);           P
        Pop(S, p);                        //弹出多入栈的空结点
        if (!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                    //弹出一个元素
            if (!Visit(p->data))           //访问之
                return -1;
            Push(S, p->rchild);             //再向右走
        }
    }
    return 0;
}

```

D   B   E   A

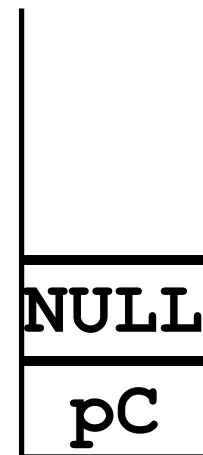
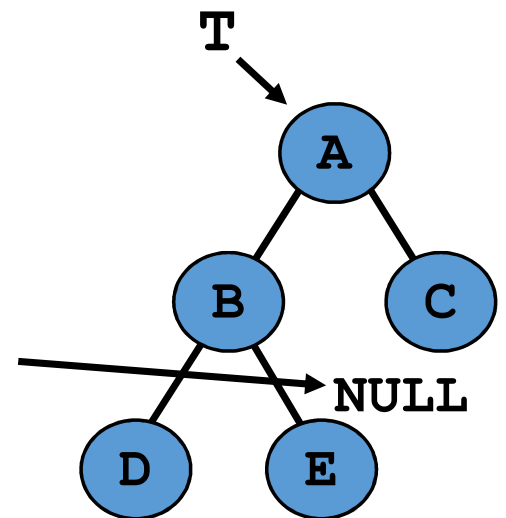


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                        //弹出多入栈的空结点
        if(!StackEmpty(S)) {              //如果栈非空
            Pop(S, p);                    //弹出一个元素
            if(!Visit(p->data))           //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D   B   E   A

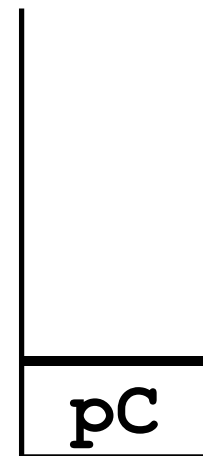
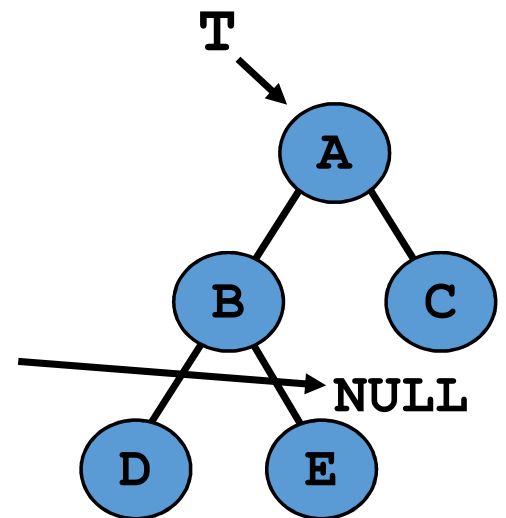


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {             //只要栈非空
        while(GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                     //弹出一个元素
            if(!Visit(p->data))            //访问之
                return -1;
            Push(S, p->rchild);             //再向右走
        }
    }
    return 0;
}

```

D   B   E   A

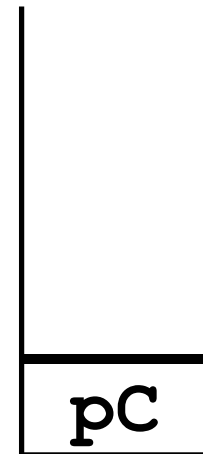
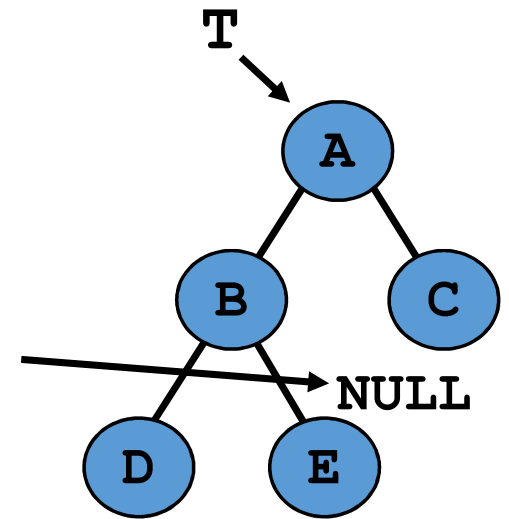


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {             //只要栈非空
        while(GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))           //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D   B   E   A

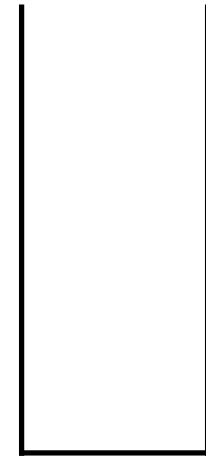
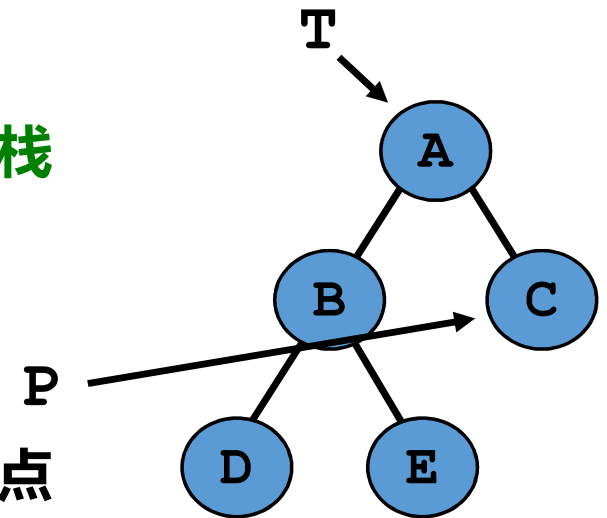


```

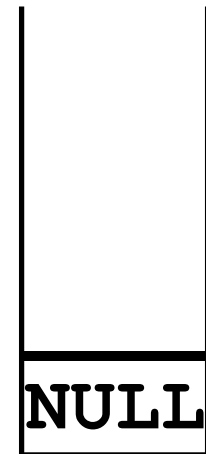
int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {             //只要栈非空
        while(GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if(!StackEmpty(S)) {            //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if(!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D   B   E   A   C

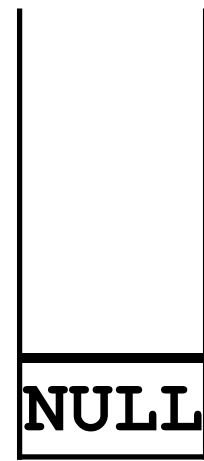


}





}



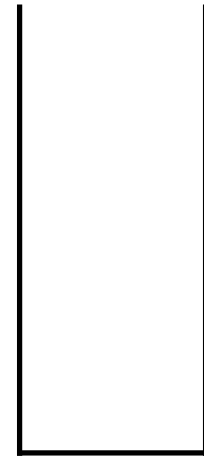
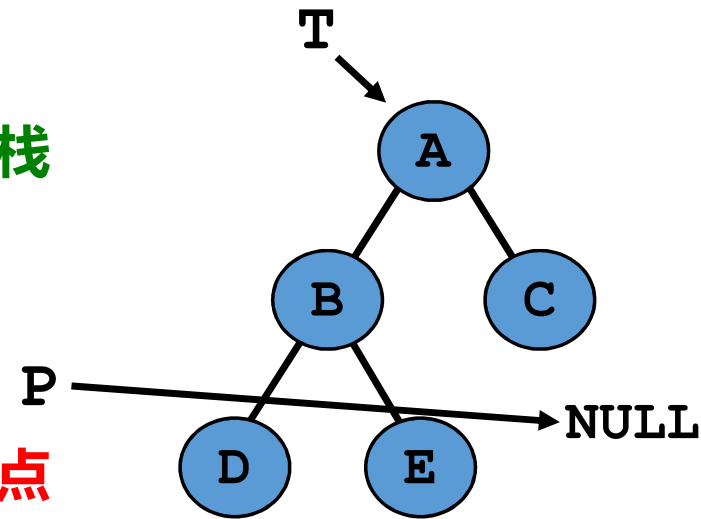


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                        //弹出多入栈的空结点
        if(!StackEmpty(S)) {              //如果栈非空
            Pop(S, p);                    //弹出一个元素
            if(!Visit(p->data))           //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D B E A C

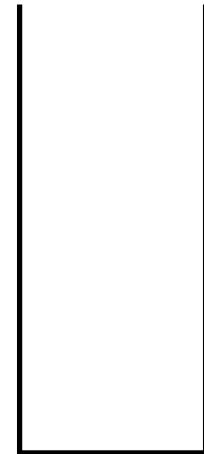
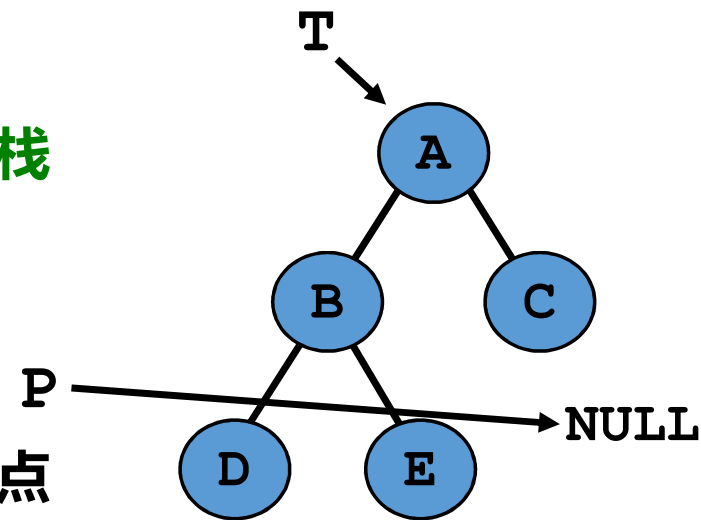


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while(!StackEmpty(S)) {              //只要栈非空
        while(GetTop(S, p) && p)          //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                        //弹出多入栈的空结点
        if(!StackEmpty(S)) {             //如果栈非空
            Pop(S, p);                    //弹出一个元素
            if(!Visit(p->data))           //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D B E A C

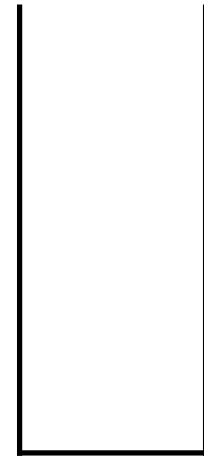
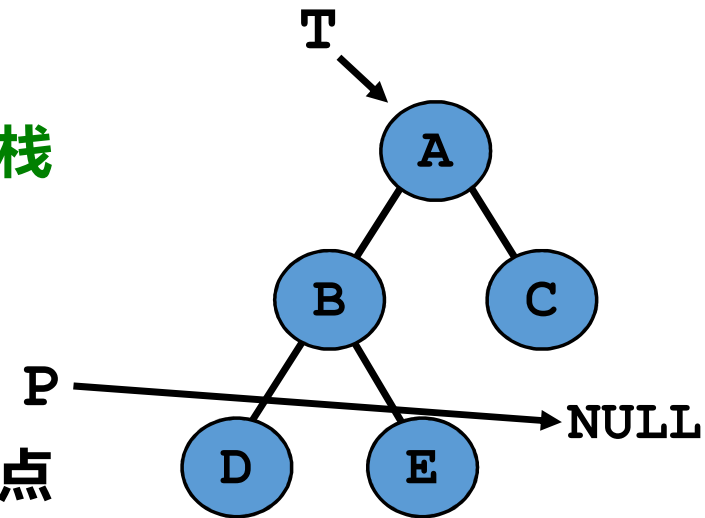


```

int InOrderTraverse (BiTree T) {
    InitStack(S);  Push(S, T);           //先把树根入栈
    while (!StackEmpty(S)) {             //只要栈非空
        while (GetTop(S, p) && p)         //向左走到头
            Push(S, p->lchild);
        Pop(S, p);                       //弹出多入栈的空结点
        if (!StackEmpty(S)) {            //如果栈非空
            Pop(S, p);                   //弹出一个元素
            if (!Visit(p->data))          //访问之
                return -1;
            Push(S, p->rchild);           //再向右走
        }
    }
    return 0;
}

```

D B E A C



```

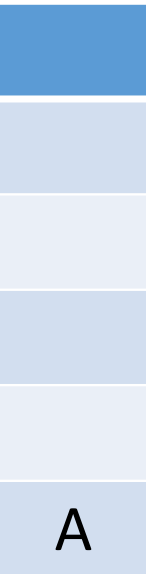
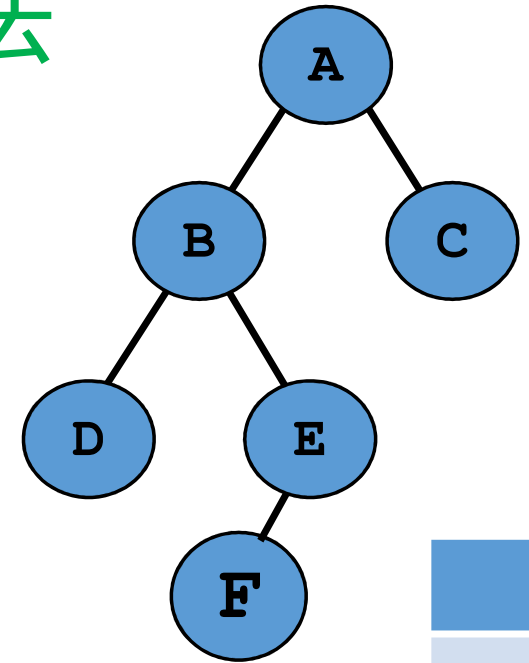
算法3  bool InOrderTraverse (BiTree T) {
    InitStack(S);           //新建一个堆栈
    p = T;                  //从树根开始
    while(p || !StackEmpty(S)) { //还有未访问的
        if(p) {             //一直向左走到底
            Push(S, p);
            p = p->lchild;
        }
        else {              //p为NULL, 说明走到了底
            Pop(S, p);       //弹出一个还没访问的结点
            Visit(p->data);  //访问之

            p = p->rchild;    //再向右走
        }
    }
    return true;
}

```

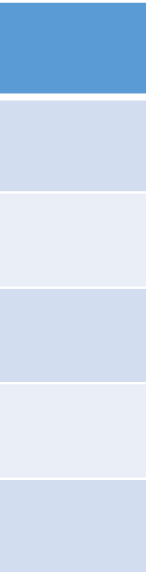
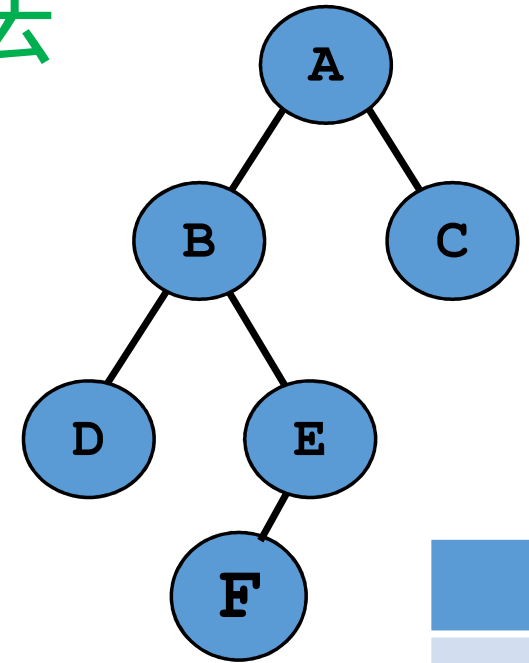
# 二叉树的先序非递归算法

```
void pre_dfs( root){  
    InitStack(S);  
    PushStack(S, root); //根结点入栈S  
    while( ! isEmpty(S) ){ //只要栈S不空  
  
    }  
}
```



# 二叉树的先序非递归算法

```
void pre_dfs( root){  
    InitStack(S);  
    PushStack(S, root); //根结点入栈S  
    while( ! isEmpty(S) ){ //只要栈S不空  
        node = S.pop(); //出栈一个结点  
        visit(node); //访问它  
    }  
}
```

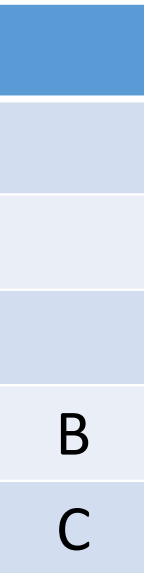
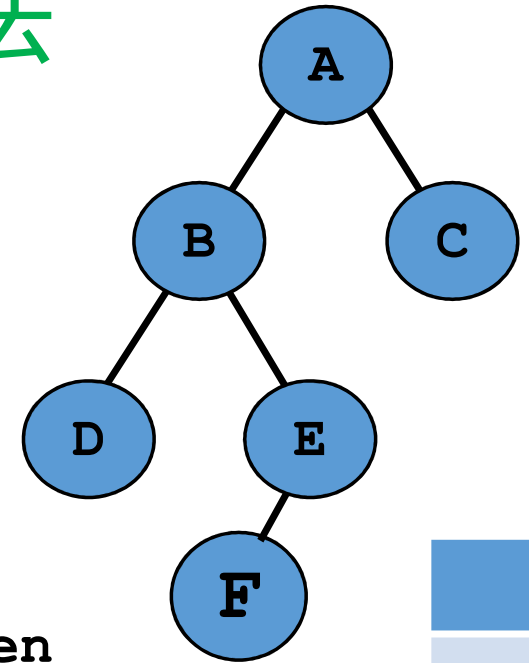


A



# 二叉树的先序非递归算法

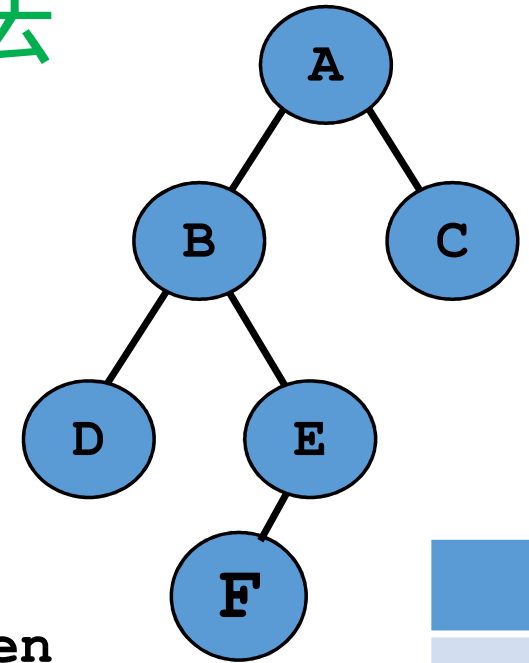
```
void pre_dfs( root){  
    InitStack(S);  
    PushStack(S, root); //根结点入栈S  
    while( ! isEmpty(S) ){ //只要栈S不空  
        node = S.pop(); //出栈一个结点  
        visit(node); //访问它  
        //push node's right and left children  
        if (node有右孩子rchild) PushStack(S, rchild);  
        if (node有左孩子lchild) PushStack(S, lchild);  
    }  
}
```



A

# 二叉树的先序非递归算法

```
void pre_dfs( root){  
    InitStack(S);  
    PushStack(S, root);    //根结点入栈S  
    while( ! isEmpty(S) ){    //只要栈S不空  
        node = S.pop();    //出栈一个结点  
        visit(node);        //访问它  
        //push node's right and left children  
        if(node有右孩子rchild) PushStack(S, rchild);  
        if(node有左孩子lchild) PushStack(S, lchild);  
    }  
}
```

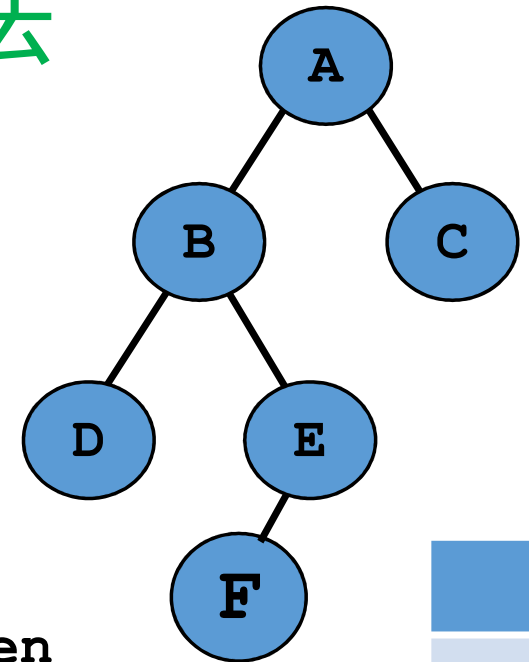


A B

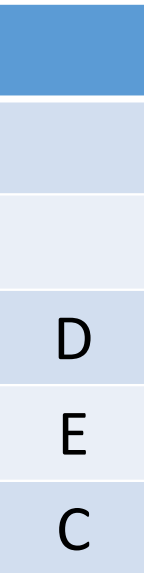
C

# 二叉树的先序非递归算法

```
void pre_dfs( root){  
    InitStack(S);  
    PushStack(S, root); //根结点入栈S  
    while( ! isEmpty(S) ){ //只要栈S不空  
        node = S.pop(); //出栈一个结点  
        visit(node); //访问它  
        //push node's right and left children  
        if(node有右孩子rchild) PushStack(S, rchild);  
        if(node有左孩子lchild) PushStack(S, lchild);  
    }  
}
```

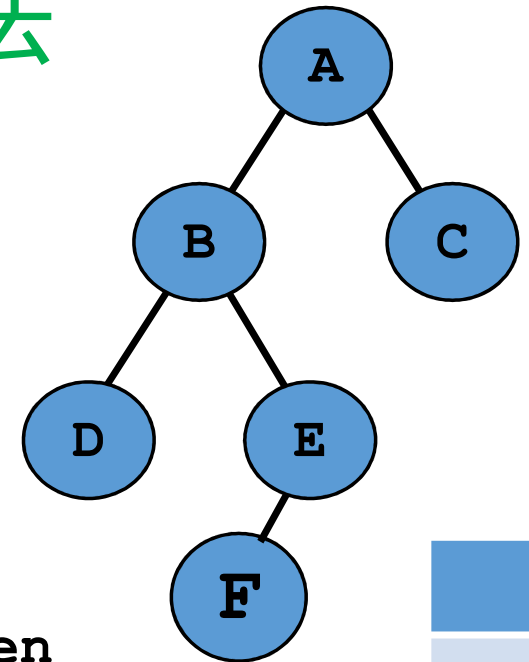


A B

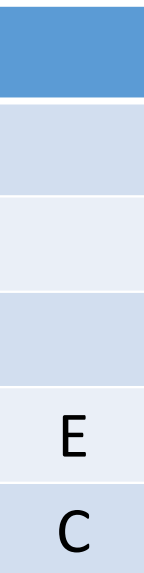


# 二叉树的先序非递归算法

```
void pre_dfs( root){  
    InitStack(S);  
    PushStack(S, root); //根结点入栈S  
    while( ! isEmpty(S) ){ //只要栈S不空  
        node = S.pop(); //出栈一个结点  
        visit(node); //访问它  
        //push node's right and left children  
        if(node有右孩子rchild) PushStack(S, rchild);  
        if(node有左孩子lchild) PushStack(S, lchild);  
    }  
}
```

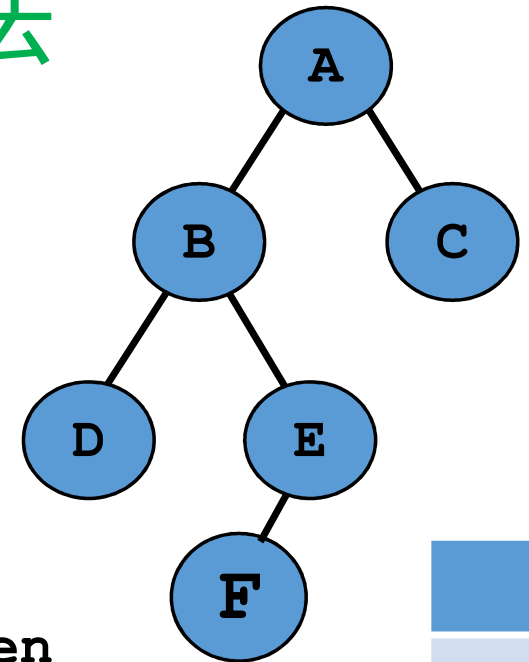


A B D



# 二叉树的先序非递归算法

```
void pre_dfs( root){  
    InitStack(S);  
    PushStack(S, root); //根结点入栈S  
    while( ! isEmpty(S) ){ //只要栈S不空  
        node = S.pop(); //出栈一个结点  
        visit(node); //访问它  
        //push node's right and left children  
        if(node有右孩子rchild) PushStack(S, rchild);  
        if(node有左孩子lchild) PushStack(S, lchild);  
    }  
}
```

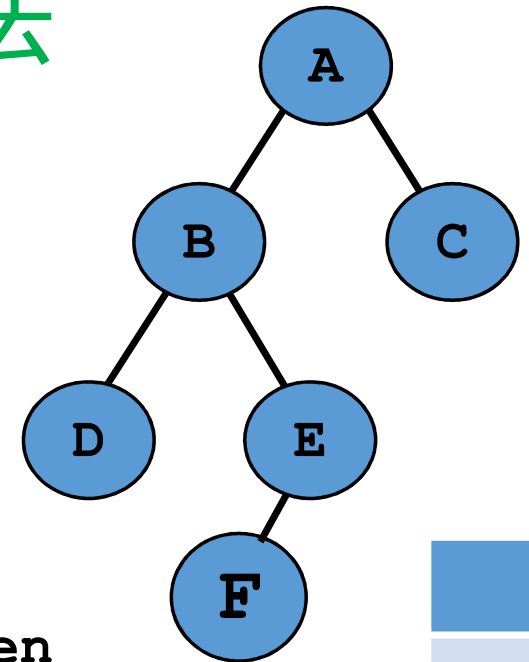


A B D E

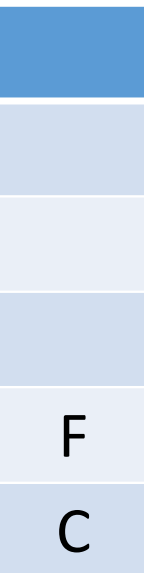
C

# 二叉树的先序非递归算法

```
void pre_dfs( root){  
    InitStack(S);  
    PushStack(S, root);    //根结点入栈S  
    while( ! isEmpty(S) ){    //只要栈S不空  
        node = S.pop();    //出栈一个结点  
        visit(node);        //访问它  
        //push node's right and left children  
        if (node有右孩子rchild) PushStack(S, rchild);  
        if (node有左孩子lchild) PushStack(S, lchild);  
    }  
}
```

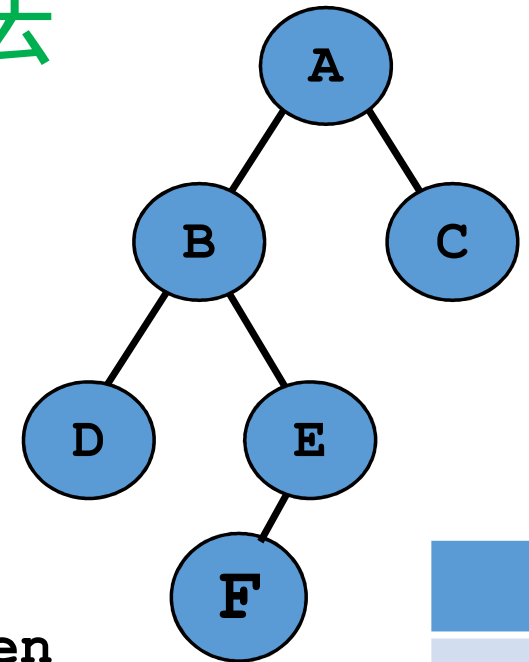


A B D E



# 二叉树的先序非递归算法

```
void pre_dfs( root){  
    InitStack(S);  
    PushStack(S, root); //根结点入栈S  
    while( ! isEmpty(S) ){ //只要栈S不空  
        node = S.pop(); //出栈一个结点  
        visit(node); //访问它  
        //push node's right and left children  
        if(node有右孩子rchild) PushStack(S, rchild);  
        if(node有左孩子lchild) PushStack(S, lchild);  
    }  
}
```

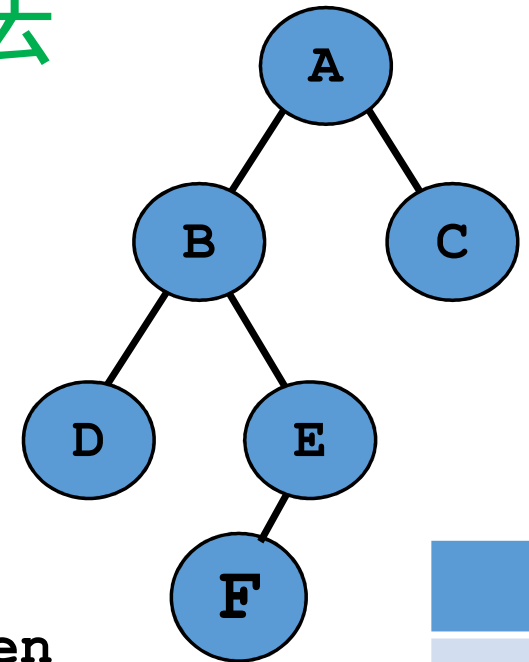


A B D E F

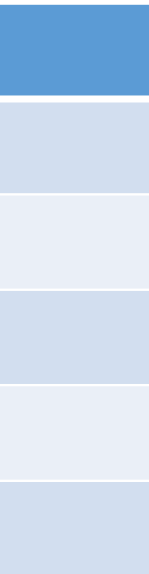
C

# 二叉树的先序非递归算法

```
void pre_dfs( root){  
    InitStack(S);  
    PushStack(S, root); //根结点入栈S  
    while( ! isEmpty(S) ){ //只要栈S不空  
        node = S.pop(); //出栈一个结点  
        visit(node); //访问它  
        //push node's right and left children  
        if(node有右孩子rchild) PushStack(S, rchild);  
        if(node有左孩子lchild) PushStack(S, lchild);  
    }  
}
```



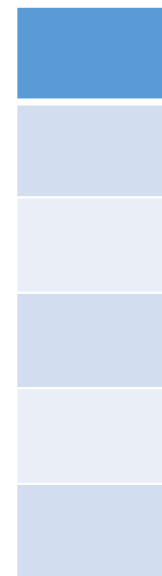
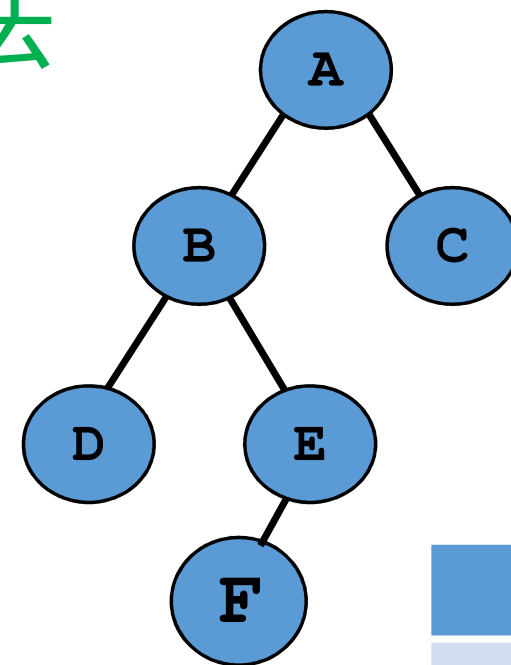
A B D E F C





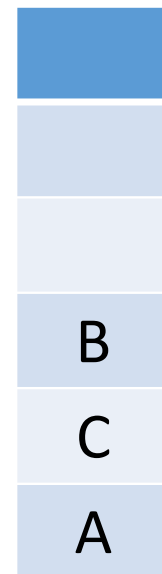
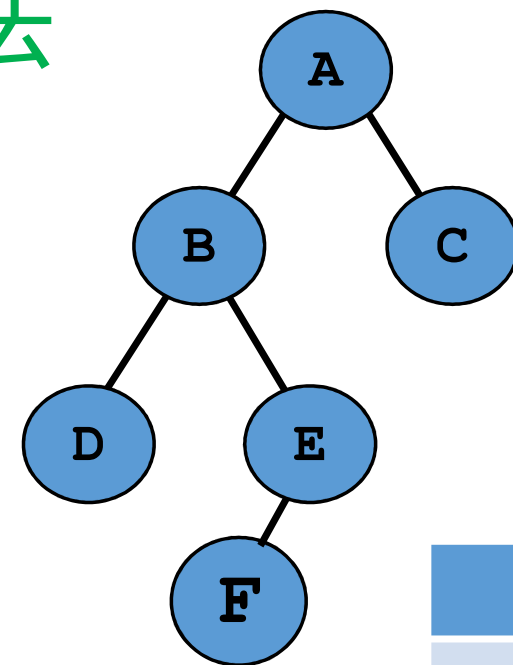
# 二叉树的后序非递归算法

- 如何做?



# 二叉树的后序非递归算法

- 如何做?



# 二叉树的后序非递归算法

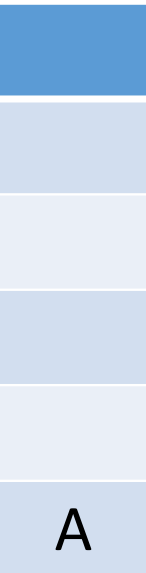
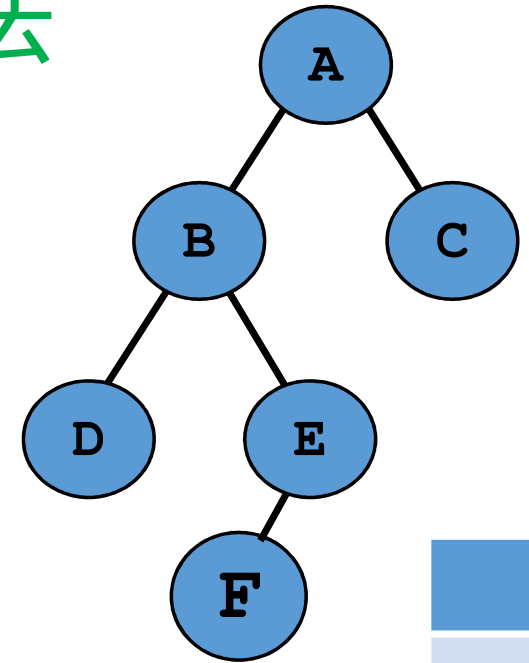
- 修改先序非递归遍历得到后序非递归遍历

先:

直接访问p;  
访问p的右子树  
访问p的左子树

后:

得到的序列逆序



# 二叉树的后序非递归算法

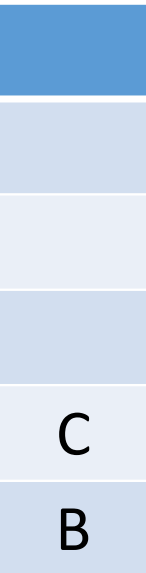
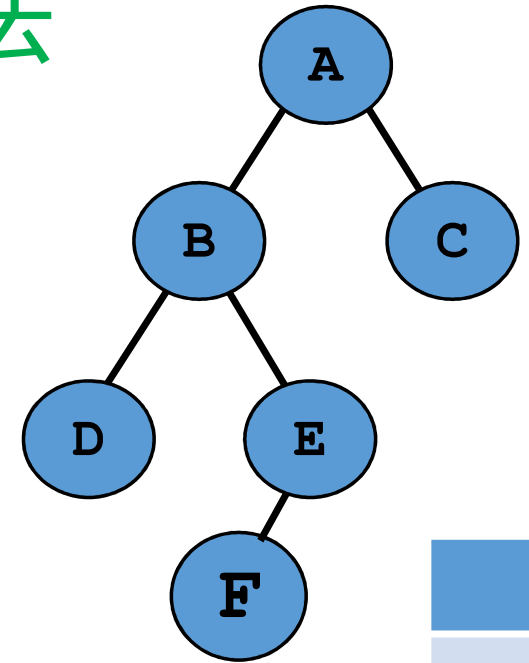
- 修改先序非递归遍历得到后序非递归遍历

先:

直接访问p;  
访问p的右子树  
访问p的左子树

后:

得到的序列逆序



A

# 二叉树的后序非递归算法

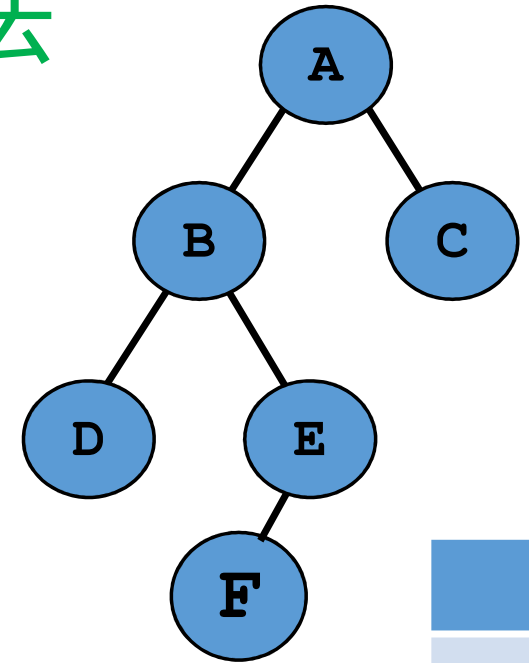
- 修改先序非递归遍历得到后序非递归遍历

先:

直接访问p;  
访问p的右子树  
访问p的左子树

后:

得到的序列逆序



A C

B

# 二叉树的后序非递归算法

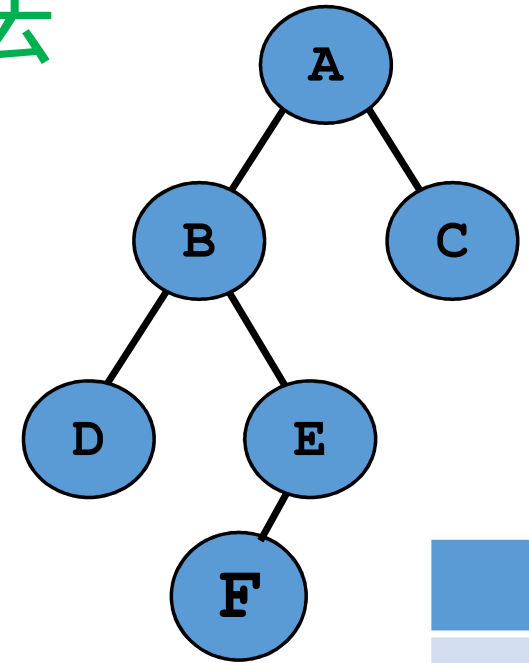
- 修改先序非递归遍历得到后序非递归遍历

先:

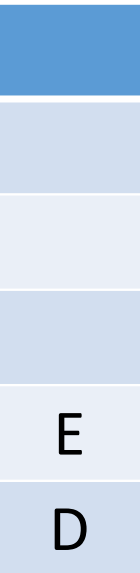
直接访问p;  
访问p的右子树  
访问p的左子树

后:

得到的序列逆序



A C B



# 二叉树的后序非递归算法

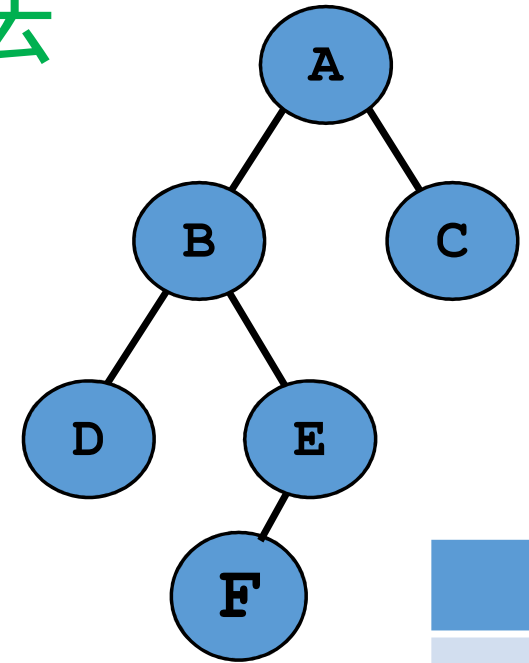
- 修改先序非递归遍历得到后序非递归遍历

先:

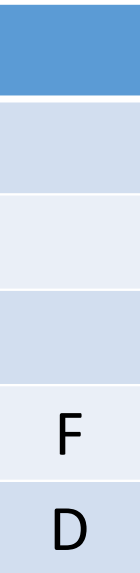
直接访问p;  
访问p的右子树  
访问p的左子树

后:

得到的序列逆序



A C B E



# 二叉树的后序非递归算法

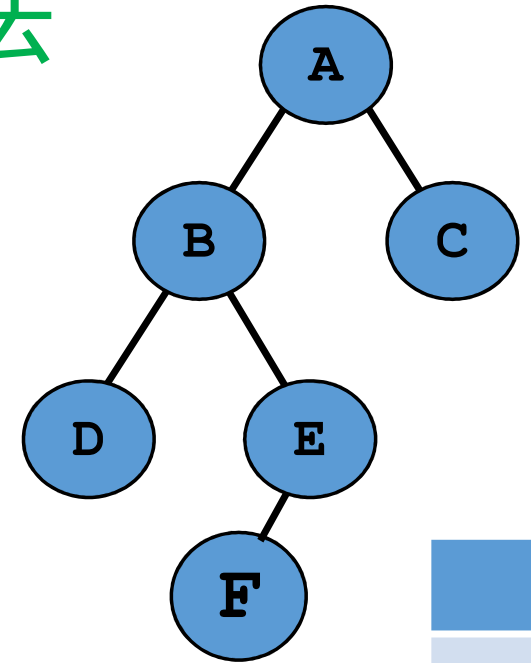
- 修改先序非递归遍历得到后序非递归遍历

先:

直接访问p;  
访问p的右子树  
访问p的左子树

后:

得到的序列逆序



A C B E F

D



# 二叉树的后序非递归算法

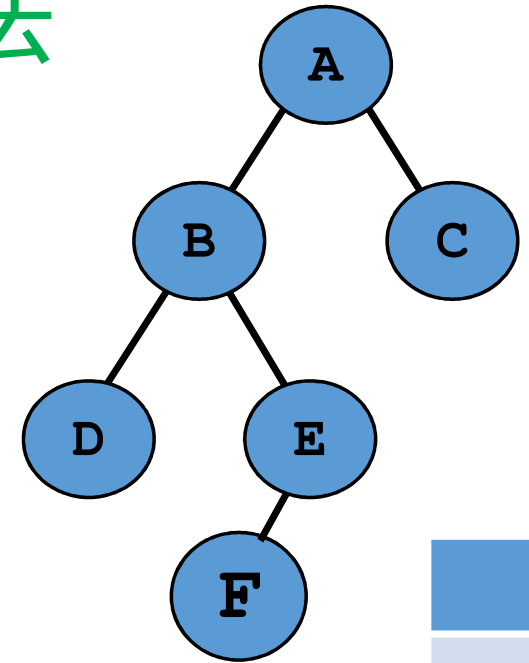
- 修改先序非递归遍历得到后序非递归遍历

先:

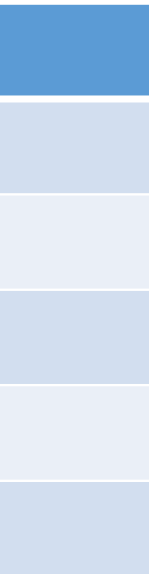
直接访问p;  
访问p的右子树  
访问p的左子树

后:

得到的序列逆序



A C B E F D



# 二叉树的后序非递归算法

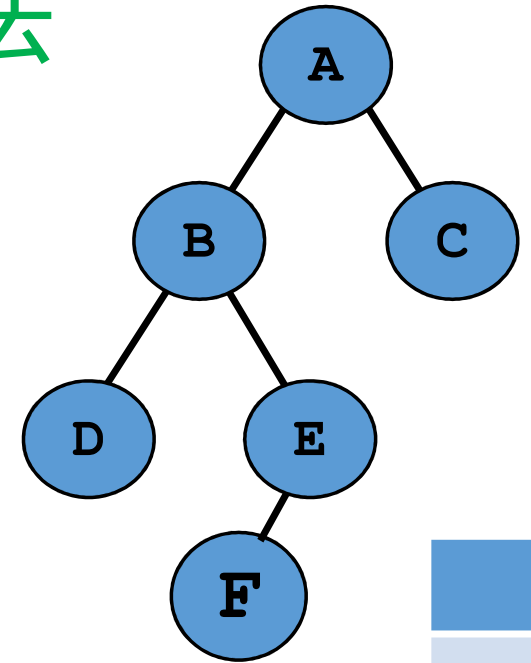
- 修改先序非递归遍历得到后序非递归遍历

先:

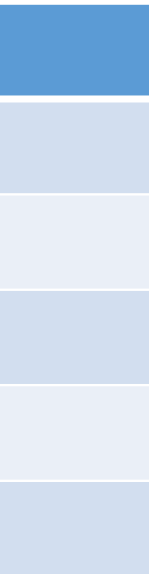
直接访问p;  
访问p的右子树  
访问p的左子树

后:

得到的序列逆序



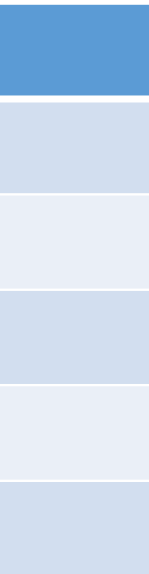
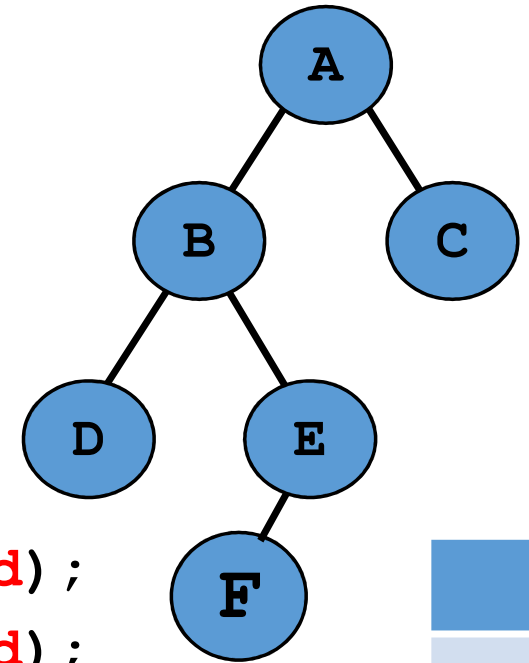
D F E B C A



```

void post_dfs( root){
    InitStack( R);
    InitStack(S);
    PushStack(S, root); //根结点入栈S
    while( ! isEmpty(S) ){ //只要栈S不空
        node = S.pop(); //出栈一个结点
        PushStack(R,node); //访问它
        //push node's right and left children
        if(node有左孩子lchild) PushStack(S, lchild);
        if(node有右孩子rchild) PushStack(S, rchild);
    }
    while( ! isEmpty(R) ){
        node = R.pop(); //出栈一个结点
        visit(node); //访问它
    }
}

```



```
vector<int> postorderTraversal(TreeNode *root) {  
    list<int> result;  
    stack<TreeNode *> S;  
    TreeNode *p = root;  
    while (!S.empty() || p != nullptr) {  
        if (p != nullptr) { //访问p,并走向右子树  
            S.push(p);  
            result.push_front(p->val);  
            p = p->right;  
        }  
        else { //右子树走完了  
            TreeNode *cur = S.top();  
            S.pop();  
            p = cur->left;  
        }  
    }  
    return vector<int>(result.rbegin(), result.rend());  
}
```

```

vector<int> postorderTraversal(TreeNode *root) {
    vector<int> result;
    stack<const TreeNode *> s;
    /* p, 正在访问的结点, q, 刚刚访问过的结点 */
    const TreeNode *p = root, *q = nullptr;

    do {
        while (p != nullptr) { /* 往左下走 */
            s.push(p);
            p = p->left;
        }
        q = nullptr;
        while (!s.empty()) {
            p = s.top();
            s.pop();
            /* 右孩子不存在或已被访问, 访问之 */
            if (p->right == q) {
                result.push_back(p->val);
                q = p; /* 保存刚访问过的结点 */
            } else {
                /* 当前结点不能访问, 需第二次进栈 */
                s.push(p);
                /* 先处理右子树 */
                p = p->right;
                break;
            }
        }
    } while (!s.empty());

    return result;
}

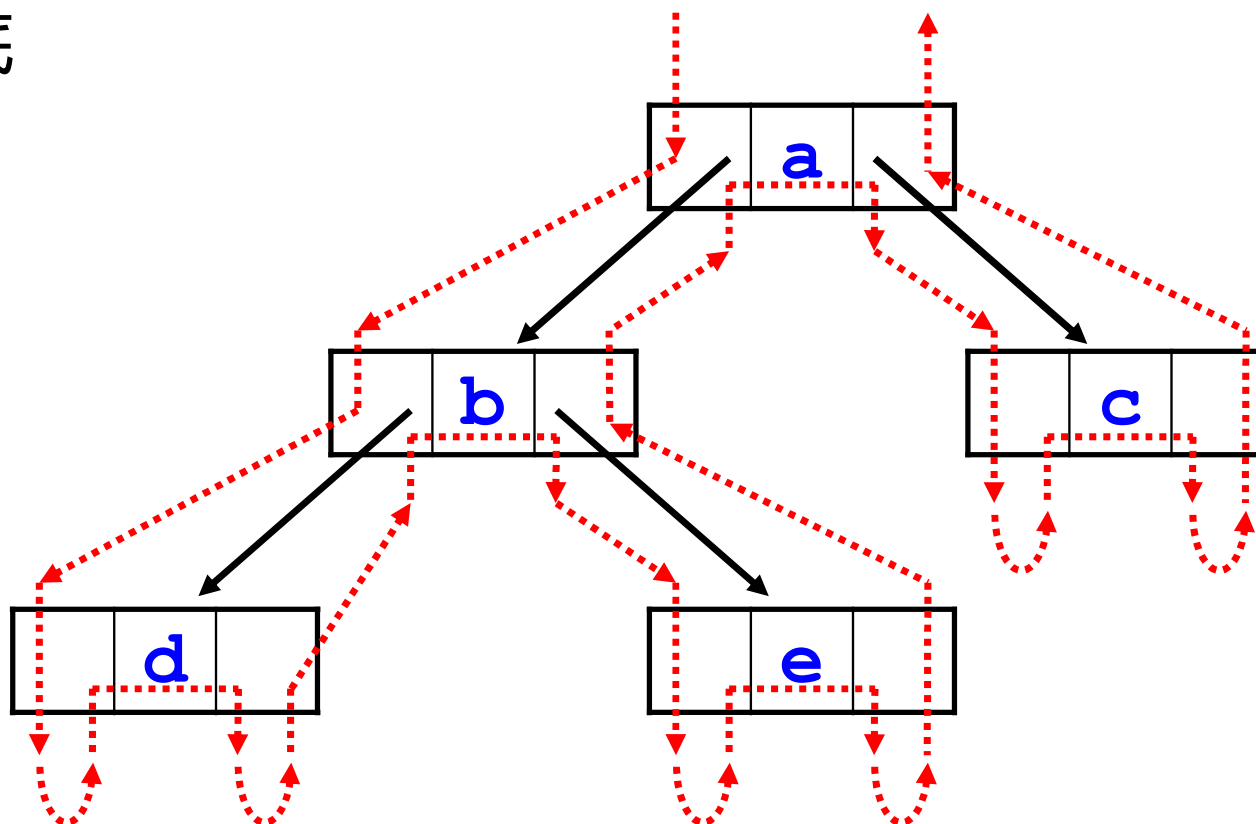
```

# 二叉树的遍历：算法效率

- 时间复杂度
  - $n$ 个结点，每一个都要访问一次
  - 显然时间复杂度为 $O(n)$ （这里 $n$ 为结点数）
- 空间复杂度
  - 不论是递归还是非递归算法都要用到堆栈
  - 堆栈的最大深度 = 树的深度
  - 所有空间复杂度为 $O(k)$ （这里 $k$ 为树的深度）

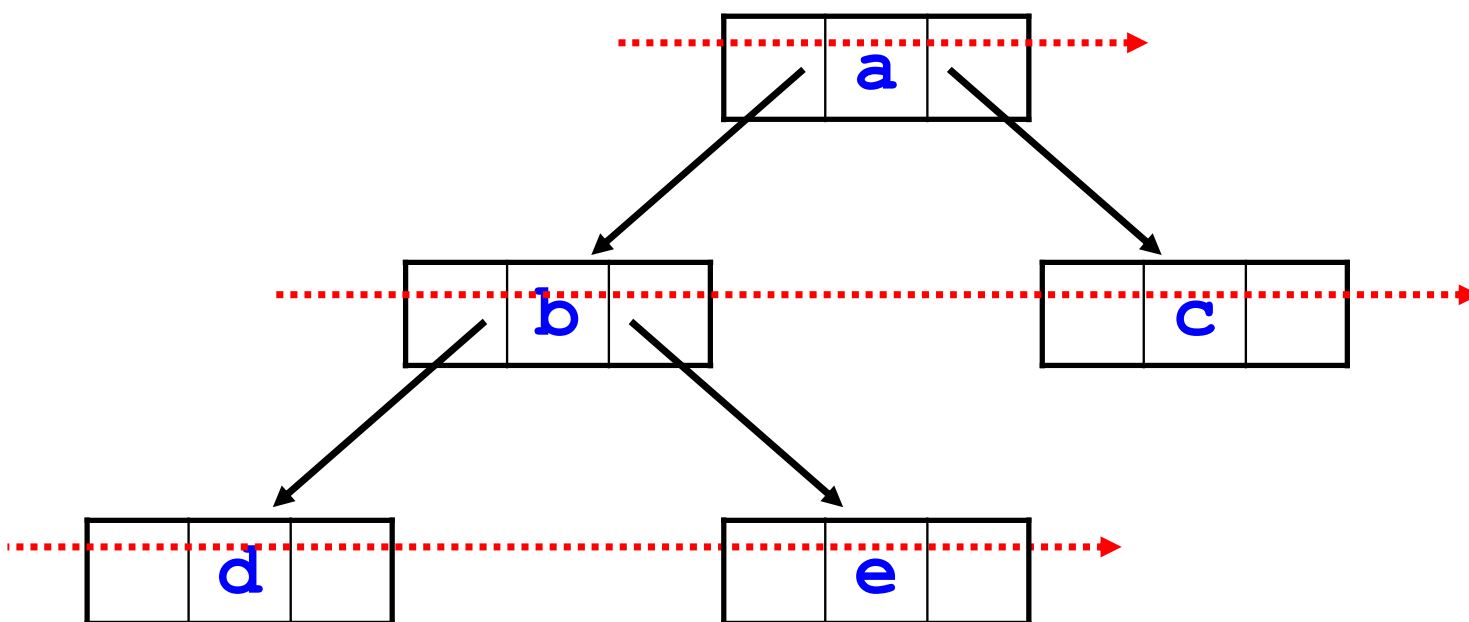
## 二叉树的深度优先遍历-同一路径

- 先走到底
- 再回头



# 二叉树的广度(层次)优先遍历

- 一层一层的访问：类似水波一层一层地扩散





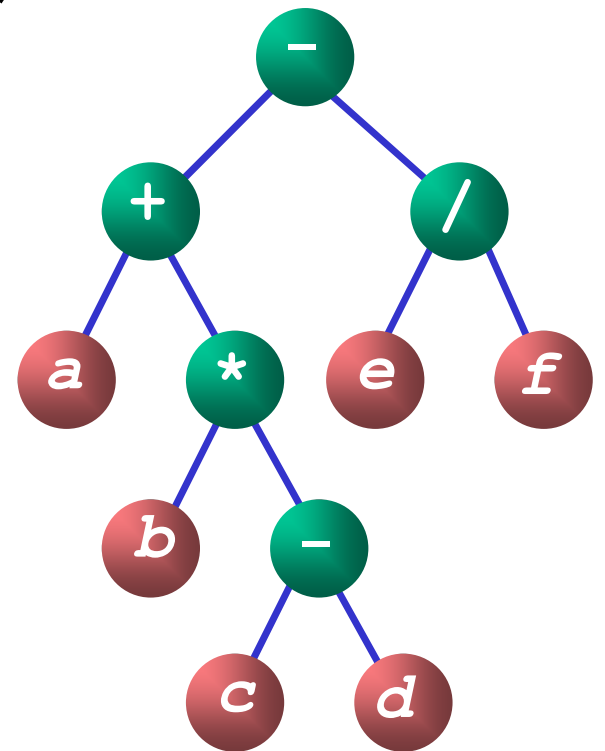
# 二叉树的广度(层次)优先遍历

- 一层一层的访问：类似水波一层一层地扩散

$- + / a * e f b - c d$

“老爸在长辈中在先，则儿子在晚辈中也在先”

“先进先出”特点，因此需要借助于“队列”



```
int LevelOrderTraverse (BiTree T){
    InitQueue(Q);  AddQueue(Q, T); //树根入队

    while (!QueueEmpty(Q)) {           //只要队列非空
        DeleteQueue(Q, p);              //出队一个结点
        if (!Visit(p->data))            //访问之
            return -1;

        if (p->lchild)                  //左孩子入队
            AddQueue(Q, p->lchild);

        if (p->rchild)                  //右孩子入队
            AddQueue(Q, p->rchild);
    }
    return 0;
}
```

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while(!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if(!Visit(p->data))
            return -1;
        if(p->lchild)
            AddQueue(Q, p->lchild);
        if(p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

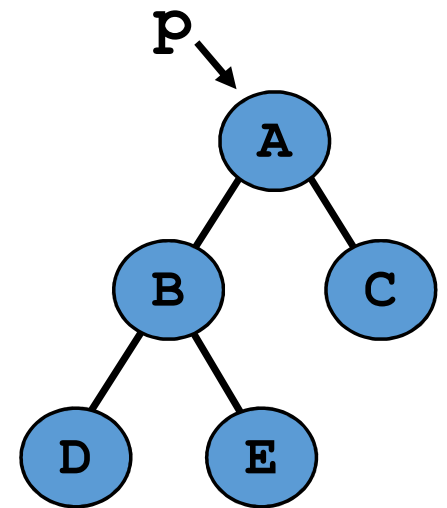
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队




---

A

---

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while(!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if(!Visit(p->data))
            return -1;
        if(p->lchild)
            AddQueue(Q, p->lchild);
        if(p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

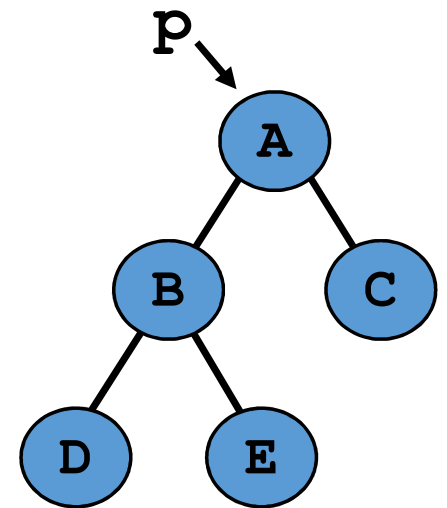
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队




---

A

---

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while(!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if(!Visit(p->data))
            return -1;
        if(p->lchild)
            AddQueue(Q, p->lchild);
        if(p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

A

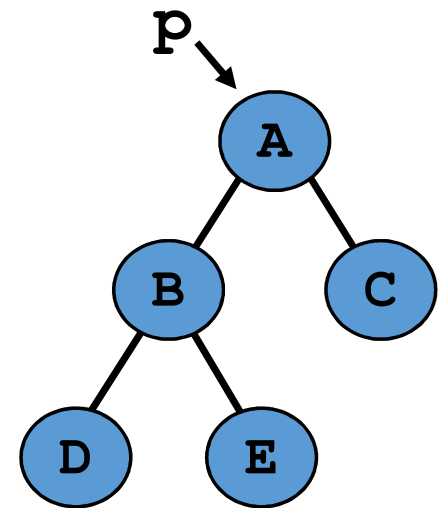
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队



B C

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while (!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if (!Visit(p->data))
            return -1;
        if (p->lchild)
            AddQueue(Q, p->lchild);
        if (p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

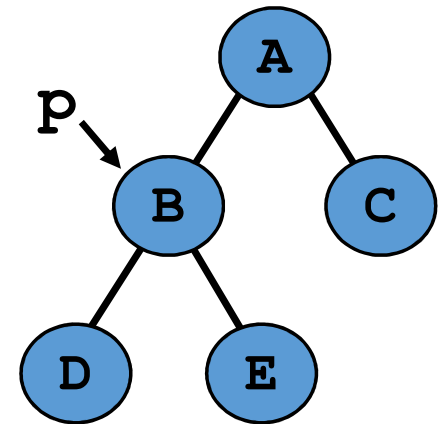
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队



A

---

B C

---

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while(!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if(!Visit(p->data))
            return -1;
        if(p->lchild)
            AddQueue(Q, p->lchild);
        if(p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

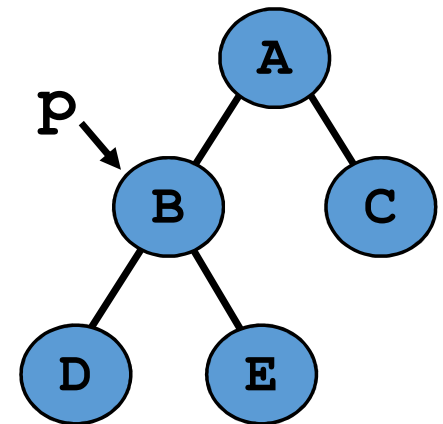
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队



A B

C

```
int LevelOrderTraverse(BiTree T) {
```

```
    InitQueue(Q); AddQueue(Q, T);
```

树根入队

```
    while (!QueueEmpty(Q)) {
```

只要队列非空

```
        DeleteQueue(Q, p);
```

出队一个节点  
并访问之

```
        if (!Visit(p->data))
```

```
            return -1;
```

```
        if (p->lchild)
```

如果p有左孩子  
则左孩子入队

```
            AddQueue(Q, p->lchild);
```

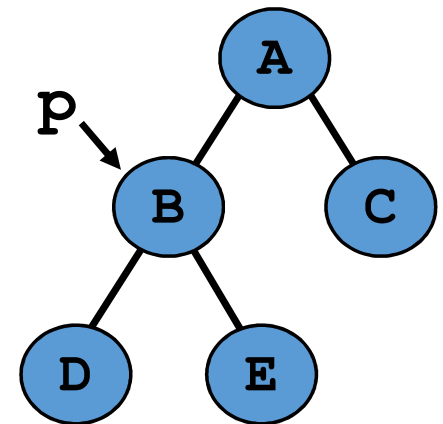
```
        if (p->rchild)
```

如果p有右孩子  
则右孩子入队

```
            AddQueue(Q, p->rchild);
```

```
    }
    return 0;
```

程序结束



A B

C

D E



```
int LevelOrderTraverse(BiTree T) {
```

```
    InitQueue(Q); AddQueue(Q, T);
```

树根入队

```
    while(!QueueEmpty(Q)) {
```

只要队列非空

```
        DeleteQueue(Q, p);
```

出队一个节点  
并访问之

```
        if(!Visit(p->data))
```

```
            return -1;
```

```
        if(p->lchild)
```

```
            AddQueue(Q, p->lchild);
```

如果p有左孩子  
则左孩子入队

```
        if(p->rchild)
```

```
            AddQueue(Q, p->rchild);
```

如果p有右孩子  
则右孩子入队

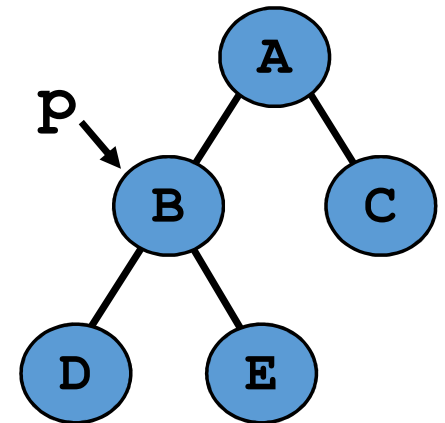
```
    }
```

```
    return 0;
```

```
}
```

程序结束

A B



---

C D E

---

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while(!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if(!Visit(p->data))
            return -1;
        if(p->lchild)
            AddQueue(Q, p->lchild);
        if(p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

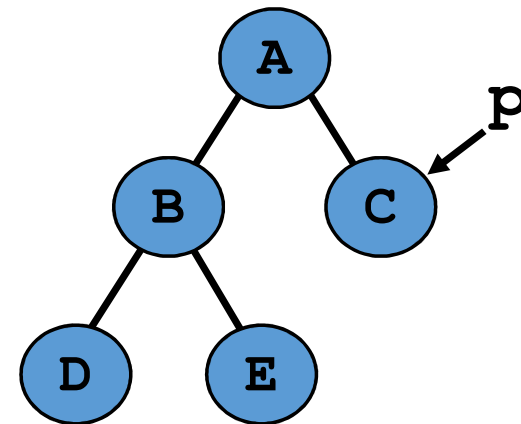
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队



A B

C D E

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while(!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if(!Visit(p->data))
            return -1;
        if(p->lchild)
            AddQueue(Q, p->lchild);
        if(p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

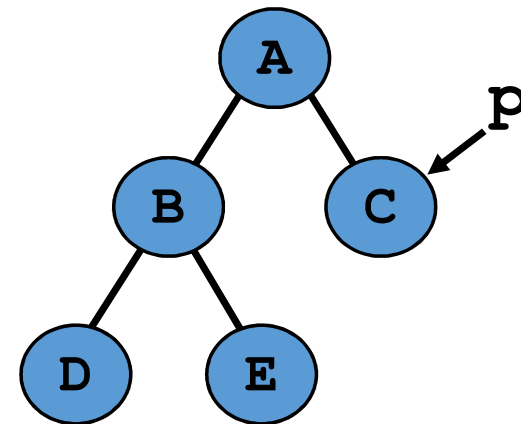
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队



A B C

D E

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while (!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if (!Visit(p->data))
            return -1;
        if (p->lchild)
            AddQueue(Q, p->lchild);
        if (p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

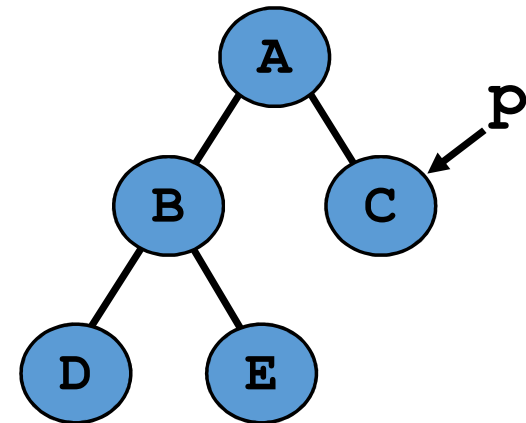
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队



A B C

D E

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while(!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if(!Visit(p->data))
            return -1;
        if(p->lchild)
            AddQueue(Q, p->lchild);
        if(p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

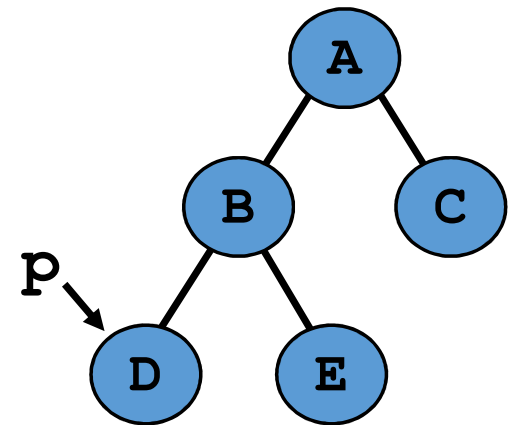
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队



A B C

D E

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while(!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if(!Visit(p->data))
            return -1;
        if(p->lchild)
            AddQueue(Q, p->lchild);
        if(p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

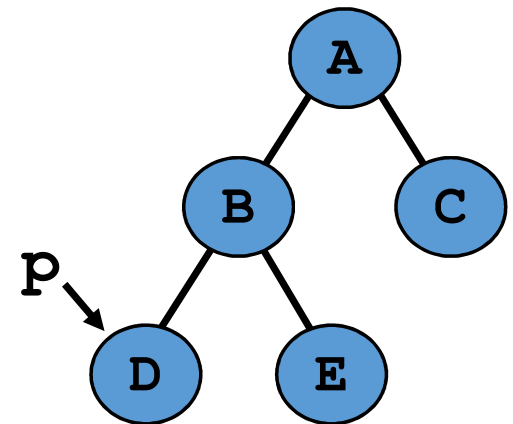
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队



A B C D

E

```
int LevelOrderTraverse(BiTree T) {
```

```
    InitQueue(Q); AddQueue(Q, T);
```

树根入队

```
    while(!QueueEmpty(Q)) {
```

只要队列非空

```
        DeleteQueue(Q, p);
```

出队一个节点  
并访问之

```
        if(!Visit(p->data))
```

```
            return -1;
```

```
        if(p->lchild)
```

如果p有左孩子  
则左孩子入队

```
            AddQueue(Q, p->lchild);
```

```
        if(p->rchild)
```

如果p有右孩子  
则右孩子入队

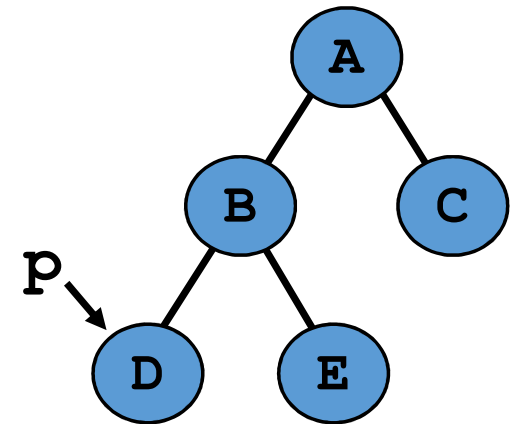
```
            AddQueue(Q, p->rchild);
```

```
    }
```

```
    return 0;
```

```
}
```

程序结束



A B C D

E

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q); AddQueue(Q, T);
    while(!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if(!Visit(p->data))
            return -1;
        if(p->lchild)
            AddQueue(Q, p->lchild);
        if(p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

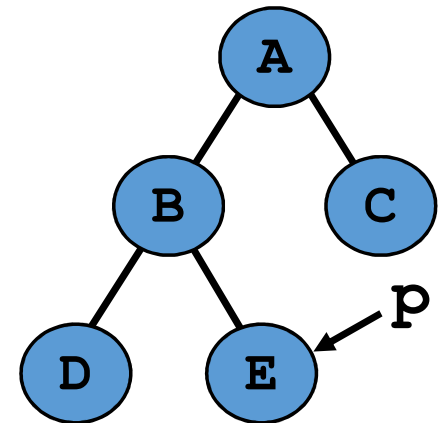
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队



A B C D

E



```
int LevelOrderTraverse(BiTree T) {
```

```
    InitQueue(Q); AddQueue(Q, T);
```

树根入队

```
    while (!QueueEmpty(Q)) {
```

只要队列非空

```
        DeleteQueue(Q, p);
```

```
        if (!Visit(p->data))
```

出队一个节点  
并访问之

```
            return -1;
```

```
        if (p->lchild)
```

如果p有左孩子  
则左孩子入队

```
            AddQueue(Q, p->lchild);
```

```
        if (p->rchild)
```

如果p有右孩子  
则右孩子入队

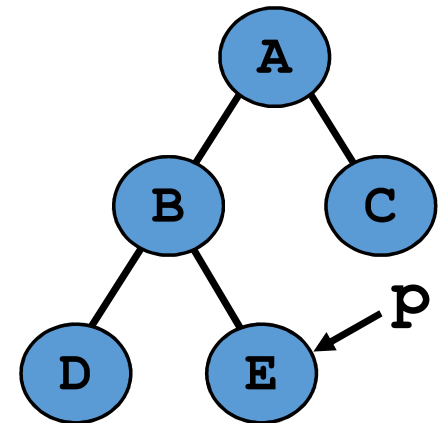
```
            AddQueue(Q, p->rchild);
```

```
    }
```

```
    return 0;
```

```
}
```

程序结束



A B C D E



```
int LevelOrderTraverse(BiTree T) {
```

```
    InitQueue(Q); AddQueue(Q, T);
```

树根入队

```
    while(!QueueEmpty(Q)) {
```

只要队列非空

```
        DeleteQueue(Q, p);
```

出队一个节点  
并访问之

```
        if(!Visit(p->data))
```

```
            return -1;
```

```
        if(p->lchild)
```

```
            AddQueue(Q, p->lchild);
```

如果p有左孩子  
则左孩子入队

```
        if(p->rchild)
```

```
            AddQueue(Q, p->rchild);
```

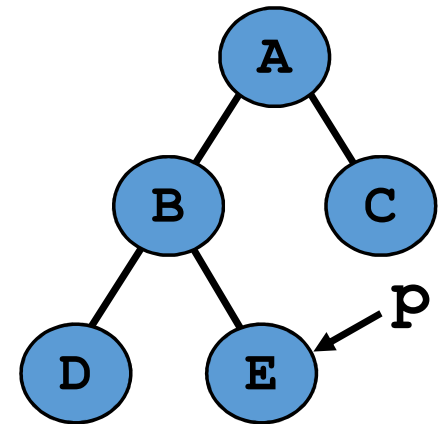
如果p有右孩子  
则右孩子入队

```
    }
```

```
    return 0;
```

```
}
```

程序结束



A B C D E

---

---

```

int LevelOrderTraverse(BiTree T) {
    InitQueue(Q);  AddQueue(Q, T);
    while(!QueueEmpty(Q)) {
        DeleteQueue(Q, p);
        if(!Visit(p->data))
            return -1;
        if(p->lchild)
            AddQueue(Q, p->lchild);
        if(p->rchild)
            AddQueue(Q, p->rchild);
    }
    return 0;
}

```

程序结束

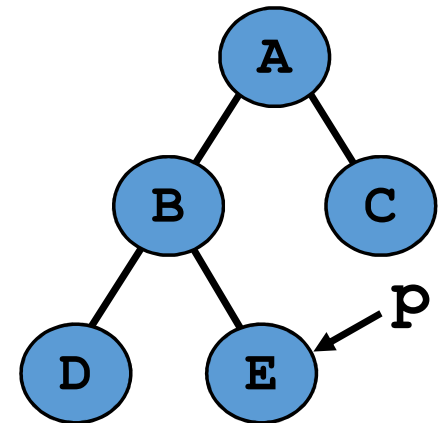
树根入队

只要队列非空

出队一个节点  
并访问之

如果p有左孩子  
则左孩子入队

如果p有右孩子  
则右孩子入队



A B C D E



# 基于遍历实现二叉树的其他操作

- 判断两棵二叉树是否相同
- 复制一棵二叉树
- 交换一棵二叉树的所有左右子树
- 计算一棵二叉树叶结点的个数
- 计算一棵二叉树的深度、宽度（每层结点数的最大值）
- 判断一棵二叉树是否是完全二叉树
- ...

# 基于遍历实现二叉树的其他操作

## 1. 求二叉树的深度(后序遍历)

```
int Depth (BiTNode* T ) {  
    if (!T) return 0;  
  
    int l = Depth( T->lchild );  
    int r = Depth( T->rchild );  
    return l>r?l+1:r+1;  
}
```

递归出口

左子树

右子树

根

# 基于遍历实现二叉树的其他操作

## 2. 求二叉树的度为1的结点数(后序遍历)

```
int Count(BiTNode* T) {  
    if (!T) return 0;  
  
    int l = Count( T->lchild );  
    int r = Count( T->rchild );  
    if( (T->lchild&&!T->rchild) ||  
        (!T->lchild&&T->rchild) )  
        return l+r+1;  
    else return l+r;  
}
```

递归出口

左子树

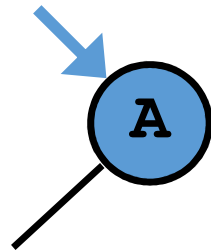
右子树

根

## 二叉树的其它操作

3. 按带空子树标记(#符号)的先序序列建立二叉链表.

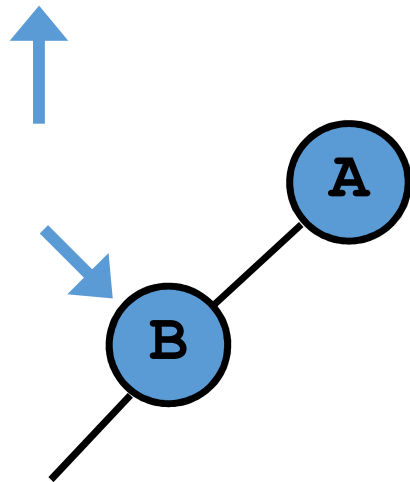
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记(#符号)的先序序列建立二叉链表.

A B D # G # # # C E # # F # #

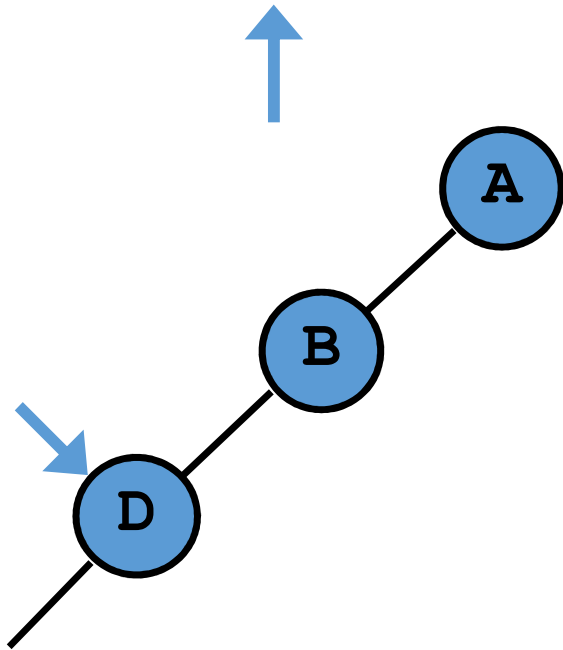




## 二叉树的其它操作

3. 按带空子树标记( # 符号)的先序序列建立二叉链表.

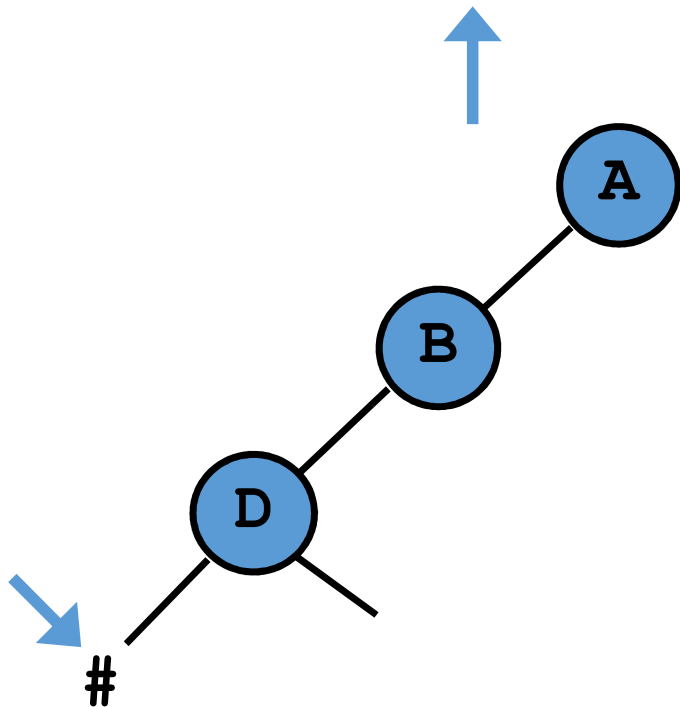
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记(#符号)的先序序列建立二叉链表.

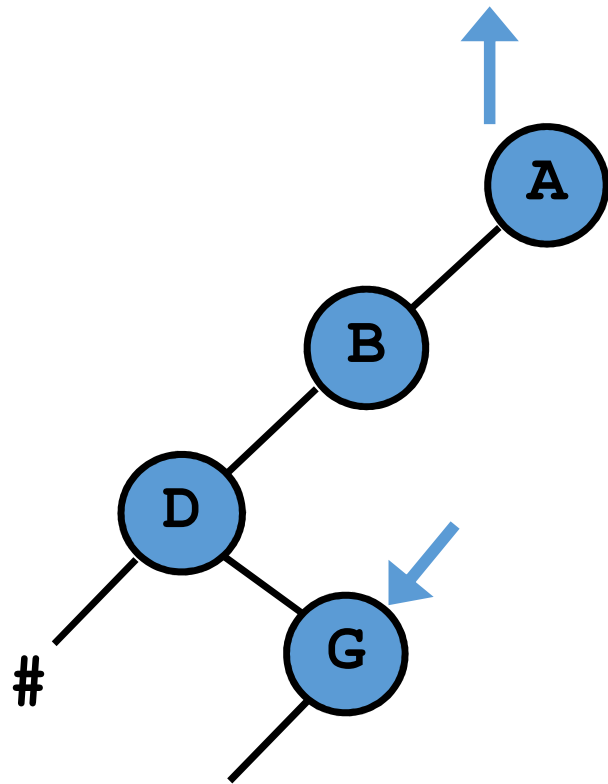
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记(#符号)的先序序列建立二叉链表.

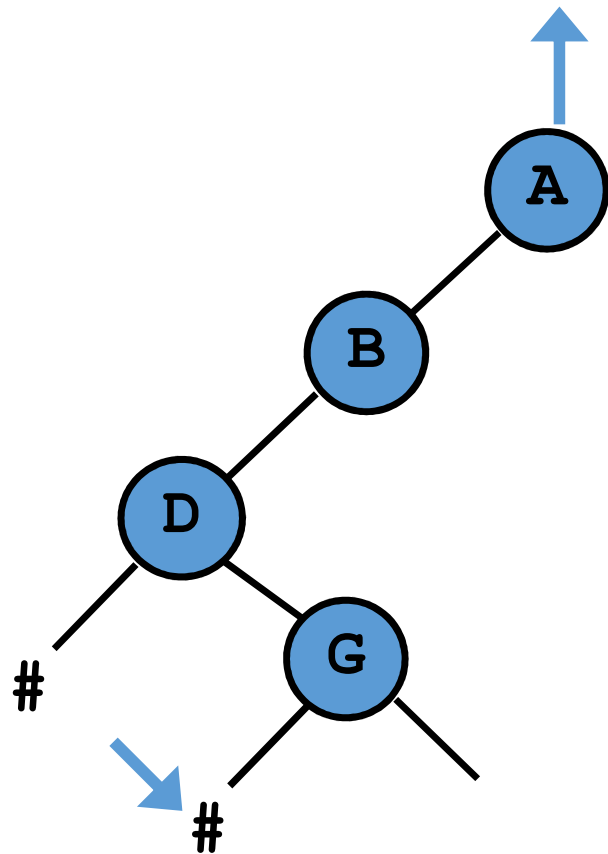
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记(#符号)的先序序列建立二叉链表.

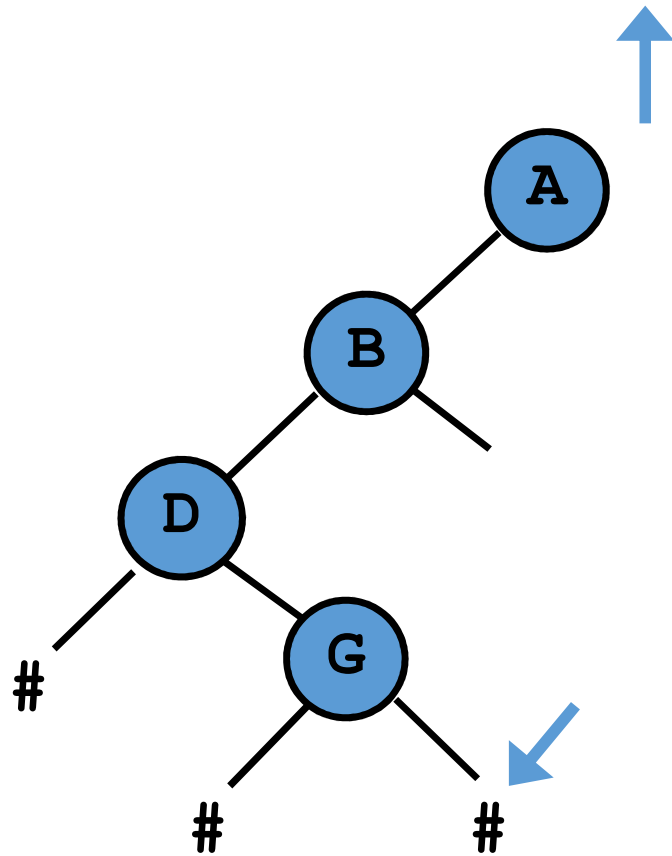
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记( # 符号)的先序序列建立二叉链表.

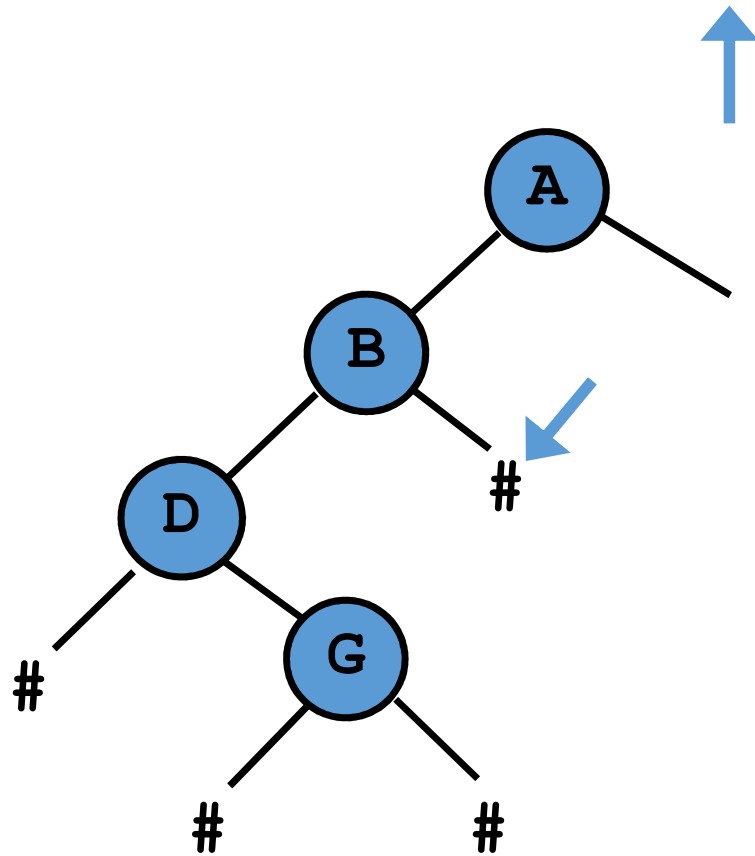
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记( # 符号)的先序序列建立二叉链表.

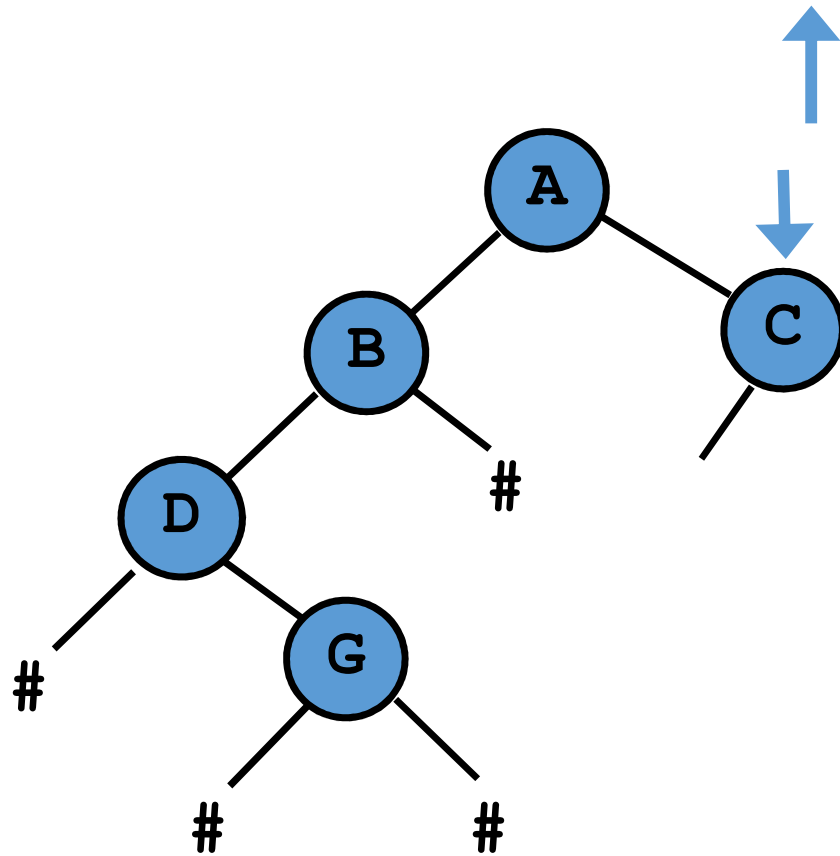
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记(#符号)的先序序列建立二叉链表.

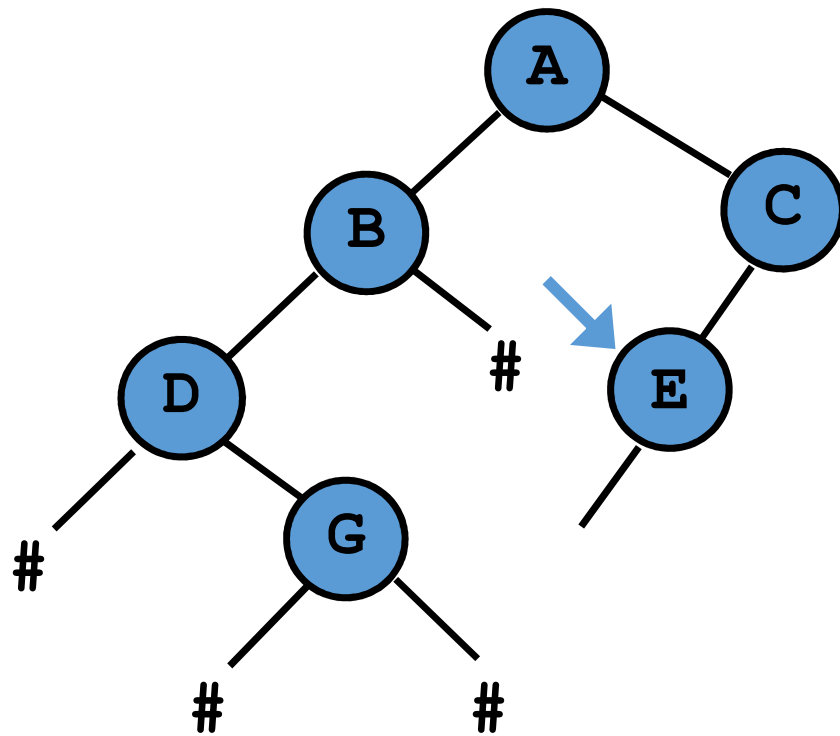
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记( # 符号)的先序序列建立二叉链表.

A B D # G # # # C E # # F # #

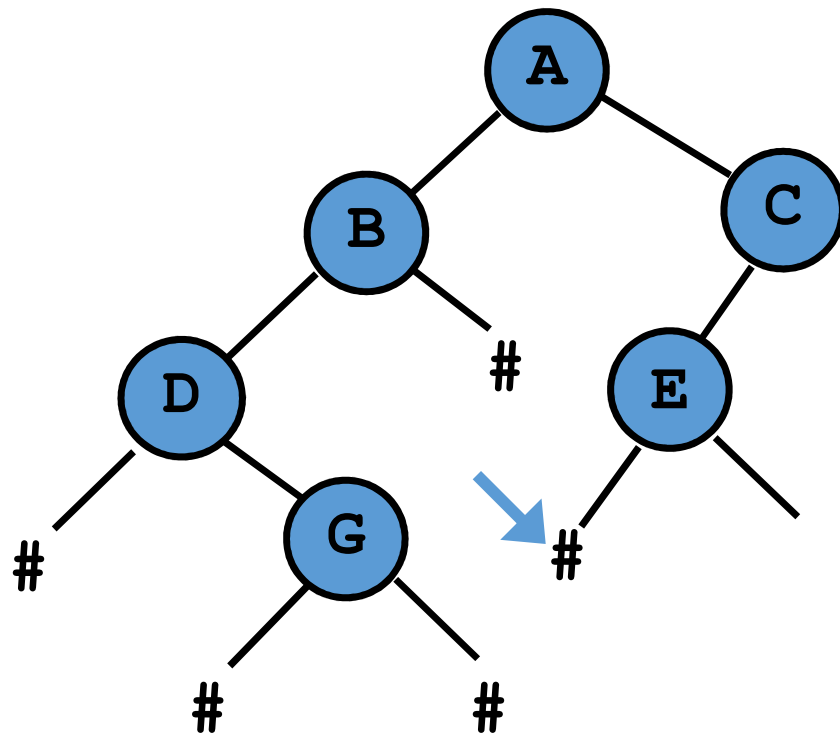




## 二叉树的其它操作

3. 按带空子树标记( # 符号)的先序序列建立二叉链表.

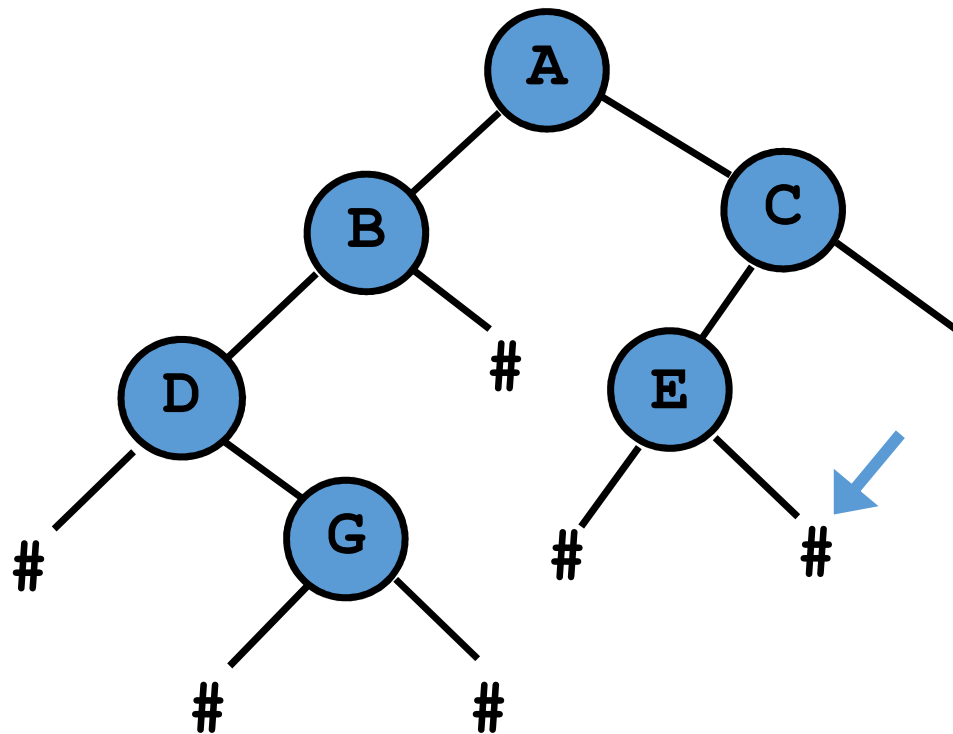
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记( # 符号)的先序序列建立二叉链表.

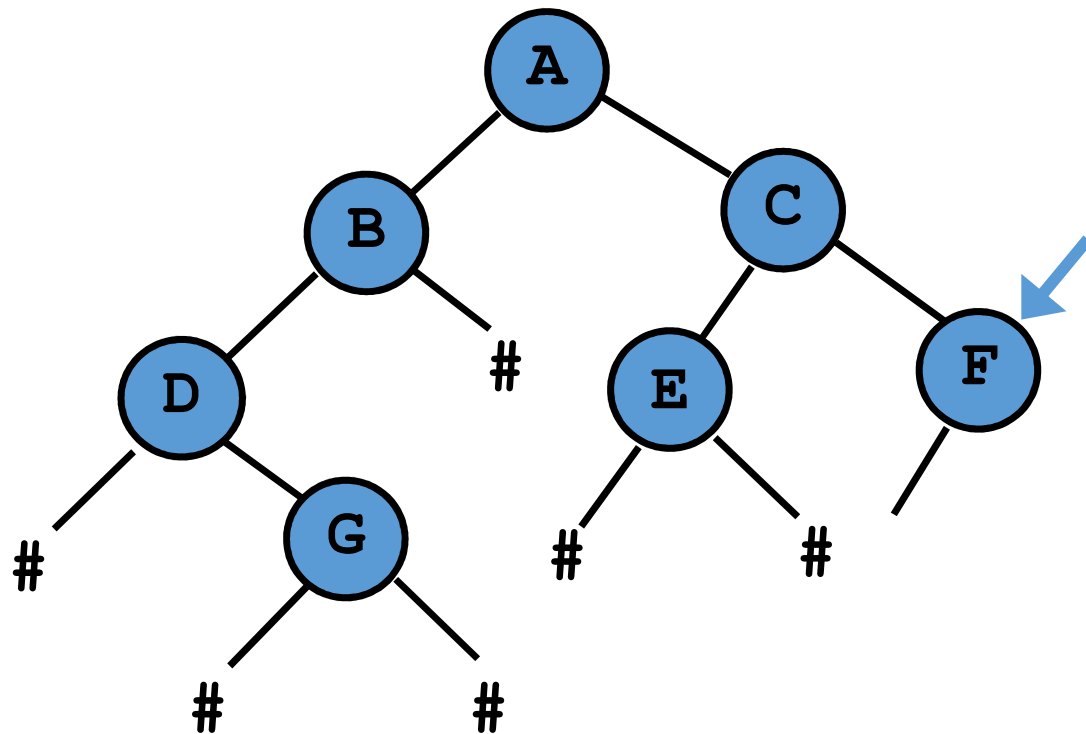
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记( # 符号)的先序序列建立二叉链表.

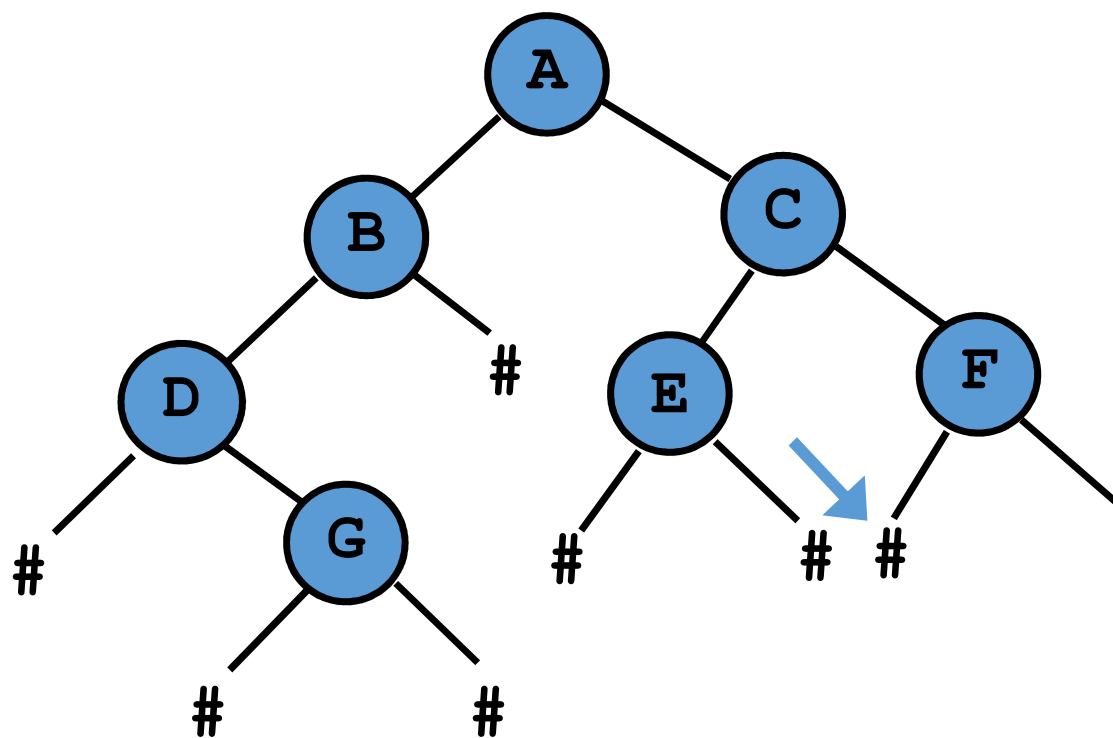
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记( # 符号)的先序序列建立二叉链表.

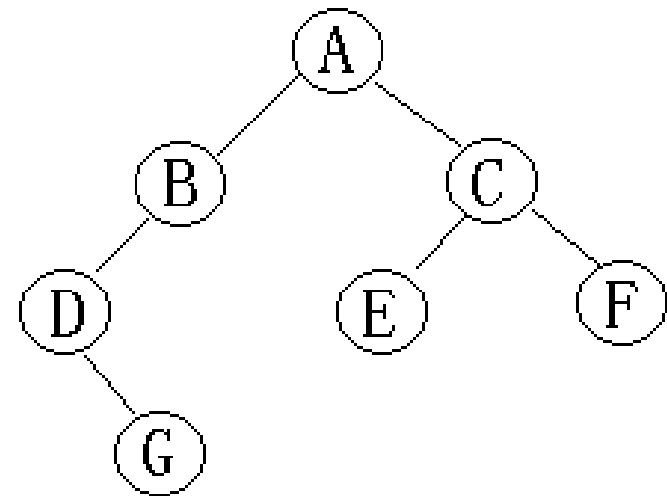
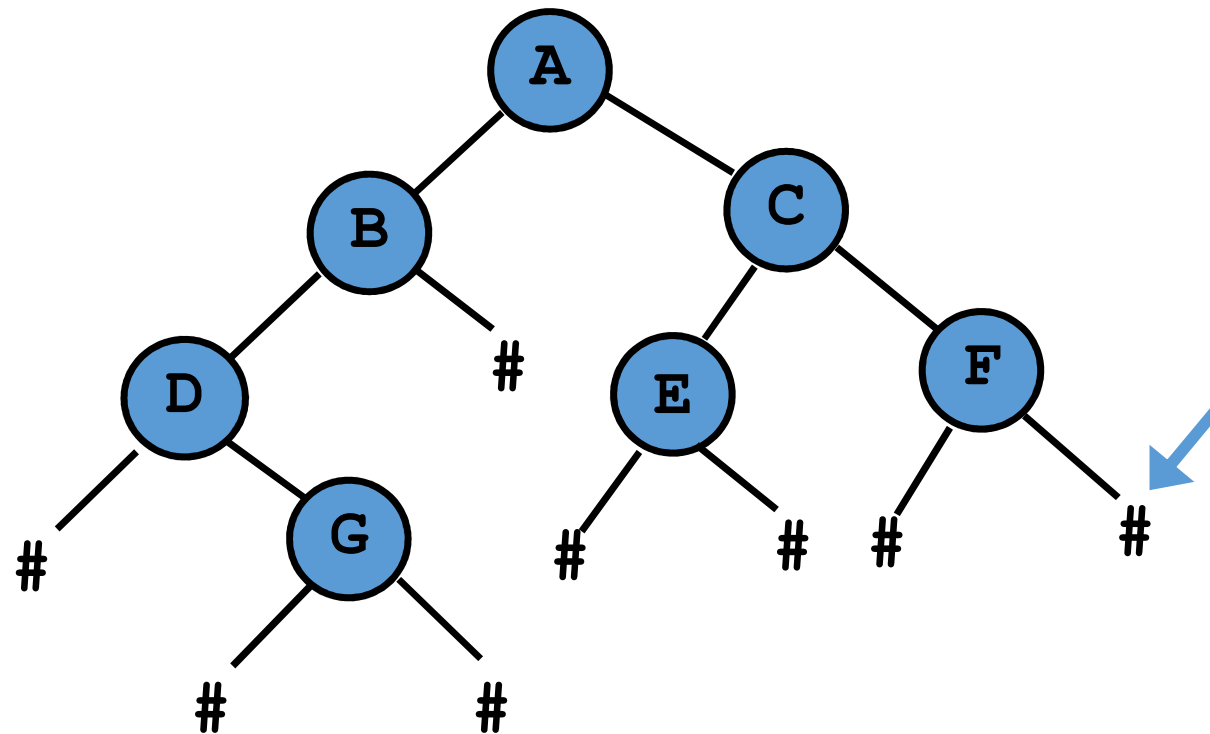
A B D # G # # # C E # # F # #



## 二叉树的其它操作

3. 按带空子树标记( # 符号)的先序序列建立二叉链表.

A B D # G # # # C E # # F # #

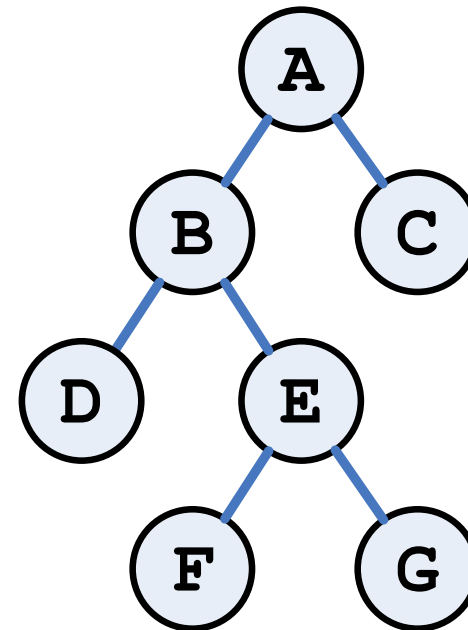


**A B D # G # # # C E # # F # #**  
↑

```
BiTNode* PreOrderCreatBiTree() {  
    char ch; BiTNode* T;  
    scanf("%c", &ch);  
    if (ch == '#') return 0;  
    else {  
        if (!(T = (BiTNode*)malloc(sizeof(BiTNode)))) {  
            printf("内存不够!\n"); return 0;  
        }  
        T->data = ch;  
        T->lchild = PreOrderCreatBiTree();  
        T->rchild = PreOrderCreatBiTree();  
        return T;  
    }  
}
```

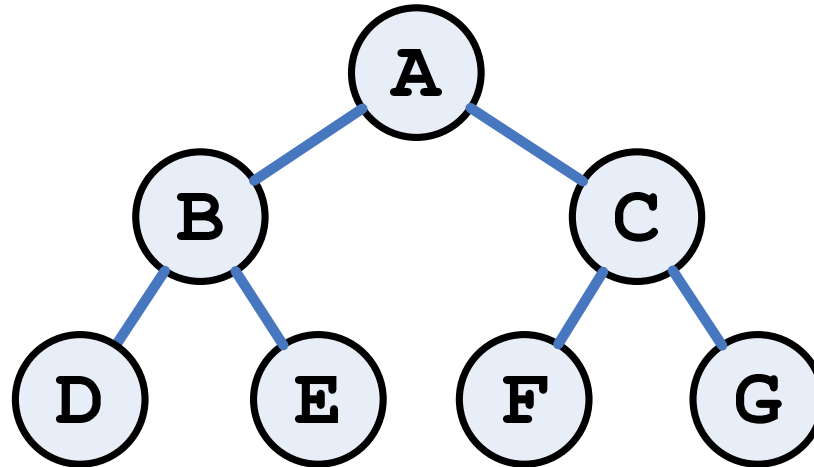
# 赫夫曼树及其应用

- 路径
  - 从一个结点到另一个结点的分支构成
- 路径长度
  - 路径上的分支的个数
- 树的路径长度
  - 从树根到每一个结点的路径长度之和



# 赫夫曼树及其应用

- 以下我们只讨论二叉树
- 问题：
  - 树的路径长度最短的是哪一种？
  - 完全二叉树

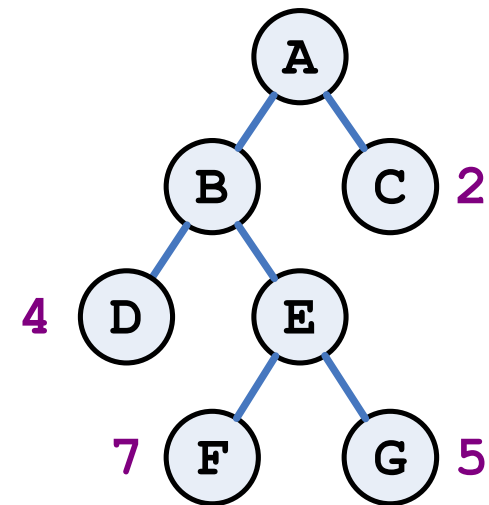




# 赫夫曼树及其应用

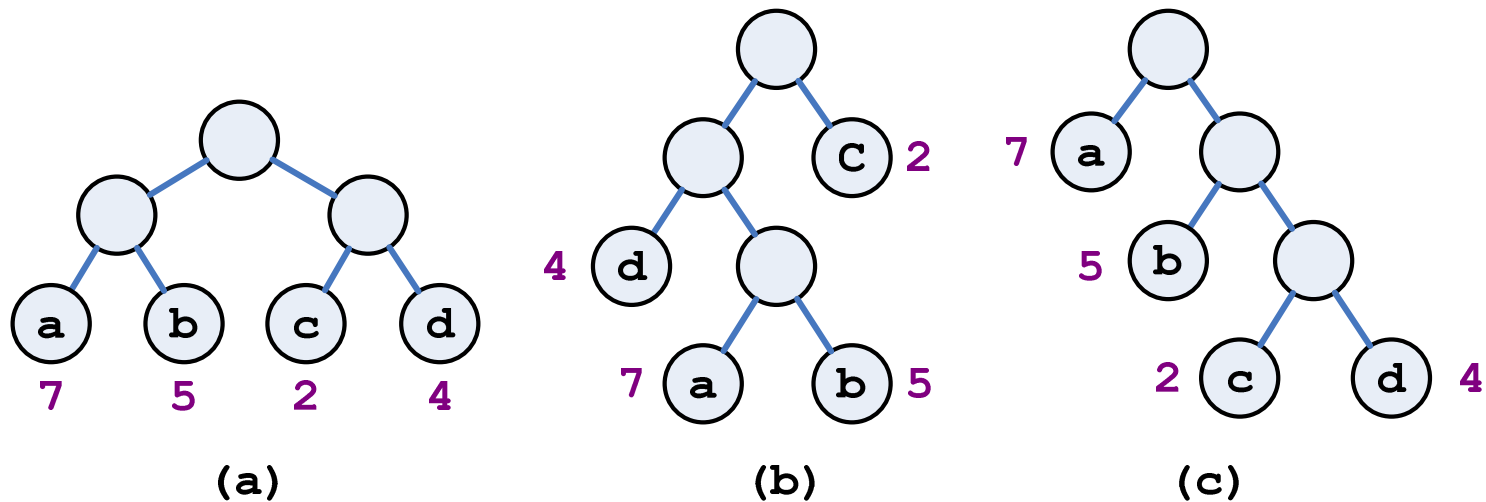
- 权(weight)
  - 给结点赋予一定的数值
- 结点的带权路径长度
  - 从树根到该节点的路径长度×该结点的权
- 树的带权路径长度
  - Weighted Path Length
  - 所有叶结点的带权路径长度之和

- $$WPL = \sum_{k=1}^n w_k l_k$$



# 赫夫曼树及其应用

• 例



$$WPL_a = 7 \times 2 + 5 \times 2 + 2 \times 2 + 4 \times 2 = 36$$

$$WPL_b = 7 \times 3 + 5 \times 3 + 2 \times 1 + 4 \times 2 = 46$$

$$WPL_c = 7 \times 1 + 5 \times 2 + 2 \times 3 + 4 \times 3 = 35$$

# 赫夫曼树及其应用

- 最优二叉树
  - 又称赫夫曼树(Huffman)
  - 给定n个叶子的权值权值 $\{w_1, w_2, \dots, w_n\}$ 
    - 如何构造一棵赫夫曼树?
    - 赫夫曼树有什么用?

# 赫夫曼树及其应用

- 例

- 一个判断成绩等级的程序

if(a < 60) b = “不及格”

else if(a < 70) b = “及格”

    else if(a < 80) b = “中等”

        else if(a < 90) b = “良好”

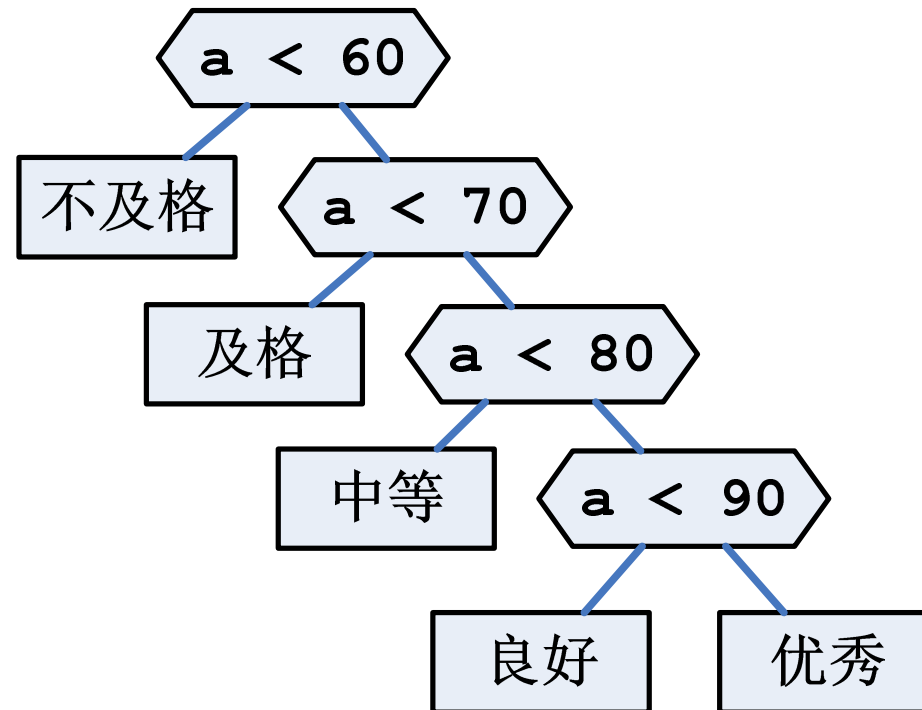
            else b = “优秀”

- 一个a最多要经过4次比较才能得出b
- 我们当然希望比较的总次数最小

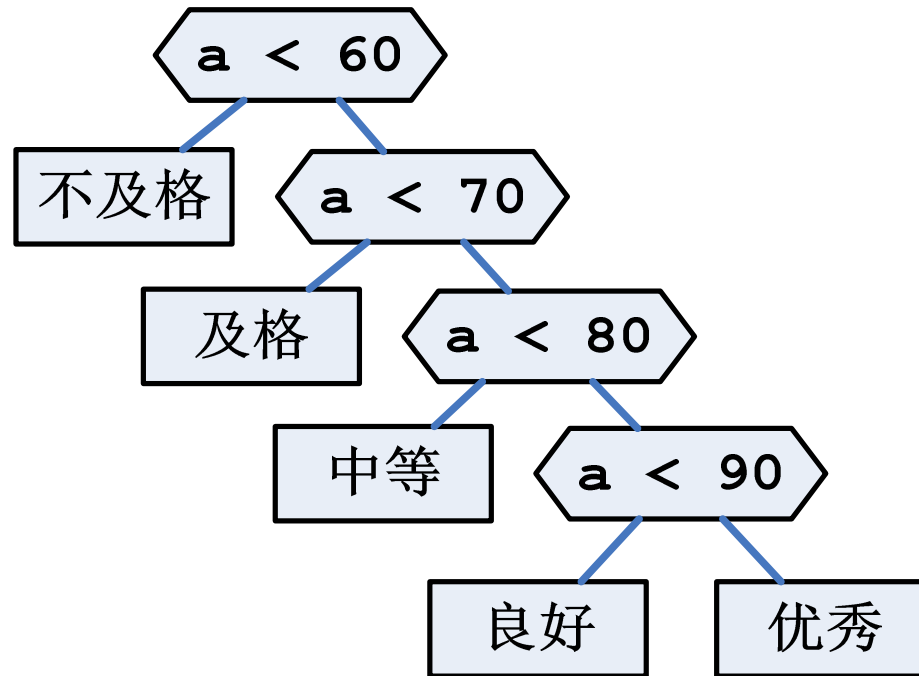
- 假设有如下统计数据

分数	0-59	60-69	70-80	80-89	90-100
概率	0.05	0.15	0.40	0.30	0.10

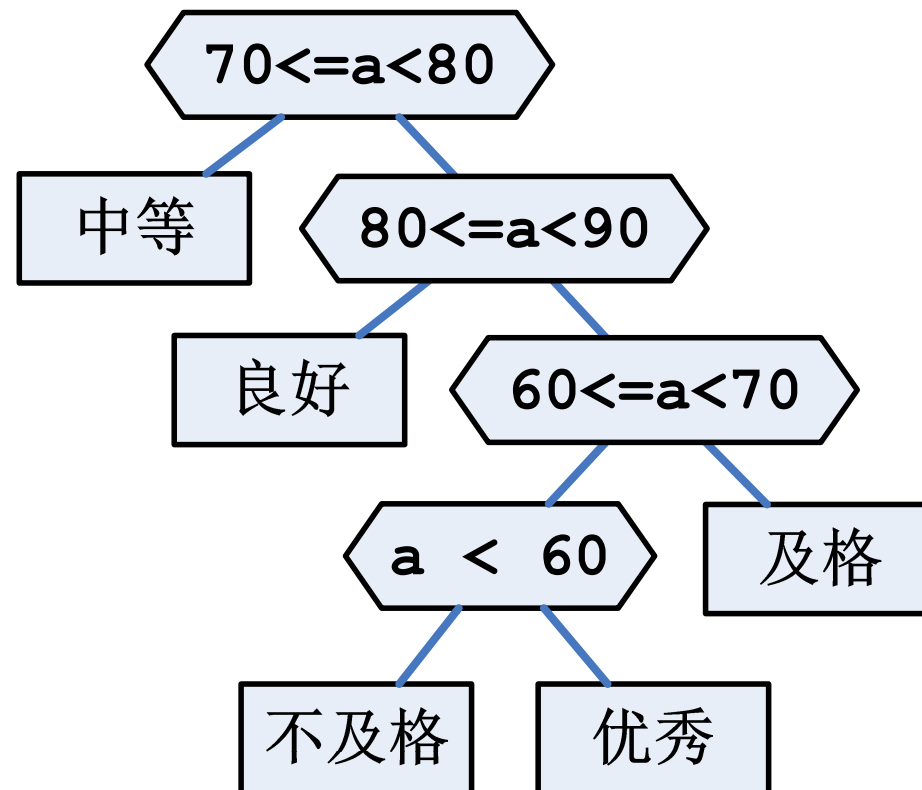
- 原判定树为：



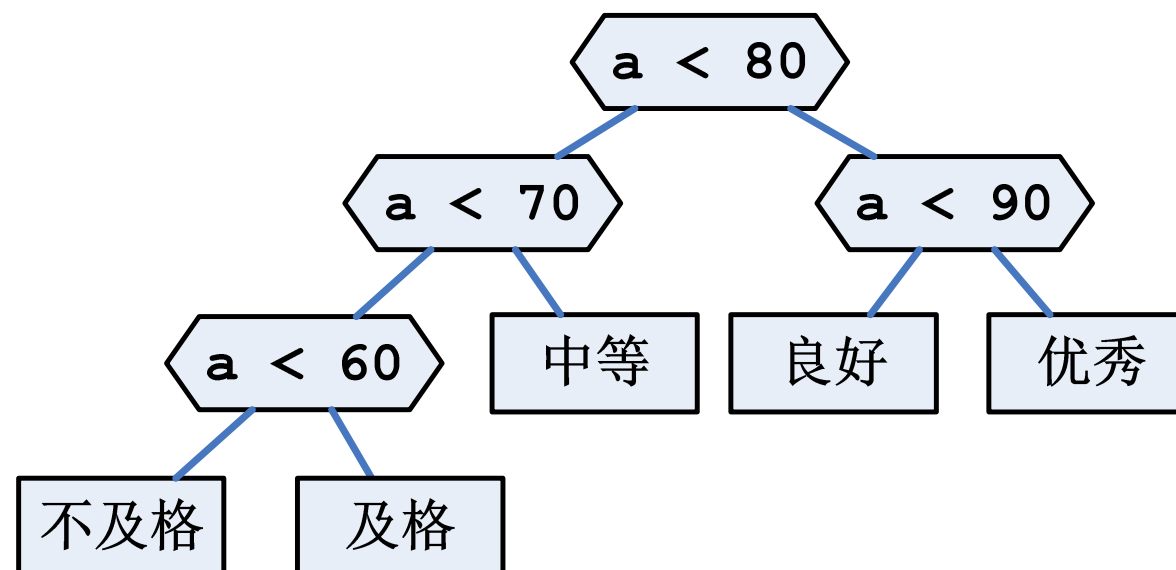
- 若一共有10000个输入数据
- 则总共的比较次数  
$$= 500*1 + 1500*2 + 4000*3 + 3000*4 + 1000*4$$
$$= 31500$$



- 构造一棵赫夫曼树
  - 有5个叶结点，权值分别为：  
0.05, 0.15, 0.4, 0.3, 0.1



- 根据这棵赫夫曼树导出新的判定树：



- 总的比较次数

$$\begin{aligned} &= 500*3 + 1500*3 + 4000*2 + 3000*2 + \\ &1000*2 \\ &= 22000 \end{aligned}$$



# 赫夫曼树及其应用

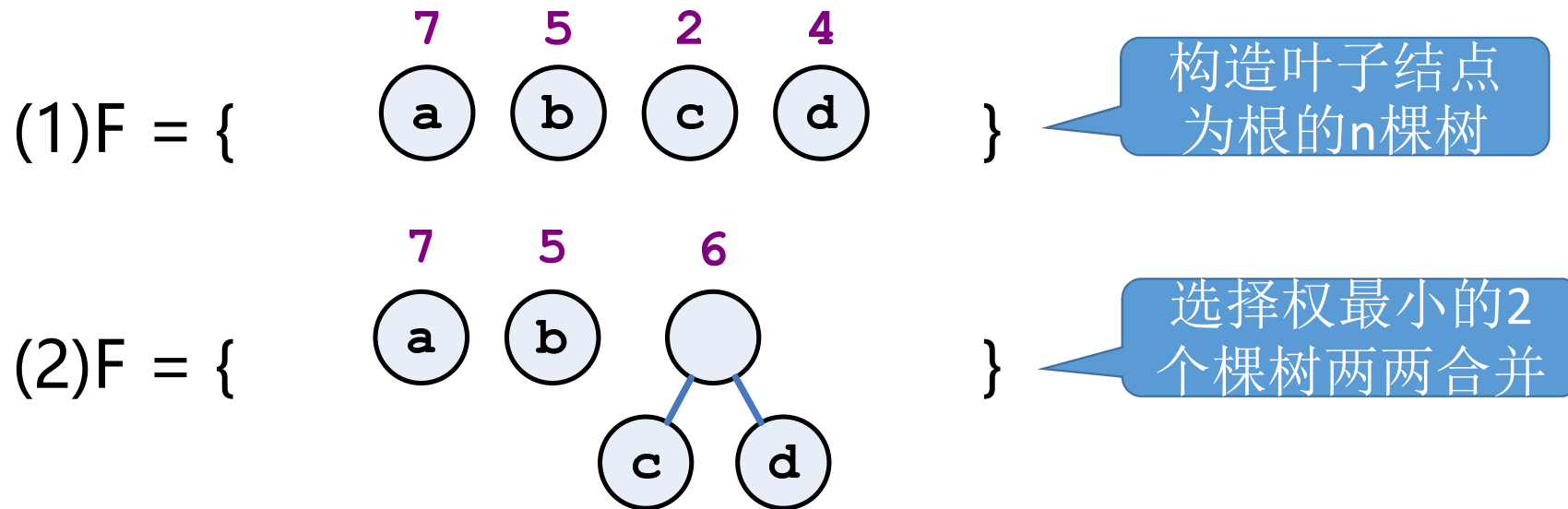
- 赫夫曼树的构造算法

- (1)根据给定的 $n$ 个权值 $\{w_1, w_2, \dots, w_n\}$ 构成 $n$ 棵二叉树的集合  $F = \{T_1, T_2, \dots, T_n\}$ ，其中每棵二叉树只含一个带权的根结点，其左右子树均空
- (2)在 $F$ 中选取两棵根结点的权值最小的树作为左右子树，构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树根结点的权值之和
- (3)在 $F$ 中删除这两棵二叉树，同时将新得到的二叉树加入 $F$
- (4)重复(2)和(3)，直到 $F$ 只含一棵二叉树

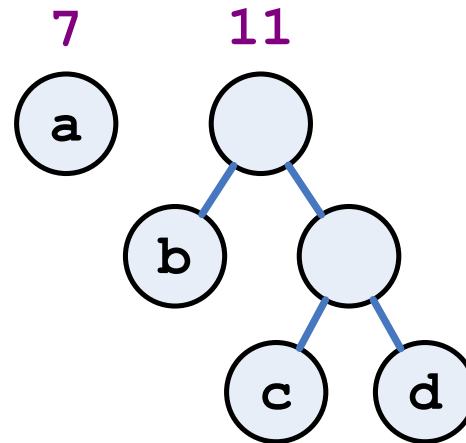
# 赫夫曼树及其应用

• 例:

- 设结点a,b,c,d的权值分别为7,5,2,4, 试构造赫夫曼树



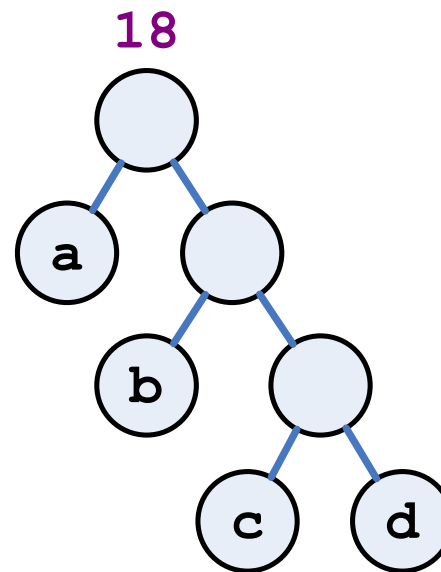
(3)F = {



}

选择权最小的2  
个棵树两两合并

(4)F = {



}

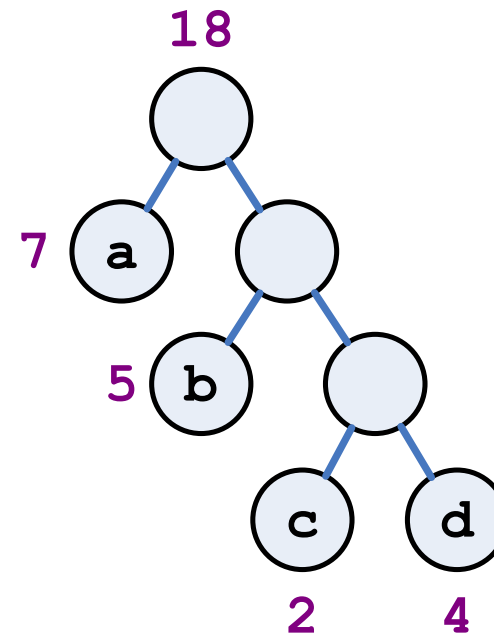
选择权最小的2  
个棵树两两合并

# 赫夫曼树及其应用

- 理解

- 为什么赫夫曼树的带权路径长度最小?
- 它把权值小的结点放在底层
- 权值大的结点放在上层

$$\sum_{k=1}^n w_k l_k$$



# 赫夫曼树的应用——赫夫曼编码

- 例

- 某系统在通信联络中只可能出现八种字符(A,B,C,D,E,F,G,H)，其使用概率分别为0.05、0.29、0.07、0.08、0.14、0.23、0.03、0.11，如何设计这些字符的二进制编码，以使通信中总码长尽可能短？

- 方案一：固定长度编码

- 8个字符，只需要3位二进制数就能表示
- 比如000代表A，001代表B，...111代表H
- 这样平均每个字符用3位二进制数表示

# 赫夫曼树的应用——赫夫曼编码

- 方案二：赫夫曼编码

A	B	C	D	E	F	G	H
0.05	0.29	0.07	0.08	0.14	0.23	0.03	0.11
0110	10	1110	1111	110	00	0111	010

- 平均每个字符的编码长度
$$\begin{aligned} &= 4 \times 0.05 + 2 \times 0.29 + 4 \times 0.07 + \\ &\quad 4 \times 0.08 + 3 \times 0.14 + 2 \times 0.23 + \\ &\quad 4 \times 0.03 + 3 \times 0.11 \\ &= 2.71 \end{aligned}$$

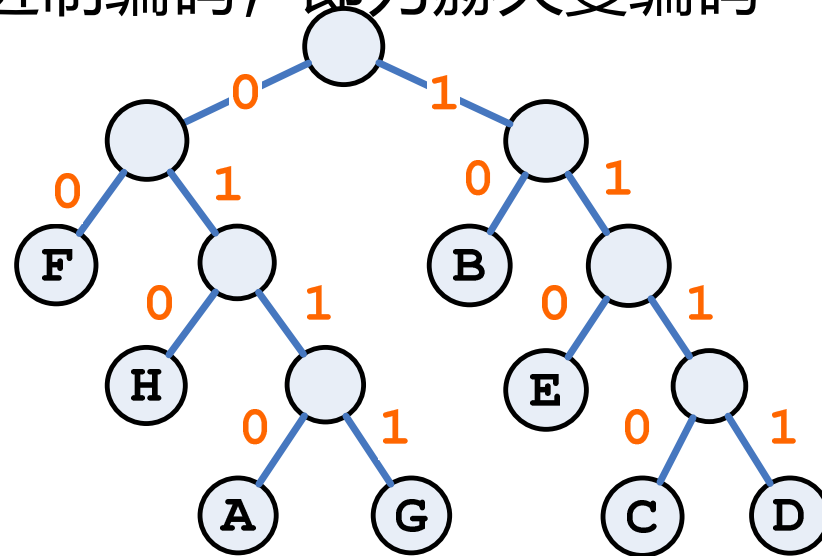
# 赫夫曼树的应用——赫夫曼编码

- 赫夫曼编码的方法

- 以字符出现频率为权值，构造赫夫曼树
- 左分支表示0，右分支表示1，把从根到叶子的路径上所有分支构成的0,1作为叶子的二进制编码，即为赫夫曼编码

- 比如

- A: 0110
- B: 10
- C: 1110
- ...

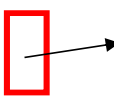


# 赫夫曼树的应用——赫夫曼编码

```
typedef struct{  
    double weight;  
    unsigned int p,lc,rc;  
}HTNode;  
typedef struct{  
    HTNode *data;  
    unsigned int n;  
}HuffmanTree;  
typedef char * HuffmanCode;
```



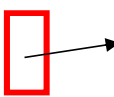
# 赫夫曼树的应用——赫夫曼编码

HT	weight	p	lc	rc	
					0
	5	0	0	0	1
	29	0	0	0	2
	7	0	0	0	3
	8	0	0	0	4
	14	0	0	0	5
	23	0	0	0	6
	3	0	0	0	7
	11	0	0	0	8
					9
					10
					11
					12
					13
					14
					15

(1)为赫夫曼树的存储开辟空间，含  $n$  个叶子结点的赫夫曼树共 $2n-1$  个结点；

(2)生成  $n$  个叶子结点，使它们的权值为给定的  $n$  个权值，双亲和左右孩子均为0 ；

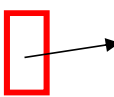
# 赫夫曼树的应用——赫夫曼编码

HT	weight	p	lc	rc	
					0
	5	9	0	0	1
	29	0	0	0	2
	7	0	0	0	3
	8	0	0	0	4
	14	0	0	0	5
	23	0	0	0	6
	3	9	0	0	7
	11	0	0	0	8
	8	0	1	7	9
					10
					11
					12
					13
					14
					15

(3)在双亲为 0 的结点中选两个权值最小的，生成以这两个结点为左、右孩子的分支结点。

(4)重复(3)，直至生成  $n-1$  个分支结点。

# 赫夫曼树的应用——赫夫曼编码

HT	weight	p	lc	rc	
					0
	5	9	0	0	1
	29	0	0	0	2
	7	10	0	0	3
	8	10	0	0	4
	14	0	0	0	5
	23	0	0	0	6
	3	9	0	0	7
	11	0	0	0	8
	8	0	1	7	9
	15	0	3	4	10
					11
					12
					13
					14
					15

(3)在双亲为 0 的结点中选两个权值最小的，生成以这两个结点为左、右孩子的分支结点。

(4)重复(3)，直至生成  $n-1$  个分支结点。

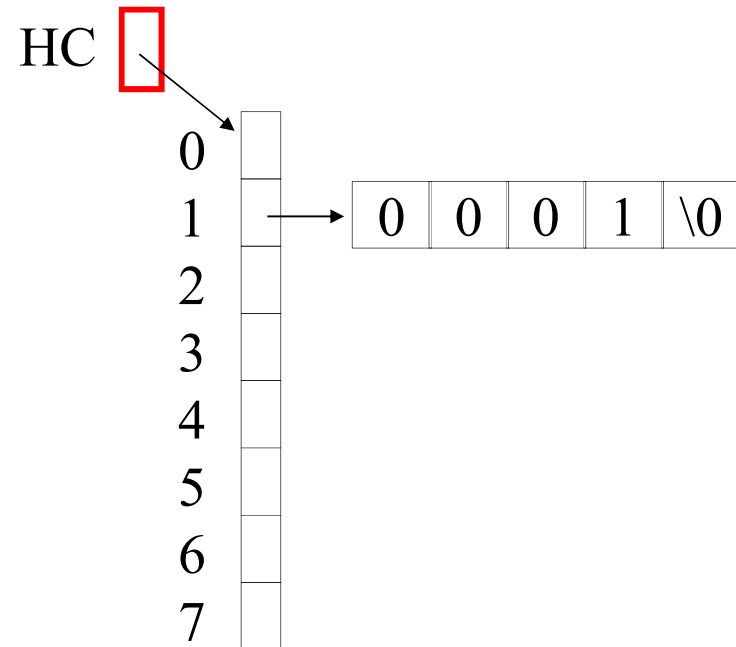
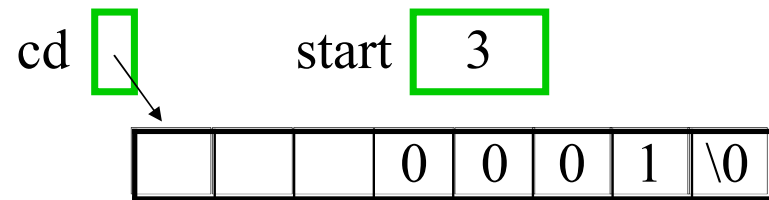
# 赫夫曼树的应用——赫夫曼编码

- 求出编码

对赫夫曼树中的每个叶子结点，求从根到它的路径：

- (1) 开辟  $n$  个存储空间，用以保存分别指向  $n$  个编码串的指针变量；
- (2) 用临时空间保存当前所求编码串
- (3) 对每个叶子结点，从该结点开始向根逆向求编码。

HT	weight	p	lc	rc	
<div></div>					0
	5	9	0	0	1
	29	14	0	0	2
	7	10	0	0	3
	8	10	0	0	4
	14	12	0	0	5
	23	13	0	0	6
	3	9	0	0	7
	11	11	0	0	8
	8	11	7	1	9
	15	12	3	4	10
	19	13	9	8	11
	29	14	5	10	12
	42	15	11	6	13
	58	15	2	12	14
	100	0	13	14	15



博客：<https://xuepro.xcguan.net>

微博： 教小白精通编程

QQ群： 101132160