第六章 树和二叉树

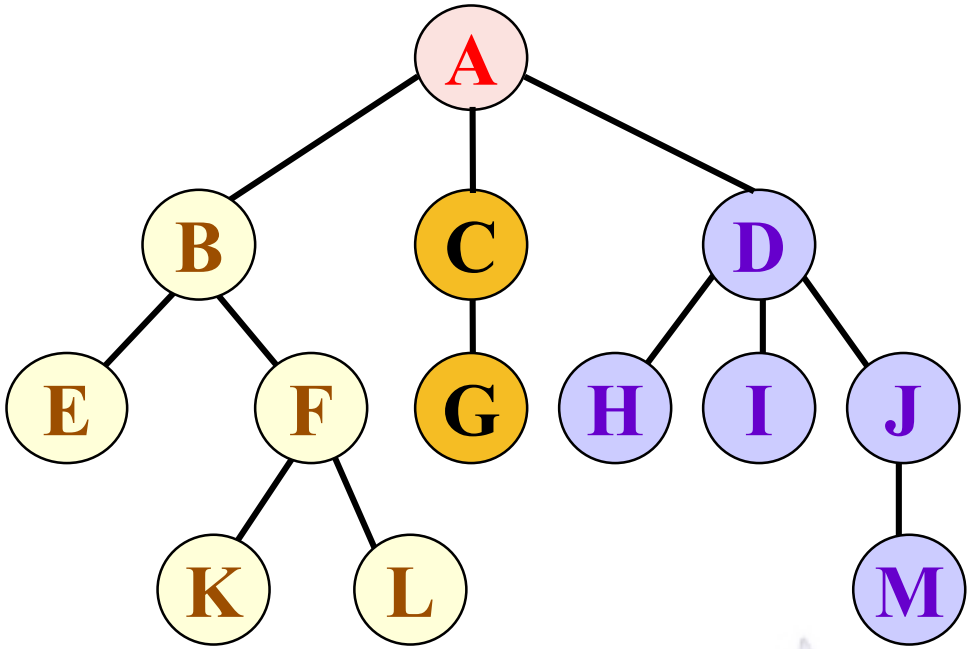
高春晓

本章内容

- 5.1 树的类型定义
- 5.2 二叉树
- 5.3 二叉树的存储结构
- 5.4 二叉树的遍历
- 5.5 线索二叉树
- 5.6 树和森林的表示方法
- 5.7 树和森林的遍历
- 5.8 哈夫曼树与哈夫曼编码

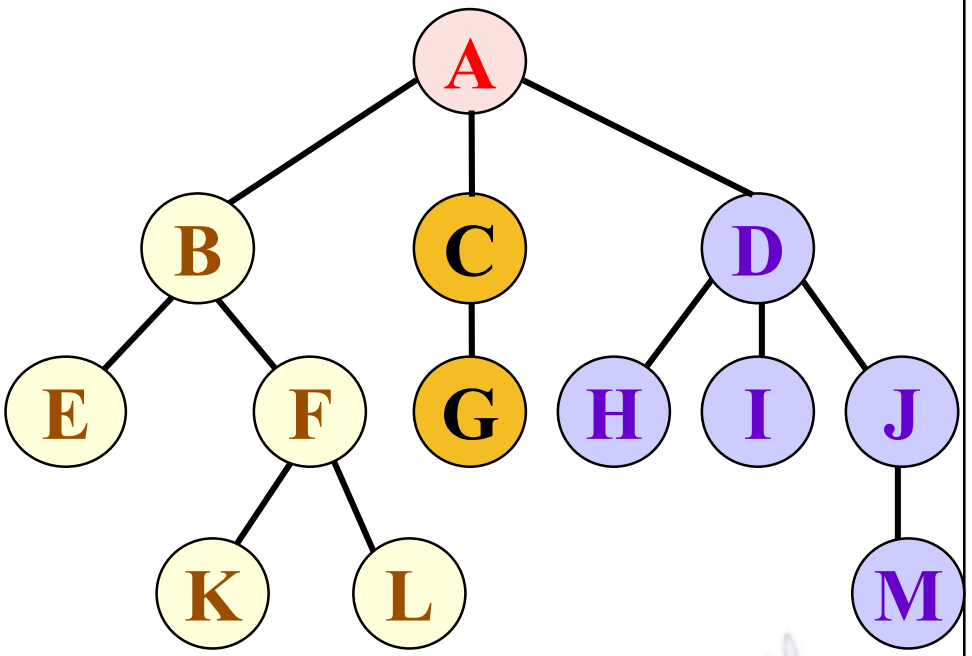
5.1 树的类型定义

- 结点:数据元素+若干指向子树的分支
- 结点的度:分支的个数
- 树的度:树中所有结点的度的最大值
- 叶子结点:度为零的结点
- 分支结点:度大于零的结点

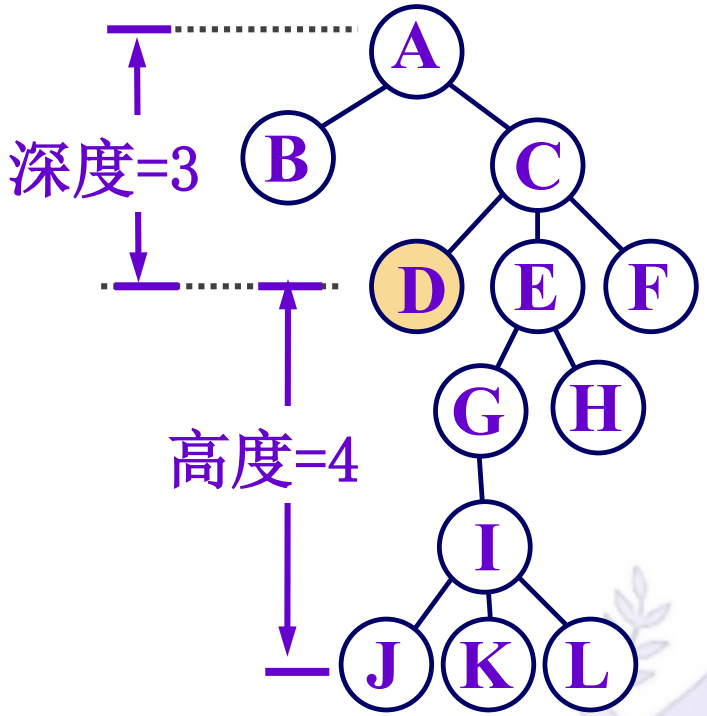


5.1 树的类型定义

- 路径：由从根到该结点所经分支和结点构成。
- 结点间的路径（path）：树中任一结点v_i经过一系列结点$v_1, v_2, \ldots, v_k$到v_j，其中$(v_i, v_1), (v_1, v_2), \ldots, (v_k, v_j)$是树中的分支。
- 孩子结点、双亲结点
- 兄弟结点、堂兄弟
- 祖先结点、子孙结点

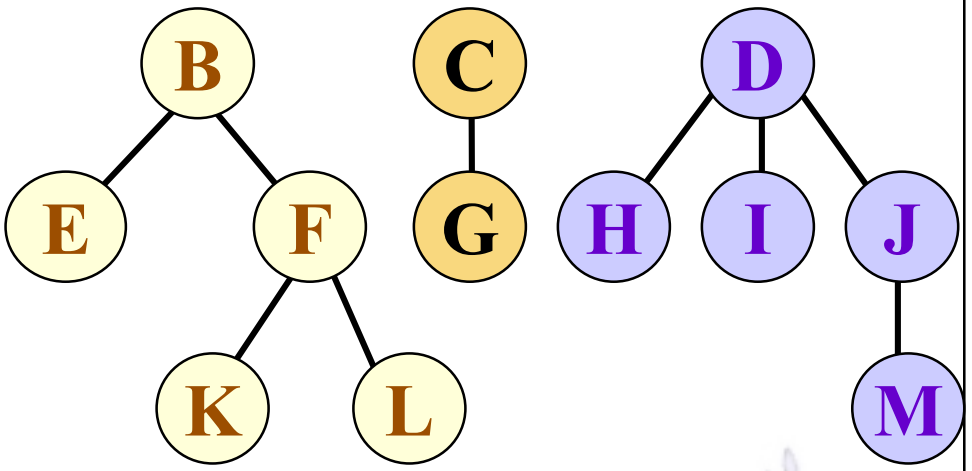


- **结点的层次**：假设根结点的层次为1，第 l 层的结点的子树根结点的层次为 $l+1$.
- **结点的高度**：从下向上逐层计算的：叶结点的高度为1，其他结点的高度是取它的所有子女结点最大高度加一。
- **树的深度**：树中叶子结点所在的最大层次.
- **树的高度**：根结点的高度。与树的深度相等。

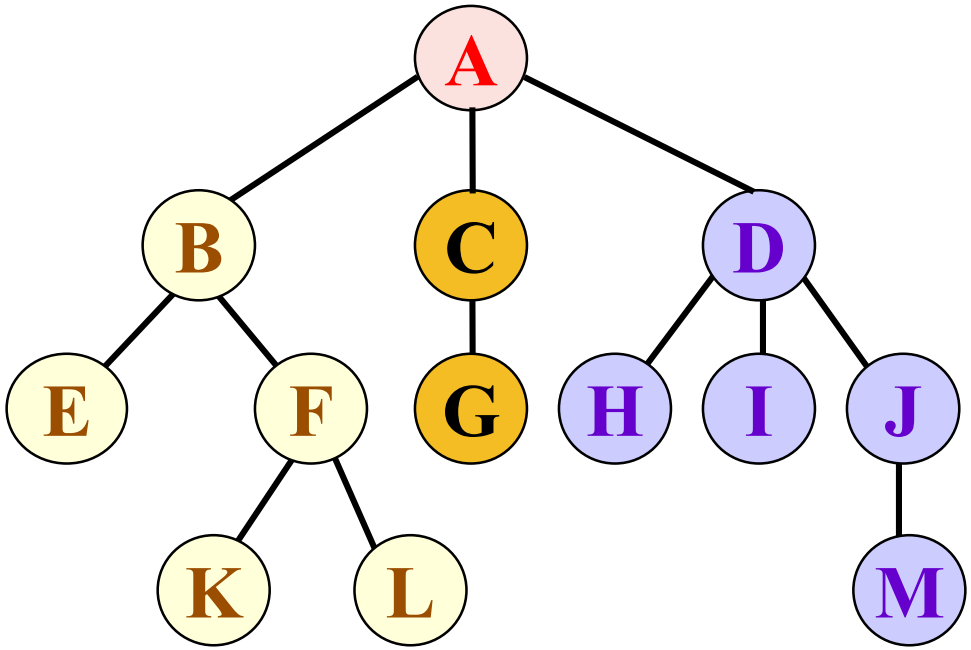


5.1 树的类型定义

- **森林：** m（m≥0）棵互不相交的树的集合
- **树的定义：** 任何一棵非空树是一个二元组
  - Tree=(root，F)
  - root 被称为根结点
  - F 被称为子树森林



5.1 树的类型定义



A(B(E, F(K, L)), C(G), D(H, I, J(M)))

树根

T_1 T_2 T_3

5.1 树的类型定义: ADT Tree

- {数据对象 D：

  D是具有相同特性的数据元素的集合。

- 数据关系 R：

  - 若D为空集，则称为空树 。否则:
  - (1) 在D中存在唯一的称为根的数据元素root;
  - (2) 当n>1时，其余结点可分为m (m>0)个互不相交的有限集T1, T2, …, Tm，每个子集与结点root之间满足特定关系
  - (3) 每一子集本身又是一棵符合本定义的树，称为根root的子树。

- 基本操作：……}

基本操作：

- 结构类
  - InitTree(&T) // 初始化置空树
  - CreateTree(&T, definition) // 按定义构造树
  - ClearTree(&T) // 将树清空
  - DestroyTree(&T) // 销毁树的结构
- 属性类
  - TreeEmpty(T) // 判定树是否为空树
  - TreeDepth(T) // 求树的深度

基本操作：

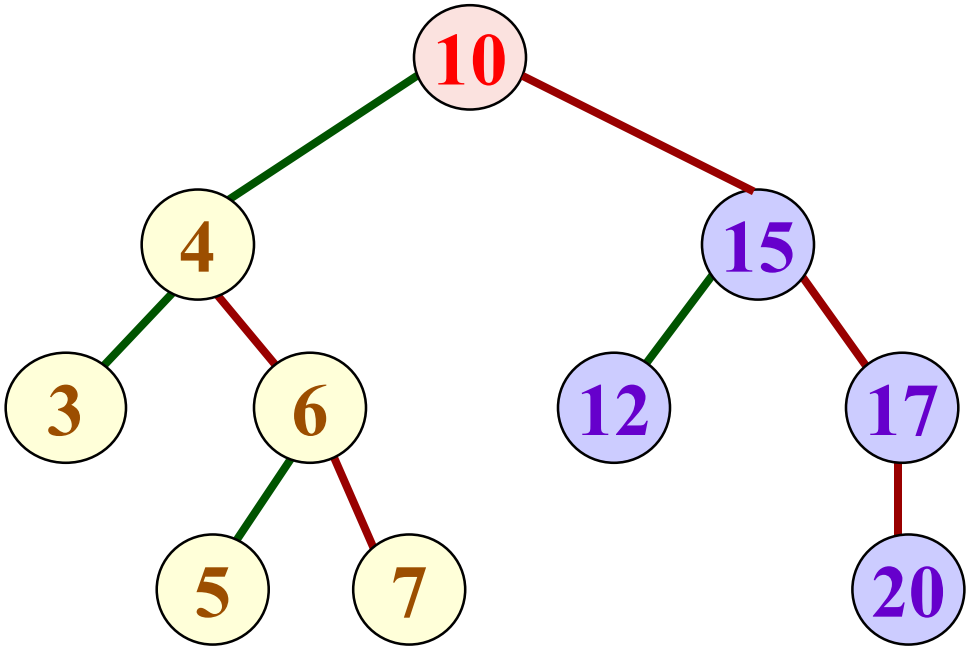
- 访问类
  - Root(T) // 求树的根结点
  - Value(T, cur_e) // 求当前结点的元素值
  - Parent(T, cur_e) // 求当前结点的双亲结点
  - LeftChild(T, cur_e) // 求当前结点的最左孩子
  - RightSibling(T, cur_e) // 求当前结点的右兄弟
  - TraverseTree(T, Visit()) // 遍历

基本操作：

- 修改类
  - Assign(T, cur_e, value) // 给当前结点赋值
  - InsertChild(&T, &p, i, c)
  - // 将以c为根的树插入为结点p的第i棵子树
  - DeleteChild(&T, &p, i) // 删除结点p的第i棵子树

5.1 树的类型定义

- 有序树：
  - 子树之间存在确定的次序关系。
- 无序树：
  - 子树之间不存在确定的次序关系。



对比树型结构和线性结构的结构特点

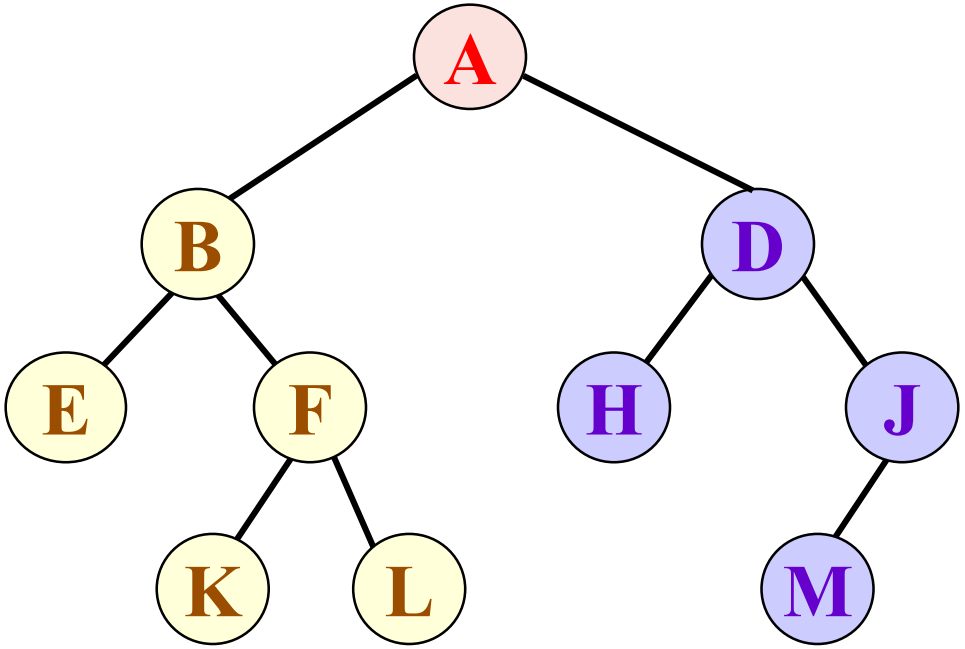
	线性结构	树型结构
第一个数据元素	无前驱	无前驱(根结点)
最后数据元素	唯一， 无后继	多个叶子结点 无后继
其它数据元素	一个前驱 一个后继	一个前驱 多个后继

树的应用

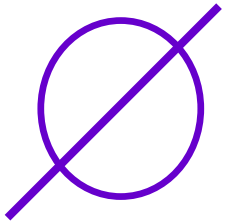
- 域名服务DNS
- 目录服务：NDS、Active Directory、LDAP
- XML DOM Parser
- 人工智能：决策树
- 自然语言理解：字典树、语法树
- 计算机图形学：二分树、八叉树
- ……

5.2 二叉树的定义及特性

- **5.2.1 二叉树的类型定义**
- 二叉树或为空树，或是由一个根结点加上两棵分别称为左子树和右子树的、互不交的二叉树组成。

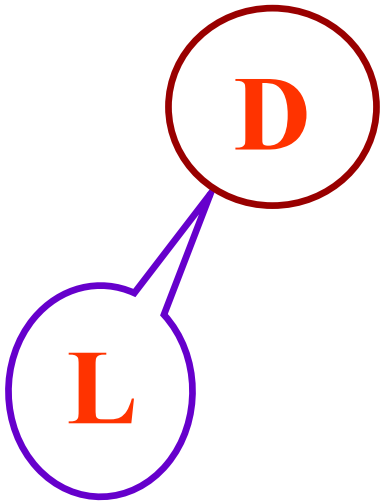


二叉树的5种基本形态

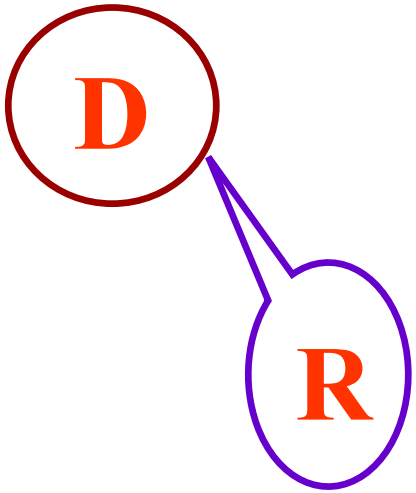


1) 空树

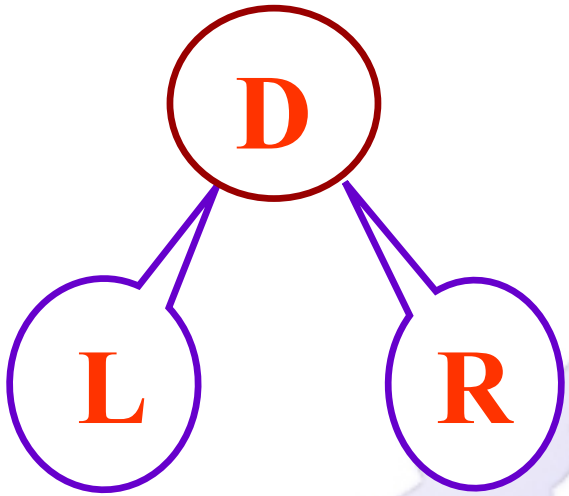
2) 只含根结点



3) 右子树为空树



4) 左子树为空树



5) 左右均不为空树

5.2.2 二叉树的性质－1

- 二叉树的主要基本操作：与树类似
- 性质1：二叉树的第 i 层上至多有 2^{i-1} 个结点。($i \geq 1$)

用归纳法证明：

归纳基：$i = 1$ 层时，只有一个根结点：

$$2^{i-1} = 2^0 = 1;$$

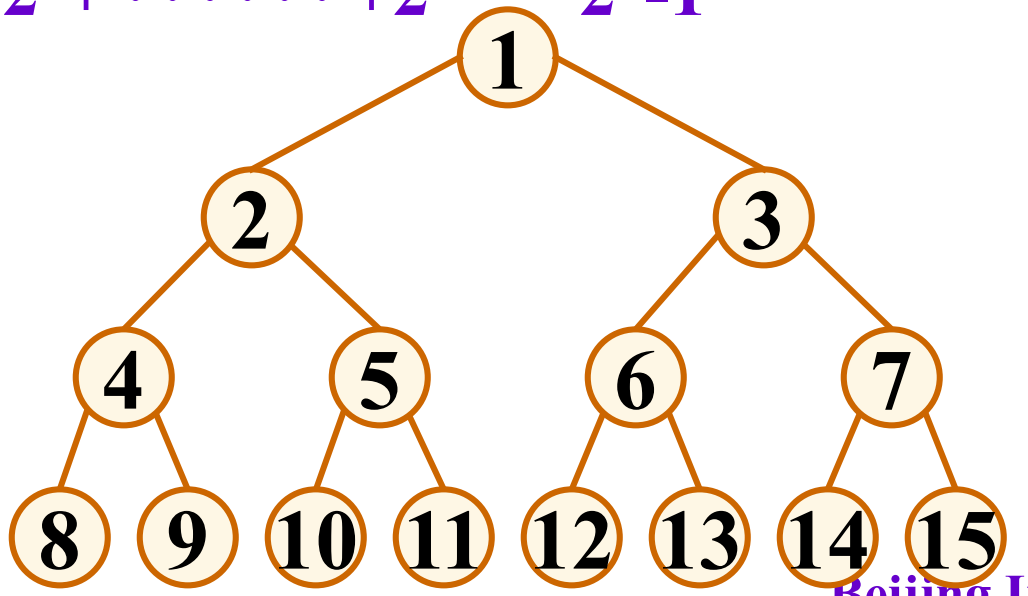
归纳假设：假设对所有的 j，$1 \leq j < i$，命题成立；

归纳证明：二叉树上每个结点至多有两棵子树，

则第 i 层的结点数 $= 2^{i-2} \times 2 = 2^{i-1}$ 。

5.2.2 二叉树的性质－ 2

- 性质 2 ： 高度为 h 的二叉树上至多含 2^h-1 个结点 ($k \geqslant 1$)。
- 证明：基于上一条性质，深度为 h 的二叉树上的结点数至多为
  - ¶ $2^0+2^1+\cdots\cdots+2^{h-1}=2^h-1$

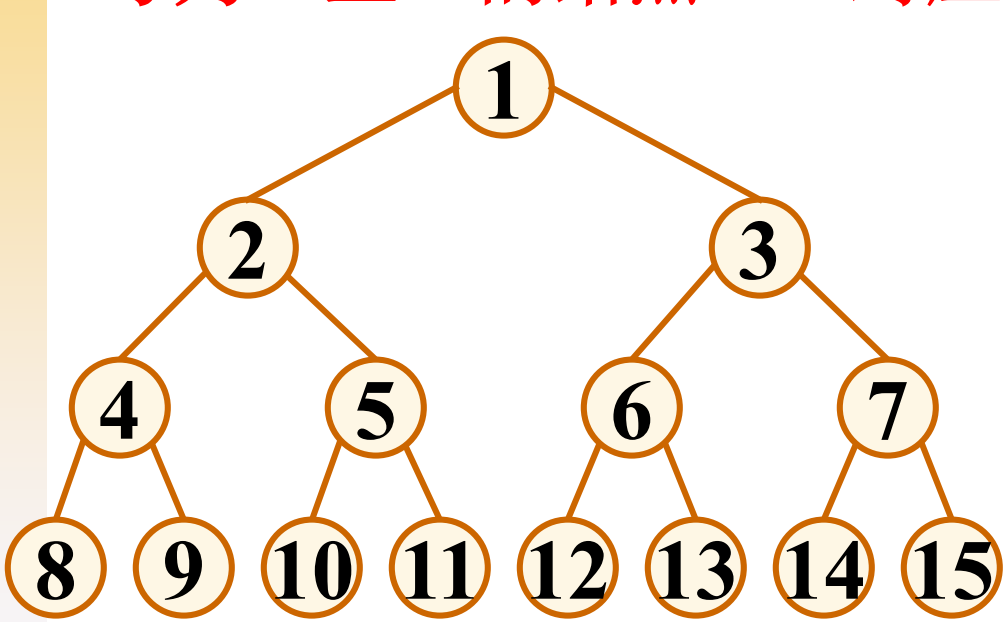


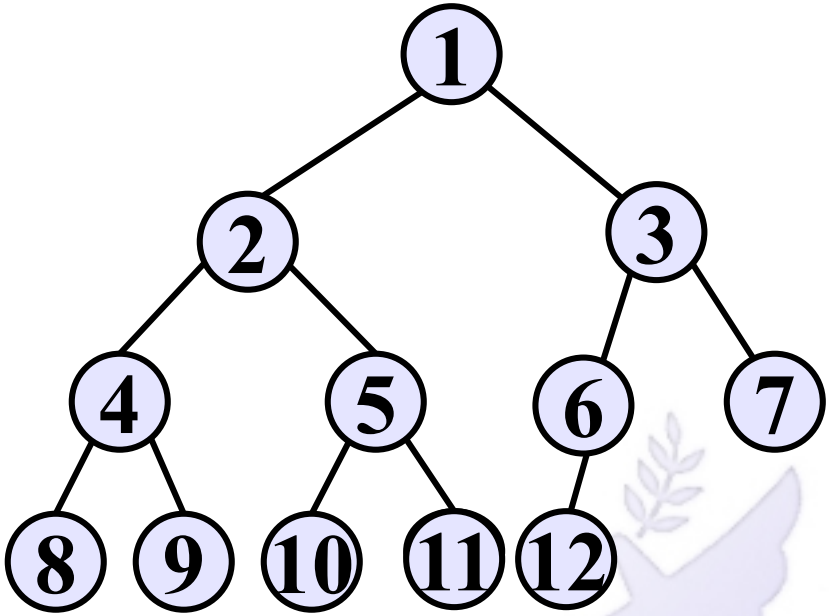
5.2.2 二叉树的性质－ 3

- 性质 3 ：对任何一棵二叉树，若它含有
  - n_0 个叶子结点;
  - n_2 个度为 2 的结点，则必存在关系式：
  - $n_0 = n_2+1$。
- 证明：
  - 设 二叉树上结点总数 $n = n_0 + n_1 + n_2$
  - 又 二叉树上分支总数 $b = n_1+2n_2$
  - 而又有 $b = n-1$
  - →→ $n_0 + n_1 + n_2 - 1 = n_1+2n_2$
  - →→ $n_0 = n_2 + 1$ 。

两类特殊的二叉树

- **满二叉树**：高度为h且含有2^h-1个结点的二叉树。
- **完全二叉树**：树中所含的 n 个结点和满二叉树中编号为 1 至 n 的结点一一对应。





5.2.2 二叉树的性质－ 4

♦ 性质 4 ： 具有 n 个结点的完全二叉树的高度为

¶ $\lceil \log_2(n+1) \rceil$。

❖证明：

❖设完全二叉树的高度为 h，

❖则根据第二条性质得:$2^{h-1}-1 < n \leq 2^h-1$

❖→→ $2^{h-1} < n+1 \leq 2^h$

❖→→ $h-1 < \log_2(n+1) \leq h$

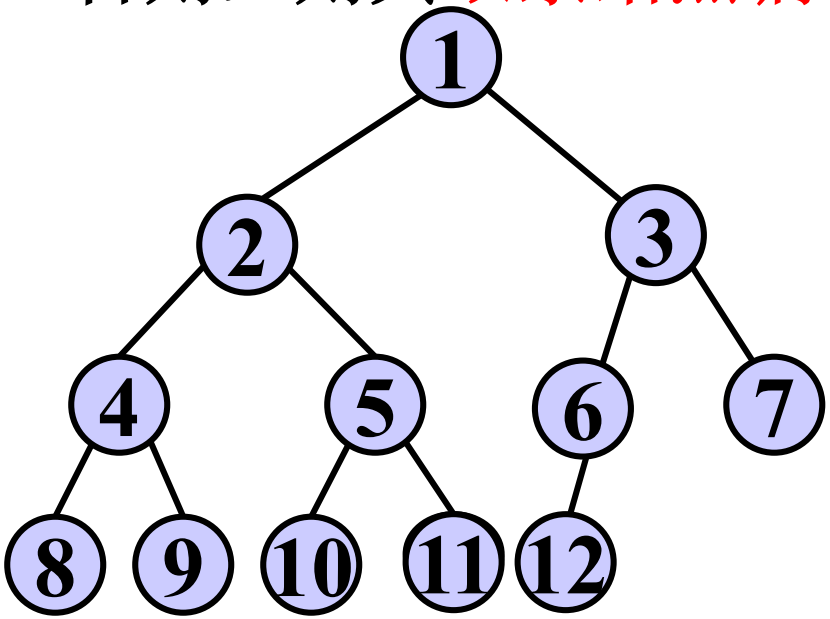
❖又因为 h 只能是整数，因此， $h = \lceil \log_2(n+1) \rceil$。

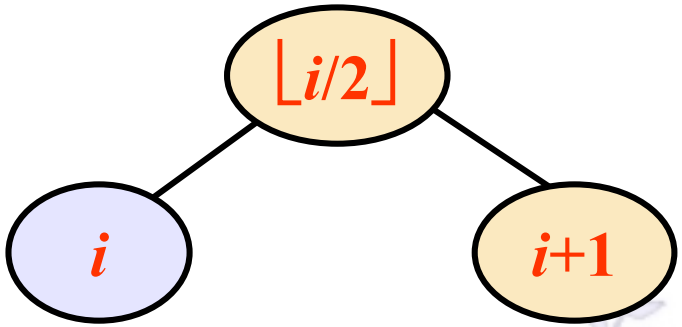
注： 当n>0时也可写为$h = \lfloor \log_2 n \rfloor + 1$。

♦性质 2 ： 深度为 h 的二叉树上至多含 2^h-1 个结点(h≥1)。

5.2.2 二叉树的性质－5

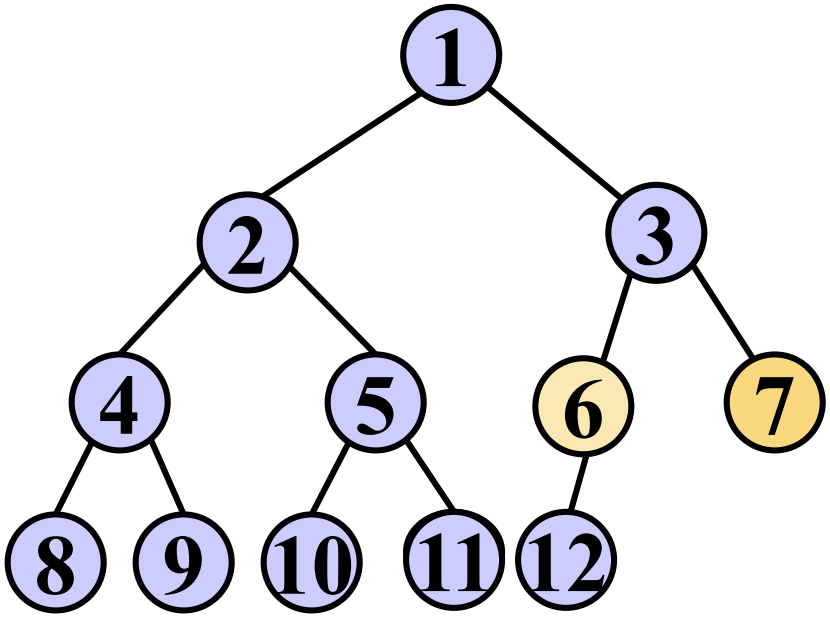
- 若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点：
- (1) 若 i=1，则结点i是二叉树的根，无双亲；否则，则其双亲结点编号为$\lfloor i/2 \rfloor$；

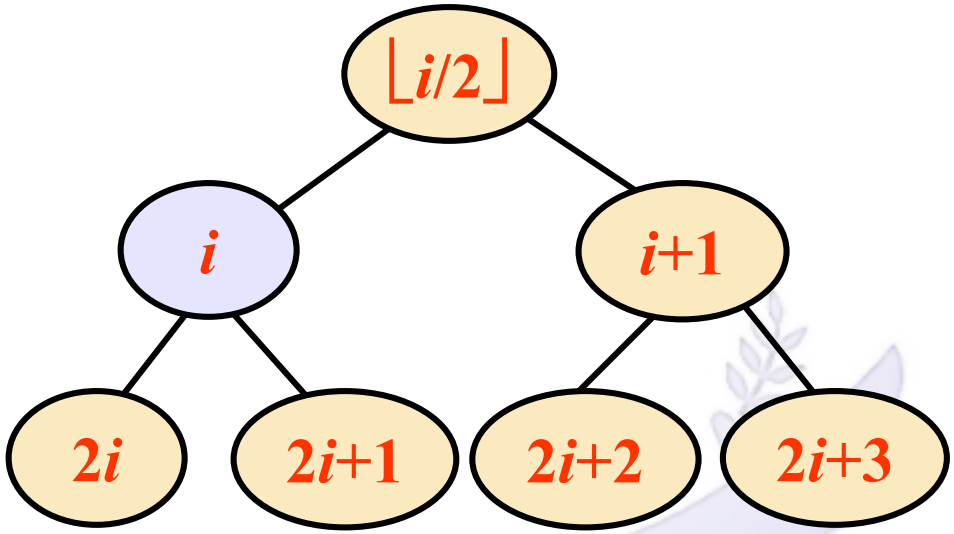




5.2.2 二叉树的性质－5

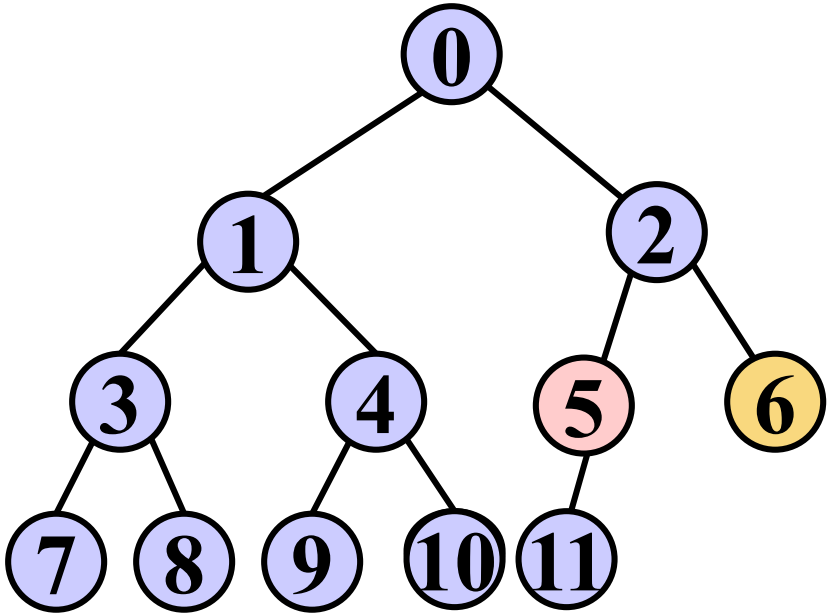
- (2) 若 $2i>n$，则结点i无左孩子；
  否则，结点i的左孩子结点的编号为 $2i$ ；
- (3) 若 $2i+1>n$，则结点i无右孩子结点；
  否则，结点i的右孩子结点的编号为$2i+1$ ；

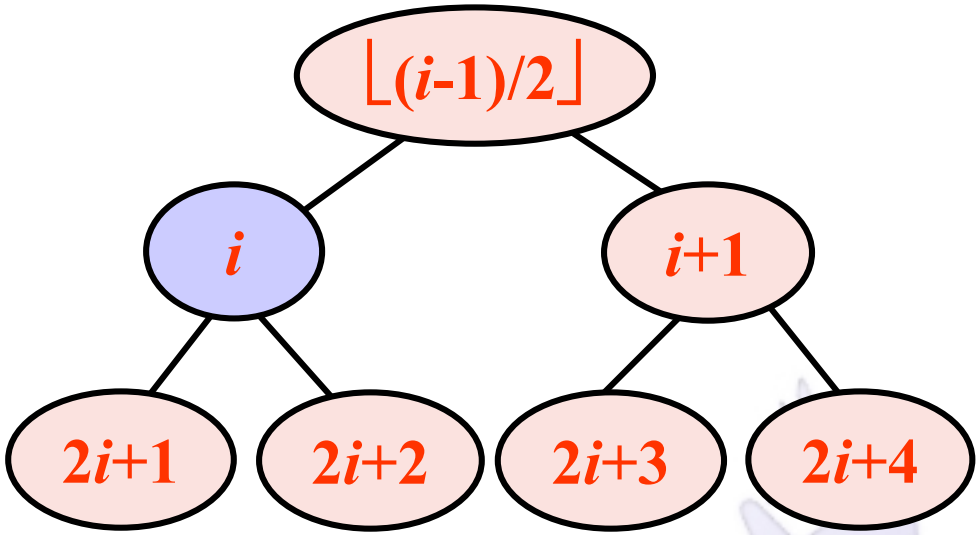




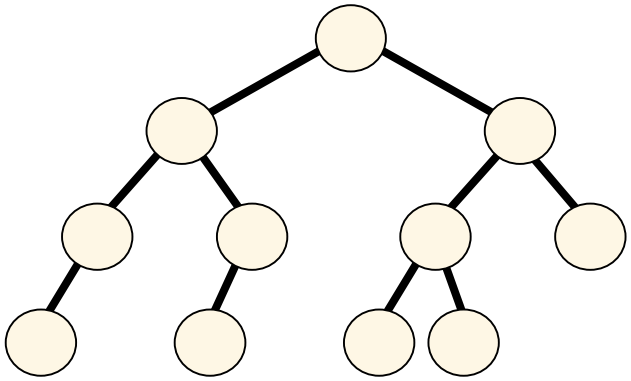
5.2.2 二叉树的性质－5

- 完全二叉树的性质
- 如果根结点从0开始编号，则结点i与其父结点和子结点的对应关系是：

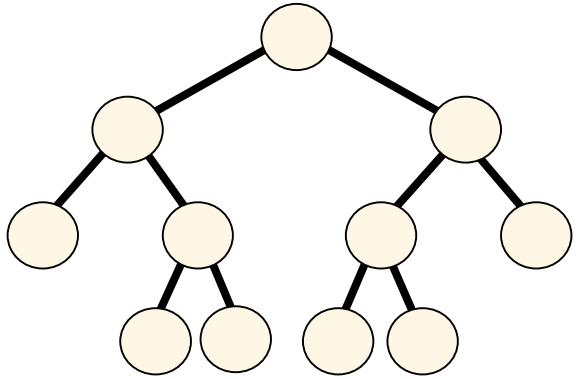




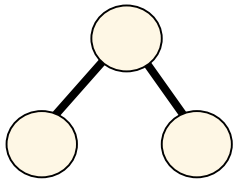
其他几种典型的二叉树



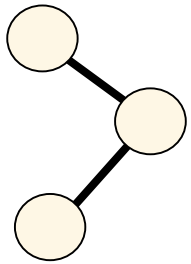
理想平衡树或丰满树



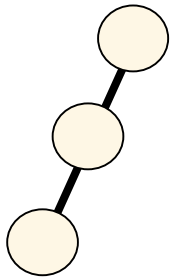
严格二叉树或正则二叉树



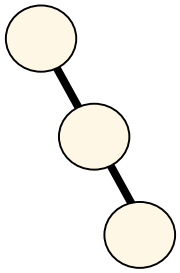
双枝树



单枝树



左斜单枝树



右斜单支树

5.3 二叉树的存储结构

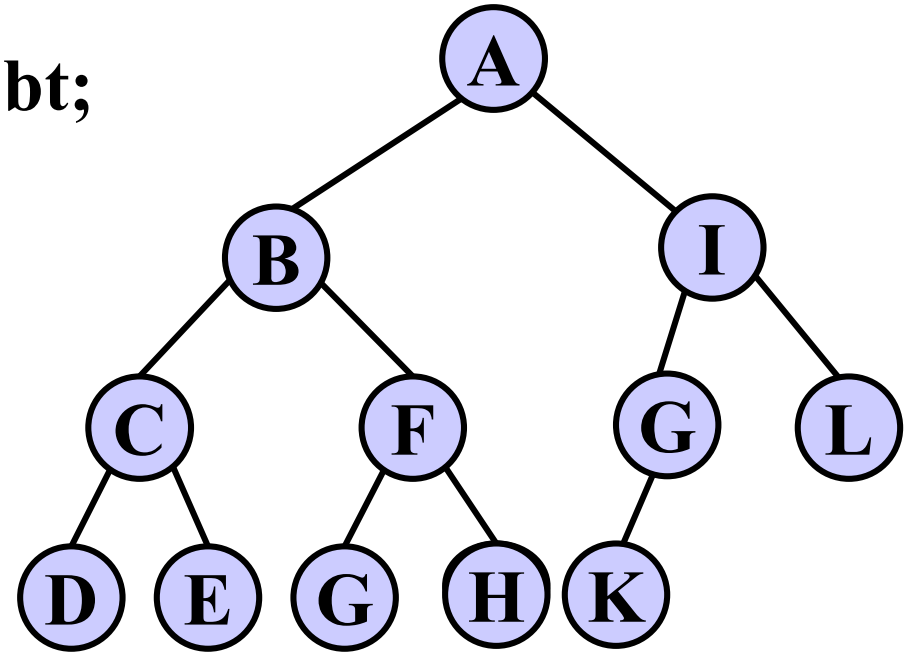
- 5.3.1 二叉树的顺序存储表示
- 5.3.2 二叉树的链式存储表示

5.3.1 二叉树的顺序存储表示

```
#define  MAX_TREE_SIZE  100
          // 二叉树的最大结点数
typedef struct {
      TElemType data[MAX_TREE_SIZE]; //存储数组
      int n;              //当前结点个数
} SqBTree;
```

5.3.1 二叉树的顺序存储表示

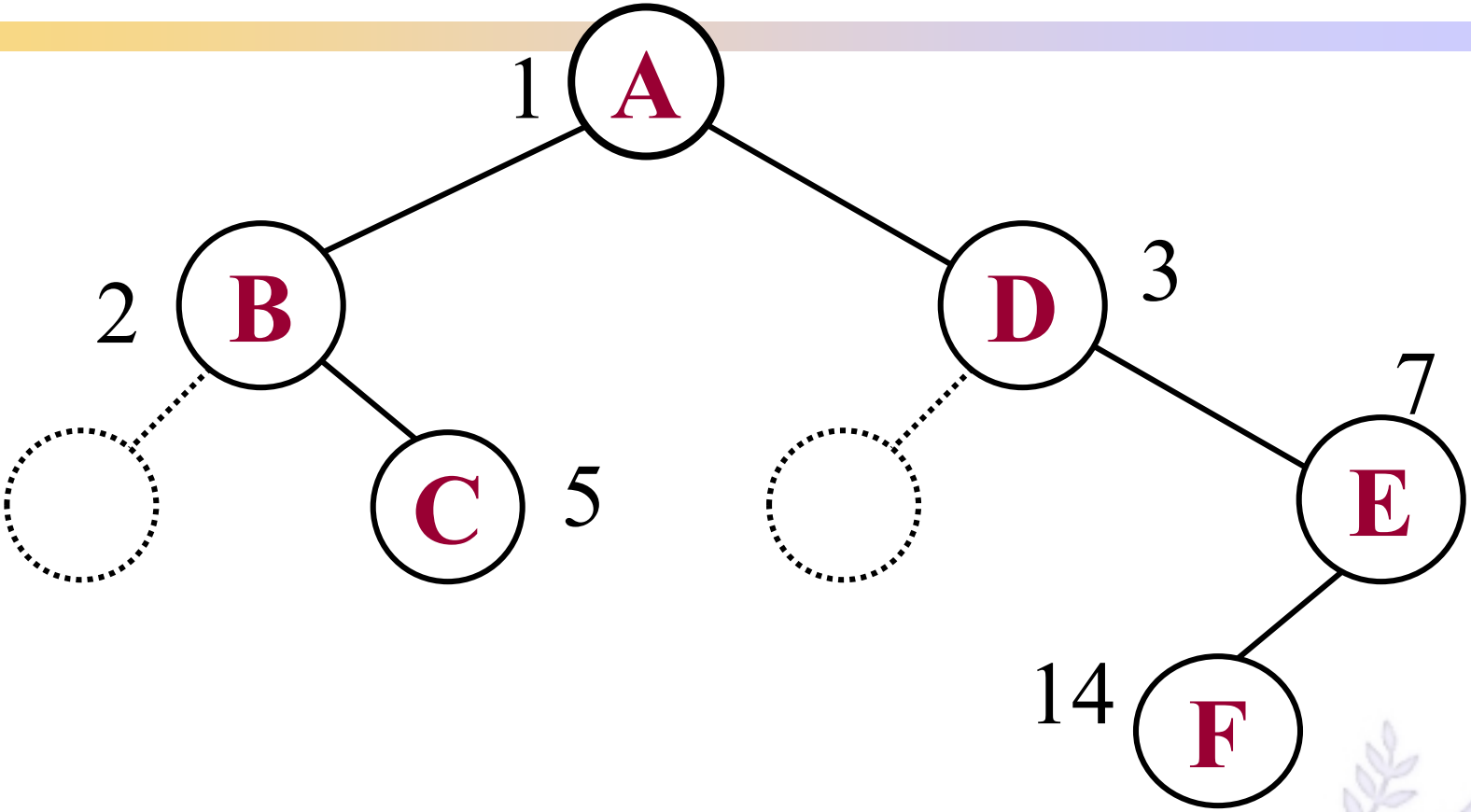
SqBTree bt;



bt

A	B	I	C	F	G	L	D	E	G	H	K			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

5.3.1 二叉树的顺序存储表示

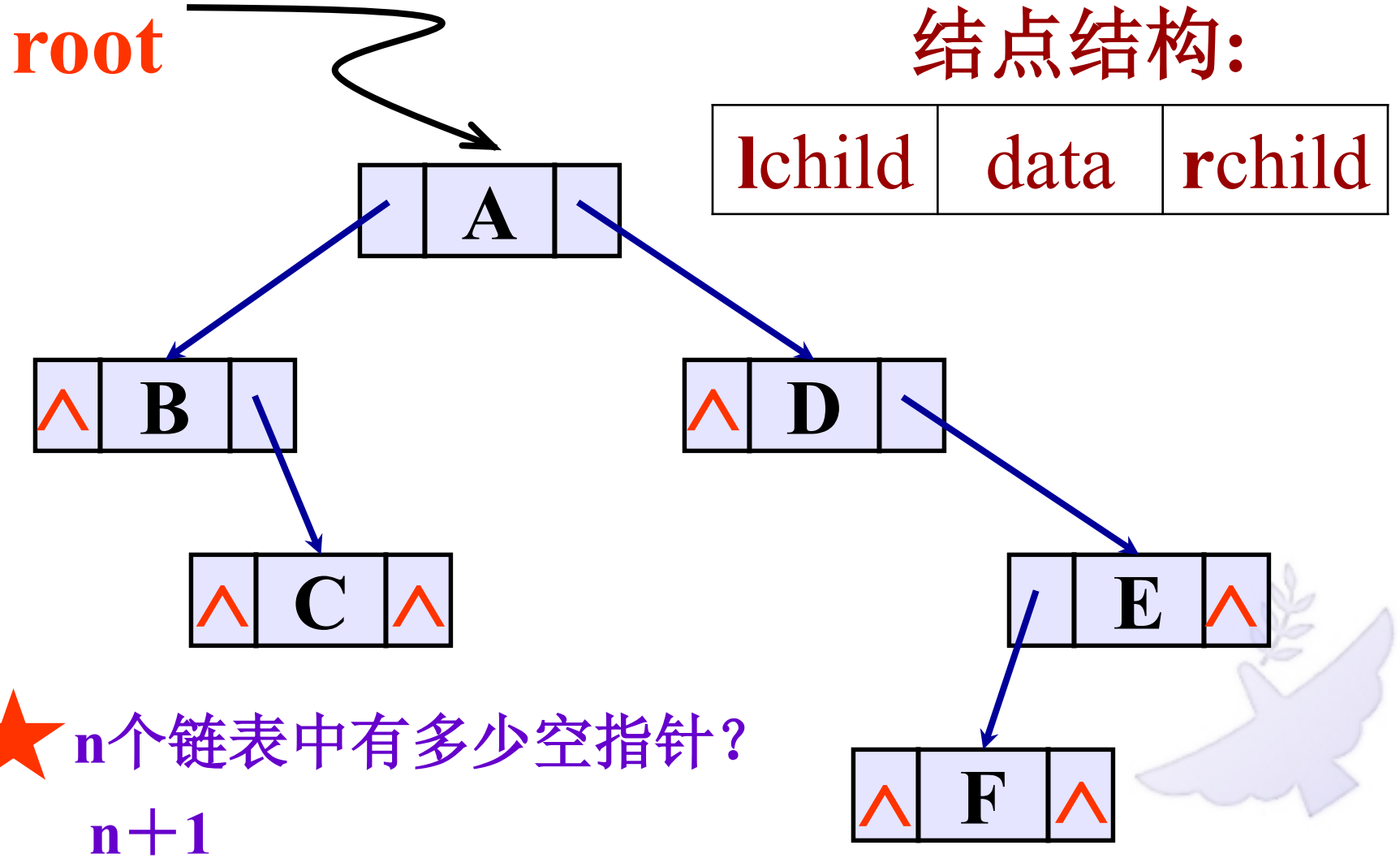


A	B	D		C		E							F	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

5.3.2 二叉树的链式存储表示

- 1) 二叉链表
- 2) 三叉链表
- 3) 双亲链表
- 4) 线索链表

1) 二叉链表



★ n个链表中有多少空指针？

n＋1

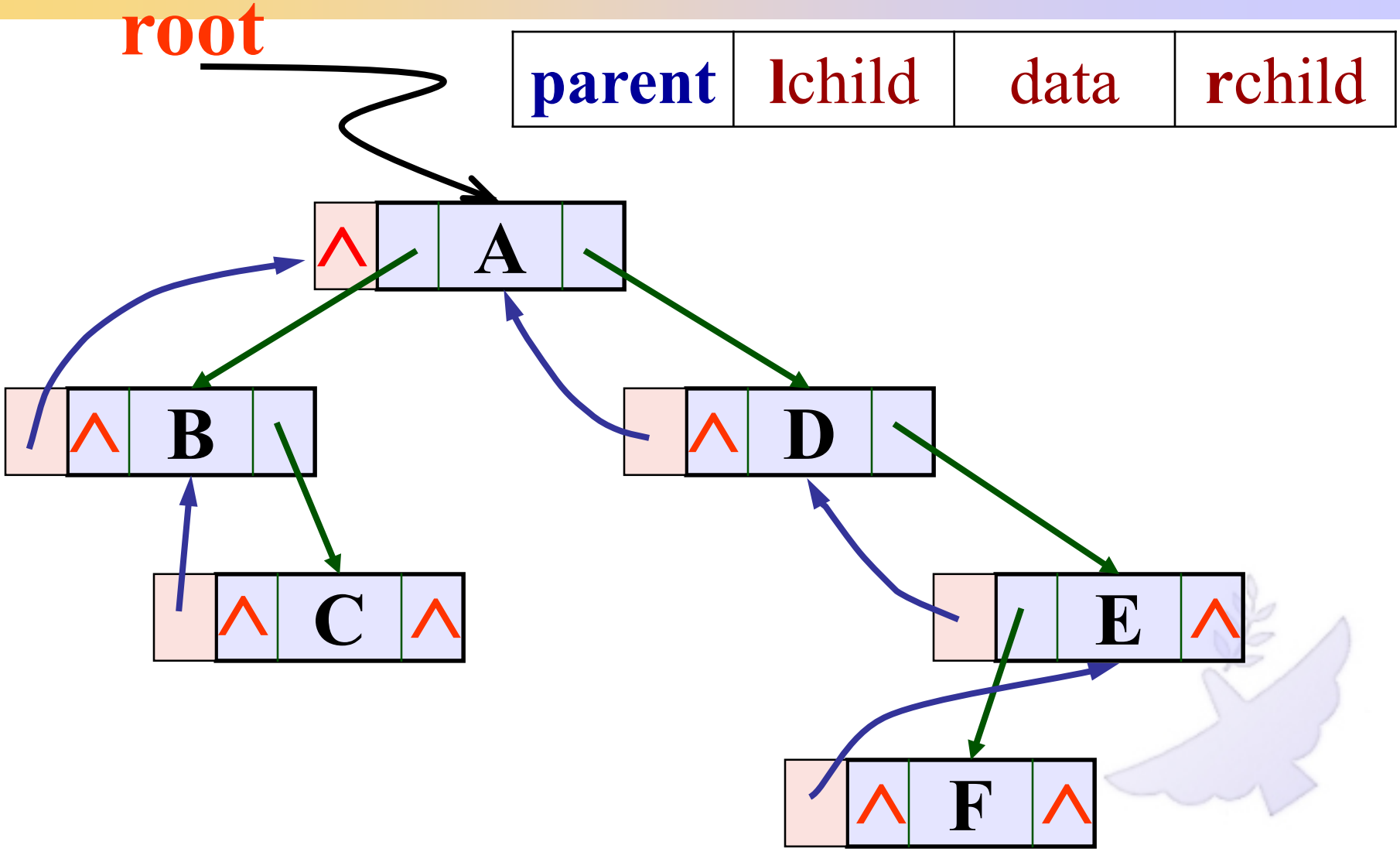
1) 二叉链表

结点结构：


```
typedef struct BiTNode { // 结点结构
    TElemType      data;
    struct BiTNode  *lchild, *rchild;
                                  // 左右孩子指针
} BiTNode, *BiTree;
```

2) 三叉链表

结点结构:



2) 三叉链表

结点结构:

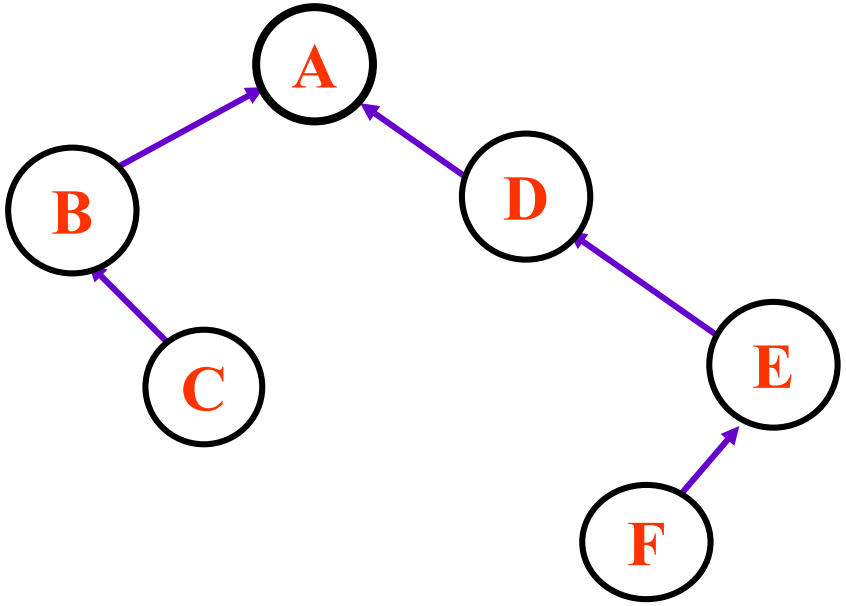
parent	lchild	data	rchild

```
typedef struct TriTNode { // 结点结构
    TElemType       data;
    struct TriTNode  *lchild, *rchild;
                          // 左右孩子指针
    struct TriTNode  *parent;  //双亲指针
} TriTNode, *TriTree;
```

3) 双亲链表

结点结构:

data	parent	LRTag



BPTree

root →

0	B	2	L
1	C	0	R
2	A	-1	
3	D	2	R
4	E	3	R
5	F	4	L
6			
7			
8			

3) 双亲链表

```
typedef struct BPTNode { // 结点结构
        TElemType  data;
        int  parent;     // 指向双亲的指针
        char  LRTag;     // 左、右孩子标志域
} BPTNode
```

```
typedef struct BPTree{ // 树结构
        BPTNode nodes[MAX_TREE_SIZE];
        int  num_node;     // 结点数目
        int  root;         // 根结点的位置
} BPTree
```

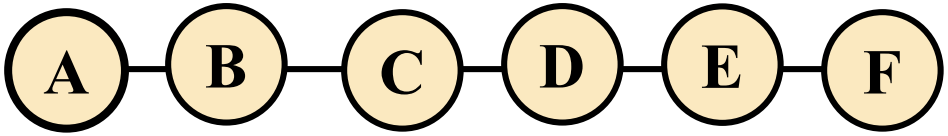
5.3.2 二叉树的链式存储表示

- 说明：
- 1、二叉链表和三叉链表也可以用静态链表表示。
- 2、访问结点复杂度：

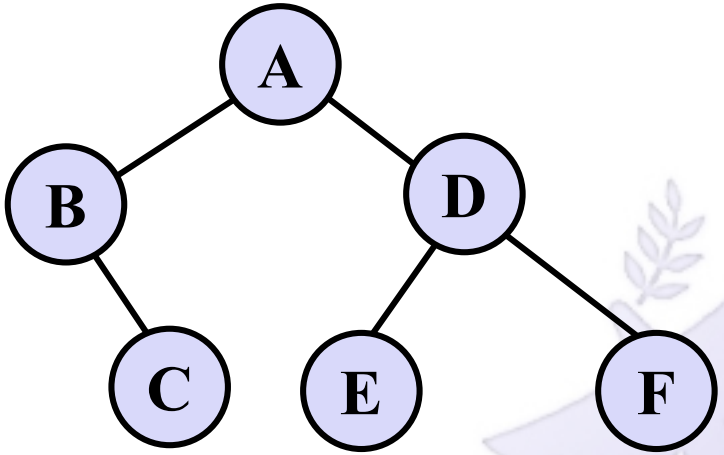
	访问子结点 复杂度	访问父结点 复杂度
二叉链表	O(1)	O(n)
三叉链表	O(1)	O(1)
双亲链表	O(n)	O(1)

5.4 二叉树的遍历

- 什么是遍历？
  - 按照某种顺序依次访问各个结点，使得每个结点均被访问一次，而且仅被访问一次。
- 线性结构
  - 只有一条访问路径

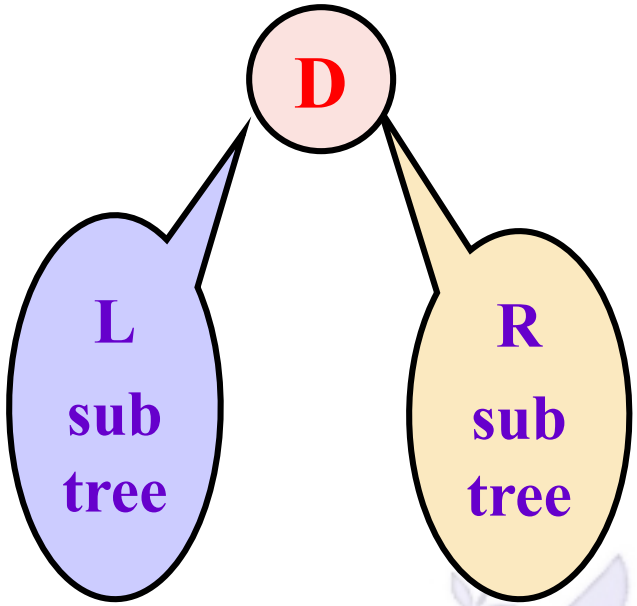


- 二叉树
  - 是非线性结构
  - 需要确定访问的顺序



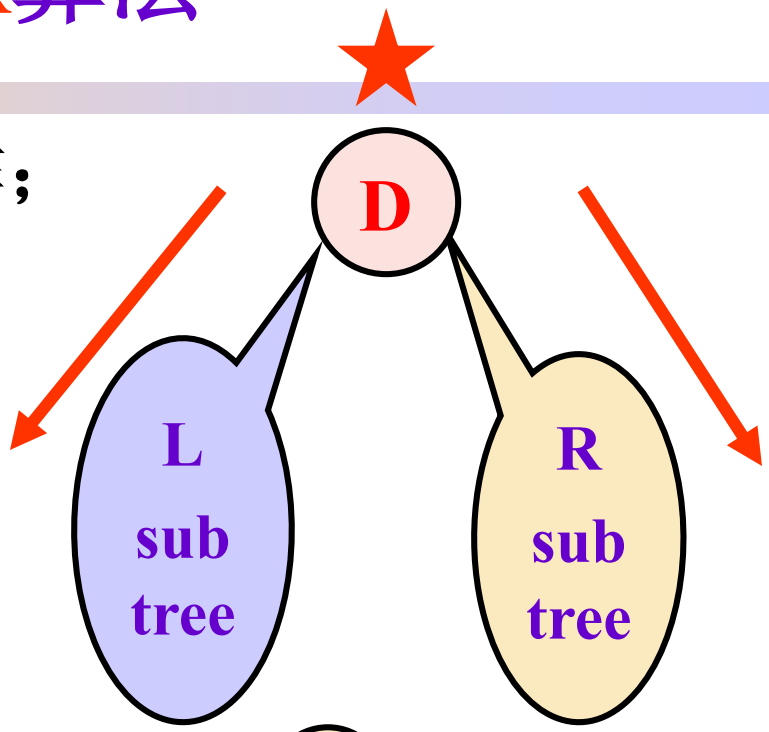
5.4.1 二叉树的访问顺序

- 六种访问顺序：
  - DLR　　先（根）序遍历
  - DRL
  - LDR　　中（根）序遍历
  - RDL
  - LRD　　后（根）序遍历
  - RLD
- 限定从左向右访问！

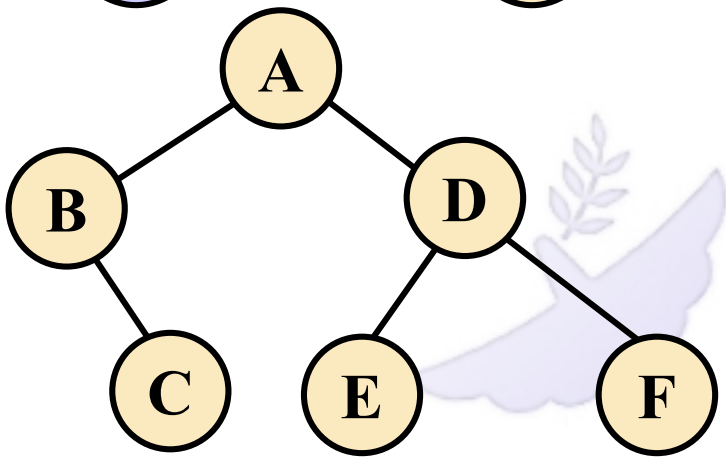


先（根）序的遍历DLR算法

- 若二叉树为空树，则空操作；
- 否则
  - （1）访问根结点；
  - （2）先序遍历左子树；
  - （3）先序遍历右子树；

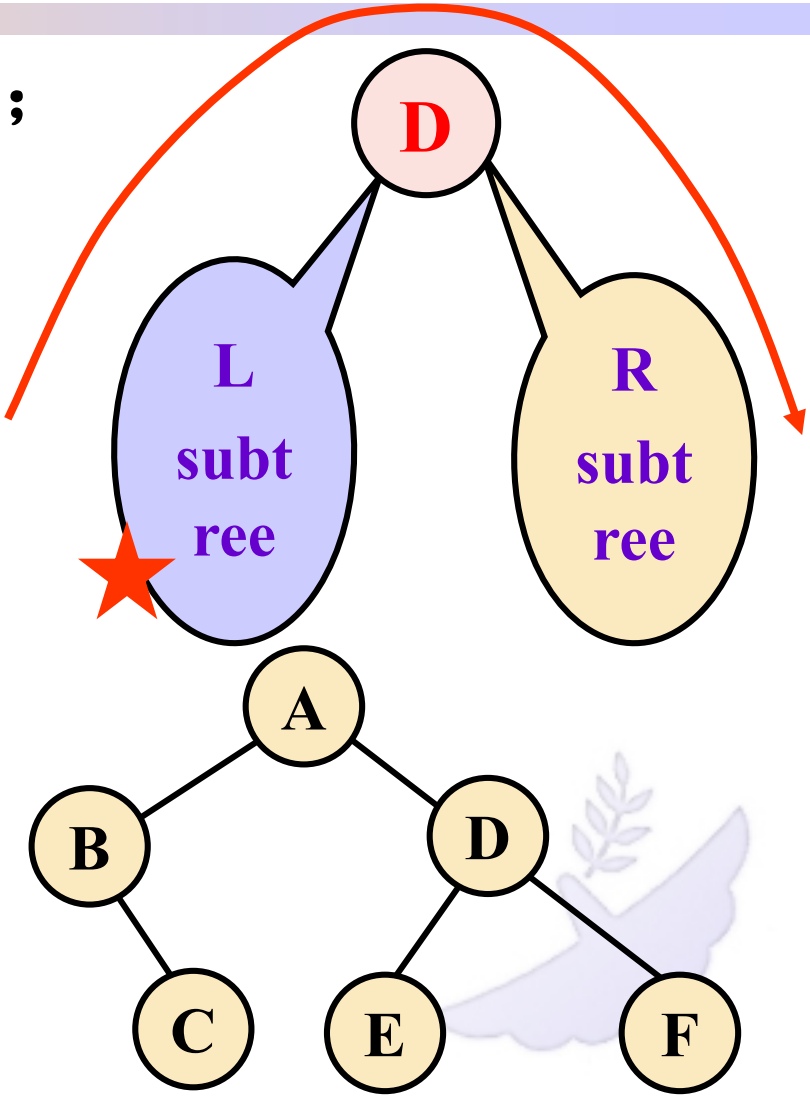


先序遍历：ABCDEF



中（根）序的遍历LDR算法

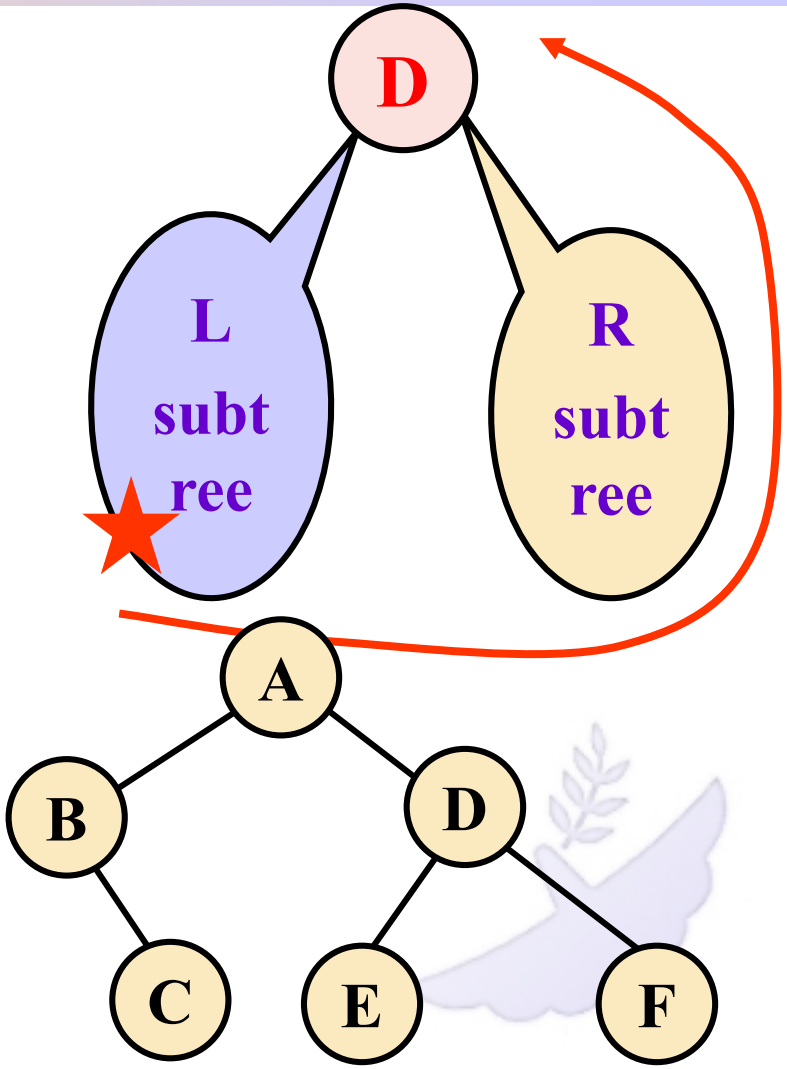
- 若二叉树为空树，则空操作；
- 否则：
  - （1）中序遍历左子树；
  - （2）访问根结点；
  - （3）中序遍历右子树；



中序遍历：BCAEDF

后（根）序的遍历LRD算法

- 若二叉树为空树，则空操作；
- 否则
  - （1）后序遍历左子树；
  - （2）后序遍历右子树；
  - （3）访问根结点；



后序遍历：CBEFDA

5.4.1 二叉树的访问顺序

先序遍历（DLR）

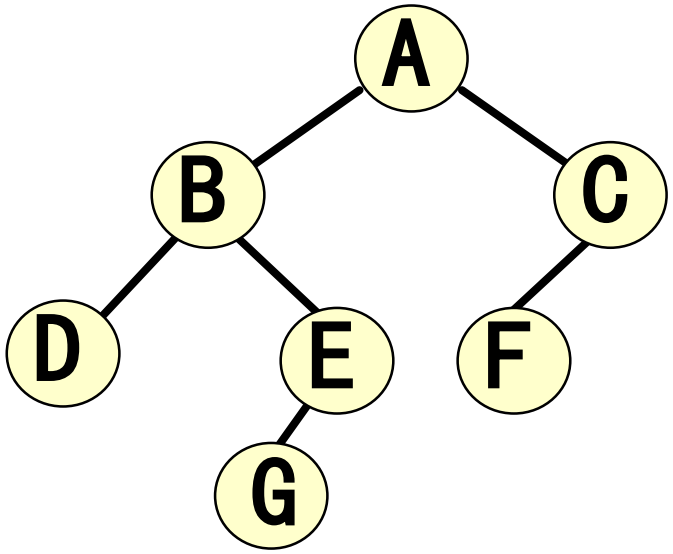
A B D E G C F

中序遍历（LDR）

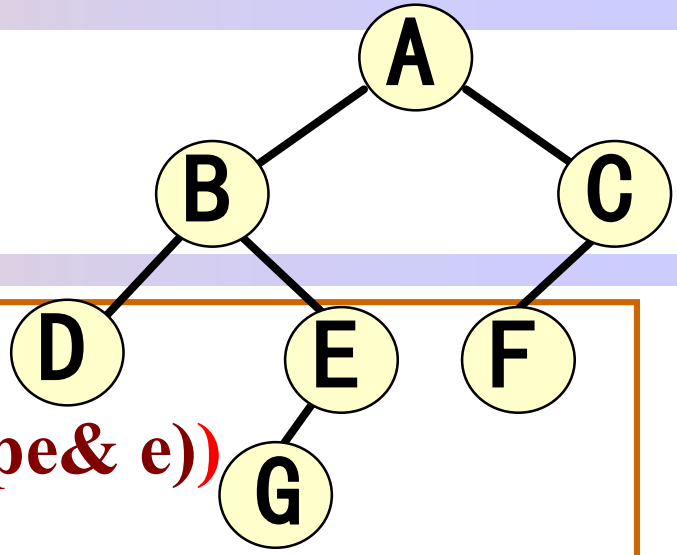
D B G E A F C

后序遍历（LRD）

D G E B F C A

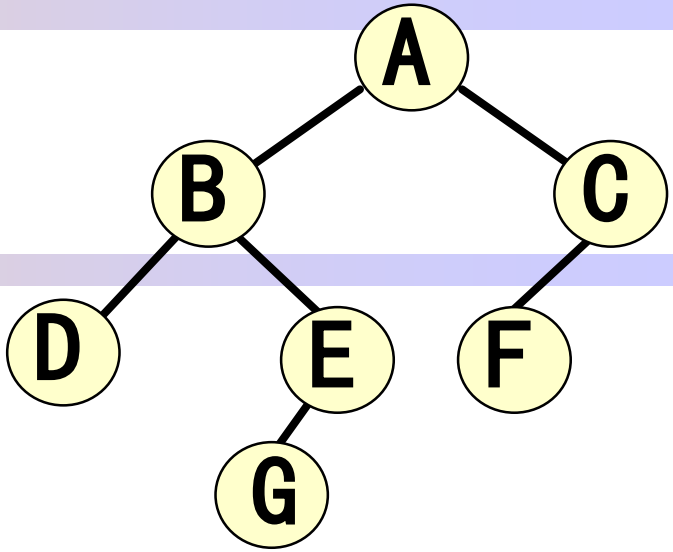


5.4.2 遍历算法的递归描述



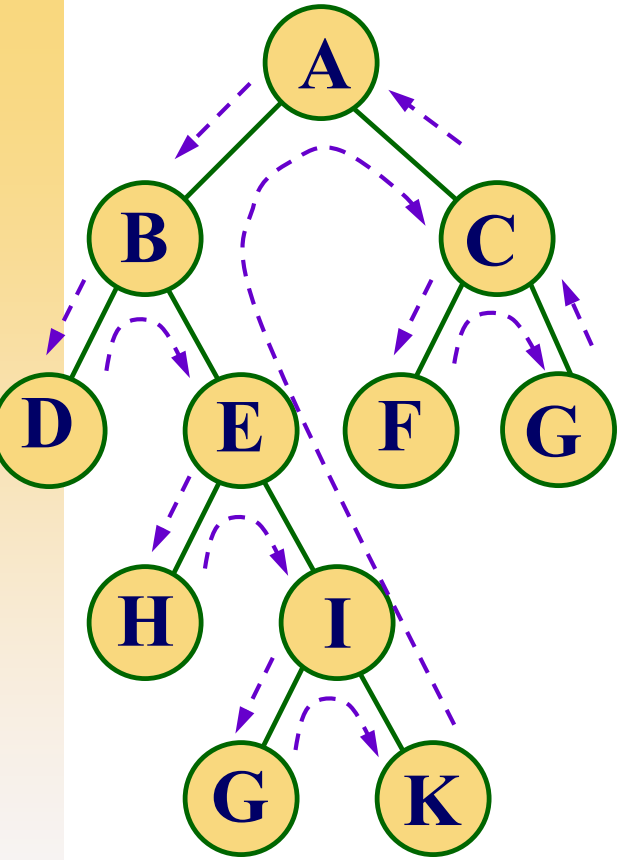

```
void Preorder (BiTree T,
               void( *visit)(TElemType& e))
{ // 先序遍历二叉树DLR
    if (T != NULL) {
        visit(T->data);          // 访问根结点
        Preorder(T->lchild, visit); //先序遍历左子树
        Preorder(T->rchild, visit);//先序遍历右子树
    }
}// Preorder
```

5.4.2 遍历算法的递归描述

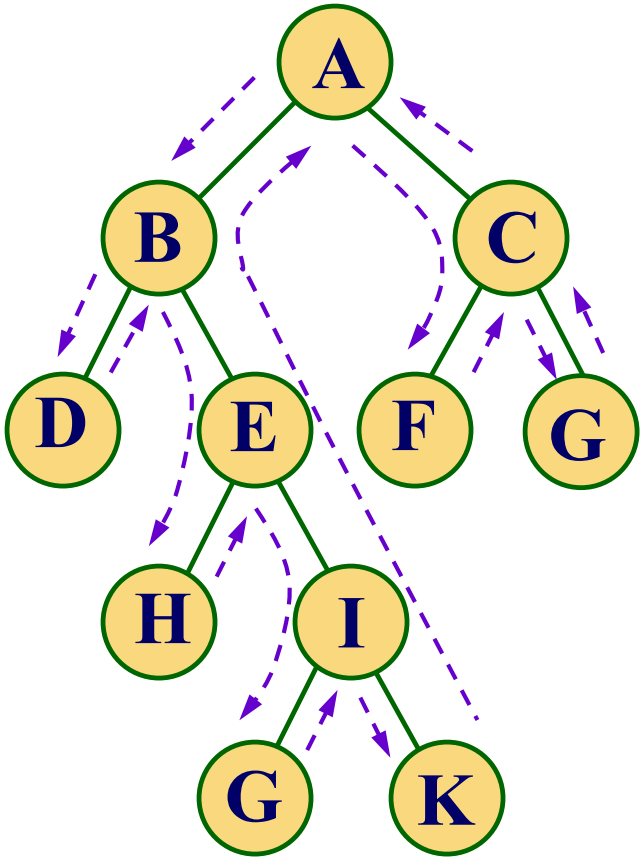



```
void Inorder (BiTree T, void( *visit)(TElemType& e))
{if (T != NULL) {
        Inorder(T->lchild, visit); //中序遍历左子树
        visit(T->data);            // 访问根结点
        Inorder(T->rchild, visit);//中序遍历右子树
    }
}// Inorder
```

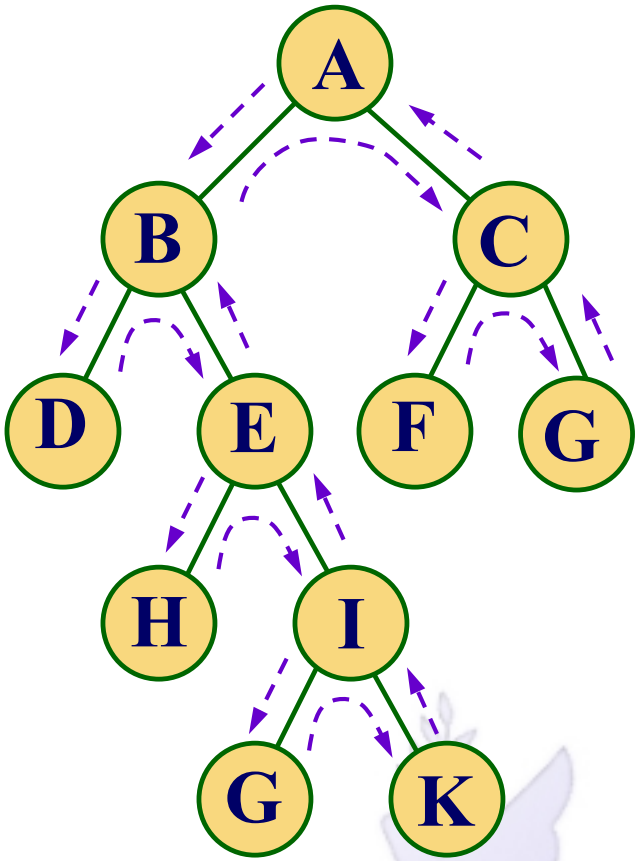
二叉树遍历过程分析



先序遍历



中序遍历

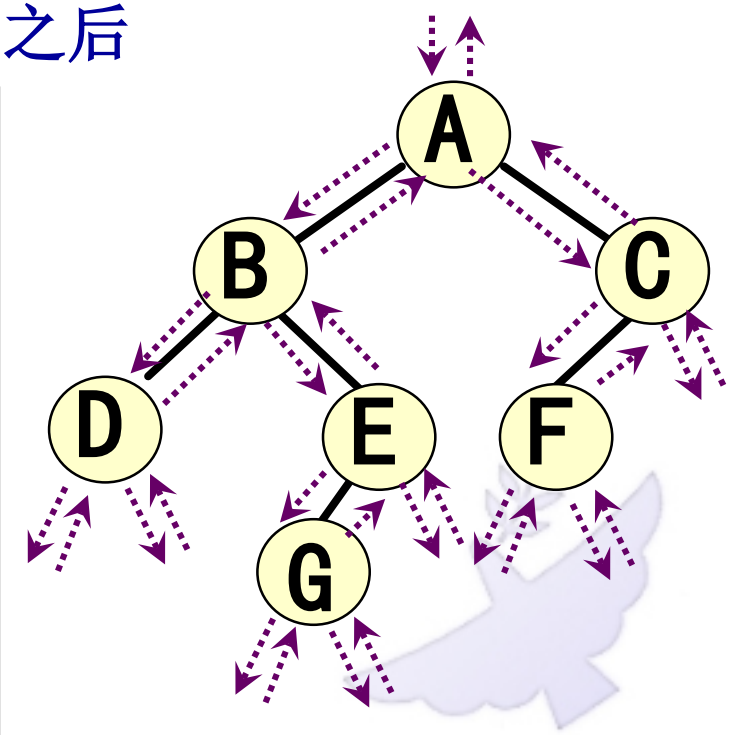


后序遍历

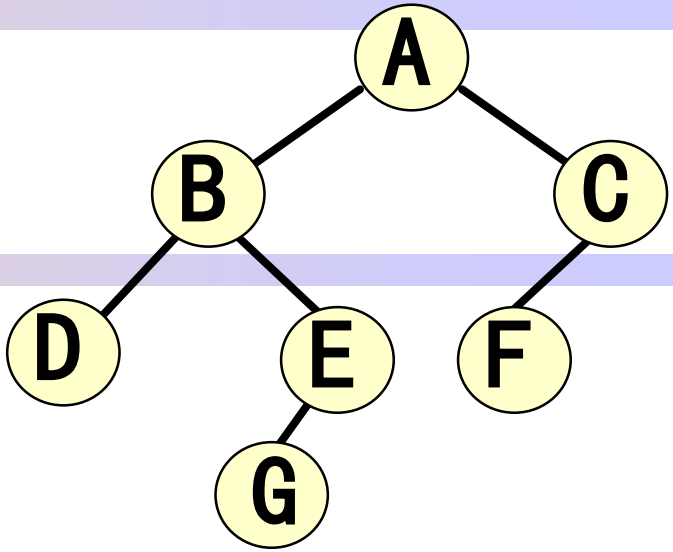
二叉树遍历过程分析

- 欧拉巡回遍历： T 的每个结点 v 都会遇到三次:
  - ① 在对v的左子树做欧拉巡回遍历之前
  - ② 在对v的左子树做欧拉巡回遍历之后
  - ③ 在对v的右子树做欧拉巡回遍历之后

```
void eulerTour (T, v) {
        visit (v);          //“先序”
        if ( v->lchild != NULL )
            eulerTour (T, v->lchild);
        visit (v);          //“中序”
        if ( v->rchild != NULL )
            eulerTour (T, v->rchild);
        visit (v);          //“后序”
} // eulerTour
```



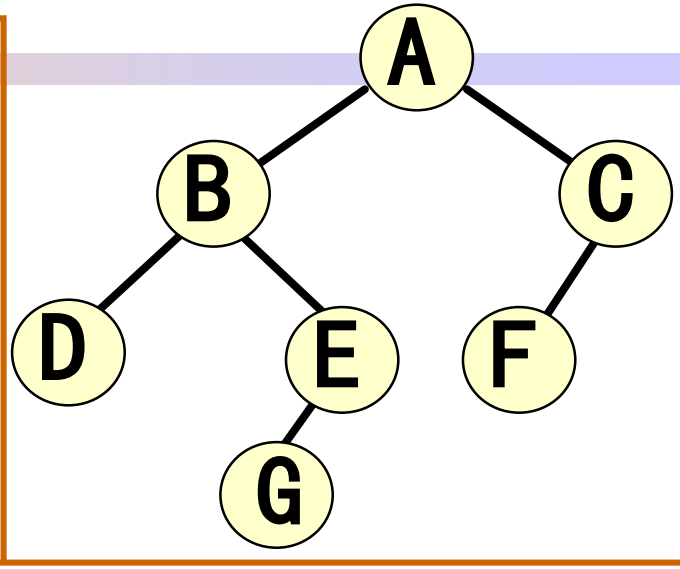
中序遍历算法的递归描述




```
void Inorder (BiTree T, void( *visit)(TElemType& e))
{if (T != NULL) {
        Inorder(T->lchild, visit); //中序遍历左子树
        visit(T->data);            // 访问根结点
        Inorder(T->rchild, visit);//中序遍历右子树
    }
}// Inorder
```

5.4.3 中序遍历LDR算法的非递归描述

- 若二叉树为空树则空操作；
- 否则
  - （1）中序遍历左子树；
  - （2）访问根结点；
  - （3）中序遍历右子树；

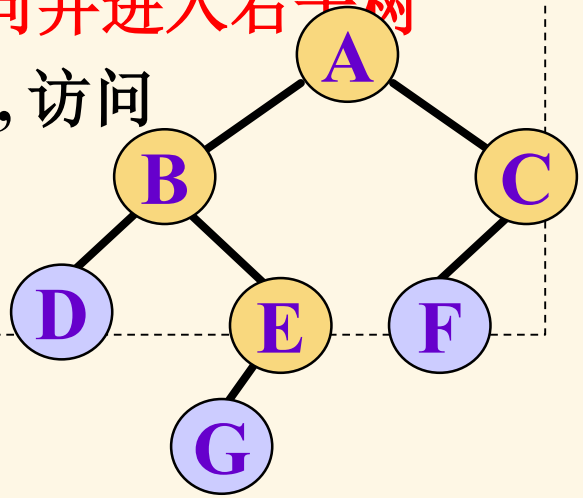


访问顺序：

1. 从根结点开始，依次进入左子树，并将经过的结点压入堆栈，直到当前结点的左指针为空；
2. 从栈中取出一个结点N访问；
3. 若N的右指针不为空，则处理其右子树，从右子树的根结点开始继续遍历。

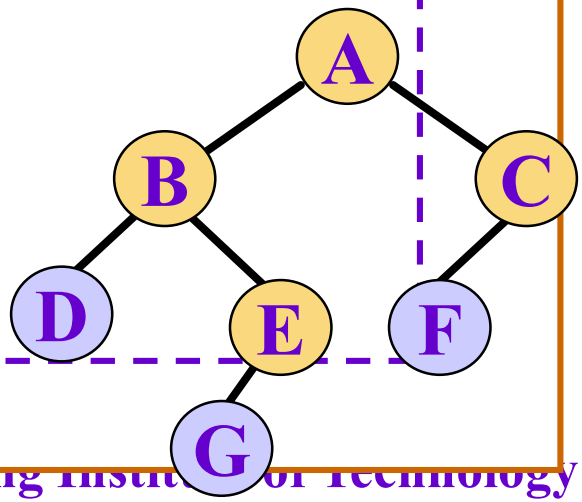
5.4.3 中序遍历算法的非递归描述

```
void inOrder_iter ( BiTree BT, void (* visit)(TElemType)) {
//利用栈实现二叉树BT的中序遍历
   InitStack(S); p = BT;   //p是遍历指针，从根开始
   do {// 当p非空 或 堆栈非空时循环访问结点
      while ( p != NULL ){   //遍历指针进到左子树
            Push(S, p);    p = p->lchild;
      }//while
      if ( !StackEmpty(S)) { //栈非空退栈访问并进入右子树
            Pop(S, p);  visit(p->data);//退栈, 访问
            p = p->rchild;//进入右子树
      }//if
   } while ( p != NULL || !StackEmpty(S));
}//inOrder_iter
```



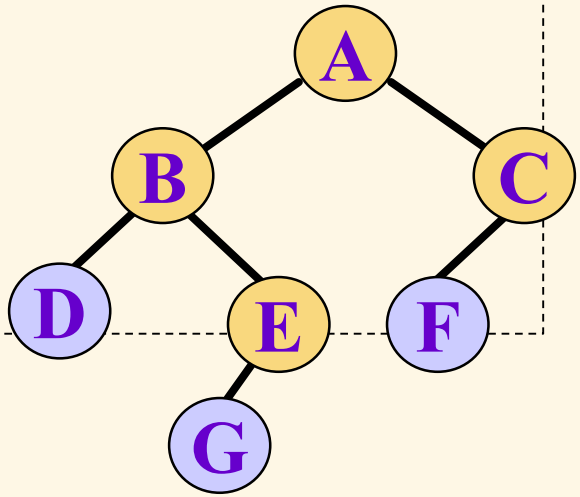
中序遍历算法的非递归描述

```
void InOrderTraverse(BiTree T,
                     void (* visit)(TElemType)) {
  InitStack(S); p = T;//从根结点开始
  while (p || !StackEmpty(S)) {
      if (p) {
          Push(S, p);  p = p->lchild; }
      else {
          Pop(S, p);  visit(p->data);
          p = p->rchild; }
  }//while
}// InOrderTraverse
```



5.4.3 先序遍历算法的非递归描述

```
void preOrder_iter ( BiTree BT, void (* visit)(TElemType)) {
//利用栈实现二叉树BT的先序遍历
   InitStack(S); p = BT;   //p是遍历指针，从根开始
   do {// 当p非空 或 堆栈非空时循环访问结点
      while ( p != NULL ){   //遍历指针进到左子女
            visit(p->data);
            if( p->rchild!=NULL) Push(S, p->rchild);
            p = p->lchild;
      }//while
      //栈不空时退栈
      if ( !StackEmpty(S))  Pop(S, p);  //退栈
   } while ( p != NULL || !StackEmpty(S));
}//inOrder_iter
```



思考:后序的非递归遍历

- 遍历二叉树的结果是结点的一个线性序列

先序遍历（DLR）

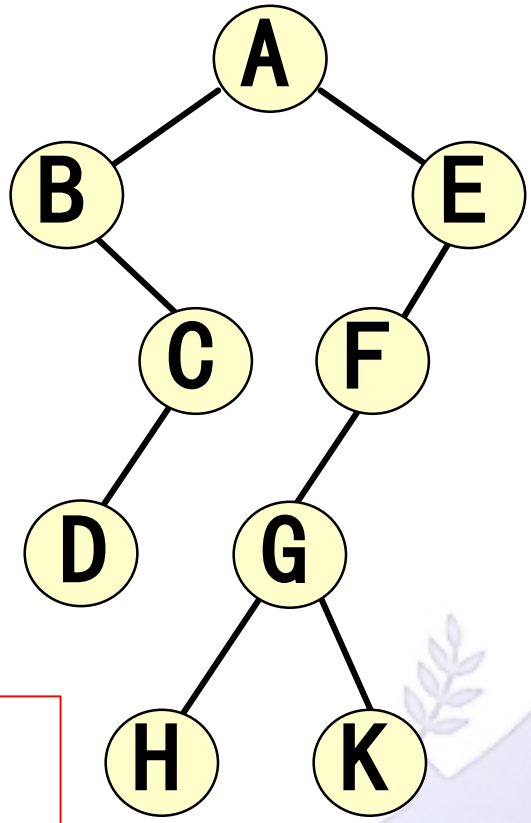
A B C D E F G H K

中序遍历（LDR）

B D C A H G K F E

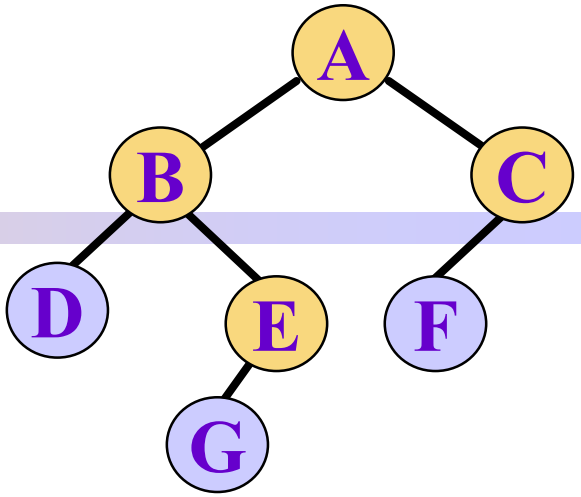
后序遍历（LRD）

D C B H K G F E A



- 后序遍历的实现方式：
  1. 设tag，记录左子树是否访问；
  2. 设前驱指针，判断右子树是否访问；

5.4.4 遍历算法的应用举例



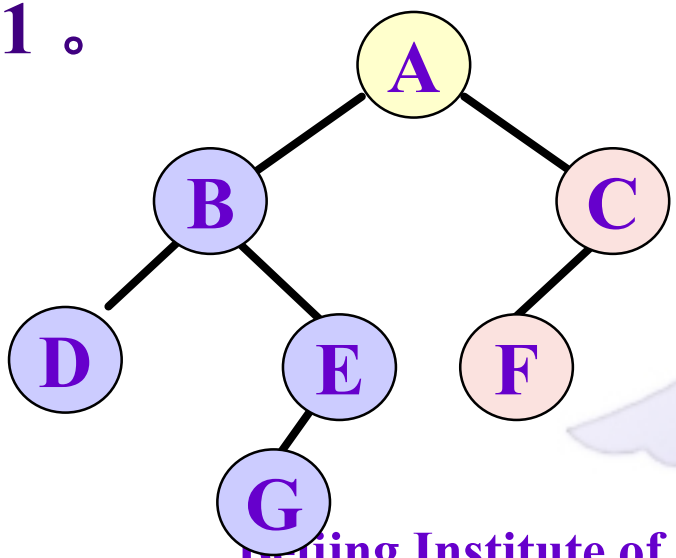
- 1) 统计二叉树中叶子结点的个数
- 叶子结点：左右指针都为空
- 基本思想：
  - ¶ 先序遍历二叉树，在遍历过程中查找叶子结点
  - ¶ 如果是空树，则返回0；
  - ¶ 如果是叶子结点，则返回1；
  - ¶ 否则为分支结点，叶子结点个数=
  - ¶ =左子树叶个数 + 右子树叶子个数。

1) 统计二叉树中叶子结点的个数

```
int  CountLeaf (BiTree T){
//递归程序，先序遍历。
    if (!T) return 0;
    if ((!T->lchild) && (!T->rchild)) //终止条件
            return 1;
    return CountLeaf(T->lchild) +
                        CountLeaf(T->rchild);
} // CountLeaf
```

2）求二叉树的深度(后序遍历)

- 二叉树的深度：为其左、右子树深度的最大值加1。
- 基本思想：
  - 若为空树，则返回0；
  - 否则分别求得左子树的深度和右子树的深度
  - 算法中“访问结点”的操作为：求左、右子树深度的最大值，然后加 1 。

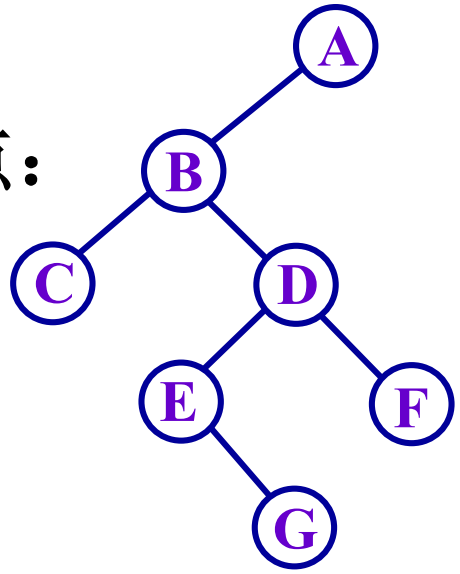


2）求二叉树的深度(后序遍历，递归)

```
int Depth (BiTree T ){ // 返回二叉树的深度
    if ( !T )   return 0; //终止条件
    else  {
        depthLeft = Depth( T->lchild );
        depthRight = Depth( T->rchild );
        depthval = 1 + (depthLeft > depthRight  ?
                        depthLeft : depthRight);
    }
    return depthval;
}// Depth (BiTree T )
```

3）用广义表方式输出二叉树

- 若二叉树为空，输出空格“ ”；
- 若二叉树非空，则先输出根结点的数据，再判断根是否是叶结点：
  - 若是叶结点，则空操作；
  - 若不是叶结点，则
  - ① 输出左括号“(”；
  - ② 递归输出左子树；
  - ③ 输出逗号“, ”；
  - ④ 递归输出右子树；
  - ⑤ 输出右括号“)”。

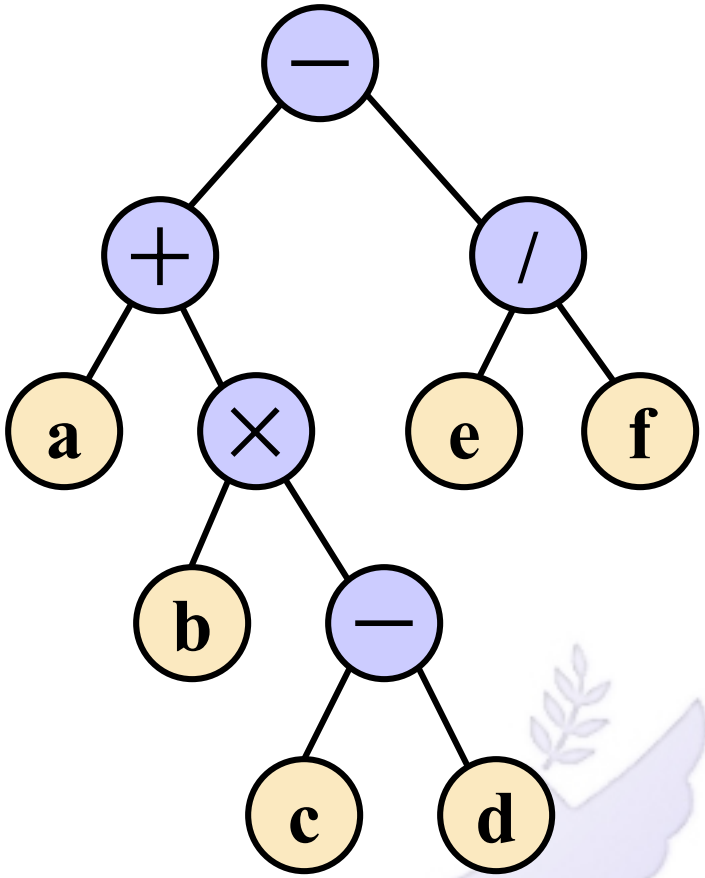


A (B (C, D (E (, G), F)))

表达式的二叉树表示

- a+b*(c-d)-e/f

- 按中序遍历二叉树, 序列为:
  - a+b*c-d-e/f：中缀表示
- 按先序遍历二叉树, 序列为:
  - －+a*b－cd/ef：前缀表示
- 按后序遍历二叉树, 序列为:
  - abcd－*+ef/－：后缀表示



5.4.5 以字符串的形式定义二叉树

- 二叉树结点：三个字符

根	左子树	右子树

字符串以“;”作为结束的标志

1) 空树

以“#;”表示

2) 只含根结点

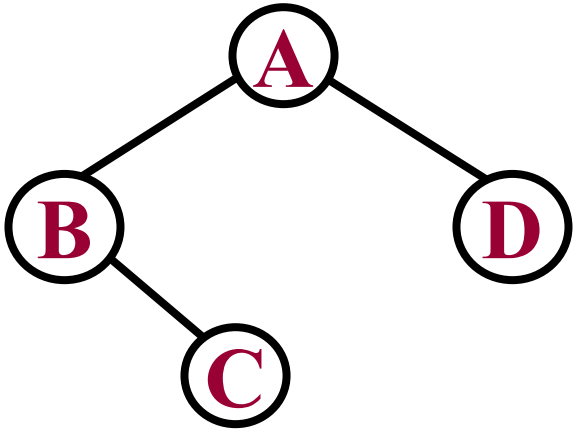
以字符串“A##;”表示

以字符串的形式定义二叉树(cont.)

定义一棵二叉树：

根	左子树	右子树

3) 含多个结点：假设按照先序遍历顺序

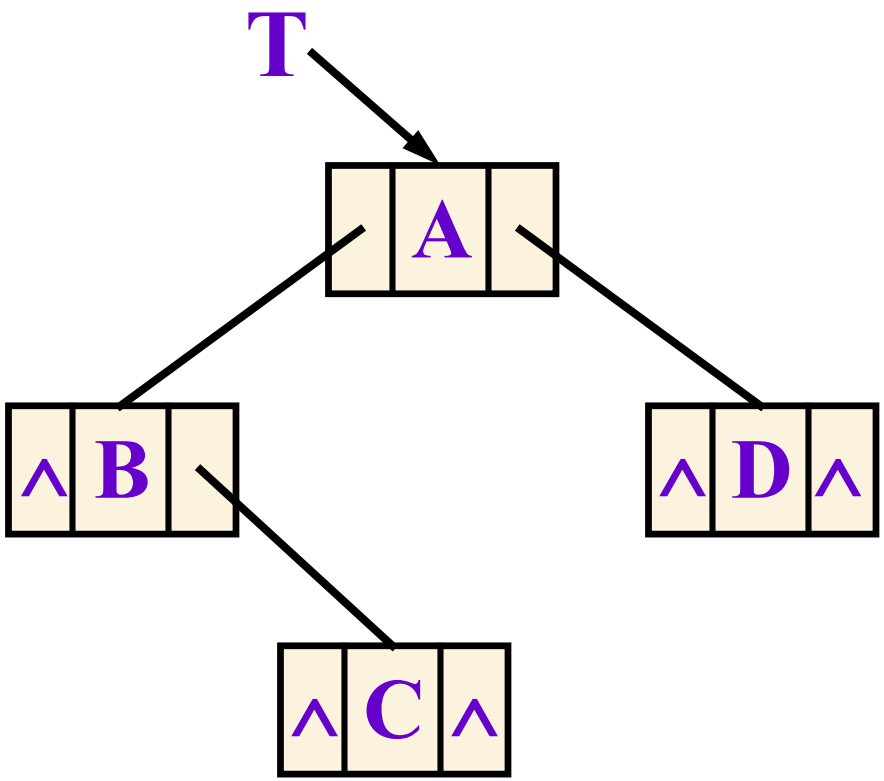


以字符串表示：

“AB#C# # D# #;”

由字符串得到二叉树（先序遍历）

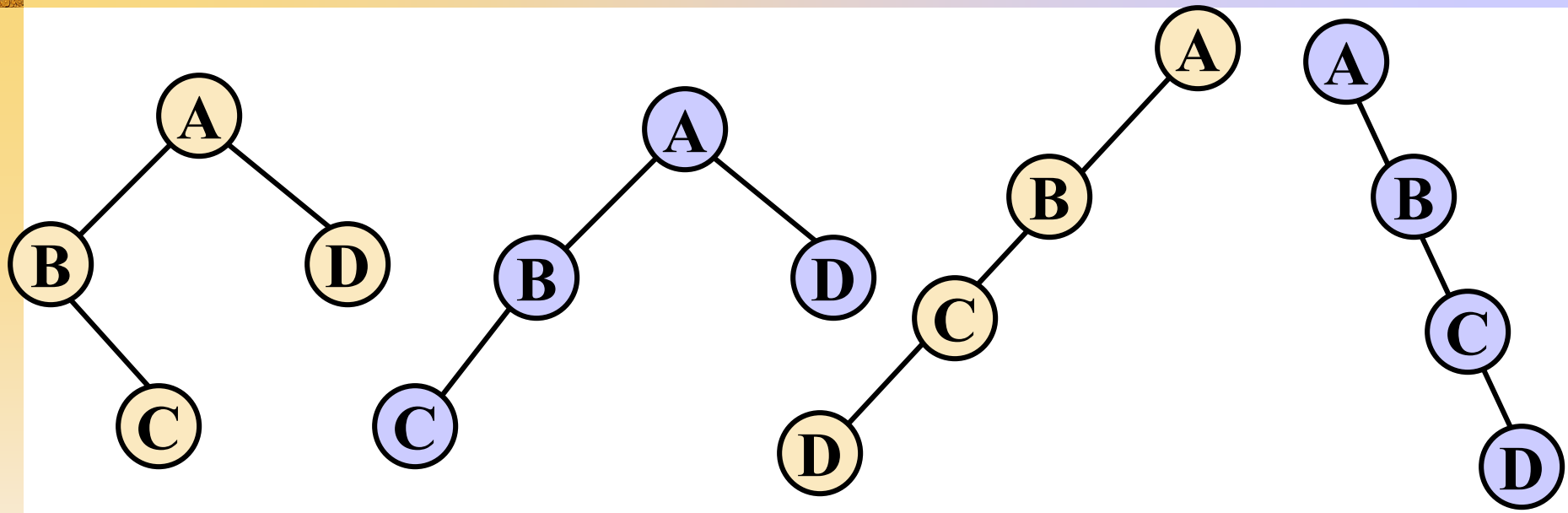
A B # C # # D # #



由字符串得到二叉树的算法（先序遍历）

```
void CreateBiTree (BiTree &T, char pre[ ], int& n ) {
    ch = pre[n++];  if ( ch == ';' ) return;
    if (ch==‘#') {T = NULL;}          //例“AB#C# # D# #;”
    else {
        if (!(T = (BiTNode *)malloc(sizeof(BiTNode))))
                    exit(OVERFLOW);
        T->data = ch;  // 生成根结点（基本操作）
        CreateBiTree(T->lchild, pre, n);  // 构造左子树
        CreateBiTree(T->rchild, pre, n);  // 构造右子树
    }//if (ch==‘#') …else
} // CreateBiTree
```

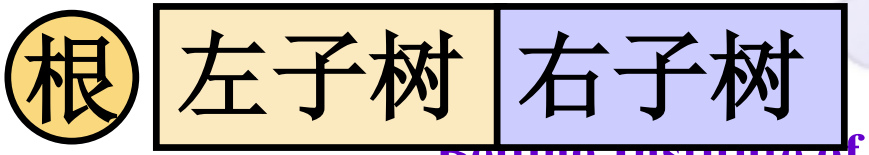
由先序序列能得到二叉树吗？(ABCD)



仅知二叉树的先序序列不能唯一确定一棵二叉树

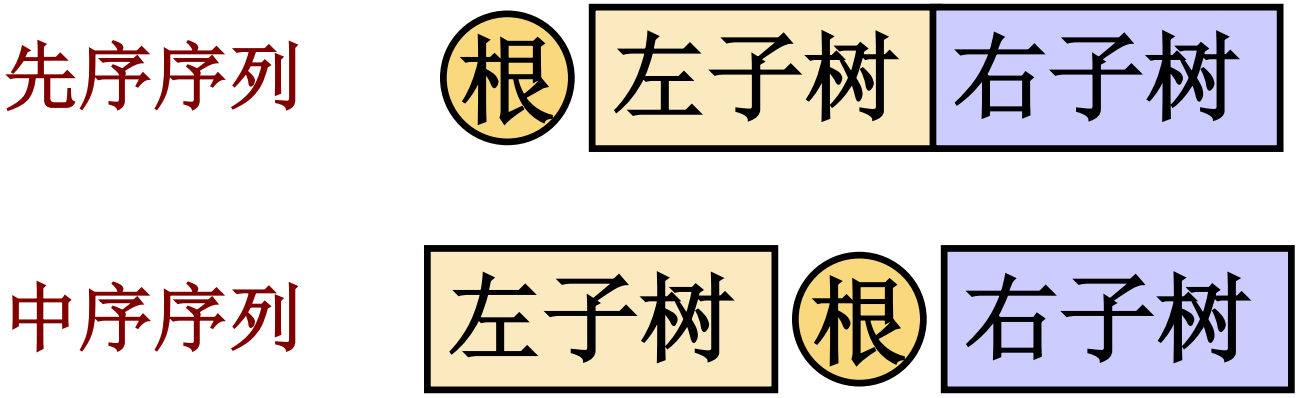
不能确定相邻字符之间的关系！不能区分左右！

先序序列



由先序和中序序列建二叉树

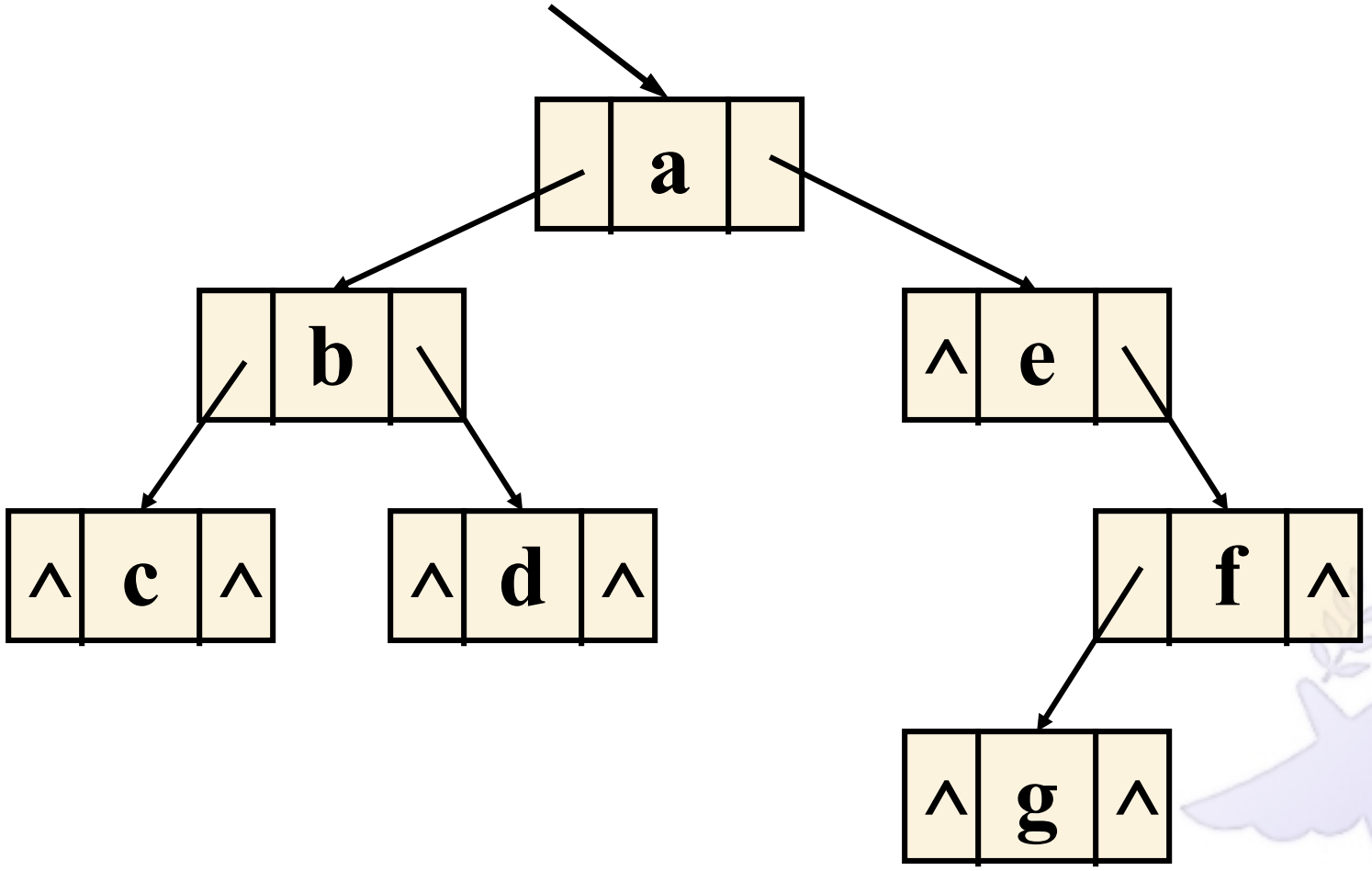
- 仅知二叉树的先序序列“abcdefg”不能唯一确定一棵二叉树
- 如果同时已知二叉树的中序序列“cbdaegf”，则会如何？



例如：

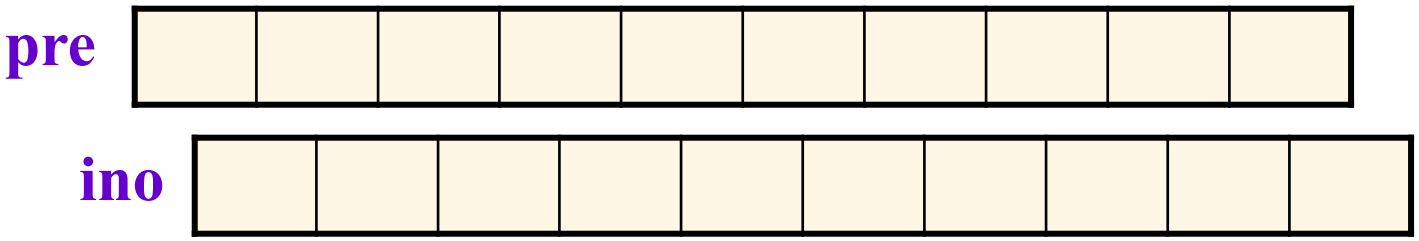
a b c d e f g　先序序列

c b d a e g f　中序序列



由先序和中序序列建二叉树算法

- void CrtBT(BiTree& T, char pre[], char ino[],
  int ps, int is, int n)
- 参数：T 构建的二叉树；
  - pre[]先序序列字符串；ino[]中序序列 字符串；
  - ps先序序列字符串第一个字符的位置；
  - is中序序列字符串第一个字符的位置；
  - n字符串长度。 pre[]和ino[]等长！



♦void CrtBT(BiTree& T, char pre[], char ino[],

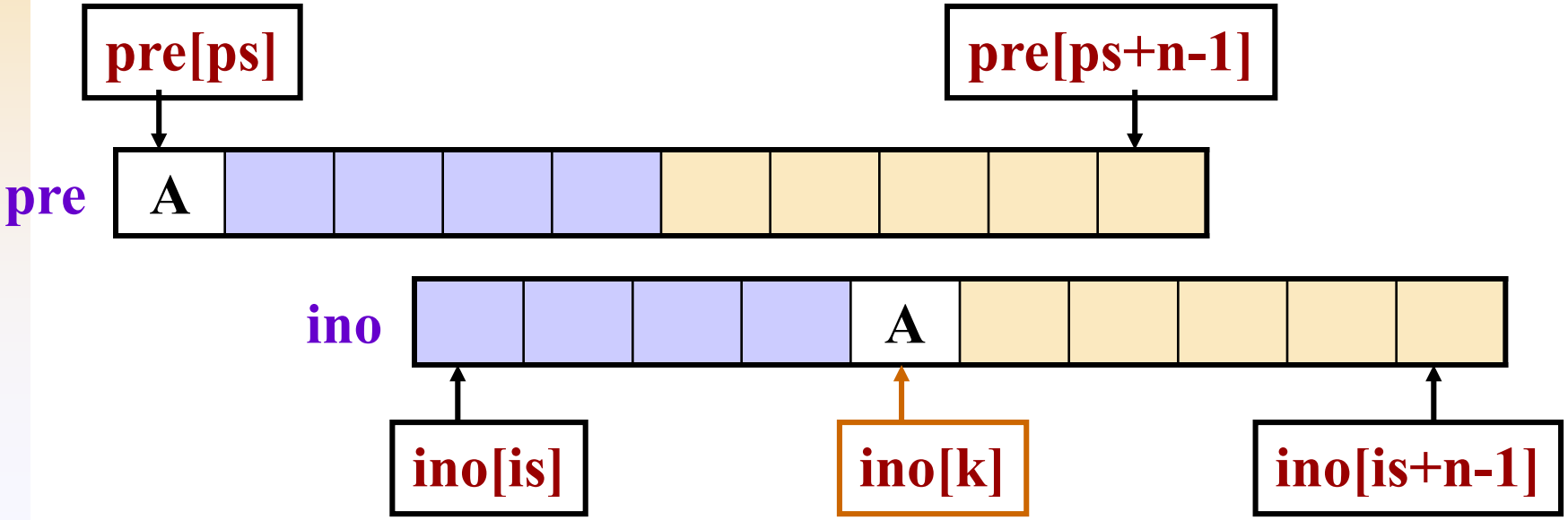
int ps, int is, int n)

1、取出pre中第一个字符A，并在中序序列中查询对应字符的位置k;

2、若能找不到对应字符，则返回错误;

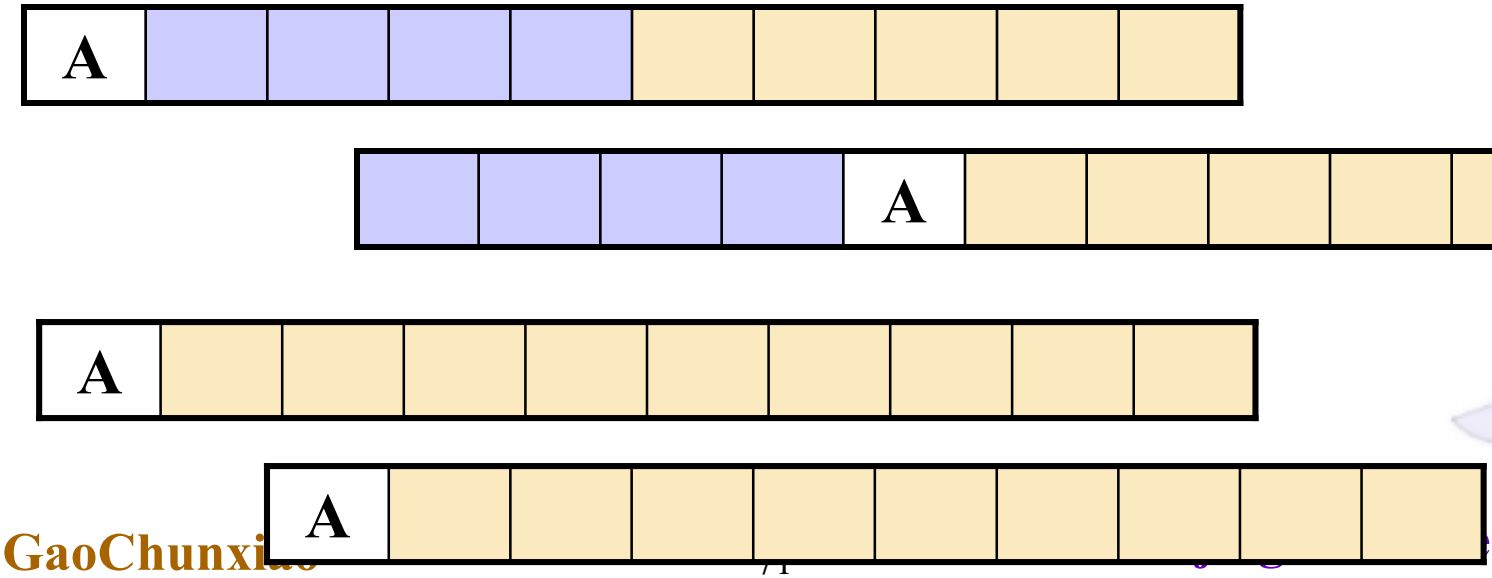
否则，1)建立根结点;

2)递归的建立左右子结点;



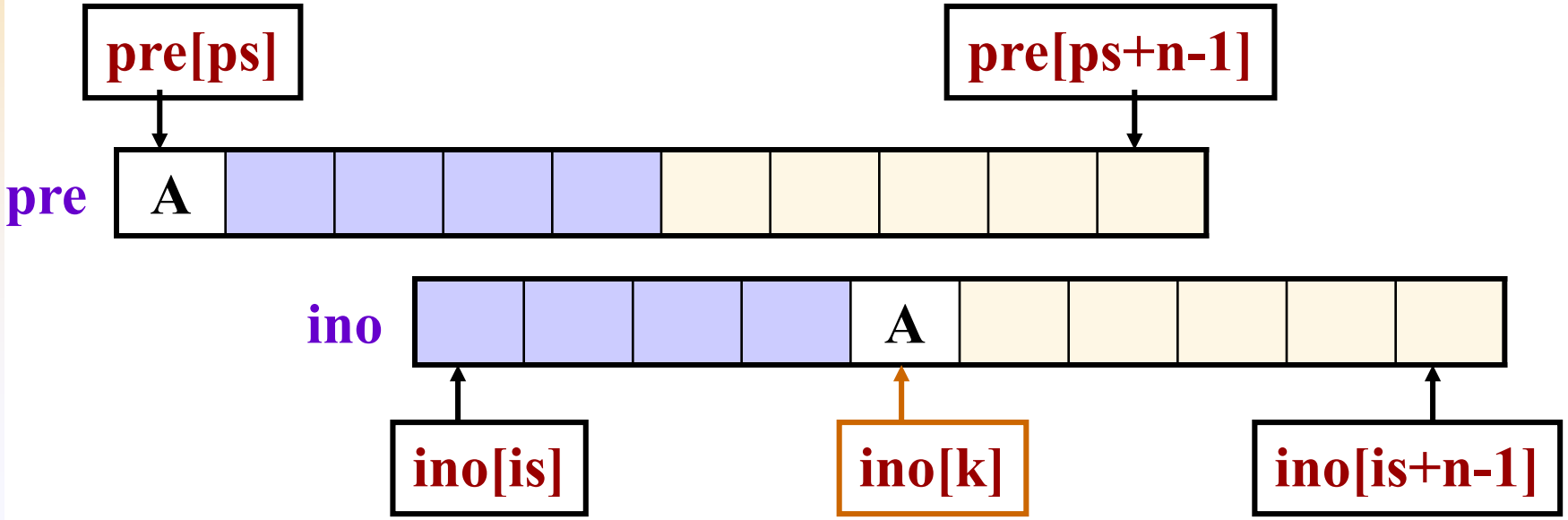
2)递归的建立左右结点；

- 需要决定是否建立左右子树，并确定递归调用时的参数。分情况讨论：
  - ¶ i→左右子树都存在；
  - ¶ ii→无左子树
  - ¶ iii→无右子树



2)递归的建立左右结点；a→如果左右子树都存在

左子树字符串长度：	k-is
在pre中的起点：	pre[ps+1]
在ino中的起点：	ino[is]
右子树字符串长度：	n-(k-is)-1
在pre中的起点：	pre[ps+(k-is) +1]
在ino中的起点：	ino[k+1]

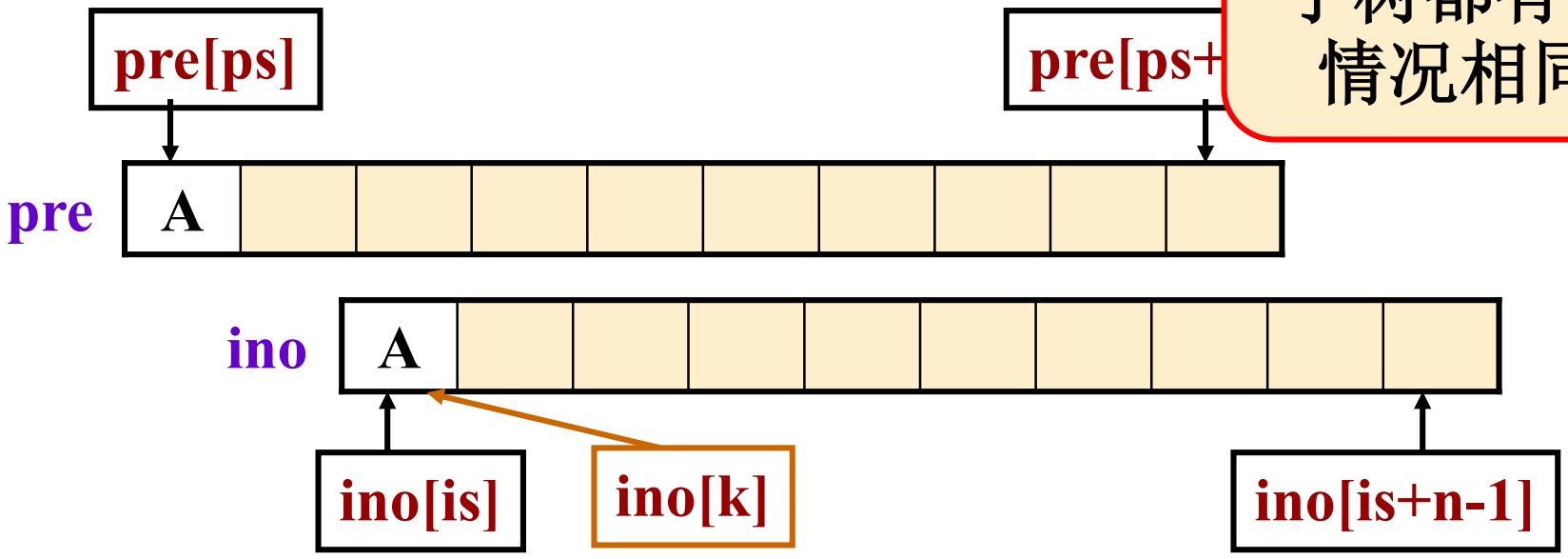


2)递归的建立左右结点； b→如果无左子树

无左子树条件：	k==is

	无左子树	左右子树都存在
右子树字符串长度：	n-1	n-(k-is)-1
在pre中的起点：	pre[ps+1]	pre[ps+(k-is) +1]
在ino中的起点：	ino[k+1]	ino[k+1]

参数与左右子树都有的情况相同

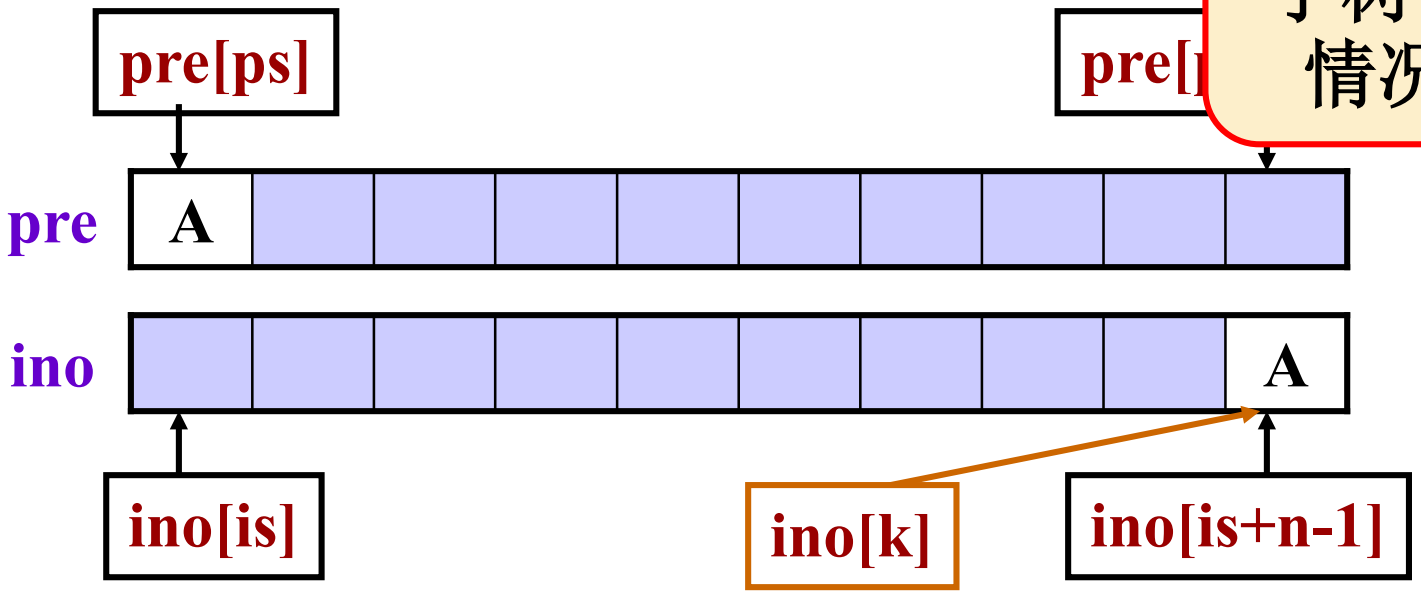


2)递归的建立左右结点； c→如果无右子树

无右子树条件：	k==is+n-1

	无右子树	左右子树都存在
左子树字符串长度：	n-1	k-is
在pre中的起点：	pre[ps+1]	pre[ps+1]
在ino中的起点：	ino[is]	ino[is]

参数与左右子树都有的情况相同



由先序和中序序列建二叉树算法

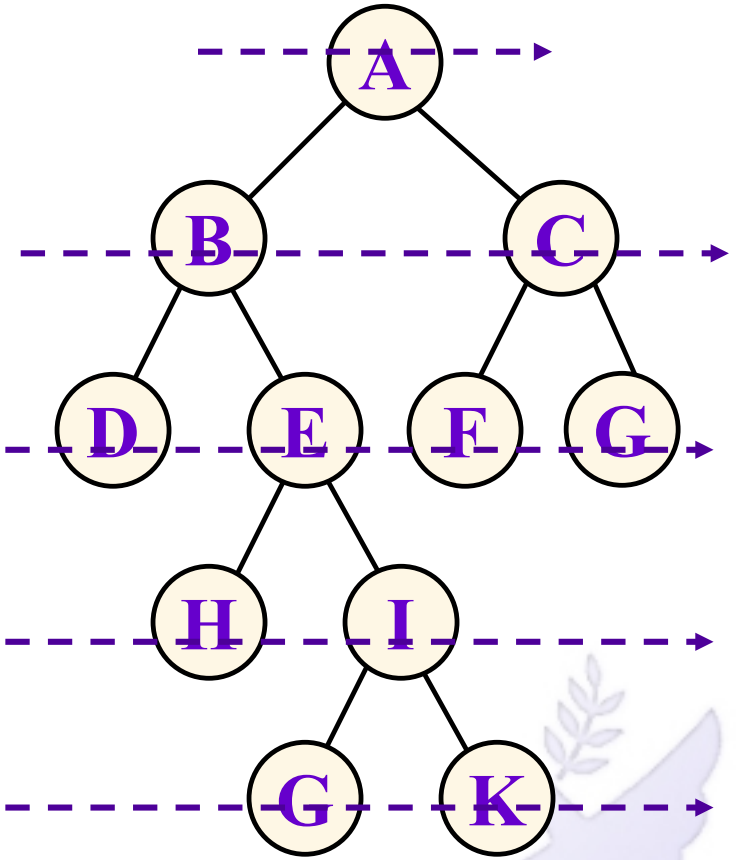
```
void CrtBT(BiTree& T, char pre[], char ino[],
                                  int ps, int is, int n ) {
  // pre[ps..ps+n-1]为二叉树的先序序列
  // ino[is..is+n-1]为二叉树的中序序列
    if (n==0) T=NULL; //递归终止条件
    else {
          k = Search(ino, pre[ps]); // 在中序序列中查询
          if (k== -1)  T=NULL;//若查不到，参数错误
          else {建立新结点，并递归建立左右子树 }
    } //if
} // CrtBT
```

由先序和中序序列建二叉树算法(续)

```
//建立新结点，并判断是否具有左右子树
 T=(BiTNode*)malloc(sizeof(BiTNode));
 T->data = pre[ps];  //建立新结点
 if (k==is)  T->Lchild = NULL;//无左子树
 else  CrtBT(T->Lchild, pre[], ino[],
                        ps+1, is, k-is );//递归创建左子树
 if (k==is+n-1) T->Rchild = NULL;//无右孩子
 else  CrtBT(T->Rchild, pre[], ino[],
       ps+1+(k-is), k+1, n-(k-is)-1 ); //递归创建右子树
```

二叉树的层次序遍历

- 按层次顺序访问二叉树需要利用一个队列。
- 在访问二叉树的某一层结点时，把下一层结点指针预先记忆在队列中，利用队列安排逐层访问的次序。



二叉树的层次序遍历

```
void levelOrder ( BiTree BT, void (* visit)(BiTNode *) ) {
//利用队列实现二叉树BT的层次序遍历。
   IniQueue(Q);  p = BT;
   EnQueue(Q, p);       //根结点入队
    while ( ! EmptyQueue(Q)) {    //队列不空时
          p = DeQueue(Q); visit(p);  //队首出队访问
          if ( p->lchild != NULL ) //若有左子结点，入队列
                EnQueue(Q, p->lchild);
          if ( p->rchild != NULL ) {//若有右子结点，入队列
                EnQueue(Q, p->rchild);
    }//while
}//levelOrder
```

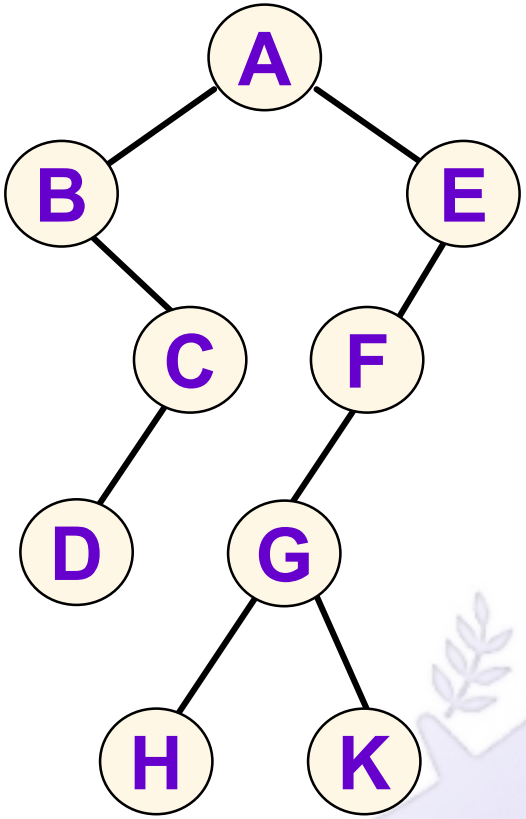
二叉树遍历算法的复杂度

算法	时间复杂度	空间复杂度
先/中/后序遍历的递归算法	O(n)	最坏：O(n) 平均：O(logn)
先/中/后序遍历的非递归算法	O(n)	最坏：O(n) 平均：O(logn)
层次遍历算法	O(n)	取决于树的宽度 最坏：O(n)

5.5 线索二叉树

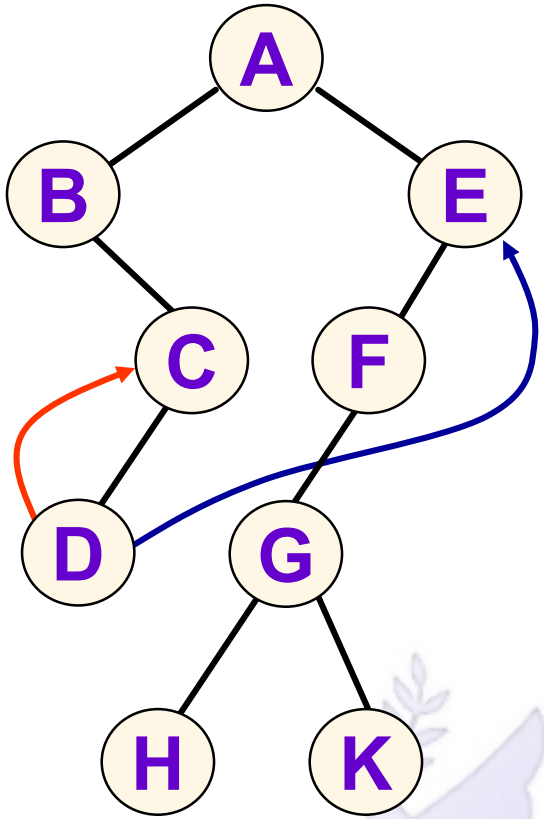
- 遍历二叉树的结果是结点的一个线性序列

先序遍历（DLR）
A B C D E F G H K
中序遍历（LDR）
B D C A H G K F E
后序遍历（LRD）
D C B H K G F E A



什么是线索树？

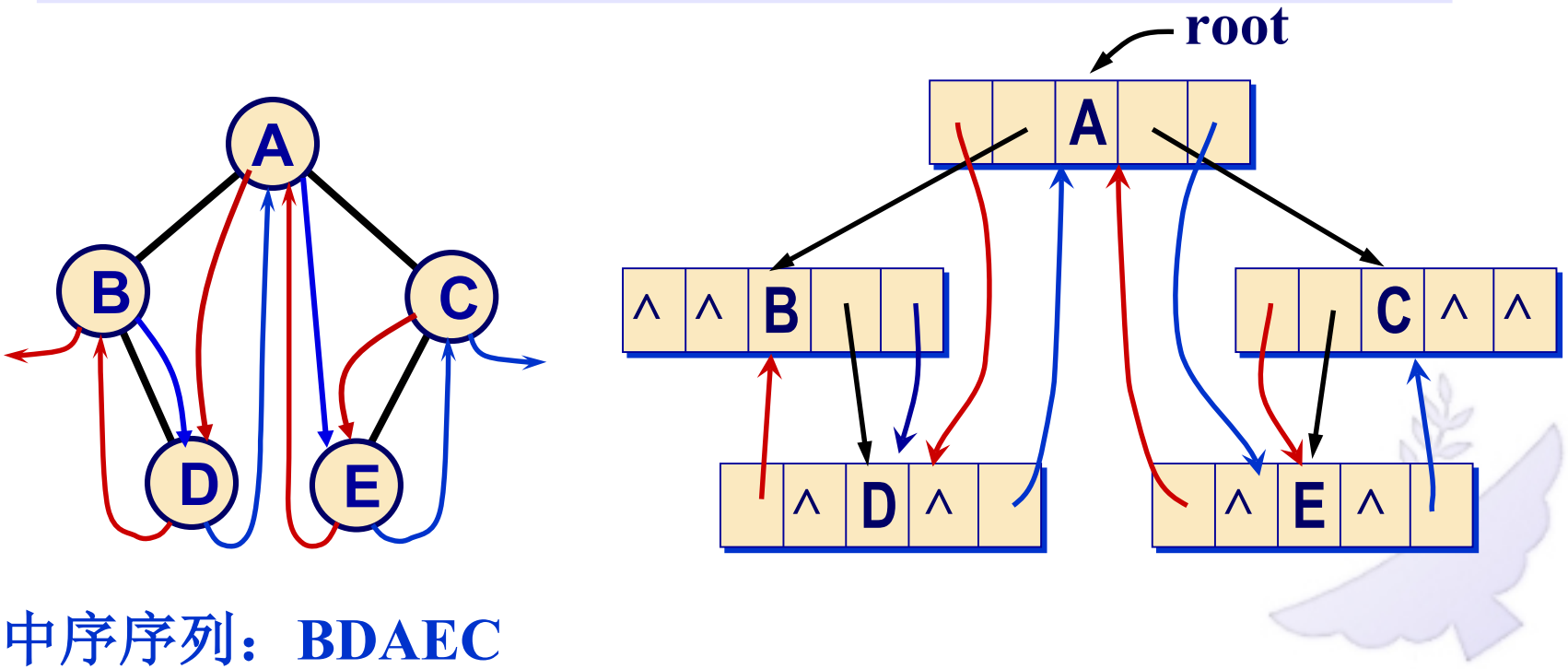
- 先序序列：A B C D E F G H K
- 指向该线性序列中的“前驱”和“后继”的指针，称作“线索”
- 包含“线索”的存储结构，称作“线索链表”
- 与其相应的二叉树，称作“线索二叉树”



线索如何保存？

- 方法1：增加前趋Pred指针和后继Succ指针的

pred	lchild	data	rchild	succ



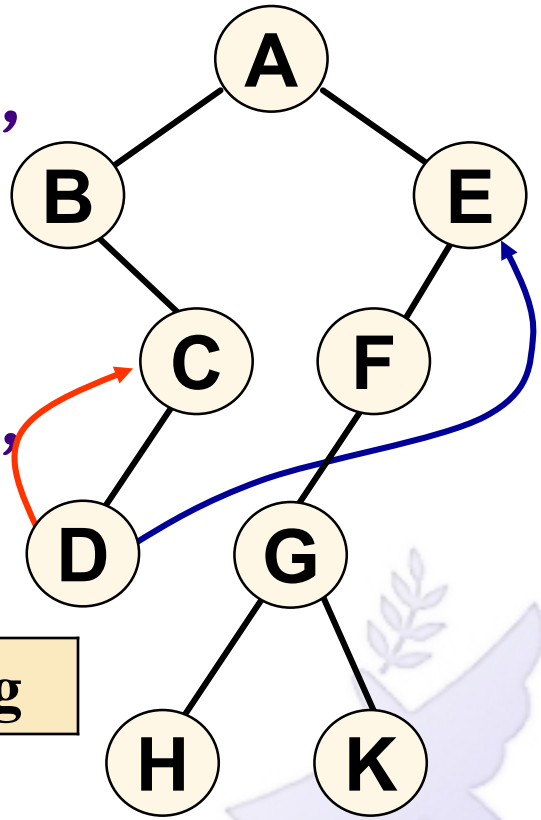
中序序列：BDAEC

pred succ

线索链表

先序序列：A B C D E F G H K

- 方法2：在二叉链表结点中增加两个标志域，并规定：
- 若该结点的左子树不空，则
  - ¶ Lchild域的指针指向其左子结点，
  - ¶ 且LTag的值为“Link”；
- 否则，
  - ¶ Lchild域的指针指向其“前驱”，
  - ¶ 且LTag的值为“Thread”。



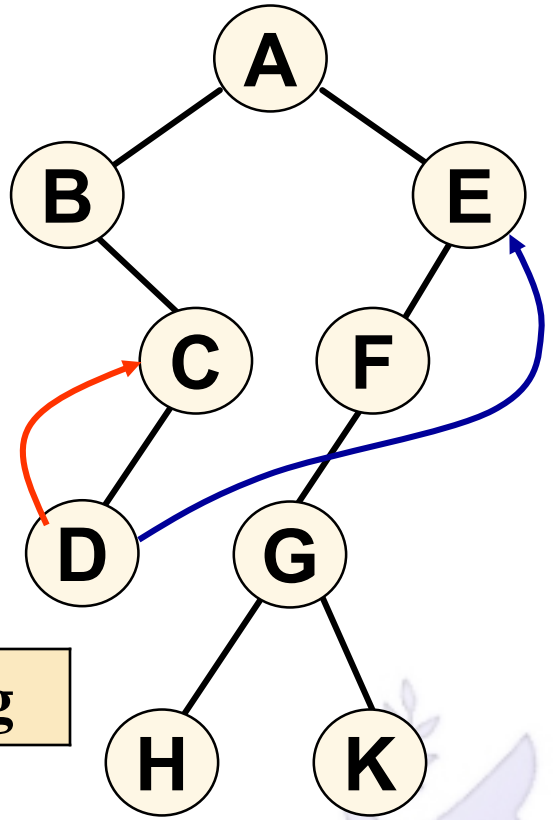
LTag	Lchild	Data	Rchild	RTag

左空指针指向前驱

线索链表

先序序列：A B C D E F G H K

- 若该结点的右子树不空，则
  - ¶ rchild域的指针指向其右子结点，
  - ¶ 且RTag的值为“Link”；
- 否则，
  - ¶ rchild域的指针指向其“后继”，
  - ¶ 且RTag的值为“Thread”。



LTag	Lchild	Data	Rchild	RTag

左空指针指向前驱

右空指针指向后继

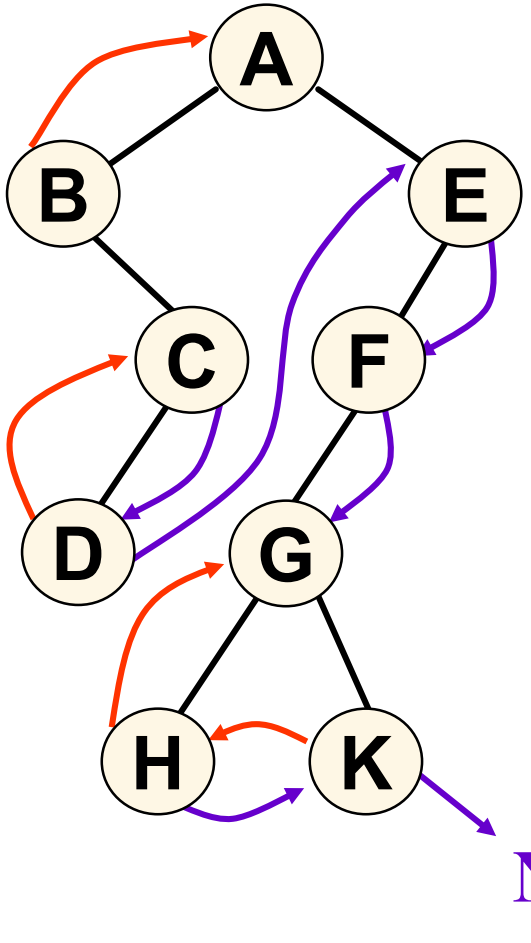
线索链表举例：先序线索链表

先序序列：

A B C D E F G H K

左空指针指向前驱

右空指针指向后继



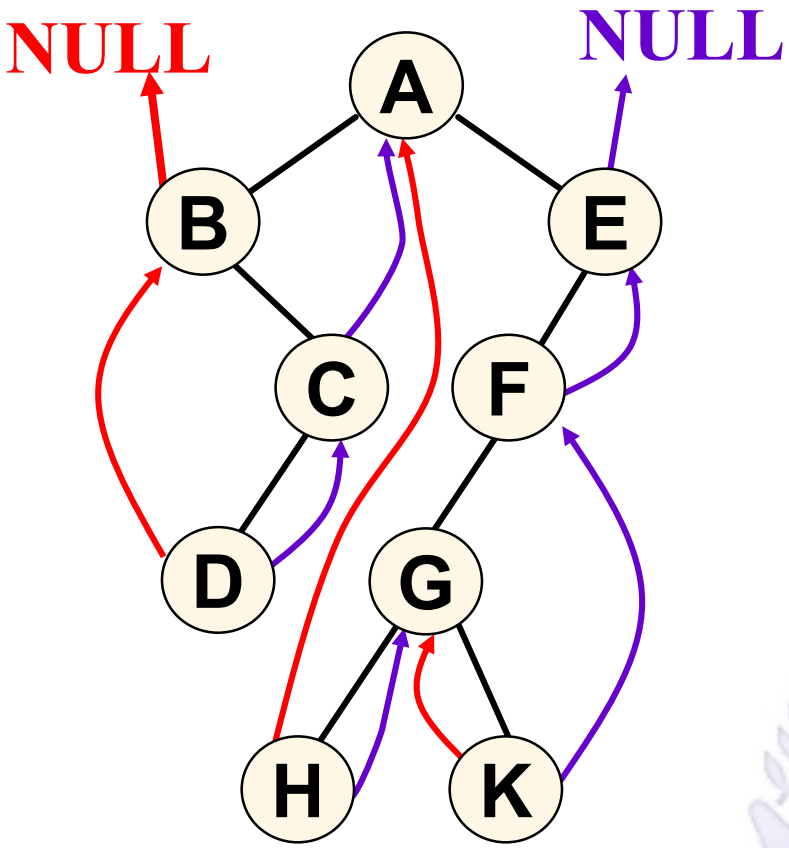
线索链表举例：中序线索链表

中序序列：

B D C A H G K F E

左空指针指向前驱

右空指针指向后继



线索链表的类型描述

```
typedef enum { Link, Thread } PointerThr;
  // Link==0:指针，Thread==1:线索
```

```
typedef struct ThreadNode {
  TElemType       data;
  struct ThreadNode  *lchild, *rchild;  // 左右指针
  PointerThr      LTag,  RTag;    // 左右标志
} ThreadNode, *ThreadTree;
```

线索链表的遍历算法

- 在线索链表中添加了“前驱”和“后继”的信息
- 所以遍历时与线性表的遍历相似：
  1. 找到第一个结点
  2. 依次找到后继结点

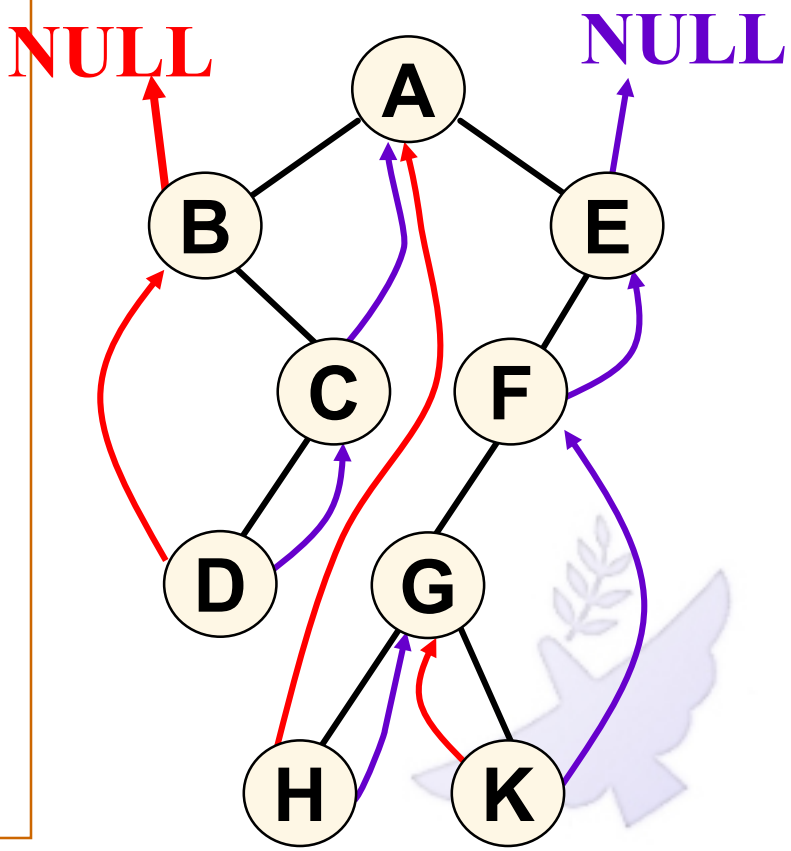
```
for ( p = firstNode(T); p; p = Succ(p) )
    Visit (p->data);
```

线索链表的中序遍历算法

- 中序遍历的第一个结点？
  - 左子树上处于“最左下”（没有左子树）的结点。
- 在中序线索化链表中结点的后继？
  - 若没有右子树，为后继线索所指结点。
  - 否则为对其右子树进行中序遍历时访问的第一个结点；

中序：

B D C A H G K F E



线索链表的中序遍历算法（续）

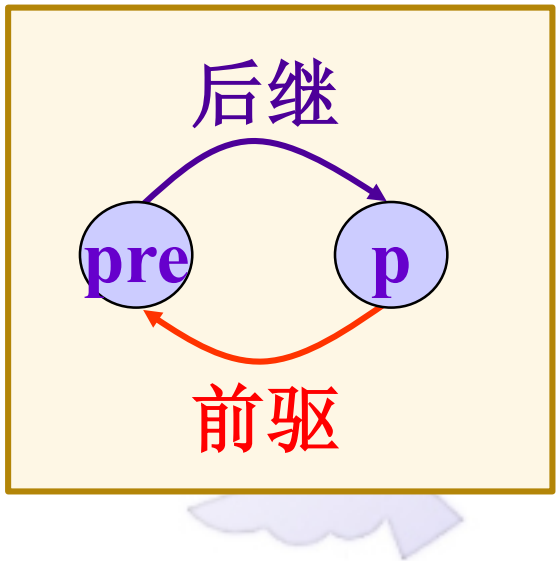
```
ThreadNode *firstNode(ThreadTree T) {
    if (T == NULL) return NULL;
    p = T;//指向根结点
    while (p->LTag == Link)
        p = p->lchild;//找最左下的结点
    return p;
}// firstNode
```

线索链表的中序遍历算法（续）

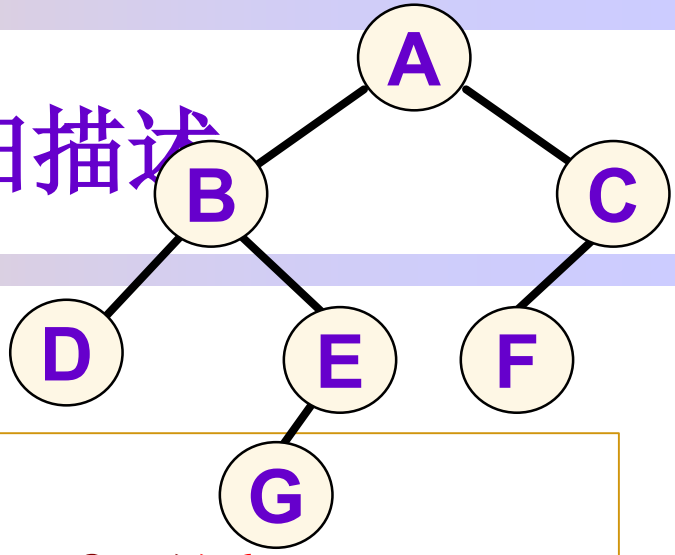
```
ThreadNode *Succ(ThreadNode *p) {
        if (p == NULL) return NULL;
        if (p->RTag==Thread)//没有右孩子
            return p->rchild;//直接取其后继
       return firstNode(p->rchild); //否则取以右孩子为根结点的子树中的第一个结点
} // Succ
```

如何建立线索链表（中序为例）？

- 基本思想：
  - 在中序遍历过程中修改结点的左、右指针域
  - 保存当前访问结点的“前驱”和“后继”信息。
- 遍历过程中
  - 设指针p：指向当前结点
  - 设指针pre：指向当前结点的前驱
- 处理当前结点p和前驱结点pre：
  - 设置结点pre和结点p的线索
  - 令pre指向当前结点p



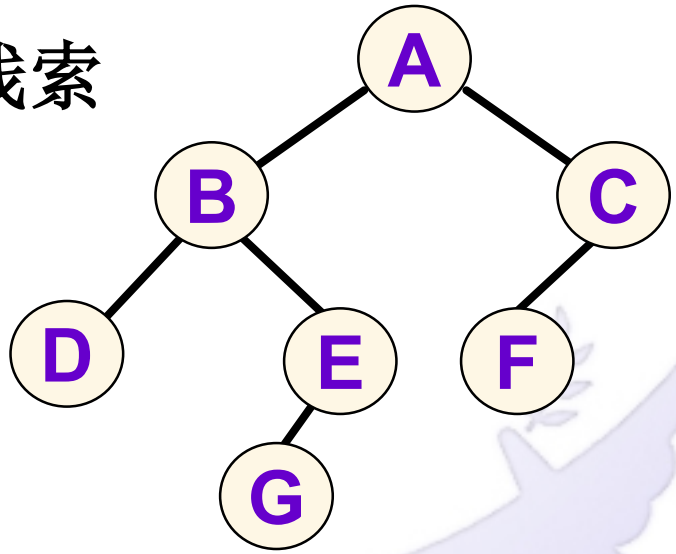
回顾：中序遍历算法的递归描述



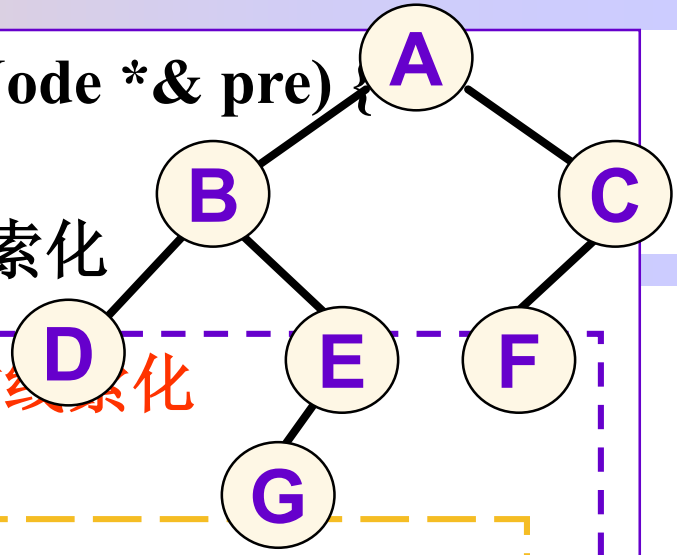

```
void Inorder (BiTree T,
              void( *visit)(TElemType& e)){
   if (T) {
      Inorder(T->lchild, visit); //中序遍历左子树
      visit(T->data);            // 访问根结点
      Inorder(T->rchild, visit);//中序遍历右子树
   }//if(T)
}// Inorder
```

如何建立线索链表（中序为例）？

- _InThreading(p, pre); //中序遍历线索化
- 基本思想：递归算法
  1. 对左子树进行线索化。
  2. 处理当前结点p
     - 设置结点pre和结点p的线索
     - 令pre指向当前结点p。
  3. 对右子树进行线索化。

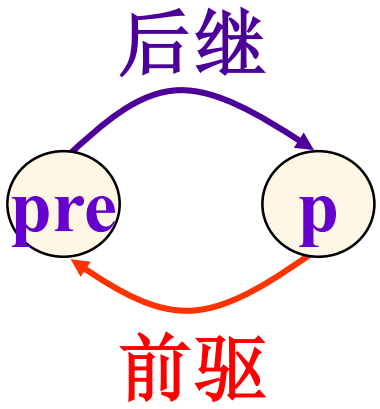


```
void _InThreading(ThreadTree p, ThreadNode *& pre) {
//递归程序
   if (p) {// 对以p为根的非空二叉树进行线索化
      _InThreading(p->lchild, pre);// 左子树线索化

        //处理当前结点
        //1、设置结点pre和结点p的线索

        //2、令pre指向当前结点p。
        pre = p; // 保持 pre 指向 下一个处理的p 的前驱

      _InThreading(p->rchild, pre);// 右子树线索化

   } // if (p)
} // InThreading
```

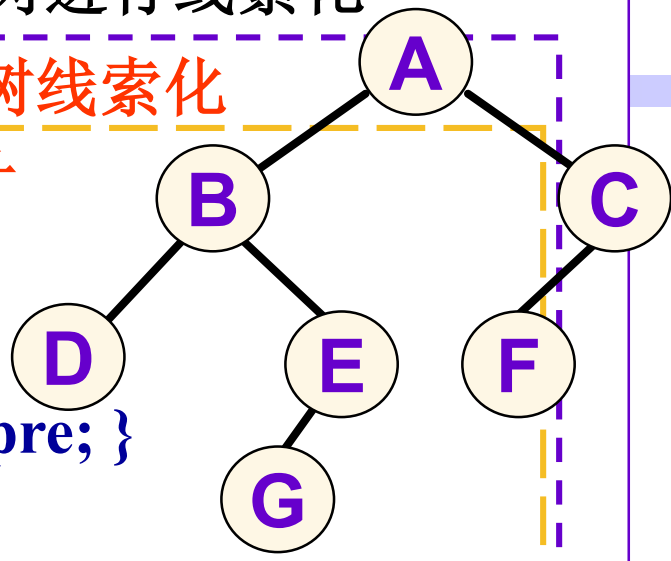


如何建立线索链表？

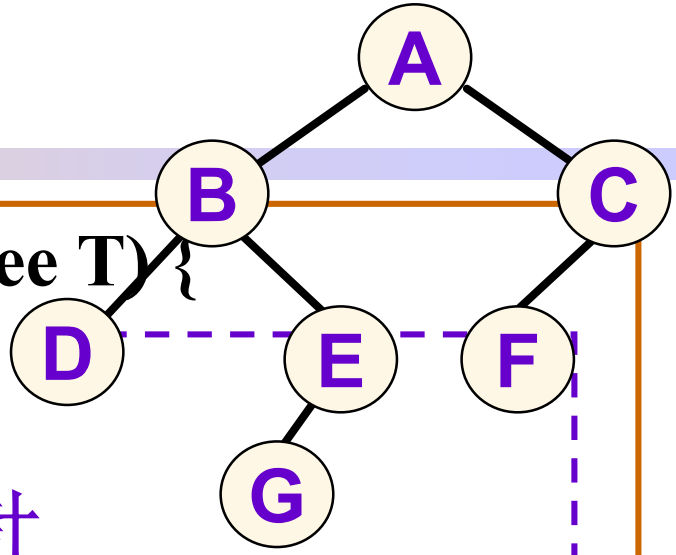
- 2. 处理当前结点p
- 1) 若p的左指针不为空，则
  - p->LTag = Link;
- 否则建p的前驱线索
  - p->LTag = Thread;
  - p->lchild = pre;
- 2) 若前驱pre不为空
  - 若pre的右指针不为空
  - pre->RTag = Link;
  - 否则建pre的后继线索
  - p->RTag = Thread;
  - pre->rchild = p;
- 3) pre = p;//更新pre指针



```
void _InThreading(ThreadTree p, ThreadNode *& pre) { //递归
   if (p != NULL) {// 对以p为根的非空二叉树进行线索化
      _InThreading(p->lchild, pre);// 左子树线索化
      if (p->lchild)//判断当前结点的左指针
                p->LTag = Link;
      else // 建前驱线索, pre是p的前驱
          { p->LTag = Thread; p->lchild = pre; }
      if (pre) {// 判断前驱结点的右指针
            if (pre->rchild)    pre->RTag = Link;
           else //建后继线索, p是pre的后继
                { pre->RTag = Thread; pre->rchild = p;}
      }// if (pre)
      pre = p; // 保持 pre 指向 下一个处理的p 的前驱
      _InThreading(p->rchild, pre);// 右子树线索化
   } // if (p)
} // InThreading
```



如何建立线索链表？



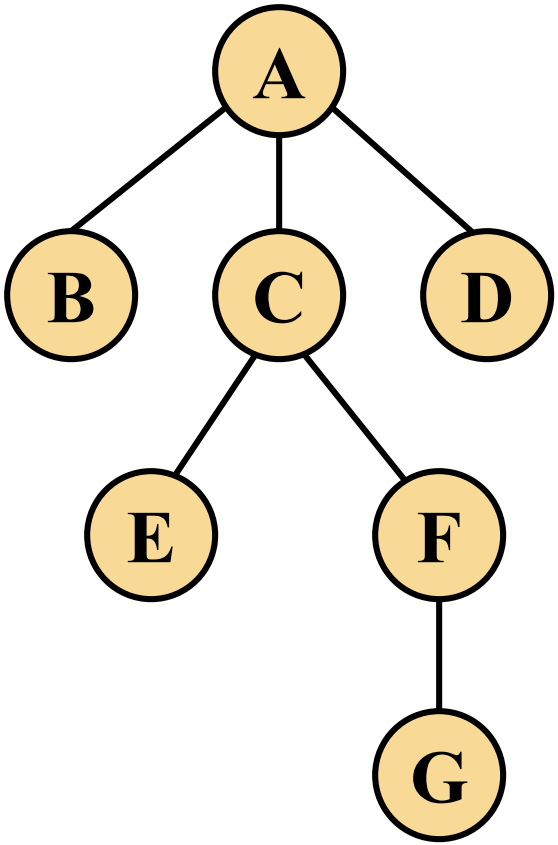

```
void InOrderThreading(ThreadTree T){
    if (!T) return;
    pre = NULL;//初始化前驱指针
    _InThreading(T, pre);//依次遍历处理所有结点
    pre->RTag = Thread; // 收尾
                //最后一个结点右子树为空
}// InOrderThreading
```

5.6 树和森林的表示方法

- **5.5.1 树的三种存储结构**
  1. 双亲表示法
  2. 孩子链表表示法
  3. 树的二叉链表(孩子-兄弟）存储表示法

5.6.1 树的三种存储结构

1) 双亲表示法



n=7

r=0

	data	parent
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	2
5	F	2
6	G	5

5.6.1 树的三种存储结构

双亲表示法的类型描述

结点结构：


```
#define MAX_TREE_SIZE  100
typedef struct PTNode {
    Elem  data;
    int    parent;   // 双亲位置域
  } PTNode;
```

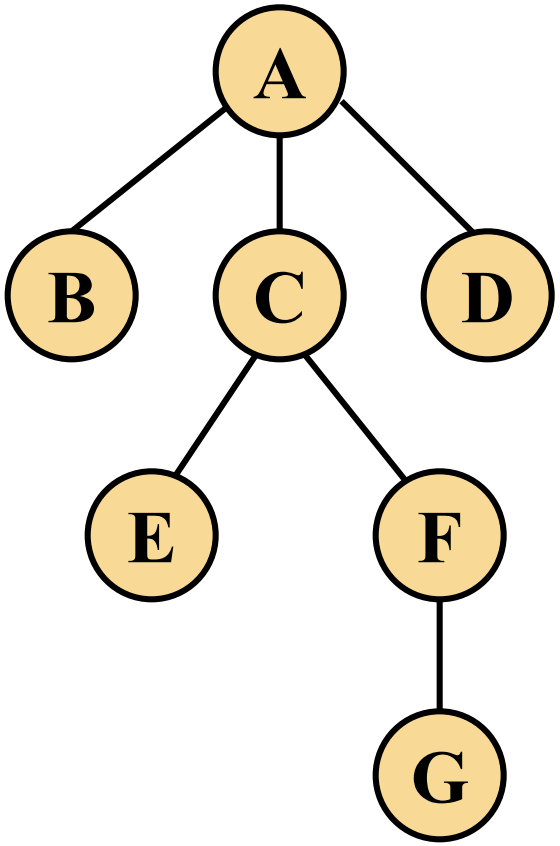
5.6.1 树的三种存储结构

双亲表示法的类型描述

树结构:

```
typedef struct {
    PTNode  nodes [MAX_TREE_SIZE];
    int    r, n; // 根结点的位置和结点个数
  } PTree;
```

2) 子女链表表示法



FC:First Child

	data	FC
0	A	→ 1 → 2 → 3 ^
1	B	^
2	C	→ 4 → 5 ^
3	D	^
4	E	^
5	F	→ 6 ^
6	G	^

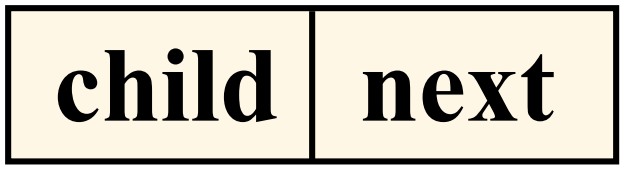
n=7

r=0

2) 子女链表表示法

孩子链表表示法的类型描述

孩子结点结构:




```
typedef struct CTNode {
    int    child;
    struct CTNode *next;
  } *ChildPtr;
```

2) 子女链表表示法

孩子链表表示法的类型描述

双亲结点结构


```
typedef struct {
  Elem    data;
  ChildPtr  firstchild; // 孩子链的头指针
} CTBox;
```

2) 子女链表表示法

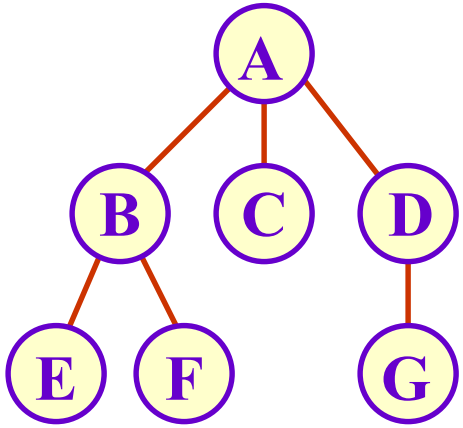
孩子链表表示法的类型描述

树结构:

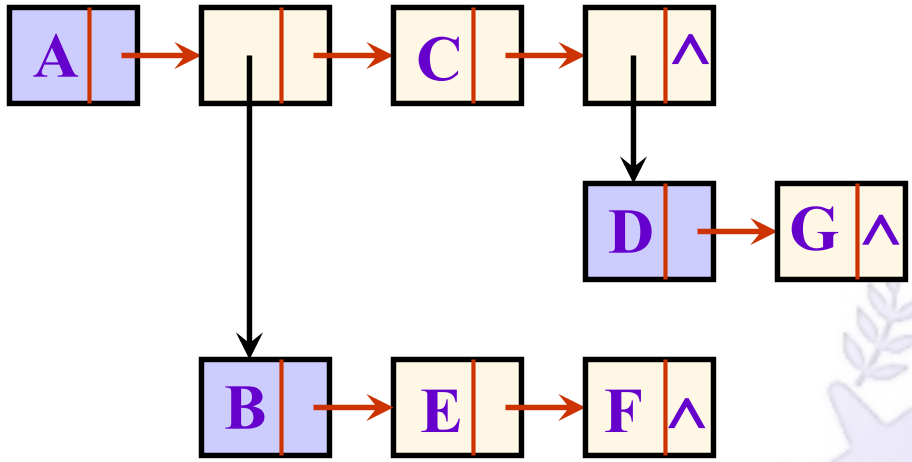
```
typedef struct {
    CTBox  nodes[MAX_TREE_SIZE];
    int    n, r;    // 结点数和根结点的位置
  } CTree;
```

3）广义表表示法

- 树的广义表一一对应
- 子表的头结点是子树的根，其后链接它所有子女。
- 如果子女是叶结点（单元素或原子），结点中直接赋值，否则结点中是子表的头指针。

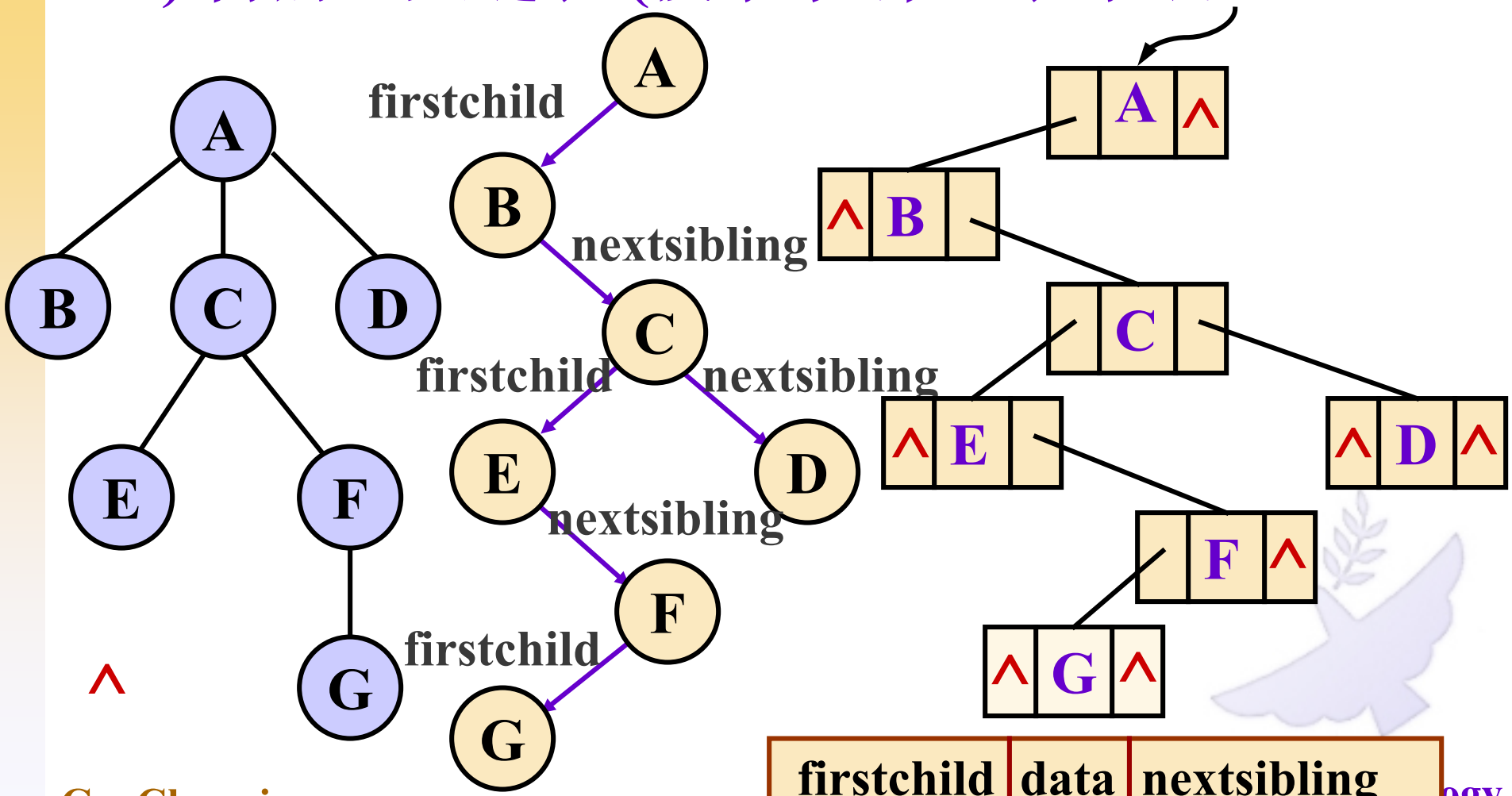


A(B(E, F), C, D(G))



5.6.1 树的三种存储结构

4) 树的二叉链表 (孩子-兄弟）表示法



5.6.1 树的三种存储结构

二叉链表类型描述

结点结构：


```
typedef struct CSNode{
    Elem        data;
    struct CSNode  *firstchild, *nextsibling;
} CSNode, *CSTree;
```

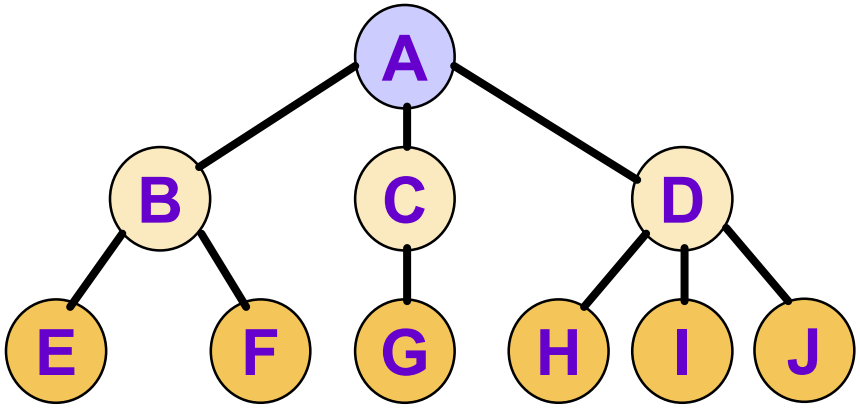
5.6.2 树与二叉树转换

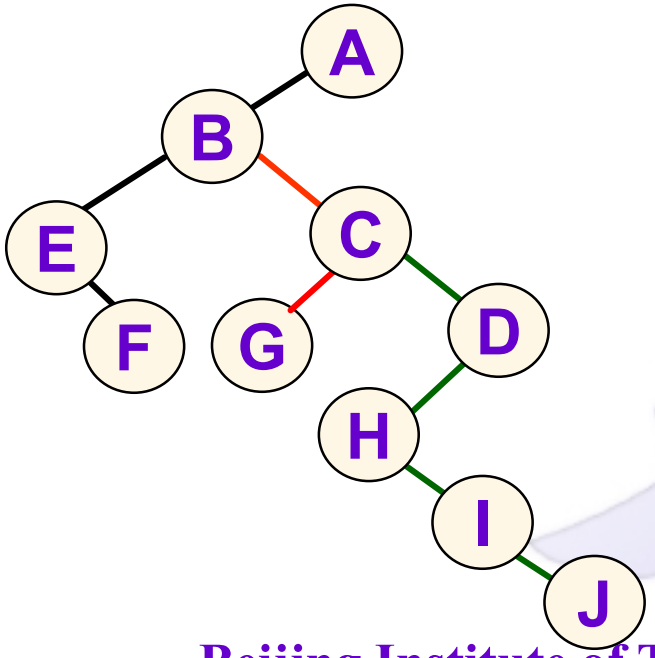
- 树与二叉树
  - 二叉树与树都可用二叉链表存贮，以二叉链表作中介，可实现树与二叉树之间的转换。
- 树与二叉树转换方法

树		二叉树
根	⟺	根
结点X的第一个孩子	⟺	结点 X 的左孩子
结点X紧邻的右兄弟	⟺	结点 X 的右孩子

5.6.2 树与二叉树转换

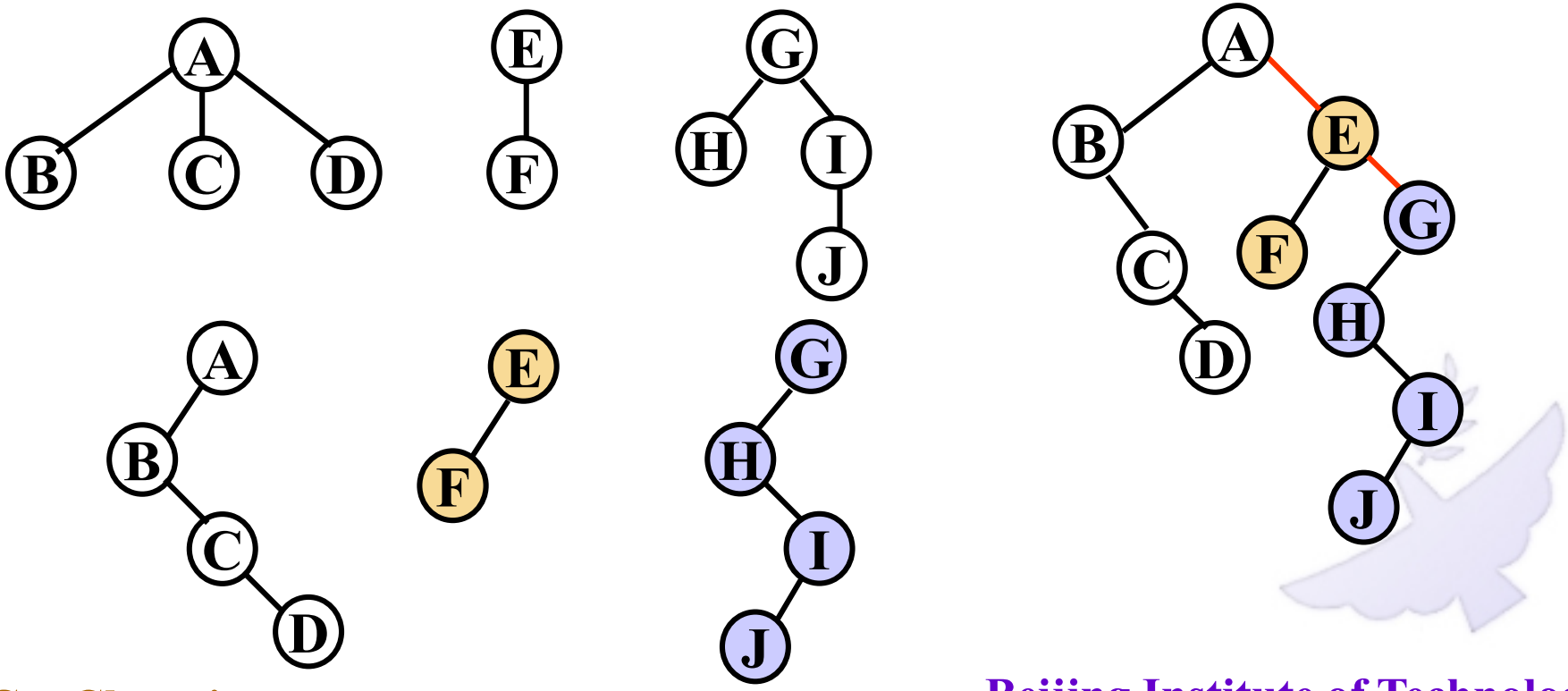
树		二叉树
根	⟺	根
结点X的第一个孩子	⟺	结点 X 的左孩子
结点X紧邻的右兄弟	⟺	结点 X 的右孩子





5.6.3 森林和二叉树的转换

- 森林：树的集合
- 将森林中树的根看成兄弟，用树与二叉树的转换方法，进行森林与二叉树转换。



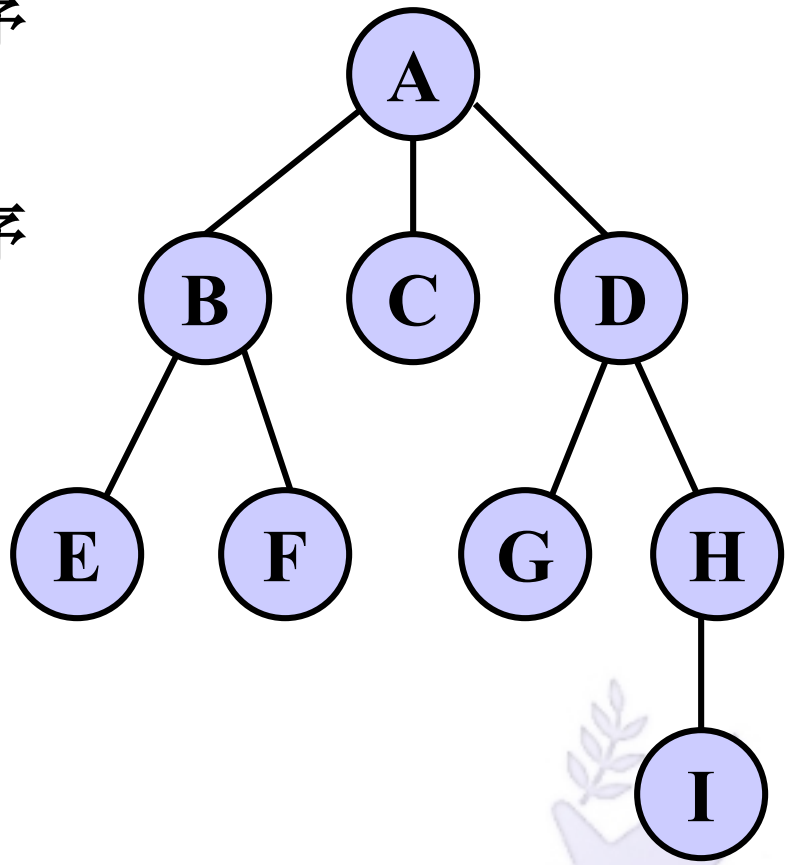
5.7 树和森林的遍历

♦5.7.1 树的遍历

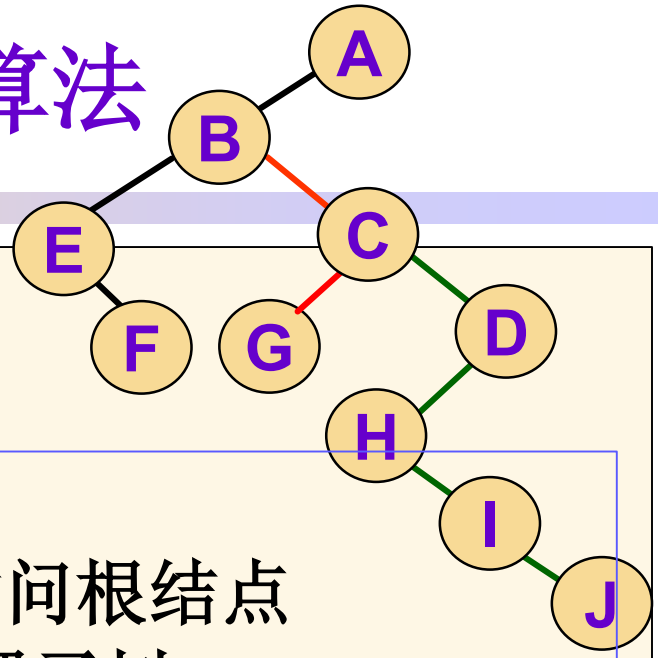
- 先根(序)遍历:
  - ¶ 若树不空，则先访问根结点，然后依次先根遍历各棵子树
- 后根(序)遍历
  - ¶ 若树不空，则先依次后根遍历各棵子树，然后访问根结点
- 按层次遍历
  - ¶ 若树不空，则自上而下自左至右访问树中每个结点

5.7.1 树的遍历

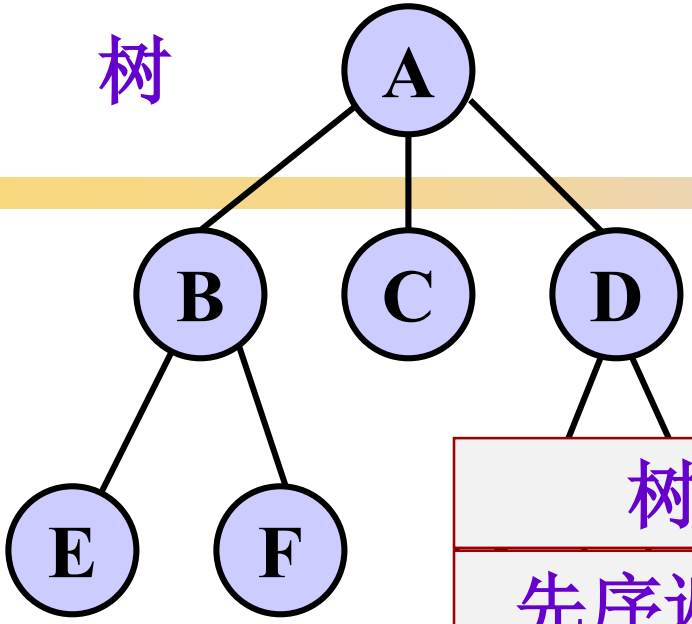
- 先序(根)遍历顶点的访问次序
  - ¶ABEFCDGHI
- 后序(根)遍历顶点的访问次序
  - ¶EFBCGIHDA
- 层次遍历时顶点的访问次序
- 也称作广度优先遍历
  - ¶ABCDEFGHI

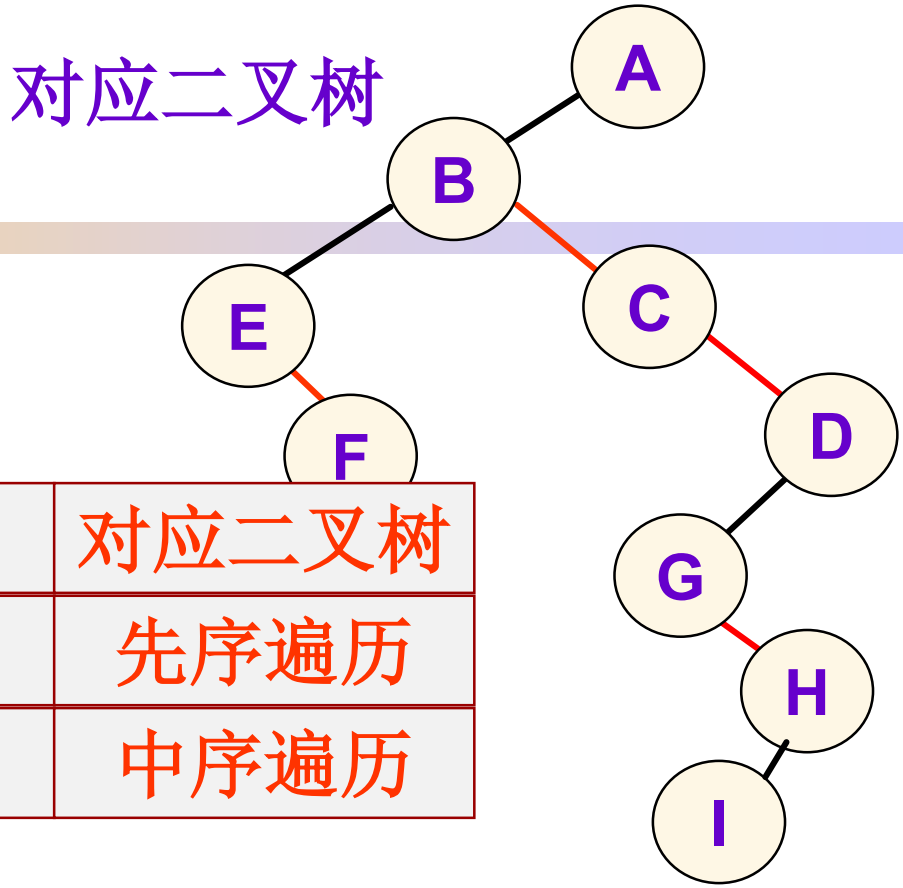


树的先根次序遍历的递归算法




```
void PreOrder ( CSNode *T ) {
//以指针 T 为根, 先根次序遍历
    if ( T != NULL ) {
            visit ( T->data );        //访问根结点
            p = T->lchild;      //第一棵子树
            while ( p != NULL ) { //依次遍历各棵子树
                PreOrder (p); //递归先根遍历子树
                p = p->rsibling;
            }// while
    }//if

}//PreOrder
```





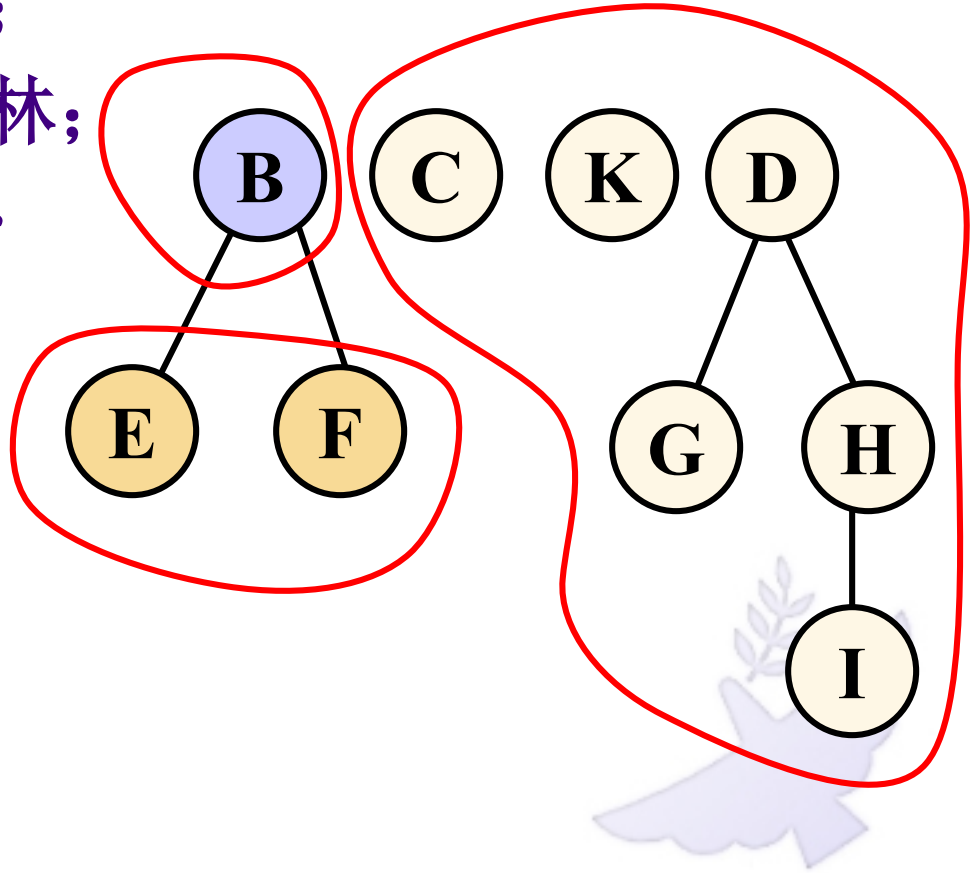
树	对应二叉树
先序遍历	先序遍历
后序遍历	中序遍历

- 树
- 先序:ABEFCDGHI
- 后序:EFBCGIHDA

- 对应二叉树
- 先序:ABEFCDGHI
- 中序: EFBCGIHDA

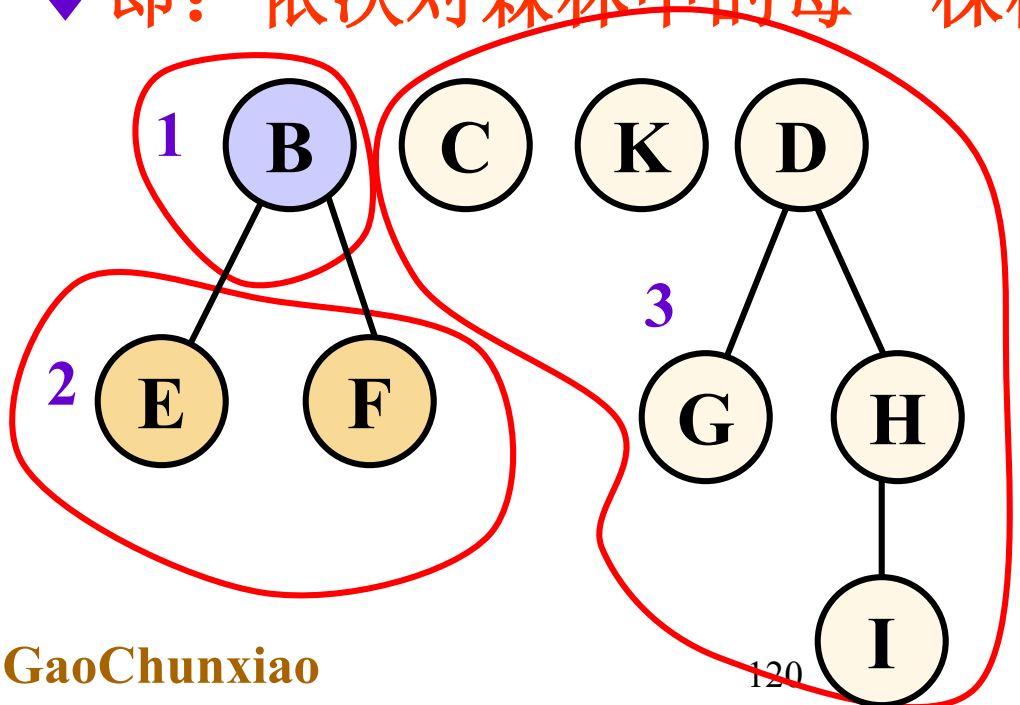
5.7.2 森林的遍历

- 森林由三部分构成：
  1. 第一棵树的根结点；
  2. 第一棵树的子树森林；
  3. 其它(除第一棵树外的)树构成的森林。



5.7.2 森林的遍历

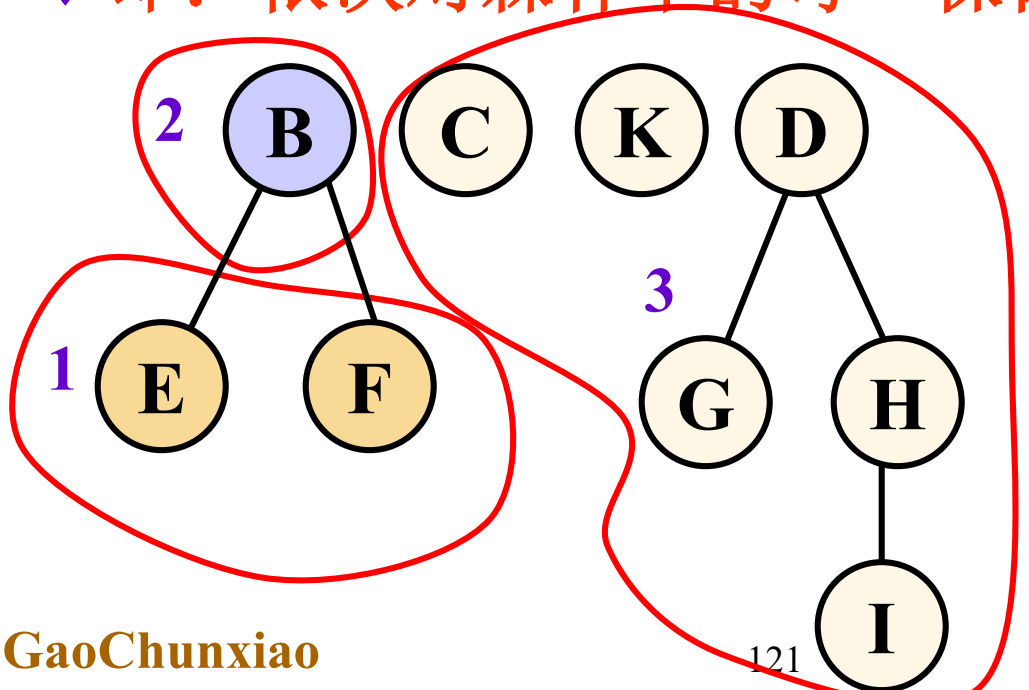
- 1. 先根次序遍历：若森林不空，则
  - ¶ 访问森林中第一棵树的根结点；
  - ¶ 先序遍历森林中第一棵树的子树森林；
  - ¶ 先序遍历森林中其余树构成的森林
- 即：依次对森林中的每一棵树进行先序遍历。



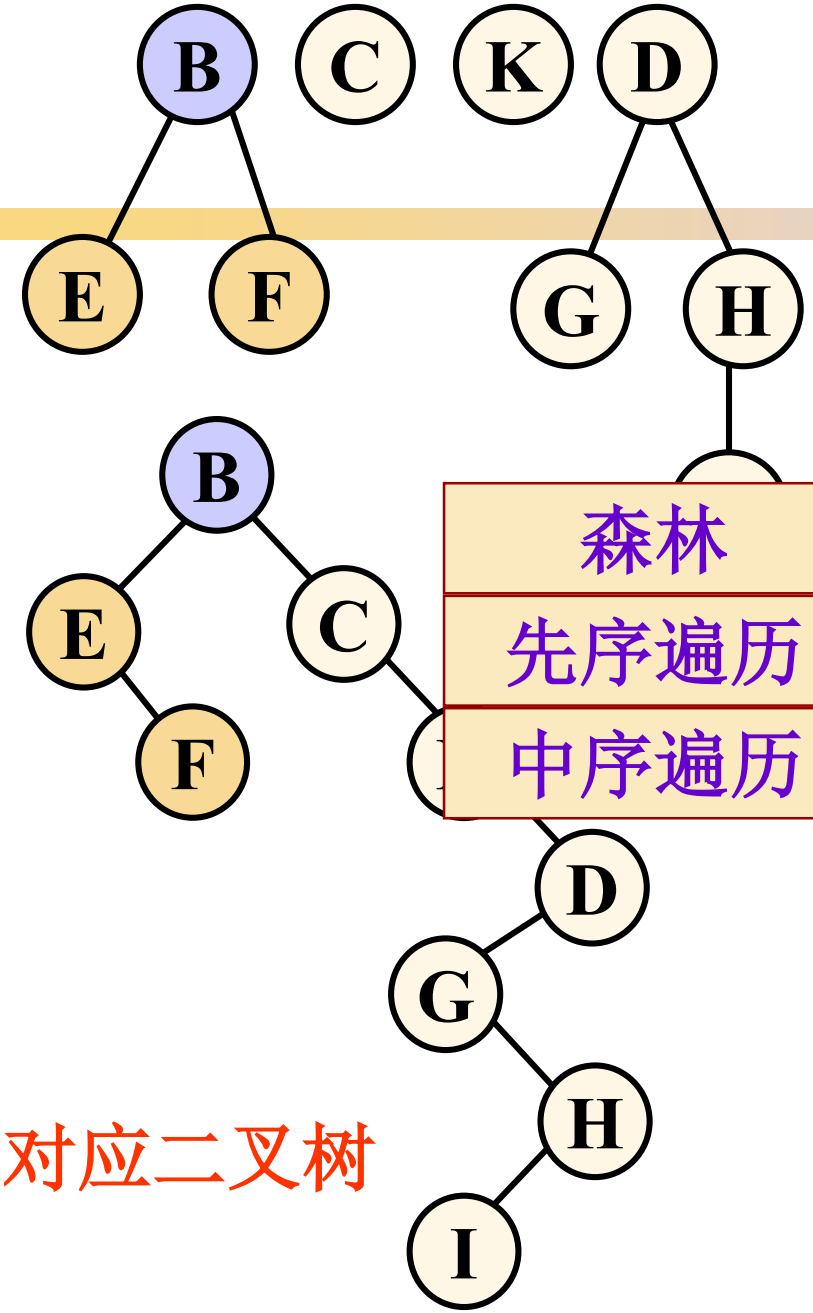
BEF－C－K－DGHI

5.7.2 森林的遍历

- 2. 中根次序遍历：若森林不空，则
  - 中序遍历森林中第一棵树的子树森林；
  - 访问森林中第一棵树的根结点；
  - 中序遍历森林中其余树构成的森林
- 即：依次对森林中的每一棵树进行后序遍历。



EFB－C－K－GIHD



对应二叉树

森林：先序遍历

BEF－C－K－DGHI

森林：中序遍历

EFB－C－K－GIHD

森林	对应二叉树
先序遍历	先序遍历
中序遍历	中序遍历

树：先序遍历

BEF－C－K－DGHI

对应二叉树：中序遍历

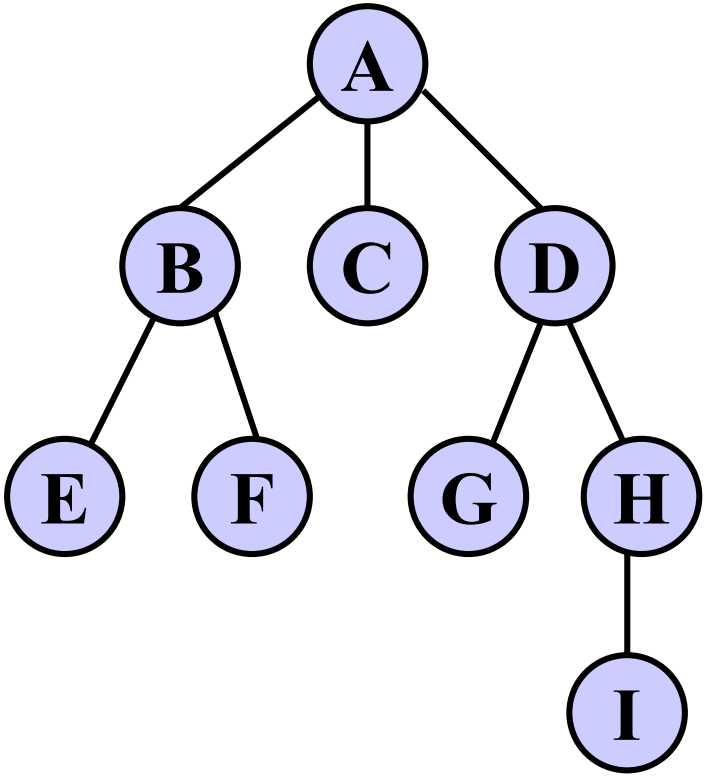
EFB－C－K－GIHD

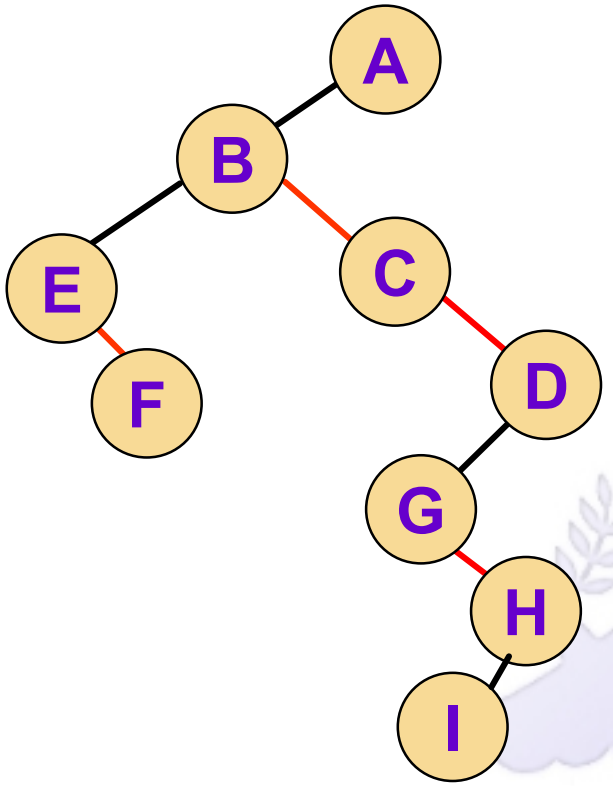
- 树、森林与二叉树遍历的关系

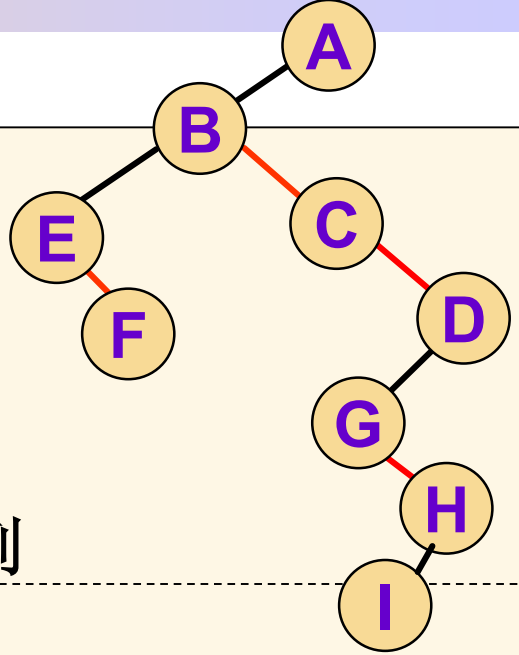
树	对应二叉树	森林
先序遍历	先序遍历	先序遍历
后序遍历	中序遍历	中序遍历

树和森林的广度优先遍历

- 广度优先遍历不是递归过程
- 类似二叉树的层次遍历
- 借助队列实现遍历过程







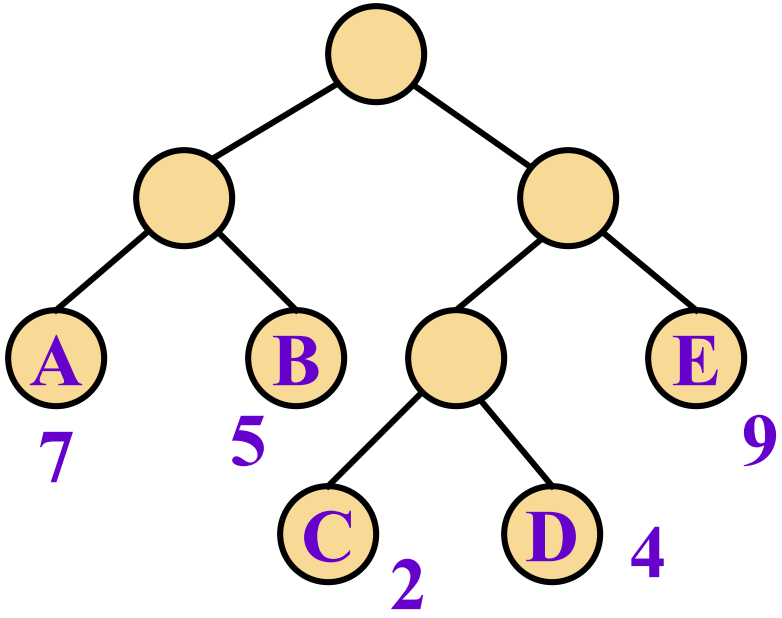

```
void LevelOrder ( CSTree T ) {
   //分层遍历树，算法用到一个队列
      Queue Q;   InitQueue(Q);
      if ( T != NULL ) {        //树不空
         EnQueue(Q, T);     //根结点进队列
         while ( ! QueueEmpty(Q) ) {
            DeQueue(Q, p);   visit(p->data);//出队并访问
            p = p->lchild; //找到第一个子结点
            while ( p != NULL ) { //结点的子结点依次进队列
               EnQueue ( Q, p );  p = p->rsibling;
            }//while ( p != NULL )
         }//while ( ! QueueEmpty(Q) )
      }//if( T != NULL )
}//LevelOrder
```

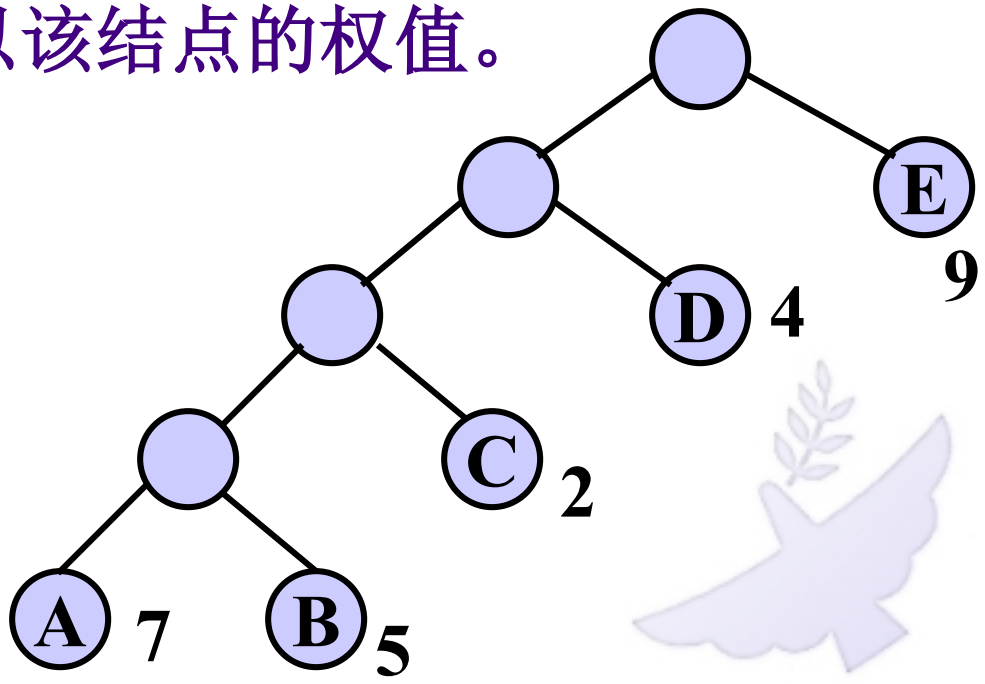
5.8 哈夫曼树与哈夫曼编码

- 5.8.1 最优树的定义
- 5.8.2 如何构造哈夫曼树
- 5.8.3前缀编码

5.8.1 最优树的定义

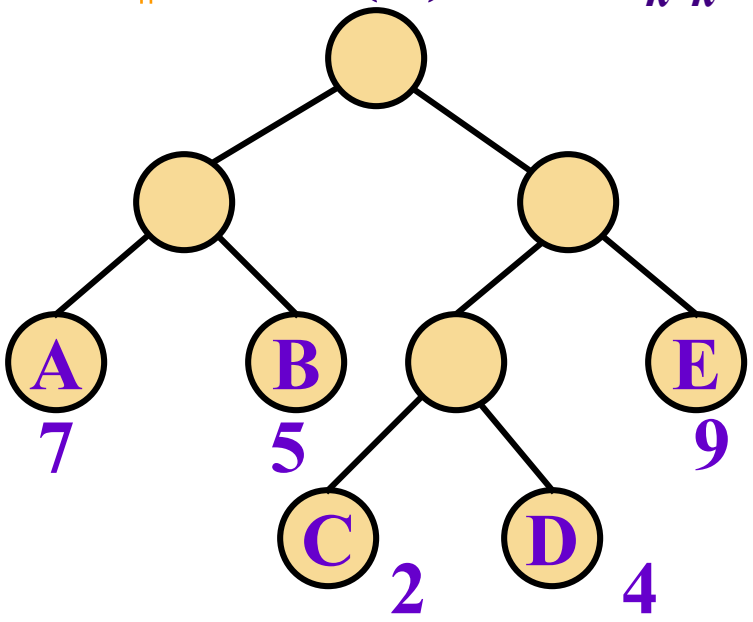
- 结点的路径长度定义为：
  - ¶ 从根结点到该结点的路径上分支的数目。
- 结点的带权路径长度定义为：
  - ¶结点的路径长度乘以该结点的权值。

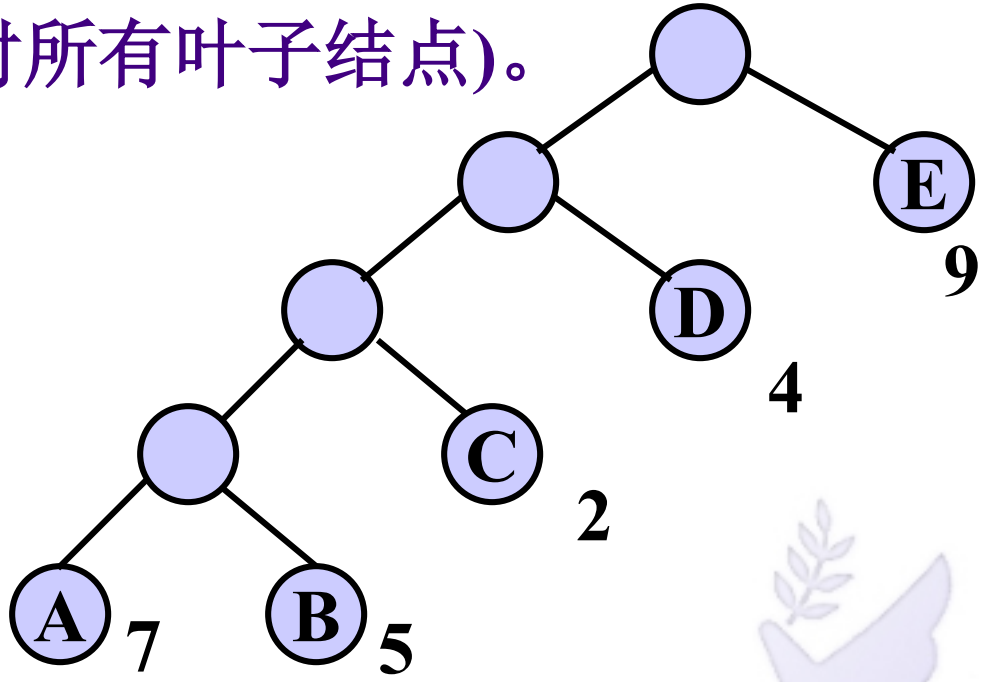




5.8.1 最优树的定义

- 树的带权路径长度定义为：
  - 树中所有叶子结点的带权路径长度之和
  - WPL(T) = $\Sigma\, w_k l_k$ (对所有叶子结点)。



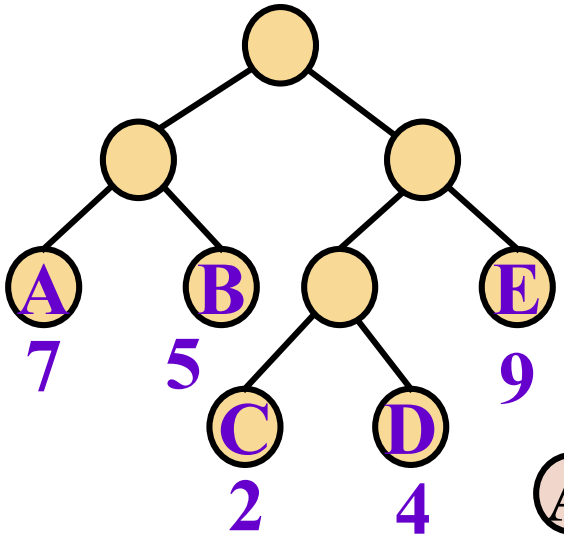


WPL(T)= 7×2+5×2+2×3+
+4×3+9×2 = 60

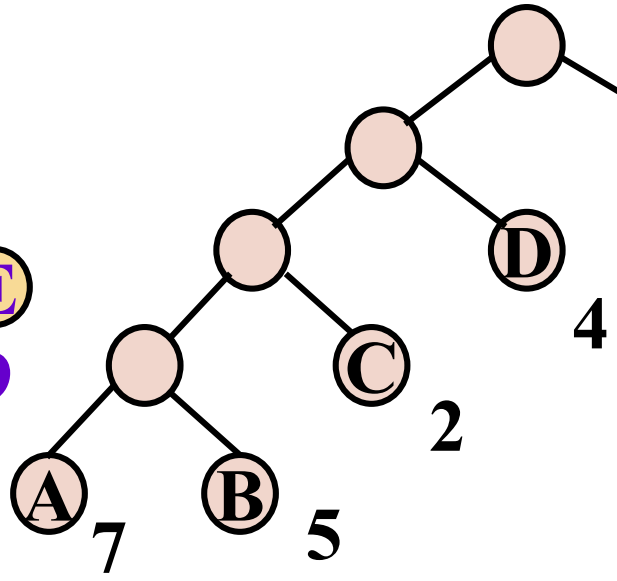
WPL(T)=7×4+5×4+2×3+
+4×2+9×1 = 69

5.8.1 最优树的定义

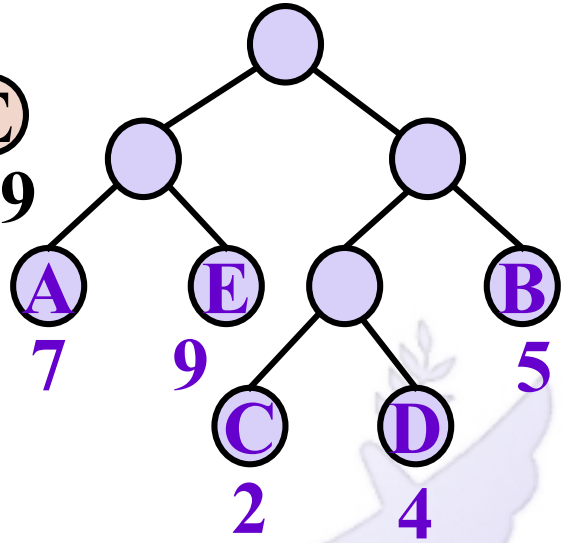
- 树的带权路径长度
  - 树中所有叶子结点的带权路径长度之和
  - WPL(T) = $\Sigma\, w_k l_k$ (对所有叶子结点)。



60



69



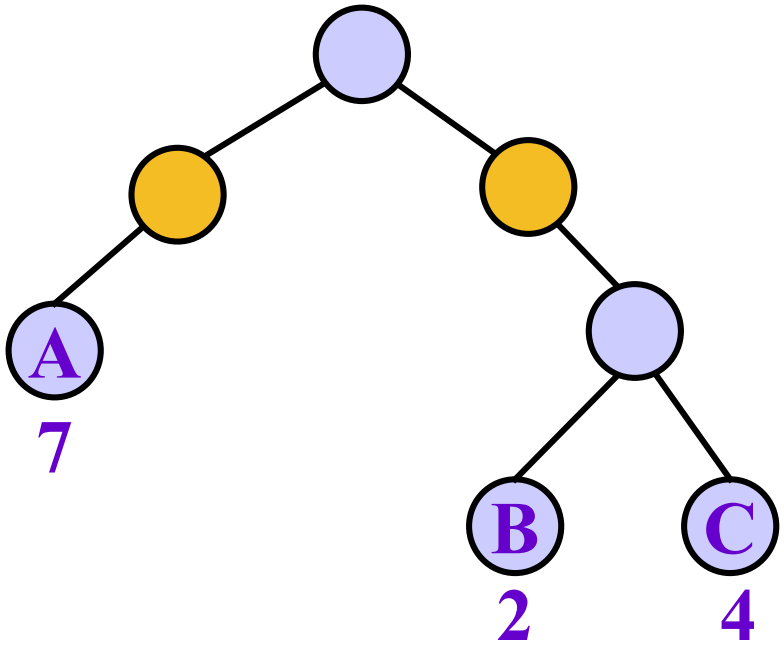
60

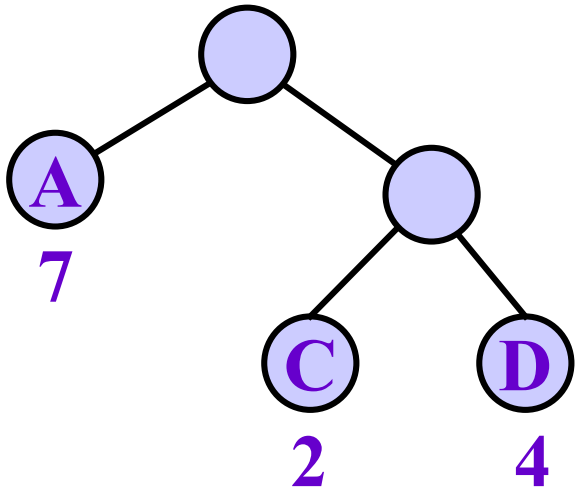
5.8.1 最优树的定义

- 最优树：
  - 在所有含 n 个叶子结点、并带相同权值的 m叉树中，必存在带权路径长度取最小值的树。
- 哈夫曼树：
  - 假设有n个权值$\{w_1, w_2, \ldots, w_n\}$，构造有n个叶子结点的二叉树，每个叶子结点有一个 w_i 作为它的权值。则带权路径长度最小的二叉树称为哈夫曼树。

哈夫曼树

- 引理：哈夫曼树一定是正则的二叉树。





5.8.2 如何构造哈夫曼树

◆ 哈夫曼算法——贪心算法

1. 根据给定的 n 个权值 $\{w_1, w_2, \ldots, w_n\}$，构造 n 棵二叉树的集合:

   ¶ $F = \{T_1, T_2, \ldots, T_n\}$，

   ¶ 其中每棵二叉树中均只含一个带权值为 w_i 的根结点，其左、右子树为空树；

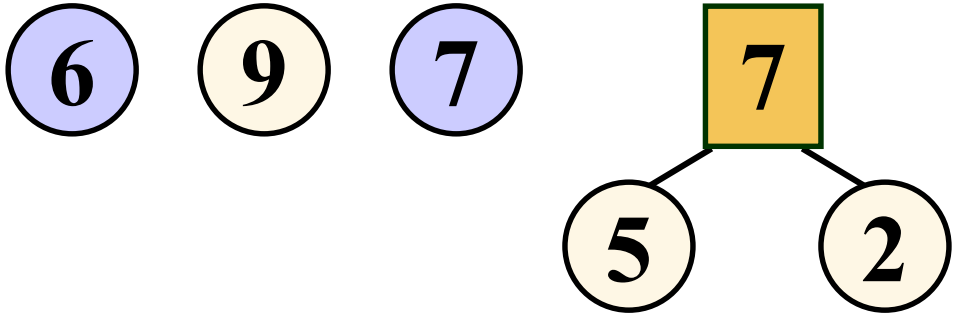
5.8.2 如何构造哈夫曼树

2. 在 F 中选取其根结点的权值为最小的两棵二叉树，
   - 分别作为左、右子树构造一棵新的二叉树
   - 并置这棵新的二叉树根结点的权值为其左、右子树根结点的权值之和；
3. 从F中删去这两棵树，同时加入刚生成的新树；
4. 重复 (2) 和 (3) 两步，直至 F 中只含一棵树为止。

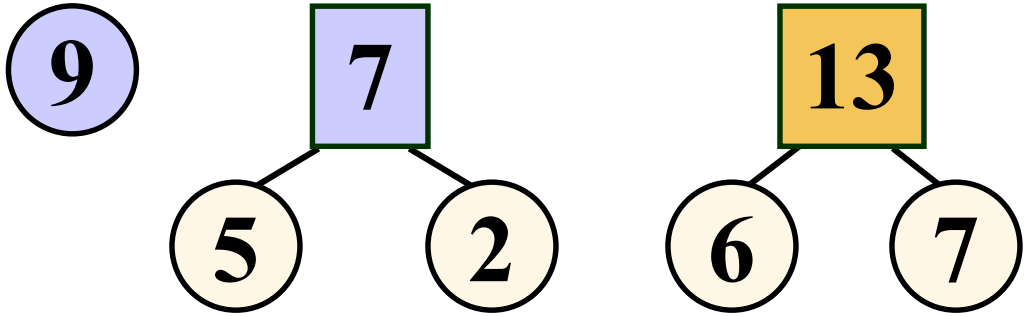
例如：已知权值 W={ 5, 6, 2, 9, 7 }

1）

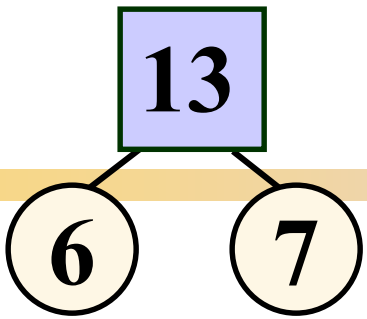
2）

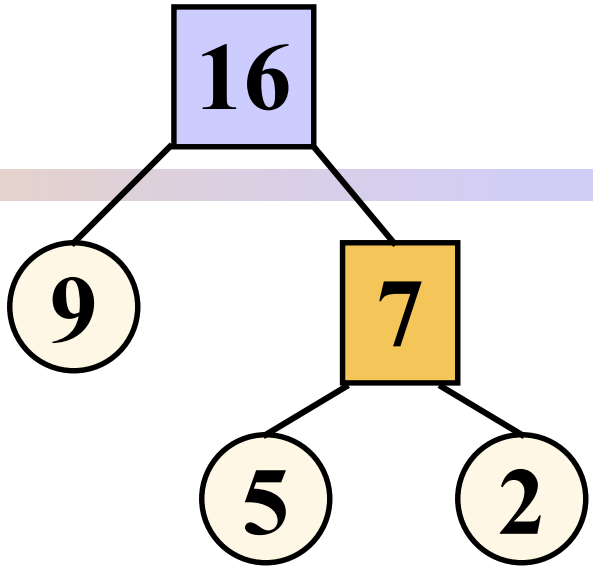


3）

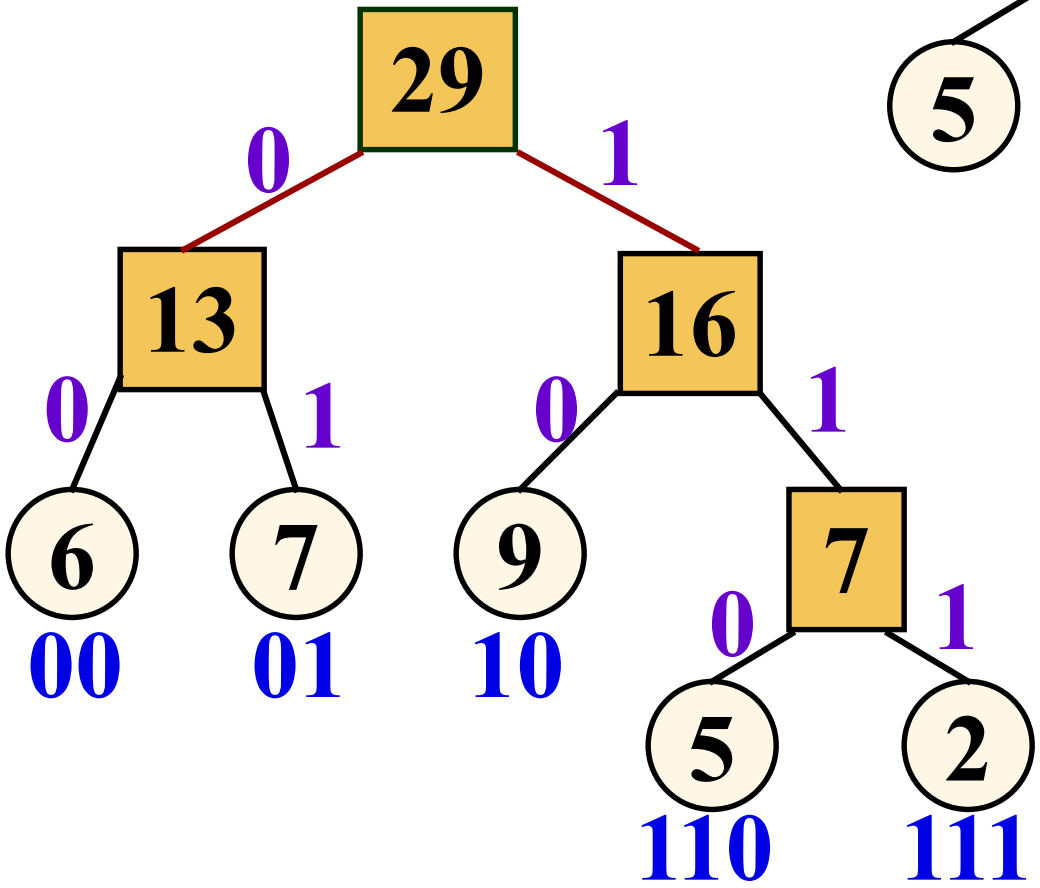


3）





4）



存储哈夫曼树

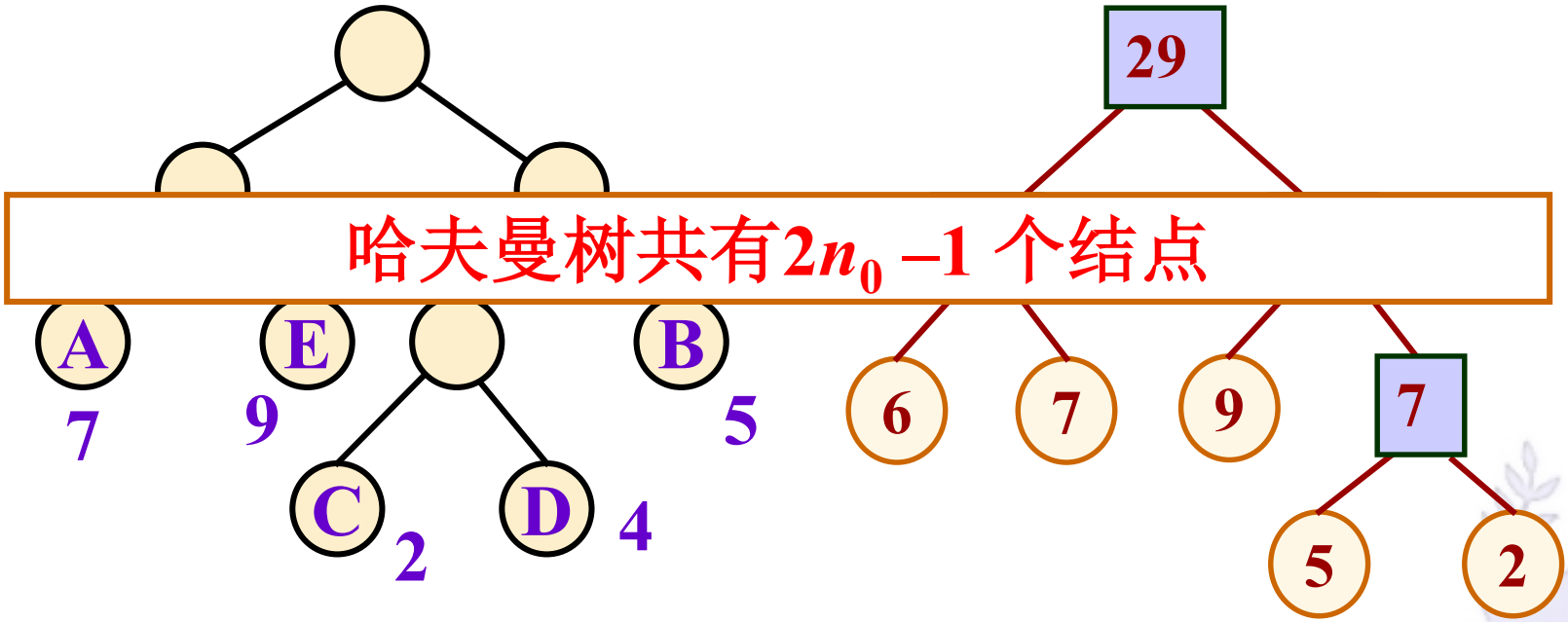
- 静态三叉链表

weight	parent	lchild	rchild

```
//静态三叉链表结点类型定义
typedef struct {
        char data;
        int weight;
        int parent, lchild, rchild;
}HTNode, *HuffmanTree;
```

存储哈夫曼树

- 哈夫曼树共有多少结点？（假设有n_0个叶结点）
- 哈夫曼树是正则的二叉树：没有度为1的结点



性质 3：对任何一棵二叉树，必存在关系式：

$n_0 = n_2+1$。 → $n_2 = n_0 -1$

存储哈夫曼树

W

算法原始数据

5	29	7	8	14	23	3	11
0	1	2	3	4	5	6	7

哈夫曼树存储空间

2n-1

	w	p	lch	rch
0	5	8	-1	-1
1	29	13	-1	-1
2	7	9	-1	-1
3	8	9	-1	-1
4	14	11	-1	-1
5	23	12	-1	-1
6	3	9	-1	-1
7	11	10	-1	-1
8	8	10	0	6
9	15	11	2	3
10	19	12	7	8
11	29	13	4	9
12	42	14	5	10
13	58	14	1	11
14	100	-1	12	13

5.8.2 构造哈夫曼树

HT →

	w	p	lch	rch
0	5	8	-1	-1
1	29	13	-1	-1
2	7	9	-1	-1
3	8	9	-1	-1
4	14	11	-1	-1
5	23	12	-1	-1
6	3	9	-1	-1
7	11	10	-1	-1
8	8	10	0	6
9	15	11	2	3
10	19	12	7	8
11	29	13	4	9
12	42	14	5	10
13	58	14	1	11
14	100	-1	12	13

W ↓

5	29	7	8	14	23	3	11
0	1	2	3	4	5	6	7

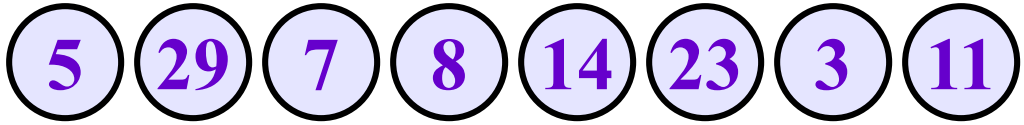
步骤0：

为哈夫曼树分配存储空间

2n-1

```
HT=(HuffmanTree)malloc(2*n-1)*
        sizeof(HTNode);
```

5.8.2 构造哈夫曼树



8棵只有一个结点的二叉树

步骤1：
构造n棵只有一个根结点的二叉树

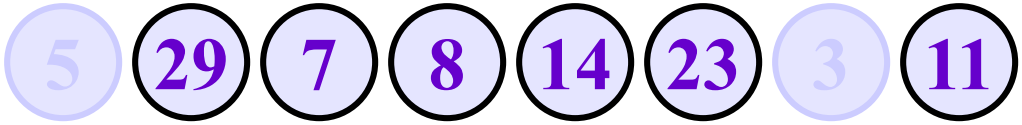
```
for  (p=HT,q=W, i=0;  i<n; ++i, ++p, ++q)
        *p={ *q, -1,-1,-1};
```

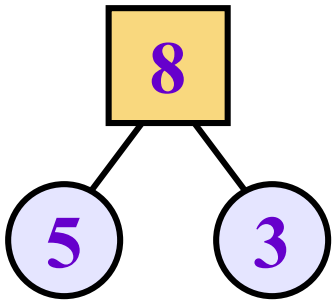
HT →

	w	p	lch	rch
0	5	8	-1	-1
1	29	13	-1	-1
2	7	9	-1	-1
3	8	9	-1	-1
4	14	11	-1	-1
5	23	12	-1	-1
6	3	9	-1	-1
7	11	10	-1	-1
8	8	10	0	6
9	15	11	2	3
10	19	12	7	8
11	29	13	4	9
12	42	14	5	10
13	58	14	1	11
14	100	-1	12	13

i=8 →

5.8.2 构造哈夫曼树





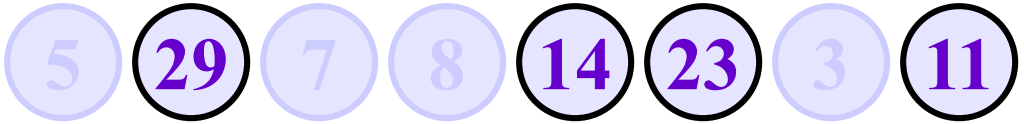
步骤2：

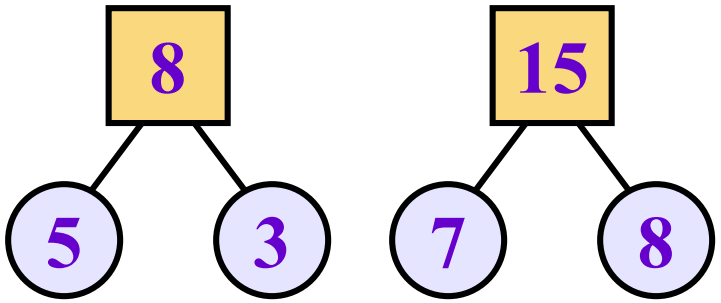
HT →

	w	p	lch	rch
0	5	8	-1	-1
1	29	13	-1	-1
2	7	9	-1	-1
3	8	9	-1	-1
4	14	11	-1	-1
5	23	12	-1	-1
6	3	8	-1	-1
7	11	10	-1	-1
8	8	-1	0	6
9				
10				
11				
12				
13				
14				

i=8 →

5.8.2 构造哈夫曼树





步骤2：

HT →

	w	p	lch	rch
0	5	8	-1	-1
1	29	13	-1	-1
2	7	9	-1	-1
3	8	9	-1	-1
4	14	11	-1	-1
5	23	12	-1	-1
6	3	8	-1	-1
7	11	10	-1	-1
8	8	-1	0	6
9	15	-1	2	3
10				
11				
12				
13				
14				

i=9 →

5.8.2 构造哈夫曼树

HT

	w	p	lch	rch
0	5	8	-1	-1
1	29	13	-1	-1
2	7	10	-1	-1
3	8	9	-1	-1
4	14	11	-1	-1
5	23	12	-1	-1
6	3	8	-1	-1
7	11	10	-1	-1
8	8	-1	0	6
9	15	-1	2	3
10				
11				
12				
13				
14				

5 29 7 8

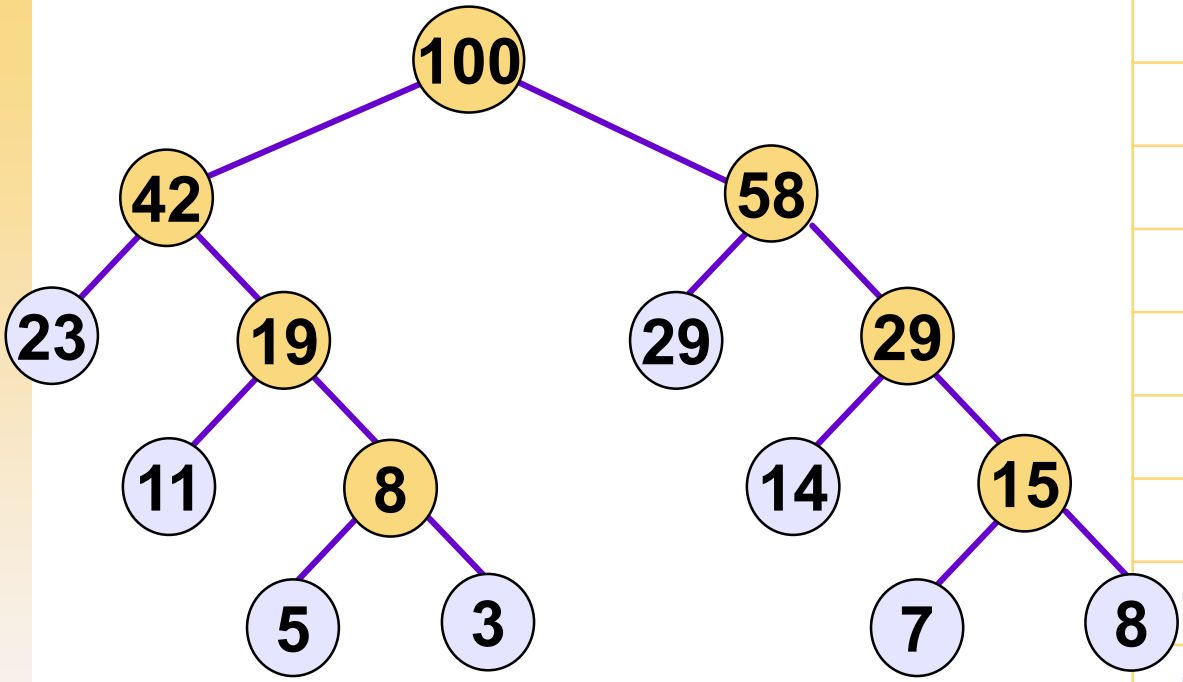
```
for (i=n; i<2*n-1; ++i; ++p)
```

步骤2：循环

在 F 中选取根结点权值最小的两棵二叉树s1和s2

- s1和s2分别作为左、右子树构造一棵新二叉树*I*
- 并置二叉树*i*根结点的权值为其左、右子树根结点的权值之和；

5.8.2 构造哈夫曼树



HT

	w	p	lch	rch
0	5	8	-1	-1
1	29	13	-1	-1
2	7	9	-1	-1
3	8	9	-1	-1
4	14	11	-1	-1
5	23	12	-1	-1
6	3	9	-1	-1
7	11	10	-1	-1
8	8	10	0	6
9	15	11	2	3
10	19	12	7	8
11	29	13	4	9
12	42	14	5	10
13	58	14	1	11
14	100	-1	12	13

5.8.2 构造哈夫曼树

```
Status HuffmanTree(HuffmanTree &HT,  int * w, int n)
{ //w 存放n 个权值（均>0），构造哈夫曼树HT
      if (n<=1) return ERROR;

  //为哈夫曼树分配存储空间
  HT=(HuffmanTree)malloc((2*n-1)*sizeof(HTNode);

  //1) 构造n棵只有一个根结点的二叉树
  for  (p=HT,q=w, i=0; i<n; ++i, ++p, ++q)
        *p={ *q,-1,-1,-1};
```

5.8.2 构造哈

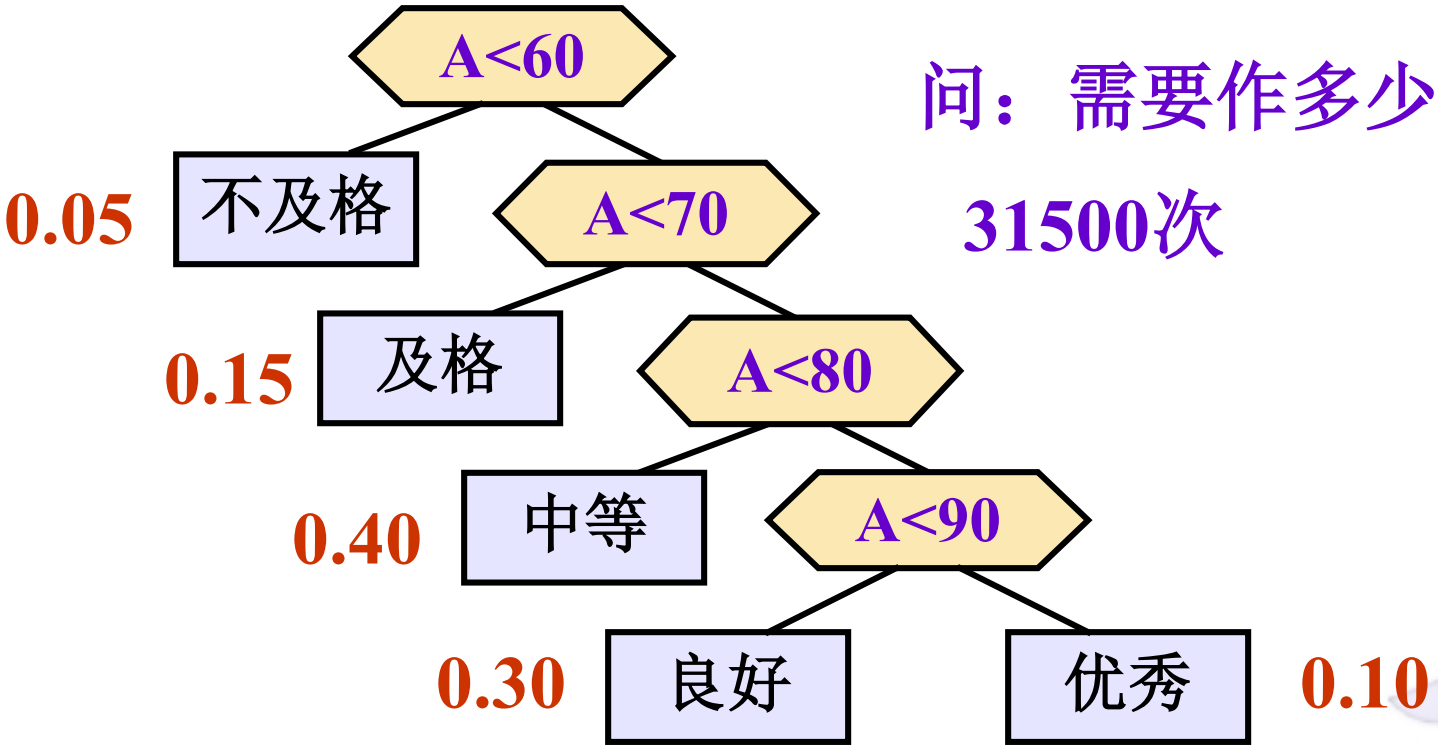
Select：在HT[1..i-1] 选择parent为0且weight最小的两个结点，其序号分别存入s1和s2

```
//循环构建哈夫曼树
  for (i=n; i<2*n-1; ++i; ++p){
     Select(HT, i-1, s1, s2); //选择权值最小的两个二叉树
     HT[s1]. parent =i;  HT[s2].parent=i;  //设父指针为i
     HT[i].lchild=s1;  HT[i].rchild=s2; //设置i的子指针
     HT[i].weight=HT[s1].weight+HT[s2].weight;//权重
  }//for
   return OK;
} // HuffmanTree
```

若Select采用扫描法，复杂度为$O(n^2)$.

例：设有10000个百分制分数要转换，设学生成绩在5个等级以上的分布如下：

分数	0-59	60-69	70-79	80-89	90-100
比例	0.05	0.15	0.40	0.30	0.10



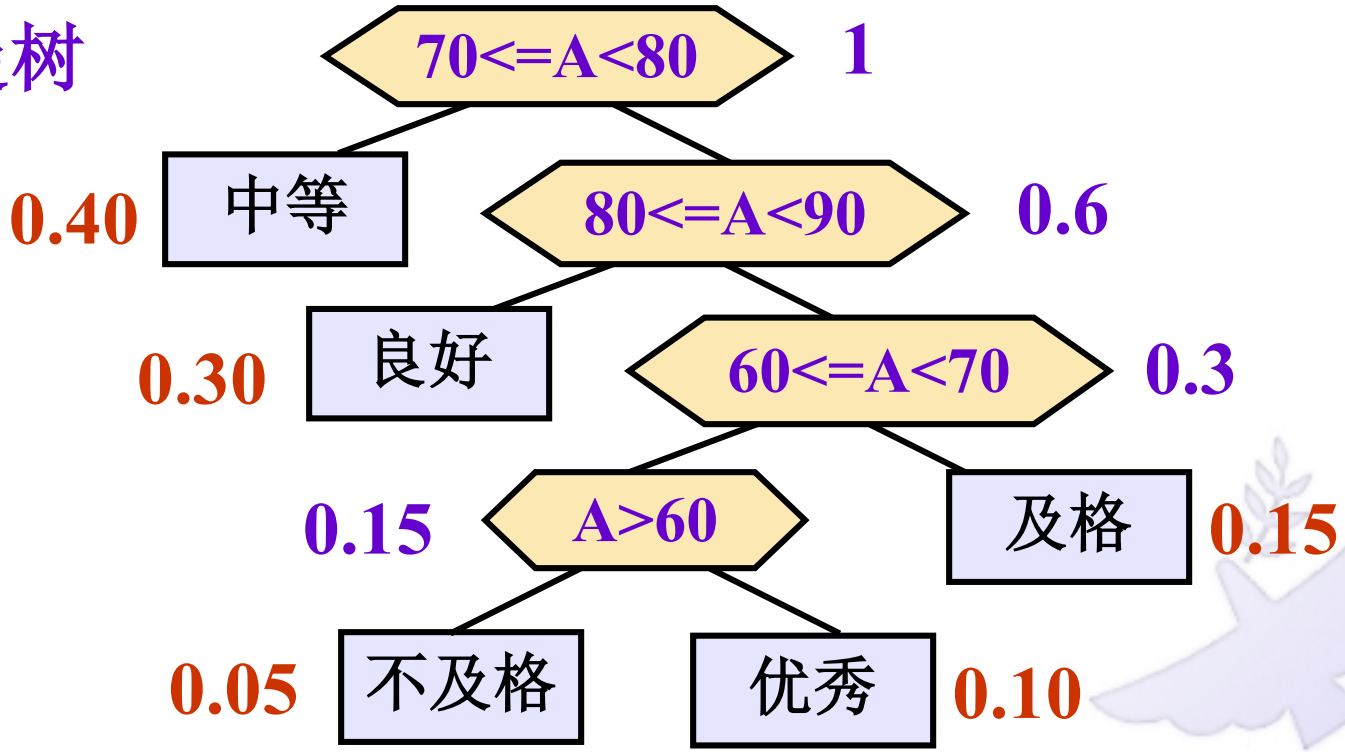
问：需要作多少次比较？

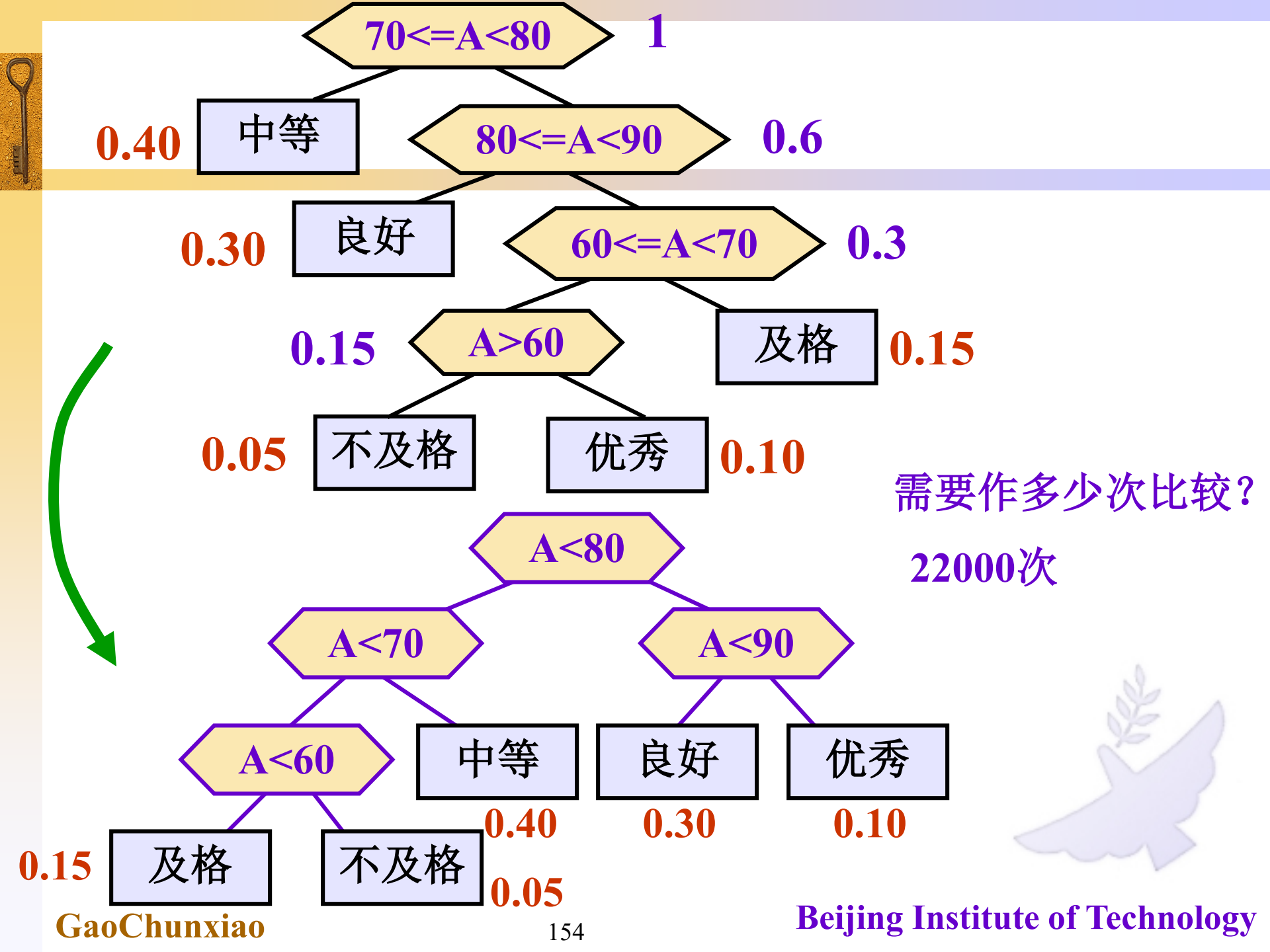
31500次

例：设有10000个百分制分数要转换，设学生成绩在5个等级以上的分布如下：

分数	0-59	60-69	70-79	80-89	90-100
比例	0.05	0.15	0.40	0.30	0.10

构建哈夫曼树




70<=A<80
1
0.40
中等
80<=A<90
0.6
0.30
良好
60<=A<70
0.3
0.15
A>60
及格
0.15
0.05
不及格
优秀
0.10
需要作多少次比较？
22000次
A<80
A<70
A<90
A<60
中等
良好
优秀
0.40
0.30
0.10
0.15
及格
不及格
0.05

5.8.3 哈夫曼编码

- 前缀编码：任何一个字符的编码都不是同一字符集中另一个字符的编码的前缀

例如：

{1, 00, 011, 0101, 01001, 01000}

{1, 00, 011, 0100, 01001, 01000} ✖

- 利用哈夫曼树
  - ¶ 可以构造一种不等长的二进制编码；
  - ¶ 并且构造所得的哈夫曼编码是一种最优前缀编码，
  - ¶ 即使所传电文的总长度最短。

例：某通讯系统只使用8种字符，设计其编码。已知字符如下：

字符	a	b	c	d	e	f	g	k

a: 000
b: 001
c: 010
d: 011
e: 100
f: 101
g: 110
k: 111

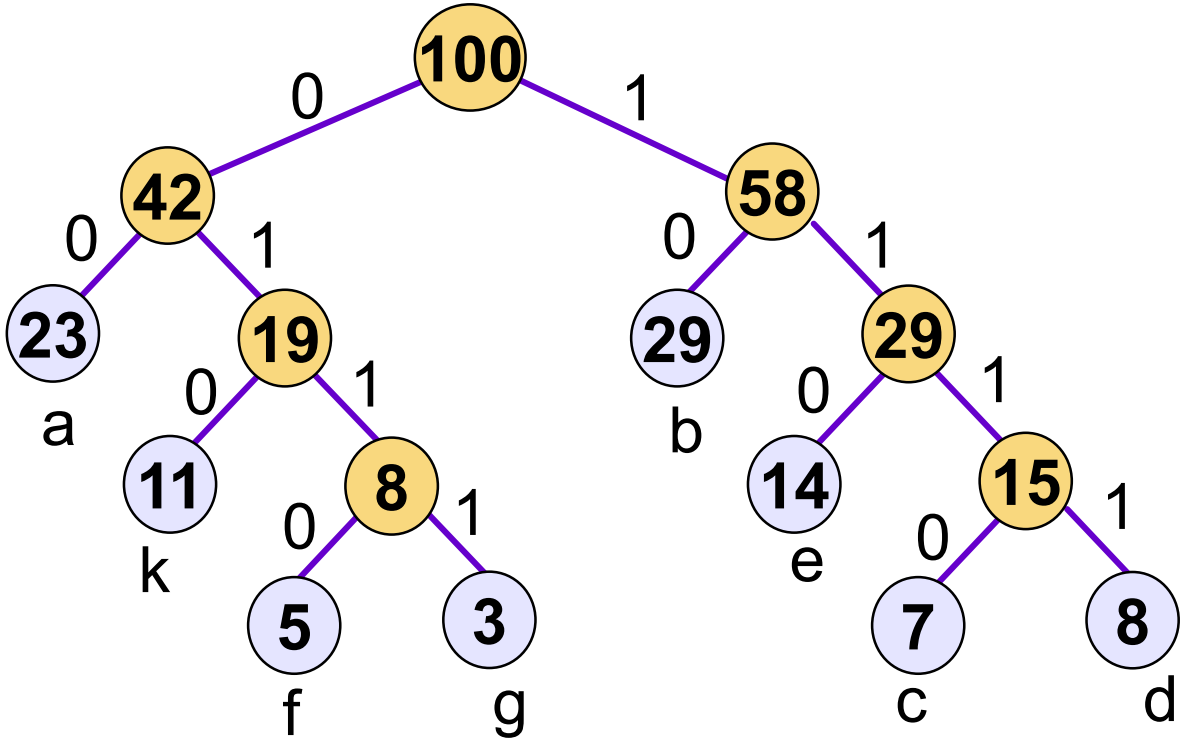
bad bag back

30bit

001000011001000110001000010111

例：某通讯系统只使用8种字符，使用哈夫曼树设计不等长编码。已知字符和对应使用频率如下：

字符	a	b	c	d	e	f	g	k
频率%	23	29	7	8	14	5	3	11



a: 00
b: 10
c: 1110
d: 1111
e: 110
f: 0110
g: 0111
k: 010

bad bag back编码是什么？需要多少bit？

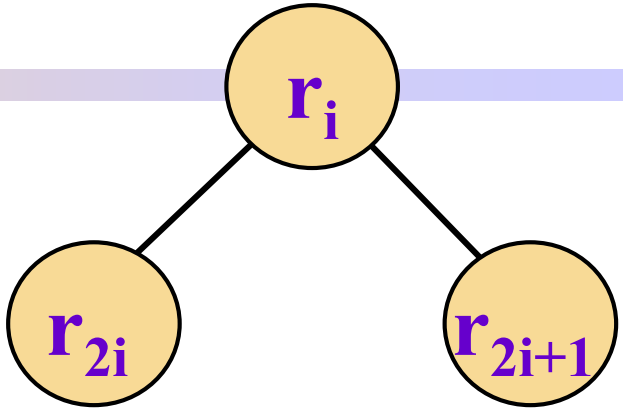
10001111100001111001110010 ——23bit

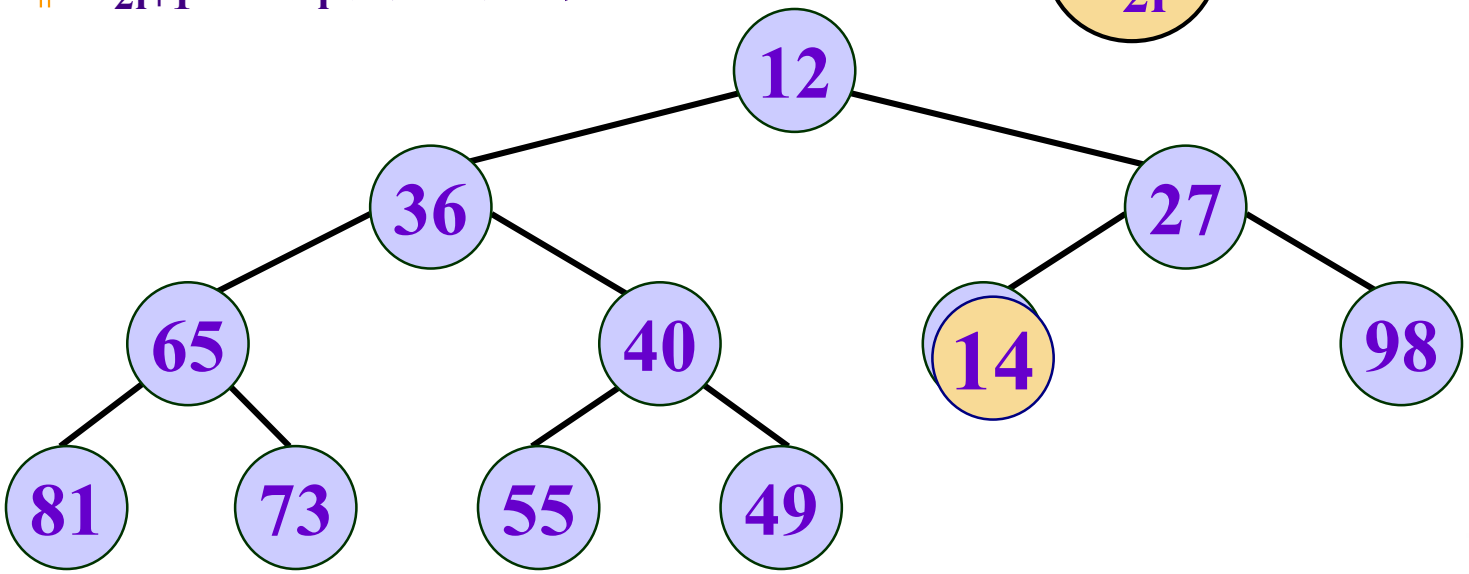
堆的简介

- 堆的定义
- 堆上的操作
  - 堆的调整过程
  - 堆的建立
  - 堆的插入和删除

堆的定义

- 完全二叉树（根结点编号为1）：
  - ¶ r_{2i} 是 r_i 的左孩子；
  - ¶ r_{2i+1} 是 r_i 的右孩子。



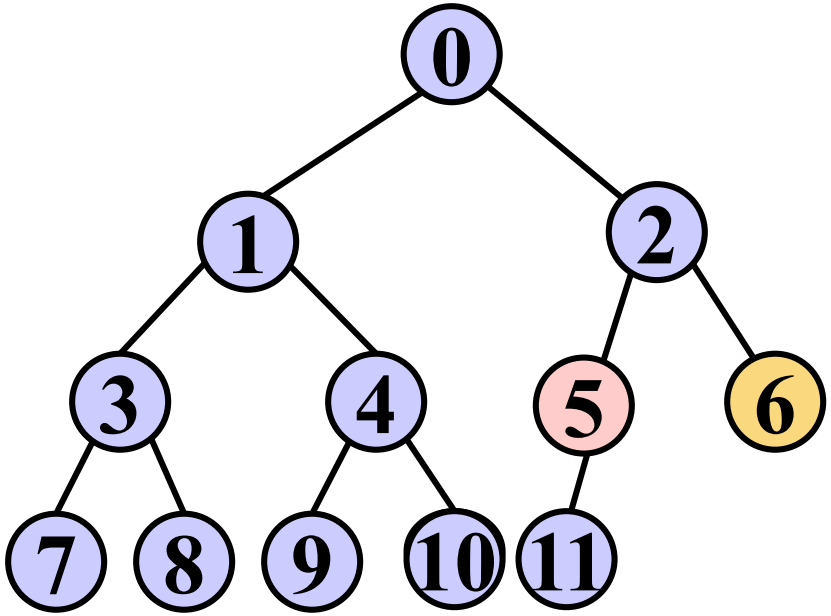


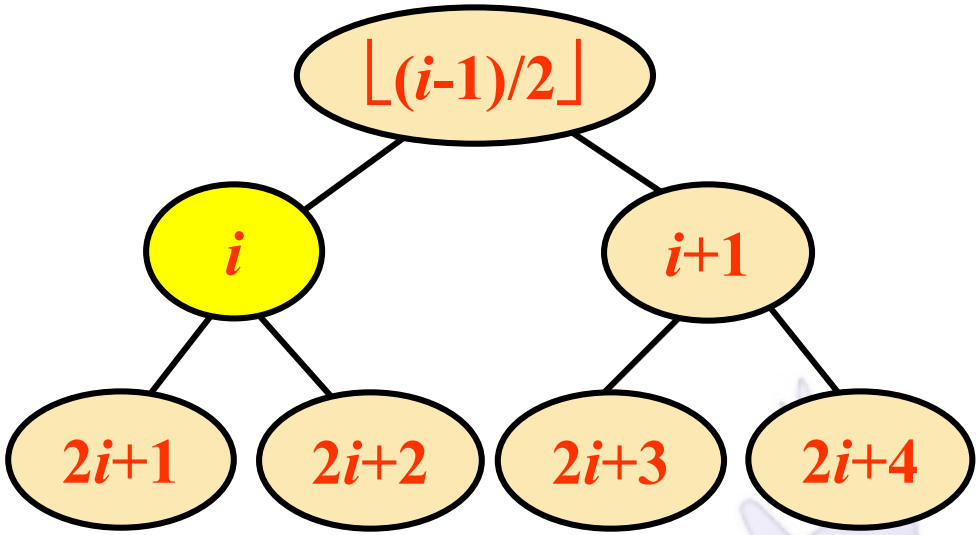
堆：R[11]{12, 36, 27, 65, 40, 34, 98, 81, 73, 55, 49}

R[11]{12, 36, 27, 65, 40, 14, 98, 81, 73, 55, 49}不是堆

5.2.2 二叉树的性质－5

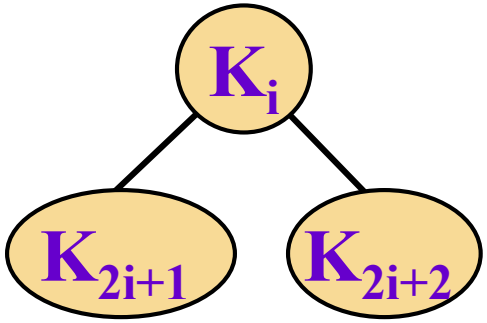
- 如果根结点从0开始编号，则结点i与其父结点和子结点的对应关系是：





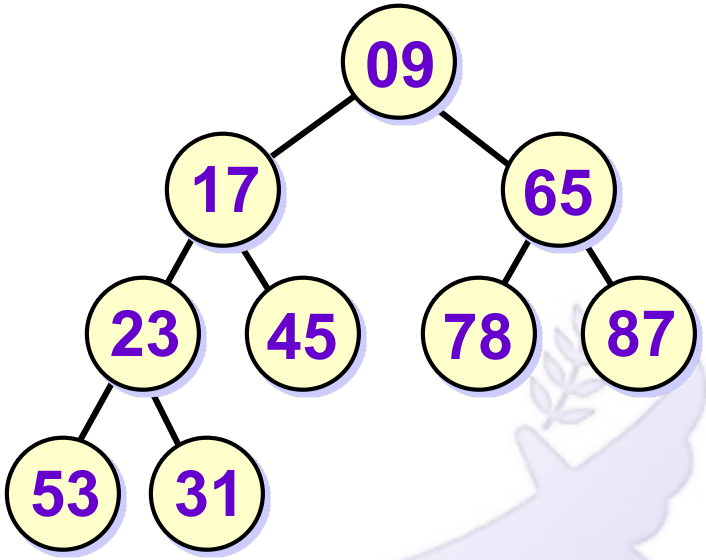
堆的定义

- 小根堆
  - ¶ $K_i \le K_{2i+1}$
  - ¶ $K_i \le K_{2i+2}$

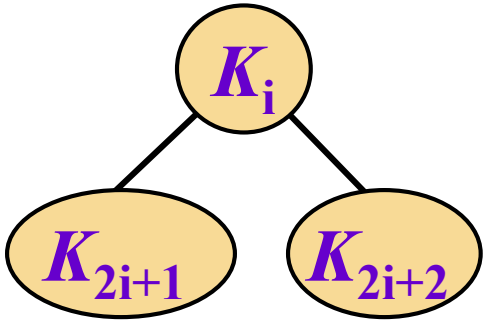


完全二叉树的顺序表示

0	1	2	3	4	5	6	7	8
09	17	65	23	45	78	87	53	31

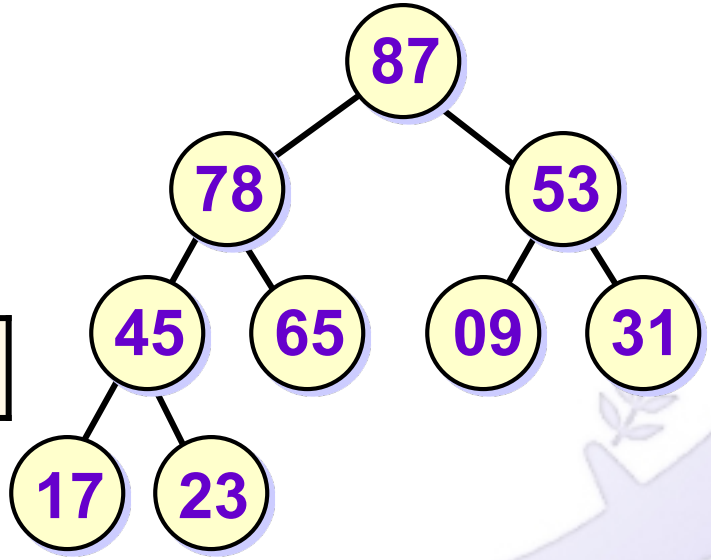


- 大根堆
  - ¶ $K_i \geq K_{2i+1}$
  - ¶ $K_i \geq K_{2i+2}$



完全二叉树的顺序表示

0	1	2	3	4	5	6	7	8
09	17	65	23	45	78	87	53	31

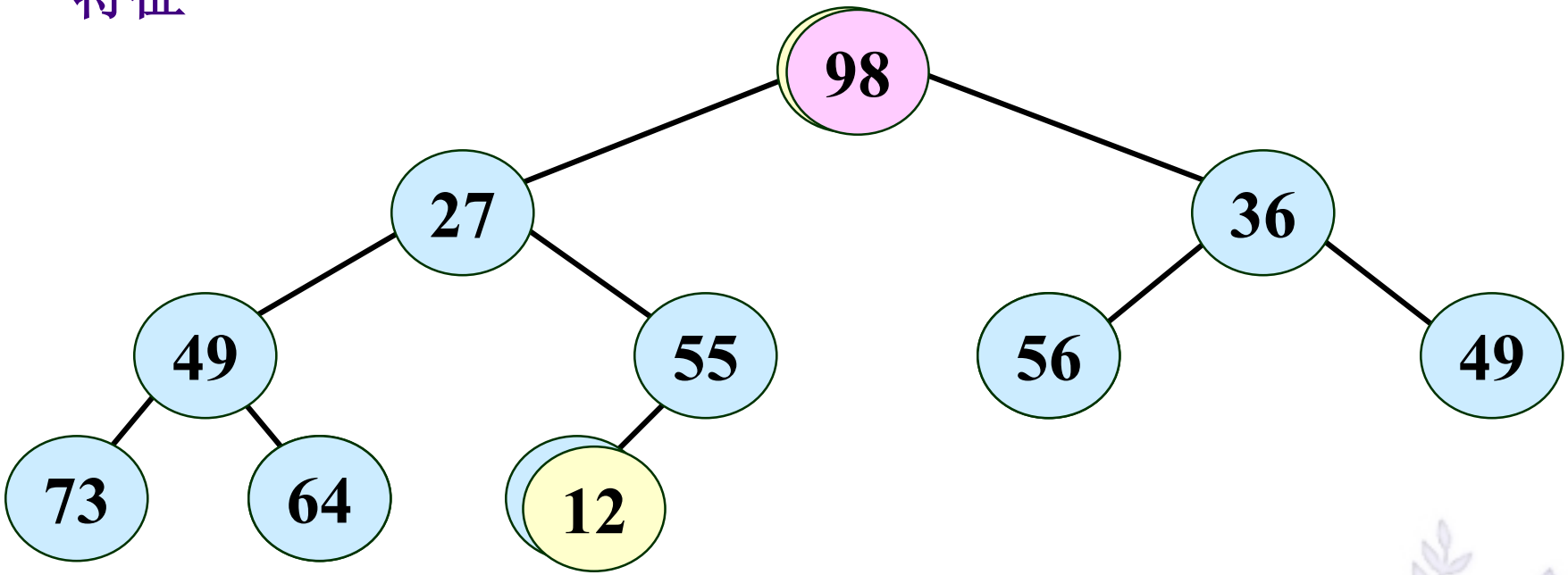


堆的定义

```
#define heapSize 100          //堆的最大元素个数
typedef int HElemType;        //堆中元素的数据类
typedef struct {              //堆结构的定义
    HElemType elem[heapSize];      //堆存储数组
    int curSize;                   //当前元素个数
} minHeap;
```

堆的操作：调整堆

- 即假设H.elem[i..m]中记录的关键字除 elem[i] 之外均满足堆的特征

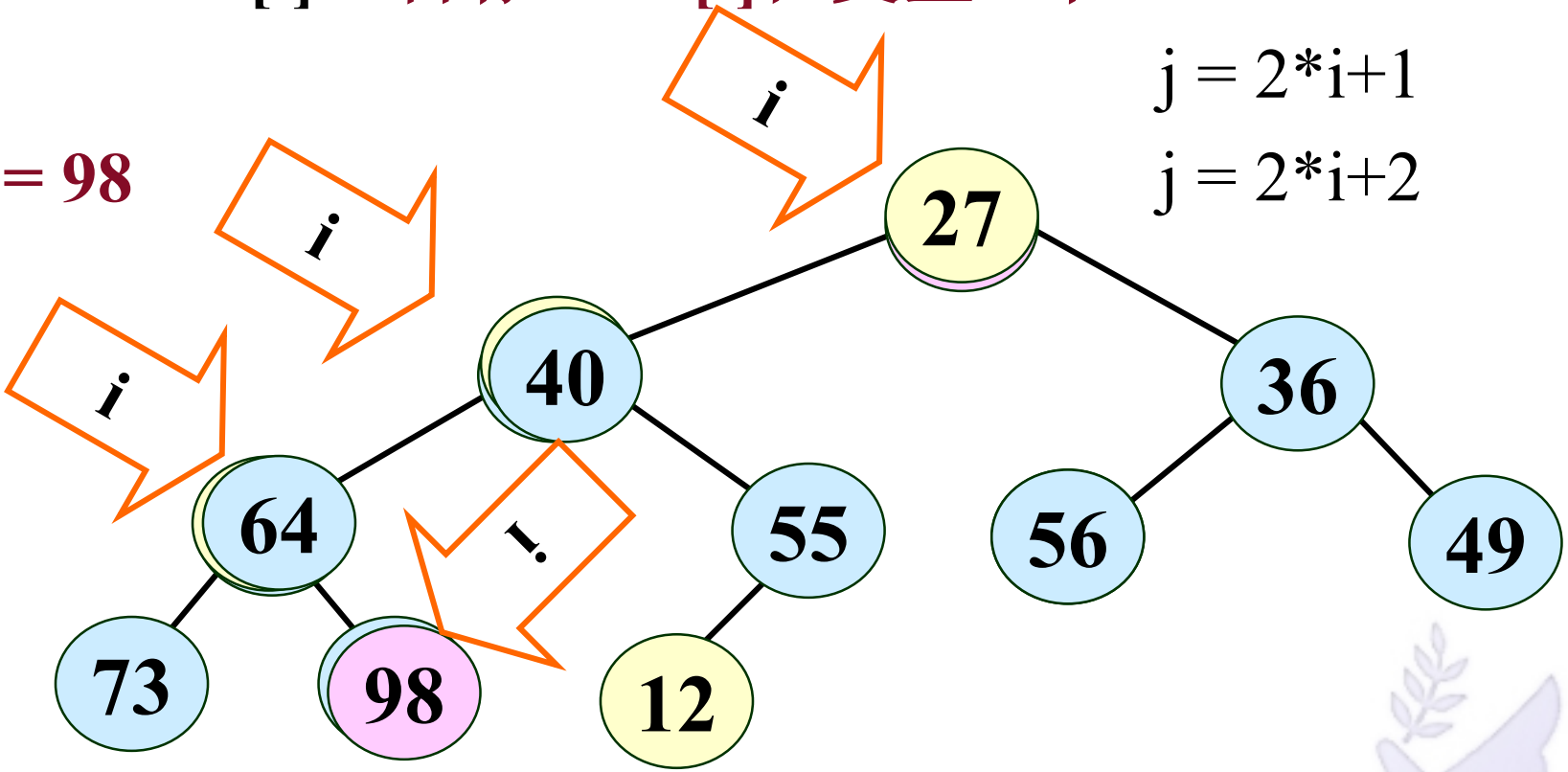


堆：R[10]{12,36,27,49,55,56,49,73,64,98}

堆：R[10]{98,36,27,49,55,56,49,73,64}{12}

调整堆

temp = **H.elem[i]** // 暂存 elem[i]在变量rc中



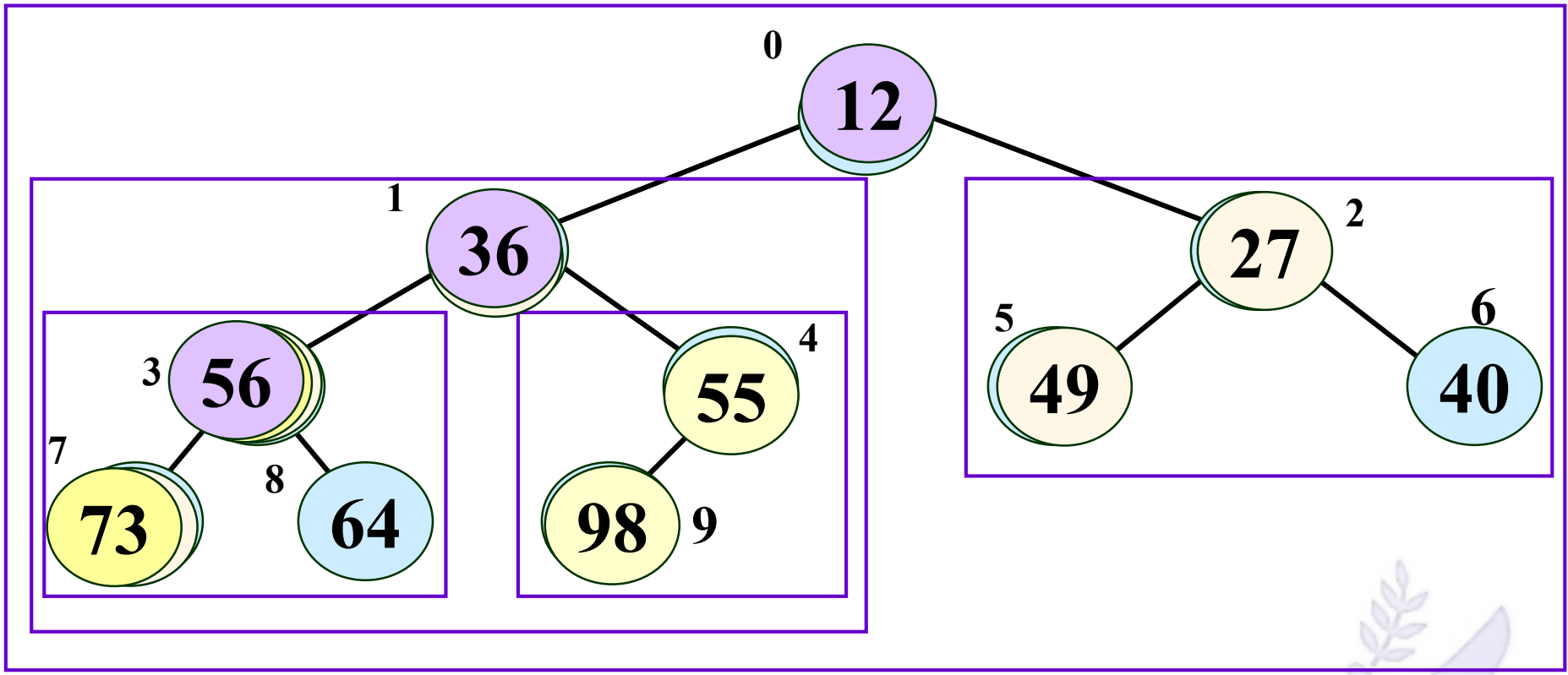
H.elem[i] = temp

调整需要多次从上向下的交换，称为筛选

小根堆的向下筛选算法

```
void siftDown ( minHeap& H, int i, int m ) {
//从结点i开始到m为止, 自上向下比较, 将一个集合局部
//调整为小根堆
    HElemType temp = H.elem[i];
    for ( int j = 2*i+1; j <= m; j = 2*j+1 ) {
        if ( j < m && H.elem[j] > H.elem[j+1] ) j++;//左右比较
        if ( temp <= H.elem[j] ) break;     //小则不做调整
        else { H.elem[i] = H.elem[j];  i = j; }     //小者上移
    }
    H.elem[i] = temp;          //回放temp中暂存的元素
}
```

建堆 小根堆

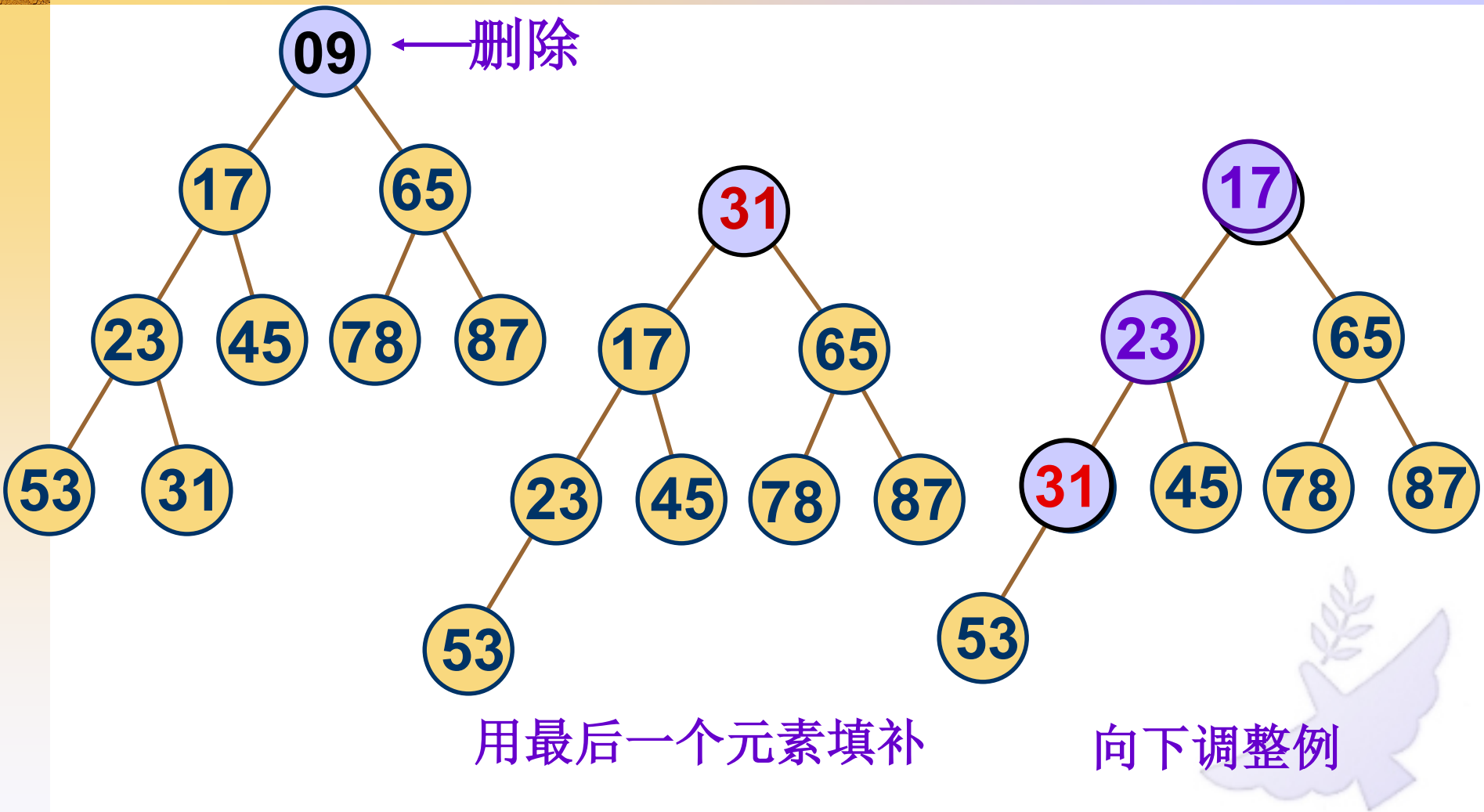


- 自下向上逐步调整为小根堆的过程，多次筛选

小根堆的建立

```
void creatMinHeap ( minHeap& H,
        HElemType arr[ ], int n ) {
//将一个数组从局部到整体，自下向上调整为小根堆
   for ( int i = 0; i < n; i++ ) H.elem[i] = arr[i];    //复制
   H.curSize = n;
   for ( i = (H.curSize-2)/2; i >= 0; i-- )
                  //自底向上逐步扩大小根堆
      siftDown ( H, i, H.curSize-1 );
                  //局部自上向下筛选
};
```

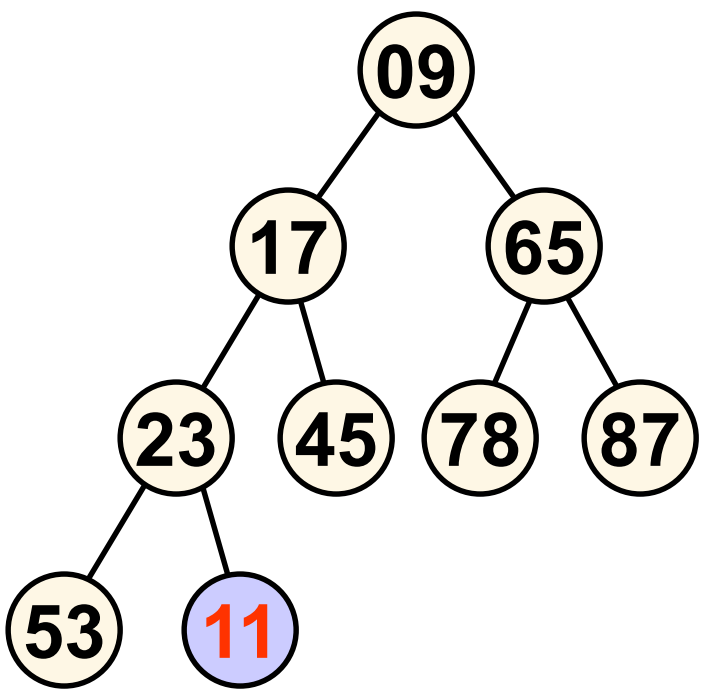
小根堆的删除和向下调整例



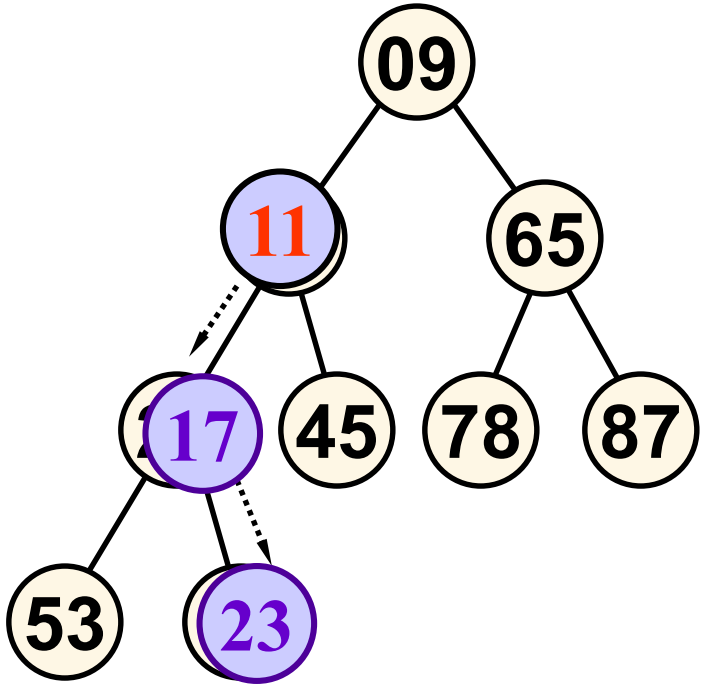
小根堆的删除算法

```
bool Remove ( minHeap& H, HElemType& x ) {
//从小根堆中删除堆顶元素并通过引用参数 x 返回
    if ( H.curSize == 0 ) return false;     //堆空, 返回
    x = H.elem[0];                  //返回最小元素
    H.elem[0] = H.elem[H.curSize-1];
                                    //最后元素填补到根结点
    H.curSize--;
    siftDown ( H, 0, H.curSize-1 );    //从新调整为堆
    return true;
}
```

小根堆的插入和向上调整例



在堆中插入新元素 11



沿通向根的路径向上调整

小根堆的插入算法

```
bool Insert ( minHeap& H, HElemType x ) {
//将x插入到小根堆中并从新调整形成新的小根堆
    if ( H.curSize == heapSize ) return false;
          //堆满，返回插入不成功信息
    H.elem[H.curSize] = x;          //插入到最后
    siftUp ( H, H.curSize );        //从下向上调整
    H.curSize++;                    //堆计数加1
    return true;
}
```

小根堆的向上筛选算法

```
void siftUp ( minHeap& H, int start ) {
//从结点start开始到结点0为止, 自下向上比较, 将集合
//重新调整为堆。
    HElemType temp = H.elem[start];
    int j = start, i = (j-1)/2;
    while ( j > 0 ) {   //沿双亲路径向上直达根
        if ( H.elem[i] <= temp ) break;     //双亲值小
        else { H.elem[j] = H.elem[i];  j = i;  i = (i-1)/2; }
    }                          //双亲的值下降，j与i的位置上升
    H.elem[j] = temp;          //start放置在正确位置
}//
```

本章要点

1. 熟练掌握二叉树的结构特性，了解相应的证明方法。
2. 熟悉二叉树的各种存储结构的特点及适用范围。
3. 遍历二叉树是二叉树各种操作的基础。实现二叉树遍历的具体算法与所采用的存储结构有关。掌握各种遍历策略的递归算法，灵活运用遍历算法实现二叉树的其它操作。层次遍历是按另一种搜索策略进行的遍历。

4. 理解二叉树线索化的实质是建立结点与其在相应序列中的前驱或后继之间的直接联系，熟练掌握二叉树的线索化过程以及在中序线索化树上找给定结点的前驱和后继的方法。二叉树的线索化过程是基于对二叉树进行遍历，而线索二叉树上的线索又为相应的遍历提供了方便。

本章要点

5. 熟悉树的各种存储结构及其特点，掌握树和森林与二叉树的转换方法。
6. 学会编写实现树的各种操作的算法。
7. 了解最优树的特性，掌握建立最优树和哈夫曼编码的方法。

END OF CHAPTER V