第六章

树和二叉树

※ 教学内容：

树的基本概念；二叉树的性质和存储结构；遍历二叉树和线索二叉树;树的存储结构和遍历；哈夫曼树及其应用；

※ 教学重点：

二叉树的结构特点；二叉树各种存储结构的特点及适用范围；按各种次序遍历二叉树的递归和非递归算法；二叉树的线索化，在中序线索树上找给定结点的前驱和后继的方法；树的各种存储结构及其特点；编写树的各种运算的算法；建立最优二叉树和哈夫曼编码的方法。

※ 教学难点：

按各种次序遍历二叉树的非递归算法。

第六章 树和二叉树

6.1 树的类型定义

6.2 二叉树的类型定义

6.3 二叉树的存储结构

6.4 二叉树的遍历

6.5 线索二叉树

6.6 树和森林

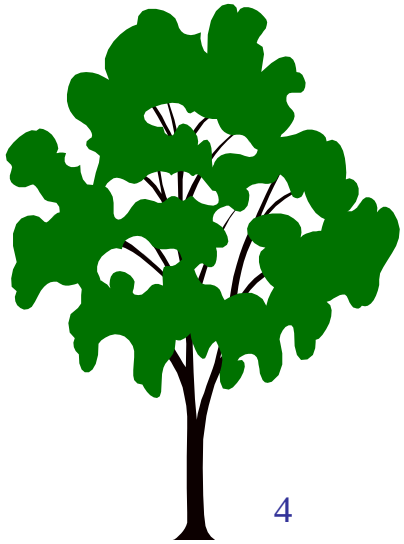
6.7 树和森林的遍历

6.8 哈夫曼树与哈夫曼编码

6.1 树的类型定义

一、树的定义

树是n（n≥0）个结点的有限集。当n=0时称为空树；在任意一棵非空树中，有且仅有一个称为根的结点，其余的结点可分为m(m≥0)个互不相交的有限集T1,T2,…,Tm,其中每一个集合又称为一棵树，并且称为根的子树。同理，每一棵子树又可以分为若干个互不相交的有限集……



二、抽象数据类型树的定义

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$T_1, T_2, \ldots, T_m$，其中每一棵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：

- 查 找 类
- 插 入 类
- 删 除 类

}ADT Tree

查找类:

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

TreeEmpty(T) //判定树是否为空树

TreeDepth(T) //求树的深度

TraverseTree(T, Visit()) //遍历

插入类：

InitTree(&T) // 初始化置空树

CreateTree(&T, definition) // 按定义构造树

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

删除类:

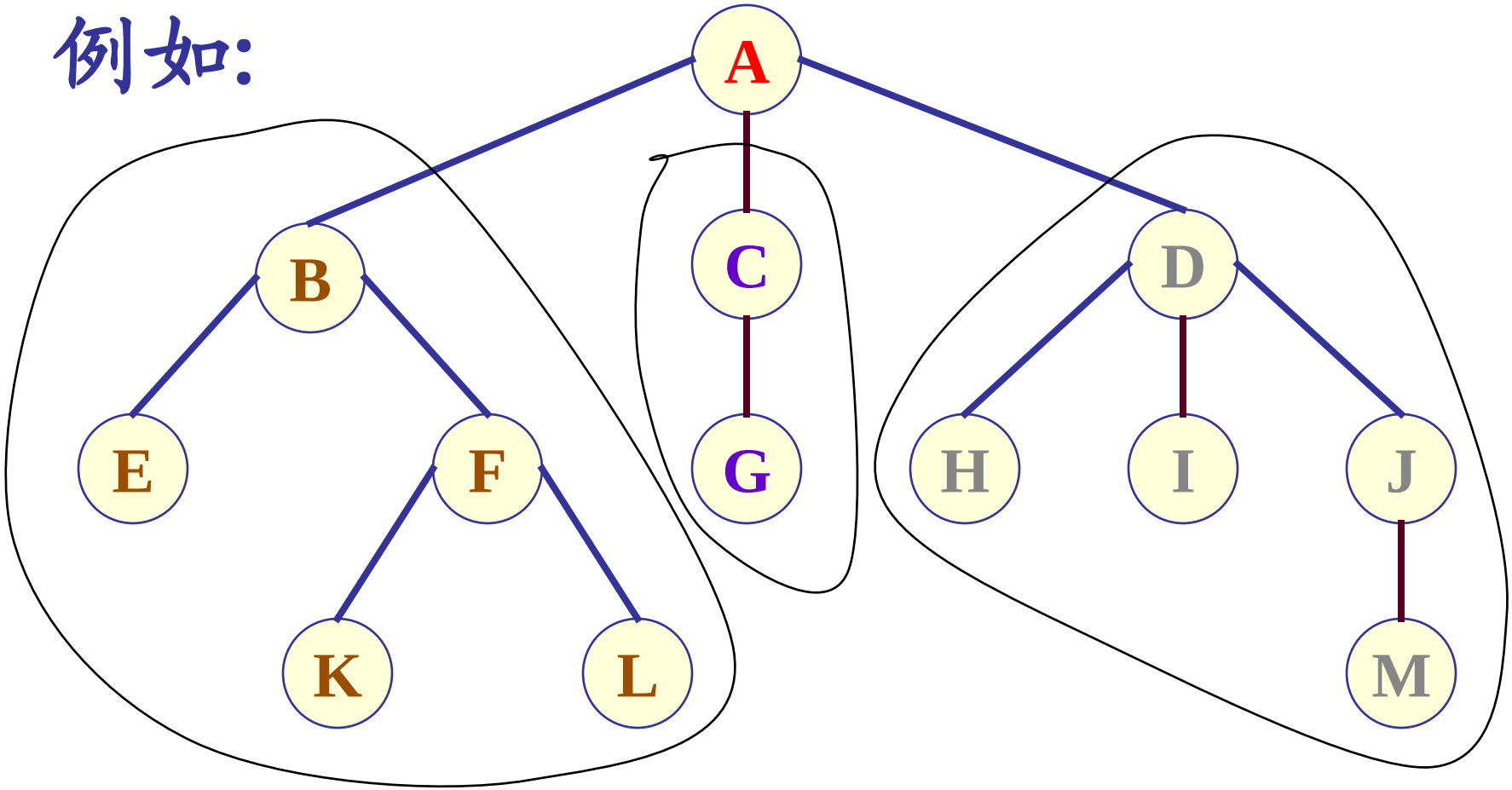
ClearTree(&T) // 将树清空

DestroyTree(&T) // 销毁树的结构

DeleteChild(&T, &p, i)

// 删除结点p的第i棵子树

例如:

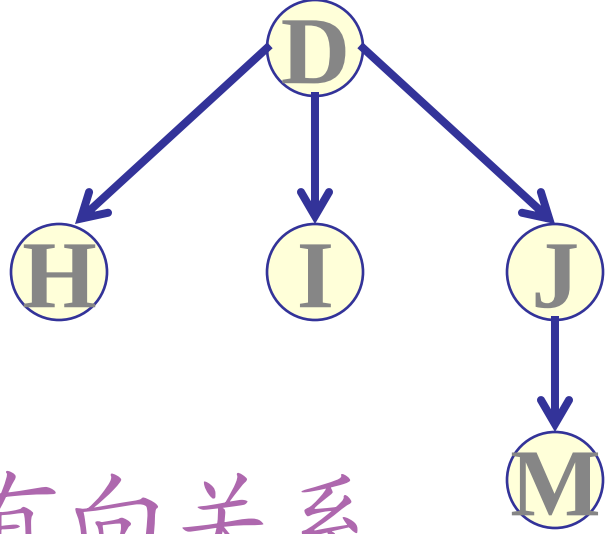


A(B(E, F(K, L)), C(G), D(H, I, J(M)))

树根

T_1 T_2 T_3

有向树:



(1)有确定的根;

(2)树根和子树根之间为有向关系。

有序树:

子树之间存在确定的次序关系。

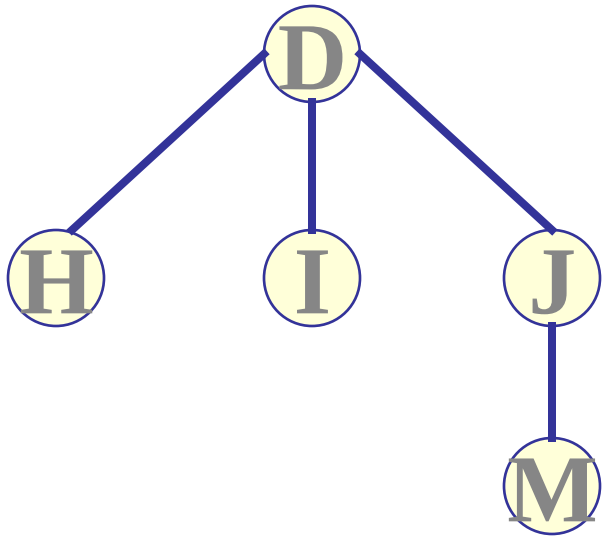
无序树:

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

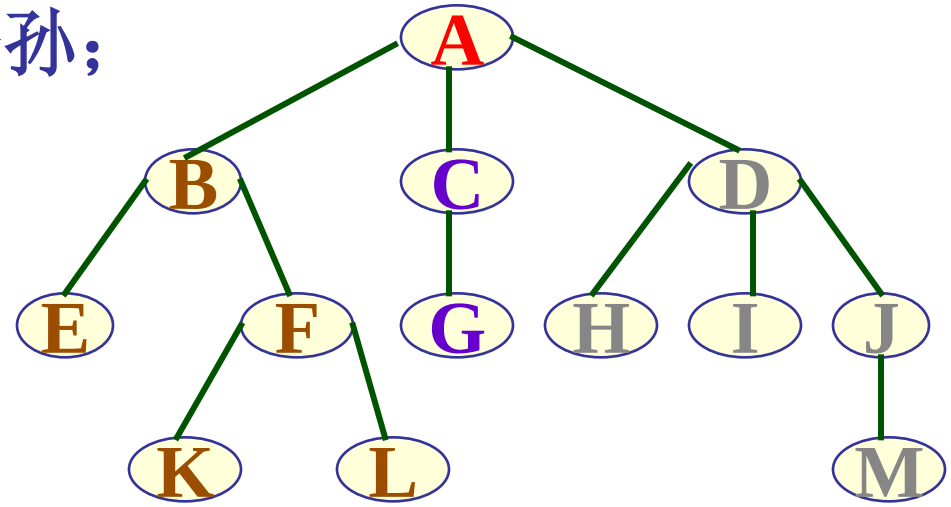
线性结构	树型结构
第一个数据元素 (无前驱)	根结点 (无前驱)
最后一个数据元素 (无后继)	多个叶子结点 (无后继)
其它数据元素 (一个前驱、 一个后继)	其它数据元素 (一个前驱、 多个后继)

三、基本术语

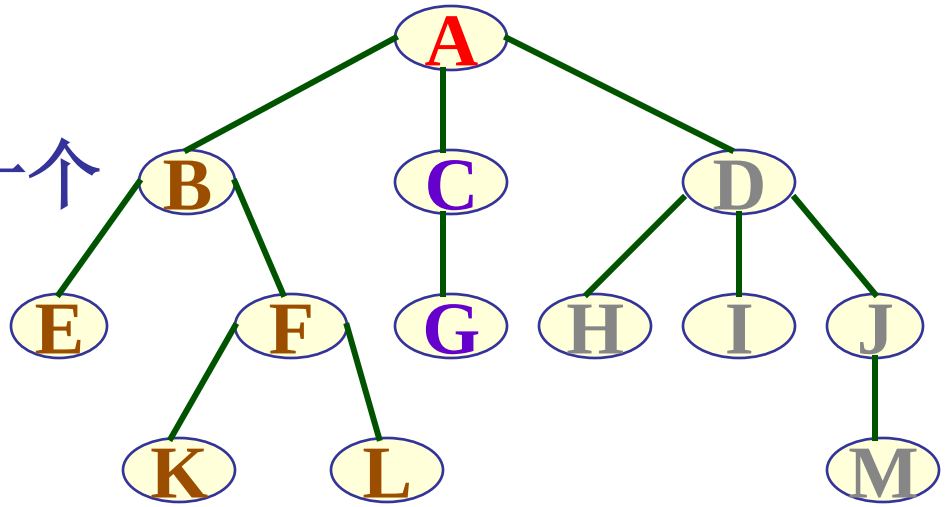


- 树的结点：包含一个数据元素及若干个指向其子树的分支；
- 结点的度：一个结点拥有的子树数目(分支数)；
- 树的度：一棵树上所有结点度的最大值；
- 叶子结点（终端结点）：度为零的结点；
- 分支结点（非终端结点）：度大于零的结点；
- (从根到结点的)路径：由从根到该结点所经分支和结点构成；

- 孩子结点：结点的子树的根称为该结点的孩子结点；
- 双亲结点：相应地，该结点称为孩子的双亲结点；
- 兄弟：具有同一父结点的子结点互称兄弟；
- 堂兄弟：其双亲在同一层的结点互为堂兄弟；
- 祖先结点：从根到该结点所经分支上的所有结点；
- 子孙结点：以某结点为根的子树中任一结点都称为该结点的子孙；



- 结点的层次：从根结点到该结点所经过的路径长度加1；
- 树的深度：树中叶子结点具有的最大层次数；
- 有序树：如果将树中结点的各子树看成从左至右是有次序的（即不能互换），则称该树为有序树；
- 第一个孩子：在有序树中，最左边的子树的根称为第一个孩子；
- 最后一个孩子：最右边的子树的根称为最后一个孩子；



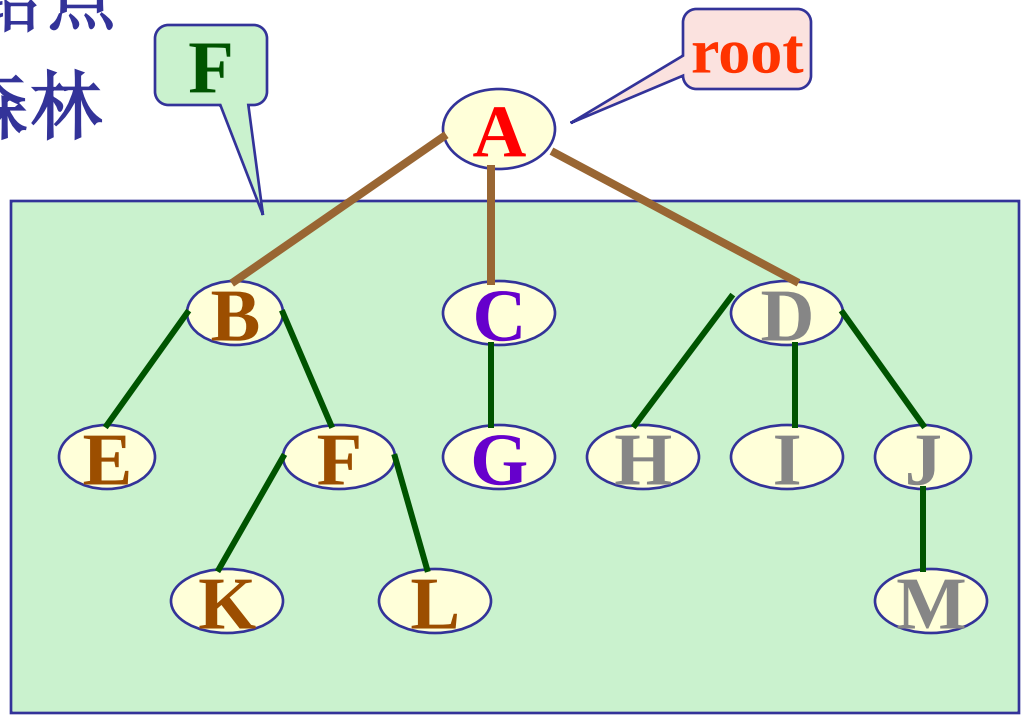
- **森林**：是m(m ≥0)棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。

任何一棵非空树是一个二元组

Tree = （root，F）

其中：root 被称为根结点

F 被称为子树森林



6.2 二叉树的类型定义

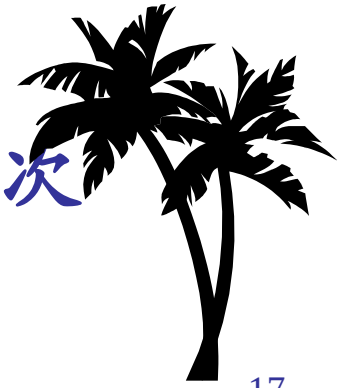
一、二叉树的定义

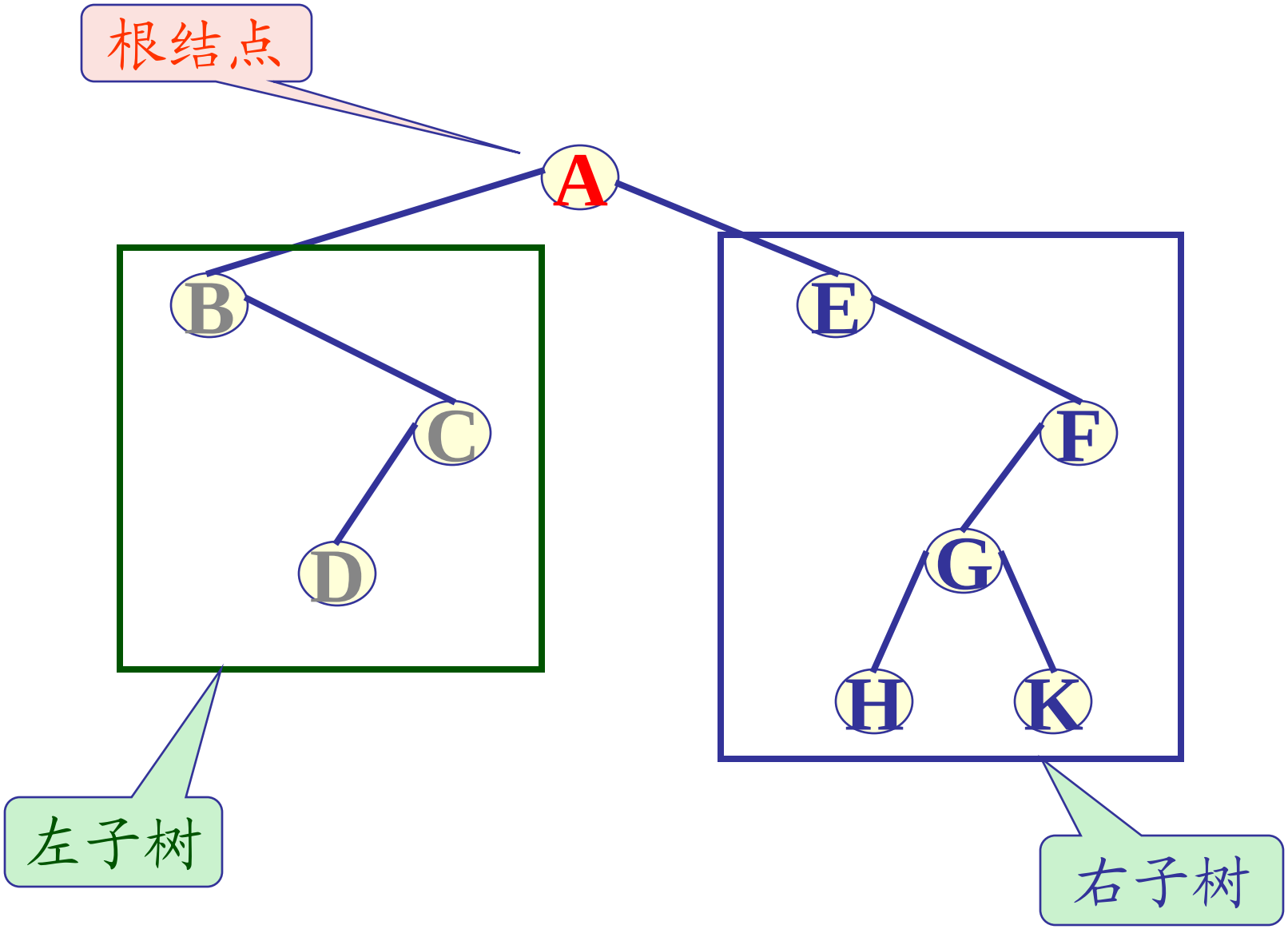
二叉树是n（n≥0）个结点的有限集合，这个集合或是空集，或是由一个根结点以及两棵互不相交的、被称为根的左子树和右子树所组成；左子树和右子树分别又是一棵二叉树。

特点：

每个结点至多只有两棵子树，

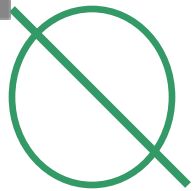
且二叉树的子树有左右之分，其次序不能任意颠倒，互不相交。




根结点
A
B
C
D
E
F
G
H
K
左子树
右子树

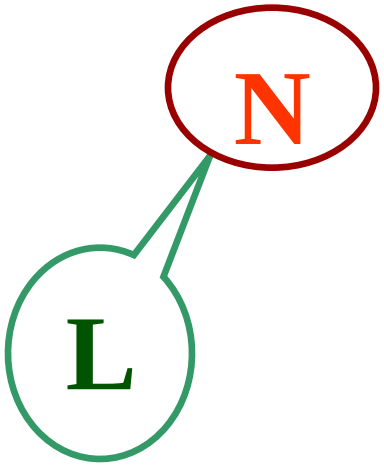
二、二叉树的五种基本形态:

空树

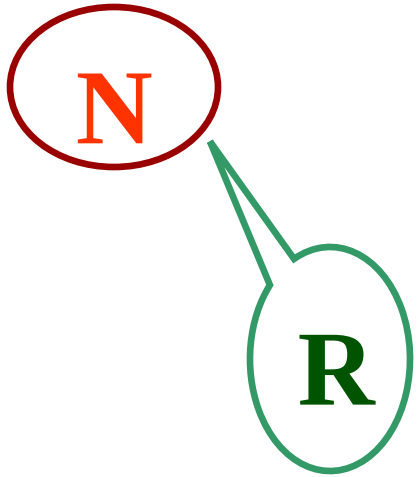


只含根结点

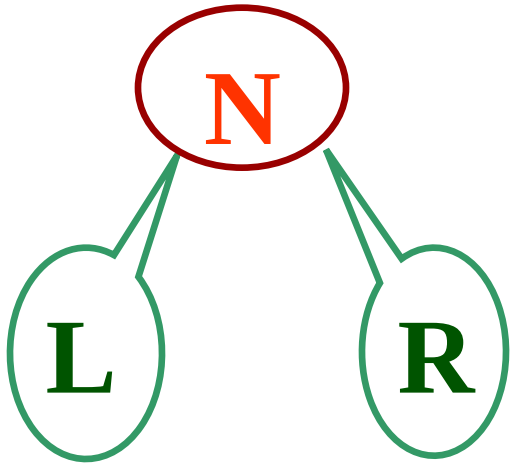
右子树为空树



左子树为空树



左右子树均不为空树



三、二叉树的主要基本操作:

- 查 找 类
- 插 入 类
- 删 除 类

查找类:

Root(T) //求根函数;

Value(T, e); //求结点函数;

Parent(T, e); //求双亲函数;

LeftChild(T, e)或RightChild(T, e); //求孩子结点函数;

LeftSibling(T, e)或RightSibling(T, e); //求兄弟函数;

BiTreeEmpty(T); //树判空函数;

BiTreeDepth(T); //求树深度函数;

PreOrderTraverse(T, Visit()); //先序遍历操作

InOrderTraverse(T, Visit()); //中序遍历操作

PostOrderTraverse(T, Visit()); //后序遍历操作

LevelOrderTraverse(T, Visit()); //层序遍历操作

插入类

InitBiTree(&T); //构造空的二叉树;

Assign(T, &e, value); //结点赋值函数

CreateBiTree(&T, definition); //建树函数;

InsertChild(T, p, LR, c); //插入P的左/右子树操作;

删除类

ClearBiTree(&T); // 将二叉树置为空树。

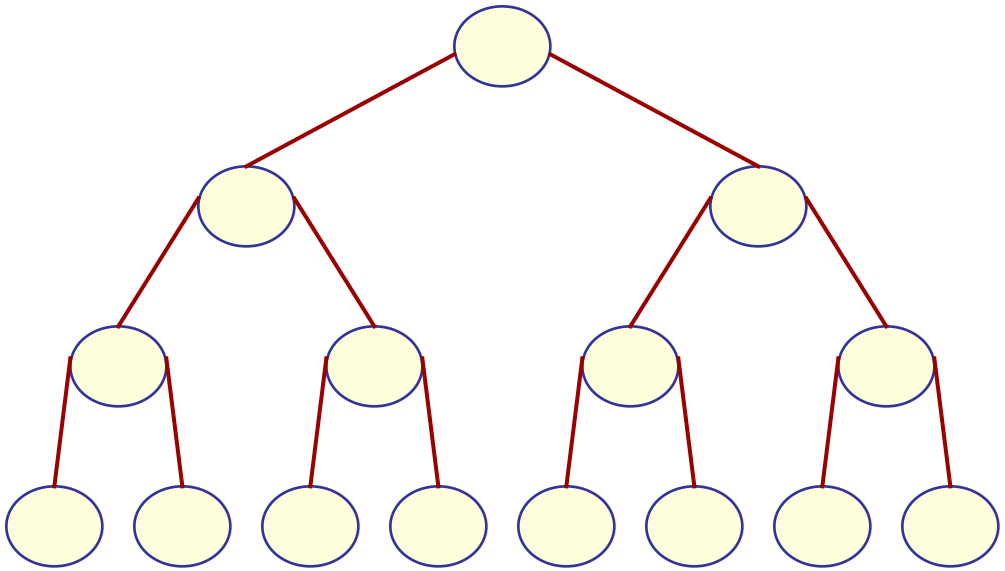
DestroyBiTree(&T); // 销毁二叉树

DeleteChild(T, p, LR); // 删除子树操作;

四、二叉树的重要特性:

- **性质 1：**

在二叉树的第 i 层上至多有 2^{i-1} 个结点。($i \geqslant 1$)



用归纳法证明性质1:

在二叉树的第 i 层上至多有2^{i-1}个结点。(i≥1)

证明:

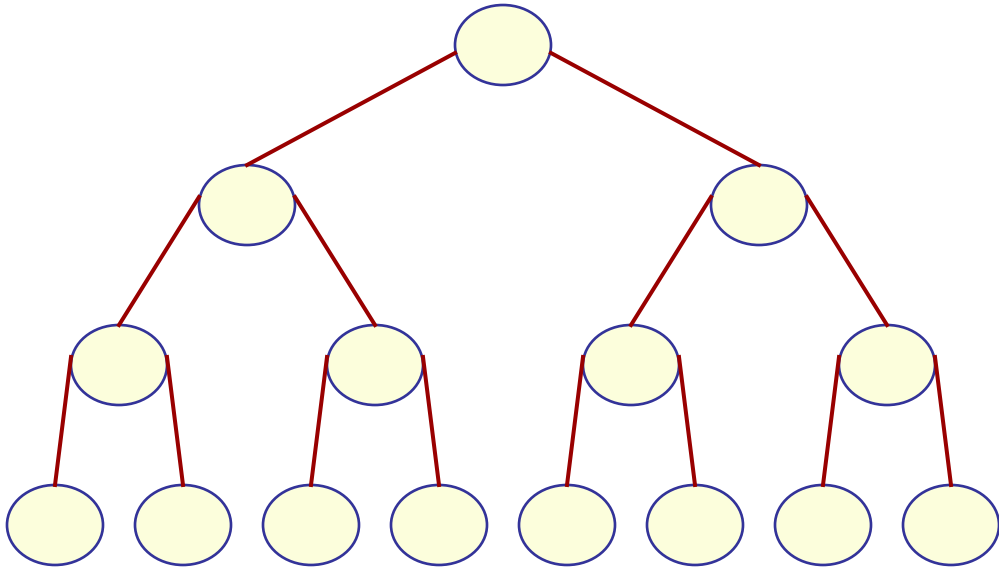
1、*i = 1* 时，只有一个根结点，显然

$2^{i-1} = 2^0 = 1$ 正确;

2、假设第i-1层上至多有2^{i-2}个结点成立，
由于二叉树的每个结点的度至多为2，
故在第i层上的最大结点数为第i-1层上的最大结点数的2倍，即$2*2^{i-2}=2^{i-1}$
证明完毕。

- **性质 2：**

深度为 k 的二叉树上至多含 2^k-1 个结点（k≥1）。



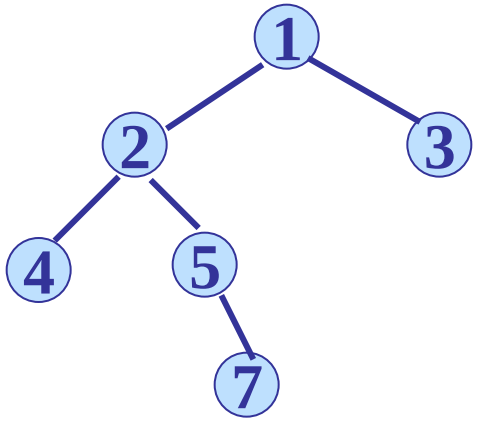
证明：

基于上一条性质，深度为 k 的二叉树上的结点数至多为：

$$2^0+2^1+\cdots\cdots+2^{k-1}=2^k-1$$ 。

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：

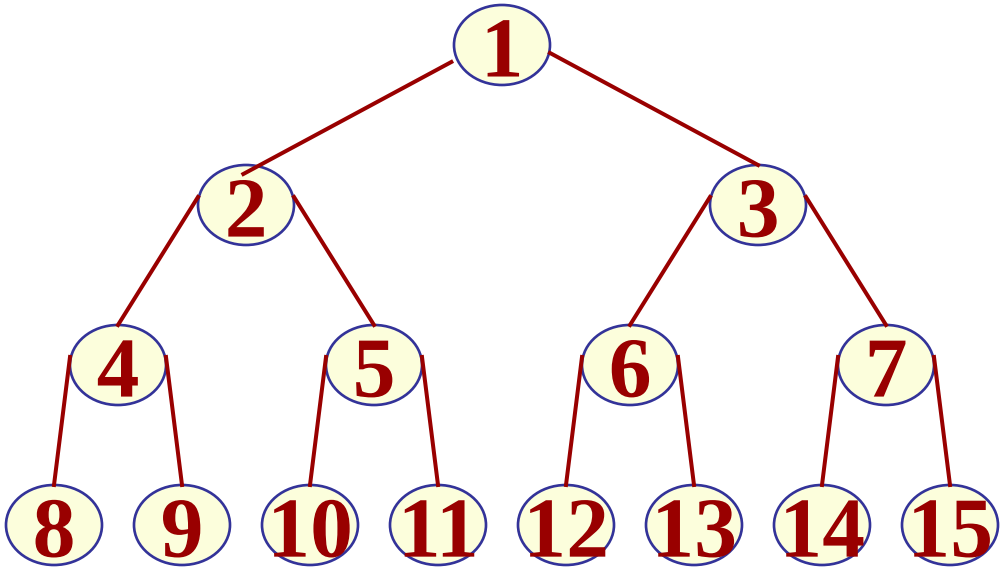
- 设 二叉树上结点总数 $n = n_0 + n_1 + n_2$
- 又 二叉树上分支总数 $b = n_1+2n_2$（1）
- 除根结点之外，其余结点都有一个分支进入，所以 $b = n-1$ ，把n代入，$b= n_0 + n_1 + n_2 - 1$（2）
- 由此（1）-（2），得 $n_0 = n_2 + 1$ 。

两类特殊的二叉树：满二叉树和完全二叉树

满二叉树：指的是深度为k且含有2^k-1个结点的二叉树。

特点：

每一层上的结点数都是最大结点数。

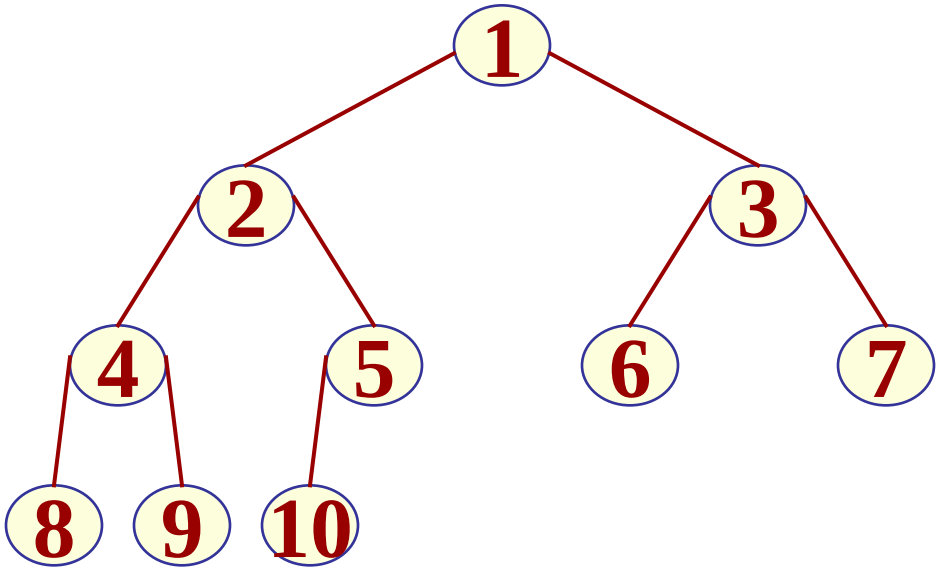


完全二叉树：树中所含的 ***n*** 个结点和满二叉树中**编号为 *1* 至 *n* 的结点一一对应**。

特点：

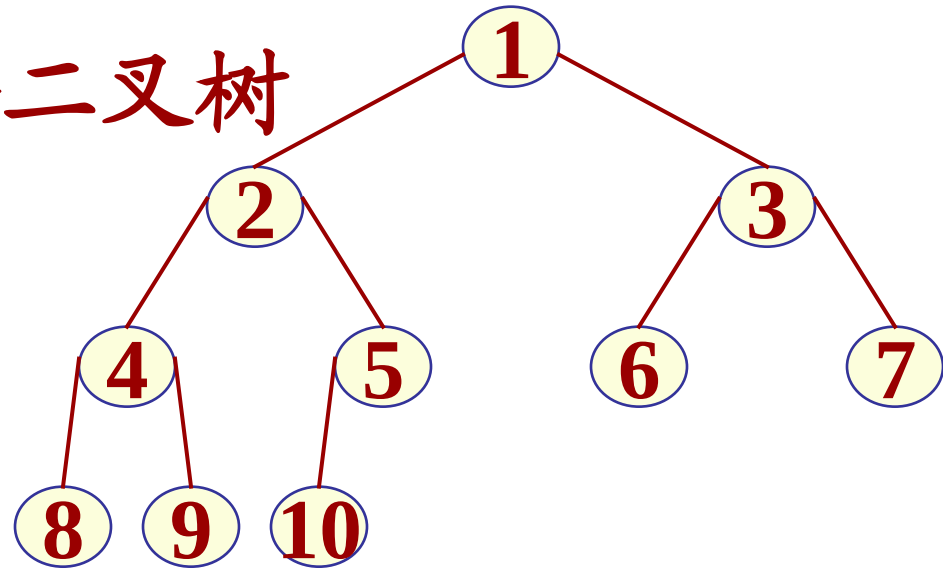
(1) 叶子结点只可能在层次最大的两层上出现；

(2) 对任意结点，若其右分支下的子孙的最大层次为L，则其左分支下的子孙的最大层次必为L或L+1 ；



性质4：（适合于完全二叉树）

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



证明：

- 设完全二叉树的深度为 k
- 则根据第二条性质得 $2^{k-1} \leq n < 2^k$

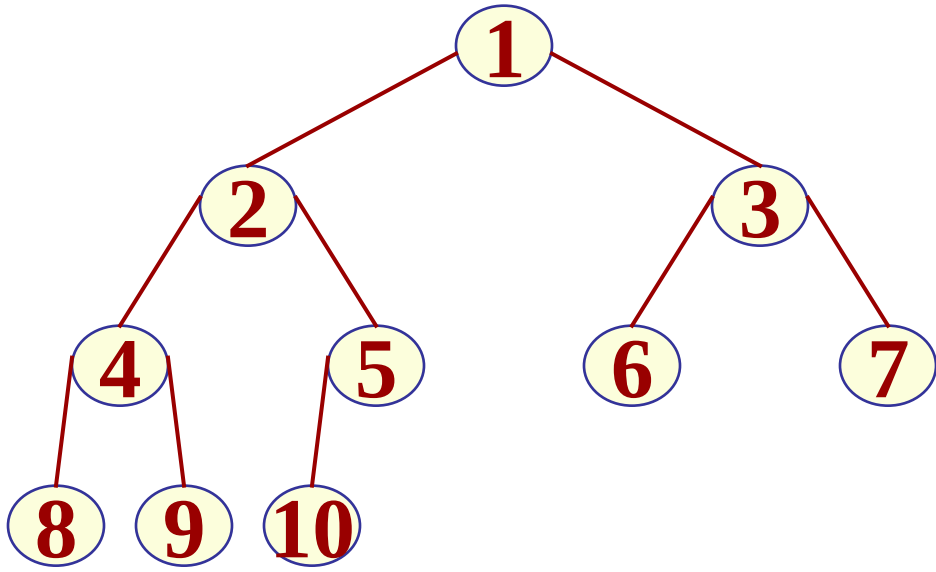
  即 $k-1 \leq \log_2 n < k$
- 因为 k 只能是整数，因此，$k = \lfloor \log_2 n \rfloor + 1$ 。

• 性质 5: (适合于完全二叉树)

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，则该结点是二叉树的根，无双亲，

否则，编号为 ⌊i/2⌋ 的结点为其双亲结点;

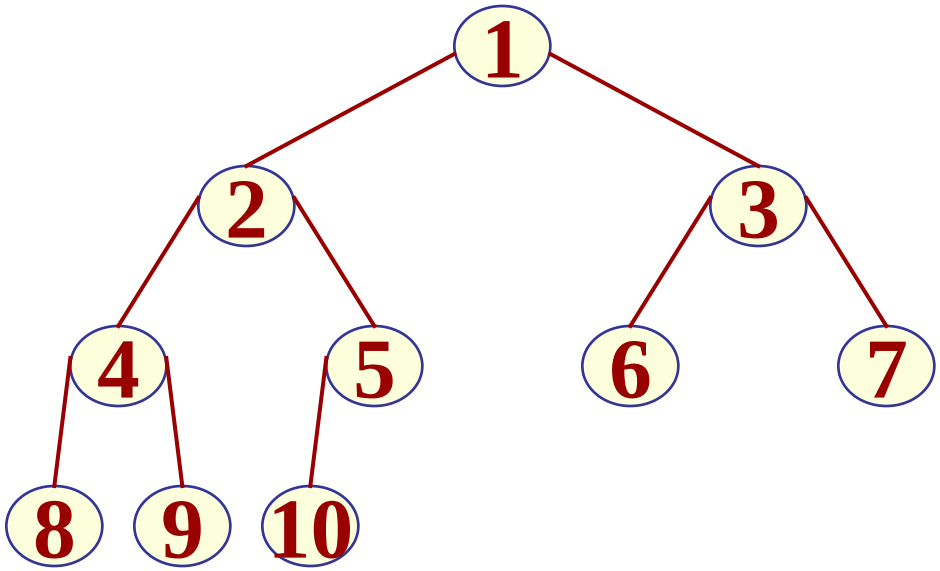


(2) 若 2i>n，则该结点无左孩子，

否则，编号为 2i 的结点为其左孩子结点；

(3) 若 2i+1>n，则该结点无右孩子结点，

否则，编号为2i+1 的结点为其右孩子结点。



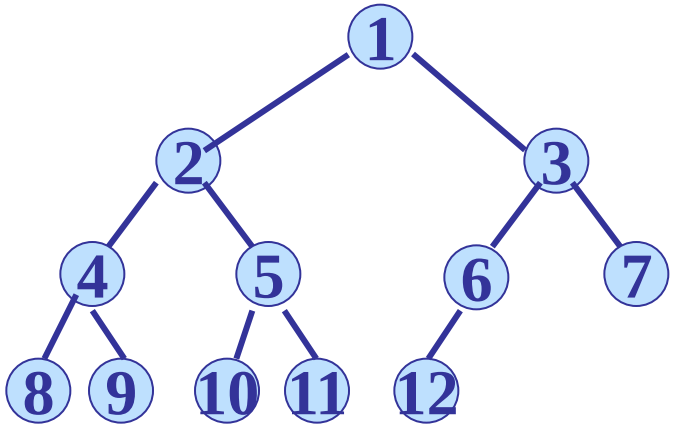
6.3 二叉树的存储结构

一、 二叉树的顺序存储表示

二、二叉树的链式存储表示

一、二叉树的顺序存储表示

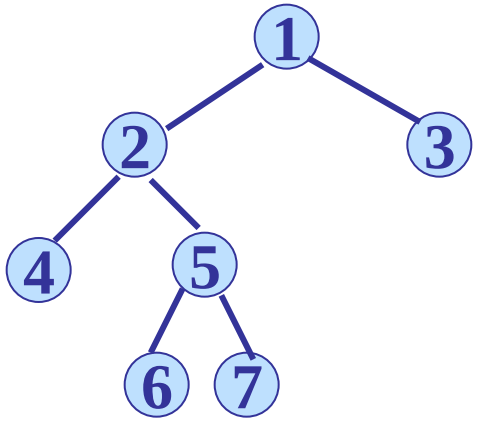
用一组地址连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素。对于一般二叉树，则应将其每个结点与完全二叉树上的结点相对照，存储在一维数组的相应分量中。



1	2	3	4	5	6	7	8	9	10	11	12

完全二叉树

顺序存储适用于完全二叉树



1	2	3	4	5	0	0	0	0	6	7

一般二叉树

（0表示不存在的结点）

用C语言描述:

```
#define  MAX_TREE_SIZE  100
                                    // 二叉树的最大结点数
typedef TElemType  SqBiTree[MAX_TREE_SIZE];
                                    // 0号单元存储根结点
SqBiTree  bt;
```

二、二叉树的链式存储表示

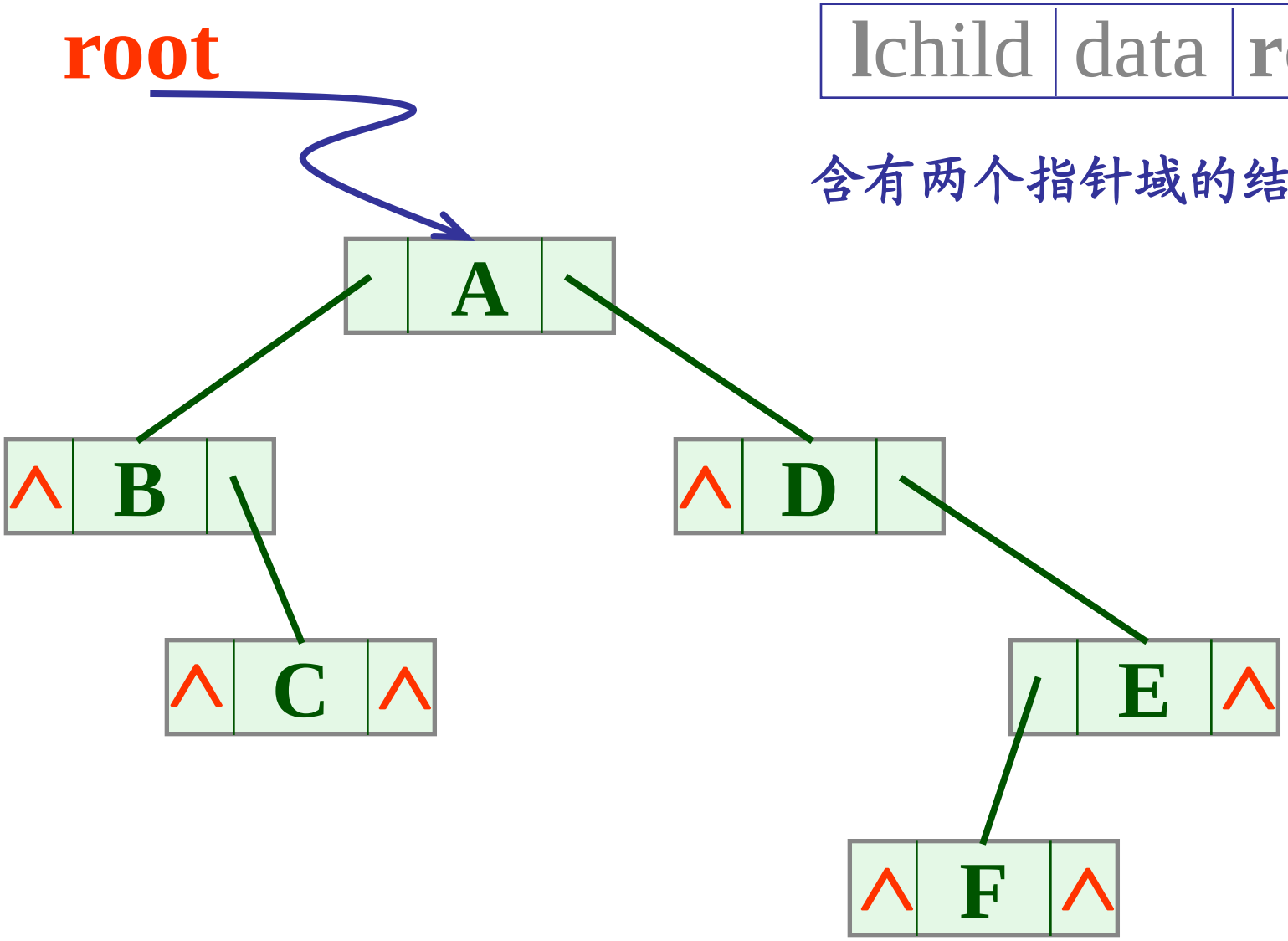
1. 二叉链表
2. 三叉链表
3. 双亲链表
4. 线索链表(见下一节)

1. 二叉链表

结点结构:

lchild	data	rchild

含有两个指针域的结点结构



C 语言的类型描述如下:

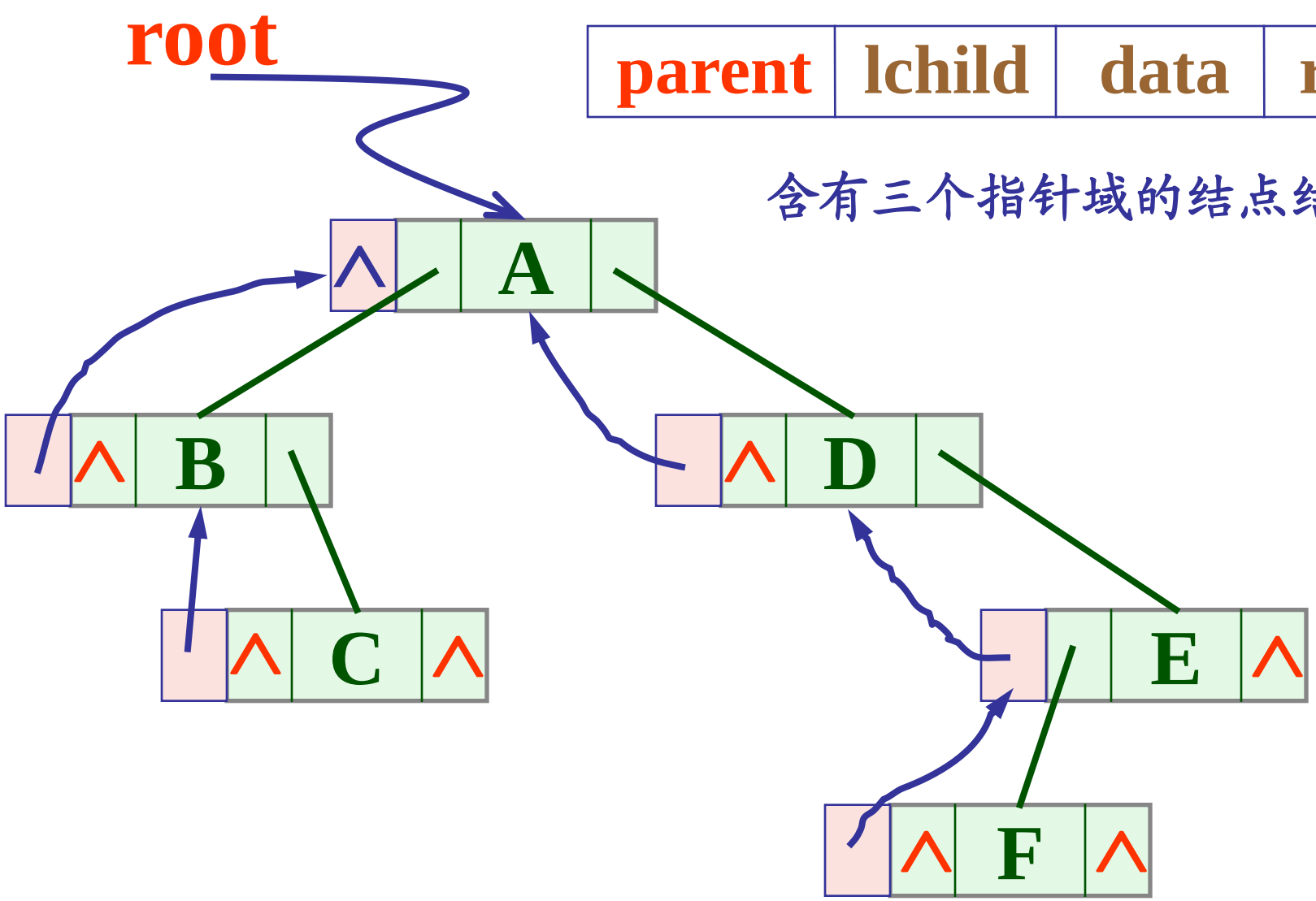
```
typedef struct BiTNode { // 结点结构
    TElemType      data;
    struct BiTNode  *lchild, *rchild;
                                  // 左右孩子指针
} BiTNode, *BiTree;
```

2. 三叉链表

结点结构:

parent	lchild	data	rchild

含有三个指针域的结点结构

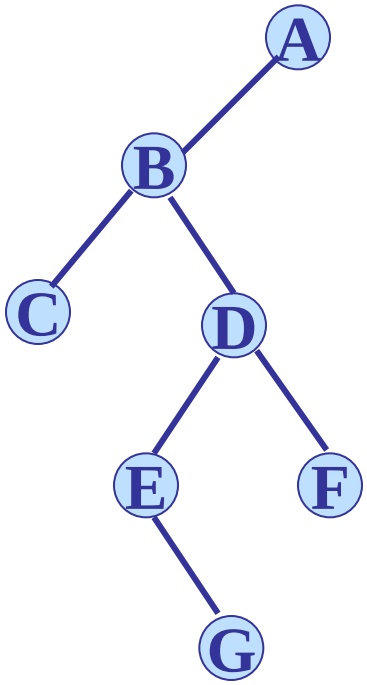


C 语言的类型描述如下:

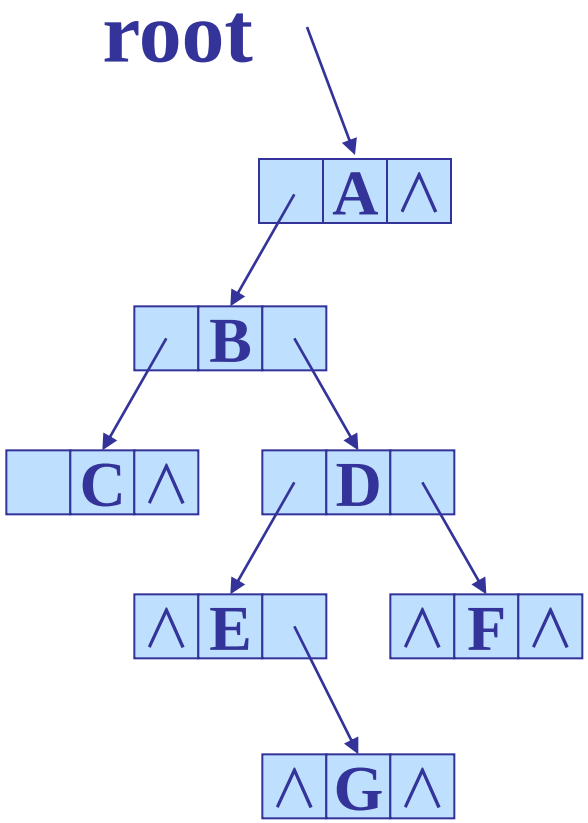
```
typedef struct TriTNode { // 结点结构
    TElemType       data;
    struct TriTNode  *lchild, *rchild;
                                  // 左右孩子指针
    struct TriTNode  *parent;  //双亲指针
} TriTNode, *TriTree;
```

结点结构:

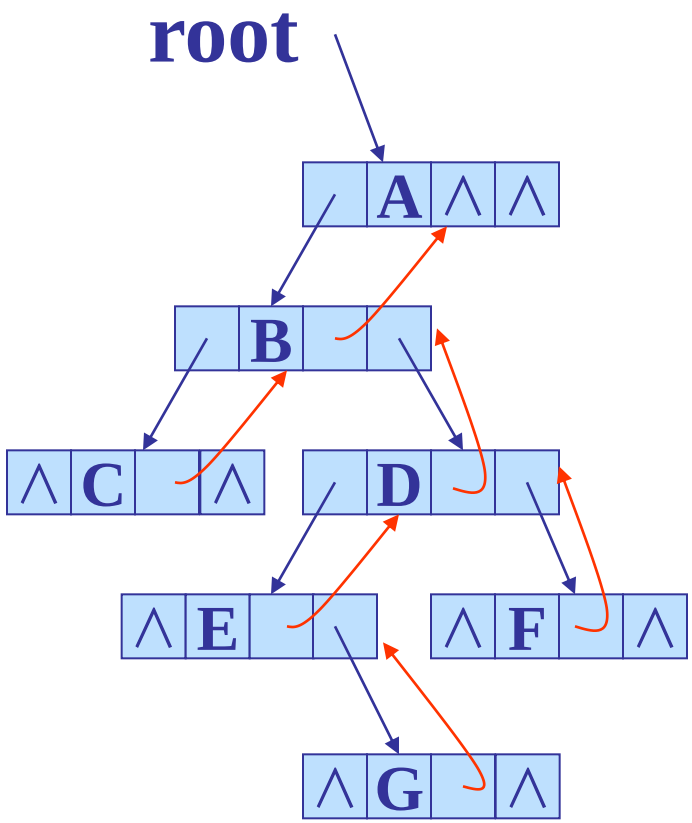
lchild	data	parent	rchild



二叉树



二叉链表



三叉链表

3. 双亲链表

结点结构:

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

```
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
```

6.4 二叉树的遍历

一、问题的提出

二、先左后右的遍历算法

三、算法的递归描述

四、遍历算法的非递归描述

五、遍历算法的应用举例

一、问题的提出

顺着某一条搜索路径**巡访**二叉树中的结点，使得每个结点**均被访问一次**，而且**仅被访问一次**。

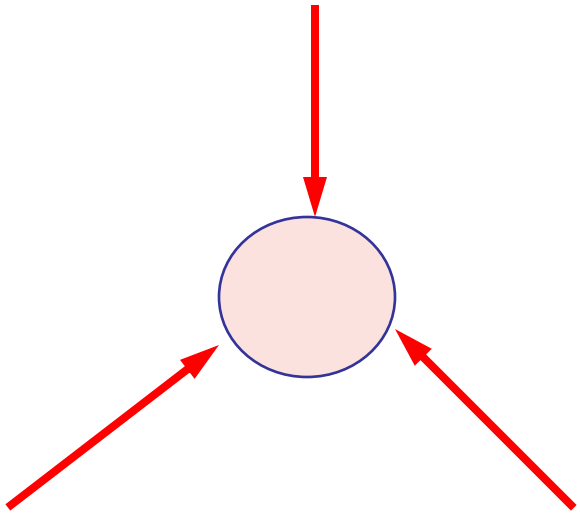
“访问”的含义可以很广，可以对结点作各种处理，如：输出结点的信息等。

“**遍历**”是任何类型均有的操作，对线性结构而言，只有一条搜索路径(因为每个结点均只有一个后继)，故不需要另加讨论。而二叉树是非线性结构，每个结点有两个后继，则存在如何遍历即按什么样的搜索路径遍历的问题。

对“二叉树”而言，可以有三条搜索路径：

- 1. **先上后下**的按层次遍历；
- 2. **先左**(子树)**后右**(子树)的遍历；
- 3. **先右**(子树)**后左**(子树)的遍历。

二、先左后右的遍历算法



先（根）序的遍历算法

中（根）序的遍历算法

后（根）序的遍历算法

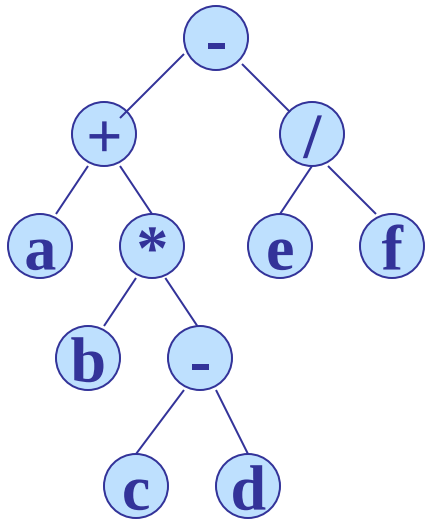
先（根）序的遍历算法:

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。



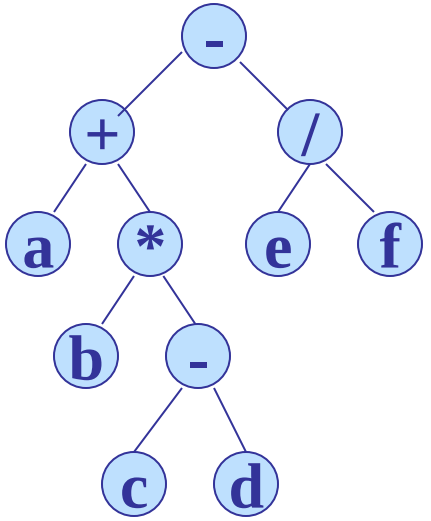
中（根）序的遍历算法:

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。



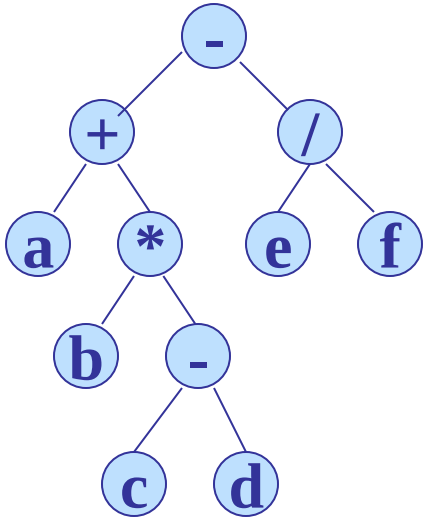
后（根）序的遍历算法:

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。



三、算法的递归描述

先序遍历二叉树算法

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

中序遍历二叉树算法

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

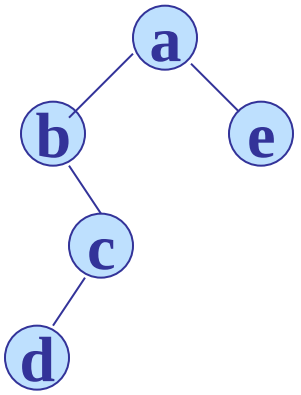
后序遍历二叉树算法

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

递归算法简明精练，但效率较低。

四、遍历二叉树的非递归算法

在非递归算法中要用栈来保存遍历过程中经历的结点的左孩子和右孩子。



使用栈来实现中序遍历二叉树的基本思想：

从二叉树的根结点开始，沿左子树一直走到末端为止，在向前搜索的过程中将所遇结点进栈，待遍历完左子树时，从栈顶退出结点并访问，然后再遍历右子树。

1、中序遍历算法的非递归描述(1)

```
BiTNode *GoFarLeft(BiTree T, Stack *S)
{   if (!T )  return NULL;
    while (T->lchild ){          // 找到最左下的结点
        Push(S, T);
        T = T->lchild;
     }
    return T;

}
```

```
void Inorder_I(BiTree T, void (*visit)
        (TelemType& e)) // 中序遍历二叉树的非递归算法一
{   Stack *S;
    t = GoFarLeft(T, S);          // 找到最左下的结点
    while(t)
    { visit(t->data);
      if (t->rchild)
          t = GoFarLeft(t->rchild, S);
      else if ( !StackEmpty(S ))    // 栈不空时退栈
          t = Pop(S);
              else    t = NULL;       // 栈空表明遍历结束
    } // while
}// Inorder_I
```

中序遍历算法的非递归描述(2)

算法思想:

假设二叉树采用二叉链表存储，用一个顺序栈保存返回的结点，先扫描根结点的所有左结点并入栈，出栈一个结点，访问之，然后扫描该结点的右结点并入栈，再扫描该右结点的所有左结点并入栈，如此这样，直到栈空为止。

实现本题功能的函数如下:

```
void inorder( BiTree  b)    // 中序遍历二叉树的非递归算法二
  { BiTree *stack[m0],*p;
    int top=0;
    p=b;
    do
     { while (p!=NULL)                    //扫描所有左结点
         { top++;
           stack[top]=p;
           p=p->lchild;
         }
```

```
        if (top>0)
         { p=stack[top];
           //p所指结点为无左子树的结点或其左子树已遍历过
             top--;
             printf("%d",p->data); //访问结点
             p=p->rchild;              //扫描右子树
           }
    } while (p!=NULL || top!=0)
}
```

2、前序遍历的非递归算法

假设二叉树采用二叉链表存储,使用一个栈 stack实现非递归的前序遍历。

算法思想:

先分析前序遍历时结点输出顺序的特点(根先输出,右孩子后输出)

实现本题功能的函数如下:

```
void preorder( BiTree  b) // 前序遍历二叉树的非递归算法
  { BiTree *stack[m0];
    int top;
    if (b!=NULL)
     { top=1;                    //根结点入栈
       stack[top]=b;
       while (top>0)             //栈不为空时循环
         { p=stack[top];  //退栈并访问该结点
           top--;
           printf("%d",p->data);
```

```
        if (p->rchild!=NULL)     //右孩子入栈
          { top++;
            stack[top]=p->rchild;}
        if (p->lchild!=NULL)    //左孩子入栈
          { top++;
            stack[top]=p->lchild;
          }
      }//while
   }//if
}
```

3、后序遍历的非递归算法

算法思想:

1. 用一个栈保存返回的结点;

2. 先扫描根结点的所有左结点并入栈，出栈一个结点，然后扫描该结点的右结点并入栈，再扫描该右结点的所有左结点并入栈，

3. 当一个结点的左右子树均访问后再访问该结点，如此这样，直到栈空为止。

如果从左子树返回到根，下一步是继续遍历右子树；如果从右子树返回到根，则应访问根结点。这就产生了一个问题：在退栈回到根结点时如何区别是从左子树返回还是从右子树返回？

这里采用两个栈 **stack** 和 **tag**，并用一个共同的栈顶指针，一个存放指针值，一个存放左右子树标志（**0**为左子树，**1**为右子树）。退栈时在退出结点指针的同时区分是遍历左子树返回的还是遍历右子树返回的，以决定下一步是继续遍历右子树还是访问根结点。

```
void postorder( BiTree  b) // 后序遍历二叉树的非递归算法
{ BiTree *stack[m0],*p;
  int tag[m0], top=0;   p=b;
  do
    { while (p!=NULL)    //扫描左结点
       { top++;   stack[top]=p; tag[top]=0;
         p=p->lchild; }
      while ((top>0) && tag[top]==1) // p的左右子树都访问过
         { printf("%d", stack[top] ->data); top--; }//访问结点并出栈
      p=stack[top];
      if ((top>0) && (tag[top]==0))  //扫描右子树
         { p=p->rchild; tag[top]=1; }
    } while (p!=NULL || top!=0)
  }
```

4、按层次顺序（同一层次自左至右）遍历二叉树的算法

算法思想:

假设二叉树采用二叉链表结构，依题意，本算法要采用一个队列q，先将二叉树根结点入队列，然后出队列，输出该结点；若它有左子树，便将左子树根结点入队列；若它有右子树，便将右子树根结点入队列，如此直到队列空为止。因为队列的特点是先进先出，从而达到按层次顺序遍历二叉树的目的。

实现本题功能的函数如下:

```
#define MAXLEN 100
void translevel(BiTree b)          //按层次遍历
  {  struct node
      { BiTree *vec[MAXLEN];
        int f,r;
      } q;
     q.f=0;
     q.r=0;                              //置队列为空队列
     if (b!=NULL) printf("%d",b->data);
       q.vec[q.r]=b;                     //结点指针进入队列
       q.r=q.r+1;
```

```
while (q.f<q.r)                    //队列不为空
 { b=q.vec[q.f];   q.f=q.f+1;   //队头出队列
   if (b->lchild!=NULL)       //输出左孩子，并入队列
    {  printf("%d",b->lchild->data);
       q.vec[q.r]=b->lchild;
       q.r=q.r+1;
    }
   if  (b->rchild!=NULL)      //输出右孩子，并入队列
       { printf("%d",b->rchild->data);
         q.vec[q.r]=b->rchild;
         q.r=q.r+1;
       }
    }
  }
```

五、遍历算法的应用举例

1、统计二叉树中叶子结点的个数
(先序遍历)

2、求二叉树的深度(后序遍历)

自学

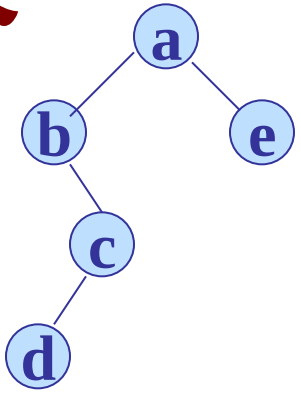
3、求二叉树值为X的祖先(后序遍历)

4、复制二叉树(后序遍历)

5、建立二叉树的存储结构

6、根据遍历序列构造二叉树

1、统计二叉树中叶子结点的个数



算法基本思想:

先序(或中序或后序)遍历二叉树，在遍历过程中查找叶子结点，并计数。由此，**需在遍历算法中增添一个“计数”的参数**，并将算法中“访问结点”的操作改为：**若是叶子，则计数器增1。**

```
void CountLeaf (BiTree T,  int& count)
                // 统计二叉树中叶子结点的个数
{   if ( T )
      {  if ((!T->lchild)&& (!T->rchild))
            count++;     // 对叶子结点计数

         CountLeaf( T->lchild, count);
         CountLeaf( T->rchild, count);

      } // if
} // CountLeaf
```

统计二叉树中叶子结点的个数方法二

```
Int CountLeaf (BiTree T)
                    //统计二叉树中叶子结点的个数方法二
{ if (T==NULL)  return 0;
    else if (T->lchild==NULL &&
                T->rchild==NULL)  return 1;
    else  return
      CountLeaf(T->lchild) +CountLeaf(T->rchild);
}
```

2、求二叉树的深度(后序遍历)

算法基本思想:

首先分析二叉树的深度和它的左、右子树深度之间的关系。

从二叉树深度的定义可知，二叉树的深度应为其左、右子树深度的最大值加1。由此，需先分别求得左、右子树的深度，算法中“访问结点”的操作为：求得左、右子树深度的最大值，然后加 1 。

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


5、建立二叉树的存储结构

建立二叉树存储结构有多种算法

(1) 以字符串的形式 根-左子树-右子树（先序形式） 定义一棵二叉树

(2) 根据表达式建相应的二叉树

(3) 由二叉树的先序和中序序列建二叉树

(1) 以字符串的形式 根-左子树-右子树（先序形式） 定义一棵二叉树

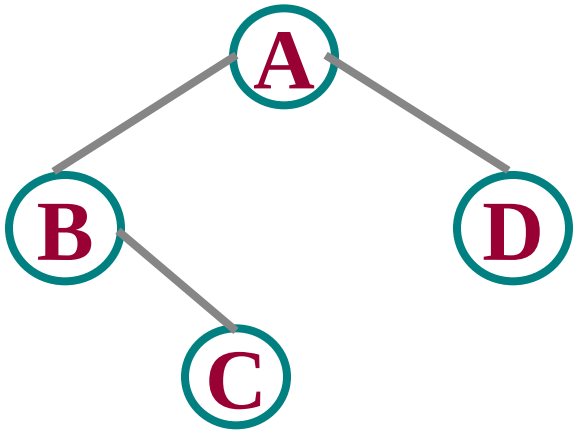
例如:

空树

以空白字符“■”表示

只含一个根结点的二叉树 A

以字符串“A(■■)”表示



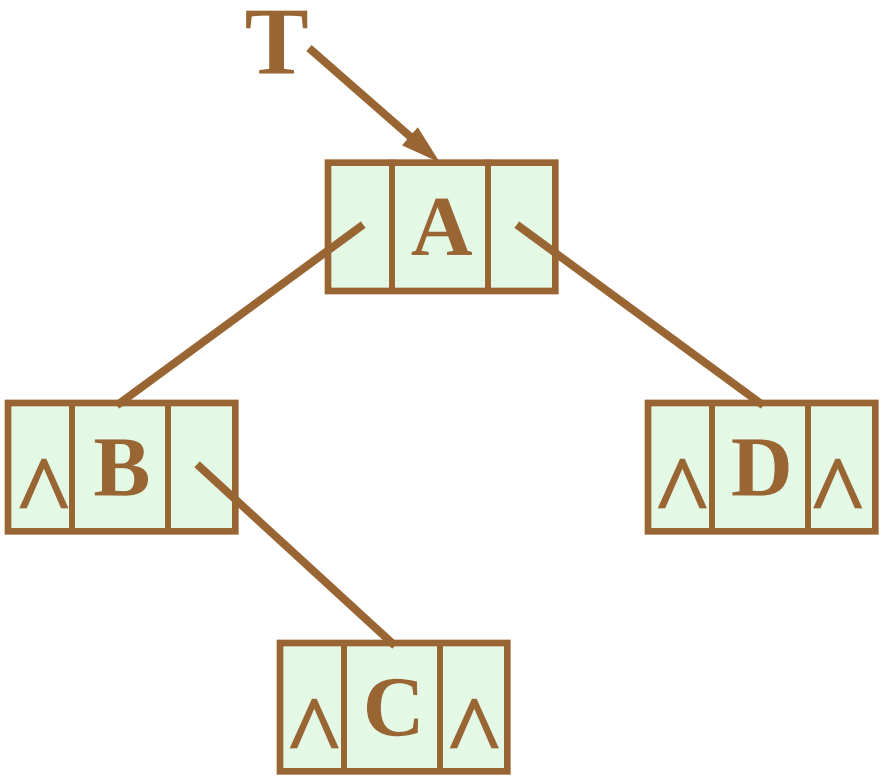
以下列字符串表示

A(B(■,C(■,■)),D(■,■))

```
Status CreateBiTree(BiTree &T) //按先序输入二叉树结点的值
{  scanf(&ch);
   if (ch==' ')  T = NULL;
   else {
      if (!(T = (BiTNode *)malloc(sizeof(BiTNode))))
               exit(OVERFLOW);
     T->data = ch;                          // 生成根结点
     CreateBiTree(T->lchild);     // 构造左子树
     CreateBiTree(T->rchild); }  // 构造右子树
   return OK;
} // CreateBiTree
```

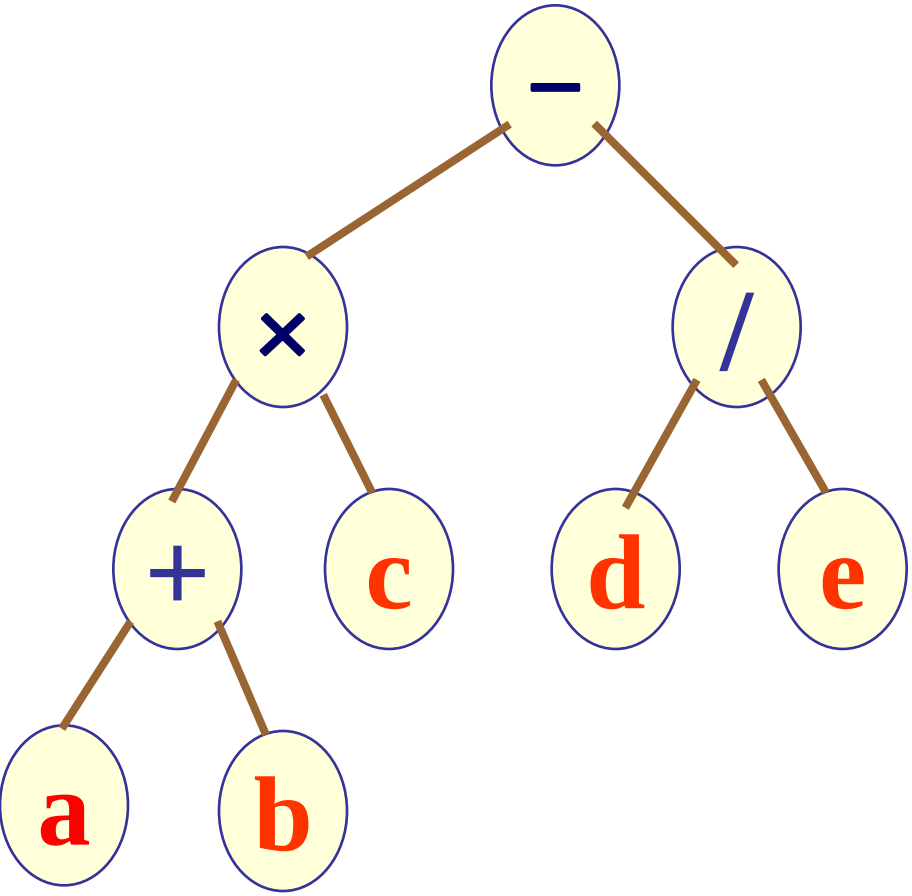
上页算法执行过程举例如下：

A B ■ C ■ ■ D ■ ■



(2) 根据表达式建相应的二叉树

已知表达式 (a+b) × c – d/e



先序遍历: - × +abc /de

---前缀表达式

中序遍历: a +b × c - d/e

---中缀表达式

后序遍历: a b +c × de / -

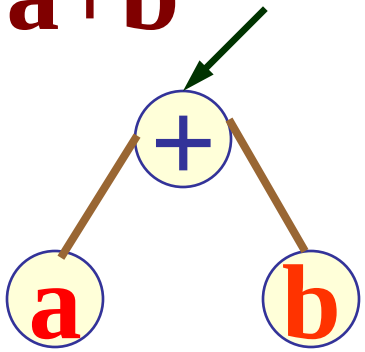
---后缀表达式

特点: 操作数为叶子结点

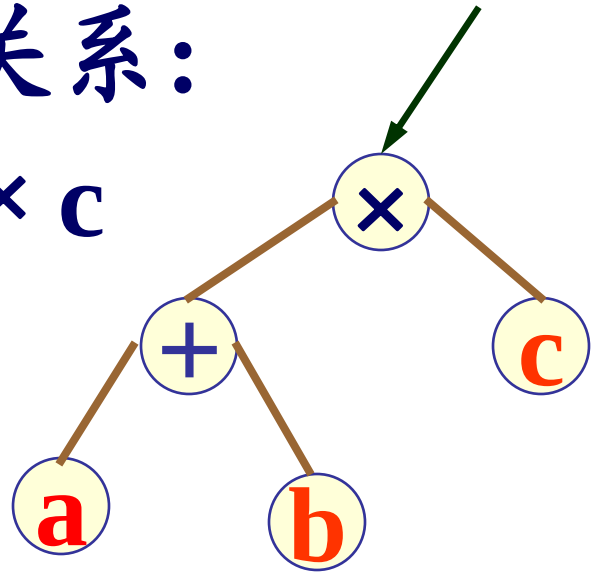
运算符为分支结点

分析表达式和二叉树的关系：

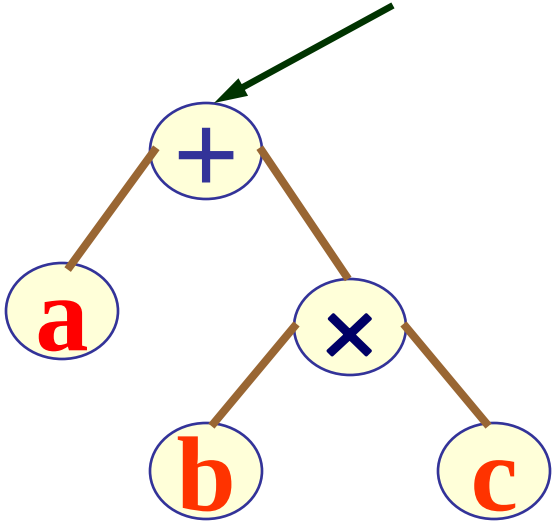
a+b



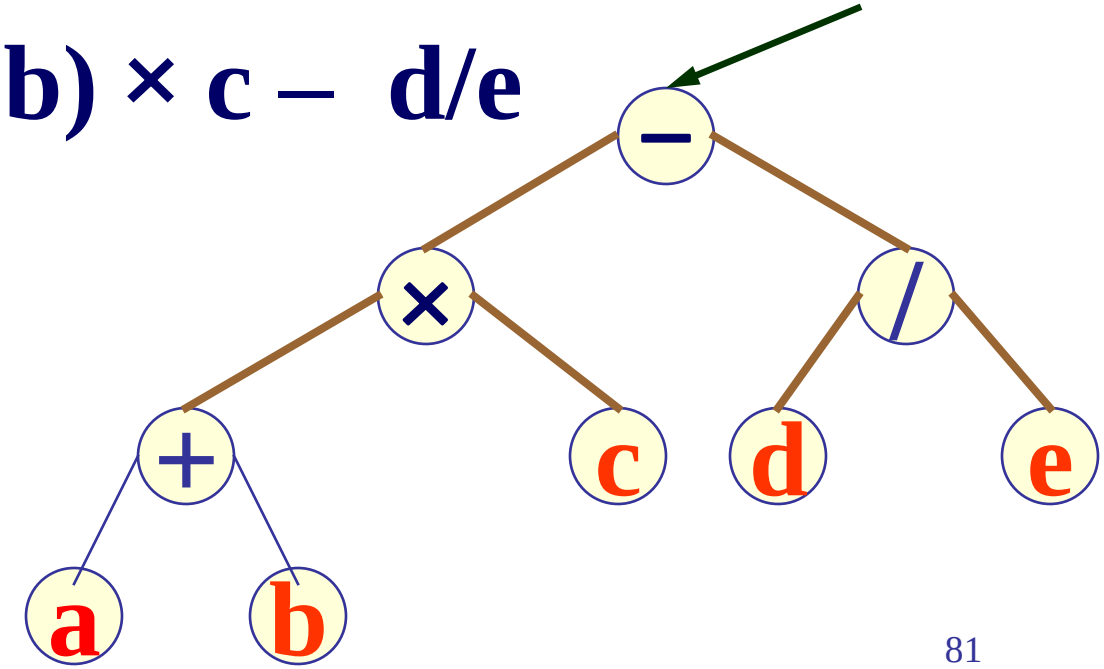
(a+b) × c



a+b × c



(a+b) × c – d/e



算法思想:

使用两个栈，一个存放叶子或子树，另一个存放运算符

1.读入一个字符;

2.如果是数, 建叶子结点,入树栈 暂存;

3.如果是运算符,则和字符栈的栈顶元素比较优先级:

1)若当前的优先级“高”,则入运算符栈暂存;

2)若栈顶的优先级“高”,则运算符栈的栈顶元素出栈(根),从树栈中弹出两个元素(右左孩子)建子树,新建的子树再入树栈保存

建叶子结点的算法为:

```
void CrtNode(BiTree& T,char ch)
{
  T=(BiTNode*)malloc(sizeof(BiTNode));
  T->data = ch;
  T->lchild = T->rchild = NULL;
  Push( PTR, T );          //叶子结点入PTR树栈保存
}
```

建子树的算法为:

```
void CrtSubtree (Bitree& T, char c)
{
   T=(BiTNode*)malloc(sizeof(BiTNode));
   T->data = c;
   Pop(PTR, rc);  T->rchild = rc;   //两个叶子结点出栈
   Pop(PTR, lc);  T->lchild = lc;   //建子树
        //遍历时先左后右，左子树先入栈，所以必须先弹出右子树
   Push(PTR, T);        //子树入PTR树栈保存
}
```

```
void CrtExptree(BiTree &T, char exp[] )
{ InitStack(S);  Push(S, '#');  InitStack(PTR);
  p = exp;  ch = *p; // PTR数栈放树和结点，S栈放字符和运算符
  while (!(GetTop(S)=='#' && ch=='#')) {
    if (!IN(ch, OP))   CrtNode( t, ch );//读入的是字符，
        //建立叶子结点并入栈， #号:表达式的开始和结束， OP: 运算符集合
    else {   … …}          //代表读入的是运算符
    if ( ch!= '#' ) { p++;  ch = *p;}//未结束时指针后移
    } // while
  Pop(PTR, T);      //从PTR栈中弹出最后的一棵树
} // CrtExptree
```

```
switch (ch) {
    case '(' : Push(S, ch); break; //表示读入的ch是“(”
    case ')' : Pop(S, c);           //表示读入的ch是“)”
               while (c!= '(' ) {   //栈顶不是“(”
                   CrtSubtree( t, c);  //按栈顶元素建子树并入栈
                   Pop(S, c) ; }
               break;
    defult :      … …    //表示读入的ch是其他运算符

} // switch
```

```
while(!Gettop(S, c) && ( precede(c,ch))) {
    //栈顶元素c的运算优先级大于刚读入的ch运算符的优先级
    CrtSubtree( t, c);   //按栈顶元素建子树
    Pop(S, c);           //弹出栈顶元素
   }
if ( ch!= '#' ) Push( S, ch);   //将ch运算符入栈
break;
```

(3) 由二叉树的先序和中序序列建树

仅知二叉树的先序序列“**abcdefg**”能否唯一确定一棵二叉树？ **不能**

如果同时已知二叉树的中序序列“**cbdaegf**”，则会如何？

二叉树的先序序列 

二叉树的中序序列 

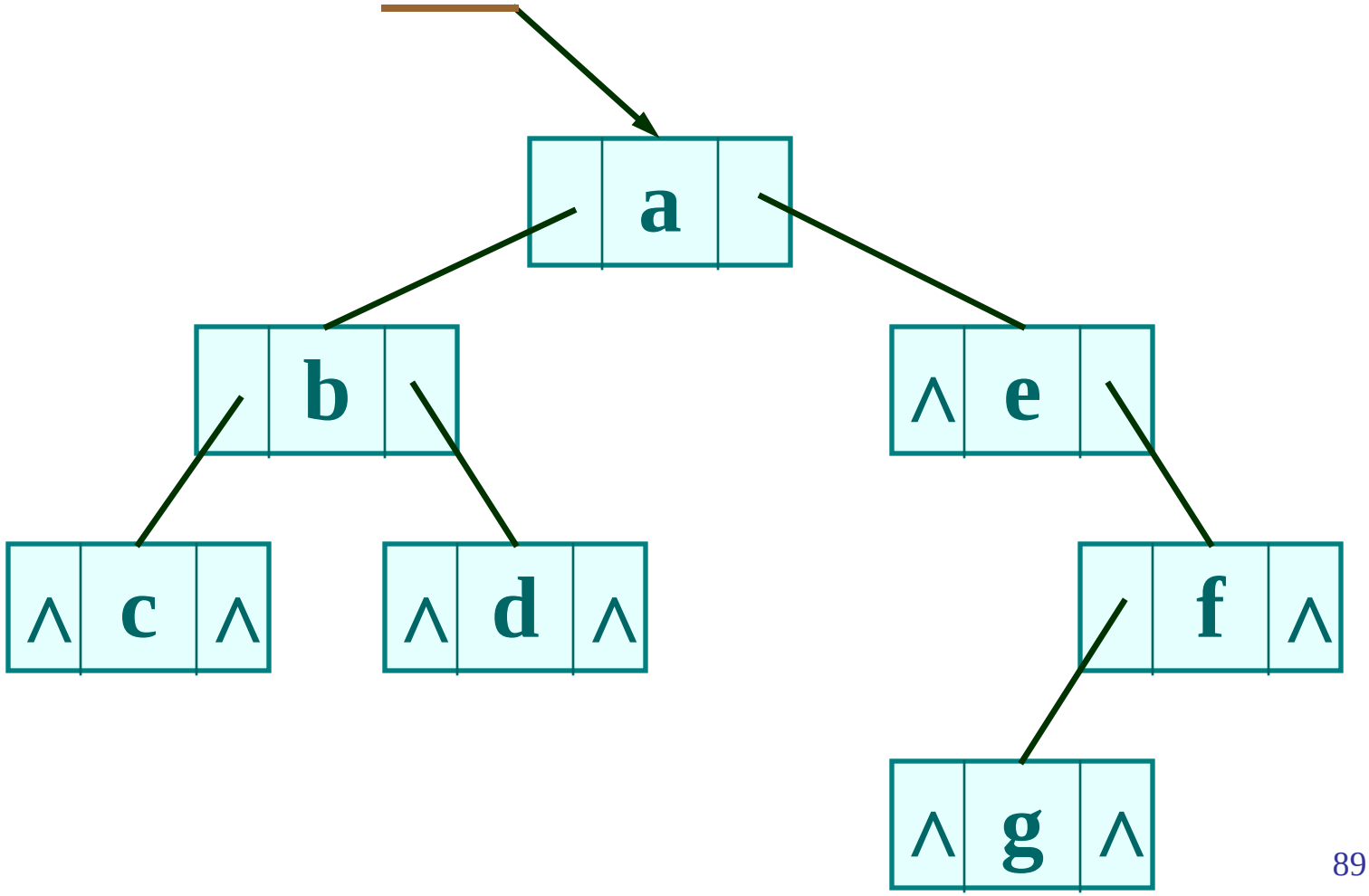
例如:

a b c d e f g

c b d a e g f

先序序列

中序序列



```
void CrtBT(BiTree& T, char pre[], char ino[],
                                    int ps, int is, int n ) {
// 已知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，本算法由此
两个序列构造二叉链表,ps为先序序列的开始位置，
is为中序序列的开始位置，n为序列长度
   if (n==0) T=NULL;
   else {
      k=Search(ino, pre[ps]); //查询先序序列中的
                    //第一个字符在中序序列中的位置
      if (k== -1)  T=NULL;
      else {      … …       }
   }
} // CrtBT
```

```
T=(BiTNode*)malloc(sizeof(BiTNode));
T->data = pre[ps];        //建立树根
if (k==is)  T->Lchild = NULL; //先序序列中第一个字
        符在中序序列中也是第一个字符，则表示没有左子树
else  CrtBT(T->Lchild, pre[], ino[],
                                ps+1, is, k-is );
if (k==is+n-1) T->Rchild = NULL; //先序序列中第一
    个字符在中序序列中是最后一个字符，则表示没有右子树
else  CrtBT(T->Rchild, pre[], ino[],
                    ps+1+(k-is), k+1, n-(k-is)-1 );
```

先序：ps↓ abcd efg (ps..ps+n-1)

中序：is↓ ↓k cbda egf (is..is+n-1)

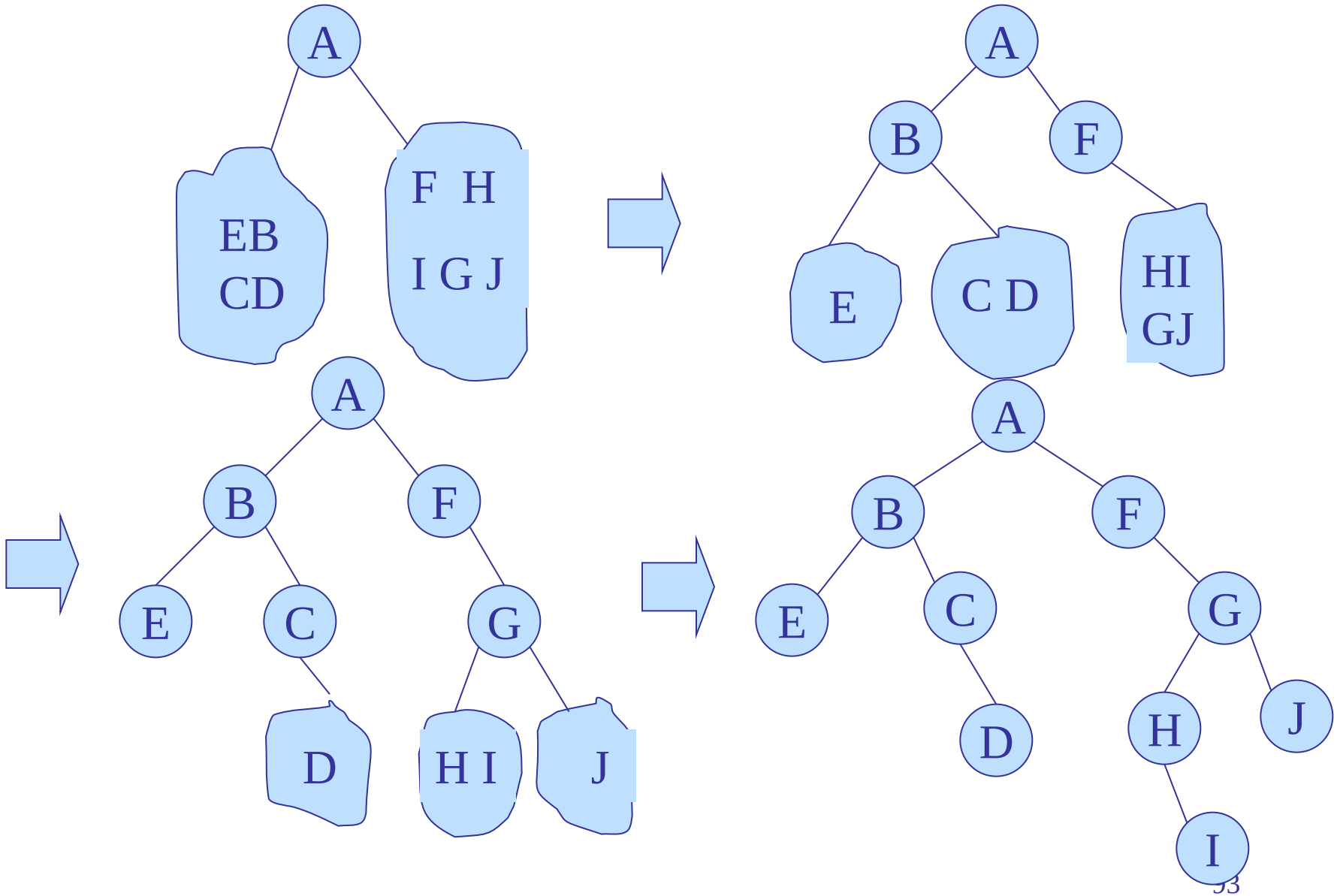
6、根据遍历序列构造二叉树

若已知二叉树的先序和中序序列，即可构造唯一的二叉树。

例：已知一棵二叉树的
先序遍历序列为ABECDFGHIJ，
中序遍历序列为EBCDAFHIGJ，
试构造这棵二叉树

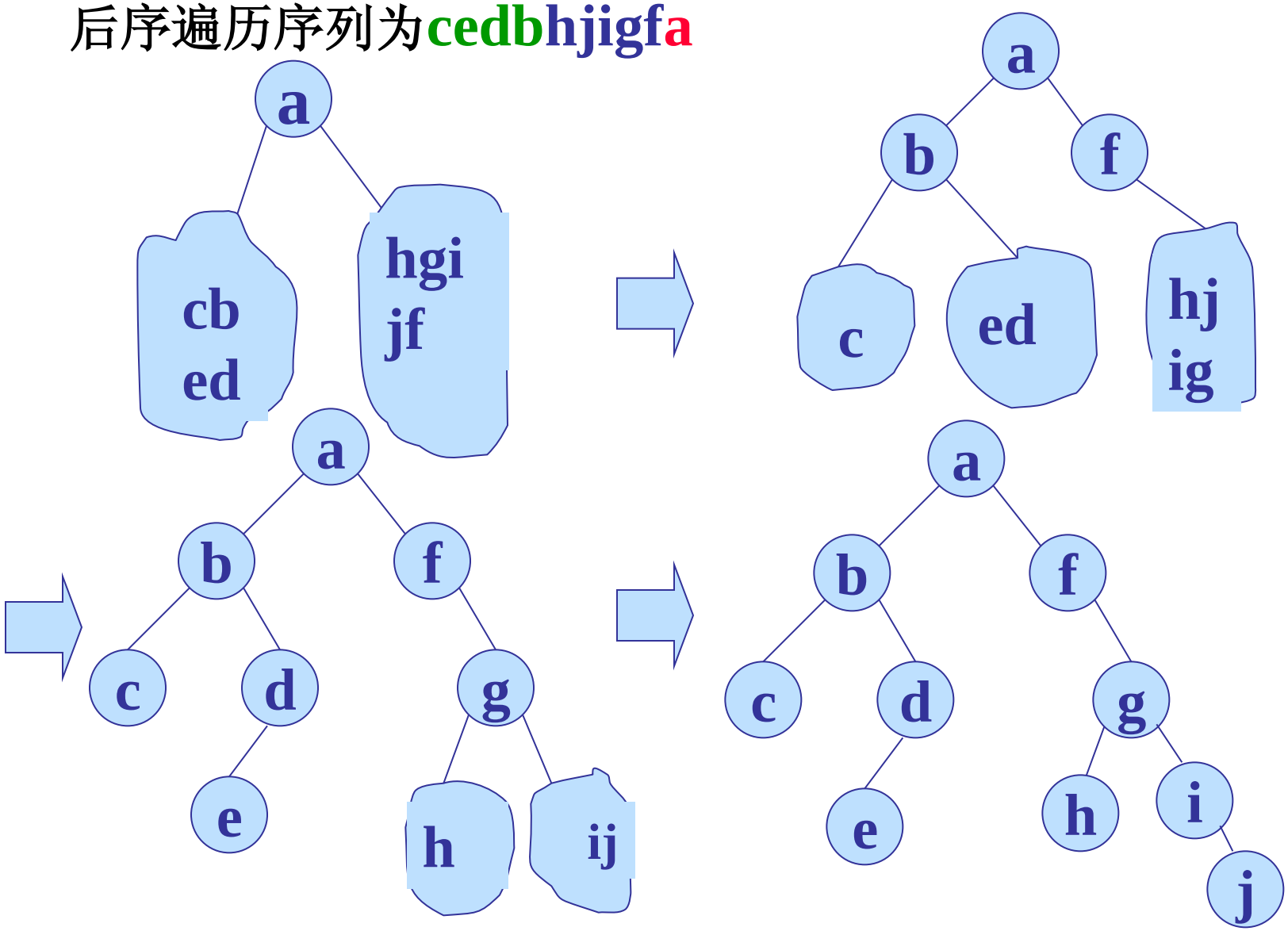
已知：　先序遍历序列为 ABECDFGHIJ

中序遍历序列为 EBCDAFHIGJ



已知:中序遍历序列为cbedafhgij

后序遍历序列为cedbhjigfa



6.1 树的类型定义

6.2 二叉树的类型定义

6.3 二叉树的存储结构

6.4 二叉树的遍历

6.5 线索二叉树

6.6 树和森林

6.7 树和森林的遍历

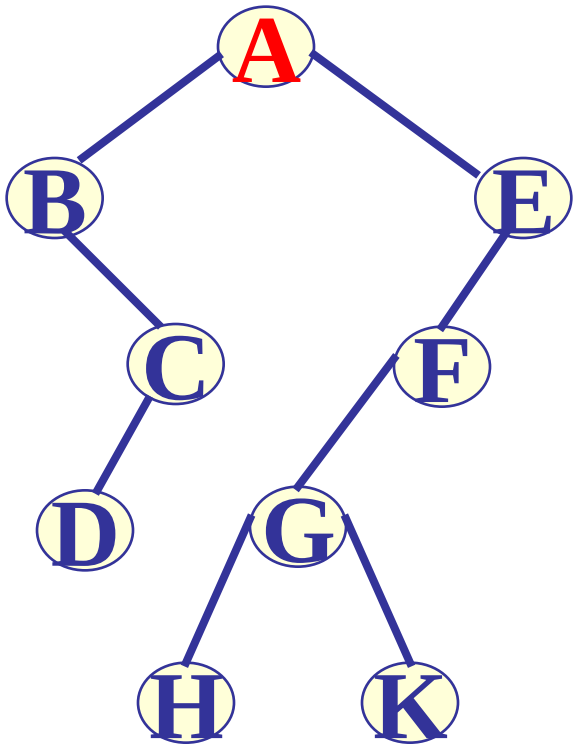
6.8 哈夫曼树与哈夫曼编码

6.5 线索二叉树

- 什么是线索二叉树?
- 线索链表的遍历算法
- 如何建立线索链表?

一、什么是线索二叉树?

当以二叉链表作为存储结构时，只能找到结点的左右孩子信息，而不能直接得到结点在任一序列的前驱和后继信息，这种信息只有在遍历的动态过程中才能得到。



能否把遍历过程中得到的结点的前驱和后继的信息保存下来呢?

方法一:

在每个结点上增加两个指针域fwd和bkwd，分别指示结点在任一次序遍历时得到的前驱和后继信息。这样做使得结构的存储密度大大降低。

方法二:

在有n个结点的二叉链表中必定存在n+1个空链域，利用这些空链域来存放结点的前驱和后继的信息。

在二叉链表的结点中**增加两个标志域**LTag和RTag

lchild	ltag	data	rtag	rchild

结点结构

其中：

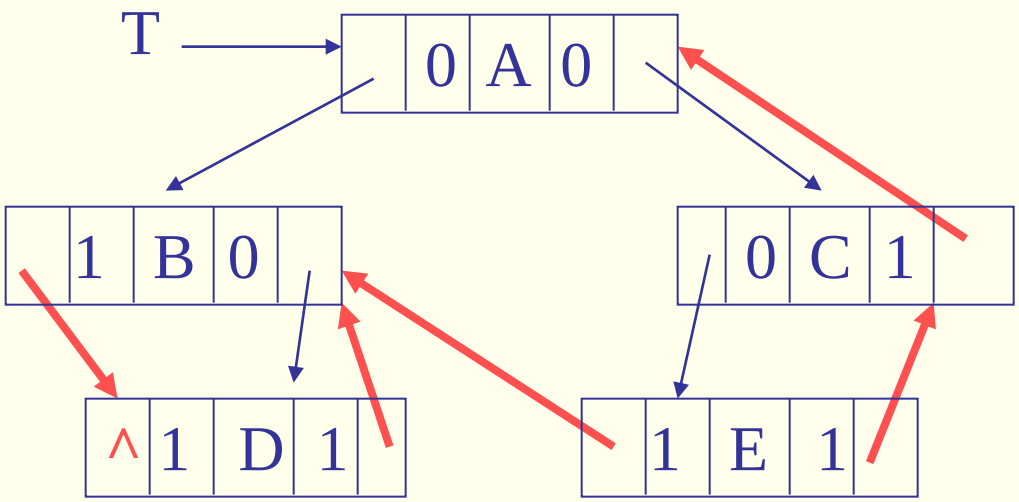
$$\text{ltag}=\begin{cases}0 & \text{lchild域指示结点的左孩子}\\ 1 & \text{lchild域指示结点的前趋}\end{cases}$$

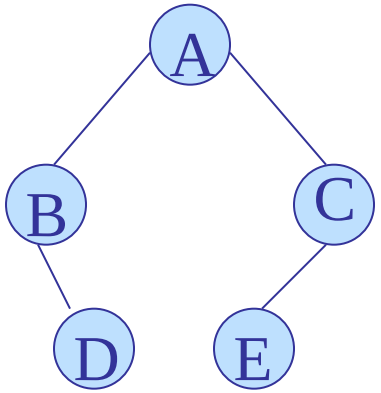
$$\text{rtag}=\begin{cases}0 & \text{rchild域指示结点的右孩子}\\ 1 & \text{rchild域指示结点的后继}\end{cases}$$

指向该线性序列中的“前驱”和“后继” 的指针，称作“线索”

如此定义的二叉树的存储结构称作“线索链表”。

例如:后序线索二叉树　　DBECA





这种带线索指针的二叉树，称作 “线索二叉树”

对二叉树以某种次序遍历使其变为线索二叉树的过程，叫做线索化。

线索链表的类型描述:

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

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

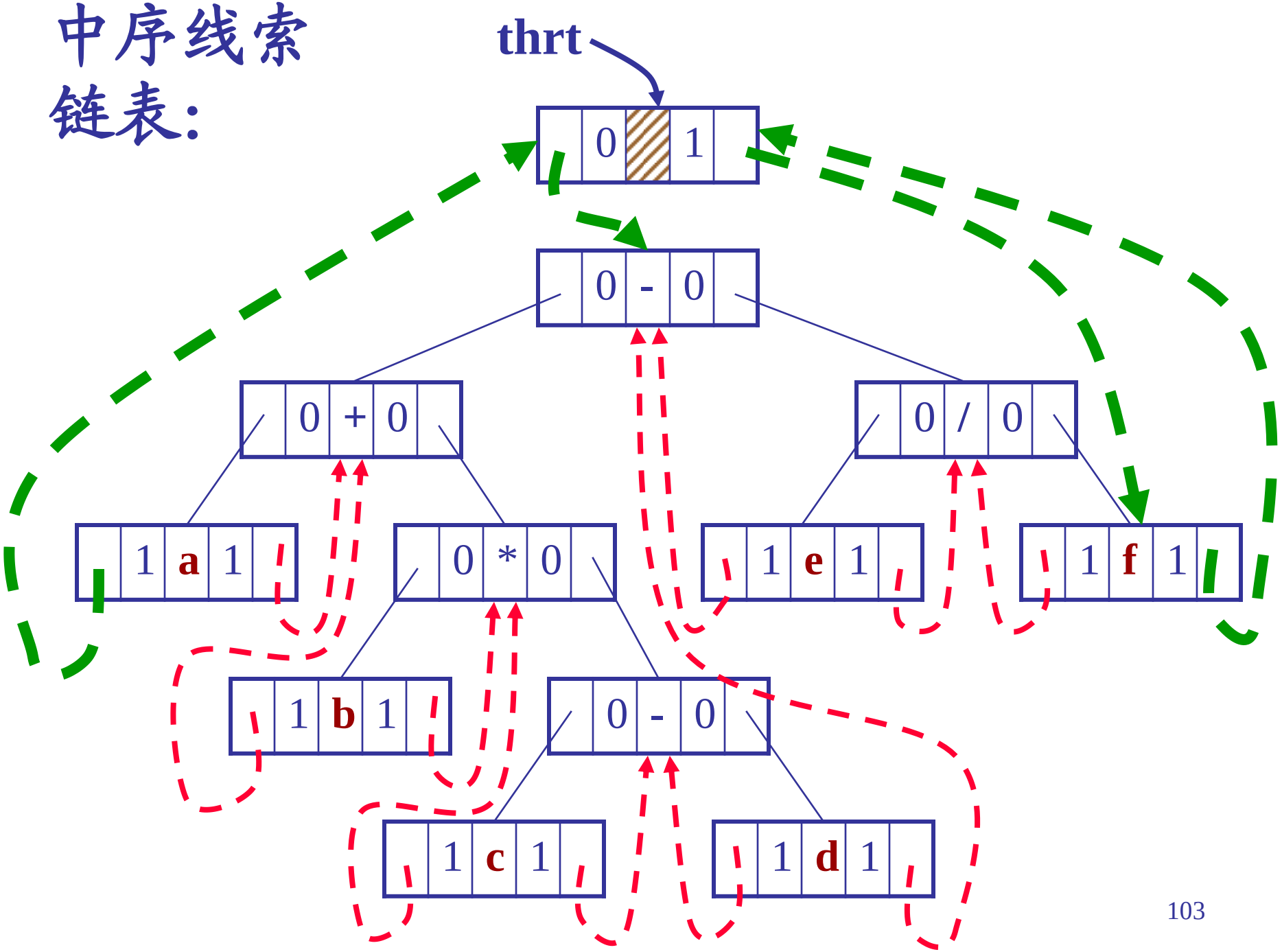
为方便起见，仿照线性表的存储结构，在二叉树的线索链表上也添加一个头结点，并令

头结点的**lchild域**的指针指向二叉树的根结点，

头结点的**rchild域**的指针指向中序遍历时访问的最后一个结点；

二叉树中序序列中的第一个结点的**lchild域**指针和最后一个结点**rchild域**的指针均指向头结点。

中序线索链表:



二、线索链表的遍历算法

以中序线索化链表的遍历算法

※ 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？

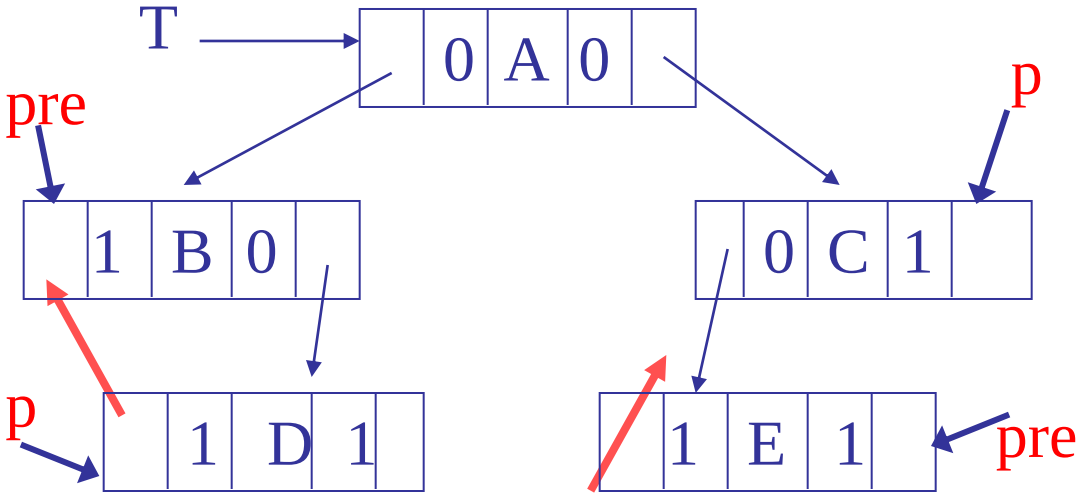
若无右子树，则为后继线索所指结点；

否则，为对其右子树进行中序遍历时访问的第一个结点。

```
void InOrderTraverse_Thr(BiThrTree T,
                         void (*Visit)(TElemType e)) {
          // T指向头结点，头结点的左链lchild指向根结点
          // 中序遍历二叉线索树T的非递归算法，
p = T->lchild;              // p指向根结点
while (p != T)              // 空树或遍历结束时，p==T
 { while (p->LTag==Link)    p=p->lchild; // 第一个结点
   if(!visit(p->data)) return error;      //访问左子树为空的结点
   while (p->RTag==Thread && p->rchild!=T)
      { p = p->rchild;  Visit(p->data); } // 访问后继结点
   p = p->rchild;                         // p进至其右子树根
  }
} // InOrderTraverse_Thr
```

三、如何建立线索链表?

- 线索化的过程即为在遍历过程中修改空指针的过程。
- 为了记下遍历过程中访问结点的先后关系，附设一个指针**pre**指向当前访问的、指针**p**所指结点的前驱。



中序：BDAEC

```
void InThreading(BiThrTree p)
{ if (p)                    // 对以p为根的非空二叉树进行中序线索化
  { InThreading(p->lchild);       // 左子树线索化
    if (!p->lchild)                // 左孩子空，建前驱线索
      { p->LTag = Thread;   p->lchild = pre; }
    if (!pre->rchild)              // 前驱的右孩子空，建后继线索
      { pre->RTag = Thread;   pre->rchild = p; }
    pre = p;                       // 保持 pre 指向 p 的前驱
    InThreading(p->rchild);        // 右子树线索化
  } // if
} // InThreading
```

```
Status InOrderThreading (BiThrTree &Thrt, BiThrTree T)
          // 构建中序线索链表Thrt指向头结点，T是二叉树的根
 { if (!(Thrt = (BiThrTree)malloc(
                                  sizeof( BiThrNode))))
      exit (OVERFLOW);          // 分配头结点
   Thrt->LTag = Link;  Thrt->RTag =Thread;
   Thrt->rchild = Thrt;              // 头结点的右指针回指

          …   …
   return OK;
} // InOrderThreading
```

```
if (!T)  Thrt->lchild = Thrt; //若二叉树空，左指针回指
else {
    Thrt->lchild = T;   pre = Thrt;
    InThreading(T);          // 中序遍历进行中序线索化
    pre->rchild = Thrt;      // 处理最后一个结点
    pre->RTag = Thread;
    Thrt->rchild = pre;
}
```

6.6 树和森林

一、树的存储结构

二、树、森林与二叉树的转换

一、树的存储结构

1、双亲表示法

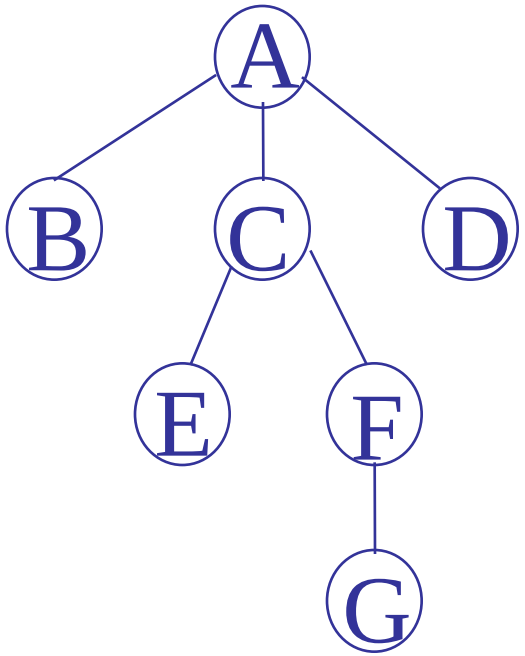
2、孩子表示法

3、孩子链表表示法

4、树的二叉链表(孩子-兄弟)表示法

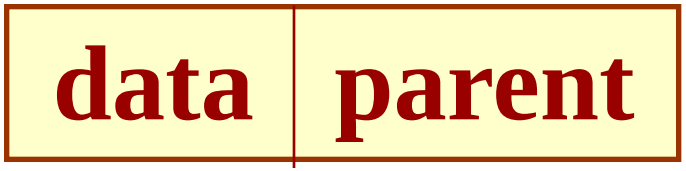
1、双亲表示法：

以一组连续空间存储树的结点，同时在每个结点中附设一个指示器指示其双亲结点的位置。



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

C语言的类型描述：




```
#define MAX_TREE_SIZE  100

typedef struct PTNode {  //结点结构
   Elem  data;
   int    parent;                  // 双亲位置域
} PTNode;

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

这种存储结构利用了每个结点（除根外）只有唯一双亲的性质。**PARENT(T,x)操作**可在常量时间内实现。反复调用**PARENT**操作，直到遇见无双亲(即：双亲的值为-1)的结点时，便找到了树的根，这就是**ROOT(x)操作**的执行过程。

但是，在这种表示法中，求结点的孩子时需要遍历整个结构。

2、孩子表示法：

由于树中每个结点可能有多棵子树，则在采用链式存储结构时可用多重链表，即每个结点有多个指针域，其中，每个指针指向一棵子树的根结点。此时，链表中的结点可有如下两种结点格式：

data	child1	child2	…	childd

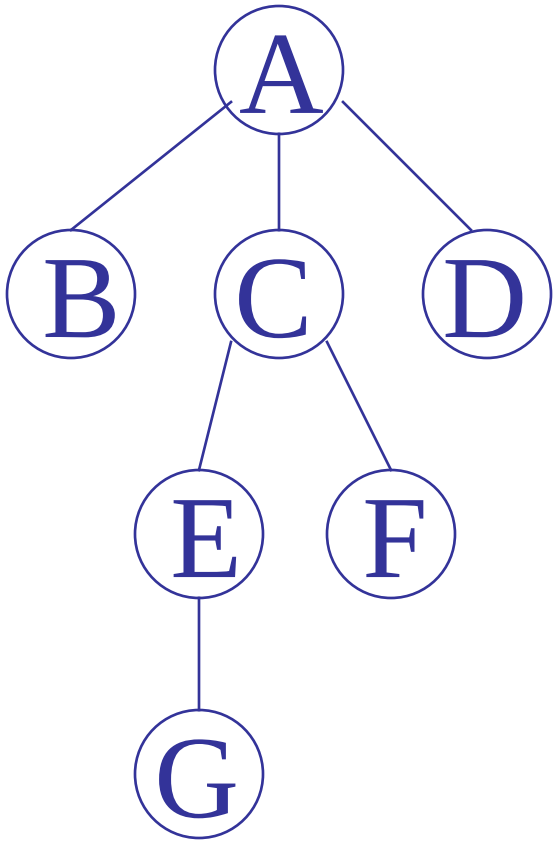
data	degree	child1	child2	…	Childd

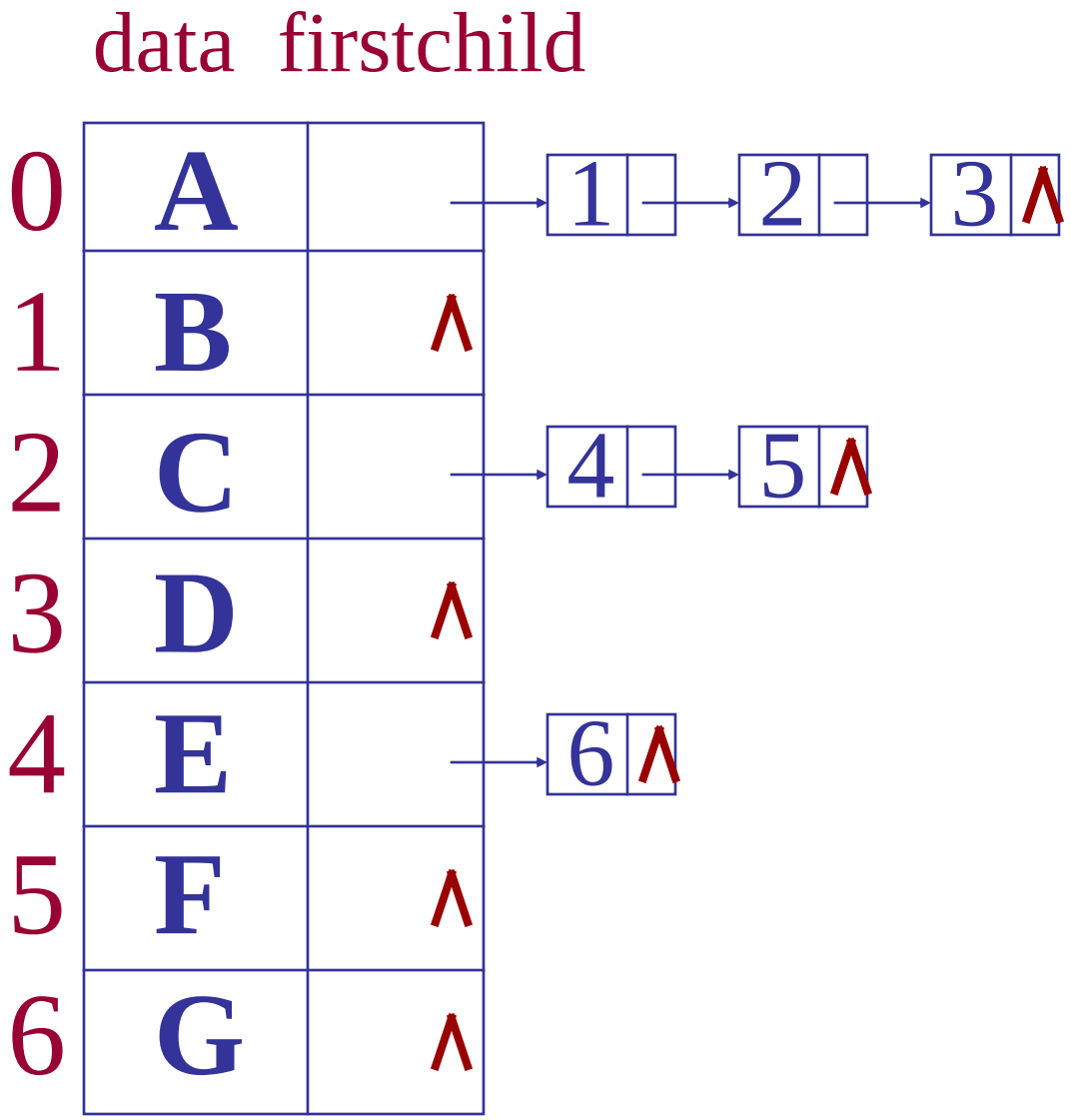
若采用第一种结点格式，则多重链表中的结点是同构的，其中d为树的度。由于树中很多结点的度小于d，所以链表中有很多空链域，空间较浪费。不难推出，在一棵有n个结点度为k的树中必有n(k-1)+1个空链域。

若采用第二种结点格式，则多重链表中的结点是不同构的，其中d为结点的度，degree域的值同d。此时，虽能节省存储空间，但操作不方便。

3、孩子链表表示法：

另一种办法是把每个结点的孩子结点排列起来，看成是一个线性表，且以单链表作存储结构，则n个结点有n个孩子链表（叶子的孩子链表为空表）。而n个头指针又组成一个线性表，为了便于查找，可采用顺序存储结构。


A
B
C
D
E
F
G


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

C语言的类型描述（三部分）：

孩子结点结构：


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

双亲结点结构：

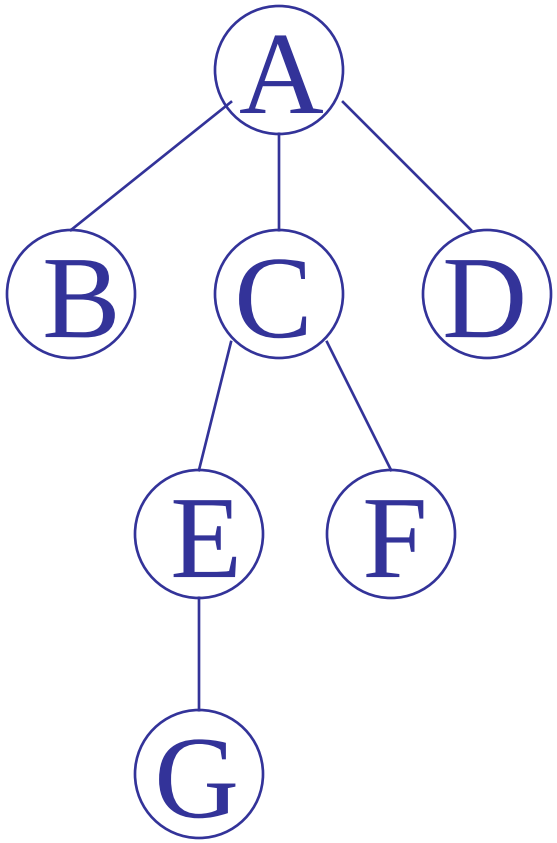

```
typedef struct {
    Elem    data;
    ChildPtr  firstchild;      //孩子链的头指针
} CTBox;
```

树结构:

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

与双亲表示法相反，孩子表示法便于那些涉及孩子的操作的实现，却不适用于**PARTEN(x)**操作。可把双亲表示法和孩子链表合在一起，组成带双亲的孩子链表。

带双亲的孩子链表



data parent firstchild

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

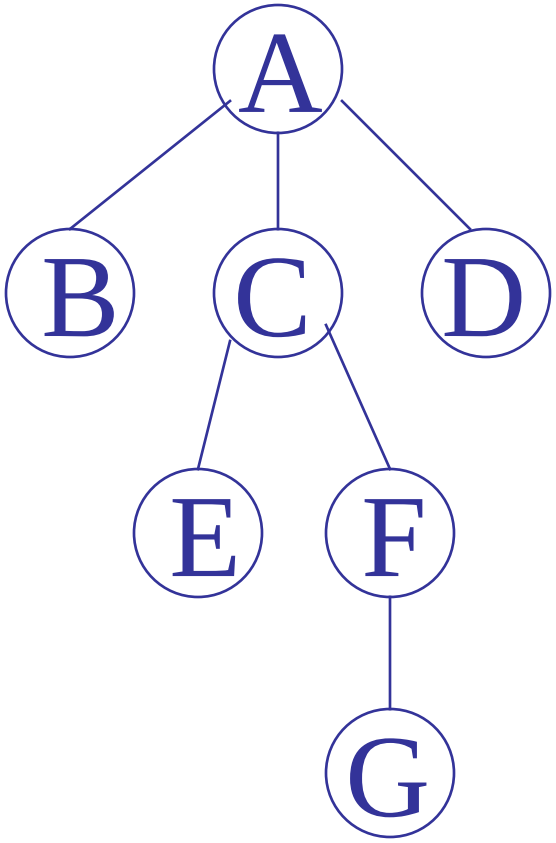
4、树的二叉链表(孩子-兄弟)存储表示法

又称二叉树表示法，或二叉链表表示法。即以二叉链表作树的存储结构。链表中结点的两个链域分别指向该结点的第一个孩子结点和下一个兄弟结点。

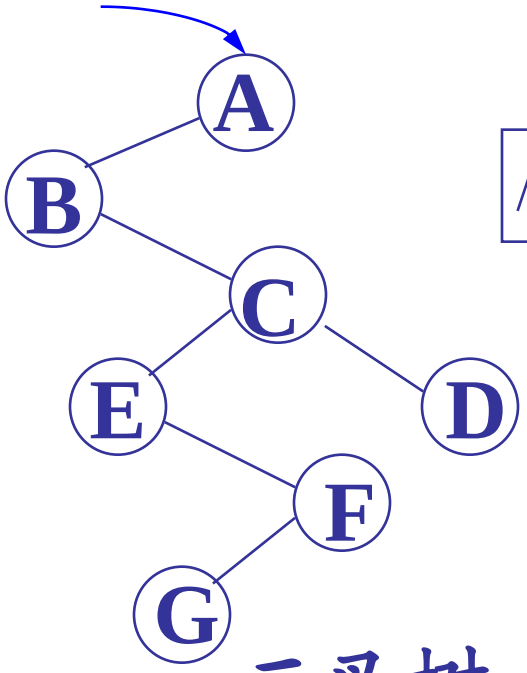
C语言的类型描述：

结点结构：


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

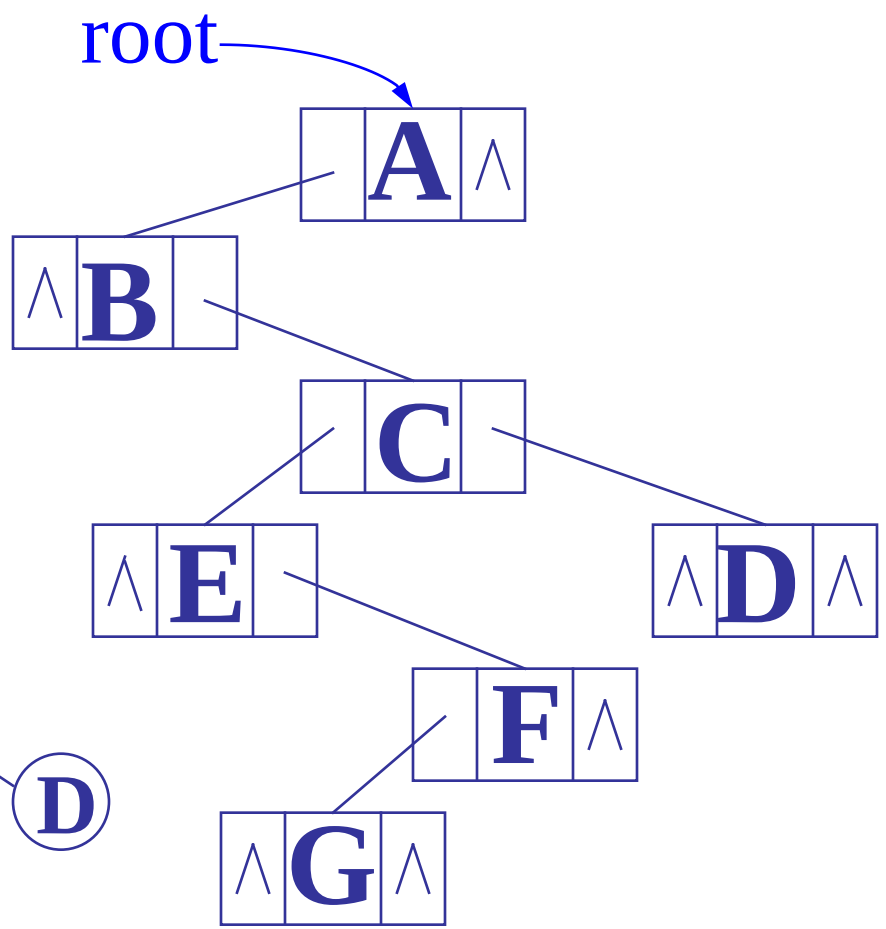


树



二叉树

二叉链表:



由上图可以得到如下结论:

给定一棵树，可以找到惟一的一棵二叉树与之对应。从树与二叉树的存储结构来看，树的二叉链表表示与对应的二叉树的二叉链表表示完全相同。只是解释不同,其左、右子树的概念改变为： 左是孩子，右是兄弟 。

任何一棵树所对应的二叉树，其右子树必为空。

利用这种存储结构便于实现各种树的操作(对应二叉树的操作来完成)。首先易于实现找结点孩子等的操作。

例如，若要访问结点x的第i 个孩子，则只要先从firstchild域找到第1个孩子，然后沿着孩子结点的nextsibling域连续走i-1步，便可找到x的第i个孩子。

二、森林与二叉树的转换

森林和二叉树的对应关系

设森林

$F = (T_1, T_2, \ldots, T_n)$;

$T_1 = (root, t_{11}, t_{12}, \ldots, t_{1m})$;

二叉树

$B = (LBT, Node(root), RBT)$;

由森林F (T1, T2, …, Tn)转换成二叉树B

的转换规则为:

若 **F = Φ**，则 **B = Φ**;

否则，

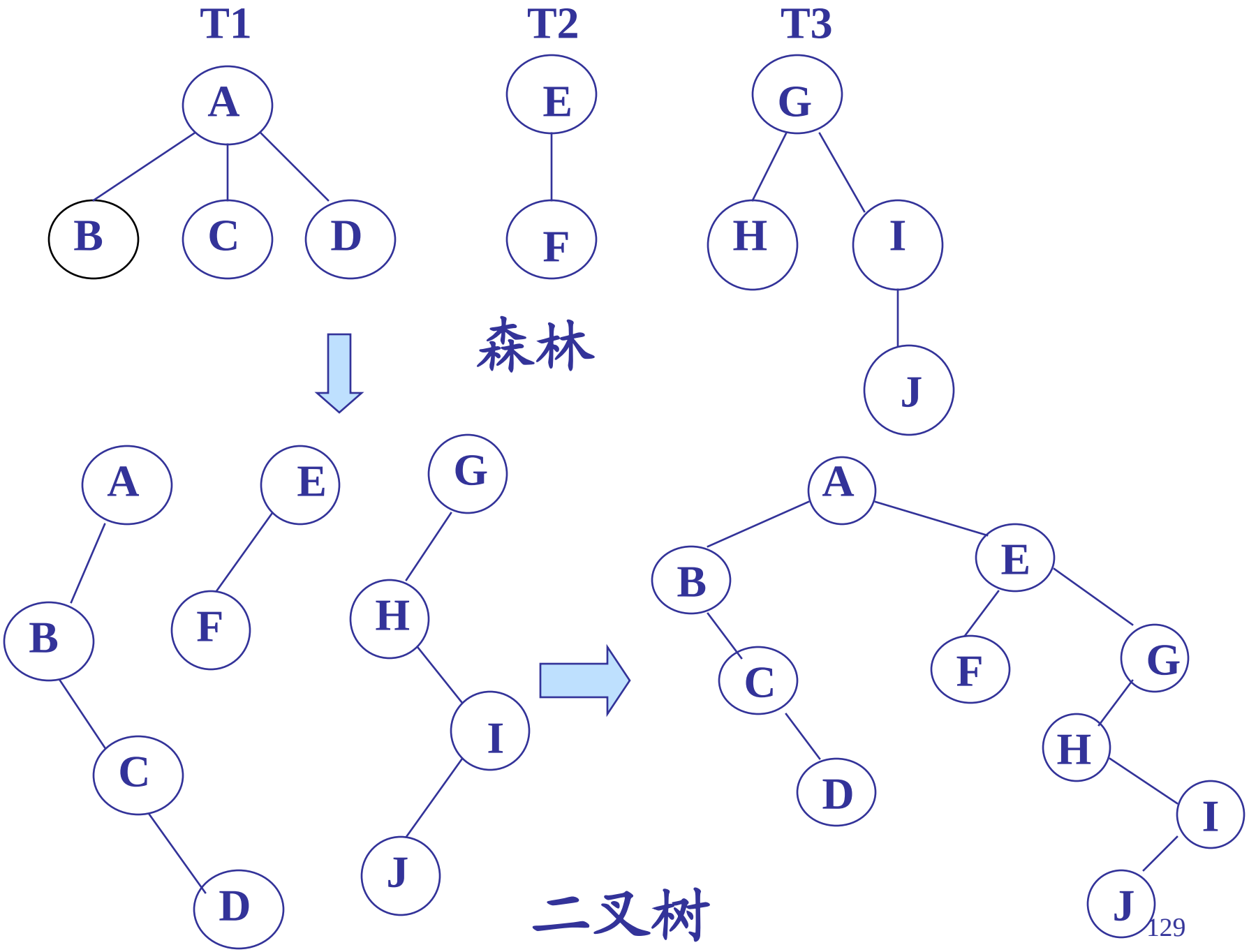
由 **ROOT(T_1)** 对应得到 **B(root)**;

由 **(t_{11}, t_{12}, …, t_{1m})** 对应转换而成的二叉树

得到 整个二叉树的**LBT**;

由 **(T_2, T_3,…, T_n)** 对应转换而成的二叉树

得到整个二叉树的**RBT**。


T1
T2
T3
A
B
C
D
E
F
G
H
I
J
森林
二叉树

由二叉树转换为森林的转换规则为：

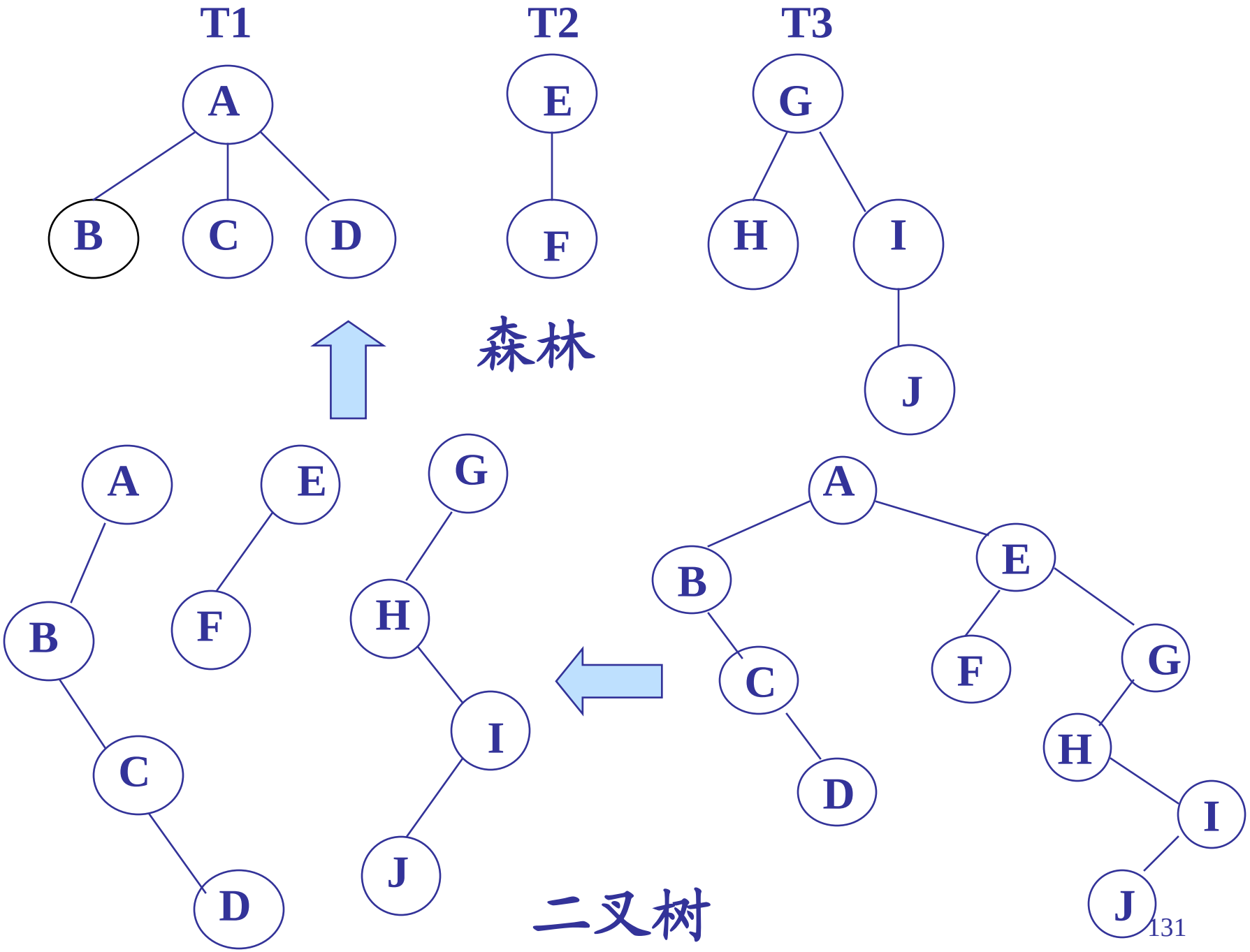
若 **B = Φ**，　则 **F = Φ**;

否则，

由 **B(root)** 对应得到 **ROOT(T_1)**;

由**LBT**对应转换得到 **(t_{11}, t_{12}, …,　t_{1m})**;

由**RBT**对应**转换**得到 **(T_2, T_3, …, T_n)**。

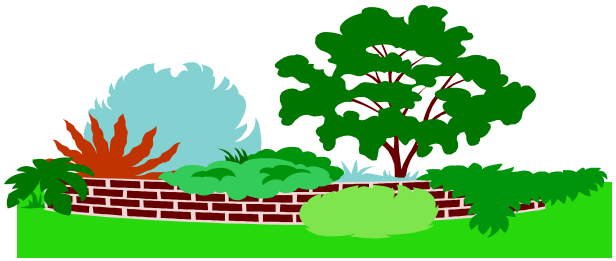

T1
T2
T3
A
B
C
D
E
F
G
H
I
J
森林
二叉树

6.7 树和森林的遍历

一、树的遍历

二、森林的遍历

三、树的遍历的应用



一、树的遍历　有三条搜索路径:

先根(次序)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。(相当于对应二叉树的先序遍历)

后根(次序)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。(相当于对应二叉树的中序遍历)

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。

先根遍历时顶点的访问次序:

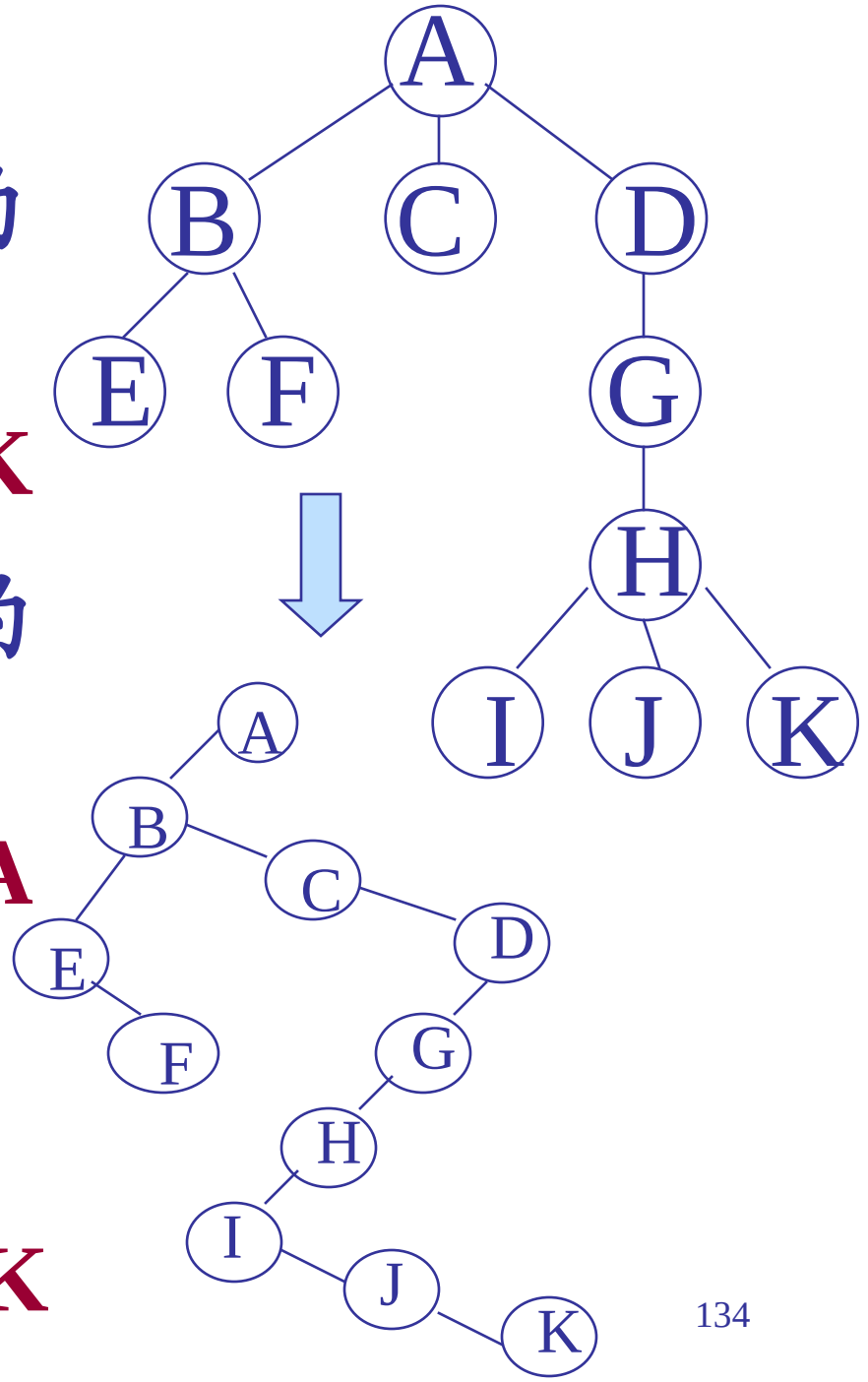
A B E F C D G H I J K

后根遍历时顶点的访问次序:

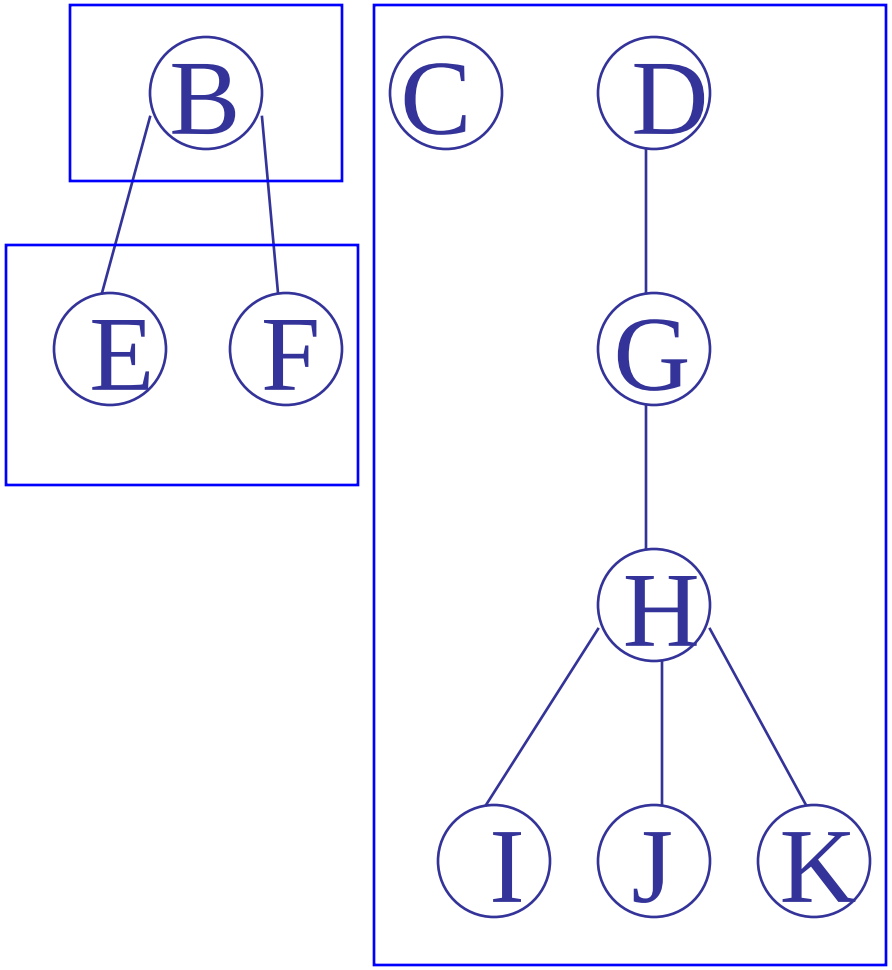
E F B C I J K H G D A

层次遍历时顶点的访问次序:

A B C D E F G H I J K



二、森林的遍历



森林由三部分构成:

1. 森林中第一棵树的根结点;

2. 森林中第一棵树的子树森林;

3. 森林中其它树构成的森林。

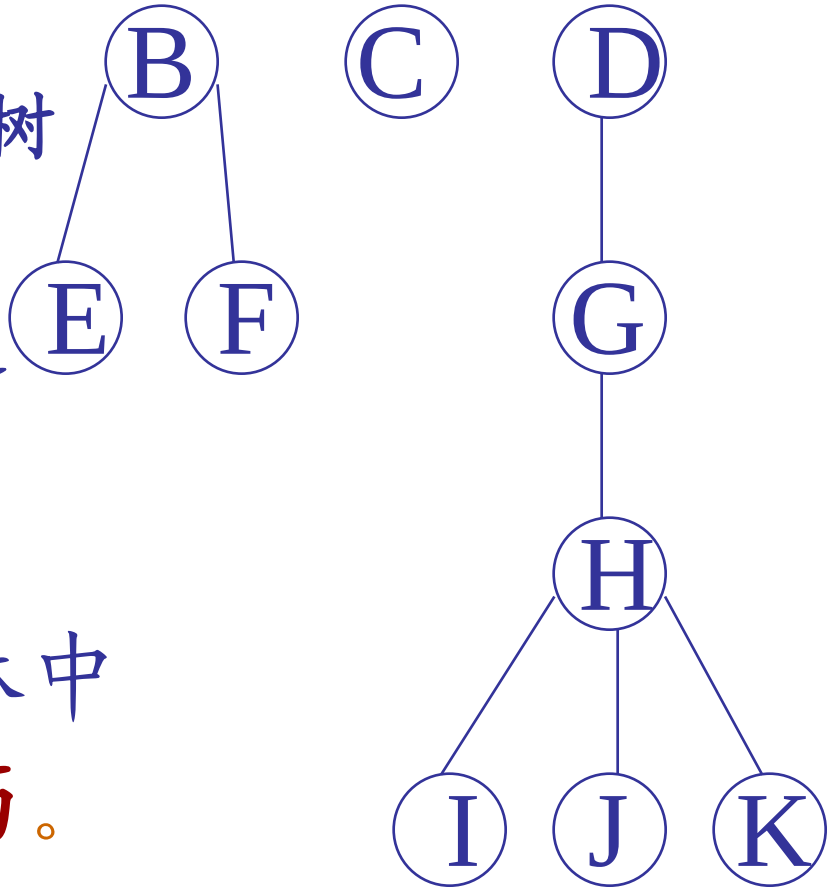
1. 先序遍历

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(除第一棵树之外)其余树构成的森林。



即：依次从左至右对森林中的每一棵树进行先根遍历。

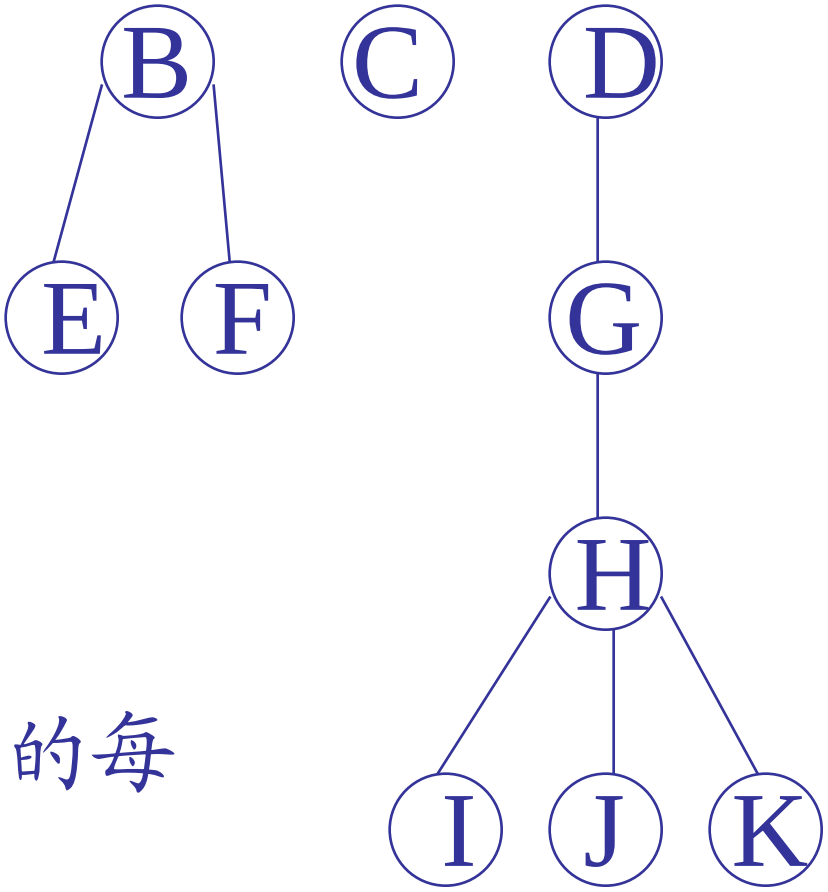
2．中序遍历

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：**依次从左至右**对森林中的每一棵**树**进行**后根遍历**。

树、森林的遍历和二叉树遍历的对应关系 ？

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

三、树的遍历的应用

设树的存储结构为孩子兄弟链表(二叉链表)

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

1、求树的深度

2、输出树中所有从根到叶子的路径

3、建树的存储结构

自学

1、求树的深度的算法:

```
int TreeDepth(CSTree T) {
  if(!T)  return 0;
  else {
     h1 = TreeDepth( T->firstchild );
     h2 = TreeDepth( T->nextsibling);
    return(max(h1+1, h2));
  }
} // TreeDepth
```

6.8 哈夫曼树与哈夫曼编码

- 最优树的定义
- 如何构造最优树
- 前缀编码

一、最优树的定义

- **路径:**

  从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径。

- **结点的路径长度:**

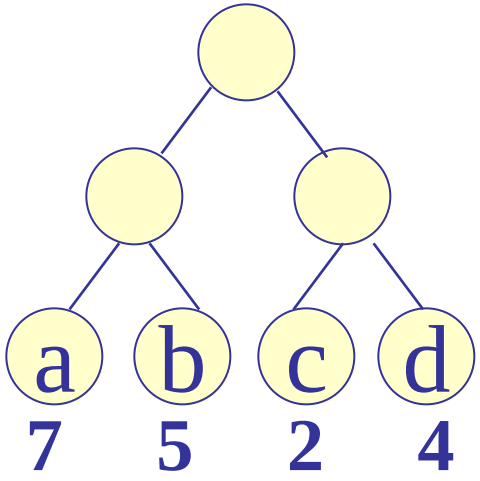
  从根结点到该结点的路径上分支的数目。

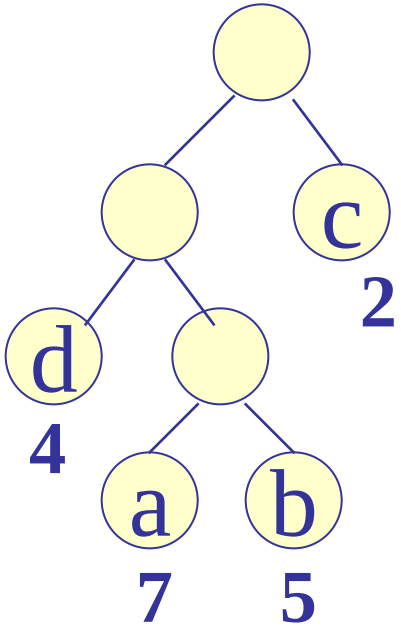
- **树的路径长度:**

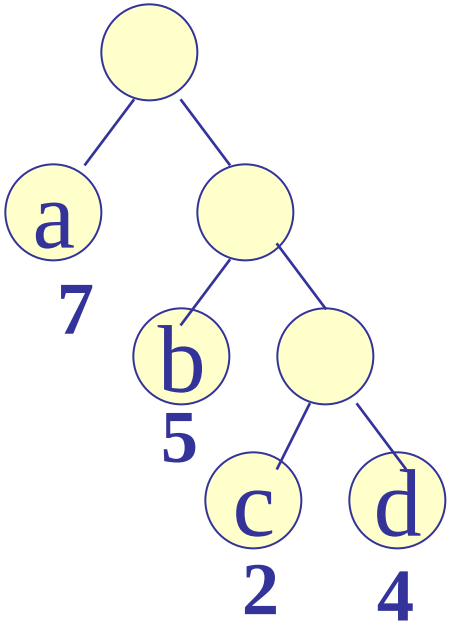
  树中每个结点的路径长度之和。

树的带权路径长度：$WPL(T) = \Sigma w_k l_k$

树中所有叶子结点的带权路径长度之和。







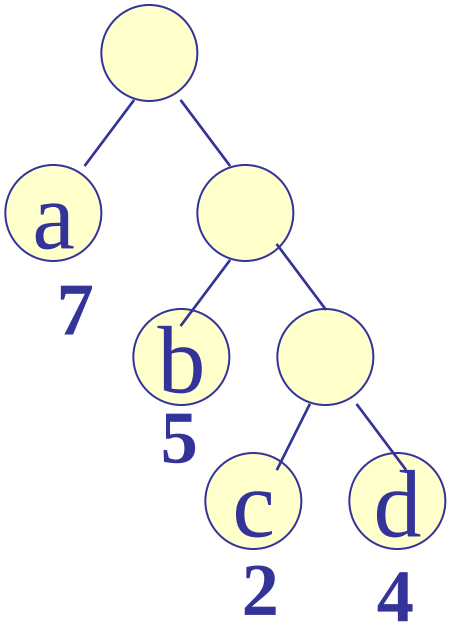
(1) **WPL=7*2+5*2+2*2+4*2=36**

(2) **WPL= 7*3+5*3+2*1+4*2=46**

(3) **WPL= 7*1+5*2+2*3+4*3=35**

最优树:

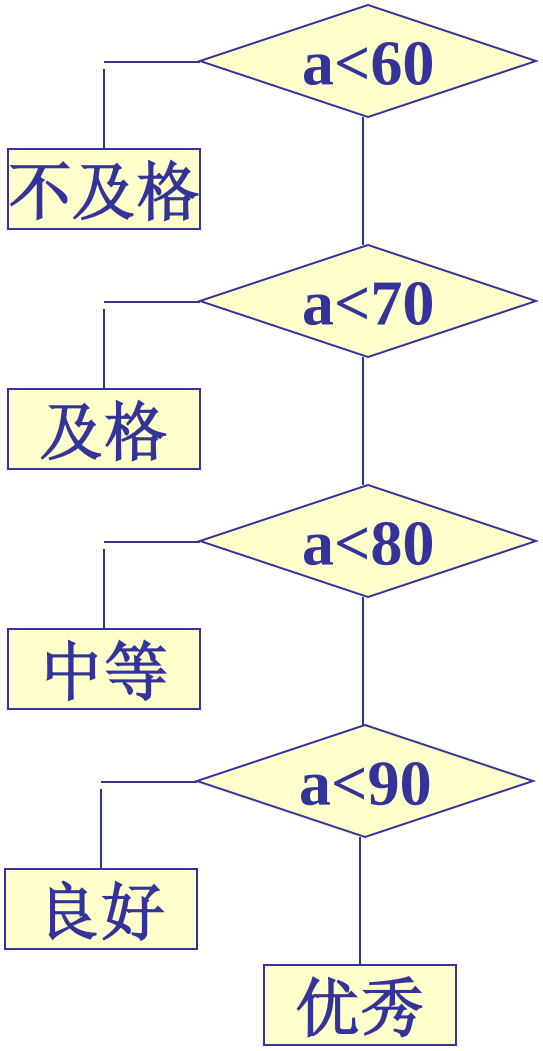
在所有含 n 个叶子结点、并带相同权值的 m 叉树中，必存在一棵其**带权路径长度取最小值**的树，称为“**最优树**”或“**哈夫曼树**”。



WPL= 7*1+5*2+2*3+4*3=35

在解某判定问题时，利用哈夫曼树可以得到最佳判定算法。

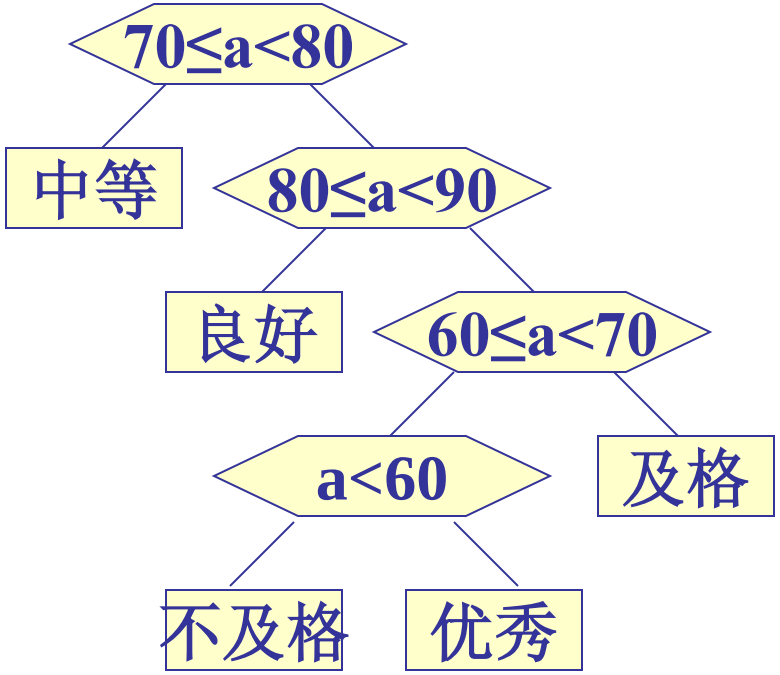
例：编程将百分制转换成五级分制。判定过程所用判定树：

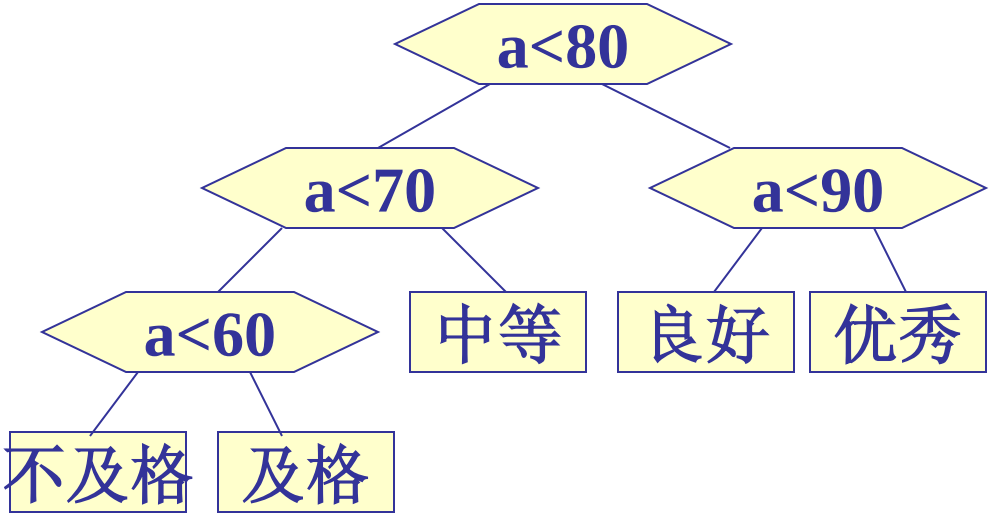


如果上述程序需反复使用，且每次的输入量很大，则应考虑上述程序的质量问题，即其操作所需的时间。因为在实际生活中，学生的成绩在五个等级上的分布是不均匀的，假设其分布规律如下表所示，则80%以上的数据需进行三次或三次以上的比较才能得出结果。

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

以学生成绩在五个等级上的分布概率为权，构造具有五个叶子结点的哈夫曼树，得图示判定过程，可使大部分数据经过较少的比较次数得出结果。





将每个判定框的两次比较分开，得到左图的判定树。

二、如何构造最优树

哈夫曼算法

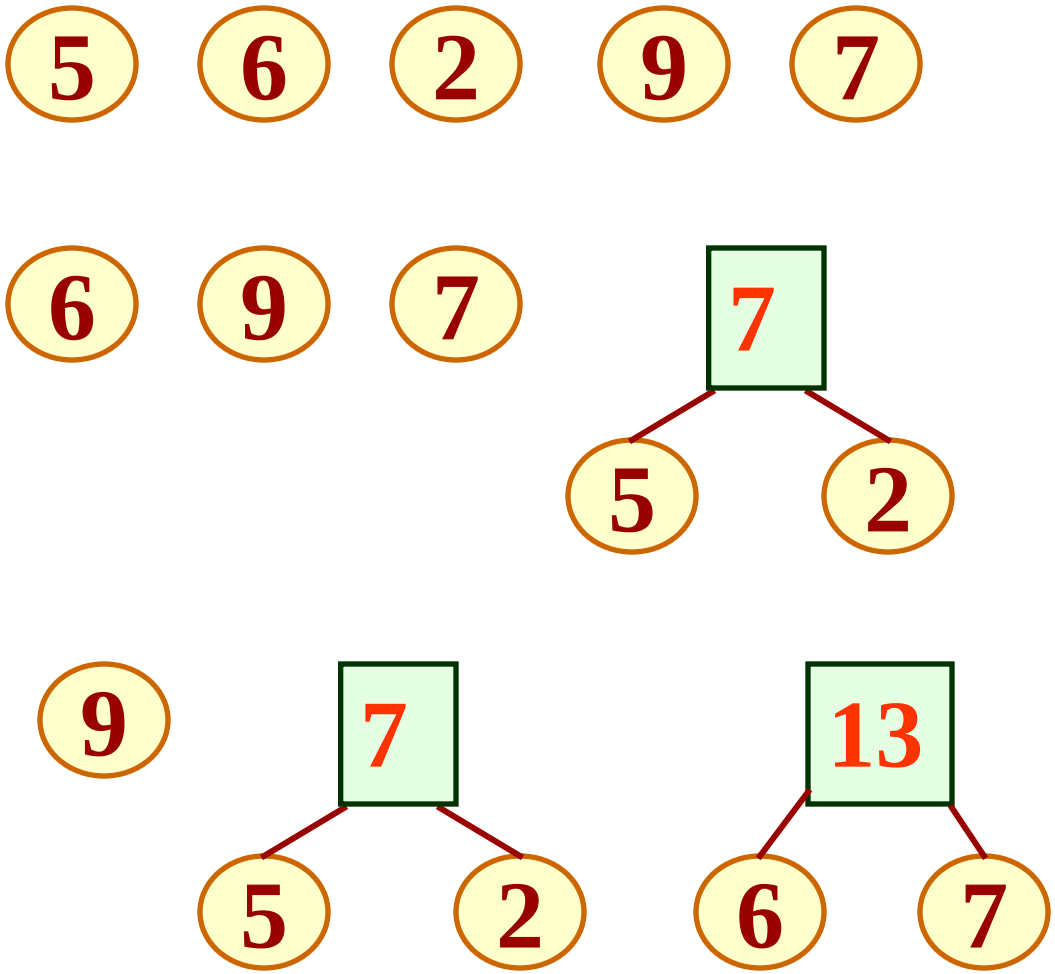
(1) 根据给定的n个权值{w1,w2,…,wn}构成n棵二叉树的集合F={T1,T2,…,Tn}，其中每棵二叉树Ti中只有一个带权为Wi的根结点，其左右子树均空;

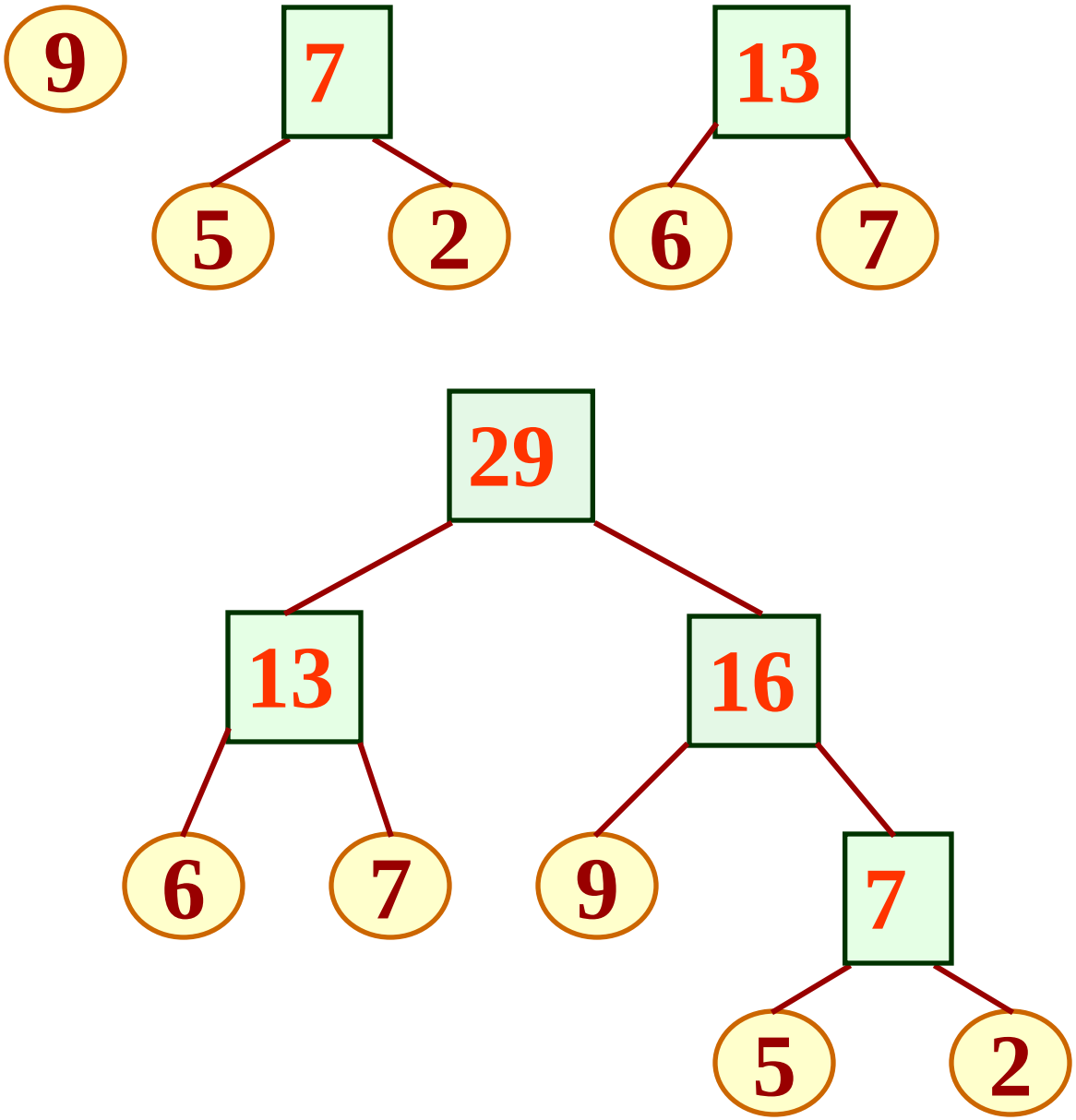
(2) 在F中选取两棵根结点的权值最小的树作为左、右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和;

(3) 在F中删除这两棵树，同时将新得到的二叉树加入F中;

(4) 重复(2)和 (3)，直到F只含一棵树为止。这棵树便是哈夫曼树。

例如: 已知权值 W={ 5, 6, 2, 9, 7 }




9
7
5
2
13
6
7
29
13
16
6
7
9
7
5
2

三、前缀编码

电文： **ABACCDA**

A:00
B:01
C:10
D:11

00 01 00 10 10 11 00
A B A C C D A

A:0
B:00
C:1
D:01

0 0 0 0 1 1 0 1 0
AAAA
B B
A B A

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。

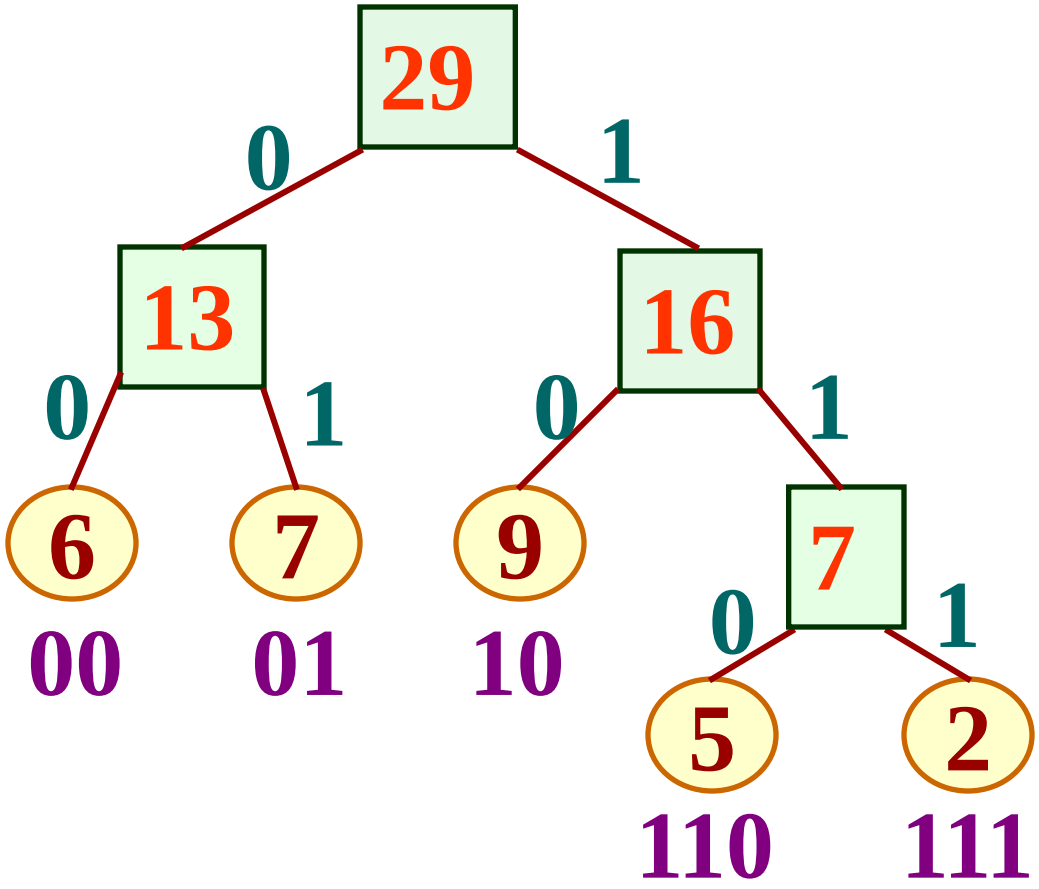
如何构造前缀编码?

利用赫夫曼树可以构造一种不等长的二进制编码，并且构造所得的赫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。

如何构造哈夫曼编码?

构造方法:

设计电文总长最短的二进制前缀编码即为以n种字符出现的频率作权，设计一棵哈夫曼树的问题，约定左分支表示字符0，右分支表示字符1，则从根结点到叶结点的路径上，分支字符组成的字符串作为该叶子结点字符的编码，由此得到的二进制前缀编码称为哈夫曼编码。


29
0
1
13
16
0
1
0
1
6
7
9
7
00
01
10
0
1
5
2
110
111

第六章 树和二叉树小结

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**递归和非递归算法**，**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**。

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。建立存储结构是进行其它操作的前提，因此应**掌握** 一至 两种**建立**二叉树和树的**存储结构的方法**。

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。

作业：6.26 6.27 6.28 6.29 6.65 6.68

6.26 假设用于通信的电文仅由8个字母组成，字母在电文中出现的频率分别为0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10。试为这8个字母设计哈夫曼编码。使用0—7的二进制表示形式是另一种编码方案，对于上述实例，比较两种方案的优缺点。

6.27 假设一颗二叉树的先序序列为**EBADCFHGIKJ**和中序序列为**ABCDEFGHIJK**，请画出该树。

6.28 假设一颗二叉树的中序序列为**DCBGEAHFIJK**和后序序列为**DCEGBFHKJIA**，请画出该树。

6.29 假设一颗二叉树的层序序列为**ABCDEFGHIJ**和中序序列为**DBGEHJACIF**，请画出该树

6.65 已知一颗二叉树的前序序列和中序序列分别存于两个一维数组中，试编写算法建立该二叉树的二叉链表。

6.68 已知一颗树的由根至叶子结点按层次输入的结点序列及每个结点的度（每层中自左至右输入），试写出构造此树的孩子—兄弟链表的算法