第7章 树和二叉树

树和森林的概念

- 树的定义：

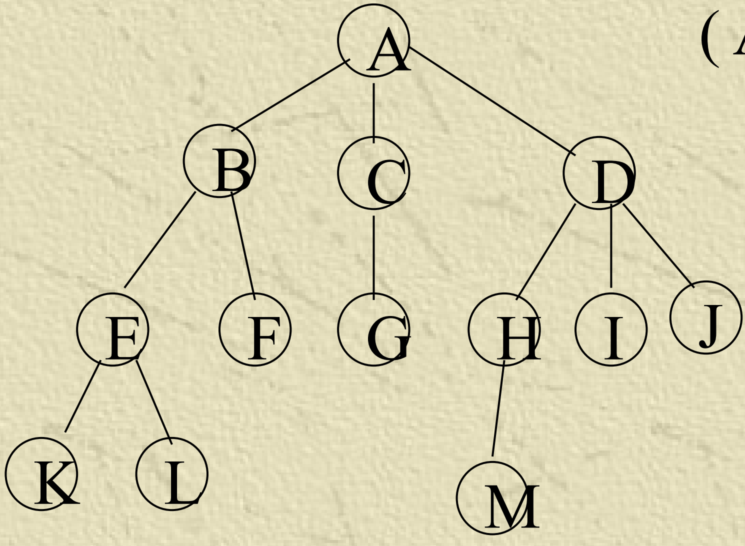
是n（n≥0)个结点的有限集合T，对于任意一棵非空树，它满足：

（1）有且仅有一个特定的称为根的结点；

（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，….，T_m，其中每个集合本身又是一棵树，称为根的子树。

显然：上述树的定义是一个递归定义。

树的表示方法:

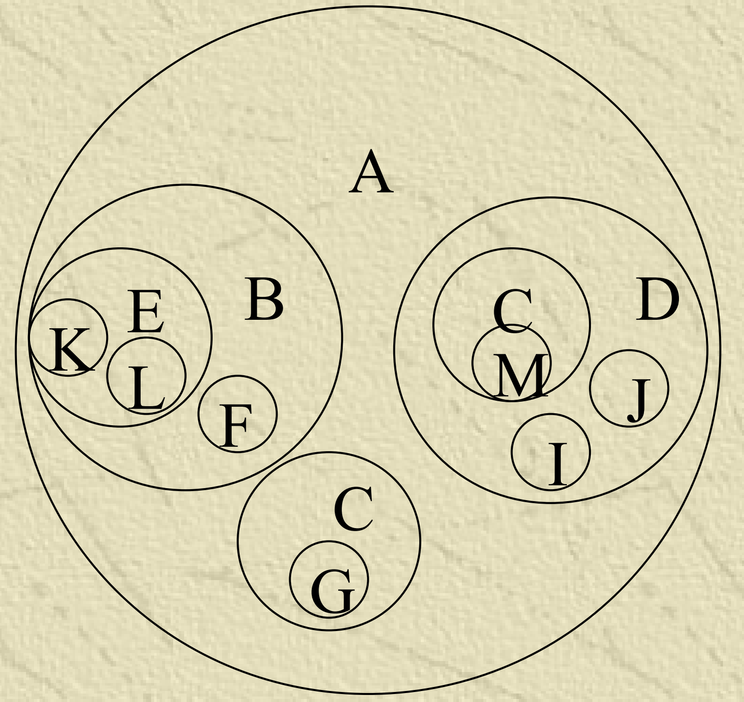


树形表示

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

广义表

凹入表



文氏图

树的基本术语：

• 结点(node)	
• 结点的度(degree)	结点的子树个数
• 分支(branch)结点	度不为0的结点
• 叶(leaf)结点	度为0的结点
• 孩子(child)结点	某结点子树的根结点
• 双亲(parent)结点	某个结点是其子树之根的双亲
• 兄弟(sibling)结点	具有同一双亲的所有结点
• 祖先(ancestor)结点	若树中结点k到k_s存在一条路径，则称k是k_s的祖先
• 子孙(descendant)结点	若树中结点k到k_s存在一条路径，则称k_s是k的子孙
• 结点所处层次(level)	根结点的层数为1，其余结点的层数为双亲结点的层数加1
• 树的高度(depth)	树中结点的最大层数
• 树的度(degree)	树中结点度数的最大值

- 有序树　　　子树的次序不能互换
- 无序树　　　子树的次序可以互换
- 森林　　　　互不相交的树的集合

树的基本操作

1、初始化 (InitTree)
2、建立树 (CreateTree)
3、求指定结点的双亲结点 (Parent)
4、求指定结点的左孩子结点 (LeftChild)
5、求指定结点的右兄弟结点 (RightSibling)
6、将一棵树插入到另一树的指定结点下作为它的子树 (InsertChild)
7、删除指定结点的某一子树 (DeleteChild)
8、树的遍历 (TraverseTree)

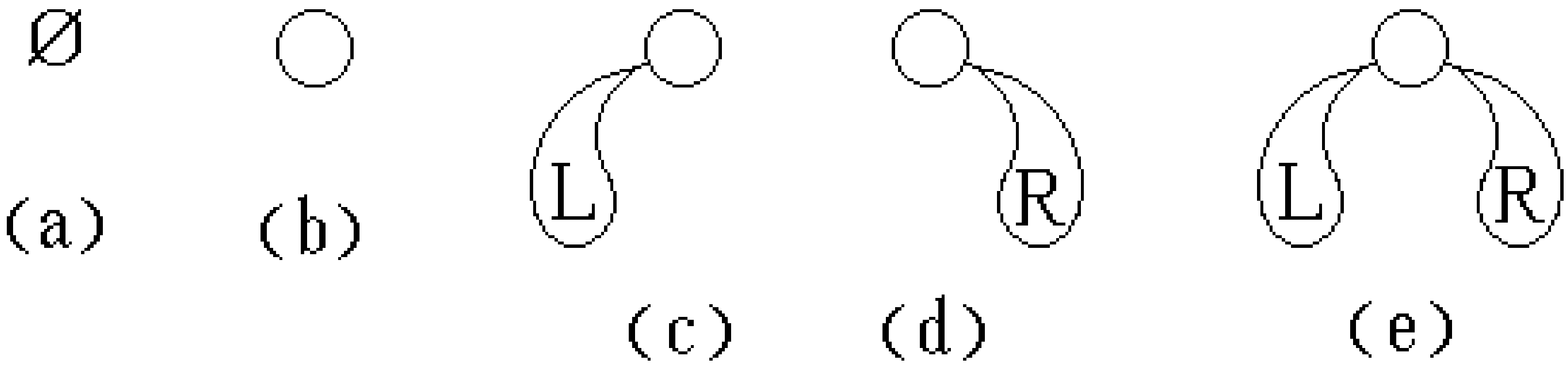
7.2 二叉树的概念与性质

二叉树的定义

一棵二叉树是n ($n \geq 0$)个结点的一个有限集合，

(1) 该集合或者为空，

(2) 或者是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。



(a) (b) (c) (d) (e)

二叉树的五种不同形态

问题：

试分别画出具有3个结点的树和3个结点的二叉树的所有不同形态。

二叉树的性质

性质1 若二叉树的层次从1开始, 则在二叉树的第 i 层最多有 2^{i-1} 个结点。($i \geq 1$)

证明:

$i = 1$ 时, 有$2^{i-1} = 2^0 = 1$, *成立*

假定: $i = k$ 时性质成立;

当 $i = k+1$ 时, 第k+1层的结点至多是第k层结点的两倍, 即总的结点个数至多为$2 \times 2^{k-1} = 2^k$

故命题成立

性质2 **高度为k的二叉树最多有 2^k-1个结点。($k \geq 1$)**

证明：仅当每一层都含有最大结点数时，二叉树的结点数最多，利用性质1可得二叉树的结点数至多为：

$$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{k-1} = 2^k-1$$

性质3 对任何一棵二叉树，如果其叶结点个数为n_0，度为2的非叶结点个数为 n_2，则有

$$n_0=n_2+1$$

证明：

1、结点总数为度为0的结点加上度为1的结点再加上度 为2的结点：

$$n = n_0 + n_1 + n_2$$

2、另一方面，二叉树中一度结点有一个孩子，二度结点有二个孩子，根结点不是任何结点的孩子，因此，结点总数为：

$$n = n_1 + 2n_2 + 1$$

3、两式相减，得到：

$$n_0 = n_2 + 1$$

定义1 满二叉树(Full Binary Tree)

一棵深度为k 且有2^k-1个结点的二叉树。

满二叉树的特点：每一层都取最大结点数

定义2 完全二叉树(Complete Binary Tree)

高度为k，有n个结点的二叉树是一棵完全二叉树，当且仅当其每个结点都与高度为k的满二叉树中层次编号1--n相对应。

完全二叉树的特点---

（1）除最后一层外，每一层都取最大结点数，最后一层结点都有集中在该层最左边的若干位置。

（2）叶子结点只可能在层次最大的两层出现。

（3）对任一结点，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为L或L+1。

性质4 **具有*n*个结点的完全二叉树的高度为 $\lfloor \log_2 n \rfloor + 1$。**

证明：

设深度为k，根据二叉树性质二知：

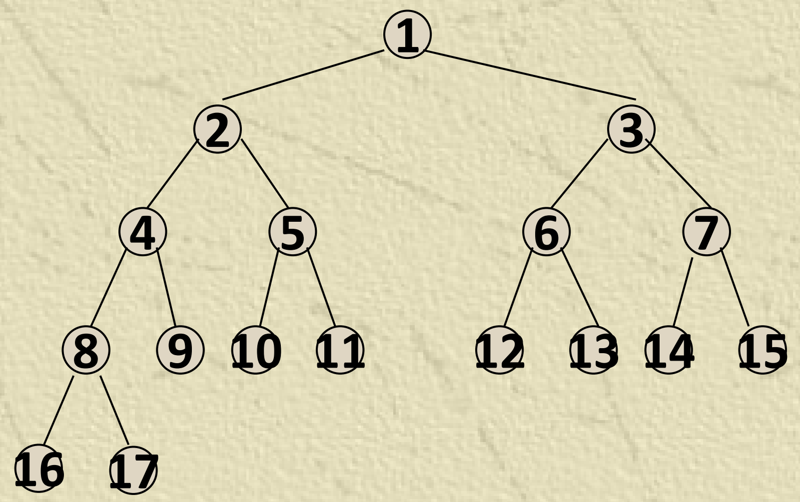
$2^{k-1}-1 < n \le 2^k-1$，**即：**

$2^{k-1} \le n < 2^k$，**于是有：**

$$k-1 \le \log_2 n < k$$

$$\because k\text{为整数}, \therefore \text{取} k = \lfloor \log_2 n \rfloor + 1$$

性质5 如果将一棵有n个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号1, 2, …, n-1,n，然后按此结点编号将树中各结点顺序地存放于一个一维数组中, 并简称编号为i的结点为结点i ($1 \le i \le n$)。则有以下关系：



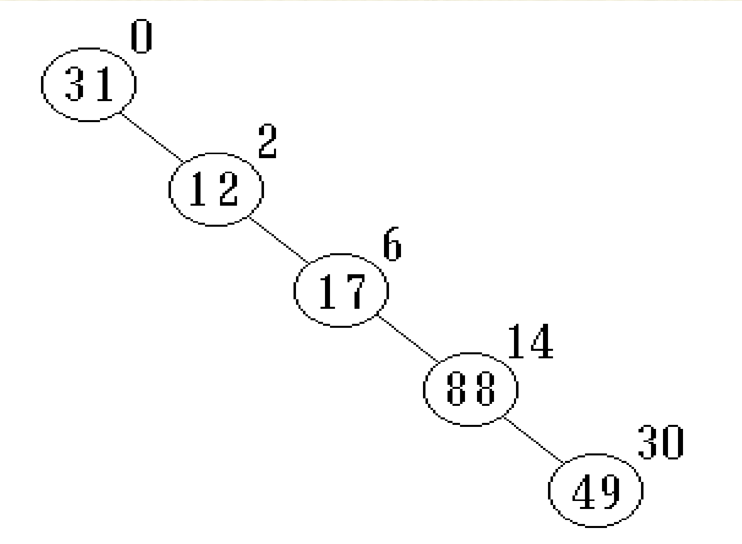
- 若i == 1, 则 i 无双亲
  若i > 1, 则 i 的双亲为$\lfloor i/2 \rfloor$
- 若2*i <= n, 则 i 的左孩子为2*i；否则，i无左孩子，必定是叶结点，二叉树中i>$\lfloor n/2 \rfloor$的结点必定是叶结点
  若2*i+1 <= n, 则 i 的右孩子为2*i+1，否则，i 无右孩子
- 若 i 为奇数, 且i不为1，则其左兄弟为i-1，否则无左兄弟；
  若 i 为偶数, 且小于 n，则其右兄弟为i+1，否则无右兄弟
- i 所在层次为$\lfloor \log_2 i \rfloor$+1

7.3 二叉树的存储

一.顺序存储结构

- 指用一组连续的存储单元存储二叉树的结点数据。
- 要求：必须把二叉树中的所有结点，按照一定的次序排成为一个线性序列，结点在这个序列中的相互位置能反映出结点之间的逻辑关系。
- 在结点的线性序列中，如何反映结点之间的逻辑关系（分支关系）？
  - 对于完全二叉树和满二叉树，结点的层次序列足以反映整个二叉树的结构。
  - 对于一般二叉树，则需要通过添加虚结点将其扩充为完全二叉树。

• 由于一般二叉树必须仿照完全二叉树那样存储，可能会浪费很多存储空间，单支树就是一个极端情况。



单支树

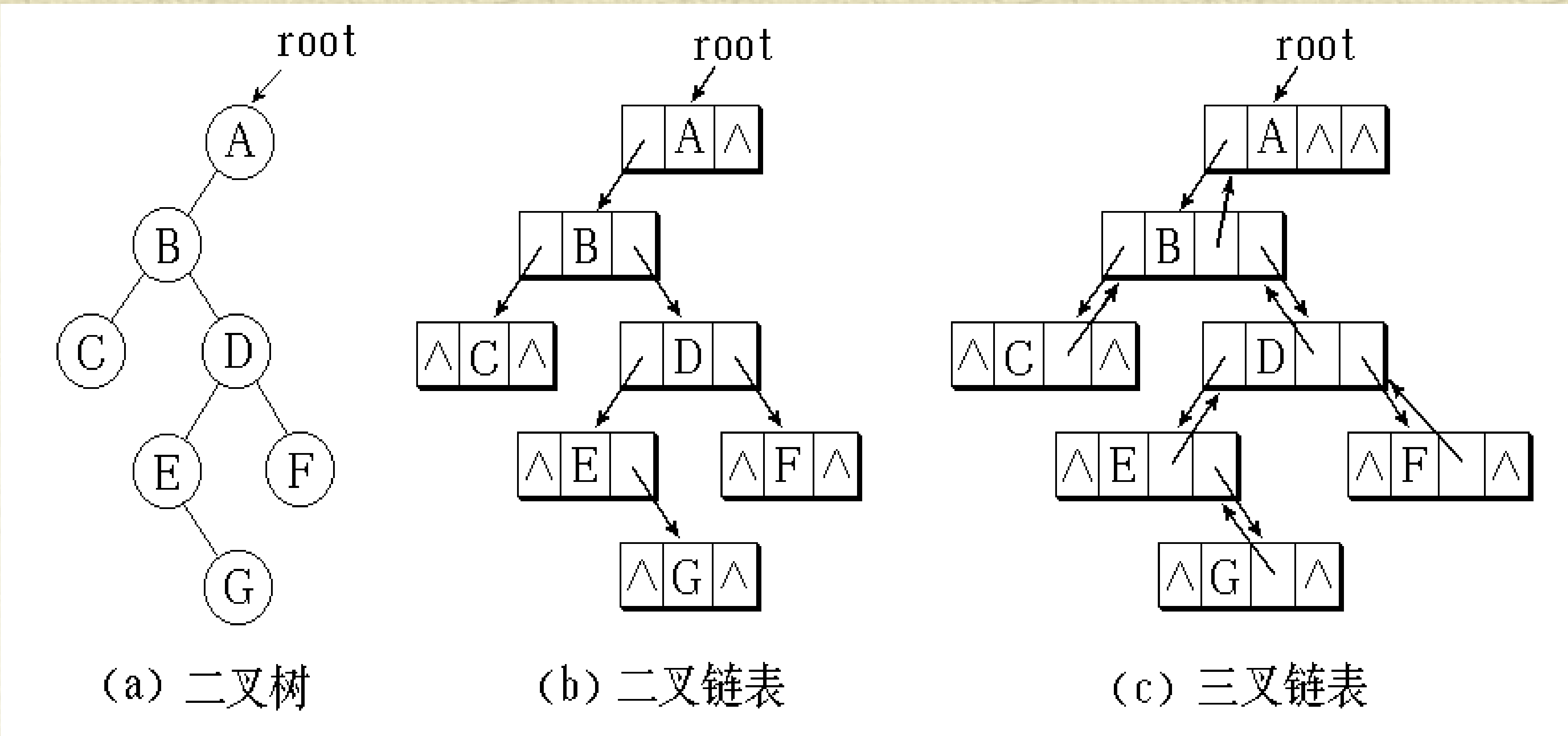
• 若要在树中经常插入和删除结点时，由于要大量移动结点，显然在这种情况下采用顺序方式并不可取。

二. 链式存储结构

- 由于二叉树的每个结点最多有左、右两个孩子，因此在采用链式存储表示时，每个结点至少需要包含三个域：数据域和左、右指针域。

lchild	Data	rchild

- 一个二叉树中所有这种形式的结点，再加上一个指向根结点的头指针，就构成了此二叉树的链式存储结构，称之为二叉链表。
- 如果想能够找到父结点，则可以增加一个指向父结点的指针域，则构成三叉链表。



二叉树链表表示的示例

二叉链表结构定义：

```
template <class T>
struct BiNode
{   T data;      // 结点数据
    BiNode<T> *lchild;    // 左孩子的指针
    BiNode<T> *rchild;    // 右孩子的指针
};
```

二叉树的类定义

```
template <class T>
class BiTree{
   BiNode<T>* root;   // 根指针
 public:
   BiTree() { root=NULL; }
   BiTree(vector<T> &pre);
   BiTree<T>::BiTree(const BiTree<T> &tree);
   ~BiTree();
   void PreOrder();
   void InOrder();
   void PostOrder();
   void LevelOrder();
   int Height();
   BiNode<T> *Search(T e);
   BiNode<T> *SearchParent(BiNode<T>*child);
};
```

6.3 二叉树的遍历

- 遍历二叉树

  按某条搜索路径访问树中每一个结点，使得每个结点均被访问一次，且仅被访问一次。

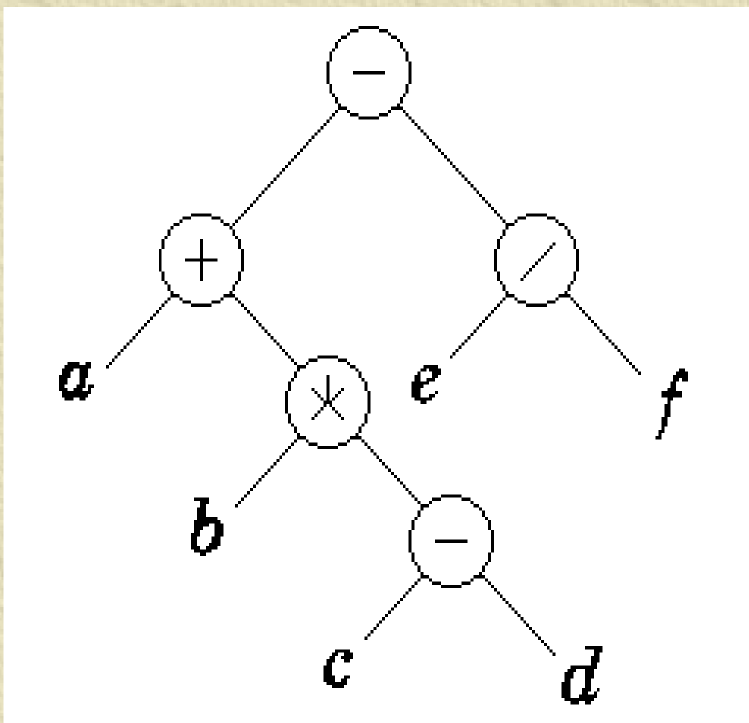
- 六种访问次序（**N**--访问根，**L**--遍历左子树，**R**--遍历右子树）：
  - **NLR**、**LNR**、**LRN**、**NRL**、**RNL**、**RLN**

- 若限定按先左后右的次序遍历，则有如下三种遍历次序：
  - 先序遍历：**NLR**
  - 中序遍历：**LNR**
  - 后序遍历：**LRN**

先序遍历

先序遍历二叉树算法的框架是

- **若二叉树为空，则空操作；**
- **否则**
  - **访问根结点 (V)；**
  - **先序遍历左子树 (L)；**
  - **先序遍历右子树 (R)。**



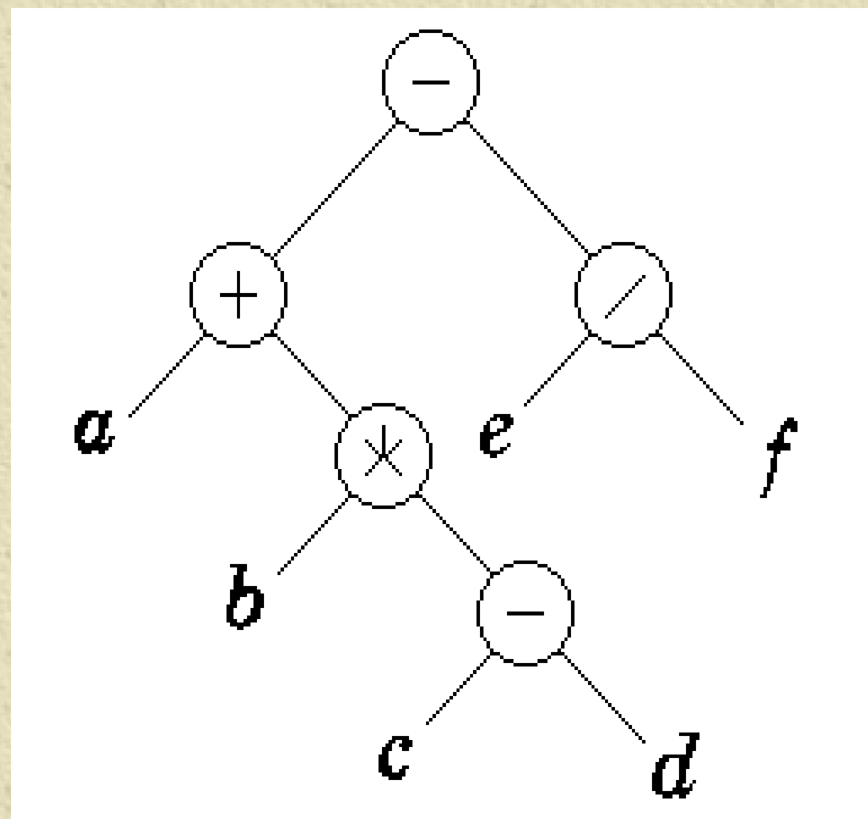
遍历结果:

- + a * b - c d / e f

中序遍历

中序遍历二叉树算法的框架是：

- 若二叉树为空，则空操作；
- 否则
  - 中序遍历左子树 (L)；
  - 访问根结点 (V)；
  - 中序遍历右子树 (R)。



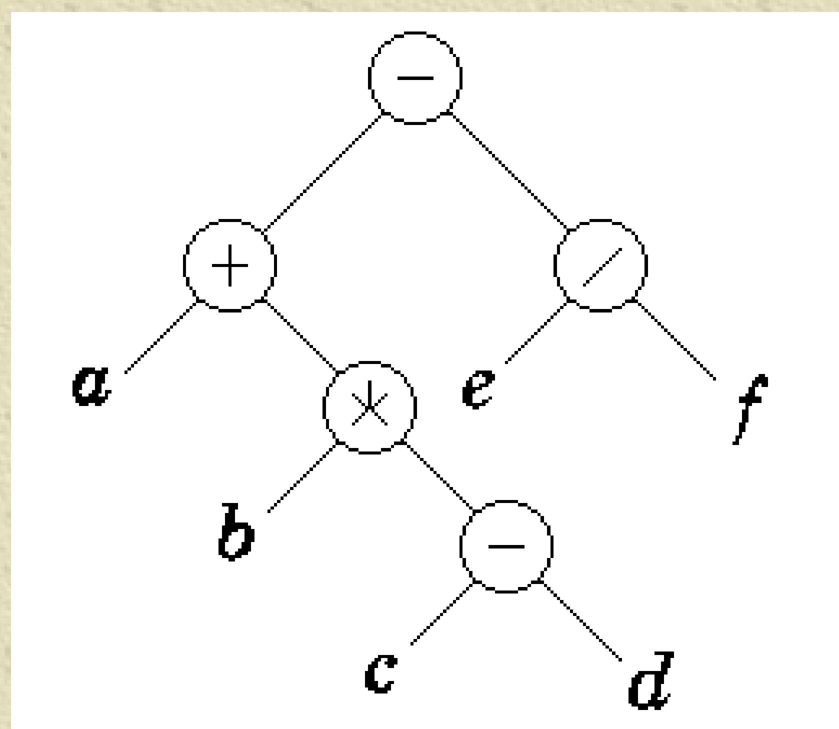
遍历结果 *a + b * c - d - e / f*

表达式语法树

后序遍历

后序遍历二叉树算法的框架是

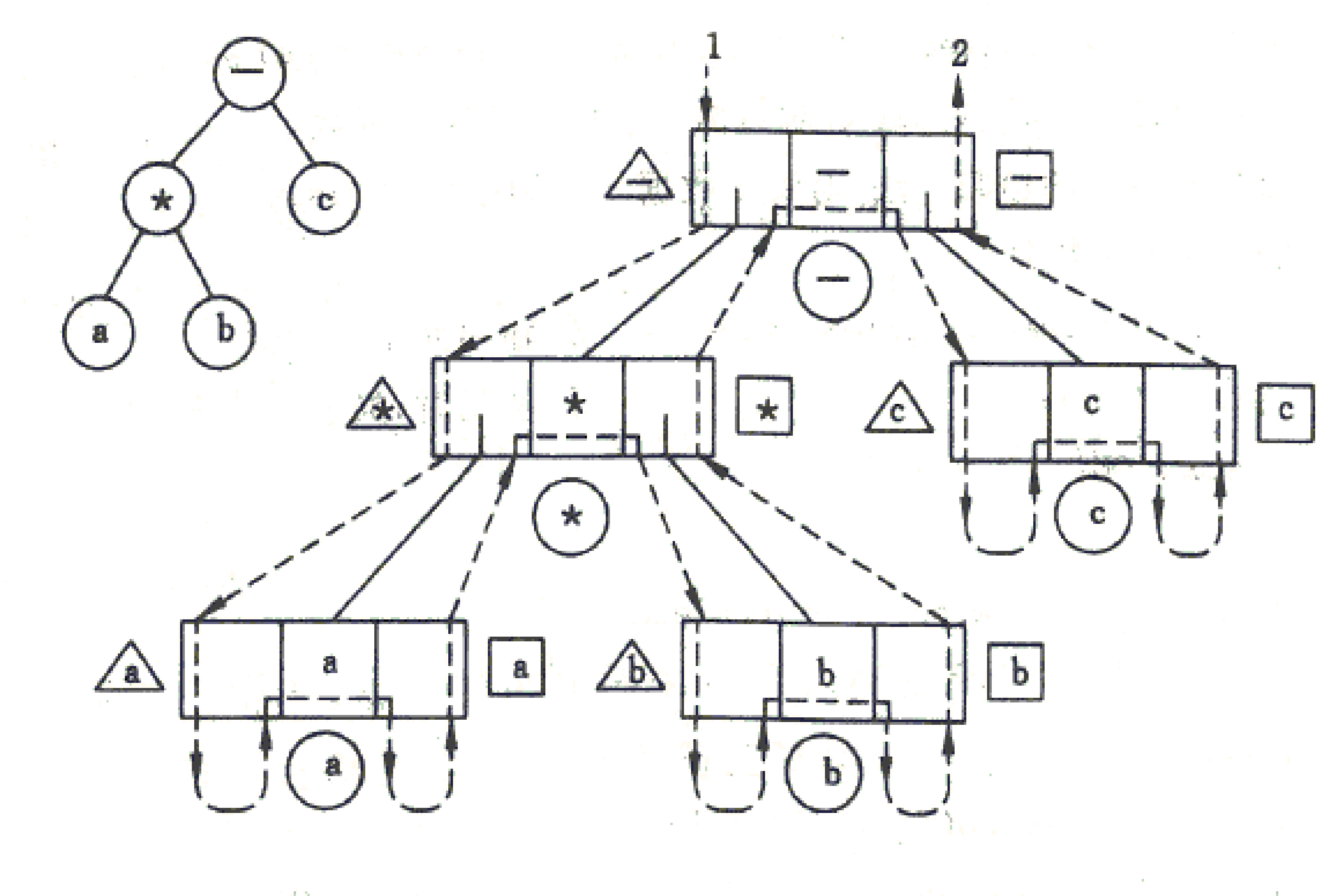
- 若二叉树为空，则空操作；
- 否则
  - 后序遍历左子树 (L)；
  - 后序遍历右子树 (R)；
  - 访问根结点 (V)。



遍历结果:

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

先、中、后序遍历的流程

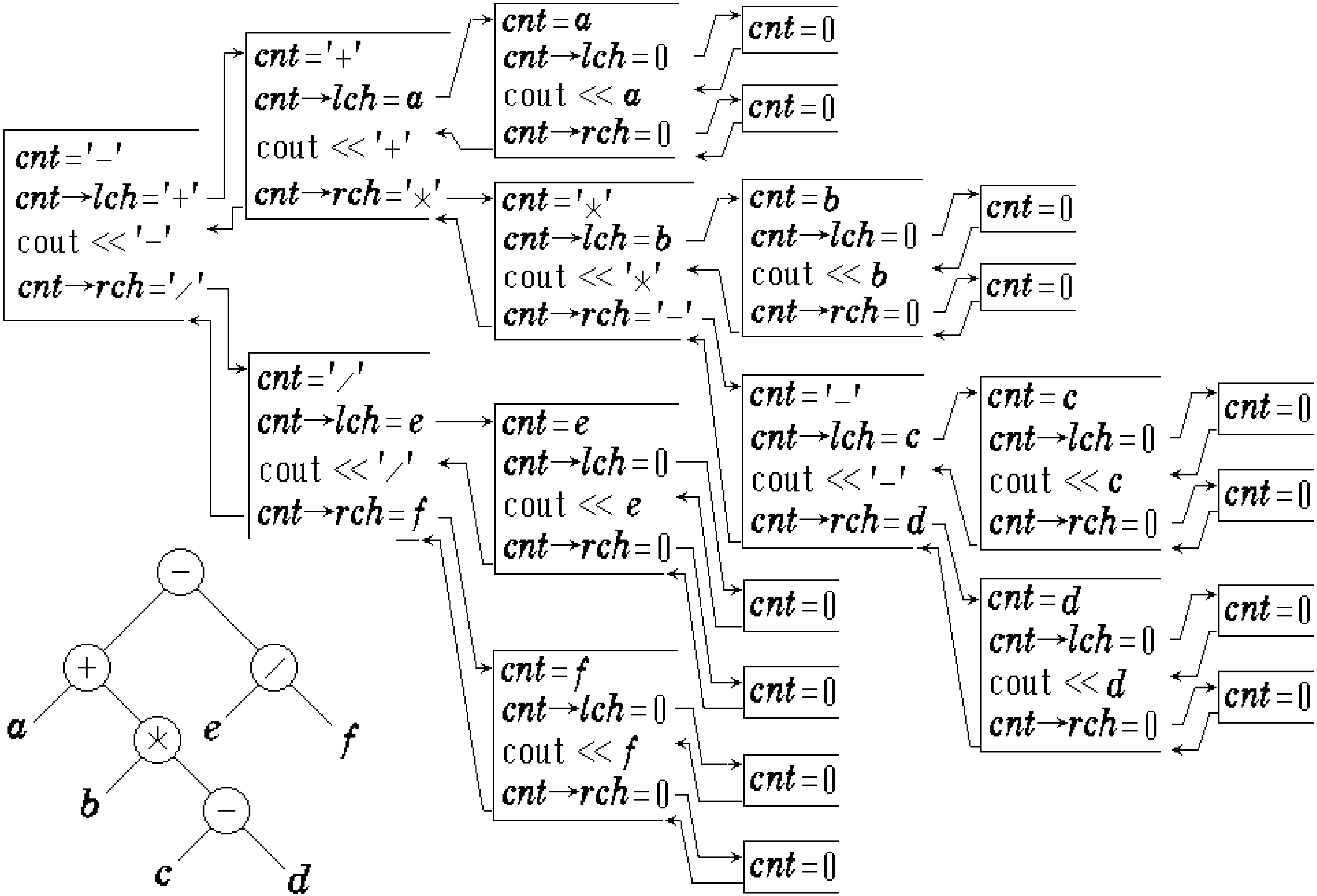


中序遍历的递归算法：

```
template <class T>
void BiTree<T>::InOrder(BiNode<T> *p)
{   if(p==NULL) return;
    InOrder(p->lchild);
    cout << p->data;
    InOrder(p->rchild);
}
```

算法的
时间复杂度？

```
template <class T>
void BiTree<T>::InOrder()
{  InOrder(root)；  }
```



中序遍历二叉树的递归过程图解

思考题

如何实现二叉树中序遍历的非递归算法？

中序遍历的非递归算法

```
template <class T>
void BiTree<T>::InOrder(BiNode<T> *t)
{   SeqStack S;     S.Push(t);
    while( !S.Empty() ){
        p=S.Top();
        while( p ){
            S.Push(p->lchild); p=p->lchild;     //向左走到尽头
        }
        p=S.Pop();     //空指针退栈
        if( !S.Empty() ){
            p=S.Pop();   cout<<p->data;   //访问结点
            S.Push(p->rchild)     //进入右子树
        }
    }
}
```

先序遍历的递归算法：

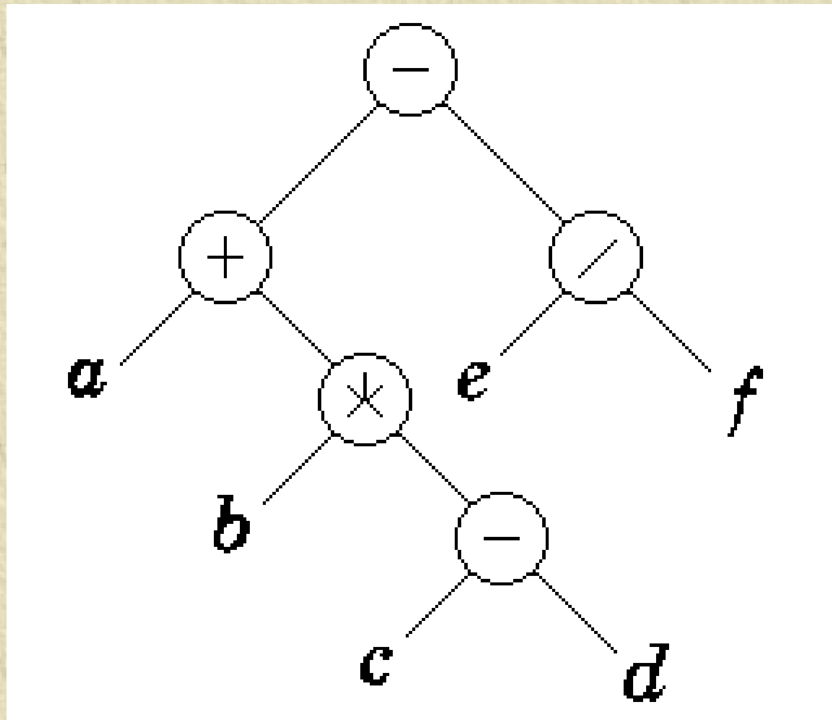
```
template <class T>
void BiTree<T>::PreOrder(BiNode<T> *p)
{   if(p==NULL) return;
    cout << p->data;
    PreOrder(p->lchild);
    PreOrder (p->rchild);
}

template <class T>
void BiTree<T>::PreOrder()
{  PreOrder(root)  }
```

层序遍历

层序遍历二叉树算法的框架是

- 若二叉树为空，则空操作；
- 将根结点入队
- 如队列不空，循环：
  - 做出队操作，队头元素作为当前结点；
  - 将当前结点的左右孩子入队
- 最后，出队序列就是层序遍历序列。



遍历结果:

- + / a * e f b - c d

二叉树的层序遍历算法

```
template <class T>
void BiTree<T>::LevelOrder() {
    Queue<BiNode<T> *> Q;   // Q为指针队列
    if( !root ) return;
    Q.EnQueue(root);
    while( !Q.Empty() )
     {   BiNode<T> *p= Q.Dequeue();
         cout<<p->data;
         if(p->lchild)  Q.EnQueue(p->lchild);
         if(p->rchild)  Q. EnQueue(p->rchild);
    }
}
```

7.4.3 二叉树的构造和析构算法

- 二叉树的建立
  - 目标：
    - 给定一棵二叉树的结点的值的序列，建立该二叉树对应的二叉链表。
- 若给定一棵二叉树的结点的先序序列，能建立一棵对应的二叉树吗？
  - 在序列中增加空指针标记

1、由单个遍历序列构造二叉树

```
template <class T>
BiNode<T> *BiTree<T>::CreateByPre()
{   e=getchar();
     if(e=='*')  return NULL;
    p=new BiNode<T>;
    p->data=e;
    p->lchild=CreateByPre();
    p->rchild=CreateByPre();
    return p;
}
```

1、由单个遍历序列构造二叉树

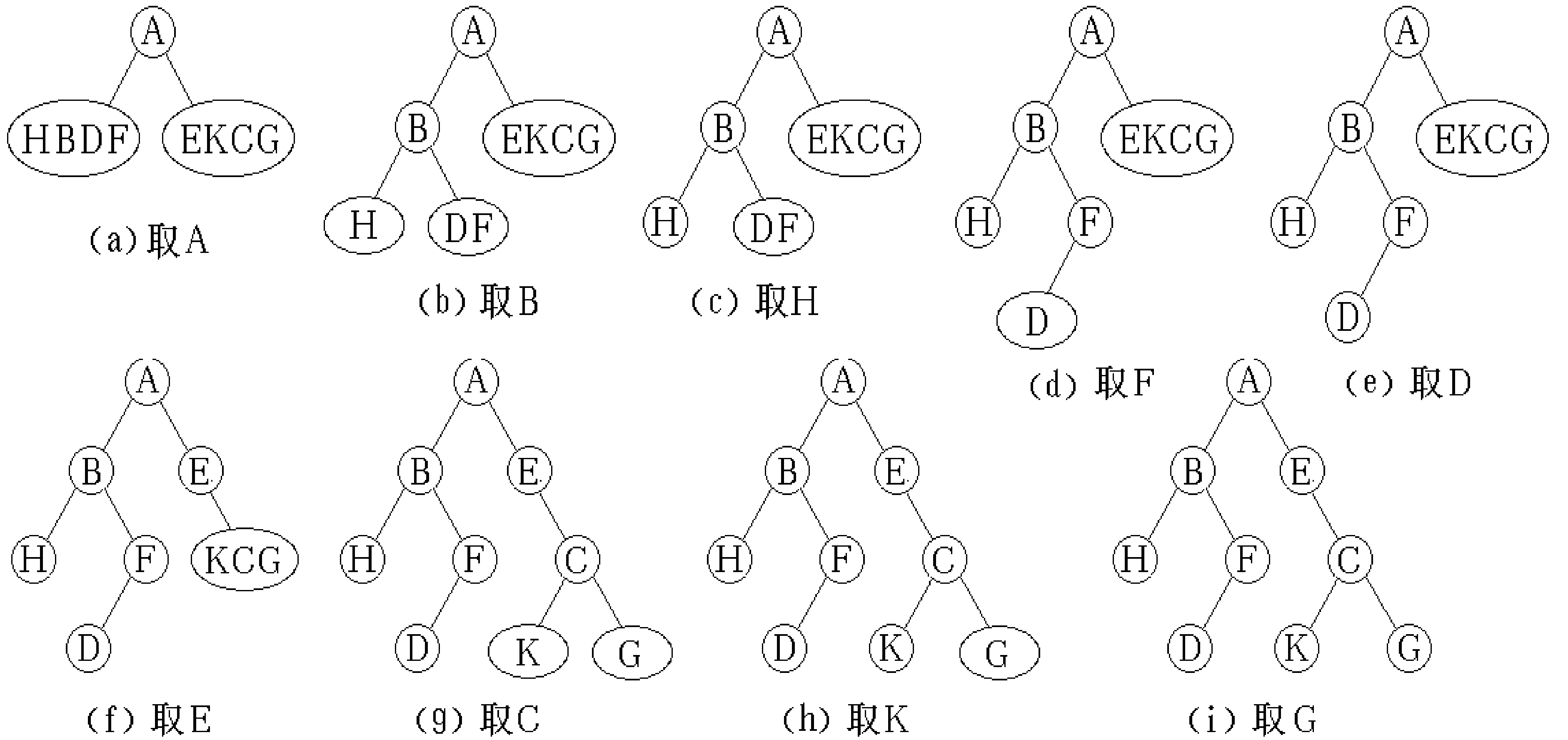
```
template <class T>
BiNode<T> *BiTree<T>::CreateByPre(vector<T> &pre,int &i)
{   e=pre[i];   i++;                // 提取当前数据
    if(e=='*')  return NULL;
    p=new BiNode<T>;  p->data=e;
    p->lchild=CreateByPre(pre, i);
    p->rchild=CreateByPre(pre, i);
    return p;
}
template <class T>
BiTree<T>::BiTree(vector<T> &pre) {
    i=0;
    root=CreateByPre(pre, i);
}
```

思考题

问：由添加空指针标记的单个中序或后序遍历序列是否可构造相应的二叉树？

2、由二个遍历序列构造二叉树

已知：先序序列 { ABHFDECKG }，
中序序列 { HBDFAEKCG }，
试构造相应的二叉树。



(a) 取A　(b) 取B　(c) 取H　(d) 取F　(e) 取D

(f) 取E　(g) 取C　(h) 取K　(i) 取G

性质：一棵二叉树的先序序列和中序序列可以唯一的确定这棵二叉树.

用归纳法证明：

1、当 $n = 1$时，结论显然成立；

2、假定当 $n <= k$ 时，结论成立；

3、当 $n = k + 1$ 时，假定先序序列和中序序列分别为：

$$\{a_1, \ldots, a_{k+1}\} \text{ 和 } \{b_1, \ldots, b_{k+1}\}$$

如中序序列中与先序序列中的a_1相同的元素:为b_j :

- ✓ $j = 1$时，二叉树无左子树，由 $\{a_2, \ldots, a_{k+1}\}$ 和 $\{b_2, \ldots, b_{k+1}\}$ 可以唯一的确定二叉树的右子树；
- ✓ $j = k+1$时，二叉树无右子树，由 $\{a_2, \ldots, a_{k+1}\}$ 和 $\{b_1, \ldots, b_k\}$ 可以唯一的确定二叉树的左子树；
- ✓ 如$2 <= j <= k$，则：
  - 子序列 $\{a_2, \ldots, a_j\}$ 和$\{b_1, \ldots, b_{j-1}\}$唯一地确定二叉树的左子树；
  - 子序列$\{a_{j+1}, \ldots, a_{k+1}\}$ 和 $\{b_{j+1}, \ldots, b_{k+1}\}$唯一地确定二叉树的右子树.

参数如何设置？

```
template <class T>
BiNode<T>* BiTree<T>::CreateByPreMid( vector<T> &pre,
                    vector<T> &mid, int ipre, int imid, int n )
{
  if(n==0) return NULL;
  p = new BiNode<T>;
  p->data = pre[ipre];
  for(i=0; i<n; i++)
      if( pre[ipre]==mid[imid+i] )  break;
  p->lchild = CreateByPreMid(pre, mid, ipre+1, imid, i);
  p->rchild = CreateByPreMid(pre, mid,
                          ipre+i+1, imid+i+1, n-i-1);
  return p;
}
```

3、拷贝构造函数

```
template <class T>
BiNode<T> * BiTree<T>::Copy(BiNode<T> *p)
{
    if( p==NULL )  return NULL;
    newp=new BiNode<T>;
    newp->data=p->data;
    newp->lchild= Copy(p->lchild);
    newp->rchild= Copy(p->rchild);
    return newp;
}
template <class T>
BiTree<T>::BiTree(const BiTree<T> &tree)
{  root=Copy(tree.root); }
```

4、析构函数

```
template <class T>
void BiTree<T>::Free(BiNode<T> *p)
{
    if(p==NULL)  return;
    Free( p->lchild );
    Free( p->rchild );
    delete p;        // 释放根结点
}
template <class T>     // 析构函数
BiTree<T>::~BiTree()
{ if(root)  Free(root); }
```

例：设计一个算法，将完全二叉树的顺序存储结构转换为二叉链表结构。

```
BiNode<T>*  turn( char A[], int n, int i ) {
    if( n<1 || i>n) return NULL;
    p=new BiNode<T>;
    p->data = A[i];
    if( 2*i <= n ) p->lchild=turn(A, n , 2*i);
    else  p->lchild=NULL;
    if( 2*i+1 <= n ) p->rchild=turn( A, n , 2*i+1);
    else  p->rchild=NULL;
    return p;
}
```

方法二 (非递归):

```
BiNode<T>* turn( char A[], int n )
{
    BiNode<T>* p=new BiNode<T>* [n+1];
    for( i=1; i<=n; i++ ){
        p[i] = new BiNode<T>;
        p[i]->data = A[i];
    }
    for( i=1; i<=n; i++ ){
        if(2*i <= n)  p[i]->lchild= p[2*i ];
        else p[i]->lchild= NULL;
        if(2*i+1 <= n) p[i]->rchild = p[2*i+1 ];
        else p[i]->rchild= NULL;
    }
    return p[1];
}
```

7.5 二叉树的其他操作算法

- 计算二叉树的结点数
- 计算二叉树的高度
- 根据关键值查找结点
- 查找结点的父结点

例1：计算二叉树结点数的算法

```
template <class T>
int BiTree<T>::Count(BiNode<T> *p)
{
    if( p==NULL )   return 0;
    left= Count(p->lchild);
    right=Count(p->rchild);
    return 1+left+right;
}
```

方法二：计算二叉树结点数的算法

```
template <class T>
void BiTree<T>::Count(BiNode<T> *p, int &num)
{
    if( p==NULL )   return;
    num++;
    left= Count(p->lchild);
    right=Count(p->rchild);
}
```

例2：求二叉树的高度

```
template <class T>
int BiTree<T>::Height( BiNode<T> *t )
{
    if(t==NULL)  return 0;
    left =Height( t->lchild );
    right=Height( t->rchild );
    if(left>right) return left+1;
    return right+1;
}
```

方法二：二叉树的高度为树中的结点的层次最大值

参数如何设置？

```
template <class T>
void BiTree<T>::Height( BiNode<T> *t，  int level,
                                          int& depth )

    if (t){
        if( level>depth) depth=level;
        Height(t->lchild, level+1, depth);
        Height(t->rchild, level+1, depth);
    }
}
```

例3：在二叉树中查找具有给定值的结点

```
template <class T>
BiNode<T> *BiTree<T>::Search(BiNode<T> *t, T e)
{
    if ( t==NULL) return NULL;
    if ( t->data == e ) return t;
    p= Search(t->lchild,e);
     if( p ) return p;
    return  Search( t->rchild,e) ;
}
```

例4：查找指定结点的父结点

```
template <class T>
BiNode<T> *BiTree<T>::SearchParent(BiNode<T> *t,
                                    BiNode<T>*child)
{
   if( t==NULL || child==NULL) return NULL;
   if(t->lchild==child || t->rchild==child)  return t;
   p= SearchParent(t->lchild, child);
   if( p ) return p;
   return SearchParent(t->rchild, child) );
}
```

练习题：设计一个算法, 在二叉树中查找关键值为key的结点的父结点。

```
template <class T>
BiNode<T>* SearchParent(BiNode<T> * t ,  T key)
{
    if( t == NULL)  return  NULL;
    if( t->lchild && t->lchild->data==key ) return  t;
    if( t->rchild && t->rchild->data==key) return  t;
    p= SearchParent(t->lchild, key);
    if( p ) return p;
    return  SearchParent( t->rchild, key) ;
}
```

7.6 线索二叉树

- 二叉链表结构的局限性：
  - 对于某个结点只能找到其左右孩子，而不能直接得到该结点在某种遍历序列中的前趋或后继结点。
  - 要想得到该信息只能通过遍历的动态过程才行。
  - 怎样保存遍历过程中得到的信息呢？
- 解决方法：
  - 可利用二叉链表结点结构中的空指针域，在空指针域中存放结点在某种遍历次序下的前驱和后继结点信息，这种附加的指针称为“线索”。
  - 为避免混淆，需改变结点结构，即增加两个标志域。

ltype	lchild	data	rchild	rtype

ltype	lchild	data	rchild	rtype

```
enum BiThrNodePointType{LINK, THREAD} ;
template <class T>
struct BiThrNode{
  BiThrNodePointType ltype, rtype;
  T data;
  BiThrNode<T> *lchild,*rchild;
};
```

标志位为0，表示指针指向孩子结点，

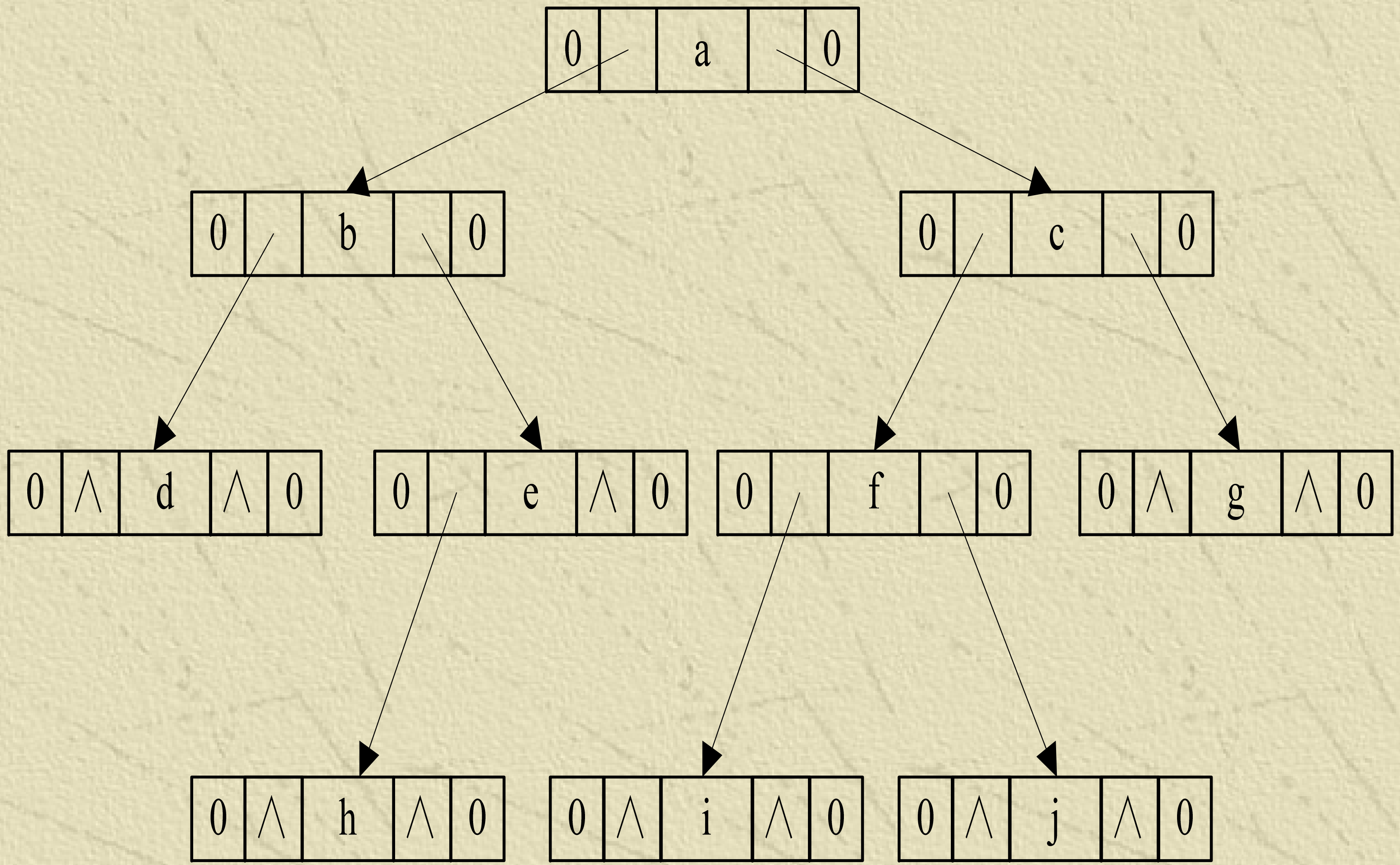
标志位为1，表示指针为线索。

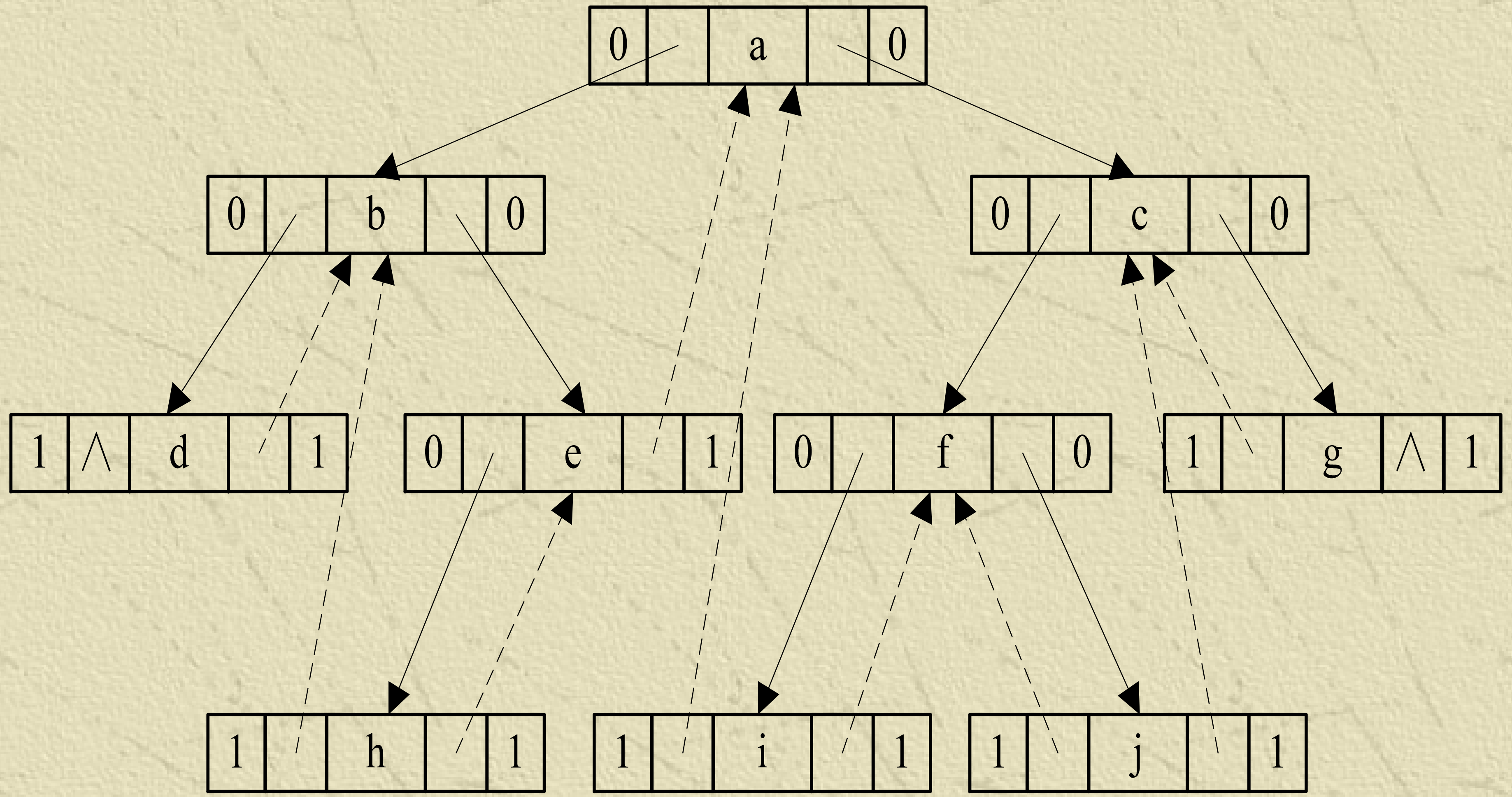
有关概念：

- 以上面结构所构成的二叉链表作为二叉树的存储结构，叫做线索链表。指向结点前驱或后继的指针叫做线索。

- 加上线索的二叉树叫线索二叉树。

- 线索化：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。


0 a 0
0 b 0
0 c 0
0 ∧ d ∧ 0
0 e ∧ 0
0 f 0
0 ∧ g ∧ 0
0 ∧ h ∧ 0
0 ∧ i ∧ 0
0 ∧ j ∧ 0



中序线索二叉树

中序线索二叉树类：

```
template <class T>
class InBiThrTree
{
    BiThrNode<T> *root;
 public:
    BinThrTree();
    ~BinThrTree();
    void InThreaded();    // 中序线索化
    ………
};
```

线索化的实现：

- 如何实现线索化？
  - 只要按该某种次序（先序、中序、后序）遍历二叉树，在遍历过程中，用线索取代空指针。
- 如何确立结点之间的前趋与后继关系？
  - 若指针**p**指向当前正在访问的结点，可另外附设一个指针**pre**，并始终保持指针**pre**指向当前访问的、指针**p**所指结点的前驱。

// 中序线索化算法：

```
template <class T>
void InBiThrTree<T>::InThreaded( BiThrNode<T> *p,
                                 BiThrNode<T> * &pre){
    if( p ){
        InThreaded(p->lchild, pre);      // 递归左子树线索化
        if( !p->lchild ) {               // 没有左孩子
            p->ltype=THREAD ;            // 前驱线索
            p->lchild=pre;               // 左孩子指针指向前驱
        }
        if( pre && !pre->rchild ){       // 前驱没有右孩子
            pre->rtype=THREAD ;          // 后继线索
            pre->rchild=p;               // 前驱右孩子指针指向后继
        }
        pre = p;                         // 保持pre指向p的前驱
        InThreaded(p->rchild, pre);  // 递归右子树线索化
    }
}
```

```
template <class T>
void InBiThrTree<T>::InThreaded()
{  if(root==NULL)  return;
    pre = NULL;      // 遍历序列中首结点的前驱为NULL
    InThreaded(root, pre);
    pre->rtype=THREAD;
    pre->rchild=NULL;   // 中序序列尾结点的后继为NULL
}
```

问题:

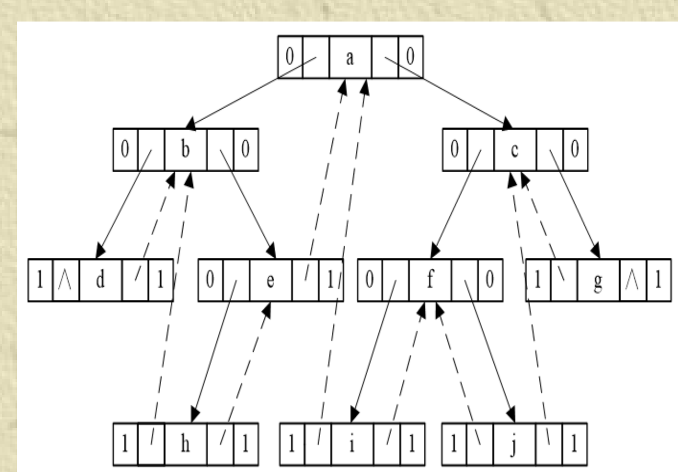
- 设计一个算法, 判断一棵给定的二叉树的中序遍历结点序列是否为非递减有序.

```
template <class T>
bool  JudgeOrder (BiNode<T> *p, BiNode<T> *&pre)
{
    if( !p ) return true;
    if( JudgeOrder (p->lchild, pre) ){
          if( pre && pre->data > p->data ) return false;
          pre = p;
          return  JudgeOrder (p->rchild, pre);
     }
     else  return false;
}
```

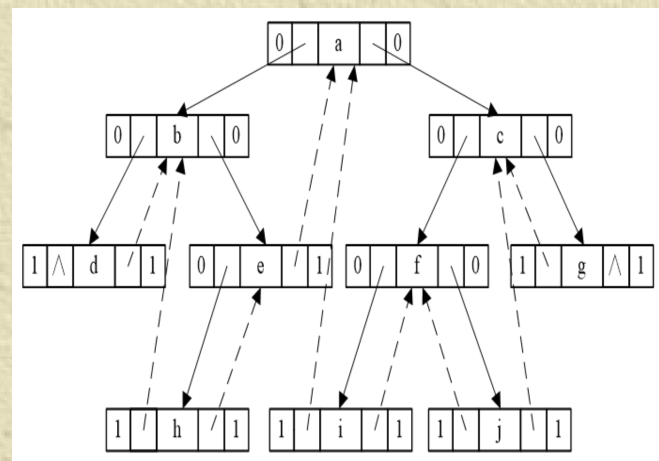
中序线索二叉树中，查找指定结点*p的中序后继结点

1、若 *p 的右子树为空，则 p->rchild 为右线索，直接指向 *p 的中序后继结点。

2、若 *p 的右子树非空，则 *p 的中序后继必是其右子树中第一个遍历到的结点，也就是从*p 的右孩子开始，沿左指针链往下查找，直到找到一个没有左孩子的结点为止。



对中序线索二叉树进行中序遍历的算法：

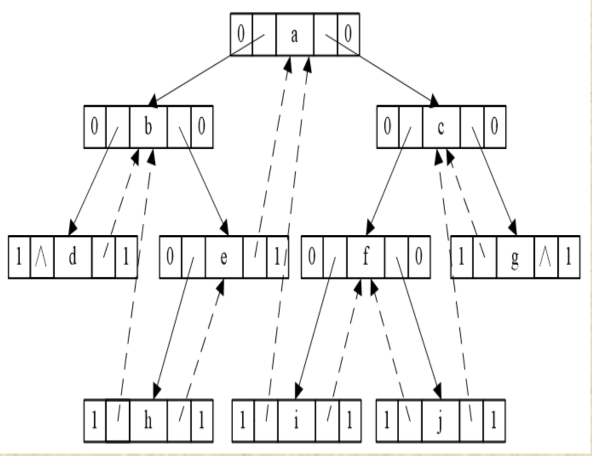



```
template <class T>
void InBiThrTree<T>::Travese() {
   p=root;
   if( !p ) return;
   while( p ){
        while( p->ltype==Link ) p = p->lchild;
        cout<< p-data;
        while( p->rtype == THREAD && p->rchild){
             p = p->rchild; cout<< p->data;
        }
        p = p->rchild;
   }
}
```

中序线索二叉树中，查找指定结点*p的中序前驱结点

1、若 *p 的左子树为空，则 p->lchild 为左线索，直接指向 *p 的中序前驱结点。

2、若 *p 的左子树非空，则从 *p 的左孩子出发，沿右指针链往下查找，直到找到一个没有右孩子的结点为止。



后序线索二叉树中，查找指定结点*p的后序后继结点

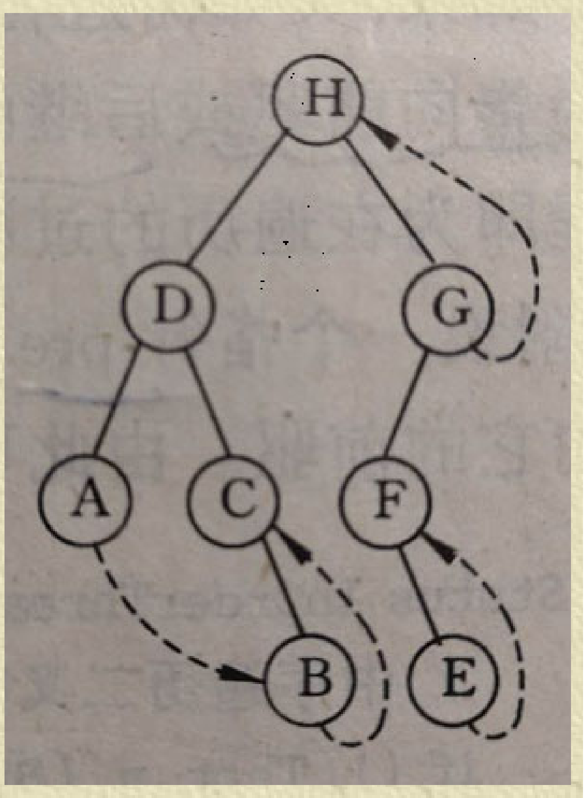
分四种情况：

1、若 *p是根，则其后继为空；

2、若 *p是其双亲的右孩子，则*p的后继就是其双亲结点；

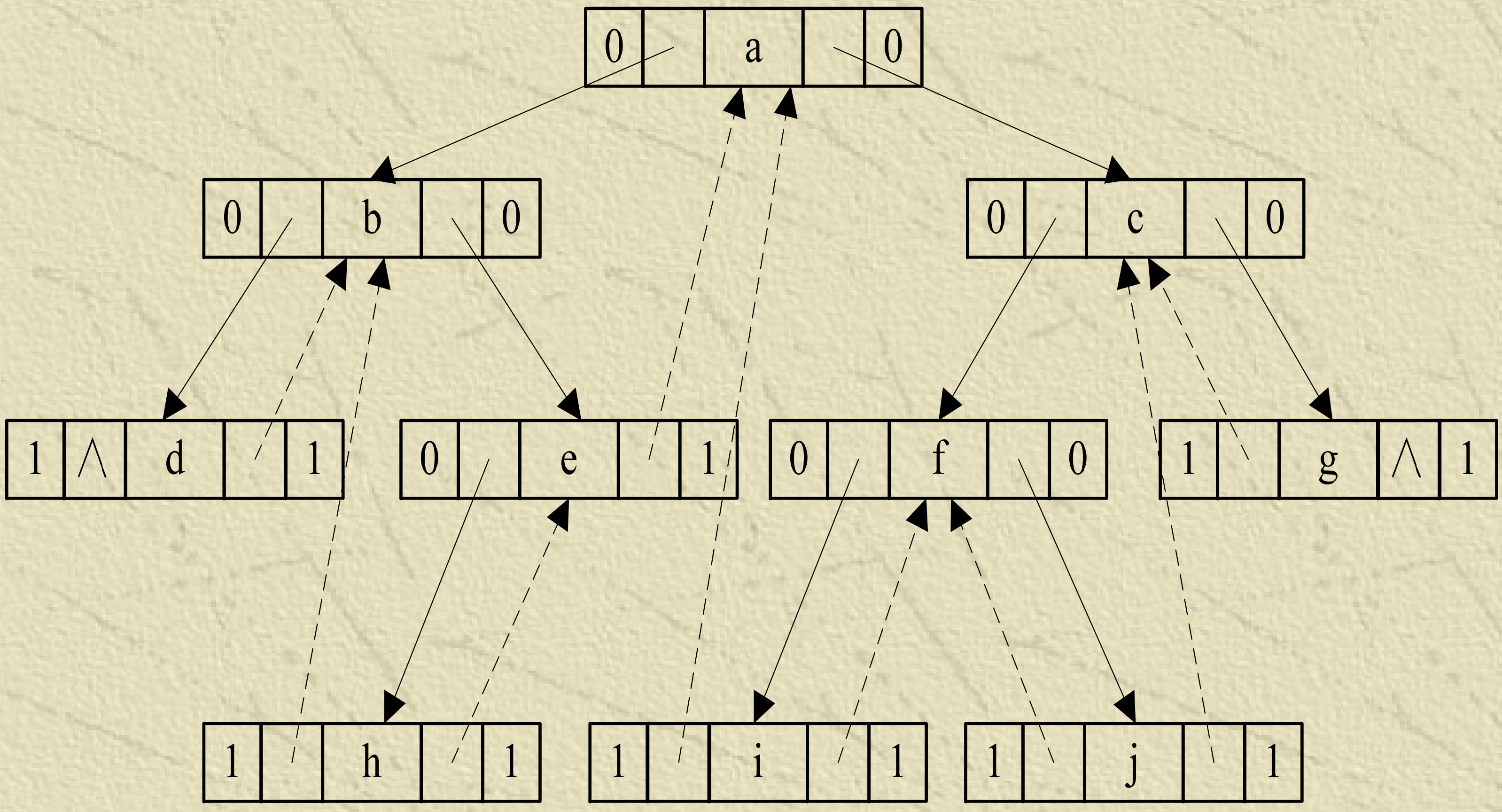
3、若 *p是其双亲的左孩子，但*p无右兄弟时，则*p的后继就是其双亲结点；

4、若 *p是其双亲的左孩子，且有右兄弟时，则*p的后继是其双亲的右子树中第一个后序遍历到的结点，它是该子树的“最左下的叶结点”。

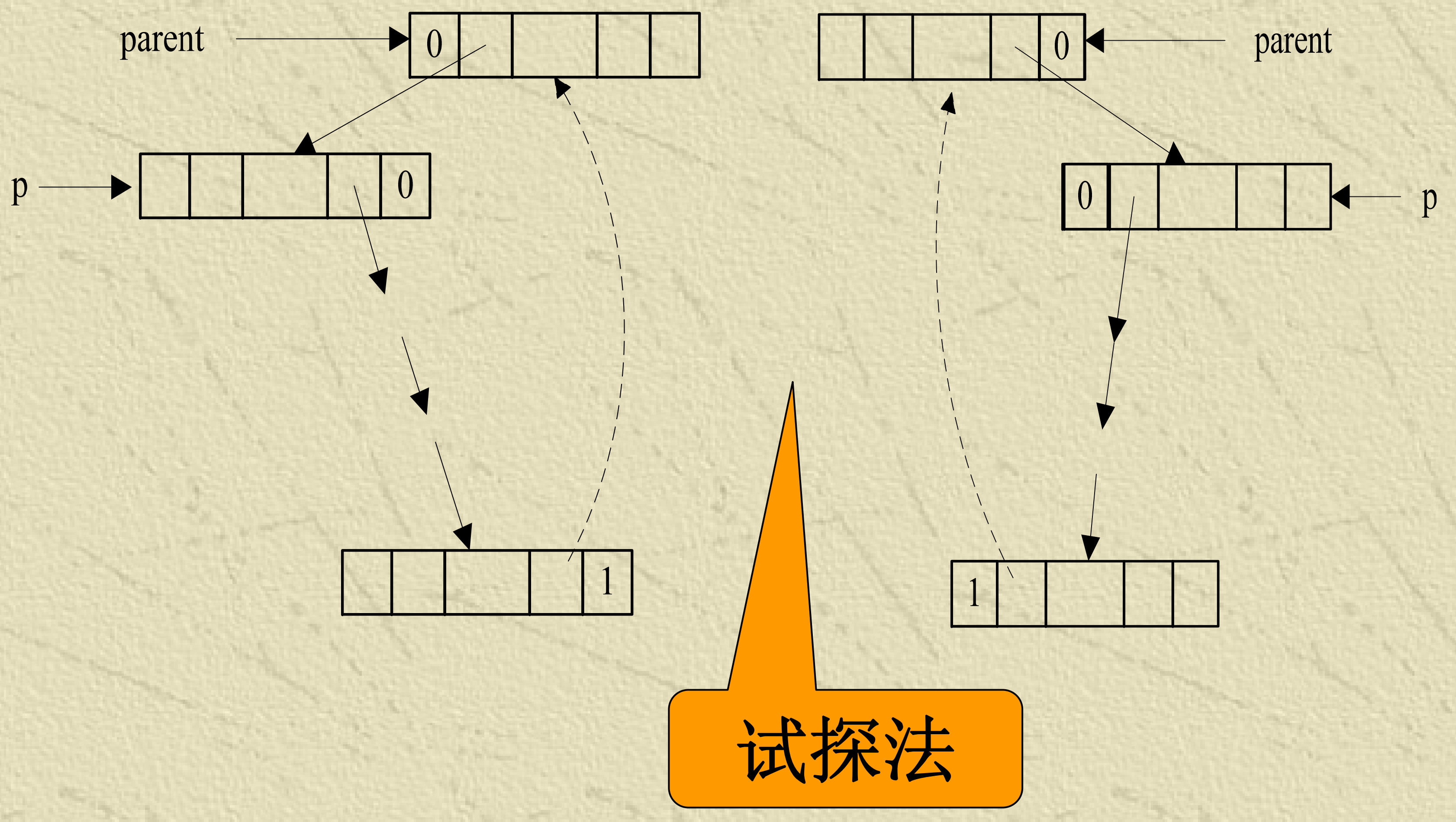


未必能找到！

4、求父结点的算法



*p和*parent存在如下两种情形：



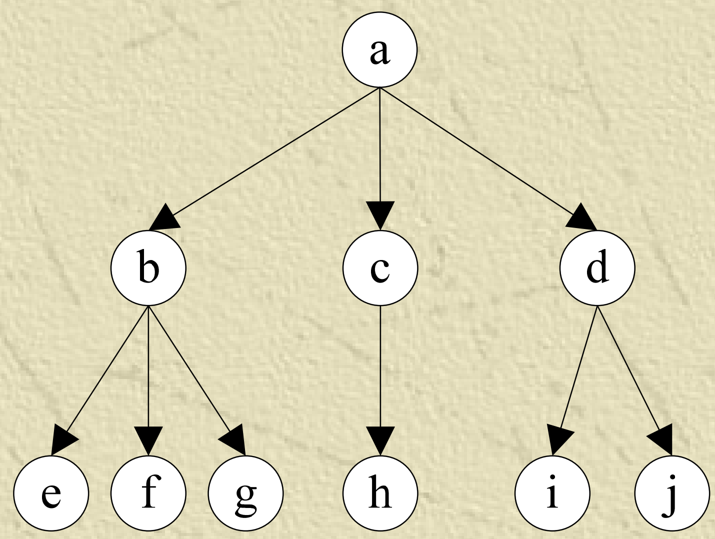
```
template <class T>
BiThrNode<T> *InBiThrTree<T>::GetParent(BiThrNode<T> *p)
{   if( !p || p==root ) return NULL;
    for( parent=p; parent->rtype==LINK; )
        parent=parent->rchild;
    parent=parent->rchild;   // parent是*p的最右下方结点的后继指针
    if( parent && parent->lchild==p) // 猜测*p是否是左孩子
        return parent;
    for( parent=p; parent->ltype==LINK; )
        parent=parent->lchild;
    parent=parent->lchild;  // parent是*p的最左下方结点的前驱指针
    return parent;       // parent一定是*p的父指针
}
```

7.7 树的存储结构与算法

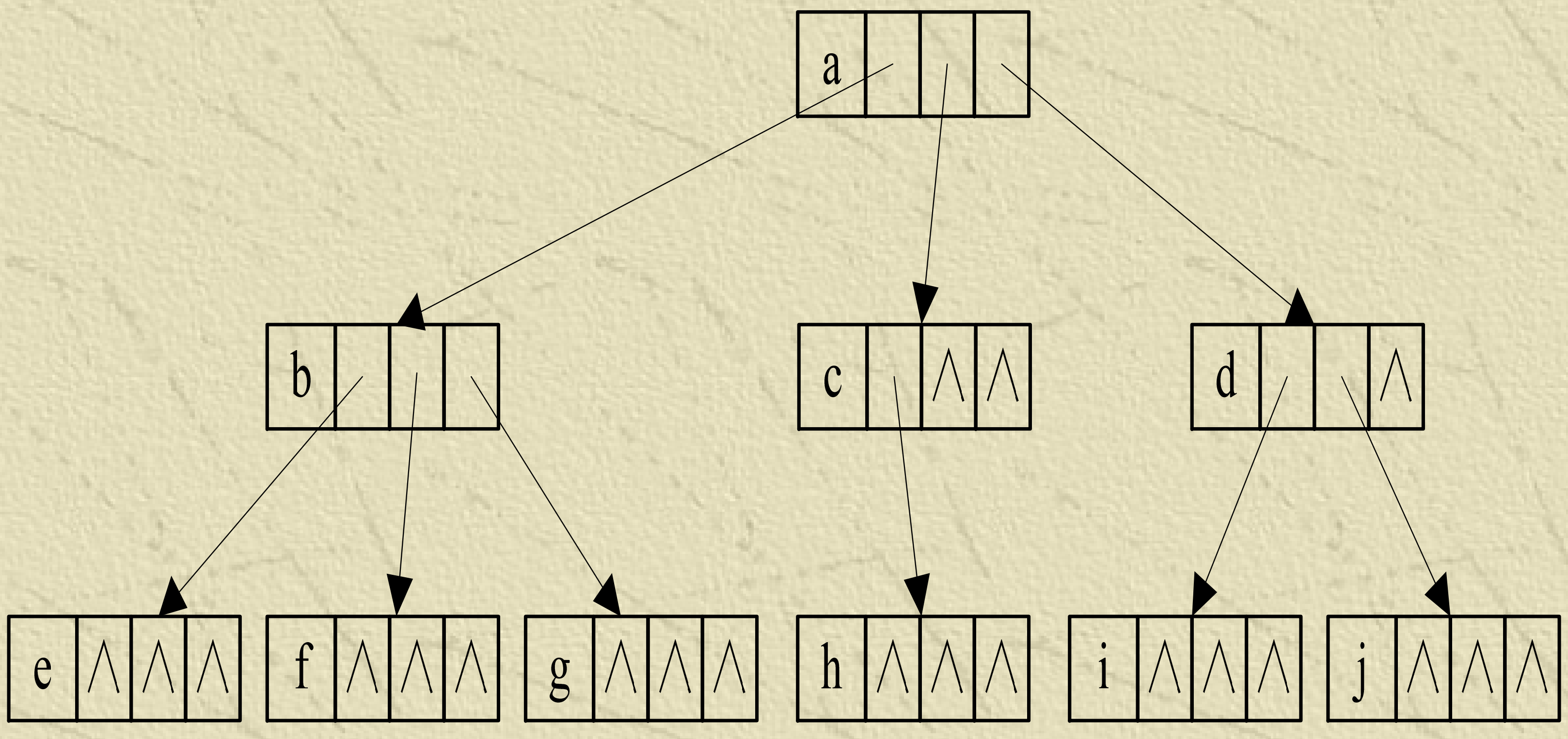
- 多叉链表表示法
- 广义表表示
- 孩子表示法
- 双亲表示法
- 双亲-孩子链表示法
- 二叉链表表示法

1. 多叉链表表示法

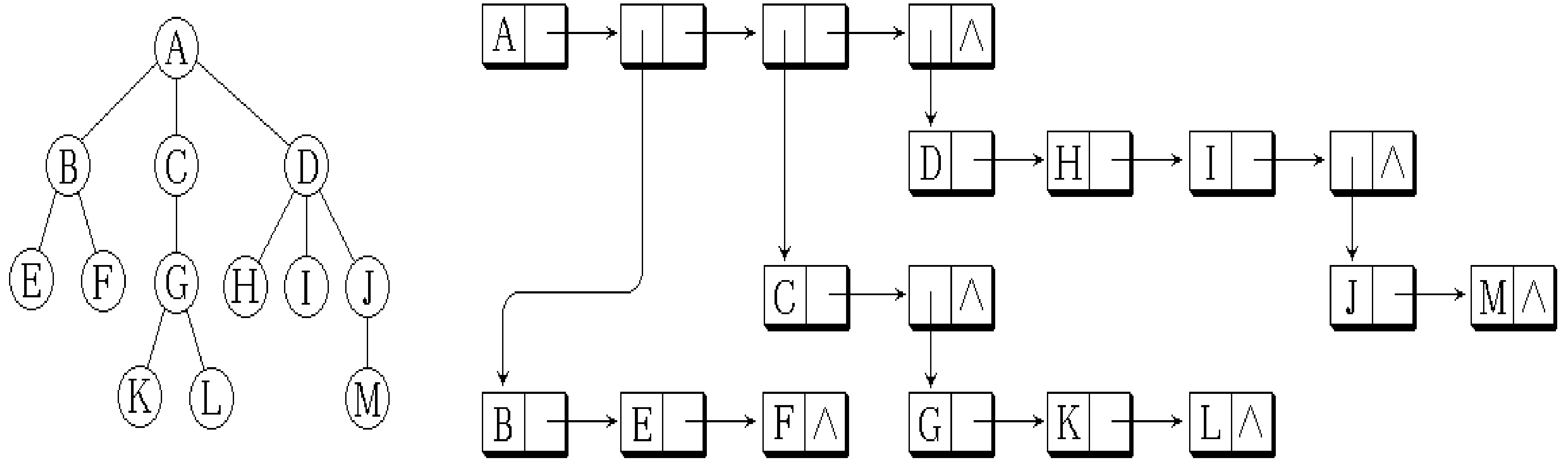
- 若树的度为*K*，则在结点结构中设置*K*个孩子指针域，使所有结点同构。



树的多叉链表表示

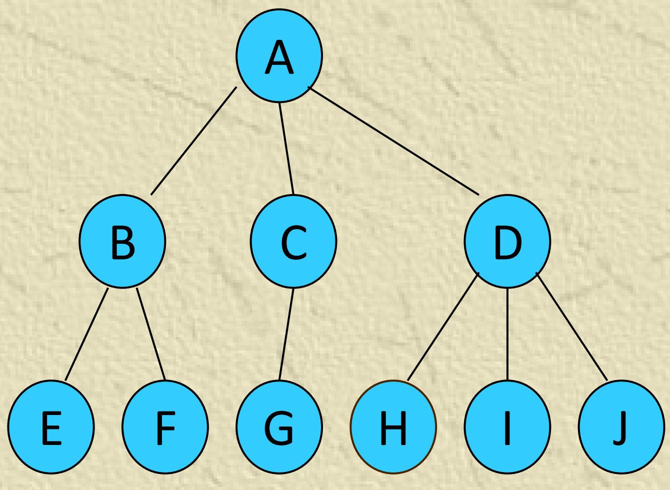


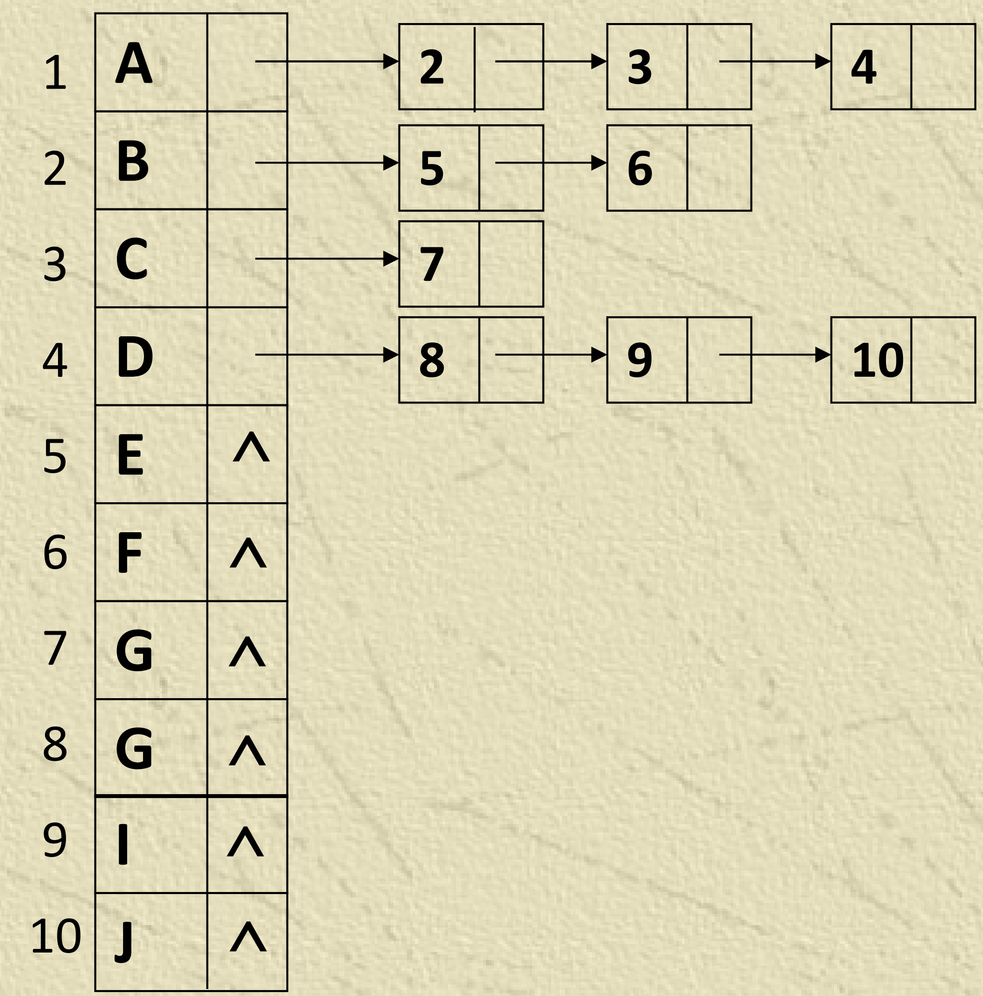
2. 树的广义表表示



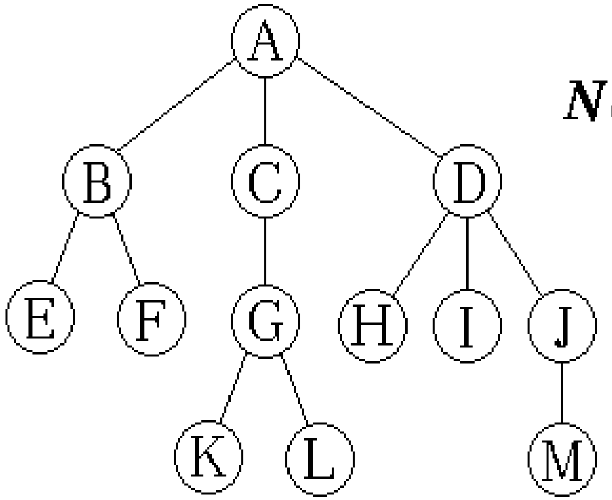
(a) 树　　(b) 广义表表示

3. 孩子表示法：





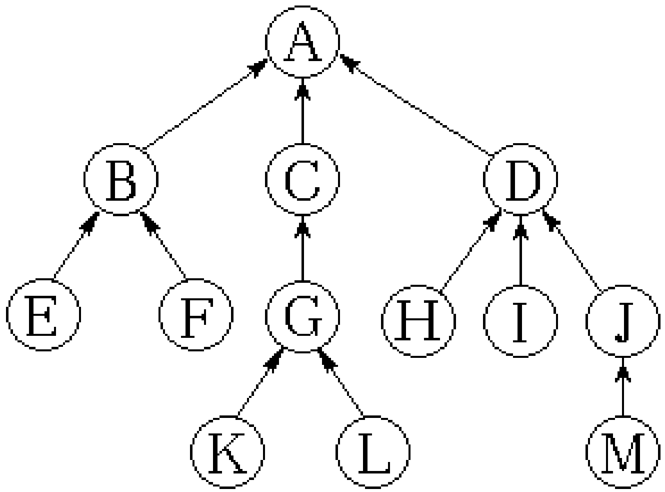
4. 双亲表示法：



（a）树

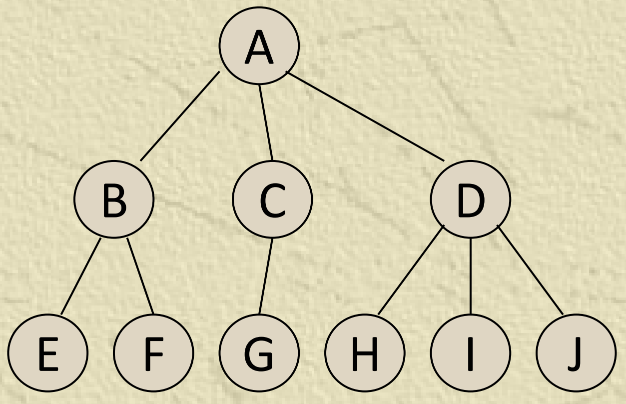
NodeList	1	2	3	4	5	6	7	8	9	10	11	12	13
data	A	B	E	F	C	G	K	L	D	H	I	J	M
parent	0	1	2	2	1	5	6	6	1	9	9	9	12

（b）双亲表示数组



（c）双亲表示图解

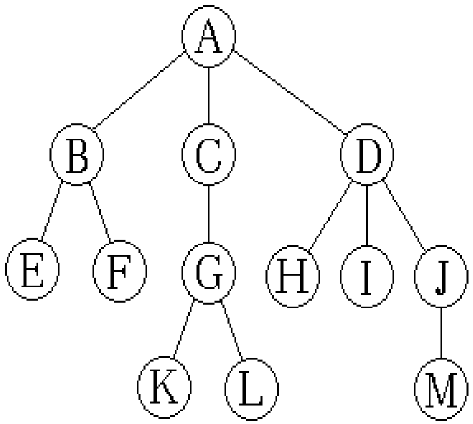
5. 双亲-孩子链表示：



1	A	0		→ 2 → 3 → 4
2	B	1		→ 5 → 6
3	C	1		→ 7
4	D	1		→ 8 → 9 → 10
5	E	2	^	
6	F	2	^	
7	G	3	^	
8	G	4	^	
9	I	4	^	
10	J	4	^	

6. 树的二叉链表存储表示法（孩子兄弟表示法）

data	***firstChild***	***nextSibling***



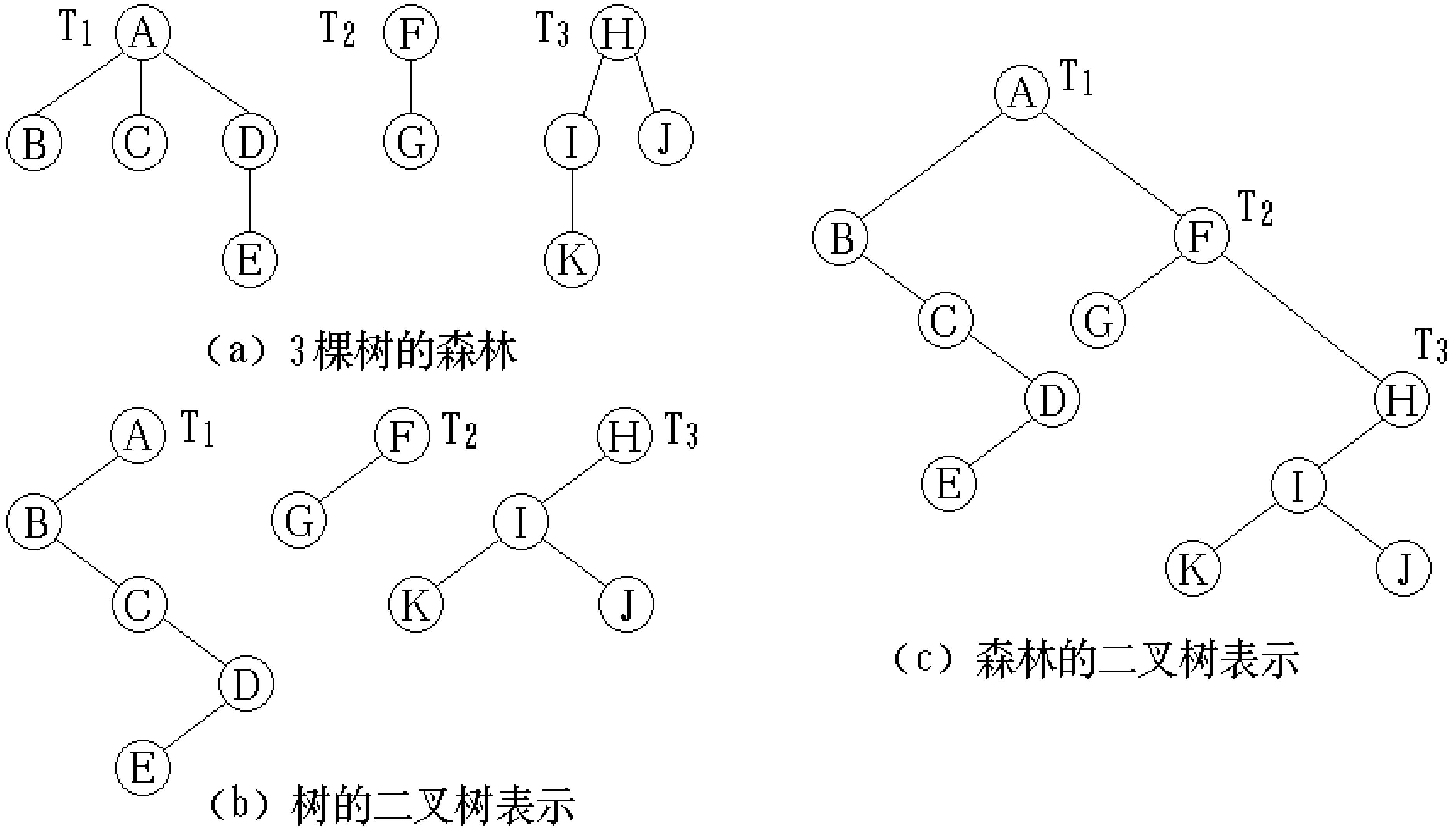
（a）树

森林与二叉树的转换

- 树、森林与二叉树之间有一个自然的一一对应关系。

- 转换方法：
  - 树 → 二叉树
    - 兄弟结点间加连接（虚线）
    - 让每个结点只与最左孩子保持联系，与其余孩子的关系去掉
  - 森林 → 二叉树
    - 森林 → 树 → 二叉树 （添加一个虚根结点）
    - 去掉虚根结点

森林与二叉树的转换



森林与二叉树的对应关系

(1) 森林转化成二叉树的形式化规则：

若F={T_1，T_2，…，T_n} 是森林，则：

① 若F为空，即n = 0，则

对应的二叉树B为空二叉树。

② 若F不空，则

对应二叉树B的根root (B)是F中第一棵树T_1的

根root (T_1)；

其左子树为B (T_{11}, T_{12}, …, T_{1m})，其中，

T_{11}, T_{12}, …, T_{1m}是root (T_1)的子树；

其右子树为B (T_2, T_3, …, T_n)，其中，

T_2, T_3, …,T_n是除T_1外其它树构成的森林。

二叉树到森林的转换

- 转换方法：
  - 若某结点是其双亲的左孩子，则把该结点的右孩子、右孩子的右孩子、…，都与该结点的双亲结点连接起来。
  - 然后去掉所有原双亲结点到右孩子的连线

- 二叉树 → 树　　　仅当根结点无右孩子；

  二叉树 → 森林　　仅当根结点有右孩子。

(2) 二叉树转换为森林的形式化规则：

若B=(root, LB, RB) 是一棵二叉树，则：

① 如果*B*为空，则

对应的森林*F*也为空。

② 如果*B*非空，则

*F*中第一棵树T_1的根为*root*；

T_1的根的子树森林{ T_{11}, T_{12}, …, T_{1m} }是由 *root* 的左子树 *LB* 转换而来；

F 中除了 T_1 之外其余的树组成的森林{ T_2, T_3, …, T_n } 是由 *root*的右子树 *RB* 转换而成的森林。

7.7.2 树的操作算法

- 树的遍历
- 计算树的高度
- 计算树中所有结点的度
- 树的构造

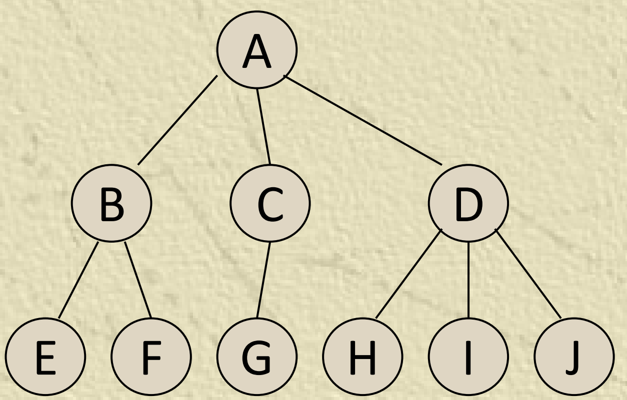
树的遍历

先根（先序）遍历

若树非空，则

1、访问根结点

2、依次先根遍历树的各子树



先序遍历序列：

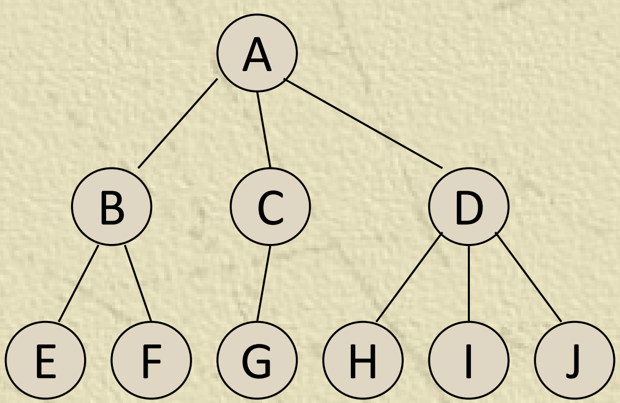
A，B，E，F，C，G，D，H，I，J

后根（后序）遍历

若树非空，则

1、依次后根遍历树的各子树

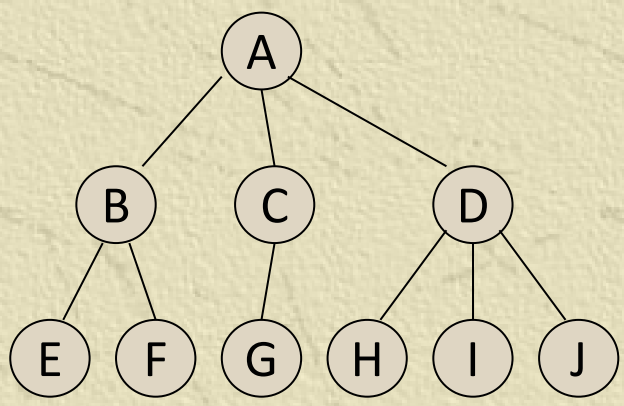
2、访问根结点



遍历序列：

E，F，B，G，C，H，I，J，D，A

层序遍历

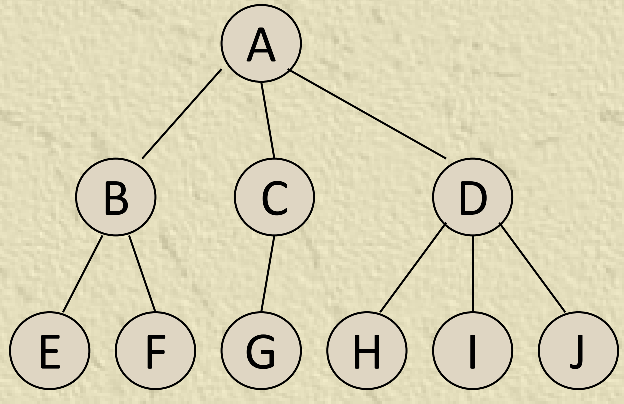


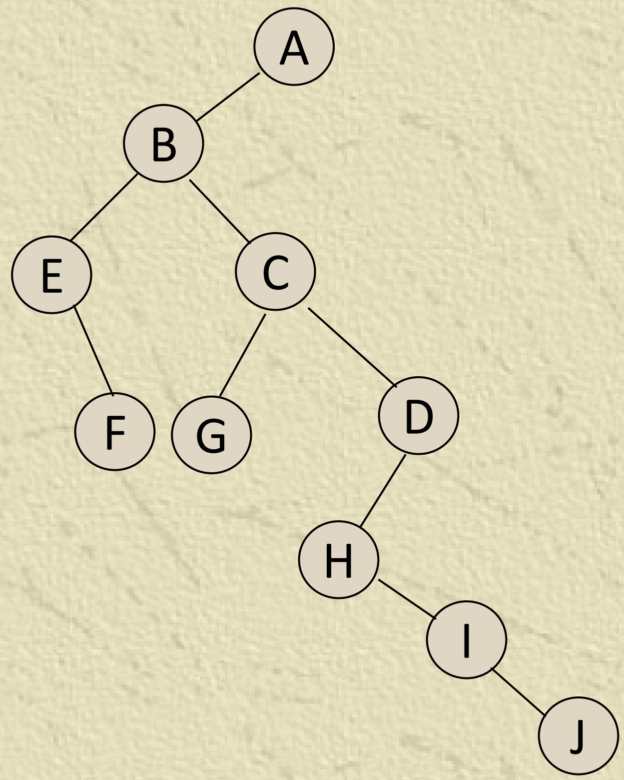
遍历序列：

A， B， C， D， E， F， G， H ， I， J

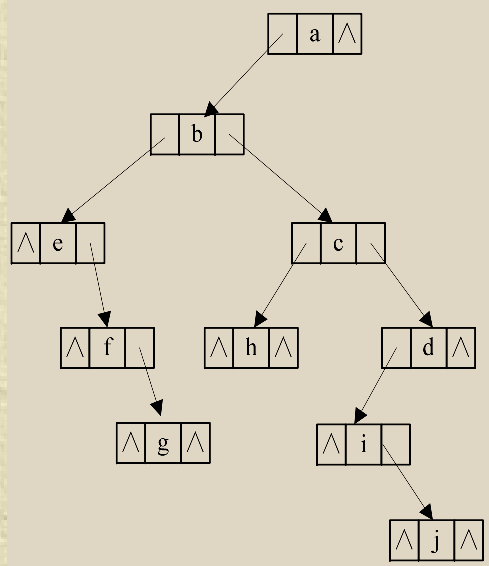
先根遍历一棵树等价于 先 序遍历该树对应的二叉树

后根遍历一棵树等价于 中 序遍历该树对应的二叉树





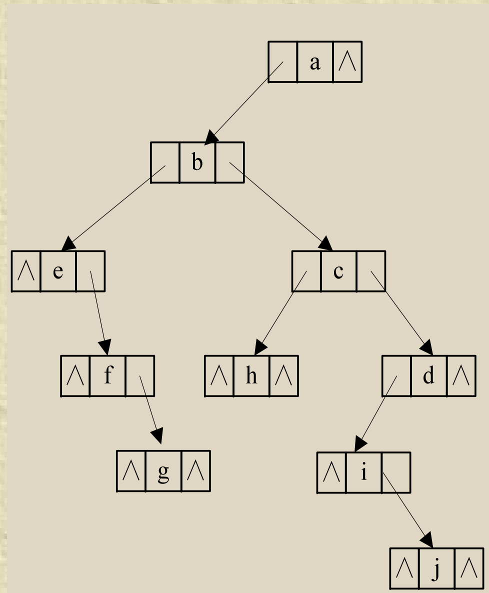
树的先根遍历算法




```
template <class T>
void CSTree<T>::PreOrder(CSNode <T> *p)
{
    if(p==NULL) return;
    cout << p->data;
    for(p=p->firstchild; p; p=p->nextsibling)
        PreOrder(p);
}
```

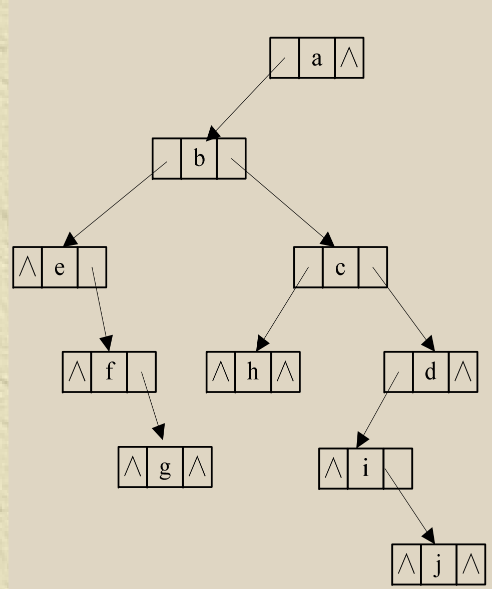
树的先根遍历算法

（改写成不用循环？）



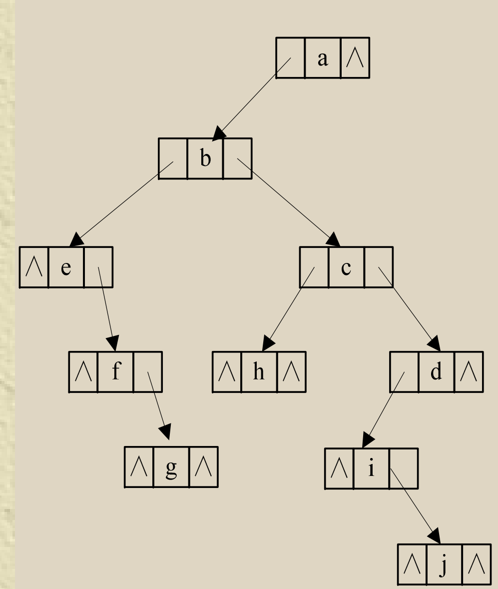

```
template <class T>
int CSTree<T>::PreOrder(CSNode <T> *p)
{
    if(p==NULL) return;
    cout << p->data;
    PreOrder(p->firstchild);
    PreOrder(p->nextsibling);
}
```

计算树的高度



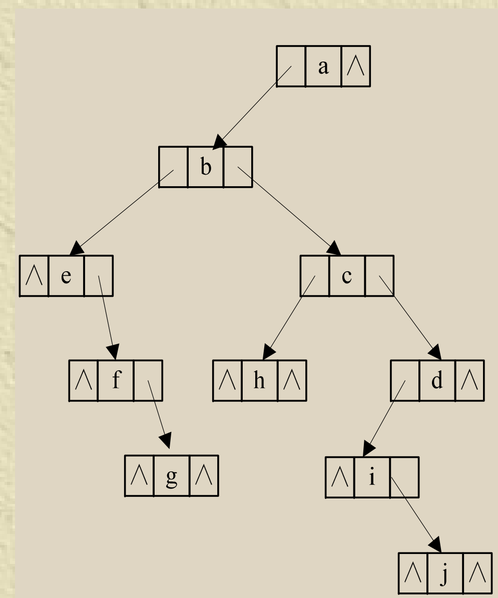

```
template <class T>
int CSTree<T>::Height(CSNode <T> *p)
{
    if(p==NULL)  return 0;
    maxheight=0;
    for(p=p->firstchild; p; p=p->nextsibling){
        int height=Height(p);      //计算各个子树的高度
        if(height>maxheight)  maxheight=height;
    }
    return maxheight+1;
}
```

计算树中所有结点的度




```
template <class T>
void CSTree<T>::Degree(CSNode <T> *p)
{
    if(p==NULL) return;
    p->degree=0;   // 假设树结点中增加一个字段
    for(CSNode<T> *child=p->firstchild; child;
                                child=child->nextsibling)
        p->degree++;
    Degree(p->firstchild);
    Degree(p->nextsibling);
}
```

计算树中所有结点的度 （改写？）

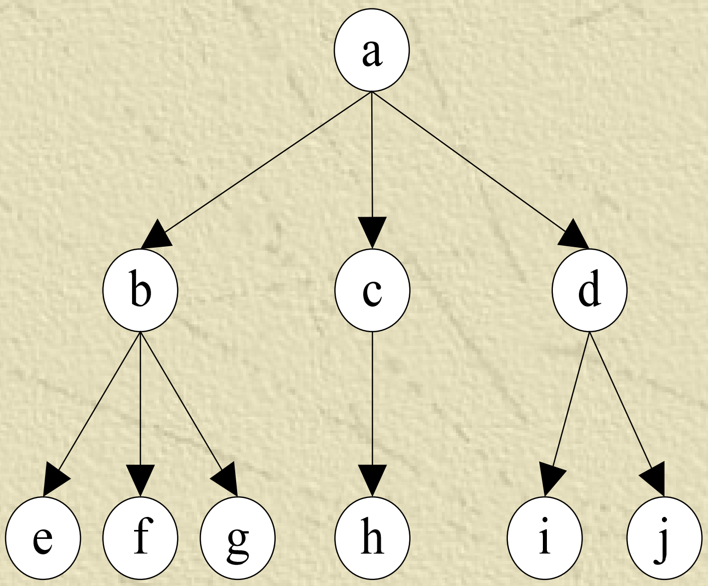



```
template <class T>
void CSTree<T>::Degree(CSNode <T> *p)
{
    if(p==NULL) return;
    p->degree=0;
    for(CSNode<T> *child=p->firstchild; child;
                                child=child->nextsibling)
    {
        p->degree++;
        Degree(child);
    }
}
```

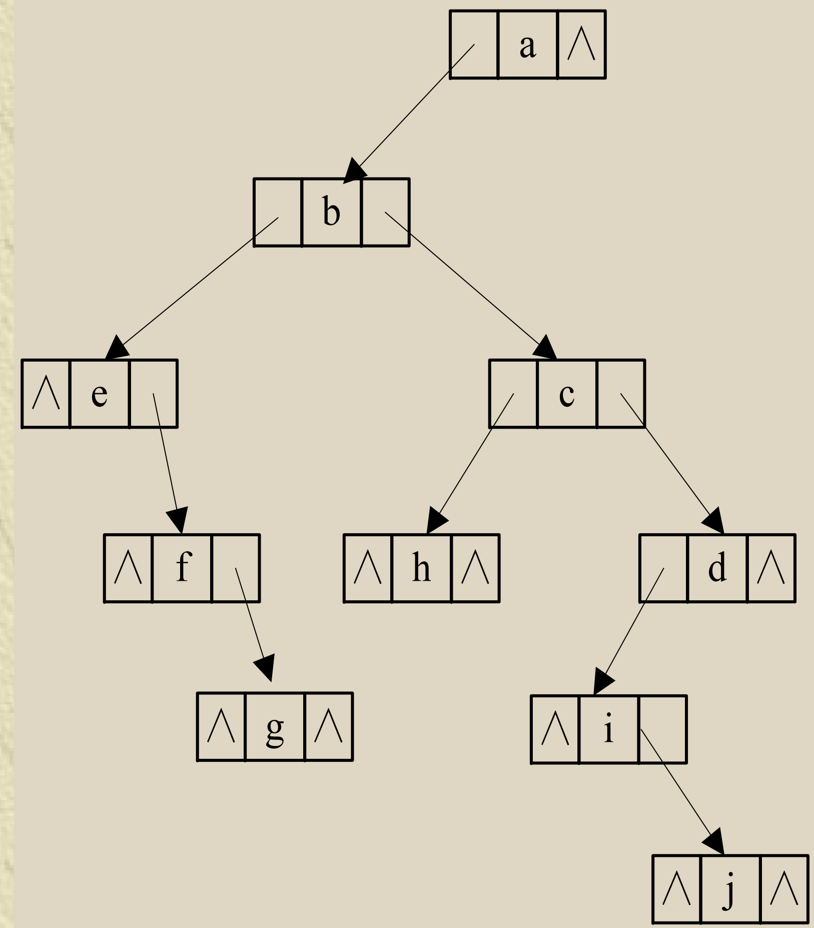
树的构造算法

- 树的建立
  - 目标：给定一棵树的结构信息，建立该树对应的孩子兄弟链表。

- 若给定一棵树的结点的先序序列（空指针标记），能建立一棵对应的树吗？



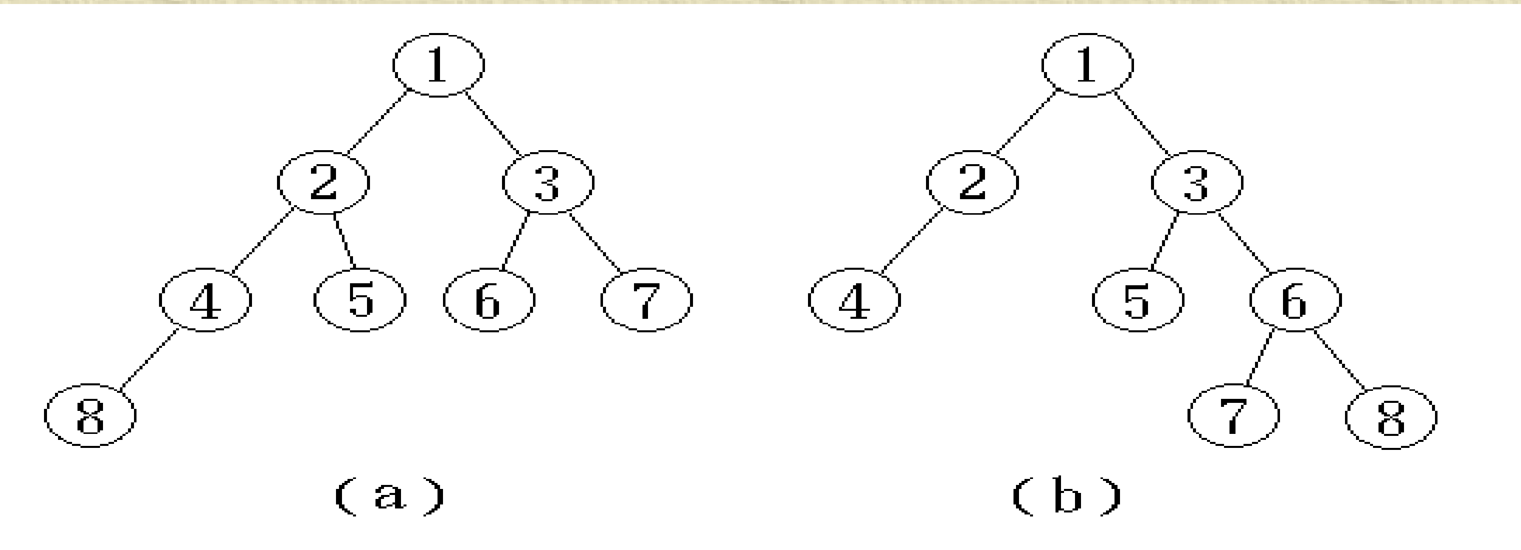
abe*f*gch**di*j*****



```
template <class T>
CSNode <T>* CSTree<T>::CreateByPre( )
{
   e=getchar();
   if(e=='*')  return NULL;
   p=new CSNode <T>;  p->data=e;
   p-> firstchild = CreateByPre();
   p-> nextsibling = CreateByPre();
   return p;
}
```

7. 8 Huffman树及其应用

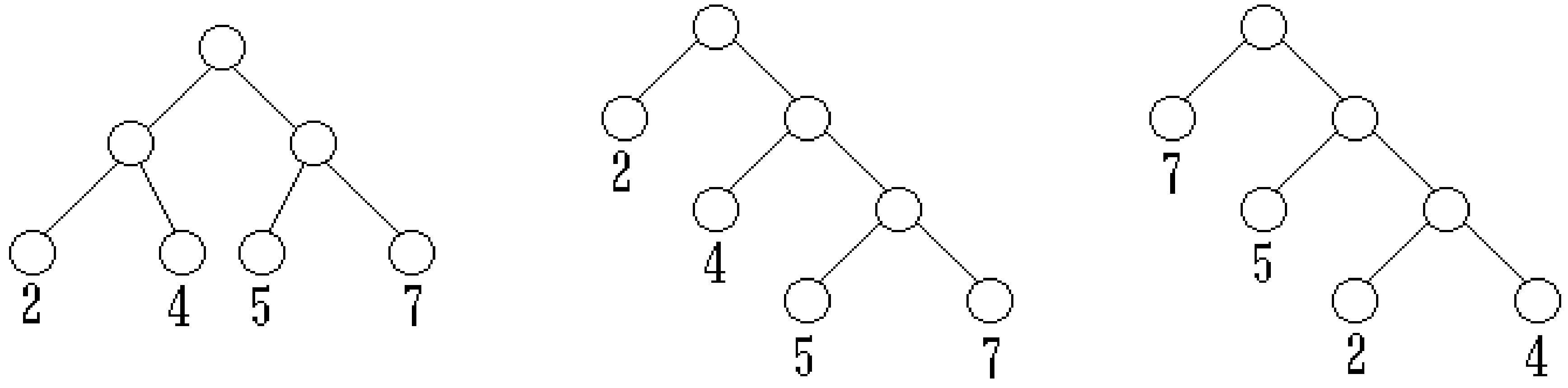
- 最优二叉树（Huffman树）的概念
  - 路径长度：**两个结点之间的路径长度是连接两结点的路径上的分支数。**
  - 树的路径长度：**是树中各结点到根结点的路径长度之和。**



具有不同路径长度的二叉树

◆ 树的带权路径长度：**是树的各叶结点所带的权值与该结点到根的路径长度的乘积的和。**

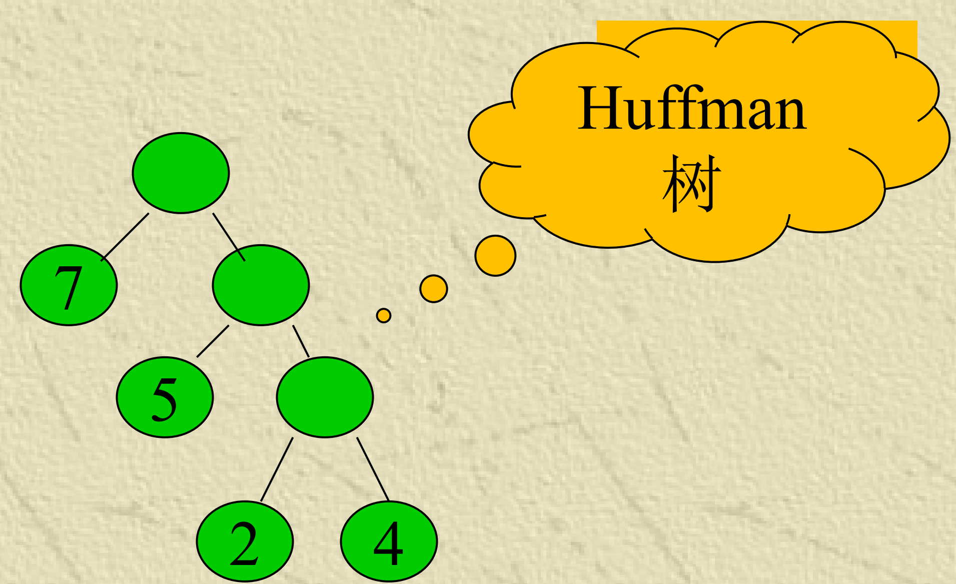
例：给定4个叶子结点，分别对应权值7，5，2，4，可以构造如下三棵二叉树：



(a) $WPL = 36$　　(b) $WPL = 46$　　(c) $WPL = 35$

Huffman树

在权为$w_1,w_2,...,w_n$的n个叶子结点的所有二叉树中，带权路径长度WPL最小的二叉树称为最优二叉树（Huffman树）。



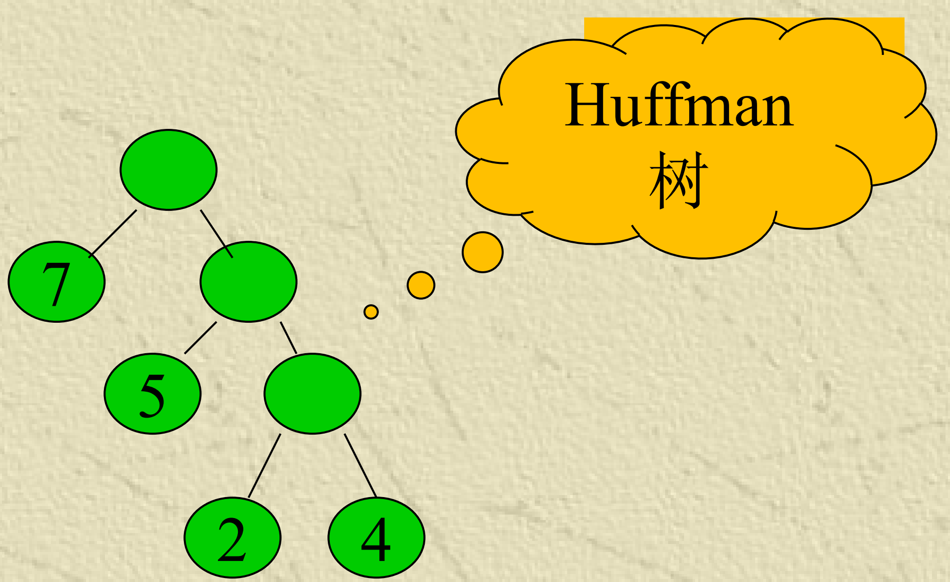
- ✓ Huffman树的特点?

Huffman树的构造方法(Huffman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;
- (2) 在F中选取其根结点的权值为最小的两棵二叉树，将这两棵树合并成一棵新树。
  - 即添加一个新结点，将所选的两棵树分别作为新结点的左、右子树，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) 重复(2)，直至F中只含一棵树为止。

Huffman算法构造的二叉树为什么是最优的?

在权为7,5,2,4的4个叶子结点的所有二叉树中，这棵Huffman树是最优（WPL值最小）。



这背后的算法思想是什么？

贪心(greedy)算法策略

✓ A greedy algorithm arrives at a solution by making a sequence of choices, each of which simply looks the best at the moment according to some criterion, without regard for the choices it will make in the future.

基于贪心策略的Huffman算法

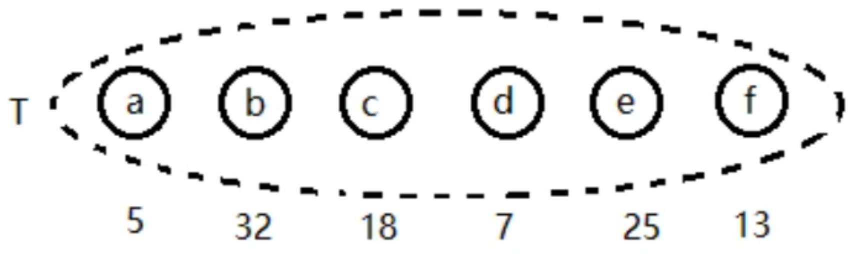


图1 叶子节点

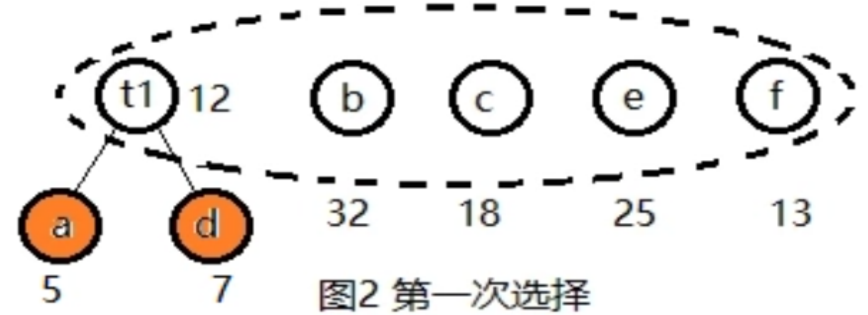


图2 第一次选择

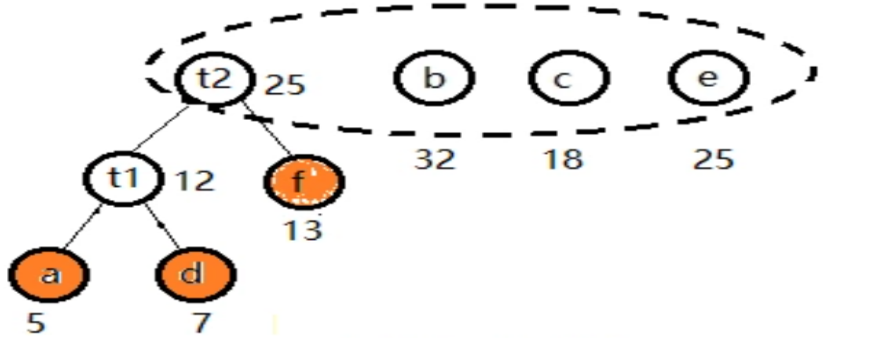


图3 第二次选择

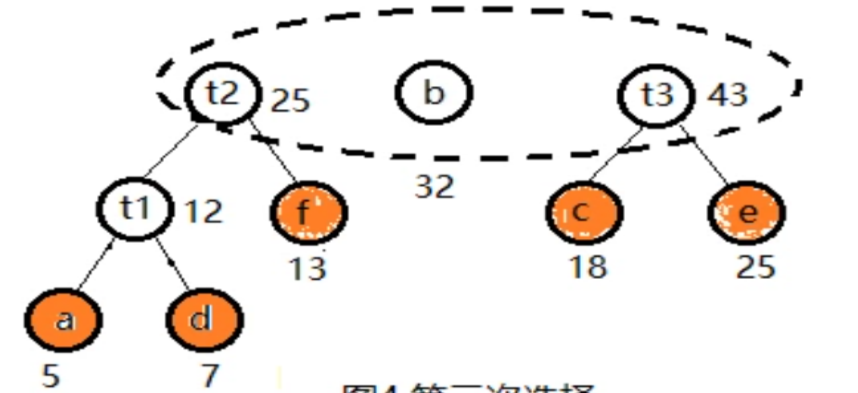


图4 第三次选择

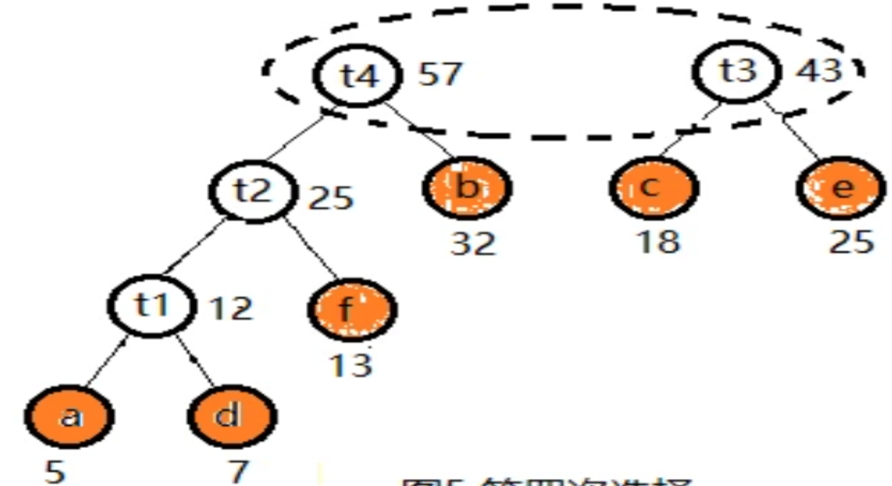


图5 第四次选择

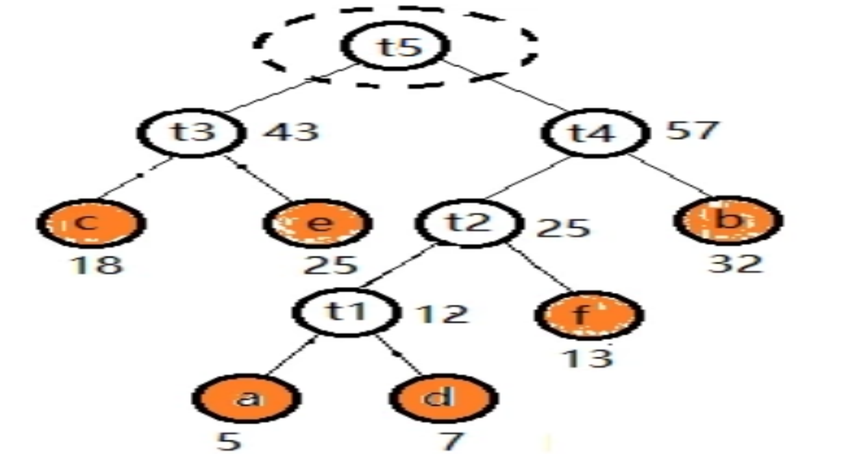


图6 第五次选择

☞ Huffman算法的应用：Huffman编码

- 在远程通讯中，要将待传字符串转换成二进制的0、1序列。
- 最简单的编码方式是采用等长编码

设要传送的字符为：　　ABACCDA

若编码为：A—00

B—01

C—10

D---11

ABACCDA ↓ 00010010101100

若将编码设计为长度不等的二进制编码，即让待传字符串中出现次数较多的字符采用尽可能短的编码，则转换后的二进制编码串便可能缩短。

设要传送的字符为：ABACCDA

若编码为：A—0

B—00

C—1

D---01

ABACCDA

000011010

重码

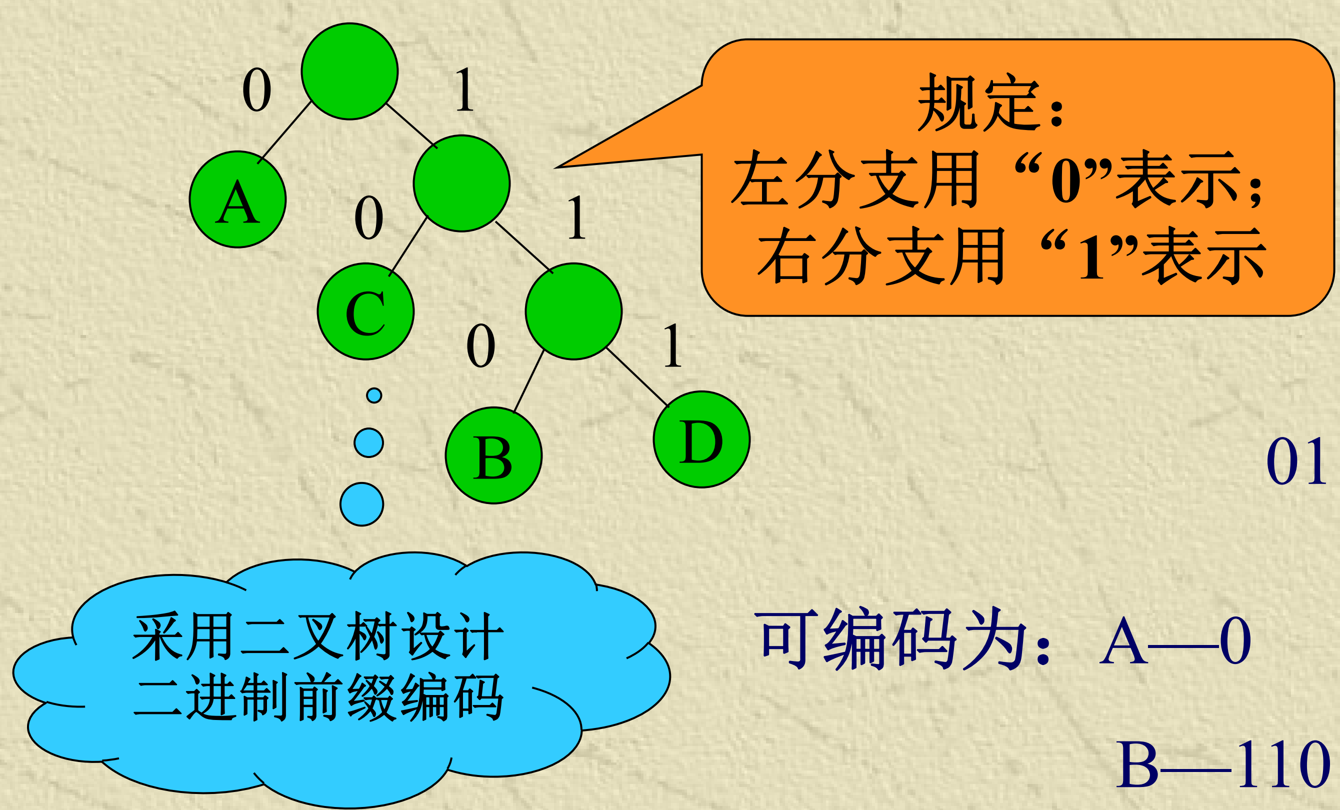
但：0000

AAAA ABA BB

关键：要设计不等长编码，则必须使任一字符的编码都不是另一个字符的编码的前缀，这种编码称之为前缀编码。

☞如何设计前缀编码：

🕮设要传送的字符为： **ABACCDA**



01100101011 10

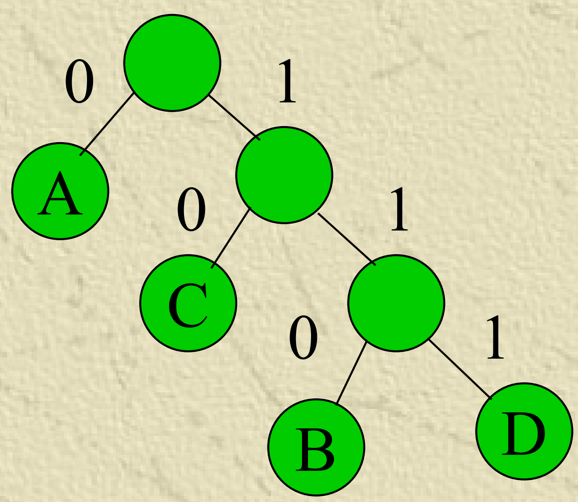
可编码为：A—0

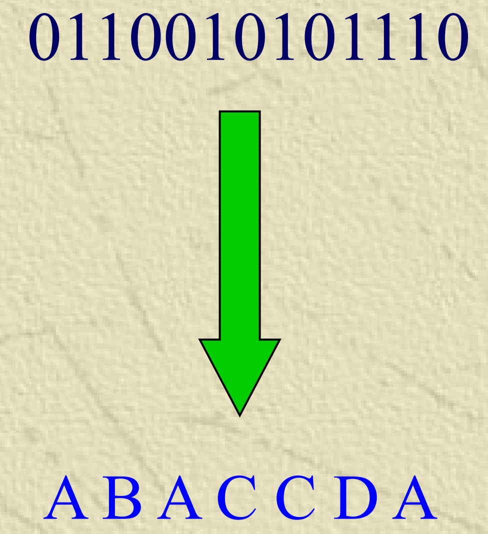
B—110

C—10

D---111

☞ 译码过程：分解接收字符串：遇“0”向左，遇“1”向右；一旦到达叶子结点，则译出一个字符，反复由根出发，直到译码完成。





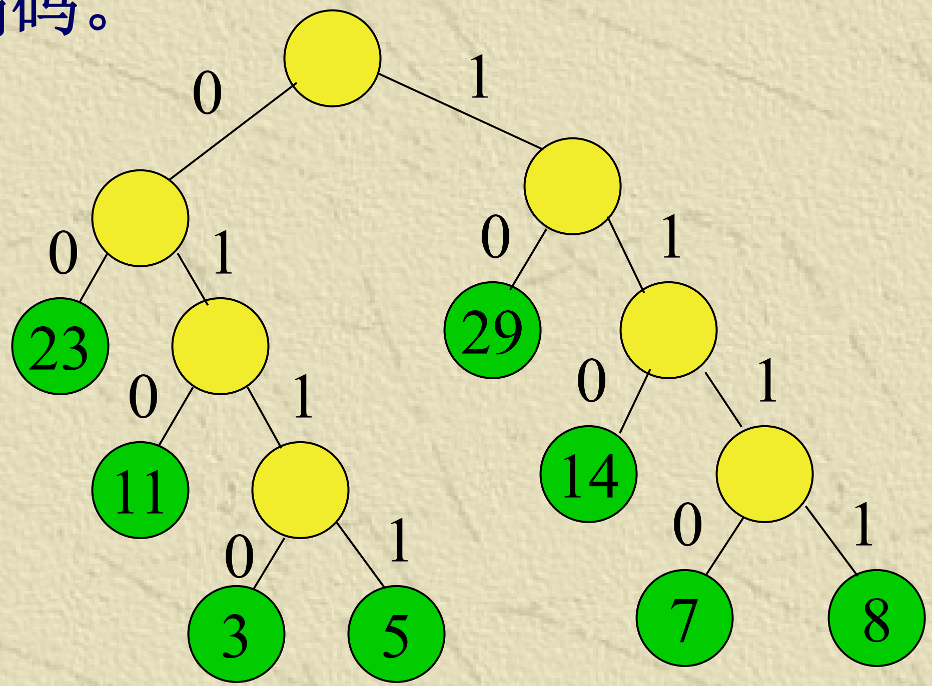
☞如何得到电文总长最短的前缀编码？

- 假设组成电文的字符集合$D=\{ d_1,d_2,\ldots,d_n \}$,每个字符 d_i 在电文中出现的次数为c_i，对应的编码长度为l_i。

- 因此，求电文的总长最短问题可转换为：以n种字符出现的频率作为权值，设计一树棵**Huffman树**。

☞设计示例：

📖 例：已知某系统在通讯时，出现八种字符其概率分别为：**0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03, 0.11,** 试设计**Huffman**编码。



出现频率越大的字符，其**Huffman**编码越短。

Huffman树的构造算法实现：

- 当给定n个叶子结点构造Huffman树时，共需要进行n-1次合并，每次合并都要产生一个新结点，合并过程中共产生n-1个新结点，因此，Huffman树中总的结点个数为2n-1个。
- 如何设计Huffman树的存储结构?
    - 将Huffman树中的2n-1个结点可以存储在一个大小为2n-1的数组中。

结点结构定义如下：

```
struct HuffmanNode
{   char data;                  // 待编码的符号
    double weight;              // 符号出现的频率
    int parent, lchild, rchild;
};
```

- Huffman树的结点结构中，增加了一个parent指针域，其作用是：
  - 区分结点与根结点
  - 从任一结点出发，可直接到达其双亲结点。
- 在存储Huffman树结点的长度为2n-1的数组中，前n个分量中存放的是叶子结点。

Huffman树的类定义如下：

```
class HuffmanTree{
    vector<HuffmanNode> nodes;
    int n;                    // 叶子结点数
public:
    HuffmanTree(vector<HuffmanNode> &leafs);
    ~HuffmanTree();
    …………
};
```

• 构造Huffman树算法的流程：

- 初始化n个叶子结点；
- 初始化n-1个非叶子结点；
- 执行n-1次合并动作：
  - 选取权值最小的两个根结点；
  - 将两棵以它们为根的树进行合并，使其成为新结点的左、右孩子，同时修改这两个结点的parent域，并给新结点赋权值.

- 例：假设某信息系统中有**5**种符号，分别记作***A***、***B***、***C***、***D***、***E***，它们各自出现的统计频率是**6%**、**14%**、**53%**、**15%**、**12%**，试构造**huffman**编码。

Huffman树初始化时的存储结构

	0	1	2	3	4	5	6	7	8
data域	**'A'**	**'B'**	**'C'**	**'D'**	**'E'**				
weight域	6%	14%	53%	15%	12%				
parent域	**-1**	**-1**	**-1**	**-1**	**-1**				
lchild域	**-1**	**-1**	**-1**	**-1**	**-1**				
rchild域	**-1**	**-1**	**-1**	**-1**	**-1**				

Huffman树初始化时的存储结构

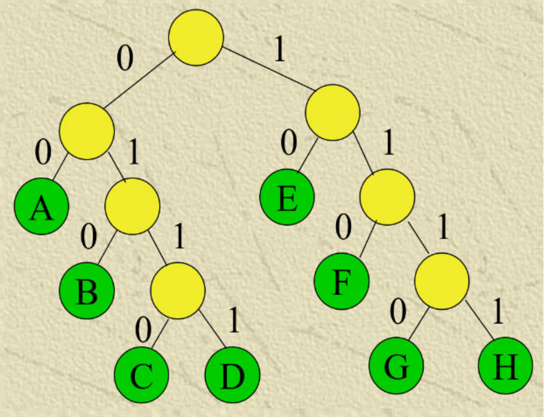
	0	1	2	3	4	5	6	7	8
data域	**'A'**	**'B'**	**'C'**	**'D'**	**'E'**				
weight域	6%	14%	53%	15%	12%				
parent域	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**
lchild域	**-1**	**-1**	**-1**	**-1**	**-1**				
rchild域	**-1**	**-1**	**-1**	**-1**	**-1**				

Huffman树建完时的存储结构

	0	1	2	3	4	5	6	7	8
data	'A'	'B'	'C'	'D'	'E'				
weight	6%	14%	53%	15%	12%	18%	29%	47%	100%
parent	5	6	8	6	5	7	7	8	-1
lchild	-1	-1	-1	-1	-1	0	1	5	7
rchild	-1	-1	-1	-1	-1	4	3	6	2

```
HuffmanTree::HuffmanTree(vector<HuffmanNode> &leafs)
{   n=leafs.size();      // 叶子结点数
    nodes.resize(2*n-1);    // 为分支结点预留向量空间;
   for( i=0; i<n; ++i ){
       nodes[i].weight= leafs[i].weight;
       nodes[i].data= leafs[i].data;
       nodes[i].parent = -1;
       nodes[i].lchild = -1;   nodes[i]. rchild = -1;
   }
  for( i=n; i<2*n-1; ++i )    nodes[i].parent = -1;
  for(i=n; i<2*n-1; ++i) {  // n-1次合并根结点
     SelectSmall( least, less, i);
     nodes[least].parent = nodes[less].parent = i;
     nodes [i].lchild=least;  nodes[i].rchild=less;
     nodes [i].weight=nodes[least].weight+nodes[less].weight;
  }
}
```

Huffman树的编码算法:

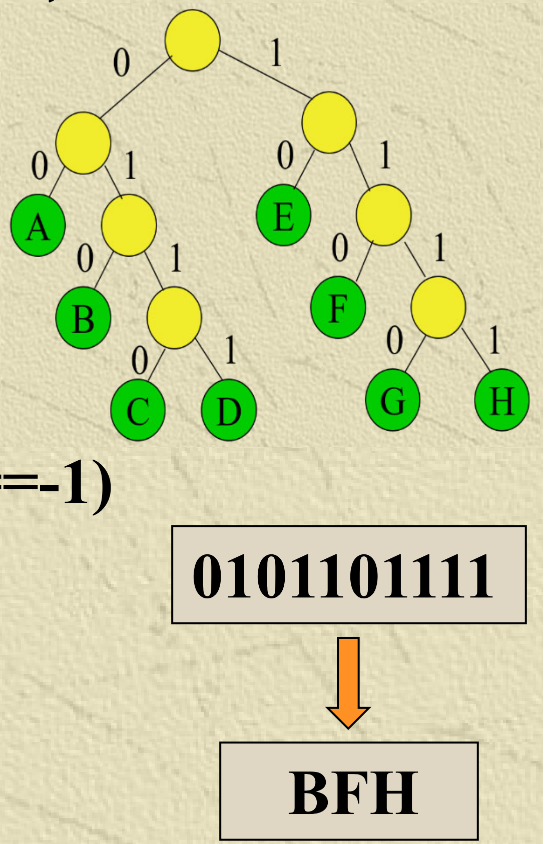


G: 1110

```
// 从叶子到根逆向求第i个字符的Huffman编码
vector<int> HuffmanTree::Encode( int i )
{   vector<char> code;  // 第i个符号的编码向量
    parent=nodes[i].parent;
    while(parent != -1)  // 只有根结点的parent域为-1
    {   if( nodes[parent].lchild==i )
              code.insert( code.begin(), '0');
        else
              code.insert( code.begin(), '1');
        i=parent;  parent= nodes[parent].parent; // 沿父指针上溯
    }
    return code;
}
```

Huffman树的译码算法:

```
string HuffmanTree::Decode( vector<int> &source )
{   string target="";      // 译码的目标：原信息符号串
    root=tree.size()-1;    // 根结点下标
    p=root;                // 将当前结点设为根结点
    for(i=0; i<source.size(); i++)
    {   if(source[i]==0)  p=nodes [p].lchild;
        else   p=data[p].rchild;
        if( nodes[p].lchild==-1 && nodes [p].rchild==-1)
        {   target= target+nodes[p].data;
            p=root;      // 退回到根结点
        }
    }
    return target;
}
```



7.9 等价类问题 (并查集)

- 等价关系的定义：
  - 如果集合S中的关系是自反的、对称的和传递的，则称这种关系是一个等价关系。
- 等价类：若R是集合S中的等价关系，x是S中的任意元素，所有与x存在等价关系的元素构成的集合，称为由x生成的一个R等价类。
- 等价类划分：等价关系R可产生集合S的唯一划分，即可以按R将S划分为若干不相交的子集S_1, S_2, ……，它们的并集是S，则这些子集S_i称为S的R等价类。

如何实现等价类划分？

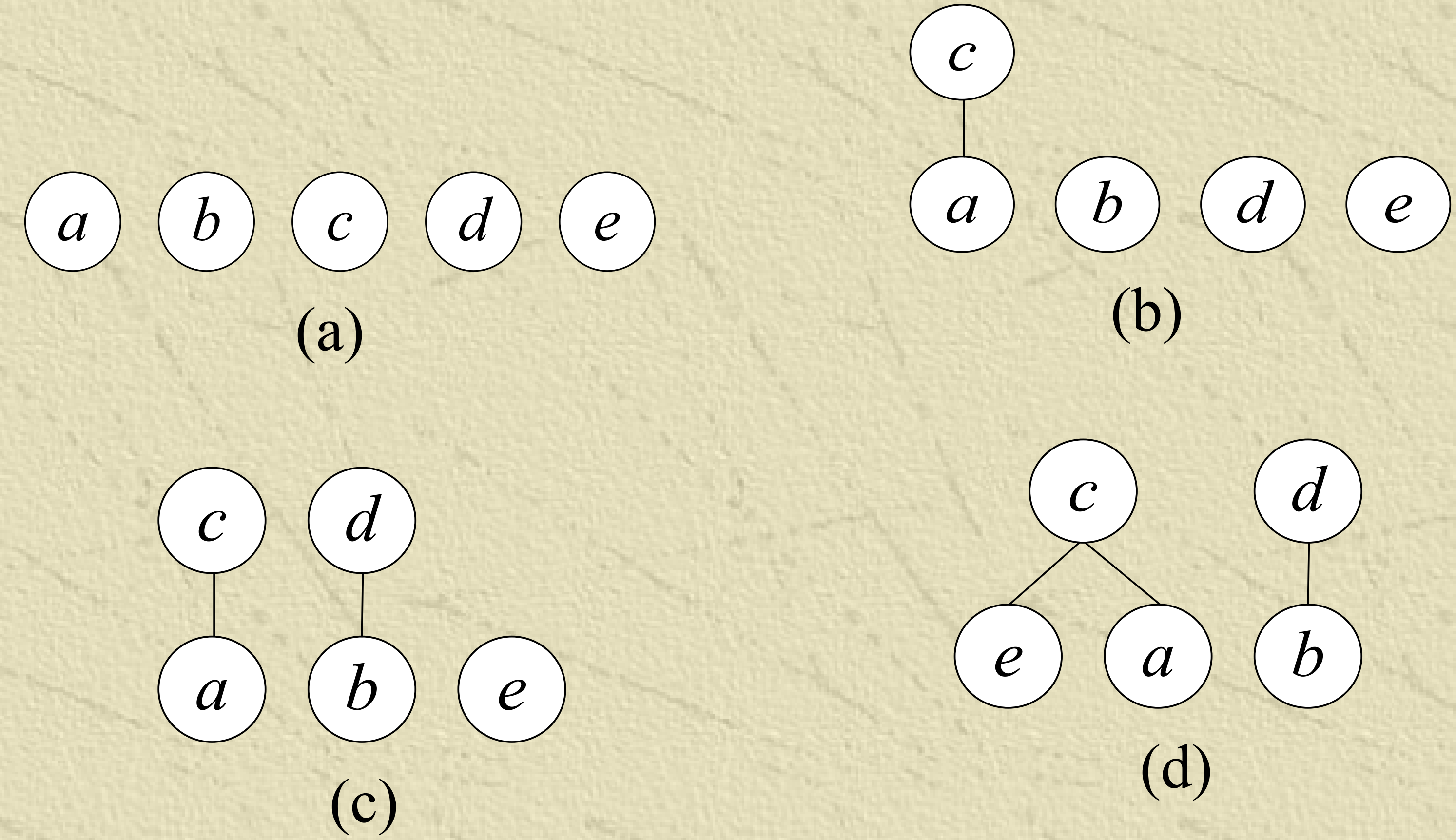
- 假定集合S有n个元素，m个形如(x, y) $(x, y \in S)$的等价偶对确定了等价关系。如何求S的划分？

- 构造等价类的算法如下：
  ① 令S中每个元素各自形成一个只含有单个成员的子集，记作$S_1, S_2, \ldots\ldots S_n$。

  ② 考察偶对(x, y)，判断x和y所属子集，假设$x \in S_i, y \in S_j$。若$S_i = S_j$，则x和y已属同一个等价类，无需操作；若$S_i \neq S_j$，则将S_i并入S_j。

  ③重复执行步骤②，当所有偶对都处理完时，所有剩余子集即为S的R等价类。

- 例：设集合$S=\{a,b,c,d,e\}$，等价偶对有$(a,c),(b,d),(c,e)$，则构造等价类的过程如下：

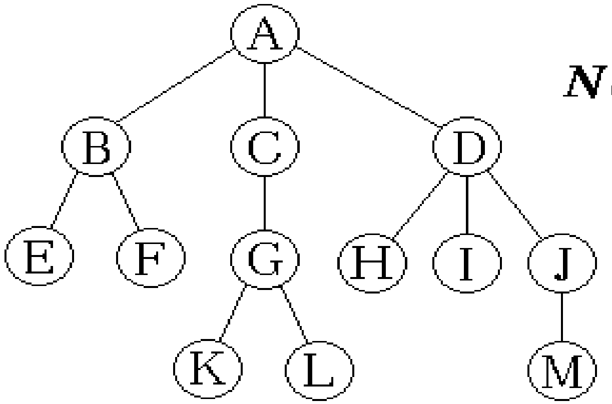


集合如何表示？

- 数组实现
- 树实现

树的存储结构？

- 每棵树对应一个子集；
- 整个集合是一个森林；
- 为便于操作的实现，采用树的双亲表示法作为存储结构。
- 约定根结点的下标作为子集的编号；



（a）树

NodeList	1	2	3	4	5	6	7	8	9	10	11	12	13
data	A	B	E	F	C	G	K	L	D	H	I	J	M
parent	0	1	2	2	1	5	6	6	1	9	9	9	12

（b）双亲表示数组

双亲表示法的树结点结构定义：

```
template <class T>
struct PTNode
{
    T data;
    int parent;  // 双亲指针域
};
```

等价类的类模板定义

```
template<class T>
class MFSet
{   vector<PTNode<T> > data;
 public:
    MFSet(vector<T> &ds);
    ~MFSet( );
    int Find(int i);
    void Merge(int i, int j);
};
```

构造函数：

```
MFSet::MFSet(vector<T> &ds)
{
    data.resize( ds.size() );
    for(i=0; i<ds.size(); i++)
    {
        data[i].data=ds[i];
        data[i].parent=-1;
    }
}
```

设置-1表示根结点

返回i所在子集的编号

函数Find的实现：

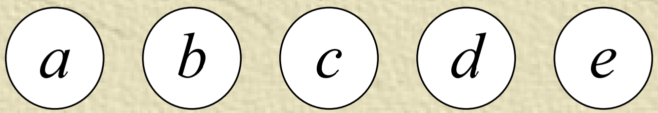
```
template<class T>
int MFSet<T>::Find( int i )  // i表示元素序号
{
   while(data[i].parent>=0)   //找到根结点
        i = data[i].parent;
   return i;
}
```

函数Merge的实现：

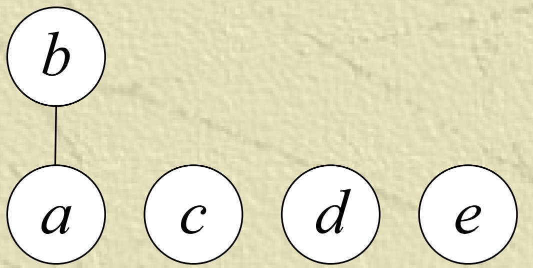
i, j表示根结点下标

```
template<class T>
void MFSet<T>::Merge( int i, int j )
{
    data[i].parent = j;
}
```

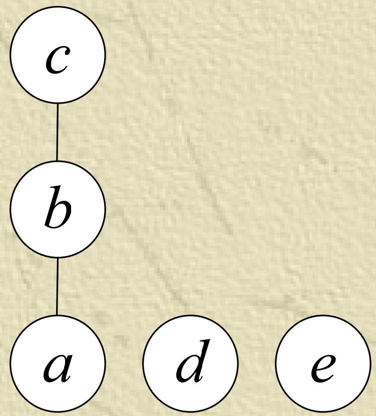
- 例：设集合$S=\{a,b,c,d,e\}$，等价偶对有$(a,b),(b,c),(c,d)$，则构造等价类的过程如下：



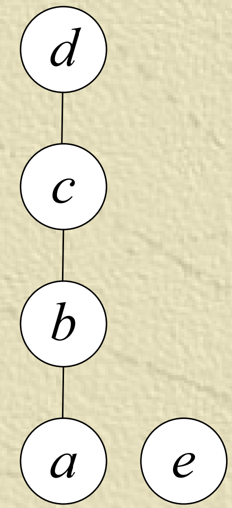
(a)



(b)



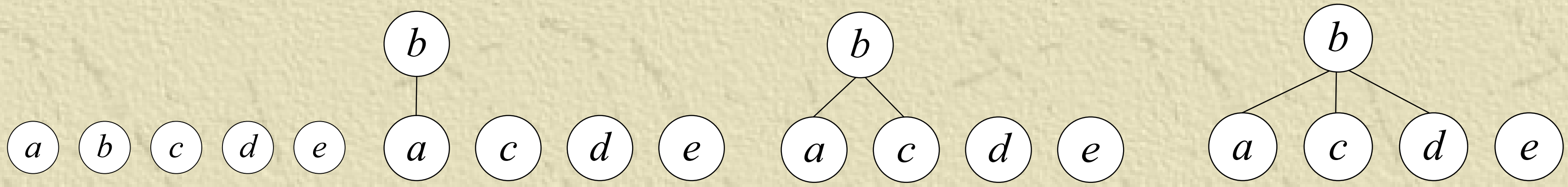
(c)



(d)

改进方法：

- 在合并两个子集时，将元素较少的树的根结点的parent域值置为元素较多的树的根结点的下标。



(a) (b) (c) (d)

- 为能较容易地读取树中的结点个数，将根结点的parent域值设为树中结点个数的负值。

函数Merge的实现：

```
template<class T>
void MFSet<T>::Merge( int i, int j )
{  if(data[i].parent < data[j].parent )
   {  data[i].parent += data[j].parent;
      data[j].parent = i;
   }
   else
   {   data[j].parent += data[i].parent;
       data[i].parent = j;
    }
}
```

本章小结

本章的主要学习要点如下：

（1）掌握树的相关概念和基本性质；

（2）掌握二叉树的相关概念和基本性质；

（3）重点掌握二叉树的存储结构；

（4）重点掌握二叉树遍历算法和各种常用算法；

（5）掌握线索二叉树的概念、结构，及线索指针的应用算法；

（6）掌握树的存储结构；

（7）掌握Huffman树的概念、构造方法和Huffman编码的产生方法；

（8）掌握等价类概念和构造方法。