第五章 树与二叉树

树的定义和基本术语

二叉树

遍历二叉树和线索二叉树

树与森林

树与等价问题

Huffman树及其应用

回溯法与树 的遍历

树 的计数

6.1 树的概念

树(tree)是一个具有相同特性的n(n>=0)个结点的有限集。

在一棵非空树中，

(1) 有且仅有 一个的称为根(root)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$T_1, T_2, \cdots, T_m$，且每一棵子集本身又是一棵符合本定义的树，并称为根root的子树(SubTree)。

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；

若D仅含一个 数据元素，则R为 空集 ，否者 R={H},

H为如下二元关系 ：

(1) 在D中存在唯一称为根的数据元素root, 它在关系H下无前驱；

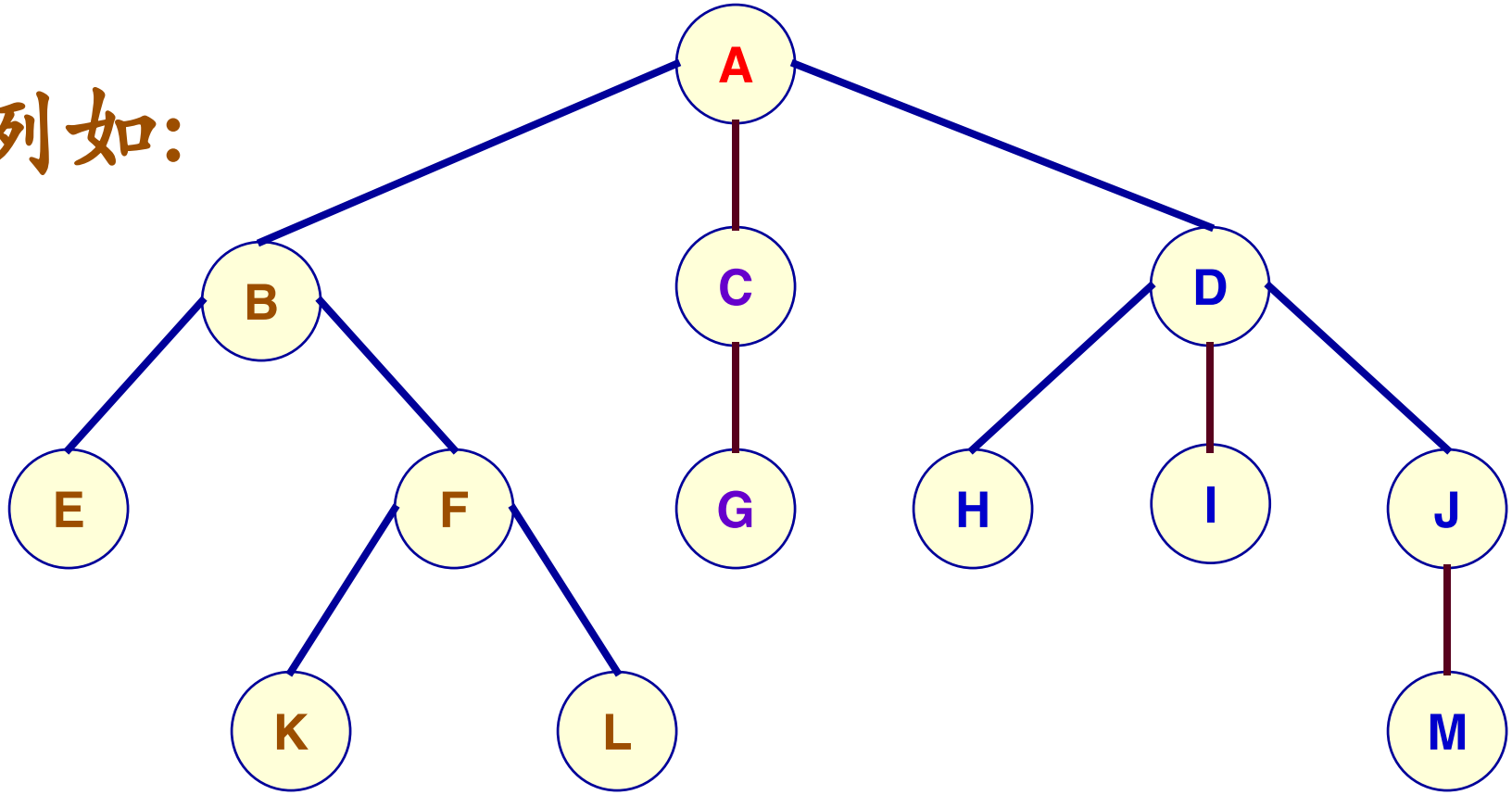
(2) 若 D-{root}<>Φ ，则存在 D-{root}的一个划分$D_1, D_2, \cdots, D_m$，有$D_j \cap D_k=\Phi$，且对任意的 i(1<=i<=m)，唯一存在数据元素 $x_i \in D_i$，有<root, x_i> $\in$ H;

(3) 对应于D-{root}的划分，H-{<root, x_1>,$\cdots$, <root, x_m>}有 唯一的划分$H_1, H_2, \cdots, H_m$，对任意的 j<>k(1<=j, k<=m)，有$H_j \cap H_k=\Phi$，且对任意的 i(1<=i<=m)， H_i是D_i上 的二元关系，(D_i,{H_i})是一棵符合本定义的树 ，称为 根 root的子树。

基本操作：

}ADT Tree

例如:



树的基本术语

- 结点：数据元素+若干指向子树的分支
- 结点的度：结点拥有 的子树个数
- 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

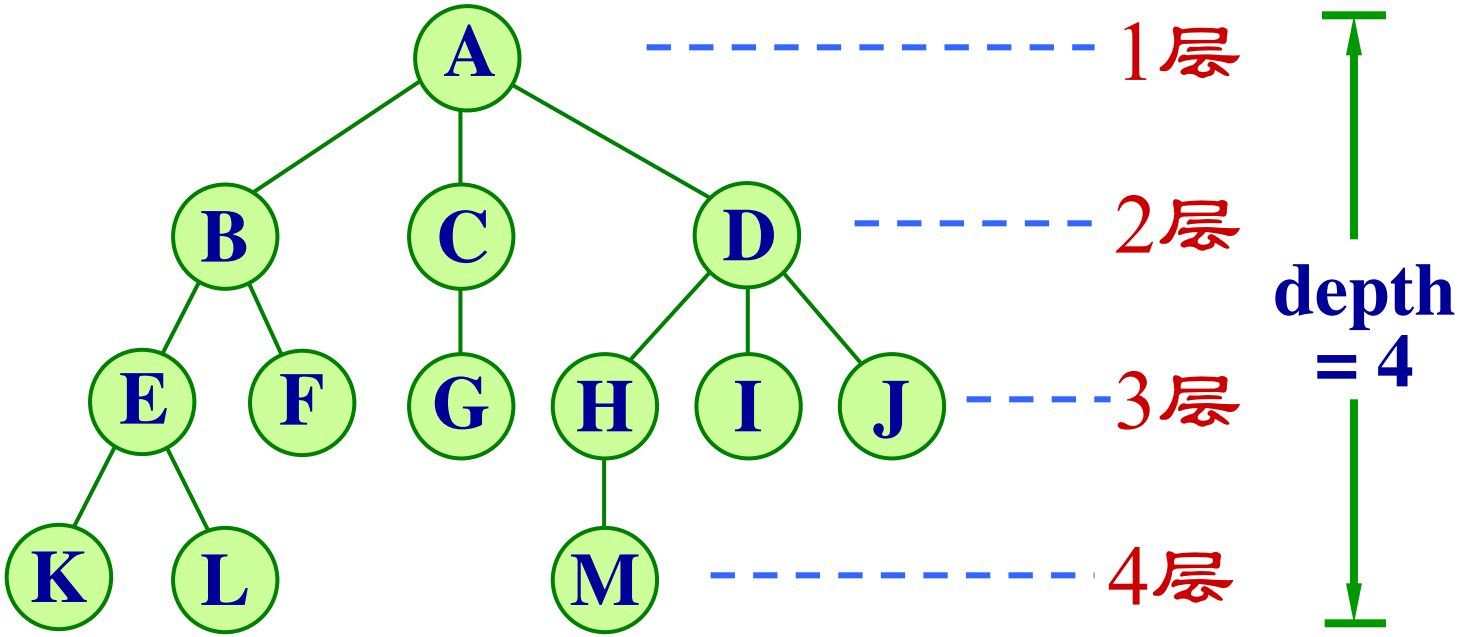
- 从根到结点的路径：由根到该结点所经分支和结点构成

- 孩子：若结点的子树非空，子树的根即为该结点的子女。
- 双亲：若结点有子女，该结点是子女双亲。
- 兄弟：同一结点的子女互称为兄弟。
- 祖先：结点的祖先是从根到该结点所经分支上的所有结点。
- 子孙：以某结点为根的子树种任一结点都是该结点的子孙。

- 结点的层次：规定根结点为第一层，第*l*层的结点的子树根结点的层次为*l+1*。
- 树的深度：树中结点的最大层次数称为树的深度。

- 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：树中结点的各棵子树 T0, T1, …是有次序的，即为有序树。
- 无序树：


A
B
C
D
E
F
G
H
I
J
K
L
M
1层
2层
3层
4层
depth
= 4

- **森林**：森林是m（m≥0）棵互不相交的树的集合。

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

基本操作：

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

线性结构

第一个数据元素
(无前驱)

最后一个数据元素
(无后继)

其它数据元素
(一个前驱、一个后继)

树型结构

根结点
(无前驱)

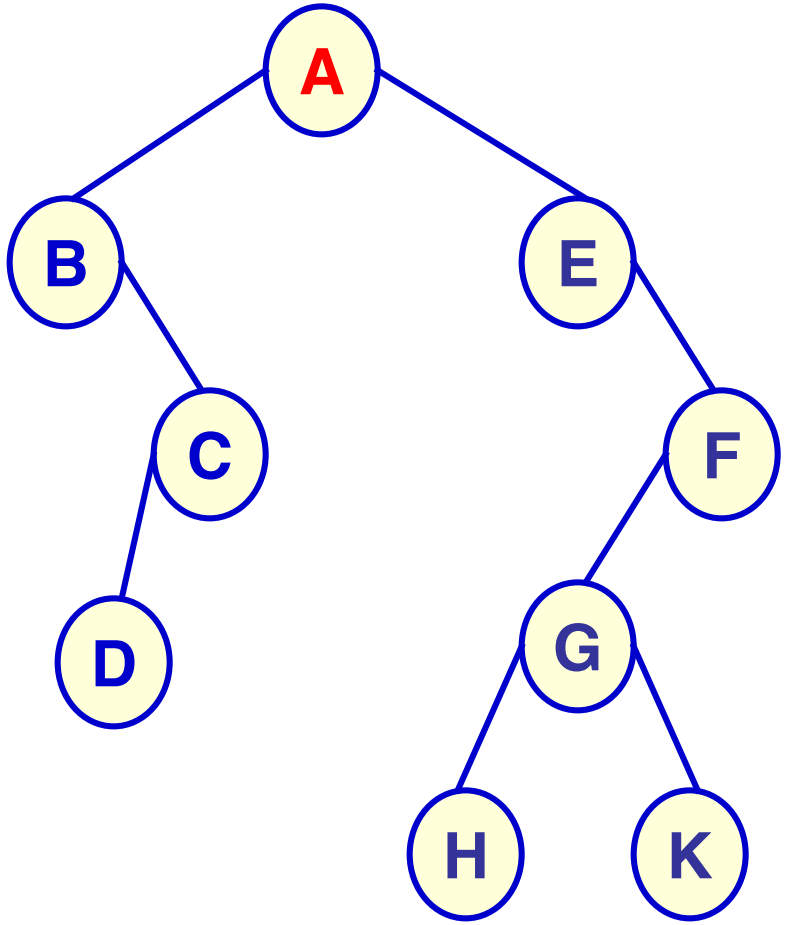
多个叶子结点
(无后继)

其它数据元素
(一个前驱、多个后继)

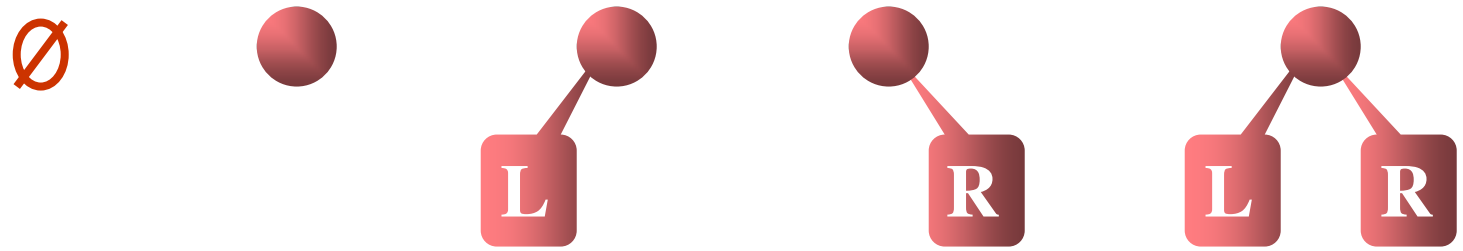
6.2 二叉树 (Binary Tree)

- 二叉树的定义

  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、互不相交的二叉树组成。



二叉树的五种不同形态



二叉树的性质

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

- 性质2　深度为 k 的二叉树最少有 k 个结点，最多有 2^k-1 个结点。($k \geqslant 1$)

- 性质3　对任何一棵二叉树，如果其叶结点有 n_0 个，度为 2 的非叶结点有 n_2 个，则有

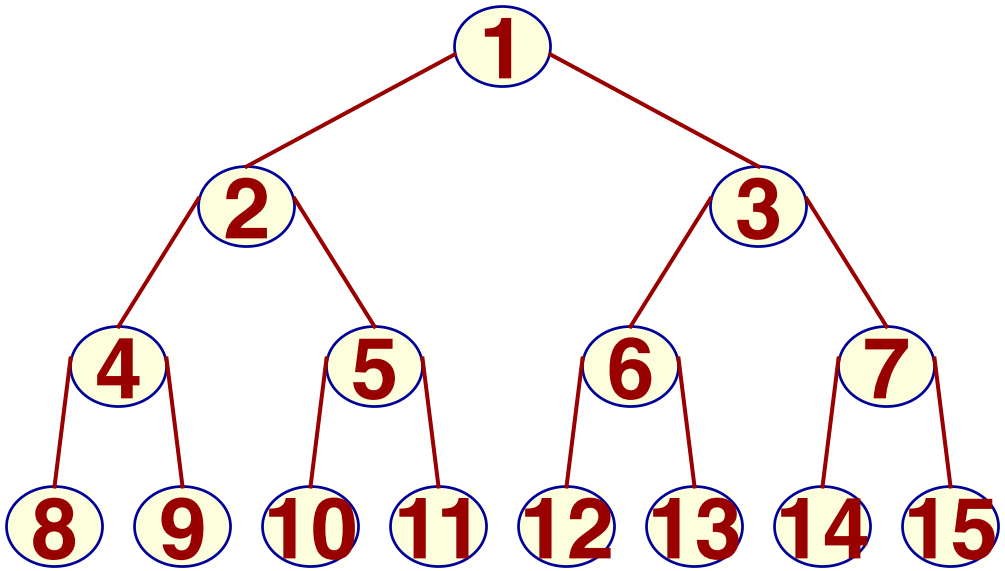
$$n_0 = n_2 + 1$$

- **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

特点:

- 没有度为1的结点，除最后一层为叶子结点外，其余结点都为度为2的结点
- 每层上的结点数都是最大的结点数

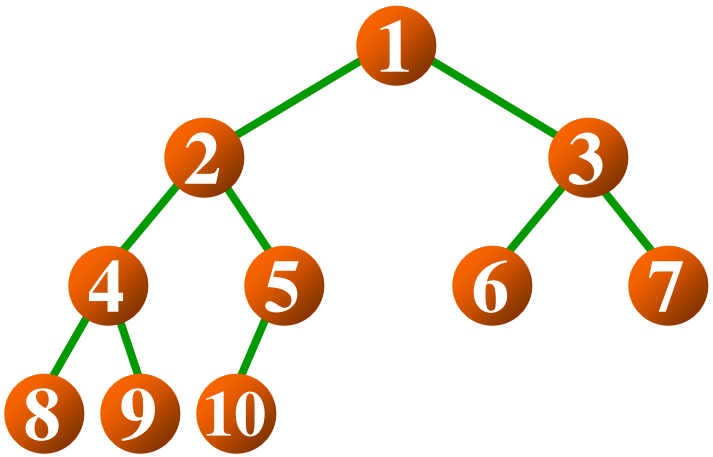


完全二叉树 (Complete Binary Tree)

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：

- 叶子结点只可能出现在层次最大的两层上。此外，除最后一层外，其它层次都是满的。
- 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必定为*l*或者*l*+1。



- **性质4**　具有 n ($n \geq 0$) 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$

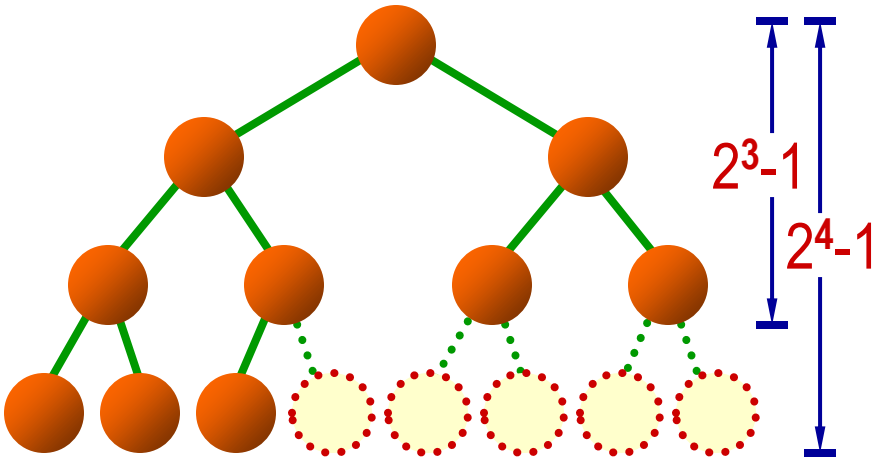
设完全二叉树的深度为k，则根据第二条性质得

$2^{k-1}-1 < n \leq 2^k-1$　和　$2^{k-1} \leq n < 2^k$

即　$k-1 \leq \log_2 n < k$

因为 k 只能是整数，

因此，$k = \lfloor \log_2 n \rfloor + 1$



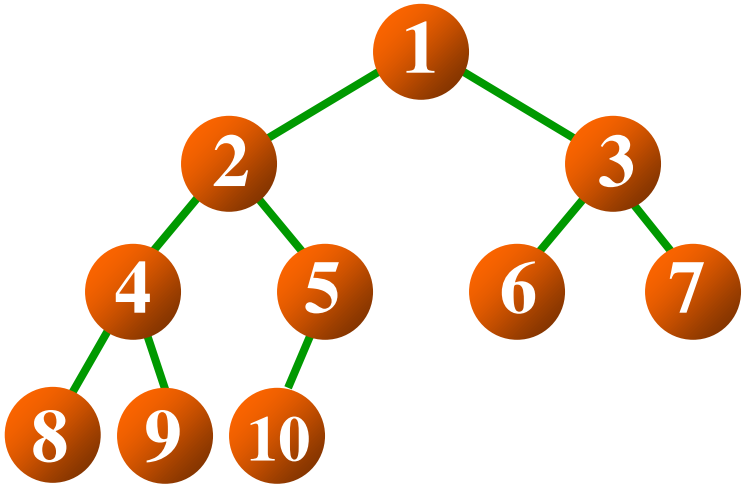
- 性质5 若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点：

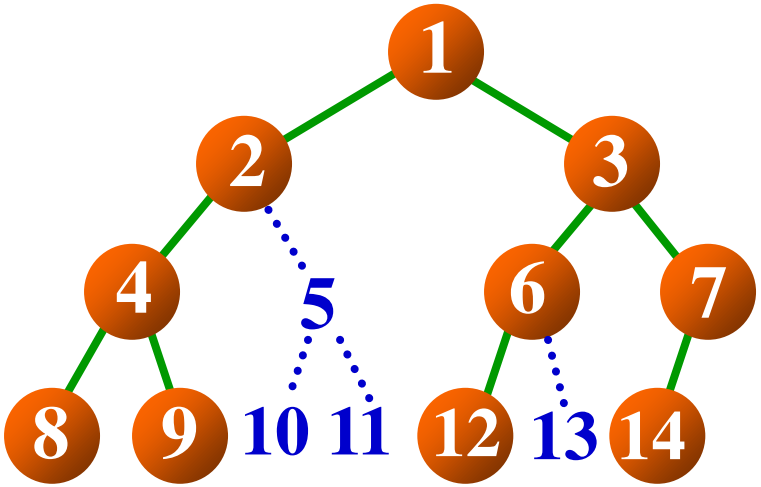
  1. 若 i=1，则该结点是二叉树的根，无双亲；否则，编号为 $\lfloor i/2 \rfloor$ 的结点为其双亲结点；
  2. 若 2i>n，则该结点无左孩子，否则，编号为 2i 的结点为其左孩子结点；
  3. 若 2i+1>n，则该结点无右孩子结点，否则，编号为2i+1 的结点为其右孩子结点。

二叉树的存储结构

一、 二叉树的顺序存储结构

```
#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;
```



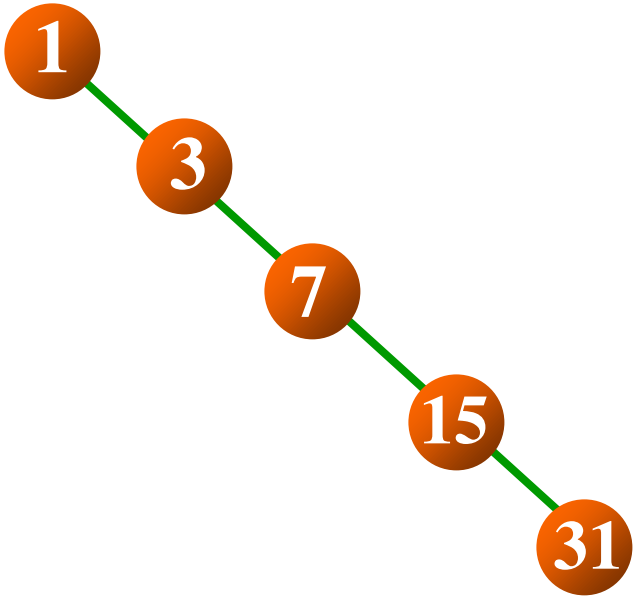


完全二叉树
的顺序表示

一般二叉树
的顺序表示

极端情形: 只有右单支的二叉树

(需要2^k-1的存储)



这种顺序存储结构仅适用于完全二叉树

二、二叉树的链式存储表示

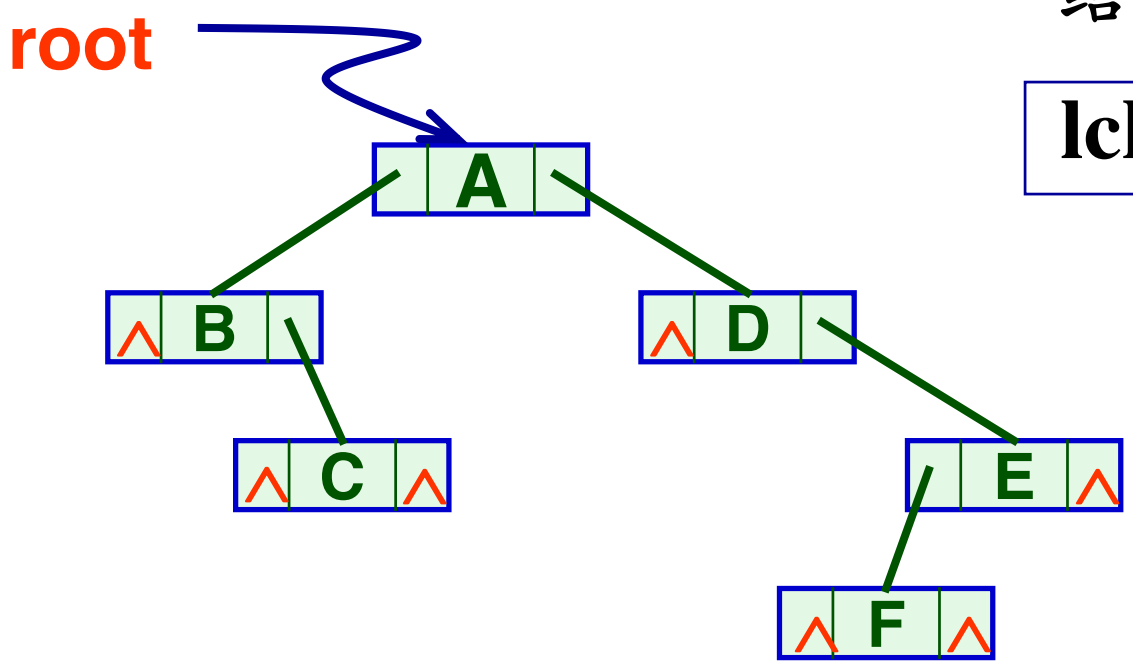
1. 二叉链表

2. 三叉链表

3. 双亲链表

4. 线索链表

1. 二叉链表



结点结构:

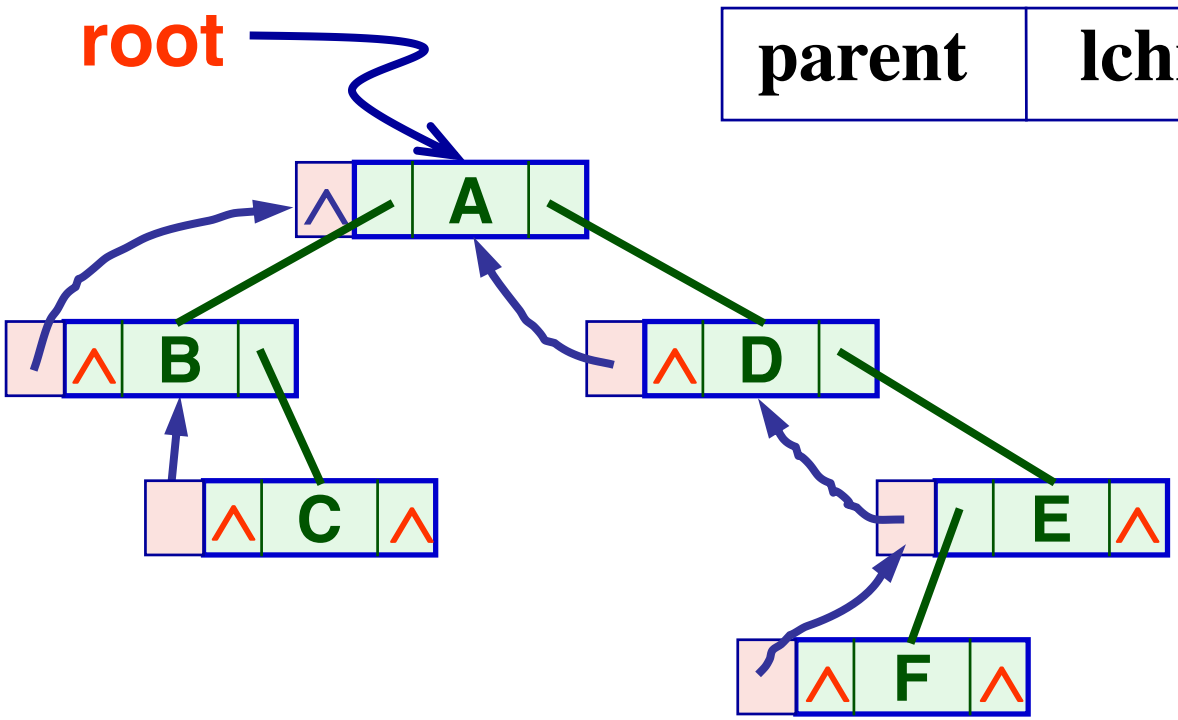
lchild	data	rchild

```
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




```
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;
```

2．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 { // 二叉树结构
    BPTNode  nodes[MAX_TREE_SIZE];
    int num_node; //结点数目
    int  root;  //根结点的位置
} BPTNode;
```

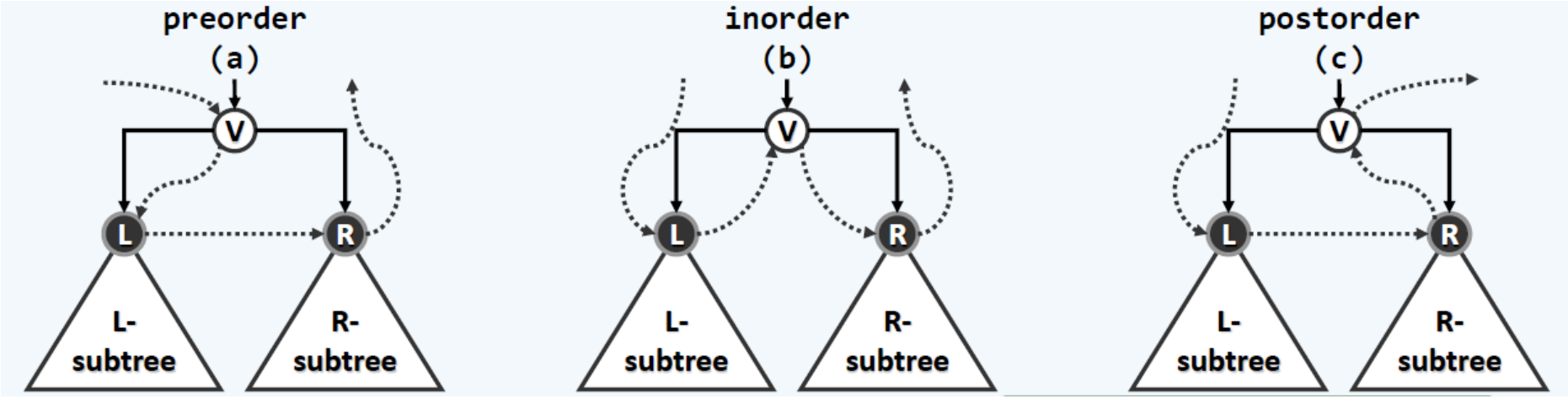
6.3 二叉树遍历

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。

由于二叉树节点的特性，二叉树的遍历次序可以归纳为以下三大类：

- 先上后下的层次遍历
- 先左后右的遍历
- 先右后左的遍历

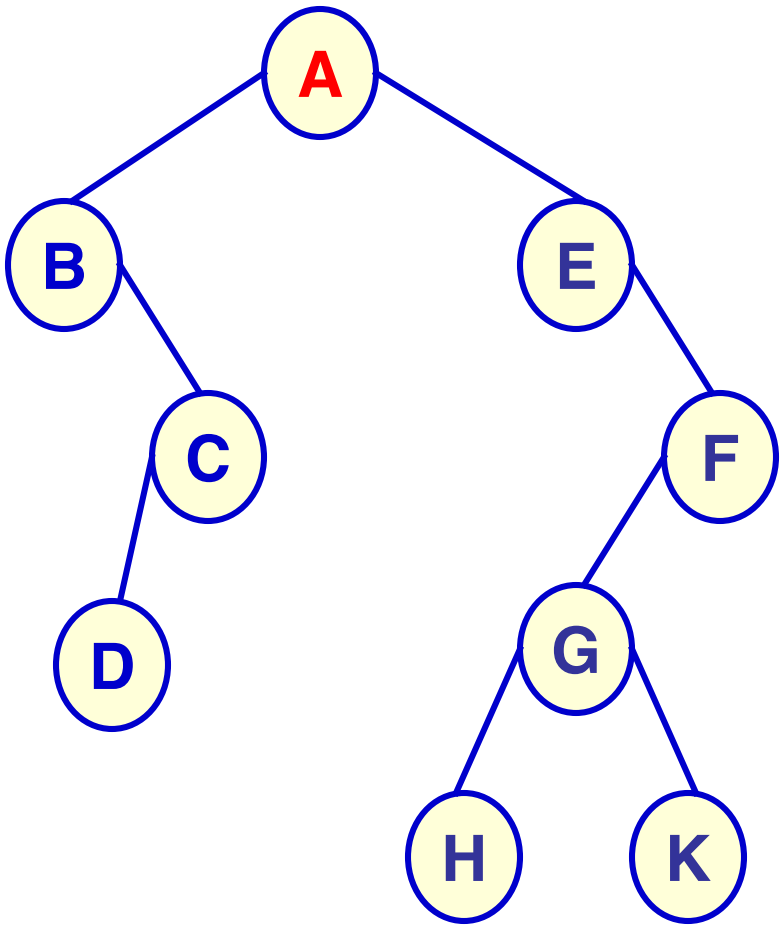
先左后右的遍历算法



DLR先（根）序　　LDR中（根）序　　LRD后（根）序

访问根结点记作 D，遍历根的左子树记作 L，遍历根的右子树记作 R

例如：



先序序列：

A B C D E F G H K

中序序列：

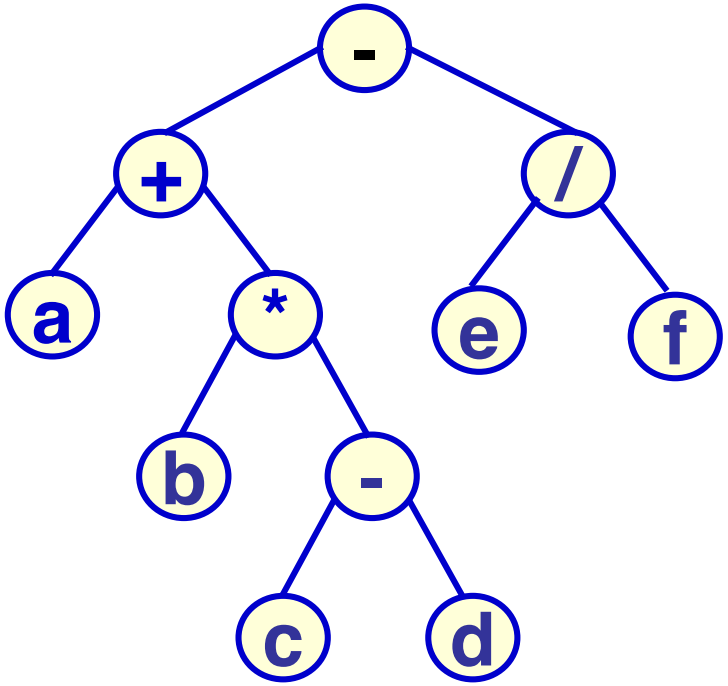
B D C A E H G K F

后序序列：

D C B H K G F E A

二叉树遍历的应用

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



先序：-+a*b-cd/ef

前缀表示（波兰式）

中序：a+b*c-d-e/f

中缀表示

后序：abcd-*+ef/-

后缀表示（逆波兰式）

I 先序遍历

递归算法

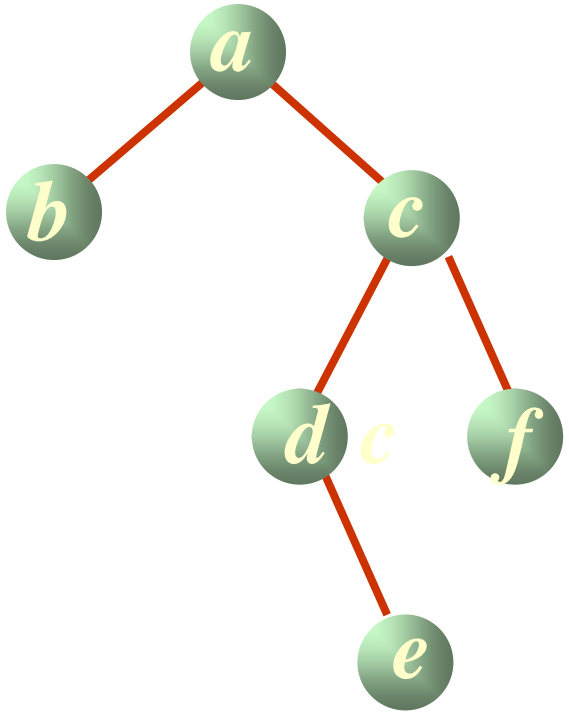
- 若二叉树为空，则空操作；否则
  - 访问根结点 (V)；
  - 先序遍历左子树 (L)；
  - 先序遍历右子树 (R)。

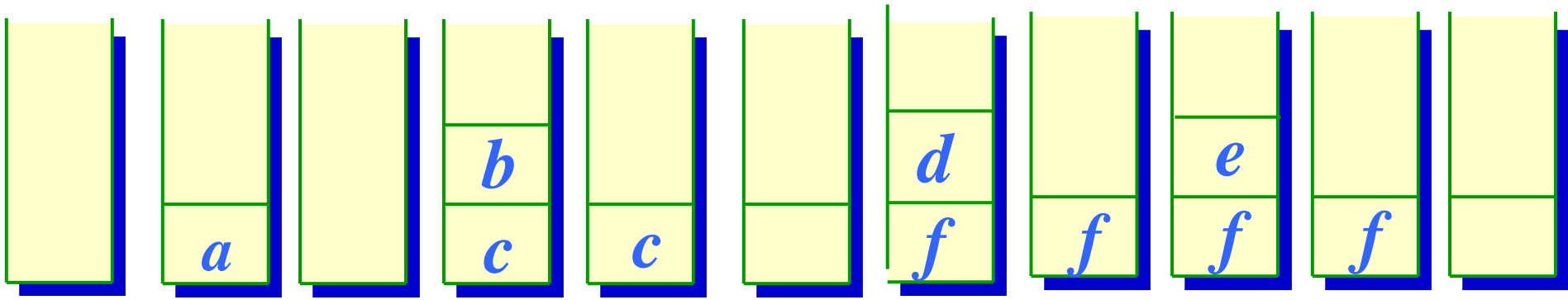
```
Status PreorderTraverse (BiTree T,  Status( *Visit)(TElemType & e)){
    if (!T)  return;
    Visit(T->data);              // 访问结点
    PreorderTraverse (T->lchild, Visit);   // 遍历左子树
    PreorderTraverse (T->rchild, Visit) ;   // 遍历右子树
}//T(n)=O(1)+T(a)+T(n-a-1)=O(n)
```

先序遍历的迭代算法1

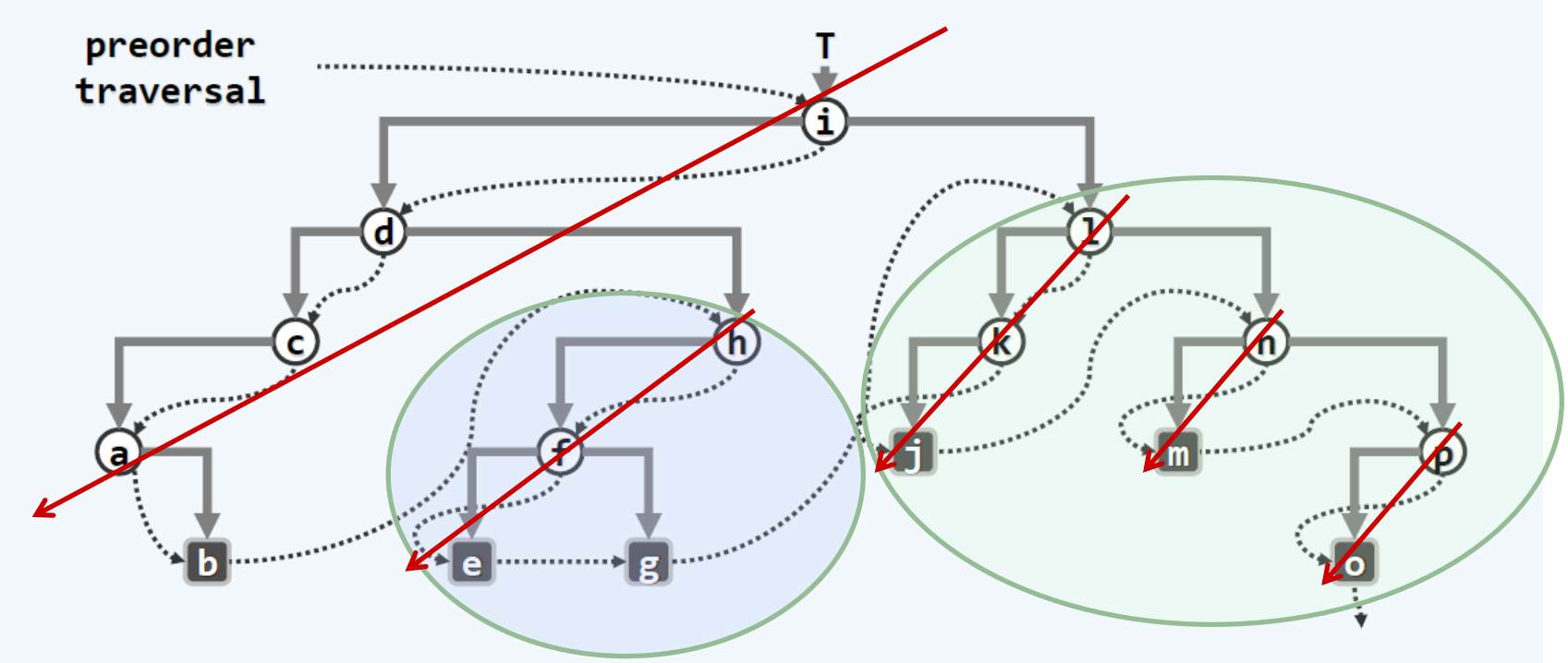
```
void PreOrder_iter1( BiTree T, Status （*Visit)(TElemType e) {
    IniStack(S);
    if (T)   Push (S, T);
    While (! StackEmpty(S)）{//在栈变空前反复循环
        Pop (S, p); visit(p);   //弹出并访问当前结点
        if (p->rchild) Push (S, p->rchild);  //右孩子先入后出
        if (p->lchild) Push (S, p->lchild);   //左孩子后入先出
    }
}
```

迭代实例：





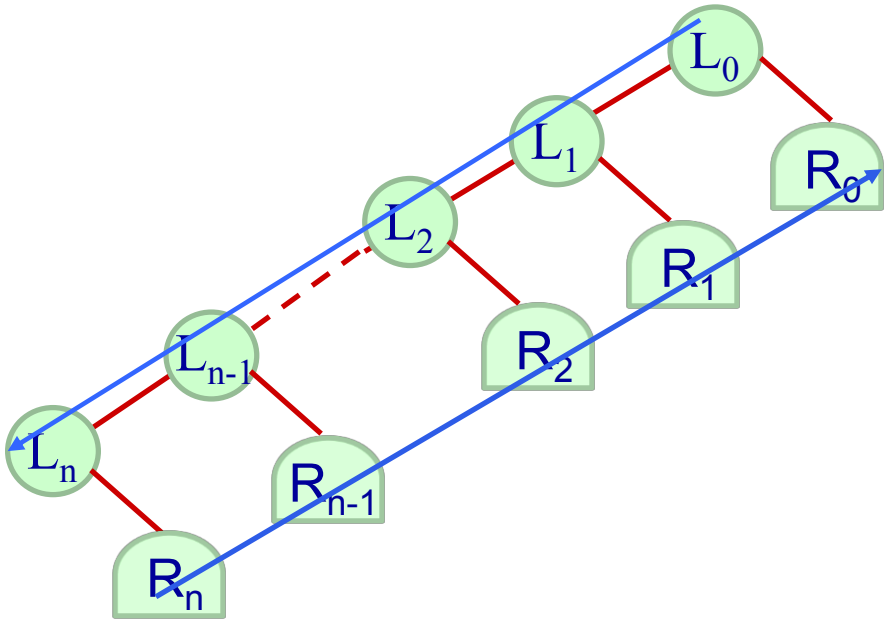
分析:



整个遍历首先

自上而下对左侧分支访问，直到找到最左下方的节点

分析:



- 沿着左侧分支

  各节点与其右孩子（可能为空）一一对应

- 整个遍历可划分为

  自上而下对左侧分支访问，直到找到最左下方的节点

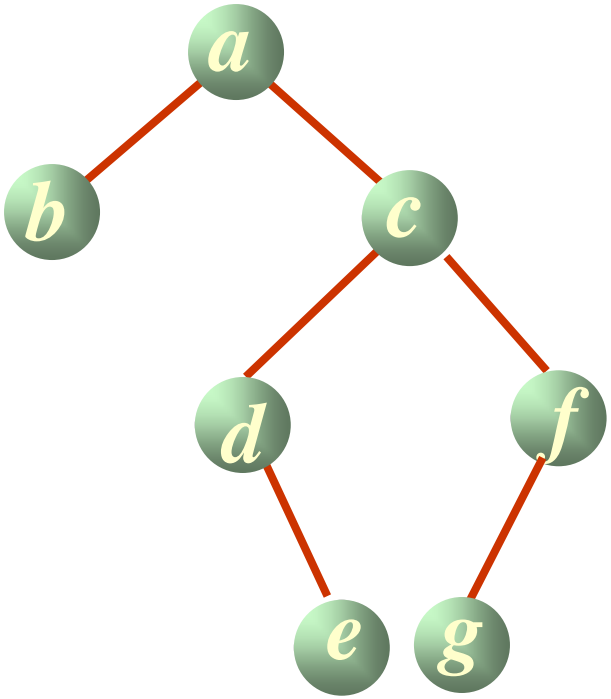
  自下而上对一系列右子树遍历

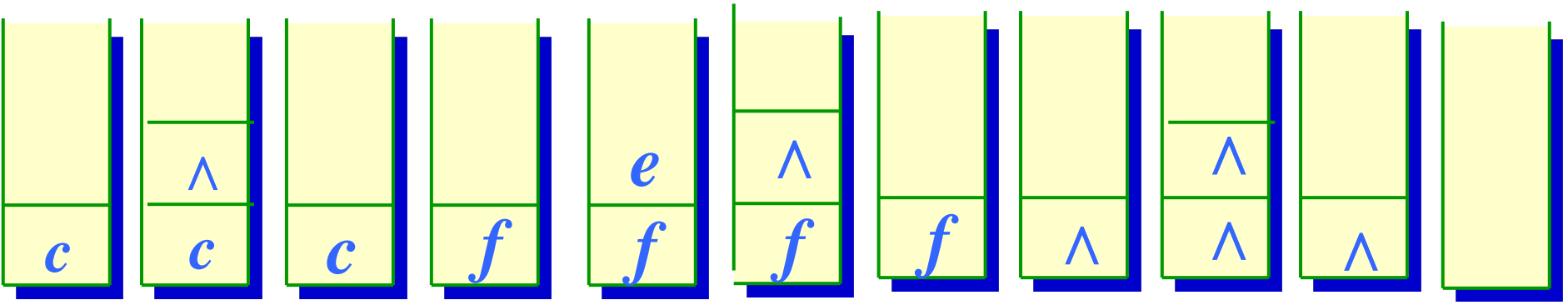
先序遍历的迭代算法2

```
void PreOrder_iter2( BiTree T, Status （*Visit)(TElemType e) {
   IniStack(S);
  while (TRUE) {//在栈变空前反复循环
      while (T){ //访问子树x的左侧链，右子树入栈缓冲
          visit(T);
          Push(S, T->rchild); //右子树入栈
          T = T->lchild; //沿左侧链下行
       }
       if ( empty(S) ) break;
       T = pop(S);
    }
}
```

```
void PreOrder_iter1( BiTree T, Status （*Visit)(TElemType e) {
   IniStack（S）
   if (T)  Push (S, T);
   While (! StackEmpty(S)）{//在栈变空前反复循环
        Pop (S, p); visit(p);  //弹出并访问当前结点
        if (p->rchild) Push (S, p->rchild); //右孩子先入后出
        if (p->lchild) Push (S, p->lchild);  //左孩子后入先出
   }
}
```

迭代实例2:





a b ^ c d e ^ f g ^ ^

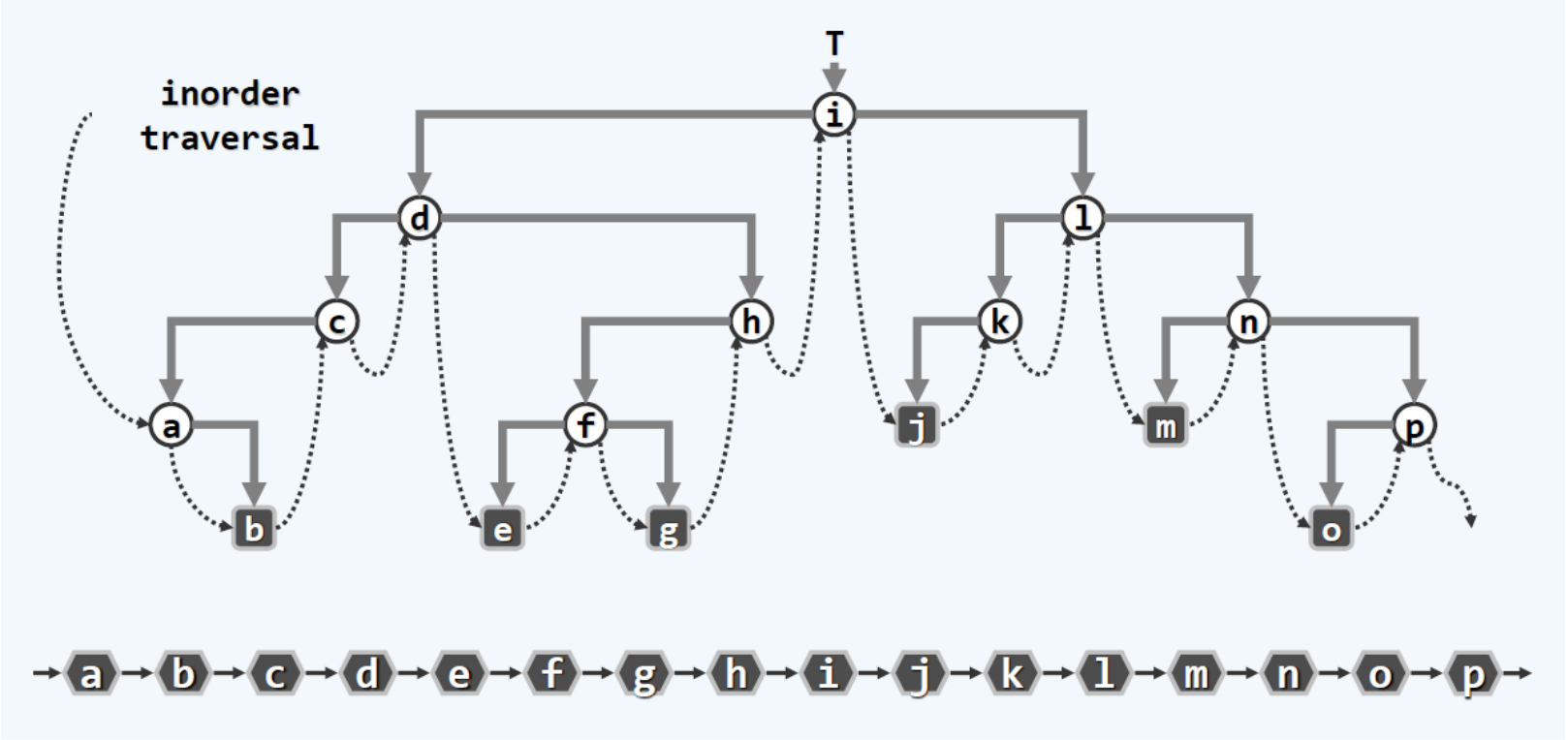
II 中序遍历

递归算法

- 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)。

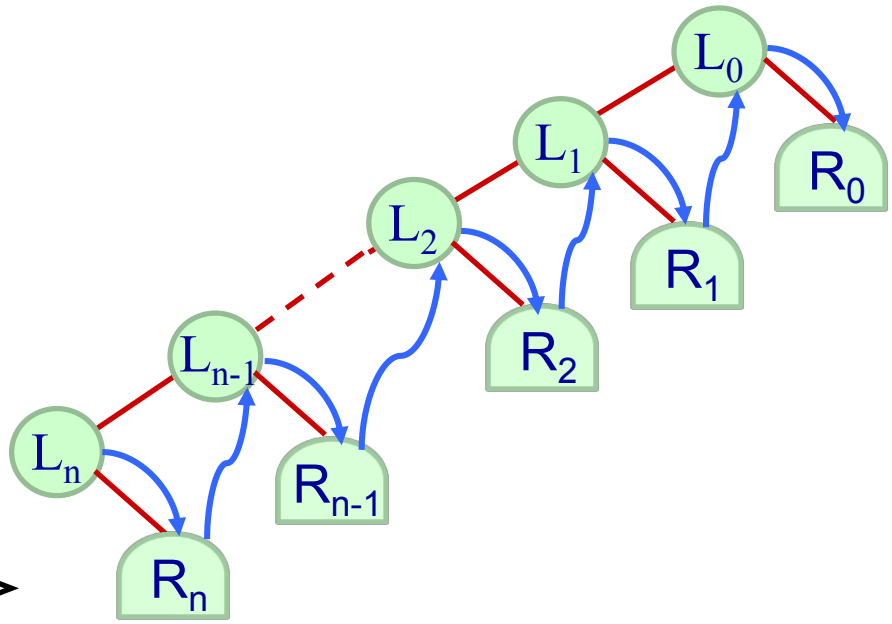
```
Status InorderTraverse (BiTree T,  Status( *Visit)(TElemType & e)){
  if (T) {
     PreorderTraverse (T->lchild, Visit) ; // 遍历左子树
     Visit(T->data);          // 访问结点
     PreorderTraverse (T->rchild, Visit) ; // 遍历右子树
   }else  return OK;
}
```

分析：



- 从根出发沿着左侧分支下行，直到最左下方的节点
  —它就是全局首先被访问者

分析:



最左下方的节点

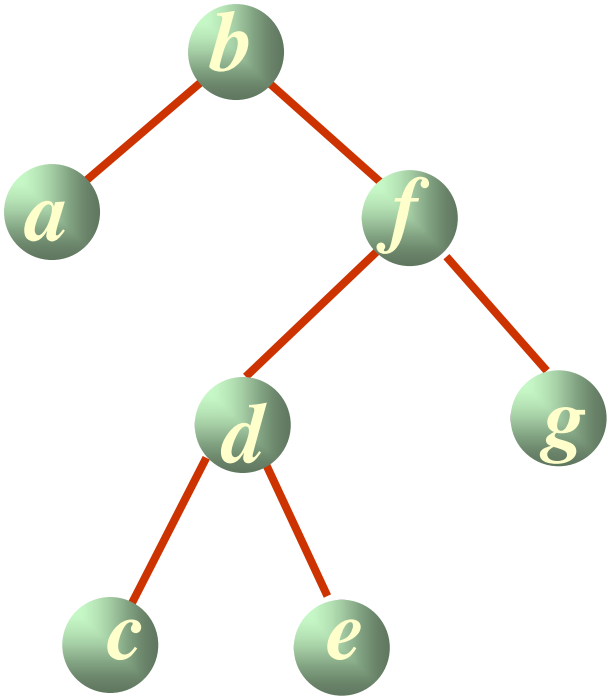
Visit(L_n)　Inorder(R_n)
Visit(L_{n-1})　Inorder(R_{n-1})
⋮
Visit(L_2)　Inorder(R_2)
Visit(L_1)　Inorder(R_1)
Visit(L_0)　Inorder(R_0)

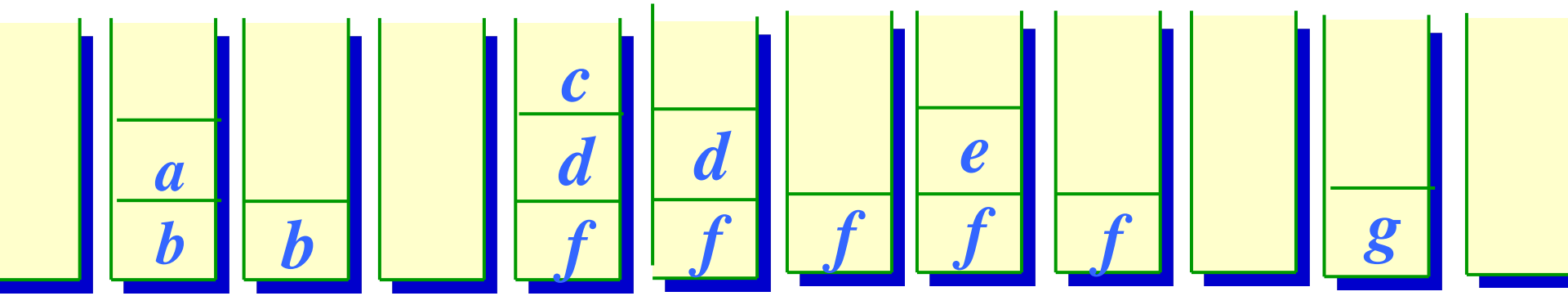
- 沿着左侧分支找到最左下方的节点的同时，将其祖先按从上向下的次序推栈保存
- 对右子树，自下而上对一系列右子树进行中序遍历

中序遍历迭代算法1

```
void InOrder_iter1( BiTree T,  Status (*Visit)(TElemType e) {
   //采用二叉链表存储结构，中序遍历二叉树的非递归算法
   IniStack（S）;
   while (TRUE){
        while (T) {  Push (S, T); T= T->lchild; }//从当前节点, 逐批进栈
        if ( empty(S) ) break;
        T = Pop (S);
        Visit (T->data);       //退栈, 访问
        T = T−>rightChild;      //遍历指针进到右子女
   } //while
};
```

迭代实例：

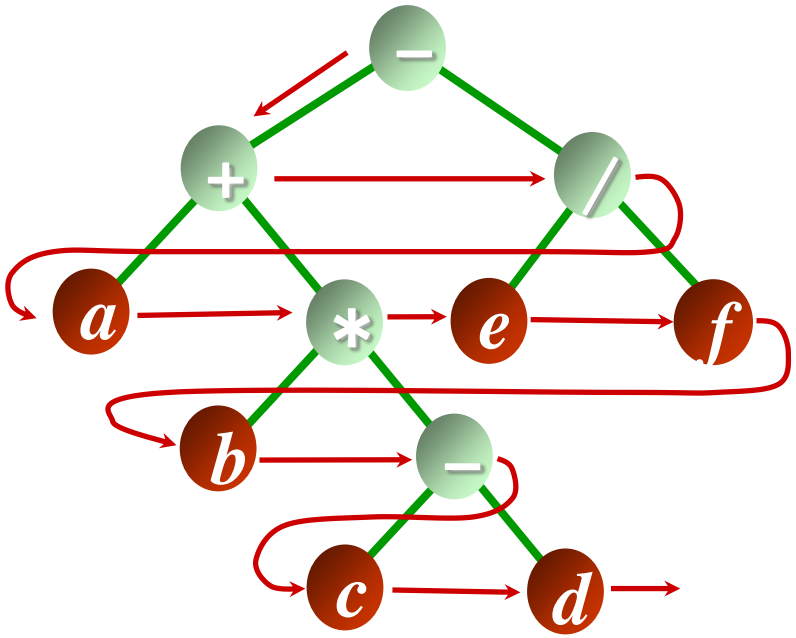




a b c d e f g

层次序遍历二叉树的算法

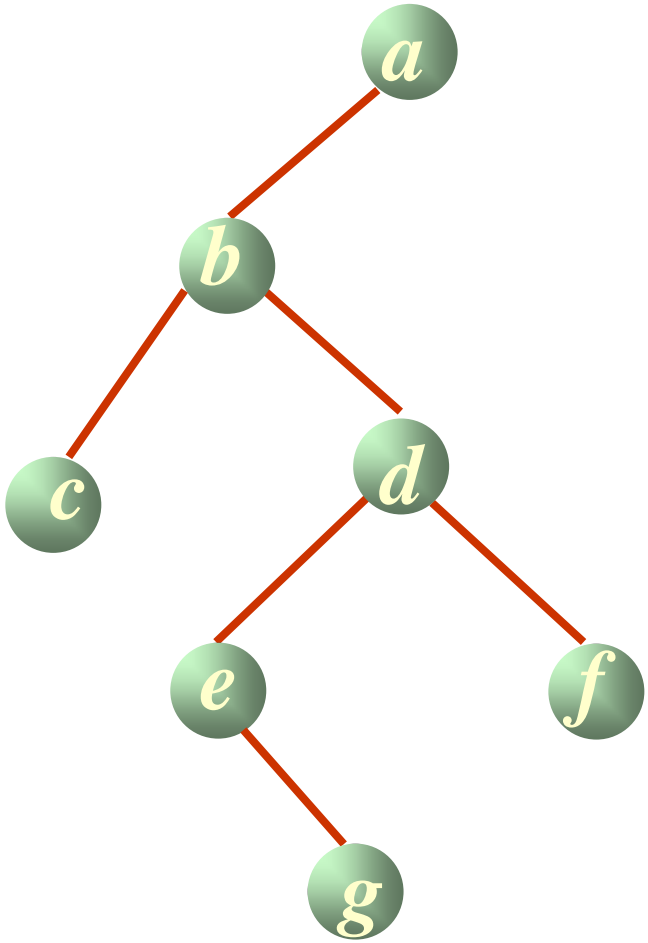
- 树的定义给出了我们在深度方向上二叉树结点的次序，而在同层次上左右节点的顺序给出了二叉树在同层次方向上的次序关系。
- 层次序遍历二叉树就是从根结点开始，按层次逐层从左至右逐一遍历，如图：



-+/a*efb-cd

```
void InOrder_iter2( BiTree T,   Status  (*Visit)(TElemType e) {
      InitQueue(Q);
      EnQueue(Q, T);
      while ( ! QueueEmpty(Q) ) {
            DeQueue(Q,  x);   Visit(x);
            if ( x->leftChild )   EnQueue( Q, x->leftChild )  ;
            if ( x->rightChild )   EnQueue( Q, x->rightChild )  ;
       }
}
```

迭代实例：



6.4 线索化二叉树 (Threaded Binary Tree)

- 二叉树的遍历实际将二叉树的非线性结构线性化，任一数据都有它的前驱和后继。但这种信息只能在遍历过程中才能得到。
- 预处理，将某种遍历顺序下的前驱、后继关系（线索）记在树的存储结构中，实现二叉树的线索化。而加上线索的二叉树称为线索二叉树。

lchild	data	rchild

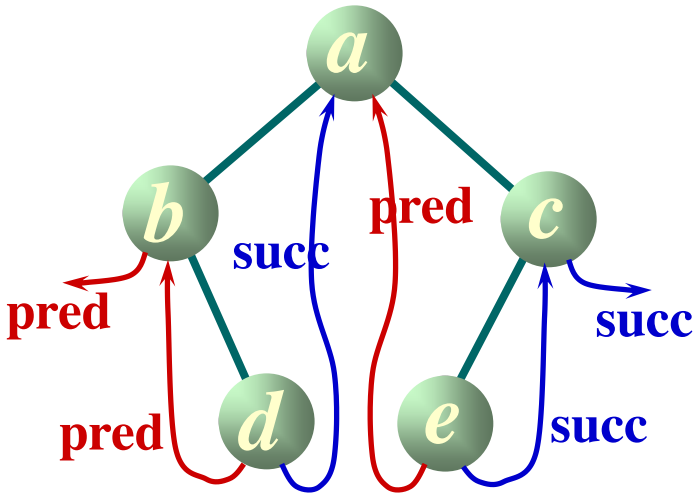
n个结点的二叉链表必定存在n+1个空链域。

- 以这n+1个leftChild 和 rightChild 的空闲指针用作pred 指针和 succ 指针，并增设两个标志 ltag 和 rtag，指明指针是指示子女还是前驱／后继线索。

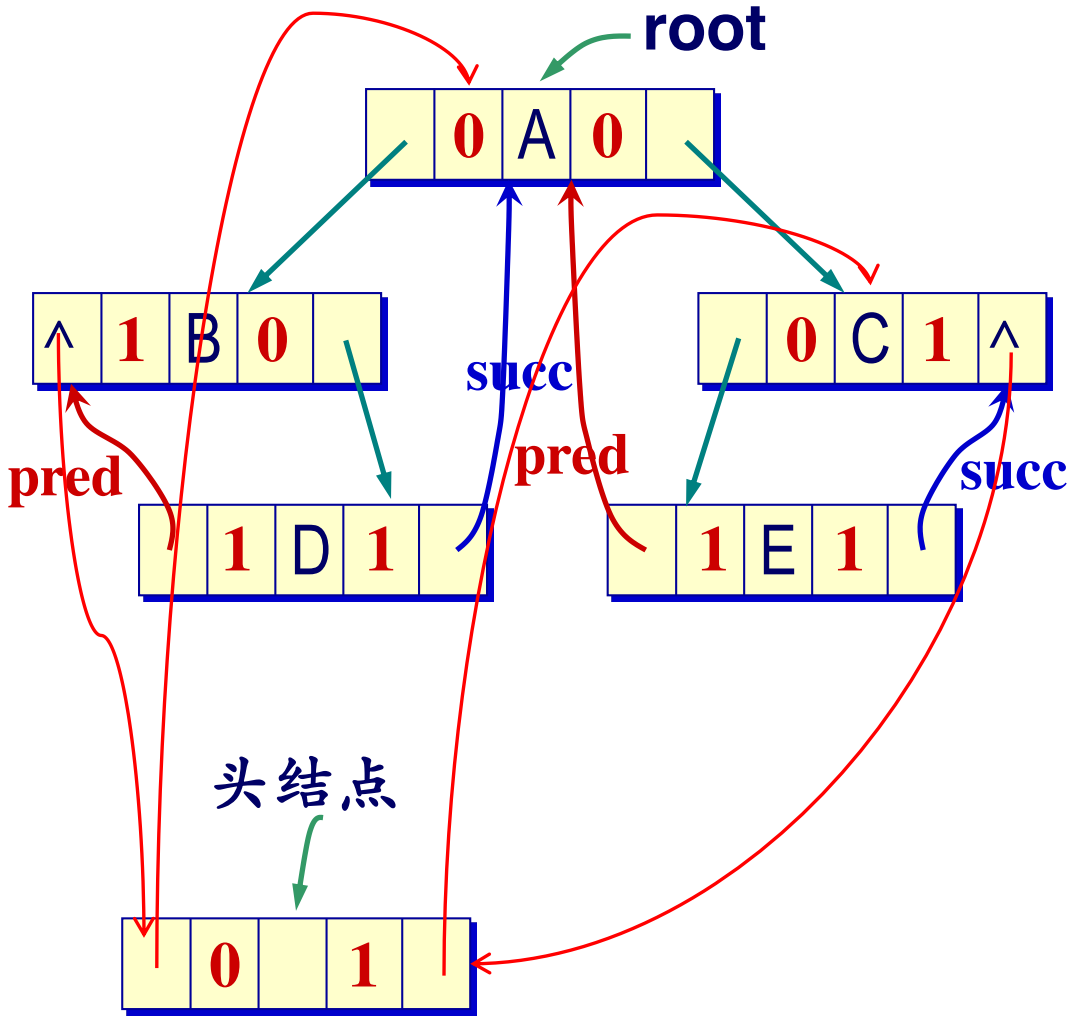
当ltag (或rtag) = 0，表示相应指针指示左子女(或右子女结点)；

当ltag (或rtag) = 1，表示相应指针为前驱(或后继)线索。

以中序遍历二叉树，看二叉树的线索表示

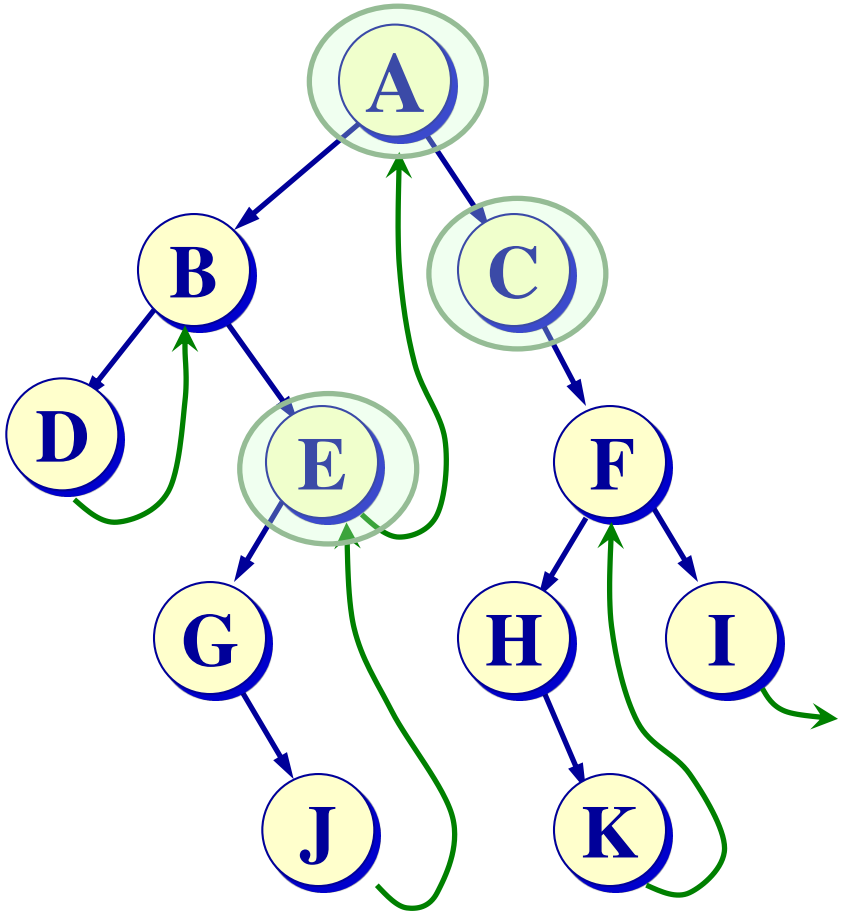


bdaec



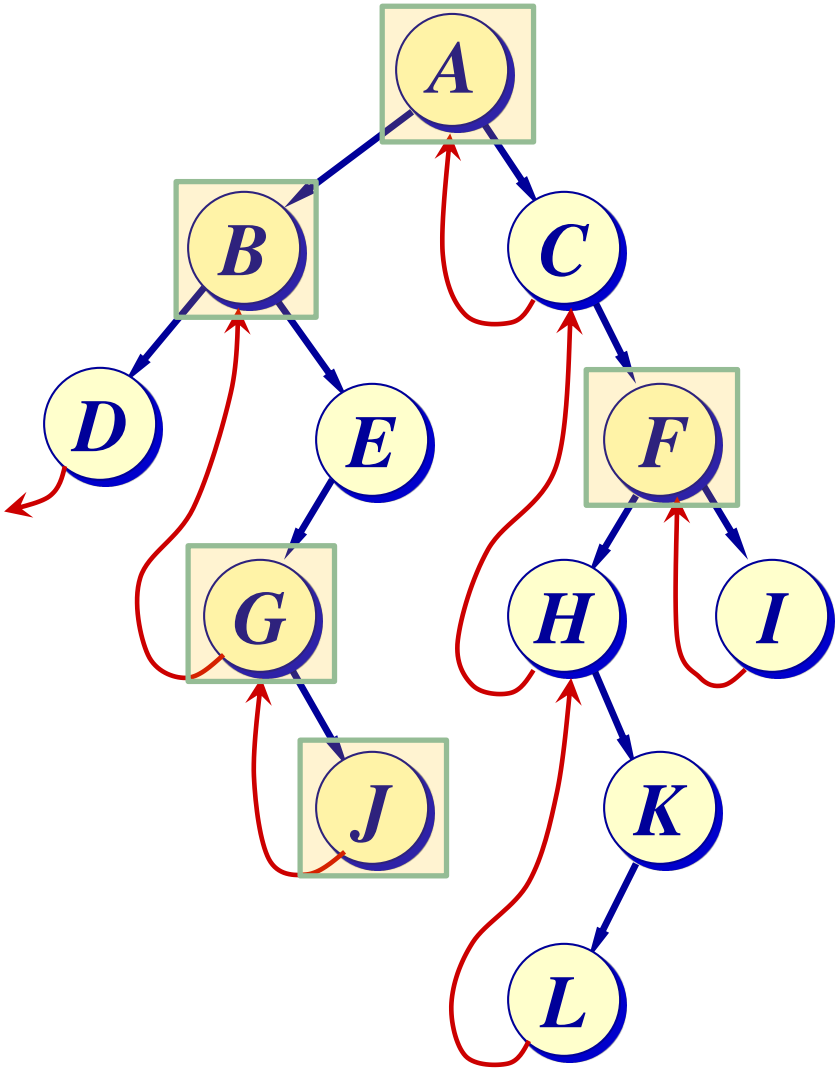
线索二叉树节点的后继：中序线索二叉树

- 右链是线索，
- 右链为指针，该节点的后继即是中序遍历该节点右子树时的第一个节点

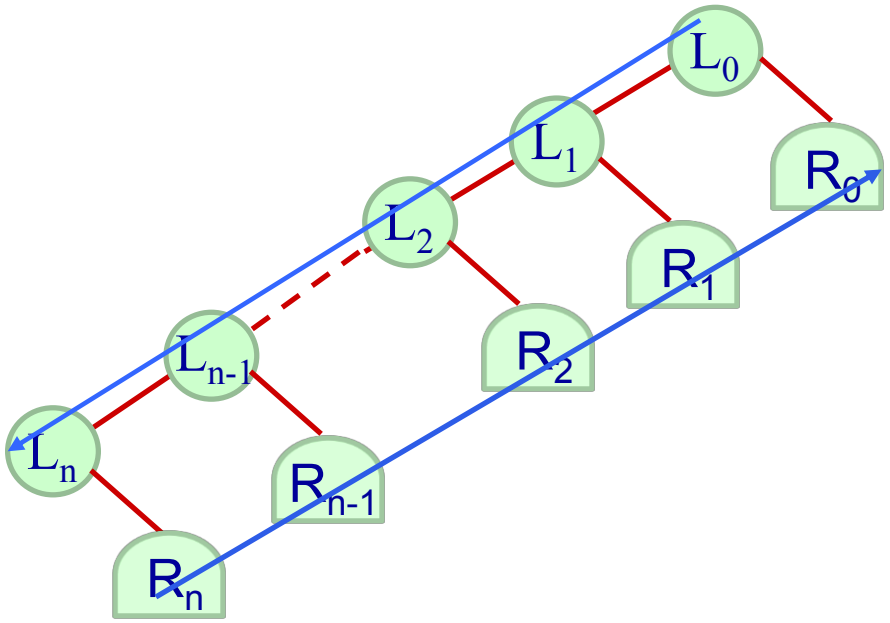


线索二叉树节点的前驱：中序线索二叉树

- 左链是线索，
- 左链为指针，该节点的前驱即是中序遍历该节点的左子树时的最后一个节点



分析：中序遍历



```
void InOrderTraverse_Thr(BiThrTree T,
                         void (*Visit)(TElemType e)) {
  //T指向头结点。中序遍历二叉树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
    Visit(p->data);
    while (p->RTag==Thread && p->rchild!=T) {//无右子树
      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
```

```
Status  InOrderThreading( BiThrTree &Thrt, BiThrTree T ) {
  Thrt->LTag= LINK;  Thrt->Rtag= Thread;  Thrt->rchild = Thrt;
  if (!T)    Thre->lchild = Thrt;
  else {  Thrt->lchild = T; pre = Thrt;
          InThreading(T);
          pre->rchild = Thrt; pre-Rtag = Thread;
          Thrt->rchild = pre;                          }
  return OK;
} // InThreading
```

```
Void InThreading(BiThrTree p){//线索化递归算法
   if (p){   InThreading(p->lchild);
             if  (!p->lchild) { p->LTag = Thread;  p->lchild = pre;}
             if  (!pre->rchild) { pre-Rtag = Thread; pre->rchiild = p;}
             pre = p;
             InThreading(p->rchild);    }
}//INThreading
```

6.6 哈夫曼树与哈夫曼编码

- 最优二叉树（哈夫曼树）的定义
- 如何构造最优树
- 前缀编码

最优树的定义

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。

- 完全 二叉树是 路径长度最短的 二叉树

结点的带权路径长度:

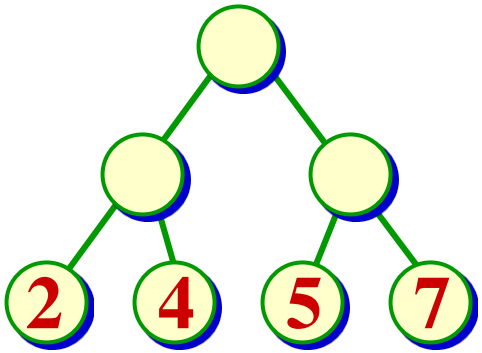
从根结点到该结点的路径长度与该结点的权的乘积。

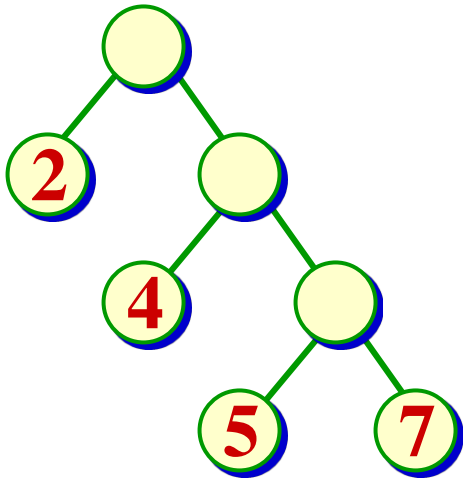
树的带权路径长度:

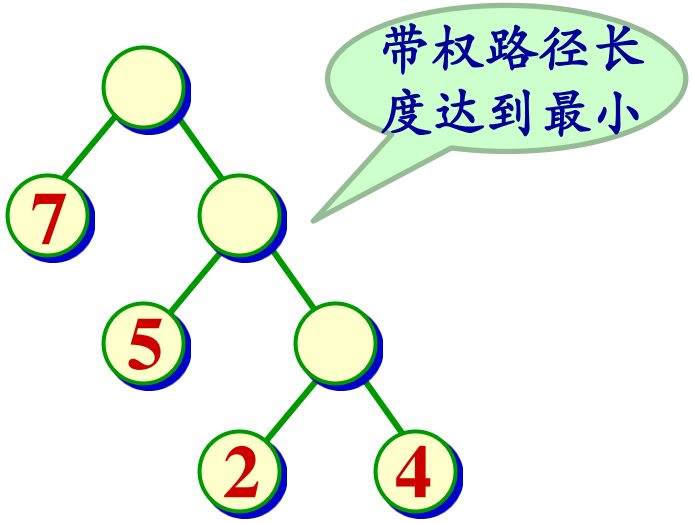
树中所有叶子结点的带权路径长度之和

$$WPL(T) = \Sigma w_k l_k \text{ (对所有叶子结点)}$$

例：







WPL = 2*2+
4*2+5*2+
7*2 = 36

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

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

最优二叉树（哈夫曼树）：

假设有 n个权子$\{w_1, w_2, \ldots w_n\}$，构造一棵又 n个叶子结点的 二叉树，每个叶子结点带权w_i，则其中 带权路径长度WPL值最小的二叉树

Huffman树的构造算法（贪心算法）

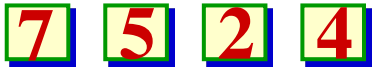
1. 由给定 n 个权值 $\{w_0, w_1, w_2, \ldots, w_{n-1}\}$，构造具有 n 棵二叉树的森林 $F = \{T_0, T_1, T_2, \ldots, T_{n-1}\}$，其中每棵二叉树 T_i 只有一个带权值 w_i 的根结点，其左、右子树均为空。
2. 重复以下步骤，直到 F 中仅剩一棵树为止：
   a) 在 F 中选取两棵根结点的权值最小的二叉树，做为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。
   b) 在 F 中删去这两棵二叉树。
   c) 把新的二叉树加入 F。

贪心算法

- 贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。
- 贪心算法不是对所有问题都能得到整体最优解，但对相当广范围的许多问题是能产生整体最优解的，或者是整体最优解的近似解。

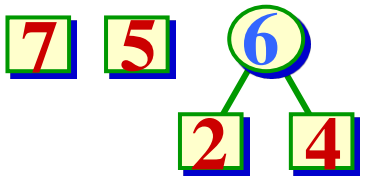
例：

F : {7} {5} {2} {4}



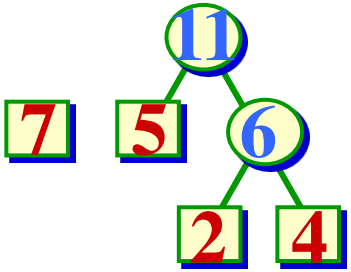
初始

F : {7} {5} {6}



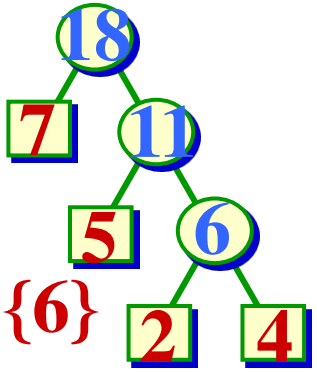
合并{2} {4}

F : {7} {11}



合并{5} {6}

F : {18}



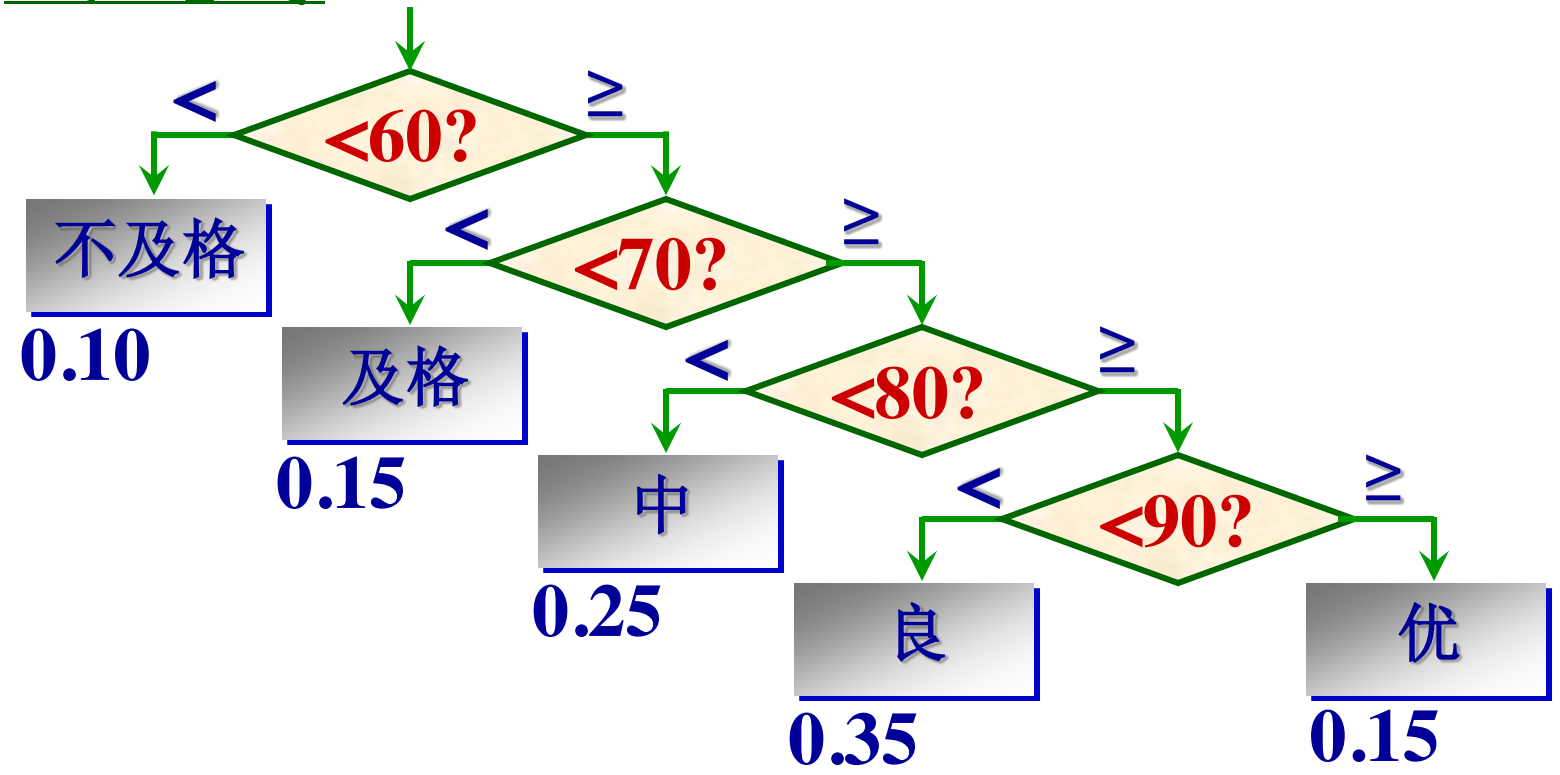
合并{5} {6}

实例1：最佳判定树

考试成绩分布表

[0, 60)	[60, 70)	[70, 80)	[80, 90)	[90, 100)
不及格	及格	中	良	优
0.10	0.15	0.25	0.35	0.15

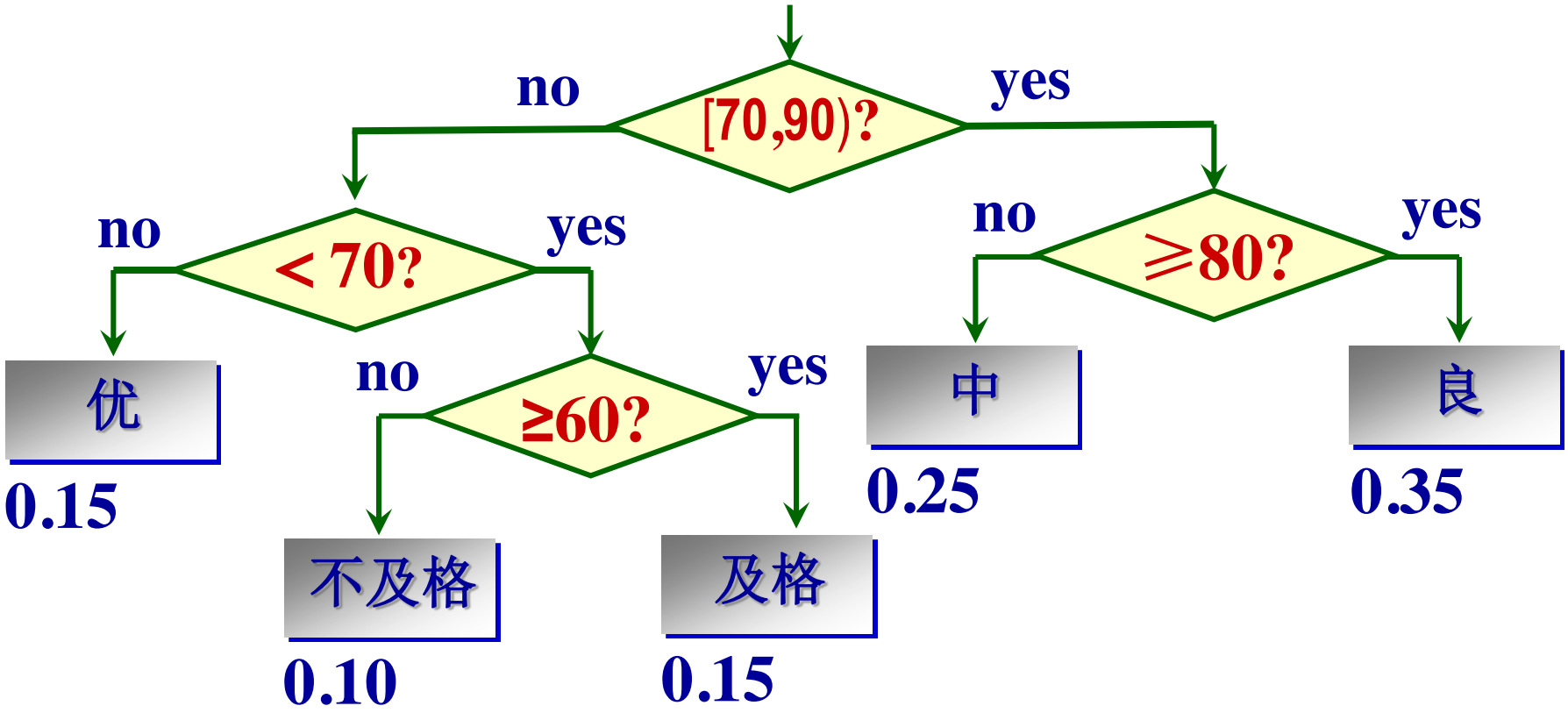
判定树



WPL = 0.10*1+0.15*2+0.25*3+0.35*4+0.15*4

= 3.15

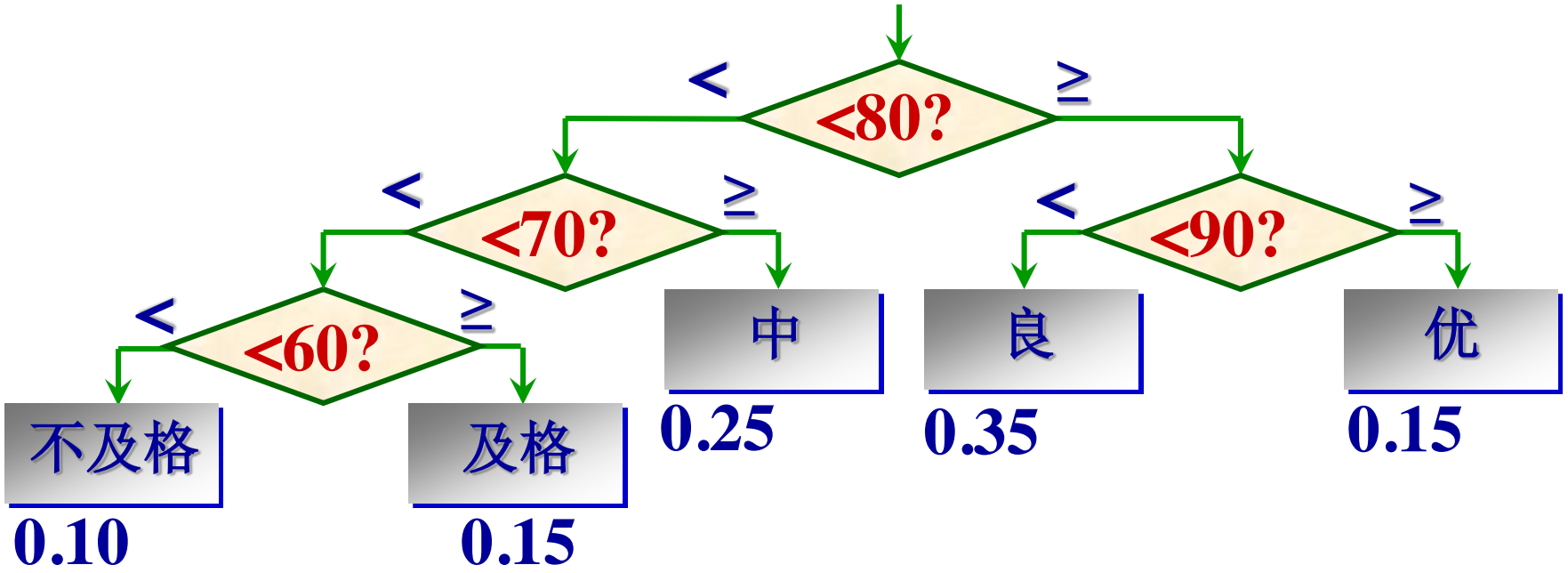
按Huffman算法改造判定树



WPL = 0.10*3+0.15*3+0.25*2+0.35*2+0.15*2

= 0.3+0.45+0.5+0.7+0.3 = 2.25

最佳判定树



WPL = 0.10*3+0.15*3+0.25*2+0.35*2+0.15*2

= 0.3+0.45+0.5+0.7+0.3 = 2.25

哈夫曼树的特点：

- 哈夫曼树中没有度为1的结点，树中任意非叶子结点都有2个儿子，这类树又称为正则二叉树；
- 一棵有n个叶子结点的哈夫曼树共有2n-1个 结点 。

实例2：Huffman编码

例： 字符集合是{C, A, S, T}，

各个字符出现的频度（次数）是 $W=\{2, 7, 4, 5\}$。

- 若给每个字符以等长编码（2位二进制足够）

  A : 00　T : 10　C : 01　S : 11

则CAST CAST SAT AT A TASA

总编码长度为 (2+7+4+5) * 2 = 36。

- 若按各个字符出现的概率不同而给予不等长编码，可望减少总编码长度。

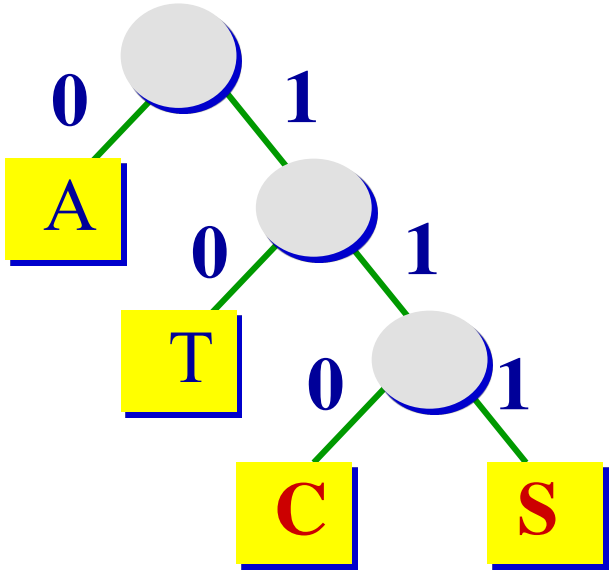
  例： A : 0　T : 10　C : 101　S : 111

  它的总编码长度：7*1+5*2+(2+4)*3 = 35。比等长编码的情形要短。

- 但是这样的电文产生歧义。

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。

- 利用二叉树可以构造一种不等长的二进制编码，而且得到的必为二进制前缀编码。



编码树

A (0)　B (10)　C (110)　S (111)

- 编码的前缀性质可以使译码方法非常简单；例如0010111010可以唯一的分解为0，0，10，111，0，10，因而其译码为AABSAB。
- 构造以出现频率为权值的哈夫曼树，就能得到相应的哈夫曼编码，这是一种最优前缀编码，即使所传电文的总长度最短。

哈夫曼编码算法1

```
typedef struct {
        unsigned int  weight;;
        unsigned int parent, lchild, rchild
} HTNode, *HuffmanTree;
Typedef char **HuffmanCode;

void HuffmanCoding(
        HuffmanTree  &HT, HuffmanCode  &HC, int *w, int n) {
  if (n<=1) return;
  m = 2*n-1; HT = (HuffmanTree )malloc((m+1)*sizeof(HTNode));
  for (p=HT, i=1; i<=n; ++i, ++p, ++w) *p={*w, 0, 0, 0);//初始化
  for (; i<=m; ++i, ++p) *p = {0, 0, 0, 0);
  for (i=n+1; i<=m; ++i){
      //在[1..i-1]选择parent为0且weight最小的两个节点s1和s2
      Select(HT, i-1, s1, s2);
      HT[s1].parent = i; HT[s2].parent = i;
      HT[i].lchild = s1; HT[i].rchild = S2;
      HT[i].weight = HT[s1].weight+HT[s2].weight;
  }
```

```
//--------从叶子到 根逆向求 每个 字符的哈夫曼编码--------
HC= （HuffmanCode） malloc( (n+1)*sizeof(char *));
cd = (char *)mallco (n *sizeof(char));
cd[n-1] = “\0”;
for (i=1; i<=n; ++i){//从下往上获取哈夫曼编码
    start = n-1;
    for (c= i, f=HT[i].parent; f!=0; c=f, f=HT[f].parent)
        if (HT[f].lchild == c) cd[--start] = “0”;
        else cd[--start] = “1”;
    HC[i] = (char *)malloc((n-start)*sizeof(char));
    strcpy(HC[i], &cd[stat]);
}
free(cd);
}//HuffmanCoding
```

```
//--------无栈非递归遍历哈夫曼树，求哈夫曼编码 --------
HC= （HuffmanCode）malloc( (n+1)*sizeof(char *));
p = m; cdlen = 0;
for （i=1； i<=m； ++i） HT[i].weight = 0; //预设访问标志位
while (p) {
    if (HT[p].weight ==0){   HP[p].weight = 1;
        if (HP[p].lchild != 0){p = HT[p].lchild; cd[cdlen++] = “0”;}
        else if (HT[p].rchild == 0){//走到了叶节点
                HC[p] = (char *)malloc((cdlen+1)*sizeof(char));
                cd[cdlen] = “\0”; strcpy(HC[p], cd);   }
    }
    else if (HT[p].weight == 1){    HT[p].weight = 2;
            if (HT[p].rchild != 0){ p = HT[p].rchild;
                                    cd[cdlen++] = “1”;    }
        } else {//HT[p].weight==2
              HT[p].weight = 0; p = HT[p].parent; --cdlen;
        }//else
}//while
```

6.4 树与森林

树的存储表示有许多，需要根据各自的要求设立。这里介绍三种有特色的表示方法：

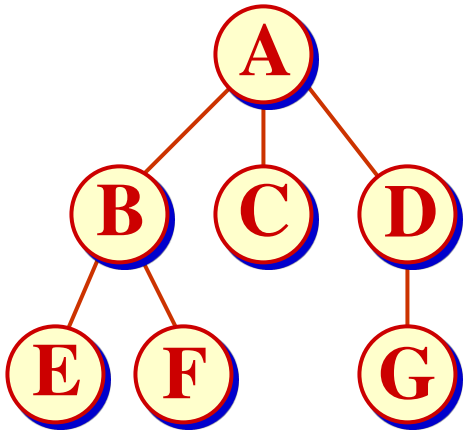
一、双亲表示法

二、孩子链表表示法

三、树的二叉链表(孩子-兄弟）存储表示法

一、双亲表示

- 利用了每个节点只有唯一的双亲的性质
- 是一种顺序存储的表示方法



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

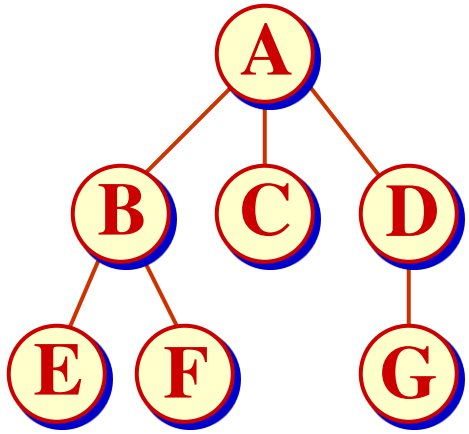
```
#define MAX_TREE_SIZE  100
```

```
typedef struct PTNode {
        Elem  data;
        int    parent;
} PTNode;
```

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

二、孩子表示

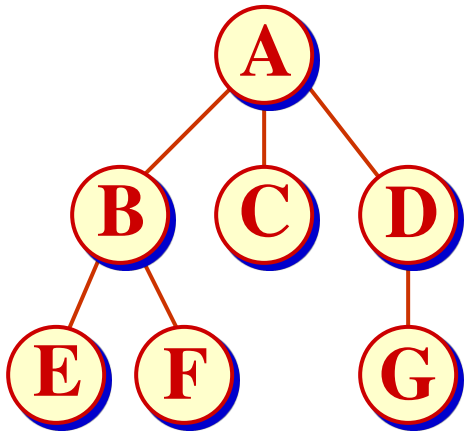
顺序存储

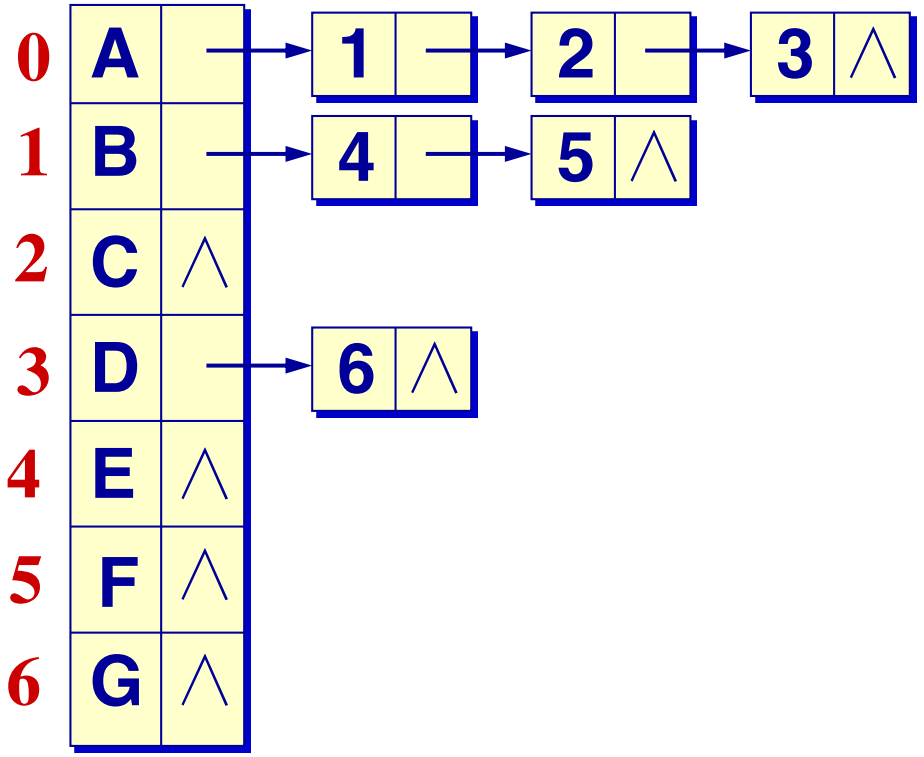


0	A	3	1	2	3
1	B	2	4	5	
2	C	0			
3	D	1	6		
4	E	0			
5	F	0			
6	G	0			

- 无序树情形链表中各结点顺序任意，有序树必须自左向右链接各个子女结点。

链表





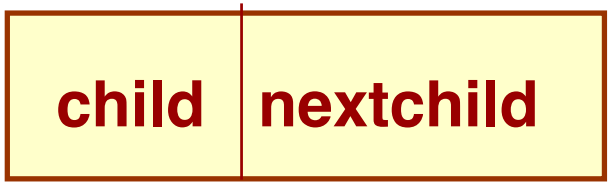
孩子结点结构

child	nextchild

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

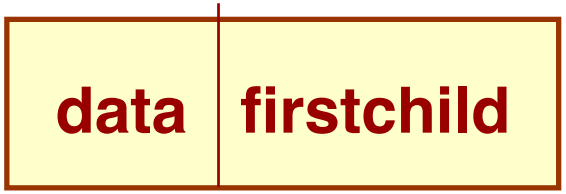
C语言的类型描述：

孩子结点结构




```
typedef struct CTNode {
    int         child;
    struct CTNode *nextchild;
} *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;
```

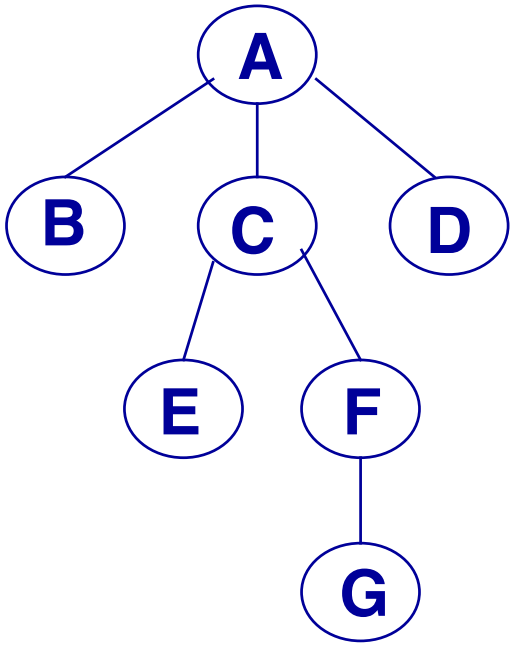
三、树的二叉链表（孩子-兄弟）存储表示法

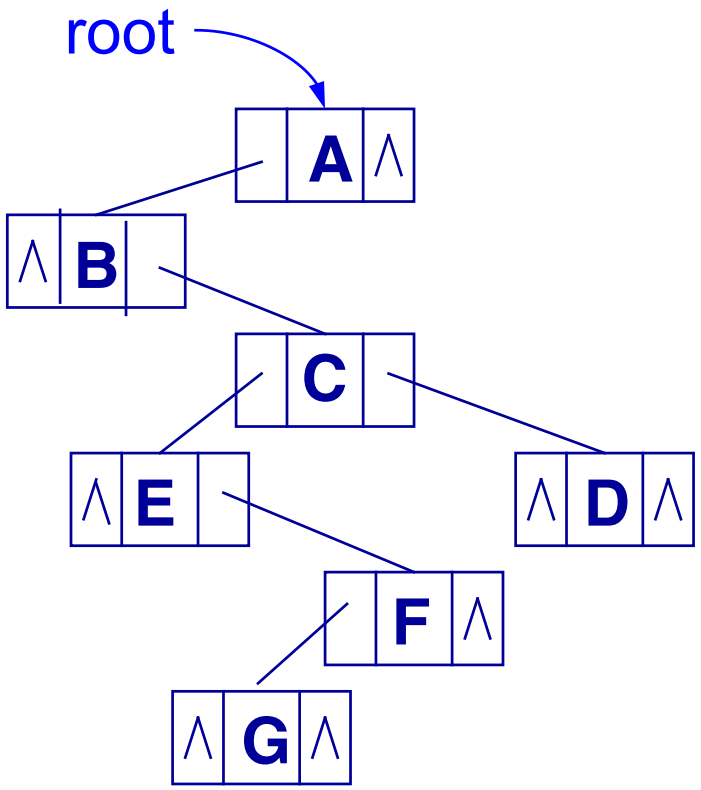
- 也称为树的二叉树表示。结点构造为：

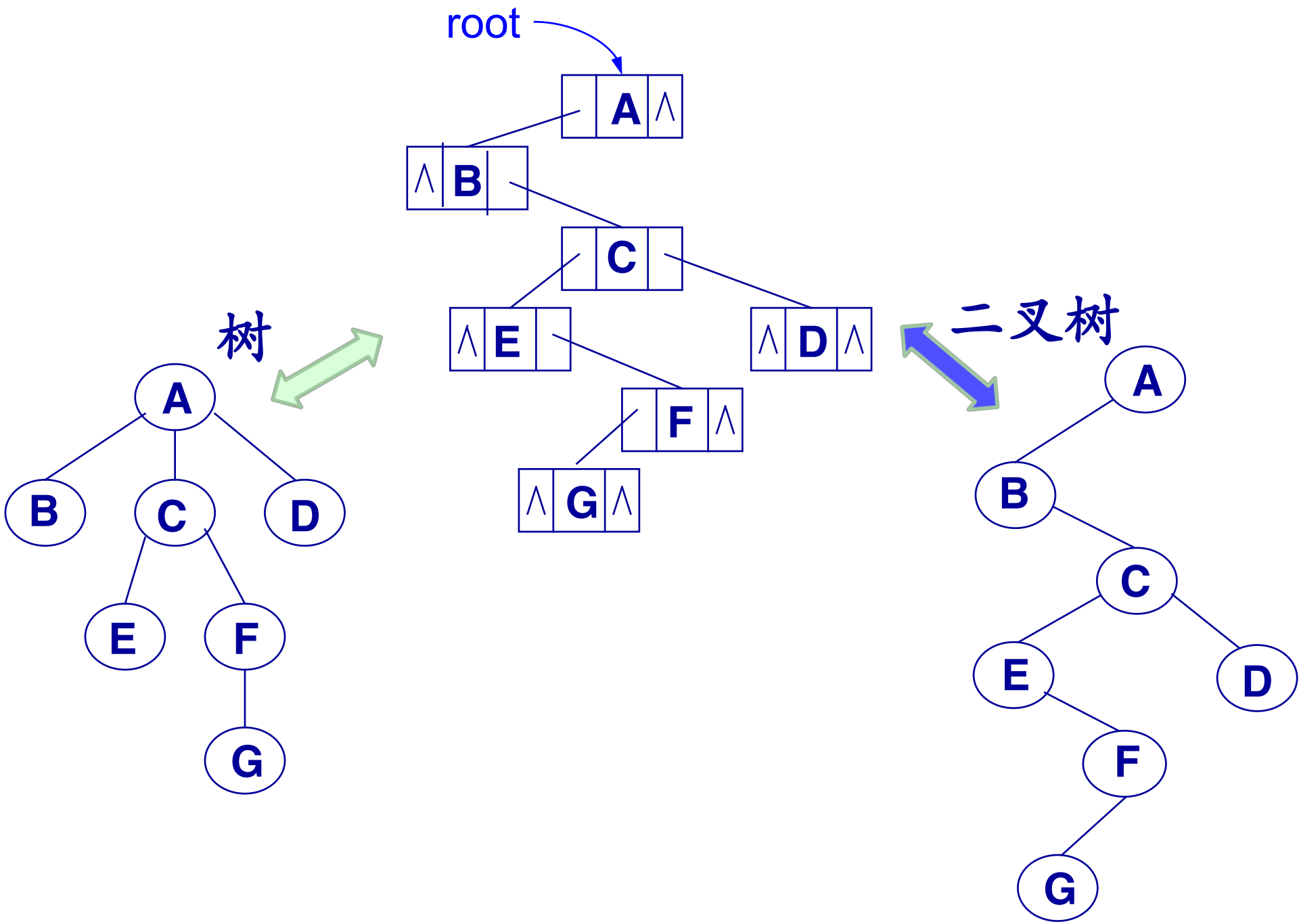
- **firstChild** 指向该结点的第一个子女结点。无序树时，可任意指定一个结点为第一个子女。
- **nextSibling** 指向本结点的下一个兄弟。

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

例：






root
A
B
C
E
D
F
G
树
二叉树
A
B
C
D
E
F
G
A
B
C
E
D
F
G

森林与二叉树的对应关系

由森林F=$\{T_1, T_2, \cdots T_m\}$转换成
二叉树B=(root, LB, RB)的转换规则为:

若 F = Φ，则 B = Φ;

否则，由 ROOT(B) = ROOT(T1)

由 ($t_{11}, t_{12}, \ldots, t_{1m}$) 对应得到 LB;

由 ($T_2, T_3, \ldots, T_n$) 对应得到 RB。

由二叉树B=(root, LB, RB)转换为
森林F=$\{T_1, T_2, \cdots T_m\}$的转换规则为:

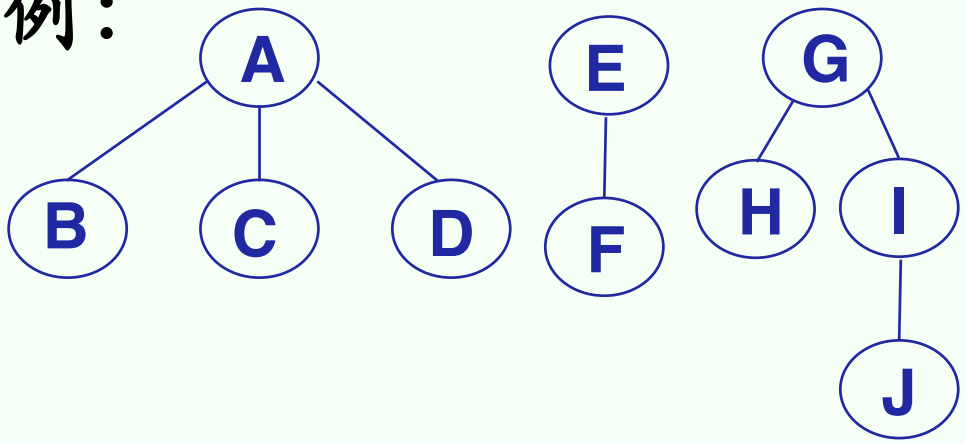
若 B = Φ, 则 F = Φ;

否则，由 ROOT(T_1)=ROOT(B);

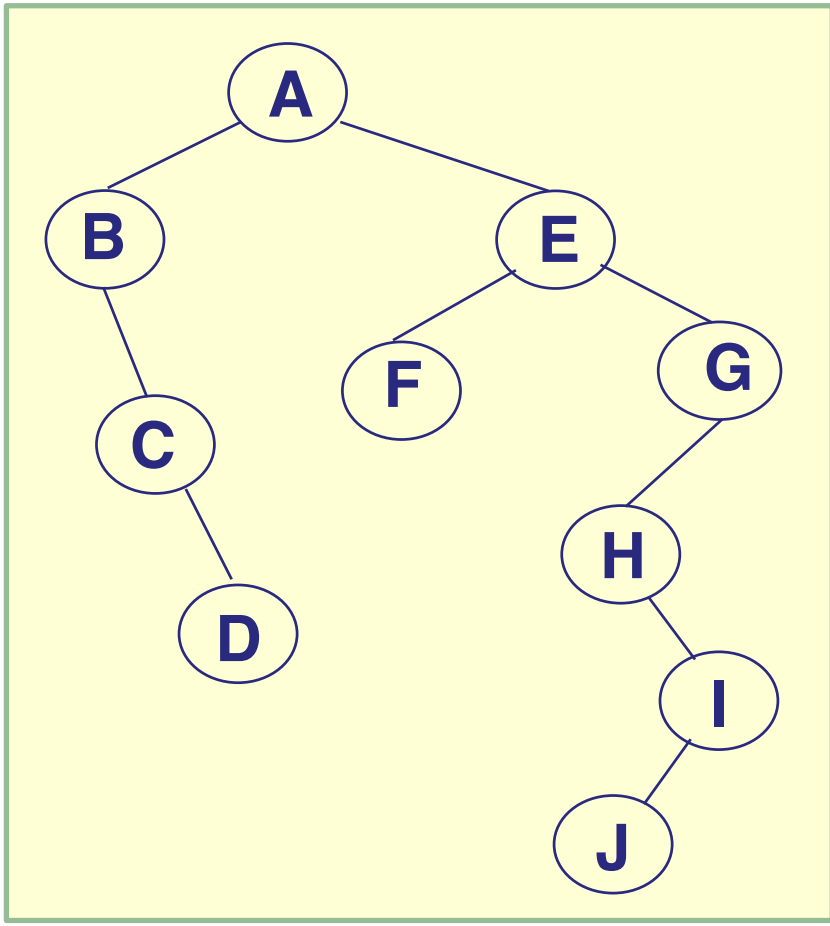
由LBT对应得到 ($t_{11}, t_{12}, \ldots, t_{1m}$);

由RB 对应得到 ($T_2, T_3, \ldots, T_n$)。

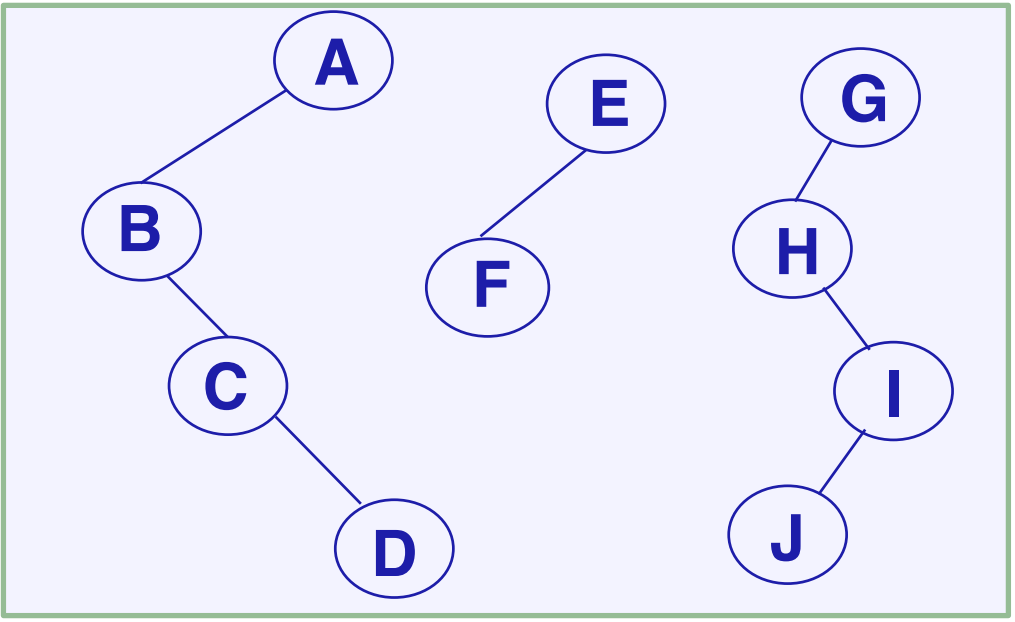
例：



森林与二叉树对应



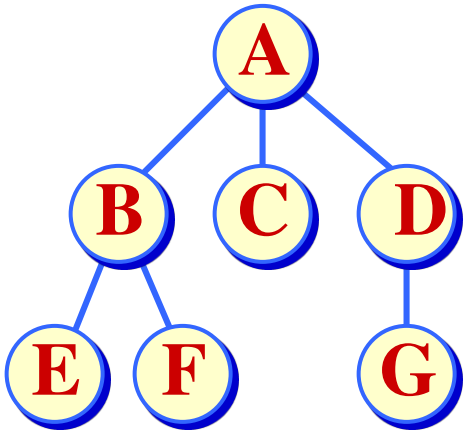
树与二叉树对应



树根相连

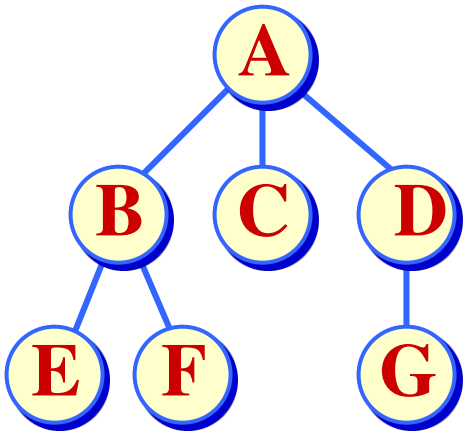
树的遍历

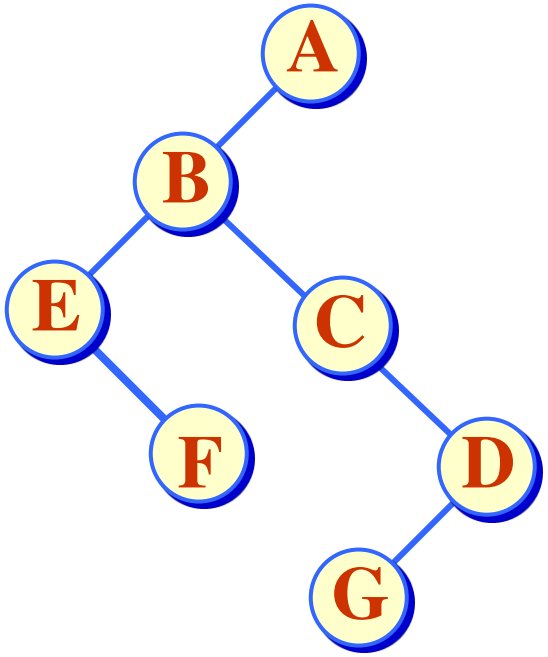
- 先根次序遍历：
- 后根次序遍历：



树先根遍历：　ABEFCDG

树后根遍历：　EFBCGDA





- 树的先根遍历可以借助对应二叉树的前序遍历算法实现
- 树的后根遍历可以借助对应二叉树的中序遍历算法实现

森林的遍历

- 先序遍历
- 中序遍历

森林的先序遍历：

依次从左到右对森林中的每棵树进行先根遍历

当森林非空时

- 访问森林第一棵树的根结点
- 先序遍历第一棵树的子树森林
- 依次先序其它树构成的森林

- 森林的先序遍历结果与其对应二叉树表示的先序遍历结果相同
- 树的先序遍历可以借助对应二叉树的先序遍历算法实现

森林的中序遍历：

依次从左到右对森林中的每棵树进行后根遍历

当森林非空时

- 中序遍历森林第一棵树的根结点的子树森林
- 访问第一棵树的根节点
- 中序遍历其它树构成的森林

- 森林的中序遍历结果与其对应二叉树表示的中序遍历结果相同
- 树的中序遍历可以借助对应二叉树的中序遍历算法实现

6.5 树与等价问题

1 等价关系

假定有n个元素的集合U，另有一个U上关系 R∈UXU。

关系R是一个等价关系，当且仅当如下条件为真时成立：

①对于所有的a，有(a,a)∈R时，即关系是自反的。

②当且仅当(b,a)∈R时(a,b)∈R，即关系是对称的。

③若(a,b)∈R且(b,c)∈R，则有(a,c)∈R，即关系是传递的。

若R是S上的等价关系，由R可以产生这个集合S的一个唯一的划分S1, S2, ….。等价类是集合中相互等价的元素的最大子集合；这些集合互不相交，其并集为S。

2　划分等价类的算法思想

① 令S中每个元素各自形成一个只含单个元素的子集，记为S1，…，Sn。

② 依次扫描m个偶对，对每个扫描的偶对（x，y），判定x和y所属的子集。假设x∈Si，y∈Sj，若Si≠Sj，则将Si并入Sj并置Si为空（或将Sj并入Si并置Sj为空）。当m个偶对都被处理后，S1，S2，…,Sn中所有非空子集即为S的R等价类。

ADS MFSet{

数据对象：**若设S是MFSet型的集合，则它由n(n>0)个子集S_i(i=1, 2, …n)构成，每个子集的成员都是子界[-maxnumber, maxnumber]内的整数**

数据关系：$S_1 \cup S_2 \cup \ldots \cup S_n = S$, $S_i \subset S$

基本操作：

Initial(&S, n, x1, x2, ….xn);

Find(S, x);　　//确定x所属子集S_i

Merge(&S, i, j);

}

以集合为基本结构的抽象数据类型有很多实现方法

例：假设集合 S={x|1 ≤x≤n是正整数}， R是S上的一个等价关系。

R= {(1,2),(3,4),(5,6),(7,8),(1,3), (5,7),(1,5),…}

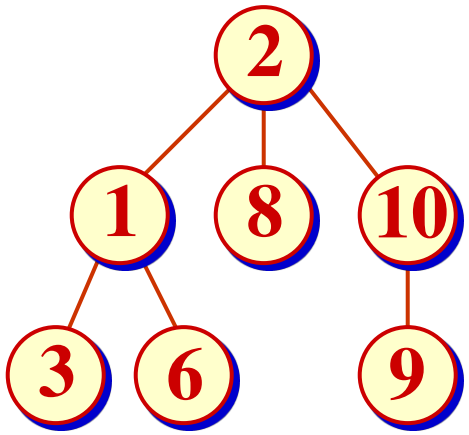
求S的等价类。

以双亲表示法作为MFSet的存储结构

```
typedef struct PTNode{
    DataType data;
    int parent;  //指示双亲位置
}PTNode;

typedef struct{
  PTNode  nodes[MAX_TREE_SIZE];
  int n;//结点数
}PTree;

Typedef  Ptree  MFSet;
```



	0	1	2	3	4	5	6
data	2	1	8	10	3	6	9
parent	−1	0	0	0	1	1	3

O(d) ，其中d是树的深度，其值和树的形成过程有关。如果在最坏情况下，在n-1次“并”下，则全部的操作时间便是$O(n^2)$。

```
//确定集合S中元素i所属子集的根
int find_mfset(MFSet S, int i){
  if (i<1|| i>S.n) return -1;

  for(j=i; S.nodes[j].parent>0; j=S.nodes[j].parent);
  return j;
}

//求Si∪Sj
Status merge_mfset(MFSet &S, int i, int j){
  // S.nodes[i]和S.nodes[j]分别为S的互不相交的两个子集Si和Sj的根结点
  if(i<1 || i>S.n || j<1 || j>S.n) return ERROR;
  S.nodes[i].parent = j;
  return OK;
}
```

O(1)

- 改进合并/查找算法性能的办法是根据"重量规则"进行合并操作。若树i结点数少于树j结点数，将j作为i的父结点，否则将i作为j的父结点。

修改相应的存储结构：令根结点的parent域存储子集中所含元素数目的负值。

```
void mix_mfset(MFSet &S, int i,  int j){
// S.nodes[i]、S.nodes[j] 为S互不相交的两个子集Si和Sj的根结点
  if (i<1 || i>S.n || j<1 || j>S.n)return ERROR;
  if(S.nodes[i].parent>S.nodes[j].parent){  // Si所含元素比Sj少
       S.nodes[j].parent += S.nodes[i].parent ;
       S.nodes[i].parent  =  j;
  }
  else{  S.nodes[i].parent += S.nodes[j].parent
         S.nodes[j].parent  = i;
   }
  return OK;
}
```

- 若按此方法构建有n个结点的树t，则t的高度最多为 ⌊$\log_2 n$⌋ +1

- 随着子集的依次合并，树的深度不断增大。为改善此种情况，当所需确定的元素i不在树的第二层时，可通过"压缩路径"功能缩短元素到达根结点的路径。

```
int fix_mfset(MFSet &S, int i){
// 确定i所在子集，将从i至根路径上所有结点变为根的孩子结点
    if(i<1 || i>S.n)return-1;   //i不是S中的任何子集的元素
    for(j=i; S.nodes[j].parent>0; j=S.nodes[j].parent);
    for(k=i;k !=j;k=t){
        t=S.nodes［k］.parent;
        S.nodes［k］.parent=j;
    }
    return j;
}
```

6.6 回溯法与树的遍历

回溯法是一种“穷举”方法，也叫试探法。它是一种系统地搜索问题的解的方法。

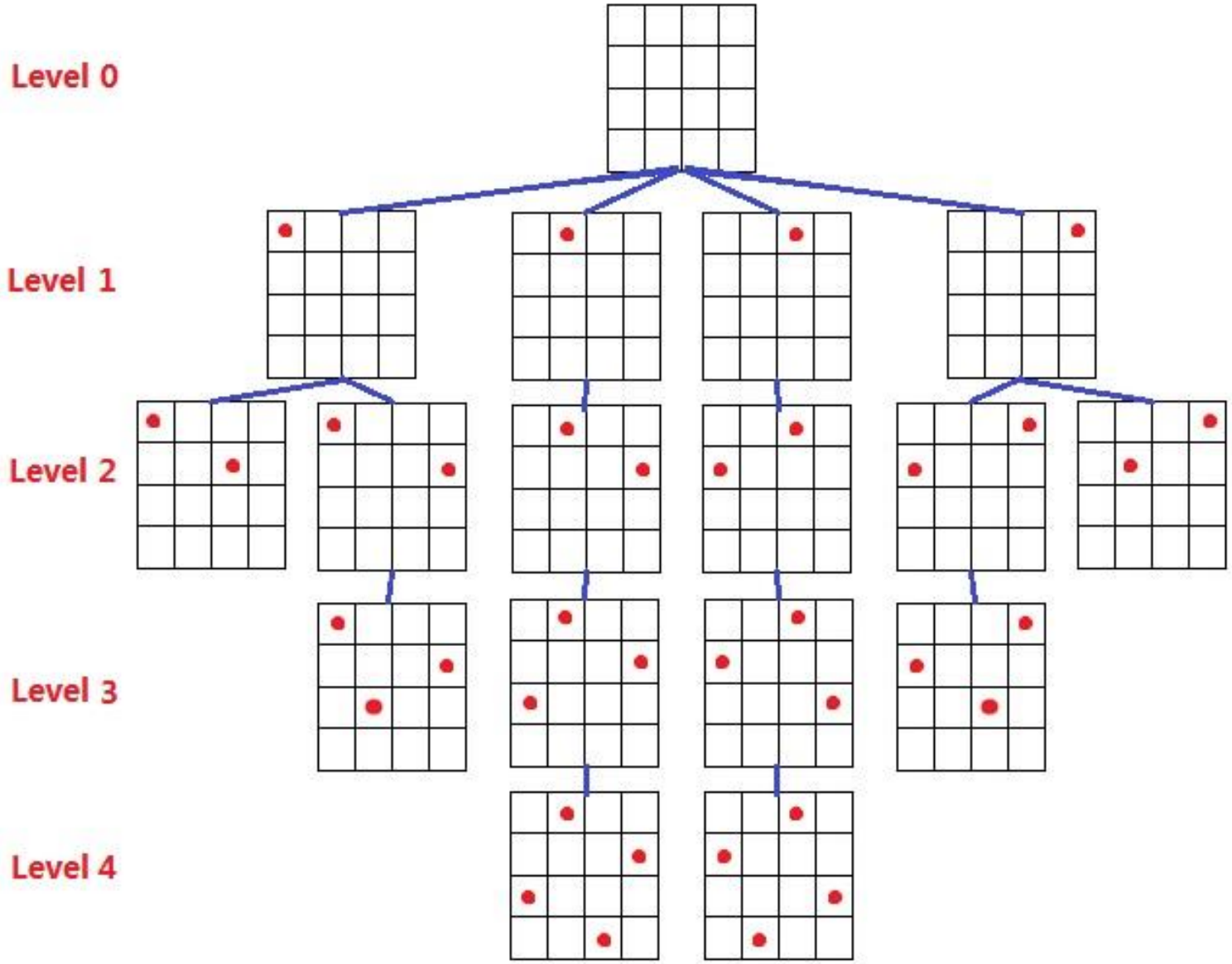
用回溯算法解决问题的一般步骤：

1. 针对所给问题，定义问题的解空间，它至少包含问题的一个（最优）解。
2. 确定易于搜索的解空间结构，使得能用回溯法方便地搜索整个解空间 。
3. 以深度优先的方式搜索解空间，并且在搜索过程中用约束函数剪枝解空间，避免无效搜索。

问题的解空间通常是在搜索问题解的过程中动态产生的，这是回溯算法的一个重要特性。

例1：四皇后问题

棋盘状态树

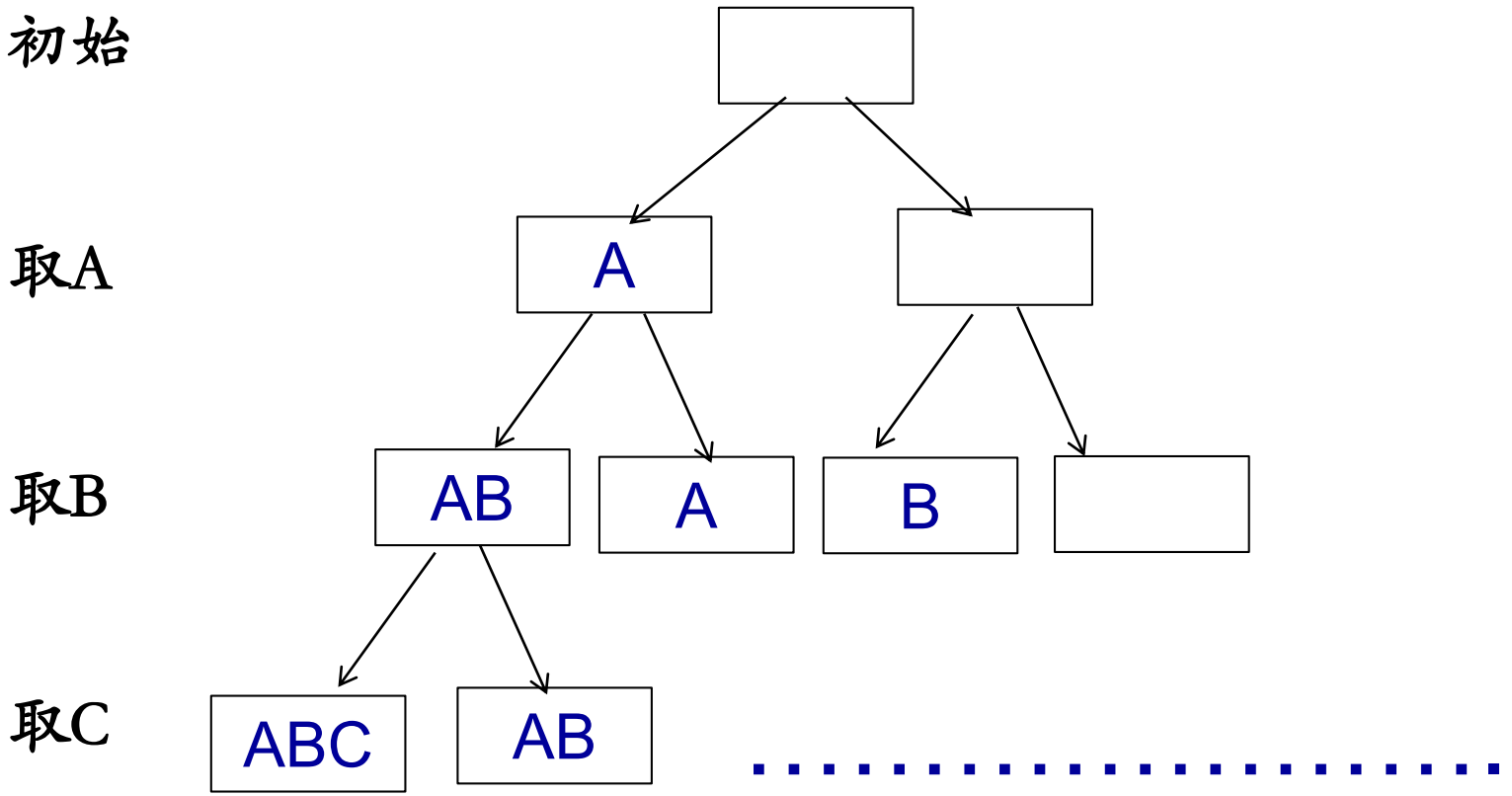


约束条件(函数)：任何2个皇后不放在同一行或同一列或同一斜线上。

```
void Trial(int i, int n) {
 //进入本函数时，在n×n棋盘前i-1行已放置了互不攻击的i-1个棋子。现从第 i 行起继续为后续棋子选择满足约束条//件的位置。当求得(i>n)的一个合法布局时，输出之。
    if (i>n) 输出棋盘的当前布局;
    else     for (j=1; j<=n; ++j) {
                在第 i 行第 j 列放置一个棋子;
                if (当前布局合法) Trial(i+1, n);
                移去第 i 行第 j 列的棋子;
             }
} // trial
```

例2：n个元素的幂集

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



总结

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算法，灵活运用遍历算法实现二叉树的其它操作。

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，因此读者应掌握 1 至 2 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。