第六章 树和二叉树

- 树的概念和基本术语
- 二叉树
- 二叉树遍历
- 二叉树的计数
- 树与森林
- 霍夫曼树

树的概念和基本术语

树的定义

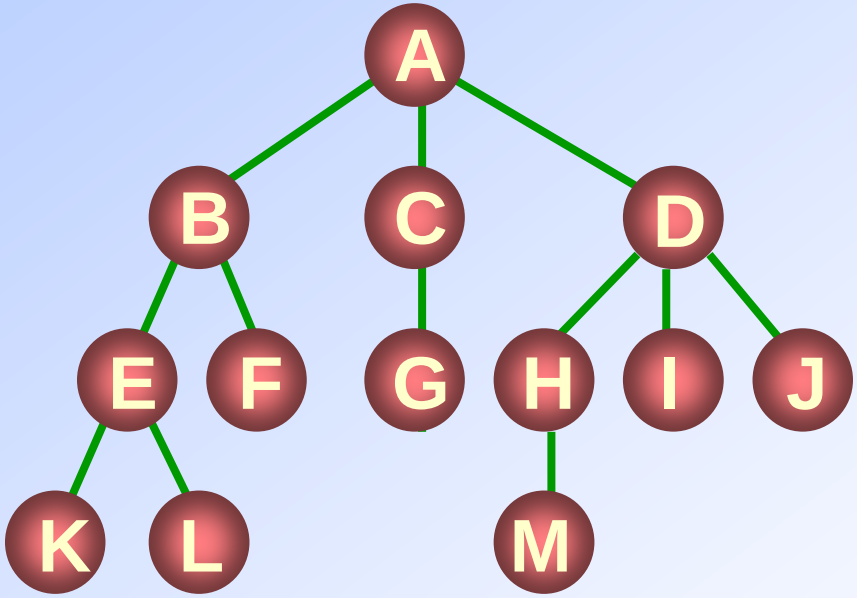
树是由 n ($n \geq 0$) 个结点的有限集合。如果 $n = 0$，称为空树；如果 $n > 0$，则

- 有且仅有一个特定的称之为根(Root)的结点；

- 当$n > 1$，除根以外的其它结点划分为 m ($m > 0$) 个互不相交的有限集 $T_1, T_2, \ldots, T_m$，其中每个集合本身又是一棵树，并且称为根的子树(SubTree)。

例如

只有根结点的树



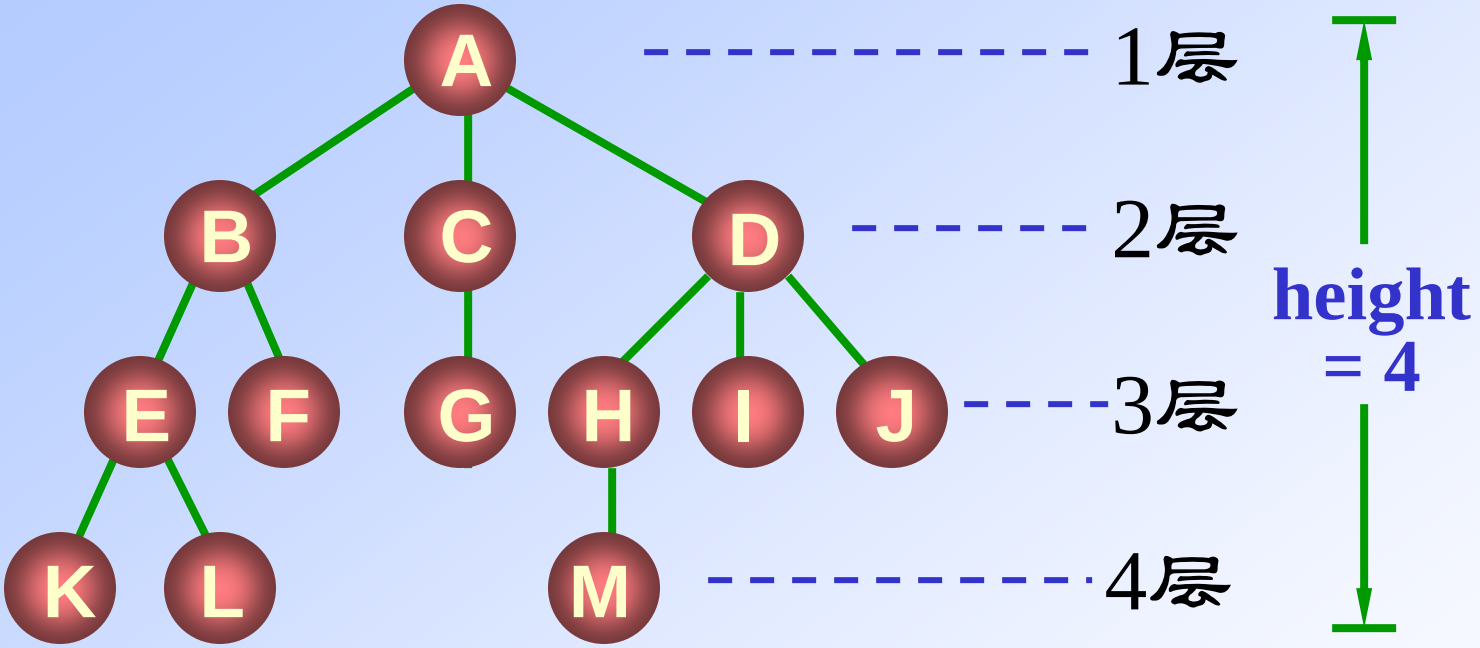
有13个结点的树

其中：A是根；其余结点分成三个互不相交的子集，

T1={B,E,F,K,L}； T2={C,G}； T3={D,H,I,J,M},

T1,T2,T3都是根A的子树，且本身也是一棵树

树的基本术语



- 结点
- 结点的度
- 叶结点
- 分支结点
- 子女
- 双亲
- 兄弟
- 祖先
- 子孙
- 结点层次
- 树的深度
- 树的度
- 森林

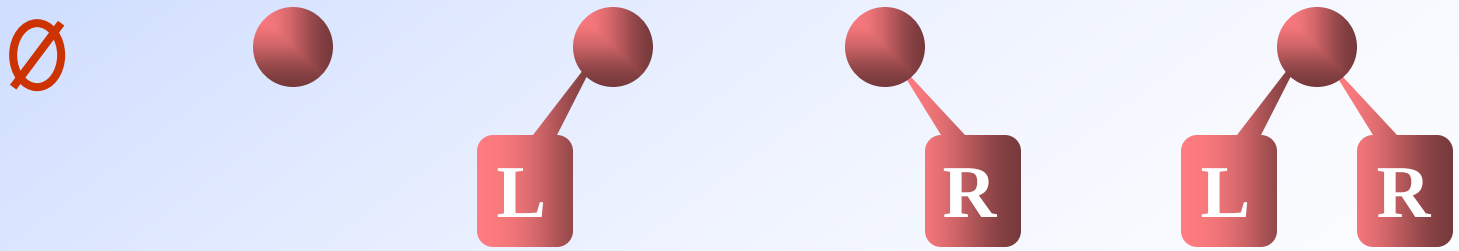
- 结点(node)：数据元素及其分支
- 结点的度(degree)：结点拥有的子树的个数
- 树的度(degree)：树中结点的度的最大值
- 分支(branch)结点：度不为0的结点
- 叶子(leaf)结点：度为0的结点
- 子(child)结点：结点子树的根
- 双亲(parent)结点：子结点的直接前驱结点
- 兄弟(sibling)结点：同一双亲的子结点互称兄弟结点
- 结点的层次(level)：根为第一层；子结点的层次比双亲结点的层次加1
- 树的深度(depth)：树中结点的最大层次
- 有序树：子树从左到右有序
- 无序树：子树无序
- 森林(forese)：m（　0）棵互不相交的树的集合

二叉树 (Binary Tree)

定义　一棵二叉树是结点的一个有限集合，该集合或者为空，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。

特点　每个结点至多只有两棵子树（二叉树中不存在度大于2的结点）

五种形态



性质1　　在二叉树的第 i 层上至多有 2^{i-1} 个结点。($i \geq 1$) [证明用归纳法]

证明：当i=1时，只有根结点，$2^{i-1}=2^0=1$。

假设对所有j，i>j≥1，命题成立，即第j层上至多有2^{j-1} 个结点。

由归纳假设第i-1 层上至多有 2^{i-2}个结点。

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$2* 2^{i-2}= 2^{i-1}$。

性质2　　深度为 k 的二叉树至多有 2^k-1 个结点($k \geq 1$)。

证明：由性质1可见，深度为k的二叉树的最大结点数为

$$\sum_{i=1}^{k}(\text{第}i\text{层上的最大结点数})$$

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

性质3　　对任何一棵二叉树T, 如果其叶结点数为 n_0, 度为2的结点数为 n_2,则$n_0 = n_2 + 1$.

证明：若度为1的结点有 n_1 个，总结点个数为 n，总边数为 e，则根据二叉树的定义，

$$n = n_0 + n_1 + n_2 \qquad e = 2n_2 + n_1 = n - 1$$

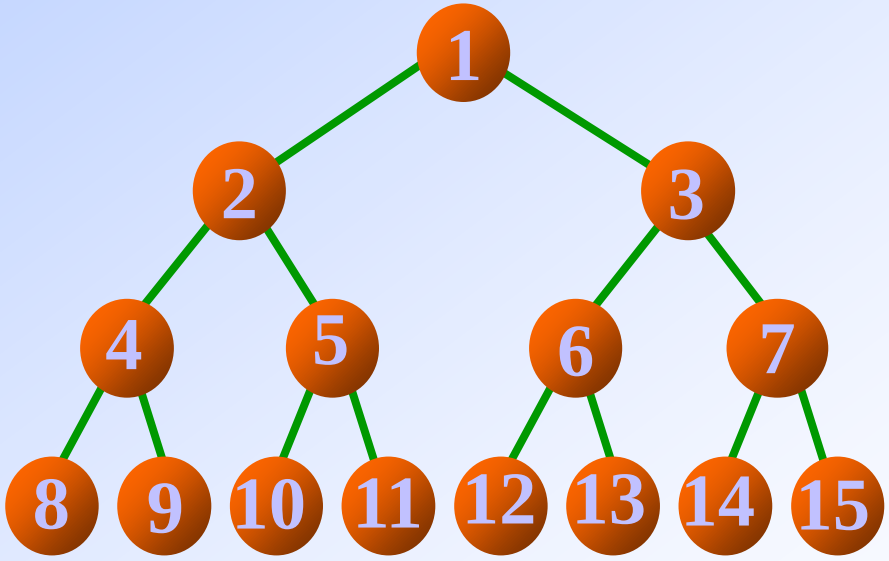
因此，有 $2n_2 + n_1 = n_0 + n_1 + n_2 - 1$

$$n_2 = n_0 - 1 \rightarrow n_0 = n_2 + 1$$

两种特殊形态的二叉树

定义1 满二叉树 (Full Binary Tree)

一棵深度为k且有2^k-1个结点的二叉树称为满二叉树。

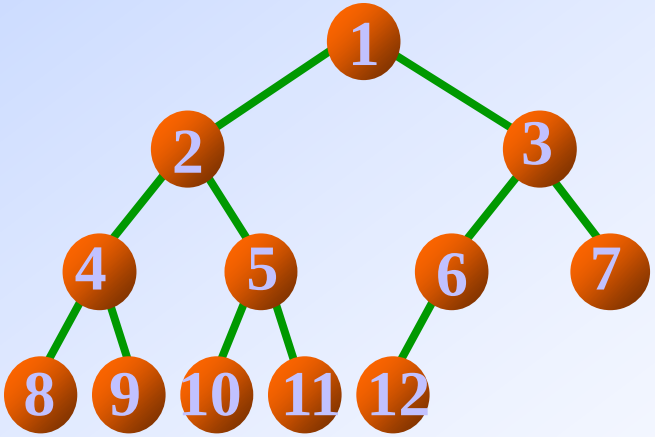


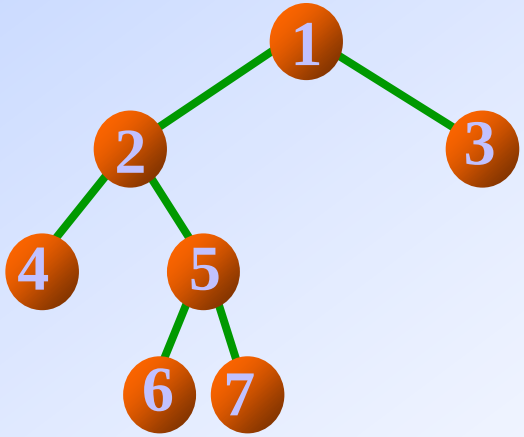
满二叉树

定义2 完全二叉树 (Complete Binary Tree)

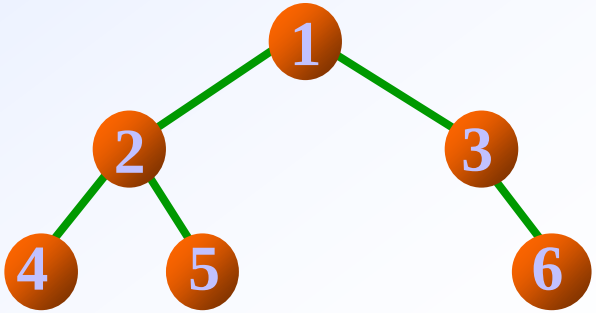
若设二叉树的高度为h，则共有h层。除第 h 层外，其它各层 (1 ~ h−1) 的结点数都达到最大个数，第 h 层从右向左连续缺若干结点，这就是完全二叉树。

完全二叉树 →





非完全二叉树



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

证明：

设完全二叉树的深度为 h，则根据性质2和完全二叉树的定义有

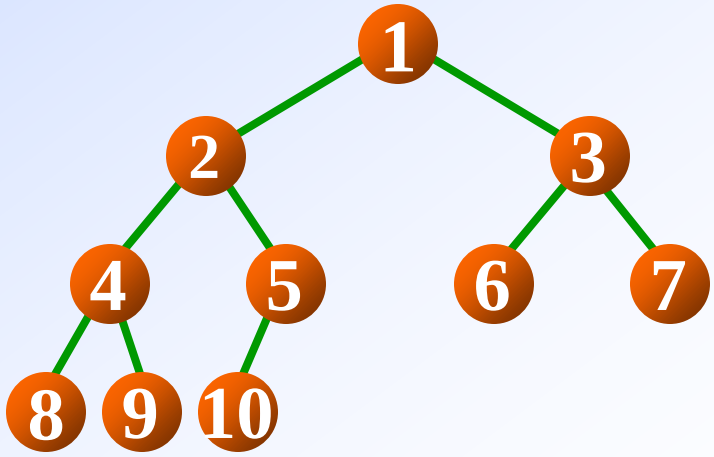
$$2^{h-1} - 1 < n \leq 2^h - 1 \text{或} 2^{h-1} \leq n < 2^h$$

取对数 $h - 1 \leq \log_2 n < h$，**又h是整数，**

因此有 $h = \lfloor \log_2(n) \rfloor + 1$

性质5　对于有n个结点的完全二叉树中的所有结点按从上到下，从左到右的顺序进行编号，则对任意一个结点i （1≤i≤n），都有：

- （1）如果i=1，则结点i是这棵完全二叉树的根，没有双亲；否则其双亲结点的编号为⌊i/2⌋。
- （2）如果2i>n，则结点i没有左孩子；否则其左孩子结点的编号为2i。
- （3）如果2i+1>n，则结点i没有右孩子；否则其右孩子结点的编号为2i+1。

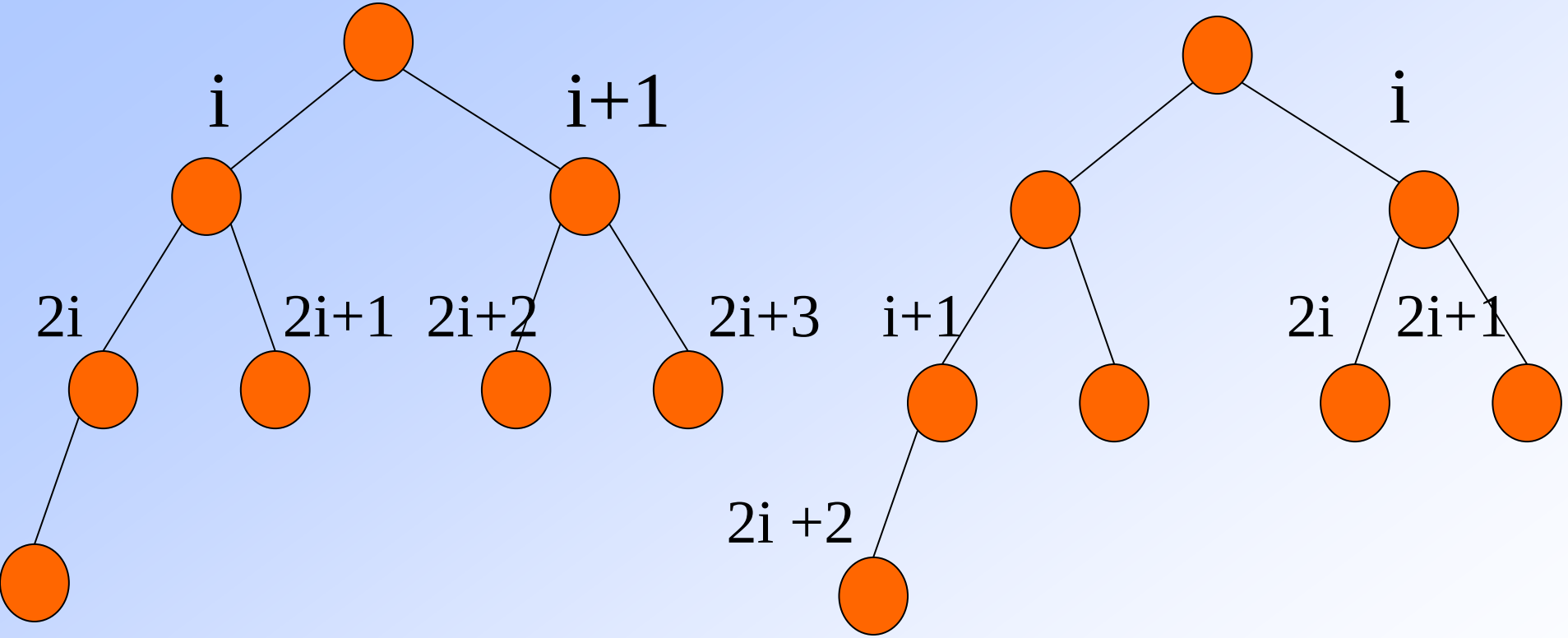


下面我们利用数学归纳法证明【性质5】

我们首先证明（2）和（3）。

当i=1时，若n≥3，则根的左、右孩子的编号分别是2，3；若n<3，则根没有右孩子；若n<2，则根将没有左、右孩子；以上对于（2）和（3）均成立。

假设：对于所有的1≤j≤i 结论成立。即：结点j的左孩子编号为2j；右孩子编号为2j+1。



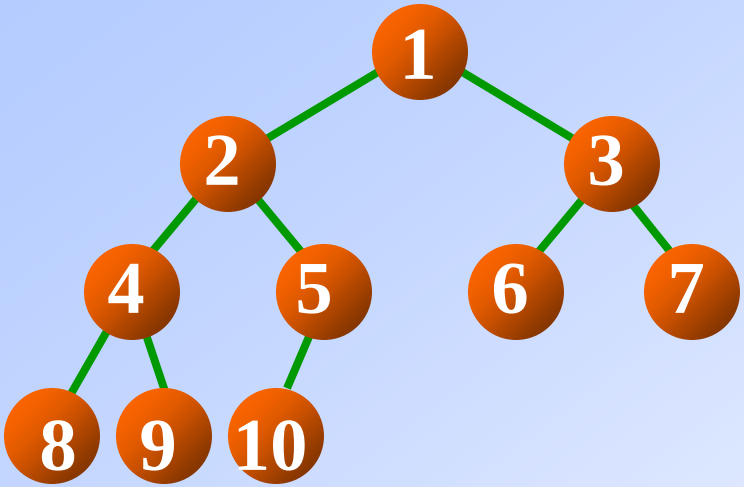
•由完全二叉树的结构可以看出：结点i+1或者与结点i同层且紧邻i结点的右侧，或者i位于某层的最右端，i+1位于下一层的最左端。

- 可以看出，i+1的左、右孩子紧邻在结点i的孩子后面，由于结点i 的左、右孩子编号分别为2i和2i+1，所以，结点i+1的左、右孩子编号分别为2i+2和2i+3，经提取公因式可以得到：2(i+1)和2(i+1)+1，即结点i+1的左孩子编号为2(i+1)；右孩子编号为2(i+1)+1。
- 又因为二叉树由n个结点组成，所以，当2(i+1)+1>n，且2(i+1)=n时，结点i+1只有左孩子，而没有右孩子；当2(i+1)>n，结点i+1既没有左孩子也没有右孩子。

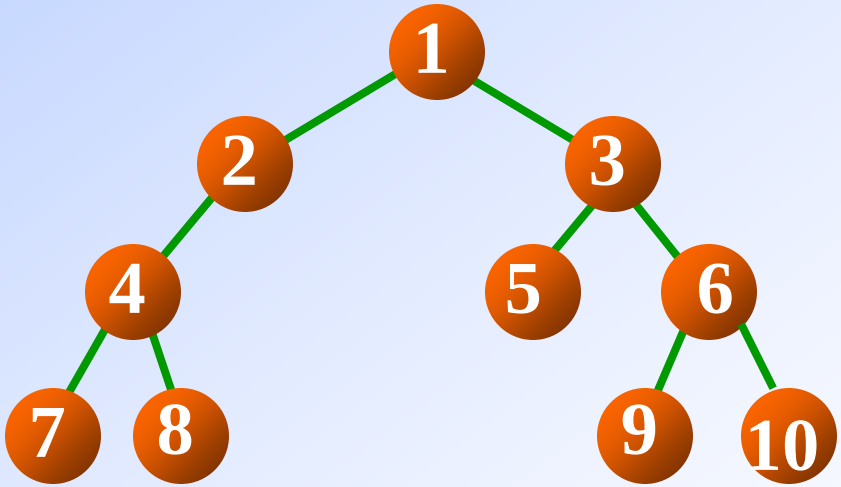
一棵野生的二叉树

二叉树的存储结构

- 顺序表示



完全二叉树
的顺序表示

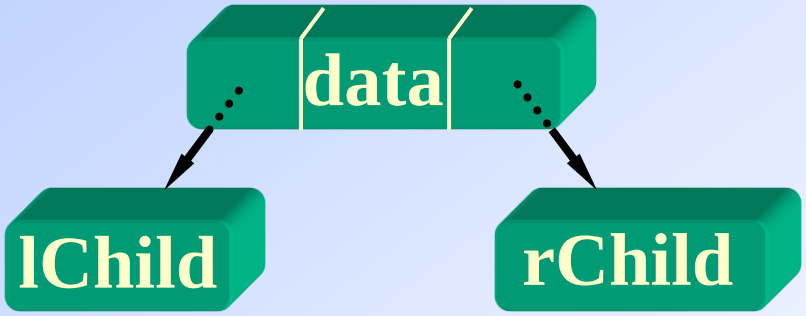


一般二叉树
的顺序表示

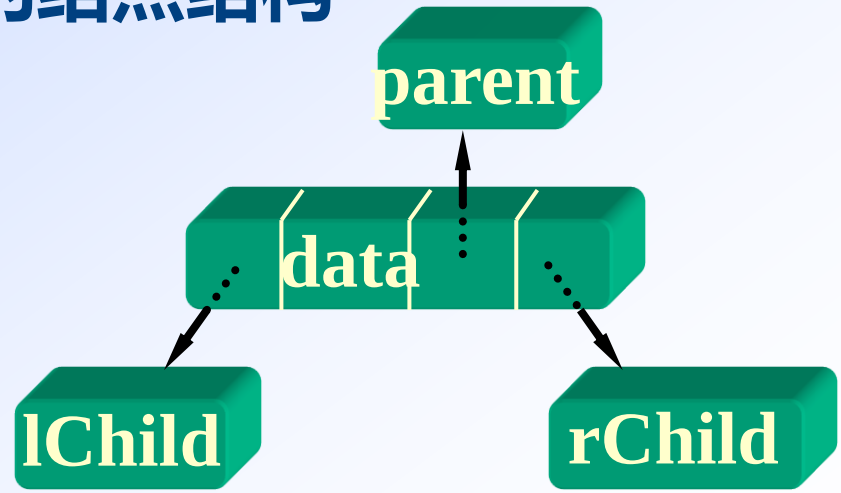
- 链表表示

含两个指针域的结点结构

含三个指针域的结点结构

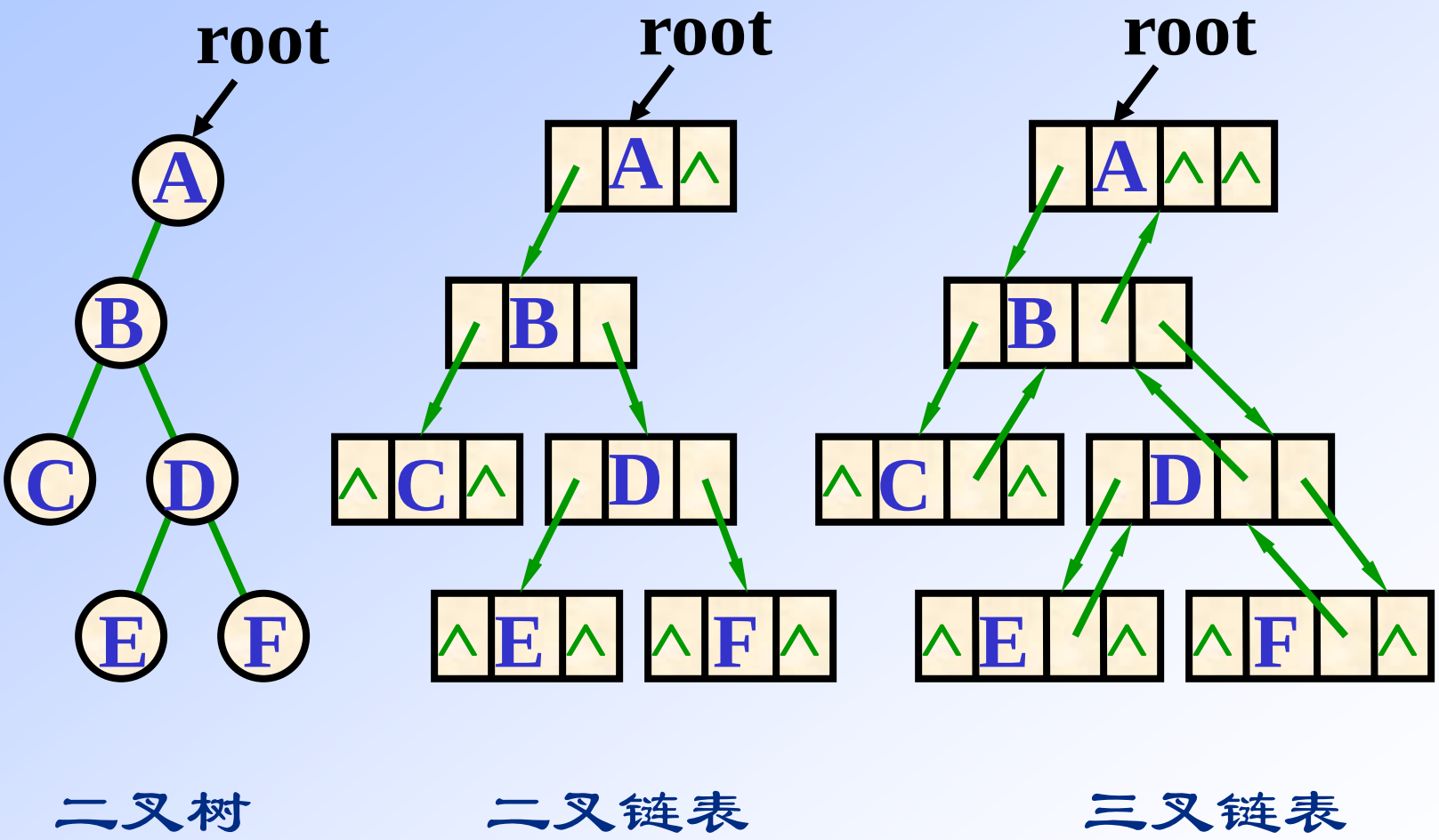


二叉链表



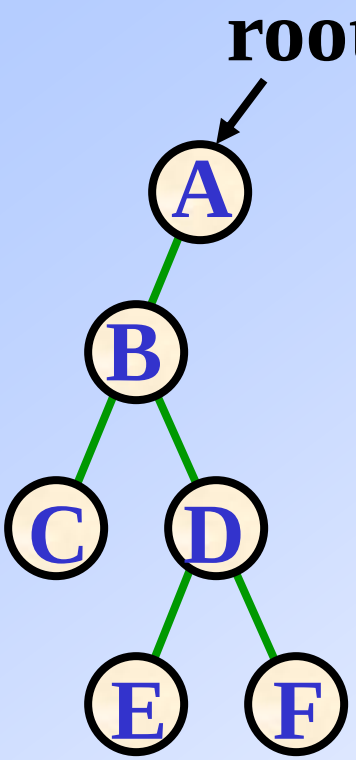
三叉链表

二叉树链表表示的示例



二叉树　　二叉链表　　三叉链表

三叉链表的静态结构



	data	parent	leftChild	rightChild
0	A	-1	1	-1
1	B	0	2	3
2	C	1	-1	-1
3	D	1	4	5
4	E	3	-1	-1
5	F	3	-1	-1

二叉链表的定义

```
typedef char TreeData;  //结点数据类型

typedef struct node {      //结点定义
    TreeData data;
    struct node * leftchild, * rightchild;
} BinTreeNode;

typedef BinTreeNode * BinTree;
                          //根指针
```

练习：

1：对一棵具有n个结点的树，其中所有度之和是多少？ n-1

2：一棵度为4的树T中，若有20个度为4的结点，10个度为3的结点，1个度为2的结点，10个度为1的结点，则树T的叶子结点个数是多少？ 82

$20+10+1+10+n_0=n$

$20*4+10*3+1*2+10*1=n-1$

练习：

1：一棵完全二叉树上有768个结点，则该二叉树中叶子结点的个数是多少？ 384

$$768= n_0 + n_1 + n_0 - 1$$

2：已知一棵完全二叉树的第6层有8个叶子结点，则该完全二叉树的结点个数最多是多少？ 111

分析：前5层共有2^5-1个结点。第6层最多有32个结点，其中有8个叶子结点。第7层有（32-8）*2=48个结点。

二叉树遍历

树的遍历就是按某种次序访问树中的结点，要求每个结点访问一次且仅访问一次。

设访问根结点记作 V

遍历根的左子树记作 L

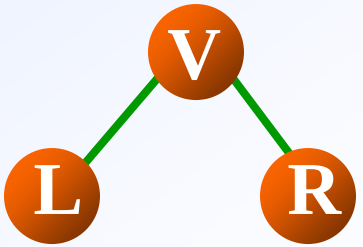
遍历根的右子树记作 R

则可能的遍历次序有

前序　VLR

中序　LVR

后序　LRV



- 线形表的遍历

（1）顺序表的遍历：

```
for(i=1; i<=v.last; i++)    visit(v.elem[i]);
```

（2）链表的遍历：

```
for(p=L->next; p!=NULL; p=p->next)    visit(p->data);
```

二叉树的遍历：非线形关系，需确定先后次序。

- 通常按对根结点的处理次序分为：

先序（DLR）、中序（LDR）、后序（LRD）。

–先序遍历二叉树(DLR)的操作定义:

(1)访问根结点;

(2)先序遍历左子树

(3)先序遍历右子树

–中序遍历二叉树(LDR)的操作定义:

(1)中序遍历左子树;

(2)访问根结点;

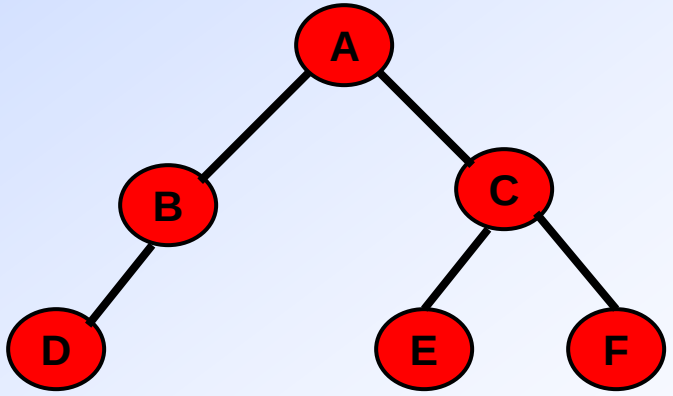
(3)中序遍历右子树;

–后序遍历二叉树(LRD)的操作定义:

(1)后序遍历左子树;

(2)后序遍历右子树;

(3)访问根结点;



先序：A B D C E F

中序：D B A E C F

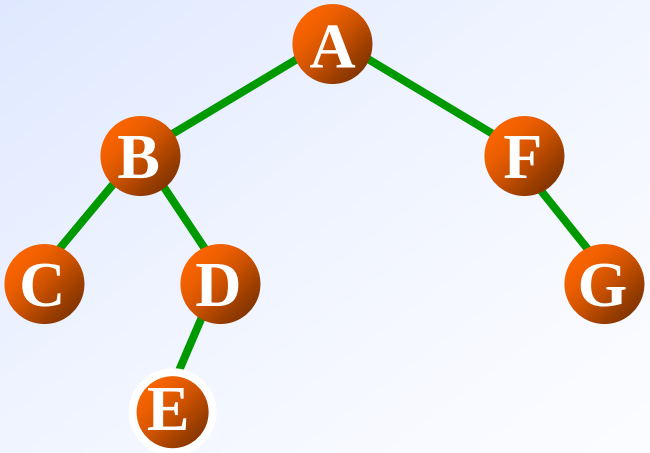
后序：D B E F C A

练习：已知结点的前序序列和中序序列分别为：

前序序列：ABCDEFG

中序序列：CBEDAFG

请构造整棵二叉树。



练习：若一棵二叉树的先序遍历序列和后序 遍历序列分别是1,2,3,4和4,3,2,1，则该二叉树的中序遍历序列不会是　　c

(a)1,2,3,4　(b)2,3,4,1

(c)3,2,4,1　(d)4,3,2,1

中序遍历 (Inorder Traversal)

- **中序遍历二叉树算法的定义：**

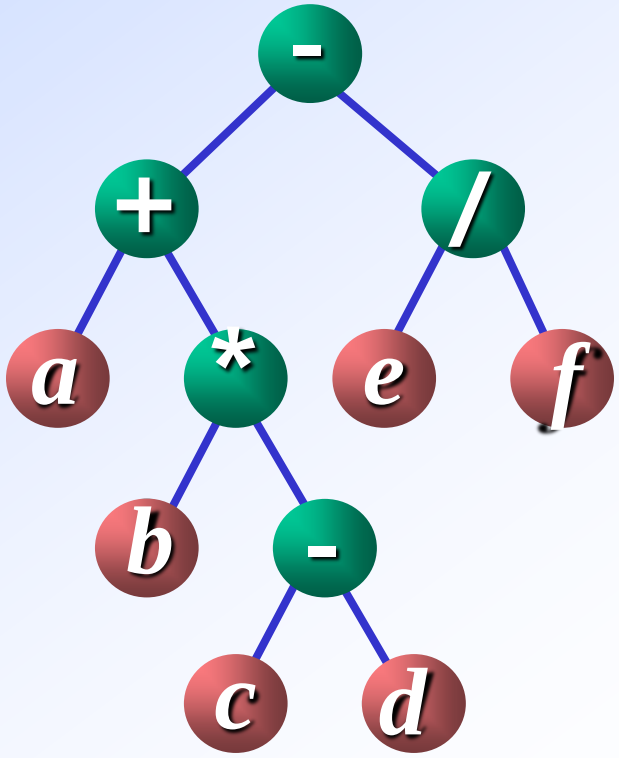
  若二叉树为空，则空操作；

  否则

  中序遍历左子树 (L)；

  访问根结点 (V)；

  中序遍历右子树 (R)。

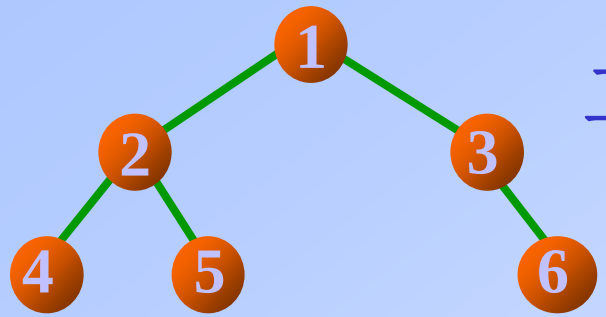


遍历结果

a + b * c - d - e / f

- **中序遍历二叉树的递归算法**

```
void InOrder ( BinTreeNode *T )
{
    if ( T != NULL ) {
        InOrder ( T->leftchild );
        Visit (T->data);
        InOrder ( T->rightchild );
    }
}
```



二叉树中序遍历：4 2 5 1 3 6

{ 遍历左子树;
打印1;
遍历右子树; }

{ 遍历左子树;
打印2;
遍历右子树;}

{左子树空;
打印3;
遍历右子树;}

{左子树空;
打印4;
右子树空;}

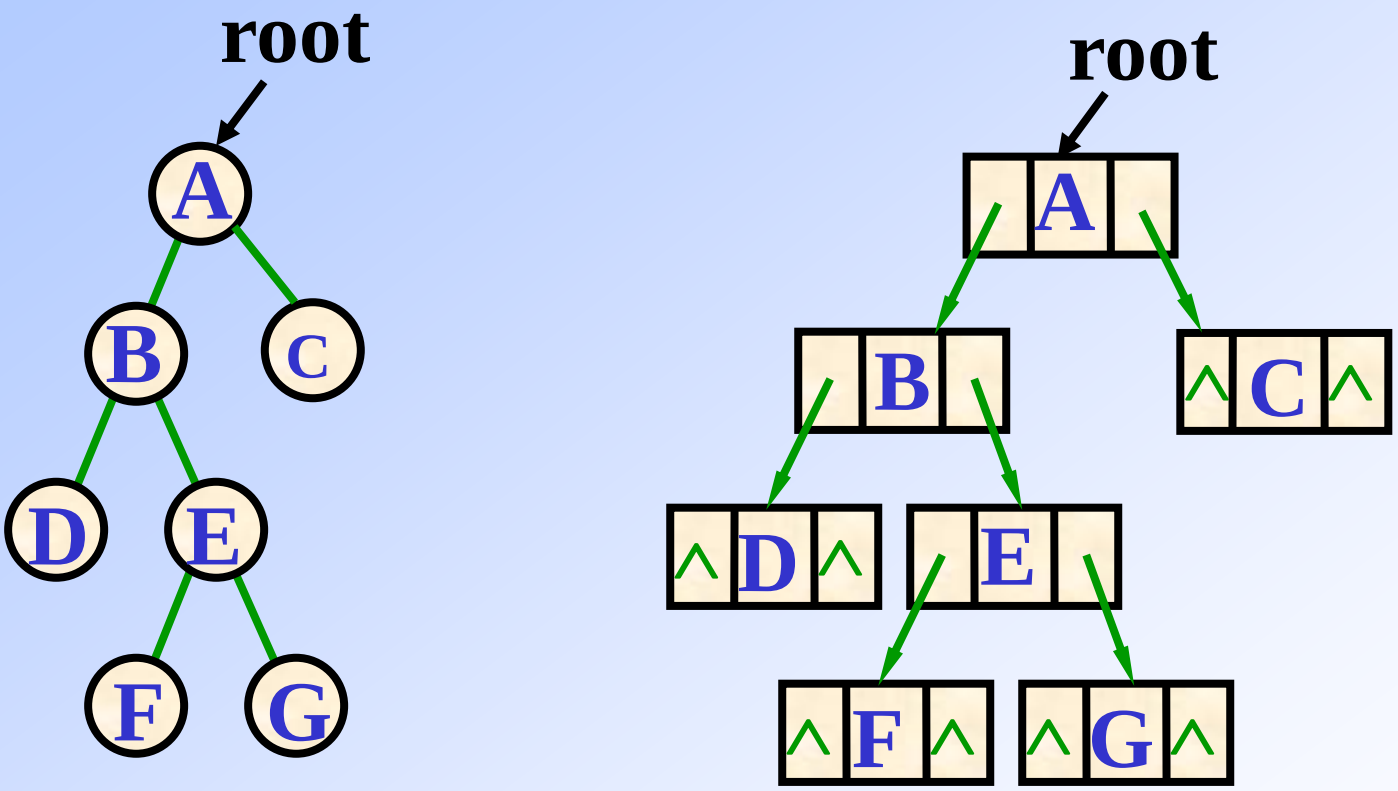
{左子树空;
打印5;
右子树空;}

{左子树空;
打印6;
右子树空;}

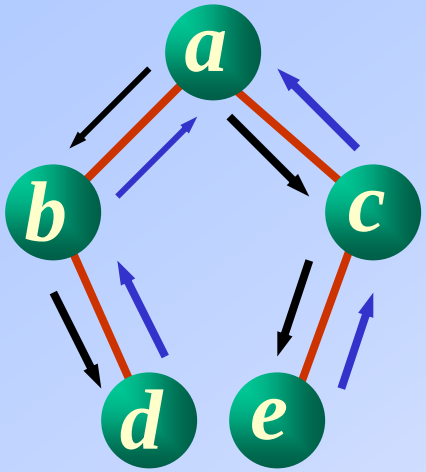
2中序遍历算法（非递归算法）：

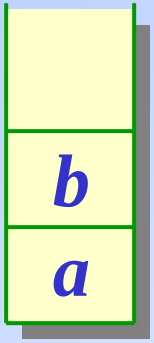
```
void inorder(BiTree T)
{  InitStack(S);
    p=T;
    while(p||!SatckEmpty(S))
    { if (p) { Push(p);
                p=p->lchild;
              }
       else  { Pop(p);
               printf(p->data);
                p=p->rchild);
             }
    }
}
```

二叉树中序遍历：DBFEGAC

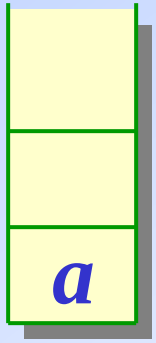


中序遍历二叉树(非递归算法)用栈实现

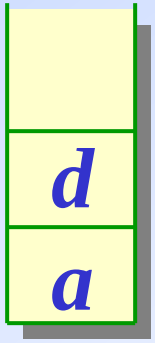




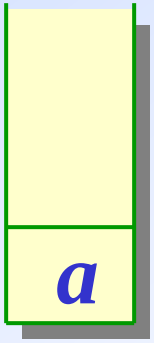
a b入栈



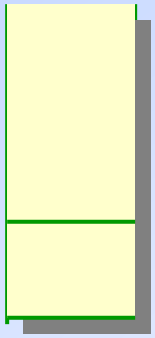
b退栈
访问



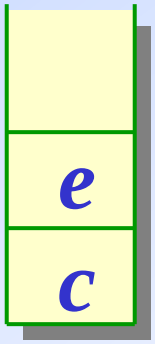
*d*入栈



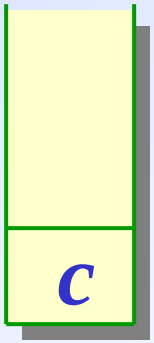
*d*退栈
访问



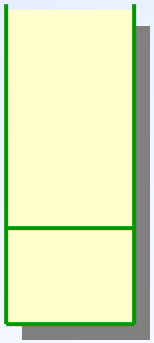
*a*退栈
访问



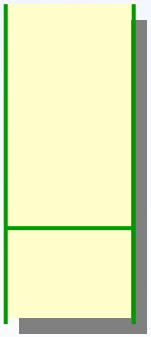
c e 入栈



e 退栈
访问



c 退栈
访问



栈空

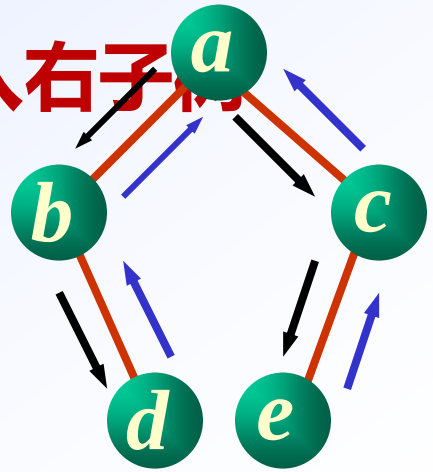
二叉链表的定义

```
typedef struct node {
    TreeData data;
    struct node * leftchild, * rightchild;
} BinTreeNode;

typedef BinTreeNode * BinTree;
```

顺序栈的类型表示:

```
typedef bintree StackData;
typedef struct {          //顺序栈定义
   StackData *base; //栈底指针
    StackData *top; //栈顶指针
    int stacksize;  //当前已分配的存储空间
} SeqStack
```

```
void InOrder ( BinTree T ) {
 BinTreeNode *p = T;          //初始化
SeqStack *S=(SeqStack*)malloc(sizeof(SeqStack));
InitStack(S );    //递归工作栈

 while ( p != NULL || !StackEmpty(S) ) {
        if( p != NULL )
          { Push(s, p);   p = p->leftchild; }
       else
          {Pop(S, &p); /退栈//访问根,进入右子树
          visit( p->data );
          p = p->rightchild;
          }//else
   }//while
}
```



前序遍历 (Preorder Traversal)

- **前序遍历二叉树算法的定义：**

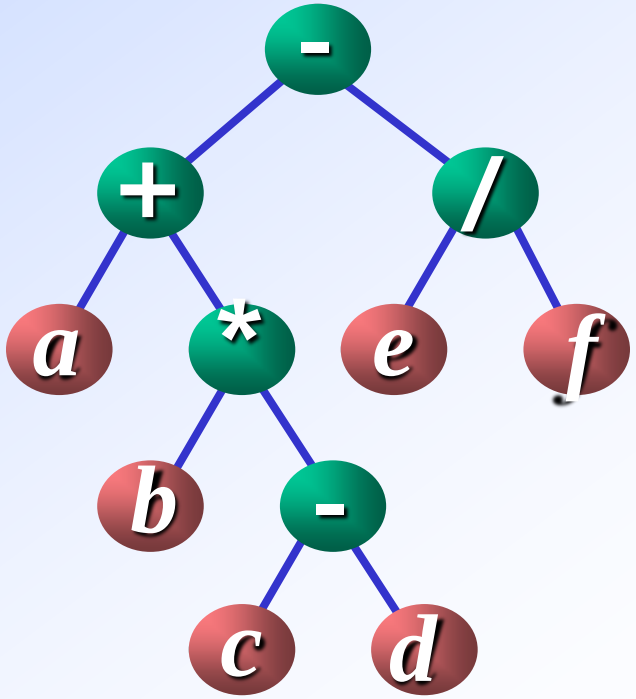
  若二叉树为空，则空操作；

  否则

  访问根结点 (V)；

  前序遍历左子树 (L)；

  前序遍历右子树 (R)。



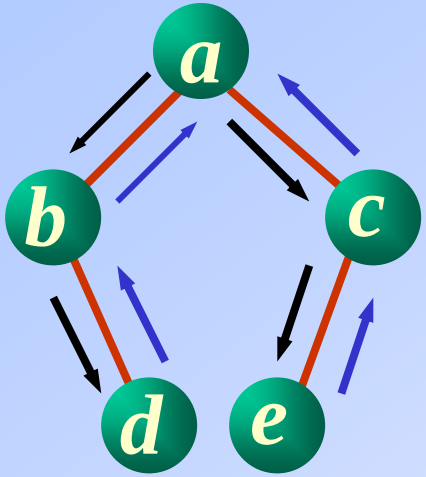
遍历结果

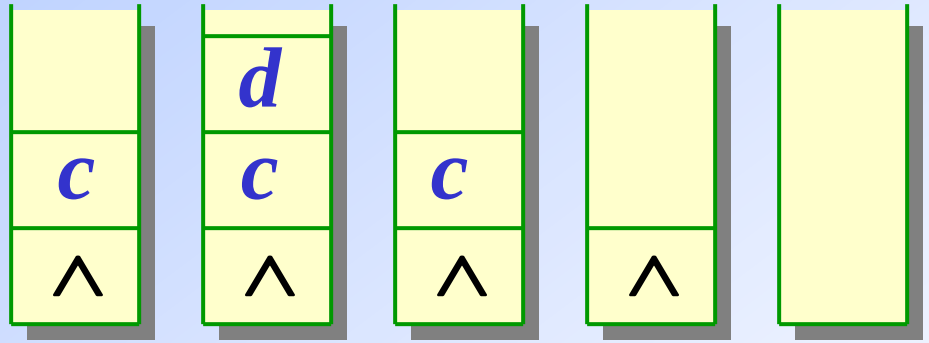
- + a * b - c d / e f

- **前序遍历二叉树的递归算法**

```
void PreOrder ( BinTreeNode *T )
{
    if ( T != NULL ) {
        Visit ( T->data);
        PreOrder ( T->leftchild );
        PreOrder ( T->rightchild );
    }
}
```

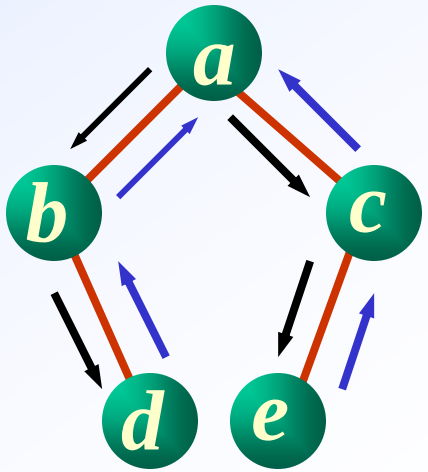
前序遍历二叉树(非递归算法)用栈实现





访问	访问	退栈	退栈	访问	
a	*b*	*d*	*c*	*e*	
进栈	进栈	访问	访问	左进	
c	*d*	*d*	*c*	*空*	
左进	左进	左进	左进	退栈	结束
b	*空*	*空*	*e*	^	

```
void PreOrder( BinTree T ) {
BinTreeNode * p = T;
SeqStack * S=(SeqStack*)malloc(sizeof(SeqStack));
InitStack(S ); //递归工作栈
Push (S, NULL);
    while ( p != NULL ) {
         visit( p->data );
        if ( p->rightChild != NULL )
            Push (S, p->rightChild );
        if  ( p->leftChild != NULL )
            p = p->leftChild;//进左子树
        else Pop(S, &p );
    }
}
```



后序遍历 (Postorder Traversal)

- **后序遍历二叉树算法的定义：**

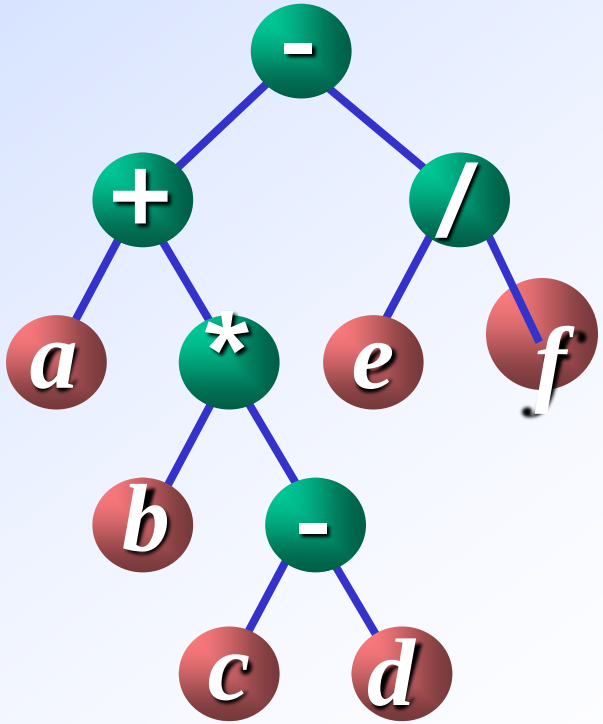
  若二叉树为空，则空操作；

  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 / -

- **后序遍历二叉树的递归算法：**

```
void PostOrder ( BinTreeNode * T ) {
    if ( T != NULL ) {
        PostOrder ( T->leftchild );
        PostOrder ( T->rightchild );
        Visit ( T->data);
    }
}
```

3 二叉树算法举例：

```
int TreeEqual(BiTree T1, BiTree T2)
//如果两树相等，返回1，否则返回0（先序）
{  if (!T1 && !T2)  return(1);
    else if ( T1 && T2)
    {  if (T1->data == T2->data)
            if (TreeEqual(T1->lchild, T2->lchild))
                return (TreeEqual(T1->rchild, T2->rchild));
    }
    return(0);
}
```

```
BiTree TreeCopy(BiTree T) //二叉树复制（先序）
{ if (T)
  { p=(BiTree)malloc(sizeof(BiTNode));
     p->data=T->data;
     p->lchild=TreeCopy(T->lchild);
     p->rchild=TreeCopy(T->rchild);
     return(p);
  }
  else return(NULL);
}
```

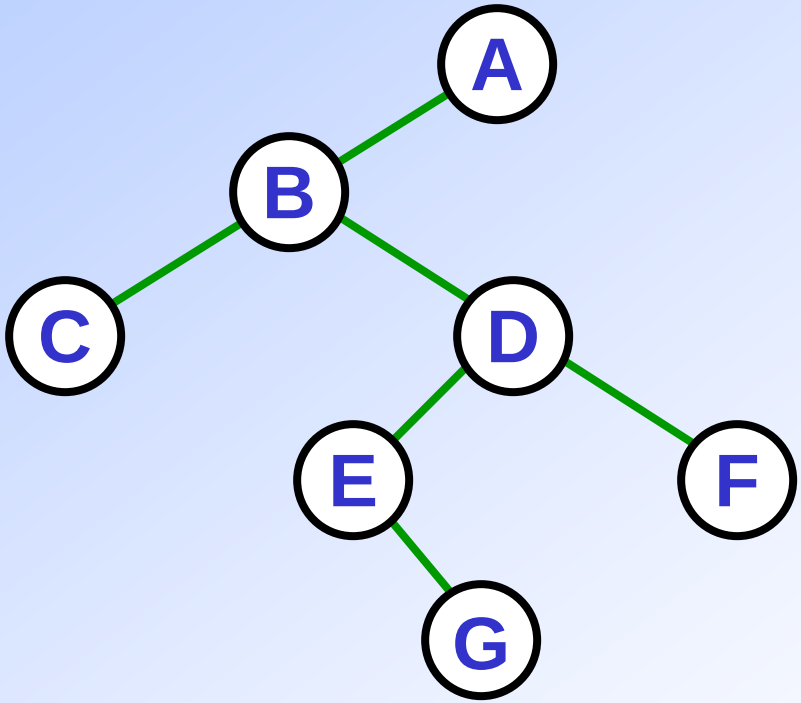
二叉树遍历应用

1. 按前序建立二叉树(递归算法)

- 以递归方式建立二叉树。
- 输入结点值的顺序必须对应二叉树结点前序遍历的顺序。并约定以输入序列中不可能出现的值作为空结点的值以结束递归，此值在RefValue中。例如用“@”或用“-1”表示字符序列或正整数序列空结点。

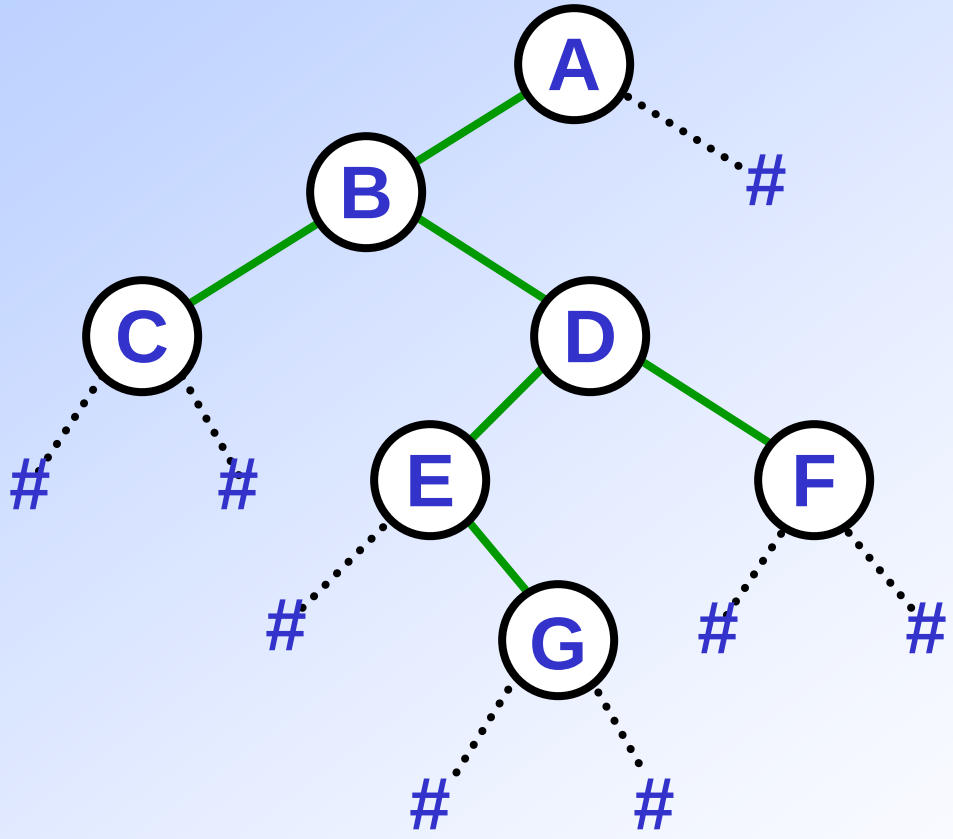
如图所示的二叉树的前序遍历顺序为

A B C D E G F



如图所示的二叉树的前序遍历顺序为

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



建立二叉树的主程序（函数）

```
BinTreeNode * Crt_BinTree ( )
 {char ch; BinTreeNode *t;
   printf("please input data:\n");
     scanf ("%c",&ch);
    if (ch=='#')   t = NULL ;
     else {
         t=(BinTreeNode *) malloc(sizeof(BinTreeNode));
         t->data = ch;
             t->leftchild=Crt_BinTree ( );
            t->rightchild=Crt_BinTree ( );
        }
    return (t);
}

Main()
{BinTreeNode *t;
 t=Crt_BinTree ( );
}
```

建立二叉树的主程序（过程）

```
void Crt_BinTree (BinTreeNode *  * T ) {
    char ch; scanf (“%c”,&ch);
    if (ch==‘ # ’)   *T == NULL ;
     else {
       (*T)=(BinTreeNode *) malloc(sizeof(BinTreeNode ))))
        (*T)->data = ch;
        Crt_BinTree (&((*T)->leftchild) );
        Crt_BinTree (&((*T)->rightchild) );
       }
}

Main()
{BinTreeNode *t;
  Crt_BinTree (&t);
}
```

2. 计算二叉树结点个数(递归算法)

```
int Count ( BinTreeNode *T ) {
    if ( T == NULL ) return 0;
    else return 1 + Count ( T->leftchild )
                  + Count ( T->rightchild );
}
```

3. 求二叉树中叶子结点的个数

```
int Leaf_Count(BinTree T)
 {//求二叉树中叶子结点的数目
    if(!T) return 0; //空树没有叶子
      else if(!T->leftchild&&!T->rightchild)   return 1;
                  //叶子结点
     else return Leaf_Count(T-leftchild)+Leaf_Count(T-rightchild);
                 //    左子树的叶子数加上右子树的叶子数
}
```

4. 求二叉树高度(递归算法)

```
int Height ( BinTreeNode * T )
{
    if ( T == NULL ) return 0;
    else{
        int m = Height ( T->leftchild );
        int n = Height ( T->rightchild ) );
        return (m > n) ? m+1 : n+1;
        }
}
```

- 思考题1：假设二叉树采用二叉链表存储结构，试设计一个算法，计算一棵给定二叉树中数值为x的结点个数。

```
Int count(BinTreeNode * T ,elemtype x)
{if (T==NULL)
        return 0;
else if (T->data==x)
       return (1+count(T->leftchild,x)+ count(T->rightchild,x)
   else
     return (count(T->leftchild,x)+ count(T->rightchild,x));
```

- 思考题2：假设二叉树采用二叉链表存储结构，试设计一个算法，求先序遍历序列中第K个节点的值。

```
typedef struct BiTNode {
    TElemType data;
    BiTNode  *lchild, *rchild;
} BiTNode, *BiTree;
```

```
TElemType GetElemType(BiTree bt,int &num,TElemType &e){
   if(bt){
      if(num == 1){
                    e = bt -> data;
                    return 0;  }
       else{  if(bt -> lchild){
                          --num;
                        GetElemType(bt -> lchild, num, e);  }
              if(bt -> rchild){
                          --num;
                         GetElemType(bt -> rchild, num, e);  }
              }
          }
      }
TElemType PreOrder(BiTree bt, int k)
{     TElemType e;
       e = '#';
       GetElemType(bt, k, e);
       return e;
  }
```

6.3.2 线索二叉树

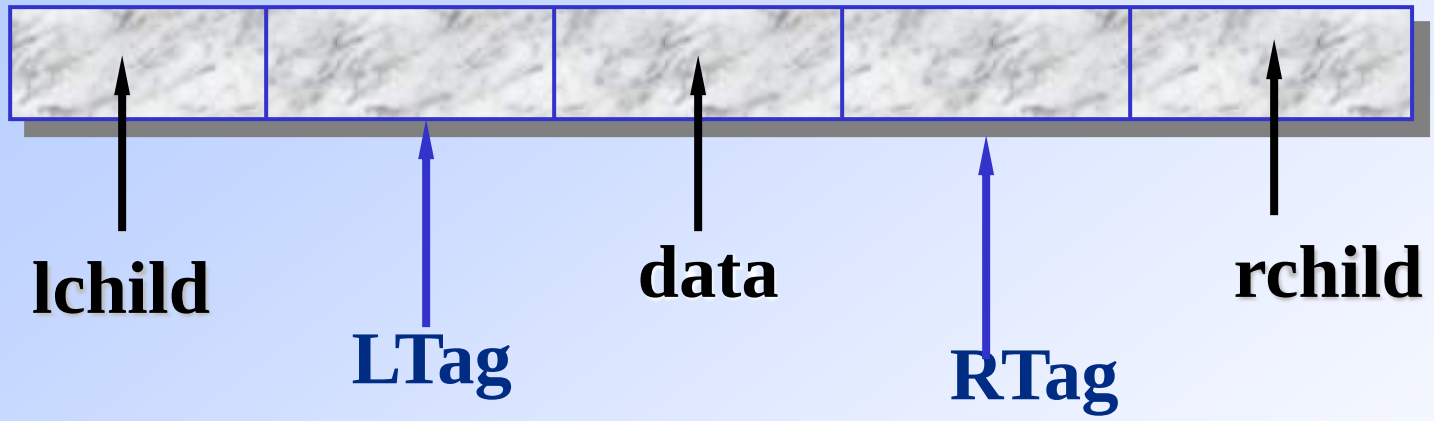
1 什么是线索二叉树

- 遍历的实质是对非线形结构的二叉树进行线形化处理。遍历序列中某结点的前驱、后继的位置只能在遍历的动态过程中得到。
- n个结点的二叉链表中有n+1个空链域，可以存放其前驱和后继的指针。
- 结点结构：

```
typedef struct BiThrNode
{  TELemType          data;
   struct BiThrNode *lchild;
   struct BiThrNode *rchild;
   unsigned              LTag:1;
   unsigned              RTag:1;
} BitThrNode, * BiThrTree;
```

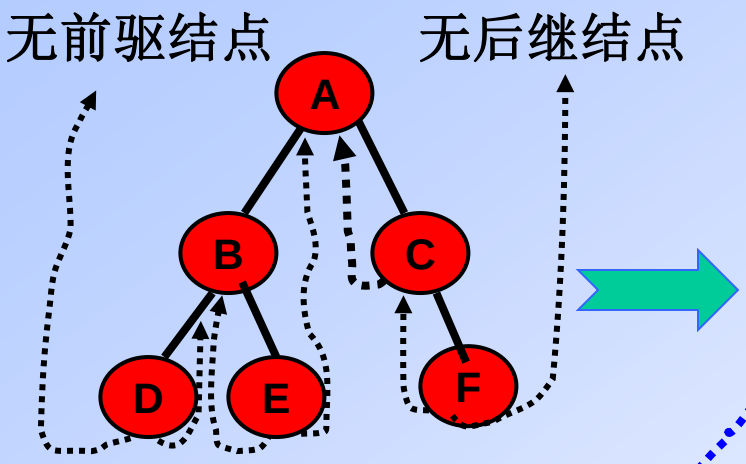
❖线索二叉树 (Threaded Binary Tree)

结点结构

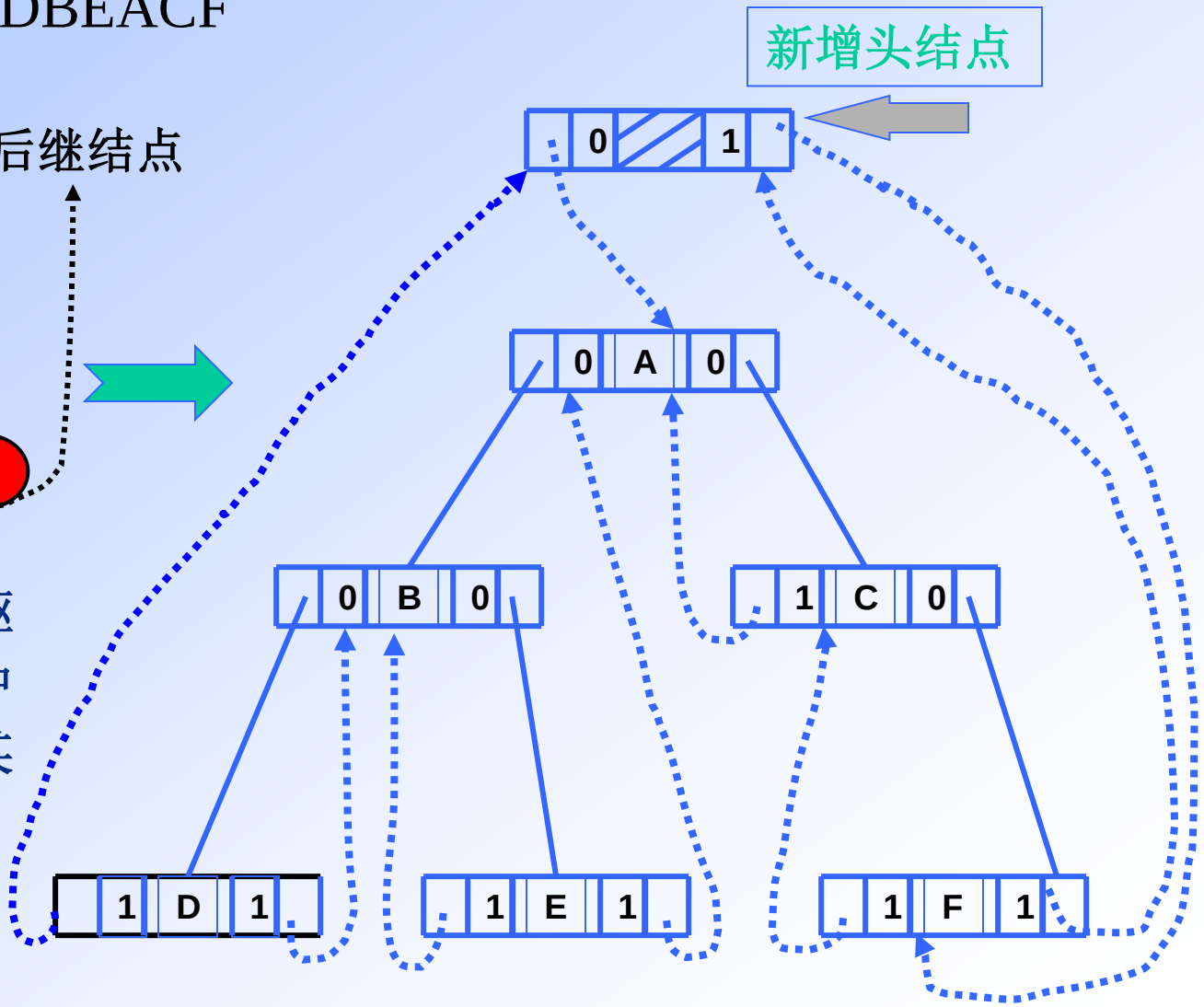


LTag=0, lchild为左子女指针
LTag=1, lchild为前驱线索
RTag=0, rchild为右子女指针
RTag=1, rchild为后继指针

中序遍历次序DBEACF



注：本小节中的前驱和后继指遍历序列中线形化逻辑关系中某结点的前驱和后继，不是其父子结点。



2 前驱与后继的查找（中序线索树）

```
BiThrNode *priornode(BiThrNode *p)
{ BiThrNode *pre=p->lchild;
  if(!p->LTag)              // 查找左子树的最右结点
     while(!pre->RTag)   pre=pre->rchild;
  return(pre);
}
```

```
BiThrNode *nextnode(BiThrNode *p)
{ BiThrNode *next=p->rchild;
  if(!p->RTag)              // 查找右子树的最左结点
     while(!next->LTag)  next=next->lchild;
  return(next);
}
```

3 线索二叉树的遍历

为方便起见，为线索二叉树增加一个头结点，使之类似一个双向循环线索链表。

```
Void inorder_thr(BiThrTree T)//中序遍历算法1
{  p=T->lchild;
   while(p!=T) {
        while(p->LTag= =0)   p=p->lchild;
        if(!visit(p->data))  return error;
       while(p->RTag= =thread && p->rchild!=T)
                {p=p->rhild; visit(p->data);}
        p=p->rchild;
     }
}
```

3 线索二叉树的遍历

补充算法2：利用nextnode函数

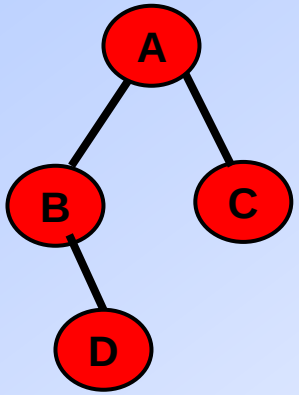
```
Void inorder_thr(BiThrTree T)//中序
{  p=T->lchild;
   while(p!=T && !p->LTag)        //查找最左结点
      p=p->lchild;
   while(p!=T)
   { printf(p->data);     p= nextnode(p);  }
}
```

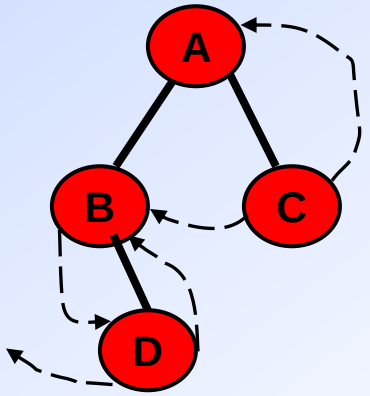
4 二叉树的线索化

```
void inorderthreading(BiThrTree &Thrt, BiThrTree T){
If（！(Thrt=BiThrTree)malloc(sizeof(BiThrNode))))exit(overflow)
Thrt->LTag=Link; THrt->RTag=Thread;//建头结点
Thrt->rchild=Thrt;                    //右指针回指
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; pre->RTag=Thread;//最后一个结点线索化
       Thrt->rchild=pre;}
Return ok;
}
```

```
Void InThreading（BiThrTree P）{
If（p){
        InThreading(P->lchild);   //左子树线索化
        if (!p->lchild) { p->LTag=Thread;p->lchild=pre;} //前驱线索
        if(!pre->rchild){pre->RTag=Thread; pre->rchild=p}//后继线索
        Pre=p;  //保持pre指向p的前驱
        InThreading(p->rchild);//右子树线索化
}
}
```

练习：请画出后序线索二叉树





后序序列DBCA

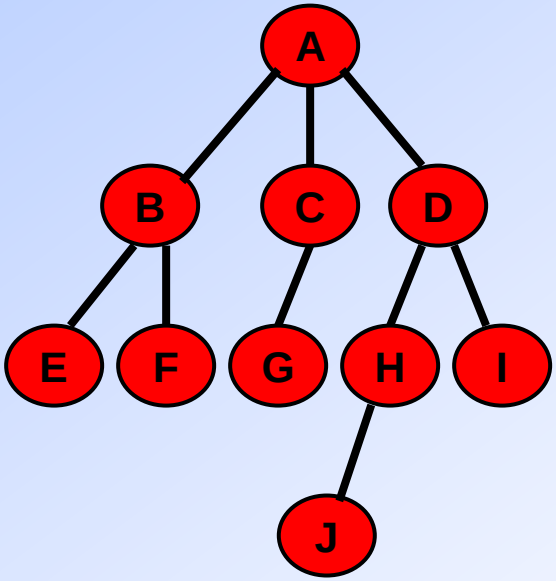
6.4 树和森林

6.4.1 树的存储结构

1 孩子表示法：由数据域、 K 个子结点指针域、父结点指针域组成。

data	child1	child2	.....	childk	parents

例: 度数 K ＝ 3 的树



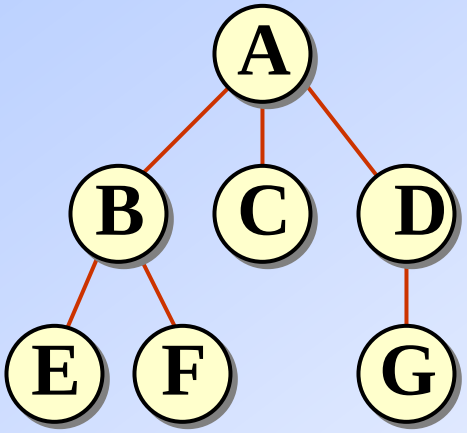
用数组（静态链式存储）表示左图的树，-1 表示空。

缺点：空指针域太多，有(K -1)× n + 1 个。

0	A	1	2	3	-1
1	B	4	5	-1	0
2	C	6	-1	-1	0
3	D	7	8	-1	0
4	E	-1	-1	-1	1
5	F	-1	-1	-1	1
6	G	-1	-1	-1	2
7	H	9	-1	-1	3
8	I	-1	-1	-1	3
9	J	-1	-1	-1	7

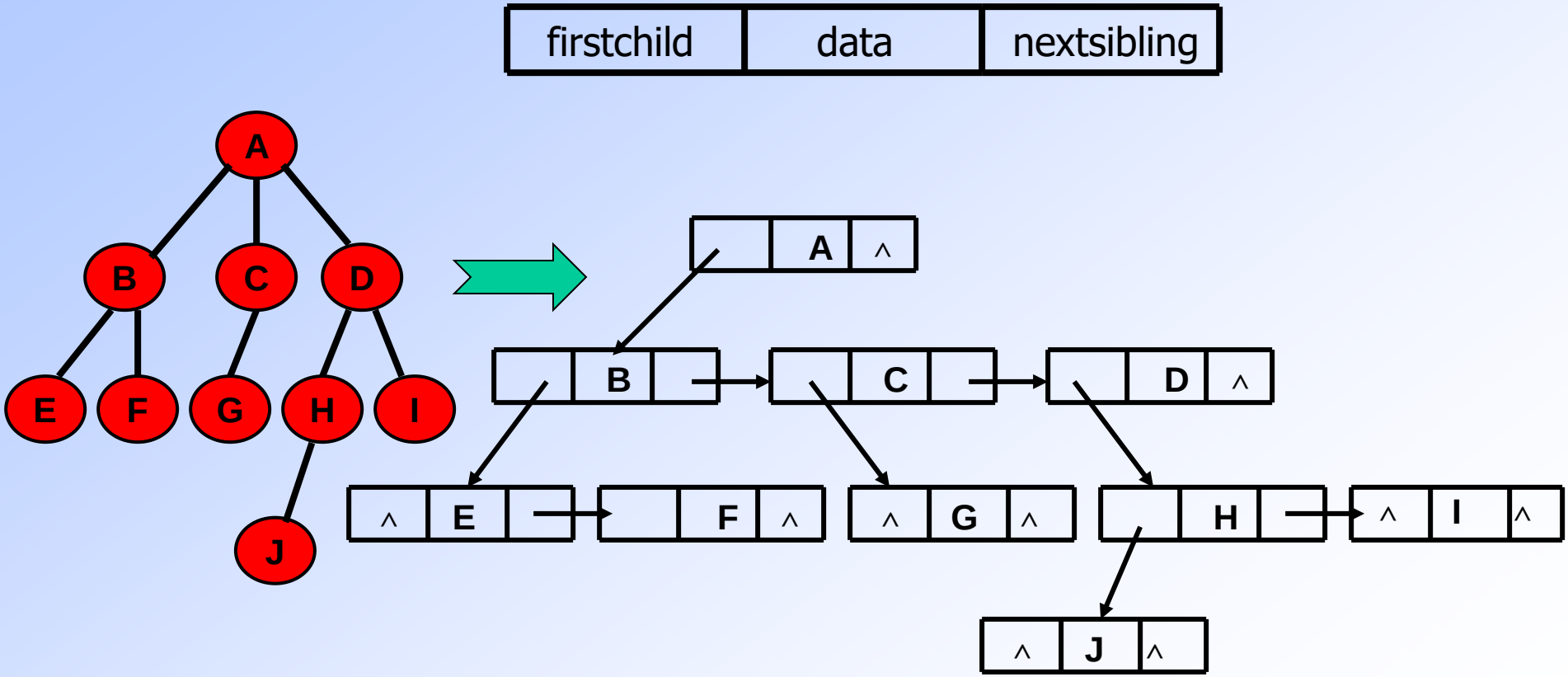
树与森林

2 双亲表示： 以一组连续空间存储树的结点，同时在结点中附设一个指针，存放双亲结点在链表中的位置。



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

3 孩子兄弟表示法：由数据域、指向它的第一棵子树树根的指针　　域、指向它的兄弟结点的指针域组成。实质上是用二叉树表示一棵树。

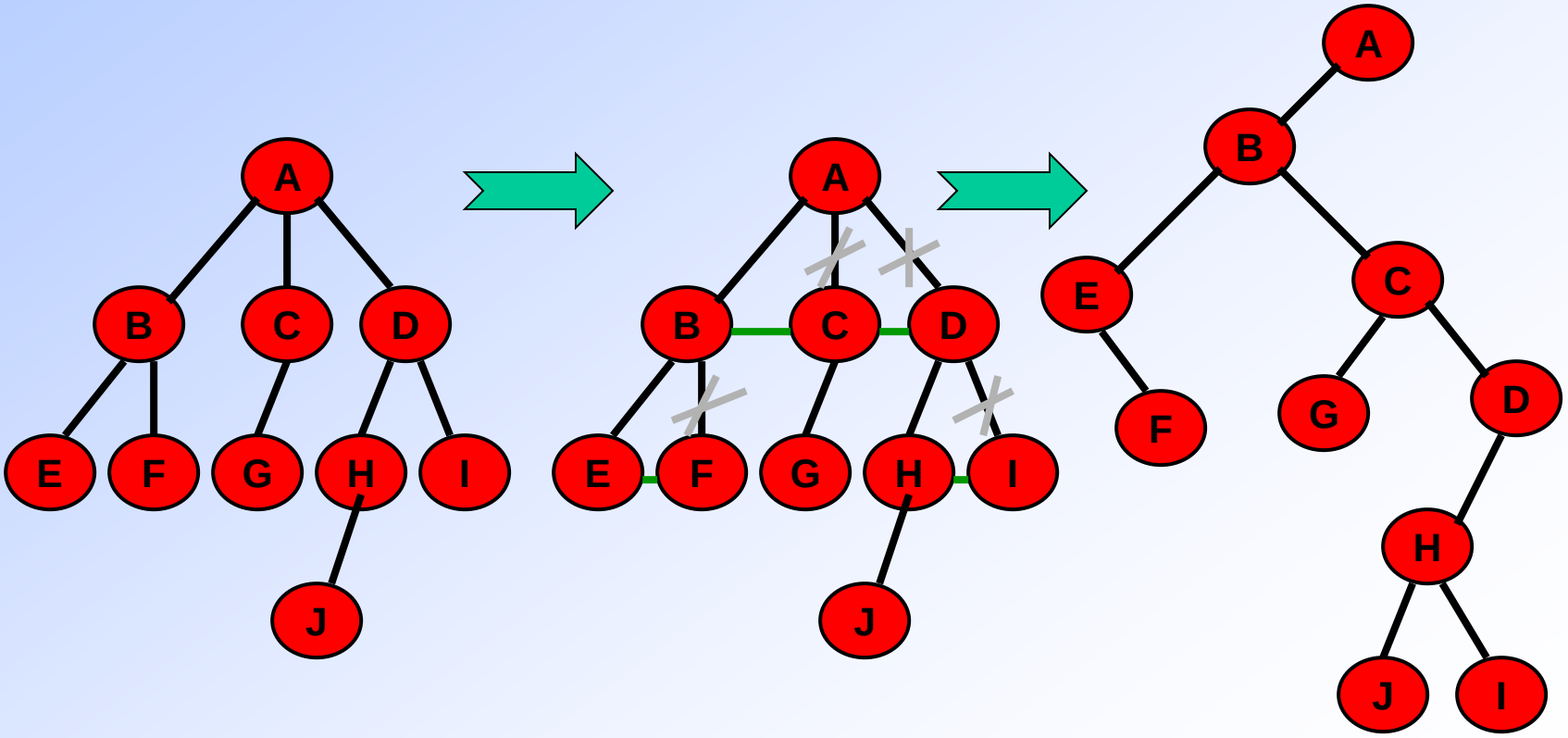


6.4.2 树、森林与二叉树的转换

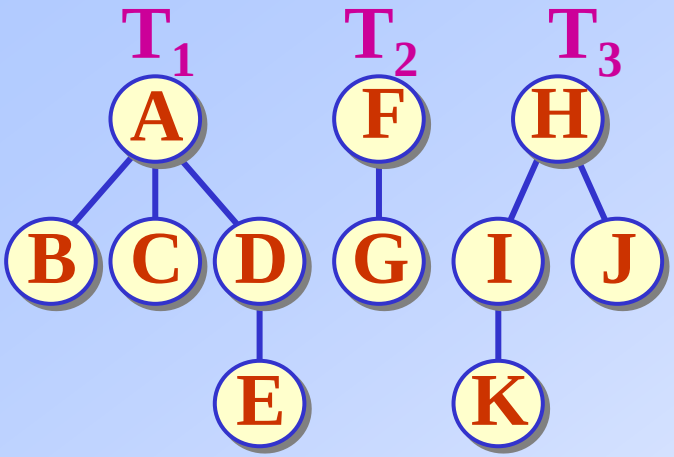
1 树转换成相对应的二叉树：

1）保留每个结点的最左面的分支，其余分支都被删除。

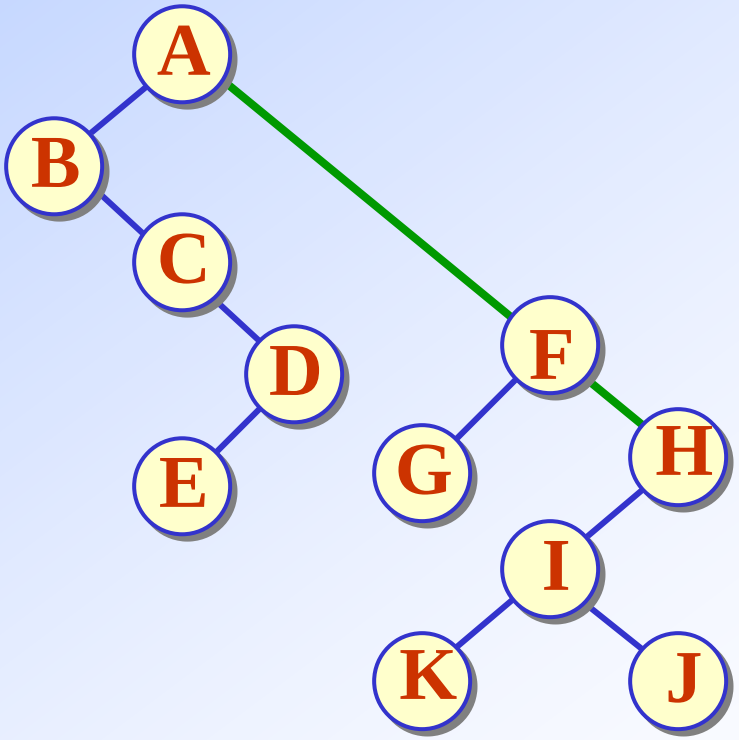
2）同一父结点下面的结点成为它的左方结点的兄弟。



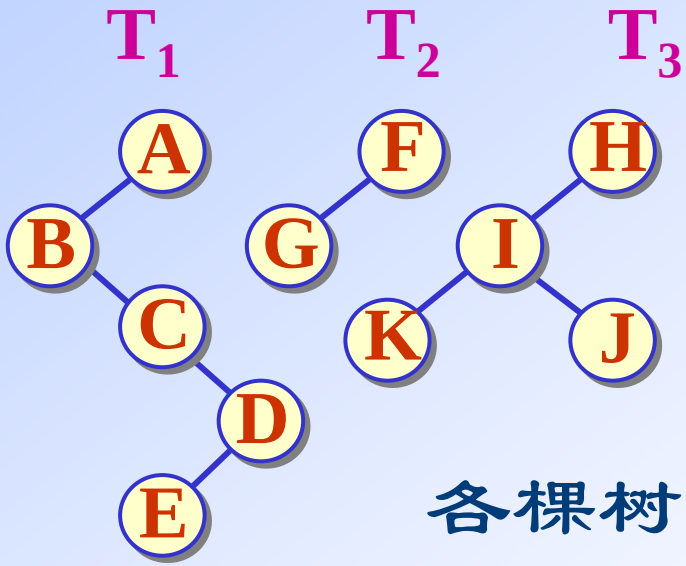
❑森林与二叉树的转换



3 棵树的森林



森林的二叉树表示

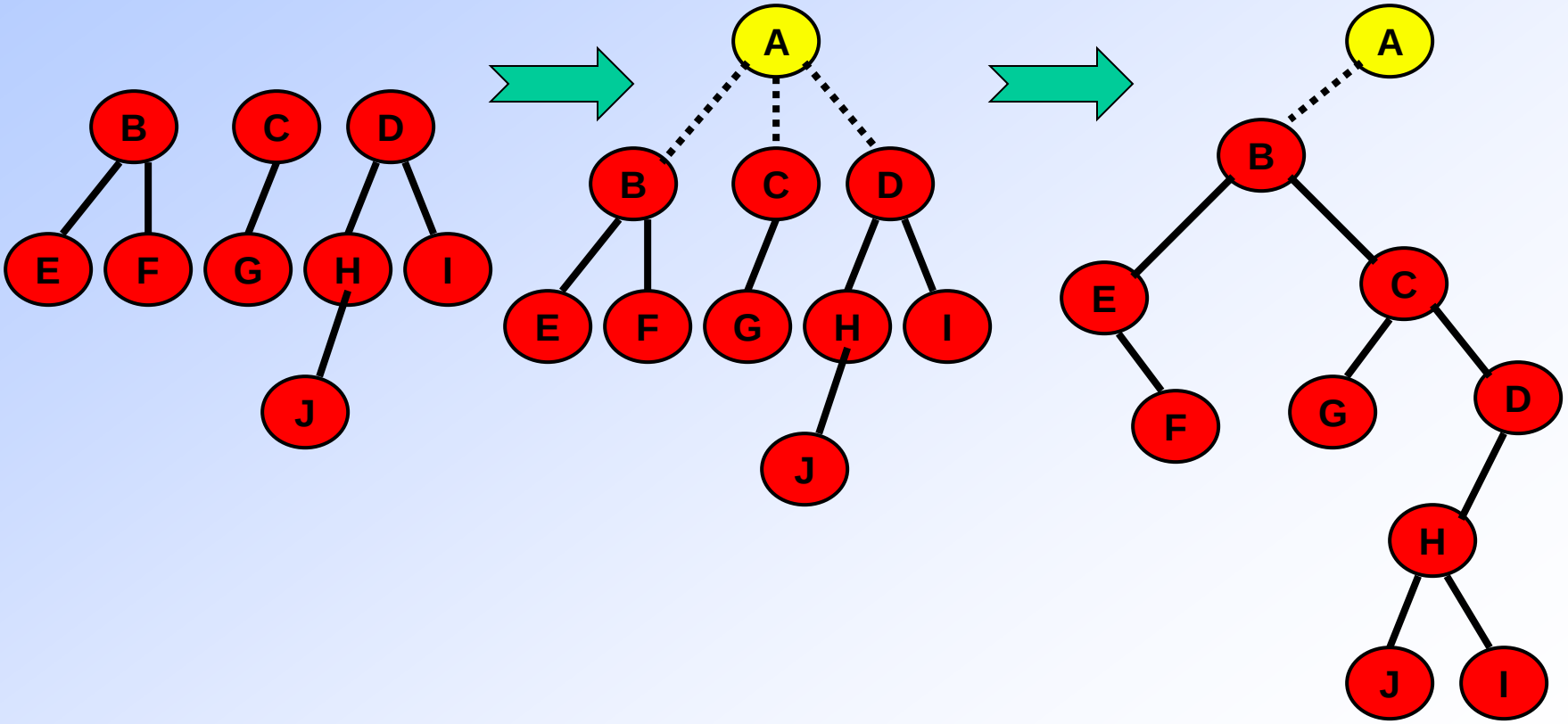


各棵树的二叉树表示

2 森林转换成相对应的二叉树：

增加一个虚拟的根结点，它的子结点为各棵树的根。

那么，就变成了树转换成二叉树的问题。

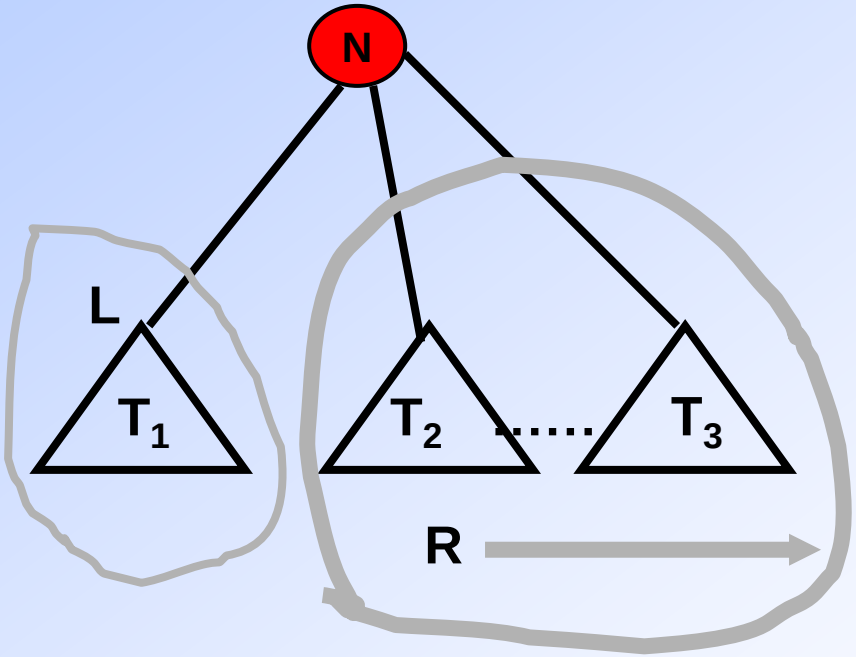


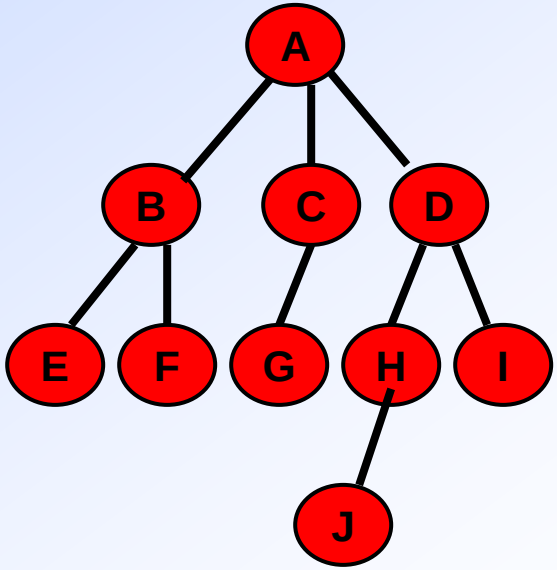
6.4.3 树、森林的遍历

1 树的先根、后根遍历：

1）先根：类似于二叉树的先序遍历（NLR） N:根；L：第一棵子树，R：其余的子树，遍历方向由第二棵子树至最后一棵子树。

2）后根：类似于二叉树的中序遍历（LRN） L：第一棵子树，R：其余的子树，遍历方向由第二棵子树至最后一棵子树， N:根。



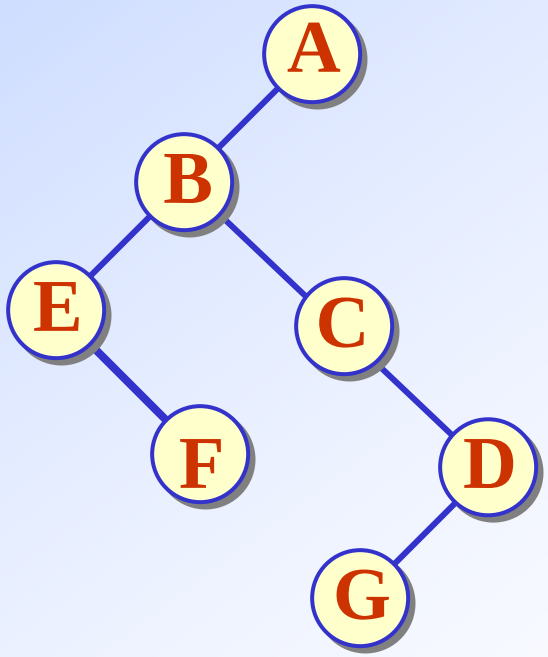


先根：A B E F C G D H J I

后根：E F B G C J H I D A

树的先根次序遍历

- 当树非空时
  - 访问根结点
  - 依次先根遍历根的各棵子树

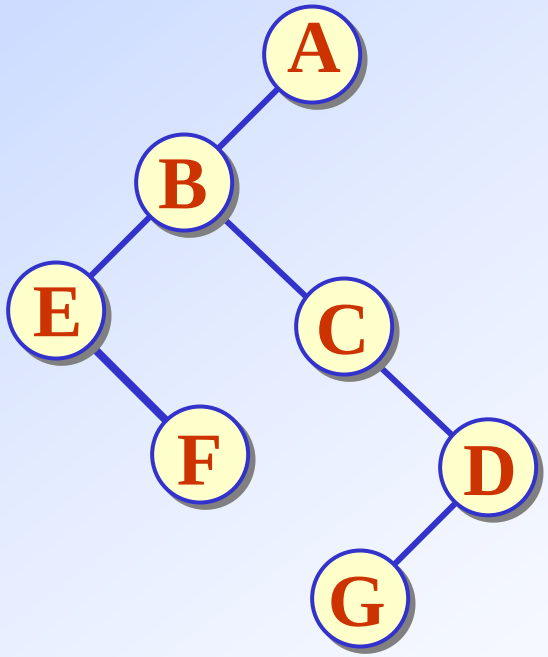


- 树先根遍历 ABEFCDG
- 对应二叉树前序遍历 ABEFCDG
- 树的先根遍历结果与其对应二叉树表示的前序遍历结果相同

- 树的先根遍历可以借助对应二叉树的前序遍历算法实现

树的后根次序遍历

- 当树非空时
  - 依次后根遍历根的各棵子树
  - 访问根结点



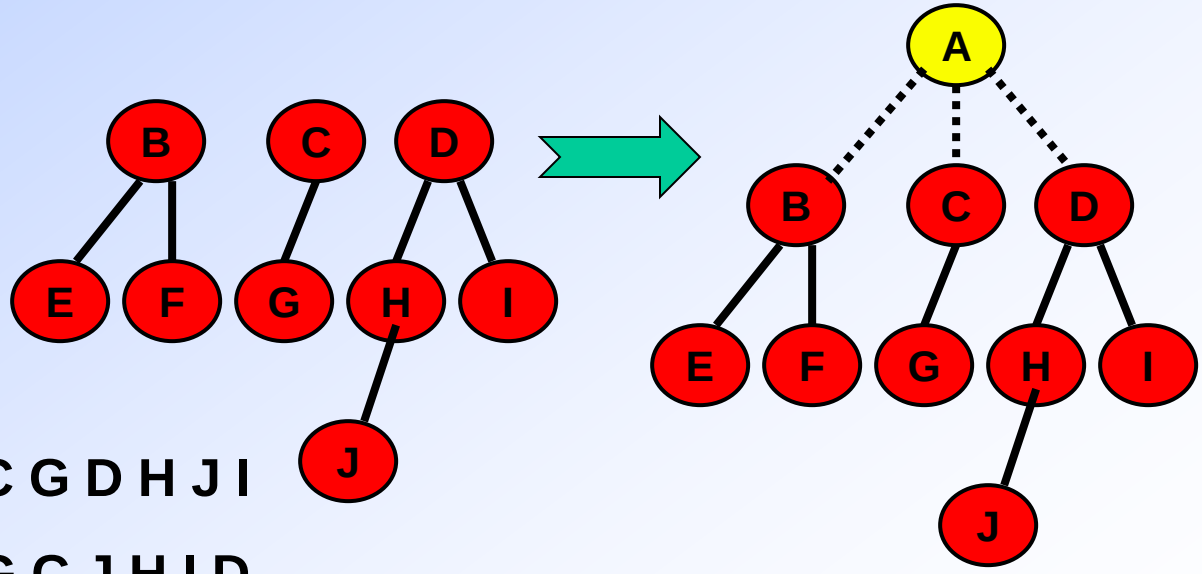
- 树后根遍历 EFBCGDA
- 对应二叉树中序遍历 EFBCGDA
- 树的后根遍历结果与其对应二叉树表示的中序遍历结果相同

- 树的后根遍历可以借助对应二叉树的中序遍历算法实现

2 森林的先序、中序遍历（对应二叉树的前序、中序遍历）

1）先序遍历类似于树的先序遍历。增加一个虚拟的根结点，它的子结点为各棵树的根。对这棵树进行先根遍历，即得到森林的先序序列（去掉树根结点）。

2）中序遍历类似于树的后序遍历。增加一个虚拟的根结点，它的子结点为各棵树的根。对这棵树进行后根遍历，即得到森林的中序序列（去掉树根结点）。



先序：B E F C G D H J I

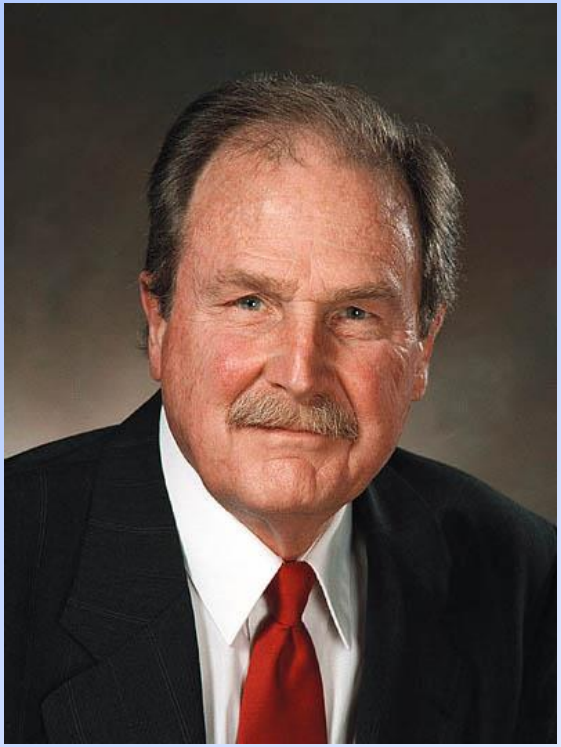
中序：E F B G C J H I D

霍夫曼（Huffman）编码

David A. Huffman1952年在麻省理工读博士的时候发明的，并发表于《一种构建极小冗余编码的方法》（A Method for the Construction of Minimum-Redundancy Codes）一文。

Huffman在Shannon和Fano阐述的编码思想上提出了一种不定长编码的方法。哈夫曼使用自底向上的方法构建二叉树，避免了次优算法Shannon-Fano编码的最大弊端—自顶向下构建树。

霍夫曼



戴维·霍夫曼(David Albert Huffman)

信息论的先驱,对计算机科学、通信等学科所作出巨大贡献。

霍夫曼除了获得计算机先驱奖以外,还在1973年获得IEEE的McDowell奖。1998年IEEE下属的信息论分会为纪念信息论创立50周年,授予他Golden Jubilee奖。霍夫曼去世前不久的1999年6月,他还荣获以哈明码发明人命名的哈明奖章(Hamming Medal)。

•Huffman编码广泛用于数据文件压缩，其压缩率通常在20%～90%之间。是一种统计编码，属于无损压缩编码。哈夫曼编码是可变字长编码(VLC)的一种，该方法完全依据字符出现概率来构造异字头的平均长度最短的码字，有时称为最佳编码，或称霍夫曼（哈夫曼）编码。

•它是一种无损压缩方法，在压缩过程中不会丢失信息熵，而且可以证明Huffman算法在无损压缩算法中是最优的。Huffman原理简单，在现在的主流压缩软件得到了广泛的应用。对应用程序、重要资料等绝对不允许信息丢失的压缩场合，Huffman算法是非常好的选择。

•基于Huffman经典算法的缺陷（如：整个图像全部输入扫描完成后构造出来），提出了一些自适应算法(或称动态算法)。不必等到全部图像输入完成才开始树的构造，并且可以根据后面输入的数据动态地对Huffman树进行调整。实用的Huffman树都是经过某种优化后的动态算法。

霍夫曼树 (Huffman Tree)

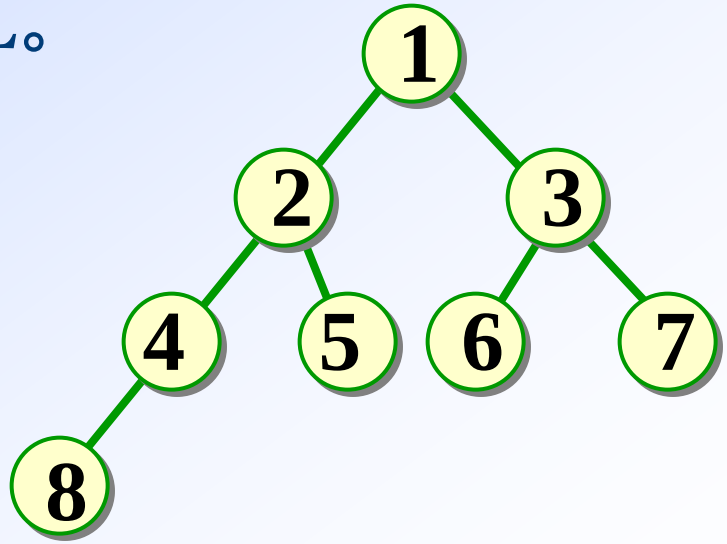
哈夫曼树又称最优二叉树，是一种带权路径长度最短的二叉树。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的路径长度。

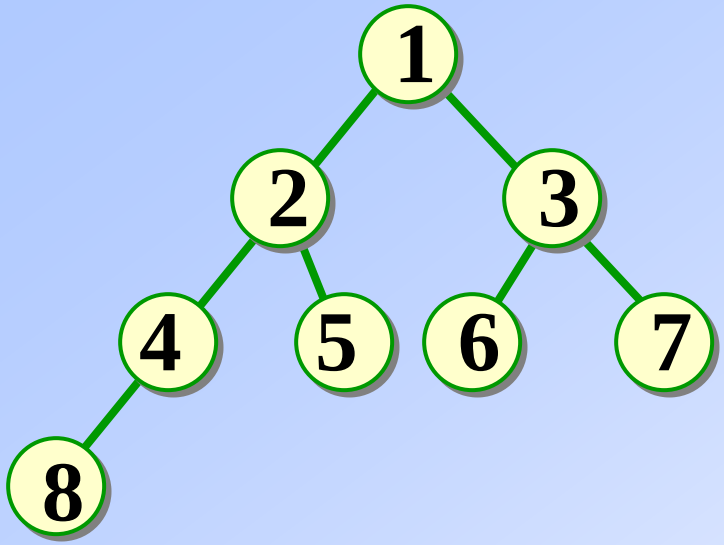
▪路径长度 (Path Length)

两个结点之间的路径长度 PL 是连接两结点的路径上的分支数。

树的路径长度是根结点到每一结点的路径长度之和 。

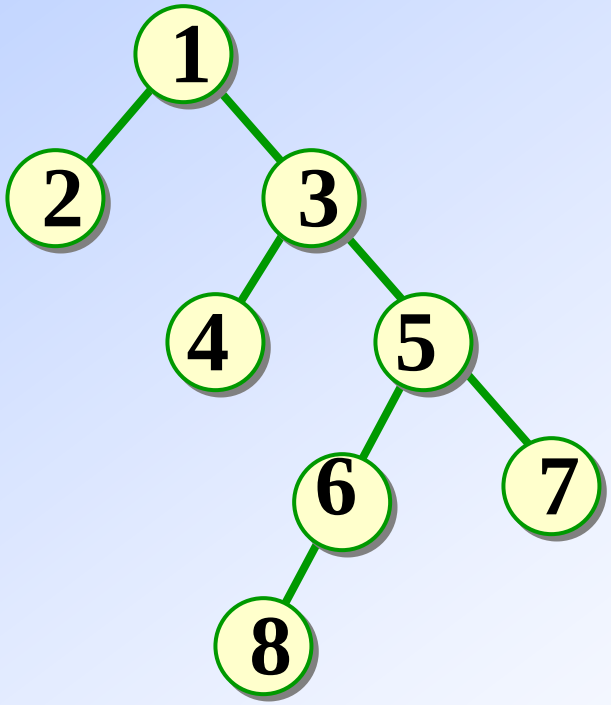
树的外部路径长度是根结点到各叶结点(外结点)的路径长度之和 EPL。





树的外部路径长度

EPL = 3*1+2*3 = 9



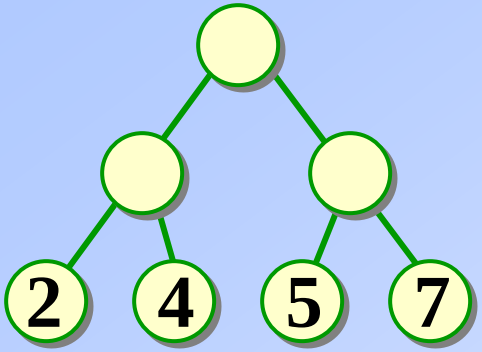
树的外部路径长度

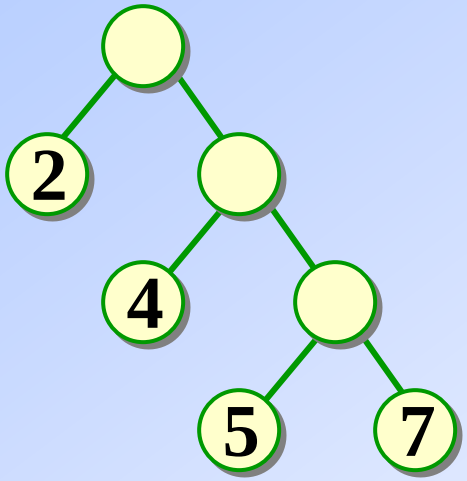
EPL = 1*1+2*1+3*1+4*1 = 10

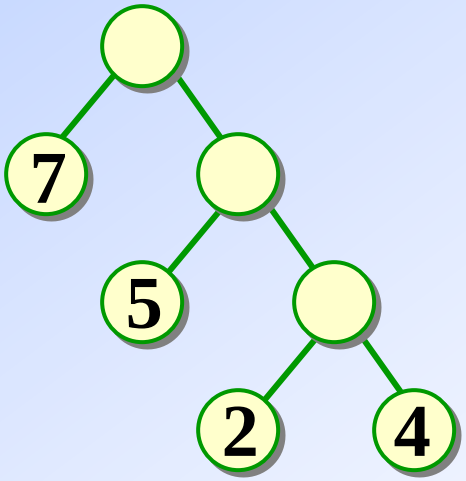
- **带权路径长度 (Weighted Path Length, WPL)**

二叉树的带权 (外部) 路径长度是树的各叶结点所带的权值 w_i 与该结点到根的路径长度 l_i 的乘积的和。

$$WPL = \sum_{i=0}^{n-1} w_i * l_i$$







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

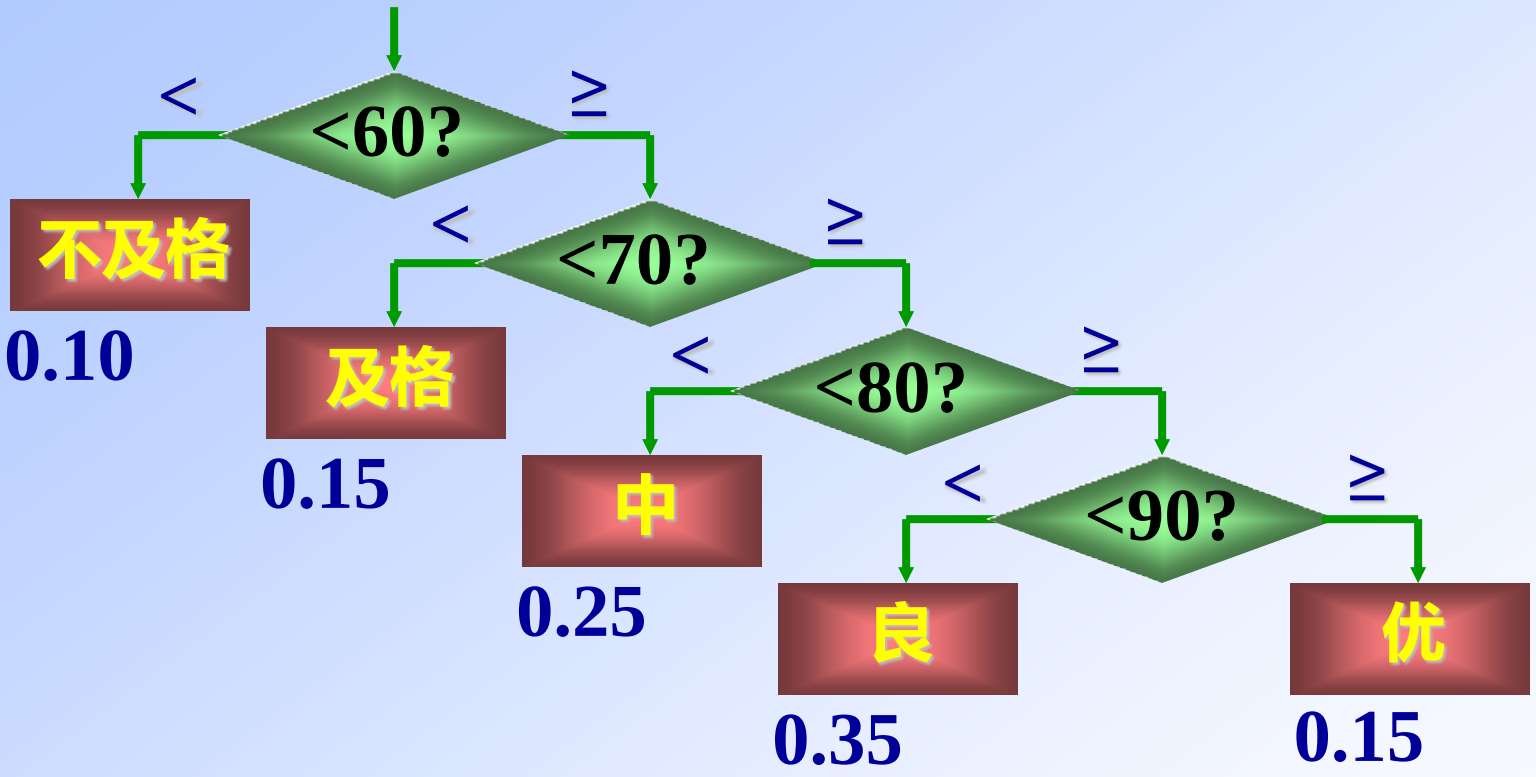
带权(外部)路径长度达到最小

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

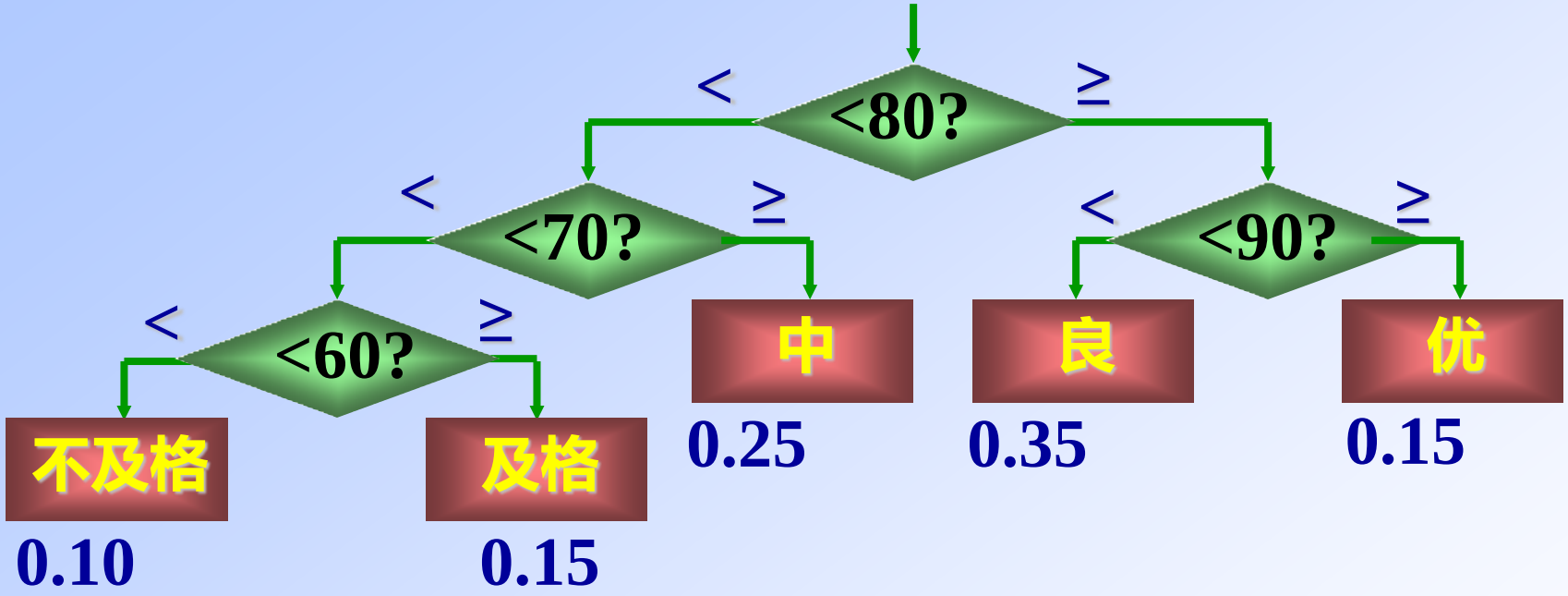
判定树



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

= 3.15

最佳判定树



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

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

霍夫曼树

- 带权路径长度达到最小的二叉树即为霍夫曼树。
- 在霍夫曼树中，权值大的结点离根最近。

霍夫曼算法　　贪心算法

(1) 由给定的 n 个权值 $\{w_0, w_1, w_2, \cdots, w_{n-1}\}$，构造具有 n 棵扩充二叉树的森林 $F = \{T_0, T_1, T_2, \cdots, T_{n-1}\}$，其中每棵扩充二叉树 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 中仅剩下一棵树为止：

① 在 F 中选取两棵根结点的权值最小的扩充二叉树，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。

② 在 F 中删去这两棵二叉树。

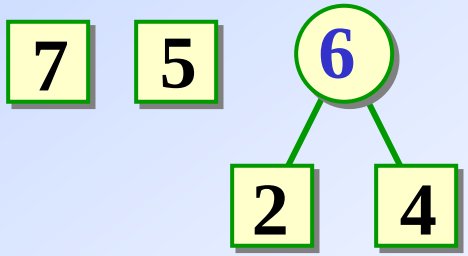
③ 把新的二叉树加入 F。

霍夫曼树的构造过程举例

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

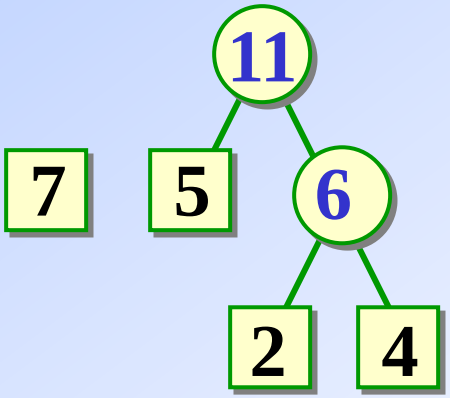
初始

F : {7} {5} {6}



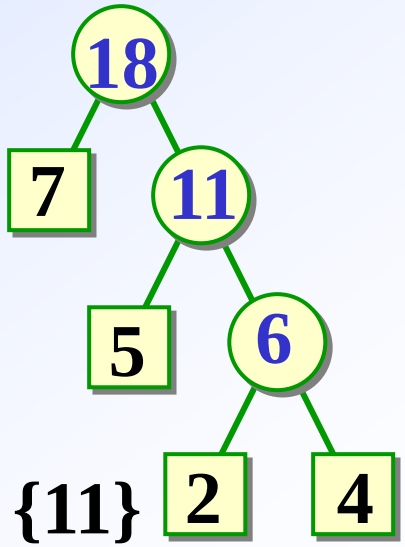
合并{2} {4}

F : {7} {11}



合并{5} {6}

F : {18}



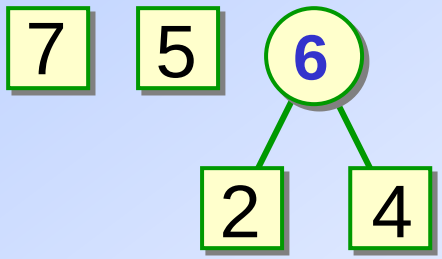
合并{7} {11}

存储结构

7 5 2 4

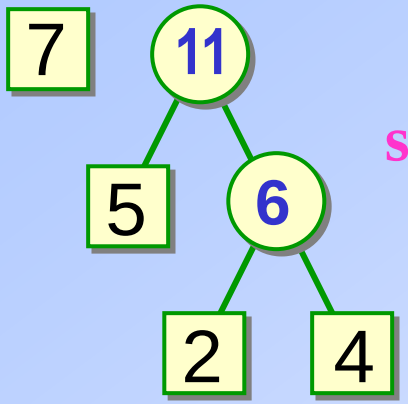
	Weight	parent	lchild	rchild
1	7	0	0	0
2	5	0	0	0
3	2	0	0	0
4	4	0	0	0
5		0	0	0
6		0	0	0
7		0	0	0

初 态



	Weight	parent	lchild	rchild
1	7	0	0	0
2	5	0	0	0
3 (s1)	2	5 ~~0~~	0	0
4 (s2)	4	5 ~~0~~	0	0
5 (i)	6	0	3 ~~0~~	4 ~~0~~
6		0	0	0
7		0	0	0

过 程

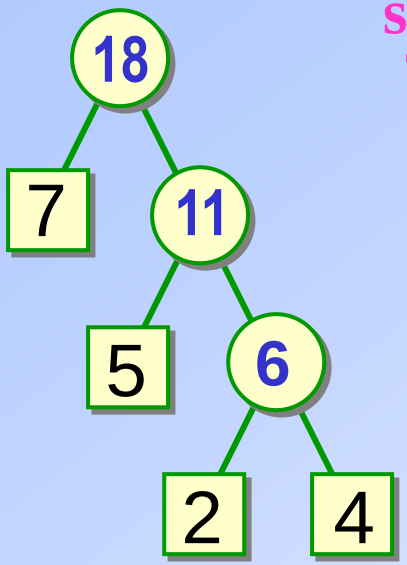


s1 →

s2 →

i →

HT:	Weight	parent	lchild	rchild
1	7	0	0	0
2	5	6 ~~0~~	0	0
3	2	5	0	0
4	4	5	0	0
5	6	6 ~~0~~	3	4
6	11	0	2 ~~0~~	5 ~~0~~
7		0	0	0



	Weight	parent	leftChild	rightChild
s1 → 1	7	7 ~~0~~	0	0
2	5	6	0	0
3	2	5	0	0
4	4	5	0	0
5	6	6	3	4
s2 → 6	11	7 ~~0~~	2	5
i → 7	18	0	1 ~~0~~	6 ~~0~~

终 态

霍夫曼树的定义

```
typedef struct {
    float    weight;
     int     parent, lchild, rchild;
} HTNode;

typedef  HTNode *   HuffmanTree;
typedef  char *  codepointer;//指向字符串的指针类型
```

建立霍夫曼树的算法

```
void CreateHuffmanTree ( HuffmanTree HT,
codepointer HC[ ], float *w, int n){
m = 2*n -1;//结点数
HT=(HuffmanTree) malloc ((m+1)*sizeof(HTNode));
                    //0号单元未用,此句可在main()中,HT定义为全局变量
for ( p=HT+1, i = 1; i <= n; ++ i,++p,++w) *p={*w,0,0,0};
for ( ; i <= m; ++ i,++p)   *p={0,0,0,0};
for ( i = n+1; i <= m; ++ i) {
     select(HT,i-1,s1,s2); //在HT[1..I-1]选择parent为0且weight最小的两结点
    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
    HT[i].parent=0;
 }
……
```

练习：对n（n>=2）个权值均不同的字符构成哈夫曼树，关于该树的叙述中，错误的是：A

（a）该树一定是一棵完全二叉树

（b）该树中一定没有度为1的结点

（c）树中两个权值最小的结点一定是兄弟结点

（d）树中任一非叶子结点的权值一定不小于下一层任一结点的权值

霍夫曼编码　无损压缩的经典算法

主要用途是实现数据压缩。例如JPEG文件、数字电视视频信号的压缩。

设给出一段报文:

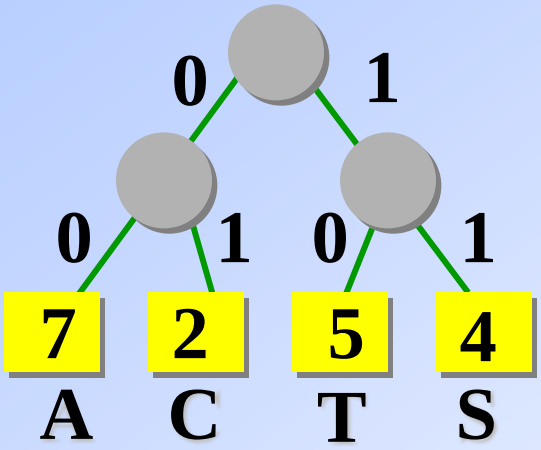
CASTCASTSATATATASA

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

A : 00　T : 10　C : 01　S : 11

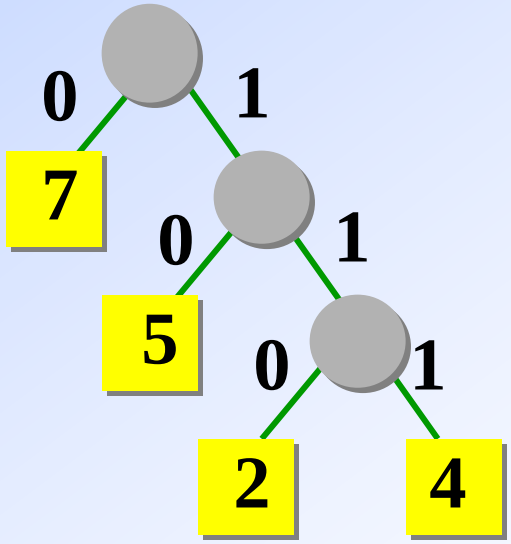
则总编码长度为 (2+7+4+5) * 2 = 36.



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。

各字符出现概率为{ 2/18, 7/18, 4/18, 5/18 },化整为 { 2, 7, 4, 5 }。以它们为各叶结点上的权值, 建立霍夫曼树。左分支赋 0，右分支赋 1，得霍夫曼编码(变长编码)。

霍夫曼编码



A : 0　T : 10　C : 110　S : 111

总编码长度：7*1+5*2+(2+4)*3 = 35

比等长编码的情形要短。　总编码长度正好等于霍夫曼树的带权路径长度WPL。

霍夫曼编码是一种前缀编码。什么是前缀编码呢?所谓的前缀编码就是任何一个字符的编码都不能是另一个字符编码的前缀，这样解码时不会混淆。

定长编码已经用相同的位数这个条件保证了任一个字符的编码都不会成为其它编码的前缀。

若5个字符有如下4种编码方案，不是前缀编码的是(D)。

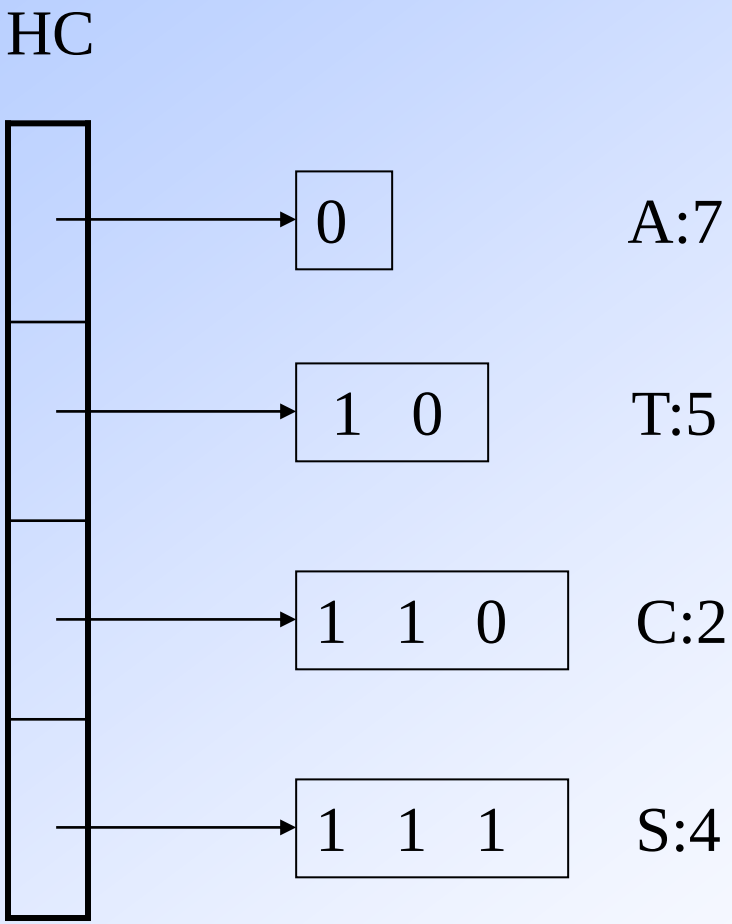
A．01，0000，0001，001，1

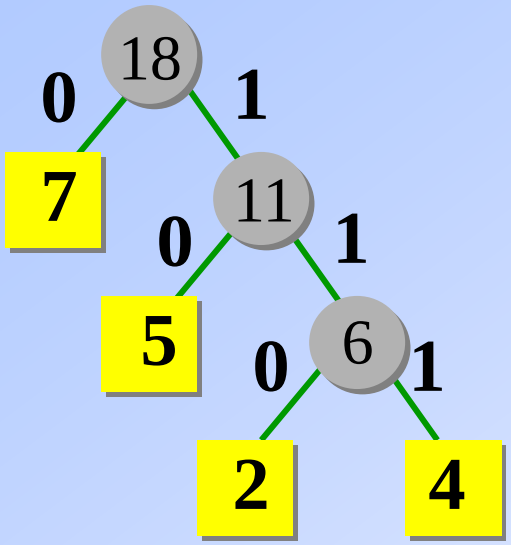
B．011，000，001，010，1

C．000，001，010，011，100

D．0，100，110，1110，1100

霍夫曼编码





	Weight	parent	leftChild	rightChild
1	7	7	0	0
2	5	6	0	0
3	2	5	0	0
4	4	5	0	0
5	6	6	3	4
6	11	7	2	5
7	18	0	1	6

终 态

```
cd=(char*)malloc(n*sizeof(char));
cd[n-1]='\0';
for (i=1;i<=n;++i){
   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[start]);
}
free(cd);}
```

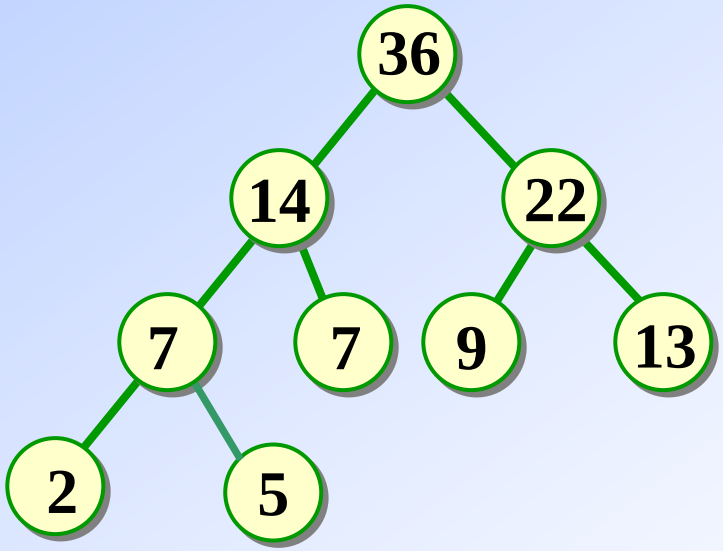
(Bb平台实践作业题)

- 农夫要修理牧场的一段栅栏，他测量了栅栏，发现需要N块木头，每块木头长度为整数L_i个长度单位，于是他购买了一条很长的、能锯成N块的木头，即该木头的长度是L_i的总和。但是农夫自己没有锯子，请人锯木的酬金跟这段木头的长度成正比。
- 请编写程序帮助农夫计算将木头锯成N块的最少花费。

- 为简单起见，不妨就设酬金等于所锯木头的长度。例如要将长度为20的木头锯成长度为8、7和5的三段，第一次锯木头花费20，将木头锯成12和8；第二次锯木头花费12，将长度为12的木头锯成7和5，总花费为32。如果第一次将木头锯成15和5，则第二次锯木头花费15，总花费为35（大于32）

- 第一行输入正整数N（<=104），表示要将木头锯成N块。
- 第2行给出N个正整数（<=50），表示每段木块的长度。
- 6
- 1 2 1 3 4 5

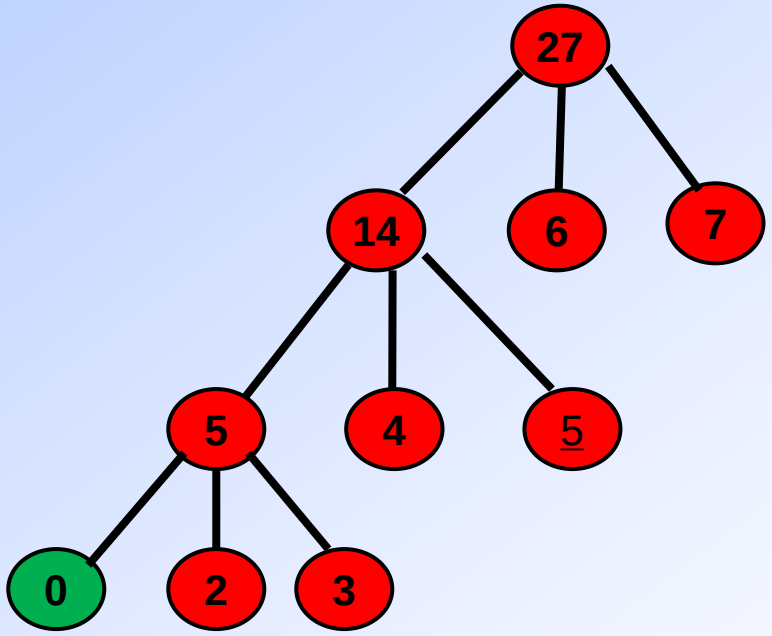
练习：以数据集{2,5,7,9,13}为权值构造一棵哈夫曼树，并计算其带权路径长度。 79



霍夫曼n叉树

（A Method for the Construction of Minimum-Redundancy Codes）

- 已知三叉树T中6个叶子结点的权分别是 2,3,4,5,6,7，则树T的带权路径长度最小是多少？ 46

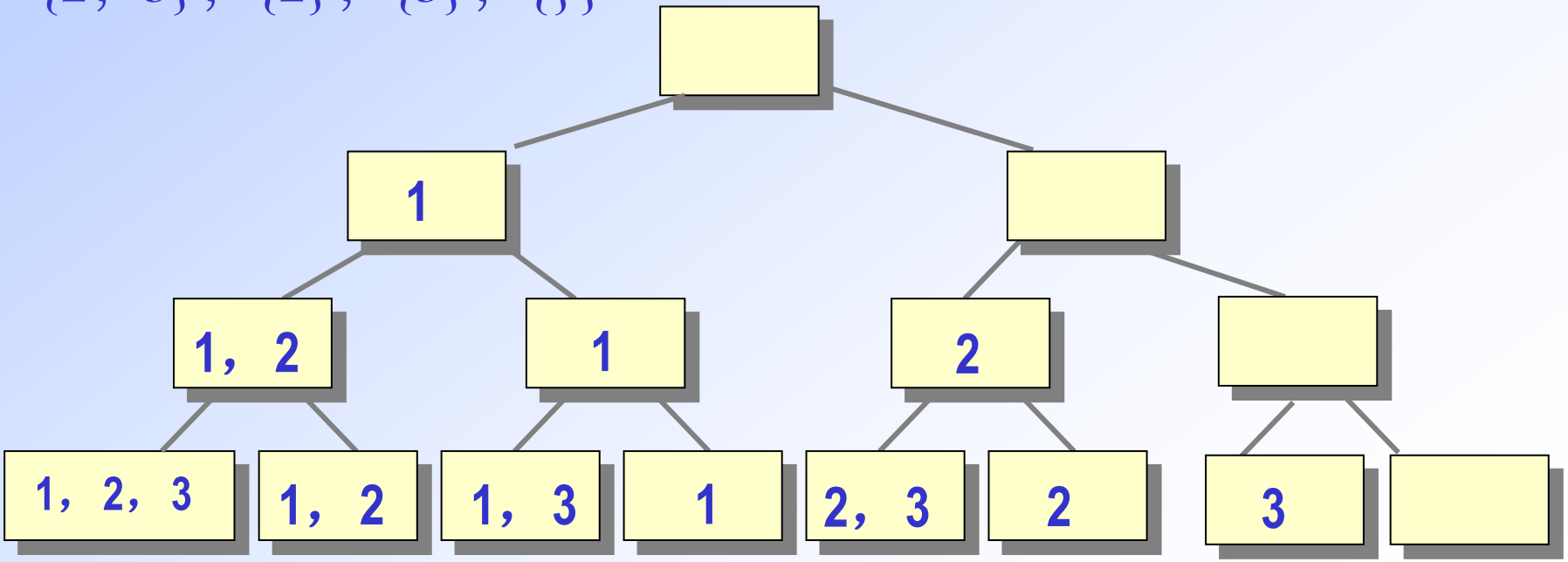


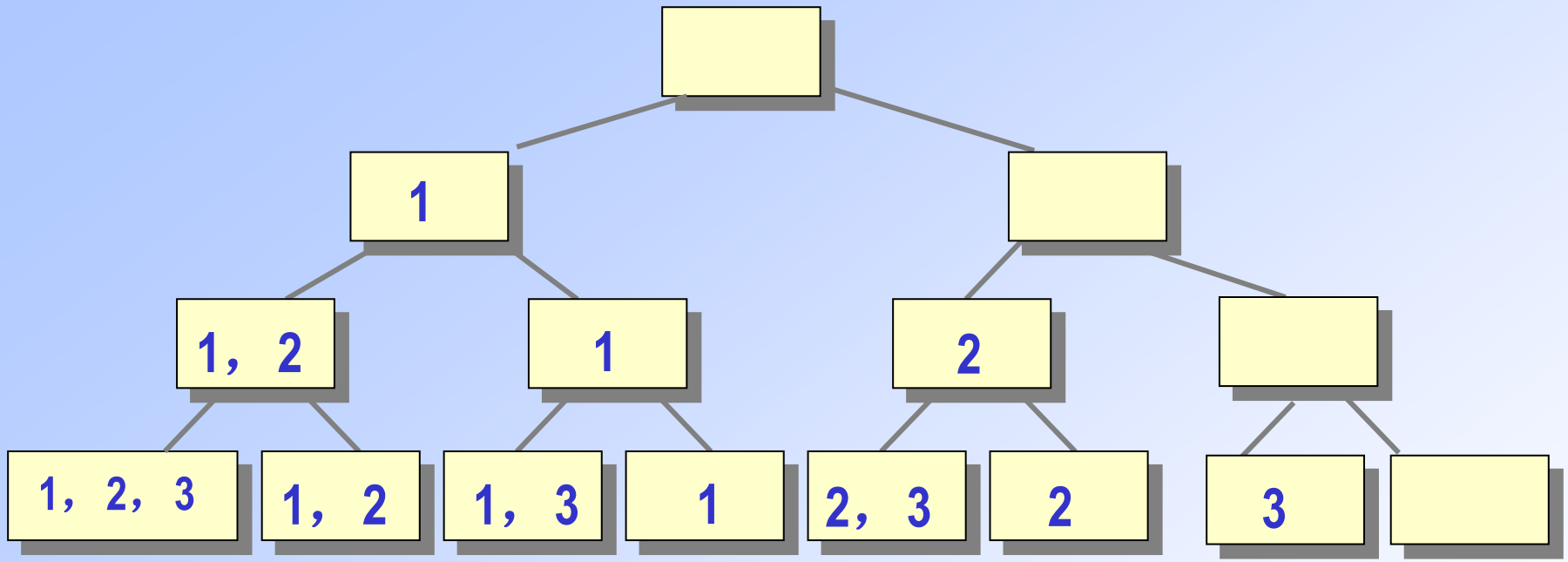
假设最初有n个点，每次合并删除k个点又放进1个点，最后有1个点。那么（n-1）是（k-1）的倍数。如果（n-1）%（k-1）！=0，那么就要再放入（k-1-（n-1）%（k-1））个虚拟点。

递归与回溯 常用于搜索过程

回溯法是搜索算法中的一种控制策略。它从初始状态出发，运用题目给出的条件规则，按照深度优先搜索顺序扩展所有可能情况。它的求解过程实质上是先序遍历一棵“状态树”的过程，这棵树的建立是隐含在遍历过程中。对采用回溯法求解的问题设计递归算法程序。 例如求含n个元素的集合的幂集。

假设A={1,2,3},则A的幂集为 {{1，2，3}，{1，2}，{1，3}，{1}，{2，3}，{2}，{3}，{}}



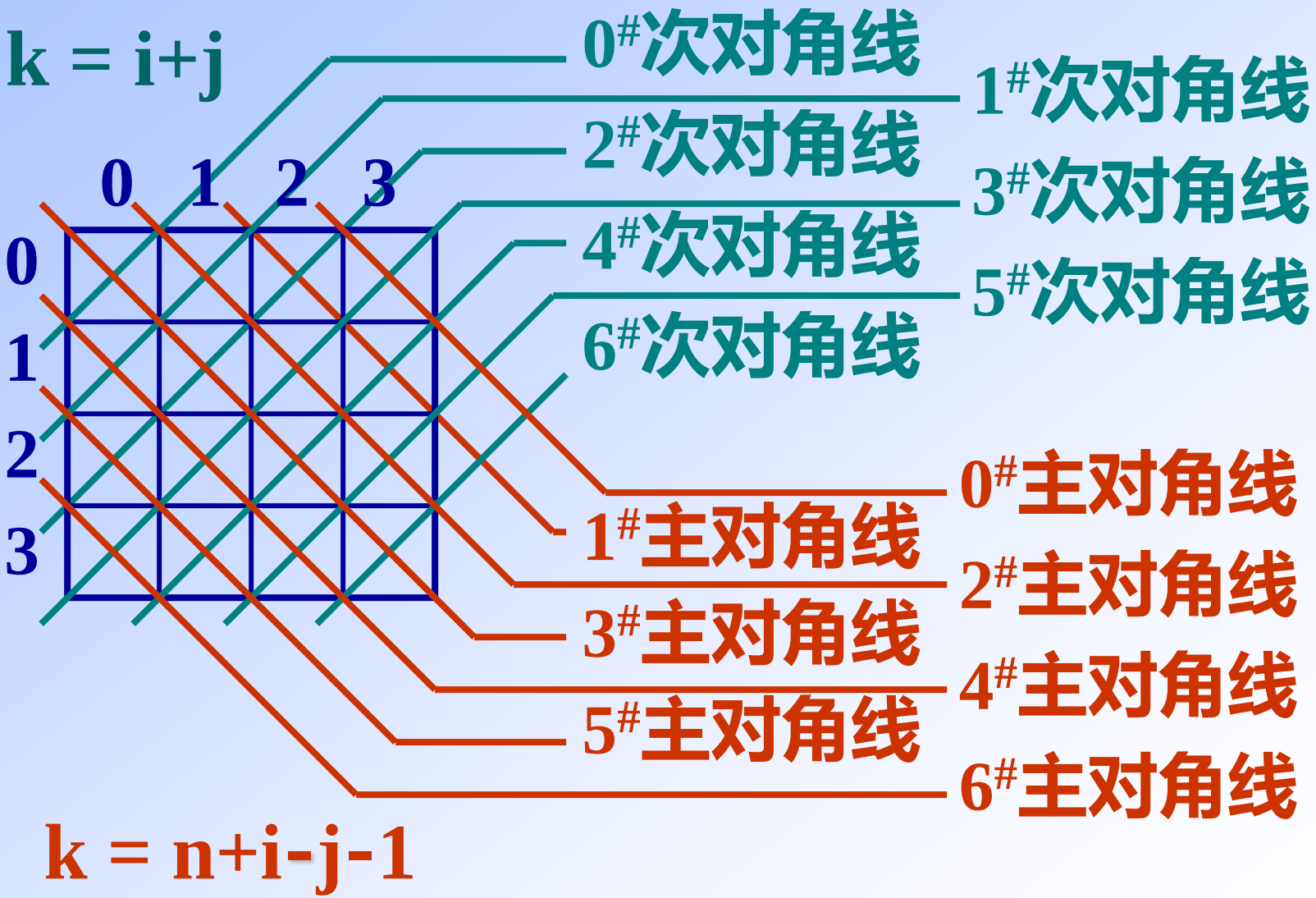



```
Void Powerset(int i,int n){
 if (i>n)   输出幂集的一个元素；
else{      取第i个元素； Powerset(i+1, n）；
           舍第i个元素； Powerset(i+1, n）； }
 }//初始调用Powerset(1, n)
```

递归与回溯 常用于搜索过程

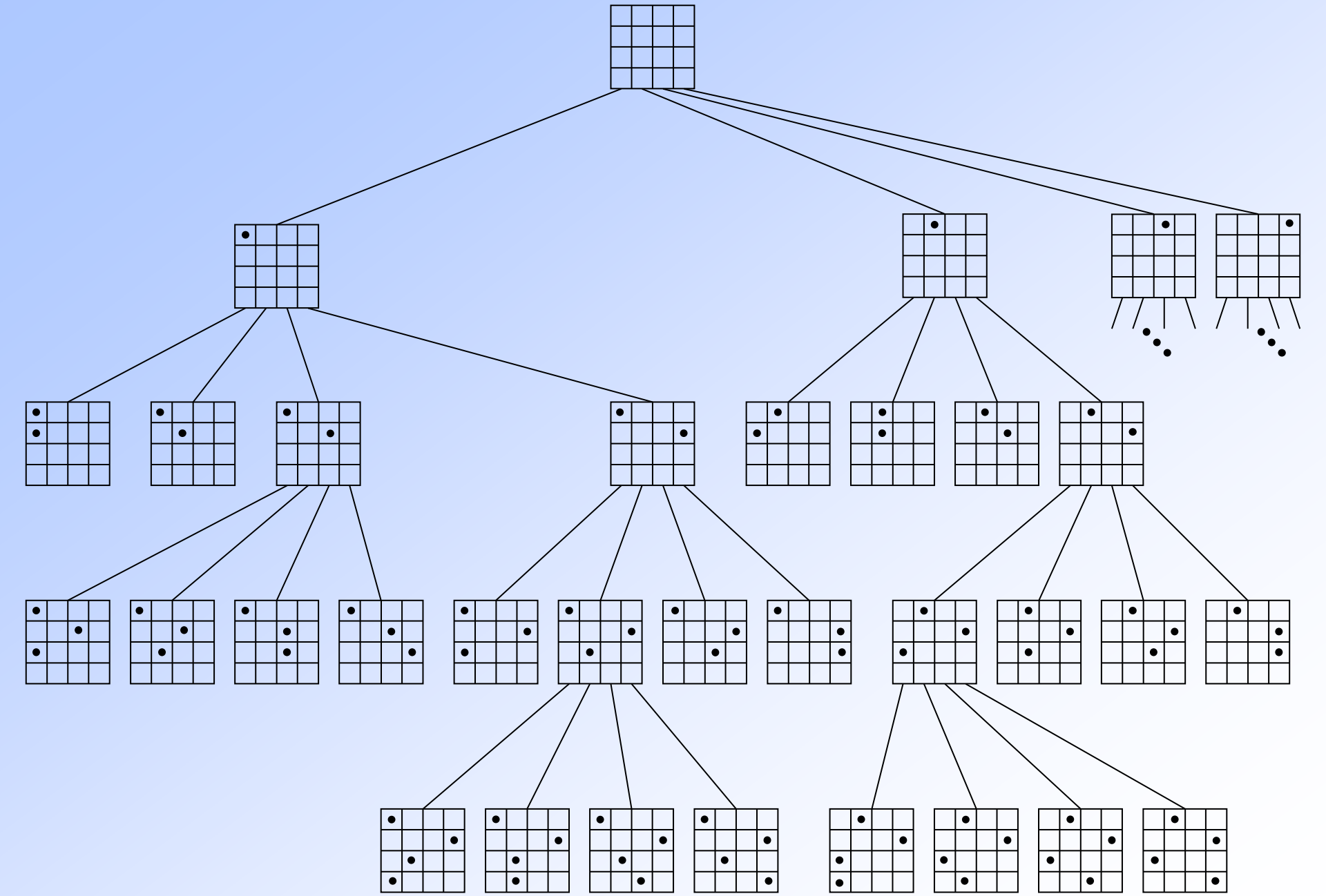
n皇后问题

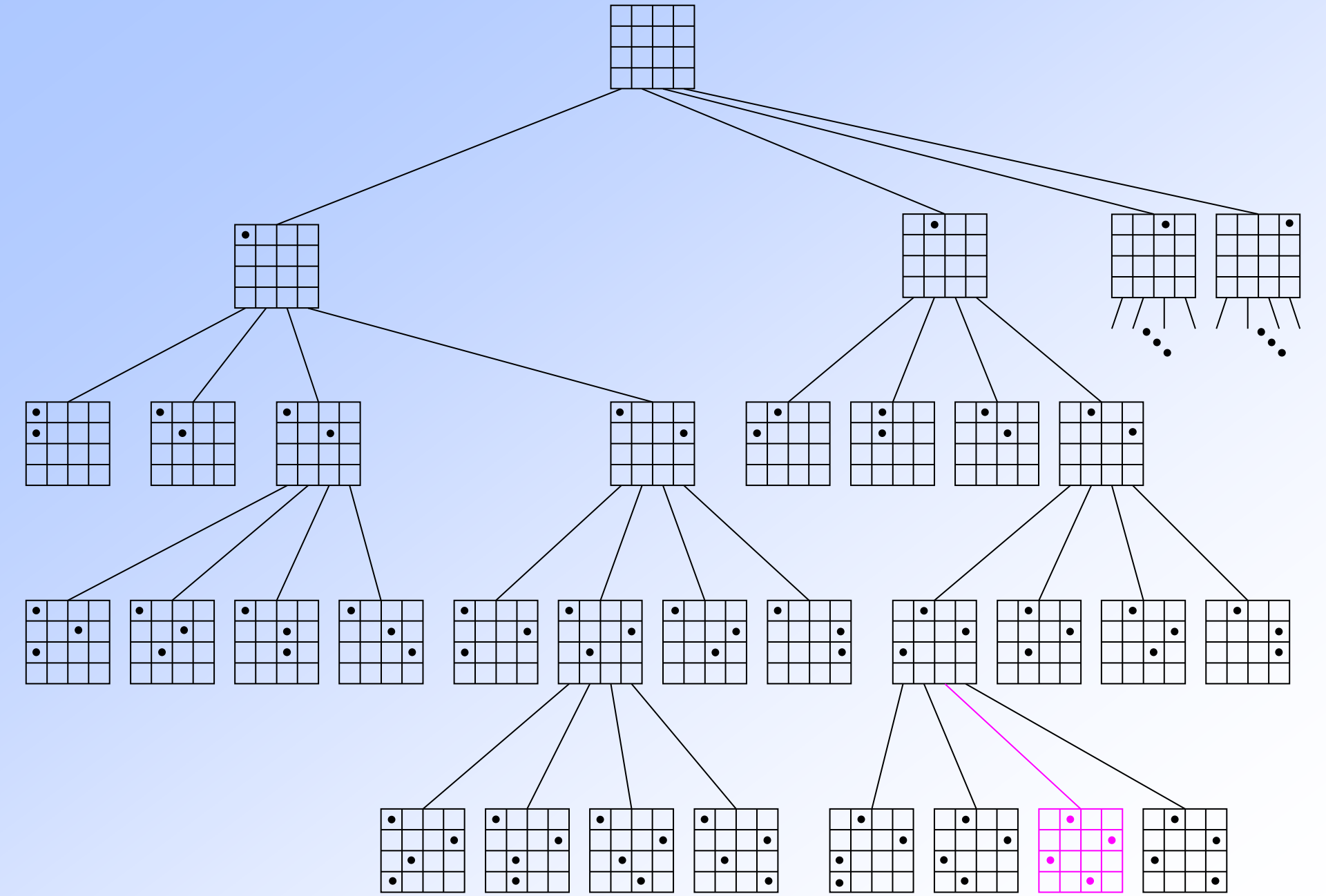
在 n 行 n 列的国际象棋棋盘上，若两个皇后位于同一行、同一列、同一对角线上，则称为它们为互相攻击。n皇后问题是指找到这 n 个皇后的互不攻击的布局。


k = i+j
0 1 2 3
0
1
2
3
0#次对角线
1#次对角线
2#次对角线
3#次对角线
4#次对角线
5#次对角线
6#次对角线
0#主对角线
1#主对角线
2#主对角线
3#主对角线
4#主对角线
5#主对角线
6#主对角线
k = n+i-j-1

解题思路

- 安放第 i 行皇后时，需要在列的方向从 0 到 n-1 试探 (j = 0, …, n-1)
- 在第 j 列安放一个皇后：
  - 如果没有出现攻击，在第 j 列安放的皇后不动，递归安放第 i+1行皇后。
  - 如果在列、主对角线、次对角线方向有其它皇后，则出现攻击，撤消在第 j 列安放的皇后。

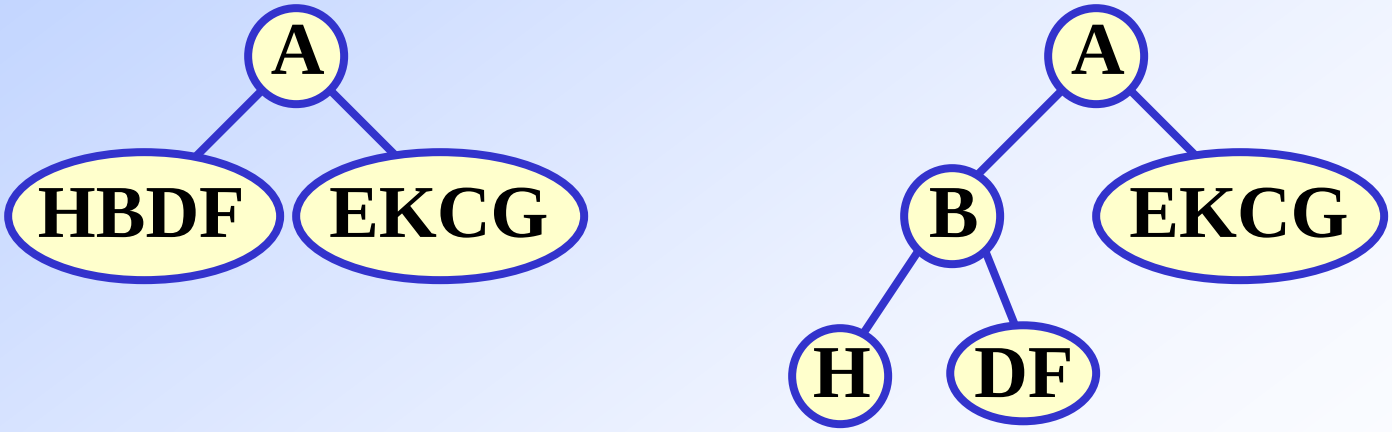


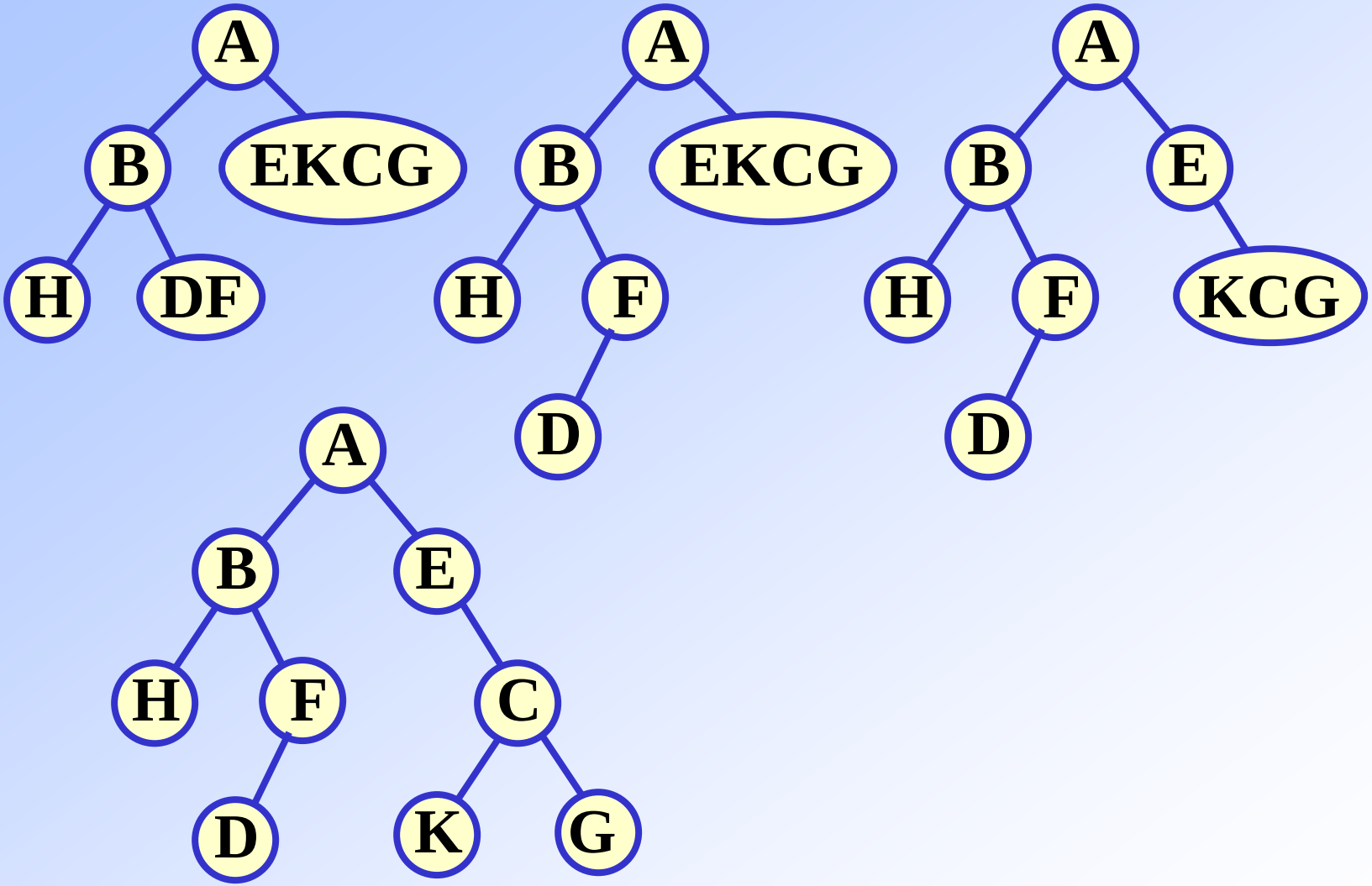


二叉树的计数

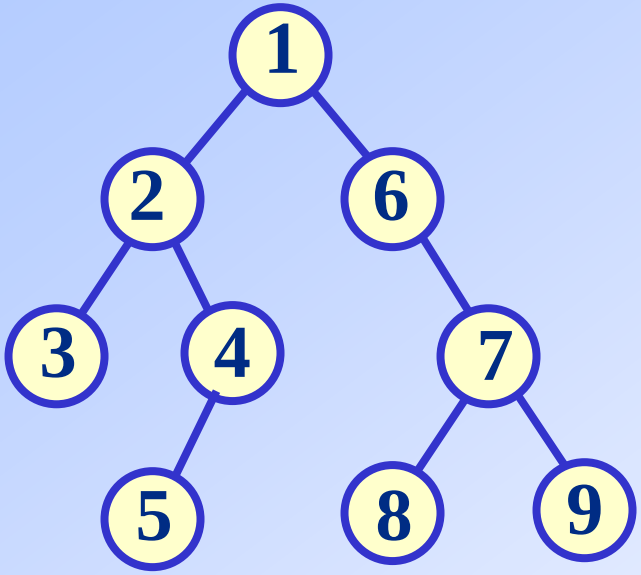
- 由二叉树的前序序列和中序序列可唯一地确定一棵二叉树。

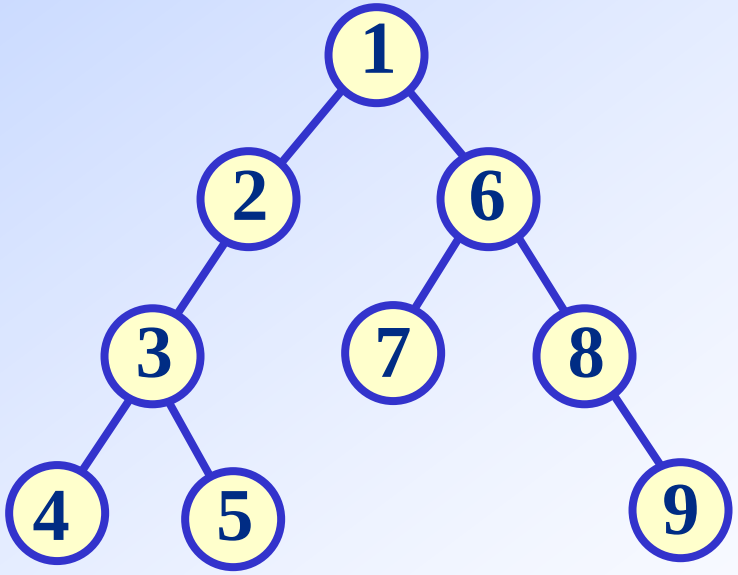
例，前序序列{ABHFDECKG}和中序序列{HBDFAEKCG}，构造二叉树过程如下：




A
B
EKCG
H
DF
A
B
EKCG
H
F
D
A
B
E
H
F
KCG
D
A
B
E
H
F
C
D
K
G

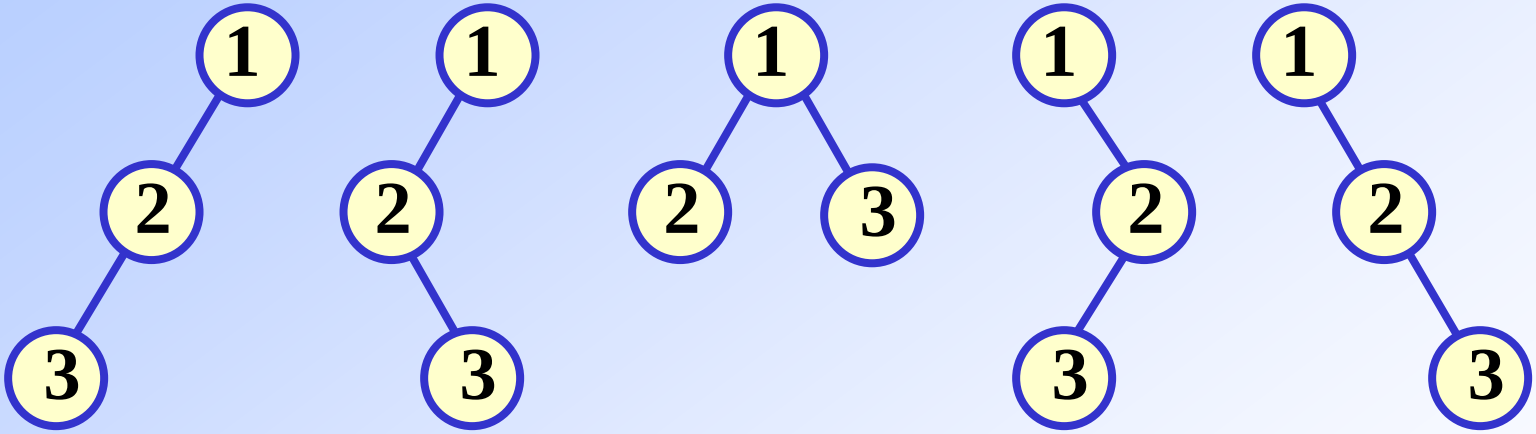
如果前序序列固定不变，给出不同的中序序列，可得到不同的二叉树。



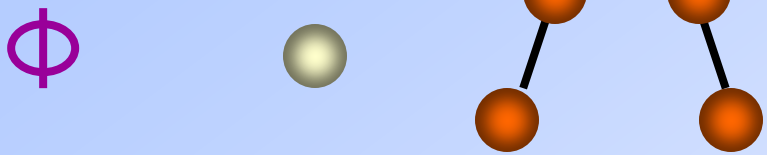


- 固定前序排列,选择所有可能的中序排列,可以构造多少种不同的二叉树?

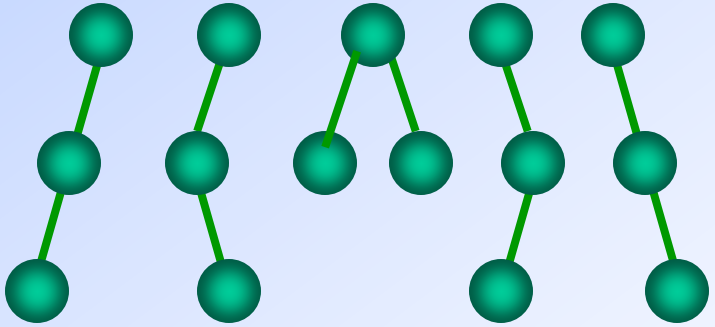
例如，有 3 个数据 { 1, 2, 3 }，可得 5 种不同的二叉树。它们的前序排列均为 123，中序序列可能是 321， 231， 213， 132，123.



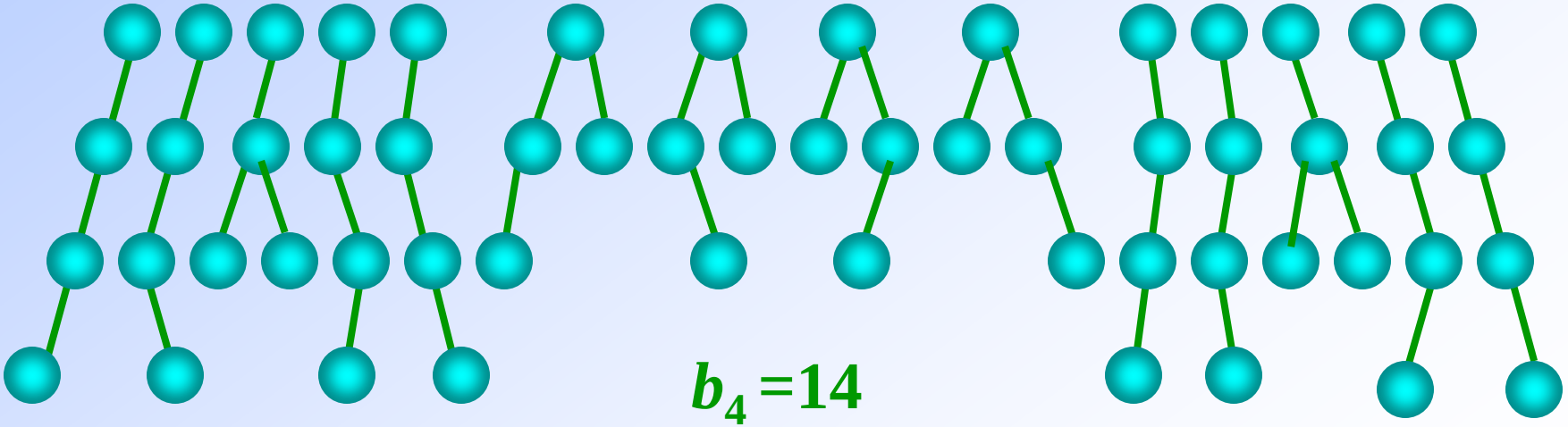
有0个, 1个, 2个, 3个、4个结点的不同二叉树



$b_0=1$ $b_1=1$ $b_2=2$



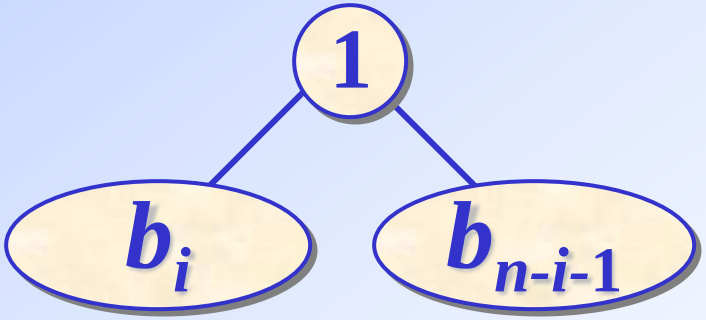
$b_3=5$



$b_4=14$

❖计算具有 n 个结点的不同二叉树的棵数

$$b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1}$$

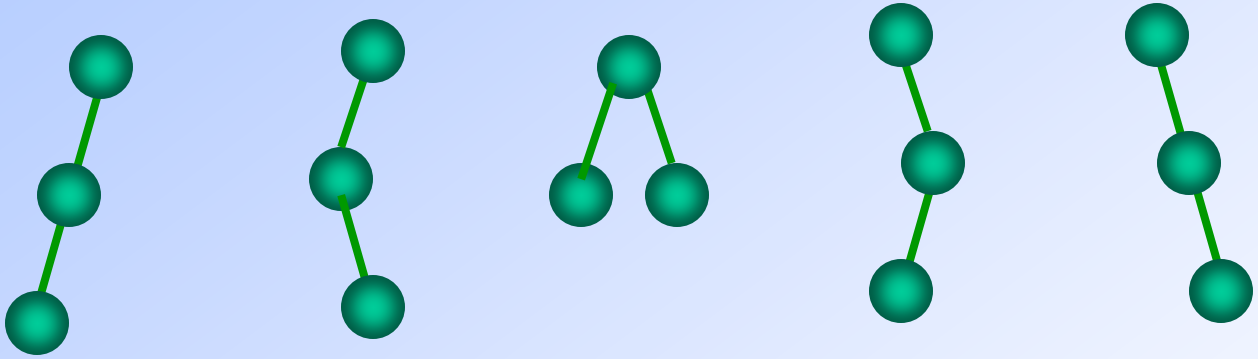


最终结果：

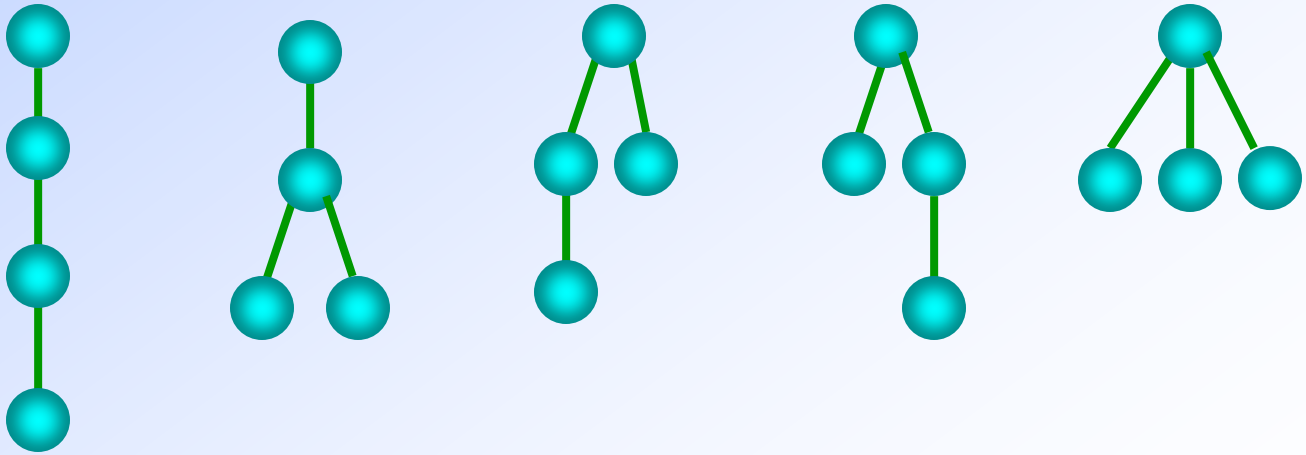
$$b_n = \frac{1}{n+1} C_{2n}^{n} = \frac{1}{n+1} \frac{(2n)!}{n! \cdot n!}$$

- 具有n个结点不同形态的树的数目和具有n-1个结点互不相似的二叉树的数目相同。

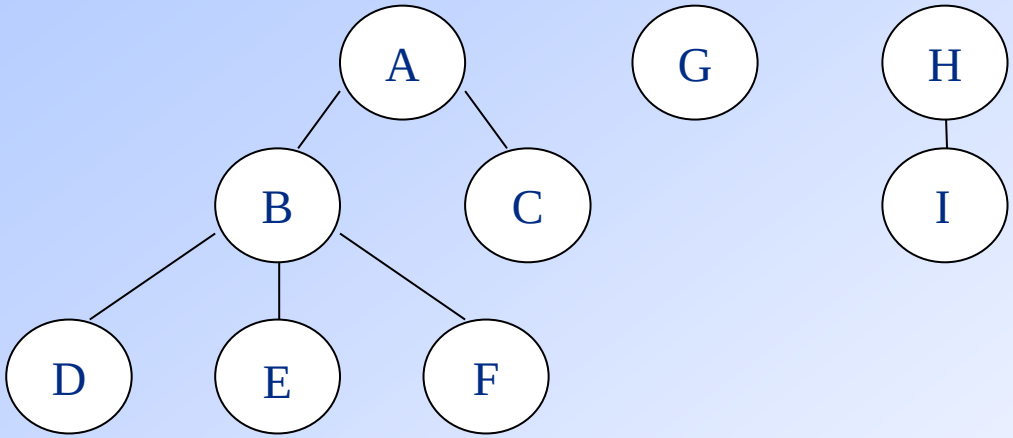
有3个结点的不同二叉树如下

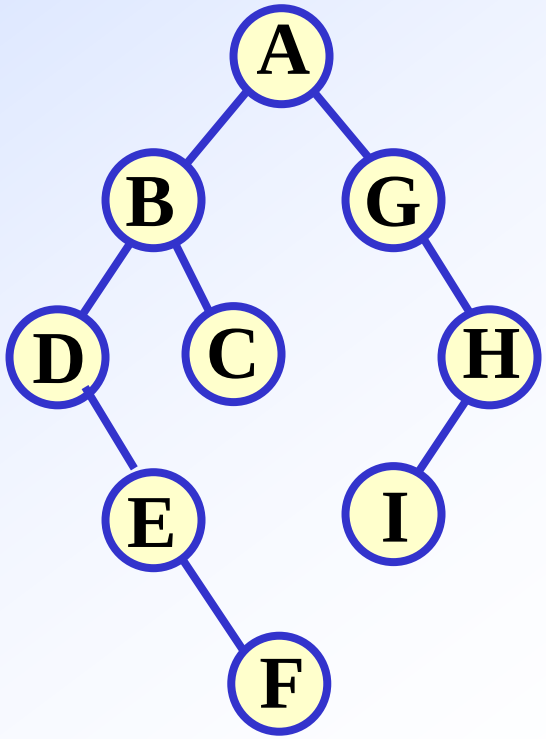


有4个结点的不同形态的树如下



一、将下图所示具有三棵树的森林转化成对应的二叉树形式，并写出中序遍历的结果。





遍历DEFBCAGIH

二、已知一棵有2011个结点的树，其叶子结点个数为116，该树对应的二叉树中无右孩子的结点个数是

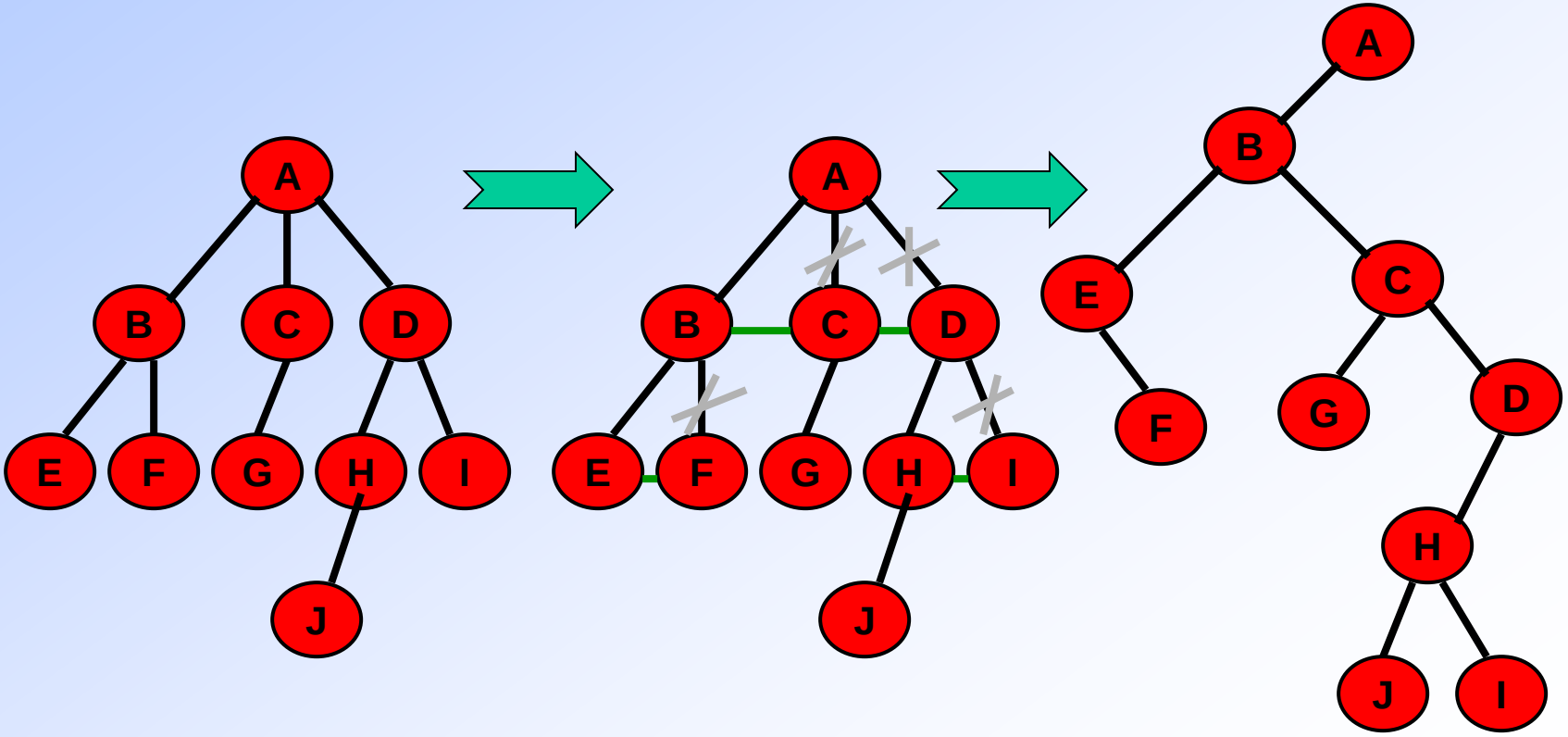
1896

说明：二叉树中无右孩子的结点个数为树（或森林）中非叶子结点个数加1

树转换成相对应的二叉树：

1）保留每个结点的最左面的分支，其余分支都被删除。

2）同一父结点下面的结点成为它的左方结点的兄弟。



- 三、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是0.07, 0.19, 0.02, 0.06, 0.32, 0.03，0.21，0.10。请设计它们相应的哈夫曼编码。使用0～7的二进制表示形式是另一种编码方案，请比较两种方案的优缺点。