第四章 数据结构

- 4.1 基本概念
- 4.2 线性表
- 4.3 栈和队列
- 4.4 树
- 4.5 二叉树
  - 4.5.1 二叉树的存储形式
  - 4.5.2 二叉树的遍历
  - 4.5.3 二叉树的顺序存储
  - 4.5.4 穿线树
- 4.6 二叉树的线性表示

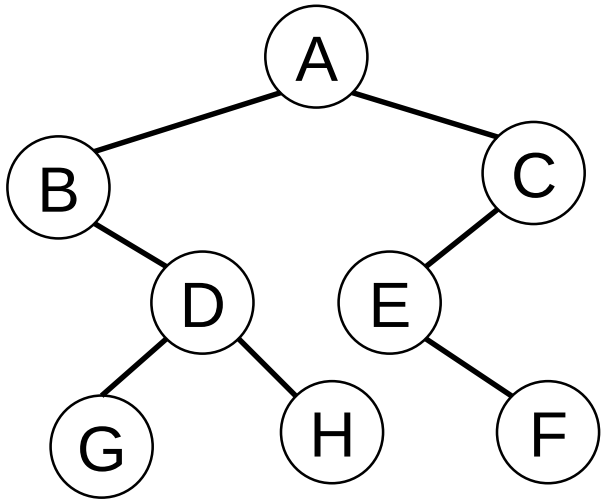
➢4.5 二叉树

➢二叉树的定义

在一棵二次树中，若规定后件是有序的，即对于任何一个结点，规定它的第一后件称为左子(左后件，左件)，第二后件称为右子(右后件，右件)，那么这是一棵二次有序树，并被称为二叉树(Binary Tree)。

二叉树的结点可以既有左子又有右子，也可以只有左子，或者只有右子，甚至没有后件。

用图形表示时，左子画在左下方，右子画在右下方。

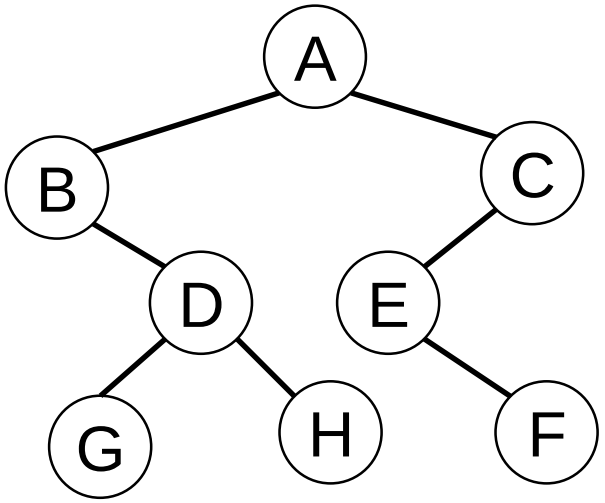


➢4.5.1 二叉树的存储形式

➢标准形式

设置两个指针场分量($p_{left}k$ 和$p_{right}k$)，分别指向左子和右子。

α k	δ k	$p_{left}k$	$p_{right}k$
10	A	20	60
20	B	Φ	30
30	D	40	50
40	G	Φ	Φ
50	H	Φ	Φ
60	C	70	Φ
70	E	Φ	100
100	F	Φ	Φ



➢4.5.1 二叉树的存储形式

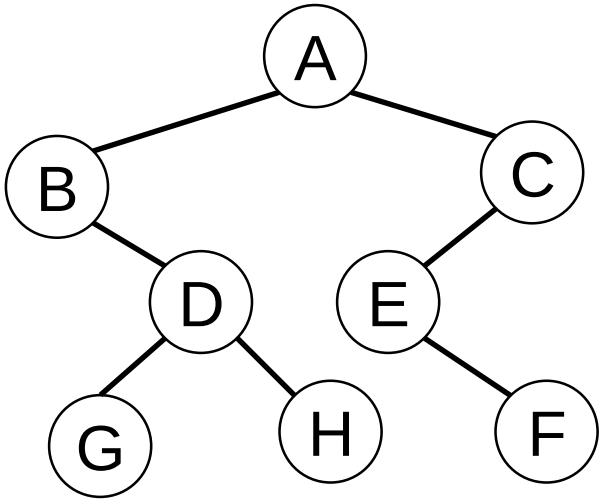
- *标准形式*
- 逆形式

设置一个指针场分量$p_{pre}k$，指向父结点。

- 扩充的标准形式

将标准形式和逆形式结合在一起，即$p_{pre}k$指向父结点，$p_{left}k$和$p_{right}k$分别指向左子和右子。

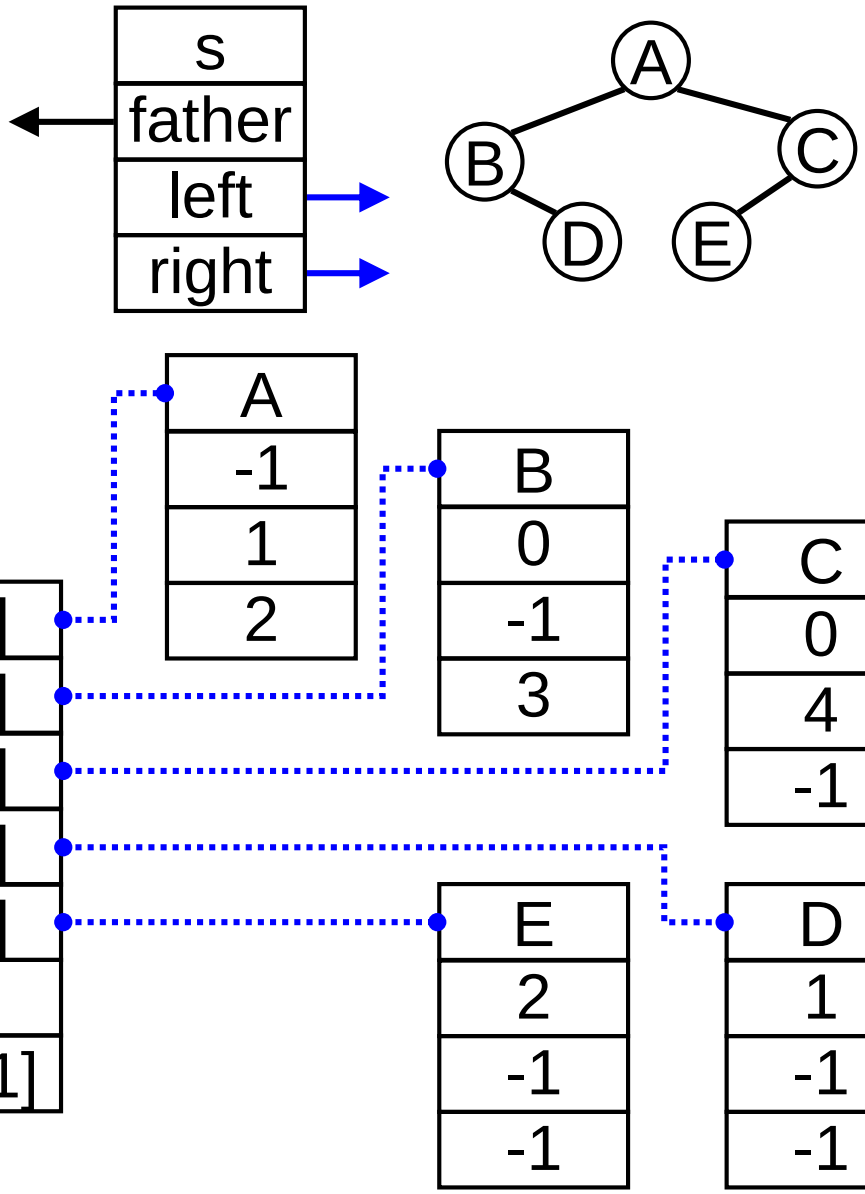
α k	δ k	$p_{pre}k$	$p_{left}k$	$p_{right}k$
10	A	Φ	20	60
20	B	10	Φ	30
30	D	20	40	50
40	G	30	Φ	Φ
50	H	30	Φ	Φ
60	C	10	70	Φ
70	E	60	Φ	100
100	F	70	Φ	Φ



➢用数组存储二叉树的扩充标准形式

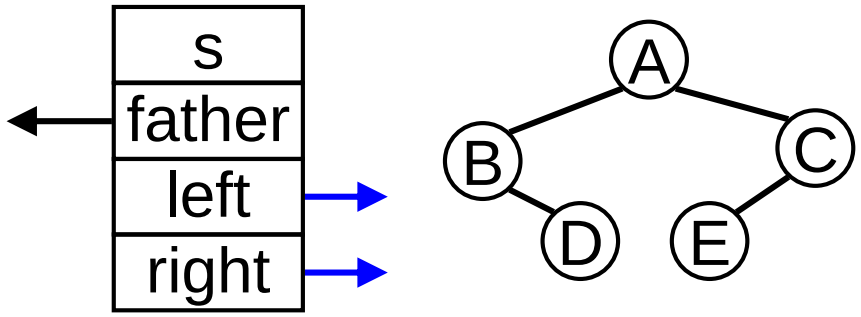
```
#define BINODE struct binode
BINODE
{
   char *s;
   short father, left, right;
};
BINODE Tree[100];
short Root;
```

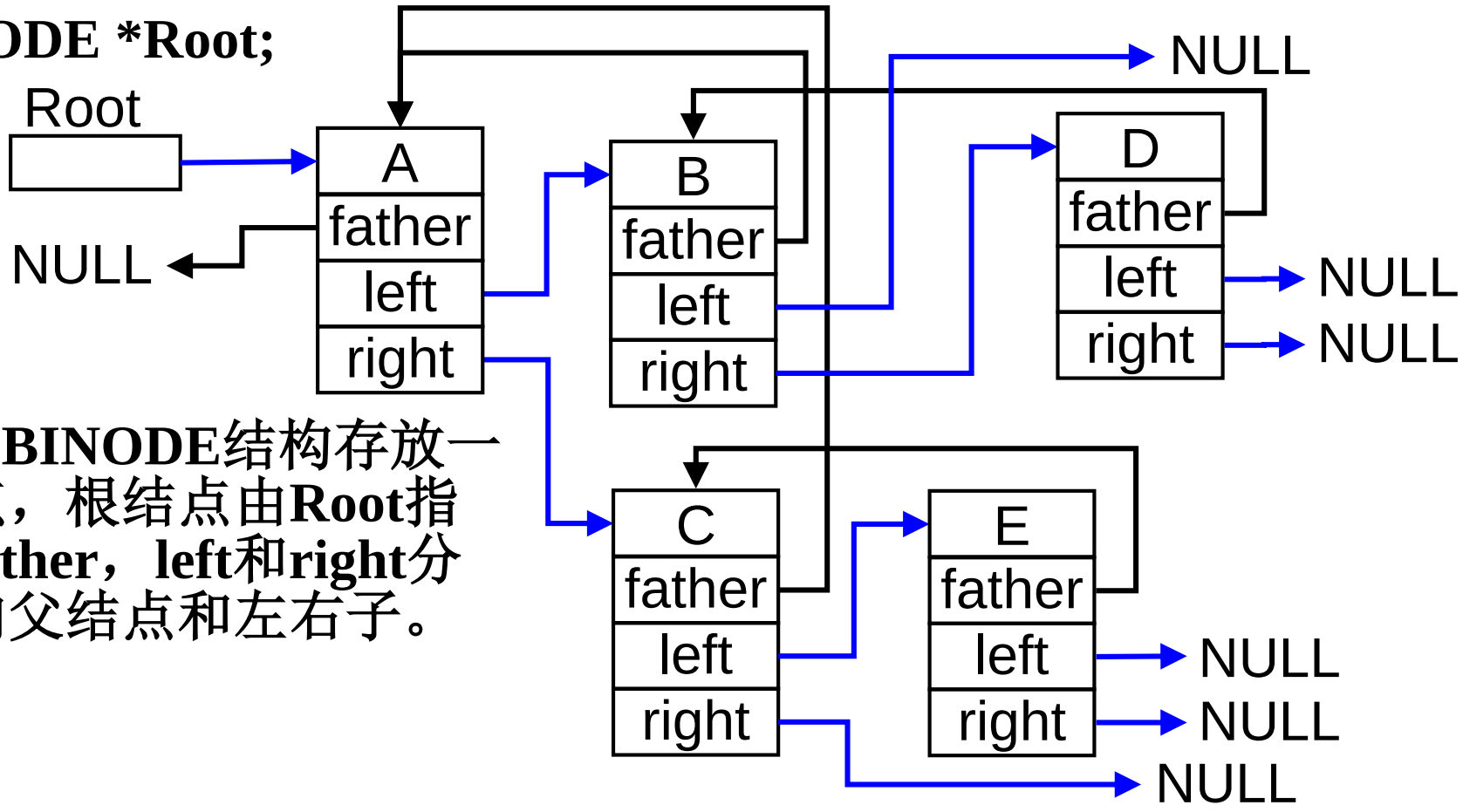
每个数组元素Tree[i](i=0,...,n-1)存放一个结点。用数组下标表示结点地址，用-1表示空地址。定义变量Root指向根结点。结点成员father，left和right分别指向父结点和左右子。



➢链接存储二叉树的扩充标准形式

```
#define BINODE struct binode
BINODE
{
   char *s;
   BINODE *father, *left,*right;
};
BINODE *Root;
```

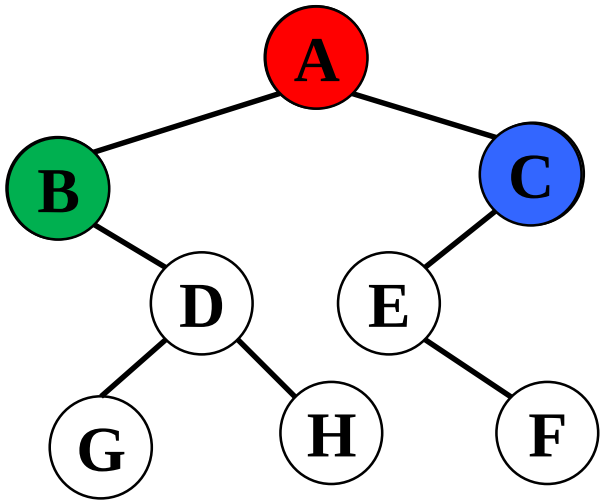




每个**BINODE**结构存放一个结点，根结点由**Root**指向，**father**，**left**和**right**分别指向父结点和左右子。

➢5.5.2 二叉树的遍历

- 前序遍历(PreOrder)



  先访问根
  按前序遍历左子树
  按前序遍历右子树

  图示树的前序遍历结果：

  A B D G H C E F

- 后序遍历(PostOrder)

  按后序遍历左子树
  按后序遍历右子树
  最后访问根

  图示树的后序遍历结果：

  G H D B F E C A

- 中序遍历(SymOrder)

  按中序遍历左子树
  访问根
  按中序遍历右子树

  图示树的中序遍历结果：

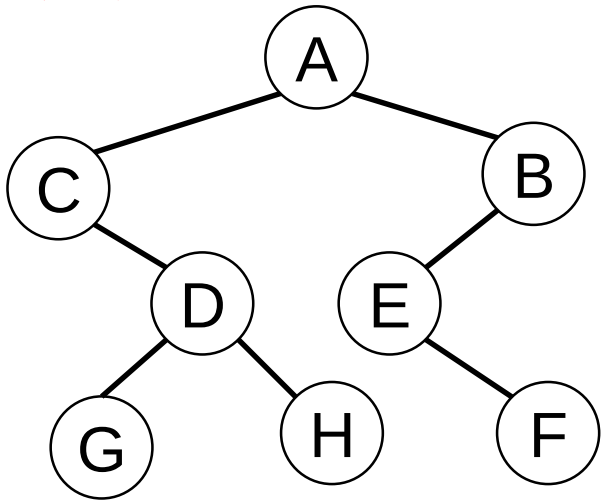
  B G D H A E F C

【例4-5.1】二叉树标准形式的数组编程

- 编程要求
  用数组存储树的标准形式
  用递归函数实现二叉树的中序遍历
  (网络课堂：exe4-5.1BiTree.c)
- 程序中的数据定义

```
#define BINODE struct binode
BINODE
{
   char  s[10];
   short left, right;
};
BINODE T[100]; 结构数组存储
short n;
```

每个数组元素T[i] (i=0,...,n-1)存放一个结点。用数组下标表示结点地址，用-1表示空地址。定义变量root指向根结点。结点成员left和right分别指向左右子。



i	T[i].s	left	right
0	A	2	1
1	B	4	-1
2	C	-1	3
3	D	6	7
4	E	-1	5
5	F	-1	-1
6	G	-1	-1
7	H	-1	-1

- 函数实现

```
void CreateBitree(short *n, short *root)
{
    int i;
    scanf("%d%d", n, root);
    for(i=0; i<*n; i++)
        scanf("%s%d%d",
            T[i].s, &T[i].left, &T[i].right);
}
main()
{
    short root;
    /* 输入并生成二叉树          */
    CreateBitree(&n, &root);
    /* 中序遍历函数              */
    symOrder(root);
}
```

输入数据为:

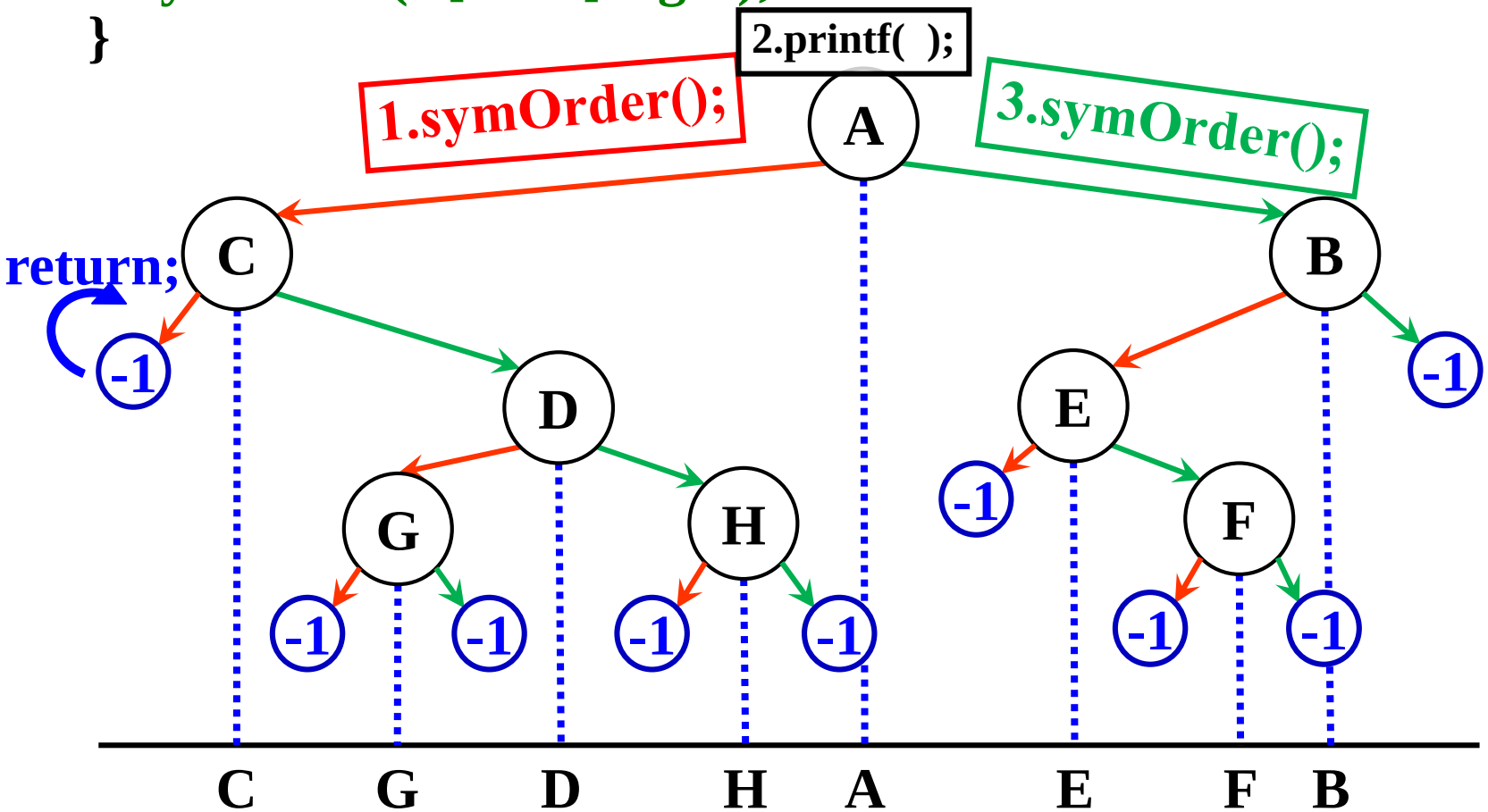
```
8   0
A   2   1
B   4  -1
C  -1   3
D   6   7
E  -1   5
F  -1  -1
G  -1  -1
H  -1  -1
```

i	T[i].s	left	right
0	A	2	1
1	B	4	-1
2	C	-1	3
3	D	6	7
4	E	-1	5
5	F	-1	-1
6	G	-1	-1
7	H	-1	-1

- 中序遍历函数及运行过程示例

```
symOrder(short node)
{
   if(node == -1) return;
   symOrder(T[node].left);
   printf("%s\n", T[node].s);
   symOrder(T[node].right);
}
```

```
main() {
   short root;
   CreateBitree(&n, &root);
   /* 中序遍历函数*/
   symOrder(root);
}
```



【例4-5.2】二叉树标准形式的链接编程

- 编程要求

  输入用括号表示的二叉树，例如：

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

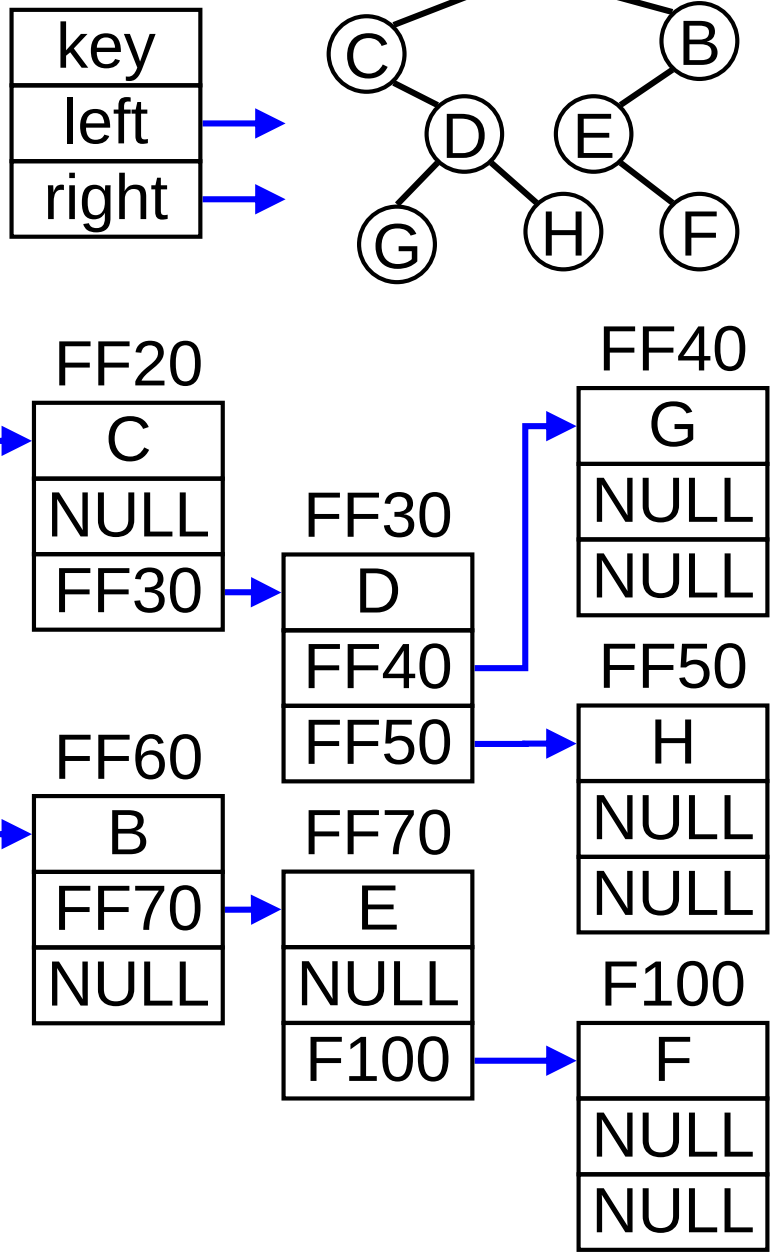
  用链接的结构存储树的标准形式

  用递归函数实现二叉树的后序遍历

  (网络课堂：exe4-5.2BiTree.c)

- 程序实现

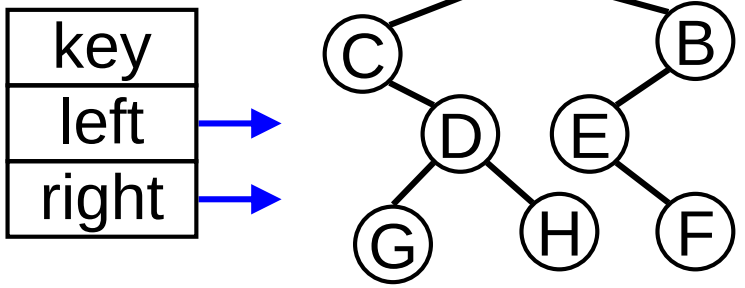
```
#define BINODE struct binode
BINODE
{
   char  *key;
   BINODE *left,*right;
};
BINODE *Root;
```

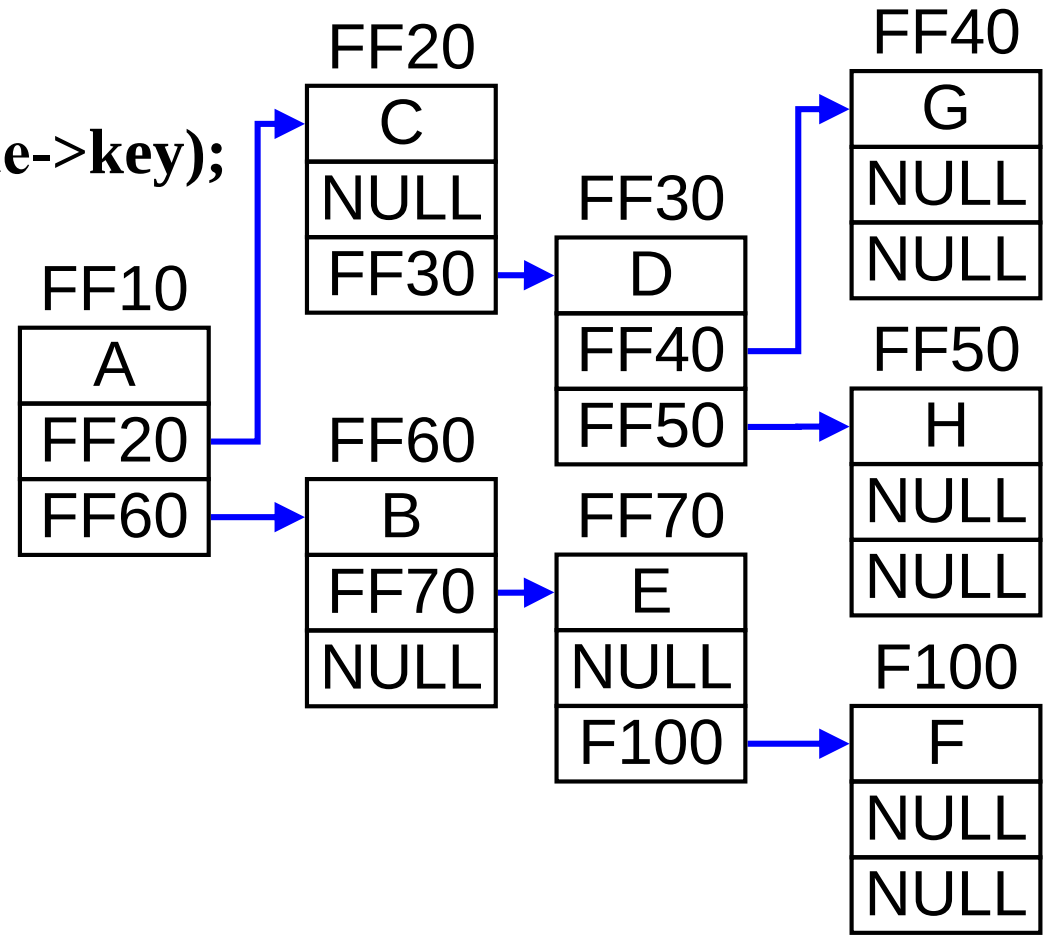


链接存储

【例4-5.2】二叉树标准形式的链接编程

```
void  postOrder(BINODE *node)
{
    if (!node)
        return;
    postOrder(node->left);
    postOrder(node->right);
    printf("node=%s\n",node->key);
}
void CreateBitree(void)
{ 见：exe4-5.2BiTree.c  }
main()
{
    /* 输入并生成二叉树 */
    CreateBitree();
    /* 后序遍历          */
    postOrder(Root);
}
```





➢4.5.3 二叉树的顺序存储

➢二叉树的顺序存储

顺序存储：若对于任何一对结点(k,k')都成立αk'=αk+s(s为存储单元的大小,固定长度), 则称r是用顺序方式存储的,简称顺序存储。

根据树的定义，每个结点的后件不止一个。严格地说，因为无法满足αk'=αk+s (k是k'的前件)，而只能用链接方式进行存储。

但是如果采用某种约定，例如根据某种遍历方式，可以实现树的顺序存储。因为遍历是一种线性的关系，当结点k是结点k'关于某种遍历的前件时，可以满足αk'=αk+s。

在二叉树的存储中，往往采用这种修正的顺序存储方式。例如，按照前序遍历实现二叉树的顺序存储，可以满足前序遍历的前后件关系，但是二叉树中每个结点还有另外一个结点，必须采取一定的约定，或者说需要附加某种存储关系。

➢采用flag约定的前序遍历顺序存储

- 约定顺序存储的定义

修正树的存储单元定义，例如，将k={αk, δk, pk}扩充为k={αk, δk, pk, flag}，即增加一个flag标志，以说明结点另外一个后件(非遍历后件)的存储信息。由于二叉树中结点的后件最多是两个，因此可以用pk和flag来存储结点的后件。

假定采用以下的flag约定的存储定义：

$$pk=\begin{cases}\alpha k_R & \text{结点k有右子}k_R\\ \phi\text{即-1} & \text{结点k无右子}k_R\end{cases}\qquad flag=\begin{cases}1 & \text{结点k有左子}k_L\\ 0 & \text{结点k无左子}k_L\end{cases}$$

规定结点k的右子k_R地址用指针场pk表示。而左子k_L的地址信息用flag说明，如果结点有左子k_L，则必定是前序遍历的后件。

因此若flag为1，表示结点k有左子k_L，即$\alpha k_L=\alpha k+s$，否则表示结点k没有左子k_L。

- **flag约定的存储单元示例**

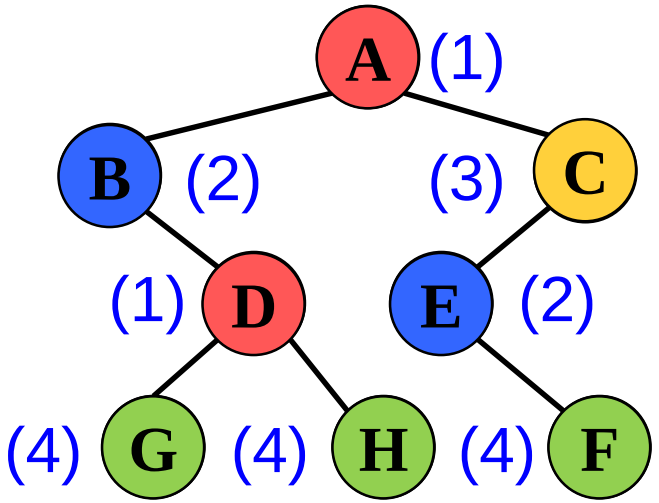
根据约定，pk和flag的取值有四种情况，分别是：

(1)结点k有左件k_L，有右件k_R
则flag=1，pk=αk_R，$\alpha k_L=\alpha k+s$
如结点A和D。

(2)结点k无左件k_L，有右件k_R
则flag=0，pk = αk_R
如结点B和E。

(3)结点k有左件k_L，无右件k_R
则flag=1，pk=-1，$\alpha k_L=\alpha k+s$
如结点C。

(4)结点k无左件k_L，无右件k_R
则flag=0， pk=-1
如结点G，H和F。



情况	αk	δk	pk	flag
(1)	0	A	5	1
(2)	1	B	2	0
(1)	2	D	4	1
(4)	3	G	-1	0
(4)	4	H	-1	0
(3)	5	C	-1	1
(2)	6	E	7	0
(4)	7	F	-1	0

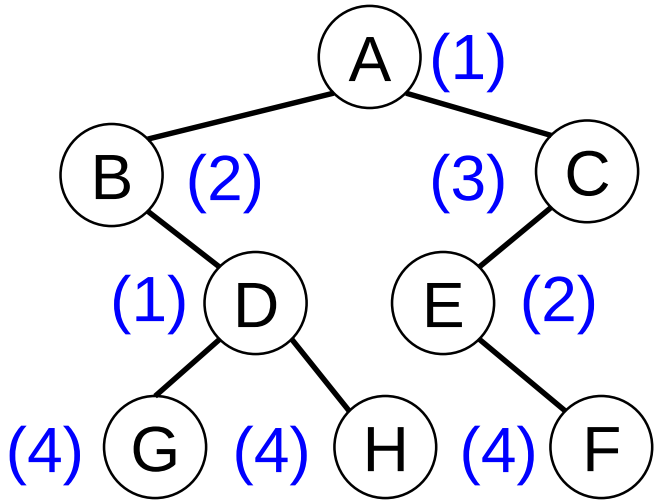
- **flag约定的顺序存储示例**

数据定义

```
#define  M  100
#define BITREE struct bitree
BITREE
{
   char *s;
   short p, flag;
};
BITREE  tree[M];
short n;                /*结点数*/
```

- **前序遍历函数**

```
void preOrder(BITREE *tree)
{
   int i;
   for(i=0; i<n ; i++)
      printf("node=%s\n", tree[i].s);
}
```

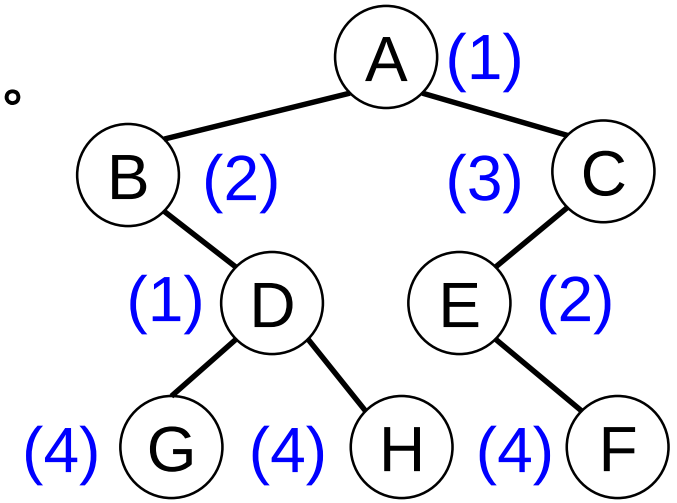


情况	αk	δk	pk	flag
(1)	0	A	5	1
(2)	1	B	2	0
(1)	2	D	4	1
(4)	3	G	-1	0
(4)	4	H	-1	0
(3)	5	C	-1	1
(2)	6	E	7	0
(4)	7	F	-1	0

- 打印前序前件的程序(数组前一元素)

  根据结点k的地址j，打印其前序前件。

```
if(j==0)
    printf("no predecessor\n");
else
    printf("predecessor is %s\n",
            tree[j-1].s);
```



情况	αk	δk	pk	flag
(1)	0	A	5	1
(2)	1	B	2	0
(1)	2	D	4	1
(4)	3	G	-1	0
(4)	4	H	-1	0
(3)	5	C	-1	1
(2)	6	E	7	0
(4)	7	F	-1	0

- 打印左子的程序(取决于flag)

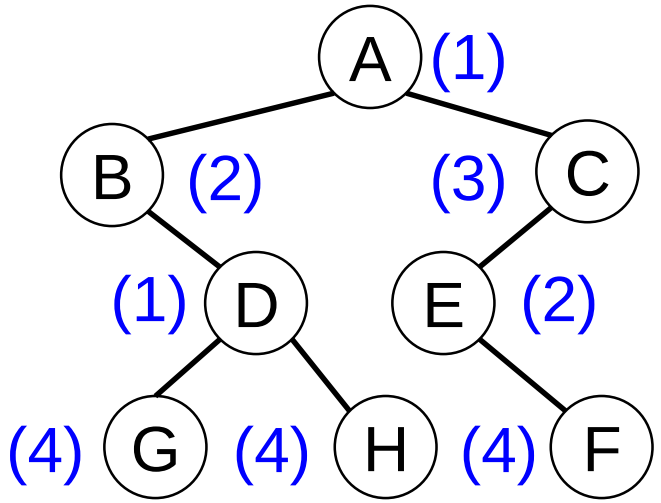
根据结点k的地址j，打印其左件。

```
if(tree[j].flag == 0)
    printf("no left child\n");
else
    printf("left child is  %s\n", tree[j+1].s);
```

- 打印右子的程序(取决于pk)

根据结点k的地址j，打印其右件。

```
if(tree[j].p == -1)
    printf("no right child\n");
else
    printf("right child is %s\n",
            tree[tree[j].p].s);
```



情况	αk	δk	pk	flag
(1)	0	A	5	1
(2)	1	B	2	0
(1)	2	D	4	1
(4)	3	G	-1	0
(4)	4	H	-1	0
(3)	5	C	-1	1
(2)	6	E	7	0
(4)	7	F	-1	0

➢采用flag约定的前序遍历顺序存储

pk | flag	$pk = \alpha k_R$ 结点k有右子k_R	$pk = \phi$ 即-1 结点k无右子k_R
flag =1 结点k有左子k_L	αk_R | 1	ϕ 即-1 | 1
flag =0 结点k无左子k_L	αk_R | 0	ϕ 即-1 | 0

➢采用一个指针场的前序遍历顺序存储

pk	$\|pk\| = \alpha k_R < M$ 结点k有右子k_R	$\|pk\| = M$ 结点k无右子k_R
pk>0 结点k有左子k_L	αk_R	M
pk<0 结点k无左子k_L	$-\alpha k_R$	-M

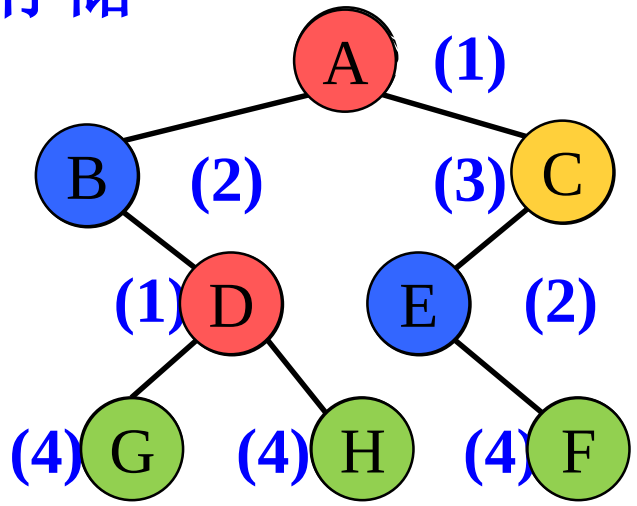
➢采用一个指针场的前序遍历顺序存储

pk>0：有左件， $\alpha k_L=\alpha k+s$

pk<0：无左件

|pk|!=M：有右件， $\alpha k_R=|pk|$

|pk|==M：无右件



(1)结点k有左件k_L，有右件k_R
则pk>0且|pk|!=M，如结点A和D。

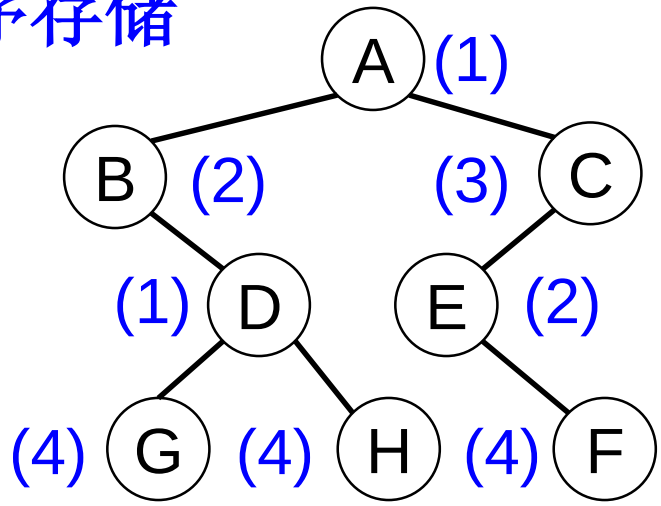
(2)结点k无左件k_L，有右件k_R
则pk<0且|pk|!=M，如结点B和E。

(3)结点k有左件k_L，无右件k_R
则pk>0且|pk|==M，如结点C。

(4)结点k无左件k_L，无右件k_R
则pk<0且|pk|==M，如结点G，H和F。

情况	αk	δk	pk
(1)	0	A	5
(2)	1	B	-2
(1)	2	D	4
(4)	3	G	-100
(4)	4	H	-100
(3)	5	C	100
(2)	6	E	-7
(4)	7	F	-100

➢采用一个指针场的前序遍历顺序存储



- 数据定义

  设存储单元为k={αk, δk, pk}，存放结点的数组定义为

```
#define  M  100
#define ABS(x)   (x>0) ? (x) : (-x)
#define BITREE struct bitree
BITREE
{
   char *s;
   short p;
};
BITREE  tree[M];
short n;    /* 结点数n<=M     */
```

- pk的定义

  (1) k有左件k_L，有右件k_R，则$pk=\alpha k_R$

  (2) k无左件k_L，有右件k_R，则$pk=-\alpha k_R$

  (3) k有左件k_L，无右件k_R，则$pk=M$

  (4) k无左件k_L，无右件k_R，则$pk=-M$

情况	αk	δk	pk
(1)	0	A	5
(2)	1	B	-2
(1)	2	D	4
(4)	3	G	-100
(4)	4	H	-100
(3)	5	C	100
(2)	6	E	-7
(4)	7	F	-100

➢采用一个指针场的前序遍历顺序存储

- 确定结点k的左子和右子地址的法则

  若pk>0，

  　则结点k有左件，且$\alpha k_L=\alpha k+s$。

  若pk<0，

  　则结点k无左件。

  若|pk|!=M，

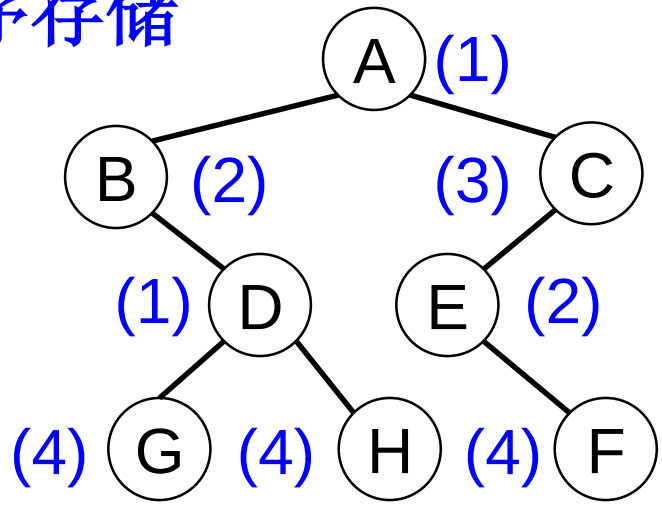
  　则结点k有右件，且$\alpha k_R=|pk|$。

  若|pk|==M，

  　则结点k无右件。

- 前序遍历函数

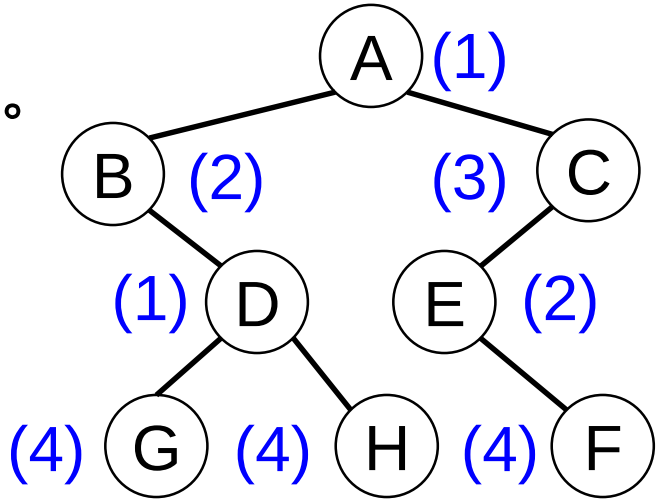
```
void preOrder(BITREE *tree)
{
    int i;
    for(i=0; i<n ; i++)
        printf("node=%s\n", tree[i].s);
}
```



情况	αk	δk	pk
(1)	0	A	5
(2)	1	B	-2
(1)	2	D	4
(4)	3	G	-100
(4)	4	H	-100
(3)	5	C	100
(2)	6	E	-7
(4)	7	F	-100

- 打印前序前件的程序(数组前一元素)
  根据结点k的地址j，打印其前序前件。

```
if(j==0)
    printf("no predecessor \n");
else
    printf("predecessor is %s\n",
            tree[j-1].s);
```



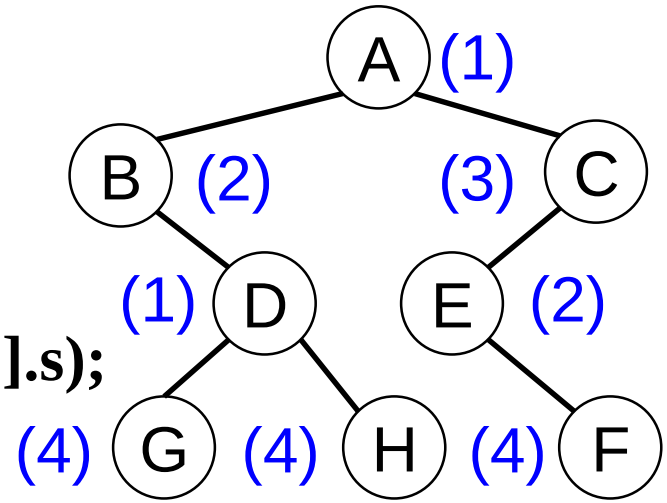
情况	αk	δk	pk
(1)	0	A	5
(2)	1	B	-2
(1)	2	D	4
(4)	3	G	-100
(4)	4	H	-100
(3)	5	C	100
(2)	6	E	-7
(4)	7	F	-100

- 打印左子的程序(判pk>0)
  根据结点k的地址j，打印其左件。

```
if(tree[j].p < 0)
    printf("no left child\n");
else
    printf("left child is %s\n", tree[j+1].s);
```

- 打印右子的程序(判绝对值|pk|)
  根据结点k的地址j，打印其右件。

```
if (ABS(tree[j].p) == M)
    printf("no right child \n");
else
    printf("right child  is %s \n",
            tree[ABS(tree[j].p)].s);
```



情况	αk	δk	pk
(1)	0	A	5
(2)	1	B	-2
(1)	2	D	4
(4)	3	G	-100
(4)	4	H	-100
(3)	5	C	100
(2)	6	E	-7
(4)	7	F	-100

➢4.5.4 穿线树

➢穿线树

考察采用标准形式存储二叉树的情况。若有n个结点，共2n个指针场分量。因为根结点没有前件，其他每个结点只有一个父结点，因而共有n-1个指针场分量被利用，而n+1个被空置(浪费)。为了提高指针的利用率，我们引入穿线树的概念。

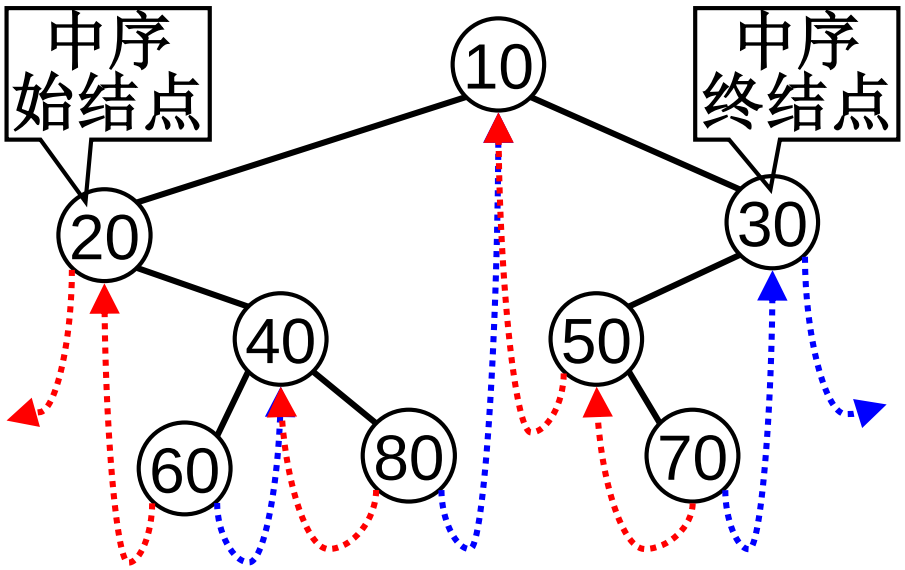
➢利用空闲指针场形成穿线树的方法

对空闲的左子指针，用于指向其中序前件。

对空闲的右子指针，用于指向其中序后件。

这样被利用的指针有n-1个，则最后只有两个指针指向空，而不能再少。其中一个为中序始结点，另一个为中序终结点。

用虚线表示穿线树的前后件关系，以便与二叉树的前后件关系有所区别。



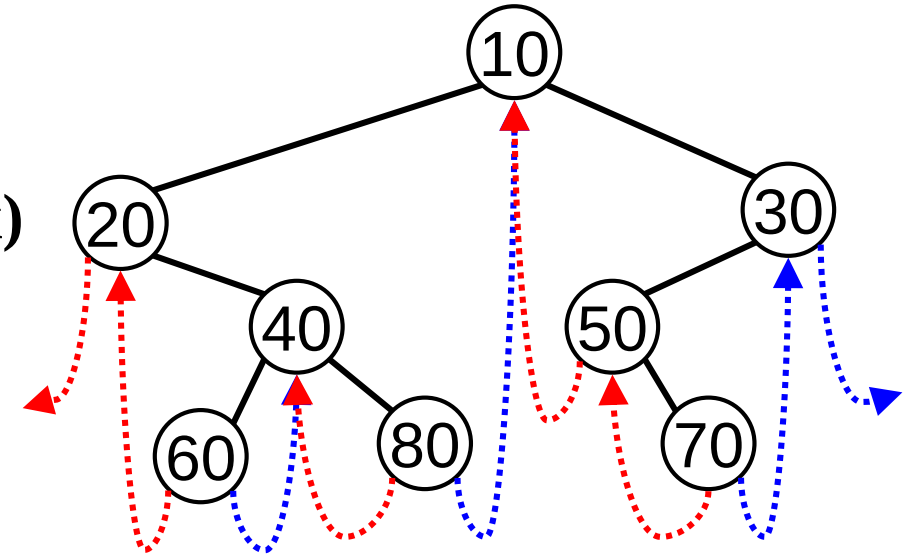
➢穿线树的存储单元定义

```
#define  M  100
#define ABS(x)   (x>0) ? (x) : (-x)
#define TH struct thread
TH
{
   short  num, pL, pR;
};
TH th[M];
short n;       /* 结点数                    */
```

按标准形式用结构数组存放穿线树。

其中， 用th[1]~th[n]存放n个结点，用th[0].num存放根结点的地址，本例中th[0].num=1。

用0表示空地址ϕ。穿线指针场的值(中序前件或者中序后件的地址)用负数表示。对中序始结点tb(如结点20)，必定有th[tb].pL==0。对中序终结点te(如结点30)，必定有th[te].pR==0。

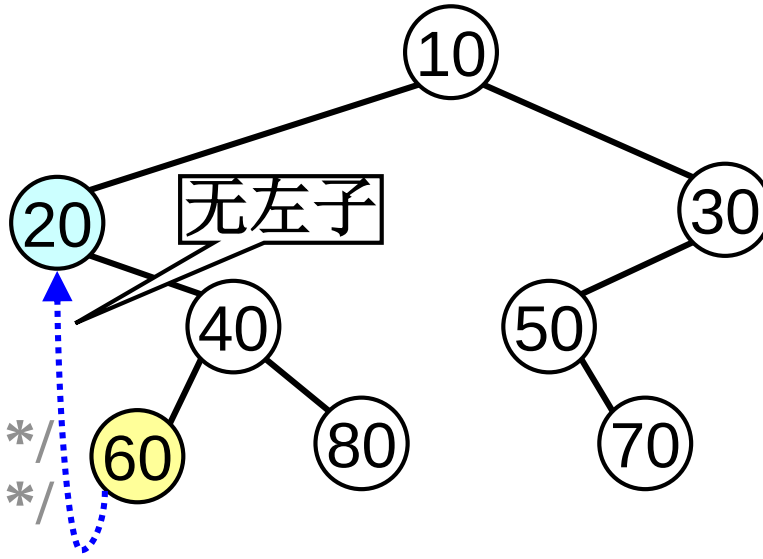


αk	δk	pL	pR
1	10	2	6
2	20	0	3
3	40	4	5
4	60	-2	-3
5	80	-3	-1
6	30	7	0
7	50	-1	8
8	70	-7	-6

➢穿线树的处理函数

- 根据结点k的地址i，求其中序前件k’

```
short getPre(TH *th, short i)
{
   if(i<1 || i>n)        /* 非法地址 */
      error();
   if((i=th[i].pL) > 0)        /* 有左子 */
      while(th[i].pR > 0)    /* 有右子 */
         i = th[i].pR;
   return(ABS(i));
}
```



αk	δk	pL	pR
1	10	2	6
2	**20**	0	3
3	40	4	5
4	60	**-2**	-3
5	80	-3	-1
6	30	7	0
7	50	-1	8
8	70	-7	-6

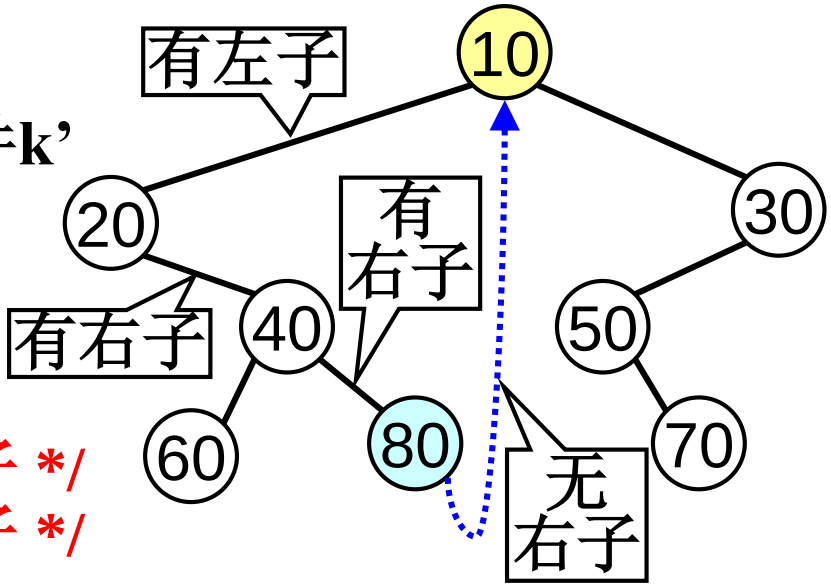
- 例如已知i=4，求结点60的中序前件k’：
  th[4].pL(=-2) < 0，无左子，得
  αk’=|-2|=2，则δk’ = 20(即th[2].num)。
  函数调用为：

```
if(k= getPre(th, 4))
   printf("prev node of %d is %d\n",
            th[i].num, th[k].num);
else
   printf("no prev node for %d.\n", th[i].num);
```

➢穿线树的处理函数

- 根据结点k的地址i，求其中序前件k’

```
short getPre(TH *th, short i)
{
   if(i<1 || i>n)    /* 非法地址 */
      error();
   if((i=th[i].pL) > 0)     /* 有左子 */
      while(th[i].pR > 0) /* 有右子 */
         i = th[i].pR;
   return(ABS(i));
}
```



- 例如已知i=1，求结点10的中序前件k’：

  th[1].pL(=2) > 0，有左子，令i=2；

  th[2].pR(=3) > 0，有右子，令i=3；

  th[3].pR(=5) > 0，有右子，令i=5；

  th[5].pR(=-1) < 0，无右子，得αk’=5，

  则δk’ = 80(即th[5].num)。

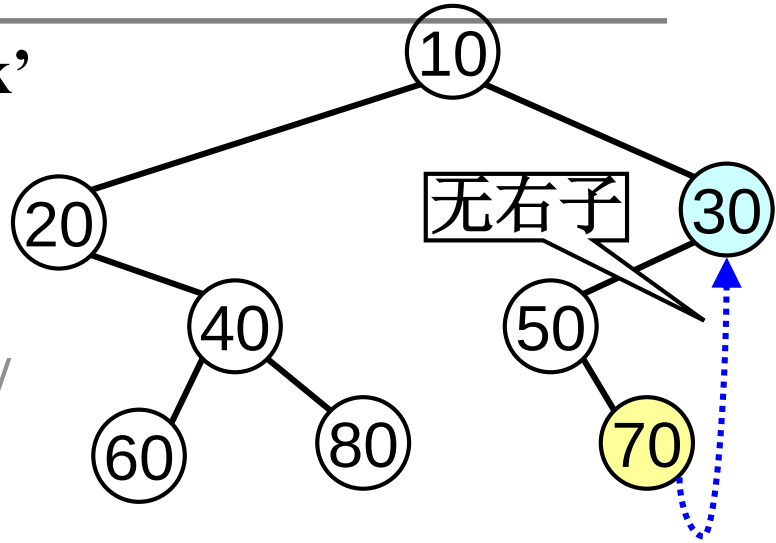
  函数调用为：

```
k= getPre(th, 1);
```

αk	δk	pL	pR
1	10	2	6
2	20	0	3
3	40	4	5
4	60	-2	-3
5	80	-3	-1
6	30	7	0
7	50	-1	8
8	70	-7	-6

- 根据结点k的地址i，求其中序后件k’

```
short getSuc(TH *th, short i)
  {
    if(i<1 || i>n)       /* 非法地址 */
      error();
    if ((i=th[i].pR)>0)    /* 有右子 */
      while(th[i].pL>0) /* 有左子 */
        i = th[i].pL;
    return(ABS(i));
  }
```



αk	δk	pL	pR
1	10	2	6
2	20	0	3
3	40	4	5
4	60	-2	-3
5	80	-3	-1
6	**30**	7	0
7	50	-1	8
8	70	-7	**-6**

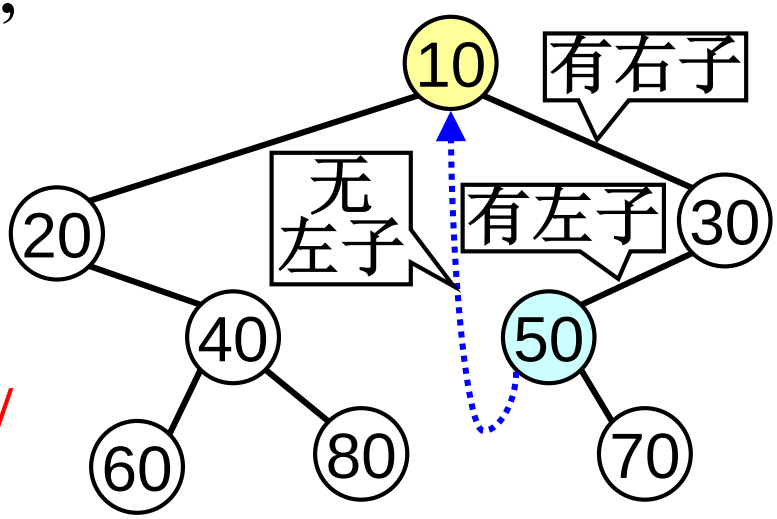
- 例如已知i=8，求结点70的中序后件k’：
  th[8].pR(=-6) < 0，无右子，得
  αk’ = |-6| = 6，
  则δk’ = 30(即th[6].num)。

函数调用为：

```
if(k= getSuc(th, 8))
  printf("suc node of %d is %d\n",
          th[i].num, th[k].num);
else
  printf("no suc node for %d.\n", th[i].num);
```

- 根据结点k的地址i，求其中序后件k’

```
short getSuc(TH *th, short i)
{
   if(i<1 || i>n)        /* 非法地址 */
      error();
   if ((i=th[i].pR)>0)     /* 有右子 */
      while(th[i].pL>0) /* 有左子 */
         i = th[i].pL;
   return(ABS(i));
}
```



- 例如已知i=1，求结点10的中序后件k’：

  th[1].pR(=6) > 0，有右子，令i=6；

  th[6].pL(=7) > 0，有左子，令i=7；

  th[7].pL(=-1) < 0，无左子，得αk’=7，

  则δk’ = 50 (即th[7].num)。

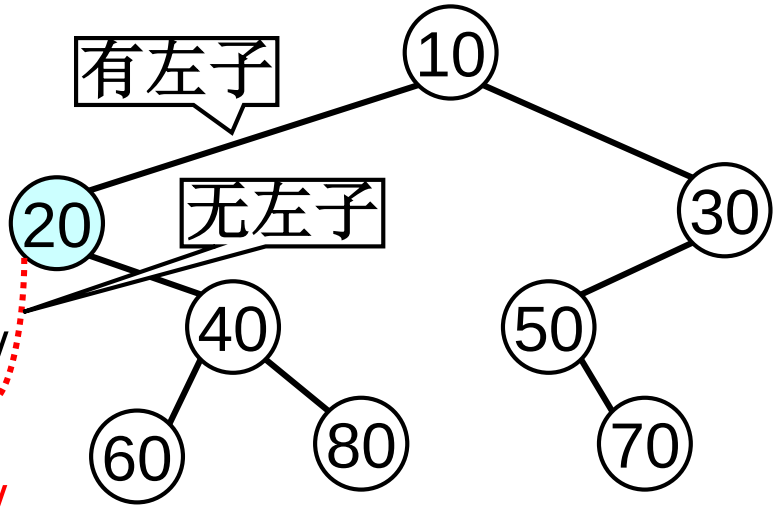
  函数调用为：

```
k= getSuc(th, 1);
```

αk	δk	pL	pR
1	10	2	**6**
2	20	0	3
3	40	4	5
4	60	-2	-3
5	80	-3	-1
6	30	**7**	0
7	**50**	**-1**	8
8	70	-7	-6

- 求中序的始结点指针αk_{first}

```
short getFirst( TH * th)
{
   short i;
   if((i=th[0].num) == 0)  /* 空树 */
      return(i);
   while(th[i].pL != 0 ) /* 有左子 */
      i = ABS(th[i].pL);
   return(i);
}
```



αk	δk	pL	pR
1	10	2	6
2	20	0	3
3	40	4	5
4	60	-2	-3
5	80	-3	-1
6	30	7	0
7	50	-1	8
8	70	-7	-6

- 例如，总是从i=th[0].num=1开始，
  th[1].pL(=2) != 0，有左子，令i=2，
  th[2].pL == 0，无左子，得αk'=2，
  则δk' = 20 (即th[2].num)。
  函数调用为：

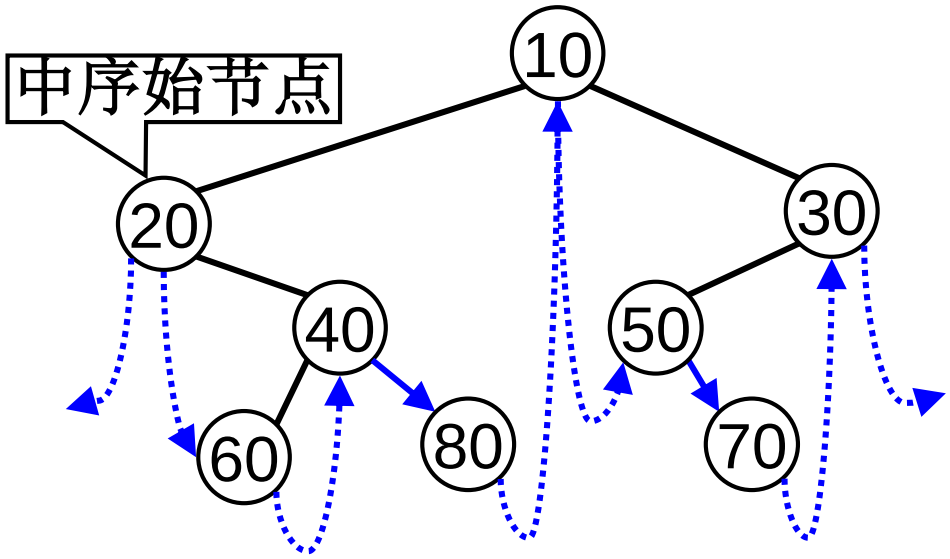
```
k= getFirst(th);
printf("first node for thread tree");
printf(" is %d.\n", th[k].num);
```

- 中序遍历

```
void  symOrder(TH * th)
{
   short p;
   /* 求中序始节点 */
   if((p=getFirst(th) == 0)
      error();       /* 空树  */
   do
      printf("node=%d\n", th[p].num);
   /* 求中序后件 */
   while(p = getSuc(th, p));
}
```

函数调用为：

```
symOrder(th);
```



αk	δk	pL	pR
1	10	2	6
2	20	0	3
3	40	4	5
4	60	-2	-3
5	80	-3	-1
6	30	7	0
7	50	-1	8
8	70	-7	-6

第四章 数据结构

- 4.1 基本概念
- 4.2 线性表
- 4.3 栈和队列
- 4.4 树
- 4.5 二叉树
- 4.6 二叉树的表示
  - 4.6.1 二叉树的层号表示
  - 4.6.2 二叉树的括号表示
  - 4.6.3 二叉树的确定

➢4.6.1 二叉树的层号表示

➢二叉树的层号及层号表示

- 图示二叉树的层号

  层号=1：A

  层号=2：B, G

  层号=3：E, F, K

  层号=4：H, J, L

- 前序遍历的层次表示

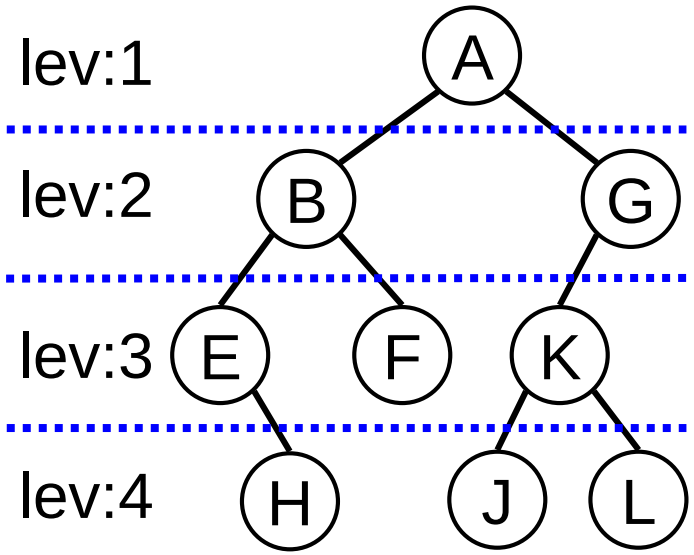
  1A, 2B, 3E, 4H, 3F, 2G, 3K, 4J, 4L

- 后序遍历的层次表示

  4H, 3E, 3F, 2B, 4J, 4L, 3K, 2G, 1A

- 中序遍历的层次表示

  3E, 4H, 2B, 3F, 1A, 4J, 3K, 4L, 2G



➢二叉树层号表示的唯一性问题

- 前序遍历的层号表示

  一个结点的前序遍历后件可为左子(如G)，也可为右子(如E)。

  所以，前序遍历的层号表示不能唯一地确定一棵二叉树。

- 后序遍历的层号表示

  一个结点的后序遍历前件可为左子(如G)，也可为右子(如E)。

  所以，后序遍历的层号表示也不能唯一地确定一棵二叉树。

- 中序遍历的层号表示

  一个结点如果有左子，左子一定是该结点的中序遍历前件，一个结点如果有右子，右子一定是该结点的中序遍历后件。

  所以，中序遍历的层次表示可以唯一地确定一棵二叉树。

- 前序遍历的层次表示

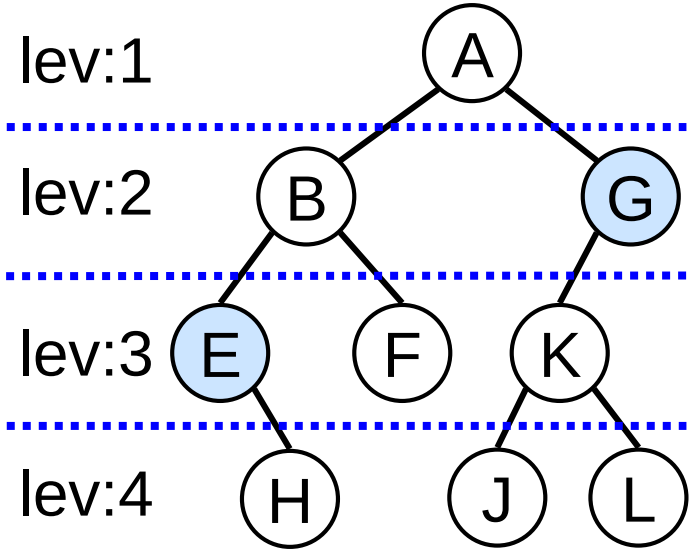
  1A, 2B, 3E, 4H, 3F, 2G, 3K, 4J, 4L

- 后序遍历的层次表示

  4H, 3E, 3F, 2B, 4J, 4L, 3K, 2G, 1A

- 中序遍历的层次表示

  3E, 4H, 2B, 3F, 1A, 4J, 3K, 4L, 2G



➢4.6.2 二叉树的括号表示

如果二叉树T只有一个结点，此结点就是它的括号表示。

如果二叉树由根结点R和左子树T_L和右子树T_R组成，则树T的括号表示就是：R(T_L的括号表示, T_R的括号表示)。

由于二叉树的左子树T_L和右子树T_R可能只出现任意一支，所以二叉树的括号表示可以是：

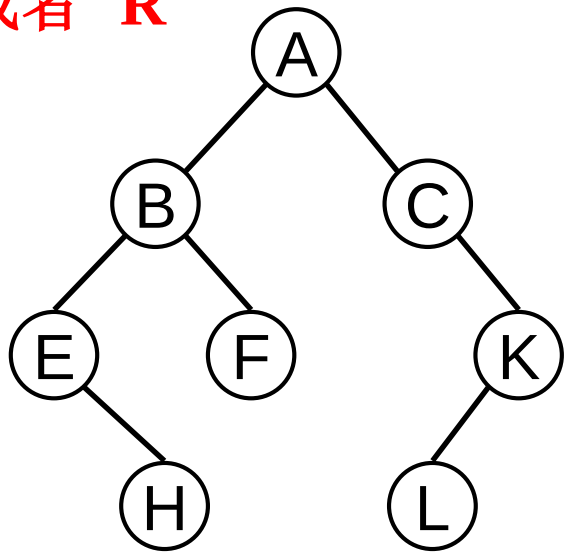
R(T_L的括号表示,) 或者 R(, T_R的括号表示) 或者

R(T_L的括号表示, T_R的括号表示) 或者 R

图示树的二叉树的括号表示

A(B(E(, H), F), C(, K(L,)))

根据二叉树的括号表示，可以唯一地确定一棵二叉树。

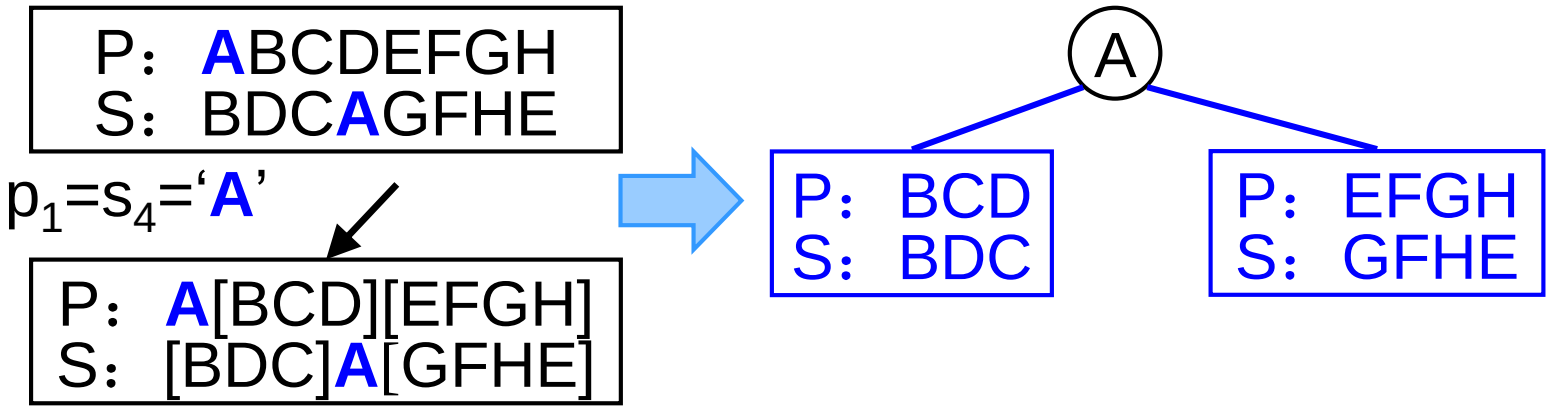


4.6.3 二叉树的确定

如果已知二叉树的前序和中序的遍历，可以唯一地确定二叉树。
如果已知二叉树的后序和中序的遍历，可以唯一地确定二叉树。

※确定二叉树的算法示例

前序和中序的字符序列分别记为$p_1,p_2, ..., p_n$和$s_1, s_2, ..., s_n$。
例如，已知：P(前序)：ABCDEFGH，S(中序)：BDCAGFHE。

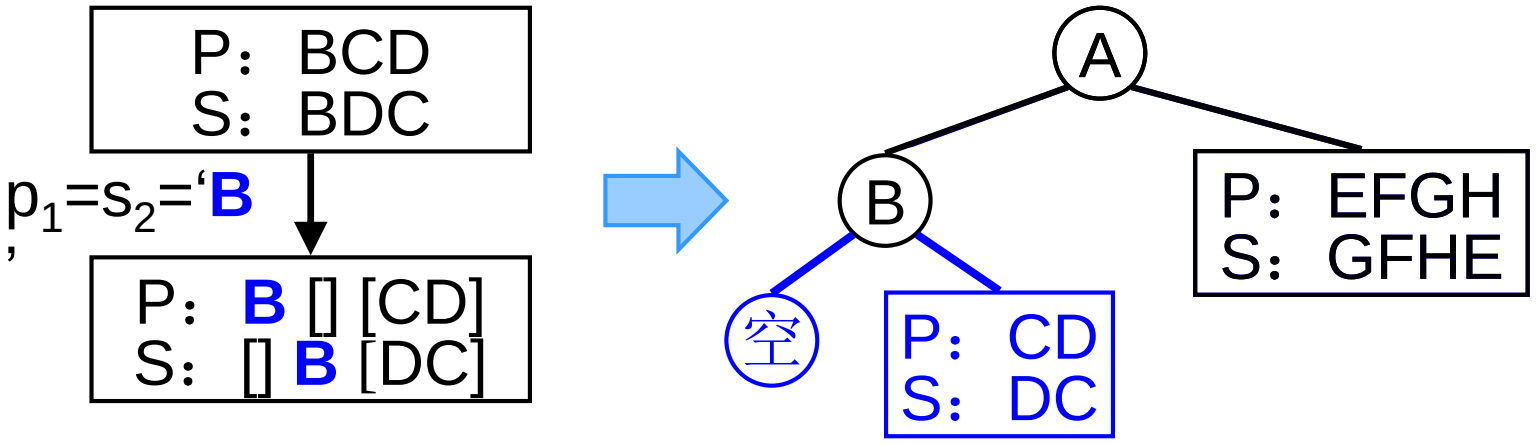


求得A为根。
由此划分左右子树：
　在P中p_1='A'，在S中s_4='A'。
分别获得左右子树的前序和中序：
　左子树的 前序：BCD
　　中序：BDC
　右子树的 前序：EFGH
　　中序：GFHE
需要分别求解左右子树。

※确定二叉树的算法示例

求解**A**的左子树。

P(前序)：BCD，S(中序)：BDC。



求得**B**为根，由此划分左右子树。

分别获得左右子树的前序和中序：

左子树的 前序：空

中序：空

右子树的 前序：**CD**

中序：**DC**

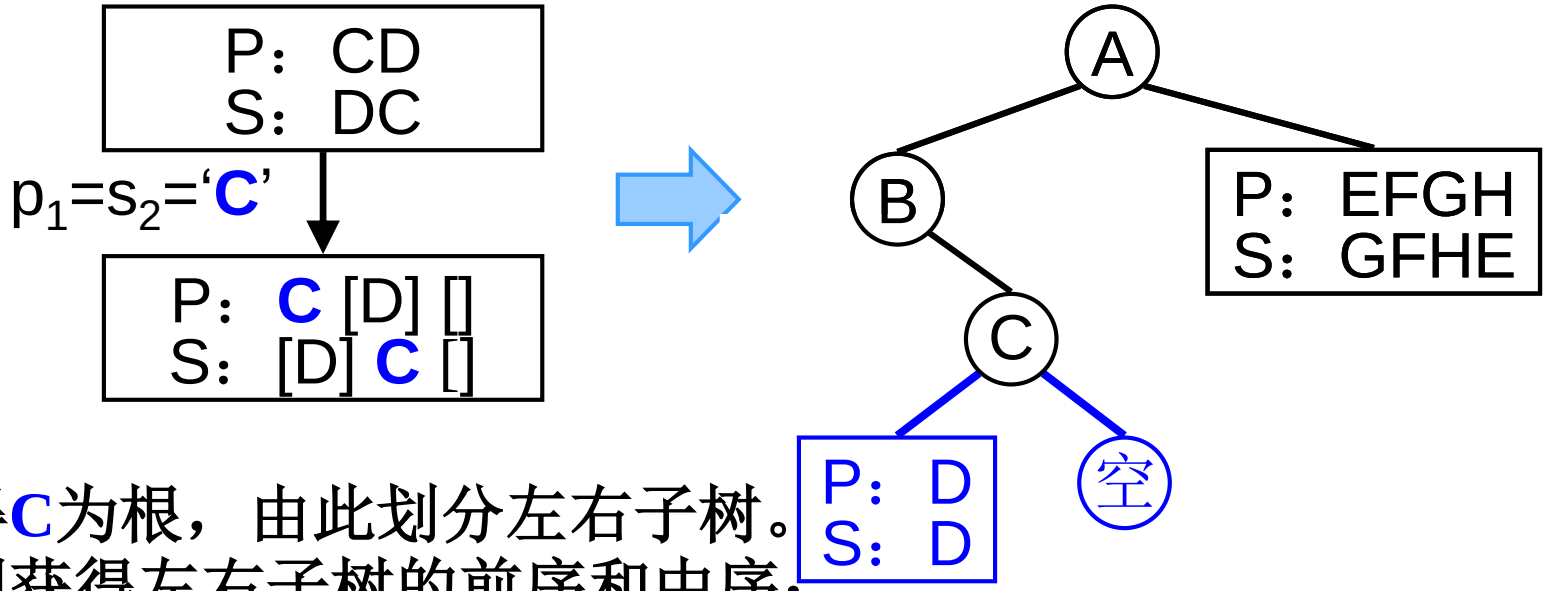
需要继续求解右子树。

左子树为空。

※确定二叉树的算法示例

求解**B**的右子树。

P(前序)：CD，S(中序)：DC。



求得**C**为根，由此划分左右子树。

分别获得左右子树的前序和中序：

左子树的 前序：**D**

中序：**D**

右子树的 前序：空

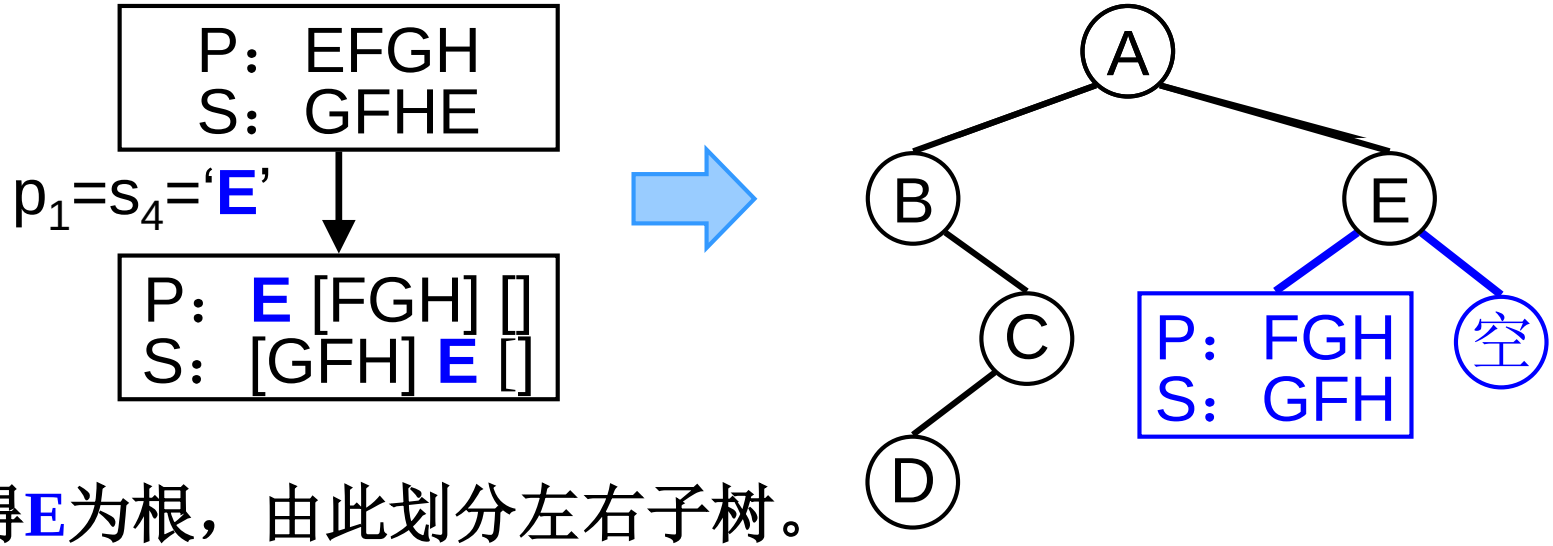
中序：空

左子树只有一个结点**D**，右子树为空，递归返回。

※确定二叉树的算法示例

求解**A**的右子树。

P(前序)：EFGH，S(中序)：GFHE。



求得**E**为根，由此划分左右子树。

分别获得左右子树的前序和中序：

左子树的 前序：**FGH**

中序：**GFH**

右子树的 前序：空

中序：空

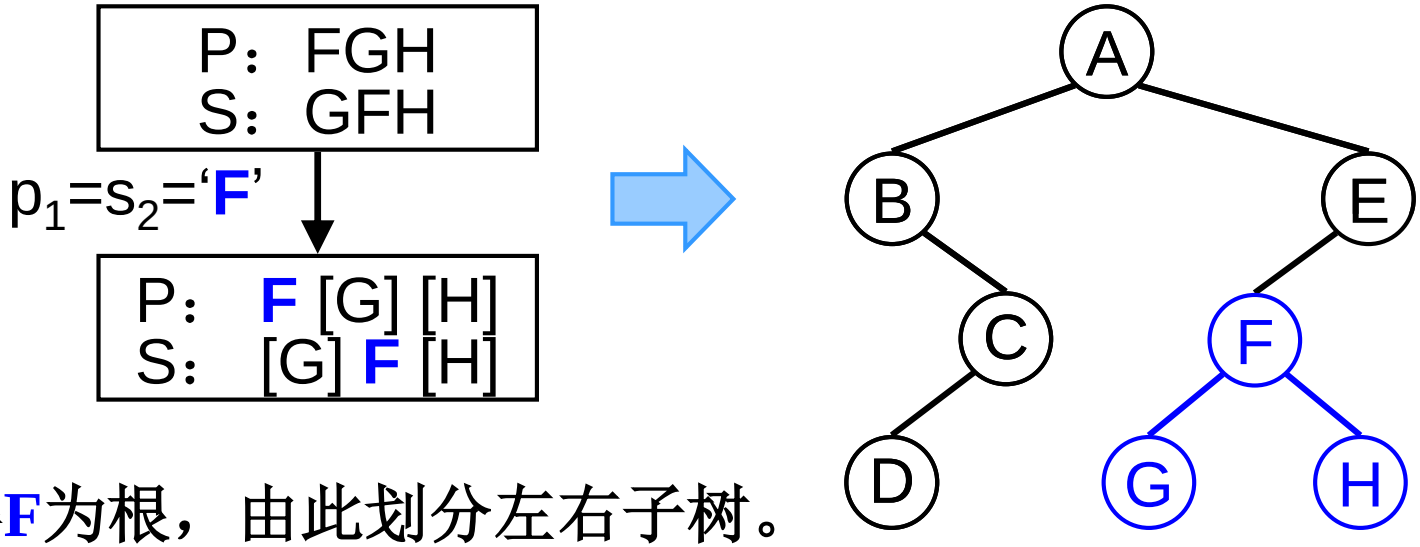
需要继续求解左子树。

右子树为空。

※确定二叉树的算法示例

求解**E**的左子树。

P(前序)：FGH，S(中序)：GFH。



求得**F**为根，由此划分左右子树。

分别获得左右子树的前序和中序：

左子树的 前序：**G**

中序：**G**

右子树的 前序：**H**

中序：**H**

左子树只有一个结点**G**，右子树只有一个结点**H**，递归返回。

※确定二叉树的算法示例

已知前序和中序的字符序列分别为$p_1,p_2, ..., p_n$和$s_1, s_2, ..., s_n$。

例如，前序：ABCDEFGH，中序：BDCAGFHE。

1) 由于根是前序的首结点，可确定根为p_1，同时可在中序中找到$s_R=p_1$。

如：A是根。在前序中p_1=‘A’，在中序中s_4=‘A’。

2) 在中序中划分左子树和右子树。根据中序，根的前面是左子树，后面是右子树。可划分为[S左], 根(=s_R), [S右]，其中：

S左=$s_1,...,s_{R-1}$, S右=$s_{R+1},...,s_n$, 分别表示左右子树的结点集合。

如：[BDC] A [GFHE]

3) 根据中序划分，可划分前序为：根(=p_1) [P左] [P右]，其中：

P左=$p_2,...,p_R$，P右=$p_{R+1},...,p_n$，S左与P左的结点相同，S右与P右的结点相同，但顺序可以不同。

如：A [BCD] [EFGH]

4) 从而分别获得左右子树的前序和中序。

如，左子树的前序：BCD，　中序：BDC

而，右子树的前序：EFGH，中序：GFHE

5) 由此递归处理，直到子树只有一个结点或为空，递归结束。

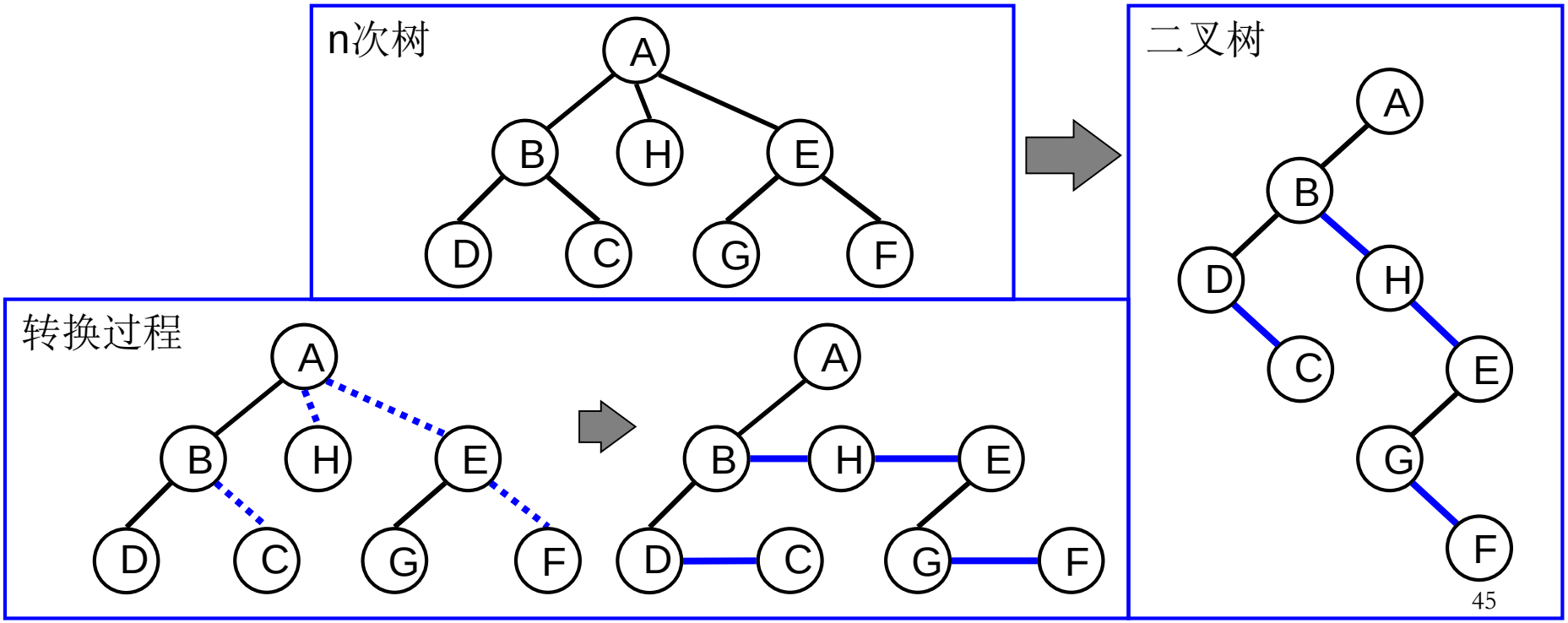
第四章 数据结构

- 4.1 基本概念
- 4.2 线性表
- 4.3 栈和队列
- 4.4 树
- 4.5 二叉树
- 4.6 二叉树的表示
- 4.7 任意次树与二叉树之间的转换

4.7 任意次树与二叉树之间的转换

- 用二叉树表示n次树

一棵任意次树(n次树)，可将其转化为二叉树。用二叉树来表示和存储n次树，有许多方便的地方。例如，可以对不同的n次树统一存储形式，而且很多算法都是针对二叉树的特点研究开发的。

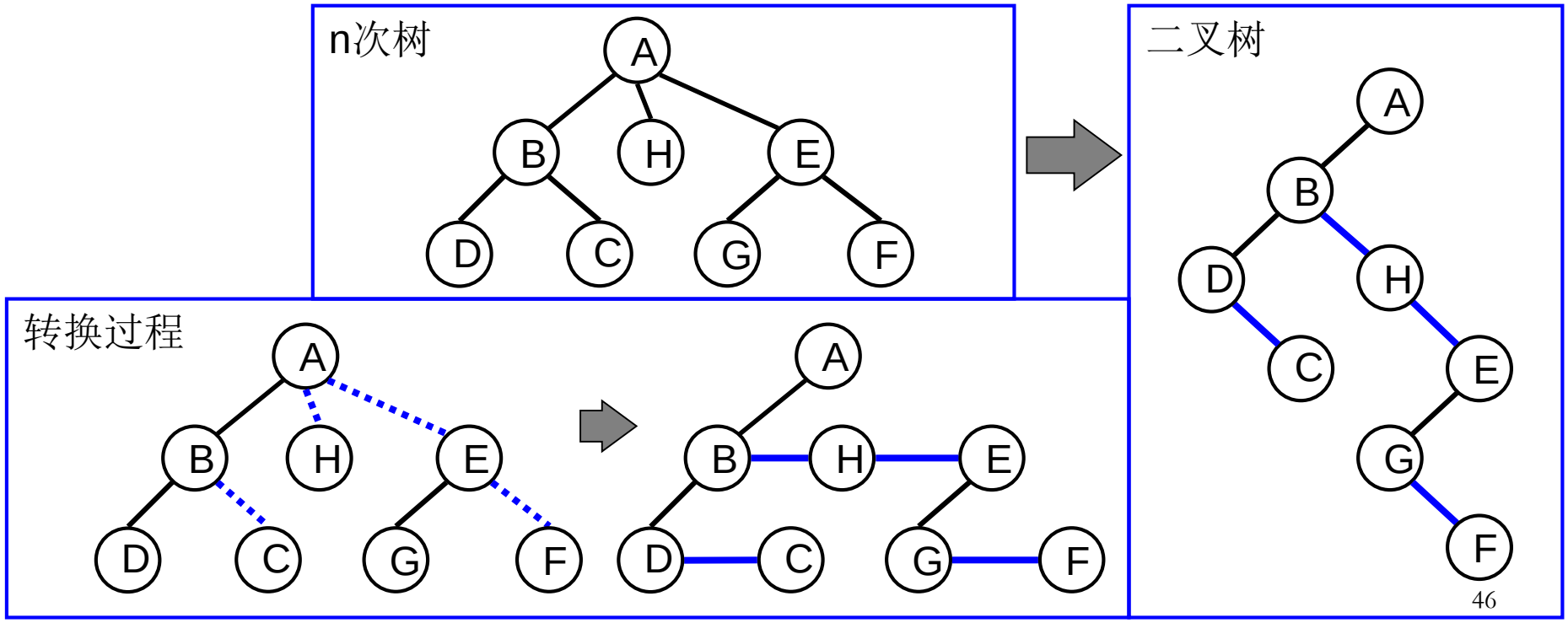


4.7 任意次树与二叉树之间的转换

- **n次树到二叉树的图形转换方法**

⊙对任一结点，保持对第一子结点的指向不变，把对其他子结点的指向改为从第一子结点起同辈中前一结点出发的水平指向

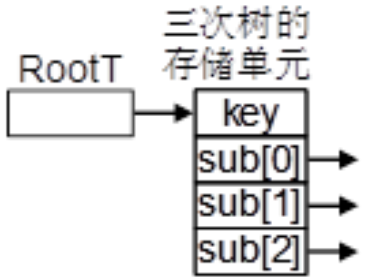
⊙将所有的水平指向顺转45度，就可得到转换后的二叉树。

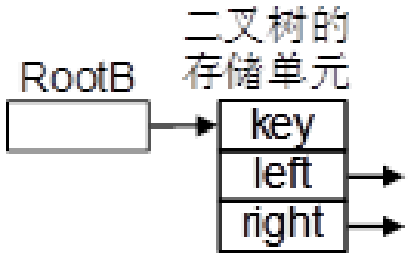


4.7 任意次树与二叉树之间的转换

【例4-7.1】三次树转换为二叉树

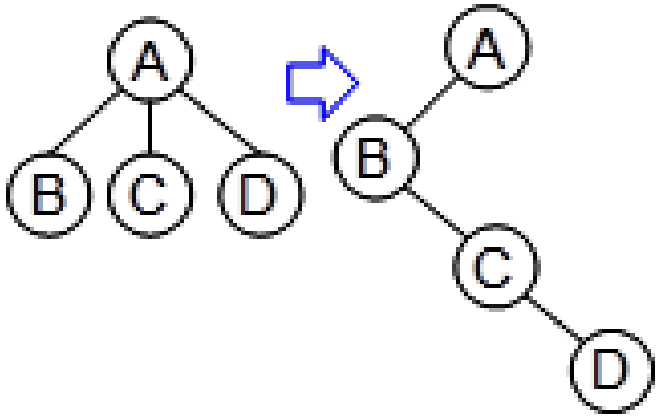
```
#define  TRINODE struct  trinode
#define  BINODE  struct  binode
TRINODE
{
        char   key;
        TRINODE  *sub[3];
};
BINODE
{
        char        key;
        BINODE *Left, *right;
};
TRINODE *RootT=NULL; /* 三次树的根指针*/
BINODE *RootB=NULL;   /* 二叉树的根指针*/
```





4.7 任意次树与二叉树之间的转换

【例4-7.1】三次树转换为二叉树




```
BINODE *T2B(TRINODE *t)
{
   BINODE  *b;
   if(t==NULL)  /* 空结点，返回*/
      return(NULL);
   if( (b=popBFree()) == NULL )
      error();
   b->key = t->key; /* 复制结点值*/
   b->Left = b->right = NULL; /* 左右子置空*/
   if(t->sub[0])
   {
      b->Left = T2B(t->sub[0]); /* 转换第一子树*/
      if(t->sub[1])
      {
         b->Left->right = T2B(t->sub[1]);          /* 转换第二子树    */
         if(t->sub[2])
            b->Left->right->right = T2B(t->sub[2]);          /* 转换第三子树    */
      }
   }
   return(b);
}
```

➢作　　业

- 习题　　4-23(穿线树)，
  4-30(层号表示)
- 上机调试和运行 ：二叉树生成和遍历的程序实现
  【例4-5.1】exe4-5.2BiTree.c
  【例4-5.2】exe4-5.1BiTree.c