第六章 树和二叉树

6.1 树的定义和基本术语

6.2 二叉树

6.3 遍历二叉树和线索二叉树

6.4 树和森林

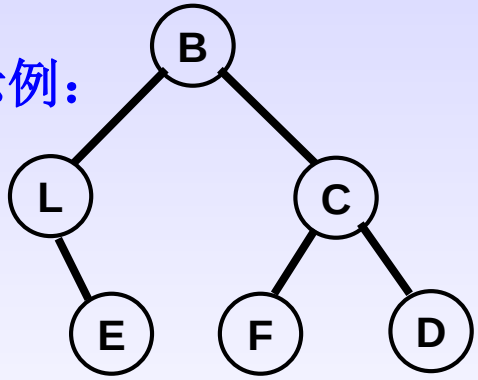
6.5 赫夫曼树及其与树的应用

6.2 二叉树

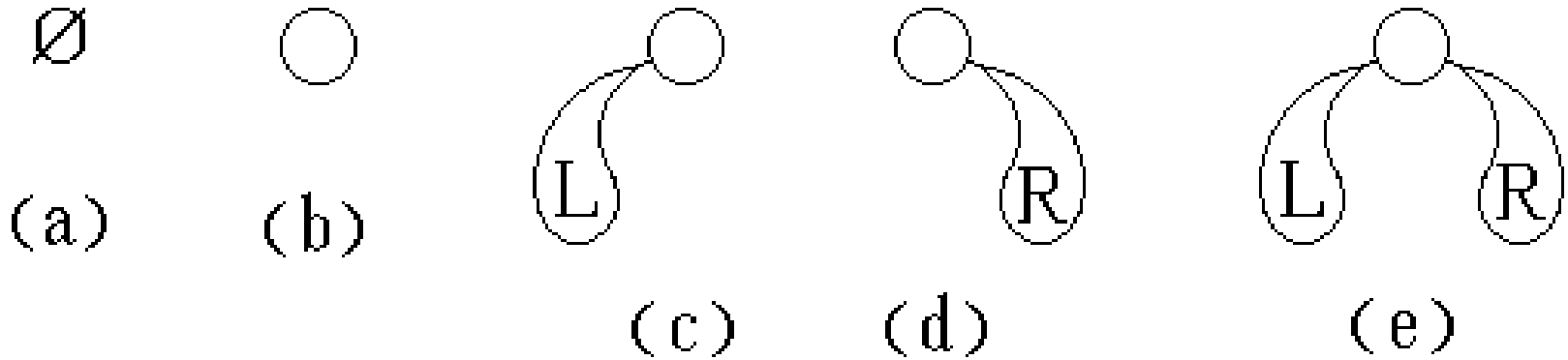
6.2.1 二叉树的定义

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、右子树的，互不相交的二叉树组成。

二叉树示例：



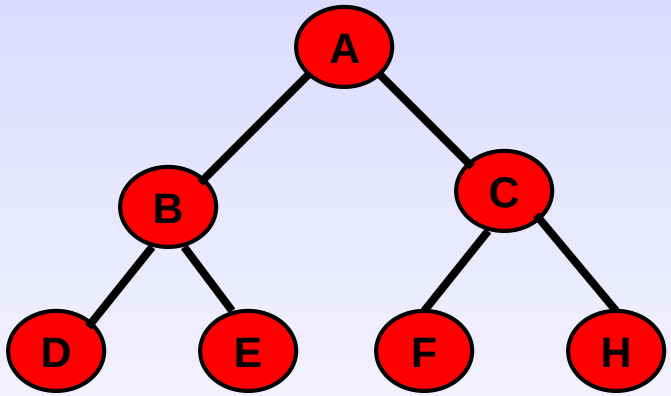
二叉树结点的子树要区分左子树和右子树，即使只有一棵子树也要进行区分，说明它是左子树，还是右子树。这是二叉树与树的最主要的差别。



6.2 二叉树

6.2.2 二叉树的性质

性质1：在二叉树的第i层上至多有2^{i-1}个结点。



1层：结点个数 $2^{1-1}=2^0$ 个

2层：结点个数 $2^{2-1}=2^1$ 个

3层：结点个数 $2^{3-1}=2^2$ 个

证：　1）　k = 1 时成立；

2）设 k = i-1 时成立；

3）则 k = i 时，第i层上至多有$2*2^{i-2} = 2^{i-1}$ 个结点；

∴　原命题成立。

6.2 二叉树

性质2：深度为K的二叉树至多有$2^k - 1$个结点。

证：利用性质1，将第1层至第k层的最多的结点数进行相加，

则：$1 + 2 + 2^2 + \cdots\cdots + 2^{k-2} + 2^{k-1} = 2^k - 1$

∴ 原命题成立。

性质3：设二叉树的叶子结点数为n0，度为2的结点数为n2，则有n0=n2+1。

证：设度为1的结点数为n1，树枝的总数为B。

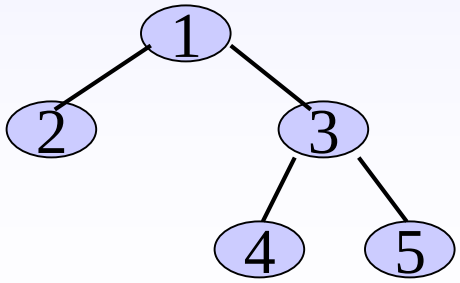
则：B = n0+n1+n2-1

又 B = n1+2*n2

∴ n1+2*n2 = n0+n1+n2-1

∴ n0 = n2 + 1

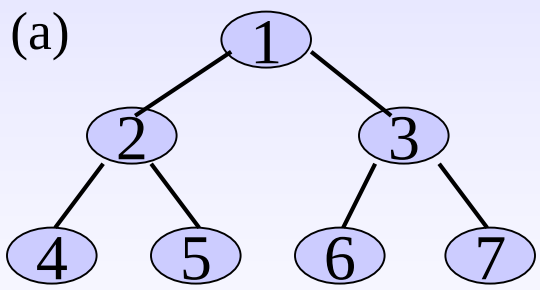
∴ 原命题成立。

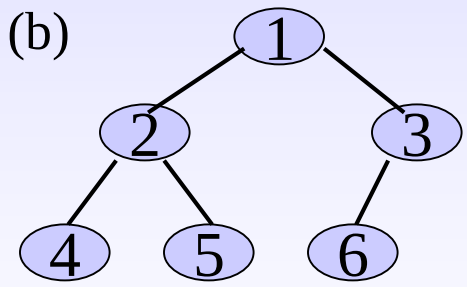


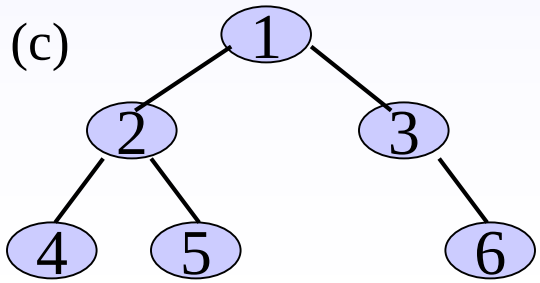
6.2 二叉树

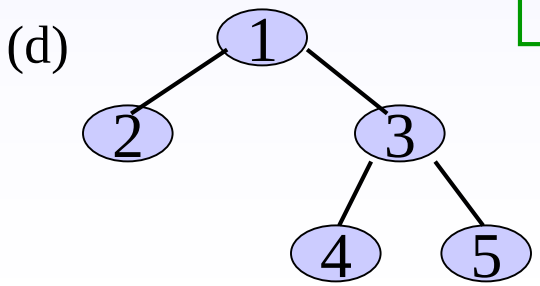
满二叉树：深度为k且有2^k-1个结点的二叉树，如下图(a)所示。

完全二叉树：如果深度为k、有n个结点的二叉树中每个结点能够与深度为k的顺序编号的满二叉树从1到n标号的结点相对应，则称这样的二叉树为完全二叉树，如下图(b)所示。图(c)、(d)是2棵非完全二叉树。满二叉树是完全二叉树的特例。









完全二叉树的所有叶子结点都出现在第k层或k-1层。

6.2 二叉树

性质4：具有n个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor + 1$。

符号$\lfloor x \rfloor$表示不大于x的最大整数。

证：设此二叉树的深度为k，则根据性质2及完全二叉树的定义

得到：$2^{k-1} - 1 < n <= 2^k - 1$ 或 $2^{k-1} <= n < 2^k$

取对数得到：$k - 1 <= \log_2 n < k$

∵ k是整数， ∴$k = \lfloor \log_2 n \rfloor + 1$。

性质5：如果对一棵有n个结点的完全二叉树的结点按层序编号（从上到下，从左到右),则对任一结点i（1<=i<=n),有：

（1）如果i＝1，则结点i无双亲，是二叉树的根；如果i>1，则其双亲是结点$\lfloor i/2 \rfloor$。

（2）如果2i>n，则结点i无左孩子；否则其左孩子是结点2i。

（3）如果2i＋1>n，则结点i无右孩子；否则其右孩子是结点2i＋1。

6.2 二叉树

可以先证明（2）和（3），然后从（2）和（3）推出（1）。

证：1、对于i＝1，由完全二叉树的定义，

其左孩子是结点2，若2>n，即不存在结点2，此时结点i无孩子。

其右孩子也只能是结点3，若结点3不存在，即3>n，此时结点i无右孩子。

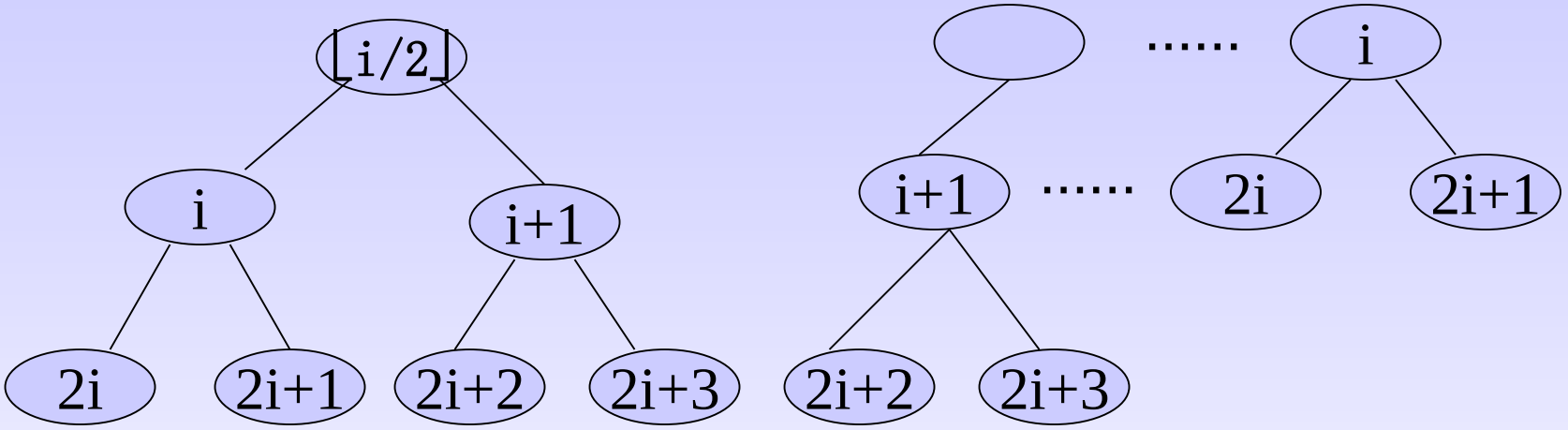
2、对于i>1，假设第j层上的某个结点编号为i(2^{j-1}<=i<2^{j}-1)，且2i＋1<n，其左孩子为2i，右孩子为2i＋1；

则编号为i＋1的结点是编号为i的结点的右兄弟或堂兄弟(同层)；

若它有左孩子，则其编号必定为2i＋2＝2*(i＋1)；

若它有右孩子，则其编号必定为2i＋3＝2*(i＋1)＋1。

6.2 二叉树



(a) i和i+1结点在同一层　　(b) i和i+1结点不在同一层

3、从（2）和（3）推出（1）

当i=1时，结点是根，因此无双亲。当i>1时，设其双亲结点编号为p，如果i为左孩子，即i=2p，则p=i/2=⌊i/2⌋；如果i为右孩子，i=2p+1, p=(i-1)/2=⌊i/2⌋ 。

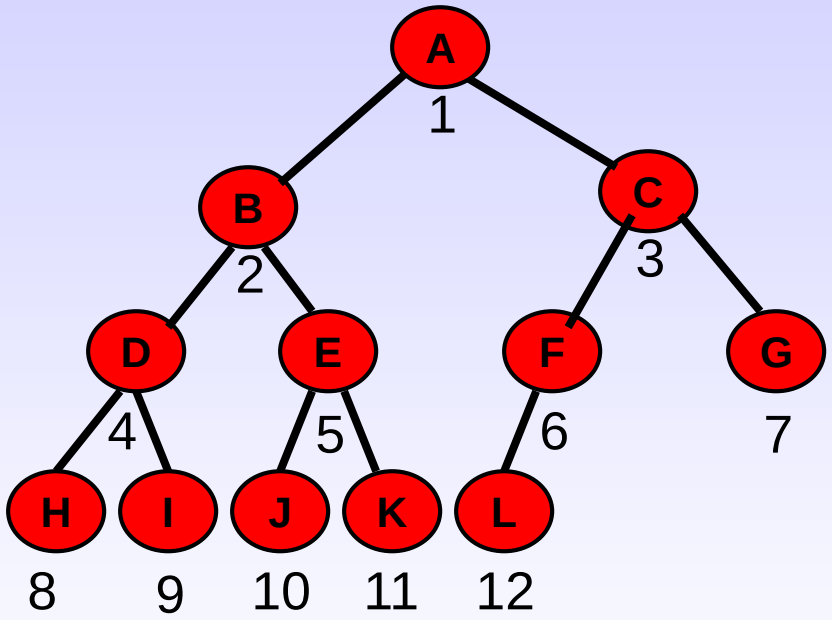
证毕。

6.2 二叉树

6.2.3 二叉树的存储结构

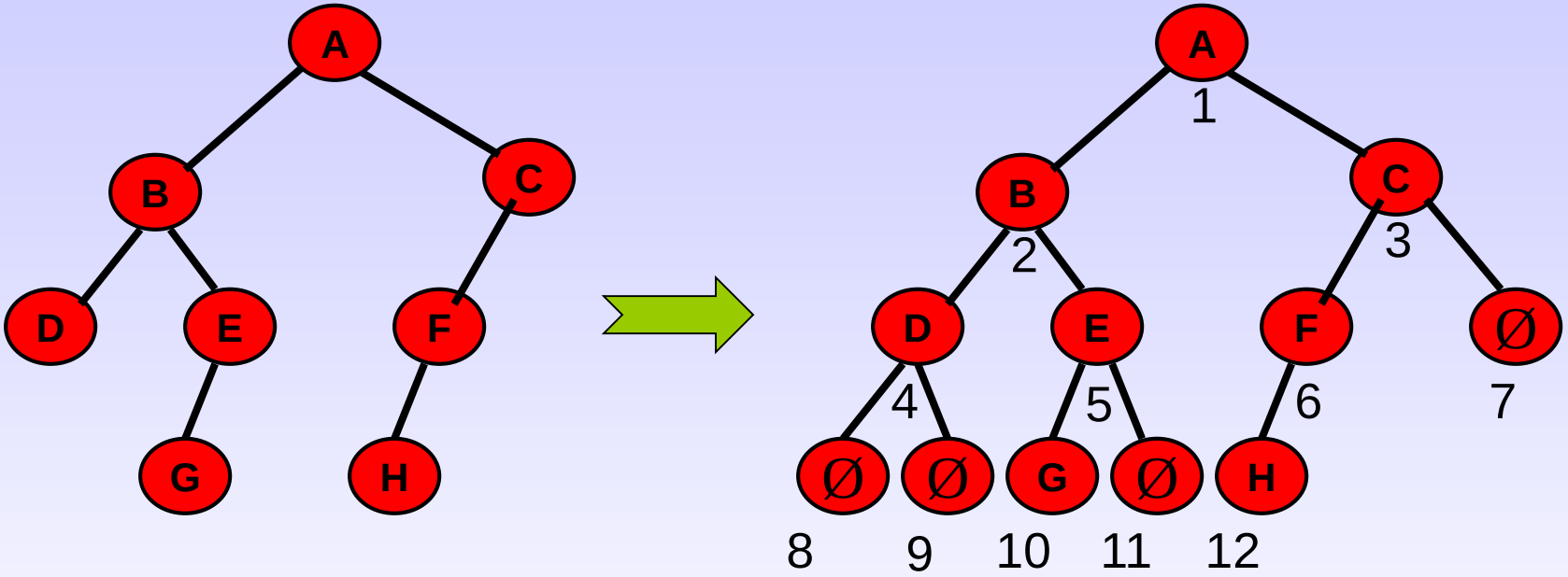
1. 顺序存储结构

用一组连续的存储单元存储二叉树的数据元素。必须把二叉树的所有结点按层序编号，结点的序号反映出结点之间的逻辑关系。



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

6.2 二叉树



A	B	C	D	E	F	Ø	Ø	Ø	G	Ø	H
1	2	3	4	5	6	7	8	9	10	11	12

顺序存储缺点是有可能对存储空间造成浪费，在最坏的情况下，一个深度为h且只有h个结点的右单支树却需要2^h-1个结点存储空间。

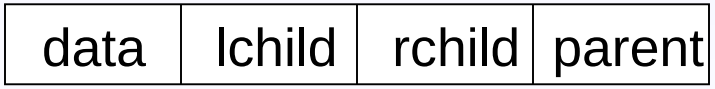
6.2 二叉树

2. 链式存储结构：

二叉链表

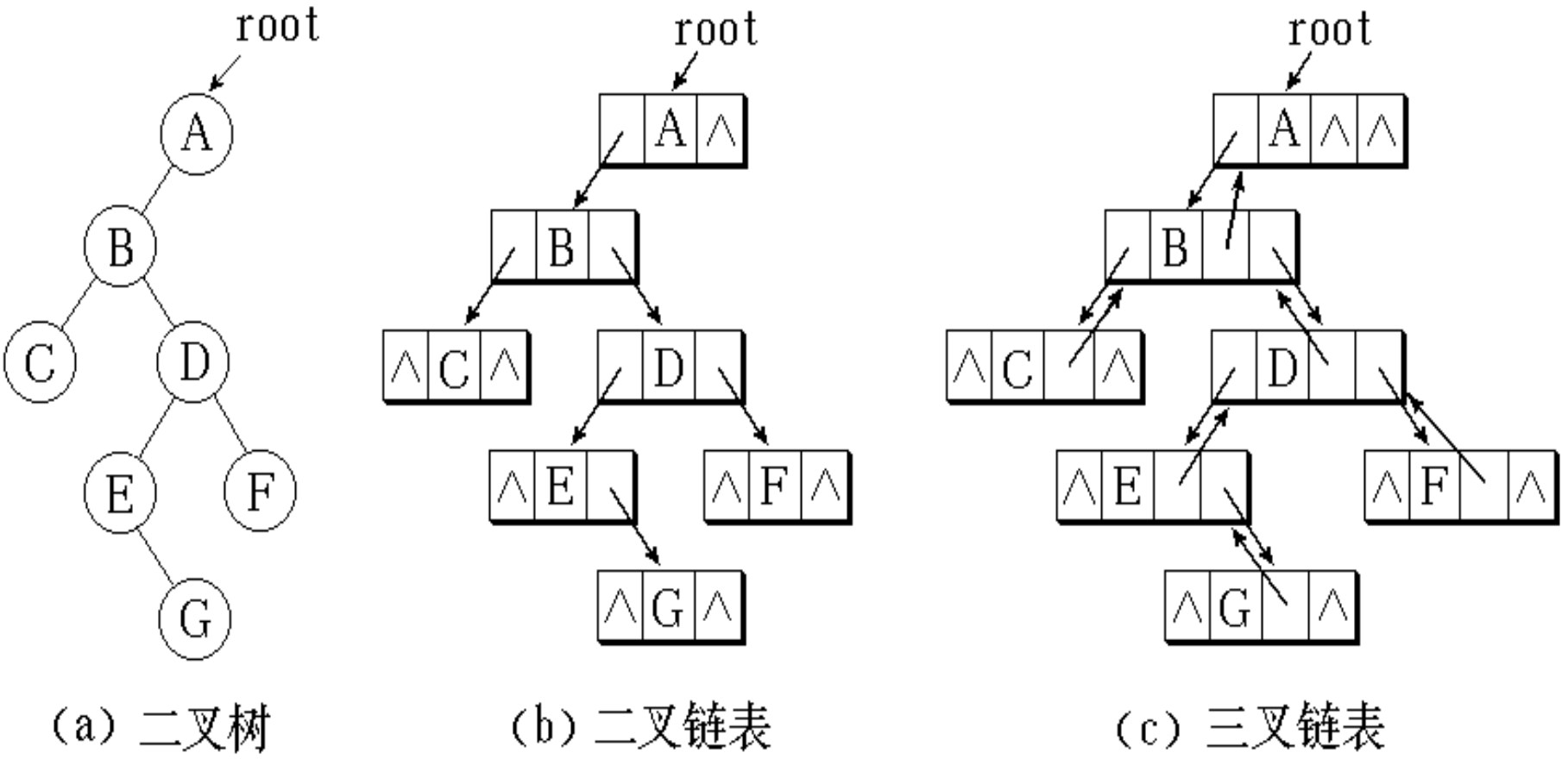

```
typedef struct BiTNode
{  TELemType       data;
   struct BiTNode  *lchild;
   struct BiTNode  *rchild;
} BiTNode, * BiTree;
```

三叉链表




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

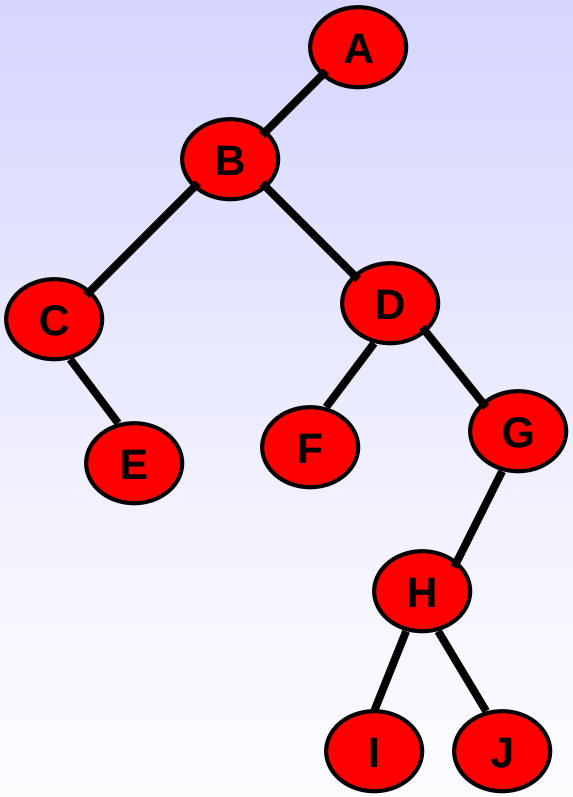
6.2 二叉树



(a) 二叉树　　(b) 二叉链表　　(c) 三叉链表

6.2 二叉树

3 静态链式存储结构：




```
#define MAX_SIZE  10
typedef struct BiTNode
{  TELemType  data;
   int            lchild;
   int            rchild;
} BiTNode;
typedef  BiTNode SqBiTree[MAX_SIZE];
```

	1	2	3	4	5	6	7	8	9	10
data	A	B	C	D	E	F	G	H	I	J
lchild	2	3	0	6	0	0	8	9	0	0
rchild	0	4	5	7	0	0	0	10	0	0

静态链式结构适用于二叉树上的结点个数已知，或不支持动态存储分配的高级语言。

6.3 遍历二叉树和线索二叉树

6.3.1 遍历二叉树

1 定义：按照某种顺序访问二叉树中的每个结点，使每个结点被访问一次且只被访问一次。

- 线性表的遍历

（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：非线性关系，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）。

6.3 遍历二叉树和线索二叉树

–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;

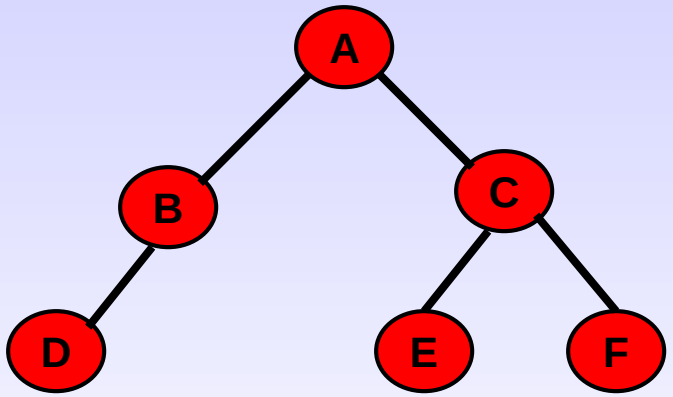
(3)中序遍历右子树;

–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;

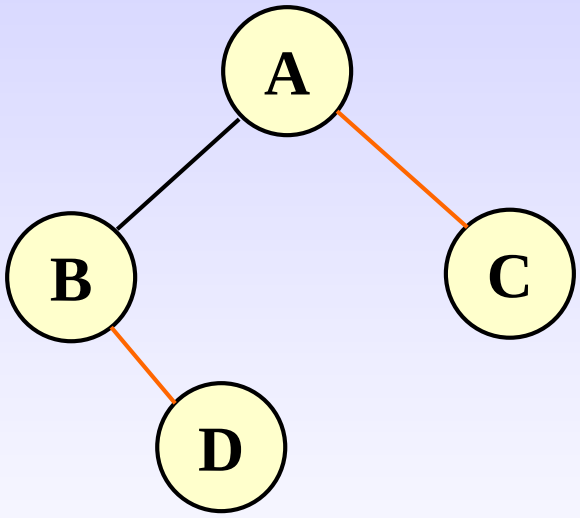
(3)访问根结点;

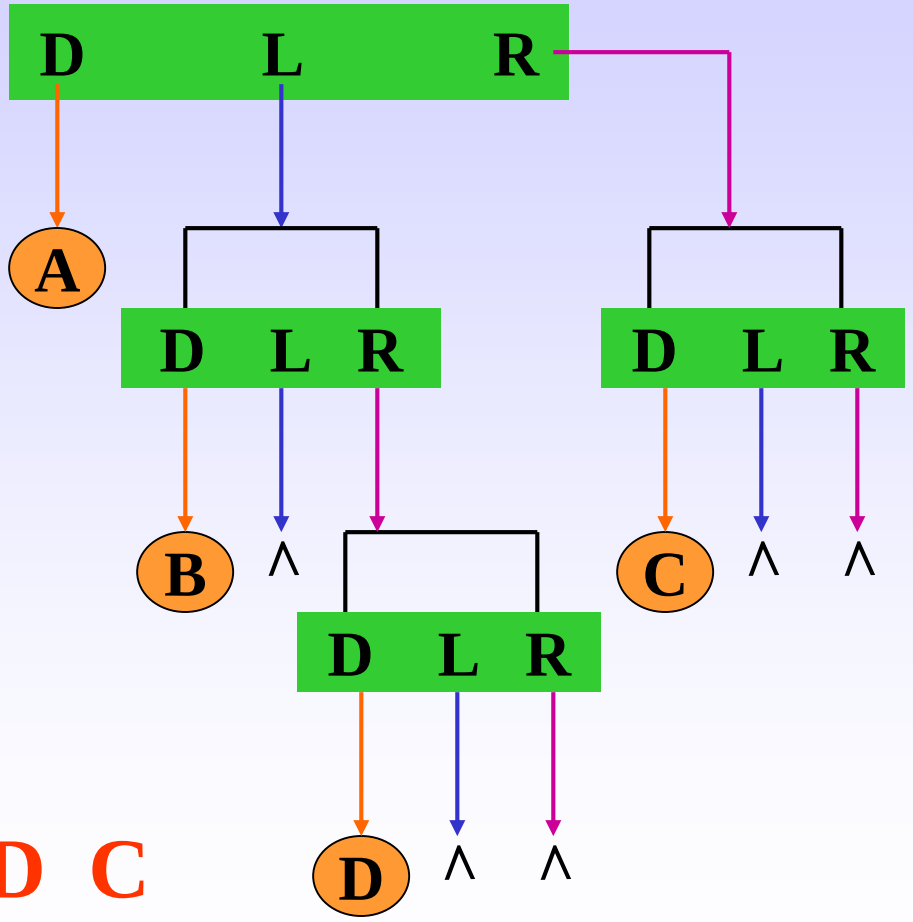


先序：A B D C E F
中序：D B A E C F
后序：D B E F C A

6.3 遍历二叉树和线索二叉树

先序遍历：

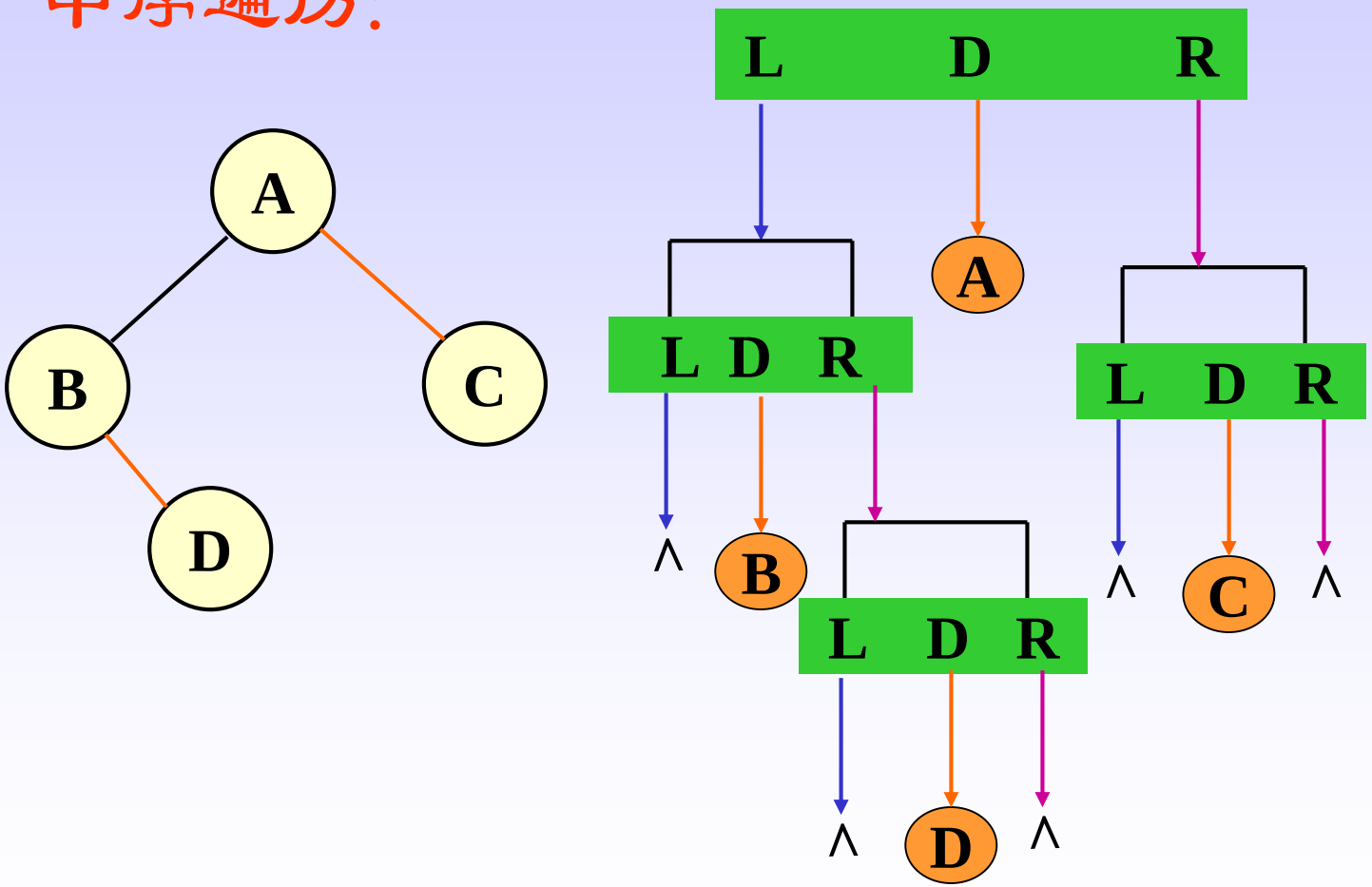




先序遍历序列：A B D C

6.3 遍历二叉树和线索二叉树

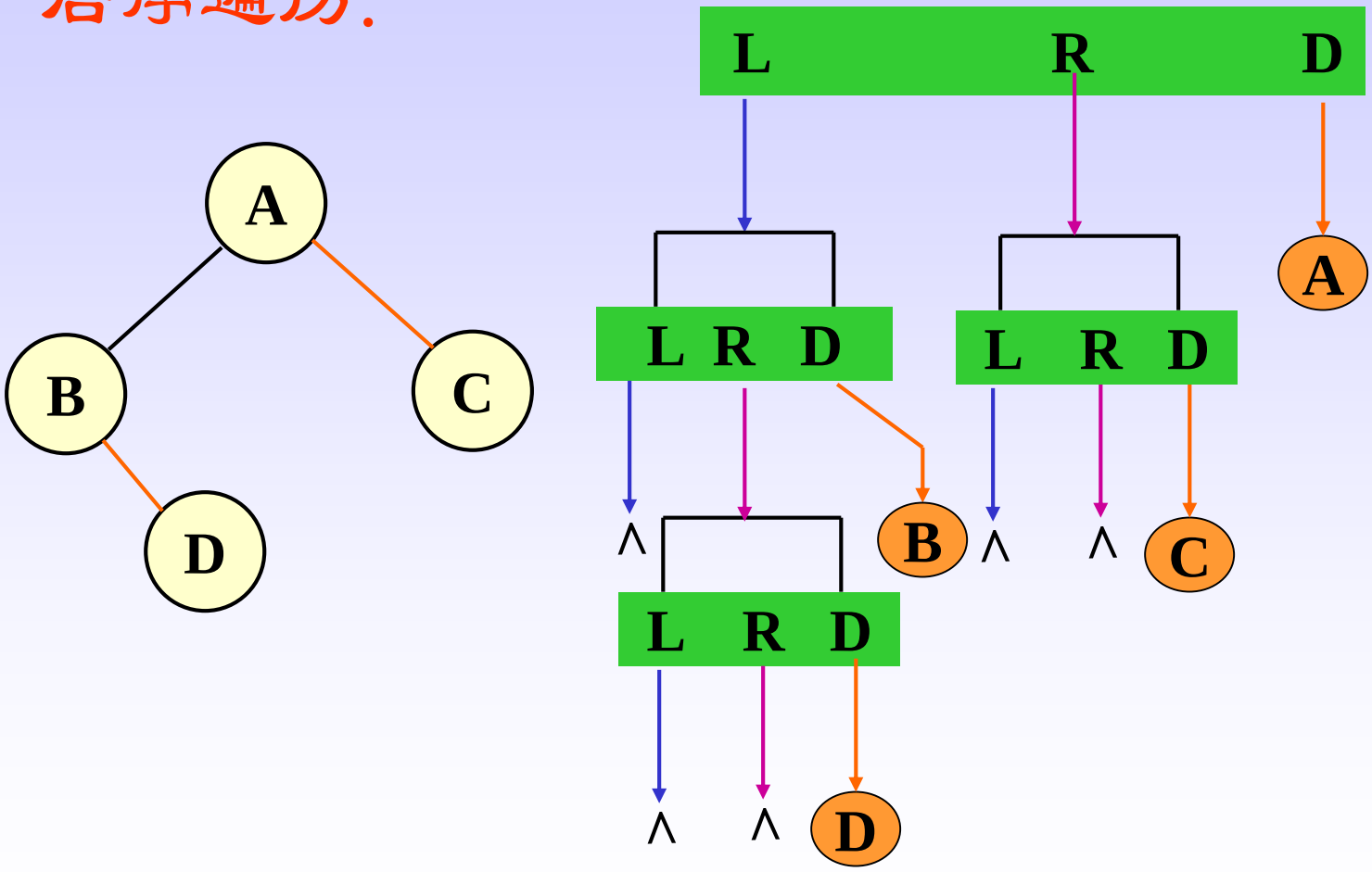
中序遍历：



中序遍历序列：B D A C

6.3 遍历二叉树和线索二叉树

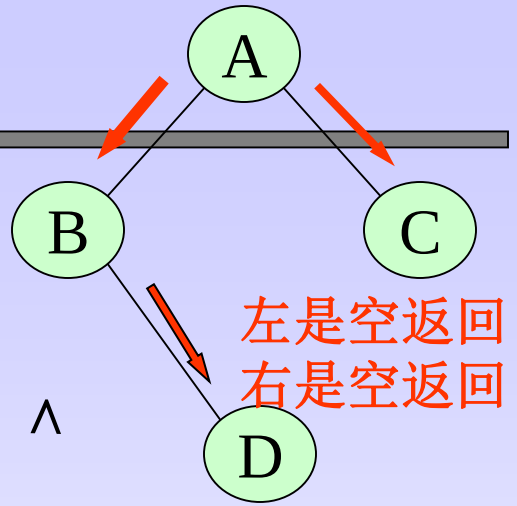
后序遍历:



后序遍历序列： D B C A

6.3 遍历二叉树和线索二叉树

```
void inorder(BiTree T)
{  if (T!=NULL)
   {  inorder(T->lchild);
      printf(T->data);
      inorder(T->rchild);
   }
}
```



左是空返回

左是空返回
右是空返回

左是空返回
右是空返回

主程序

iod(T)

T → A
iod(T→L);
printf(A);
iod(T→R);

T → B
iod(T→L);
printf(B);
iod(T→R);

T → C
iod(T→L);
printf(C);
iod(T→R);

T → D
iod(T→L);
printf(D);
iod(T→R);

T → ∧
返回

T → ∧
返回

T → ∧
返回

T → ∧
返回

中序序列：**B D A C**

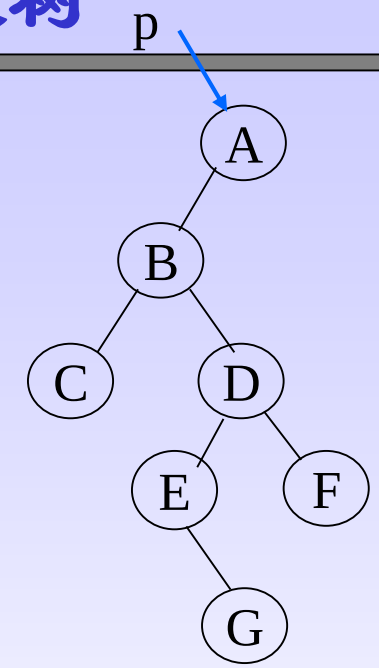
6.3 遍历二叉树和线索二叉树

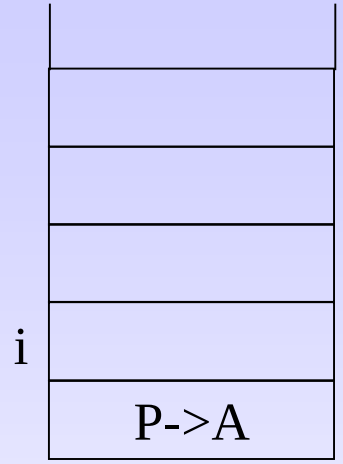
2 非递归算法：

```
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;
             }
     }
}
```

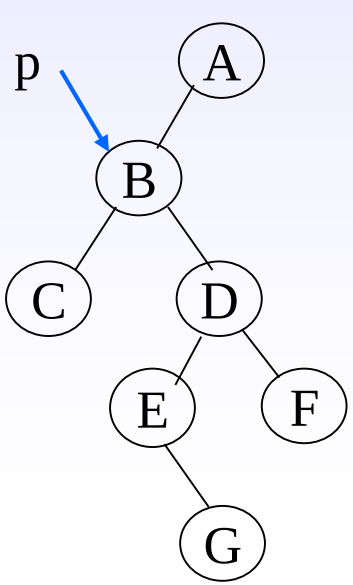
6.3 遍历二叉树和线索二叉树

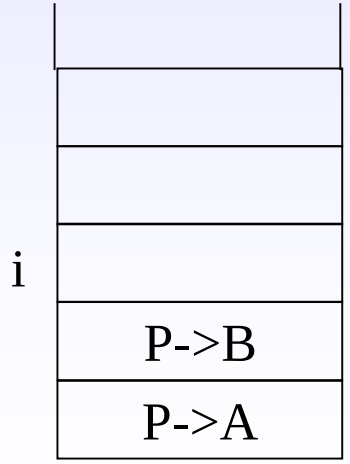
```
void inorder(BiTree T)
{  InitStack(S);
    p=T;
    while(p||!SatckEmpty(S))
    { if (p) {
        Push(p); p=p->lchild;
      }
      else{
        Pop(p);
        printf(p->data);
        p=p->rchild;
      }
    }
}
```





(1)

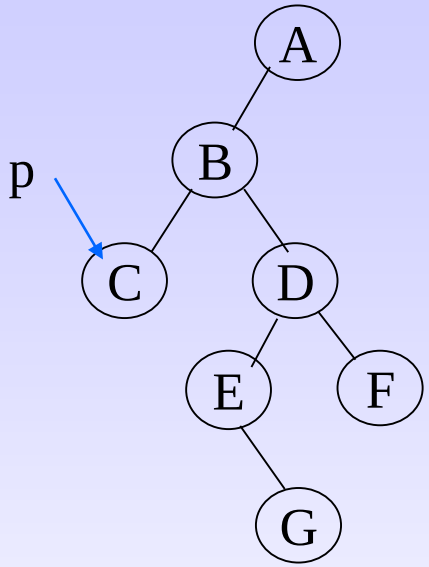


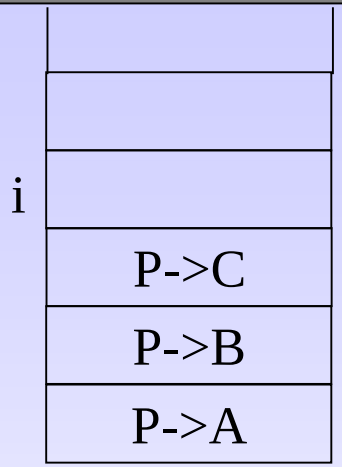


(2)

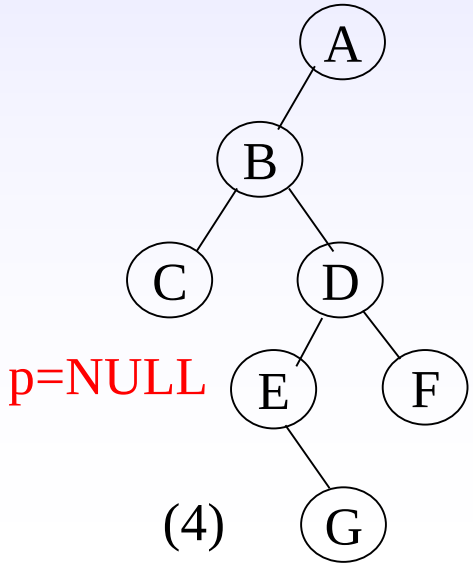
6.3 遍历二叉树和线索二叉树

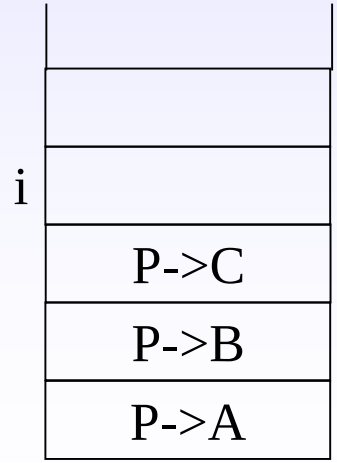
```
void inorder(BiTree T)
{  InitStack(S);
    p=T;
    while(p||!SatckEmpty(S))
    { if (p) {
        Push(p); p=p->lchild;
      }
      else{
        Pop(p);
        printf(p->data);
        p=p->rchild);
      }
    }
}
```





(3)

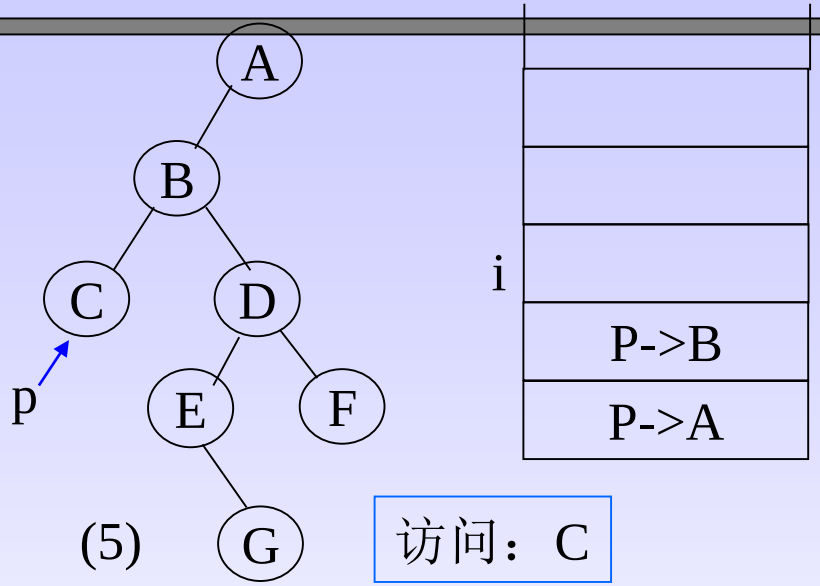


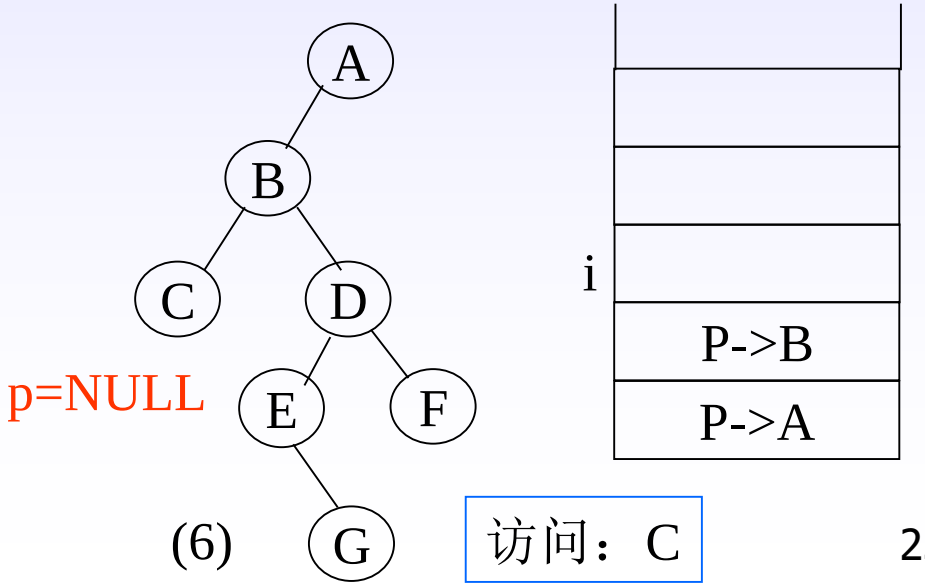


(4)

6.3 遍历二叉树和线索二叉树

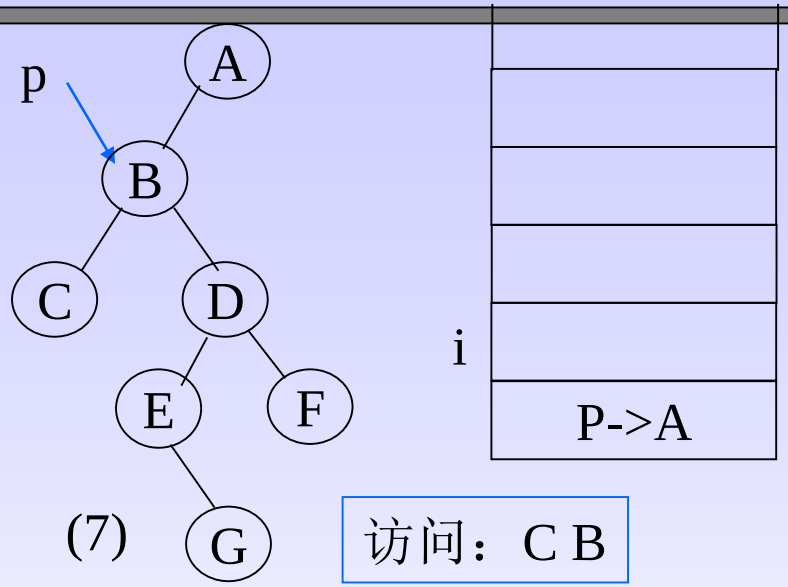
```
void inorder(BiTree T)
{  InitStack(S);
    p=T;
    while(p||!SatckEmpty(S))
    { if (p) {
        Push(p); p=p->lchild;
      }
      else{
        Pop(p);
        printf(p->data);
        p=p->rchild);
      }
    }
}
```

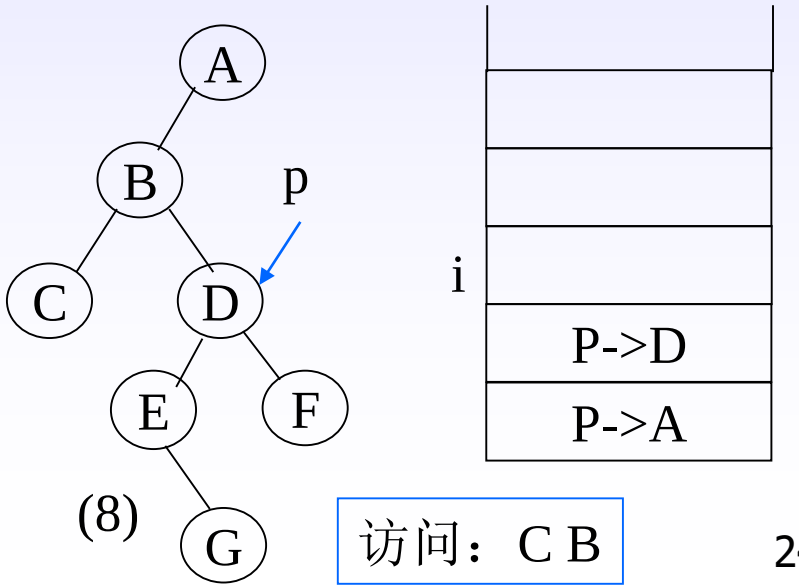




6.3 遍历二叉树和线索二叉树

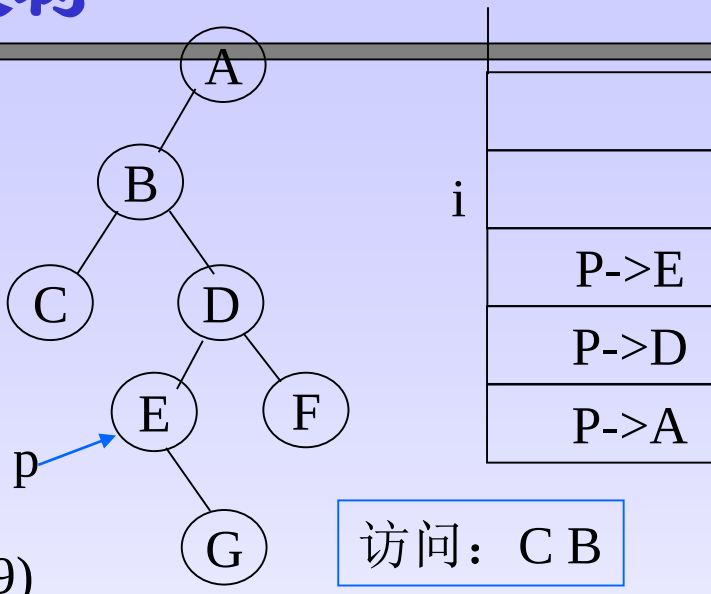
```
void inorder(BiTree T)
{  InitStack(S);
    p=T;
    while(p||!SatckEmpty(S))
    { if (p) {
        Push(p); p=p->lchild;
      }
      else{
        Pop(p);
        printf(p->data);
        p=p->rchild);
      }
    }
}
```

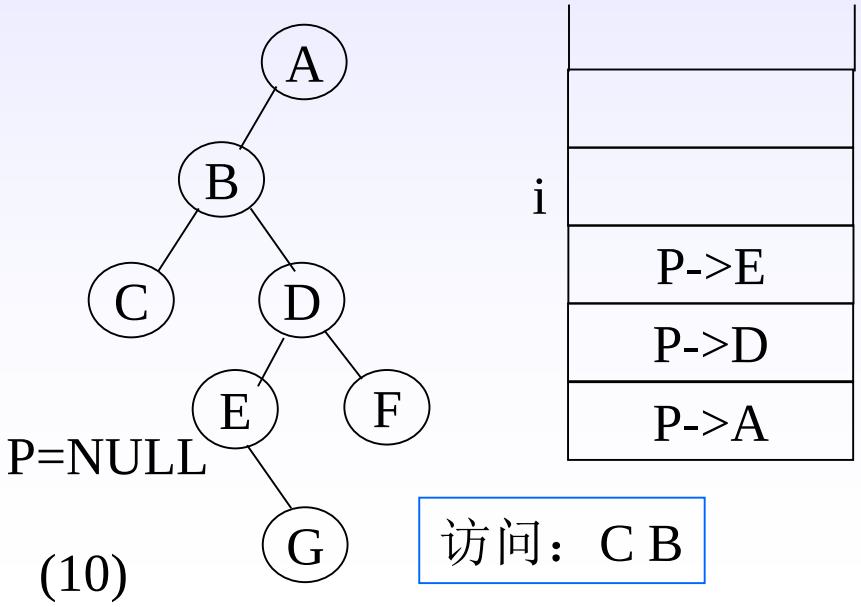




6.3 遍历二叉树和线索二叉树

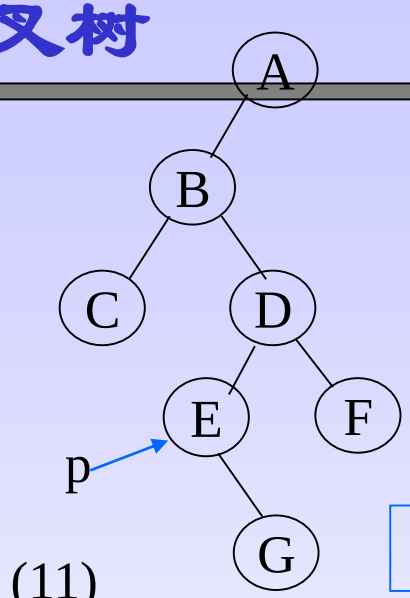
```
void inorder(BiTree T)
{  InitStack(S);
    p=T;
    while(p||!SatckEmpty(S))
    { if (p) {
        Push(p); p=p->lchild;
      }
      else{
        Pop(p);
        printf(p->data);
        p=p->rchild);
      }
    }
}
```

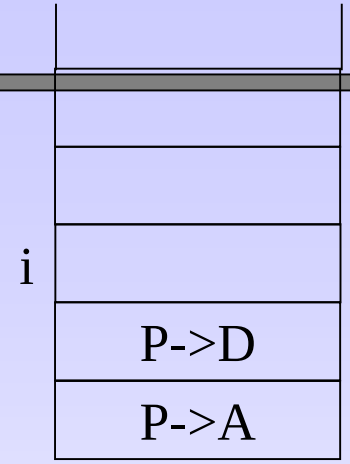




6.3 遍历二叉树和线索二叉树

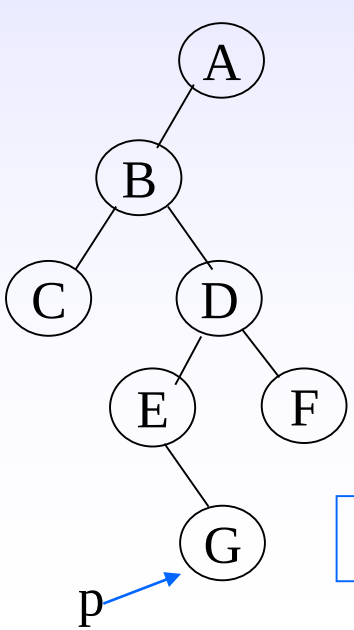
```
void inorder(BiTree T)
{  InitStack(S);
    p=T;
    while(p||!SatckEmpty(S))
    { if (p) {
        Push(p); p=p->lchild;
      }
      else{
        Pop(p);
        printf(p->data);
        p=p->rchild);
      }
    }
}
```

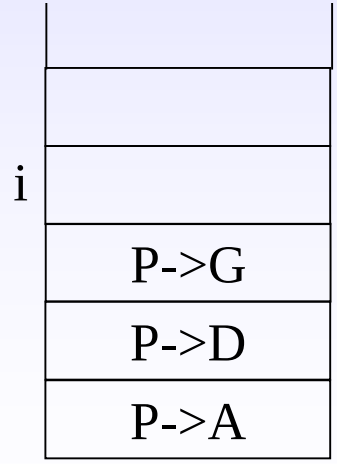




(11)

访问：C B E



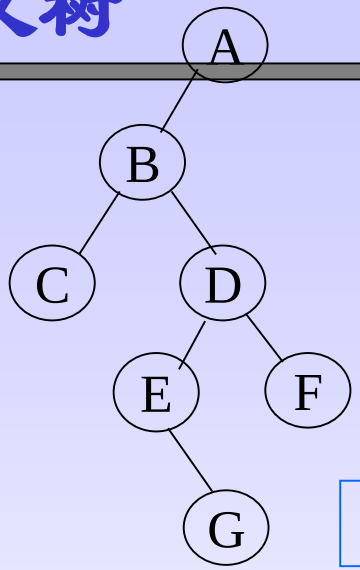


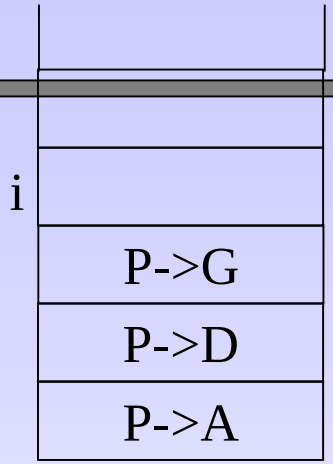
访问：C B E

(12)

6.3 遍历二叉树和线索二叉树

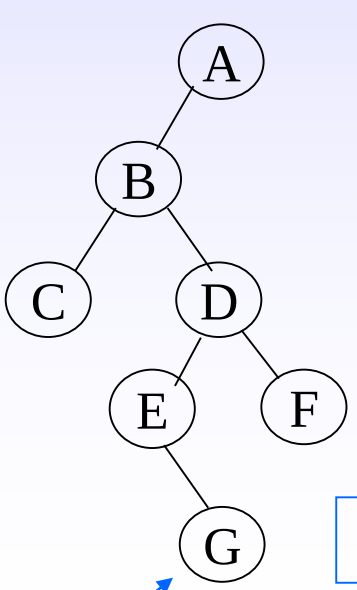
```
void inorder(BiTree T)
{  InitStack(S);
    p=T;
    while(p||!SatckEmpty(S))
    { if (p) {
        Push(p); p=p->lchild;
      }
      else{
        Pop(p);
        printf(p->data);
        p=p->rchild);
      }
    }
}
```

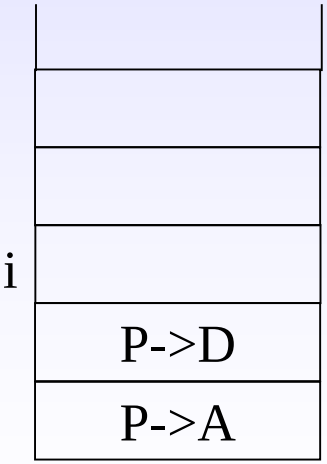




访问：C B E (13)

P=NULL

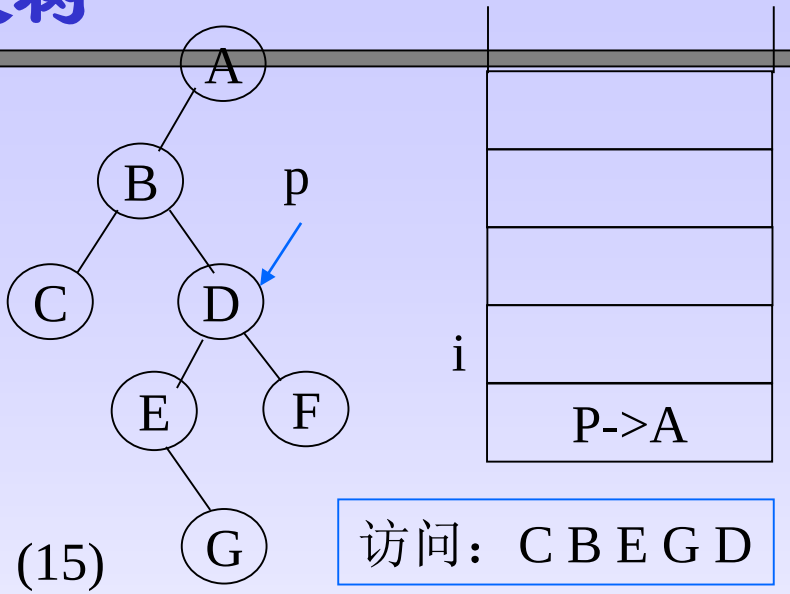


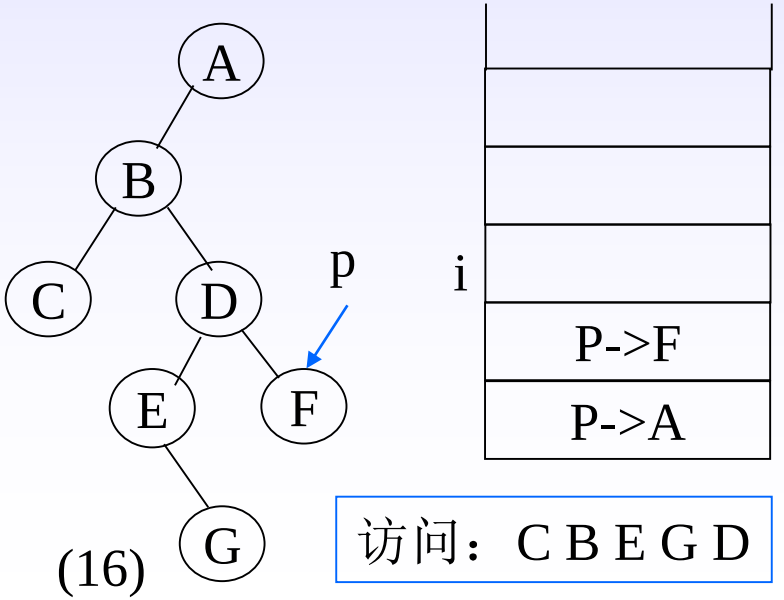


访问：C B E G (14)

6.3 遍历二叉树和线索二叉树

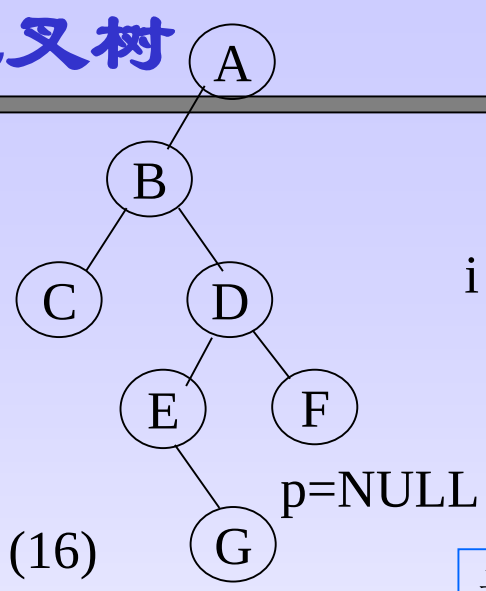
```
void inorder(BiTree T)
{  InitStack(S);
    p=T;
    while(p||!SatckEmpty(S))
    { if (p) {
        Push(p); p=p->lchild;
      }
      else{
        Pop(p);
        printf(p->data);
        p=p->rchild);
      }
    }
}
```



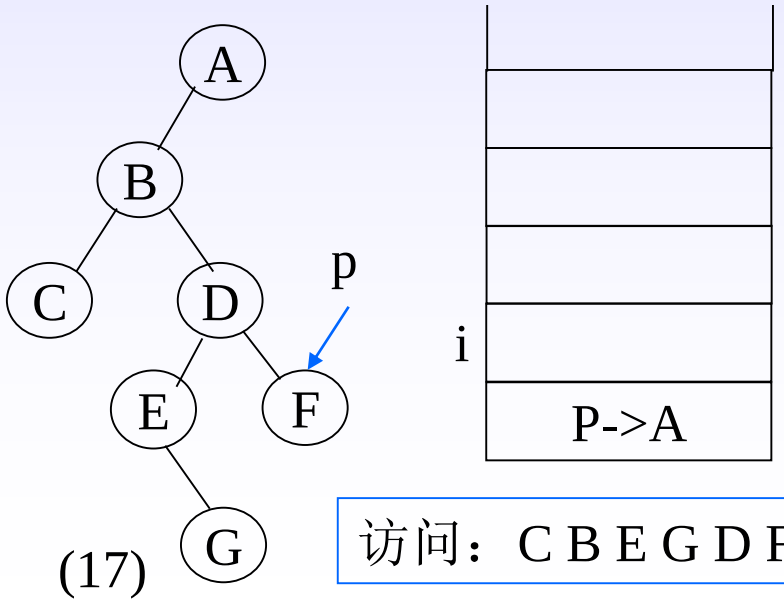


6.3 遍历二叉树和线索二叉树

```
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);
      }
    }
}
```



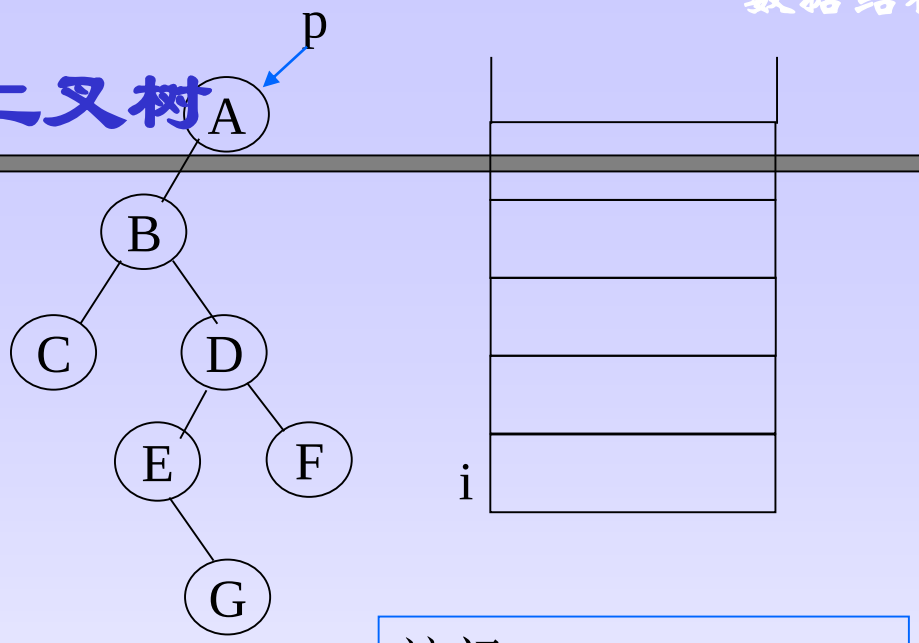
访问：C B E G D



访问：C B E G D F

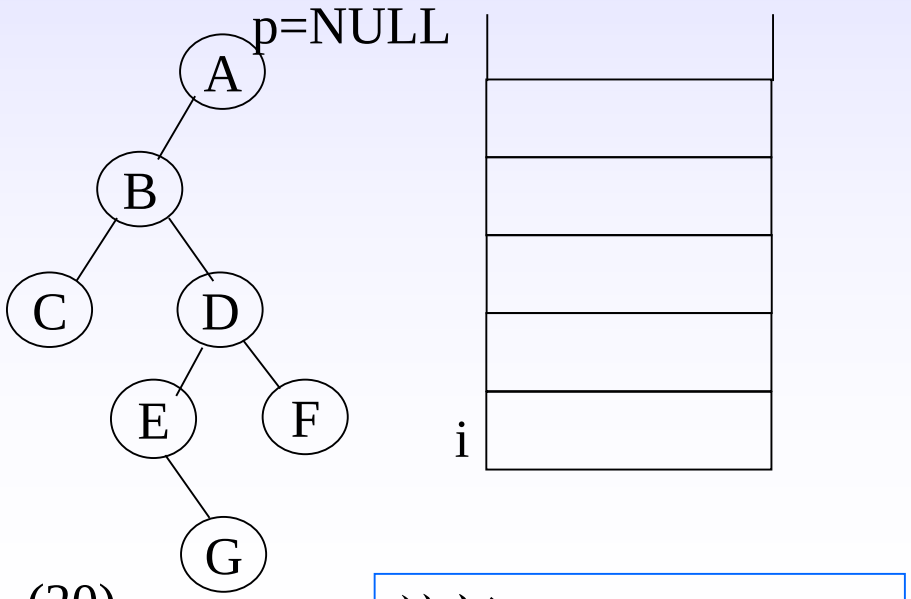
6.3 遍历二叉树和线索二叉树

```
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);
      }
    }
}
```



(19)

访问：C B E G D F A



(20)

访问：C B E G D F A

6.3 遍历二叉树和线索二叉树

二叉树的先序序列 根 左子树 右子树

二叉树的中序序列 左子树 根 右子树

二叉树的后序序列 左子树 右子树 根

由二叉树的先序序列可唯一地确定一棵二叉树？

6.3 遍历二叉树和线索二叉树

二叉树的先序序列 根 左子树 右子树

二叉树的中序序列 左子树 根 右子树

二叉树的后序序列 左子树 右子树 根

由二叉树的中序序列可唯一地确定一棵二叉树？

6.3 遍历二叉树和线索二叉树

二叉树的先序序列 根 左子树 右子树

二叉树的中序序列 左子树 根 右子树

二叉树的后序序列 左子树 右子树 根

由二叉树的后序序列可唯一地确定一棵二叉树？

6.3 遍历二叉树和线索二叉树

二叉树的先序序列 根 左子树 右子树

二叉树的中序序列 左子树 根 右子树

二叉树的后序序列 左子树 右子树 根

由二叉树的先序序列和后序序列唯一地确定一棵二叉树？

6.3 遍历二叉树和线索二叉树

二叉树的先序序列 根 左子树 右子树

二叉树的中序序列 左子树 根 右子树

二叉树的后序序列 左子树 右子树 根

由二叉树的先序序列和中序序列唯一地确定一棵二叉树？

6.3 遍历二叉树和线索二叉树

二叉树的先序序列 根 左子树 右子树

二叉树的中序序列 左子树 根 右子树

二叉树的后序序列 左子树 右子树 根

由二叉树的中序序列和后序序列唯一地确定一棵二叉树？

6.3 遍历二叉树和线索二叉树

二叉树的先序序列 根 左子树 右子树

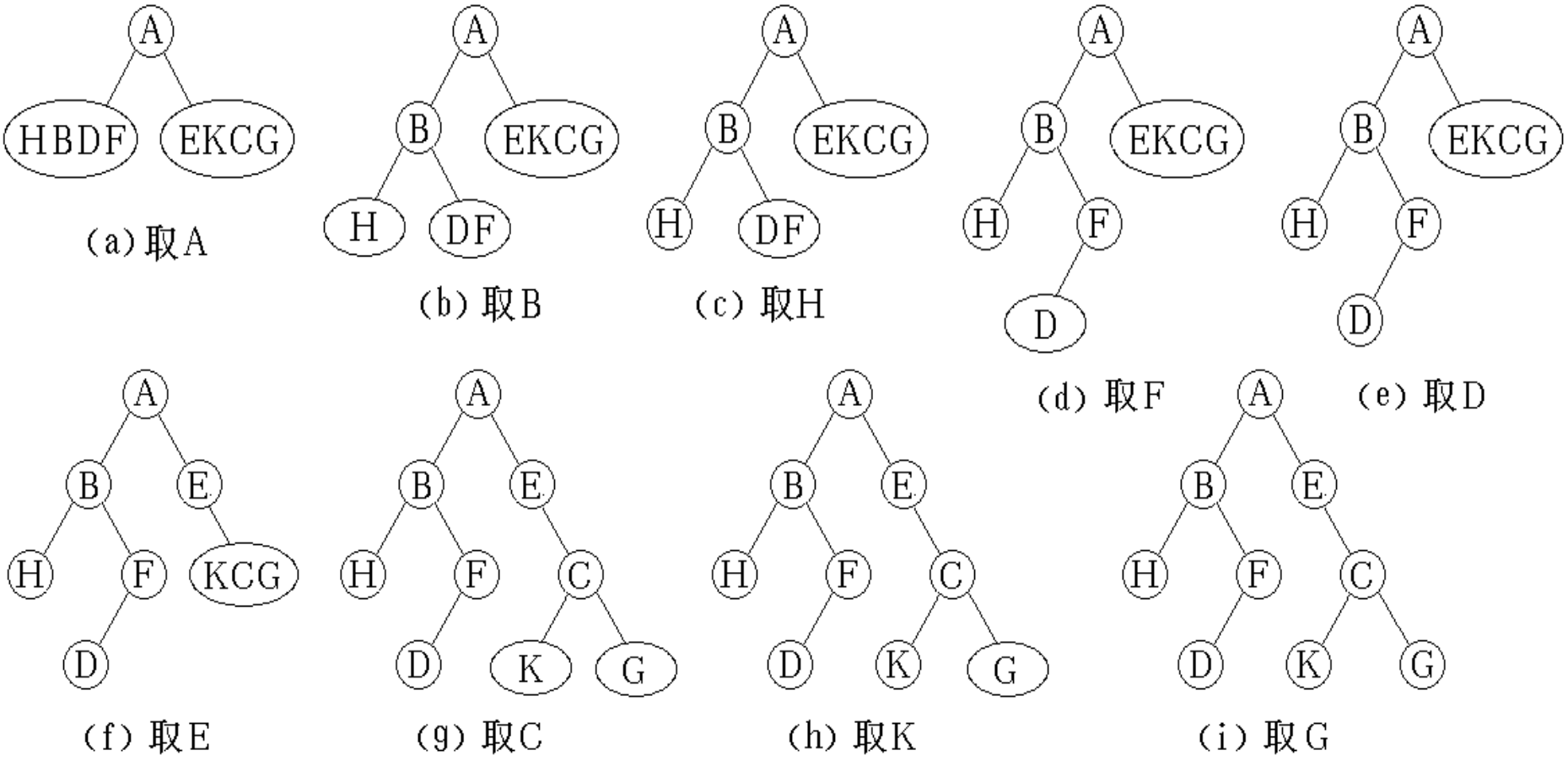
二叉树的中序序列 左子树 根 右子树

二叉树的后序序列 左子树 右子树 根

由二叉树的先序序列（或后序序列）和中序序列可唯一地确定一棵二叉树。

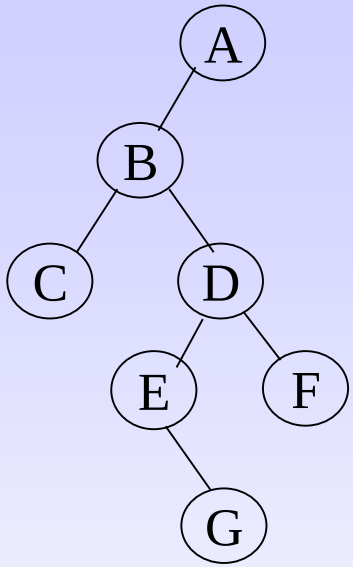
6.3 遍历二叉树和线索二叉树

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



6.3 遍历二叉树和线索二叉树

3. 层序遍历算法




```
void levelorder(BiTree T) //二叉树层序遍历
{ InitQueue(Q);
   if(T) EnQueue(Q,T);
   while(!QueueEmpty(Q)) // 队列非空
   {  DeQueue(Q,p);
      printf(p->data);        // 先序访问根结点
      if(p->lchild)  EnQueue(Q,p->lchild);
      if(p->rchild)  EnQueue(Q,p->rchild);
   }
}
```

6.3 遍历二叉树和线索二叉树

4. 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
{  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);
}
```

6.3 遍历二叉树和线索二叉树

4. 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（什么序？）
{  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);
}
```

6.3 遍历二叉树和线索二叉树

4. 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);
}
```

6.3 遍历二叉树和线索二叉树

```
BiTree TreeCopy(BiTree T) //二叉树复制（什么序？）
{ 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);
}
```

6.3 遍历二叉树和线索二叉树

```
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);
}
```

6.3 遍历二叉树和线索二叉树

```
BiTNode *findnode(BiTree T, TElemtype x)
  // 二叉树查找（什么序？）
{  if ( T == NULL)
        return(NULL);
    else  if ( T->data == x)
        return(T);
    else
        return(findnode(T->lchild,x)||findnode(T->rchild,x));
}
```

6.3 遍历二叉树和线索二叉树

```
BiTNode *findnode(BiTree T, TElemtype x)
  // 二叉树查找（先序）
{  if ( T == NULL)
        return(NULL);
    else  if ( T->data == x)
        return(T);
    else
        return(findnode(T->lchild,x)||findnode(T->rchild,x));
}
```

6.3 遍历二叉树和线索二叉树

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性结构的二叉树进行线性化处理。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，可存放其前驱和后继的指针。

6.2 二叉树

链式存储结构：

二叉链表


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

6.3 遍历二叉树和线索二叉树

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性结构的二叉树进行线性化处理。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，可存放其前驱和后继的指针。
- 结点结构：

lchild	ltag	data	rtag	rchild

```
typedef struct BiThrNode
{  TELemType          data;
   struct BiThrNode *lchild;
   struct BiThrNode *rchild;
   unsigned           ltag;
   unsigned           rtag;
} BitThrNode, *BiThrTree;
```

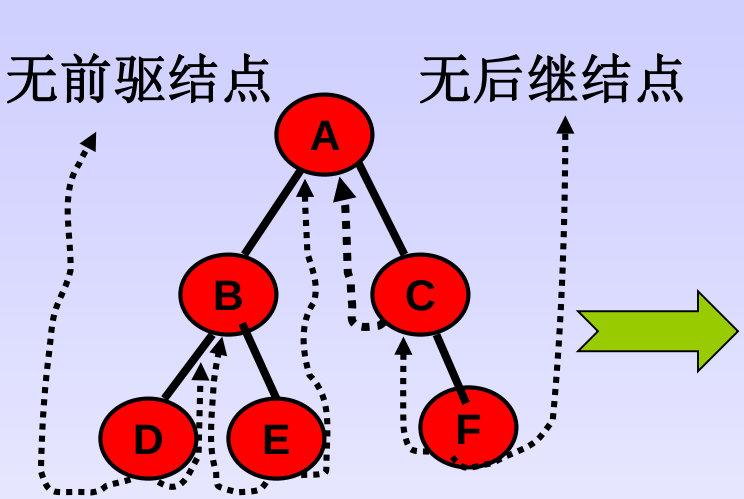
ltag=0，lchild指向左子结点

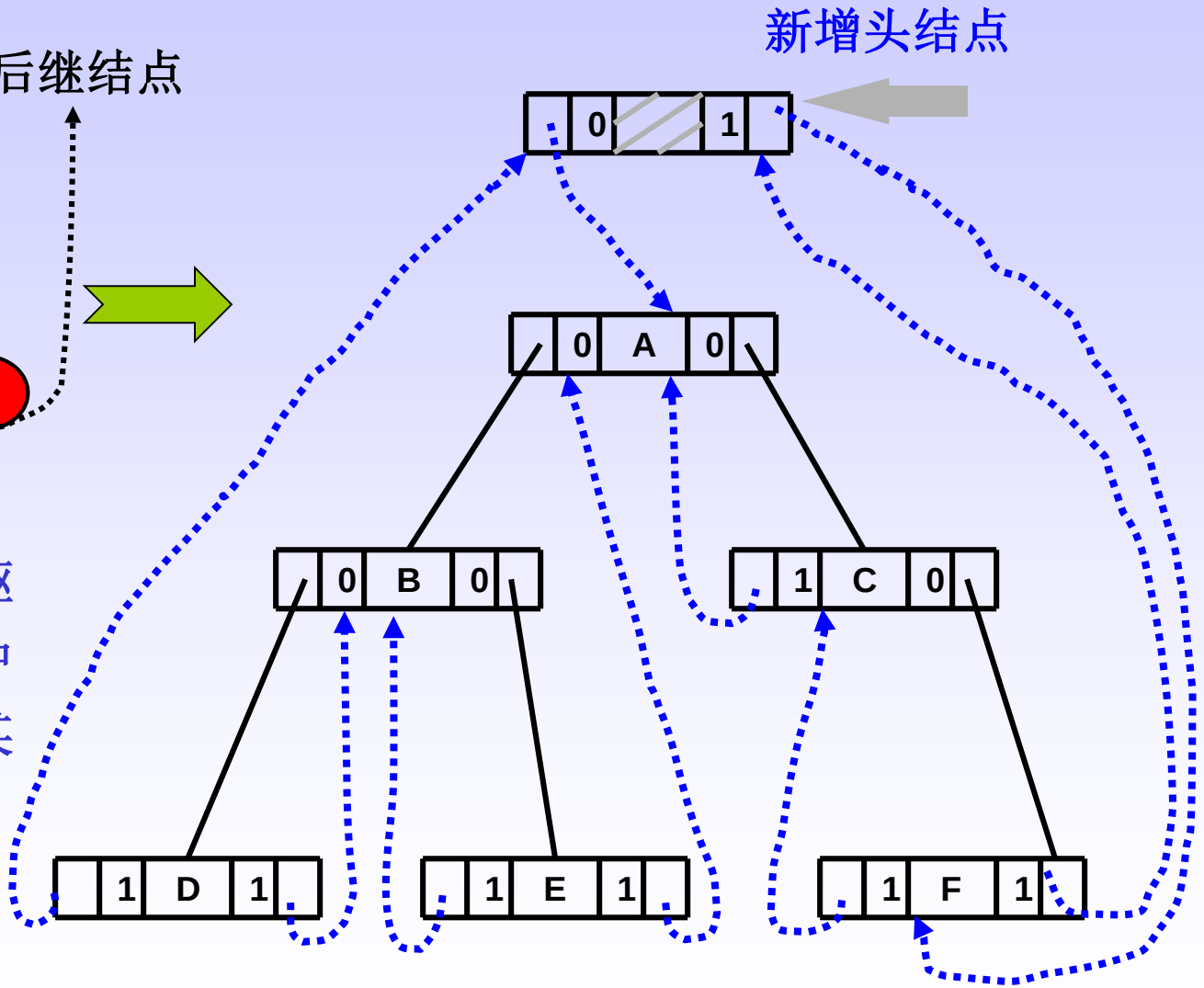
ltag=1，lchild指向前驱结点

rtag=0，rchild指向右子结点

rtag=1，rchild指向后继结点

6.3 遍历二叉树和线索二叉树



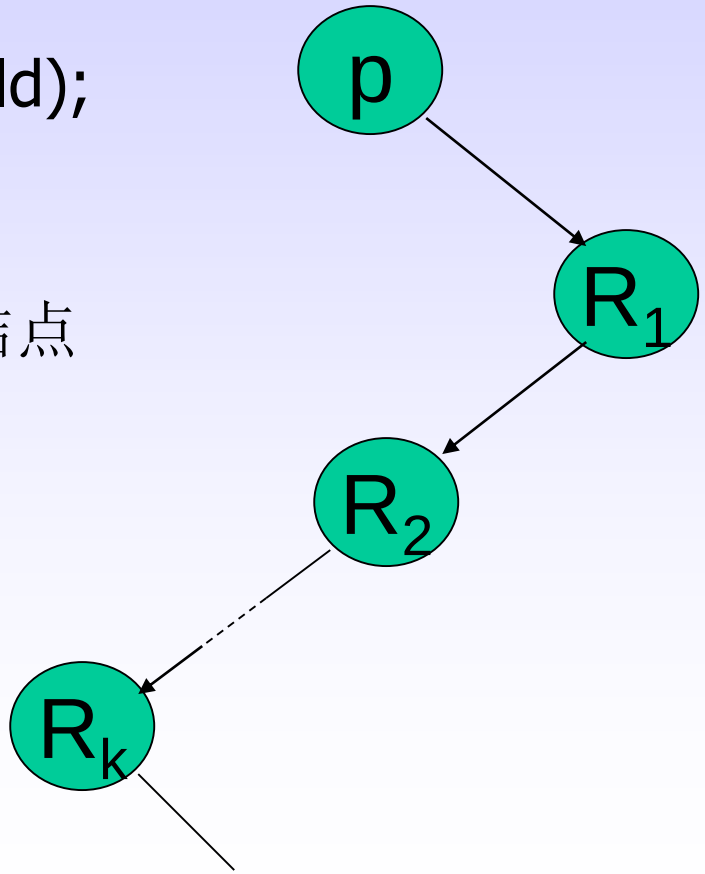


注：本小节中的前驱和后继指遍历序列中线性化逻辑关系中某结点的前驱和后继，不是其父子结点。

6.3 遍历二叉树和线索二叉树

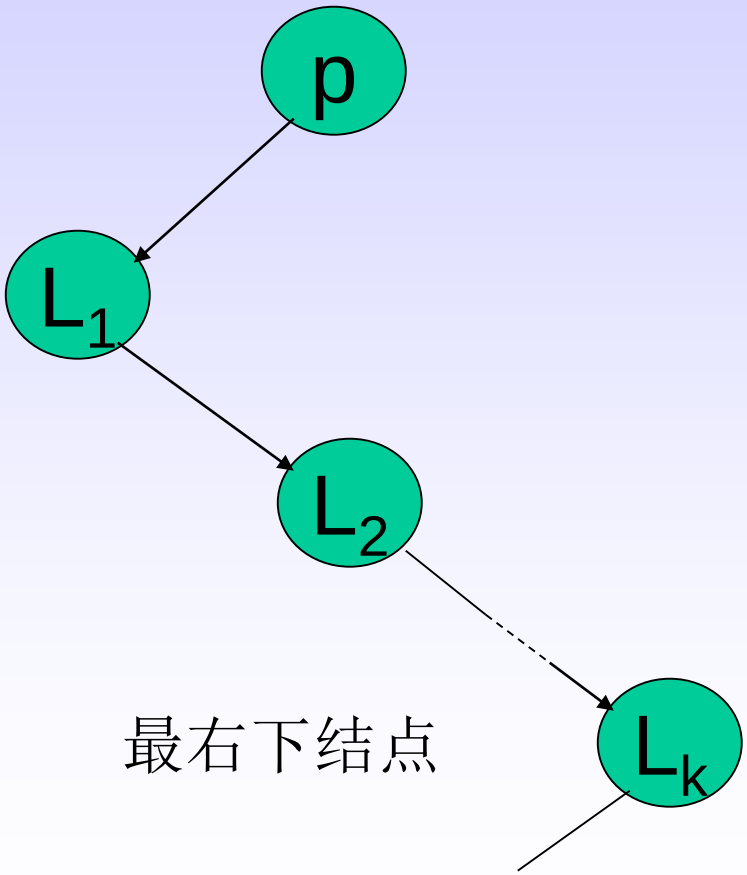
2 后继的查找

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



6.3 遍历二叉树和线索二叉树

前驱的查找




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

6.3 遍历二叉树和线索二叉树

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)
{  p=T->lchild;
   while(p->Ltag==0)
      p=p->lchild; //查找最左结点
   while(p!=T){
      printf(p->data);
      p= nextnode(p);
   }
}
```