第6章 树和二叉树

6.1 树的定义和基本术语

6.2 二叉树

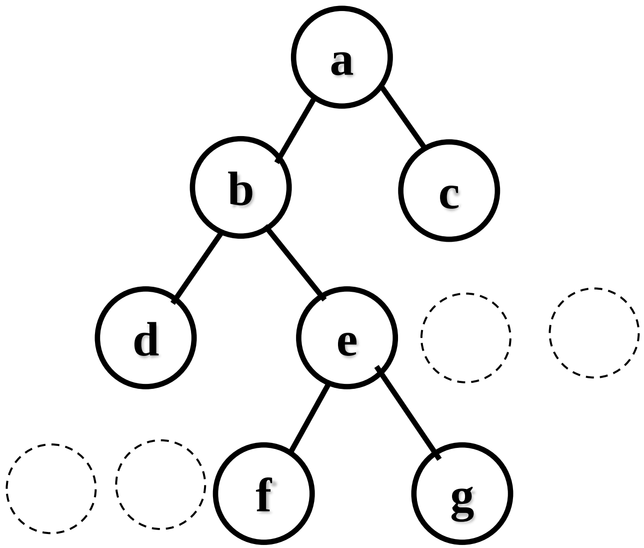
6.3 遍历二叉树和线索二叉树

6.4 树和森林

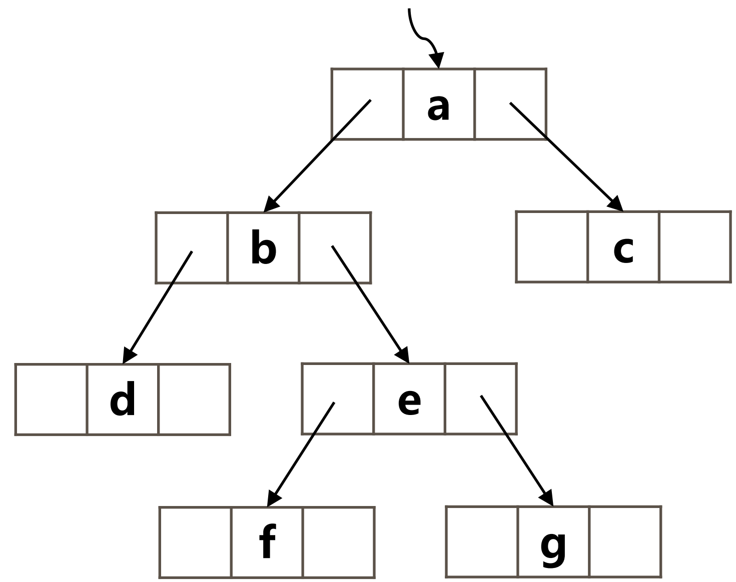
6.5 哈夫曼树及其应用

回顾

□ 二叉树的顺序存储结构



1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	0	0	0	0	f	g



□ 二叉树的链式存储结构

二叉链表、三叉链表、静态链表

回顾

二叉树的遍历

先序（根）遍历、中序（根）遍历、后序（根）遍历、层次遍历

```
void Traversal (BiTree T, void(*visit)(TElemType& e)){
  if (T!==NULL) {
    Traversal (T->lchild, visit); // 遍历左子树
    Traversal (T->rchild, visit);// 遍历右子树
    visit(T->data);
  }
}
```

非递归算法

先序（根）遍历、中序（根）遍历、后序（根）遍历——栈

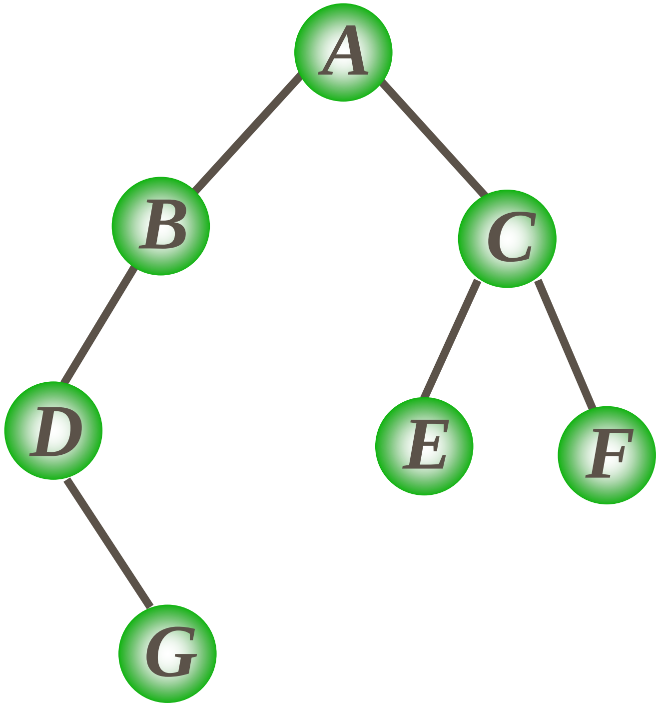
层次遍历——队列

回顾

例：编写 求二叉树的叶子结点个数的算法
输入：二叉树的二叉链表
结果：二叉树的叶子结点个数

基本思想：遍历操作访问二叉树的每个结点，而且每个结点仅被访问一次。所以可在二叉树遍历的过程中，统计叶子结点的个数。

```
void leaf(BiTree T) {
//n计数二叉树的叶子结点的个数，初值n=0
if(T) {
    if(T->lchild==null&&T->rchild==null)
        n=n+1;
    leaf(T->lchild);
    leaf(T->rchild);
   }//if
}//leaf
```



回顾

例：求二叉树结点个数

```
int Size(BiTree T)
{
   if (T==NULL)
     return 0;
   else
     return 1 + Size(T->lchild ) + Size(T->rchild);
}
```

回顾

例：利用二叉树后序遍历计算二叉树的深度

```
int Depth(BiTree T){
    int depl,depr;
    if (T){
        depl=Depth(T->lchild);
        depr=Depth(T->rchild);
        if (depl>=depr) return (depl+1);
        else return (depr+1);
    }
    return 0;
}
```

回顾

例：左右子树互换

```
void Exchange(BiTree &T)
{
   BiNode* p;
   if (T) {
      p=T->lchild;
      T->lchild=T->rchild;
      T->rchild=p;
      Exchange(T->lchild);
      Exchange(T->rchild);
   }
}
```

回顾

例：复制二叉树

```
void CopyTree(BiTree T,BiTree &T1){
  if(T){
     T1=(BiTree)malloc(sizeof(BiTNode));
     if (!T1){
       printf("Overflow\n");
       exit(1);
     }
     T1->data=T->data;
     T1->lchild=T1->rchild=NULL;
     CopyTree(T->lchild,T1->lchild);
     CopyTree(T->rchild,T1->rchild);
  }
}
```

6.3.2 构造二叉树

由遍历序列恢复二叉树

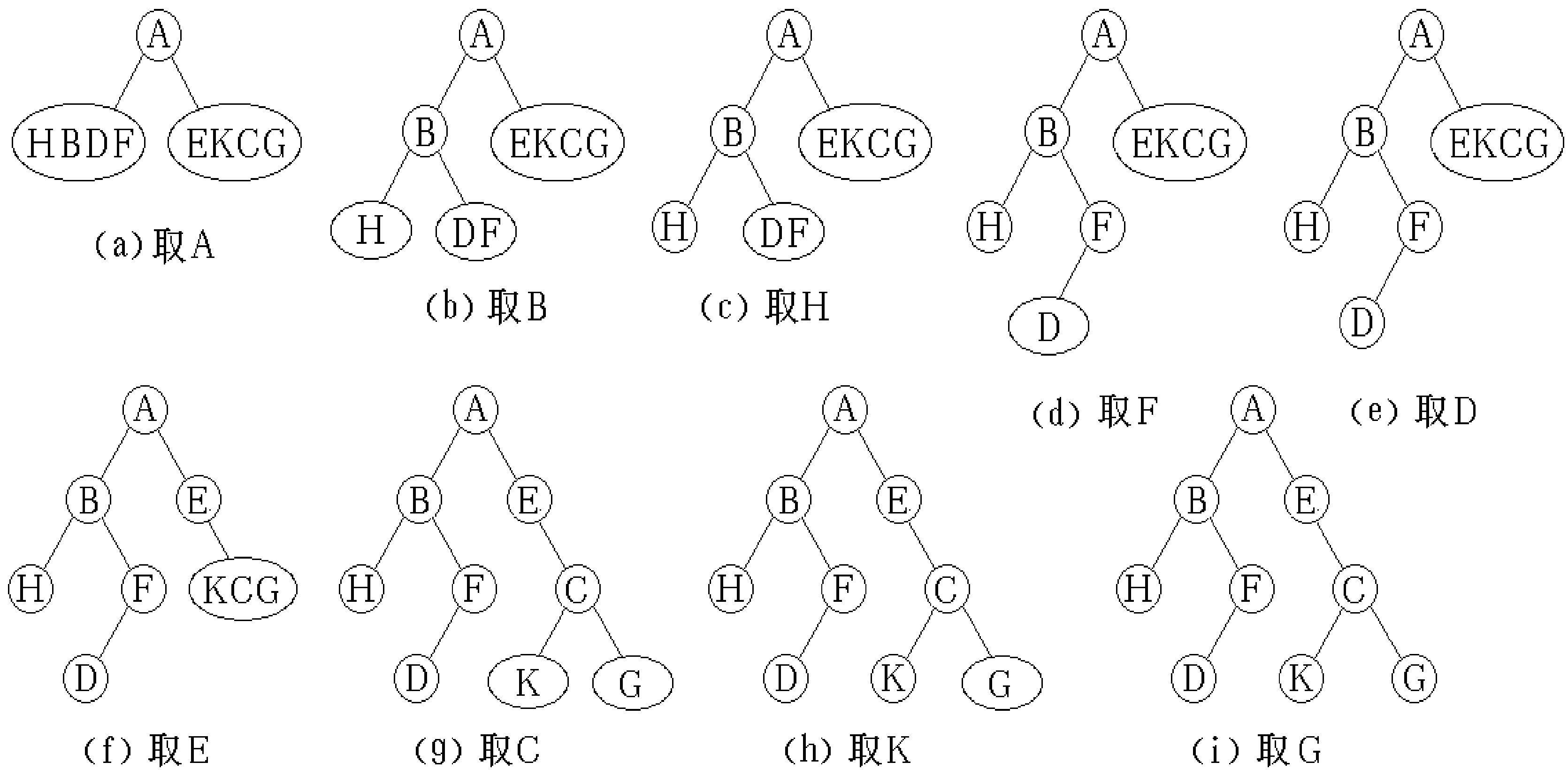
若已知一棵二叉树的先序（或中序，或后序，或层序）序列，能否惟一确定这棵二叉树呢？

遍历的性质

- 性质1、由一棵二叉树的先序序列和中序序列可惟一确定这棵二叉树
- 性质2、由一棵二叉树的后序序列和中序序列可惟一确定这棵二叉树

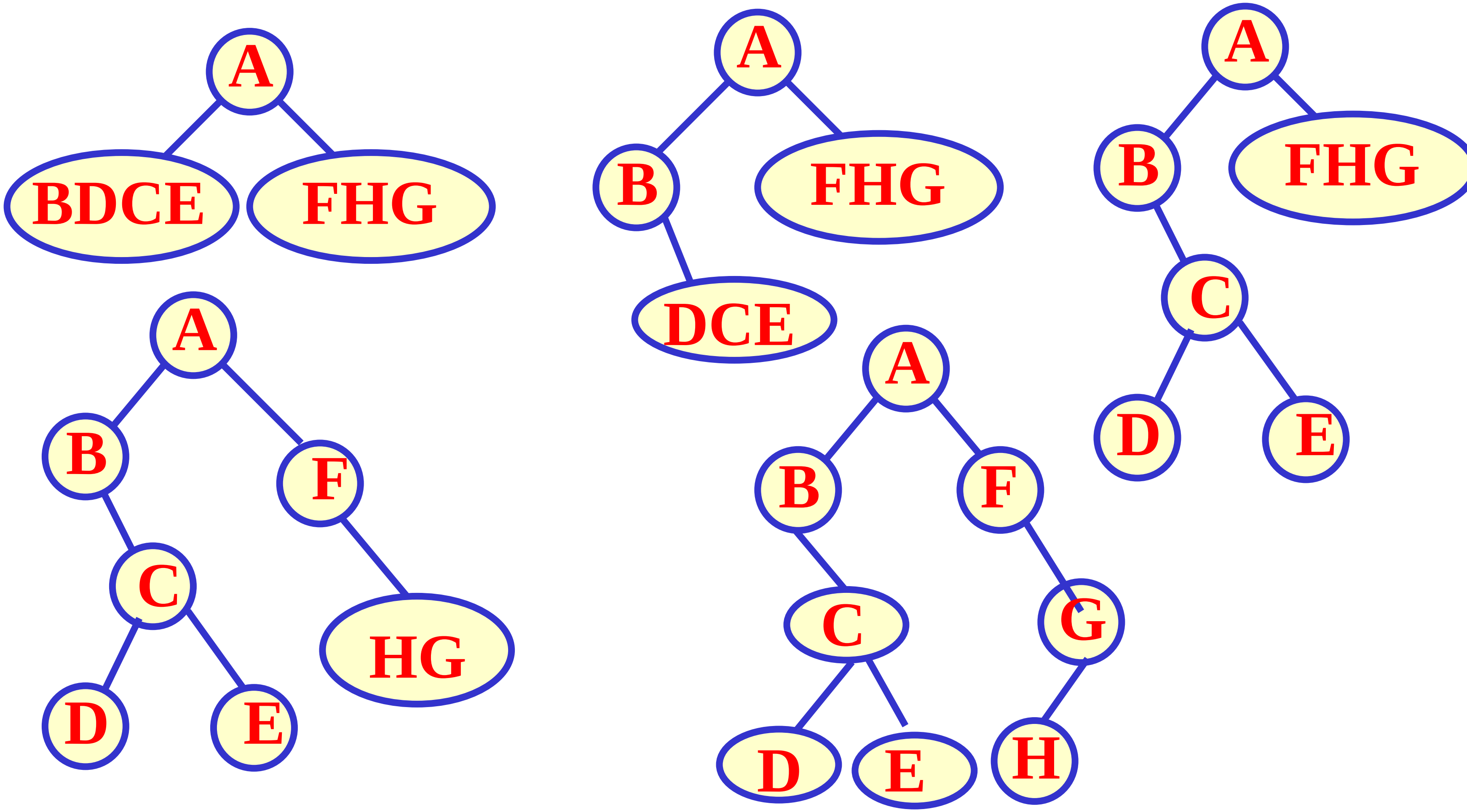
6.3.2 构造二叉树

由二叉树的先序序列和中序序列可唯一地确定一棵二叉树。
例，先序序列｛ABHFDECKG｝和中序序列｛HBDFAEKCG｝，构造二叉树过程如下：



6.3.2 构造二叉树

例如，已知一棵二叉树的中序序列和后序序列分别为 **BDCEAFHG和DECBHGFA，试画出这棵二叉树。**



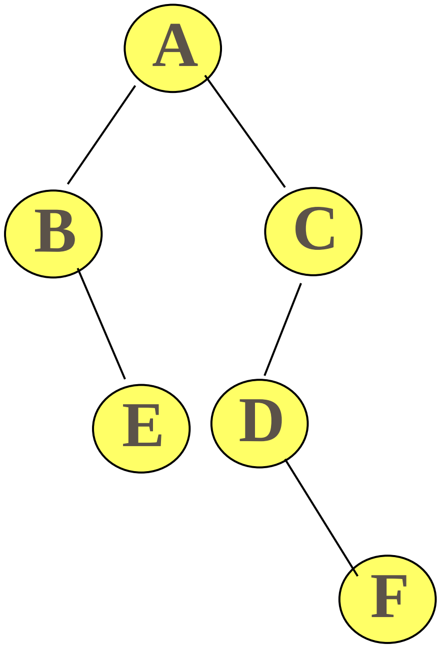
6.3.2 构造二叉树

例：已知前序和中序遍历序列，画出二叉树，写出后序遍历序列。

前序序列为：A B E C D F

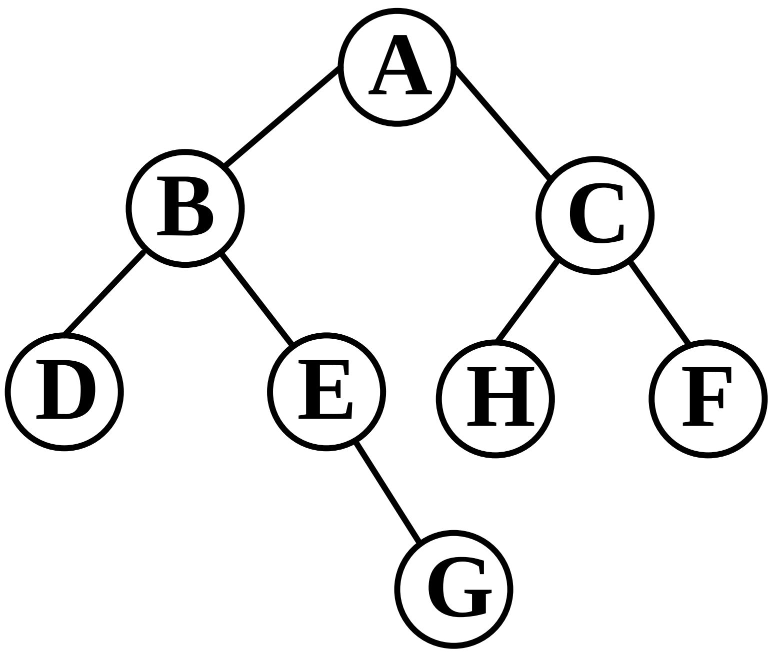
中序序列为：B E A D F C

后序序列为：E B F D C A



6.3.2 构造二叉树

例 已知一棵二叉树后序遍历序列为DGEBHFCA，中序遍历序列为DBEGAHCF，画出该二叉树，并写出二叉树的先序遍历序列。



先序遍历序列：ABDEGCHF

6.3.2 构造二叉树

为二叉树建立二叉存储链表

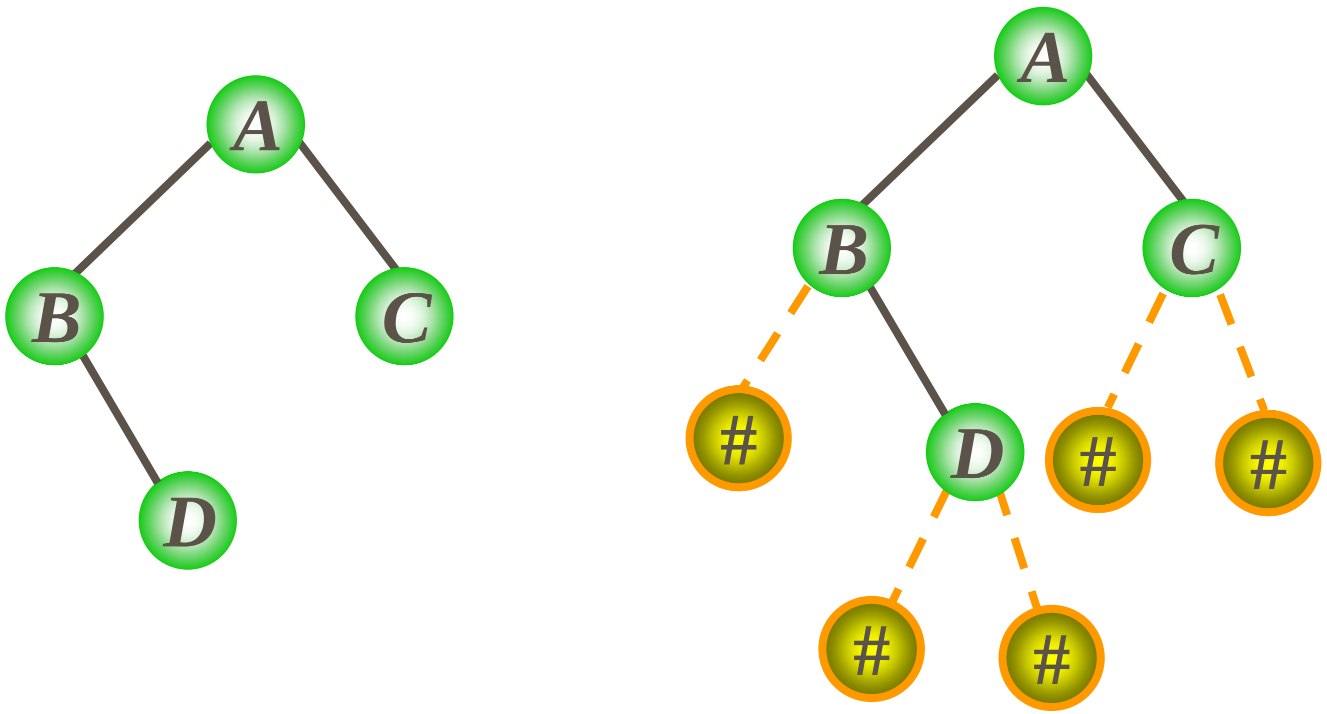
输入：二叉树的先序序列

结果：建立二叉树的二叉存储链表

为了建立一棵二叉树，将二叉树中每个结点的空指针引出一个虚结点，其值为一特定值如“#”，以标识其为空，把这样处理后的二叉树称为原二叉树的扩展二叉树。

按先序编历的顺序输入先序序列（设每个元素是一个字符），建立二叉链表的所有结点并完成相应结点的链接。

6.3.2 构造二叉树



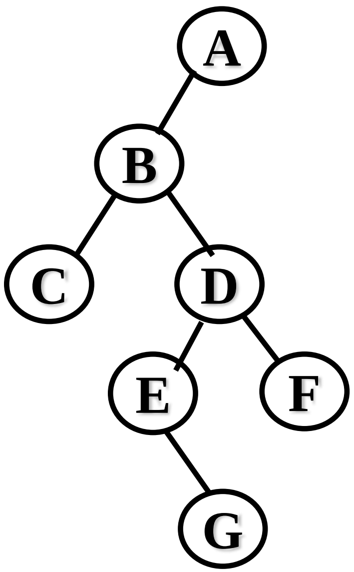
扩展二叉树的先序遍历序列: *A B # D # # C # #*

6.3.2 构造二叉树

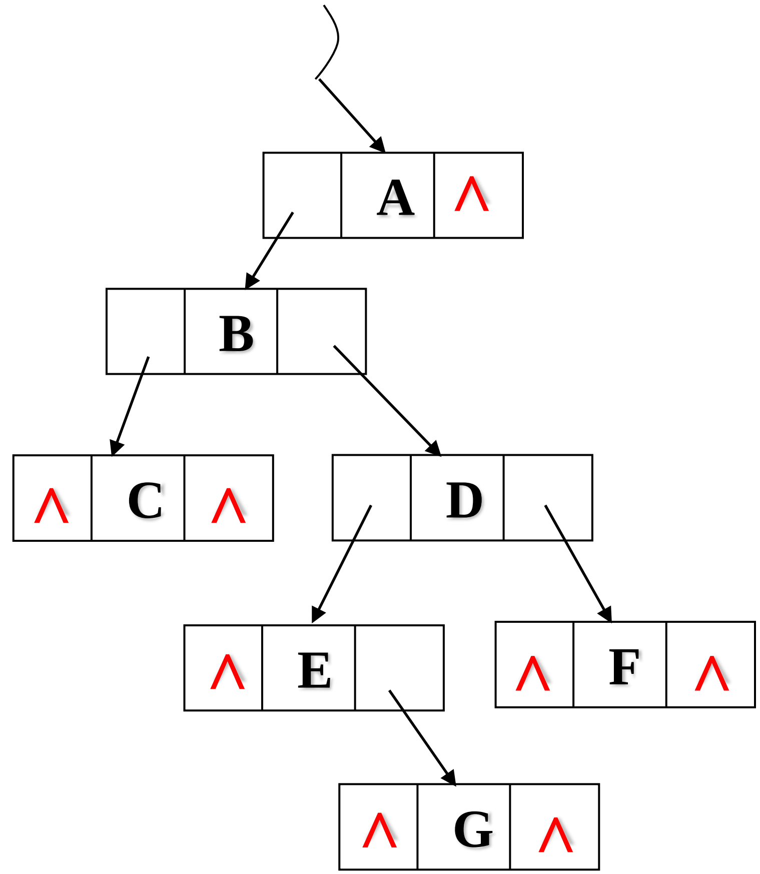
先序建立二叉树的递归算法(p131, 算法6.4)

```
Status CreateBiTree(BiTree &T){
// 假设扩展二叉树的先序遍历序列由键盘输入，T为指向根结点的指针
    scanf(&ch);
    if (ch == ‘#’) T = NULL; // 若ch== ‘#’ 则T=NULL返回
    else{ //   若ch! = ‘  ’
        if (!(T = (BiTNode *)malloc(sizeof(BiTNode))))
            exit(OVERFLOW);
        T->data = ch;           // 建立（根）结点
        CreateBiTree(T->lchild); // 构造左子树，并将左子树根结点指针
                                 // 赋 给（根）结点 的左孩子域
        CreateBiTree(T->rchild); // 构造右子树，并将右子树根结点指针
                                 // 赋给（根）结点 的右孩子域
    }
    return OK;
}
```

6.3.3 线索二叉树

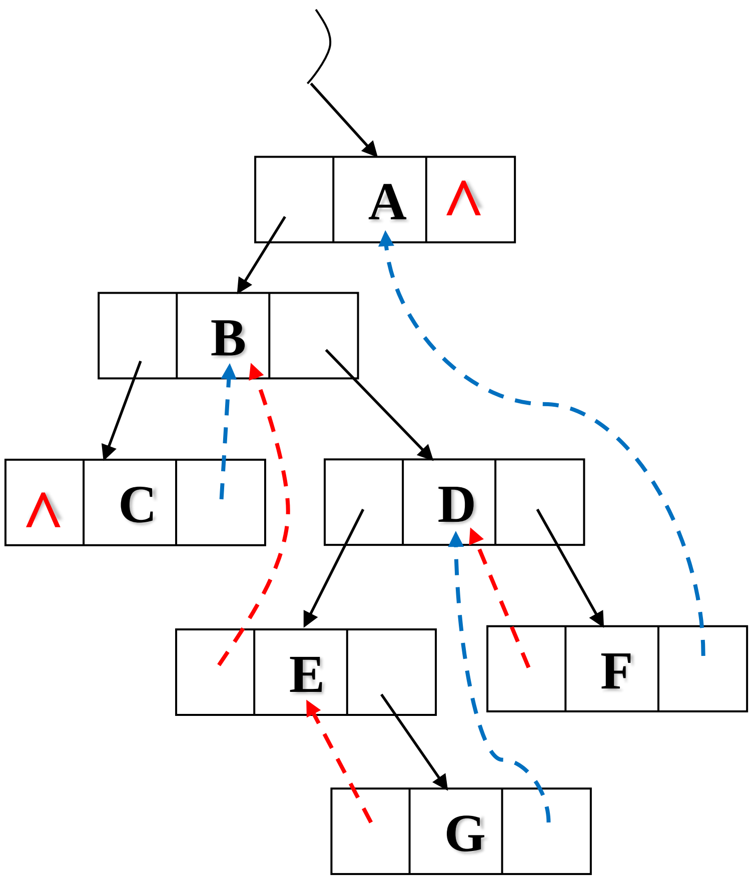


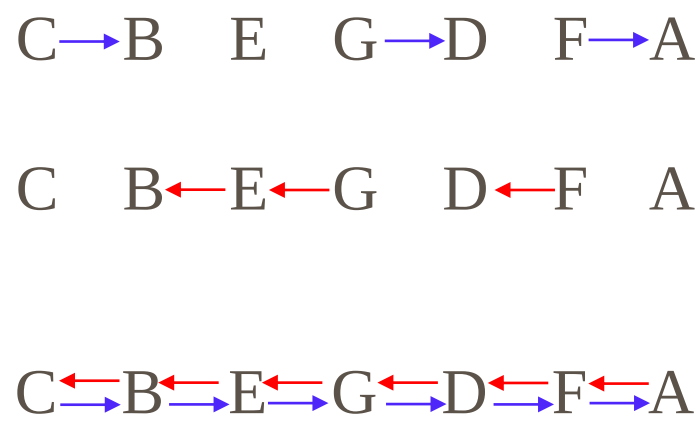
能否充分利用空指针域?



注意：在n个结点的二叉链表中，有n+1个空指针域:2n-(n-1)=n+1

6.3.3 线索二叉树





LTag	lchild	data	rchild	RTag

LTag：0为指向左孩子，1为指向前驱
RTag：0为指向右孩子，1为指向后驱

6.3.3 线索二叉树

有关线索二叉树的几个术语

线索链表： 用含Tag的结点样式所构成的二叉链表

线　索： 指向结点前驱和后继的指针

线索二叉树： 加上线索的二叉树

线　索　化： 对二叉树以某种次序遍历使其变为线索二叉树的过程

实质：对一个非线性结构进行线性化操作，使每个结点（除第一和最后一个外）在这些线性序列中有且仅有一个直接前驱和直接后继。

6.3.3 线索二叉树

线索二叉树的存储表示 p133

```
typedef enum{Link,Thread}PointerTag;
//Link==0: 指针，指向孩子结点
//Thread==1: 线索，指向前驱或后继结点
typedef struct BiThrNode{
    TElemType data;
    struct BiThrNode *lchild,*rchild;
    PointerTag LTag,RTag;
}BiThrNode, *BiThrTree;
BiThrTree T;
```

6.3.3 线索二叉树

线索化过程就是在遍历过程中修改空指针的过程:

① 将空的**lchild**改为结点的直接前驱;

② 将空的**rchild**改为结点的直接后继。

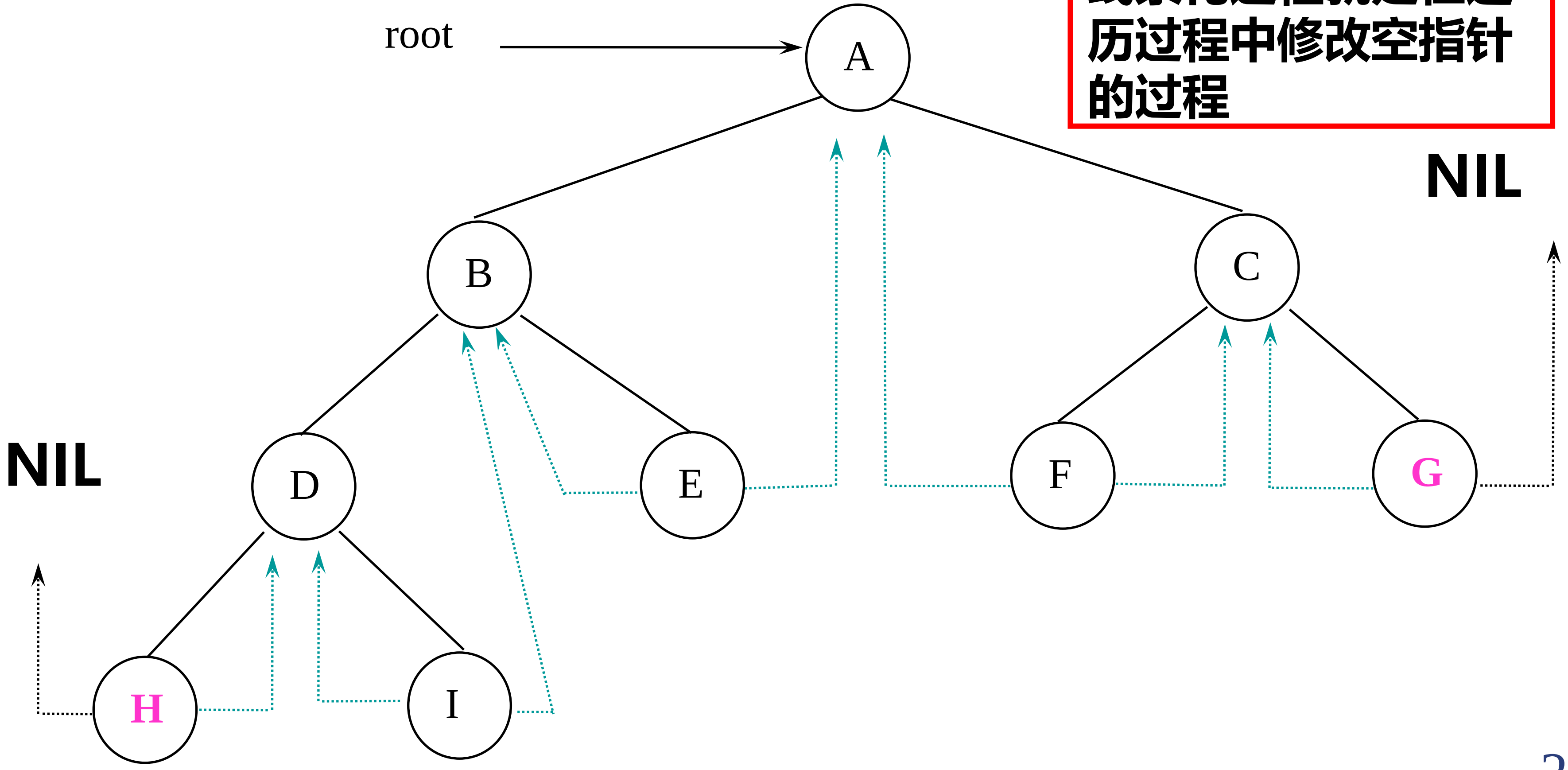
③ 非空指针呢? 仍然指向孩子结点（称为“正常情况”）

6.3.3 线索二叉树

例：画出以下二叉树对应的中序线索二叉树。

解：对该二叉树中序遍历的结果为：H, D, I, B, E, A, F, C, G
所以添加线索应当按如下路径进行：

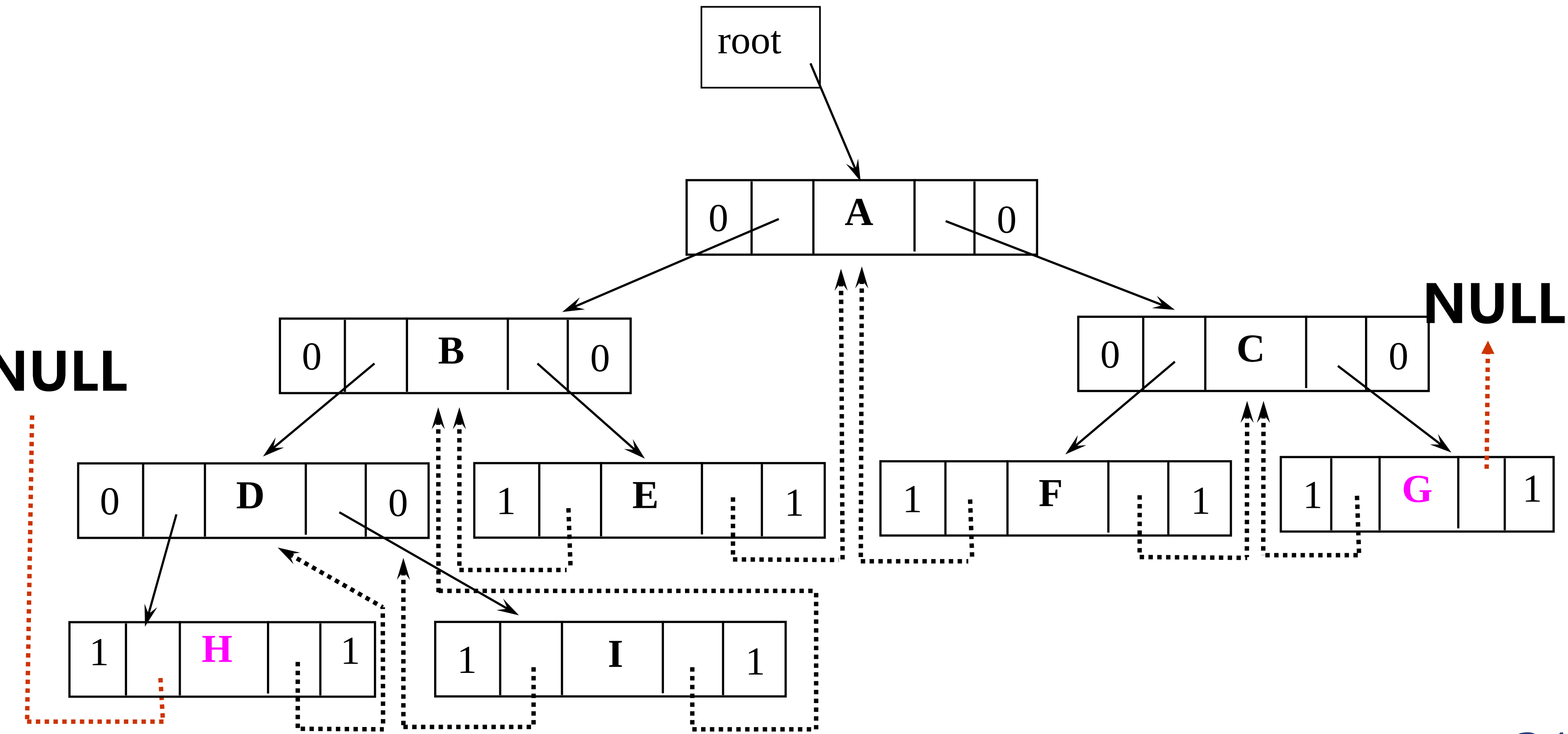
线索化过程就是在遍历过程中修改空指针的过程



6.3.3 线索二叉树

对应的中序线索二叉树存储结构如图所示：

注： 此图中序遍历结果为：H, D, I, B, E, A, F, C, G



6.3.3 线索二叉树

线索二叉树的生成算法

思想：在遍历二叉树的过程中修改空指针，添加前驱或后继的线索，使之成为线索二叉树。

为了记下遍历过程中访问结点的先后次序，需要设置两个指针：

p指针→当前结点之指针；pre指针→当前结点的前趋结点指针。

每次只修改前驱结点的右指针（后继）和本结点的左指针（前驱）

6.3.3 线索二叉树

通过中序遍历建立中序线索化二叉树(算法6.7)

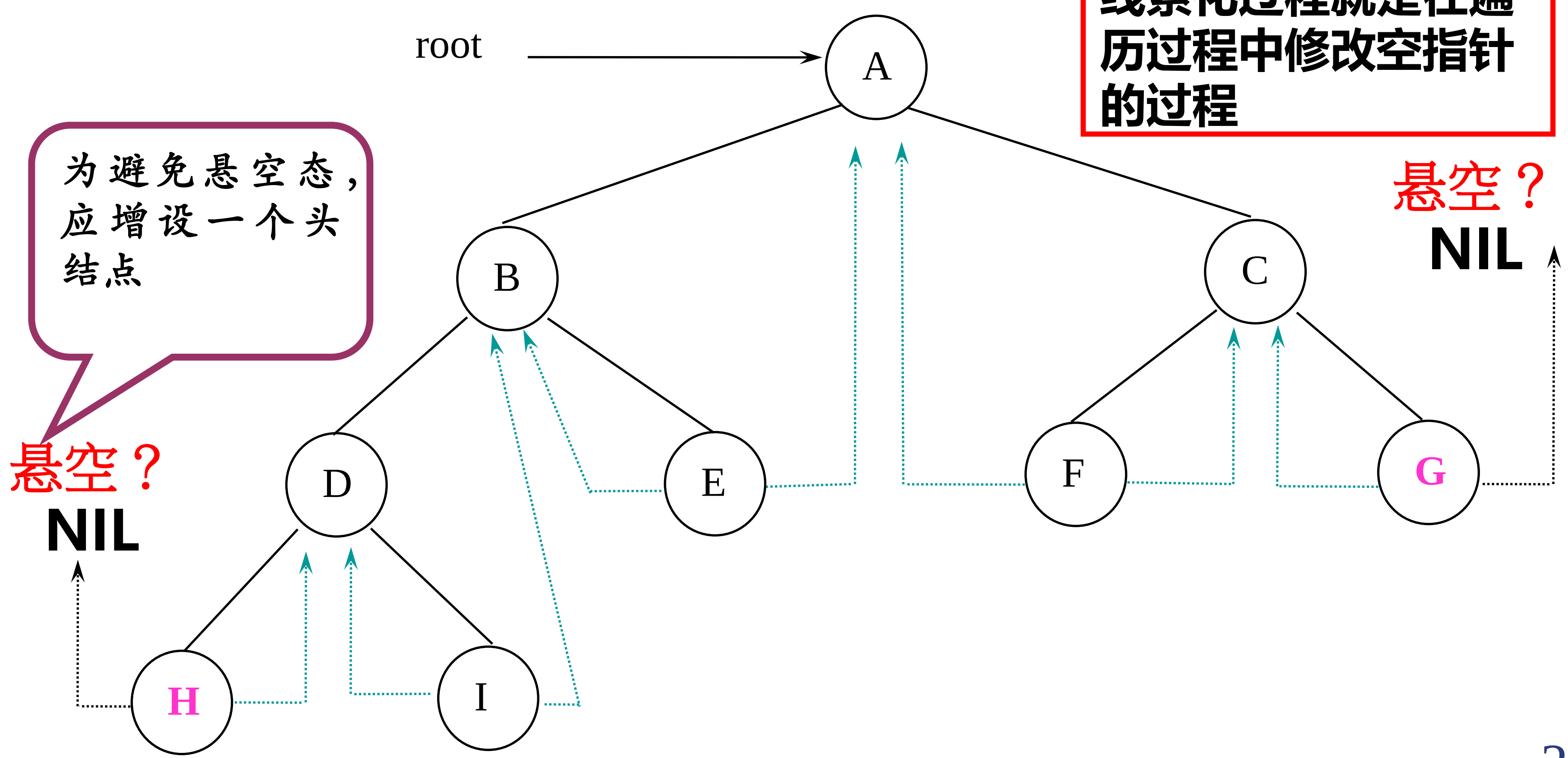
```
void InThreading(BiThrTree p){
  if (p){
     InThreading(p->lchild);//左子树线索化
     if (!p->lchild){
        p->LTag=Thread; p->lchild=pre;
     } //前驱线索
     if (!pre->rchild){
        pre->RTag=Thread;
        pre->rchild=p;//前驱右孩子指向后续（当前节点p）
     }
     pre=p;
     InThreading(p->rchild); //右子树线索化
  }
}
```

6.3.3 线索二叉树

例：画出以下二叉树对应的中序线索二叉树。

解：对该二叉树中序遍历的结果为： H, D, I, B, E, A, F, C, G
所以添加线索应当按如下路径进行：

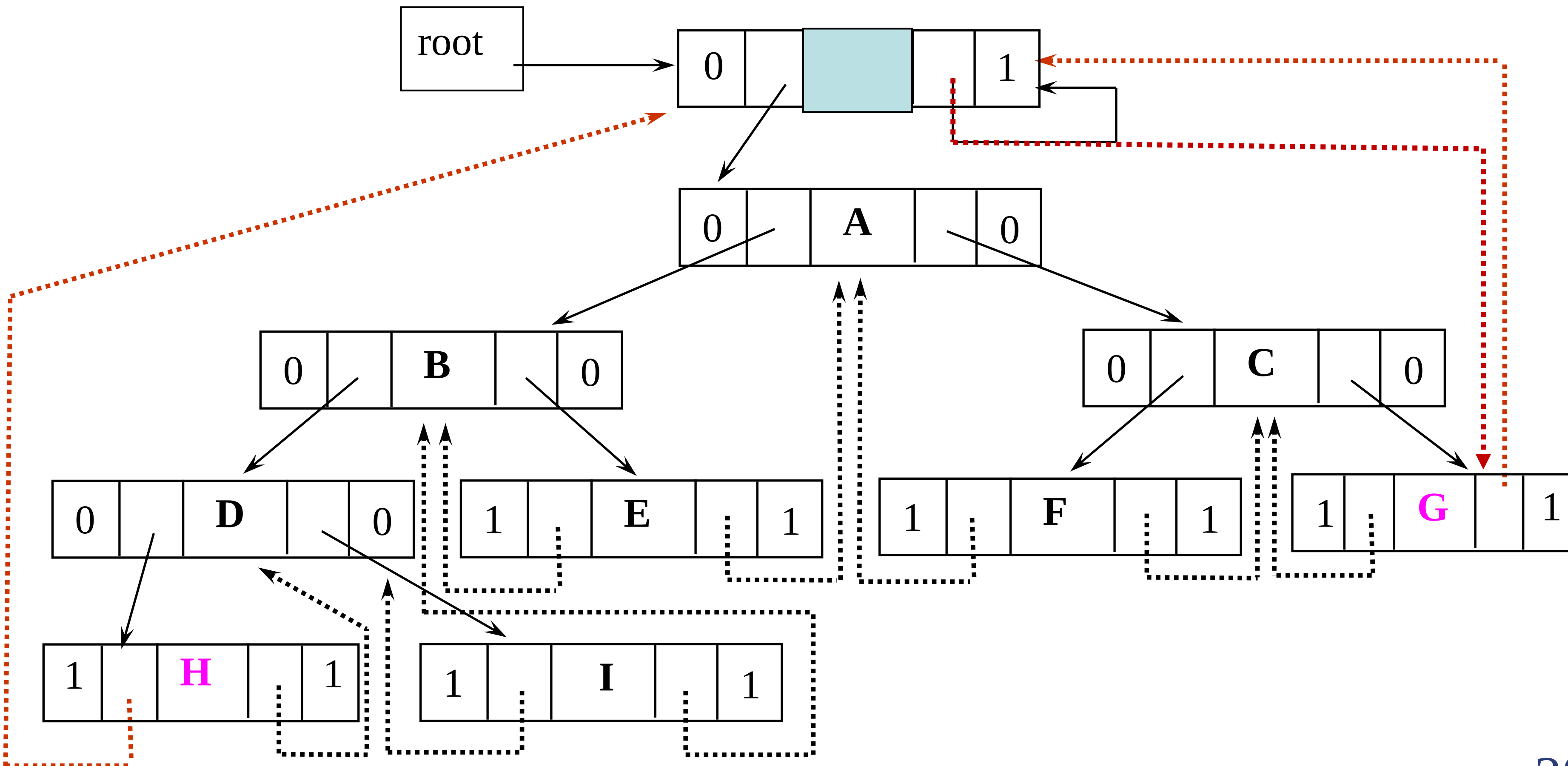
线索化过程就是在遍历过程中修改空指针的过程



6.3.3 线索二叉树

对应的中序线索二叉树存储结构如图所示：

注：此图中序遍历结果为：H, D, I, B, E, A, F, C, G



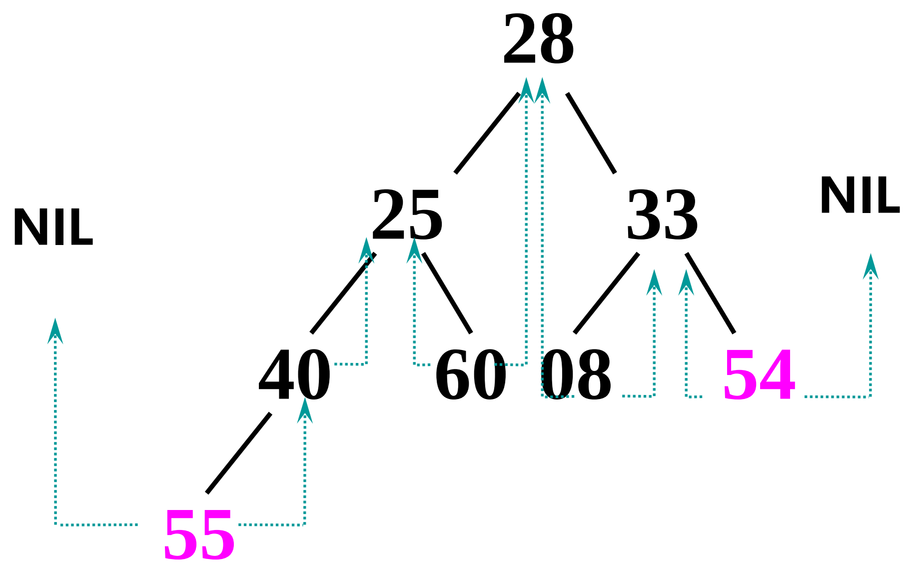
6.3.3 线索二叉树

二叉树的线索化，加头节点

```
Status lnOrderThreading(BiThrTree & Thrt,  BiThrTree T) {
    if (!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode))))
        exit(OVERFLOW) ;
    Thrt->LTag = Link; Thrt->RTag = Thread;
    Thrt ->rchild = Thrt;
    if (!T) Thrt ->lchild = Thrt;
    else {
        Thrt->lchild = T; pre = Thrt;
        InThreadrng(T) ;
        pre->rchild = Thrt; pre->RTag = Thread;//最后一个节点线索化
        Thrt-> rchild = pre;
    }
    return OK;
}
```

6.3.3 线索二叉树

例：给定如图所示二叉树T，画出与其对应中序线索二叉树。



解:因为中序遍历序列是：55 40 25 60 28 08 33 54

对应线索树应当按此规律连线，即在原二叉树中添加虚线。

6.3.3 线索二叉树

对于线索二叉树的遍历，只要找到序列中的第一个结点，然后依次访问结点的后继直到后继为空为止。（因为建立线索时已遍历一次，相当于线性化了！）

难点：在线索化二叉树中，并不是每个结点都能直接找到其后继的，当标志为0时，则需要通过一定运算才能找到它的后继。

6.3.3 线索二叉树

以中序线索二叉树为例：

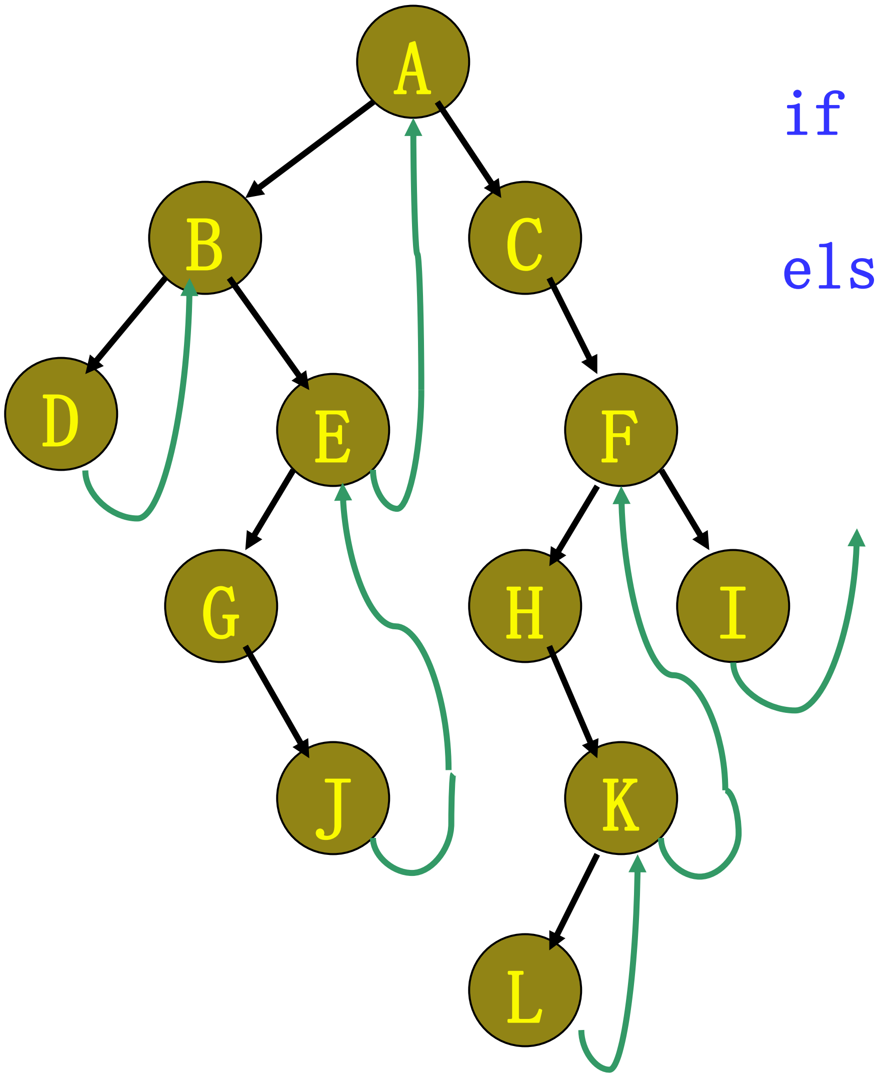
当RTag=1时，直接后继指针就在其rchild域内；

当RTag=0时，直接后继是当前结点右子树最左下方的结点；

请注意中序遍历规则是LDR，先左再根再右

6.3.3 线索二叉树

<u>寻找当前结点在中序下的后继</u>



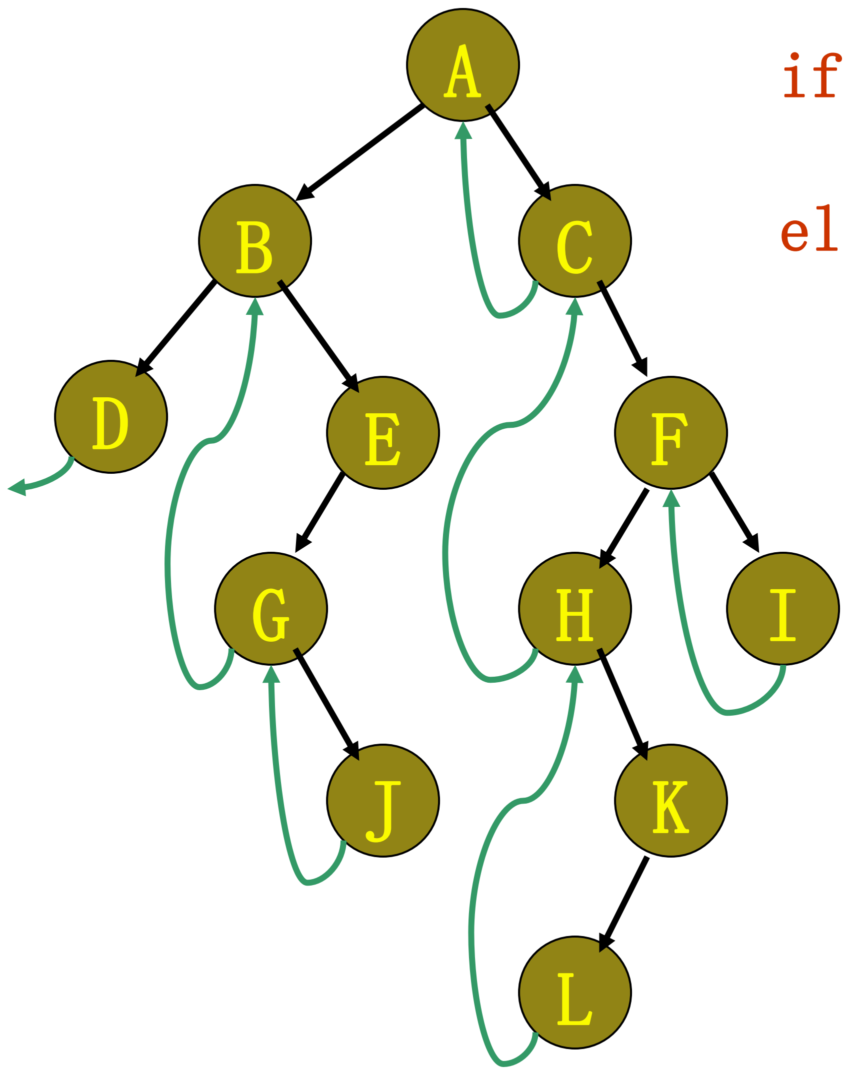
if (current→RTag ==Thread)
　　后继为 current→rchild
else //current→RTag ==Link
　　后继为当前结点右子树
　　　的中序下的第一个结点

中序后继线索二叉树

DBGJEACHLKFI

6.3.3 线索二叉树

<u>寻找当前结点在中序下的前驱</u>



if (current→LTag ==Thread)

前驱为current→lchild

else //current→LTag ==Link

前驱为当前结点左子树的中序下的最后一个结点

中序前驱线索二叉树

中序序列：DBGJEACHLKFI

6.3.3 线索二叉树

中序线索二叉树遍历步骤（算法6.5）：

有后继找后继，无后继找右子树的最左子孙

1）设置一个搜索指针p;

2）先寻找中序遍历首结点（即最左下角结点），方法是：

当LTag=0时（表示有左孩子），p=p->lchild; 直LTag=1（无左孩子，已到最左下角）；首先访问p->data;

6.3.3 线索二叉树

中序线索二叉树遍历步骤（算法6.5）：

有后继找后继，无后继找右子树的最左子孙

3）接着进入该结点的右子树，检查RTag 和p->rchild；

4） 若该结点的RTag=1(表示有后继线索)，则p=p->rchild；访问p->data ;并重复4），直到后继结点的RTag==0;

5）当RTag==0时（表示有右孩子），此时应当从该结点的右孩子开始(p=p->rchild）查找左下角的子孙结点；即回到2）

6.3.3 线索二叉树

遍历中序线索二叉树(带头结点)

```
Status InOrderTraverse(BiThrTree T,Status(* Visit)(TElemType e)){
    p=T->lchild;
    while(p!= T){
        while(p->LTag==Link) p=p->lchild;
        Visit(p->data);
        while(p->RTag==Thread && p->rchild!=T){
            p=p->rchild; Visit(p->data);
        }
        p=p->rchild;
    }
    return OK;
}
```

6.3.3 线索二叉树

遍历中序线索二叉树(不带头结点)

```
void inorder1_Thr(BiThrTree T){
    BiThrTree p=T;
    while (p) {
        while(p->LTag==Link) p=p->lchild;
        printf("%c",p->data);
        while(p->RTag==Thread && p->rchild) {
            p=p->rchild;
            printf("%c",p->data);
        }
        p=p->rchild;
    }
}
```

6.3.3 线索二叉树

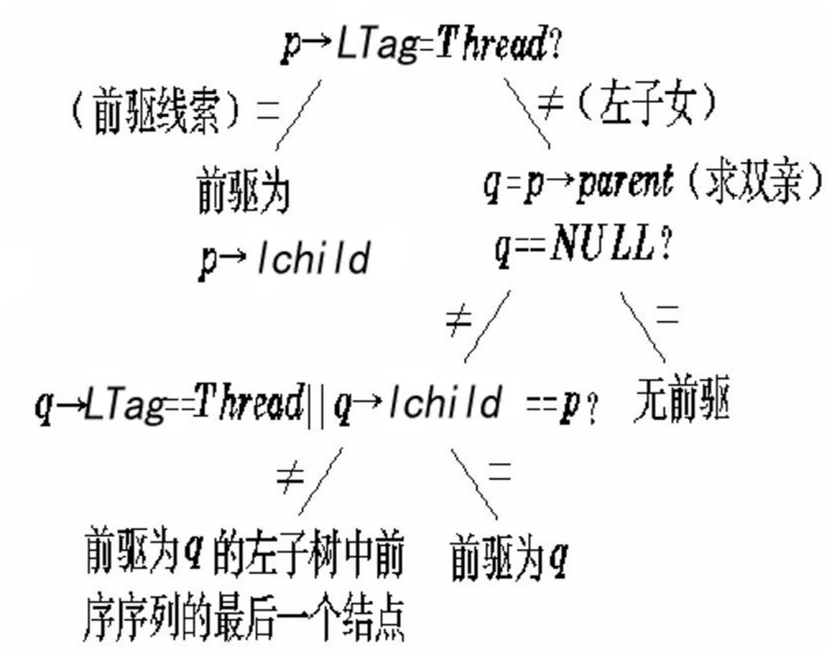
遍历中序线索二叉树(不带头结点)

```
void inorder2_Thr(BiThrTree T) {
   BiThrTree p=T;
   while (p->LTag==Link) p=p->lchild;
   printf("%c",p->data);
   while (p->rchild) {
      if(p->RTag==Link) {
          p=p->rchild;
          while(p->LTag==Link) p=p->lchild;
      }
    else p=p->rchild;
    printf("%c",p->data);
   }
}
```

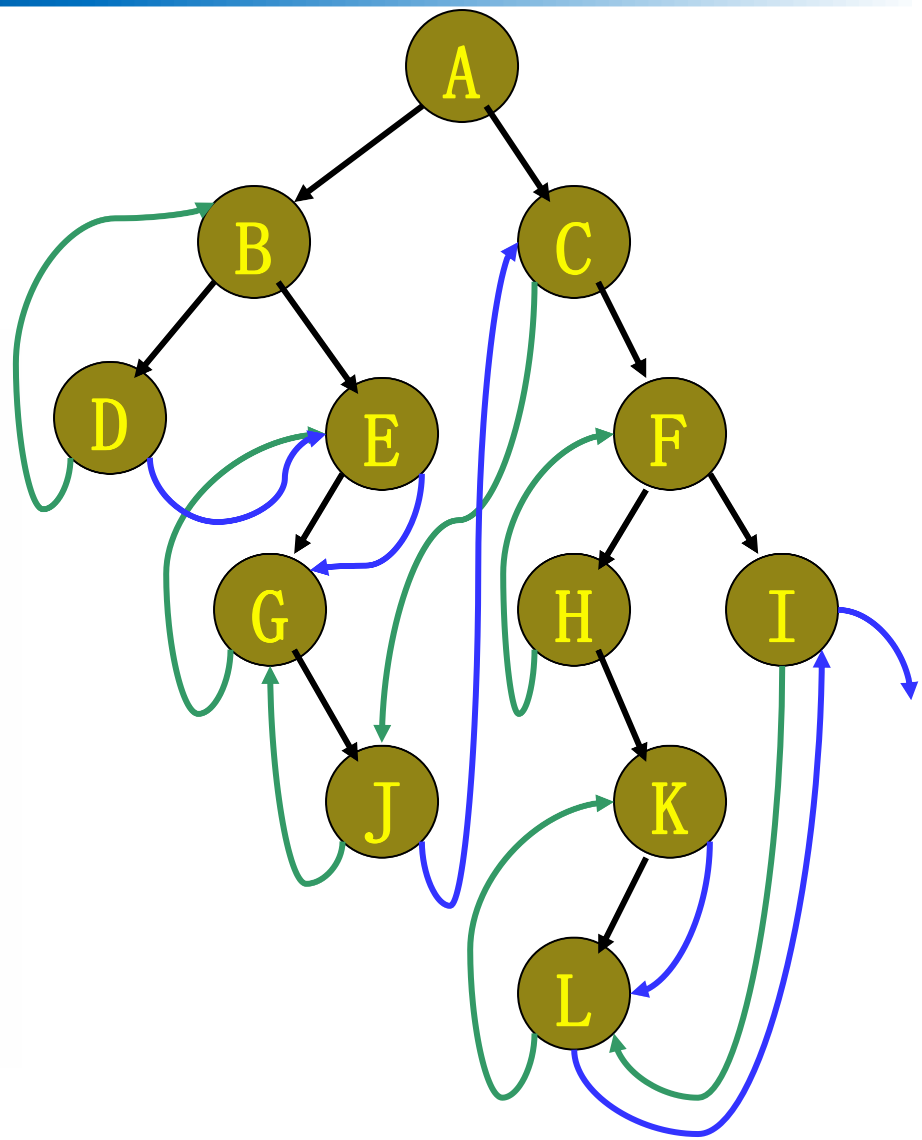
6.3.3 线索二叉树

先序线索二叉树

ABDEGJCFHKLI



(b) 求结点p的前驱



6.3.3 线索二叉树

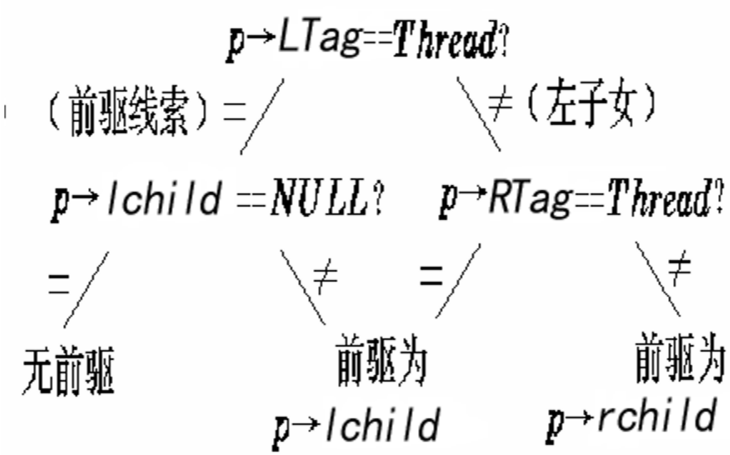
遍历先序线索二叉树(不带头结点)

```
void preorder_Thr(BiThrTree T) {
    BiThrTree p=T;
    printf("%c",p->data);
    while (p->rchild) {
        if (p->LTag==Link) p=p->lchild;
        else p=p->rchild;
        printf("%c",p->data);
    }
}
```

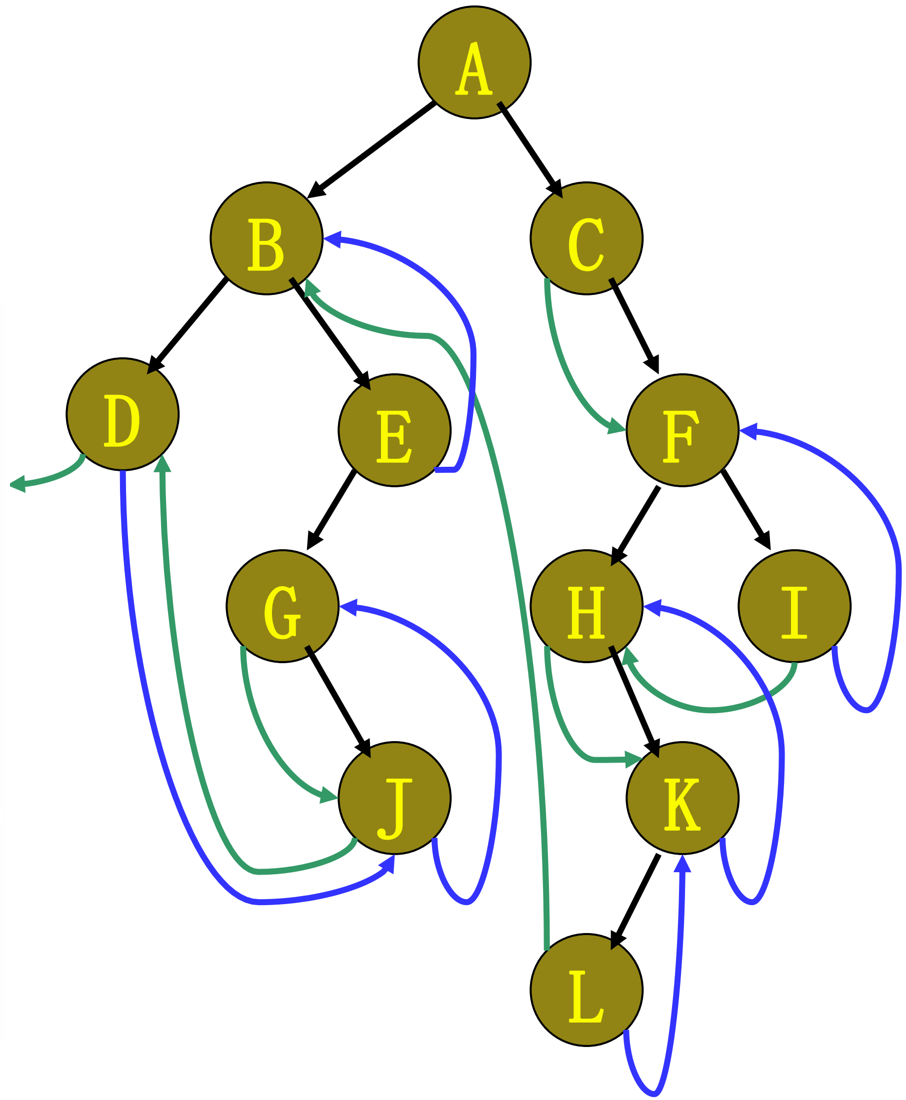
6.3.3 线索二叉树

后序线索二叉树

DJGEBLKHIFCA



(b) 求结点p的前驱



遍历后序线索二叉树(不带头结点）

```
void postorder_Thr(TriThrTree T){
    TriThrTree f, p = T;
    do{
        while (p->LTag == Link) p = p->lchild;
        if (p->RTag == Link) p = p->rchild;
    } while (p->LTag != Thread || p->RTag != Thread);
    printf("%c", p->data);
    while (p != T){
        if (p->RTag == Link){
            f = p->parent;
            if (f->RTag == Thread || p == f->rchild) p = f;
            else{
                p = f->rchild;
                do{
                    while (p->LTag == Link) p = p->lchild;
                    if (p->RTag == Link) p = p->rchild;
                } while (p->LTag != Thread || p->RTag != Thread);
            }
        }
        else p = p->rchild;
        printf("%c", p->data);
    }
}
```

6.3.3 线索二叉树

二叉树的线索化

- 将未线索过的二叉树给予线索
  ——在遍历的前提下，按照先、中、后序中的一种
- 线索化
  - 后继线索化——处理前驱结点
  - 前继线索化——处理后驱结点

6.3.3 线索二叉树

先序线索化

```
void PreThreading(BiThrTree p) {
    if (p){
        if (!p->lchild) {
            p->LTag=Thread;p->lchild=pre;
        }
     if (!p->rchild) p->RTag=Thread;
        if (pre && pre->RTag==Thread) pre->rchild=p;
     pre=p;
     if (p->LTag==Link) PreThreading(p->lchild);
     if (p->RTag==Link) PreThreading(p->rchild);
    }
}
```

6.3.3 线索二叉树

后序线索化

```
void PostThreading(TriThrTree P){
   if (P){
      PostThreading(P->lchild);
      PostThreading(P->rchild);
    if (!P->lchild) {
         P->LTag=Thread; P->lchild=pre;
      }
      if (!P->rchild) P->RTag=Thread;
    if (pre && pre->RTag==Thread) pre->rchild=P;
    pre=P;
   }
}
```

6.3.3 线索二叉树

中序线索化二叉树非递归完整算法

写字板文档

6.4 树和森林

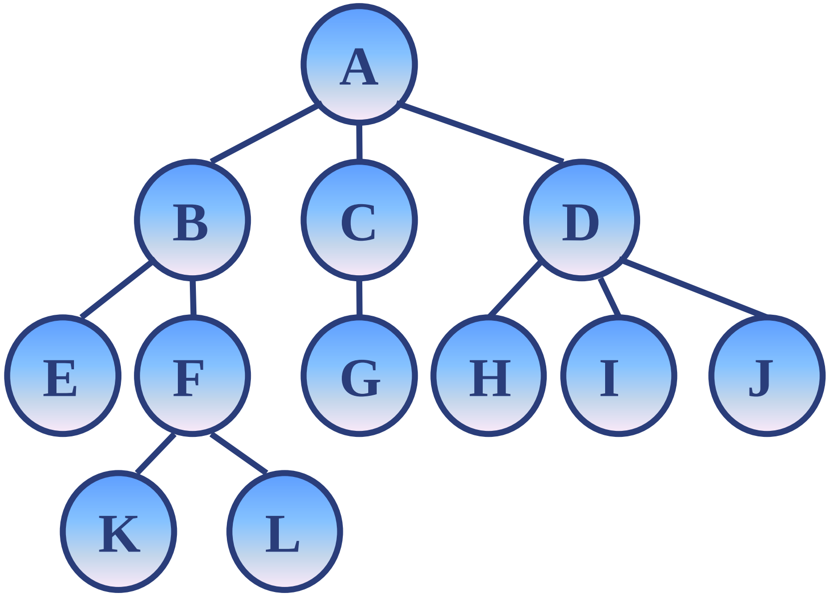
6.4.1 树和森林的存储方式

6.4.2 树和森林与二叉树的转换

6.4.3 树和森林的遍历

6.4.1 树和森林的存储结构

树结构 （除了一个称为根的结点外）每个元素都有且仅有一个直接前趋，有且仅有零个或多个直接后继。



结点的度：结点所拥有的子树的个数。
树的度：树中各结点度的最大值。

6.4.1 树和森林的存储结构

树有三种常用存储方式:

①双亲表示法

②孩子表示法

③孩子—兄弟表示法

6.4.1 树和森林的存储结构

1、双亲表示法

基本思想：用一维数组来存储树的各个结点（一般按层序存储），数组中的一个元素对应树中的一个结点，包括结点的数据信息以及该结点的双亲在数组中的下标。

data	parent

data：存储树中结点的数据信息

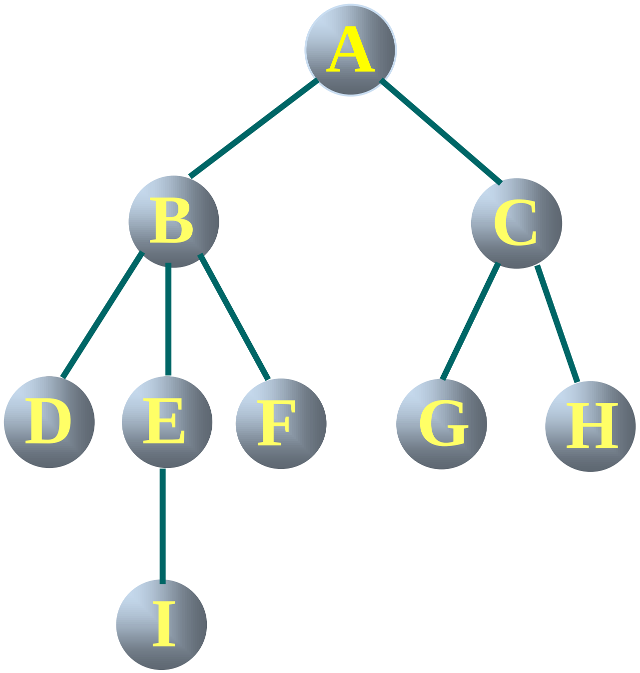
parent：存储该结点的双亲在数组中的下标

6.4.1 树和森林的存储结构

```
#define MAX_TREE_SIZE  100
type struct PTNode          //结点结构
{
    TEleType      data;      //数据域
    int           parent;    //双亲在数组中的下标
} PTNode;
Typedef struct {             //树结构
    PTNode   nodes[MAX_TREE_SIZE]
    int      r,n             //根的位置和结点数
}PTree
```

树的双亲表示法实质上是一个静态链表。

6.4.1 树和森林的存储结构



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

找某一结点的双亲，按照该结点的双亲下表即可找到。但求某一结点的孩子，要遍历整个结构。

6.4.1 树和森林的存储结构

2、孩子表示法

链表中的每个结点包括一个数据域和多个指针域，每个指针域指向该结点的一个孩子结点。

方案一：指针域的个数等于树的度

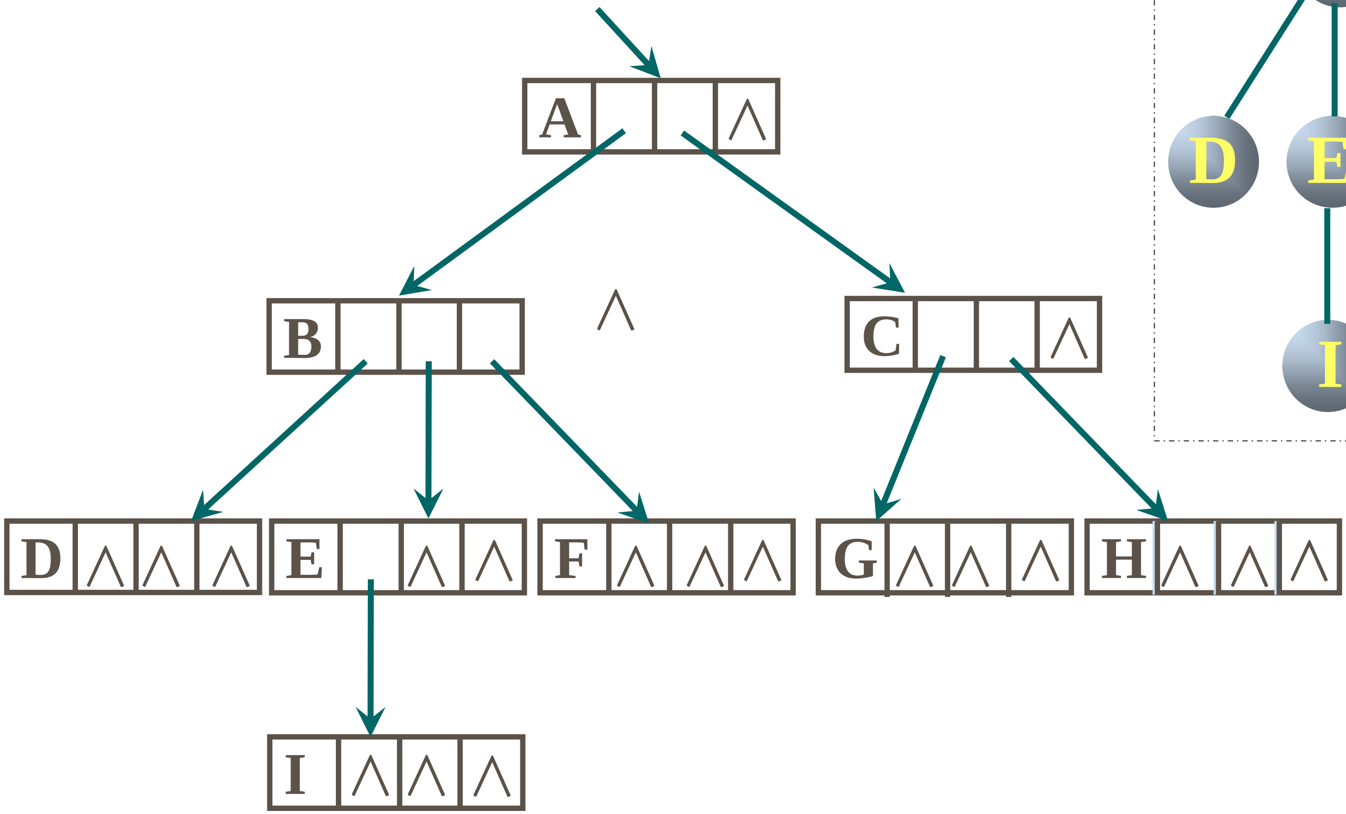
data	**child1**	**child2**	**……**	**childd**

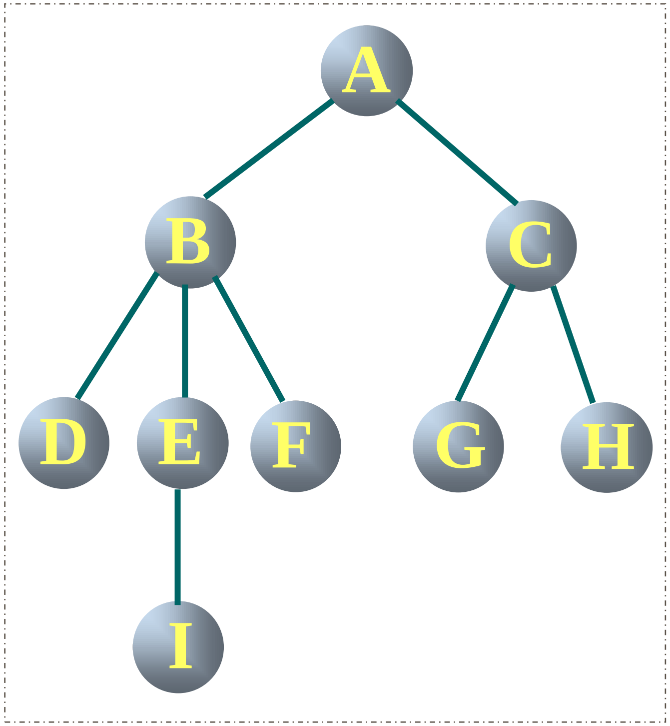
其中：**data**：数据域，存放该结点的数据信息；

child1~childd：指针域，指向该结点的孩子。

6.4.1 树和森林的存储结构

缺点：浪费空间





6.4.1 树和森林的存储结构

方案二： 指针域的个数等于该结点的度

data	degree	child1	child2	……	childd

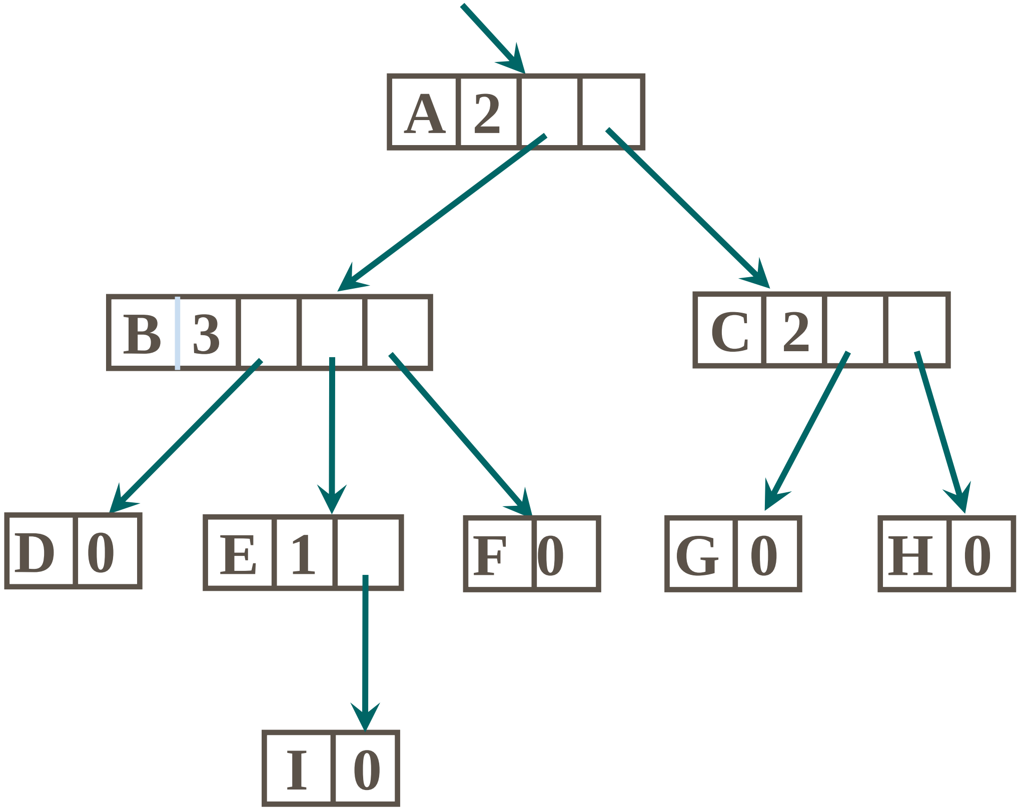
其中：data：数据域，存放该结点的数据信息；

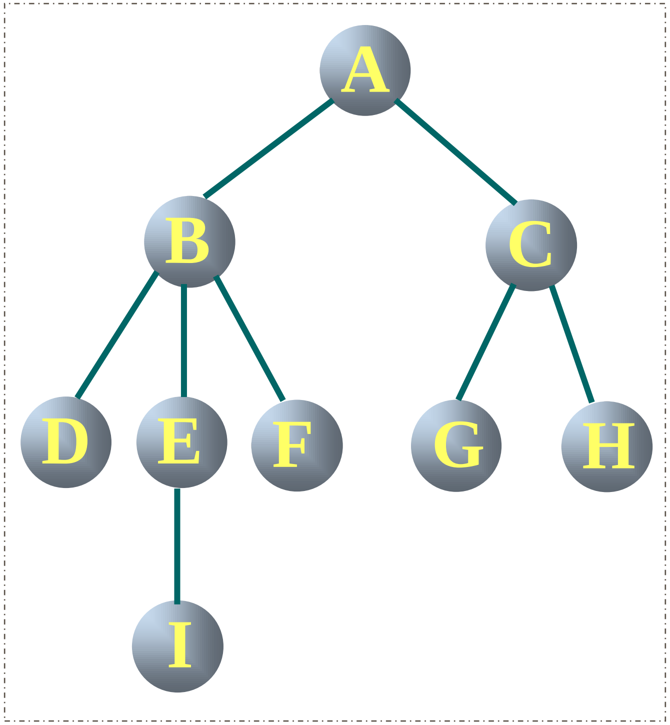
degree：度域，存放该结点的度；

child1~childd：指针域，指向该结点的孩子。

6.4.1 树和森林的存储结构

缺点：结点结构不一致





6.4.1 树和森林的存储结构

多重链表（标准存储结构）

- 定长结构 （n为树的度）指针利用率不高

data	$child_1$	$child_2$	$child_3$	....	$child_n$

- 不定长结构 d为结点的度，节省空间，但算法复杂

data	*d*	$child_2$	$child_3$	....	$child_d$

- 一般采用定长结构
  - ➢如有n个结点，树的度为k，则共有n*k个指针域，只有n-1个指针域被利用，而未利用的指针域为：n*k-(n-1) =n(k-1)+1，未利用率为：(n(k-1)+1)/nk ＞ n(k-1)/nk=(k-1)/k

    二次树：1/2 ； 三次树：2/3 ； 四次树：3/4
  - ➢树的度越高，未利用率越高，由于二叉树的利用率较其他树高，因此用二叉树。

6.4.1 树和森林的存储结构

方案三：将结点的所有孩子放在一起，构成线性表。

基本思想：

把每个结点的孩子排列起来，看成是一个线性表，且以单链表存储，则*n*个结点共有 ***n* 个孩子链表**。这 *n* 个单链表共有 *n* 个头指针，这 *n* 个头指针又组成了一个线性表，构成孩子链表的**表头数组**。

6.4.1 树和森林的存储结构

孩子结点

child	next

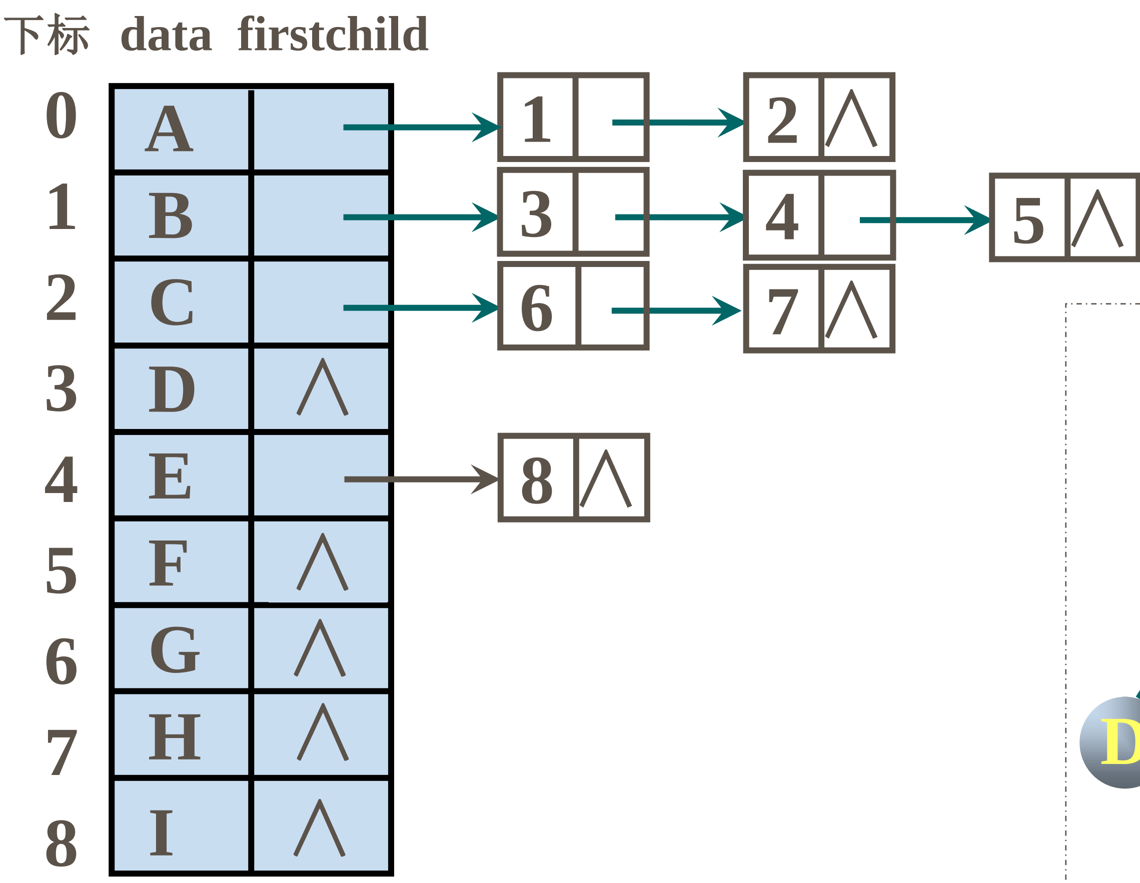
```
typedef struct CTNode
{    int        child;
     struct  CTNode *next;
};
```

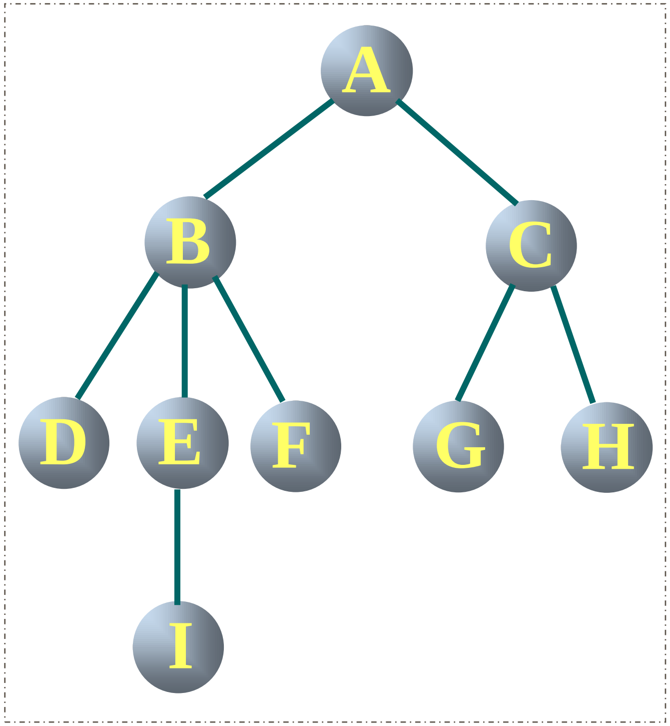
表头结点

data	firstchild

```
Typedef  struct
{   TElemType   data;
    ChildPtr  *firstchild;
}CTBox;
typedef struct
{ CTBox   nodes[MAX_TREE_SIZE]
  int          n,r;
}CTree;
```

6.4.1 树和森林的存储结构

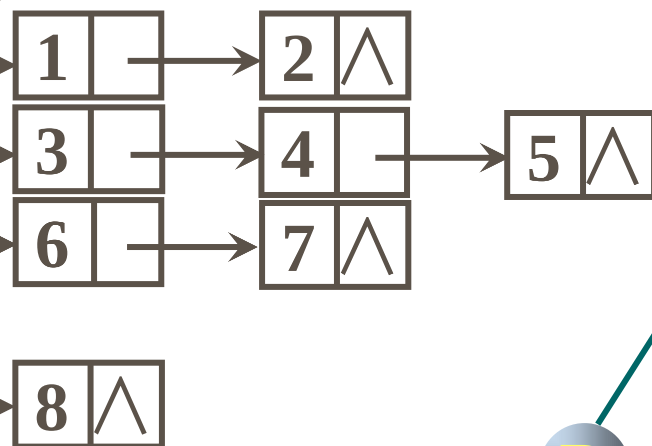


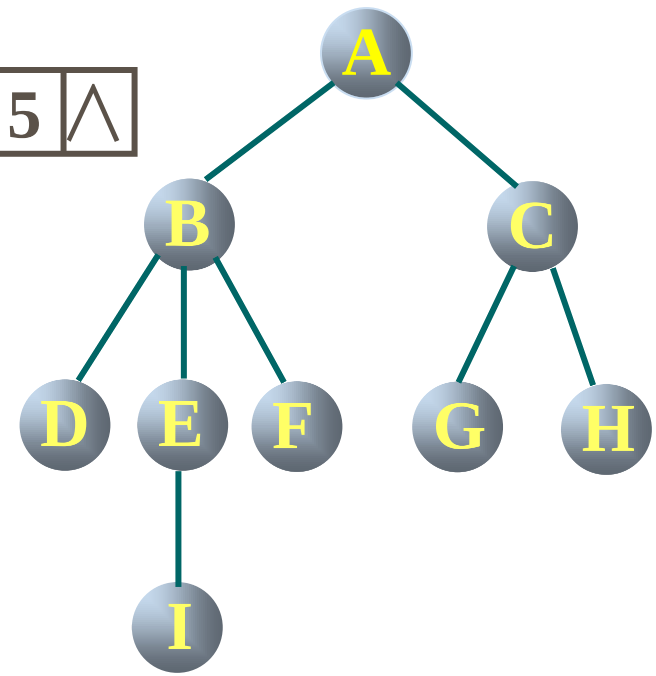


6.4.1 树和森林的存储结构

带双亲的孩子链表：将双亲表示法和孩子表示法结合

下标	para	data	firstchild
0	−1	A	
1	0	B	
2	0	C	
3	1	D	∧
4	1	E	
5	1	F	∧
6	2	G	∧
7	2	H	∧
8	4	I	∧





6.4.1 树和森林的存储结构

3、孩子兄弟表示法—又称二叉树表示法

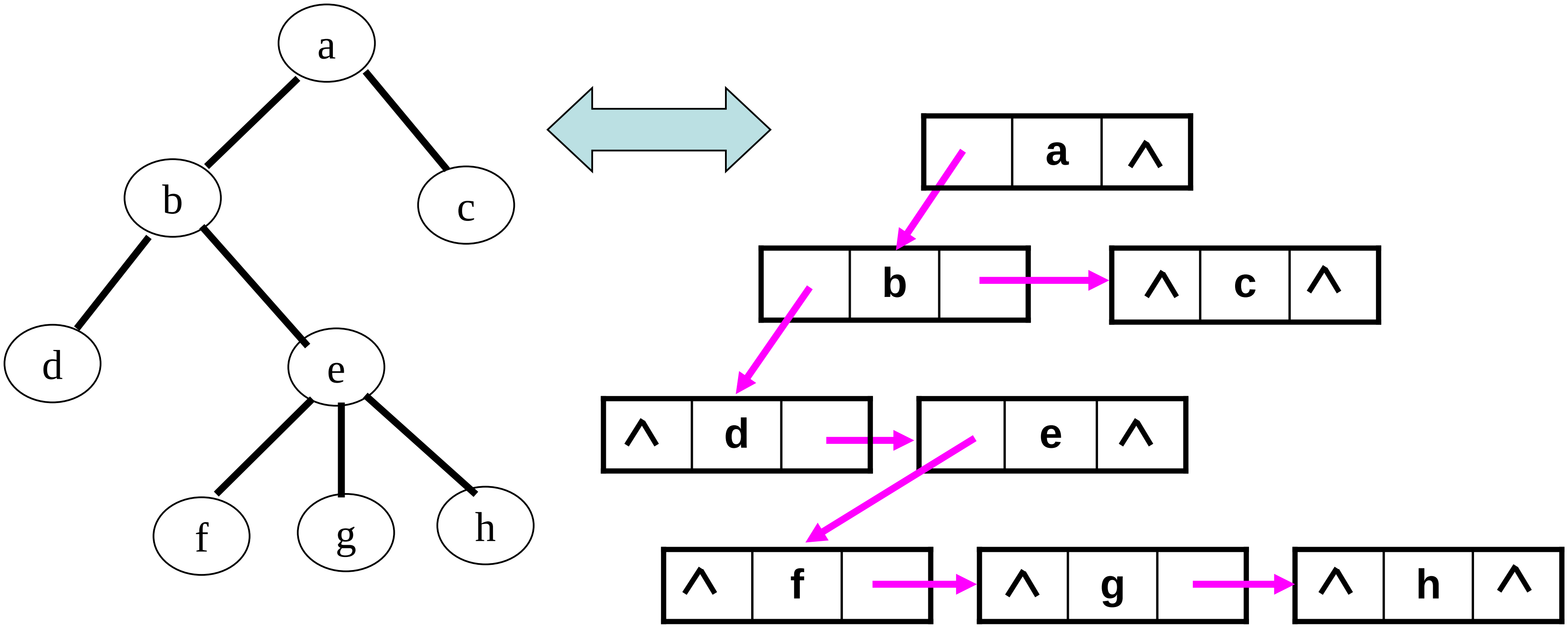
结点结构

firstchild	data	nextsibling

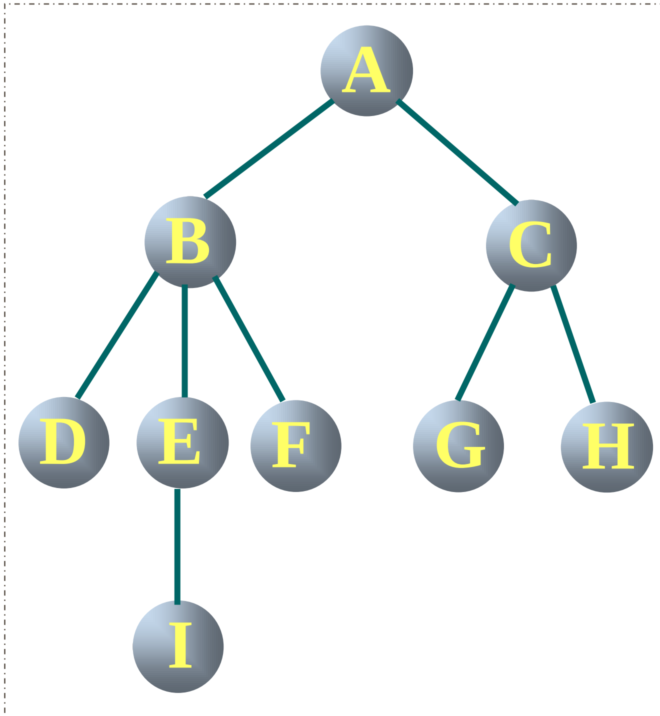
data： 数据域，存储该结点的数据信息；
firstchild： 指针域，指向该结点第一个孩子；
nextsibling： 指针域，指向该结点的右兄弟结点。

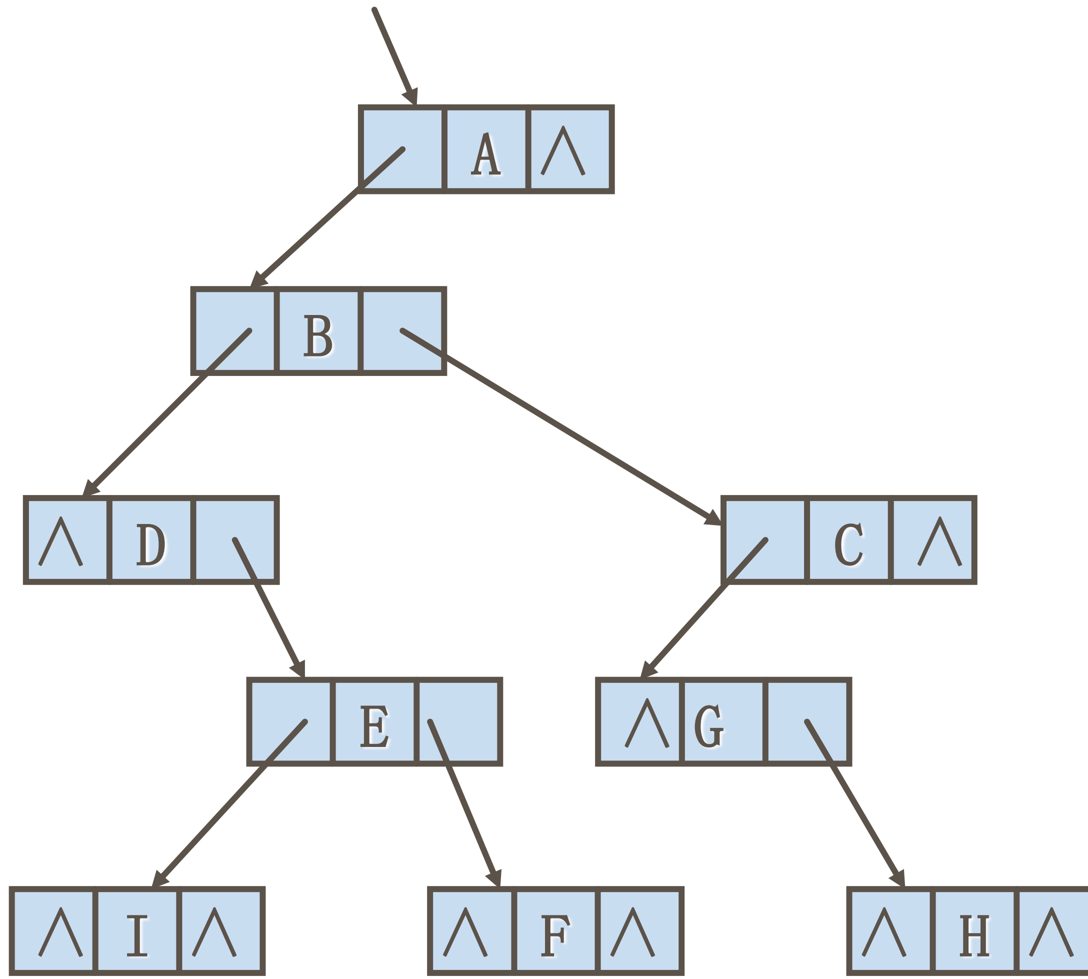
```
typedef  struct CSNode   {
{
    ElemType         data;
    struct CSNode  *firstchild, *nextsibling;
}CSNode,*CSTree;
```

6.4.1 树和森林的存储结构



6.4.1 树和森林的存储结构





正在答疑