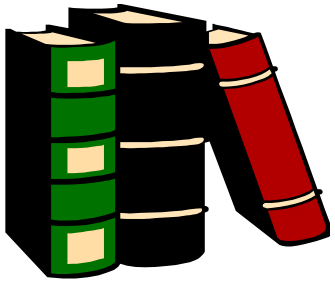
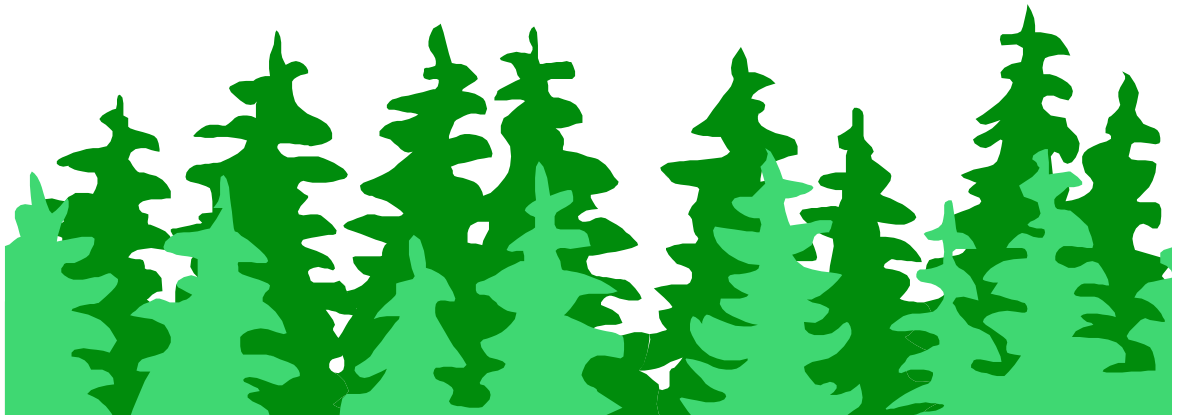
6.4 树和森林

- 树和森林的存储
- 树和森林的遍历

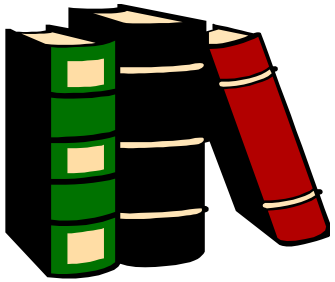




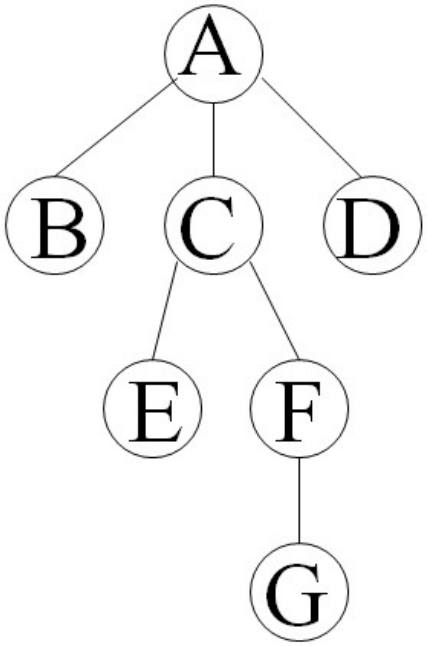
6.4.1树的存储结构

- **树的非顺序存储映像：**

1. 双亲表示法
2. 孩子表示法
3. 树的二叉链表(孩子-兄弟）存储表示法



树的双亲表示法：



r=0
n=7

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

定义：用一维数组存放树中的每一结点的值(data)和双亲位置（parent,逻辑关系）

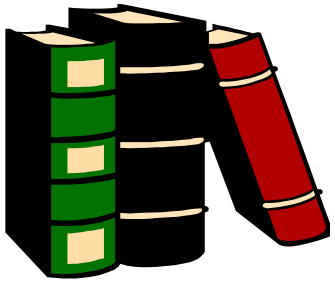
```
#define MAX_TREE_SIZE  100

typedef struct PTNode {
    ElemType  data;
    int    parent;   // 双亲位置
} PTNode;

typedef struct {
    PTNode  nodes
              [MAX_TREE_SIZE]
    int    r, n;
} PTree;
```

说明：结点存放无顺序要求，
根结点不一定存在第一个位置；
每个数组元素对应树中一个结点，
存放结点的值和双亲位置
r—根结点位置，n—树中结点个数
特点：找祖先易，找子孙难

6.4.1树的存储结构

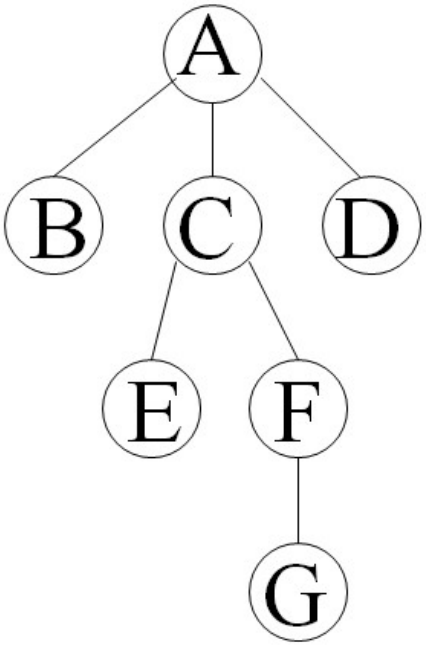
- 树的孩子表示法：

1. 定长结点的多重链表
2. 不定长结点的多重链表
3. 孩子单链表



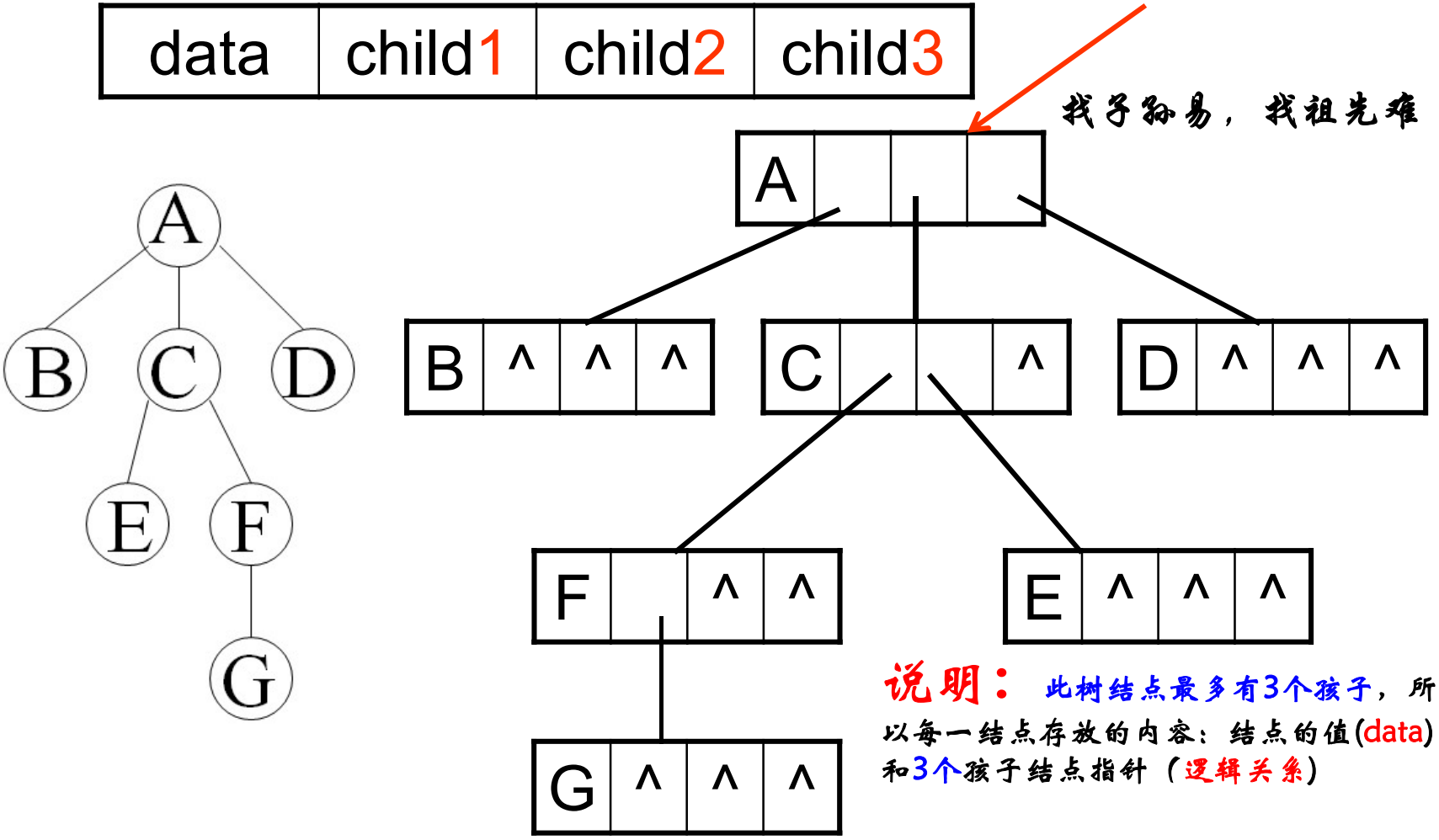
树的孩子表示法--定长结点的多重链表

data	child1	child2	…	childd



定义：链表存放树中的每一结点的值(data)和孩子结点位置（childi，逻辑关系），每个结点的孩子指针的个数＝树中孩子最多的结点的孩子个数

树的孩子表示法--定长结点的多重链表

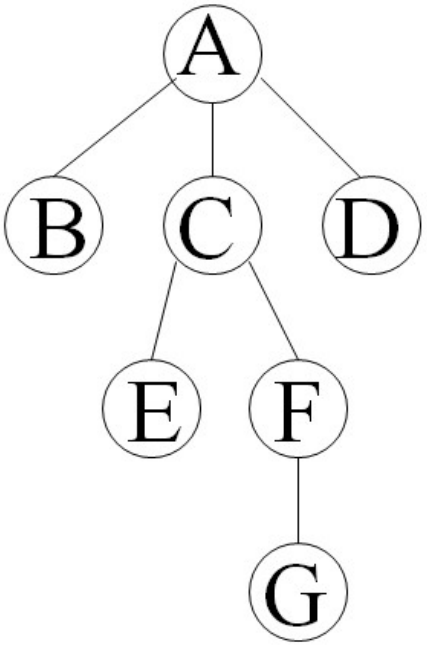


特点：结点的结构统一，若树的度为d，则结点包含一个数据域，d个孩子指针域

缺点：空指针多，浪费空间

树的孩子表示法--不定长结点的多重链表

data	degree	child1	child2	…	childd

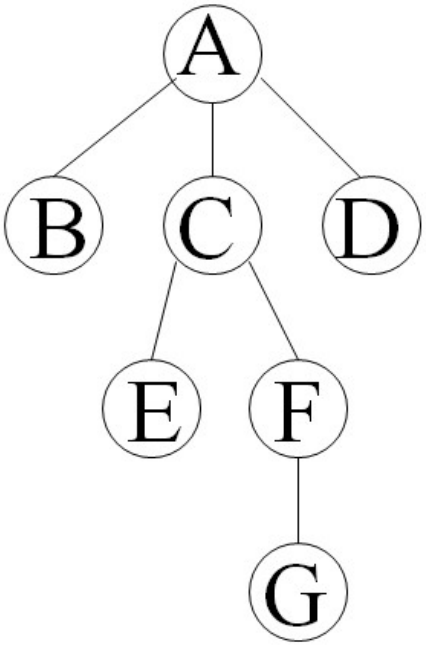


定义：链表存放树中的每一结点的值(data)和孩子结点位置（childi，逻辑关系），每个结点的孩子指针的个数＝该结点的孩子个数

树的度为d,该树的不定长结点的多重链表中结点结构有几种?

树的孩子表示法--不定长结点的多重链表

data	degree	child1	child2	…	childd



data	3	child1	child2	child3

data	2	child1	child2

data	1	child1

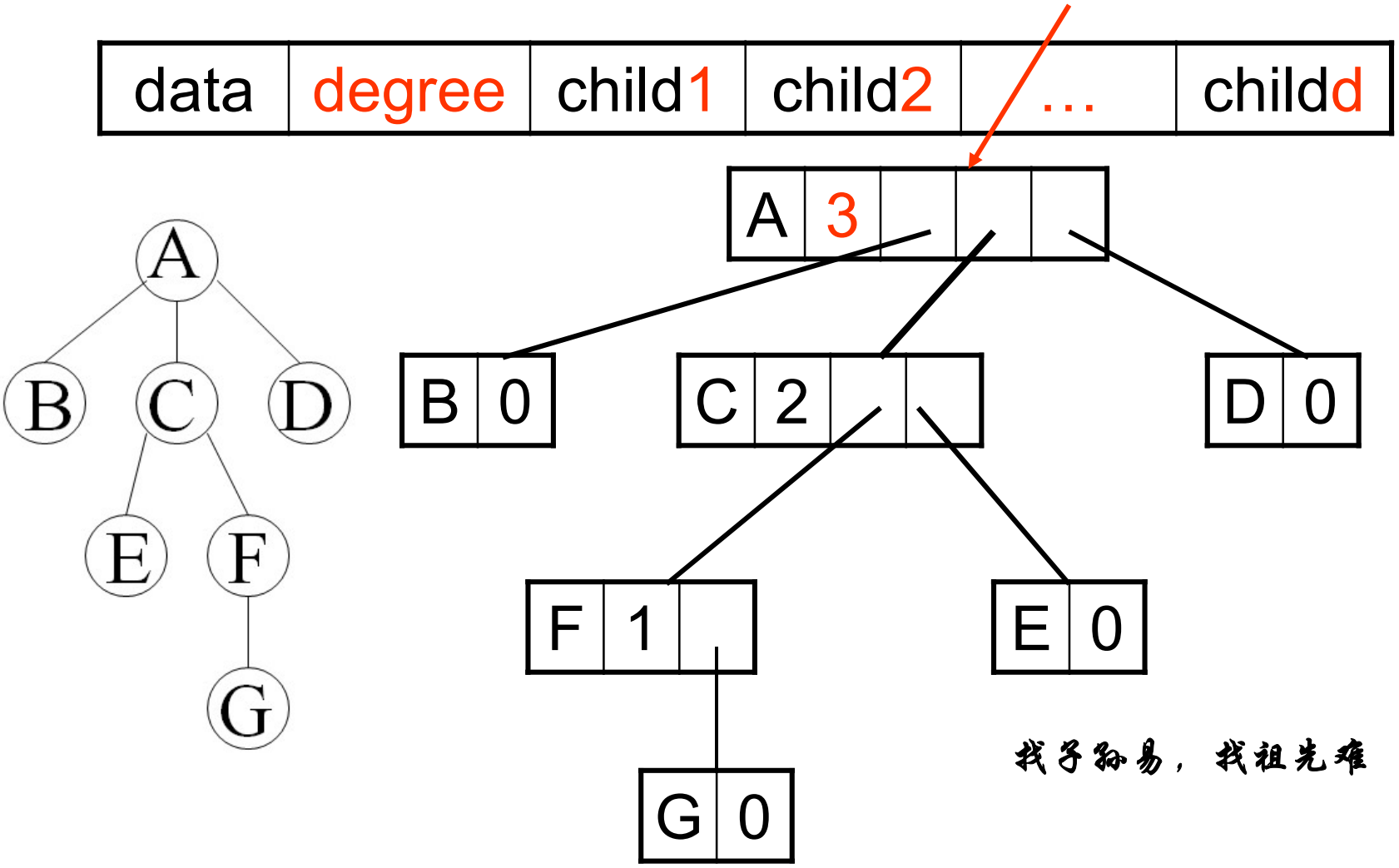
data	0

树的度为d=3,该树的不定长结点的多重链表中结点结构有4种

特点：结点的结构不统一，包含一个数据域，结点的度d，d个孩子指针域

缺点：操作较复杂

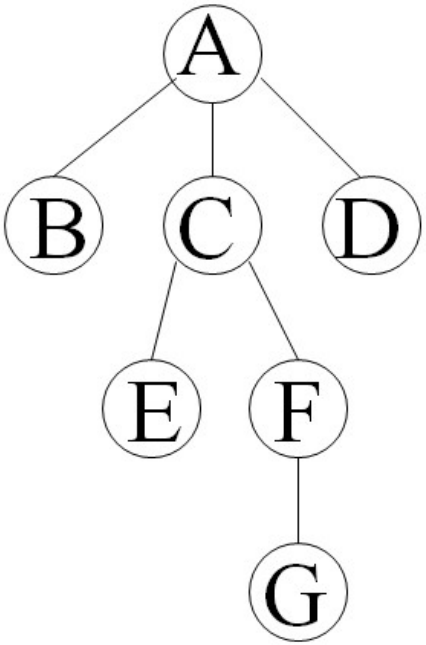
树的孩子表示法--不定长结点的多重链表

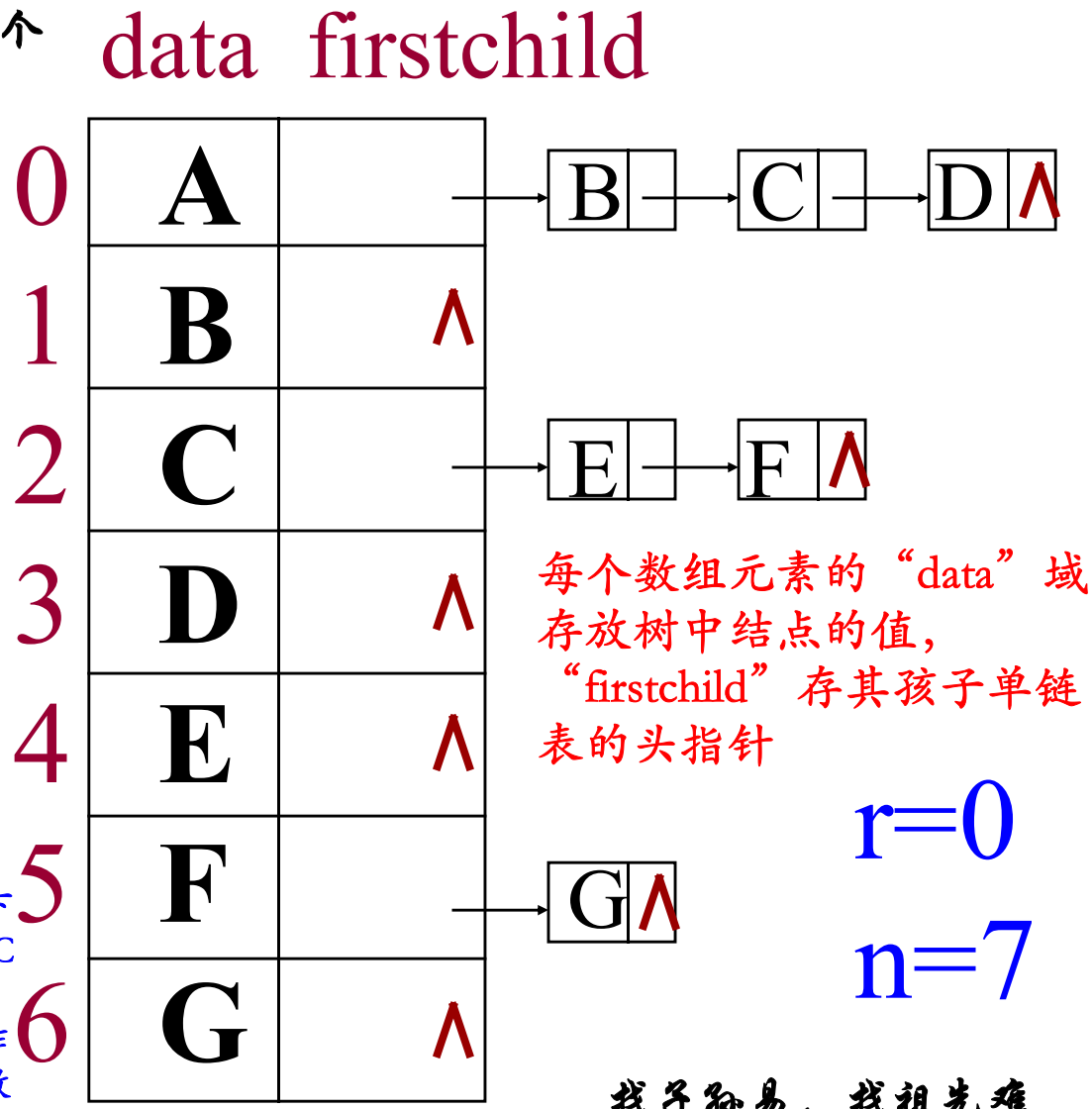


找子孙易，找祖先难

树的孩子表示法--孩子单链表表示法

将每个结点的孩子结点拉成一个单链表





每个数组元素的“data”域存放树中结点的值，“firstchild”存其孩子单链表的头指针

r=0

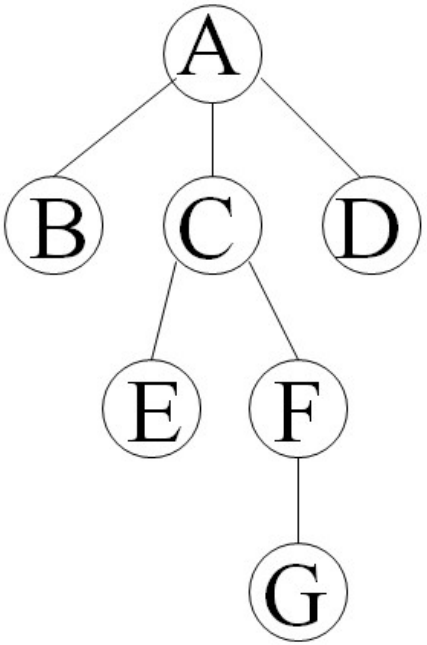
n=7

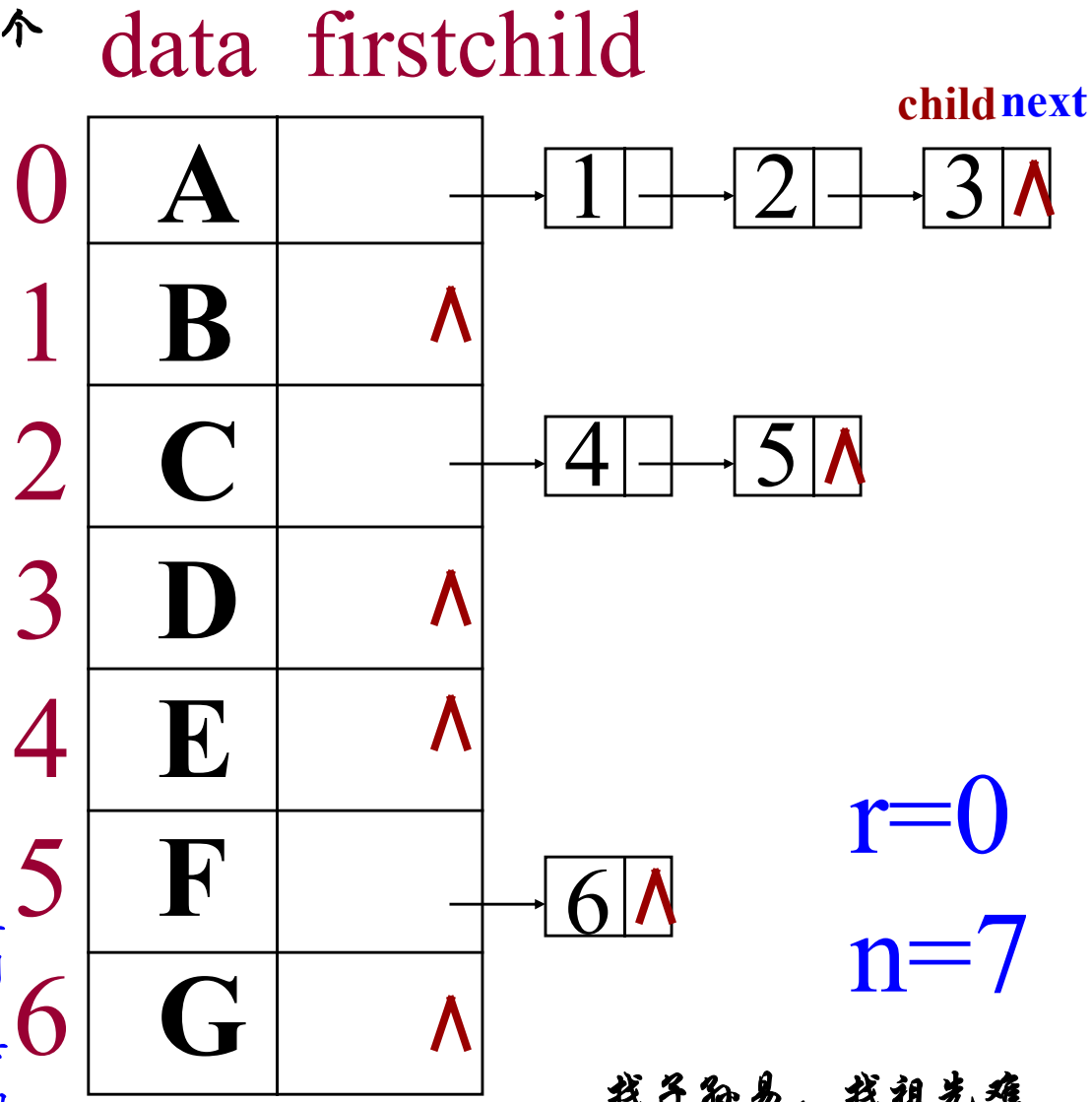
结点C在孩子表示法中存了2次：一次出现在下标为2的数组元素中，该数组元素同时保存了C的孩子单链表的头指针；一次出现在结点A的孩子单链表中。数据元素存放多次，更新操作比较麻烦，更新一个数据元素，所有保存该数据元素的地方均要更新，否则信息不一致！

找子容易，找祖先难

树的孩子表示法--孩子单链表表示法

将每个结点的孩子结点拉成一个单链表





为节省存储空间、方便更新操作和数据维护，每个数据元素只在数组中存放一次；在孩子单链表中只存放这个孩子在数组中的位置。如图所示：结点A的孩子单链表中第一个孩子结点是下标为1的数组元素（B），第二个孩子是下标为2的数组元素（C），第三个孩子是下标为3的数组元素（D）。

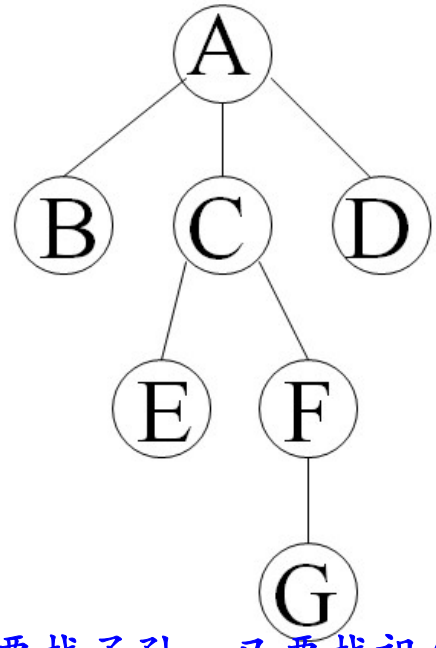
r=0

n=7

找子容易，找祖先难

树的孩子表示法--孩子单链表表示法

将每个结点的孩子结点拉成一个单链表



	data	pa	firstchild
0	A	-1	→ 1 → 2 → 3 ∧
1	B	0	∧
2	C	0	→ 4 → 5 ∧
3	D	0	∧
4	E	2	∧
5	F	2	→ 6 ∧
6	G	5	∧

r=0

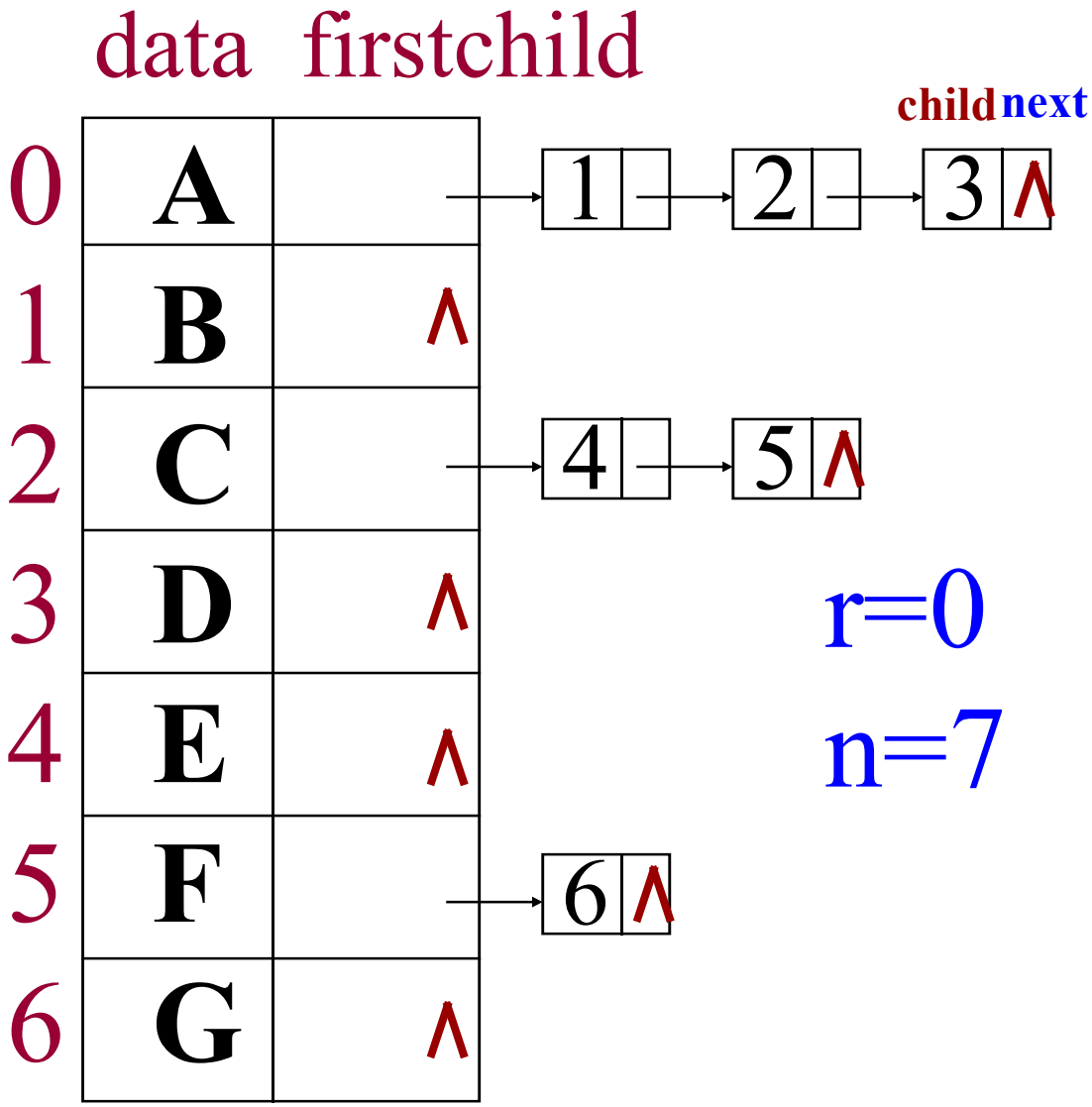
n=7

若既要找子孙，又要找祖先，可将孩子单链表和双亲表示法结合在一起

每个数组元素的“data”域存放数据元素的值，“pa”域存放双亲结点在数组中的位置，“firstchild”存其孩子单链表的头指针

将双亲表示法和孩子表示法结合

C语言的类型描述：



孩子单链表的结点结构：

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

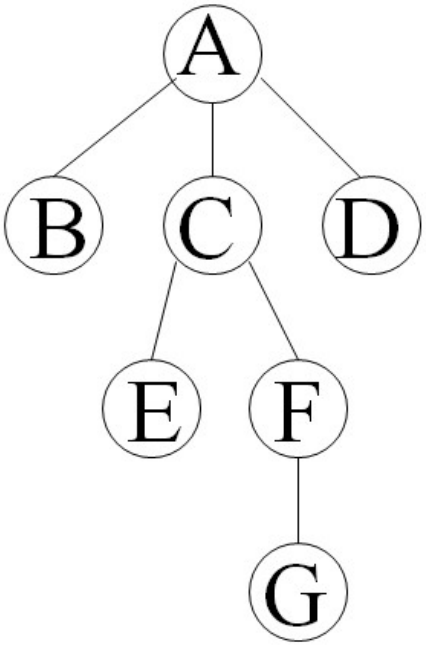
数组元素类型：

```
typedef struct {
    ElemType    data;
    ChildPtr  firstchild;
                // 孩子单链表的头指针
} CTBox;
```

树：

```
typedef struct {
    CTBox  nodes[MAX_TREE_SIZE];
    int    n, r;
        //树的结点数和根结点的位置
} CTree;
```

CTree T;



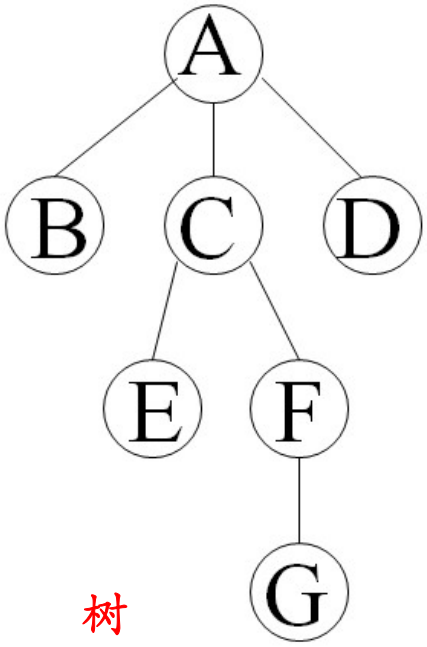
T.nodes[ ] data firstchild

	data	firstchild
0	A	→ 1 → 2 → 3 ∧
1	B	∧
2	C	→ 4 → 5 ∧
3	D	∧
4	E	∧
5	F	→ 6 ∧
6	G	∧

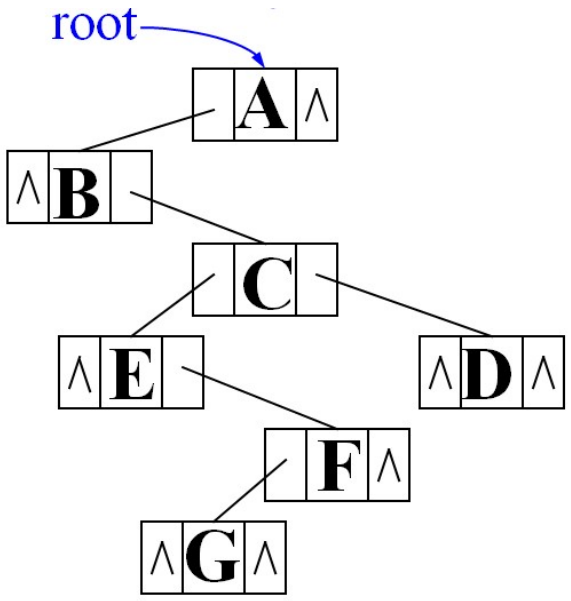
T.r=0

T.n=7

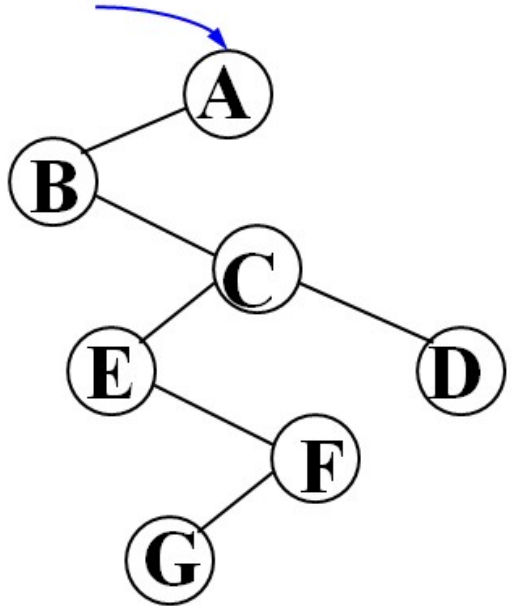
树的二叉链表（孩子-兄弟）存储表示法



树



孩子-兄弟存储表示



对应的二叉树

fc	data	nb

```
typedef struct CSNode{
    ElemType        data;
    struct CSNode
        *fc, *nb;
} CSNode, *CSTree;
```

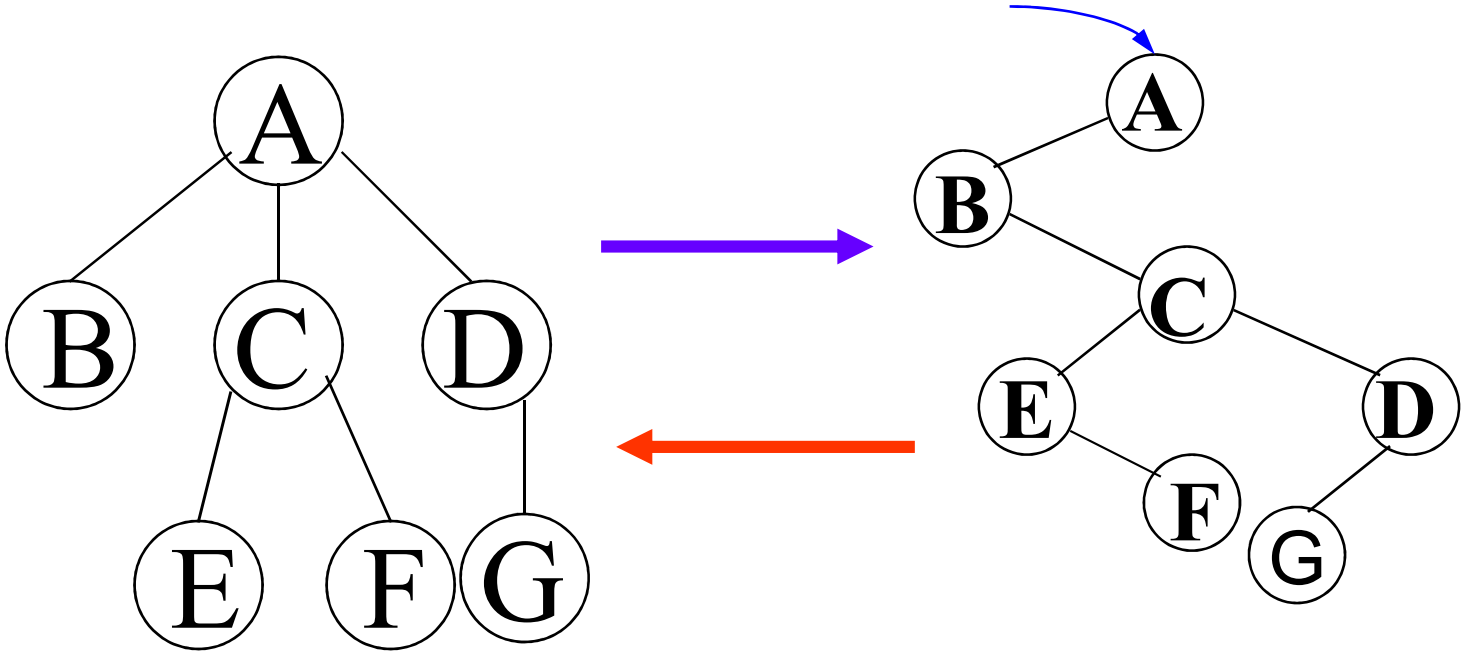
树中每个结点三部分：
数据域（data），长子指针域(fc)，
右邻兄弟指针域(nb)

树和二叉树的转换

- 树以孩子兄弟表示法存，相当于将树转换成二叉树，但此二叉树根结点无右子树
- 好处：借助二叉树的操作实现树的操作
- 要求：

➢ 掌握树和二叉树的转换

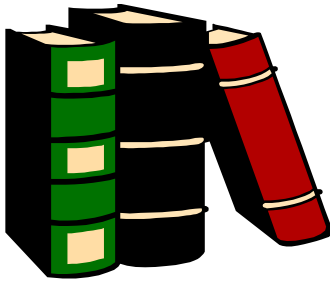
➢ 利用二叉树的操作实现树的操作



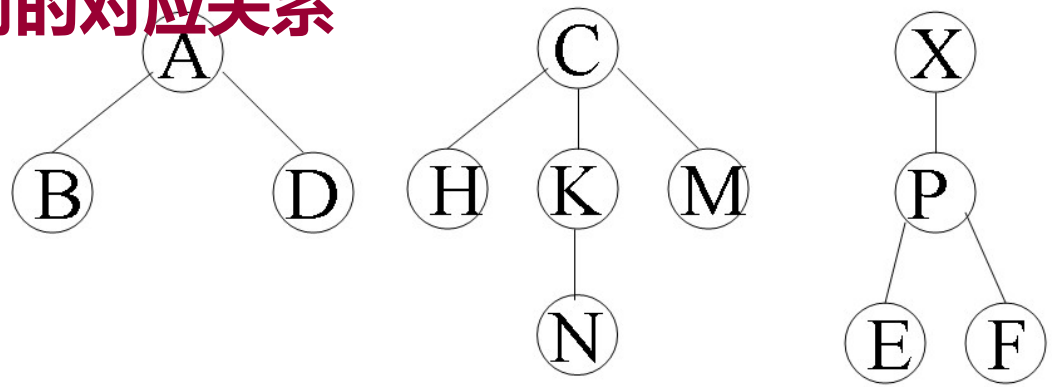
树和二叉树的转换示例

6.4.2 森林与二叉树的转换

- 树采用二叉链表(孩子-兄弟）存储表示法，转换成二叉树
- 森林由多棵树组成：$F=(T_1, T_2, \ldots, T_n)$;，将其每棵树转换成二叉树$BT_1, BT_2, \ldots, BT_n$;
- 每棵二叉树BT的根的右子树皆为空树，从BT_n开始依次将其根结点链为前一棵二叉树的根的右孩子
- 将森林转换成一棵二叉树，森林的操作可借助二叉树的操作完成

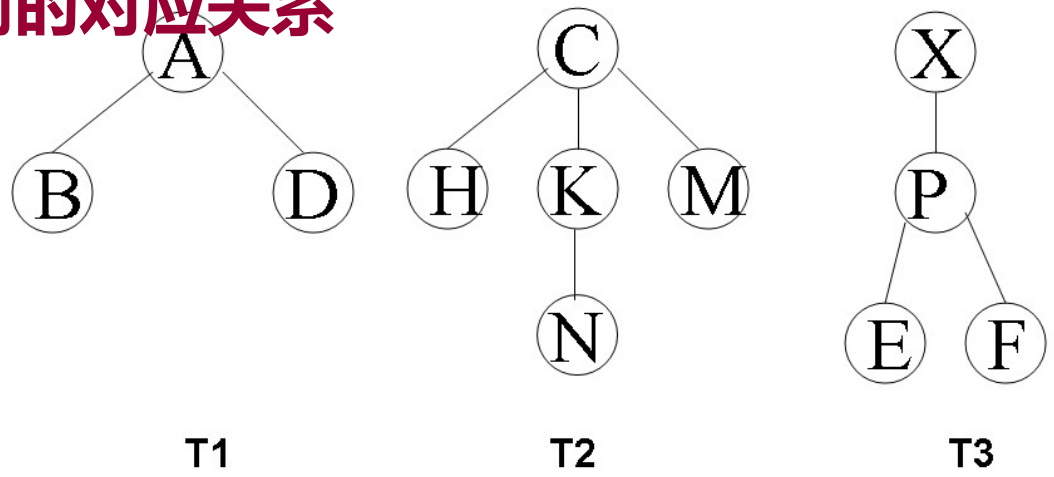


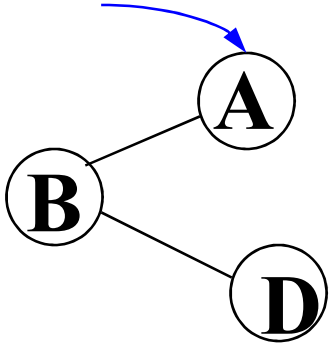
森林和二叉树的对应关系



T1 T2 T3

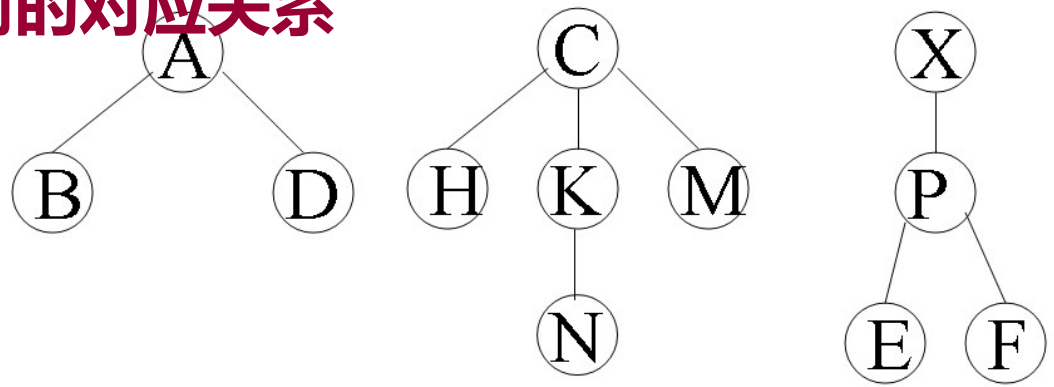
森林和二叉树的对应关系



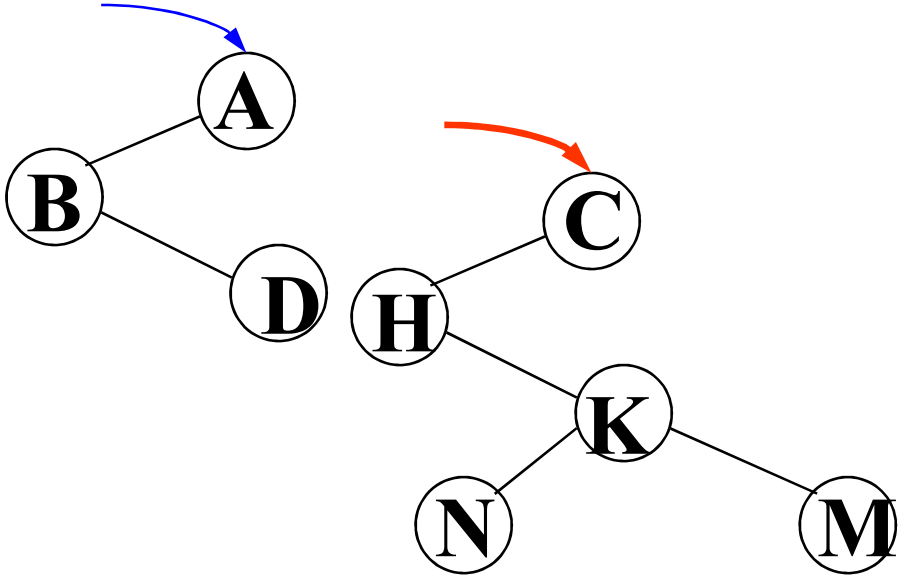


BT1

森林和二叉树的对应关系

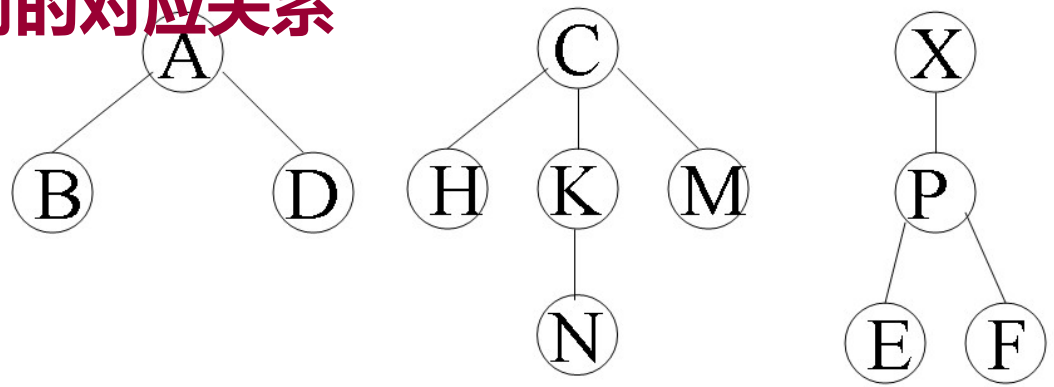


T1　　T2　　T3

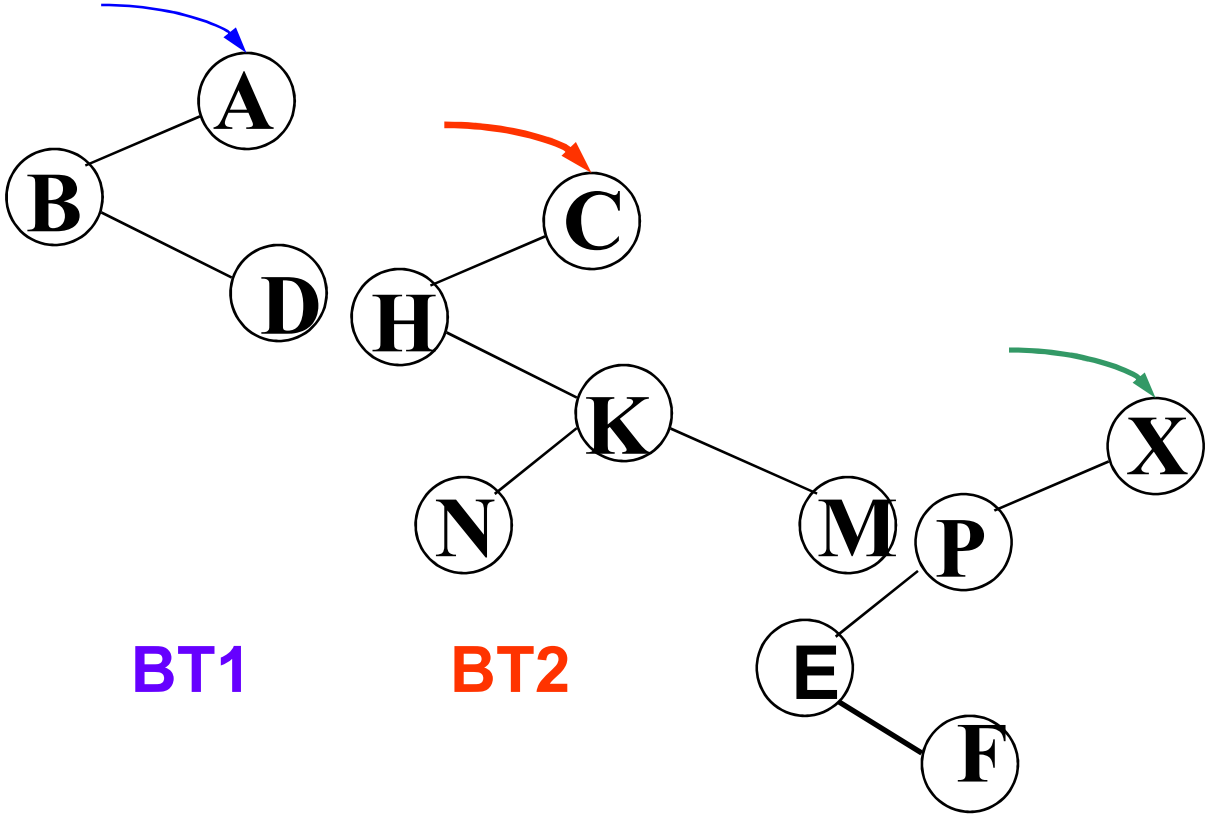


BT1　　BT2

森林和二叉树的对应关系

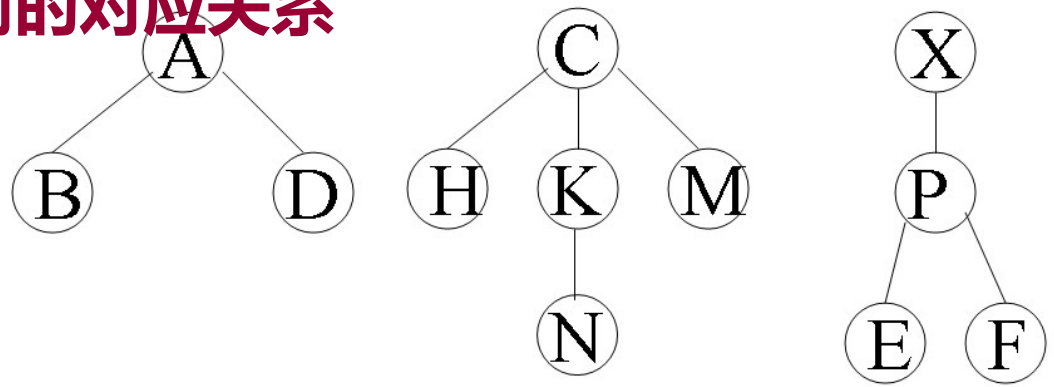


T1　　T2　　T3

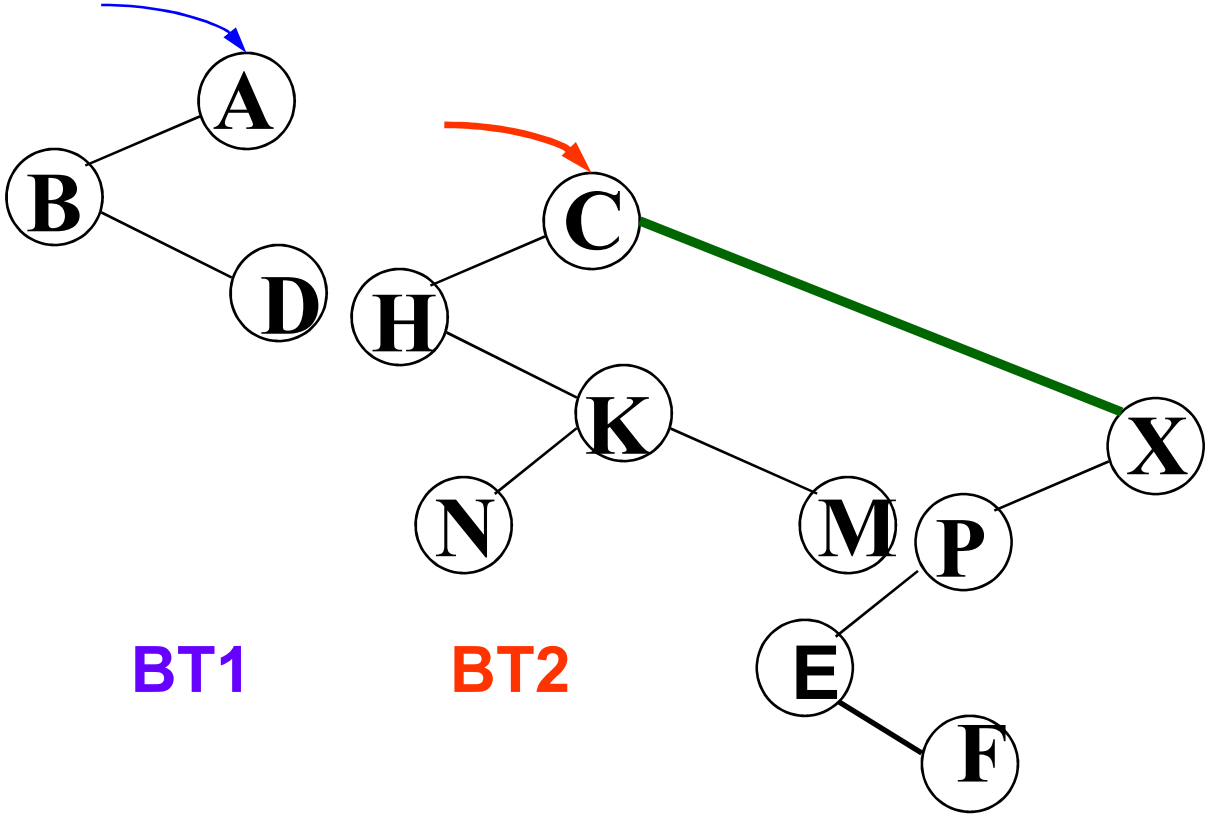


BT1　　BT2　　BT3

森林和二叉树的对应关系

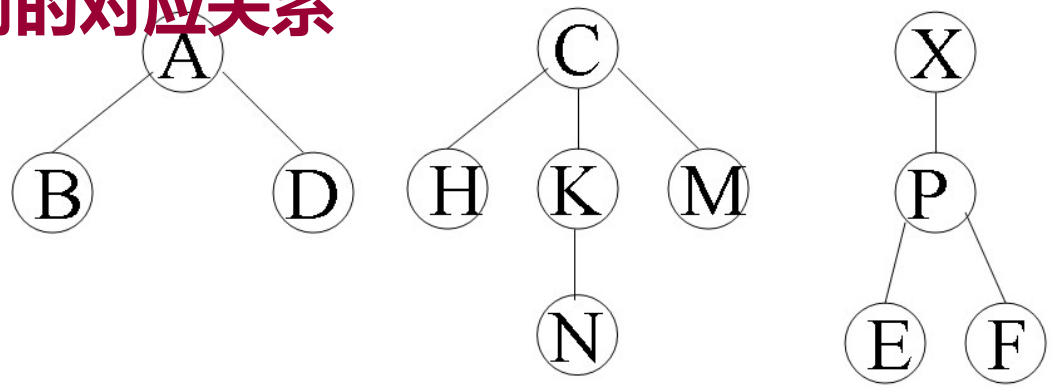


T1　　T2　　T3

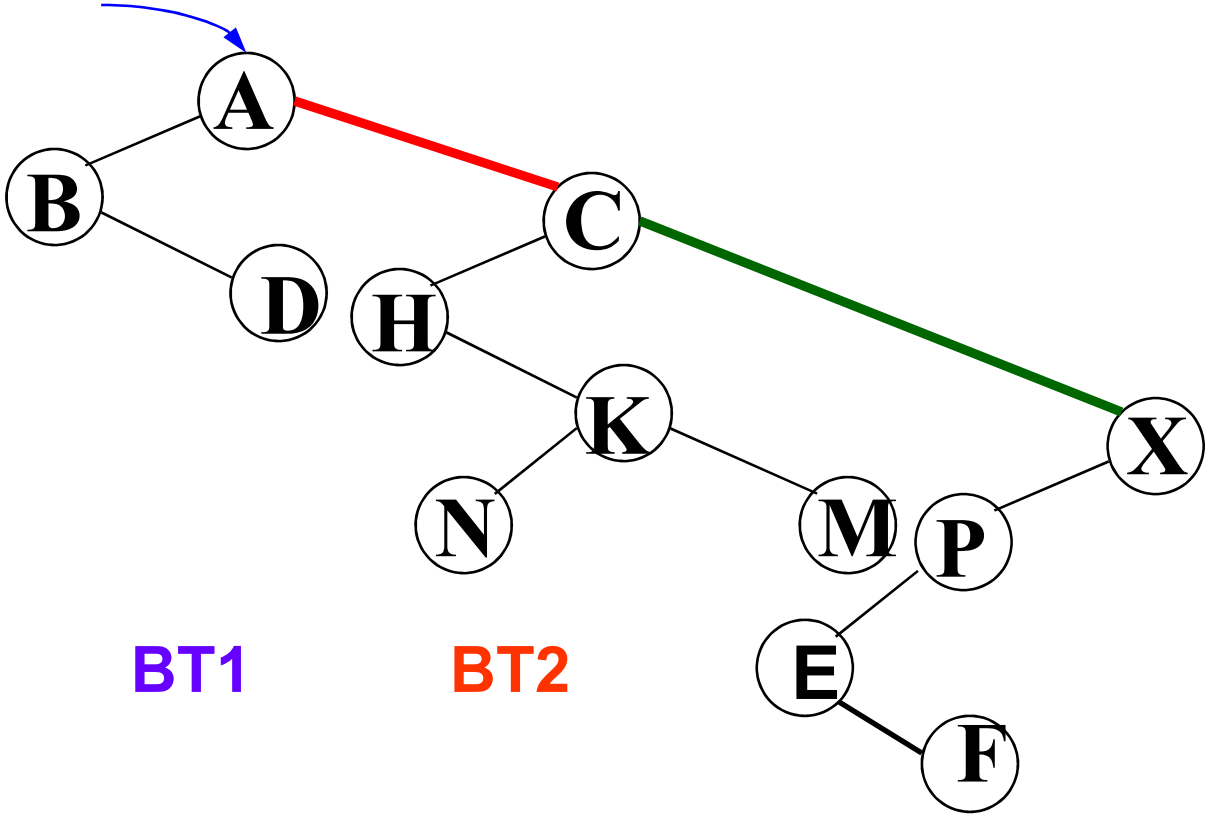


BT1　　BT2　　BT3

森林和二叉树的对应关系



T1　　T2　　T3



BT1　　BT2　　BT3

森林和二叉树的对应关系

设森林:

$F = (T_1, T_2, \ldots, T_n)$; $T_1 = (root, t_{11}, t_{12}, \ldots, t_{1m})$;

二叉树:

$B = (LBT, Node(root), RBT)$;

若 $F = \Phi$，则 $B = \Phi$;

否则,

由 $ROOT(T_1)$ 对应得到 $Node(root)$;

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

由 $(T_2, T_3, \ldots, T_n)$ 对应得到 RBT。

森林和二叉树的转换

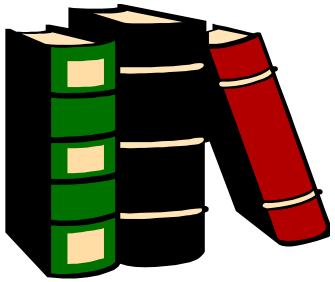
- 森林以孩子兄弟表示法存，相当于将森林转换成二叉树
- 好处：借助二叉树的操作实现森林的操作
- 要求：

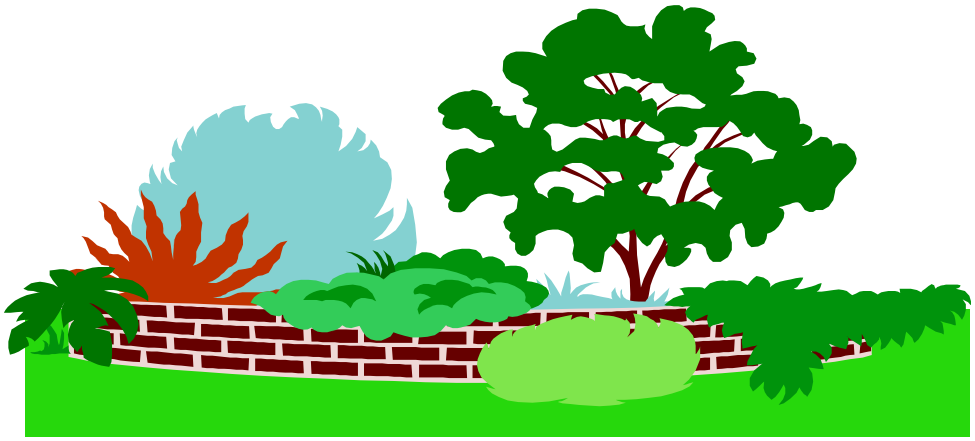
➢ 掌握森林和二叉树的转换

➢ 利用二叉树的操作实现森林的操作

6.4.3 树和森林的遍历

- 树的遍历
- 森林的遍历
- 树的遍历的应用





6.4.3 树和森林的遍历

- 树的遍历可有三条搜索路径:
  - 先根(次序)遍历:若树不空，则先访问根结点，然后依次先根遍历各棵子树。
  - 后根(次序)遍历:若树不空，则先依次后根遍历各棵子树，然后访问根结点。
  - 按层次遍历:若树不空，则自上而下自左至右访问树中每个结点。

先根遍历时顶点的访问次序：

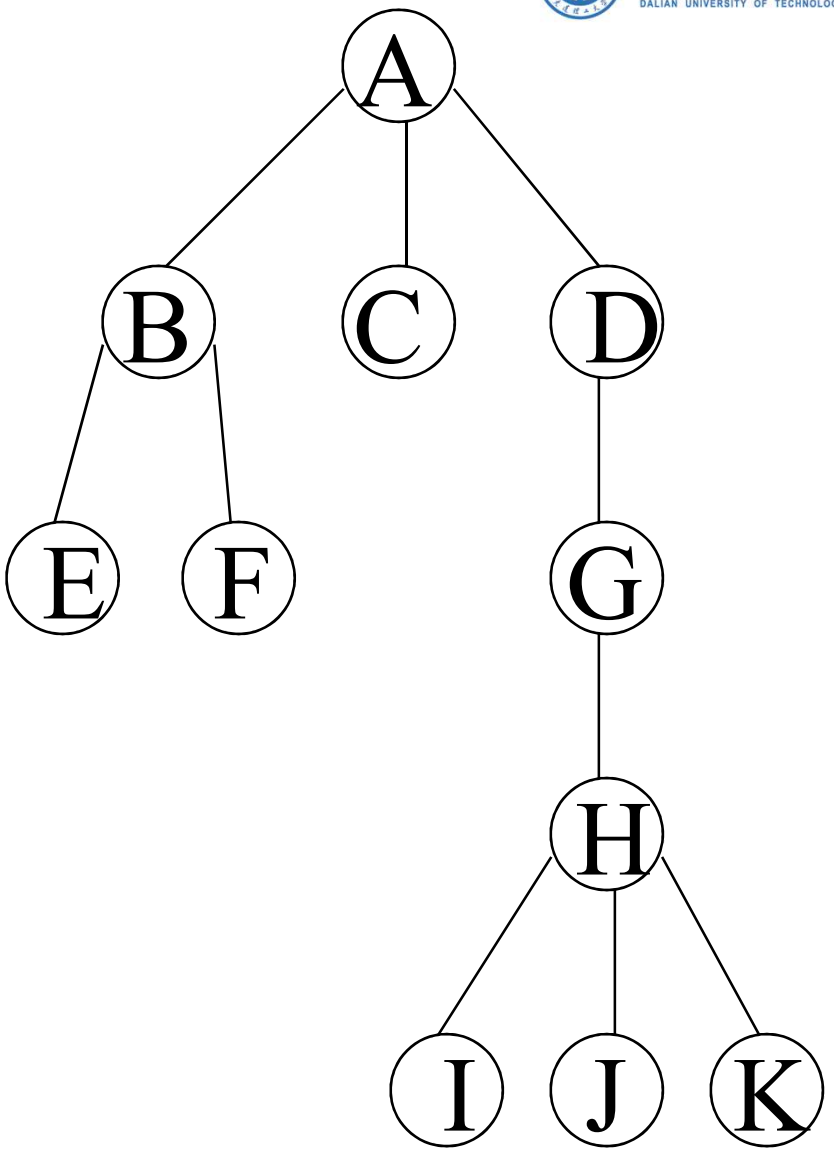
A B E F C D G H I J K

后根遍历时顶点的访问次序：

E F B C I J K H G D A

层次遍历时顶点的访问次序：

A B C D E F G H I J K



树的先根(次序)遍历非递归算法

```
void PreorderTraverse(BiTree T){//二叉树的先序遍历非递归算法
  SeqStack  s ;
  s.top=-1; p = T;
  while(p){
    while(p){printf("%c",p->data);
             if(p->rchild)
               if(s.top==MAX-1) exit (0);
               else s.data[++s.top]=p->rchild;
                 p =p->lchild;}
    if (s.top!=-1)  p=s.data[s.top--];
   }
}
```

```
typedef struct CSNode{
    ElemType        data;
    struct CSNode
        *fc, *nb;
} CSNode, *CSTree;
//树的孩子兄弟表示法

typedef struct BiTNode {
   ElemType      data;
  struct BiTNode  *lchild, *rchild;
} BiTNode, *BiTree;
//二叉树的二叉链表表示法
```

先根(次序)遍历与对应的二叉树的先序遍历相同

树的先根(次序)遍历非递归算法

```
void PreorderTraverse(CSTree T){//树的先根遍历非递归算法
  SeqStack  s ;
  s.top=-1; p = T;
  while(p){
    while(p){printf("%c",p->data);
             if(p->nb)
               if(s.top==MAX-1) exit (0);
               else s.data[++s.top]=p-> nb;
                 p =p->fc;}
    if (s.top!=-1)  p=s.data[s.top--];
  }
}
```

```
typedef struct CSNode{
    ElemType        data;
    struct CSNode
        *fc, *nb;
} CSNode, *CSTree;
//树的孩子兄弟表示法
```

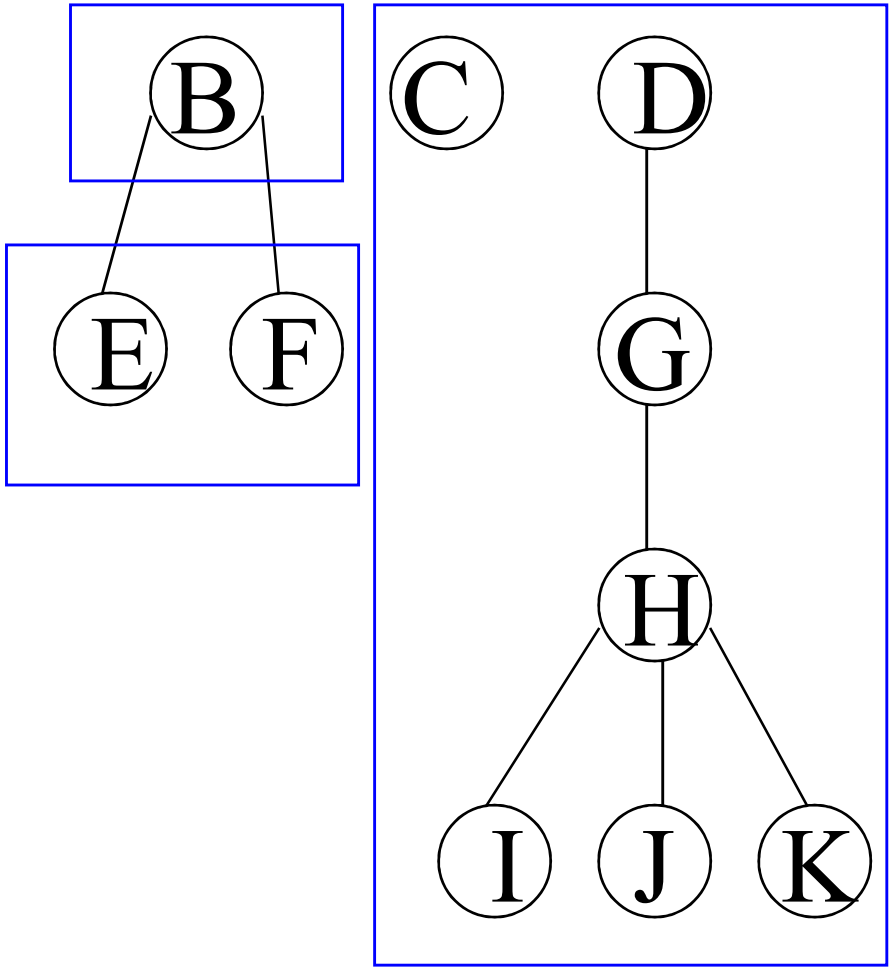
```
typedef struct BiTNode {
   ElemType      data;
  struct BiTNode  *lchild, *rchild;
} BiTNode, *BiTree;
//二叉树的二叉链表表示法
```

树的后根遍历非递归算法

后根(次序)遍历与对应的二叉树的中序遍历相同

先根(次序)遍历与对应的二叉树的先序遍历相同

森林由三部分构成:



1．森林中第一棵树的根结点;

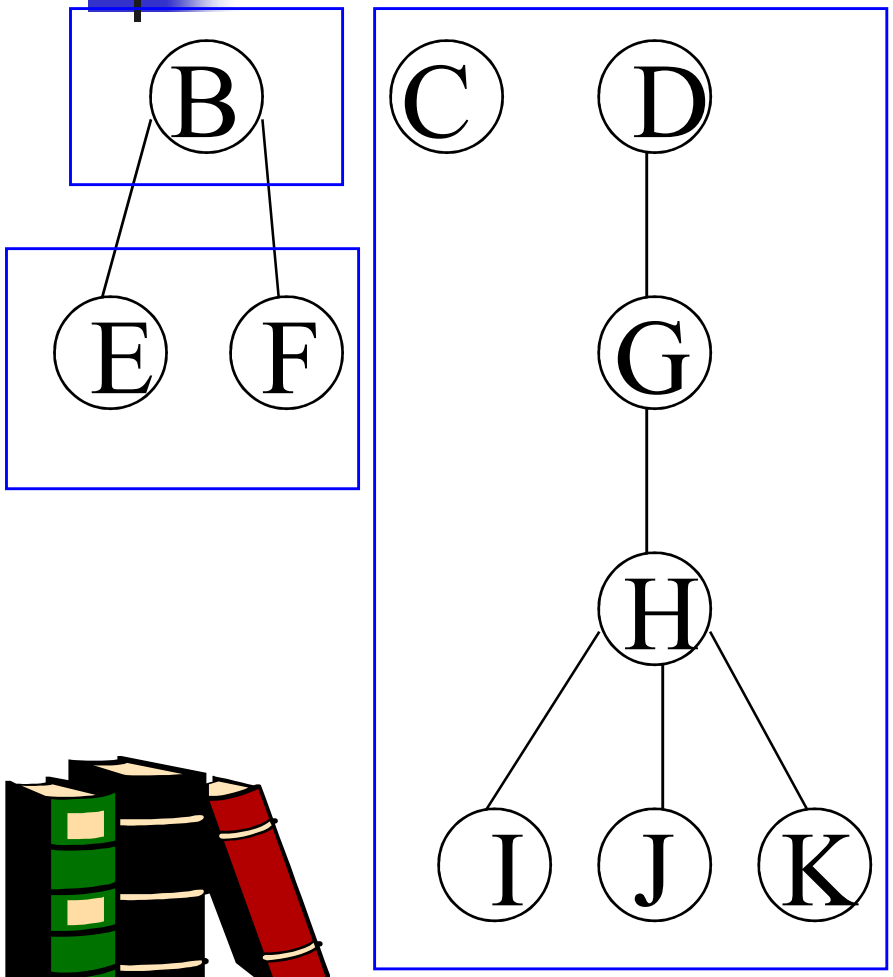
2．森林中第一棵树的子树森林;

3．森林中其它树构成的森林。

6.4.3 树和森林的遍历--森林的遍历

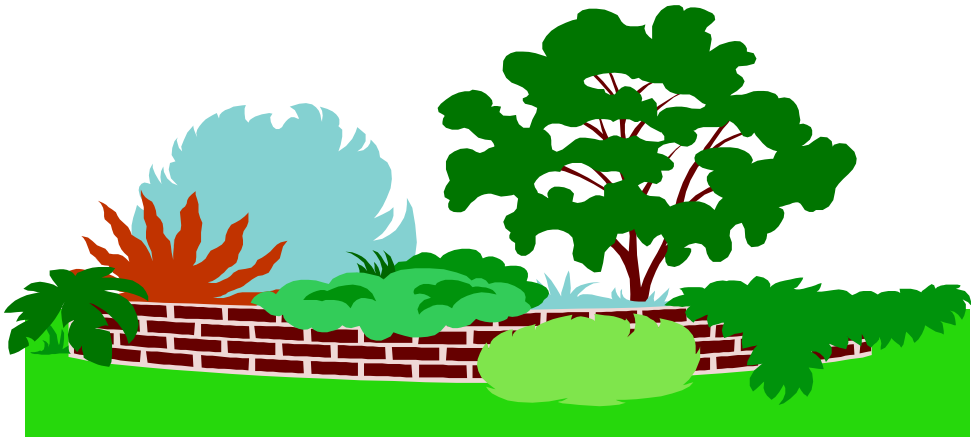
- 先序遍历：若森林不空，则访问森林中第一棵树的根结点；先序遍历森林中第一棵树的子树森林；先序遍历森林中(除第一棵树之外)其余树构成的森林----即：依次从左至右对森林中的每一棵树进行先根遍历。
- 中序遍历：若森林不空，则中序遍历森林中第一棵树的子树森林；访问森林中第一棵树的根结点；中序遍历森林中(除第一棵树之外)其余树构成的森林----即：依次从左至右对森林中的每一棵树进行后根遍历。

6.4.3 树和森林的遍历--森林的遍历



先序遍历：BEFCDGHIJK

中序遍历：EFBCIJKHGD



树的遍历和二叉树遍历的对应关系？

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历