



北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

数据结构与算法设计

课程内容



课程内容：数据结构部分

概述

线性表

栈与队列

数组与广义表

串

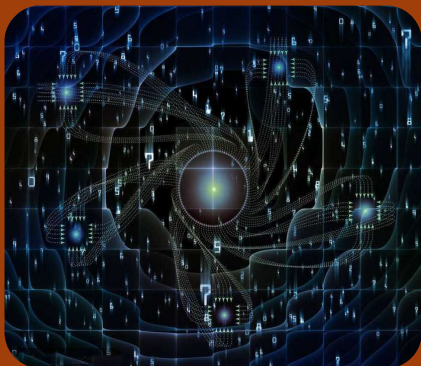
树

图

查找

内部排序

外部排序



课程内容：算法设计部分

概述

分治

动态规划

贪心

回溯

.....

.....

.....

.....

计算模型

可计算理论

计算复杂性



树的定义与基本术语

二叉树

遍历二叉树与线索二叉树

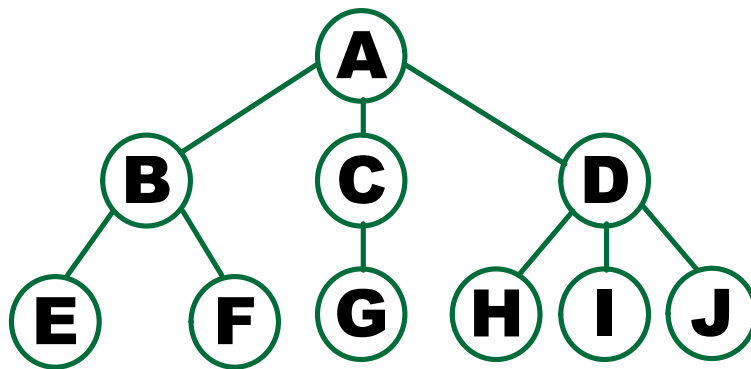
树和森林

最优二叉树（赫夫曼树）

5.1 树的定义与基本术语

树是 n 个结点的有限集合，在任意一棵非空树中：

- (1) 有且仅有一个称为根root的结点。
- (2) 其余结点可分为若干个互不相交的集合，且这些集合中的每一集合本身又是一棵树，称为根的子树。



树是递归结构，树的定义是递归定义。

5.1 树的定义与基本术语

- 数据对象D

D是具有相同特性的数据元素的集合。

- 数据关系 R

若D为空集，则称为空树。

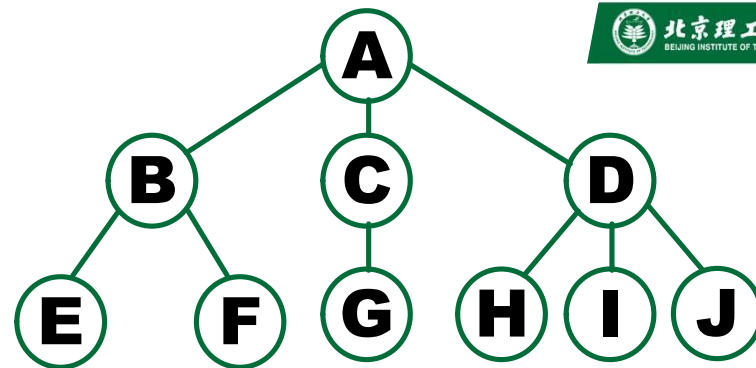
否则：

(1) D中存在唯一的称为根的数据元素root；

(2) 当 $n > 1$ 时，其余结点可分为 $m(m > 0)$ 个互不相交的有限集 T_1, T_2, \dots, T_m ，其中每一棵子集本身又是一棵符合本定义树，称为根root的子树。

5.1 树的定义与基本术语

例：右面的图是一棵树 T 。



$T = \{ A, B, C, D, E, F, G, H, I, J \}$

A是根，其余结点可以划分为3个互不相交的集合：

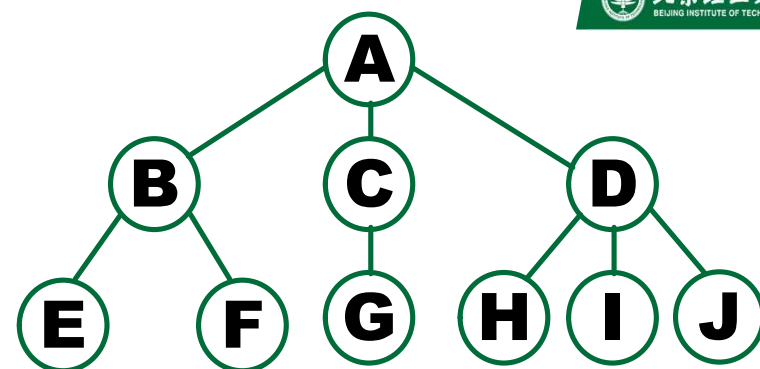
$T_1 = \{ B, E, F \}$ $T_2 = \{ C, G \}$ $T_3 = \{ D, H, I, J \}$

这些集合中的每一集合本身又都是一棵树，它们是根 A 的子树。

对于 T_1 ，B是根，其余结点可以划分为两个互不相交的集合：

$T_{11} = \{ E \}$ $T_{12} = \{ F \}$ ， T_{11} ， T_{12} 是B的子树。

5.1 树的定义与基本术语

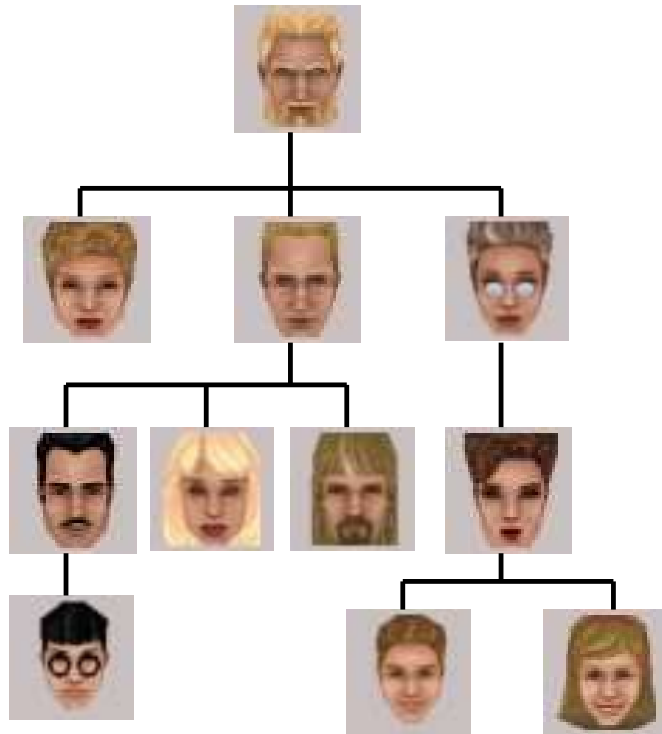


从逻辑结构看

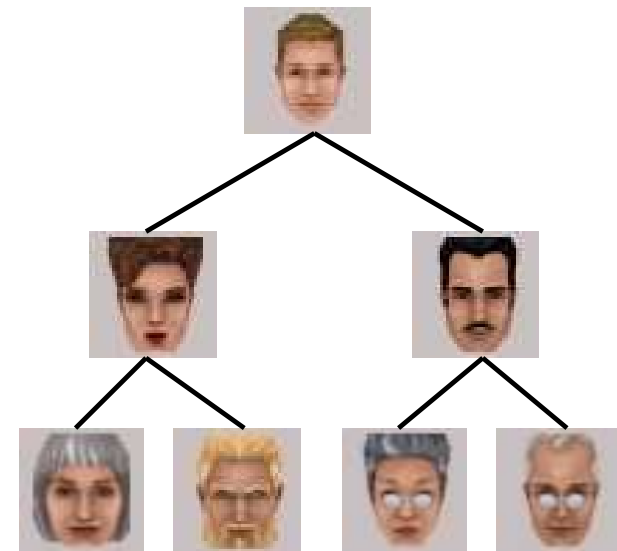
- 1) 树中只有**根结点**没有前趋;
- 2) 除**根**外, 其余结点都有且仅有一个前趋;
- 3) **树的结点**, 可以有零个或多个后继;
- 4) 树是一种分支结构 (除了一个称为**根**的结点之外) 每个元素都有且仅有一个直接前趋, **有且仅**有零个或多个直接后继。
- 5) 除根之外的其它结点, 都存在惟一一条从根到该结点的**路径**;

5.1 树的定义与基本术语

树的应用



家族树



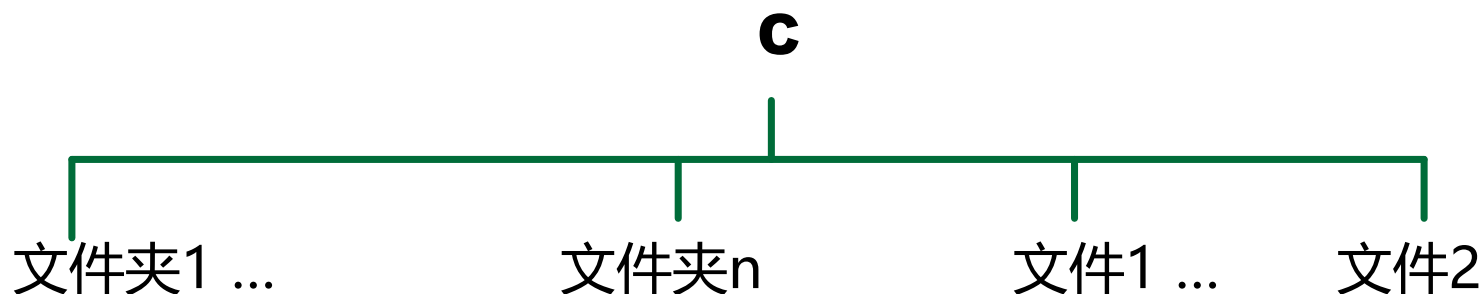
血统树
(二叉树)

5.1 树的定义与基本术语

- 树的应用

常用的数据组织形式——计算机的文件系统。

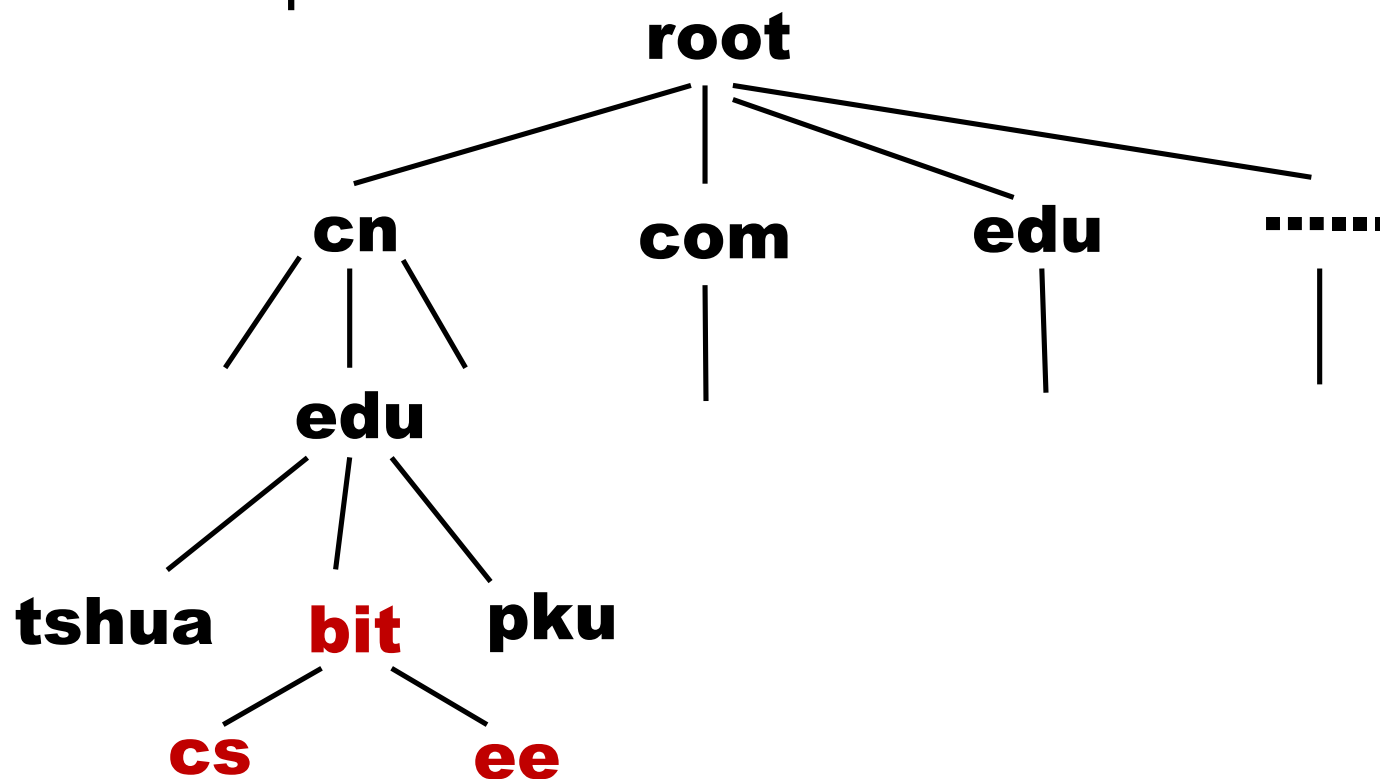
不论是DOS文件系统还是window文件系统，所有的文件都是用树的形式进行组织。



5.1 树的定义与基本术语

树的应用

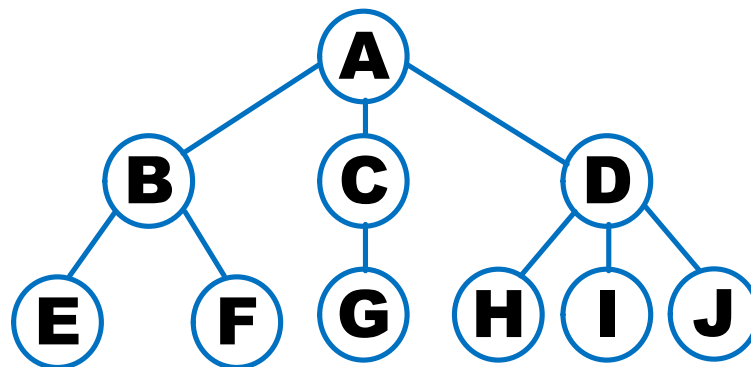
DNS Name Space.



5.1 树的定义与基本术语

树的表示

- 1) 图示表示
- 2) 二元组表示
- 3) 文氏图表示
- 4) 广义表表示
- 5) 凹入表示法 (类似书的目录)



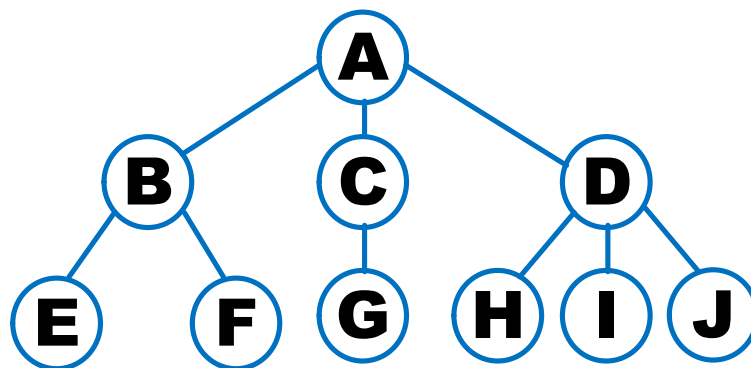
5.1 树的定义与基本术语

树的表示

2) 二元组表示

$T = \{ A, B, C, D, E, F, G, H, I, J \}$

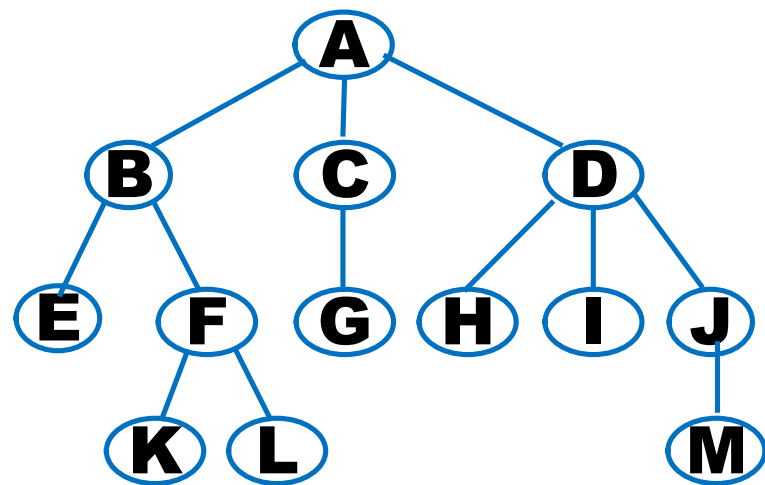
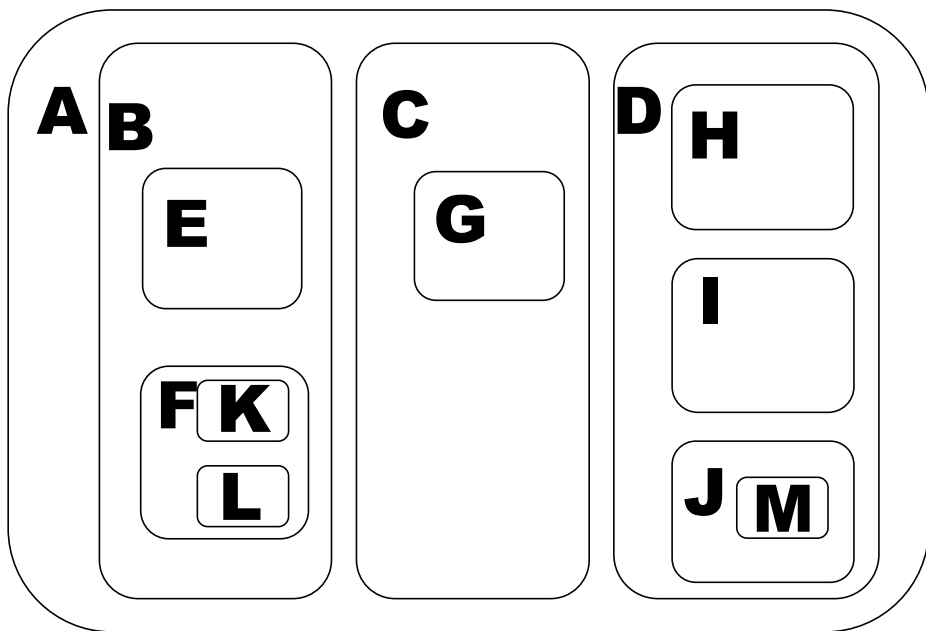
$R = \{ \langle A, B \rangle, \langle A, C \rangle, \langle A, D \rangle, \langle B, E \rangle, \langle B, F \rangle, \langle C, G \rangle, \langle D, H \rangle, \langle D, I \rangle, \langle D, J \rangle \}$



5.1 树的定义与基本术语

树的表示

3) 文氏图表示

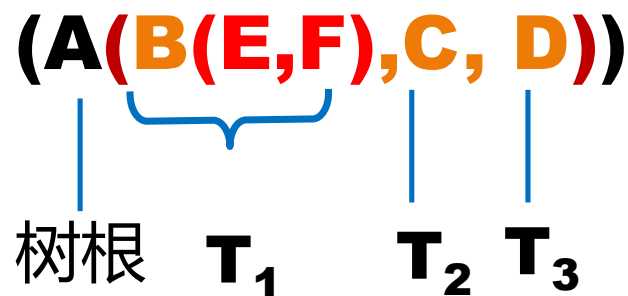
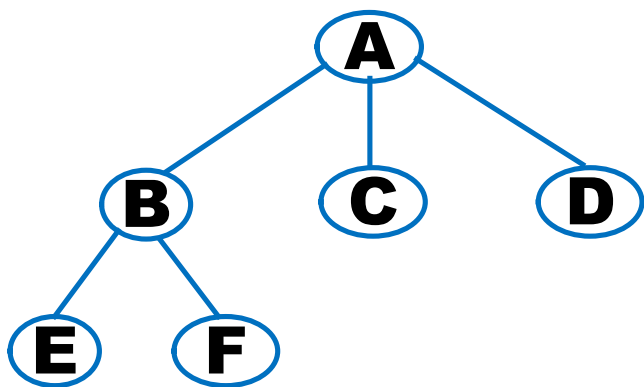


5.1 树的定义与基本术语

树的表示

4) 广义表表示

假设树的根为root, 子树为 T_1, T_2, \dots, T_n , 与该树对应的广义表为L, 则: $L = (\text{原子}(\text{子表1}, \text{子表2}, \dots, \text{子表n}))$, 其中原子对应root, 子表 $i (1 \leq i \leq n)$ 对应 T_i .

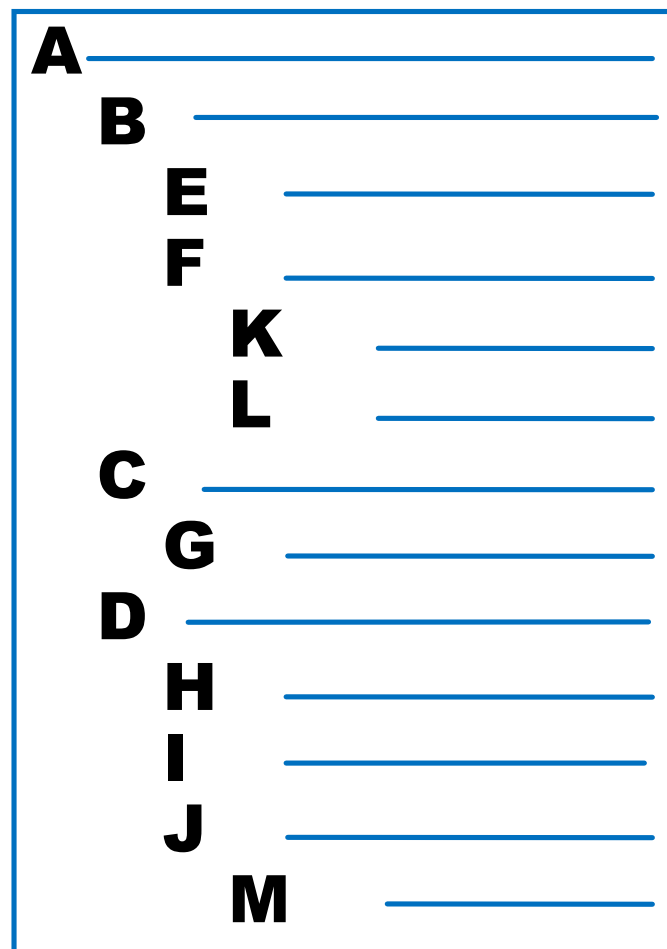
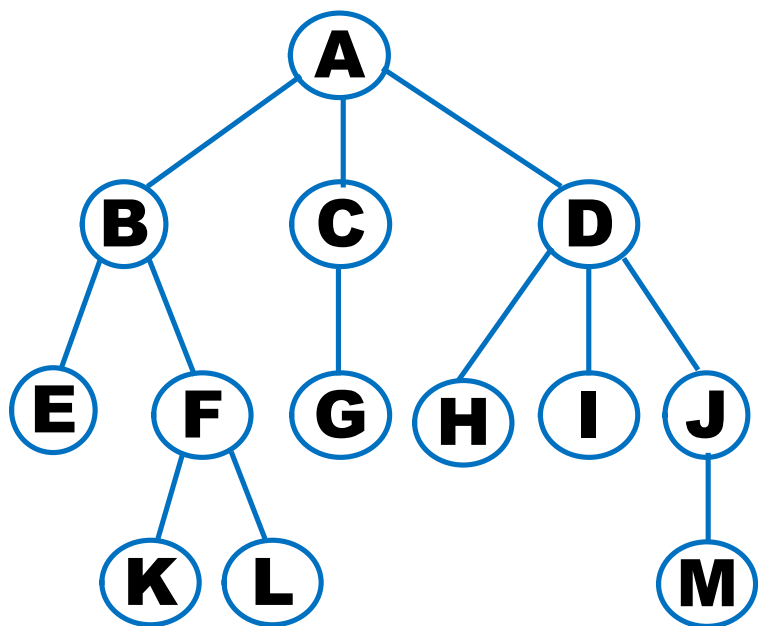


(A,(B,((E),(F))),(C),(D)))

5.1 树的定义与基本术语

树的表示

5) 凹入表示



5.1 树的定义与基本术语

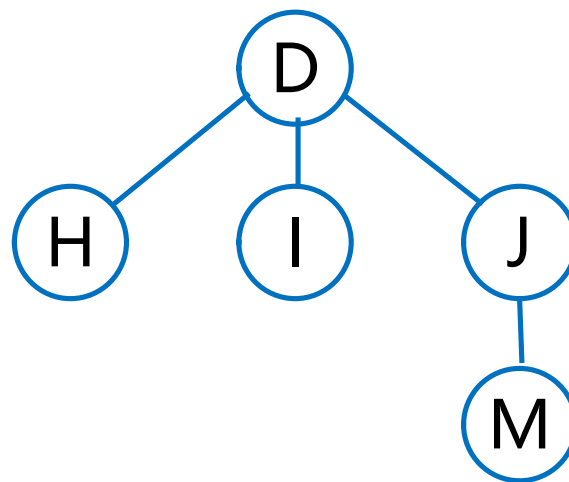
结点: 数据元素+若干指向子树的分支

结点的度: 分支的个数

树的度: 树中所有结点的度的最大值

叶子结点: 度为零的结点

分支结点: 度大于零的结点

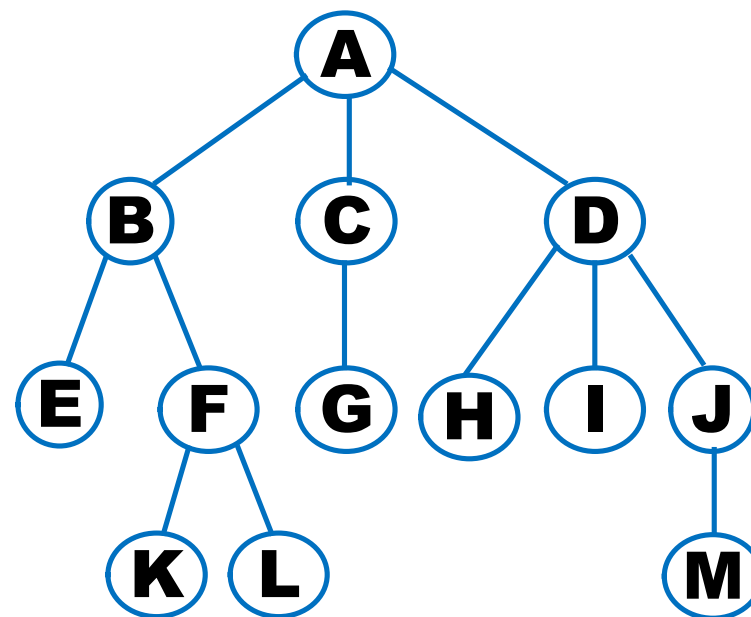


5.1 树的定义与基本术语

(从根到结点的)路径:

由从根到该结点所经分支
和结点构成

孩子结点、双亲结点、
兄弟结点、堂兄弟
祖先结点、子孙结点



结点的层次: 假设根结点的层次为1,第 i 层的
结点的子树根结点的层次为 $i+1$

树的深度: 树中叶子结点所在的最大层次

5.1 树的定义与基本术语

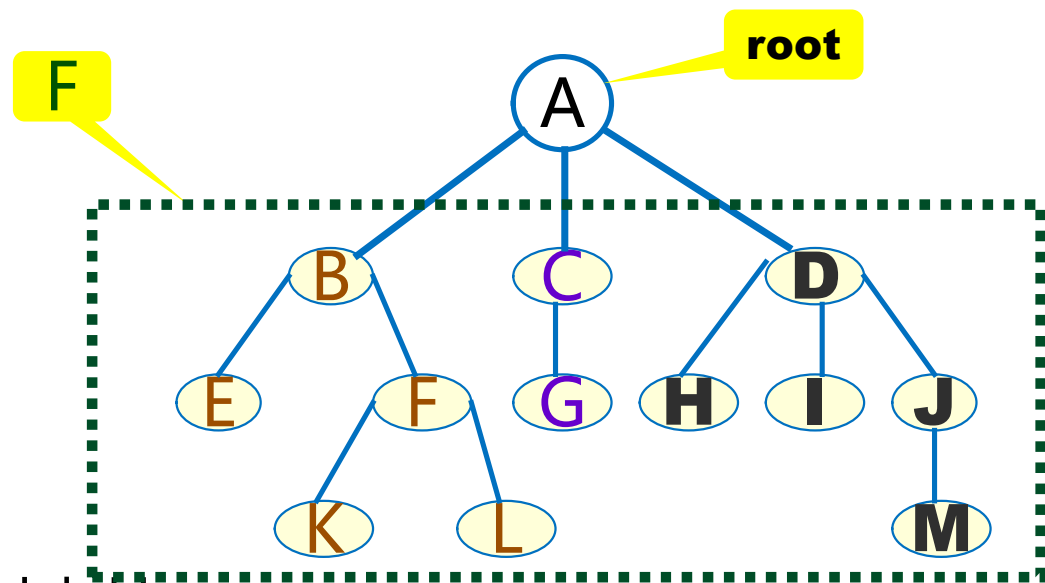
森林：

是 m ($m \geq 0$) 棵互不相交的树的集合。

任何一棵非空树是一个二元组

$$\text{Tree} = (\text{root}, F)$$

其中：root 称为根结点，F 称为子树森林。



有序树：子树之间存在明确的次序关系的树。

无序树：子树之间没有顺序要求。

5.1 树的定义与基本术语

树的基本操作

- 1) InitTree (&T);
构造空树 T。
- 2) DestroyTree (&T);
销毁树 T。
- 3) CreateTree (&T, definition);
按 definition 构造树 T。
- 4) ClearTree (&T);
将树 T 清空。
- 5) TreeEmpty (T);
若树 T 为空, 返回 TURE, 否则返回 FALSE。
- 6) TreeDepth (T);
返回树 T 的深度。

5.1 树的定义与基本术语

树的基本操作

7) Root (T);

返回 T 的根结点。

8) Value (T, &cur_e);

返回 T 树中 cur_e 结点的值。

9) Assign (T, cur_e, value);

将 T 树中结点 cur_e 的值赋值为value。

10) Parent (T, cur_e);

返回 T 树 cur_e 结点的双亲。

11) LeftChild (T, cur_e);

返回 T 树 cur_e 结点的最左孩子。

5.1 树的定义与基本术语

树的基本操作

12) RightSibling (T, cur_e);

返回 T 树 cur_e 结点的右兄弟。

13) InsertChild (&T, &p, i, c);

将 c 插入到树 T 中 p 所指向的第 i 棵子树中。

14) DeleteChild (&T, &p, i);

删除树 T 中 p 所指向的第 i 棵子树。

15) **TraverseTree** (T, Visit());

按某种次序对 T 树的每个结点调用函数Visit()一次且至多一次。也称为按照某种次序对树进行遍历。

5.1 树的定义与基本术语

线性结构

第一个数据元素
(无前驱)

最后一个数据元素
(无后继)

其它数据元素
(一个前驱、
一个后继)

树型结构

根结点
(无前驱)

多个叶子结点
(无后继)

其它数据元素
(一个前驱、
多个后继)

5.1 树的定义与基本术语

线性结构	树型结构
第一个数据元素 (无前驱)	根结点 (无前驱)
最后一个数据元素 (无后继)	多个叶子结点 (无后继)
其它数据元素 (一个前驱、一个后继)	其它数据元素 (一个前驱、多个后继)

5.2 二叉树

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。

形式定义

数据关系 R 满足：

若 $D = \Phi$ ，则 $R = \Phi$ ，称为是空二叉树。

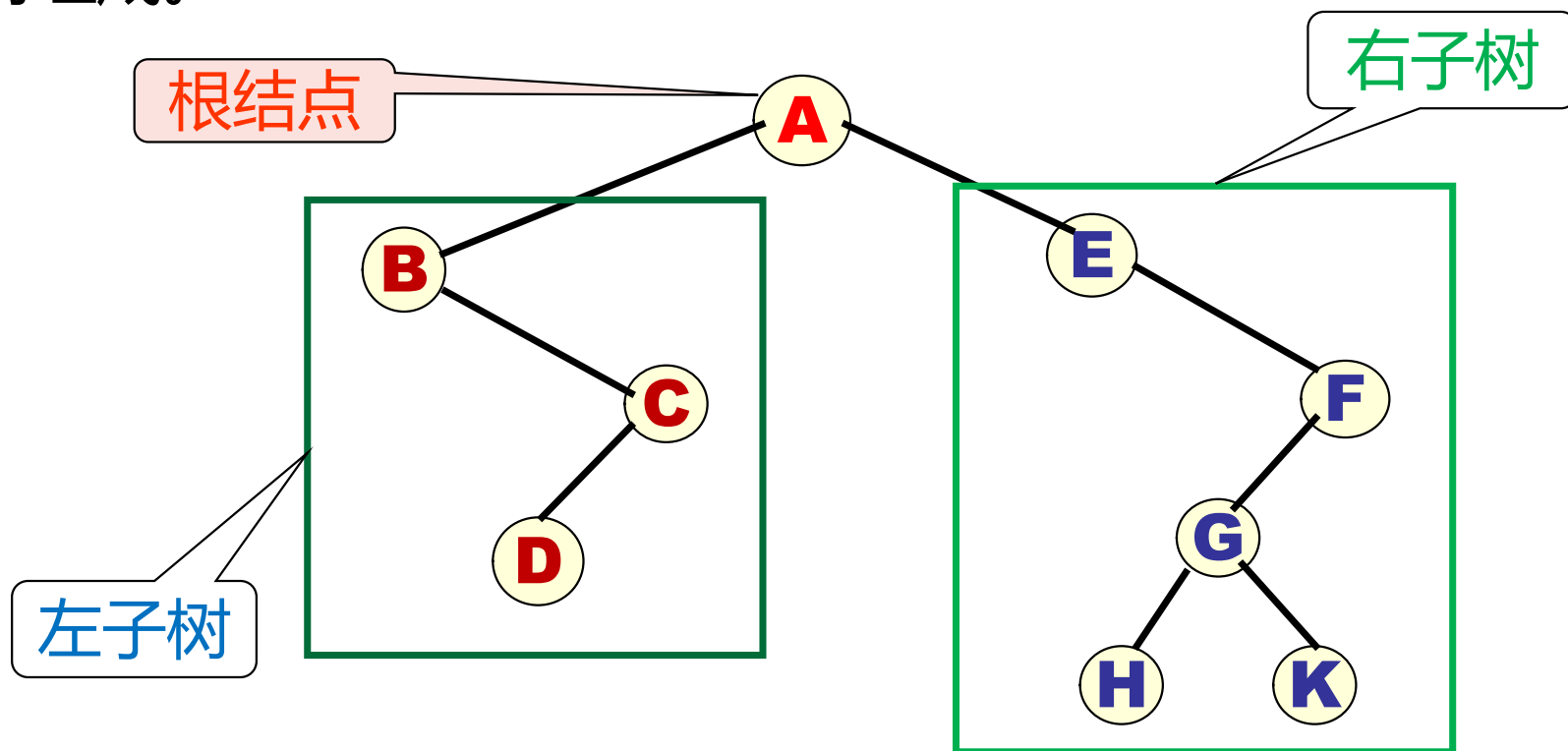
若 $D \neq \Phi$ ，则 $R = \{ H \}$ ， H 是如下二元关系：

- (1) 在 D 中存在惟一的称为根的数据元素 $root$ ，它在关系 H 下无前驱；
- (2) 若 $D - \{root\} \neq \Phi$ ，则存在 $D - \{root\} = \{D_l, D_r\}$ ，且 $D_l \cap D_r = \Phi$ 。

5.2 二叉树

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。



5.2 二叉树

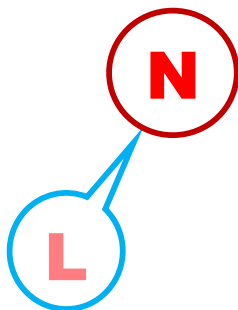
二叉树的五种基本形态



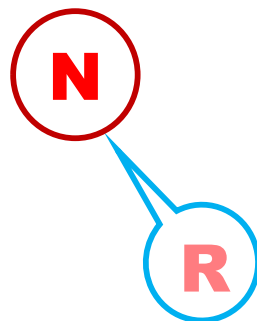
空树



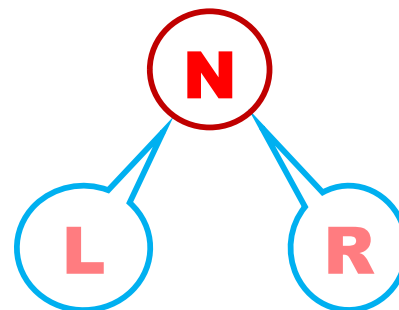
只有根结
点



只有左子树



只有右子树



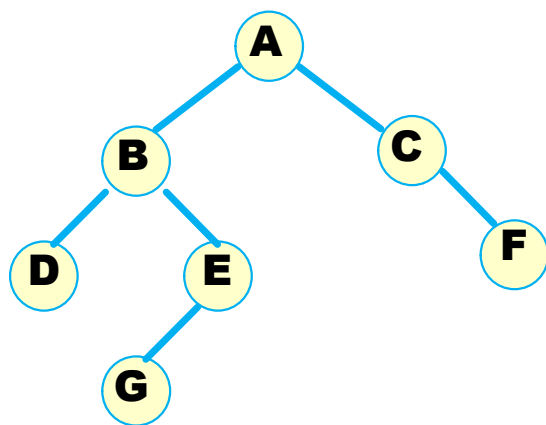
左右子树均非空

说明：

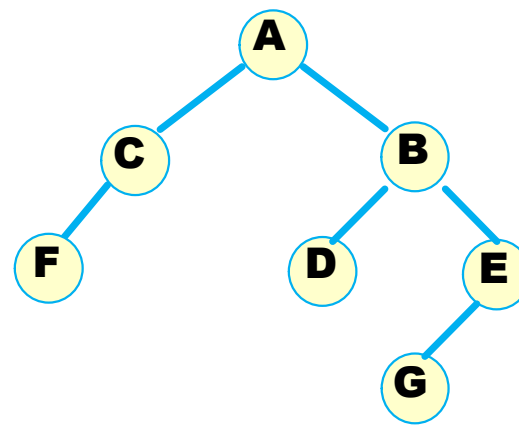
- 1) 二叉树中每个结点最多有两棵子树；二叉树每个结点度小于等于**2**；
- 2) 左、右子树不能颠倒——**有序树**。

5.2 二叉树

二叉树



(a)



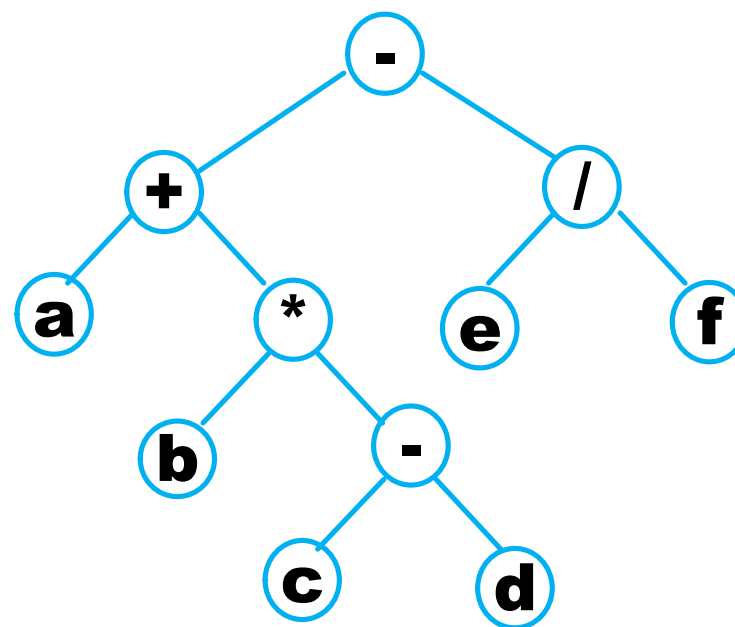
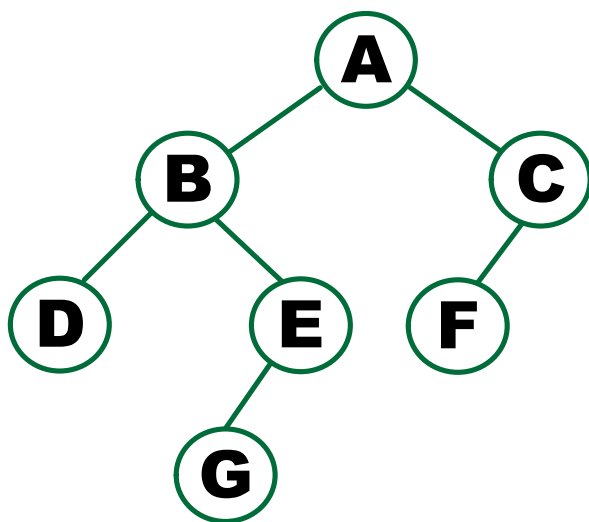
(b)

(a)、(b)是不同的二叉树

(a)的左子树有四个结点，(b)的左子树有两个结点

5.2 二叉树

二叉树的应用

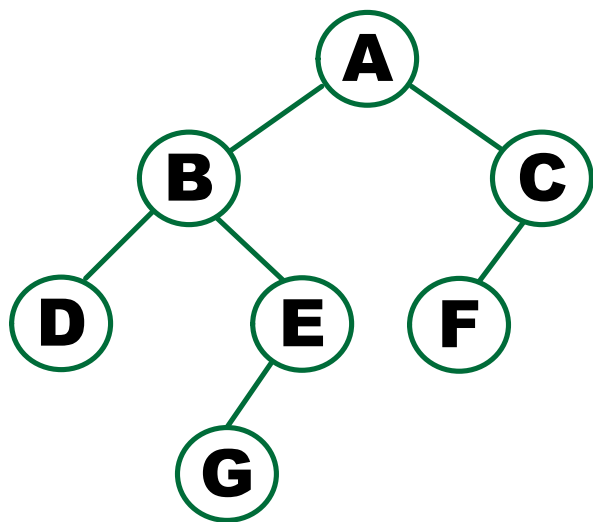


$$a + b * (c - d) - e / f$$

5.2 二叉树

性质1:

在二叉树的第 i 层上至多有 2^{i-1} 个结点($i \geq 1$)。



$i=1$: 最多1个结点

$i=2$: 最多2个结点

$i=3$: 最多4个结点

5.2 二叉树

性质1:

在二叉树的第 i 层上至多有 2^{i-1} 个结点($i \geq 1$)。

用归纳法证明:

归纳基: $i = 1$ 层时, 只有一个根结点,
 $2^{i-1} = 2^0 = 1$;

归纳假设: 假设对所有的 j , $1 \leq j < i$, 命题成立;

归纳证明: 二叉树上每个结点至多有两棵子树, 则第 i 层的结点数=第 $i-1$ 层的结点数乘以2,
即 $2^{i-2} * 2 = 2^{i-1}$ 。

5.2 二叉树

性质2:

深度为 k 的二叉树上至多含 $2^k - 1$ 个结点 ($k \geq 1$)。

证明：基于上一条性质，深度为 k 的二叉树上的结点数至多为：

$$2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$$

5.2 二叉树

性质3:

对任何一棵二叉树, 若它含有 n_0 个叶子结点、 n_2 个度为 2 的结点, 则必存在关系式: $n_0 = n_2 + 1$

证明: 设二叉树上结点总数: $n = n_0 + n_1 + n_2$

又二叉树上分支总数: $b = n_1 + 2n_2$

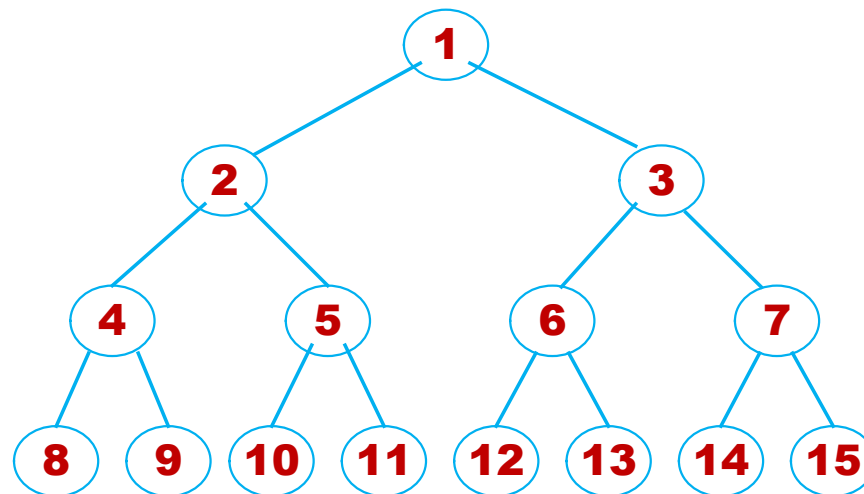
而 $b = n - 1 = n_0 + n_1 + n_2 - 1$

由此, $n_0 = n_2 + 1$

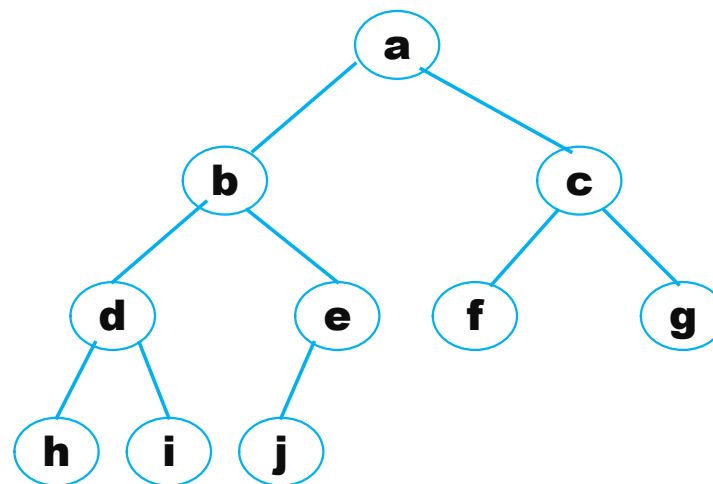
5.2 二叉树

- 两类特殊的二叉树

满二叉树：指的是深度为 k 且含有 $2^k - 1$ 个结点的二叉树。



完全二叉树：树中所含的 n 个结点和满二叉树中编号为 1 至 n 的结点一一对应。



5.2 二叉树

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

证明:

设 完全二叉树的深度为 k

则根据第二条性质得 $2^{k-1} \leq n < 2^k$

$$\text{即 } k-1 \leq \log_2 n < k$$

因为 k 只能是整数, 因此, $k = \log_2 n + 1$

5.2 二叉树

性质5:

若对含 n 个结点的完全二叉树从上到下且从左到右进行1到 n 的编号, 则对完全二叉树中任意一个编号为 i 的结点:

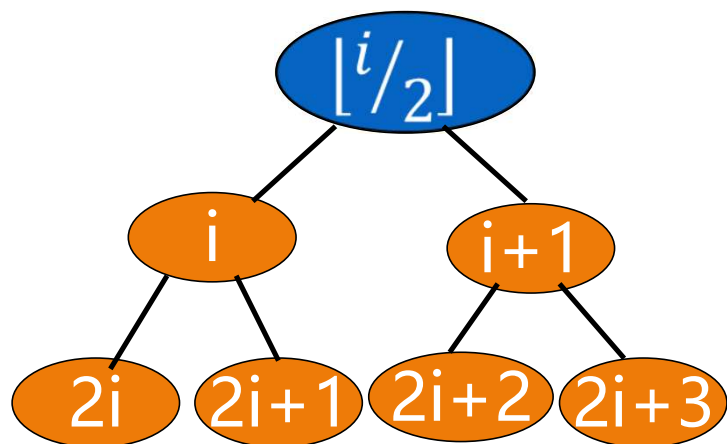
(1) 若 $i=1$, 则该结点是二叉树的根, 无双亲; 否则 ($i>1$), 编号为 $\lfloor i/2 \rfloor$ 的结点为其双亲结点;

(2) 若 $2i>n$, 则该结点无左孩子; 否则, 编号为 $2i$ 的结点为其左孩子结点;

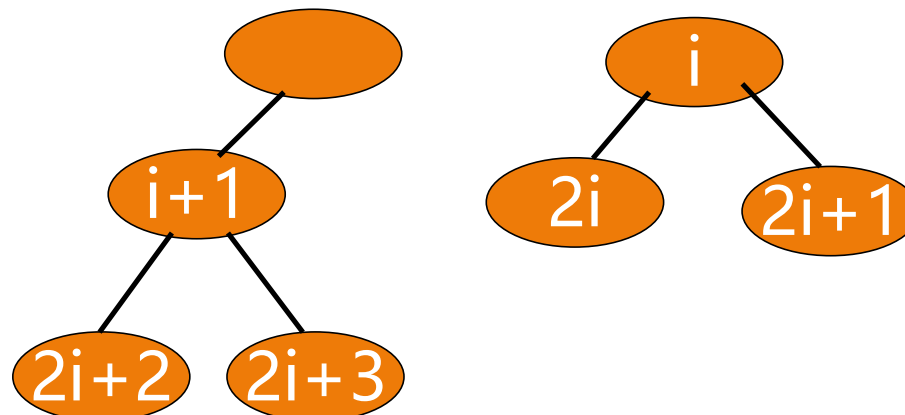
(3) 若 $2i+1>n$, 则该结点无右孩子; 否则, 编号为 $2i+1$ 的结点为其右孩子结点;

5.2 二叉树

性质5:



(a) i 和 $i+1$ 结点在同一层



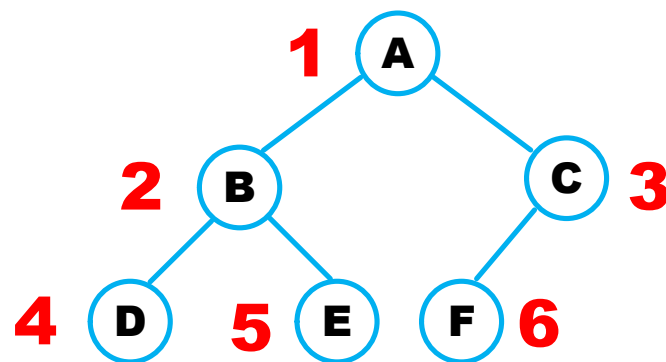
(b) i 和 $i+1$ 结点不在同一层

5.2 二叉树

二叉树的存储结构

1. 二叉树的顺序结构

对于完全二叉树，采用一组连续的内存单元，按编号顺序依次存储完全二叉树的结点。



1 2 3 4 5 6 7

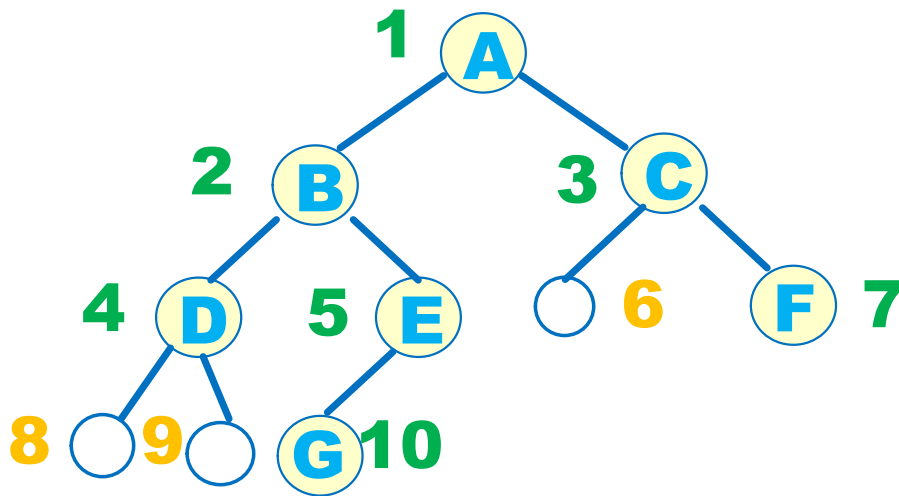
m-1

A B C D E F ...

.....

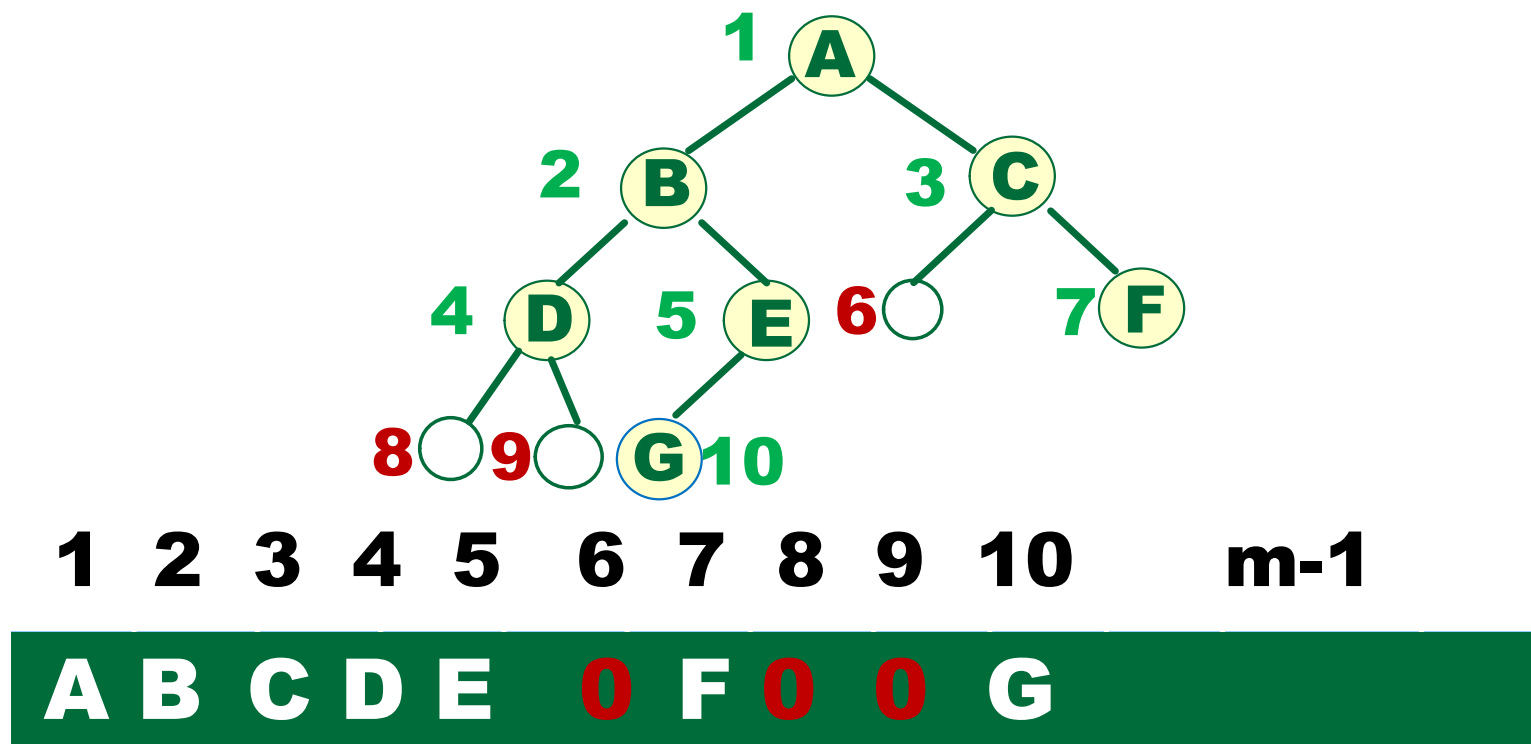
5.2 二叉树

对于一棵一般的二叉树，如果补齐构成完全二叉树所缺少的那些结点，便可以对二叉树的结点进行编号。



5.2 二叉树

将二叉树原有的结点按编号存储到内存单元“相应”的位置上。



对于一些“退化二叉树”，顺序存储结构存在突出缺点：比较浪费空间。



BT[15]

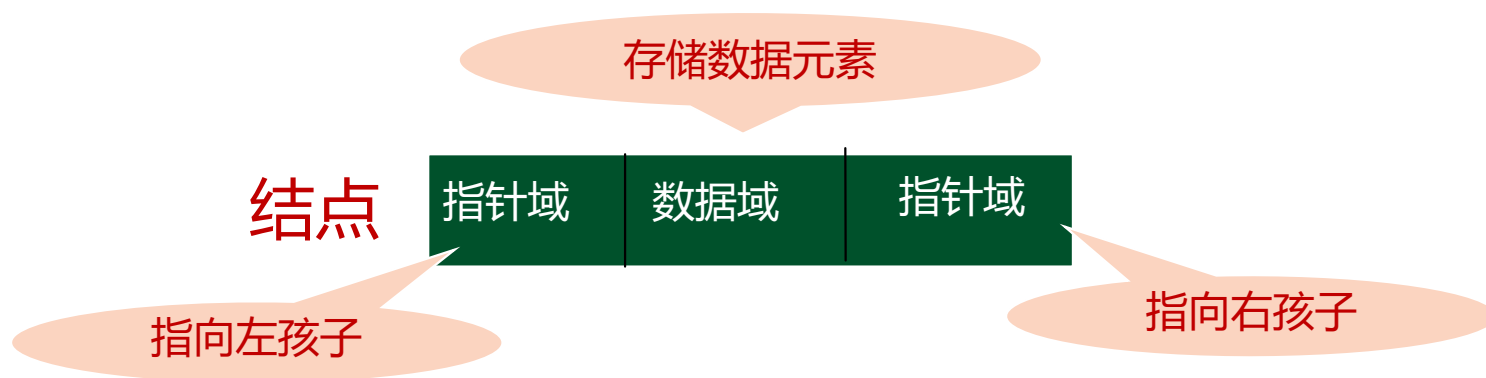
AB  **C**    **D**        

5.2 二叉树

2. 二叉链表

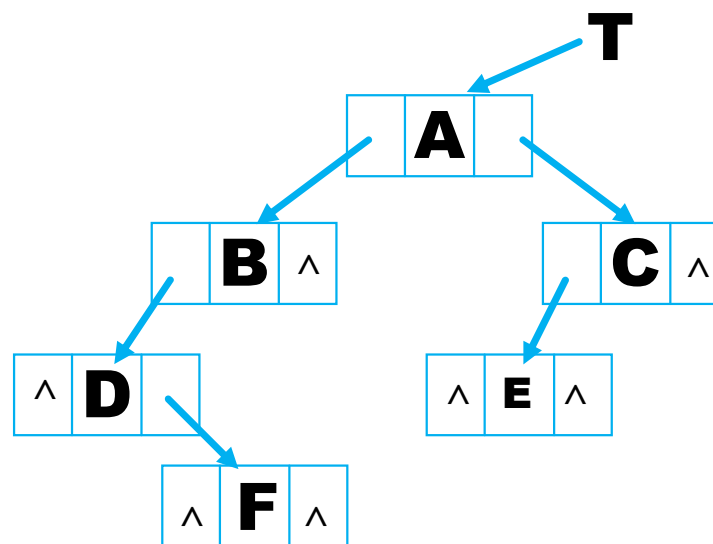
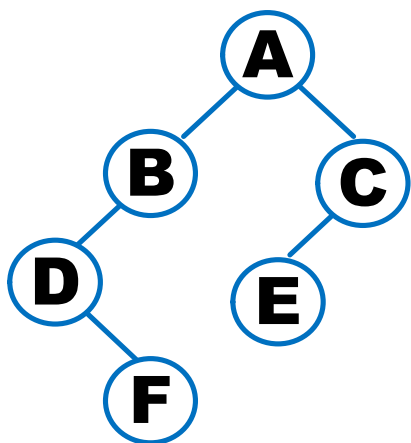
二叉链表中每个结点包含三个域：数据域、左指针域、右指针域。

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



5.2 二叉树

二叉树的二叉链表表示



二叉链表图示

5.2 二叉树

三叉链表

三叉链表中每个结点包含四个域：数据域、双亲指针域、左指针域、右指针域。

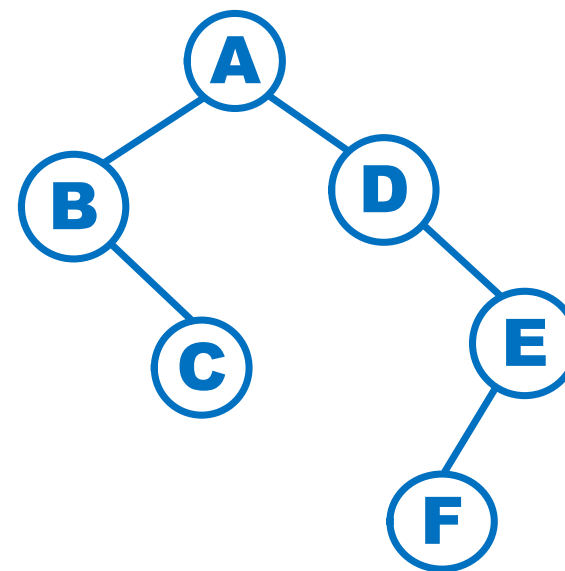
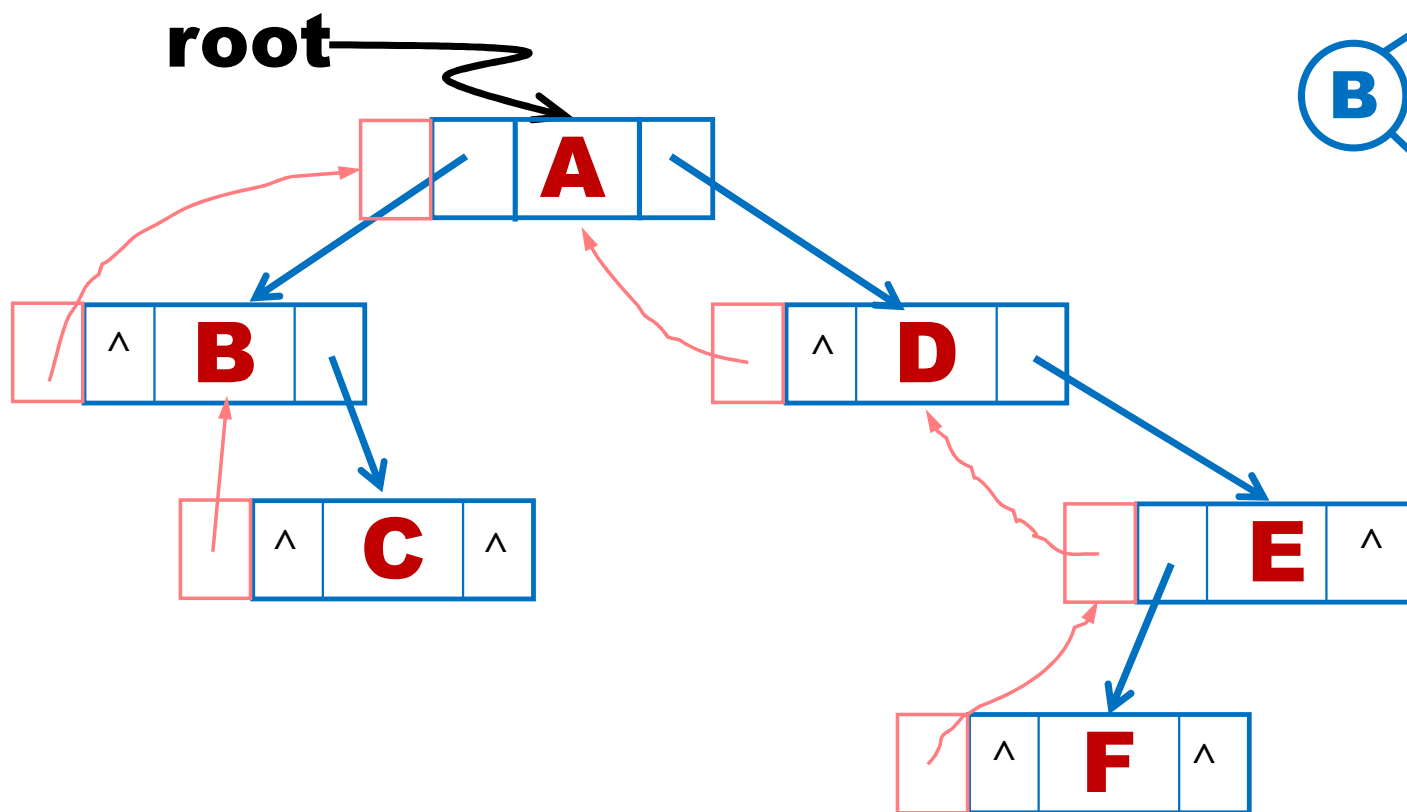
```
typedef struct BiTNode
{
    ElemType data;
    struct BiTNode * lchild, * rchild;
    struct BiTNode * parent;
} BiTNode, *BiTree;
```

结点结构：

parent	lchild	data	rchild
---------------	---------------	-------------	---------------

5.2 二叉树

二叉树的三叉链表表示



5.2 二叉树

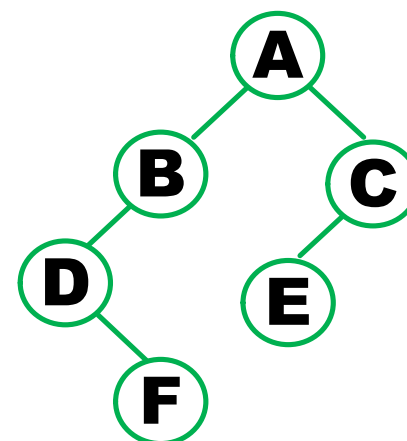
4. 静态二叉链表 采用数组存储。

root = 0

孩子结点在数组中的位置。用-1表示无左孩子或右孩子

Lchild data Rchild

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



5.2 二叉树

4. 静态二叉链表

```
typedef struct BPTNode { // 结点结构
    TElemType data;
    int lchild, rchild ;
} BNode
typedef struct BTree { // 树结构
    BNode nodes[ MAX_TREE_SIZE ];
    int num_node;    // 结点数目
    int root;        // 根结点的位置
} BTree;
```

5.2 二叉树

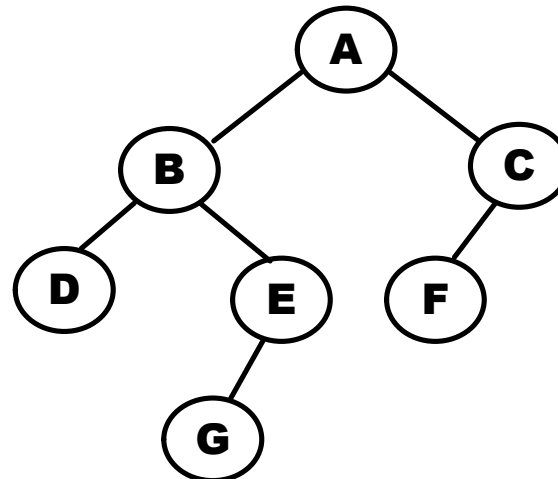
5. 双亲链表

data parent

0	B	2	L
1	C	2	R
2	A	-1	
3	D	0	L
4	E	0	R
5	F	1	L
6	G	4	L

结点

data	parent	LRTag
------	--------	-------



5.2 二叉树

5. 双亲链表

```
typedef struct BPTNode { // 结点结构
    TElemType data;
    int parent;          // 指向双亲的指针
    char LRTag;          // 左、右孩子标志域
} BPTNode
typedef struct BPTree { // 树结构
    BPTNode nodes[ MAX_TREE_SIZE ];
    int num_node;        // 结点数目
    int root;            // 根结点的位置
} BPTree;
```


5.3 遍历二叉树与线索二叉树

遍历的基本概念

遍历：按某种搜索路径访问二叉树中的每个结点，且每个结点仅被访问一次。

访问：含义很广，可以是对结点的各种处理，如修改结点数据、输出结点数据等。

遍历是各种数据结构最基本的操作，许多其它的操作可以在遍历基础上实现。

遍历对线性结构来说很容易解决，但二叉树每个结点都可能有两棵子树，因而需要寻找一种规律，使得二叉树上的结点能线性排列。

5.3 遍历二叉树与线索二叉树

二叉树的遍历

二叉树的遍历，就是按某种次序访问二叉树中的结点，要求每个结点访问一次且仅访问一次。

二叉树由根、左子树、右子树三部分组成。

二叉树的遍历可以分解为：

访问根、遍历左子树、遍历右子树 三个步骤。

5.3 遍历二叉树与线索二叉树

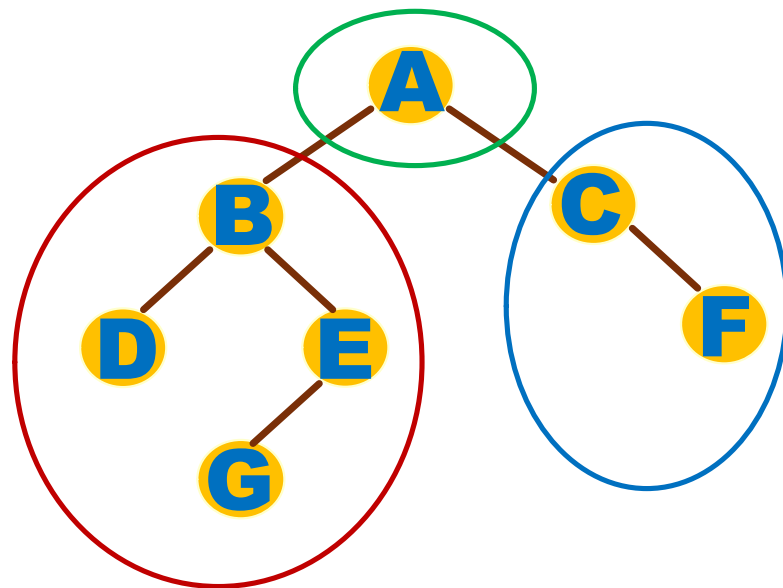
二叉树的遍历

令：**L**：遍历左子树
D：访问根结点
R：遍历右子树

有六种遍历方法：

基本：**DLR**, **LDR**, **LRD**

镜象：**DRL**, **RDL**, **RLD**



约定先左后右，有三种遍历方法：**DLR**, **LDR**, **LRD**，分别根据访问根结点的次序称为：**先序遍历**、**中序遍历**和**后序遍历**。

5.3 遍历二叉树与线索二叉树

二叉树的先序遍历 (**D****L****R**)

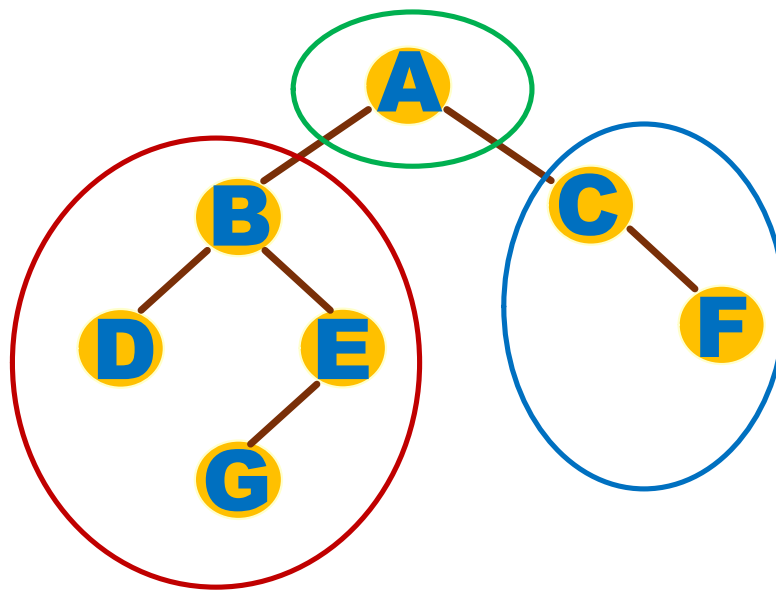
先序遍历 (**D****L****R**)

若二叉树非空

- (1) 访问根结点;
- (2) 先序遍历左子树;
- (3) 先序遍历右子树。

例：先序遍历二叉树

- 1) 访问根结点A
- 2) 先序遍历左子树：即按 **D****L****R** 的顺序遍历左子树
- 3) 先序遍历右子树：即按 **D****L****R** 的顺序遍历右子树



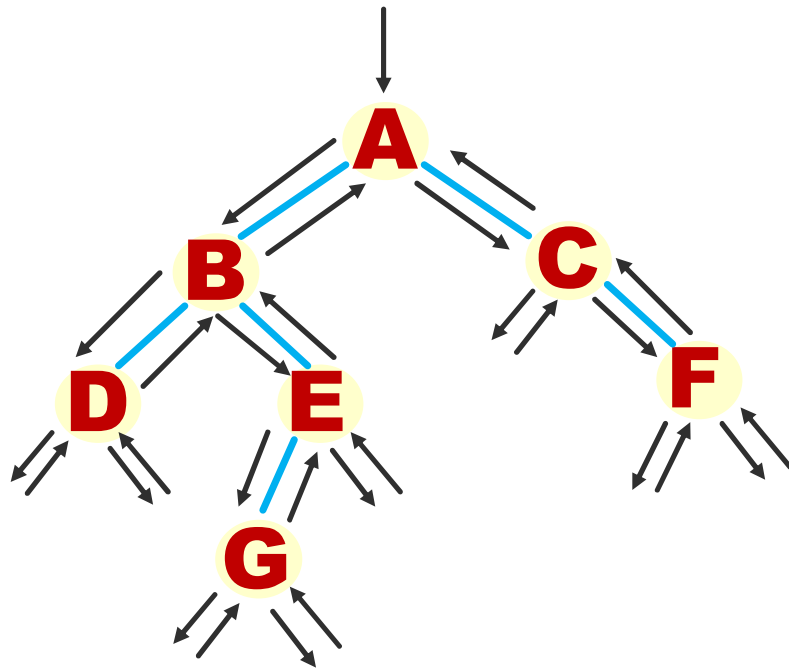
5.3 遍历二叉树与线索二叉树

二叉树的先序遍历 (DLR)

先序遍历 (DLR)

若二叉树非空

- (1) 访问根结点;
- (2) 先序遍历左子树;
- (3) 先序遍历右子树。



先序遍历序列: A, B, D, E, G, C, F

5.3 遍历二叉树与线索二叉树

二叉树的中序遍历 (LDR)

中序遍历 (LDR)

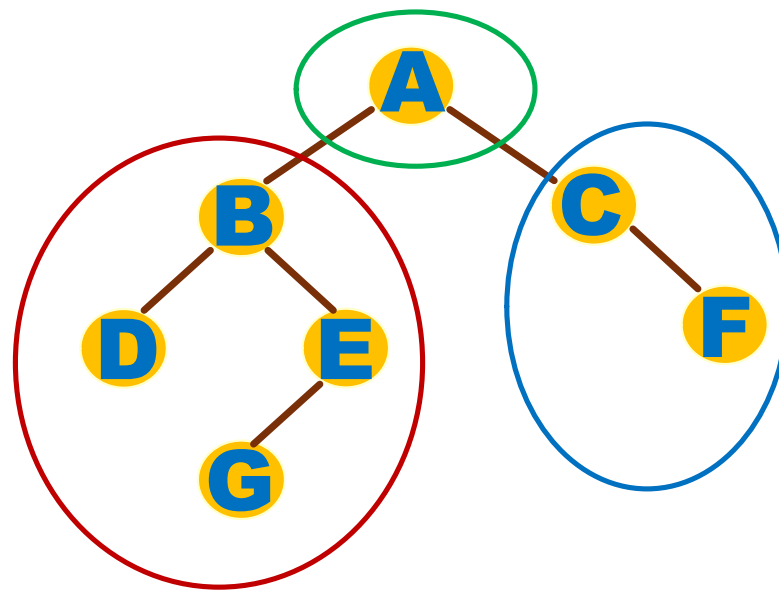
若二叉树非空

- (1) 中序遍历左子树;
- (2) 访问根结点;
- (3) 中序遍历右子树。

例：中序遍历二叉树

- 1) 中序遍历左子树：即按 LDR 的顺序遍历左子树
- 2) 访问根结点A
- 3) 中序遍历右子树：即按 LDR 的顺序遍历右子树

中序遍历序列： **D,B,G,E, A, C,F**



5.3 遍历二叉树与线索二叉树

二叉树的后序遍历 (LRD)

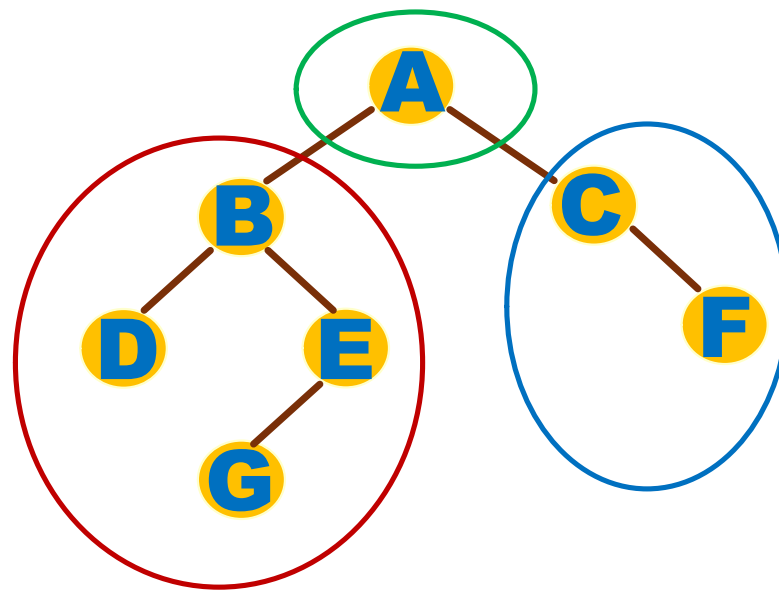
后序遍历 (LRD)

若二叉树非空

- (1) 后序遍历左子树;
- (2) 后序遍历右子树。
- (3) 访问根结点;

例：后序遍历二叉树

- 1) 后序遍历左子树：即按 LRD 的顺序遍历左子树
- 2) 后序遍历右子树：即按 LRD 的顺序遍历右子树
- 3) 访问根结点A



后序遍历序列：D,G,E,B, F,C, A

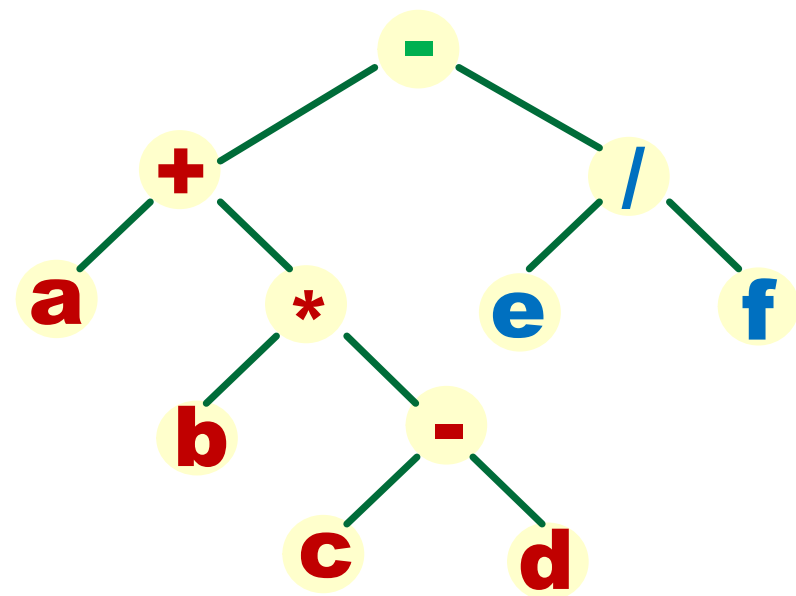
5.3 遍历二叉树与线索二叉树

例：先序遍历、中序遍历、后序遍历二叉树。

先序遍历序列：**-**,**+**,**a**,*****,**b**,**-**,**c**,**d**,**/**,**e**,**f**

中序遍历序列：**a**,**+**,**b**,*****,**c**,**-**,**d**,**-**,**e**,**/**,**f**

后序遍历序列：**a**,**b**,**c**,**d**,**-**,*****,**+**,**e**,**f**,**/**,**-**



5.3 遍历二叉树与线索二叉树

遍历的递归算法

先序遍历 (DLR) 的定义:

若二叉树非空

- (1) 访问根结点;
- (2) 先序遍历左子树;
- (3) 先序遍历右子树;

上面先序遍历的定义等价于:

若二叉树为空, 结束——基本项 (也叫终止项)

若二叉树非空——递归项

- (1) 访问根结点;
- (2) 先序遍历左子树;
- (3) 先序遍历右子树。

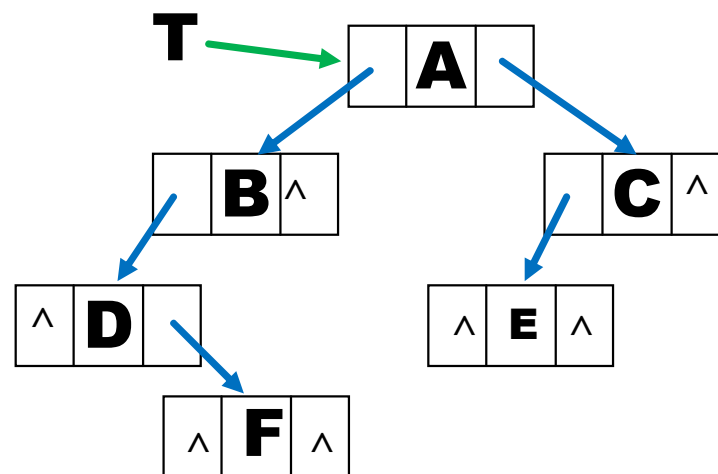
5.3 遍历二叉树与线索二叉树

1、先序遍历递归算法

```
void PreOrderTraverse ( BiTree T, Status ( * Visit ) ( ElemType e ) )
{ //采用二叉链表存贮二叉树, visit()是访问结点的函数
  //本算法先序遍历以T为根结点指针的二叉树
  if ( T ) { //若二叉树不为空
    Visit( T->data ); //访问根结点
    PreOrderTraverse(T->lchild, Visit); //先序遍历T的左子树
    PreOrderTraverse(T->rchild, Visit); //先序遍历T的右子树
  }
} //PreOrderTraverse
```

最简单的 Visit 函数是:

```
Status PrintElement( const TElemType e )
{ //输出元素e的值
  output( e ); return OK;
}
```



5.3 遍历二叉树与线索二叉树

先序遍历递归算法

```
void PreOrderTraverse ( BiTree T, Status ( * Visit ) ( ElemType e ) )
{ // 采用二叉链表存贮二叉树, visit()是访问结点的函数
  if ( T ) {
    if ( Visit( T->data ) ) // 如果访问根结点成功, 则继续
      if ( PreOrderTraverse(T->lchild, Visit) ) //左子树
        if ( PreOrderTraverse(T->rchild, Visit) ) //右子树
          return OK;
    return ERROR;
  }
  else return ok;
} // PreOrderTraverse
```

5.3 遍历二叉树与线索二叉树

2、中序遍历递归算法

```
void InOrderTraverse( BiTree T, Status ( * Visit ) ( ElemType e ) )
{ // 采用二叉链表存贮二叉树, visit()是访问结点的函数
  // 中序遍历以T为根结点指针的二叉树
  if ( T )
  { // 若二叉树不为空
    InOrderTraverse( T->lchild, Visit ); // 中序遍历T的左子树
    Visit ( T->data ); // 访问根结点
    InOrderTraverse( T->rchild, Visit ); // 中序遍历T的右子树
  }
} // InOrderTraverse
```

5.3 遍历二叉树与线索二叉树

3、后序遍历递归算法

```
void PostOrderTraverse( BiTree T , Status ( * Visit ) ( ElemType e ) )
{   // 采用二叉链表存贮二叉树, visit( )是访问结点的函数
    // 后序遍历以T为根结点指针的二叉树
    if ( T ) { // 若二叉树不为空
        PostOrderTraverse( T->lchild, Visit ); // 后序遍历左子树
        PostOrderTraverse( T->rchild, Visit ); // 后序遍历右子树
        Visit ( T->data );                      // 访问根结点
    }
} // PostOrderTraverse
```

5.3 遍历二叉树与线索二叉树

例1：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

```
void leaf ( BiTree T )
{ // 二叉链表存贮二叉树，计算二叉树的叶子结点个数
  // 先序遍历的过程中进行统计，初始全局变量 n=0
  if ( T )
  { if ( T->lchild==null && T->rchild==null )
      n += 1; //若T所指结点为叶子结点则计数
    else { leaf ( T->lchild );
          leaf ( T->rchild );
        }
  } // if
} // leaf
```

5.3 遍历二叉树与线索二叉树

例1：编写求二叉树的叶子结点个数的算法。

```
int Countleave( BiTree T )
{ // 采用二叉链表存贮二叉树，返回叶子结点的个数
  if ( !T ) return 0;
  if ( T->lchild==NULL && T->rchild==NULL )
    return 1;
  else
    Countleave( T->lchild ) + Countleave( T->rchild );
  return;
}
```

5.3 遍历二叉树与线索二叉树

例1：求二叉树的深度（后序遍历）

二叉树的深度=左右子树深度的最大值+1

基本思想：

若T为空树，则返回0；

否则，计算T的左子树深度，T的右子树深度，取最大值；

修改Visit函数：

取左子树深度和右子树深度的最大值，加1

5.3 遍历二叉树与线索二叉树

例1：求二叉树的深度（后序遍历）

```
int getDepth(Bitree T) { // 返回二叉树的深度
    if(!T) return 0;
    else {
        left_depth = getDepth(T->lchild);
        right_depth = getDepth(T->rchild);
        return 1+(left_depth>right_depth? left_depth:right_depth);
    }
}
```

5.3 遍历二叉树与线索二叉树

例3：以广义表的方式输出二叉树

若二叉树为空，输出空格 “ ” ；

Step1：输出根结点的数据；

Step2：判断根结点是否为叶子结点（左右子树为空）

Step3：若是叶子结点，则返回；

 若不是叶子结点，则：

 输出左括号 “ (” ；

 递归输出左子树；

 输出逗号 “ , ” ；

 递归输出右子树；

 输出右括号 “) ” ；

5.3 遍历二叉树与线索二叉树

例4：以字符串的形式建立二叉链表。
结果：二叉树的二叉链表。

是否可利用“遍历”，建立二叉链表的所有结点
并完成相应结点的链接？

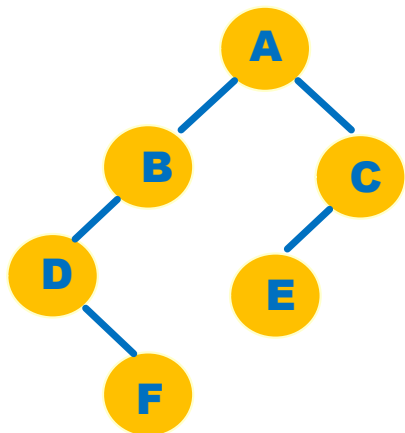
基本思想：

输入（在空子树处添加字符 * 的二叉树的）先序序列（设每个元素是一个字符）。
按先序遍历的顺序，建立二叉链表的所有结点并完成相应结点的链接。

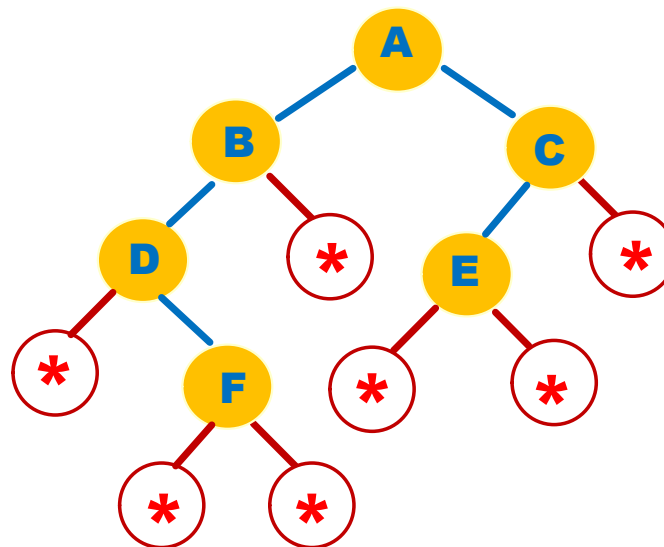
5.3 遍历二叉树与线索二叉树

对原来的二叉树进行扩充，在空子树处添加 *。

例4：建立二叉链表。



先序序列: **A B D F C E**



A B D * F * * * C E * * *

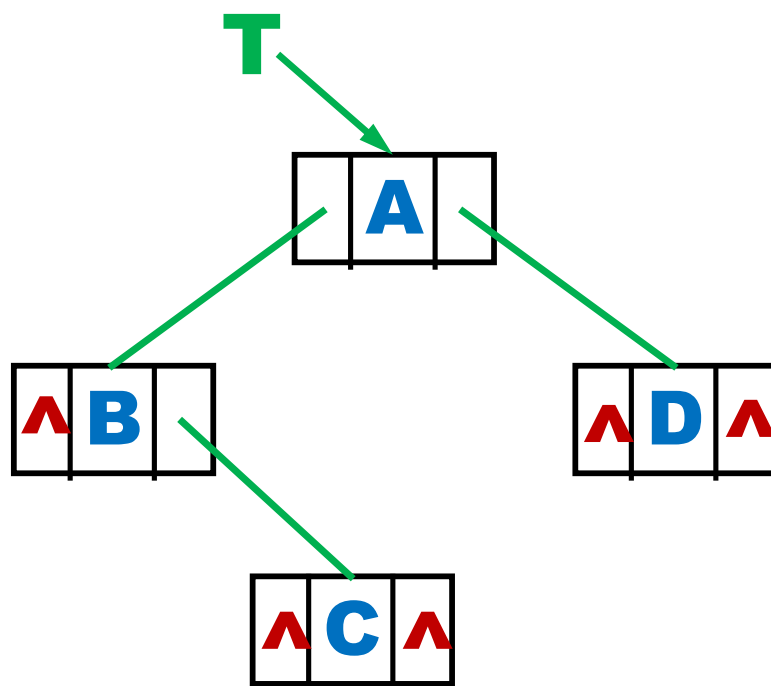
5.3 遍历二叉树与线索二叉树

```
Status CreateBiTree ( BiTree &T )
{ // 按先序遍历的顺序建立二叉链表
  scanf ( &ch ); if (ch== ";" ) reurn;
  if ( ch == '*' ) T=NULL; // 若ch== '*' 则表示空子树
  else {
    if ( ! (T=( BiTNode * ) malloc( sizeof( BiTNode ))) )
      exit( OVERFLOW );
    T->data = ch;           // 建立 (根) 结点
    CreateBiTree( T->lchild ); // 递归构造左子树链表
    CreateBiTree( T->rchild ); // 递归构造右子树链表
  }
  return OK;
} //CreateBiTree
```

5.3 遍历二叉树与线索二叉树

例4：建立二叉链表。

A B * C * * D * *

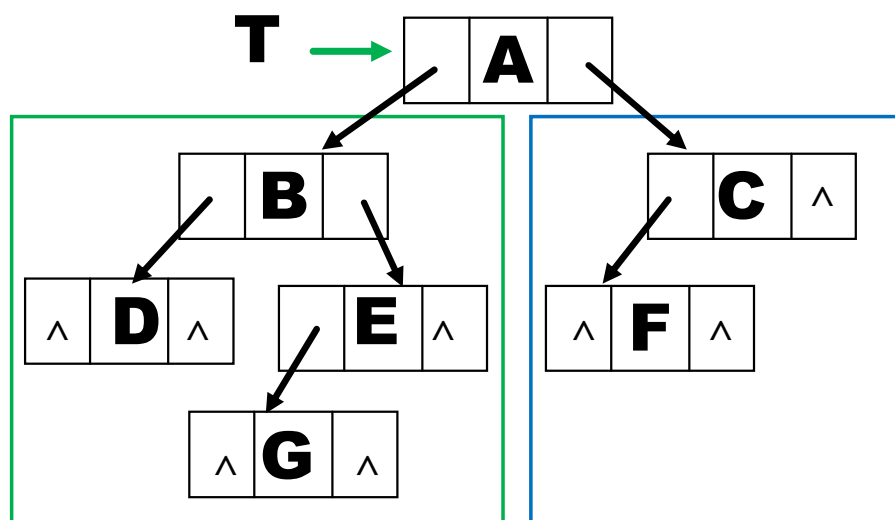


5.3 遍历二叉树与线索二叉树

例3：复制二叉链表。

输入：二叉链表

结果：复制的新二叉链表



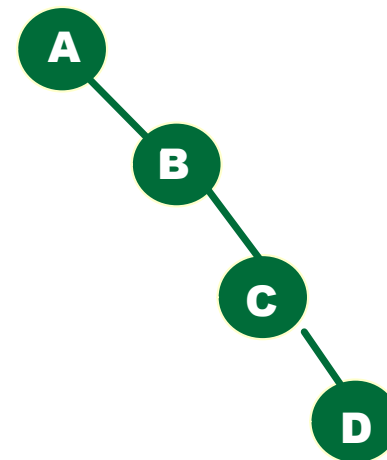
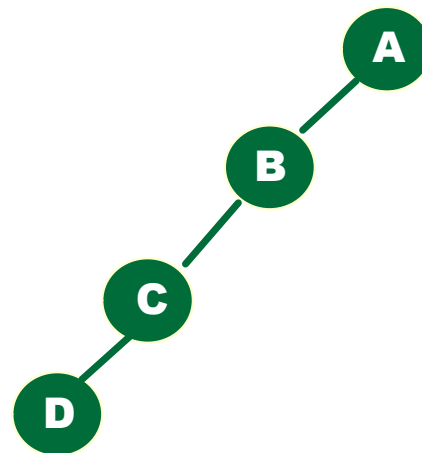
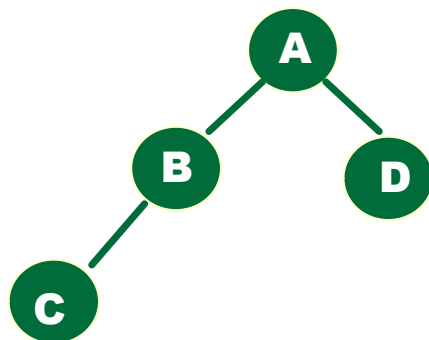
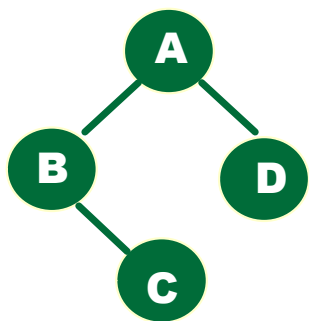
5.3 遍历二叉树与线索二叉树

例3：复制二叉链表。

```
void CopyBiTree( BiTree T, BiTree &newT )
{ // 采用后序遍历的框架，新二叉链表根为 newT
  if ( !T ) newT=NULL;
  else
  { CopyBiTree( T->lchild, plchild ); // 复制左子树
    CopyBiTree( T->rchild, prchild ); // 复制右子树
    newT = (BiTree) malloc( sizeof(BiTNode) );
    newT->data = T->data;      // 复制结点
    newT->lchild = plchild;    // 链接新结点的左子树
    newT->rchild = prchild;    // 链接新结点的右子树
  }
}
```


5.3 遍历二叉树与线索二叉树

已知某二叉树的先序序列为：ABCD，能还原该二叉树吗？

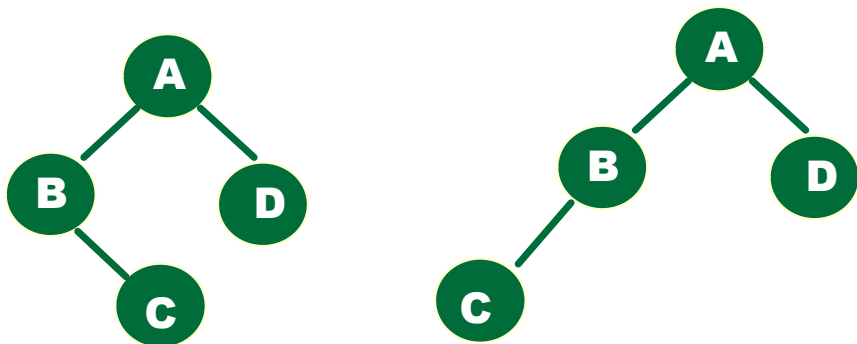


5.3 遍历二叉树与线索二叉树

已知某二叉树：

先序序列为：ABCD，

后序序列为：CBDA，能还原该二叉树吗？

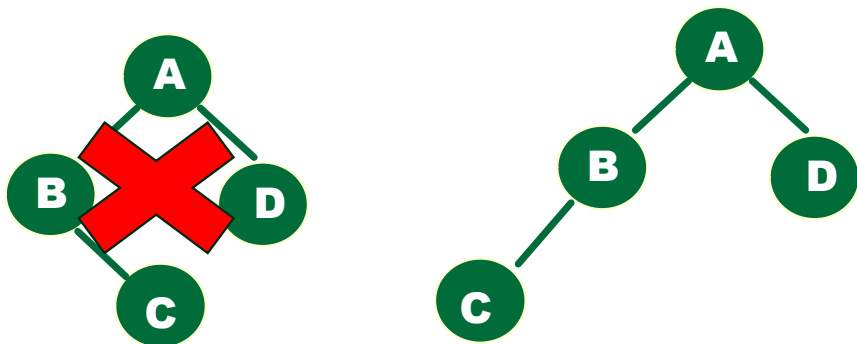


5.3 遍历二叉树与线索二叉树

已知某二叉树：

先序序列为：ABCD，

中序序列为：CBAD，能还原该二叉树吗？



5.3 遍历二叉树与线索二叉树

已知某二叉树，哪些序列，能还原二叉树：

	先序	中序	后序
先序	x	√	x
中序	√	x	√
后序	x	√	x

5.3 遍历二叉树与线索二叉树

遍历算法的举例

1. 查询二叉树中某个结点
2. 求二叉树的深度（后序遍历）
3. 判断二叉树相等
4. 建立二叉树的存储结构（给出全部左右子树）
5. 由二叉树的先序序列和中序序列建立二叉树

5.3 遍历二叉树与线索二叉树

由二叉树的先序序列和中序序列建立二叉树

```
void createBT(BiTree &T,char pre[],char ino[],int ps,int is,int n)
```

函数参数说明：

T, 构建的二叉树 (引用)

pre[], 先序遍历序列的字符数组,

ino[], 中序遍历序列的字符数组,

ps, 先序遍历数组的下标;

is, 中序遍历数组的下标;

n, 字符数组的长度, (pre数组和ino数组等长)

5.3 遍历二叉树与线索二叉树

由二叉树的先序序列和中序序列建立二叉树

```
void createBT(BiTree &T, char pre[], char ino[], int ps, int is, int n)
```

编写思路：

Step1: 取出pre中的第一个字符A，并在ino序列中查找对应的字符的位置k；若找不到，则报错；

Step2: 1) 建立根结点

2) 递归建立左右子结点；



5.3 遍历二叉树与线索二叉树

Step2: 1) 建立根结点

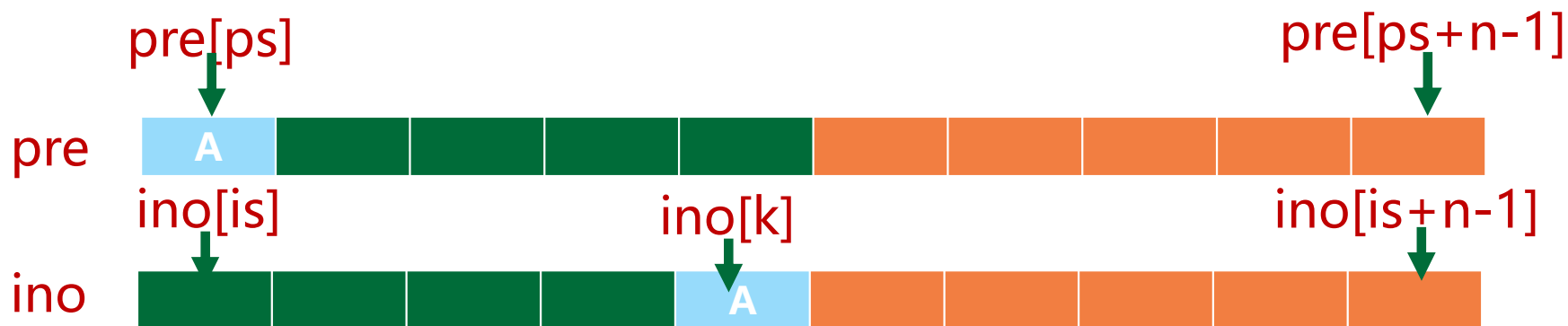
2) 递归建立左右子结点;

分三种情况讨论, 确定函数调用的参数:

i: 左右子树都存在;

ii: 无左子树

iii: 无右子树

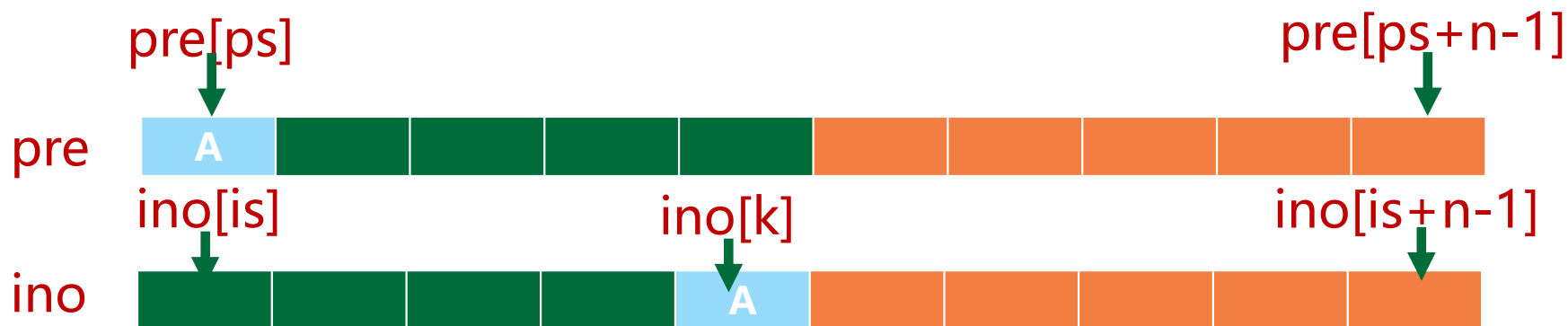


5.3 遍历二叉树与线索二叉树

递归建立左右子结点；分三种情况讨论，确定函数调用的参数：

i: 左右子树都存在；

	长度	在pre中的起点	在ino中的起点
左子树	$k-is$	$pre[ps+1]$	$ino[is]$
右子树	$n-(k-is)-1$	$pre[ps+(k-is)+1]$	$ino[k+1]$



5.3 遍历二叉树与线索二叉树

递归建立左右子结点；分三种情况讨论，确定函数调用的参数：

ii: 无左子树；条件是 $k == is$

	长度	在pre中的起点	在ino中的起点
左子树	$k == is$		
右子树	$n - (k - is) - 1$	$pre[ps + (k - is) + 1]$	$ino[k + 1]$
	$n - 1$	$pre[ps + 1]$	$ino[k + 1]$

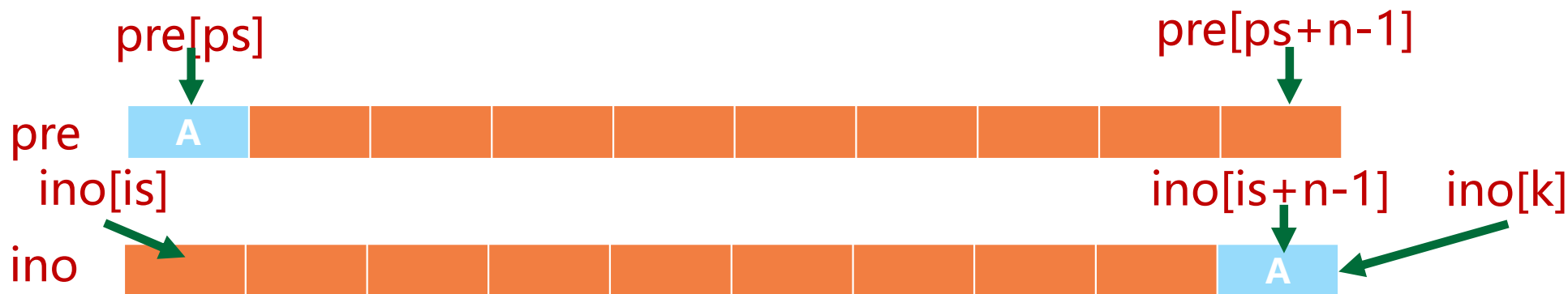


5.3 遍历二叉树与线索二叉树

递归建立左右子结点；分三种情况讨论，确定函数调用的参数：

iii: 无右子树；条件是 $k == is + n - 1$

	长度	在pre中的起点	在ino中的起点
左子树	$k - is$	$pre[ps + 1]$	$ino[is]$
	$n - 1$	$pre[ps + 1]$	$ino[is]$
右子树	$k == is + n - 1$		



5.3 遍历二叉树与线索二叉树

由二叉树的先序序列和中序序列建立二叉树

```
void createBT(BiTree &T,char pre[],char ino[],int ps,int is,int n) {  
    if (n==0) T=NULL;  
    else {  
        k = Search(ino,pre[ps]);  
        if (k==-1) T=NULL;  
        else {  
            建立新结点, 递归建立左右子树  
        }  
    }  
}
```

5.3 遍历二叉树与线索二叉树

建立新结点，递归建立左右子树

```
T = (BiTNode *) malloc (sizeof (BiTNode) );
```

```
T->data = pre[ps]
```

```
if (k==is) T->lchild = NULL; //无左子树
```

```
else
```

```
    createBT(BiTree &T,char pre[],char ino[],ps+1,is,k-is);
```

```
if (k==is+n-1) T->rchild = NULL; //无右子树
```

```
else
```

```
    createBT(BiTree &T,char pre[],char ino[],ps+1+(k-is),k+1,n-(k-is)-1;
```

5.3 遍历二叉树与线索二叉树

遍历的非递归算法

栈是实现递归的最常用的结构。

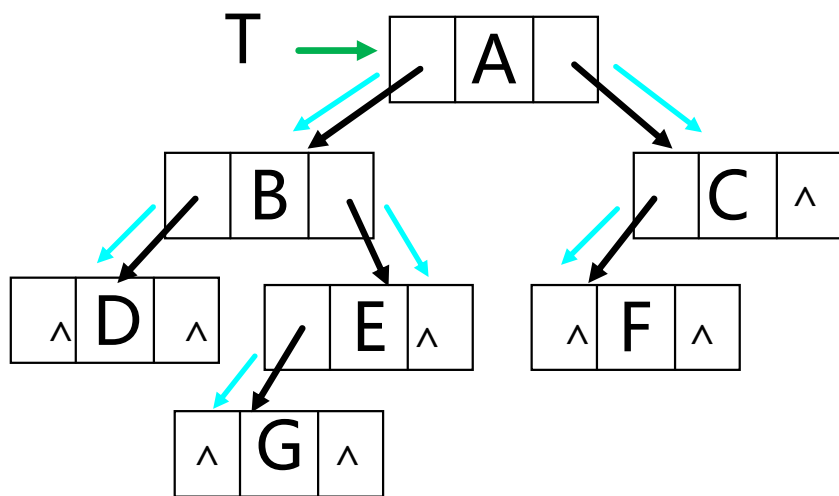
利用一个栈来记下尚待遍历的结点或子树，以备以后访问。

5.3 遍历二叉树与线索二叉树

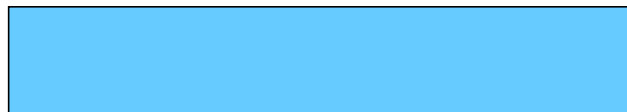
中序遍历的非递归算法：

中序遍历的第一个结点 二叉树的最左下结点。

当前结点的后继结点 若当前结点有右子树，**后继结点**为右子树的最左下结点；否则**后继结点**为栈顶结点。



D B G **E** **A** F C



5.3 遍历二叉树与线索二叉树

中序遍历的非递归算法：

中序遍历的访问顺序：

- 1、从根结点开始，依次进入左子树，并将所经过的所有结点压入栈，知道当前结点的左子树为空第一个结点；
- 2、从栈中弹出一个结点N，进行访问；
- 3、若N结点有右子树，则处理其右子树，从右子树的根结点开始继续遍历；

5.3 遍历二叉树与线索二叉树

中序遍历的非递归算法：

```
Status InTrav( BiTree T, void(* Visit)(TelemType e) )
{ InitStack(S);  p = T;
  while ( p || ! StackEmpty(S) ) // 树非空 或 栈非空
  { if ( p ) // 若树非空
    { Push(S, p); p = p->lchild; }
    // p有左子树则p结点入栈，循环入栈，一直找到最左下结点
  else // (最左下结点) 退栈，访问
    { Pop (S, p); Visit( p->data );
      p = p->rchild; // p指向右子树
    }
  } // while
  DesrroyStack(S);
  return OK;
} // InTrav
```

5.3 遍历二叉树与线索二叉树

先序遍历的非递归算法：

```
Status PreOrderTrav( BiTree T, void(* Visit)(TelemType e) )
{ InitStack(S);  p = T;
  while ( p || ! StackEmpty(S) ) // 树非空 或 栈非空
  { while( p ) // 若树非空
    { visit(p->data);
      if (p->rchild!=NULL) push(S,p->rchild);
      p=p->lchild;
    }
    if (!stackEmpty(S)) pop(S,p); //退栈
  } // while
  DesrroyStack(S);
  return OK;
} // PreOrderTrav
```

5.3 遍历二叉树与线索二叉树

后序遍历的非递归算法：

```
Status PostOrderTrav( BiTree T, void(* Visit)(TelemType e) )
{ InitStack(Stack1); InitStack(Stack2);
  p=T; if( p) push(stack1, p); else return NULL;
  while (!isStackEmpty(stack1)) {
    TreeNode *node = pop(stack1);    push(stack2, node);
    if (node->left) {    push(stack1, node->left);    }
    if (node->right) {    push(stack1, node->right);    }
  }
  while (!isStackEmpty(stack2)) {
    TreeNode *node = pop(stack2); visit(node);
  }
  return OK;
} // PostOrderTrav
```

5.3 遍历二叉树与线索二叉树

按层次遍历二叉树:

按层次遍历二叉树, 需要借助一个队列, 在访问某一层结点时, 先将下一层结点指针预先送进队列中, 利用队列逐层访问;

```
Status levelOrderTrav( BiTree T, void(* Visit)(TelemType e) )
{
    initQueue(Q); p = T;
    Enqueue (Q, p);
    while (!isEmptyQueue(Q)) {
        p = Dequeue(Q);    visit(p);
        if (p->left) {Enqueue (Q, p->left);    }
        if (p->right) {Enqueue (Q, p->right); }
    }
    return OK;
} // PostOrderTrav
```

5.3 遍历二叉树与线索二叉树

算法	时间复杂度	空间复杂度
先/中/后序遍历的递归算法	$O(n)$	最坏: $O(n)$ 平均: $O(\log n)$
先/中/后序遍历的非递归算法	$O(n)$	最坏: $O(n)$ 平均: $O(\log n)$
层次遍历算法	$O(n)$	取决于树的宽度 最坏: $O(n)$

5.3 遍历二叉树与线索二叉树

线索二叉树

遍历二叉树的结果可求得结点的一个**线性序列**。

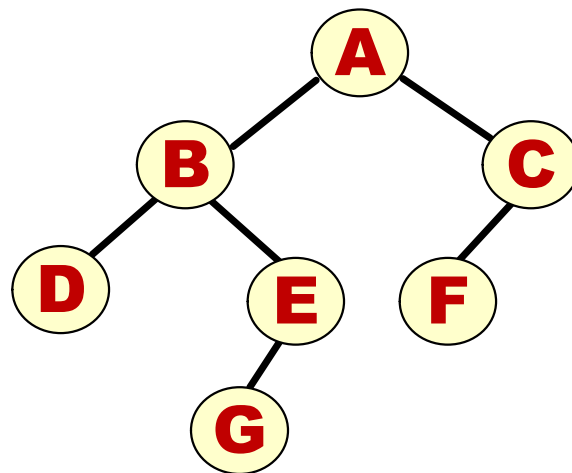
指向线性序列中的“前趋”和“后继”的指针，称作“**线索**”。

例如：

先序序列：A B D E G C F

中序序列：D B G E A F C

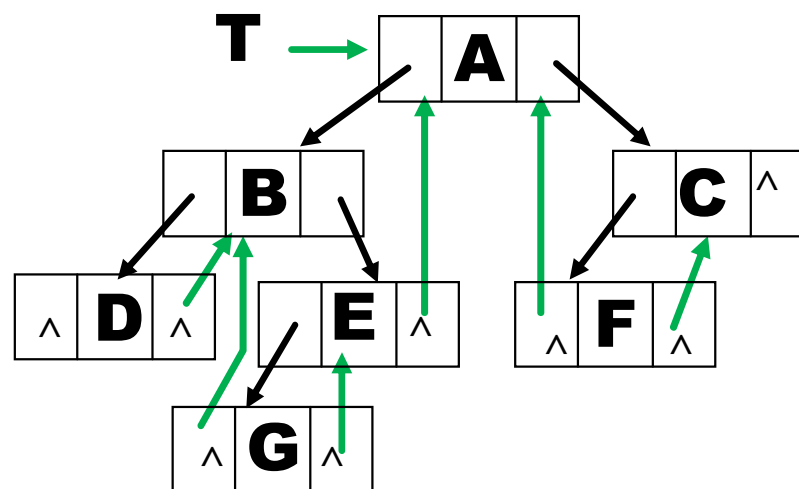
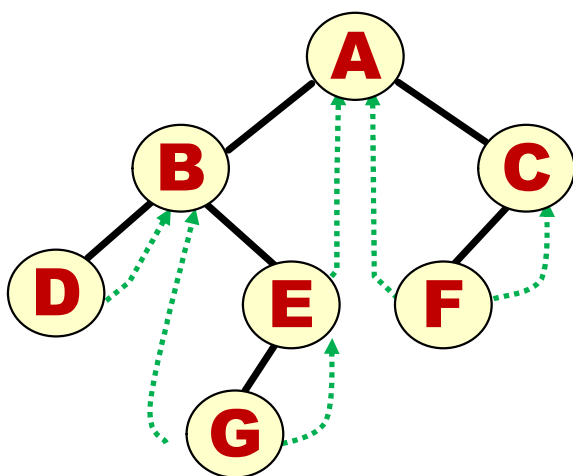
后序序列：D G E B F C A



5.3 遍历二叉树与线索二叉树

线索二叉树

如何在二叉链表中保存线索？——借用结点的空链域保存线索



中序序列：D B G E A F C

包含“线索”的存储结构，称作“线索链表”。

5.3 遍历二叉树与线索二叉树

线索链表中的结点

在二叉链表的结点中增加两个标志域Ltag, Rtag:

ltag(p)= **0** 表示 **lchild(p)** 为指向左孩子的指针
1 表示 **lchild(p)** 为指向直接前驱的线索

rtag(p)= **0** 表示 **rchild(p)** 为指向右孩子的指针
1 表示 **rchild(p)** 为指向直接后继的线索

lchild	ltag	data	rtag	rchild

5.3 遍历二叉树与线索二叉树

线索链表的类型说明

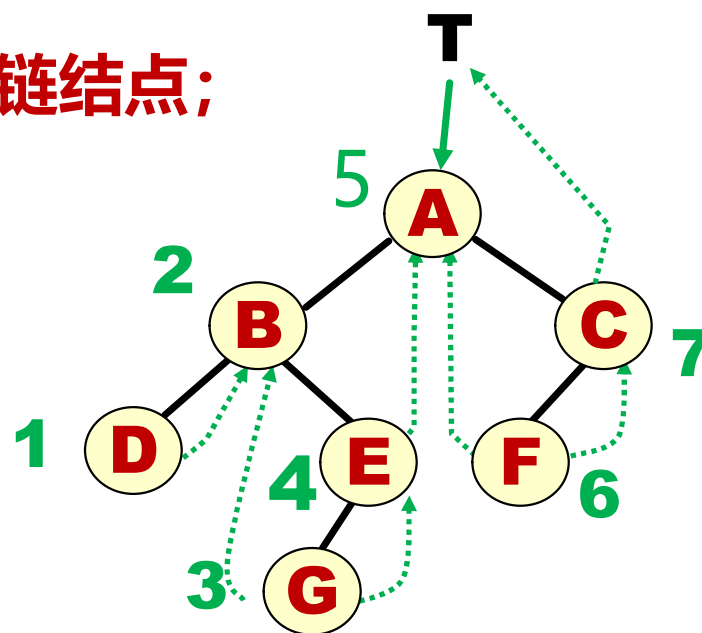
```
typedef enum{ link, thread } PointerTag;  
                                     // link=0, thread=1  
typedef struct BiThrNode {  
    TElemType      data;  
    struct BiThrNode *lchild, *rchild;  
    PointerTag  Ltag, Rtag; //左、右标志域  
} BiThrNode, * BiThrTree;
```

lchild	ltag	data	rtag	rchild

5.3 遍历二叉树与线索二叉树

线索二叉树的遍历(中序线索二叉树)

- ◆ 中序遍历的**第一个结点**
二叉树的最左下结点
- ◆ 当前结点的**后继结点**
 若结点的右链域为线索, 则**后继结点为右链结点**;
 否则, **后继结点为右子树的最左下结点**

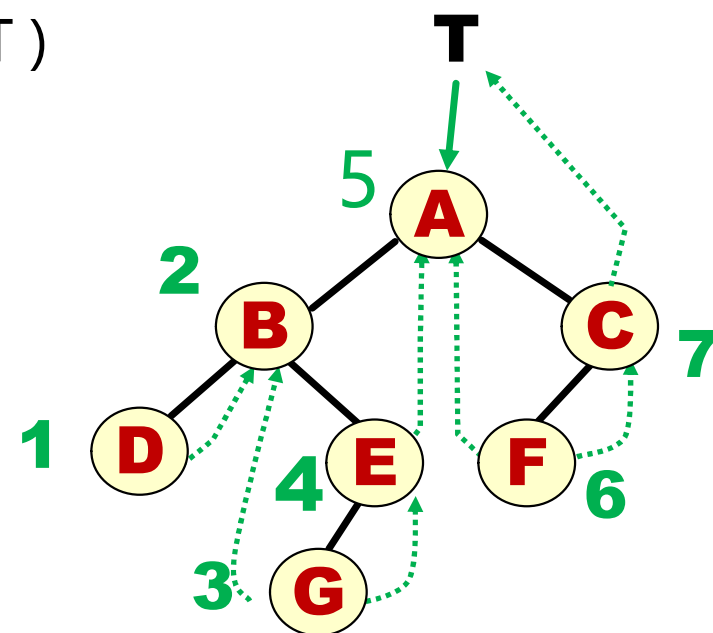


5.3 遍历二叉树与线索二叉树

```

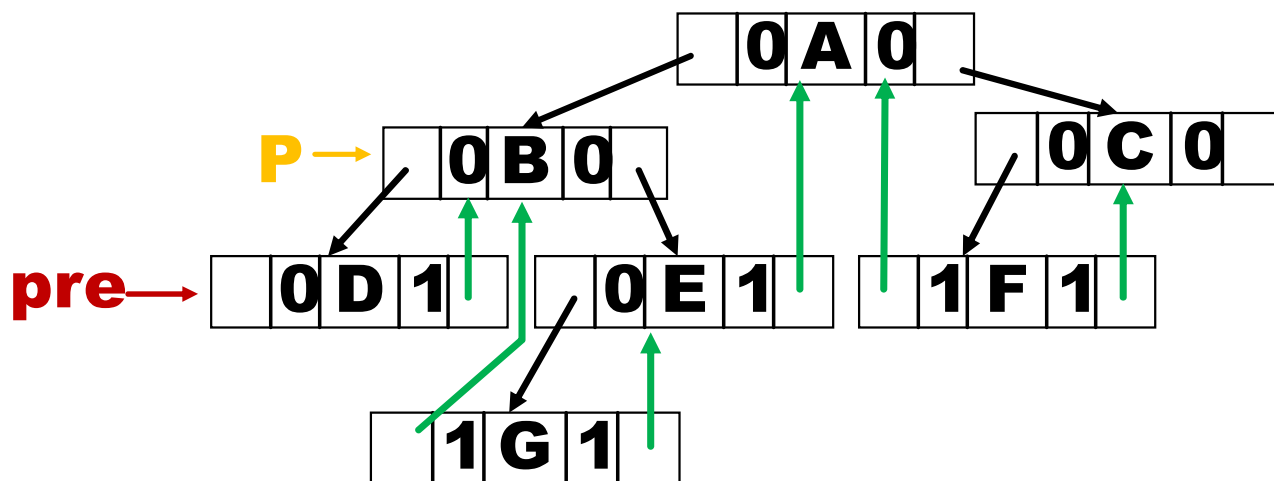
Status InTra_ThrT ( BiThrTree ThrT, void(*Visit) (TElemType e) )
{ p = ThrT -> lchild;    // p指向根结点, pre是全局变量
  while ( p != ThrT )
  { while ( p->Ltag==link ) p=p->lchild; // 最左下结点
    Visit( p->data );    // p->Ltag== thread
    while ( p->Rtag==Thread && p->rchild!=ThrT )
    { // 若右孩子域是线索
      p = p->rchild; Visit( p->data );
    }
    p = p->rchild; // 若右孩子域不是线索
  }
  return OK;
} // InTra_ThrT

```



5.3 遍历二叉树与线索二叉树

- 二叉树的线索化(建立中序线索二叉树)
 - ◆ 在中序遍历过程中为二叉树结点加线索
 - ◆ 增加指针 **pre**、**p**, 保持指针 **p** 指向当前访问的结点, **pre** 指向当前访问结点的前趋。



5.3 遍历二叉树与线索二叉树

```

Status InOrderThreading( BiThrTree & Thrt, BiThrTree T )
{ if ( Thrt = (BiThrTree) malloc( sizeof(BiThrNode) )
  exit ( OVERFLOW );
  Thrt->Ltag = Link;           // 0
  Thrt->Rtag = Thread;         // 1. 建空树的头结点, 将左孩子指向根
  Thrt->rchild = Thrt;         // 右指针指向自己
  if (!T) Thrt->lchild = Thrt; // 若二叉树为空, 则左指针回指
  else {
    Thrt->lchild = T ; pre = Thrt;
    Inthreading(T);
    pre->rchild = Thrt;         // 最后一个结点线索化
    pre->Rtag = Thread;
    Thrt->rchild = pre;         // 指向整个树的最后一个节点
  }
  return OK;
} // InThreading

```



5.3 遍历二叉树与线索二叉树

```
void InThreading( BiThrTree p ) // 中序线索化二叉树
{ // pre为全局变量, 初值为NULL
  if ( p )
  { InThreading( p->lchild ); // 左子树线索化
    if ( p->lchild == NULL ) { // 为当前结点加前趋线索
      p->Ltag = Thread; p->lchild = pre;
    }
    if ( pre->rchild == NULL ) { // 为前趋结点加后继线索
      pre->Rtag = Thread; pre->rchild = p;
    }
    pre = p; // pre指向p
    InThreading( p->rchild );
  } // if
} // InThreading
```

5.4 树和森林

树的存贮结构

1、双亲表示法

通过保存树每个结点的双亲结点的位置，来表示树中结点之间的结构关系。

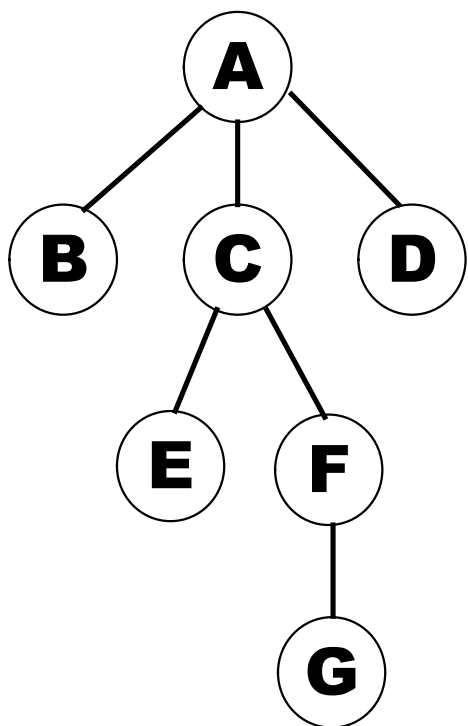
5.4 树和森林

- 双亲表示的类型定义

```
#define MAX_TREE_SIZE 100
typedef struct PTNode
{  ElemType  data;
   int  parent;  // 双亲位置域
} PTNode;
typedef struct
{  PTNode  nodes [ MAX_TREE_SIZE ];
   int  r, n;      // r为根的位置, n为结点数
} Ptree;
```


5.4 树和森林

双亲表示法



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

r=0
n=7

5.4 树和森林

树的存储结构

2、孩子表示法

通过保存树中每个结点的孩子结点的位置，表示树中结点之间的结构关系。

5.4 树和森林

方法1：多重链表（类似二叉链表，采用多叉结点）
两种方式：定长结点和不定长结点

定长结点：优点是结点结构一致，便于实现树的操作。
缺点是浪费一些内存空间。

data	child1	child2	childd
------	--------	--------	-------	--------

不定长结点：优点是节省内存空间。
缺点是不定长的结点会使一些操作实现变复杂。

data	degree	child1	child2	childd
------	--------	--------	--------	-------	--------

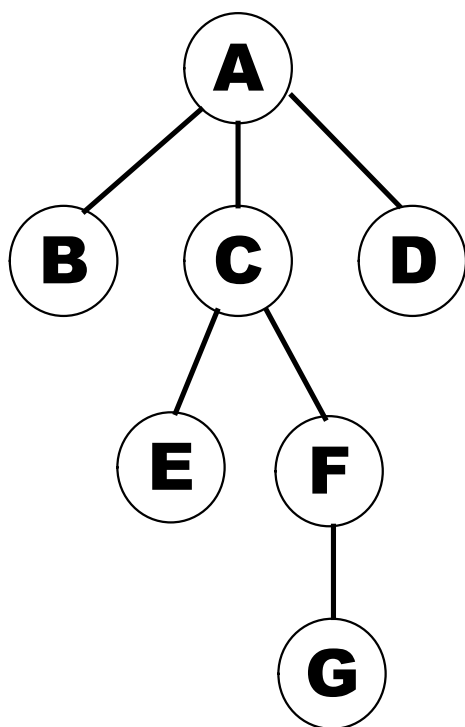
5.4 树和森林

方式2：孩子链表

将树中的每个结点的孩子排列起来，看成一个线性表，采用线性链表进行存贮。

5.4 树和森林

孩子链表表示法



0
1
2
3
4
5
6

data firstchild

A		→ 1 → 2 → 3 ^
B	^	
C		→ 4 → 5 ^
D	^	
F		→ 6 ^
E	^	
G	^	

同一结点的
孩子构成链表

根的位置
和结点数

r=0
n=7

5.4 树和森林

- 树的孩子链表类型定义

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

// 孩子结点

```
typedef struct
{
    ElemType data;
    ChildPtr firstchild;
} CTBox;
```

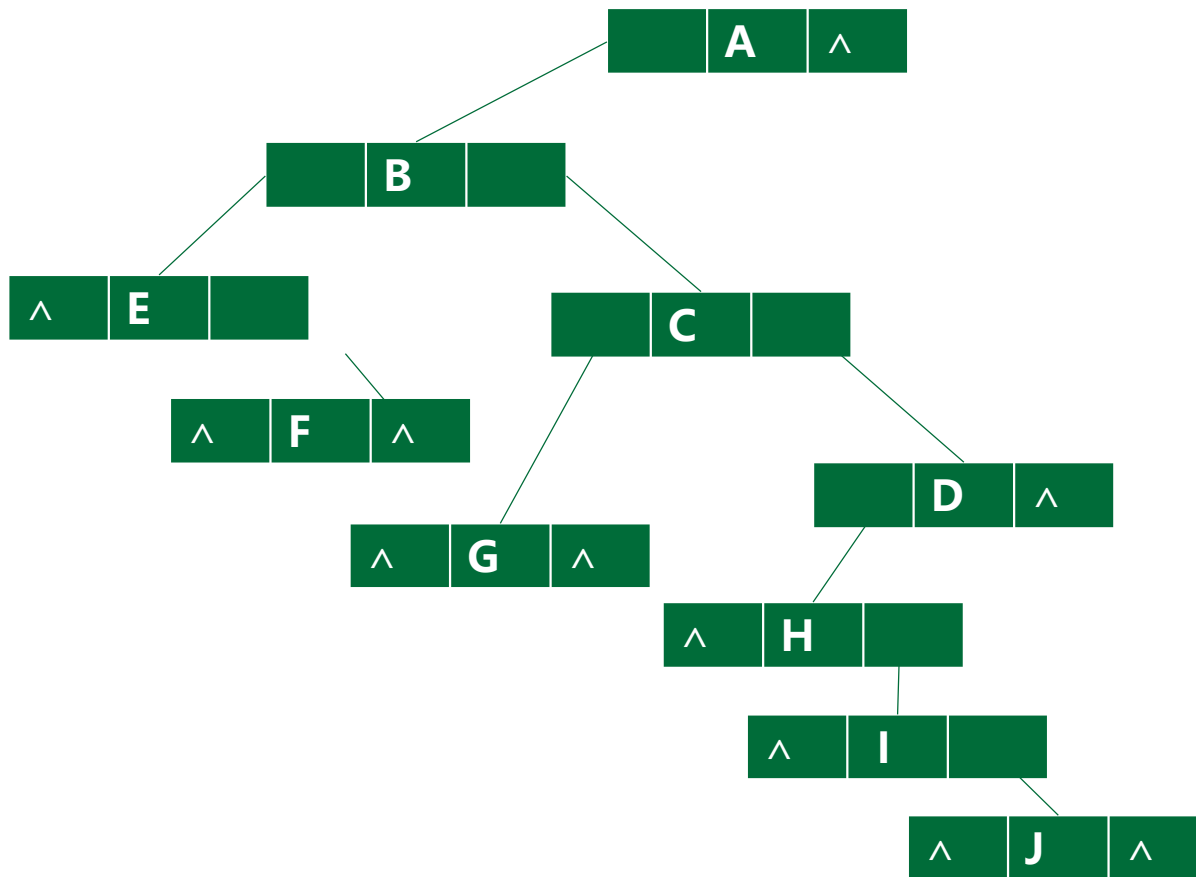
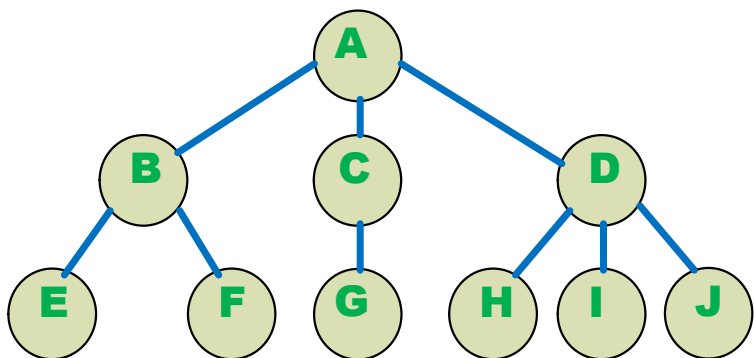
// 孩子链表头指针

```
typedef struct
{
    CTBox nodes [ MAX_TREE_SIZE ];
    int n , r;
} CTree;
```

// 结点数和根的位置

5.4 树和森林

3、孩子兄弟表示法



5.4 树和森林

3、孩子兄弟表示法

用二叉链表作为树的存贮结构。

孩子兄弟表示法类型定义

```
typedef struct CSNode  
{
```

```
    ElemType data;
```

```
    struct CSNode * firstchild,    //指向第一个孩子
```

```
    struct CSNode * nextsibling; //指向下一个兄弟
```

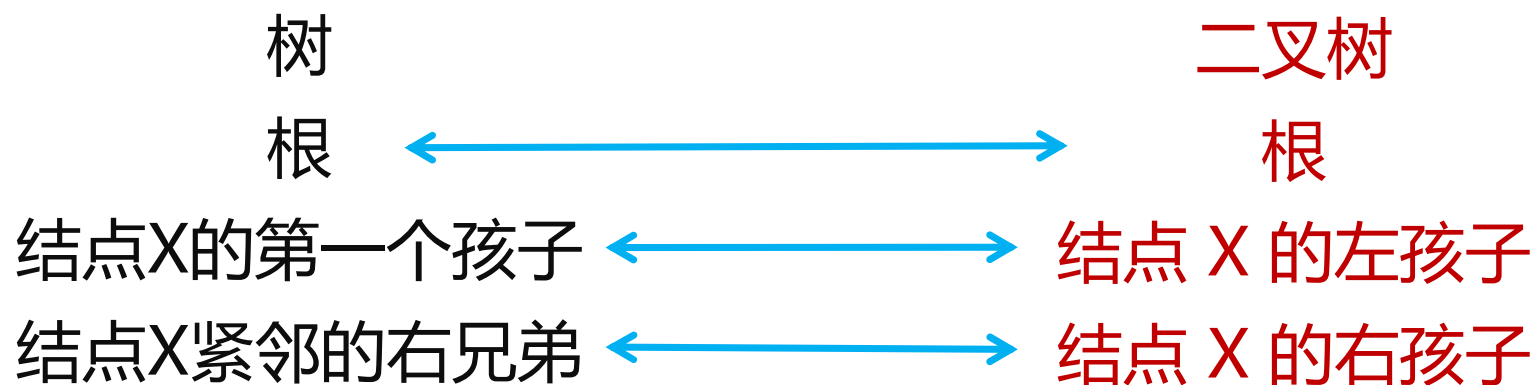
```
} CSNode, *CSTree;
```


5.4 树和森林

- 树与二叉树

二叉树与树都可用二叉链表存贮，以二叉链表作中介，可实现树与二叉树之间的转换。

树与二叉树转换方法



5.4 树和森林

树
根

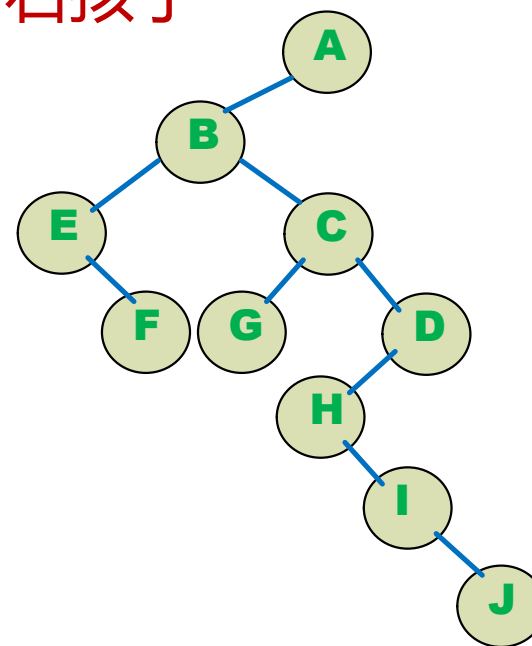
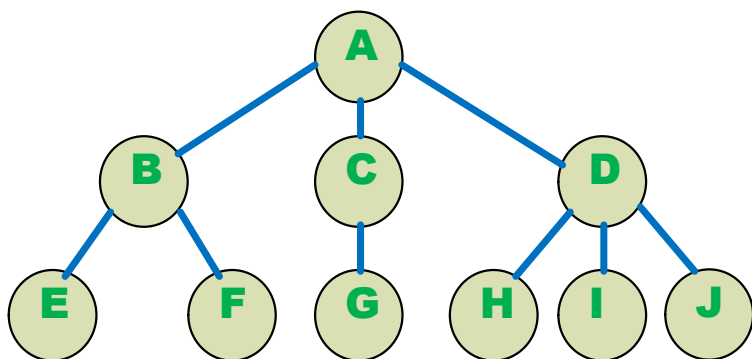
二叉树
根

结点X的第一个孩子

结点 X 的左孩子

结点X紧邻的右兄弟

结点 X 的右孩子

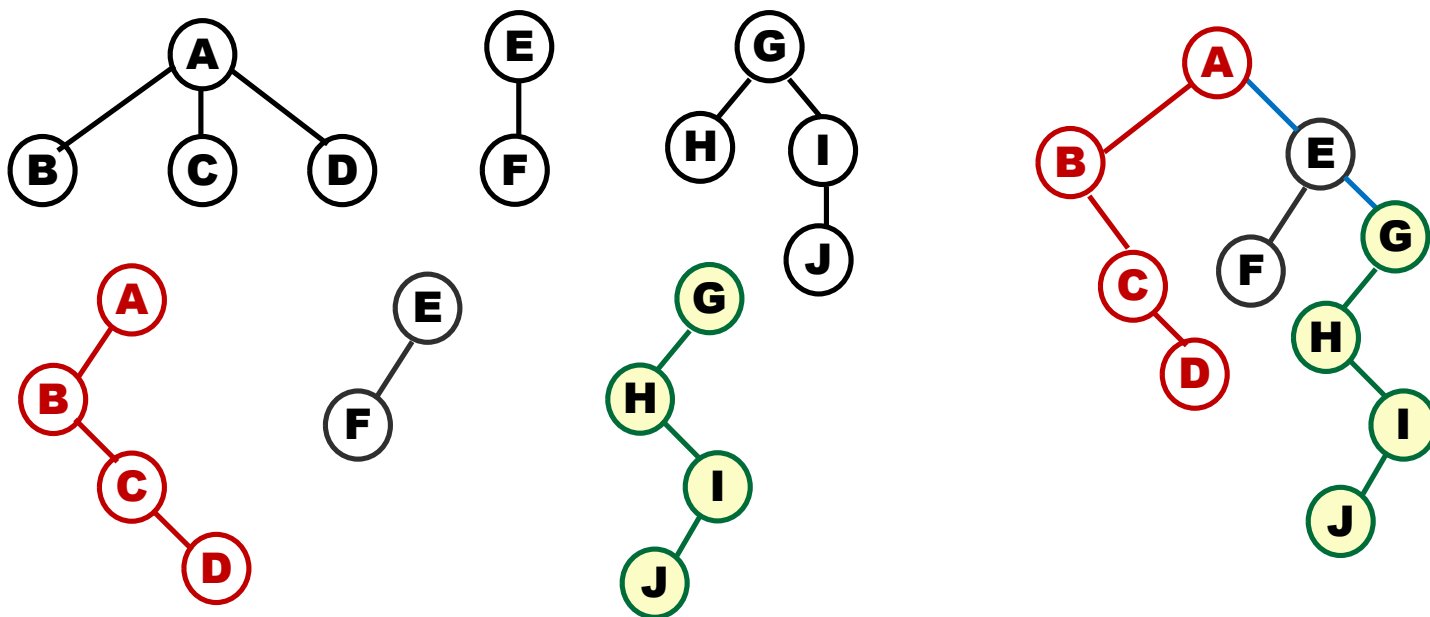


5.4 树和森林

- 森林与二叉树的转换

森林：树的集合

将森林中树的根看成兄弟，用树与二叉树的转换方法，进行森林与二叉树转换。



5.4 树和森林

树的遍历

遍历——按一定规律走遍树的各个顶点，且使每一顶点仅被访问一次，即找一个完整而有规律的走法，以得到树中所有结点的一个线性排列。

常用方法

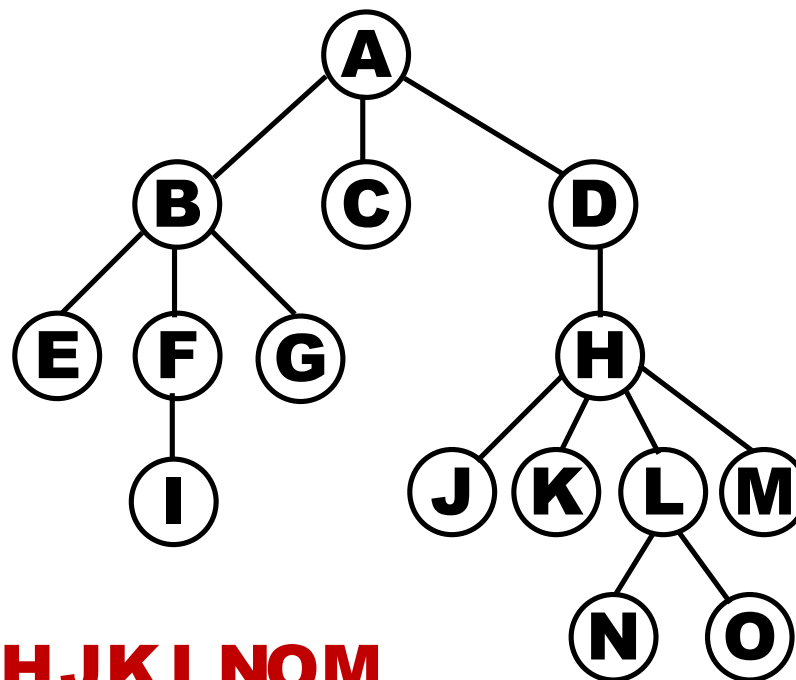
先根（序）遍历：先访问树的根结点，然后依次先根遍历根的每棵子树。

后根（序）遍历：先依次遍历每棵子树，然后访问根结点。

按层次遍历：先访问第一层上的结点，然后依次遍历第二层，……第n层的结点。

5.4 树和森林

树的遍历实例



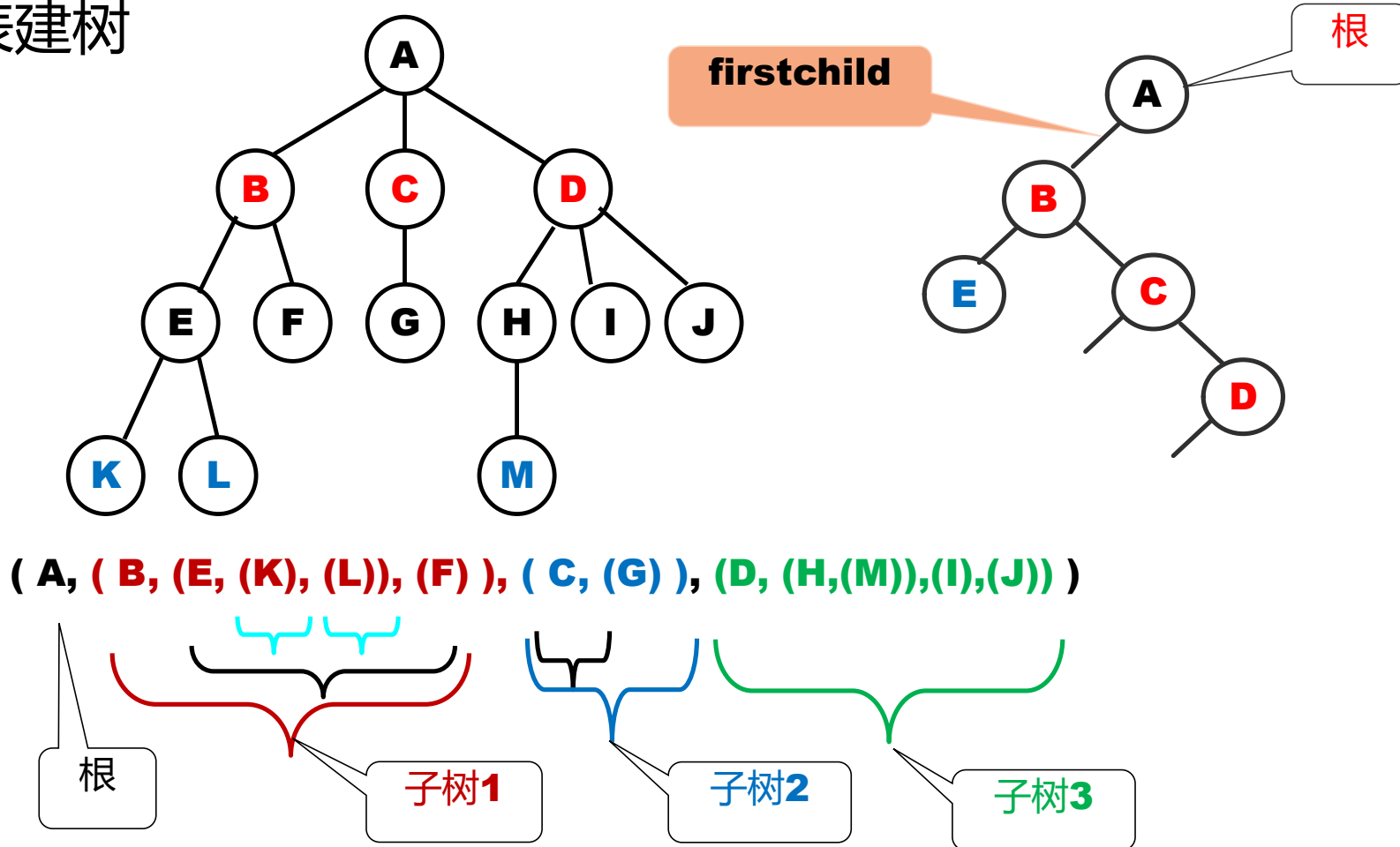
先序遍历: **A B E F I G C D H J K L N O M**

后序遍历: **E I F G B C J K N O L M H D A**

层次遍历: **A B C D E F G H I J K L M N O**

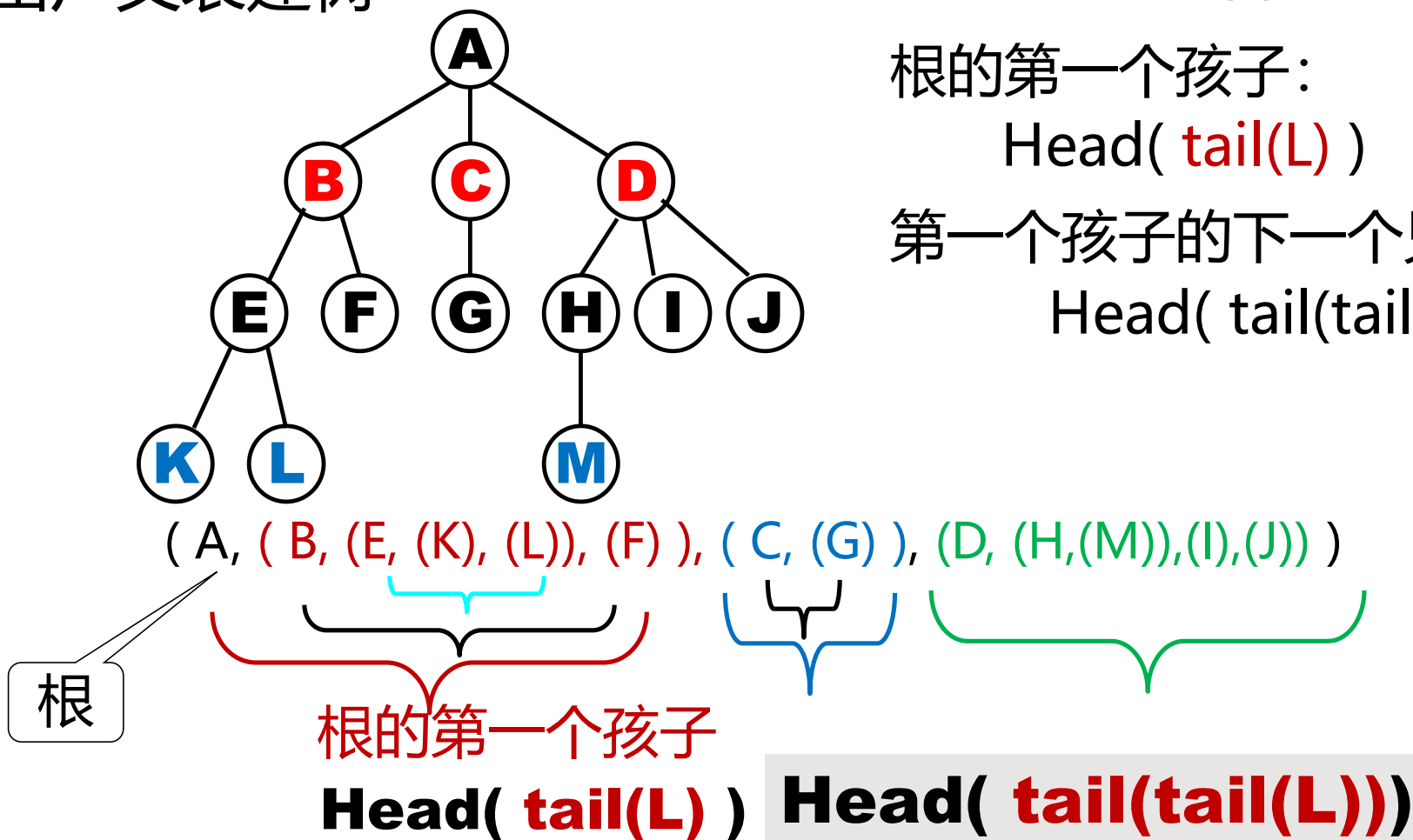
5.4 树和森林

- 例：由广义表建树



5.4 树和森林

- 例：由广义表建树



树根: $\text{Head}(\text{L})$

根的第一个孩子:

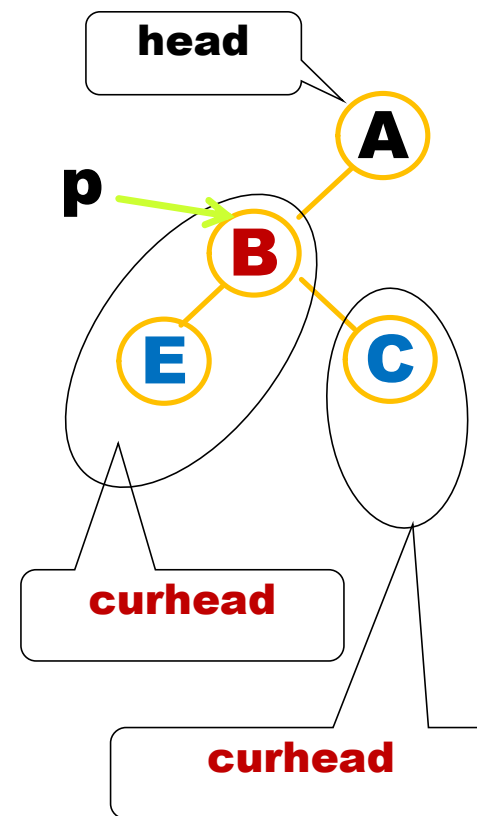
$\text{Head}(\text{tail}(\text{L}))$

第一个孩子的下一个兄弟:

$\text{Head}(\text{tail}(\text{tail}(\text{L})))$

5.4 树和森林

```
void CreateTree( Tree &t, GList L )
{ Create root node t from head( L );
  if ( tail(L) is not empty )
  { curhead = head( tail(L) ); //表尾的头
    curtail = tail( tail(L) ); //表尾的表尾
    CreateTree( t->firstchild, curhead );
    p = t->firstchild;
    while ( curtail ) // 若还有子树(表尾非空)
    { curhead = head( curtail ); // 下一个兄弟
      curtail = tail( curtail );
      CreateTree( p->nextsibling, curhead );
      p = p->nextsibling;
    }
    p->nextsibling = NULL;
  }
  return;
}
```



$$T(N) = O(N)$$

5.5 最优二叉树 (赫夫曼树)

路径:

从一个祖先结点到子孙结点之间的分支构成这两个结点间的路径;

路径长度:

路径上的分支数目称为路径长度;

结点的权:

给树中结点所赋的具有物理意义的值;

结点的带权路径长度:

从根到该结点的路径长度与该结点权的乘积。

5.5 最优二叉树 (赫夫曼树)

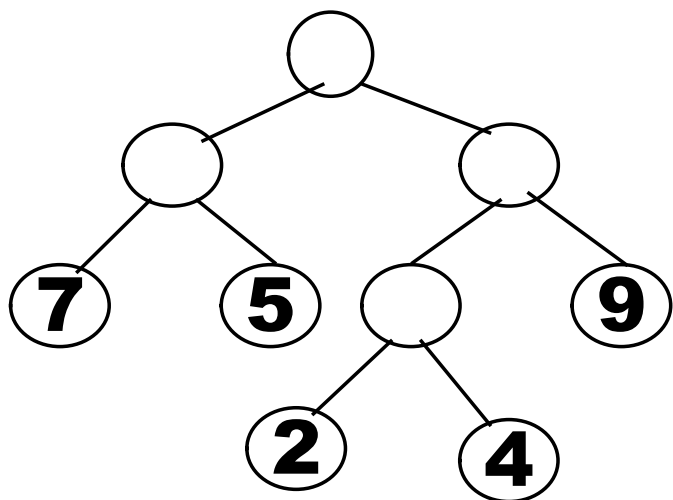
树的带权路径长度=树中所有叶子结点的带权路径之和,通常记作:

$$WPL = \sum_{k=1}^n W_k L_k$$

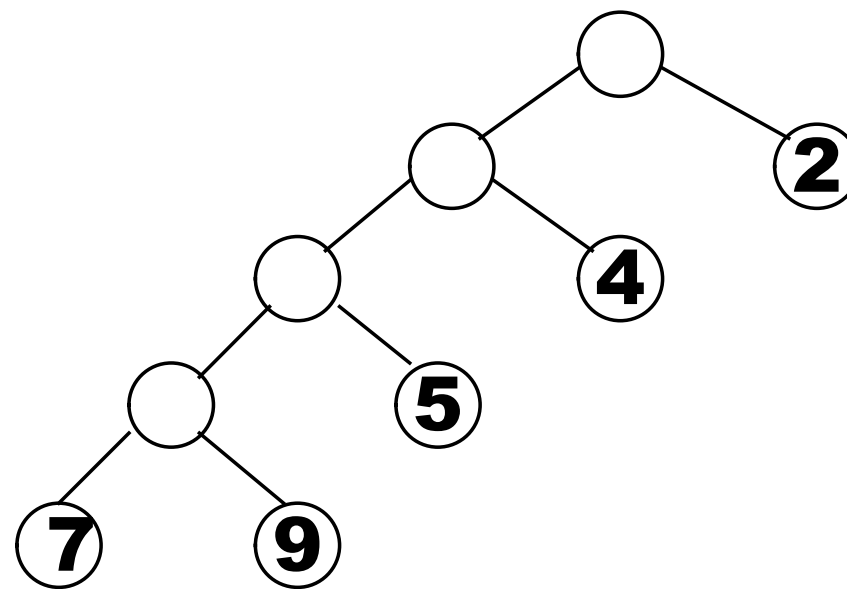
哈夫曼树: 假设有n个权值 (w_1, w_2, \dots, w_n) , 构造有 n 个叶子结点的二叉树, 每个叶子结点有一个 w_i 作为它的权值。则带权路径长度最小的二叉树称为哈夫曼树。

5.5 最优二叉树 (赫夫曼树)

• 实例



$$\begin{aligned} \text{WPL}(T) &= \\ 7 * 2 + 5 * 2 + 2 * 3 + 4 * 3 + 9 * 2 \\ &= 60 \end{aligned}$$



$$\begin{aligned} \text{WPL}(T) &= \\ 7 * 4 + 9 * 4 + 5 * 3 + 4 * 2 + 2 * 1 \\ &= 89 \end{aligned}$$

5.5 最优二叉树（赫夫曼树）

构造Huffman树步骤

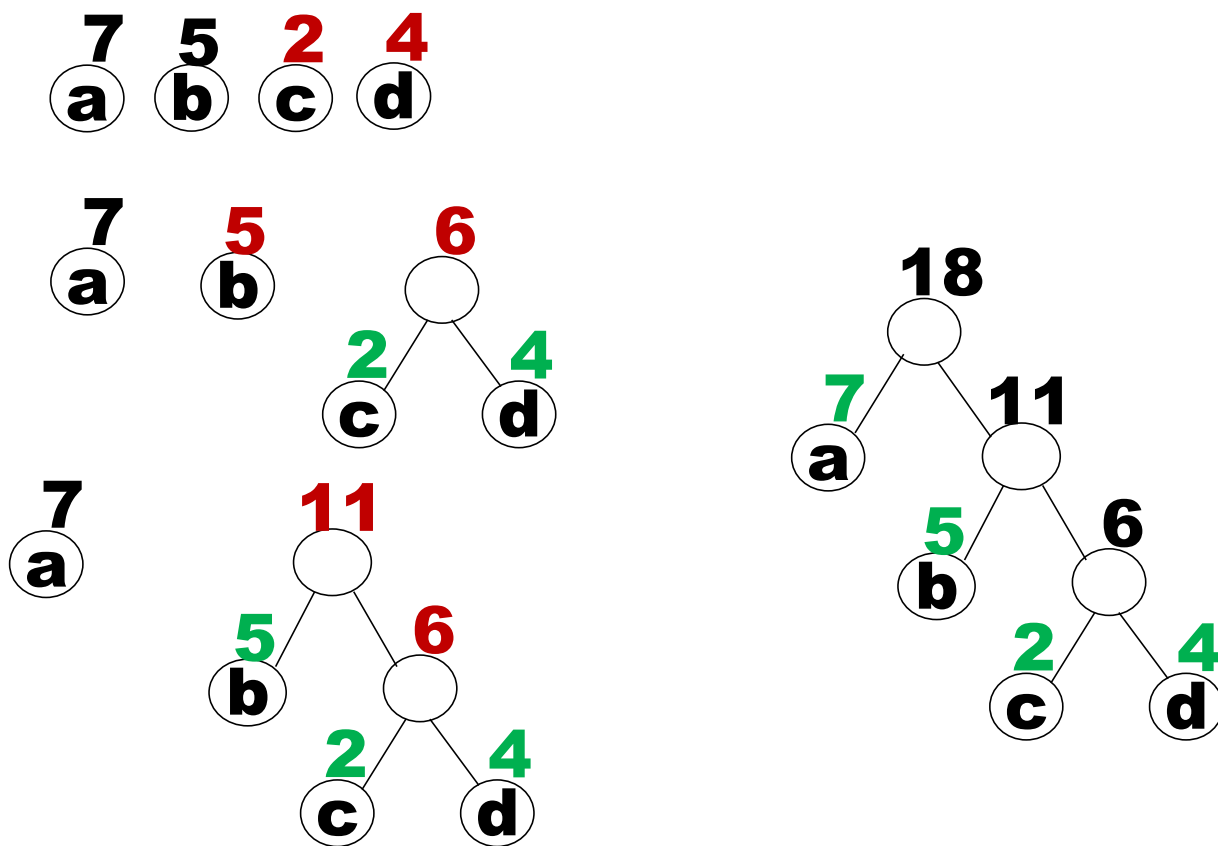
根据给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$ ，构造 n 棵只有根结点的二叉树，令其权值为 w_j

在森林中选取两棵根结点权值最小的树作左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和。

在森林中删除这两棵树，同时将新得到的二叉树加入森林中。重复上述两步，直到只含一棵树为止，这棵树即哈夫曼树。

5.5 最优二叉树 (赫夫曼树)

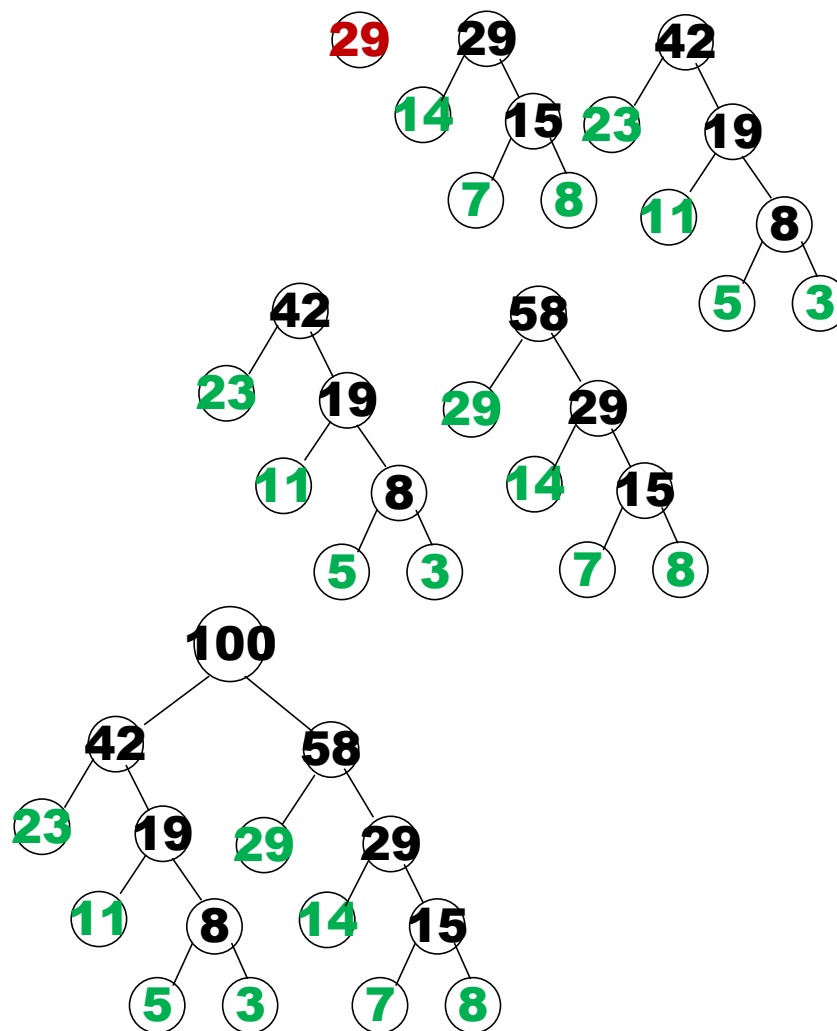
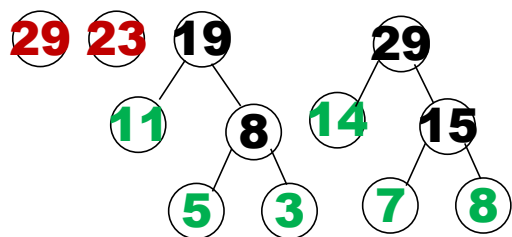
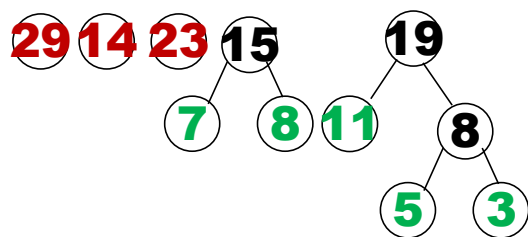
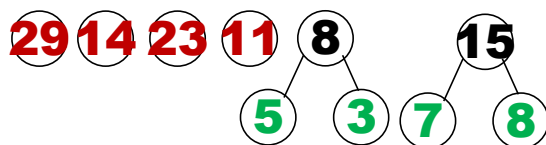
实例



5.5 最优二叉树 (赫夫曼树)

- 实例: $w=\{5,29,7,8,14,23,3,11\}$

5 29 7 8 14 23 3 11



5.5 最优二叉树（赫夫曼树）

Huffman编码（数据通信用的二进制编码）

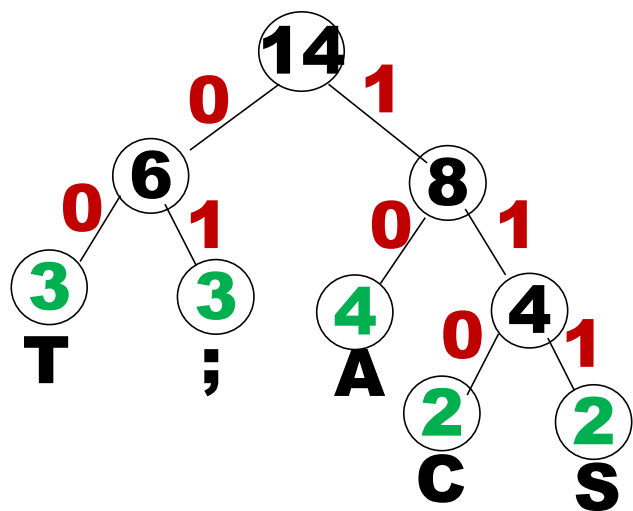
用赫夫曼树可以构造一种不等长的二进制编码，并且构造所得的赫夫曼编码是一种最优前缀编码，即使得所传电文的总长度最短。

思想：根据字符出现频率编码，使电文总长最短。

编码：根据字符出现频率构造Huffman树，然后将树中结点引向其左孩子的分支标为“0”，引向其右孩子的分支标为“1”；每个字符的编码即为从根到每个叶子的路径上得到的0、1序列。

5.5 最优二叉树 (赫夫曼树)

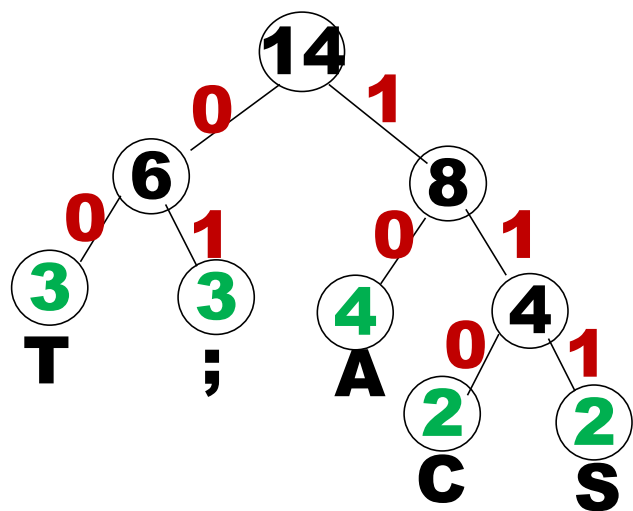
例 要传输的字符集 $D=\{C,A,S,T,;\}$
 字符出现频率 $w=\{2,4,2,3,3\}$



T : 00
; : 01
A : 10
C : 110
S : 111

5.5 最优二叉树 (赫夫曼树)

编码：从Huffman树根开始，从待译码电文中逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则编出一个字符；再重新从根出发，直到电文结束



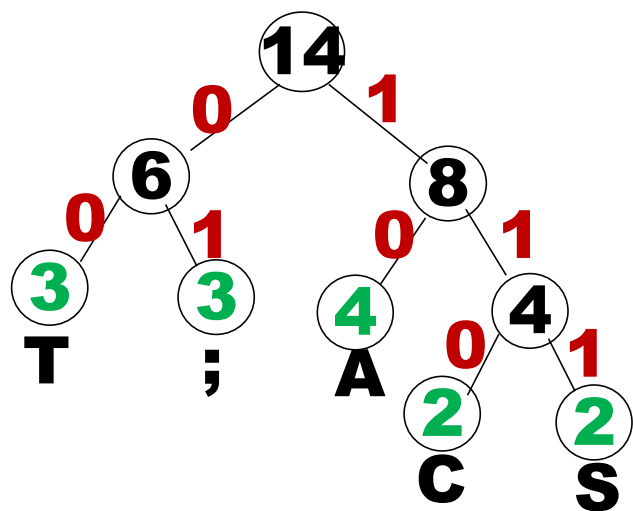
T : 00
; : 01
A : 10
C : 110
S : 111

例：电文是 { **CAS; CAT; SAT; AT** }

其编码：**110 10 111 01 1101000011111000011000**

5.5 最优二叉树 (赫夫曼树)

- 译码：从Huffman树根开始，从待译码电文中逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束



T : 00
; : 01
A : 10
C : 110
S : 111

例：电文编码：**11010111**

译文为：**C A S**

1. 熟练掌握二叉树的结构特性，了解相应的证明方法。
2. 熟悉二叉树的各种存储结构的特点及适用范围。
3. 遍历二叉树是二叉树各种操作的基础。实现二叉树遍历的具体算法与所采用的存储结构有关。掌握各种遍历策略的递归算法，灵活运用遍历算法实现二叉树的其它操作。层次遍历是按另一种搜索策略进行的遍历。
4. 理解二叉树线索化的实质是建立结点与其在相应序列中的前驱或后继之间的直接联系，熟练掌握二叉树的线索化过程以及在中序线索化树上找给定结点的前驱和后继的方法。二叉树的线索化过程是基于对二叉树进行遍历，而线索二叉树上的线索又为相应的遍历提供了方便。

5. 熟悉树的各种存储结构及其特点，掌握树和森林与二叉树的转换方法。
6. 学会编写实现树的各种操作的算法。
7. 了解最优树的特性，掌握建立最优树和哈夫曼编码的方法。

1、找出所有满足下列条件的二叉树：

- a. 它们在先序遍历和中序遍历时，得到的结点访问序列相同；
- b. 它们在后序遍历和中序遍历时，得到的结点访问序列相同；
- c. 它们在先序遍历和后序遍历时，得到的结点访问序列相同；

2. 假设一棵二叉树的先序序列为EBADCFHGIKJ和中序序列为ABCDEFGHIJK。请画出该树。

3. 编写递归算法，将二叉树中所有结点的左、右子树相互交换。

4. 假设用于通讯的电文由8个字母组成，字母出现的频率为0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10。试为这8个字母设计哈夫曼编码,并举例说明。

4. 请对下图所示二叉树进行后序线索化，为每个空指针建立相应的前驱或后继线索

