数据结构与算法设计

2021-09

课程内容简介

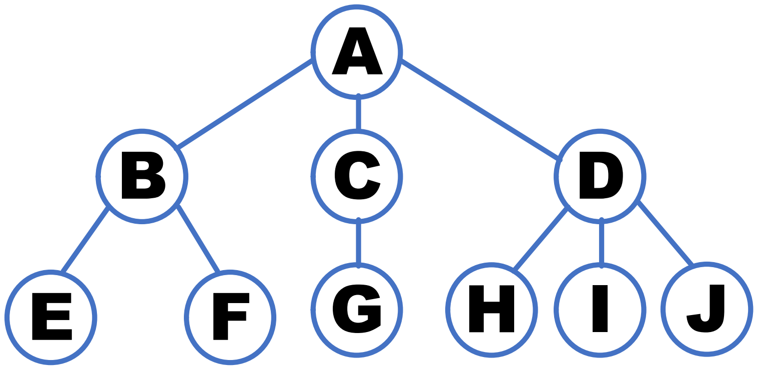
第1章 绪论	第8章 排序与分治	串与串匹配算法
第2章 线性表	第9章 外部排序	红黑树
第3章 栈和队列	第10章 动态规划算法	k-d树
第4章 数组和广义表	第11章 有限自动机	复杂图算法
第5章 树、二叉树、回溯法	第12章 图灵机	文本检索技术
第6章 图与贪心算法	第13章可判定性	分支限界算法
第7章 查找	第14章 时间复杂性	随机化算法
		上下文无关文法

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。

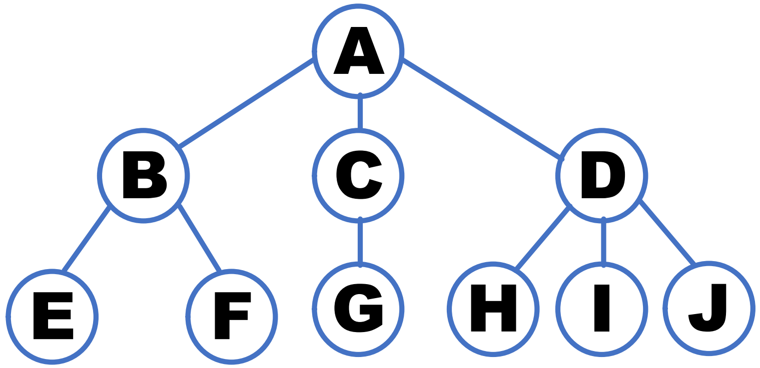
  否则：

  (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，…，T_m，其中每一棵子集本身又是一棵符合本定义的树，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：

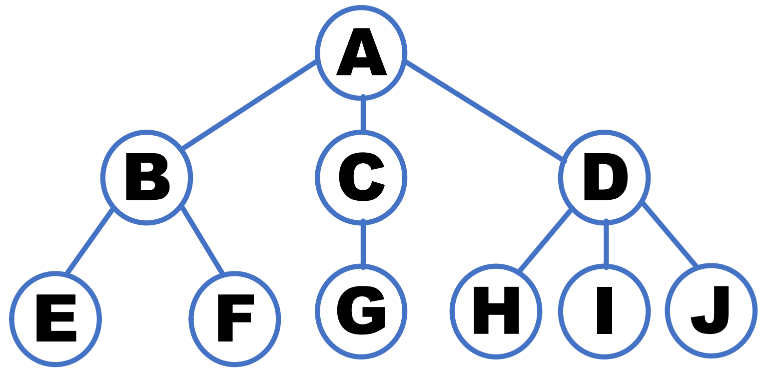
T1={ B,E,F } T2={ C,G } T3={ D,H,I,J }

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**T1**，**B**是根，其余结点可以划分为两个互不相交的集合：

T11={ E } T12={ F } T11，**T12**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；

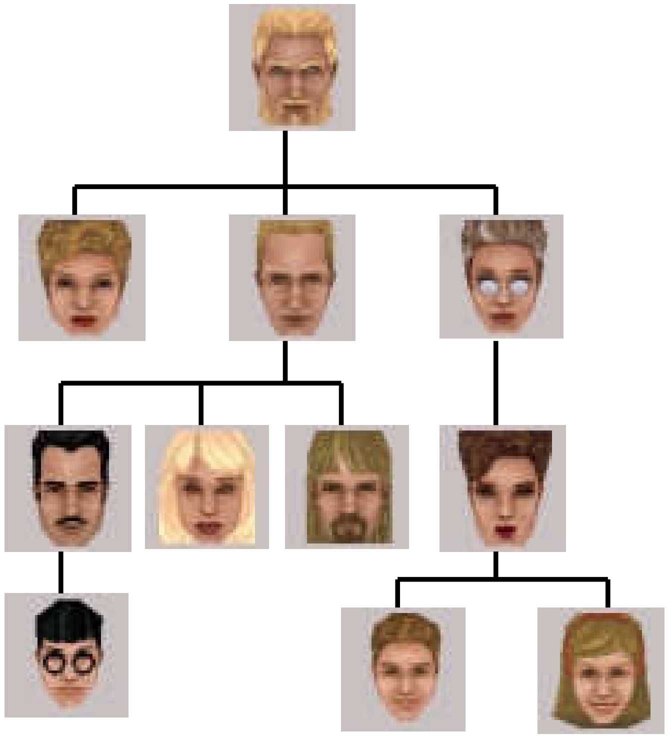
　　3）树的结点，可以有零个或多个后继；

　　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。

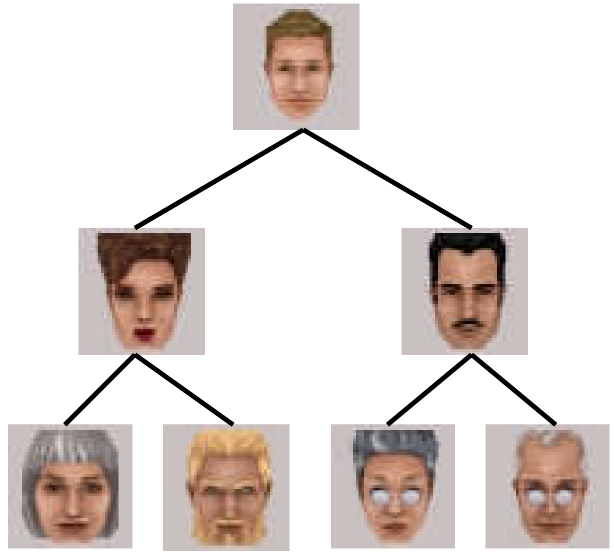
　　5）除根之外的其它结点，都存在惟一一条从根到该结点的路径；

5.1 树的定义与基本术语

树的应用



家族树



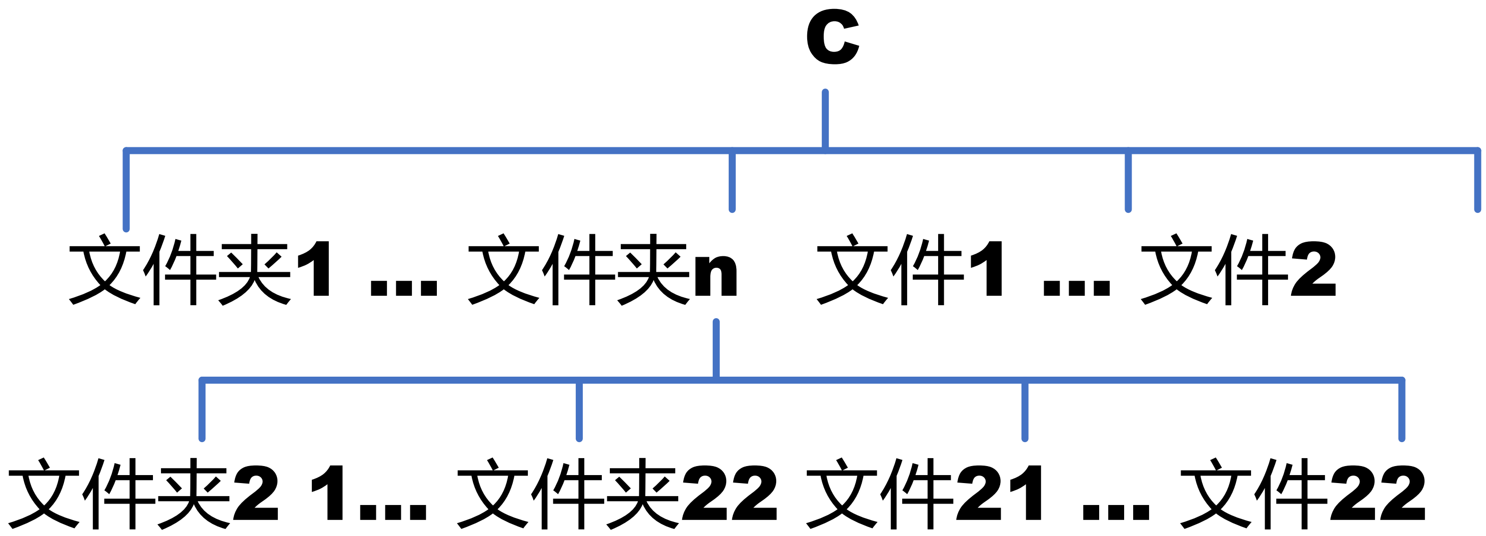
血统树
（二叉树）

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。

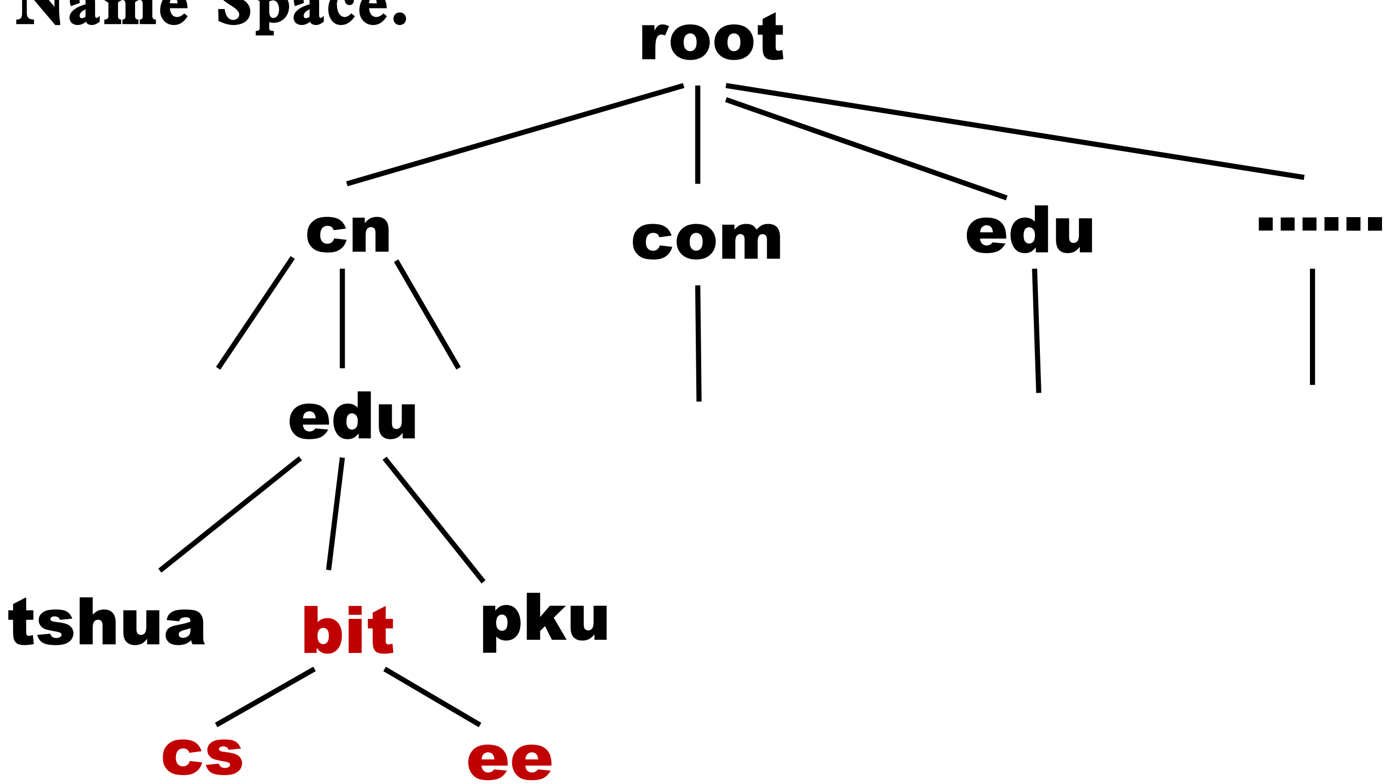
不论是DOS文件系统还是window文件系统，所有的文件都是用树的形式进行组织。



5.1 树的定义与基本术语

树的应用

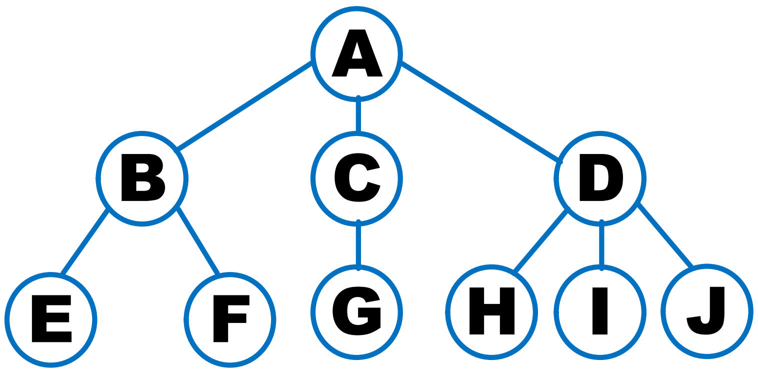
DNS Name Space.



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）



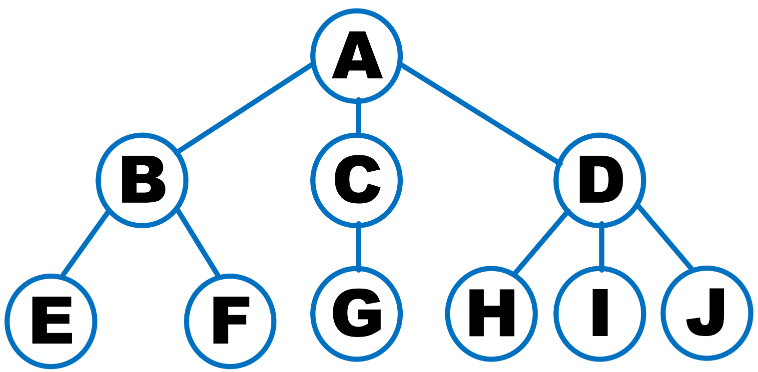
5.1 树的定义与基本术语

树的表示

2）二元组表示

T ={ A， B， C， D， E， F， G， H， I， J }

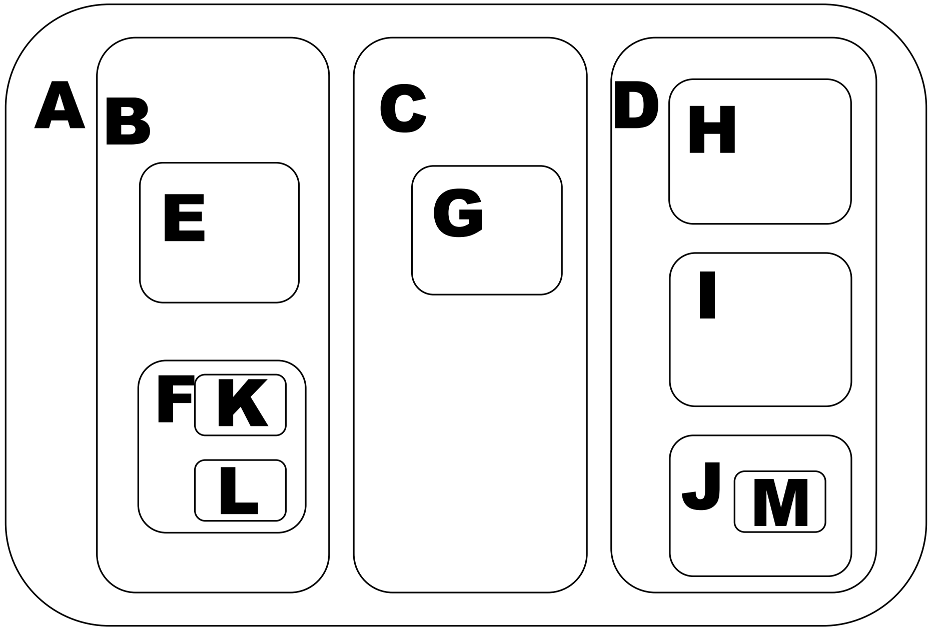
R = { <A,B>, <A,C>, <A,D>, <B,E>, <B,F>, <C,G>, <D,H>, <D,I>, <D,J> }

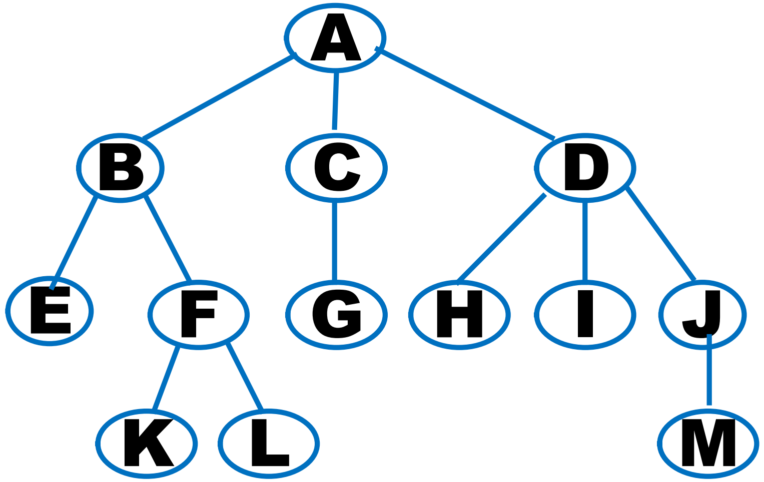


5.1 树的定义与基本术语

树的表示

3）文氏图表示



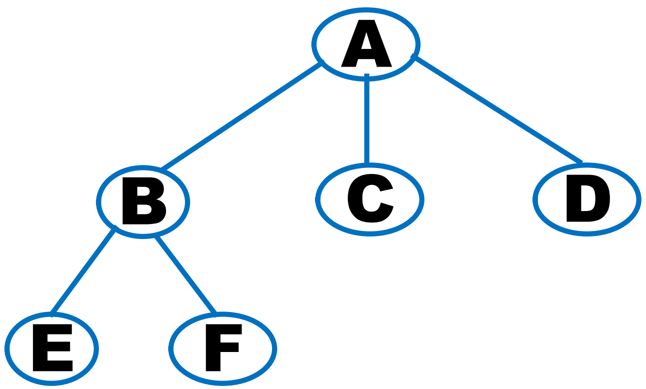


5.1 树的定义与基本术语

树的表示

4）广义表表示

假设树的根为root，子树为$T_1, T_2, \cdots, T_n$，与该树对应的广义表为L，则：L=(原子(子表1,子表2, …, 子表n))，其中原子对应root，子表 i(1<i<=n) 对应 Ti。



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

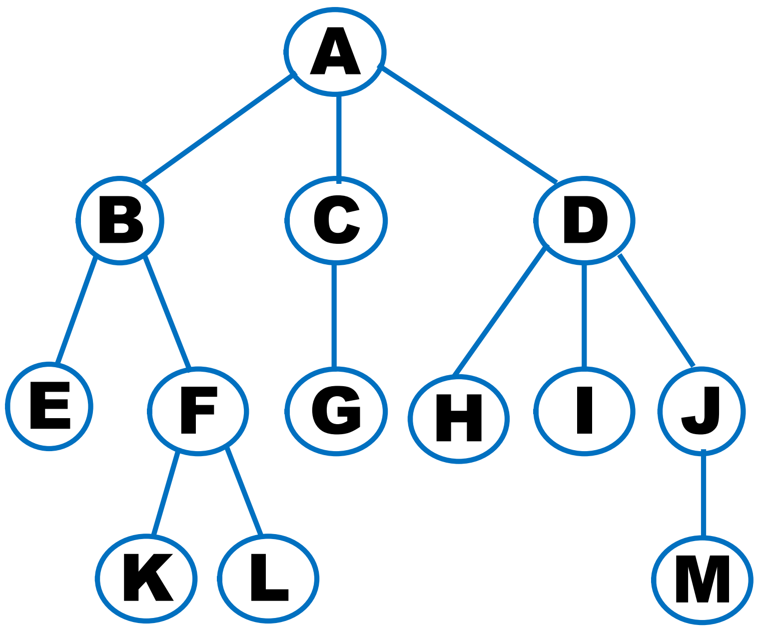
树根 T_1 T_2 T_3

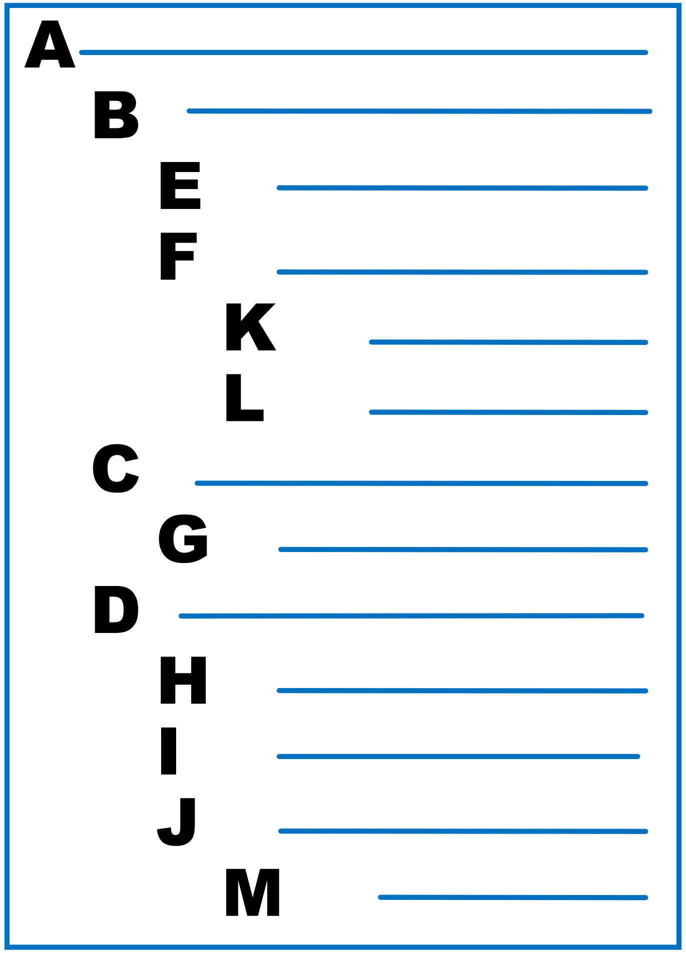
(A,(B,((E),(F)),(C),(D)))

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

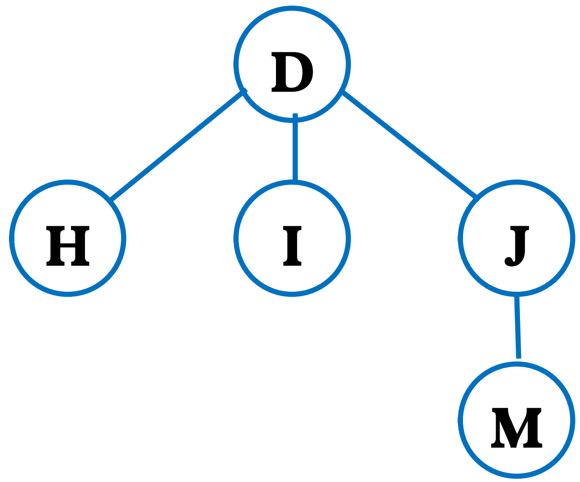
结点： 数据元素+若干指向子树的分支

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

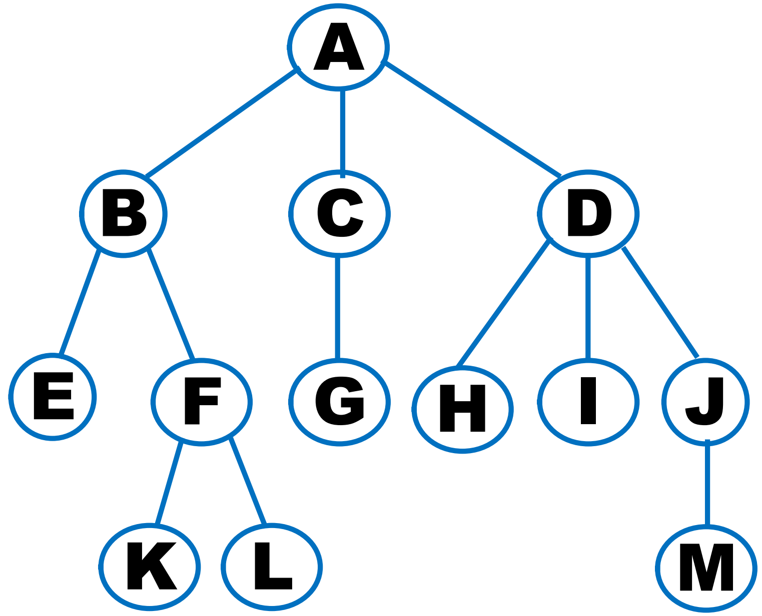
分支结点： 度大于零的结点



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

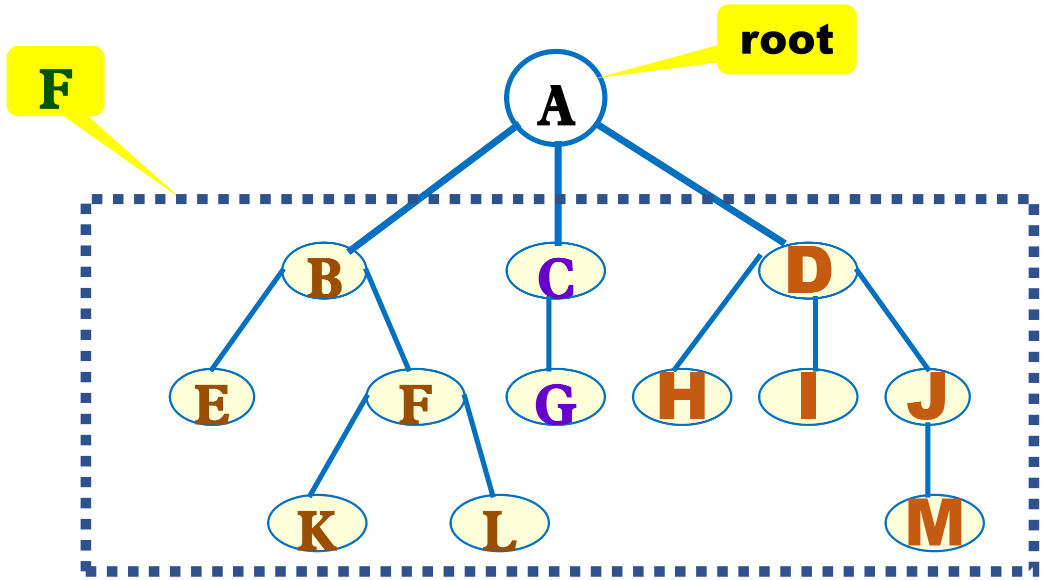
结点的层次：假设根结点的层次为1,第*l*层的结点的子树根结点的层次为*l*+1

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≥0）棵互不相交的树的集合。



任何一棵非空树是一个二元组

Tree =（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	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:

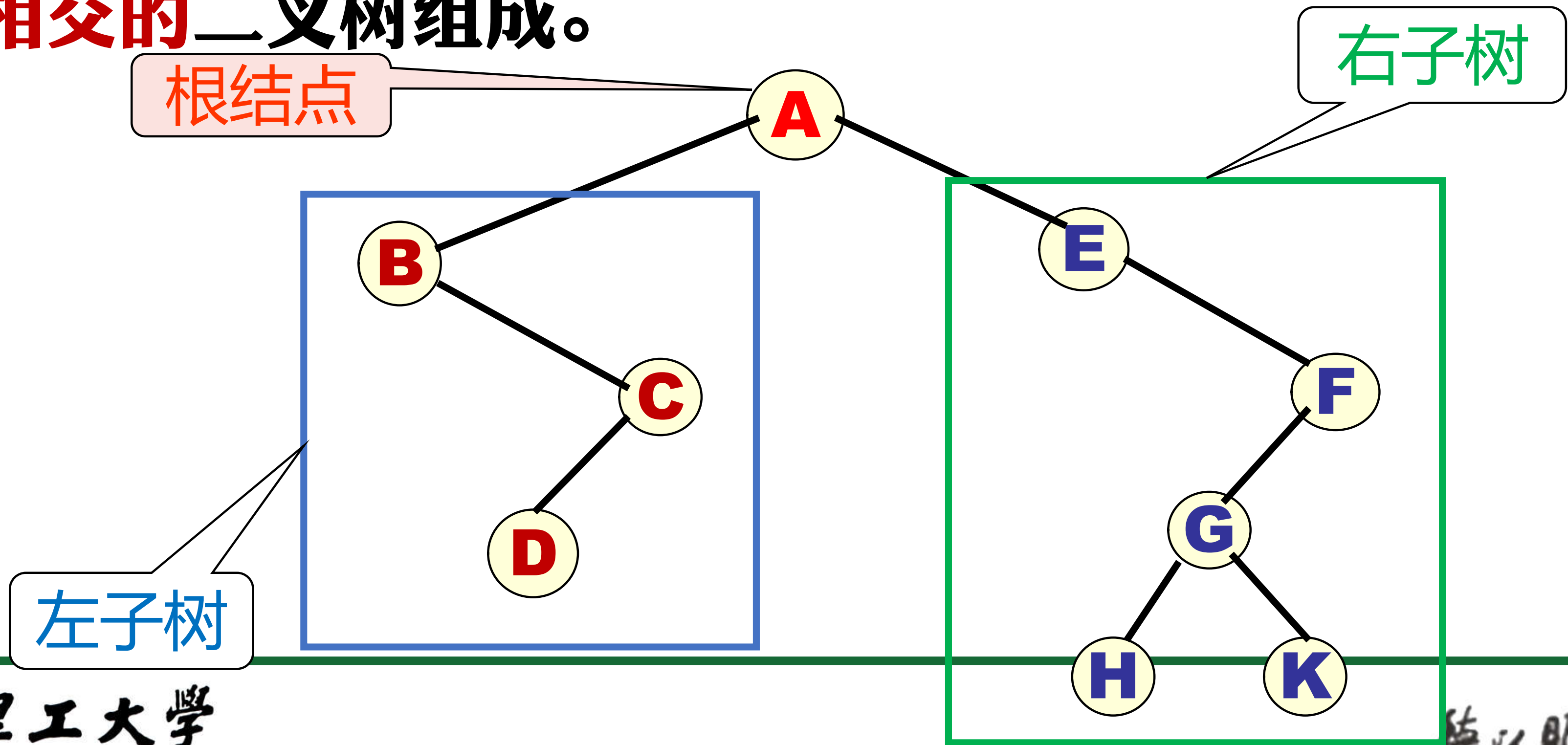
(1) 在D中存在惟一的称为根的数据元素root，它在关系H下无前驱;

(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

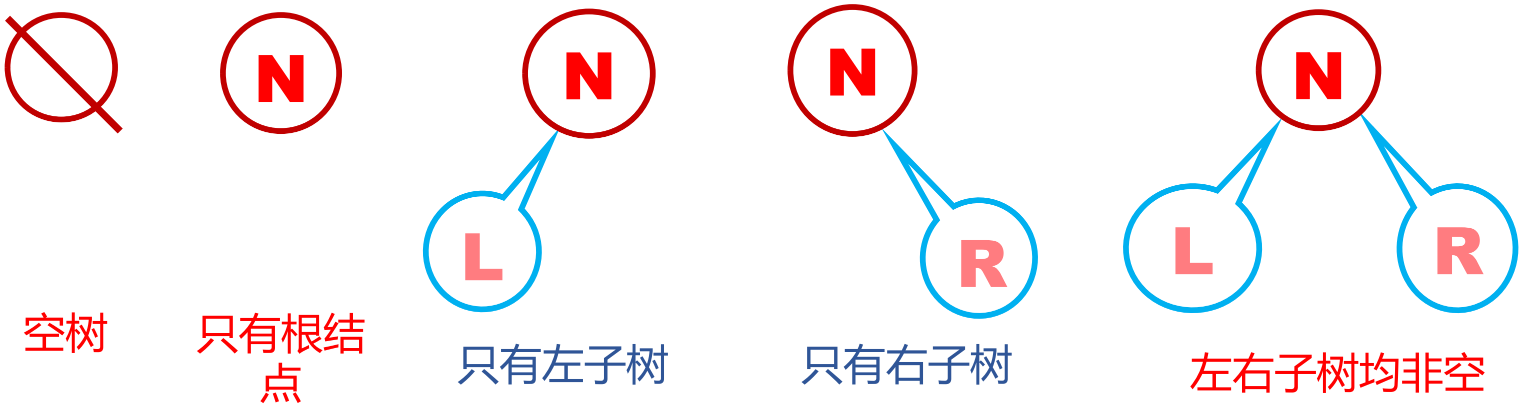
定义

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



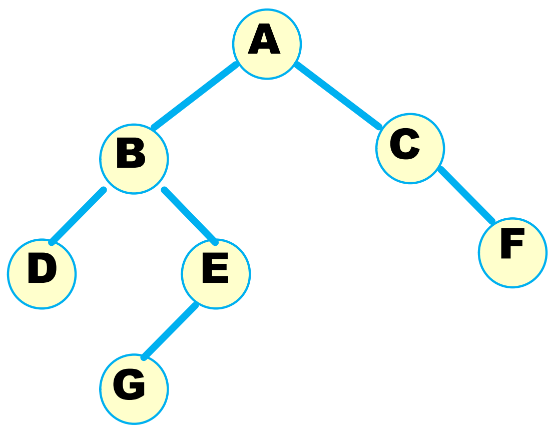
说明:

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**；

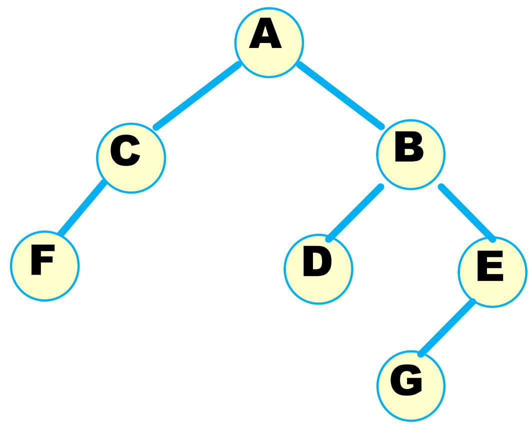
2）左、右子树不能颠倒——**有序树**。

5.2 二叉树

二叉树



(a)



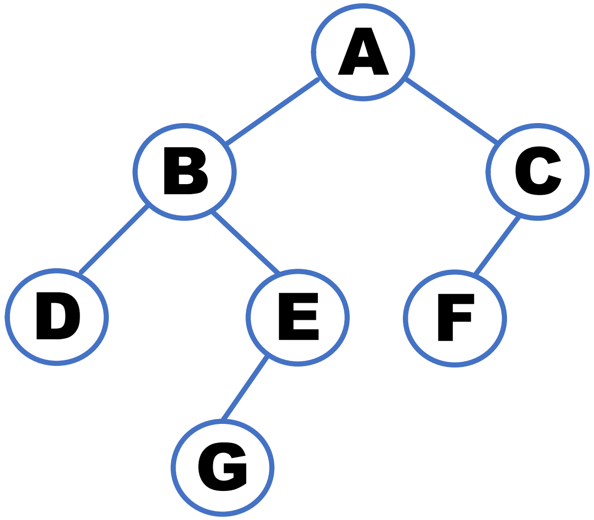
(b)

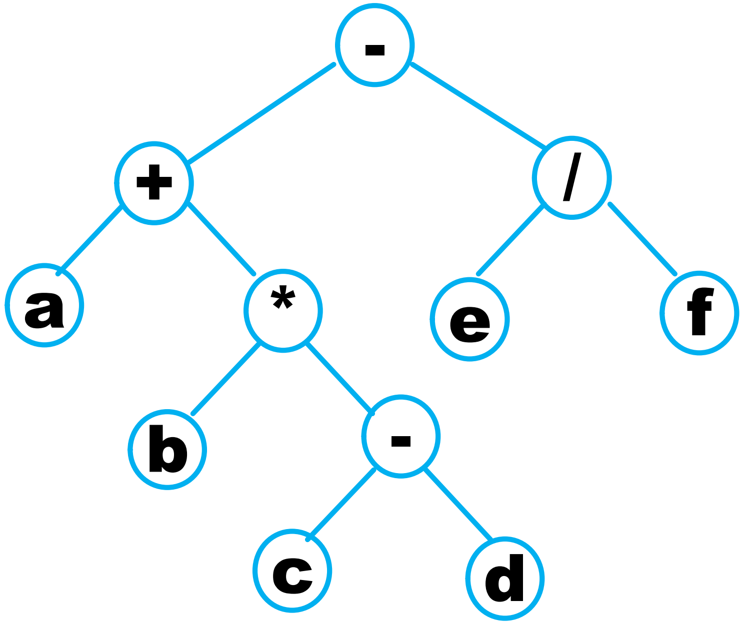
(a)、(b)是不同的二叉树

(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



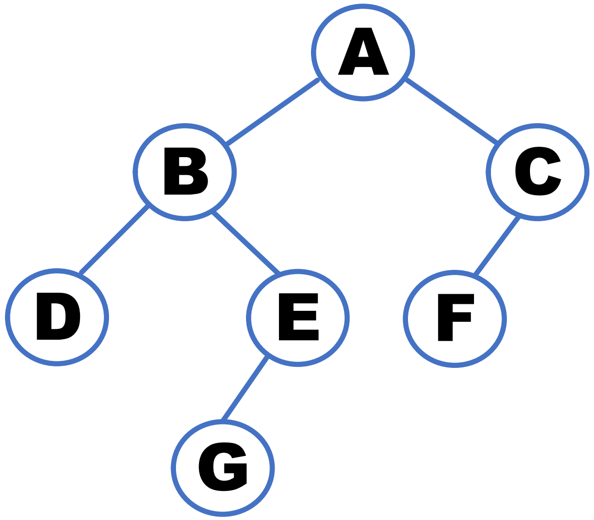


a + b * (c – d) – e / f

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≥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≥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$=2^{i-2}*2=2^{i-1}$。

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 ≥ 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：

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

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：$b = n_1 + 2n_2$

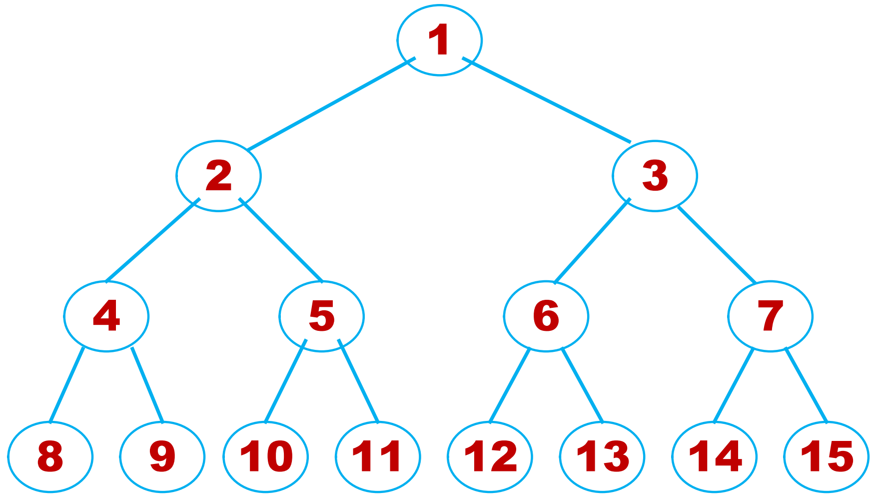
而 $b = n-1 = n_0 + n_1 + n_2 - 1$

由此， $n_0 = n_2 + 1$

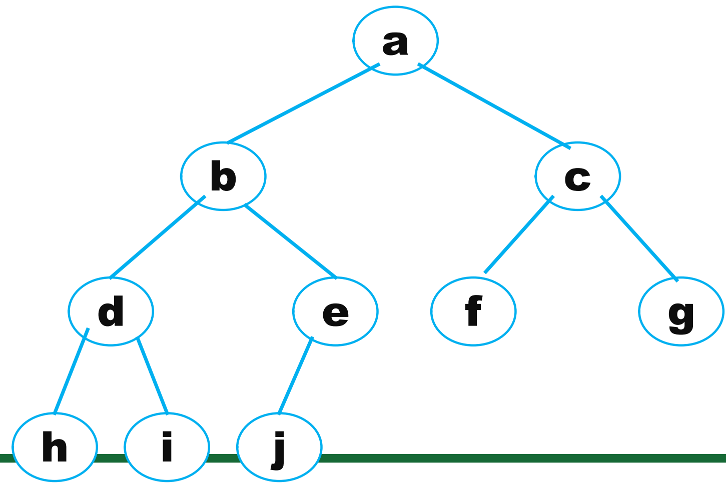
5.2 二叉树

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_2n +1。

证明：

设 完全二叉树的深度为 k

则根据第二条性质得 $2^{k-1} \leqslant n < 2^k$

即 $k-1 \leqslant log_2 n < k$

因为 k 只能是整数，因此，$k = log_2n + 1$

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至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点：

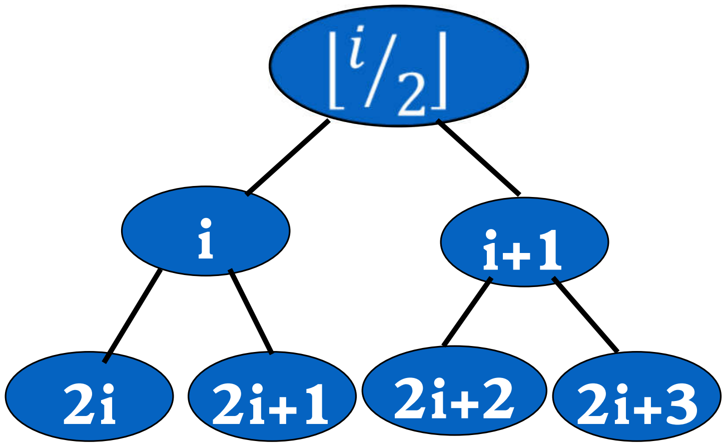
  (1) 若 $i=1$，则该结点是二叉树的根，无双亲； 否则$i>1$，编号为 $\lfloor i/2 \rfloor$的结点为其双亲结点；

  (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；

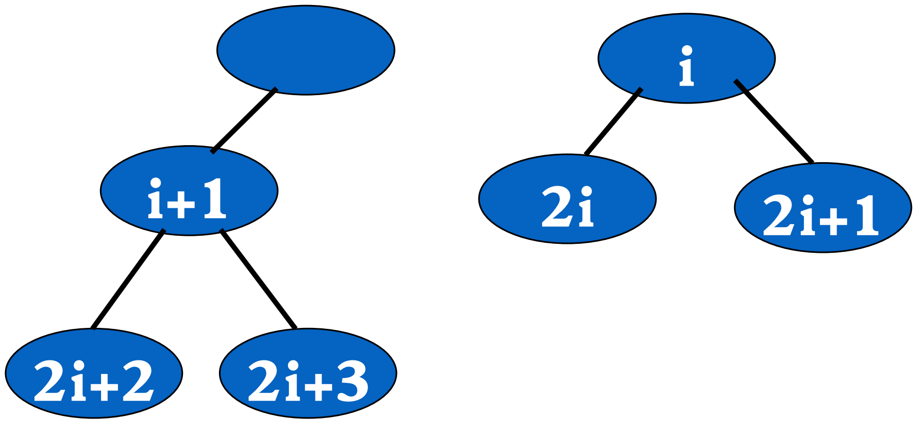
  (3) 若 $2i+1>n$，则该结点无右孩子结点；否则，编号为$2i+1$ 的结点为其右孩子结点。

5.2 二叉树

性质5：



(a) i 和 i+1 结点在同一层



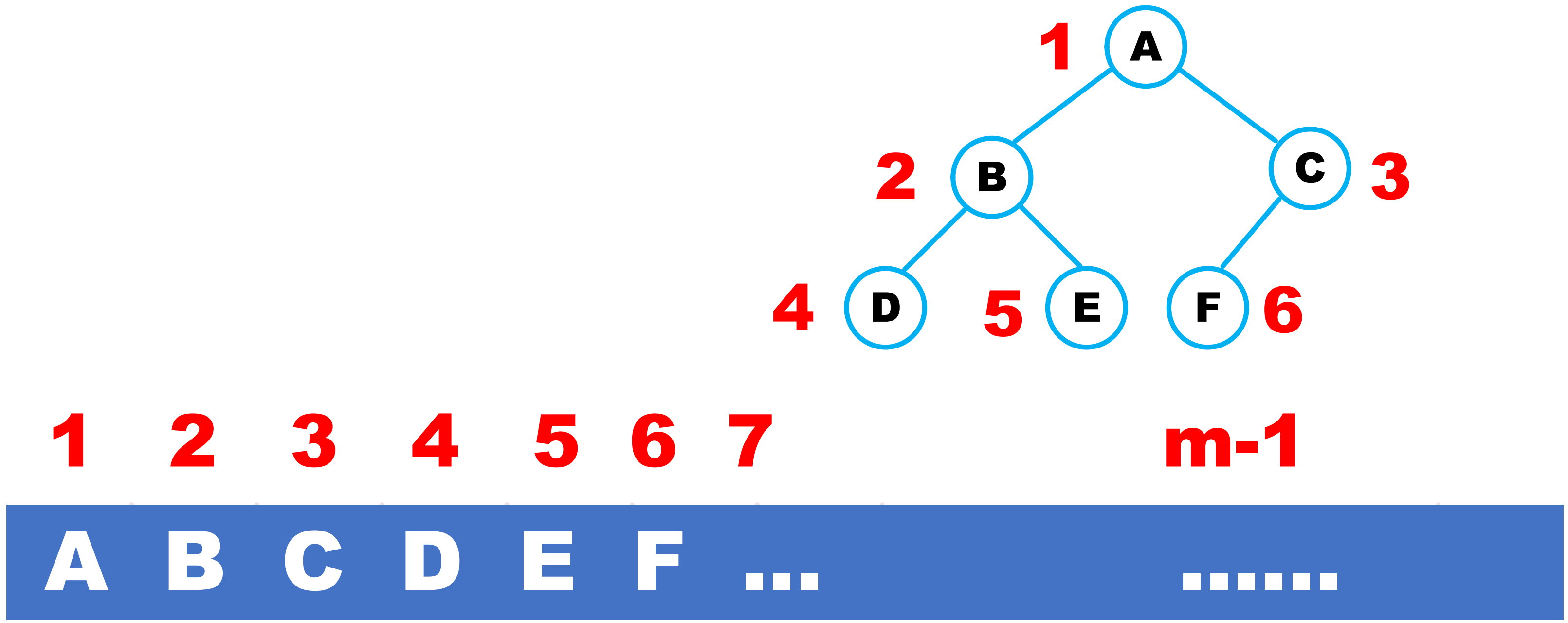
(b) i 和 i+1 结点不在同一层

5.2 二叉树

二叉树的存储结构

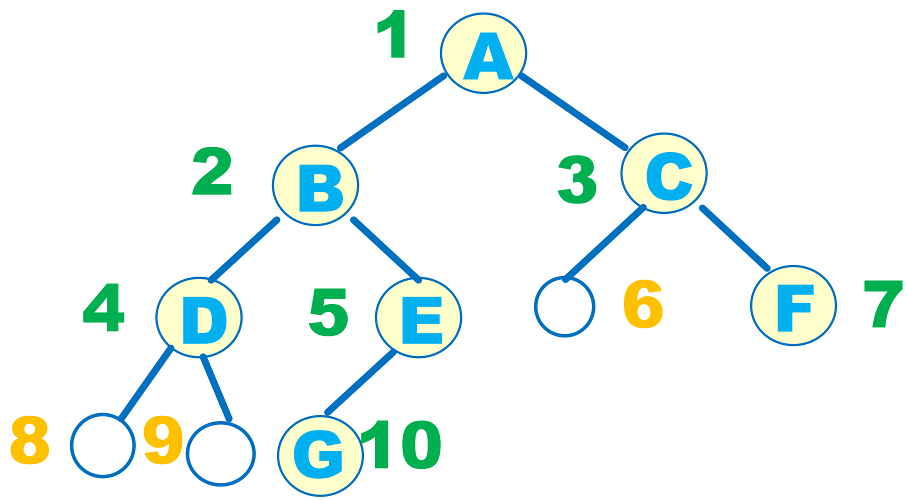
1. 二叉树的顺序结构

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。



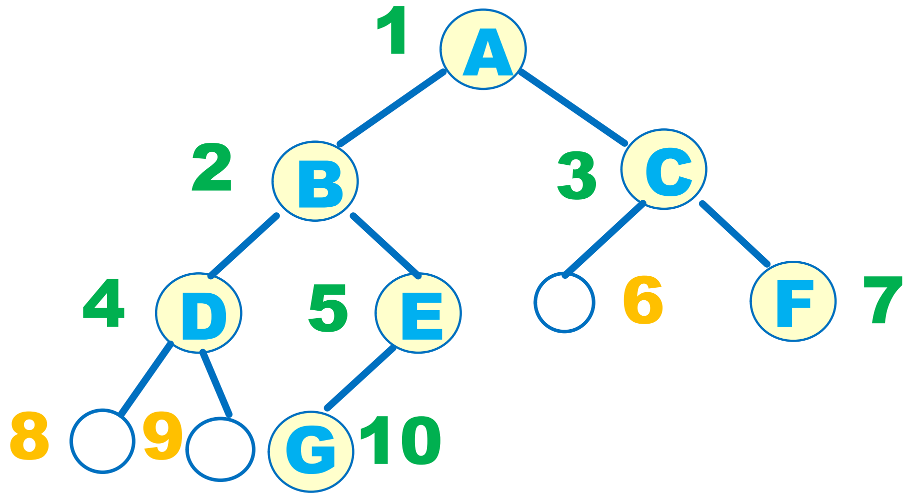
5.2 二叉树

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

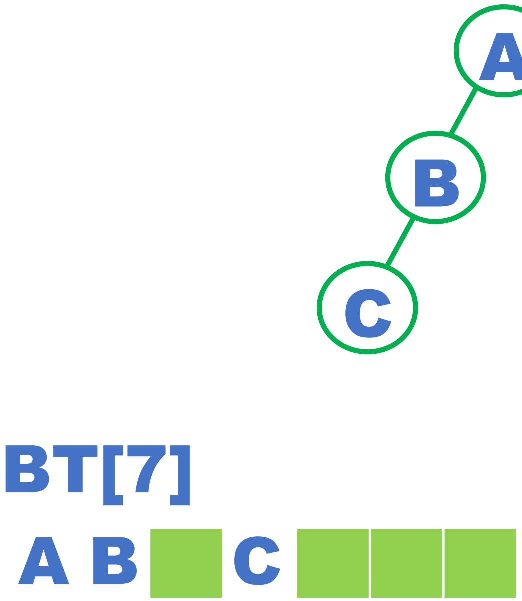
将二叉树原有的结点按编号存储到内存单元“相应”的位置上。

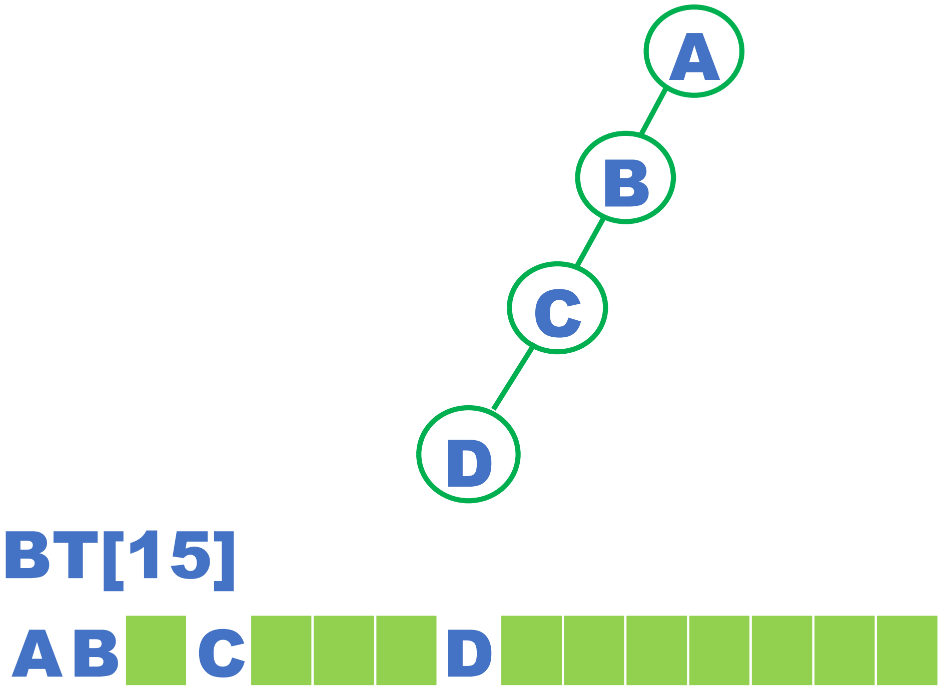


1	2	3	4	5	6	7	8	9	10	m-1
A	B	C	D	E	0	F	0	0	G	

5.2 二叉树

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。



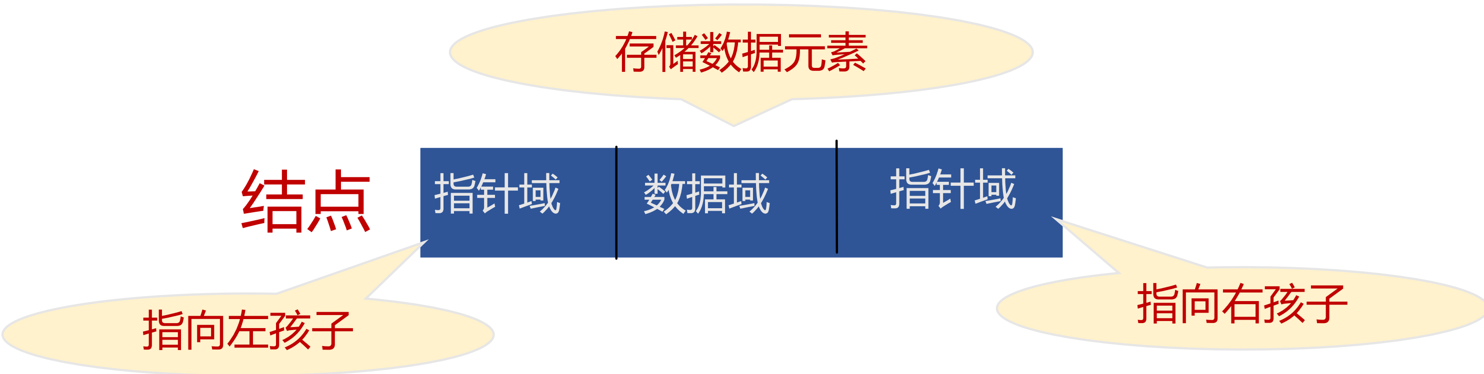


5.2 二叉树

2. 二叉链表

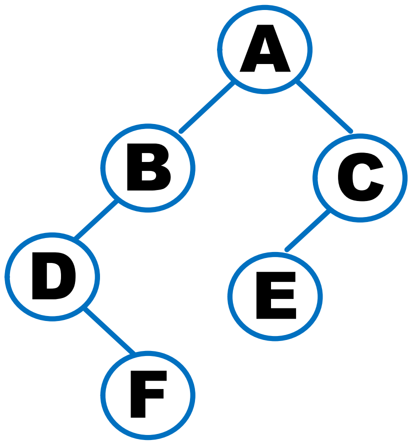
二叉链表中每个结点包含三个域：数据域、左指针域、右指针域。

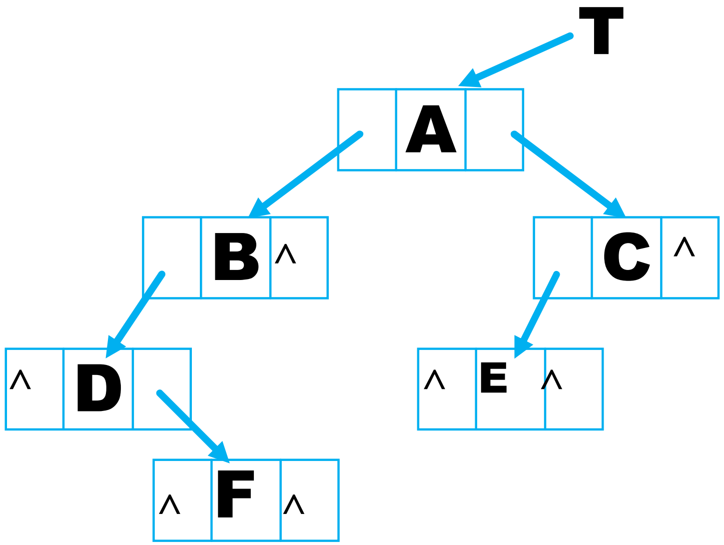
```
typedef struct BiTNode
{   ElemType data;
    struct BiTNode  * lchild, * rchild;
} BiTNode, * BiTree;
```



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。

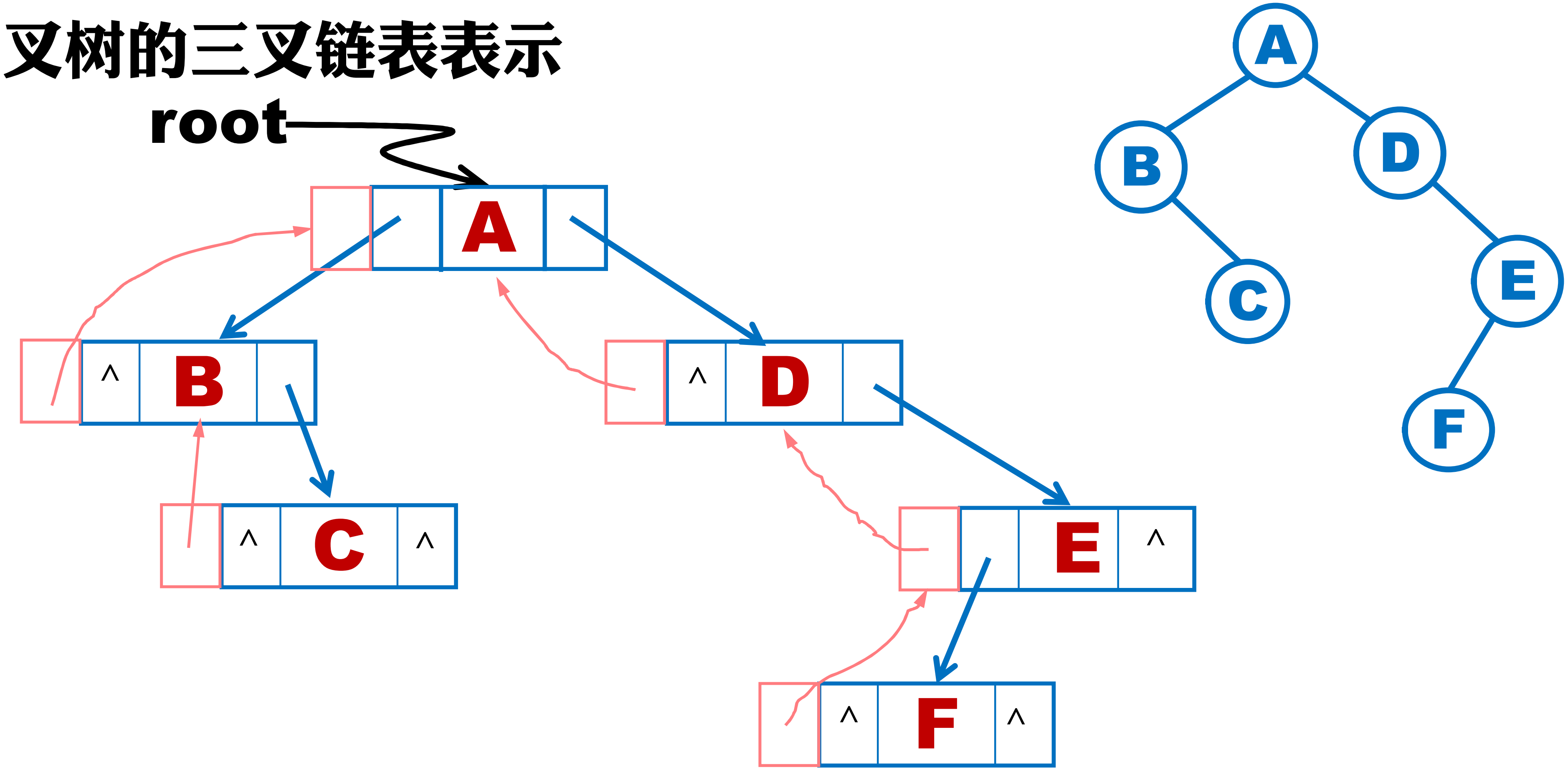
```
typedef struct BiTNode
{   ElemType data;
    struct BiTNode  * lchild, * rchild;
    struct BiTNode  * parent;
} BiTNode,*BiTree;
```

结点结构：

parent	lchild	data	rchild

5.2 二叉树

二叉树的三叉链表表示



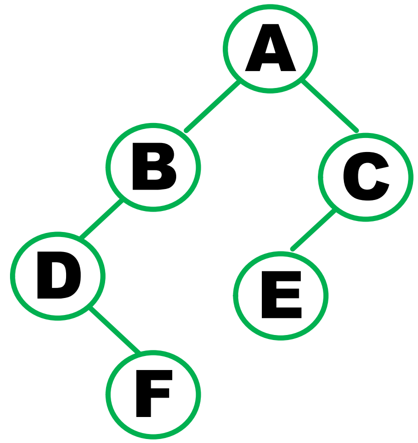
5.2 二叉树

4. 静态二叉链表

采用数组存贮。

root = 0

	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			



孩子结点在数组中的位置。用**-1**表示无左孩子或右孩子

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

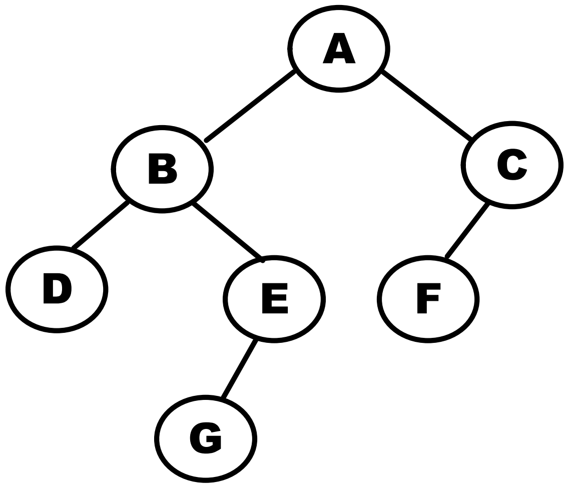
5. 双亲链表

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
```

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。

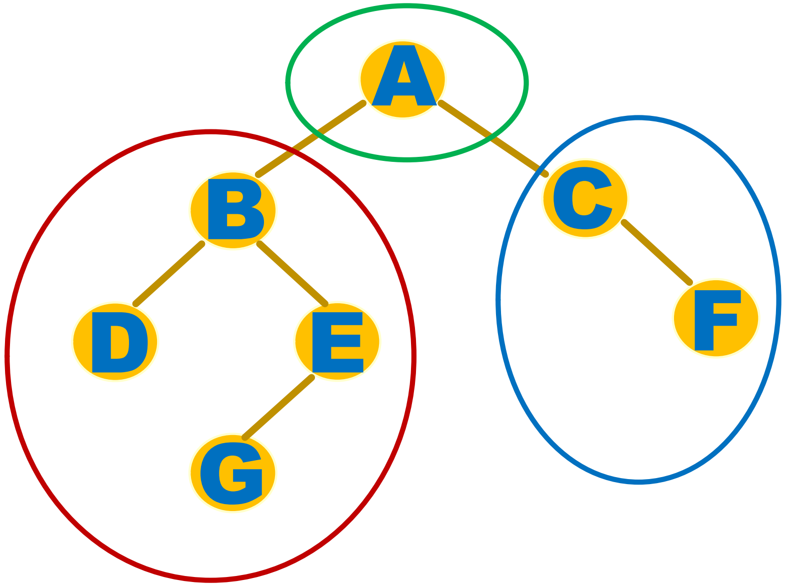
二叉树由根、左子树、右子树三部分组成。

二叉树的遍历可以分解为：访问根，遍历左子树和遍历右子树。

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：

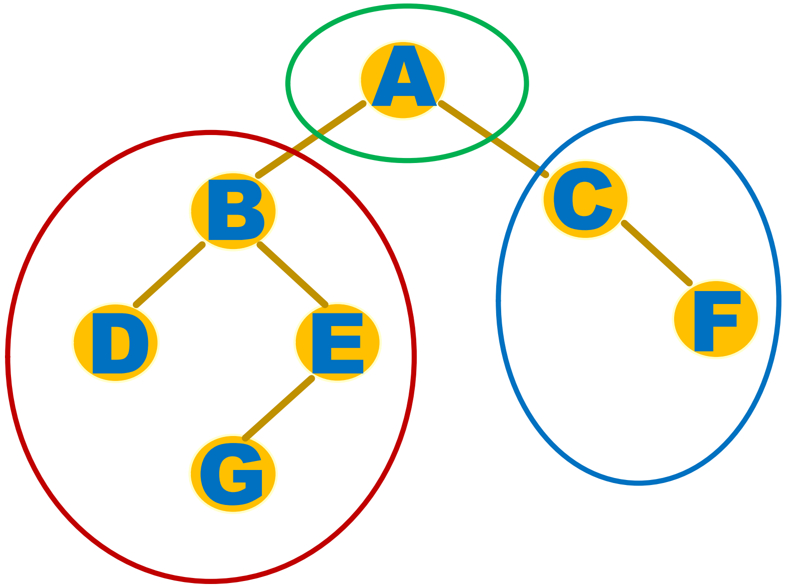
基本：DLR，LDR，LRD

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（ DLR ）



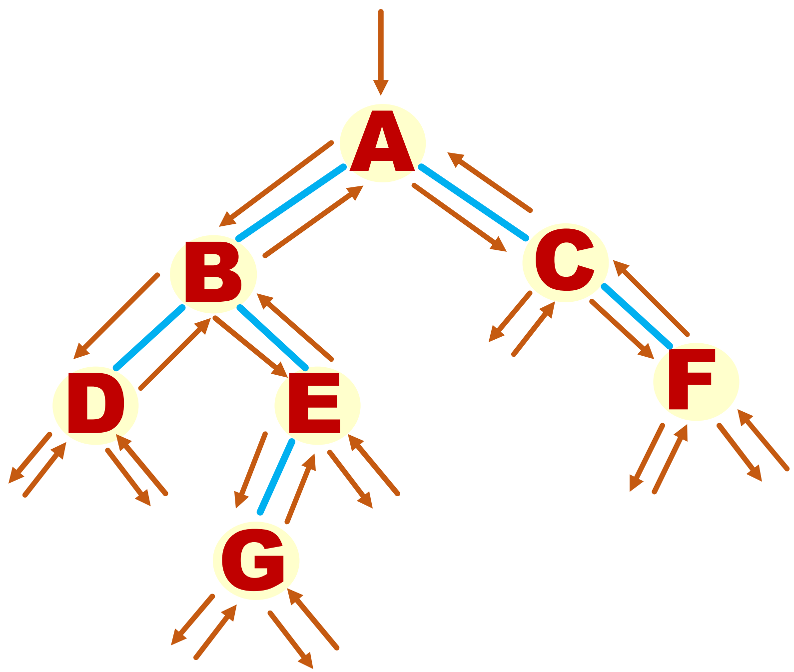
先序遍历（ DLR ）
若二叉树非空
（1）访问根结点；
（2）先序遍历左子树；
（3）先序遍历右子树。

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 DLR 的顺序遍历左子树
3）先序遍历右子树：即按 DLR 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 ）

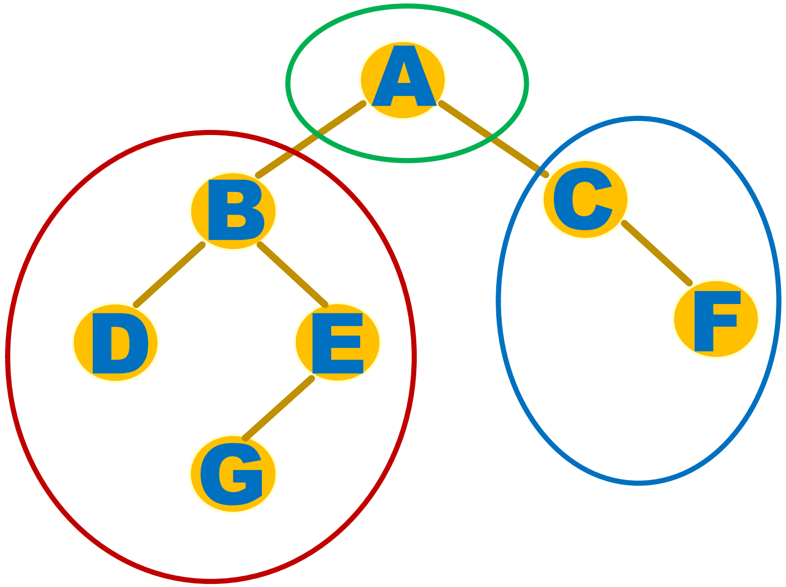
先序遍历（ DLR ）
若二叉树非空
（1）访问根结点;
（2）先序遍历左子树;
（3）先序遍历右子树。



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

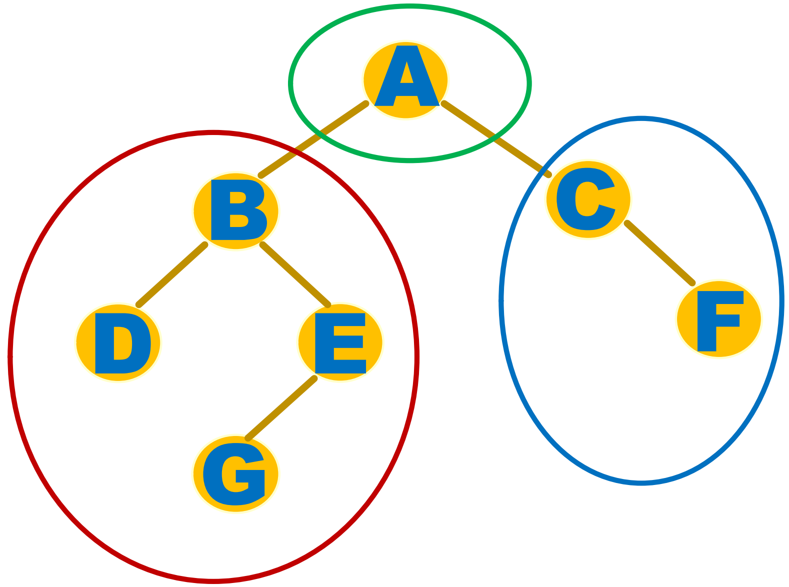
2）访问根结点**A**

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

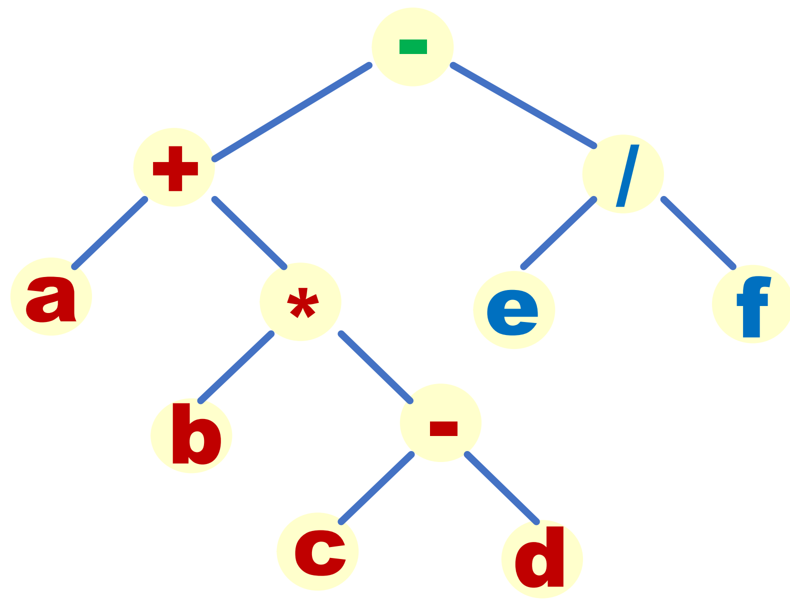
5.3 遍历二叉树与线索二叉树

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）

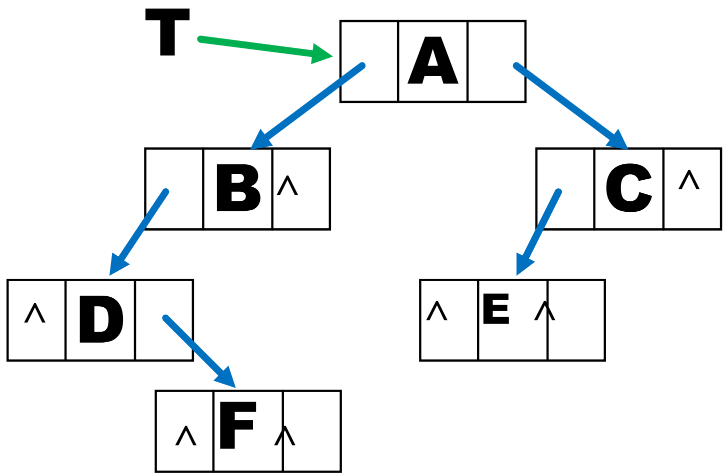
若二叉树非空——递归项

（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( 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
      // 本算法中序遍历以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
       // 本算法后序遍历以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2：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：

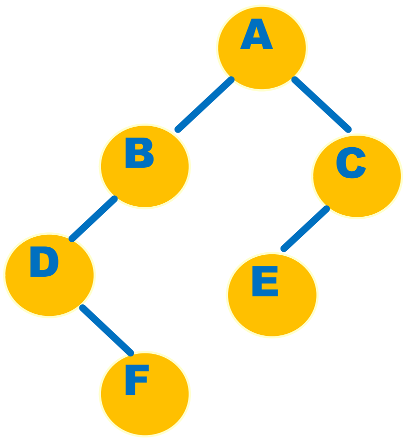
输入（在空子树处添加字符 * 的二叉树的）先序序列（设每个元素是一个字符）。

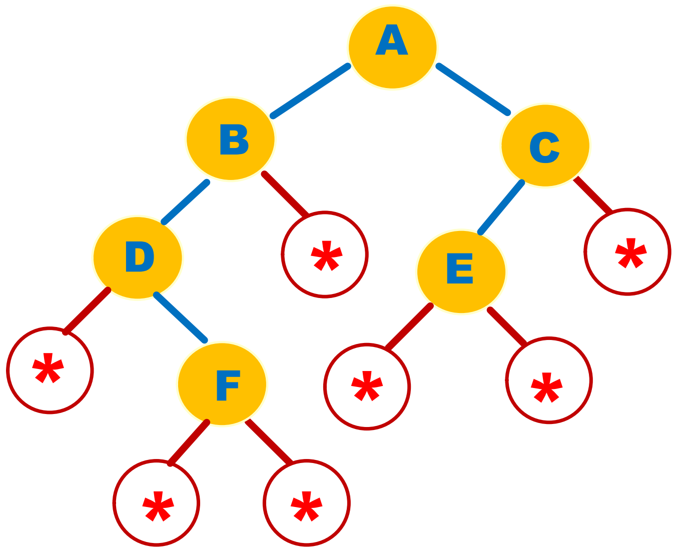
按先序遍历的顺序，建立二叉链表的所有结点并完成相应结点的链接。

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2：建立二叉链表。





先序序列：**A B D F C E**

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

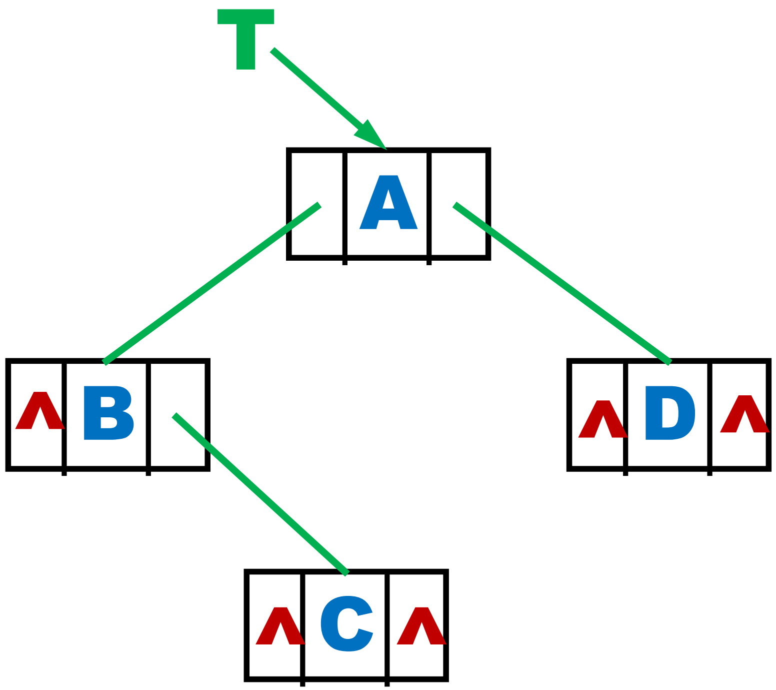
5.3 遍历二叉树与线索二叉树

```
Status CreateBiTree ( BiTree &T )
{  // 按先序遍历的顺序建立二叉链表
    scanf ( &ch );
    if ( ch ==  '*'  )    T=NULL;   // 若ch== '*' 则表示空子树
    else  {
        if ( ! (T=( BiTNode * ) malloc( sizeof( BiTNode ))) )
              exit( OVERFLOW );
        T->date = 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2：建立二叉链表。

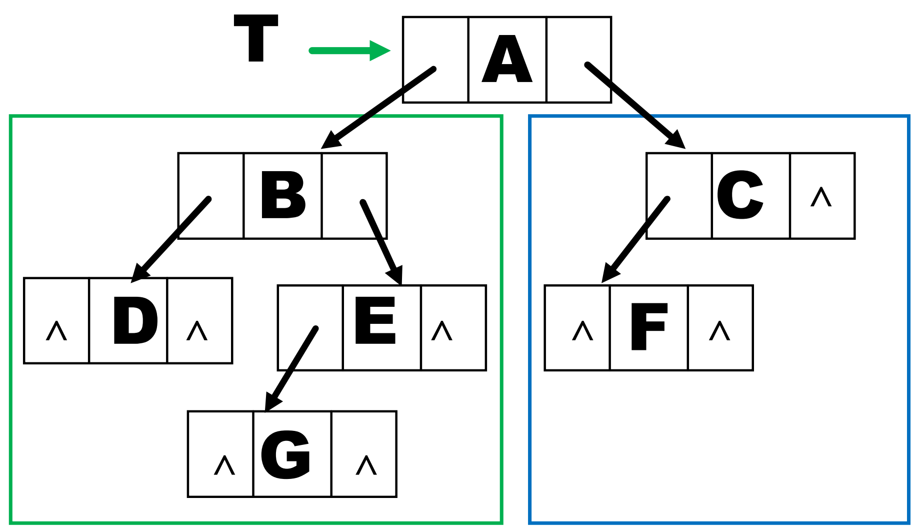
A B * C * * D * *



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

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

遍历的非递归算法

栈是实现递归的最常用的结构。

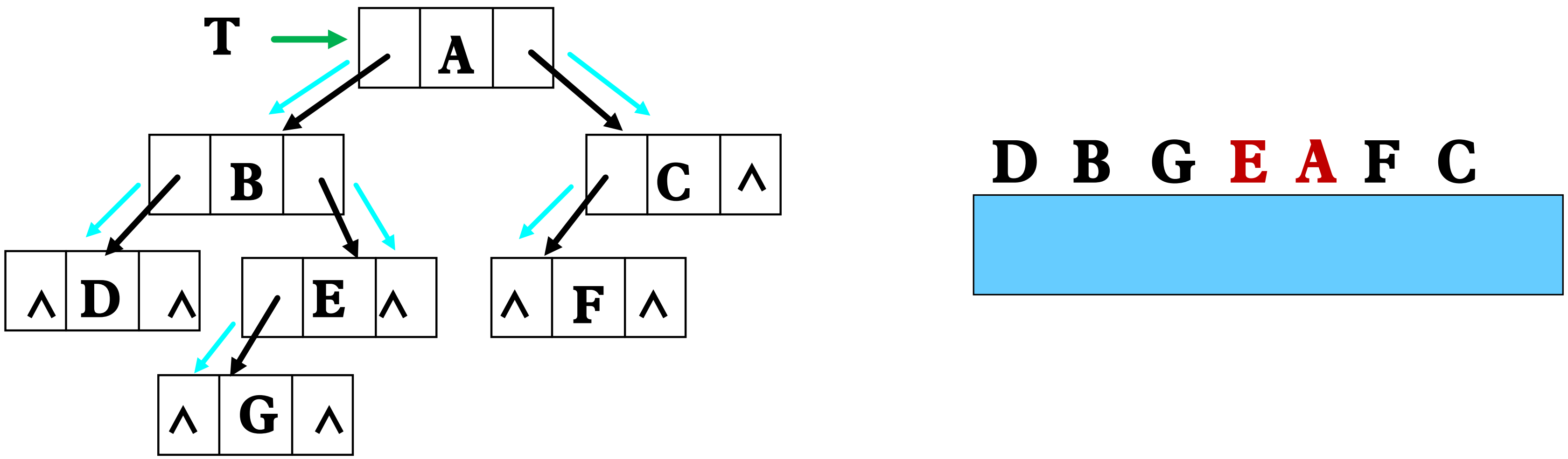
利用一个栈来记下尚待遍历的结点或子树，以备以后访问。

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。**



5.3 遍历二叉树与线索二叉树

```
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，直到找到最左下结点
       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
```

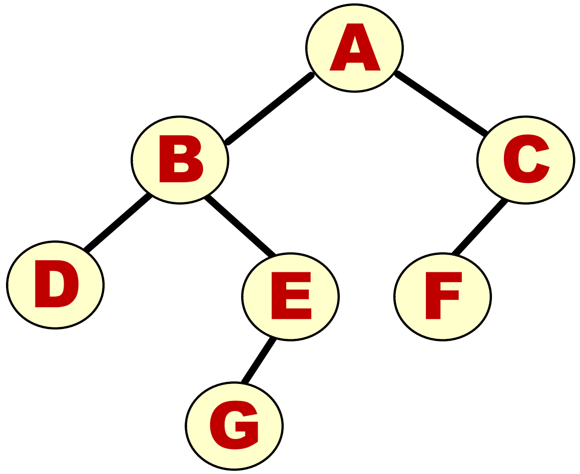
5.3 遍历二叉树与线索二叉树

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**

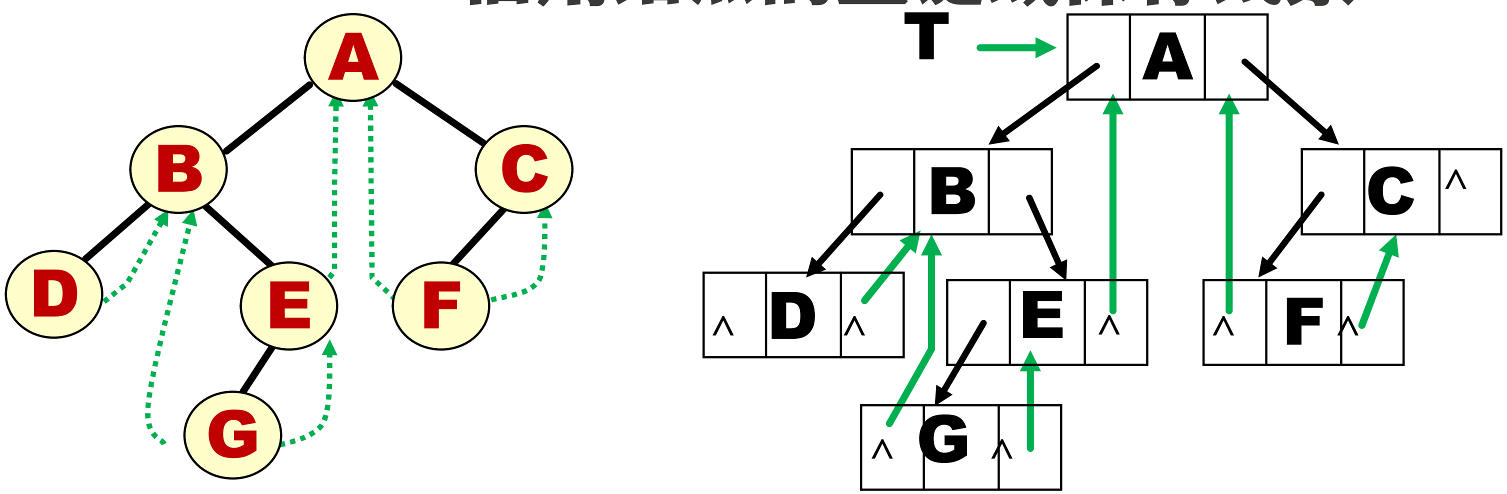
后序序列： **D G E B F C A**

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)

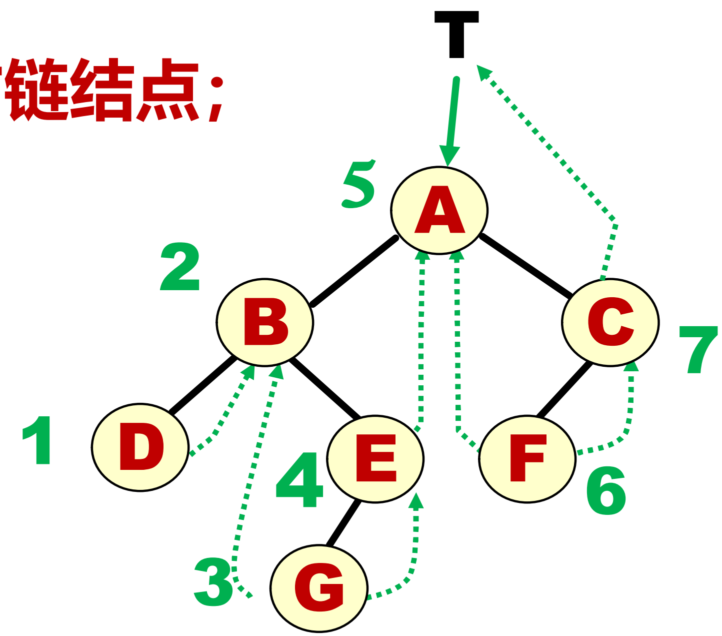
- 中序遍历的第一个结点

  二叉树的最左下结点

- 当前结点的后继结点

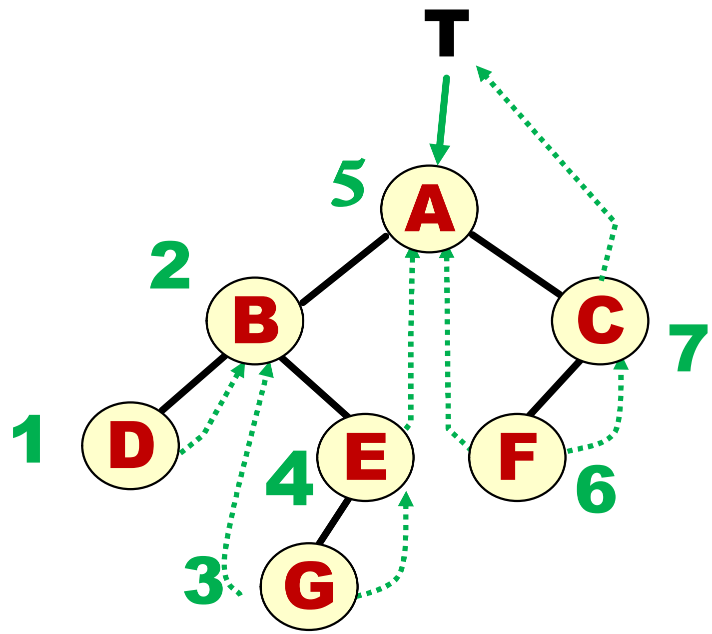
  若结点的右链域为线索，则后继结点为右链结点；

  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

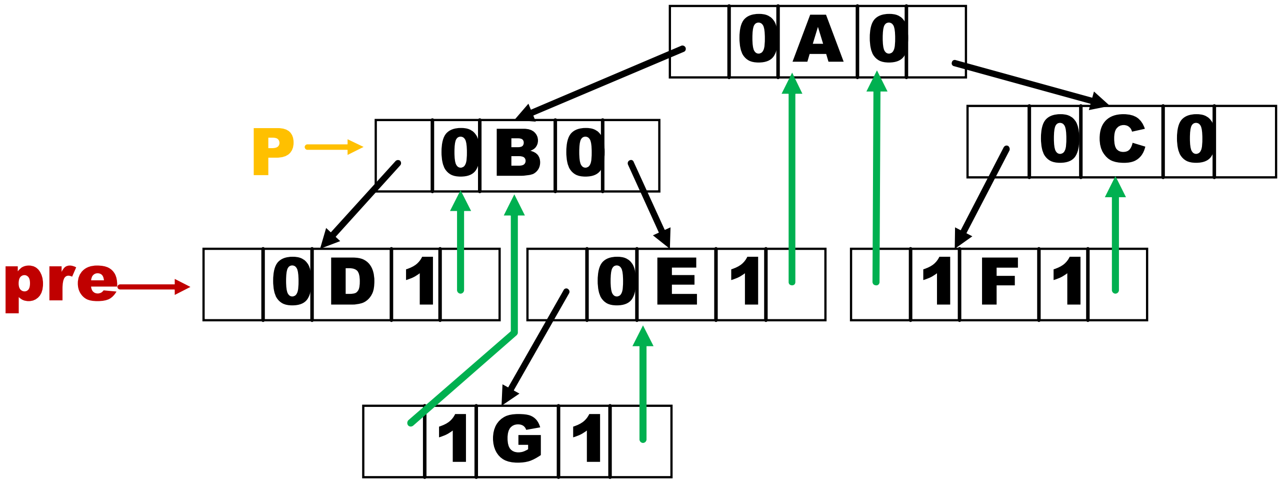
```
Status  InTra_ThrT ( BiThrTree ThrT, void(*Visit) (TElemType  e) )
{  p = ThrT ->lchild;        // p指向根结点
   while ( p != ThrT )
   {   while ( p->Ltag==link ) p=p->lchild; // 最左下结点
       Visit( p->data );        // p->Ltag== thread
       while ( p->Rtag==Thread && p->rchild!=ThrT )
       { // 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

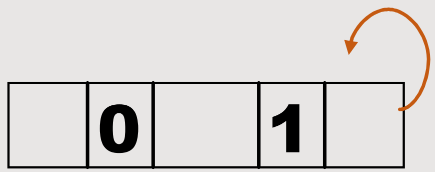
二叉树的线索化(建立中序线索二叉树)

- 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. 建头结点
   Thrt->rchild = Thrt;           // 右指针指向头结点
   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。

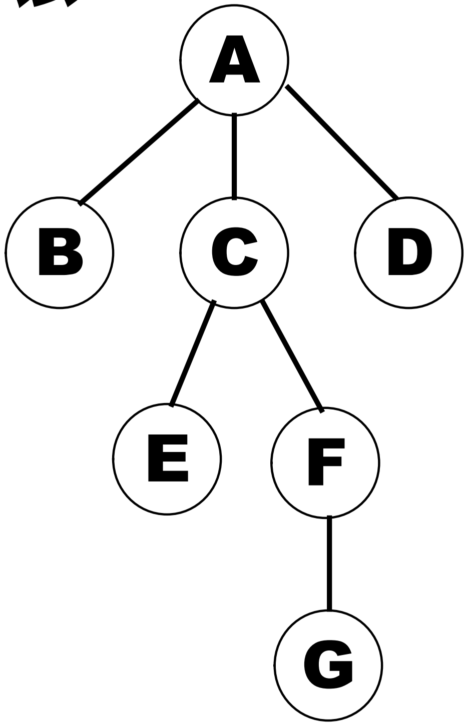
5.4 树和森林

双亲表示的类型定义

```
#define MAX_TREEE_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

树的存贮结构

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

data	child1	child2	……	childd

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。

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

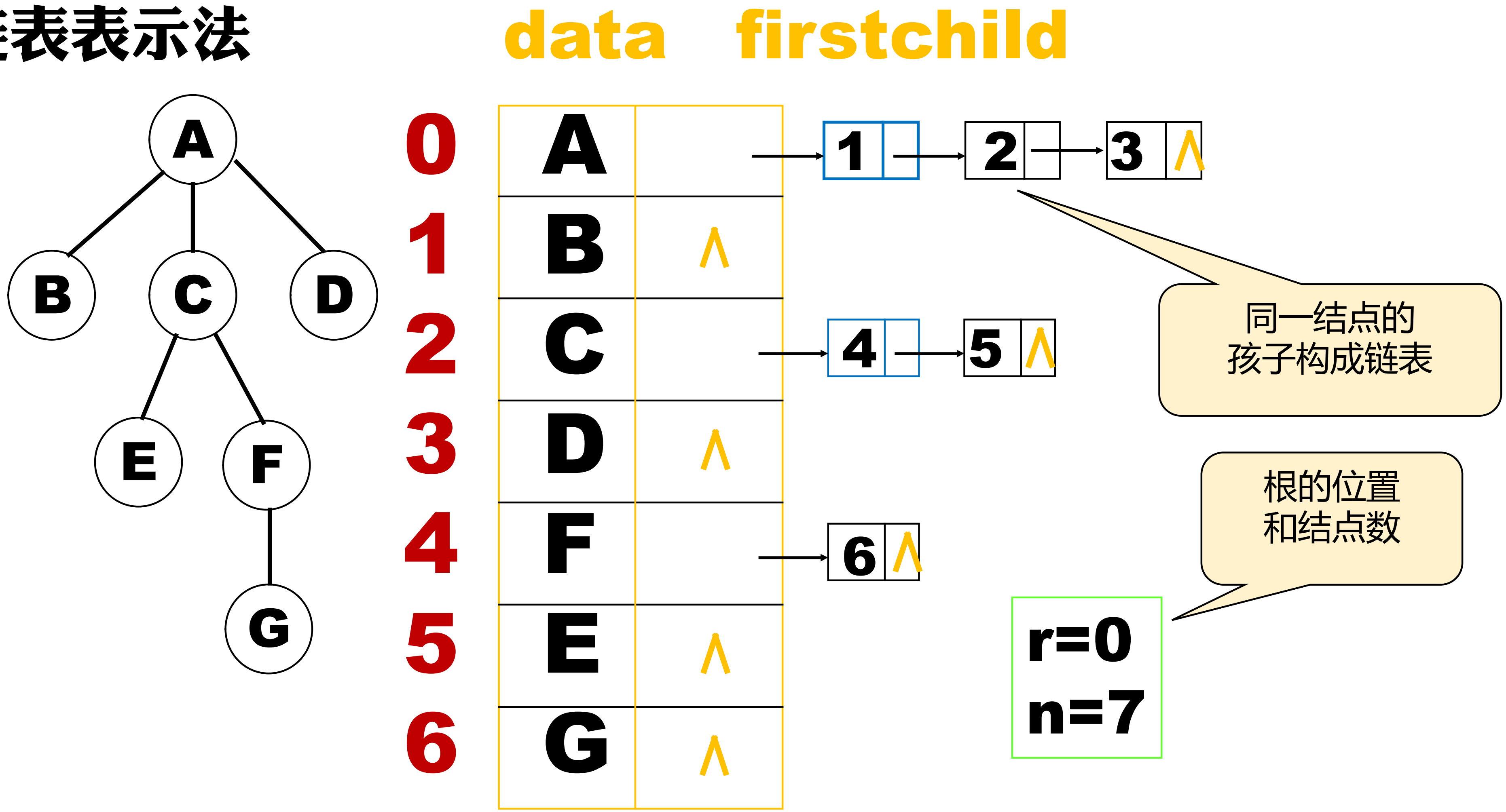
5.4 树和森林

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

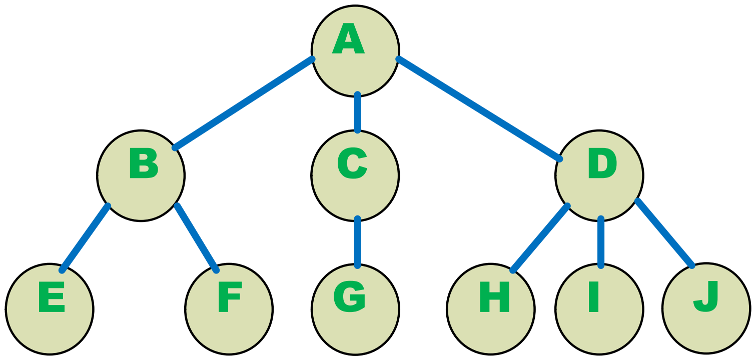


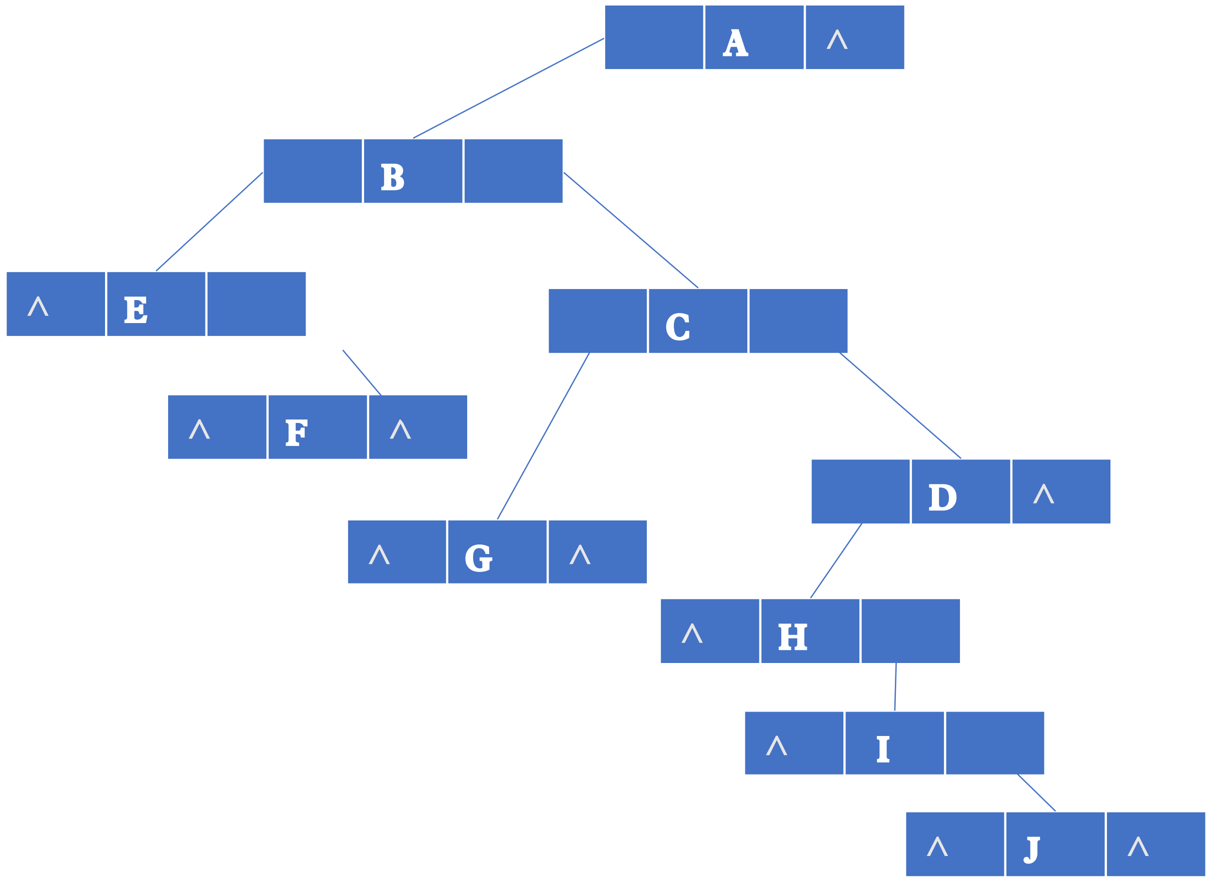
5.4 树和森林

树的孩子链表类型定义

```
typedef  struct  CTNode            // 孩子结点
{       int     child;
        struct  CTNode   *next;
} * ChildPtr;
typedef  struct
{       ElemType   data;
        ChildPtr     firstchild;       // 孩子链表头指针
} CTBox;
typedef  struct
{      CTBox  nodes [ MAX_TREE_SIZE ];
       int    n , r;                 // 结点数和根的位置
} CTree;
```

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

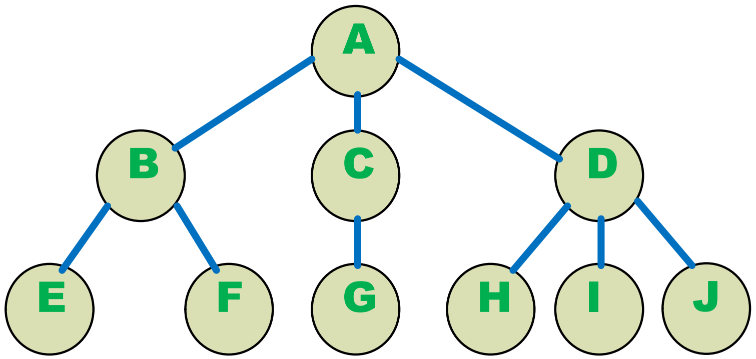
二叉树与树都可用二叉链表存贮，以二叉链表作中介，可实现树与二叉树之间的转换。

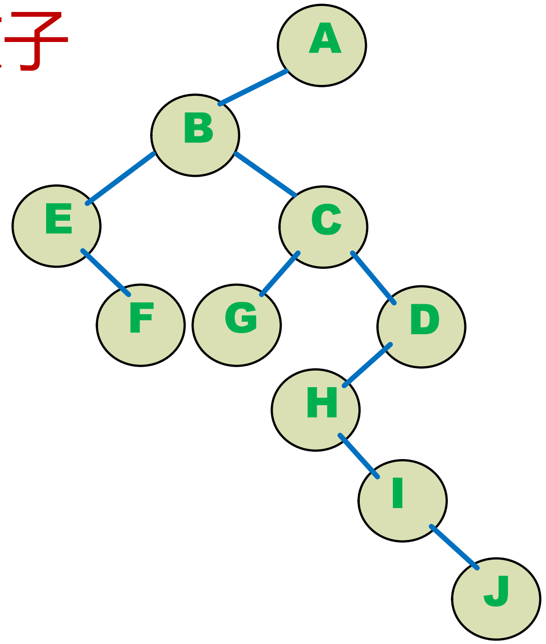
树与二叉树转换方法

树	二叉树
根	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

树		二叉树
根	⟷	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



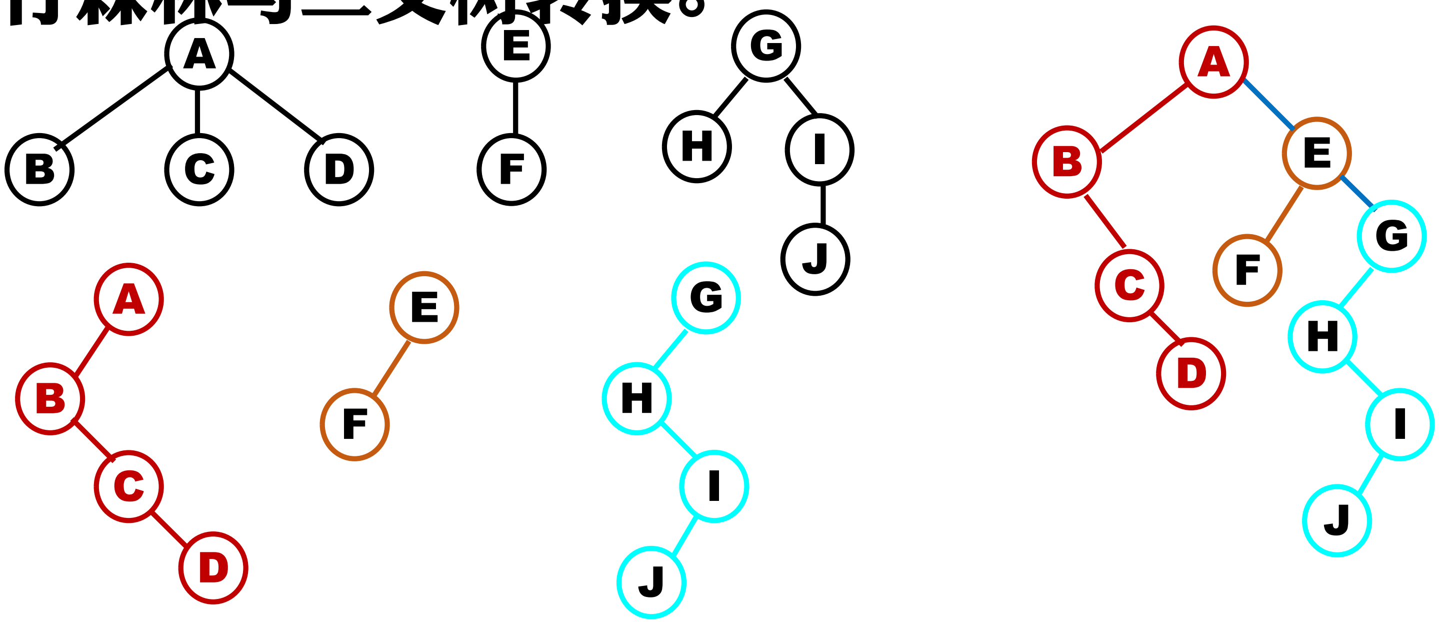


5.4 树和森林

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

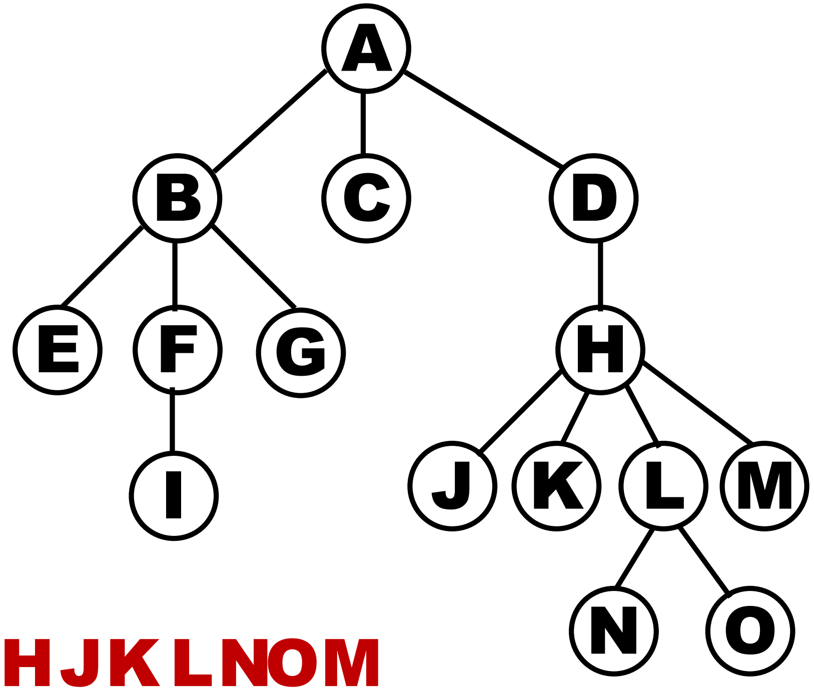
先根（序）遍历：先访问树的根结点，然后依次先根遍历根的每棵子树。

后根（序）遍历：先依次后根遍历每棵子树，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



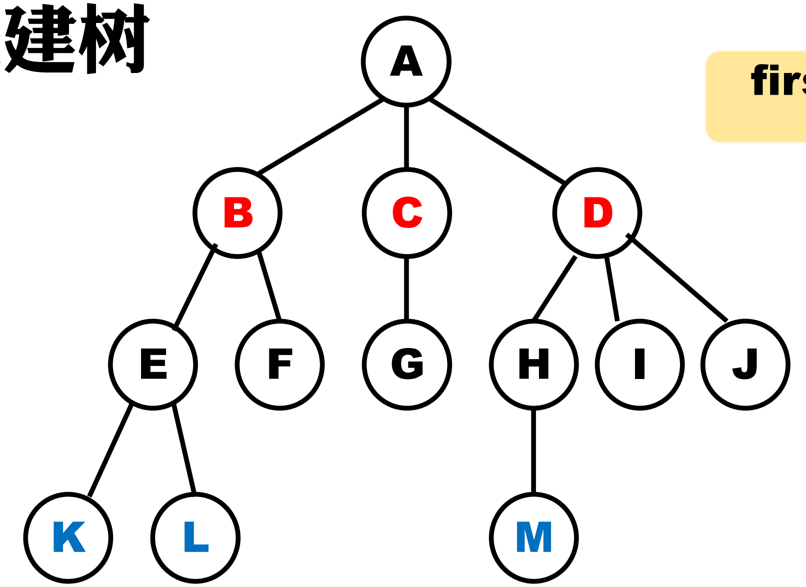
先序遍历： ABEFIGCDHJKLNOM

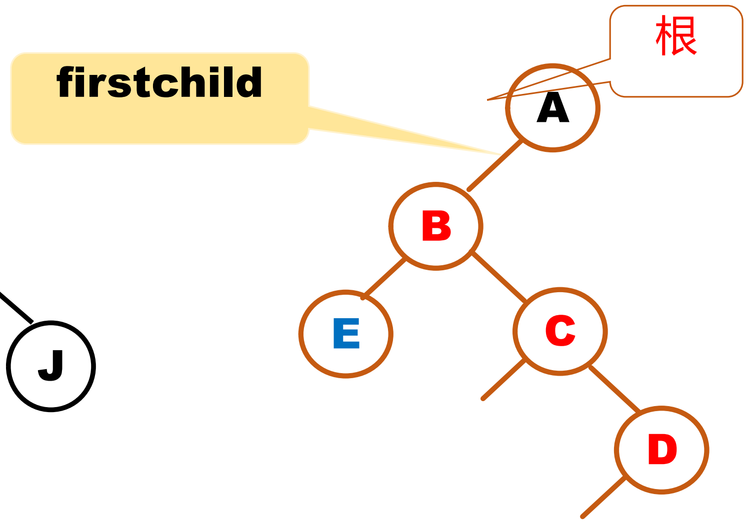
后序遍历： EIFGBCJKNOLMHDA

层次遍历： ABCDEFGHIJKLMNO

5.4 树和森林

例：由广义表建树





(A, (B, (E, (K), (L)), (F)), (C, (G)), (D, (H,(M)),(I),(J)))

根

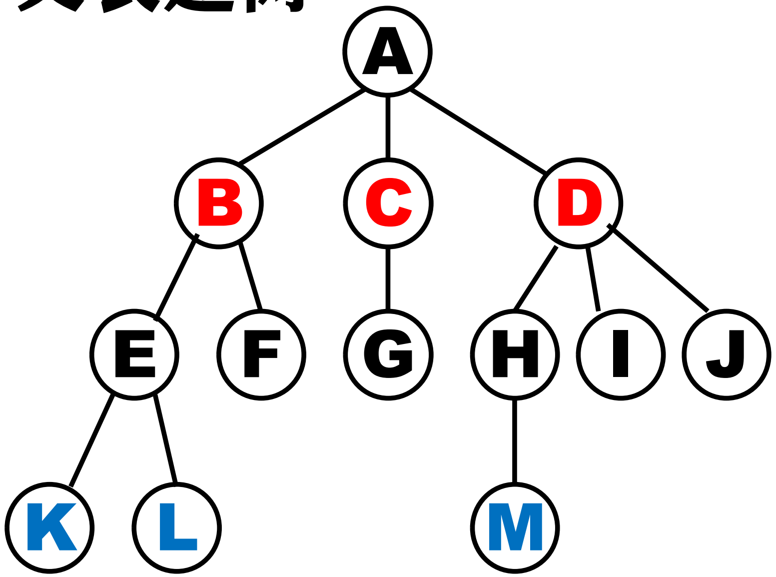
子树1

子树2

子树3

5.4 树和森林

例：由广义表建树



树根： **Head(L)**

根的第一个孩子：

Head(tail(L))

第一个孩子的下一个兄弟：

Head(tail(tail(L)))

(A, (B, (E, (K), (L)), (F)), (C, (G)), (D, (H,(M)),(I),(J)))

根

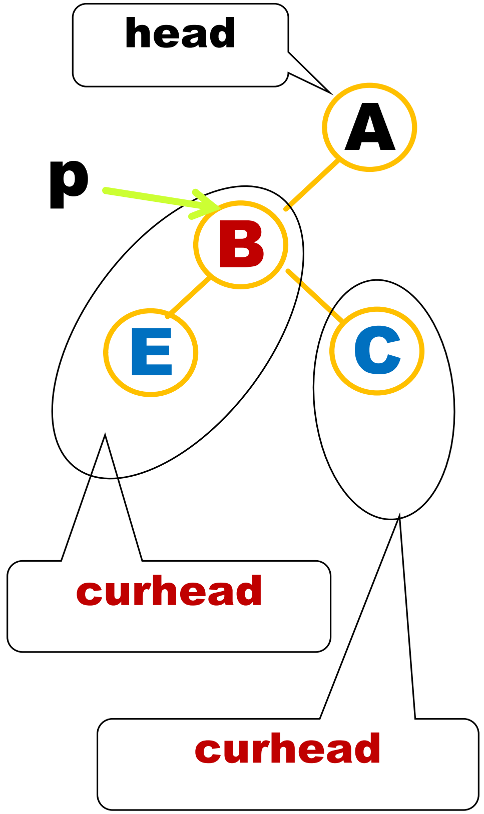
根的第一个孩子

Head(tail(L))

Head(tail(tail(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）

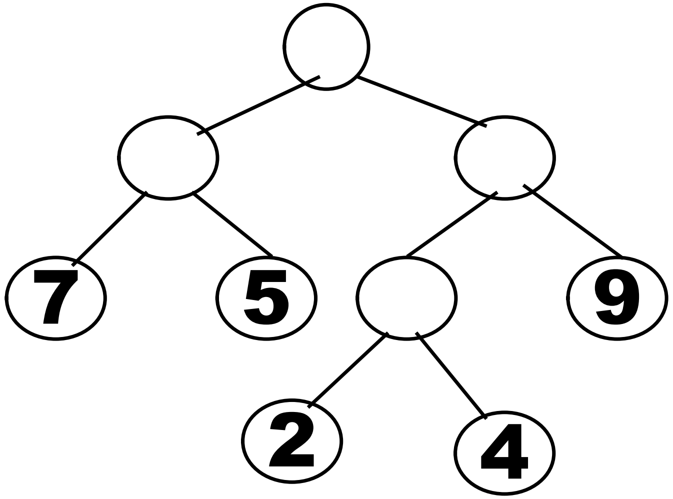
树的带权路径长度=树中所有叶子结点的带权路径之和,通常记作:

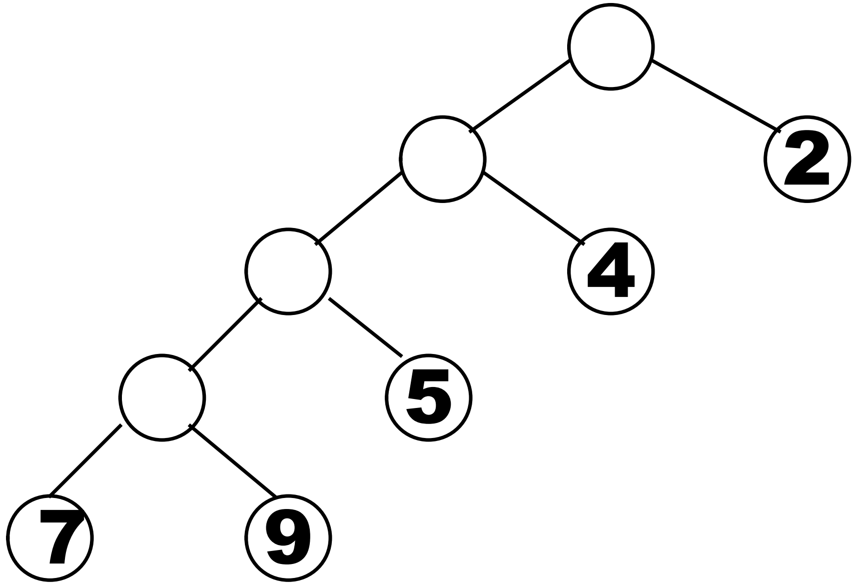
$$WPL = \sum_{k=1}^{n} W_k L_k$$

哈夫曼树：假设有n个权值（$w_1, w_2, \cdots, w_n$），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





WPL(T) =

7*2+5*2+2*3+4*3+9*2

= 60

WPL(T) =

7*4+9*4+5*3+4*2+2*1

= 89

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 $\{w_1, w_2, \cdots\cdots w_n\}$，构造n棵只有根结点的二叉树，令其权值为$w_j$

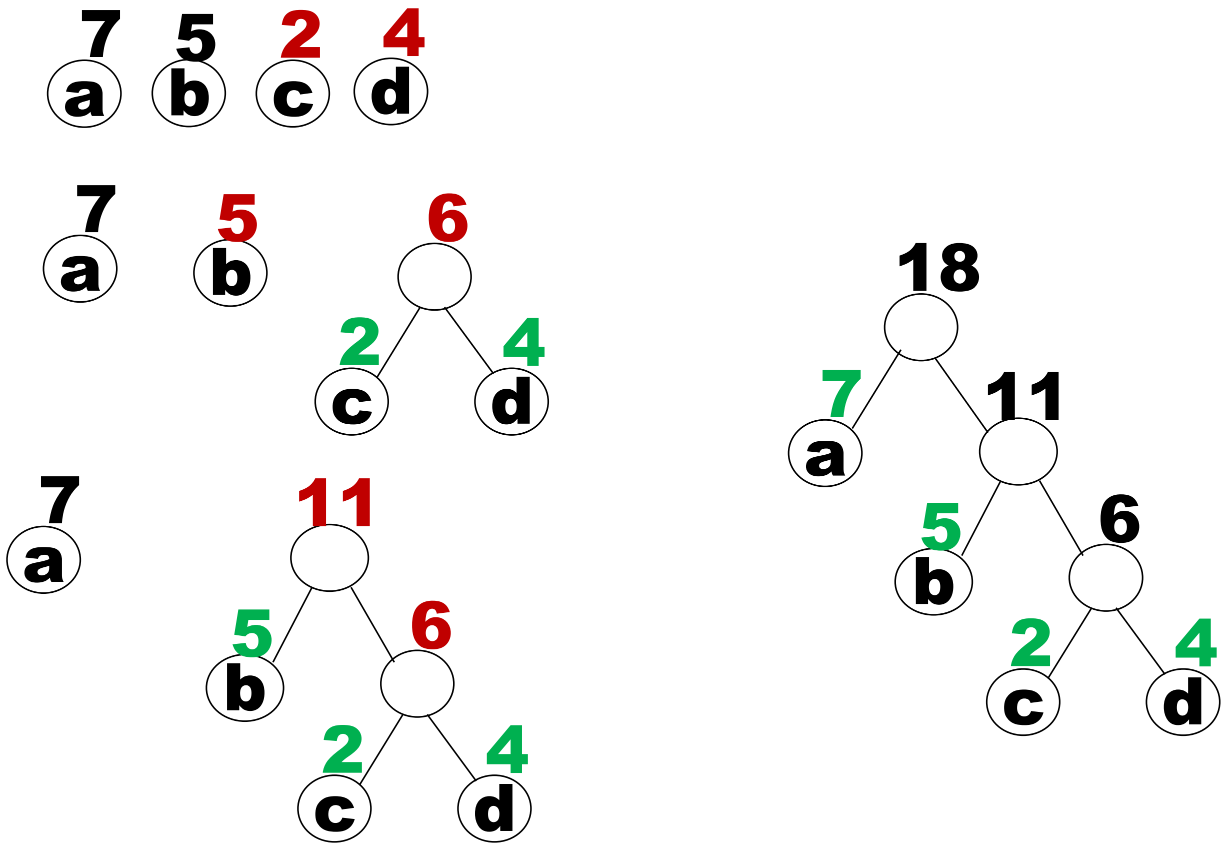
在森林中**选取两棵根结点权值最小的树**作左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和。

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。

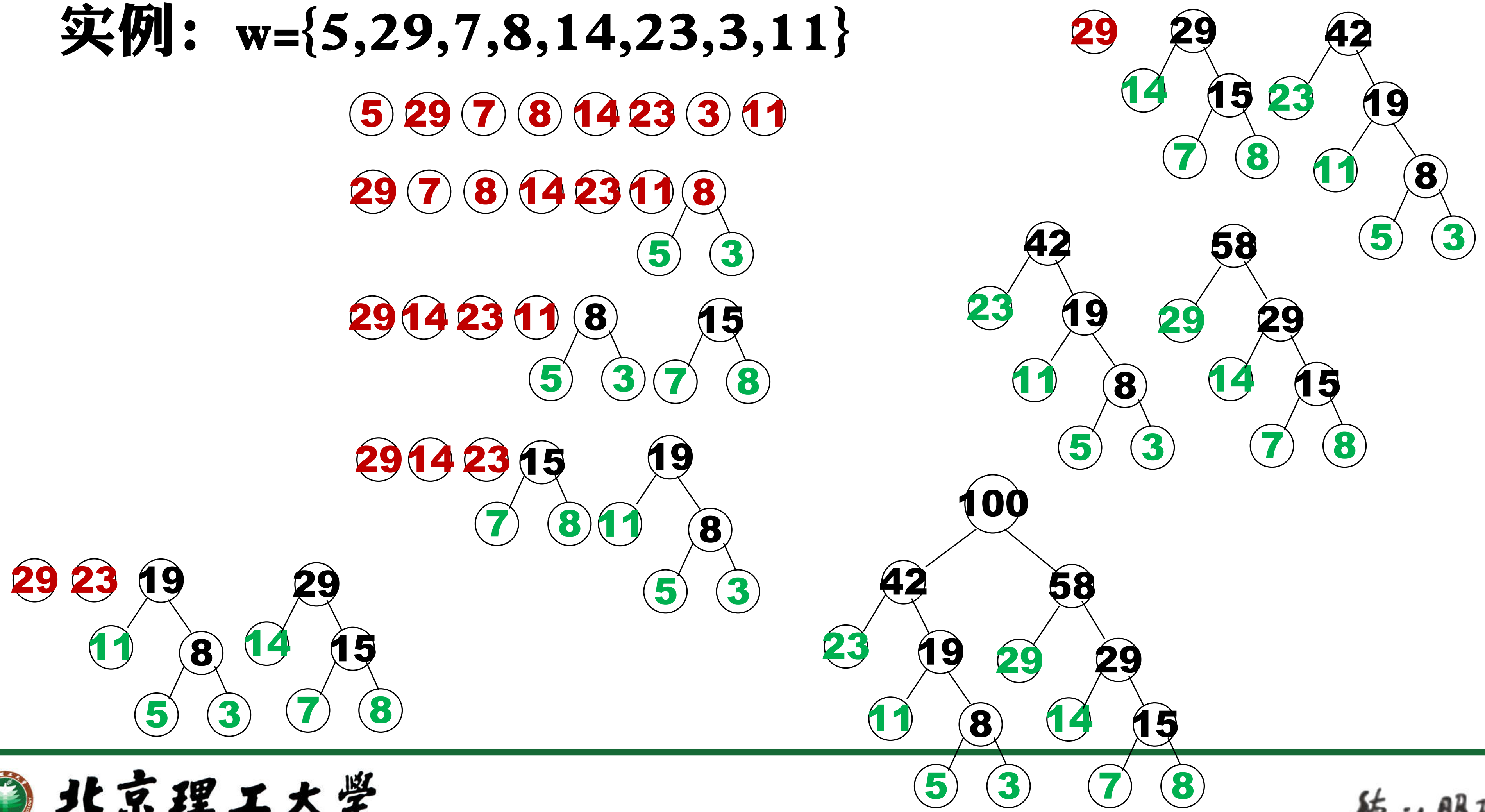
5.5 最优二叉树（赫夫曼树）

实例



5.5 最优二叉树（赫夫曼树）

实例：w={5,29,7,8,14,23,3,11}



5.5 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。

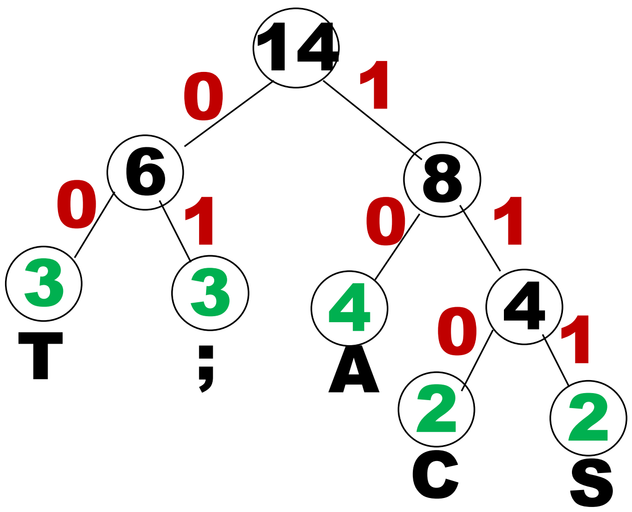
思想：根据字符出现频率编码，使电文总长最短。

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

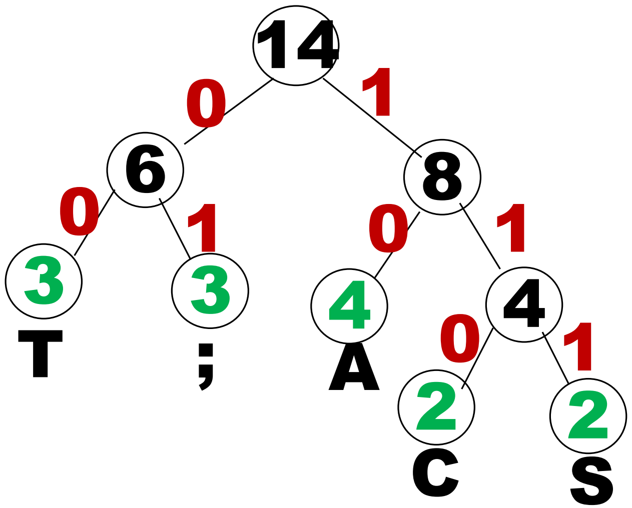
A :　10

C :　110

S :　111

5.5 最优二叉树（赫夫曼树）

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



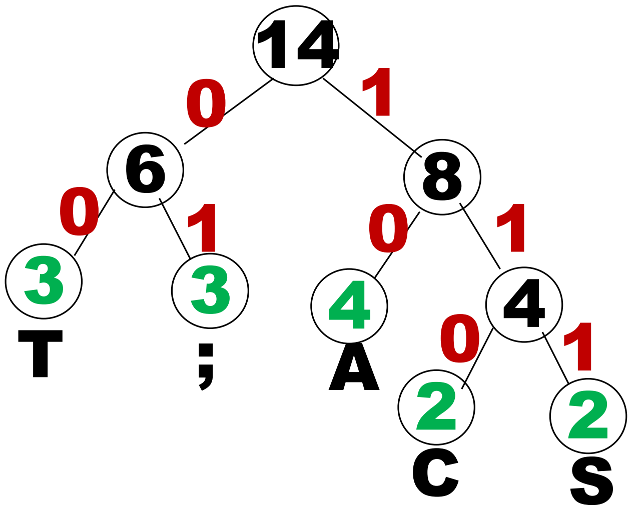
T： 00
; ： 01
A： 10
C： 110
S： 111

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 T