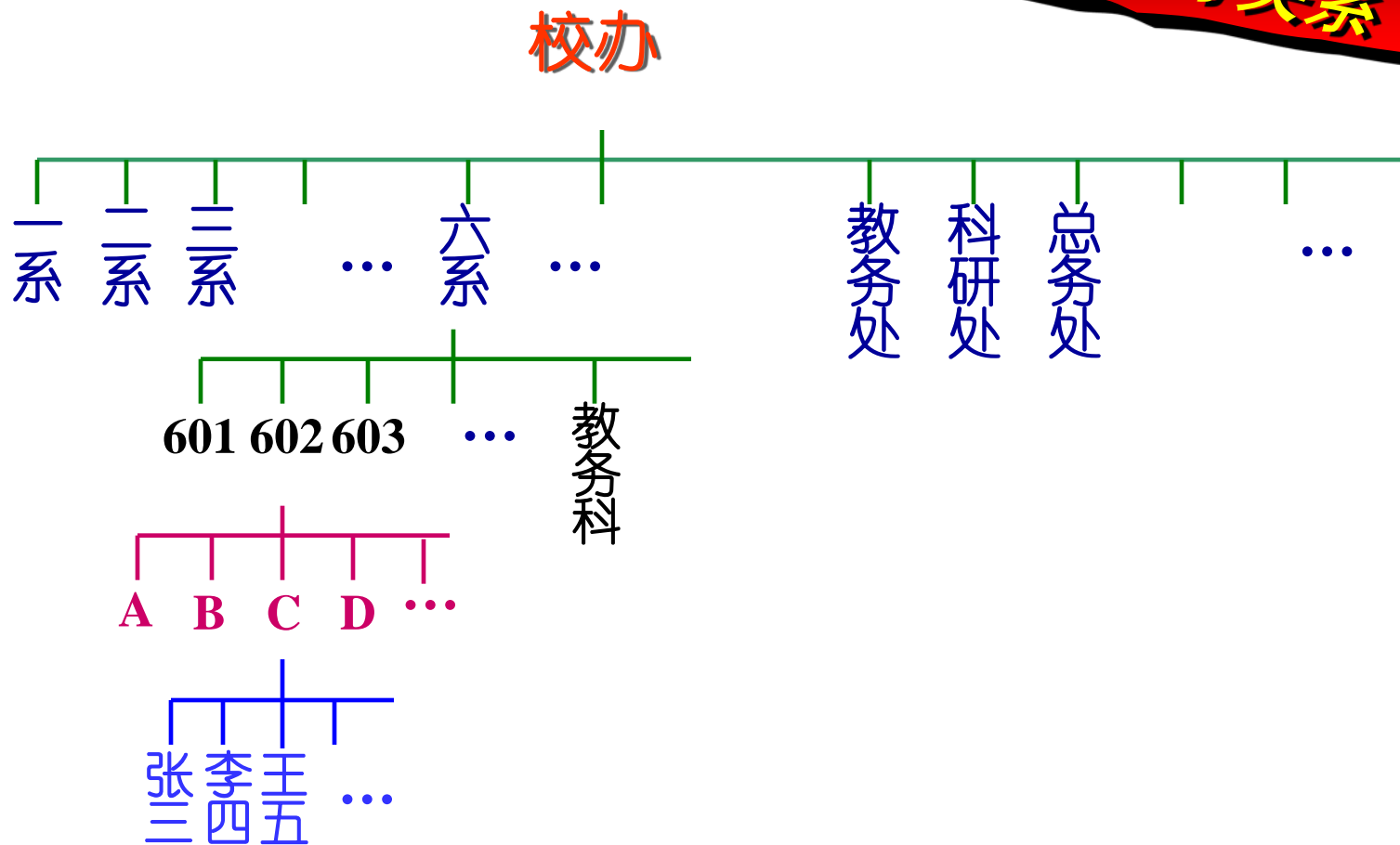


第七章 树与二叉树



例1

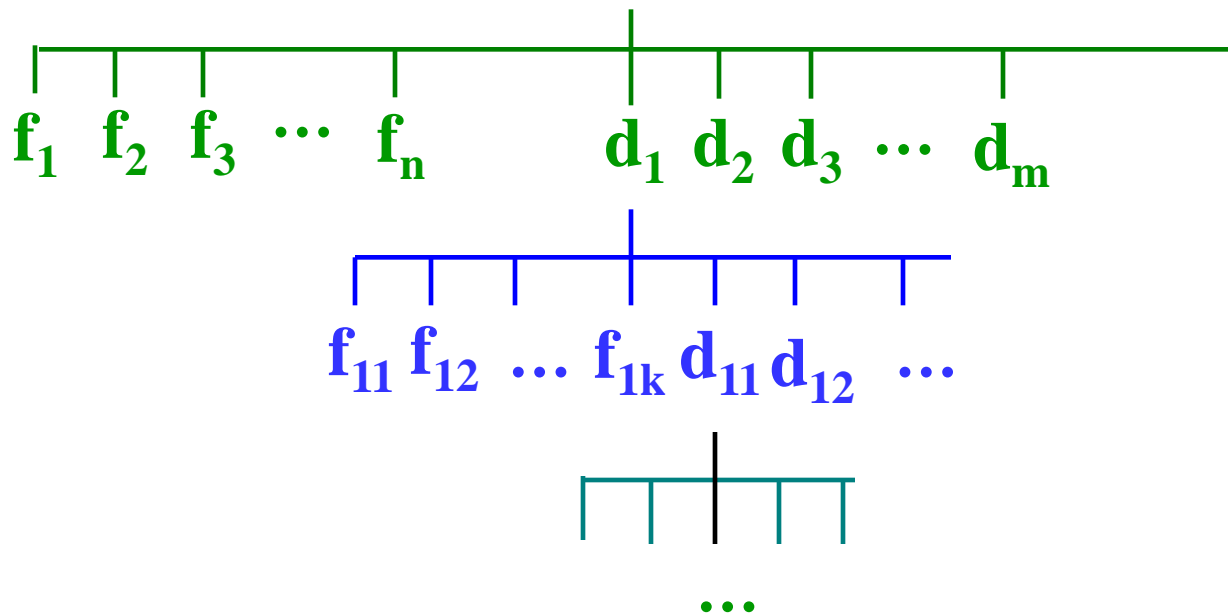
企事业单位行政领导与被领导关系



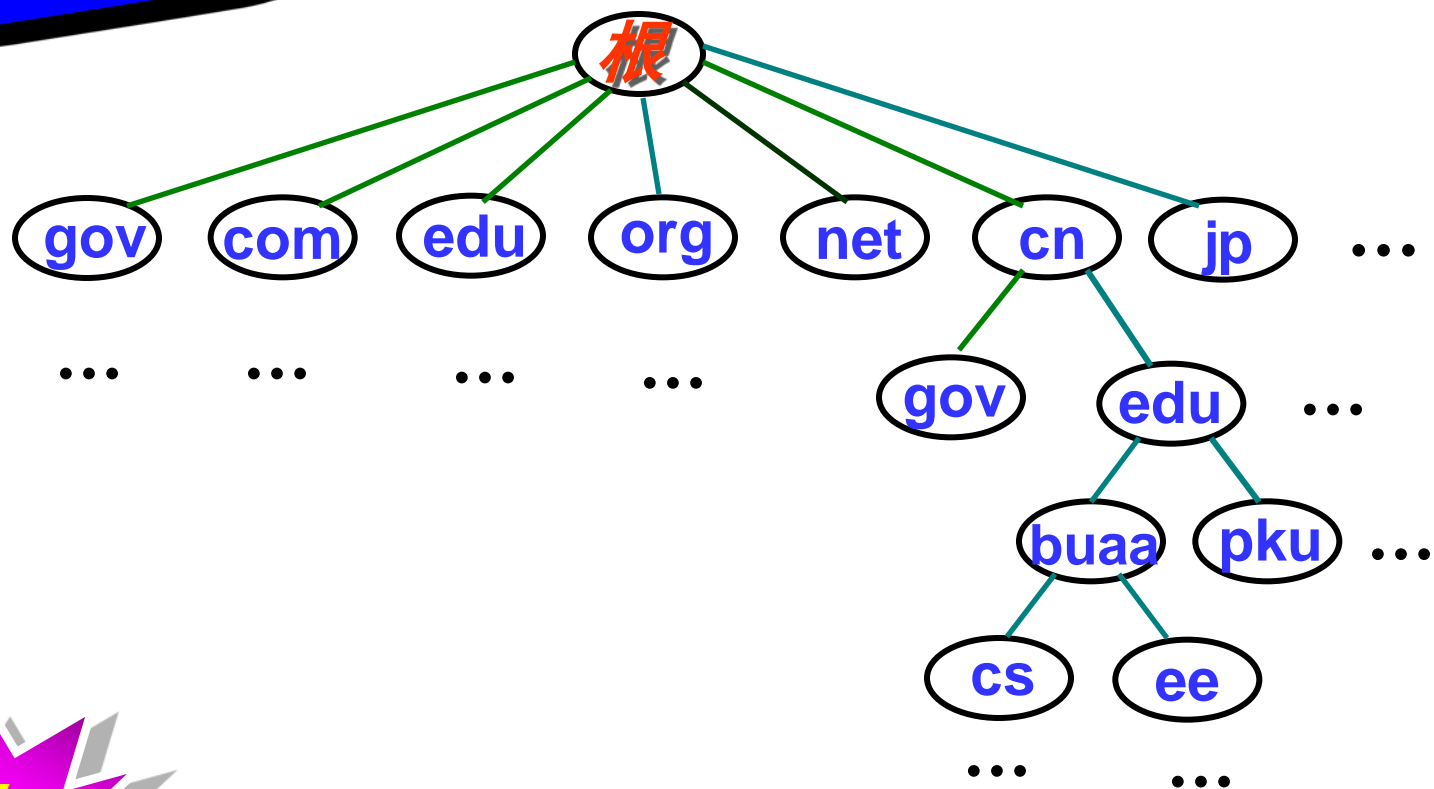
操作系统中的
文件管理

例2

(根目录)

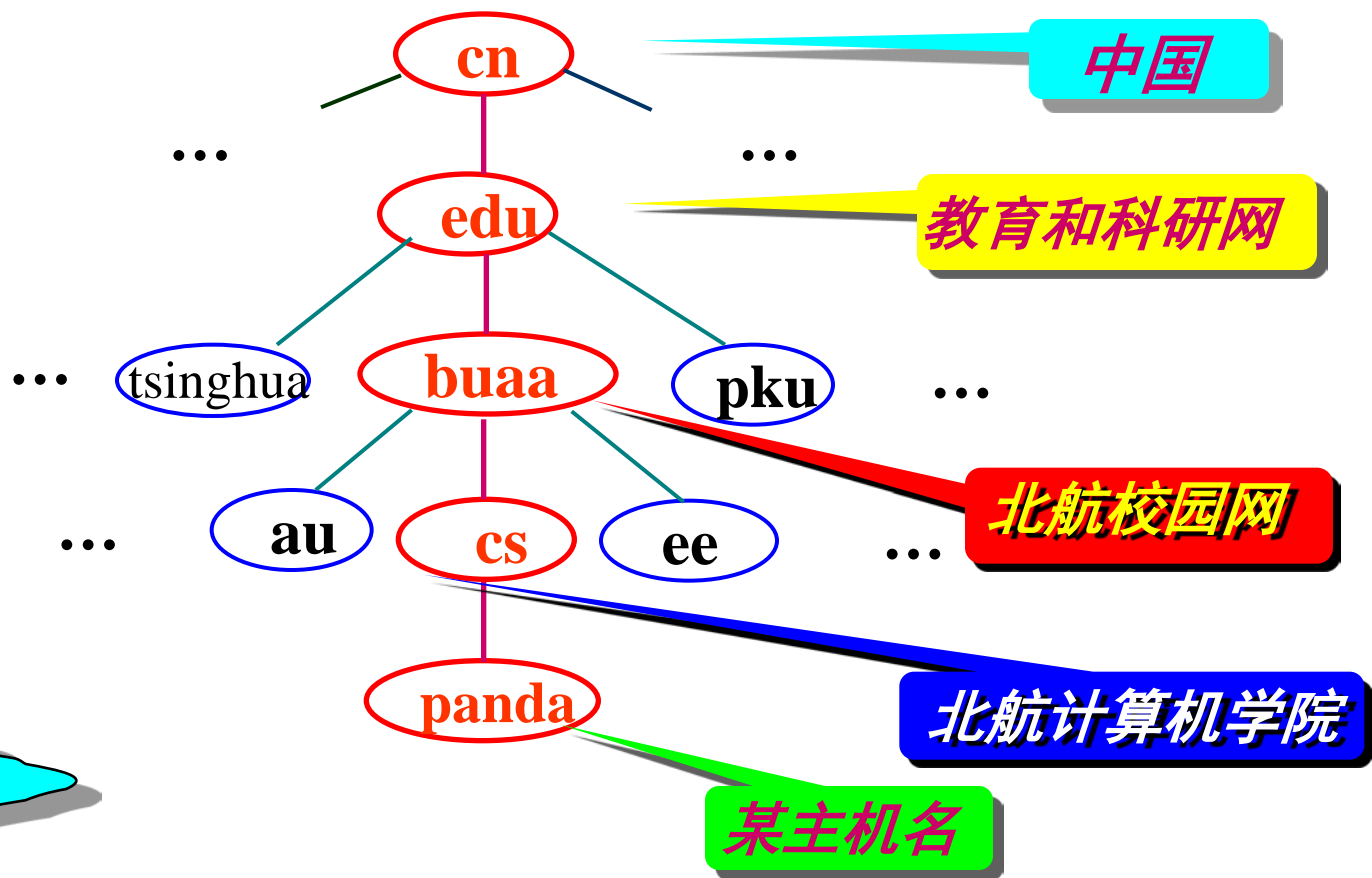


网络系统中的域名管理



例3

<http://WWW.panda.cs.buaa.edu.cn>

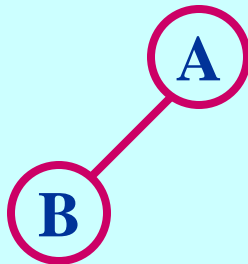


例4

一个数据元素： 一个**结点**



数据元素(结点)之间的关系：**分支**



本章内容

7.1 树的基本概念

7.2 树的存储结构

7.3 二叉树

7.4 二叉树的存储结构

7.5 二叉树的遍历

7.6 线索二叉树

7.7 二叉排序树

*7.8 平衡二叉树

7.9 哈夫曼(Huffman)树

重点

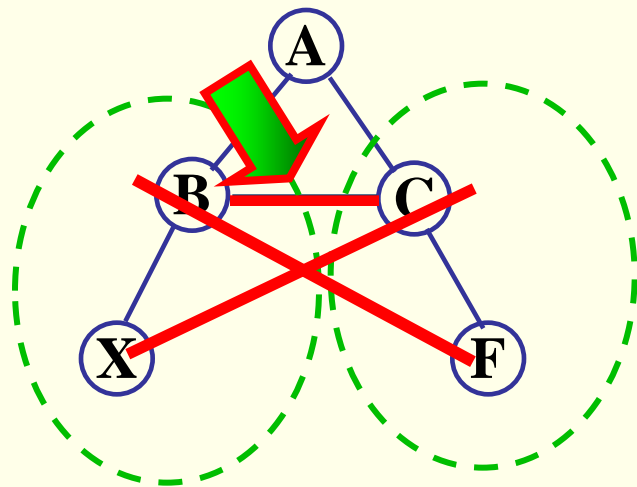
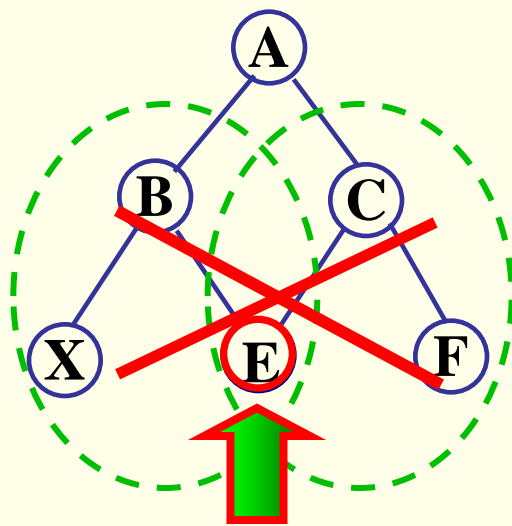
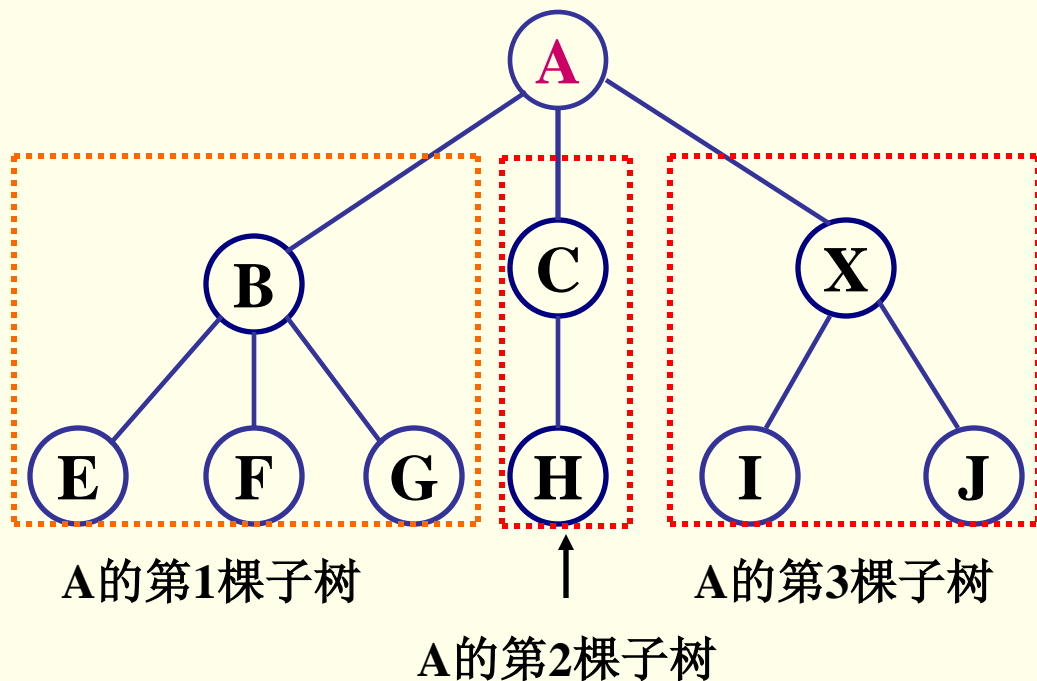
7.1 树的基本概念

一. 树的定义

树是由 $n \geq 0$ 个结点组成的有穷集合(不妨用符号 D 表示)以及结点之间关系组成的集合构成的结构, 记为 T 。当 $n=0$ 时, 称 T 为空树;

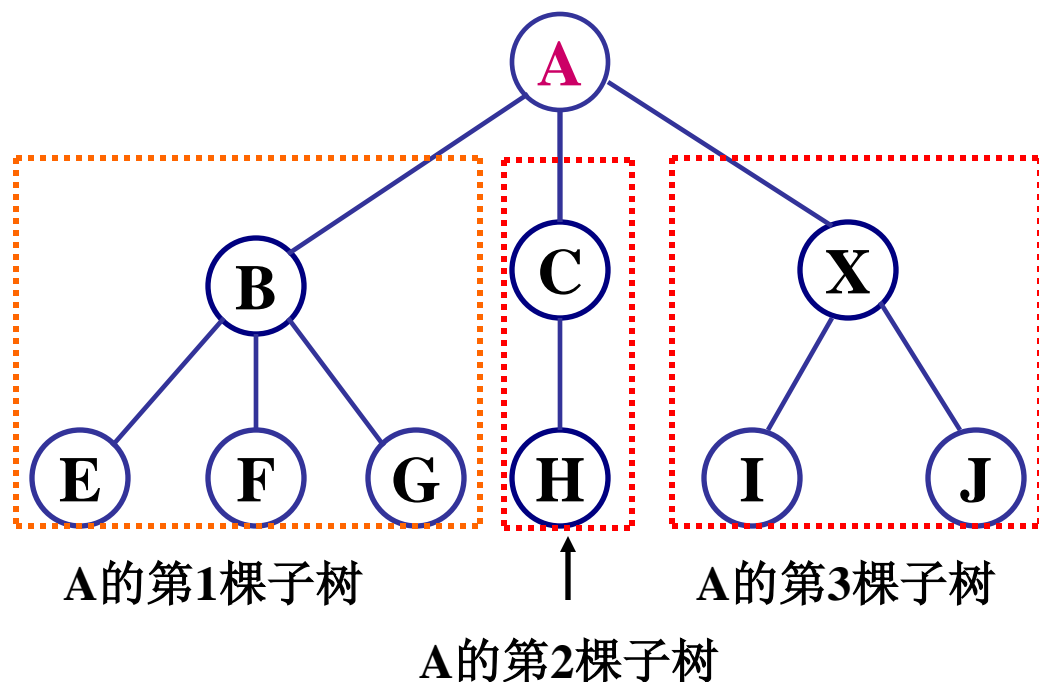
在任何一棵非空的树中, 有一个特殊的结点 $t \in D$, 称之为该树的根结点; 其余结点 $D - \{t\}$ 被分割成 $m > 0$ 个不相交的子集 D_1, D_2, \dots, D_m , 其中每一个子集 D_i 又为一棵树, 分别称之为 t 的**子树**。

递归定义



不相交的子集





二. 树(逻辑上)的特点

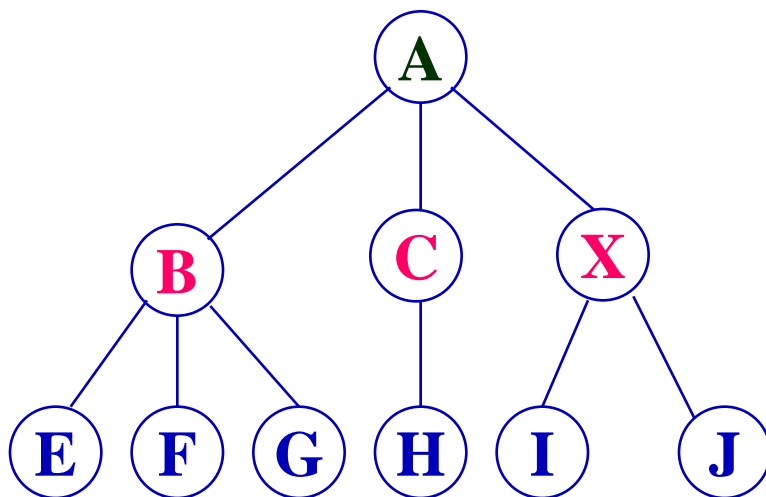
1. 有且仅有一个结点没有前驱结点, 该结点为树的根结点。
2. 除了根结点外, 每个结点有且仅有一个直接前驱结点。
3. 包括根结点在内, 每个结点可以有多个后继结点。

三. 树的逻辑表示方法

1. 文氏图表示法
2. 凹入表示法
3. 嵌套括号表示法(广义表表示法)

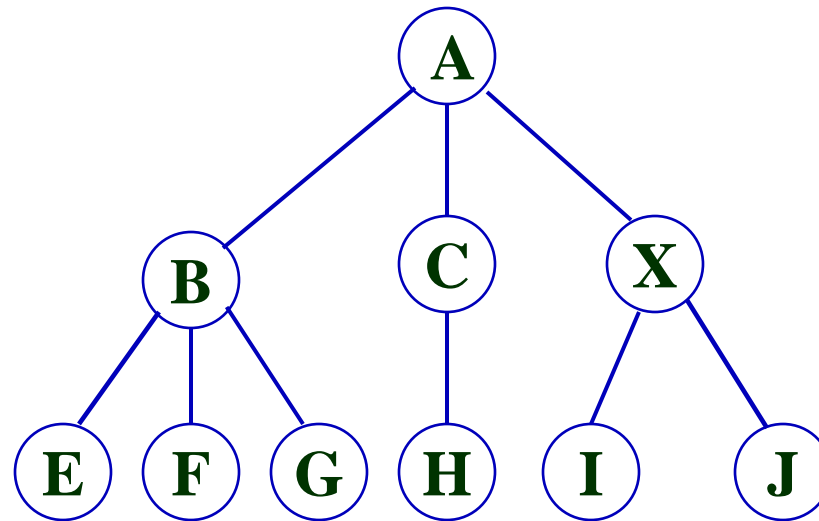
教材P161页

$A(\underline{B(E, F, G)}, \underline{C(H)}, \underline{X(I, J)})$



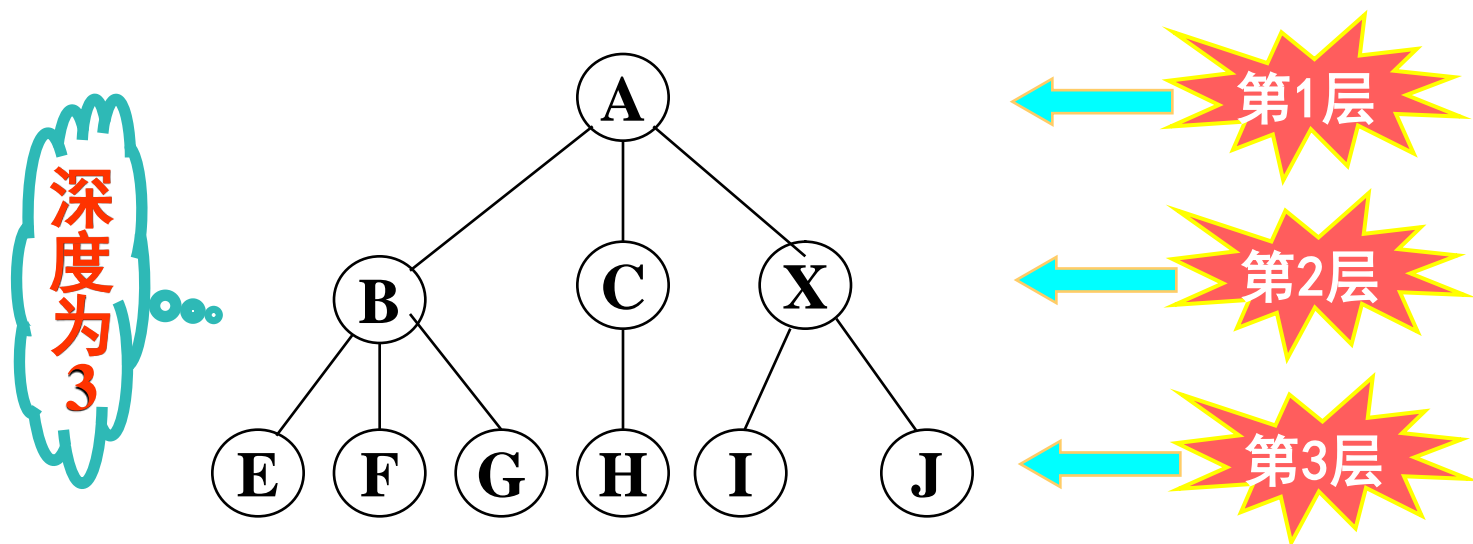
4. 树形表示法

借助自然界中一棵倒置的树的形状来表示数据元素之间层次关系的方法。



四. 基本名词术语

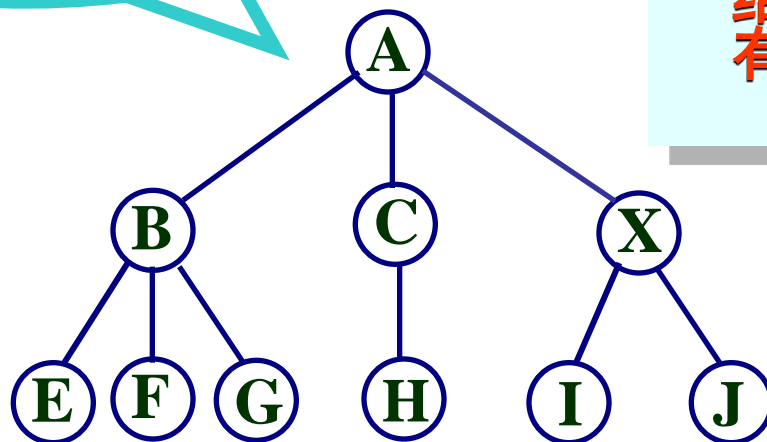
1. **结点的度**: 该结点拥有的子树的数目
2. **树的度**: 树中结点度的最大值。
3. **叶结点**: 度为0 的结点。
4. **分支结点**: 度非0 的结点。
5. **层次的定义**: 根结点为第一层, 若某结点在第 i 层, 则其孩子结点(若存在)为第 $i+1$ 层。
6. **树的深度**: 树中结点所处的最大层次数。



7. 路径：若在树中存在一个结点序列 d_1, d_2, \dots, d_j , 使得 d_i 是 d_{i+1} 的双亲($1 \leq i < j$), 则称该结点序列是从 d_1 到 d_j 的一条路径。路径的长度为 $j-1$ 。

8. 祖先与子孙：若树中结点 d 到 d_s 存在一条路径，则称 d 是 d_s 的祖先， d_s 是 d 的子孙。

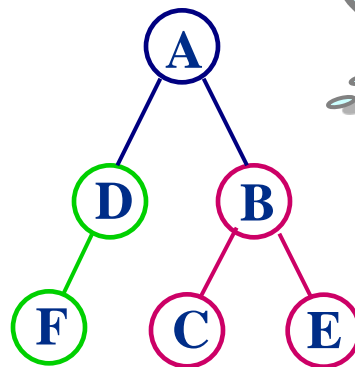
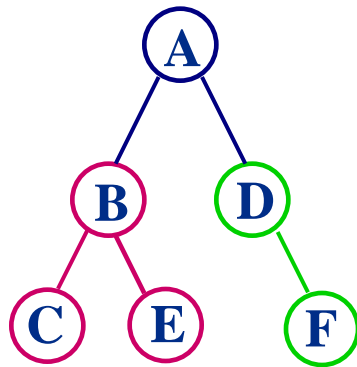
从根结点到树中
其余结点均分别存在
一条唯一路径



一个结点的祖先是根结点到该结点路径上所有经过的结点；而一个结点的子孙则是以该结点为根的子树上的所有其他结点。

9. 树林(森林): $m \geq 0$ 棵不相交的树组成的树的集合。

10. 树的有序性: 若树中结点的子树的相对位置不能随意改变, 则称该树为**有序树**, 否则称该树为**无序树**。



7.2 树的存储结构

主要取决于要对树进行何种操作

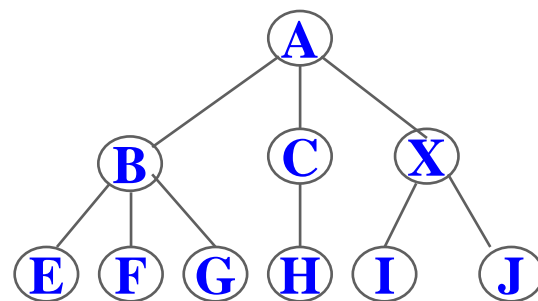
- 顺序存储结构
- 链式存储结构（居多）

无论采用何种存储结构，需要存储的信息有：

- 结点本身的数据信息；
- 结点之间存在的关系（分支）。

一. 多重链表结构

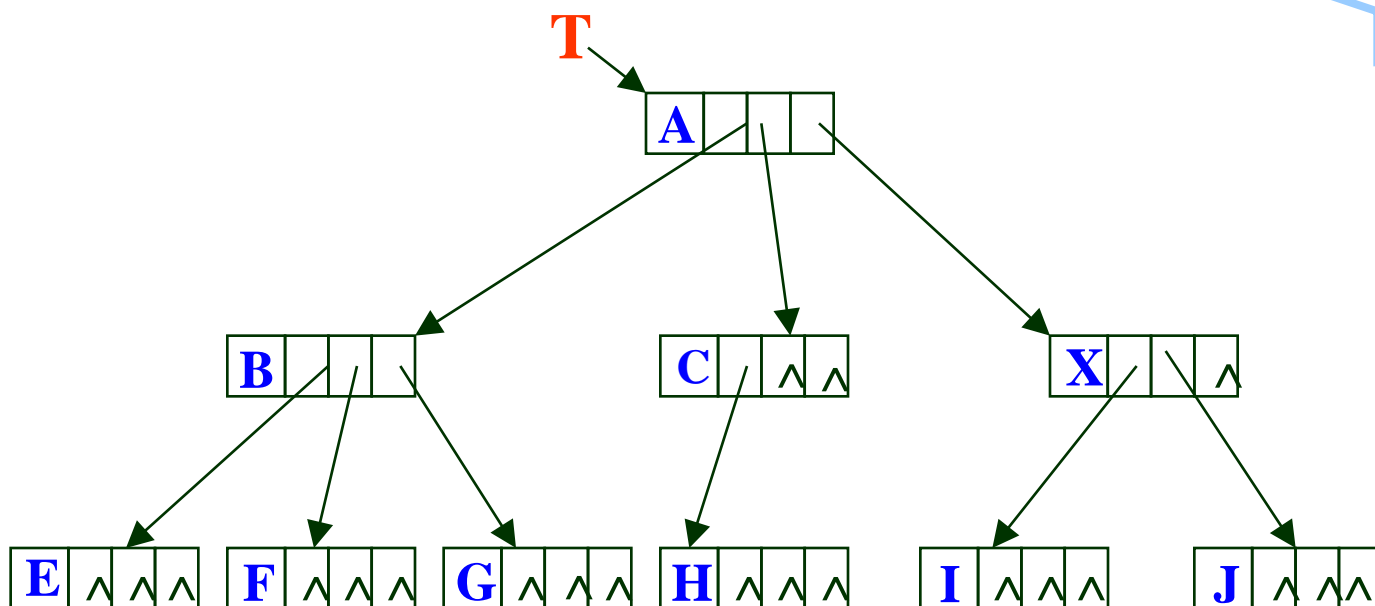
1. 定长结点的多重链表结构



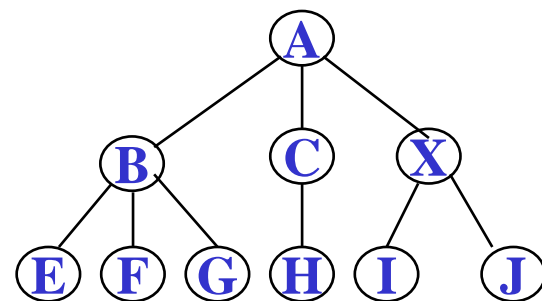
链结点的构造



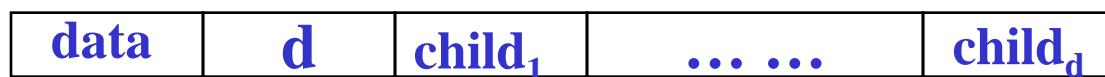
*n*为树的度



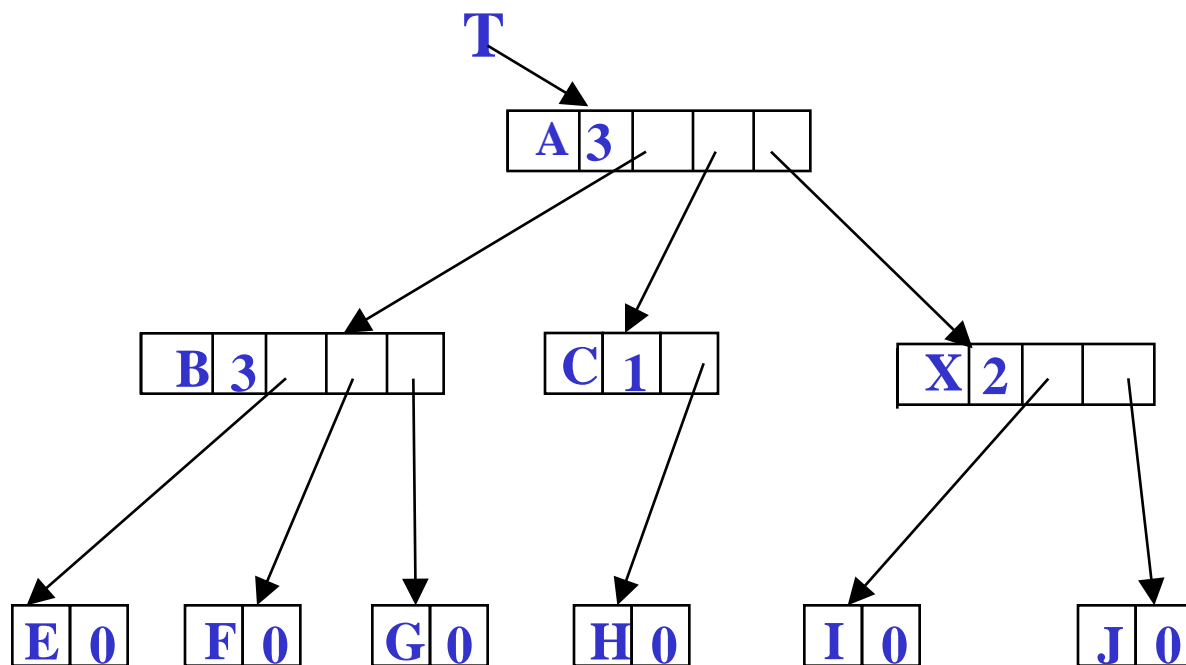
2. 不定长结点的多重链表结构



链结点的构造



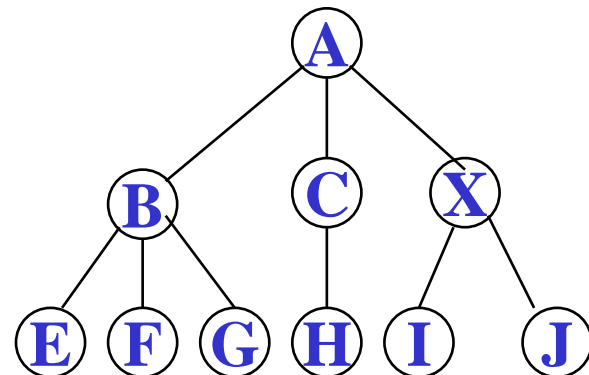
*d*为该结点的度



二. 三重链表结构

链结点的构造

data	child	parent	brother
------	-------	--------	---------

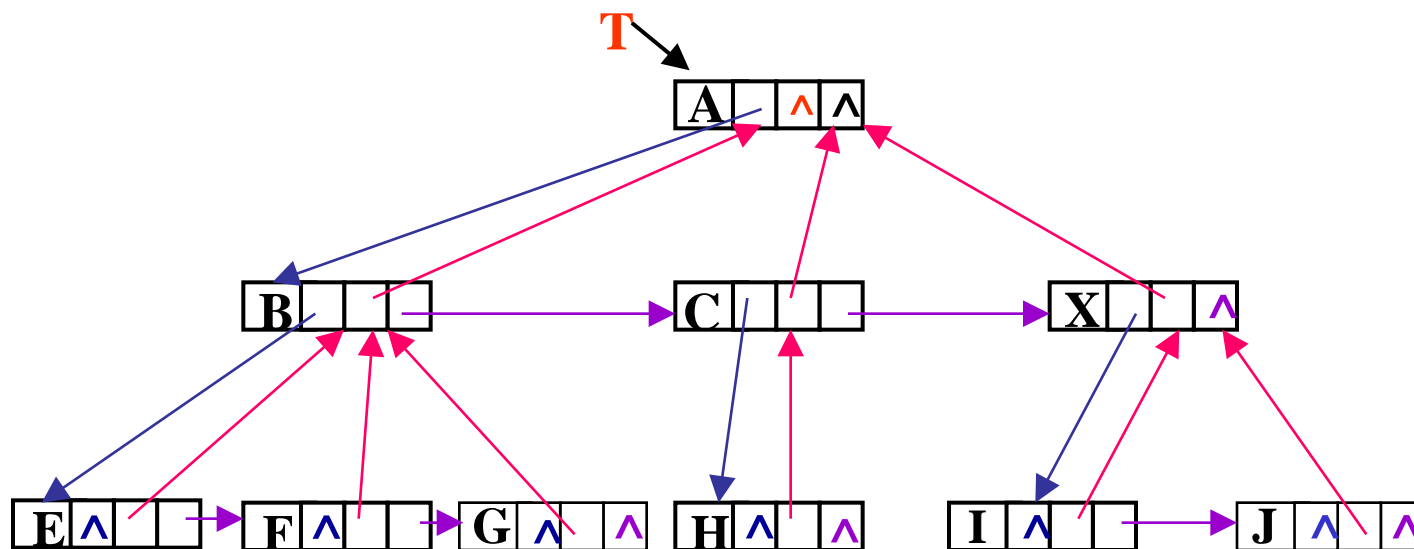


其中, data 为数据域;

child 为指针域, 指向该结点的第1个孩子结点;

parent 为指针域, 指向该结点的双亲结点;

brother 为指针域, 指向右边第一个兄弟结点.



7.3 二叉树

一. 二叉树的定义

二叉树是有序树

二叉树 是 $n \geq 0$ 个结点的有穷集合 D 与 D 上关系的集合 R 构成的结构。当 $n=0$ 时，称该二叉树为空二叉树；否则，它为包含了一个根结点以及两棵不相交的、分别称之为**左子树**与**右子树**的二叉树。

递归定义

二叉树的基本形态: 5种!

(空)

根

根

根

根

左子树

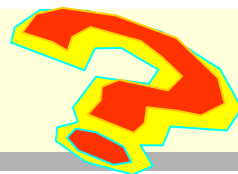
右子树

左子树

右子树

思考

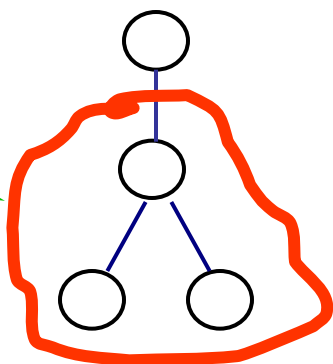
下面的说法正确与否



✗ 1. 度为2的**树**是二叉树。

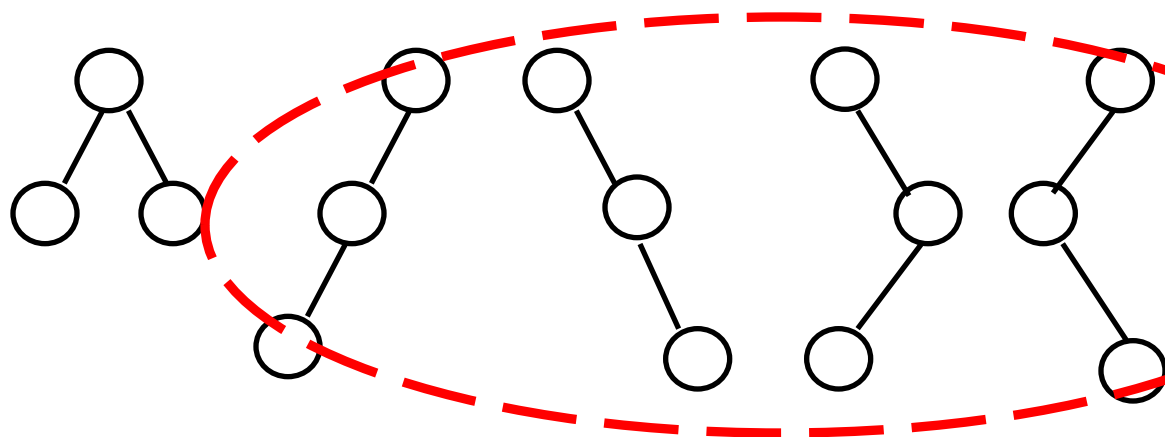
✗ 2. 度为2的**有序树**是二叉树。

没有区分
左右子树



结论: 子树有严格的
左右之分且度 ≤ 2 的
树是二叉树。

✗ 3. 具有三个结点的**树**可以有五种形态:

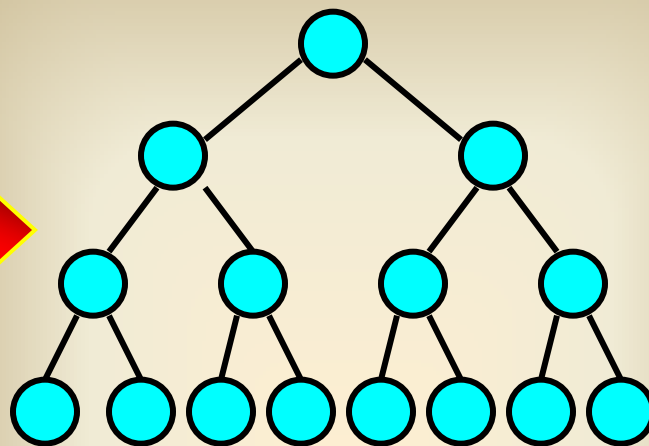


1种

二. 两种特殊形态的二叉树

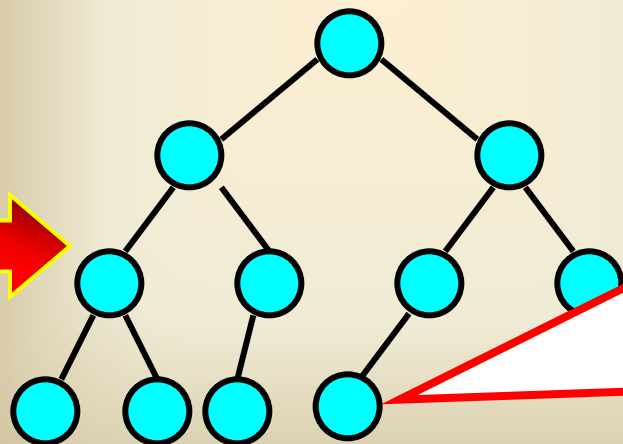
1. 满二叉树

若一棵二叉树中的结点, 或者为叶结点, 或者具有两棵非空子树, 并且叶结点都集中在二叉树的最下面一层. 这样的二叉树为满二叉树.



2. 完全二叉树

若一棵二叉树中只有最下面两层的结点的度可以小于2, 并且最下面一层的结点(叶结点)都依次排列在该层从左至右的位置上。这样的二叉树为完全二叉树.



是完全二叉树?

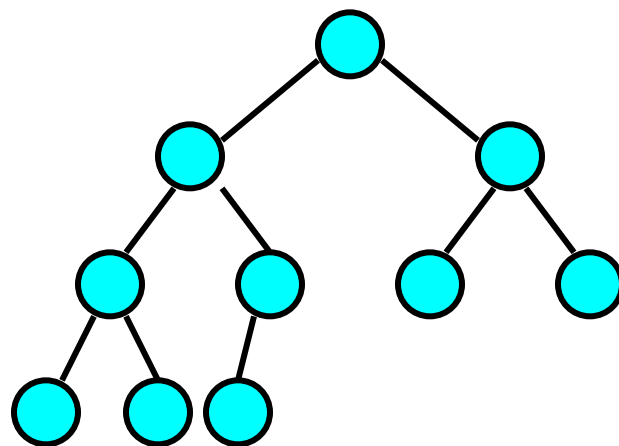
三. 二叉树的性质

1. 具有 n 个结点的非空二叉树共有 $n-1$ 个分支。

证明：

除了根结点以外，每个结点有且仅有一个双亲结点，即每个结点与其双亲结点之间仅有一个分支存在，因此，具有 n 个结点的非空二叉树的分支总数为 $n-1$ 。

证毕



2. 非空二叉树的第*i*层最多有 2^{i-1} 个结点($i \geq 1$)。

证明 (采用归纳法)

- (1). 当 $i=1$ 时, 结论显然正确。非空二叉树的第1层有且仅有一个结点, 即树的根结点。
- (2). 假设对于第*j*层 ($1 \leq j \leq i-1$) 结论也正确, 即第*j*层最多有 2^{j-1} 个结点。
- (3). 由定义可知, 二叉树中每个结点最多只能有两个孩子结点。若第*i-1*层的每个结点都有两棵非空子树, 则第*i*层的结点数目达到最大. 而第*i-1*层最多有 2^{i-2} 个结点已由假设证明, 于是, 应有

$$2 \times 2^{i-2} = 2^{i-1}$$

证毕。

3. 深度为h 的非空二叉树最多有 2^h-1 个结点。

证明：

由性质2可知, 若深度为h的二叉树的每一层的结点数目都达到各自所在层的最大值, 则二叉树的结点总数一定达到最大。即有

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

证毕

思考

深度为h 的完全二叉树至少有多少个结点?

结论：深度为h且具有 2^h-1 个结点的二叉树为满二叉树

$$2^h-1$$

4. 若非空二叉树有 n_0 个叶结点, 有 n_2 个度为2的结点, 则 $n_0 = n_2 + 1$

证明:

设该二叉树有 n_1 个度为1的结点, 结点总数为 n , 有

$$n = n_0 + n_1 + n_2 \quad \text{----- (1)}$$

设二叉树的分支数目为 B , 根据性质1, 有 $B = n - 1$ ----- (2)

即 这些分支来自度为1的结点与度为2结点,

$$B = n_1 + 2n_2 \quad \text{----- (3)}$$

联列关系 (1), (2) 与 (3), 可得到 $n_0 = n_2 + 1$

证毕。

推论: 对于一般的树 (度为 m), 有
$$n_0 = n_2 + 2n_3 + 3n_4 + \dots + (m-1)n_m + 1$$

5. 具有 n 个结点的非空完全二叉树的深度为
 $h = \lfloor \log_2 n \rfloor + 1$.

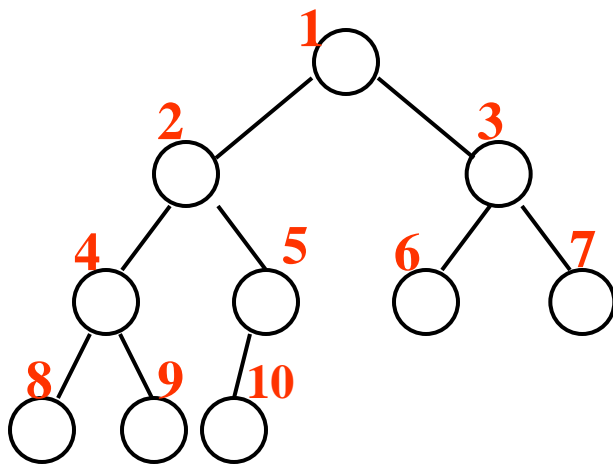
证明：
(略)

思考

具有 n 个结点的二叉树
的最小深度是多少？

6. 若对具有 n 个结点的**完全二叉树**按照层次从上到下, 每层从左到右的顺序进行编号, 则编号为 i 的结点具有以下性质:
- (1) 当 $i=1$, 则编号为 i 的结点为二叉树的根结点;
若 $i>1$, 则编号为 i 的结点的双亲的编号为 $\lfloor i/2 \rfloor$;
 - (2) 若 $2i>n$, 则编号为 i 的结点无左子树;
若 $2i\leq n$, 则编号为 i 的结点的左孩子的编号为 $2i$;
 - (3) 若 $2i+1>n$, 则编号为 i 的结点无右子树;
若 $2i+1\leq n$, 则编号为 i 的结点的右孩子的编号为 $2i+1$ 。

$n=10$





若一棵二叉树有1001个结点, 且无度为1的结点, 则叶结点的个数为__**D**__

- A) 498 B) 499 C) 500 D) 501

若一棵深度为6的完全二叉树的第6层有3个叶结点, 则该二叉树共有叶结点的个数为__**A**__

- A) 17 B) 18 C) 19 D) 20

四. 二叉树的操作

若 x 是二叉树的根结点，
或二叉树中不存在结点 x ，
则返回“空”

1. **INITIAL(T)** 创建一颗二叉树
2. **ROOT(T)** 或 **ROOT(x)** 求二叉树的根结点，或求结点 x 所在二叉树的根结点。
3. **PARENT(T,x)** 求二叉树 T 中结点 x 的双亲结点。
4. **LCHILD(T,x)** 或 **RCHILD(T,x)** 分别求二叉树 T 中结点 x 的左孩子或右孩子结点。
5. **LDELETE(T,x)** 或 **RDELETE (T,x)** 分别删除二叉树 T 中以结点 x 为根的左子树或右子树。
6. **TRAVERSE(T)** 按照某种次序（或原则）依次访问二叉树 T 中各个结点，得到由该二叉树的所有结点组成的序列。
7. **LAYER(T,x)** 求二叉树 T 中结点 x 所处的层次。
8. **DEPTH(T)** 求二叉树 T 的深度。
9. **DESTROY(T)** 销毁一颗二叉树

.....

删除 T 的所有结点，并
释放结点空间，使之成
为一颗空二叉树

重
点

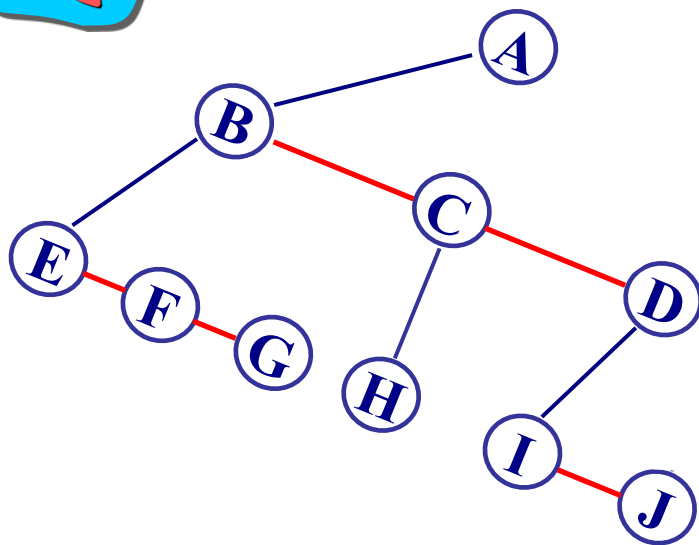
五. 二叉树与树、树林之间的转换

1. 树与二叉树的转换

步骤

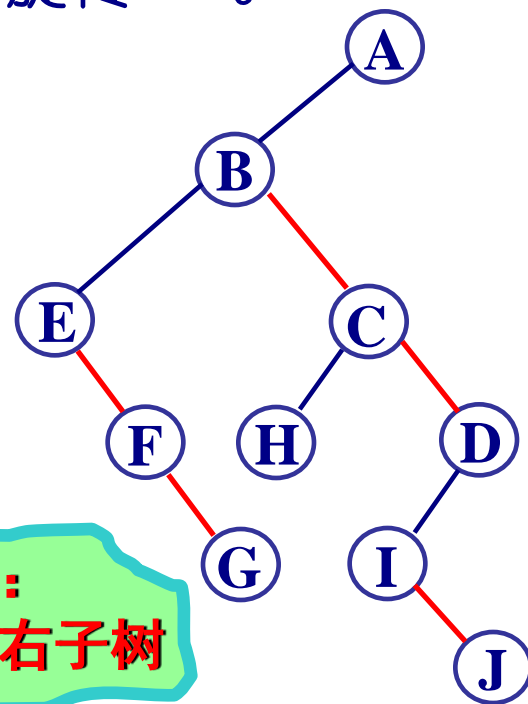
- (1) 在所有相邻的兄弟结点之间分别加一条连线；
- (2) 对于每一个分支结点，除了其最左孩子外，删除该结点与其他孩子结点之间的连线；
- (3) 以根结点为轴心，顺时针旋转 45° 。

例



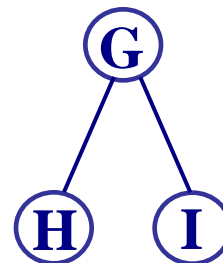
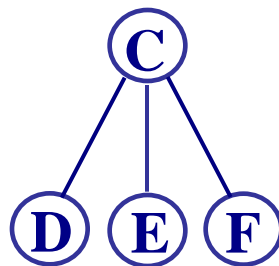
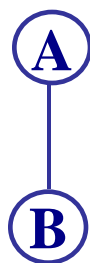
顺时针旋转
 45°

特点：
根结点无右子树



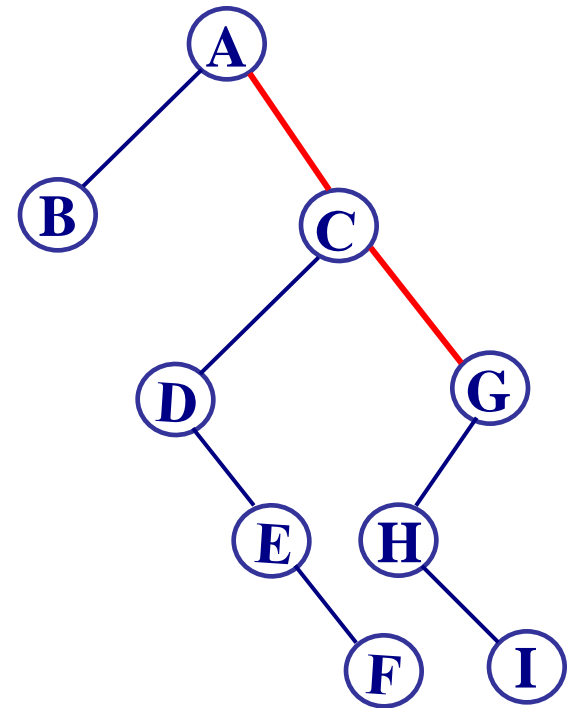
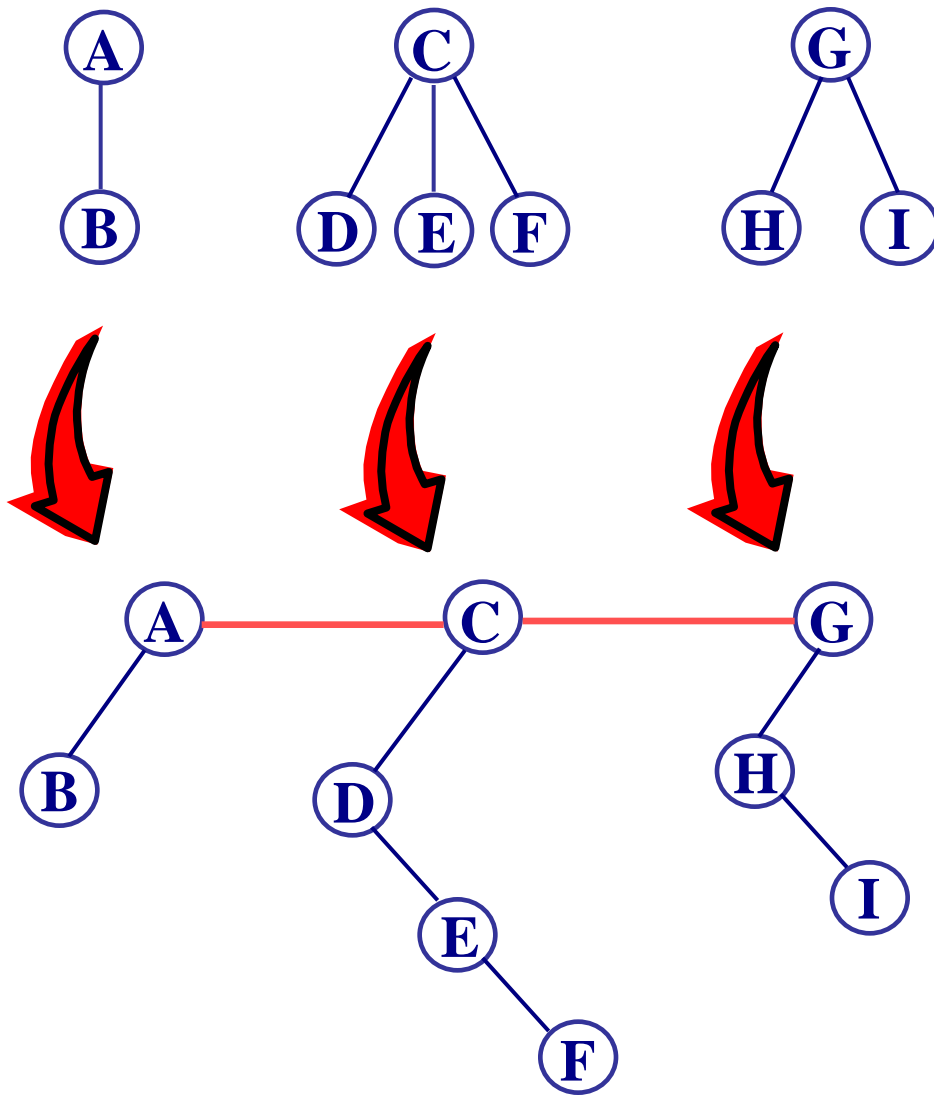
2. 树林与二叉树的转换

例



步骤

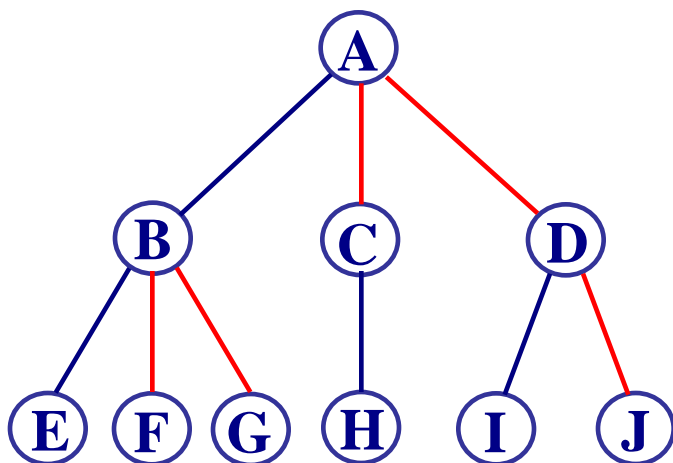
- (1) 分别将树林中每一棵树转换为一棵二叉树；
- (2) 从最后那一棵二叉树开始，依次将后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子，直到所有二叉树都这样处理。这样得到的二叉树的根结点是树林中第一棵二叉树的根结点。



转换后的二叉树

分别将每一棵树转换为二叉树

3. 二叉树还原为树



前提

由一棵树转换而来

步骤

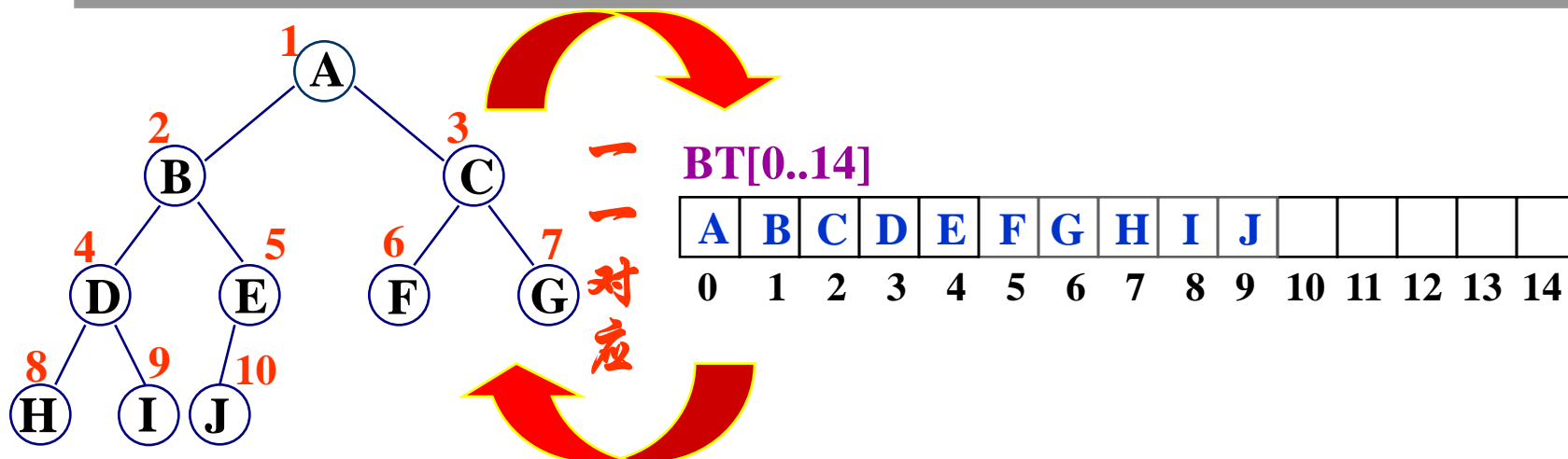
1. 若某结点是其双亲结点的左孩子，则将该结点的右孩子以及当且仅当连续地沿此右孩子的右子树方向的所有结点都分别与该结点的双亲结点用一根虚线连接；
2. 删除二叉树中所有双亲结点与其右孩子的连线；
3. 规整图形(即使各结点按照层次排列),并将虚线改成实线。

7.4 二叉树的存储结构

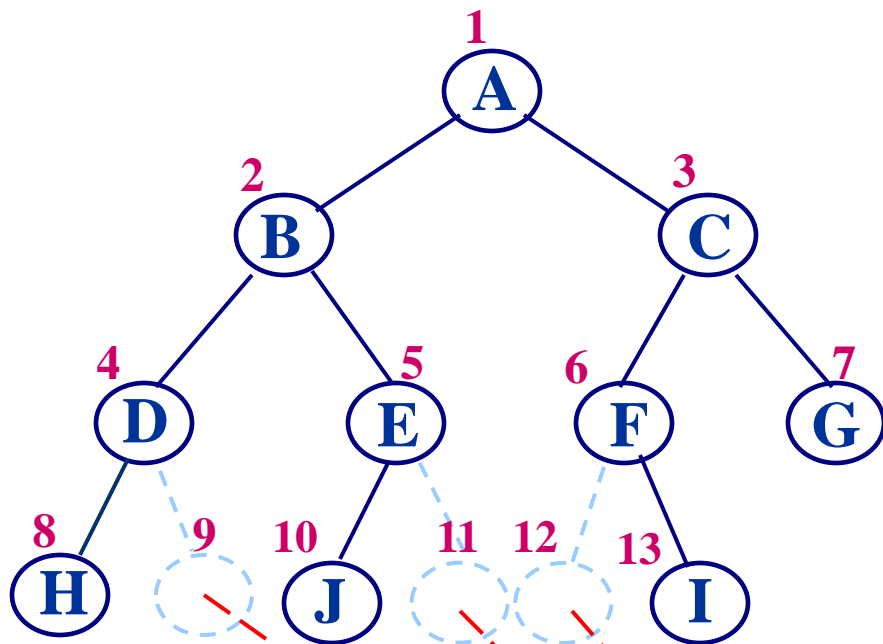
一. 二叉树的顺序存储结构

1. 完全二叉树的顺序存储结构

根据完全二叉树的性质6, 对于深度为 h 的完全二叉树, 将树中所有结点的数据信息按照编号的顺序依次存储到一维数组 $BT[0..2^h-2]$ 中, 由于编号与数组下标一一对应, 该数组就是该完全二叉树的顺序存储结构。



2. 一般二叉树的顺序存储结构



“完全二叉树”

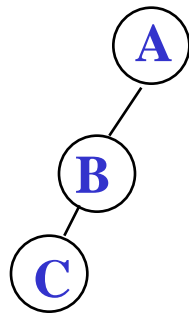
BT[0..14]

A	B	C	D	E	F	G	H		J			I		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

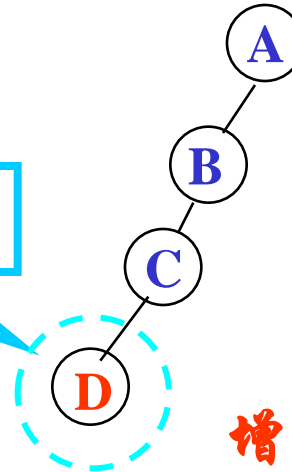
结论 (对于一般二叉树)

对于一般二叉树, 只须在二叉树中“添加”一些实际上二叉树中并不存在的“虚结点”(可以认为这些结点的数据信息为空), 使其在形式上成为一棵“完全二叉树”, 然后按照完全二叉树的顺序存储结构的构造方法将所有结点的数据信息依次存放于数组BT[0.. $2^h - 2$]中。

对于一些称为“**退化二叉树**”的二叉树，
顺序存储结构的空间开销浪费的缺点比较突出。



深度增加一层



增加1倍之多

BT[0..6]



BT[0..14]



二. 二叉树的链式存储结构

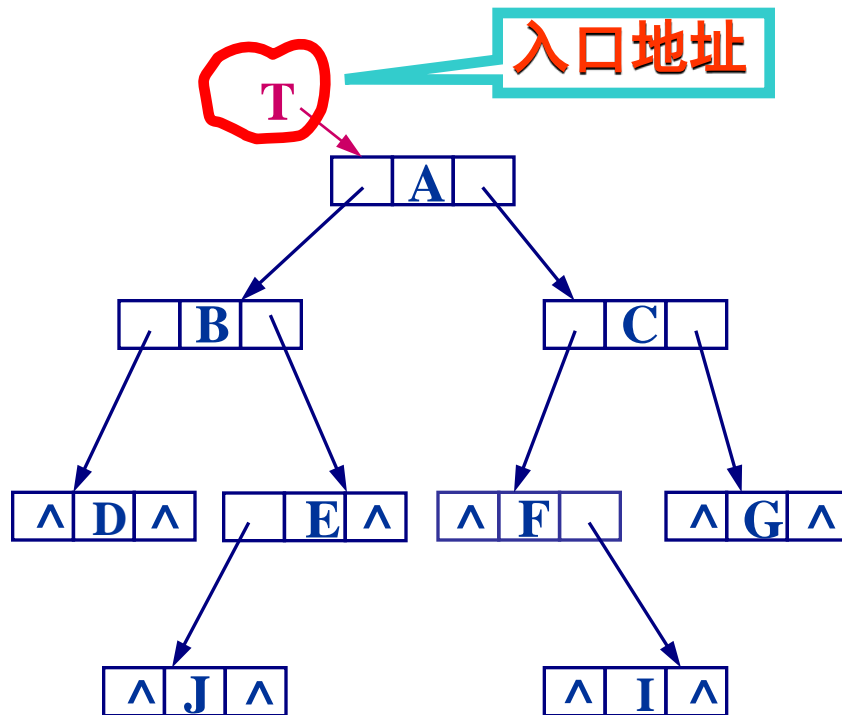
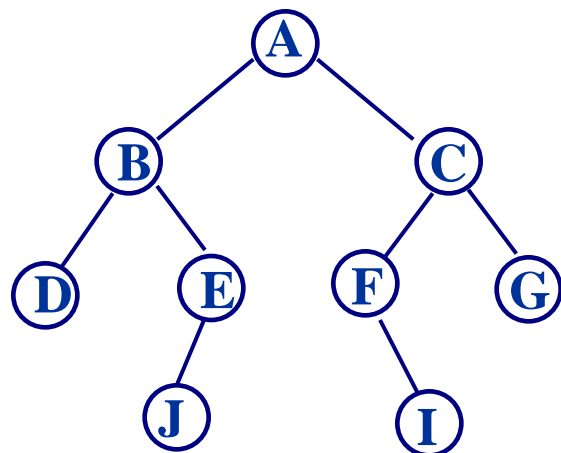
1. 二叉链表

链结点的构造为

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

其中, data 为数据域;

lchild 与 rchild 分别为指向左、右子树的指针域。

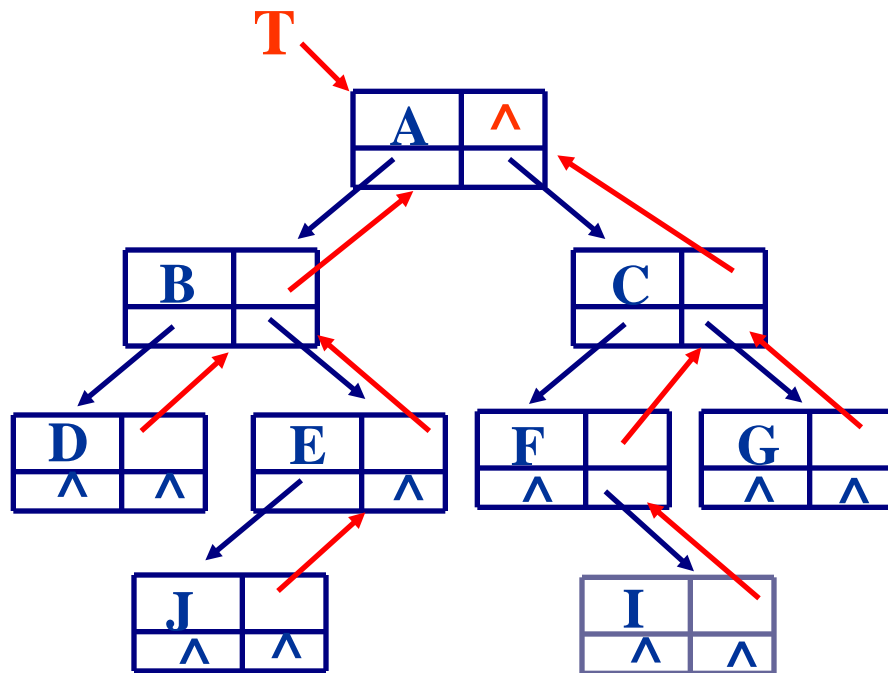
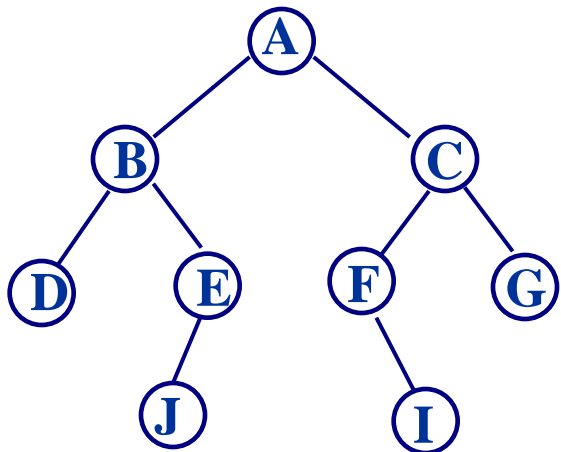


2. 三叉链表

链结点的构造为

data	parent
lchild	rchild

其中, data 为数据域, parent 为指向双亲结点的指针;
lchild 与 rchild 分别为指向左、右孩子结点的指针。



二叉链表的结点类型可定义为：

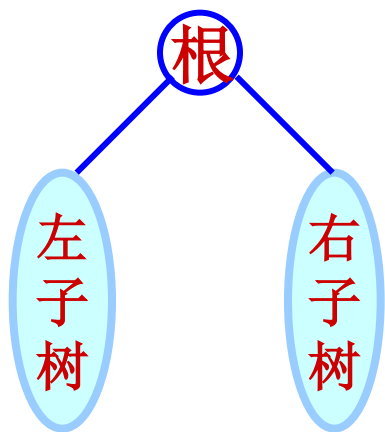
```
typedef struct node {  
    datatype data;  
    struct node *lchild, *rchild;  
} BTreeNode, *BTREE;
```



7.5 二叉树的遍历

一. 什么是二叉树的遍历

按照一定的**顺序**（原则）对二叉树中每一个结点都访问一次（仅访问一次），得到一个由该二叉树的所有结点组成的序列，这一过程称为**二叉树的遍历**。



常用的二叉树的遍历方法：

1. 前序遍历 DLR
2. 中序遍历 LDR
3. 后序遍历 LRD
4. 按层次遍历

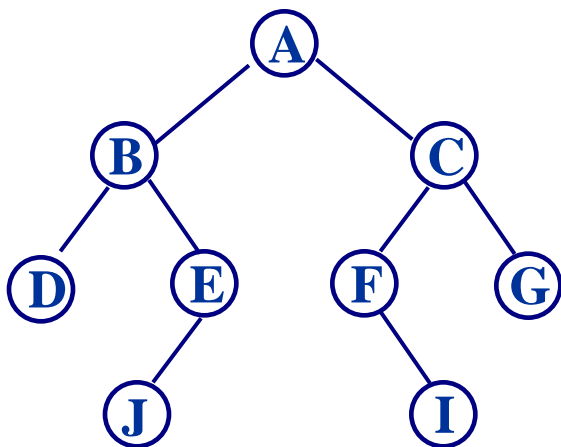
二. 前序遍历

原则:

若被遍历的二叉树非空, 则

1. 访问根结点;
2. 以前序遍历原则遍历根结点的左子树;
3. 以前序遍历原则遍历根结点的右子树.

递归



前序序列:

A B D E J C F I G

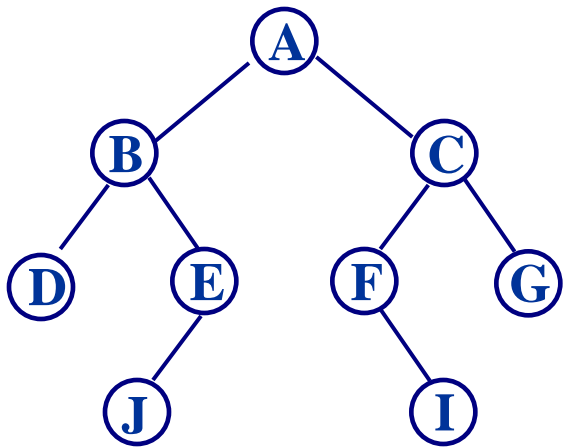
三. 中序遍历

原则:

递归

若被遍历的二叉树非空, 则

1. 以中序遍历原则遍历根结点的左子树;
2. 访问根结点;
3. 以中序遍历原则遍历根结点的右子树.



中序序列:

D B J E A F I C G

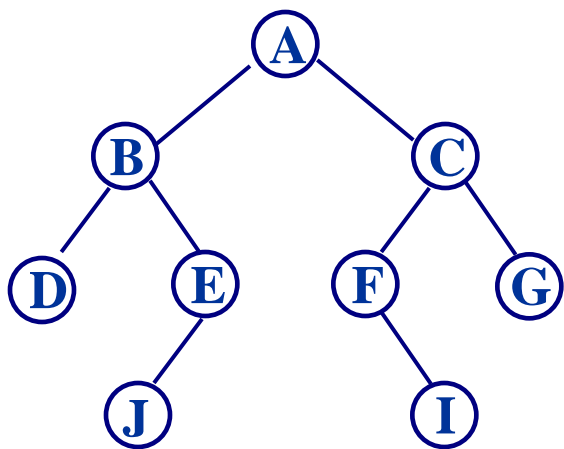
四. 后序遍历

原则：

递归

若被遍历的二叉树非空，则

1. 以后序遍历原则遍历根结点的左子树；
2. 以后序遍历原则遍历根结点的右子树。
3. 访问根结点。



后序序列：

D J E B I F G C A

前序遍历

```
void PREORDER( BTREE T )
{
    if (T!=NULL) {
        VISIT( T ); /* 访问T指结点 */
        PREORDER( T->lchild );
        PREORDER( T->rchild );
    }
}
```

后序遍历

```
void POSTORDER( BTREE T )
{
    if ( T!=NULL ) {
        POSTORDER( T->lchild );
        POSTORDER( T->rchild );
        VISIT( T ); /* 访问T指结点 */
    }
}
```


中序遍历

```
void INORDER( BTREE T )
{
    if ( T!=NULL ) {
        INORDER( T->lchild );
        VISIT( T );
        INORDER( T->rchild );
    }
}
```

排队

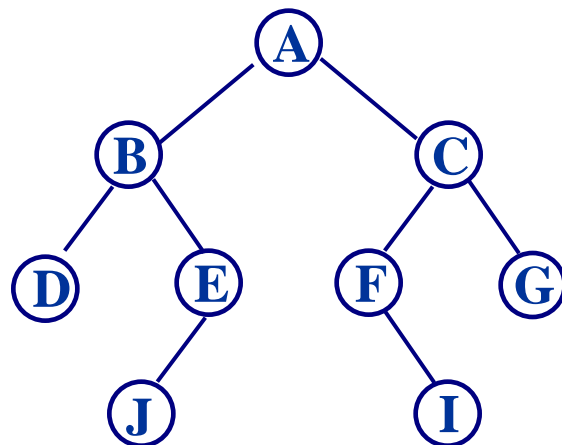
原则：

若被遍历的二叉树非空，则按照层次从上到下，每一层从左到右依次访问结点。

五. 按层次遍历

按层次遍历序列：

A, B, C, D, E, F, G, J, I



```
#define NodeNum 100
```

```
void LAYEORDER( BTREE T )
```

```
{
```

```
    BTREE QUEUE[NodeNum], p=T;
```

```
    int front, rear;
```

```
    if ( T!=NULL ) {
```

```
        QUEUE[0]=T;
```

```
        front=-1;
```

```
        rear=0;
```

```
        while(front<rear){
```

```
            p=QUEUE[++front];
```

```
            VISIT(p);
```

```
            if (p->lchild!=NULL)
```

```
                QUEUE[++rear]=p->lchild;
```

```
            if((p->rchild!=NULL)
```

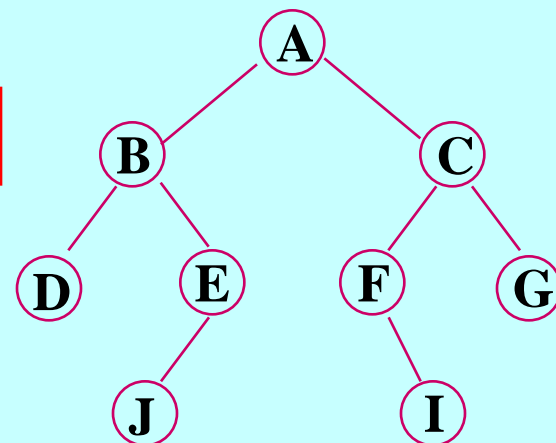
```
                QUEUE[++rear]=p->rchild;
```

```
        }
```

```
    }
```

```
}
```

队列



/*访问一个结点*/

/*若左孩子非空*/

/*若右孩子非空*/

对二叉树进行遍历的
时间复杂度均为 $O(n)$

前序遍历

若被遍历的二叉树非空，则

1. 访问根结点；
2. 以前序遍历原则遍历根结点的左子树；
3. 以前序遍历原则遍历根结点的右子树。

递归

中序遍历

若被遍历的二叉树非空，则

1. 以中序遍历原则遍历根结点的左子树；
2. 访问根结点；
3. 以中序遍历原则遍历根结点的右子树。

后序遍历

若被遍历的二叉树非空，则

1. 以后序遍历原则遍历根结点的左子树；
2. 以后序遍历原则遍历根结点的右子树。
3. 访问根结点。

前序遍历

```
void INORDER( BTREE T )
{
    if ( T!=NULL ) {
        VISIT( T );          /*访问T指结点*/
        INORDER( T->lchild );
        INORDER( T->rchild );
    }
}
```

中序遍历

```
void INORDER( BTREE T )
{
    if ( T!=NULL ) {
        INORDER( T->lchild );
        VISIT( T );          /*访问T指结点*/
        INORDER( T->rchild );
    }
}
```

后序遍历

```
void INORDER( BTREE T )
{
    if ( T!=NULL ) {
        INORDER( T->lchild );
        INORDER( T->rchild );
        VISIT( T );          /*访问T指结点*/
    }
}
```

递归算法

五. 递归问题的非递归算法的设计

1. 递归算法的优点

- (1) 问题的数学模型或算法设计方法本身就是递归的，采用递归算法来描述它们非常自然；
- (2) 算法的描述直观，结构清晰，且算法的正确性证明比非递归算法容易。

2. 递归算法的不足

- (1) 问题的执行时间与空间开销往往比非递归算法要大，当问题规模较大时尤为明显；
- (2) 对算法进行优化比较困难；
- (3) 分析跟踪算法的执行过程比较麻烦；
- (4) 描述算法的语言不具有递归功能时，算法无法描述。

结论

谨慎使用递归，因为它简洁可能会掩盖它的低效率。

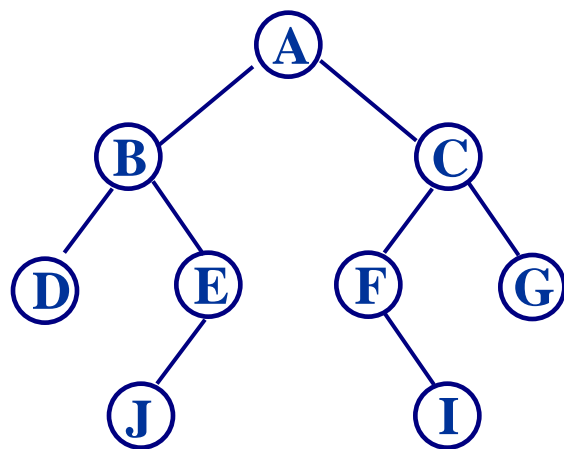
以中序遍历为例

中序遍历的递归算法

```
void INORDER( BTREE T )
{
    if ( T!=NULL ) {
        INORDER( T->lchild );
        VISIT( T );
        INORDER( T->rchild );
    }
}
```

非递归算法如何设计





对它遍历时要记住各子树中根结点的位置
采用什么样的数据结构能达到这个目的？

堆栈！

STACK[0: M-1] --- 保存遍历过程中结点的地址;
top --- 栈顶指针, 初始为-1;
p --- 为遍历过程中使用的指针变量, 初始时指向根结点。

用自然语言表达的算法

1. 若p指向的结点非空, 则将p指向的结点的地址进栈, 然后, 将p指向左子树的根;
 2. 若p指向的结点为空, 则从堆栈中退出栈顶元素送p, 访问该结点, 然后, 将p指向右子树的根;
- 重复上述过程, 直到p为空, 并且堆栈也为空。

p=p->lchild;

p=p->rchild;

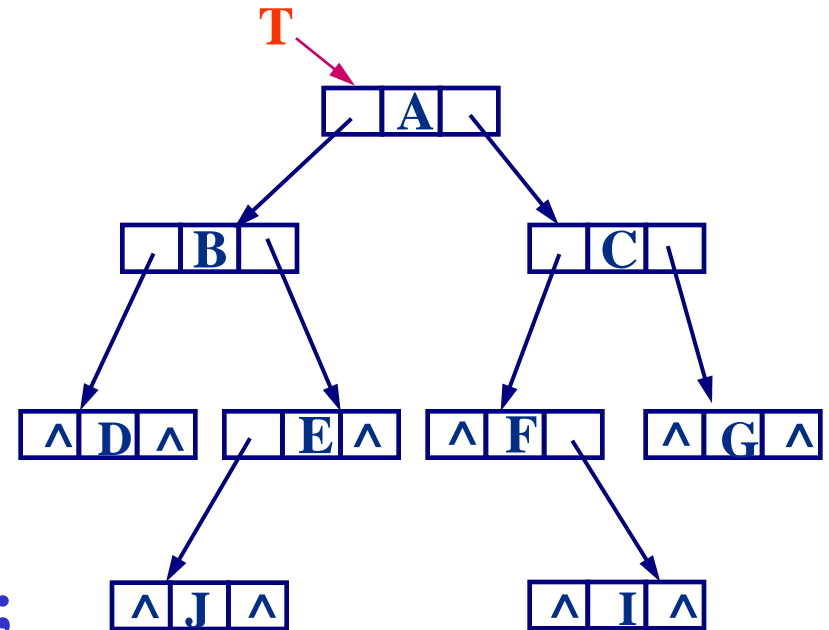
算法

```
void INORDER( BTREE T)
{
    BTREE STACK[M], p=T;
    int top= -1;
    if ( T!=NULL )
        do{
            while (p!=NULL ) {
                STACK[++top]=p;
                p=p->lchild;
            }
            p=STACK[top--];
            VISIT(p);
            p=p->rchild;
        } while( p!=NULL || top!=-1 );
}
```

前序遍历

中序序列:

D, B, J, E, A, F, I, C, G



思考1

前序遍历借助于中序遍历的算法，是否可对该前序遍历算法改进一下，栈中不再存放已经访问过的结点指针而是直接存放它的非空右子树的指针

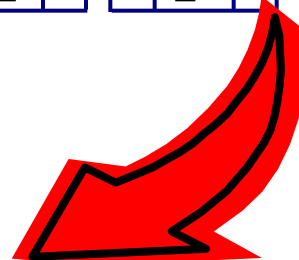
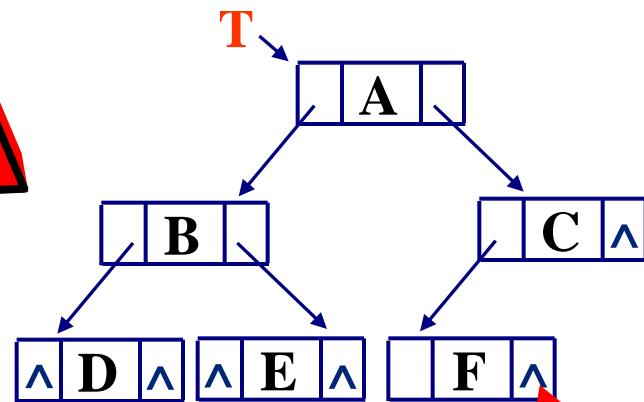
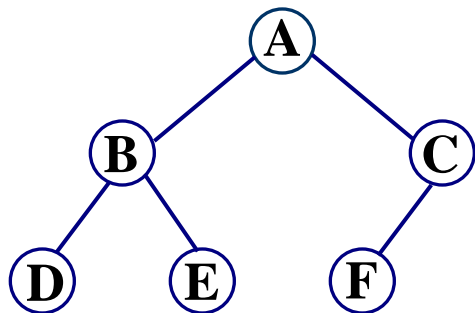


思考2

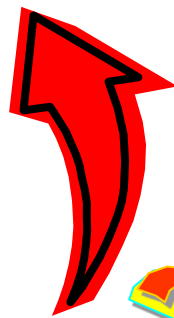
已知具有 n 个结点的完全二叉树采用顺序存储结构，结点的数据信息依次存放于一维数组 $BT[0..n-1]$ 中，写出中序遍历二叉树的非递归算法。



完全二叉树



中序序列: DBEAF C



BT[0..6]

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



算法

STACK[0..M-1] -- 保存遍历过程中结点的位置;

top -- 栈顶指针, 初始为-1;

i -- 为遍历过程中使用的位置变量, 初始时指向根结点。

$i=0, BT[0]$

1. 若 i 指向的结点非空, 则将 i 进栈, 然后, 将 i 指向左子树的根; $i=i*2+1;$
2. 若 i 指向的结点为空, 则从堆栈中退出栈顶元素送 i, 访问该结点, 然后, 将 i 指向右子树的根; $i=i*2+2;$
重复上述过程, 直到 $i=n$, 且堆栈也为空。

算法

```
#define MaxN 100
void INORDER(datatype BT[],int n)
{
    int STACK[MaxN],i,top=-1;
    i=0;
    if(n>0){
        do{
            while(i<n){
                STACK[++top]=i;          /* BT[i]的位置i进栈*/
                i=i*2+1;                  /* 找到i的左孩子的位置 */
            }
            i=STACK[top--];               /* 退栈*/
            VISIT(BT[i]);                 /* 访问结点BT[i] */
            i=i*2+2;                      /* 找到i的右孩子的位置*/
        }while(i<n||top>=0);
    }
}
```

一般二叉树 

例1

前序遍历序列:

A, B, D, K, J, G, C, F, I, E, H

中序遍历序列:

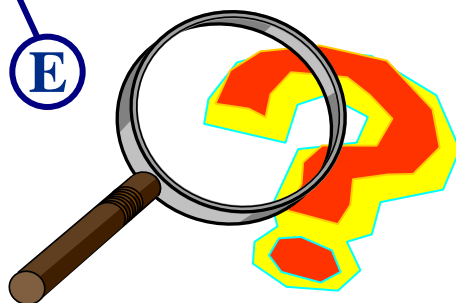
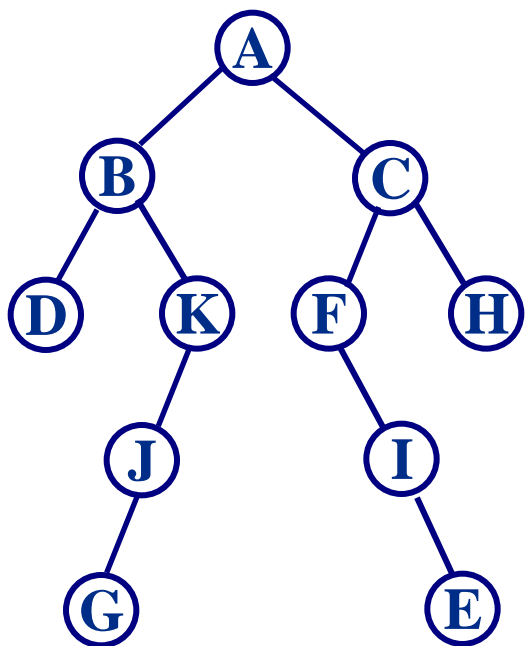
D, B, G, J, K, A, F, I, E, C, H

后序遍历序列:

D, G, J, K, B, E, I, F, H, C, A

按层次遍历序列:

A, B, C, D, K, F, H, J, I, G, E



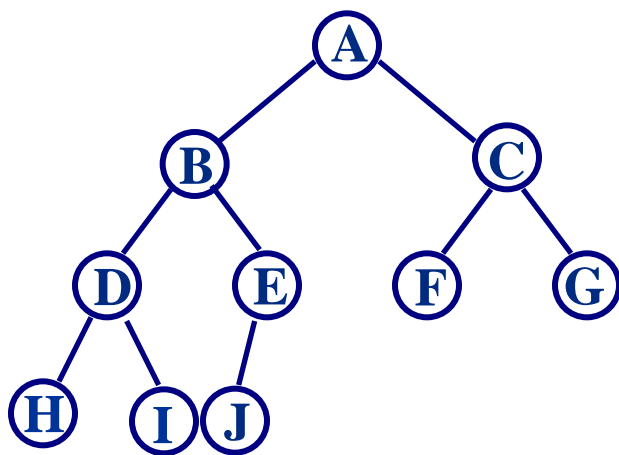
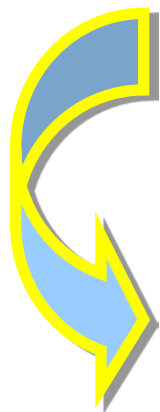
例2

若某完全二叉树采用顺序存储结构，结点存放的次序为A,B,C,D,E,F,G,H,I,J，请给出该二叉树的后序序列。

BT[0..14]

A	B	C	D	E	F	G	H	I	J					
---	---	---	---	---	---	---	---	---	---	--	--	--	--	--

后序序列



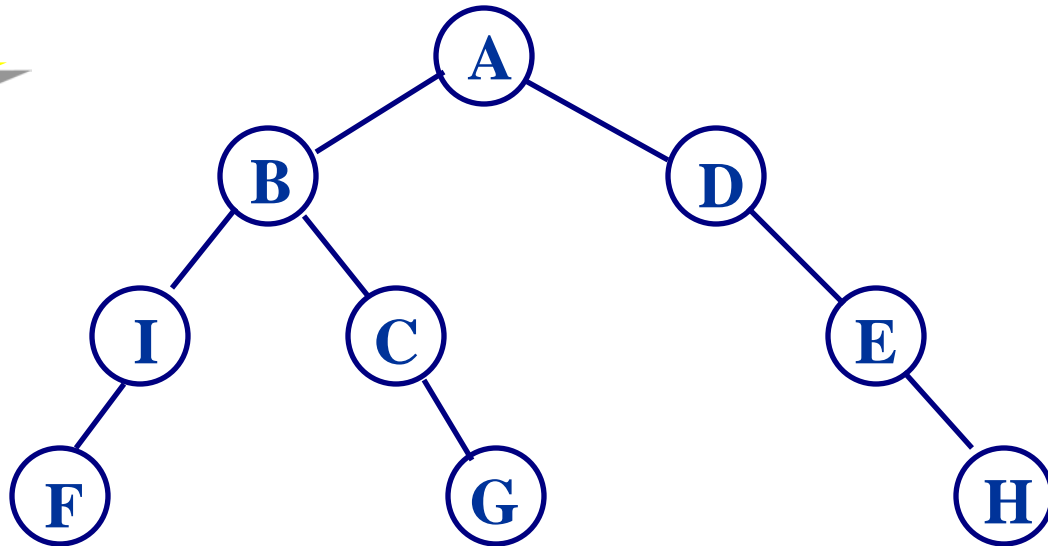
H I D J E B F G C A

一般二叉树

BT[0..14]

A	B	D	I	C		E	F			G				H
---	---	---	---	---	--	---	---	--	--	---	--	--	--	---

例3



前序序列: A B I F C G D E H

中序序列: F I B C G A D E H

后序序列: F I G C B H E D A

思考

能否利用遍历序列恢复二叉树



利用前序序列和中序序列恢复二叉树



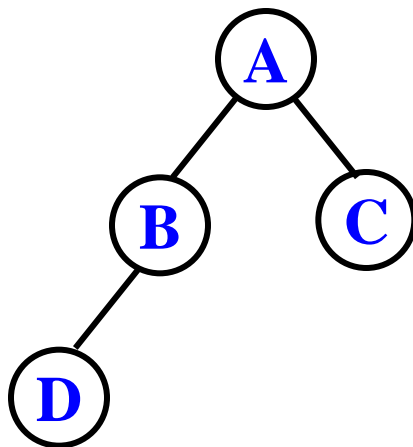
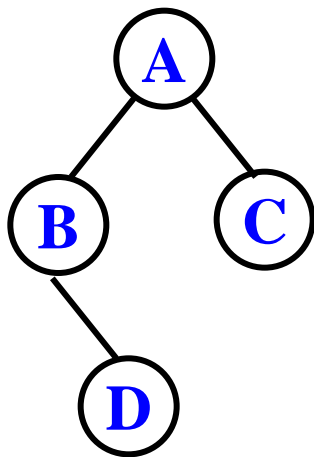
利用中序序列和后序序列恢复二叉树



利用前序序列和后序序列恢复二叉树



例



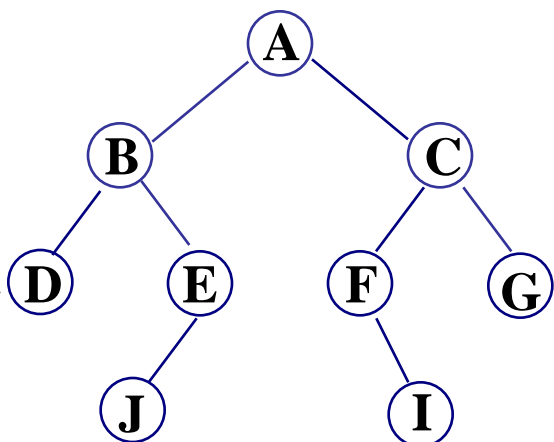
前序序列: A,B,D,C

后序序列: D,B,C,A

若已知二叉树的
前序序列与中序序列，
如何求二叉树的后序
序列



六. 由遍历序列恢复二叉树



前序序列:

A, B, D, E, J, C, F, I, G

中序序列:

D, B, J, E, A, F, I, C, G

后序序列:

?!

前序序列:

A, B, D, E, J, C, F, I, G

中序序列:

D, B, J, E, A, F, I, C, G

后序序列:

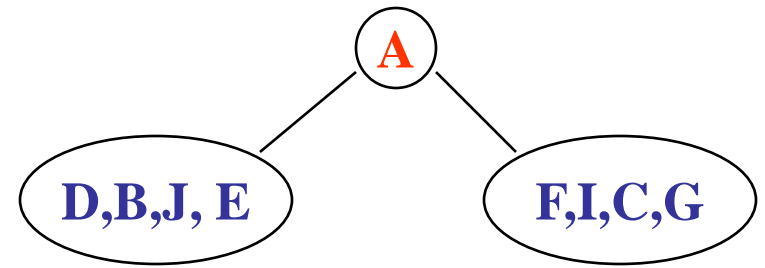
D, J, E, B, I, F, G, C, A

前序序列:

A, **B**, **D**, **E**, **J**, **C**, **F**, **I**, **G**

中序序列:

D, **B**, **J**, **E**, **A**, **F**, **I**, **C**, **G**

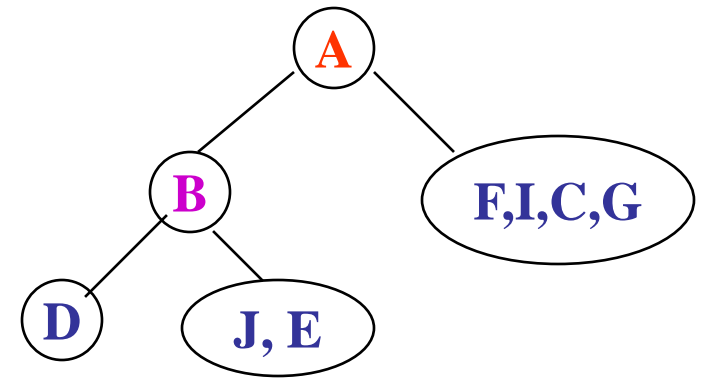


前序序列:

A, **B**, **D**, **E**, **J**, C, F, I, G

中序序列:

D, **B**, **J**, **E**, A, F, I, C, G

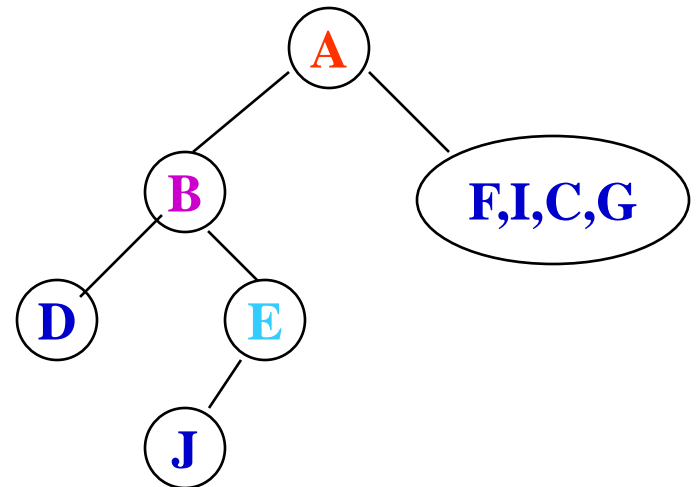


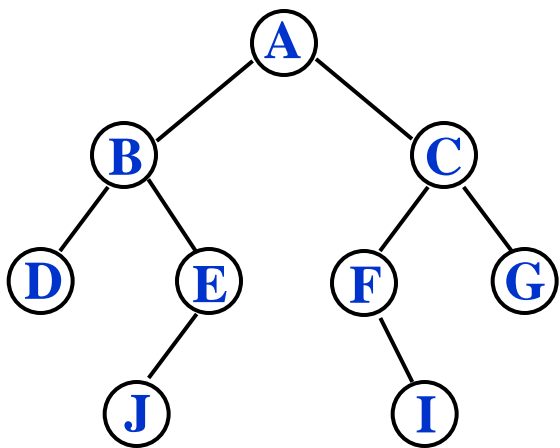
前序序列:

A, B, **D**, **E**, **J**, C, F, I, G

中序序列:

D, B, **J**, **E**, A, F, I, C, G





后序序列:

D, J, E, B, I, F, G, C, A

规律

已知前序和中序，恢复二叉树

在前序序列中寻找根；
到中序序列中分左右。

已知中序和后序，恢复二叉树

在后序序列中寻找根；
到中序序列中分左右。

例4

中序序列：

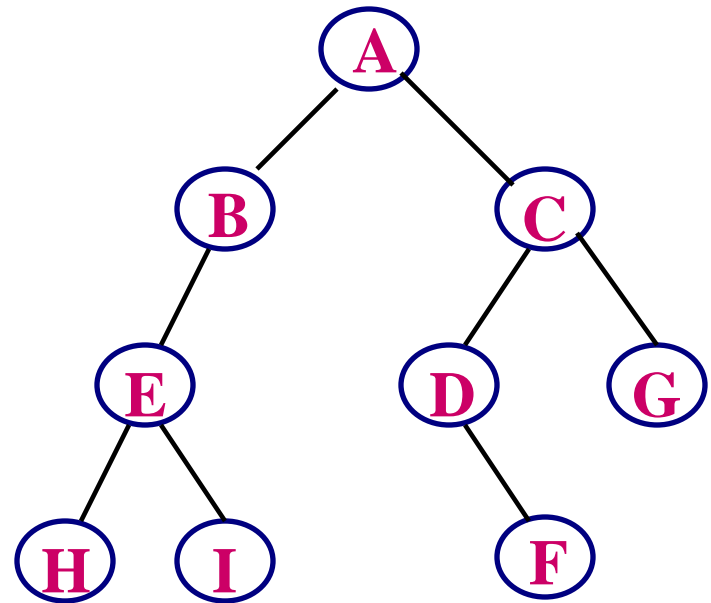
H E I B A D F C G

后序序列：

H I E B F D G C A

前序序列：

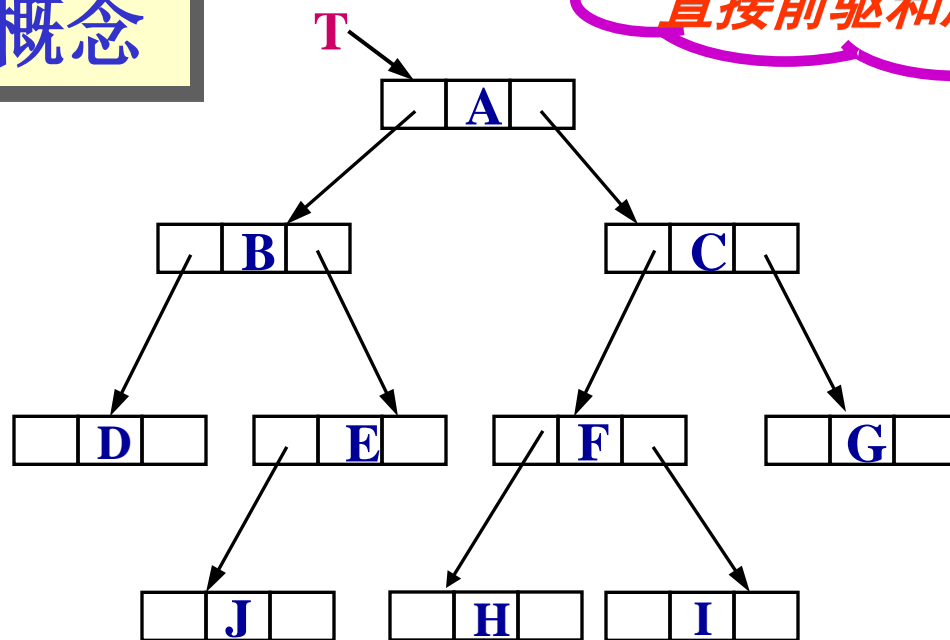
A B E H I C D F G



7.6 线索二叉树

一、基本概念

如何在存储结构中
方便地知道结点的
直接前驱和后继



中序序列:

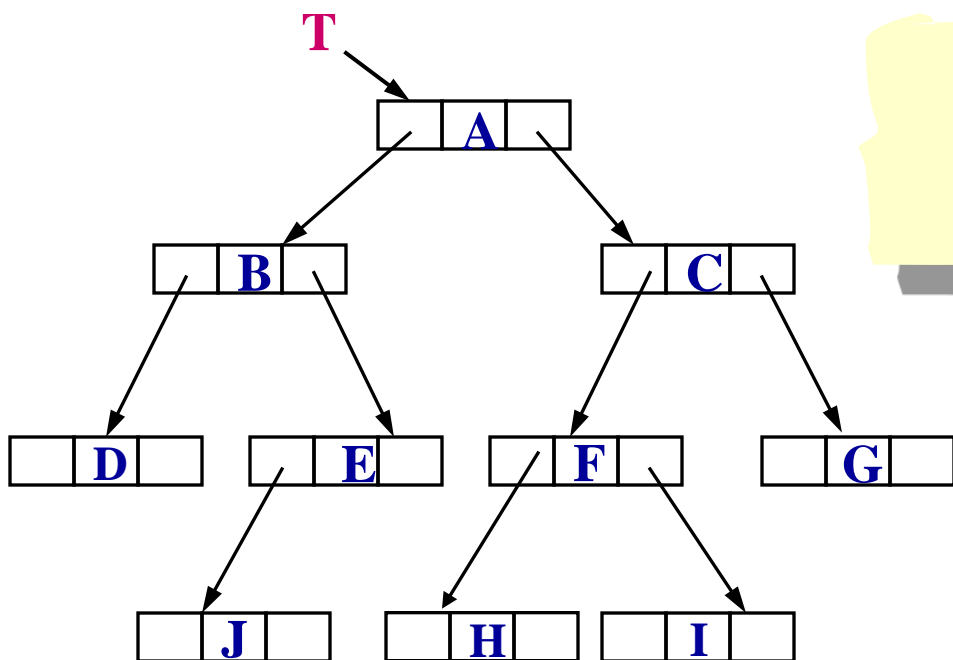
D, B, J, E, A, H, F, I, C, G

二、什么是线索二叉树

利用二叉链表中空的指针域指出结点在遍历序列中的直接前驱和直接后继;这些指向前驱和后继的指针称为**线索**,加了线索的二叉树称为**线索二叉树**.

三、线索二叉树的构造

利用链结点的**空的左指针域**存放该结点的直接前驱的地址, **空的右指针域**存放该结点直接后继的地址;而非空的指针域仍然存放结点的左孩子或右孩子的地址.



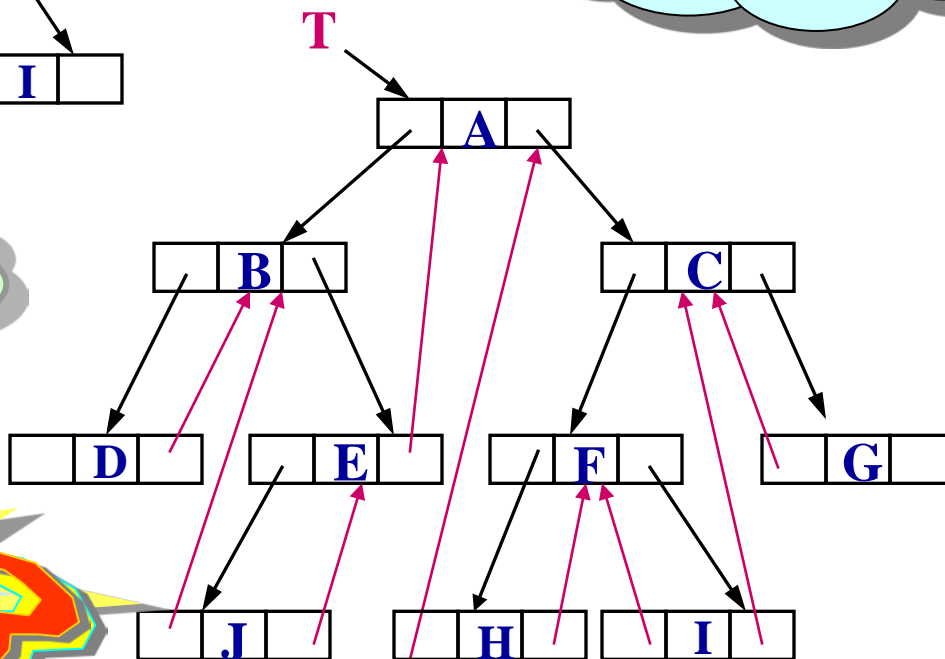
中序序列:

D, B, J, E, A, H, F, I, C, G

以中序线索
二叉树为例

中序线索
二叉树

如何区分
指针和线索



指针与线索的区分方法之一：

lbit	lchild	data	rchild	rbit
-------------	---------------	-------------	---------------	-------------

$\text{lbit}(p) = \begin{cases} 0 & \text{表示 } \text{lchild}(p) \text{ 为指向直接前驱的线索} \\ 1 & \text{表示 } \text{lchild}(p) \text{ 为指向左孩子的指针} \end{cases}$

$\text{rbit}(p) = \begin{cases} 0 & \text{表示 } \text{rchild}(p) \text{ 为指向直接后继的线索} \\ 1 & \text{表示 } \text{rchild}(p) \text{ 为指向右孩子的指针} \end{cases}$

指针与线索的区分方法之二：

不改变链结点的构造, 而是在作为线索的地址前加一个负号, 即“负地址”表示线索, “正地址”表示指针.

本课程采用方法一

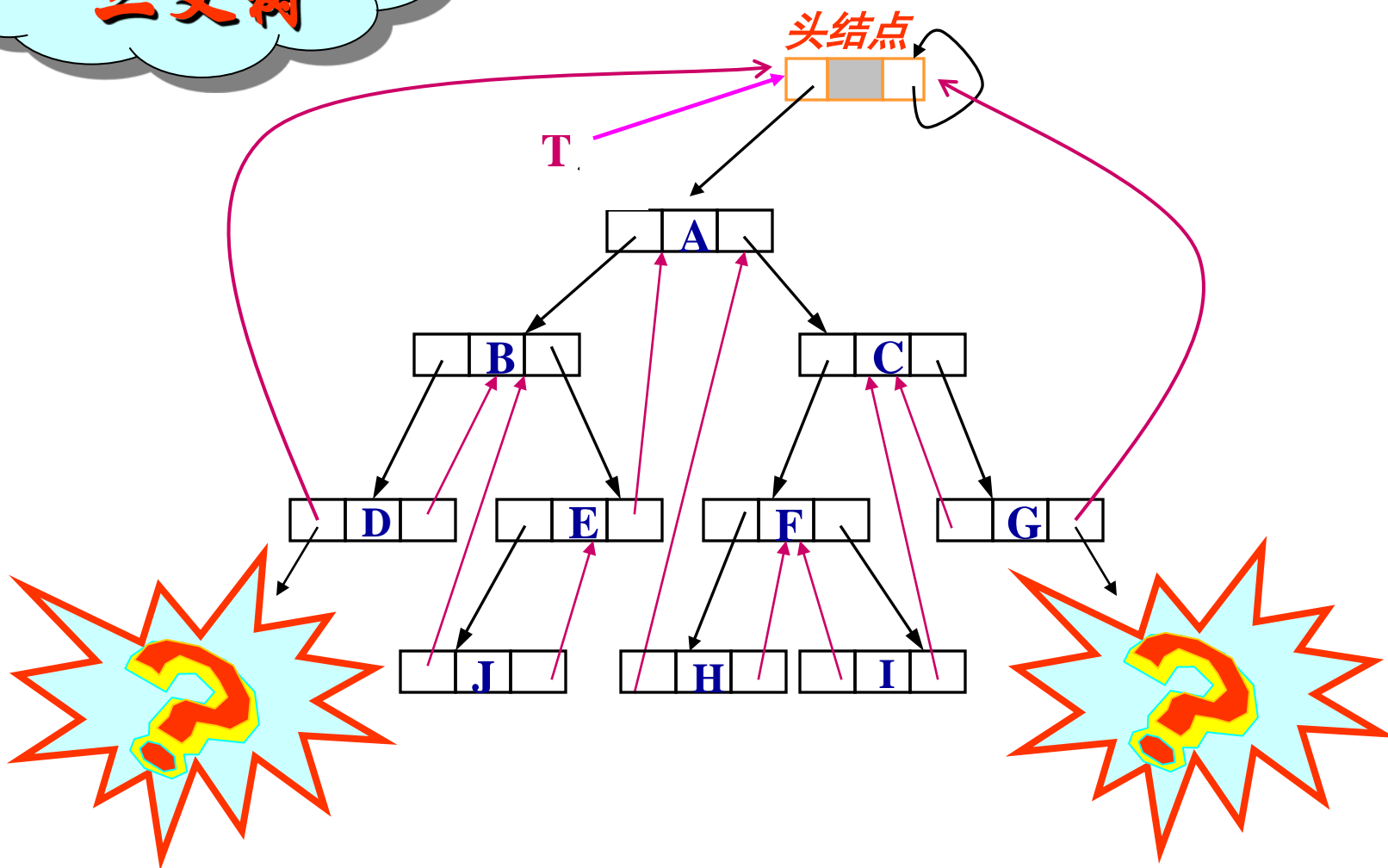
lbit	lchild	data	rchild	rbit
-------------	---------------	-------------	---------------	-------------

线索二叉树链结点类型可定义为：

```
typedef struct node {  
    datatype data;  
    struct node *lchild, *rchild;  
    int libt, rbit;  
} TBTNode, *TBTREE;
```

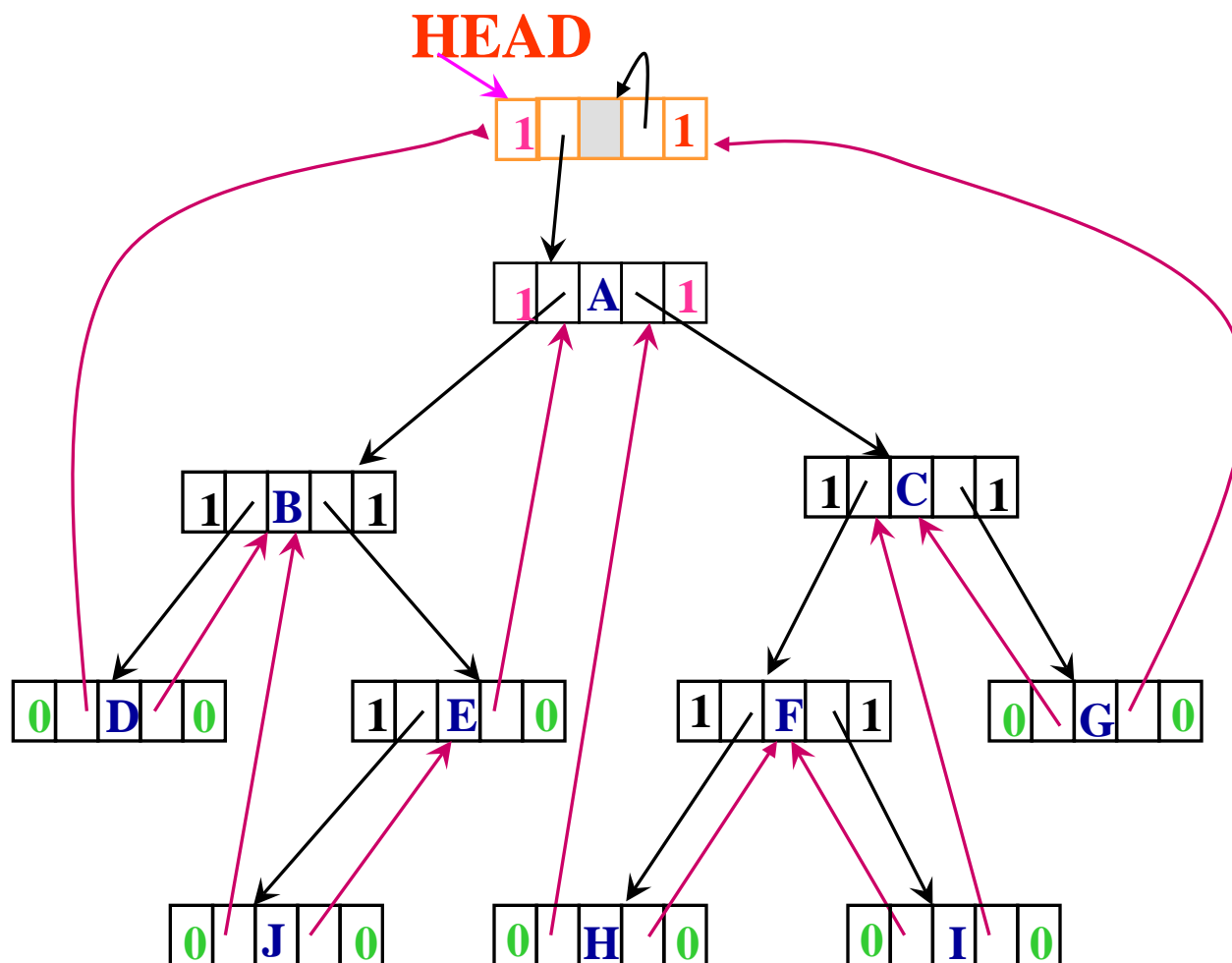
libt	lchild	data	rchild	rbit
------	--------	------	--------	------

中序线索二叉树



中序序列: **D, B, J, E, A, H, F, I, C, G**

一棵完整的中序线索二叉树



四、线索二叉树的利用

在中序线索二叉树中确定
地址为x 的结点的直接后继。

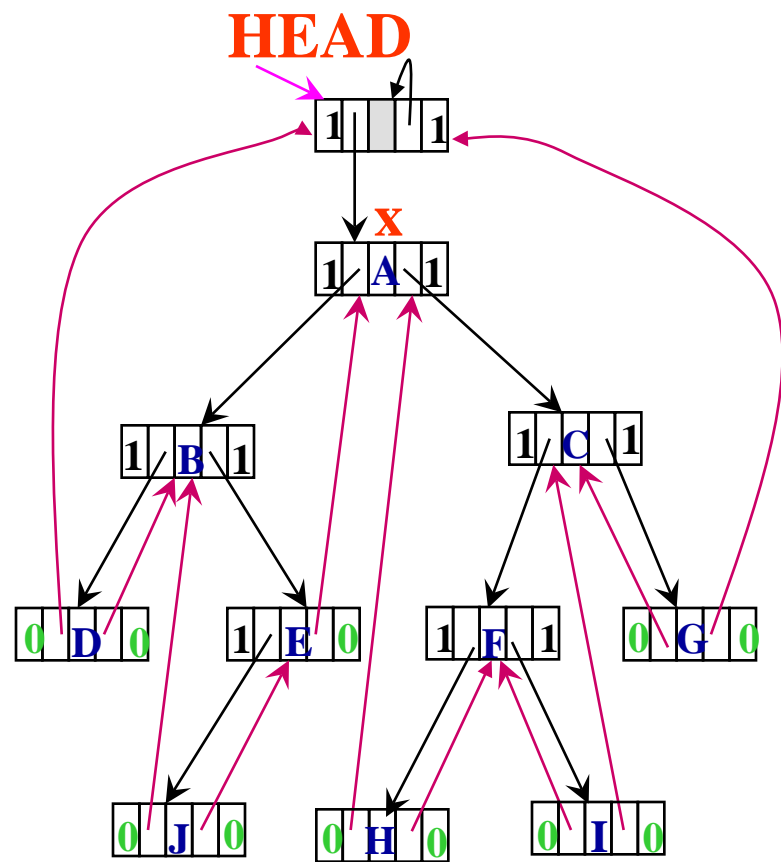
规律

- (1) 当 $x \rightarrow rbit = 0$ 时, $x \rightarrow rchild$ 指出的结点就是x的直接后继结点。
- (2) 当 $x \rightarrow rbit = 1$ 时, 沿着x的右子树的根的左子树方向查找, 直到某结点的 lchild 域为线索时, 此结点就是x结点直接后继结点。

该结点的lbit=0

中序序列:

D, B, J, E, A, H, F, I, C, G



BTREE INSUCC(BTREE x)

```

{
    BTREE s;
    s=x->rchild;
    if ( x->rbit==1)
        while (s->lbit==1)
            s=s->lchild(s);
    return(s);
}
    
```

确定X的直接后继

中序序列:

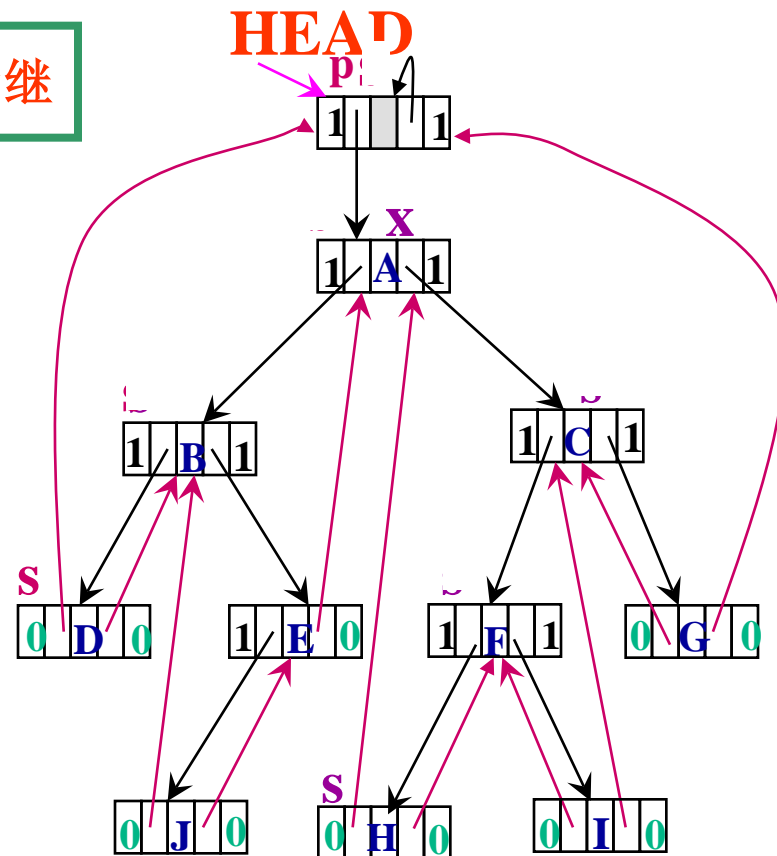
D, B, J, E, A, H, F, I, C, G

void TORDER(BTREE HEAD)

```

{
    BTREE p=HEAD;
    while(1) {
        p=INSUCC(p);
        if (p==HEAD)
            break;
        VISIT(p);
    }
}
    
```

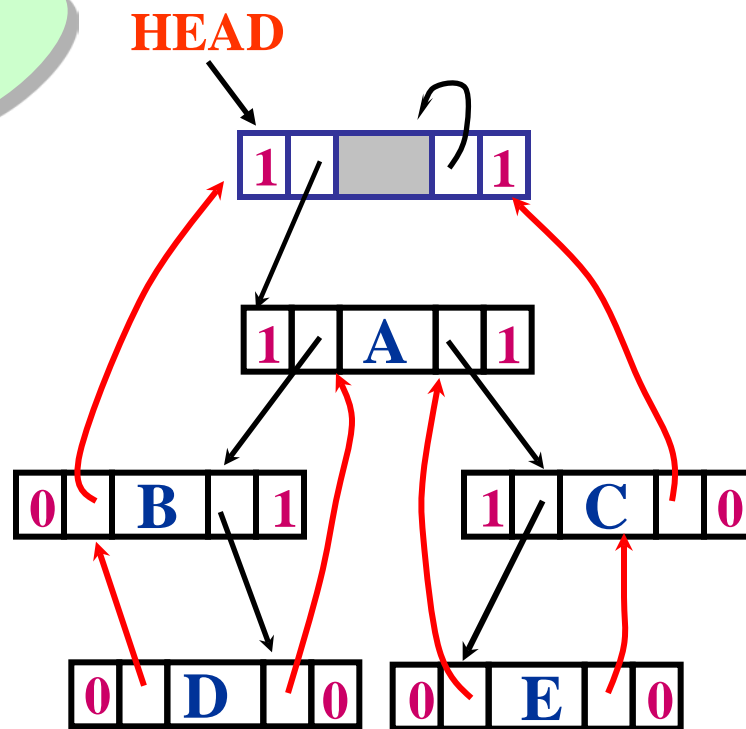
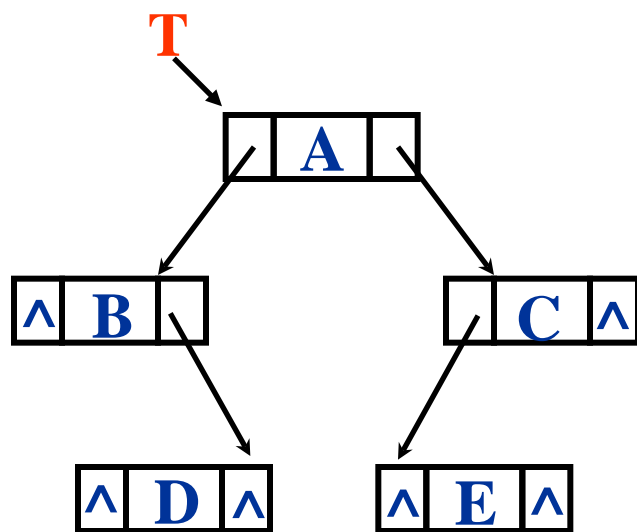
中序遍历



五、线索二叉树的建立

以中序线索
二叉树为例

中序序列: B,D,A,E,C

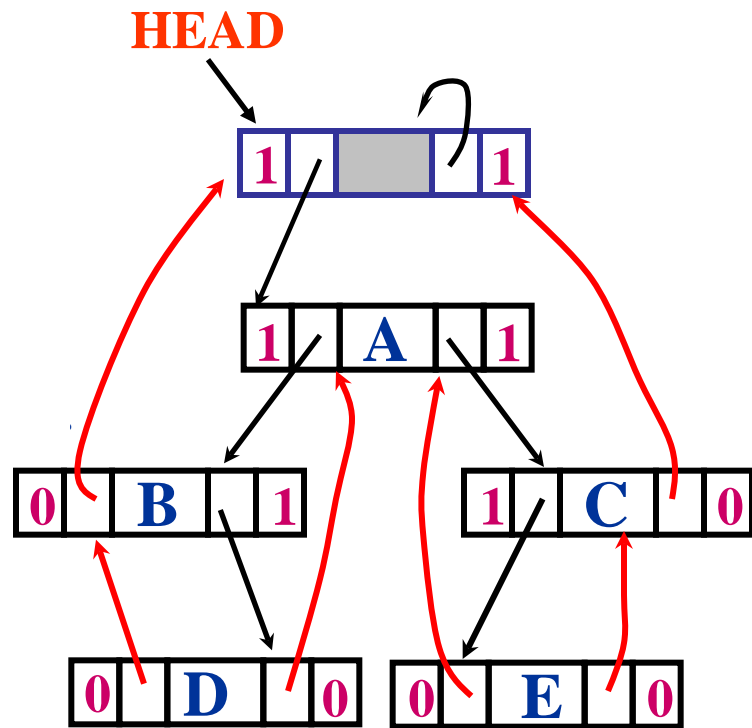


中序线索
二叉树



在遍历过程中 中建立线索

prior: 指向前一次访问结点
p: 指向当前访问结点

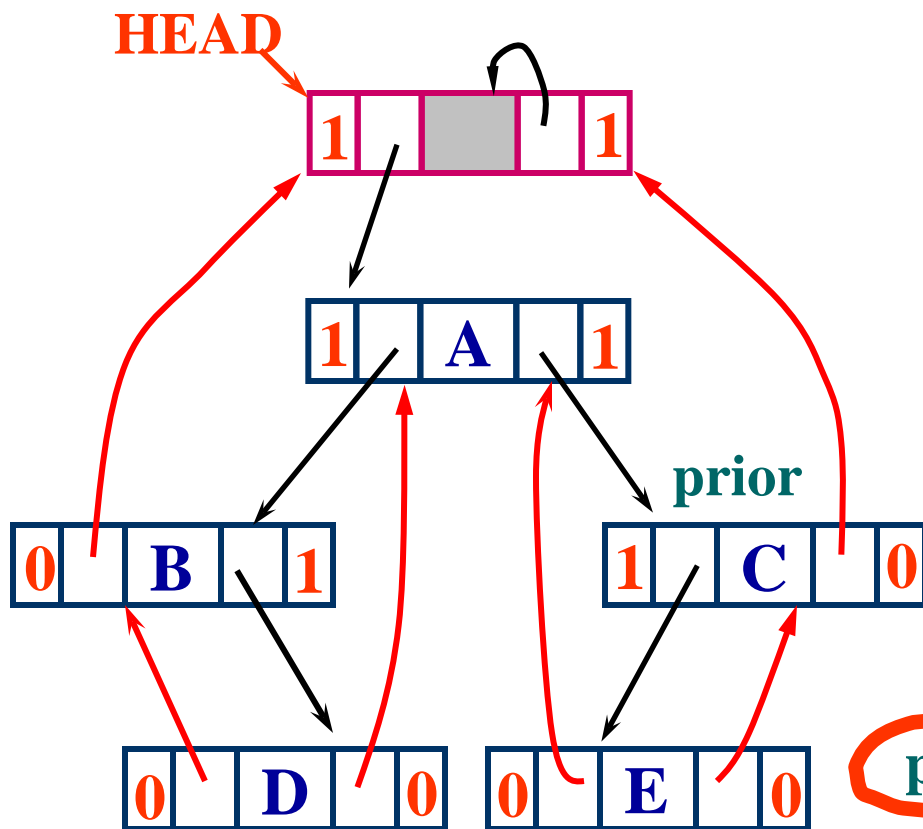


中序序列: **B,D,A,E,C**

建立线索的规律

- 若当前访问的结点p的左指针域为空, 则它指向prior指的结点;
- 若prior所指结点的右指针域为空, 则它指向当前访问的结点p。

规律



● 若当前访问的结点p的左指针域为空, 则左指针域指向prior指的结点,同时置lbit为0, 否则, 置lbit为1;

● 若prior所指结点的右指针域为空, 则右指针域指向当前访问的结点p, 同时置rbit为0, 否则, 置rbit为1。

p=NULL

遍历结束

中序序列: B,D,A,E,C

prior: 指向前一次访问结点
p: 指向当前访问结点

#define NodeNum 100

/* 定义二叉树中结点最大数目 */

TBTREE INTTHREAD(TBTREE T)

{ TBTREE HEAD,p=T,prior,STACK[NodeNum];

int top=-1;

HEAD=(TBTREE)malloc(sizeof(TBNode)); /* 申请线索二叉树的头结点空间 */

HEAD->lchild=T; HEAD->rchild=HEAD; HEAD->lbit=1; HEAD->rbit=1;

prior=HEAD; /* 假设中序序列的第1个结点的“前驱”为头结点 */

do{

while(p!=NULL){

STACK[++top]=p;

p=p->lchild;

/* p指结点的地址进栈 */

/* p移到左孩子结点 */

}

p=STACK[top--];

/* 退栈 */

if(p->lchild==NULL){

/* 若当前访问结点的左孩子为空 */

p->lchild=prior;

/* 当前访问结点的左指针域指向前一次访问结点 */

p->lbit=0;

/* 当前访问结点的左标志域置0(表示地址为线索) */

}else

p->lbit=1;

/* 当前访问结点的左标志域置1(表示地址为指针) */

if(prior->rchild==NULL){

/* 若前一次访问的结点的右孩子为空 */

prior->rchild=p;

/* 前一次访问结点的右指针域指向当前访问结点 */

prior->rbit=0;

/* 前一次访问结点的右标志域置0(表示地址为线索) */

}else

prior->rbit=1;

/* 前一次访问结点的右标志域置1(表示地址为指针) */

prior=p;

/* 记录当前访问的结点的地址 */

p=p->rchild;

/* p移到右孩子结点 */

}while(p!=NULL || top!=-1);

prior->rchild=HEAD;

/* 设中序序列的最后结点的后继为头结点 */

prior->rbit=0;

/* prior指结点的右标志域置0(表示地址为线索) */

return HEAD;

/* 返回线索二叉树的头结点指针 */

}

算法

建立线索

访问一个结点

7.7 二叉排序树

一. 二叉排序树的定义

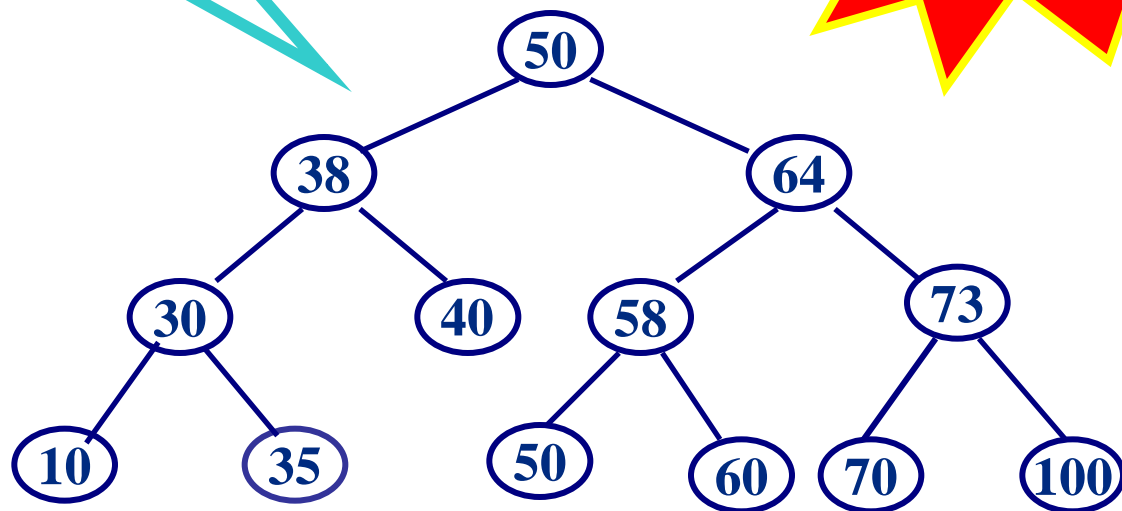
二叉排序树或者为空二叉树, 或者为具有以下性质的二叉树:

- 若根结点的左子树不空, 则左子树上所有结点的值都小于根结点的值;
 - 若根结点的右子树不空, 则右子树上所有结点的值都大于或者等于根结点的值;
- 每一棵子树分别也是**二叉排序树**.

递归定义

一棵二叉排序树

中序遍历



中序序列

10, 30, 35, 38, 40, 50, 50, 58, 60, 64, 70, 73, 100

二. 二叉排序树的建立 (逐点插入法)

设 $K=(k_1, k_2, k_3, \dots, k_n)$ 为具有 n 个数据元素的序列。从序列的第一个元素开始, 依次取序列中的元素, 每取一个元素 k_i , 按照下述原则将 k_i 插入到二叉树中:

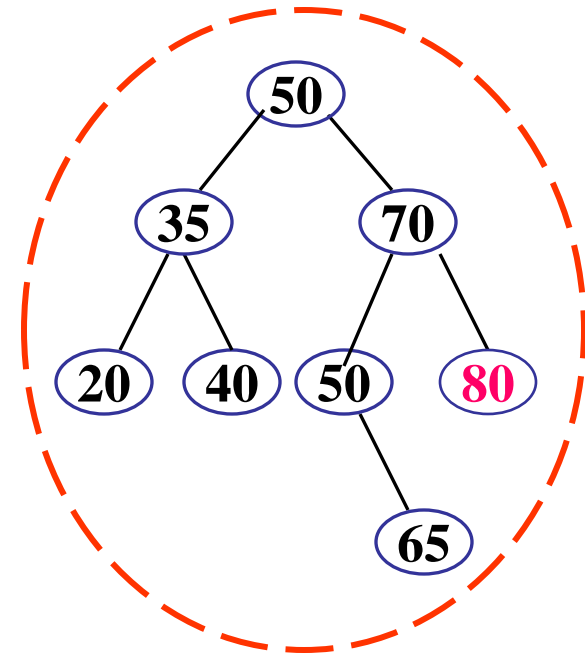
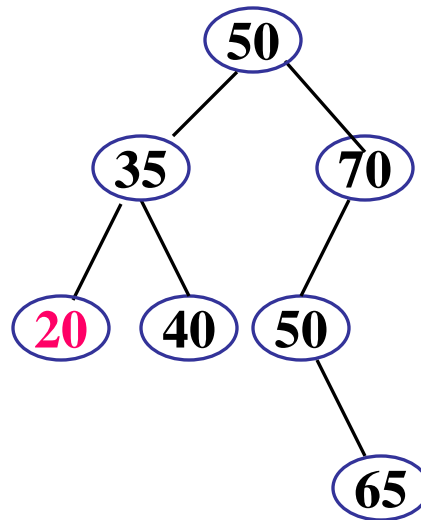
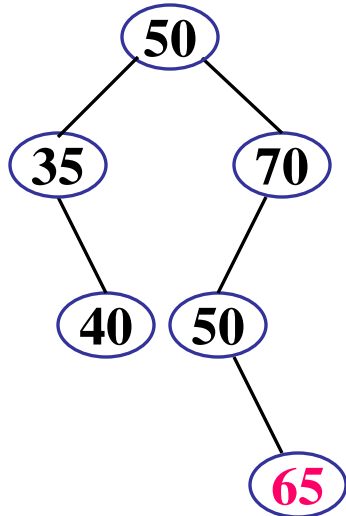
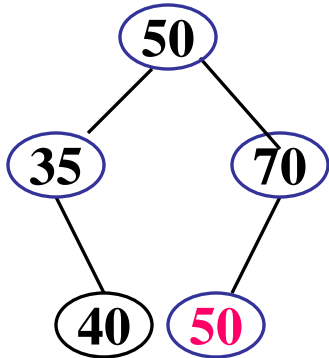
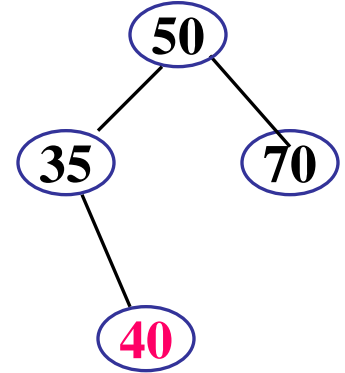
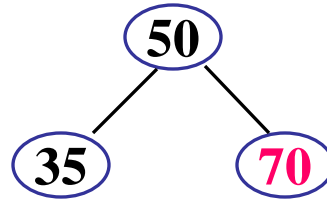
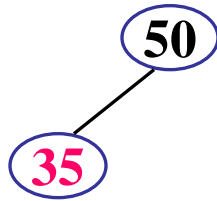
1. 若**二叉树**为空, 则 k_i 作为该二叉树的根结点;
2. 若**二叉树**非空, 则将 k_i 与该二叉树的根结点的值进行比较, 若 k_i 小于根结点的值, 则将 k_i 插入到根结点的左子树中; 否则, 将 k_i 插入到根结点的右子树中。

将 k_i 插入到左子树或者右子树中仍然遵循上述原则.

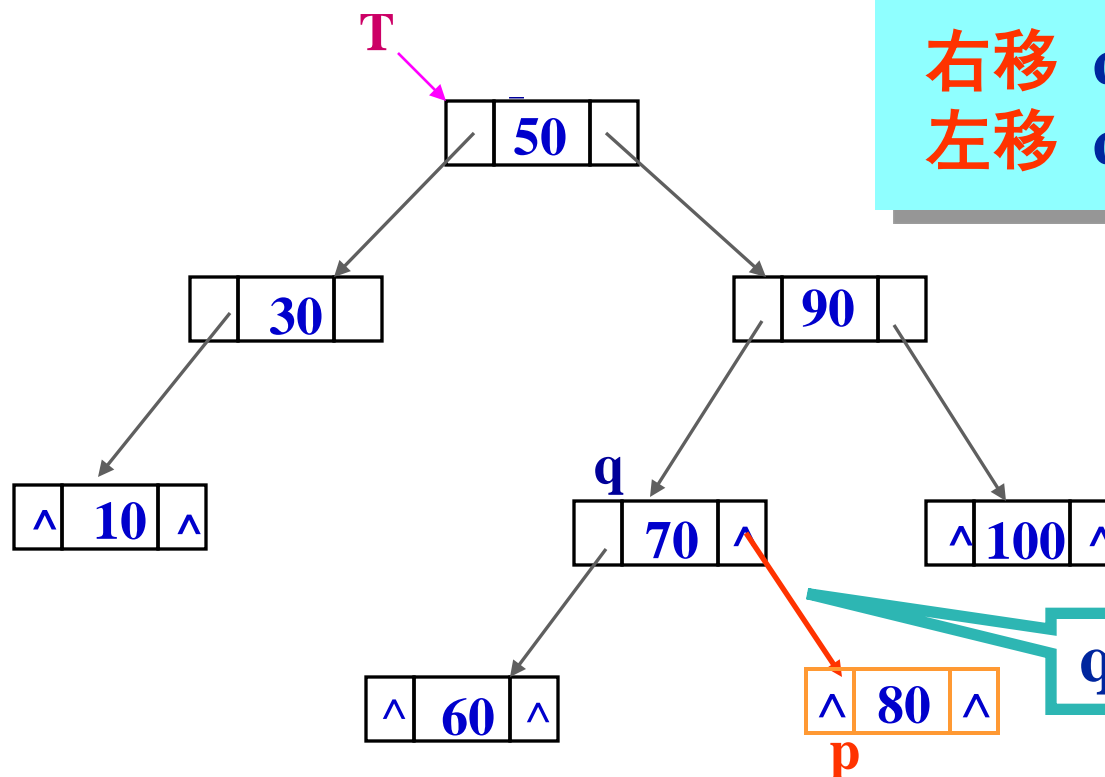
例

$K = (\underline{50}, \underline{35}, \underline{70}, \underline{40}, \underline{50}, \underline{65}, \underline{20}, \underline{80})$

空



二叉排序树如何插入一个元素



右移 $q=q \rightarrow rchild;$
左移 $q=q \rightarrow lchild;$

插入

item=80

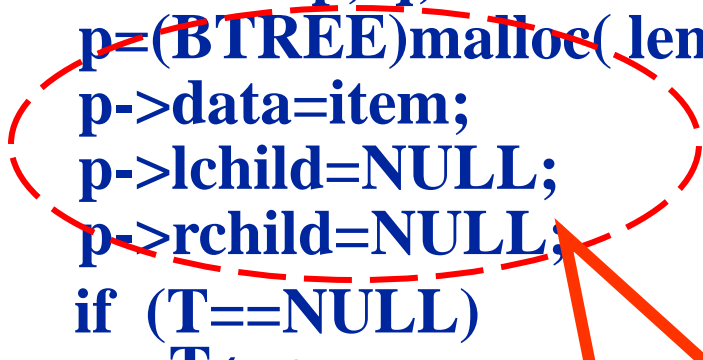
$q \rightarrow rchild = p;$

二叉树采用二叉链表存储结构。

非递归算法

功能: 将一个数据元素item
插入到根为T的二叉
排序树中。

```
#define len sizeof(BTNode)
void INSERTBST( BTREE &T,
                typedata item )
{
    BTREE p, q;
    p=(BTREE)malloc( len );
    p->data=item;
    p->lchild=NULL;
    p->rchild=NULL;
    if (T==NULL)
        T←p
    else {
```



建立一个新结点

插入一个结点

```
q=T;
while(1)
    if (item<q->data)
        if (q->lchild==NULL){
            q->lchild=p;
            break;
        }
        else
            q=q->lchild;
    else
        if (q->rchild==NULL){
            q->rchild=p;
            break;
        }
        else
            q=q->rchild;
}
```

建立二叉排序树的算法

```
BTREE SORTTREE( datatype K[ ], int n )  
{  
    BTREE T=NULL;  
    int i;  
    for ( i=0; i<n; i++ )  
        INSERTBST( T, K[i] );  
    return T;  
}
```

调用插入算法

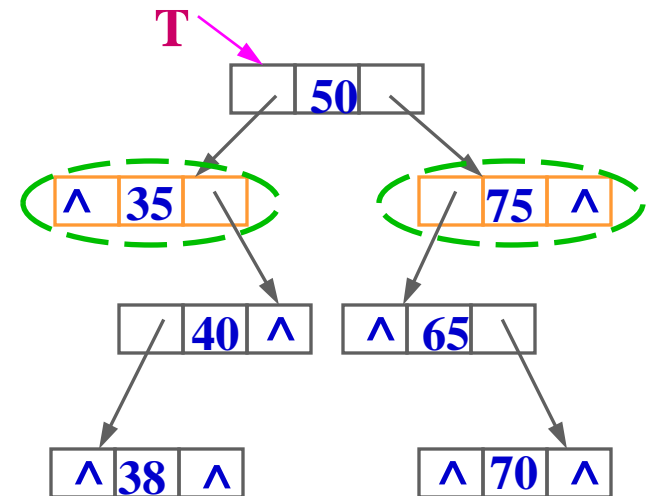
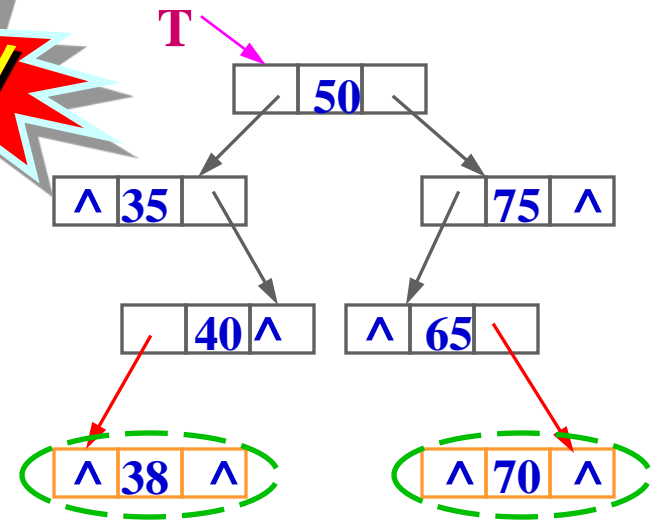
三. 二叉排序树的删除

原则

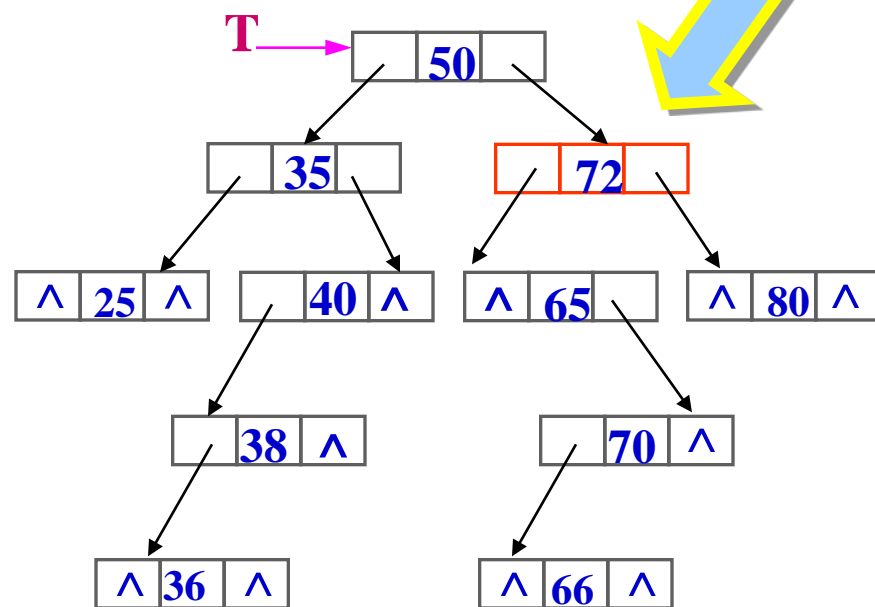
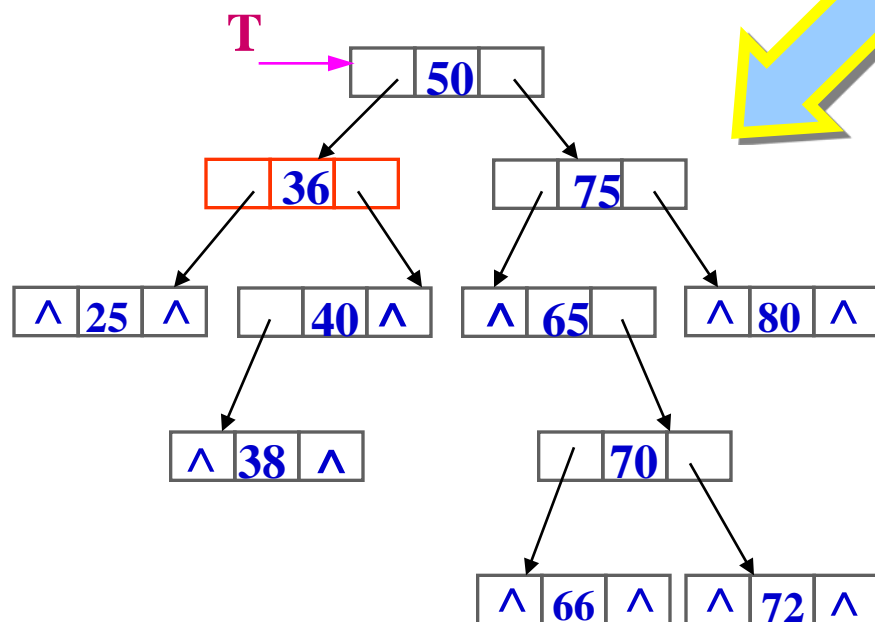
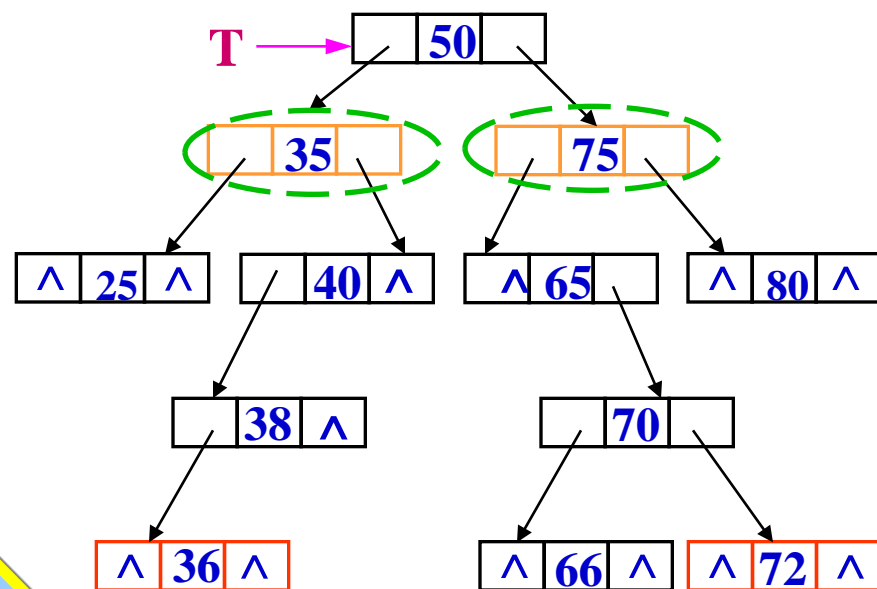
1. 被删除结点为叶结点，
则直接删除。

2. 被删除结点无左子树，
则用右子树的根结点
取代被删除结点。

3. 被删除结点无右子树，
则用左子树的根结点
取代被删除结点。



4. 被删除结点的左、右子树都存在,则用被删除结点的右子树中值最小的结点(或被删除结点的左子树中值最大的结点)取代被删除结点.



注意：这种结点未必都是叶结点

四. 二叉排序树的查找

递归过程

1. 查找过程

若二叉排序树为空, 则查找失败, 结束。

若二叉排序树非空, 则将被查找元素与二叉排序树的根结点的值进行比较,

- 若等于根结点的值, 则查找成功, 返回被查到元素所在链结点的地址, 查找结束;

- 若小于根结点的值, 则到根结点的左树中重复上述查找过程;

- 若大于根结点的值, 则到根结点的右子树中重复上述查找过程;

直到查找成功或者失败。

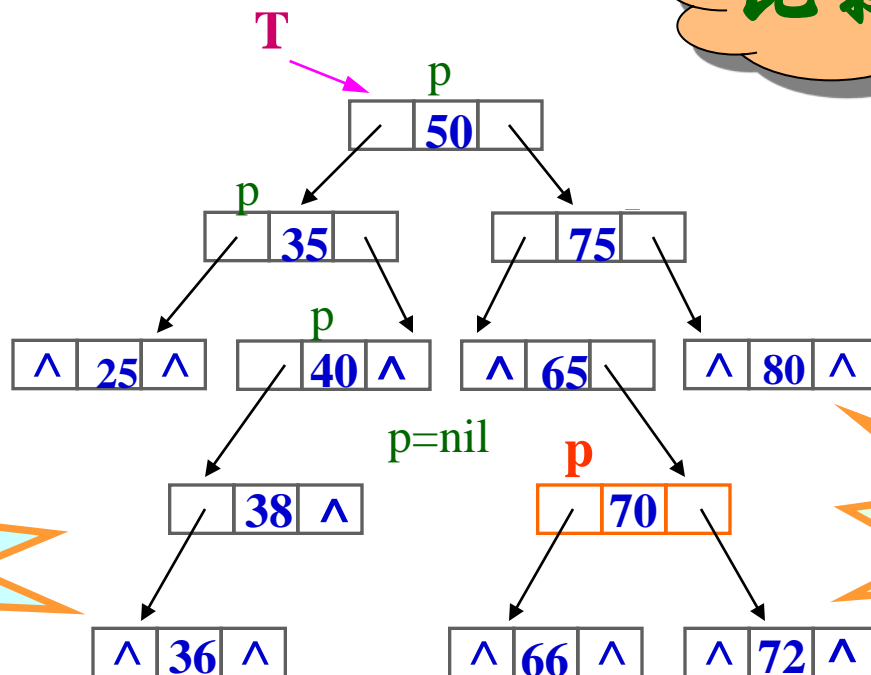
约定

若查找成功，给出被查找元素所在结点的地址；
若查找失败，给出信息NULL.

二叉树采用二叉链表存储结构

例

比较次数?



查找失败!

查找成功!

key=45

查找

key=70

查找成功的比较次数等于该结点所处的层次数

2. 查找算法

递归算法

```
BTREE SORTSEARCH1( BTREE T, datatype key )
{
    if ( T!=NULL ) {
        if ( T->data==key )
            return( T );
        if ( T->data<key )
            return( SORTSEARCH1( T->rchild, key ) );
        else
            return( SORTSEARCH1( T->lchild, key ) );
    }
    else
        return( NULL );
}
```

/* 查找成功 */
/* 查找T的右子树 */
/* 查找T的左子树 */
/* 查找失败 */

非递归算法

```
BTREE SORTSEARCH2( BTREE T, datatype key )
{
    BTREE p=T;
    while ( p!=NULL ) {
        if ( p->data==key )
            return(p);          /* 查找成功 */
        if ( p->data<key )
            p=p->rchild;         /* 将p移到右子树的根结点 */
        else
            p=p->lchild;         /* 将p移到左子树的根结点 */
    }
    return( NULL );            /* 查找失败 */
}
```

需要说明

如果被插入的元素序列是随机序列，或者序列的长度较小，“逐点插入法”可以接受。如果建立的二叉排序树中出现结点子树的深度之差较大时（即产生不平衡），就有必要采用其他方法建立二叉排序树，即建立所谓“**平衡二叉树**”。

查找时间

$O(\log_2 n)$

比较理想的情况

$O(n)$

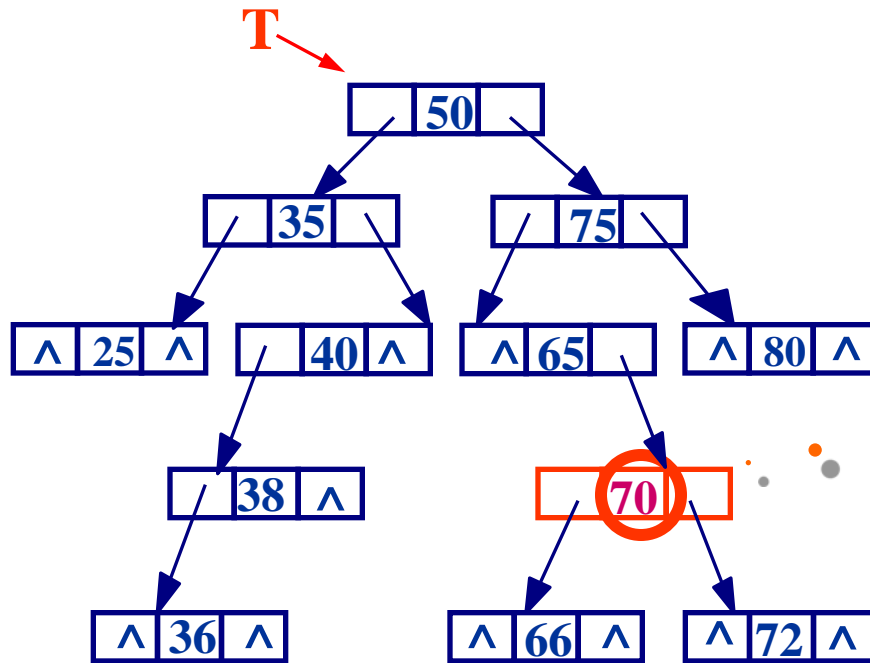
当出现“退化二叉树”
时

一般二叉树



例

已知二叉排序树采用二叉链表存储结构，根结点地址为T，请写一非递归算法，打印数据信息为item的结点的所有祖先结点。



从根结点到该结点的所有分支上经过的结点

祖先结点:
50, 75, 65

非递归算法

没有考虑item
可能不存在的情况

```
void SORTSEARCH( BTREE T, datatype item )
{
    BTREE p=T;
    while ( p!=NULL ) {
        print( p->data );
        if ( p->data==item )
            break;
        if ( p->data<item )
            p=p->rchild;
        else
            p=p->lchild;
    }
}
```

printf(“%d”,p->data);

/* 查找成功 */

/* 将p 移到右子树的根结点 */

/* 将p 移到左子树的根结点 */

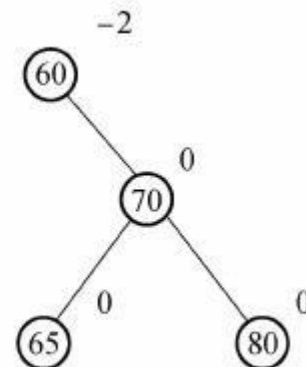
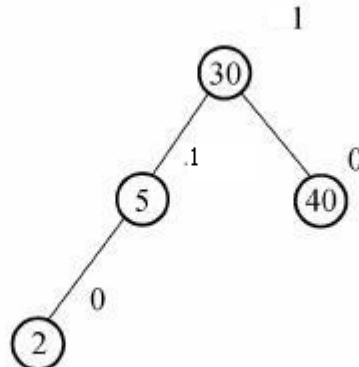
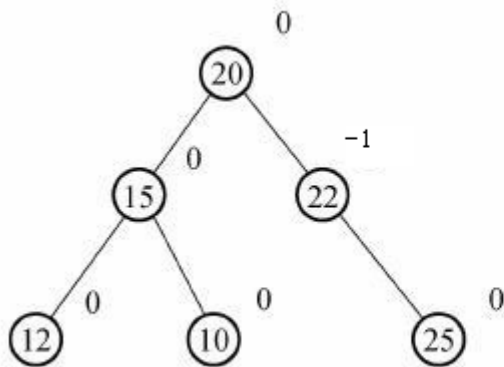
7.8 平衡二叉树

二叉排序树的缺陷—— 树的形态无法预料、随意性大。得到的可能是一个不平衡的树，即树的深度差很大。丧失了利用二叉树组织数据带来的好处。

一、平衡二叉树的定义

平衡二叉树又称AVL树。它或者是一棵空树,或者是具有下列性质的二叉树:

它的左子树和右子树都是平衡二叉树, 且左子树和右子树的深度之差的绝对值不超过1。若将二叉树的平衡因子定义为该结点左子树深度减去右子树深度, 则平衡二叉树上所有结点的平衡因子只可能是-1、0和1。



二、平衡二叉树的调整（略）

7.9 哈夫曼树及其应用

一、哈夫曼树的基本概念

1. 树的带权路径长度

若给具有 m 个叶结点的二叉树的每个叶结点赋予一个权值，则该二叉树的带权路径长度定义为

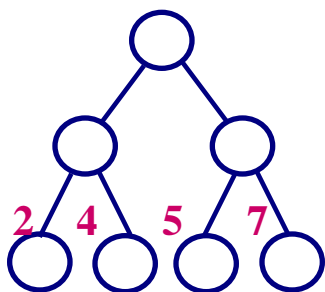
$$WPL = \sum_{i=1}^m w_i l_i$$

其中， w_i 为第 i 个叶结点被赋予的权值， l_i 为第 i 个叶结点的路径长度。

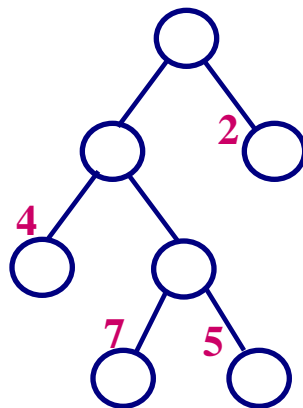
结点之间的路径——这两个结点之间的分支。**路径长度**——路径上经过的分支数目。
树的路径长度——根结点到所有结点的路径长度之和。

例

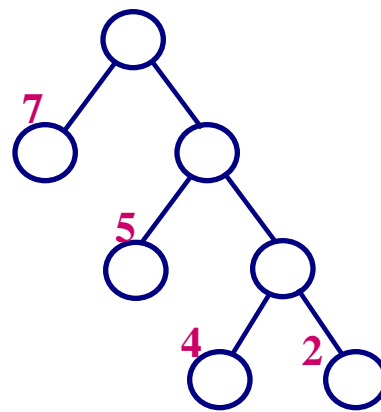
对于一组权值{ 7, 5, 2, 4 }，可以构造出如下不同带权二叉树



WPL=36



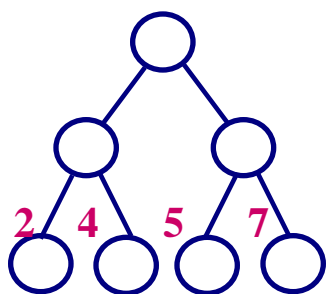
WPL=46



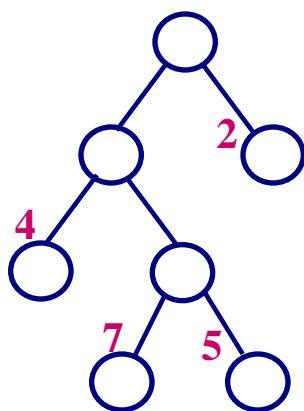
WPL=35

2. 哈夫曼树的定义

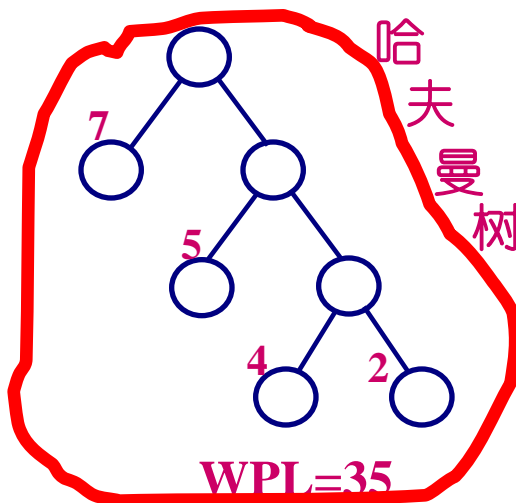
给定一组权值，构造出的具有最小带权路径长度的二叉树称为**哈夫曼树**。



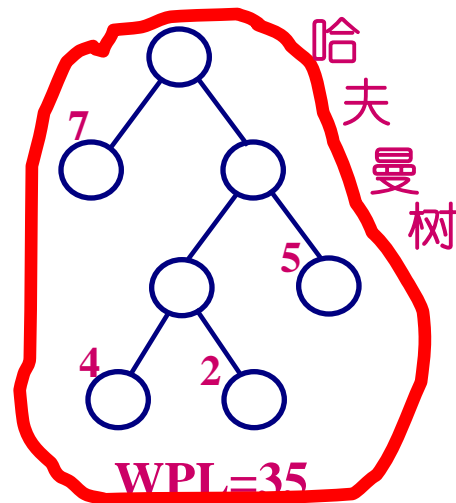
WPL=36



WPL=46



WPL=35



WPL=35

一组权值 { 7, 5, 2, 4 }

3. 哈夫曼树的特点

- (1) 权值越大的叶结点离根结点越近，权值越小的叶结点离根结点越远；(这样的二叉树WPL最小，也叫最优二叉树)
- (2) 无度为1的结点；
- (3) 哈夫曼树不是惟一的。

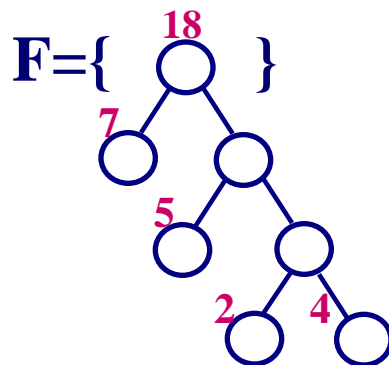
二、哈夫曼树的构造

核心思想

1. 对于给定的权值 $W=\{w_1, w_2, \dots, w_m\}$, 构造出树林 $F=\{T_1, T_2, \dots, T_m\}$, 其中, $T_i (1 \leq i \leq m)$ 为左、右子树为空, 且根结点(叶结点)的权值为 w_i 的二叉树。
2. 将 F 中根结点权值最小的两棵二叉树合并成为一棵新的二叉树, 即把这两棵二叉树分别作为新的二叉树的左、右子树, 并令新的二叉树的根结点权值为这两棵二叉树的根结点的权值之和, 将新的二叉树加入 F 的同时从 F 中删除这两棵二叉树。
3. 重复步骤2, 直到 F 中只有一棵二叉树。

例

$W=\{7, 5, 2, 4\}$



哈夫曼树

三、哈夫曼编码

例

要传送的文字
'ABACCD A'
由二进制代码组成的电文
00010010101100

若假设 A=00 B=01 C=10 D=11 (编码等长)

若假设 A=0 B=00 C=1 D=01 (编码不等长)

要传送的文字
'ABACCD A'
由二进制代码组成的电文
000011010

AAAA ABA BB

优点：电文中出现频率大的字符编码短，电文总长减少。

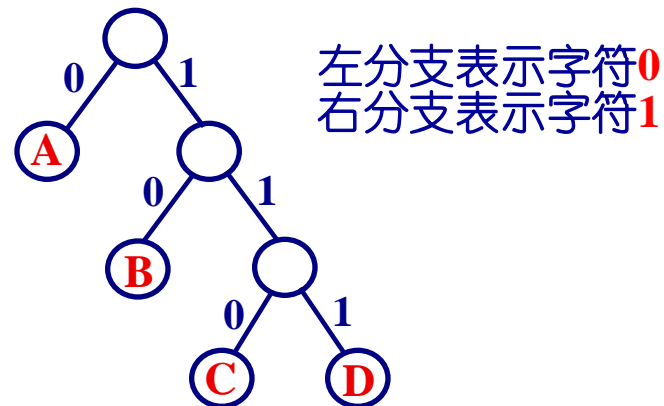
缺点：编码表示不惟一，致使电文难以翻译。

要求：任意一个字符的编码不能是另一个字符的编码的前缀。

前缀编码

从根结点到叶结点的路径
上分支字符组成的字符串作为
该叶结点字符的编码。

A: 0 B: 10 C: 110 D: 111



如何得到使电文
总长最短的编码?



哈夫曼编码

设 电文中包含 n 种字符;
 w_i 为每种字符在电文中出现的次数;
 l_i 为相应的编码长度。

则 电文总长度为 $\sum_{i=1}^n w_i l_i$ (二叉树) w_i 为叶结点的权值;
 l_i 为根结点到叶结点的路径长度

二叉树的带权路径长度

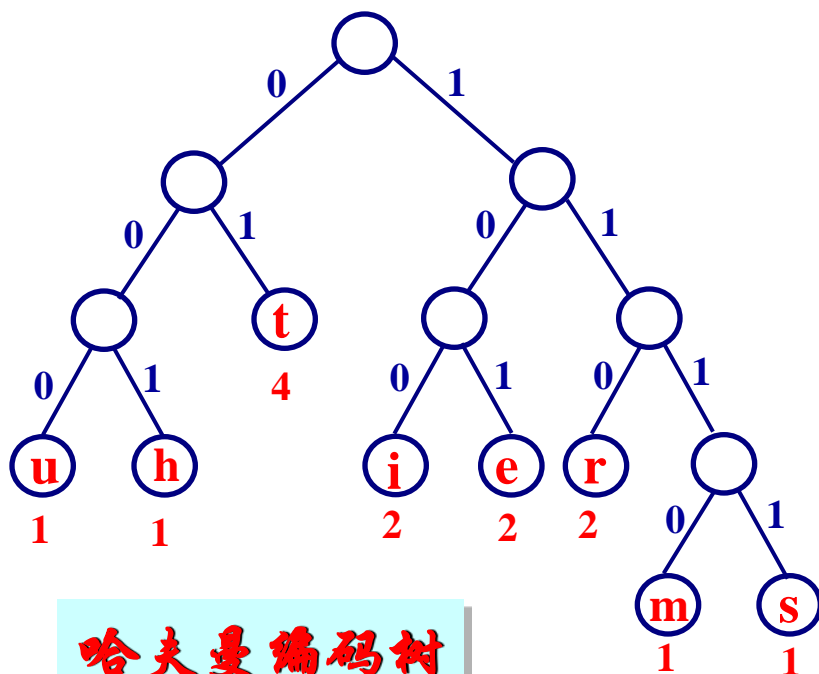
例

电文: **time tries truth**

字符集: { t, i, m, e, r, s, u, h }

字符在电文中出现的次数:

{ 4, 2, 1, 2, 2, 1, 1, 1 }



哈夫曼编码树

哈夫曼编码

t: 01
i: 100
m: 1110
e: 101
r: 110
s: 1111
u: 000
h: 001

01 100 1110 101 01 110 100 101 1111 01 110 000 01 001

本章内容小结



1. 树的基本概念

- 树的定义
- 树型结构逻辑上的特点
- 树的逻辑表示方法

文氏图表示法、凹入表示法、嵌套括号表示法、
树型表示法

- 基本名词术语

结点的度与树的度、叶结点与分支结点、层次的定义、深度的定义、有序树与无序树、森林

2. 树的存储方法

- 多重链表方法

定长的多重链表与不定长的多重链表

- 三重链表方法

3. 二叉树的基本概念

- 二叉树的定义
 - 二叉树逻辑上的特点
- 两种特殊形态的二叉树
 - 满二叉树 完全二叉树
- 二叉树基本性质（6个）
- 二叉树与树、树林之间的转换

4. 二叉树的存储结构

- 顺序存储结构
 - 构造原理、特点
 - 由顺序存储结构（一维数组）恢复二叉树
- 链式存储结构
 - 二叉链表与三叉链表

5. 二叉树的遍历

- 什么是二叉树的遍历

- 最常用的遍历方法有哪几种？

前序遍历、中序遍历、后序遍历与层次遍历

每一种遍历方法的过程、规律和特点

遍历方法对应的递归和非递归算法的设计

由遍历序列恢复二叉树

6. 线索二叉树

- 什么是线索二叉树？

- 线索二叉树的构造

- 如何利用线索二叉树对二叉树进行遍历

- 线索二叉树的建立算法

7. 二叉排序树

- 什么是二叉排序树？

二叉排序树的定义、特点

- 二叉排序树的建立

建立二叉排序树的“逐点插入方法”

- 二叉排序树的删除（原则）

- 二叉排序树的查找

不作重点

查找过程

查找算法（递归算法、非递归算法）

8. 平衡二叉树（定义）

9. 哈夫曼树

- 基本概念（带权路径长度、定义）
- 哈夫曼树构造及编码

习题课

(4)

