



## 第6章 树

6.1 树的定义

6.2 树的遍历

6.3 树的表示法

6.4 二叉树的基本概念

6.5 ADT二叉树

6.6 线索二叉树

6.7 树的应用

## 学习要点:

- 理解树的定义和与树相关的结点、度、路径等术语。
- 理解树是一个非线性层次数据结构。
- 掌握树的前序遍历、中序遍历和后序遍历方法。
- 了解树的父结点数组表示法。
- 了解树的儿子链表表示法。
- 了解树的左儿子右兄弟表示法。
- 理解二叉树的概念。
- 了解二叉树的顺序存储结构。
- 了解二叉树的结点度表示法。
- 掌握用指针实现二叉树的方法。
- 理解线索二叉树结构及其适用范围。

## 6.1 树的定义

### ❁ 定义

#### ❁ 递归定义

(1) 单个结点是一棵树，该结点就是树根。

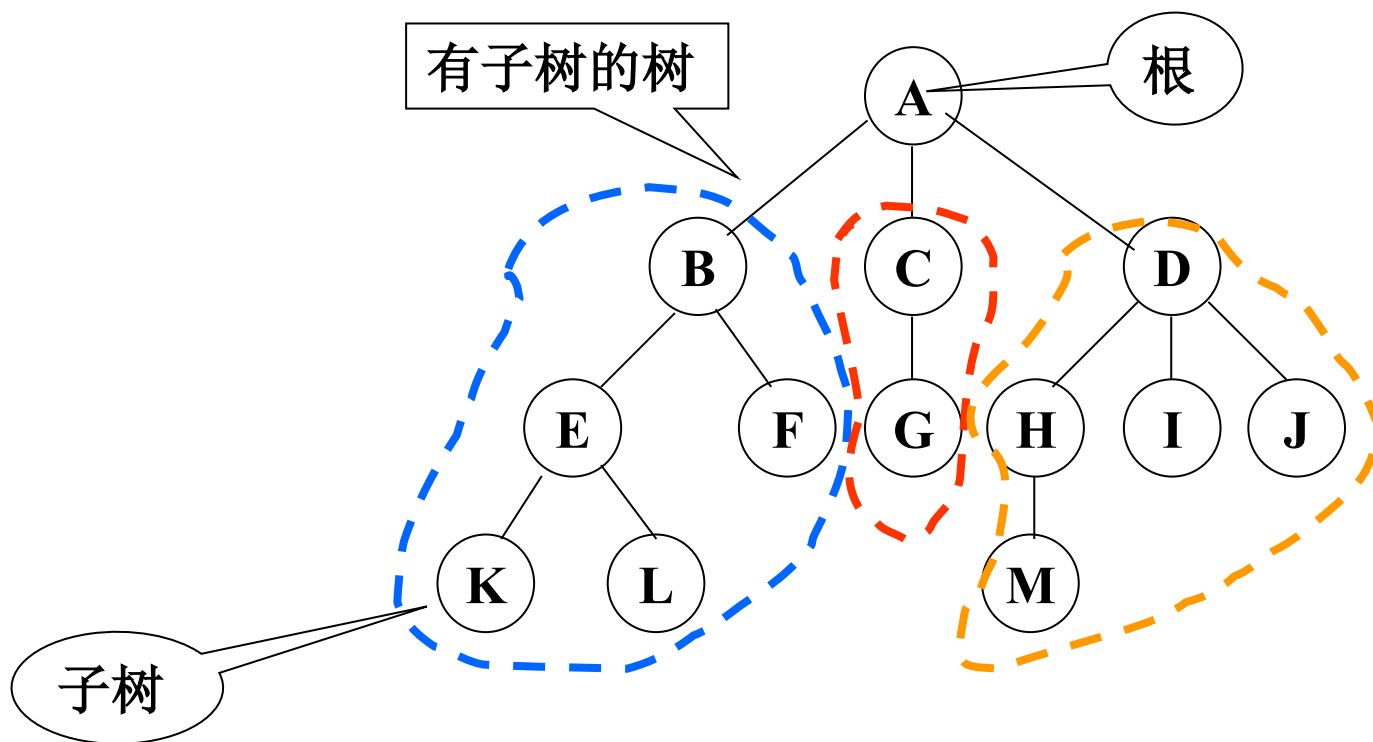
(2) 设 $T_1, T_2, \dots, T_k$ 都是树，它们的根结点分别为 $n_1, n_2, \dots, n_k$ ，而 $n$ 是另一个结点且以 $n_1, n_2, \dots, n_k$ 为儿子，则 $T_1, T_2, \dots, T_k$ 和 $n$ 构成一棵新树。结点 $n$ 就是新树的根。称 $n_1, n_2, \dots, n_k$ 为一组兄弟结点。还称 $T_1, T_2, \dots, T_k$ 为结点 $n$ 的子树。

为了方便起见，空集合也看作是树，称为空树，并用 $\wedge$ 来表示。空树中没有结点。

只有根结点的树



有子树的树

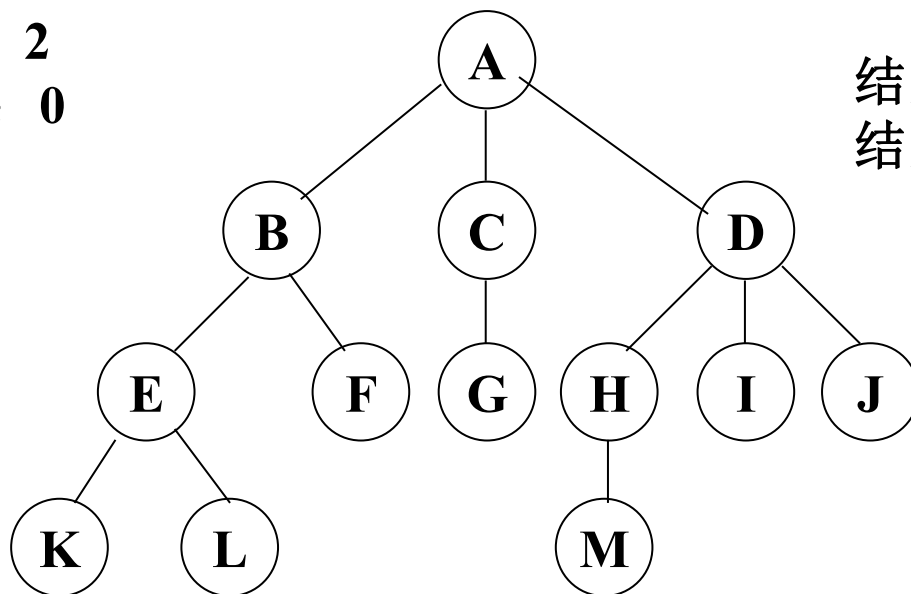


## ❁ 基本术语

- ❁ 结点——表示树中的元素，包括数据项及若干指向其子树的分支
- ❁ 结点的度——结点的儿子结点个数
- ❁ 树的度——一棵树中最大的结点度数
- ❁ 叶结点——度为0的结点
- ❁ 分枝结点——度不为0的结点
- ❁ 路径——若存在树中的一个节点序列 $k_1, k_2, \dots, k_j$ ，使得结点 $k_i$ 是 $k_{i+1}$ 的父结点( $1 \leq i < j$ )，则称该结点序列是树中从结点 $k_1$ 到结点 $k_j$ 的一条路径。
- ❁ 路径长度——路径所经过的边的数目。
- ❁ 祖先、子孙
- ❁ 结点的高度——从该结点到各叶结点的最长路径长度
- ❁ 树的高度——根结点的高度
- ❁ 结点的深度(或层数)——从树根到任一结点 $n$ 有唯一的路径，称该路径的长度为结点 $n$ 的深度(或层数)。从根结点算起，根为第0层，它的孩子为第1层.....
- ❁ 有序树——为树的每一组兄弟结点定义一个从左到右的次序
- ❁ 左儿子、右兄弟
- ❁ 森林—— $m(m \geq 0)$ 棵互不相交的树的集合

结点A的度: 3  
结点B的度: 2  
结点M的度: 0

树的度: 3



结点A的孩子: B, C, D  
结点B的孩子: E, F

结点A的层次: 0  
结点M的层次: 3

结点A是结点F, G的祖先  
结点B, C, ...是结点A的子孙

叶结点: K, L, F, G, M, I, J  
分枝节点: A, B, C, D, E, H

结点I的双亲: D  
结点L的双亲: E

结点B, C, D为兄弟  
结点K, L为兄弟

结点A的高度: 3  
结点D的高度: 2  
树的高度: 3

[返回章节目录](#)

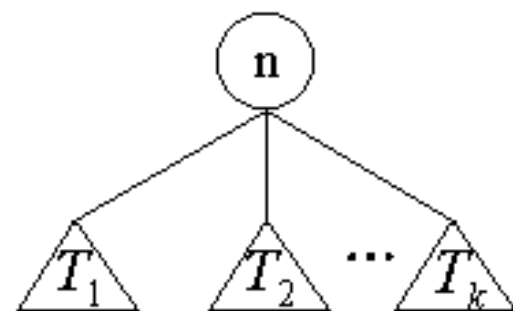
## 6.2 树的遍历

### ④ 树的遍历

④ 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。

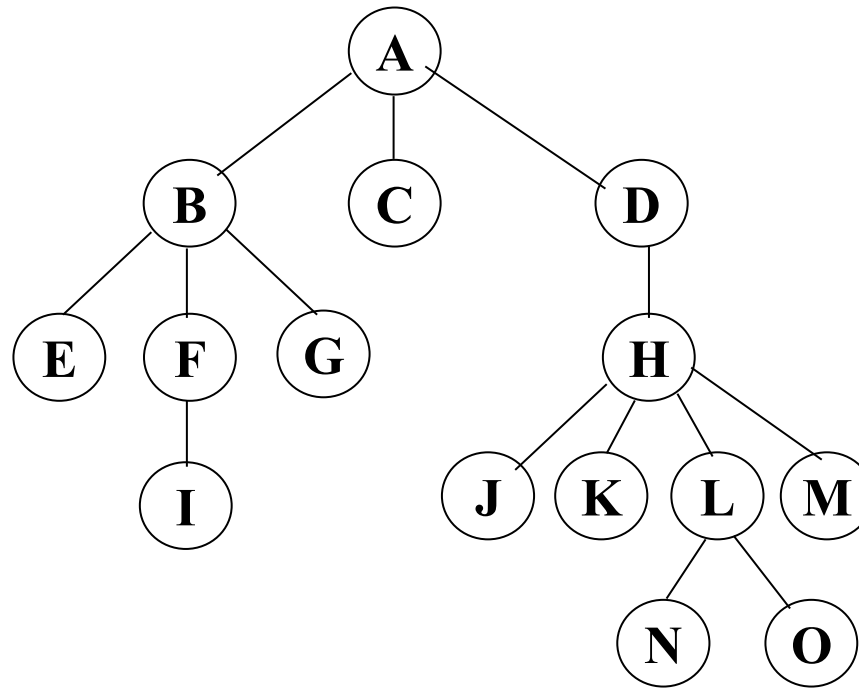
④ 树T的3种遍历方式的递归定义：(T 如图所示)

- (1) 前序遍历——先访问树根 $n$ ，然后依次前序遍历 $T_1, T_2, \dots, T_k$ 。
- (2) 中序遍历——先中序遍历 $T_1$ ，然后访问树根 $n$ ，接着依次对 $T_2, T_3, \dots, T_k$ 进行中序遍历。
- (3) 后序遍历——先依次对 $T_1, T_2, \dots, T_k$ 进行后序遍历，最后访问树根 $n$ 。



树T





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

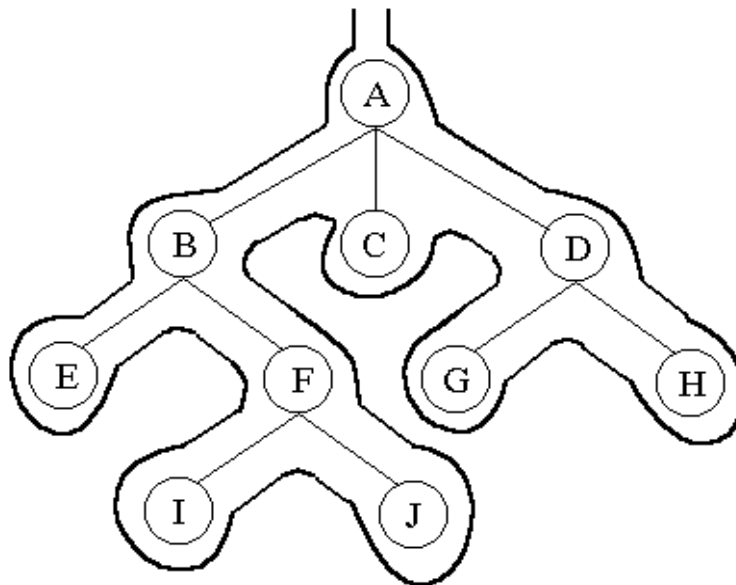
中序遍历: **E B I F G A C J H K N L O M D**

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**



## 有序树T的3种遍历的非递归方式



按第一次经过的时间次序将各个结点列表:

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

前序列表

按最后一次经过的时间次序将各个结点列表:

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

后序列表

叶结点在第一次经过时列出, 而内部结点在第2次经过时列出:

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

中序列表



## 有序树T的3种遍历能得到什么信息？

- (1) 在3种不同方式中，各树叶之间的相对次序是相同的，它们都按树叶之间从左到右的次序排列，其差别仅在于内部结点之间以及内部结点与树叶之间的次序有所不同。
- (2) 后序遍历有助于查询结点间的祖先和子孙关系，因为
$$\text{postorder}(y) - \text{desc}(y) \leq \text{postorder}(x) \leq \text{postorder}(y)。$$
其中 $y$ 是 $T$ 中的任一结点； $x$ 是 $y$ 的子孙； $\text{desc}(y)$ 是 $T$ 中 $y$ 的真子孙数； $\text{postorder}(y)$ 是 $T$ 中 $y$ 的后序序号。
- (3) 前序遍历也有助于查询结点间的祖先和子孙关系，因为
$$\text{preorder}(y) \leq \text{preorder}(x) \leq \text{preorder}(y) + \text{desc}(y)。$$
其中 $y$ 是 $T$ 中的任一结点； $x$ 是 $y$ 的子孙； $\text{desc}(y)$ 是 $T$ 中 $y$ 的真子孙数； $\text{preorder}(y)$ 是 $T$ 中 $y$ 的前序序号。

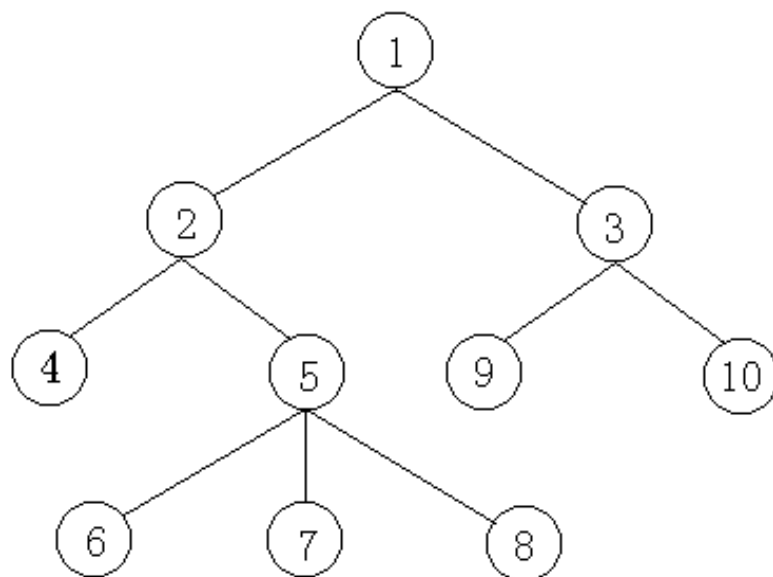
[返回目录](#)

## 6.3 树的表示法

### • 树的存储结构

#### • 父结点数组表示法

- (1) 树中的结点数字化为它们的编号 $1, 2, \dots, n$ 。
- (2) 用一个一维数组存储每个结点的父结点。即：  
**father[k]**中是存放结点**k**的父结点的编号。
- (3) 由于树中每个结点的父结点是唯一的，所以父结点数组表示法可以唯一表示任何一棵树。
- (4) 实例：见下页





(a)

0	1	1	2	2	5	5	5	3	3
1	2	3	4	5	6	7	8	9	10

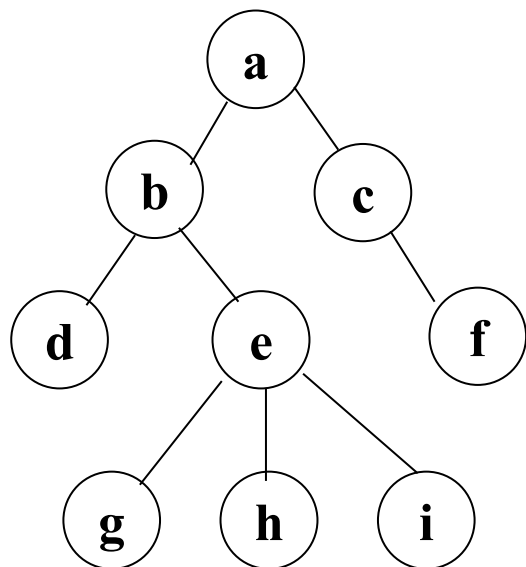
(b)

如何找孩子结点

## 效率分析

-  寻找一个结点的父结点只需要 $O(1)$ 时间。
-  于涉及查询儿子结点和兄弟结点信息的运算，可能要遍历整个数组。

## 孩子表示法

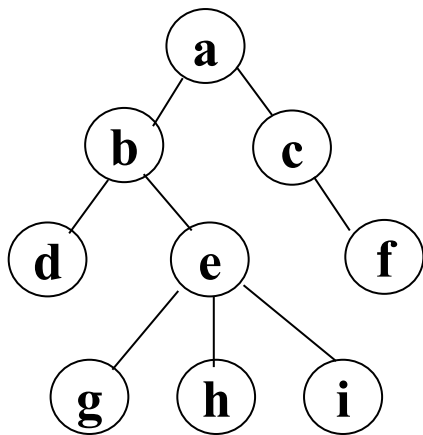


如何找双亲结点

data

0		
1	a	→ 2 → 3 ^
2	b	→ 4 → 5 ^
3	c	→ 6 ^
4	d	^
5	e	→ 7 → 8 → 9 ^
6	f	^
7	g	^
8	h	^
9	i	^

## 带双亲的孩子链表

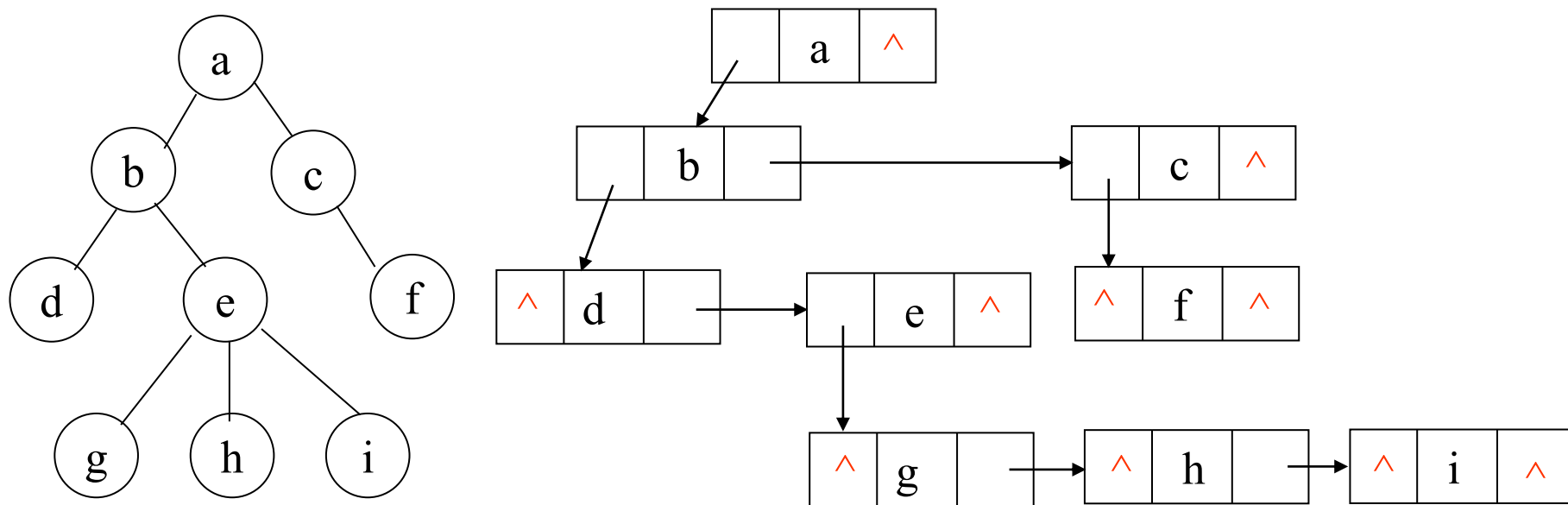


	data	parent	
1	a	0	→ 2 → 3 ^
2	b	1	→ 4 → 5 ^
3	c	1	→ 6 ^
4	d	2	^
5	e	2	→ 7 → 8 → 9 ^
6	f	3	^
7	g	5	^
8	h	5	^
9	i	5	^



## 左儿子右兄弟表示法

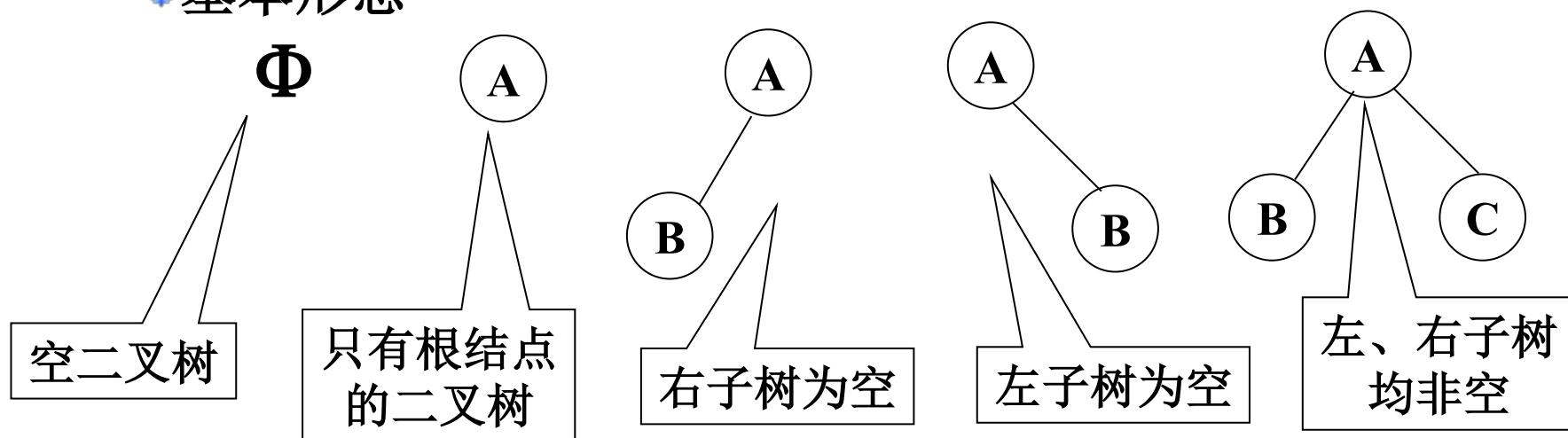
- 实现：用二叉链表作树的存储结构，链表中每个结点的两个指针域分别指向其最左儿子和右邻兄弟



[返回章节目录](#)

## 6.4 二叉树的基本概念

- 定义：二叉树是 $n(n \geq 0)$ 个结点的有限集，它或为空树( $n=0$ )，或由一个根结点和两棵分别称为左子树和右子树的互不相交的二叉树构成。
- 特点
  - 每个结点至多有二棵子树(即不存在度大于2的结点)
  - 二叉树的子树有左、右之分，且其次序不能任意颠倒
- 基本形态





- 具有n个结点的不同形态的二叉数的数目即所谓的n阶卡特兰数(Catalan number) :

$$B_n = \frac{1}{n+1} \binom{2n}{n}$$

从第零项开始,  $B_n$  的前几项为

1,1,2,5,14,42,132,429,1430,.....

应用:

- ①n个结点的构成的二叉树种类
- ②与1,2,3,...,n的入栈序列对应的出栈序列种类数



# 卡特兰数的多种定义

$$B_0 = B_1 = 1$$

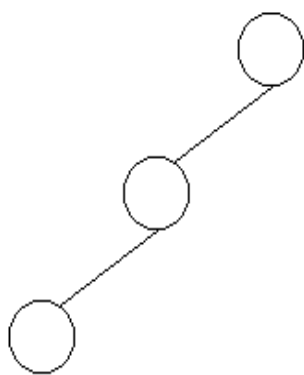
递归定义: 
$$B_n = \sum_{k=0}^{n-1} B_k B_{n-1-k}$$
$$= B_0 B_{n-1} + B_1 B_{n-2} + \dots + B_{n-1} B_0, \text{ 其中 } n \geq 2$$

递推公式: 
$$B_n = \frac{4n-2}{n+1} B_{n-1}$$

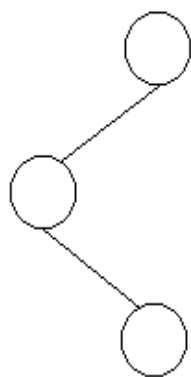
通项公式: 
$$B_n = \frac{1}{n+1} B_{2n}^n = B_{2n}^n - B_{2n}^{n-1}$$

$$B_n = \frac{1}{n+1} \sum_{i=0}^n (B_n^i)^2$$

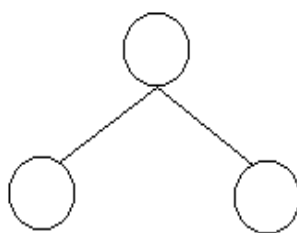
例如：具有3个结点的不同形态的二叉数的数目 $B_3=5$



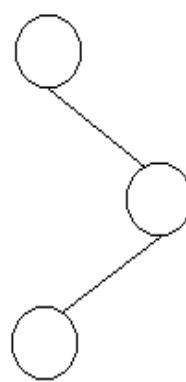
(a)



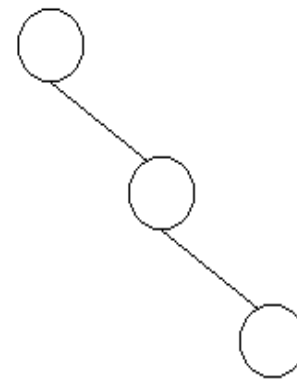
(b)



(c)



(d)



(e)

## ❖ 二叉树性质

- ❖ 高度为 $h \geq 0$ 的二叉树至少有 $h+1$ 个结点。
- ❖ 高度为 $h \geq 0$ 的二叉树至多有 $2^{h+1}-1$ 个结点。
- ❖ 约定空二叉树的高度为 $-1$ 。
- ❖ 含有 $n \geq 1$ 个结点的二叉树的高度至多为 $n-1$ 。
- ❖ 含有 $n \geq 1$ 个结点的二叉树的高度至少为 $\lfloor \log n \rfloor$ ，因此为 $\Omega(\lfloor \log n \rfloor)$ 。  $2^{h+1}-1=n$

- 重要性质：对任何一棵二叉树T，如果其终端结点数为 $n_0$ ，度为2的结点数为 $n_2$ ，  
则： $n_0 = n_2 + 1$

证明： $n_1$ 为二叉树T中度为1的结点数

因为：二叉树中所有结点的度均小于或等于2

所以：其结点总数 $n = n_0 + n_1 + n_2$

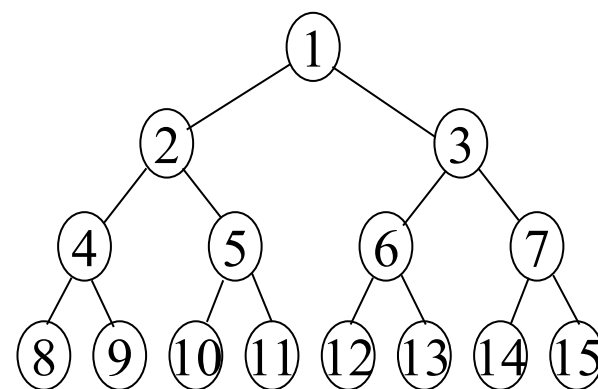
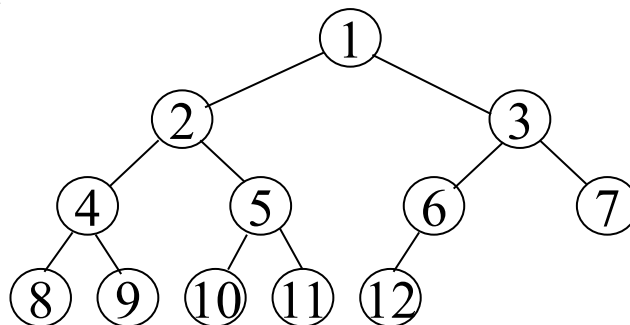
又二叉树中，除根结点外，其余结点都只有一个分支进入  
设B为分支总数，则 $n = B + 1$

又：分支由度为1和度为2的结点射出

$$\therefore B = n_1 + 2n_2$$

$$\text{于是， } n = B + 1 = n_1 + 2n_2 + 1 = n_0 + n_1 + n_2$$

$$\therefore n_0 = n_2 + 1$$





## ◆ 几种特殊形式的二叉树

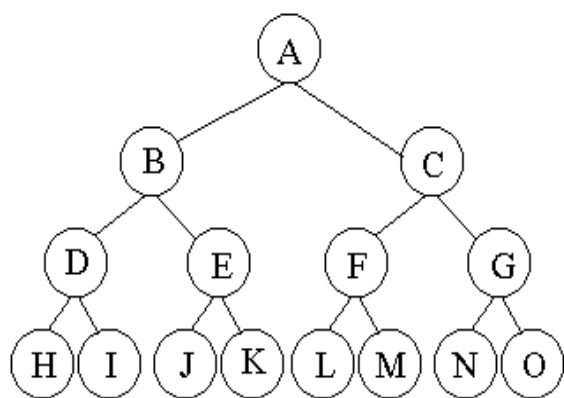
### ◆ 满二叉树

◆ 定义：一棵高度为 $k$ 且有 $2^{k+1}-1$ 个结点的二叉树称为 ~

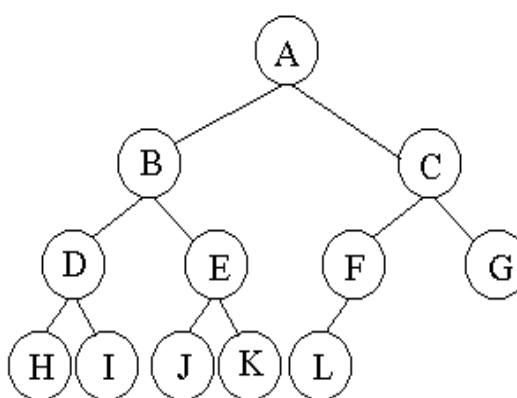
◆ 特点：每一层上的结点数都是最大结点数

### ◆ 近似满二叉树（完全二叉树）

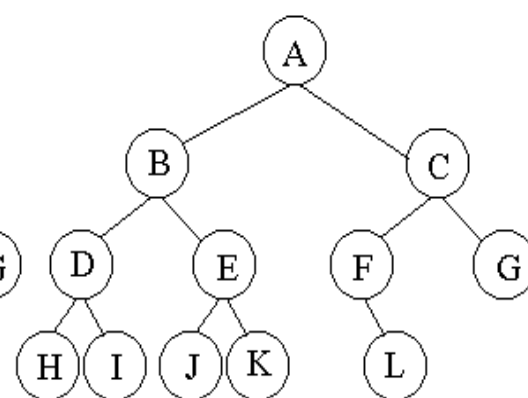
◆ 定义：若一棵二叉树最多只有最下面的2层上结点的度数可以小于2，并且最下面一层上的节点都集中在该层的**最左边**，则这种二叉树称为近似满二叉树



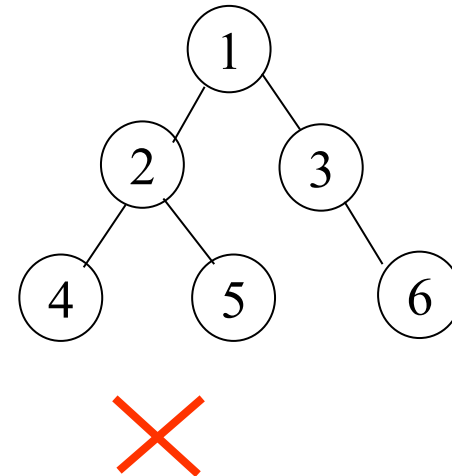
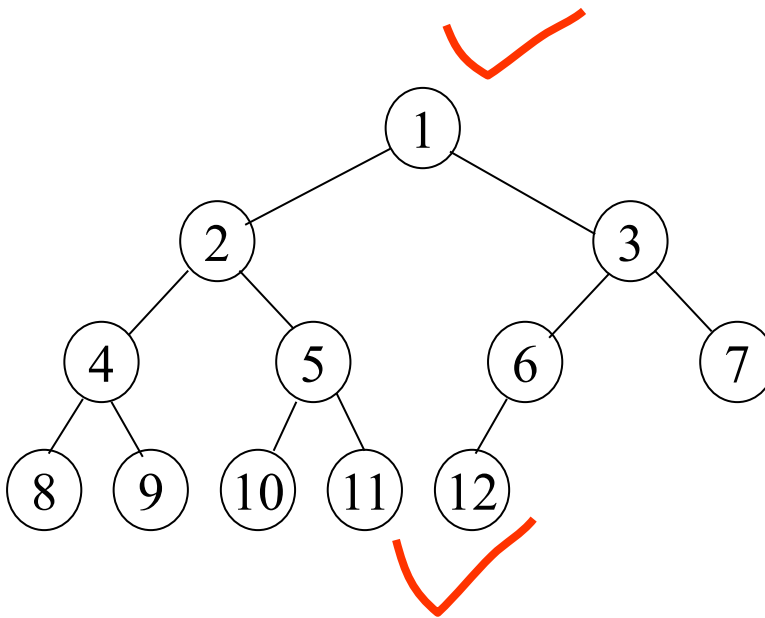
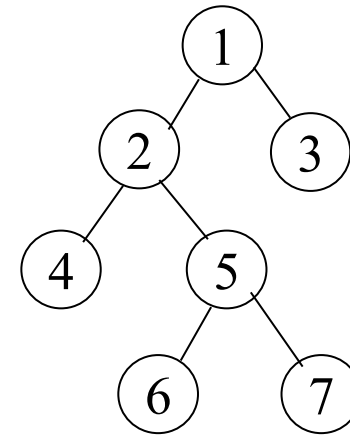
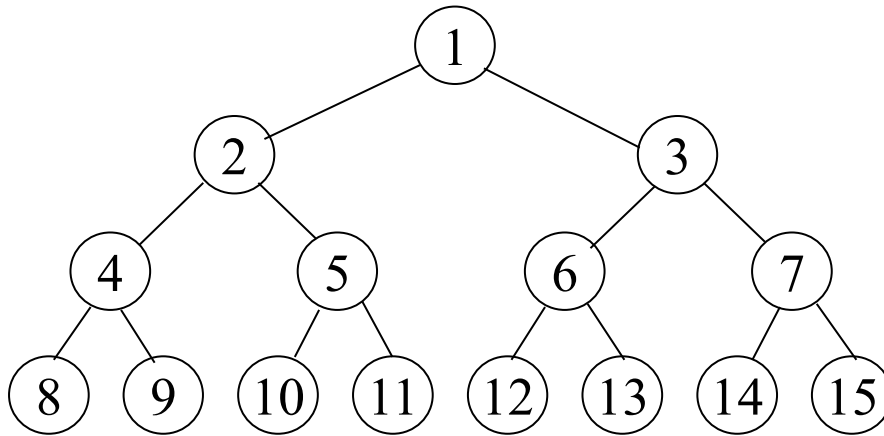
(a) 满二叉树



(b) 近似满二叉树



(c) 非近似满二叉树



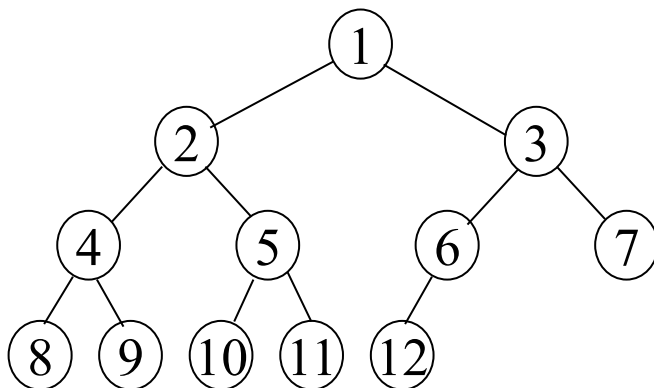
## 近似满二叉树性质：

如果对一棵有 $n$ 个结点的完全二叉树的结点按层序编号，则对任一结点 $i$  ( $1 \leq i \leq n$ )，有：

(1) 如果 $i=1$ ，则结点 $i$ 是二叉树的根，无双亲；如果 $i>1$ ，则其双亲是 $\lfloor i/2 \rfloor$

(2) 如果 $2i>n$ ，则结点 $i$ 无左孩子；如果 $2i \leq n$ ，则其左孩子是 $2i$

(3) 如果 $2i+1>n$ ，则结点 $i$ 无右孩子；如果 $2i+1 \leq n$ ，则其右孩子是 $2i+1$

[返回章节目录](#)

## 6.5 ADT二叉树

### 6.5.1 ADT二叉树的运算

**ADT二叉树支持的主要基本运算：**

**(1)BinaryInit()** 创建一棵空二叉树。

**(2)BinaryEmpty( $T$ )** 判断给定的二叉树是否为空。

**(3)Root( $T$ )** 返回给定二叉树 $T$ 的根结点标号。

**(4)MakeTree( $x, T, L, R$ )**以 $x$ 为根结点元素,以 $L$ 、 $R$ 分别为左、右子树,构建一棵新的二叉树 $T$ 。

**(5)BreakTree( $T, L, R$ )**函数**MakeTree**的逆运算，即将二叉树 $T$ 拆分为根结点元素，左子树 $L$ 和右子树 $R$ 等3部分。



## 6.5 ADT二叉树

### 6.5.1 ADT二叉树的运算

ADT二叉树支持的主要基本运算(续)

(6) **PreOrder(visit,  $T$ )** 前序遍历给定的二叉树。

(7) **InOrder(visit,  $T$ )** 中序遍历给定的二叉树。

(8) **PostOrder(visit,  $T$ )** 后序遍历给定的二叉树。

(9) **PreOut( $T$ )** 给定的二叉树的前序列表。

(10) **InOut( $T$ )** 给定的二叉树的中序列表。

(11) **PostOut( $T$ )** 给定的二叉树的后序列表。

(12) **Delete( $T$ )** 删除给定的二叉树。

(13) **Height( $T$ )** 求给定的二叉树的高度。

(14) **Size( $T$ )** 求给定的二叉树的结点数。

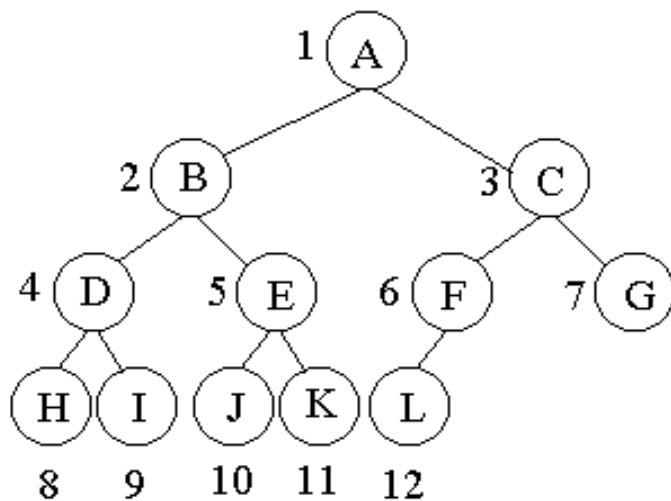


## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

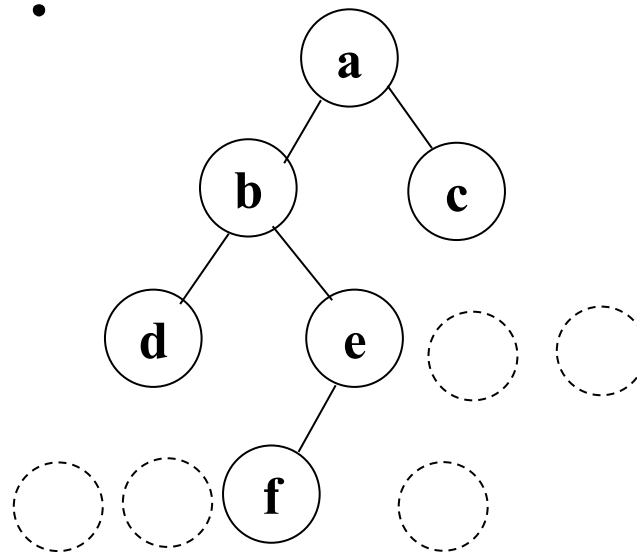
#### 6.5.2.1 用顺序存储结构实现(一种无边表示)

- 适用的对象：近似满二叉树
- 基本想法：若将所有的结点按层自上而下每层自左至右，从1开始编号。



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

❖ 不适用的例子：



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



## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.2 二叉树的结点度表示法(另一种无边表示)

✿ 例

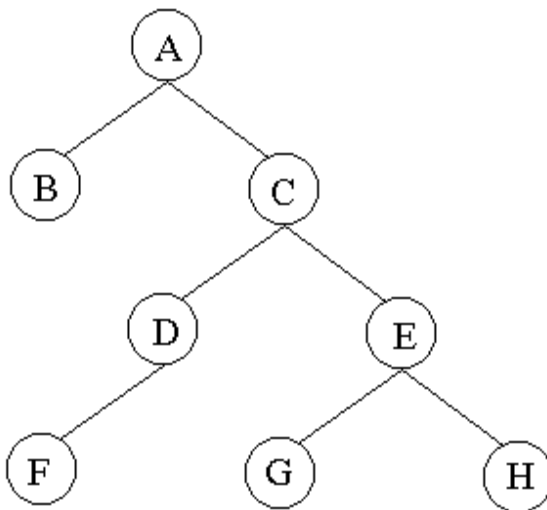
后序列表排列

0: 叶子节点

1: 只有左儿子

2: 只有右儿子

3: 有两个儿子



(a)

(B, 0)	(F, 0)	(D, 1)	(G, 0)	(H, 0)	(E, 3)	(C, 3)	(A, 3)
--------	--------	--------	--------	--------	--------	--------	--------

(b)



## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—二叉树结点结构定义

```
typedef struct btnode *btlink;  
typedef struct btnode {  
    Treeltem element;  
    btlink left;    /*左子树*/  
    btlink right;   /*右子树*/  
} Btnode ;
```

## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—ADT二叉树结构定义

```
typedef struct binarytree * Binarytree;  
typedef struct binarytree  
{  
    btlink root; //树根  
}BTree;
```

## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—创建一棵空二叉树

```
Binarytree BinaryInit ( )  
{  
    BinaryTree T=(BinaryTree*)malloc(sizeof *T);  
    T->root=0;  
    return T;  
}
```

## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—BinaryEmpty(T)、Root(T)实现

```
int BinaryEmpty (BinaryTree T)
{
    return T->root==0;
}

Treeltem Root(BinaryTree T)
{
    if (BinaryEmpty(T)) Error("Tree is empty");
    return T->root->element;
}
```

## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—MakeTree ( $x, T, L, R$ ) 的实现

```
void MakeTree (TreeItem x, BinaryTree T,  
               BinaryTree L, BinaryTree R)  
{ /*以x为根结点元素，L和R分别为左右子树构建一棵新的二叉树T*/  
    T->root=NewBNode();  
    T->root->element=x;  
    T->root->left=L->root;  
    T->root->right=R->root;  
    L->root=R->root=0;  
}
```



## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—BreakTree ( $T, L, R$ )的实现

```
TreeItem BreakTree (BinaryTree T,  
                    BinaryTree L, BinaryTree R)  
{ /*将二叉树T拆分为根结点元素element, 左子树L和右子树R三部分*/  
    TreeItem x;  
    if (!T->root) Error("Tree is empty");  
    x=T->root->element;  
    L->root=T->root->left;  
    R->root=T->root->right;  
    T->root=0;  
    return x;  
}
```



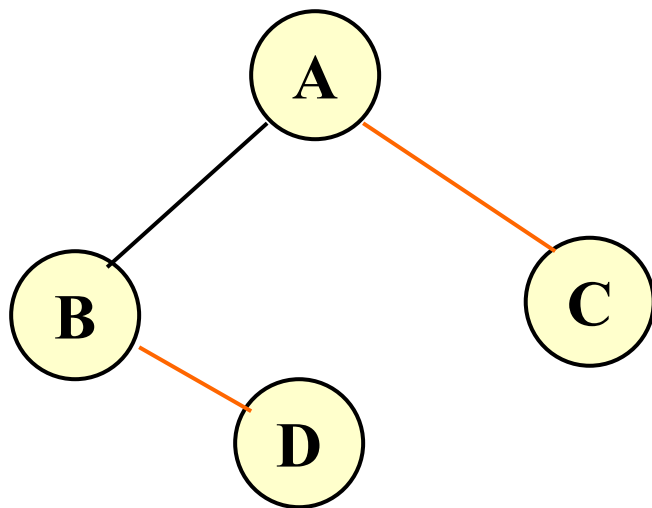
## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

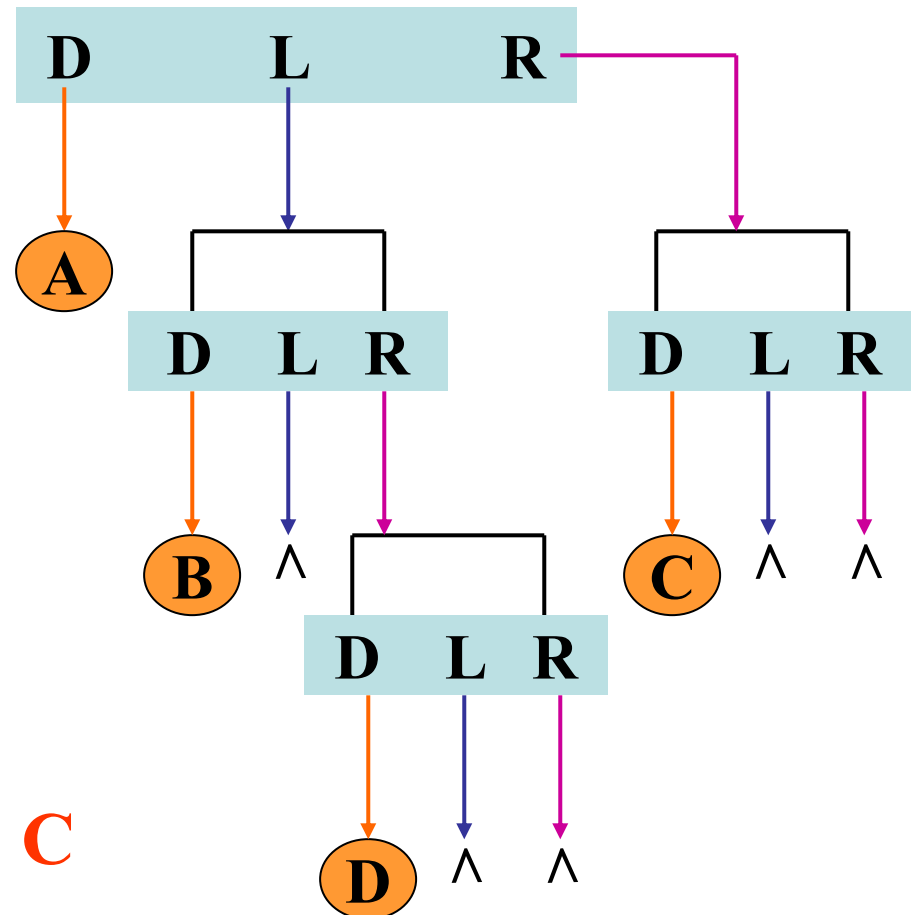
#### 6.5.2.3 二叉树的指针实现—前序遍历运算的递归实现

```
void PreOrder (void(*visit) (btlink u), btlink t)
{
    if ( t ) {
        (*visit) (t);
        PreOrder (visit, t->left);
        PreOrder (visit, t->right);
    }
}
```

## 前序遍历过程：



前序遍历序列：A B D C



## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—前序遍历运算的非递归实现

```
void PreOrder (void(*visit) (btlink u), btlink t)
{
    Stack s=StackInit ( );
    Push (t, s);
    while (!StackEmpty (s)) {
        (*visit) (t=Pop (s));
        if (t->right) Push (t->right, s);
        if (t->left) Push (t->left, s);
    }
}
```



## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—中序遍历运算的递归实现

```
void InOrder (void(*visit) (btlink u), btlink t)
{
    if ( t ) {
        InOrder (visit, t->left);
        (*visit) (t);
        InOrder (visit, t->right);
    }
}
```

```

graph TD
    A((A)) --- B((B))
    A --- C((C))
    B --- D((D))
    C --- D

```

The diagram illustrates the construction of a decision tree from a sequence of splits. The root node is a light blue rectangle labeled **L**, **D**, **R**. Three arrows (blue, orange, purple) point from **L**, **D**, **R** respectively to three child nodes. The left child is a light blue rectangle labeled **L**, **D**, **R**. The middle child is an orange circle labeled **A**. The right child is a light blue rectangle labeled **L**, **D**, **R**. From the left child, three arrows point to a light blue rectangle labeled **L**, **D**, **R**, an orange circle labeled **B**, and a light blue rectangle labeled **L**, **D**, **R**. From the middle child, three arrows point to a light blue rectangle labeled **L**, **D**, **R**, an orange circle labeled **C**, and a light blue rectangle labeled **L**, **D**, **R**. From the right child, three arrows point to a light blue rectangle labeled **L**, **D**, **R**, an orange circle labeled **D**, and a light blue rectangle labeled **L**, **D**, **R**.



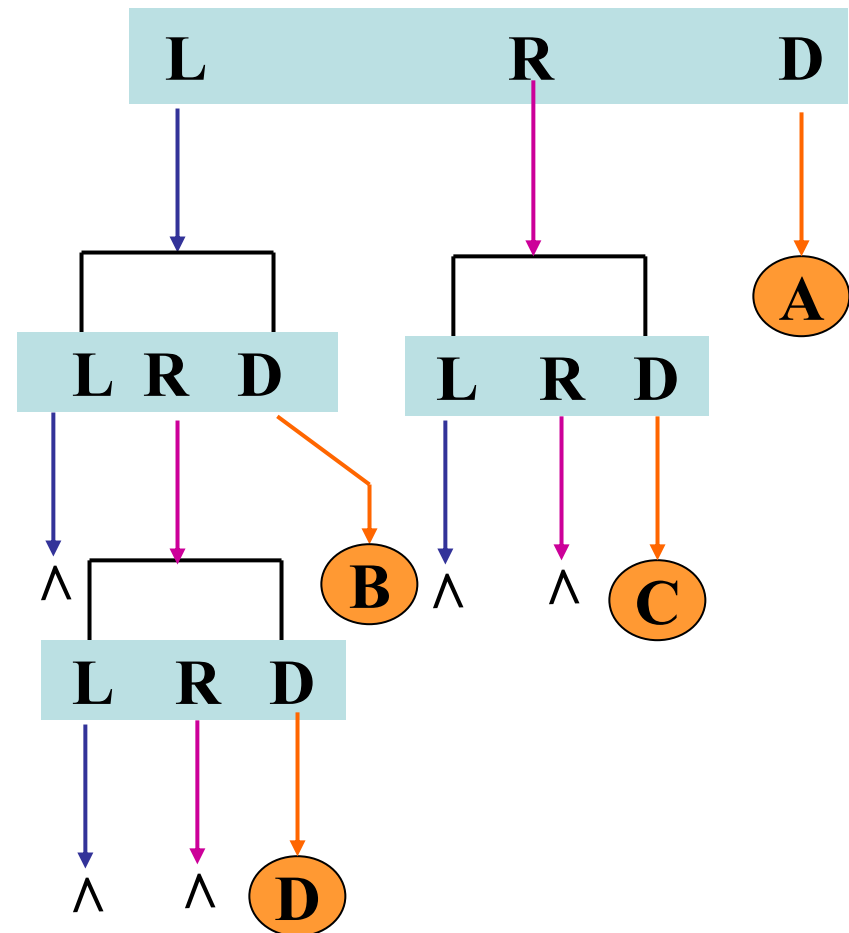
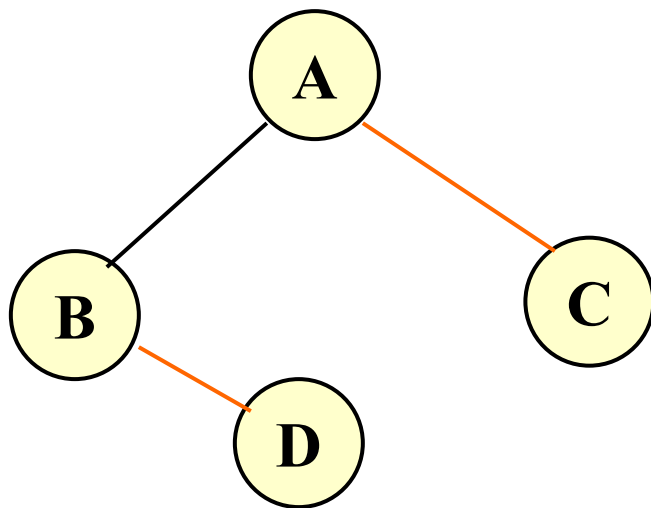
## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—后序遍历运算的递归实现

```
void PostOrder (void(*visit) (btlink u), btlink t)
{
    if ( t ) {
        PostOrder (visit, t->left);
        PostOrder (visit, t->right);
        (*visit) (t);
    }
}
```

## 后序遍历过程：



后序遍历序列： D B C A



## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—层次遍历运算的实现

```
void LevelOrder (void(*visit) (btlink u), btlink t)
{
    Queue q=QueueInit ( );
    EnterQueue (t, q);
    while (!QueueEmpty (q)) {
        (*visit) (t=DeleteQueue (q));
        if (t->left) EnterQueue (t->left, q);
        if (t->right) EnterQueue (t->right, q);
    }
}
```



## 6.5 ADT二叉树

### 6.5.2 ADT二叉树的实现

#### 6.5.2.3 二叉树的指针实现—求二叉树的结点数

```
int Size (btlink t)
{
    int lsize, rsize;
    if ( !t ) return 0;
    lsize=Size(t->left);
    rsize=Size(t->right);
    return lsize+rsize+1;
}
```

[返回章节目录](#)

## 6.6 线索二叉树

### 6.6.1 引入线索二叉树的动因

用指针实现二叉树时，每个结点只有指向其左、右儿子结点的指针，所以从任一结点出发直接只能找到该结点的左、右儿子。在一般情况下无法直接找到该结点在某种遍历序下的前驱和后继结点。若在每个结点中增加指向其前驱和后继结点的指针，虽可提高遍历的效率却降低了存储效率。注意到用指针实现二叉树时， $n$ 个结点二叉树中有 $n+1$ 个空指针。若利用这些空指针存放指向结点在某种遍历次序下的前驱或后继的指针，那么，可以期望提高遍历的效率。

## 6.6 线索二叉树

### 6.6.2 有关概念和术语

- ✿ 线索：所引入的非空指针称为线索。
- ✿ 线索二叉树：加上了线索的二叉树称为线索二叉树。
- ✿ 线索标志位：为了区分一个结点的指针是指向其儿子结点的指针，还是指向其前驱或后继结点的线索，在每个结点中增加2个位—leftThread、rightThread分别称为左、右线索标志位。
- ✿ 线索化：对一棵非线索二叉树以某种次序遍历使其变为一棵线索二叉树的过程称为二叉树的线索化。



## 6.6 线索二叉树

### 6.6.3 线索二叉树结点类型定义

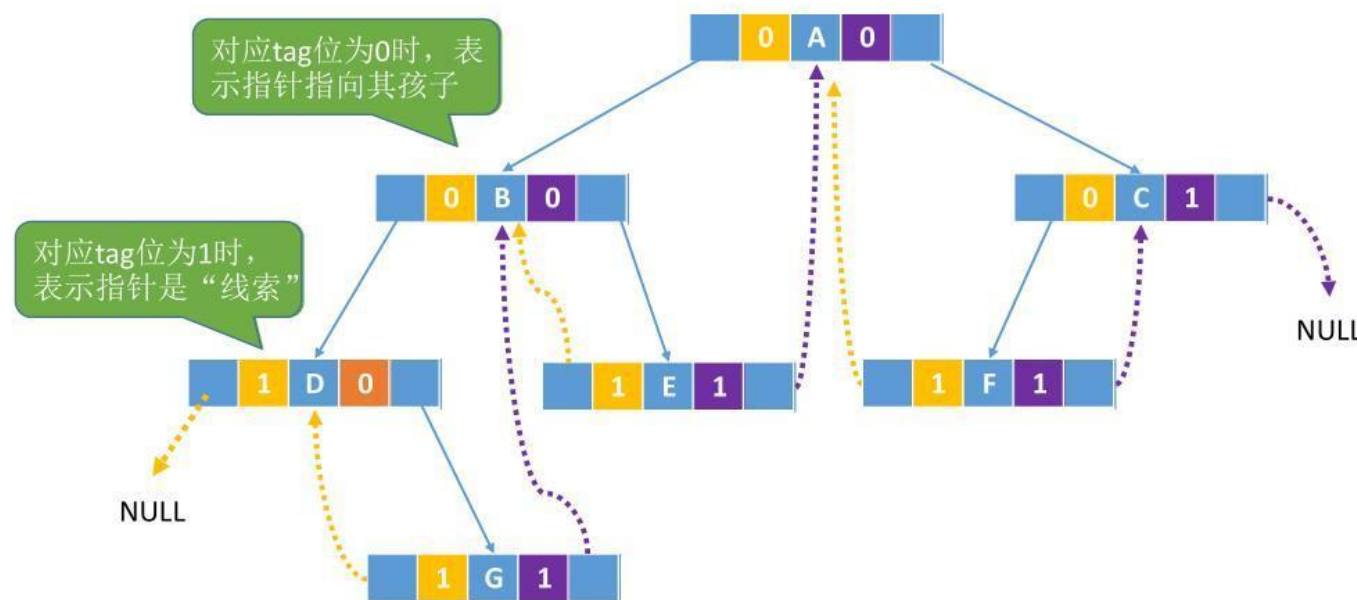
```
typedef struct btnode *tbtlink;
typedef struct tbtnode {
    Treeltem element;
    tbtlink left;          /*左子树 */
    tbtlink right;         /*右子树*/
    int leftThread,        /*左线索标志 */
        rightThread;      /*右线索标志*/
} ThreadedNode ;
```

## 6.6 线索二叉树

### 6.6.4 二叉树线索化：

将二叉树中的空指针改为指向其前驱结点或后继结点的线索(并做上线索标志)

线索化的过程是在对二叉树遍历的过程中修改空指针的过程。

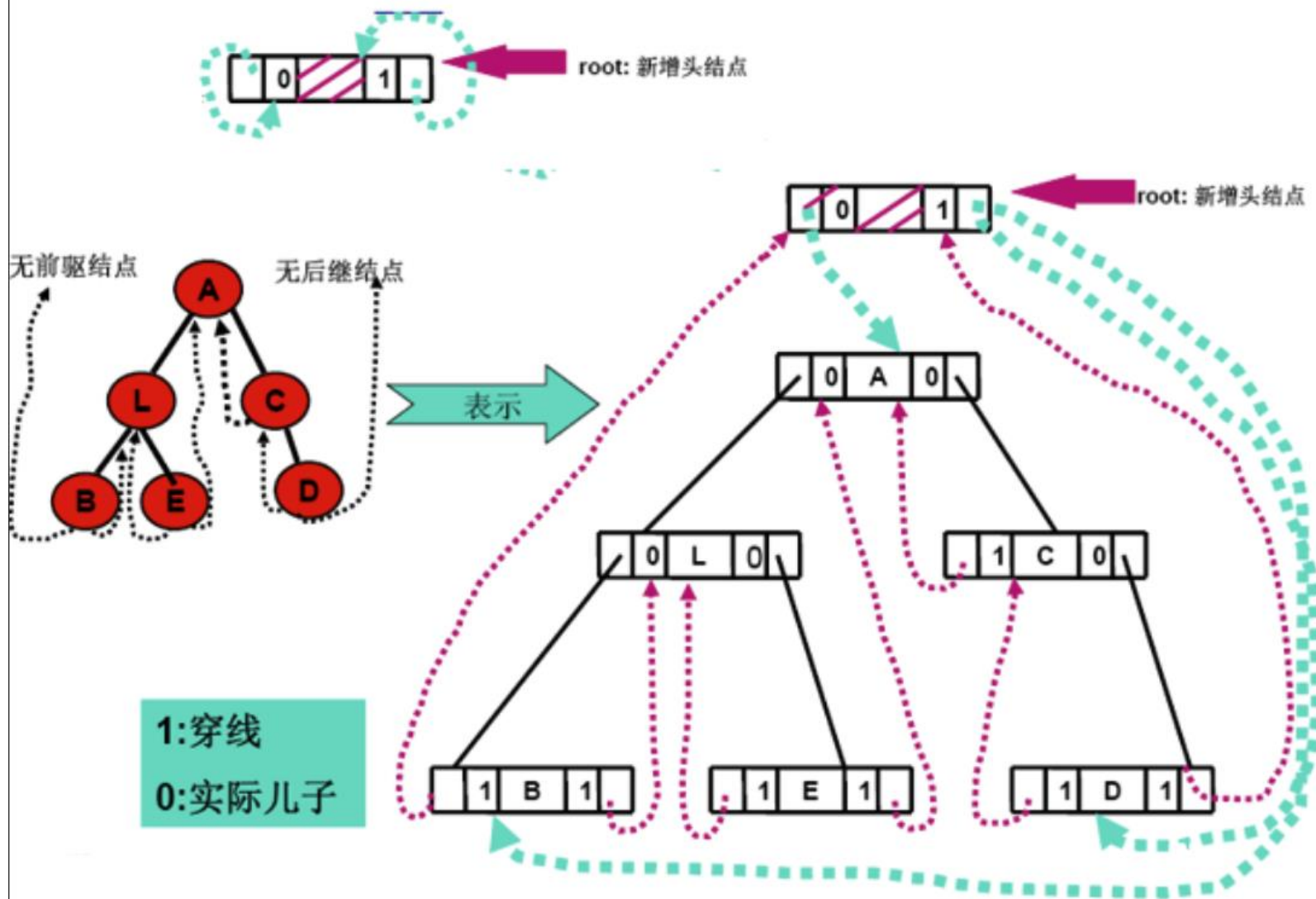


## 6.6 线索二叉树

### 6.6.4 二叉树的中序线索化

增加一个**头结点**，其LeftChild指针指向二叉树的根结点，其RightChild指针指向中序遍历的最后一个结点。而最后一个结点的RightChild指针指向头结点。这样一来，就好象为二叉树建立了一个双向线索链表，既可从  
中序遍历的第一个结点起进行中序的遍历；也可从中序遍历的最后一个结点起进行逆中序的遍历。





## 6.6 线索二叉树

### 6.6.5 线索二叉树与非线索二叉树比较

✚ 优点：

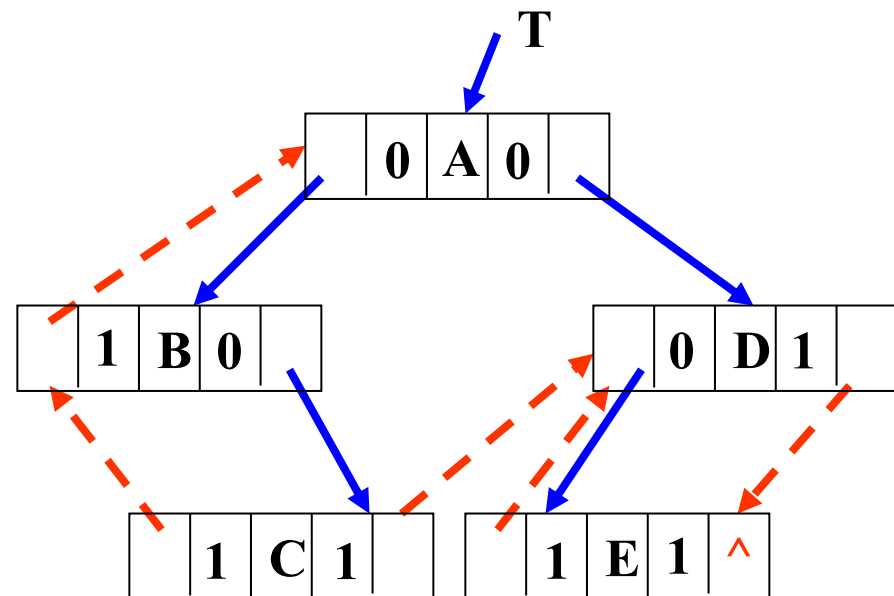
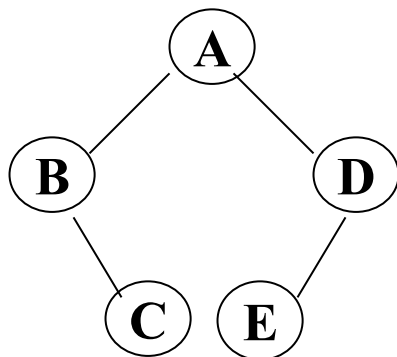
对于找前驱和后继结点2种运算而言，线索二叉树优于非线索二叉树。

✚ 缺点：

在进行结点插入和删除运算时，线索二叉树比非线索二叉树的时间开销大。原因在于在线索二叉树中进行结点插入和删除时，除了修改相应指针外，还要修改相应的线索。

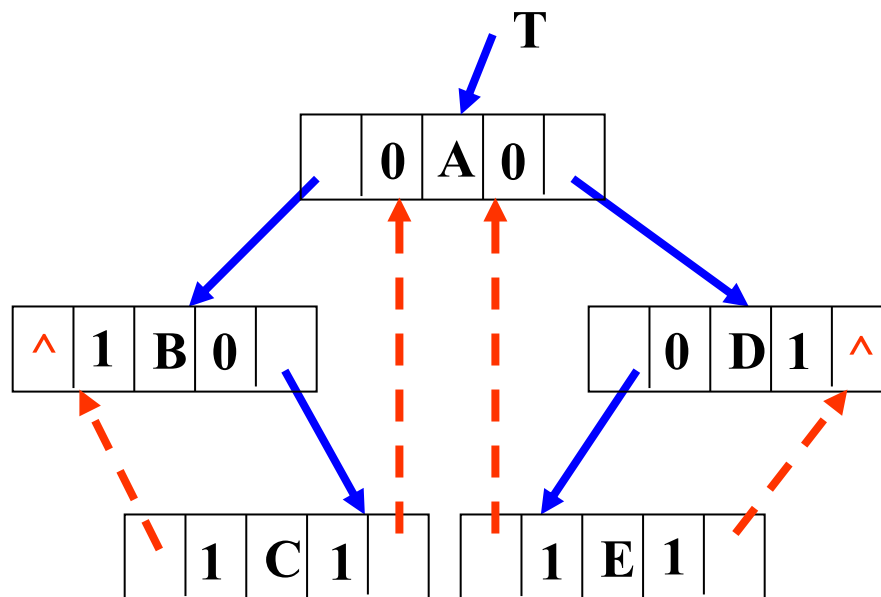
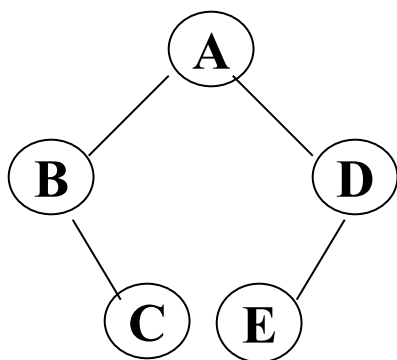


## 前序线索二叉树示例：



前序序列：ABCDE  
前序线索二叉树

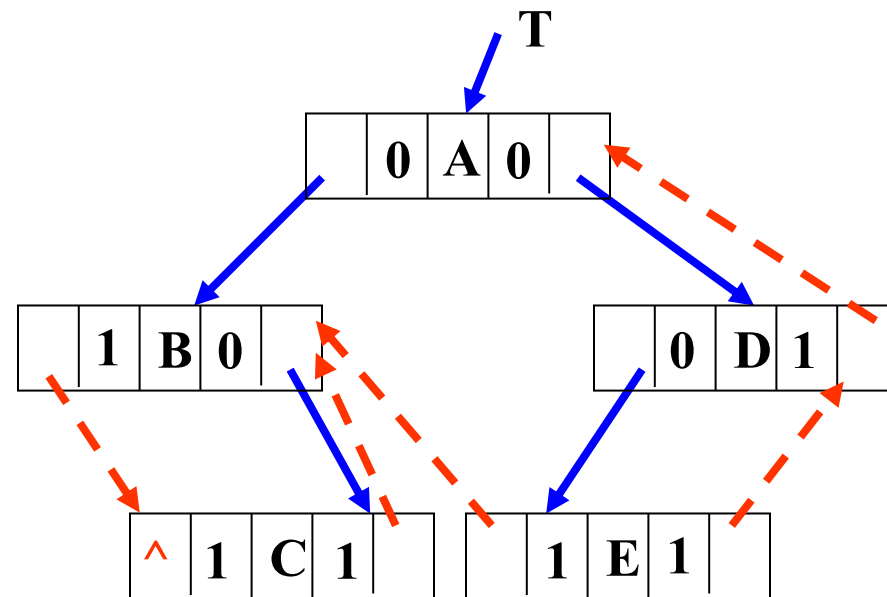
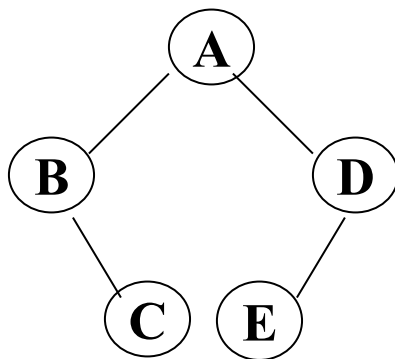
## 中序线索二叉树示例：



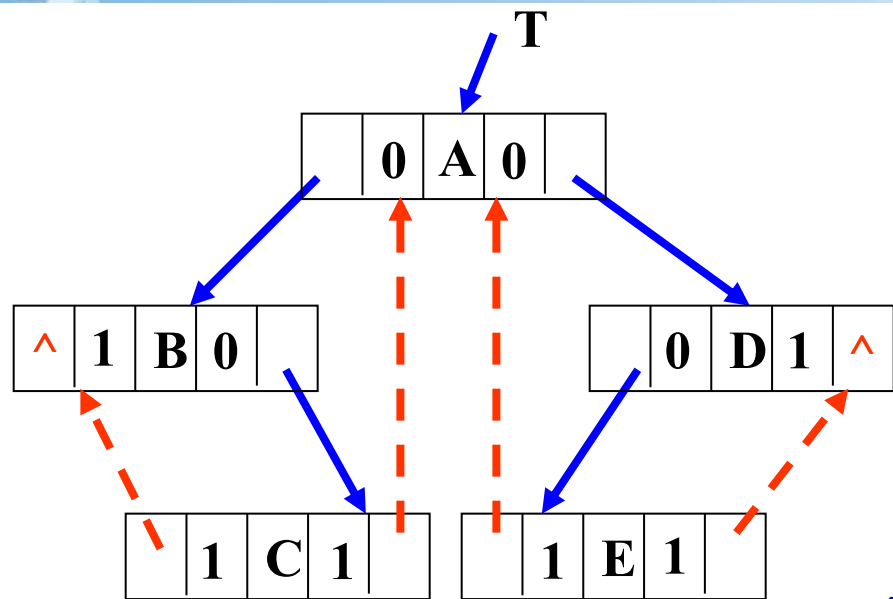
中序序列：BCAED

中序线索二叉树

## 后序线索二叉树示例：



后序序列：CBEDA  
后序线索二叉树



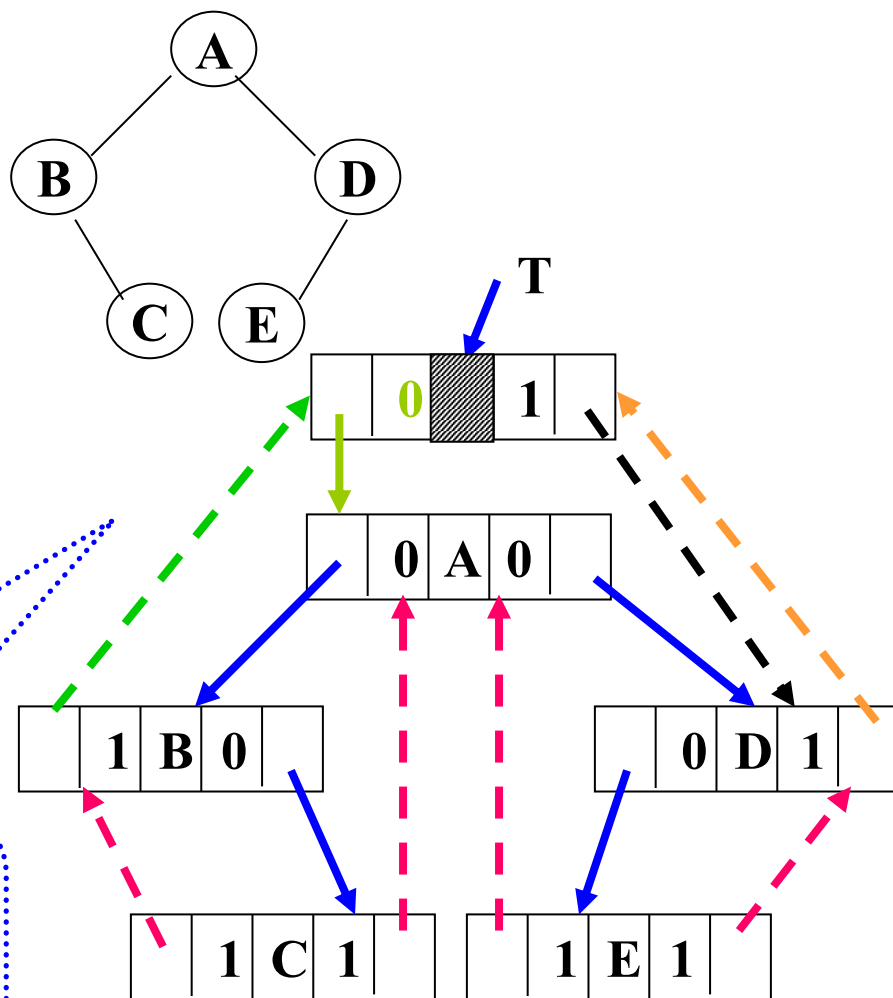
中序序列: BCAED  
中序线索二叉树

头结点:

leftThread=0, left指向根结点

rightThread=1, right指向遍历序列中最后一个结点

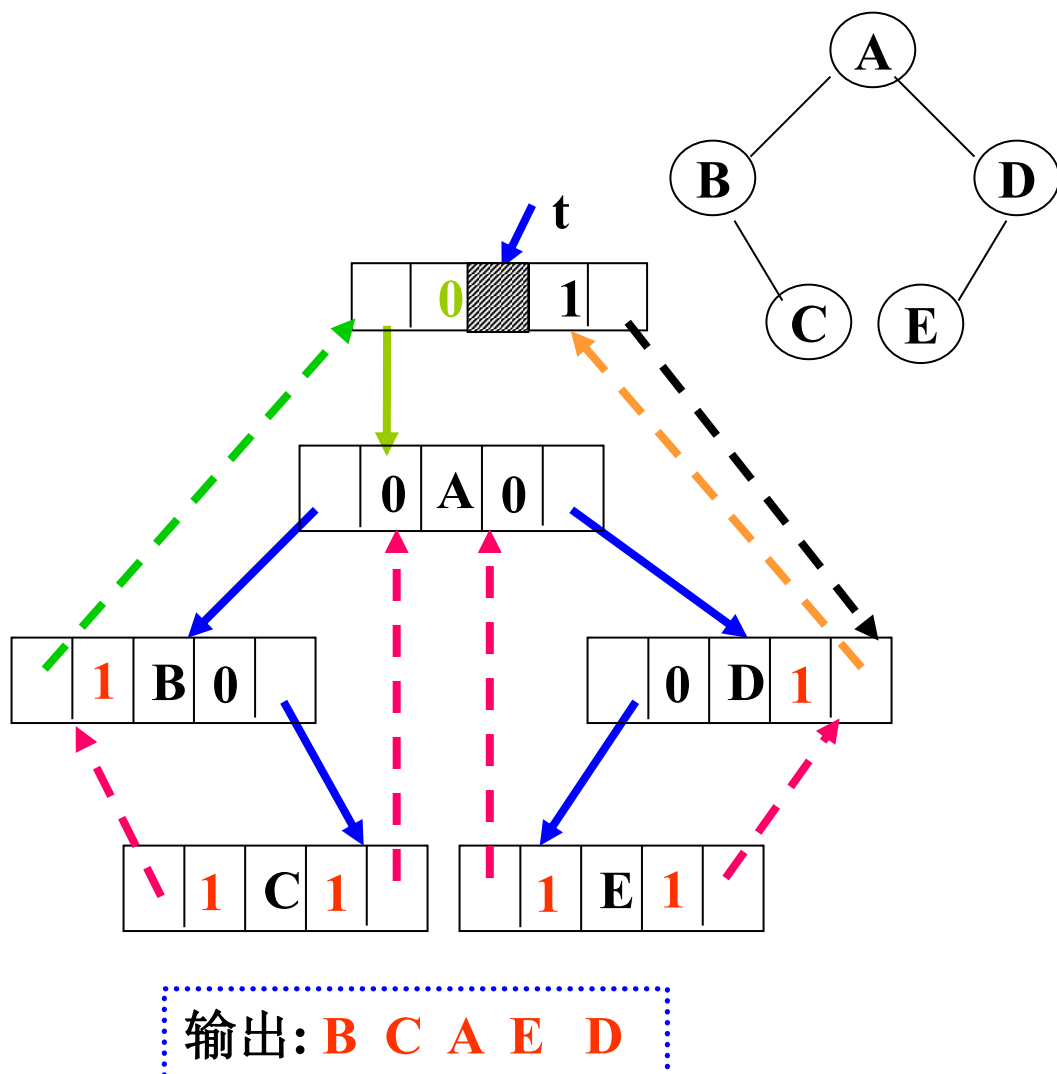
遍历序列中第一个结点的left域和最后一个结点的right域都指向头结点



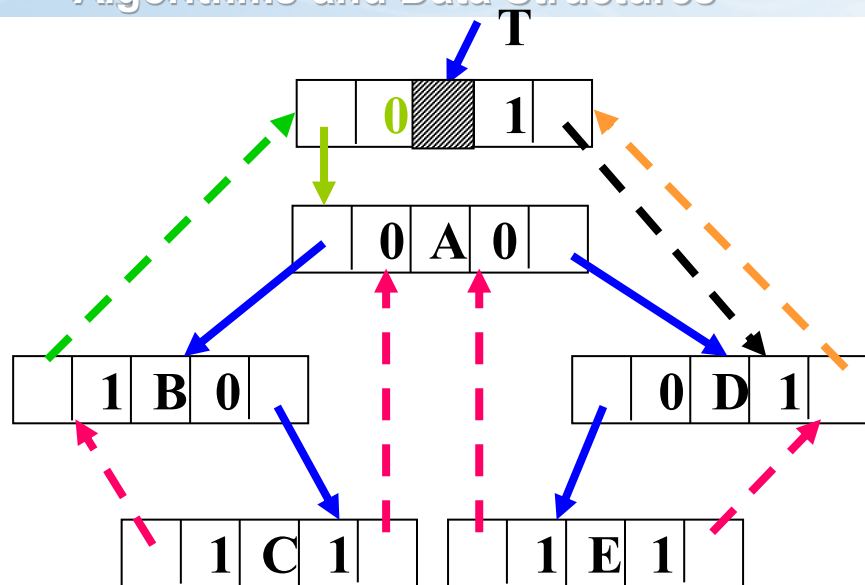
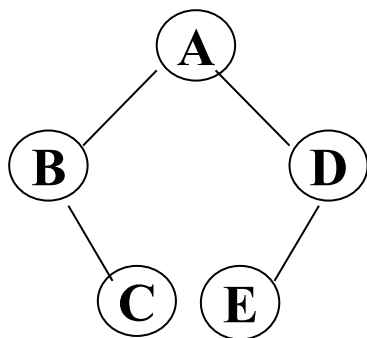
中序序列: BCAED  
带头结点的中序线索二叉树

## 算法

### 按中序线索化二叉树



## 算法—按中序线索化二叉树 遍历中序线索二叉树



中序序列: **BCAED**

带头结点的中序线索二叉树

在中序线索二叉树中找结点后继的方法:

- (1) 若`rightThread=1`, 则`right`域直接指向其后继
- (2) 若`rightThread=0`, 则结点的后继应是其右子树的最左下(`leftThread=1`)的结点

在中序线索二叉树中找结点前驱的方法:

- (1) 若`leftThread=1`, 则`left`域直接指向其前驱
- (2) 若`leftThread=0`, 则结点的前驱应是其左子树的最右下 (`rightThread=1`)的结点

[返回章节目录](#)



## 6.7 树的应用—信号增强装置布局问题

### ★问题描述:

各种资源传输网络的功能是将始发地的资源通过网络传输到一个或多个目的地。例如，通过石油或者天然气输送管网可以将油田开采的石油和天然气传送给消费者。同样，通过高压传输网络可以将发电厂生产的电力传送给用电消费者。为了使问题更具一般性，用术语信号统称网络中传输的资源（石油，天然气，电力等等）。各种资源传输网络统称为信号传输网络。信号经信号传输网络传输时，需要消耗一定的能量，并导致传输能量的衰减（油压，气压，电压等等）。当传输能量衰减量（压降）达到某个阈值时，将导致传输故障。为了保证传输畅通，必需在传输网络的适当位置放置信号增强装置，确保传输能量的衰减量不超过其衰减量容许值。

为了简化问题，假定给定的信号传输网络是以信号始发地为根的一棵树  $T$ 。在树  $T$  的每一个结点处（除根结点外）可以放置一个信号增强装置。树  $T$  的结点也代表传输网络的消费结点。信号经过树  $T$  的结点传输到其儿子结点。树的每一边上的正权是流经该边的信号所发生的信号衰减量。信号衰减量是可加的。

信号增强装置问题要求对于一个给定的信号传输网络，计算如何放置最少的信号增强装置来保证网络传输的畅通。

THE END