

# 第六章 树和二叉树

## 第六章 树和二叉树

- 6.1 树的有关概念
- 6.2 二叉树
- 6.3 二叉树的遍历
- 6.4 遍历的应用
- 6.5 线索二叉树
- 6.6 树和森林
- 6.7 哈夫曼树及应用

## 6.1 树的有关概念

1. 树的概念
2. 树的应用
3. 树的表示
4. 树的有关术语
5. 树的基本操作

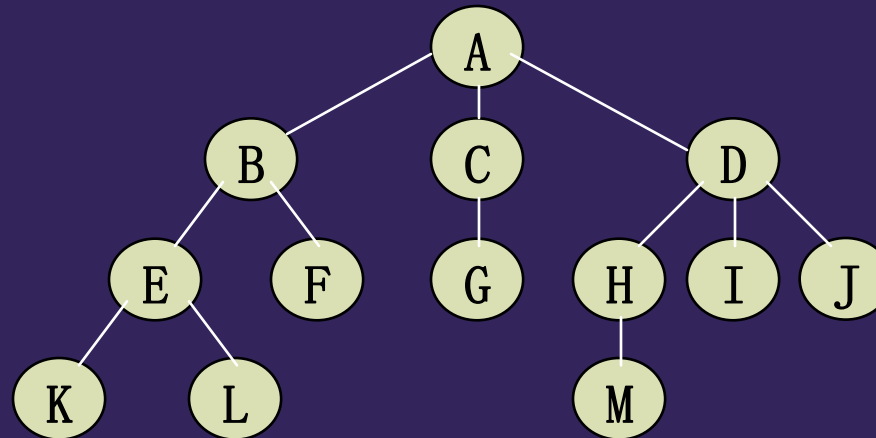
## 6.1 树的有关概念

### 1. 树的概念

树形结构是一种重要的非线性结构，讨论的是层次和分支关系。

树是 $n$ 个结点的有限集合，在任一非空树中：

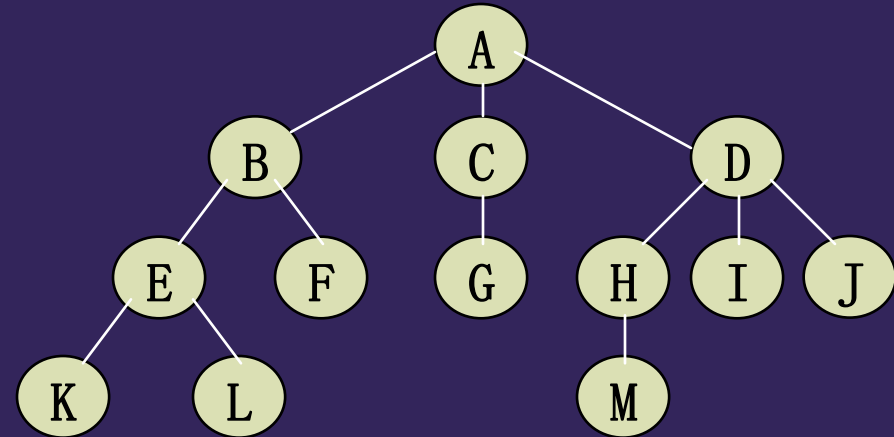
- (1) 有且仅有一个称为根的结点。
- (2) 其余结点可分为个互不相交的集合，而且这些集合中的每一集合都本身又是一棵树，称为根的子树。



树是递归结构，在树的定义中又用到了树的概念

## 6.1 树的有关概念

例：下面的图是一棵树



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

A是根，其余结点可以划分为3个互不相交的集合：

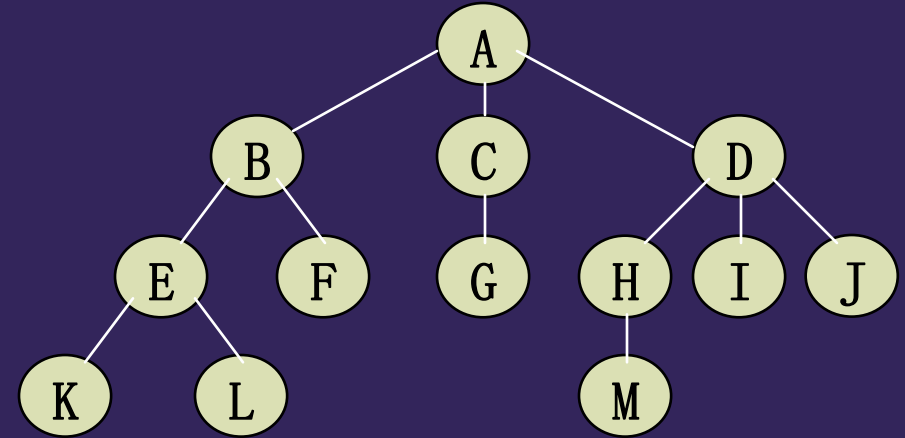
$T_1 = \{B, E, F, K, L\}$  ,  $T_2 = \{C, G\}$  ,  $T_3 = \{D, H, I, J, M\}$

这些集合中的每一集合都本身又是一棵树，它们是A的子树。

例如 对于  $T_1$ ，B是根，其余结点可以划分为2个互不相交的集合：

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

## 6.1 树的有关概念



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) 除根外的其他结点，都存在唯一一条从根到该结点的路径；
- 5) 树是一种分枝结构（除了一个称为根的结点外）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。

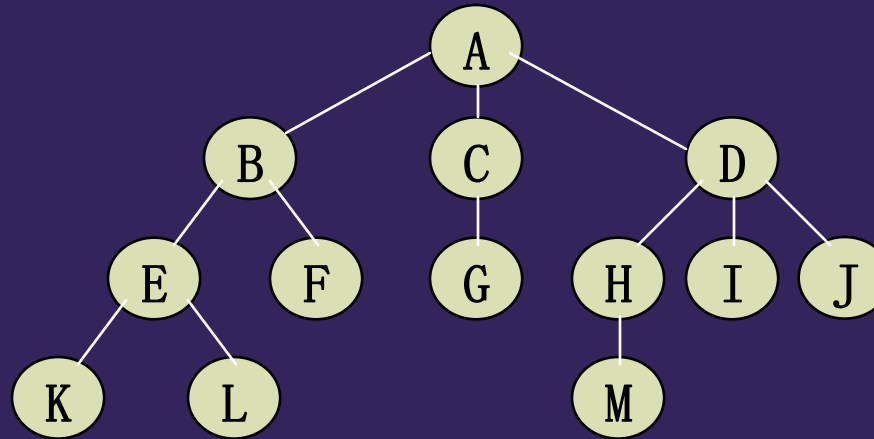
# 6.1 树的有关概念

## 2. 树的应用

### 1) 树可表示具有分枝结构关系的对象

#### 例1. 家族族谱

设某家庭有13个成员A、B、C、D、E、F、G、H、I、J、K、L、M  
他们之间的关系可下图所示的树表示：



#### 例2. 单位行政机构的组织关系

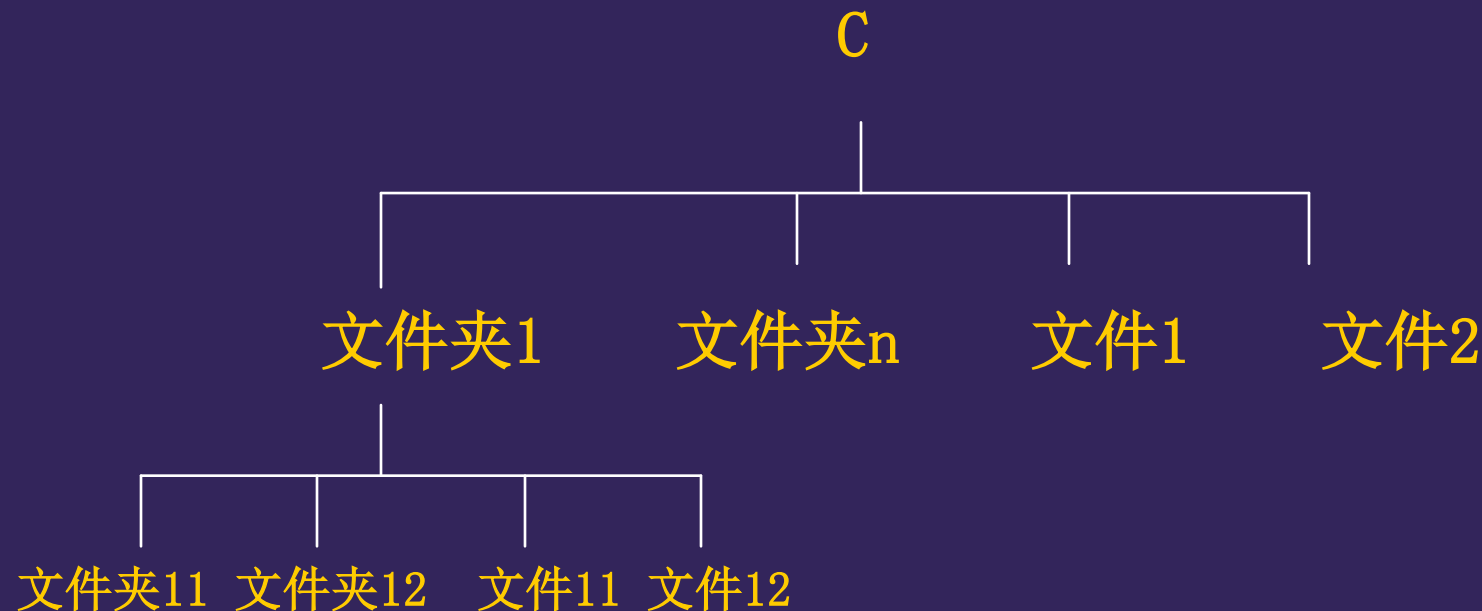
## 6.1 树的有关概念

### 2) 树是常用的数据组织形式

有些应用中数据元素之间并不存在间分支结构关系，但是为了便于管理和使用数据，将它们用树的形式来组织。

### 例3 计算机的文件系统

不论是DOS文件系统还是window文件系统，所有的文件是用树的形式来组织的。





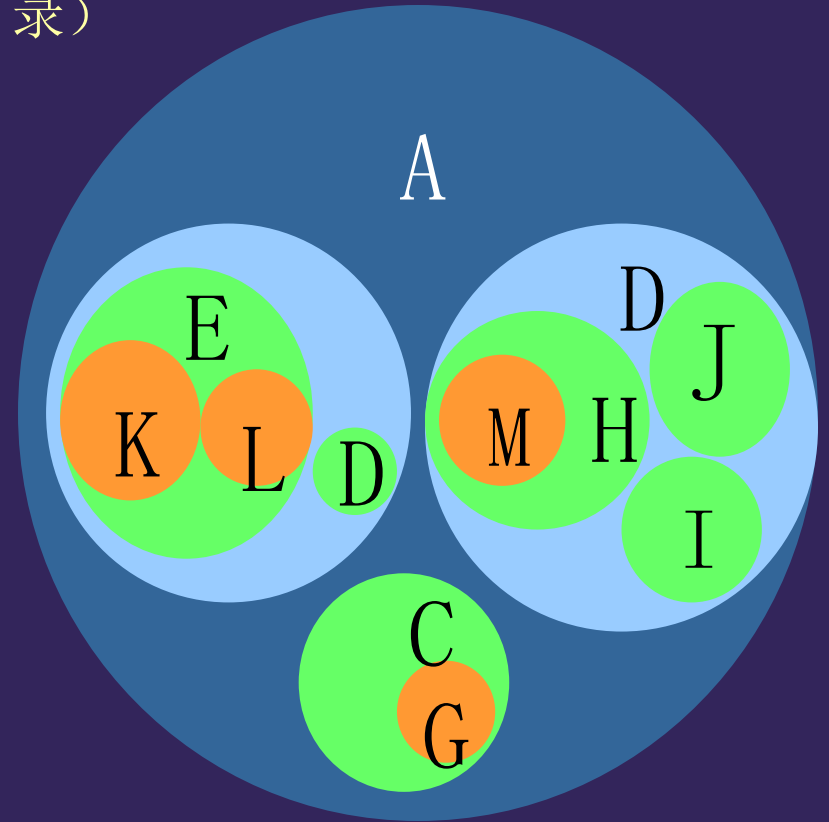
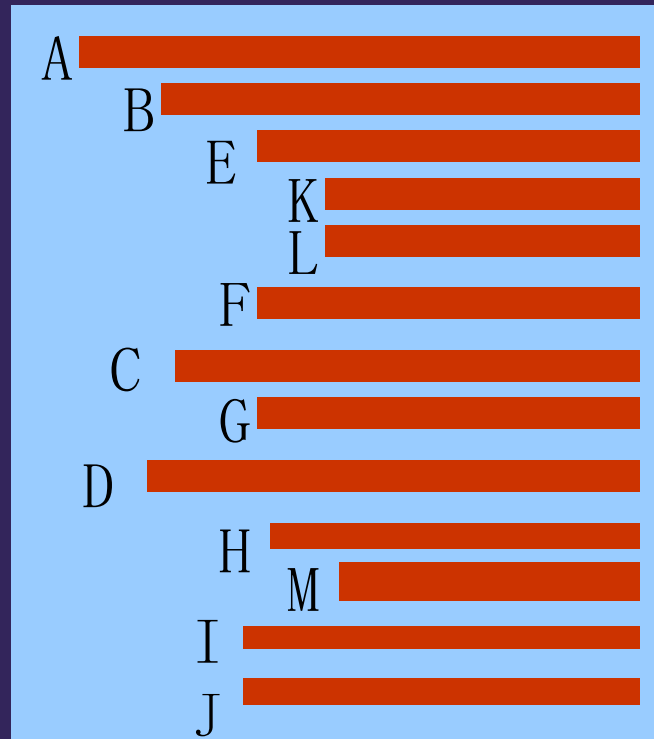
## 6.1 树的有关概念

### 3 树的表示

1) 图示表示 2) 二元组表示

3) 嵌套集合表示 4) 凹入表示法 (类似书的目录)

## 5) 广义表表示


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

## 6.1 树的有关概念

### 4 树的基本术语

**树的结点：**包含一个数据元素及若干指

向子树的分支；

**孩子结点：**结点的子树的根称为该结点的孩子；

**双亲结点：**B 结点是A 结点的孩子，则A

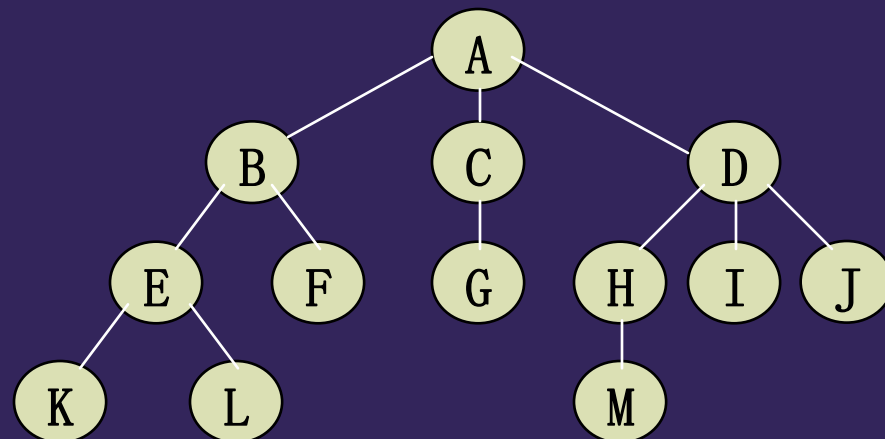
结点是B 结点的双亲；

**兄弟结点：**同一双亲的孩子结点；

**堂兄结点：**同一层上结点；

**祖先结点：**从根到该结点的所经分支上的所有结点

**子孙结点：**以某结点为根的子树中任一结点都称为该结点的子孙



# 6.1 树的有关概念

## 4 树的基本术语

**结点层：**根结点的层定义为1；根的孩子为第二层结点，依此类推；

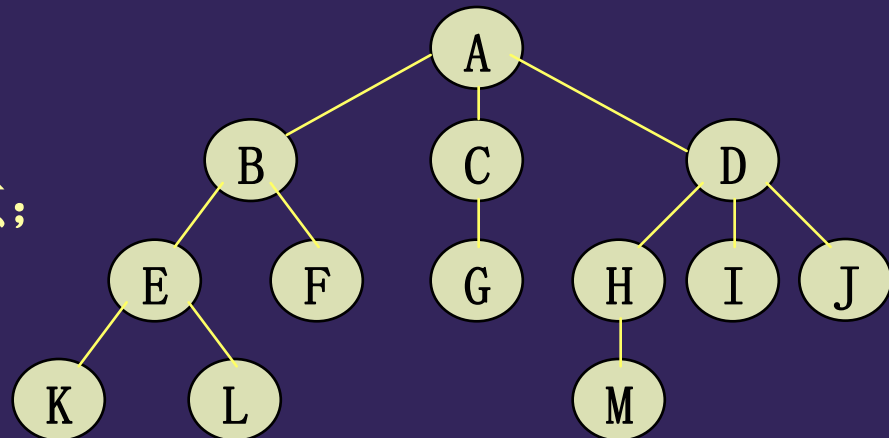
**树的深度：**树中最大的结点层

**结点的度：**结点子树的个数

**树的度：**树中最大的结点度。

**叶子结点：**也叫终端结点，是度为0的结点；

**分枝结点：**度不为0的结点；



**有序树：**子树有序的树，如：家族树；

**无序树：**不考虑子树的顺序；

**森林：**互不相交的树集合；森林和树之间的联系是：一棵树去掉根，其子树构成一个森林；一个森林增加一个根结点成为树。

## 6.1 树的有关概念

### 5 树的基本操作

树的应用很广，应用不同基本操作也不同。下面列举了树的一些基本操作：

- 1) **initiate (T)**; T 树的初始化，包括建树。
- 2) **root (T)**; 求T 树的根。
- 3) **parent (T, x)**: 求T 树中 x 结点的双亲结点。
- 4) **Child (T, x, i)**: 求 T 树中 x 结点的第 i 个孩子结点。
- 5) **right\_sibling (T, x)**: 求T 树中 x 结点的右兄弟
- 6) **insert\_Child (y, i, x)**: 将根为 x 的子树置为 y 结点的第 i 个孩子
- 7) **del\_child (x, i)**; 删除 x 结点的第i 个孩子
- 8) **traverse (T)**; 遍历T树。按某个次序依次访问树中每一个结点，并使每个结点都被 访问且只被访问一次。
- 9) **clear (T)**; 置空T 树

树是一种分枝结构的对象，在树的概念中，对每一个结点孩子的个数没有限制，因此树的形态多种多样，本章我们主要讨论一种最简单的树——二叉树。

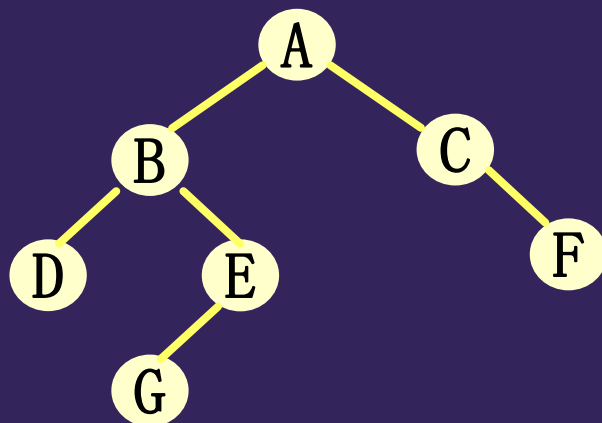
## 6.2 二叉树

- 一 二叉树的概念
- 二 二叉树的性质
- 三 二叉树的存储结构

### 一 二叉树的概念

#### 1 二叉树的定义

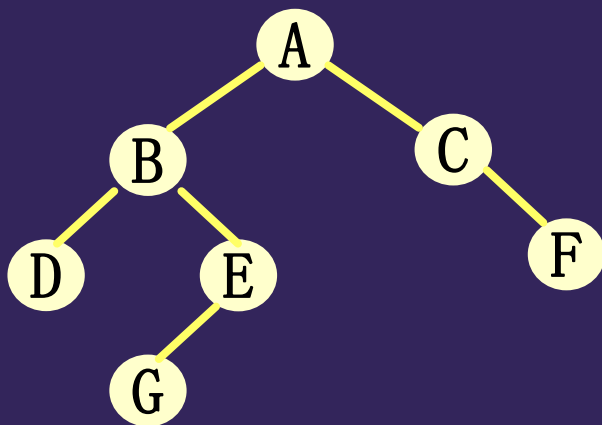
二叉树： 或为空树，或由根及两颗不相交的左子树、右子树构成，并且左、右子树本身也是二叉树。



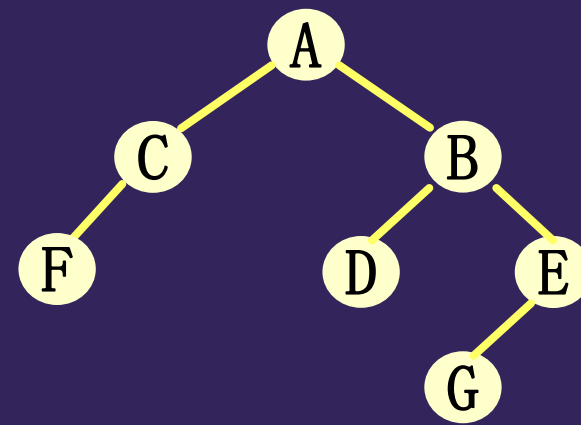
#### 说明

- 1) 二叉树中每个结点最多有两颗子树；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；
- 3) 二叉树是递归结构，在二叉树的定义中又用到了二叉树的概念；

## 6.2 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，



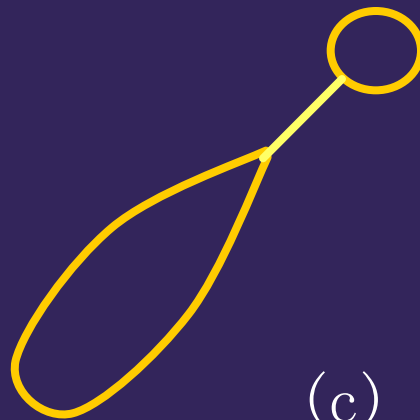
### 2. 二叉树的基本形态

$\varnothing$

(a) 空树



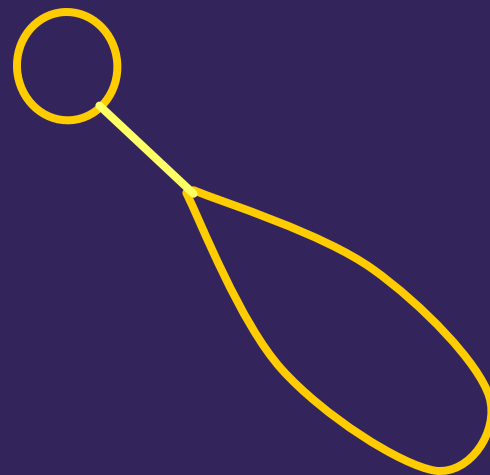
(b) 仅有根



(c) 右子树空



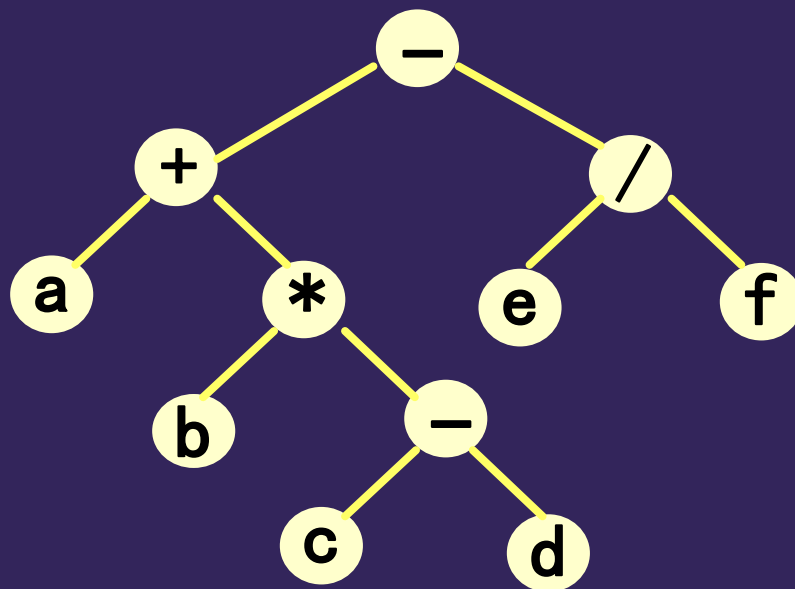
(d) 左、右子树均在



(e) 左子树空

### 3. 应用举例

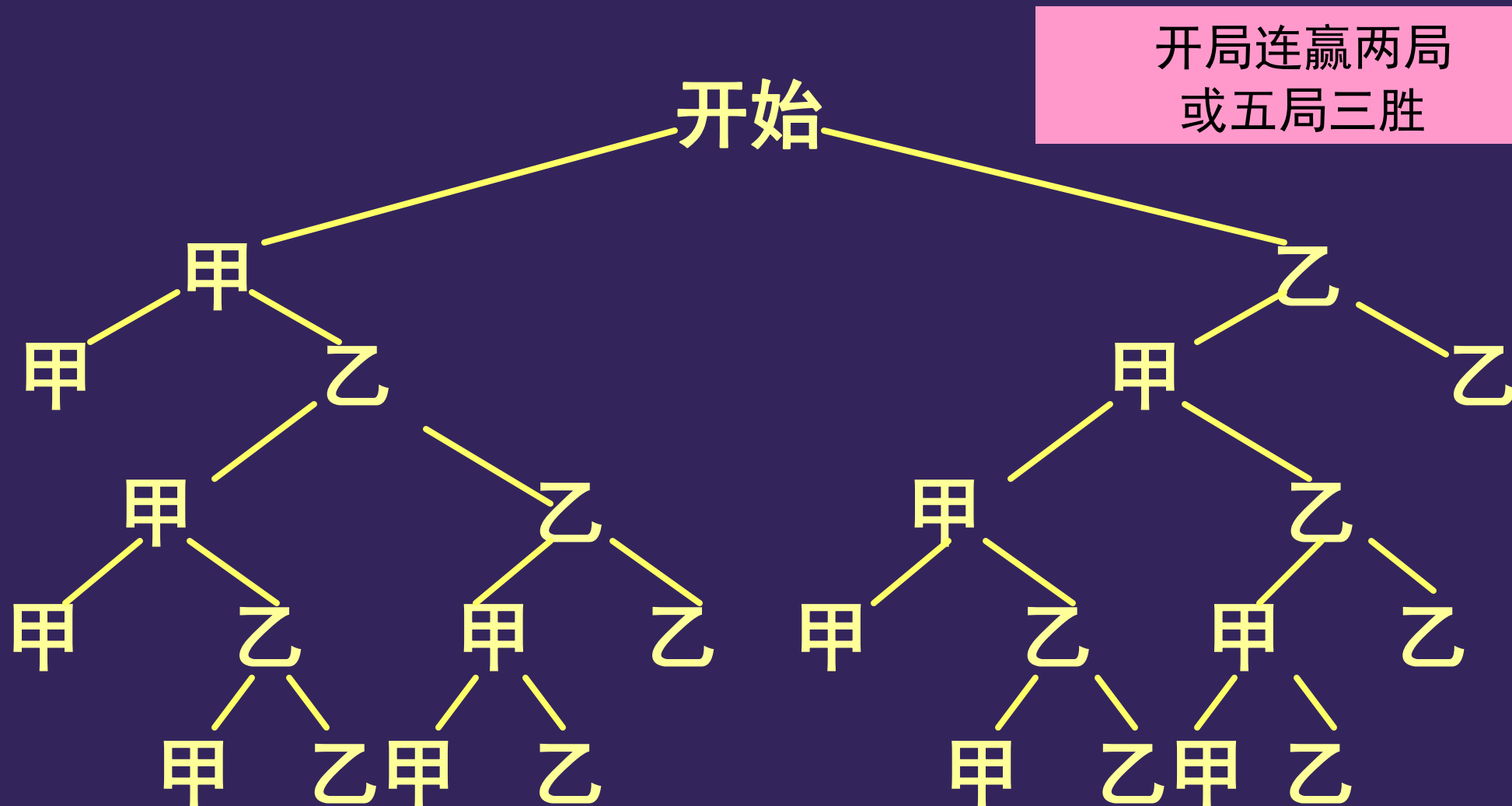
例1 可以用二叉树表示表达式



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

## 6.2 二叉树

## 例2 双人比赛的所有可能的结局



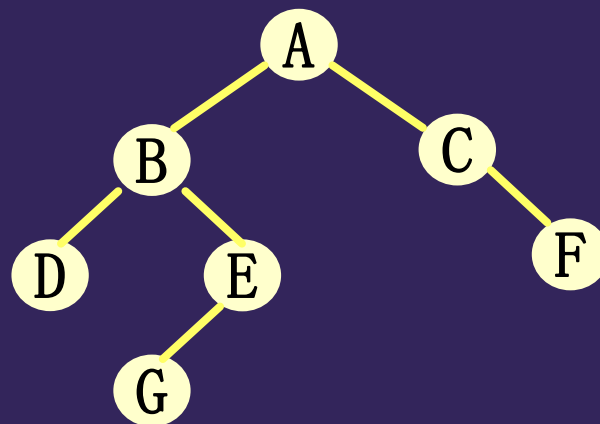
## 6.2 二叉树

### 二 二叉树性质

性质1 在二叉树的第 $i$ 层上最多有 $2^{i-1}$ 个结点

性质2 深度为 $k$ 的二叉树最多有  $2^k-1$  个结点

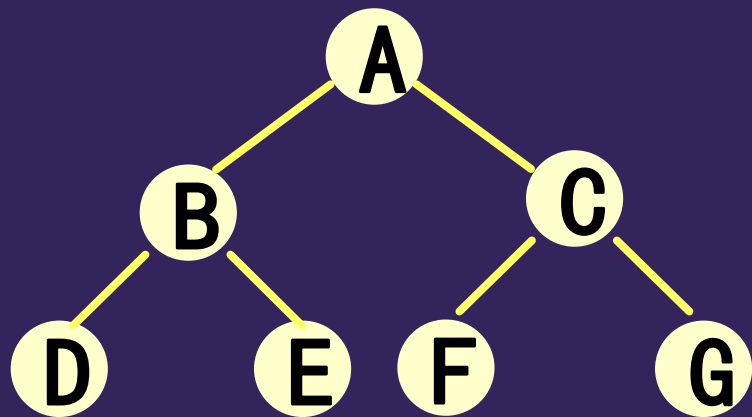
性质3 设二叉树叶子结点数为 $n_0$ ，度为2的结点 $n_2$ ，则 $n_0 = n_2 + 1$



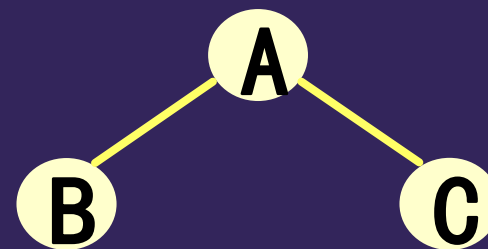
## 6.2 二叉树

### 两种特殊的二叉树

**满二叉树：** 如果深度为 $k$ 的二叉树，有 $2^k-1$ 个结点则称为满二叉树；



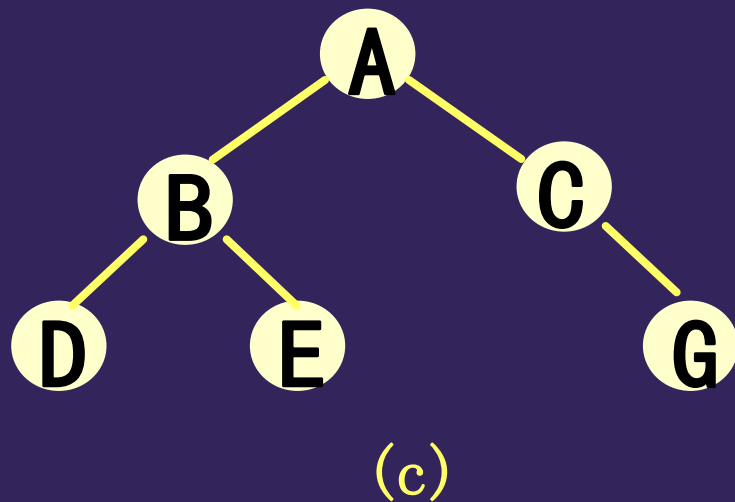
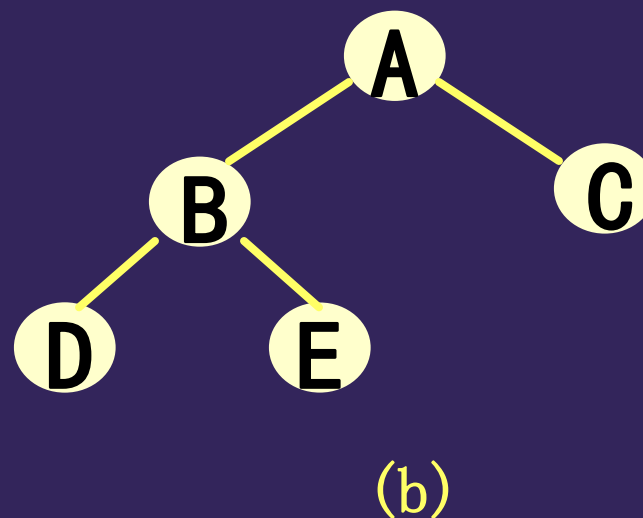
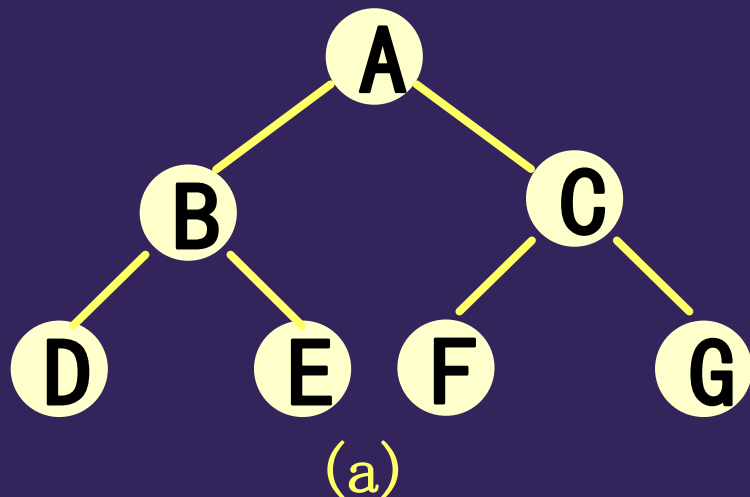
K=3的满二叉树



K=2的满二叉树

## 6.2 二叉树

**完全二叉树：**如果一颗二叉树只有最下一层结点数可能未达到最大，并且最下层结点都集中在该层的最左端，则称为完全二叉树；



(a)、(b) 完全二叉树  
(c) 不是完全二叉树

## 6.2 二叉树

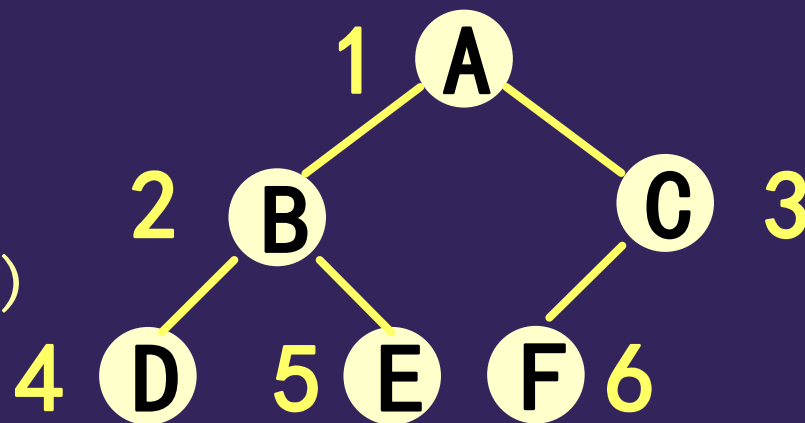
下面是两个关于完全二叉树的性质

**性质4** 具有 $n$ 个结点的完全二叉树的深度为： $\text{trunc}(\log_2 n) + 1$ .  
 $\text{trunc}(x)$  为取整函数。

**对完全二叉树的结点编号：**从上到下，每一层从左到右

**性质5：**在完全二叉树中编号为 $i$ 的结点

- 1) 若有左孩子，则左孩编号为 $2i$
- 2) 若有右孩子，则右孩子结点编号为 $2i+1$
- 3) 若有双亲，则双亲结点编号为 $\text{trunc}(i/2)$



## 6.2 二叉树

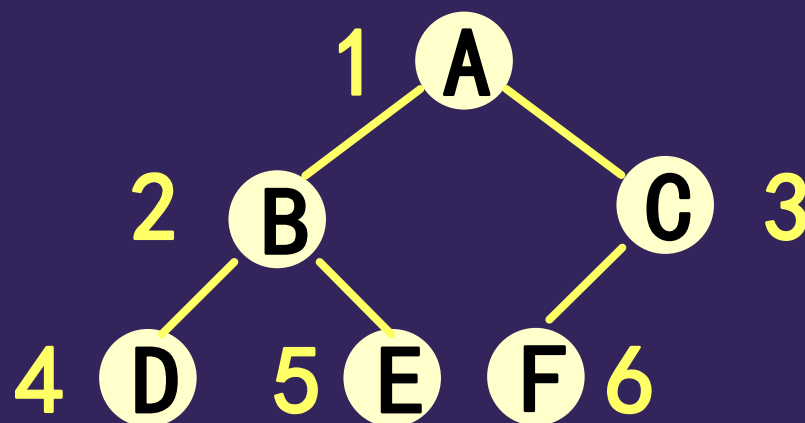
### 三. 二叉树存贮结构

#### 1 二叉树的顺序结构

满二叉树或完全二叉树的顺序结构

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元素. 例如, 用一维数组  $bt[ ]$  存放一棵完全二叉树, 将标号

为  $i$  的结点的数据元素存放在分量  $bt[i-1]$  中。存储位置隐含了树中的关系, 树中的关系是通过完全二叉树的性质实现的。例如,  $bt[5]$  ( $i=6$ ) 的双亲结点标号是  $k=\text{trunc}(i/2)=3$ , 双亲结点所对应的数组分量  $bt[k-1]=bt[2]$



顺序结构图示

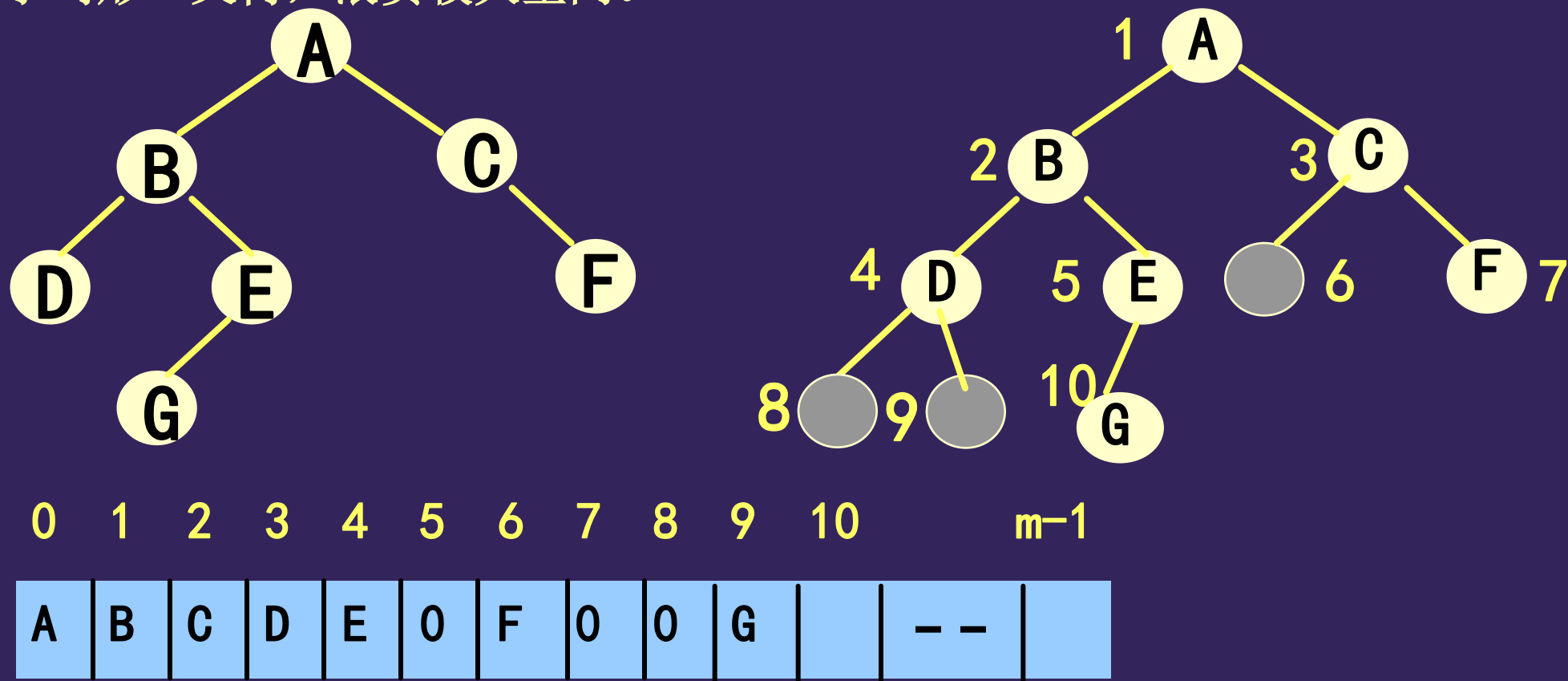




## 6.2 二叉树

### 非完全二叉树的顺序结构

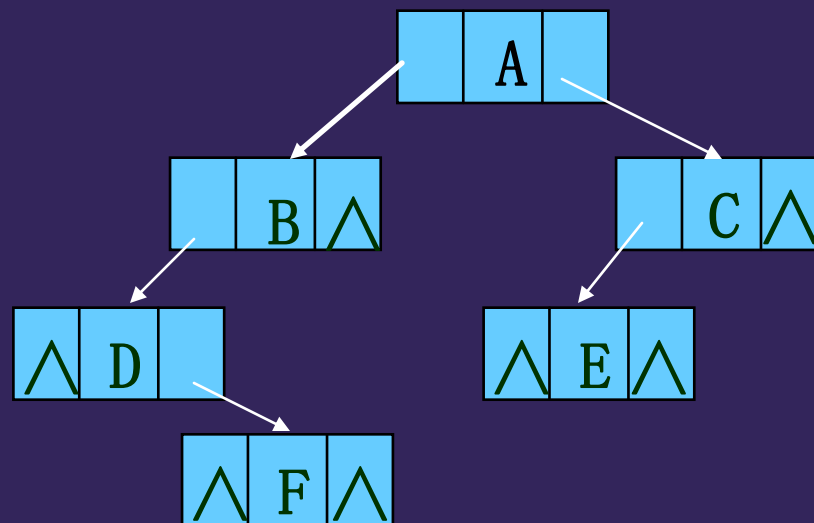
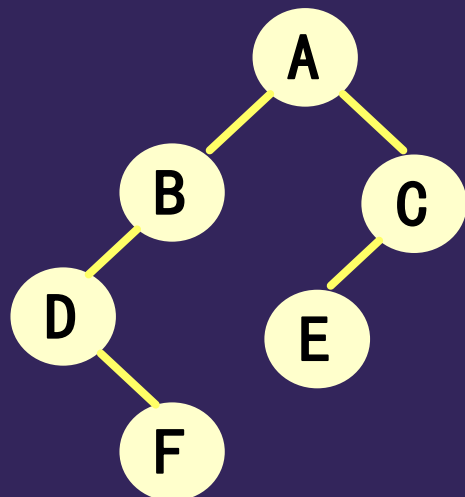
按完全二叉树的形式补齐二叉树所缺少的那些结点，对二叉树结点编号，将二叉树原有的结点按编号存储到内存单元“相应”的位置上。但这种方式对于畸形二叉树，浪费较大空间。



## 6.2 二叉树

### 2 二叉链表

域  
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



二叉链表图示

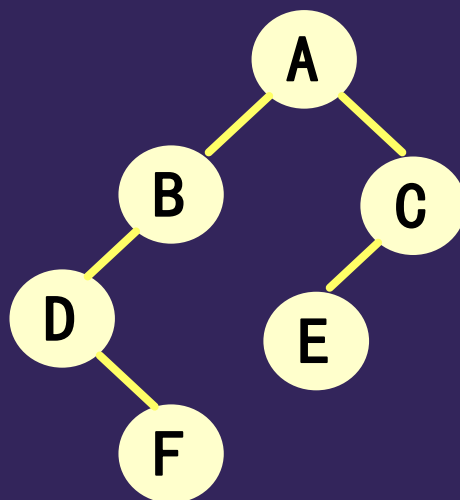
Struct node

```
{ int data;  
  struct node *lch,*rch;  
};
```

## 6.2 二叉树

### 3 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



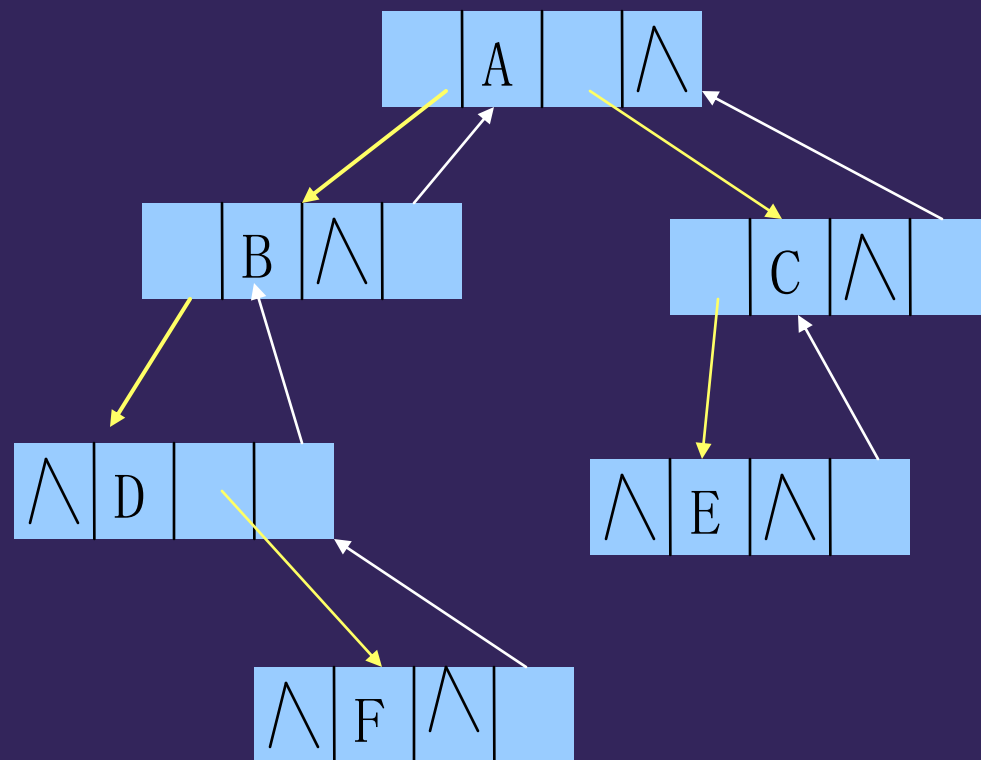
Struct node

```
{ int data;
```

```
  struct node
```

```
*lch,*rch,*parent;
```

```
};
```



## 6.3 二叉树的遍历

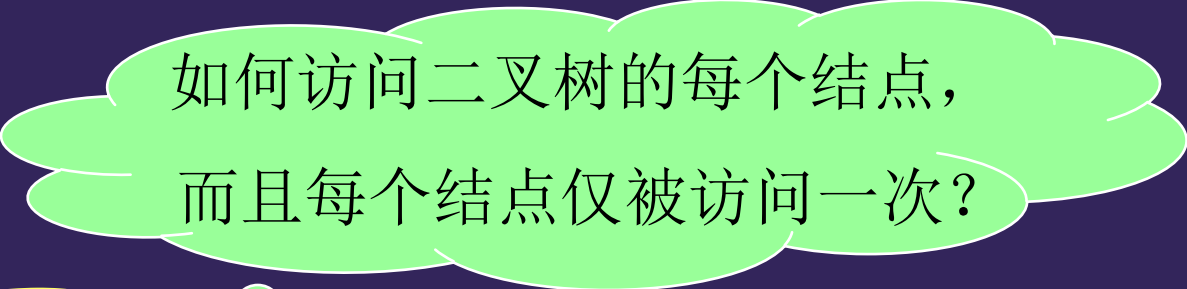
- 一. 二叉树的遍历方法
- 二. 遍历的递归算法
- 三. 遍历的非递归算法

## 6.3 二叉树的遍历

**遍历：**按某种搜索路径访问二叉树的每个结点，而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。

遍历是各种数据结构最基本的操作，许多其他的操作可以在遍历基础上实现。



如何访问二叉树的每个结点，  
而且每个结点仅被访问一次？



## 6.3 二叉树的遍历

### 一 二叉树的遍历方法

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

令：L：遍历左子树

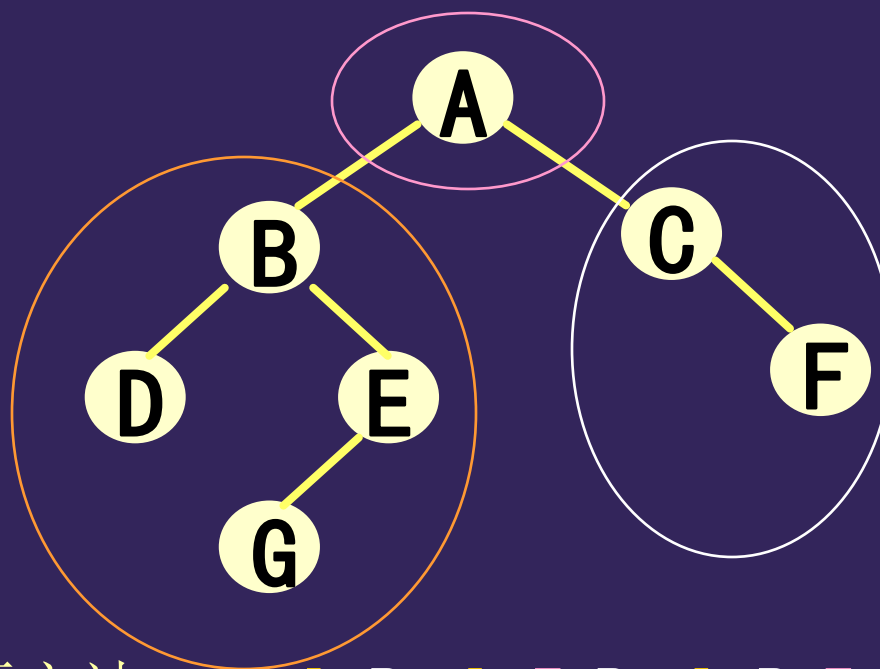
T：访问根结点

R：遍历右子树

有六种遍历方法：

T L R, L T R, L R T,

T R L, R T L, R L T



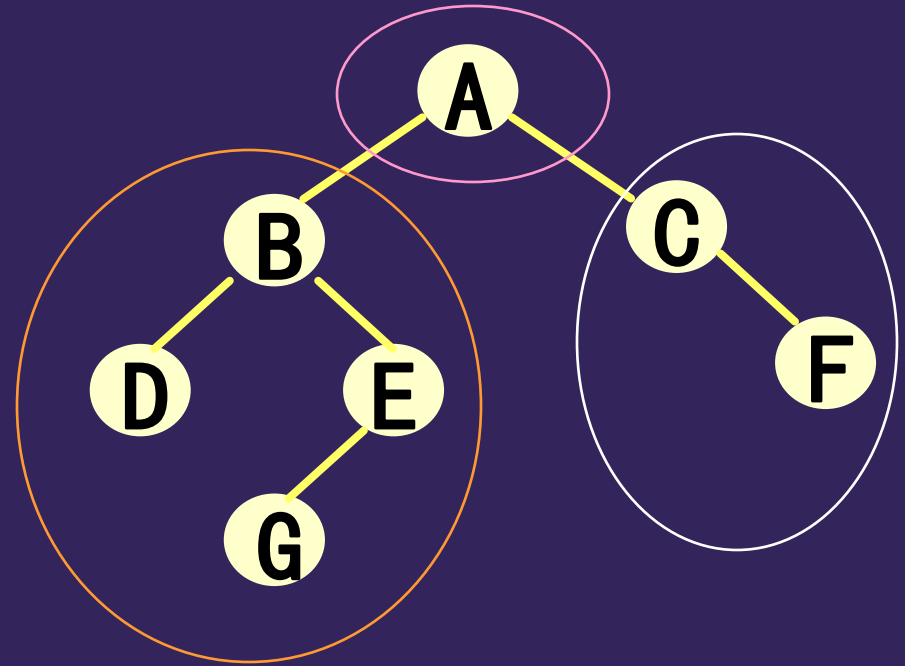
约定先左后右, 有三种遍历方法：T L R、L T R、L R T，分别称为先序遍历、中序遍历、后序遍历

## 6.3 二叉树的遍历

先序遍历 (T L R)

若二叉树非空

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



例：先序遍历右图所示的二叉树

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

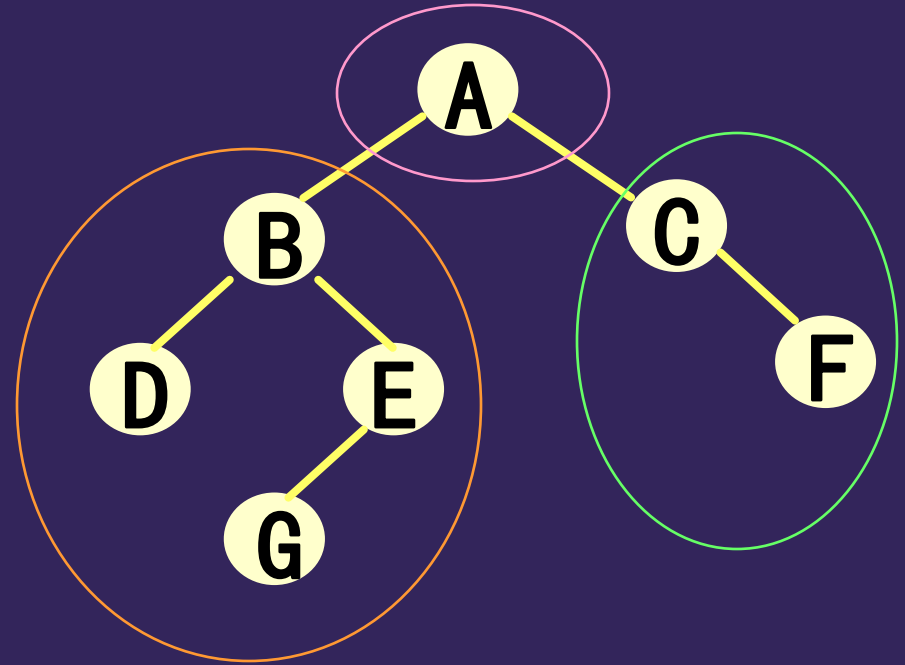
先序遍历序列：A, B, D, E, G, C, F

## 6.3 二叉树的遍历

### 中序遍历 (L T R)

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：中序遍历右图所示的二叉树

- (1) 中序遍历左子树：即按 L T R 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 L T R 的顺序遍历右子树

中序遍历序列： D, B, G, E, A, C, F

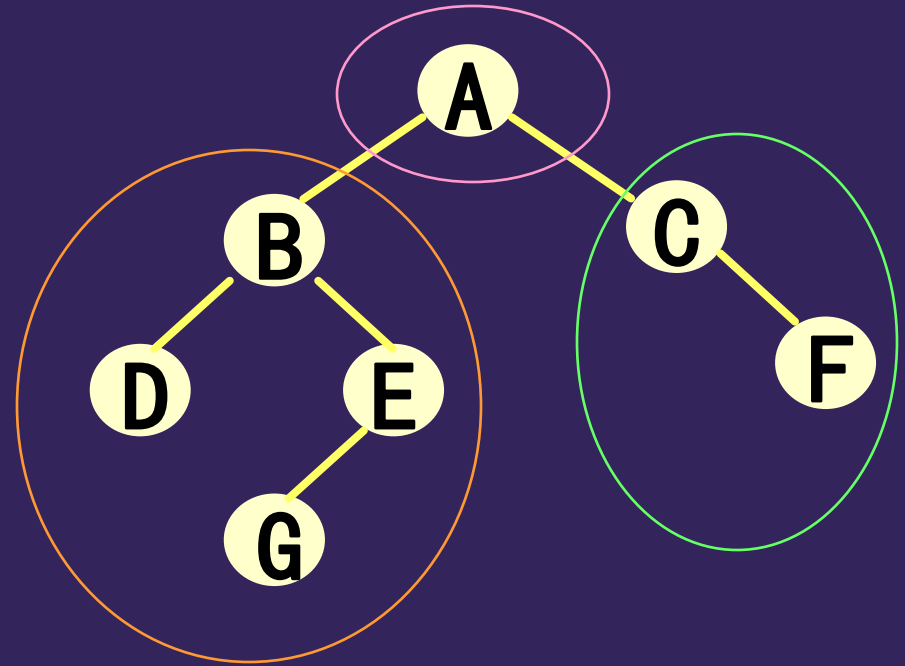


## 6.3 二叉树的遍历

后序遍历 (L R T)

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



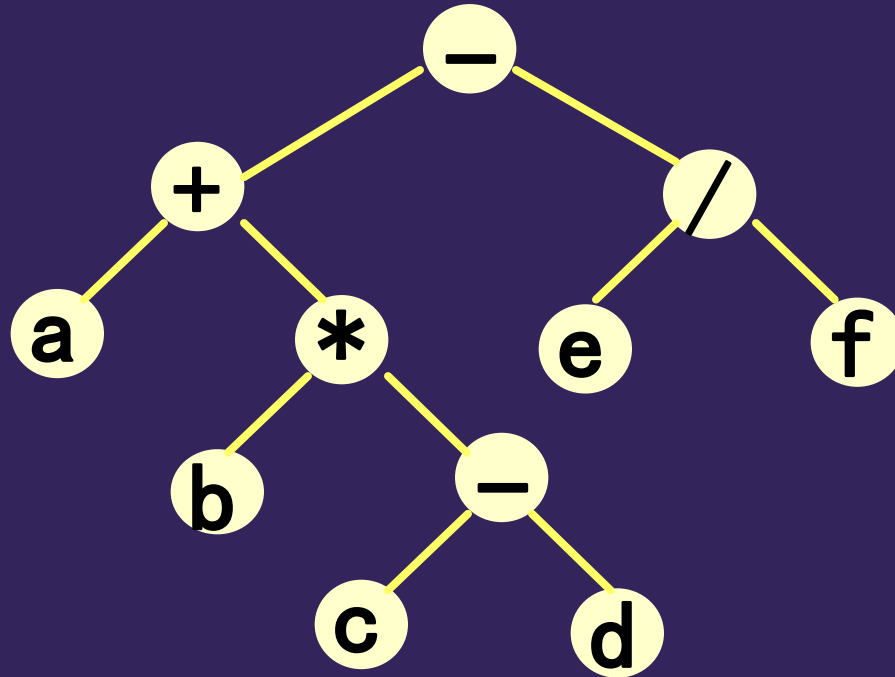
例：后序遍历右图所示的二叉树

- (1) 后序遍历左子树：即按 L R T 的顺序遍历左子树
- (2) 后序遍历右子树：即按 L R T 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

## 6.3 二叉树的遍历

例：先序遍历、中序遍历、后序遍历下图所示的二叉树



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： $a$ ,  $b$ ,  $c$ ,  $d$ ,  $-$ ,  $*$ ,  $+$ ,  $e$ ,  $f$ ,  $/$ ,  $-$

## 6.3 二叉树的遍历

### 二. 遍历的递归算法

先序遍历（**T L R**）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；

## 6.3 二叉树的遍历

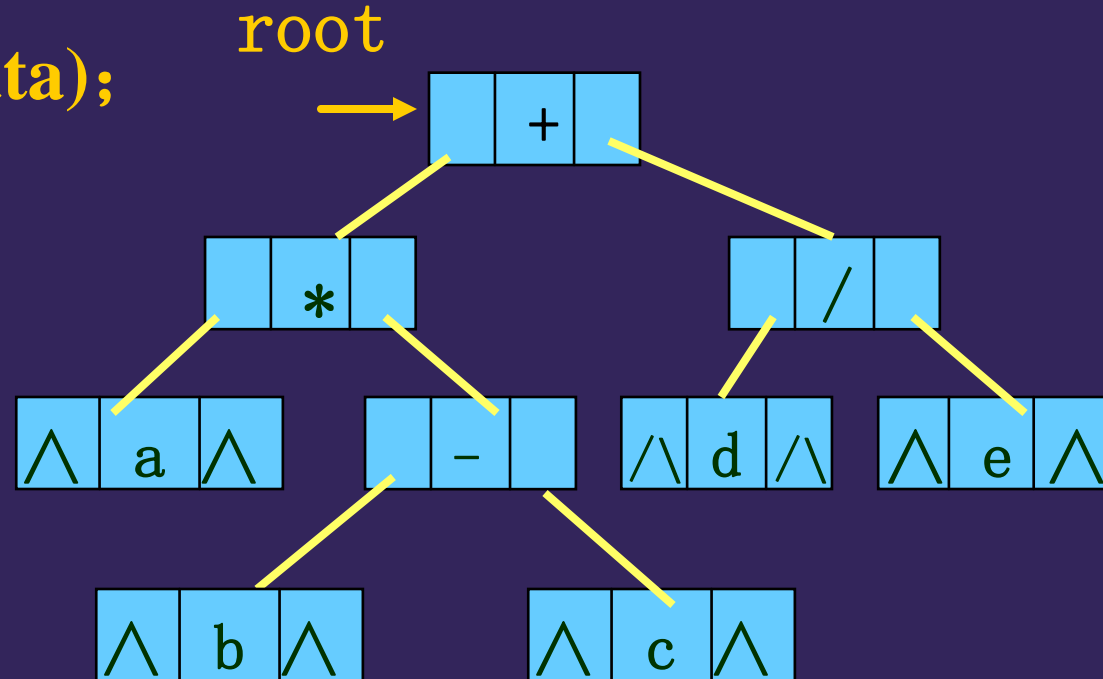
### 1 先序遍历递归算法

```
void prev (NODE *root)
{ if (root!=NULL)
  { printf("%d,", root->data);
    prev(root->lch);
    prev(root->rch);
  }
}
```

先序序列为

+ \* a - b c / d e

称为前缀表达式



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

## 6.3 二叉树的遍历

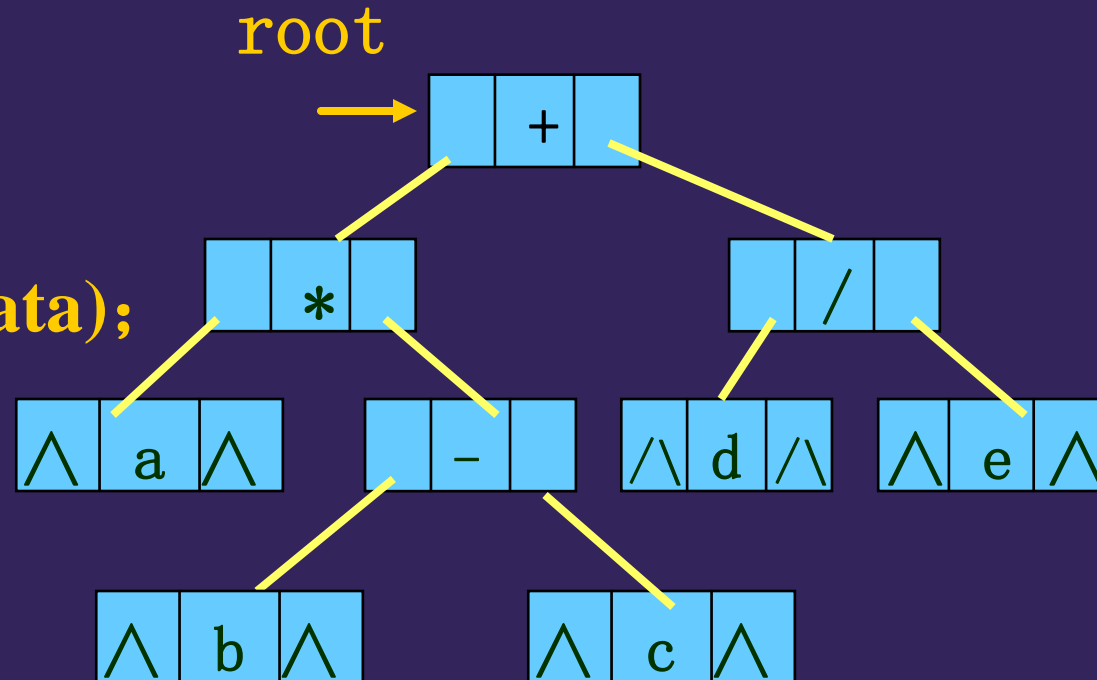
### 2 中序遍历递归算法

```
void mid (NODE *root)
{ if (root!=NULL)
  {prev(root->lch);
   printf("%d,", root->data);
   prev(root->rch);
  }
}
```

中序序列为

$a * b - c + d / e$

称为中缀表达式



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

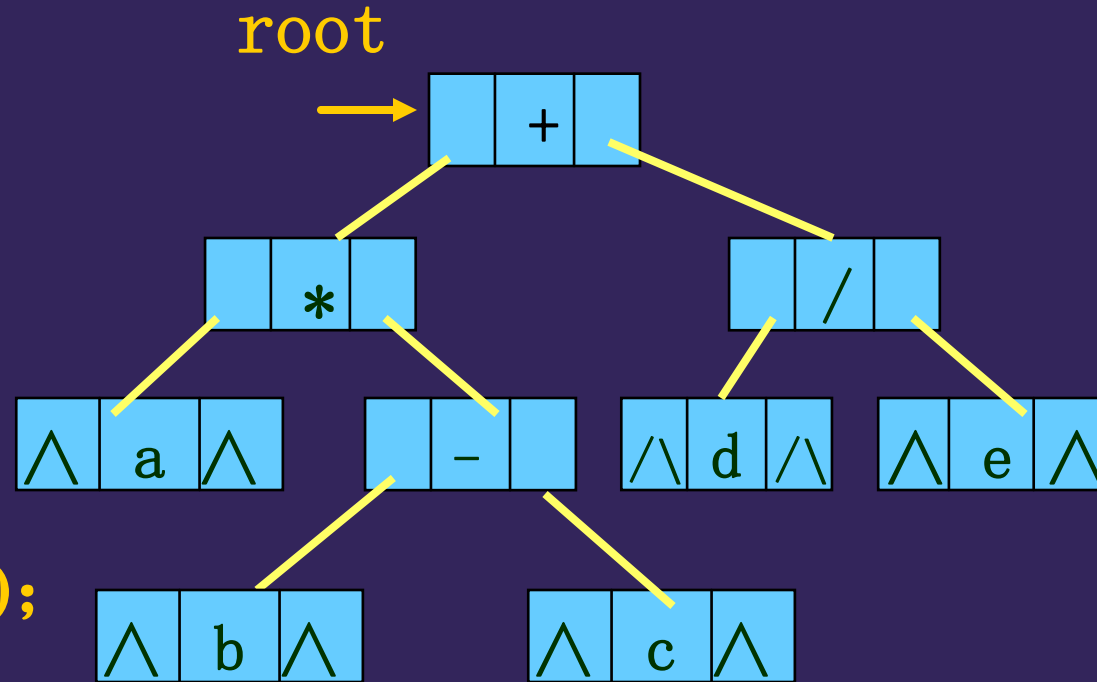
你能写出后序遍历递归算法了吧

## 6.3 二叉树的遍历

### 3 后序遍历递归算法

```
void prev (NODE *root)
{ if (root!=NULL)
  { prev(root->lch);
    prev(root->rch);
    printf("%d,", root->data);
  }
}
```

后序序列为  $a b c - * d e / +$   
称为后缀表达式



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

## 6.3 二叉树的遍历

### 三. 二叉树遍历的非递归算法

递归算法逻辑清晰、易懂，但在实现时，由于函数调用栈层层叠加，效率不高，故有时考虑非递归算法。

#### 1 先序遍历（T L R）的非递归算法。

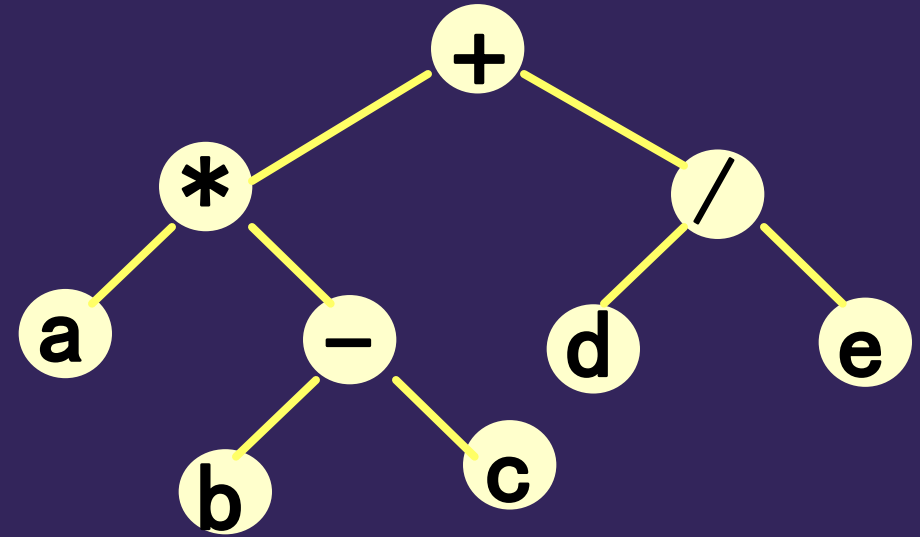
对每个结点，在访问完后，沿其左链一直访问下来，直到左链为空，这时，所有已被访问过的结点进栈。然后结点出栈，对于每个出栈结点，即表示该结点和其左子树已被访问结束，应该访问该结点的右子树。

- (1) 当前指针指向根结点。
- (2) 打印当前结点，当前指针指向其左孩子并进栈，重复（2），直到左孩子为NULL
- (3) 依次退栈，将当前指针指向右孩子
- (4) 若栈非空或当前指针非NULL，执行（2）；否则结束。

## 6.3 二叉树的遍历

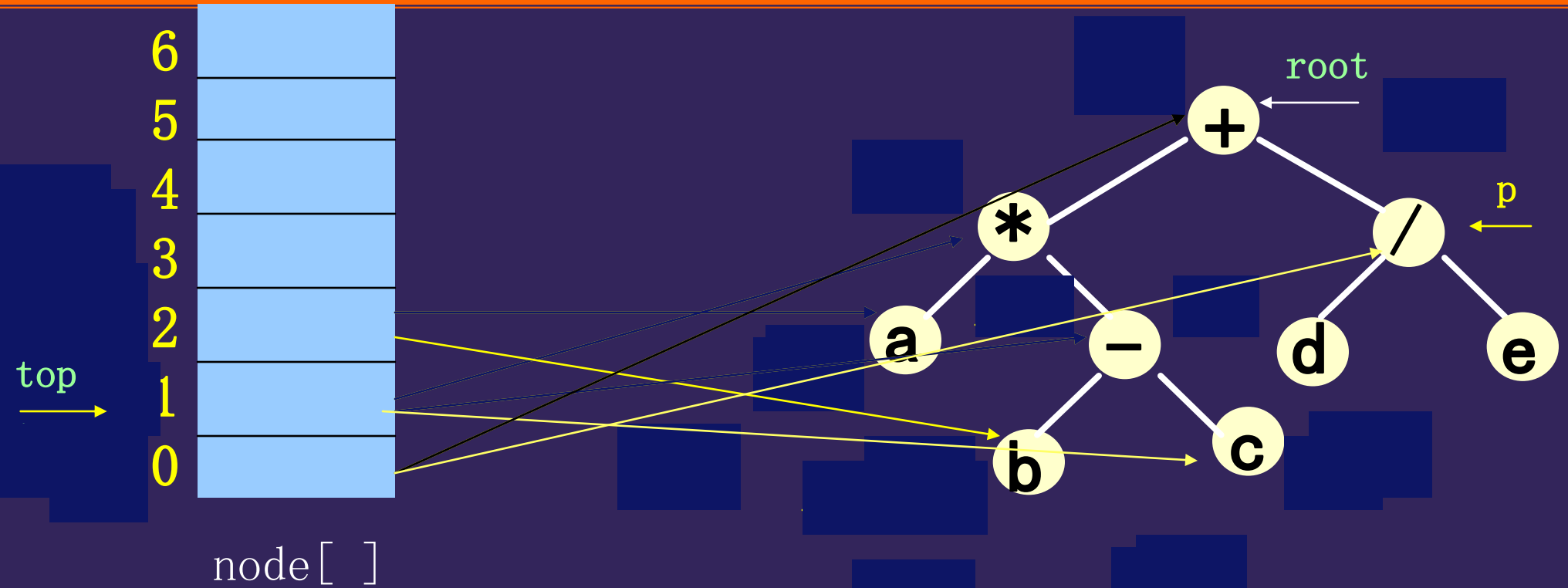
先序遍历的非递归算法

```
void prev (NODE *root)
{ NODE *p, *node[MAX];
  int top=0; p=root;
do
{ while( p!=NULL)
  { printf("%d,", root->data) ;
    node[top]=p;top++;
    p=p->lch;
  }
  if (top>0)
  {top - -; p=node[top]; p=p->rch; }
} while(top>0||p!=NULL);
}
```





## 6.3 二叉树的遍历



```
printf("%d,", root->data);
```

```
node[top]=p;    top++;
```

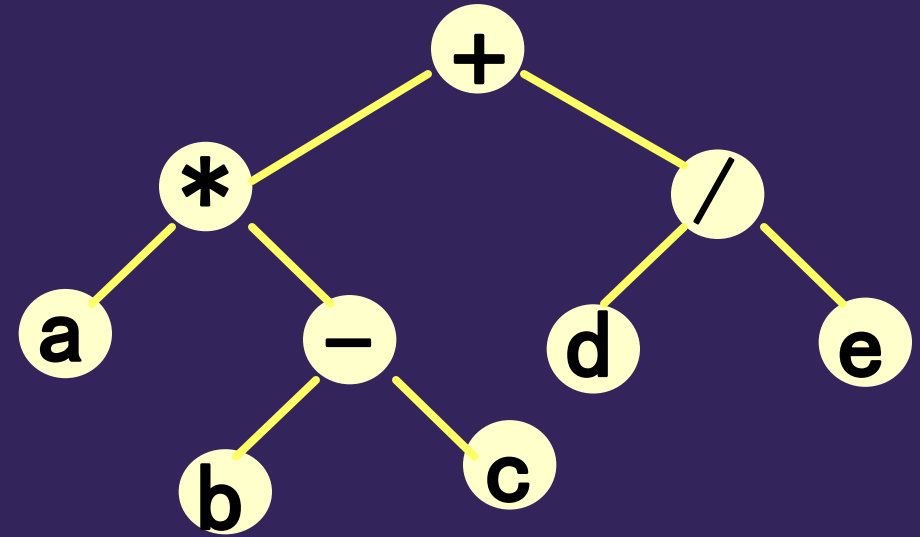
```
p=p->lch;
```

+ \* a - b c / d

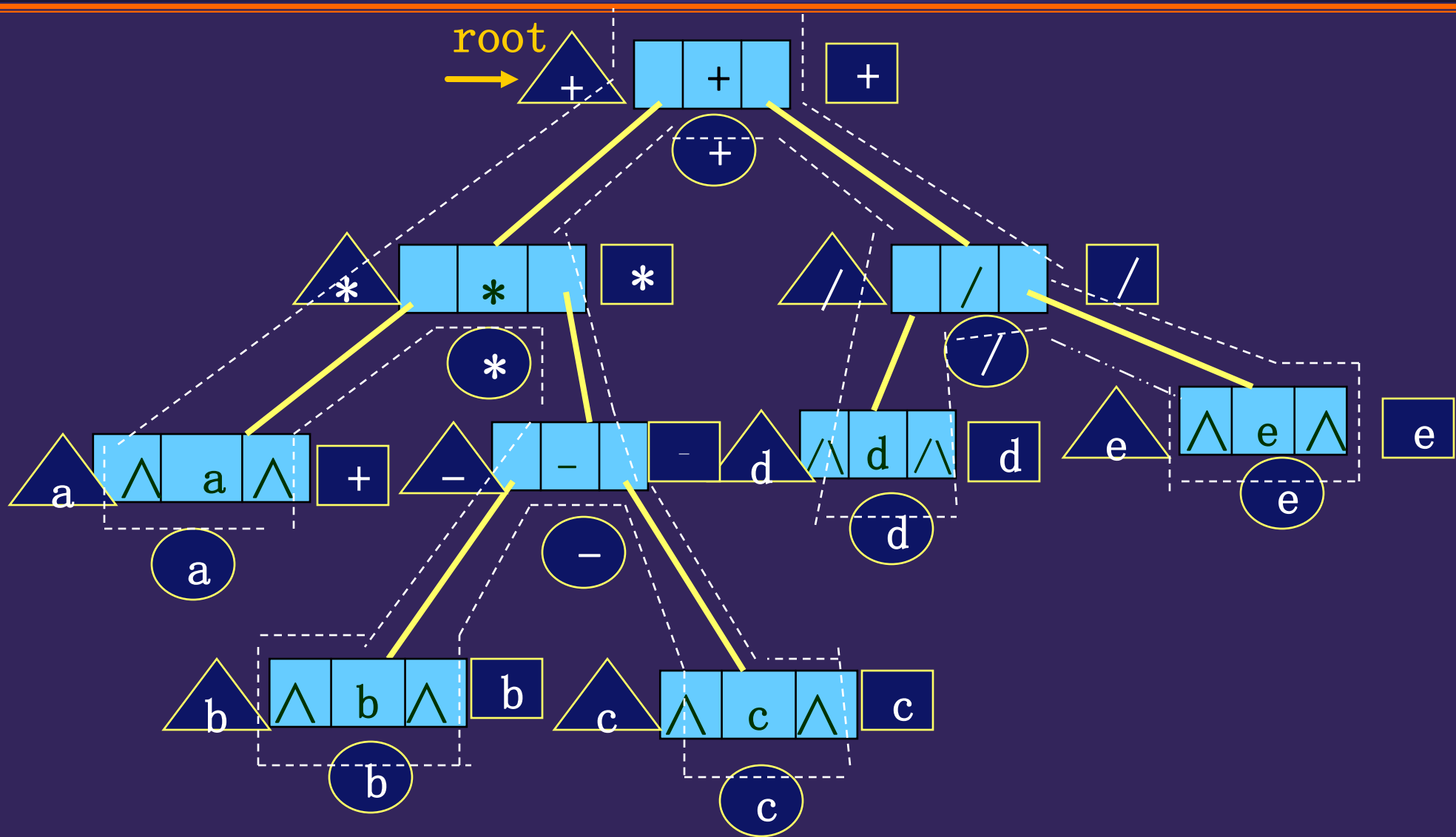
## 6.3 二叉树的遍历

中序遍历的非递归算法

```
void min (NODE *root)
{ NODE *p, *node[MAX];
  int top=0; p=root;
do
{ while( p!=NULL)
{ node[top]=p; top++; p=p->lch;
}
if (top>0)
{ top - -; p=node[top];
  printf("%d,", root->data) ;
  p=p->rch; }
} while(top>0||p!=NULL);
}
```



## 6.3 二叉树的遍历



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

遍历是二叉树的基本操作：二叉树许多操作可在遍历过程中完成，本节再例子举几个二叉树遍历应用实例。

## 6.3 遍历的应用

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

与先序遍历算法比较一下！

```
void leaf(NODE *root)
```

```
//采用二叉链表存贮二叉树，n为全局变量，用于累加二叉树的叶子结点
```

```
//的个数。本算法在先序遍历二叉树的过程中，统计叶子结点的个数
```

```
//第一次被调用时，n=0
```

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

```
{if(root->lch==NULL&&root->rch==NULL)
```

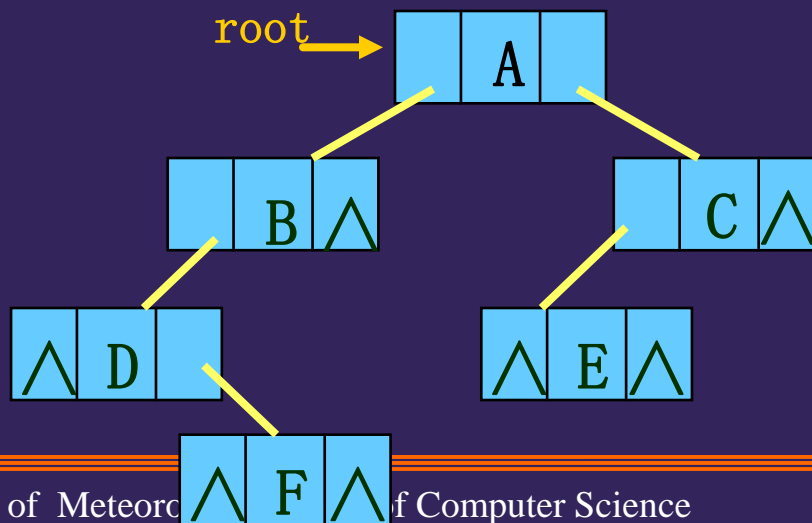
```
n=n+1; //若root所指结点为叶子，则累加
```

```
leaf(root->lch);
```

```
leaf(root->rch);
```

```
}
```

```
}
```



## 6.3 遍历的应用

```
void prev (NODE *root)
```

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

```
{ printf(“%d,”, root->data);
```

访问结点时  
调用printf( )

```
prev(root->lch);
```

```
prev(root->rch); }
```

```
}
```

```
void leaf(NODE *root)
```

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

访问结点时  
统计叶子结点的个数

```
{if(root->lch==NULL&&root->rch==NULL) n=n+1;
```

```
leaf(root->lch);
```

函数名不同

```
leaf(root->rch); }
```

结构类似

## 6.3 遍历的应用

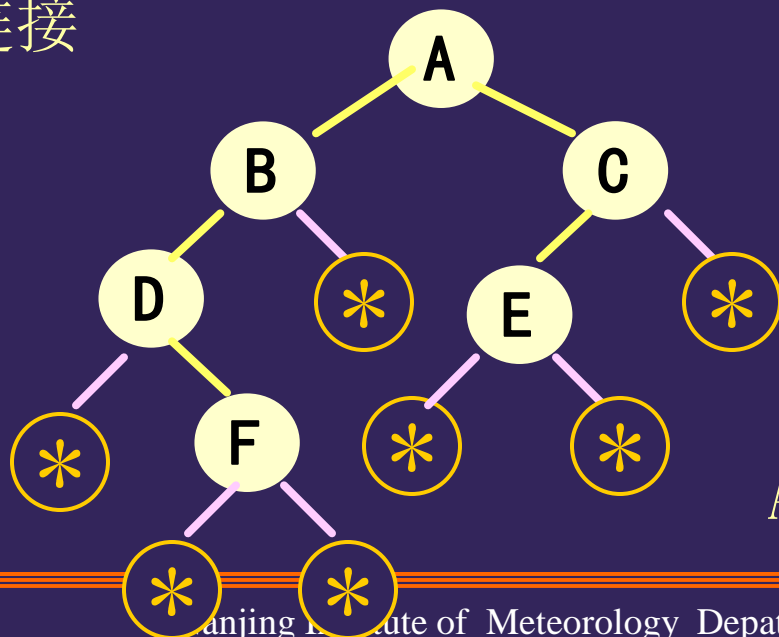
### 例2 建立二叉链表

输入：二叉树的先序序列

结果：二叉树的二叉链表

遍历操作访问二叉树的每个结点，而且每个结点仅被访问一次。是否可在利用遍历，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



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

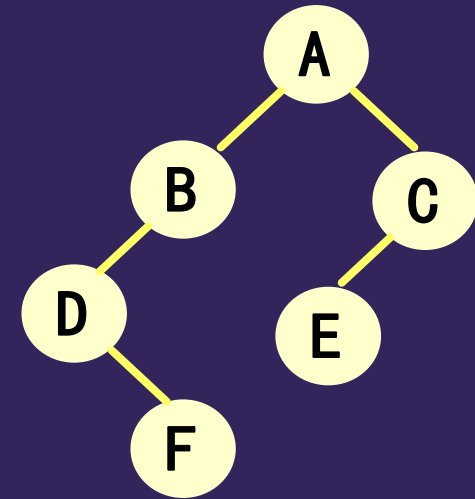
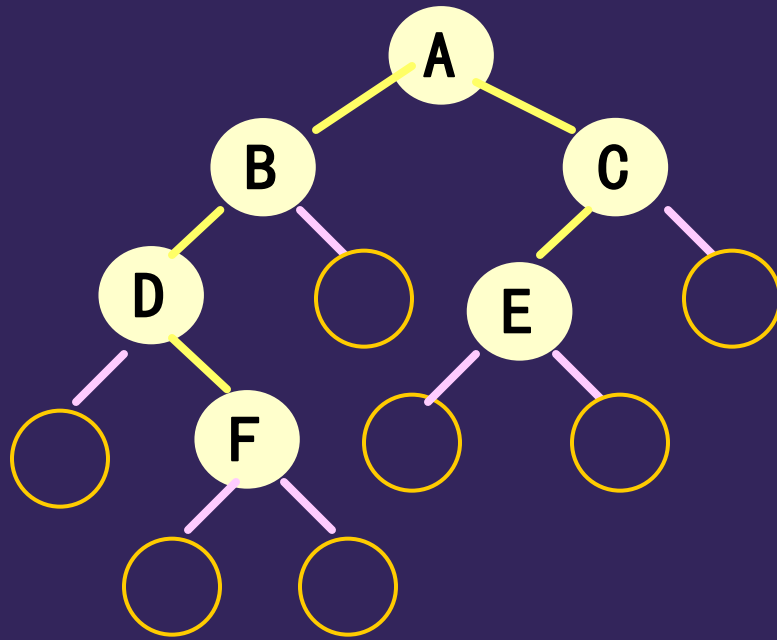
## 6.3 遍历的应用

输入（在空子树处添加空格字符的二叉树的）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，并将该二叉链表根结点指针赋给root

```
NODE *create_tree(NODE *root)
{ char ch;
  scanf (&ch);
  if (ch == ' ') root=NULL;           // 若ch == '*' 则root=NULL返回
  else {                               // 若ch != '*'
    root=( NODE * )malloc(sizeof(NODE) ;
    root->date = ch;                 // 建立（根）结点
    root->lch=create_tree(root->lch); //构造左子树链表，并将左子树根结点指针赋给（根）结点的左孩子域
    root->rch=create_tree(root->rch); //构造右子树链表，并将右子树根
    } return (root);                 //结点指针赋给（根）结点的右孩子域
}
```



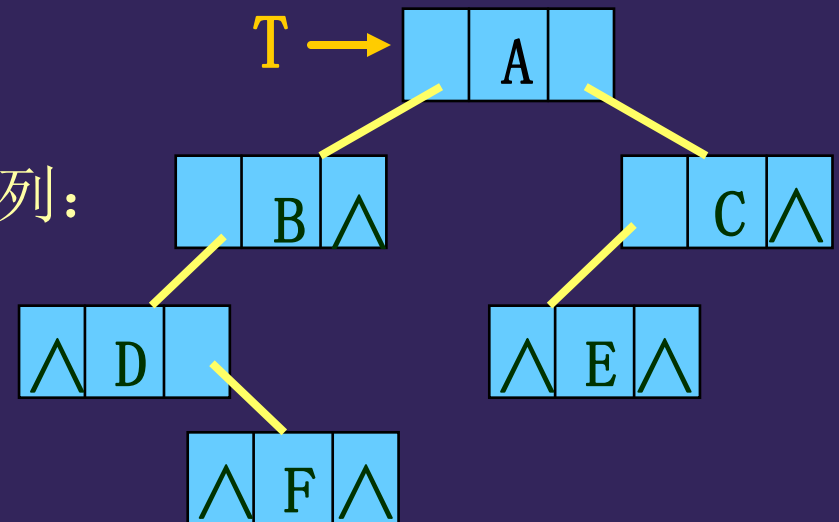
## 6.3 遍历的应用



先序序列: A B D F C E

(在空子树处添加\*的二叉树的) 先序序列:

A B D ■ F ■ ■ ■ C E ■ ■ ■



## 小 结

- 1 二叉树： 或为空树，或由根及两颗不相交的左子树、右子树构成，并且左、右子树本身也是二叉树；
- 2 二叉树即可以用顺序结构存储，也可用链式结构存储；
- 3 遍历：按某种搜索路径访问二叉树的每个结点，每个结点仅被访问一次。 **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**，本课程介绍了三种遍历算法：**先序遍历、中序遍历、后序遍历**；

### 6.4 树和森林

- 一. 树的存储结构
- 二. 树和二叉树的转换
- 三. 树的遍历
- 四. 森林

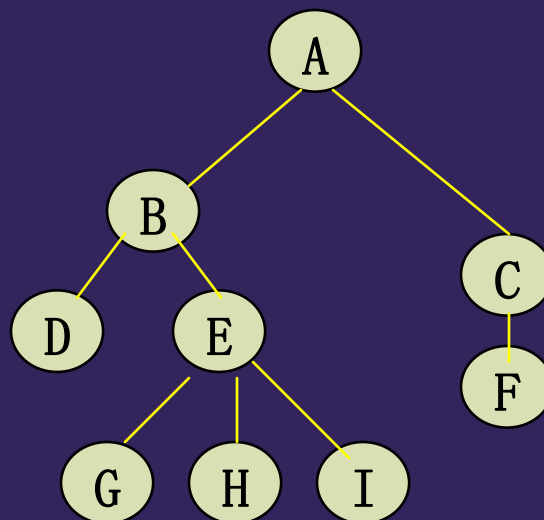
### 一. 树的存储结构

#### 1 双亲表示法

采用一组连续空间存储树的结点，通过保存每个结点的双亲结点的位置，表示树中结点之间的结构关系。

#### 双亲表示类型定义

```
#define MAX 100
struct node{
    char data;
    int parent; //双亲位置域
}; typedef struct node NODE;
NODE tree[MAX];
```



结点数

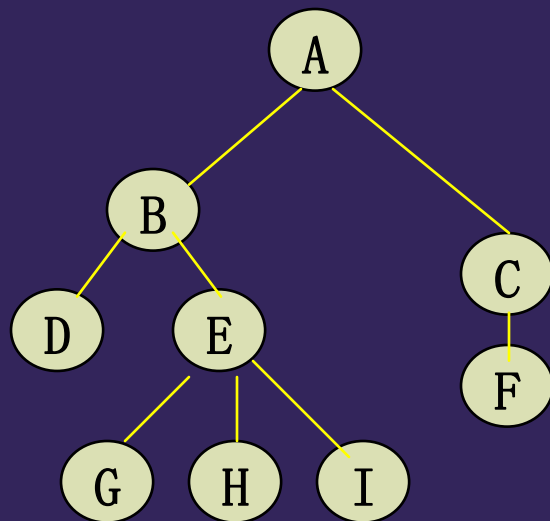
	data	parent
0		9
1	A	0
2	B	1
3	C	1
4	D	2
5	E	2
6	F	3
7	G	5
8	H	5
9	I	5

#### 2、孩子表示法

通过保存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。

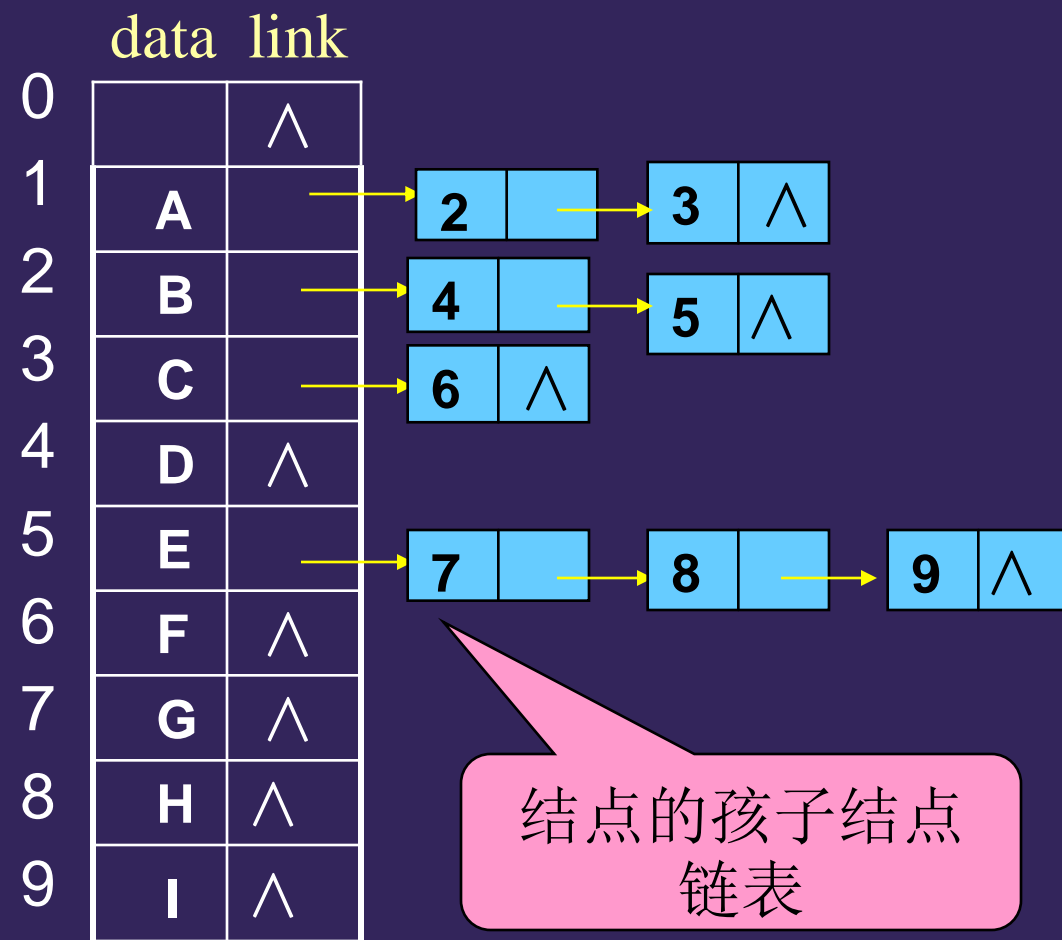
## 6.4 树和森林

孩子链表：对树的每个结点用线性链表存贮它的孩子结点



树的孩子链表图示

找一个结点的孩子十分方便，  
但要找一个结点的双亲则要遍  
历整个结构



## 6.4 树和森林

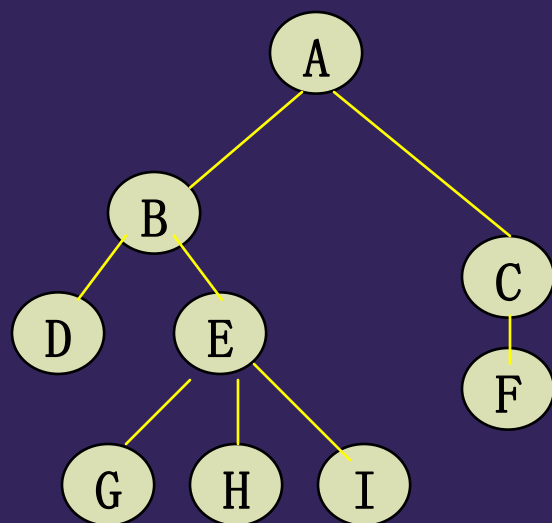
树的孩子链表类型定义

```
struct node      //孩子结点
{
    int  child;
    struct node * link;
};

struct t_node
{
    char data;
    struct t_node * link; //孩子链表头指针
} tree[MAX];
```

## 6.4 树和森林

双亲孩子表示法：结合双亲表示法和孩子表示法



	data	parent	link
0		9	^
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	^

结点的孩子结点链表

带双亲孩子链表

## 6.4 树和森林

树的双亲孩子链表类型定义

```
struct node      //孩子结点
{ int  child;
  struct node * link;
};

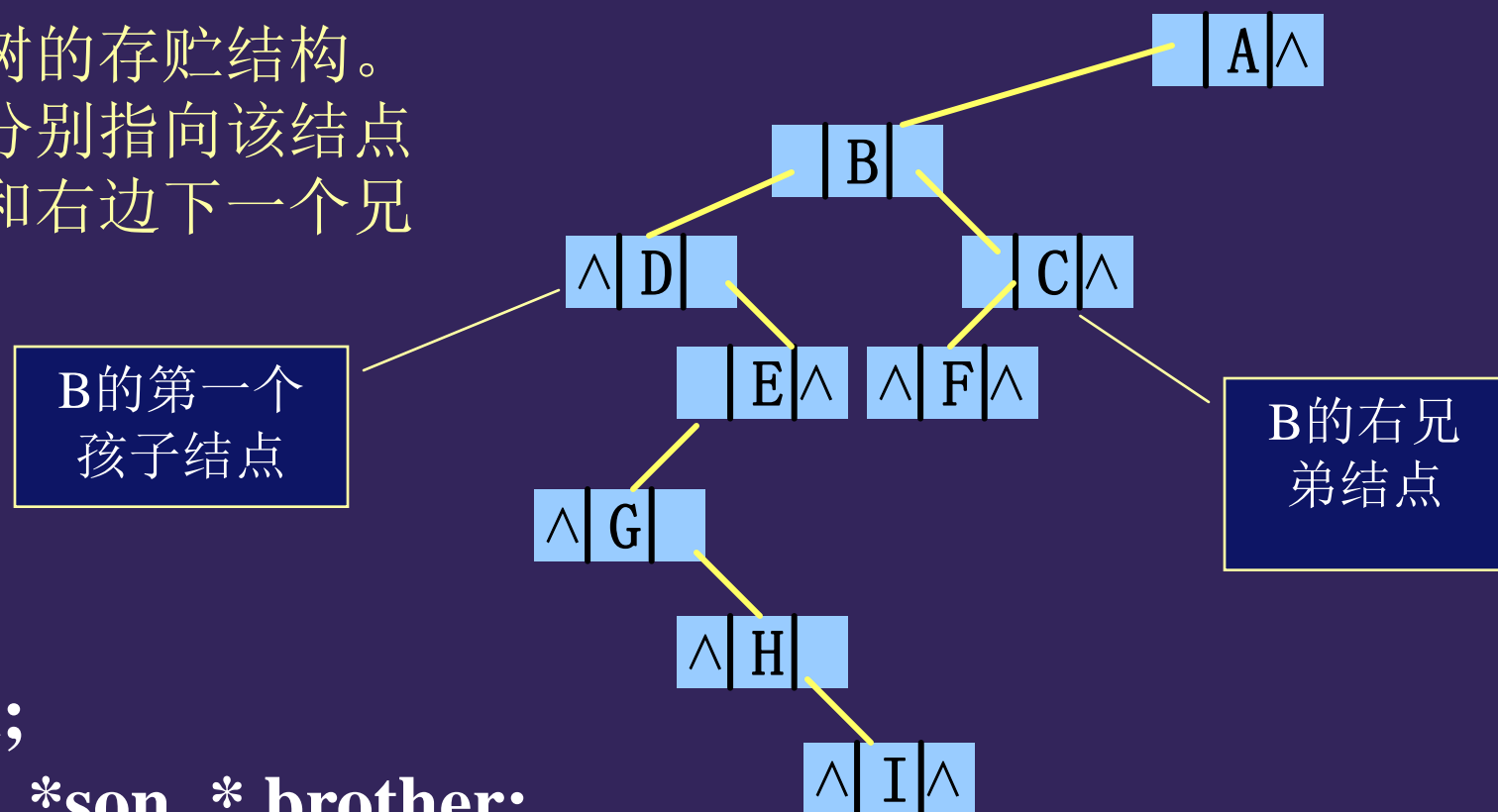
struct t_node
{ char data;
  int parent;
  struct t_node * link; //孩子链表头指针
} tree[MAX];
```



## 6.4 树和森林

### 3 孩子兄弟表示法

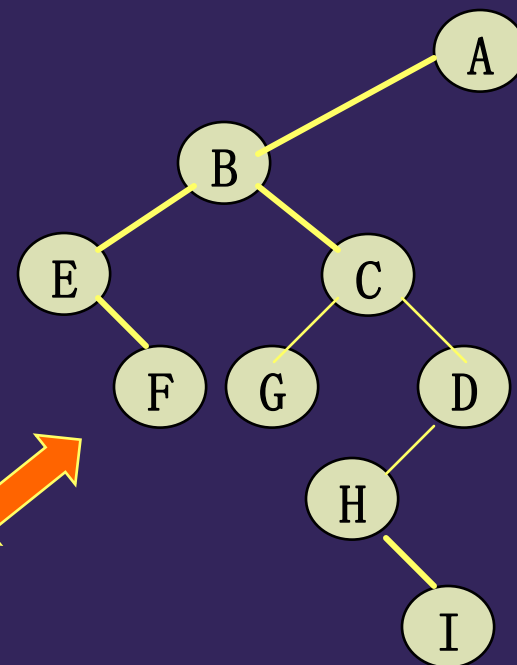
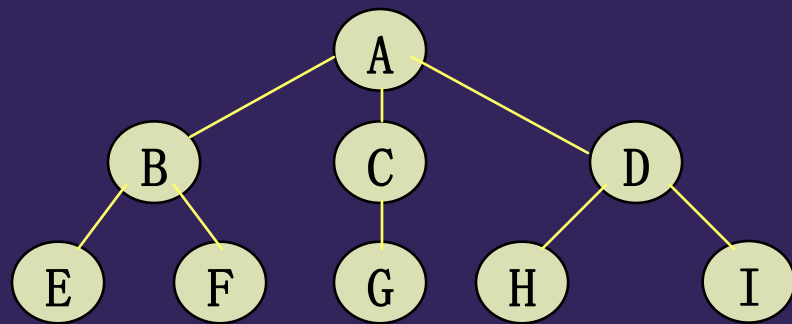
用二叉链表作为树的存贮结构。  
链表的两个指针域分别指向该结点的  
的第一个孩子结点和右边下一个兄  
弟结点。



```
struct node
{ char data;
  struct node *son, * brother;
};
```

树的孩子兄弟表示法图示

## 6.4 树和森林



(a)

(b)

此二叉链表既是树 (a) 的孩子兄弟表示又是二叉树 (b) 的二叉链表

由此可将  
树与二叉树  
对应起来

### 二 树与二叉树的转换

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

#### 树与二叉树转换方法



## 6.4 树和森林

树

根

结点 X 的第一个孩子

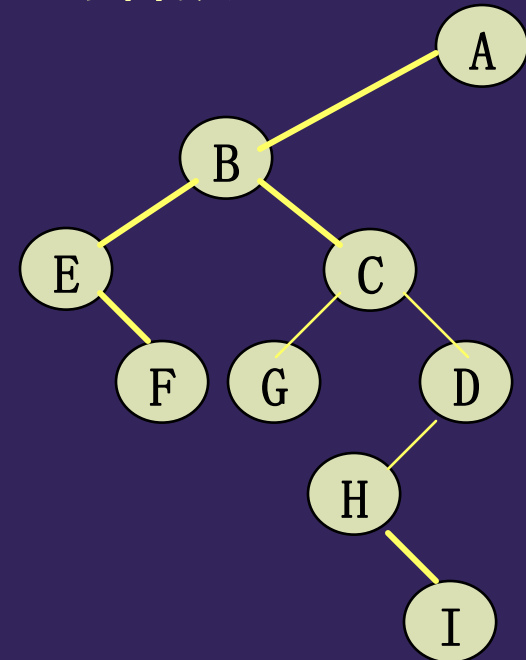
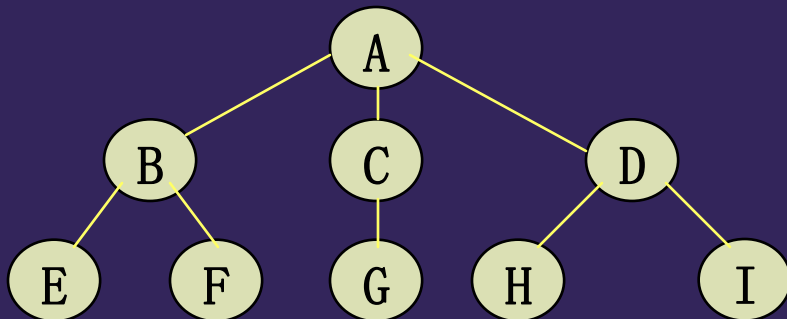
结点 X 紧邻的右兄弟

二叉树

根

结点 X 的左孩子

结点 X 的右孩子



### 四 森林：树的集合

将森林中树的根看成兄弟，可用树孩子兄弟表示法存储森林；用**树与二叉树的转换方法**，进行森林与二叉树转换；从树的二叉链表示的定义可知,任何一棵和树对应的二叉树,其右子树必为空。所以只要将森林中所有树的根结点视为兄弟，即将各个树转换为二叉树；再按森林中树的次序，依次将后一个树作为前一棵树的右子树，并将第一棵树的根作为目标树的根，就可以将森林转换为二叉树。

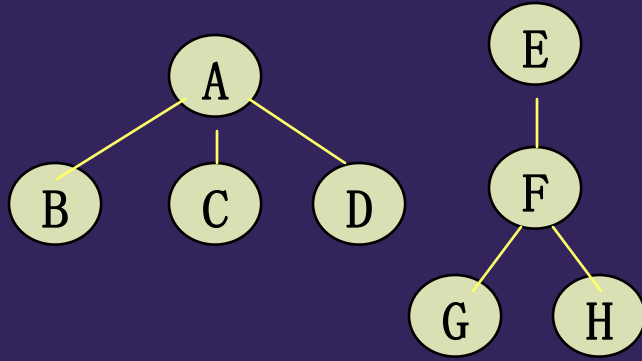
转换规则：

若  $F = \{ T_1, T_2, T_3, \dots, T_n \}$  是森林，则  $B(F) = \{\text{root}, \text{LB}, \text{RB}\}$

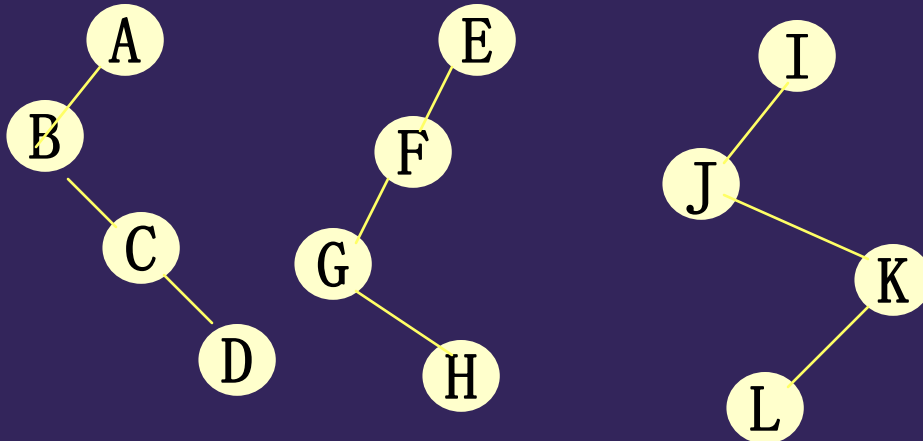
(1) 若  $F$  为空，即  $n=0$ ，则  $B(F)$  为空树。

(2) 若  $F$  非空，则  $B(F)$  的根是  $T_1$  的根，其左子树为  $\text{LB}$ ，是从  $T_1$  根结点的子树森林  $F_1 = \{T_{11}, T_{12}, \dots, T_{1m}\}$  转换而成的二叉树；其右子树为  $\text{RB}$ ，是从除  $T_1$  外的森林  $F' = \{T_2, T_3, \dots, T_n\}$  转换而成的二叉树；

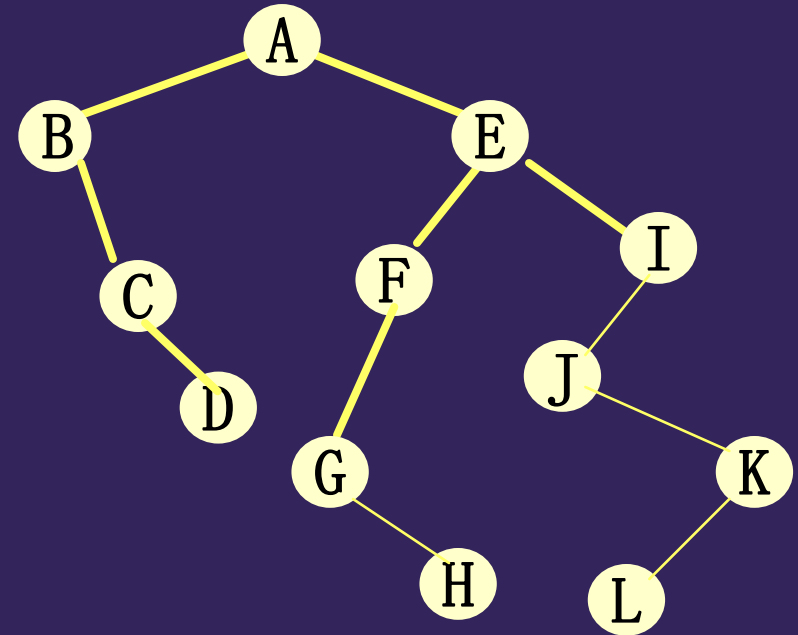
## 6.4 树和森林



包含3棵树的森林



每棵树对应的二叉树



森林对应的二叉树

### 二叉树还原为森林

转换规则：

若  $B(F) = \{\text{root}, \text{LB}, \text{RB}\}$  是一棵二叉树，则转换为森林  $F = \{T_1, T_2, T_n\}$  的规则为

- (1) 若  $B$  为空，则  $F$  为空树。
- (2) 若  $B$  非空，则  $F$  第一棵树  $T_1$  的根是二叉树的根， $T_1$  中根结点的子森林  $F_1$  是由  $B$  的左子树  $\text{LB}$  转换而成的森林， $F$  中除  $T_1$  外其余树组成的森林  $F' = \{T_2, T_3, \dots, T_n\}$  是由  $B(F)$  的右子树  $\text{RB}$  转换而成的；

### 6.5 树的应用

6.5.1 二叉排序树

6.5.2 哈夫曼树以及应用

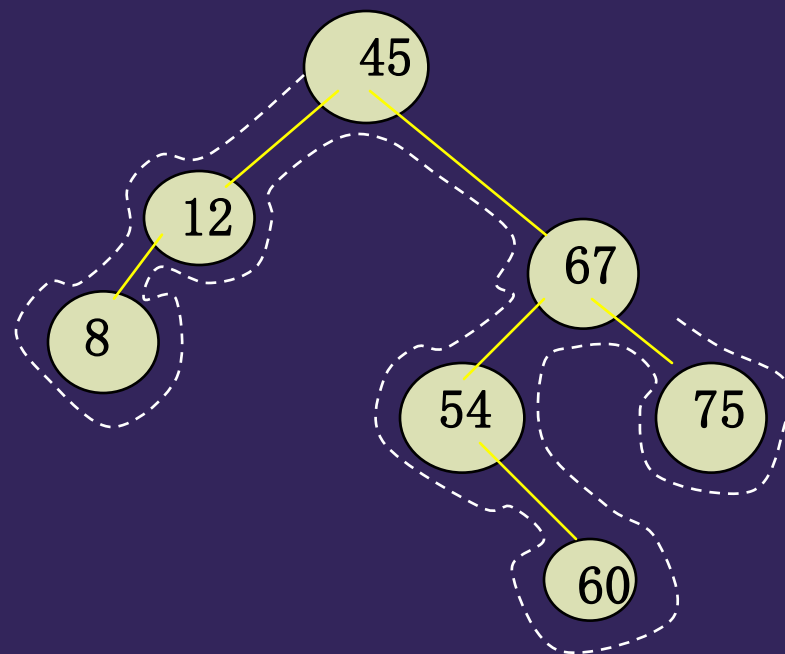


### 6.5.1 二叉排序树

#### 1. 二叉排序树的定义

一种特殊的二叉树，它的每个结点数据中都有一个关键值，并有如下性质：对于每个结点，如果其左子树非空，则左子树的所有结点的关键值都小于该结点的关键值；如果其右子树非空，则右子树的所有结点的关键值都大于该结点的关键值。

二叉排序树有查找效率高，增、删方便的优点，且对二叉排序树进行中序遍历，将得到一个按结点关键值递增有序的中序线性序列。所以，它被广泛用来作为动态查找的数据结构。



二叉排序树

中序遍历：

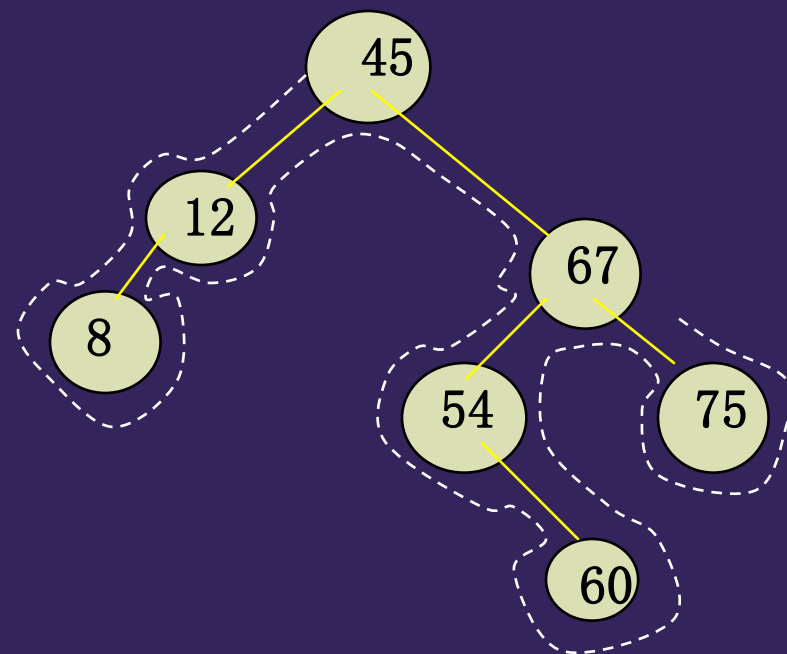
8 — 12 — 45 — 54 — 60 — 67 — 75

## 6.5 树的应用

**例如，**用二叉排序树作为目录树，把一个记录的关键码和记录的地址作为二叉排序树的结点，按关键码值建成二叉排序树。这样，既能像有序表那样进行高效查找，又能像链表那样灵活删除，而不要移动其它结点。

### 2. 二叉排序树的查找

在二叉排序树中进行查找，将要查找的值从树根开始比较，若与根的关键值相等，则查找成功，若比根值小，则到左子树找，若比根值大，则到右子树找，直到查找成功或查找子树为空（失败）。



二叉排序树

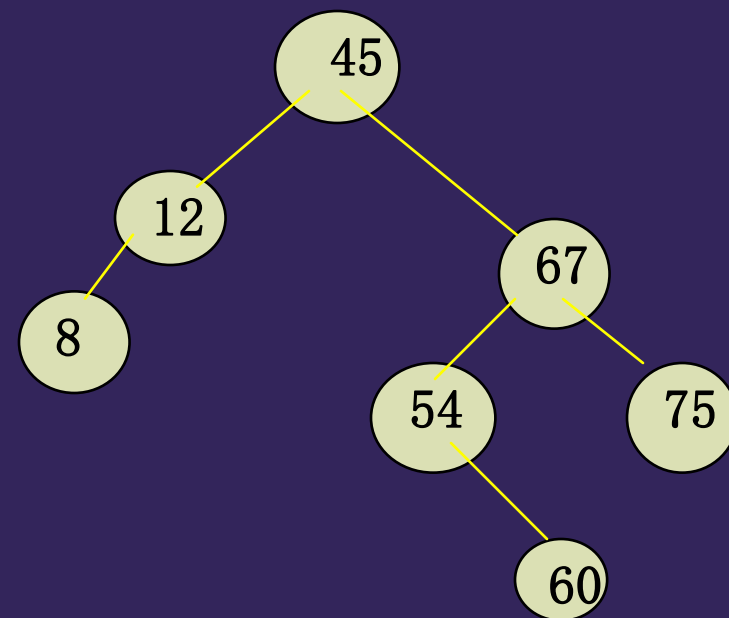
## 6.5 树的应用

二叉排序树的查找递归算法:

```
NODE *search(int x, NODE *root)  
{ if ((root==NULL)|| (root->data==x)) return(root);  
  
if(root->data<x) return(search(x,root->rch));  
  
else return(search(x,root->lch);  
  
}
```

### 3. 二叉排序树的插入和生成

插入：二叉排序树的插入是一个动态的查找过程。例如，要插入关键值  $x$ ，首先在树中查找，若不存在则插入。因为查找失败的情况是一直查到查找路径的末端，仍然存在  $x$ ，则待插入结点必作为树叶插入树中。



二叉排序树

二叉排序树的插入递归算法：

```
NODE *insert(NODE *p, NODE *root)
```

```
{ if (root==NULL) return(p);
```

```
if(p->data<root->data) root->lch=insert(p,root->lch);
```

```
else root->rch=insert(p,root->rch); return(root);
```

```
}
```

```
main( )
```

```
{ int a[MAX]={45,24,53,45,12,24,90};
```

```
  NODE *new, *head=NULL; int i;
```

```
  for (i=0;i<MAX;i+ +)
```

```
  { new=(NODE *)malloc(sizeof(NODE));
```

```
    new->data=a[i]; new->lch=new->rch=NULL;
```

```
    head=insert(new,head); }
```

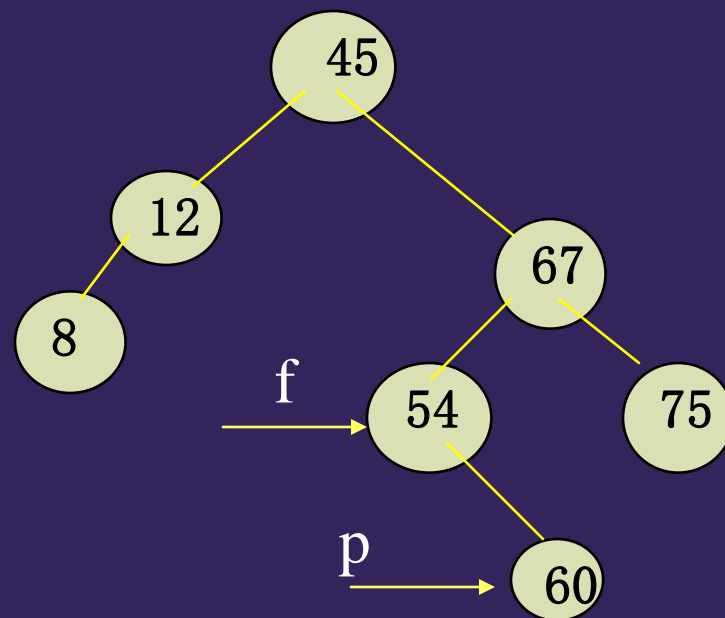
```
}
```

### 4. 二叉排序树的删除

删除：首先查找到要删除的结点,删除后二叉排序树的中序序列仍然是按关键值递增有序。分三种情况：

(1) 若 $*p$ 结点是叶子，则直接删除。即：将双亲结点指向它的指针设置为空。

( $f \rightarrow rch = \text{NULL}$ )

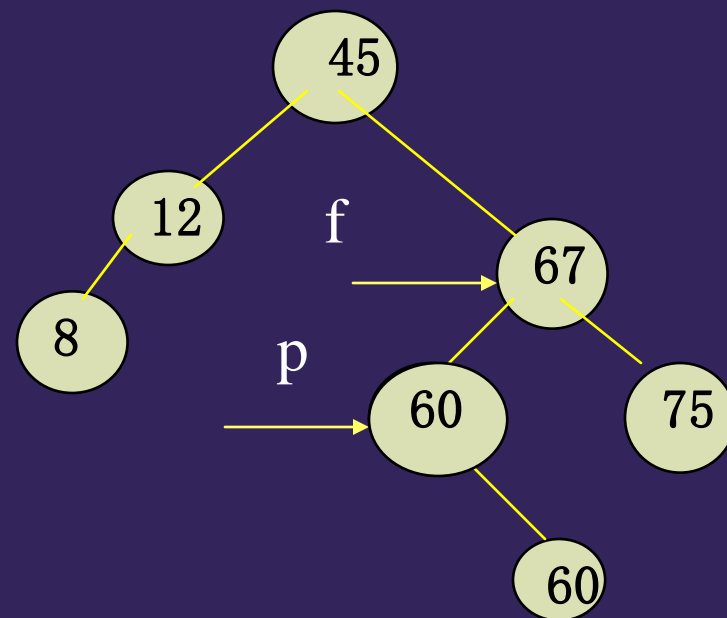


删除树叶结点 $*p$

### 4. 二叉排序树的删除

(2) 若 $*p$ 为单分支结点，则只需用它唯一子树的根去继承它的位置。

( $f \rightarrow lch = p \rightarrow rch$ )

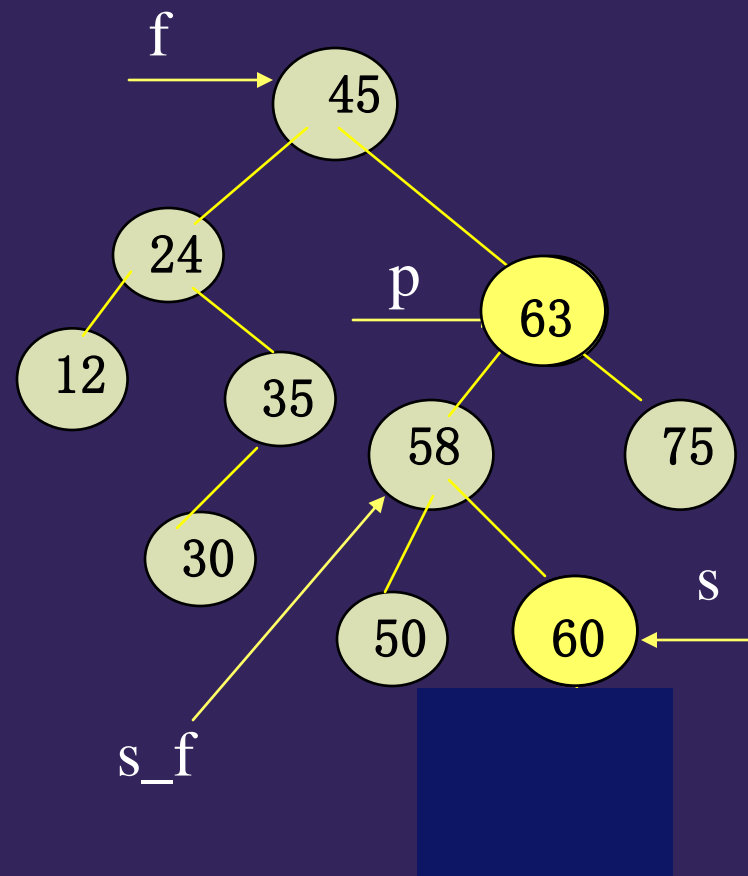


删除单分支结点 $*p$

### 4. 二叉排序树的删除

(3) 若 $*p$ 为双分支结点，用左子树中序遍历的最后一个结点(左子树的最右结点) 替换 $*p$ 。首先，沿 $*p$ 的左子树的右链，查找到右指针域为空的结点 $*s$ ，然后用 $*s$ 的数据替换 $*p$ 的数据，最后,删除 $*s$ 结点

删除两分支结点 $*p$





## 6.5 树的应用

二叉排序树的删除结点算法:

```
NODE *del( NODE *root, int x)  
{ NODE *p, *f, *s, *s_f;  
    if (root==NULL) return(root);  
    f=NULL; p=root;  
    while(p!=NULL && p->data!=x)  
    { if(p->data>x)  
        if (p->lch!=NULL) {f=p;p=p->lch;} else break;  
        else if(p->rch!=NULL){f=p;p=p->rch;} else break;  
    }  
    if( p==NULL|| p->data!=x) return(root);  
    if (p==root)return;
```

## 6.5 树的应用

```
if (p->lch==NULL && p->rch==NULL)
{ if( f==NULL) {free(p); return(NULL);}
  if ( f->lch==p) f->lch=NULL;
    else f->rch=NULL;
    free(p);return(root);
}
if (p->lch==NULL)
{ if (f==NULL){ root=p->rch;free(p);return(root);}
  if(f->lch==p) f->lch=p->rch;
    else f->rch=p->rch;
    free(p);return(root);
}
```

## 6.5 树的应用

```
if (p->rch==NULL)
{
    if (f==NULL){ root=p->lch;free(p);return(root);}
    if(f->lch==p) f->lch=p->lch;
    else f->rch=p->lch;
    free(p);return(root);
}

s=f;p=s->lch;
while (s->rch!=NULL)
{
    s_f=s;s=s->rch;
}
p->data=s->data;
if(s->lch==NULL) s_f->rch=NULL;
else s_f->rch=s->lch;
free(s);return(root);
}
```

## 6.5.2 哈夫曼树(最优树) 及应用

### 1 哈夫曼树的概念

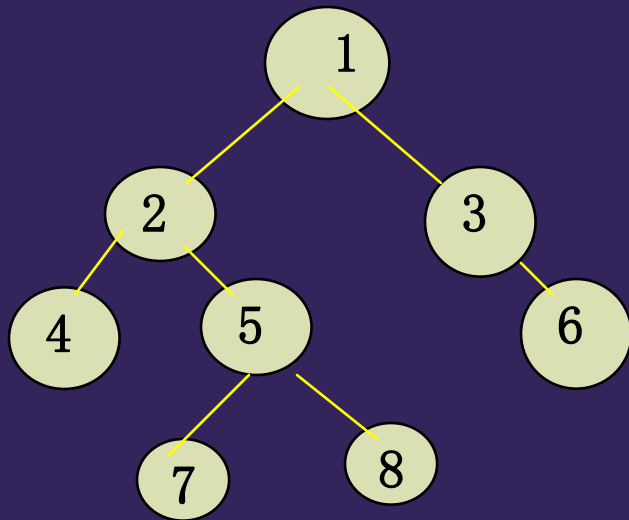
**路径：**从一个结点到另一个结点之间的若干个分支

**路径长度：**路径上的分支数目称为路径长度；

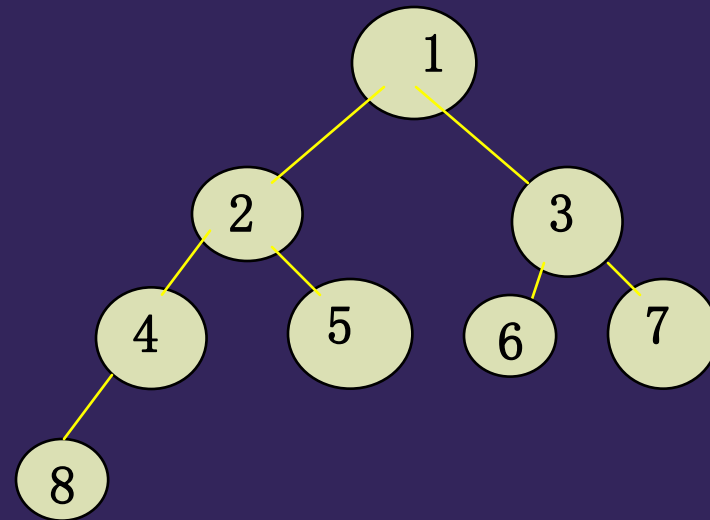
**结点的路径长度：**从根到该结点的路径长度

**树的路径长度：**树中所有叶子结点的路径长度之和；一般记为PL。

在结点数相同的条件下，完全二叉树是路径最短的二叉树。



非完全二叉树



完全二叉树

## 6.5.2 哈夫曼树(最优树) 及应用

### 1 哈夫曼树的概念

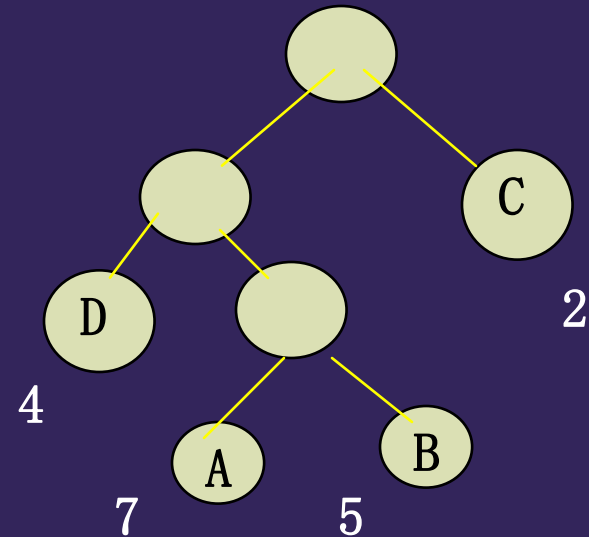
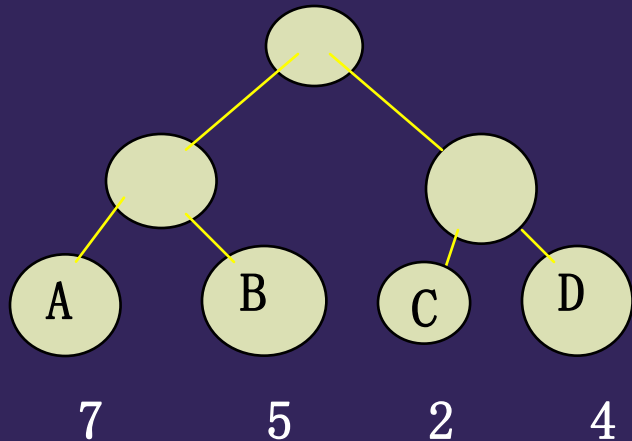
**结点的权：**根据应用的需要可以给树的结点赋权值；

**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；

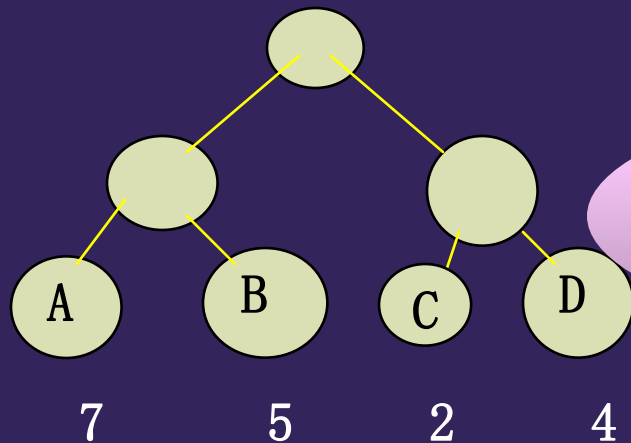
**树的带权路径长度=**树中所有叶子结点的带权路径之和；通常记作

$$WPL = \sum w_i \times L_i$$

**哈夫曼树：**假设有n个权值( $w_1, w_2, \dots, w_n$ )，构造有n个叶子结点的严格二叉树，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严格二叉树称为**哈夫曼树**。

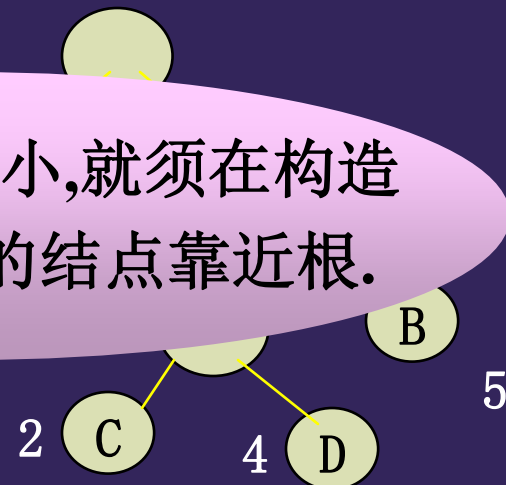


## 6.5.2 哈夫曼树(最优树) 及应用

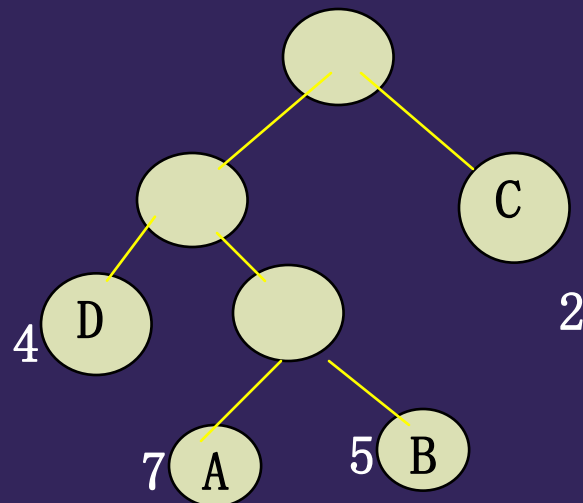


$$WPL=7*2+5*2+2*2+4*2=36$$

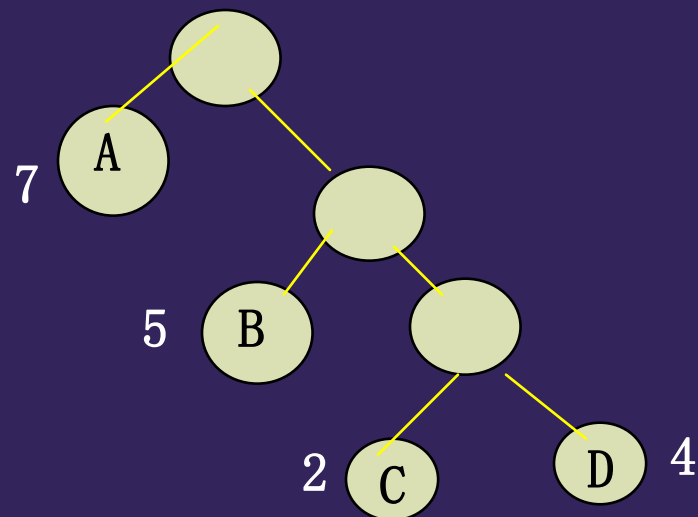
要使二叉树WPL小,就须在构造  
树时,将权值大的结点靠近根.



$$WPL=7*1+5*2+2*3+4*3=35$$



$$WPL=7*3+5*3+2*1+4*2=46$$



$$WPL=7*1+5*2+2*3+4*3=35$$

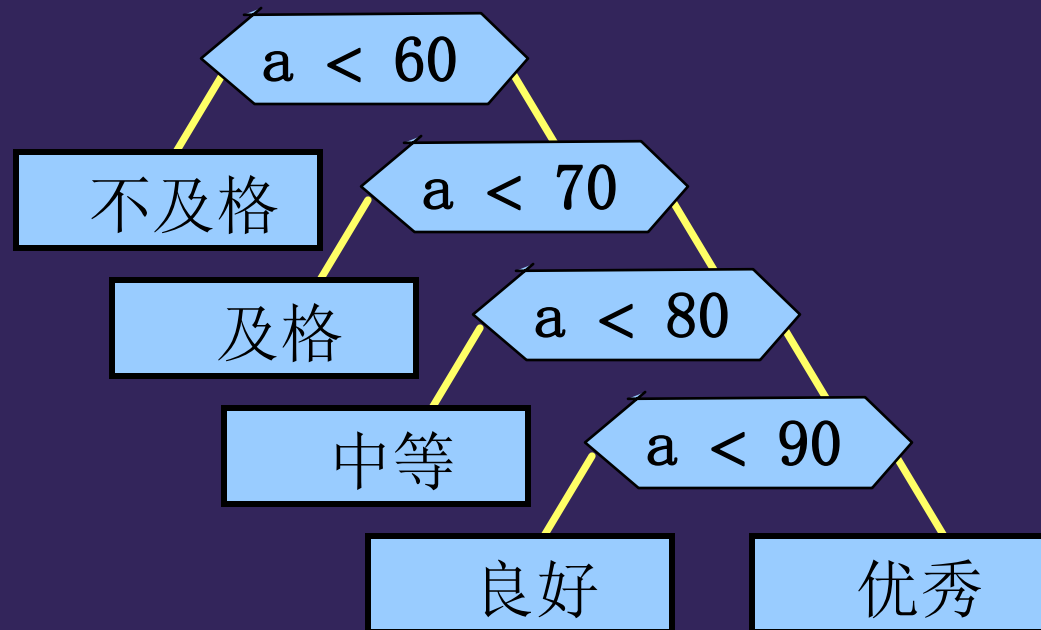
## 6.5.2 哈夫曼树及应用

### 2 应用举例

在求得某些判定问题时，利用哈夫曼树获得最佳判定算法。

例 编制一个将百分制转换成五分制的程序。

最直观的方法是利用if语句来的实现。可用二叉树描述判定过程。



## 6.5.2 哈夫曼树及应用

设有10000个百分制分数要转换，设学生成绩在5个等级以上的分布如下：

分数	0-59	60-69	70-79	80-
比例数	0.05	0.15	0.40	0.30

构造以分数的分布比例为权值的哈夫曼树

按图的判定过程：

转换一个分数所需的比较次数=

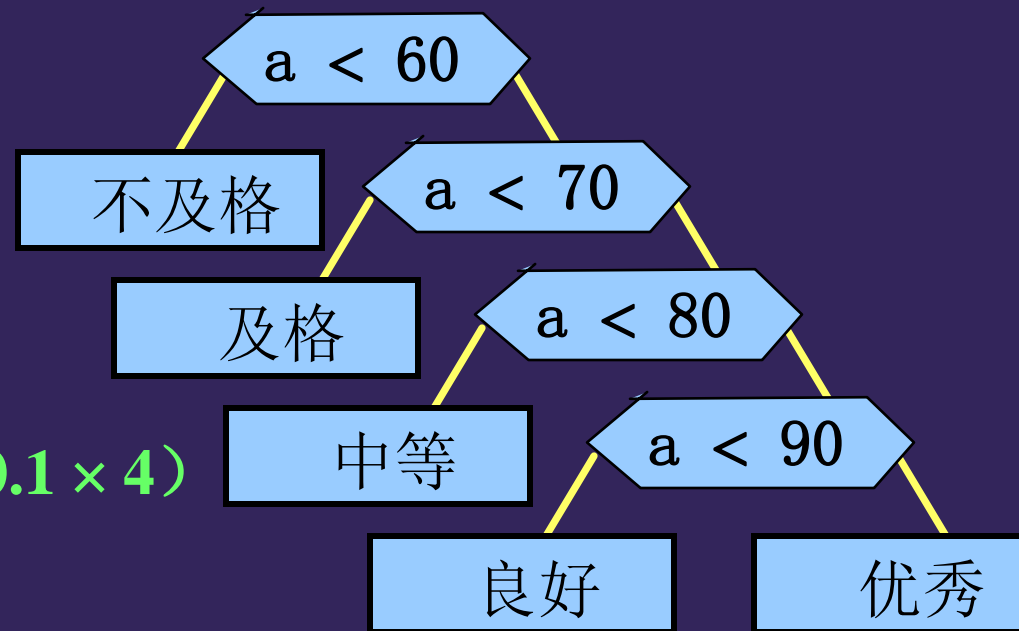
从根到对应结点的路径长度

转换10000个分数所需的总比较次数=

$10000 \times$

$(0.05 \times 1 + 0.15 \times 2 + 0.4 \times 3 + 0.3 \times 4 + 0.1 \times 4)$

二叉树的  
带权路径长度





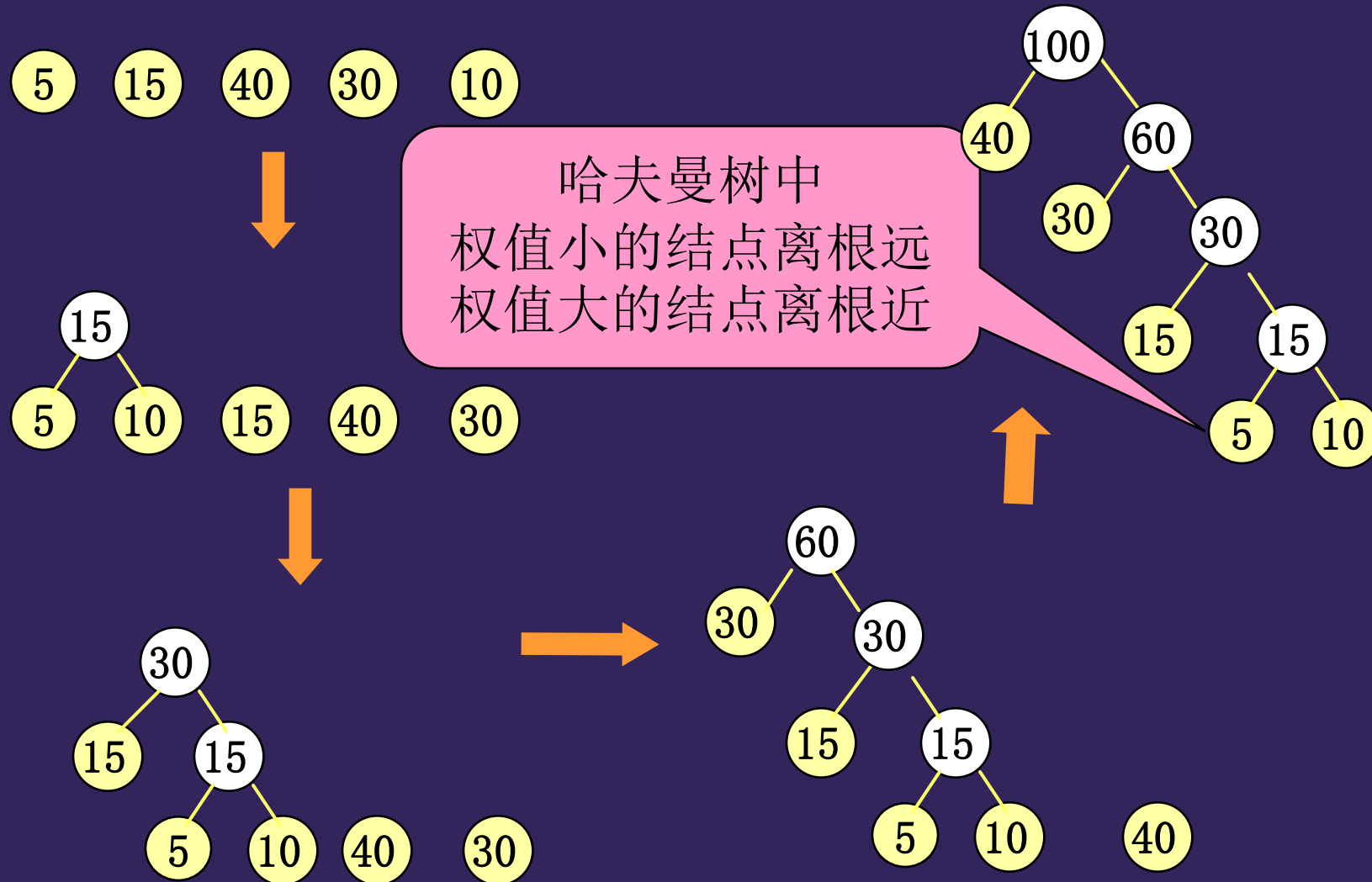
### 3 哈夫曼树的构造

#### 构造哈夫曼树的步骤:

1. 根据给定的 $n$ 个权值，构造 $n$ 棵只有一个根结点的二叉树， $n$ 个权值分别是这些二叉树根结点的权。设 $F$ 是由这 $n$ 棵二叉树构成的集合
2. 在 $F$ 中选取两棵根结点权值最小的树作为左、右子树，构造一颗新的二叉树，置新二叉树根的权值=左、右子树根结点权值之和；
3. 从 $F$ 中删除这两棵树，并将新树加入 $F$ ；
4. 重复 2、3，直到 $F$ 中只含一颗树为止；

## 6.5.2 哈夫曼树及应用

例：构造以 $W = (5, 15, 40, 30, 10)$ 为权的哈夫曼树。



## 6.5.2 哈夫曼树及应用

### 6.7.2 哈夫曼编码

在进行数据通讯时，涉及数据编码问题。所谓数据编码就是数据与二进制字符串的转换。

例如：邮局发电报：



例 要传输的原文为ABACCD A

等长编码 A: 00 B: 01 C: 10 D: 11

发送方：将ABACCD A 转换成 00010010101100

接收方：将 00010010101100 还原为 ABACCD A

不等长编码 A: 0 B: 00 C: 1 D: 01

发送方：将ABACCD A 转换成 000011010

接收方：000011010 转换成  $\left\{ \begin{array}{l} \text{AAAACCD A} \\ \text{BBCCD A} \end{array} \right.$

A的编码是  
B的前缀

## 6.5.2 哈夫曼树及应用

前缀编码： 任何字符编码不是其它字符编码的前缀

设 A: 0 B: 110 C: 10 D: 111

发送方： 将ABACCDA 转换成 0110010101110 总长度是13

所得的译码是唯一的

## 6.5.2 哈夫曼树及应用

可利用二叉树设计前缀编码:

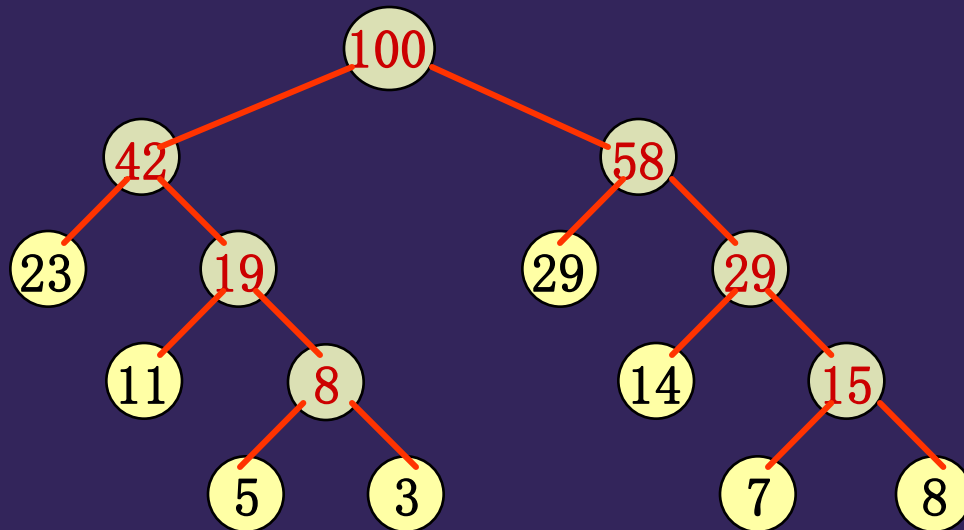
**例** 某通讯系统只使用8种字符a、b、c、d、e、f、g、h，其使用频率分别为0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03, 0.11，利用二叉树设计一种不等长编码：

- 1) 构造以a、b、c、d、e、f、g、h为叶子结点的二叉树；
- 2) 将该二叉树所有左分枝标记1，所有右分枝标记0；
- 3) 从根到叶子结点路径上标记作为叶子结点所对应字符的编码；

## 6.7 哈夫曼树及应用

如何得到使二进制  
串总长最短编码

构造以字符使用频率  
作为权值的哈夫曼树



a: 0110

b: 10

c: 1110

d: 1111

e: 110

f: 00

g: 0111

h: 010

# 第五章结束