数据结构与算法

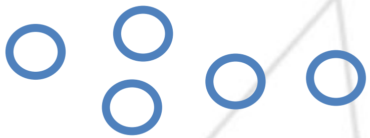
陈越编写

高等教育出版社出版

讲解人：张家琦

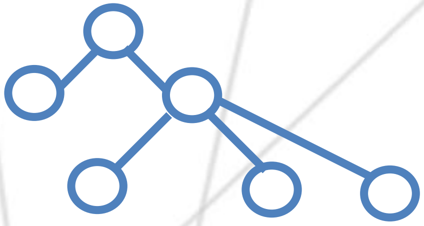
日　期：2022.09

- 集合——数据元素间除“同属于一个集合”外，无其它关系

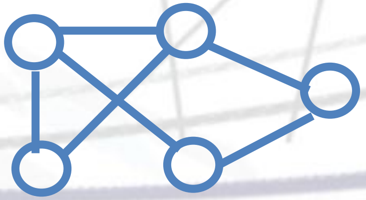


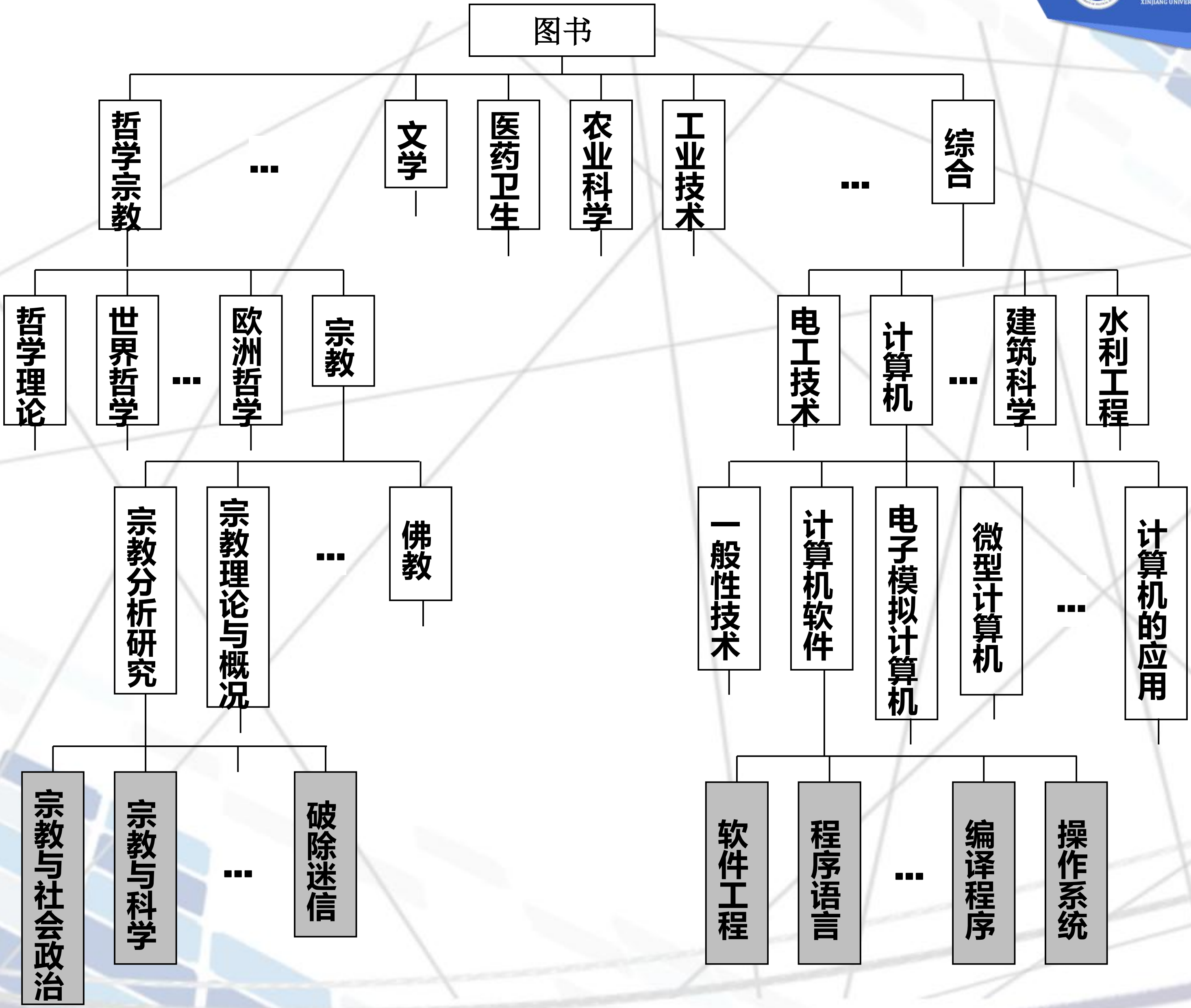
- 线性结构——一个对一个，如线性表、栈、队列

- 树形结构——一个对多个，如树



- 图形结构——多个对多个，如图




图书
哲学宗教
…
文学
医药卫生
农业科学
工业技术
…
综合
哲学理论
世界哲学
…
欧洲哲学
宗教
电工技术
计算机
…
建筑科学
水利工程
宗教分析研究
宗教理论与概况
…
佛教
一般性技术
计算机软件
电子模拟计算机
微型计算机
…
计算机的应用
宗教与社会政治
宗教与科学
…
破除迷信
软件工程
程序语言
…
编译程序
操作系统

❖ 查找（Searching）的定义：

根据某个给定的关键字*K*，从集合*R*中找出关键字与*K*相同的记录，这个过程称为“查找”。

- 查找可分静态和动态两种情况考虑；
- 手段分为利用比较和利用映射两种思路

【定义】所谓静态查找，是指集合中的记录是固定的，不涉及对记录的插入和删除操作，而仅仅是按关键字查找记录。

所谓动态查找，是指集合中的记录是动态变化的，即记录可能要发生插入和删除操作。

- 查找的效率主要用“平均查找长度”（**ASL**，**Average Search Length**）来衡量。

❖ 静态查找：通常是从一个线性表中查找数据元素

- 线性表的数组存储结构的定义：

```
typedef  struct LNode *PtrToLNode;
struct LNode{
   ElementType  Data[MAXSIZE];
   Position  Last;
};
typedef  PtrToLNode List;
```

- 线性表的链表存储结构的定义：

```
typedef  struct LNode *PtrToLNode;
struct LNode{
      ElementType  Data;
      PtrToLNode Next;
};
typedef  PtrToLNode Position;
typedef  PtrToLNode List;
```

方法1：顺序查找

```
Position SequentialSearch ( List Tbl, ElementType K )
{ /* 在顺序存储的表Tbl中查找关键字为K的数据元素 */
  Position i;
  Tbl->Data[0] = K; /*建立哨兵*/
  for( i = Tbl->Last; Tbl->Data[i] != K; i--);
  return i; /* 查找成功返回数据元素所在单元下标；查找不成功返回0 */
}
```

顺序查找算法的时间复杂度为$O(n)$。

方法2：二分查找

➢ 当线性表中数据元素是按大小排列存放时，可以改进顺序查找算法，以得到更高效率的新算法----二分法 (折半查找)。

❖ 假设n个数据元素的关键字满足有序（从小到大或从大到小）

$$k_1 < k_2 < \cdots < k_n$$

并且是连续存放（数组），那么可以进行二分查找。

❖ 二分查找是每次在要查找的数据集合中取出中间元素关键字 K_{mid} 与要查找的关键字K进行比较，根据比较结果确定是否要进一步查找。当 $K_{mid}=K$，查找成功；否则，将在 K_{mid} 的左半部分（当 $K_{mid}>K$）或者右半部分（当 $K_{mid}<K$）继续下一步查找。以此类推，每步的查找范围都将是上一次的一半。

[例4.1] 假设有13个数据元素，它们的关键字为 51，202，16，321，45，98，100，501，226，39，368，5，444。若按关键字由小到大顺序存放这13个数，二分查找关键字为444的数据元素过程如下：

5	16	39	45	51	98	100	202	226	321	368	444	501
1	2	3	4	5	6	7	8	9	10	11	12	13

left　　　　　　　　mid　　　　　　　　right

1、left = 1, right = 13; mid = (1+13)/2 = 7:   100 < 444;

2、left = mid+1=8, right = 13; mid = (8+13)/2 = 10:   321 < 444;

3、left = mid+1=11, right = 13; mid = (11+13)/2 = 12:   444 = 444查找结束;

[例4.2] 仍然以上面13个数据元素构成的有序线性表为例，二分查找关键字为 43 的数据元素如下：

5	16	39	45	51	98	100	202	226	321	368	444	501
1	2	3	4	5	6	7	8	9	10	11	12	13

left mid right

1、left = 1, right = 13; mid = (1+13)/2 = 7: 100 > 43;

2、 left = 1, right = mid-1= 6; mid = (1+6)/2 = 3: 39 < 43;

3、left = mid+1=4, right = 6; mid = (4+6)/2 = 5: 51 > 43;

4、left = 4, right = mid-1= 4; mid = (4+4)/2 = 4: 45 > 43;

5、left = 4, right = mid-1= 3; left > right ? 查找失败，结束;

❖ 二分查找算法

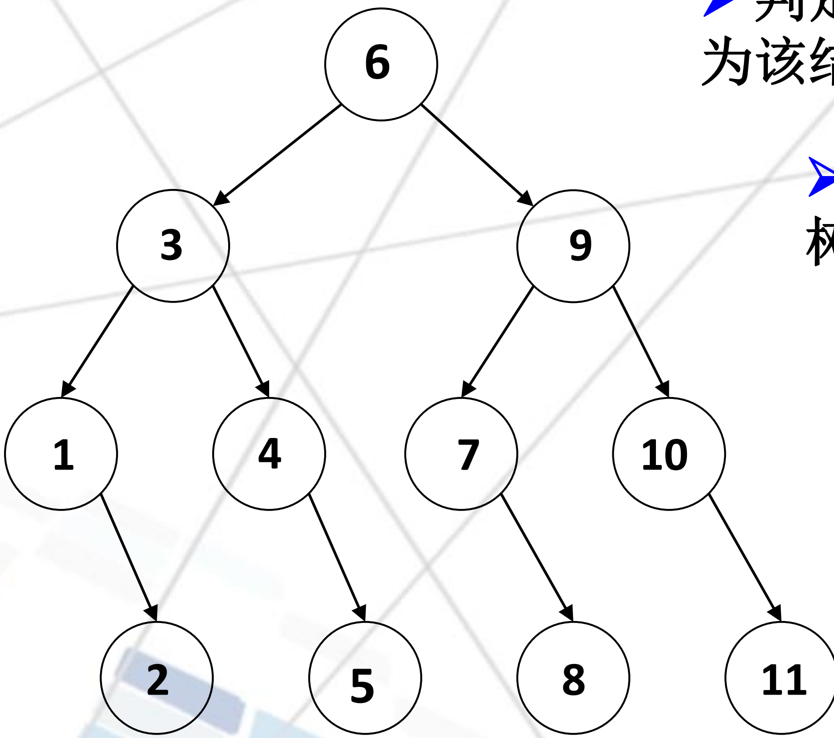
```
#define NotFound 0 /* 找不到则返回0 */

Position BinarySearch( List Tbl, ElementType K )
{ /* 在顺序存储的表Tbl中查找关键字为K的数据元素 */
    Position left, right, mid;

    left = 1;          /* 初始左边界 */
    right = Tbl->Last; /* 初始右边界 */
    while( left<=right )
    {
        mid = (left+right)/2;  /* 计算中间元素坐标 */
        if( K<Tbl->Data[mid] )      right = mid - 1; /* 调整右边界 */
        else if( K>Tbl->Data[mid] ) left = mid + 1;  /* 调整左边界 */
        else return mid;   /* 查找成功，返回数据元素的下标 */
    }
    return NotFound;   /* 返回查找不成功的标识 */
}
```

➢ 二分查找算法具有对数的时间复杂度O(logN)

❖ 11个元素的二分查找判定树



11个元素的判定树

- 判定树上每个结点需要的查找次数刚好为该结点所在的层数;
- 查找成功时查找次数不会超过判定树的深度
- ASL = (4*4+4*3+2*2+1)/11 = 3
- n个结点的判定树的深度为$[\log_2 n]+1$.
- 折半查找的算法复杂度为$O(\log_2 n)$

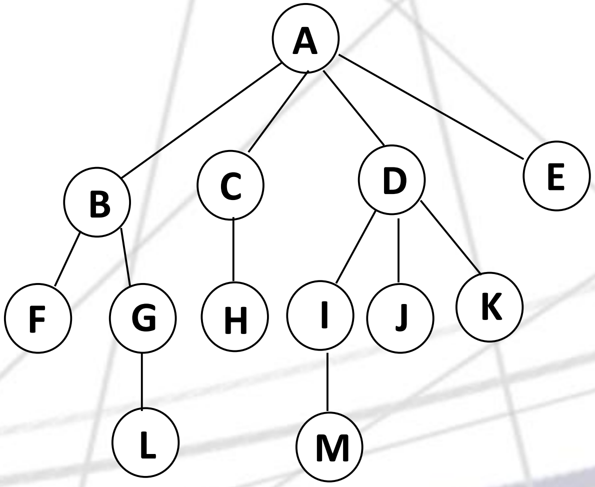
❖ 树的定义

➢ 树（Tree）是n（n≥0）个结点构成的有限集合。当n=0时，称为空树；对于任一棵非空树（n> 0），它具备以下性质：

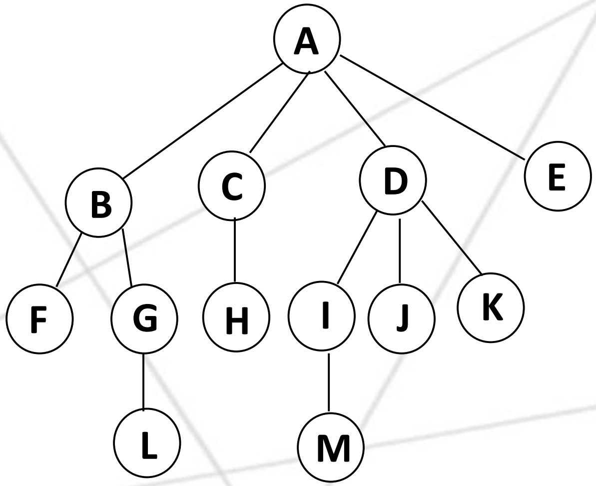
1. 树中有一个称为“根（Root）”的特殊结点，用 *r* 表示；

2. 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，其中每个集合本身又是一棵树，这些树称为原来树的“子树（SubTree）”。每个子树的根结点都与 *r* 有一条相连接的边，*r* 是这些子树根结点的“父结点(Parent)”。

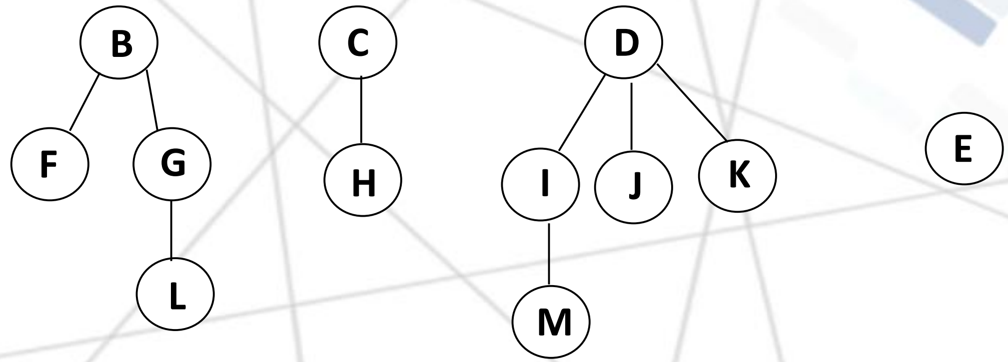
➢ 子树是不相交的；
➢ 除了根结点外，每个结点有且仅有一个父结点；
➢ 一棵N个结点的树有N-1条边。



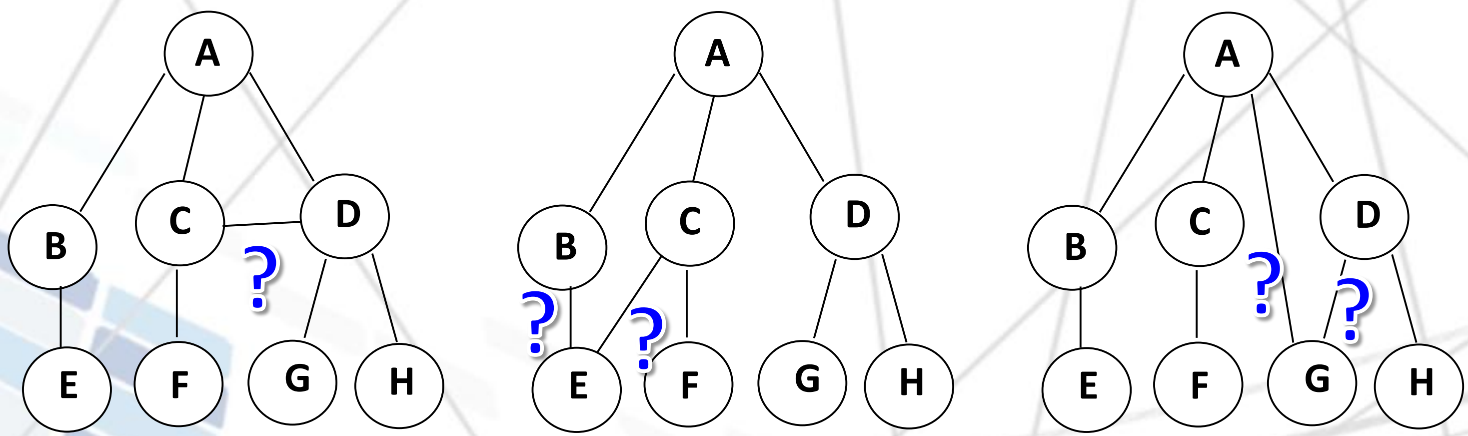
❖ 树与非树



(a) 树*T*

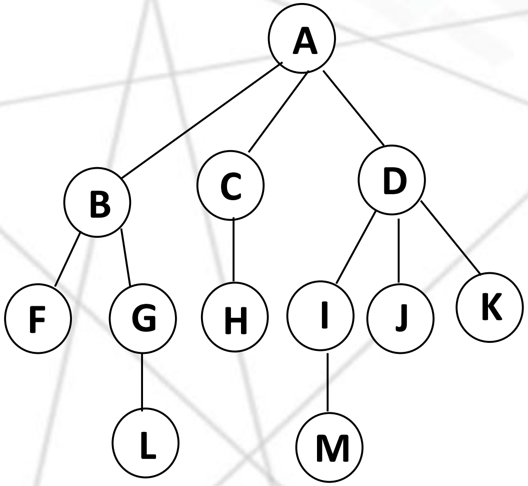


(b) 子树T_{A1}　(c) 子树T_{A2}　(d) 子树T_{A3}　(e)子树T_{A4}



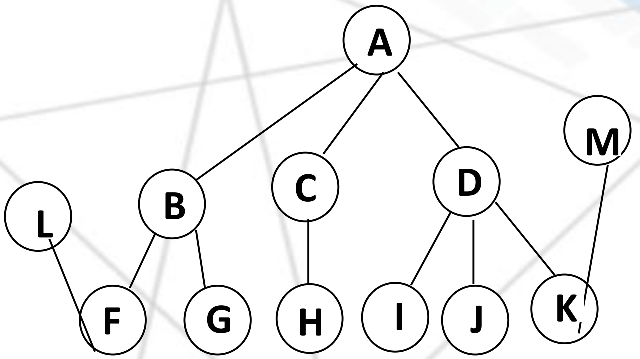
❖ 树的一些基本术语

1. 结点的度（Degree）：一个结点的度是其子树的个数。
2. 树的度：树的所有结点中最大的度数。
3. 叶结点（Leaf）：是度为0的结点；叶结点也可称为端结点。
4. 父结点（Parent）：有子树的结点是其子树的根结点的父结点。
5. 子结点（Child）：若A结点是B结点的父结点，则称B结点是A结点的子结点；子结点也称孩子结点。
6. 兄弟结点（Sibling）：具有同一父结点的各结点彼此是兄弟结点。



❖ 树的一些基本术语

7. 分支：树中两个相邻结点的连边称为一个分支。
8. 路径和路径长度：从结点n_1到n_k的路径被定义为一个结点序列$n_1, n_2, \dots, n_k$，对于$1 \le i \le k$, n_i是n_{i+1}的父结点。一条路径的长度为这条路径所包含的边（分支）的个数。
9. 祖先结点(Ancestor)：沿树根到某一结点路径上的所有结点都是这个结点的祖先结点。
10. 子孙结点(Descendant)：某一结点的子树中的所有结点是这个结点的子孙。
11. 结点的层次（Level）：规定根结点在1层，其它任一结 点的层数是其父结点的层数加1。
12. 树的高度（Height）：树中所有结点中的最大层次是这棵树的高度。（也有把根定义成高度为1的）

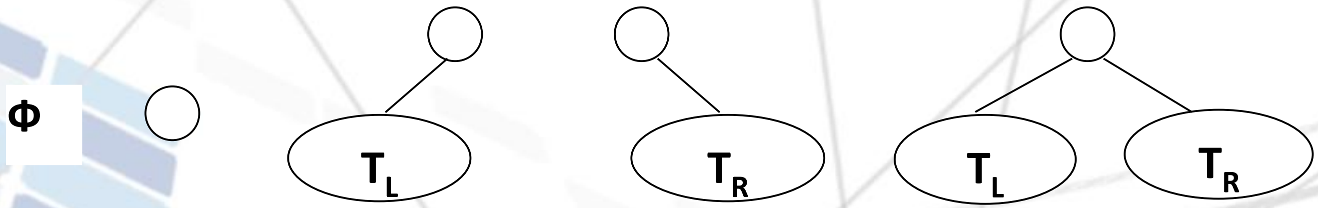


❖ 二叉树的定义

【定义】一棵二叉树T是一个有穷的结点集合。这个集合可以为空，若不为空，则它是由根结点和称为其左子树T_L和右子树T_R的两个不相交的二叉树组成。可见左子树和右子树还是二叉树。

➢ 二叉树具体五种基本形态

（1）空二叉树；
（2）只有根结点的二叉树；
（3）只有根结点和左子树T_L的二叉树；
（4）只有根结点和右子树T_R的二叉树；
（5）具有根结点、左子树T_L和右子树T_R的二叉树。

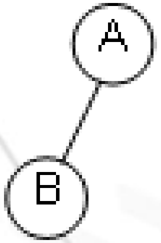


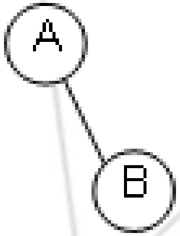
(a) (b) (c) (d) (e)

❖ 二叉树的定义

【定义】一棵二叉树T是一个有穷的结点集合。这个集合可以为空，若不为空，则它是由根结点和称为其左子树T_L和右子树T_R的两个不相交的二叉树组成。可见左子树和右子树还是二叉树。

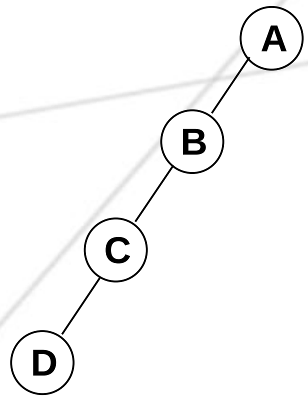
- 二叉树与树不同，除了每个结点至多有两棵子树外，子树有左右顺序之分。
- 例如，下面两个树按一般树的定义它们是同一个树；而对于二叉树来讲，它们是不同的两个树。



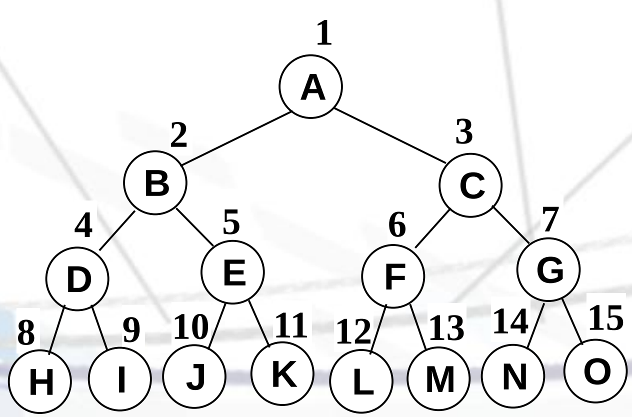


❖ 特殊二叉树

- 二叉树的深度小于结点数*N*，可以证明平均深度是 $O(\sqrt{N})$
- “斜二叉树(Skewed Binary Tree)”（也称为退化二叉树）；

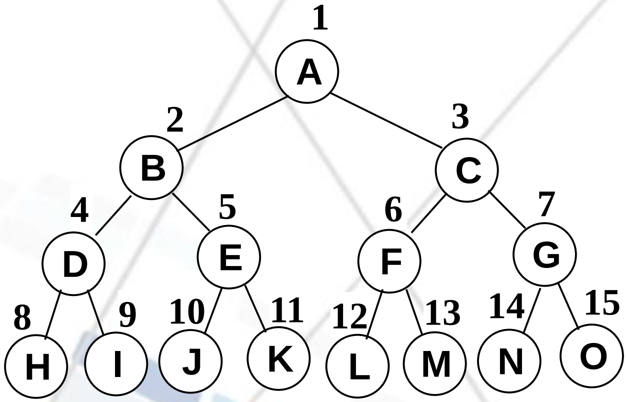


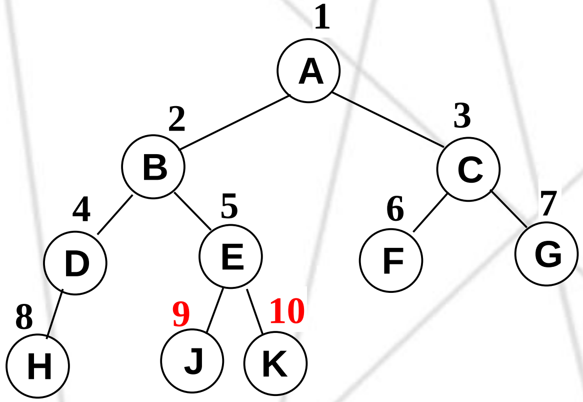
- “完美二叉树(Perfect Binary Tree)”。（也称为满二叉树）。

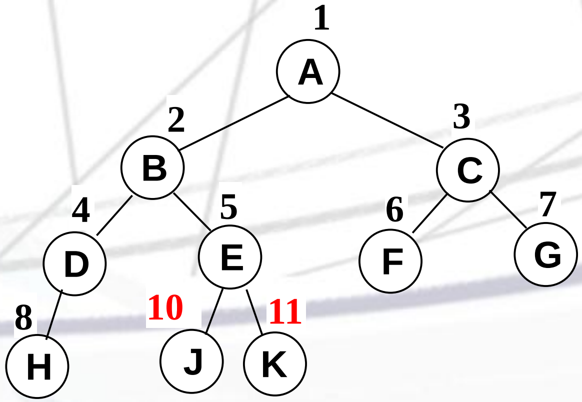


❖ 特殊二叉树

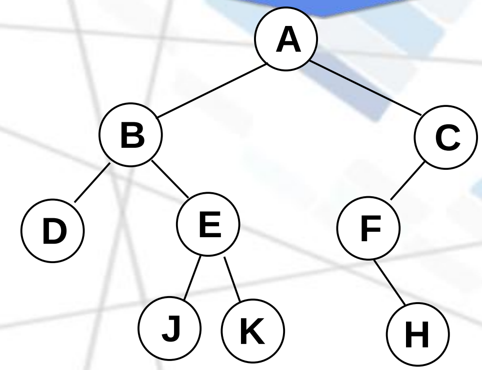
➢ 一棵深度为k的有n个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为i（1 ≤ i ≤ n）的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，则这棵二叉树称为“完全二叉树(Complete Binary Tree)”。







❖ 二叉树的几个重要的性质



- 一个二叉树第 i 层的最大结点数为：2^{i-1}，$i \geq 1$。
- 深度为k的二叉树有最大结点总数为：$2^{k}-1$，$k \geq 1$。
- n个结点的完全二叉树的深度为k 为：$\lfloor \log_2 n \rfloor + 1$
- 对任何非空的二叉树 T，若n_0表示叶结点的个数、n_2是度为2的非叶结点个数，那么两者满足关系$n_0 = n_2 + 1$。

- $n_0 = 4$，$n_1 = 2$，
- $n_2 = 3$；
- $n_0 = n_2 + 1$

证明：设 n_1 是度为1结点数, n 是总的结点数. 那么 ①

$$n = n_0 + n_1 + n_2$$

设 B 是全部分枝数. 则 $n \sim B$? $n = B + 1.$ ②

因为所有分枝都来自度为1或2的结点, 所以 $B \sim n_1 \,\&\, n_2$?

$B = n_1 + 2\,n_2.$ ③

$$\Rightarrow \quad n_0 = n_2 + 1$$

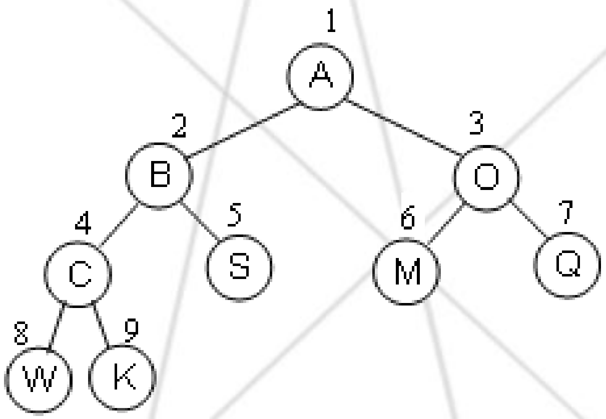
❖ 二叉树的存储结构

1. 顺序存储结构

- 完全二叉树最适合这种存储结构。
- n个结点的完全二叉树的结点父子关系，简单地由序列号决定：

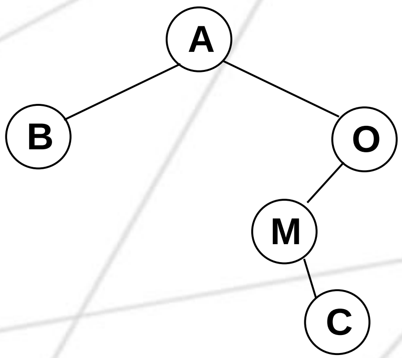
数据	A	B	O	C	S	M	Q	W	K
编号	1	2	3	4	5	6	7	8	9

(b)相应的顺序存储结构

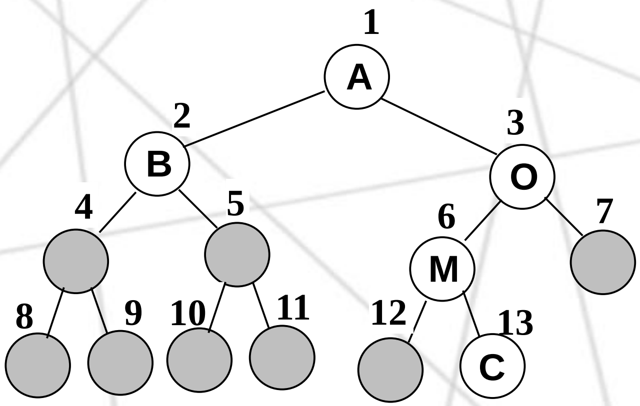


(a) 完全二叉树

➢一般二叉树最适合采用这种结构将造成空间浪费……



(a)一般二叉树



(b) 对应(a)的完全二叉树

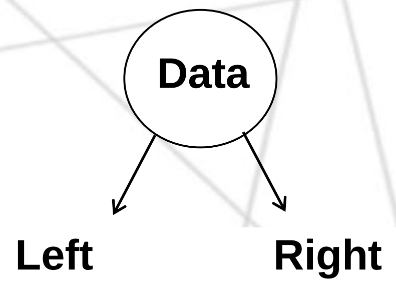
数据	A	B	O	∧	∧	M	∧	∧	∧	∧	∧	∧	C
编号	1	2	3	4	5	6	7	8	9	10	11	12	13

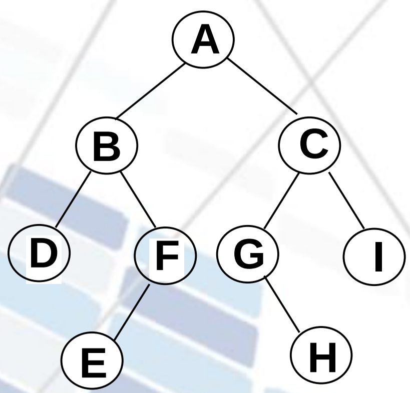
造成空间严重浪费！

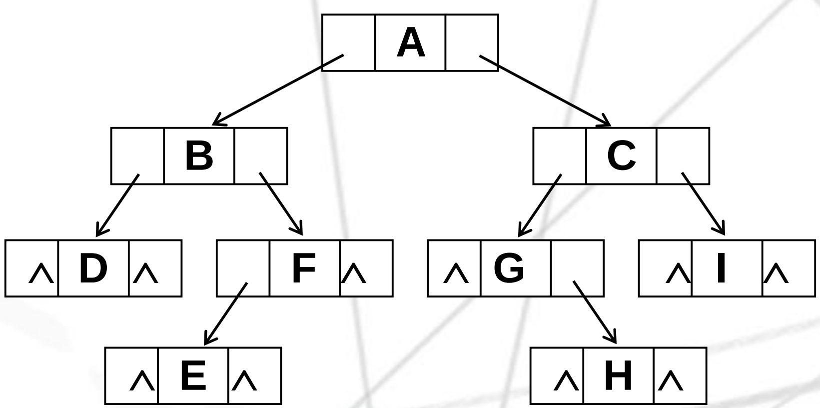
2. 二叉树的链表存储

Left	Data	Right

```
typedef struct TNode *Position;
typedef Position BinTree; /* 二叉树类型 */
struct TNode{ /* 树结点定义 */
        ElementType Data; /* 结点数据 */
        BinTree Left;     /* 指向左子树 */
        BinTree Right;    /* 指向右子树 */
};
```







【定义】“二叉树(Binary Trees)”抽象数据类型定义。

类型名称：二叉树（BinTree）

数据对象集：一个有穷的结点集合。这个集合可以为空，若不为空，则它是由根结点和其左、右二叉子树组成。

操作集：对于所有 BT∈ BinTree，重要的操作有：

1、bool IsEmpty(BinTree BT)： 若BT为空返回true; 否则返回false；

2、void Traversal(BinTree BT)：二叉树的遍历，即按某一顺序访问二叉树中的每个结点仅一次；

3、BinTree CreatBinTree( )：创建一个二叉树。

常用的遍历方法有：

1、void InOrderTraversal(BinTree BT)：根结点的访问次序在左、右子树之间；

2、void PreOrderTraversal(BinTree BT)：根结点的访问次序在左、右子树之前；

3、 void PostOrderTraversal(BinTree BT)：根结点的访问次序在左、右子树之后 。

4、 void LevelOrderTraversal(BinTree BT)：按层从小到大、从左到右的次序遍历。

1. 二叉树的遍历

❖ 树的遍历是指访问树的每个结点，且每个结点仅被访问一次。

二叉树的遍历可按二叉树的构成以及访问结点的顺序分为四种方式，即先序遍历、中序遍历、后序遍历和层次遍历。

（1） 中序遍历

其遍历过程为：

① 中序遍历其左子树；

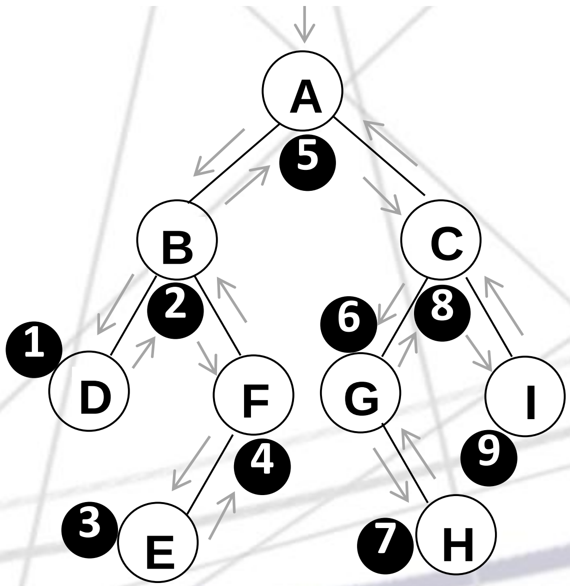
② 访问根结点；

③ 中序遍历其右子树。

(D B E F) **A** *(G H C I)*

中序遍历=> D B E F A G H C I

```
void InOrderTraversal( BinTree BT )
{
    if( BT ) {
        InOrderTraversal( BT->Left );
        printf("%d", BT->Data);
        InOrderTraversal( BT->Right );
    }
}
```



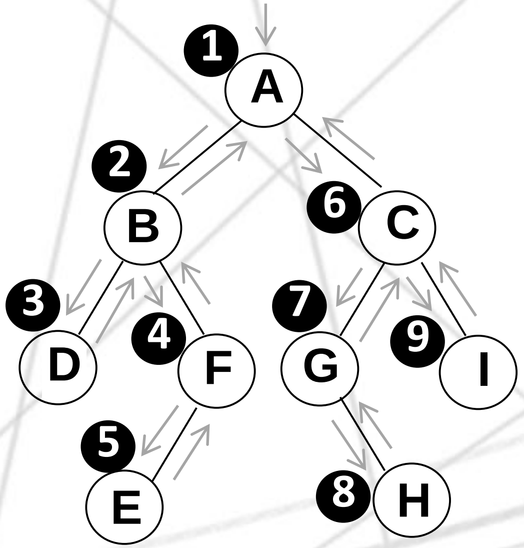
（2）　先序遍历

其遍历过程为：
① 访问根结点；
② 先序遍历其左子树；
③ 先序遍历其右子树。

A （B D F E ） （C G H I）

先序遍历=>　A B D F E C G H I

```
void PreOrderTraversal( BinTree BT )
{
    if( BT ) {
        printf("%d", BT->Data);
        PreOrderTraversal( BT->Left );
        PreOrderTraversal( BT->Right );
    }
}
```



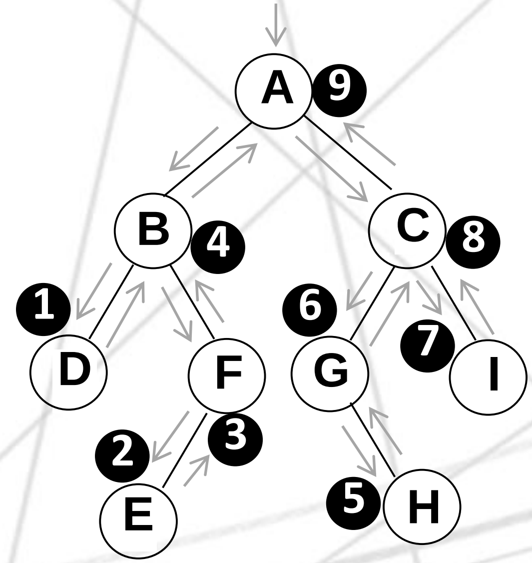
（3） 后序遍历

其遍历过程为：
① 后序遍历其左子树；
② 后序遍历其右子树；
③ 访问根结点。

（D E F B ） （H G I C） A

后序遍历=> D E F B H G I C A

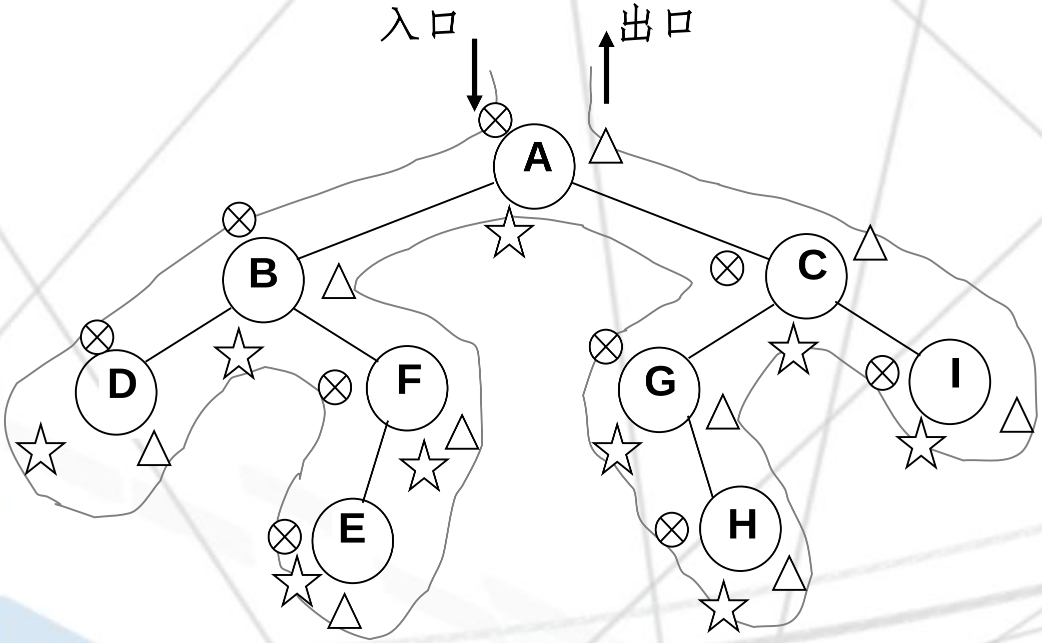
```
void PostOrderTraversal( BinTree BT )
{
    if( BT ) {
        PostOrderTraversal( BT->Left );
        PostOrderTraversal( BT->Right);
        printf("%d", BT->Data);
    }
}
```



（4） 二叉树的非递归遍历

❖ 从二叉树先序、中序和后序的遍历过程的遍历路径来看，都是从根结点A开始的，且在遍历过程中经过结点的路线是一样的，只是访问各结点的时机不同而已。

❖ 在图4.16中，并在从入口到出口的曲线上用⊗、☆ 和△三种符号分别标记出了先序、中序和后序遍历各结点的时刻。



❖中序遍历非递归遍历算法

- 遇到一个结点，就把它压栈，并去遍历它的左子树；
- 当左子树遍历结束后，从栈顶弹出这个结点并访问它；
- 然后按其右指针再去中序遍历该结点的右子树。

```
void InorderTraversal( BinTree BT )
{ BinTree T;
  Stack S = CreateStack();
  T = BT; /* 从根结点出发 */
  while( T || !IsEmpty(S) ){
       while( T ){ /* 一直向左并将沿途结点压入堆栈 */
              Push(S, T);
              T = T->Left;
       }
       T = Pop(S); /* 结点弹出堆栈 */
       printf("%d ", T->Data); /*（访问）打印结点 */
       T = T->Right; /* 转向右子树 */
  }
}
```

❖先序遍历的非递归遍历算法

- 遇到一个结点，就把它压栈并访问它，然后去遍历它的左子树；
- 当左子树遍历结束后，从栈顶弹出这个结点；
- 然后按其右指针再去中序遍历该结点的右子树。

❖后序遍历非递归遍历算法

- 遇到一个结点，就把它（附带标志0以后）压栈，并去遍历它的左子树；
- 当左子树遍历结束后，检查栈顶元素的附带标志是否为0；
- 若标志为0，则把标志改成1，并按其右指针再去遍历该结点的右子树；
- 若标志为1，则从栈顶弹出这个结点并访问它。

（5） 层序遍历

❖ 具体的算法实现可以设置一个队列结构，遍历从根结点开始，首先将根结点指针入队，然后开始执行下面三个操作（直到队列空）：

① 从队列中取出一个元素；
② 访问该元素所指结点；
③ 若该元素所指结点的左、右孩子结点非空，则将其左、右孩子的指针顺序入队。

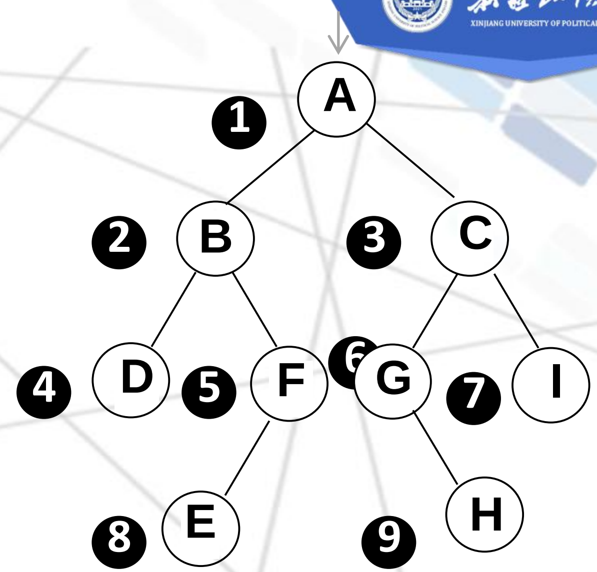
```
void LevelorderTraversal ( BinTree BT )
{ Queue Q;  BinTree T;
  if ( !BT ) return; /* 若是空树则直接返回 */
  Q = CreatQueue(); /* 创建空队列Q */
  AddQ( Q, BT );
  while ( !IsEmpty(Q) ) {
       T = DeleteQ( Q );
       printf("%d ", T->Data); /*访问取出队列的结点*/
       if ( T->Left )   AddQ( Q, T->Left );
       if ( T->Right )  AddQ( Q, T->Right );
  }
}
```

（5） 层序遍历

工作队列：

A B C D F ……

层序遍历 => A B C D F G I E H



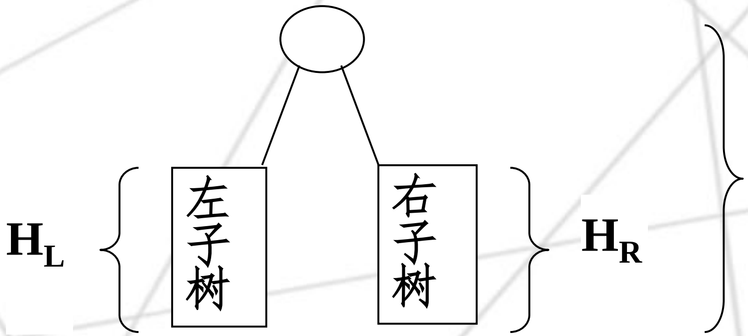

```
void LevelorderTraversal ( BinTree BT )
{ Queue Q;  BinTree T;
  if ( !BT ) return; /* 若是空树则直接返回 */
  Q = CreatQueue(); /* 创建空队列Q */
  AddQ( Q, BT );
  while ( !IsEmpty(Q) ) {
       T = DeleteQ( Q );
       printf("%d ", T->Data); /*访问取出队列的结点*/
       if ( T->Left )   AddQ( Q, T->Left );
       if ( T->Right )  AddQ( Q, T->Right );
  }
}
```

【例4.3】遍历二叉树的应用：输出二叉树中的叶子结点。

- 在二叉树的遍历算法中增加检测结点的“左右子树是否都为空”。

```
void PreOrderPrintLeaves( BinTree BT )
{
    if( BT ) {
        if ( !BT-Left && !BT->Right )
            printf(“%d”, BT->Data );
        PreOrderPrintLeaves ( BT->Left );
        PreOrderPrintLeaves ( BT->Right );
    }
}
```

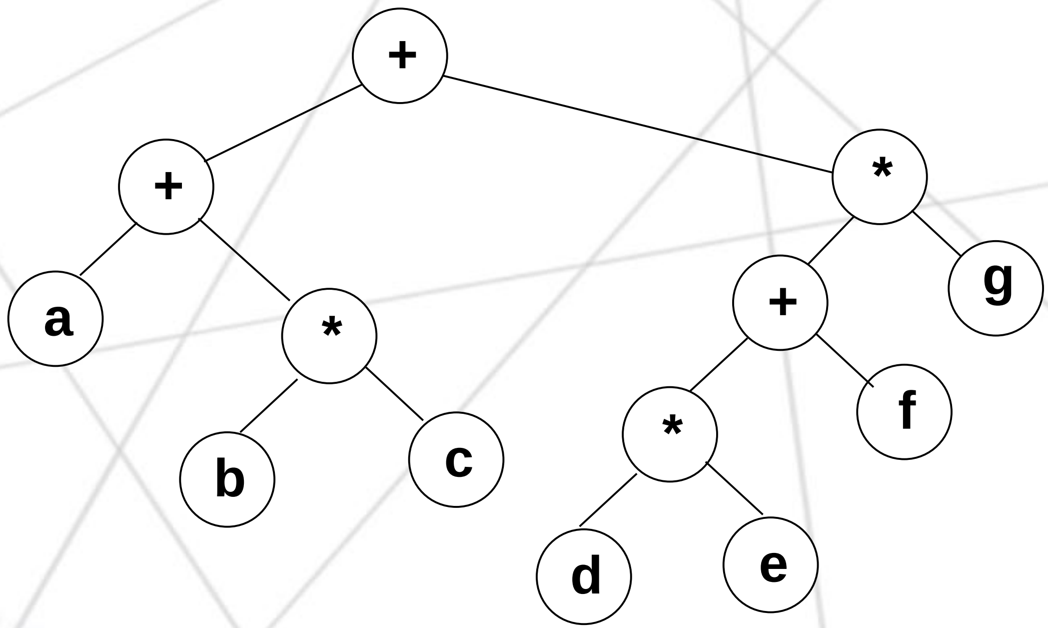
【例4.4】求二叉树的高度。



$Height=max(H_L, H_R)+1$

```
int GetHeight( BinTree BT )
{ int HL, HR, MaxH;
  if( BT ) {
        HL = GetHeight(BT->Left);    /* 求左子树的高度 */
        HR = GetHeight(BT->Right);   /* 求右子树的高度 */
        MaxH = HL > HR ? HL : HR;    /* 取左右子树较大的高度 */
        return ( MaxH + 1 );         /* 返回树的高度 */
  }
  else  return 0; /* 空树高度为0 */
}
```

【例4.5 】 二元运算表达式树及其遍历



❖ 三种遍历可以得到三种不同的访问结果：

➢ 中序遍历得到中缀表达式：*a + b * c + d * e + f * g*

➢ 先序遍历得到前缀表达式：*+ + a * b c * + * d e f g*

➢ 后序遍历得到后缀表达式：*a b c * + d e * f + g * +*

【例4.6 】 由两种遍历序列确定二叉树

已知三种遍历中的任意两种遍历序列，
能否唯一确定一棵二叉树呢？

答案是：
必须要有中序遍历才行！

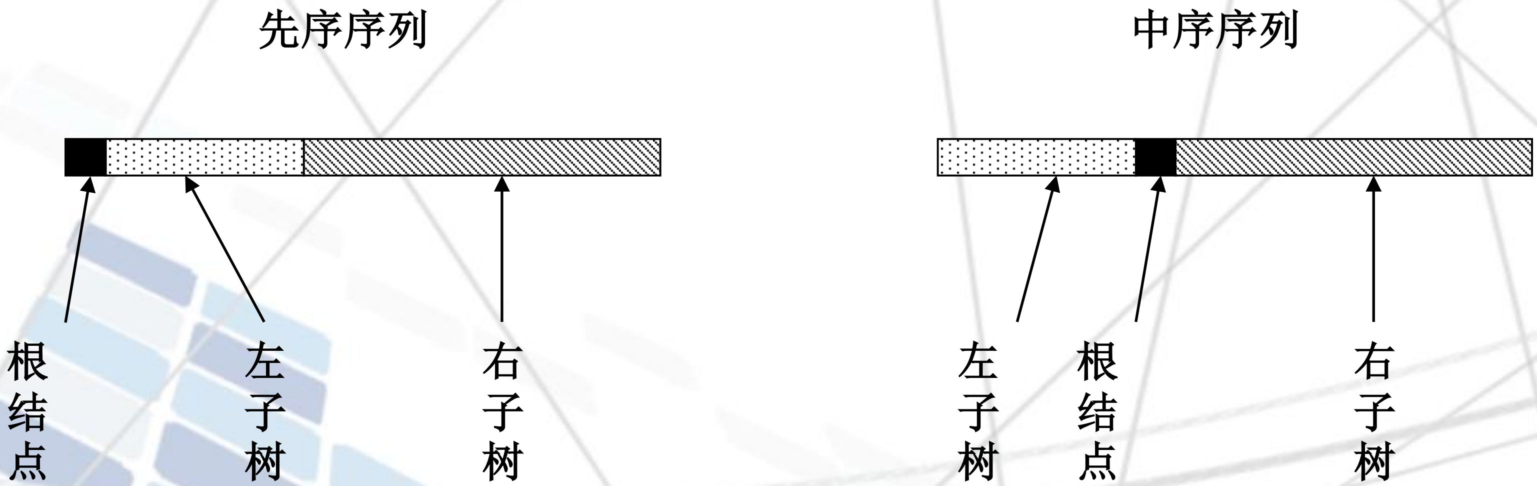
❖ 没有中序的困扰：
- ➢ 先序遍历序列：A B
- ➢ 后序遍历序列：B A

❖ 先序和中序遍历序列来确定一棵二叉树

〖分析〗先序遍历序列的第一个结点就是根结点；
这个根结点能够在中序遍历序列中将其余结点分割成两个子序列，根结点前面部分是左子树上的结点，而根结点后面的部分是右子树上的结点。

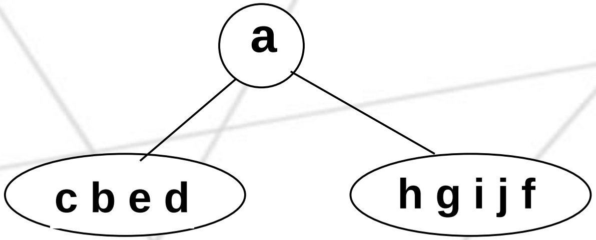
根据这两个子序列，在先序序列中找到对应的左子序列和右子序列，它们分别对应左子树和右子树。

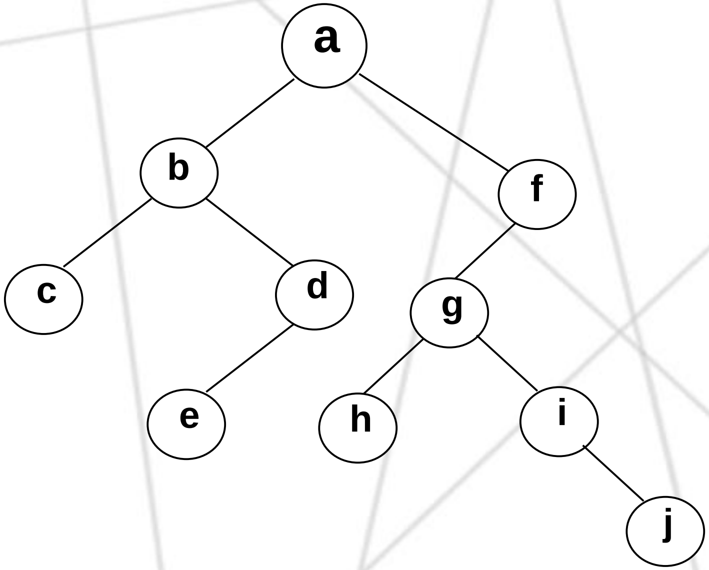
然后对左子树和右子树分别递归使用相同的方法继续分解。



〖例〗 先序序列： a $\underline{\text{b c d e}}$ $\underline{\text{f g h i j}}$

中序序列： $\underline{\text{c b e d}}$ a $\underline{\text{h g i j f}}$

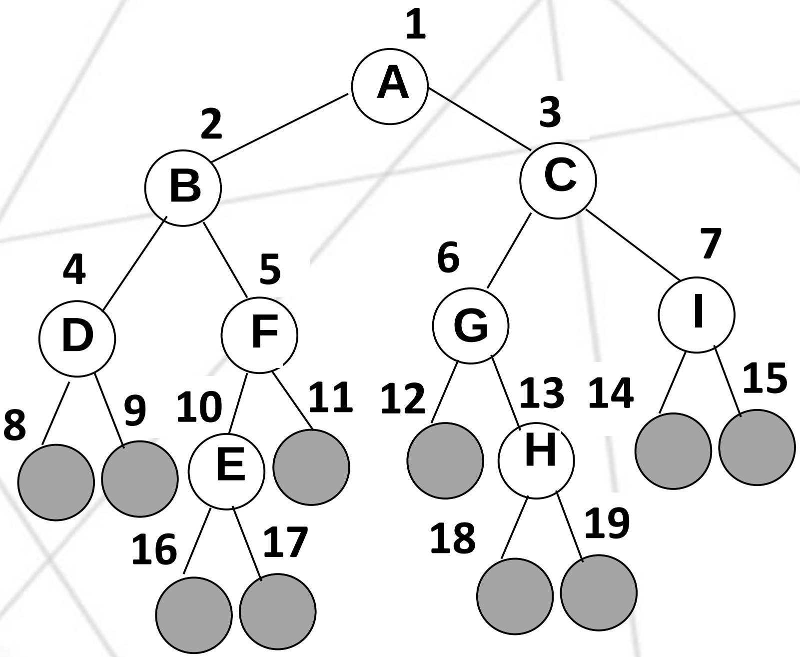




❖ 类似地，配套的后序和中序遍历序列也可以确定一棵二叉树。

2. 二叉树的创建

❖ 常有的方法是先序创建和层序创建两种。



先序创建的输入序列： A, B, D, 0, 0, F, E, 0, 0, 0, C, G, 0, H, 0, 0, I, 0, 0

层序创建的输入序列： A, B, C, D, F, G, I, 0, 0, E, 0, 0, H, 0, 0, 0, 0, 0, 0