四

算法与数据结构

第四章 树

CONTENTS

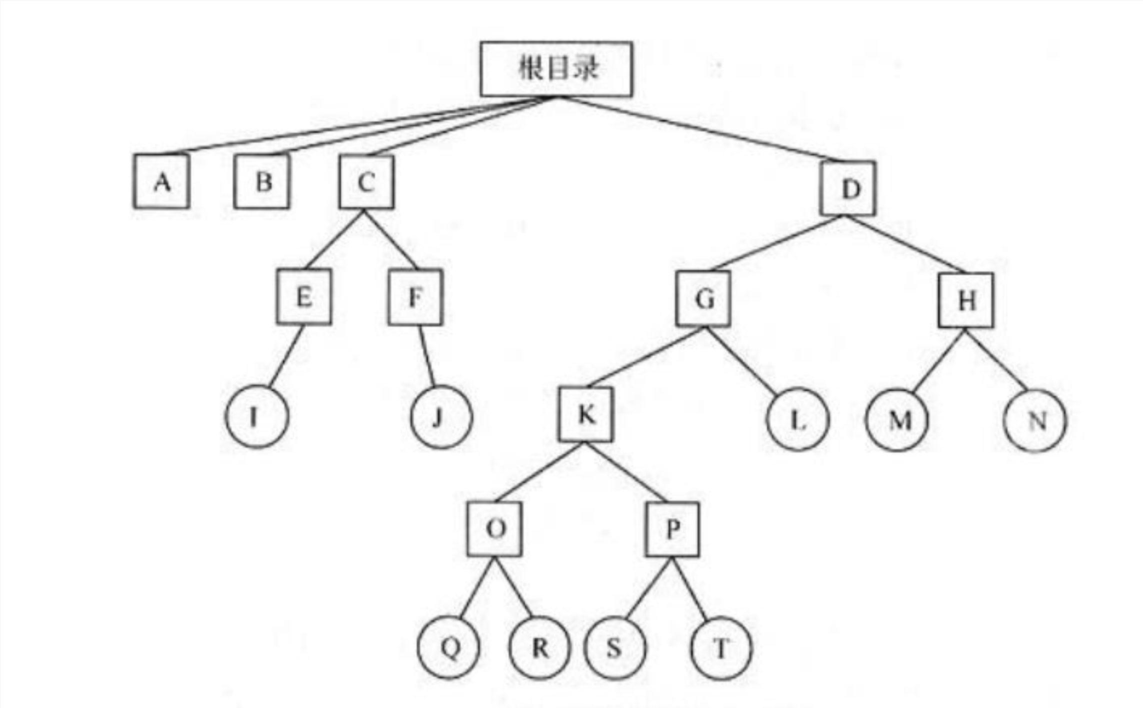
目录

引子——什么是树?

例
- 自然界：树
- 人类社会
  - 家谱
  - 行政组织机构
- 计算机领域
  - 硬盘管理：用树结构组织文件
  - 数据库系统：用树组织信息
  - 算法分析：用树描述执行过程

树型结构（非线性结构）

- **结点之间有分支**
- **具有层次关系**



分层次组织在管理上具有更高的效率！

4.1.2 查找

❖ 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”。

- 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。
  - 例如：查找字典
- 动态查找：是指集合中的记录是动态变化的
  - 除了查找，记录可能要发生插入和删除操作。

静态查找

❖ **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 {
       ElementType  *Data;
                /* 数组的起始地址 */
       int Last;
                     /* 数组长度 */
} List;
```

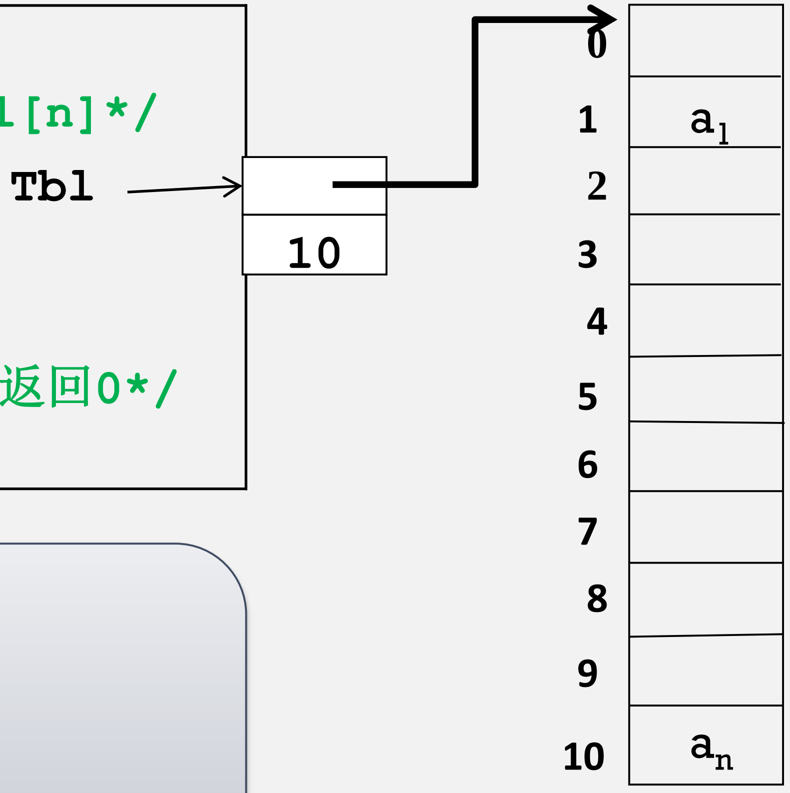
➢ 线性表的**链表**存储结构的定义：

```
typedef  struct Node *PtrToNode;
typedef  PtrToNode  List;
struct  Node {
       ElementType  data;
                     /* 结点的值域 */
       PtrToNode Next;
                /* 下一个结点的指针域*/
};
```

方法1：顺序查找

从线性表一端开始，向另一端逐个去除数据元素的关键字，与K比较，判断是否存在要找的数据元素

```
int SequentialSearch (List Tbl, ElementType K)
{    /*在表Tbl中查找关键字为K的数据元素，数据元素：Tbl[1]~Tbl[n]*/
     int 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*/
}
```



算法复杂度分析：

- 查找成功时平均查找长度为（n+1）/2
- 查找失败时平均查找长度为：n+1

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**）

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

并且是连续存放的（**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，
  - 当$K_{mid}>K$，将在K_{mid}的左半部分进行查找
  - 当$K_{mid}<K$，将在K_{mid}的右半部分继续查找。
- 循环上述过程，直到子集(left>right)为空为止。

[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。

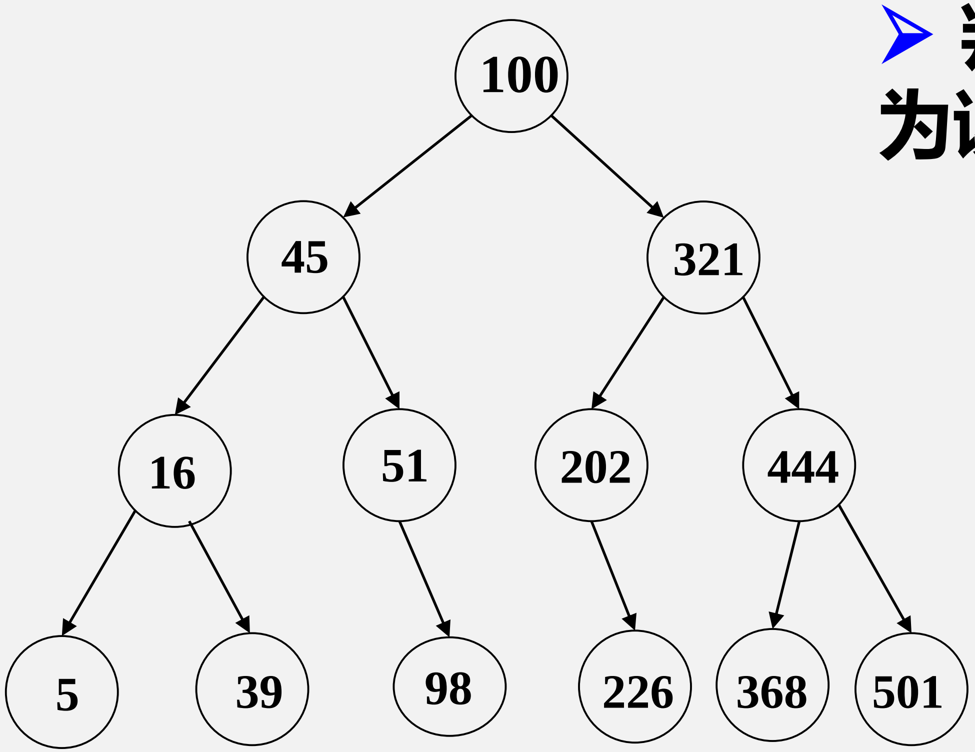
二分查找算法

```
int  BinarySearch ( List  Tbl, ElementType K)
{   /*在表Tbl中查找关键字为K的数据元素*/
    int left, right, mid, NotFound=-1;

    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;           /*查找不成功，返回-1*/
}
```

二分查找算法的时间复杂度为O(logN)

❖ 13个元素的二分查找树



13个元素的二叉搜索树

- 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 = (6*4+4*3+2*2+1)/13 = 3
    - n个结点的树的深度为$[\log_2 n]+1$
    - 折半查找的算法复杂度为$O(\log_2 n)$

4.2 树的基本概念

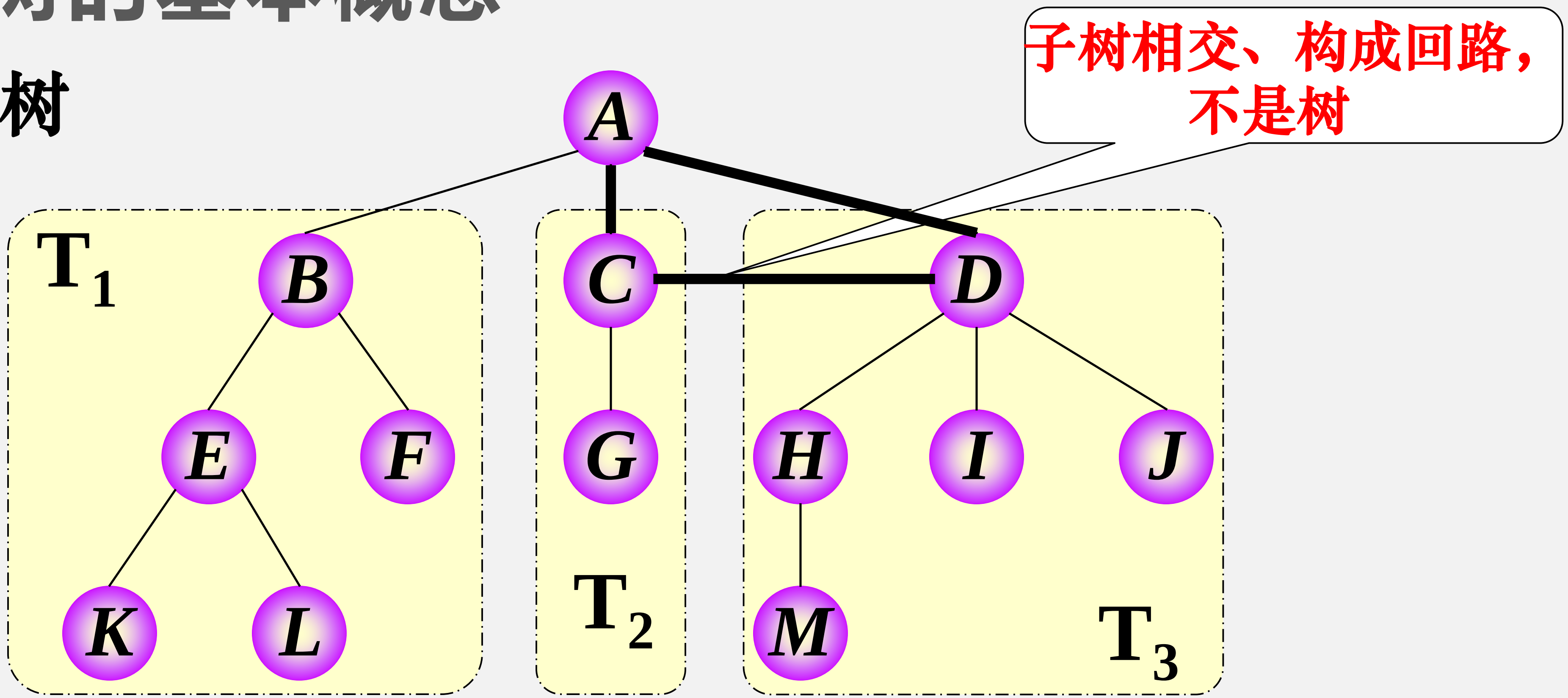
- 定义：

树 (Tree) 是 n ($n\geq 0$) 个结点的有限集。

- 若 $n = 0$，称为空树；
- 若 $n > 0$，则它满足如下两个条件：
  - (1) 有且仅有一个特定的称为根 (Root) 的结点；
  - (2) 其余结点可划分为 m ($m\geq 0$) 个互不相交的有限集 $T_1, T_2, \ldots, T_m$ 其中每一个集合本身又是一棵树，并称为根的子树 (SubTree)，每棵子树的根结点与Root都有一条相连接的边。

4.2 树的基本概念

- 树与非树



➢ 树的定义是一个递归的定义。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。

4.2 树的基本概念

基本术语

结　点： 即树的数据元素。

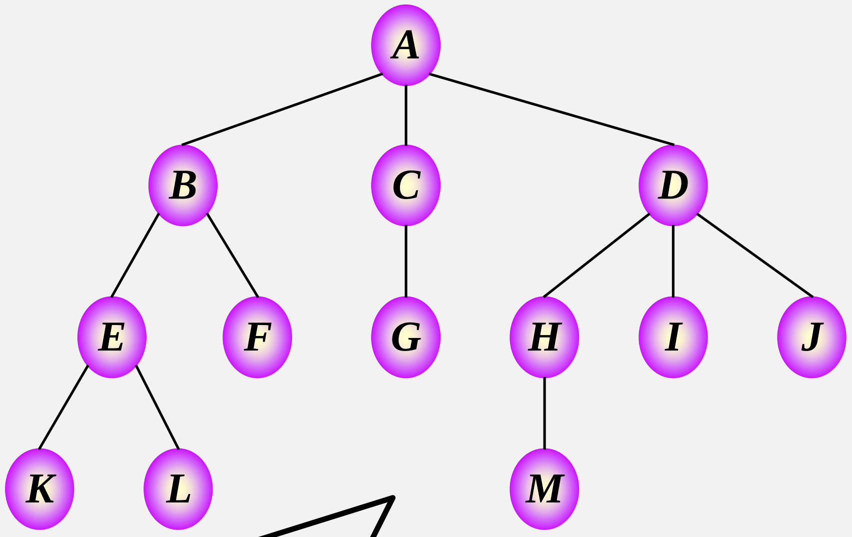
根结点： 非空树中无前驱结点的结点 。

结点的度： 结点拥有的子树的个数。

度 = 0　叶结点/终端结点

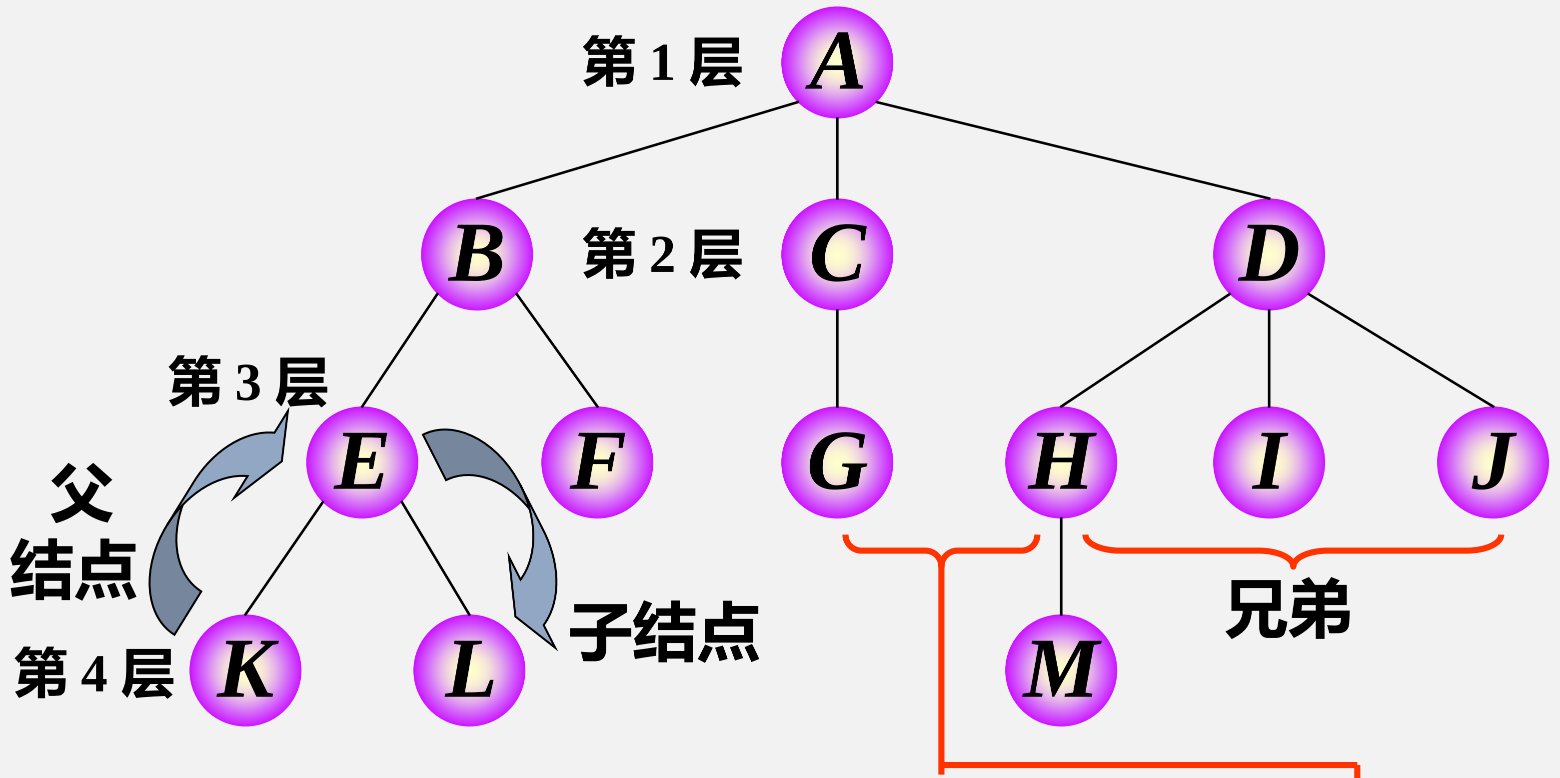
度 ≠ 0　分支结点

树的度： 树内各结点中最大的度数。



- A结点为根结点；
- K,L,F等结点为叶结点，度为0；
- B,C等为分支结点，B结点的度为2；
- 树的度为3

4.2 树的基本概念



树的逻辑结构：树中任一结点都可以有零个或多个直接后继结点

但至多只能有一个直接前趋结点。

父结点——具有子树，该结点是子树根结点的父结点(直接前驱)

子结点——子树的根结点是该结点的子结点(直接后继)

兄　弟——具有同一父节点的各结点，彼此是兄弟结点

堂兄弟——即双亲位于同一层的结点

祖　先——即从根到该结点所经路径的所有结点

子　孙——即该结点的子树中的任一结点

结点的层次：根结点的层数=1

其他任意一结点的层数=其父结点的层数+1

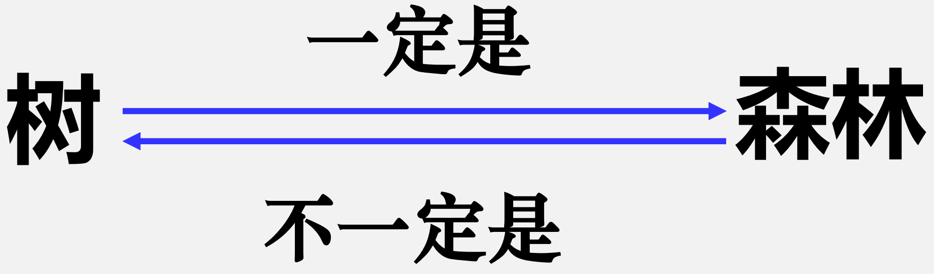
树 的 深 度：树中所有结点的最大层次

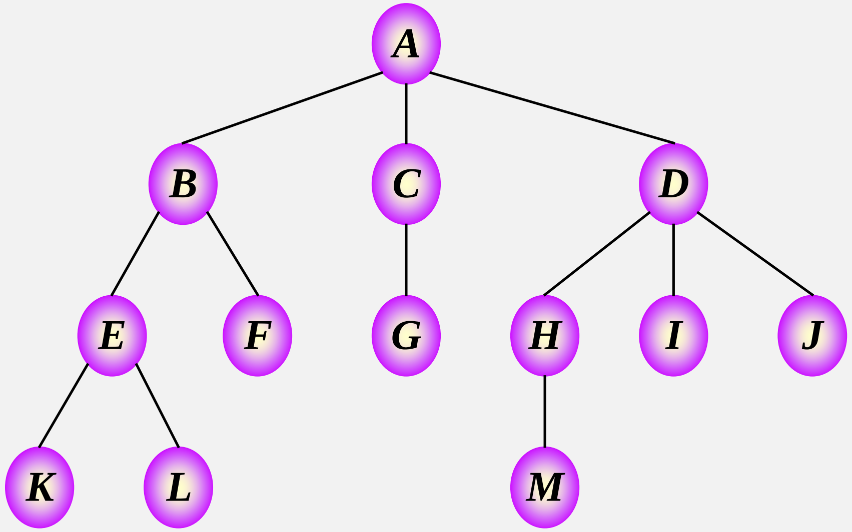
分　　　　支：树中两个相邻结点的连边称为一个分支

路　　　　径：从结点n_1到n_k的路径被定义为一个结点序列n_1, $n_2, \ldots, n_k$，对于任意i，满足n_i是n_{i+1}的父结点

路 径 长 度：是路径所包含的边的个数

- **森林：是 m ($m \geq 0$) 棵互不相交的树的集合。**
  - 一棵树可以看成是一个特殊的森林。
  - 把根结点删除树就变成了森林。
  - 给森林中的各子树加上一个双亲结点，森林就变成了树。



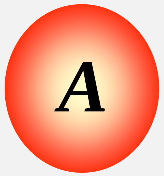


树的表示形式

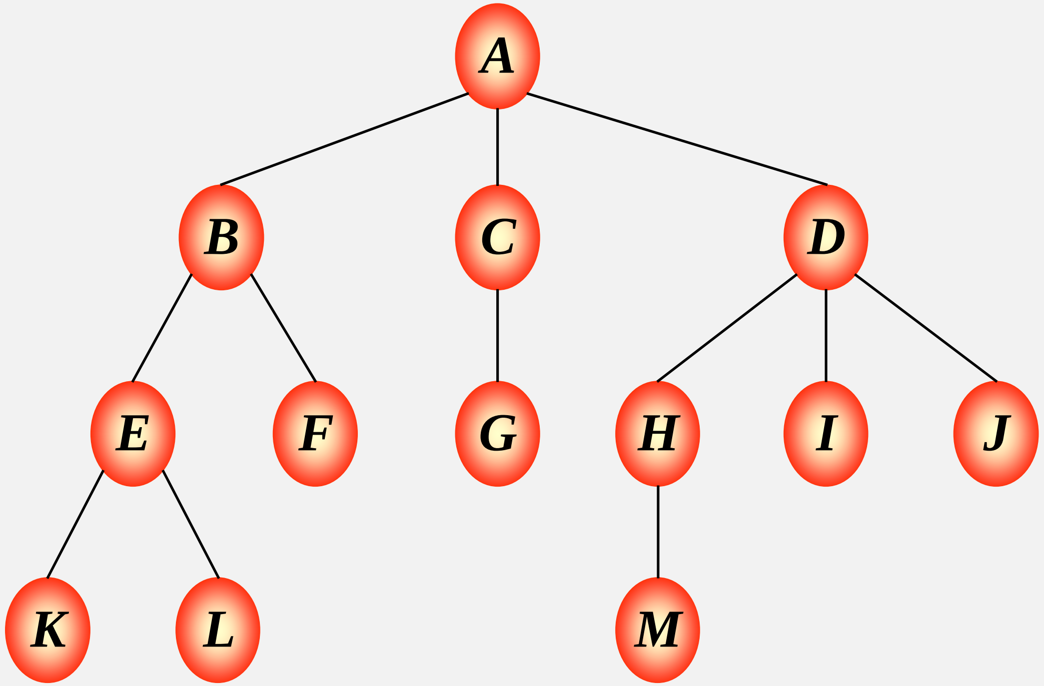
1. 树形表示法

ϕ

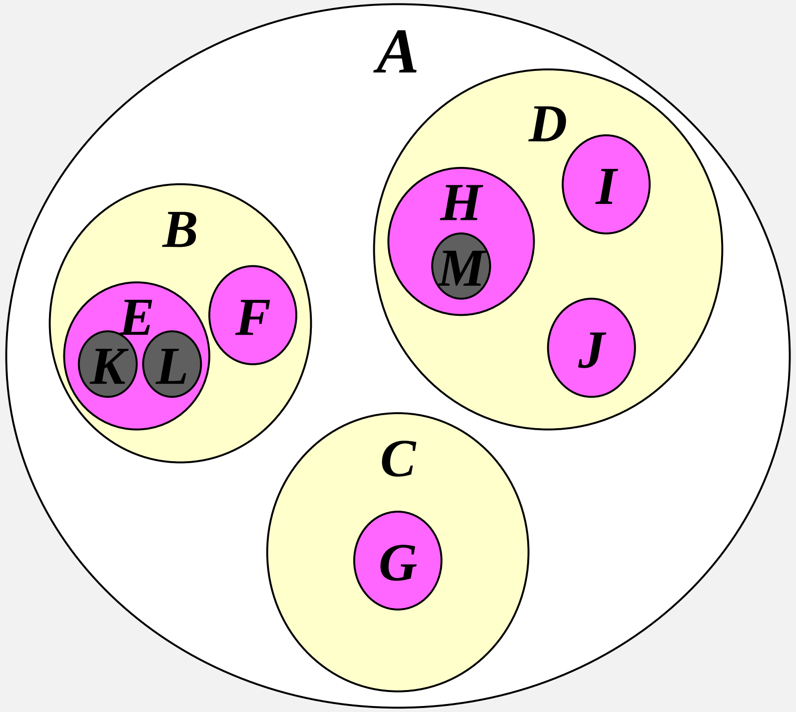
A 空树



B 仅含有根结点的树

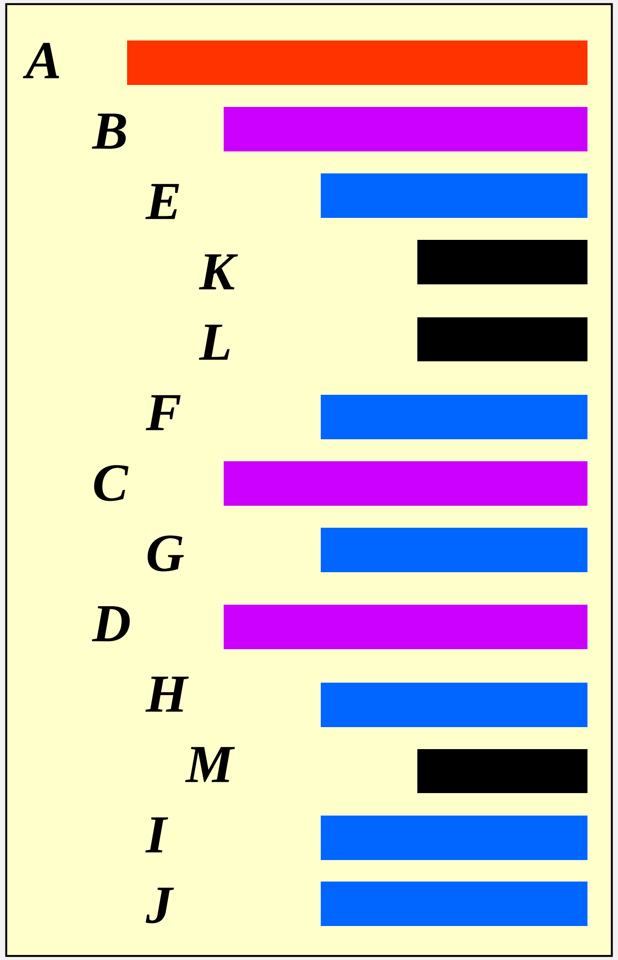


2．嵌套集合（文氏）表示法



一些集合的集体，对于其中任何两个集合，或不相交，或嵌套

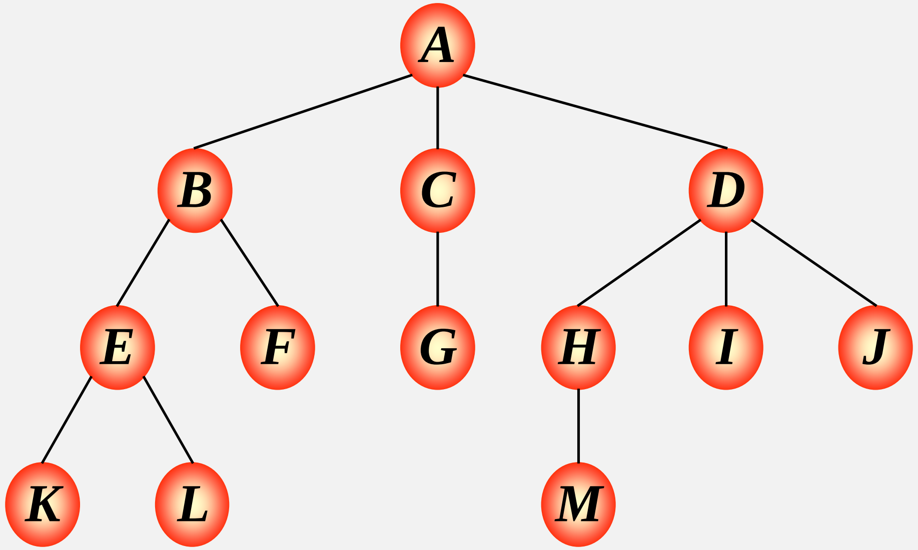
3．凹入表示法



4．广义表表示法

根作为由子树森林组成的表的名字写在表的左边，子结点作为表中元素。

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



4.3.1 二叉树的概念

- **定义**

二叉树是 n ($n \geq 0$) 个结点的有限集，

(1)集合为空 ($n = 0$)，空树；

(2)或者由一个**根结点**及**两棵互不相交**的分别称作这个根的**左子树**和**右子树**的二叉树组成。

1）二叉树的许多操作算法简单，且能有效解决树的存储结构及其运算中存在的复杂性

2）任何树都可以与二叉树相互转换

4.3.1 二叉树的概念

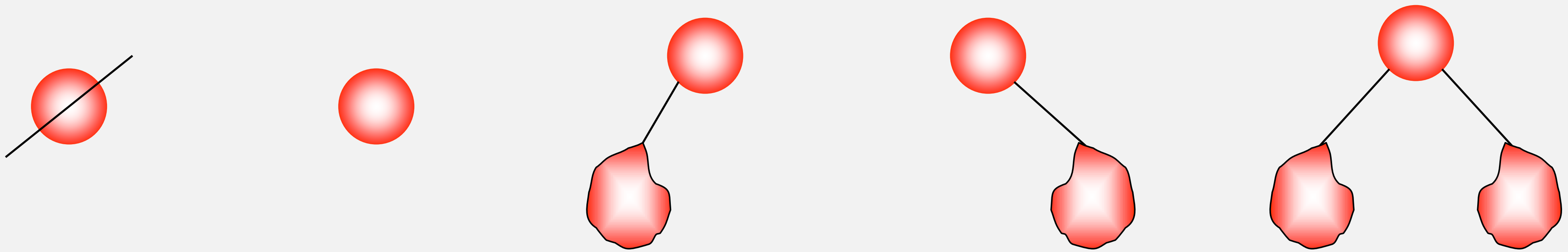
- **特征**

1、每个结点最多有俩孩子（二叉树中不存在度大于 2 的结点）。

2、子树有左右之分，其次序不能颠倒。

3、二叉树可以是空集合，根可以有空的左子树或空的右子树。

二叉树的 5 种基本形态



(*a*) 空二叉树

(*b*) 根和空的左右子树

(*c*) 根和左子树

(*d*) 根和右子树

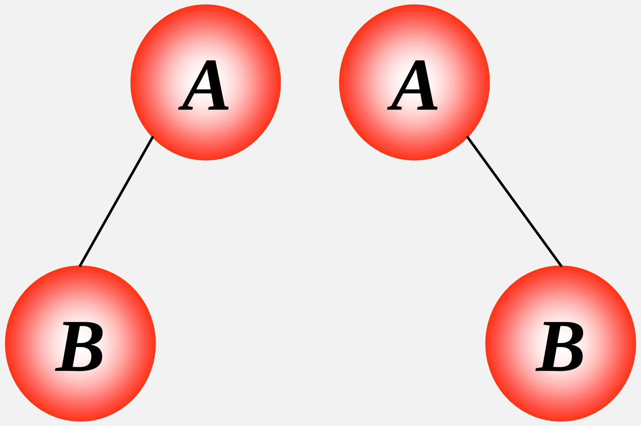
(*e*) 根和左右子树

注：虽然二叉树与树概念不同，
但有关树的基本术语对二叉树都适用。

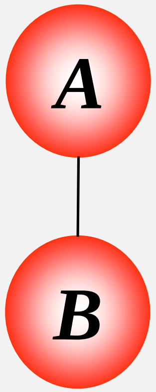
注 二叉树**不是**树的特殊情况，它们是两个概念。

	二叉树	树
左右子树	区分左子树和右子树	无须区分
位置	结点位置或者说次序都是固定，可以是空，但是不可以说它没有位置	结点位置是相对于别的结点，当没有别的结点时，它就无所谓左右

【例】二叉树和树的区别



具有两个结点的
二叉树有两种状态

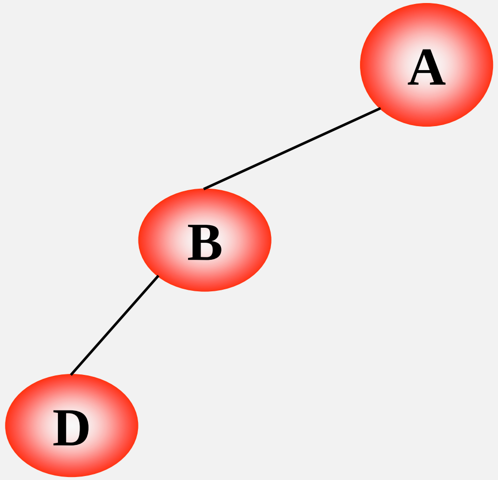


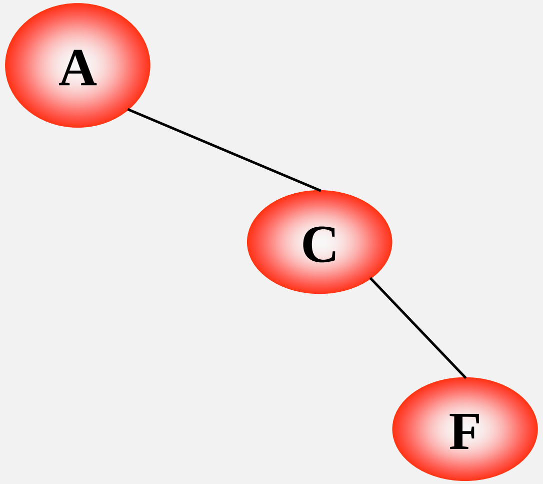
具有两个结点的
树只有一种状态

特例——斜二叉树

斜二叉树/退化二叉树是指二叉树仅存左分支/右分支

特点：结构最差，深度达到最大值N， 退化为线性表。



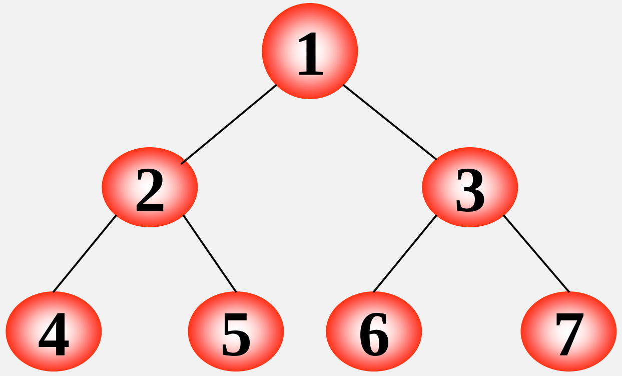


特例——满二叉树

一棵深度为 k 且有 2^k-1 个结点的二叉树称为**满二叉树**

特点：所有分支都存在左右子树,并且叶子全部在最底层。

编号规则：从根结点开始，自上而下，自左而右。



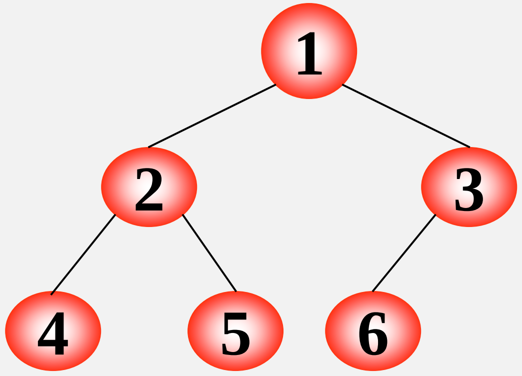
特例——完全二叉树

深度为 k 的具有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号为 1~ n 的结点一一对应时，称之为完全二叉树

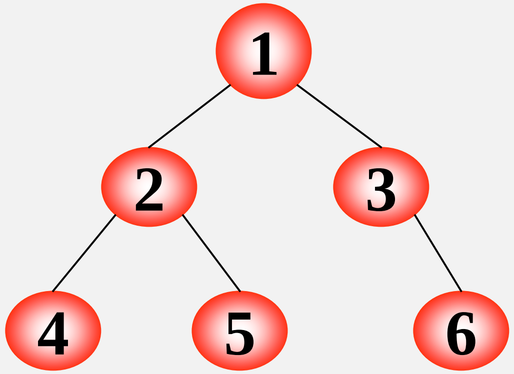
特点：

1）叶子只可能分布在最下层和次下层；

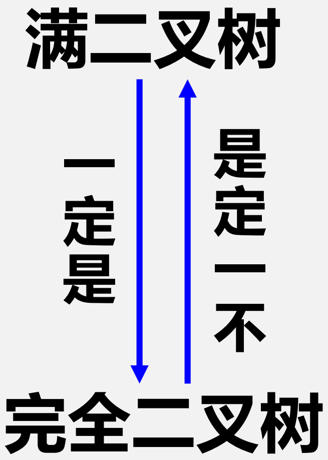
2）对任一结点，如果其右子树的最大层次为 l，则其左子树的最大层次必为 l 或 $l+1$。



完全二叉树



非完全二叉树



4.3.2 二叉树的性质

性质 1： 在二叉树的第 i 层上至多有 2^{i-1} 个结点 ($i \geq 1$)。

证： 采用数学归纳法证明

初始情况： 当 $i = 1$ 时，只有一个根结点，$2^{i-1} = 2^0 = 1$，命题成立。

归纳假设： 设任意 j $(1 \leq j < i)$，命题成立，即第 j 层上至多有 2^{j-1} 个结点。

归纳证明： 由归纳假设可知，第 $i-1$ 层上至多有 2^{i-2} 个结点。由于二叉树每个结点的度最大为 2，因此，**在第 i 层上最大结点数为第 $i-1$ 层上最大结点数的 2 倍**，即：$2 \times 2^{i-2} = 2^{i-1}$。

证毕。

性质 2：深度为 k 的二叉树至多有 2^k-1 个结点（$k\geq 1$）

证：由性质 1 可知，深度为 k 的二叉树的最大结点数为：

$$\sum_{i=1}^{k}(\text{第}\,i\,\text{层上的最大结点数})=\sum_{i=1}^{k}2^{i-1}=2^k-1$$

证毕。

性质 3： 对任何一棵二叉树 T，如果其叶子数为 n_0，度为 2 的结点数为 n_2，则 $n_0 = n_2 + 1$。

证：设 n_1 为二叉树 T 中度为 1 的结点数，其结点总数n为：

$$n = n_0 + n_1 + n_2$$

设 B 为分支总数，由于除根结点外，其余结点都有一个分支进入：

$$n = B + 1$$

由于这些分支都是由度为 1 和 2 的结点射出的，所以：

$$B = n_1 + 2n_2$$

计算得到：

$$n = B + 1 = n_1 + 2n_2 + 1$$

$$n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$$

$$n_0 = n_2 + 1$$

完全二叉树性质

性质 4：具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$。

证：假设此二叉树的深度为 k，则根据性质 2 及完全二叉树的定义得到：

$2^{k-1} - 1 < n \leq 2^k - 1$ 或 $2^{k-1} \leq n < 2^k$

取对数得： $k - 1 \leq \log_2 n < k$

因为 k 是整数，所以有：

$$k = \lfloor \log_2 n \rfloor + 1$$

性质 5： 如果对一棵有 n 个结点的完全二叉树 (深度为 $\lfloor \log_2 n \rfloor+1$) 的结点按层序编号 (从第 1 层到第 $\lfloor \log_2 n \rfloor+1$ 层，每层从 左到右)，则对任一结点 i $(1 \le i \le n)$，有：

(1) 如果 $i>1$，其父结点是结点 $\lfloor i/2 \rfloor$；否则 $i=1$，为二叉树的根结点；

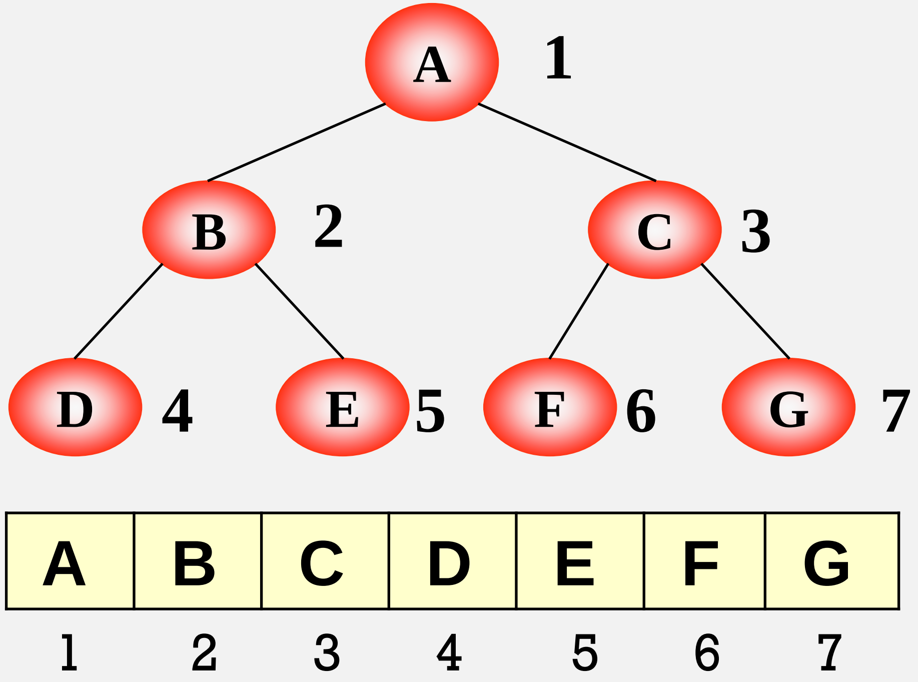
(2) 如果 $2i \le n$，其左孩子是结点 $2i$；否则，结点 i 无左孩子；

(3) 如果 $2i+1 \le n$，其右孩子是结点 $2i+1$；否则，则结点 i 无右孩子。

4.3.3 二叉树的存储结构

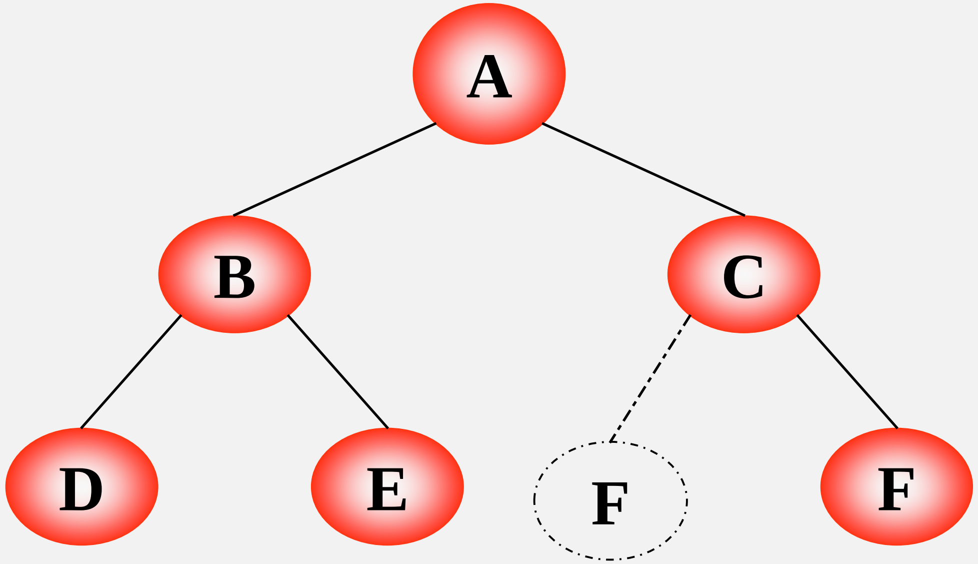
1 顺序存储结构

- 用一组连续的存储单元（例如数组）存储二叉树结点的数据，通过相对位置反映结点的父子关系，不需附加存储单元存放指针。
- 从树的根结点开始，从上层至下层，每层从左到右，依次给每个结点编号并将数据存放到一个数组的对应单元中。



一般二叉树的顺序存储

- 为了满足顺序存储要求而增加“虚”结点，在相应的存储单元中存放特殊的数值，区别其他的“实结点”。

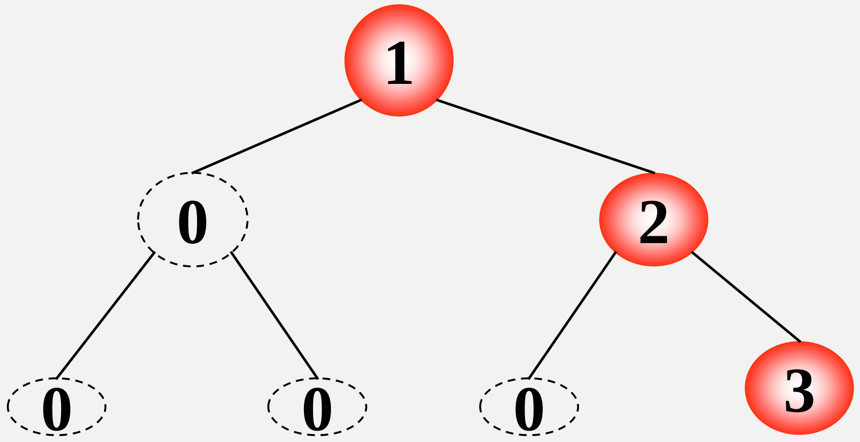


A	B	C	D	E	0	F
1	2	3	4	5	6	7

结点之间的关系：

- 任任意非根结点父结点为：$\lfloor i / 2 \rfloor$
- 左孩子结点为：$2i$
- 右孩子结点为：$2i+1$

最坏情况： 深度为 k 的且只有 k 个结点的右单支树需要长度为 2^k-1 的一维数组。



右单支树

1	0	2	0	0	0	3

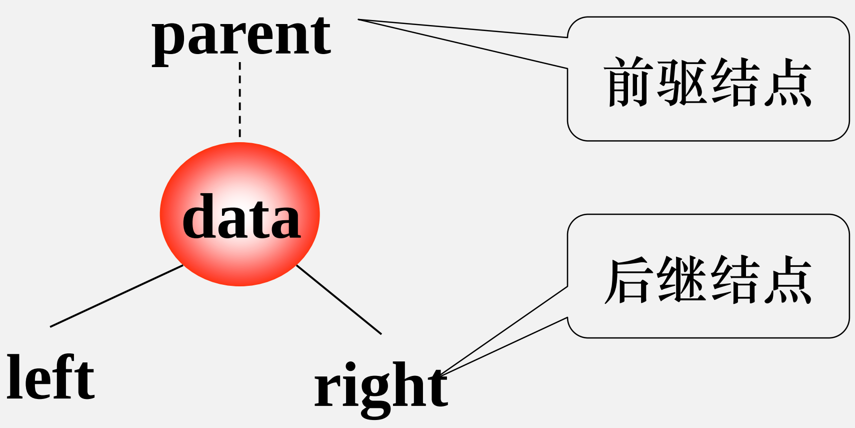
缺点：

（1）一般二叉树空间浪费；

（2）不易实现增加、删除操作；

2 链式存储结构

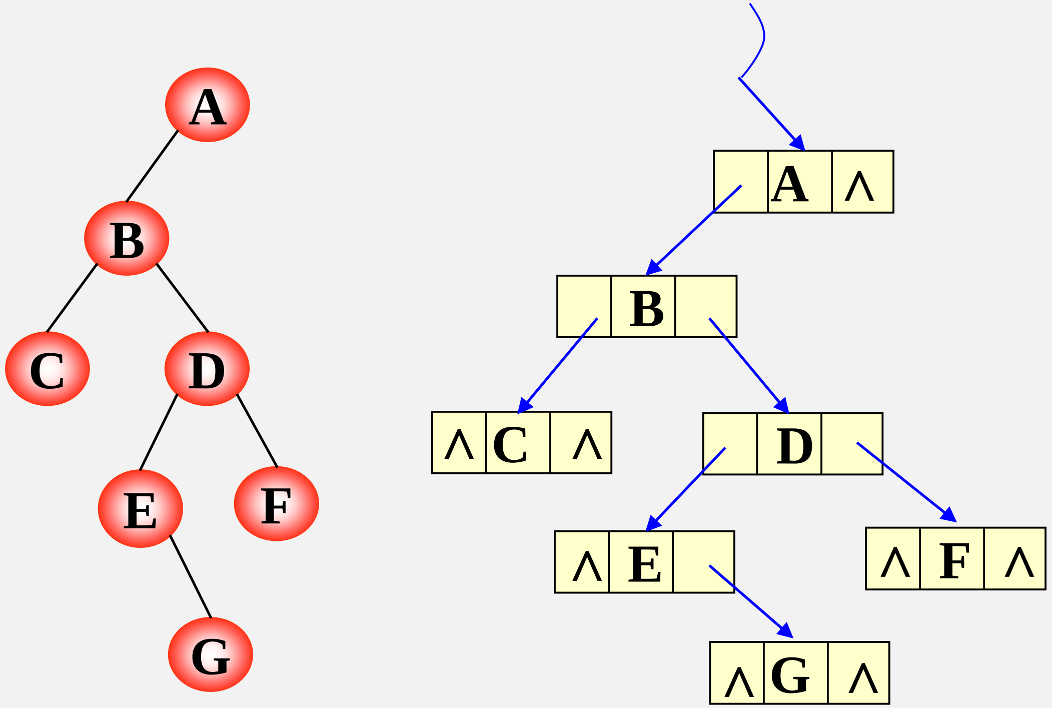
- 存储方式：每个结点三个数据成员组成：由数据域、左右指针域

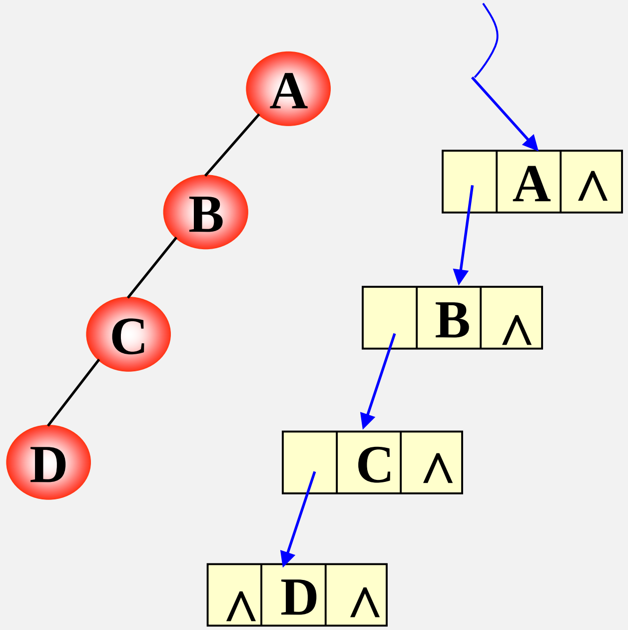


结点结构

```
typedef  struct  TreeNode *BinTree;   //二叉树类型
typedef  BinTree Position;
struct  TreeNode{     //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
};
```

【例】树的链表表示





在 n 个结点的二叉链表中，有 $n+1$ 个空指针域。

4.3.4 二叉树的主要操作

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, Item∊ElementType，操作：

1、Boolean 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。

4.3.4 二叉树的主要操作

常用的遍历方法有：

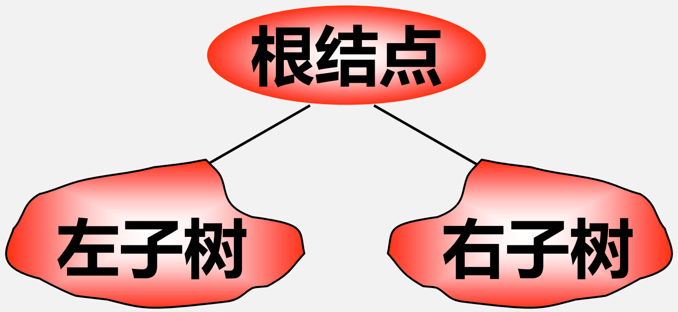
- void InOrder(BinTree BT)：根结点的访问次序在左、右子树之间（中序遍历）；
- void PreOrder(BinTree BT)：根结点的访问次序在左、右子树之前（先序遍历）；
- void PostOrder(BinTree BT)：根结点的访问次序在左、右子树之后 （后序遍历）；
- void LevelOrder(BinTree BT)：按层从小到大、从左到右的次序遍历（层序遍历）。

1、二叉树的遍历

遍历：顺着某一条搜索路径**巡访**二叉树中的结点，使得每个结点**均被访问一次**，而且**仅被访问一次**。

“**访问**”的含义很广，可以是对结点作各种处理，如：输出结点的信息、修改结点的数据值等，但要求这种访问**不破坏原来的数据结构**。

遍历方法



假设：L：遍历左子树　　V：访问根结点　　R：遍历右子树，
则遍历整个二叉树方案共有：

VLR、LVR、LRV、VRL、RVL、RLV 六种。

规定左子树的遍历总在右子树之前，考虑三种遍历：LVR、VLR、LRV

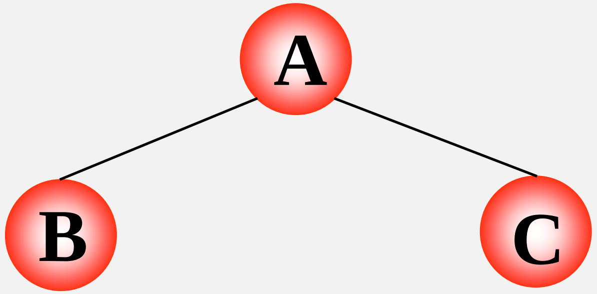
(1) 先序遍历二叉树的操作定义：

若二叉树为空，则空操作；否则

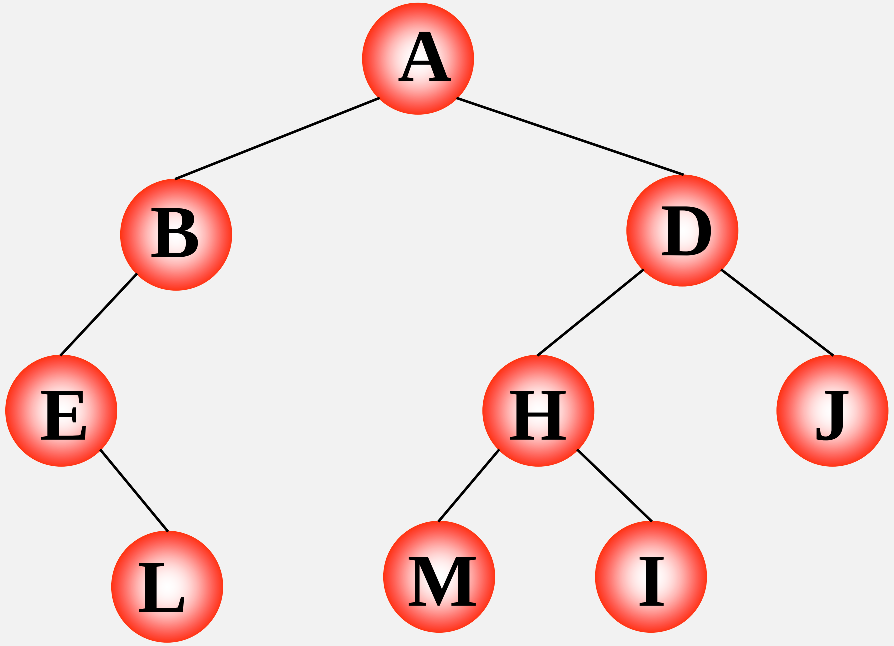
(1) 访问根结点；

(2) 先序遍历左子树；

(3) 先序遍历右子树。



先序遍历的顺序为：ABC

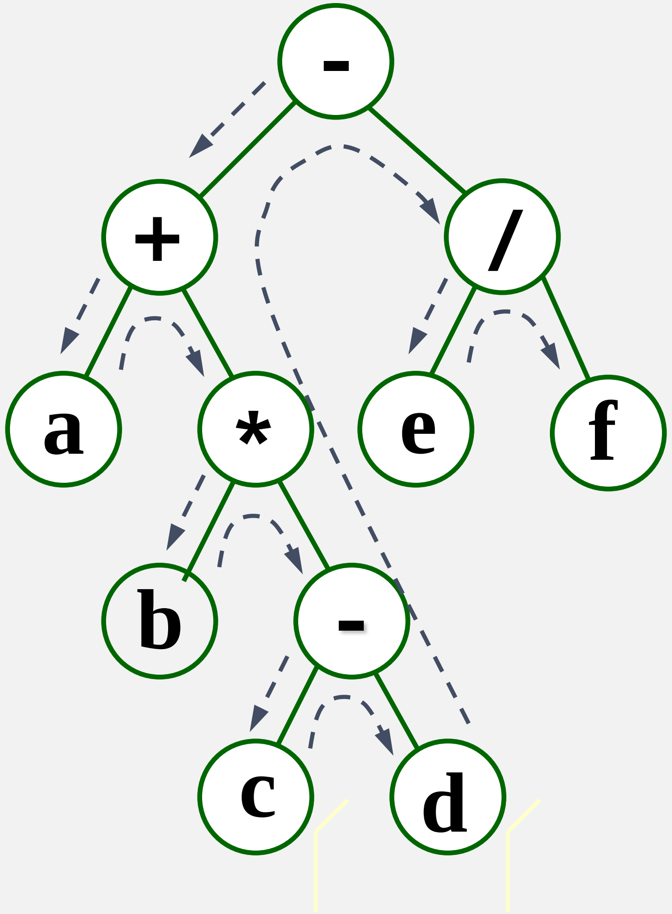


先序遍历的顺序为：

ABELDHMIJ

先序遍历二叉树基本操作的递归算法在二叉链表上的实现：

```
void PreOrder (BinTree BT)
{
  if (T!=NULL)
    {
      cout<<BT->data<<endl;
      PreOrder(BT->left);
      PreOrder(BT->right);
    }
}
```



遍历结果

- + a * b - c d / e f

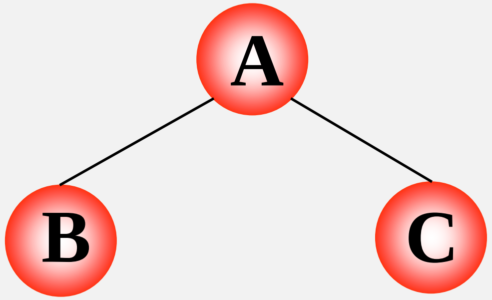
(2)中序遍历二叉树的操作定义：

若二叉树为空，则空操作；否则

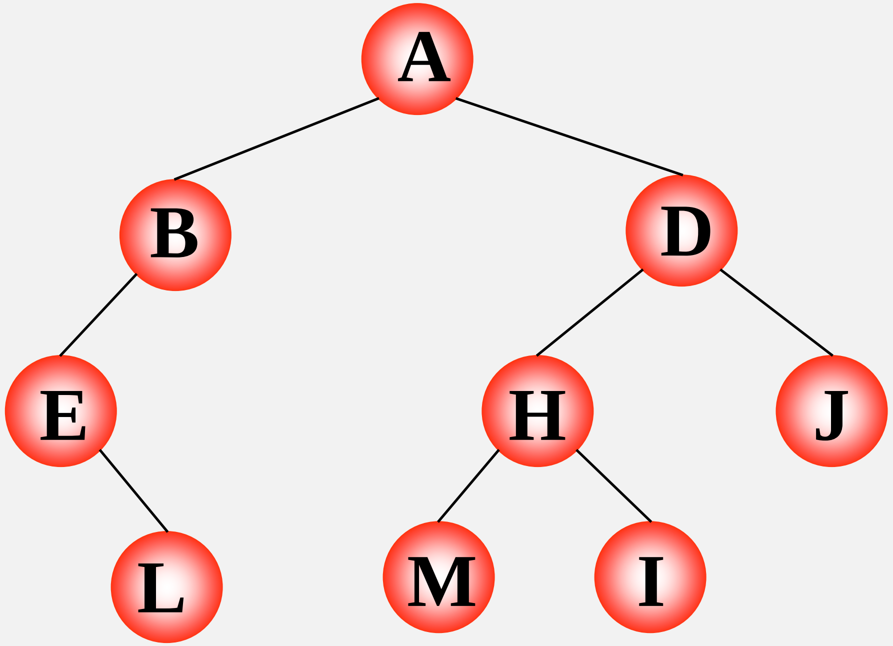
(1) 中序遍历左子树；

(2) 访问根结点；

(3) 中序遍历右子树。



中序遍历的顺序为：BAC

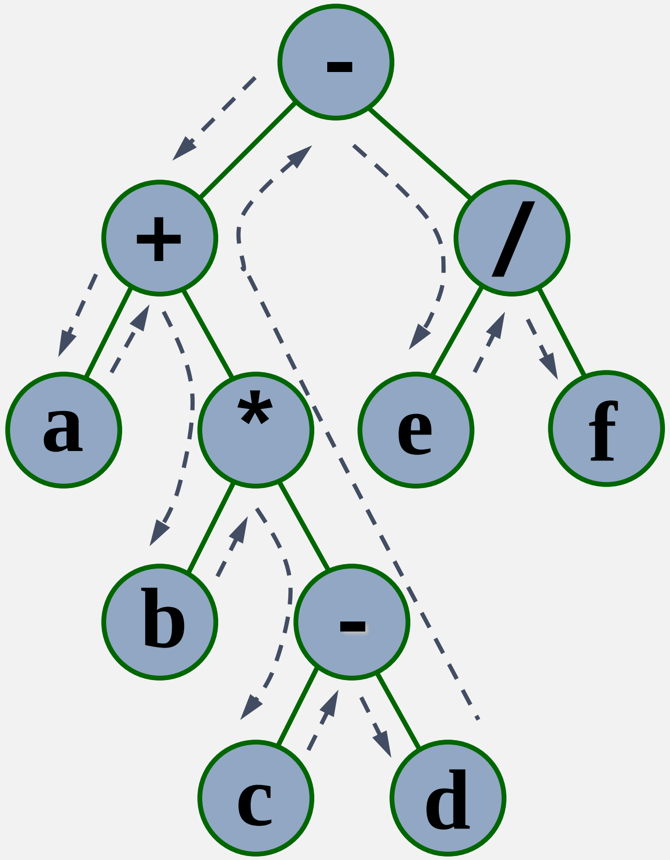


中序遍历的顺序为：

ELBAMHIDJ

中序遍历二叉树基本操作的递归算法在二叉链表上的实现：

```
void InOrder (BinTree BT)
{
  if (BT!=NULL)
    {
        InOrder(BT->left);
        cout<<BT->data<<endl;
        InOrder(BT->right);
     }
}
```



遍历结果

a + b * c - d - e / f

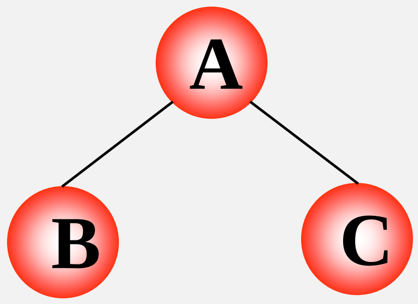
(3)后序遍历二叉树的操作定义：

若二叉树为空，则空操作；否则

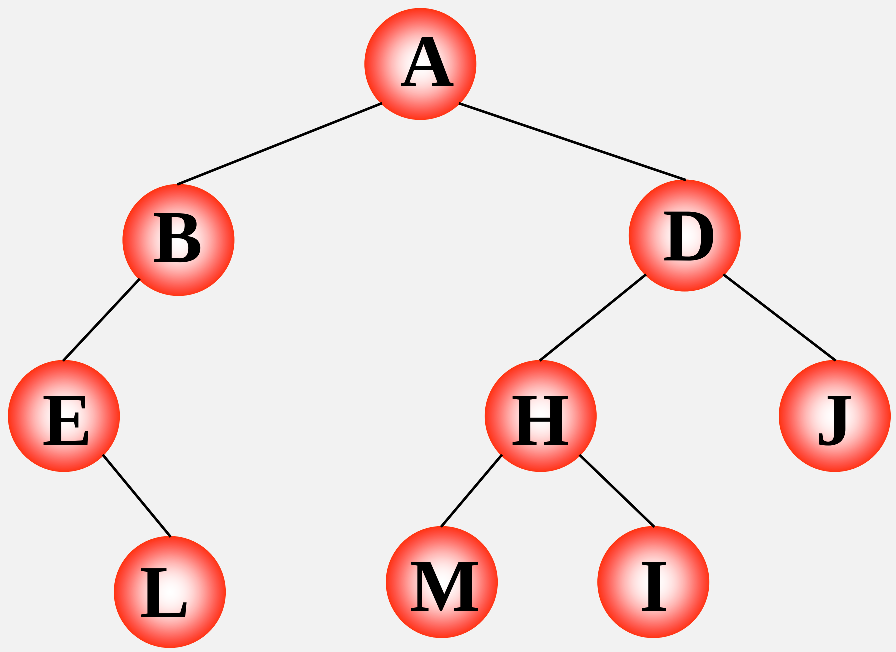
(1) 后序遍历左子树；

(2) 后序遍历右子树；

(3) 访问根结点。



后序遍历的顺序为：BCA

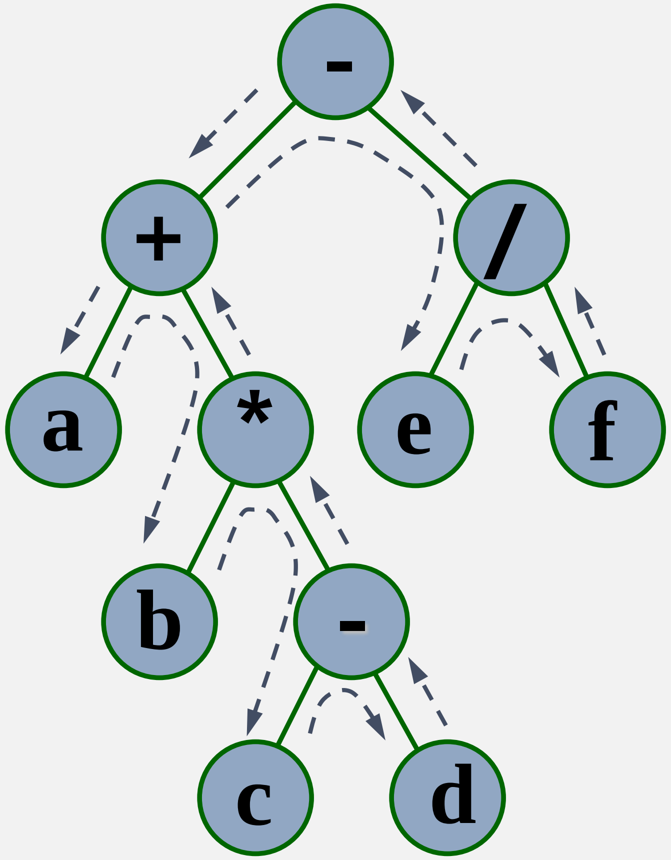


后序遍历的顺序为：

LEBMIHJDA

- **后序遍历**二叉树基本操作的**递归**算法在二叉链表上的实现：

```
void PostOrder (BinTree BT)
{
   if (BT!=NULL)
      {
          PostOrder(BT->left);
          PostOrder(BT->righ);
          cout<<BT->data<<endl;
       }
}
```



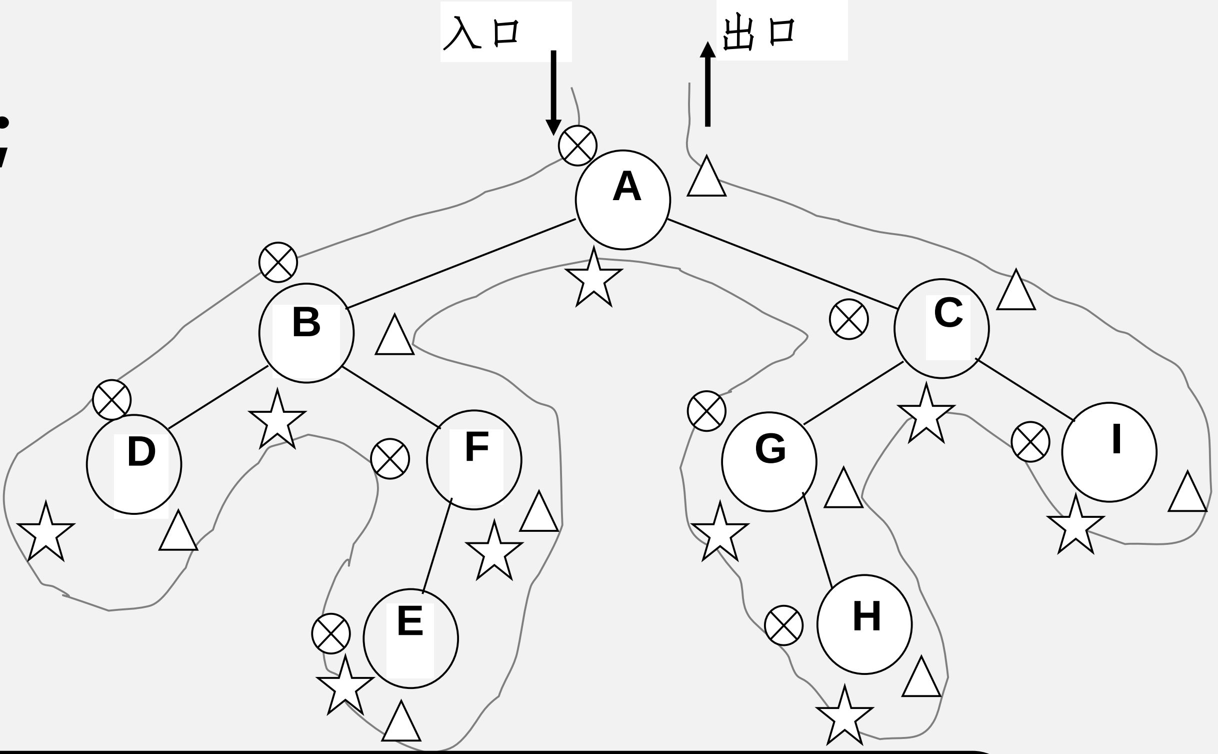
遍历结果

a b c d - * + e f / -

对遍历的分析

三种遍历算法的访问路径是相同的，只是访问结点的时机不同。

- 先序遍历是**深入时**遇到结点访问；
- 中序遍历是从**左子树返回**时遇到结点访问；
- 后序遍历是从**右子树返回**时遇到访问结点；



返回结点的顺序与进入结点顺序相反（后进先出），符合栈结构的特点

（4） 中序遍历的非递归算法

- 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 InOrder( BinTree BT )
{  BinTree T;
   Stack S = CreatStack( MaxSize ); /*创建并初始化堆栈S*/
   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;   }
       if(!IsEmpty(S)){
           T = Pop(S); /*结点弹出堆栈*/
           printf(“%5d”, T->Data); /*（访问）打印结点*/
           T = T->Right; /*转向右子树*/
       }
   }}
```

(5) 层序遍历

❖ 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）：

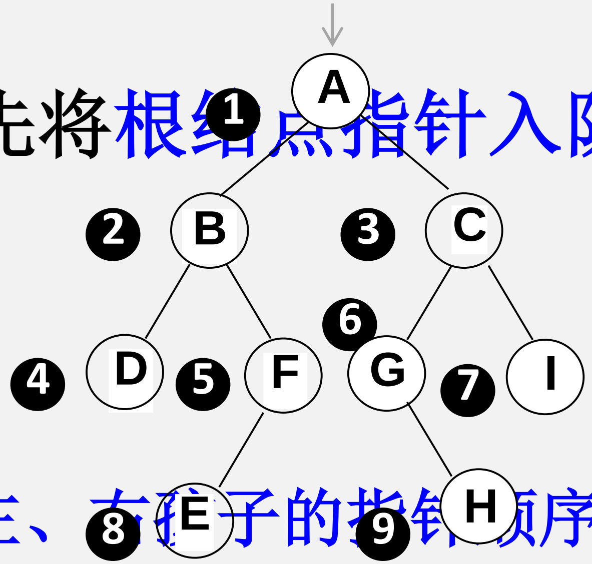
工作队列：A B C D F G I E H

① 从队列中取出一个元素；

② 访问该元

层序遍历 => A B C D F G I E H

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 LevelOrder( BinTree BT )
{    Queue Q;   BinTree T;
     if ( !BT ) return; /* 若是空树则直接返回 */
     Q = CreatQueue( MaxSize ); /*创建并初始化队列Q*/
     AddQ( Q, BT );
     while ( !IsEmptyQ( Q ) ) {
         T = DeleteQ( Q );
         printf("%d\n", 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 );
     }}
```

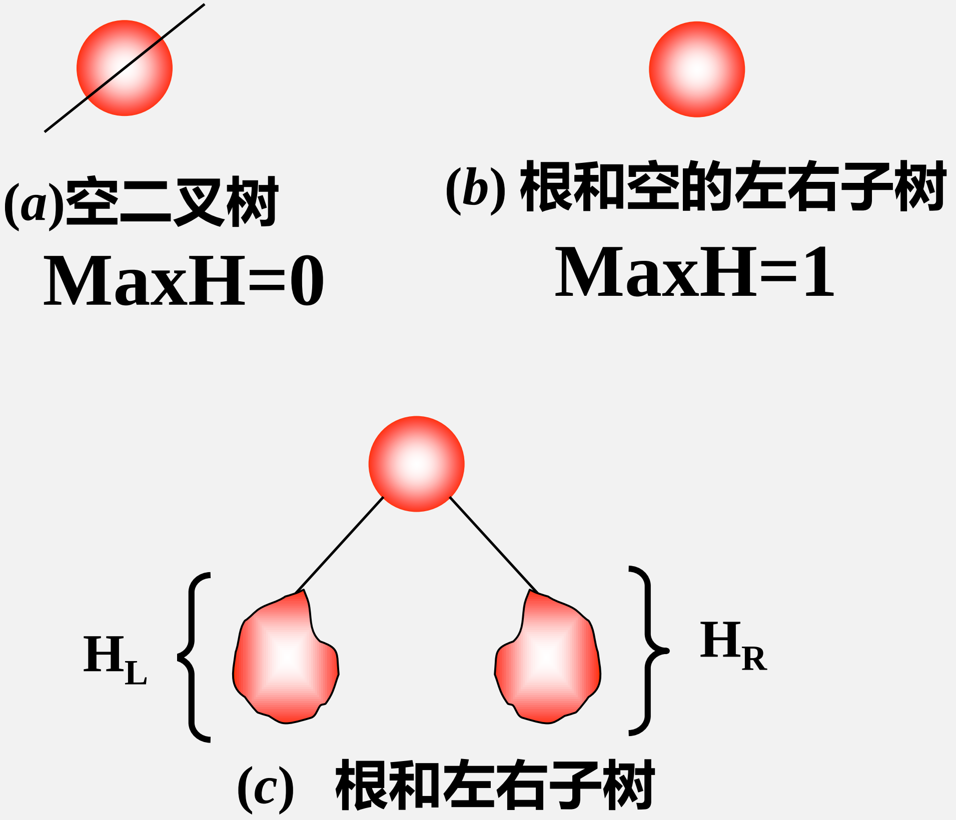
【例】输出二叉树中的叶子结点。

➢有条件输出问题。可以在二叉树的任意一个遍历算法中增加检测结点的“左右子树是否都为空”条件判断语句。

➢在先序遍历算法的基础上修改：

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

【例】求二叉树的高度。



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

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

【例】 由两种遍历序列确定二叉树

…历中的任意两种遍历序列，
…确定一棵二叉树呢？

必须要有中序遍历才行！

❖ 没有中序的困扰：

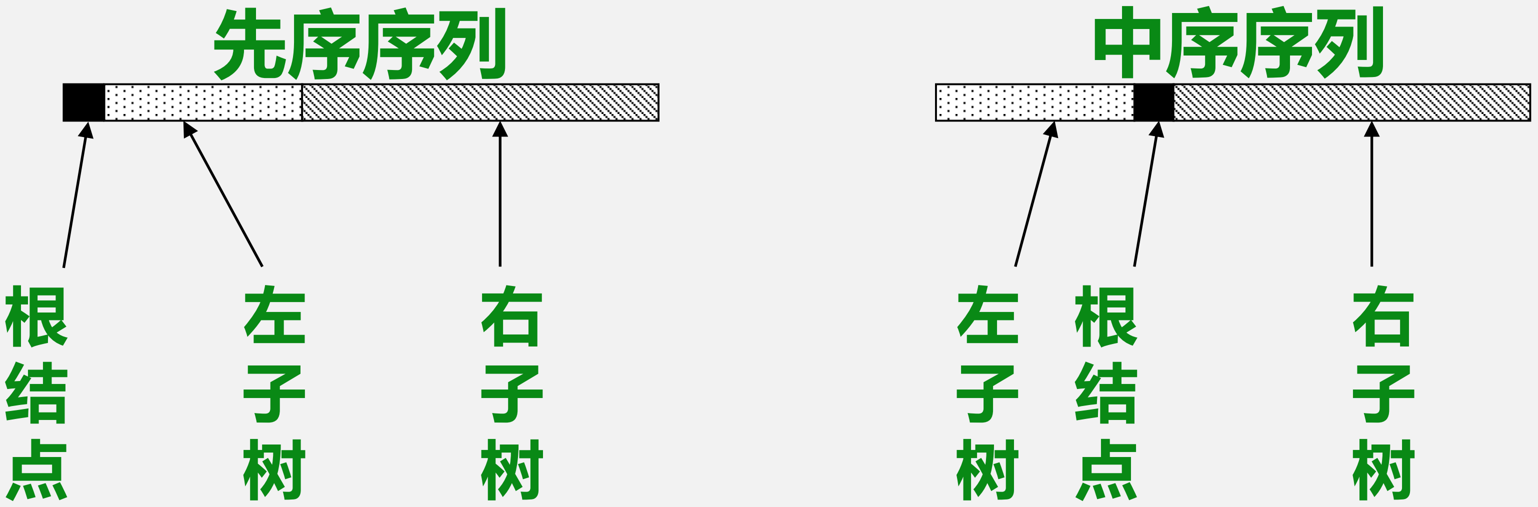
➢ 先序遍历序列：A B

➢ 后序遍历序列：B A

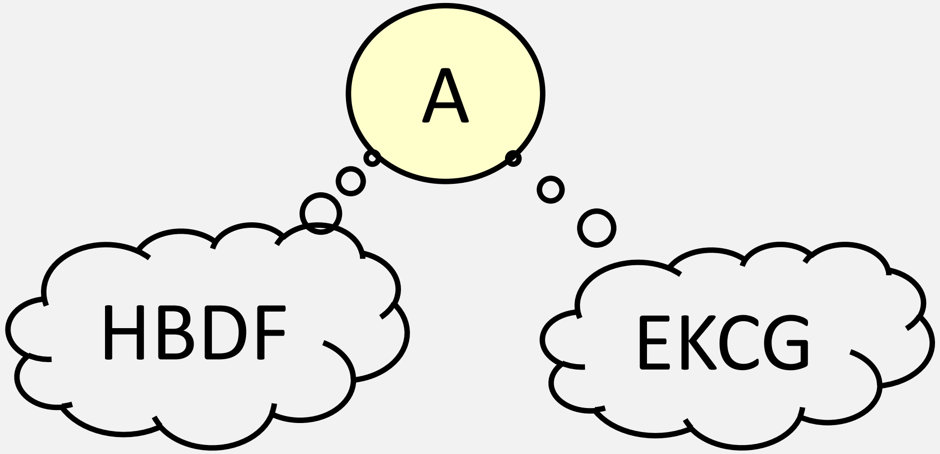
❖ 先序和中序遍历序列来确定一棵二叉树

〖分析〗 先序遍历序列的第一个结点就是根结点；

- 利用根结点将中序遍历序列中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。

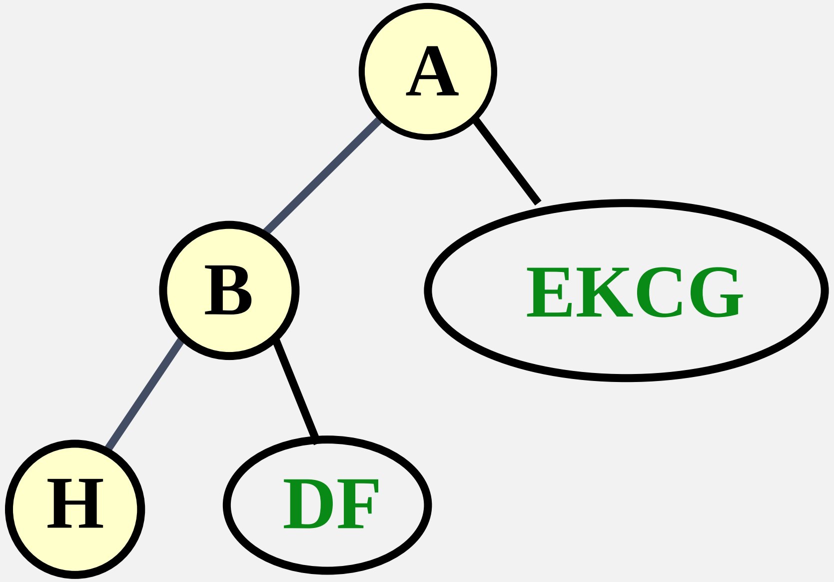


【例】给定先序遍历序列 { ABHFDECKG } 和中序遍历序列 { HBDFAEKCG }, 构造一棵二叉树。

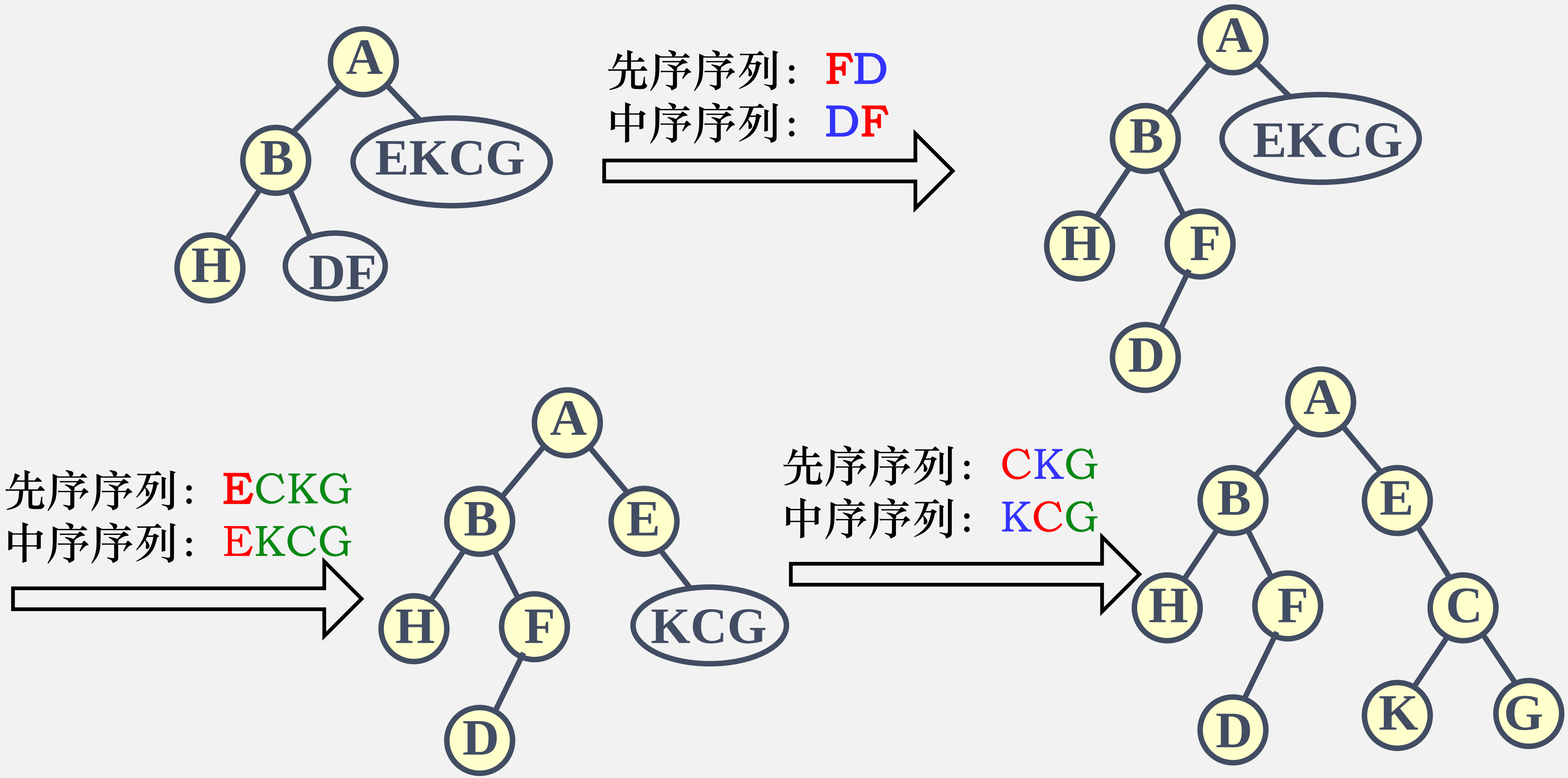


左子树　　　右子树

前序序列：ABHFDECKG
中序序列：HBDFAEKCG



先序序列：BHFD
中序序列：HBDF

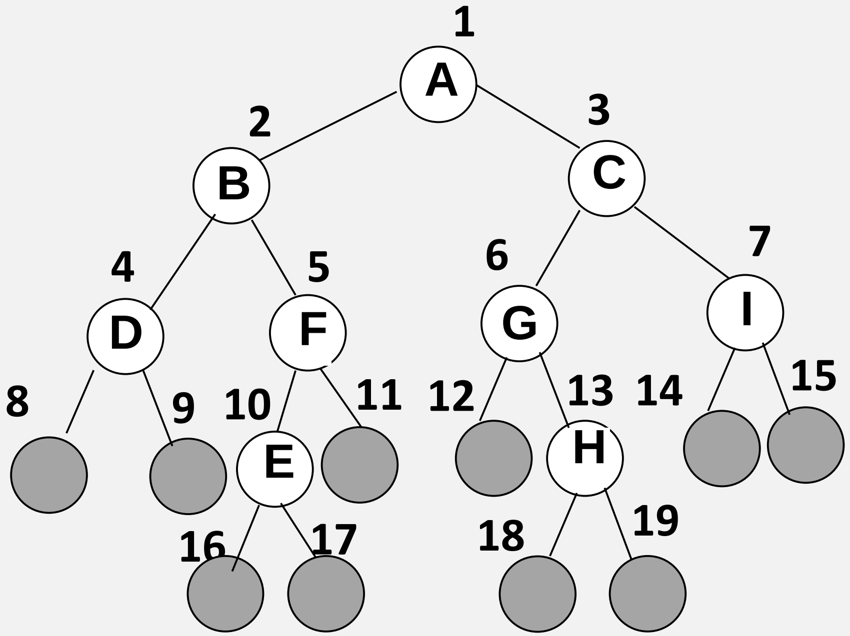


A
B
EKCG
H
DF
先序序列：FD
中序序列：DF
A
B
EKCG
H
F
D
先序序列：ECKG
中序序列：EKCG
A
B
E
H
F
KCG
D
先序序列：CKG
中序序列：KCG
A
B
E
H
F
C
D
K
G

2、二叉树的创建

- 创建一颗二叉树必须首先确定树中结点的输入顺序。常见的方法：先序创建和层次创建

  - 层序创建方法：输入序列按从上至下、从左到右的顺序输入，各层空结点输入0值；
  - 需要定义一个队列暂时存储各类结点地址。

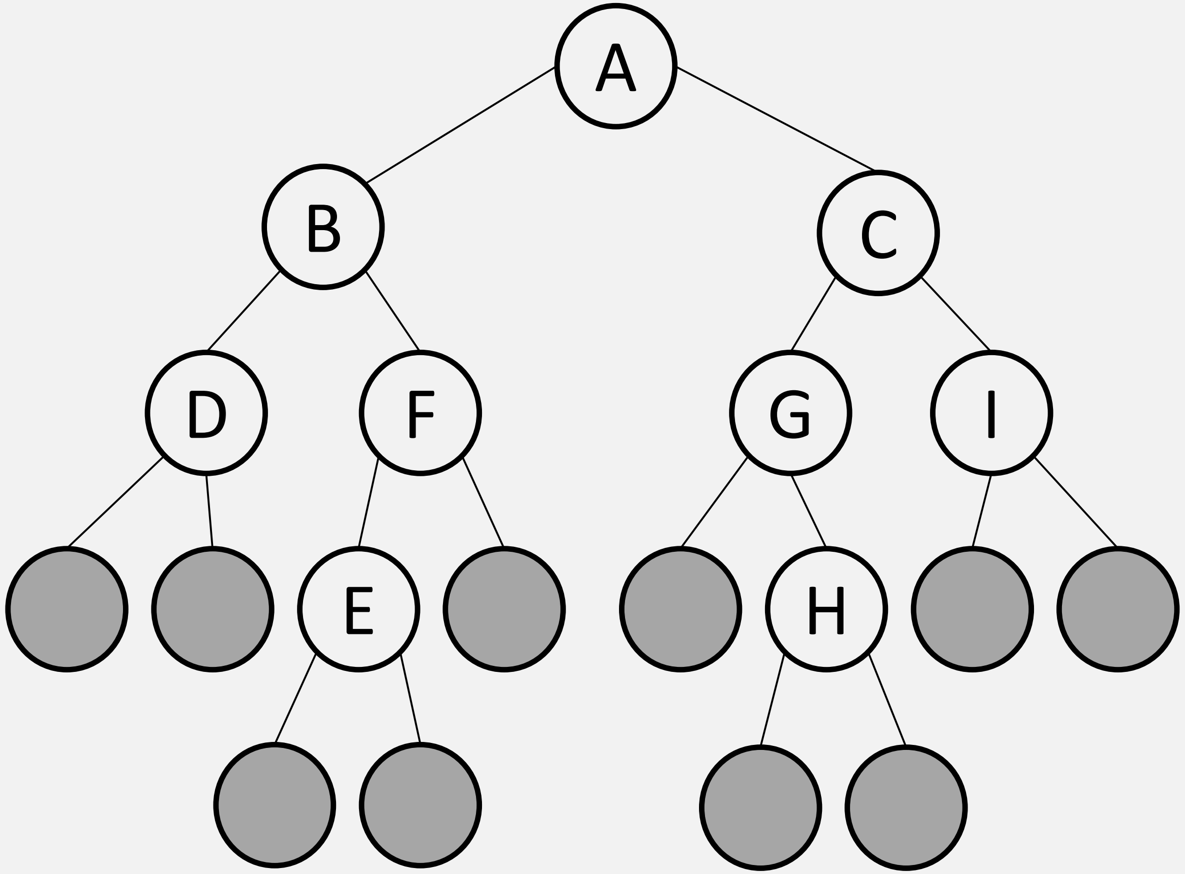


层序创建的输入序列：A, B, C, D, F, G, I, 0, 0, E, 0, 0, H, 0, 0, 0, 0, 0, 0

【例】输入层序序列：A, B, C, D, F, G, I, 0, 0, E, 0, 0, H, 0, 0, 0, 0, 0, 0

输入序列： **A B C D F G I 0 0 E 0 0 H ………**

工作队列： **A B C D F G I E H**



层序创建过程如下：

① 输入一个元素，若不为0，动态分配一个结点单元，存入数据，将该地址入队列；否则此树为空，空指针赋值给根结点，构造完毕；

```
#define NoInfo 0
BinTree CreateBinTree()
{
    int data;
    BinTree BT, T;
    Queue Q = CreatQueue();

    scanf("%d", &data);
    if(data != NoInfo){  //分配根结点单元，并将结点地址入队列
        BT = (BinTree)malloc(sizeof(struct TNode));
        BT->data = data;
        BT->left = BT->right = NULL;
        Add(Q, BT);
    }else  return NULL;
```

② 如果队列不空，从队列中取出一个结点地址，并建立该结点的左右孩子：

- 从输入序列中读入下一个数据：如果数据为0，将左孩子指针置为空；否则创建左孩子结点，同时将此孩子地址入队；
- 读入下一个数据：如果数据为0，将右孩子指针置为空；否则创建右孩子结点，同时将此孩子地址入队；

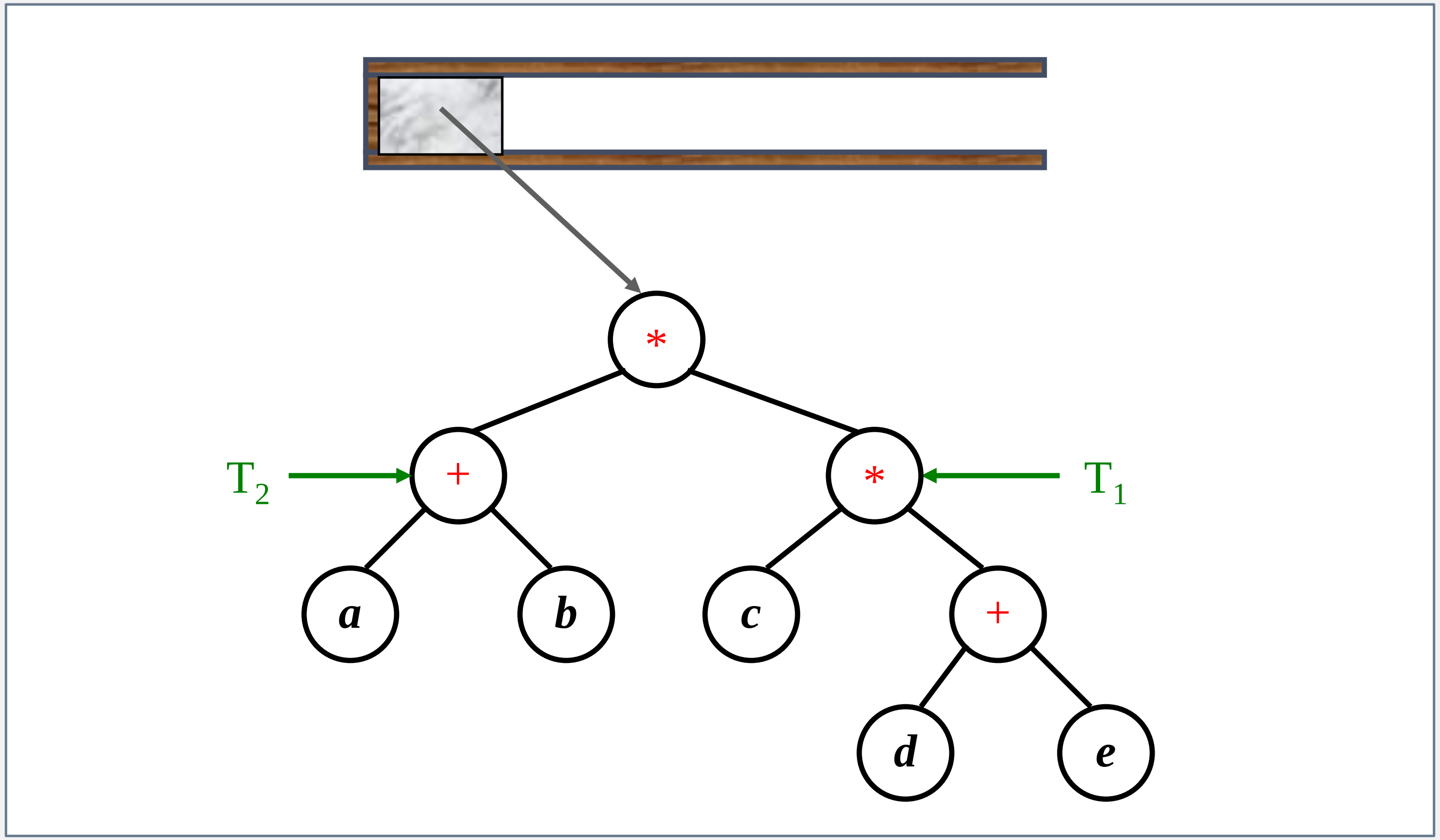
③ 重复步骤②，直到队列为空为止。

```
while(!IsEmpty(Q)){
    T = Delete(Q);//让队首结点出队，即读出队首结点的地址
    scanf("%d", &data);//读入T结点左儿子的信息
    if(data == NoInfo) T->left = NULL;  //无左孩子
    else{  //分配新结点作为出队结点左孩子；新结点入队
        T->left = (BinTree)malloc(sizeof(struct TNode));
        T->left->data = data;
        T->left->left = T->left->right = NULL;
        Add(Q, T->left);
    }
    scanf(“%d”, &data); //读入T结点右孩子
    if(data == NoInfo)  T->right = NULL;
    else{
        T->right = (BinTree)malloc(sizeof(struct TNode));
        T->right->data = data;
        T->right->left = T->right->right = NULL;
        Add(Q, T->right);}
}
return BT;
}
```

3、表达式树的构造

【例】构造(a + b) * (c * (d + e)) 表达式树

输入：后缀表达式a b + c d e + * *



表达式树的构造过程如下：

- 依次读入字符：
  - 如果读入的是运算数，将其作为单个结点构造一棵二叉树，并将指向这颗树的指针入栈；
  - 如果读入的是运算符，从栈中弹出两个元素连同读入的运算符构成一棵新的二叉树，树根为所读入的运算符，左子树是从栈中后弹出的元素，右子树是先弹出的元素，新二叉树指针入栈
- 重复上述过程