

提纲

CONTENTS

第6章 树和二叉树

6.1 树

6.2 二 叉 树

6.3 二叉树先序、中序和后序遍历

6.4 二叉树的层次遍历

6.5 二 叉 树的构造

6.6 线索二叉树

6.7 哈夫曼树

6.8 二叉树与树、森林之间的转换

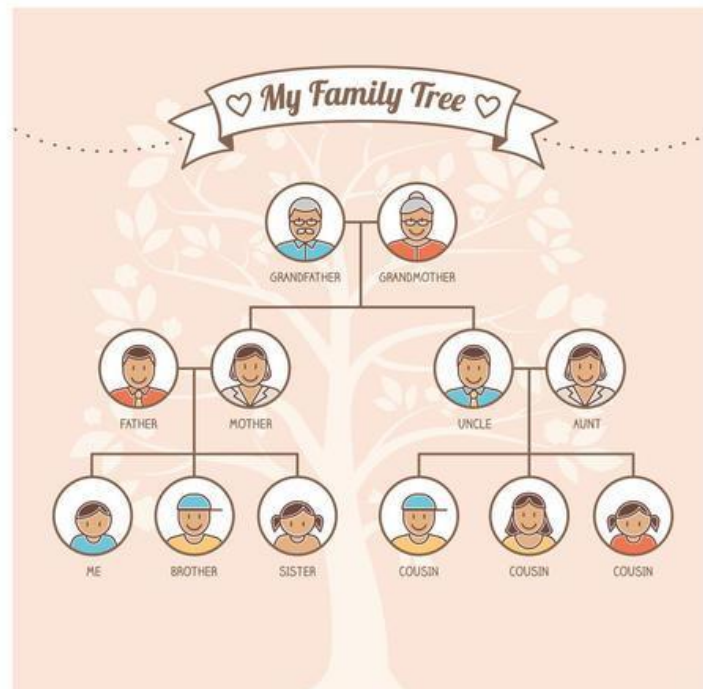
6.9 并查集

客观世界很多事物存在层次关系

- 人类社会家谱
- 社会组织结构
- 图书管理
- 信息文件管理

分层次组织在管理上具有更高的效率！

数据管理的基本操作之一：查找
以图书管理或文件管理为例说明！



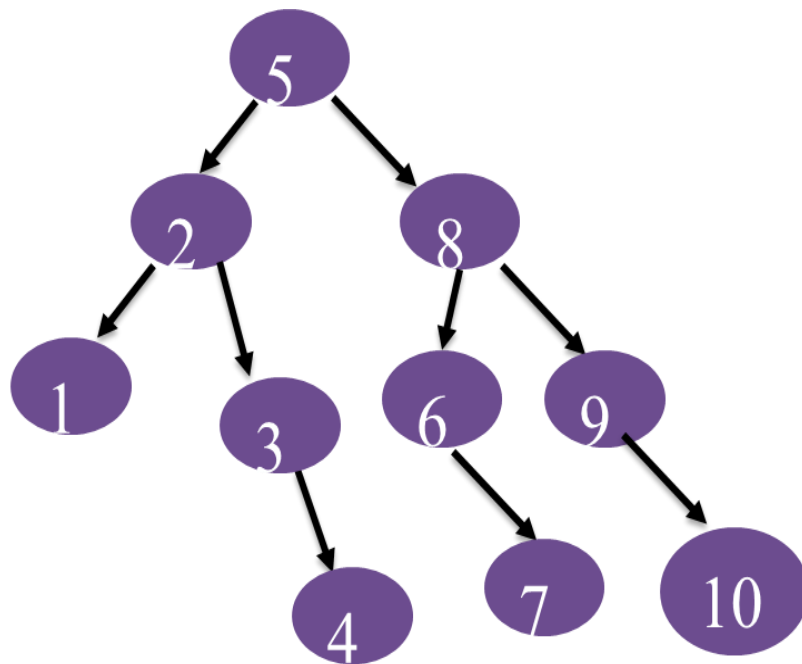
二分查找

这种查找方法我们经常在用！如 线路故障检测等！

让计算机进行二分查找，首先要把元素有序存放在数组中，默认升序存放，即 $x_1 < x_2 < \dots < x_n$ 。

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

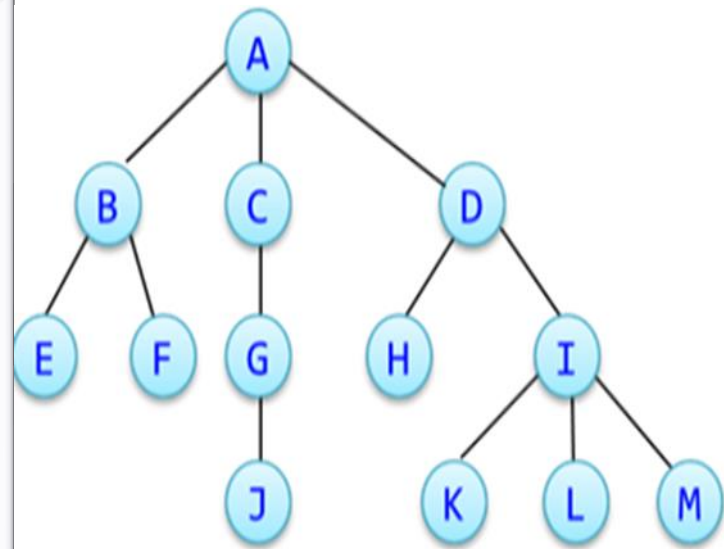
二分查找比较操作是分层次进行的！



6.1 树

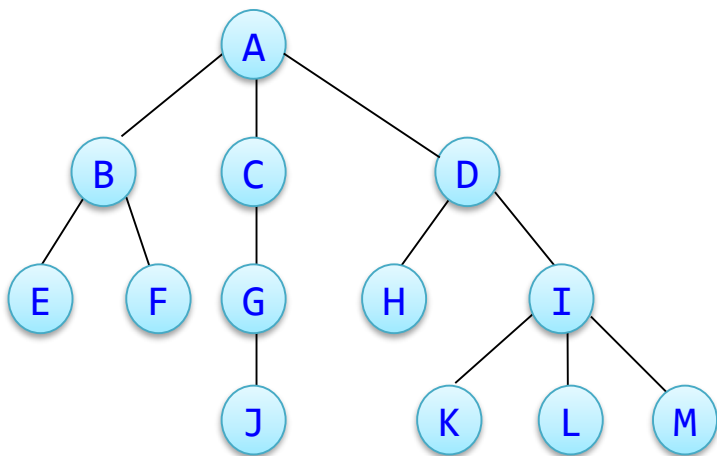
6.1.1 树的定义

- 树是由 n ($n \geq 0$) 个结点组成的有限集 T ,
- 当 $n=0$ 时, T 为空树;
- 当 $n>0$ 时,
 - (1) 有且仅有一个称为 T 的根的结点,
 - (2) 当 $n>1$ 时, 其余结点可分为 m ($m \geq 0$) 个互不相交的有限集 T_1 、 T_2 、 \dots 、 T_m , 每个 T_i 也是一棵树, 且称为根的子树。

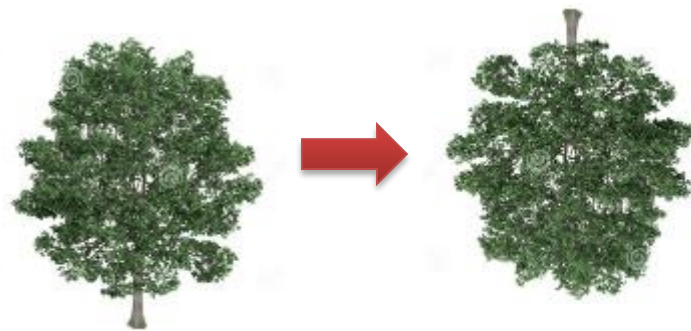


6.1.2 树的逻辑结构表示方法

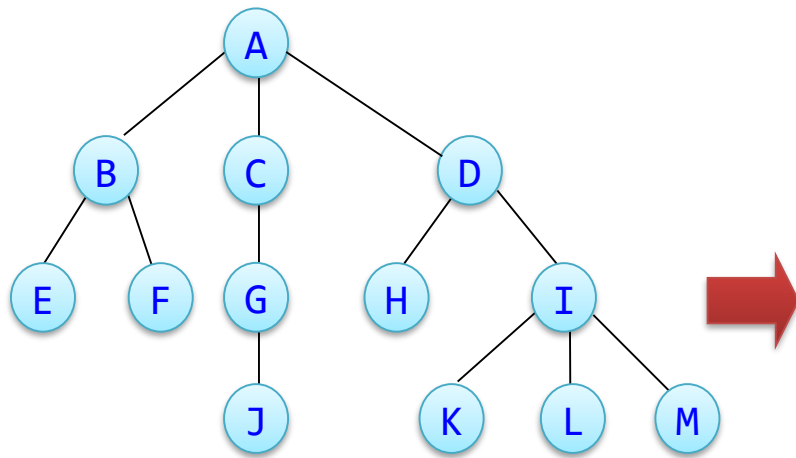
● **树形表示法** 这是树的最基本的表示，使用一棵倒置的树表示树结构，非常直观和形象。



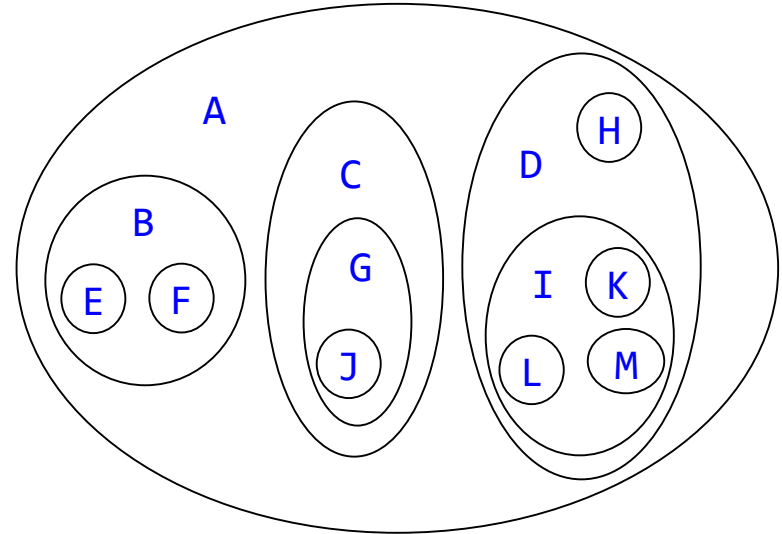
(a) 树形表示法



● **文氏图表示法** 用集合以及集合的包含关系描述树结构。

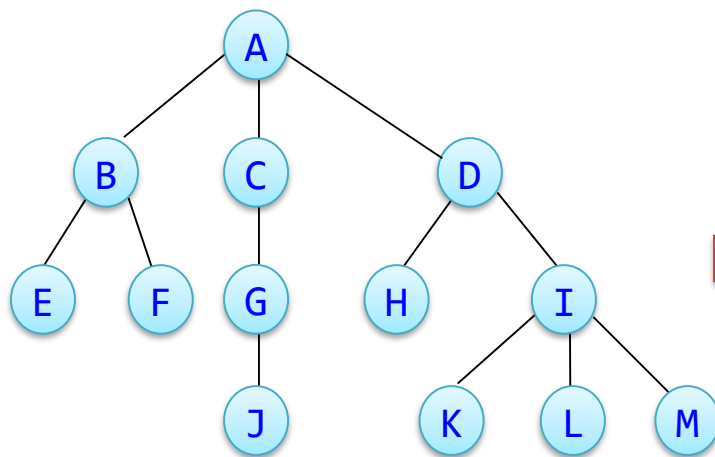


(a) 树形表示法

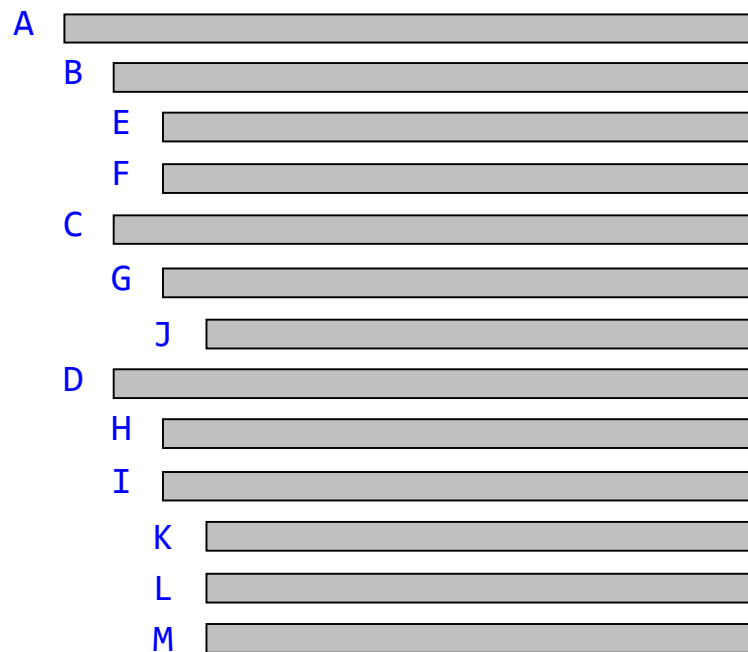


(b) 文氏图表示法


 **凹入表示法** 使用线段的伸缩关系描述树结构。

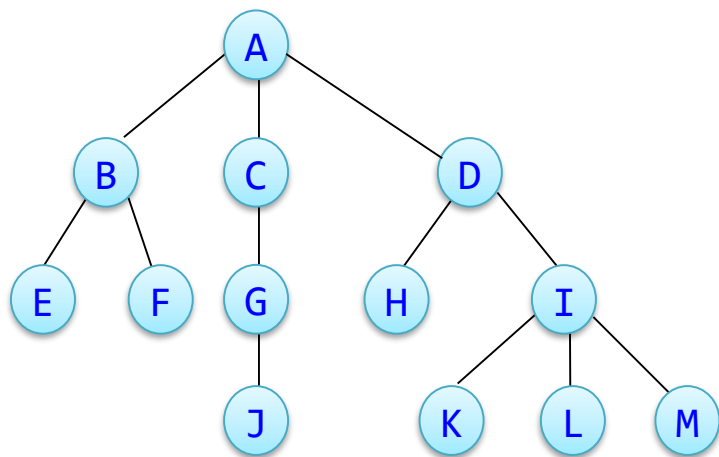


(a) 树形表示法



(c) 凹入表示法

 **括号表示法** 将树的根结点写在括号的左边，除根结点之外的其余结点写在括号中并用逗号分隔。



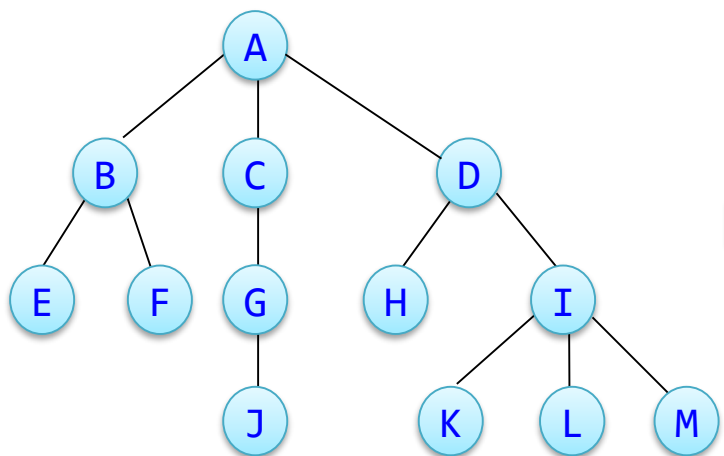
(a) 树形表示法



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

根(子树1, 子树2, ..., 子树 m)

● **列表表示法** “[根结点，子树₁，子树₂，…，子树_m]”。

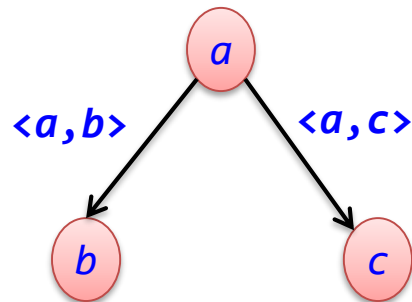


```
['A',  
  ['B',['E'],['F']],    #第1棵子树  
  ['C',['G'],['J']],    #第2棵子树  
  ['D',['H'],['I'],['K'],['L'],['M']]  
  ]
```

(a) 树形表示法

树是一种非线性数据结构，具有以下特点：

- 每一结点可以有零个或多个后继结点，但有且只有一个前驱结点（根结点除外）。
- 数据结点按分支关系组织起来，清晰地反映了数据元素之间的层次关系。



抽象数据类型树的描述

ADT Tree

{

数据对象:

$D = \{a_i \mid 0 \leq i \leq n-1, n \geq 0, a_i \text{ 为 } E \text{ 类型}\}$

数据关系:

$R = \{r\}$

$r = \{ \langle a_i, a_j \rangle \mid a_i, a_j \in D, 0 \leq i, j \leq n-1, \text{ 其中每个结点最多只有一个前驱结点、可以有零个或多个后继结点, 有且仅有一个结点即根结点没有前驱结点} \}$

基本运算:

CreateTree(): 由树的逻辑结构表示建立其存储结构。

DispTree(): 输出树的括号表示串。

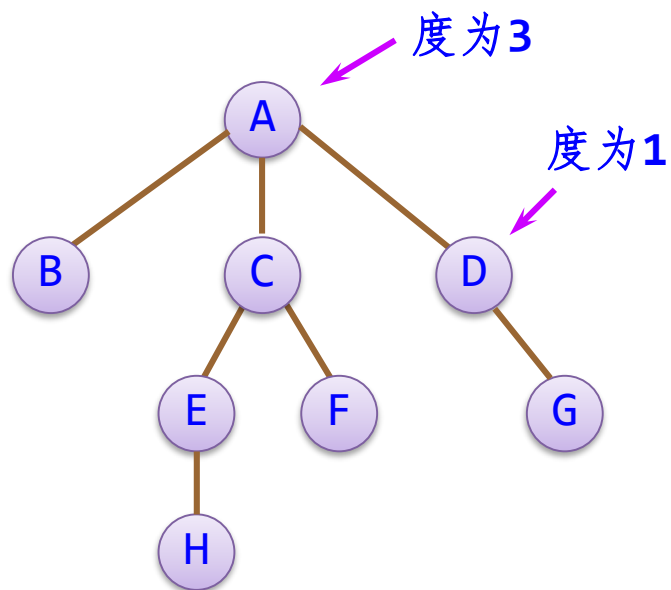
E GetParent(int i): 求编号为*i*的结点的双亲结点值。

...

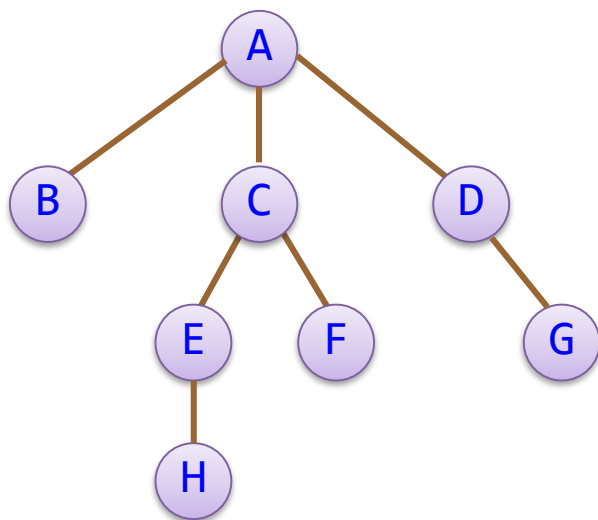
}

6.1.3 树的基本术语

- **结点的度** 树中每个结点具有的子树数（后继结点数）称为该结点的度。

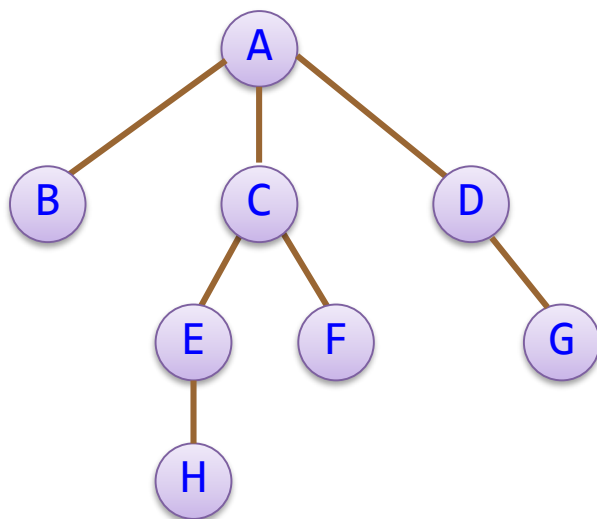


- **树的度** 树中所有结点的度的最大值称之为**树的度或树的次数**。



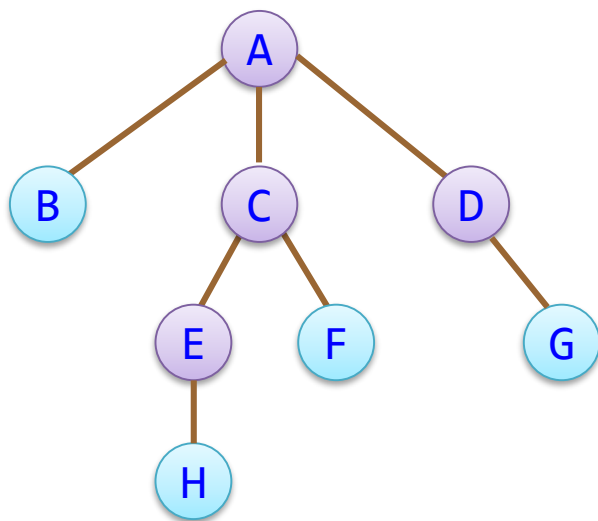
树的度为3

- **分支结点** 度大于0的结点称为分支结点或非终端结点。度为1的结点称为单分支结点，度为2的结点称为双分支结点，依次类推。



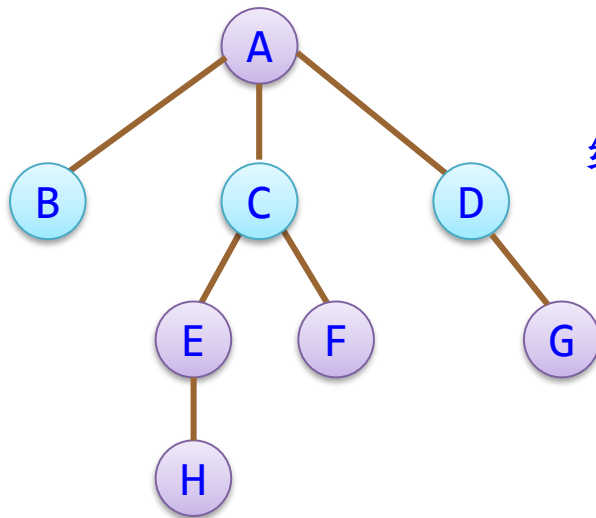
A、C、D、E为分支结点

- **叶子结点 (或叶结点)** 度为零的结点称为叶子结点或终端结点。



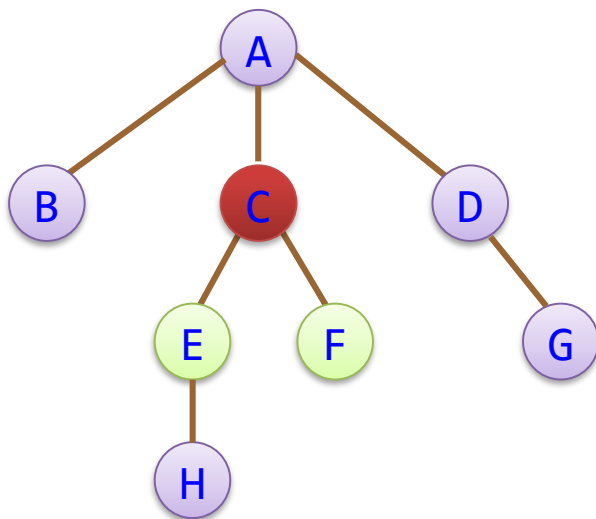
*B、H、F、G*为叶子结点

- **孩子结点** 一个结点的后继称之为该结点的孩子结点。



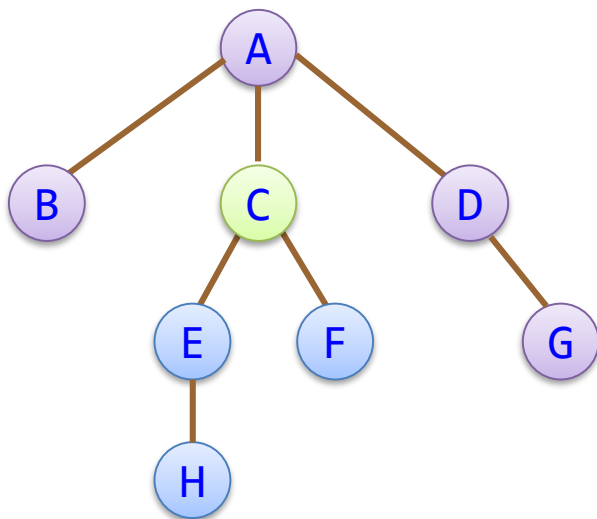
结点A的孩子结点为B、C和D

- **双亲结点（或父亲结点）** 一个结点称为其后继结点的双亲结点。



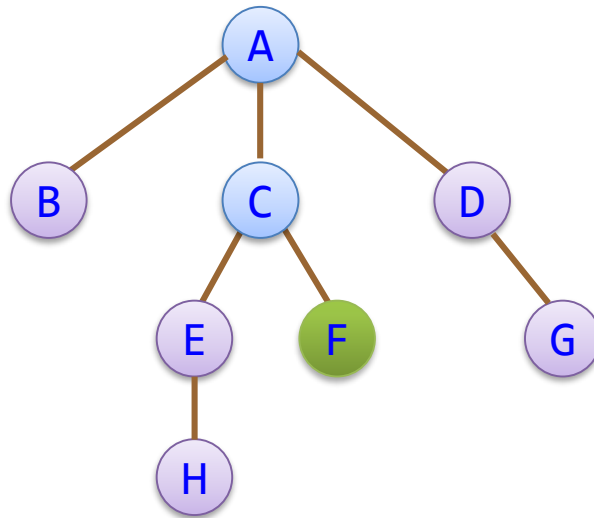
结点*E*和*F*的双亲结点均为*C*

- **子孙结点** 一个结点的子树中除该结点外的所有结点称之为该结点的子孙结点。



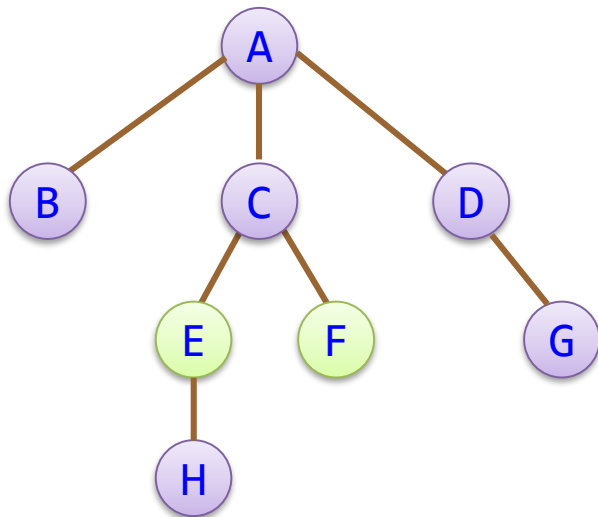
*C*结点的子孙结点为*E*、*F*和*H*

- **祖先结点** 从树根结点到达某个结点的路径上通过的所有结点称为该结点的祖先结点（不含该结点自身）。



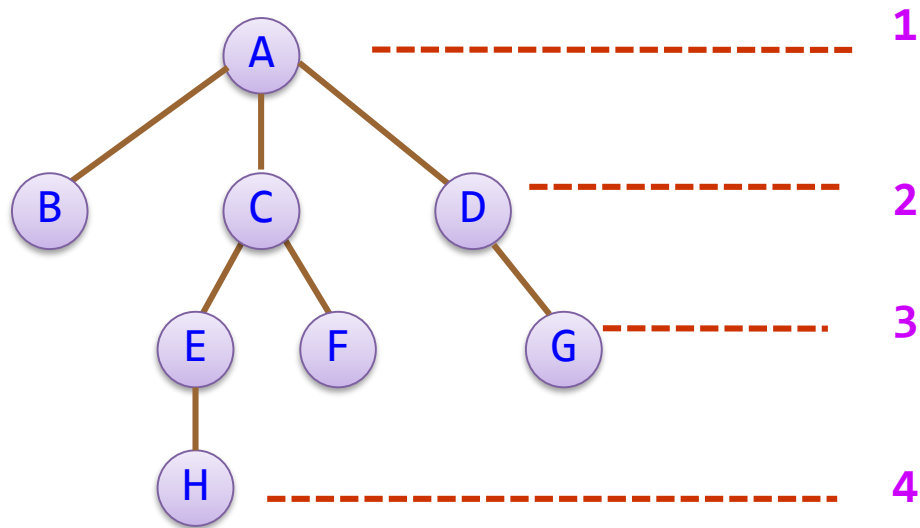
结点**F**的祖先结点为**A**、**C**

- **兄弟结点** 具有同一双亲的结点互相称之为兄弟结点。

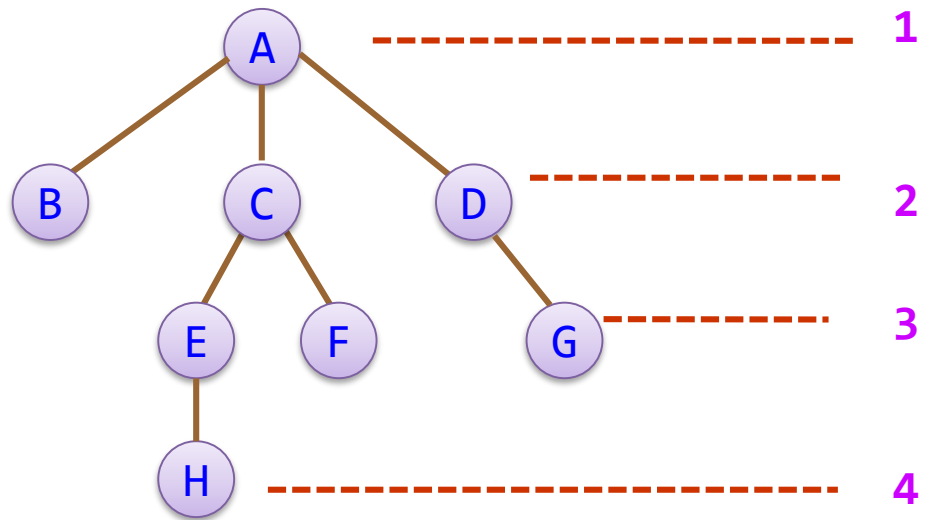


结点E和F是兄弟结点

- **结点层次** 树是一种层次结构，根结点为第一层，其孩子结点为第二层，以此类推得到每个结点的层次。

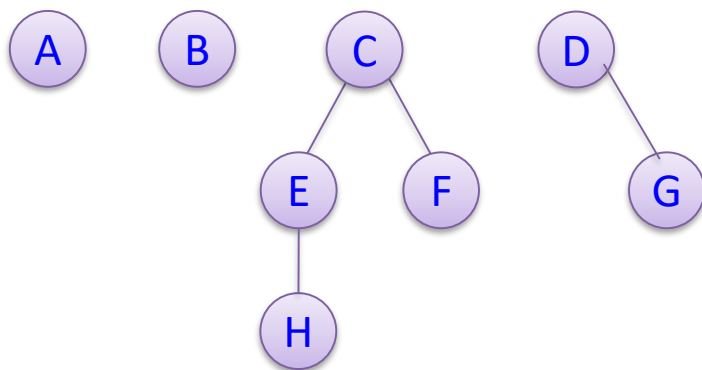


■ **树的高度** 树中结点的最大层次称为树的**高度或深度**。



高度是4

■ **森林** 零棵或多棵互不相交的树的集合称为森林。



4棵树构成的森林

任何一棵非空树可表示为一个二元组 $\text{Tree}=(\text{root}, F)$
其中 **root**为根节点;
F为子树构成的森林

6.1.4 树的基本运算

树的运算主要分为三大类：

- 查找满足某种特定关系的结点，如寻找当前结点的双亲结点等；
- 插入或删除某个结点，如在树的当前结点上插入一个新结点或删除当前结点的第 i 个孩子结点等；
- 遍历树中每个结点。

树的遍历运算是指按某种方式访问树中的每一个结点且每一个结点只被访问一次。

有以下3种遍历方法：

- 先根遍历
- 后根遍历
- 层次遍历

● 先根遍历:

若树不空，则先访问根结点，然后依次先根遍历各棵子树。

● 后根遍历:

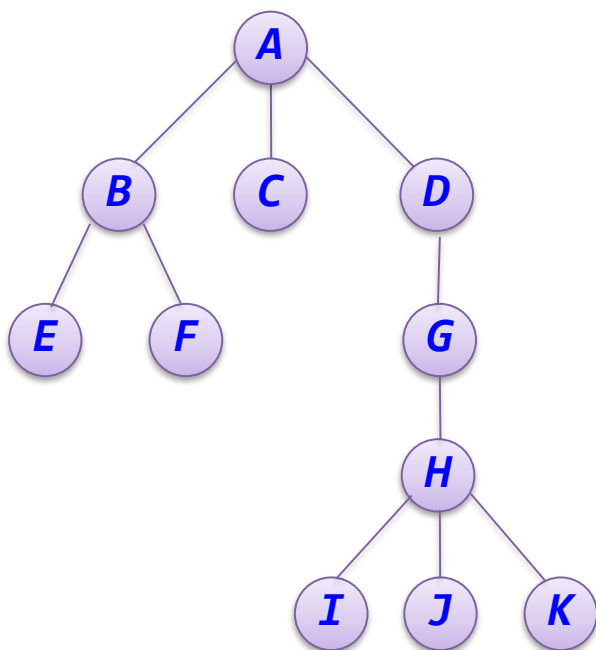
若树不空，则先依次后根遍历各棵子树，然后访问根结点。

● 层次遍历:

若树不空，则自上而下自左至右访问树中每个结点。

注意

先根和后根遍历算法都是递归的。



先根遍历的顶点访问次序:

A B E F C D G H I J K

后根遍历的顶点访问次序:

E F B C I J K H G D A

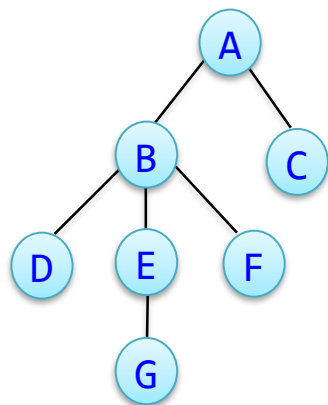
层次遍历的顶点访问次序:

A B C D E F G H I J K

6.1.5 树的存储结构

1. 双亲存储结构

这种存储结构是一种顺序存储结构，用一组连续空间存储树的所有结点，同时每个结点中附设一个伪指针指示其双亲结点的位置。



位置

0

data

parent

A

-1

1

B

0

2

C

0

3

D

1

4

E

1

5

F

1

6

G

4



`t=[['A',-1], ['B',0], ['C',0], ['D',1], ['E',1], ['F',1], ['G',4]]`

双亲存储结构

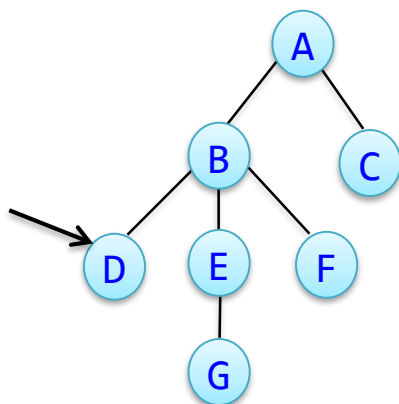
优缺点

- 利用了每个结点（根结点除外）只有唯一双亲的性质。
- 这种存储结构中，求某个结点的双亲结点十分容易，但求某个结点的孩子结点时需要遍历整个结构。

【例6.2】 若一棵树采用双亲存储结构 t 存储，设计一个算法求指定索引是 i 的结点的层次。

例

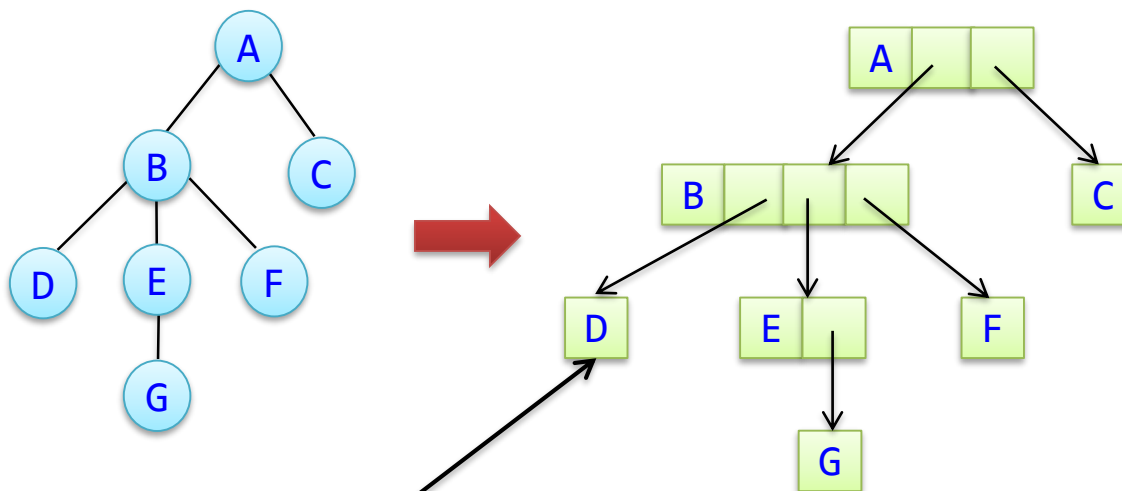
层次：移动到根
经过的边数+1



```
def Level(t,i):  
    assert i>=0 and i<len(t)           #求t中索引i的结点的层次  
                                         #检测参数  
    cnt=1  
    while t[i][1]!=-1:                  #没有到达根结点时循环  
        cnt+=1  
        i=t[i][1]                       #移动到双亲结点  
    return cnt
```

2. 孩子链存储结构

每个结点包含结点值和所有孩子结点指针，可按一个结点的度设计结点的孩子结点指针个数。**Python**中所有孩子结点指针用列表表示。



```
class SonNode:
    def __init__(self,d=None):
        self.data=d
        self.sons=[]
```

#孩子链存储结构结点类
#构造方法
#结点的值
#指向孩子结点的指针列表

孩子链存储结构

优缺点

- 优点是查找某结点的孩子结点十分方便。
- 缺点是查找某结点的双亲结点比较费时。

【例6.3】若一棵树采用孩子链存储结构 t 存储，设计一个算法求其高度。

解：一棵树的高度为根的所有子树高度的最大值加1。求整棵树的高度为“大问题”，求每棵子树高度为“小问题”。设 $f(t)$ 为求树 t 的高度，对应的递归模型如下：

$$f(t) = 0$$

当 t 为空树

$$f(t) = 1$$

当 t 为叶子结点时

$$f(t) = \max_i f(t.\text{sons}[i]) + 1$$

其他情况



```
def Height(t):
```

```
    if t==None: return 0;
```

```
    if len(t.sons)==0: return 1
```

```
    maxsh=0
```

```
    for i in range(len(t.sons)):
```

```
        sh=Height(t.sons[i])
```

```
        maxsh=max(maxsh,sh)
```

```
    return maxsh+1
```

#求 t 的高度

#空树高度为0

#叶子结点的高度为1

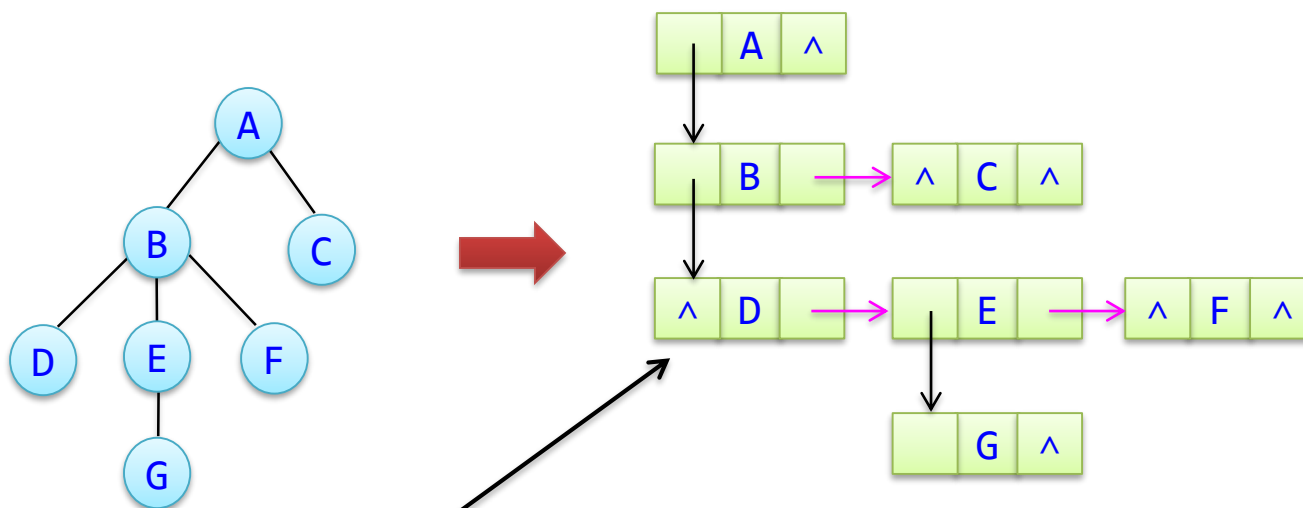
#遍历所有子树

#求子树 $t.\text{sons}[i]$ 的高度

#求所有子树的最大高度

3. 长子兄弟链存储结构

长子兄弟链存储结构是为每个结点设计三个域：一个数据元素域，一个指向该结点的长子的指针域，一个指向该结点的下一个兄弟结点指针域。



```
class EBNode:
```

```
    def __init__(self, d=None):
```

```
        self.data=d
```

```
        self.brother=None
```

```
        self.eson=None
```

#长子兄弟链中结点类

#构造方法

#结点的值

#指向兄弟

#指向长子结点

长子兄弟链存储结构

优缺点

- 优点是查找某结点的孩子结点十分方便。
- 缺点是查找某结点的双亲结点比较费时。

【例6.4】若一棵树采用长子兄弟链存储结构 t 存储，设计一个算法求其高度。

解：一棵树的高度为根的所有子树高度的最大值加1。求整棵树的高度为“大问题”，求每棵子树高度为“小问题”。设 $f(t)$ 为求树 t 的高度，对应的递归模型如下：

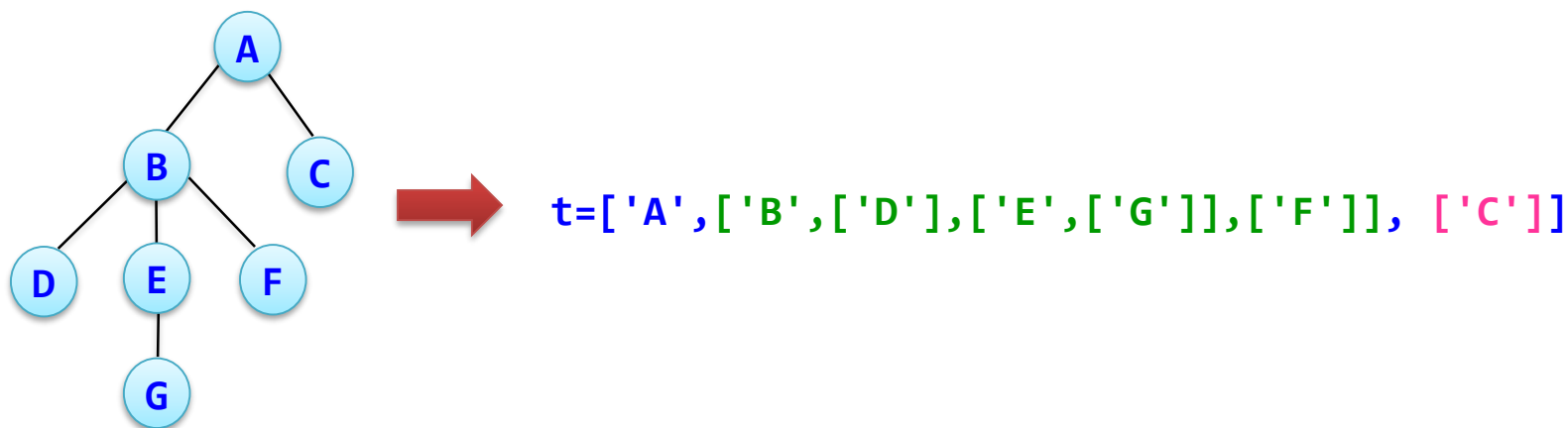
$f(t) = 0$	当 t 为空树
$f(t) = 1$	当 t 为叶子结点时
$f(t) = \max_i f(t.\text{sons}[i]) + 1$	其他情况



def Height(t):	#求 t 的高度
if t==None: return 0	#空树返回0
maxsh=0	
p=t.eson;	#p指向 t 结点的长子
while p!=None:	
q=p.brother	#q临时保存结点p的后继结点
sh=Height(p)	#递归求结点p的子树的高度
maxsh=max(maxsh, sh)	#求结点t的所有子树的最大高度
p=q	
return maxsh+1	#返回maxsh+1

4. 列表存储结构

将树的列表表示（树的逻辑表示法之一）直接采用Python中的列表数据类型表示，称为树的**列表存储结构**。



树的列表存储结构简单直观，其优缺点与孩子链存储结构相同。

【例6.5】若一棵树采用列表存储结构 t 存储，设计一个算法求其高度。

解：一棵树的高度为根的所有子树高度的最大值加1。求整棵树的高度为“大问题”，求每棵子树高度为“小问题”。设 $f(t)$ 为求树 t 的高度，对应的递归模型如下：

$f(t) = 0$	当 t 为空树
$f(t) = 1$	当 t 为叶子结点时
$f(t) = \max_i f(t.\text{sons}[i]) + 1$	其他情况



def Height(t):	#求 t 的高度
if len(t)==1: return 1	#叶子结点高度为1
m=len(t)	
maxsh=0	
for i in range(1,m):	#遍历所有子树
sh=Height(t[i])	#求子树 $t[i]$ 的高度
maxsh=max(maxsh,sh)	#求所有子树的最大高度
return maxsh+1	