



Data Structure & Algorithm Analysis

Trees & Binary Trees

Zibin Zheng (郑子彬)

School of Data and Computer Science , SYSU

<http://www.inpluslab.com>

课程主页: <http://inpluslab.sysu.edu.cn/dsa2016/>

Tree

- 1 Binary trees
- 2 Traversal of binary trees
- 3 Application: binary trees
- 4 Tree Specifications and term
- 5 Tree and forest

线性结构

- 线性结构：线性结构是一个有序数据元素的**集合**。

- 顺序表
- 链表
- 栈
- 队列
- 字符串
- 数组

特征：

1. 集合中必存在唯一的一个"第一个元素"；
2. 集合中必存在唯一的一个"最后的元素"；
3. 除最后元素之外，其它数据元素均有唯一的"后继"；
4. 除第一元素之外，其它数据元素均有唯一的"前驱"。

数据结构中线性结构指的是数据元素之间存在着“一对一”的线性关系的数据结构。

如 $(a_1, a_2, a_3, \dots, a_n)$ ， a_1 为第一个元素， a_n 为最后一个元素，此集合即为一个线性结构的集合。

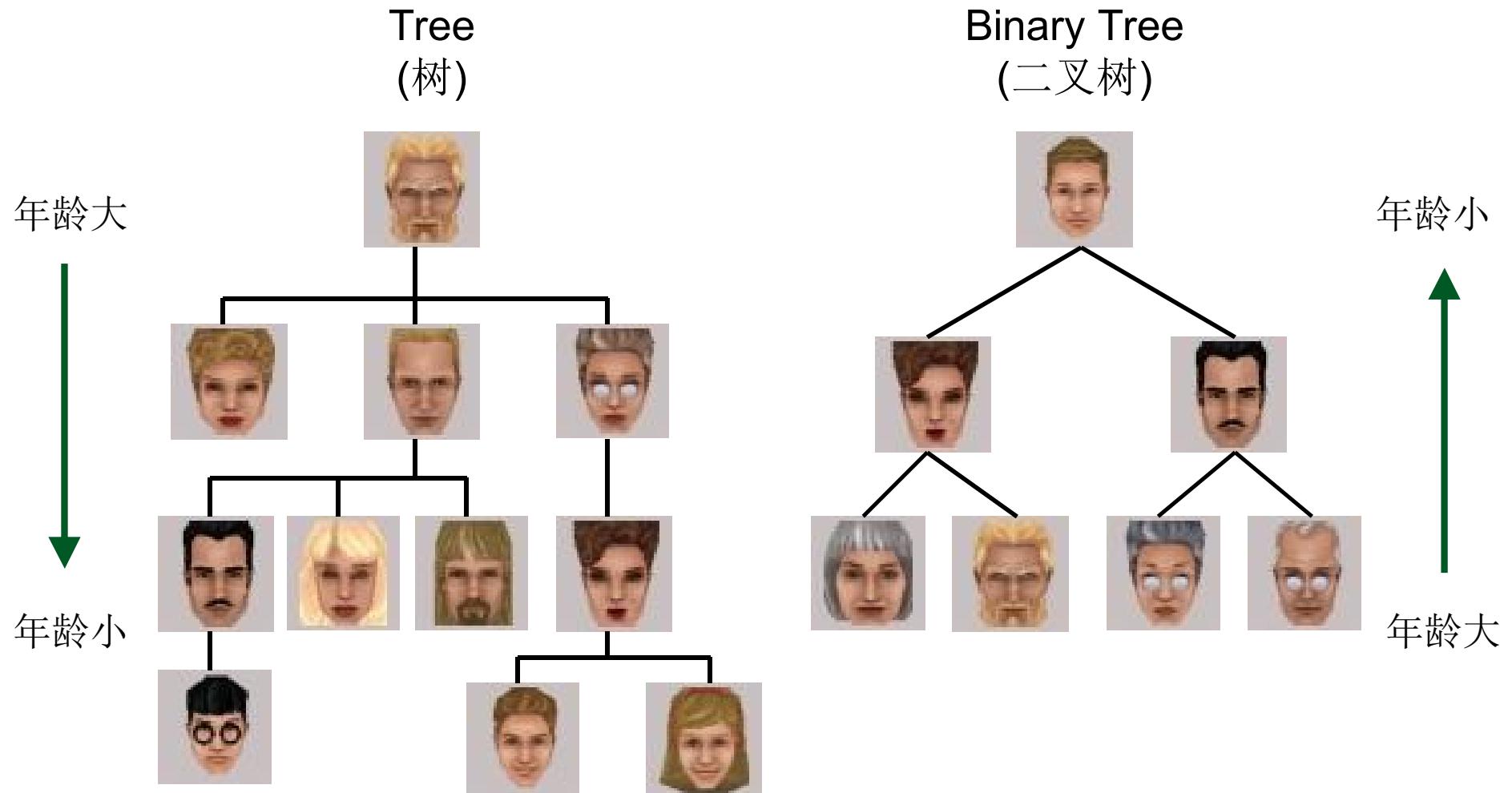
非线性结构

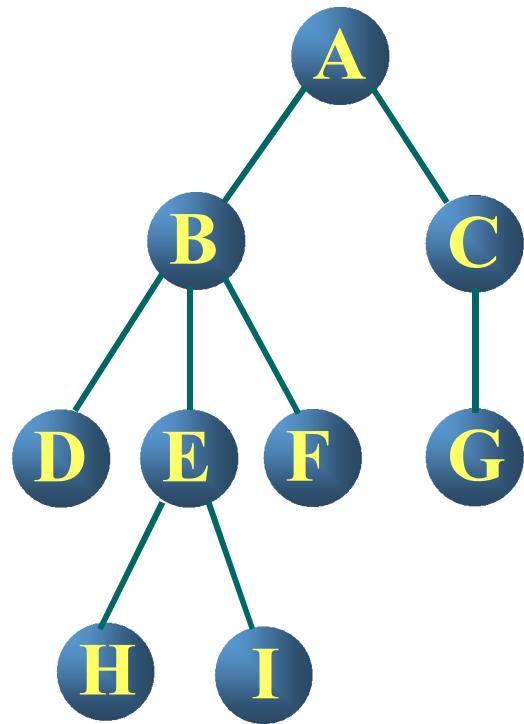
- 所谓**非线性结构**是指，在该结构中至少存在一个数据元素，有两个或两个以上的直接前驱(或直接后继)元素。相对应于线性结构，非线性结构的逻辑特征是一个结点元素可能对应多个直接前驱和多个后继。
- 树型结构和图型就是其中十分重要的非线性结构，可以用来描述**客观世界**中广泛存在的层次结构和网状结构的关系，如家族谱、城市交通等。
- 树在**计算机领域**中也有着广泛的应用，例如在编译程序中，用树来表示源程序的语法结构；在数据库系统中，可用树来组织信息；在分析算法的行为时，可用树来描述其执行过程等等。

对树的认识

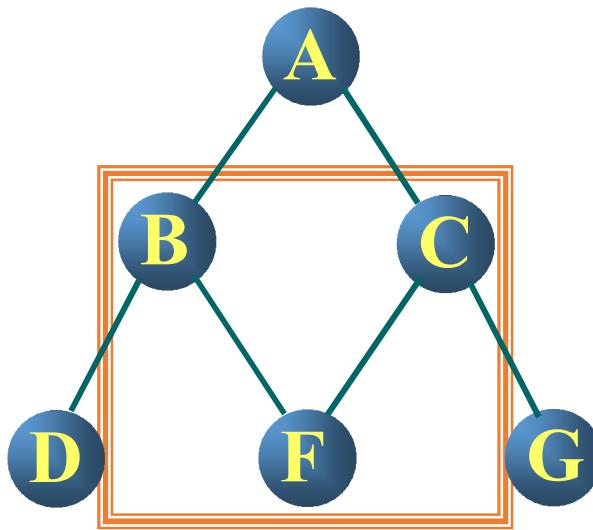


倒过来的树

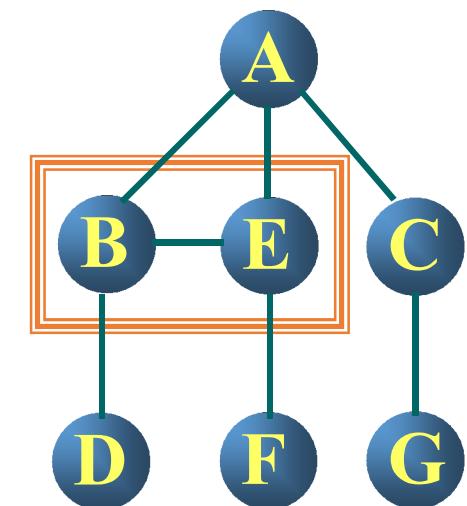




(a) 一棵树结构

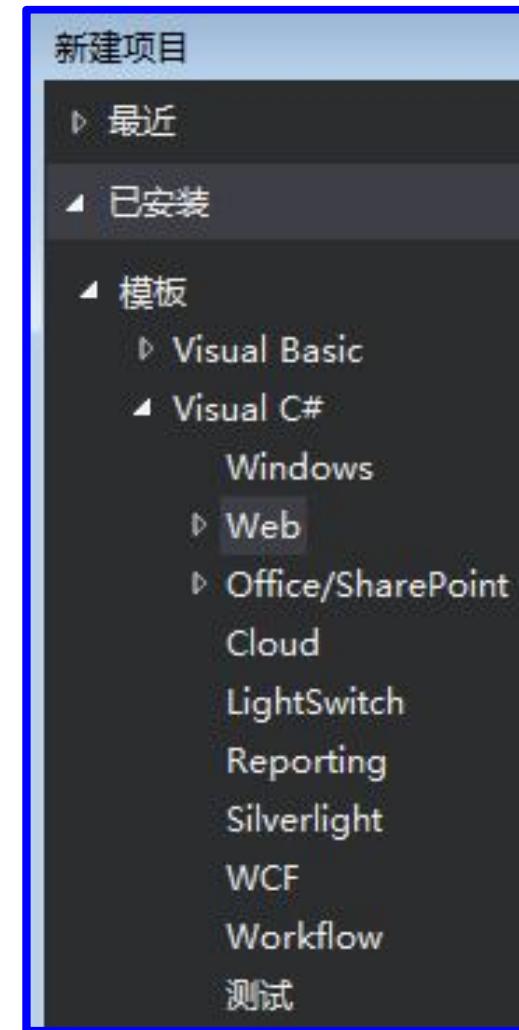
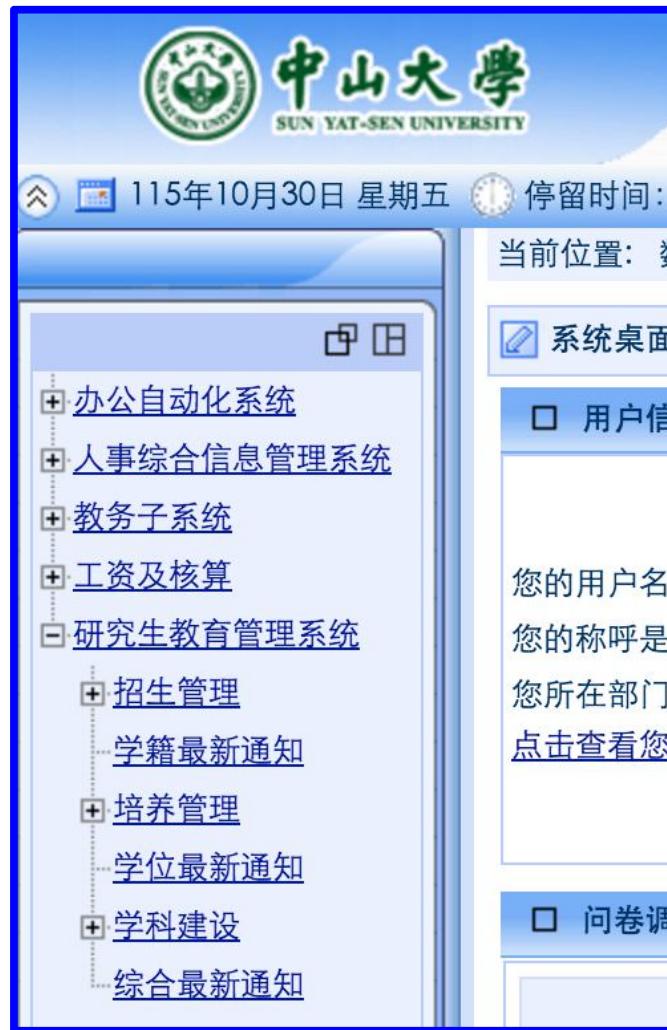
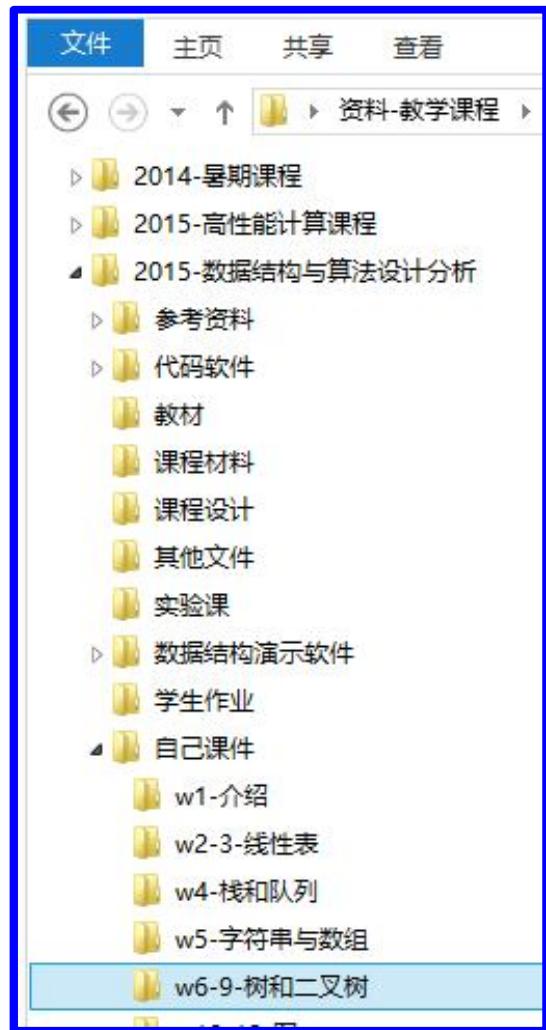


(b)一个非树结构



(c)一个非树结构

计算机应用中的树

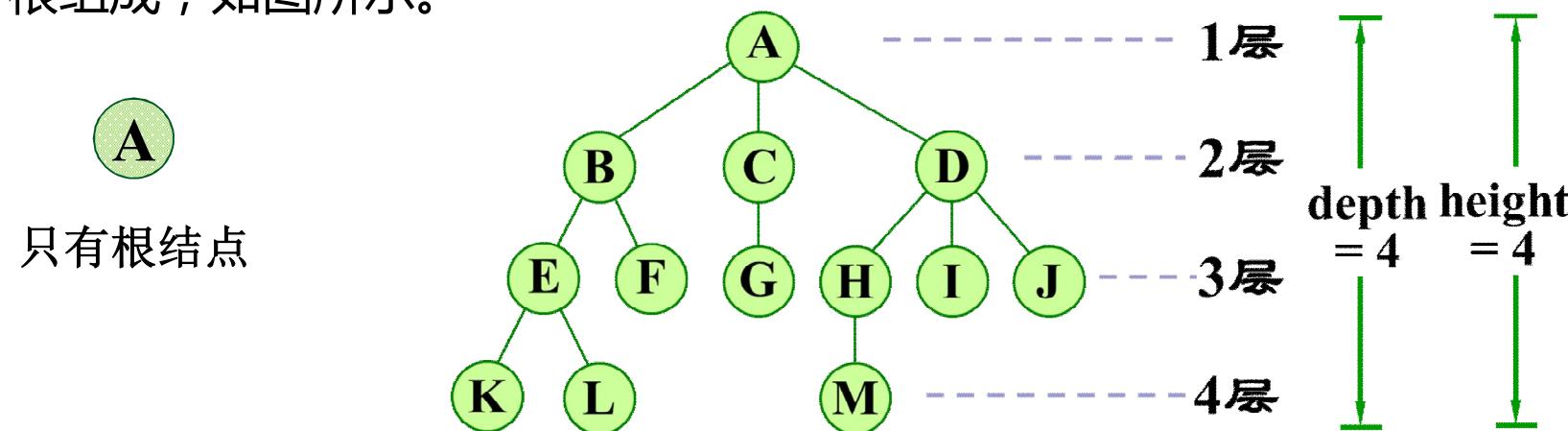


树的定义

树(Tree)是 $n(n \geq 0)$ 个结点的有限集合 T ，若 $n=0$ 时称为空树，否则：

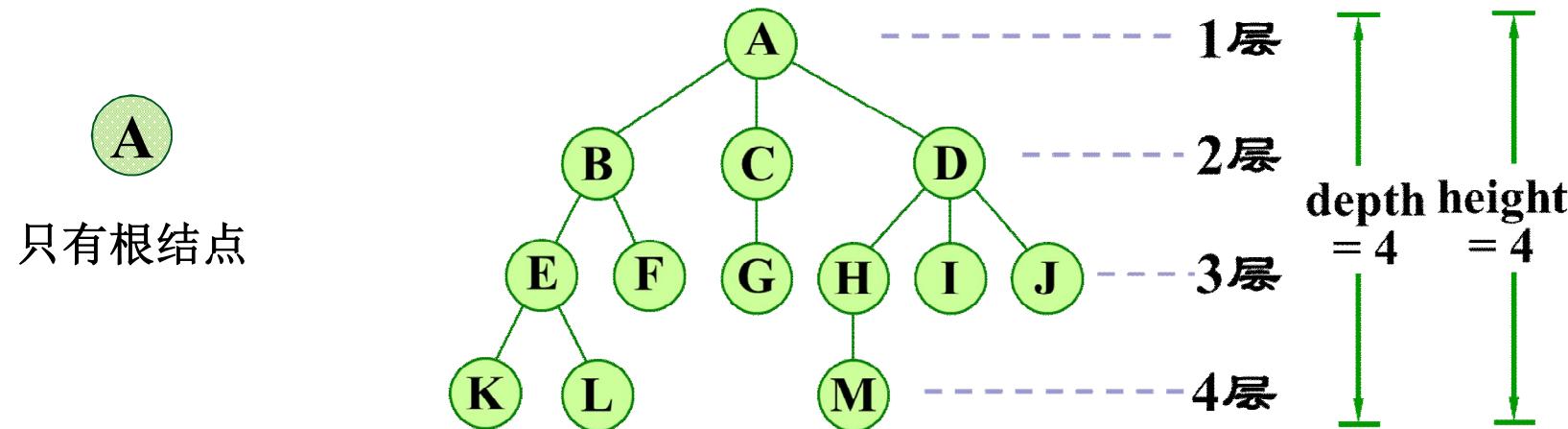
- (1) 有且只有一个特殊的称为树的根(Root)结点；
- (2) 若 $n > 1$ 时，其余的结点被分为 $m(m > 0)$ 个互不相交的子集 $T_1, T_2, T_3 \dots T_m$ ，其中每个子集本身又是一棵树，称其为根的子树(Subtree)。

这是树的递归定义，即用树来定义树，而只有一个结点的树必定仅由根组成，如图所示。



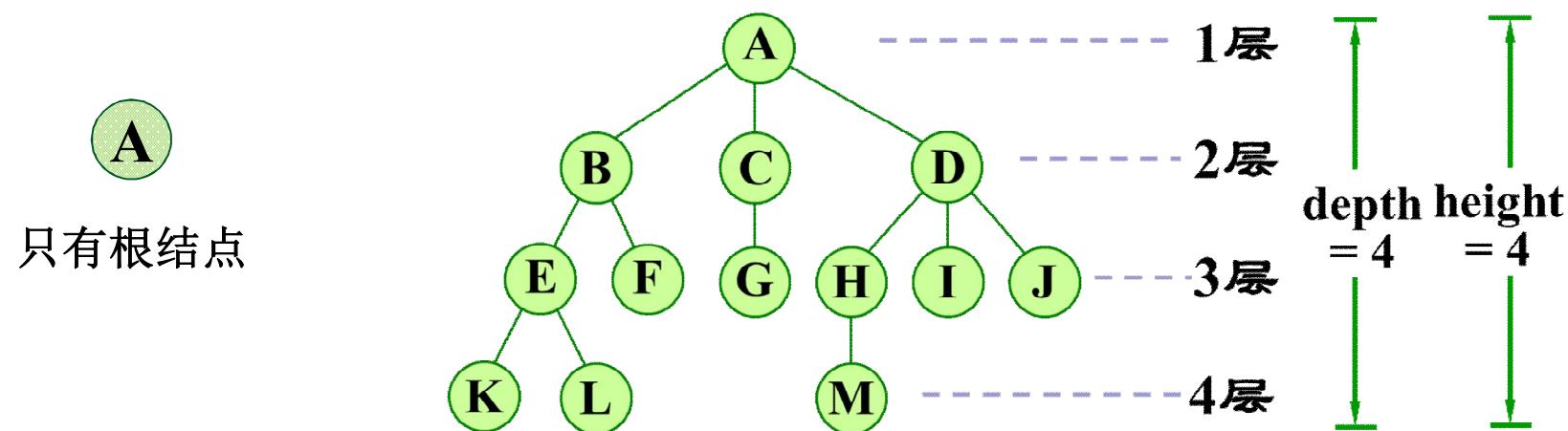
树的相关概念

- **结点(node)** : 一个数据元素及其若干指向其子树的分支。
- **结点的度(degree)** : 结点所拥有的子树的个数称为该结点的度。
- **树的度** : 树中各结点度的最大值称为该树的度 , 下图中为3。



树的相关概念

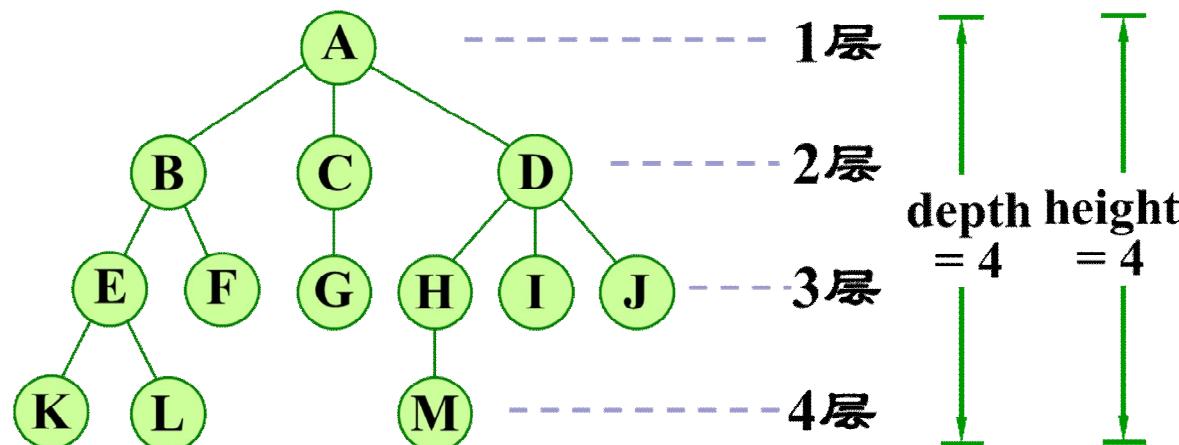
- **叶子(left)结点**：树中度为0的结点称为叶子结点(或终端结点)。
- **非叶子结点**：度不为0的结点称为非叶子结点(或非终端结点或分支结点)。除根结点外，分支结点又称为内部结点。
- 如图中结点F、G、I、J、K、L、M是叶子结点，而所有其它结点都是分支结点。



树的相关概念

孩子结点、双亲结点、兄弟结点

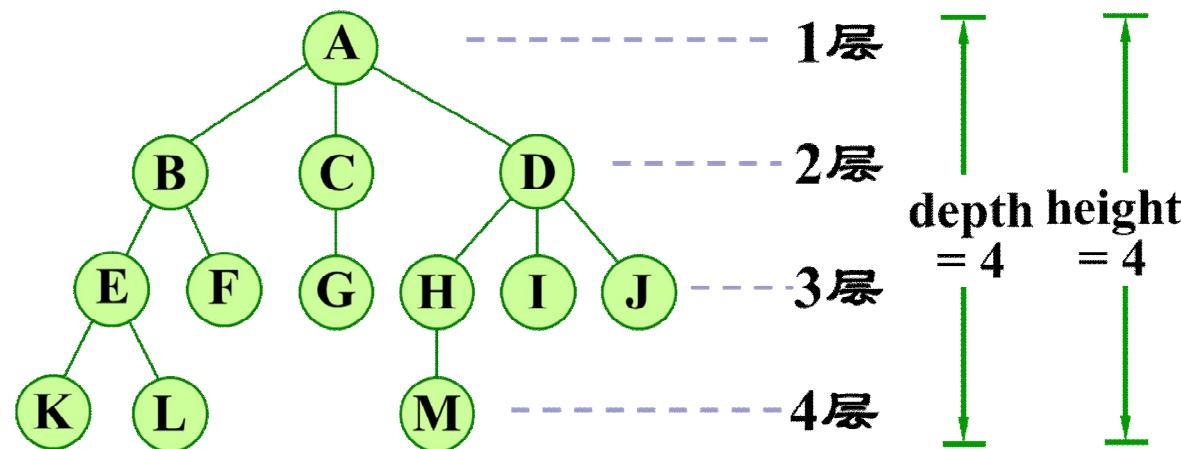
- 一个结点的子树的根称为该结点的孩子结点(child)或子结点；相应地，该结点是其孩子结点的**双亲结点(parent)**或父结点。
 - 如图中结点B、C、D是结点A的子结点，而结点A是结点B、C、D的父结点；类似地结点E、F是结点B的子结点，结点B是结点E、F的父结点。
- 同一双亲结点的所有子结点互称为**兄弟结点**。
 - 如图中结点B、C、D是兄弟结点；结点E、F是兄弟结点。



树的相关概念

层次、堂兄弟结点

- 规定树中根结点的层次为1，其余结点的**层次**等于其双亲结点的层次加1。
- 若某结点在第 $l(l \geq 1)$ 层，则其子结点在第 $l+1$ 层。
- 双亲结点在同一层上的所有结点互称为**堂兄弟结点**。如图中结点G与E、F、H、I、J互为堂兄弟。

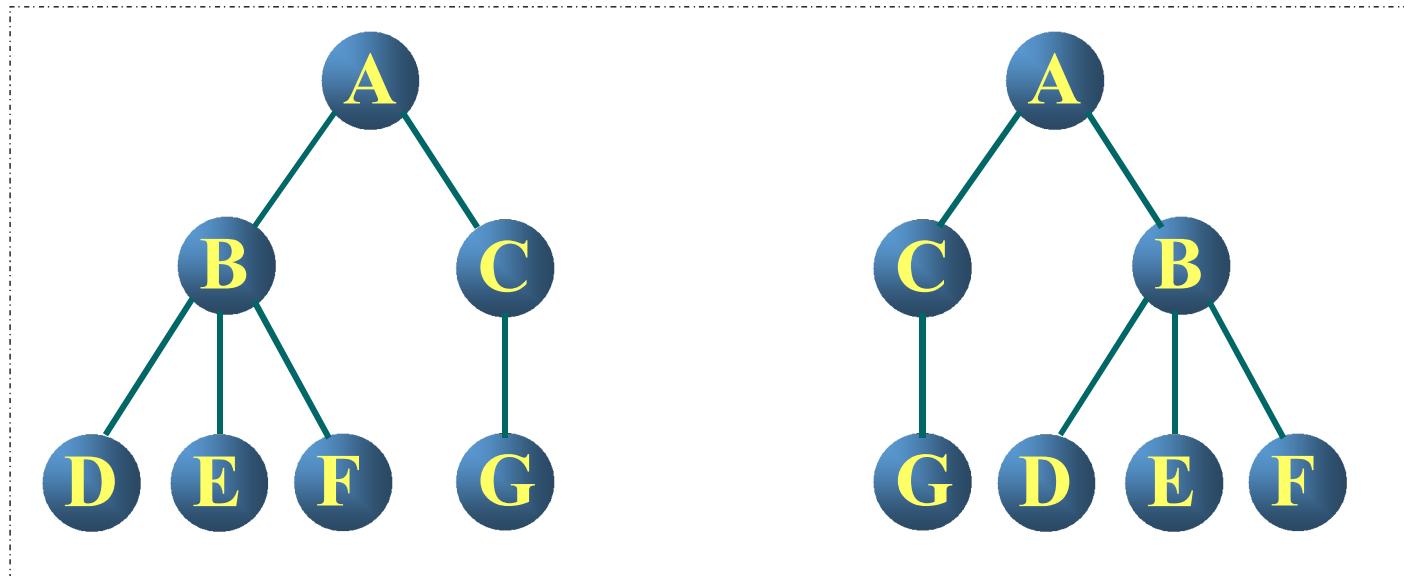


树的相关概念

结点的层次路径、祖先、子孙

- 从根结点开始，到达某结点p所经过的所有结点成为结点p的**层次路径**（有且只有一条）。
- 结点p的层次路径上的所有结点（p除外）称为p的**祖先**(ancestor)。
- 以某一结点为根的子树中的任意结点称为该结点的**子孙结点**(descent)。
- 树的深度(depth)**：树中结点的最大层次值，又称为树的高度，如图中树的高度为4。
- 有序树和无序树**：对于一棵树，若其中每一个结点的子树（若有）具有一定的次序，则该树称为有序树，否则称为无序树。
- 森林(forest)**：是 $m(m \geq 0)$ 棵互不相交的树的集合。显然，若将一棵树的根结点删除，剩余的子树就构成了森林。

有序树、无序树：如果一棵树中结点的各子树从左到右是有次序的，称这棵树为有序树；反之，称为无序树。

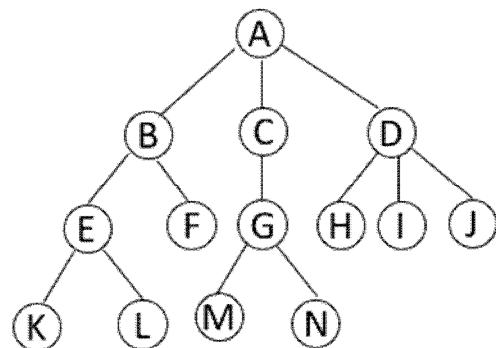


数据结构中讨论的一般都是有序树

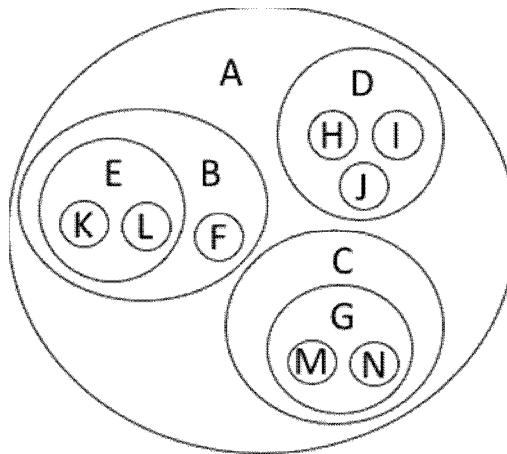
树的表示形式

树的表示形式

- (1) 倒悬树。是最常用的表示形式，如图(a)。
- (2) 嵌套集合。是一些集合的集体，对于任何两个集合，或者不相交，或者一个集合包含另一个集合。图(b)是树的嵌套集合形式。
- (3) 广义表形式。图(c)是树的广义表形式。



(a) 倒悬树



(b) 嵌套集合形式

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

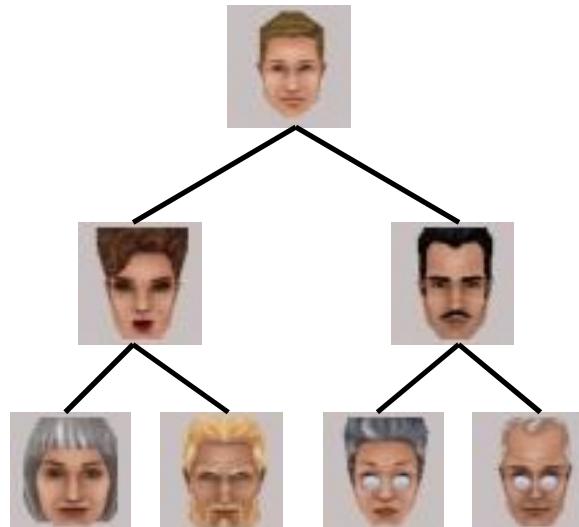
(c) 广义表

Tree

- 1 Binary trees
- 2 Traversal of binary trees
- 3 Application: binary trees
- 4 Tree Specifications and term
- 5 Tree and forest

Binary trees

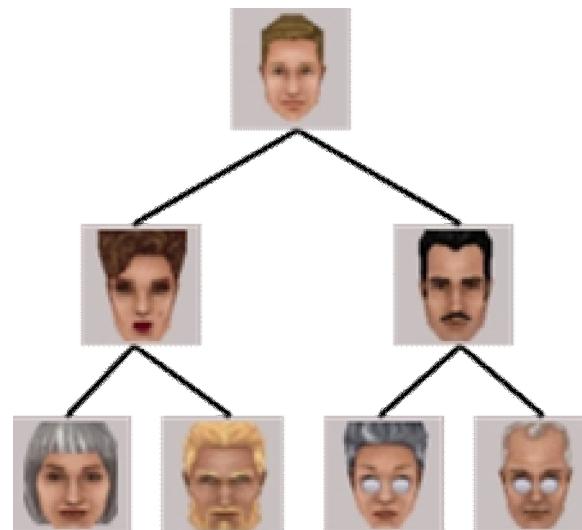
- 二叉树在树结构的应用中起着非常重要的作用，原因在于：
 - 对二叉树的许多操作算法简单
 - 任何树都可以与二叉树相互转换，这样就解决了树的存储结构及其运算中存在的复杂性。



Specifications of Binary Trees

- 定义

- 二叉树是由 $n(n \geq 0)$ 个结点的有限集合构成，此集合或者为**空集**，或者由一个根结点及两棵互不相交的**左右子树组成**，并且左右子树都是二叉树。
- 这是一个**递归定义**。二叉树可以是空集合，根可以有空的左子树或空的右子树。**二叉树不是树的特殊情况**，它们是两个概念。

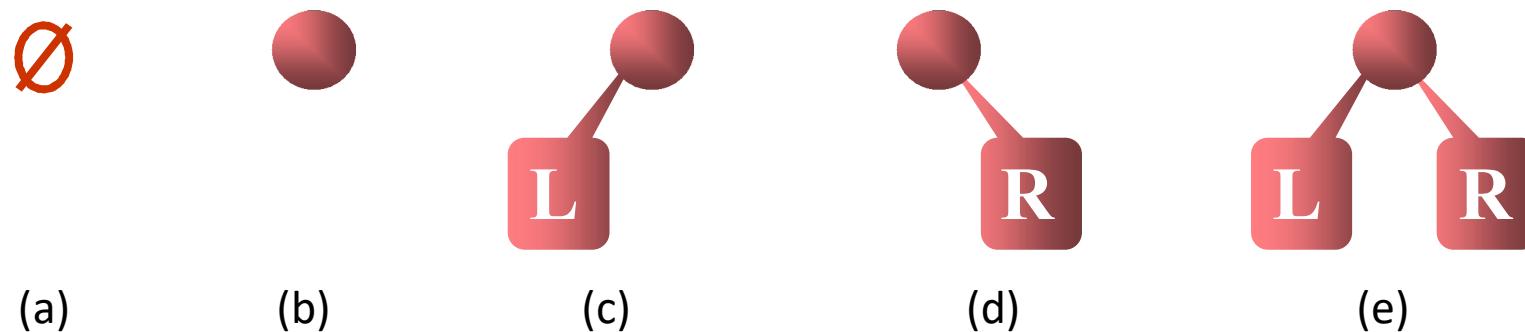


Specifications of Binary Trees

- **二叉树不是树的特例**
- 二叉树与无序树不同
 - 二叉树中，每个结点最多只能有两棵子树，并且有左右之分。二叉树并非是树的特殊情形，它们是两种不同的数据结构。
- 二叉树与度数为2的有序树不同
 - 在有序树中，虽然一个结点的孩子之间是有左右次序的，但是若该结点只有一个孩子，就无须区分其左右次序。
 - 而在二叉树中，即使是一个孩子也有左右之分。

Specifications of Binary Trees

- 二叉树结点的子树要区分左子树和右子树，即使只有一棵子树也要进行区分，说明它是左子树，还是右子树。**这是二叉树与树的最主要的区别。**
- 下图列出二叉树的5种基本形态，图(c)和图(d)是不同的两棵二叉树。



二叉树的五种不同形态

二叉树的相关概念

- **满二叉树**：在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶结点都在同一层上，这样的一棵二叉树称作满二叉树。

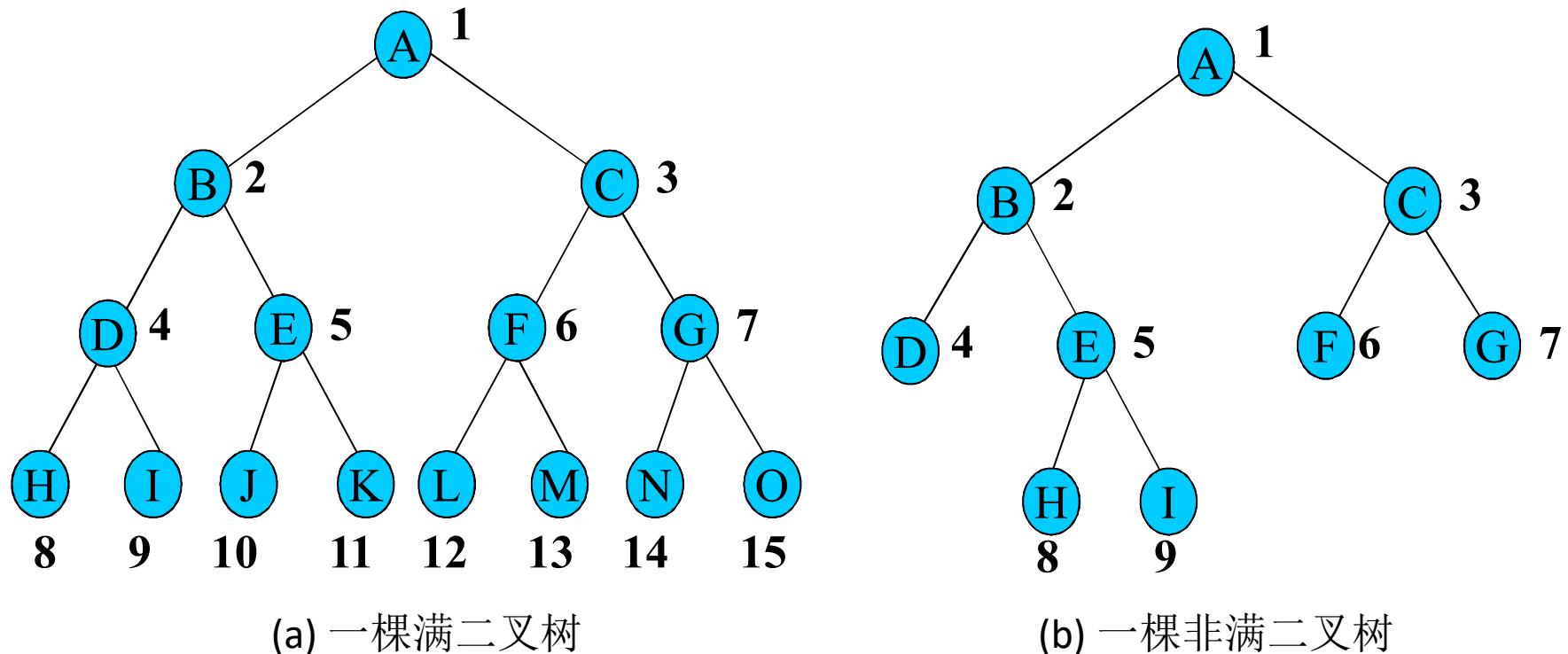
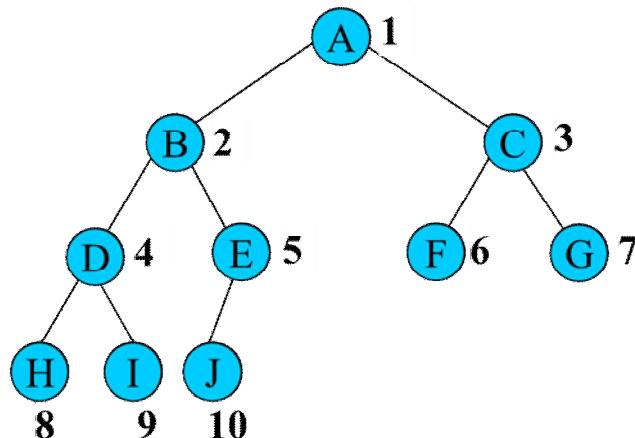


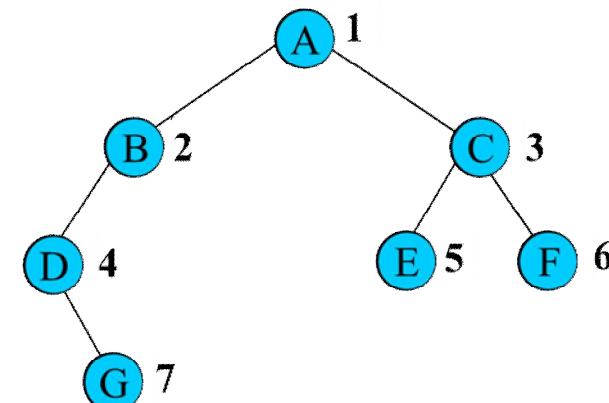
图6.2 满二叉树和非满二叉树示意图

二叉树的相关概念

- **完全二叉树**：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($1 \leq i \leq n$)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。
 - 特点：叶结点只能出现在最下层和次下层，且最下层的叶结点集中在树的左部。
- 显然，一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。如(a)为一棵完全二叉树，图(b)不是完全二叉树。



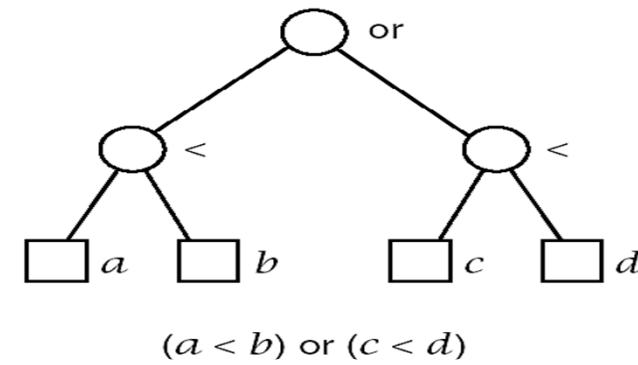
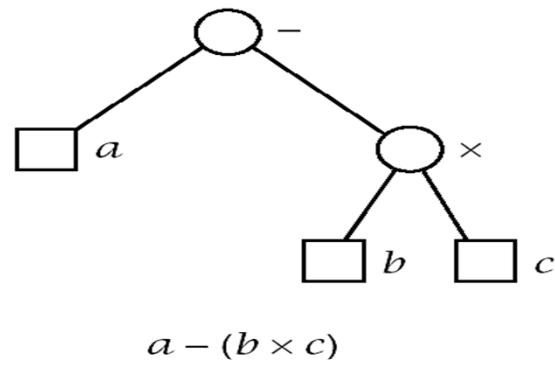
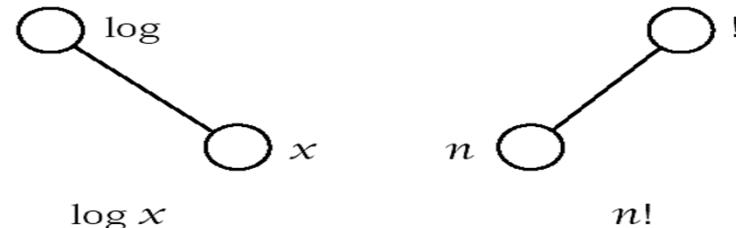
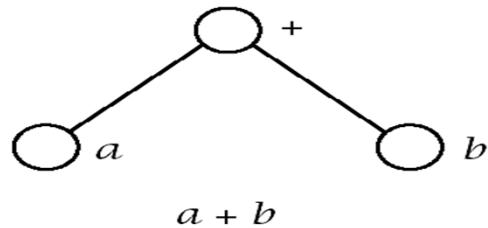
(a) 一棵完全二叉树



(b) 一棵非完全二叉树

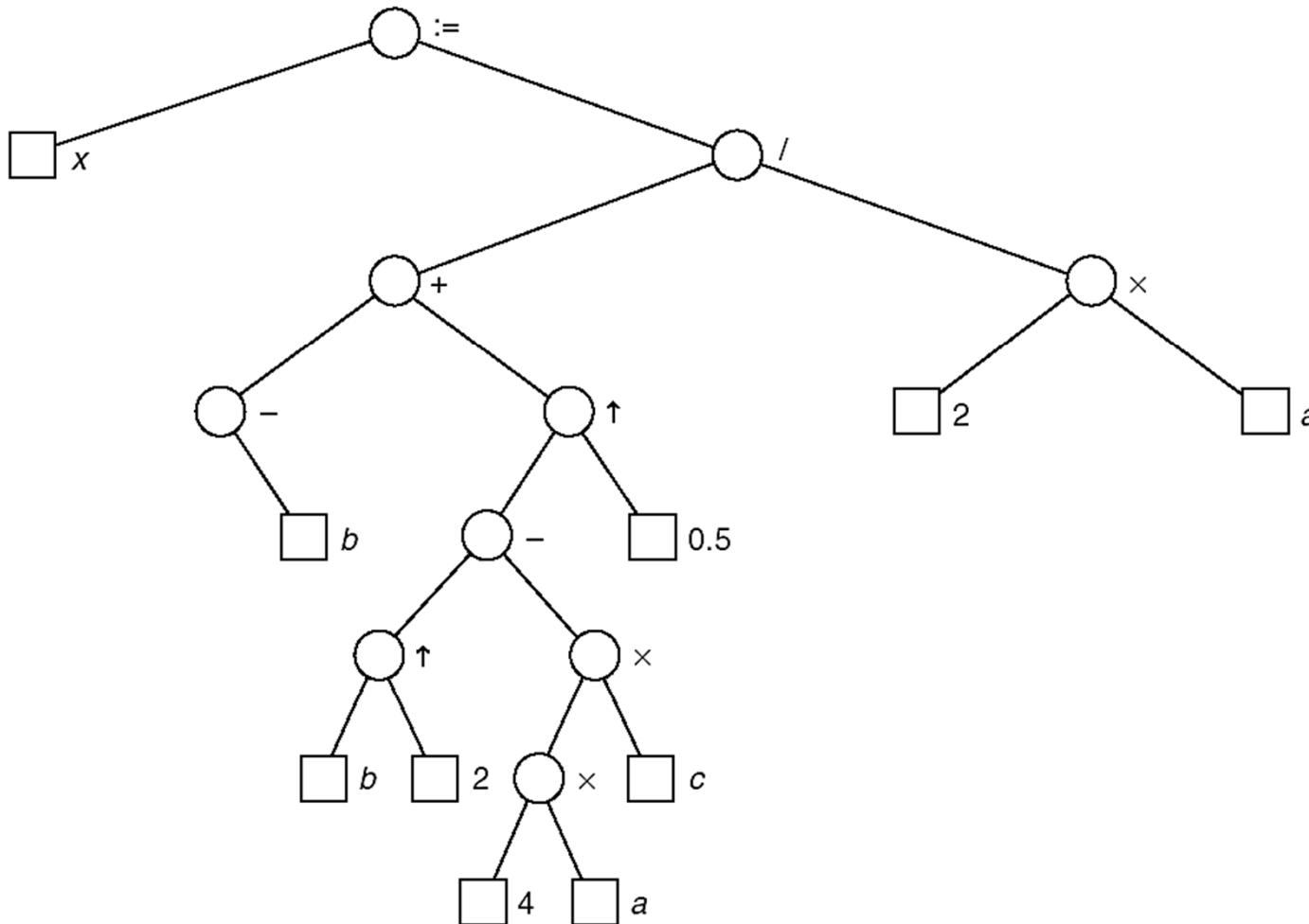
图6.3 完全二叉树和非完全二叉树示意图

应用：表达式树



<i>Expression:</i>	$a + b$	$\log x$	$n!$	$a - (b \times c)$	$(a < b) \text{ or } (c < d)$
<i>Preorder :</i>	$+ \ a \ b$	$\log \ x$	$! \ n$	$- \ a \ \times \ b \ c$	$\text{or} \ < \ a \ b \ < \ c \ d$
<i>Inorder :</i>	$a + b$	$\log x$	$n !$	$a - b \times c$	$a < b \text{ or } c < d$
<i>Postorder :</i>	$a \ b +$	$x \ \log$	$n !$	$a \ b \ c \ \times -$	$a \ b < c \ d < \text{or}$

应用：表达式树



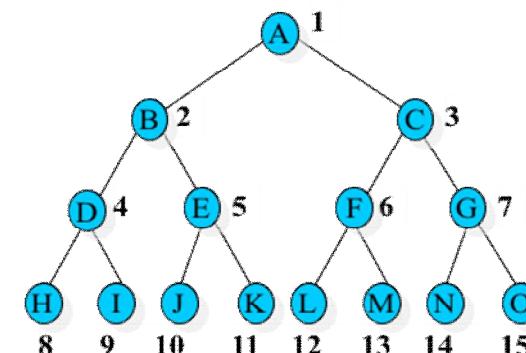
$$x := (-b + (b \uparrow 2 - 4 \times a \times c) \uparrow 0.5) / (2 \times a)$$

Properties of Binary Trees

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

【证明】采用归纳法证明此性质。

- 当*i=1*时，只有一个根结点， $2^{i-1}=2^0=1$ ，命题成立。
- 现在假定对所有的*j*, $1 \leq j < i$, 命题成立，即第*j*层上至多有 2^{j-1} 个结点，那么可以证明*j=i*时命题也成立。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，故在第*i*层上最大结点数为第*i-1*层上最大结点数的二倍，即 $2 \times 2^{i-2} = 2^{i-1}$ 。
- 命题得到证明。

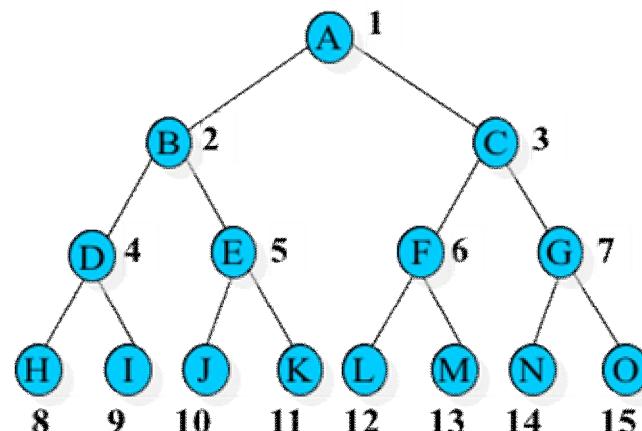


Properties of Binary Trees

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

【证明】设第 i 层的结点数为 x_i ($1 \leq i \leq k$)，深度为 k 的二叉树的结点数为 M ， x_i 最多为 2^{i-1} ，则有：

$$M = \sum_{i=1}^k x_i \leq \sum_{i=1}^k 2^{i-1} = 2^k - 1$$



Properties of Binary Trees

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

【证明】设 n 为二叉树的结点总数， n_1 为二叉树中度为1的结点数，则有：

$$n = n_0 + n_1 + n_2 \quad (1)$$

- 在二叉树中，除根结点外，其余结点都有唯一的一个进入分支。设 B 为二叉树中的分支数，那么有：

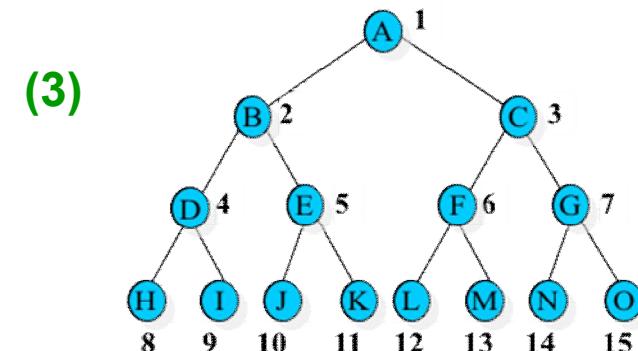
$$B = n - 1 \quad (2)$$

- 这些分支是由度为1和度为2的结点发出的，一个度为1的结点发出一个分支，一个度为2的结点发出两个分支，所以有：

$$B = n_1 + 2n_2$$

- 综合(1)、(2)、(3)式可以得到：

$$n_0 = n_2 + 1$$



(3)

Properties of Binary Trees

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

【证明】根据完全二叉树的定义和性质2可知，当一棵完全二叉树的深度为k、结点个数为n时，有

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

即

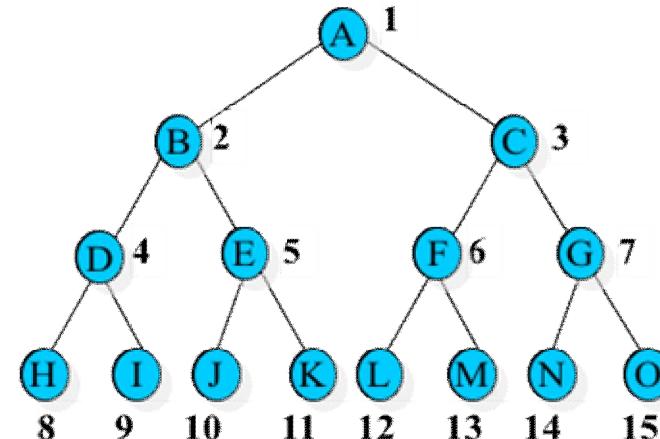
$$2^{k-1} \leq n < 2^k$$

对不等式取对数，有

$$k-1 \leq \log_2 n < k$$

由于k是整数，所以有 $k = \lfloor \log_2 n + 1 \rfloor$ 。

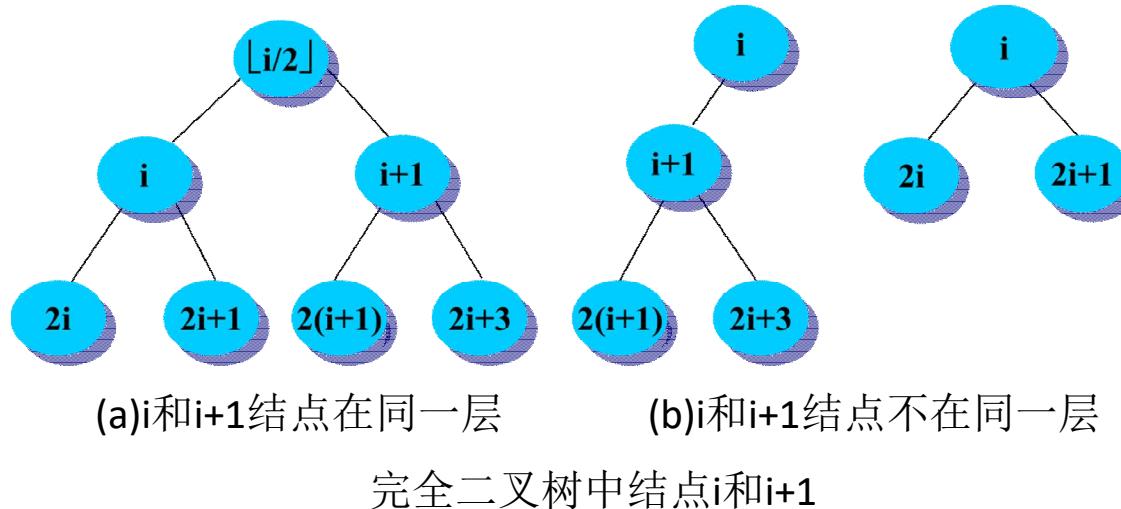
注：符号 $\lfloor x \rfloor$ 表示不大于x的最大整数。



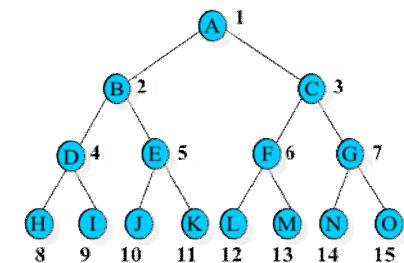
Properties of Binary Trees

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

- (1)如果 $i=1$ ，则结点*i*无双亲，是二叉树的根；如果 $i>1$ ，则其双亲是结点 $\lfloor i/2 \rfloor$ 。
- (2)如果 $2i>n$ ，则结点*i*为叶结点，无左孩子；否则，其左孩子是结点 $2i$ 。
- (3)如果 $2i+1>n$ ，则结点*i*无右孩子；否则，其右孩子是结点 $2i+1$ 。



- 在此过程中，可以从(2)和(3)推出(1)，所以先证明(2)和(3)。



Properties of Binary Trees

【证明】

- 对于 $i=1$, 由完全二叉树的定义, 其左孩子是结点2, 若 $2>n$, 即不存在结点2, 此时结点*i*无孩子。结点*i*的右孩子也只能是结点3, 若结点3不存在, 即 $3>n$, 此时结点*i*无右孩子。
- 对于 $i>1$, 可分为两种情况:
 1. 设第 $j(1 \leq j \leq \lfloor \log_2 n \rfloor)$ 层的第一个结点的编号为*i*, 由二叉树的性质2和定义知 $i=2^{j-1}$ 。结点*i*的左孩子必定为的 $j+1$ 层的第一个结点, 其编号为 $2^j=2 \times 2^{j-1}=2i$ 。如果 $2i>n$, 则无左孩子; 其右孩子必定为第 $j+1$ 层的第二个结点, 编号为 $2i+1$ 。若 $2i+1>n$, 则无右孩子。
 2. 假设第 $j(1 \leq j \leq \lfloor \log_2 n \rfloor)$ 层上的某个结点编号为*i* ($2^{j-1} \leq i \leq 2^j-1$), 且 $2i+1 < n$, 其左孩子为 $2i$, 右孩子为 $2i+1$, 则编号为*i+1*的结点时编号为*i*的结点的右兄弟或堂兄弟。若它有左孩子, 则其编号必定为 $2i+2=2 \times (i+1)$: 若它有右孩子, 则其编号必定为 $2i+3=2 \times (i+1)+1$ 。

Properties of Binary Trees

【证明】

- 当*i*=1时，就是根，因此无双亲，当*i*>1时，如果*i*为左孩子，即 $2 \times (i/2) = i$ ，则*i/2*是*i*的双亲；如果*i*为右孩子， $i = 2p + 1$ ，*i*的双亲应为

， $p = (i-1)/2 = \lfloor i/2 \rfloor$ 。证毕。

1. 顺序存储结构

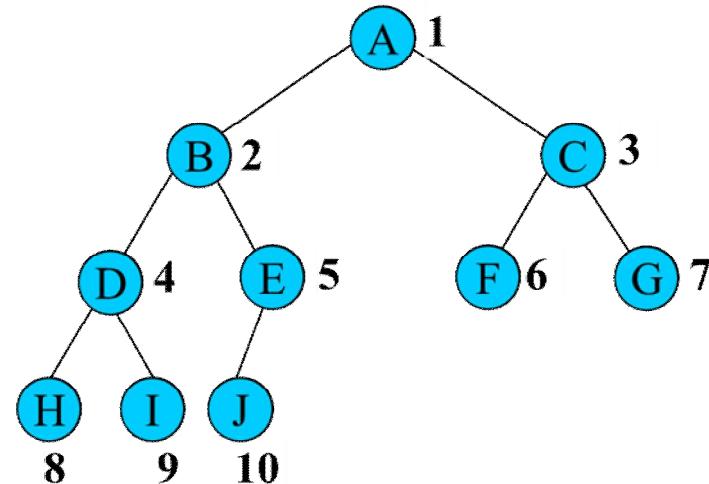
- 用一维数组存储存放二叉树中的结点。一般是按照二叉树结点从上至下、从左到右的顺序存储。
- 这样结点在存储位置上的前驱后继关系并不一定就是它们在逻辑上的邻接关系，然而只有通过一些方法确定某结点在逻辑上的前驱结点和后继结点，这种存储才有意义。
- 因此，依据二叉树的性质，完全二叉树和满二叉树采用顺序存储比较合适，树中结点的序号可以唯一地反映出结点之间的逻辑关系，这样既能够最大可能地节省存储空间，又可以利用数组元素的下标值确定结点在二叉树中的位置，以及结点之间的关系。

Implementation of Binary Trees

- 图6.5给出的图6.3(a)所示的完全二叉树的顺序存储示意。

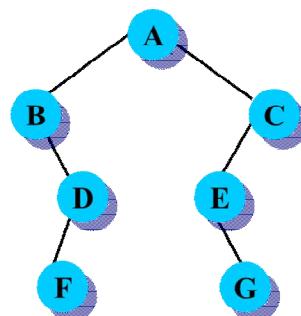


- 数组下标 0 1 2 3 4 5 6 7 8 9
- 图6.5 完全二叉树的顺序存储示意图

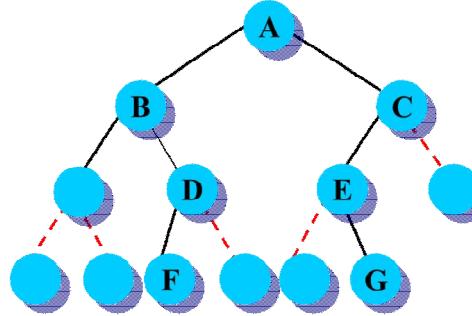


Implementation of Binary Trees

- 对于一般的二叉树，如果仍按从上至下和从左到右的顺序将树中的结点顺序存储在一维数组中，则数组元素下标之间的关系不能够反映二叉树中结点之间的逻辑关系，只有增添一些并不存在的空结点，使之成为一棵完全二叉树的形式，然后再用一维数组顺序存储。
- 这种存储对于需增加许多空结点才能将一棵二叉树改造成为一棵完全二叉树的存储时，会造成空间的大量浪费，不宜用顺序存储结构。
- 最坏的情况是右单支树，一棵深度为 k 的右单支树，只有 k 个结点，却需分配 $2^k - 1$ 个存储单元



(a) 一棵二叉树

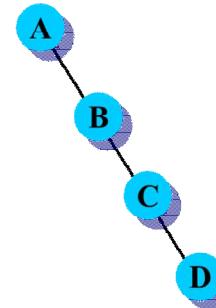


(b) 改造后的完全二叉树

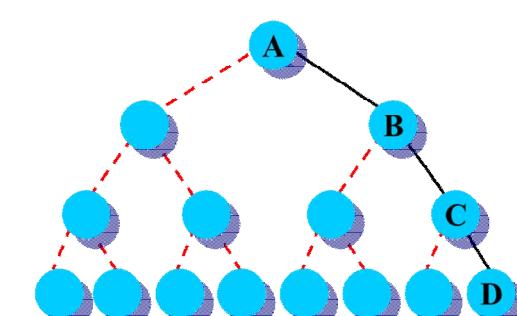
A		B		C		Λ		D		E		Λ		Λ		Λ		F		Λ		Λ		G
---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---

(c) 改造后完全二叉树顺序存储状态

一般二叉树及其顺序存储示意图



(a) 一棵右单支二叉树



(b) 改造后的右单支树对应的完全二叉树

A		Λ		B		Λ		Λ		Λ		C		Λ		Λ		Λ		Λ		Λ		D
---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---

(c) 单支树改造后完全二叉树的顺序存储状态

右单支二叉树及其顺序存储示意图

Implementation of Binary Trees

- 二叉树的顺序存储表示可描述为：

```
#define MaxNode 100           // 二叉树的最大结点数  
typedef ElemenType SqBiTree[MaxNode]  
                                // 0号单元存放根结点  
SqBiTree bt;
```

- 即将bt定义为含有**MaxNode**个**ElemenType**类型元素的一维数组。

Implementation of Binary Trees

2. 链式存储结构

- 二叉树的链式存储结构是用链表来表示一棵二叉树，即用链来指示着元素的逻辑关系。通常有下面两种形式。

(1) 二叉链表存储

- 链表中每个结点由三个域组成，除了数据域外，还有两个指针域，分别用来给出该结点左孩子和右孩子所在的链结点的存储地址。



- data域存放结点的数据；lchild与rchild分别存放指向左孩子和右孩子的指针，当左孩子或右孩子不存在时，相应指针域值为空(用^或NULL表示)。

Implementation of Binary Trees

二叉链表的定义

Binary_tree class:

```
template <class Entry>
class Binary_tree {
public:
    // Add methods here.

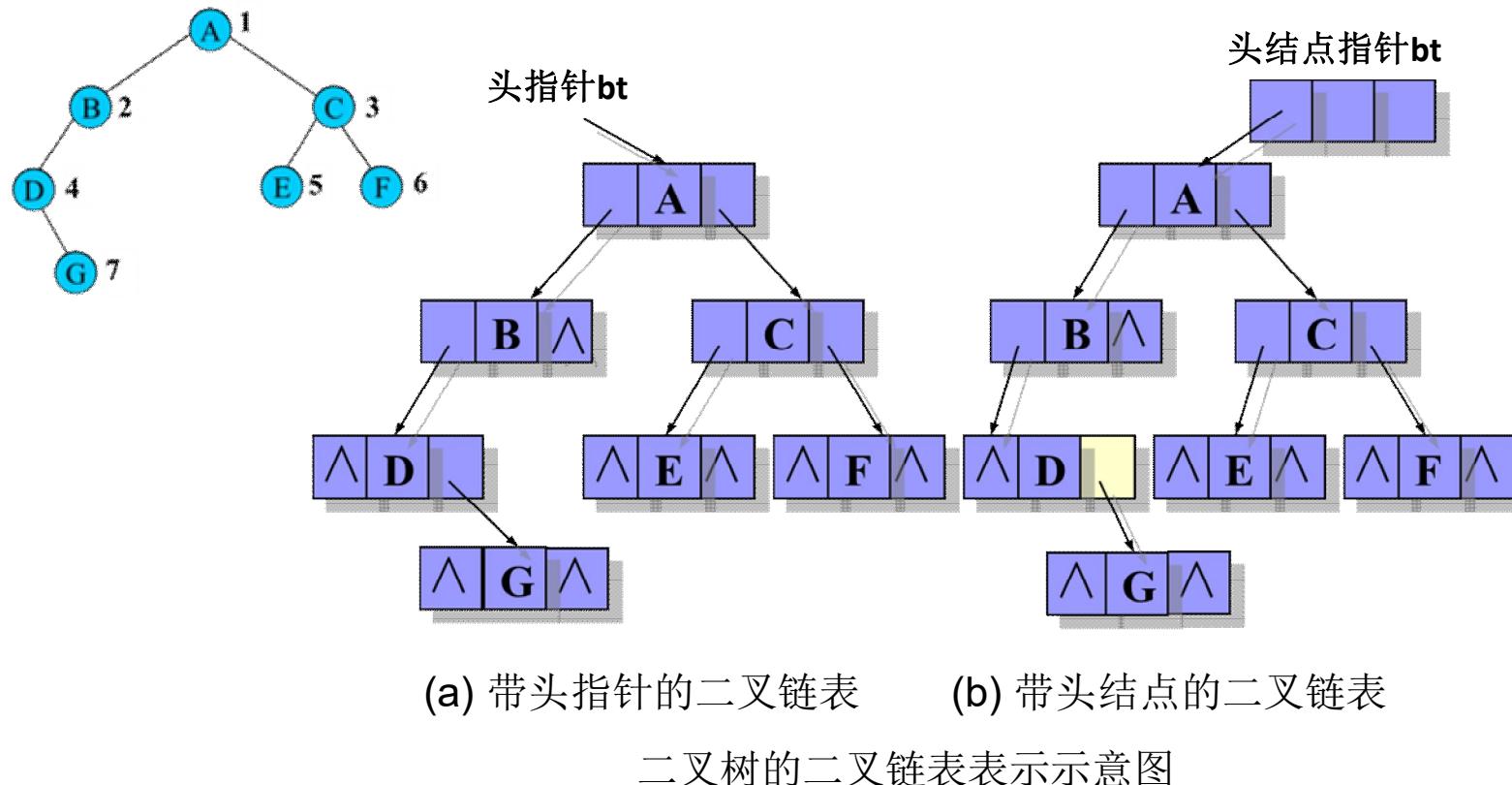
protected:
    // Add auxiliary function prototypes here.
    Binary_node<Entry> *root;
};
```

Binary_node class:

```
template <class Entry>
struct Binary_node {
    // data members:
    Entry data;
    Binary_node<Entry> *left;
    Binary_node<Entry> *right;
    // constructors:
    Binary_node( );
    Binary_node(const Entry &x);
};
```

Implementation of Binary Trees

- 下图给出了一棵二叉树的二叉链表示。
- 二叉链表也可以带头结点的方式存放，如图(b)所示。



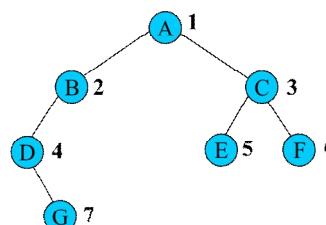
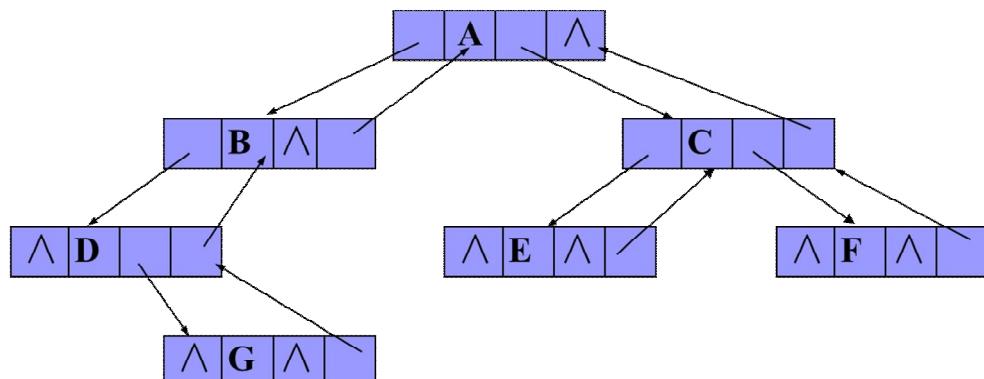
Implementation of Binary Trees

(2) 三叉链表存储

- 每个结点由四个域组成，具体结构为：



- 其中，data、lchild以及rchild三个域的意义同二叉链表结构；parent域为指向该结点双亲结点的指针。这种存储结构既便于查找孩子结点，又便于查找双亲结点；但是，相对于二叉链表存储结构而言，它增加了空间开销。
- 下图是一棵二叉树的三叉链表示。



- 尽管在二叉链表中无法由结点直接找到其双亲，但由于二叉链表结构灵活，操作方便，对于一般情况的二叉树，甚至比顺序存储结构还节省空间。因此，二叉链表是最常用的二叉树存储方式。

Operations of Binary Trees

二叉树的基本操作

二叉树的基本操作通常有以下几种：

- (1) `Initiate(bt)`建立一棵空二叉树。
- (2) `Create(x,lbt,rbt)`生成一棵以x为根结点的数据域信息，以二叉树lbt和rbt为左子树和右子树的二叉树。
- (3) `InsertL(bt,x,parent)`将数据域信息为x的结点插入到二叉树bt中作为结点parent的左孩子结点。如果结点parent原来有左孩子结点，则将结点parent原来的左孩子结点作为结点x的左孩子结点。

Operations of Binary Trees

- (4) InsertR(bt,x,parent)将数据域信息为x的结点插入到二叉树bt中作为结点parent的右孩子结点。如果结点parent原来有右孩子结点，则将结点parent原来的右孩子结点作为结点x的右孩子结点。
- (5) DeleteL(bt,parent)在二叉树bt中删除结点parent的左子树。
- (6) DeleteR(bt,parent)在二叉树bt中删除结点parent的右子树。
- (7) Search(bt,x)在二叉树bt中查找数据元素x。
- (8) Traverse(bt)按某种方式遍历二叉树bt的全部结点。

Operations of Binary Trees

算法的实现

- 算法的实现依赖于具体的存储结构，当二叉树采用不同的存储结构时，上述各种操作的实现算法是不同的。下面讨论基于二叉链表存储结构的上述操作的实现算法。

二叉树的类定义

```
template <class Entry>
class Binary_tree {
public:
    Binary_tree( );
    bool empty( ) const;
    void preorder(void (*visit)(Entry &));
    void inorder(void (*visit)(Entry &));
    void postorder(void (*visit)(Entry &));
    int size( ) const;
    void clear( );
    int height( ) const;
    void insert(const Entry &);

    Binary_tree (const Binary_tree<Entry> &original);
    Binary_tree & operator = (const Binary_tree<Entry> &original);
    ~Binary_tree( );

protected:
    // Add auxiliary function prototypes here.
    Binary_node<Entry> *root;
};
```

Operations of Binary Trees

二叉树操作的实现

Constructor:

```
template <class Entry>
Binary_tree<Entry> :: Binary_tree()
/* Post: An empty binary tree has been created. */
{
    root = NULL;
}
```

Empty:

```
template <class Entry>
bool Binary_tree<Entry> :: empty( ) const
/* Post: A result of true is returned if the binary tree is empty.
   Otherwise, false is returned. */
{
    return root == NULL;
}
```

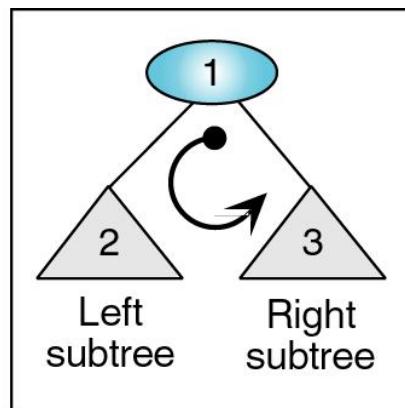
Traversal of Binary Trees

• 遍历二叉树

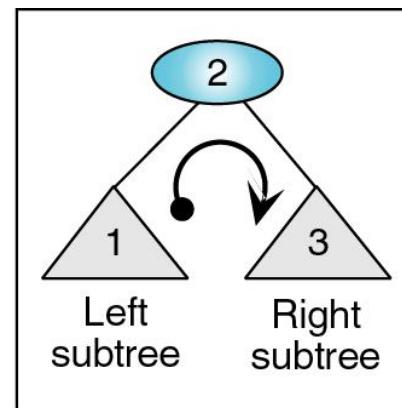
- 按照某种顺序访问二叉树中的每个结点，使每个结点被访问一次且仅被访问一次。
- 遍历是二叉树中经常要用到的一种操作。因为在实际应用问题中，常常需要按一定顺序对二叉树中的每个结点逐个进行访问，查找具有某一特点的结点，然后对这些满足条件的结点进行处理。
- 通过一次完整的遍历，可使二叉树中结点信息由非线性排列变为某种意义上的线性序列。也就是说，遍历操作使非线性结构线性化。

Traversal of Binary Trees

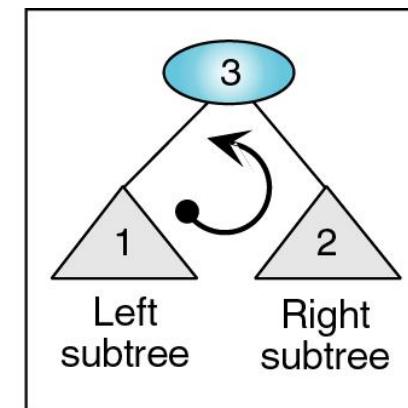
- 由二叉树的定义可知，一棵由根结点、根结点的左子树和根结点的右子树三部分组成。因此，只要依次遍历这三部分，就可以遍历整个二叉树。
- 若以D、L、R分别表示访问根结点、遍历根结点的左子树、遍历根结点的右子树，则二叉树的遍历方式有六种：DLR、LDR、LRD、DRL、RDL和RLD。如果限定先左后右，则只有前三种方式，即
- DLR** (称为先序遍历)、**LDR** (称为中序遍历)、**LRD** (称为后序遍历)。



(a) Preorder traversal



(b) Inorder traversal



(c) Postorder traversal

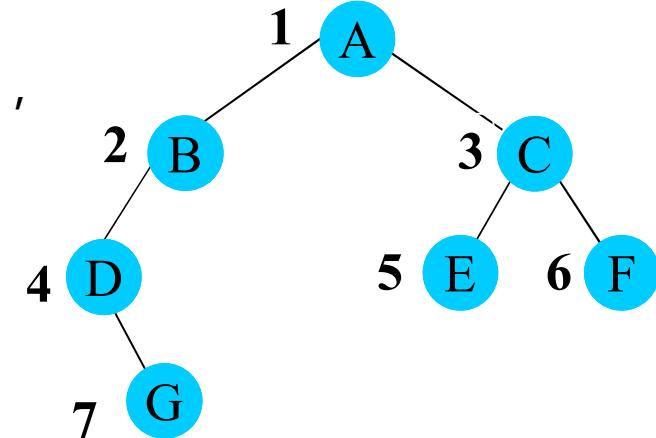
Traversal of Binary Trees

1. 先序遍历(DLR)

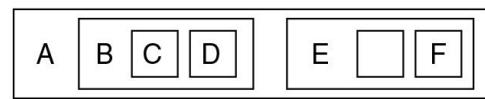
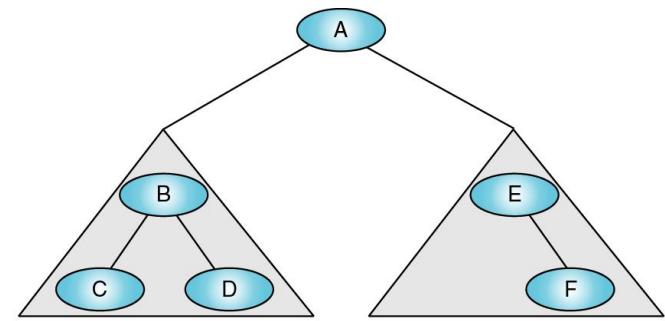
- 递归过程为：若二叉树为空，遍历结束。否则，
 - (1)访问根结点；
 - (2)先序遍历根结点的左子树；
 - (3)先序遍历根结点的右子树。
- 先序遍历二叉树的递归算法：

```
void PreOrder(BiTTree bt)           // 先序遍历二叉树bt
{
    if (bt==NULL) return; // 递归调用的结束条件
    Visite(bt->data);           // 访问结点的数据域
    PreOrder(bt->lchild); // 先序递归遍历bt的左子树
    PreOrder(bt->rchild); // 先序递归遍历bt的右子树
}
```

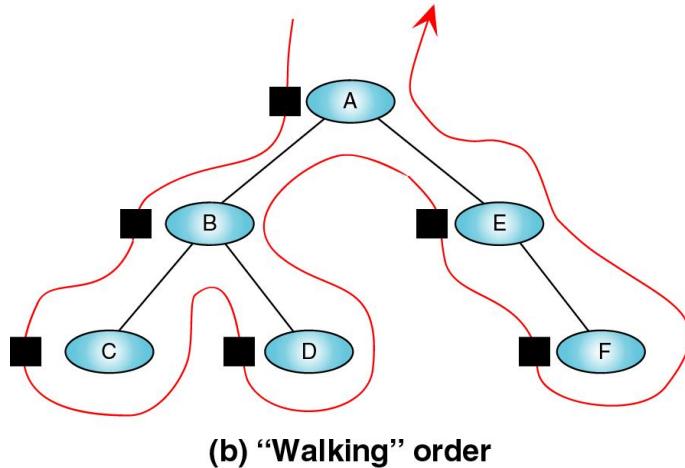
对于上图所示的二叉树，按先序遍历所得到的结点序列为：**ABDGCEF**



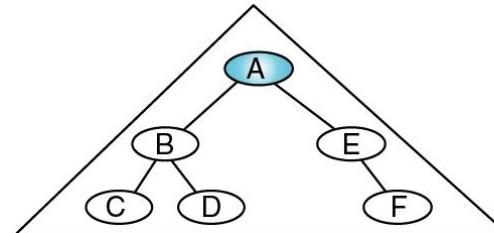
Traversal of Binary Trees



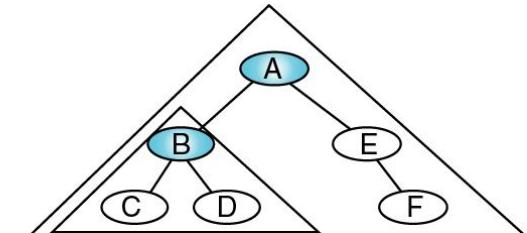
(a) Processing order



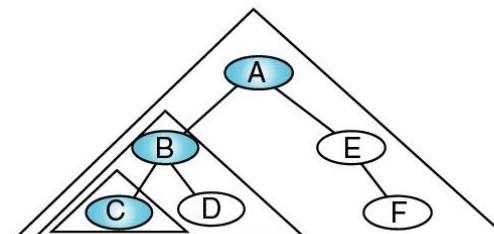
(b) "Walking" order



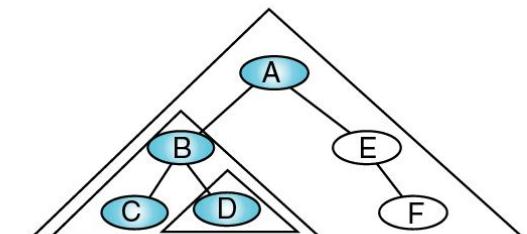
(a) Process tree A



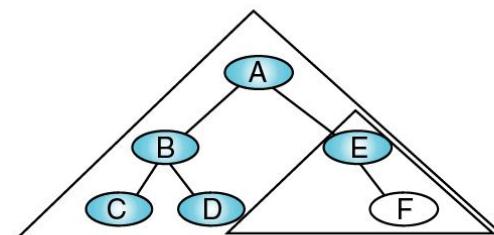
(b) Process tree B



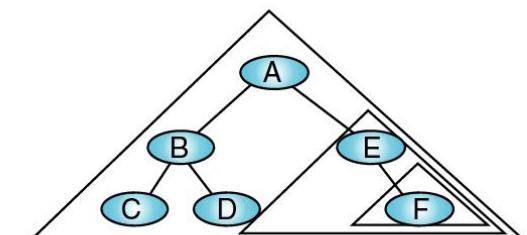
(c) Process tree C



(d) Process tree D



(e) Process tree E



(f) Process tree F

Traversal of Binary Trees

2. 中序遍历(LDR)

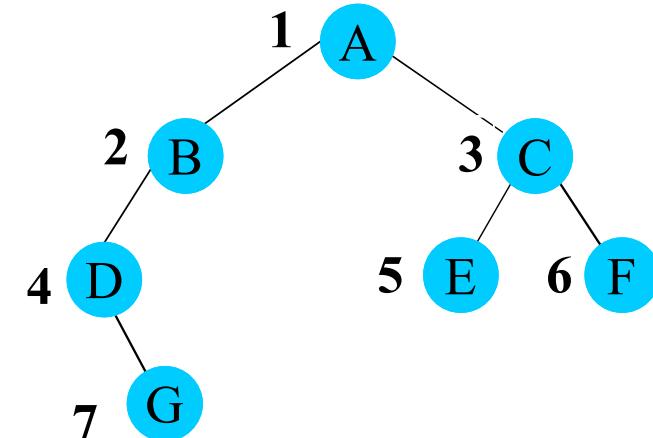
- 递归过程为：若二叉树为空，遍历结束。否则，

- (1)中序遍历根结点的左子树；
- (2)访问根结点；
- (3)中序遍历根结点的右子树。

- 中序遍历二叉树的递归算法：

```
void InOrder(BiTree bt)          // 中序遍历二叉树bt
{
    if (bt==NULL) return;        // 递归调用的结束条件
    InOrder(bt->lchild);       // 中序递归遍历bt的左子树
    Visite(bt->data);          // 访问结点的数据域
    InOrder(bt->rchild);       // 中序递归遍历bt的右子树
}
```

对于上图所示的二叉树，按中序遍历所得到的结点序列为：**DGBAEFC**



Traversal of Binary Trees

3. 后序遍历(LRD)

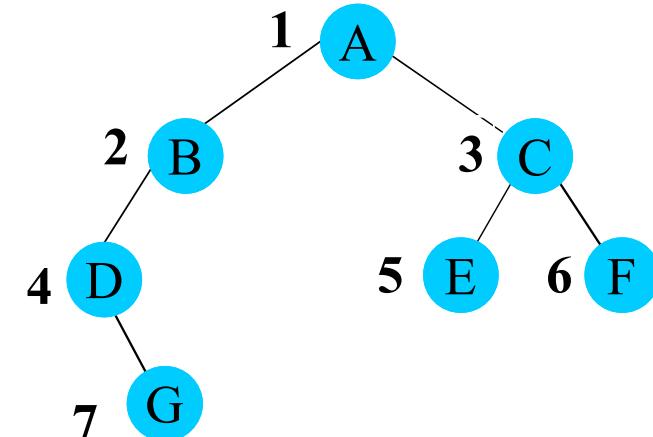
- 递归过程为：若二叉树为空，遍历结束。否则，

- (1)后序遍历根结点的左子树；
- (2)后序遍历根结点的右子树。
- (3)访问根结点；

- 后序遍历二叉树的递归算法：

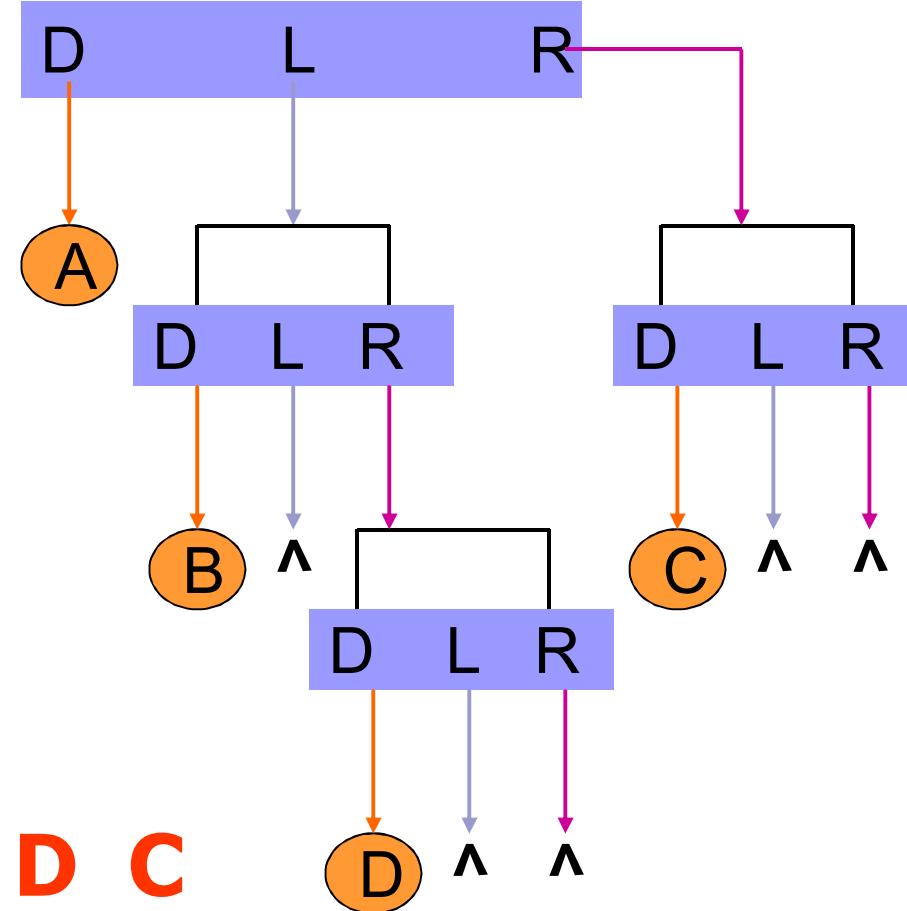
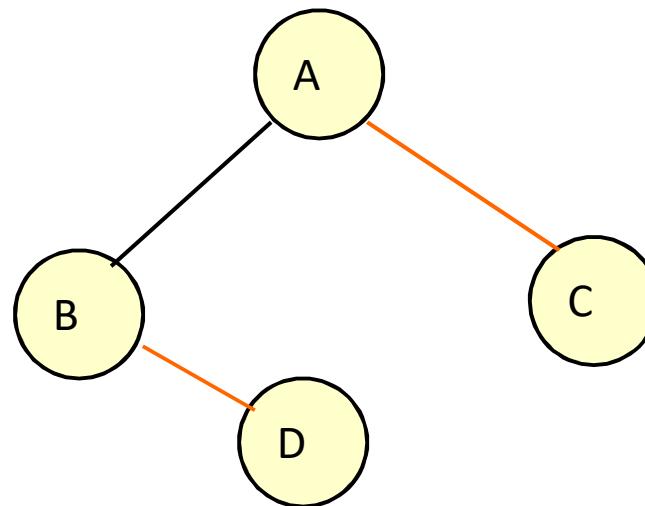
```
void PostOrder(BiTree bt)          // 后序遍历二叉树bt
{
    if (bt==NULL) return;           // 递归调用的结束条件
    PostOrder(bt->lchild);        // 后序递归遍历bt的左子树
    PostOrder(bt->rchild);        // 后序递归遍历bt的右子树
    Visite(bt->data);            // 访问结点的数据域
}
```

对于上图所示的二叉树，按先序遍历所得到的结点序列为：**GDBEFC**A



Traversal of Binary Trees

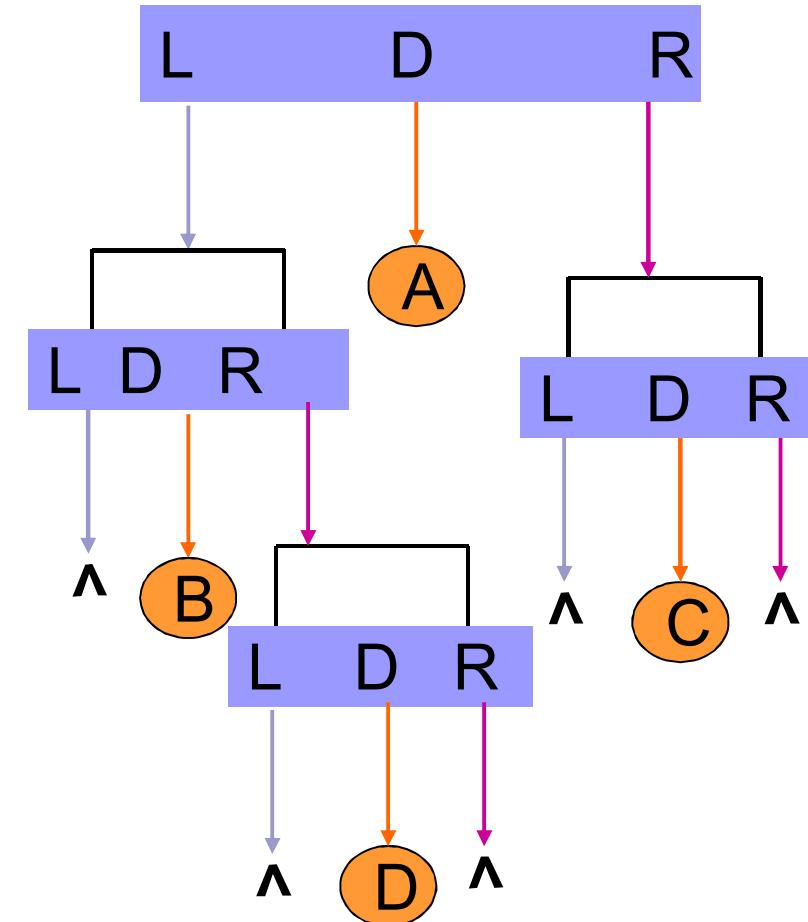
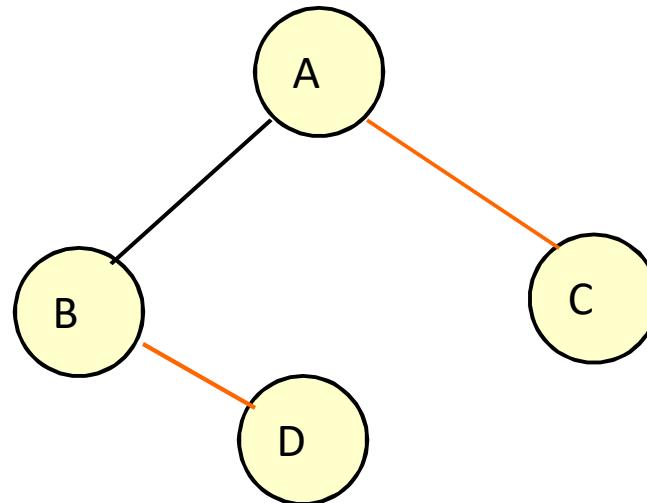
先序遍历：



先序遍历序列： **A B D C**

Traversal of Binary Trees

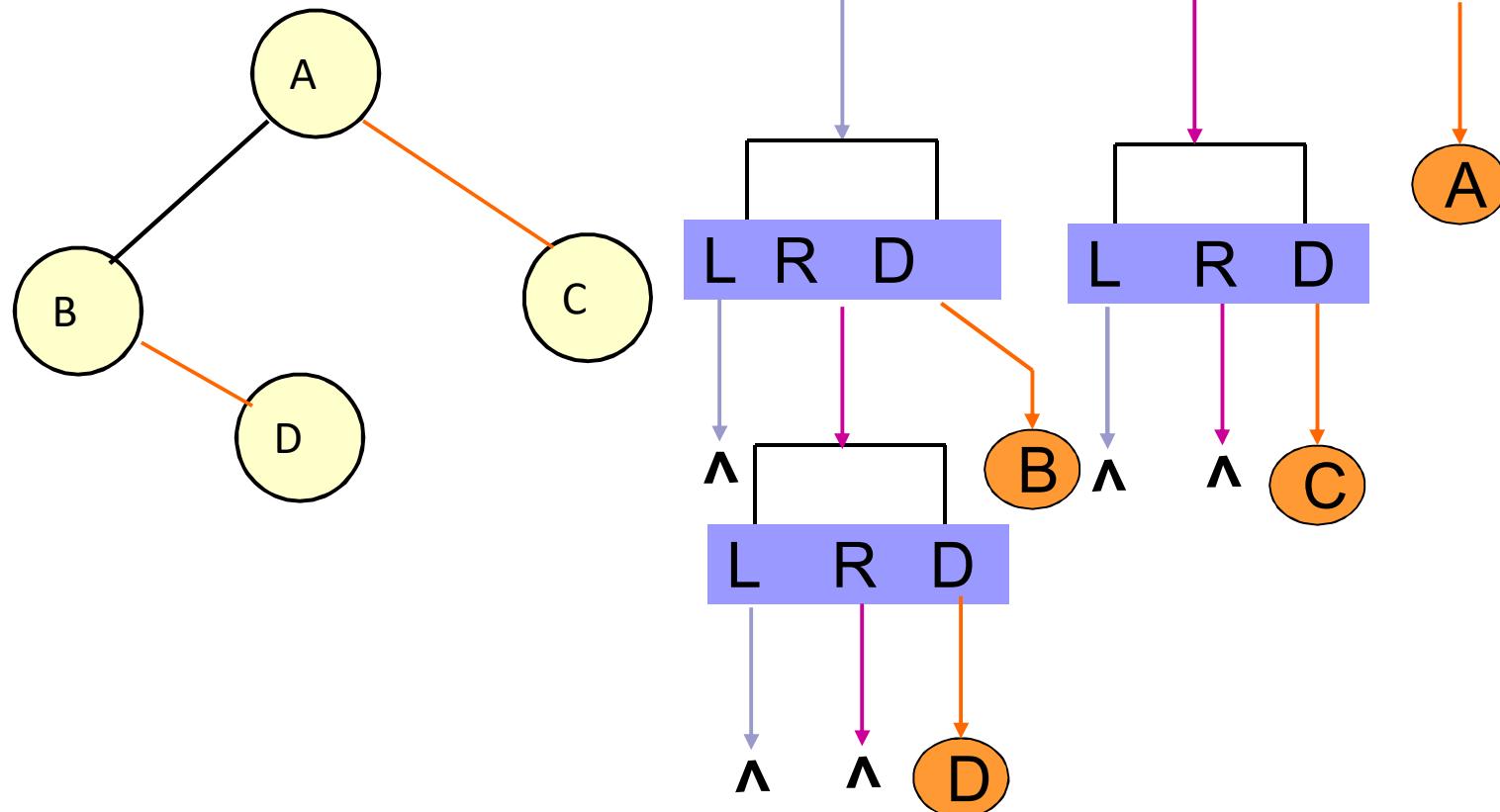
中序遍历:



中序遍历序列: **B D A C**

Traversal of Binary Trees

后序遍历:



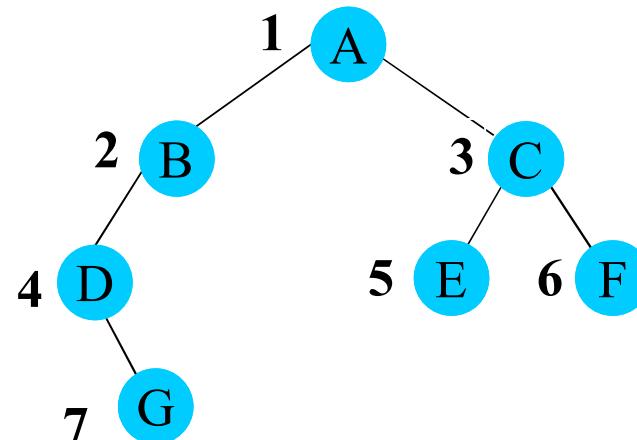
后序遍历序列: **D B C A**

Traversal of Binary Trees

层次遍历

- 所谓二叉树的层次遍历，是指从二叉树的第一层(根结点)开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问。

对于图6.3(b)所示的二叉树，按层次遍历所得到的结果序列为：**ABCDEFG**



Traversal of Binary Trees

- 由层次遍历的定义可以推知，在进行层次遍历时，对一层结点访问完后，再按照它们的访问次序对各个结点的左孩子和右孩子顺序访问，这样一层一层进行，先遇到的结点先访问，这与队列的操作原则比较吻合。因此，在进行层次遍历时，可设置一个队列结构，遍历从二叉树的根结点开始，首先将根结点指针入队列，然后从对头取出一个元素，每取一个元素，执行下面两个操作：
 - (1) 访问该元素所指结点；
 - (2) 若该元素所指结点的左、右孩子结点非空，则将该元素所指结点的左孩子指针和右孩子指针顺序入队。
- 此过程不断进行，当队列为空时，二叉树的层次遍历结束。

Traversal of Binary Trees

- 在下面的层次遍历算法中，二叉树以二叉链表存放，一维数组Queue[MaxNode]用以实现队列，变量front和rear分别表示当前对首元素和队尾元素在数组中的位置。

```
void LevelOrder(BiTree bt) // 层次遍历二叉树bt
{
    BiTree Queue[MaxNode];
    int front,rear;
    if (bt==NULL) return;
    front=-1; rear=0; queue[rear]=bt;
    while(front!=rear)
    {
        front++;
        Visite(queue[front]->data);      // 访问队首结点的数据域
        if (queue[front]->lchild!=NULL) // 将队首结点的左孩子结点入队列
        {
            rear++;
            queue[rear]=queue[front]->lchild;
        }
        if (queue[front]->rchild!=NULL) // 将队首结点的右孩子结点入队列
        {
            rear++;
            queue[rear]=queue[front]->rchild;
        }
    }
}
```

Case

【例】右下图(1)所示的二叉树表达式

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

- 若先序遍历此二叉树，按访问结点的先后次序将结点排列起来，其先序序列为：

$$-+a\times b-cd/ef$$

- 按中序遍历，其中序序列为：

$$a+b\times c-d-e/f$$

- 按后序遍历，其后序序列为：

$$abcd-\times +ef/-$$

- 人喜欢中缀形式的算术表达式，对于计算机，使用后缀易于求值。

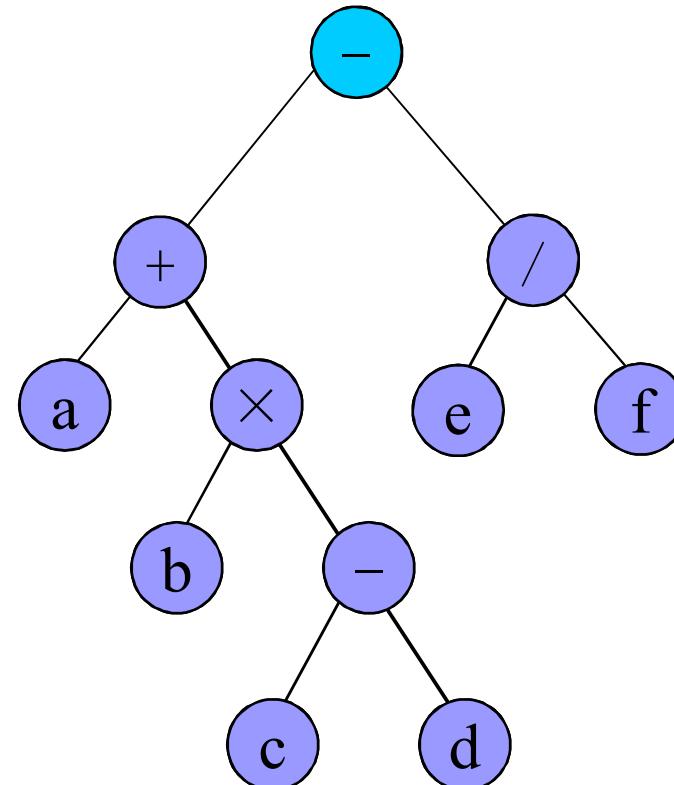


图 (1)

谢谢！

