数据结构

北京邮电大学 网络空间安全学院

武 斌

上章内容

上一章（数组和广义表）内容：

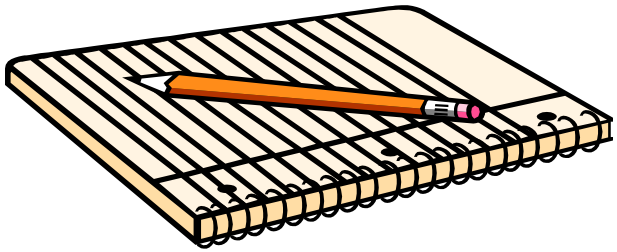
- 理解数组类型的特点及其在高级编程语言中的存储表示和实现方法
- 掌握数组在"以行为主"的存储表示中的地址计算方法
- 掌握特殊矩阵的存储压缩表示方法
- 理解稀疏矩阵的两类存储压缩方法的特点及其适用范围
- 掌握广义表的结构特点及其存储表示方法

本次课程学习目标

学习完本次课程（树和二叉树(上)），您应该能够：

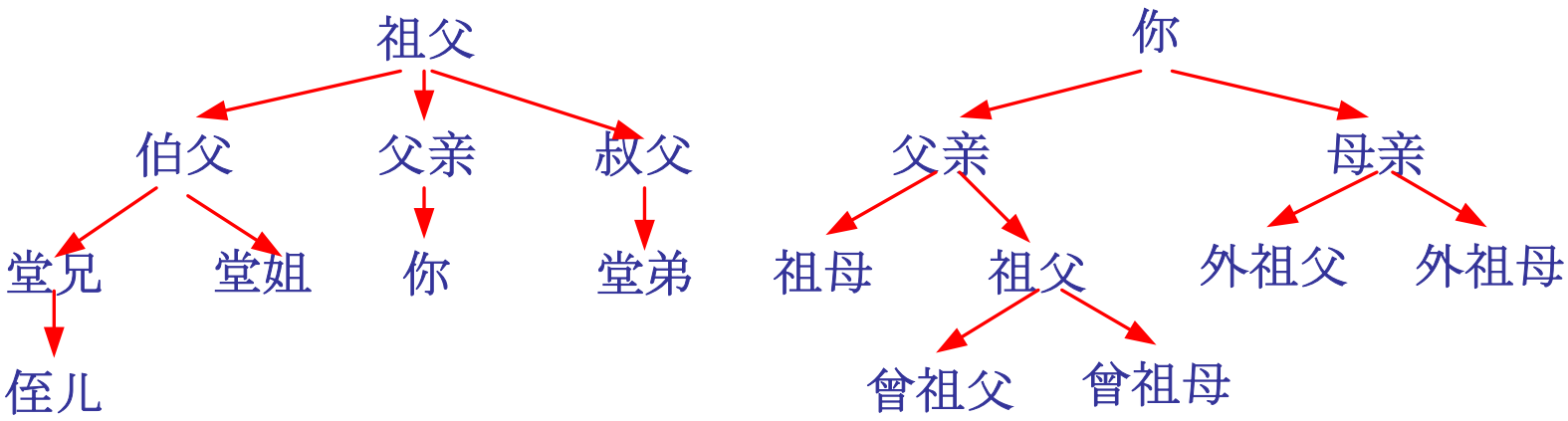
- 领会树和二叉树的类型定义，理解树和二叉树的结构差别
- 熟记二叉树的主要特性，并掌握它们的证明方法
- 熟练掌握二叉树和树的各种存储结构及其建立的算法
- 学会编写实现树的各种操作的算法

本章课程内容（第六章 树和二叉树(上)）



- 6.1 树的定义和基本术语
- 6.2 二叉树
- 6.3 遍历二叉树和线索二叉树
- 6.4 树和森林
- 6.5 赫夫曼树及其应用

家族谱系图



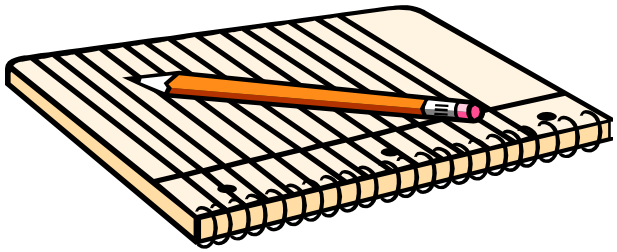
- 上列家族谱系图可用如下关系表示：

<祖父，伯父>，<祖父，父亲>，<祖父，叔父>，<伯父，堂兄>，<伯父，堂姐>，<父亲，你>， <叔父，堂弟>，<堂兄，侄儿>

第六章 树和二叉树

- 树形结构是一类重要的非线性结构。树型结构是结点之间有分支，并且具有层次关系的结构，它非常类似于自然界中的树。树结构在客观世界中是大量存在的，例如：
  - 家谱
  - 行政组织机构
- 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。

树的定义和基本术语



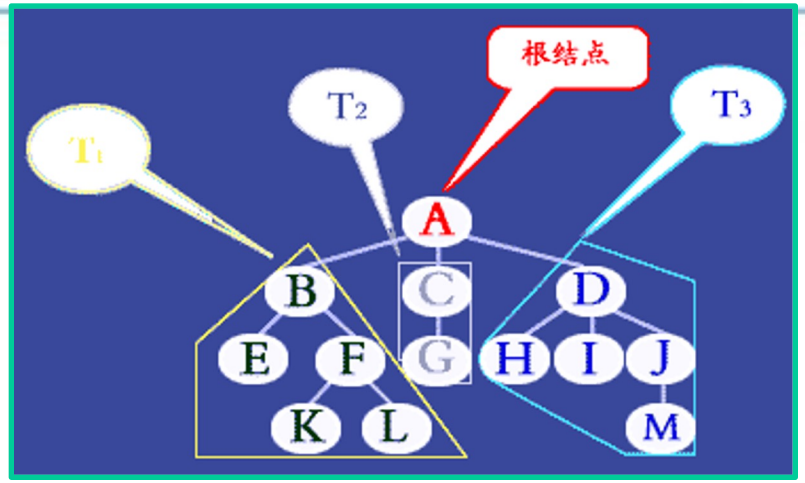
- 6.1 树的定义和基本术语
- 6.2 二叉树
- 6.3 遍历二叉树和线索二叉树
- 6.4 树和森林
- 6.5 赫夫曼树及其应用

树的定义和基本术语

- 定义：树(Tree)是n(n>=0)个结点的有限集T，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)其余的结点可分为m(m>=0)个互不相交的子集$T_1,T_2,T_3...T_m$，其中每个子集又是一棵树，并称其为子树(Subtree)。
- T为空时称为空树。

树的定义和基本术语

●例如：

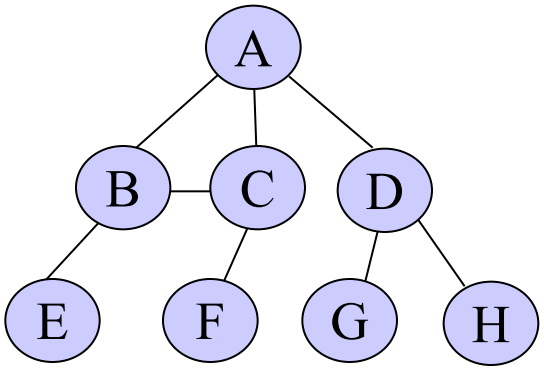


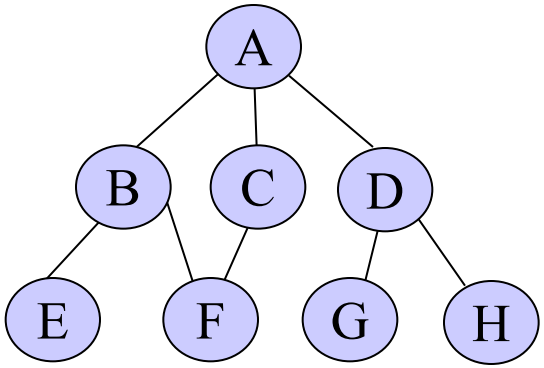
- **A为“根”**，其余12个数据元素分为**3个互不相交的子集 T1={B,E,F,K,L}、T2={C,G}和T3={D,H,I,J,M}**。
- 每个子集都是一棵树，称为**A的子树**，它们的根结点都是**A的后继**。
  - 这是一个递归的定义，如在子树T1中，**B是根**，其余元素分为**2个互不相交的子集T11={E}和 T12={F,K,L}**，每个子集构成一棵**B的子树**，子树中的根结点是**B的后继**，依次类推。
- 13个数据元素之间存在下列关系：

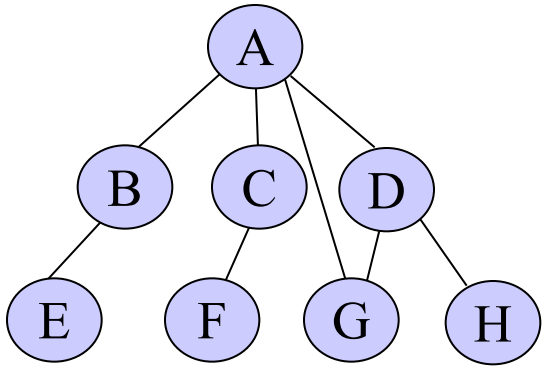
<A,B>,<A,C>,<A,D>,<B,E>,<B,F>,<C,G>,<D,H>,<D,I>,<D,J>,<F,K>,<F,L>,<J,M>}。

树的定义和基本术语

- 数与非树？







- 子树是不相交的
- 除根节点外，每个节点有且只有一个父结点
- 一棵N个结点的树有N-1条边

树的定义和基本术语

- 树是一种层次分明的结构，约定**根的层次为1**，其余元素层次的定义为：**若根的层次为L，则子树根的层次为 L+1**。
- 子树之间可能存在两种情况，如果子树之间映射客观存在次序关系，则为“**有序树**”，否则为 “**无序树**”。
  - **分支**：称根和子树根之间的连线为“**分支**”；
  - **结点**：数据元素和所有指向子树根的分支构成树中一个“**结点**”；
  - **结点的度**：其分支的个数定义为“**结点的度**”，如结点B的度为2，D的度为3。
  - **树的度**：树中所有结点度的**最大值**定义为“**树的度**”。
  - **叶子、分支结点**：称**度为零的结点**为"**叶子**"或"**终端结点**"，如结点 K,L,G 等，反之所有**度不为零的结点**被称作"**分支结点**"，如结点 F,J 等。

树的定义和基本术语

- 树的深度定义为树中叶子结点所在最大层次数。
- 从树的定义可知，“根”即为树中没有前驱的结点。称根结点为子树根的“双亲”。
- 称子树根为根结点的“孩子”。“最左孩子”指的是在存储结构中存放的第一棵子树的根。
- 根的所有子树根互为“兄弟”。“右兄弟”指存储结构中确定的有相同双亲的下一棵子树的根。
- 结点的“祖先”指从根结点到该结点所经分支上的所有结点。
- 以某结点为根的子树中的任一结点都称为该结点的“子孙”。

树的定义和基本术语

- 树的抽象数据类型的定义如下：

ADT Tree {

数据对象：D是具有相同特性的数据元素的集合。

数据关系：

若 D 为空集，则称为空树；

若 D 中仅含一个数据元素，则关系R为空集；

否则 R={H}，

(1) 在D中存在**唯一的称为根的数据元素 root**，它在关系H下无前驱；

(2) 当n>1时，**其余数据元素可分为 m(m>0) 个互不相交的(非空)有限集 T1,T2,…,Tm**，其中每一个子集本身又是一棵符合本定义的树，称为**根 root 的子树**，每一棵子树的根 x_i 都是根 root 的后继，即 $<root,x_i>\in H, i=1,2,\ldots,m$。

树的定义和基本术语

- 基本操作：

  {结构初始化}

  InitTree(&T);

  操作结果：构造空树 T。

  CreateTree(&T,definition);

  初始条件：definition 给出树T的定义。

  操作结果：按 definition 构造树 T。

  {销毁结构}

  DestroyTree(&T);

  初始条件：树 T 存在。

  操作结果：销毁树 T。

  {引用型操作}

  TreeEmpty(T);

  初始条件：树 T 存在。

  操作结果：若 T 为空树，则返回 TRUE，否则返回 FALSE。

树的定义和基本术语

{引用型操作}

TreeDepth(T);

初始条件：树T存在。

操作结果：返回T的深度。//树的深度定义为树中叶子结点所在最大层次数

Root(T);

初始条件：树 T 存在。

操作结果：返回 T 的根。

Value(T, cur_e);

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：返回 cur_e 的值。

Parent(T, cur_e);

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：若 cur_e 是T的非根结点，则返回它的双亲，否则返回“空”。

LeftChild(T, cur_e);

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：若 cur_e 是T的非叶子结点，则返回它的最左孩子，否则返回“空”。

树的定义和基本术语

{引用型操作}

RightSibling(T, cur_e);

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：若 cur_e 有右兄弟，则返回它的右兄弟，否则返回“空”。

TraverseTree(T, visit());

初始条件：树T存在，visit 是对结点操作的应用函数。

操作结果：按某种次序对 T 的每个结点调用函数 visit() 一次且至多一次。一旦 visit() 失败，则操作失败。

{加工型操作}

Assign(T, cur_e, value);

初始条件：树T存在，cur_e 是 T 中某个结点。

操作结果：结点 cur_e 赋值为 value。

ClearTree(&T);

初始条件：树 T 存在。

操作结果：将树 T 清为空树。

树的定义和基本术语

{加工型操作}

InsertChild(&T, &p, i, c);

初始条件：树 T 存在，p 指向T中某个结点，1≤i≤p 所指结点的度＋1，非空树 c 与 T 不相交。

操作结果：插入 c 为 T 中 p 所指结点的第 i 棵子树。

DeleteChild(&T, &p, i);

初始条件：树 T 存在，p 指向 T 中某个结点，1≤i≤p 指结点的度。

操作结果：删除 T 中 p 所指结点的第 i 棵子树。

} ADT Tree

树的定义和基本术语

- 从另一个角度来定义树：
  - 定义森林为 m(m≥0) 棵互不相交的树的集合。则对树中每个结点而言，其子树的集合即为森林。
  - 可定义树是一个二元组

    Tree = (root，F)，其中，root 是数据元素，称作树的根，F 是子树森林，记作F=($T_1,T_2,\ldots,T_m$)，其中 $T_i=(r_i, F_i)$ 称作根 root 的第 i 棵子树，当 m≠0 时，在树根和其子树森林之间存在下列关系：

    $$RF=\{ <root, r_i> \mid i=1,2,\ldots,m, m>0\}$$

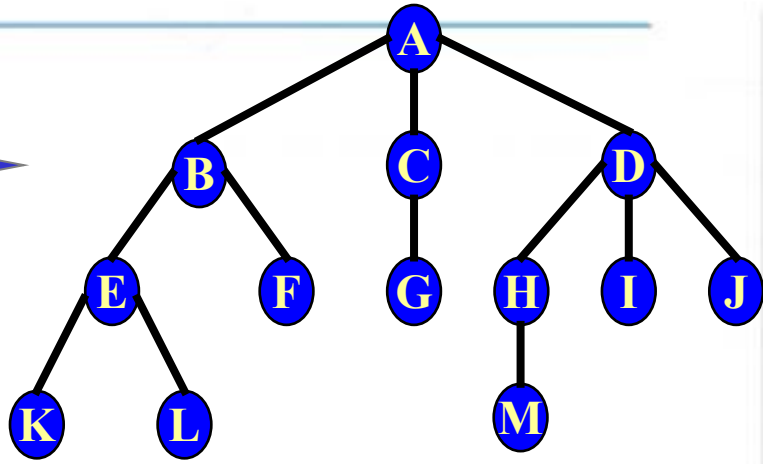
树的其它表示形式

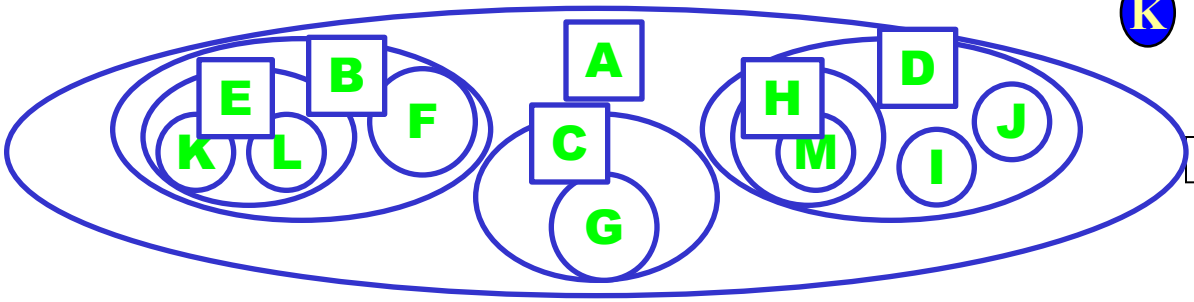
树的几种表示法

- 树形图表示
- 树的其他表示法

① 嵌套集合表示法

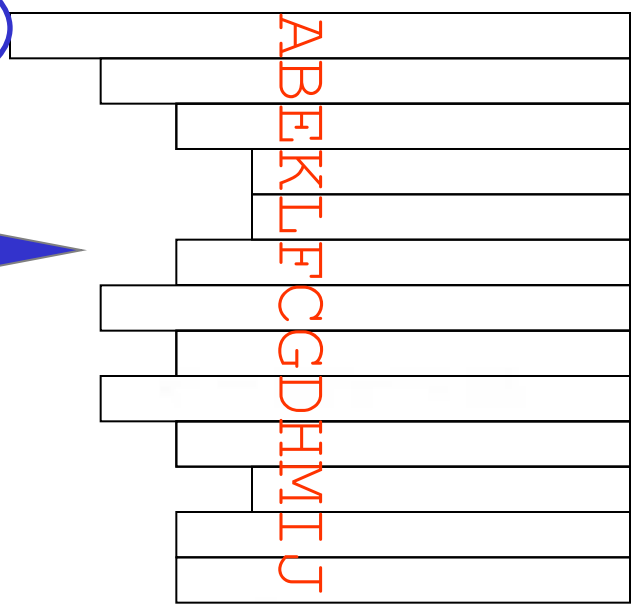
是用集合的包含关系来描述树结构。





② 凹入表表示法

类似于书的目录



③ 广义表表示法

用广义表的形式表示的。

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

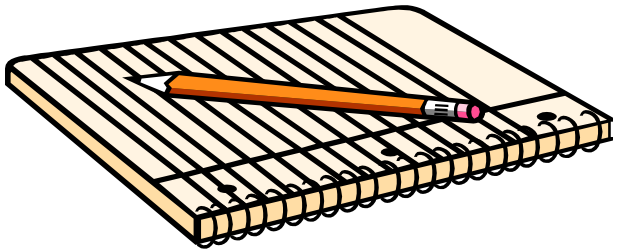
树的定义和基本术语

- 就结构中数据元素之间存在的关系，可将树和线性结构作如下对照：

线性结构	树结构
存在唯一的没有前驱的"首元素"	存在唯一的没有前驱的"根结点"
存在唯一的没有后继的"尾元素"	存在多个没有后继的"叶子"
其余元素均存在唯一的"前驱元素"和唯一的"后继元素"	其余结点均存在唯一的"前驱(双亲)结点"和多个"后继(孩子)结点"

- 可见，由于**线性结构是一个"序列"**，元素之间存在的是**"一对一"**的关系，而**树是一个层次结构**，元素之间存在的是**"一对多"**的关系。

二叉树



- 6.1 树的定义和基本术语
- 6.2 二叉树
- 6.3 遍历二叉树和线索二叉树
- 6.4 树和森林
- 6.6 赫夫曼树及其应用

二叉树

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

  - **1、二叉树(Binary Tree)的定义**

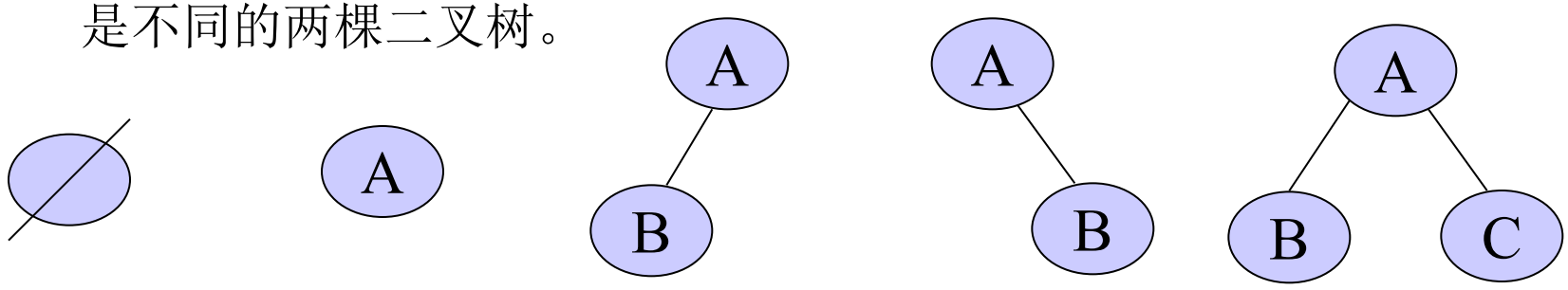
    二叉树是由n(n>=0)个结点的有限集合构成，此集合或者为空集，或者由一个根结点及两棵互不相交的左、右子树组成，并且左右子树都是二叉树。

  - **2、二叉树(Binary Tree)的特点**

    - 每个结点至多只有两棵子树（二叉树中不存在度大于2的结点）
    - 二叉树的子数有左右之分，次序不能任意颠倒

二叉树

- 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差别**。图1列出二叉树的5种基本形态，图1(c)和图1(d)是不同的两棵二叉树。



(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

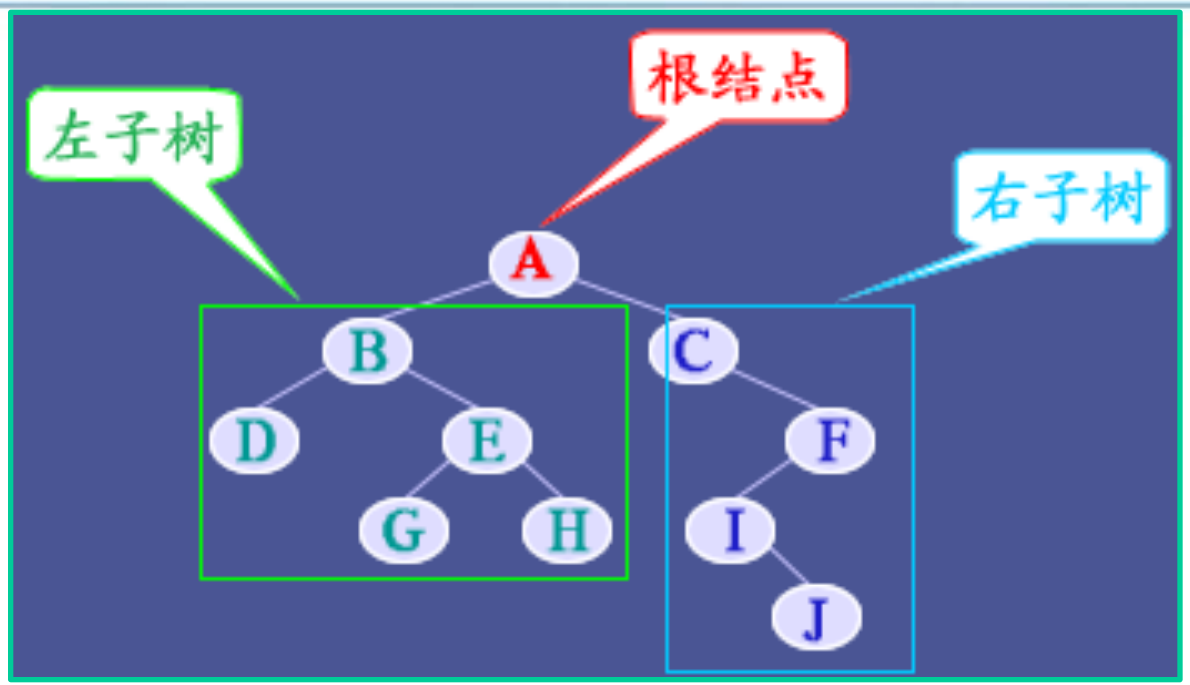
图1 二叉树的5种形式

二叉树

- 二叉树和树是两种不同的树型结构，二叉树不等同于度为2的有序树。

- 例：如果根结点只有一棵子树，那么对树而言，它就是一个"独生子"，无次序可分，但对二叉树而言，必须明确区分它是根的左子树还是根的右子树，两者将构成不同的二叉树。

二叉树



- 如图示为含10个元素的二叉树，其中 A为“根”，无前驱。其余9个元素分为两个互不相交的子集 L={B,D,E,G,H}，R={C,F,I,J}，分别为 A 的左子树和右子树。在右子树R 中，C 为根，其余元素分为两个子集 { } 和 {F,I,J} 分别构成 C 的左子树和右子树。

二叉树

- 二叉树的抽象数据类型定义如下：

ADT BinaryTree {

数据对象：D 是具有相同特性的数据元素的集合。

数据关系：

若 D 为空集，称 BinaryTree 为空二叉树；

若D为非空集合，则关系 R={H}：

(1) 在 D 中存在唯一的称为根的数据元素 root，它在关系 H 下无前驱；

(2) D 中其余元素必可分为两个互不相交的子集 **L** 和 **R**，每一个子集都是一棵符合本定义的二叉树，并分别为 root 的左子树和右子树。如果左子树 L 不空，则必存在一个根结点 x_L ，它是 root 的“左后继”(<root, x_L>∈H)，如果右子树 R 不空，则必存在一个根结点 为 x_R root 的“右后继”(<root, x_R >∈H)。 //参考教材的数据关系

二叉树

➔ 基本操作：

{结构初始化}

InitBiTree(&T);

操作结果：构造空二叉树 T。

CreateBiTree(&T, definition);

初始条件：definition 给出二叉树 T 的定义。

操作结果：按 definition 构造二叉树 T。

{销毁结构}

DestroyBiTree(&T);

初始条件：二叉树 T 存在。

操作结果：销毁二叉树 T。

{引用型操作}

BiTreeEmpty(T);

初始条件：二叉树 T 存在。

操作结果：若T为空二叉树，则返回 TRUE，否则返回 FALSE。

二叉树

{引用型操作}

BiTreeDepth(T);

初始条件：二叉树 T 存在。

操作结果：返回 T 的深度。

Root(T);

初始条件：二叉树 T 存在。

操作结果：返回 T 的根。

Value(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的值。

Parent(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：若e是T的非根结点，则返回它的双亲，否则返回“空”。

LeftChild(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的左孩子。若 e 无左孩子，则返回“空”。

二叉树

{引用型操作}

RightChild(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的右孩子。若 e 无右孩子，则返回“空”。

LeftSibling(T, e);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的左兄弟。若 e 是其双亲的左孩子或无左兄弟，则返回“空”。

RightSibling(T, e);

初始条件：二叉树 T 存在，e 是 T 的结点。

操作结果：返回 e 的右兄弟。若 e 是其双亲的右孩子或无右兄弟，则返回“空”。

PreOrderTraverse(T, visit());

初始条件：二叉树 T 存在，visit 是对结点操作的应用函数。

操作结果：先序遍历 T，对每个结点调用函数 visit 一次且仅一次。一旦 visit() 失败，则操作失败。

二叉树

{引用型操作}

InOrderTraverse(T, vsit());

初始条件：二叉树 T 存在，visit 是对结点操作的应用函数。

操作结果：中序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 visit() 失败，则操作失败。

PostOrderTraverse(T, visit());

初始条件：二叉树T存在，visit 是对结点操作的应用函数。

操作结果：后序遍历 T，对每个结点调用函数 visit 一次且仅一次。一旦 visit() 失败，则操作失败。

LevelOrderTraverse(T, visit());

初始条件：二叉树 T 存在，visit 是对结点操作的应用函数。

操作结果：层序遍历 T，对每个结点调用函数 visit 一次且仅一次.一旦 visit() 失败，则操作失败。

{加工型操作}

ClearBiTree(&T);

初始条件：二叉树 T 存在。

操作结果：将二叉树 T 清为空树。

二叉树

{加工型操作}

Assign(&T, &e, value);

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：结点 e 赋值为 value。

InsertChild(&T, p, LR, c);

初始条件：二叉树 T 存在，p 指向 T 中某个结点，LR 为 0 或 1，非空二叉树 c 与 T 不相交且右子树为空。

操作结果：根据 LR 为 0 或 1，插入 c 为 T 中 p 所指结点的左或右子树。p 所指结点原有左或右子树成为 c 的右子树。

DeleteChild(&T, p, LR);

初始条件：二叉树 T 存在，p 指向 T 中某个结点，LR 为 0 或 1。

操作结果：根据 LR 为 0 或 1，删除 T 中 p 所指结点的左或右子树。

} ADT BinaryTree

二叉树

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>=1)。**

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。

现在假定多所有的j，1≤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。

二叉树

- **性质2：深度为k的二叉树至多有2^k-1个结点（k>=1).**

深度为k的二叉树的最大的结点时为二叉树中每层上的最大结点数之和，由性质1得到每层上的最大结点数：

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

二叉树

- **性质3： 对任何一棵二叉树，如果其终端结点数为n_0，度为2的结点数为n_2，则$n_0=n_2+1$。**

设二叉树中度为1的结点数为n_1，二叉树中总结点数为N，因为二叉树中所有结点均小于或等于2，所以有：

$$N=n_0+n_1+n_2 \qquad (6\text{-}1)$$

再看二叉树中的分支数，除根结点外，其余结点都有一个进入分支，设B为二叉树中的分支总数，则有：$N=B+1$。

由于这些分支都是由度为1和2的结点射出的，所以有：

$$B=n_1+2\times n_2,$$

$$N=B+1=n_1+2\times n_2+1 \qquad (6\text{-}2)$$

由式（6-1）和（6-2）得到：

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

$$n_0=n_2+1$$

二叉树

- 满二叉树和完全二叉树

一棵深度为k且由2^k-1个结点的二叉树称为满二叉树。图2就是一棵满二叉树，对结点进行了顺序编号。

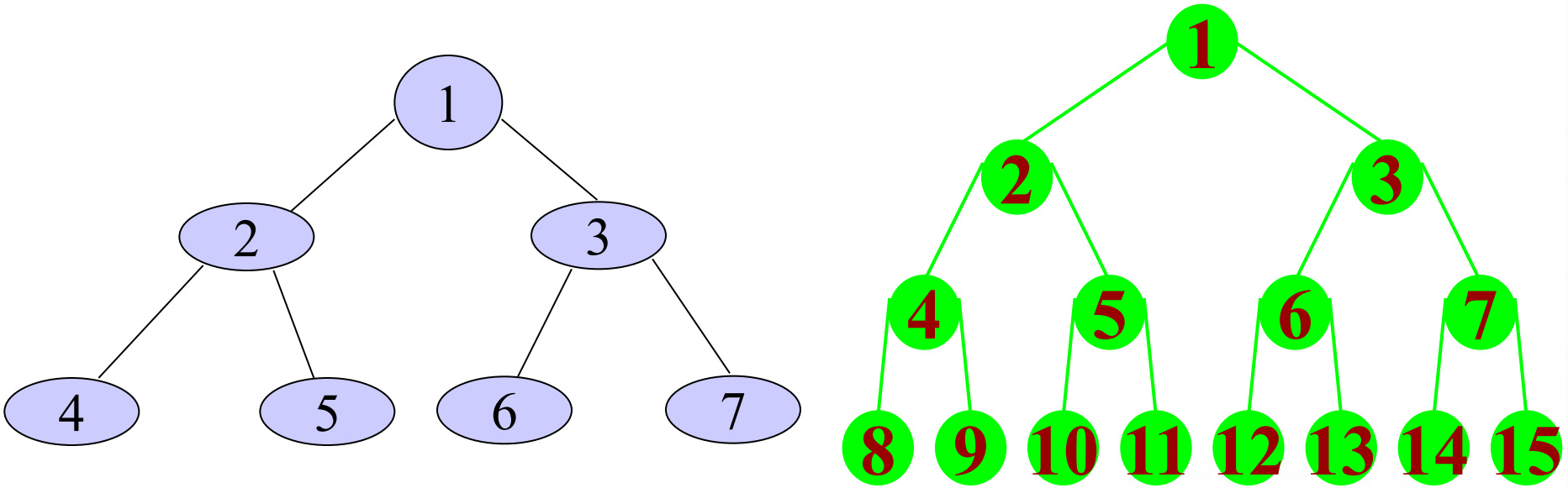
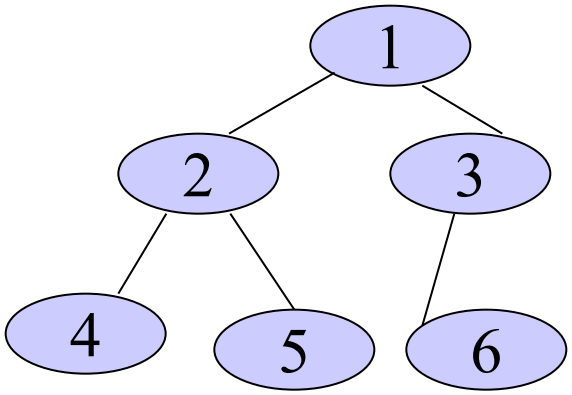


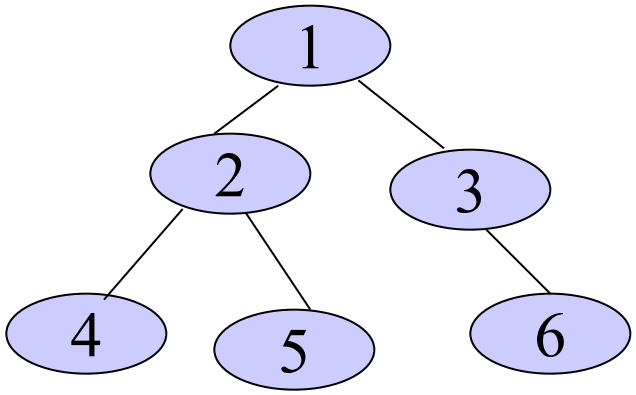
图2 满二叉树

二叉树

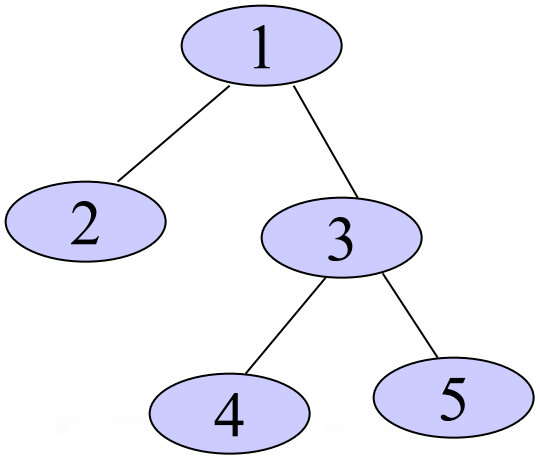
- 如果深度为k、由n个结点的二叉树中，当且仅当其每一个结点能够与深度为k的顺序编号的**满二叉树从1到n标号**的结点相对应，则称这样的二叉树为**完全二叉树**，图(a)是完全二叉树，而图(b)、(c)是两棵非完全二叉树。
- **满二叉树是完全二叉树的特例。**



(a)完全二叉树



(b)非完全二叉树



(c)非完全二叉树

二叉树

- 完全二叉树的特点是：
  - (1) 所有的叶结点只可能在层次最大的两层上出现。
  - (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为L 或 L ＋1。

二叉树

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

  符号[x]表示不大于x的最大整数。

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

  取对数得到：$k-1\leq \log_2 n<k$ ， 即$\log_2 n<k\leq \log_2 n+1$

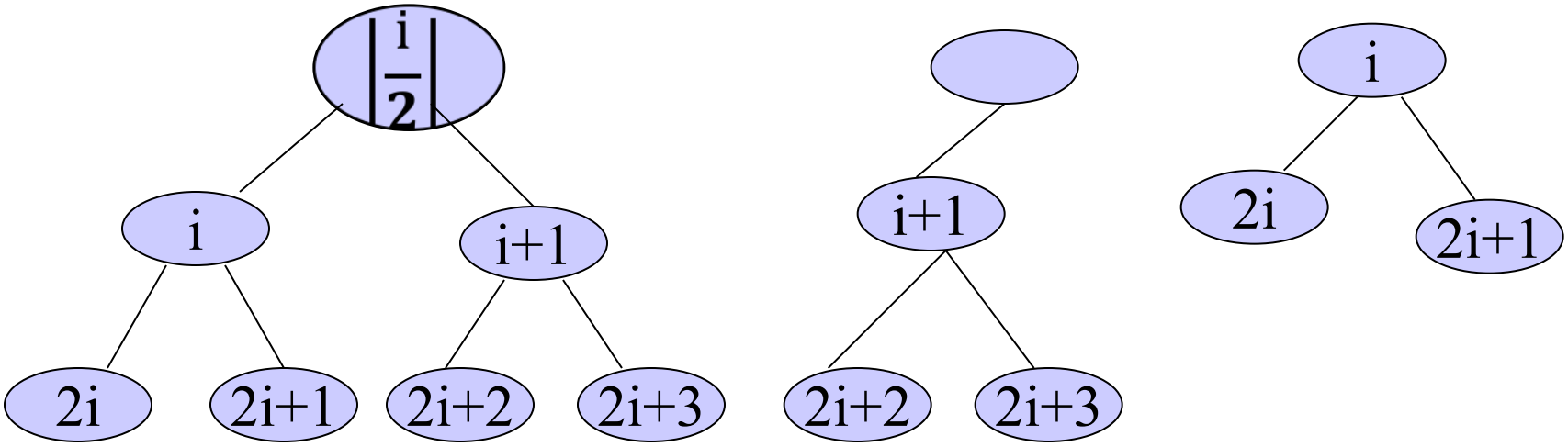
  因为k是整数。所以有：$k=\lfloor log_2 n \rfloor +1$。

二叉树——性质5

- **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$\lfloor log_2 n \rfloor + 1$层，每层从左到右),则**对任一结点i**（1≤i≤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PARENT(i)是结点$\left\lfloor \frac{i}{2} \right\rfloor$。
  - （2）如果2i>n，则结点i为叶子结点，无左孩子；否则，其左孩子LCHILD(i)是结点2i。
  - （3）如果2i＋1>n，则结点i无右孩子；否则，其右孩子RCHILD(i)是结点2i＋1。

二叉树——性质5

- 如图3所示为完全二叉树上结点及其左右孩子结点之间的关系。



(a) i 和i+1结点在同一层　　(b) i 和i+1结点不在同一层

图3 完全二叉树中结点 i 和i+1的左、右孩子

二叉树——性质5

- 在此过程中，可以从(2)和(3)推出(1)，所以先证明(2)和(3)。
  - 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无左孩子。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≤j≤ [log2n])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$2^j=2\times2^{j-1}=2i$。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≤j≤[log2n])层上的某个结点编号为 i ($2^{j-1}\le i\le 2^j-1$)，且2i＋1<n，其左孩子为2i，右孩子为2i＋1，则编号为i＋1的结点是编号为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×(i＋1)：若它有右孩子，则其编号必定为2i＋3＝2×(i＋1)＋1。

二叉树——性质5

- 由此：
  - 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×(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，p＝(i-1)/2=[i/2]。
- 证毕。

二叉树——顺序存储结构

- **3、二叉树的存储结构**

(1) 顺序存储结构

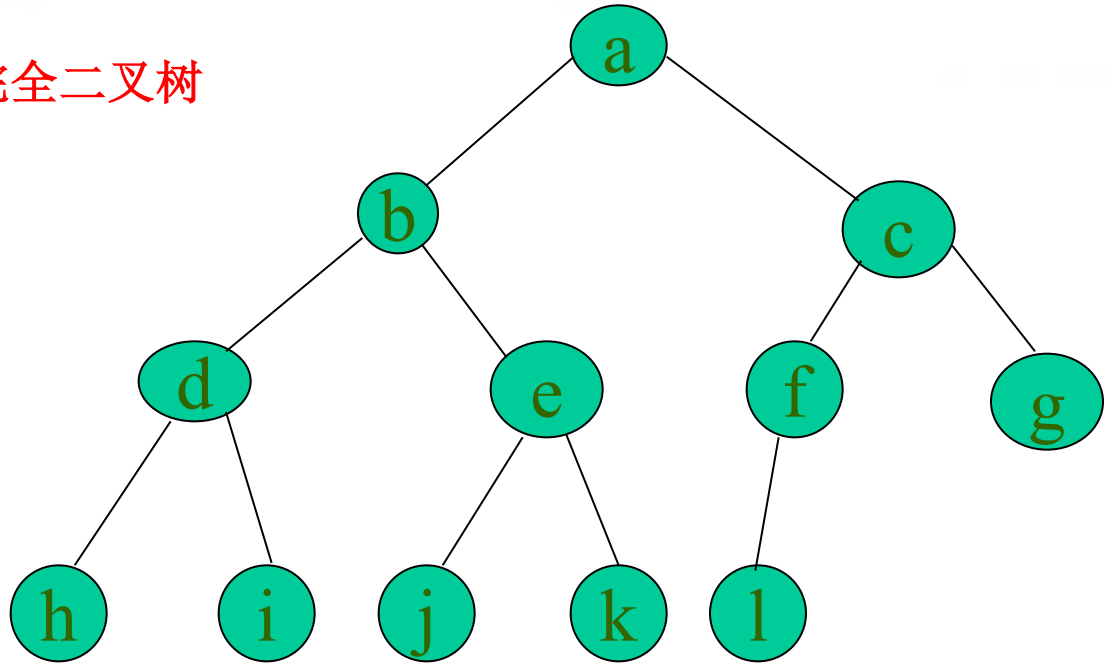
它是用**一组连续的存储单元**存储二叉树的数据元素。因此，必须把二叉树的所有结点安排成为一个恰当的序列，结点在这个序列中的相互位置能反映出结点之间的逻辑关系，用编号的方法：

```
#define MAX_TREE_SIZE  100
    typedef   TElemType  SqBiTree[MAX_TREE_SIZE ];
    SqBiTree  bt;
```

按照自上而下、自左至右的顺序，遍历完全二叉树上的结点元素，并顺次存储在 一维数组中。

二叉树——顺序存储结构

- 例如：完全二叉树

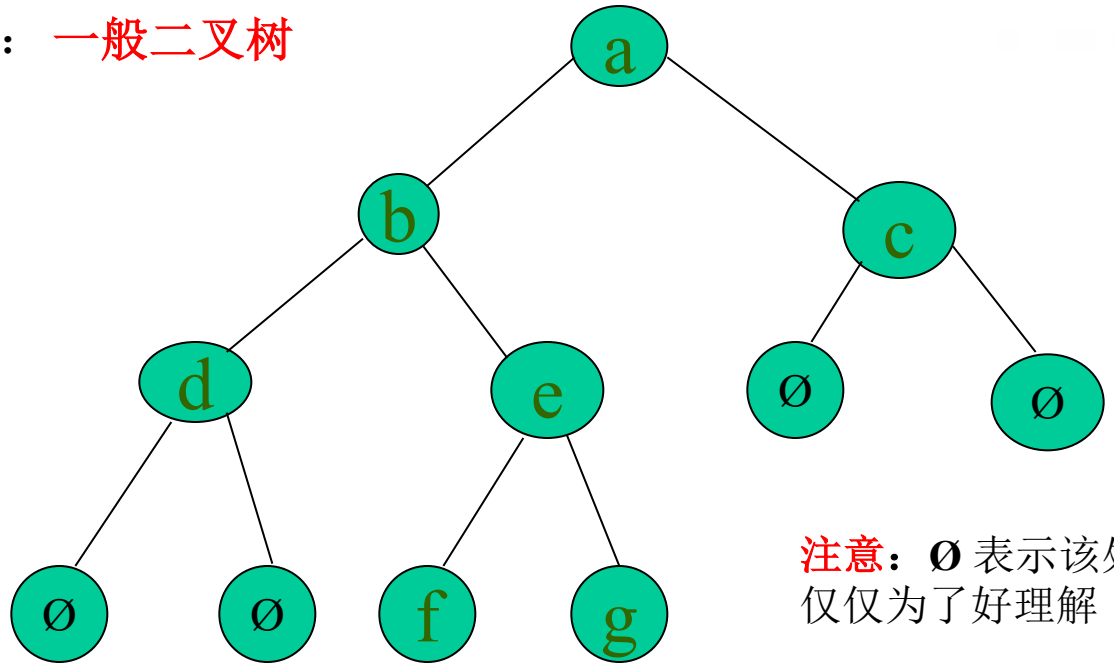


1	2	3	4	5	6	7	8	9	10	11	12
a	b	c	d	e	f	g	h	i	j	k	l

图4　完全二叉树的顺序存储结构

二叉树——顺序存储结构

● 例如： 一般二叉树



注意：Ø 表示该处没有元素存在
仅仅为了好理解

1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	Ø	Ø	Ø	Ø	f	g

图5 一般二叉树的顺序存储结构

二叉树——顺序存储结构

- 从树根起，自上层至下层，每层自左至右的给所有结点编号的缺点是：有可能对存储空间造成极大的浪费；
- 在最坏的情况下，一个深度为h且只有h个结点的右单支树确需要2^h-1个结点存储空间。
- 而且，若经常需要插入与删除树中结点时，顺序存储方式不是很好。

二叉树——二叉链表法

(2) 二叉链表法

二叉树的常用存储结构是链表。

- 二叉链表的结点结构：

在二叉链表中虽然没有指向双亲结点的指针，但可以通过指向孩子结点的指针找到“双亲”，因此二叉链表中的信息是完备的。

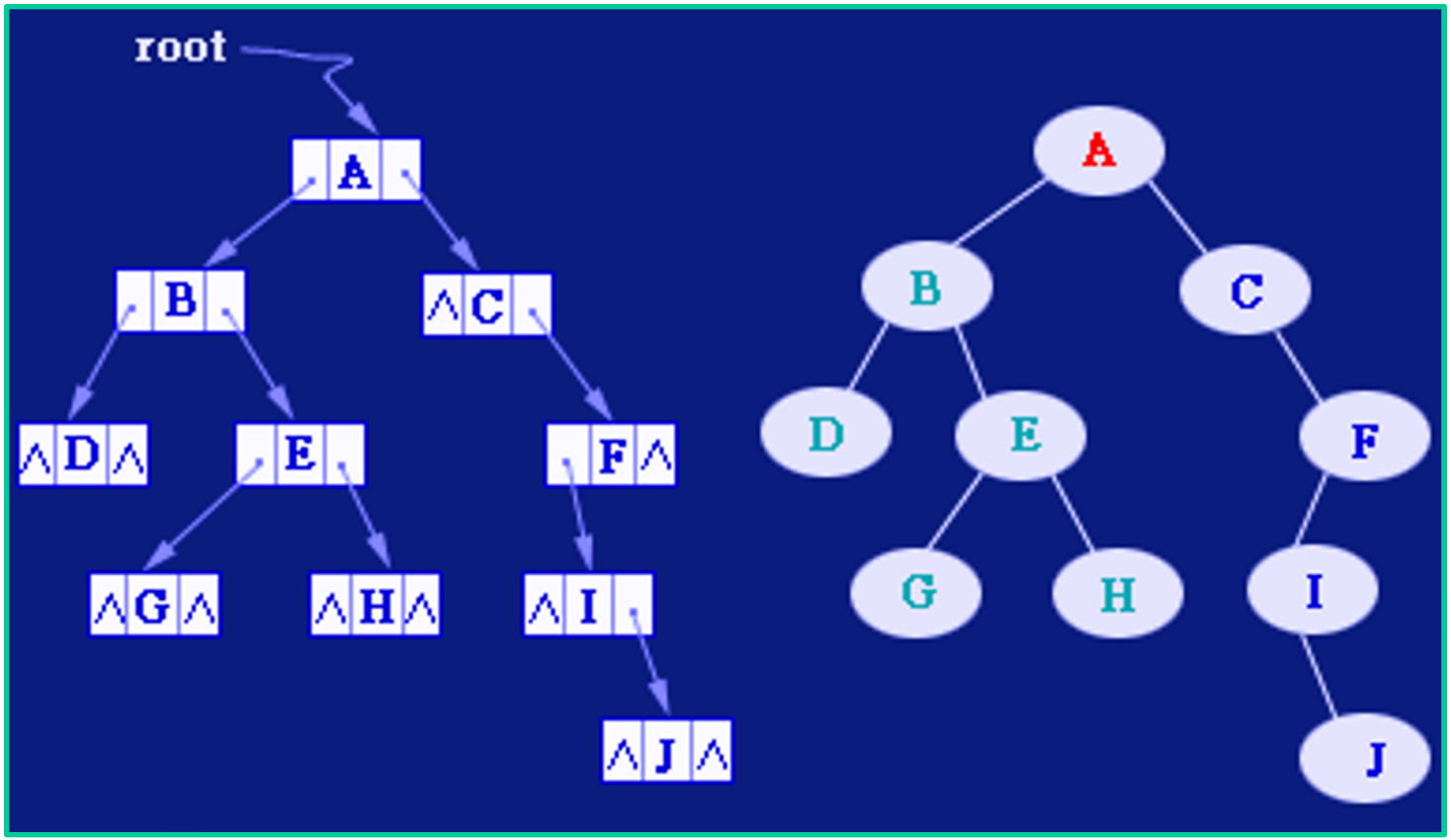
- 二叉树的二叉链表存储表示：

```
typedef struct BiTNode  {
    TElemType data;
    struct BiTNode *lchild,*rchild;     // 左右孩子指针
 } BiTNode,*BiTree;
```

有时也可用数组的下标来模拟指针，即开辟三个一维数组data,lchild,rchild分别存储结点的元素及其左，右指针域。

二叉树——二叉链表法

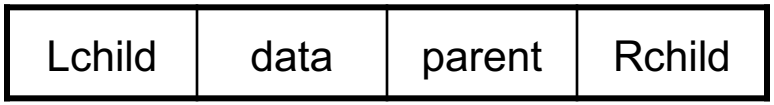
- 整个二叉树可以通过一个指向根结点的指针表示，下列右图所示的二叉树的二叉链表如下列左图所示。



二叉树——三叉链表法

➔ **(3) 三叉链表法**

➢ 三叉链表的结点结构：



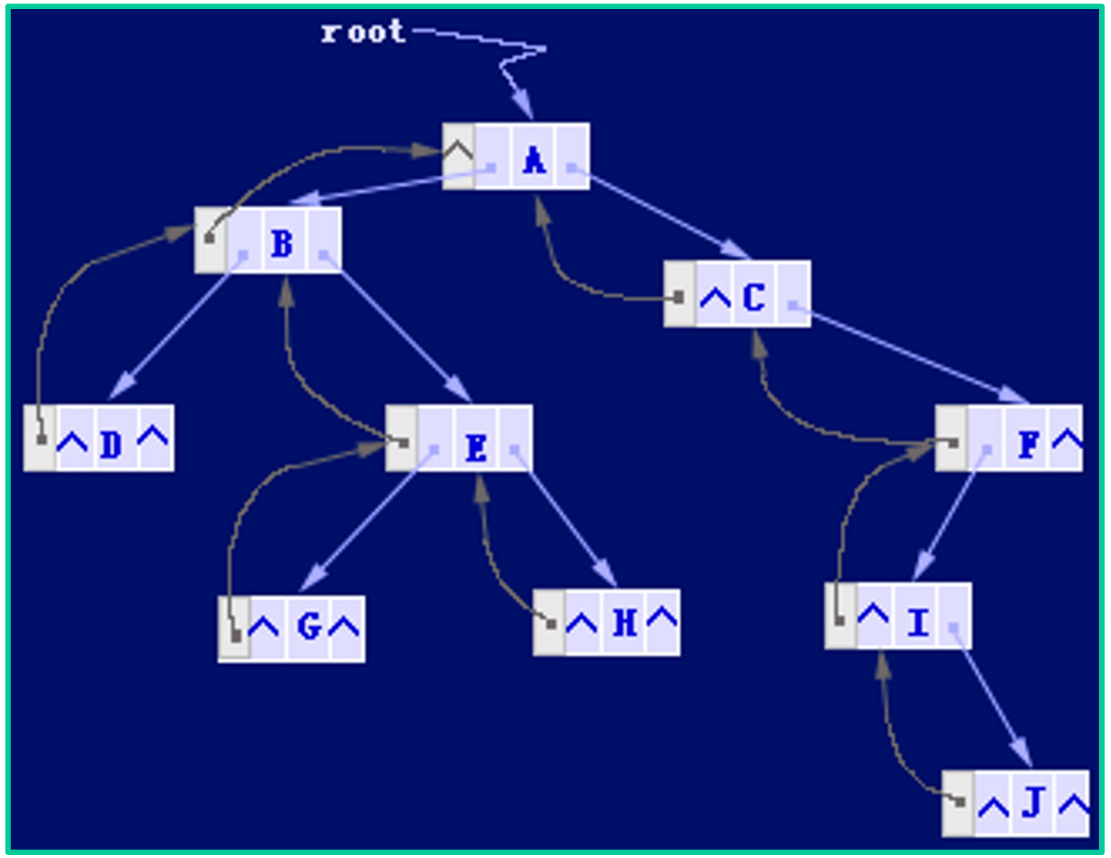
类似于线性表的双向链表，在二叉树的三叉链表中既有指示“后继”的信息，也有指示“前驱”的信息。

➢ 二叉树的三叉链表存储表示：

```
typedef struct TriTNode {
    TElemType data;
    struct TriTNode *Lchild, *Rchild;     // 左、右孩子指针
  struct TriTNode *parent;                // 双亲指针
} *TriTree;
```

二叉树——三叉链表法

- 和二叉链表相同，表示整个二叉树只需要一个指向根结点的指针即可。和上页相同的二叉树的三叉链表如下图所示。



本章小结

- 本章讨论**树和二叉树**两种数据类型的**定义**以及它们的**实现方法**。
- 树是以分支关系定义的层次结构，结构中的数据元素之间存在着“一对多”的关系，因此它为计算机应用中出现的具有层次关系或分支关系的数据，提供了一种自然的表示方法。
- **二叉树**是和树不同的另一种树型结构，它**有明确的左子树和右子树**；二叉树的几个重要特性也是我们应该熟练掌握的。

本章知识点与重点

- **知识点**

树的类型定义、二叉树的类型定义、二叉树的存储表示、二叉树的遍历以及其它操作的实现、线索二叉树、树和森林的存储表示、树和森林的遍历以及其它操作的实现、最优树和赫夫曼编码

- **重点和难点**

二叉树和树的遍历及其应用是本章的学习重点，而编写实现二叉树和树的各种操作的递归算法也恰是本章的难点所在 。