



数据结构与程序设计 (信息类)

Data Structure & Programming

北京航空航天大学 数据结构课程组

软件学院 林广艳

2023年春

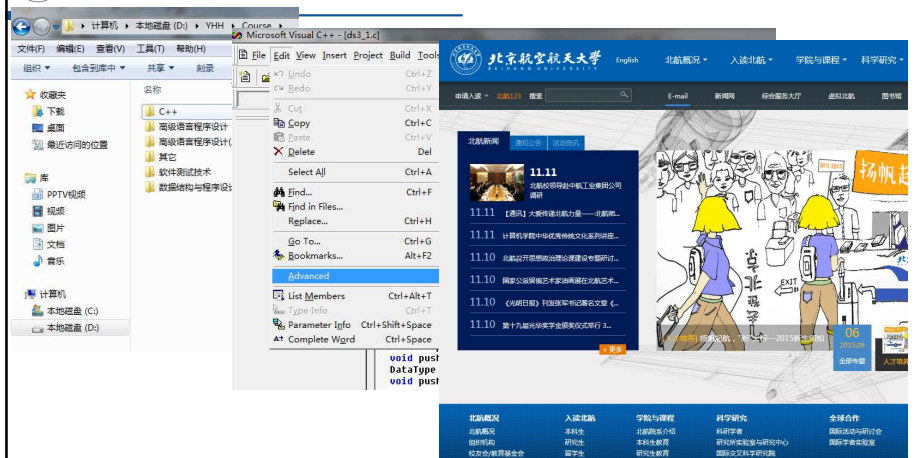


提纲：树和二叉树

- ◆ 树的基本概念
- ◆ 树的存储结构
- ◆ 二叉树的基本概念
- ◆ 二叉树的存储结构
- ◆ 二叉树的遍历
- ◆ 线索二叉树
- ◆ 二叉查找树
- ◆ 堆和表达式树
- ◆ 哈夫曼树

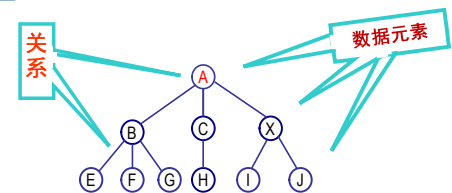
2

树是计算机中常见的数据组织方式



线性结构与树结构

$$A = (a_1, a_2, a_3, \dots, a_n)$$



线性结构	树结构
第一个数据元素 (无前驱)	根结点 (无前驱)
最后一个数据元素 (无后继)	叶子结点 (无后继)
其他数据元素 (一个前驱、一个后继)	树中其他结点 (一个前驱、多个后继)



提纲：树和二叉树

- ◆ **树的基本概念**
- ◆ 树的存储结构
- ◆ 二叉树的基本概念
- ◆ 二叉树的存储结构
- ◆ 二叉树的遍历
- ◆ 线索二叉树
- ◆ 二叉查找树
- ◆ 堆和表达式树
- ◆ 哈夫曼树

5



1. 树

- ◆ 树是由 $n \geq 0$ 个结点组成的有穷集合（用符号 D 表示）以及结点之间关系组成的集合构成的结构，记为 T
 - ✓ 当 $n=0$ 时，称 T 为空树
 - ✓ 在任何一棵非空的树中，有一个特殊的结点 $t \in D$ ，称之为该树的**根结点**；其余结点 $D - \{t\}$ 被分割成 $m > 0$ 个**不相交的子集** D_1, D_2, \dots, D_m ，其中，每一个子集 D_i 分别构成一棵树，称之为 t 的子树

递归定义

6

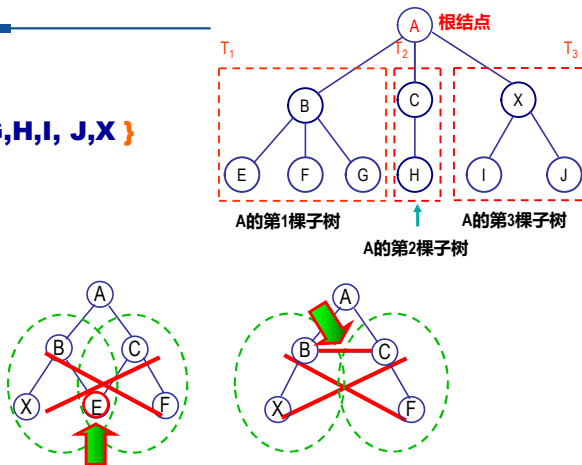


树结构

结点集合

$D = \{ A, B, C, E, F, G, H, I, J, X \}$

不相交的子集

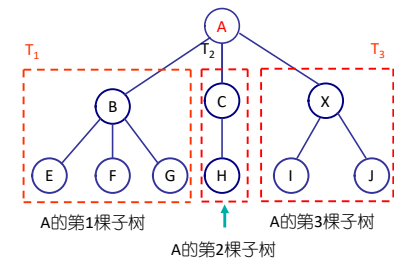


7



1.2 树的特点

- ◆ 树（逻辑上）的特点
 - ✓ 非空树中，**有且仅有一个**结点没有前驱结点，该结点为树的**根结点**
 - ✓ 除了根结点外，每个结点**有且仅有一个**直接前驱结点
 - ✓ 包括根结点在内，每个结点**可以有多个**后继结点



8

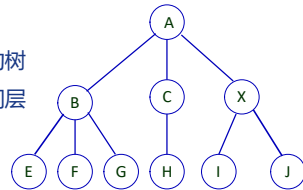


1.3 树的逻辑表示方法

1*. 文氏图表示法或凹入表示法

2. 树形表示法

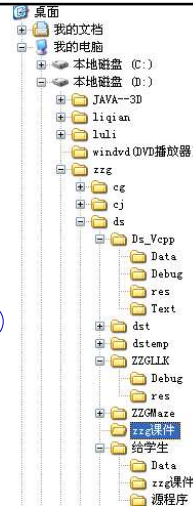
- ✓ 借助于自然界中一棵倒置的树的形状来表示数据元素之间层次关系的方法



3*. 嵌套括号法(广义表表示法)

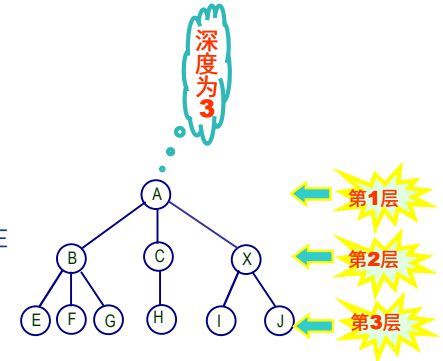
$A(B(E, F, G), C(H), X(I, J))$

第1棵子树 第2棵子树 第3棵子树



1.4 基本名词术语

- ◆ **结点的度**：该结点拥有的子树数目
- ◆ **树的度**：树中结点度的最大值
- ◆ **叶结点**：度为0的结点 (终端结点)
- ◆ **分支结点**：度非0的结点 (非终端结点)
- ◆ **树的层次**：根结点为第1层，若某结点在第i层，则其孩子结点(若存在)为第i+1层
- ◆ **树的深度**：树中结点所处的最大层次数 (高度)

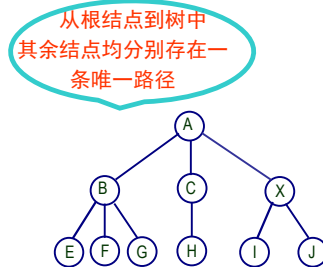


10



基本名词术语 (续)

- ◆ **路径**：对于树中任意两个结点 d_i 和 d_j ，若在树中存在一个结点序列 $d_1, d_2, \dots, d_i, \dots, d_j$ ，使得 d_i 是 d_{i+1} 的双亲 ($1 \leq i < j$)，则称该结点序列是从 d_1 到 d_j 的一条路径。路径的长度为 $j-1$
- ◆ **祖先与子孙**：若树中结点 d 到 d_s 存在一条路径，则称 d 是 d_s 的祖先， d_s 是 d 的子孙



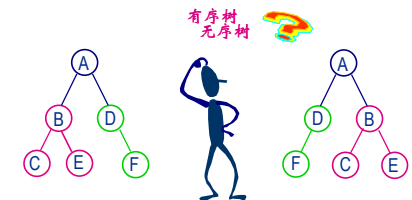
一个结点的祖先是根结点到该结点路径上所经过的所有结点
而一个结点的子孙则是以该结点为根的子树上的所有其他结点

11



基本名词术语 (续)

- ◆ **树林 (森林)**： $m \geq 0$ 棵不相交的树组成的树的集合
- ◆ **树的有序性**：若树中结点的子树的相对位置不能随意改变，则称该树为有序树，否则称该树为无序树



结点间关系：结点的子树的根称为该结点的孩子 (child)，相应地，该结点称为孩子结点的父结点 (或双亲, parent)。同一个双亲的孩子之间互称兄弟。



提纲：树和二叉树

- ◆ 树的基本概念
- ◆ **树的存储结构**
- ◆ 二叉树的基本概念
- ◆ 二叉树的存储结构
- ◆ 二叉树的遍历
- ◆ 线索二叉树
- ◆ 二叉查找树
- ◆ 堆和表达式树
- ◆ 哈夫曼树

13



2. 树的存储结构

◆ 两种存储结构

- ✓ 顺序存储结构
- ✓ 链式存储结构 (居多)

主要取决于要对
树进行何种操作

◆ 无论采用何种存储结构，需要存储的信息包括

- ✓ 结点本身的数据信息
- ✓ 结点之间存在的关系 (分支)

14

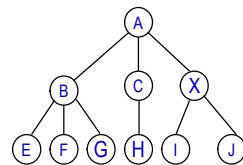


2.1 多重链表结构

◆ 定长结点的多重链表结构

链结点的构造:

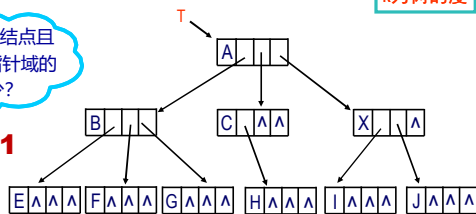
data	child ₁	child ₂	...	child _k
------	--------------------	--------------------	-----	--------------------



k为树的度

对于具有n个结点且
度为k的树,空指针域的
数目是多少?

$$n(k-1) + 1$$



缺点: 存储空间
比较浪费

15

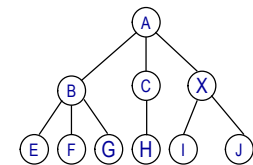


2.1 多重链表结构

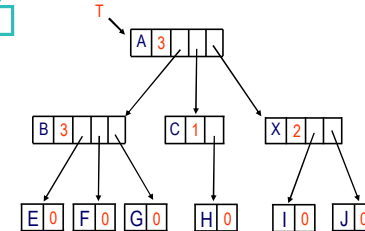
◆ 不定长结点的多重链表结构

链结点的构造:

data	k	child ₁	child ₂	child _k
------	---	--------------------	--------------------	-------	--------------------



结点的度



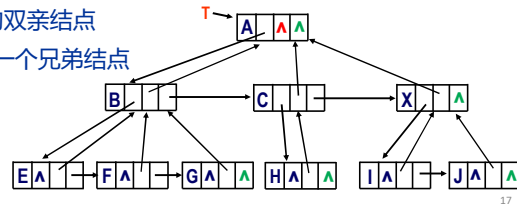
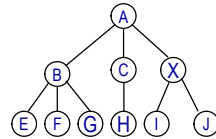
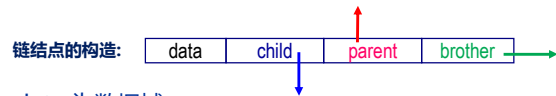
缺点: 操作不方便

16



2.2 三重链表结构

◆ 定长结点（孩子-双亲-兄弟）的多重链表结构



17



提纲：树和二叉树

- ◆ 树的基本概念
- ◆ 树的存储结构
- ◆ **二叉树的基本概念**
- ◆ 二叉树的存储结构
- ◆ 二叉树的遍历
- ◆ 线索二叉树
- ◆ 二叉查找树
- ◆ 堆和表达式树
- ◆ 哈夫曼树

18



3. 二叉树的基本概念

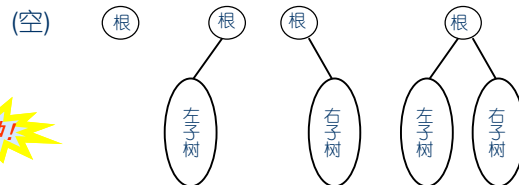
◆ 二叉树是 $n \geq 0$ 个结点的有穷集合 D 与 D 上关系的集合 R 构成的结构, 记为 T

- ✓ 当 $n=0$ 时, 称 T 为空二叉树
- ✓ 否则, 它为包含了一个根结点以及两棵不相交的、分别称之为 **左子树** 与 **右子树** 的二叉树

◆ 二叉树是有序树, 区分左右子树

二叉树的基本形态

5种!



19



思考：下面的说法正确与否

- ✗ 1. 度为2的**树**是二叉树
- ✗ 2. 度为2的**有序树**是二叉树



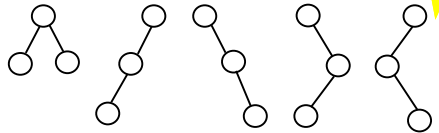
结论: 子树有严格的左、右之分且度 ≤ 2 的树是二叉树

20



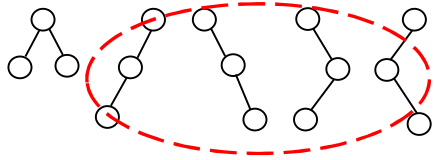
思考：下面的说法正确与否（续）

3. 具有三个结点的**二叉树**可以有多少种形态？



5种

✗ 4. 具有三个结点的**树**可以有 5 种形态



1种

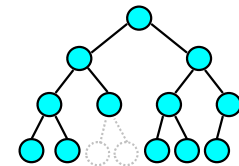
21



3.2 两种特殊形态的二叉树

◆ 满二叉树

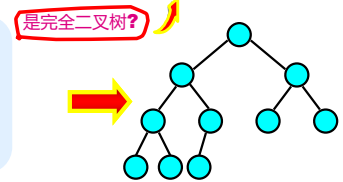
若一棵二叉树中的结点,或者为叶结点,或者具有两棵非空子树,并且叶结点都集中在二叉树的最下面一层,这样的二叉树为满二叉树。



不是完全二叉树

◆ 完全二叉树

若一棵二叉树中只有最下面两层的结点的度可以小于2,并且最下面一层的结点(叶结点)都依次排列在该层从左至右的位置上,这样的二叉树为完全二叉树。



22

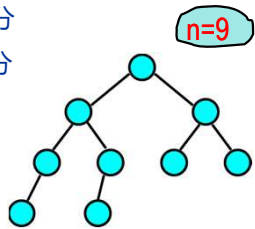


3.3 二叉树的性质

1. 具有n个结点的非空二叉树共有 **$n-1$** 个分支。

证明:

除了根结点以外, 每个结点有且仅有一个双亲结点, 即每个结点与其双亲结点之间仅有一个分支存在, 因此, 具有n个结点的非空二叉树的分支总数为 $n-1$ 。



23



3.3 二叉树的性质

2. 非空二叉树的第i层最多有 2^{i-1} 个结点($i \geq 1$)。

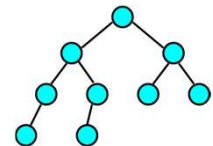
证明(采用归纳法)

(1) 当 $i=1$ 时,结论显然正确。非空二叉树的第1层有且仅有一个结点,即树的根结点

(2) 假设对于第j层($1 \leq j \leq i-1$)结论也正确,即第j层最多有 2^{j-1} 个结点

(3) 由定义可知, 二叉树中每个结点最多只能有两个孩子结点。若第 $i-1$ 层的每个结点都有两棵非空子树, 则第i层的结点数目达到最大。而第 $i-1$ 层最多有 2^{i-2} 个结点已由假设证明, 于是,应有

$$2 \times 2^{i-2} = 2^{i-1}$$

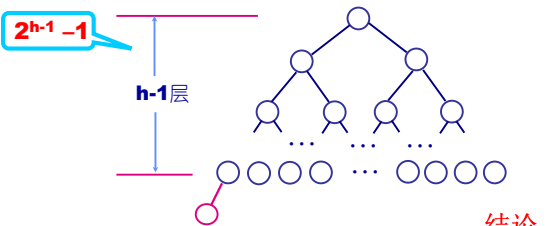


24



3.3 二叉树的性质

3. 深度为h 的非空二叉树最多有 $2^h - 1$ 个结点



思考

深度为h 的完全二叉树
至少有多少个结点

结论

2^{h-1}

$$S_n = \frac{a_1(1-q^n)}{1-q}$$

25



3.3 二叉树的性质

4. 若非空二叉树有 n_0 个叶结点,有 n_2 个度为2的结点,则 $n_0 = n_2 + 1$

证明:

设该二叉树有 n_1 个度为1的结点,结点总数为 n , 有

$$n = n_0 + n_1 + n_2 \quad \text{----- (1)}$$

设二叉树的分支数目为 B , 根据性质1, 有 $B = n - 1$ ----- (2)

这些分支来自度为1的结点与度为2结点, 即

$$B = n_1 + 2n_2 \quad \text{----- (3)}$$

由关系(1),(2)与(3),可得到 $n_0 = n_2 + 1$

推论

$$n_0 = n_2 + 2n_3 + 3n_4 + \dots + (m-1)n_m + 1 \quad (\text{P230, 习题7-4})$$

26



3.3 二叉树的性质

5. 具有n个结点的非空**完全二叉树**的深度为 $h = \lfloor \log_2 n \rfloor + 1$

证明:

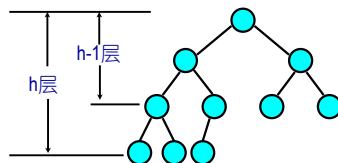
设深度为h,

则: $2^{h-1} - 1 < n \leq 2^h - 1$

即: $2^{h-1} \leq n < 2^h$

即: $h-1 \leq \log_2 n < h$

故: $h = \lfloor \log_2 n \rfloor + 1$



27



3.3 二叉树的性质

5. 具有n个结点的非空**完全二叉树**的深度为 $h = \lfloor \log_2 n \rfloor + 1$

思考

具有n个结点的**二叉树**
的**最小深度**是多少

28

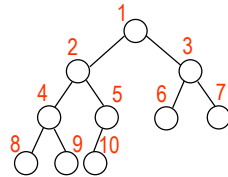


3.3 二叉树的性质

6. 若对具有 n 个结点的**完全二叉树**按照层次从上到下,每层从左到右的顺序进行编号,则编号为 i 的结点具有以下性质:

- (1) 当 $i=1$,则编号为 i 的结点为二叉树的根结点;
若 $i>1$,则编号为 i 的结点的双亲的编号为 $\lfloor i/2 \rfloor$;
- (2) 若 $2i>n$,则编号为 i 的结点无左子树;
若 $2i\leq n$,则编号为 i 的结点的左孩子的编号为 $2i$;
- (3) 若 $2i+1>n$,则编号为 i 的结点无右子树;
若 $2i+1\leq n$,则编号为 i 的结点的右孩子的编号为 $2i+1$.

$n=10$



29

单选题 1分

设置

若一棵二叉树有1001个结点,且无度为1的结点,则叶结点的个数为:

- ☐ A 498
- ☐ B 499
- ☐ C 500
- ☒ D 501

提交

30

单选题 1分

设置

若一棵深度为6的完全二叉树的第6层有3个叶结点,则该二叉树共有叶结点的个数为

- ☒ A 17
- ☐ B 18
- ☐ C 19
- ☐ D 20

提交

31

单选题 1分

设置

一个具有767个结点的完全二叉树,其叶子结点个数为

- ☐ A 383
- ☒ B 384
- ☐ C 385
- ☐ D 386

提交

32



3.4 二叉树的基本操作

1. **INITIAL(T)** 初始(创建)一棵二叉树。
2. **ROOT(T)**或**ROOT(x)** 求二叉树T的根结点, 或求结点x所在二叉树的根结点。
3. **PARENT(T,x)** 求二叉树T中结点x的双亲结点。
4. **LCHILD(T,x)**或**RCHILD(T,x)** 分别求二叉树T中结点x的左孩子结点或右孩子结点。
5. **LDELETE(T,x)**或**RDELETE(T,x)** 分别删除二叉树T中以结点x为根的子树或右子树。
6. **TRAVERSE(T)** 按照某种次序(或原则)依次访问二叉树T中各个结点, 得到由该二叉树的所有结点组成的序列。
7. **LAYER(T,x)** 求二叉树中结点x所处的层次。
8. **DEPTH(T)** 求二叉树T的深度。
9. **DESTROY(T)** 销毁一棵二叉树。
-

重点

遍历

33

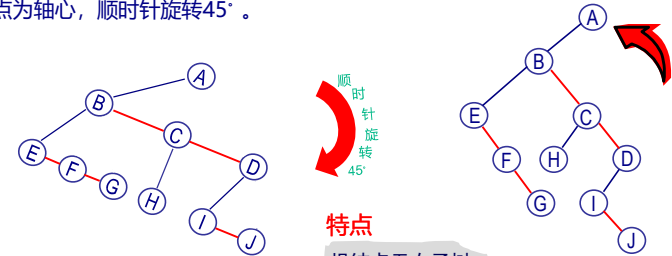


3.5 二叉树与树、树林之间的转换*

◆ 树与二叉树的转换步骤

- (1) 在所有相邻的兄弟结点之间分别加一条连线;
- (2) 对于每一个分支结点, 除了其最左孩子外, 删除该结点与其他孩子结点之间的连线;
- (3) 以根结点为轴心, 顺时针旋转45°。

例



特点

根结点无右子树

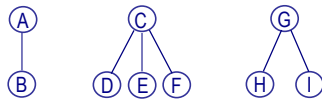
34



树林与二叉树的转换

◆ 树林与二叉树的转换

例



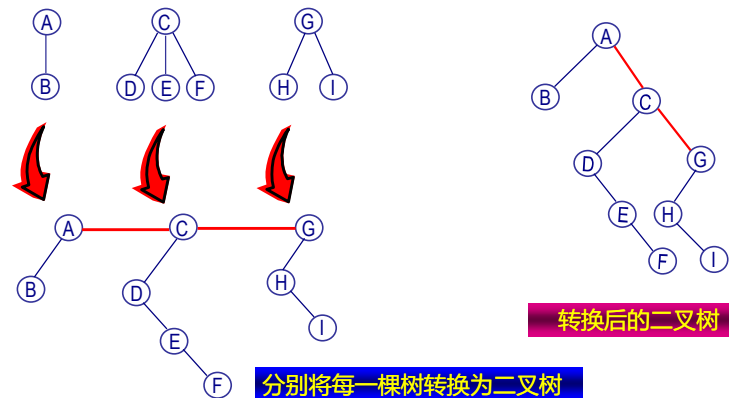
步骤

- (1) 分别将树林中每一棵树转换为一棵二叉树;
- (2) 从最后那一棵二叉树开始, 依次将后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子, 直到所有二叉树都这样处理。这样得到的二叉树的根结点是树林中第一棵二叉树的根结点。

35



示例：树林转换为二叉树



转换后的二叉树

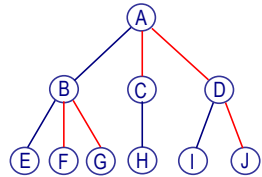
分别将每一棵树转换为二叉树

36



树林与二叉树的转换

◆ 二叉树还原为树



步骤

1. 若某结点是其双亲结点的左孩子, 则将该结点的右孩子以及当且仅当连续地沿此右孩子的右子树方向的所有结点都分别与该结点的双亲结点用一根虚线连接;
2. 去掉二叉树中所有双亲结点与其右孩子的连线;
3. 规整图形(即使各结点按照层次排列), 并将虚线改成实线。

37



提纲: 树和二叉树

- ◆ 树的基本概念
- ◆ 树的存储结构
- ◆ 二叉树的基本概念
- ◆ **二叉树的存储结构**
- ◆ 二叉树的遍历
- ◆ 线索二叉树
- ◆ 二叉查找树
- ◆ 堆和表达式树
- ◆ 哈夫曼树

38



4. 二叉树的存储结构

- ◆ 二叉树的顺序存储结构
- ◆ 二叉树的链式存储结构

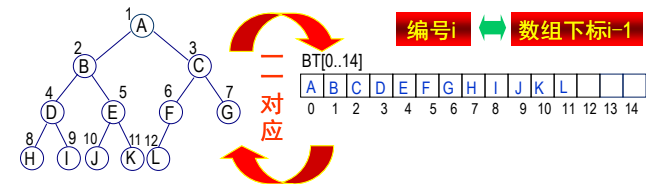
39



4.1 二叉树的顺序存储结构

◆ 完全二叉树的顺序存储结构

根据完全二叉树的性质6, 对于深度为h的完全二叉树, 将树中所有结点的数据信息按照编号的顺序依次存储到一维数组BT[0..2^h-2]中, 由于编号与数组下标一一对应, 该数组就是该完全二叉树的顺序存储结构。



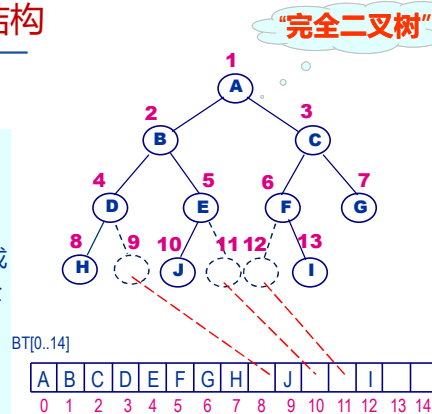
40



4.1 二叉树的顺序存储结构

◆ 一般二叉树的顺序存储结构

结论: 对于一般二叉树, 只须在二叉树中“添加”一些实际上二叉树中并不存在的“**虚结点**”(可以认为这些结点的数据信息为空), 使其在形式上成为一棵“**完全二叉树**”, 然后按照完全二叉树的顺序存储结构的构造方法将所有结点的数据信息依次存放于数组 $BT[0..2^h-2]$ 中。



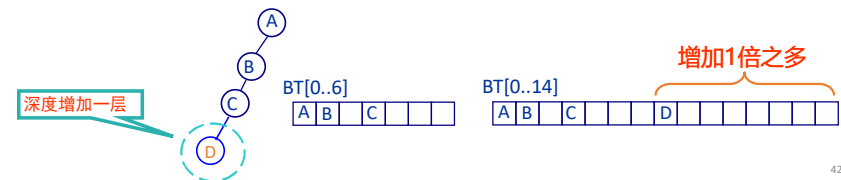
41



4.1 二叉树的顺序存储结构

◆ 二叉树的顺序存储结构特点

- ✓ 顺序存储结构比较适合满二叉树, 或者接近于满二叉树的完全二叉树, 对于一些称为“退化二叉树”的二叉树, 顺序存储结构的时空开销浪费的缺点表现比较突出
- ✓ 顺序存储结构便于结点的检索 (由双亲查子、由子查双亲)
- ✓ 顺序存储结构需要事先分配存储空间, 对于动态数据容易溢出



42



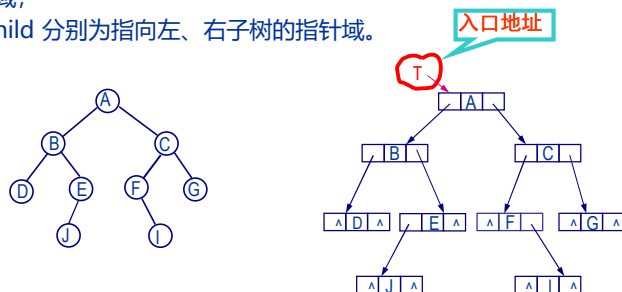
4.2 二叉树的链式存储结构

◆ 二叉链表

链结点的构造为 $\boxed{\text{lchild} \mid \text{data} \mid \text{rchild}}$

其中, data 为数据域;

lchild 与 rchild 分别为指向左、右子树的指针域。



43



二叉链表的C语言实现

二叉链表的结点类型定义为

```
typedef struct _Node
{
    ElemType data;
    struct _Node *lchild, *rchild;
} BTreeNode, *BTreeNodePtr;
```

BTreeNodePtr T, p, q;

44



4.2 二叉树的链式存储结构

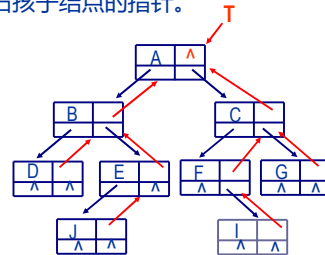
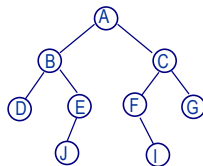
◆ 三叉链表

链结点的构造为

data	parent
lchild	rchild

其中: data 为数据域, parent为指向双亲结点的指针;

lchild 与rchild 分别为指向左、右孩子结点的指针。



45



问题4.1-查家谱*

同姓氏中国人见面常说的一句话是“我们五百年前可能是一家”。从当前目录下的文件in.txt中读入一家谱,从标准输入读入两个人的名字(两人的名字肯定会在家谱中出现),编程查找判断这两个人相差几辈,若同辈,还要查找两个人共同的最近祖先以及与他(她)们的关系远近。假设输入的家谱中每人最多有两个孩子,例如下图所示是根据输入形成的一个简单家谱。

若要查找的两个人是wangqinian和wangguoan,从家谱中可以看出两人相差两辈;若要查找的两个人是wangping和wanglong,可以看出两人共同的最近祖先是wangguoan,和两人相差两辈。

【输入示例】

假设家谱文件中内容为:

6

wangliang wangguoping wangguoan

wangguoping wangtian wangguang

wangguoan wangxiang wangsong

wangtian wangqinian NULL

wangxiang wangping NULL

wangsong wanglong NULL

从标准输入读取: wangping wanglong

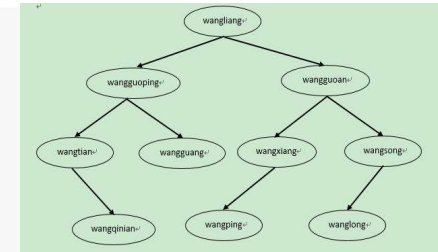
【输出示例】

wangguoan wangping 2

wangguoan wanglong 2

【说明】

wangping和wanglong共同的最近祖先是wangguoan,该祖先与两人相差两辈。



46



问题4.1*: 问题分析与设计

- ◆ 查家谱: 实际上就是查找相应结点。如果能得到结点至根的路径信息,就很容易计算出两个结点关系(如是否同辈、相差几辈、共同的祖先等)
- ◆ 如何在查找一个结点时得到其(从根结点至该结点的)路径信息:
 - ✓ 在前序查找过程中设置一个栈来保存路径信息
 - ✓ 一个简单的方法: 为每个结点增加一个指向父结点的指针信息, 这样在找到结点的同时, 也就获得了相应的路径

47



提纲: 树和二叉树

- ◆ 树的基本概念
- ◆ 树的存储结构
- ◆ 二叉树的基本概念
- ◆ 二叉树的存储结构
- ◆ 二叉树的遍历
- ◆ 线索二叉树
- ◆ 二叉查找树
- ◆ 堆和表达式树
- ◆ 哈夫曼树

48



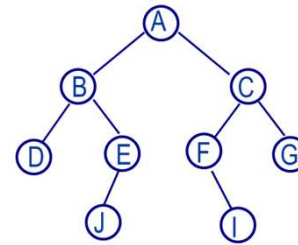
5. 二叉树的遍历

- ◆ **什么是二叉树的遍历**
- ◆ 二叉树的前序遍历
- ◆ 二叉树的中序遍历
- ◆ 二叉树的后序遍历
- ◆ 递归问题的非递归算法的设计*
- ◆ 二叉树的层次遍历
- ◆ 二叉树的顺序存储及遍历
- ◆ 由遍历序列恢复二叉树
- ◆ 树的遍历

49



5.1 什么是二叉树的遍历



- ◆ 输出所有结点的信息;
- ◆ 找出或统计满足条件的结点;
- ◆ 求二叉树的深度;
- ◆ 求指定结点所在的层次;
- ◆

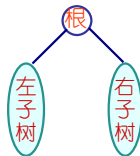
50



5.1 什么是二叉树的遍历

按照一定的顺序(原则)对二叉树中每一个结点都**访问一次(仅访问一次)**, 得到一个由该二叉树的所有结点组成的序列,这一过程称为**二叉树的遍历**

L--表示遍历左子树
R--表示遍历右子树
D--表示访问根结点



常用的遍历方法:

- 前序遍历(DLR)
- 中序遍历(LDR)
- 后序遍历(LRD)
- 按层次遍历

51

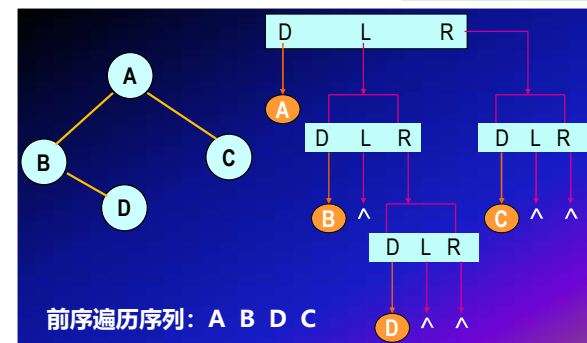


5.2 前序遍历

若被遍历的二叉树非空, 则

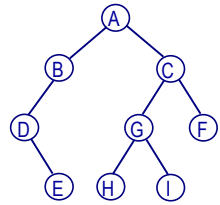
1. 访问根结点;
2. 以前序遍历原则遍历根结点的左子树;
3. 以前序遍历原则遍历根结点的右子树。

递归





5.2 前序遍历 (续)



前序序列: **ABDECGHIF**

递归算法

```
void preorder(BTNodeptr t)
{
    if (t != NULL)
    {
        VISIT(t); //访问t结点
        preorder(t->lchild);
        preorder(t->rchild);
    }
}
```

```
struct node{
    Datatype data;
    struct node *lchild, *rchild;
};
typedef struct node BTNode;
typedef struct node *BTNodeptr;
```



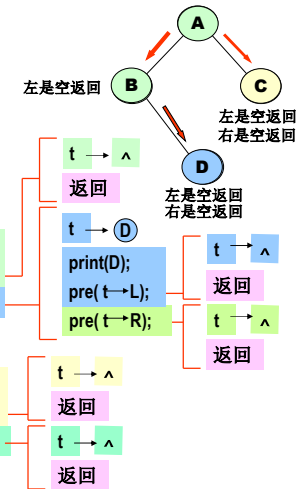
5.2 前序遍历 (续)

```
void preorder(BTNodeptr t){
    if (t != NULL)
    {
        VISIT(t); //访问t结点
        preorder(t->lchild);
        preorder(t->rchild);
    }
}
```

主程序

Pre(T)

前序序列: A B D C



54

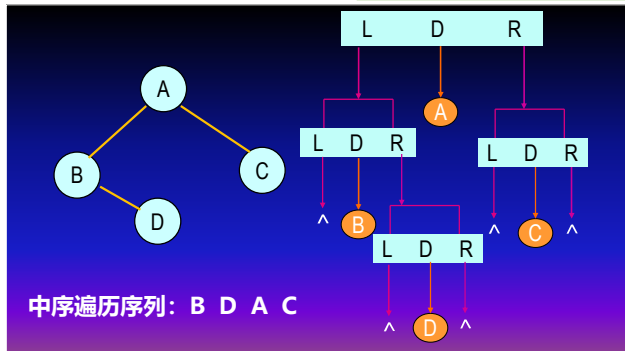


5.3 中序遍历

若被遍历的二叉树非空, 则

1. 以中序遍历原则遍历根结点的左子树;
2. 访问根结点;
3. 以中序遍历原则遍历根结点的右子树。

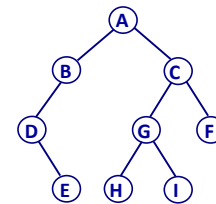
递归



中序遍历序列: **B D A C**



5.3 中序遍历 (续)



前序序列: **ABDECGHIF**

中序序列: **DEBAHGICF**

递归算法

```
void inorder(BTNodeptr t)
{
    if (t != NULL)
    {
        inorder(t->lchild);
        VISIT(t); //访问t结点
        inorder(t->rchild);
    }
}
```

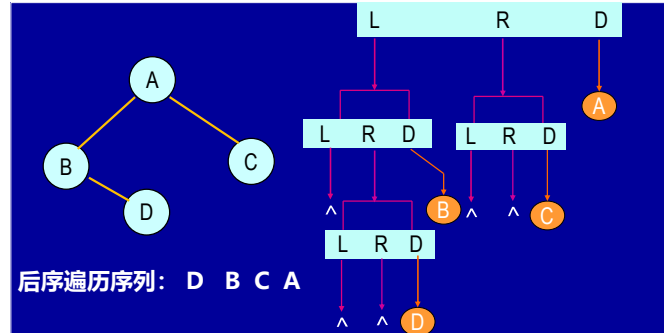
56

5.4 后序遍历

若被遍历的二叉树非空, 则

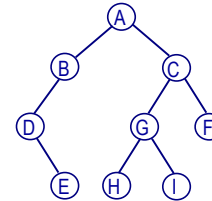
1. 以后序遍历原则遍历根结点的左子树;
2. 以后序遍历原则遍历根结点的右子树;
3. 访问根结点。

递归



57

5.4 后序遍历 (续)



前序序列: ABDECGHIF

中序序列: DEBAHGICF

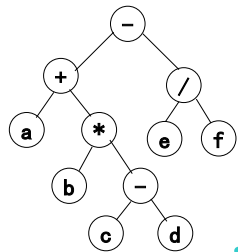
后序序列: EDBHIGFCA

递归算法

```
void postorder(BTNodeptr t){
    if (t != NULL)
    {
        postorder(t->lchild);
        postorder(t->rchild);
        VISIT(t); // 访问t结点
    }
}
```

58

练习: 三种遍历方法



前序遍历: - + a * b - c d / e f

中序遍历: a + b * c - d - e / f

后序遍历: a b c d - * + e f / -

这是一个表达式树

59

二叉树的遍历算法应用

//二叉树的拷贝

```
BTNodeptr copyTree(BTNodeptr src)
{
    BTNodeptr obj;
    if (src == NULL) obj = NULL;
    else
    {
        obj = (BTNodeptr)malloc(sizeof(BTNode));
        obj->data = src->data;
        obj->lchild = copyTree(src->lchild);
        obj->rchild = copyTree(src->rchild);
    }
    return obj;
}
```

- 1) 树拷贝时先拷贝当前结点, 再拷贝子结点 (同前序遍历);
- 2) 树删除时先删除子结点, 再删除当前结点。 (同后序遍历)

//二叉树的删除

```
void destoryTree(BTNodeptr p){
    if (p != NULL)
    {
        destoryTree(p->lchild);
        destoryTree(p->rchild);
        free(p);
        p = NULL;
    }
}
```

//求二叉树的高度

```
int max(x, y)
{if((x > y) return x; else return y;}

int heightTree(BTNodeptr p){
    if (p == NULL)
        return 0;
    else
        return 1 + max(heightTree(p->lchild),
                        heightTree(p->rchild));
}
```



小结：二叉树的遍历（递归算法）

前序遍历

若被遍历的二叉树非空，则

1. 访问根结点;
2. 以前序遍历原则遍历根结点的左子树;
3. 以前序遍历原则遍历根结点的右子树。

中序遍历

若被遍历的二叉树非空，则

1. 以中序遍历原则遍历根结点的左子树;
2. 访问根结点;
3. 以中序遍历原则遍历根结点的右子树。

后序遍历

若被遍历的二叉树非空，则

1. 以后序遍历原则遍历根结点的左子树
2. 以后序遍历原则遍历根结点的右子树;
3. 访问根结点。

61



小结：二叉树的遍历（递归算法）

//前序遍历

```
void preorder(BTNodeptr t){
    if (t != NULL) {
        VISIT(t); //访问t结点
        preorder(t->lchild);
        preorder(t->rchild);
    }
}
```

//中序遍历

```
void inorder(BTNodeptr t){
    if (t != NULL){
        inorder(t->lchild);
        VISIT(t); //访问t结点
        inorder(t->rchild);
    }
}
```

//后续遍历

```
void postorder(BTNodeptr t){
    if (t != NULL){
        postorder(t->lchild);
        postorder(t->rchild);
        VISIT(t); //访问t结点
    }
}
```



5. 二叉树的遍历

- ◆ 什么是二叉树的遍历
- ◆ 二叉树的前序遍历
- ◆ 二叉树的中序遍历
- ◆ 二叉树的后序遍历
- ◆ 递归问题的非递归算法的设计*
- ◆ 二叉树的层次遍历
- ◆ 二叉树的顺序存储及遍历
- ◆ 由遍历序列恢复二叉树
- ◆ 树的遍历

63



5.5 递归算法的非递归算法设计*

◆ 递归算法的优点

- (1) 问题的数学模型设计方法本身就是递归的，采用递归算法来描述非常自然
- (2) 算法的描述直观，结构清晰，简洁
- (3) 算法的正确性证明比非递归算法容易

◆ 递归算法的不足

- (1) 算法的执行时间与空间开销往往比非递归算法要大，当问题规模较大时尤为明显
- (2) 对算法进行优化比较困难
- (3) 分析和跟踪算法的执行过程比较麻烦
- (4) 描述算法的语言不具有递归功能时，算法无法描述

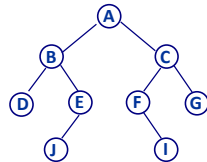
64



二叉树遍历的非递归算法设计

递归算法
(中序)

```
void inorder(BTNodeptr t){
    if (t != NULL)
    {
        inorder(t->lchild);
        VISIT(t); // 访问T指结点
        inorder(t->rchild);
    }
}
```



如何设计非递归算法?

65



中序遍历的非递归算法实现

中序遍历非递归算法思路

1. 若p指向的结点非空, 则将p指的结点的地址进栈, 然后, 将p指向左子树的根;
 2. 若p指向的结点为空, 则从堆栈中退出栈顶元素送p, 访问该结点, 然后, 将p指向右子树的根;
- 重复上述过程, 直到p为空, 并且堆栈也为空。

p=p->lchild;

p=p->rchild;

STACK[0..M-1] -- 保存遍历过程中结点的地址;
top -- 栈顶指针, 初始为-1;
p -- 为遍历过程中使用的指针变量, 初始时指向根结点。

66



中序遍历的非递归算法

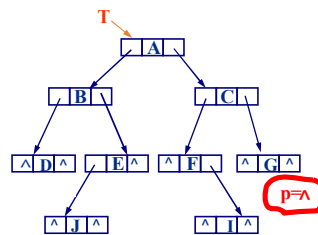
```
void inorder(BTNodeptr t){
    BTNodeptr stack[M], p = t;
    int top = -1;
    if (t != NULL)
    do{
        for (; p!=NULL; p=p->lchild)
        {
            stack[++top] = p;
        }
        p = stack[top--];
        VISIT(p);
        p = p->rchild;
    } while (p != NULL || top != -1);
}
```

前序遍历

堆栈

中序序列:

D, B, J, E, A, F, I, C, G



DBJEAFICG

67



后序遍历的非递归算法

stack1[0..M-1] -- 保存遍历过程中结点的地址;
stack2[0..M-1] -- 栈中结点是否可被访问标志: 0为不可访问, 1为可访问;
两个栈使用一个栈顶指针top;
p -- 为遍历过程中使用的指针变量, 初始时指向根结点。

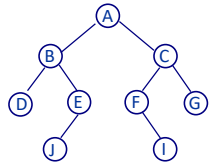
后序遍历算法基本思路

1. 若p所指结点非空, 则将p所指结点的地址及不可访问的标志进栈, 然后将p指向其左子树的根, 循环直到p为空;
 2. 栈顶结点的地址退栈送至P, 相应标志位退栈送至flag
 3. 若flag为不可访问标志, 则将p所指结点地址及该结点可访问标志重新进栈, 移动p指向其右孩子结点; 否则访问P所指结点, 并置P为空 (表明以P为根的子树已经访问完成)
- 重复上述过程, 直到p为空, 并且堆栈也为空。

68



后序遍历的非递归算法



后序序列: D, J, E, B, I, F, G, C, A

```

void postorder(BTNodeptr t){
    BTNodeptr stack1[M], p = t;
    int stack2[M], top = -1, flag;
    if (t != NULL)
        do{
            while (p != NULL){
                stack1[++top] = p; //当前P所指结点的地址进栈
                stack2[top] = 0; //当前P所指结点不可访问标志进栈
                p = p->lchild;
            } // P指向其左孩子结点
            p = stack1[top];
            flag = stack2[top--]; //退栈
            if (flag == 0){
                stack1[++top] = p; //当前P所指结点再次进栈
                stack2[top] = 1; //当前P所指结点可访问标志进栈
                p = p->rchild;
            }
            else{VISIT(p);
                p = NULL; //表明以P结点为根的树访问完成
            }
        }while (p != NULL || top != -1);
}
  
```

69



5. 二叉树的遍历

- ◆ 什么是二叉树的遍历
- ◆ 二叉树的前序遍历
- ◆ 二叉树的中序遍历
- ◆ 二叉树的后序遍历
- ◆ 递归问题的非递归算法的设计*
- ◆ **二叉树的层次遍历**
- ◆ 二叉树的顺序存储及遍历
- ◆ 由遍历序列恢复二叉树
- ◆ 树的遍历

70



5.6 按层次遍历

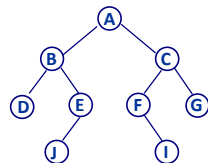
◆按层次遍历

✓若被遍历的二叉树非空, 则按照层次从上到下, 每一层从左到右依次访问结点

按层次遍历序列:

A B C D E F G J I

队列



前序、中序及后序遍历实质为**深度优先算法 (DFS)**

层次遍历为**一种广度优先算法 (BFS)**

71



层次遍历算法实现

```

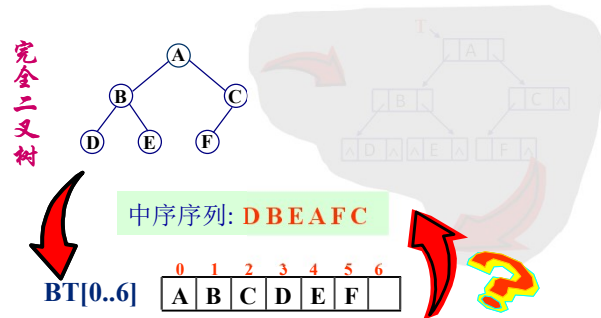
#define NodeNum 100
void layerorder(BTNodeptr t){
    BTNodeptr queue[NodeNum], p;
    int front, rear;
    if (t != NULL) {
        queue[0] = t;
        front = -1;
        rear = 0;
        while (front < rear){ // 若队列不空
            p = queue[++front];
            VISIT(p); // 访问p指结点
            if (p->lchild != NULL) // 若左孩子非空
                queue[++rear] = p->lchild;
            if (p->rchild != NULL) // 若右孩子非空
                queue[++rear] = p->rchild;
        }
    }
}
  
```

72



5.7 二叉树的顺序存储及遍历

- ◆ 已知具有 n 个结点的**完全二叉树**采用顺序存储结构，结点的数据信息依次存放于一维数组 $BT[0..n-1]$ 中，写出中序遍历二叉树的非递归算法



73



遍历算法的核心思想

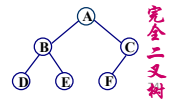
遍历算法的核心思想:

设置一个 **栈**，保存遍历过程中结点的**位置**；
设置一个 **变量**，初始时给出根结点的位置；

反复执行下列过程：

1. 若变量所指位置上的结点存在，则将该变量所指**位置**进栈，然后将该变量移到左孩子；
2. 若变量所指位置上的结点不存在，则从栈中退出栈顶元素送变量，访问该变量位置上的结点，然后将该变量移到右孩子；

直到变量所指位置上结点不存在，同时堆栈也为空。



算法设计

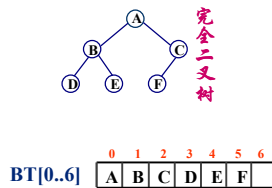
$STACK[0..M-1]$ -- 保存遍历过程中结点的**位置(下标)**；
 top -- 栈顶指针，初始为-1；
 i -- 为遍历过程中使用的 **位置** 变量，初始指向根结点。

$i=0, BT[0]$

1. 若 i 指向的结点非空 则将 i 进栈，然后，将 i 指向左子树的根： **$i=2*i+1$** ；
2. 若 i 指向的结点为空，则从堆栈中退出栈顶元素送 i ，访问该结点，然后，将 i 指向右子树的根： **$i=2*i+2$** ；

重复上述过程，直到 i 指结点不存在，并且栈空。

顺序存储结构



75



算法实现

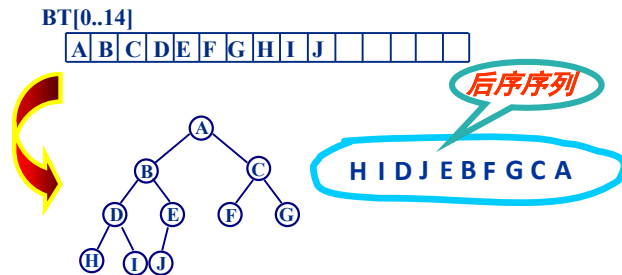
```
#define MaxSize 100
void inorder(Datatype bt[], int n){
    int stack[MaxSize], i, top = -1;
    i = 0;
    if (n >= 0){
        do {
            while (i < n)
            {
                stack[++top] = i; /* bt[i]的位置i进栈*/
                i = i * 2 + 1; /* 找到i的左孩子的位置 */
            }
            i = stack[top--]; /* 退栈*/
            VISIT(bt[i]); /* 访问结点bt[i] */
            i = i * 2 + 2; /* 找到i的右孩子的位置*/
        } while (!(i > n - 1 && top == -1));
    }
}
```

76



例3：完全二叉树的顺序存储

- ◆ 若某完全二叉树采用顺序存储结构,结点存放的次序为 A,B,C,D,E,F,G,H,I,J, 请给出该二叉树的后序序列



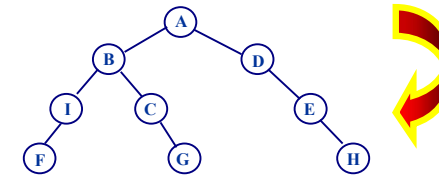
77



一般二叉树的顺序存储

BT[0..14]

A	B	D	I	C	E	F		G						H
---	---	---	---	---	---	---	--	---	--	--	--	--	--	---



前序序列: A B I F C G D E H
中序序列: F I B C G A D E H
后序序列: F I G C B H E D A



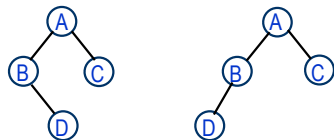
78

思考

能否利用遍历序列恢复二叉树



例



先序序列: A,B,D,C
后序序列: D,B,C,A
中序序列: B,D,A,C

先序序列: A,B,D,C
后序序列: D,B,C,A
中序序列: D,B,A,C

- 利用先序序列和中序序列恢复二叉树 ✓
利用中序序列和后序序列恢复二叉树 ✓
利用先序序列和后序序列恢复二叉树 ✗

若已知二叉树的前序序列与中序序列, 如何求二叉树的后序序列?



5. 二叉树的遍历

- ◆ 什么是二叉树的遍历
- ◆ 二叉树的前序遍历
- ◆ 二叉树的中序遍历
- ◆ 二叉树的后序遍历
- ◆ 递归问题的非递归算法的设计*
- ◆ 二叉树的层次遍历
- ◆ 二叉树的顺序存储及遍历
- ◆ 由遍历序列恢复二叉树
- ◆ 树的遍历

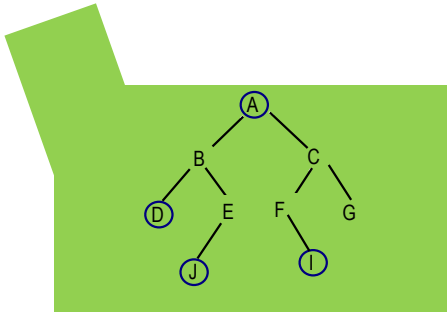
80



5.8 由遍历序列恢复二叉树

前序序列: A, B, D, E, J, C, F, I, G

中序序列: D, B, J, E, A, F, I, C, G



后序序列:

D, J, E, B, I, F, G, C, A

81



由遍历序列恢复二叉树

◆ 规律

已知前序序列和中序序列, 恢复二叉树

在前序序列中确定根;

到中序序列中分左右。

已知中序序列和后序序列, 恢复二叉树

在后序序列中确定根;

到中序序列中分左右。

82



练习

前序序列:

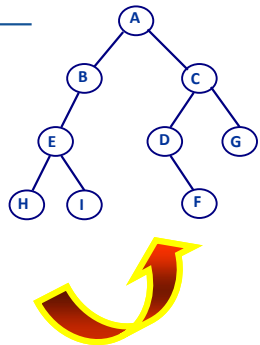
ABEHICDFG

中序序列:

HEIBADFCG

后序序列:

HIEBFDGCA



83



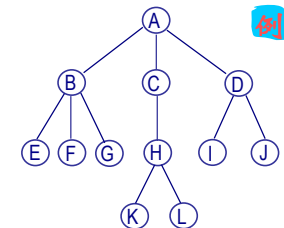
5.9 树的遍历

◆ 前序遍历 (深度优先)

若被遍历的树非空, 则:

- ① 访问根结点;
- ② 依次按前序遍历方式遍历根结点的每一棵子树。

类似转换后的二叉树的前序遍历



前序遍历序列: ABEFGCHKLDIJ

◆ 后序遍历 (深度优先)


若被遍历的树非空, 则:

- ① 依次按后序遍历方式遍历根结点的每一棵子树;
- ② 访问根结点。

类似转换后的二叉树的中序遍历

后序遍历序列: EFGBKLCIJD A

84



树的遍历算法

data	child ₁	child ₂	child _{MAXD}
------	--------------------	--------------------	-------	-----------------------

深度优先遍历算法

广度优先遍历算法

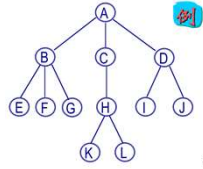
```


void DFSTree(TNodeptr t){
    int i;
    if (t != NULL) {
        VISIT(t); /* 访问t指向结点 */
        for (i = 0; i < MAXD; i++)
            if (t->next[i] != NULL)
                DFSTree(t->next[i]);
    }
}

void BFSTree(TNodeptr t){
    TNodeptr p; int i;
    if (t != NULL) {
        enqueue(t);
        while (!isEmpty()){ //若队列不空
            p = dequeue();
            VISIT(p);
            for (i = 0; i < MAXD; i++) //依次访问p指向的子结点
                if (p->next[i] != NULL) enqueue(p->next[i]);
        }
    }
}
      
```

```

#define MAXD 3 //树的度
struct node{
    DataType data;
    struct node *next[MAXD];
};
typedef struct node TNode;
typedef struct node *TNodeptr;
      
```





5. 二叉树的遍历

- ◆ 什么是二叉树的遍历
- ◆ 二叉树的前序遍历
- ◆ 二叉树的中序遍历
- ◆ 二叉树的后序遍历
- ◆ 递归问题的非递归算法的设计*
- ◆ 二叉树的层次遍历
- ◆ 二叉树的顺序存储及遍历
- ◆ 由遍历序列恢复二叉树
- ◆ 树的遍历

86