1. **树和二叉树**

**6.1树（tree）的概念**

**在自然界和日常生活中，可以见到很多情形可以归结为树结构。如：家族谱系、行政管理机构、Windows磁盘文件管理系统等。**

**自然界的树是树根朝下，枝干和叶子向上生长，而我们讨论的树在生长方向上正好与其相反，它是倒长的树，即根朝上，枝干和叶子朝下。**

**例1：某家族谱系的一部分**

**例2：国家行政管理机构的一部分**

**例3：Windows磁盘文件的一部分**

**C:\**

**TC20**

**VC6.0**

**数据结构课件**

**数据结构讲稿**

**第一章**

**第二章**

**……**

**MyTc程序**

**Tc1**

**Tc2**

**……**

**MyVc程序**

**Vc1**

**Vc2**

**……**

**树是一种层次结构，属于非线性结构。我们学过的线性表可以灵活组织数据，但它受到线性结构的限制，表达层次结构不太方便。**

**6.1.1树的定义**

**●树（Tree）是n（n≥0）个结点的有限集合。它满足：**

**（1）仅有一个特定的结点，称为根（root）结点;**

**（2）其余结点分为m(m≥0)个互不相交的非空有限**

**集合,,……,,其中每个集合自身又是一棵树，称为根的子树（subtree）。**

**本条即是说，树结点之间的路径不能形成回路，否则称为图（下章介绍）。**

**●为了表述方便，把没有结点的树称为空树。**

**●树的定义具有递归性：即一棵树是由根及若干棵子树构成的，而子树又是由根及若干棵子树构成的，……。**

**●表达树的方法通常有4种：树形、凹入、集合和广义表**

**第一种：树形表示法**

A

B

C

D

E

F

G

H

**第二种：凹入表示法**

A

B

C

E

F

D

G

H

**第三种：集合嵌套表示法**

**第四种：广义表表示法 T(A(B,C(E,F),D(G,H)))**

**6.1.3 树的基本术语**

**为了对树的形态表述清楚和形象，通常引用树和人**

**的特征及术语来描述。**

**(1)结点和树的度（degree）**

**结点所拥有的子树的个数称为该结点的度，而树中各结点的度的最大值称为该树的度。**

A

B

C

D

E

F

G

H

**如：**

**●结点B、E、F、G和H的度数是0；**

**●结点C和D的度数都是2；**

**●结点A的度数是3，显然它也是树的度数。**

**(2)叶子（leaf）结点和分支结点**

**度为0的结点称为叶子（终端）结点；度不为0的结点称为分支（非终端）结点。**

**一棵树除了叶子结点就是分支节点。**

A

B

C

D

E

F

G

H

**如:**

**●结点 B、E、F、G和H的度数均为0，所以是叶子结点；**

**●结点A、C和D的度数均不为0，所以是分支结点。**

**(3)孩子（child）结点、双亲（parents又称父亲）结点、兄弟（brother）及堂兄弟结点**

**树中一个结点的子树的根（或说后继）称为该结点的孩子，该结点称为其孩子结点的双亲结点。同一个双亲的孩子结点互称为兄弟。双亲在同一层的结点互为堂兄弟。**

A

B

C

D

E

F

G

H

**如:**

**●A是B、C、D的双亲结点，反之，B、C、D是A的孩子结点；**

**●C是E、F的双亲结点，反之，E、F是C的孩子结点；**

**●D是G、H的双亲结点，反之，G、H是D的孩子结点；**

**●显然A没有双亲结点；**

**●由于B、C、D具有同一个双亲结点，所以他们互为兄弟，同样，E和F及G和H也互为兄弟。但E、F为一方和G、H为另一方互为堂兄弟。**

**(4)祖先（ancestor）和子孙（descendant）**

**祖先是从根到该所经分支上的所有结点。反之，以某结点为根的子树中的任一结点称为该结点的子孙。显然祖先和子孙关系是父子关系的延伸。**

A

B

C

D

E

F

G

H

**如：**

**●A是B、C、D、E、F、G和H的祖先；反之，结点B、C、D、E、F、G、H是的A的子孙。**

**(5)结点的层数（level）和树的深度(depth，或称高度height）**

**结点的层次从根结点开始算起，根结点的层数为1，其余结点的层数等于其双亲结点的层数加1。比如，如果某个结点的层数为h，则其子树就在第h+1层。**

**树中各个结点层数的最大值称为树的深度（高度）。**

A

B

C

D

E

F

G

H

**如:**

**●A的层数为1，B、C和D的层数为2，E、F、G、H的层数为3；**

**●树的深度（高度）为3。**

**(6)有序树（ordered tree）和无序树（unordered tree）**

**若一棵树中结点的各子树从左到右是有次序的，即若交换了某结点各子树的相对位置就构成不同的树，则称这棵树为有序树，否则称为无序树。**

**(7)路径（path）从树中的一个结点到另一个结点的路途（路径只能由上向下，不能横向或由下向上）**

A

B

C

D

E

F

G

H

**如：A-C-E,A-D-H等**

**但B-A-D-G不是路径。**

**(8)森林（forest）：m（m≥0）棵互不相交的树的集合**

A

B

C

D

E

F

G

H

A

B

C

D

**树的逻辑关系：**

**●树的任一结点都可以有0个或多个直接的后继结点（孩子结点，这也是非线性的原因），但至多只能有一个直接的前驱结点（双亲结点）；**

**●只有根结点没有前驱，叶子结点（终端结点）没有后继；**

**●祖先与子孙关系是父子关系的延伸，它定义了树中各结点之间的纵向次序；**

**●在有向树中，同一组兄弟结点从左至右有长幼之分。它定义了树中各结点之间的横向次序。**

**6.1.4树的基本操作**

**常见的操作：建立、遍历、查询、求结点的双亲和孩子、求树的深度、求结点的层数和判空等。**

**6.2 二叉树**

**一般的树规律性差，二叉树结构简单，存储和处理相对容易，而且一般的树可以转化为二叉树处理。**

**6.2.1二叉树的定义**

**●二叉树是n（n≥0）个结点的有限集合，除了空树（n=0）之外，由一个根结点及两棵不相交的左子树和右子树组成；**

**●二叉树每个结点的度数≤2；**

**●二叉树的定义是递归的。**

**二叉树有五种基本形态：**

**(1)空树 Φ**

**(2)只有根结点**

**(3)只有左子树**

**(4)只有右子树**

**(5)完整二叉树**

**注意：二叉树的子树一定要分出左右，否则不能称作二叉树。如下列树可以称作树但不能称作二叉树（因为结点B不能断定是左子树的分支**）。

**6.2.2 二叉树的性质**

**性质1：二叉树的第i层的结点数量最多为。**

A

B

C

D

E

F

G

**证明：第1层的结点数目最多为1（），第2层的结点数目最多为2（），……，则第i层的结点数目最多为。（由以下可知这种情形会达到满二叉树。）**

**性质2：深度为k的二叉树结点数目最多为。**

**证明：当其每层的结点数达到最大值时，该二叉树的结点数最多，由性质1可知，深度为k的二叉树结点数是：**

****

**（由以下可知这种情形称为满二叉树。）**

**性质3：在任意二叉树中，若叶子结点（度数为0的结点）数为，度数为2的结点数为，则有。**

**本性质是说，任意一颗二叉树，叶子结点比度数为2的结点的个数多一个。**

**证明：因为二叉树任意一个结点的度数均不超过2（即0,1,2）。设n为结点总数，**

**分别是度数为0,1,2的结点数，则****（公式1），**

**先分析二叉树的结点度数和分支数的关系（由上向下分析）：由于度数为0的结点（叶子结点）没有分支，度数为1的结点有一个分支，度数为2的结点有两个分支，所以总的分支数为即=；**

**此外，再分析二叉树的双亲结点和分支数的关系（由下向上分析）：除根结点之外的每个结点都是其双亲结点的一个分支，所以总分支数为，可得关系式，即****（公式2）；**

**由公式1和2，即和**

**可得，证毕。**

**如：叶子结点数为3（C，E，F），度数为2的结点数为2（A，B），满足性质3。**

**又如：叶子数结点为2（F，G），度数为2的结点数为1（A），满足性质3。**

**●满二叉树的定义：深度为k（k≥1）且结点数为的二叉树。**

**满二叉树的结点数达到最大值。**

**如：深度为3的满二叉树结点达到7个**

A1

B2

C3

D4

E5

F6

G7

**●完全二叉树的定义：对于满二叉树的结点，按下列规则编号：**

**(1)从根结点开始，自上而下；**

**(2)同一层自左至右。**

**满二叉树的结点编号后，任意取满二叉树的前若干**

**个连续的结点所对应的二叉树，称为完全二叉树。**

**完全二叉树的特点：除最后一层外，其余各层均是满的，最后一层，结点连续出现在左边。**

**请注意：满二叉树要求太特殊且严格，一般不容易满足，而完全二叉树条件低一些，容易满足，今后会经常用到它，所以要注意它。**

**如：深度为4的完全二叉树**

**又如：第2层结点不是连续出现在左侧，所以不是完全二叉树**

**再如：第3层未满，所以不是完全二叉树**

**性质4：具有n个结点的完全二叉树的深度为：**

**。**

**证明：由完全二叉树和满二叉树的关系可知，一棵深度为k的完全二叉树的结点数必介于深度为k-1和深度为k的满二叉树的结点数之间，则有（取等号的情况是：完全二叉树是深度为k的满二叉树），由此推出，即，由于k取整数,所以,证毕。**

**例如：若n=10,k=1+3=4（见后面）**

**性质5：如果一棵完全二叉树有n个结点，对所有结点按层次编号（即从上到下），每层从左到右从1开始编号，则对任意结点i（1≤i≤n），有：**

**(1)若i=1，则其为根，没有双亲。若i>1，则其双亲为(int)(i/2);**

**(2)若2i≤n（即它有左子树），则其左子树编号为2i;若2i>n，则其无左子树。若2i+1≤n（即它有右子树），则其右子树的编号为2i+1; 若2i+1>n，则其无右子树。**

**分析：编号为i=4的结点D，该树的结点数n=10；因i>1，所以其双亲结点是(int)(i/2)=2(即B)；又因2i=2\*4≤10，所以该结点有左子树（结点为H编号为2i=8）；再因2i+1=9≤10, 所以该结点有右子树（结点为I编号为2i+1=9）**

**再分析：编号为i=5的结点E，因i>1，所以其双亲结点是(int)(i/2)=2(即B)；又因2i=2\*5≤10，所以该结点有左子树（结点为J编号为2i=10）；再因2i+1=11>10, 所以该结点没有右子树。**

**6.3 二叉树的存储结构**

**常用顺序存储结构和链式存储结构。**

**6.3.1 二叉树的顺序存储结构**

**二叉树的形状可能繁多且不固定，不好掌握规律，而进行顺序存储恰恰相反，要求规律性强。所以这种存储一定是规律性较强的二叉树才适合。完全二叉树符合这一点，这也是它被定义的原因之一。**

**●对于完全二叉树进行结点编号（自上而下，自左至右）后，编号可以反映结点的分支和从属关系，将这些结点存入一维数组时，编号和数组下标可以对应起来。**

**例如：**

**存储方式：**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **位置** | **[1]** | **[2]** | **[3]** | **[4]** | **[5]** | **[6]** | **[7]** | **[8]** | **[9]** | **[10]** |
| **结点** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** |

**●对于一般的二叉树，不易直接采用顺序存储，可以虚补成完全二叉树后再用顺序存储的方法存储。**

**如：一棵普通二叉树**

**由上图用虚结点（@）补成的完全二叉树再顺序存储**

**存储方式：**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **位置** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| **结点** | **A** | **B** | **C** | **@** | **D** | **E** | **@** | **@** | **@** | **F** | **@** | **@** | **G** |

**定义：**

**#define MAXSIZE 100**

**typedef char DataType;**

**typedef struct**

**{**

**DataType bt[MAXSIZE];**

**int num;**

**}SeqBTree;**

**6.3.2 二叉树的链式存储结构**

**结点模式：**

Rchild

Data

Lchild

A

C

B

E

D

G

F

**定义：**

**typedef struct node**

**{**

**DataType data;**

**struct node \*lchild,\*rchild;**

**}BTree;**

**此种定义称为二叉树的二叉链表。**

**在链式结构中，通过每个结点的左右指针域，可以找到其孩子结点，但要找双亲结点不方便，因此，可以增加一个指针域保存双亲结点的地址，称为“带双亲指针的二叉链表”或称为“三叉链表”。**

**结点模式：**

Rchild

Parents

Data

Lchild

A

C

B

D

E

G

F

**定义：**

**typedef struct node**

**{**

**DataType data;**

**struct node \*lchild,\*parents,\*rchild;**

**}ThTree;**

**6.4 二叉树的遍历（又称访问，周游，走）**

**●遍历是指按指定的规律从根结点开始，对二叉树中的每个结点遍历一次且仅遍历一次。所谓遍历是指对结点做某种处理。如：输出信息、修改结点的值等。**

**●遍历可以采用递归方法（程序简单）和非递归方法（程序稍复杂）。从中可以寻出“足迹”。**

**例如下列一颗简单的二叉树：**

**先（左） 后（右）**

**下（中）**

**遍历二叉树，可有3+1种方法（实际有6+1种）：**

**先序、中序、后序和层次法（另外逆先、中和后序三种方法不用）。**

**以下前三种方法从根部开始逆时针方向绕过各结点，形成一条蜿蜒“足迹”。**

**(1)先序法（又称先根法）**

**先序遍历：根，左子树，右子树**

**遍历的结果：A，B，C**

**遍历的足迹：沿途经过各结点的“左部”**

**(2)中序法（又称中根法）**

**中序遍历：左子树，根，右子树**

**遍历的结果：B，A，C**

**遍历的足迹：沿途经过各结点的“下部”**

**(3)后序法（又称后根法）**

**后序遍历：左子树，右子树，根**

**遍历的结果：B，C，A**

**遍历的足迹：沿途经过各结点的“右部”**

**(4)层次法**

**层次遍历：从根开始，层次自上到下，同层结点自左至右进行。**

**遍历的结果：A，B，C**

**遍历的足迹：第一层A，第二层B，C**

**例：下列二叉树的四种遍历**

**(1)先序遍历的结果：A，B，D，F，C，E，G**

**观察：A（根）**

**B，D，F（先序根的左子树）**

**C，E，G（先序根的右子树）**

**(2)中序遍历的结果：B，F，D，A，E，G，C**

**观察：B，F，D（中序根的左子树）**

**A（根）**

**E，G ，C（中序根的右子树）**

**(3)后序遍历的结果：F，D，B，G，E，C，A**

**观察：F，D，B（后序根的左子树）**

**G，E，C（后序根的右子树）**

**A（根）**

**注意：这种蜿蜒的路线以后不再画出，熟悉后放在我们的心里。**

**(4)按层次遍历的结果：A，B，C，D，E，F，G**

**观察：A根 （第一层）**

**B，C （第二层）**

**D，E （第三层）**

**F，G （第四层）**

**现象：左右子树次序打乱**

**A（根），B（左），C（右），D（左），E（右），F（左），G（右）**

**需要指出的是：先、中、后序同等重要，下面的情况可以看出，中序更重要，所以对其更要关注，以后还会使用到中序。**

**可以证明的是：若二叉树的结点各不相同（空或只有一个根结点的情况除外，即结点个数≥2）**

**(1)若中序遍历序列确定，若再有先序（或后序）遍历序列也确定，则该二叉树唯一确定；**

**(2)若二叉树的先序遍历和后序遍历确定，则该二叉树不能唯一确定；**

**如：下列两棵不同的二叉树先序遍历都是（A，B），而后序遍历都是（B，A）。**

* + 1. **二叉树的先序遍历及其算法**

**遍历规律：先遍历根结点，再遍历左子树，最后遍历右子树**

**●先序遍历的递归算法和程序（较简单）**

**void PreOrder(BTree \*bt)**

**{**

**if(bt!=NULL)**

**{**

**printf(“%c ”,bt🡪data);**

**//遍历根结点（输出数据）**

**PreOrder(bt🡪lchild); //递归遍历左子树**

**PreOrder(bt🡪rchild); //递归遍历右子树**

**}**

**}**

**●先序遍历的非递归算法和程序（稍复杂）**

**使用一个一维数组作为栈，存储二叉链表中的结点。思路为：从二叉树的根结点开始，沿左子树一直走到末端（左孩子为空）为止，在遍历过程中，依次把所遇结点入栈，当左子树为空时，从栈中退出栈顶结点，并将指针指向该结点的右孩子。如此重复，直到栈为空或指针为空时止。**

**void PreOrder1(BTree \*bt)**

**{**

**BTree \*s[100],\*p=bt; //数组s作为栈**

**int top=0; //top为栈顶指针**

**while(p!=NULL||top>0)**

**{**

**while(p!=NULL) //遍历根和左子树**

**{ printf(“%c ”,p🡪data);**

**s[++top]=p; p=p🡪lchild;**

**}**

**p=s[top--];p=p🡪rchild; //遍历右子树**

**}**

**}**

* + 1. **二叉树的中序遍历**

**遍历规律：先遍历左子树，再遍历根结点，最后遍历右子树。**

**●中序遍历的递归算法和程序（较简单）**

**void InOrder(BTree \*bt)**

**{**

**if(bt!=NULL)**

**{**

**InOrder(bt🡪lchild); //递归遍历左子树**

**printf(“%c ”,bt🡪data);**

**//遍历根结点（输出数据）**

**InOrder(bt🡪rchild); //递归遍历右子树**

**}**

**}**

**●中序遍历的非递归算法和程序（稍复杂）**

**使用一个一维数组作为栈，存储二叉链表中的结点。思路为：从二叉树的根结点开始，沿左子树一直走到末端（左孩子为空）为止，在走的过程中，把依次遇到的结点入栈，待左子树为空时，从栈中退出结点并访问，然后再转向它的右子树。如此重复，直到栈为空或指针为空为止。**

**void InOrder1(BTree \*bt)**

**{**

**BTree \*s[100],\*p=bt; //数组s作为栈**

**int top=0; //top为栈顶指针**

**while(p!=NULL||top>0)**

**{**

**while(p!=NULL){s[++top]=p;p=p🡪lchild;}**

**//遍历左子树**

**p=s[top--];**

**printf(“%c ”,p🡪data);p=p🡪rchild;**

**//遍历根和右子树**

**}**

**}**

* + 1. **二叉树的后序遍历**

**遍历规律：先遍历左子树，再遍历右子树，最后遍历根结点。**

**●后序遍历的递归算法和程序（较简单）**

**void PostOrder(BTree \*bt)**

**{**

**if(bt!=NULL)**

**{**

**PostOrder(bt🡪lchild); //递归遍历左子树**

**PostOrder(bt🡪rchild); //递归遍历右子树**

**printf(“%c ”,bt🡪data);**

**//遍历根结点（输出数据）**

**}**

**}**

**●后序遍历的非递归算法和程序（稍复杂）**

**利用栈来实现二叉树的后序遍历比先序和中序复杂得多，在后序遍历中，当搜索指针指向某一个结点时，不能马上进行访问，而先要遍历左子树，所以此结点先要入栈保存，当遍历完它的左子树后，再次回到该结点，还不能访问它，再次退栈时，才能访问该结点。**

**为了区分同一结点的两次进栈，引入一个次数标志，一个元素第一次进栈标志为0，第二次为1，并将标志存入另一个栈中，当从标志栈中退出的元素为1时，访问结点。**

**void PostOrder1(BTree \*bt)**

**{**

**BTree \*s1[100],\*p=bt;**

**//栈s1存放树中的结点**

**int s2[100],b,top=0; //栈s2存放进栈标志**

**do**

**{**

**while(p!=NULL) //遍历左子树**

**{s1[top]=p;s2[top++]=0;**

**//第一次进栈标志为0**

**p=p🡪lchild;}**

**if(top>0)**

**{**

**b=s2[--top]; p=s1[top];**

**if(b==0)**

**{s1[top]=p;s2[top++]=1;**

**//第二次进栈标志为1**

**p=p🡪rchild;} //遍历右子树**

**else {printf(“%c ”,p🡪data); p=NULL;}**

**//遍历根**

**}**

**}while(top>0);**

**}**

**例：算术表达式a+b\*c-d-e/f按不同的次序遍历此二叉树，将访问的结点按先后次序排列起来的次序是：**

**先序序列为(前缀表达式)：-+a\*b-cd/ef**

**中序序列为(中缀表达式)：a+b\*c-d-e/f**

**后序序列为(后缀表达式)：abcd-\*+ef/-**