

## 第 8 章 广义表

广义表是一种特殊的结构，它兼有线性表、树、图等结构的特点。从各层元素各自具有的线性关系讲，它应该是线性表的拓广；从元素的分层方面讲，它有树结构的特点，但从元素的递归性和共享性等方面讲，它应该属于图结构。总之，它是一种更为复杂的非线性结构。

### § 8.1 广义表的逻辑结构

#### § 8.1.1 基本概念

广义表(Lists)是一个二元组

$$\text{Lists}=(D, R)$$

其中，

$$D=\{d_i \mid i=1, 2, \dots, n; n \geq 0; d_i \in D_0 \text{ 或属于某广义表, } D_0 \text{ 是数据对象}\}$$

$$R=\{LR\}$$

$$LR=\{\langle d_{i-1}, d_i \rangle \mid d_i \in D, 1 \leq i \leq n\}$$

广义表也常简称为表。通常，广义表记为（称为**广义表表达式**）：

$$Ls=(\alpha_1, \alpha_2, \dots, \alpha_n)$$

其中，每个  $\alpha_i$  称为  $Ls$  的一个**直接元素**（也称  $\alpha_i$  直属  $Ls$ ），其或为数据对象成员，或为满足本定义的广义表。例如，下面的

$$L = ((a,b), c, (d, (e)))$$

就是一个合法的广义表表达式。

由上列定义知，若不考虑  $\alpha_i$  的内部结构，则  $Ls$  是一种线性表。但若考虑到  $\alpha_i$  的内部结构， $Ls$  可能是非常复杂的。

下面给出几个相关概念。

**子表：**若某广义表  $L$  的某元素结点  $a$  本身也是一个广义表，则称  $a$  为广义表  $L$  的子表。

**长度：**我们仍然称  $Ls$  中的元素个数（即各  $\alpha_i$  的个数，不计  $\alpha_i$  内部的元素个数）为广义表  $Ls$  的**长度**，它与线性表的长度的概念是相同的。

**单元素、单元素表：**若数据元素为非表元素（即数据对象成员，或说简单元素，如基本数据类型、结构/记录等），则称其为**单元素**；若  $L_s$  中的元素均为**单元素**，则称其为**单元素表**。

**空表：**表内无元素（长度为 0）的表称为空表。

**表头：**称  $L_s$  的第 1 个元素为  $L_s$  的表头。

**表尾：**称  $L_s$  中除去表头后其余元素构成的表为表尾。

显然，表尾一定是表，但表头不一定。

**深度：** $L_s$  的深度  $\text{Depth}(L_s)$  递归地定义为：

$$\text{Depth}(L_s) = \begin{cases} 0 & : \text{若 } L_s \text{ 为单元素} \\ 1 & : \text{若 } L_s \text{ 为空表} \\ 1 + \text{MAX}_i (\text{Depth}(a_i)) & : \text{其它情况} \end{cases}$$

从定义知，广义表的深度，相当于广义表表达式中括号的最大嵌套层数。这里的定义，为了递归，我们将单元素也看作表。

**层：**对任一表  $L_s$ ，我们称  $L_s$  为  $L_s$  的第 1 层（层号为 1）， $L_s$  的各直接元素均为  $L_s$  的第二层元素（层号均为 2）；对任意其他元素  $x$ ，它的直接元素的层号就等于它的层号加 1。层号相同者称为**同层结点**。在这个定义中，我们将不同出现（是不同时出现吗？）答：正确，是指不同位置的出现的相同元素也看作是不同的元素，因此，同一个对象，就可能有不同的层号。**显然， $L_s$  的深度等于它中层号最大的元素的层号。**

**系列广义表：**由于广义表的元素往往也是广义表，所以需要一同考虑。它们称可追溯到同属于某一广义表的各广义表的全体为一个广义表系列（或系列广义表）。

**递归表：**若表  $L_s$  中某成员含有自己（即  $L_s$ ），则称  $L_s$  为递归表。

下面的叙述中，我们用大写字母表示具体的广义表，用小写字母代表数据对象的成员（单元素）。在同一个广义表系列中，相同字母表示同一对象。

下面给出了一个广义表系列。

- ①  $A=()$ ：空表，无头，无尾，长度为 0，深度为 1。
- ②  $B=(a,b,c)$ ：单元素表，头为  $a$ ，尾为  $(b,c)$ ，长度为 3，深度为 1。
- ③  $C=(a, (b, c, a))$ ：非单元素表，头为  $a$ ，尾为  $((b,c,d))$ ，长度为 2，深度为 2。
- ④  $D=(B, ((a,b), c))$ ：非单元素表，头为  $B$ ，尾为  $((a,b),c))$ ，长度为 2，深度为 3。
- ⑤  $E=(a, E)$ ：非单元素表，头为  $a$ ，尾为  $(E)$ ，长度为 2，深度为  $\infty$ ，事实上， $E=(a, E) = (a, (a, (\dots)))$  是个递归表。

有时，为了强调广义表名称，可将表名写在表的左括号前面，如上例中的  $D$  可写为：

$$D( B(a, b, c), F( G(a, b), c) )$$

### § 8.1.2 广义表逻辑图

为了表达广义表的逻辑结构，这里用一种图形来形象地表示广义表，称为广义表图。

设广义表为  $L(A_1, A_2, \dots, A_n)$ ，则它的广义表图形成方法为：

- 画一结点，标为  $L$ ，称其为广义表  $L$  的根；
- 对每个  $A_i$ ，若其为单元素，则画一个结点，标为  $A_i$  的值(或值的名称)，称该结点为终结结点，其即为  $A_i$  的根；
- 对每个  $A_i$ ，若其为子表，则按该规则画出  $A_i$  的广义表图；
- 从  $L$  的根到各  $A_i$  的根分别画一条边，将各  $A_i$  的根连在一起。

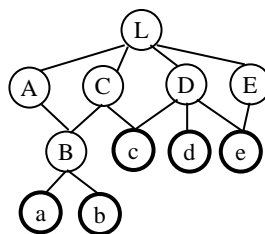


图 8-1 广义表  $L$  (见下式) 的逻辑图  
 $L(A(B(a,b)), C(B(a,b), c), D(c,d,e), E(e))$

**错误!未找到引用源。** 给出了广义表图的一个例子。

显然，广义表图中画出了广义表的所有结点。图中结点之间的边表示“包含/属于”关系，即若结点  $A$  到  $B$  有边，则表示  $B$  是  $A$  的子表之一 ( $A$  包含  $B$ )。

广义表图与树相似，只是由于存在共享元素，使得广义表图中某些结点的“父亲”不唯一。

### § 8.1.3 广义表的遍历

对广义表的遍历，如图结构类似，也有两种方式：深度优先和广度优先。具体遍历方式是把广义表对应的广义表图看作图结构，在该图结构上进行相应的遍历。

例如，对**错误!未找到引用源。**所示的广义表图，深度优先遍历结果为：

$L A B a b C c D d e E$

如果只访问单元素，则结果为：

$a b c d e$

对**错误!未找到引用源。**所示的广义表图，广度优先遍历结果为：

$L A C D E B c d e a b$

如果只访问单元素，则结果为：

$c d e a b$

有时候，将共享元素做为不同出现列出。(注：此处是否有例子？例子在哪？----

**答：删除掉“例如”）**

当然，也可以规定使共享元素不重复出现。例如，对广义表

$L=A(a, B(c, C(a, b), d), E(e, F(f)))$

深度优先（前序）结果为（共享元素按不同元素列出）：

$a, c, a, b, d, e, f$

### § 8.1.4 基本特性

从定义看出，广义表具有下列特性

- **宏观线性性**：对任意广义表，若不考虑它的元素的内部结构，则它是一个线性表，即它的直接元素之间是线性关系。

- **元素分层性**：如果将广义表中不同出现的元素看作不同的元素，则广义表是层次结构，即对任一元素，它只直属层号比它大 1 的元素。不过，广义表的层次性与树不同，在树中，任一元素只直属一个元素。

- **元素复合性**：广义表中的元素分两种：单元素和复合元素（子表）。其中，复合元素是由单元素根据广义表构成规则复合而成。因此，广义表元素的类型不统一。一个复合元素，在某一层上被当作元素，但就它本身的结构而言，也是广义表，因此也代表一个数据对象（数据元素的集合）。在其他数据结构中，并不把复合元素看作元素。

- **元素递归性**：广义表的任一元素，又可以是一个广义表（其它广义表或自身）。这种递归性使得广义表具有强有力的表达能力，这是广义表最重要的特性。

- **元素共享性**：在同一广义表中，任一元素（单元素或表）均可以出现多次，同一元素的多次出现都代表的是同一个目标，可以认为它们是共享同一目标。对多次出现的元素，显然可以从不同的位置（路径）访问它们，因此，具有该特性的表也称**再入表** (Reentrant List)。

### § 8.1.5 基本操作

广义表兼有线性表和树的特性，因此，它的基本操作也兼有线性表和树的一些操作，此外，还有一些特有操作，如求表头和表尾。

求表头和表尾是广义表的重要操作，通过它们，可以按递归方法处理广义表，也可实现一般访问。著名的人工智能语言 LISP 和 Prolog 其实就是以广义表为数据结构，通过求表头和表尾实现对象的操作。

## § 8.2 广义表的存贮结构

### § 8.2.1 基本存储方法

一般而言，为了能全面体现广义表的逻辑特性，广义表的存贮结构应能适应上节所指出的广义表的三种特性（宏观线性、元素递归性、元素共享性），这就决定了广义表存贮结构具有较高的复杂性，因此，除了在少数简单情况下使用顺序存储方法外，一般只采用链式存贮结构。

#### (一) 顺序存储方法

广义表的顺序存储，也只能是“广义”的顺序存储，因为我们不可能只存储基本元素，也不可能将各种元素（包括子表）用一个统一方法存储。“广义”的顺序存储，是指

按顺序存储方法存储广义表表达式，即广义表表达式中的括号、基本元素都存储在一片连续空间内。为了使存储的内容有统一的长度，对基本元素的存储，可以存储它们的地址（一般是相对地址，即序号）。**错误!未找到引用源。**给出的广义表的顺序存储为：

(	(	(	a	b	)	)	(	(	a	b	)	c	)	(	c	d	e	)	(	e	)	)	)
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

## (二) 链式存储

链式存储也可以有多种形式，具体使用时，应根据具体问题的要求选择不同的存贮结构。对广义表的几个逻辑特性，若有的与具体问题无关，则在选择存贮结构时就不必考虑，以简化存贮结构。下面给出一种通用的链式存贮结构——**分枝单链表法**，它可满足广义表的三个逻辑特性的要求，是一种常用的方法。

分枝单链表存贮的基本思想是用多个分枝单链表表示广义表。在这种方法中，为广义表的每个直接元素设置一个表结点（称为元素的头结点），各头结点依它们的逻辑次序链接成一个单链表，每个头结点设一个指针指向它所对应的直接元素的内容（若此直接元素为单元素），或指向它所对应的直接元素的广义表的头（若此直接元素为子表）。具体地讲，广义表  $Ls=(\alpha_1, \alpha_2, \dots, \alpha_n)$  的分枝单链表存贮方法为：

(a) 为每个  $\alpha_i$  设置一个结点  $H_i$ ，称其为  $\alpha_i$  的头结点，其结构为：

tag	...	pElem	next
-----	-----	-------	------

**tag:** 指示  $\alpha_i$  是单元素还是广义表(我们设 tag 为 0 时表示  $\alpha_i$  为单元素)。

**pElem:** 若  $\alpha_i$  为单元素，则指向  $\alpha_i$  的内容，否则指向  $\alpha_i$  对应的广义表(可以用第一个头结点  $H_1$  的指针代表广义表)。

**next:** 单链表链指针，即指向  $H_{i+1}$ 。

(b)  $\alpha_i$  为单元素时， $\alpha_i$  的内容应对应一片连续区域，可在它中设置一个长度字段。

(c) 直接元素构成的单链表，根据具体问题的需要，可以是带头结点的，或循环式的，或双向链式的或这些的组合。

(d) 如果不共享单元素内容，且单元素内容的长度相等（或差别不大），则可将单元素内容直接放到元素头结点中。

下面给出上例中给出的一个广义表序列中的五个广义表所对应的分枝单链表存储结构（图 8-0）。这里，单链表是非循环的且另设头结点，它们的第 1 个元素结点即充当各自的头结点。

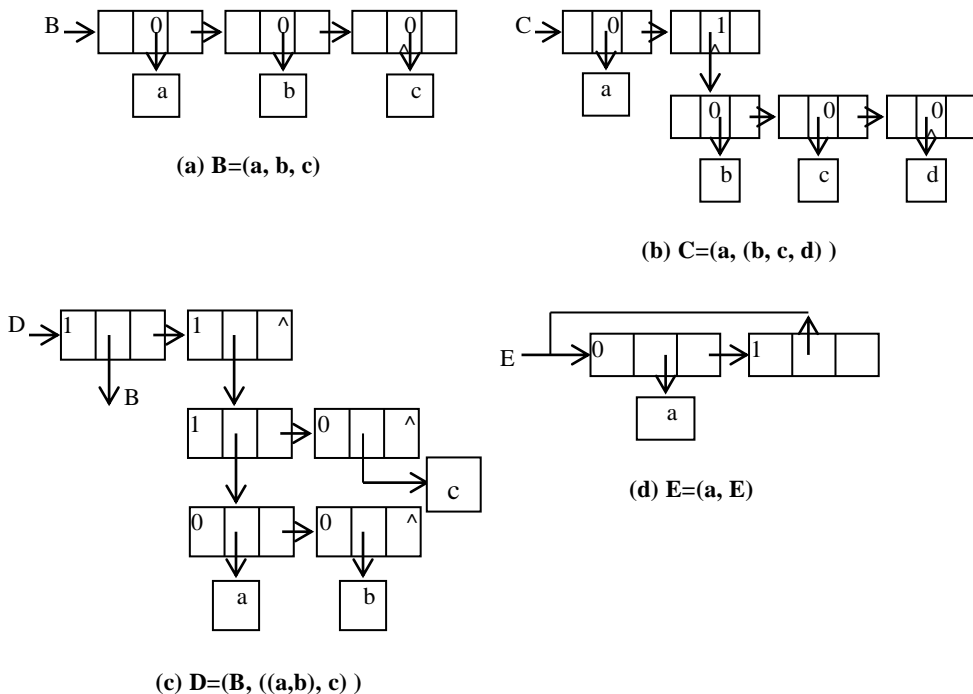


图 8-2 广义表存储结构示例

### § 8.2.2 链式结构的高级语言描述

这里，仅从数据结构方面给出广义表的分枝单链表的 C++描述。广义表的这种存储结构看上去比较复杂，其实只有一种结构：广义表结点 TGLListNode，其定义如下。

```
class TGLListNode
{
    char tag;
    char name[CNST_SizeListnodeName];
    TGLListNode *next;
    union
    {
```

```
char *pElem;  
TGListNode *pSub;  
};  
};
```

**tag:** 广义表元素结点种类标志，它表明对应结点是单元素(0)还是表(1)。

**name:** 结点名称。当结点是表时，也是该结点所对应的（广义）表的名称。

**next:** 指向同层中下个结点。

**pElem:** 指向结点对应的元素内容（地址）。

**pSub:** 指向结点对应的子表的头。这里先假定子表用指向它的第一个结点的指针表示。

因为子表与元素内容不同时出现，所以 pSub 与 pElem 可以共享存储区。这里我们用 union 实现共享。

### § 8.3 广义表对象模型\*

这里，我们将广义表作为一种抽象对象，建立它的访问接口。这是广义表使用和具体的存储实现的规范（模型）。从广义特点出发，我们将访问接口分为两部分：广义表元素（结点）接口和广义表接口。

#### § 8.3.1 广义表元素接口

首先我们考虑广义表元素的抽象模型。广义表元素的抽象模型同样是指与具体存储无关的属性。

广义表结点比较复杂，具有递归性，即它本身既可以是基本元素，也可以是子表的代表。对这种关联关系复杂的结点的接口的设计，一般的方法都“左右为难”。一般有两种方法（策略）：(a)结点观点：本着“本地原则”，重点提供对结点自己的属性的访问，而对表的操作，专门设立“表对象”，将它们放在表对象中；(b)表观点：将结点看作子表（当然也是广义表）的代表，将广义表的大部分操作都设置在结点上。

这两种方式各有优缺点。方式 a 比较简洁，对一些涉及到表的全局的操作，也容易处理，但不自然不灵活；方式 b 比较自然，并且使用灵活，但结点的接口变得复杂，而且对于一些涉及全局的操作（例如，对子表，若要在子表所在的整个表中查找，就是个全局操作问题）也不好处理，运行（被使用）效率也不及方式 a。我们这里选择方式 a，下面是具体的定义。

```
enum TTraverseMode {PreOrder, LevelOrder}; //定义一个枚举类型，用于表示遍历类型  
int const CNST_SizeListNodeName=10;  
class TGList0; //超前声明（因为在下面声明的 TGListNode0 中用到此类）
```

```

class TGListNode0 //广义表结点类
{
public: //protected:
    char tag; //结点标识: 0-单元素, 其他-广义表
    char name[CNST_SizeListNodeName]; //广义表名称
    unsigned long no; //结点编号

public:
    char IsList() { return tag; };

    char *GetName(){ return name; };
    void SetName(char *mName) { strncpy(name, mName, CNST_SizeListNodeName); };
    unsigned long GetNo(){ return no; };
    void SetNo(unsigned long mNo) { no=mNo; };
    char GetTag(){ return tag; };
    void SetTag(char mTag) { tag=mTag; };
};

```

下面对 TGListNode0 中主要成分做一说明。

**IsList():** 若本结点是非单元素结点, 则返回假 (0), 否则返回真 (1)。

**GetName():** 返回本结点的名字串的指针。


**SetName(char \*mName):** 为本元素结点设置 (复制) 名字。

**GetNo():** 返回本结点的编号。

**SetNo(unsigned long mNo):** 为本元素结点设置编号。

**GetTag():** 返回本结点的标志。

**SetTag(char mTag):** 为本元素结点设置标志。

 在该类中, 我们没有设置用于描述结点间关系的量, 如指示同层中下个元素的 next, 指示子表的 pSub 等。之所以这样, 是考虑到这些量与具体的存储结构密切相关, 在抽象结构中给出会影响灵活性。

对具体的存储结构下的广义表结点, 应该从该类派生。例如, 上节给出的 TGListNode, 就应该是 TGListNode0 的派生类:

```

class TGListNode : public TGListNode0
{
    TGListNode *next;
    union
    {
        char *pElem;
    }
};

```



```

    TGListNode *pSub;
}
//下面是 TGListNode0 中定义的成员函数的原形
... ..
};

```

### § 8.3.2 广义表接口

广义表接口是关于整个广义表（或广义表系列）的操作/访问规范。由于我们采用了结点简单接口（前面指出的方式 a），所以，关于结点所代表的子表的操作接口大都放在这里的广义表接口实现。广义表接口具体定义如下。

```

class TGList0 //广义表类
{
public:
    virtual long GetLen(TGListNode0 *pNode)=0;
    virtual long GetDepth(TGListNode0 *pNode)=0;
    virtual TGListNode0 *Copy(TGListNode0 *pH, long &k)=0;
    virtual TGListNode0 *GetHead(TGListNode0 *pNode)=0;
    virtual TGListNode0 *GetTail(TGListNode0 *pNode)=0;

    virtual TGList0 *GetList(TGListNode0 *pNode)=0;
    virtual TGListNode0 *GetElemNode(long idx)=0;

    virtual long GetFathers(TGListNode0 *pNode, TGListNode0 **fathers)=0;
    virtual TGListNode0 *GetFirstFather(TGListNode0 *pNode)=0;
    virtual TGListNode0 *GetNextFather(TGListNode0 *pNode)=0;

    virtual long IsMenber(TGListNode0 *p)=0;
    virtual long Locate(char *pE, TGListNode0 **pNodes,
                        TTraverseMode tm=PreOrder)=0;
    virtual long Cluster(TGListNode0 *pNode, char **e,
                        TTraverseMode tm=PreOrder)=0;
    virtual long Cluster(TGListNode0* pNode, TGListNode0 **pNodes,
                        TTraverseMode tm=PreOrder)=0;
    virtual long ClusterJuniors(TGListNode0 * pNode, TGListNode0 **pNodes,
                        TTraverseMode tm=PreOrder, int startLevel=1, int endLevel=-1)=0;
    virtual long ClusterSeniors(TGListNode0 *pNode, TGListNode0 **pNodes,
                        TTraverseMode tm=PreOrder, int startLevel=1, int endLevel=-1)=0;

```

```

virtual TGListNode0 *InsertNode(TGListNode0 *p,long sn)=0;
virtual TGListNode0 *DeleteNode(long sn=1)=0;
virtual long DeleteNode(long sn1=1, long sn2=1, TGListNode0 **pNodes)=0;
};

```

**GetLen(TGListNode0 \*pNode):** 返回 pNode 的子表的长度。

**GetDepth(TGListNode0 \*pNode):** 返回 pNode 的子表的深度。

**Copy(TGListNode0 \*pH, long &k):** 复制 pH 代表的表，并返回新复制的表的头结点指针（即对应于 pH 所指结点的结点的指针）。所复制的结点总数返回到 k 中（调用时 k 值应为 0）。

**GetHead(TGListNode0 \*pNode):** 返回 pNode 的子表的头所对应的结点的指针。

**GetTail(TGListNode0 \*pNode):** 返回 pNode 的子表的尾的第一个结点的指针。

**GetList(TGListNode0 \*pNode):** 返回结点 pNode 所对应的子表对象指针。若 pNode 无相应的对象（比如说，pNode 为单元素），则返回空，并触发异常。若 pNode 原来没有对应表对象，则返回的表对象是新创建的临时对象，只一次使用有效。当不使用时，系统负责将其撤销。

🔊 **GetList** 操作主要用于语法上的转换。在广义表中，每个元素可能是单元素，也可能是表。当元素为表的时候，元素的结点就对应该表。该操作就是要返回元素结点所对应的表的对象指针。一旦获得表对象指针，就可继续按表对象（TGList0）访问子表，因此，凡是返回表结点(TGListNode0)的操作，都可以借助该操作，转化为表对象(TGList0)。这就使得表结点和表对象间有机地过渡。例如，若表的头是表（尾肯定是表），则通过 GetList(GetHead())或 GetList(GetTail())，可继续按表处理头或尾。由于我们这里设计的是与具体存储实现无关的抽象操作，所以，这样处理可很好地顾及各种具体的存储实现。

**GetElemNode(long idx):** 返回本表中序号为 idx 的元素所对应的结点的地址。

**GetSub(TGListNode0 \*pNode, long idx):** 返回结点 pNode 对应的子表中的序号为 idx 的结点。若无子表，或无序号为 idx 的子表，则返回空。

**GetFathers(TGListNode0 \*pNode, TGListNode0 \*\*fathers):** 将结点 pNode 的各父结点的指针存入 fathers 数组中。

🔊 由于广义表的共享性和递归性，广义表的每个元素都有可能属于多个子表。将元素 elem 所属的子表所对应的元素（的结点），称为 elem 的父元素（结点）。

**GetFirstFather(TGListNode0 \*pNode):** 返回结点 pNode 对应的父结点中的第一个结点。

**GetNextFather(TGListNode0 \*pNode):** 返回结点 pNode 对应的父结点中的下一个结点。

GetFirstfather 和 GetNextFather 用于依次访问结点的各父结点。

**IsMenber(TGListNode \*p):** 检查结点 p 是否是本结点（对象本身）的成员。若是，则返回 p 的层号（相对与本结点），否则返回 0。

**Locate(char \*pE, TGListNode0 \*\*pNodes, TTraverseMode tm=PreOrder):** 在本表中查找内容的地址为 pE 的结点, 将它们的地址存入数组 pNodes。Tm 指出查找的方式 (前序或层序)。返回值为所找到的结点的数目。

**Cluster(TGListNode0 \*pNode, char \*\*e, TTraverseMode tm=PreOrder):** 串行化 (聚集) 操作。按遍历方式 tm, 将以 pNode 为头结点的表中的各元素结点的值的地址, 存入数组 e 中, 返回数组中元素的个数。

**Cluster(TGListNode0 \*pNode, TGListNode0 \*\*pNodes, TTraverseMode tm=PreOrder):** 串行化 (聚集) 操作。按遍历方式 tm, 将以 pNode 为头结点的表中的各元素结点的地址, 存入数组 pNodes 中, 返回数组中元素的个数。

**ClusterJuniors(TGListNode0 \*pNode, TGListNode0 \*\*pNodes, TTraverseMode tm=PreOrder, int startLevel=1, int endLevel=-1):** 按遍历方式 tm, 将结点 pNode 对应的子表中的层号 (相对于 pNode) 在 startLevel 和 endLevel 之间的结点的地址, 存入数组 pNodes, 返回数组中元素个数。

**ClusterSeniors(TGListNode0 \*pNode, TGListNode0 \*\*pNodes, TTraverseMode tm=PreOrder, int startLevel=1, int endLevel=-1):** 按遍历方式 tm, 将结点 pNode 对应的子表中的层号在 startLevel 和 endLevel 之间的结点的地址, 存入数组 pNodes, 返回数组中元素个数。这里, pNode 的层号为 1, pNode 的父结点层号为 2, 余类推。

**InsertNode(TGListNode \*p, long sn):** 在本表中的第 sn 个结点之前插入结点 p。返回被插入结点的地址。sn 可正可负, 具体含义同前 (线性表)。

**DeleteNode(long sn=1):** 在对应结点的子表中删除第 sn 个元素, 并将其地址返回。sn 含义同上。

**DeleteNode(long sn1=1, long sn2=1, TGListNode0 \*\*pNodes):** 在对应结点的子表中删除第 sn1~sn2 个元素, 并将被删元素的地址存入数组 pNodes, 返回实际被删元素的个数。

🔊 广义表具有线性表的某些特征, 这是因为, 对某个 (层) 广义表而言, 它的直接元素之间呈线性关系。因此, 许多关于线性表的操作, 同样适合广义表。具体就不在这里讨论了。

🔍 我们看到, 这里的 TGList0 类中大多数成员函数都有一个广义表结点指针作为参数。这样设置原因有二: 一是为了递归实现的方便; 二是为了增强灵活性。当然, 如果每个成员函数都带结点指针参数, 则它们都可以设置到结点类中! 这也是一种不错的选择!

## § 8.4 广义表的分枝单链表对象\*

这里, 我们以前面介绍的广义表的分枝单链表存贮结构为背景, 设计广义表的对象。

### § 8.4.1 结点对象

首先我们设置广义表元素结点类 `TGListNode`，每个 `TGListNode` 型对象，代表构成广义表的一个元素，它是前面介绍的抽象类 `TGListNode0` 的派生类，其数据部分是我们介绍存储结构时介绍的类。

```
struct TGListNode : public TGListNode0
{
    TGListNode *next; //指向同层中下个结点。
    union
    {
        char *pElem; //指向结点对应的元素内容。
        TGListNode *pSub; //指向结点对应的子表（的头）。
    };

    virtual TGListNode *GetNext(){return next;};
    virtual void SetNext(TGListNode *p){next = p;};
    virtual char *GetVal(){return pElem;};
    virtual void SetVal(char *pe ){pElem= pe;};

    virtual TGListNode *GetSub(){return pSub;};
    virtual void SetSub(TGListNode *p){pSub=p;};
    virtual TGListNode *GetParent(){return parent;};
    virtual void SetParent(TGListNode *p){parent=p;tag=1;};
};
```

该类中的主要成员已说明如下。

**GetNext():** 返回本结点代表的元素的后继元素所对应的结点的指针。

☞ 这实质上是线性表的求后继的操作。对于非顺序存储结构，每个元素都应设立表示后继或前驱关系的数据域，因此，该操作是对具体的存储结构的后继关系的数据表示的抽象。我们这里没有规定前驱操作(`GetPrior()`)，主要是考虑到，对某些存储结构，同时实现前驱和后继比较浪费存储空间，而在结点对象中直接实现求另一个关系，比较困难。对确需同时提供两个关系的直接访问的应用，可从该类派生新类，在新类中增加相应的关系的数据表示。

**SetNext(TGListNode0 \*p):** 为本结点代表的元素设置后继结点。注意，这与在表中插入一个结点是不同。该操作主要用来隐蔽结点的后继数据域。

**GetVal():** 返回本结点对应的元素的内容的地址。若本结点为子表，则返回空。这里，为了使用方便和适应范围广，将结点内容看作字节流（`char` 型）。

**SetVal(char \*pe):** 为本元素结点设置内容。

**GetSub():** 返回本结点对应的子表的头结点的地址。若本结点不是子表, 则返回空。

**SetSub(TGLListNode \*p):** 为本元素结点设置子表，即将以 pH 为头的表设置为本结点的子表。

**GetParent():** 返回本结点所直属的子表的头结点的地址。

**SetParent (TGLListNode \*p):** 为本结点设置父表，相当于强行将本结点作为以 pP 结点的子表。

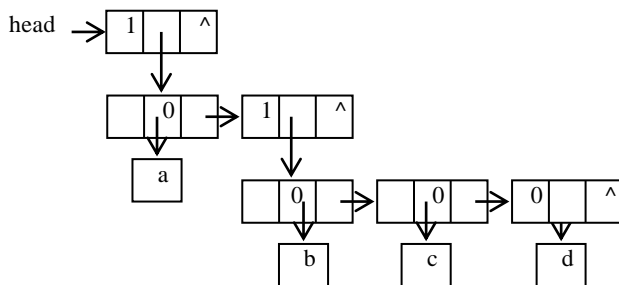


图 8-3 带总头结点的广义表分枝单链表

🔔上面这些操作，我们在 `TGListNode` 的父类中没有定义，这是为了简化问题。如果要考虑接口的统一，这些操作就需要在父类中定义了。

### § 8.4.2 分枝单链表对象

由于广义表结构信息已包含在结点类 `TGLListNode` 中, 故针对分枝单链表的广义表类的形式, 与前面给出的 `TGLList0` 类似。主要不同之处, 是在类中设立一个指向广义表头结点的指针 `head`。

为了处理方便，为每个独立的广义表设置一个头结点，令它的 `pSup` 指向广义表中第一个元素对应的结点，如图 8-0 所示。

```
class TGList: public TGList0<TElem>
{
    TGListNode *head; //指向广义表的头结点

public:
    //下面是在 TGList0 中定义的函数的声明（不含“=0”部分）。
};
```

至于操作的实现，我们放在下节中讨论。

## § 8.5 广义表操作的实现\*

本节通过几个有代表性的算法的实现，说明广义表的相关算法实现方法。

### § 8.5.1 一般问题

如果抛开广义表元素的内部结构，广义表就相当于线性表，因此，广义表的不涉及元素内部结构的操作的实现，与普通线性表类似。不涉及元素内部结构的操作主要是我们在 TGList0 中定义的操作，如 InsertNode()、DeleteNode()等。涉及整个表或元素内部结构的操作主要有 GetDepth()、GetFather()、Locate()、Cluster()、IsMenber()等。

对这类涉及整个表或元素内部结构的操作的实现，有一个重要问题需注意：循环问题。

广义表是允许递归的，即某表的元素中，可以包含自己（以自己为元素）。在存储结构中，这种情况表现为：从某个结点出发，顺着某条链路，可以返回到自己位置。这样，处理广义表时，若不知道递归元素是否已被处理，就可能导致无限循环。解决的方法是，为每个结点设立一个访问标志，每访问一个结点，就将其访问标志置为“已访问”。

这种处理方法又存在一个问题，就是访问标志如何设置。有两种方法：

a) 设立一个线性结构（一维数组，称为已访问数组），使每个元素对应一个广义表元素（结点），其值表示是否已访问。这种方法容易进行初始化（将所有元素都置为“未访问”），但不易按元素标识定位（已知元素标识，访问对应的已访问数组元素值）。

b) 在广义表结点中增设一个“已访问”标志位。这种方式的特点是易按元素标识定位（已知元素结点的指针时，可直接访问该元素的“已访问”标志），但初始化问题不易解决。可以在建立广义表结构（或插入新结点）时，将结点的“已访问”标志置为“未访问”，但是，当要重新初始化为“未访问”时，问题就没那么容易了，因为，此时的初始化已相当于广义表的遍历！

由于广义表是递归结构，所以相关算法采用递归法实现比较方便。

### § 8.5.2 遍历操作

下面以深度优先遍历操作说明广义表的遍历的实现。至于广度优先遍历，留作练习。广义表的深度优先遍历的含义与图或树的深度优先遍历类似。对型如

$A(A_1, A_2, \dots, A_n)$

的广义表，遍历过程为：

```
PreTraversal(A)
{
    for (i=1; i<=n; i++)
        if (Ai 未访问)
```

```

    if ( $A_i$  是单元素)
    {
        访问  $A_i$ ;
        为  $A_i$  置已访问标志;
    }
    else PreTraversal( $A_i$ );
}

```

下面是具体的程序。

```

long TGList::Cluster(TGListNode *pH, char **pE, long &k)
{//深度优先遍历以 pH 为头结点的广义表,
//将遍历到的元素结点 (单元素结点) 的值的地址存入一维数组 pE, 返回遍历到的单元素结点个数
TGListNode *q;

if (pH==NULL) return 0;

q = pH;
while (q!=NULL)
{
    if (q->visited==0) //若 q 未被访问
    {
        q->visited = 1; //置 q 已访问标志;
        if (q->tag==0)
            pE[k++]=q->name;
        else
            Cluster(q->pSub, pE, k); //遍历 q 的子表
    }
    q = q->next;
} //while
return k;
} //Cluster()

long TGList::Cluster(TGListNode *pH, char **pE)
{//该程序封装上面的带 k 参数的 Cluster(), 以简化接口
long k=0;
SetVisitTag(pH,0); //置各结点的访问标记为“未访问”
return Cluster(pH, pE,k);
}

```

### § 8.5.3 广义表统计计数

统计计数操作有求深度、求层号、求结点（元素）个数（包括求满足指定条件的结点个数）等。下面以广义表求深度操作说明。

广义表的深度定义为它的各直接结点中深度最大者的深度值加 1。这是个递归定义，所以用递归算法解决很自然。该问题的解决，实质上类似于遍历问题，下面是具体的程序。

```
long TGList::GetDepth(TGListNode0 *pH)
{ //求以 pH 为总头结点的广义表的深度，并返回其
  TGListNode *q;
  long dep, dep0;

  if (pH==NULL) return 0;

  dep=0;
  q = ((TGListNode *)pH)->pSub; //令 q 指向广义表 pH 的第一个元素结点
  while (q!=NULL) //依次求出 pH 的各直接元素结点的深度
  {
    if (q->tag==0) dep0=0; //单元素结点的深度定义为 0
    else dep0=GetDepth(q); //求出以 q 为总头结点的广义表（即 q 对应的子表）的深度
    if (dep < dep0) dep =dep0; //保留最大深度
    q = q->next; //令 q 指向 pH 的下个直接元素结点
  } //while
  return dep+1; //pH 的深度等于它的各直接结点的最大深度加 1
}
```

### § 8.5.4 广义表的串行化与逆串行化

广义表的串行化是指根据广义表（内存）结构，输出对应的广义表表达式。这种串行化可以通过改造广义表遍历算法实现，具体做法留作练习。这里我们只介绍逆串行化，即根据广义表表达式，创建（内存结构）广义表。

我们假定广义表表达式是带名字的，即每个表（子表）前都有一个名字。为了简化问题，突出主题，我们这里规定，广义表表达式中，每个成分是一个字符，各成分间没有间隔（空格），广义表名字用大写字母，单元素用小写字母。此外，还假设表达式无错误。在实际使用中，一般需对表达式进行预处理，使其符合这里的假设。例如，下面的表达式就是一个满足我们这里规定的广义表表达式：

A(a, B(b,C(d)),e,F(G(h,i)))

下面考虑实现方法。首先，我们采用从左到右依次读取表达式中每个成分（符号）



的方式。显然,当读到的是表名,则创建该表的头结点;如果读到的是单元素,则创建元素结点。每创建一个结点,就需要将它链到相应的单链表中(next 链,链接同层中各结点),此外,若创建的结点是某子表中第一个结点,则还要令该子表的头结点的 pSub 指向它。为此,结点创建后,要记录到容易访问到的地方。显然,记录的次序和访问次序相反,所以应该使用栈来记录,即每创建一个结点,就将其压入栈中。

那么,如何控制栈的进出?对新创建的结点,不管是元素结点还是头结点,均进栈,当读到逗号或左括号时,表示最前面一个结点已完全处理完,所以最前面的结点应出栈。此时,最前面结点就在栈顶,所以可以直接出栈。当前出栈的结点(比如说 p),对下次创建的结点(比如说 q)而言,是同层中的前驱,所以,当下次创建结点时,应令 p 的 next 指向 q。下面是具体的程序。

```
TGListNode *TGList::GListExprToGListStruct(int *gListExpr, int n)
{
    //gListExpr: 一维数组,存放广义表表达式。
    //在 gListExpr 中,假定结点用单个英文字母表示,各符号间也不留空,表达式中无语法错误
    //返回所创建的广义表的头结点指针
    long k, top;
    TGListNode *h,*pre,*p, **s;
    char firstList;

    if (n<=3)return NULL; //表达式中成分数目不足 4 时,认为是空表
    if (gListExpr[0]<'A' || gListExpr[0]>'Z' || gListExpr[1]!='(') return NULL;
        //第一个字符不是大写字母时,或第二个字符不是左括号时,认为非法,返回空
    s = new TGListNode *[n]; //申请栈空间
    top=0; //栈顶指示器置 0,表示空栈.

    k=0;
    h=new TGListNode; //生成一个结点,用作广义表的总头结点
    h->tag=1;
    h->name[0]=gListExpr[k]; h->name[1]=0;
    h->visited = 0;

    pre=NULL;
    top++;s[top]=h; //总头结点进栈
    k++;
    firstList=1; //firstList 为"最左结点"标志,值为 1 时,表示表达式中上个符号是左括号,
        //当前的成分为某表 L 的最左结点,则 L 的 pSub 应该指向该结点

    while (top>0 && k<n-1)
```

```
{
    k++;
    switch (gListExpr[k])
    {
        case ',': //读到逗号或右括号时,出栈
        case ')':
            pre=s[top]; top--;
            firstList=0;
            break;
        case '(': //左括号后的结点, 是上个结点的第一个子表
            firstList=1;
            continue;
        default: //读到表名或单元素,创建新结点
            p = new TGLListNode;
            p->visited = 0;
            p->next=NULL;
            if (gListExpr[k]>='A' && gListExpr[k]<='Z') //规定子表名为大写字母
            {
                p->tag=1;
                p->name[0]=gListExpr[k];p->name[1]=0;
            }
            else //当前创建的结点是单元素
            {
                p->tag=0;
                p->name[0]=gListExpr[k];p->name[1]=0;
            }
            //如果当前创建的结点是所属表中第一个结点,则建立子表关系,即令最近生成的结点的 pSub 指向它
            if (firstList)
                s[top]->pSub=p;
            else
                if (pre!=NULL) pre->next = p; //建立 next 链,pre 为当前生成的结点的左兄弟的指针

            top++; s[top]=p; //新生成的结点进栈
            firstList=0;
        } //switch
    };
    head=h;
    return h; //返回所创建的表的头结点.
```

```
}//
```

### § 8.5.5 广义表的复制与求尾

广义表的复制，也是生成结构与内容完全相同的另一个表。该过程的实现，实质上也是一种深度优先遍历。具体程序如下。

```
TGListNode0 *TGList::Copy(TGListNode0 *pH, long &k)
{//复制以 pH 为头结点的表，返回新复制的表的头结点指针
    TGListNode *q, *p, *pH0, *pRear;

    if (pH==NULL) return 0;

    pRear=NULL;
    q = (TGListNode *)pH;
    while (q!=NULL)
    {
        if (q->visited==0) //若 q 未被访问
        {
            k++;
            p=new TGListNode; //新申请一个结点，作为 q 的复制品
            p->tag=q->tag;
            p->no=q->no;
            p->visited=0;
            p->next=NULL;
            strcpy(p->name, q->name);

            if (pRear!=NULL) pRear->next= p;
            else pH0=p;
            pRear=p;
            q->visited = 1; //置 q 已访问标志;
            if (q->tag==0)
                p->pElem=q->pElem;
            else
                p->pSub=(TGListNode *)Copy(q->pSub, k); //复制 p 的子表
        }
        q = q->next; //令 q 指向 pH 的同层中的下一个结点
    } //while
    return pH0;
```

```
}

```

在该程序的支持下，可方便实现求尾操作：

```
TGListNode0 *TGList::GetTail(TGListNode0 *pNode)
{ //返回以 pNode 为头结点的表（即 pNode 对应的子表）的尾（一个新表）的头结点指针
  TGListNode *p,*q;
  long k=0;

  if (pNode==NULL || pNode->tag==0 || ((TGListNode *)pNode)->pSub->next==NULL)
    return NULL;
  p=(TGListNode *)Copy(pNode,k); //复制以 pNode 为头结点的表，新表的头结点指针为 p
  q=p->pSub; //下面三句将 pNode 的复制品中的第一个元素删除
  p->pSub=q->next;
  delete q;

  return p;
}
```

## § 8.6 广义表结构的应用

广义表最适合描述递归结构和分层结构，如分层索引结构（如图书的目录、计算机操作系统中的文件目录结构）及其它一些多叉树形结构，下面用几个典型的例子说明。

### § 8.6.1 多元多项式的表示

多元多项式具有多个变元，例如下面是一个三元多项式

$$P(x_1, x_2, x_3) = x_1^5 x_2^3 x_3 + 2x_1^5 x_2^2 x_3^4 + 5x_1^5 x_2^3 x_3^3 + 3x_1 x_2^4 x_3^2 + x_2 x_3 + 100$$

用计算机处理这种多项式时，就要考虑选取合适的数据结构。一种直观的数据结构是仿照一元多项式的结构，即令多项式每项对应一个结点，结点应含一个系数域和  $m$  个（ $m$  为变元个数）指数域。但这种表示法有下列几个方面的缺陷：①结点的结构随多项式的变数元数目而变化，这造成了概念上的不美观，缺少通用性；②不适合某些应用要求的操作的实现，如某些操作要求将  $m$  个变元中某个变元视为主变元，其它变元视为常量，而在处理主变元的系数时，又将它视为一个  $(m-1)$  元多项式进行处理。这种情况最适合使用广义表结构。

对任意一个  $m$  元多项式，都可以任选一个变元（称为第 1 变元或主变元），按此变元合并同类项，使该  $m$  元多项式变为如下形式：

$$p_m(x_1 \dots x_m) = A_n x_1^{e_n} + A_{n-1} x_1^{e_{n-1}} + \dots + A_1 x_1^{e_1}$$

这里,  $e_n > e_{n-1} > \dots > e_1$ ,  $A_i$  为关于变元  $x_2 \dots x_m$  的  $(m-1)$  元多项式,  $i=1, 2, \dots, n$ ,

下步可对  $A_i$  选  $x_2$  为主变元进行类似的处理, 如此进行, 最后一回选  $x_m$  为主变元, 对各个以  $x_{m-1}$  为主变元的多项式进行同样的处理。

例如, 对前面给出的  $P_3(x_1, x_2, x_3)$  处理如下:

$$\begin{aligned} P(x_1, x_2, x_3) &= x_1^5 x_2^3 x_3 + 2x_1^5 x_2^2 x_3^4 + 5x_1^5 x_2^3 x_3^3 + 3x_1 x_2^4 x_3^2 + x_2 x_3 + 100 \\ &= (x_2^3 x_3 + 2x_2^2 x_3^4 + 5x_2^3 x_3^3) x_1^5 + (3x_2^4 x_3^2) x_1 + (x_2 x_3 + 100) \\ &= ((5x_3^3 + x_3) x_2^3 + (2x_3^4) x_2^2) x_1^5 + (3x_2^4 x_3^2) x_1 + (x_2 x_3 + 100) \end{aligned}$$

将上式中每一项用一个如下形式的二元组表示:

(系数, 指数)

其中, “系数”可能是一个多元多项式, 也可能是基本元素。并且将各二元组按指数降序排列, 则所成结构为广义表, 这就是  $m$  元多项式的广义表表示。

对上式有:

$$\begin{aligned} P &= ((A_3, 5), (A_2, 1), (A_1, 0)) && // \text{以 } x_1 \text{ 为主变量} \\ A_3 &= ((A_{32}, 3), (A_{31}, 2)) && // \text{以 } x_2 \text{ 为主变量 } A_3 = (5x_3^3 + x_3) x_2^3 + (2x_3^4) x_2^2 \\ A_2 &= ((A_{21}, 4)) && // \text{以 } x_2 \text{ 为主变量 } A_2 = (3x_3^2) x_2^4 \\ A_1 &= ((A_{11}, 1), (100, 0)) && // \text{以 } x_2 \text{ 为主变量 } A_1 = x_2 x_3 \\ A_{32} &= ((5, 3), (1, 1)) && // \text{以 } x_3 \text{ 为主变量 } A_{32} = 5x_3^3 + x_3 \\ A_{31} &= ((2, 4)) && // \text{以 } x_3 \text{ 为主变量 } A_{31} = 2x_3^4 \\ A_{21} &= ((3, 2)) && // \text{以 } x_3 \text{ 为主变量 } A_{21} = 3x_3^2 \\ A_{11} &= ((1, 1)) && // \text{以 } x_3 \text{ 为主变量 } A_{11} = x_3 \end{aligned}$$

从该例子看出, 用广义表表示具体对象, 关键是分层, 即以某种方式 (抓住某个因素) 先将对象组织为一个线性结构, 然后, 再按同样方式细化线性结构中各元素。

### § 8.6.2 层次结构的表示

$n$  叉树是一种层次结构, 故  $n$  叉树是一种广义表, 但反之未必, 因为  $n$  叉树的定义中排除了树结点的共享性。因此, 总可以用广义表表示任意  $n$  叉树。

例如, 我们在第一章给出的家族树就是一种层次结构。我们仍以它为例说明 (将它的图重列于图 8-0)。

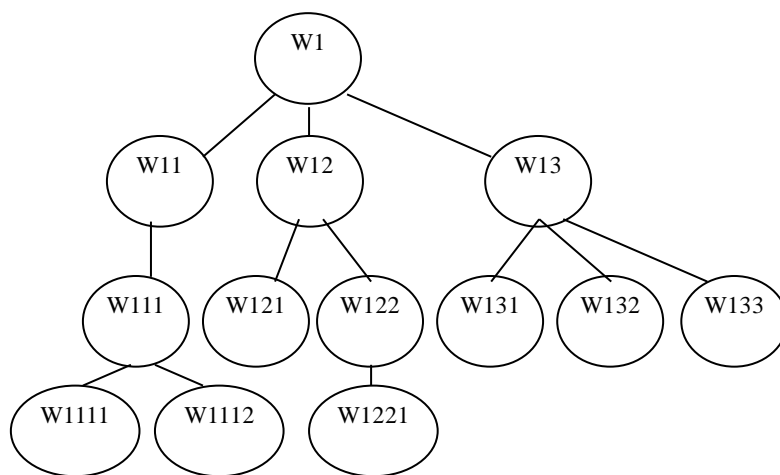


图 8-4 一个家族结构的树表示

对这种层次结构，我们采用下面的表示法：

对层次结构中任一结点  $X$  (设  $A$  为结点名称)，若  $X$  无下属结点 ( $X$  是单元素)，则将  $X$  的广义表示为：

$X$  .....当  $X$  不是最高层，

$(X)$  .....当  $X$  是最高层

若  $X$  有下属结点 ( $X$  是子表)，则  $X$  的广义表表示为 “

$X(X_1, X_2, \dots, X_n)$

这里， $X_1, X_2, \dots, X_n$  为  $X$  的  $n$  个下属结点 ( $n > 0$ ) 对应的广义表表示。

显然，这是个递归定义。

例如，对图 8-0，它的广义表表示为：

$W1(W11(W111(W1111, W1112)), W12(W121, W122(W1221)), W13(W131, W132, W133))$

层次结构是不含循环（递归）的。显然，若容许层次结构含循环（准层次结构），则它的广义表表示法仍可按上面的定义。

## 本章小结

广义表是一种很特殊的数据结构，从某一层的各元素的关系看，是线性表，但从分层结构看与树类似（无共享结点时是树，有共享结点但不同出现看作相同时为单纯的层次结构），所以从整体看，它仍然是一种受限的结构，不象图结构那样结点关系没有限制。此外，广义表的元素也很特别，每个元素的结构都可能不同，而且也可以是广义表。事

实上, 广义表中, 元素分两大类, 一类是单元素, 与其他数据结构下的元素类似, 另一类是非单元素, 为复合元素量, 即由单元素复合而成的元素。在其他数据结构中, 一般不把复合元素看作元素, 例如, 在树中, 子树本来也是复合元素, 但一般不认为是树中的元素。

求头和求尾是两项重要的操作, 通过这两个操作的配合, 可以访问到表中任意元素。如果结合表的联合运算( $\cup$ ), 就可以实现对表的重组和加工, 所以, 这两种操作往往被做为广义表的标准操作。

广义表的接口设计也很复杂。它兼有线性表、树、图的特点, 所以照顾这方方面面很困难。本章给出的设计, 就是力求兼顾这些方面。

广义表的存储结构是复杂的, 一般都采用链式结构。本章给出的分枝单链表是广义表的一种通用的存储方法, 完整地描述了广义表的主要特性。

## 习 题

1. 对下列广义表系列, 分别求出它们的长度、深度、头、尾, 并画出存贮结构图(同名元素或表共享, 小写字母为单元素)

$$\textcircled{1} A = ((a, b), (a, c))$$

$$\textcircled{2} c = ((A, a), (b, c))$$

$$\textcircled{3} D = ((a, (b, a)), A)$$

$$\textcircled{4} E = (E, a, b)$$

2. 请通过 GetHead() 和 GetTail() 操作, 将下列各表中的 apple 分离出来。

$$A1 = (\text{banana}, \text{pear}, \text{orange}, (\text{apple}))$$

$$A2 = ((\text{apple}, \text{pear}), \text{orange})$$

$$A2 = (\text{pear}, (\text{peach}, (\text{plum}, \text{apple}), \text{fig}), \text{apple}, \text{date})$$

3. 画出  $L = A(a, B(c, C(a, b), d), E(e, F(f)))$  的广义表图, 并分别给出其深度优先和广度优先遍历结果。

4. 给出一个 3 元多项式

$$P(x_1, x_2, x_3) = 10x_1^5x_2^2x_3 + 2x_1^5x_2x_3 + 3x_1^5 + x_1^2x_2^2x_3 + x_1^2x_2x_3 + x_2^2x_3 + 10$$

写出它的广义表表达式(逻辑表示), 并画出相应的分枝单链表存贮结构图。

5. 编写广义表的广度优先遍历程序。

6. 编写程序, 求广义表中任一指定结点的层号(结点的深度)。

7. 编写程序, 输出(内存结构)广义表的表达式。

8. 编写程序, 实现操作:

```
long Locate(char *pE, TGListNode0 **pNodes, TTraverseMode tm=PreOrder)=0;
```