高等学校计算机专业教材精选·算法与程序设计

新编数据结构算法考研指导

朱东生 赵建利 孙召伟 等编著

清华大学出版社

高等学校计算机专业教材精选·算法与程序设计

新编数据结构算法考研指导

朱东生 赵建利 孙召伟 等编著

清华大学出版社
北京

内 容 简 介

本书不是数据结构算法的简单赘述，而是以算法的功能为基础，对算法进行纵向分类，挖掘算法之间内在的联系，构建数据结构算法的统一体系，使考研同学顺利掌握算法设计要领。书中将数据结构知识分解为4类问题：递归、递归转非递归、回溯、技巧型算法，以及其他小概率特殊问题的算法。

本书突破以往的同类教程以线性表、栈和队列、串、数组和广义表、树、图、查找、排序给出各章的算法描述的讲述方式，避免了由于算法较多、难度较大，学生学习起来多会感觉烦琐、凌乱而迷茫。

本书采用实例教学法，在讲清基本知识点的基础上，尽量使用实例加以说明，因此书中包含了大量实用例子，绝大部分例子都给出了详细的分析过程及程序代码，代码短小精悍，容易理解。

因此学习本书将使读者对“数据结构”课程的理解产生质的飞跃。

图书在版编目（CIP）数据

新编数据结构算法考研指导/朱东生，赵建利，孙召伟等编著. —北京：清华大学出版社，2010.7
（高等学校计算机专业教材精选·算法与程序设计）
ISBN 978-7-302-22098-5

Ⅰ.①新… Ⅱ.①朱… ②赵… ③孙… Ⅲ.①数据结构—研究生—入学考试—自学参考资料 ②算法分析—研究生—入学考试—自学参考资料 ③C语言—程序设计—研究生—入学考试—自学参考资料 Ⅳ.①TP311.12

中国版本图书馆CIP数据核字(2010)第029027号

责任编辑：汪汉友
责任校对：时翠兰
责任印制：孟凡玉

出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦A座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62795954，jsjjc@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：北京市清华园胶印厂
经　　销：全国新华书店
开　　本：185×260　　**印　张**：11.25　　**字　数**：268千字
版　　次：2010年7月第1版　　**印　次**：2010年7月第1次印刷
印　　数：1～4000
定　　价：19.00元

产品编号：034434-01

出版说明

我国高等学校计算机教育近年来迅猛发展，应用所学计算机知识解决实际问题，已经成为当代大学生的必备能力。

时代的进步与社会的发展对高等学校计算机教育的质量提出了更高、更新的要求。现在，很多高等学校都在积极探索符合自身特点的教学模式，涌现出一大批非常优秀的精品课程。

为了适应社会的需求，满足计算机教育的发展需要，清华大学出版社在进行了大量调查研究的基础上，组织编写了《高等学校计算机专业教材精选》。本套教材从全国各高校的优秀计算机教材中精挑细选了一批很有代表性且特色鲜明的计算机精品教材，把作者们对各自所授计算机课程的独特理解和先进经验推荐给全国师生。

本系列教材特点如下。

(1) 编写目的明确。本套教材主要面向广大高校的计算机专业学生，使学生通过本套教材，学习计算机科学与技术方面的基本理论和基本知识，接受应用计算机解决实际问题的基本训练。

(2) 注重编写理念。本套教材作者群为各高校相应课程的主讲，有一定经验积累，且编写思路清晰，有独特的教学思路和指导思想，其教学经验具有推广价值。本套教材中不乏各类精品课配套教材，并力图努力把不同学校的教学特点反映到每本教材中。

(3) 理论知识与实践相结合。本套教材贯彻从实践中来到实践中去的原则，书中的许多必须掌握的理论都将结合实例来讲，同时注重培养学生分析问题、解决问题的能力，满足社会用人要求。

(4) 易教易用，合理适当。本套教材编写时注意结合教学实际的课时数，把握教材的篇幅。同时，对一些知识点按教育部教学指导委员会的最新精神进行合理取舍与难易控制。

(5) 注重教材的立体化配套。大多数教材都将配套教师用课件、习题及其解答，学生上机实验指导、教学网站等辅助教学资源，方便教学。

随着本套教材陆续出版，我们相信它能够得到广大读者的认可和支持，为我国计算机教材建设及计算机教学水平的提高，为计算机教育事业的发展做出应有的贡献。

清华大学出版社

前　言

自2008年全国硕士研究生入学考试起，计算机专业综合课被列为全国统考课程，满分150分，其中“数据结构”课程占45分。在这45分当中，算法设计的题目占25分之多，这也是大多数考研同学最没有把握拿到手的。以往的“数据结构”教程及辅导丛书，多以线性表、栈和队列、串、数组和广义表、树、图、查找、排序等为模板给出各章的算法描述，由于算法较多、难度较大，学生学习起来多会感觉烦琐、凌乱和迷茫。本书不是以往“数据结构”算法的简单赘述，而是以算法的功能为基础，对算法进行纵向分类，挖掘算法之间内在的联系，构建数据结构算法的统一体系，使考研同学顺利掌握算法设计要领。

本书是作者在拜读了严蔚敏、李春葆、徐孝凯等先生的相关著作的基础上，结合自己多年教学体会写成的。

作者认为：数据结构虽然有4种关系，但实质上数据的关系只是前驱、后继的顺序关系，即数据结构所研究的关系是单纯的，因此4种关系之间必然存在某种相同性。而且数据结构的物理结构，有人认为也是4类，但用得比较多的是2类：顺序结构和链式结构。而且，几乎所有数据结构都能用链式结构描述。这就更使得4类数据结构的表示方法具有很大的一致性。

所有的线性关系，都可以理解成由第1个元素和剩余的元素（除去第1个元素）组成，而剩余的元素又是一个相对较小的线性关系。对于顺序结构一般认为由最后1个元素与前$n-1$个元素组成。

对于广义表，其本身就是线性表的拓展，只是其元素既可是单元素，也可是广义表。既可把它理解成n个广义表，也可理解成由第1个元素和剩余元素（除去第1个元素）的广义表组成。

树是由根及其子树组成。

图，可以理解成由某一顶点（即起始点，任意顶点都可作为起始点）及所有后继结点为起始点形成的子图所组成。

综上所述，数据结构可概括成一句话：它们都是由某一个元素和所有后继构成的相同的数据结构所组成。这说明，数据结构的定义基本上是递归的，运用递归实现算法简单明了。因此，数据结构中用递归实现的算法占据很大篇幅，我们就将“递归算法”安排在了本书的第1章。当然递归所能解决的问题不止数据结构中的各个算法。在第1章中，读者将会学习到递归算法的组成部分、一般设计模型及递归算法的分类。

递归算法解题过程中涉及的堆栈操作是很大的开销，系统为每层的返回点、局部量等开辟了栈来存储，每次都需要保存当前函数的所有信息，算法运行效率较低。在递归调用的过程当中，递归次数过多容易造成栈溢出等问题。为了避免这样的情况，人们想到了用算法实现递归到非递归的转换。从递归到非递归的转换过程中可能会涉及有关栈、队列等数据结

构的综合应用，这也是其吸引出题老师的一个地方，因此曾经是不少学校研究生入学考试的考点。第2章读者就会学习到如何将递归算法转换为非递归算法，书中根据递归算法的分类给出了若干个转换模型，简单实用。同样，数据结构中的算法也可直接用非递归的方法给予解决。

回溯是一类特殊的递归问题，但它又有别于一般的递归算法，初学者往往搞不清回溯和递归到底有什么区别，在实际问题中该用递归呢、还是该用回溯。本书第3章，读者将会学习到回溯法的应用场合、回溯法的一般模型以及回溯和递归的主要区别。

本书第4章是技巧型算法。数据结构算法试题部分一般包括：数据结构的基本操作以及对基本操作的应用(变异、扩充、组合)。前者其他数据结构的书籍都做过详细介绍，本书不做过多讨论。后者笔者称为技巧。读者将会在这章学习到在什么样的情境下会用到哪个数据结构的哪些基本操作，以及如何应用。第4章的内容是考试的重点。

还有一类算法设计技术，适用范围仅限于特殊的问题，而算法本身理解起来很费时，且算法的可推广性一般，我们认为属于小概率事件。这些问题都被收录在第5章，特殊问题的算法。

本书采用实例教学法，在讲清基本知识点的基础上，尽量使用实例加以说明，因此书中包含了大量实用例子，绝大部分例子都给出了详细的分析过程及程序代码，代码短小精悍，容易理解。因此学习本书将使读者对数据结构的理解产生质的飞跃。

本书由朱东生组织编写，朱东生、赵建利、孙召伟编写了本书主要章节。许研、胡春叶、张艳格、董雪梅、韩红根也参加了本书编写、整理等工作。

由于水平所限，加之时间仓促，本书中仍难免出现错误和缺点，恳切希望得到广大读者特别是讲授此课程的老师们的批评和指正。

作　者

2010年6月

目　录

第1章 递 归

程序调用自身的编程技巧称为递归(recursion)。它是一个过程或函数在其定义或说明中又直接或间接调用自身的一种方法。它通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解。如果原问题可以划分成若干子问题,而这些子问题,要么有解,要么能用相同的方法划分成小的子问题,而这些小的子问题能够继续划分成更小的子问题,一直划分到最小问题,并且最小问题有解,在这种情况下就可以用递归。递归以其思路明确、代码简洁在数据结构算法中、特别是在解决树、图等问题时得到了广泛应用。

但数据结构中递归算法纷繁复杂、且其实现过程对用户透明,这使得对递归算法的理解变得无比困难,有时甚至难以掌握。

笔者认为,数据结构虽然有4种关系(集合关系、线性关系、树形关系、图形关系),但实质上数据的关系只是前驱、后继的顺序关系,说明所研究的关系是单纯的,4种关系之间必然存在某种相同性。而且其物理结构,有人认为也是4类,但用得比较多的是两类:顺序结构和链式结构,而几乎所有数据结构都能用链式结构描述,更使得它们的表示方法具有很大的一致性。这就表示五大数据结构在逻辑上可以有统一的表现形式,即它们的递归本质定义。

1.1 数据结构的递归本质分析

所有的线性关系,可以理解成由第1个元素和剩余的元素(除去第1个元素)组成,而剩余的元素又是一个相对较小的线性关系,如图1.1所示。对于顺序结构,一般认为由最后一个元素与前 $n-1$ 个元素组成。

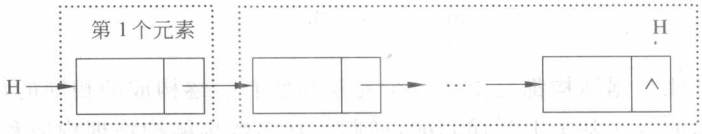

图1.1 线性表的递归结构

对于广义表,它本身就是线性表的拓展,只是其元素既可是单元素,也可是广义表。既可把它理解成 n 个广义表,也可理解成由第一个元素和剩余元素(除去第一个元素)的广义表组成。如图1.2所示,广义表GH中的第一个元素是一个广义表,除第一个元素外的剩余元素又构成了一个新的广义表。

树是由根和除根外的若干棵子树组成,如图1.3所示。

图可以理解成由某一顶点(即起始点,任意顶点都可作为起始点)及所有后继结点为起始点形成的子图所组成,如图1.4所示。

集合可以看作是由其中某一元素及除这一元素外其他元素组成的集合构成。

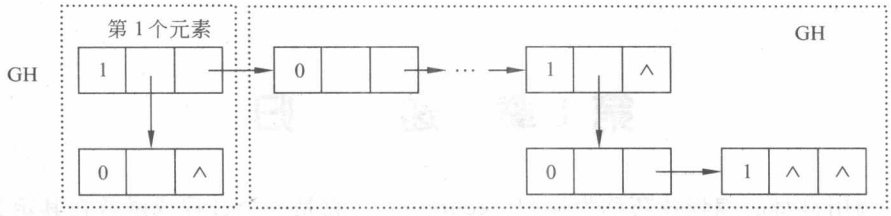

图 1.2　广义表的递归结构

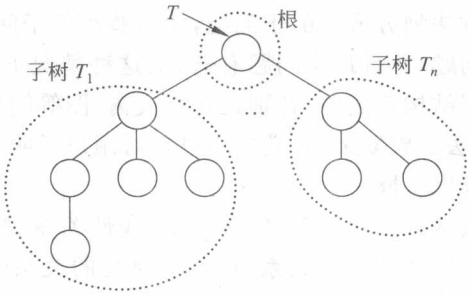

图 1.3　树的递归结构

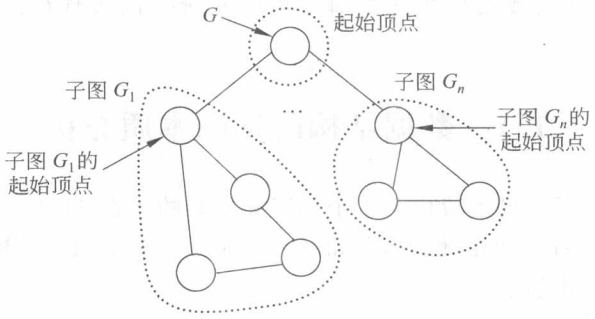

图 1.4　图的递归结构

综上所述，这些数据结构都是由某一个元素和所有后继构成的相同的数据结构所组成。这说明，数据结构的定义基本上是递归的，递归是众多数据结构物理构造和逻辑意义上的统一体。

1.2　数据结构定义

为了后面讨论算法方便，本节给出了部分数据结构的 C++ 语言描述。

```
typedef int ElemType;
const int MaxSize=32767;
const int Mininum=-32768;      //表示最小的数
```

(1) 顺序表。

```
typedef struct List
{
    ElemType list[MaxSize];
    int size;
}SqList;
```

(2) 单链表结点。

```
typedef struct node
{
    ElemType data;
    struct node * next;
}Node;
```

(3) 广义表。

```
enum Boolean{false, true};
struct GLNode
{
    Boolean tag;
    Union{
        ElemType data;
        GLNode * sublist;
    };
    GLNode * next;
};
```

(4) 二叉树。

```
typedef struct BTreeNode
{
    ElemType data;
    struct BTreeNode * left;
    struct BTreeNode * right;
}BTreeNode;
```

(5) AVL 树(平衡树)。

```
typedef struct aTreeNode
{
    ElemType data;
    int bf;                          //平衡因子
    struct BTreeNode * left;
    struct BTreeNode * right;
}AVLTreeNode;
```

(6) 图。

① 临接矩阵。

```
typedef ElemType vexlist[MaxSize];                    //存储顶点信息的数组类型
typedef int adjmaxtrix[MaxSize][MaxSize];             //存储邻接矩阵的数组类型
```

② 临接表。

```
struct edgenode
{                                  //定义邻接表中的边结点类型
    int adjvex;                    //邻接点域
    int weight;                    //权值域,无权图可省去
    edgenode * next;               //指向下一个边结点的链域
}
typedef edgenode * adjlist[MaxSize];
```

1.3 递归算法模型设计

1.3.1 递归算法一般形式

递归算法由两部分组成：

(1) 最小问题,称为基本项；

(2) 原问题与子问题的关系,称为递归项(递归项包含两部分内容,一是可以继续分解的子问题;二是不能继续划分的子问题,称为有解子问题)。一般的表现形式如下：

形式 1：如果最小问题存在行为。

```
if  (最小问题)
    基本项行为;
else
{
    有解子问题的处理;
    可继续划分的子问题 1, 2, …, n;
}
```

形式 2 ：如果最小问题什么都不做。

```
if  (!最小问题)
{
    有解子问题的处理;
    可继续划分的子问题 1, 2, …, n;
}
```

最小问题决定了递归的出口,有时最小问题可能会有多种情况,那就要分情况讨论每种情况时的基本项是什么,体现在算法描述上就是一个多分支的 if 语句。

不能继续划分的子问题(即有解子问题)是按照问题的要求被处理,比如输出、比较大小等。研究递归算法,主要是研究有解子问题,线性表处理的是第一个元素,树处理的是树根,广义表处理的是第一个结点,图处理的是起始顶点。因此处理有解子问题是“数据结构”众多递归算法的区别所在,是本书区别于其他文献的主要特点,这也是学习递归算法的难点。

至于可继续划分的子问题，则被写成了同样形式的递归调用语句，且有几个这样的子问题就有几条递归语句，体现了子问题与原问题的处理方式一样，只是问题规模变小了。一般有三种情况的划分。

① 可划分成一个子问题的：线性表、二叉排序树的查找。

② 可划分成二个子问题的：广义表、二叉树。

③ 可划分成多个子问题的：图。

根据递归项中有解子问题的处理顺序，将递归分为前序递归、中序递归和后序递归。下面以形式 2 为例给出三种递归算法的解题模型。

1.3.2 前序递归

1. 基本形式

```
if  (!最小问题)
{
    有解子问题的处理;
    可继续划分的子问题 1;
    可继续划分的子问题 2;
    ⋮
    可继续划分的子问题 n;
}
```

如图 1.5 所示，递归项中先处理有解子问题，再处理可继续划分的子问题。因为人们总是习惯先解决有解的子问题，再解决可继续划分的子问题，和五大数据结构相关的大多数的递归问题都是前序递归。

通过上面的分析可以看到，无论算法的功能如何变化，同种数据结构递归算法的解题模型都是一样的，不同的只是对第一个元素的处理方式不同。下面是形式 2 下与数据结构相关的前序递归模型。

(1) 线性表。

```
Algorithm1: Linear_list_Recursion(H: Linear_list )
{
    if (H!=NULL)
    {
        Process H;                    //根据算法的具体功能,处理线性表的第一个元素
        Linear_list_Recursion(H-> next);
                                      //递归处理除第一个元素之外的其他元素组成的线性表
    }
}
```

(2) 广义表。

```
Algorithm2:Generalized_list_Recursion(GH: Generalized_list )
{
    if (GH!=NULL)
    {  //处理第一个元素,因为有单元素和子表两种可能,所以要分情况进行讨论
```

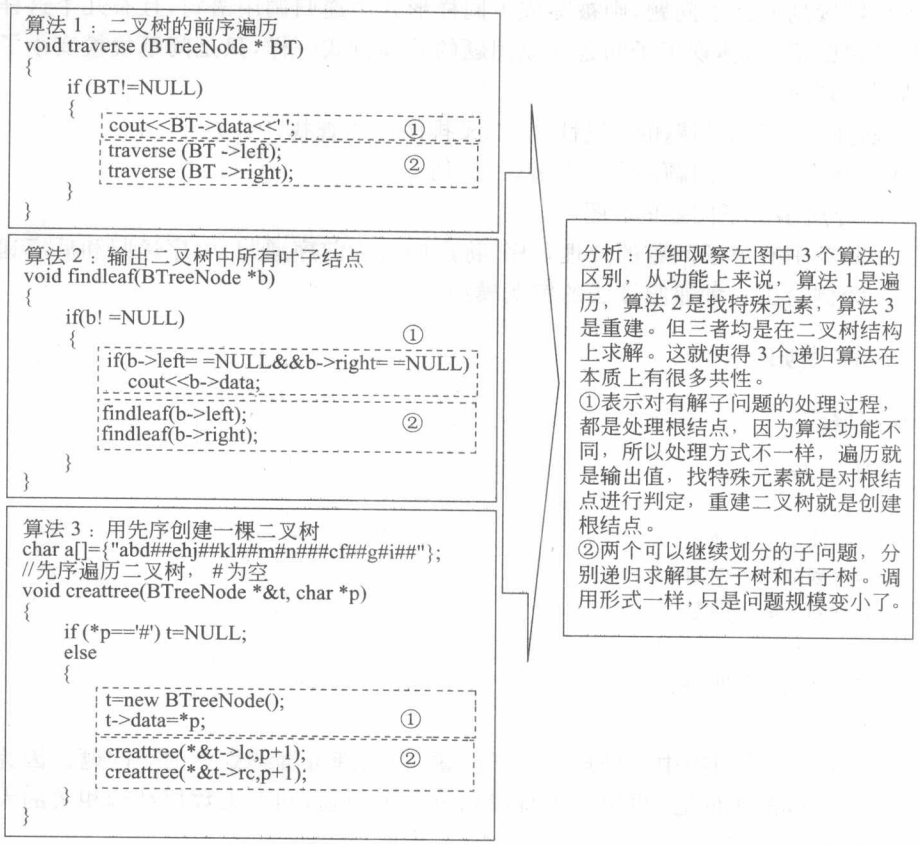

图 1.5　前序递归算法分析

```
    if (GH->tag==false)
        Process GH; //如果该元素为单元素时,则根据算法的具体功能,处理这个元素
    else
        Generalized_list_Recursion(GH-> sublist); //该元素为子表,则递归处理
    Generalized_list_Recursion (GH-> next);
    //递归处理除第一个元素之外的其他元素组成的广义表
}
```

(3) 树。

```
Algorithm3:Tree_Recursion(T: TreeNode )
{
    if (T!=NULL)
    {
        Process T;                              //根据算法的具体功能,处理树的根结点
        Tree_Recursion (T-> firstsubtree);      //递归处理以树的第一棵子树
        Tree_Recursion (T-> secondsubtree);     //递归处理以树的第二棵子树
        …
        Tree_Recursion (T-> lastsubtree);       //递归处理以树的第 n 棵子树
```

```
    }
}
```

(4) 图。

```
Algorithm4:Graph_Recursion(G: GraphNode , int i)
{                                  //G 表示图,i 表示起始结点编号
    Process G;                     //根据算法的具体功能,处理图的起始结点
    visited[i]=true;               //标识 i 结点已被访问
    while (G 的邻结点!=NULL)
    {
        if (G 的第一个邻接点没有被访问过)
            Graph_Recursion (G 的第一个邻结点, 第一个邻接点的编号);
        G=G-> next;
    }
}
```

说明：

(1) 树结构与图结构的递归算法模型是不一样的。

其一，树结构中罗列了若干个对子树访问的递归调用，体现了有几棵子树就有几个递归语句的特点；而图结构中则是用一个 while 循环来依次访问当前结点的所有临接点。这是因为图的邻接点有统一的描述方式，所以可用一个 while 循环来处理。而树结构中，树根的孩子是用不同的名称分别进行描述的，如二叉树要分左孩子和右孩子，所以处理时要用列表方式依次写出对每个孩子的处理过程(即递归调用)。如果树结构中所有孩子的描述方法一致了，也可用一个 while 循环来替代列表方式，即和图结构的递归模型就相似了。

其二，图结构中有判定某结点是否被访问过的语句("if (*G* 的第一个邻接点没有被访问过)")，而树结构中没有。这是由于树结构和图结构中结点的逻辑关系不同而造成的。树中结点之间是一对多的关系，从某结点到根只有唯一的一条路；而图结构中，结点之间是多对多的关系，从起始结点到当前结点可能会有多条路，所以要判定是否已经访问过了。

(2) 上面只是给出抽象模型，真正在实现时要根据数据结构定义时的数据类型进行局部调整。处理可继续划分子问题(即递归调用)时，入口参数也要根据实际情况进行设计。

2. 分类

(1) 遍历。之所以将遍历放在第一类，是因为数据结构中许多问题的解决都依赖于遍历。即要按照某种顺序依次访问结构中的每一个元素，访问方式的不同就演变成了不同的问题。例如：求和、比大小、找特殊元素等。

【例 1.1】 有一个不带表头结点的单链表，设计如下递归算法：正向显示以 h 为头指针的单链表的所有结点值。

```
void traverse(Node * h)
{
    if(h!=NULL)
    {
        cout<<h-> data;
        traverse(h-> next);
    }
}
```

【例 1.2】 假设二叉树采用二叉链存储结构存储，前序遍历该二叉树。

```
void traverse (BTreeNode * BT)
{
    if (BT!=NULL)
    {
        cout<<BT->data<<'';
        traverse (BT -> left);
        traverse (BT -> right);
    }
}
```

【例 1.3】 打印广义表所有原子结点上的数据域。

分析：对于广义表的递归结构有两种认识，所以对应的算法就有两种表现形式。

① 一个广义表由第一个元素(可能是广义表)和除第一个元素以外的其他元素组成的广义表构成。

```
if (h->tag==true)
    递归访问子表                          //类似二叉树
else
    递归访问下一个结点                    //类似线性表
```

```
PreOrder(GLNode * h)
{
    if(h!=NULL)
    {
        if(h->tag==false)                 //为原子结点时
            cout<<h->data;
        else
            PreOrder(h->sublist);         //为子表时
        PreOrder(h->next);
    }
}
```

② 一个广义表由 n 个子表构成。

```
while (h!=NULL)
{
    递归语句;
    h=h->next;
}
```

```
PreOrder(GLNode * h)
{
    while (h!=NULL)
    {
        if(h->tag==false)                    //为原子结点时
            cout<<h->data;
        else
            PreOrder(h->sublist);            //为子表时
        h=h->next;
    }
}
```

今后再讨论到有关广义表的算法时，可能都有这两种形式的解决方案，到时就不再重复说明。

【例 1.4】 图的深度优先搜索遍历。(采用临接表存储图)。

```
void DFS(adjlist GL, int i, int n)
{
    cout<<i<<' ';
    visited[i]=true;
    edgenode * p=GL[i];
    while (p!=NULL)
    {
        int j=p->adjvex;
        if (!visited[j])
            DFS(GL,j,n);
        p=p->next;
    }
}
```

细心的读者会发现例 1.4 的算法与例 1.3 的②几乎是一样的，正是因为它们的数据结构在定义上的递归本质，使得解决方案几乎相同。

(2) 找特殊元素。

【例 1.5】 设计一个算法 change(* h,s,t)，将一个广义表 h 中的所有原子 s 替换成 t。

例如，change("(a,(a,b),((a,b),c))",'a','x')，返回的结果为“(x,(x,b),((x,b),c))”。