第5章 递归与广义表

**一、复习要点**

本章主要讨论递归过程和广义表。一个递归的定义可以用递归的过程计算，一个递归的数据结构可以用递归的过程实现它的各种操作，一个递归问题也可以用递归的过程求解。因此，递归算法的设计是必须掌握的基本功。递归算法的一般形式：

**void** p ( 参数表 ) **{**

**if** ( 递归结束条件)

可直接求解步骤； 基本项

**else** p ( 较小的参数 )**;** 归纳项

**}**

在设计递归算法时，可以先考虑在什么条件下可以直接求解。如果可以直接求解，考虑求解的步骤，设计基本项；如果不能直接求解，考虑是否可以把问题规模缩小求解，设计归纳项，从而给出递归求解的算法。必须通过多个递归过程的事例，理解递归。但需要说明的是，递归过程在时间方面是低效的。

广义表是一种表，它的特点是允许表中套表。因此，它不一定是线性结构。它可以是复杂的非线性结构，甚至允许递归。可以用多重链表定义广义表。在讨论广义表时，特别注意递归在广义表操作实现中的应用。

本章复习的要点：

**1、**基本知识点

要求理解递归的概念：什么是递归？递归的定义、递归的数据结构、递归问题以及递归问题的递归求解方法。理解递归过程的机制与利用递归工作栈实现递归的方法。通过迷宫问题，理解递归解法，从而掌握利用栈如何实现递归问题的非递归解法。在广义表方面，要求理解广义表的概念，广义表的几个性质，用图表示广义表的方法，广义表操作的使用，广义表存储结构的实现，广义表的访问算法，以及广义表的递归算法。

**2、**算法设计

⮚ 求解汉诺塔问题，掌握分治法的解题思路。

⮚ 求解迷宫问题、八皇后问题，掌握回溯法的解题思路。

⮚ 对比单链表的递归解法和非递归解法，掌握单向递归问题的迭代解法。

⮚ 计算广义表结点个数，广义表深度，广义表长度的递归算法。

⮚ 输出广义表各个原子所在深度的非递归算法。

⮚ 判断两个广义表相等的递归算法。

⮚ 广义表的按深度方向遍历和按层次（广度）方向遍历的递归算法。

⮚ 使用栈的广义表的按深度方向遍历的非递归算法。

⮚ 递归的广义表的删除算法

**二、难点与重点**

1、递归：递归的定义、递归的数据结构、递归问题用递归过程求解

⮚ 链表是递归的数据结构，可用递归过程求解有关链表的问题

2、递归实现时栈的应用

⮚ 递归的分层(树形)表示：递归树

⮚ 递归深度(递归树的深度)与递归工作栈的关系

⮚ 单向递归与尾递归的迭代实现

3、广义表：广义表定义、长度、深度、表头、表尾

* 用图形表示广义表的存储结构
* 广义表的递归算法，包括复制、求深度、求长度、删除等算法

**三、教材中习题的解析**

5-1 已知A[n]为整数数组，试写出实现下列运算的递归算法：

(1) 求数组A中的最大整数。

(2) 求n个整数的和。

(3) 求n个整数的平均值。

【解答】

**#include** <iostream.h>**class** RecurveArray **{** //数组类声明**private: int** \*Elements**;** //数组指针 **int** ArraySize**;** //数组尺寸

**int** CurrentSize**;** //当前已有数组元素个数**public :**

RecurveArray ( **int** MaxSize =10 ) **:**

ArraySize ( MaxSize )**,** Elements ( **new int**[MaxSize] )**{ }** ~RecurveArray ( ) **{ delete** [ ] Elements**; }** **void** InputArray()**;** //输入数组的内容 **int** MaxKey ( **int** n )**;** //求最大值 **int** Sum ( **int** n )**;** //求数组元素之和

**float** Average ( **int** n )**;** //求数组元素的平均值

**};**

**void** RecurveArray **::** InputArray ( )**{**  //输入数组的内容

**cout** << "Input the number of Array: \n"**;**

**for** ( **int** i = 0**;** i < ArraySize**;** i++ ) **cin** >> Elements[i]**;**

**}**

**int** RecurveArray **::** MaxKey ( **int** n ) **{** //递归求最大值

**if** ( n == 1 ) **return** Elements[0]**;**

**int** temp = MaxKey ( n - 1 )**;**

**if** ( Elements[n-1] > temp ) **return** Elements[n-1]**;**

**else return** temp**;**

**}**

**int** RecurveArray **::** Sum ( **int** n ) **{**  //递归求数组之和

**if** ( n == 1) **return** Elements[0]**;**

**else return** Elements[n-1] + Sum (n-1)**;**

**}**

**float** RecurveArray **::** Average ( **int** n ) **{** //递归求数组的平均值

**if** ( n == 1) **return** (**float**) Elements[0]**;**

**else return** ( (**float**) Elements[n-1] + ( n - 1) \* Average ( n - 1 ) ) / n**;**

**}**

**int** main ( **int** argc**, char**\* argv [ ] ) **{ int** size = -1**;**

**cout** << "No. of the Elements : "**;**

**while** ( size < 1 ) **cin** >> size**;**

RecurveArray ra ( size )**;**

ra.InputArray();

**cout**<< "\nThe max is: " << ra.MaxKey ( ra.MaxSize ) **<< endl;**

**cout**<< "\nThe sum is: " << ra.Sum ( ra.MaxSize ) << **endl;**

**cout**<< "\nthe avr is: " << ra.Average ( ra.MaxSize ) << **endl;**

**return** 0**;**

**}**

5-2 已知Ackerman函数定义如下：



(1) 根据定义，写出它的递归求解算法；

(2) 利用栈，写出它的非递归求解算法。

【解答】

(1) 已知函数本身是递归定义的，所以可以用递归算法来解决：

**unsigned** akm ( **unsigned** m, **unsigned** n ) **{**

**if** ( m == 0 ) **return** n+1**;** // m == 0

**else if** ( n == 0 ) **return** akm ( m-1, 1 )**;** // m > 0, n == 0

**else return** akm ( m-1, akm ( m, n-1 ) )**;** // m > 0, n > 0

**}**

(2) 为了将递归算法改成非递归算法，首先改写原来的递归算法，将递归语句从结构中独立出来：

**unsigned** akm ( **unsigned** m, **unsigned** n ) **{**

**unsigned** v**;**

**if** ( m == 0 ) **return** n+1**;** // m == 0

**if** ( n == 0 ) **return** akm ( m-1, 1 )**;** // m > 0, n ==0

v = akm ( m, n-1 ) )**;** // m > 0, n > 0

**return** akm ( m-1, v )**;**

**}**

计算akm(2, 1)的递归调用树如图所示：

akm(2, 1)

akm = 5

v = 3

akm = 5

akm(1, 3)

v =akm(2, 0)

v = 4

akm = 3

akm(0, 4) = 5

v = akm(1, 2)

akm(1, 1)

v = 3

akm(0, 3) = 4

v *=* akm(1, 1)

akm(0, 2) = 3

v =akm(1, 0)

v = 2

akm(0, 2) = 3

akm(0, 1) = 2

*v =* akm(1, 0)

v = 2

akm(0, 1) = 2

用到一个栈记忆每次递归调用时的实参值，每个结点两个域**{**vm, vn**}**。对以上实例，栈的变化如下：

vm vn vm vn vm vn vm vn vm vn vm vn

1 0 0 1

2 0 1 1 1 1 1 1 0 2

改akm(m-1,1) 改akm(m-1,1) v = n+1= 2 改akm(m-1,v) 改akm(m-1,v)

v = n+1 = 3

2 1 2 1 2 1 2 1 2 1 1 3

vm vn vm vn vm vn vm vn vm vn vm vn

1 1 1 1 0 2

1 0 0 1

1 2 1 2 1 2 0 3

1 3 1 3 1 3 1 3 0 4

改akm(m-1,1) v = n+1 = 2 改akm(m-1,v) 改akm(m-1,v) 改akm(m-1,v) 栈空, 返回v = 5

v = n+1 = 3 v = n+1 = 4 v = n+1 = 5

相应算法如下

#**include** <iostream.h>

#**include** “stack.h”

#**define** maxSize 3500**;**

**unsigned** akm ( **unsigned** m, **unsigned** n ) **{**

**struct** node **{ unsigned** vm, vn**; }**

stack<node> st ( maxSize )**;**  node w**; unsigned** v**;**

w.vm = m**;** w.vn = n**;** st.Push (w)**;**

**do {**

**while** ( st.GetTop( ).vm > 0 ) **{** //计算akm(m-1, akm(m, n-1) )

**while** ( st.GetTop( ).vn > 0 ) //计算akm(m, n-1), 直到akm(m, 0)

**{** w.vn--**;** st.Push( w )**;** **}**

w = st.GetTop( )**;** st.Pop( )**;**  //计算akm(m-1, 1)

w.vm--**;** w.vn = 1**;** st.Push( w )**;**

**}** //直到akm( 0, akm( 1, \* ) )

w = st.GetTop()**;** st.Pop( )**;** v = w.vn++**;** //计算v = akm( 1, \* )+1

**if** ( st.IsEmpty( ) == 0 ) //如果栈不空, 改栈顶为( m-1, v )

**{** w = st.GetTop()**;** st.Pop( )**;** w.vm--**;** w.vn = v**;**  st.Push( w )**; }**

**} while** ( st.IsEmpty( ) == 0 );

**return** v**;**

**}**

5-3 【背包问题】设有一个背包可以放入的物品的重量为s，现有n件物品，重量分别为w[1], w[2], …, w[n]。问能否从这n件物品中选择若干件放入此背包中，使得放入的重量之和正好为s。如果存在一种符合上述要求的选择，则称此背包问题有解(或称其解为真)；否则称此背包问题无解(或称其解为假)。试用递归方法设计求解背包问题的算法。（提示：此背包问题的递归定义如下：）



【解答】

根据递归定义，可以写出递归的算法。

**enum** boolean **{** False, True **}**

boolean Knap( **int** s, **int** n ) **{**

**if** ( s == 0 ) **return** True**;**

**if** ( s < 0 || s > 0 **&&** n < 1 ) **return** False**;**

**if** ( Knap ( s – W[n], n-1 ) == True ) **{ cout** << W[n] << ‘ , ’**; return** True**; }**

**return** Knap( s, n-1 )**;**

**}**

若设w = **{** 0, 1, 2, 4, 8, 16, 32 **}**，s = 51, n = 6。则递归执行过程如下

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 递归 | Knap(51, 6) |  |  |  |  | **return** True, 完成 |
| Knap(51-32, 5) |  |  |  |  | **return** True, 打印32 |
| Knap(19-16, 4) |  |  |  |  | **return** True, 打印16 |
| Knap(3-8, 3) | **return** False | Knap(3, 3) |  |  | **return** True, 无动作 |
| s = -5 < 0 | **return** False | Knap(3-4, 4) | **return** False | Knap(3, 2) | **return** True, 无动作 |
|  |  | s = -1 < 0 | **return** False | Knap(3-2, 1) | **return** True, 打印2 |
|  |  |  |  | Knap(1-1, 0) | **return** True, 打印1 |
|  |  |  |  | s = 0 | **return** True |

5-4 【八皇后问题】设在初始状态下在国际象棋棋盘上没有任何棋子(皇后)。然后顺序在第1行，第2行，…。第8行上布放棋子。在每一行中有8个可选择位置，但在任一时刻，棋盘的合法布局都必须满足3个限制条件，即任何两个棋子不得放在棋盘上的同一行、或者同一列、或者同一斜线上。试编写一个递归算法，求解并输出此问题的所有合法布局。(提示：用回溯法。在第n行第j列安放一个棋子时，需要记录在行方向、列方向、正斜线方向、反斜线方向的安放状态，若当前布局合法，可向下一行递归求解，否则可移走这个棋子，恢复安放该棋子前的状态，试探本行的第j+1列。)

【解答】

此为典型的回溯法问题。

1# 次对角线

0# 次对角线

0 1 2 3

2# 次对角线

3# 次对角线

➊

4# 次对角线

➋

5# 次对角线

6# 次对角线

➍

➌

0# 主对角线

1# 主对角线

0

1

2

3

3# 主对角线

2# 主对角线

4# 主对角线

6# 主对角线

5# 主对角线

## 在解决8皇后时，采用回溯法。在安放第 i 行皇后时，需要在列的方向从 1 到 n 试探( j = 1, …, n )：首先在第 j 列安放一个皇后，如果在列、主对角线、次对角线方向有其它皇后，则出现攻击，撤消在第 j 列安放的皇后。如果没有出现攻击，在第 j 列安放的皇后不动，递归安放第 i+1行皇后。

# 解题时设置 4 个数组：

### col [n] ：col[i] 标识第 i 列是否安放了皇后

### md[2n-1] ：md[k] 标识第 k 条主对角线是否安放了皇后

### sd[2n-1] ：sd[k] 标识第 k 条次对角线是否安放了皇后

### q[n] ：q[i] 记录第 i 行皇后在第几列

# 利用行号i和列号j计算主对角线编号k的方法是k = n+i-j-1；计算次对角线编号k的方法是k = i+j。n皇后问题解法如下：

## **void** Queen( **int** i ) **{**

## **for** ( **int** j = 0**;** j < n**;** j++ ) **{**

## **if** ( col[j] == 0 **&&** md[n+i-j-1] == 0 **&&** sd[i+j] == 0 ) **{** //第 i 行第 j 列没有攻击

## col[j] = md[n+i-j-1] = sd[i+j] = 1**;** q[i] = j**;** //在第 i 行第 j 列安放皇后

**if** ( i == n ) **{** //输出一个布局

**for** ( j = 0**;** j < n**;** j++ ) **cout <<** q[j] **<< ‘,’;**

**cout << endl;**

**}**

**else {** Queen ( i+1)**;** //在第i+1行安放皇后

col[j] = md[n+i-j-1] = sd[i+j] = 0**;** q[i] = 0**;** //撤消第 i 行第 j 列的皇后

**}**

**}**

**}**

5-5 已知f为单链表的表头指针, 链表中存储的都是整型数据，试写出实现下列运算的递归算法：

(1) 求链表中的最大整数。

(2) 求链表的结点个数。

(3) 求所有整数的平均值。

【解答】

#**include** <iostream.h>//定义在头文件"RecurveList.h"中

**class** List**;**

**class** ListNode **{** //链表结点类

**friend class** List**;**

**private:**

**int** data**;** //结点数据

ListNode \*link**;** //结点指针

**public:**

ListNode ( **const int** item ) **:** data(item), link(NULL) **{ }** //构造函数

**};**

**class** List **{**  //链表类

**private:**

ListNode \*first, current**;**

**int** Max ( ListNode \*f )**;**

**int** Num ( ListNode \*f )**;**

**float** Avg ( ListNode \*f, **int&** n )**;**

**public:**

List ( ) **:** first(NULL), current (NULL) **{ }** //构造函数

~List ( )**{ }** //析构函数

ListNode\* NewNode ( **const int** item )**;** //创建链表结点, 其值为item

**void** NewList ( **const int** retvalue )**;** //建立链表, 以输入retvalue结束

**void** PrintList ( )**;** //输出链表所有结点数据

**int** GetMax ( ) **{ return** Max ( first )**; }** //求链表所有数据的最大值

**int** GetNum ( ) **{ return** Num ( first )**; }** //求链表中数据个数

**float** GetAvg ( ) **{ return** Avg ( first )**; }** //求链表所有数据的平均值

**};**

ListNode\* List **::** NewNode ( **const int** item ) **{** //创建新链表结点

ListNode \*newnode = **new** ListNode (item)**;**

**return** newnode**;**

**}**

**void** List **::** NewList ( **const int** retvalue ) **{** //建立链表, 以输入retvalue结束

first = NULL**; int** value**;** ListNode \*q**;**

**cout** << "Input your data:\n"**;**  //提示

**cin** >> value; //输入

**while** ( value != retvalue ) { //输入有效

q = NewNode ( value )**;**  //建立包含value的新结点

**if** ( first == NULL ) first = current = q**;** //空表时, 新结点成为链表第一个结点

**else {** current->link = q**;** current = q**; }**  //非空表时, 新结点链入链尾

**cin** >> value**;** //再输入

**}**

current->link = NULL**;**  //链尾封闭

**}**

**void** List **::** PrintList ( ) **{** //输出链表

**cout** << "\nThe List is : \n"**;**

ListNode \*p = first**;**

**while** ( p != NULL ) **{ cout** << p->data << ' '**;** p = p->link**; }**

**cout << ‘**\n**’;**

**}**

(1) 求链表中的最大整数

**int** List **::** Max ( ListNode \*f ) **{** //递归算法 **:** 求链表中的最大值

**if** ( f ->link == NULL ) **return** f ->data**;**  //递归结束条件

**int** temp = Max ( f ->link )**;** //在当前结点的后继链表中求最大值

**if** ( f ->data > temp ) **return** f ->data**;** //如果当前结点的值还要大, 返回当前检点值

**else return** temp**;** //否则返回后继链表中的最大值

**}**

(2) 求链表的结点个数

**int** List **::** Num ( ListNode \*f ) **{** //递归算法 **:** 求链表中结点个数

**if** ( f == NULL ) **return** 0**;** //空表, 返回0

**return** 1+ Num ( f ->link )**;** //否则, 返回后继链表结点个数加1

**}**

(3) 求所有整数的平均值

**float** List **::** Avg ( ListNode \*f , **int&** n ) **{** //递归算法 **:** 求链表中所有元素的平均值

**if** ( f ->link == NULL ) //链表中只有一个结点, 递归结束条件

**{** n = 1**; return** ( **float** ) (f ->data )**; }**

**else { float** Sum = Avg ( f ->link, n ) \* n**;** n++**;** **return** ( f ->data + Sum ) / n**; }**

**}**

#**include** "RecurveList.h" //定义在主文件中

**int** main ( **int** argc, **char**\* argv[ ] ) **{**

List test**; int** finished**;**

**cout** << “输入建表结束标志数据 ：”**;**

**cin** >> finished**;** //输入建表结束标志数据

test.NewList ( finished )**;** //建立链表

test.PrintList ( )**;** //打印链表

**cout** << "\nThe Max is : " << test.GetMax ( )**;**

**cout** << "\nThe Num is : " << test.GetNum ( )**;**

**cout** << "\nThe Ave is : " << test.GetAve () << '\n'**;**

printf ( "Hello World!\n" );

**return** 0**;**

**}**

5-6 画出下列广义表的图形表示和它们的存储表示：

(1) D(A(c), B(e), C(a, L(b, c, d)))

(2) J1(J2(J1, a, J3(J1)), J3(J1))

【解答】

(1) D(A(c), B(e), C(a, L(b, c, d))) (2) J1(J2(J1, a, J3(J1)), J3(J1))

J1

J2

J3

C

B

A

D

c

b

a

e

L

a

d

c

0 J1

2

2

∧

J1

∧

A

0 B

∧

B

0 A

1 e

2

∧

2

0 D

D

1 c

2

J2

∧

1 a

2

0 J2

2

0 C

2

1 a

C

∧

∧

0 J3

J3

2

∧

0 L

1 d

1 c

1 b

L

5-7 利用广义表的head和tail操作写出函数表达式，把以下各题中的单元素banana从广义表中分离出来：

(1) L1(apple, pear, banana, orange)

(2) L2((apple, pear), (banana, orange))

(3) L3(((apple), (pear), (banana), (orange)))

(4) L4((((apple))), ((pear)), (banana), orange)

(5) L5((((apple), pear), banana), orange)

(6) L6(apple, (pear, (banana), orange))

【解答】

(1) Head (Tail (Tail (L1) ) )

(2) Head (Head (Tail (L2) ) )

(3) Head (Head (Tail (Tail (Head (L3) ) ) ) )

(4) Head (Head (Tail (Tail (L4) ) ) )

(5) Head (Tail (Head(L5) ) )

(6) Head (Head (Tail (Head (Tail (L6) ) ) ) )

5-8 广义表具有可共享性，因此在遍历一个广义表时必需为每一个结点增加一个标志域mark，以记录该结点是否访问过。一旦某一个共享的子表结点被作了访问标志，以后就不再访问它。

(1) 试定义该广义表的类结构；

(2) 采用递归的算法对一个非递归的广义表进行遍历。

(3) 试使用一个栈，实现一个非递归算法，对一个非递归广义表进行遍历。

【解答】

(1) 定义广义表的类结构

为了简化广义表的操作，在广义表中只包含字符型原子结点，并用除大写字母外的字符表示数据，表头结点中存放用大写字母表示的表名。这样，广义表中结点类型三种：表头结点、原子结点和子表结点。

**class** GenList**;** //GenList类的前视声明

**class** GenListNode **{** //广义表结点类定义

**friend class** Genlist**;**

**private:**

**int** mark, utype**;** // utype ＝0 / 1 / 2, mark是访问标记

GenListNode\* tlink**;** //指向同一层下一结点的指针

**union {** //联合

**char** listname**;** // utype = 0, 表头结点情形：存放表名

**char** atom**;** // utype = 1, 存放原子结点的数据

GenListNode\* hlink**;** // utype = 2, 存放指向子表的指针

**}** value**;**

**public:**

GenListNode ( **int** tp**, char** info ) **:** mark (0), utype (tp), tlink (NULL) //表头或原子结点构造函数

**{ if** ( utype == 0 ) value.listname = info**; else** value.atom = info**; }**

GenListNode (GenListNode\* hp ) //子表构造函数

**:** mark (0), utype (2), value.hlink (hp) **{ }**

**char** Info ( GenListNode\* elem ) //返回表元素elem的值

**{ return** ( utype == 0 ) **?** elem->value.listname **:** elem->value.atom**; }**

**};**

**class** GenList **{** //广义表类定义

**private:**

GenListNode \*first**;** //广义表头指针

**void** traverse ( GenListNode \* ls )**;** //广义表遍历

**void** Remove ( GenListNode\* ls )**;** //将以ls为表头结点的广义表结构释放

**public:**

Genlist ( **char&** value )**;**  //构造函数, value是指定的停止建表标志数据

~GenList ( )**;** //析构函数

**void** traverse ( ); //遍历广义表

**}**

(2) 广义表遍历的递归算法

**void** GenList **::** traverse ( ) **{** //共有函数  
 traverse ( first )**;**

**}**

#**include** <iostream.h>

**void** GenList **::** traverse ( GenListNode \* ls ) **{** //私有函数, 广义表的遍历算法

**if** ( ls != NULL ) **{**

ls->mark = 1**;**

**if** ( ls->utype == 0 ) **cout** << ls->value.listname << ‘(’**;** //表头结点

**else if** ( ls->utype == 1 ) **{** //原子结点

**cout** << ls->value.atom**;**

**if** ( ls->tlink != NULL ) **cout** << ‘,’**;**

**}**

**else if** ( ls->utype == 2 ) **{** //子表结点

**if** ( ls->value.hlink->mark == 0 ) traverse( ls->value.hlink )**; //**向表头搜索

**else cout <<** ls->value.hlink->value.listname**;**

**if** ( ls->tlink != NULL ) **cout** << ‘,’**;**

**}**

traverse ( ls->tlink )**;** //向表尾搜索

**}**

**else cout** << ‘)’**;**

∧

**}**

0 0 A

0 2

0 2

∧

0 1 a

0 0 C

∧

∧

0 1 e

0 2

0 0 D

0 2

0 1 y

0 1 x

0 0 E

对上图所示的广义表进行遍历，得到的遍历结果为A(C(E(x, y), a), D(E, e) )。

1. 利用栈可实现上述算法的非递归解法。栈中存放回退时下一将访问的结点地址。

#**include** <iostream.h>

**#include** “stack.h”

**void** GenList **::** traverse ( GenListNode \*ls ) **{**

Stack **<**GenListNode**<Type>** \*> st**;**

**while** ( ls != NULL ) **{**

ls->mark = 1**;**

**if** ( ls->utype == 2 ) **{** //子表结点

**if** ( ls->value.hlink->mark == 0 ) //该子表未访问过

**{** st.Push ( ls->tlink )**;** ls =ls->value.hlink**; }** //暂存下一结点地址, 访问子表

**else {**

**cout <<** ls->value.hlink->value.listname**;** //该子表已访问过, 仅输出表名

**if** ( ls->tlink != NULL ) **{** **cout** << ‘,’**;** ls **=** ls->tlink**; }**

**}**

**}**

**else {**

**if** ( ls->utype == 0 ) **cout** << ls->value.listname << ‘(’**;** //表头结点

**else if** ( ls->utype == 1 ) **{** //原子结点

**cout** << ls->value.atom**;**

**if** ( ls->tlink != NULL ) **cout** << ‘,’**;**

**}**

**if** ( ls->tlink == NULL ) **{** //子表访问完, 子表结束处理

**cout** >> ‘)’**;**

**if** ( st.IsEmpty( ) == 0 ) **{** //栈不空

ls = st.GetTop ( )**;** st.Pop ( )**;** //退栈

**if** ( ls != NULL ) **cout <<** ‘,’**;**

**else cout <<** ‘)’**;**

**}**

**}**

**else** ls **=** ls->tlink**;** //向表尾搜索

**}**

**}**

**}**

(4) 广义表建立操作的实现

#**include** <iostream.h>

**#include** <ctype.h>

**#include** “stack.h”

**const int** maxSubListNum = 20**;** //最大子表个数

GenList **::** GenList ( **char&** value ) **{**

Stack <GenListNode\* > st (maxSubListNum)**;** //用于建表时记忆回退地址

SeqList <**char>** Name (maxSubListNum)**;** //记忆建立过的表名

SeqList <GenListNode \* > Pointr (maxSubListNum)**;** //记忆对应表头指针

GenListNode \* p, q, r**;** **Type** ch**;** **int** m = 0, ad, br**;** //m为已建表计数, br用于对消括号

**cout** << “广义表停止输入标志数据value **:** ”**; cin** >> value**;**

**cout** << “开始输入广义表数据, 如A(C(E( x, y ), a ), D(E(x, y), e) )”

**cin** >> ch**;** first = q = **new** GenListNode ( 0, ch )**;** //建立整个表的表头结点

**if** ( ch != value ) **{** Name.Insert (ch, m)**;** Pointr.Insert (q, m)**;** m++**; }** //记录刚建立的表头结点

**else** **return** 1**;**  //否则建立空表, 返回1

**cin** >> ch**;** **if** ( ch == ‘(’ ) st.Push ( q )**;** //接着应是‘(’, 进栈

**cin** >> ch**;**

**while** ( ch != value ) **{** //逐个结点加入

**switch** ( ch ) **{**

**case** ‘(’ **:** p = **new** GenListNode **<Type>** ( q )**;** //建立子表结点, p->hlink = q

r = st.GetTop( )**;** st.Pop( )**;** r->tlink = p**;** //子表结点插在前一结点r之后

st.Push( p )**;** st.Push( q )**;**  //子表结点及下一表头结点进栈

**break;**

**case** ‘)’ **:** q->tlink = NULL**;** st.pop( )**;** //子表建成, 封闭链, 退到上层

**if** ( st.IsEmpty ( ) == 0 )q = st.GetTop( )**;** //栈不空, 取上层链子表结点

**else return** 0**;** //栈空, 无上层链, 算法结束

**break;**

**case** ‘,’ **:** **break;**

**default :** ad = Name.Find (ch)**;** //查找是否已建立, 返回找到位置

**if** ( ad == -1 ) **{**  //查不到, 建新结点

p = q**;**

**if** ( isupper (ch) ) **{** //大写字母, 建表头结点

q **= new** GenListNode ( 0, ch )**;**

Name.Insert (ch, m)**;** Pointr.Insert (q, m)**;** m++**;**

**}**

**else** q = **new** GenListNode ( 1, ch )**;** //非大写字母, 建原子结点

p->tlink = q**;** //链接

**}**

**else {** //查到, 已加入此表

q = Pointr.Get (ad)**;** p = **new** GenListNode (q)**;** //建立子表结点, p->hlink = q

r = st.GetTop ( )**;** st.Pop ( )**;** r->tlink = p**;** st.Push (p)**;** q = p**;**

br = 0**;** //准备对消括号

**cin >>** ch**; if** ( ch == ‘(’ ) br++**;** //若有左括号, br加1

**while** ( br == 0 ) **{** //br为0表示括号对消完, 出循环

**cin >>** ch**;**

**if** ( ch == ‘(’ ) br++**; else if** ( ch == ‘)’ ) br--**;**

**}**

**}**

**}**

**cin** >> ch**;**

**}**

**}**

**四、其他练习题**

5-9 填空题

(1) 设有一个递归算法如下

**int** fact ( **int** n ) **{**

**if** ( n <= 0 ) **return** 1**;**

**else return** n\*fact ( n-1 )**;**

**}**

下面正确的叙述是\_\_\_\_\_\_\_\_。

A．计算fact(n) 需要执行n次递归

B．fact(7) = 5040

C．此递归算法最多只能计算到fact(8)

D．以上结论都不对

(2) 设有一个递归算法如下

**int** X ( **int** n ) **{**

**if** ( n <= 3 ) **return** 1**;**

**else return** X (n-2) + X (n-4) + 1**;**

**}**

试问计算X (X (8) ) 时需要计算\_\_\_\_\_\_\_\_次X函数

A．8次 B．9次 C．16次 D．18次

(3) 在Hanoi塔问题中，若A塔上有3片圆盘，都要搬到C塔上去，则下列语句\_\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_是错误的。

1. 第1步将最小圆盘从A塔搬到B塔
2. 第4步将最大圆盘从A塔搬到C塔
3. 第7步将最小圆盘从A塔搬到C塔
4. 需要8次才能完成工作

(4) 有一个递归算法如下

**int** maze ( **int** a, **int** b, **int** c ) **{**

**if** ( a < b ) **return** a**;**

**else return** c \* maze ( a / b, b, c ) + a % b**;**

**}**

下列叙述正确的是\_\_\_\_\_\_\_\_。

A．maze ( 1020, 10, 7 ) = 356 B．maze ( 352, 4, 11 ) = 16214

C．maze ( 16, 2, 2 ) = 8 D．以上三个答案都不对

(5) 设有广义表D (a, b, D)，其长度为\_\_\_\_\_\_\_\_，深度为\_\_\_\_\_\_\_\_。

A．∞ B．3 C．2 D．5

(6) 递归表、再入表、纯表、线性表之间的关系为\_\_\_\_\_\_\_\_。

A．再入表 > 递归表 > 纯表 > 线性表

B．递归表 > 线性表 > 再入表 > 纯表

C．递归表 > 再入表 > 纯表 > 线性表

D．递归表 > 再入表 > 线性表 > 纯表

(7) 广义表A (a)，则表尾为\_\_\_\_\_\_\_\_。

A．a B．( ( ) ) C．空表 D．(a)

(8) 广义表A ( (x, (a, b) ), (x, (a, b), y) )，则运算Head (Head (Tail (A) ) ) 为（ ）。

A．x B．(a, b) C．(x, (a, b) ) D．A

(9) 下列广义表是线性表的有（ ）。

A．E (a, (b, c) ) B．E (a, E) C．E (a, b) D．E (a, L ( ) )

(10) 下列广义表用图表示时，分支结点最多的是（ ）。

A．L ( (x, (a, b) ), (x, (a, b), y) ) B．A (s, (a, b) )

C．B ( (x, (a, b), y) ) D．D ( (a, b), (c, (a, b), D )

【解答】

(1) B (2) D (3) A, D (4) B (5) B, A

(6) C (7) C (8) A (9) C (10) D

5-10 试编写一个把10进制正整数转换为B进制（2 ≤B≤9）数输出的递归算法。

【解答】

**void** Transform ( **long** num, **int** B ) **{**

//把10进制正整数转换为B进制数的递归算法

**int** k**;**

**if** ( num != 0 ) **{**

k = num % B**;**

Transform ( num / B, B **);**

**cout** << k**;**

**}**

**}**

5-11 试编写一个算法，求解最大公因数问题：在求两个正整数m和n的最大公因数常常使用辗转相除法，反复计算直到余数为零为止。其递归定义为：

例如，求724和344的最大公因数，即计算GCD(724, 344)：

GCD(724, 344) = GCD(344, 724 % 344) = GCD(344, 36)

= GCD(36, 344 % 36) = GCD(36, 20)

= GCD(20, 36 % 20) = GCD(20, 16)

= GCD(16, 20 % 16) = GCD(16, 4)

= GCD(4, 16 % 4 ) = GCD(4, 0)

= 4

【解答】

求解最大公因数的算法如下：

**int** GCD ( **int** m, **int** n ) **{**

**if** ( n == 0 ) **return** m**;**

**else return** GCD ( n, m % n )**;**

**}**



5-12 求组合的公式：组合的数学定义为：

例如，Combin(m, n) = Combin(m-1, n) + Combin(m-1, n-1)

Combin(5, 2) = Combin(4, 2) + Combin(4, 1)

= Combin(3, 2) + Combin(3, 1) + Combin(3, 1) + Combin(3, 0)

= Combin(2, 2) + Combin(2, 1) + 2\* (Combin(2, 1) + Combin(2, 0) ) + 1

= 1 + Combin(1, 1) + Combin(1, 0) + 2 \* (Combin(1, 1) + Combin(1, 0) +1) + 1

= 1 + 1 + 1 + 2 \* ( 1 + 1 + 1) + 1

= 10

【解答】

求解组合Combin( m, n ) 的算法为：

**int** Combin ( **int** m**, int** n ) **{**

**if** ( m == n || n == 0 ) **return** 1**;**

**else return** Combin ( m-1, n ) + Combin ( m-1, n-1 )**;**

**}**

5-13 试说明以下递归函数的功能：

**int** unknown ( **int** \*A**, int** n ) **{**

**if** ( n <= 1 ) **return** 1**;**

**if** ( A[0] < A[1] ) **return** 0**;**

**return** unknown( A+1, n-1)；

**}**

【解答】

此递归函数用于判断数组A[ ] 的前n个元素是否是不递增的。不递增返回1，否则返回0。

5-14 编写一个递归算法，找出从自然数1, 2, 3, …, n中任取r个数的所有组合。例如n = 5, r = 3时所有组合为543，542，541，532，531，521，432，431，421，321。

【解答】

问题的想法是：为从自然数1, 2, …, n中任取r个数的所有组合，可轮流固定第一个数（n, n-1, …, r）。当组合的第一个数字选定时，其后的数字是从余下的n-1个数中取r-1个数的组合，这就将求n个数中取r个数的组合问题转化为求n-1个数中取r-1个数的组合问题。为保存求得的组合，设置一个全局数组result[ ]。result[0]存放组合的位数。函数将组合的第一个数字放入数组后，有良种可能的选择：一是还未确定组合的其余数字，继续递归以确定组合的后续数字；一是已经确定了组合的所有数字，输出这个组合。

求解组合的算法如下：

#**include** <iostream.h>

#**define** MaxNum 100

**int** result[MaxNum]**;**

**void** combin ( **int** n, **int** r ) **{**

**for** ( **int** i = n**;** i >= r**;** i*--* ) **{** //顺序选择组合的最大数字

result[r] = i;

**if** ( r > 1 ) combin ( i-1, r-1 )**;** //组合中还应有其他数字

**else {** //找到了一个组合，将组合中的数字输出

**for** ( **int** j = result[0]**;** j > 0**;** j*--* ) **cout** << result[j]**;**

**cout << endl;**

**}**

**}**

**}**

**void** main( ) **{**

result[0] = 3**;** combin(5, 3)**;**

**}**

5-15 编写一个递归算法，输出自然数1, 2, …, n这n个元素的全排列。

【解答】

由排列的定义，n个数字的全排列共有n! 种。例如，1, 2, 3的全排列有123, 132, 213, 231, 321, 312，共3! = 6种。对于n个数字，可把它们分别放入到n个位置上，让第一个位置依次取每一个数字，共有n种不同取法，对其后n-1个位置上的n-1个数字，共有 (n-1)! 种不同的排列，所以总共有n\*(n-1)! 种不同的排列；同样，对于从第二个位置开始的所有数字，让第二个位置依次取除第一个位置上的数字之外的剩余n-1个数字，共有n-1种不同的取法，对其后n-2个位置上的n-2个数字，共有 (n-2)! 种不同的排列，依次类推；当进行到第n位置时，只有一种取法，因为前n-1个位置已经固定了n-1个数字，剩余的一个数字被放在这个位置上。求解n个数字的全排列的递归算法如下：

**void** Permute ( **int** a[ ], **int** s, **int** n ) **{**

//对a[s]~a[n-1]中的n-s个元素进行全排列，s的初值为0

**int** i, temp**;**

**if** ( s == n-1 ) **{** //当递归结束时输出a中保存的一种排列

**for** ( i = 0**;** i < n**;** i++ ) **cout** << a[i] << " "**;**

**cout** << **endl;**

**}**

**else**  //继续递归排列

**for** ( i = s**;** i < n**;** i++ ) **{** //每循环一次使a[s]取一个新值

temp = a[s]**;** a[s] = a[i]**;** a[i] = temp**;** //交换a[s]与a[i]的值

Permute (a, s+1, n)**;** //对a[s+1]~a[n-1]中的数字递归排列

temp = a[s]**;** a[s] = a[i]**;** a[i] = temp**;** //恢复a[s]与a[i]的原有值

**}**

**}**

#**include** <iostream.h>

**const int** UpperLimit = 6**;** //定义全排列的元素个数的最大值

**void** main ( ) **{**

**int** a[UpperLimit], n**;** //定义存储n个整型元素的数组

**cout** << "Enter a number 'n' between 1 and " << UpperLimit << ":"**;**

**cin** >> n**;** //输入待全排列的元素的个数

**for** ( **int** i = 0**;** i < n**;** i++) a[i] = i+1**;** //给数组a赋初值

**cout << endl;**

Permute ( a, 0, n )**;** //对数组a中的n个元素（1~n）全排列

**cout << endl;**

**}**