北京理工大学数学与统计学院
《数据结构》（C 语言版）
编程作业参考答案

王鹏翔（2016 级）

如有问题请联系我: shawn.pxwang@qq.com

2020-4-30 V1.0

目录 CONTENTS

打星号

问题描述

大家回忆一下刚学 C 语言时的打菱形星号，当输入 4 时输出

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

每一行前面的空格数分别为 3 2 1 0 1 2 3，每一行的 * 个数为 1 3 5 7 5 3 1

参考答案

```c
#include<stdio.h>

int main()
{
    int n, i, j;
    scanf("%d", &n);
    for(i = 1; i <= n; ++i)
    {
        for(j = 1; j <= n - i; ++j)
            printf(" ");
        for(j = 1; j <= 2 * i - 1; ++j)
            printf("*");
        printf("\n");
    }
    i = i - 2;
    for(; i >= 1; --i)
    {
        for(j = 1; j <= n - i; ++j)
            printf(" ");
        for(j = 1; j <= 2 * i - 1; ++j)
            printf("*");
        printf("\n");
    }

```

```
    system("pause");
    return 0;
}
```

一元多项式相乘

问题描述

题目说明:

要求采用链表形式，求两个一元多项式的乘积：$h_3 = h_1 * h_2$。函数原型为:

`void multiplication( NODE * h1, NODE * h2, NODE * h3 )`

输入:

输入数据为两行，分别表示两个一元多项式。每个一元多项式以指数递增的顺序输入多项式各项的系数（整数）、指数（整数）。

例如：$1+2x+x^2$ 表示为：<1,0>,<2,1>,<1,2>,

输出:

以指数递增的顺序输出乘积： <系数,指数>,<系数,指数>,<系数,指数>,

零多项式的输出格式为：<0,0>,

预设代码

```
/* PRESET CODE BEGIN - NEVER TOUCH CODE BELOW */

#include <stdio.h>
#include <stdlib.h>

typedef struct node
{   int    coef, exp;
    struct node  *next;
} NODE;

void multiplication( NODE *, NODE * , NODE * );
void input( NODE * );
void output( NODE * );

void input( NODE * head )
{   int flag, sign, sum, x;
    char c;
```

```
18.
19.     NODE * p = head;
20.
21.     while ( (c=getchar()) !='\n' )
22.     {
23.         if ( c == '<' )
24.         {    sum = 0;
25.              sign = 1;
26.              flag = 1;
27.         }
28.         else if ( c =='-' )
29.              sign = -1;
30.         else if( c >='0'&& c <='9' )
31.         {    sum = sum*10 + c - '0';
32.         }
33.         else if ( c == ',' )
34.         {    if ( flag == 1 )
35.              {    x = sign * sum;
36.                   sum = 0;
37.                   flag = 2;
38.           sign = 1;
39.              }
40.         }
41.         else if ( c == '>' )
42.         {    p->next = ( NODE * ) malloc( sizeof(NODE) );
43.              p->next->coef = x;
44.              p->next->exp  = sign * sum;
45.              p = p->next;
46.              p->next = NULL;
47.              flag = 0;
48.         }
49.     }
50. }
51.
52. void output( NODE * head )
53. {
54.     while ( head->next != NULL )
55.     {   head = head->next;
56.         printf("<%d,%d>,", head->coef, head->exp );
57.     }
58.     printf("\n");
59. }
60.
61. int main()
```

```
{   NODE * head1, * head2, * head3;

    head1 = ( NODE * ) malloc( sizeof(NODE) );
    input( head1 );

    head2 = ( NODE * ) malloc( sizeof(NODE) );
    input( head2 );

    head3 = ( NODE * ) malloc( sizeof(NODE) );
    head3->next = NULL;
    multiplication( head1, head2, head3 );

    output( head3 );

    return 0;
}

/* PRESET CODE END - NEVER TOUCH CODE ABOVE */
```

参考答案

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
    int    coef, exp;
    struct node  *next;
} NODE;

NODE *add(NODE *, NODE *);
void freepolynomial(NODE * &);
NODE *mo_poly_multiplication(int, int, NODE *);
void multiplication(NODE *, NODE *, NODE * &);
void input(NODE *);
void output(NODE *);

void input(NODE * head)
{
    int flag, sign, sum, x;
    char c;

    NODE * p = head;

```

```
24.     while ((c = getchar()) != '\n')
25.     {
26.         if (c == '<')
27.         {
28.             sum = 0;
29.             sign = 1;
30.             flag = 1;
31.         }
32.         else if (c == '-')
33.             sign = -1;
34.         else if (c >= '0'&& c <= '9')
35.         {
36.             sum = sum * 10 + c - '0';
37.         }
38.         else if (c == ',')
39.         {
40.             if (flag == 1)
41.             {
42.                 x = sign * sum;
43.                 sum = 0;
44.                 flag = 2;
45.                 sign = 1;
46.             }
47.         }
48.         else if (c == '>')
49.         {
50.             p->next = (NODE *)malloc(sizeof(NODE));
51.             p->next->coef = x;
52.             p->next->exp = sign * sum;
53.             p = p->next;
54.             p->next = NULL;
55.             flag = 0;
56.         }
57.     }
58. }
59.
60. void output(NODE * head)
61. {
62.     //打印多项式
63.     if (head->next == NULL)  printf("<0,0>,"); //零多项式的输出格式
64.     while (head->next != NULL)   //说明是带头结点的单链表
65.     {
66.         head = head->next;
67.         printf("<%d,%d>,", head->coef, head->exp);
```

```
68.     }
69.     printf("\n");
70.}
71.
72.NODE *add(NODE * h1, NODE * h2) {
73.     //将多项式 h1 和 h2 相加，返回和多项式的头指针
74.     NODE * h3;  //和多项式的头指针
75.     h3 = (NODE *)malloc(sizeof(NODE));
76.     NODE * cursor_h1 = h1->next, *cursor_h2 = h2->next, *cursor_h3 = h3;
77.     while (cursor_h1 != NULL && cursor_h2 != NULL)
78.     {
79.         if (cursor_h1->exp == cursor_h2->exp && cursor_h1->coef + cursor_h2->coef ==
0) //某一项相加抵消的情况，直接跳过
80.         {
81.             cursor_h1 = cursor_h1->next; cursor_h2 = cursor_h2->next;
82.             continue;
83.         }
84.
85.         cursor_h3->next = (NODE *)malloc(sizeof(NODE));
86.         cursor_h3 = cursor_h3->next;
87.         if (cursor_h1->exp < cursor_h2->exp)
88.         {
89.             cursor_h3->exp = cursor_h1->exp;
90.             cursor_h3->coef = cursor_h1->coef;
91.             cursor_h1 = cursor_h1->next;
92.         }
93.         else if (cursor_h2->exp < cursor_h1->exp)
94.         {
95.             cursor_h3->exp = cursor_h2->exp;
96.             cursor_h3->coef = cursor_h2->coef;
97.             cursor_h2 = cursor_h2->next;
98.         }
99.         else
100.         {
101.             cursor_h3->exp = cursor_h1->exp;
102.             cursor_h3->coef = cursor_h1->coef + cursor_h2->coef;
103.             cursor_h1 = cursor_h1->next; cursor_h2 = cursor_h2->next;
104.         }
105.     }
106.     if (!(cursor_h1 == NULL && cursor_h2 == NULL))  //把剩下的那一部分原版复制
107.     {
108.         while (cursor_h1 != NULL)
109.         {
110.             cursor_h3->next = (NODE *)malloc(sizeof(NODE));
```

```
111.             cursor_h3 = cursor_h3->next;
112.             cursor_h3->coef = cursor_h1->coef;
113.             cursor_h3->exp = cursor_h1->exp;
114.
115.             cursor_h1 = cursor_h1->next;
116.         }
117.         while (cursor_h2 != NULL)
118.         {
119.             cursor_h3->next = (NODE *)malloc(sizeof(NODE));
120.             cursor_h3 = cursor_h3->next;
121.             cursor_h3->coef = cursor_h2->coef;
122.             cursor_h3->exp = cursor_h2->exp;
123.
124.             cursor_h2 = cursor_h2->next;
125.         }
126.     }
127.
128.     cursor_h3->next = NULL;
129.     return h3;
130. }
131.
132. void freepolynomial(NODE * &h) {
133.     //释放多项式的内存空间
134.     NODE * forward = h, *cleaner = h;
135.     while (forward != NULL)
136.     {
137.         forward = forward->next;
138.         free(cleaner);
139.         cleaner = forward;
140.     }
141.     //free(h);  //不知道为什么，加上这句代码就会报错...所以就干脆不要了
142.     h = NULL;
143. }
144.
145. NODE *mo_poly_multiplication(int coef_mo, int exp_mo, NODE * poly) {
146.     //计算单项式与多项式的乘积，coef_mo 为单项式的系数，exp_mo 为单项式的指数，poly 为多项式（头指针），返回乘积多项式的头指针
147.     NODE * result;  //乘积多项式的头指针
148.     result = (NODE *)malloc(sizeof(NODE));
149.     NODE * cursor_result = result;
150.     if (coef_mo != 0)   //非零单项式
151.     {
152.         NODE *cursor_poly = poly->next;  //cursor_poly 用于遍历多项式的每一项
153.         while (cursor_poly != NULL) //复制多项式，改变每一项的系数和指数
```

```
154.         {
155.             if (cursor_poly->coef == 0)  //系数为0的项，不用管
156.             {
157.                 cursor_poly = cursor_poly->next;
158.                 continue;
159.             }
160.             cursor_result->next = (NODE *)malloc(sizeof(NODE));
161.             cursor_result = cursor_result->next;
162.             cursor_result->coef = cursor_poly->coef * coef_mo;
163.             cursor_result->exp = cursor_poly->exp + exp_mo;
164.
165.             cursor_poly = cursor_poly->next;
166.         }
167.     }
168.     cursor_result->next = NULL;
169.     return result;
170. }
171.
172. void multiplication(NODE * h1, NODE * h2, NODE * &h3) {
173.     //将多项式h1和h2相乘，乘积多项式的头指针存放于h3中
174.     //方法：乘法分配律，将h2拆成一个个单项式，分别计算与这些h1与这些单项式的乘积，最后加起来
175.     NODE * cursor_h2 = h2->next, * added, * recycled, * temp;  //cursor_h2用于遍历h2（遍历组成h2的每个单项式），没有用的可以释放的多项式都放在recycled
176.     while (cursor_h2 != NULL)
177.     {
178.         added = mo_poly_multiplication(cursor_h2->coef, cursor_h2->exp, h1);
179.         temp = add(h3, added); recycled = h3; h3 = temp;
180.         freepolynomial(recycled);   //用后即焚
181.         recycled = added; freepolynomial(recycled); //用后即焚
182.
183.         cursor_h2 = cursor_h2->next;
184.     }
185. }
186.
187. int main()
188. {
189.     NODE * head1, * head2, * head3;
190.
191.     head1 = (NODE *)malloc(sizeof(NODE));
192.     input(head1);
193.
194.     head2 = (NODE *)malloc(sizeof(NODE));
195.     input(head2);
```

```
196.
197.     head3 = (NODE *)malloc(sizeof(NODE));
198.     head3->next = NULL;
199.     multiplication(head1, head2, head3);
200.
201.     output(head3);
202.
203.     system("pause");
204.     return 0;
205. }
```

约瑟夫问题

问题描述

(本题要求用循环链表实现)

约瑟夫问题是一个经典的问题。已知 n 个人（不妨分别以编号 1，2，3，...，n 代表）围坐在一张圆桌周围，从编号为 k 的人开始，从 1 开始顺时针报数 1, 2, 3, ...，顺时针数到 m 的那个人，出列并输出。然后从出列的下一个人开始，从 1 开始继续顺时针报数，数到 m 的那个人，出列并输出，...依此重复下去，直到圆桌周围的人全部出列。

输入：n,k,m

输出：按照出列的顺序依次输出出列人的编号，编号中间相隔一个空格,每 10 个编号为一行。

非法输入的对应输出如下

a)

输入:：n、k、m 任一个小于 1

输出：n,m,k must bigger than 0.

b)

输入：k>n

输出：k should not bigger than n.

例

输入

9,3,2

输出

4 6 8 1 3 7 2 9 5

参考答案

```c
#include<stdio.h>
#include<stdlib.h>


typedef int Status;
#define OK 1
#define ERROR 0
#define OVERFLOW -2

typedef int ElemType;

#define HEAD_INSERTION 1
#define TAIL_INSERTION 0

#define DELETED 1
#define UNDELETED 0

typedef struct Lnode {
    ElemType data;
    int flag;   //flag 为 DELETED 表示已删除，UNDELETED 表示未删除
    struct Lnode *next;
}LNode, *LinkList;

Status InitList_CL(LinkList &L) {
    //初始化操作：构造一个空的（带头结点的）循环链表
    L = (LinkList)malloc(sizeof(LNode));
    if (!L) exit(OVERFLOW);
    L->next = L; //循环
    L->flag = DELETED;   //技巧：把头结点标记为已删除，在 PaceForward 的时候会自动跳过头结点
    return OK;
}

Status ListInsert_CL(LinkList &L, int i, ElemType e) {
    //插入操作：在（带头结点的）循环链表的某个位置插入一个新的元素
    //i 表示插入元素的位序，e 表示插入的元素
    LinkList s; int k; LinkList cursor = L;
    if (!L || i < 1) exit(ERROR);    //i 不合法，L 是空指针
    s = (LinkList)malloc(sizeof(LNode));    //新结点
    s->data = e;
    s->flag = UNDELETED;
    for (k = 0; k < i - 1; k++)
    {
        if (cursor->next == L)   exit(ERROR);    //i 不合法
```

```
44.         cursor = cursor->next;
45.     }
46.     s->next = cursor->next;
47.     cursor->next = s;
48.     return OK;
49. }
50. 
51. Status Create_ONE_TO_N_List_CL(LinkList &L, int InsertType, int n) {
52.     //创建从数字 1 到 n 的（带头结点的）循环链表
53.     //InsertType 表示插入类型，HEAD_INSERTION 为头插法，TAIL_INSERTION 为尾插法
54.     //n 表示循环链表的长度
55.     int i;
56.     InitList_CL(L);
57.     for (i = 1; i <= n; i++)
58.     {
59.         if (InsertType == HEAD_INSERTION)
60.             ListInsert_CL(L, 1, i);
61.         else if (InsertType == TAIL_INSERTION)
62.             ListInsert_CL(L, i, i);
63.         else
64.             exit(ERROR);
65.     }
66.     return OK;
67. }
68. 
69. Status GetElem_L(LinkList L, int i, LinkList &pe, ElemType &e) {
70.     //取元素操作
71.     //i 为要获取的元素位序，pe 存放指向该元素的指针，e 存放获取到的元素
72.     int k; LinkList cursor = L;
73.     if (!L || i < 0) exit(ERROR);     //i 不合法，L 是空指针
74.     if (i == 0)
75.     {
76.         pe = L;
77.         return OK;
78.     }
79.     cursor = cursor->next;
80.     for (k = 0; k < i - 1; k++)
81.     {
82.         if (cursor->next == L)    exit(ERROR);     //i 不合法
83.         cursor = cursor->next;
84.     }
85.     pe = cursor;
86.     e = cursor->data;
87.     return OK;
```

```
88.}
89.
90.Status PaceForward_L(LinkList L, LinkList start, int i, LinkList &destination, ElemType &e) {
91.    //在某个位置的基础上向前进 i 个位序（跳过已删除的）
92.    //start 为指向起点结点的指针，i 为要前进的步数，destination 存放指向终点结点的指针，e 存放终点结点的元素
93.    int k; LinkList cursor = start;
94.    if (!L || i < 1) exit(ERROR);    //i 不合法，L 是空指针
95.
96.    for (k = 0; k < i; k++)
97.    {
98.        while (cursor->next->flag == DELETED) cursor = cursor->next;
99.        cursor = cursor->next;
100.    }
101.    destination = cursor;
102.    e = cursor->data;
103.    return OK;
104.}
105.
106.int main() {
107.    LinkList L, cursor; int n, k, m; ElemType e; int counter = 0;
108.    scanf_s("%d,%d,%d", &n, &k, &m);
109.    if (n < 1 || k < 1 || m < 1)
110.    {
111.        printf("n,m,k must bigger than 0.\n");
112.        exit(0);
113.    }
114.    if (k > n)
115.    {
116.        printf("k should not bigger than n.\n");
117.        exit(0);
118.    }
119.
120.    Create_ONE_TO_N_List_CL(L, TAIL_INSERTION, n);
121.
122.    GetElem_L(L, k - 1, cursor, e); //因为第 k 个人报数 1，相当于第 k-1 个人报数"0"
123.    PaceForward_L(L, cursor, m, cursor, e); //通过引用参数 destination 更新 cursor
124.    printf("%d", e);
125.    cursor->flag = DELETED;
126.    for(counter = 1; counter < n; counter++) //如果跳过的已删除结点的个数超过 n（绕了整整一圈），说明所有结点已经删除，结束
127.    {
128.        if (counter % 10 == 0)  printf("\n");
```

```
        PaceForward_L(L, cursor, m, cursor, e); //通过引用参数 destination 更新 cursor
        if (counter % 10 != 0)  printf(" ");
        printf("%d", e);
        cursor->flag = DELETED;
    }
    printf("\n");

    system("pause");
    return 0;
}
```

括号匹配

问题描述

假设一个算术表达式中包含圆括号、方括号两种类型的括号，试编写一个判断表达式中括号是否匹配的程序，匹配返回 Match succeed!，否则返回 Match false!。

例

[1+2*(3+4*(5+6))]括号匹配

(1+2)*(1+2*[(1+2)+3)括号不匹配

输入

包含圆括号、方括号两种类型括号的算术表达式

输出

匹配输出 Match succeed!

不匹配输出 Match false!

例

输入[1+2* (3+4*(5+6))]

输出 Match succeed!

参考答案

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

typedef int Status;
```

```
6. #define OK 1
7. #define ERROR 0
8. #define OVERFLOW -1
9. #define UNDERFLOW -2
10.
11.#define STACK_INIT_SIZE 100
12.#define STACKINCREMENT 10
13.
14.typedef struct {
15.    //用于存放括号的栈，只存储'(', ')', '[', ']'
16.    char * base;
17.    char * top;
18.    int stacksize;
19.}SqStack;
20.
21.Status InitStack(SqStack &S) {
22.    //初始化操作：建立空栈
23.    S.base = (char *)malloc(STACK_INIT_SIZE * sizeof(char));
24.    if (S.base == NULL) return OVERFLOW;    //malloc 失败
25.    S.top = S.base;
26.    S.stacksize = STACK_INIT_SIZE;
27.    return OK;
28.}
29.
30.Status GetTop(SqStack S, char &e) {
31.    //取栈顶元素，结果保存到 e 中
32.    if (S.top == S.base) return ERROR;  //空栈
33.    e = *(S.top - 1);
34.    return OK;
35.}
36.
37.Status Push(SqStack &S, char e) {
38.    //入栈操作，入栈元素为 e
39.    if (S.top - S.base == S.stacksize)  //栈满，增加空间
40.    {
41.        S.base = (char *)realloc(S.base, (S.stacksize + STACKINCREMENT) * sizeof(cha
r));
42.        if (S.base == NULL) exit(OVERFLOW); //realloc 失败
43.        S.stacksize += STACKINCREMENT;
44.    }
45.    * S.top = e;
46.    S.top++;
47.    return OK;
48.}
```

```
49.
50.Status Pop(SqStack &S, char &e) {
51.    //出栈操作，出栈元素存放在 e 中
52.    if (S.top == S.base)    return UNDERFLOW;   //空栈，会发生下溢
53.    S.top--;
54.    e = * S.top;
55.    return OK;
56.}
57.
58.int main() {
59.    char expression[100]; SqStack S; int i; char c, top, temp;
60.    scanf_s("%s", expression, 100);
61.    InitStack(S);
62.    for (i = 0; i < strlen(expression); ++i)
63.    {
64.        c = expression[i]; top = 0; //top 归零，防止上一次循环的值被利用
65.        switch (c)
66.        {
67.        case '(':Push(S, c); break; //左括号都入栈
68.        case '[':Push(S, c); break;
69.        case ')':
70.            GetTop(S, top);
71.            if (top == '(') //右括号，如果和栈顶配对，就让配对的左括号出栈
72.                Pop(S, temp);
73.            else //否则匹配失败
74.            {
75.                printf("Match false!\n");
76.                exit(0);
77.            }
78.            break;
79.        case ']':
80.            GetTop(S, top);
81.            if (top == '[')
82.                Pop(S, temp);
83.            else
84.            {
85.                printf("Match false!\n");
86.                exit(0);
87.            }
88.            break;
89.        default:
90.            break;  //如果不是这四种括号，啥也不干，直接跳过
91.        }
92.    }
```

```
    if (S.top == S.base)    //如果存括号的栈为空，匹配成功
        printf("Match succeed!\n");
    else
        printf("Match false!\n");

    system("pause");
    return 0;
}
```

从中缀向后缀转换表达式

问题描述

中缀表达式就是我们通常所书写的数学表达式，后缀表达式也称为逆波兰表达式，在编译程序对我们书写的程序中的表达式进行语法检查时，往往就可以通过逆波兰表达式进行。我们所要设计并实现的程序就是将中缀表示的算术表达式转换成后缀表示，例如，将中缀表达式

(A-(B*C+D)*E)/(F+G)

转换为后缀表示为：

ABC*D+E*--FG+/

注意：为了简化编程实现，假定变量名均为单个字母，运算符只有 +,-,*,/ 和 ^ （指数运算），可以处理圆括号（），并假定输入的算术表达式正确。

要求：使用栈数据结构实现 ， 输入的中缀表达式以 # 号结束

输入

整数 N。表示下面有 N 个中缀表达式

N 个由单个字母和运算符构成的表达式

输出

N 个后缀表达式。

参考答案

```
#include<stdio.h>
#include<stdlib.h>

/*
注意：本程序简单处理了很多报错机制，遇到某些错误时会直接 exit(0)退出程序。
如果闪退，请检查以下错误是否可能出现：
初始化栈 malloc 失败
```

```
    进栈上溢
    出栈下溢
    取空栈栈顶元素
    查找给定操作符集合 {'#','+','-','*','/','^','(',')'} 以外的操作符的栈内优先级或栈外优先级
    */


    #define STACK_SIZE 20

    //不可扩充栈
    typedef struct {
        char * base;
        char * top;
    }Stack;

    void InitStack(Stack &S) {
        //构建一个空栈
        S.base = (char *)malloc(STACK_SIZE * sizeof(char));
        if (!S.base) exit(0);
        S.top = S.base;
    }

    void Push(Stack &S, char e) {
        //进栈
        if (S.top == S.base + STACK_SIZE) exit(0);
        *S.top++ = e;
    }

    char Pop(Stack &S) {
        //出栈
        if (S.top == S.base) exit(0);
        char e = *(--S.top);
        return e;
    }

    char GetTop(Stack S) {
        //取栈顶元素
        if (S.base == S.top)    exit(0);
        return *(S.top - 1);
    }

    int isp(char c) {
        //栈内优先级（in stack priority）
```

```
51.         switch (c)
52.         {
53.         case '#':return 0; break;
54.         case '(':return 1; break;
55.         case '^':return 6; break;
56.         case '*':return 5; break;
57.         case '/':return 5; break;
58.         case '+':return 3; break;
59.         case '-':return 3; break;
60.         case ')':return 8; break;
61.         default:exit(0); break;
62.         }
63.     }
64.
65.     int icp(char c) {
66.         //栈外优先级（in coming priority）
67.         switch (c)
68.         {
69.         case '#':return 0; break;
70.         case '(':return 8; break;
71.         case '^':return 7; break;
72.         case '*':return 4; break;
73.         case '/':return 4; break;
74.         case '+':return 2; break;
75.         case '-':return 2; break;
76.         case ')':return 1; break;
77.         default:exit(0); break;
78.         }
79.     }
80.
81.     int IsOperator(char c) {
82.         //检查是否是操作符，如果是返回 1，不是返回 0
83.         if (c == '#' || c == '+' || c == '-
' || c == '*' || c == '/' || c == '^' || c == '(' || c == ')')
84.             return 1;
85.         else return 0;
86.     }
87.
88.     int main()
89.     {
90.         Stack OPTR; int N, i, j; char expressions[10][100], ch, op;
91.         scanf("%d", &N);
92.         for (i = 0; i < N; ++i)
93.             scanf("%s", &expressions[i]);
```

```
94.
95.         for (i = 0; i < N; ++i)
96.         {
97.             InitStack(OPTR);    //每次处理新的表达式时初始化栈（不释放原来栈的空间，反复利
用）
98.             Push(OPTR, '#');    //拿栈内优先级最低的操作符'#'垫底
99.             j = 0;
100.            while ((ch = expressions[i][j]) != '\0')
101.            {
102.                if (!IsOperator(ch))    //读取到操作数，直接输出
103.                {
104.                    putchar(ch);
105.                    j++;    //读取下一个字符
106.                }
107.                else //读取到操作符
108.                {
109.                    op = GetTop(OPTR);  //栈顶操作符
110.
111.                    if (isp(op) < icp(ch))   //栈顶操作符优先级小于当前操作符
112.                    {
113.                        Push(OPTR, ch); //当前操作符进栈
114.                        j++;    //读取下一个字符
115.                    }
116.                    else if (isp(op) > icp(ch))  //栈顶操作符优先级大于当前操作符
117.                        putchar(Pop(OPTR)); //栈顶操作符退栈并输出，不读取下一个字符
118.                    else //栈顶操作符优先级与当前操作符相等（有两种情况：两个'#'配对、括号配
对）
119.                    {
120.                        Pop(OPTR);  //栈顶操作符(左'#'或左括号）退栈，不输出，读取下一个字
符
121.                        j++;    //读取下一个字符
122.                    }
123.
124.                }
125.            }
126.            printf("\n");
127.        }
128.
129.        system("pause");
130.        return 0;
131.    }
```

游戏 2048

问题描述

《2048》是一款数字益智游戏，在 4*4 的方格中通过上下左右滑动来控制数字的变化，游戏胜利的条件是出现 2048 这个数字。

游戏规则如下：

1、玩家每次可以选择上下左右其中一个方向去滑动，定义滑动的方向为前，滑动的反方向为后，每滑动一次，所有的数字方块都会向前移动靠拢至边缘。

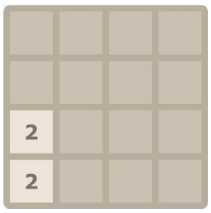
2、每一行（列）从最前方第二个方块依次向前方方块发起撞击，相撞的两个方块数字不同时不发生变化，撞击发起块向后顺延，相撞的两个方块相同时变成一个新的数值相加的数字块，后续的数字块依次向前递补空位，撞击发起块变为新生成数字块的后面第二个数字块。

3、撞击结束后系统会在空白的地方随机出现一个数字方块 2 或者 4。

对 4*4 方格中的 16 格分别赋予编号 1-16

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

初始状态：



当玩家游戏一段时间后，出现状态 1，

状态 1：

此时玩家向右滑动，出现状态 2，

状态 2：

当玩家游戏一段时间后出现状态 3：

状态 3：

在状态 3 的情况下向下滑动出现状态 4

状态 4：

本题输入：

输入 1 ：

按格子编号 1-16 输入 2048 游戏的一个状态序列，编号对应的格子没有数字则输入 0，如输入 0 0 0 0 4 0 2 0 4 0 2 2 2 8 8 8 表示状态 1

输入 2：一个用户操作和新增块地址序列，a 表示向左滑动，s 表示向下滑动，d 表示向右滑动，w 表示向上滑动

如输入 w 1 2 a 5 4 s 11 2 d 13 4 d 9 2 ，表示用户依次进行了如下 5 次操作：

玩家向上滑动一次，之后在编号 1 的位置新出现一个数值为 2 的新增块

玩家向左滑动一次，之后在编号 5 的位置新出现一个数值为 4 的新增块

玩家向下滑动一次，之后在编号 11 的位置新出现一个数值为 2 的新增块

玩家向右滑动一次，之后在编号 13 的位置新出现一个数值为 4 的新增块

玩家向右滑动一次，之后在编号 9 的位置新出现一个数值为 2 的新增块

如果编号所在的位置不为空，则修改 编号=编号%16+1，并探索编号所在位置是否为空，弱编号位置不为空，则重复 编号=编号%16+1，直至探索编号位置为空，并增加新增块

输出：从编号 1 到编号 16 方格的数字，格子为空则输出 0

PS：有兴趣的同学可以自行补全 胜利判断、随机数出现功能，完成一个完整的 2048 游戏。附游戏地址 http://www.3366.com/flash/106550.shtml

参考答案

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4.
5. void moving_to_left(int panel[16]) {
```

```
    //向左移动操作
    int pos, i, j, k, counter, step, NonZero[4];

    for (pos = 0; pos <= 12; pos = pos + 4)  //按行执行向左移动操作，panel[0]、panel[4]、panel[8]、panel[12]为面板各行第一个元素
    {
        for (i = 0; i < 4; ++i)  //NonZero 数组归零
            NonZero[i] = 0;
        counter = 0; j = 0; k = 0;
        for (i = 0; i < 4; ++i)
        {
            if (panel[pos + k] == 0) { k++; continue; }
            NonZero[j++] = panel[pos + k];  //NonZero 为该行非零元素（有顺序）
            counter++;  //counter 计数非零元素个数
            k++;
        }

        step = 0; k = 0;
        for (i = 0; i < 4; ++i)
        {
            if (i >= counter)    //避免两个 0 比较大小
                continue;
            if (i != 3 && NonZero[i] == NonZero[i + 1]) //相邻元素相等的，合并
            {
                panel[pos + k++] = 2 * NonZero[i];
                i++;    //前进两个
                step++;
            }
            else if (i == 3 && NonZero[i - 1] == NonZero[i] && NonZero[i - 2] != NonZero[i - 1]);
            else
            {
                panel[pos + k++] = NonZero[i];
                step++;
            }
        }
        for (i = 0; i < 4 - step; ++i)   panel[pos + k++] = 0;   //补上后面的 0，一共补(4 - step)个 0
    }
}

void change_direction_to_left(int panel[16], char orig_direction) {
    //改变游戏面板的方向：面板的 orig_direction 这个边变换为左边（保持面板在屏幕上做平面刚体运动）
```

```
46.     //为了方便，将游戏面板方向统一调成向左的方向，在完成 moving_to_left 操作后变回原来的方向
47.     int temp[4][4], i, j, k;     //数组 temp 临时存放改变方向的面板
48.     k = 0;
49.     switch (orig_direction)
50.     {
51.     case 'a':   //原地不动
52.         for (i = 0; i < 4; ++i)
53.             for (j = 0; j < 4; ++j)
54.                 temp[i][j] = panel[k++];
55.         break;
56.     case 's':   //顺时针旋转 90 度
57.         for (i = 0; i < 4; ++i)
58.             for (j = 0; j < 4; ++j)
59.                 temp[j][3 - i] = panel[k++];
60.         break;
61.     case 'd':   //关于中间竖线轴对称变换（不采用旋转 180 度即中心对称的方法）
62.         for (i = 0; i < 4; ++i)
63.             for (j = 0; j < 4; ++j)
64.                 temp[i][3 - j] = panel[k++];
65.         break;
66.     case 'w':   //逆时针旋转 90 度
67.         for (i = 0; i < 4; ++i)
68.             for (j = 0; j < 4; ++j)
69.                 temp[3 - j][i] = panel[k++];
70.         break;
71.     default:
72.         break;
73.     }
74.     //将临时面板 temp 赋值给原面板
75.     k = 0;
76.     for (i = 0; i < 4; ++i)
77.         for (j = 0; j < 4; ++j)
78.             panel[k++] = temp[i][j];
79. }
80.
81. void moving(int panel[16], char direction) {
82.     //移动操作（移动整个游戏面板）
83.     //direction：移动方向
84.     switch (direction)
85.     {
86.     case 'a':
87.         moving_to_left(panel);
88.         break;
```

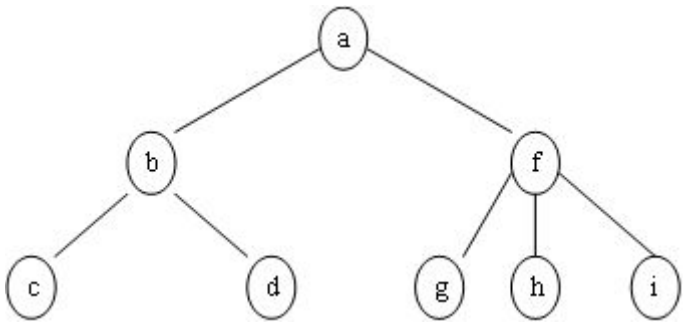
```
89.     case 's':
90.         change_direction_to_left(panel, 's');    //顺时针旋转 90 度
91.         moving_to_left(panel);
92.         change_direction_to_left(panel, 'w');    //逆变换：逆时针旋转 90 度
93.         break;
94.     case 'd':
95.         change_direction_to_left(panel, 'd');    //关于中间竖线轴对称变换
96.         moving_to_left(panel);
97.         change_direction_to_left(panel, 'd');    //逆变换和正变换相同
98.         break;
99.     case 'w':
100.         change_direction_to_left(panel, 'w');    //逆时针旋转 90 度
101.         moving_to_left(panel);
102.         change_direction_to_left(panel, 's');    //逆变换：顺时针旋转 90 度
103.         break;
104.     default:
105.         break;
106.     }
107.}
108.
109.void add_new_number(int panel[16], int new_number_pos, int new_number) {
110.     //在指定空白位置添加一个新数字
111.     //new_number_pos 为新数字的位置，new_number 为新数字
112.     if(panel[new_number_pos - 1] == 0)
113.         panel[new_number_pos - 1] = new_number;
114.     else
115.     {
116.         int i = new_number_pos;
117.         while (panel[i] != 0)   i++;
118.         panel[i] = new_number;
119.     }
120.}
121.
122.int main() {
123.     //游戏面板(panel)
124.     //键盘方向(direction)：'a'表示向左移动，'s'表示向下移动，'d'表示向右移动，'w'表示向
上移动
125.     //新增块位置(new_number_pos)，新增数字(new_number)
126.     int panel[16], new_number_pos[10], new_number[10], i, counter;
127.     char direction[10], c;
128.
129.     //读取游戏信息
130.     counter = 0;
131.     for (i = 0; i < 16; ++i)
```

```
132.         scanf("%d", panel + i);
133.     getchar();
134.     do {
135.         direction[counter] = getchar(); getchar();
136.         new_number_pos[counter] = getchar() - '0'; getchar();
137.         new_number[counter] = getchar() - '0';
138.         counter++;
139.     } while ((c = getchar()) != '\n');
140.
141.     //执行操作
142.     for (i = 0; i < counter; ++i)
143.     {
144.         moving(panel, direction[i]);
145.         add_new_number(panel, new_number_pos[i], new_number[i]);
146.     }
147.
148.     //输出移动后结果
149.     for (i = 0; i < 15; ++i)
150.         printf("%d ", panel[i]);
151.     printf("%d\n", panel[15]);
152.
153.     system("pause");
154.     return 0;
155. }
```

树的建立与基本操作

问题描述

在本实验中，程序的输入是一个表示树结构的广义表。假设树的根为 root，其子树森林 F = (T_1, T_2 , … , T_n)，设与该树对应的广义表为 L，则 L = (原子，子表 1，子表 2, … ，子表 n)，其中原子对应 root，子表 i (1< i <= n) 对应 T_i。例如：广义表 (a,(b,(c),(d)),(f,(g),(h),(i))) 表示的树如图所示：



程序的输出为树的层次结构、树的度以及各种度的结点个数。

在输出树的层次结构时，先输出根结点，然后依次输出各个子树，每个子树向里缩进 4 个空格，如：针对上图表示的树，输出的内容应为：

```
a
    b
        c
        d
    f
        g
        h
        i
Degree of tree: 3
Number of nodes of degree 0: 5
Number of nodes of degree 1: 0
Number of nodes of degree 2: 2
Number of nodes of degree 3: 1
```

例：（下面的黑体为输入）

(a,(b),(c,(d),(e,(g),(h)),(f)))

```
a
    b
   c
        d
        e
            g
            h
        f
Degree of tree: 3
Number of nodes of degree 0: 5
Number of nodes of degree 1: 0
Number of nodes of degree 2: 2
```

Number of nodes of degree 3: 1

参考答案

```c
#include<stdio.h>
#include<stdlib.h>

typedef int Status;
#define OK 1
#define ERROR 0
#define OVERFLOW -2

#define MAX_NUMBER_OF_NODE 50
#define STACK_SIZE 50

//（不可扩充）栈定义
typedef struct {
    int *base, *top;
}Stack;


Status InitStack(Stack &S) {
    //初始化栈
    if(!(S.base = (int *)malloc(STACK_SIZE * sizeof(int)))) exit(OVERFLOW);
    S.top = S.base;
    return OK;
}

Status Push(Stack &S, int e) {
    //进栈
    if (S.top - S.base == STACK_SIZE)   return ERROR;   //上溢
    *S.top++ = e;
    return OK;
}

Status Pop(Stack &S) {
    //出栈
    if (S.top == S.base)    return ERROR;   //下溢
    S.top--;
    return OK;
}

Status GetTop(Stack S, int &e) {
    //取栈顶元素
```

```
41.     if (S.top == S.base)    return ERROR;   //空栈
42.     e = *(S.top - 1);
43.     return OK;
44.}
45.
46.void putchar_with_indent(char c, int space_num) {
47.     //带缩进输出字符，c 为输出的字符，space_num 为字符前面的空格数
48.     int i;
49.     for (i = 0; i < space_num; ++i)
50.         putchar(' ');
51.     printf("%c\n", c);
52.}
53.
54.int max(int a[], int n) {
55.     //求整型数组 a 前 n 个的最大者
56.     int i, max = a[0];
57.     for (i = 1; i < n; ++i)
58.         if (max < a[i])
59.             max = a[i];
60.     return max;
61.}
62.
63.int main() {
64.     char GListStr[100], c; int Degree[MAX_NUMBER_OF_NODE] = { 0 }, num_of_letter = 0
; Stack S; int i, top, degree_sta[20] = { 0 }, degree, degree_of_tree;
65.     //GListStr 为树的广义表表示的字符串，degree[i]表示第 i 个出现的字母对应结点的度数，
num_of_letter 表示字母（结点）的个数
66.     //对第 i 个出现的字母，称 i - 1 为字母的编号。用字母编号而不是字母本身，可以解决字母重
复出现的问题
67.     //S 为用于解决括号层数的栈，本应存放括号，但由于左括号与字母一一对应，因此存放左括号右边
的第一个字母的编号即可
68.     scanf("%s", GListStr);
69.     InitStack(S);
70.
71.     for (i = 0; (c = GListStr[i]) != '\0'; ++i)
72.     {
73.         if (c >= 'a' && c <= 'z') //左括号与字母一一对应，无视左括号，只看左括号右边的第
一个字母
74.         {
75.             putchar_with_indent(c, 4 * (S.top - S.base));
76.             Push(S, num_of_letter); //左括号（右边的第一个字母的编号）进栈
77.             num_of_letter++;
78.         }
79.         if (c == ',')   //统计当前括号层逗号的个数（等于当前结点的度数）
```

```
        {
            GetTop(S, top); //当前结点
            Degree[top]++;
        }
        if (c == ')')   //遇到右括号，销括号
            Pop(S);
    }

    //输出结果
    degree_of_tree = max(Degree, num_of_letter);
    printf("Degree of tree: %d\n", degree_of_tree);
    //degree_sta 用于最后统计每个度数的结点个数。这样做的好处是只需扫描数组 Degree 一遍
    for (i = 0; i < num_of_letter; ++i)
        degree_sta[Degree[i]]++;
    for (degree = 0; degree <= degree_of_tree; ++degree)
        printf("Number of nodes of degree %d: %d\n", degree, degree_sta[degree]);

    system("pause");
    return 0;
}
```

计算 WPL

问题描述

Huffman 编码是通信系统中常用的一种不等长编码，它的特点是：能够使编码之后的电文长度最短。

输入：

第一行为要编码的符号数量 n

第二行 ~ 第 n+1 行为每个符号出现的频率

输出：

对应哈夫曼树的带权路径长度 WPL

参考答案

```
#include<stdio.h>
#include<stdlib.h>

```

```
4. typedef int Status;
5. #define OK 1
6. #define LESS_THAN_TWO_NUMBER 0
7.
8.
9. Status find_two_min_pos(int * num, int * flag, int n, int &pos1, int &pos2) {
10.     //在数组 num 的前 n 个 flag 不为 1 的数中找到最小的两个数，其位置分别存放于 pos1 和 pos2
11.     int i, start;
12.     for (start = 0; start < n && flag[start]; ++start);  //找到第一个 flag 不为 1 的位置，作为遍历的起点 start
13.     if (start >= n - 1)  return LESS_THAN_TWO_NUMBER;     //start == n：没有 flag 不为 1 的数；start == n - 1：只有一个 flag 不为 1 的数
14.
15.     for (i = start + 1, pos1 = start; i < n; ++i)     //第一遍遍历，找到最小的数，其位置存放于 pos1
16.     {
17.         if (flag[i])     continue;    //跳过 flag 为 1 的数
18.         if (num[i] < num[pos1])  pos1 = i;
19.     }
20.     flag[pos1] = 1; //目的：第二遍遍历时跳过此位置，下一次调用函数 find_two_min_pos 时跳过此位置
21.
22.     if (pos1 == start)  //如果原来的 start 正好是最小的数，还需继续向前找新的遍历的起点 start
23.         for (start = pos1; start < n && flag[start]; ++start);
24.     if (start == n) return LESS_THAN_TWO_NUMBER;     //start == n：只有一个 flag 不为 1 的数（即 pos1）
25.     for (i = start + 1, pos2 = start; i < n; ++i)     //第二遍遍历，找到第二小的数，其位置存放于 pos2
26.     {
27.         if (flag[i])     continue;
28.         if (num[i] < num[pos2])  pos2 = i;
29.     }
30.     flag[pos2] = 1; //目的：下一次调用函数 find_two_min_pos 时跳过此位置
31.
32.     return OK;
33.}
34.
35.int main() {
36.     int n, freq[2][100] = { 0 }, i, pos1, pos2, wpl = 0;
37.     //数组 freq 第一行存放这 n 个数（结点），第二行作为 flag（用于确定函数 find_two_min_pos 的查找范围。flag 为 1 表示跳过，0 表示不跳过）
38.     scanf("%d", &n);
39.     for (i = 0; i < n; ++i)
```

```
        scanf("%d", &freq[0][i]);

    for (; find_two_min_pos(freq[0], freq[1], n, pos1, pos2); ++n)  //结束条件：find_two_min_pos 返回 LESS_THAN_TWO_NUMBER
        freq[0][n] = freq[0][pos1] + freq[0][pos2]; //新结点放在最后面，值最小的两个的和。通过 n++扩大 freq 的范围把新结点包含到下一次 find_two_min_pos 的范围内

    //计算 wpl：把所有结点（除去根结点）的 freq 值求和
    for (i = 0; i < n - 1; ++i)
        wpl += freq[0][i];
    printf("WPL=%d\n", wpl);

    system("pause");
    return 0;
}
```

连连看游戏辅助

问题描述

《连连看》是由一款益智类游戏。

《连连看》只要将相同的两张牌用三根以内的直线连在一起就可以消除，规则简单容易上手。

本题编写程序模拟人进行连连看游戏，电脑模拟人玩连连看的过程如下：

1、分析本局游戏，将本局游戏转化成一个二位数组。0 表示空白区域，数字表示该游戏牌为出现在游戏中的第几类游戏牌。

2、利用广度优先搜索算法，判断两个游戏牌是否可以消除。

3、程序模拟人点击可以消除的一对游戏牌进行消除。

对某局游戏转化成二维数组如下：

0 0 0 0 0

0 1 2 0 0

0 0 3 4 0

0 0 0 1 0

输入二位数组的行、列，二位数组、起始元素的的坐标、终止元素的坐标，判断起始游戏牌和终止游戏牌是否可以消除。

对上例输入

4 5
0 0 0 0 0　0 1 2 0 0　0 0 3 4 0　0 0 0 1 0
1 1
3 3
输出 TRUE

参考答案

```
#include<stdio.h>
#include<stdlib.h>

typedef int Status;
#define OK 1
#define ERROR 0
#define OVERFLOW -2
#define UNDERFLOW -3

typedef int BOOL;
#define TRUE 1
#define FALSE 0

typedef int PanelElemType;  //游戏面板元素类型，一般用整型数据标记
typedef struct {
    PanelElemType **panel;  //存储游戏面板信息的二维数组
    int rownum, colnum; //二维数组的行数、列数
}Panel; //游戏面板类型
typedef struct {
    int row;    //行坐标
    int col;    //列坐标
}Position;  //位置（坐标）类型
typedef enum { UP, DOWN, LEFT, RIGHT, UP_AND_DOWN, LEFT_AND_RIGHT }DIRECTION;   //方向类型

Position aim = { -1, -1 };  //全局变量，用于存放可与第一个位置消去的位置

/* 以下为循环队列定义及操作 */
#define MAX_QUEUE_SIZE 100
typedef struct {
    Position pos;
    DIRECTION limited_dir;
}QElemType;
typedef struct {
    QElemType *base;
```

```
35.     int front, rear;
36. }Queue;
37. Queue Q;     //全局变量
38. Status InitQueue(Queue &Q) {
39.     //初始化队列
40.     Q.base = (QElemType *)malloc(MAX_QUEUE_SIZE * sizeof(QElemType));
41.     if (!Q.base)    exit(OVERFLOW);
42.     Q.front = 0; Q.rear = 0;
43.     return OK;
44. }
45. Status EnQueue(Queue &Q, QElemType e) {
46.     //入队
47.     if ((Q.rear + 1) % MAX_QUEUE_SIZE == Q.front)   return OVERFLOW;
48.     Q.base[Q.rear] = e;
49.     Q.rear = (Q.rear + 1) % MAX_QUEUE_SIZE;
50.     return OK;
51. }
52. Status DeQueue(Queue &Q, QElemType &e) {
53.     //出队
54.     if (Q.rear == Q.front)  return UNDERFLOW;
55.     e = Q.base[Q.front];
56.     Q.front = (Q.front + 1) % MAX_QUEUE_SIZE;
57.     return OK;
58. }
59. /* 以上为循环队列定义及操作 */
60.
61. PanelElemType GetPanelElement(Panel panel, Position pos) {
62.     //根据坐标取面板元素
63.     return panel.panel[pos.row][pos.col];
64. }
65.
66. BOOL EliminateStraightIf(Panel panel, Position pos, DIRECTION dir, PanelElemType val
ue, int &potential_path_num) {
67.     //判断面板 pos1 位置在 dir 方向是否存在可直线消去 pos1_value 的元素
68.     //若存在，返回 TRUE；否则返回 FALSE
69.     //如果不存在，则将该方向中间的 0 的个数存放于 potential_path_num
70.     int i, j;
71.     potential_path_num = 0; //计数器归零
72.     switch (dir)
73.     {
74.     case UP:
75.         for (j = pos.col, i = pos.row - 1; i >= 0; --i)  //向上扫描
76.             if (!GetPanelElement(panel, { i, j }))  //如果没有遇到非 0 位置
77.             {
```

```
78.                 EnQueue(Q, { { i, j }, LEFT_AND_RIGHT });   //将中间的 0 的位置（潜在路径的拐弯位置）入队，同时将限制的下一遍扫描方向也入队。限制的下一遍扫描方向与此方向垂直
79.                 potential_path_num++;
80.             }
81.             else //遇到非 0 位置
82.             {
83.                 if (GetPanelElement(panel, { i, j }) == value)
84.                 {
85.                     aim = { i, j }; //判断是否相等（可消去），如果相等，将消去位置存入 aim 并返回 TRUE，整个程序结束（main 函数里设置了 goto 语句）
86.                     return TRUE;
87.                 }
88.
89.                 else return FALSE;  //否则返回 FALSE
90.             }
91.         break;
92.     case DOWN:
93.         for (j = pos.col, i = pos.row + 1; i < panel.rownum; ++i)
94.             if (!GetPanelElement(panel, { i, j })) { EnQueue(Q, { { i, j }, LEFT_AND_RIGHT }); potential_path_num++; }
95.             else { if (GetPanelElement(panel, { i, j }) == value) { aim = { i, j }; return TRUE; } else return FALSE; }
96.         break;
97.     case LEFT:
98.         for (i = pos.row, j = pos.col - 1; j >= 0; --j)
99.             if (!GetPanelElement(panel, { i, j })) { EnQueue(Q, { { i, j }, UP_AND_DOWN }); potential_path_num++; }
100.            else { if (GetPanelElement(panel, { i, j }) == value) { aim = { i, j }; return TRUE; } else return FALSE; }
101.        break;
102.    case RIGHT:
103.        for (i = pos.row, j = pos.col + 1; j < panel.colnum; ++j)
104.            if (!GetPanelElement(panel, { i, j })) { EnQueue(Q, { { i, j }, UP_AND_DOWN }); potential_path_num++; }
105.            else { if (GetPanelElement(panel, { i, j }) == value) { aim = { i, j }; return TRUE; } else return FALSE; }
106.        break;
107.    default:
108.        break;
109.    }
110.    return FALSE;   //没有遇到非 0 位置就扫描到边界，返回 FALSE
111.}
112.
113.int main() {
```

```
114.     Panel panel; int i, j, k; Position pos1, pos2; BOOL pos1Eliminatepos2If;
115.
116.     //读取游戏面板信息
117.     scanf("%d%d", &panel.rownum, &panel.colnum);
118.     panel.panel = (PanelElemType **)malloc(panel.rownum * sizeof(PanelElemType *));
119.     for (i = 0; i < panel.rownum; ++i)
120.     {
121.         panel.panel[i] = (PanelElemType *)malloc(panel.colnum * sizeof(PanelElemTyp
e));
122.         for (j = 0; j < panel.colnum; ++j)
123.             scanf("%d", &panel.panel[i][j]);
124.     }
125.     scanf("%d%d", &pos1.row, &pos1.col);
126.     scanf("%d%d", &pos2.row, &pos2.col);
127.
128.     //程序主体
129.     //预检查
130.     PanelElemType pos1_value = GetPanelElement(panel, pos1), pos2_value = GetPanelE
lement(panel, pos2);
131.     if (!pos1_value || !pos2_value || pos1_value != pos2_value) { pos1Eliminatepos2
If = FALSE; goto end; }
132.
133.     Position pos_turn_first, pos_turn_second; QElemType qe; DIRECTION limited_dir;
134.     int potential_UP_path_num, potential_DOWN_path_num, potential_LEFT_path_num, po
tential_RIGHT_path_num, potential_STRAIGHT_path_num, potential_TURN_ONCE_path_num, sum;
135.     pos1Eliminatepos2If = FALSE;
136.     InitQueue(Q);
137.
138.     /* 按拐弯次数广度优先遍历 */
139.
140.     //检查是否可以直线（拐弯 0 次）消去
141.     if (EliminateStraightIf(panel, pos1, UP, pos1_value, potential_UP_path_num) ||
EliminateStraightIf(panel, pos1, DOWN, pos1_value, potential_DOWN_path_num) || EliminateSt
raightIf(panel, pos1, LEFT, pos1_value, potential_LEFT_path_num) || EliminateStraightIf(pa
nel, pos1, RIGHT, pos1_value, potential_RIGHT_path_num)) //四个方向都扫描
142.         if (pos2.row == aim.row && pos2.col == aim.col)
143.         {
144.             pos1Eliminatepos2If = TRUE;
145.             goto end;   //跳出二重循环
146.         }
```

```
147.     potential_STRAIGHT_path_num = potential_UP_path_num + potential_DOWN_path_num + potential_LEFT_path_num + potential_RIGHT_path_num; //如果不可直线消去，统计潜在路径个数，为下一遍扫描做准备
148.
149.     //检查是否可以拐弯 1 次消去
150.     potential_TURN_ONCE_path_num = 0;
151.     for (k = 0; k < potential_STRAIGHT_path_num; ++k)    //遍历上一遍扫描得到的潜在路径，共 potential_STRAIGHT_path_num 个
152.     {
153.         DeQueue(Q, qe); pos_turn_first = qe.pos; limited_dir = qe.limited_dir;  //前 potential_STRAIGHT_path_num 个出队的元素为潜在路径的第一次拐弯位置（包含限制的扫描方向）
154.         switch (limited_dir)
155.         {
156.         case UP_AND_DOWN:
157.             if (EliminateStraightIf(panel, pos_turn_first, UP, pos1_value, potential_UP_path_num) || EliminateStraightIf(panel, pos_turn_first, DOWN, pos1_value, potential_DOWN_path_num)) //只上下扫描
158.                 if (pos2.row == aim.row && pos2.col == aim.col)
159.                 {
160.                     pos1Eliminatepos2If = TRUE;
161.                     goto end;
162.                 }
163.             sum = potential_UP_path_num + potential_DOWN_path_num;
164.             break;
165.         case LEFT_AND_RIGHT:
166.             if (EliminateStraightIf(panel, pos_turn_first, LEFT, pos1_value, potential_LEFT_path_num) || EliminateStraightIf(panel, pos_turn_first, RIGHT, pos1_value, potential_RIGHT_path_num))   //只左右扫描
167.                 if (pos2.row == aim.row && pos2.col == aim.col)
168.                 {
169.                     pos1Eliminatepos2If = TRUE;
170.                     goto end;
171.                 }
172.             sum = potential_LEFT_path_num + potential_RIGHT_path_num;
173.             break;
174.         default:
175.             break;
176.         }
177.         potential_TURN_ONCE_path_num += sum;    //如果不可拐弯 1 次消去，统计潜在路径个数，为下一遍扫描做准备
178.     }
179.     //检查是否可以拐弯 2 次消去
180.     for (k = 0; k < potential_TURN_ONCE_path_num; ++k)   //遍历上一遍扫描得到的潜在路径，共 potential_TURN_ONCE_path_num 个
```

```
    {
        DeQueue(Q, qe); pos_turn_second = qe.pos; limited_dir = qe.limited_dir; //前 potential_STRAIGHT_path_num 个出队的元素为潜在路径的第二次拐弯位置（包含限制的扫描方向）
        switch (limited_dir)
        {
        case UP_AND_DOWN:
            if (EliminateStraightIf(panel, pos_turn_second, UP, pos1_value, potential_UP_path_num) || EliminateStraightIf(panel, pos_turn_second, DOWN, pos1_value, potential_DOWN_path_num))
                if (pos2.row == aim.row && pos2.col == aim.col)
                {
                    pos1Eliminatepos2If = TRUE;
                    goto end;
                }
            break;
        case LEFT_AND_RIGHT:
            if (EliminateStraightIf(panel, pos_turn_second, LEFT, pos1_value, potential_LEFT_path_num) || EliminateStraightIf(panel, pos_turn_second, RIGHT, pos1_value, potential_RIGHT_path_num))
                if (pos2.row == aim.row && pos2.col == aim.col)
                {
                    pos1Eliminatepos2If = TRUE;
                    goto end;
                }
            break;
        default:
            break;
        }
        //注：最后一遍扫描无需统计潜在路径个数
    }
    end:;

    //输出结果
    if (pos1Eliminatepos2If)    printf("TRUE\n");
    else printf("FALSE\n");

    system("pause");
    return 0;
}
```

平衡二叉树

问题描述

程序输入一个字符串（只包含小写字母），请按照字符的输入顺序建立平衡二叉排序树，并分别输出二叉树的先序序列、中序序列和后序序列，最后输出该二叉树向左旋转 90 度后的结构。

例如：向左旋转 90 度后，以每层向里缩进 4 个空格的方式输出，输出结果为：

```
        i
    g
        f
a
        d
    c
        b
```

输入：agxnzyimk

输出：

Preorder: xigamknzy

Inorder: agikmnxyz

Postorder: agknmiyzx

Tree:

```
    z
        y
x
            n
        m
            k
    i
        g
            a
```

参考答案

```
#include<stdio.h>
#include<stdlib.h>

typedef unsigned char Status;
#define FOUND 1
```

```
6. #define NOT_FOUND 0
7.
8. //平衡二叉树定义（带双亲域的二叉链表）
9. typedef struct BiTNode {
10.     char data;
11.     int bf; //平衡因子 = 左子树深度 - 右子树深度
12.     BiTNode *lchild, *rchild, *parent;
13.}BiTNode, *BiTree;
14.typedef enum { N, L, R } Direction;
15.
16.
17.void PreOrderTraverse(BiTree &T) {
18.     //先序（先左后右）遍历二叉树（递归算法），打印先序遍历序列
19.     if (!T) return;
20.     printf("%c", T->data);
21.     PreOrderTraverse(T->lchild);
22.     PreOrderTraverse(T->rchild);
23.}
24.
25.void InOrderTraverse1(BiTree &T) {
26.     //中序（先左后右）遍历二叉树（递归算法），打印中序遍历序列
27.     if (!T) return;
28.     InOrderTraverse1(T->lchild);
29.     printf("%c", T->data);
30.     InOrderTraverse1(T->rchild);
31.}
32.
33.void PostOrderTraverse(BiTree &T) {
34.     //后序（先左后右）遍历二叉树（递归算法），打印后序遍历序列
35.     if (!T) return;
36.     PostOrderTraverse(T->lchild);
37.     PostOrderTraverse(T->rchild);
38.     printf("%c", T->data);
39.}
40.
41.void InOrderTraverse2(BiTree &T, int current_level) {
42.     //中序（先右后左）遍历二叉树（递归算法），打印二叉树的凹入表示
43.     //current_level 用于记录当前子树的层数
44.     if (!T) return;
45.     InOrderTraverse2(T->rchild, current_level + 1);
46.     for (int i = 0; i < 4 * (current_level - 1); ++i)    printf(" ");    //根据当前层
数确定打印缩进量
47.     printf("%c\n", T->data);
48.     InOrderTraverse2(T->lchild, current_level + 1);
```

```
49.}
50.
51.int BiTreeDepth(BiTree T) {
52.    //求二叉树的深度（递归算法）
53.    if (!T) return 0;
54.    int ldepth = BiTreeDepth(T->lchild), rdepth = BiTreeDepth(T->rchild);
55.    return ((ldepth >= rdepth) ? ldepth : rdepth) + 1;
56.}
57.
58.Status SearchBST(BiTree BST, char key, BiTNode *&insert_aim_node, Direction &L_or_R)
 {
59.    //查找二叉排序树 BBST 中是否存在关键字为 key 的记录，非递归算法
60.    //若存在，返回 FOUND；否则，返回 NOT_FOUND，待插入位置存放于 insert_aim_node 和
L_or_R
61.    if (!BST)   return NOT_FOUND;
62.    BiTNode *p = BST;
63.    while (1)
64.    {
65.        if (key == p->data) { insert_aim_node = NULL; return FOUND; }
66.        if (key < p->data)
67.        {
68.            if (!p->lchild) { insert_aim_node = p; L_or_R = L; return NOT_FOUND; }
69.            else p = p->lchild;
70.        }
71.        if (key > p->data)
72.        {
73.            if (!p->rchild) { insert_aim_node = p; L_or_R = R; return NOT_FOUND; }
74.            else p = p->rchild;
75.        }
76.    }
77.}
78.
79.void L_Rotate(BiTree &p, BiTree &BST) {
80.    //对二叉排序树 BBST 的 p 子树作左旋处理，并链接到原来的位置。处理结束后 p 指向该子树新的
根
81.    BiTNode *Parent = p->parent, *q = p; p = p->rchild;   //Parent 总是指向该子树的双
亲结点，q 总是指向该子树原来的根，p 总是指向该子树新的根
82.    BiTree hang = p->lchild; //需要摘下来挂到对面的子树
83.    //重新链接相关结点
84.    p->lchild = q;   //颠倒 p,q 的亲子关系
85.    q->rchild = hang;    //挂到对面
86.    //更新相关结点的 parent 域
87.    p->parent = Parent;
88.    q->parent = p;
```

```
89.     if(hang)    hang->parent = q;
90.     //链接到原来的位置
91.     if (Parent)
92.     {
93.         if (Parent->lchild == q) Parent->lchild = p;  //原来是左孩子，就链接到左孩子的
位置
94.         if (Parent->rchild == q) Parent->rchild = p;  //原来是右孩子，就链接到右孩子的
位置
95.     }
96.     else BST = p;   //双亲不存在（处理的子树就是 BBST 本身），直接将 p 赋值给 BBST
97.     //更新平衡因子发生改变的结点的 bf 域
98.     q->bf = 0; p->bf = 0; //p 结点平衡因子由-2 变为 0，q 结点平衡因子由-1 变为 0
99.     while (Parent)  //所有祖先结点平衡因子都可能发生变化
100.     {
101.         Parent->bf = BiTreeDepth(Parent->lchild) - BiTreeDepth(Parent->rchild);
//重新计算平衡因子
102.         Parent = Parent->parent;
103.     }
104. }
105.
106. void R_Rotate(BiTree &p, BiTree &BST) {
107.     //右旋处理，与左旋类似
108.     BiTNode *Parent = p->parent, *q = p; p = p->lchild;
109.     BiTree hang = p->rchild;
110.     p->rchild = q;
111.     q->lchild = hang;
112.     p->parent = Parent;
113.     q->parent = p;
114.     if (hang)   hang->parent = q;
115.     if (Parent)
116.     {
117.         if (Parent->lchild == q) Parent->lchild = p;
118.         if (Parent->rchild == q) Parent->rchild = p;
119.     }
120.     else BST = p;
121.     q->bf = 0; p->bf = 0;
122.     while (Parent)
123.     {
124.         Parent->bf = BiTreeDepth(Parent->lchild) - BiTreeDepth(Parent->rchild);
125.         Parent = Parent->parent;
126.     }
127. }
128.
129. void InsertBBST(BiTree &BBST, char &insert_elem) {
```

```
130.     //在平衡二叉树 BBST 中插入关键字为 insert_elem 的记录（查找不成功时）
131.     if (!BBST)  //如果树为空，直接建立根结点
132.     {
133.         BBST = (BiTNode *)malloc(sizeof(BiTNode));
134.         BBST->data = insert_elem; BBST->lchild = BBST->rchild = NULL; BBST->bf = 0;
 BBST->parent = NULL;
135.         return;
136.     }
137.     BiTNode *insert_aim_node, *new_node; Direction L_or_R;
138.     if (!SearchBST(BBST, insert_elem, insert_aim_node, L_or_R)) //查找不成功时
139.     {
140.         //在 SearchBST 找到的相应位置插入记录
141.         new_node = (BiTNode *)malloc(sizeof(BiTNode));
142.         new_node->data = insert_elem; new_node->lchild = new_node->rchild = NULL; n
ew_node->parent = insert_aim_node; new_node->bf = 0;
143.         if (L_or_R == L)    insert_aim_node->lchild = new_node;
144.         else insert_aim_node->rchild = new_node;
145.         //更新平衡因子发生改变的结点（所有祖先结点）的 bf 域，并找到因插入结点而失去平衡的
最小子树根结点 unblcd_tree
146.         BiTNode *p = new_node; BiTree unblcd_tree = NULL; int find_unblcd = 0; Dire
ction dp = N, dpp;
147.         while (p->parent)
148.         {
149.             if (p->parent->lchild == p)
150.             {
151.                 if (!find_unblcd) { dpp = dp; dp = L; } //不断记录先后两个拐弯方向，
当找到第一个不平衡结点时停止记录
152.                 p = p->parent;
153.                 if ((p->bf = BiTreeDepth(p->lchild) - BiTreeDepth(p->rchild)) >= 2
&& !find_unblcd)
154.                 {
155.                     unblcd_tree = p;
156.                     find_unblcd = 1;
157.                 }
158.             }
159.             else
160.             {
161.                 if (!find_unblcd) { dpp = dp; dp = R; }
162.                 p = p->parent;
163.                 if ((p->bf = BiTreeDepth(p->lchild) - BiTreeDepth(p->rchild)) <= -
2 && !find_unblcd)
164.                 {
165.                     unblcd_tree = p;
166.                     find_unblcd = 1;
```

```
167.                 }
168.             }
169.         }
170.         //如果找到了不平衡的结点，则需旋转处理
171.         if (find_unblcd)
172.         {
173.             if (dp == L && dpp == L)    //LL 型不平衡：右旋
174.                 R_Rotate(unblcd_tree, BBST);
175.             if (dp == R && dpp == R)    //RR 型不平衡：左旋
176.                 L_Rotate(unblcd_tree, BBST);
177.             if (dp == L && dpp == R)    //LR 型不平衡：先左旋再右旋
178.             {
179.                 L_Rotate(unblcd_tree->lchild, BBST);
180.                 R_Rotate(unblcd_tree, BBST);
181.             }
182.             if (dp == R && dpp == L)    //RL 型不平衡：先右旋再左旋
183.             {
184.                 R_Rotate(unblcd_tree->rchild, BBST);
185.                 L_Rotate(unblcd_tree, BBST);
186.             }
187.         }
188.     }
189. }
190.
191. int main() {
192.     BiTree BBST = NULL; char insert_elem;
193.     while (1)   //建立平衡二叉树 BBST
194.     {
195.         insert_elem = getchar();
196.         if (insert_elem == '\n')    break;
197.         InsertBBST(BBST, insert_elem);
198.     }
199.     printf("Preorder: "); PreOrderTraverse(BBST); printf("\n");
200.     printf("Inorder: "); InOrderTraverse1(BBST); printf("\n");
201.     printf("Postorder: "); PostOrderTraverse(BBST); printf("\n");
202.     printf("Tree:\n"); InOrderTraverse2(BBST, 1);
203.
204.     system("pause");
205.     return 0;
206. }
```

排序二叉树

问题描述

建立并中序遍历一个排序二叉树。

排序二叉树是指左子树的所有节点的值均小于它根节点的值,右子树的所有节点的值均大于它根节点的值。

输入:

输入有一行，表示若干个要排序的数，输入 0 时停止

输出

二叉树的凹入表示和二叉树的中序遍历序列

例

输入

56 78 34 89 12 35 67 77 22 57 0

输出

```
        12
            22
    34
        35
56
            57
        67
            77
    78
        89
12 22 34 35 56 57 67 77 78 89
```

参考答案

```c
#include<stdio.h>
#include<stdlib.h>

typedef unsigned char Status;
#define FOUND 1
#define NOT_FOUND 0

typedef struct BiTNode {
    int data;
    BiTNode *lchild, *rchild;
```

```
11.}BiTNode, *BiTree;
12.#define L 1
13.#define R 2
14.
15.int BST_node_num, K, *InOrderTraverseSequence;  //全局变量，用于存储中序遍历序列
16.
17.void InOrderTraverse(BiTree &T, int current_level) {
18.    //中序遍历二叉树（递归算法），打印二叉树的凹入表示，并存储中序遍历序列
19.    //current_level 用于记录当前子树的层数
20.    if (!T) return;
21.    InOrderTraverse(T->lchild, current_level + 1);
22.    for (int i = 0; i < 4 * (current_level - 1); ++i)    printf(" ");    //根据当前层数确定打印缩进量
23.    printf("%d\n", T->data);
24.    InOrderTraverseSequence[K++] = T->data;  //存储中序遍历序列
25.    InOrderTraverse(T->rchild, current_level + 1);
26.}
27.
28.Status SearchBST(BiTree BST, int key, BiTNode *&insert_aim_node, int &L_or_R) {
29.    //查找二叉排序树 BST 中是否存在关键字为 key 记录，非递归算法
30.    //若存在，返回 FOUND；否则，返回 NOT_FOUND，待插入位置存放于 insert_aim_node 和 L_or_R
31.    if (!BST)   return NOT_FOUND;
32.    BiTNode *node_ptr = BST;
33.    while (1)
34.    {
35.        if (key == node_ptr->data)  { insert_aim_node = NULL; return FOUND; }
36.        if (key < node_ptr->data)
37.        {
38.            if (!node_ptr->lchild) { insert_aim_node = node_ptr; L_or_R = L; return NOT_FOUND; }
39.            else node_ptr = node_ptr->lchild;
40.        }
41.        if (key > node_ptr->data)
42.        {
43.            if (!node_ptr->rchild) { insert_aim_node = node_ptr; L_or_R = R; return NOT_FOUND; }
44.            else node_ptr = node_ptr->rchild;
45.        }
46.    }
47.}
48.
49.void InsertBST(BiTree &BST, int &insert_elem) {
50.    //在二叉排序树 BST 中插入关键字为 insert_elem 的记录（查找不成功时）
```

```
51.     if (!BST) { BST = (BiTNode *)malloc(sizeof(BiTNode)); BST->data = insert_elem; B
ST->lchild = BST->rchild = NULL; return; }
52.     BiTNode *insert_aim_node, *new_node; int L_or_R;
53.     if (!SearchBST(BST, insert_elem, insert_aim_node, L_or_R))
54.     {
55.         new_node = (BiTNode *)malloc(sizeof(BiTNode));
56.         new_node->data = insert_elem; new_node->lchild = new_node->rchild = NULL;
57.         if (L_or_R == L)    insert_aim_node->lchild = new_node;
58.         else insert_aim_node->rchild = new_node;
59.     }
60.}
61.
62.int main() {
63.     BiTree BST = NULL; int insert_elem;
64.
65.     BST_node_num = 0;
66.     while (1)
67.     {
68.         scanf("%d", &insert_elem);
69.         if (!insert_elem)   break;
70.         InsertBST(BST, insert_elem); BST_node_num++;
71.     }
72.     InOrderTraverseSequence = (int *)malloc(BST_node_num * sizeof(int)); K = 0;
73.     InOrderTraverse(BST, 1);
74.     printf("\n");
75.     for (int i = 0; i < BST_node_num; ++i)
76.         printf(" %d", InOrderTraverseSequence[i]);
77.     printf("\n");
78.
79.     system("pause");
80.     return 0;
81.}
```

无向图的各连通分支

问题描述

求解无向图的各连通分支。

输入：

第一行为图的节点数 n（节点编号 0 至 n-1，0<n<=10）

从第二行开始列出图的边，-1 表示输入结束

输出：

输出每个连通分支的广度优先搜索序列（从连通分支的最小编号开始），不同分支以最小编号递增顺序列出

例

输入

8

0 5

5 2

4 5

5 6

6 2

3 7

0 2

-1

输出

0-5-2-4-6

1

3-7

参考答案

```c
#include<stdio.h>
#include<stdlib.h>

typedef unsigned char BOOL;
#define TRUE 1
#define FALSE 0

typedef int VertexType;
//无向图（UDG）的边集数组表示
typedef struct {
    int adjvex1, adjvex2;
}Edge;  //边类型
typedef struct {
    VertexType *vexs;   //顶点向量
    BOOL *visited;  //访问标记
```

```
16.     Edge *edges;    //边集数组
17.     int vexnum, edgenum;    //顶点数、弧数
18.}MUDG;  //无向图类型
19.
20./* 以下为循环队列定义及操作 */
21.typedef int Status;
22.#define OK 1
23.#define ERROR 0
24.#define OVERFLOW -2
25.#define UNDERFLOW -3
26.
27.typedef int QElemType;  //队列中存储顶点的编号（int 型）
28.#define MAX_QUEUE_SIZE 100
29.typedef struct {
30.     QElemType *base;
31.     int front, rear;
32.}Queue;
33.Queue Q;    //全局变量
34.
35.Status InitQueue(Queue &Q) {
36.     //初始化队列
37.     Q.base = (QElemType *)malloc(MAX_QUEUE_SIZE * sizeof(QElemType));
38.     if (!Q.base)    exit(OVERFLOW);
39.     Q.front = 0; Q.rear = 0;
40.     return OK;
41.}
42.
43.BOOL QueueEmpty(Queue Q) {
44.     //队列判空
45.     if (Q.rear == Q.front)  return TRUE;
46.     else return FALSE;
47.}
48.
49.Status EnQueue(Queue &Q, QElemType e) {
50.     //入队
51.     if ((Q.rear + 1) % MAX_QUEUE_SIZE == Q.front)   return OVERFLOW;
52.     Q.base[Q.rear] = e;
53.     Q.rear = (Q.rear + 1) % MAX_QUEUE_SIZE;
54.     return OK;
55.}
56.
57.Status DeQueue(Queue &Q, QElemType &e) {
58.     //出队
59.     if (Q.rear == Q.front)  return UNDERFLOW;
```

```
60.     e = Q.base[Q.front];
61.     Q.front = (Q.front + 1) % MAX_QUEUE_SIZE;
62.     return OK;
63.}
64./* 以上为循环队列定义及操作 */
65.
66.void CreateMUDG(MUDG &G) {
67.     //建立无向图
68.     /*输入格式:
69.     顶点数
70.     边 1 依附顶点 边 1 依附顶点
71.     边 2 依附顶点 边 2 依附顶点
72.     ...
73.     -1
74.     */
75.     int i, v1, v2;
76.     //输入顶点数
77.     scanf("%d", &G.vexnum);
78.     //初始化顶点向量、邻接矩阵
79.     G.vexs = (VertexType *)malloc(G.vexnum * sizeof(VertexType));
80.     for (i = 0; i < G.vexnum; i++)   G.vexs[i] = i;  //顶点数据即为顶点编号
81.     //输入边
82.     G.edgenum = 0;
83.     scanf("%d", &v1);
84.     if (v1 == -1)   return;
85.     G.edges = (Edge *)malloc(sizeof(Edge));
86.     do{
87.         G.edges = (Edge *)realloc(G.edges, (G.edgenum + 1) * sizeof(Edge));
88.         scanf("%d", &v2);
89.         G.edges[G.edgenum].adjvex1 = v1; G.edges[G.edgenum].adjvex2 = v2; G.edgenum+
+;
90.         scanf("%d", &v1);
91.     } while (v1 != -1);
92.}
93.
94.BOOL VexVisitedIf(MUDG G, int vex_index) {
95.     //判断无向图 G 中的顶点是否被访问过
96.     //顶点编号为 vex_index，如果已被访问，返回 TRUE，否则返回 FALSE
97.     if (G.visited[vex_index])   return TRUE;
98.     else return FALSE;
99.}
100.
101.BOOL AllVexVisitedIf(MUDG G, int &unvisited_vex_index) {
102.     //判断无向图 G 中是否有未被访问的顶点
```

```
    //如果有，将编号最小的未被访问顶点存放于 unvisited_vex_index，返回 FALSE；否则返回 TRUE
    for (int i = 0; i < G.vexnum; ++i)
        if (!VexVisitedIf(G, i))
        {
            unvisited_vex_index = i;
            return FALSE;
        }
    return TRUE;
}

int cmp(const void *a, const void *b) {
    //qsort 所需的比较函数
    return *(int*)a - *(int*)b;
}

void GetAllAdjVex(MUDG G, int vex_index, int *adjvex_index, int &adjvex_num) {
    //寻找无向图 G 中某顶点的所有邻接点
    //顶点编号为 vex_index，邻接点编号存放于数组 adjvex_index，邻接点个数存放于 adjvex_num
    int i; adjvex_num = 0;
    for (i = 0; i < G.edgenum; ++i)
    {
        if (G.edges[i].adjvex1 == vex_index)
            adjvex_index[adjvex_num++] = G.edges[i].adjvex2;
        if (G.edges[i].adjvex2 == vex_index)
            adjvex_index[adjvex_num++] = G.edges[i].adjvex1;
    }
    qsort(adjvex_index, adjvex_num, sizeof(int), cmp);  //对邻接点编号从小到大排序（采用快速排序库函数 qsort）
}

void PrintBFS(MUDG G) {
    //格式化输出图 G 的广度优先遍历序列
    int i, vex_index, unvisited_vex_index, *adjvex_index = (int *)malloc(G.vexnum * sizeof(int)), adjvex_num;
    //初始化各顶点的访问标记
    G.visited = (BOOL *)malloc(G.vexnum * sizeof(BOOL));
    for (i = 0; i < G.vexnum; ++i)   G.visited[i] = FALSE;

    InitQueue(Q);
    while (!AllVexVisitedIf(G, unvisited_vex_index))    //终止条件：图中没有访问的顶点
    {
```

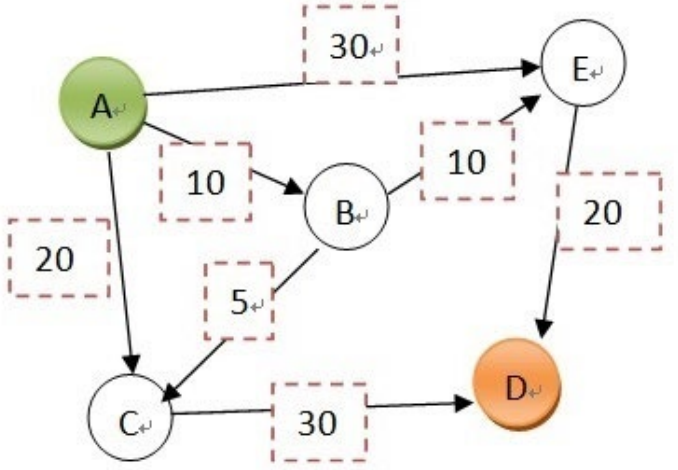
```
142.         printf("%d", G.vexs[unvisited_vex_index]); G.visited[unvisited_vex_index] =
 TRUE;   //访问该顶点并标记已访问
143.         //广度优先遍历此连通分量中的其他顶点
144.         vex_index = unvisited_vex_index;
145.         while (1)
146.         {
147.             //访问 vex_index（称为“开花点”）的所有未被访问的邻接点，并标记已访问（注：此
时“开花点”已访问）
148.             GetAllAdjVex(G, vex_index, adjvex_index, adjvex_num);
149.             for (i = 0; i < adjvex_num; ++i)
150.                 if (!VexVisitedIf(G, adjvex_index[i]))
151.                 {
152.                     printf("-%d", G.vexs[adjvex_index[i]]); G.visited[adjvex_index[
i]] = TRUE;
153.                     EnQueue(Q, adjvex_index[i]);
154.                 }
155.             if (QueueEmpty(Q))  break;  //队空时结束
156.             DeQueue(Q, vex_index);  //出队，作为下一次的开花点
157.         }
158.         printf("\n");
159.     }
160. }
161.
162. int main() {
163.     MUDG G;
164.     CreateMUDG(G);
165.     PrintBFS(G);
166.
167.     system("pause");
168.     return 0;
169. }
```

求两点之间的最短路径

问题描述

最短路径问题是经典图论问题之一。从工程意义上讲，最短路径问题是对大量工程问题的直观抽象。

最典型的例子是在地图上寻找最短驾车路径。



寻找从 A 到 D 的最短路径。

参考答案

```
#include<stdio.h>
#include<stdlib.h>

typedef char * VertexType;
#define INFINITY 100000000
//带权有向图（DN）的邻接矩阵表示
typedef struct {
    VertexType *vexs;    //顶点向量
    int **arcs; //邻接矩阵
    int vexnum, arcnum; //顶点数、弧数
}MDN;    //带权有向图类型

//路径的单链表表示（不带头结点）
typedef struct Node {
    int vex_index;  //路径中顶点的编号
    struct Node *next;
}PathNode;  //链表结点类型
typedef struct
{
    PathNode *head, *tail;  //链表的头指针、尾指针
}PathLinkList;  //链表类型
typedef struct {
    int source, destination;     //路径起点顶点、终点顶点的编号
    PathLinkList path;  //路径链表
    int dist;    //路径长度
}Path;  //路径类型

```

```
28.void CreateMDN(MDN &G) {
29.    //建立带权有向图
30.    /*输入格式:
31.    顶点数,边数
32.    顶点 1,顶点 2,...,顶点 n
33.    <弧 1 起点,弧 1 终点,弧 1 权>,<弧 2 起点,弧 2 终点,弧 2 权>,...,<弧 e 起点,弧 e 终点,弧 e 权>
34.    */
35.    int i, j, v1, v2, weight;
36.    //输入顶点数、弧数
37.    scanf("%d,%d", &G.vexnum, &G.arcnum); getchar();
38.    //初始化顶点向量、邻接矩阵
39.    G.vexs = (VertexType *)malloc(G.vexnum * sizeof(VertexType));
40.    G.arcs = (int **)malloc(G.vexnum * sizeof(int *));
41.    for (i = 0; i < G.vexnum; ++i)
42.    {
43.        G.arcs[i] = (int *)malloc(G.vexnum * sizeof(int));
44.        for (j = 0; j < G.vexnum; ++j)
45.            G.arcs[i][j] = INFINITY;
46.    }
47.    //输入顶点
48.    char vexs_str[1000]; int comma_pos = 0, head;
49.    scanf("%s", vexs_str); getchar();   //存储顶点格式化输入的字符串
50.    for (i = 0, head = 0; i < G.vexnum; ++i) //字符串处理
51.    {
52.        while (vexs_str[comma_pos] != ',' && vexs_str[comma_pos] != '\0')   comma_po
s++;
53.        G.vexs[i] = (char *)malloc((comma_pos - head) * sizeof(char));
54.        for (j = head; j < comma_pos; ++j)
55.            G.vexs[i][j - head] = vexs_str[j];
56.        G.vexs[i][j - head] = '\0';
57.        head = comma_pos + 1; comma_pos++;
58.    }
59.    //输入弧
60.    for (i = 0; i < G.arcnum; ++i)
61.    {
62.        scanf("<%d,%d,%d>", &v1, &v2, &weight); getchar();
63.        G.arcs[v1][v2] = weight;
64.    }
65.}
66.
67.void Copy_PathLinkList(PathLinkList L1, PathLinkList &L2) {
68.    //复制路径单链表 L1 为 L2
69.    if (L1.head == NULL) { L2.head = NULL; return; }
70.    PathNode *NodePtr_L1 = L1.head, *NodePtr_L2;
```

```
71.     L2.head = (PathNode *)malloc(sizeof(PathNode)); NodePtr_L2 = L2.head;
72.     while (1)
73.     {
74.         NodePtr_L2->vex_index = NodePtr_L1->vex_index;
75.         if (!NodePtr_L1->next)   break;
76.         NodePtr_L1 = NodePtr_L1->next;
77.         NodePtr_L2 = (NodePtr_L2->next = (PathNode *)malloc(sizeof(PathNode)));
78.     }
79.     NodePtr_L2->next = NULL;
80.     L2.tail = NodePtr_L2;   //复制尾指针
81.}
82.
83.void InserttoTail_PathLinkList(PathLinkList &L, int e) {
84.     //在路径单链表 L 的尾部插入元素 e
85.     if (L.head == NULL)
86.     {
87.         L.head = (PathNode *)malloc(sizeof(PathNode));
88.         L.head->vex_index = e; L.head->next = NULL;
89.         L.tail = L.head;
90.     }
91.     else
92.     {
93.         L.tail = (L.tail->next = (PathNode *)malloc(sizeof(PathNode)));  //后移尾指
针，申请新结点
94.         L.tail->vex_index = e; L.tail->next = NULL;
95.     }
96.}
97.
98.void ShortestPath_DIJKSTRA(MDN G, int source_vex_index, Path *&ShortestPath) {
99.     //求带权有向图的单源最短路径，Dijkstra 算法
100.     //源点编号为 source_vex_index，源点到编号 i 顶点的最短路径存放于 ShortestPath[i]
101.     int i, k, min_pos, min_dist; Path JumpBoard;
102.     //初始化 ShortestPath
103.     ShortestPath = (Path *)malloc(G.vexnum * sizeof(Path));
104.     for (k = 0; k < G.vexnum; ++k)
105.     {
106.         ShortestPath[k].source = source_vex_index; ShortestPath[k].destination = k;

107.         ShortestPath[k].path.head = NULL; ShortestPath[k].path.tail = NULL;
108.         ShortestPath[k].dist = (k == source_vex_index) ? 0 : INFINITY;
109.     }
110.     //初始化路径跳板 JumpBoard
111.     JumpBoard.path.head = (PathNode *)malloc(sizeof(PathNode));
```

```
112.    JumpBoard.path.head->vex_index = source_vex_index; JumpBoard.path.head->next = NULL; JumpBoard.path.tail = JumpBoard.path.head;   //第一遍的跳板为 v0 本身
113.    JumpBoard.source = source_vex_index; JumpBoard.destination = source_vex_index;

114.    JumpBoard.dist = 0; //这里故意设置为 0，为了第一遍能够顺利进行
115.
116.    int *Flag = (int *)malloc(G.vexnum * sizeof(int));  //Flag[i]标记 ShortestPath[i]是否已确定
117.    for (k = 0; k < G.vexnum; ++k)   Flag[k] = 0;    //初始化 Flag
118.    Flag[source_vex_index] = 1; //源点到源点的最短路径设为已确定
119.    for (i = 0; i < G.vexnum - 1; ++i)   //共进行 n - 1 遍
120.    {
121.        //更新最短路径
122.        min_dist = INFINITY;
123.        min_pos = source_vex_index; //为防止后面用到没有初始化的 min_pos
124.        for (k = 0; k < G.vexnum; ++k)
125.        {
126.            if (Flag[k])    continue;   //跳过已确定最短路径的顶点
127.            if (JumpBoard.dist + G.arcs[JumpBoard.destination][k] < ShortestPath[k].dist)    //若借助当前跳板跳到顶点 k 的路径长度更短，更新 ShortestPath[k]为该路径
128.            {
129.                Copy_PathLinkList(JumpBoard.path, ShortestPath[k].path);
130.                InserttoTail_PathLinkList(ShortestPath[k].path, k);
131.                ShortestPath[k].dist = JumpBoard.dist + G.arcs[JumpBoard.destination][k];
132.            }
133.            if (ShortestPath[k].dist < min_dist) { min_pos = k; min_dist = ShortestPath[k].dist; }   //在更新过程中寻找适合做跳板的路径最短者
134.        }
135.        Flag[min_pos] = 1;  //该路径最短者已确定为源点到相应顶点的最短路径
136.        //更新跳板为该路径最短者
137.        Copy_PathLinkList(ShortestPath[min_pos].path, JumpBoard.path);
138.        JumpBoard.dist = ShortestPath[min_pos].dist;
139.        JumpBoard.destination = min_pos;
140.    }
141.}
142.
143.void PrintPath(MDN G, Path path) {
144.    //格式化输出路径
145.    PathNode *p = path.path.head;
146.    if (!p) return;
147.    while (p->next)
148.    {
149.        printf("%s-", G.vexs[p->vex_index]);
```

```
        p = p->next;
    }
    printf("%s\n", G.vexs[p->vex_index]);
}

int main() {
    MDN G; Path *ShortestPath = NULL;
    CreateMDN(G);
    ShortestPath_DIJKSTRA(G, 0, ShortestPath);
    PrintPath(G, ShortestPath[G.vexnum - 1]);

    system("pause");
    return 0;
}
```

快速排序

问题描述

要求根据给定输入，按照课堂给定的快速排序算法进行排序，输出排序结果和 median3 的返回值。

midian3 是指从头尾和中间取 3 个元素，将头部元素和 3 个元素中大小的中间值交换，以避免选出最大元素或者最小元素的情况出现。

注：1、cutoff 值为 5，元素个数不足 cutoff 使用插入排序。

2、输入、输出格式参见测试用例 0。

参考答案

```
#include<stdio.h>
#include<stdlib.h>

#define MAX_NUM 1000
#define CUTOFF 5

void StraightInsertSort(int *L, int low, int high) {
    //直接插入排序
    //待排序序列数组 L 下标从 low 到 high 的位置（包括 L[low] 和 L[high] ）
    int start = low, i, inserted_elem;
    for (low++; low <= high; ++low)
```

```
    {
        inserted_elem = L[low];
        for (i = low - 1; L[i] > inserted_elem && i >= start; --i)
            L[i + 1] = L[i];
        L[i + 1] = inserted_elem;
    }
}

int Partition(int *L, int low, int high) {
    //分割序列：调整数组使序列被枢轴分成两部分，枢轴左边都比枢轴小，右边都比枢轴大
    //待分割序列为数组 L 下标从 low 到 high 的位置（包括 L[low] 和 L[high] ）
    //返回枢轴的下标
    int pivot_value = L[low], temp; //枢轴可以是任意的，这里统一设为第一个元素
    while (low < high)   //结束条件：low 与 high 相遇
    {
        //从左向右和从右向左交替搜索，不断与枢轴互换位置
        while (low < high && L[high] >= pivot_value)  high--; //high 向左搜索，直到找到比枢轴小的元素或与 low 相遇
        L[low] = L[high];   //假交换技巧：在每次交换时不将枢轴赋值到相应位置，结束后才将枢轴记录到位
        while (low < high && L[low] <= pivot_value)   low++;  //low 向右搜索，直到找到比枢轴大的元素或与 high 相遇
        L[high] = L[low];
    }
    L[low] = pivot_value;   //最后将枢轴赋值到相应的位置
    return low;
}

void QuickSorting(int *L, int low, int high) {
    //快速排序，递归算法
    //待排序序列为数组 L 下标从 low 到 high 的位置（包括 L[low] 和 L[high] ）
    if (low >= high) return; //递归终止条件，待排序序列长度小于或等于 1
    int pivot_index = Partition(L, low, high);  //用枢轴分割序列
    QuickSorting(L, low, pivot_index - 1);  //递归地排序枢轴左边的序列
    QuickSorting(L, pivot_index + 1, high); //递归地排序枢轴右边的序列
}

void QuickSort(int *L, int N, int *median3) {
    //启动快速排序
    //待排序序列为数组 L 的前 N 个元素

    //预处理：从头尾和中间取 3 个元素，将头部元素和 3 个元素中大小的中间值交换，以避免选出最大元素或者最小元素的情况出现，以提高快排的效率。并将换序后的三个数依次存入数组 median3 中，以便打印
```

```
51.     median3[0] = L[0], median3[1] = L[N / 2 - 1], median3[2] = L[N - 1];
52.     int min = 0, max = 0, mid, temp;
53.     for (int i = 1; i < 3; ++i)  //找到最大、最小元素
54.     {
55.         if (median3[i] > median3[max])   max = i;
56.         if (median3[i] < median3[min])   min = i;
57.     }
58.     mid = 0 + 1 + 2 - max - min;    //推算出中间值元素
59.     if (mid != 0) { temp = median3[0]; median3[0] = median3[mid]; median3[mid] = tem
p; }    //中间值元素不是头部元素，则与头部元素交换
60.     L[0] = median3[0]; L[N / 2 - 1] = median3[1]; L[N - 1] = median3[2];    //更新待
排序序列的相应元素
61.
62.     QuickSorting(L, 0, N - 1);  //开始快速排序
63.}
64.
65.void PrintArray(int *L, int low, int high) {
66.     //格式化输出序列
67.     //待打印序列为数组 L 下标从 low 到 high 的位置（包括 L[low] 和 L[high] ）
68.     for (int i = low; i <= high; ++i)
69.         printf("%d ", L[i]);
70.}
71.
72.int main() {
73.     int L[MAX_NUM], N = 0, median3[3];
74.     while (scanf("%d", L + N))  N++;
75.     if (N <= CUTOFF) StraightInsertSort(L, 0, N - 1);    //元素个数不足 CUTOFF 使用插入
排序
76.     else QuickSort(L, N, median3);
77.
78.     printf("After Sorting:\n");
79.     PrintArray(L, 0, N - 1);
80.     printf("\nMedian3 Value:\n");
81.     if (N <= CUTOFF) printf("none\n");
82.     else printf("%d %d %d \n", median3[0], median3[1], median3[2]);
83.
84.     system("pause");
85.     return 0;
86.}
```

堆排序

问题描述

实验要求：用堆排序算法按关键字递减的顺序排序。
程序输入：待排序记录数（整数）和待排序记录（整数序列）；
程序输出：建堆结果和建堆后第一、第二次筛选结果。（注：待排序记录数大于等于 3）

参考答案

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. #define TIMES 2
5.
6. void HeapAdjust(int *record, int start, int end) {
7.     //堆排序的筛选过程（大根堆）。筛选对象为 record 中下标为 start 的元素，end 记录序列末尾的位置
8.     int max_child_index;
9.     record[0] = record[start];  //暂存待筛选元素
10.    while (start <= end / 2) //一直调整到叶子结点
11.    {
12.        //找到左右孩子中较大者的下标
13.        max_child_index = 2 * start;
14.        if (2 * start + 1 <= end)
15.            if (record[2 * start + 1] > record[2 * start])
16.                max_child_index++;
17.        //若左右孩子有比待筛选元素大的，与较大者交换，否则停止调整
18.        if (record[0] < record[max_child_index])
19.        {
20.            record[start] = record[max_child_index];    //假交换
21.            start = max_child_index;
22.        }
23.        else break;
24.    }
25.    record[start] = record[0];  //待筛选元素记录到位
26.}
27.
28.void PrintRecord(int *record, int start, int end) {
29.    //格式化输出记录，数组 record 下标从 start 到 end 的元素
30.    for (int i = start; i <= end; ++i)
31.        printf("%d ", record[i]);
```

```
    printf("\n");
}

int main() {
    int *record, N, i, k;
    scanf("%d", &N);    //待排序记录数
    record = (int *)malloc((N + 1) * sizeof(int));  //存储待排序记录
    for (i = 1; i <= N; ++i) //为了方便，从下标 1 开始存储待排序记录。下标为 0 的位置当做临时变量用
        scanf("%d", record + i);
    //初始建堆
    for (int start = N / 2; start >= 1; --start) //从第一个非叶子结点开始，从后向前筛选
        HeapAdjust(record, start, N);
    PrintRecord(record, 1, N);

    //TIMES 遍筛选
    int end = N;
    for (k = 0; k < TIMES; ++k)
    {
        record[1] = record[end--];  //堆顶元素用最后一个元素替代
        HeapAdjust(record, 1, end);
        PrintRecord(record, 1, end);
    }

    system("pause");
    return 0;
}
```