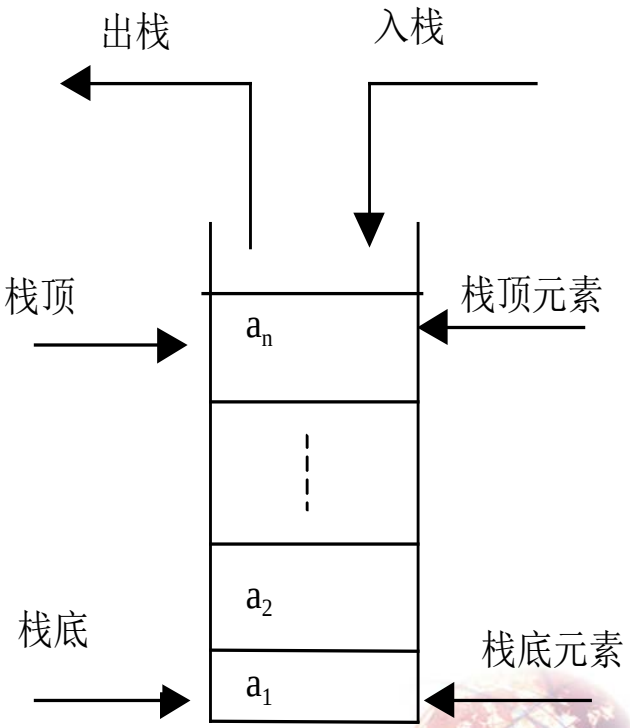
第3章 栈和队列

- 栈和队列是运算受限的线性表。
- 栈
  - 存储结构（顺序栈和链栈 ）
  - 栈的应用举例（递归调用、地图染色问题）
- 队列
  - 存储结构（顺序队列和链队列 ）
  - 队列应用举例（划分子集问题）

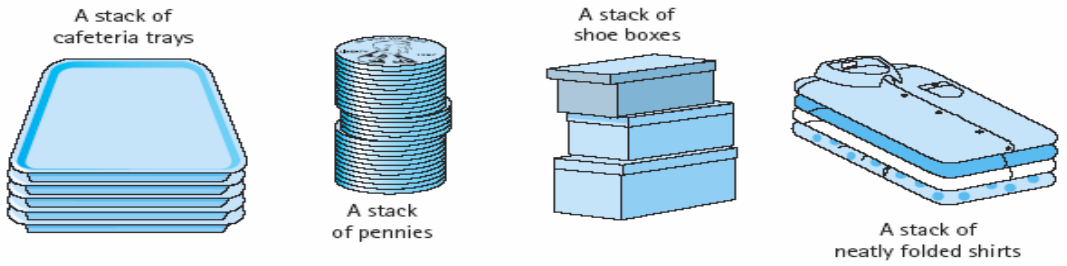
栈的基本概念

- 定义：
  - 栈是限定仅在表尾进行插入和删除运算的线性表；
  - 表尾称为栈顶；
  - 表头称为栈底；
  - 当栈中没有数据元素时称为空栈。
- 例： $S=(a_1, \ldots, a_n)$
  - 可以形象描述为右图所示形式：
  - a_1是栈底元素；
  - a_n是栈顶元素；
  - 入栈指插入数据元素；
  - 出栈指删除数据元素；



栈的基本概念

- 对于栈来说，最后进栈的数据元素，最先出栈，故把栈称为后进先出(LIFO-Last In First Out)的数据结构，或先进后出(FILO-First In Last Out)的数据结构。
- 栈的用途非常广泛，例如，汇编处理程序中的句法识别、表达式计算以及回溯问题就是基于栈实现的。栈还经常使用在函数调用时的参数传递和函数值的返回方面。



栈的常用运算

- 栈的常用运算有以下五种：
  - 置空栈—SetNull(S)，完成对栈的初始化。
  - 判断栈空—Empty(S)，若栈S为空则返回真，否则返回假。
  - 进栈——Push(S，e)，在栈S的栈顶插入数据元素e。
  - 出栈—Pop(S)，删除栈S的栈顶数据元素，并将数据元素返回。
  - 取栈顶元素—GetTop(S)，取栈S的栈顶数据元素，并把数据元素返回。该操作完成后，栈的状态不变。

栈的存储结构

- 栈的存储结构有两种：
  - 顺序存储结构
    采用顺序表存储的栈称为顺序栈。
  - 链式存储结构
    采用单链表存储的栈称为链栈。

顺序栈的定义

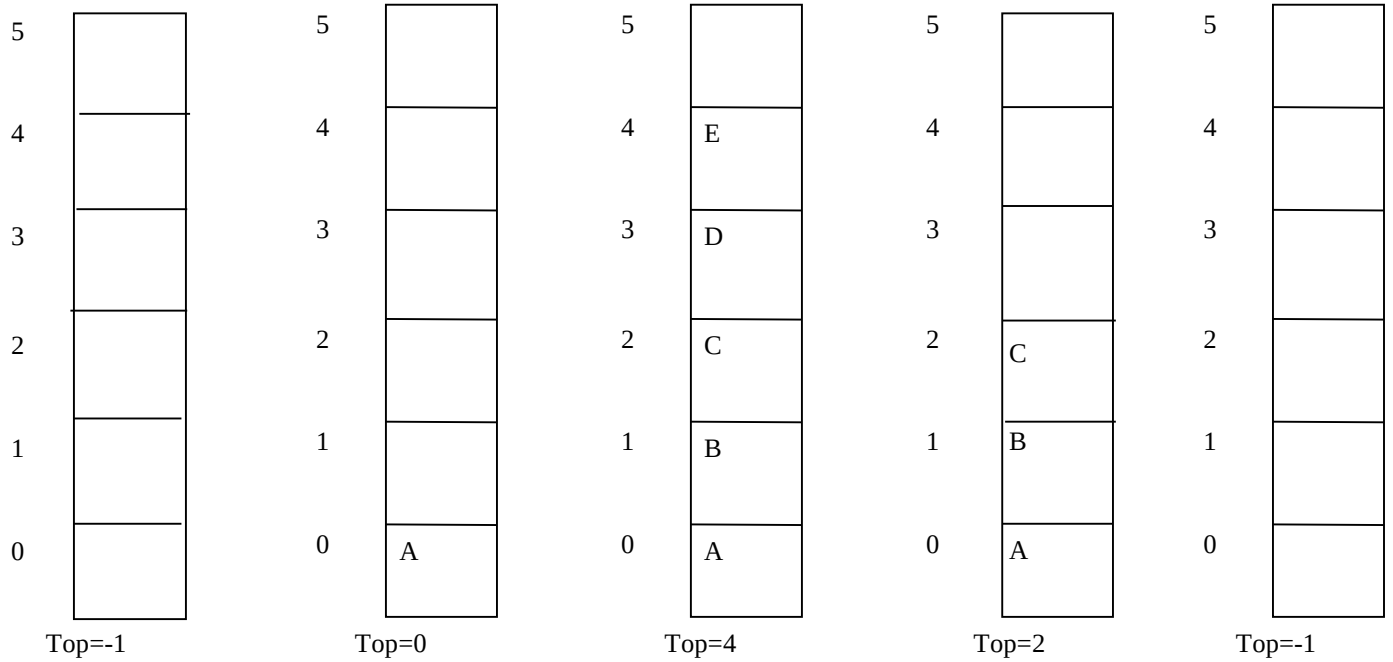
- 定义
  - 栈的顺序存储结构定义为：

```
struct Stack
{ datatype elements[maxsize];
  int Top;
 };
```

  - 其中：

    maxsize是栈的容量。
    datatype是栈中数据元素的数据类型。
    Top指示当前栈顶位置。

栈操作说明



(a)空栈　(b)A进栈　(c)BCDE进栈　(d)ED出栈　(F)CBA出栈

栈运算实现-1

- 置空栈：

```
struct Stack *SetNuLLS(struct Stack *S)
{     S→Top=-1；return  S;
} /* SetNuLLS */
```


栈运算实现-1

- 置空栈：

```
struct Stack *SetNuLLS(struct Stack *S)
{     S→Top=-1；return  S;
} /* SetNuLLS */
```

- 判断栈是否为空：

```
int EmptyS(struct Stack *S)
{     if (S→Top>=0) return 0；
       else return 1；
}   /* EmptyS */
```

栈运算实现-1

- 置空栈：

```
struct Stack *SetNuLLS(struct Stack *S)
{     S→Top=-1；  return  S;
}  /* SetNuLLS */
```

- 判断栈是否为空：

```
int EmptyS(struct Stack *S)
{     if (S→Top>=0) return 0；
       else return 1；
}   /* EmptyS */
```

- 进栈：

```
struct Stack *PushS(struct Stack *S, datatype e)
{   if (S→Top>=maxsize-1){ printf(“Stack Overflow”);
                    /* 上溢现象*/
            return NULL；
     } else { S→Top++；    S→elements[S→Top]=e；  }
      return S；
}   /* PushS*/
```

栈运算实现-2

- 出栈：

```
datatype *POPS(struct Stack *S)
{    datatype *ret;
     if (EmptyS(S)) {printf("Stack Underflow");
                     return NULL;
     } else  {S→Top--;
              ret=(datatype *)malloc(sizeof(datatype));
             *ret= S→elements[S→Top+1];
              return ret;    }
}  /* PopS */
```

栈运算实现-2

- 出栈：

```
datatype *POPS(struct Stack *S)
{    datatype *ret;
     if (EmptyS(S)) {printf("Stack Underflow");
                          return NULL;
     } else  {S→Top--;
               ret=(datatype *)malloc(sizeof(datatype));
              *ret= S→elements[S→Top+1];
               return ret;    }
}  /* PopS */
```

- 取栈顶元素：

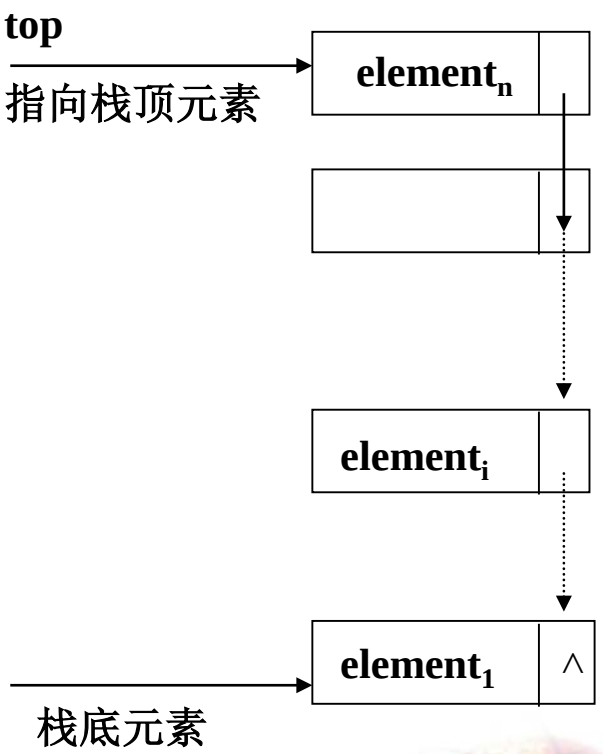
```
datatype *GetTopS(struct Stack *S)
{    datatype *temp;
     if (EmptyS(S)) {printf("Stack is empty");
                          return NULL;
     } else { temp=(datatype *)malloc(sizeof(datatype));
              *temp= S→elements[S→Top];    return temp;}
}    /* GetTopS */
```

链栈的定义

- 定义
  - 栈的链式存储结构称为链栈。它是运算受限的单链表，其插入和删除操作仅在表头进行。
  - 链栈定义如下：

```
struct Node
{ datatype element;
  struct Node *next;
};
struck Node *top;
```

- 链栈示意图如右图
  - 栈顶是top指针，它惟一地确定一个链栈。
  - 当top等于NULL时，该链栈为空栈。



链栈的定义

- Q：能否将链栈中的指针方向反过来，从栈底到栈顶？

链栈的定义

- Q：能否将链栈中的指针方向反过来，从栈底到栈顶？
- A：不行，如果反过来的话，删除栈顶元素时，为修改其前驱指针，需要从栈底一直找到栈顶。

链栈运算

- 进栈：

```
strct Node *PushL(struct Node *S, datatype e)
{       struct Node *p;
        p=(struct Node*)malloc(sizeof(struct Node));
        p→element=e;   p→next=S;   S=p; return S;
}  /* PushL */
```

链栈运算

- 进栈：

```
strct Node *PushL(struct Node *S, datatype e)
{       struct Node *p;
        p=(struct Node*)malloc(sizeof(struct Node));
        p→element=e;   p→next=S;   S=p; return S;
}  /* PushL */
```

- 出栈：

```
datatype *PopL(struct Node *S)
{       datatype *X; struct Node *temp;
        if(S==NULL){ printf("Stack is  underflow");
                              return  NULL;
        } else { X=(datatype *)malloc(sizeof(datatype));
                    *X=S→element;     temp=S;   S =S →next;
                     free(temp);    return X;
        }
}   /* PopL */
```

栈的应用

- 栈的典型应用有：
  - 过程递归调用；
  - “回溯”问题的求解。

递归调用

- 以计算Fibonacci序列为例
  - Fibonacci序列定义为：

$$Fib(n)=\begin{cases}0 & (n=0)\\ 1 & (n=1)\\ Fib(n-1)+Fib(n-2) & (n>1)\end{cases}$$

递归调用

- 以计算Fibonacci序列为例
  - Fibonacci序列定义为：

$$Fib(n)=\begin{cases}0 & (n=0)\\ 1 & (n=1)\\ Fib(n-1)+Fib(n-2) & (n>1)\end{cases}$$

  - 算法描述

```
int Fib (int n)
{ int fib;
        if(n==0)  fib=0;
        else      if (n==1)  fib=1;
                  else       fib=Fib(n-1)+Fib(n-2);
        return (fib);
}  /* Fib */
```

递归调用

```
int Fib (int n)
{ int fib;
        if(n==0)  fib=0;
        else      if (n==1)  fib=1;
                  else       fib=Fib(n-1)+Fib(n-2);
        return (fib);
   }   /* Fib */
```

递归调用

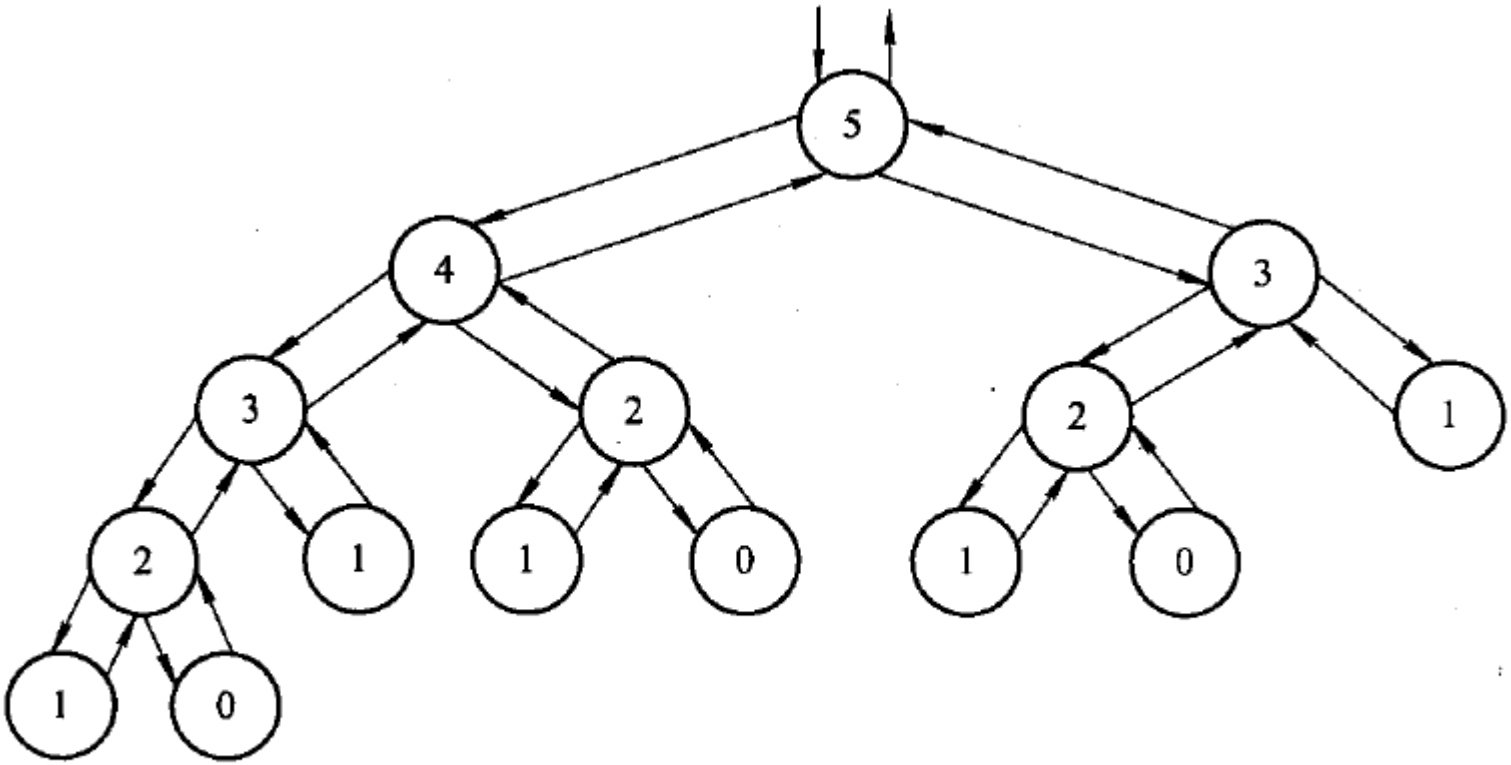


图 3-6　递归执行过程

递归调用

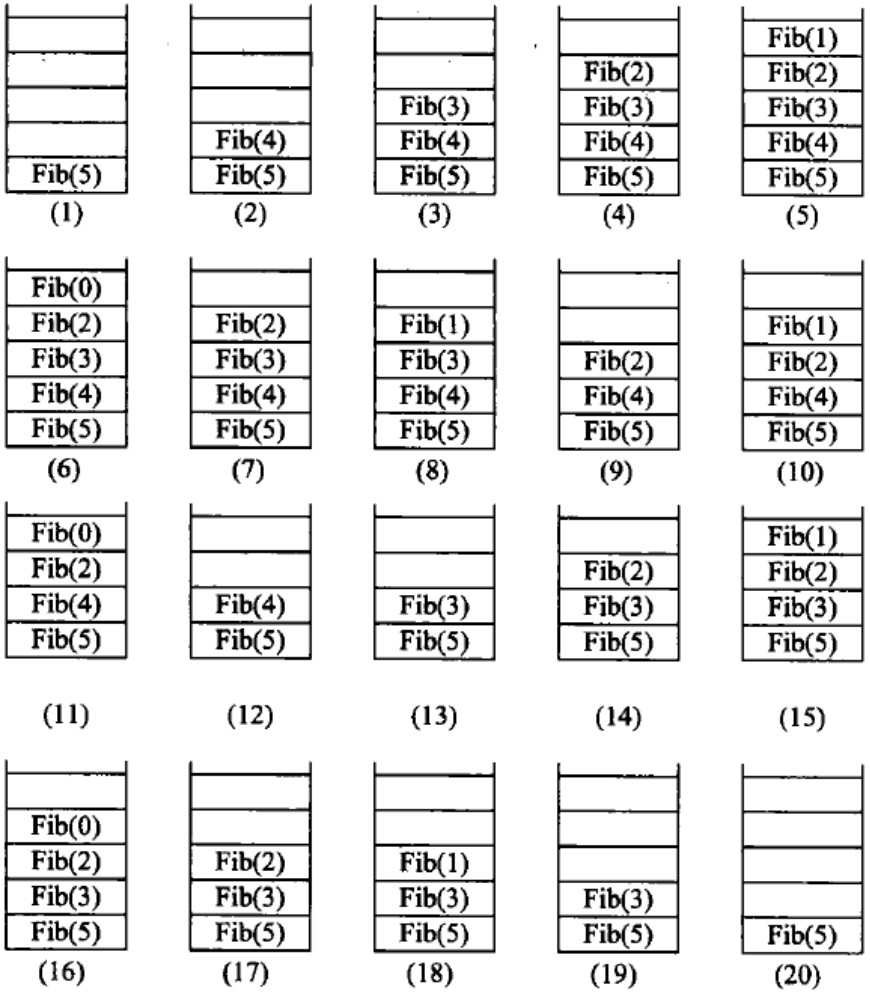
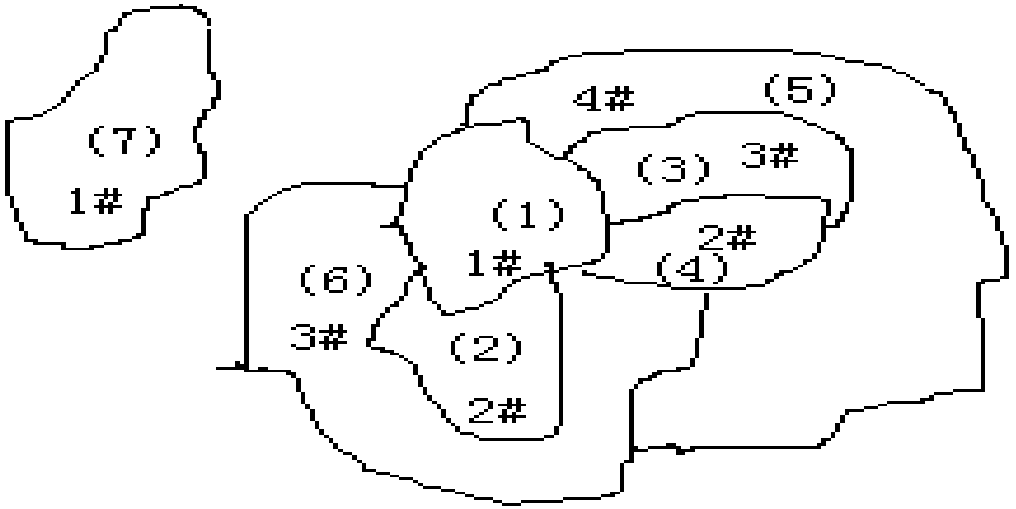


图 3-7　递归调用过程中栈的变化

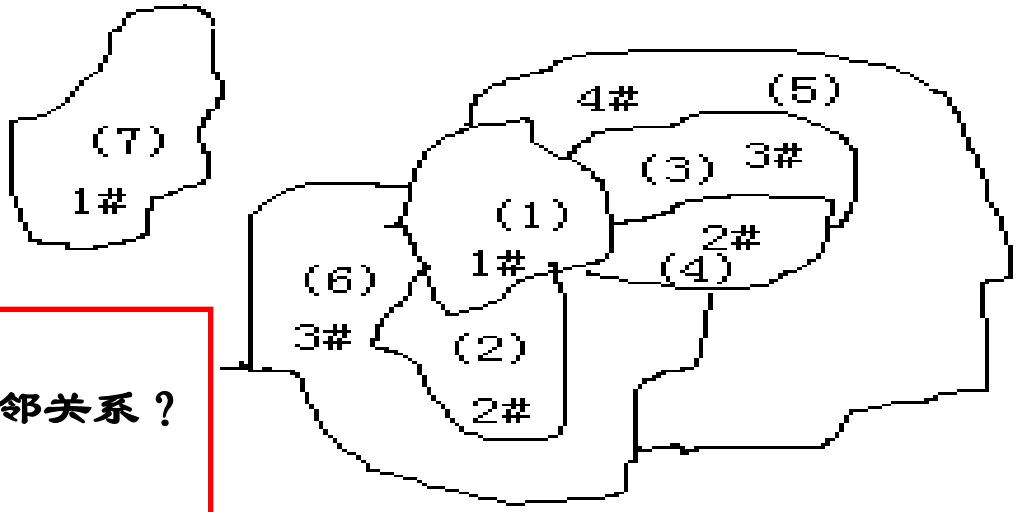
地图染色问题

- 四色定理是指可以用不多于四种颜色对地图着色，使相邻的行政区域不重色。
- 问题：假设已知地图的行政区如图所示，对每个行政区编号分别是(1)、(2)、(3)、(4)、(5)、(6)、(7)。同时用1#、2#、3#、4#表示各行政区的颜色。



地图染色问题

- 四色定理是指可以用不多于四种颜色对地图着色，使相邻的行政区域不重色。
- 问题：假设已知地图的行政区如图所示，对每个行政区编号分别是(1)、(2)、(3)、(4)、(5)、(6)、(7)。同时用1#、2#、3#、4#表示各行政区的颜色。



如何着色？
如何表示行政区域及其相邻关系？
如何表示颜色？
算法如何设计？

基本思想和主要数据结构

- 算法的基本思想
  - 每个区域逐次用颜色1#、2#、3#、4#进行试探染色；
  - 可染，则用栈记下该区域的颜色序号；
  - 若出现用1#到4#颜色均与相邻区域的颜色重色，则需退栈回溯，即修改当前栈顶的颜色序号，再进行试探。
- 数据结构
  - 行政区域间的相邻关系矩阵R[n][n]：

$$R[i][j]=\begin{cases}1 & \text{行政区域}i+1\text{和}j+1\text{间是相邻的；}\\ 0 & \text{行政区域}i+1\text{和}j+1\text{间是不相邻的；}\end{cases}$$

  - 记录行政区域的所染颜色的序号栈S：

    S[i]=k

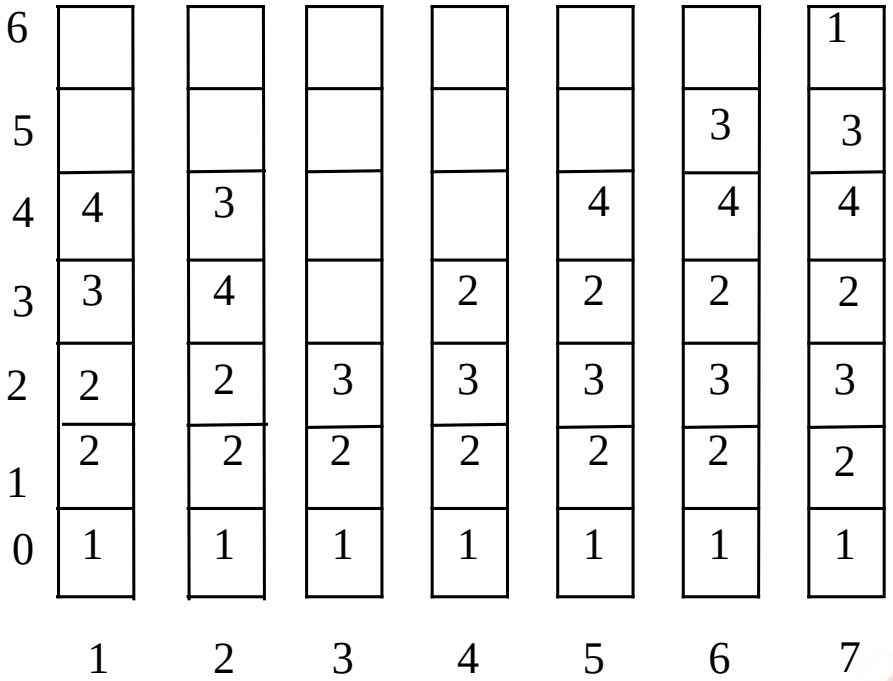
    表示行政区域i+1所染颜色的序号为k，

R矩阵和栈S的变化过程

R[7][7]

	0	1	2	3	4	5	6
0	0	1	1	1	1	1	0
1	1	0	0	0	0	1	0
2	1	0	0	1	1	0	0
3	1	0	1	0	1	1	0
4	1	0	1	1	0	1	0
5	1	1	0	1	1	0	0
6	0	0	0	0	0	0	0

S[7]



地图染色算法描述

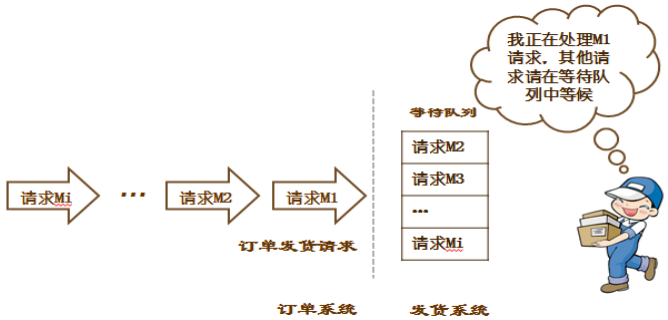
```
#define N 7
void mapcolor (int R[][], int n, int S[])
{ int color，area，k;
   S[0]=1;          /* 第一行政区域染1#号颜色 */
   area=1;          /* 从第二行政区域开始试探染色 */
   color=1;         /* 从1#颜色开始试探 */
   while (area<N)
     {  while (color<=4)
          { k=0;          /* 指示已染色区域  */
            while ((k<area)&&(S[k]*R[area][k]!=color))
                { k++;          /* 判断当前area区域与k区域是否重色*/
                }
            if (k<area)  color++;     /* area区域与k区域重色 */
            else {  S[area]=color;     /* area区域与k区域不重色*/
                    area++;
                    if (area==N) break;
                    color=1; }    /*试探下一个行政区域  */
          }    /*  while (color<=4) end  */
        if (color>4) {                /* area区域找不到合适的颜色 */
              area-=1; color=s[area]+1; } /* 回溯修改area-1区域颜色  */
      }
} /* mapcolor */
```

队列的基本概念

- 定义：队列也是一种运算受限的线性表。
  - 它只允许在表的一端进行插入，该端称为队尾(Rear)
  - 它只允许在表的另一端进行删除，该端称为队头(Front)。
  - 队列亦称作先进先出(First In First Out)的线性表。
  - 当队列中没有元素时称为空队列。
- 队列的示意图

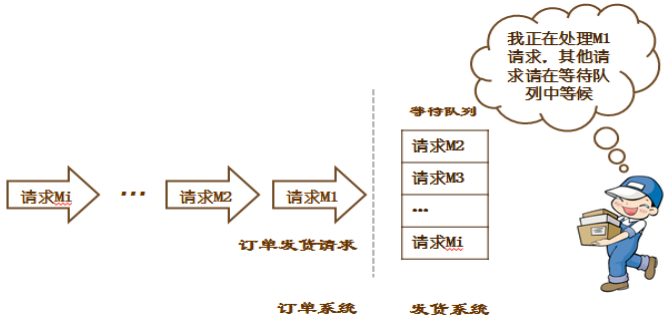
队列的基本概念

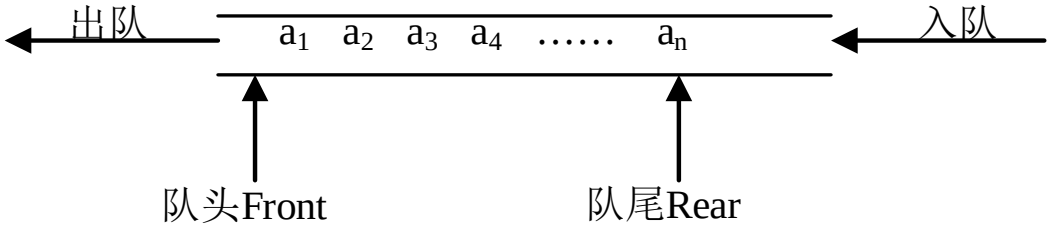
- 定义：队列也是一种运算受限的线性表。
  - 它只允许在表的一端进行插入，该端称为队尾(Rear)
  - 它只允许在表的另一端进行删除，该端称为队头(Front)。
  - 队列亦称作先进先出(First In First Out)的线性表。
  - 当队列中没有元素时称为空队列。
- 队列的示意图



队列的基本概念

- 定义：队列也是一种运算受限的线性表。
  - 它只允许在表的一端进行插入，该端称为队尾(Rear)
  - 它只允许在表的另一端进行删除，该端称为队头(Front)。
  - 队列亦称作先进先出(First In First Out)的线性表。
  - 当队列中没有元素时称为空队列。
- 队列的示意图





队列的基本运算

- 队列的基本运算有以下五种：
  - **SetNull(Q):** 置Q为一个空队列。
  - **Empty(Q):** 判队列Q是否为空队列。当Q是空队列时，返回“真”值，否则返回“假”值。
  - **Front(Q):** 取队列Q的队头元素，队列中元素保持不变。
  - **Enqueue(Q,X):** 将元素X插入队列Q的队尾。简称为入队(列)。
  - **Dequeue(Q):** 删除队列Q的队头元素，简称为出队(列)。函数返回原队头元素。

队列的存储结构

- 队列的存储结构有两种：
  - 顺序存储结构
    以顺序存储结构存储的队列称为顺序队列。
  - 链式存储结构
    以链式存储结构（单链表）存储的队列称为链队列。

顺序队列

- 定义：队列的顺序存储结构称为顺序队列。
- 顺序队列的形式说明如下：
  - **struct sequeue**
    {datatype data[maxsize];
    int front，rear;
    }; /* 顺序队列的类型 */
    struct sequeue *sq /* sq 是顺序队列的指针 */

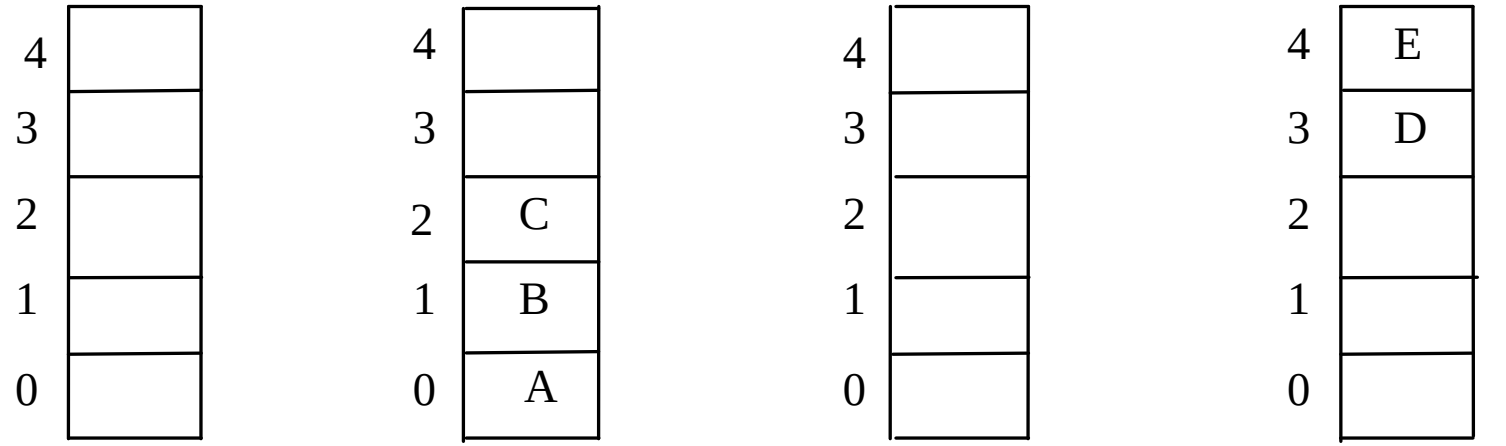
顺序队列

- 定义：队列的顺序存储结构称为顺序队列。
- 顺序队列的形式说明如下：
  - struct sequeue
    {datatype data[maxsize];
     int front，rear;
    }; /* 顺序队列的类型 */
    struct sequeue *sq /* sq 是顺序队列的指针 */
- 注：
  - 规定头指针front总是指向当前队头元素的前一个位置，尾指针rear指向当前队尾元素的位置。
  - 初始时，队列的头、尾指针指向向量空间下界的前一个位置，在此设置为-1。
  - 若不考虑溢出，
    入队运算：sq->rear++; sq->data[sq->rear]=x;
    出队运算：sq->front++; *temp = sq->data[sq->front];

顺序队列概念的实例说明

- 下图说明了在顺序队列中进行出队和入队运算时队列中的数据元素及其头、尾指针的变化情况。

(sq→rear)-(sq→front)==maxsize



sq→rear=-1
sq→front=-1

sq→rear=2
sq→front=-1

sq→rear=2
sq→front=2

sq→rear=4
sq→front=2

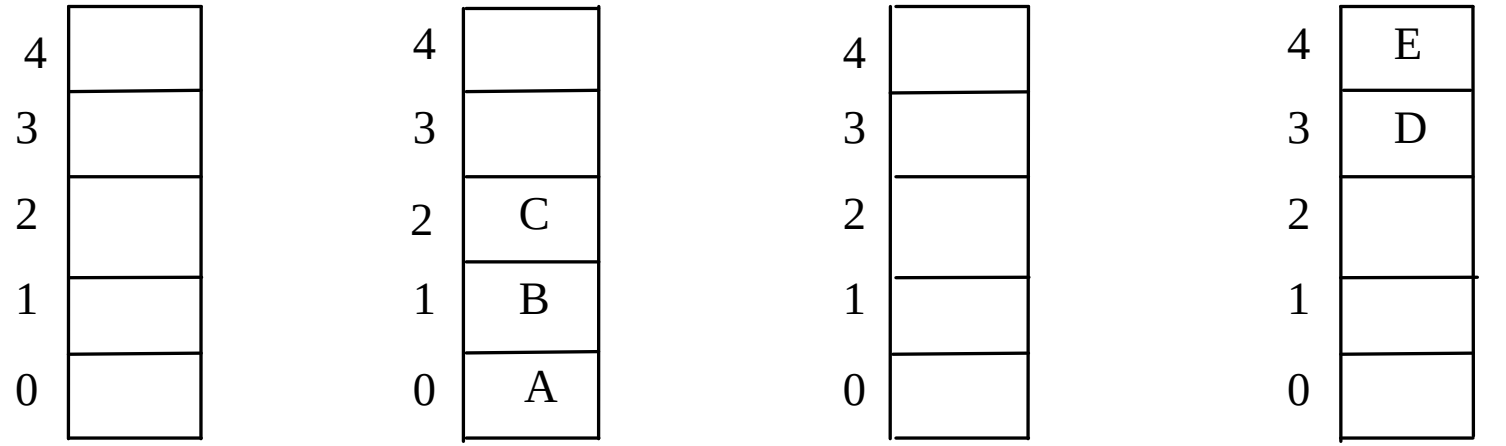
(a)空队列　(b)ABC 相继入队　(c)ABC 相继出队　(d)DE 相继入队

顺序队列运算时的头、尾指针变化情况

顺序队列概念的实例说明

- 下图说明了在顺序队列中进行出队和入队运算时队列中的数据元素及其头、尾指针的变化情况。

(sq→rear)-(sq→front)==maxsize



(a)空队列　(b)ABC 相继入队　(c)ABC 相继出队　(d)DE 相继入队

顺序队列运算时的头、尾指针变化情况

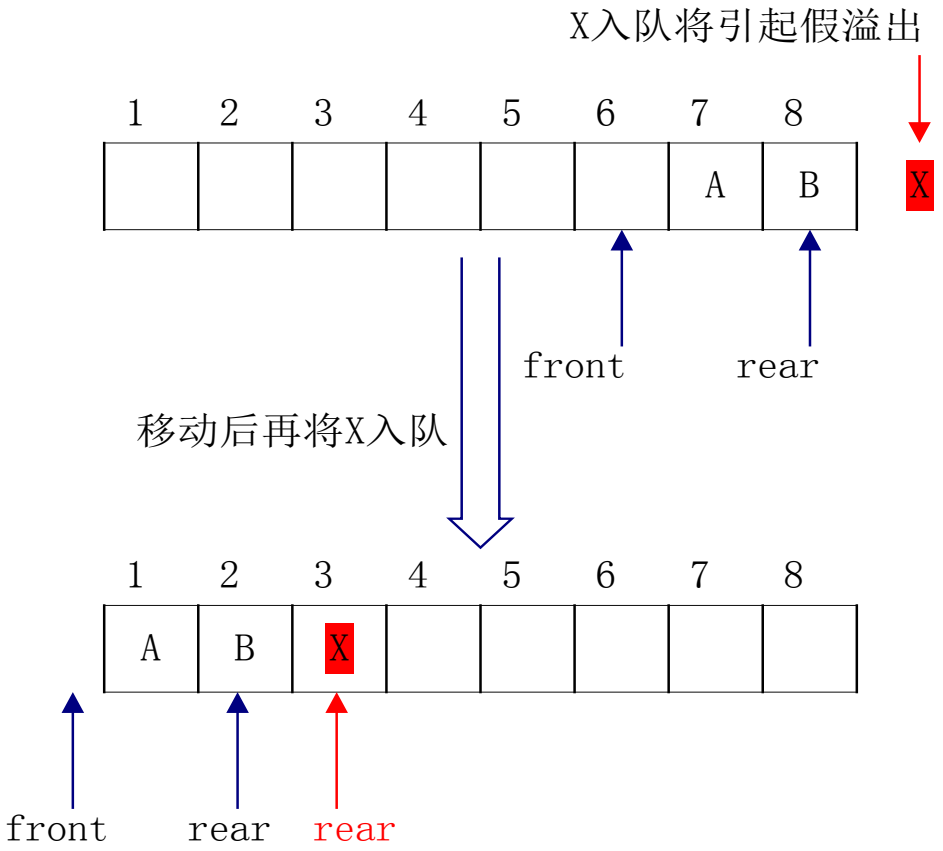
存在问题：假溢出

存在问题及解决办法

- 存在问题
  - 假上溢：顺序队列中存在未用的存储单元，但不能继续进行入队操作。
- 解决办法：
  - 移动“靠近队尾”的数据元素至“靠近对头”的位置。
  - 将顺序队列构造为环状，形成循环队列。
  - 链队列。

移动数据元素

- 解决办法：
  - 将数据元素移至队头，修改队头、队尾指针。

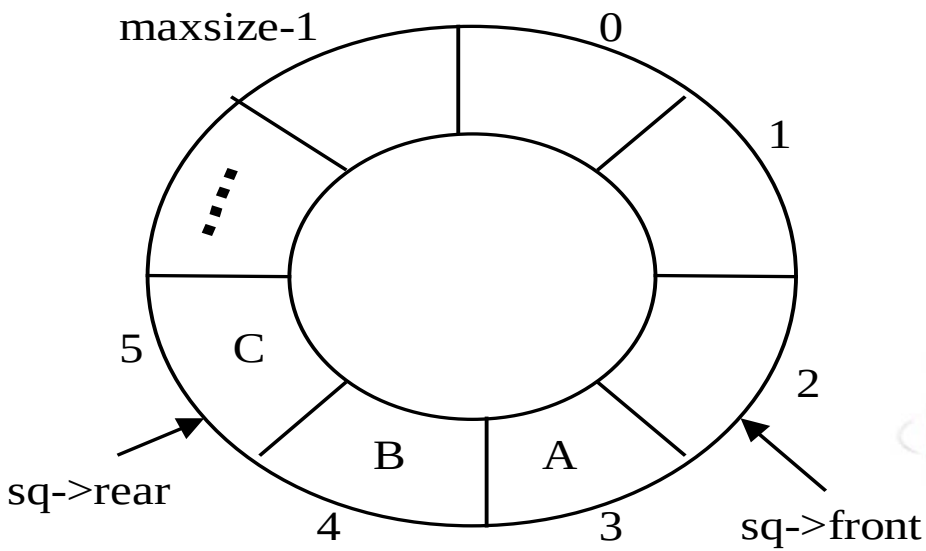


循环队列

- 解决办法：
  - 设想向量sq→data[maxsize]是一个首尾相接的圆环，即sq→data[0]接在sq→data[maxsize-1]之后，即循环队列。如果利用“模”运算，上述循环意义下的尾/头指针加/减1操作可以更简洁地描述为：

    sq→rear=(sq→rear±1)%maxsize

循环队列示意图

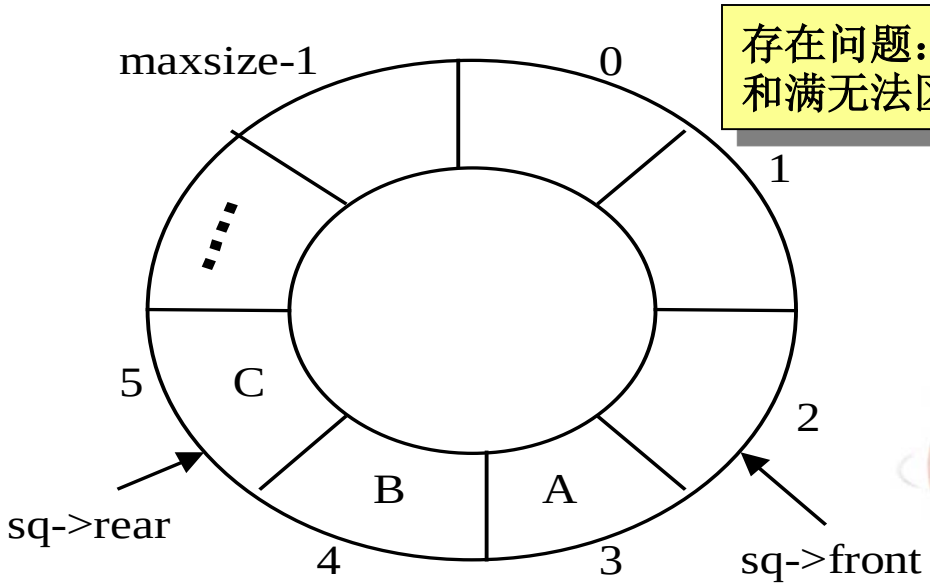


循环队列

- 解决办法：
  - 设想向量sq→data[maxsize]是一个首尾相接的圆环，即sq→data[0]接在sq→data[maxsize-1]之后，即循环队列。如果利用“模”运算，上述循环意义下的尾/头指针加/减1操作可以更简洁地描述为：

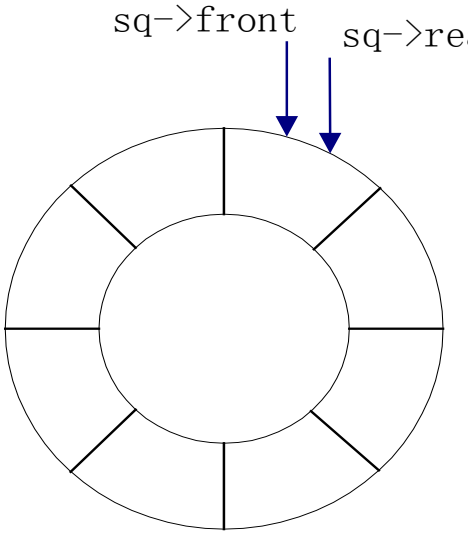
    sq→rear=(sq→rear±1)%maxsize

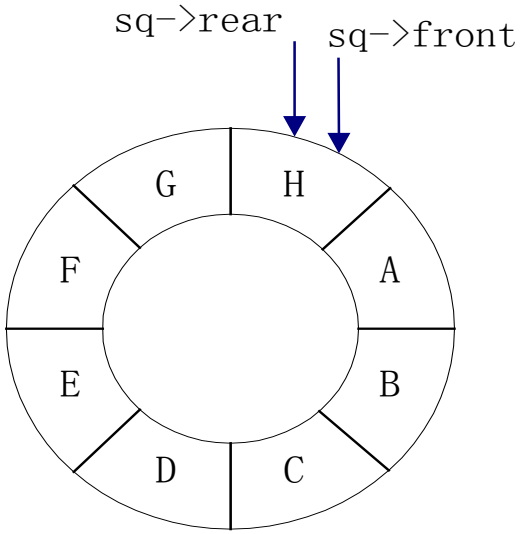
循环队列示意图



循环队列空和满无法区分示例

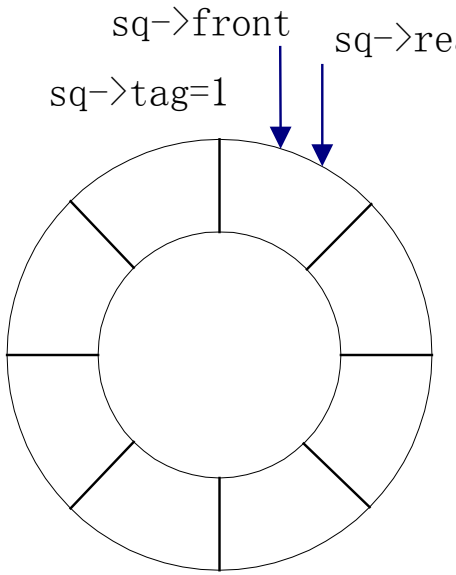
- 循环队列空情况如左图
- 循环队列满情况如右图
- 解决办法：
  - 设置空/满标志
  - 牺牲一个存储单元

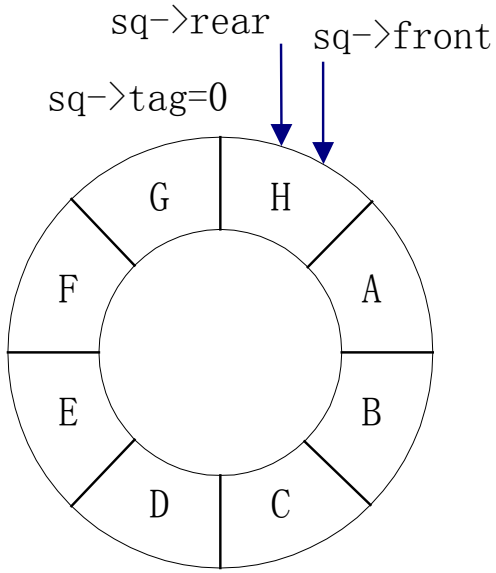




设置空/满标志

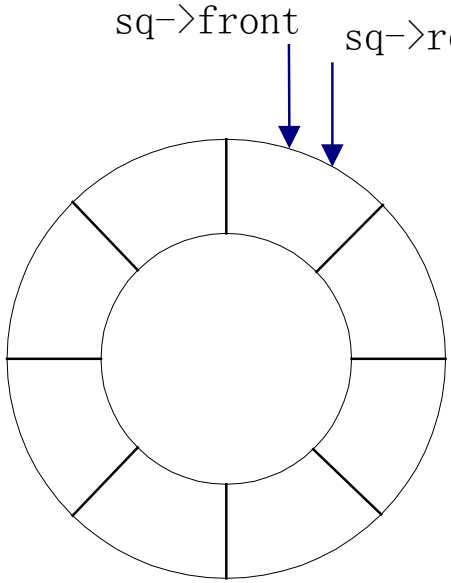
- 具有空/满标志的循环队列的数据结构
- **struct sequeue**
  {datatype data[maxsize];
  int front，rear，tag; /*tag=1表示空；tag=0表示满*/
  }; /* 循环队列的类型 */

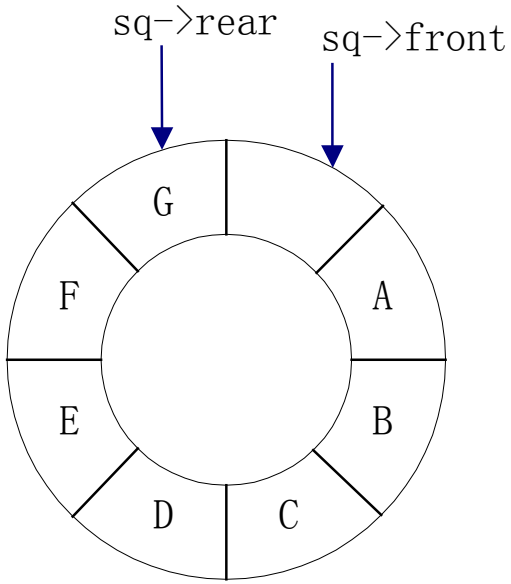




牺牲一个存储单元的循环队列

- 牺牲一个存储单元的循环队列的数据结构
- **struct sequeue**
  {datatype data[maxsize];
  int front，rear;
  }; **/* 循环队列的类型 */**





以牺牲一个存储单元的循环队列为例说明循环队列的基本运算

空：sq->front==sq->rear　　满：sq->front==(sq->rear+1)%maxsize

循环队列的基本运算-1

- 置空队

```
struct sequeue *SetNullQ(sequeue *sq)   / * 置队列sq为空队 */
{sq→front=maxsize-1;
  sq→rear=maxsize-1;
  return sq; }   /* SetNullQ */
```

循环队列的基本运算-1

- 置空队

```
struct sequeue *SetNullQ(sequeue *sq)   / * 置队列sq为空队 */
{sq→front=maxsize-1;
  sq→rear=maxsize-1;
  return sq; }   /* SetNullQ */
```

- 判队空

```
int EmptyQ(sequeue * sq)      /* 判别队列sq 是否为空 */
{if(sq→rear= =sq→front)  return 1;
   else  return 0 ;}  /* EmptyQ */
```

循环队列的基本运算-1

- 置空队

```
struct sequeue *SetNullQ(sequeue *sq)   / * 置队列sq为空队 */
{sq→front=maxsize-1;
 sq→rear=maxsize-1;
 return sq; }   /* SetNullQ */
```

- 判队空

```
int EmptyQ(sequeue * sq)      /* 判别队列sq 是否为空 */
{if(sq→rear= =sq→front)  return 1;
   else  return 0 ;}  /* EmptyQ */
```

- 取队头元素

```
datatype  *FrontQ(sequeue * sq)        /* 取 队列sq的队头元素 */
{datatype *temp;
 if (EmptyQ(sq) {printf("queue is empty");  return  NULL;}
 else {  temp=(datatype *)malloc(sizeof(datatype));
         *temp= sq→data[(sq→front+1)％ maxsize];  return temp;}
}  /* FrontQ */
```

循环队列的基本运算-2

- 入队

```
int EnqueueQ(sequeue * sq, datatype x)
   /* 将新元素x插入队列* sq的队尾 */
{ if (sq→front==(sq→rear+1)％ maxsize)
                { printf("queue is full"); return 0; }   /* 队满上溢 */
     else {     sq→rear=(sq→rear+1)％maxsize;
                sq→data[sq→rear]=x;     return 1;}
}    /* EnqueueQ */
```

循环队列的基本运算-2

- 入队

```
int EnqueueQ(sequeue * sq, datatype x)
  /* 将新元素x插入队列* sq的队尾 */
{ if (sq→front==(sq→rear+1)％ maxsize)
            { printf("queue is full"); return 0; }   /* 队满上溢 */
    else {    sq→rear=(sq→rear+1)％maxsize;
              sq→data[sq→rear]=x;    return 1;}
}   /* EnqueueQ */
```

- 出队

```
datatype *DequeueQ(sequeue * sq)
   /* 删除队列 *sq的头元素，并返回该元素 */
{ datatype *temp;
  if (EmptyQ(sq))
       { printf("queue is enpty");  return  NULL;}   /* 队空下溢 */
  else  { temp=(datatype *)malloc(sizeof(datatype));
          sq→front=(sq→front+1) ％ maxsize;
          *temp = sq→data[sq→front];    return temp;}
}
```

循环队列应用举例

编写一个打印二项式系数表（即杨辉三角）的算法。

```
      1    1
    1   2   1
  1   3   3   1
1   4   6   4   1
.....................
```

系数表中的第 k行有 k+1个数，除了第一个和最后一个数为1之外，其余的数则为上一行中位其左、右的两数之和。

循环队列应用举例

编写一个打印二项式系数表（即杨辉三角）的算法。

```
      1    1
    1   2   1
  1   3   3   1
1   4   6   4   1
.....................
```

系数表中的第 k行有 k+1个数，除了第一个和最后一个数为1之外，其余的数则为上一行中位其左、右的两数之和。

如果要求计算并输出杨辉三角前 n 行的值，则队列的最大空间应为 n+2。假设队列中已存有第 k 行的计算结果，并为了计算方便，在两行之间添加一个“0”作为行界值，则在计算第 k+1 行之前，头指针正指向第 k 行的“0”的前一个位置，而尾元素为第 k+1 行的"0"。

杨辉三角基本算法

- 由此从左到右依次输出第 k 行的值，并将计算所得的第 k+1 行的值插入队列的基本操作为：

```
for(i=0; i<n-1; i++)
 {do {
      s = DequeueQ(Q);   // s 为二项式系数表第 k 行中“左上方”的值
      e = FrontQ(Q);      // e 为二项式系数表第 k 行中“右上方”的值
        printf(“%d ”, *e);                  // 输出 e 的值
        EnqueueQ(Q, *s+*e);    // 计算所得第 k+1 行的值入队列
      } while (e!=0);
   sq→rear=(sq→rear+1)％maxsize;
   sq→data[sq→rear]=0;
 }
```

可以看到，当计算完所有k+1行的值后，队列中的元素除了k+1个系数外，头尾还有两个“0”。所以这种情况下队列的最大空间应为 n+4。

链队列的基本概念

- 定义：
  - 队列的链式存储结构简称为链队列，它是限制仅在表头删除和表尾插入的单链表。
  - 解决了假溢出问题，同时采用了一种新的存储方法。
- 链队列的形式说明

```
typedef struct
{ linklist *front, *rear;
} linkqueue;
linkqueue *q;    /* q是链队列指针 */
```

单链表：

```
typedef int datatype;
typedef struct node
{datatype data;
 struct node *next;
}linklist;
```

- 注：
  - 队头结点前附加一个头结点，且头指针指向头结点。
  - 链队列*q 为空的条件：q→front==q→rear

链队列的图示说明

空的链队列

非空的链队列示意图

链队列的基本运算-1

- 置空队

```
linkqueue *SetNullQL(linkqueue *q)    /* 生成空链队列 *q  */
{  q→front=malloc(sizeof(linklist));  /*  申请头结点 */
   q→front→next=NULL;          /* 尾指针也指向头结点 */
   q→rear= q→front;
   return q;
}  / * SetNullQL */
```

- 判队空

```
int  EmptyQL(linkqueue *q)      /* 判别队列q是否为空 */
{   if (q→front==q→rear)
            return 1;                  /*  空队列返回“真”  */
    else return 0;
}  /* EmptyQL */
```

链队列的基本运算-2

- 取队头结点数据

```
datatype *FrontQL(linkqueue *q)   /* 取出链队列 q 的队头元素 */
{datatype *temp=(datatype *)malloc(sizeof(datatype));
 if (EmptyQL(q))        /* 队列空 */
    {  printf("queue is empty");     return  NULL;  }
 else * temp=q→front→next →data;
  return temp;       /* 返回队头元素  */
}  /* FrontQL */
```

- 入队

```
void EnqueueQL(linkqueue *q, datatype x) /* 将结点x加入队列q
的尾端 */
{    q→rear→next=malloc(sizeof(linklist));    /* 新结点插入尾端 */
     q→rear=q→rear→next;          /* 尾指针指向新结点 */
     q→rear→data=x;               /*  给新结点赋值  */
     q→rear→next=NULL;
}   /* EnqueueQL */
```

链队列的基本运算-3

- 出队

```
datatype *DequeueQL(linkqueue *q)     /* 将队头元素删除 */
{  datatype *temp;
   linklist *s;
   if (EmptyQL(q)){  printf("queue is empty");  return NULL; }
      else{   s = q→front→next;        /* 指向被删除的头结点 */
              if (s→next= =NULL)       /* 当前链队列的长度等于1 */
                { q→front→next=NULL;   q→rear=q→front;
                }else                  /* 当前链队列的长度大于1 */
                  { q→front→next=s→next;   /*  修改头结点的指针 */ }
               temp=(datatype *)malloc(sizeof(datatype));
               *temp=s→data;
                free (s);
                return temp;       /* 返回被删除结点的值 */
          }
}   /* DequeueQL */
```

划分子集问题

- 已知集合$A=\{a_1,a_2,...a_n\}$，并已知集合上的关系$R=\{(a_i,a_j)|a_i,a_j$属于A,i不等于j$\}$，其中(a_i,a_j)表示a_i与a_j之间的冲突关系。现要求将集合A划分成互不相交的子集$A_1,A_2...A_m(m\leq n)$，使任何子集上的元素均无冲突关系，同时要求划分的子集个数较少。
- 这类问题可以有各种各样的实际应用背景。例如在安排运动会比赛项目的日程时，需要考虑如何安排比赛项目，才能使同一运动员参加的不同项目不在同一日进行，同时又使比赛日程较短。

划分子集问题

- 我们就运动会比赛日程安排来说明解决这类问题的方法。
- 设共有9个比赛项目，则A={1,2,3,4,5,6,7,8,9}。项目报名汇总后得到有冲突的项目如下：
  R={(2,8),(9,4),(2,9),(2,1),(2,5),(6,2),(5,9),(5,6),(5,4),(7,5),(7,6),(3,7),(6,3)}
  问题是如何划分A，使A的子集A_i中的项目不冲突且子集数最少，即比赛天数较少。

划分子集算法思想

- 算法思想
  - 采用循环筛选法，以第一个元素开始，凡与第一个元素无冲突且与该组中的其它元素也无冲突的元素划归一组作为一个子集A_1；再将A中剩余元素按同样的方法找出互不冲突的元素划归第二组，即子集A_2；以此类推，直到A中所有元素都划归不同的组(子集)。

划分子集算法的数据结构

- 采用循环队列cq[n]，存放集合A的元素；
- 数组Result[n]用来存放每个元素的分组号；
- newr[n]为工作数组。
- 集合中元素的冲突关系设置一个冲突关系矩阵，由一个二维数组R[n][n]表示，若第i个元素与第j个元素有冲突则R[i][j]=1，否则R[i][j]=0。
- R={(2,8),(9,4),(2,9),(2,1),(2,5),(6,2),(5,9),(5,6),(5,4),(7,5),(7,6),(3,7),(6,3)}的矩阵表示如下：

R的矩阵表示

- R={(2,8),(9,4),(2,9),(2,1),(2,5),(6,2),(5,9),(5,6),(5,4), (7,5),(7,6),(3,7),(6,3)}的矩阵表示

R[q][q]：

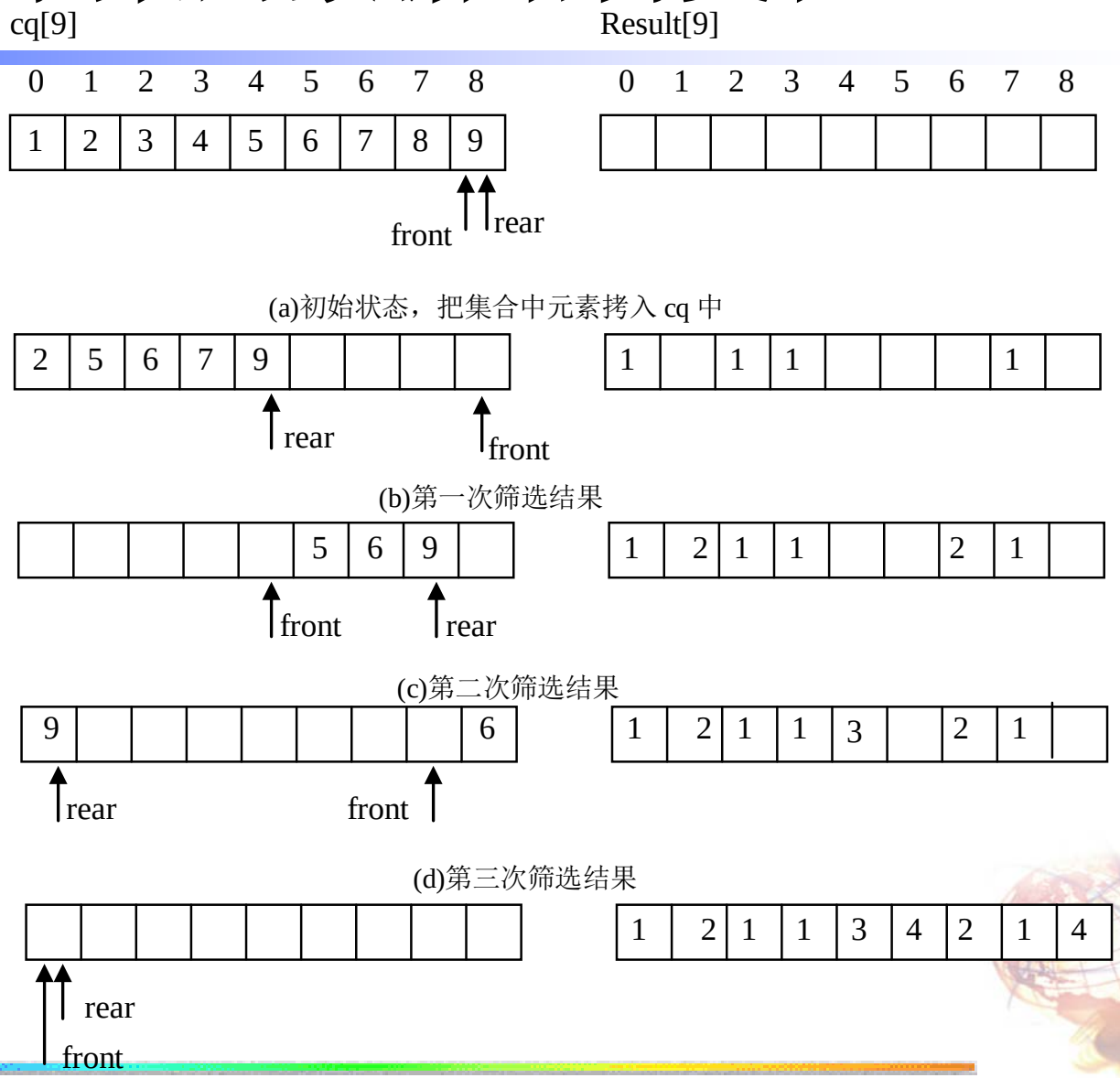
	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	0	0	0	0
1	1	0	0	0	1	1	0	1	1
2	0	0	0	0	0	1	1	0	0
3	0	0	0	0	1	0	0	0	1
4	0	1	0	1	0	1	1	0	1
5	0	1	1	0	1	0	1	0	0
6	0	0	1	0	1	1	0	0	0
7	0	1	0	0	0	0	0	0	0
8	0	1	0	1	1	0	0	0	0

划分子集的工作过程

- 初始状态：集合A中的元素放入cq中，Result和newr置零，设组号group=1。
- (1)第一个元素出队，将R中的第一行元素拷入newr中的对应位置，得到：　　newr[]={0，1，0，0，0，0，0，0，0}
- (2)考察第二个元素。因为第二个元素与第一个元素有冲突，不能将R中的第二行加到newr中去。newr保持不变。
- (3)考察第三个元素。由于它与第一个元素无冲突，将其归入group=1的组。
- (4)考察第四个元素。由于它与第一、第三个元素均无冲突故将其归入该组。newr变为：　newr[]={0，1，0，0，1，1，1，0，1}
- (5)第五至第七个元素与第一、第三或第四有冲突，故不归入。
- (6)考察第八个元素，由于与第一、三或四个元素无冲突，故归入该组。newr变为：　　newr[]={0，2，0，0，1，1，1，0，1}
- (7)第九个元素与元素一、三、四、八有冲突，故不能归入该组。
- 故第一组的元素应是{1，3，4，8}=A_1。

划分子集的数据结构变化

- 设group=2,newr清零，此时cq变为：
  cq[]={2，5，6，7，9} 重复上述过程即可完成分组，即子集的划分。
  其最后结果为：
  A_1={1,3,4,8},
  A_2={2,7},
  A_3={5},
  A_4={6,9}。



划分子集的算法实现

```
void DivideIntoGroup(int n, int R[][n], int cq[n], int result[n])
{       int front,rear,group,pre,I,i;
        front=n-1;
        rear=n-1;
        for(i=0;i<n;i++)
           { newr[i]=0;   cq[i]=i+1;  }
        group=1;        /*  以上是初始化过程  */
        pre=0;          /* 前一个出队元素的编号 */
        do{front =(front+1)％n;
             I=cq[front];
             if (I<pre)        /* 开始下一次筛选的准备 */
                { group=group+1;
                  result[I-1]=group;
                  for(i=0;i<n;i++)     newr[i]=R[I-1][i];
                } else if(newr[I-1]!=0)       /*发生冲突的元素重新入队*/
                        {   rear=(rear+1)％n;  cq[rear]=I;
                        }   else{
                                    result[I-1]=group;   /*不冲突，归入一组*/
                                    for(i=0; i<n; i++)   newr[i]+=R[I-1][i]; }
               pre=I;  /*下一次筛选*/
            }while(rear!=front)
}   /* DivideIntoGroup */
```

结论和习题

- 结论
  - 栈和队列都属线性结构，因此它们的存储结构和线性表非常类似，同时由于它们的基本操作要比线性表简单得多，因此它们在相应的存储结构中实现的算法都比较简单。
  - 这一章的重点在于栈和队列的应用。
- 习题和上机

  11, 16

例一

- 对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数？

  例如：$(1348)_{10} = (2504)_8$

例一答案

```
void conversion ()
{
// 对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数
   SetNullS(S);        // 构造空栈
   scanf("%d", &N);              // 输入一个十进制数
   while(N)
   {
      PushS(S, N % 8);    // "余数" 入栈
      N = N/8;              // 非零 "商" 继续运算
   } // while
   while (! EmptyS(S))
   {                          // 和 "求余" 所得相逆的顺序输出八进制的各位数
      e=PopS(S);
      printf("%d", *e);
   } // while
} // conversion
```

例二

- 迷宫求解问题：从入口进去之后是如何找到出口的？

  （东、南、西、北）

例二

- 迷宫求解问题：从入口进去之后是如何找到出口的？

  （东、南、西、北 ）

求迷宫中一条路径的算法的基本思想是：

若当前位置"可通"，则纳入"当前路径"，并继续朝"下一位置"探索；若当前位置"不可通"，则应顺着"来的方向"退回到"前一通道块"，然后朝着除"来向"之外的其他方向继续探索；若该通道块的四周四个方块均"不可通"，则应从"当前路径"上删除该通道块。

设定当前位置的初值为入口位置；

```
do{
    若当前位置可通，
    则{
        将当前位置插入栈顶；                    // 纳入路径
        若该位置是出口位置，则算法结束；
        // 此时栈中存放的是一条从入口位置到出口位置的路径
        否则切换当前位置的东邻方块为新的当前位置；
      }
    否则
    {
      若栈不空且栈顶位置尚有其他方向未被探索，
      则设定新的当前位置为: 沿顺时针方向旋转找到的栈顶位置的下一相邻块；
      若栈不空但栈顶位置的四周均不可通，
      则{ 删去栈顶位置；                        // 从路径中删去该通道块
          若栈不空，则重新测试新的栈顶位置，
          直至找到一个可通的相邻块或出栈至栈空；
        }
    }
} while (栈不空);
```

例三

- Hanoi塔（梵塔 ）问题的求解。例如：有三根针，设为A、B、C。A针上从大到小套着大小互不相等的n个金片，大的在下，小的在上。要求把这n个金片从A针移到C针，在移动过程中只能借助B针，每次只允许移动一个金片，同时要求无论在哪根针上都只允许小的金片压在大的上面。

例三

- Hanoi塔（梵塔 ）问题的求解。例如：有三根针，设为A、B、C。A针上从大到小套着大小互不相等的n个金片，大的在下，小的在上。要求把这n个金片从A针移到C针，在移动过程中只能借助B针，每次只允许移动一个金片，同时要求无论在哪根针上都只允许小的金片压在大的上面。

可以把这个问题分解为3个步骤：

（1） 首先把A针上的n-1个金片通过C针移到B针上。

（2） 把A针上余下的一个金片移到C针上。

（3） 最后借助A针将B针上的n-1个金片移到C针上。

例三答案

```
void hanoi (int n, char x, char y, char z)
      // 将塔座 x 上按直径由小到大且至上而下编号为1至 n
      // 的 n 个圆盘按规则搬到塔座 z 上，y 可用作辅助塔座。
 {
   if (n==1)
   {
      move(x, 1, z);            // 将编号为1的圆盘从 x 移到 z
   }
   else {
      hanoi(n-1, x, z, y); // 将 x 上编号为1至 n-1 的圆盘移到 y, z 作辅助塔
      move(x, n, z);            // 将编号为 n 的圆盘从 x 移到 z
      hanoi(n-1, y, x, z); // 将 y 上编号为1至 n-1 的圆盘移到 z, x 作辅助塔
   }
}
```

第2章习题

- 17.

```
int InitList(void)
{
    int num, i;
    printf("please input no.:\n");        //输入元素的个数
    scanf("%d", &num);
    if(num>MAXSIZE)
    {           printf("input error!\n");
        return (-1);
    }
     L = (sequenlist *)malloc(sizeof(sequenlist));   //为L分配内存空间
      if (L==NULL)
    {           printf("malloc memory error!");
        return (-1);
    }
     printf("please input L  ( %d elements):\n", num);    //输入数据元素内容
    for (i=0; i<num; i++)
        scanf("%d", &L->data[i]);
    L->last = i-1;
    return (0);
}
```

```
typedef int datatype;
#define maxsize 1024
typedef struct
 { datatype data[maxsize];
    int last;
  } sequenlist;
main()
{  ....
```

```
for(i=0; i<k; i++)
{ x=L->data[L->last];
   for(j=L->last; j>=1; j--)
      L->data[j]=L->data[j-1];
    L->data[0]=x;
}
```

时间复杂度：O(k×n)

```
[L = (sequenlist *)malloc(sizeof(sequenlist))]
  ....
}
```

- 21. 顺序表

```
int InverseList(void)
{
    int i,j;
    datetype temp;
    j = (L->last+1)/2;
    for(i=0; i<j; i++)
    {
     temp = L->data[i];
     L->data[i] = L->data[L->last-i];
     L->data[L->last-i] = temp;
    }
    return (1);
}
```

单链表：

```
int InverseLink(void)
{   linklist *p  ,*q ,*r;
    if( head->next && head->next->next)
       {     //当链表不是空表或单结点时
           p=head->next;       q=p->next;
           p->next=NULL;
                  //将开始结点变成终端结点
           r=p;
           while (q)
           {   //将后一结点变成开始结点
                p=q;   q=q->next ;
                p->next = r;  r = p;
           }
          head->next=r;   return  (1);
       }
    else return (0);
}
```

- 27.

```
linklist *UNION (linklist *headA, linklist *headB)
{ linklist *pa,*pb,*pc,*qc,*s;
  pa=headA->next;  pb=headB->next; qc=NULL;
  while((pa!=NULL)&&(pb!=NULL) )
   { if (pa->data<=pb->data)
        { pc=pa;
          pa=pa->next;
        }
     else { pc=pb; pb=pb->next;}
     pc->next=qc;
     qc=pc;
   }
   If (pa=NULL) pc=pb;
   else pc=pa;
   s=pc->next;
   pc->next=qc;
   qc=pc;
   while (s)
    {   pc=s;
        s=s->next;
        pc->next=qc;
        qc=pc;
     }
    headA->next=pc;  free(headB);
   return(headA)
}
```