数据结构与算法

Data Structure and Algorithms

西安交通大学自动化系

蔡忠闽 周亚东

第二章　线 性 表

- 2.1 线性表的类型定义
- 2.2 线性表的顺序表示和实现
- 2.3 线性表的链式表示和实现
  - 2.3.1　线性链表
  - 2.3.2　循环链表
  - 2.3.3　双向链表
- 2.4 一元多项式的表示及相加

第三章 栈和队列

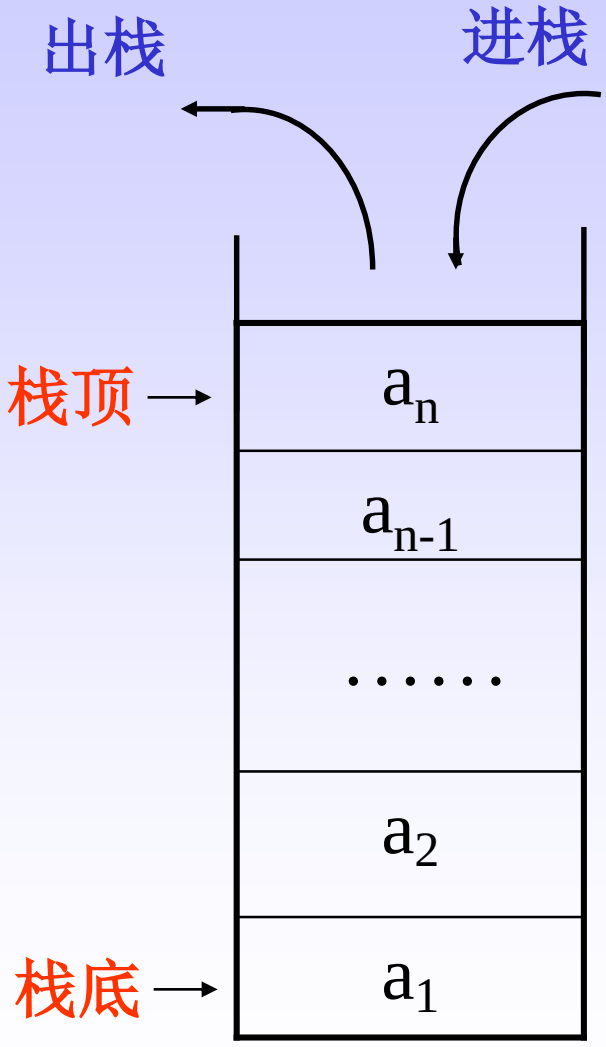
- 3.1、栈
- 3.2、栈的应用举例
- 3.3、队列

3.1 栈(stack)

3.1.1 定义

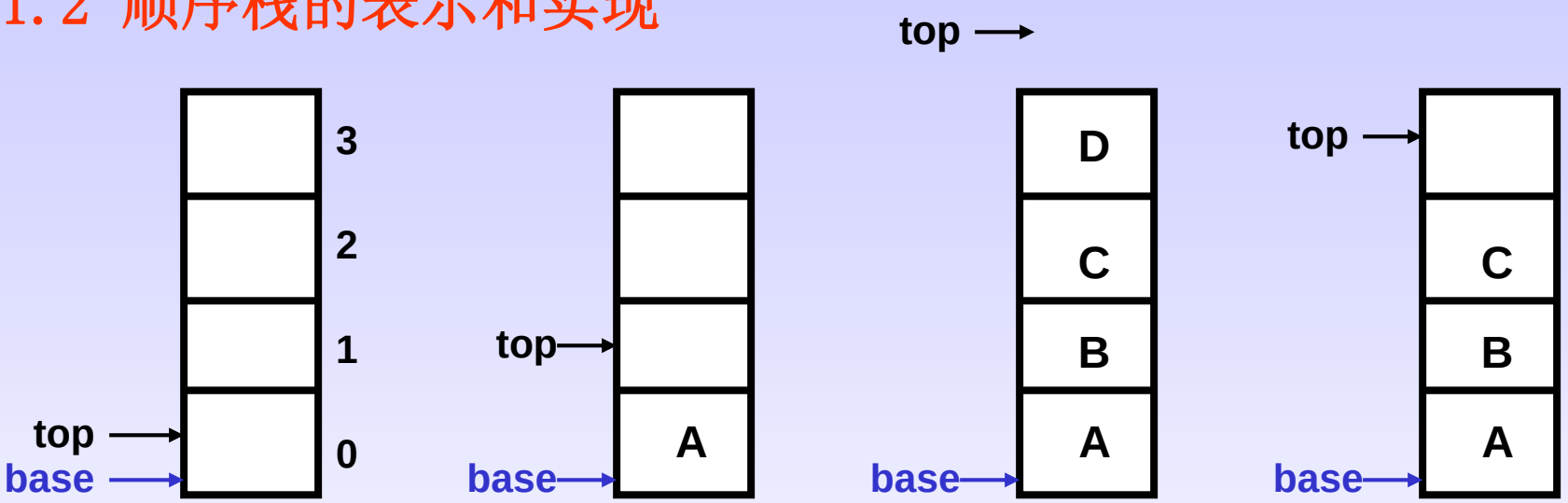
栈(Stack)是限制在表的一端进行插入和删除运算的线性表，通常称插入、删除的这一端为栈顶(Top)，另一端为栈底(Bottom)。当表中没有元素时称为空栈。

假设栈S=(a1，a2，a3，…an)，则a1称为栈底元素，an为栈顶元素。栈中元素按a1，a2，a3，…an的次序进栈，退栈的第一个元素应为栈顶元素。换句话说，栈的修改是按后进先出的原则进行的。因此，栈称为后进先出表（Last In First Out，LIFO）。



3.1 栈

3.1.2 顺序栈的表示和实现



利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时附设指针top指示栈顶元素在顺序栈的位置

```
typedef struct {
    ElemType *base;
    ElemType *top;
    int      stacksize;
} SqStack;
```

注意：因为 **base == top** 是栈空标志，所以 **top** 指针只能指示真正的栈顶元素之上的数组元素的下标地址。否则造成矛盾。

栈满时的处理方法：

1、提示出错，返回操作系统。

2、分配更大的空间。

3.1 栈

2 顺序栈的 Push 操作实现：

```
Status Push (SqStack &s, SElemType e)
{  if ( s.top - s.base >= s.stacksize )  // 判断是否栈满
   { s.base=(SElemType *) realloc( s.base,
            (s.stacksize+STACK_INCREMENT)*sizeof(SElemType));
     if ( !s.base )   exit ( OVERFLOW );
     s.top = s.base + s.stacksize ;
     s.stacksize  += STACK_INCREMENT;
   }
   *s.top ++=  e;    // 相当于： * s.top = e;  s.top ++ 两条指令;
   return OK;
}  // Push;
```

3.1 栈

3 顺序栈的 pop 操作实现：

```
Status Pop (SqStack &s, SElemType &e)
{    if ( s.top == s.base ) // 判断是否栈空
         return ERROR;
     s.top--;
     e = *s.top
     return OK ;
} // Pop
```

e = * - - s.top

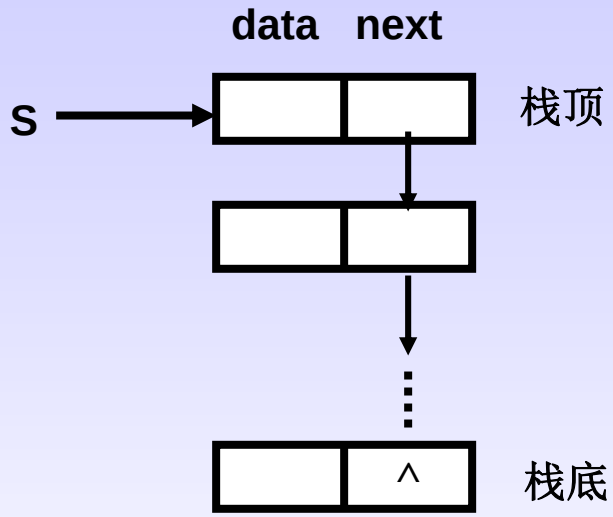
Note:

e = * (s.top - 1) → GetTop 操作

3.1 栈

3.1.2 链式栈的表示和实现

```
typedef struct Snode {
        SElemType          data;
        struct   Snode     *next;
} Snode,  *LinkStack;
```



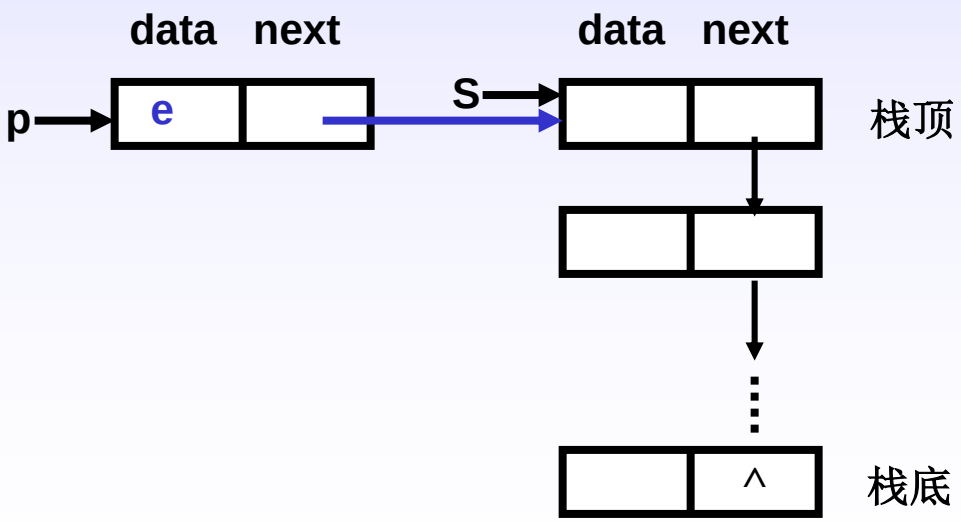
1 链式的栈的初始化实现：

```
void InitlinkStack(LinkStack &s)
{ s = NULL; }  // InitlinkStack，没有头结点，顶，底？
```

3.1 栈

2 链式栈的 Push 操作实现：

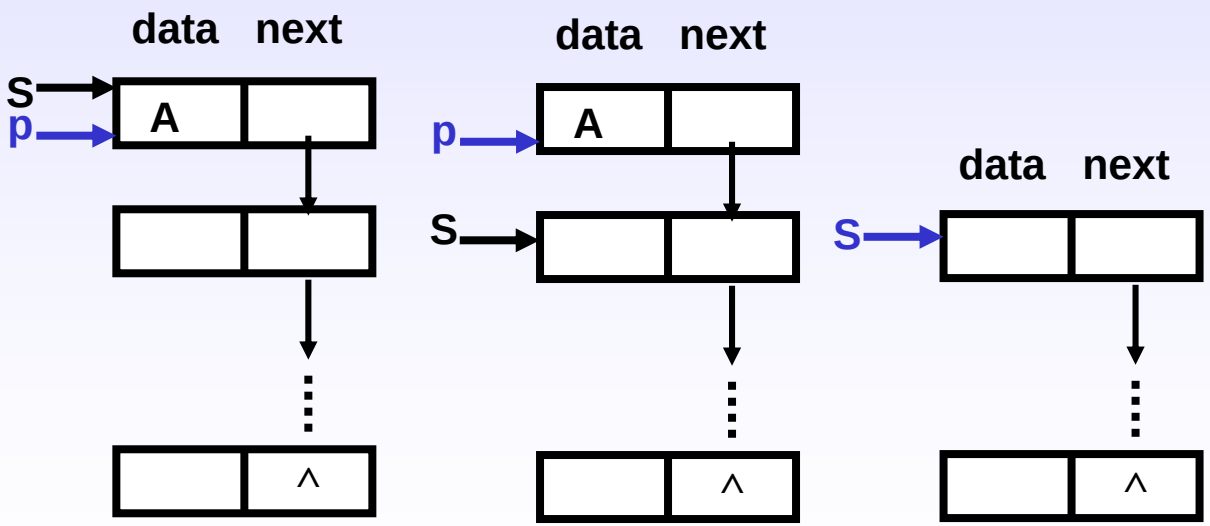
```
Status Push(LinkStack &s, SElemType e)
{  p = ( Snode * ) malloc (sizeof(Snode) );
   p->data = e;    p->next = s;
   s = p;
   return OK;
 }  // Push;
```



3.1 栈

3 链式栈的 Pop 操作实现：

```
Status Pop(LinkStack &s, SElemType & e)
{  if ( !s )
        return ERROR;
   e = s->data;
   p = s;
   s = s->next;
   free(p);
   return OK;
 }  // Pop;
```



3.2 栈的应用--数制转换

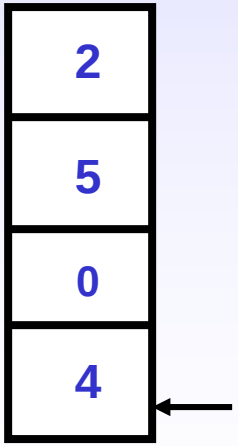
公式： **N=(N div d)*d + N mod d** （div为整除, mod为求余）

```
void conversion ( )
{  InitStack( S );
   scanf( "%d", &N );
   while (N)
   {  Push(S, N%8);
      N = N/8;
   }
   while ( !Stackempty(S))
   {  Pop(S, e);
      printf ("%d" ,  e );
   }
}
```

例如10 进制和 8 进制之间的数的转换。

$(1348)_{10} = 8^3 * a_3 + 8^2 * a_2 + 8 * a_1 + 8^0 * a_0$

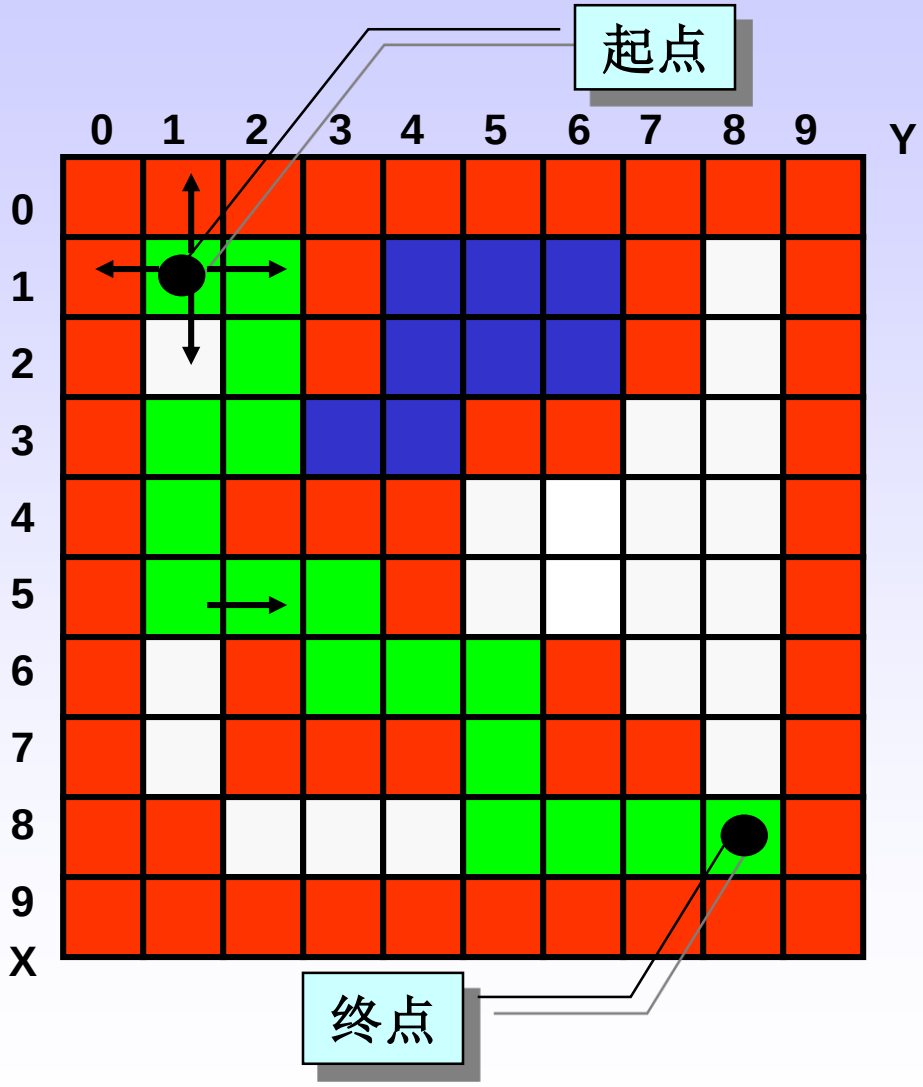
N	N div 8	N % 8	
1348	168	4	a0
168	21	0	a1
21	2	5	a2
2	0	2	a3



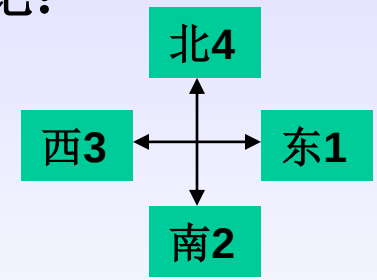
3.2 栈的应用--括号匹配检查

```
Status  clarity(Sqlist  L)  // L中存放需要检查的表达式
{  Initstack(S);
   for(i=1; i<=Listlength(L); i++)
   {  GetElem(L, i, e1);
      if(( e1 == '(' ) || ( e1 == '[' ))   Push(S, e1); //左括号入栈
      else if(( e1 == ')' ) || ( e1==']' ))   //右括号出栈
           if(( Pop(S, e2) == ERROR )   // 左括号少于右括号
              || ( e1== ')'  && e2 != '(' ) // 左右括号不匹配
             || ( e1== ']'  && e2 != '[' ) )
            return ERROR;
   }
   if  StackEmpty(S)   return OK;
   else  return ERROR;                    // 左括号多于右括号
}
```

3.2 栈的应用——求从起点到终点的简单路径



- 方格的四种类型：
  - ·非墙且未经试探的方格
  - ·墙
  - ·已在路径上的方格
  - ·已试探过的无发展前途的方格
- 方向标记：



- 起点：(x=1,y=1);

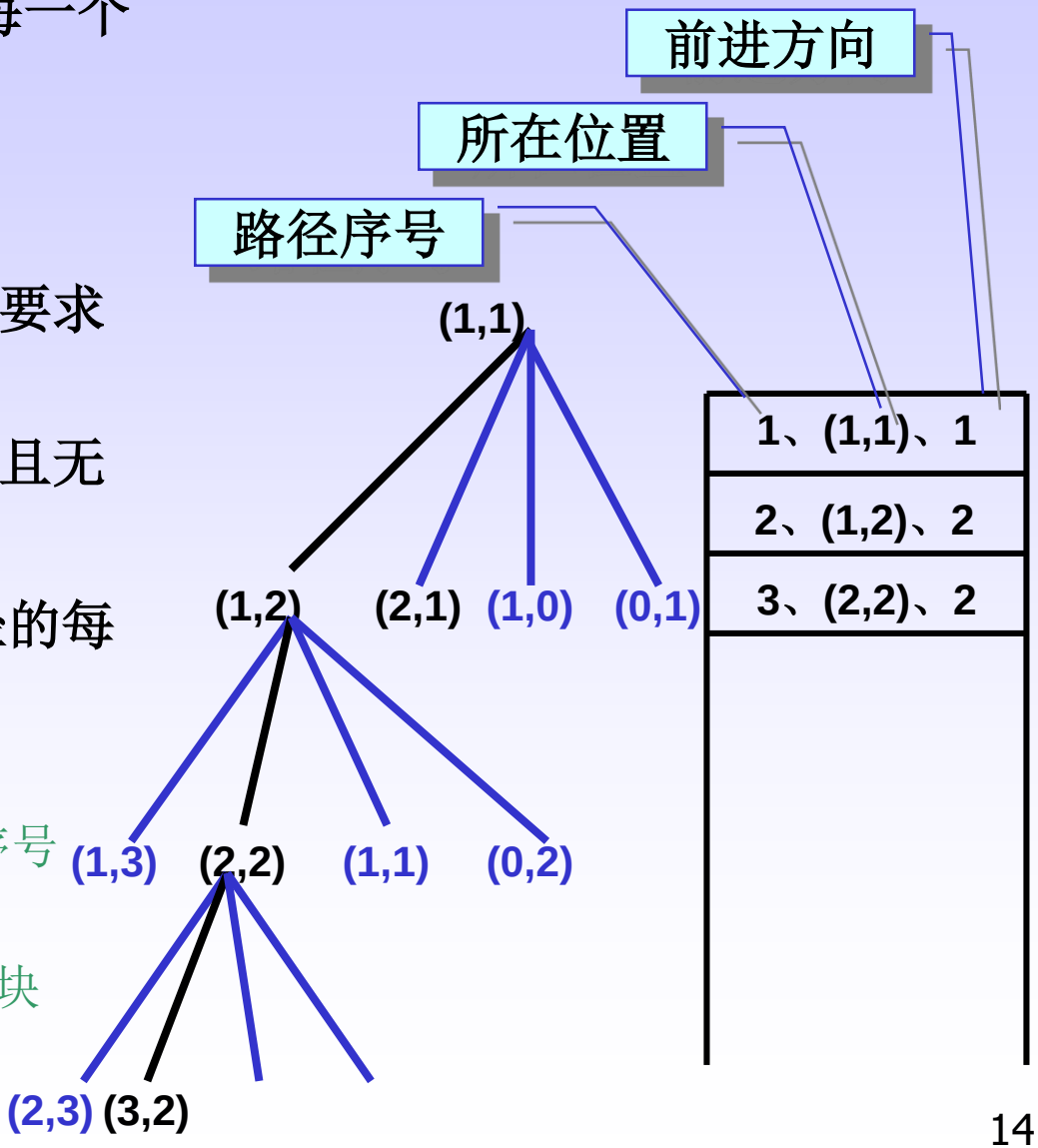
  东(1,2) 南(2,1) 西(1,0) 北(0,1)

演示

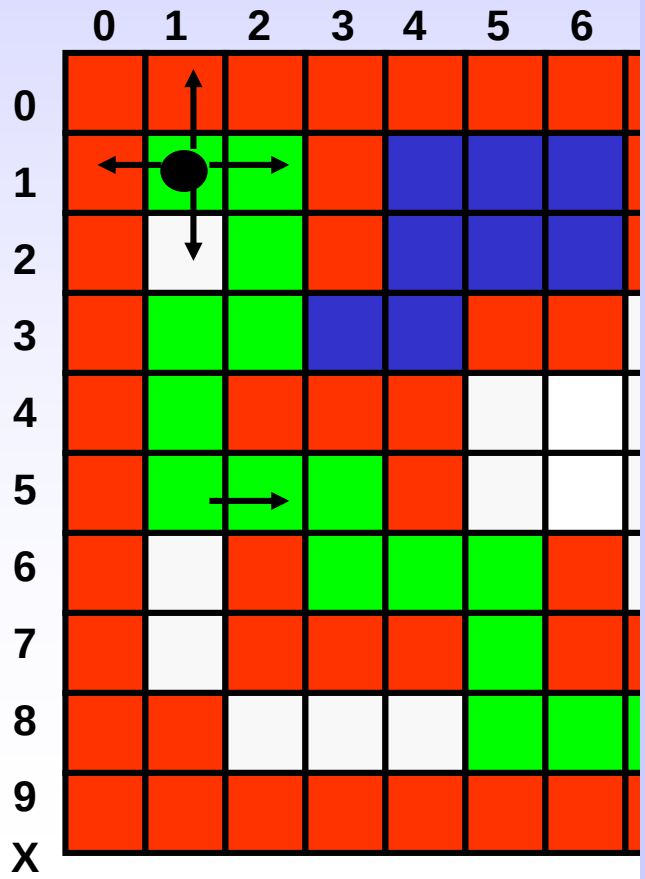
3.2 栈的应用--求从起点到终点的简单路径

- 试探方法： 穷举求解，试探每一个可能的方向。
- 可能的方向：
  - 非墙新方格
  - 不是在路径上的方格（因为要求从起点到终点的简单路径）
  - 不是曾经纳入过路径的方格且无发展前途的
- 堆栈中记录的数据（组成路径的每个点）类型：

```
typedef struct
 { int        step; 块在路经的序号
   PosType seat; 块的坐标位置
   int        di; 从此块走向下一块
                  的方向
 } ElemType;
```



```
Status MazePath ( MazeType maze, PosType start, PosType end  )
{  Initstack(S);  curpos = start; curstep = 1;
   do{ if ( Pass(curpos) )                    // 判断当前位置能否通过
       { FootPrint(curpos);                   // 标记已访问过的位置
         e = ( curstep,curpos,1 ); Push( S, e );  // 加入路径
         if  ( curpos == end  )  return ( TRUE  );  // 已到达终点
         curpos = NextPos( curpos, 1);  // 得到东邻位置，注意1代表东。
         curstep ++;
       } // if
       else {  // 当前位置不能通过
       if  ( !StackEmpty( S ) ) {
         Pop( S,e );                          // 退回到上次来的位置
        while ( e.di == 4 && !StackEmpty(S) )  // 当前位置不在路径上
          { MarkPrint(maze, e.seat); Pop(S,e); }  // 设置非路径标志, 后退一步
        if ( e.di < 4) { e.di++; Push(S,e); curpos = NextPos(e.seat, e.di); }
       }  // 在下一个方向继续探索
   } while ( !StackEmpty(S));
   return ( FALSE );  //  不存在从起点到终点的路径
} // MazePath
```




```
Status MazePath ( MazeType maze, PosType start, Pos
{  Initstack(S);  curpos = start; curstep = 1;
   do{ if ( Pass(curpos) )                    // 判断当前位置
       { FootPrint(curpos);                   // 标记已访问过
         e = ( curstep,curpos,1 ); Push( S, e );  // 加
         if  ( curpos == end  )  return ( TRUE  );  //
         curpos = NextPos( curpos, 1);  // 得到东邻位
         curstep ++;
       } // if
       else {  // 当前位置不能通过
       if  ( !StackEmpty( S ) ) {
         Pop( S,e );                          // 退回到上次来
        while ( e.di == 4 && !StackEmpty(S) )  // 当
          { MarkPrint(maze, e.seat); Pop(S,e); }  // 该
        if ( e.di < 4) { e.di++; Push(S,e);
              curpos = NextPos(e.seat, e.di); }}  // 在
   } while ( !StackEmpty(S));
   return ( FALSE );  //  不存在从起点到终点的路径
} // MazePath
```

3.2 栈的应用--求从起点到终点的简单路径

回溯

- 对一个包含有许多结点，且每个结点有多个分支的问题，可以先选择一个分支进行搜索。当搜索到某一结点，发现无法再继续搜索下去时，可以沿搜索路径回退到前一结点，沿另一分支继续搜索。
- 如果回退之后没有其他选择，再沿搜索路径回退到更前结点，….依次执行，直到搜索到问题的解，或搜索完全部可搜索的分支没有解存在为止。
- 回溯法与分治法本质相同，可用递归求解。

递归

一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，成为递归函数。递归是程序设计中的强有力工具。

1）很多函数是递归定义的，如阶乘函数；

2）有的数据结构本身具有递归特性，如二叉树，其操作可以采用递归描述；

3）有些问题采用递归求解更为简单，如八皇后问题。

自顶向下、逐步分解的策略

- 子问题应与原问题做同样的事情，且更为简单；
- 解决递归问题的策略是把一个规模比较大的问题分解为一个或若干规模比较小的问题，分别对这些比较小的问题求解，再综合它们的结果，从而得到原问题的解。

— 分而治之策略（分治法）

- 这些比较小的问题的求解方法与原来问题的求解方法一样。

递归

1）把父问题分解成子问题；

2）有一个可解的子问题。

递归

1）把父问题分解成子问题；

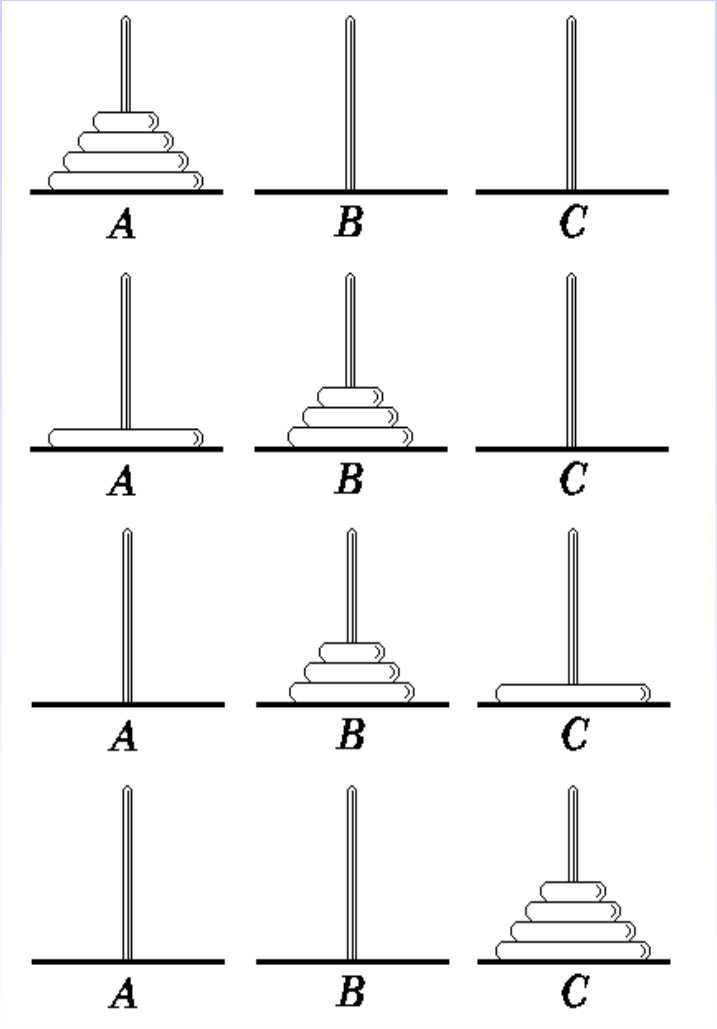
2）有一个可解的子问题。

数学归纳法

3.2 栈的应用—hanoi塔问题

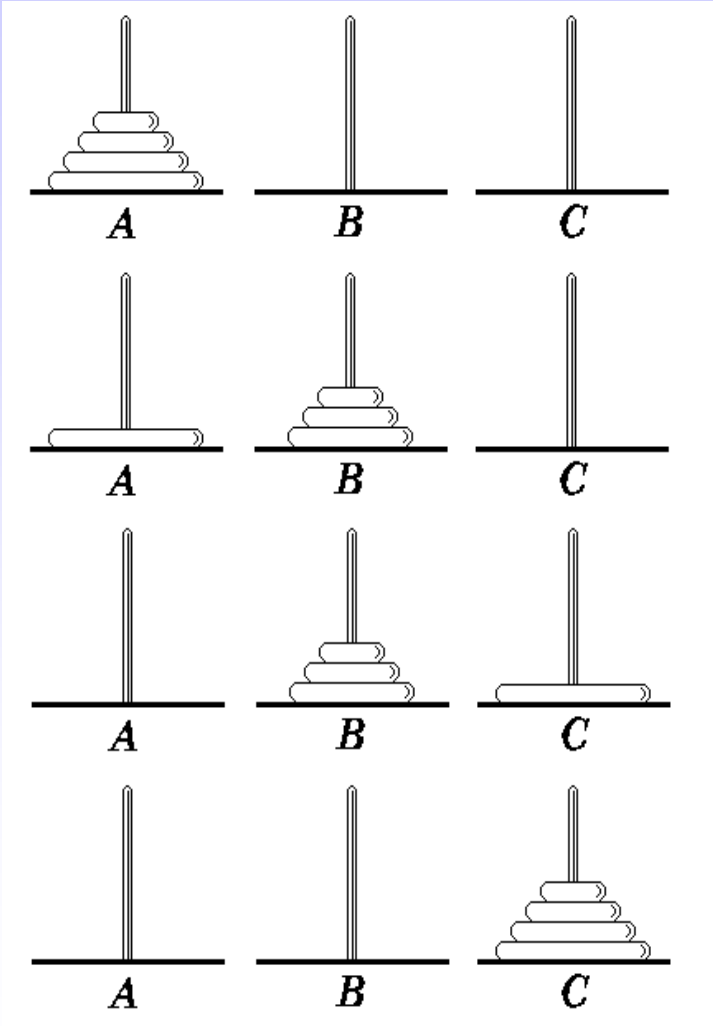
```
void Hanoi (int n, char A, char B, char C)
{

}
```



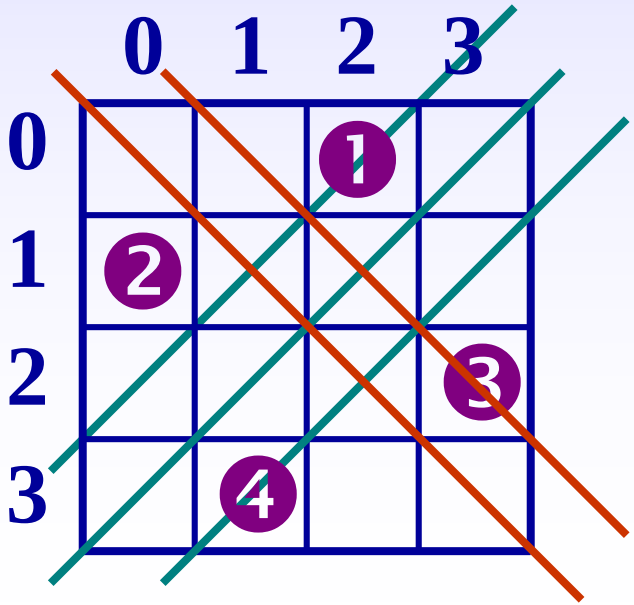
3.2 栈的应用—hanoi塔问题

```
void Hanoi (int n, char A, char B, char C)
{
//解决汉诺塔问题的算法
    if (n == 1)
        Move(A, 1, C)
    else {
        Hanoi(n-1, A, C, B);
        Move(A, n, C)
        Hanoi(n-1, B, A, C);
        }
}
```



3.2 栈的应用--n皇后问题

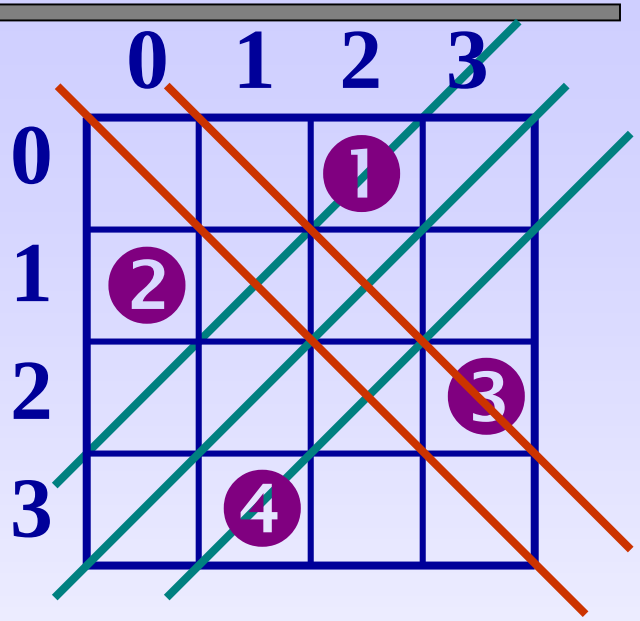
- 在 n 行 n 列的国际象棋棋盘上，若两个皇后位于同一行、同一列、同一对角线上，则称为它们为互相攻击。n 皇后问题是指找到这 n 个皇后的互不攻击的布局。



- 安放第 i 行皇后时，需要在列的方向从 0 到 n−1 试探 (j = 0, …, n−1)
- 在第 j 列安放一个皇后：
  - 如果在列、主对角线、次对角线方向有其它皇后，则出现攻击，撤消在第 j 列安放的皇后。
  - 如果没有出现攻击，在第 j 列安放的皇后不动，递归安放第 i+1行皇后。

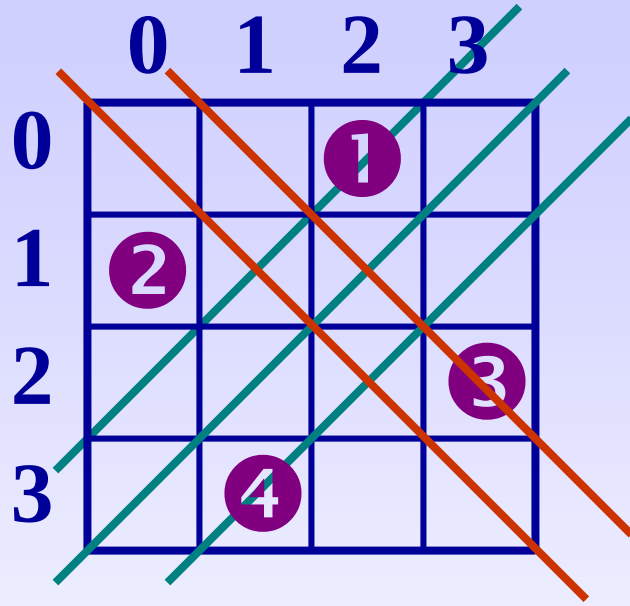
3.2 栈的应用--n皇后问题

- NQueen(int i, int * qipan, int n)
- Right_position(i, j, qipan)
- setposition (i, j, qipan)
- removeposition(i, j, qipan)



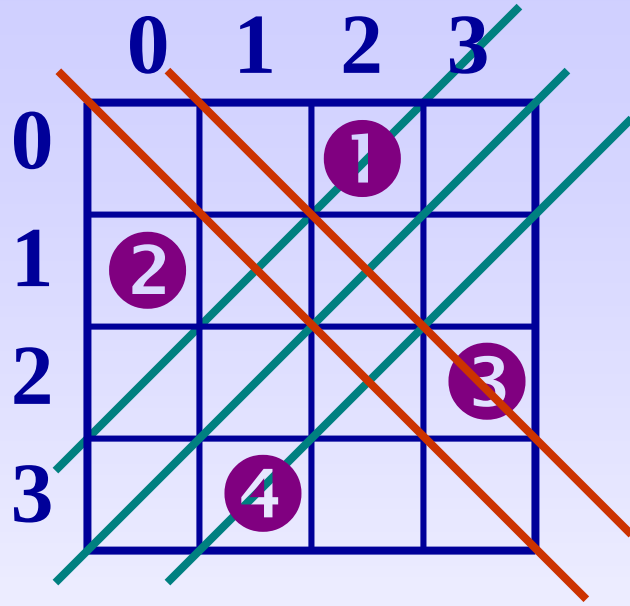
3.2 栈的应用--n皇后问题

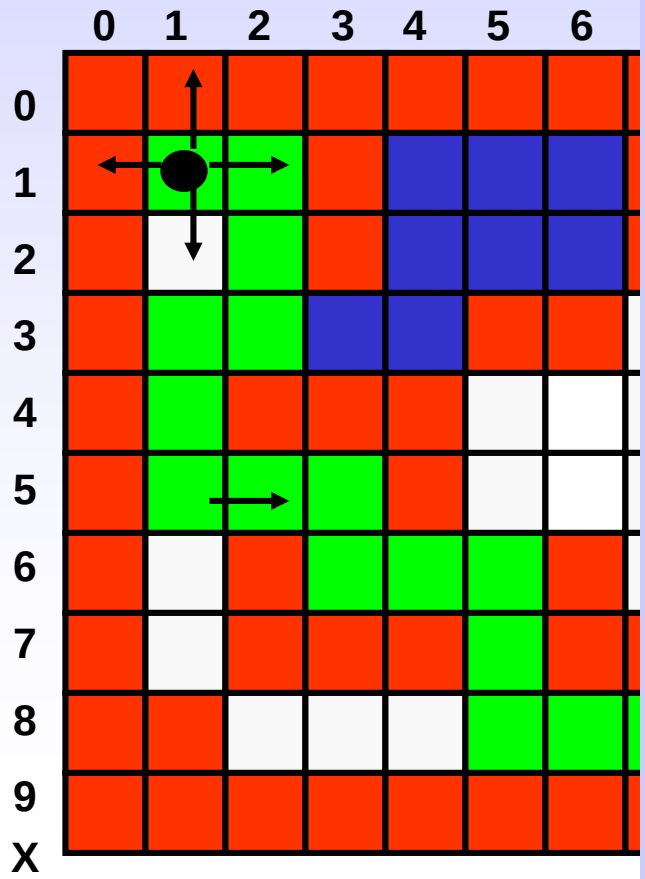
```
Boolean NQueen(int i, int * qipan, int n)
{
    for (j=1; j<=n;j++)
    {
        if Right_position(i, j, qipan)
        {
            setposition (i, j,qipan)
            if Nqueen(i+1,qipan,n)
                return true;
            removeposition(i,j,qipan)
        }
     Return false
    }
}
```



3.2 栈的应用--n皇后问题

```
Boolean NQueen(int i, int * qipan, int n)
{
   for (j=1; j<=n;j++)
   {
      if Right_position(i, j, qipan)
      {
         setposition (i,j,qian)
         if (i==n) || Nqueen(i+1,qipan,n)
            return true;
         removeposition(i,j,qipan)
       }
     Return false
   }
}
```






```
Status MazePath ( MazeType maze, PosType start, Pos
{  Initstack(S);  curpos = start; curstep = 1;
   do{ if ( Pass(curpos) )                    // 判断当前位置
       { FootPrint(curpos);                   // 标记已访问过
         e = ( curstep,curpos,1 ); Push( S, e );  // 加
         if  ( curpos == end  )  return ( TRUE  );  //
         curpos = NextPos( curpos, 1);  // 得到东邻位
         curstep ++;
       } // if
       else {  // 当前位置不能通过
       if  ( !StackEmpty( S ) ) {
         Pop( S,e );                          // 退回到上次来
        while ( e.di == 4 && !StackEmpty(S) )  // 当
          { MarkPrint(maze, e.seat); Pop(S,e); }  // 该
        if ( e.di < 4) { e.di++; Push(S,e);
              curpos = NextPos(e.seat, e.di); }}  // 在
   } while ( !StackEmpty(S));
   return ( FALSE );  //  不存在从起点到终点的路径
} // MazePath
```

3.2 栈的应用—表达式求值

基本思想：

表达式有操作数、运算符和界限符组成，称之为单词。

计算要符合四则运算法则：

1) 计算从左到右

2) 先乘除，后加减

3) 先算括号内，后算括号外

运算符和界限符统称为算法，其集合为OP，故算符q_1 与 q_2 满足以下关系（表3.1）：

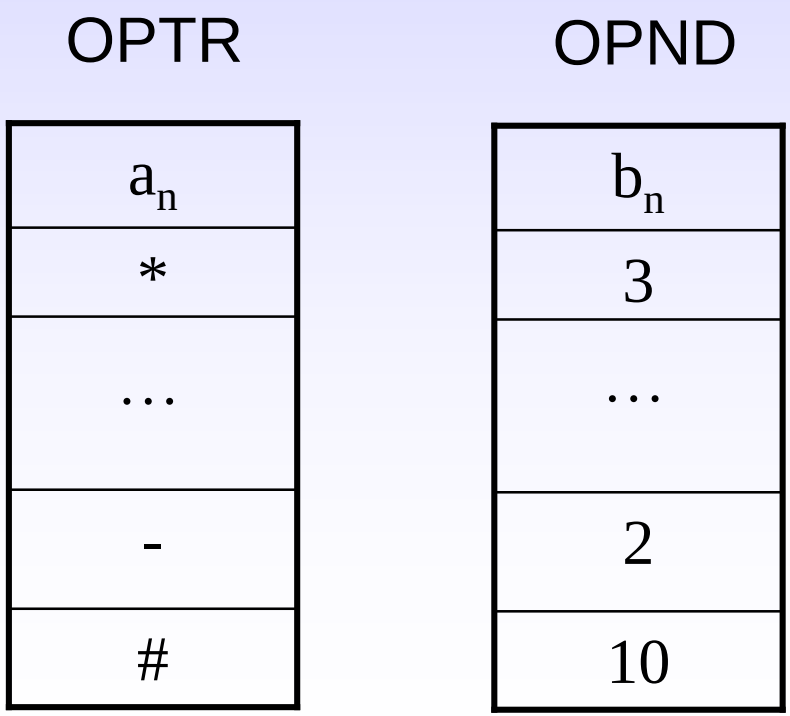
$q_1 < q_2$　　q_1 的优先权低于 q_2

$q_1 = q_2$　　q_1的优先权等于 q_2

$q_1 > q_2$　　q_1的优先权高于 q_2

3.2 栈的应用—表达式求值

- 1) 置操作数栈OPND为空栈， 运算符栈的栈底元素为#；
- 2) 依次读入字符，操作数进OPND栈，运算符则与OPTR栈的栈顶元素比较优先权后进行相应操作，直至结束。



OperandType EvaluateExpression()

```
// OPTR和OPND分别是运算符栈和运算数栈，OP是运算符集合；
InitStack (OPTR);  Push (OPTR, '#');
InitStack (OPND);  c = getchar ();
while (c != '#' || GetTop (OPTR) !=  '#' ){
    if ( ! In(c, OP)) {Push ((OPND, c); c = getchar ();}
    else
        switch (Precede(GetTop(OPTR), c)) {
            case  '<': Push (OPTR, c); c = getchar (); break;  //栈顶元素优先权低
            case  '=': Pop(OPTR, x); c = getchar (); break;
            case  '>': Pop(OPTR, theta); Pop(OPND, b);      //栈顶元素优先权高
                       Pop(OPND, a); Push(OPND, Operater(a, theta, b));
                       break;
        }
    return GetTop(OPND);
}
```

第三章 栈和队列

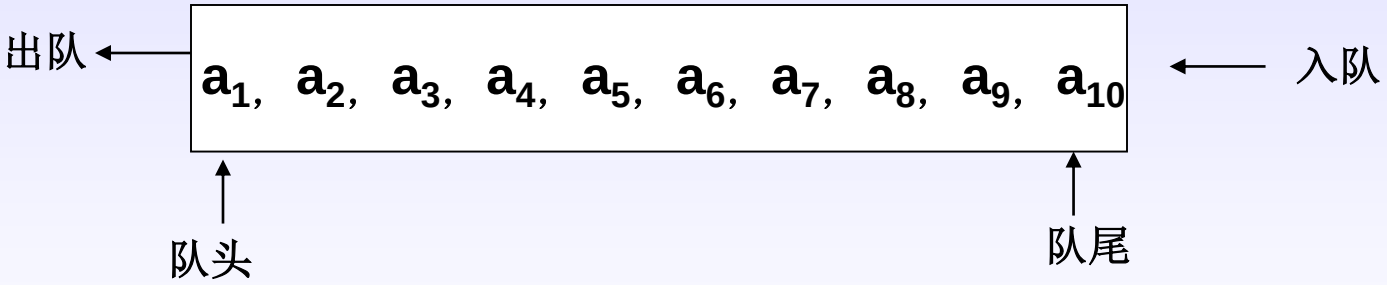
- 3.1、栈
- 3.2、栈的应用举例
- 3.3、队列

3.4 队列

3.4.1 定义：

队列：在表的一端进行插入，而在另一端进行删除的线性表。

特点：先进先出（First In First Out,FIFO）。



3.1 队列(Queue)

- ADT Queue {

数据对象：D＝{ ai | ai ∈ElemSet, i=1,2,...,n, n≥0 }

数据关系：R1＝{ <ai-1 ,ai >| ai-1, ai ∈D, i=2,...,n }

约定a_n端为对列尾，a_1端为对列头

基本操作：

InitQueue (&Q)　　//构造一个空对列

DestroyQueue (& Q)　//销毁对列

ClearQueue (& Q)　　//将S清为空对列

QueueEmpty(Q)　　//判断是否为空对列，是则返回True

QueueLength(Q)　　//返回对列的长度

GetHead (Q, &e)　　// 返回队头元素

EnQueue (& Q, e)　　//插入元素e为新的队尾元素

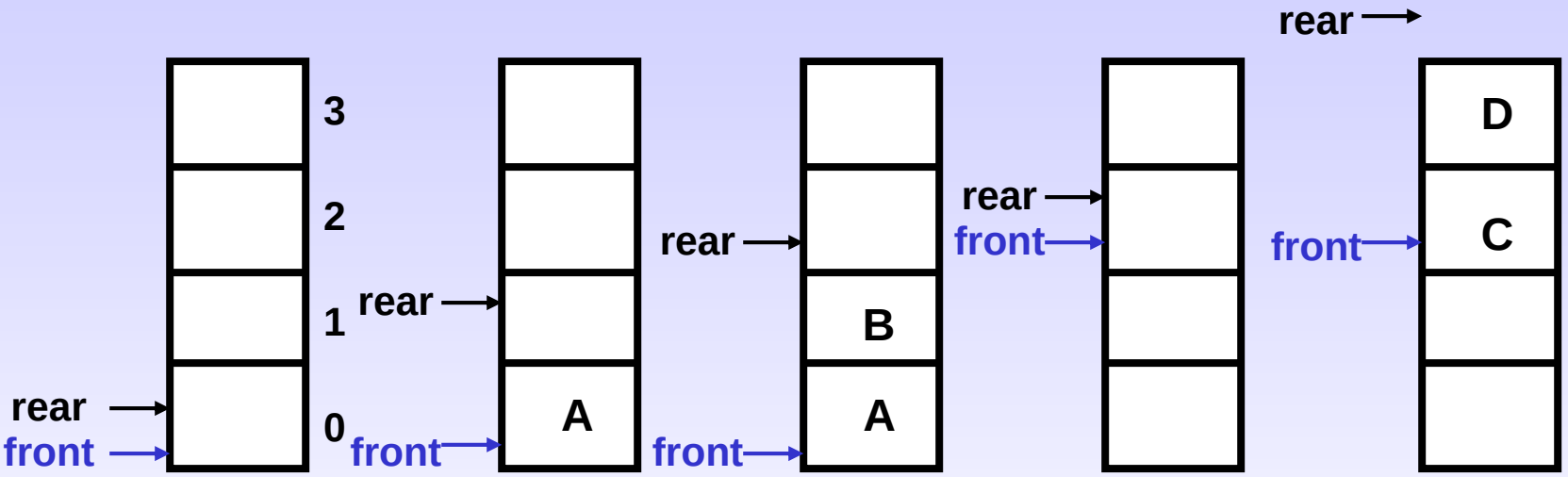
DeQueue (& Q, &e)　// 删除队头元素，并用e返回

QueueTraverse(Q, visit())　//对每个元素都调用visit函数，如调用失败，则操作失效

}

3.4 队列

3.4.2 顺序表示的队列：




```
typedef struct {
   QElemType   *base;
   int          front;
   int          rear;
} SqQueue ;
```

入队: **base[rear++]=x;**

出队: **e=base[front++];**

队空标志: **front == rear**

队满标志: **rear>=MAXQSIZE (假溢出)**

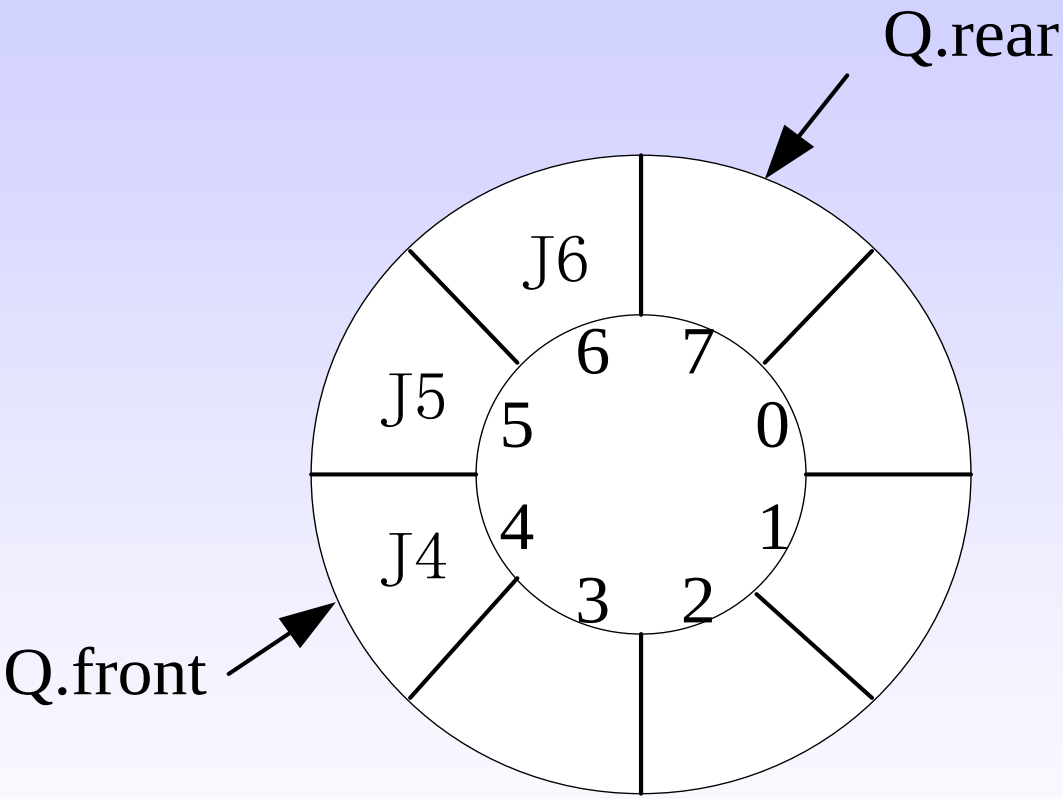
3.4 队列

- 为充分利用空间，克服上述假溢出现象的方法是将空间想象为一个首尾相接的圆环（图3.13），并称这种队列为循环队列。
- 在循环队列中进行出队、入队操作时，头尾指针仍要加1，朝前移动。只不过当头尾指针指向向量上界（MAXQSIZE-1）时，其加1操作的结果是指向向量的下界0。这种循环意义下的加1操作可以描述为：

```
if(Q.rear+1 == MAXQSIZE)        Q.rear=0;
else     Q.rear++;
```

利用模运算可简化为： Q.rear = (Q.rear+1) % MAXQSIZE

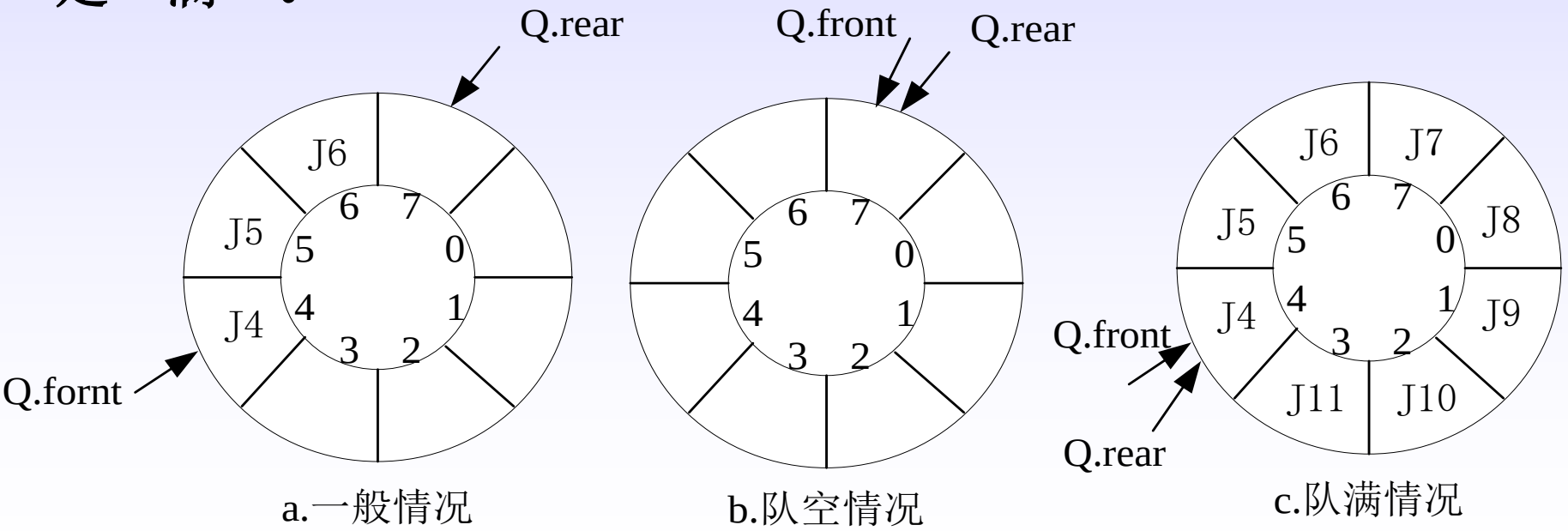
3.4 队列



a.一般情况

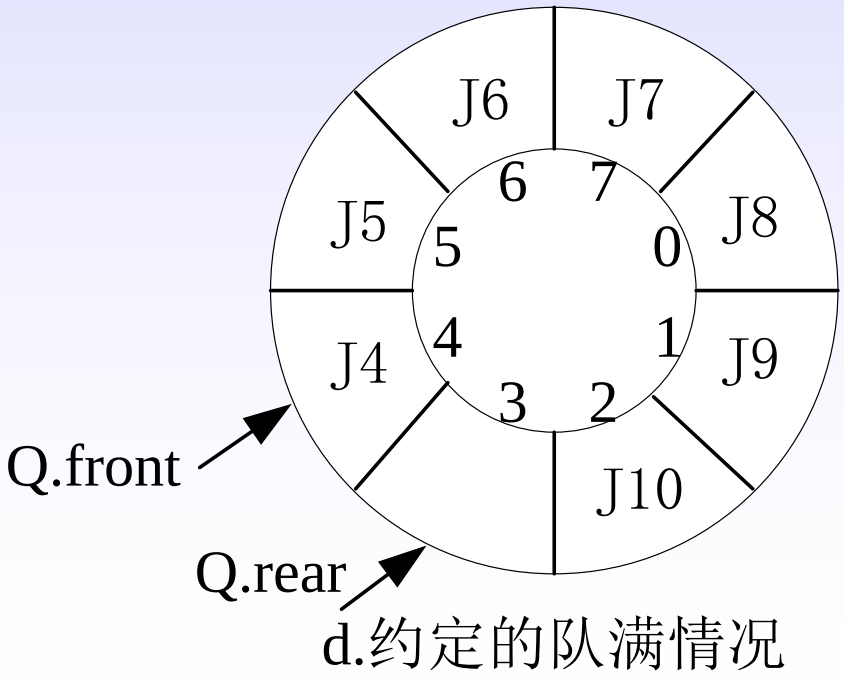
3.4 队列

• 如图所示：由于入队时尾指针向前追赶头指针，出队时头指针向前追赶尾指针，故队空和队满时头尾指针均相等（图b、c）。因此，无法通过Q.front==Q.rear来判断队列“空”还是“满”。



3.4 队列

解决此问题的方法是少用一个元素的空间，约定入队前，测试尾指针在循环意义下加1后是否等于头指针，若相等则认为队满（图d，注意：Q.rear所指的单元始终为空）。



d.约定的队满情况

3.4 队列的状态判断

队满：

条件 ((Q.rear + 1) % MAXQSIZE) == Q.front

注意 Q.rear 是否会由最高下标跳至最低下标(循环)

队空：

条件 Q.rear == Q.front

注意 Q.front 是否会由最高下标跳至最低下标(循环)

3.4 队列

基本操作的实现程序：

```
Status InitQueue (SqQueue &Q )
{  Q.base = (QElemType *)
                malloc(MAXQSIZE * sizeof(QElemType));
                // 分配队列的存储空间;
   if ( !Q.base )  exit( OVERFLOW ) ;
   Q.front = Q.rear = 0; // 队头、尾指针清0
    return OK;
}
```

3.4 队列

入队时应先判队是否满：

条件 ((Q.rear + 1) % MAXQSIZE) == Q.front

注意 Q.rear 是否会由最高下标跳至最低下标(循环)

```
Status EnQueue (SqQueue &Q,  QElemType  e )
{ if ((( Q.rear + 1) % MAXQSIZE ) == Q. front )
        return( ERROR ) ;  // 思考：此处能否采用realloc函数
   Q.base [ Q.rear ] = e;  //          扩大队列的存储空间？
   Q.rear = (( Q.rear + 1) % MAXQSIZE );
   return OK;
 }  // EnQueue ;
```

3.4 队列

出队时应先判断队是否空：

条件 Q.rear == Q.front

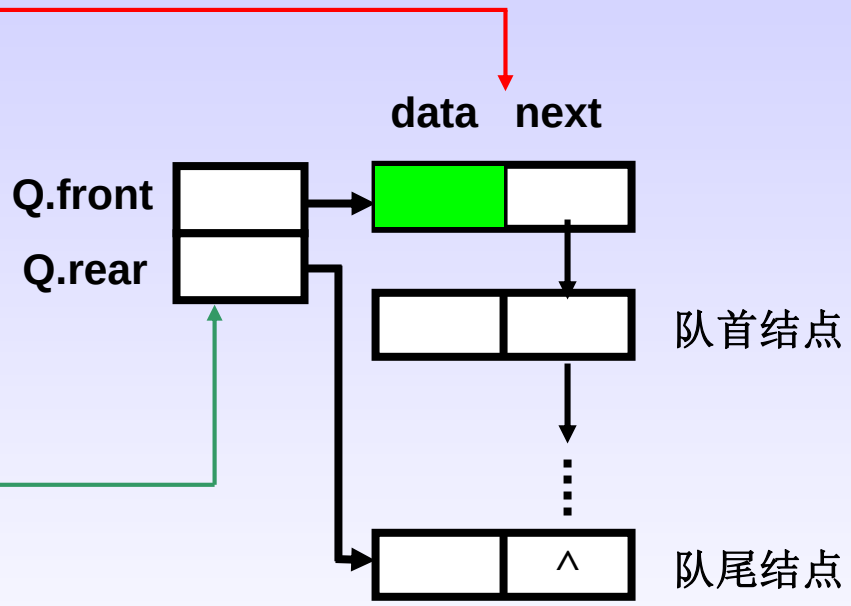
注意 Q.front 是否会由最高下标跳至最低下标(循环)

```
Status DeQueue (SqQueue &Q,  QElemType  &e )
{ if (  Q.rear == Q.front)
      return( ERROR ) ;
   e = Q.base [ Q.front ] ;
   Q.front = ( Q.front + 1) % MAXQSIZE ;
   return OK;
 }  // DeQueue ;
```

3.4 队列

3.4.3 链式表示的队列：

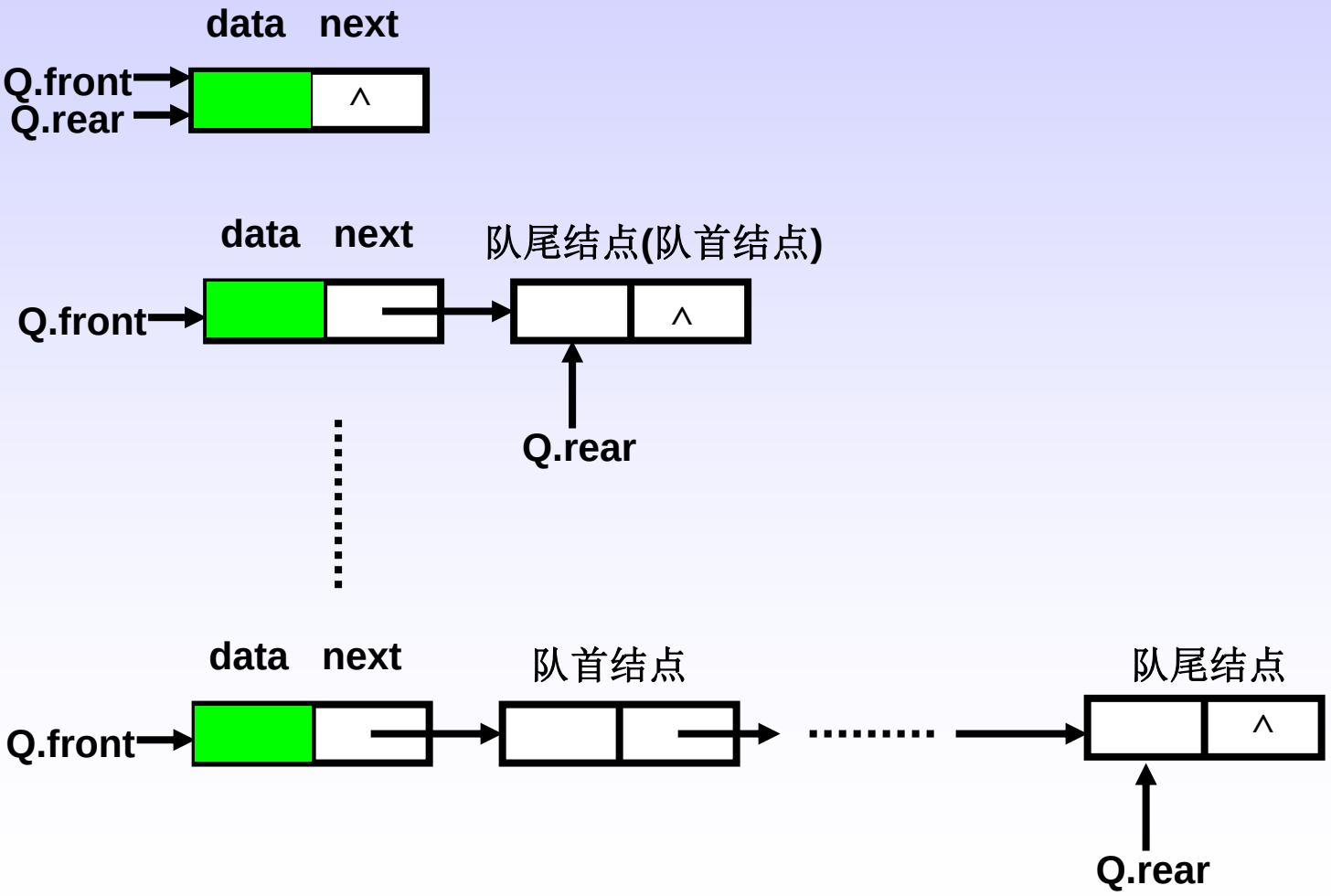
```
typedef struct Qnode {
        QElemType       data;
        struct   Qnode  *next;
} Qnode;
typedef struct  {
        Qnode   *front;
        Qnode   *rear;
} LinkQueue;
```



Q.front 和 Q.rear 分别是队首和队尾指针。它们指示着队首的前一结点和队尾结点。

3.4 队列

链接队列的操作：



3.4 队列

```
Status EnQueue (LinkQueue &Q,  QElemType  e )
{   p = ( Qnode * ) malloc ( sizeof (Qnode));
    if( p==NULL )
        exit( OVERFLOW );
    p->data = e;
    p->next = NULL;
    Q.rear->next = p;
    Q.rear = p;
    return OK;
 }  // EnQueue ;
```

3.4 队列

```
Status DeQueue (LinkQueue &Q,  QElemType  &e )
{ if ( Q.rear == Q.front)
      return( ERROR ) ;
  p = Q.front->next;
  e = p->data;
  Q.front->next = p->next;
  if( Q.rear == p )          // 出队前队列中只有一个元素时，
     Q.rear = Q.front ;      // 需修改队尾指针，指向头结点。
  free( p );
  return OK;
} // DeQueue ;
```