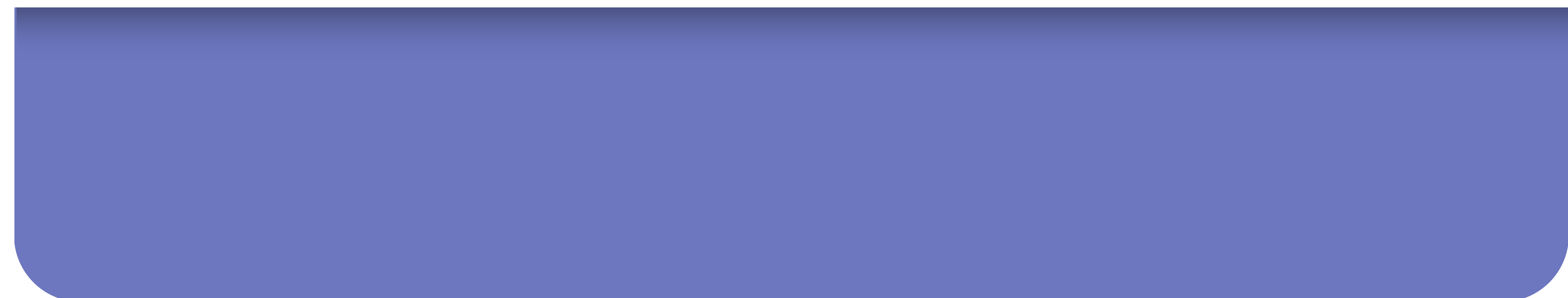
第3章　栈和队列



栈和队列是两种重要的数据结构。

栈和队列也是线性表，是操作受限的线性表，又称为限定性的数据结构。

3.1　栈

栈的抽象数据类型定义:

栈，是限定仅在表尾进行插入或删除操作的线性表。表尾端称为栈顶(top)，表头端称为栈底(bottom)。

不含元素的空表称为空栈。

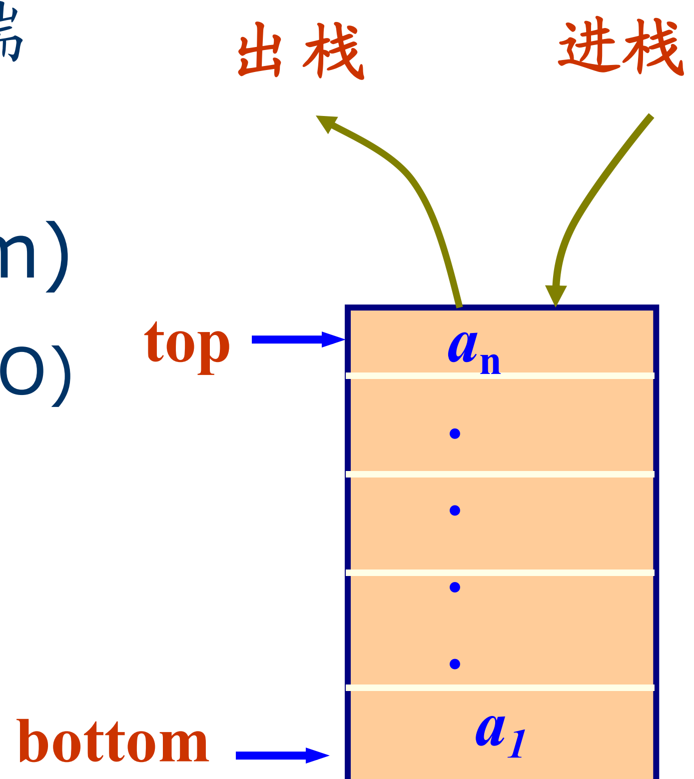
栈又称为LIFO(后进先出）的线性表。

栈（Stack）

- 允许插入和删除的一端称为栈顶(top),

另一端称为栈底(bottom)

- 特点： 后进先出 (LIFO)



栈的基本操作包括：

InitStack(&S);
DestroyStack(&S);
ClearStack(&S);
StackEmpty(S);
StackLength(S);
GetTop(S,&e);
Push(&S,e);
Pop(&S,&e);
StackTraverse(S,visit());

栈的表示和实现

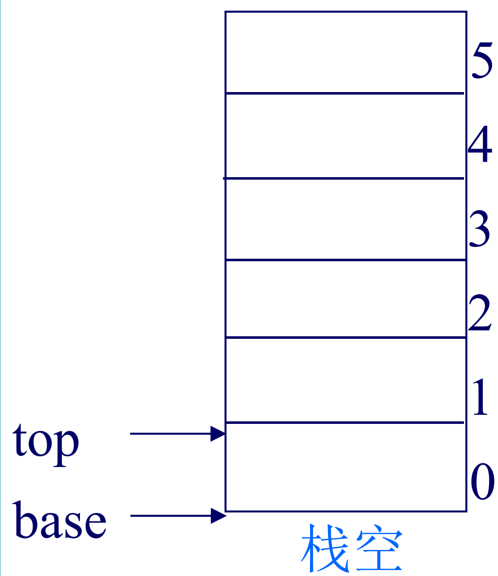
- 顺序栈
- 链栈

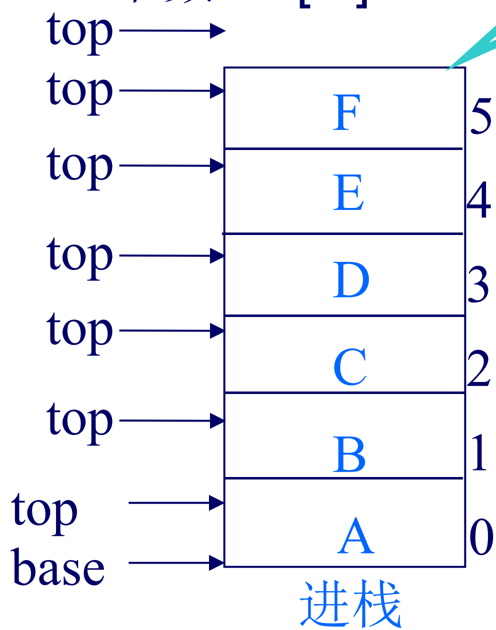
顺序栈，利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。 通常情况下，以**top=0**表示空栈。但是鉴于**C**语言数组的下标约定从**0**开始，如此设定会带来很大不便。因此，在用**C**语言数组实现栈时，我们采用不一样的方式表示空栈。

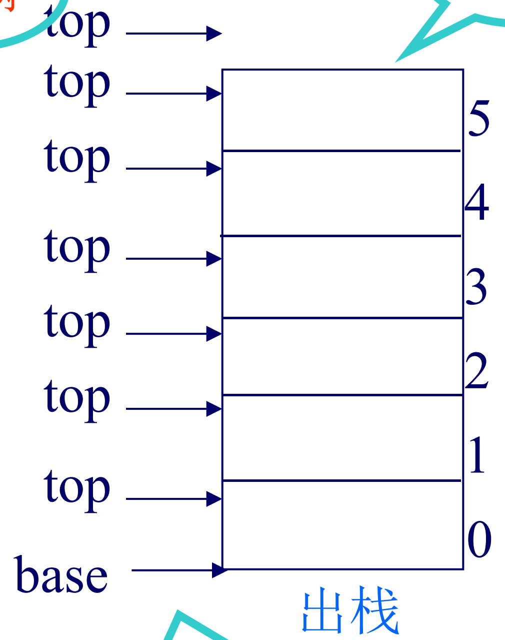
顺序栈的定义：

```
typedef struct{
    SElemType *base;  ---栈底指针，始终指向栈底的位置
    SElemType *top; ---栈顶指针，始终指向栈顶元素下一个位置
    int          stacksize; ---指示栈的当前可使用的最大容量
}SqStack;
```

- 栈的存储结构
  - 顺序栈
    - 实现：一维数组s[M]







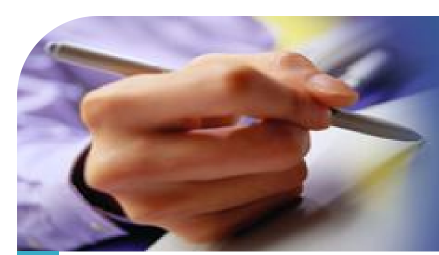
当top=bottom，栈空

设数组维数为M
top=base,栈空，此时出栈，则下溢（underflow)
top=base +M,栈满，此时入栈，则上溢（overflow)

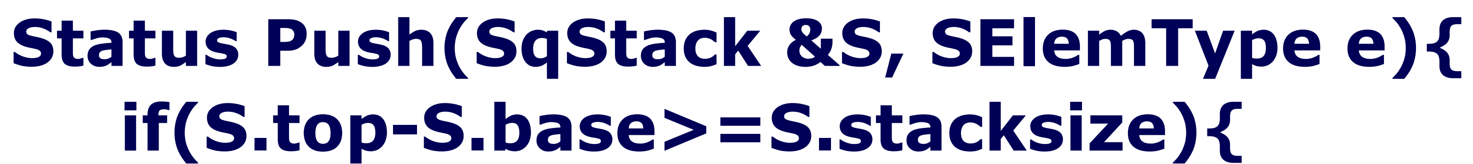
基本操作的算法描述：

```
Status InitStack(SqStack  &S){

   S.base=(SElemType*)malloc(STACK_INIT_SIZE*sizeof(SElemType));
     if(!S.base)  exit (OVERFLOW);
     S.top=S.base;
     S.stacksize=STACK_INIT_SIZE;
     return OK;
}
```

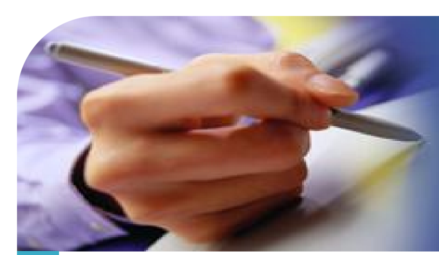


```
Status  GetTop(SqStack  S, SElemType
  &e){
     if(S.top==S.base) return ERROR;
     e=*(S.top-1);
     return  OK;
}
```



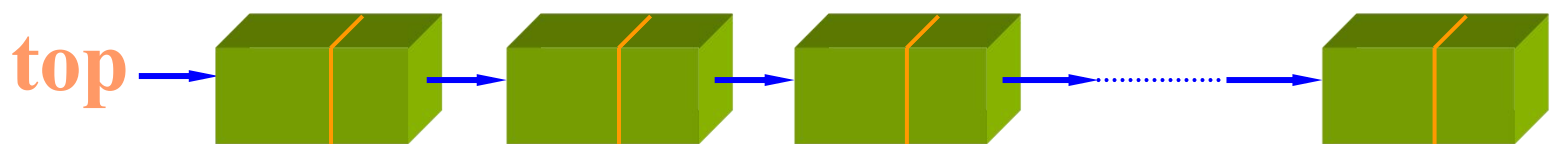
```
Status Push(SqStack &S, SElemType e){
    if(S.top-S.base>=S.stacksize){

   S.base = (SElemType
   *)realloc(S.base,(S.stacksize+STACKINCRE
   MENT)*sizeof(SElemType));
      if(!S.base) exit(OVERFLOW);
      S.top=S.base+S.stacksize;
      S.stacksize+=STACKINCREMENT;
   }
   *S.top++=e;return OK;
}
```



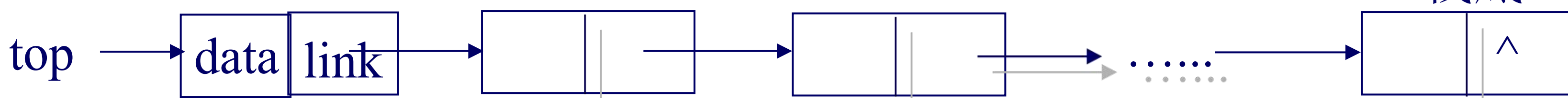
```
Status Pop(SqStack &S, SElemType
  &e){
      if(S.top==S.base)  return ERROR;
      e=*--S.top;
      return OK;
}
```

- 链式栈无栈满问题，空间可扩充
- 插入与删除仅在栈顶处执行
- 链式栈的栈顶在链头



• 链栈

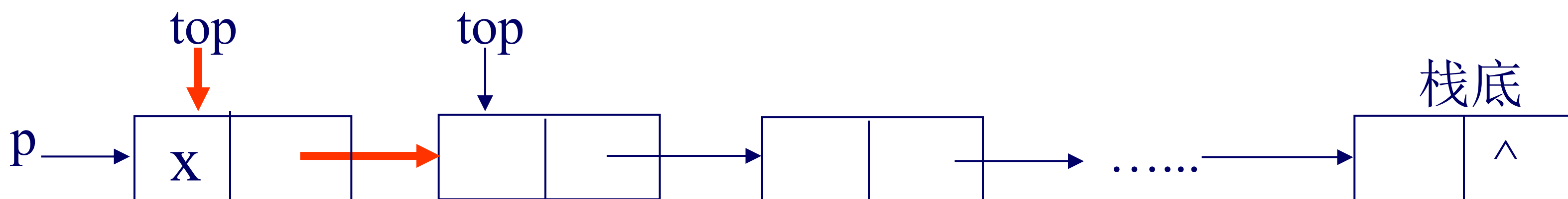
栈顶



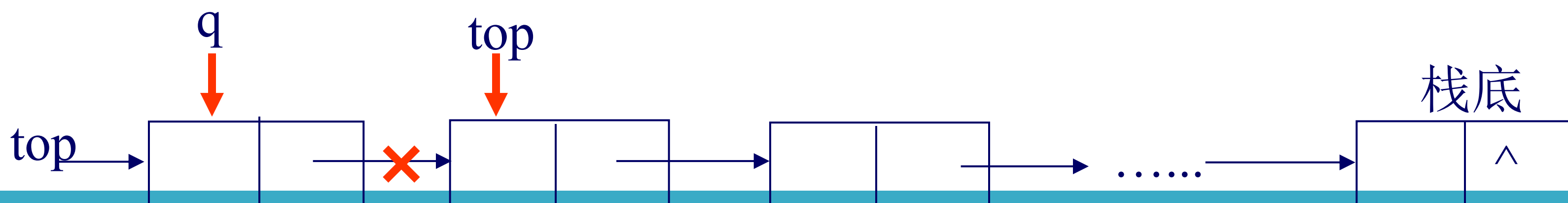
– 结点定义

```
typedef   struct node
{   int data;
    struct  node  *link;
}JD;
```

– 入栈算法



– 出栈算法



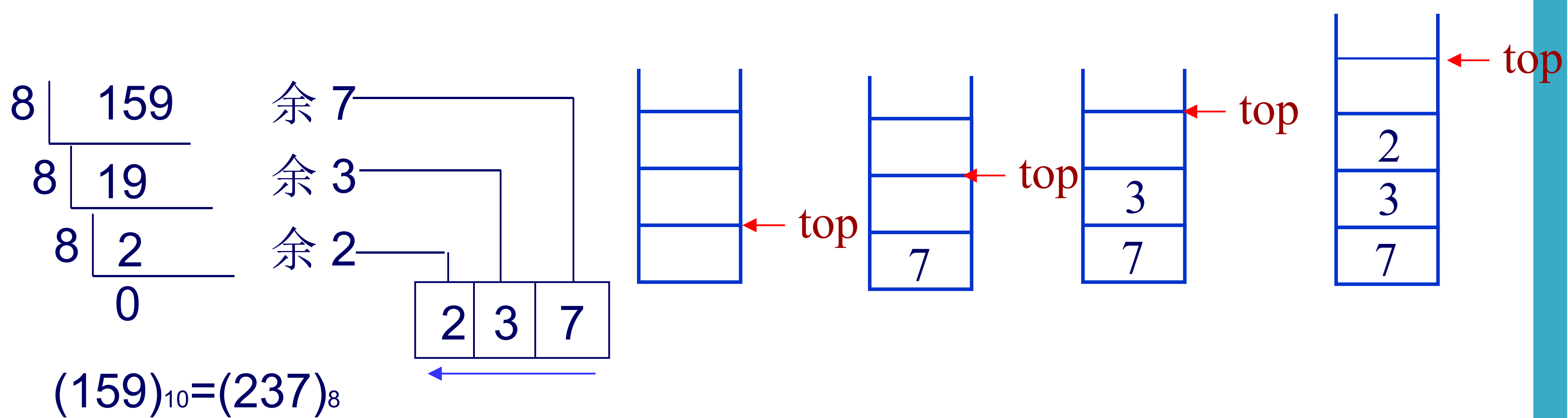
- 数制转换
- 括号匹配的检验
- 行编辑程序
- 迷宫求解
- 表达式求值

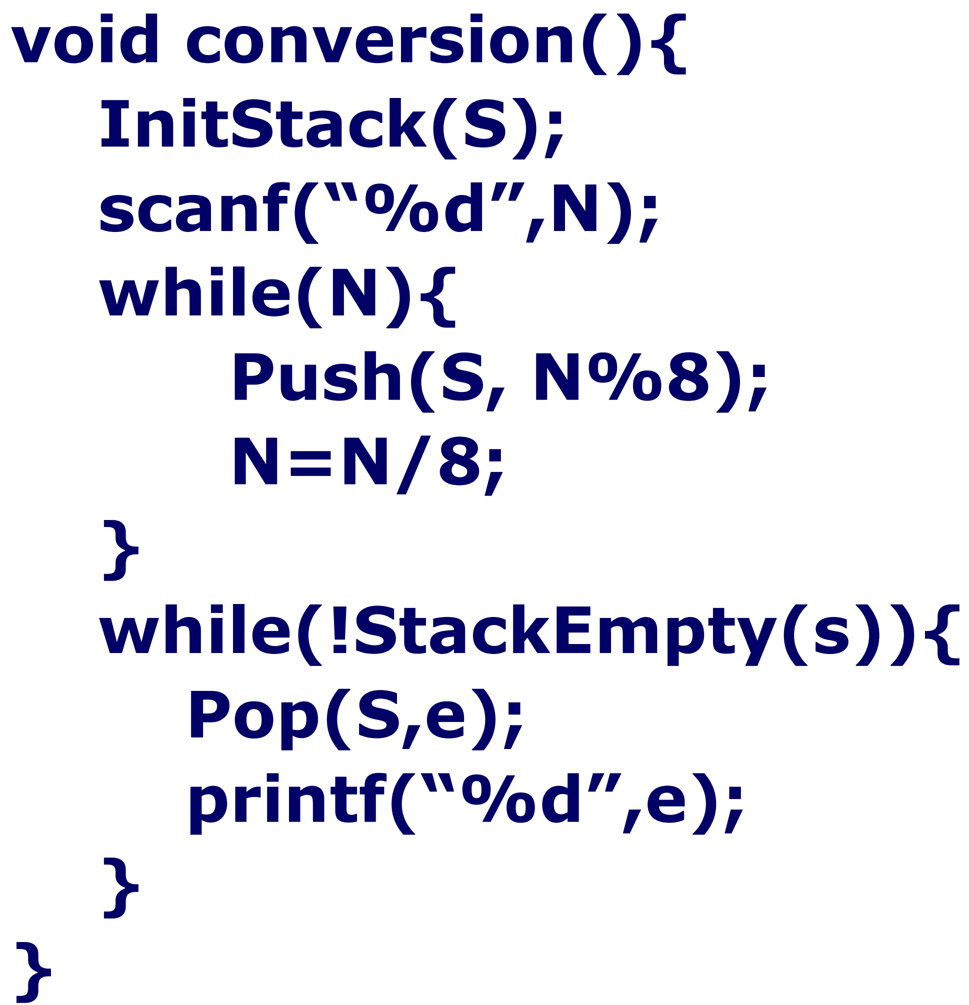
1、数制转换

十进制数**N**与其他**d**进制数的转换是计算机实现计算的基本问题，其中一个简单算法基于下列原理：

N=(N div d)*d + N mod d

例 把十进制数159转换成八进制数





```
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);
    }
}
```

2、括号匹配的检验

假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，即（[ ]（））或[（[ ][ ]）]等为正确的格式，[（]）或（[（））均为不正确的格式。

检测括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。

考虑下列括号序列

[（[ ][ ]）]

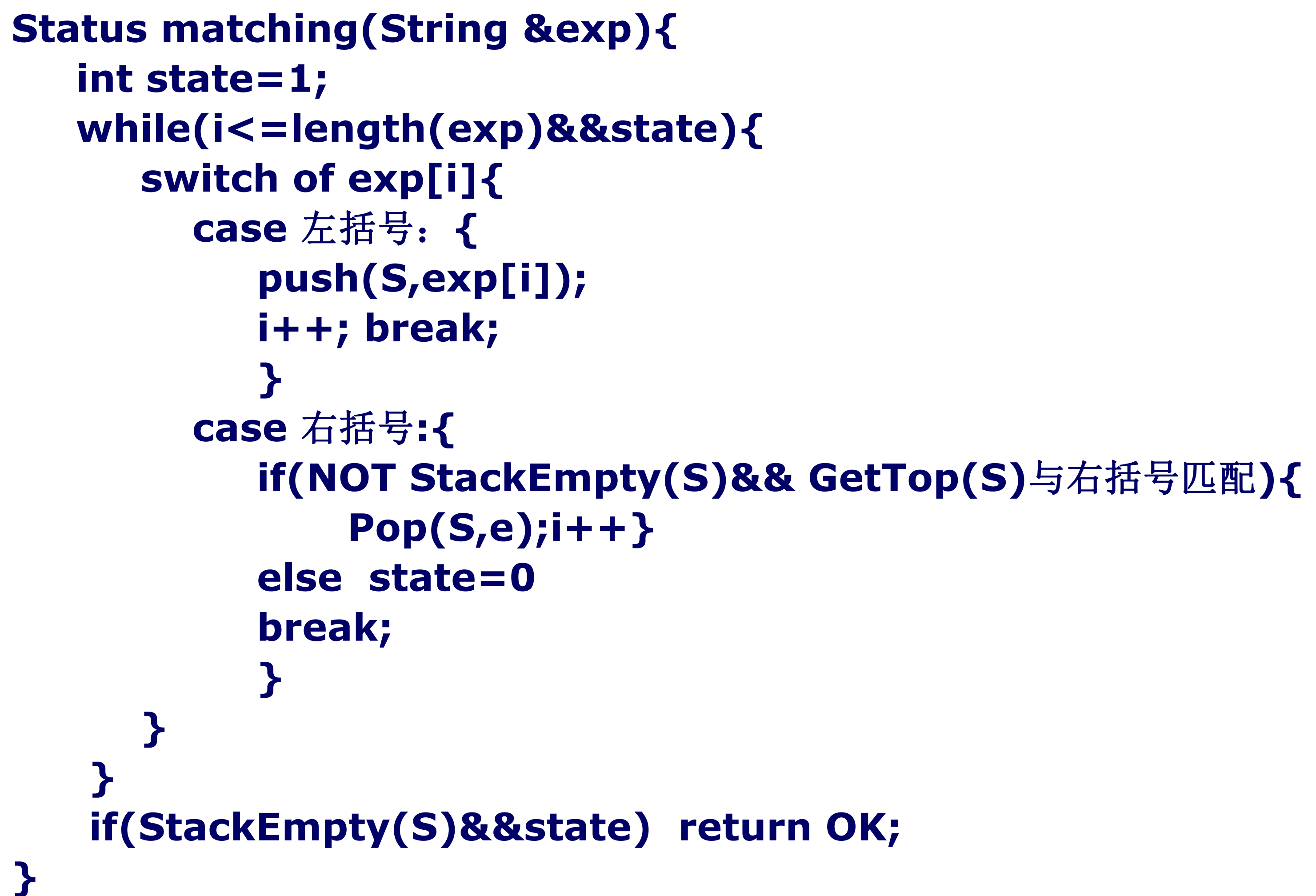
1 2 3 4 5 6 7 8

当计算机接受了第一个括号后，它期待着与它匹配的第**8**个括号的出现，然而等来的却是第**2**个括号，此时第**1**个“**[**”只能暂时靠边，而迫切等待与第**2**个括号相匹配的第**7**个“）”的出现，类似的，因等来的是第**3**个“**[**”，其期待匹配的程度较第**2**个“（”迫切，则第**2**个括号也只能暂时靠边，让位于第**3**个符号；在接受了第**4**个括号之后，第 **3**个括号的期待得到满足，消解之后，第**2**个括号的期待匹配就成为当前最紧迫的任务，依此类推。

- 由此，在算法中设置一个栈，每读入一个括号，若是右括号，则或者使置于栈顶的最急迫的期待得以消解，或者是不合法的情况；若是左括号，则作为一个新的更急迫的期待压入栈中，自然使原有的在栈中的所有未消解的期待的急迫性都降了一级。
- 另外在算法的开始和结束时，栈都应该是空的。
- 算法的设计思想：
- **1**、凡出现左括号，则进栈；
- **2**、凡出现右括号，首先检查栈是否为空。若栈空，则表明右括号多了，否则和栈顶元素比较，若匹配，则左括号出栈，否则不匹配。
- **3**、表达式检验结束时，若栈空，则匹配结束，否则表明左括号多了。

[([ ] [ ])]

[
[(
[([
[(
[([
[(
[



Status matching(String &exp){
int state=1;
while(i<=length(exp)&&state){
switch of exp[i]{
case 左括号：{
push(S,exp[i]);
i++; break;
}
case 右括号:{
if(NOT StackEmpty(S)&& GetTop(S)与右括号匹配){
Pop(S,e);i++}
else state=0
break;
}
}
}
if(StackEmpty(S)&&state) return OK;
}

3、行编辑程序

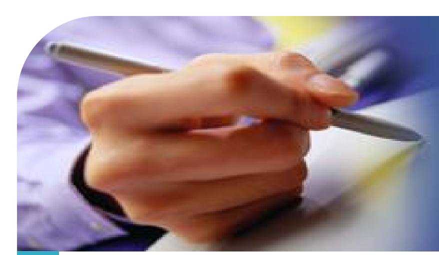
设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区。允许用户输入出差错，并在发现有误时可以及时更正。

例如，当用户发现刚输入的一个字符有错时，可补进一个退格符“#”，以表示前一个字符无效；如果发现当前输入的行内差错较多或难以补救，可以键入一个退行符“@”，以表示当前行中的字符均无效。

```
Whil##ilr#e(s#* s)
  outcha@putchar(*s=#++);
```

则实际有效的输入为

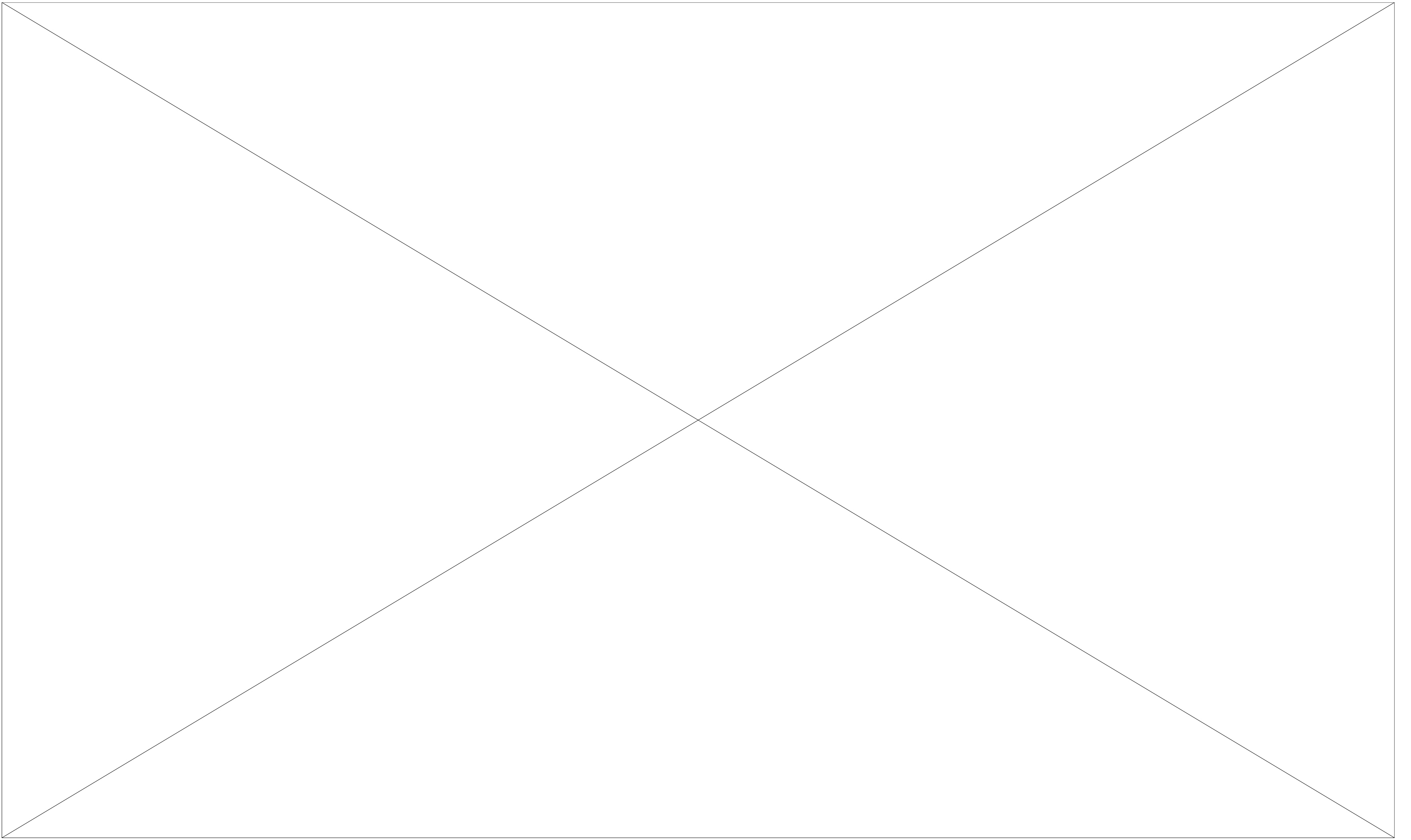
```
While (*s)
  putchar (*s++);
```



- 为此，可设该缓冲区为一个栈结构，每当从终端接受了一个字符以后先作如下判别：如果它既不是退格符也不是退行符，则将该字符压入栈顶；如果是退格符，则从栈顶删去一个字符；如果它是一个退行符，则将字符栈清为空栈。

```
Void LineEdit(){
        InitStack(s);
        ch=getchar();
        while (ch != EOF) { //EOF为全文结束符
                while (ch != EOF && ch != '\n') {
                    switch (ch) {
                        case '#' : Pop(S, c); break;
                        case '@': ClearStack(S); break;
                                //重置S为空栈
                        default : Push(S, ch);  break;
                        }
                ch = getchar(); // 从终端接收下一个字符
                }
                将从栈底到栈顶的栈内 字符传送至调用过程的缓冲区
                ClearStack(S);  //重置S为空栈
                if(ch != EOF) ch = getchar();
          }
          DestroyStack(S)
}
```

迷宫求解

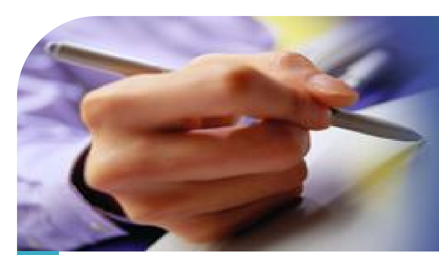


❖迷宫路径算法的基本思想

❖假设“当前位置”指的是“在搜索过程中某一时刻所在图中某个方块的位置”，则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“下一个位置”探索，切换“下一个位置”为当前位置，如此重复直至到达出口；
- 若当前位置“不可通”，则应顺着“来向”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；
- 若该通道块四周4个方块均“不可通”，则应从“当前路径”上删除该通道快。

```
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;
               若该位置是出口位置，则算法结束;
               否则切换当前位置的东邻方块为新的当前位置;
       }
   否则 {
           ………..
   }
} while (栈不空）;
```



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{

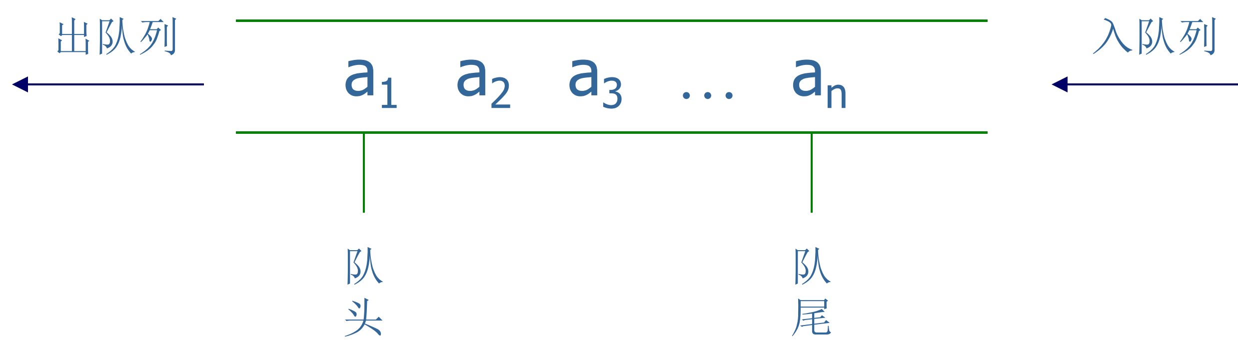
删去栈顶位置;

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;

}

队列

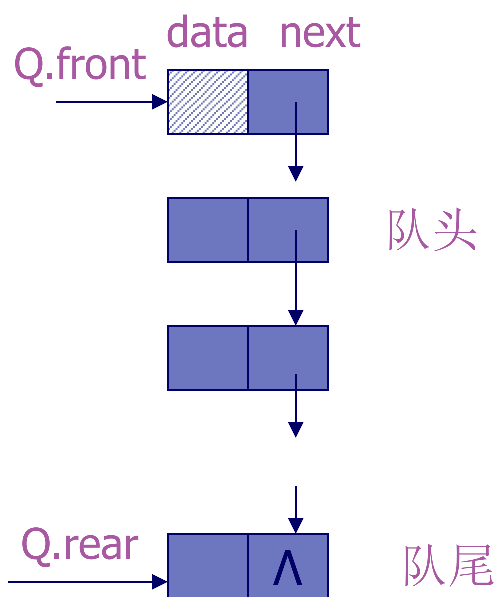
- 和栈相反，队列（**Queue**）是一种先进先出**FIFO**的线性表，它只允许在表的一端进行插入，而在另一端删除元素。
- 在队列中允许插入的一端叫做队尾**rear**，允许删除的一端则称为队头**front**。

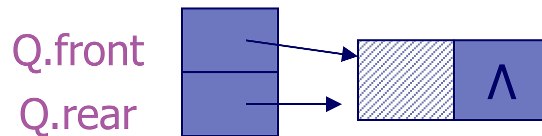


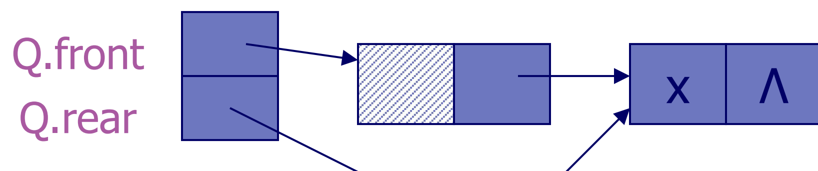
队列的基本操作有：

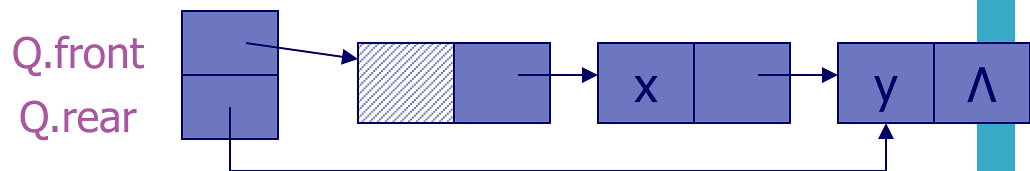
```
InitQueue(&Q);
DestroyQueue(&Q);
ClearQueue(&Q);
QueueEmpty(Q);
QueueLength(Q);
GetHead(Q,&e);
EnQueue(&Q,e);
DeQueue(&Q,&e);
```

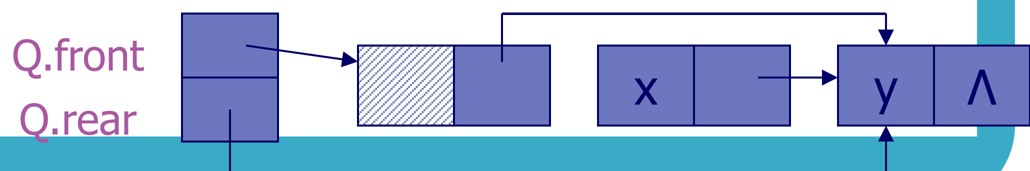
- 链队列----队列的链式表示和实现
- 用链表表示的队列简称为链队列。









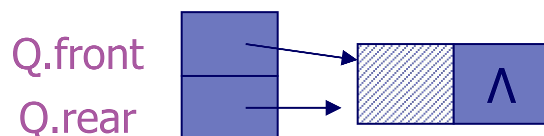


```
typedef struct QNode{
      QElemType      data;
      struct QNode    *next;
}QNode,*QueuePtr;
typedef struct{
      QueuePtr  front;
      QueuePtr  rear;
}LinkQueue;
```

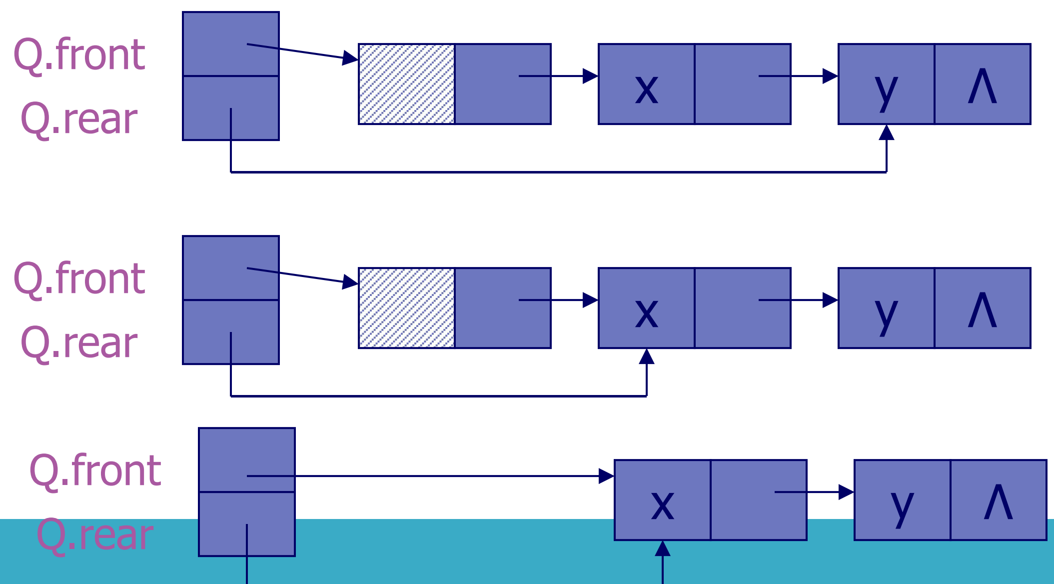
队列的基本操作的算法描述：

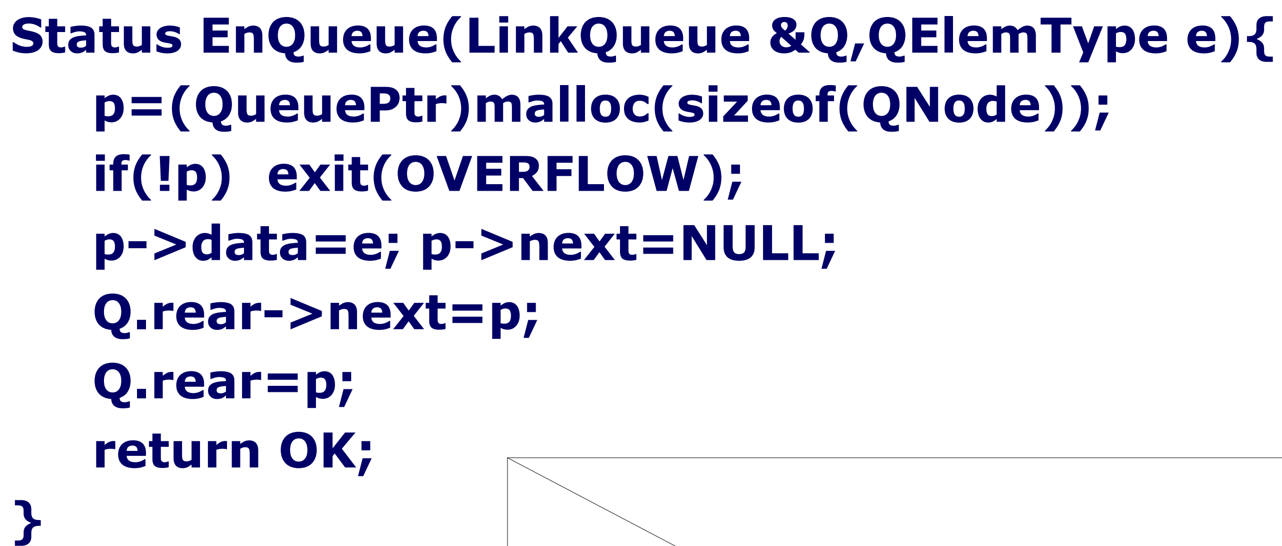
```
Status InitQueue(LinkQueue &Q){

   Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
    if(!Q.front)   exit(OVERFLOW);
    Q.front->next=NULL;
    return OK;
}//构造一个空队列
```

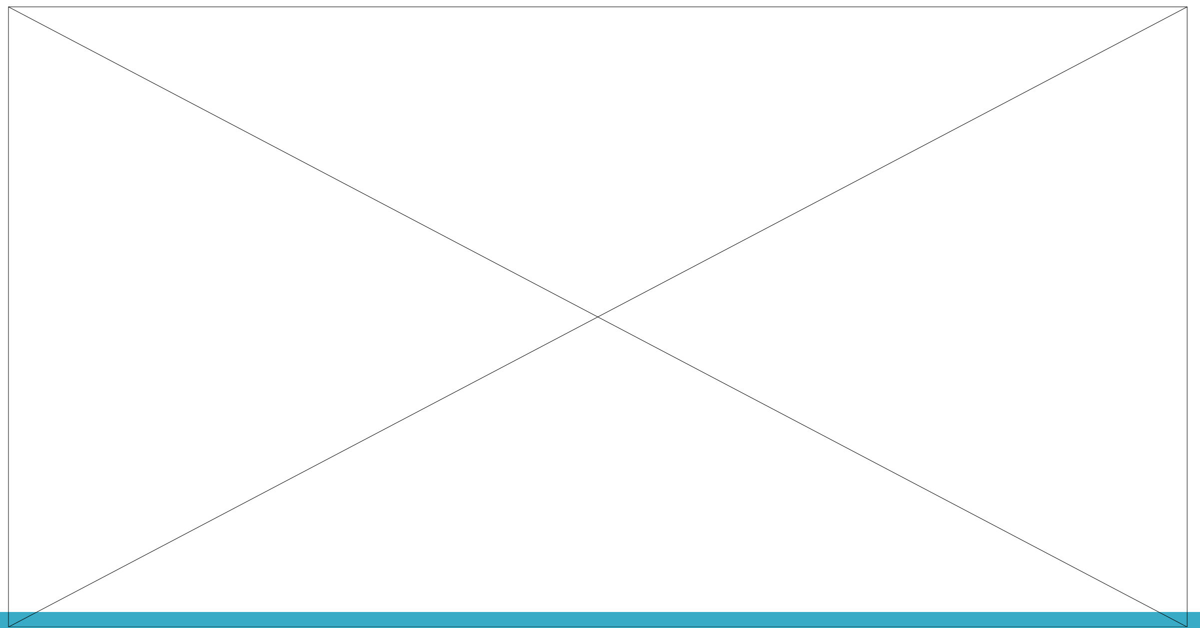


```
Status DestroyQueue(LinkQueue &Q){
    while (Q.front){
        Q.rear=Q.front->next;
        free(Q.front);
        Q.front=Q.rear;
    }
    return OK;
}
```

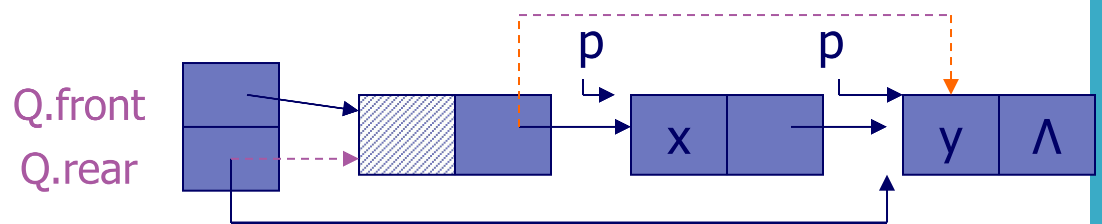




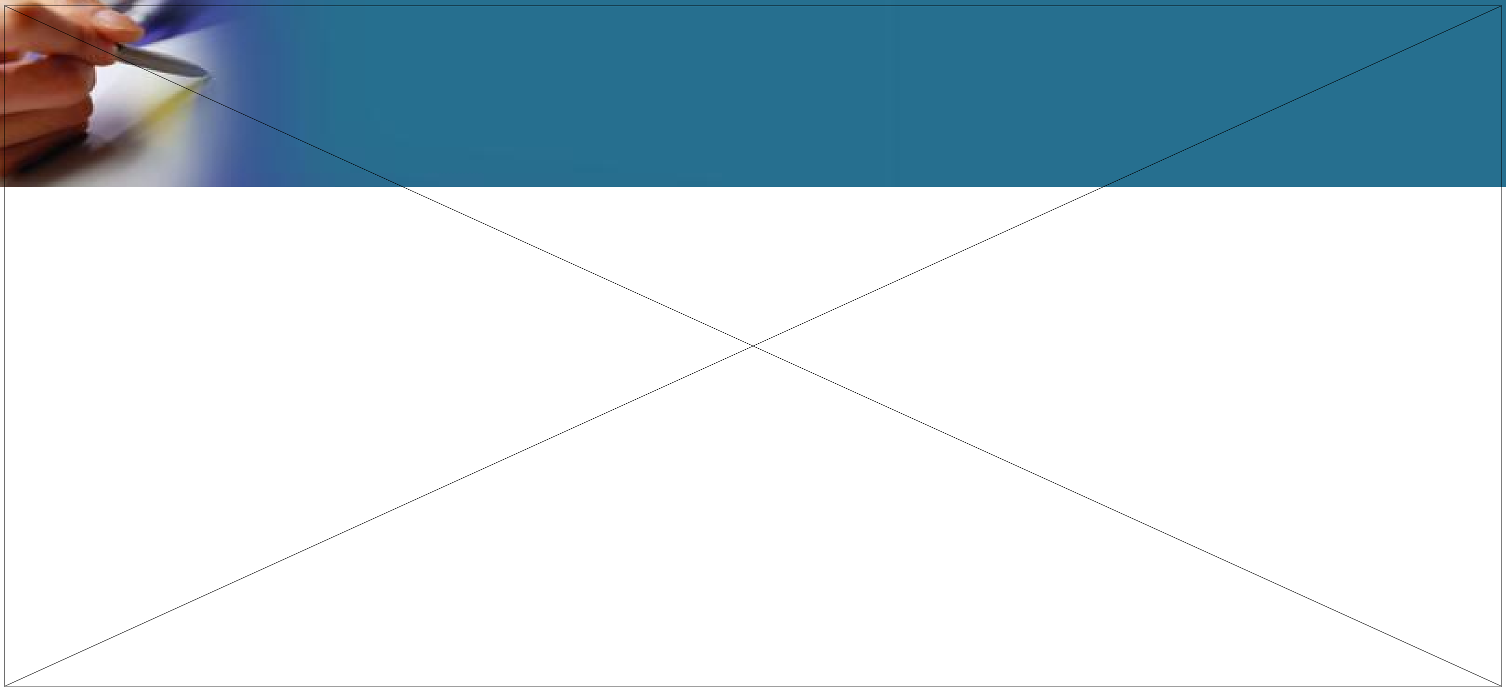
```
Status EnQueue(LinkQueue &Q,QElemType e){
    p=(QueuePtr)malloc(sizeof(QNode));
    if(!p)  exit(OVERFLOW);
    p->data=e; p->next=NULL;
    Q.rear->next=p;
    Q.rear=p;
    return OK;
}
```

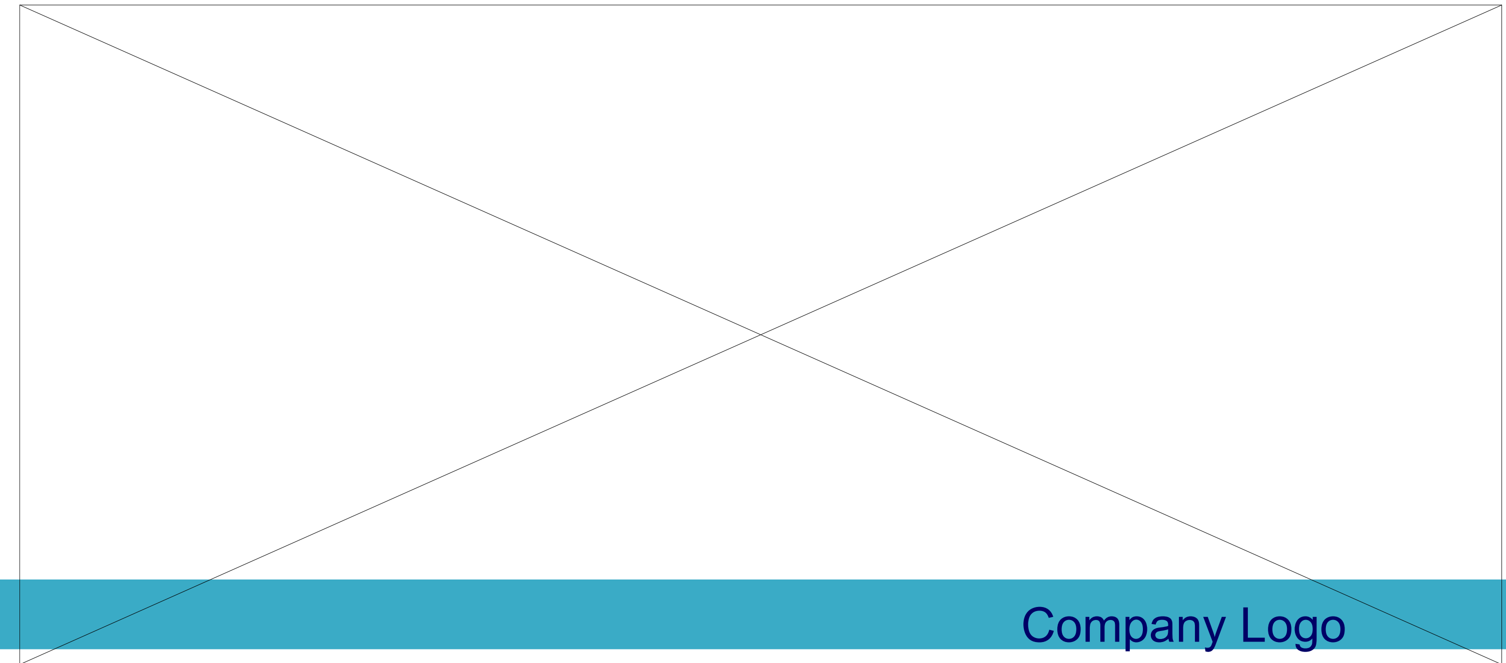


```
Status DeQueue(LinkQueue &Q,QElemType &e){
    if(Q.front == Q.rear) return ERROR;
    p=Q.front->next;
    e=p->data;
    Q.front->next=p->next;
    if(Q.rear == p)  Q.rear = Q.front;
    free(p);
    return OK;
}
```



- Q.front->next=p->next=NULL;

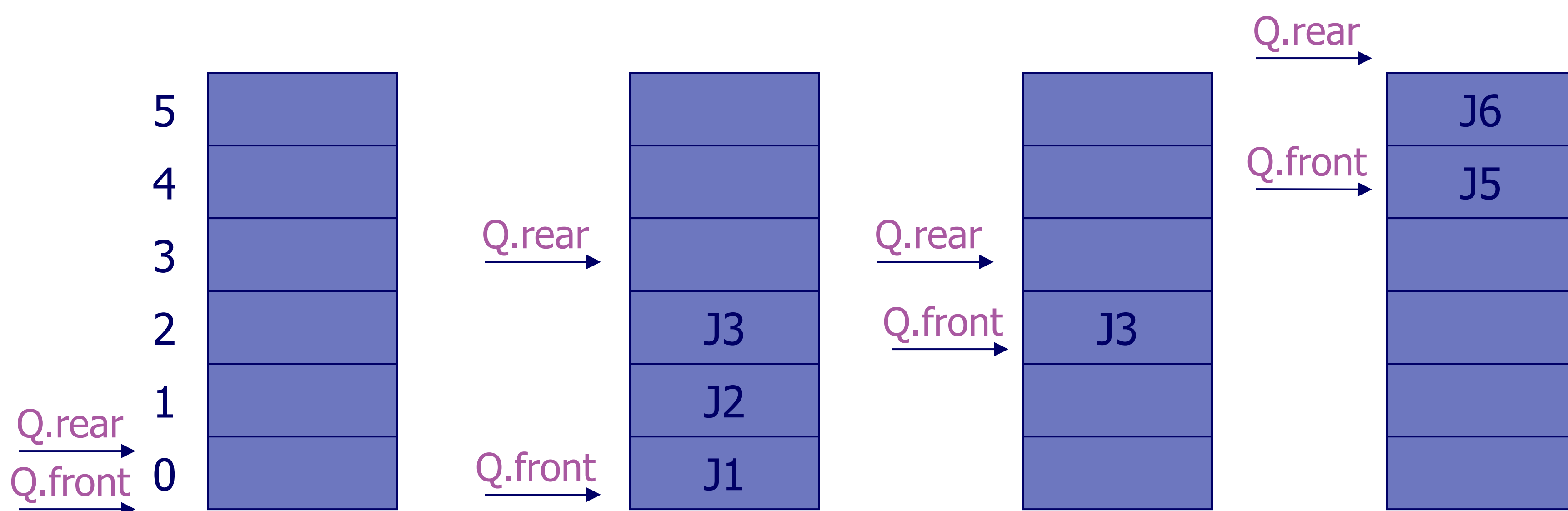


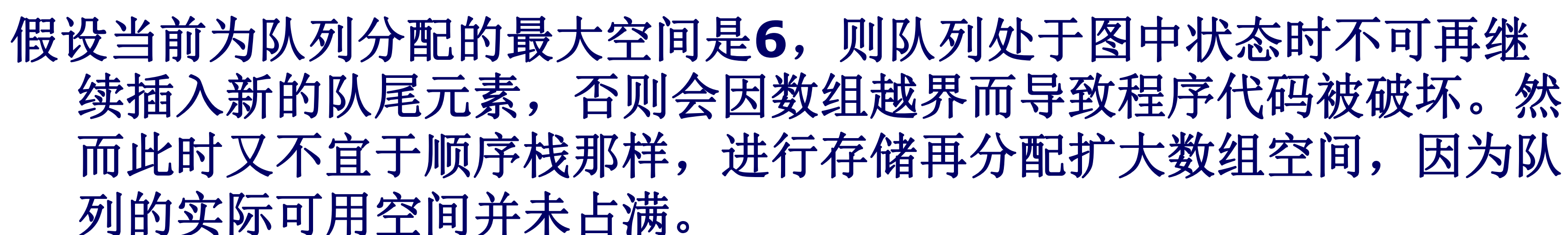


Company Logo

队列的顺序表示和实现

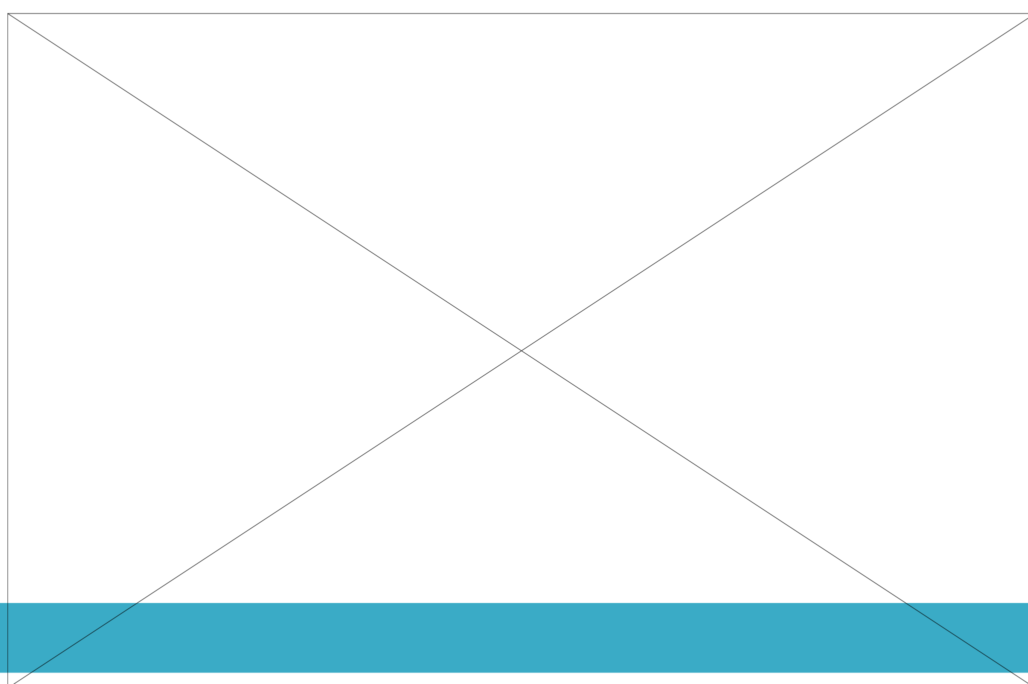
- 和顺序栈相类似，在队列的顺序存储结构中，除了用一组地址连续的存储单元依次存放从队列头到队列尾的元素之外，还需附设两个指针**front**和**rear**分别指示队列头元素和队列尾元素的位置。
- 在非空队列中，头指针始终指向队列头元素，而尾指针始终指向队列尾元素的下一个位置。

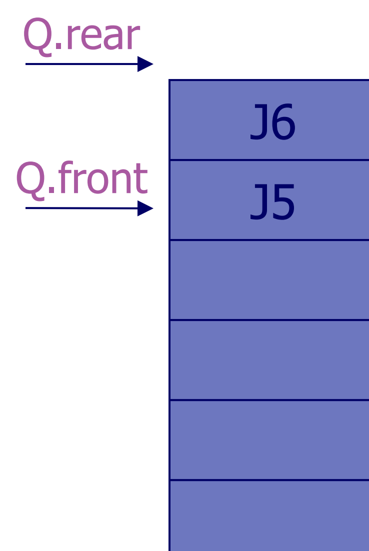


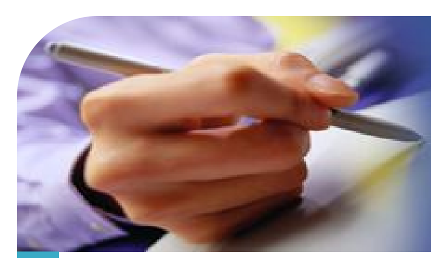


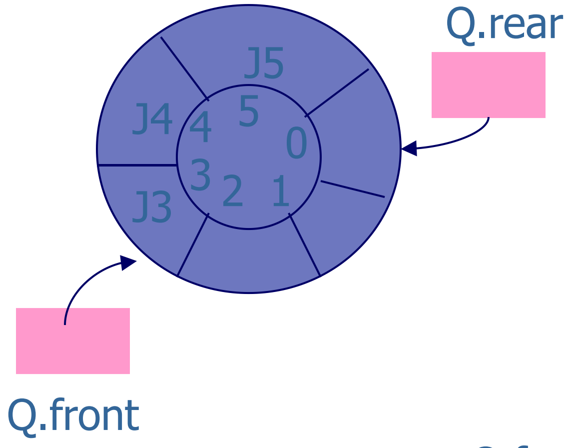
假设当前为队列分配的最大空间是**6**，则队列处于图中状态时不可再继续插入新的队尾元素，否则会因数组越界而导致程序代码被破坏。然而此时又不宜于顺序栈那样，进行存储再分配扩大数组空间，因为队列的实际可用空间并未占满。

一个巧妙的办法是将顺序队列想象为一个环状的空间，称为循环队列。

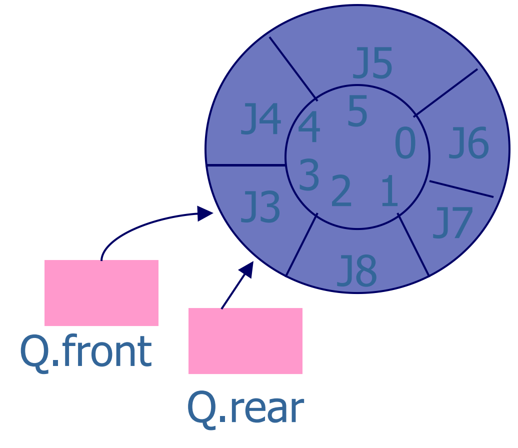




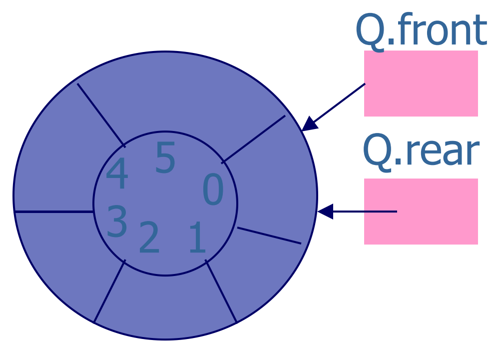




Q.rear
J5
J4
5
4
0
3
2
1
J3
Q.front

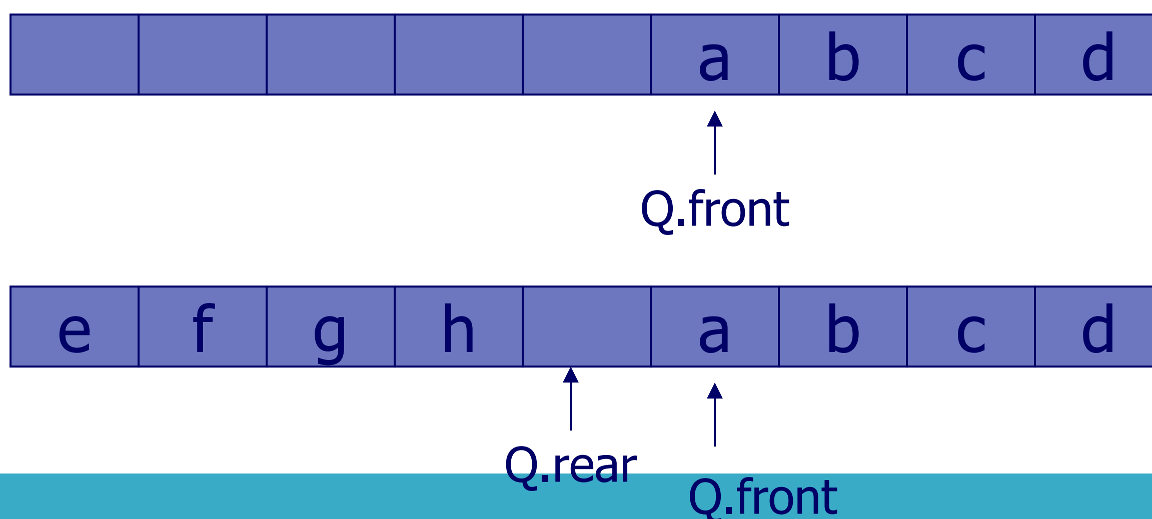


J5
J4
5
4
0
J6
3
2
1
J3
J7
J8
Q.front
Q.rear



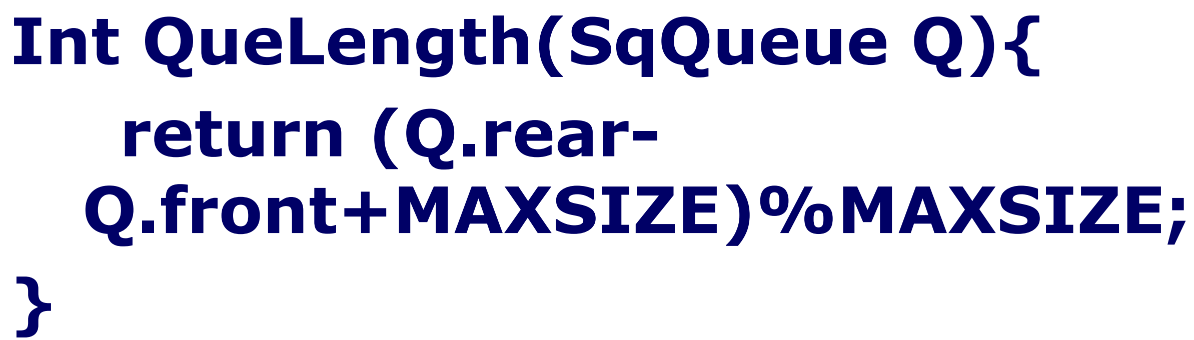
Q.front
Q.rear
5
4
0
3
2
1

- 由上图可以看出，当队列满时或队列空时，均有**Q.front=Q.rear**成立。因此，仅凭等式**Q.front=Q.rear**无法判别队列空间是空还是满，可有两种处理方法：
- **1**、增加一个标记位，区别队列是“空”还是“满”；
- **2**、少用一个元素空间，约定以“队列头指针在队列尾指针的下一位置上”作为队列呈“满”状态的标志。



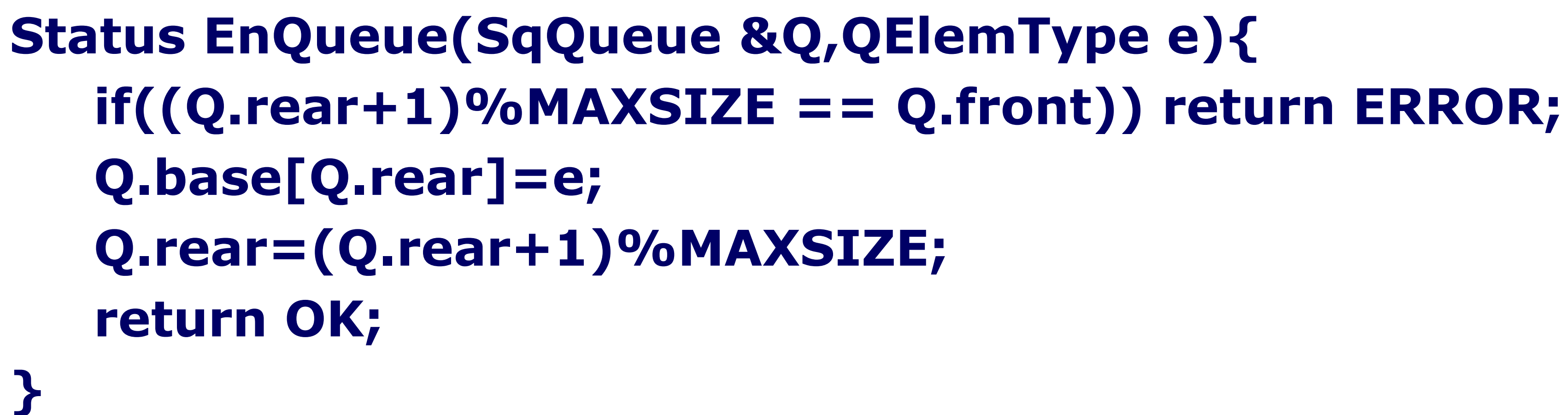
```
#define MAXQSIZE  100
typedef struct {
    QElemType  *base;
    int front;
    int rear;
}SqQueue;
```

```
Status InitQueue(SqQueue &Q){
    Q.base=(QElemType
  *)malloc(MAXSIZE*sizeof(QElemType));
    if(!Q.base)  exit(OVERFLOW);
    Q.front=Q.rear=0;
    return OK;
}//构造一个空队列Q
```

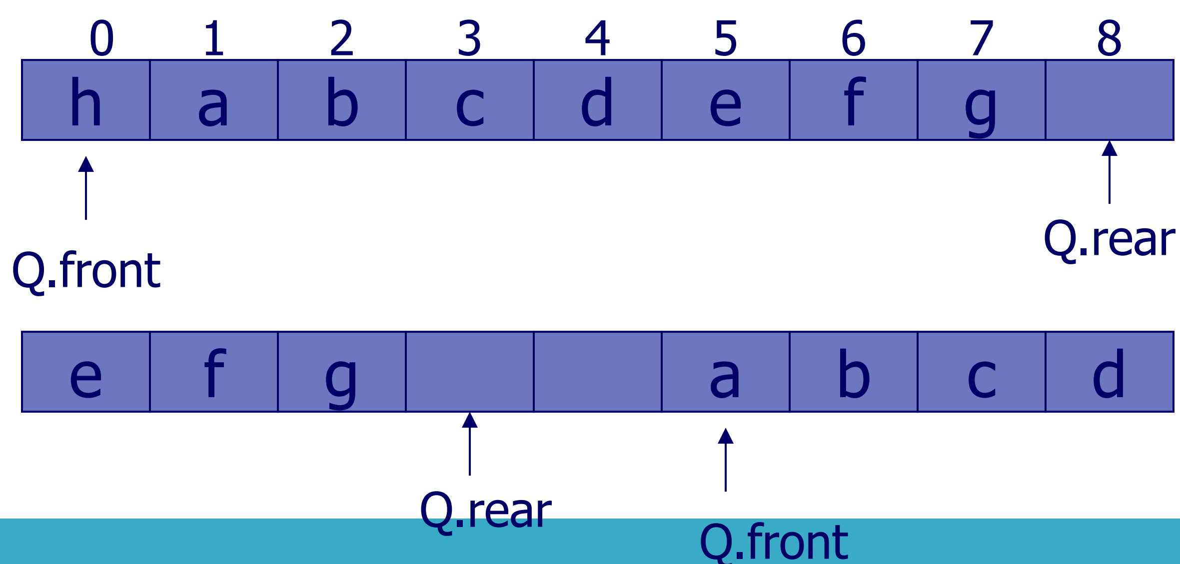


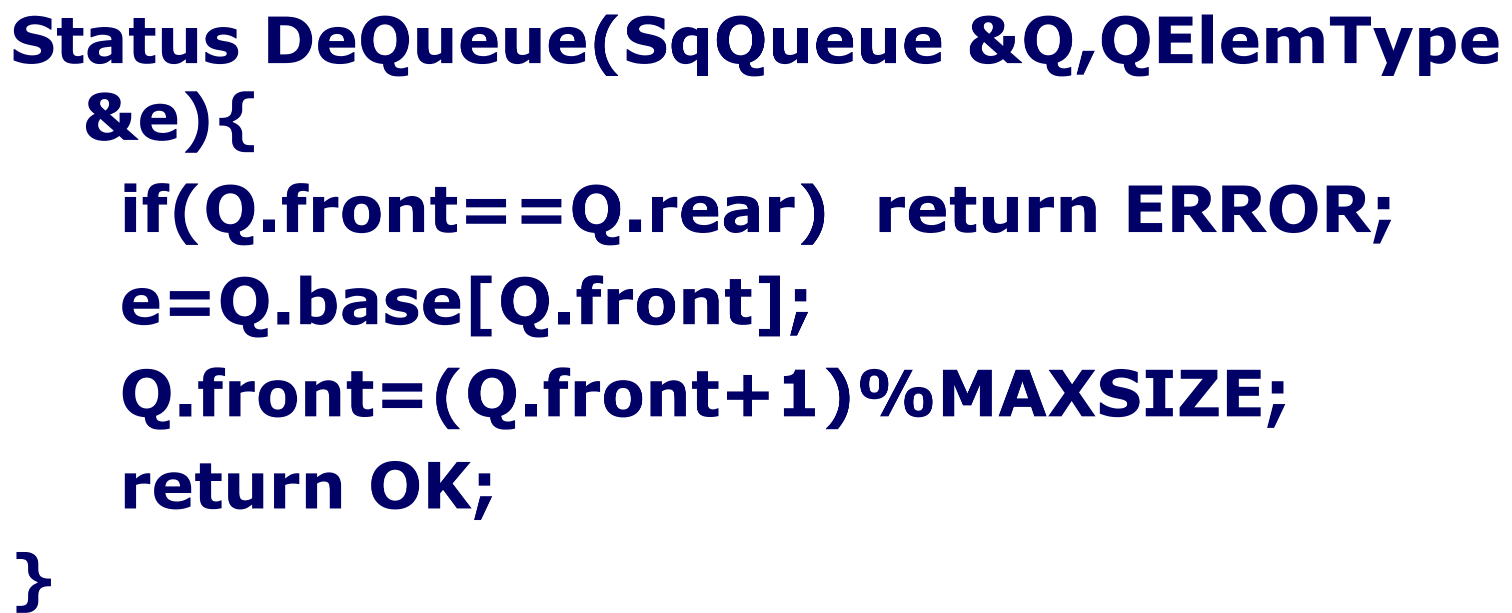
```
Int QueLength(SqQueue Q){
    return (Q.rear-
  Q.front+MAXSIZE)%MAXSIZE;
}
```

Q.rear=3, Q.front=5



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



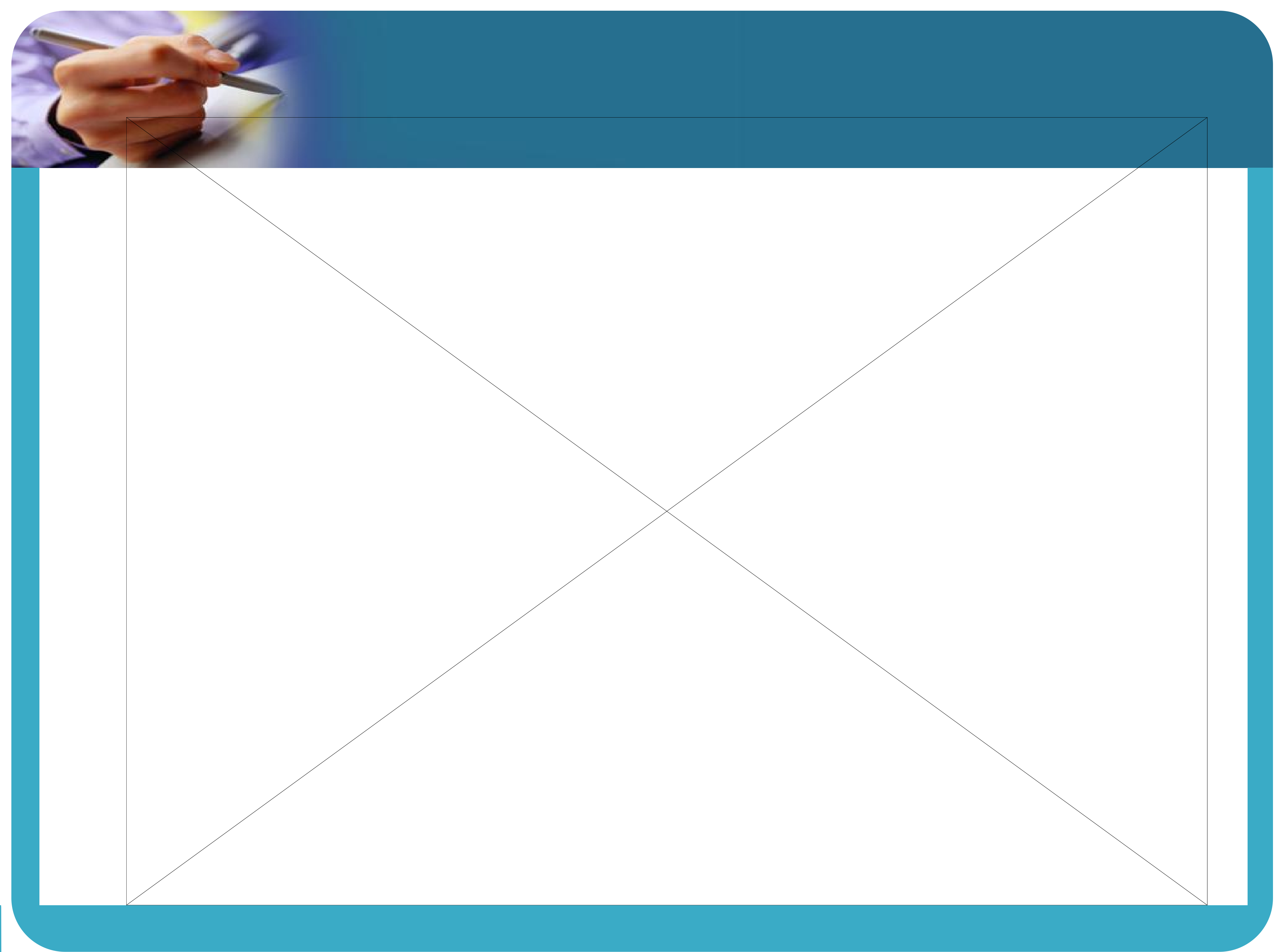


```
Status DeQueue(SqQueue &Q,QElemType
   &e){
    if(Q.front==Q.rear)  return ERROR;
    e=Q.base[Q.front];
    Q.front=(Q.front+1)%MAXSIZE;
    return OK;
}
```

排队问题的系统模拟

- 编制一个事件驱动仿真程序以模拟理发馆内一天的活动，要求输出在一天的营业时间内，到达的顾客人数、顾客在馆内的平均逗留时间和排队等候理发的平均人数以及在营业时间内空椅子的平均数。

为计算出每个顾客自进门到出门之间在理发馆内逗留的时间，只需要在顾客"进门"和"出门"这两个时刻进行模拟处理即可。习惯上称这两个时刻发生的事情为"事件"，整个仿真程序可以按事件发生的先后次序逐个处理事件，这种模拟的工作方式称为"事件驱动模拟"，程序将依事件发生时刻的顺序依次进行处理，整个仿真程序则以事件表为空而告终。



- "顾客进门"事件的处理：

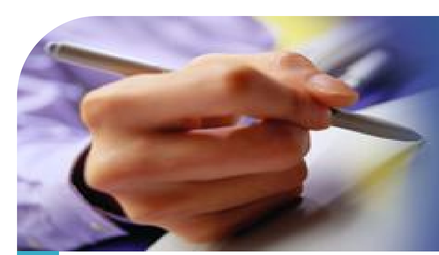
  1．生成"本顾客理发所需时间 **durtime**" 和 "下一顾客到达的时间间隔 **intertime**" 两个随机数；

  2．若下一顾客到达的时刻没有超过营业时间，则产生一个新的"进门事件"插入事件表；

  3．若此时理发馆内尚有空理发椅，则空椅子数减**1**且产生一个新的"出门事件"插入事件表，并累计顾客逗留时间；

  4．否则将该"顾客"插入"候理队列"；

  5．累计顾客人数和排队长度。

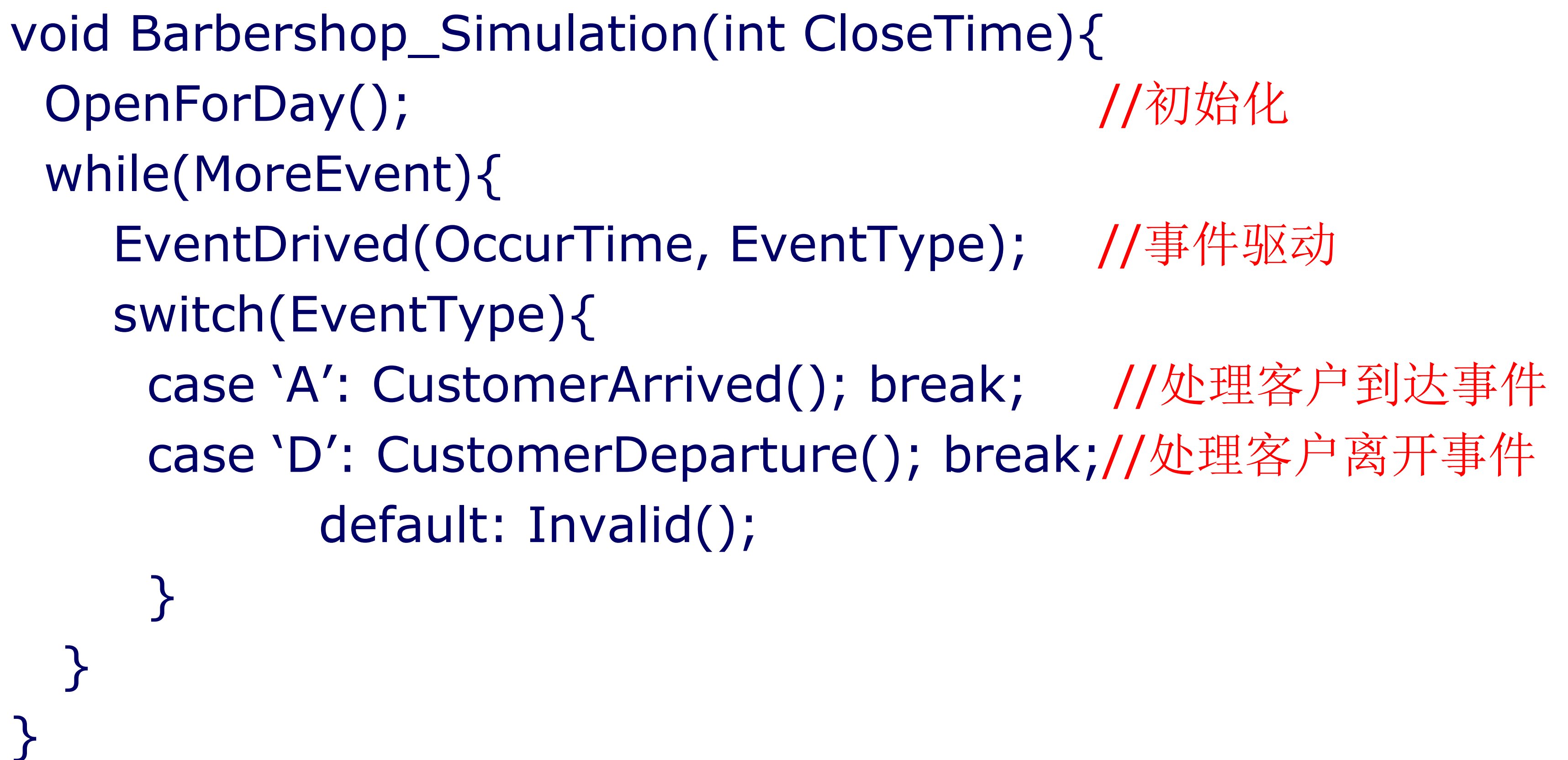


❖"顾客出门"事件的处理：

　　1．若候理队列为空，则空椅子数增**1**；

　　2．否则删除队头"顾客"，并产生一个新的"出门事件"插入事件表，且累计顾客逗留时间；

　　3．累计空椅子数。


void Barbershop_Simulation(int CloseTime){
OpenForDay(); //初始化
while(MoreEvent){
EventDrived(OccurTime, EventType); //事件驱动
switch(EventType){
case ‘A’: CustomerArrived(); break; //处理客户到达事件
case ‘D’: CustomerDeparture(); break;//处理客户离开事件
default: Invalid();
}
}
}

- **1. 用不带头结点的单链表存储队列时,其队头指针指向队头结点,其队尾指针指向队尾结点，则在进行删除操作时(　　)。**

  A. 仅修改队头指针　　B. 仅修改队尾指针

  C. 队头、队尾指针都要修改　D. 队头、队尾指针都可能要修改

- **2. 若一个栈的输入序列为1,2,3,…,n，输出序列的第一个元素是i，则第j个输出元素是（　　）。**

  A. i-j-1　　B. i-j

  C. j-i+1　　D. 不确定的

- **3.** 已知一个栈**s**的输入序列为**abcd**，下面两个序列能否通过栈的**push**和**pop**操作输出？如果能，请写出操作序列；如果不能，请说明原因。

  1、**dbca**　**2**、**cbda**

- **4.** 设栈**s**和队列**Q**的初始状态为空，元素**a**、**b**、**c**、**d**、**e**、**f**依次通过栈**s**，一个元素出栈后即进入队列**Q**。若这**6**个元素出队列的顺序是**b**，**d**，**c**，**f**，**e**，**a**，则栈**s**的容量至少是（ ）。

作业

- **1**. 简述栈和线性表的差别。
- **2.** 简述队列和栈这两种数据类型的相同点和差异处。
- **3.** 写出下列程序段的输出结果（栈的元素类型 **SElemType** 为 **char**）。

```
void main( ){
   Stack S;
   char x, y;
   InitStack(S);
   x='c'; y='k';
   Push(S, x); Push(S, 'a'); Push(S, y);
   Pop(S, x); Push(S, 't'); Push(S, x);
   Pop(S, x); Push(S, 's');
   while (!StackEmpty(S)) { Pop(S, y);
printf(y); };
   printf(x);
}
```

❖ **4.** 简述以下算法的功能（栈和队列的元素类型均为 **int**）。

```
void algo3(Queue &Q)
{
  Stack S; int d;
  InitStack (S);
  while (!QueueEmpty(Q))
  {
    DeQueue(Q, d); Push(S, d);
  }
  while (!StackEmpty(S))
  {
    Pop(S, d); EnQueue(Q, d);
  }
}
```