第3章 栈和队列

3.1 栈

3.2 栈的应用举例

3.3 队列

本章重点难点

重点：(1)栈、队列的定义、特点、性质和应用；(2)ADT栈、ADT队列的设计和实现以及基本操作及相关算法。

难点：(1)循环队列中对边界条件的处理；(2)分析栈和队列在表达式求值、括号匹配、数制转换、迷宫求解中的应用实例，提高利用栈和队列解决实际问题的应用水平。

第3章 栈和队列

3.1 栈

3.1.1 抽象数据类型栈的定义

3.1.2 栈的表示和实现

3.1.1 抽象数据类型栈的定义

栈的定义

栈(Stack)是一种特殊的线性表，其插入和删除操作均在表的一端进行，是一种运算受限的线性表。

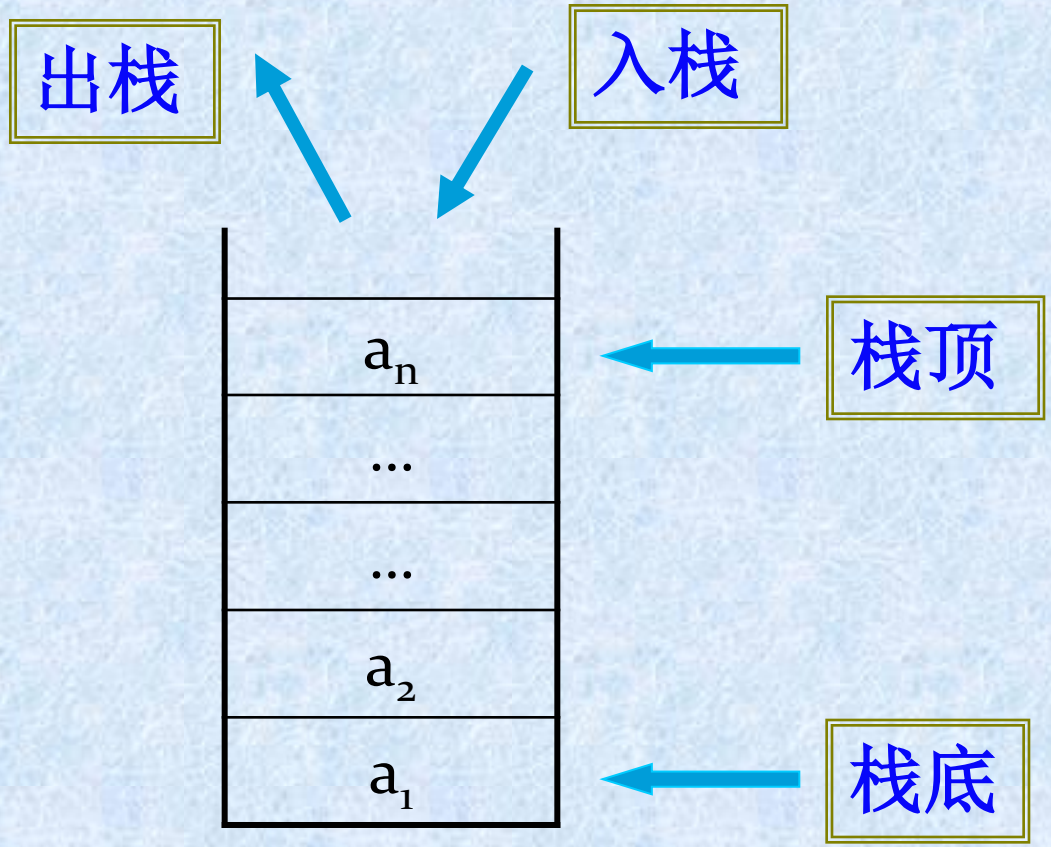
栈的术语

栈顶(top)是栈中允许插入和删除的一端。

栈底(bottom)是栈顶的另一端。

3.1.1 抽象数据类型栈的定义

- 栈的示意图



- 栈的特点

后进先出(Last In First Out，简称LIFO)。
又称栈为后进先出表(简称LIFO结构)。

3.1.1 抽象数据类型栈的定义

□抽象数据类型栈

ADT Stack {

数据对象：$D=\{ a_i \mid a_i \in ElemSet, i=1,2,...,n, n\geq 0 \}$

数据关系：$R1=\{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,...,n \}$

约定a_n 端为栈顶，a_1 端为栈底。

基本操作：见下页

} ADT Stack

3.1.1 抽象数据类型栈的定义

栈的基本操作

```
InitStack(&S)              //初始化栈
DestroyStack(&S)           //销毁栈
ClearStack(&S)             //清空栈
StackEmpty(S)              //判栈空
StackLength(S)             //求栈长度
GetTop(S, &e)              //取栈顶元素
Push(&S, e)                //入栈
Pop(&S, &e)                //出栈
StackTravers(S, visit())   //遍历栈
```

3.1 栈

3.1.1 抽象数据类型栈的定义

3.1.2 栈的表示和实现

顺序栈的C语言实现

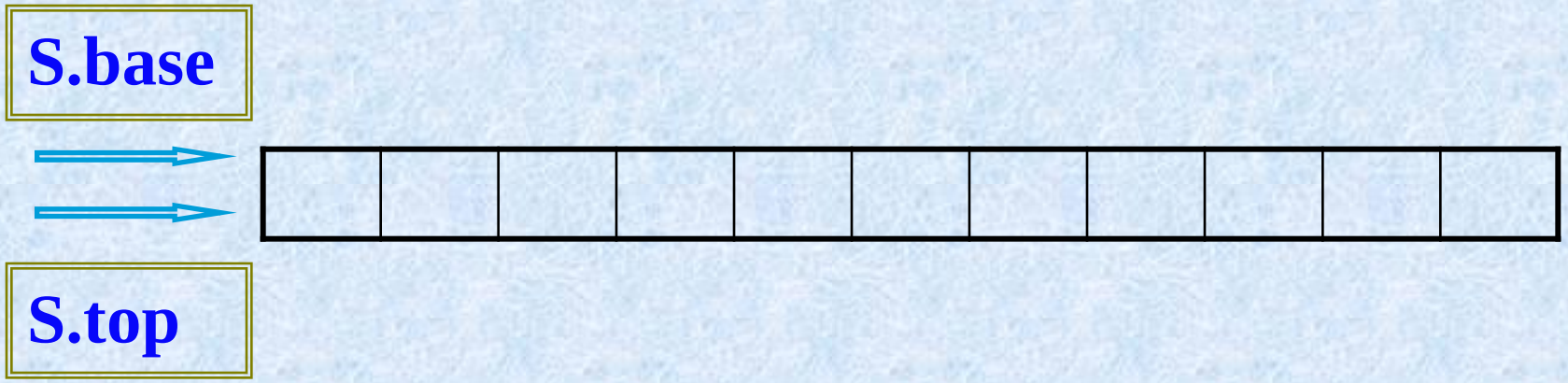
```
//----- 栈的顺序存储表示 -----
#define  STACK_INIT_SIZE  100; //栈容量
#define  STACKINCREMENT   10;  //栈增量
typedef struct {
  SElemType  *base;    //基地址
  SElemType  *top;     //栈顶指针
  int  stacksize;      //栈容量
} SqStack;

  SqStack  S;
```

栈初始化过程演示

(1) 给栈S申请栈空间

(2) 设置基地址S.base和栈顶地址S.top

(3) 设置栈容量S.stacksize=STACK_INIT_SIZE



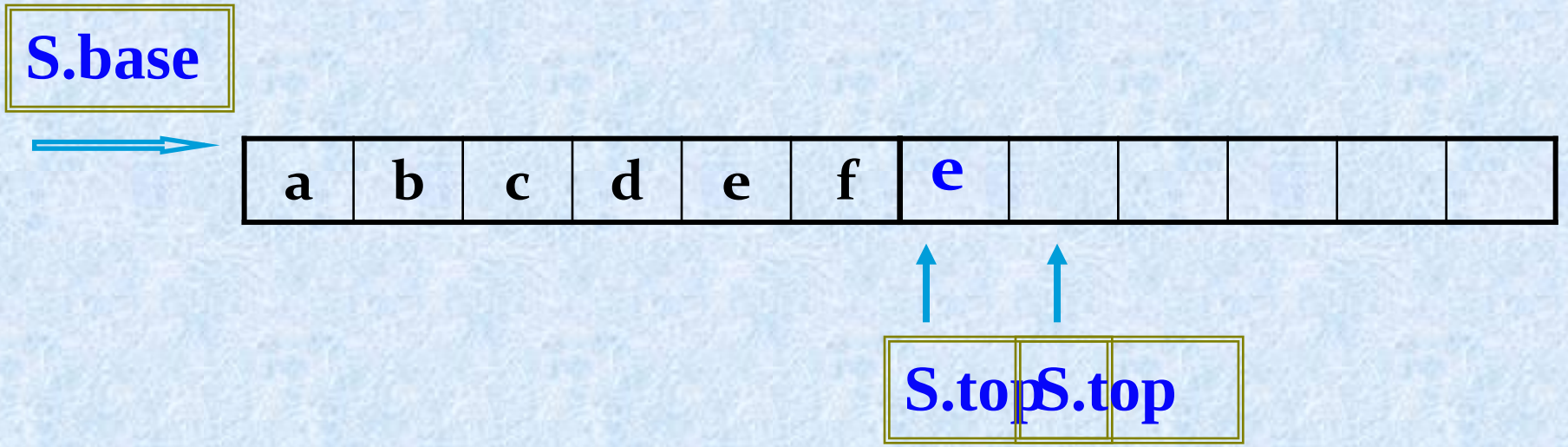
栈初始化算法

```
Status InitStack (SqStack &S){
                                        // 构造一个空栈S
 S.base=(ElemType*)malloc(STACK_INIT_SIZE*
                                sizeof(ElemType));
  if (!S.base) exit (OVERFLOW); //存储分配失败
  S.top = S.base;
  S.stacksize = STACK_INIT_SIZE;
  return OK;
}
```

□入栈操作演示

(1) 如果栈满，给栈增加容量

(2) 将数据存入栈顶位置，栈顶后移一位



□入栈操作演示

```
Status Push (SqStack &S, SElemType e) {
  if (S.top - S.base >= S.stacksize)
    { S.base = (ElemType *) realloc ( S.base,
       (S.stacksize + STACKINCREMENT) *
                                    sizeof (ElemType));
      if (!S.base) exit (OVERFLOW); //存储分配失败
      S.top = S.base + S.stacksize;
      S.stacksize += STACKINCREMENT;
    }
  *S.top++ = e; return OK;
}
```

3.1.2 栈的表示和实现

□其它栈操作讨论

```
DestroyStack(&S)          //销毁栈
ClearStack(&S)            //清空栈
StackEmpty(S)             //判栈空
StackLength(S)            //求栈长度
GetTop(S, &e)             //取栈顶元素
Pop(&S, &e)               //出栈
StackTravers(S, visit())  //遍历栈
```

链栈的实现与链表的实现基本相同，头结点作为栈顶位置。

□链栈的C语言类型定义

```
Typedef struct  SNode {
        ElemType     data;      //数据域
        struct Snode *next;     //链域
 }SNode, *LinkStack;
```

3.1.2 栈的表示和实现

- 讨论链栈基本操作的实现

```
InitStack(&S)                  //初始化栈
DestroyStack(&S)               //销毁栈
ClearStack(&S)                 //清空栈
StackEmpty(S)                  //判栈空
StackLength(S)                 //求栈长度
GetTop(S, &e)                  //取栈顶元素
Push(&S, e)                    //入栈
Pop(&S, &e)                    //出栈
StackTravers(S, visit())       //遍历栈
```

3.2 栈的应用举例

3.2.1 数制转换

3.2.2 括号匹配的检验

3.3.4 行编辑程序问题

3.2.4 迷宫求解

3.2.5 表达式求值

□进制转换原理：

任何X进制数N转换成Y进制数其结果都是要化成如下形式。

$$N_X = a_n \times Y^n + \ldots + a_2 \times Y^2 + a_1 \times Y + a_0 \times Y^0$$

转换的过程就是通过取余运算求出$a_0,a_1,\ldots,a_n$，而先求出的是个位，十位…，与我们写数的习惯先从高位写正好相反，通过栈先进后出的特点，很容易实现倒过来的过程。

例 将10进制1346转换成8进制

过程如下：

计算顺序

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

输出顺序

10进制数N转换成8进制的算法

```
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 );
    }
} // conversion
```

3.2.2 括号匹配的检验

□问题描述

一个表达式中，包含三种括号“(”和“)”，“[”和“]”和“{”和“}”，这三种括号可以按任意的合法次序使用。

设计算法检验表达式中所使用括号的合法性。

□问题讨论

讨论：如果第一次遇到的右括号是“]”，哪么前面出现的左括号有什么特点。

结论：如果第一次遇到的右括号是“]”，哪么前面出现的左括号最后一个必然是“[”，否则不合法。

3.2.2 括号匹配的检验

□算法过程

(1) 当遇到左括号时，进栈，遇到右括号时出栈；

(2) 当遇到某一个右括号时，栈已空，说明到目前为止，右括号多于左括号；

(3) 从栈中弹出的左括号与当前检验的右括号类型不同，说明出现了括号交叉情况；

(4) 算术表达式输入完毕，但栈中还有没有匹配的左括号，说明左括号多于右括号。

3.2.2 括号匹配的检验

□括号匹配检验算法

```
Status  check( ) {
  char ch;  InitStack(S);
  while ((ch=getchar())!='#') {
     switch (ch) {
        case (ch=='('||ch=='['||ch=='{'):Push(S,ch);break;
        case (ch== ')'):
           if (StackEmpty(S)) retrun FALSE;
           else
              {Pop(S,e);if(e!= '(') return FALSE;}
           break;
```

括号匹配检验算法

```
case (ch== ']'):
  if (StackEmpty(S)) retrun FALSE;
  else
  {Pop(S,e);if(e!= '[') return FALSE;}
   break;……
 default:break;
}
}
if (StackEmpty(S))  return TRUE;
else return FALSE;
}
```

3.2.3 行编辑程序问题

问题描述

在用户输入一行的过程中，允许用户输入出差错，并在发现有误时可以及时更正。

解决办法

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，并假设“#”为退格符，“@”为退行符。

例 假设从终端接受了这样两行字符：

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

则实际有效的是下列两行：

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

3.2.3 行编辑程序问题

□行编辑问题算法

```
void LineEdit(){
  //利用字符栈S，从终端接收一行并传送至调
  //用过程的数据区
    InitStack(S);
    ch=getchar();
    while (ch != EOF) { //EOF为全文结束符
      while (ch != EOF && ch != '\n') {……}
     ……
    DestroyStack(S);
}
```

3.2.3 行编辑程序问题

□行编辑问题算法

```
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
```

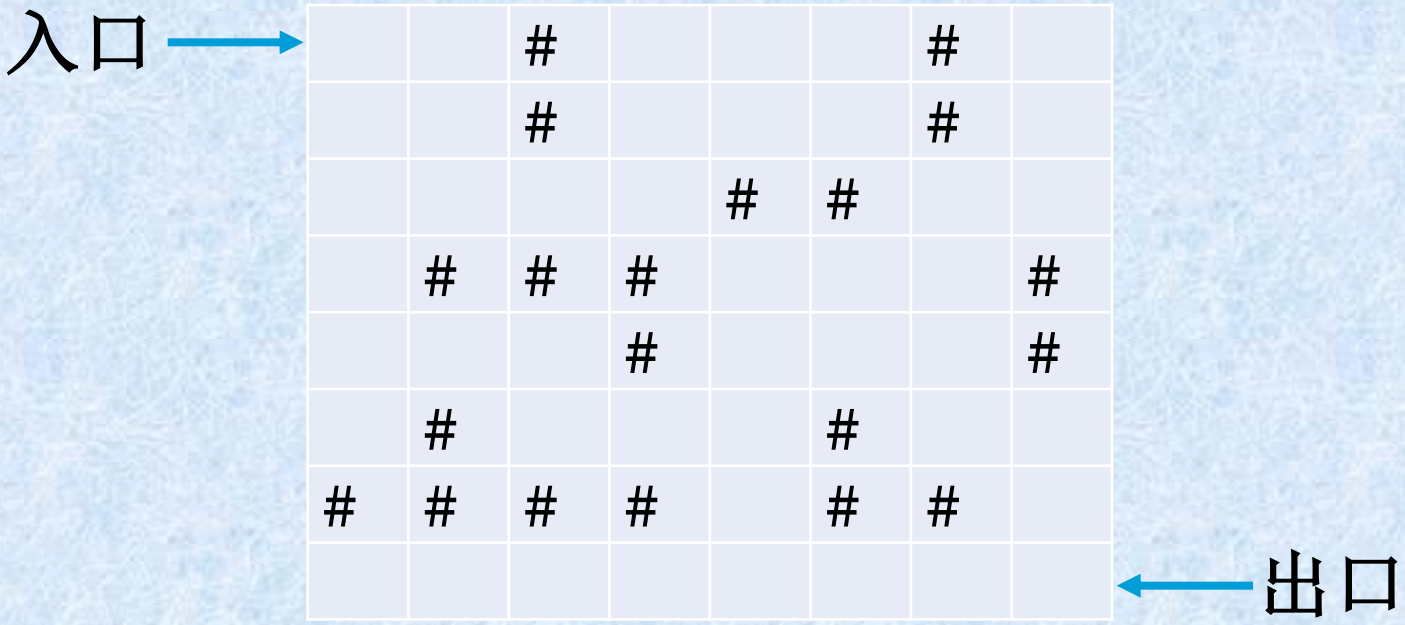
栈中字符传送至调用过程的数据区；

```
ClearStack(S);          // 重置S为空栈
if (ch != EOF)  ch = getchar();
```

3.2.4 迷宫求解

□问题描述

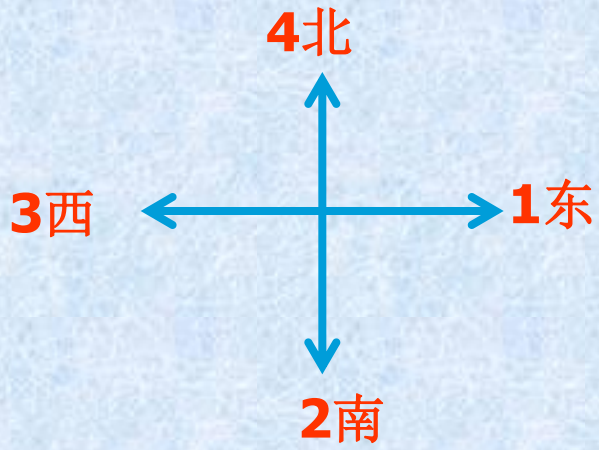
如图表示一个迷宫，有一个入口，一个出口，从一个位置可以向4个方向走，空表示能走通，#表示走不通，求从入口到出口的路径。



		#				#	
		#				#	
				#	#		
	#	#	#				#
			#				#
	#				#		
#	#	#	#		#	#	

四周墙壁解决办法如图

	0	1	2	3	4	5	6	7	8	9
0	#	#	#	#	#	#	#	#	#	#
1	#	→	↓	#	$	$		#		#
2	#		↓	#	$	$	$	#		#
3	#	↓	←	$	$	#	#			#
4	#	↓	#	#	#				#	#
5	#	→	→	↓	#				#	#
6	#		#	→	→	↓	#			#
7	#	#	#	#	#	↓	#	#		#
8	#					→	→	→	Θ	#
9	#	#	#	#	#	#	#	#	#	#



箭头：走向
$：走不通
Θ：成功走出

maze[10][10]

3.2.4 迷宫求解

- 用栈实现迷宫路径求解的过程

(1) 入口位置入栈，作标记以免重复进入。

(2) 从栈顶位置顺时针从东（1）开始选一个可以进的位置，并将此位置标记避免重复，新进位置入栈。

(3) 重复(2)，若栈顶位置无路可走，则退栈，从新的栈顶重复(2)。

(4) 直到找到入口，从栈底到栈顶即是路径，或者栈空，表示从入口到出口没通路。

3.2.4 迷宫求解

□迷宫路径求解算法

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）；

3.2.4 迷宫求解

□迷宫路径求解算法

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；}

若栈空，则表明迷宫没有通路。

问题描述

一个表达式由操作数(operand)、运算符(operator)、界限符(delimiter)组成。写出“算符优先法”求值的算法。

例 求3*(2+3*5)+6的值

□算法求解过程

设置两个栈，一个存操作数，栈名为OPND，一个存操作符，栈名为OPTR栈。

(1) 首先置操作数栈为空，表达式起始符#为运算符栈的栈底元素；

(2)依次读入表达式中每个字符，若是操作数则进OPND栈，若是运算符则和OPTR栈的栈顶运算符比较优先权后作相应操作，直到整个表达式操作完毕。

3.2.5 表达式求值

□算法求解过程

(1)若栈顶运算符小于输入运算符，输入运算符进栈OPTR；

(2)若栈顶运算符等于输入运算符（只有栈顶是“（”，输入是“）”，或者栈顶是“#”，输入是“#）”两种情况，分别去除一对括号，或结束。

(3)若栈顶运算符大于输入运算符，弹出栈顶运算符，从OPND中弹出两个操作数与弹出运算符计算后再存入OPND栈，继续。

3.2.5 表达式求值

□表达式求值算法

```
OperandType EvaluateExpression(){
  initStack(OPTR);Push(OPTR,'#');
  initStack(OPND);c=getchar();//读入第一个符号c
    while(c!='#')||GetTop(OPTR)!='#'){//同时是#结束
      if(!In(c,OP))//如果c不是运算符
     {Push((OPND,c);c=getchar();//读入下一个符号}
      else
          switch(Precede(GetTop(OPTR),c)){//优先级比较
```

3.2.5 表达式求值

□表达式求值算法

```
        case '<': Push(OPTR,c);c=getchar();break;
        case '=': Pop(OPTR,x);c=getchar;break;
        case '>': Pop(OPTR,theta);
                  Pop(OPND,a); Pop(OPND,b);
                  Push(OPND,Operate(a,theta,b));break;
        } //switch语句结束
    } //while 语句结束
    return(GetTop(OPND));
} //算法结束
```

第3章 栈和队列

3.4.1 抽象数据类型队列的定义

队列的定义

队列(Queue)——是一种运算受限的特殊的线性表，它只允许在表的一端进行插入，而在表的另一端进行删除。

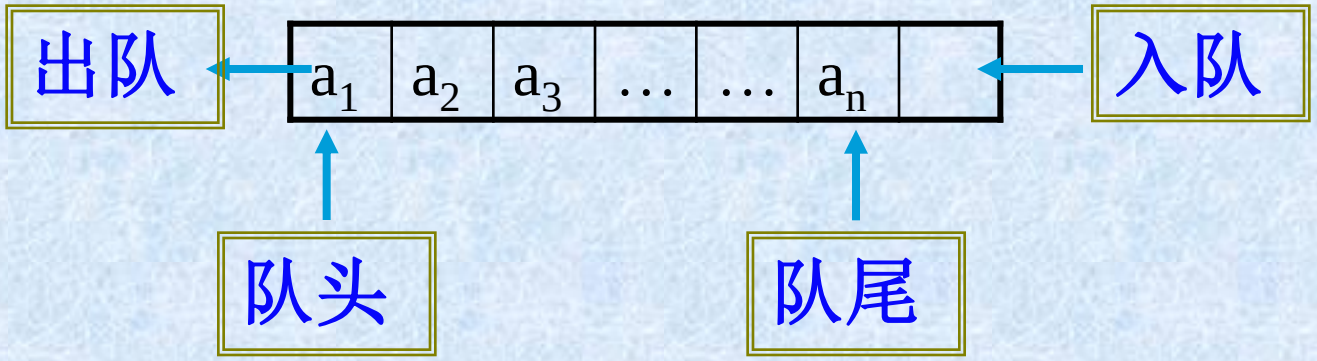
队列的术语

队尾(rear)是队列中允许插入的一端。

队头(front)是队列中允许删除的一端。

3.4.1 抽象数据类型队列的定义

队列示意图



队列的特点

先进先出(First In First Out，简称FIFO)。
又称队列为先进先出表。

3.1.1 抽象数据类型栈的定义

□抽象数据类型栈

ADT Queue {

数据对象：$D=\{a_i \mid a_i \in ElemSet, i=1,2,...,n, n\geq 0\}$

数据关系：$R1=\{ <a_{i-1},a_i> \mid a_{i-1}, a_i \in D, i=2,...,n\}$

约定其中a_1端为队列头，a_n端为队列尾

基本操作：

见下页

} ADT Queue

3.1.1 抽象数据类型栈的定义

□队列的基本操作

```
InitQueue(&Q)                  //初始化队列
DestroyQueue(&Q)               //销毁队列
QueueEmpty(Q)                  //判队列是否空
QueueLength(Q)                 //求队列长度
GetHead(Q, &e)                 //取队头元素
ClearQueue(&Q)                 //清空队列
EnQueue(&Q, e)                 //入队列
DeQueue(&Q, &e)                //出队列
QueueTravers(Q, visit())       //遍历队列
```

3.4.2 链队列

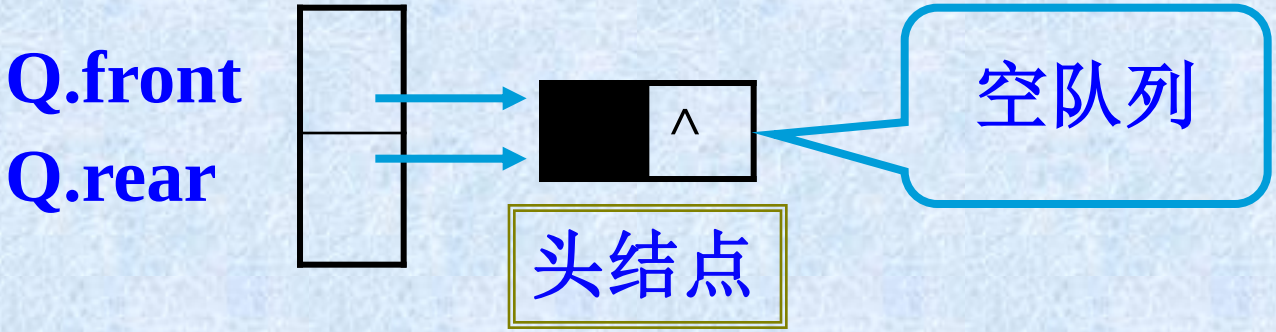
□链队列结点实现

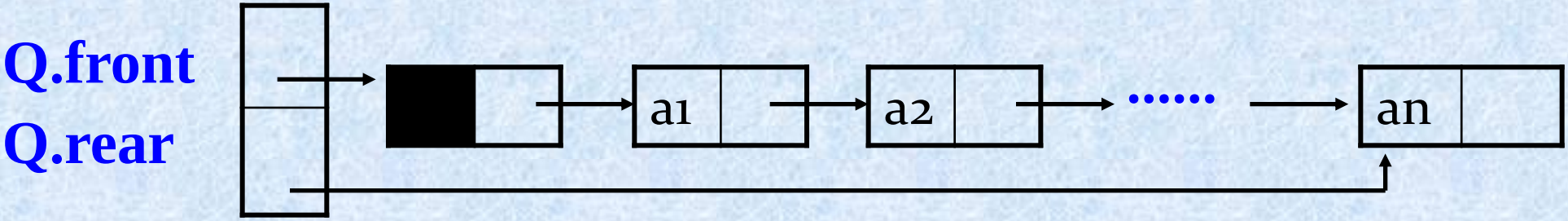
```
typedef struct QNode { // 结点类型
  QElemType     data;
  struct QNode  *next;
} QNode, *QueuePtr;
```

□链队列数据类型实现

```
typedef struct {          // 链队列类型
  QueuePtr  front;   // 队头指针
  QueuePtr  rear;    // 队尾指针
} LinkQueue;
```

- 带头结点的链队列示意图





- 带头结点的链队列初始化

```
Status InitQueue (LinkQueue &Q) {
                                    // 构造一个空队列Q
  Q.front = Q.rear =
          (QueuePtr)malloc(sizeof(QNode));
  if (!Q.front) exit (OVERFLOW);
                                    //存储分配失败
  Q.front->next = NULL;
  return OK;
}
```

带头结点的链队列入队算法

```
Status EnQueue (LinkQueue &Q, QElemType e)
{  // 插入元素e为Q的新的队尾元素
   p = (QueuePtr) malloc (sizeof (QNode));
   if (!p)  exit (OVERFLOW);   //存储分配失败
   p->data = e;   p->next = NULL;
   Q.rear->next = p;    Q.rear = p;
   return OK;
}
```

□带头结点的链队列出队算法

```
Status DeQueue (LinkQueue &Q, QElemType &e)
{ // 若队列不空，则删除Q的队头元素，
  //用 e 返回其值，并返回OK；否则返回ERROR
  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;
}
```

- 带头结点的链队列其它操作讨论

```
DestroyQueue(&Q)          //销毁队列
QueueEmpty(Q)             //判队列是否空
QueueLength(Q)            //求队列长度
GetHead(Q, &e)            //取队头元素
ClearQueue(&Q)            //清空队列
QueueTravers(Q, visit())  //遍历队列
```

3.4.3 循环队列

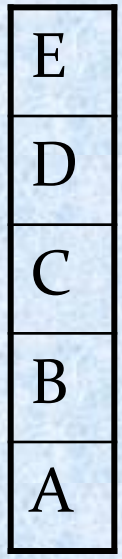
□顺序队列讨论

循环队列属于顺序队列的一种，讨论在采用一般顺序队列时出现的问题。

Sq.rear=0
Sq.front=0

Sq.rear=1
Sq.front=0

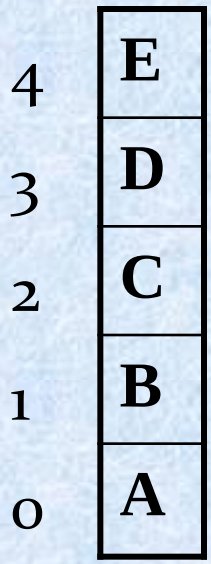
Sq.rear=2
Sq.front=0



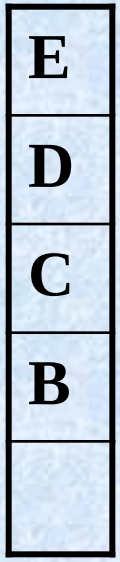
Sq.rear=5
Sq.front=0

3.4.3 循环队列

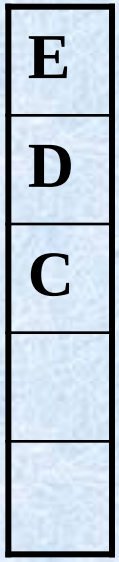
□顺序队列讨论



Sq.rear=5
sq.front=0



Sq.rear=5
Sq.front=1



Sq.rear=5
Sq.front=2

Sq.rear=5
Sq.front=5

讨论结论：在采用一般顺序队列时出现假上溢现象？

3.4.3 循环队列

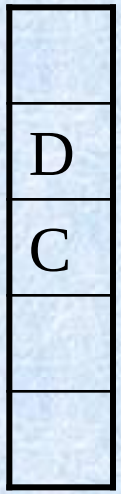
□顺序队列数据类型实现

```
#define MAXQSIZE  100  //最大队列长度
  typedef struct {
    QElemType  *base;  // 动态分配存储空间
    int  front;     // 头指针，若队列不空，
                    //  指向队列头元素
    int  rear;     // 尾指针，若队列不空，指向
                      队列尾元素 的下一个位置
  } SqQueue;

    SqQueue    Sq;
```

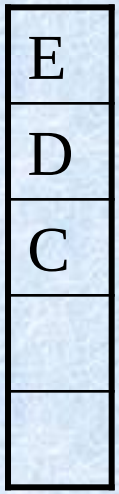
3.4.3 循环队列

□循环队列的定义

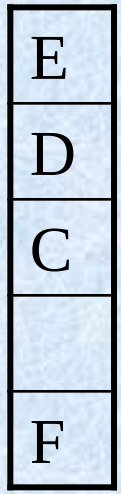
循环队列是顺序队列的一种特例，它是把顺序队列构造成一个首尾相连的循环表。指针和队列元素之间关系不变。



Sq.rear=4
Sq.front=2



Sq.rear=0
Sq.front=2

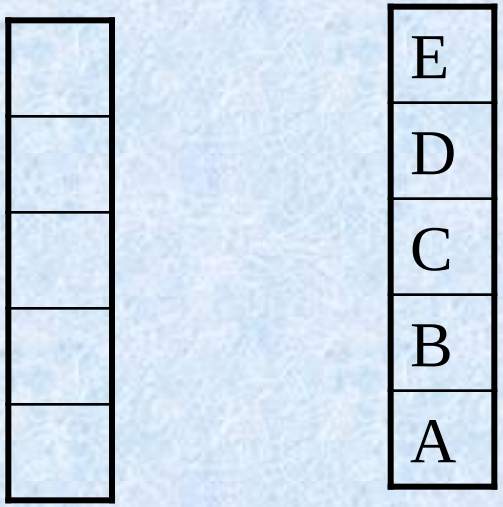


Sq.rear=1
Sq.front=2

3.4.3 循环队列

□循环队列空状态和满状态的讨论

空状态　　满状态



Sq.rear=0　　**Sq.rear=0**
Sq.front=0　　**Sq.front=0**

讨论结论：循环队列空状态和满状态都满足
Q.front=Q.rear

- 循环队列空状态和满状态的判别

(1) 另设一个标志区别队列是空还是满；

例如：设一个变量count用来记录队列中元素个数，当count==0时队列为空，当count= MAXQSIZE时队列为满。

□循环队列空状态和满状态的判别

(2)队满条件：以队头指针在队列尾指针的下一位置作为队列呈满状态的标志，牺牲一个存储空间；

队满条件为：

(sq.rear+1) mod maxsize==sq.front

队空条件为：**sq.rear==sq.front**

队列初始化算法

```
Status InitQueue (SqQueue &Q) {
                                    // 构造一个空队列Q
  Q.base = (ElemType *) malloc
          (MAXQSIZE *sizeof (ElemType));
  if (!Q.base) exit (OVERFLOW);
                                    // 存储分配失败
  Q.front = Q.rear = 0;
  return OK;
}
```

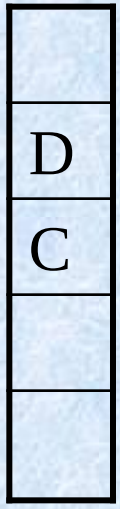
□入队列算法

```
Status EnQueue (SqQueue &Q, ElemType e) {
                                    // 插入元素e为Q的新的队尾元素
    if ((Q.rear+1) % MAXQSIZE == Q.front)
        return ERROR; //队列满
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear+1) % MAXQSIZE;
    return OK;
}
```

出队列算法

```
Status DeQueue (SqQueue &Q, ElemType &e) {
  // 若队列不空，则删除Q的队头元素，
  // 用e返回其值，并返回OK；否则返回ERROR
   if (Q.front == Q.rear)  return ERROR;
   e = Q.base[Q.front];
   Q.front = (Q.front+1) % MAXQSIZE;
   return OK;
}
```

分析以下两种状态如何求队列长度



sq->rear=4
sq->front=2

sq->rear=1
sq->front=4

求队列长度算法

```
int QueueLength(SqQueue Q)
{
    return (Q.rear-Q.front+MaxSize)%MaxSize;
}
```

单选题 1分

设置

在具有n个单元的顺序存储的循环队列中，假定front和rear分别为队首指针和队尾指针，则判断队满的条件是( )。

A rear%n==front

B (rear-1)%n==front

C (rear-1)%n==rear

D (rear+1)%n==front

提交

单选题 1分

设置

输入序列为ABC，可以变为BCA时，经过的栈操作为

A push, pop, push, pop, push, pop

B push, push, push, pop, pop, pop

C push, push, pop, push, pop, pop

D push, pop, push, push, pop, pop

提交

单选题 1分

设置

在具有n个单元的顺序存储的循环队列中，假定front和rear分别为队首指针和队尾指针，则计算队列长度的是( )。

A rear-front

B (rear-front)%n

C (rear-front+n)%n

D (rear+fornt-n)%n

提交

本章小结

熟练掌握：

(1)栈、队列的定义、特点和性质；

(2)ADT栈、ADT队列的设计和实现以及基本操作及相关算法。

重点学习：

ADT栈和队列在表达式求值、括号匹配、数制转换、迷宫求解中的应用，提高利用栈和队列解决实际问题的应用水平。