



数据结构

第三章 栈和队列

主讲：陈锦秀

厦门大学信息学院计算机系

第三章 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

3.2 栈的应用举例

3.2.1 数制转换

3.2.2 括号匹配的检验

3.2.3 行编辑程序

3.2.4 迷宫求解

3.2.5 表达式求值

第三章 栈和队列

3.3 栈与递归的实现

3.4 队列

3.4.1 抽象数据类型队列的定义

3.4.2 链队列——队列的链式表示和实现

3.4.3 循环队列——队列的顺序表示和实现

3.5 离散事件模拟

通常称，栈和队列是限定插入和删除只能在表的“端点”进行的线性表。

线性表

Insert(L, **i**, x)

$1 \leq i \leq n+1$

Delete(L, **i**)

$1 \leq i \leq n$

栈

Insert(S, **n+1**, x)

Delete(S, **n**)

队列

Insert(Q, **n+1**, x)

Delete(Q, **1**)

栈和队列是两种常用的数据类型

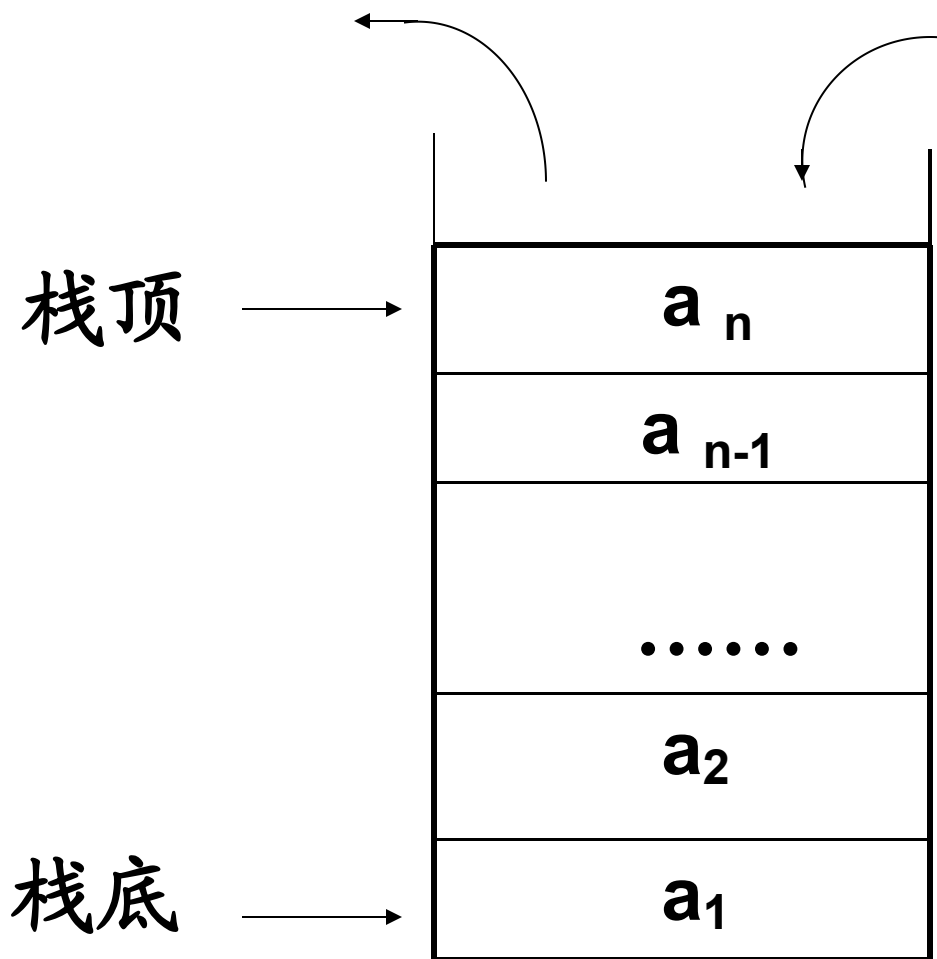
3.1 栈

3.1.1 栈的定义及基本运算

- 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 $S=(a_1, a_2, a_3, \dots a_n)$ ，则 a_1 称为栈底元素， a_n 为栈顶元素。栈中元素按 $a_1, a_2, a_3, \dots a_n$ 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 (**LIFO**)。

例、一叠书或一叠盘子。

栈的抽象数据类型的定义如下： **P₄₅**



3.1.1 栈的定义及基本运算

栈：只能在表的一端进行插入和删除操作的线性表。

栈顶 (top)：允许插入和删除的一端。

栈底 (bottom)：不允许插入和删除的另一端。

空栈：不含元素的空表。

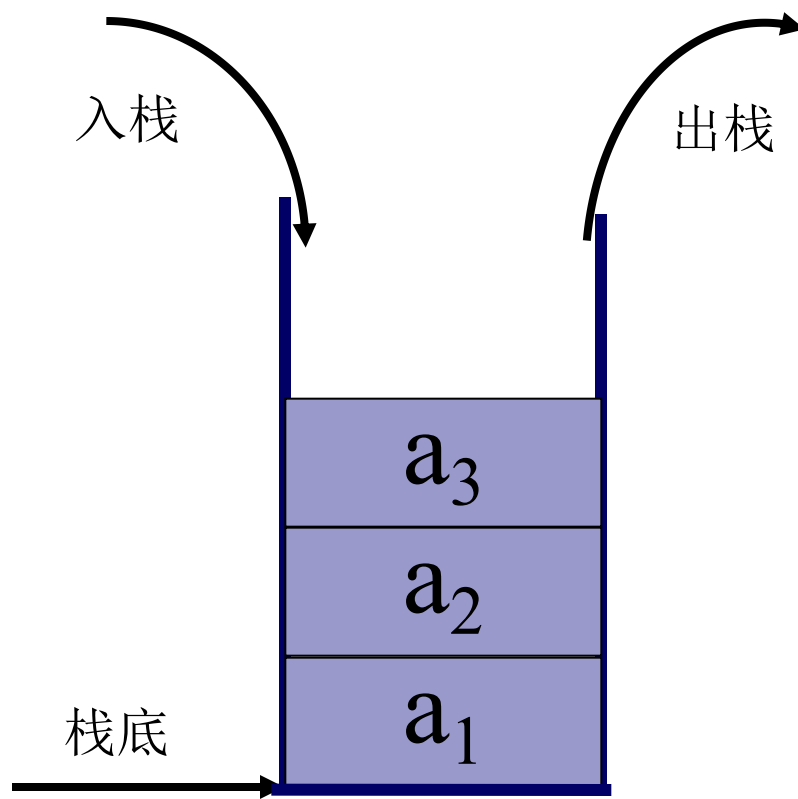
插入：入栈、进栈、压栈

删除：出栈、弹栈

特点：

先进后出 (**FILO**)

后进先出 (**LIFO**)



3.1.1 栈的定义及基本运算

栈除了在栈顶进行进栈与出栈外，还有初始化、判空等操作，常用的基本操作有：

- (1) **构造栈**，即构造一个空栈S。
- (2) **判栈空**。若栈S为空，则返回1；否则返回0。
- (3) **判栈满**。若栈S已满，则返回1，否则返回0，该运算只适用于栈的顺序存储结构。
- (4) **进栈**。若栈S未满，将数据元素x插入栈S中，使其为栈S的栈顶元素。
- (5) **出栈**。若栈非空，从栈S中删除当前栈顶元素。
- (6) **取栈顶元素**。若栈S非空，取栈顶元素，操作结果只是读取栈顶元素，栈S不发生变化。

3.1.1 栈的定义及基本运算

栈的抽象数据类型定义:

ADT Stack {

数据对象:

$$D = \{ a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$$

数据关系:

$$R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$$

约定 a_n 端为栈顶, a_1 端为栈底。

基本操作:

} ADT Stack

基本操作:

InitStack(&S)

操作结果: 构造
一个空栈 **S**。

DestroyStack(&S)

初始条件: 栈 **S** 已存在。
操作结果: 栈 **S** 被销毁。

StackEmpty(s)

初始条件: 栈 **S** 已存在。
操作结果: 若栈 **S** 为空栈, 则
返回 **TRUE**, 否则 **FALSE**。

StackLength(S)

初始条件: 栈 **S** 已存在。
操作结果: 返回 **S** 的元素个数,
即栈的长度。

ClearStack(&S)

初始条件: 栈 **S** 已存在。
操作结果: 将 **S** 清为空栈。

GetTop(S, &e)

Push(&S, e)

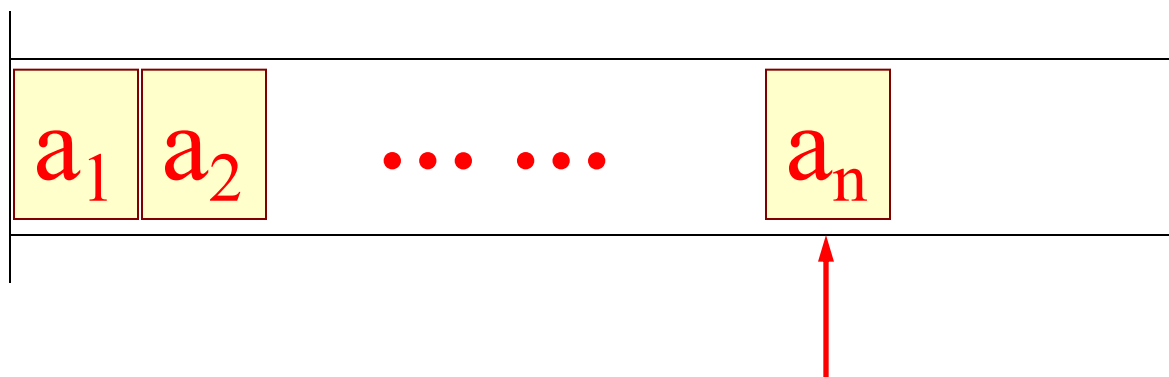
Pop(&S, &e)

StackTravers(S, visit())

GetTop(S, &e)

初始条件：栈 S 已存在且非空。

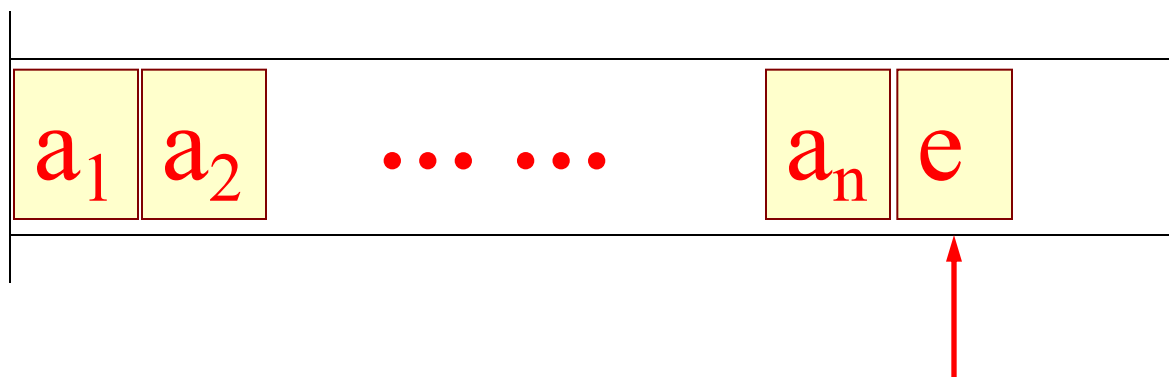
操作结果：用 e 返回 S 的栈顶元素。



Push(&S, e)

初始条件：栈 S 已存在。

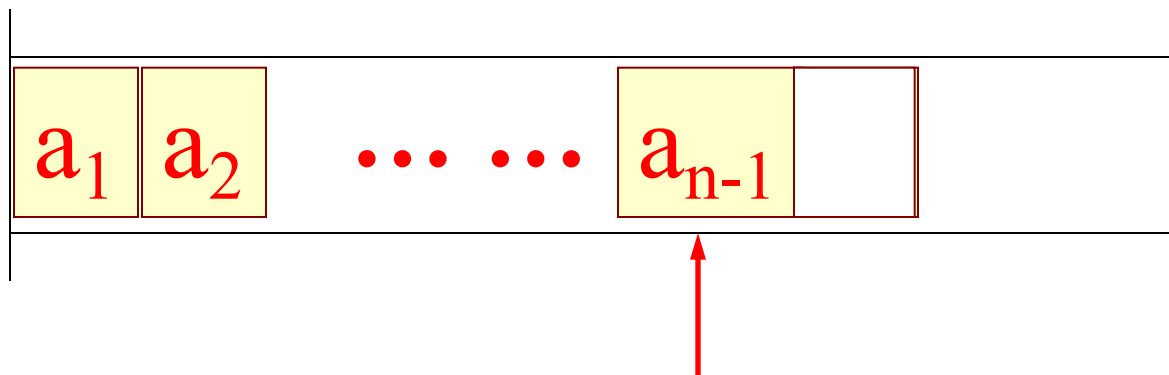
操作结果：插入元素 e 为新的栈顶元素。



Pop(&S, &e)

初始条件：栈 S 已存在且非空。

操作结果：删除 S 的栈顶元素，并用 e 返回其值。

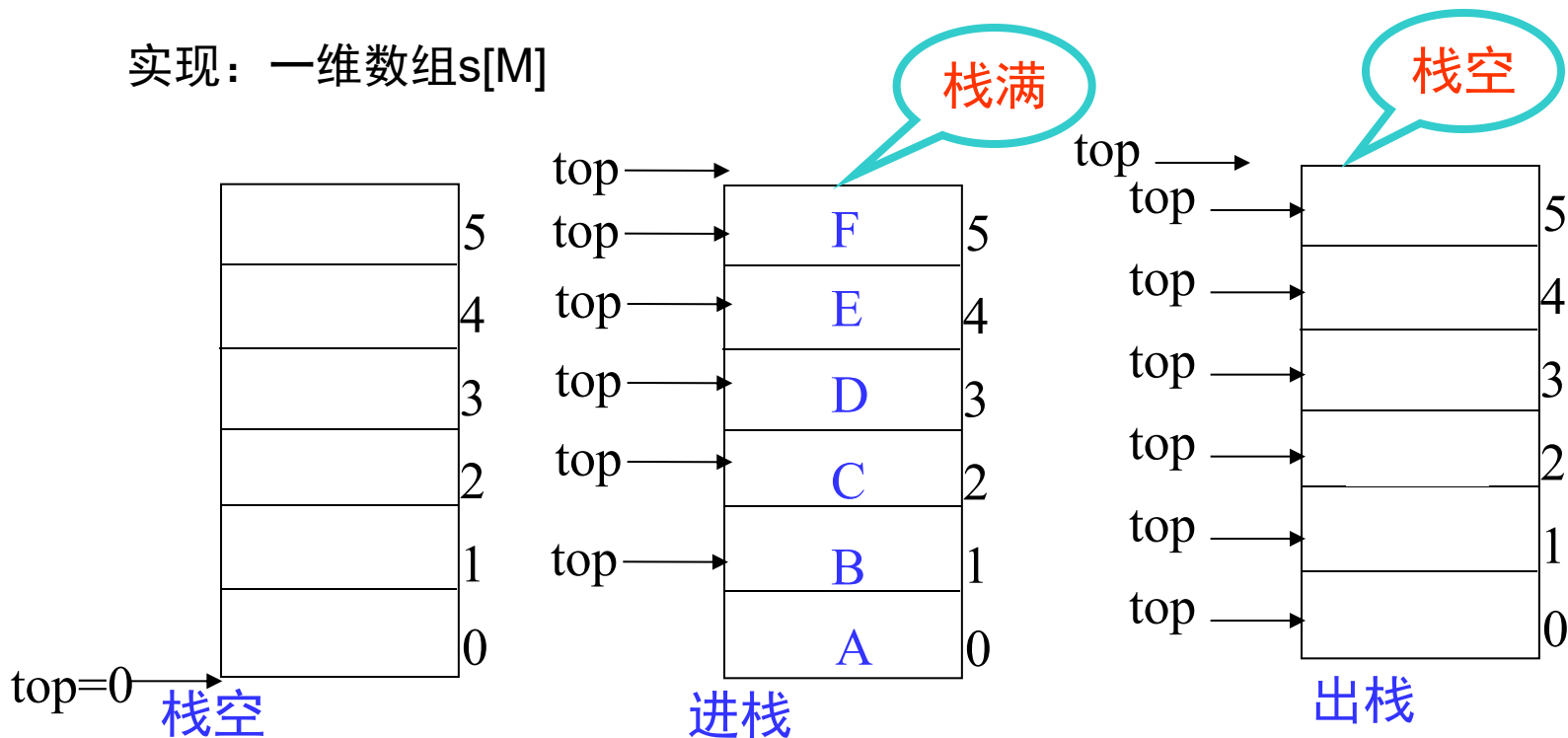


3.1.2 顺序栈

- 由于栈是运算受限的线性表，因此线性表的存储结构对栈也适应。
- 栈的顺序存储结构简称为顺序栈，它是运算受限的线性表。因此，可用数组来实现顺序栈。因为栈底位置是固定不变的，所以可以将栈底位置设置在数组的两端的任何一个端点；栈顶位置是随着进栈和退栈操作而变化的，故需用一个整型变量top来表示。

3.1.3 栈的表示和实现——顺序栈

实现：一维数组s[M]



栈顶指针 top ,指向实际栈顶
后的空位置, 初值为0
进栈: top 加1

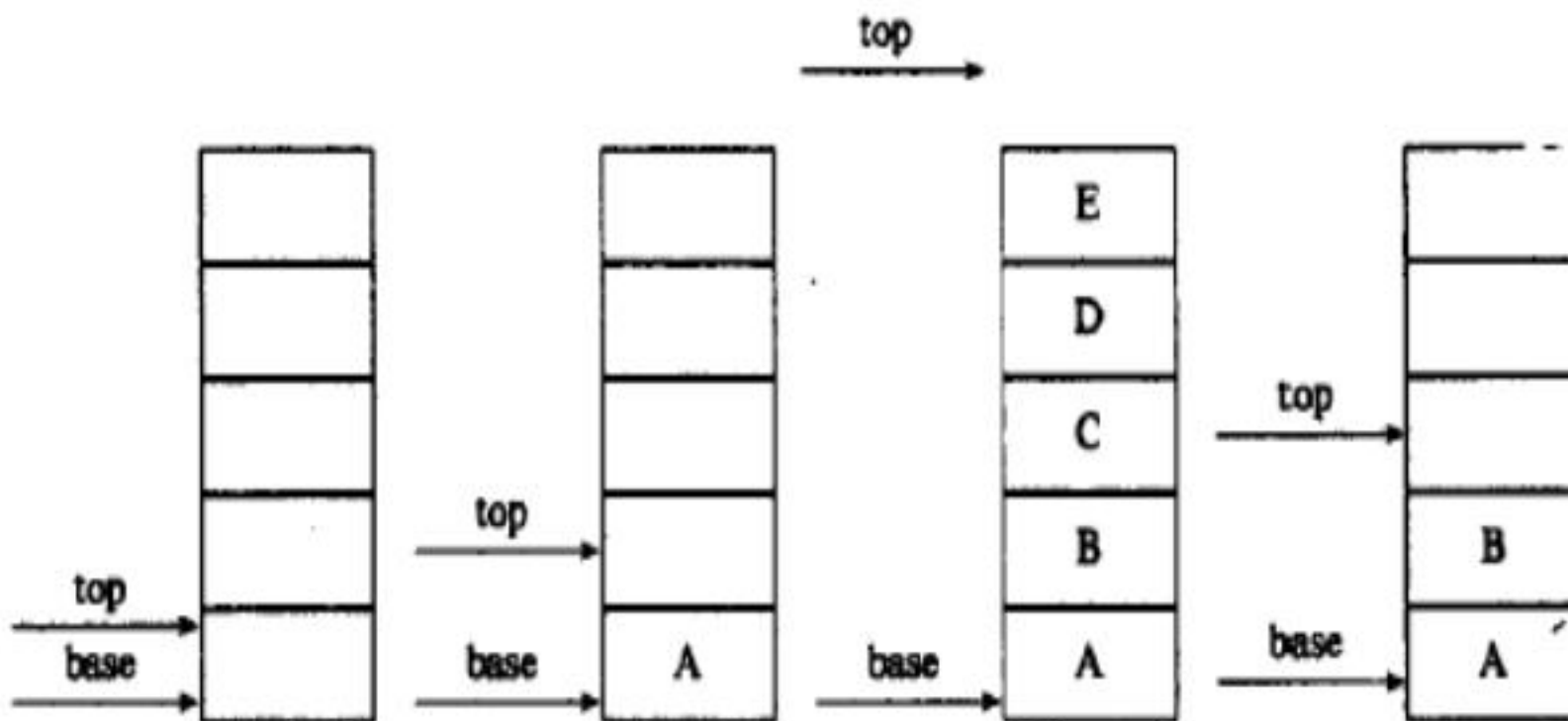
设数组大小为 M
 $top=0$,栈空, 此时出栈, 则下溢 (underflow)
 $top=M$,栈满, 此时入栈, 则上溢 (overflow)
出栈: top 减1

3.1.2 顺序栈

- top用来指示当前栈顶的位置，通常称top为栈顶指针。
- 顺序栈的类型定义如下：

```
#define STACK_INIT_SIZE 100; //存储空间初始分配量
#define STACKINCREMENT 10; //存储空间分配增量
typedef struct{
    SElemType *base; //在栈构造之前和销毁之后，为null
    SElemType *top; //栈顶指针
    int stacksize; //当前已分配的存储空间，以元素为单位
}SqStack;
```


栈顶指针和栈中元素之间的关系：



3.1.2 顺序栈

- 设S是**SqStack**类型的指针变量。
 - 若栈底位置在向量的低端，即**s->base**是栈底元素，那么栈顶指针**s->top**是正向增加的，即进栈时需将**s->top**加1，退栈时需将**s->top**减1。
 - **s->top==s->base**表示空栈，当栈空时再做退栈运算也将产生溢出，简称“**下溢**”。
 - **s->top-s->base == stacksize**表示栈满。当栈满时再做进栈运算必定产生空间溢出，简称“**上溢**”。

3.1.2 顺序栈——基本操作的算法

1、初始化栈

算法：

[构建栈]

1.1 分配空间并检查空间是否分配失败，若失败则返回错误

1.2 设置栈底和栈顶指针

1.3 设置栈大小

[算法结束]

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;

}

2、退栈

算法:

[退栈]

4.1 判断栈是否为空，为空则返回错误。

4.2 获取栈顶元素 e

4.3 栈顶指针减1

[算法结束]



```
Status Pop (SqStack &S, SElemType &e) {
```

```
    // 若栈不空，则删除S的栈顶元素，
```

```
    // 用e返回其值，并返回OK;
```

```
    // 否则返回ERROR
```

```
    if (S.top == S.base) return ERROR;
```

```
    e = *--S.top;
```

```
    return OK;
```

```
}
```

3、入栈

算法:

[初值]

获取入栈元素 e

[入栈]

3.1 判断栈空间是否存在, 若无空间则重新分配更大的空间, 并调整栈底, 栈顶指针以及栈大小。

3.2 元素 e 压入栈中的栈顶位置

3.3 栈顶指针增加1

[算法结束]

```
Status Push (SqStack &S, SElemType e) {  
    if (S.top - S.base >= S.stacksize) {//栈满，追加存储空间  
        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;  
}
```


2、取栈顶元素

算法:

[取元素]

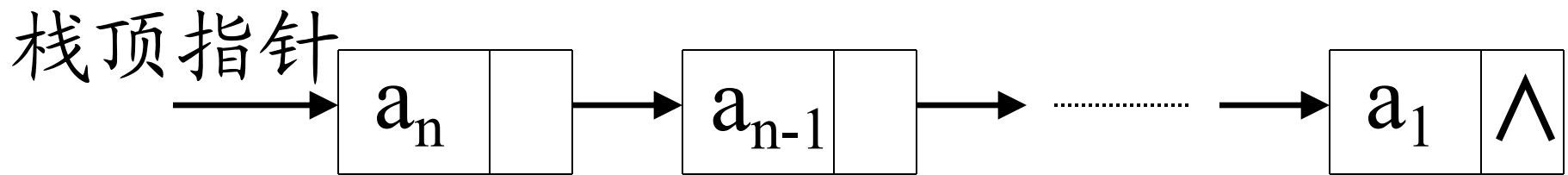
2.1 判断栈是否是空栈，若是空栈则返回错误

2.2 通过栈顶指针获取栈顶元素

[算法结束]

3.1.3 链栈

- 栈的链式存储结构称为链栈，它是运算受限的单链表，插入和删除操作仅限制在表头位置上进行。
- 由于只能在链表头部进行操作，故链表没有必要像单链表那样附加头结点。栈顶指针就是链表的头指针。



注意：链栈中
指针的方向

3.1.3 链栈

- 链栈的类型说明如下：

```
typedef struct stacknode{  
    datatype data;  
    struct stacknode *next;  
} stacknode;  
typedef struct {  
    struct stacknode *top;  
} linkstack;
```



```
void initstack(linkstack *p)
```

```
{  
    p->top=null;  
}
```

```
int stackempty(linkstack *p)
```

```
{  
    return p->top==null;  
}
```



```
void push(linkstack *p,datatype x)
```

```
{
```

```
    stacknode *q;
```


```
    q=(stacknode*)malloc(sizeof(stacknode));
```

```
    q->data=x;
```


```
    q->next=p->top;
```

```
    p->top=q;
```

```
}
```



```
datatype pop(linkstack *p)  
{  
    datatype x;  
    stacknode *q=p->top;  
    if(stackempty(p))  
        error("stack underflow.");  
    x=q->data;  
    p->top=q->next;  
    free(q);  
    return x;  
}
```



```
datatype stacktop(linkstack *p)  
{  
    if(stackempty(p))  
        return ERROR;  
    return p->top->data;  
}
```

3.2 栈的应用举例

- 由于栈结构具有的后进先出的固有特性，致使栈成为程序设计中常用的工具。以下是几个栈应用的例子。

3.2.1 数制转换

- 十进制n和其它d进制数的转换是计算机实现计算的基本问题, 其解决方法很多, 其中一个简单算法基于下列原理:

$$n = (n \text{ div } d) * d + n \text{ mod } d$$

(其中:div为整除运算, mod为求余运算)

3.2.1 数制转换

例如 $(1348)_{10} = (2504)_8$ ，其运算过程如下：

	n	n div 8	n mod 8	
计算顺序 ↓	1348	168	4	↑ 输出顺序
	168	21	0	
	21	2	5	
	2	0	2	

```
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);  
    }  
}
```

3.2.2 括号匹配的检验

- 假设表达式中允许括号嵌套, 则检验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。

例:

(() () (()))
1 2 3 4 5 6 7 8 10 11

- 处理过程也和栈的特点相吻合。在算法中设置一个栈, 每读入一个括号,
 - 若是右括号, 则或者是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。

算法的设计思想:

- 1) 凡出现左括弧, 则进栈;
- 2) 凡出现右括弧, 首先检查栈是否空
若栈空, 则表明该“右括弧”多余,
否则和栈顶元素比较,
若相匹配, 则“左括弧出栈”,
否则表明不匹配。
- 3) 表达式检验结束时,
若栈空, 则表明表达式中匹配正确,
否则表明“左括弧”有余。

```

Status matching(string& exp) {
    int state = 1;
    while (i<=Length(exp) && state) {
        switch of exp[i] {
            case “(” :{Push(S,exp[i]); i++; break;}
            case “)” : {
                if (NOT StackEmpty(S)&&GetTop(S)=“(“)
                    {Pop(S,e); i++;}
                else {state = 0;}
                break; } ... ..
        }
        if (StackEmpty(S)&&state) return OK; .....
    }

```

3.2.3 行编辑程序

“每接受一个字符即存入存储器” ?
并不恰当！！

- 在编辑程序中，设立一个输入缓冲区，用于接受用户输入的一行字符，然后逐行存入用户数据区。允许用户输入错误，并在发现有误时可以及时更正。
(假设“#”为退格符，“@”为退行符)

3.2.3 行编辑程序

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++);
```

行编辑程序算法如下：

```
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
```

```
}
```

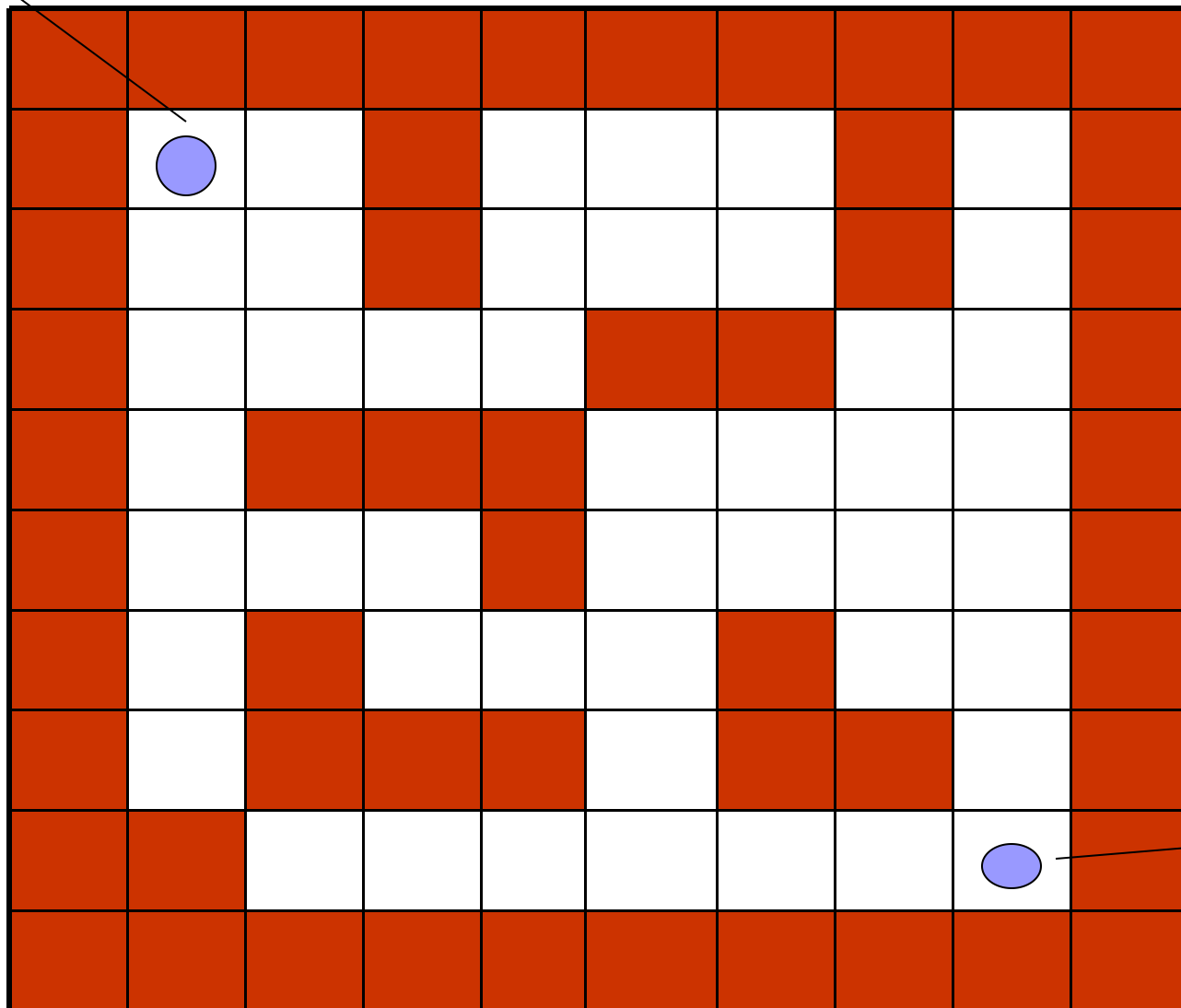
将从栈底到栈顶的字符传送至调用过程的数据区；

```
ClearStack(S); // 重置S为空栈
```

```
if (ch != EOF) ch = getchar();}
```


3.2.4 迷宫求解

入口



出口

通常用的是“穷举求解”的方法：

#	#	#	#	#	#	#	#	#	#
#	→	↓	#	\$	\$	\$	#		#
#		↓	#	\$	\$	\$	#		#
#	↓	←	\$	\$	#	#			#
#	↓	#	#	#				#	#
#	→	→	↓	#				#	#
#		#	→	→	↓	#			#
#	#	#	#	#	↓	#	#		#
#					→	→	→	⊙	#
#	#	#	#	#	#	#	#	#	#

求迷宫路径算法的基本思想是：

- 若当前位置“可通”，则纳入路径，继续前进；
- 若当前位置“不可通”，则后退，换方向继续探索；
- 若四周“均无通路”，则将当前位置从路径中删除出去。

求迷宫中一条从入口到出口的路径的算法:

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);

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；
}

若栈空，则表明迷宫没有通路。

3.2.5 表达式求值

- 例如： $4+2*3-10/5$

计算顺序： $=4+6-10/5=4+6-2=10-2=8$

- 表达式由操作数、运算符、界限符组成



运算符

- 任意两个相继出现的算符 θ_1 和 θ_2 之间的优先关系 (见P53\表3.1)
- **算符优先算法**：使用两个工作栈，分别用以寄存运算符和操作数。

3.2.5 表达式求值

```
■ OperandType EvaluateExpression(){
    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;
                case '=': Pop(OPTR, x); c=getchar(); break;
                case '>': Pop(OPTR, theta); Pop(OPND, b);
                    Pop(OPND, a); Push(OPND, Operate(a, theta, b)); break;
            }
    }
    return GetTop(OPND);
}
```

运算符栈，表达式
起始符“#”为运
算符栈的栈底用
寄存器操作数或运
算结果不是运
算符则栈顶元
素优先
权低

退栈并将
运算结果
入栈

脱括号并
接收下一
字符

3.3 栈与递归的实现

- 主程序在调用子程序的时候，符合“**后进先出**”的特点，可以使用栈进行模拟。
- **当父函数将控制转交给子函数的时候**，需要完成三件任务：
 - 将所有的实参传递给子函数的形参，还要传递返回地址。
 - 为子函数的局部变量分配空间
 - 将控制转移到子函数的入口。
- **当子函数返回父函数的时候**，依次完成如下工作：
 - 保存被调函数的计算结果；
 - 释放被调函数的数据空间；
 - 根据返回地址将控制转交给父函数。
- 例如P56图3.6

3.3 栈与递归的实现

- 递归程序是一种特别的子函数调用，自己调用自己，但参数会越来越小，直到接近出口。
- 因此，递归程序，可以**实现非递归化**：
 - 1、尾递归可以使用循环实现非递归化，如 $n! = n * (n-1)!$
 - 2、其他递归可以使用栈来实现非递归化

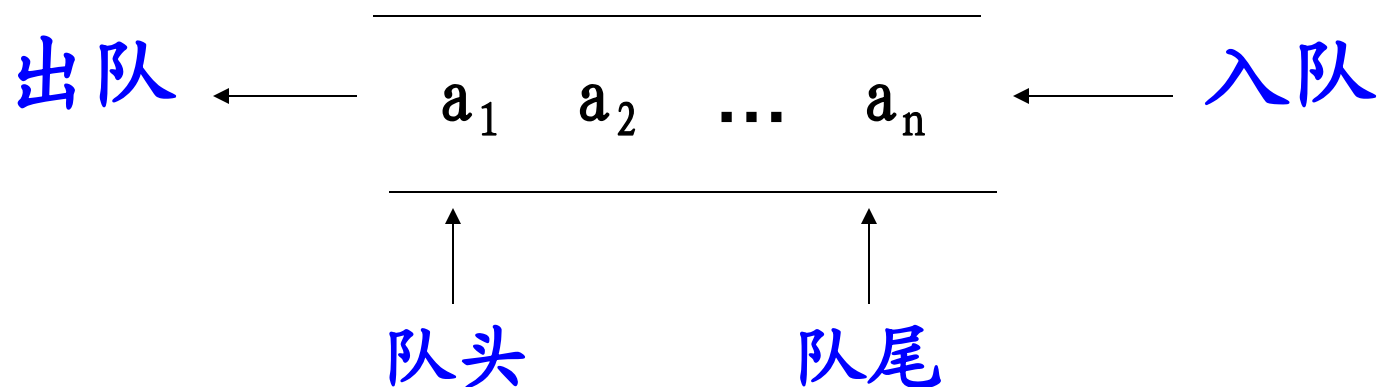
3.4 队列

3.4.1 抽象数据类型队列的定义

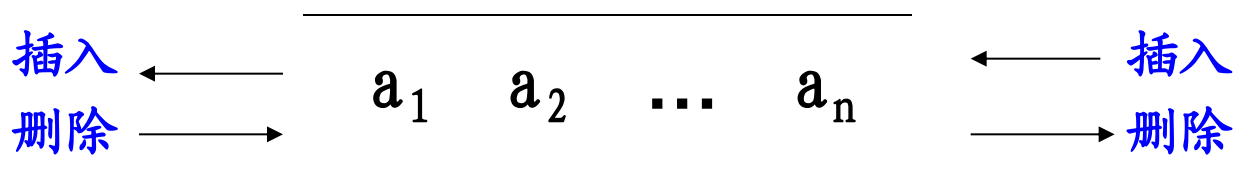
- 队列(Queue)也是一种**运算受限的线性表**。它只允许在表的一端进行插入，而在另一端进行删除。允许删除的一端称为队头(front)，允许插入的一端称为队尾(rear)。
 - 例1：排队购物。
 - 例2：操作系统中的作业排队。
- 先进入队列的成员总是先离开队列。因此**队列亦称作先进先出(First In First Out)的线性表**，简称FIFO表。
- 当队列中没有元素时称为**空队列**。在空队列中依次加入元素 a_1, a_2, \dots, a_n 之后， a_1 是队头元素， a_n 是队尾元素。显然退出队列的次序也只能是 a_1, a_2, \dots, a_n ，也就是说队列的修改是依先进先出的原则进行的。

3.4.1 抽象数据类型队列的定义

- 下图是队列的示意图：



- **双端队列**：限定插入和删除操作在表的两端进行的线性表 P60



3.4.1 抽象数据类型队列的定义

ADT Queue {

数据对象:

$$D = \{a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0\}$$

数据关系:

$$R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$$

约定其中 a_1 端为队列头, a_n 端为队列尾

基本操作:

} ADT Queue

队列的基本操作:

InitQueue(&Q)

DestroyQueue(&Q)

QueueEmpty(Q)

QueueLength(Q)

GetHead(Q, &e)

ClearQueue(&Q)

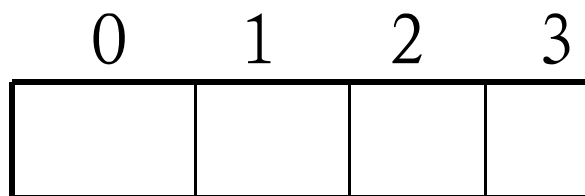
EnQueue(&Q, e)

DeQueue(&Q, &e)

QueueTravers(Q, visit())

3.4.2 循环队列——队列的顺序表示和实现

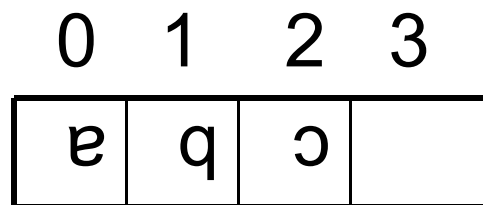
- 队列的顺序存储结构称为**顺序队列**
- 顺序队列实际上是运算受限的顺序表，和顺序表一样，顺序队列也是必须用一个**向量空间来存放当前队列中的元素**。
- 由于队列的队头和队尾的位置是变化的，因而要**设两个指针分别指示队头和队尾元素在队列中的位置**，
 - 它们的**初始值**在队列初始化时均应置为**0**；
 - **入队时**将新元素插入队尾，然后将尾指针加 1；
 - **出队时**，删去所指的元素，然后将头指针加 1 并返回被删元素。
 - 由此可见，当**头尾指针相等时**队列为空。
 - 在非空队列里，头指针始终指向队头元素，而尾指针始终指向队尾元素的下一位置。



Front

rear

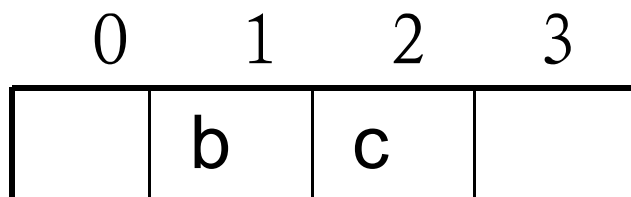
(a) 队列初始为空



Front

rear

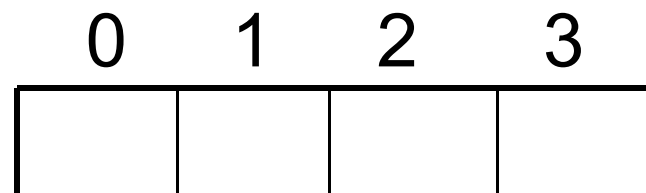
(b) A, B, C入队



front

rear

(c) a出队



front

rear

(d) b, c出队, 队为空

3.4.2 循环队列——队列的顺序表示和实现

- 和栈类似，队列中亦有上溢和下溢现象。此外，顺序队列中还存在“假上溢”现象。
- 因为在入队和出队的操作中，头尾指针只增加不减小，致使被删除元素的空间永远无法重新利用。
- 因此，尽管队列中实际的元素个数远远小于向量空间的规模，但也可能由于尾指针已超出向量空间的上界而不能做入队操作。该现象称为假上溢。
- 为充分利用向量空间，克服假上溢现象：
 - 将向量空间想象为一个首尾相接的圆环，并称这种向量为循环向量，存储在其中的队列称为循环队列（Circular Queue）。

3.4.2 循环队列——队列的顺序表示和实现

- 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（**QueueSize-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(i+1==QueueSize)

i=0;

else

i++;

利用模运算可简化为：**i=(i+1)%QueueSize**

3.4.2 循环队列——队列的顺序表示和实现

- 因为循环队列元素的空间可以被利用，除非向量空间真的被队列元素全部占用，否则不会上溢。因此，除一些简单的应用外，真正实用的顺序队列是循环队列。
- 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。因此，我们无法通过 $\text{front}=\text{rear}$ 来判断队列“空”还是“满”。

3.4.2 循环队列——队列的顺序表示和实现

■ 解决此问题的方法至少有三种：

1. 另设一个标志位以区别队列的空和满；
2. 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（注意：rear所指的单元始终为空）；
3. 其三是使用一个计数器记录队列中元素的总数（实际上是队列长度）。

3.4.2 循环队列——队列的顺序表示和实现

■ 循环队列的类型定义:

```
#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;
```

■ 实现算法见P64-P65

```
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;  
}
```



```
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;
```

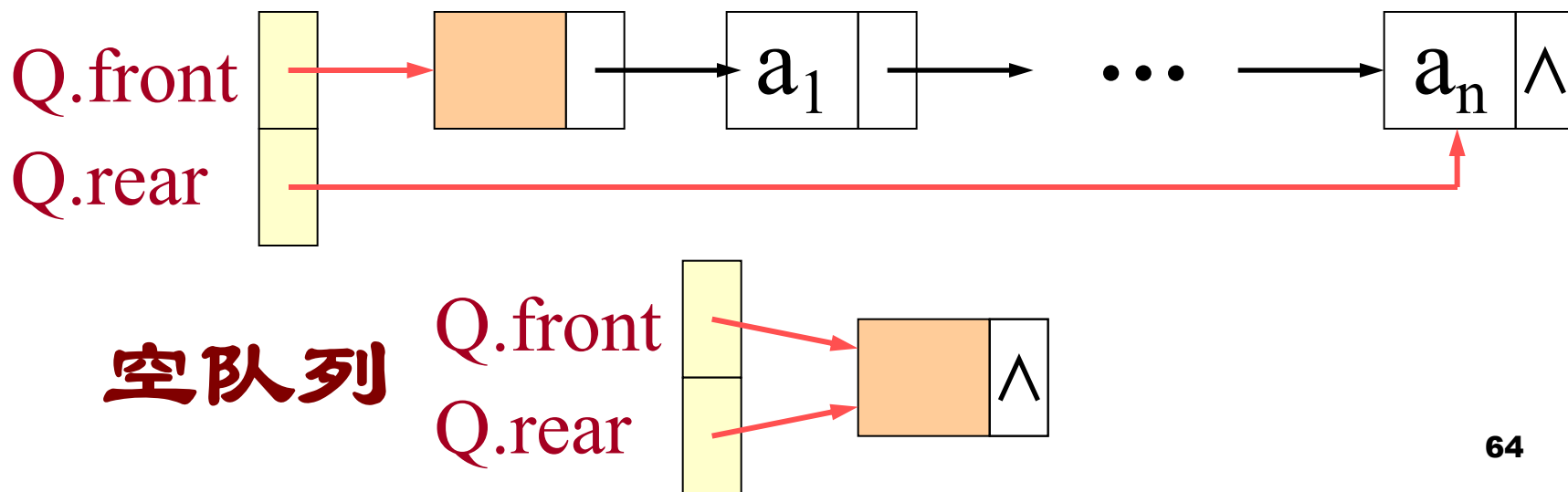
```
}
```



```
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;  
}
```

3.4.3 链队列——队列的链式表示和实现

- 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**。
- 显然仅有单链表的头指针不便于在表尾做插入操作，为此再**增加一个尾指针**，指向链表的最后一个结点。于是，一个链队列由一个头指针唯一确定。
- 和顺序队列类似，我们也是将这两个指针封装在一起，将链队列的类型LinkQueue定义为一个结构类型。



3.4.3 链队列——队列的链式表示和实现

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

typedef struct {          // 链队列类型
    QNode *front; // 队列头指针
    QNode *rear;  // 队列尾指针
} LinkQueue;
```

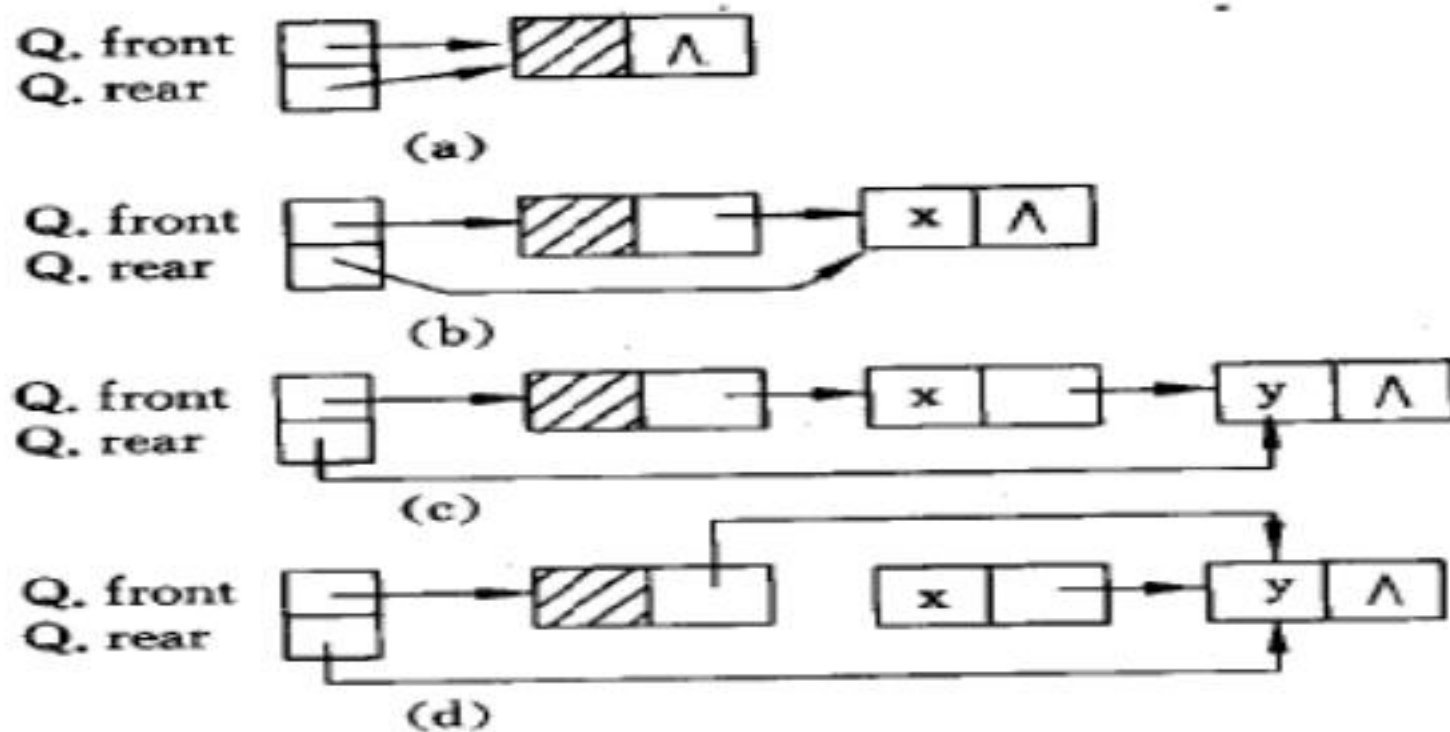


图 3.11 队列运算指针变化状况

- (a) 空队列； (b) 元素 x 入队列；
 (c) 元素 y 入队列； (d) 元素 x 出队列。



```
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;
```

```
}
```



```
Status DestroyQueue (LinkQueue &Q) {
```

```
// 销毁队列Q
```

```
while (Q.front) {
```

```
    Q.rear = Q.front->next;
```

```
    free(Q.front);
```

```
    Q.front=Q.rear;
```

```
}
```

```
return OK;
```

```
}
```

```
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;  
}
```

```
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;  
}
```

本章学习要点

1. 掌握栈和队列类型的特点，并能在相应的应用问题中正确选用它们。
2. 熟练掌握栈类型的两种实现方法，特别注意栈满和栈空的条件以及它们的描述方法。
3. 熟练掌握循环队列和链队列的基本操作实现算法，特别注意队满和队空的描述方法。
4. 理解递归算法执行过程中栈的状态变化过程。

作业

- 自习3.3栈与递归的实现、 3.5离散事件模拟
- 题集3.1, 3.7, 3.15, 3.19
- 题集3.28,3.32