第三章 堆栈和队列

- 3.1 堆栈
- 3.2 队列

3.1 堆 栈

3.1.1 堆栈的定义和主要操作

3.1.2 顺序栈

3.1.3 链式栈

3.1.4 顺序栈与链式栈的比较

3.1.5 堆栈的应用

1、堆栈的定义

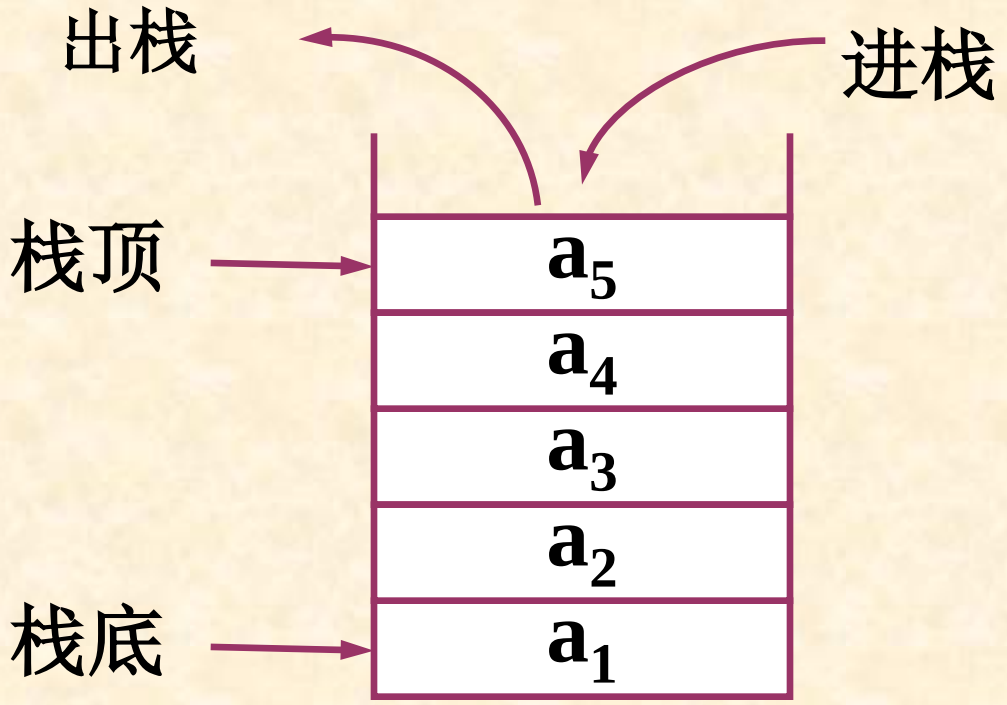
栈的定义：是一种操作受限的线性表，只允许在表的同一端进行插入和删除操作，且这些操作是按后进先出的原则进行的。

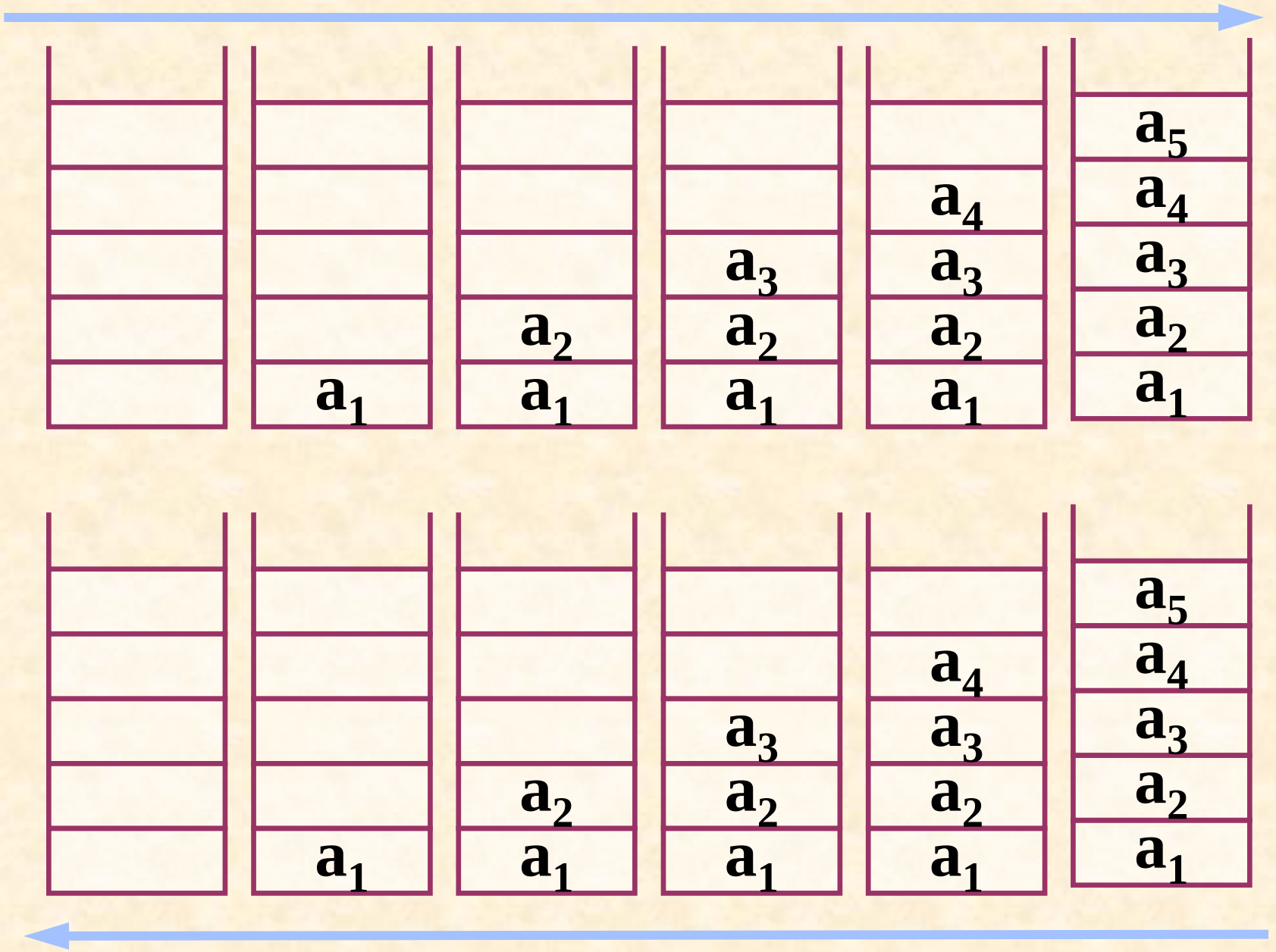
栈顶：进行插入、删除的一端；

栈底：另一端；

空栈：栈中无元素时。

[例] 线性表 $(a_1, a_2, \cdots, a_5)$，进栈出栈情况




a1
a2
a1
a3
a2
a1
a4
a3
a2
a1
a5
a4
a3
a2
a1
a1
a2
a1
a3
a2
a1
a4
a3
a2
a1
a5
a4
a3
a2
a1

●栈的后进先出性：先进后出（FILO）或后进先出（LIFO），可以对输入序列部分或全局求逆；凡符合后进先出性，都可应用栈，如十进制数与其它数制的转换、递归的实现、算数表达式求值等问题。

●栈的封闭性：除了栈顶元素外，其他元素不会被改变。因而，栈的封闭性非常好，使用起来非常安全。

堆栈的应用——括号匹配

- 高级语言程序设计中的各种括号应该匹配。例如：“(”与“)”匹配、“[”与“]”匹配、“{”与“}”匹配等。

```
if (!In(*c, OPSET)) {
        Dr[0]=*c;
        Dr[1]='\0';
      strcat(TempData,Dr);
      c++;
      if(In(*c,OPSET)) {
        Data=(float)atof(TempData);
        Push(OPND, Data);
        strcpy(TempData,"\0");
      }
}
```

练习

例：设栈的输入序列是（1，2，3，4），则（　）不可能是其出栈序列。

A. 1243
B. 2134
C. 1432
D. 4312

练习

例：设栈的输入序列是（1，2，3，4），则（D）不可能是其出栈序列。

A. 1243

B. 2134

C. 1432

D. 4312

2、堆栈的基本操作

（1）栈初始化

（2）进栈 Push

（3）出栈 Pop

（4）读取栈顶元素 Peek

（5）判栈空 StackEmpty

（6）判栈满 StackFull

（7）置空栈

栈的基本操作

- 教材P44-45关于栈的抽象描述

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。

ClearStack(&S)
初始条件：栈 S 已存在。
操作结果：将 S 清为空栈。

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。

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。

StackTraverse(S, visit())
初始条件：栈 S 已存在且非空。
操作结果：从栈底到栈顶依次对 S 的每
个数据元素调用函数 visit()。
一旦 visit()失败，则操作失效。

3.1 堆 栈

3.1.1 堆栈的定义和主要操作

3.1.2 顺序栈

3.1.3 链式栈

3.1.4 顺序栈与链式栈的比较

3.1.5 堆栈的应用

堆栈的顺序存储

使用数组存放栈元素，栈的规模必须小于或等于数组的规模，当栈的规模等于数组的规模时，就不能再向栈中插入元素。

存放堆栈元素的数组：

T stackArray [MaxStackSize];

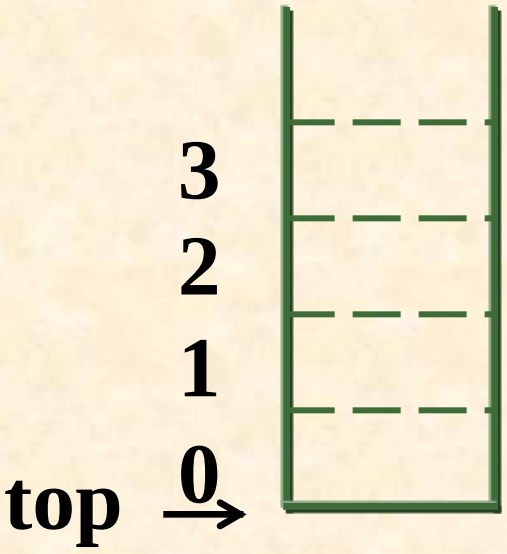
栈顶所在数组元素的下标：

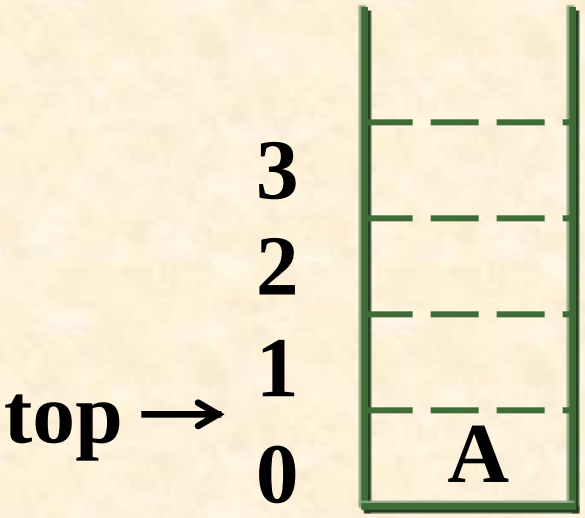
int top（栈顶指针top,指向实际栈顶后的空位置，初值为0）；

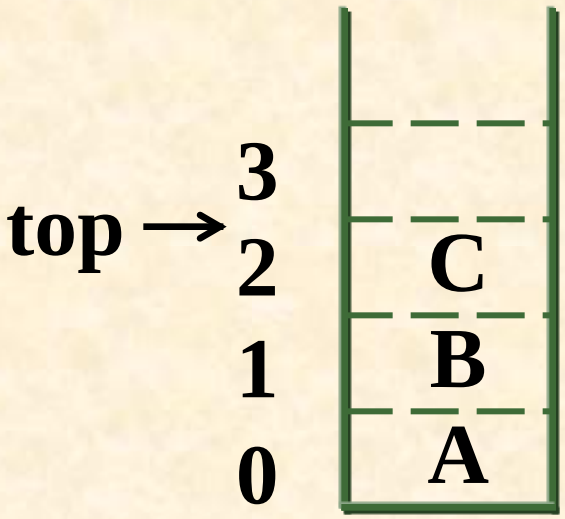
堆栈空： **top = 0**

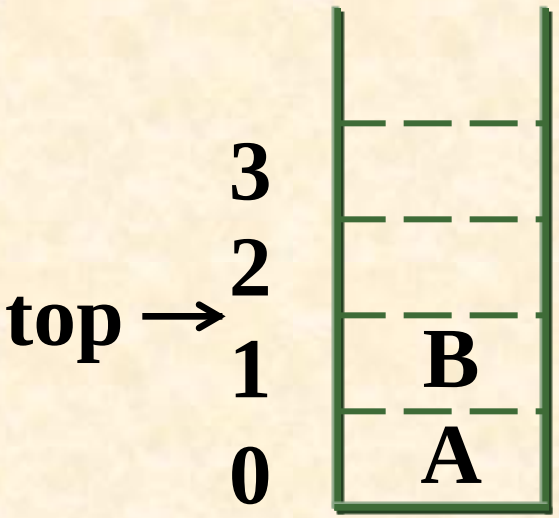
堆栈满： **top = MaxStackSize**

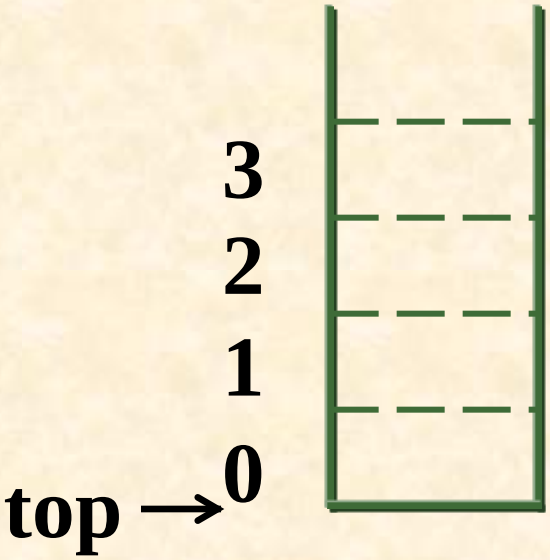
栈内变化情况





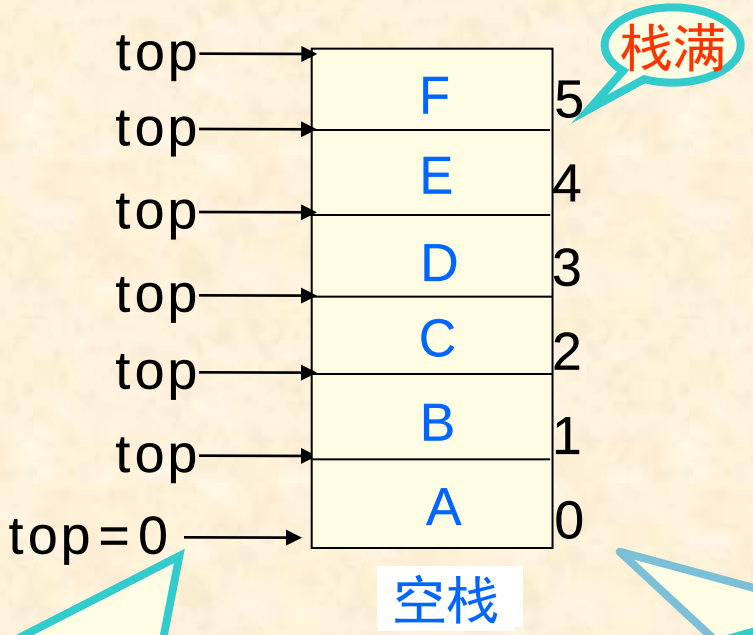


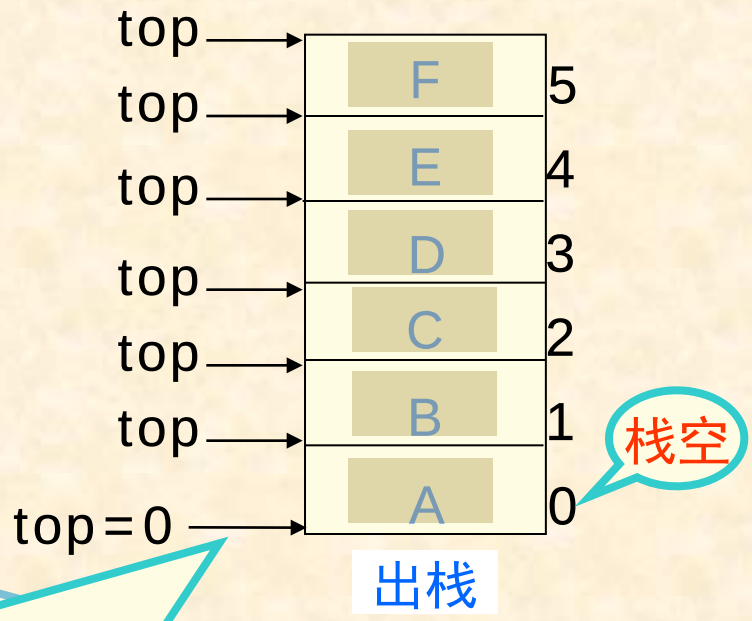




栈的顺序存储结构

- 顺序栈实现：一维数组s[M]
- 当堆栈中数据全部弹出后，在内存中的是什么？





栈顶指针top,指向实际栈顶后的空位置，初值为0

top=0栈空,此时出栈则下溢(underflow)
top=M栈满,此时入栈则上溢(overflow)

栈的顺序存储表示

```
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10
Typedef struct {
    SElemType *base; //在栈构造之前和销毁之后，base值为NULL
    SElemType *top; //栈顶指针
    int stacksize;
} SqStack;
```

栈的基本操作

- Status InitStack(SqStack &S);
- Status DestroyStack(SqStack &S);
- Status ClearStack(SqStack &S);
- Status StackEmpty(SqStack S);
- int StackLength(SqStack S);
- Status GetTop(SqStack S, SElemtype &e);
- Status Push(SqStack &S, SElemType e);
- Status Pop(SqStack &S, SElemType &e);
- Status StackTraverse(SqStack S, Status(*visit)());

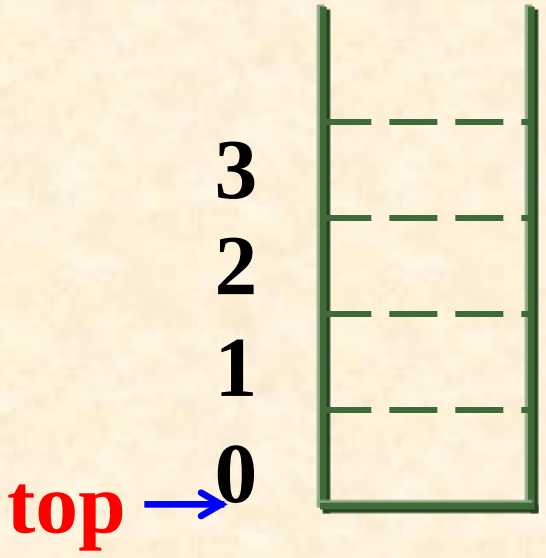
顺序栈算法

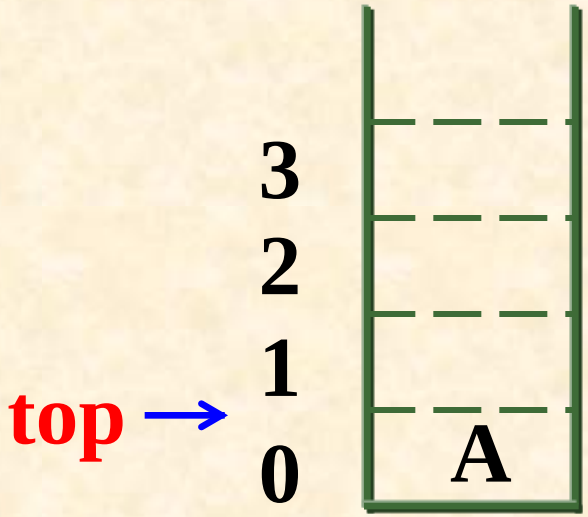
```
Status InitStack(SqStack &S) {
  S.base =
   (SElemType*)malloc(STACK_INIT_SIZE*sizeof(SElemType));
  if(!S.base) exit(OVERFLOW);
  S.top = S.base;
  S.stacksize = STACK_INIT_SIZE;
  return OK;
}
```

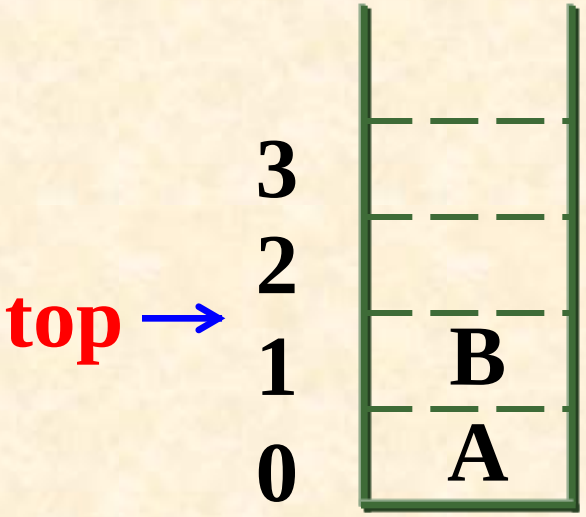
❑入栈算法

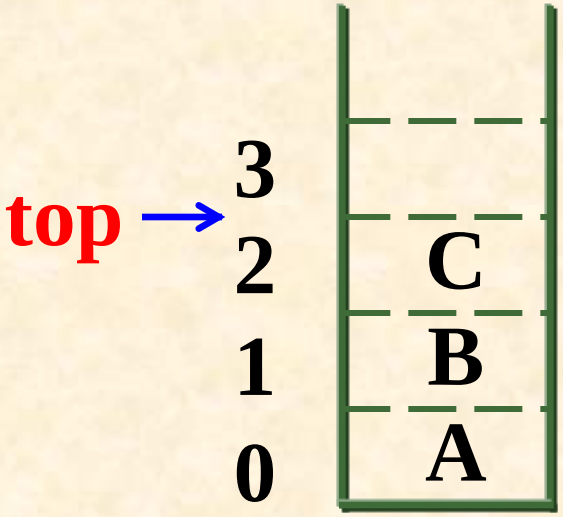
```
Status  push(SqStack &S, SElemType e) {
   if(S.top – S.base >= S.stacksize) {
      S.base = (SElemType *)realloc(S.base,
           (S.stacksize+STACKINCREMENT)*siezeof(SElemType));
      if(!S.base) exit(OVERFLOW);
      S.top = S.base + S.stacksize;
      S.stacksize += STACKINCREMENT;
   }
   *S.top++ = e;
   return OK;
}
```

堆栈变化情况









❑出栈算法

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

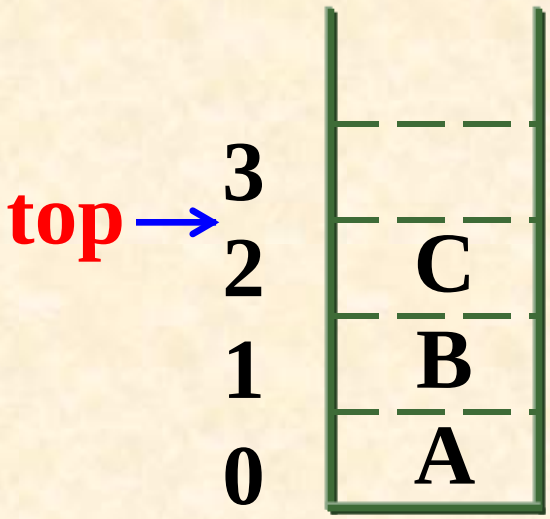
❑获取栈顶元素算法

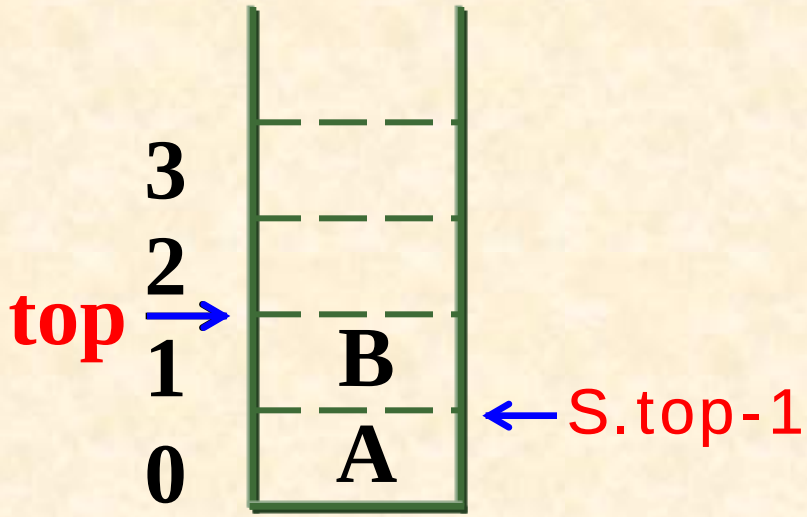
```
Status GetTop(SqStack S, SElemType &e)
［读取栈顶元素］
{
    if(S.top==S.base) return ERROR;
    e = *(S.top-1);
    return OK;
}
```

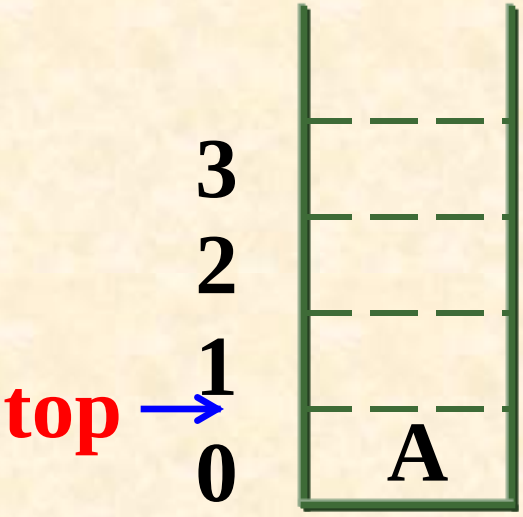
与Pop的区别是什么？ top ←? top - 1

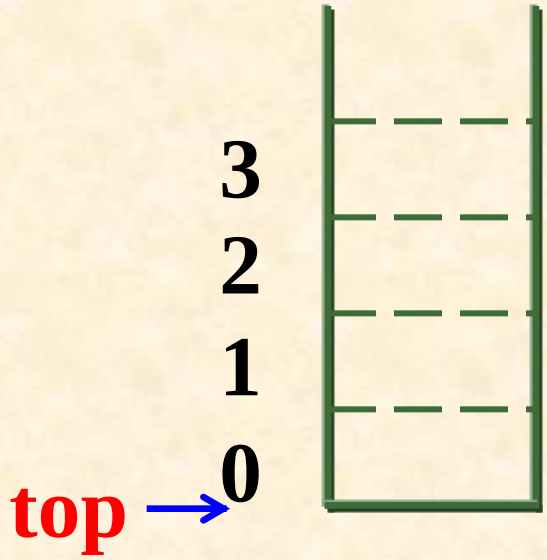
- ❑ 用数组实现的顺序栈的评价
  - ■ 栈的容量在使用前难以估计
  - ■ 操作简便

堆栈变化情况



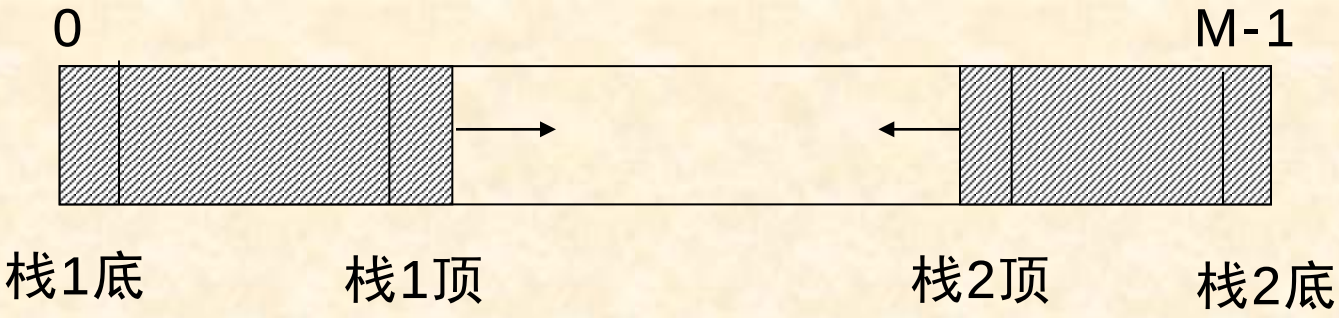






共享顺序栈

- 顺序栈所需容量在使用前难以估计，而且有些CPU内存有限，可供堆栈使用的空间有限，如单片机。常在一个程序中将两个堆栈使用的空间放在一起。



3.1 堆 栈

3.1.1 堆栈的定义和主要操作

3.1.2 顺序栈

3.1.3 链式栈

3.1.4 顺序栈与链式栈的比较

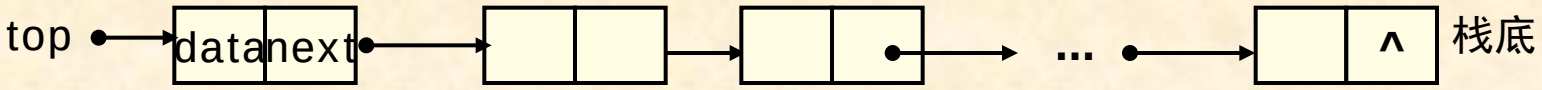
3.1.5 堆栈的应用

3.1.3 堆栈的链式存储

- 用单链表实现堆栈要为每个栈元素分配一个额外的指针空间。
- 考虑栈顶对应链表的表头还是表尾。因为堆栈主要操作（插入、删除、存取）的对象是栈顶元素，若栈顶对应表尾，则每次操作的时间复杂性为$O(n)$；若栈顶对应表头，则每个操作的时间复杂性是$O(1)$，显然，栈顶对应表头是合理的。
- 另外，链式栈中不需要哨位结点。

栈的链式存储结构

- 若一个程序中要使用多于两个的栈，则可以采用链表作为存储结构，这种存储结构通常称为链栈(linked-stack).



- 结点定义

```
typedef  struct tagLinkedStack
{
   int data;
   struct tagLinkedStack *next;
} LinkedStack;
```

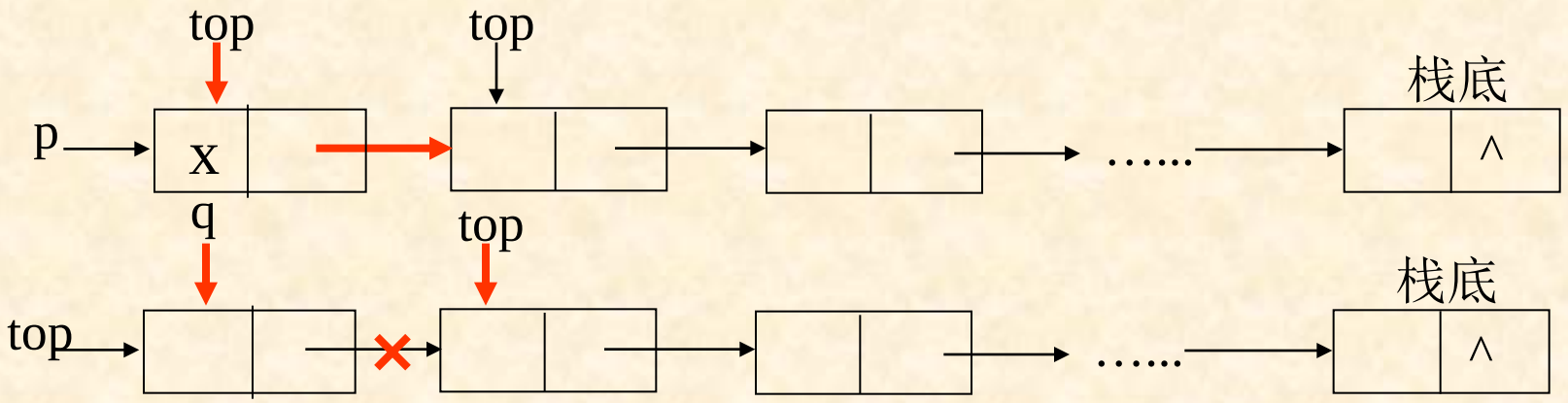
- 入栈算法
- 出栈算法

链栈算法

- 链栈通常都在链表头端操作。在尾端操作呢？

```
LinkedStack *LSPush(
   LinkedStack *top, int x)
{ LinkedStack *p;
   p =(LinkedStack *)
          malloc(sizeof(LinkedStack));
   p->data=x;
   p->next=top;
   top=p;
   return(p);
}
```

```
LinkedStack *LSPop(
   LinkedStack *top,int *x)
{  LinkedStack *q;
   if (top) {
      q=top;
      *x=top->data;
      top=top->next;
      free(q); }
   return(top);
}
```



3.1 堆 栈

3.1.1 堆栈的定义和主要操作

3.1.2 顺序栈

3.1.3 链式栈

3.1.4 顺序栈与链式栈的比较

3.1.5 堆栈的应用

顺序栈与链式栈的比较

- **在空间复杂性上，**顺序栈必须初始就申请固定的空间，当栈不满时，必然造成空间的浪费；链式栈所需空间是根据需要随时申请的，其代价是为每个元素提供空间以存储其next指针域。
- **在时间复杂性上，**对于针对栈顶的基本操作（压入、弹出和栈顶元素存取），顺序栈和链式栈的时间复杂性均为$O(1)$.

3.1 堆 栈

3.1.1 堆栈的定义和主要操作

3.1.2 顺序栈

3.1.3 链式栈

3.1.4 顺序栈与链式栈的比较

3.1.5 堆栈的应用

堆栈的应用——括号匹配

- 高级语言程序设计中的各种括号应该匹配，例如：“(” 与 “)”匹配、“[”与 “]” 匹配、“{”与 “}” 匹配等。字符串**{a=(b*c}+free( )]**中的括号就没有匹配上，因为串中第一个关括号 “}” 和最近的未匹配开括号 “(” 不匹配。

算法思想

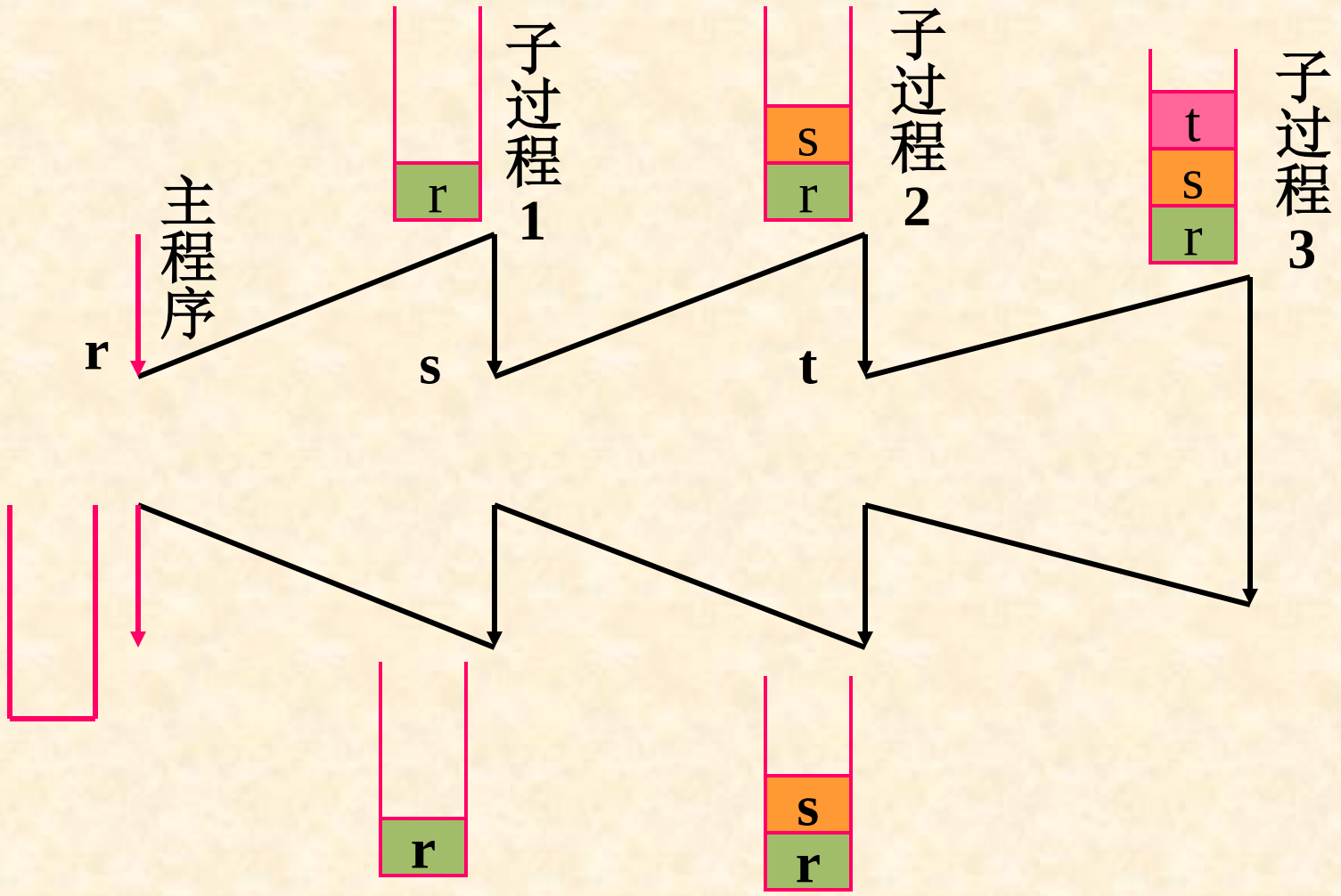
- 根据匹配规则：后遇到的开括号先匹配
- 采用栈存放开括号，当前闭括号和栈顶开括号匹配
- 考虑匹配失败的情况
- 考虑放松的匹配规则

堆栈的应用——回文游戏

- 顺读与逆读字符串一样(不含空格)
- 例如：
  - ini
  - madam im adam
- 算法
  - 读入字符串
  - 去掉空格（原串）
  - 压入栈
  - 原串字符与出栈字符依次比较
    - 若不等，非回文
    - 若直到栈空都相等，回文

堆栈的应用——过程的嵌套调用

- 子程序调用-调用时入栈，返回时出栈



堆栈的应用——数制转换

例：十进制数转换成八进制数：$(66)_{10}=(102)_8$

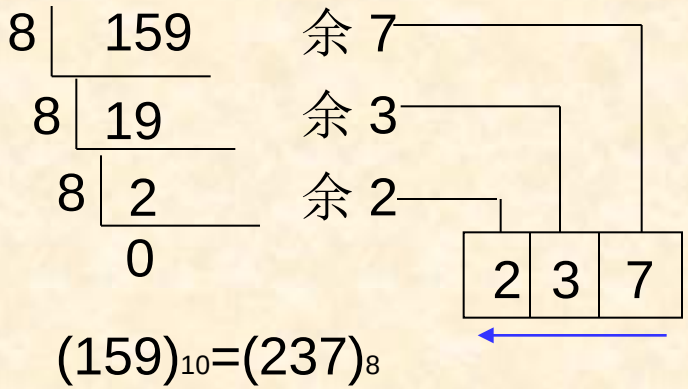
66/8=8 余 2

8/8=1 余 0

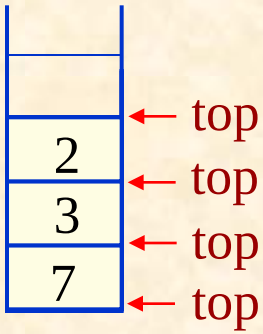
1/8=0 余 1

结果为余数的逆序:102。先求得的余数在写出结果时最后写出，最后求出的余数最先写出，符合栈的后进先出的性质，故可用栈来实现数制转换。

❑ 把十进制数159转换成八进制数



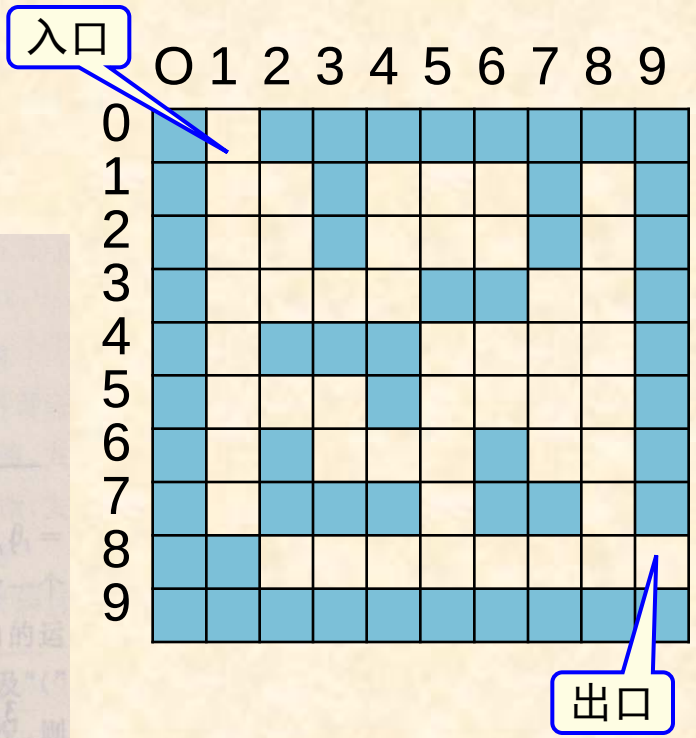
$(159)_{10}=(237)_8$




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

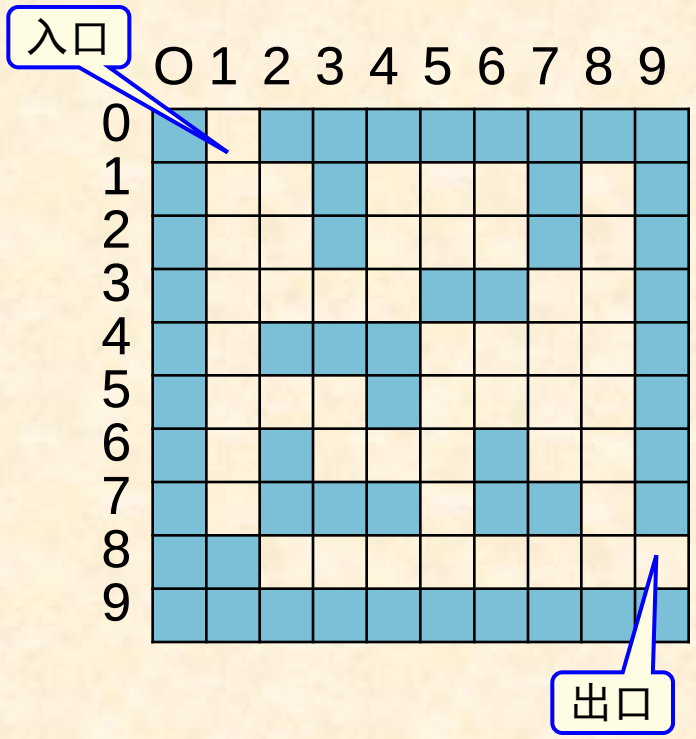
堆栈的应用——走迷宫

```
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,则结束;              // 求得路径存放在栈中
       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经探索,
       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);
```



```
typedef struct {
  int         ord;          // 通道块在路径上的"序号"
  PosType     seat;         // 通道块在迷宫中的"坐标位置"
  int         di;           // 从此通道块走向下一通道块的"方向"
}SElemType;                 // 栈的元素类型

Status MazePath ( MazeType maze, PosType start, PosType end ) {
  // 若迷宫 maze 中存在从入口 start 到出口 end 的通道,则求得一条存放在栈中(从栈底到栈
  // 顶),并返回 TRUE;否则返回 FALSE
    InitStack(S);  curpos = start;        // 设定"当前位置"为"入口位置"
    curstep = 1;                          // 探索第一步
    do {
      if (Pass (curpos)) {   // 当前位置可以通过,即是未曾走到过的通道块
          FootPrint (curpos);             // 留下足迹
          e = ( curstep, curpos, 1 );
          Push (S,e);                     // 加入路径
          if (curpos == end)  return (TRUE);  // 到达终点(出口)
          curpos = NextPos ( curpos, 1 );     // 下一位置是当前位置的东邻
          curstep++ ;                         // 探索下一步
      }//if
      else {  // 当前位置不能通过
       if (!StackEmpty(S)) {
        Pop (S,e);
        while (e.di == 4 && !StackEmpty(S)) {
         MarkPrint (e.seat);  Pop (S,e);      // 留下不能通过的标记,并退回一步
        }// while
        if (e.di<4) {
          e.di++ ;   Push ( S, e);            // 换下一个方向探索
          curpos = NextPos (e.seat e.di );    // 设定当前位置是该新方向上的相邻块
         }// if
       }// if
      }// else
    }while ( !StackEmpty(S) );
  return (FALSE);
}// MazePath
```

算法 3.3



堆栈的应用——表达式求值

- 运算规则：
  - 从左到右
  - 先乘除、后加减
  - 先括号内、后括号外
- 中缀表达式

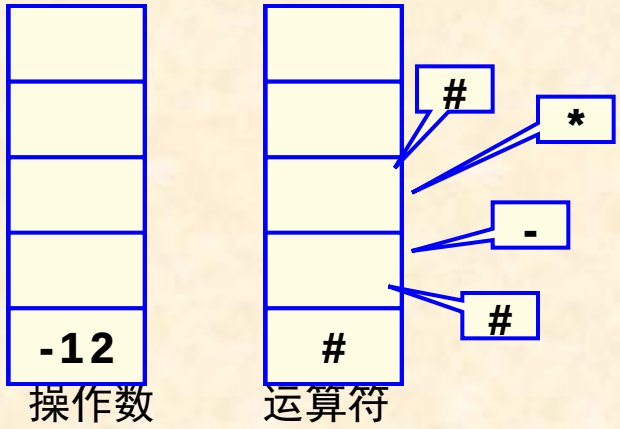
  a*b+c

  a+b*c

  a+(b*c+d)/e
- 操作数栈和运算符栈
- 例　计算 2+4-3*6
- 算法见教材P53.

算符间的优先关系

θ_1 \ θ_2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=



```
OperandType  EvaluateExpression() {
  // 算术表达式求值的算符优先算法。设 OPTR 和 OPND 分别为运算符栈和运算数栈，
  // OP 为运算符集合。
   InitStack (OPTR);   Push (OPTR,'#');
   initStack (OPND);   c = getchar();
   while (c!='#' || GetTop(OPTR)!='#') {
     if (!In(c, OP)){Push((OPND, c); c = getchar(); }     // 不是运算符则进栈
     else
       switch (Precede(GetTop(OPTR), c)) {
         case '<':   // 栈顶元素优先权低
                Push(OPTR, c);  c = getchar();
                break;
         case '=':   // 脱括号并接收下一字符
                Pop(OPTR, x);   c = getchar();
                break;

         case '>':   // 退栈并将运算结果入栈
                Pop(OPTR, theta);
                Pop(OPND, b);  Pop(OPND, a);
                Push(OPND, Operate(a, theta, b));
                break;
         }// switch
   }// while
   return GetTop(OPND);
}// EvaluateExpression
```

算法 3.4

堆栈的应用——递归程序设计

例 3-2 （n 阶 Hanoi 塔问题）假设有 3 个分别命名为 X、Y 和 Z 的塔座，在塔座 X 上插有 n 个直径大小各不相同、依小到大编号为 1,2,…，n 的圆盘（如图 3.5 所示）。现要求将 X 轴上的 n 个圆盘移至塔座 Z 上并仍按同样顺序叠排，圆盘移动时必须遵循下列规则：

（1）每次只能移动一个圆盘；

（2）圆盘可以插在 X、Y 和 Z 中的任一塔座上；

（3）任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。

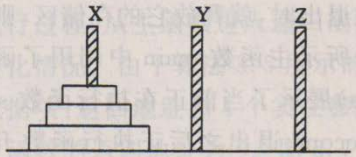


图 3.5 3 阶 Hanoi 塔问题的初始状态

```
void hanoi (int n, char x, char y, char z)
    // 将塔座 x 上按直径由小到大且自上而下编号为 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。
    // 搬动操作 move (x, n, z) 可定义为(c 是初值为 0 的全局变量,对搬动计数):
    // printf("%i. Move disk %i from  %c  to  %c\n", ++c, n, x, z);
{
    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
    }
}
```

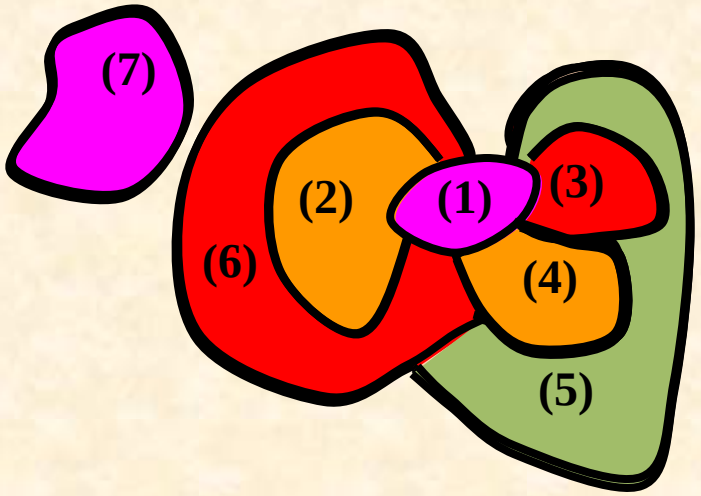
算法 3.5

堆栈的应用——地图着色问题

- 地图着色问题

邻接矩阵R[7][7]

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



1	2	3	4	5	6	7
1	2	3	2	4	3	1

已经证明，地图可以只用四种颜色着色

1# 紫色
2# 黄色
3# 红色
4# 绿色

第三章　、堆栈和队列

- 3.1 堆栈
- 3.2 队列

3.2 队 列

3.2.1 队列的定义和主要操作

3.2.2 顺序队列

3.2.3 链式队列

3.2.4 顺序队列与链式队列的比较

3.2.5 队列与堆栈的扩展

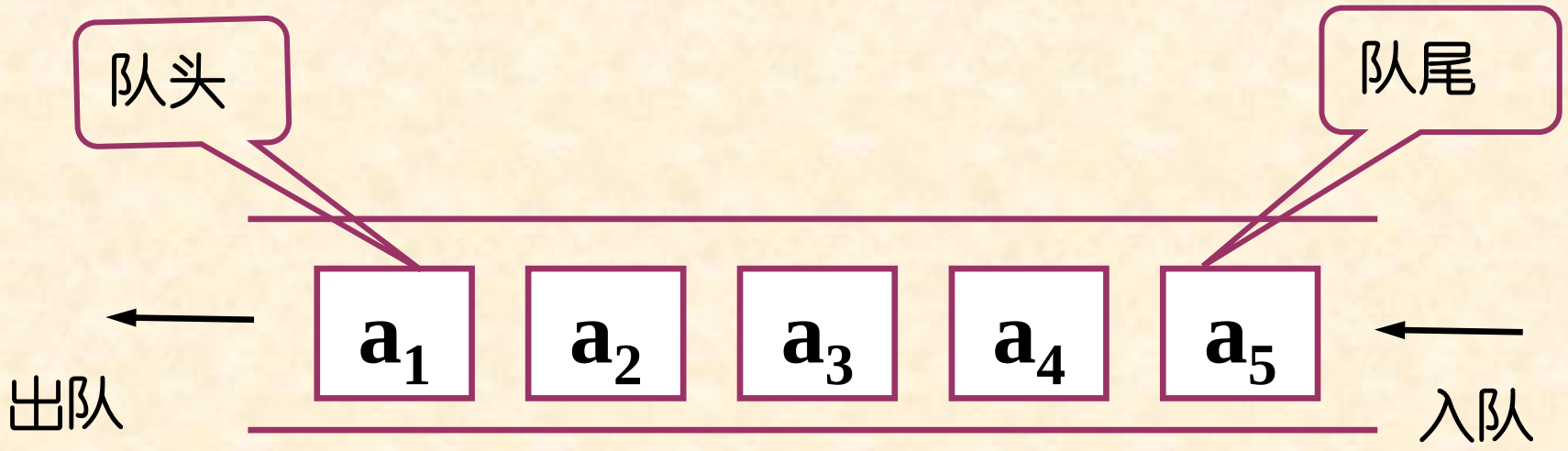
1、队列的定义

队列的定义：是一个操作受限的线性表，对于它的所有插入都在表的一端进行，所有的删除（以至几乎所有的存取）都在表的另一端进行，且这些操作又都是按着先进先出（FIFO）的原则进行的。

能进行删除的一端称为**队头**（front）；

能进行插入的一端称为**队尾**（rear）；

没有元素的队列称为**空队列**。



●**队列的先进先出性：**可以对输入序列起缓冲作用；凡符合先进先出性，都可应用队列，如操作系统中**作业调度**、**树的层次遍历**、**图的广度优先搜索**等问题。

●**队列的封闭性：**和栈类似，队列的封闭性也非常好，使用起来非常安全。

2、队列的基本操作

(1) 队列初始化
(2) 入队 (插入)
(3) 出队 (删除)
(4) 读取队首元素
(5) 判断队列是否空
(6) 确定队列中元素个数
(7) 置空队列

队列的抽象描述

ADT Queue {

数据对象：D = { a_i | $a_i \in$ ElemSet, i = 1, 2, …, n, n ≥ 0 }

数据关系：R1 = { <a_{i-1}, a_i> | 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：

InitQueue(&Q)

操作结果：构造一个空队列 Q。

DestroyQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：队列 Q 被销毁，不再存在。

ClearQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：将 Q 清为空队列。

QueueEmpty(Q)

初始条件：队列 Q 已存在。

操作结果：若 Q 为空队列，则返回 TRUE，否则 FALSE。

QueueLength(Q)

初始条件：队列 Q 已存在。

操作结果：返回 Q 的元素个数，即队列的长度。

GetHead(Q, &e)

初始条件：Q 为非空队列。

操作结果：用 e 返回 Q 的队头元素。

EnQueue(&Q, e)

初始条件：队列 Q 已存在。

操作结果：插入元素 e 为 Q 的新的队尾元素。

DeQueue(&Q, &e)

初始条件：Q 为非空队列。

操作结果：删除 Q 的队头元素，并用 e 返回其值。

QueueTraverse(Q, visit())

初始条件：Q 已存在且非空。

操作结果：从队头到队尾，依次对 Q 的每个数据元素调用函数 visit()。一旦 visit() 失败，则操作失败。

} ADT Queue

3.2 队 列

3.2.1 队列的定义和主要操作

3.2.2 顺序队列

3.2.3 链式队列

3.2.4 顺序队列与链式队列的比较

3.2.5 队列与堆栈的扩展

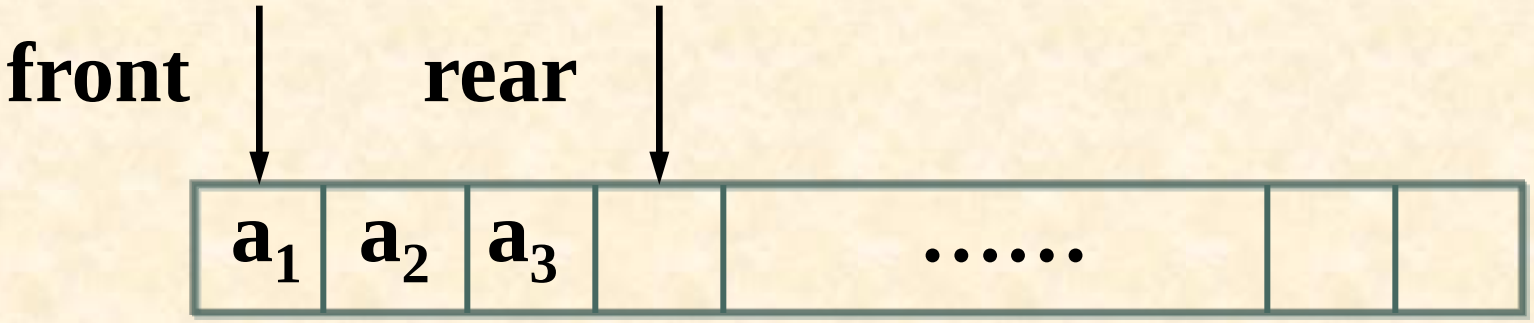
队列的顺序存储

- 存放队列元素的数组：

  T qlist[MaxQSize]

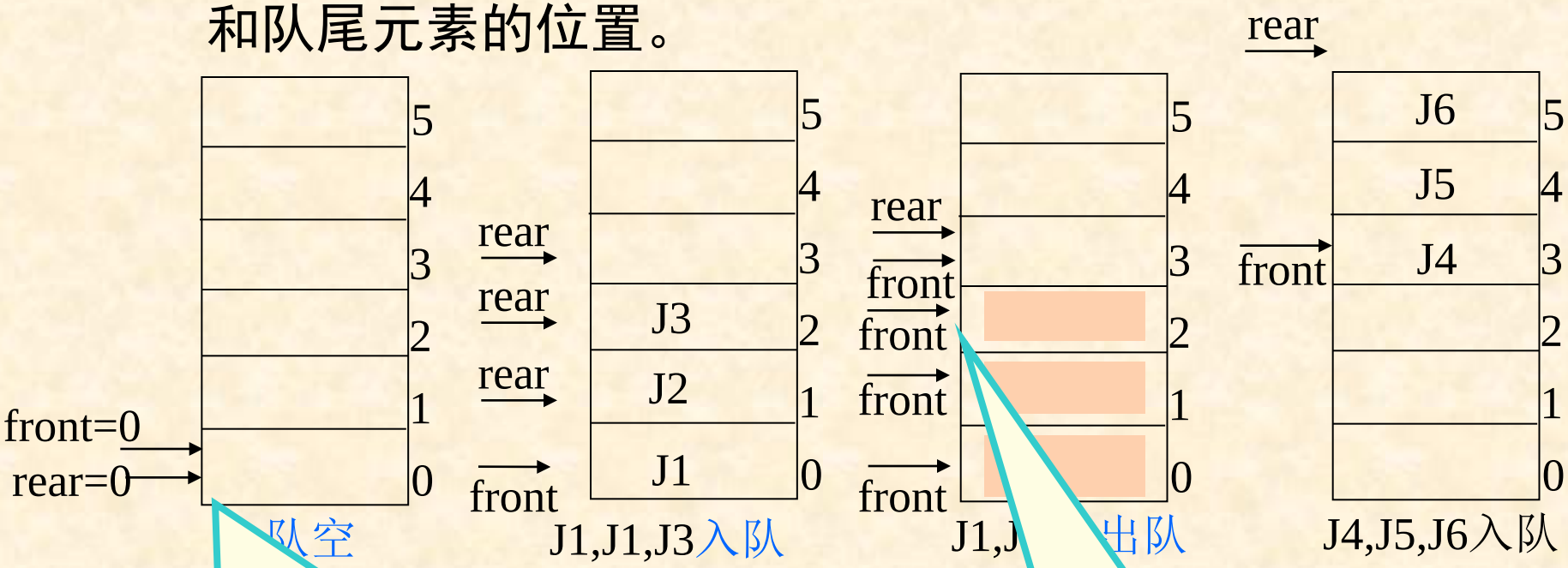
  front 队首元素的数组下标

  rear (要入队元素的下标) 队尾元素的下标加1



队列的顺序表示

- 队列作为线性表，也可以用顺序存储结构存储。除了可以用一组地址连续的存储单元依次存放从队头到队尾的元素之外，还需附设两个指针front和rear分别指向队头元素和队尾元素的位置。



设两个指针front,rear,约定：
rear指示队尾元素的下一个位置；
front指示队头元素
初值front=rear=0

空队列条件：front==rear
入队列：sq[rear++]=x;
出队列：x=sq[front++];

顺序队列存在的问题

- 设数组维数为M，则-
  - 当front=0,rear=M时，当再次有元素入队时发生溢出——真溢出
  - 当front≠0,rear=M时，当再次有元素入队时发生溢出——假溢出
- 解决方案

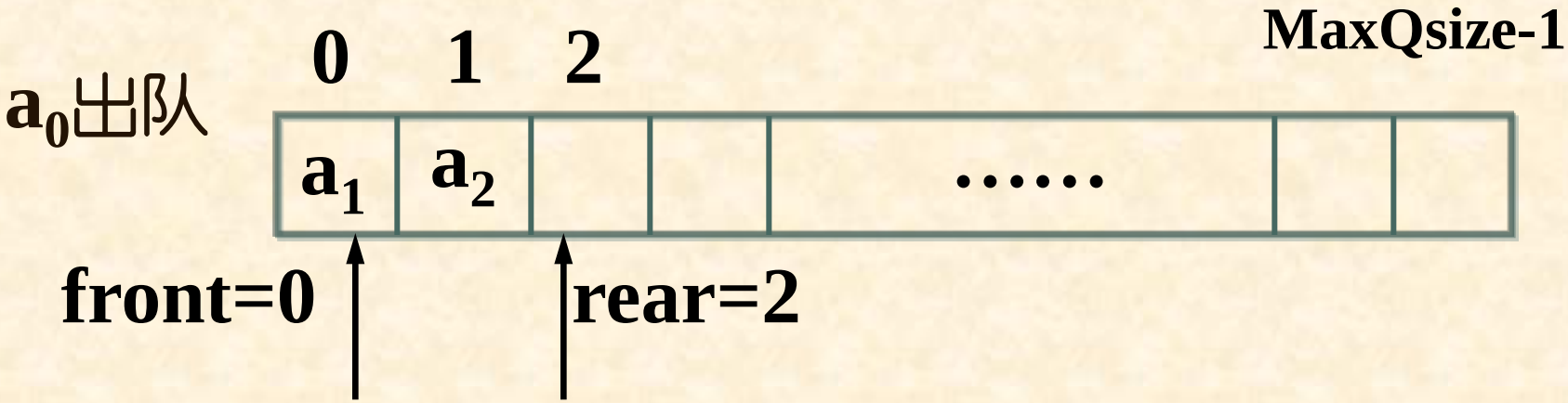
1. 队首固定，每次出队剩余元素向下移动——浪费时间

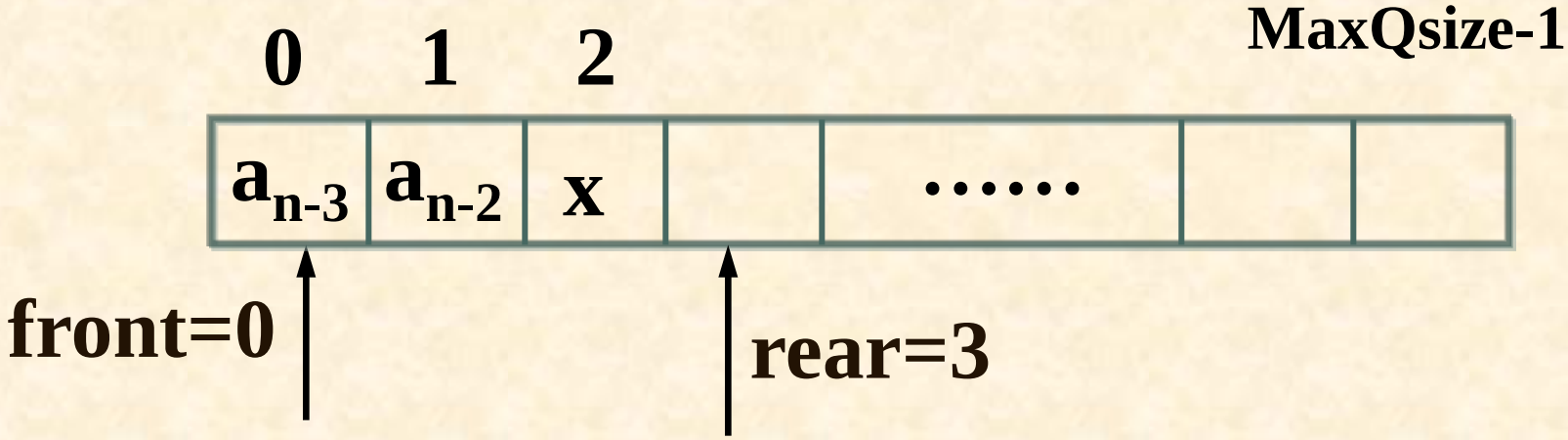
2. 循环队列

✧基本思想：把队列设想成环形，让sq[0]接在sq[M-1]之后，若rear==M，则令rear=0;

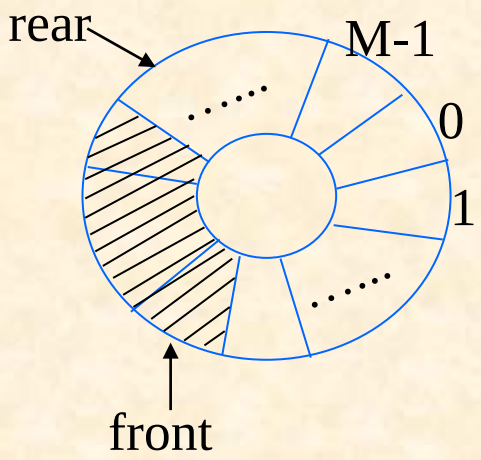
✧重复利用已经用过的空间

1、队首固定，每次出队剩余元素向前移动，front总等于0



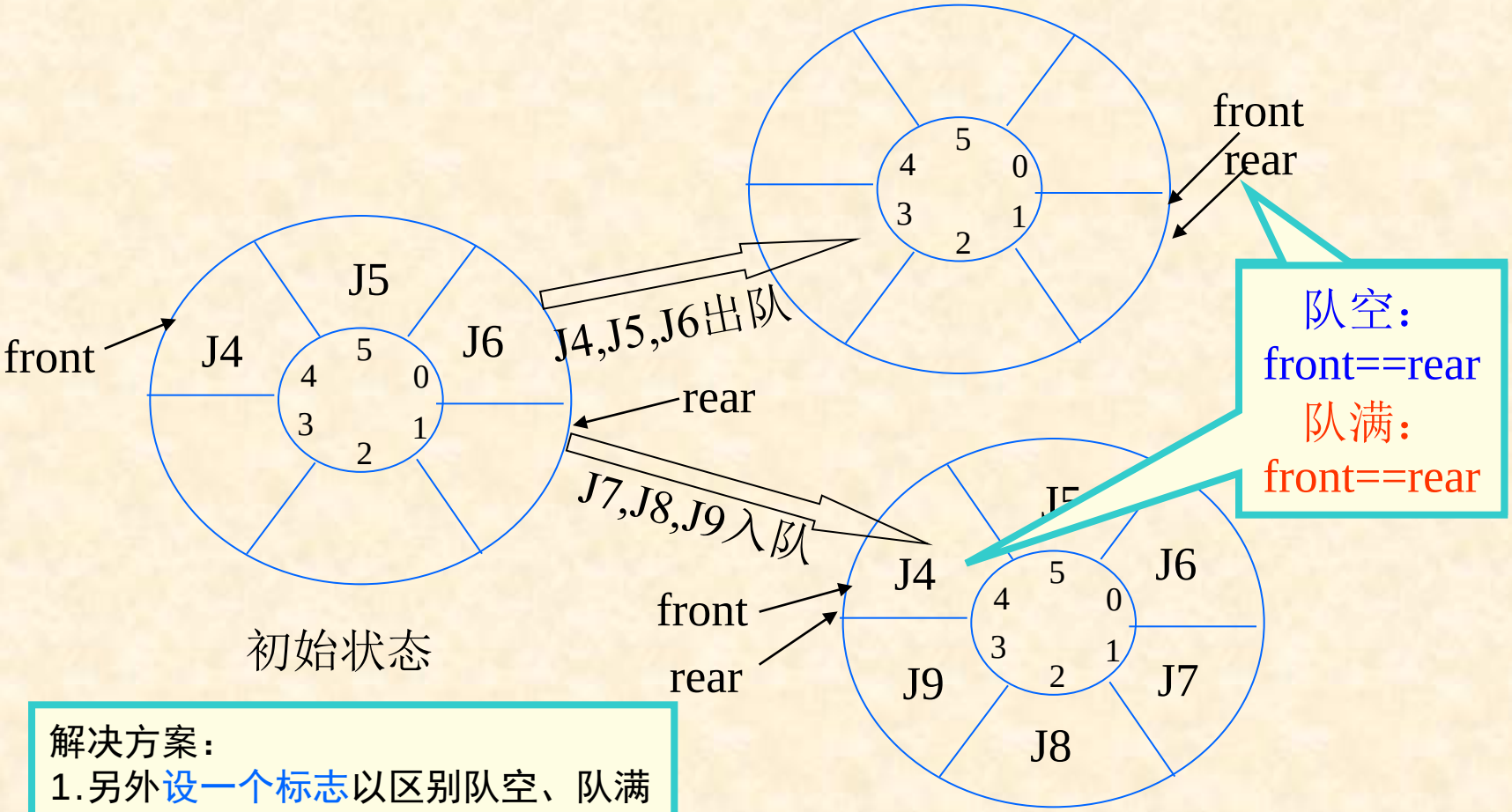


2、循环队列



- 基本思想：把队列sq设想成环形，让sq[0]接在sq[M-1]之后，若rear+1==M, 则令rear=0;
- 实现：利用“模”运算
  - 入队： sq[rear]=x; rear=(rear+1)%M;
  - 出队： x=sq[front]; front=(front+1)%M;
- 队满、队空判定条件

循环队列插入与删除



解决方案：

1.另外设一个标志以区别队空、队满

2.少用一个元素空间：

队空：front==rear

队满：(rear+1)%M==front

采用环状模型来实现队列，各数据成员的意义如下：

- front指定队首位置，删除一个元素就将front顺时针移动一位；
- rear指向元素要插入的位置，插入一个元素就将rear顺时针移动一位；
- count存放队列中元素的个数，当count等于MaxQSize时，不可再向队列中插入元素。

队空：count=0
队满：count= MaxQSize

循环队列的顺序存储

```
#define MAXQSIZE 100
Typedef struct {
   QElemType *base;
   int front;
   int rear;
}SqQueue;
Status  InitQueue(SqQueue &Q) {
   Q.base = (QElemType *)malloc(MAXQSIZE*sizeof(QElemType));
   if(!Q.base) exit(OVERFLOW);
   Q.front = Q.rear = 0;
   return OK;
}
```

循环队列插入与删除

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

3.2 队 列

3.2.1 队列的定义和主要操作

3.2.2 顺序队列

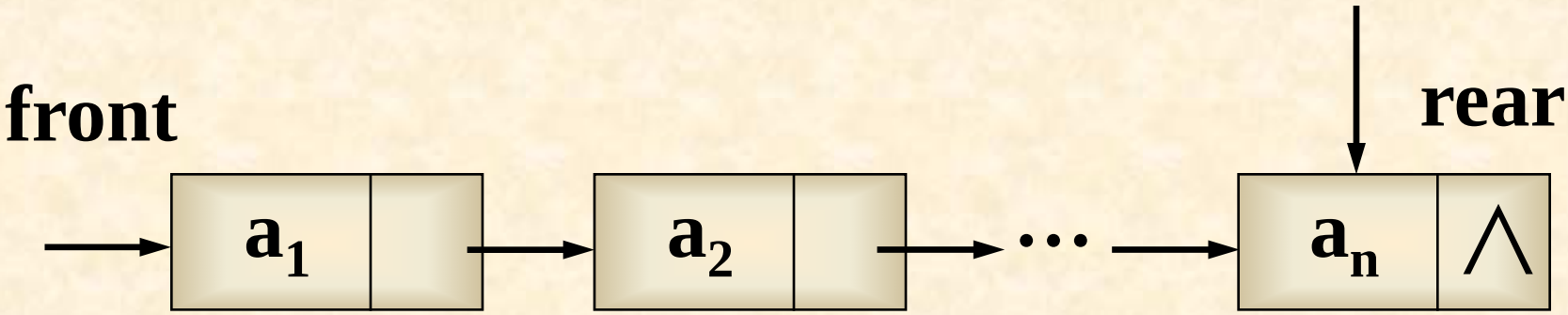
3.2.3 链式队列

3.2.4 顺序队列与链式队列的比较

3.2.5 队列与堆栈的扩展

队列的链接存储

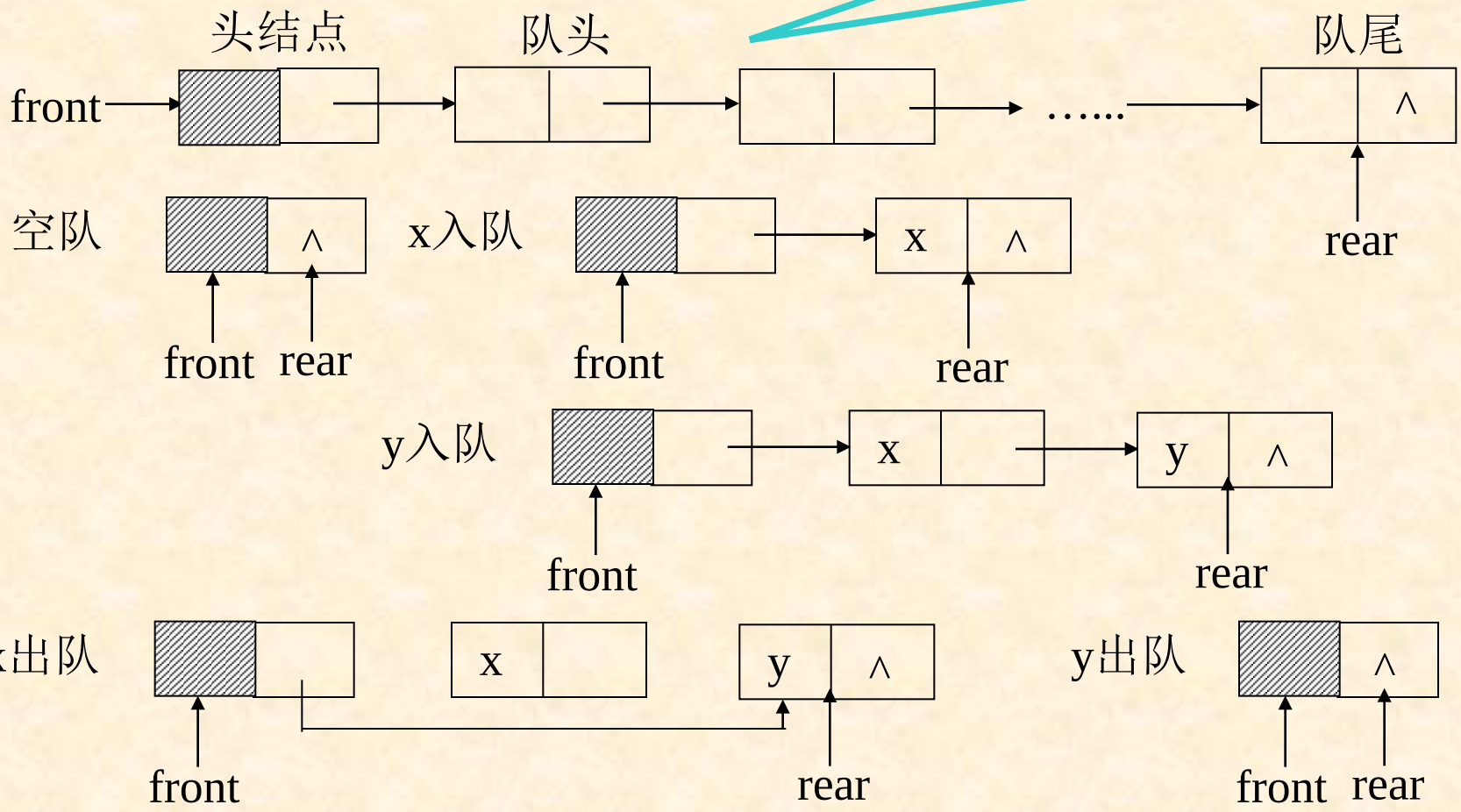
链式队列的结构： $(a_1, a_2, \ldots, a_n)$



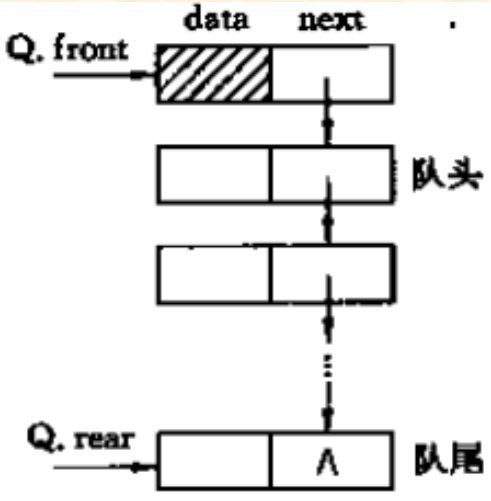
链队列

- 用链表表示的队列称链队列。

设队首、队尾指针front和rear，front指向头结点，rear指向队尾



链队列基本操作




```
// 队列的链式存储结构
typedef struct QNode {
    QElemType       data;
    struct QNode    * next;
}QNode,  * QueuePtr;
typedef struct {
    QueuePtr   front; // 队头指针
    QueuePtr   rear;  // 队尾指针
}LinkQueue;
```

```
// 基本操作
Status InitQueue (LinkQueue &Q)
    // 构造一个空队列 Q
Status DestroyQueue (LinkQueue &Q)
    // 销毁队列 Q,Q 不再存在
Status ClearQueue (LinkQueue &Q)
    // 将 Q 清为空队列
Status QueueEmpty (LinkQueue Q)
    // 若队列 Q 为空队列,则返回 TRUE,
       否则返回 FALSE
int QueueLength (LinkQueue Q)
    // 返回 Q 的元素个数,即为队列的长度
Status GetHead (LinkQueue Q, QElemType &e)
    // 若队列不空,则用 e 返回 Q 的队头元素,
    // 并返回 OK;否则返回 ERROR
Status EnQueue (LinkQueue &Q, QElemType e)
    // 插入元素 e 为 Q 的新的队尾元素
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
Status QueueTraverse(LinkQueue Q, visit())
    // 从队头到队尾依次对队列 Q 中每个元素
    //  调用函数 visit()。
    // 一旦 visit 失败,则操作失败。
```

链队列基本操作算法实现

```
//基本操作的算法实现
Status InitQueue (LinkQueue &Q)
{   // 构造一个空队列 Q
    Q.front = Q.rear =
     (QueuePtr)malloc(sizeof(QNode));
    //存储分配失败
    if (! Q.front) exit (OVERFLOW);
    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));
    //存储分配失败
    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)
{   // 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;
    free (p);
    return OK;
}
```

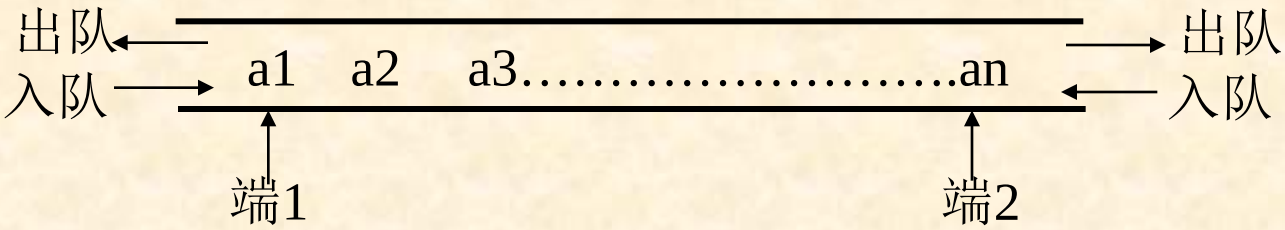
链队列基本操作算法实现

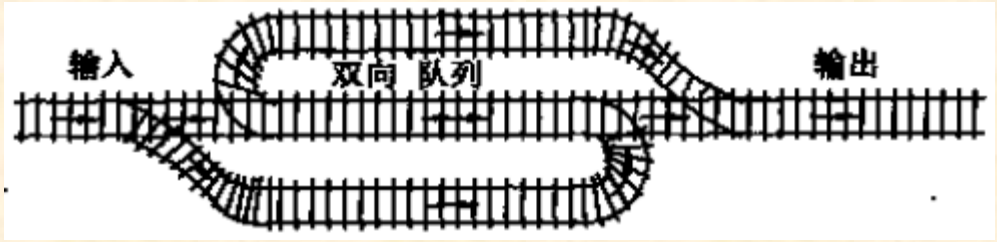
```
//基本操作的算法实现
Status InitQueue (LinkQueue &Q)
{   // 构造一个空队列 Q
    Q.front = Q.rear =
     (QueuePtr)malloc(sizeof(QNode));
    //存储分配失败
    if (! Q.front) exit (OVERFLOW);
    Q.front->next = NULL;
    return OK;
}

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));
    //存储分配失败
    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)
{   // 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;
}
```

双端队列

- 定义：限定插入和删除操作可在表的两端进行的线性表。





3.2 队 列

3.2.1 队列的定义和主要操作

3.2.2 顺序队列

3.2.3 链式队列

3.2.4 顺序队列与链式队列的比较

3.2.5 队列与堆栈的扩展

顺序队列与链式队列的比较

- 在空间复杂性上，顺序队列必须初始就申请固定的空间，当队列不满时，必然造成空间的浪费；链式队列所需空间是根据需要随时申请的，其代价是为每个元素提供空间以存储其next指针域。
- 在时间复杂性上，对于队列的基本操作（入队、出队和存取），顺序队列和链式队列的时间复杂性均为 $O(1)$.

顺序队列与链式队列的比较

- 顺序队列有固定的存储空间，不适于队列大小无法估计的情况，而且当队列很大时，顺序队列的空间效率低下，但链式队列却可满足这些应用环境。
- 对于那些大致能预测队列统计特性的应用环境，可估计缓冲队列长度，为顺序队列选择比较恰当的存储空间。

3.2 队 列

3.2.1 队列的定义和主要操作

3.2.2 顺序队列

3.2.3 链式队列

3.2.4 顺序队列与链式队列的比较

3.2.5 队列与堆栈的扩展

队列与堆栈的扩展

- 基于栈和队列还可设计一些变种的栈或队列，虽然它们的应用不够广泛，但在一些特定场合却很有应用价值。

——**双端队列：** 插入和删除都可在且只能在线性表的两端进行. 它好像一个特别的书架，只能从左右两边的端点取书和存书。

——**双栈：** 两个底部相连的栈。

——**超队列：** 是一种被限制的双端队列，删除操作限制在一端进行，而插入操作允许在两端进行。

——**超栈：** 是一种插入受限的双端队列，即插入限制在一端而删除仍允许在两端进行。当栈溢出时，它允许将栈中保存最久的底部元素删除。