福州大学至诚学院
FUZHOU UNIVERSITY ZHICHENG COLLEGE

高级语言程序设计（C语言与数据结构）

杨雄

83789047@qq.com

复　习

1. 线性表的定义

- 线性表是具有相同数据类型的n(n≥0)个数据元素的有限序列（其中n为表长，当n=0时该线性表是一个空表）
- 一般表示：
  - 线性表L：(a_1, a_2 ,..., a_{i-1}, a_i, a_{i+1}, ..., a_n)
  - ★特点：

    a_1是唯一的“第一个”数据元素

    a_n是唯一的“最后一个”数据元素

    除第一个元素外，每个元素有且仅有一个直接前驱

    除最后一个元素外，每个元素有且仅有一个直接后继

2. 线性表的特点

- 表中的数据元素的个数有限
- 表中元素具有逻辑上的顺序性，在序列中各个元素排序有**先后次序**
- 表中元素的**数据类型都相同**。这意味着每一个数据元素占有相同数量的存储空间
- 表中数据元素具有抽象性。**仅讨论数据元素之间的逻辑关系**，不考虑数据元素究竟表示什么内容

3. 线性表的操作

- 数据结构的基本操作就是指其最核心、最基本的操作，其他较复杂的操作可以通过调用基本操作来实现
- InitList(&L) -- 初始化表 – 构造一个空的线性表
- Length(L) – 求表长度 – 返回线性表的长度，即L中数据元素的个数
- ★ LocateElem(L, e) – 按值查找操作—在表L中查找具有给定关键字值的元素
- ★ GetElem(L, i, &e) – 按位查找操作 – 获取表L中第i个位置的元素的值
- ★ ListInsert(&L, i, e) – 插入操作 – 在表L中第i个位置上插入元素e
- ★ ListDelete(&L, i, &e) – 删除操作 删除表L中第i个位置的元素，并用e返回删除元素的值

4. 线性表的顺序存储

- 线性表的顺序存储又称顺序表
- ★它是用一组地址连续的存储单元，依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻
- 特点：
  1. 表中元素的逻辑顺序与其物理顺序相同
  2. ★可以进行随机访问，即通过首地址和元素序号可以在O(1)时间内找到指定的元素
  3. 存储密度高，每个节点只存储数据元素
  4. ★逻辑上相信的元素物理上也相邻，所以插入和删除操作需要移动大量元素O(n)

4. 线性表的链式存储

- 特性：
  1. 链式存储线性表时，不需要使用地址连接单元，即它不要求逻辑上的相邻的两个元素在物理位子上也相邻。
  2. 它是通过“链”建立起数据元素之间的逻辑关系。
  3. ★对线性表插入、删除不需要移动元素，只需要修改指针。
- 优缺点：
  1. ★优点：新增和删除元素很方便 ，不需要移动元素
  2. ★缺点：不方便随机访问元素
- ★单链表：为了建立起数据元素之间的线性关系，对每个链表节点，除了存放元素自身的信息之外，还需要存放一个指向其后继的指针。

3.栈和队列

01 栈

02 栈的应用举例

03 队列

学习目的

理解栈的定义、特点，学习它的各种组织方式及算法；

掌握它的空和满的判断条件；

掌握队列的数据结构和链队列的相关操作；

掌握循环队列的相关内容；

熟悉栈的简单应用

Part.1

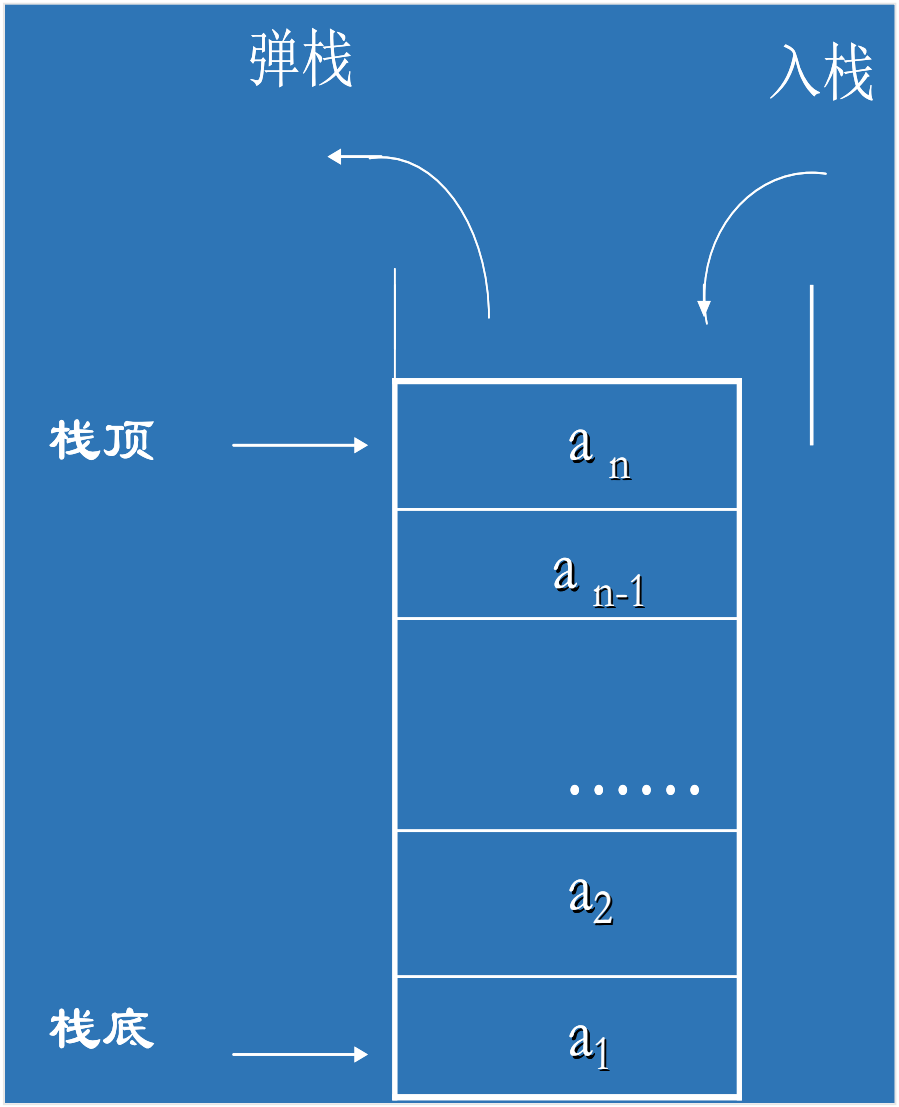
3.1 栈

➢ 栈的概念

- 栈的定义：栈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。

3.1 栈

- 假设栈S=(a_1，a_2，a_3，...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，...a_n的次序进栈，退栈的第一个元素应为栈顶元素，即栈的修改是按后进先出的原则进行的。因此栈又称为后进先出表(LIFO)。



3.1 栈

➢栈的抽象数据类型

ADT Stack {

数据对象：D = {a_i | a_i属于Elemset,　　(i=1,2,…,n, n≥0)}

数据关系：R_1 = {＜a_{i-1},a_i＞|a_{i-1},a_i属于D (i=2,3,…,n)}　约定a_n为栈顶, a_1为栈底

基本操作：

InitStack(&S);　　DestroyStack(&S);

ClearStack(&S);　　StackEmpty(S);

StackLength(S) ;　　**GetTop(S,&e);**

Push(&S,e);　　**Pop(&S,&e);**

}ADT Stack

➢ 基本操作(一)

- InitStack(&S)
  - 操作结果:构造一个空的栈S。
- DestroyStack(&S)
  - 初始条件：栈S已经存在。
  - 操作结果：销毁栈S。
- ClearStack(&S)
  - 初始条件：栈S已经存在。
  - 操作结果：将栈S重置为空栈。

➢ 基本操作(二)

- StackEmpty(S)
  - 初始条件: 栈S已经存在。
  - 操作结果: 若栈S为空栈，则返回TURE;否则返回FALSE。
- StackLength(S)
  - 初始条件: 栈S已经存在。
  - 操作结果: 返回栈S中的数据元素个数。
- GetTop(S,&e)
  - 初始条件: 栈S已经存在且非空。
  - 操作结果: 用e返回栈S中栈顶元素的值。

➢ 基本操作(三)

- Push(&S,e)
  - 初始条件: 栈S已经存在。
  - 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已经存在且非空。
  - 操作结果: 删除S的栈顶元素并用e返回其值。

➢栈的顺序表示与实现---(顺序栈)

栈的顺序存储结构简称为顺序栈，和线性表相类似，用一维数组来存储栈。根据数组是否可以根据需要增大，又可分为静态顺序栈和动态顺序栈。

- 静态顺序栈实现简单，但不能根据需要增大栈的存储空间；
- 动态顺序栈可以根据需要增大栈的存储空间，但实现稍为复杂。

3.1 栈

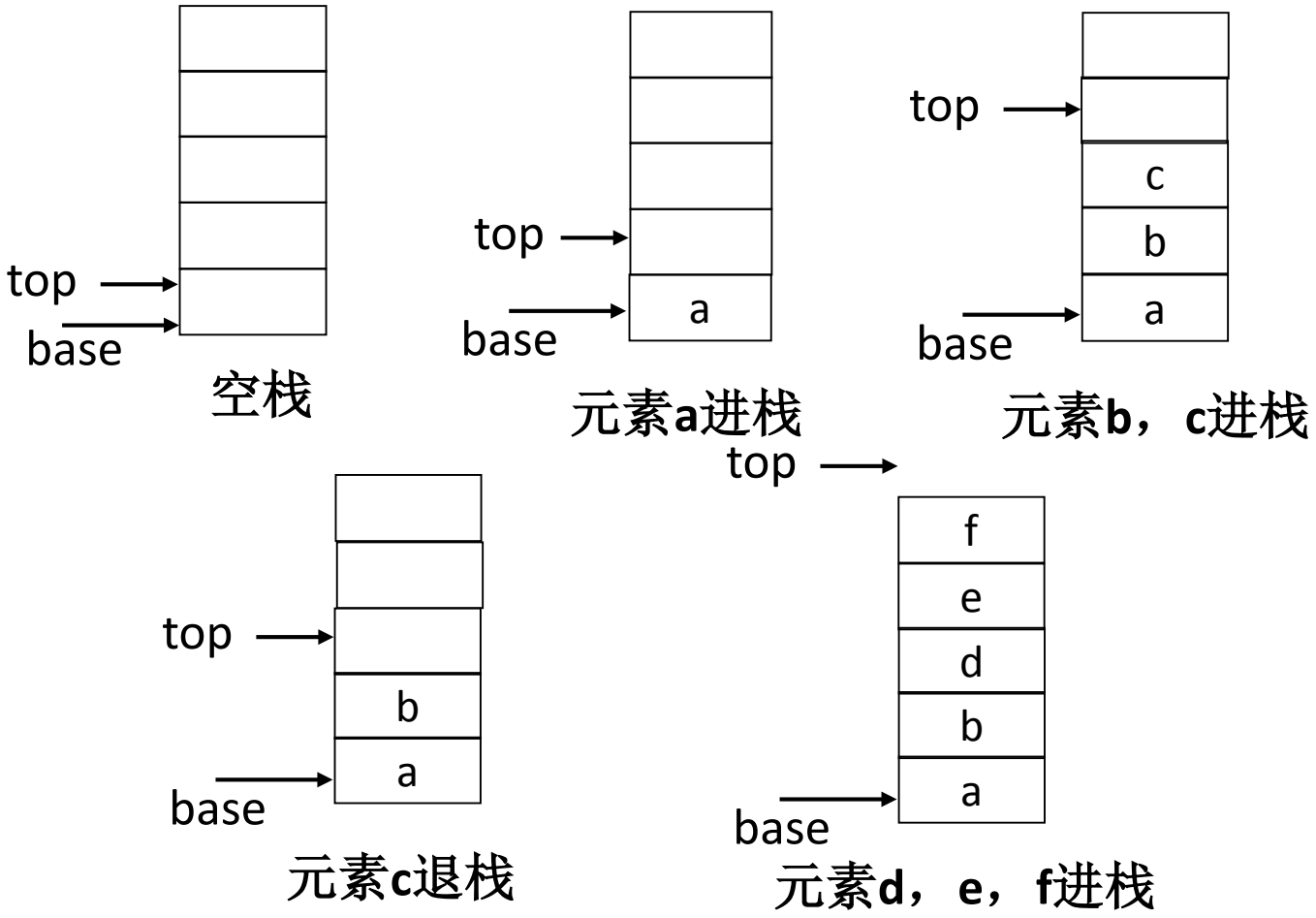
采用动态一维数组来存储栈。所谓动态，指的是栈的大小可以根据需要增加。

- 用base表示栈底指针，栈底固定不变的；栈顶则随着进栈和退栈操作而变化。用top(称为栈顶指针)指示当前栈顶位置。
- 用top=base作为栈空的标记，每次top指向栈顶数组中的下一个存储位置。

◆ 结点进栈：首先将数据元素保存到栈顶(top所指的当前位置)，然后执行top加1，使top指向栈顶的下一个存储位置;

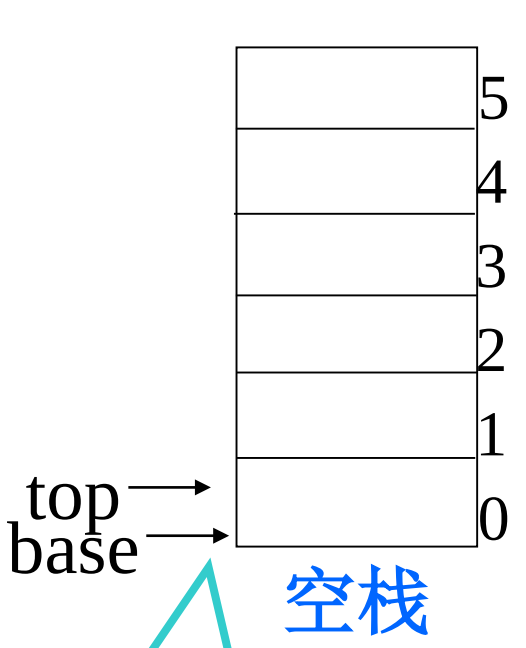
◆ 结点出栈： 首先执行top减1，使top指向栈顶元素的存储位置，然后将栈顶元素取出。

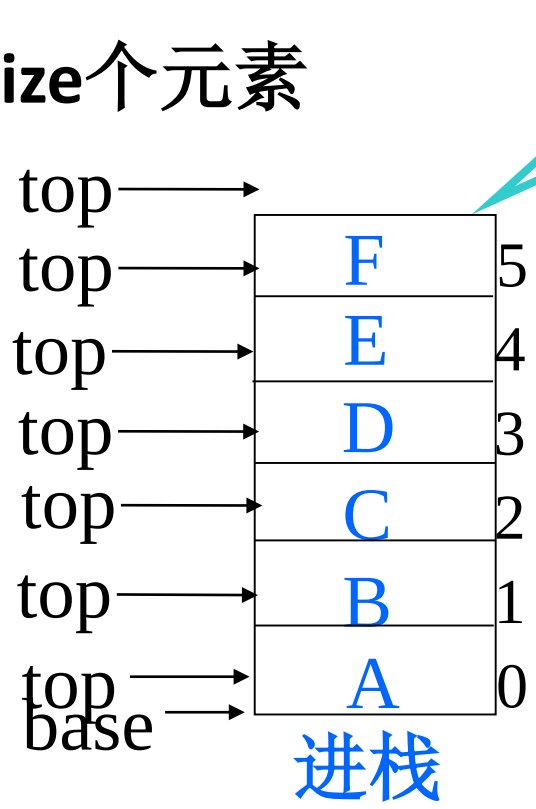
3.1 栈

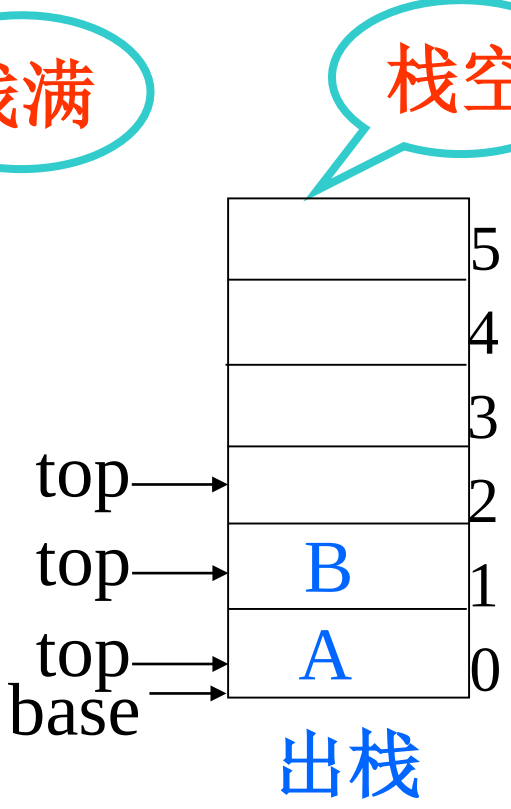


(动态)栈变化示意图

若栈的数组有Maxsize个元素







栈底指针base,始终指向栈底位置；栈顶指针top,其初值指向栈底，始终在栈顶元素的下一个位置

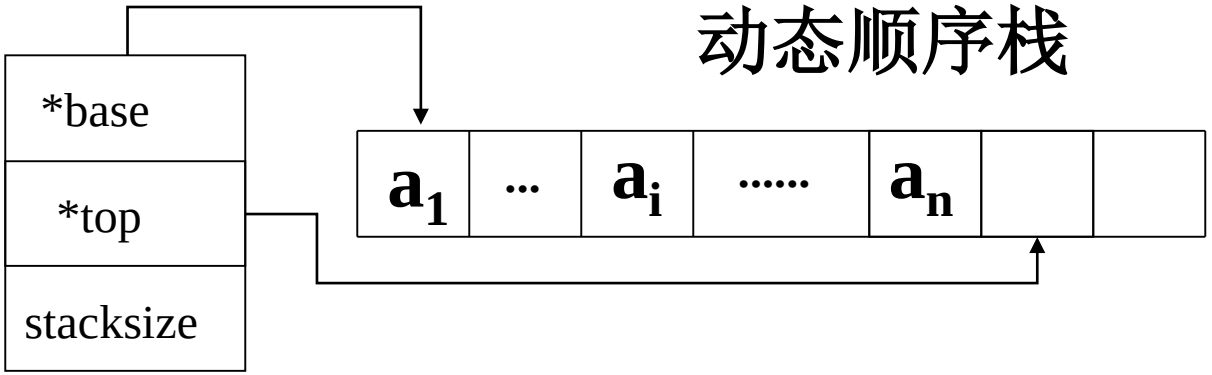
top=base,栈空，此时出栈，则下溢；top=base+stacksize,栈满，此时入栈，则上溢。

3.1 栈

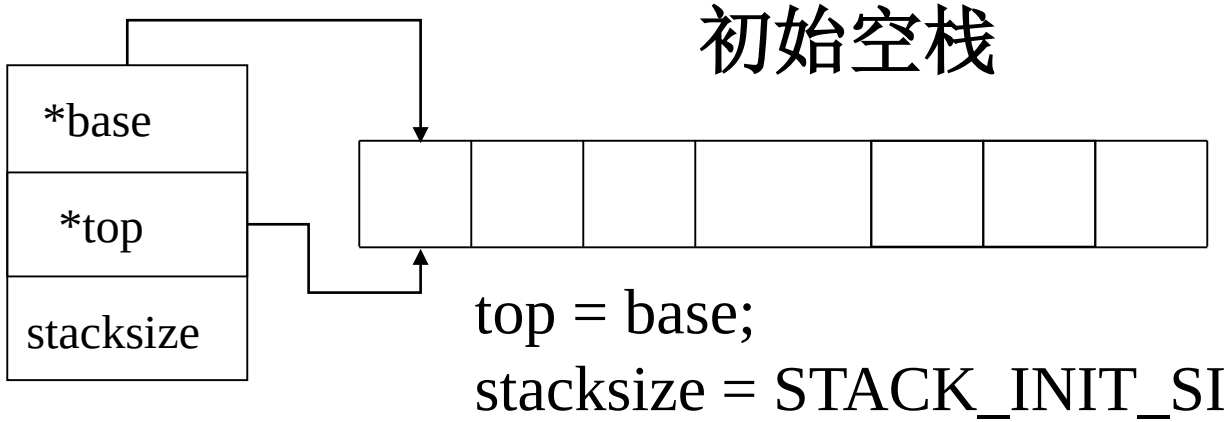
```
#define STACK_INIT_SIZE 100

#define STACKINCREMENT 10

typedef struct{

   SSSElemType *base;  /*在栈构造之前和销毁之后,base的值为
  NULL*/

   SSSElemType *top;   /*栈顶指针*/

   int stacksize;   /*当前已分配的存储空间，以元素为单位*/

}SqStack;
```

3.1 栈

- 非空栈中的栈顶指针始终在栈顶元素的下一个位置



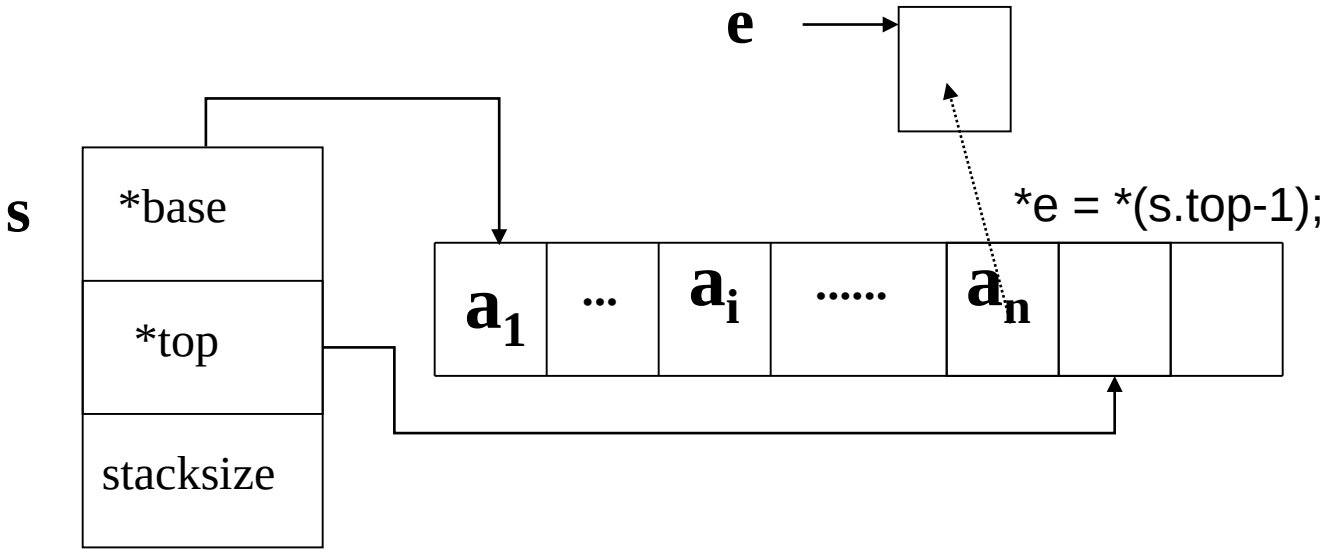
- 空栈的栈顶指针指向栈底



➢ 动态顺序栈的操作实现

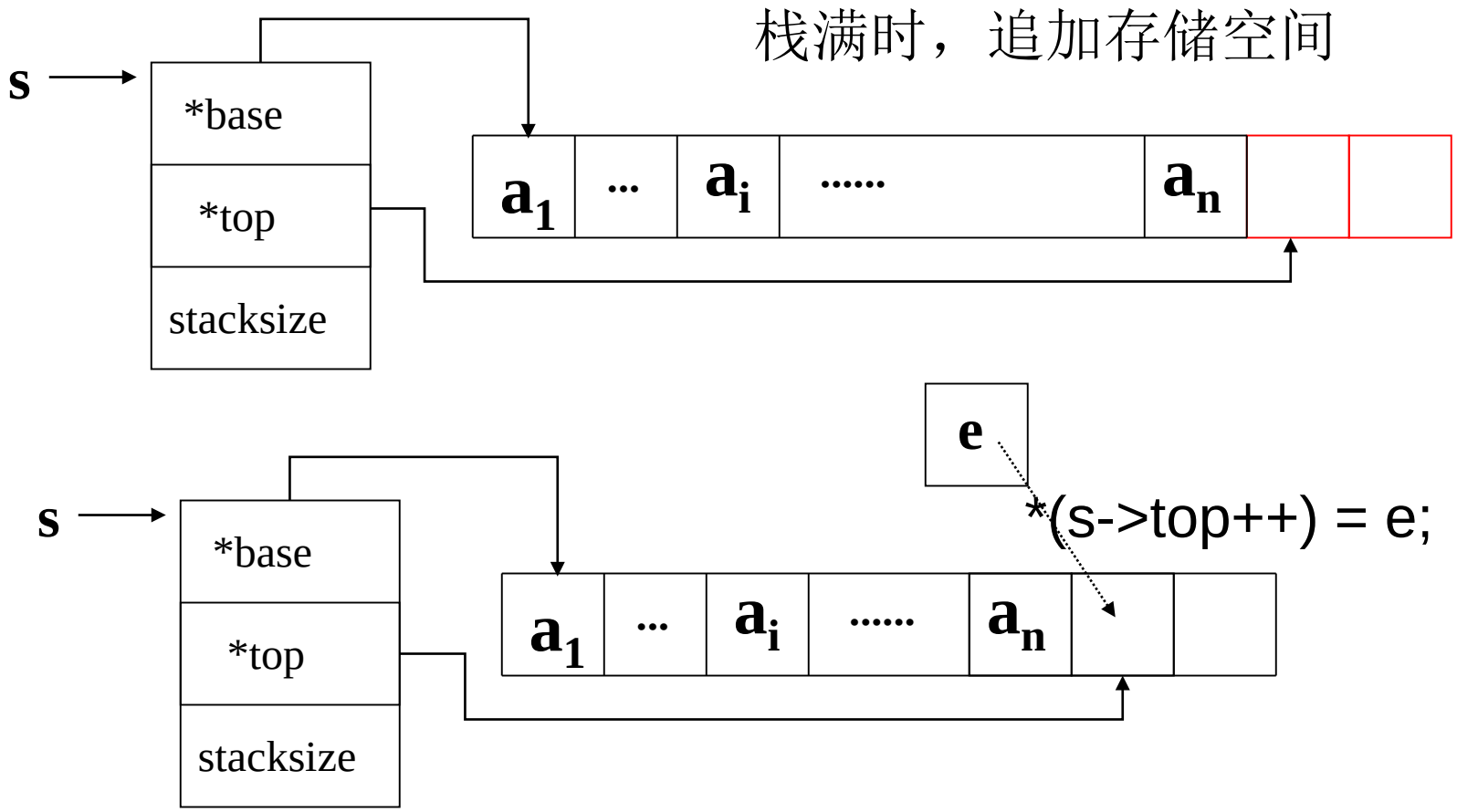
```
/*InitStack :构造一个空的栈S*/
Status InitStack(SqStack *s)
{
    s->base=(SSSElemType *)malloc(STACK_INIT_SIZE *
                    sizeof(SSSElemType));
    if(!s -> base) return(OVERFLOW);
    s->top=s->base;
    s->stacksize = STACK_INIT_SIZE;
    return OK;
}
```

```
/*GetTop: 返回栈S中栈顶元素*/
Status  GetTop(SqStack s, SSSElemType *e)
{
  if (s.top == s.base)return ERROR;
  *e = *(s.top-1);
    return OK;
}
```



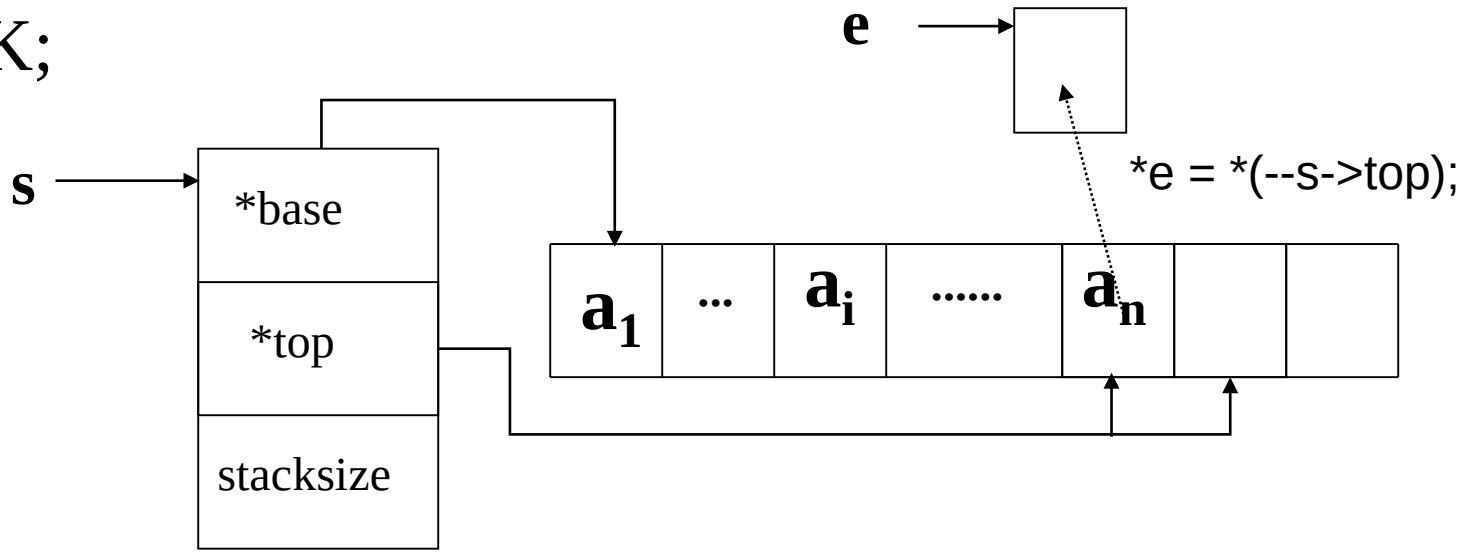
插入新栈顶元素时，堆栈变化示意图

- Push:插入元素e为新的栈顶元素



```
Status  Push( SqStack *s,  SSSElemType e )
{
      if (s->top – s->base >= s->stacksize){
        temp=(SSSElemType*)realloc(s->base,(s->stacksize+
STACKINCREMENT)*sizeof(SSSElemType));
        if (!temp) return(OVERFLOW);
        s->base = temp;
        s->top = s->base + s->stacksize;
        s->stacksize += STACKINCREMENT;
      }
      *(s->top++) = e;
       return OK;
}
```

```
/*Pop:删除S的栈顶元素*/
Status  Pop(SqStack *s, SSSElemType *e)
{
    if (s->top == s->base)return ERROR;
   *e = *(--s->top);
    return OK;
}
```



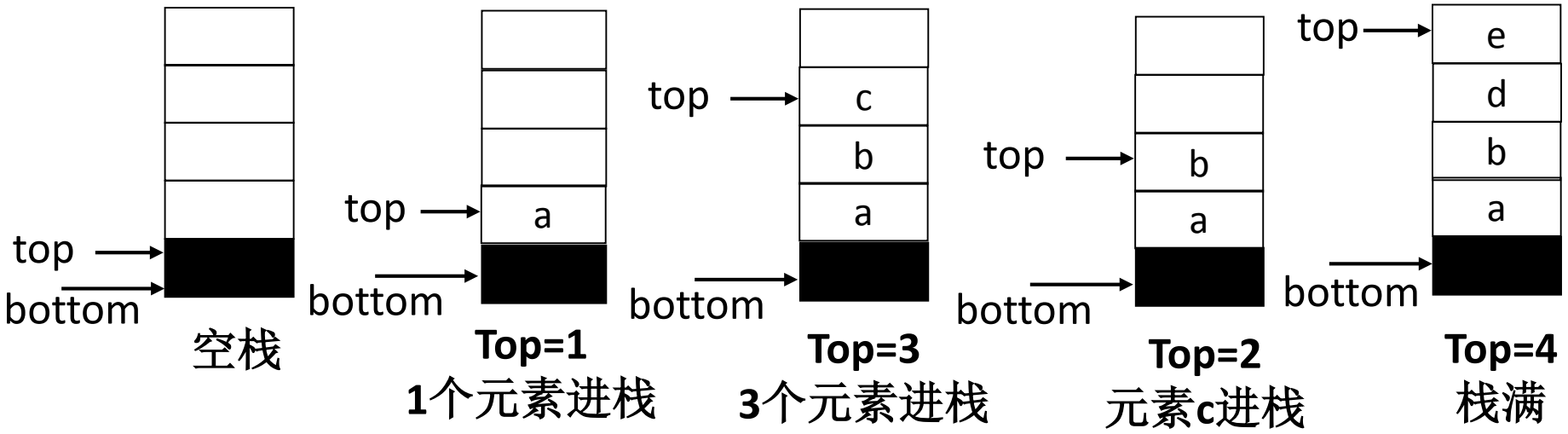
采用静态一维数组来存储栈。

栈底固定不变的，而栈顶则随着进栈和退栈操作变化的，

- 栈底固定不变的；栈顶则随着进栈和退栈操作而变化，用一个整型变量top(称为栈顶指针)来指示当前栈顶位置。
- 用top=0表示栈空的初始状态，每次top指向栈顶在数组中的存储位置。

- 结点进栈：首先执行top加1，使top指向新的栈顶位置，然后将数据元素保存到栈顶(top所指的当前位置)。
- 结点出栈：首先把top指向的栈顶元素取出，然后执行top减1，使top指向新的栈顶位置。

若栈的数组有Maxsize个元素，则top=Maxsize-1时栈满。



静态栈变化示意图

基本操作的实现

① 栈的类型定义

```
# define  MAX_STACK_SIZE  100      /*  栈向量大小  */

typedef struct {

    SSSElemType stack_array[MAX_STACK_SIZE] ;

    int  top;

}SqStack ;
```

② 栈的初始化

```
Status Init_Stack(SqStack  &S)

{

    S.bottom=S.top=0;

    return OK;

}
```

③ 压栈(元素进栈)

```
Status Push(SqStack &S , SSSElemType e)
   /*   使数据元素e进栈成为新的栈顶   */
  {
     if  (S.top==MAX_STACK_SIZE-1)
        return  ERROR;     /* 栈满，返回错误标志   */
     S.top++ ;     /*    栈顶指针加1 */
     S.stack_array[S.top]=e ;  /* e成为新的栈顶    */
     return OK;       /* 压栈成功    */
  }
```

④ 弹栈(元素出栈)

```
Status  Pop( SqStack   &S, SSSElemType &e )
   /*弹出栈顶元素*/
   {
      if ( S.top==0 )
         return ERROR ;      /* 栈空，返回错误标志   */
      e=S.stack_array[S.top] ;
      S.top-- ;
      return OK ;
   }
```

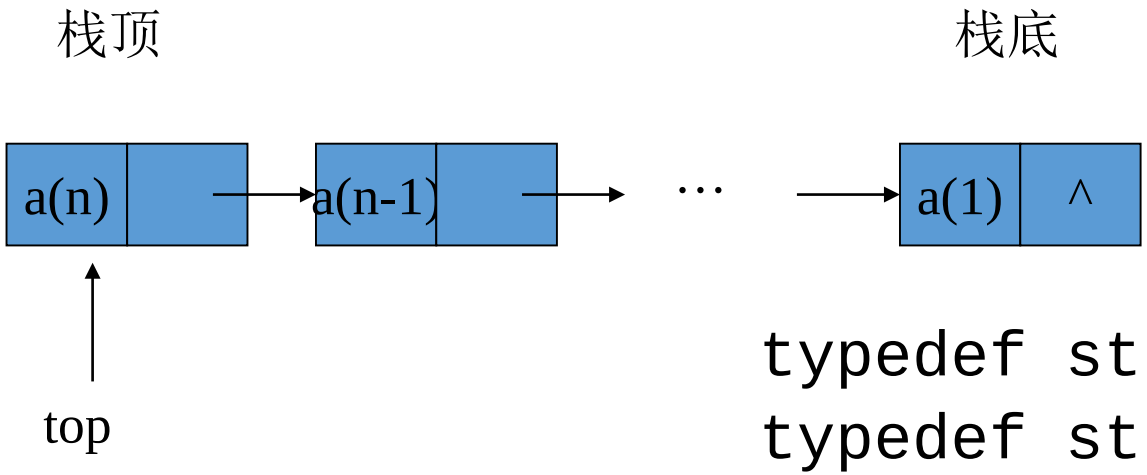
当栈满时做进栈运算必定产生空间溢出，简称“上溢”。上溢是一种出错状态，应设法避免。

当栈空时做退栈运算也将产生溢出，简称“下溢”。下溢则可能是正常现象，因为栈在使用时，其初态或终态都是空栈，所以下溢常用来作为控制转移的条件。

➢ 栈的链式表示

- 链栈:栈的链式存储结构称为**链栈**，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，故链表没有必要附加头结点。
- 指向栈顶表元素的指针就是链表的头指针，链式栈无栈满的问题，空间可扩充。

3.1 栈



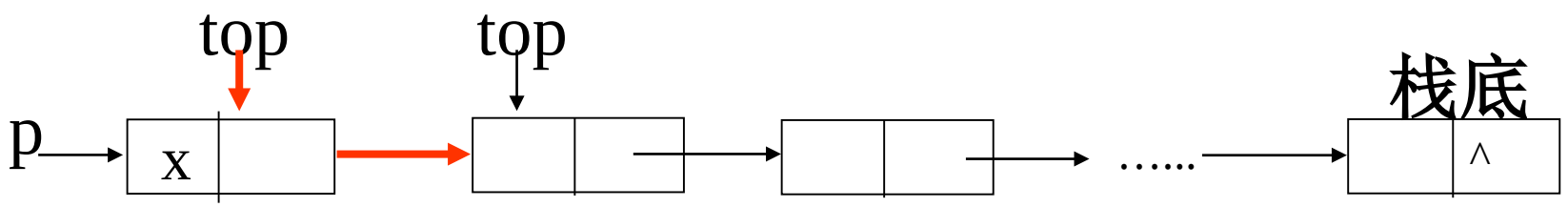
- 栈的链式表示

```
typedef struct snode *slink ;
typedef struct snode {
 StackItem data;
 slink next ;
} StackNode ;

typedef struct lstack {
  slink top ; // 栈顶结点指针
  int stackSize ;
} Lstack ;
```

➢ 链栈的初始化

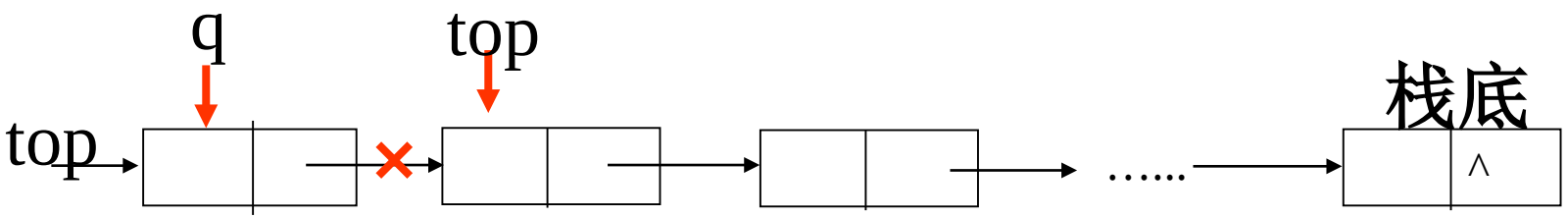
```
Status InitLinkStack(Lstack *s)
{
    s->top = NULL;
    s->stackSize = 0;
    return OK;
}
```

- 入栈操作



p->next=top ; top=p;

- 出栈操作



q=top ; e=q->data; top=top->next; free(q);

➢ 压栈(元素进栈)

```
Status Push(Lstack *s , SSSElemType e)
{
   StackNode   *p ;
   p=(StackNode   *)malloc(sizeof(StackNode  )) ;
   if (!p)  return  ERROR;
     /*  申请新结点失败，返回错误标志 */
   p->data=e ;
   p->next=s->top ;
   s->top=p ;
   s->stackSize ++;
   return OK;
}
```

- 弹栈(元素出栈)

```
Status Pop(Lstack *s , SSSElemType *e)
/*  将栈顶元素出栈  */
{
    StackNode    *q;     SElemType  e ;
    if  (s->top==NULL )
      return ERROR ;    /*  栈空，返回错误标志    */
    q=s->top ;
    e=q->data ;    /*  取栈顶元素  */
    s->top=s->top->next;          /*  修改栈顶指针  */
    s->stackSize --;
    free(q) ;
    return OK ;
}
```

Part.2

3.2 栈的应用举例

3.2 栈的应用举例

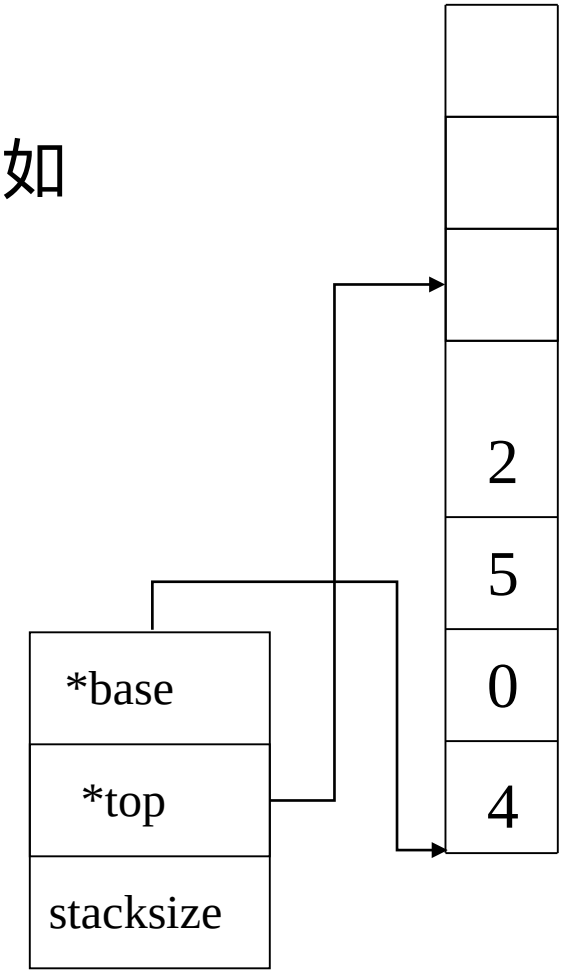
例1.数制转换:将十进制数N转换为其他进制数

基本原理：N=(n div d)*d + n mod d

“div”是整除运算，mod为求余运算，例如

$(1348)_{10}=(2504)_{8}$.

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



• 输入一个非负的十进制数，输出等值的八进制数

```
void conversion()
{
   int n ;
   SqStack s ;
   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 栈的应用举例

例2. 括号匹配的检测

匹配思想：从左至右扫描一个字符串(或表达式)，则每个右括号将与最近遇到的那个左括号相匹配。

则可以在从左至右扫描过程中把所遇到的左括号存放到堆栈中。

每当遇到一个右括号时，就将它与栈顶的左括号(如果存在)相匹配，同时从栈顶删除该左括号。

3.2 栈的应用举例

算法思想：设置一个栈，当读到左括号时，左括号进栈。

当读到右括号时，则从栈中弹出一个元素，与读到的左括号进行匹配，若匹配成功，继续读入；

否则匹配失败，返回FLASE。

例如：[([ ] [ ])]

算法描述

```
#define TRUE  0
#define  FLASE  -1
SqStack  S ;
S=InitStack() ;  /*堆栈初始化*/
int Match_Brackets( )
{
     char ch , x ;
     scanf(“%c” , &ch) ;
     while (asc(ch)!=13)
```

```
{   if  ((ch==‘(’)||(ch==‘[’))  Push(S , ch) ;
    else if  (ch==‘]’)
        {  x=Pop(S) ;
           if (x!=‘[’)
                 {  printf(“’[’括号不匹配”  ) ;
                    return FLASE  ;  }   }
    else if  (ch==‘)’)
         {  x=Pop(S) ;
            if (x!=‘(’)
               { printf(“’(’括号不匹配”  ) ;
                 return FLASE  ;}
         }
}
```

```
    if  (!StackEmpty(s))
    {
       printf(“括号数量不匹配！”);
       return FLASE  ;
     }
    else  return TRUE  ;
}
```

例3 数学表达式求值

- $9+(3-1)\times 3+10\div 2$

一种不需要括号的后缀表达法

3.2 栈的应用举例

- $9+(3-1)\times 3+10\div 2$

后缀表达式：9 3 1 - 3 * + 10 2 / +

3.2 栈的应用举例

后缀表达式：9 3 1 - 3 * + 10 2 / +

规则：

① 从左到右遍历表达式的每个数字和符号，

② 遇到是数字就进栈，

③ 遇到是符号，就将处于栈顶两个数字出栈，进行运算，运算结果再进栈

④ 一直到最终获得结果。

3.2 栈的应用举例

① 初始化一个空栈。此栈用来对要运算的数字进出使用。
② 后缀表达式中前三个都是数字，所以9、3、1进栈。
③ 接下来是“-”，所以将栈中的1出栈作为减数，3出栈作为被减数，并运算3-1得到2，再将2进栈
④ 接着是数字3进栈
⑤ 后面是“*”，也就意味着将栈中的3和2出栈，2与3相乘，得到6，并将6进栈
⑥ 接下面是“+”，所以栈中的6和9出栈，相加得到15，将15进栈
⑦ 接着是10与2两数字进栈
⑧ 接下来是符号“/”，因此，栈顶的2与10出栈，10与2相除得到5，将5进栈
⑨ 最后一个是符号“+”，所以15与5出栈相加得到20，将20进栈
⑩ 结果是20出栈，栈变为空。

3.2 栈的应用举例

- 中缀表达式：$9+(3-1)\times3+10\div2$
- 后缀表达式：9 3 1 – 3 * + 10 2 / +

规则：

①从左到右遍历中缀表达式的每个数字和符号；

②若是数字就输出，即成为后缀表达式的一部分；

③若是符号，则判断其与栈顶符号的优先级，如果是右括号或优先级不高于栈顶符号则栈顶元素依次出栈并输出，并将当前符号进栈；

④直到最终输出后缀表达式为止。

运算符间的优先关系

θ_1 \ θ_2	+	-	*	/	(	)
+	>	>	<	<	<	>
-	>	>	<	<	<	>
*	>	>	>	>	<	>
/	>	>	>	>	<	>
(	<	<	<	<	<	≒
)	>	>	>	>		>

① 初始化一个空栈。此栈用来对符号进出栈使用；

② 第一个字符是数字9，输出9，后面是符号“+”进栈；

③ 第三个字符是“（”，因其只是左括号，还未配对，故进栈；

④ 第四个字符是数字3，输出，总表达式为9 3，接着是“-”进栈

⑤ 接下来是数字1，输出，总表达式为9 3 1，后面是符号“）”。此时需要去匹配此前的“（”，所以栈顶依次出栈，并输出，直到“（”出栈为止。此时左括号上方只有“-”，因此输出“-”，总的输出表达式为9 3 1 -；

⑥ 紧接着是符号“*”，因为此时的栈顶符号为“+”，优先级低于“*”，因此不输出，“*”进栈，接着是数字3，输出，总的表达式为9 3 1 - 3；

⑦ 之后是符号“+”，此时栈顶元素是“ *”，比“+”优先级高，因此栈中元素出栈，总的表达式为9 3 1 - 3 * +，然后再将这个符号“+”进栈。

⑧ 紧接着是数字10，输出，表达式为9 3 1 - 3 x + 10，后面是符号÷，所以“/”进栈。

⑨ 最后一个数字2，输出，总表达式为9 3 1 - 3 x + 10 2。因为到最后，所以栈中符号全部出栈并输出，为： 9 3 1 - 3 x + 10 2 / +。

3.2 栈的应用举例

① 将中缀表达工转化为后缀表达式

（栈用来存进出运算的符号）

② 将后缀表达式进行运算得出结果

（栈用来存进出运算的数字）

复　习

1. 线性表的定义

- 线性表是具有相同数据类型的n(n≥0)个数据元素的有限序列（其中n为表长，当n=0时该线性表是一个空表）
- 一般表示：
  - 线性表L：$(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$

2. 线性表的插入和操作

- ★ **ListInsert(&L, i, e) – 插入操作 – 在表L中第i个位置上插入元素e**

  要求： 1≤i≤n+1

- ★ **ListDelete(&L, i, &e) – 删除操作 删除表L中第i个位置的元素，并用e返回删除元素的值**

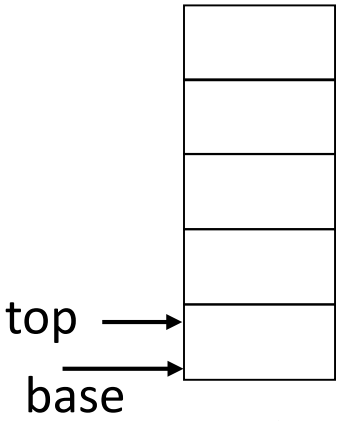
  要求： **1≤i≤n**

复　习

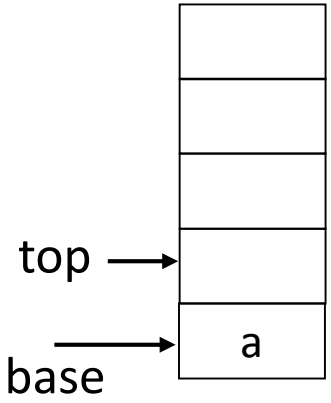
1. 栈的定义

- 栈是限制在表的一端进行插入和删除运算的线性表
- 插入、删除的这一端为栈顶Top，另一端为栈底Bottom

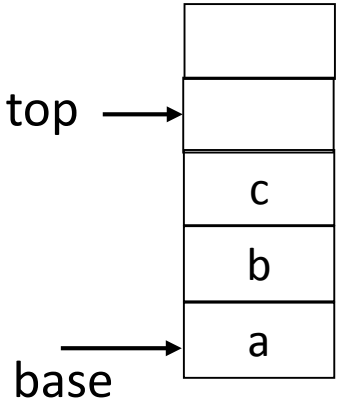
2. 栈的入栈和出栈操作



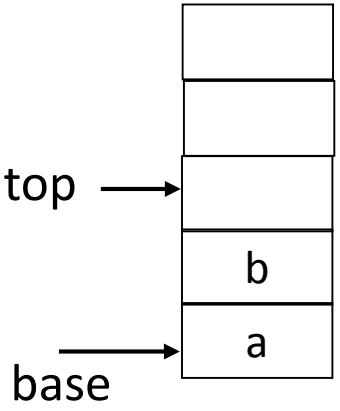
空栈



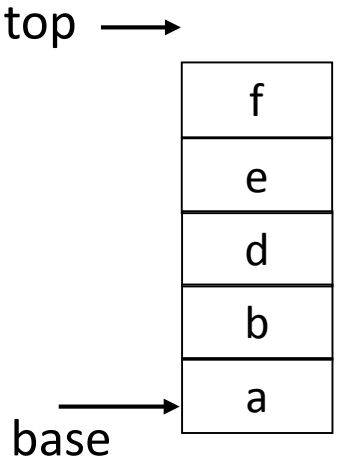
元素a进栈



元素b，c进栈



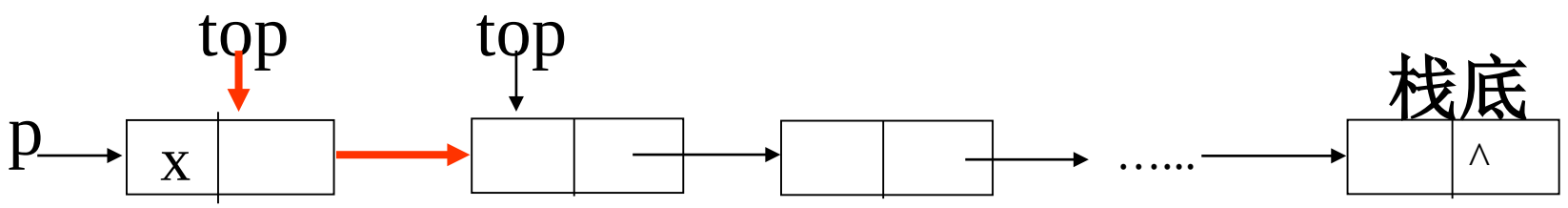
元素c退栈



元素d，e，f进栈

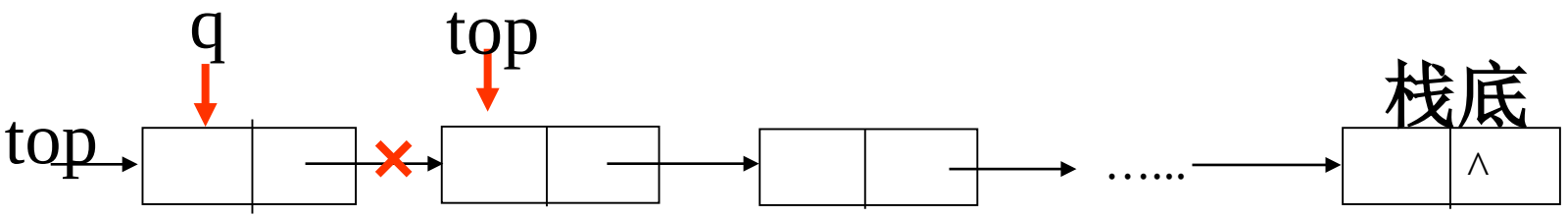
(动态)栈变化示意图

- 入栈操作



p->next=top ; top=p;

- 出栈操作



q=top ; e=q->data; top=top->next; free(q);

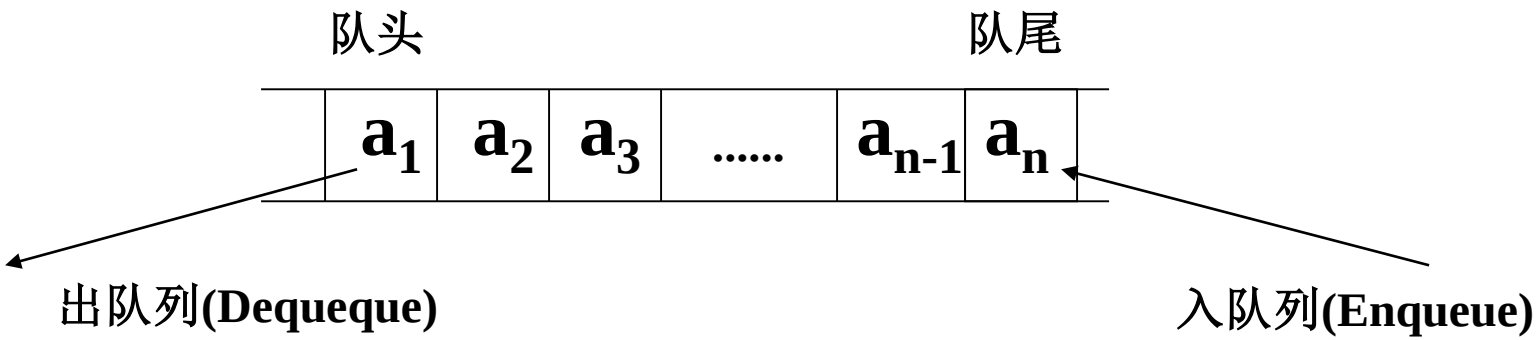
链栈变化示意图

Part.3

3.3 队列

3.3 队列

- 队列（Queue）：先进先出(First In First Out)
  - 仅在**队尾进行插入**和**队头进行删除**操作的线性表。
  - (缩写为FIFO）的线性表。
- 队头（front）：线性表的表头端，即可删除端。
- 队尾（rear）：线性表的表尾端，即可插入端。



➢队列抽象数据类型的定义

ADT Queue {

数据对象：D = {a_i | a_i属于Elemset, (i=1, 2, …, n, n≥0)}

数据关系：R_1 = {＜a_{i-1},a_i＞|a_{i-1},a_i属于D,(i=2,3,…,n)。

基本操作：

InitQueue(&Q); DestroyQueue (& Q);

ClearQueue (& Q); QueueEmpty(Q);

QueueLength(Q) ; **GetHead(Q,&e);**

EnQueue (& Q,e); **DeQueue (& Q,&e);**

}ADT Queue

3.3 队列

- 队列的基本操作(之一)
  - InitQueue (&Q)
    - 操作结果:构造一个空的队列Q。
  - DestroyQueue (&Q)
    - 初始条件: 队列Q已经存在。
    - 操作结果: 销毁队列Q。
  - ClearQueue (&Q)
    - 初始条件: 队列Q已经存在。
    - 操作结果: 将队列Q重置为空队列。

队列的基本操作(之二)

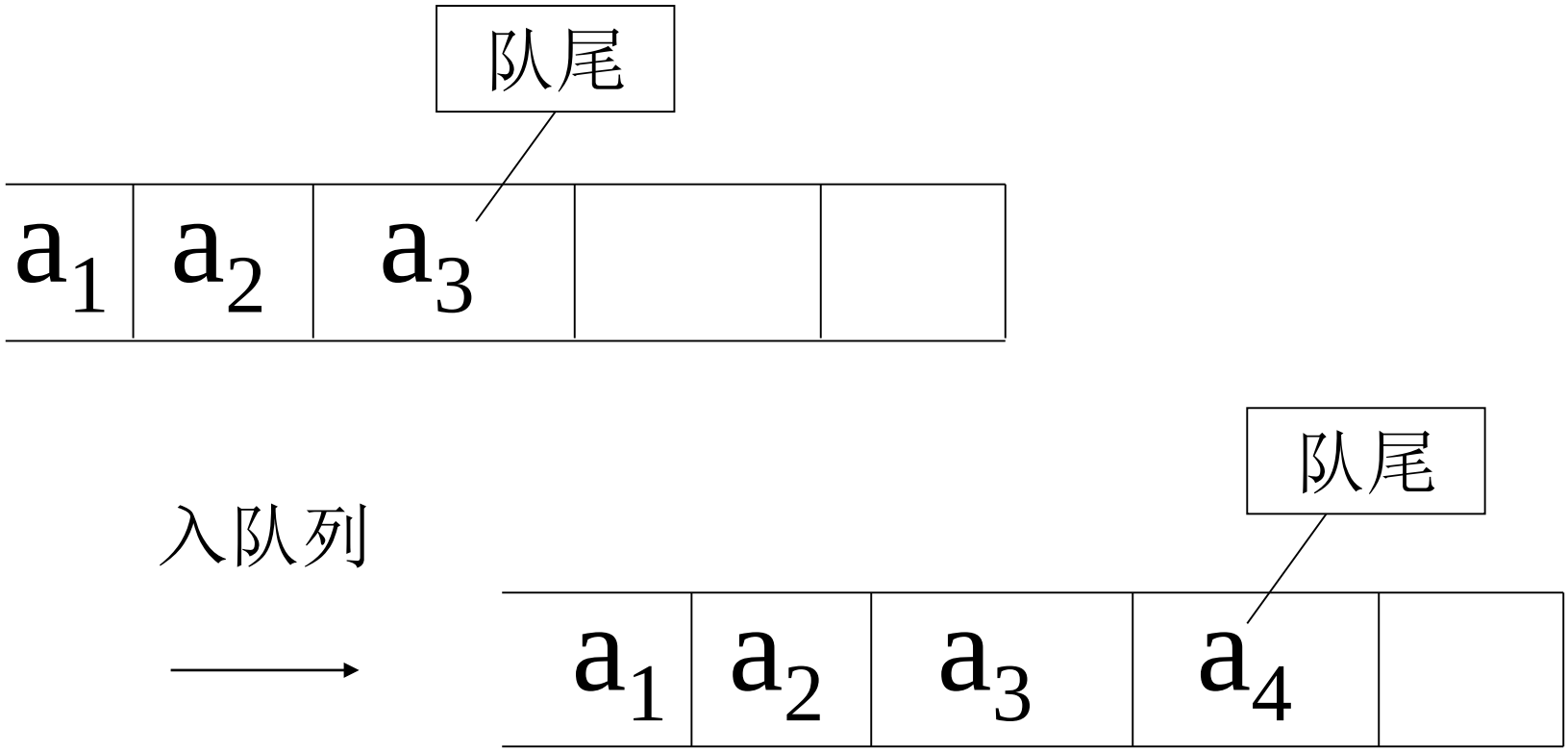
- QueueEmpty(Q)
  - 初始条件:队列Q已经存在。
  - 操作结果:若队列Q为空队列,则返回TURE;否则返回FALSE。
- QueueLength(Q)
  - 初始条件:队列Q已经存在。
  - 操作结果:返回队列Q中的数据元素个数,即队列Q的长度。
- GetHead(Q,&e)
  - 初始条件:队列Q已经存在且非空。
  - 操作结果:用e返回队列Q中队头元素的值。

➢ 队列的基本操作(之三)

- EnQueue (&Q,e)
  - 初始条件: 队列Q已经存在。
  - 操作结果: **插入元素e为新的队尾元素**。
- DeQueue (&Q,&e)
  - 初始条件: 队列Q已经存在且非空。
  - 操作结果: **删除队列Q 的队头元素**并用e返回其值。

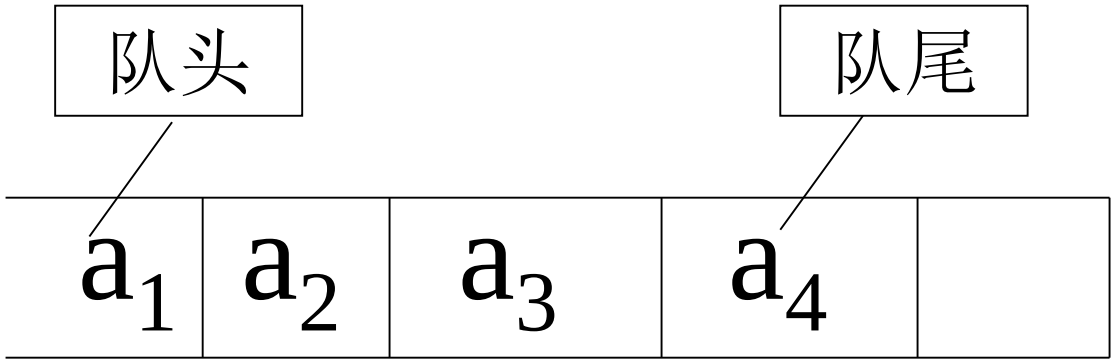
3.3 队列

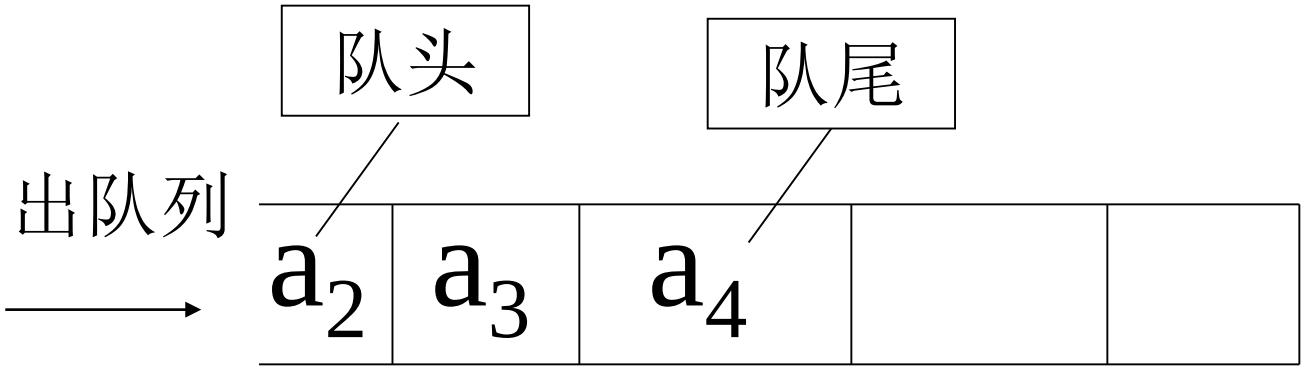
- 队列的顺序存储



3.3 队列

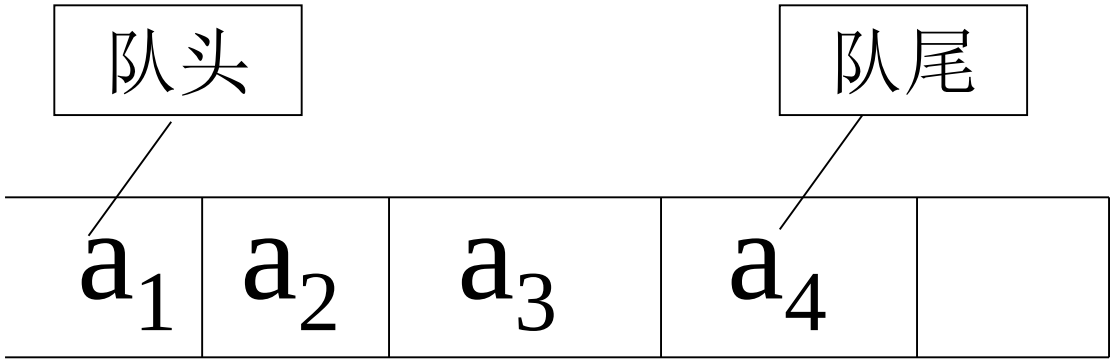
- 队列的顺序存储

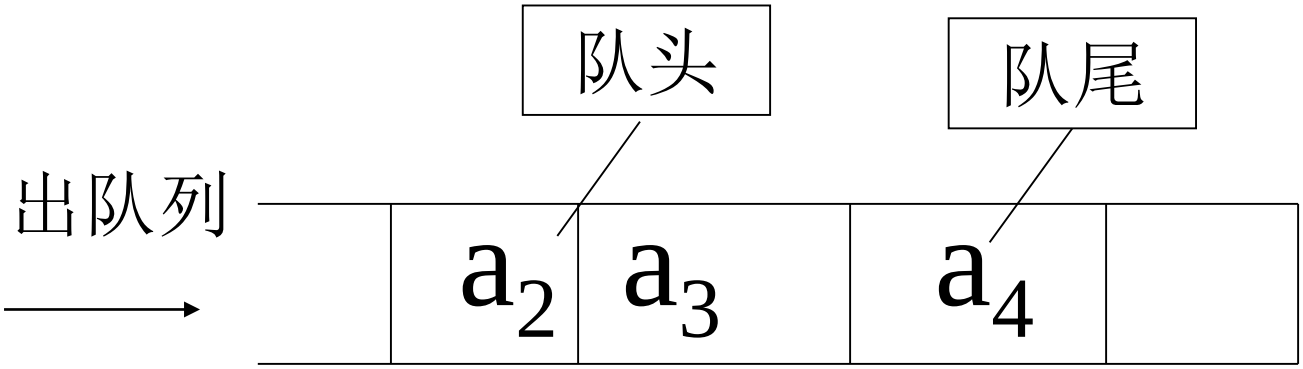




3.3 队列

- 队列的顺序存储



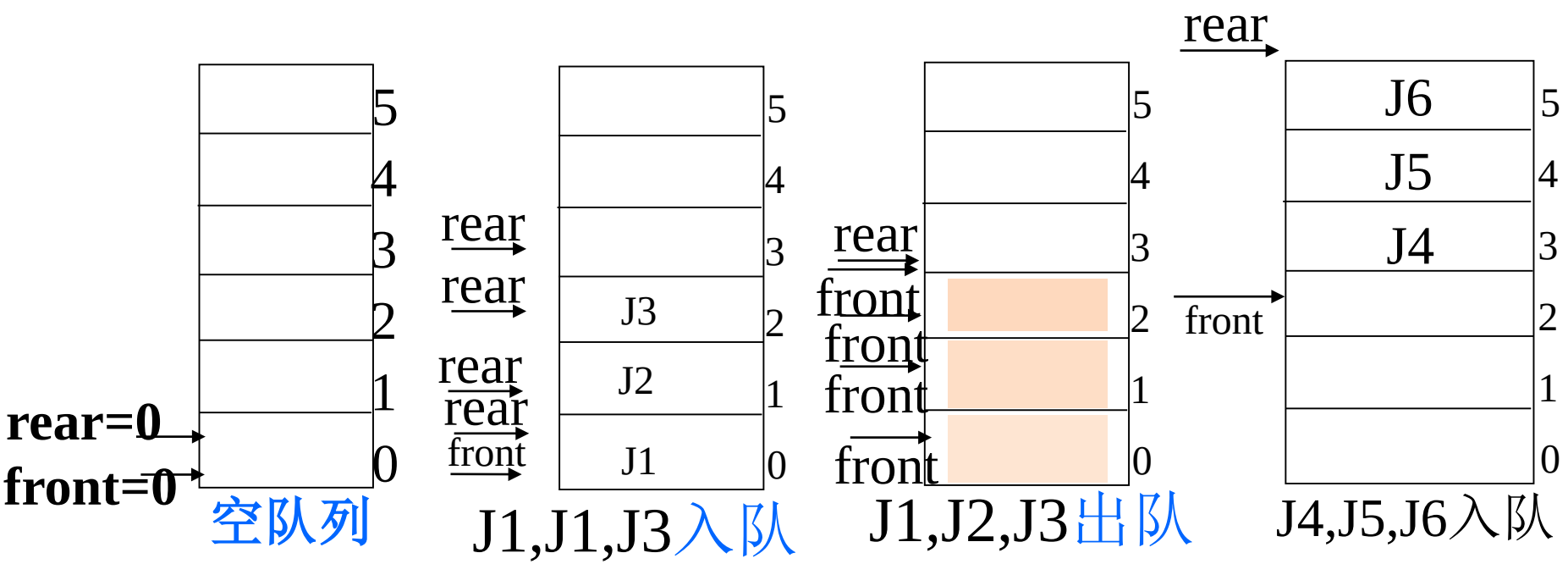


3.3 队列

➢队列的顺序存储

```
#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：队列长度为6(Maxsize)



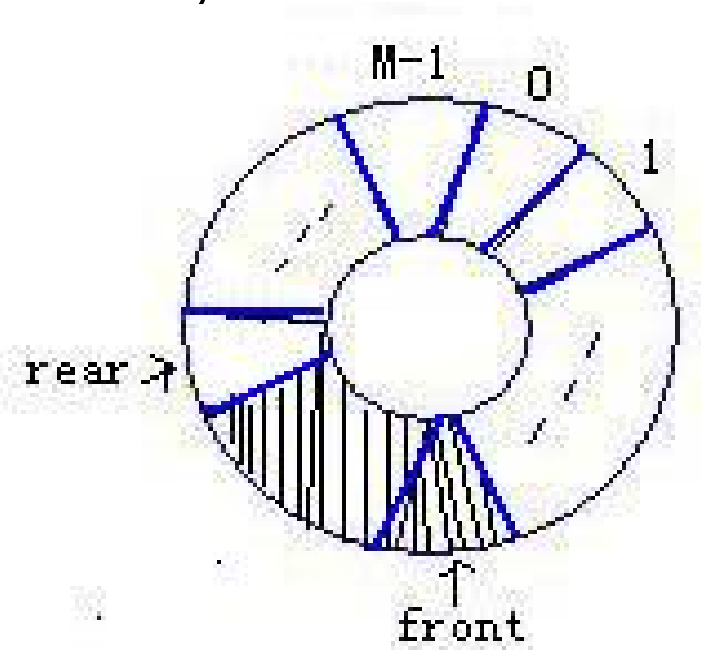
❖**存在问题：**

- **当front≠0,rear=M时再有元素入队发生溢出——假溢出**
- **当front=0,rear=M时再有元素入队发生溢出——真溢出**

3.3 队列

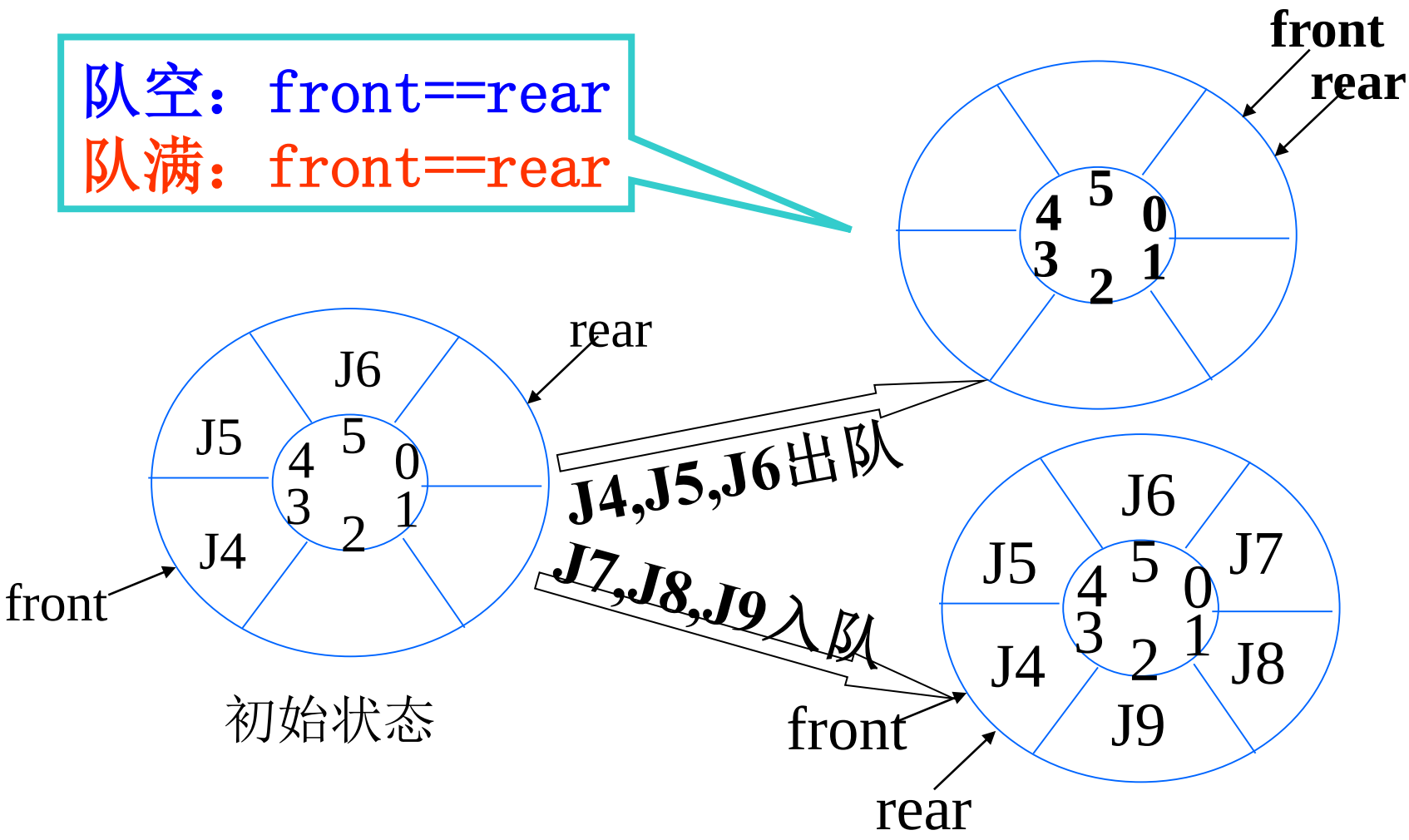
➢循环队列

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



➢ 循环队列

队空：front==rear
队满：front==rear



- 循环队列

解决方案：

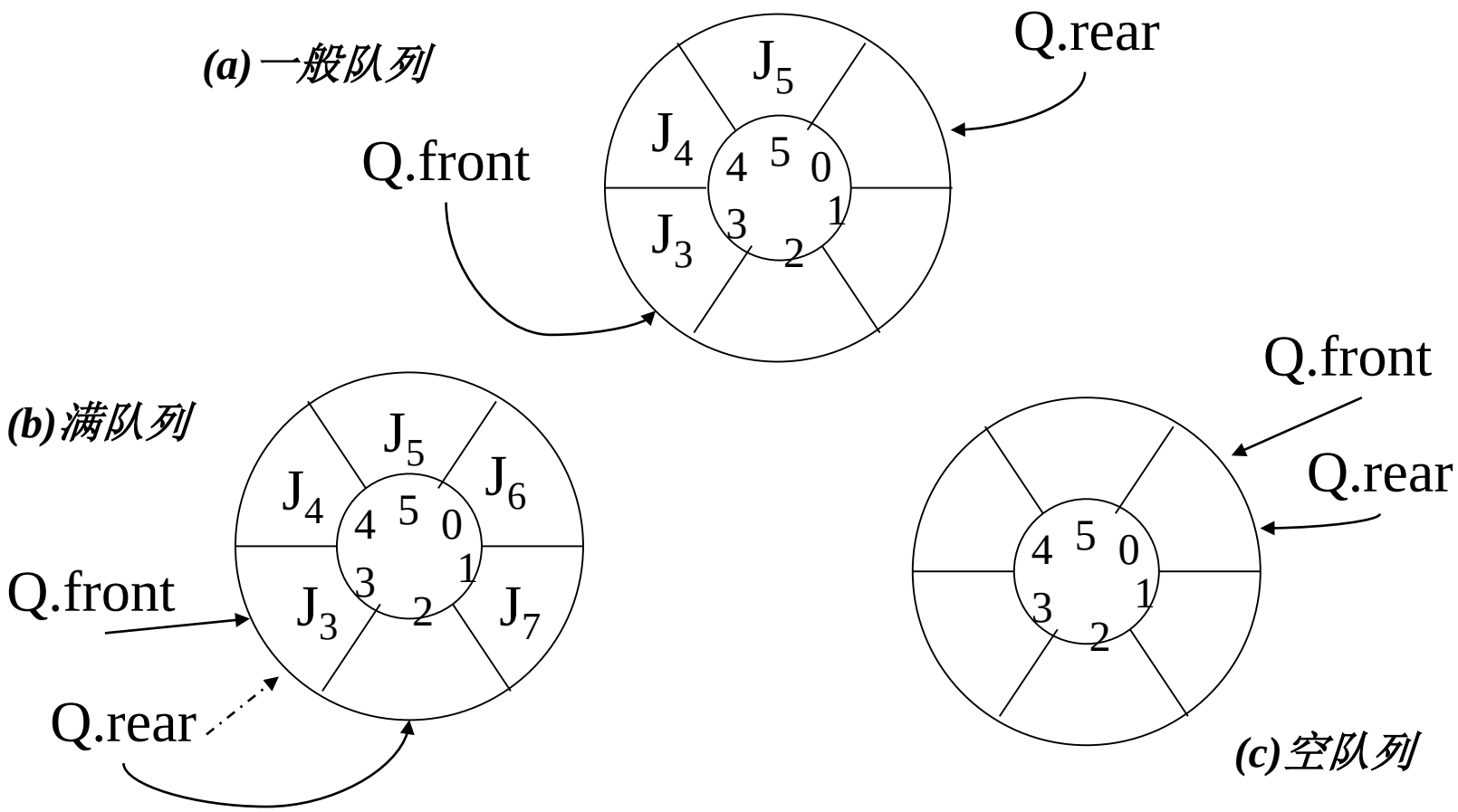
①设一个布尔变量以区别队列的空和满

②少用一个元素空间，约定当循环数组中元素个数达到maxsize-1时队列为满，即以队列头指针指向队列尾指针的下一位置（指环的下一位置)上作为队列呈”满“的标志

队空：front==rear

队满：(rear+1)%M==front

➢ 循环队列空或满的判断


(a)一般队列
Q.rear
Q.front
J5
J4
J3
4
5
0
3
1
2
(b)满队列
J5
J4
J6
J3
J7
Q.front
Q.rear
(c)空队列
Q.front
Q.rear

➢循环队列实现

```
typedef struct {
   QElemType  *base; //  存储空间基地址
   int front;        //  队头指针
   int rear;//队尾指针，若队列不空，指向 队列尾元素的下一个位置
} SqQueue;
#define MAXSIZE 100
```

➢循环队列实现

```
Status InitQueue(SqQueue &Q)
{
    Q.base=(QElemType*)malloc(sizeof(QElemType)*MAXSIZE );
    if(!Q.base) return(OVERFLOW);
    Q.front  = Q.rear = 0;
    return OK;
}
```

➢循环队列实现

```
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;
}
```

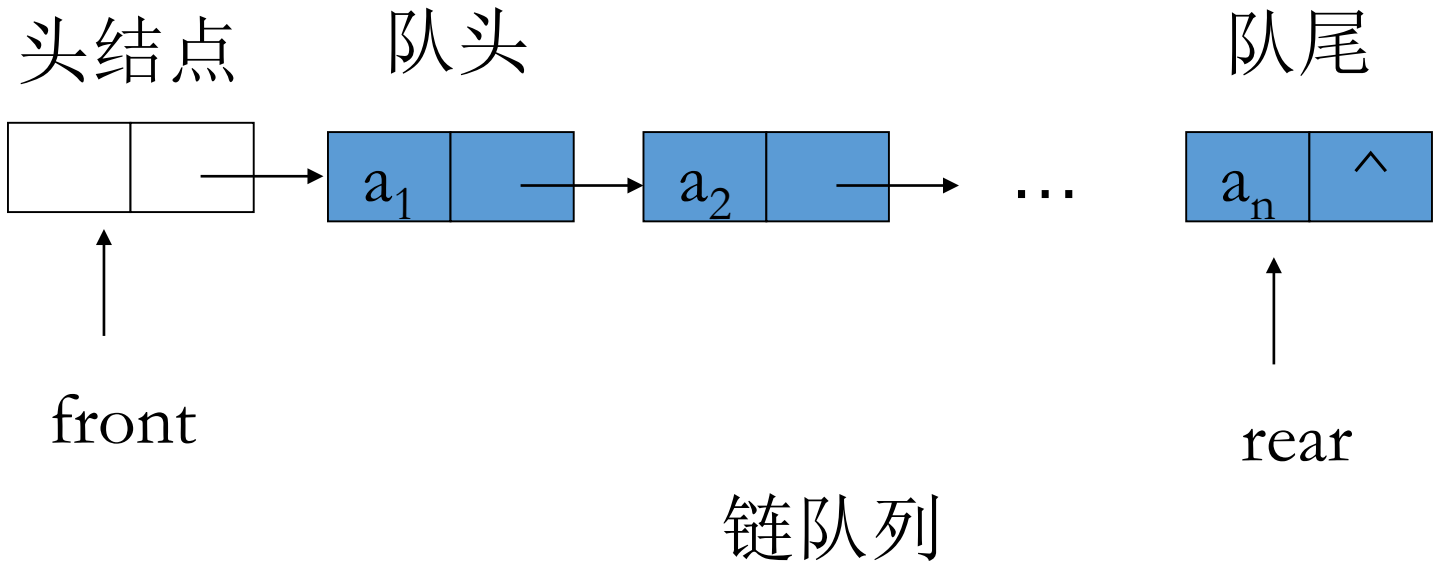
➢循环队列实现

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

➢链队列-队列的链式表示与实现

- 链队列：队列的链式存储结构简称为链队列，

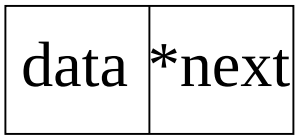
  它是限制仅在表头删除和表尾插入的单链表。

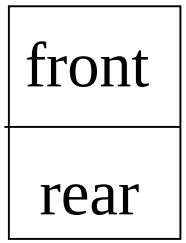


链队列

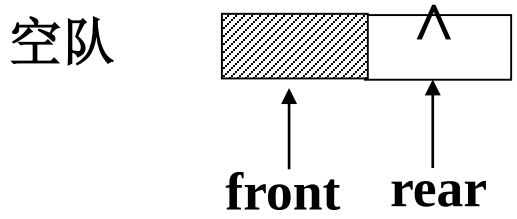
➢ 链队列-队列的链式表示与实现

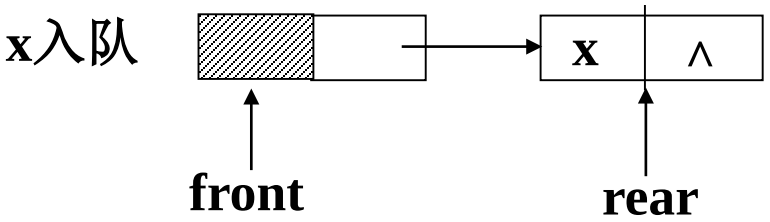
```
typedef struct QNode{
  QElemType data;
  struct QNode *next;
}Qnode, *QueuePtr;
```





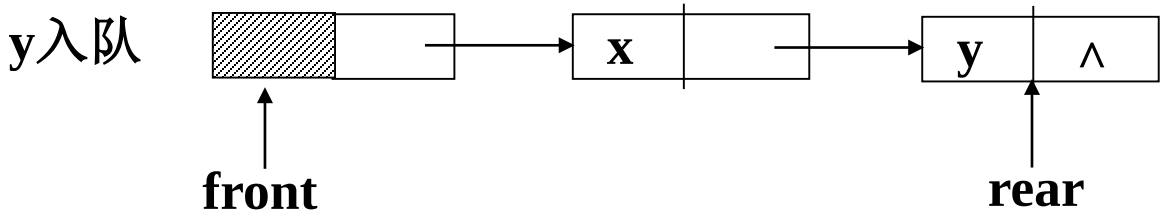

```
typedef struct {
   QueuePtr front;  //  队头指针
  QueuePtr rear;   //  队尾指针
}LinkQueue;
```

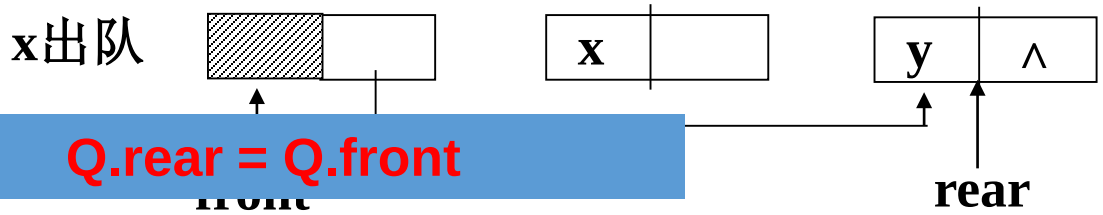




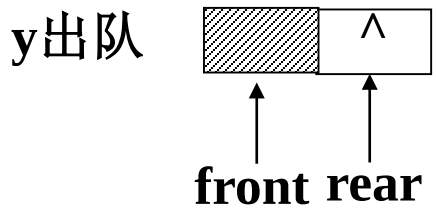
Q.rear -> next=p

Q.rear=p





Q.rear = Q.front



p= Q.front -> next

Q.front -> next = p -> next

➢ 链队列-队列的链式表示与实现

• InitQueue:构造一个空的队列Q

```
Status InitQueue(LinkQueue &Q)
{
   Q.rear=(QueuePtr)malloc(sizeof(QNode) );
   Q.front = Q.rear;
   if(!Q.front) return(OVERFLOW);
   Q.front -> next = NULL;
   return OK;
} // InitQueue
```

➢ 链队列–队列的链式表示与实现

• DestroyQueue:销毁队列Q

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

➢ 链队列–队列的链式表示与实现

• EnQueue:插入元素e为新的队尾元素

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

➢ 链队列–队列的链式表示与实现

• DeQueue:删除队列Q 的队头元素并用e返回其值

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

➢ 循环队列与链队列比较

- 时间
- 空间

在可以确定队列长度最大值的情况下，建议用循环队列。

如果无法预估队列的长度时，则用链队列。

Part.4

总结

总 结

- 栈：限定仅在栈顶进行插入和删除操作的线性表。
- 队列：允许在一端进行插入操作，而在另一端进行删除操作的线性表。

栈

- 顺序栈
- 链栈

队列

- 顺序队列
  - 循环队列
- 链队列

总 结

1. 掌握栈和队列这两种抽象数据类型的特点，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，即**两种存储结构表示时的基本操作实现算法**。特别应**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。