数据结构与算法设计

课程内容

课程内容：数据结构部分

概述	数组与广义表	图	外部排序
线性表	串	查找	
栈与队列	树	内部排序	

课程内容：算法设计部分

概述	贪心	……	计算模型
分治	回溯	……	可计算理论
动态规划	……	……	计算复杂性

Contents

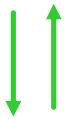
本章内容

- 栈的表示与实现
- 栈的应用
- 队列的表示和实现
- 队列的应用
- 小结

3.1.1什么是栈(stack)

栈是仅能在表头或表尾进行插入、删除操作的线性表。

$$(a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$$



插入

删除

3.1.1什么是栈(stack)

$(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$

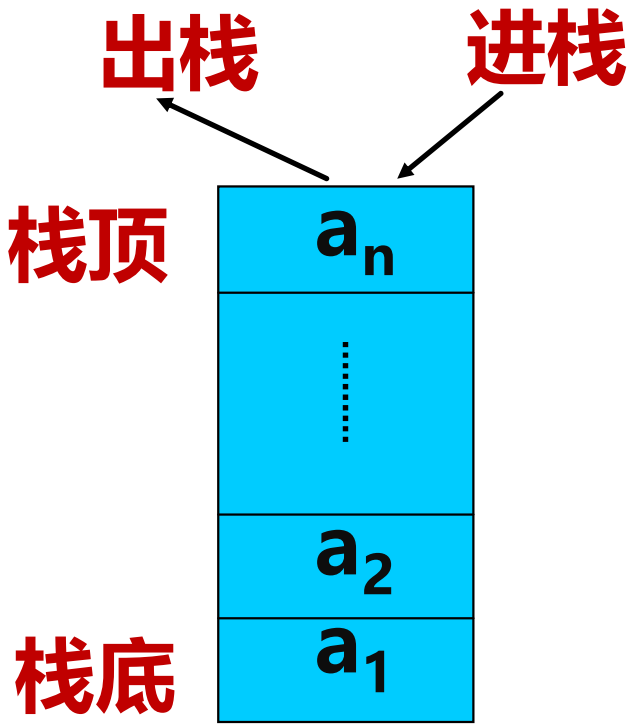
删除 ↓ ↑ 插入

能进行插入和删除的一端称为栈顶(top),另一端称为栈底(bottom)。

称插入操作为进栈(push)，删除操作为出栈(pop)。

栈的特征是后进先出（LIFO， Last_in First_out)

第一个进栈的元素在栈底
最后一个进栈的元素在栈顶
第一个出栈的元素为栈顶元素
最后一个出栈的元素为栈底元素



栈的示意图

3.1.1栈的类型定义

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：……

} ADT Stack

3.1.1栈的基本操作-1

1)初始化操作 InitStack(&S)

功能：构造一个空栈S。

2)销毁栈操作 DestroyStack(&S)

功能：销毁一个已存在的栈。

3)置空栈操作 ClearStack(&S)

功能：将栈S置为空栈。

3.1.1栈的基本操作-2

4)取栈顶元素操作 GetTop(S, &e)

功能：取栈顶元素，并用e 返回。

5)进栈操作 Push(&S, e)

功能：元素e进栈。

6)出栈操作 Pop(&S, &e)

功能：栈顶元素出栈，并用e返回。

7)判空操作 StackEmpty(S)

功能：若栈S为空，则返回True，否则，栈不空返回False

3.1.2 栈的物理实现方式

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

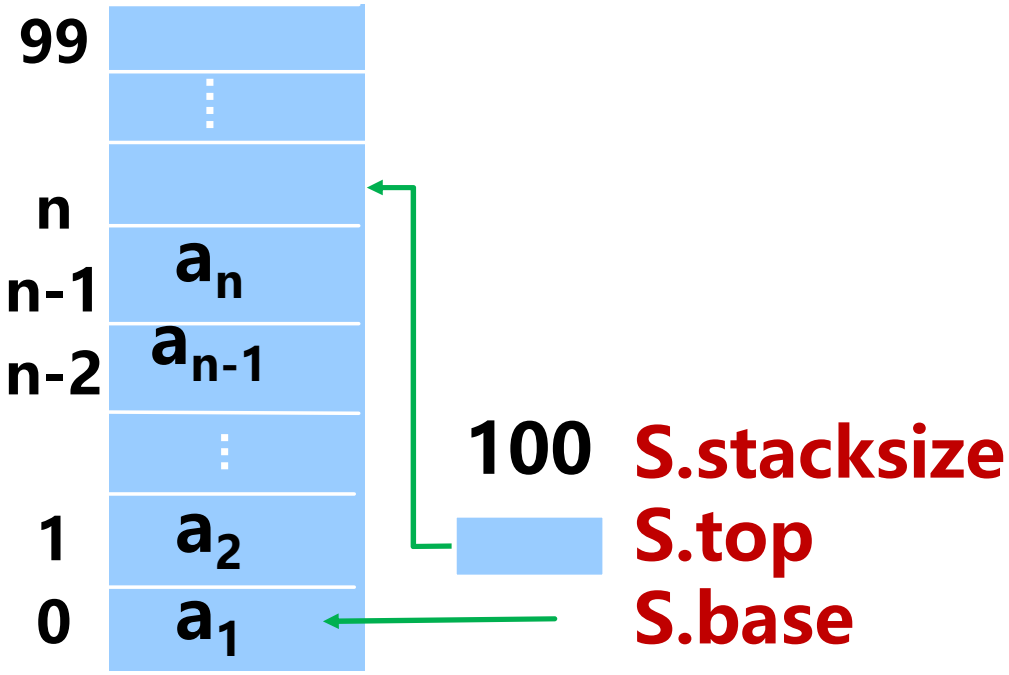
顺序结构（数组）

链表方式

3.1.2 栈的顺序存储结构

```
#define STACK_INIT_SIZE 100    // 栈存储空间的初始分配量
#define STACKINCREMENT 10  // 空间的分配增量
typedef struct
{   ElemType  *base;       //栈空间基址
    ElemType  *top;        //栈顶指针
    int stacksize;         //当前分配的栈空间大小
} SqStack;
```

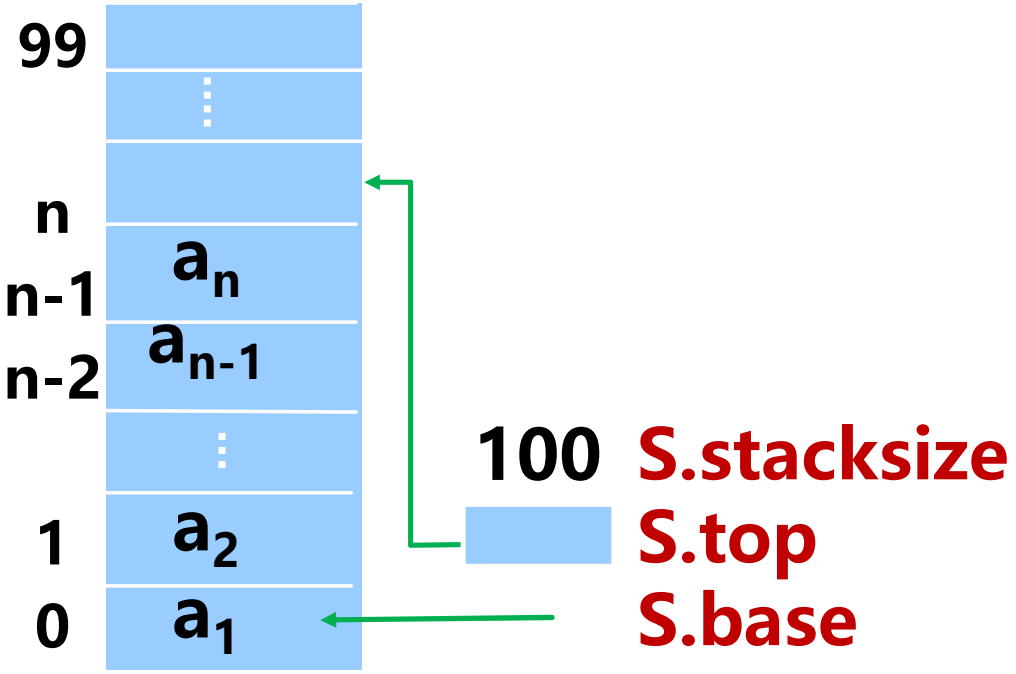
3.1.2 栈的顺序存储和实现



顺序栈的图示

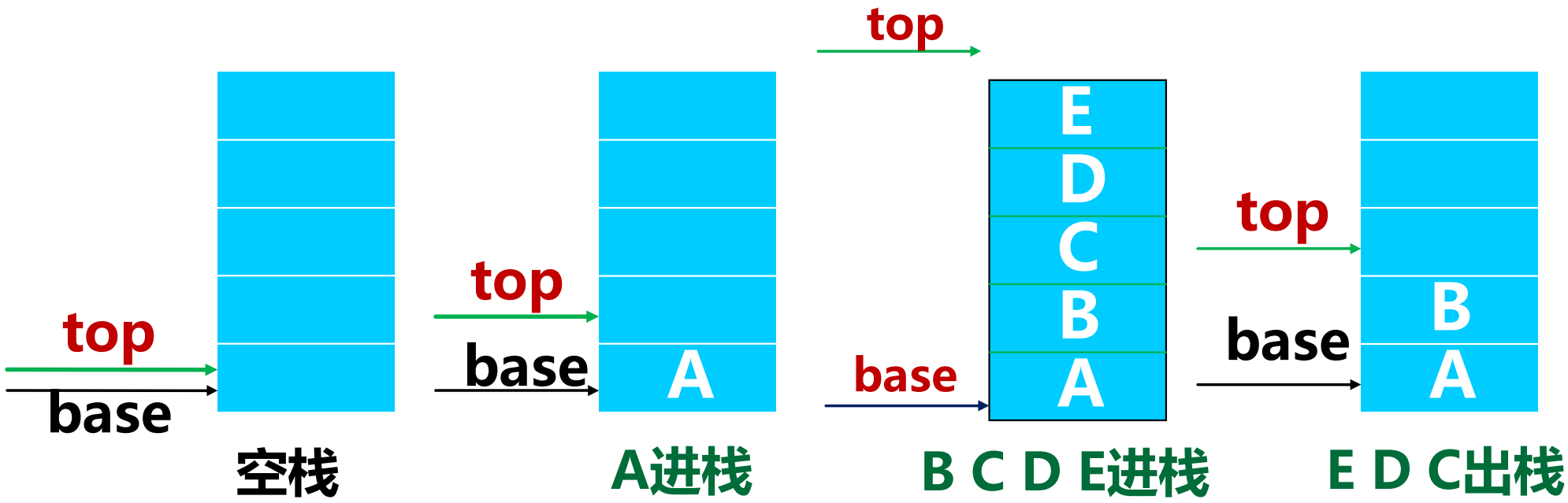
3.1.2 栈的顺序存储和实现

栈顶指针指向哪里?
A）栈顶第一个空闲位置
B）栈顶元素的位置



顺序栈的图示

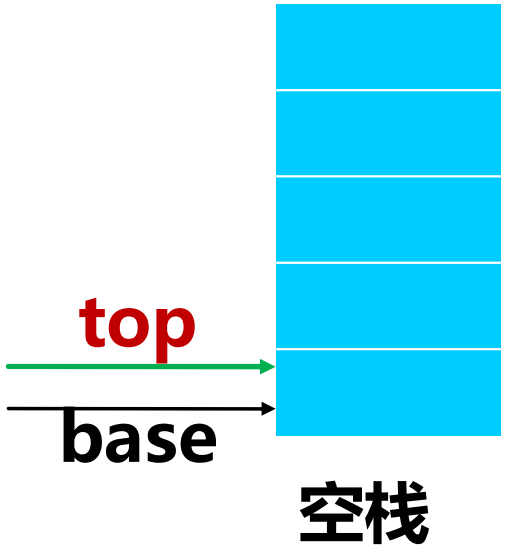
3.1.2 栈的顺序存储和实现



空栈　top = base

栈满　top-base = stacksize (无可分配空间)

栈空



空栈

空栈　top == base

栈顶指针top，指向实际栈顶后的空位置，初值为 base

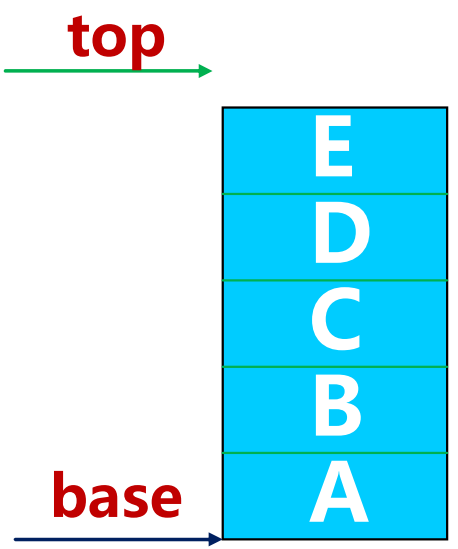
top==base，已栈空，此时再出栈，则下溢(underflow)

栈满　top-base = stacksize (无可分配空间)

设栈的初始分配量为 Stacksize = STACK_INIT_SIZE。

若 top-base == Stacksize，栈满，此时入栈，则需扩充栈空间，每次扩充 STACK_INCREMENT；

若无可利用的存储空间，则上溢 (overflow)。



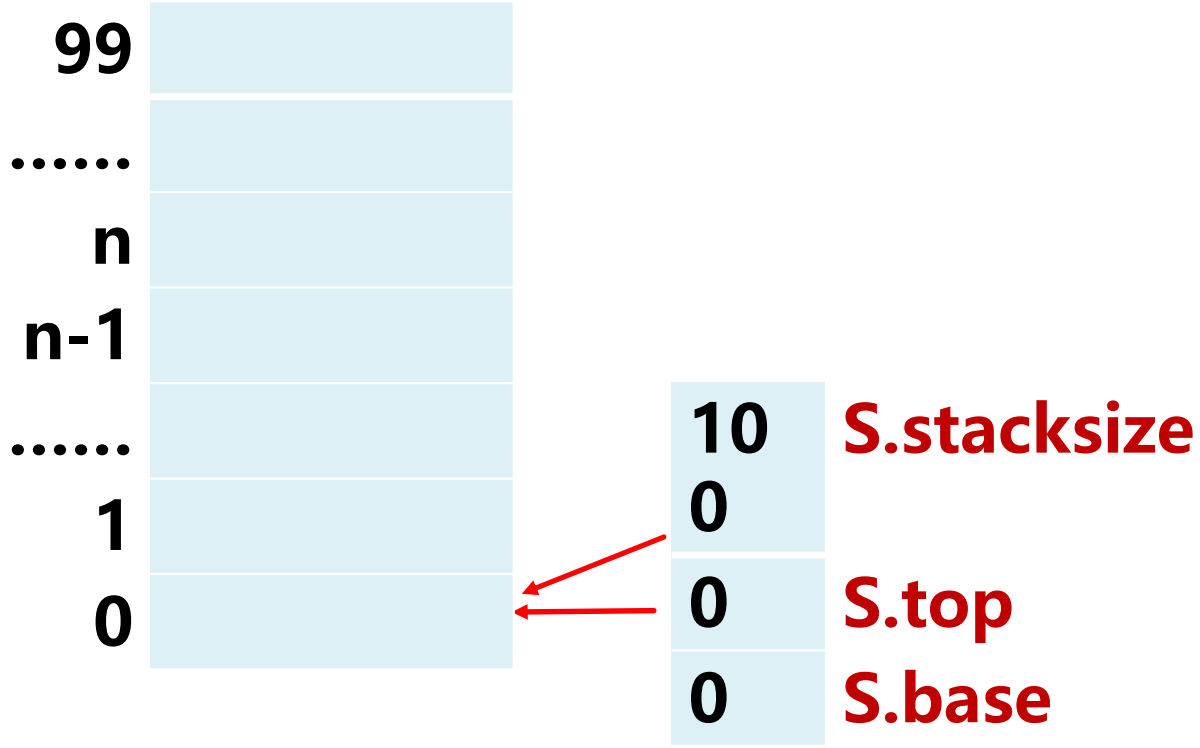
B C D E进栈

二、顺序栈基本操作的算法

1）初始化操作 InitStack（SqStack &S）

参数：S是存放栈的结构变量

功能：建一个空栈S

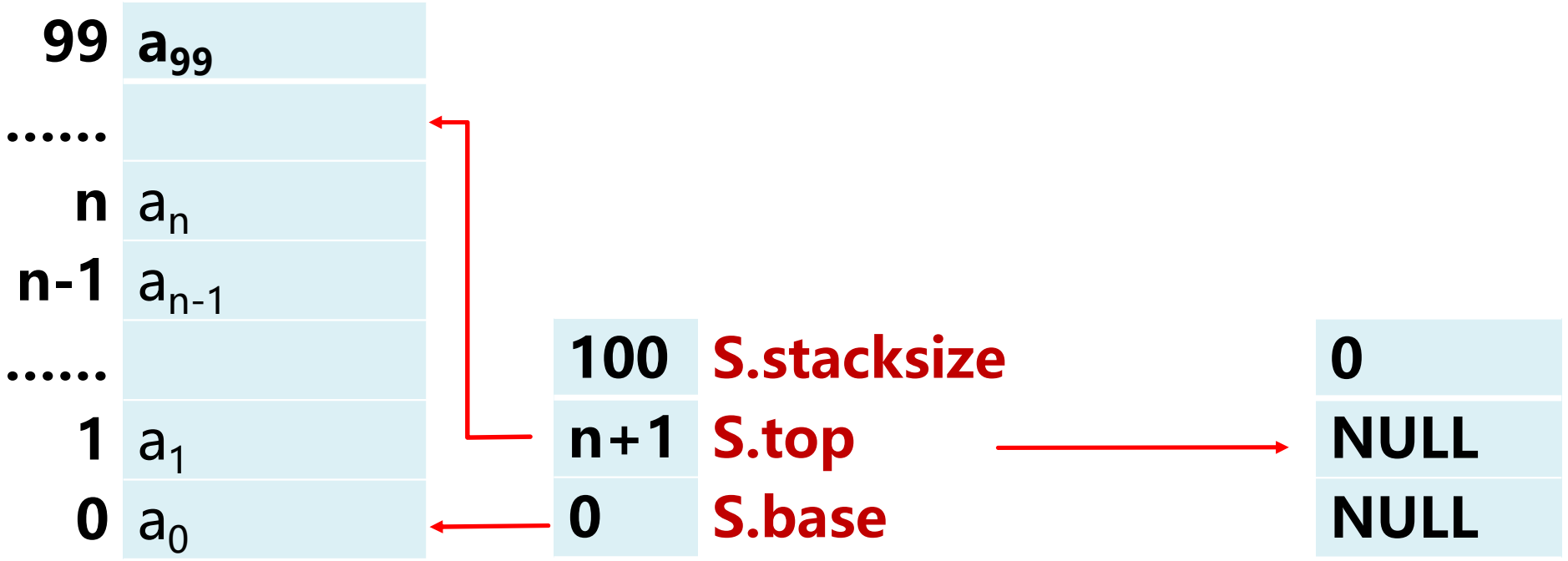


初始化操作 InitStack (SqStack &S)

```
Status InitStack ( SqStack  &S ) //构造一个空栈S
 {   S.base = ( ElemType *)   malloc
                                    ( STACK_INIT_SIZE * sizeof(ElemType) );
     //为顺序栈动态分配存储空间
     if ( ! S. base)          exit( OVERFLOW );   //分配失败
     S.top = S.base;
     S.stacksize = STACK_INIT_SIZE;
     return OK;
}  // InitStack
```

销毁栈操作 DestroyStack(SqStack &S)

功能：销毁一个已存在的栈

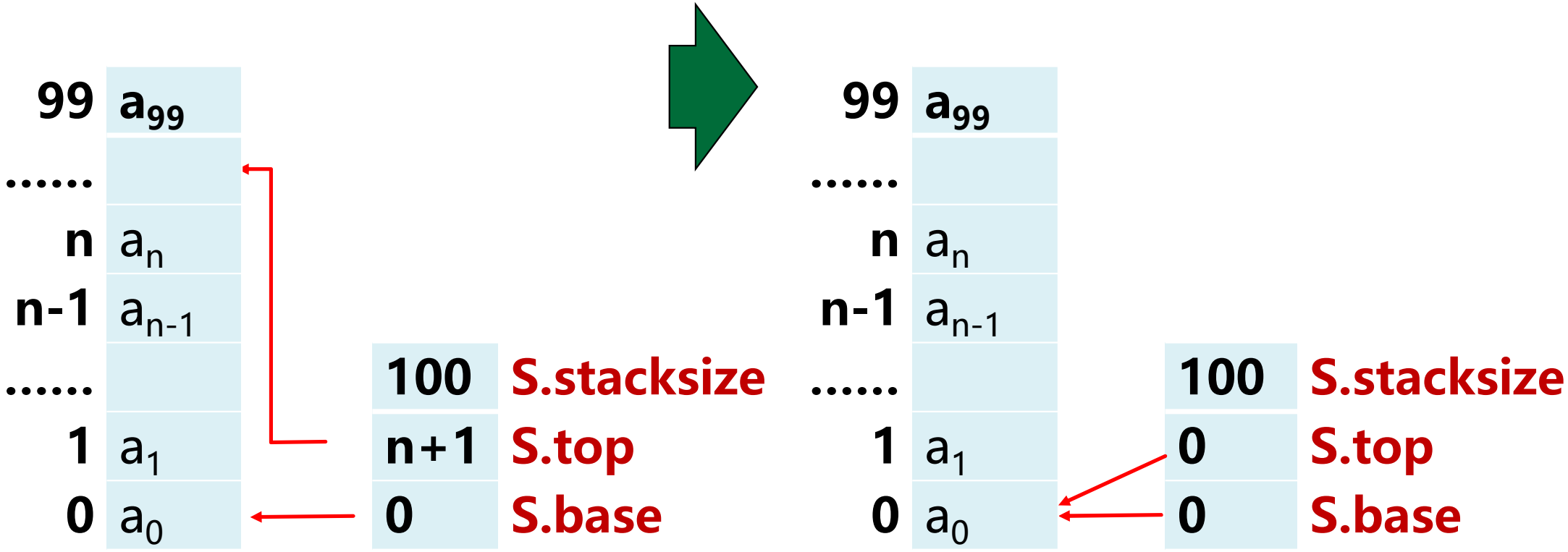


销毁栈操作 DestroyStack(SqStack &S)

```
Status DetroyStack ( SqStack  &S )
 {  if ( ! S.base )
        return ERROR;
              //若栈未建立（尚未分配栈空间）
     free ( S.base );                          //回收栈空间
     S.base = S.top = NULL;
     S.stacksize = 0;
     return OK;
}  //DetroyStack
```

置空栈操作ClearStack (SqStack &S)

功能：将栈S置为空栈



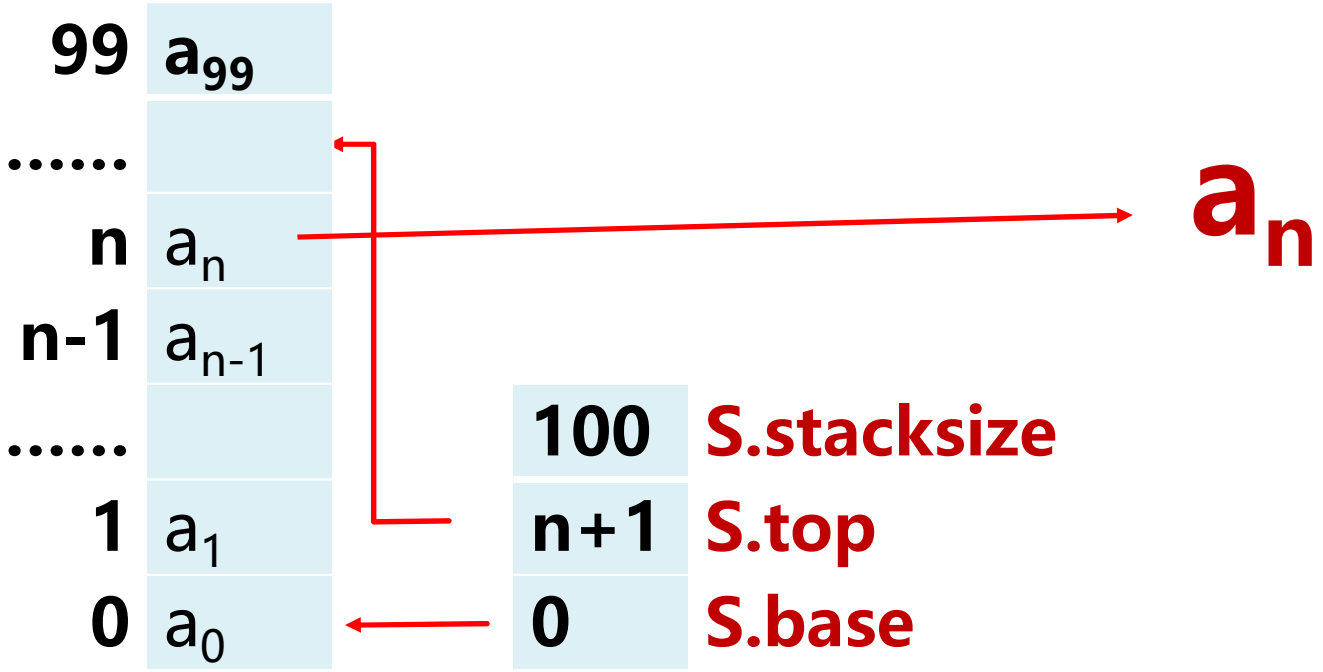
置空栈操作ClearStack (SqStack &S)

```
Status ClearStack ( SqStack &S )
 {  if ( ! S.base )
          return ERROR;    // 若栈未建立（尚未分配栈空间）
     S.top = S.base;
     return OK;
}  //ClearStack
```

取栈顶元素操作

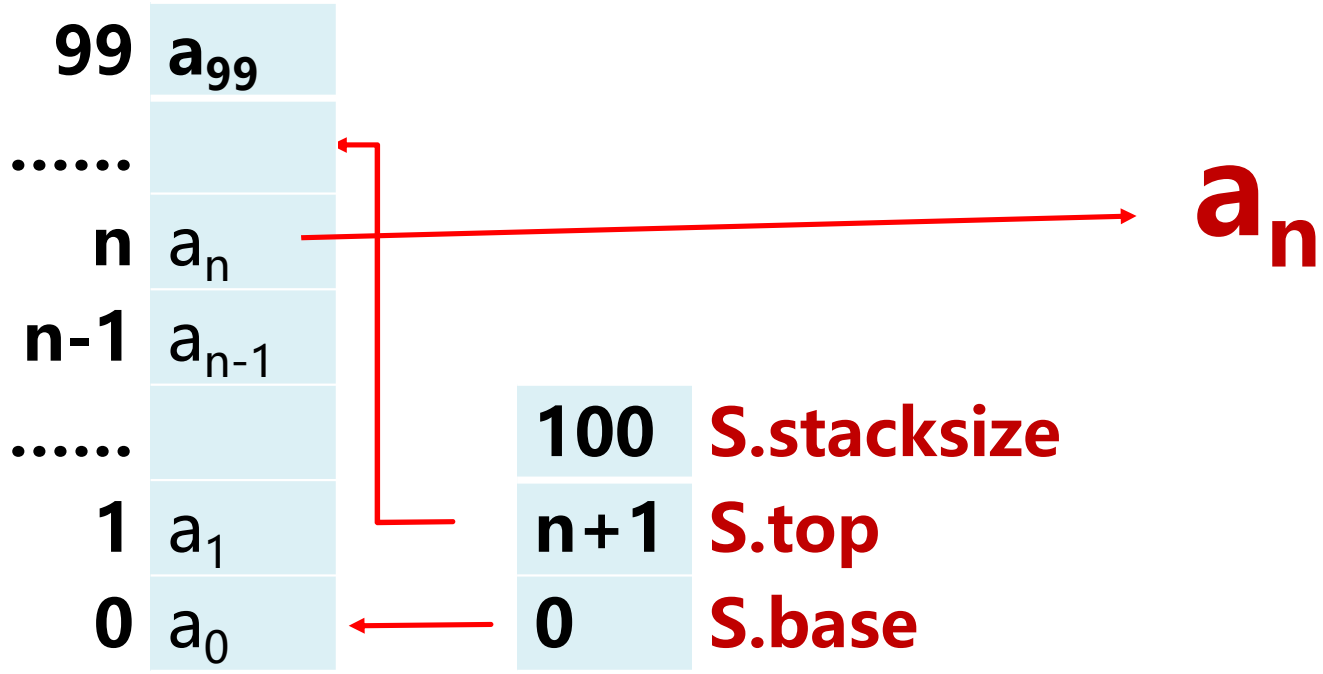
GetTop（SqStack S，ElemType &e）

功能：取栈顶元素，并用e返回



取栈顶元素操作

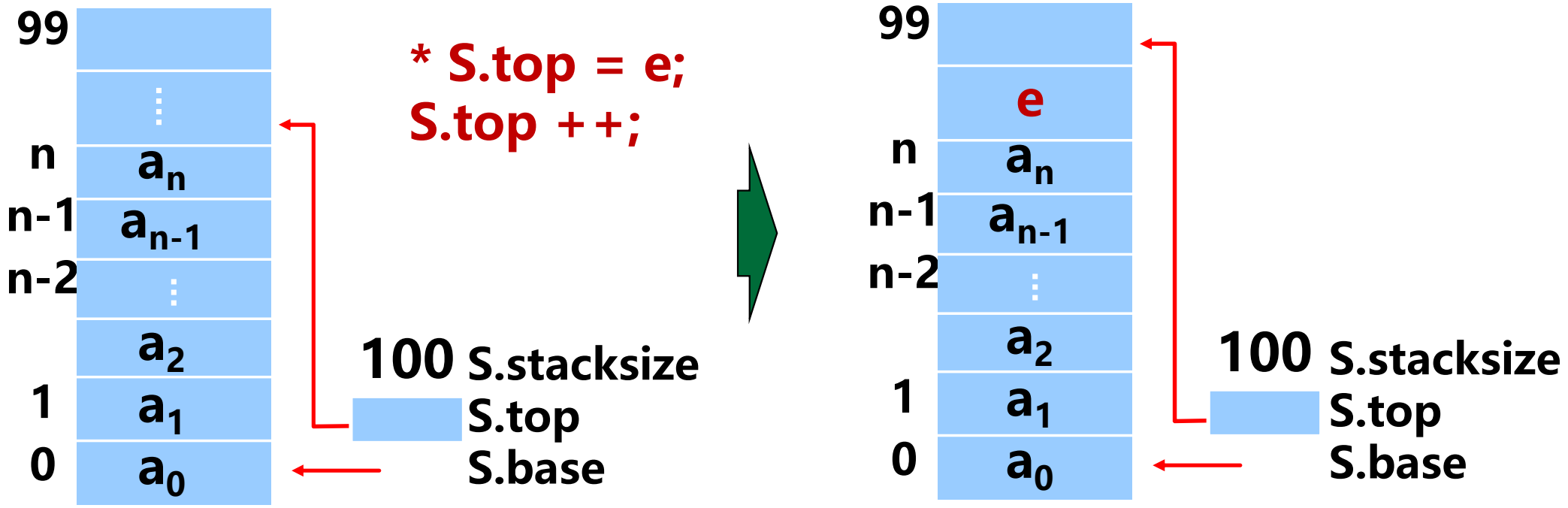
```
Status GetTop ( SqStack S, ElemType &e )
{  if ( S.top==S.base )
        return ERROR;        //栈空
    e = *(S.top-1);
    return OK;
}  //GetTop
```



进栈操作

Push (SqStack &S, ElemType e)

功能：元素 e 进栈。



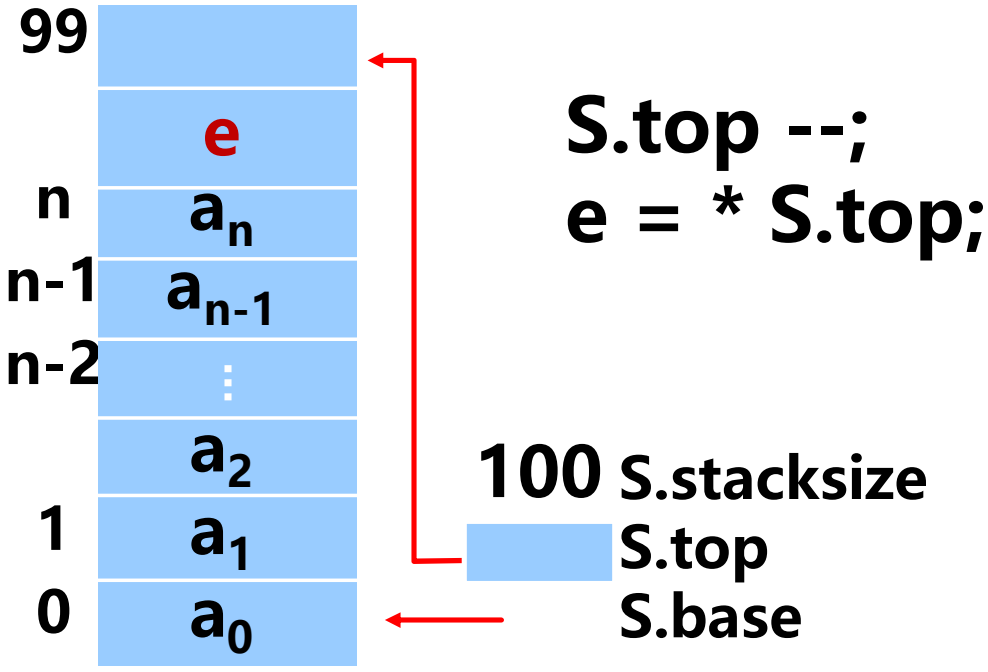
进栈操作

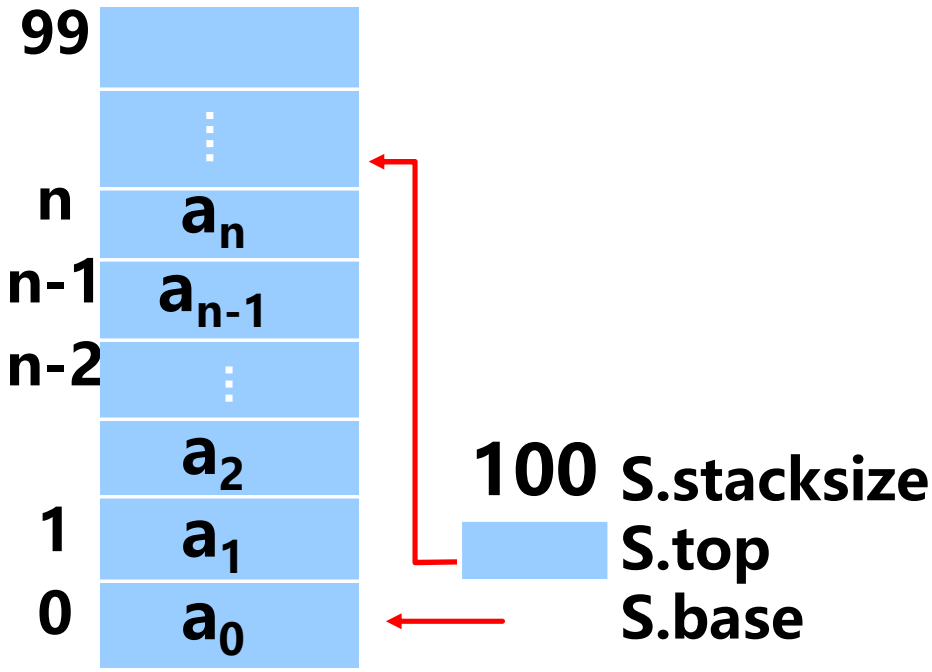
```
Status Push ( SqStack &S, ElemType e )
{   //将元素e插入栈中，使其成为新的栈顶元素
    if ( S.top-S.base>=S.stacksize ) // 若栈满则追加存储空间
    {  S.base =  (ElemType * ) realloc ( S.base,
(S.stacksize +STACKINCREMENT) * sizeof(ElemType));
        if  ( ! S. base )    exit(OVERFLOW);   //存储分配失败
        S.top = S.base + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    * S.top ++ = e;   //元素e 插入栈顶，后修改栈顶指针
    return OK;
}  //Push
```

出栈操作

Pop (SqStack &S, ElemType &e)

功能：栈顶元素退栈，并用 e 返回。





出栈操作

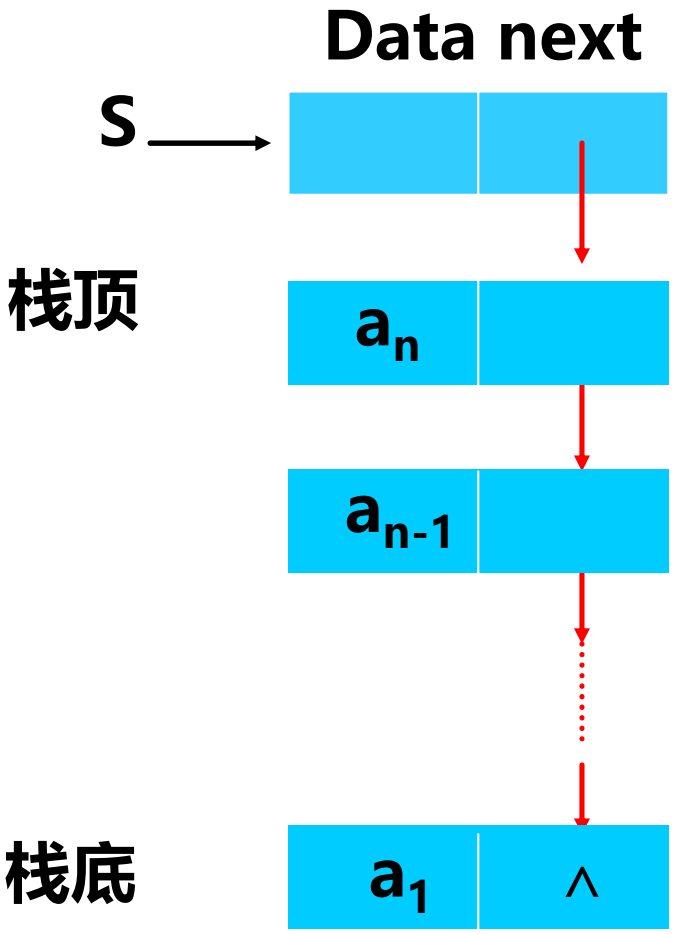
```
Status  Pop ( SqStack &S,  ElemType &e )
 {  if ( S.top==S.base )
          return ERROR;     // 栈空，下溢
     e = * -- S.top;   // 相当于--S.top; e=*S.top;
     return OK;
}  //Pop
```

另一种约定：

栈顶指针指向栈顶元素。该如何处理？

3.1.3 栈的链式存储和实现

在前面学习了线性链表的插入、删除操作算法，不难写出链式栈初始化、进栈、出栈等操作的算法。



顺序栈和链式栈的比较

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

栈的特点：后进先出

体现了元素之间的透明性

时间效率

所有操作都只需常数时间

顺序栈和链式栈在时间效率上难分伯仲

空间效率

顺序栈须说明一个固定的长度

链式栈的长度可变，但增加结构性开销

顺序栈和链式栈的比较

实际应用中，顺序栈比链式栈用得更广泛

顺序栈容易根据栈顶位置，进行相对位移，快速定位并读取栈的内部元素

顺序栈读取内部元素的时间为O(1)，而链式栈则需要沿着指针链游走，显然慢些，读取第k个元素需要时间为O(k)

一般来说，栈不允许“读取内部元素”，只能在栈顶操作

3.2 栈的应用

1. 数制转换
2. 括号匹配的检验
3. 行编辑程序
4. 迷宫求解
5. 表达式求值

例一、 数制转换

算法基于原理：

$N = (N \text{ div } d) \times d + N \bmod d$

例如： $(1348)_{10} = (?)_8$

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

例一、 数制转换

算法思路：

用栈实现先计算，后输出

操作步骤

1. 初始化栈，InitStack(S);
2. 依次计算当前的d进制数，并将其压栈
   Push(S, N % 8); N = N/8;
3. 依次从栈中取出计算的结果，并输出
   Pop(S,e);
4. 删除栈 DestroyStack(S);

例一、 数制转换

```
void conversion () {
    InitStack(S); //步骤1： 初始化栈
        scanf ("%d",N);
        while (N !=0 ) {   //步骤2： 余数压栈
            Push(S, N % 8);        N = N/8;
        }
        while (!StackEmpty(S)) { //步骤3出栈并输出
            Pop(S,e);    printf ( "%d", e );
        }
        DestroyStack(S); // 步骤4： 删除栈

} // conversion
```

例二、 括号匹配的检验

正确的格式：

([] ()）

[([ ] [ ])]

错误的格式：

[( ])

([( )) 或 (()])

判断是否正确方法

检验括号是否匹配

匹配：按照最近匹配原则

匹配：按照期待的急迫程度

例二、 括号匹配的检验

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：
   1. 若栈空，则表明表达式中匹配正确；
   2. 否则表明“左括弧”有余。

例三、行编辑程序问题

在接受用户输入时，以一行作为基本单位
允许用户在输入中及时更正。
方法是：
设立一个缓冲区，接受用户输入的一行字符；
然后逐行存入用户数据区；
假设“#”为退格符，“@”为退行符。
例：
whli##ilr#e（s#*s);
putcha@putchar(*s=#++);
输出:while (*s)　　putchar(*s++);

例三、行编辑程序问题

算法思路：

1、初始化栈

2、依次接受用户输入的字符

如果当前的字符是普通字符，则压栈

如果是＃，则删除栈顶字符

如果是@，则将栈清空

3、将从栈底到栈顶的字符传送至调用过程的数据区

4、删除栈

例三、行编辑程序问题

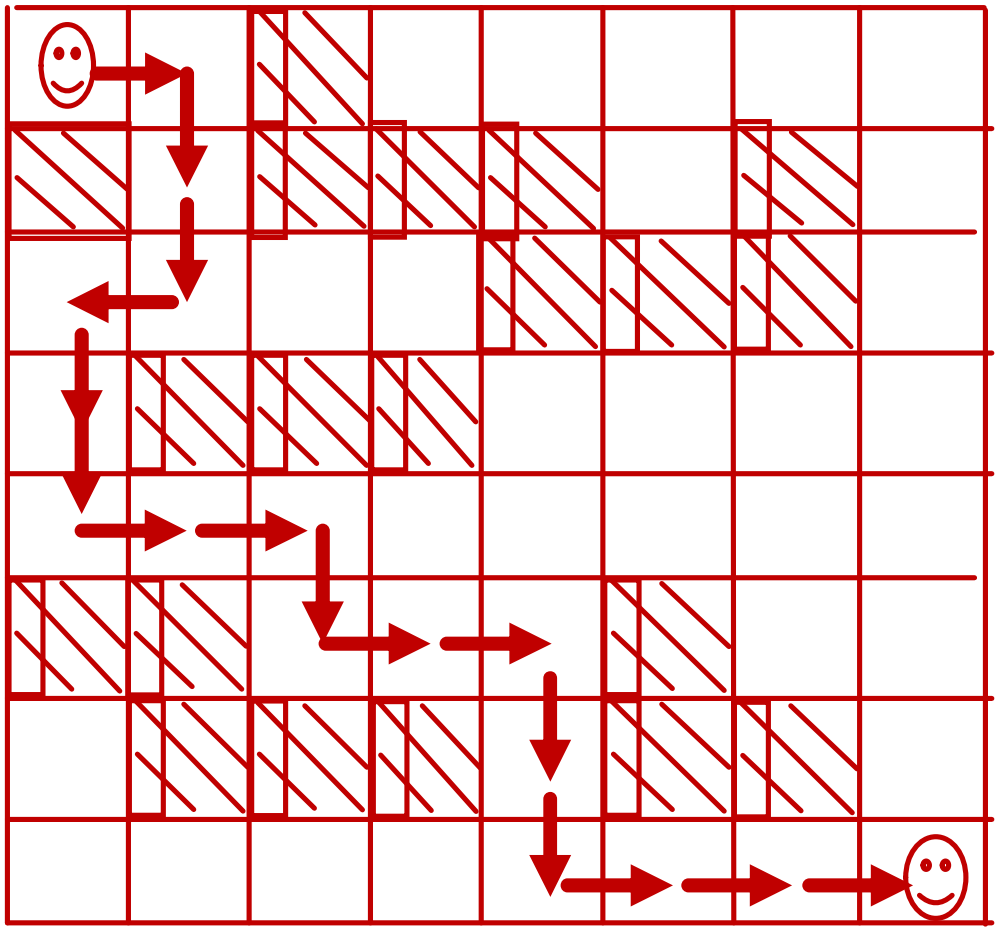
```
ch = getchar();  // 从终端接一个字符
while (ch != EOF && ch != '\n') {
        //EOF为全文结束符

    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;
DestroyStack(S); //
```

迷宫求解

入口



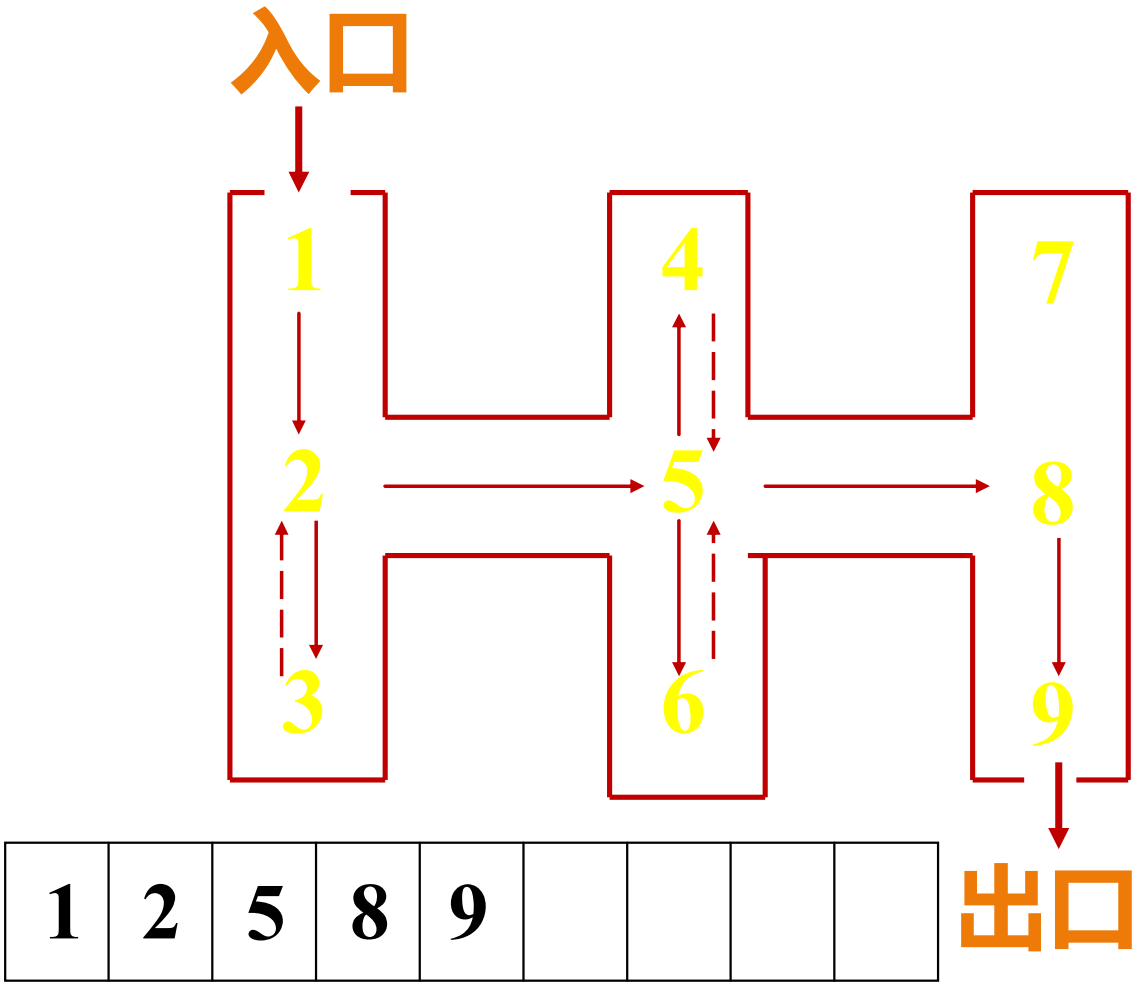
出口

求迷宫路径算法

若当前位置“可通”，则纳入路径，继续前进；

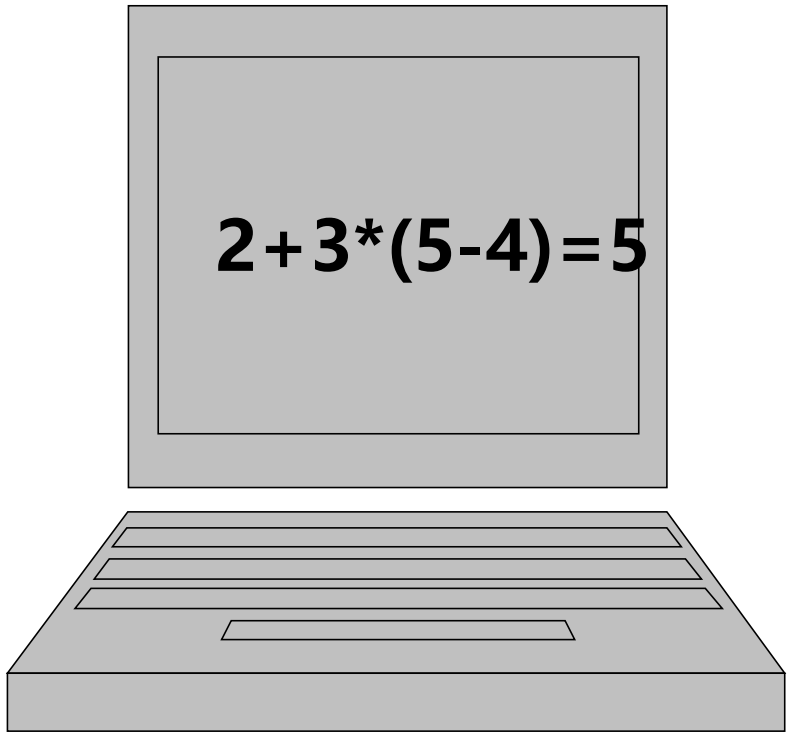
若当前位置“不可通”，后退一个位置，换方向继续探索；

若四周“均已探索”，则后退一步，将当前位置从路径中删除出去，换方向继续探索。



栈的应用举例 - 表达式求值

从键盘一次性输入一串算术表达式，给出计算结果。



栈的应用举例 - 表达式求值

限于二元运算符的表达式定义:

表达式 => (操作数) + (运算符) + (操作数)

操作数(operand) => 简单变量 | 表达式

简单变量 => 标识符 | 无符号整数

运算符(operator)

表达式的三种标识方法：Exp = S1 + OP + S2

OP + S1 + S2　　为前缀表示法 –波兰表示法

S1 + OP + S2　　为中缀表示法

S1 + S2 + OP　　为后缀表示法-逆波兰表示法

（RPN：Reverse Polish notation，波兰逻辑学家J.Lukasiewicz于1929年提出。）

栈的应用举例 - 中缀表达式的求值

3 2 1 4

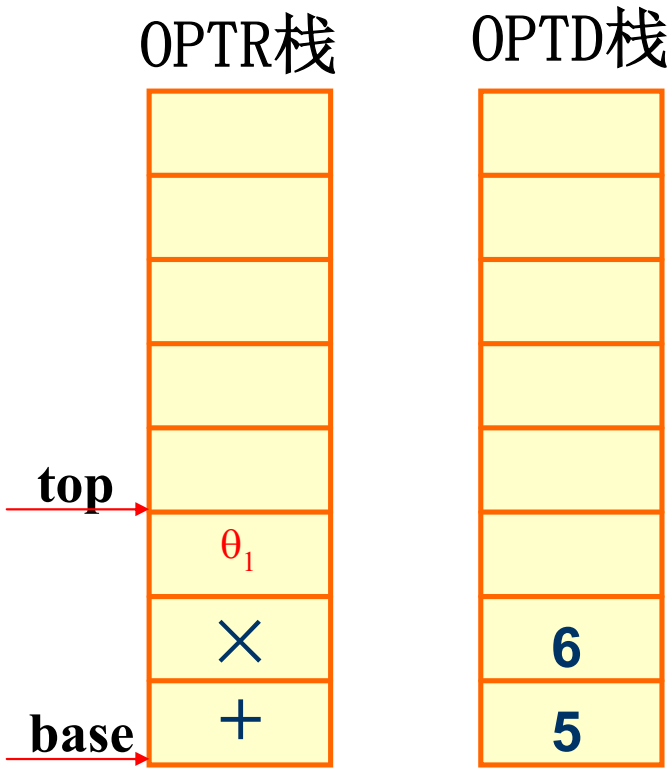
5 + 6 × (1 + 2) – 4

= 5 + 6 × 3 - 4

= 5 + 18 - 4

= 23 - 4

=19



表达式求值

操作数

( )、#

- 2）表达式的构成 操作数+运算符+界符

+、-、*、/

- 3）表达式的求值：
- 例：5+6×(1+2)-4
- 按照四则运算法则，上述表达式的计算过程为：
- 5+6×(1+2)-4 = 5+6×3-4 = 5+18-4 = 23-4 = 19

1 2 3 4

算符优先关系表

- 4)　表达式中任何相邻运算符 θ_1、运算符θ_2 的优先关系有：
  $\theta_1 < \theta_2$：θ_1的优先级 低于 θ_2
  $\theta_1 = \theta_2$：θ_1的优先级 等于 θ_2
  $\theta_1 > \theta_2$：θ_1的优先级 高于 θ_2

- **注：θ_1、θ_2是相邻算符，θ_1在左，θ_2在右**

算符优先关系表

- 表达式中任何相邻运算符
- θ_1、θ_2 的优先关系有：
  $\theta_1 < \theta_2$：θ_1的优先级 低于 θ_2
  $\theta_1 = \theta_2$：θ_1的优先级 等于 θ_2
  $\theta_1 > \theta_2$：θ_1的优先级 高于 θ_2

	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	=
#	<	<	<	<	<		=

表达式运算

在算符优先算法中，建立了两个工作栈。一个是OPTR栈，用以保存运算符；一个是OPND栈，用以保存操作数或运算结果。

算法的基本思想是：

1、首先置操作数栈为空栈，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，则与OPTR栈的栈顶运算符比较优先级后作相应操作。

直至整个表达式求值完毕（即OPTR栈的栈顶元素和当前读入的字符均为“#”）。

表达式运算

5）算符优先算法　　5 + 4 x (1 + 2) - 6

从左向右扫描表达式，**用两个栈分别保存扫描过程中遇到的操作数和运算符**：

遇操作数——保存；

遇运算符号θ_j——与前面的刚扫描过的运算符θ_i比较：

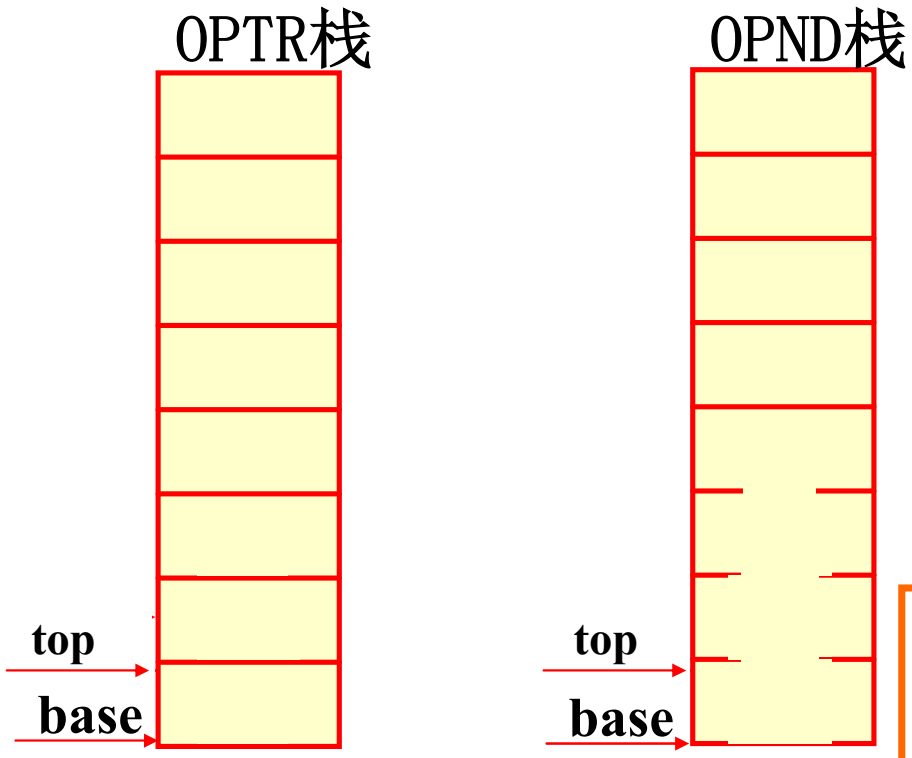
若$\theta_i < \theta_j$则保存θ_j（因此已保存的运算符的优先关系）

若$\theta_i > \theta_j$则说明θ_i是已扫描的运算符中优先级最高者，可进行运算

若$\theta_i = \theta_j$则说明括号内的式子已计算完，需要消去括号

表达式求值示意图（算符优先算法）

读入表达式：# 5+6 ×(1+ 2)−4 #　=19



1+2=3

6×3=18

5+18=23

23-4=19

程序结束标志？

OPTR栈顶元素为 #
当前字符也为 #

算符优先算法操作步骤

1. 初始化运算符OPTR栈和操作数OPND栈
2. 将‘#’压入OPTR栈
3. 依次读入每个字符，直到表达式求值完毕
   若是操作数，则压入OPND栈
   若是运算符，则和OPTR栈顶元素比较优先级
4. 返回运算结果

1. InitStack(OPTR); InitStack(OPND);
2. Push (OPTR, #);
3. while(c!=‘#’ || GetTop(OPTR)!=‘#’)
   - if (!In (c, OP)) Push(OPND, c);
   - else{ switch (Precede(GetTop(OPTR), c) …}
4. return GetTop(OPND);

算符优先算法操作步骤（续）

```
switch (Precede(GetTop(OPTR)，c)
    case <：//栈顶元素优先级低,压栈并接收下一字符
        Push(OPTR，c); c = getchar();
    case =:      // 脱括号并接收下一字符
        Pop(OPTR, x); c = getchar();
    case >: //退栈，并将运算结果压栈
        Pop(OPTR，theta);//取出运算符
        Pop (OPND, b);    Pop(OPND, a);//取出操作数
        Push(OPND, Operate(a, theta, b)); //结果压栈
```

算符优先算法伪代码

```
OperandType EvaluateExpression( )
{ //算术表达式求值的算符优先算法。设OPTR和OPND分别为运算符栈和操作数栈，OP为运算符集合。
    InitStack(OPTR); InitStack(OPND); //步骤1
    Push (OPTR, #); c=getchar( );          //步骤2
    while(c!=‘ #’ || GetTop(OPTR)!=‘#’){ //步骤3
      if (!In (c, OP)) //操作数进栈OPND
        { Push(OPND,  c); c=getchar( ); }
      else
        {
```

```
            switch (Precede(GetTop(OPTR),  c)
        {   case <:     //栈顶元素优先级低
                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，并将运算结果压栈
                Pop(OPTR,  theta);
                Pop (OPND, b);    Pop(OPND, a);
                Push(OPND, Operate(a, theta, b));

          }// switch
       }// if (!In (c, OP))
    }//while
    return GetTop(OPND);              //步骤4
} //EvaluateExpression
```

3.3 栈与递归

递归在数学和程序设计等许多领域中都会用到。

任何一个递归程序都可以通过非递归程序实现。

栈在实现函数递归调用中所发挥的作用

栈的一个最广泛的应用：高级语言运行时环境(runtime)提供的函数调用机制。运行时环境(runtime)指的是目标计算机上用来管理存储器并保存指令执行过程中所需信息的寄存器及存储器的结构。

3.3 栈与递归

函数调用过程

在非递归情况下，数据区的分配可以在程序运行前进行，直到整个程序运行结束才释放，这种分配称为**静态分配**。

采用静态分配时，函数的调用和返回处理比较简单，不需要每次分配和释放被调函数的数据区。

3.3 栈与递归

函数的递归调用过程

在递归调用的情况下，被调函数的局部变量不能静态地分配某些固定单元，而必须每调用一次就分配一份新的局部变量，以存放当前函数所使用的数据，当函数结束返回时随即释放。

故其存储分配只能在执行调用时（程序运行过程中）才能进行，即所谓的**动态分配。**

在内存中要开辟一个称为运行栈（runtime stack）的足够大的动态区，以处理运行数据。

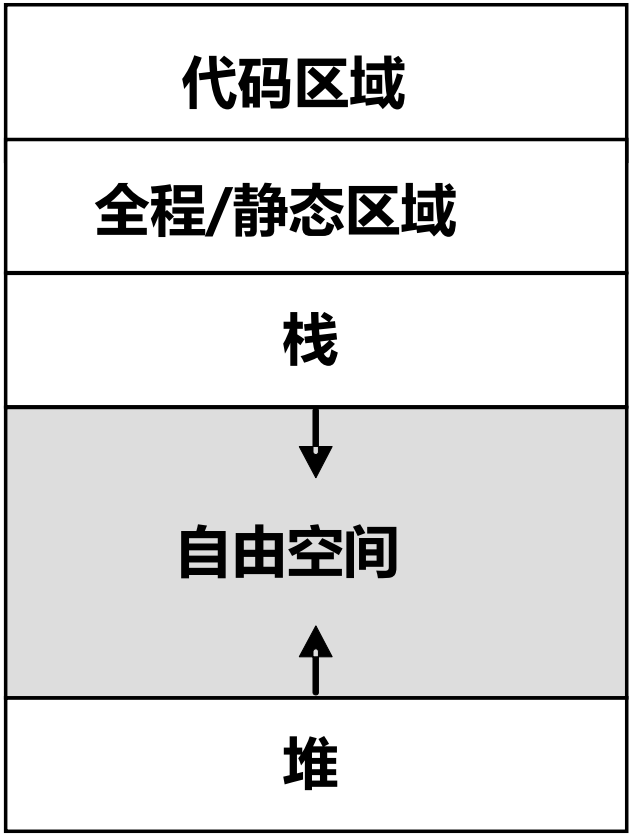
3.3 栈与递归

函数运行时的动态分配过程

用作动态数据分配的存储区可按多种方式组织。典型的组织是将这个存储器分为**栈（stack）**区域和**堆（heap）**区域：

栈区域用于分配发生在后进先出**LIFO**风格中的数据（诸如函数的调用）。

堆区域则用于不符合LIFO（诸如指针的分配）的动态分配。

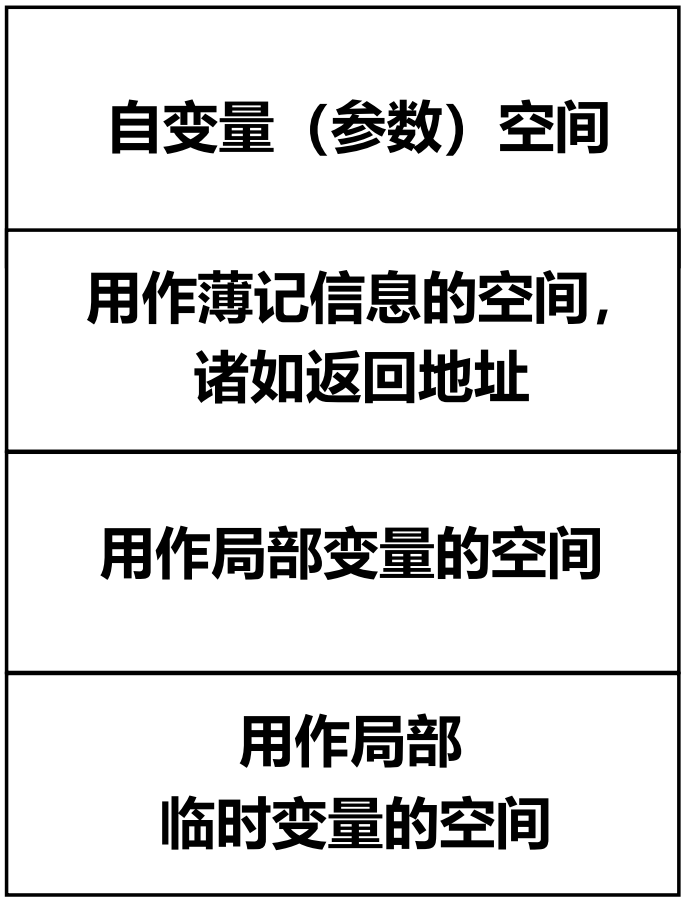


3.3 栈与递归

函数活动记录（activation record）

是动态存储分配中一个重要的单元。

当调用或激活函数时，函数的活动记录包含了为其局部数据分配的存储空间。



3.3 栈与递归

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

运行栈的动态变化

每次调用时，执行进栈操作，把被调函数的活动信息压入栈中，即当进行一个新的函数调用时，都要在栈的顶部为新的活动记录分配空间。

在每次从函数返回时，执行出栈操作，释放本次的活动记录，恢复到上次调用所分配的数据区中。

被调函数中变量地址全部采用相对于栈顶的相对地址来表示。

3.3 栈与递归

一个函数在运行栈上可以有若干不同的活动记录，每个活动记录都代表了一次不同的调用

对于递归函数来说，递归的深度就决定了其在运行栈中活动记录的数目。

当函数递归调用时，函数体的同一个局部变量，在不同的递归层次要分配不同的存储空间，放在内部栈的不同位置。

函数调用时的三个步骤

调用函数发送调用信息。调用信息包括调用方要传送给被调方的信息，诸如实参、返回地址等。

为被调函数分配需要的局部数据区，用来存放被调方定义的局部变量、形参变量（存放实参）、返回地址等，并接受调用方传送来的调用信息。

调用方暂停，把计算控制转到被调方，即自动转移到被调用的函数的程序入口。

当被调方结束运行，返回到调用方时，其返回处理也分解为三步进行

1、传送返回信息，包括被调方要传回给调用方的信息，诸如计算结果等。

2、释放分配给被调方的数据区。

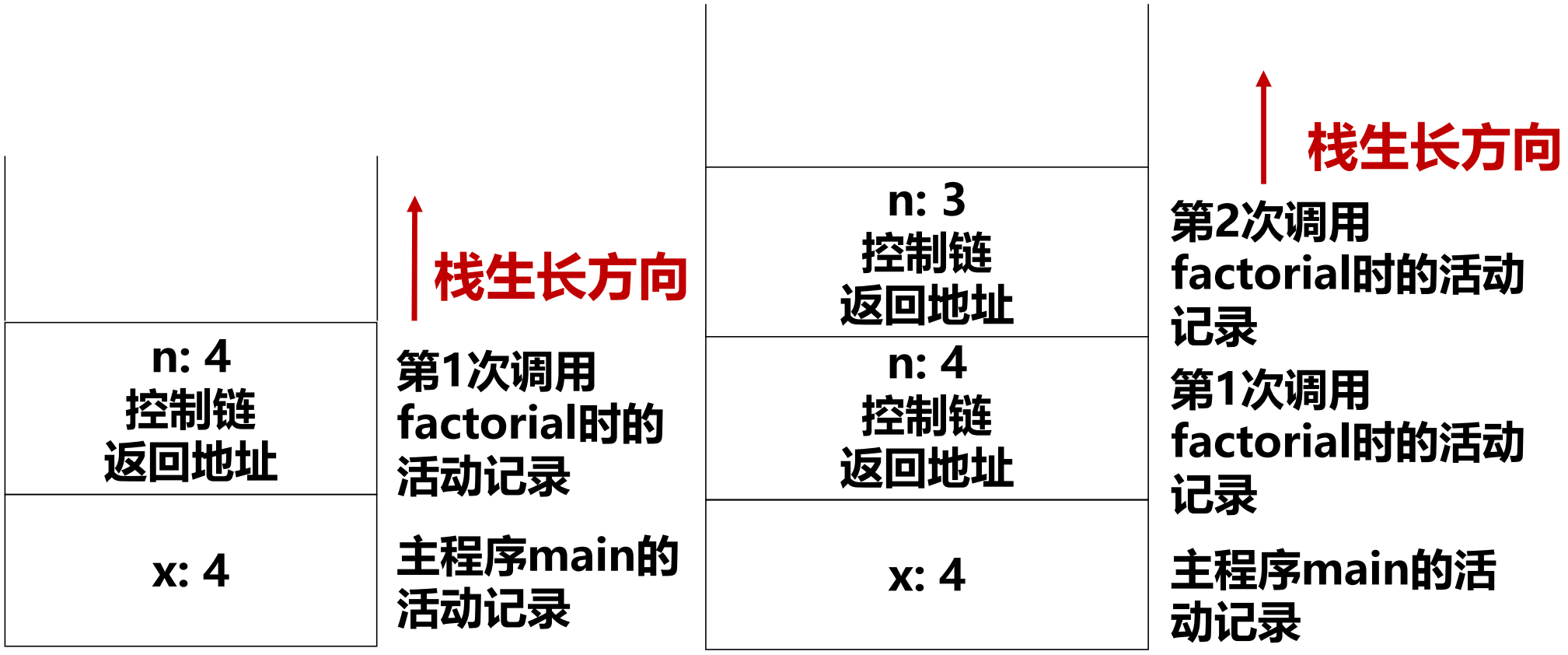
3、按返回地址把控制转回调用方。

3.3 栈与递归

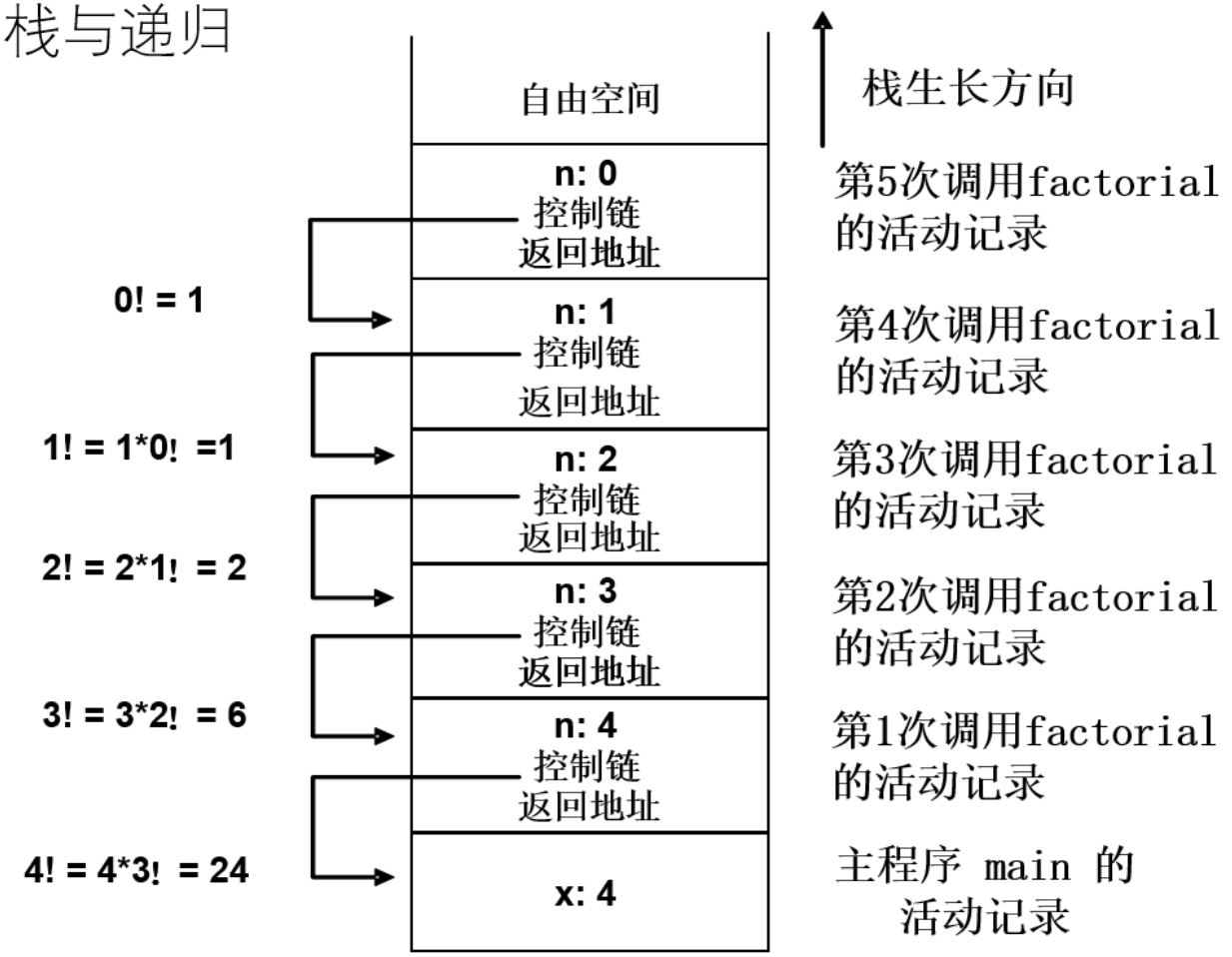
以阶乘为例：

```
#include <stdio.h>
main( ) { int x;
    scanf(" %d", &x);
    printf(" %d\n", factorial(4));
}
long factorial( long n ) {
   if ( n == 0 )
      return 1;
    else
      return n * factorial( n-1); // 递归调用
}
```

递归函数调用过程


栈生长方向
n: 4
控制链
返回地址
x: 4
第1次调用
factorial时的
活动记录
主程序main的
活动记录
栈生长方向
n: 3
控制链
返回地址
n: 4
控制链
返回地址
x: 4
第2次调用
factorial时的活动
记录
第1次调用
factorial时的活动
记录
主程序main的活
动记录

递归函数调用过程


自由空间
n: 0
控制链
返回地址
n: 1
控制链
返回地址
n: 2
控制链
返回地址
n: 3
控制链
返回地址
n: 4
控制链
返回地址
x: 4
0! = 1
1! = 1*0! =1
2! = 2*1! = 2
3! = 3*2! = 6
4! = 4*3! = 24
栈生长方向
第5次调用factorial的活动记录
第4次调用factorial的活动记录
第3次调用factorial的活动记录
第2次调用factorial的活动记录
第1次调用factorial的活动记录
主程序 main 的活动记录

栈的应用举例

假定元素进栈的顺序是1234，那么出栈的顺序可否为3142;

若用S表示进栈，X表示出栈，假定元素入栈的顺序是1234，为了得到1342的出栈顺序，请给出对应的SX操作序列。

一个栈的进栈序列为123...N，可以得到出栈序列p1p2...pN，若p1为3，则p2的可能取值的个数有多少个?

一个栈的进栈序列为123...N，则所有可能的出栈的序列有多少个?

栈的应用举例

我们把n个元素的出栈个数的记为f(n), 那么对于1,2,3, 我们很容易得出：

f(1) = 1　　//即 1

f(2) = 2　　//即 12、21

f(3) = 5　　//即 123、132、213、321、231

来考虑f(4), 我们给4个出栈元素编号为a,b,c,d, 那么考虑：元素a可能出现在1号位置，2号位置，3号位置和4号位置(很容易理解，一共就4个位置，比如abcd,元素a就在1号位置)。

栈的应用举例

- 1) 如果元素a在1号位置，那么只可能a进栈，马上出栈，此时还剩元素b、c、d等待操作，就是子问题f(3)；
- 2) 如果元素a在2号位置，那么一定有一个元素比a先出栈，即有f(1)种可能顺序（只能是b），还剩c、d，即f(2)，根据乘法原理，一共的顺序个数为f(1) * f(2)；
- 3) 如果元素a在3号位置，那么一定有两个元素比1先出栈，即有f(2)种可能顺序（只能是b、c），还剩d，即f(1)，根据乘法原理，一共的顺序个数为f(2) * f(1)；
- 4) 如果元素a在4号位置，那么一定是a先进栈，最后出栈，那么元素b、c、d的出栈顺序即是此小问题的解，即f(3)；

栈的应用举例

结合所有情况，即f(4) = f(3) + f(2) * f(1) + f(1) * f(2) + f(3);

我们定义f(0) = 1；于是f(4)可以重新写为：
f(4) = f(0)*f(3) + f(1)*f(2) + f(2) * f(1) + f(3)*f(0)

推广到n：f(n) = f(0)*f(n-1) + f(1)*f(n-2) + f(2)*f(n-3)+..... + f(n-1)*f(0)

$$f(n) = \sum_{i=0}^{n-1} f(i)f(n-1-i)$$

栈的应用举例

对于每一个数来说，必须进栈一次、出栈一次。我们把进栈设为状态‘1’，出栈设为状态‘0’。

n个数的所有状态对应n个1和n个0组成的2n位二进制数。

假定等待入栈的操作数按照1..n的顺序排列，

在出栈的过程中，入栈的操作数b大于等于出栈的操作数a(a≤b)，

输出序列的总数目=由左而右扫描由n个1和n个0组成的2n位二进制数，其中：任何一个位置上之前出现的1的累计数不小于0的组合数相加是所有可行方案的总数。

栈的应用举例

不符合要求的数的特征：

由左而右扫描时，必然在某一奇数位2m+1位上首先出现m+1个0的累计数和m个1的累计数，此后的2(n-m)-1位上有n-m个 1和n-m-1个0。如若把后面这2(n-m)-1位上的0和1互换，使之成为n-m个0和n-m-1个1，结果得1个由n+1个0和n-1个1组成的2n位数，即一个不合要求的数对应于一个由n+1个0和n-1个1组成的排列。

同理：

由左而右扫描时，必然在某一偶数位2m任何一个由m+1个0和m-1个1组成的2m位二进制数，由于0的个数多2个，2m为偶数，故必在某一个奇数位上出现0的累计数超过1的累计数。

因此，不符合要求的2n位数与n+1个0，n - 1个1组成的排列一一对应。

栈的应用举例

输出序列的总数目：
c(2n,n)-c(2n,n+1)=c(2n,n)/(n+1)，

卡特兰数

=(2n)!/(n!*n!)/(n+1)

$$C_{2n}^{n}/(n+1)$$

栈的应用举例

(1)买票找零

有2n个人排成一行进入剧场。入场费5元。其中只有n个人有一张5元钞票，另外n人只有10元钞票，剧院无其它钞票，问有多少种方法使得只要有10元的人买票，售票处就有5元的钞票找零？(将持5元者到达视作将5元入栈，持10元者到达视作使栈中某5元出栈)

(2) 洗碗

饭后，姐姐洗碗，妹妹把姐姐洗过的碗一个一个地放进碗橱摞成一摞。一共有n个不同的碗，洗前也是摞成一摞的，也许因为小妹贪玩而使碗拿进碗橱不及时，姐姐则把洗过的碗摞在旁边，问：小妹摞起的碗有多少种可能的方式?

通用的递归转非递归方法

- 尾递归

```
long  fact(int n){
   if (n<0) return 1;
   return n*fact(n-1);
}
```

```
int fun(int n){
        if (n==1) return 1;
        else return n+fun(n-1);
}
```

通用的递归转非递归方法

- 尾递归?

```
int FibonacciRecur(int n) {
  if (0==n) return 0;
  if (1==n) return 1;
  return FibonacciRecur(n-1)+FibonacciRecur(n-2);
}

int FibonacciTailRecur(int n, int acc1, int acc2) {
  if (0==n) return acc1;
  return FibonacciTailRecur(n-1, acc2, acc1+acc2);
}
```

通用的递归转非递归方法

- 设有一个背包，可以放入的物品重量为s，现有n件商品，重量分别为w_1，w_2……w_n,问能否从这些商品中选择若干个放入到背包中，使得其重量之和正好为s。如果存在这样的一组商品，则问题有解（true），否则问题无解（false）。

$$
knap(s,n)\begin{cases} true，当s=0 \\ false，当s<0 或 s>0 且 n<1 \\ knap(s-w_{n-1},n-1) \,||\, knap(s,n-1) \end{cases}
$$

通用的递归转非递归方法

- 设有一个背包，可以放入的物品重量为s，现有n件商品，重量分别为w_1，w_2……w_n,问能否从这些商品中选择若干个放入到背包中，使得其重量之和正好为s。如果存在这样的一组商品，则问题有解（true），否则问题无解（false）。

```
bool knap(int s,int n){
   if (0 == s) return true;
   if ( (s<0) || ( s>0 && n<1) )  return false;
   if (knap(s-w_{n-1},n-1)) {
         cout << w[n-1];
         return true;
   } else return knap(s,n-1);
}
```

通用的递归转非递归方法

- 设有一个背包，可以放入的物品重量为s，现有n件商品，重量分别为w_1，w_2……w_n,问能否从这些商品中选择若干个放入到背包中，使得其重量之和正好为s。如果存在这样的一组商品，则问题有解（true），否则问题无解（false）。

```
bool knap(int s,int n){
   if (0 == s) return true;
   if ( (s<0) || ( s>0 && n<1) )  return false;
   if (knap(s-w_{n-1},n-1)) {
         cout << w[n-1];
         return true;
   } else return knap(s,n-1);
}
```

通用的递归转非递归方法

```
bool knap(int s,int n){
   if (0 == s) return true;
   if ( (s<0) || ( s>0 && n<1) )  return false;
   if (knap(s-w_{n-1},n-1)) {
         cout << w[n-1];
         return true;
   } else return knap(s,n-1);
   ;
}
```

```
enum rdType {0,1,2};
public class knapNode {
   int s,n;
   rdType rd;
   bool   k;
}
```

```
Stack<knapNode> stack;
KnapNode temp,x;
bool knap(int s,int n){

    //整个函数的入口处
                                temp.s = s; temp.n = n; temp.rd = 0;
                                stack.push(temp);

                                lable0:
                                  stack.pop(&temp);
                                                        if (0 == temp.s ) {
                                                          temp.k = true;
                                                          stack.push(temp);
                                                          goto label3;
                                                        }
      if ((temp.s<0) || ( temp.s>0 && temp.n<1)  ) {
                                                          temp.k = false;
                                                          stack.push(temp);
                                                          goto label3;
                                                        }
      stack.push(temp);
                                                        // 第1个入口
      x.s = temp.s - w[temp.n - 1];
                                                        x.n = temp.n - 1;
                                                        x.rd = 1;
                                                        stack.push(x);
                                                        goto Lable0;
                                label1:
                                  stack.pop(&x);
                                                        if (temp.k == true) {
                                                          x.k =true;
                                                                                  stack.push(x);
                                                                                  cout << w[x.n-1] << endl;
                                                                                  goto label3;
                                                        }
      stack.push(x);
      temp.s=x.s
                                                        temp.n=x.n-1;
                                                        temp.rd = 2;
                                                        stack.push(temp);
                                                        goto label0;
                                label2:
      stack.pop(&x);
      x.k = temp.k;
                                                        stack.push(x);
                                label3:
      stack.push(&temp);
      switch(temp.rd) {

                                                          case 0: return temp.k;
                                                          case 1: goto label1;
                                                          case 2: goto label2;
                                                        }
}
```

通用的递归转非递归方法

1、设计一个工作栈，保存当前记录，包含函数中出现的所有参数，局部变量，返回语句标号，返回值等信息，类似操作系统所作的工作；

2、设置t+2个语句标号；label0标示入口第一个可执行语句；lablet+1 则放置在函数结尾出口处；labeli为第i个递归返回的语句；

3、增加非递归入口

4、替换第i个规则：push（）；goto label0；labeli：pop（x）...可能需要继续传递结果；

5、增加出口标记；

6、增加出口语句：switch

7、优化代码

递归转非递归的另一个例子

```
int exmp(int n) {
  if(n<2) return n+1;
  else return exmp(n/2) * exmp(n/4);
}
```

```
void exmp(int n,int& f)
{
    int u1,u2;
    if(n<2) f = n+1;
    else {
        exmp(n/2,u1);
        exmp(n/4,u2);
        f = u1*u2;
    }
}
```

```
typedef struct elem  //存储递归函数的现场信息
{
    int rd;         //返回语句的标号，rd=0~t+1
    int pn,pf;      //函数形参,pn表示参数n,pf表示参数f
    int q1,q2;      //局部变量,q1表示u1,q2表示u2
}ELEM;
class nonrec
{
    private:
        std::stack<ELEM> S;
    public:
        nonrec(void){}
        void replace1(int n,int& f);
        void replace2(int n,int& f);
};
void nonrec::replace1(int n,int& f)
{
    ELEM x,tmp;
    x.rd=3;                     //因为exmp内共调用2次递归子函数，t=2，所以td=t+1=3,压到栈底作监视哨
    x.pn=n;
    S.push(x);                  //调用最开始函数,递归的总入口。相当于调用exmp(7,f)
label0:
    if( (x = S.top()).pn < 2)    //处理递归出口,所有递归出口处需要增加语句goto label t+1。这也是递归语句第一条可执行语句
    {
        S.pop();
        x.pf = x.pn + 1;        //获得函数的解pf
        S.push(x);
        goto label3;            //因为递归出口语句执行完后需要处理函数返回,而lable t+1是用来处理函数返回需要做的工作的，所以需要goto lable3
      }

      x.rd = 1;                 //调用第一个递归函数,位于label0的后面，所以如果不满足递归出口会不断调用这里，直到满足递归出口
      x.pn = (int)(x.pn/2);
      S.push(x);                //一次调用使用一个堆栈数据
      goto label0;              //调用后开始进入函数内部，由于函数的第一条执行语句位于lable0，所以需要goto label0


label1: tmp = S.top();          //label1处理第1个递归函数返回时需要进行的处理，通常是pop自己的数据，然后把计算结果放到调用者对应的数据内
        S.pop();
        x = S.top();
        S.pop();
        x.q1 = tmp.pf;          //获取第1个递归函数计算的结果，并回传给上层函数的q1
        S.push(x);

        x.rd = 2;               //调用第二个递归函数
        x.pn = (int)(x.pn/4);
        S.push(x);
        goto label0;

label2: tmp = S.top();          //从第二个递归函数中返回
        S.pop();
        x = S.top();
        S.pop();
        x.q2 = tmp.pf;
        x.pf = x.q1 * x.q2;
        S.push(x);
label3:                         //递归出口（label0)结束后会调用这里
        switch((x=S.top()).rd)
        {
            case 1:
            goto label1;
            break;
            case 2:
            goto label2;
            break;
            case 3:             //t+1处的label：表示整个函数结束
            tmp = S.top();
            S.pop();
            f = tmp.pf;         //最终的计算结果
            break;
            default:
            break;
        }
}
```

3.3 队列

队列是仅能在**表头进行删除**，**表尾进行插入**的线性表。

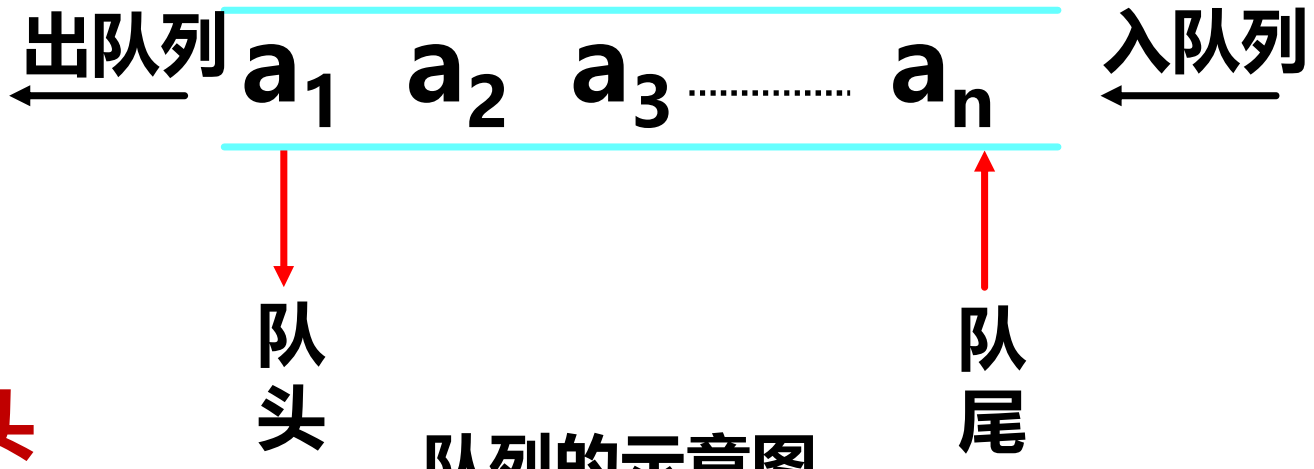
$(a, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$

删除 ↓　　↑ 插入

能进行插入的一端称为**队尾(rear)**，能进行删除的一端称为**队头(front)**。

称插入操作为**入队(enqueue)**，删除操作为**出队(dequeue)**。

队列的特点：先进先出（FIFO）



队列的示意图

第一个入队的元素在**队头**

最后一个入队的元素在**队尾**

第一个出队的元素为队头元素

最后一个出队的元素为队尾元素

队列的基本操作-1

1）初始化操作 InitQueue(&Q)

功能：构造一个空队列Q。

2）销毁操作 DestroyQueue(&Q)

功能：销毁已存在队列Q。

3）置空操作 ClearQueue(&Q)

功能：将队列Q置为空队列。

4）出队操作 DeQueue(&Q,&e)

功能：删除Q的队头元素。

队列的基本操作-2

5）取队头元素操作 GetHead(Q,&e)

功能：取队头元素，并用e返回。

6）入队操作 EnQueue(&Q, e)

功能：将元素e插入Q的队尾。

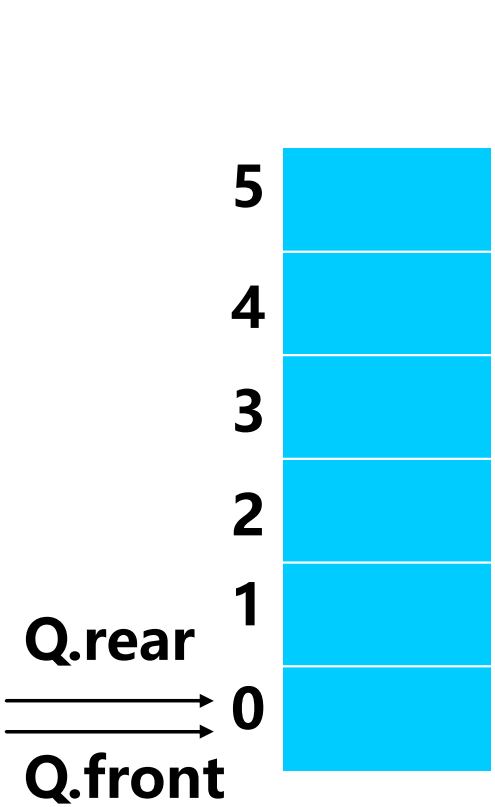
7）判空操作 QueueEmpty(Q)

功能：若队列Q为空，则返回True，否则返回False。

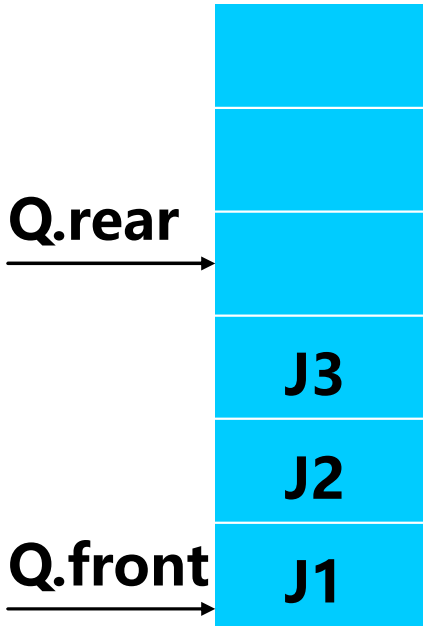
一、队列的顺序存贮结构

- #define MAXSIZE 100 //最大队列长度
  typedef struct
- { ElemType * base; //初始化时分配存储空间的基址
  int front; //队头指针，指向队头元素
  int rear; //队尾指针，指向队尾元素的下一个位置
- }SqQueue;

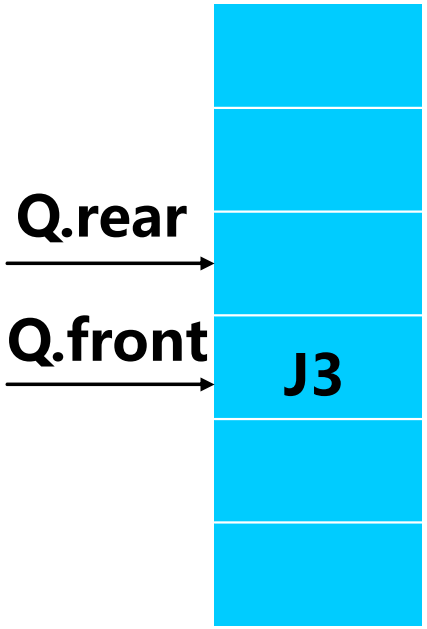
队列的入队出队



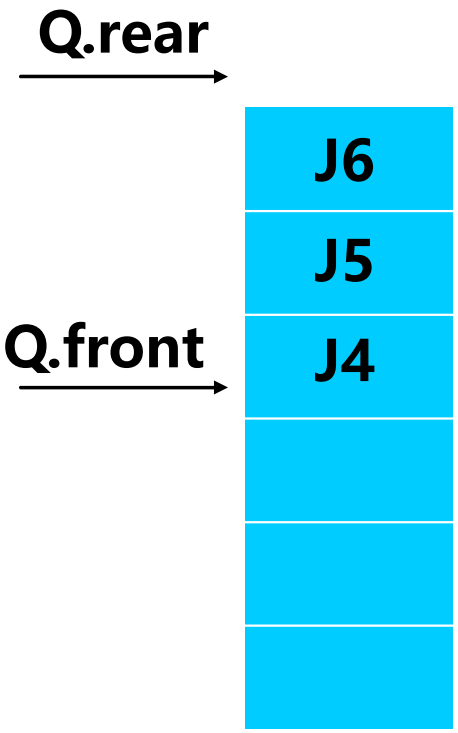
(a)空队列



(b)J1,J2和J3相继入队列

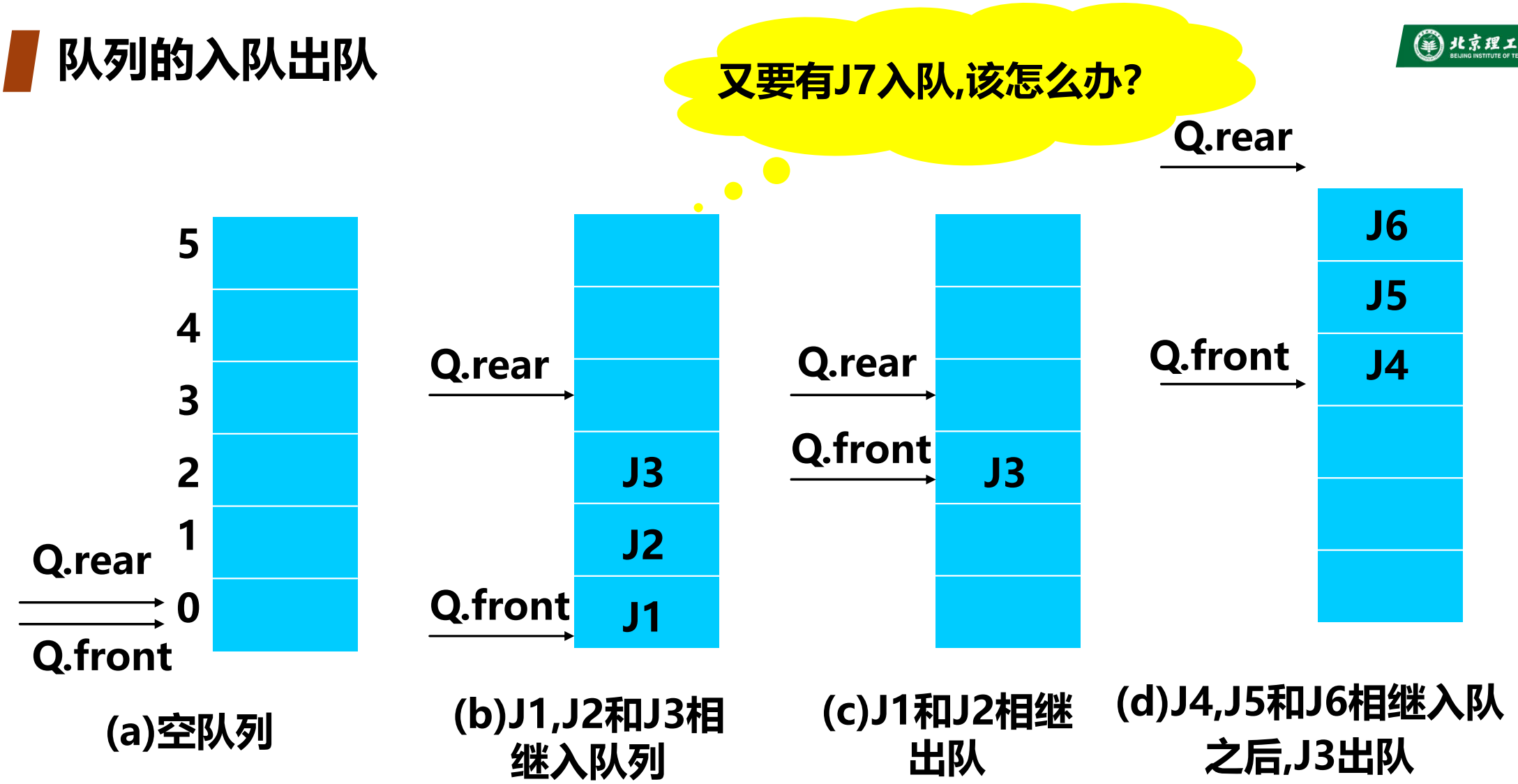


(c)J1和J2相继出队



(d)J4,J5和J6相继入队之后,J3出队

队列的入队出队


又要有J7入队,该怎么办?
5
4
3
2
1
0
Q.rear
Q.front
J3
J2
J1
J6
J5
J4
(a)空队列
(b)J1,J2和J3相继入队列
(c)J1和J2相继出队
(d)J4,J5和J6相继入队之后,J3出队

队列的设计

存在问题

设数组大小为M，则：

当front=0，rear= M 时，再入队发生溢出——真溢出

当front≠0，rear= M 时，再入队发生溢出——假溢出

解决方案

A、队首固定：每次出队后将剩余元素向下移动——浪费时间

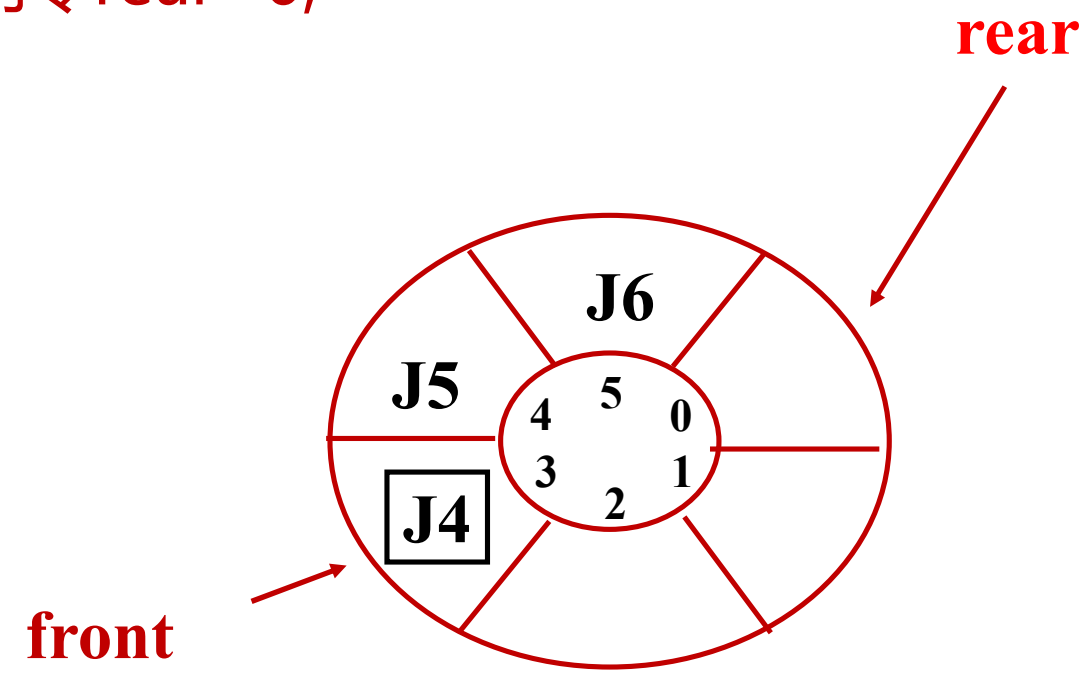
B、循环队列：

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

循环队列

循环队列

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



循环队列的队空、队满判定

实现：利用“模”运算

入队：Q.base[rear]=e;

rear=(rear+1)%M;

出队：e=Q.base[front];

front=(front+1)%M;

队满、队空判定条件?

队空：front==rear

队满：front==rear

解决方案：

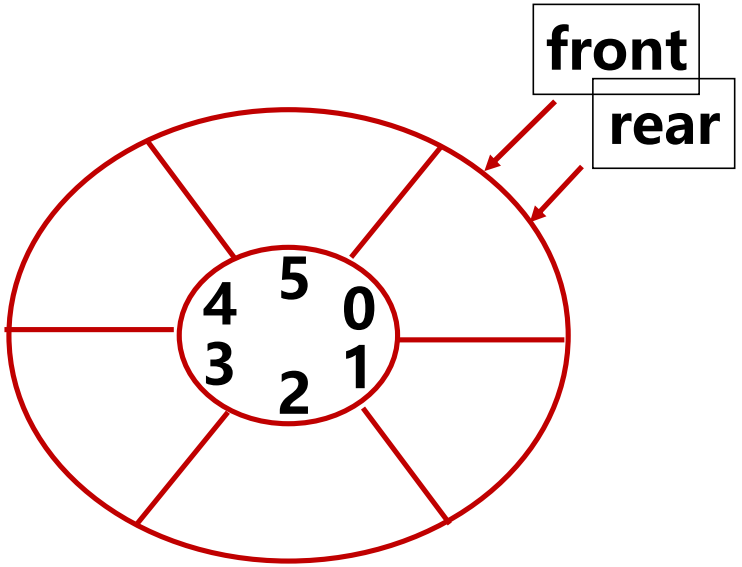
1.另外设一个标志以区别队空、队满

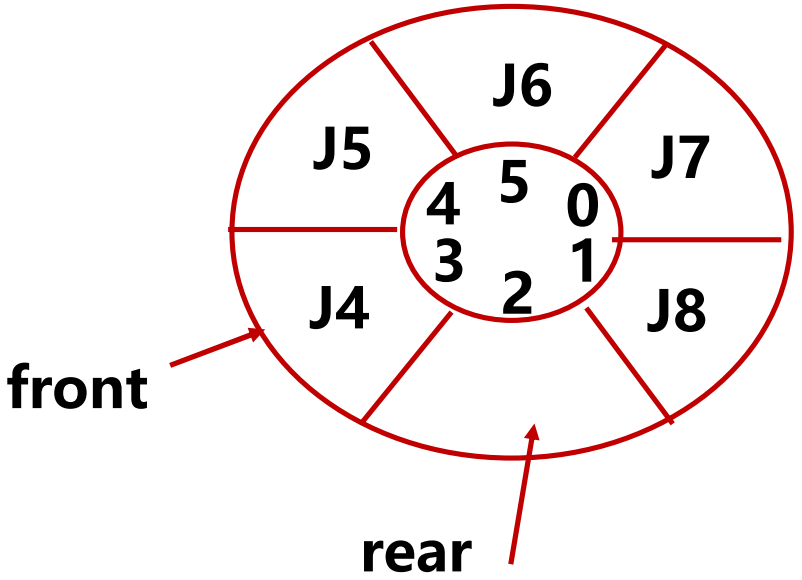
2.?

队满

少用一个元素空间：

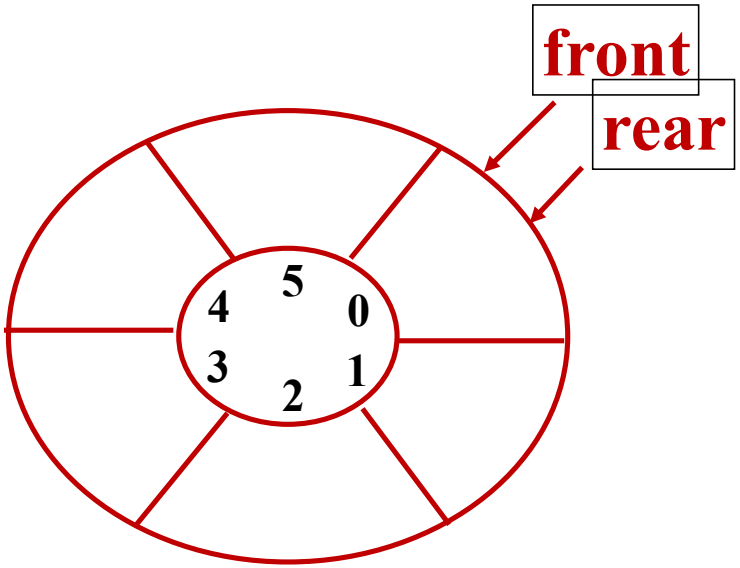
队空：front==rear
队满：(rear+1)%M==front





队列的基本操作

- 1）初始化操作 InitQueue_Sq（SqQueue &Q）
- 参数：Q是存放队列的结构变量
- 功能：建一个空队列Q
- 算法：

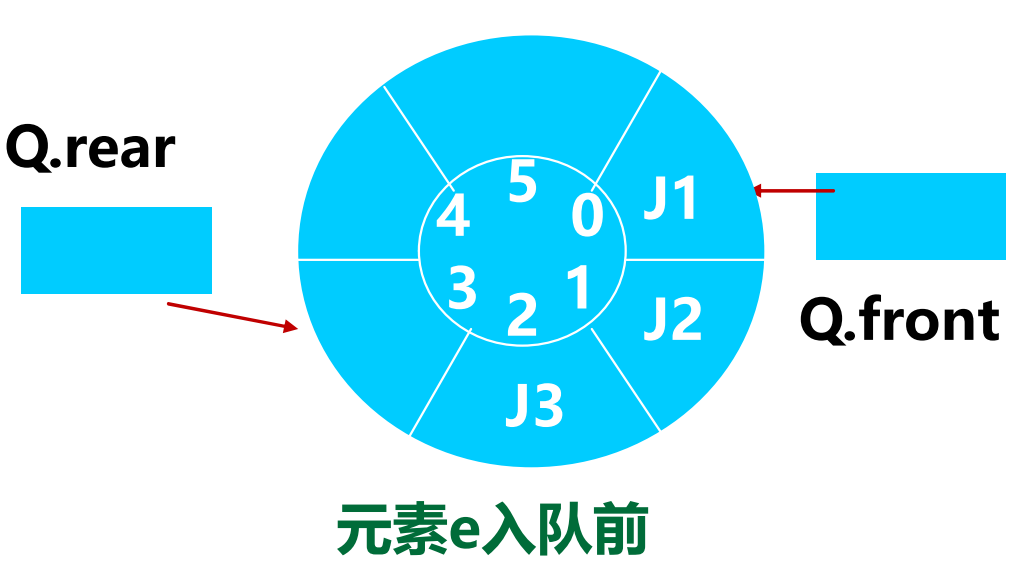


初始化操作 InitQueue_Sq

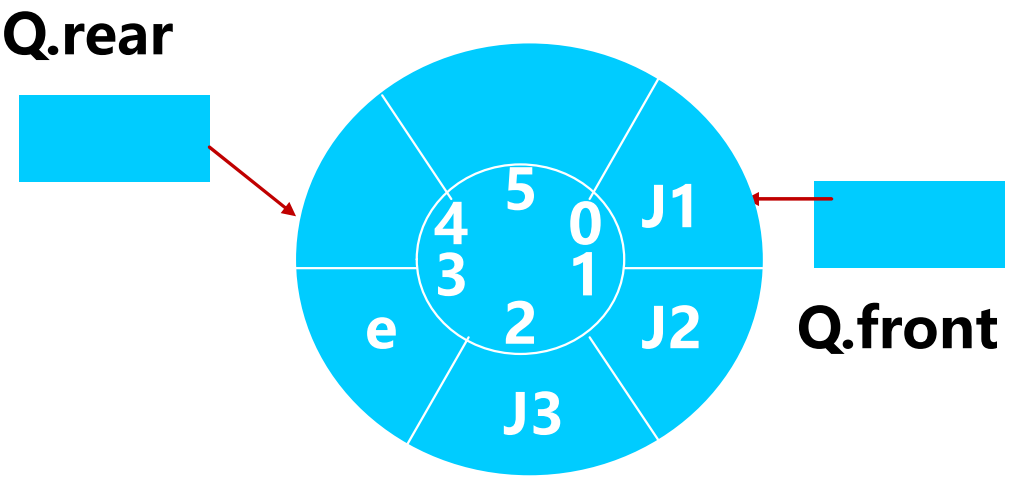
```
Status InitQueue_Sq ( SqQueue &Q )
{   //构造一个空队列Q
    Q.base = ( ElemType * ) malloc (MAXSIZE * sizeof
(ElemType));
    if ( !Q.base )     exit (OVERFLOW);      //存储分配失败
    Q.front = Q.rear = 0;
    Return OK;
}  //InitQueue_Sq
```

入队操作 EnQueue_Sq (SqQueue &Q, ElemType e)

- **功能：将元素 e 插入队尾**



元素e入队前



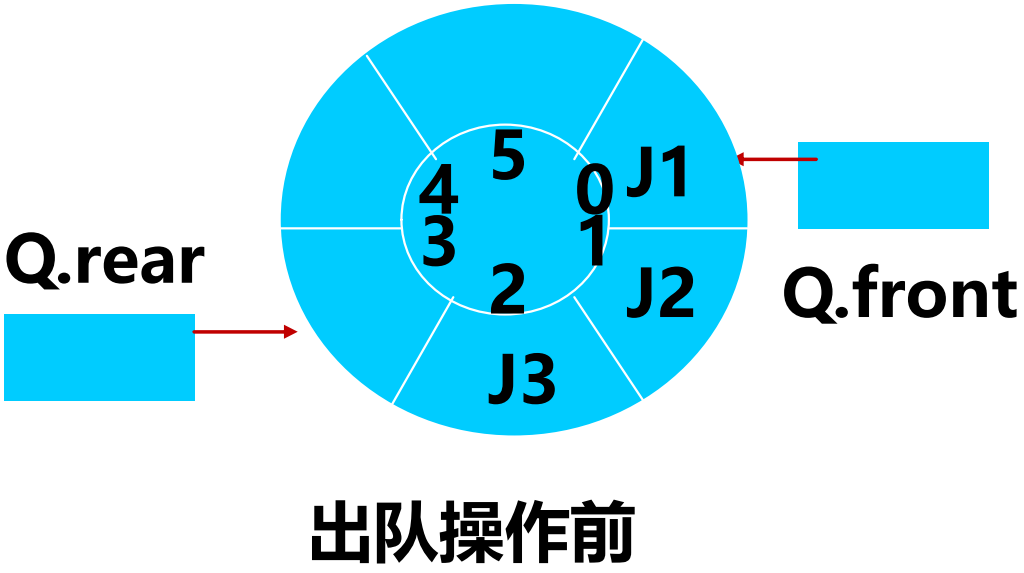
元素e入队后

队列的进队操作

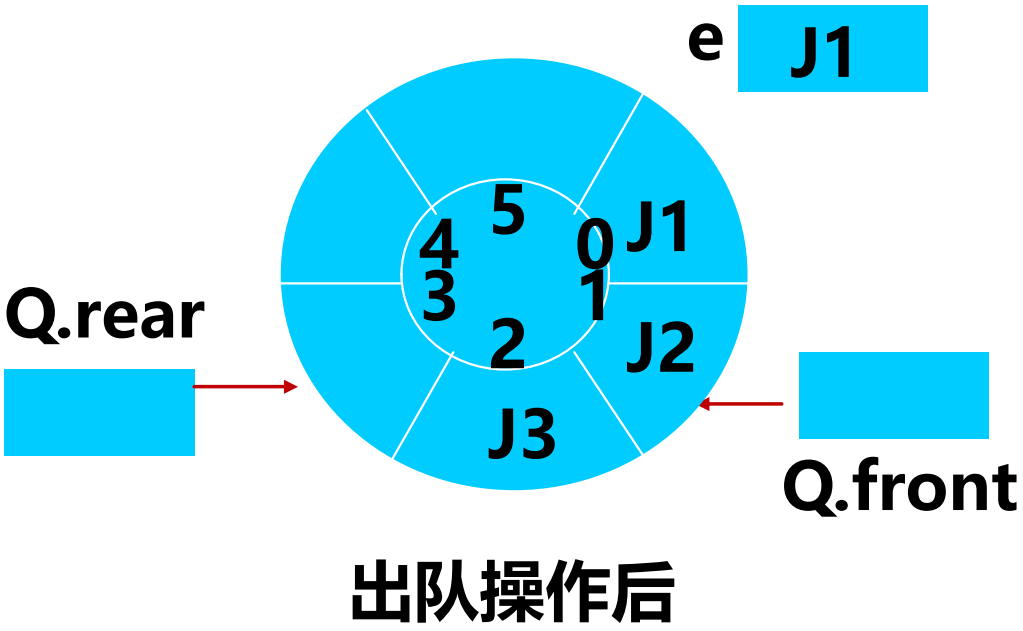
```
Status EnQueue_Sq ( SqQueue  &Q, ElemType  e )
{     //将元素e插入队尾
      if ( (Q.rear+1)%MAXSIZE= =Q.front )
            return ERROR ;                    //队满
      Q.base[Q.rear] = e ;                    //将元素e插入队尾
      Q.rear = (Q.rear+1)%MAXSIZE;            //修改队尾指针
      return OK;
} //EnQueue_Sq
```

出队操作 DeQueue_Sq (SqQueue &Q, QElemType &e)

功能：删除队头元素，用e返回其值



出队操作前

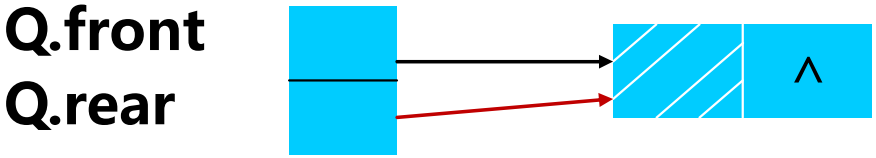


出队操作后

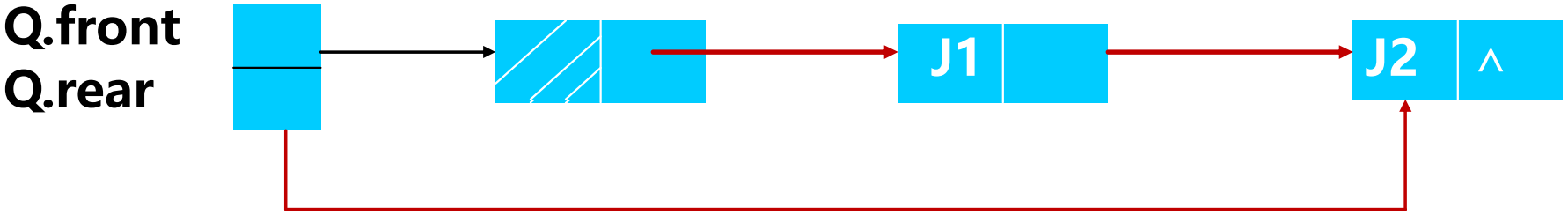
出队操作算法：

```
Status DeQueue_Sq ( SqQueue  &Q, ElemType &e )
{   //删除队头元素，用e返回其值，并返回OK；否则返回ERROR
    if ( (Q.rear= =Q.front )
          return ERROR ;                    //队空
    e = Q.base[Q.front] ;                    // 取队头元素 e
    Q.front = (Q.front+1)%MAXSIZE;  //修改队头指针
    return OK;
} //EnQueue_Sq
```

链队列



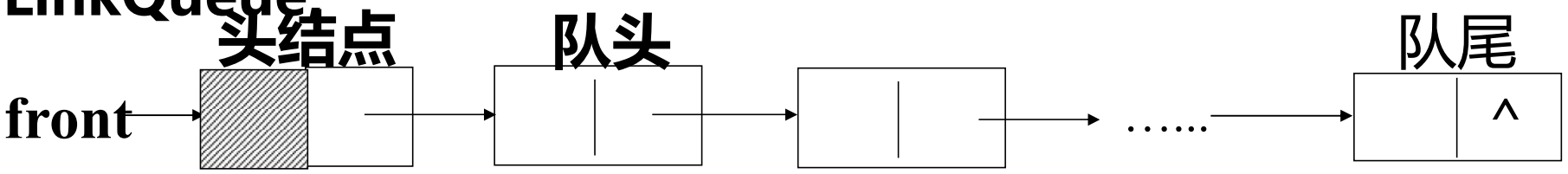
空链队列



链队列图示

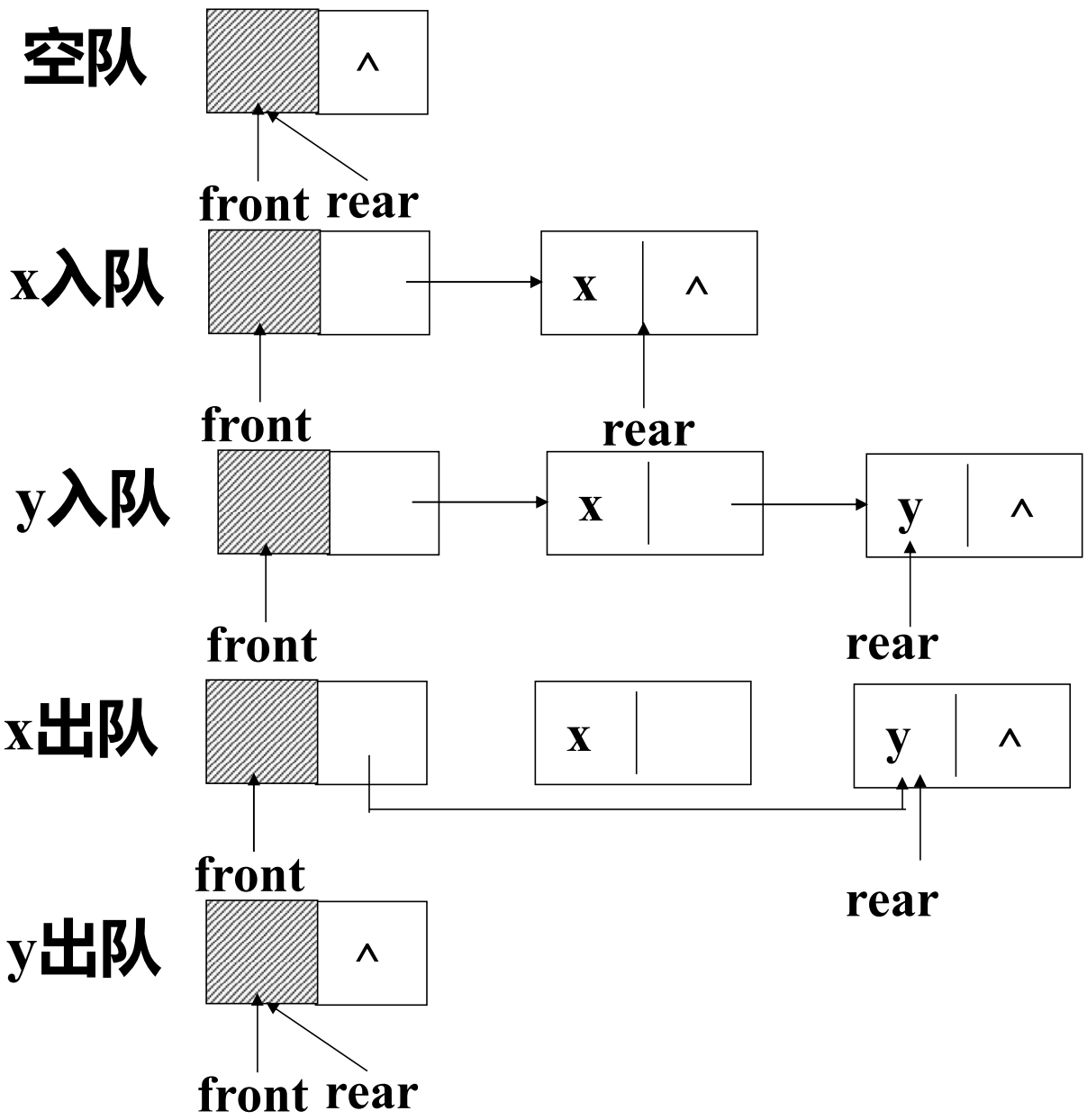
二、链队列的类型定义

```
typedef struct QNode    //链队列结点的类型定义
{  ElemType      data;
   struct QNode  * next;
}QNode, * QueuePtr;
typedef struct  // 链队列的表头结点的的类型定义
{  QueuePtr front;  //队头指针，指向链表的头结点
   QueuePtr rear;   //队尾指针，指向队尾结点 }
LinkQueue;
```



队列的操作

- 判断队空的条件：
- front==rear



三、队列的应用

- 1）解决计算机主机与外设不匹配的问题；
  2）解决由于多用户引起的资源竞争问题；

在操作系统课程中讲到

- 3）离散事件的模拟——模拟实际应用中的各种排队现象；
- 4）用于处理程序中具有先进先出特征的过程。

队列的应用

问题：编程判断一个字符序列是否是回文（回文是指一个字符序列以中间字符为基准两边字符完全相同）。

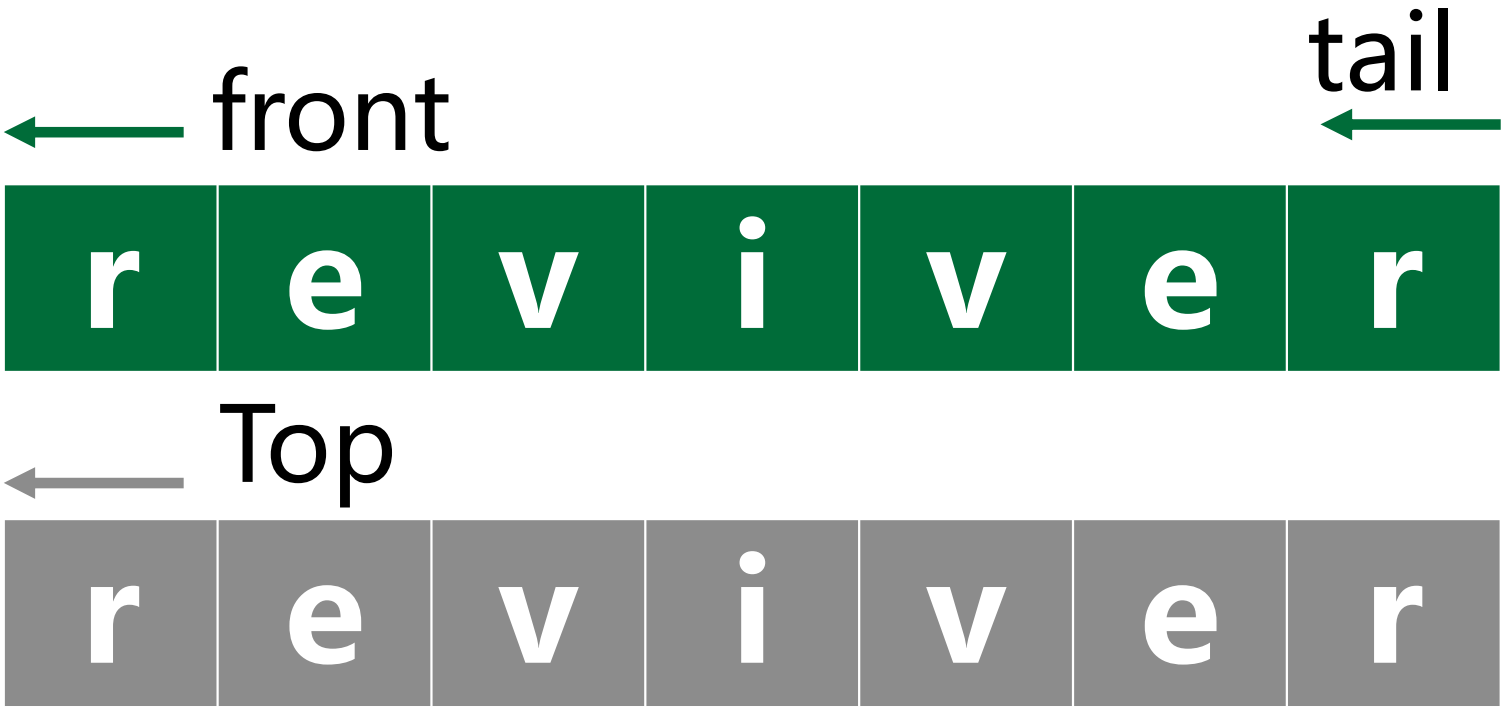
回文的算法设计

- 算法设计

  程序从键盘接受一个字符序列存入字符串str中，字符串长度≤80，输入字符序列以回车符为结束标记，字符串str中不包括回车换行符。

  将字符串中字符逐个分别存入队列和栈，然后逐个出队和退栈，比较出队的元素和退栈的元素是否相等，若全部相等则该字符序列是回文，否则就不是回文。

算法设计



队列的应用

提出问题

某运动会设立 N 个比赛项目，每个运动员可以参加1～3个项目。试问如何安排比赛日程既可以使同一运动员参加的项目不安排在同一单位时间进行，又可使总的竞赛日程最短。

运动会算法排期

问题抽象

若将此问题抽象成数学模型，则归属于“划分子集”问题。

N 个比赛项目构成一个大小为 n 的集合，有同一运动员参加的项目则抽象为“冲突”关系。

运动会排期算法设计

例如：某运动会设有 9 个项目：

A = { 0，1，2，3，4，5，6，7，8 },

七名运动员报名参加的项目分别为：

（1,4,8）、（1,7）、（8,3）、（1,0,5）、（3,4）、（5,6,2）、（6,4）

运动会排期算法设计

它们之间的冲突关系为：

R ={(1,4),(4,8),(1,8),
(1,7),
(8,3),
(1,0),(0,5),(1,5),
(3,4),
(5,6), (5,2),(6,2),
(6,4) }

构造 9×9 的“冲突”矩阵clash。

运动会排期算法设计

- 数据表示

R =

{(1,4),
(4,8),(1,8),
(1,7),(8,3),
(1,0),(0,5),
(1,5),(3,4),
(5,6),(5,2),
(6,2),(6,4)}构造对称矩阵

	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	1	0	0	0
1	1	0	0	0	1	1	0	1	1
2	0	0	0	0	0	1	1	0	0
3	0	0	0	0	1	0	0	0	1
4	0	1	0	1	0	0	1	0	1
5	1	1	1	0	0	0	1	0	0
6	0	0	1	0	1	1	0	0	0
7	0	1	0	0	0	0	0	0	0
8	0	1	0	1	1	0	0	0	0

运动会排期算法设计

问题分析

“划分子集”问题即为：

将 集合A 划分成 k 个互不相交的子集：

$$A1，A2，...，Ak（k\leq n），$$

使同一子集中的元素均无冲突关系，并要求划分的子集数目尽可能地少。对前述例子而言，问题即为：同一子集中的项目为可以同时进行的项目，显然希望运动会的日程尽可能短。

运动会排期算法设计

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

算法设计

利用队列先进先出的特点，将待划分的集合A中的所有元素放入一个队列中，然后依次取出元素放入一个待分配的组中，若当前元素与改组中已经入选的元素无冲突，则出栈，如果产生冲突则继续放在队列的尾部；当遍历考察一轮队列中的所有元素后，产生一个无冲突的子集，如此循环直到所有元素都被分配完成时结束分配。

运动会排期算法设计

为了减少重复察看 R 数组的时间，可另设一个数组clash[n]，记录和当前已入组元素发生冲突的元素的信息。

每次新开辟一组时，令 clash 数组各分量的值均为“0”，当序号为“i”的元素入组时，将和该元素发生冲突的信息记入clash数组。

	0	1	2	3	4	5	6	7	8
0	0	2	0	0	1	2	1	0	1
1	1	0	0	0	1	1	0	1	1
2	0	0	0	0	0	1	1	0	0
3	0	0	0	0	1	0	0	0	1
4	0	1	0	1	0	0	1	0	1
5	1	1	1	0	0	0	1	0	0
6	0	0	1	0	1	1	0	0	0
7	0	1	0	0	0	0	0	0	0
8	0	1	0	1	1	0	0	0	0

012345678

(0237)

14568

(16)

458

(45)

8

运动会排期算法设计

```
pre (前一个出队列的元素序号) = n;  组号 = 0;
全体元素入队列;
while ( 队列不空 )
{   队头元素 i 出队列;
    if ( i < pre )     // 开辟新的组
     {   组号++;  clash 数组初始化;
     }
    if ( i 能入组 )
    {   i 入组，记下序号为 i 的元素所属组号;
        修改 clash 数组;
    }
    else   i 重新入队列;
    pre = i;
}
```

结果：
(0,2,3,7)，
(1,6)，
(4,5)，
(8)

运动会排期算法设计

考虑一下?

在循环队列中为何不建议使用动态扩展数组?

队列的其他应用场景

日常生活中的排队

搭乘公共汽车；

顾客到商店购买物品；

病员到医院看病；

旅客到售票处购买车票；

学生去食堂就餐

“无形”排队：多个顾客打电话叫出租车，如果出租汽车站无足够车辆、则部分顾客只得在各自的要车处等待，他们分散在不同地方，却形成了一个无形队列在等待派车

队列的其他应用场景

物体队列：

通讯卫星与地面若干待传递的信息；
生产线上的原料、半成品等待加工；
因故障停止运转的机器等待工人修理；
码头的船只等待装卸货物；
要降落的飞机因跑道不空而在空中盘旋等等

队列的其他应用场景

排队论(Queuing Theory)，又称随机服务系统理论(Random Service System Theory),是一门研究拥挤现象(排队、等待)的科学。

具体地说，它是在研究各种排队系统概率规律性的基础上，解决相应排队系统的最优设计和最优控制问题。例如：

1. 停车场：如何设计停车场的车位和停车规则，提高停车场的使用效率。
2. 信号交叉口：根据不同的车流量，设计区域内所有信号灯的变化模式。减少由于设计不当造成车流堆积现象。
3. 公共交通系统运营优化：减少乘客等待时间，并尽量降低运营成本。

队列的其他应用场景

排队系统的最优设计和最优控制问题，例如：

银行现有8个窗口，没有顾客排队的现象。

在现有的顾客量和服务方式的情况下，能否减少窗口数？

4个行不行？

排队论的三要素

输入过程

排队规则

服务机构

队列的其他应用场景

1、输入过程

- 输入过程就是指要求接受服务的顾客是按照一种什么样的规律到达排队系统的一个过程，有时候，也把它叫做顾客流。如果要想完整的刻画一个输入过程，一般可以从 3 个方面来进行描述：

  A.顾客总体数（顾客源）

  B.顾客的到达方式：单个到达，还是成批到达

  C.顾客流的概率分布

队列的其他应用场景

1、输入过程

由于顾客到达的间隔时间和服 务时间不可能是负值，因此，它的分布是非负随机变量的分布。

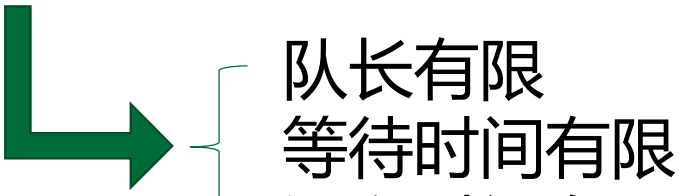
最常用的分布有泊松分布、确定型分布，指数分布和爱尔朗分布。

队列的其他应用场景

- 2.排队规则

主要分为三类：

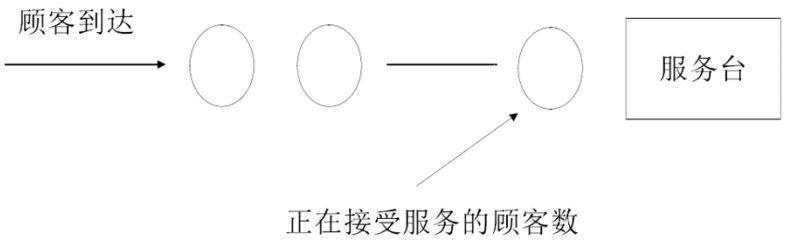
- A.损失制
- B.等待制
- C.混合制



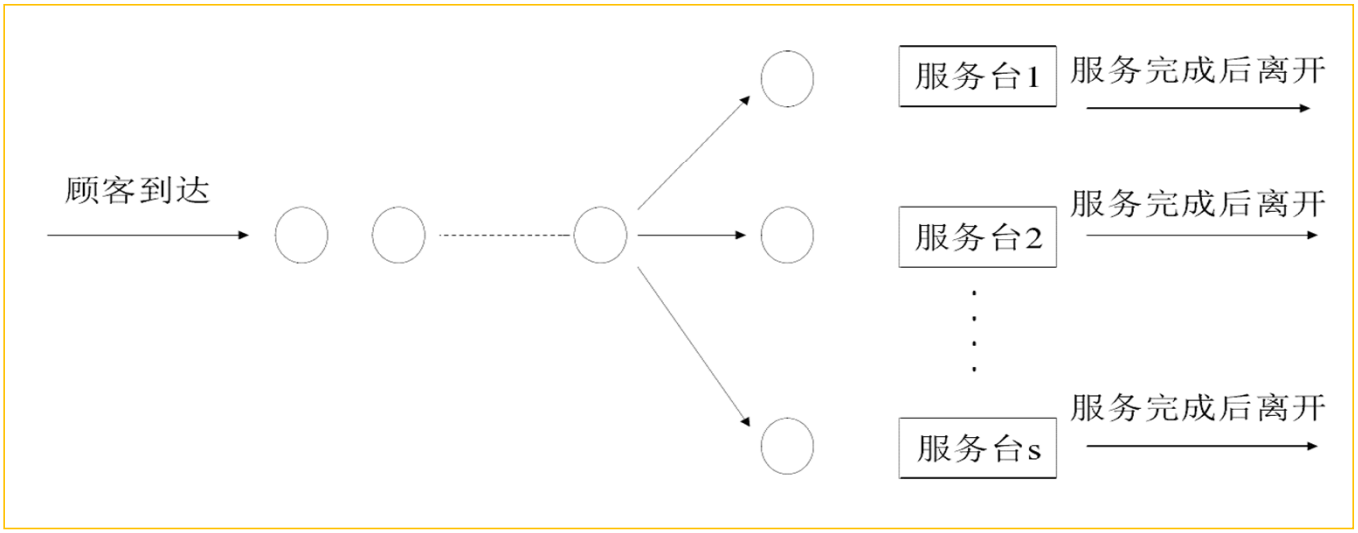
队长有限
等待时间有限
逗留时间有限

先到先服务
后到先服务
随机服务
优先权服务
循环排队
多个服务机构

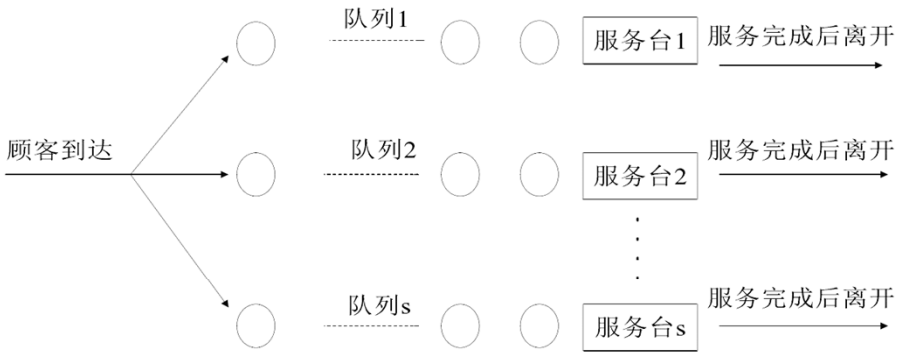
- 3.服务机构



单个服务机构

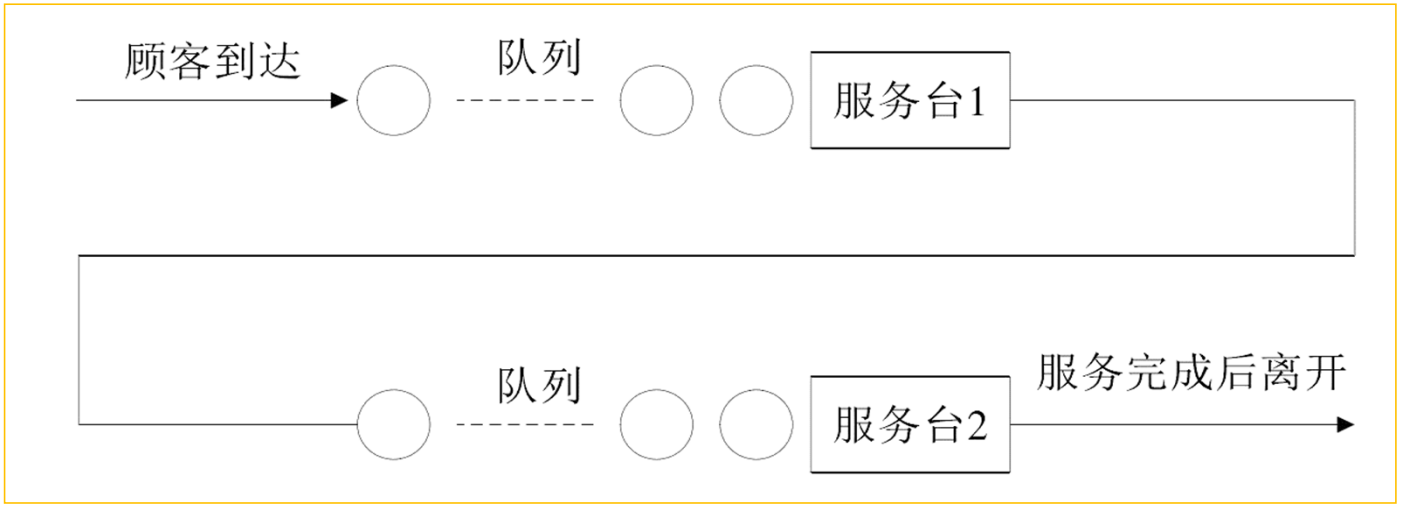


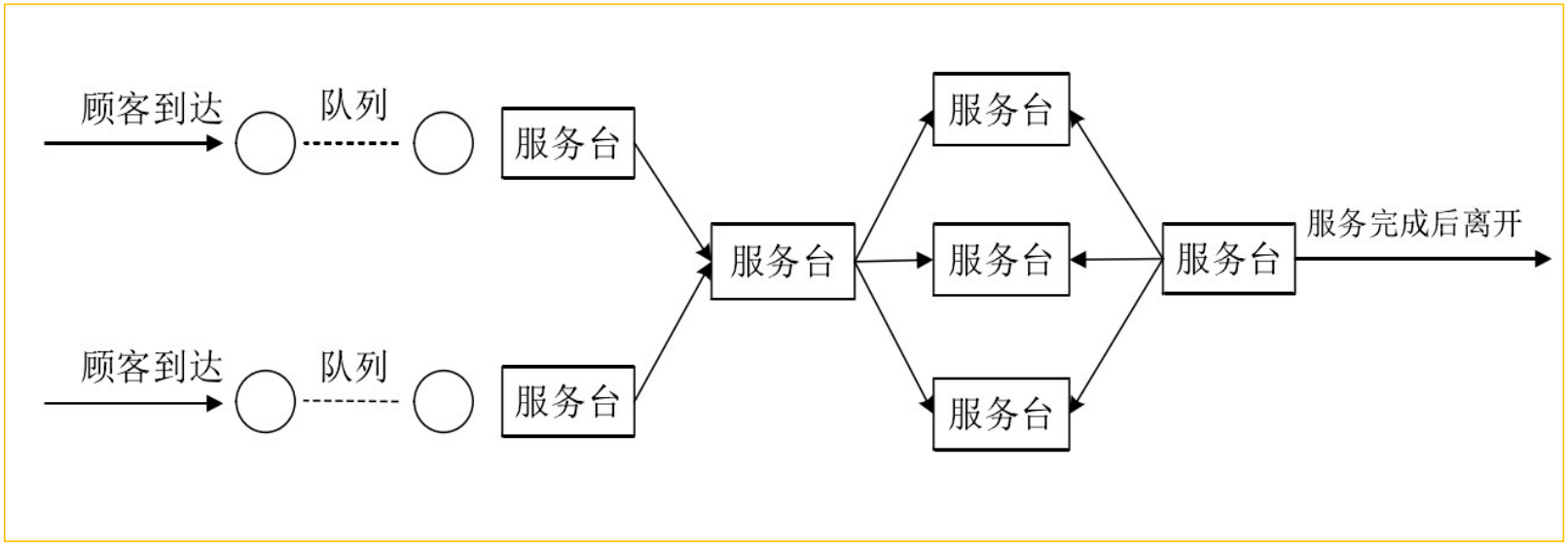
多个服务机构



多队列多服务机构

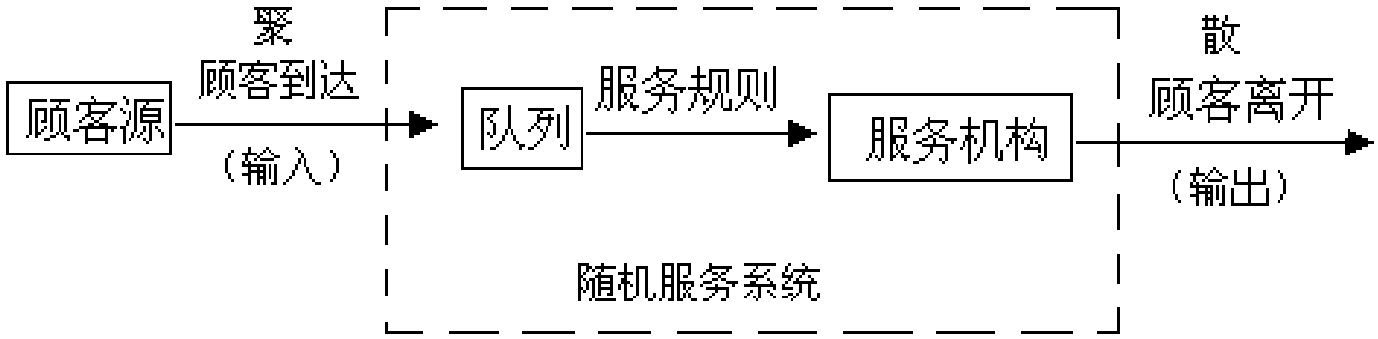
单队列多服务机构





多队列多服务机构混合网络结构

各种排队系统的统一描述



评价排队系统的优劣

1、队长与排队长

(1)队长(L)：系统中的顾客数（n）；

(2)排队长(Lq)：系统中排队等待服务的顾客数;

2、逗留时间与等待时间

(1)逗留时间:W

——指一个顾客在系统中的全部停留时间;

(2)等待时间: Wq

——指一个顾客在系统中的排队等待时间;

忙期和闲期

(1)忙期:

是指从顾客到达空闲着的服务机构起，到服务机构再次成为空闲止的这段时间，即服务机构连续忙的时间。

(2)闲期:

与忙期相对的是闲期，即服务机构连续保持空闲的时间。

3、其他相关指标

(1)忙期服务量：指一个忙期内系统平均完成服务的顾客数;

(2)损失率: 指顾客到达排队系统，未接受服务而离去的概率;

（对损失制或系统容量有限而言）

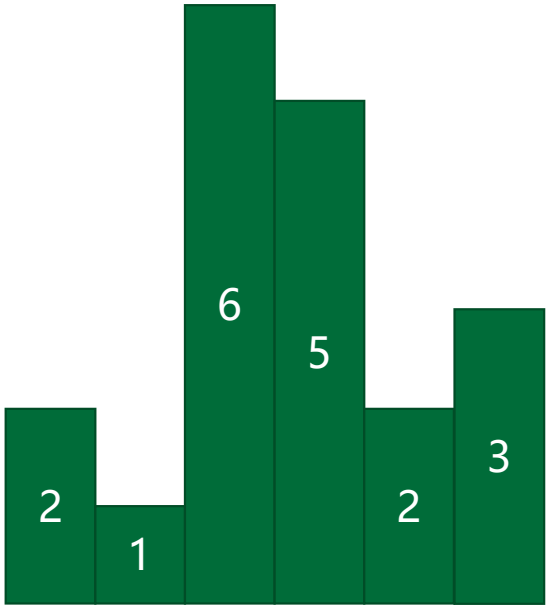
(3) 服务强度：　　=　　/s　；

综合练习

柱状图的最大矩形（Leetcode84）

给定n个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为1。求在该柱状图中，能够勾勒出来的矩形的最大面积。

测例：6，7，5，2，4，9，5，3



综合练习

滑动窗口最大值（Leetcode239）：

给定一个数组nums,有一个大小为k的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的k个数字。滑动窗口每次只向右移动1位。要求：返回滑动窗口的最大值。

输入：nums = [1, 3, -1, -3, 5, 3, 6, 7], k=3

输出: [3, 3, 5, 5, 6, 7]

队列的应用例题

某写字楼共有22层有6部电梯，每部电梯可乘用10人，每天有四个时段出入比较繁忙，这段时间的人群按常数分布进出：

早晨：08:00-09:00（上班时间），入:20人/分钟，出:2人/分钟
上午：11:00-12:00（下班时间），入:2人/分钟，出:20人/分钟
下午：13:00-14:00（上班时间），入:20人/分钟，出:2人/分钟
晚上：18:00-19:00（下班时间），入:2人/分钟，出:20人/分钟

其他时间采用泊松分布出入：

队列的应用例题

每个人到达1-22层的概率是相等的；
电梯运行每一层：停留时间2分钟，不停留：0.5分钟
6部电梯按最先抵达优先响应；
要求：
记录每个用户（早8:00-晚22:00期间）等待电梯的平均时间
低于3分钟，满意，高于6分钟，不满意
统计：
1、每一天的顾客满意率；
2、增加/减少几部电梯，能够提高/保持平均满意率大于70%

离散事件模拟

银行现有8个窗口，没有顾客排队的现象。

在现有的顾客量和服务方式的情况下，能否减少窗口数?

4个行不行?

Number of customers per minute	Percentage of one-minute intervals	Range
0	15	1–15
1	20	16–35
2	25	36–60
3	10	61 70
4	30	71–100

(a)

The amount of of time needed for service in seconds	Percentage of customers	Range
0	0	—
10	0	—
20	0	—
30	10	1 10
40	5	11–15
50	10	16–25
60	10	26–35
70	0	—
80	15	36–50
90	25	51–75
100	10	76–85
110	15	86–100

(b)

离散事件模拟

假设某银行有4个窗口对外接待客户，从早晨银行开门起不断有客户进入银行。

每个窗口一次只能接待一个客户，人数众多时需要在每个窗口前排队。

刚进入银行的客户，如果有窗口空闲则可上前办理。

若窗口均有客户，他便排在人数最少的队伍后面。

要求：编制一个程序模拟银行的这种业务活动，并计算一天中客户在银行逗留的平均时间

离散事件模拟

要求：编制一个程序模拟银行的这种业务活动，并计算一天中客户在银行逗留的平均时间

需要记录每个客户的逗留时间和客户总数

需要模拟客户到达、排队、办事和离开的过程

客户逗留时间：客户离开时间 - 客户达到时间

需要模拟客户哪些信息呢?

客户到达时间

客户办理业务的时间

怎么模拟客户到达呢?

假设第一个客户到达时间为0时刻

之后每一个客户到达的时刻在前一个客户到达的时候设定，即产生一个随机数:

下一客户达到的时间间隔（intertime）

客户到达后做什么?

- 产生一个随机数，表示业务办理时间（durtime）
- 先选择最短队排队
- 如果某个队列为空，则去办理业务，计算该客户的离开时间

怎么模拟客户办理业务并离开呢?

- 队列队首的客户办理业务，具体办理过程忽略
- 计算队首客户的逗留时间=离开时间 - 达到时间
- 计算本队列中下一个客户的离开时间

产生客户到达事件的方式：

假设第一个客户到达时间为0时刻

之后每一个客户到达的时刻在前一个客户到达的时候设定，即产生一个随机数：

下一客户达到的时间间隔（intertime）

产生客户离开事件的方式：

情况1：在队列中第一个客户到达的时候设定

情况2：当处理一个客户的离开事件时，设置该客户所在队列下一个客户（队首元素）的离开时间。

数据结构设计

事件(event)表：客户到达银行和离开银行两个时间发生的事情为事件。

- 事件表：按照事件发生时间排序的有序表
- 基本信息：事件发生时间、事件类型
- 基本操作：插入、删除

客户排队的队列

- 4个队列
- 基本信息：客户到达时间、办理时间
- 基本操作：插入、删除

```
typedef struct{    //事件结构
        int OccurTime; //事件发生时间
        int  NType; //事件类型。0：到达事件；
                          // 1234：四个窗口的离开事件
} Event, Elemtype;
Typedef  LinkList  EventList; //事件链表定义为按照事件发生时间排序的有序链表
```

```
typedef struct{// 排队队列元素
        int ArrivalTime; //到达时刻
        int Duration; //办理事务所需时间
} QElemtype;//队列元素
```

- 事件驱动模式(event-driven)：按照时间发生的先后顺序进行模拟
  - 若用户到达，则处理到达事件CustomerArrived()
  - 若用户离开，则处理离开事件CustomerDepature()

```
//变量定义
Event List    ev;//事件表
Event         en;//当前事件
LinkQueue     q[5];//4个客户队列
QElemType     customer;//客户记录
int           TotalTime;//累计逗留时间
int           CustomerNum;//客户总人数
```

```
void Bank_Simulation( int ClostTime){
    OpenForDay();//初始化
    while(!ListEmpty(ev)){//依次从事件表中取出事件
        DelFirst(GetHead(ev),p);
        en = GetCurElem(p);// 检查事件类型
        if (en.Ntype == 0)
            CustomerArrived();//处理达到事件
        else  CustomerDeparture(); //处理离开事件
    }//while
    printf( "the average time is %f\n",
                (float)TotleTime/CustomerNum);
}// Bank_Simulation
```

1. 客户数目加1。
2. 生成下一个客户到达事件，随机产生durtime、 intertime。
3. 将下一个客户的到达事件插入事件表ev，若在下班前到达。
4. 将当前客户插入最短的队列i中。
5. 若队列i中只包有当前客户，则将当前客户的离开事件插入到事件表。

```
1. ++CustomerNum;
2. Random(durtime, intertime);
3. t = en.OccurTime + intertime;
   if(t < CloseTime) OrderInsert(ev, (t, 0), cmp);
4. i = Minimum(q); EnQueue( q[i], (en.OccurTime, durtime));
5. if(QueueLength(q[i] ==1) OrderInsert(ev, (en.OccurTime +
   durtime, i), cmp);
```

1. 删除队列i的队首客户
2. 累计客户逗留时间：离开时间 - 客户达到时间
3. 设定队列i当前队首客户的离开事件，插入事件表

```
1.  i = en.NType; DeQueue(q[i], customer);
2.  TotleTime += en.OccurTime – customer.ArrivalTime;
3.  if(!QueueEmpty(q[i])){
        GetHead(q[i], customer);
        OrderInsert(ev, (en.OccurTime + customer.Duration, i),          cmp);
}
```

```
void OpenForDay(){
    TotalTime = 0; CustomerNum = 0;
    InitList( ev );//初始化事件表
    en. Occurtime = 0; en.Ntype = 0;
    OrderInsert( ev, en, cmp);//插入第一个到达的客户的到达事件
    for( i = 1; i <=4; i++) InitQueue( q[i]);//初始化4个排队队列
}// OpenForDay
```

- 剩余的问题：如何实现有序表的插入操作?

```
//普通队列的入队操作
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; //插在队尾
   return OK;
}
```

- 找到第一个大于等于e的结点，插在它之前

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
4. 理解递归算法执行过程中栈的状态变化过程