

第三章 栈和队列

1. 栈的定义和栈的存储结构
2. 栈的应用
3. 栈与递归
4. 队列的定义和链队列
5. 循环队列

3.1 栈的定义和存储结构

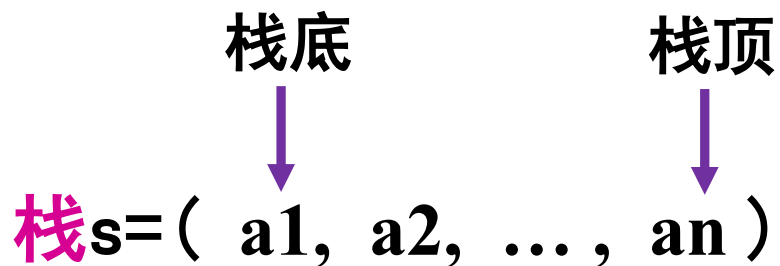
1 栈的定义

栈(Stack): 限定仅只能在表尾进行插入和删除的线性表。

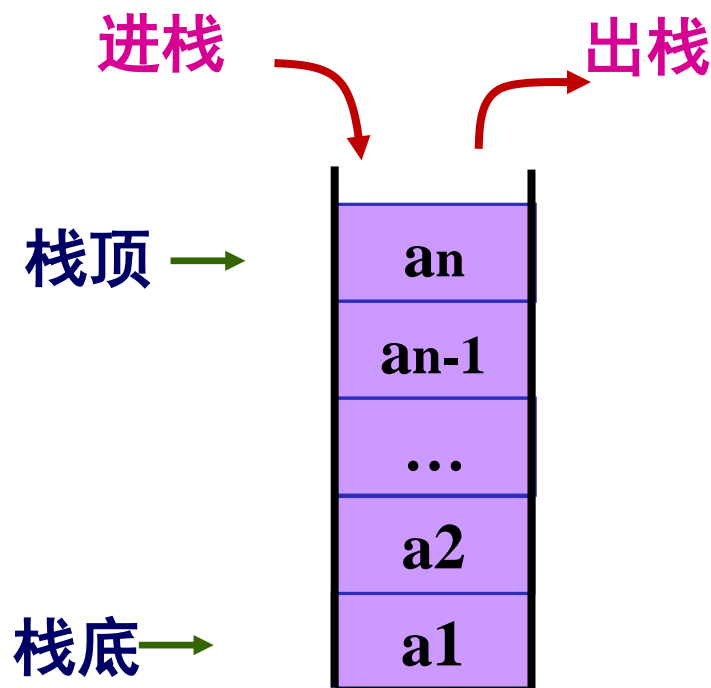
栈顶(Top): 表尾。

栈底(Bottom): 表头。

空栈: 不含元素的空表。



栈 $s = (a_1, a_2, \dots, a_n)$



栈中元素按 a_1, a_2, \dots, a_n 的次序进栈，退栈的第一个元素应为栈顶元素。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 (Last In First Out) 的线性表。

栈的抽象数据类型定义：

ADT Stack {

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

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

约定 a_n 端为栈顶， a_1 端为栈底。

基本操作：

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。

} ADT Stack

2 栈的存储结构

1. 顺序栈:

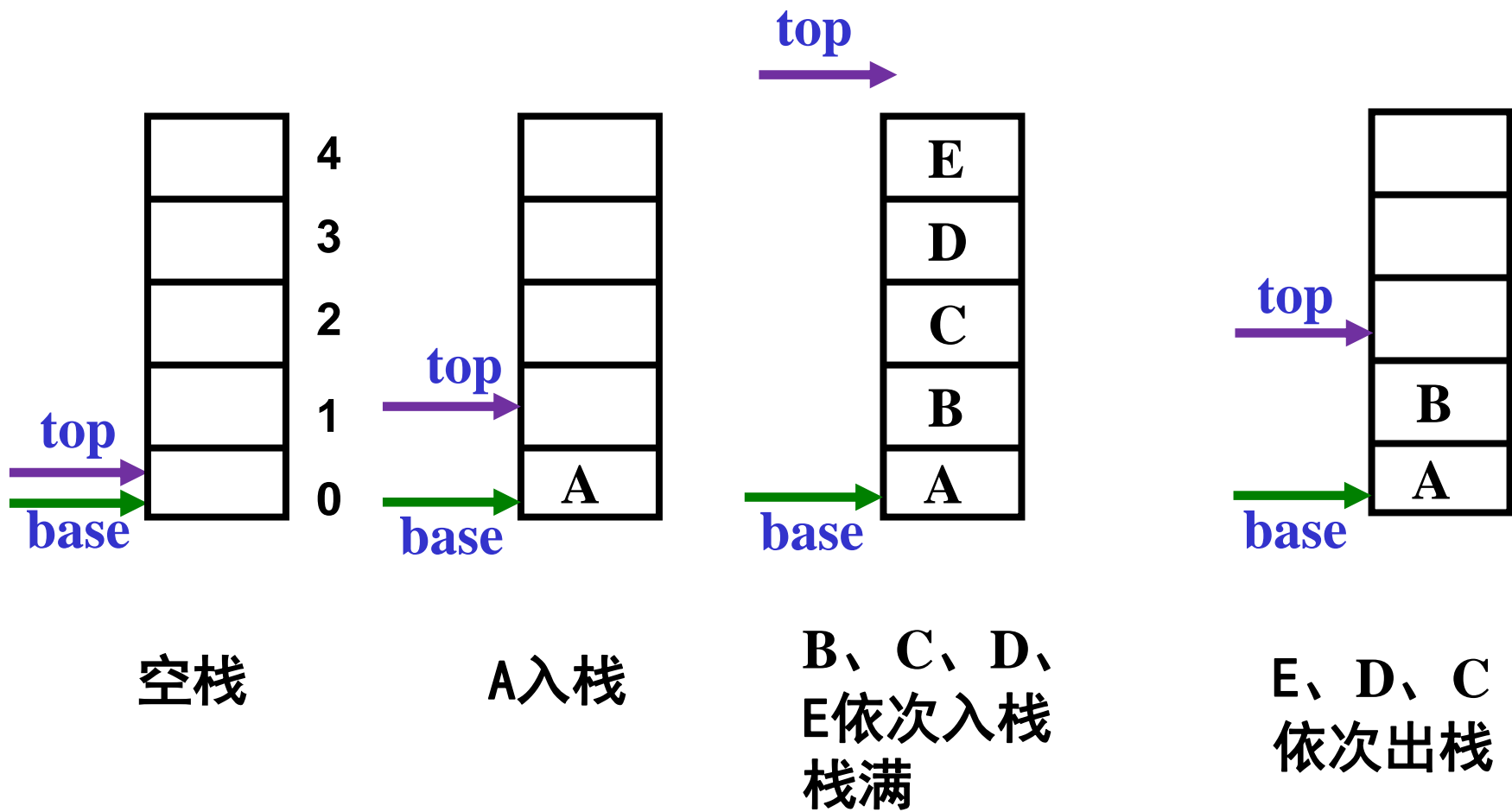
利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素, 同时附设指针top指示栈顶元素在顺序栈中的位置。

2. 链栈: 利用链表实现。

顺序栈的类型声明:

```
#define INIT_SIZE 100      // 存储空间初始分配量
#define INCREMENT 10      // 存储空间分配增量

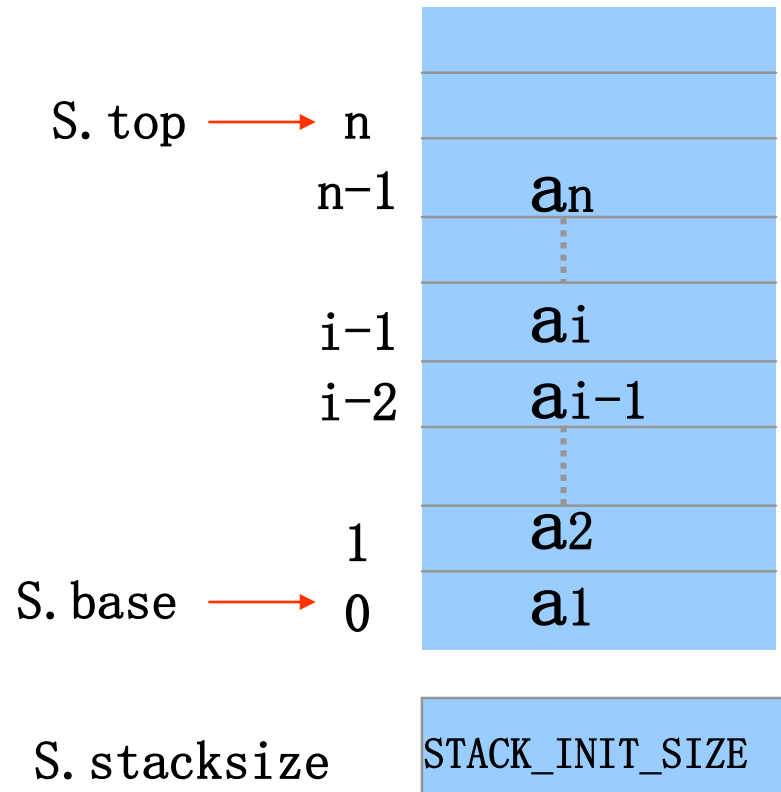
typedef struct{
    SElemType *base;      // 栈底指针
    SElemType *top;       // 栈顶指针
    int StackSize;        // 栈的当前已分配的存储空间
}SqStack;
```



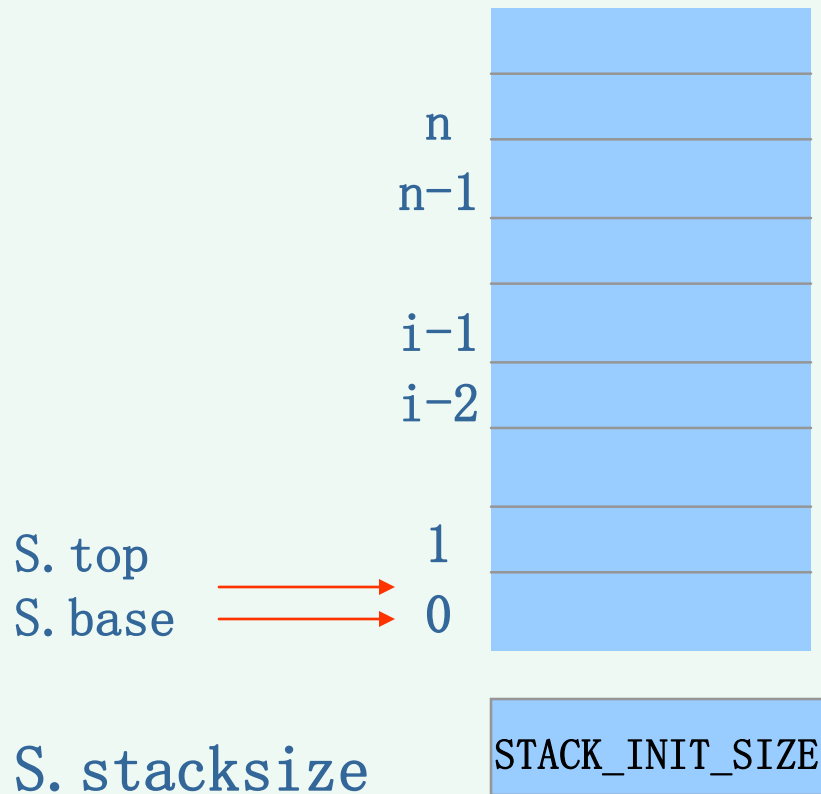
$\text{top} == \text{base}$ 是栈空标志
 $\text{stacksize} = 5$

插入新的栈顶元素时，指针 top 增1；删除栈顶元素时，指针 top 减1，因此，非空栈中的栈顶指针始终在栈顶元素的下一个位置上。

顺序栈的图示



初始化操作图示



顺序栈基本操作的实现

顺序栈的初始化 InitStack (&S)

```
Status InitStack (SqStack &S)
{ //构造一个空栈S

    S.base=(SElemType *)malloc(INIT_SIZE *sizeof(SElemType));

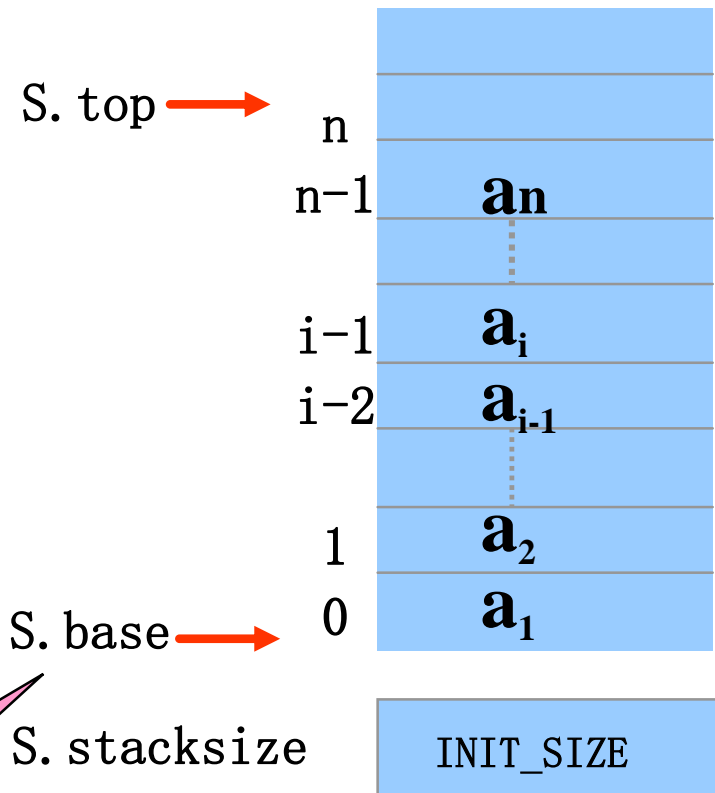
    if(!S.base) exit(OVERFLOW); //存储分配失败

    S.top=S.base;    //top初始为base

    S.stacksize=INI_SIZE;

    return OK;
}
```

顺序栈的置空 `ClearStack(&S)`



`S.base = S.top`
表明栈为空

置空栈操作图示

```
Status ClearStack (SqStack &S)
{  //将S清为空栈

    If (!S.base) return ERROR;

    S.top=S.base;
}
```

顺序栈的判空 **StackEmpty (S)**

```
Status StackEmpty (SqStack S)
{ //若栈S为空栈，则返回TRUE，否则FALSE。

    If (!S.base) return ERROR;

    If (S.top==S.base)
        return TRUE;
    else
        return FALSE;
}
```

顺序栈的销毁 **DetroyStack (&S)**

```
Status DetroyStack( SqStack &S) //销毁一个已存在的栈
{
    If (!S.base) return ERROR;
    free (S.base);                // 回收栈空间

    S.base = S.top = NULL;
    S.stacksize = 0;

    return OK;
}
```

入栈 `Push(&S, e)`

在栈顶插入一个新的元素。

```
Status Push(SqStack &S, SElemType e)
```

```
{ //插入元素e为新的栈顶元素
```

```
  if(S.top - S.base >= S.stacksize) { //栈满，追加存储空间
```

```
    newbase = (SElemType *) realloc(S.base,
```

```
        (S.stacksize + INCREMENT) * sizeof(SElemType));
```

```
    if(!newbase) exit(OVERFLOW); //存储分配失败
```

```
    S.base = newbase;
```

```
    S.top = S.base + S.stacksize;
```

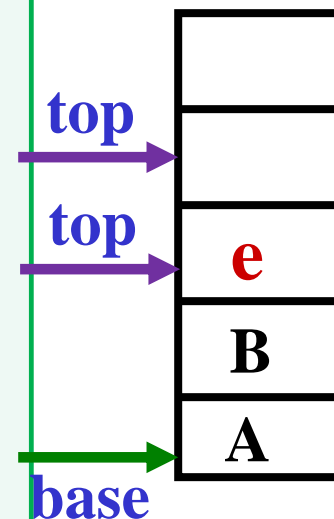
```
    S.stacksize += INCREMENT;
```

```
  }
```

```
  *S.top++ = e; //元素e压入栈顶，栈顶指针加1
```

```
  return OK;
```

```
}
```



出栈 Pop(&S, &e)

将栈顶元素删除。

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

```
{ //删除S的栈顶元素，用e返回其值
```

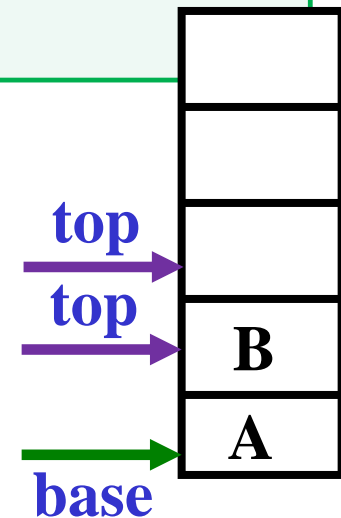
```
    if (S.top==S.base ) return ERROR;
```

```
    e=*--S.top; //栈顶指针减1，将栈顶元素赋值给e
```

```
    return OK;
```

```
}
```

e B



取栈顶元素 **GetTop(S, &e)**

用e返回S的栈顶元素，栈顶指针保持不变。

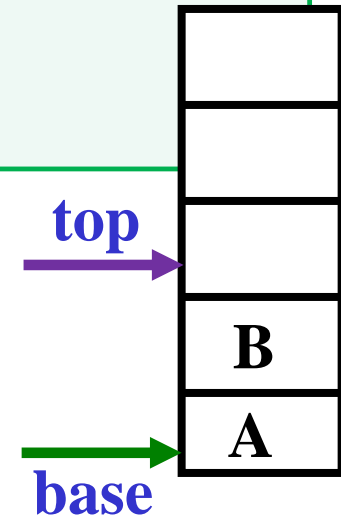
```
Status GetTop(SqStack S, SElemType &e)
{ //用e返回S的栈顶元素

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

    e=*(S.top-1);

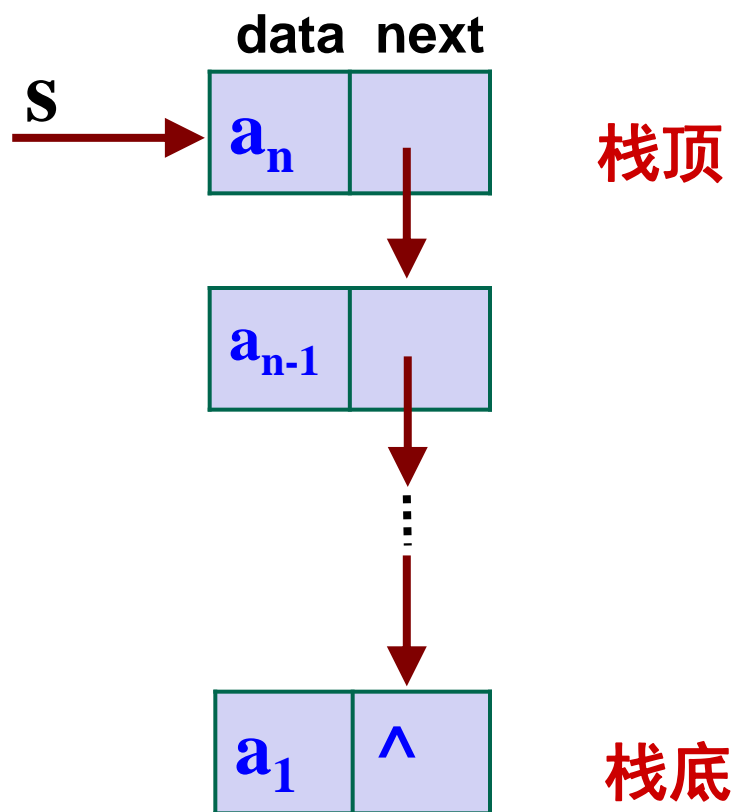
    return OK;
}
```

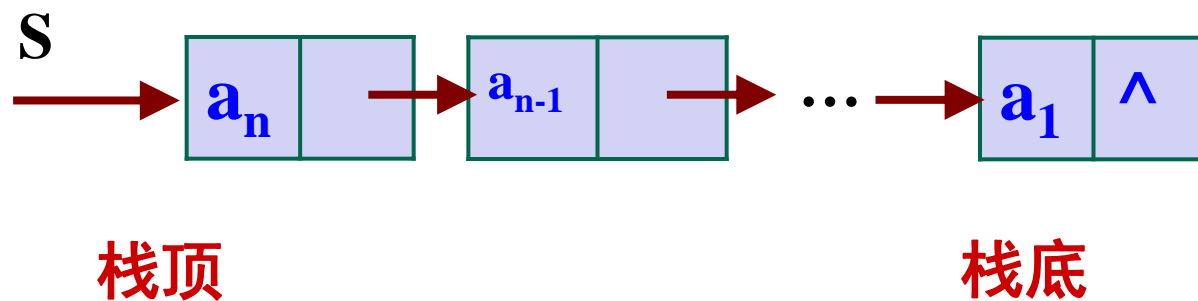
e **B**



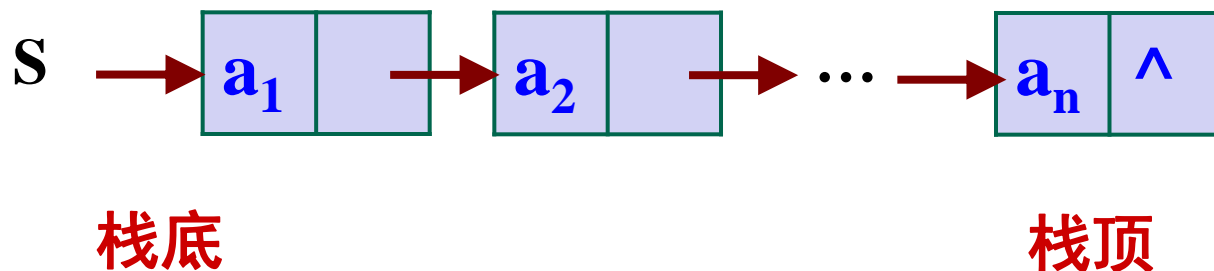
栈的链式存储结构

链栈：栈的链式存储结构，通常用单链表来表示。它是运算是受限的单链表，即插入和删除操作仅限制在链表头位置上进行。栈顶指针就是链表的头指针。





如果插入和删除操作仅限制在链表尾进行呢？



链栈中结点类型的声明:

```
typedef struct StackNode {  
    SElemType      data;      // 数据域  
    struct StackNode *next; // 指针域  
}SNode, *LinkStack;
```

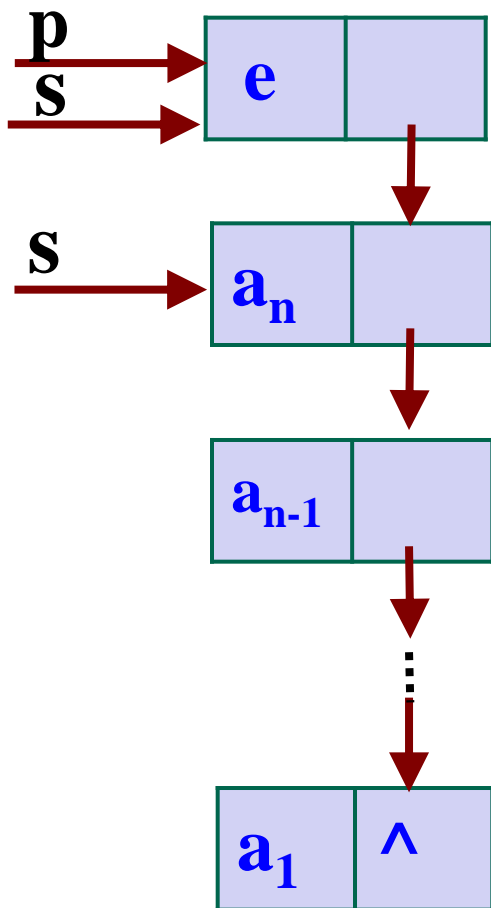
链栈基本操作的实现

1. 链栈的初始化 InitStack (&S)

```
void InitStack(LinkStack &S)
{ //构造一个空栈S，栈顶指针置空
    S=NULL;
}
```

2. 入栈 `Push(&S, e)`

在栈顶插入一个新的元素。



```
void Push(LinkStack &S, SElemType e)
{ //插入元素e为新的栈顶元素
  p=(LinkStack)malloc(sizeof(SNode));
  //生成x新结点
  p->data=e;
  p->next=S; //将新结点插入栈顶

  S=p; //修改栈顶指针为p
}
```

链栈的入栈过程

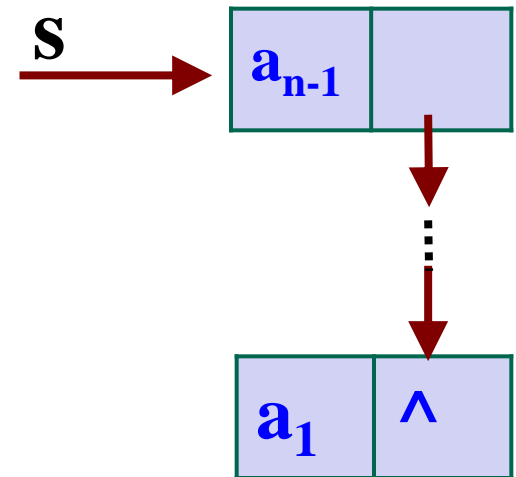
3. 出栈 Pop(&S, &e)

将栈顶元素删除。

```
Status Pop(LinkStack &S, SElemType &e)
{ //删除S的栈顶元素，用e返回其值

    if (S == NULL) return ERROR; //栈空
    e=S->data;
    p=S;    //用p临时保存栈顶元素空间，以备释放
    S=S->next; //修改栈顶指针
    free(p);  //释放原栈顶元素的空间

    return OK;
}
```



链栈的出栈过程

3.2 栈的应用

由于栈结构具有的后进先出的固有特性，致使栈成为程序设计中常用的工具。

1 . 数制转换

十进制数N  其它R进制数

除R取余数法：

例如 $(1348)_{10} = (2504)_8$ ，其运算过程如下：

	8	余数
1348		
168		4
21		0
2		5
0		2

2
5
0
4

2	5	0	4
---	---	---	---

```
void conversion (int N)
```

```
{// 对于一个非负十进制整数，打印输出与其等值的八进制数
```

```
    InitStack(S);    // 构造空栈S
```

```
    while(N){
```

```
        Push(S, N%8); // 把N与8求余得到的数入栈
```

```
        N=N/8;      //N更新为N与8的商
```

```
    }
```

```
    while(! StackEmpty(S)){ //当栈S非空时，循环
```

```
        Pop(S, e);          //弹出栈顶元素e
```

```
        printf(“%d”,e);     //输出e
```

```
    }
```

```
}
```

2. 括号匹配的检验

假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，即 $([] ())$ 或 $[([] [])]$ 等为正确的格式， $[(])$ 或 $([())$ 或 $(()])$ 均为不正确的格式。检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。例如考虑下列括号序列：

[([]	[])]
1	2	3	4	5	6	7	8

```
status Matching(string& exp) {  
// 检验表达式中所含括弧是否正确嵌套，若是，则返回OK，否则返回ERROR  
InitStack(S);  
int state = 1; //state表明状态， state=1,正确； state=0,已出错。  
while (i<=length(exp) && state) {  
    switch exp[i] {  
        case 左括弧: { Push(S,exp[i]); i++; break; }  
        case ")":  
            { if (! StackEmpty(S) && GetTop(S) = "(")  
                { Pop(S,e); i++; }  
            else { state = 0; }  
            break;  
        }  
        ..... }  
    }  
    if ( state && StackEmpty(S) ) return OK  
    else return ERROR;  
}
```

3 行编辑程序

一个简单的行编辑程序的功能是：接受用户从终端输入的程序或数据，并存入用户的数据区。

“每接受一个字符即存入用户数据区”不恰当。

较好的做法：设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区。允许用户输入出差错，并在发现有误时可以及时更正。

例如，可用一个退格符“#”表示前一个字符无效；可用一个退行符“@”，表示当前行中的字符均无效。

例如，

假设从终端接受了这样两行字符： 则实际有效的是下列两行：

whli###ilr#e (s#*s)

while (*s)

outcha@putchar(*s=#++); putchar(*s++);

可设这个输入缓冲区为一个栈结构，每当从终端接受一个字符后先作如下判别：

- a. 如果它既不是退格符也不是退行符，则将它进栈；
- b. 如果是退格符，则从栈顶删去一个字符；
- c. 如果是退行符，则将栈清空。

```
void LineEdit() {
```

```
// 利用字符栈S，从终端接收一行并传送至调用过程 的数据区。
```

```
InitStack(S); //构造空栈S
```

```
ch = getchar(); //从终端接收第一个字符
```

```
while (ch != EOF) { //EOF为全文结束符
```

```
    while (ch != EOF && ch != '\n') {
```

```
        switch (ch) {
```

```
            case '#' : Pop(S, c); break; // 仅当栈非空时退栈
```

```
            case '@' : ClearStack(S); break; // 重置S为空栈
```

```
            default : Push(S, ch); break;
```

```
                // 有效字符进栈，未考虑栈满情形
```

```
        }
```

```
        ch = getchar(); // 从终端接收下一个字符
```

```
}
```

```
    将从栈底到栈顶的字符传送至调用过程的数据区；
```

```
    ClearStack(S); // 重置S为空栈
```

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

```
}
```

```
    DestroyStack(S);
```

```
}
```


4. 迷宫求解

求迷宫中从入口到出口的所有路径是一个经典的程序设计问题。由于计算机解迷宫时，通常用的是“**穷举求解**”的方法，即从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续探索，直至所有可能的通路都探索到为止。

为了保证在任何位置上都能沿原路退回，显然需要用一个**后进先出**的结构来保存从入口到当前位置的路径。因此，在求迷宫通路的算法中应用“栈”也就是自然而然的事了。

假设迷宫如下图所示:

[illegible]

假设“当前位置”指的是“在搜索过程中某一时刻所在图中某个方块位置”

,

则求迷宫中一条路径的算法的基本思想是：

若当前位置“可通”，则纳入“当前路径”，并继续朝“下一位置”探索，即切换“下一位置”为“当前位置”，如此重复直至到达出口；

若当前位置“不可通”，则应顺着“来向”退回到“前一通道块”，然后朝着除“来向”之外的其他方向继续探索；

若该通道块的四周四个方块均“不可通”，则应从“当前路径”上删除该通道块。

所谓“下一位置”指的是“当前位置”四周四个方向（东、南、西、北）上相邻的方块。假设以栈S记录“当前路径”，则栈顶中存放的是“当前路径上最后一个通道块”。由此，“纳入路径”的操作即为“当前位置入栈”；“从当前路径上删除前一通道块”的操作即为“出栈”。

求迷宫中一条从入口到出口的路径的算法可简单描述如下：

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被探索，

 则设定新的当前位置为：沿顺时针方向旋转找到的栈顶位置的
 下一相邻块；

 若栈不空，但栈顶位置的四周均不可通，

 则 { 删去栈顶位置； // 从路径中删去该通道块

 若栈不空，则重新测试新的栈顶位置，

 直至找到一个可通的相邻块或出栈至栈空；

 }

 }

} 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
```

5. 表达式求值

表达式求值是编译系统中最基本的一个问题, 其实现是栈应用的一个典型例子。

要把一个表达式翻译成正确求值的一个机器指令序列, 或者直接对表达式求值, 首先要能够正确解释表达式。**算符优先算法**就是根据算术四则运算法则确定的运算优先关系, 实现对表达式的编译或解释执行的。

表达式的构成：操作数+运算符+界限符（如括号）

操作数：常数、变量或常量标识符

运算符： / * + -

界限符： () #

运算符和界限符统称**算符**

算符优先关系表：

表达式中任何相邻算符 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

由四则运算法则， 可得到如下的算符优先关系表：

$\theta_1 \backslash \theta_2$	+	-	*	/	()	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

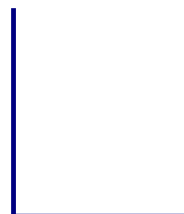
注： θ_1 、 θ_2 是相邻算符， θ_1 在左， θ_2 在右

算符间的优先关系表

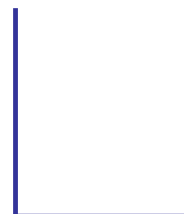
算符优先算法：

在算符优先算法中，建立了两个工作栈。

- **OPTR栈** 用以保存运算符；
- **OPND栈** 用以保存操作数或运算结果。



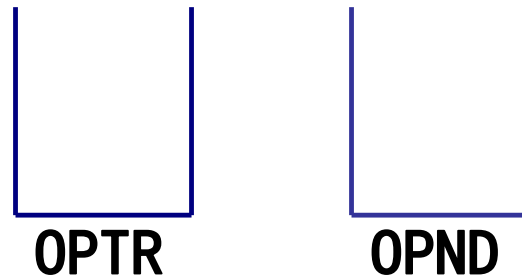
OPTR



OPND

算法基本思想：

- 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，则和 OPTR 栈的栈顶运算符比较优先权后作相应操作，直至整个表达式求值完毕。



OperandType EvaluateExpression()

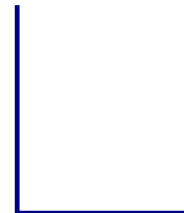
{//算术表达式求值的算符优先算法。设OPTR和OPND分别为运算符栈和运算数栈

InitStack(OPTR); //初始化OPTR栈

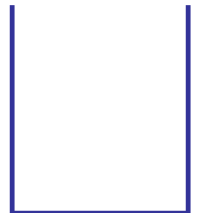
Push (OPTR, #); //将表达式起始符 “#” 压入OPTR 栈

InitStack(OPND);

c=getchar();



OPTR



OPND

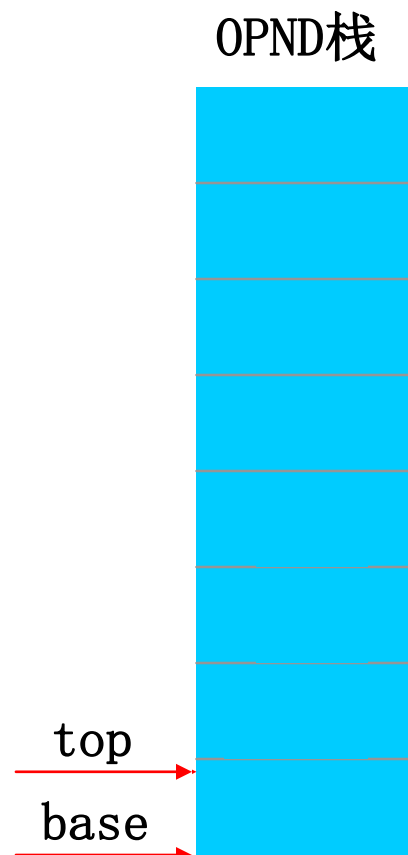
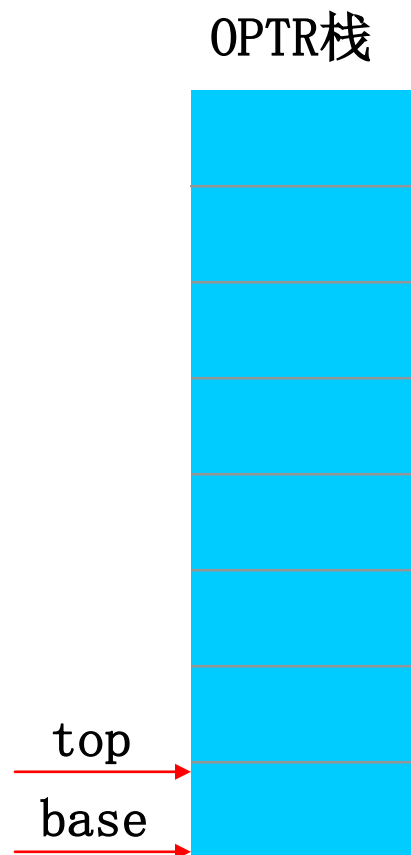
```

while(c!=' #' || GetTop(OPTR)!='#')
{
    //表达式没有扫描完毕或OPTR的栈顶元素不是“#”
    if (! In (c, OP)) {    // In(c, OP)判断c是否是运算符的函数
        Push(OPND, c); c=getchar( );} //c不是运算符则进OPND栈
    else
        switch (Precede(GetTop(OPTR), c) { //比较优先级
            case '<':    // 新输入的算符c优先级高, c进栈
                Push(OPTR, c); c=getchar( ); break;
            case '=':    // 脱括号并接收下一字符
                Pop(OPTR, x); c=getchar( ); break;
            case '>':    //新输入的算符c优先级低, 即栈顶算符优先权高
                Pop(OPTR, theta);
                Pop(OPND, b); Pop(OPND, a); //弹出OPND栈的两个运算数
                Push(OPND, Operate(a, theta, b)); //将运算结果入栈OPND
                break;
        }
    }
    return GetTop(OPND); //OPND栈顶元素即为表达式求值结果
}

```

表达式求值示意图: $5+6\times(1+2)-4=19$

读入表达式过程: $5+6\times(1+2)-4\#$



$$1+2=3$$

$$6\times 3=18$$

$$5+18=23$$

$$23-4=19$$

3.3 栈与递归

1. 什么是递归

递归是算法设计中最常用的手段，它通常把一个大型复杂问题的描述和求解过程变得简洁和清晰。因此递归算法常常比非递归算法更易设计。

递归定义：

简单地说，一个用自己定义自己的概念，称为递归定义。

例 $n! = 1 * 2 * 3 * 4 * (n-1) * n$

$n!$ 递归定义 $\begin{cases} n! = 1 & \text{当 } n=1 \text{ 时} \\ n! = n (n-1)! & \text{当 } n > 1 \text{ 时} \end{cases}$

用 $(n-1)!$ 定义 $n!$

递归函数：一个直接调用自己或通过一系列调用间接调用自己的函数称为递归函数。

```
A( ) {  
    ...  
    A( );  
    ...  
}
```

A 直接调用自己

<pre>B() { ... C(); ... }</pre>	<pre>C() { ... B(); ... }</pre>
---	---

B间接调用自己

2. 递归算法的编写

- 1) 将问题用递归的方式描述（定义）
- 2) 根据问题的递归描述（定义）编写递归算法

问题的递归描述（定义）

递归定义包括两项

递归项：

将问题分解为与原问题性质相同，但规模较小的问题；

基本项（终止项）：描述递归终止时问题的求解；

例 $n!$ 的递归定义

基本项：	$n! = 1$	当 $n = 1$
递归项：	$n! = n (n-1)!$	当 $n > 1$

例1 编写求解 阶乘 $n!$ 的递归算法

首先给出阶乘 $n!$ 的递归定义

$n!$ 的递归定义

基本项: $n!=1$ 当 $n=1$

递归项: $n!=n (n-1)!$ 当 $n> 1$

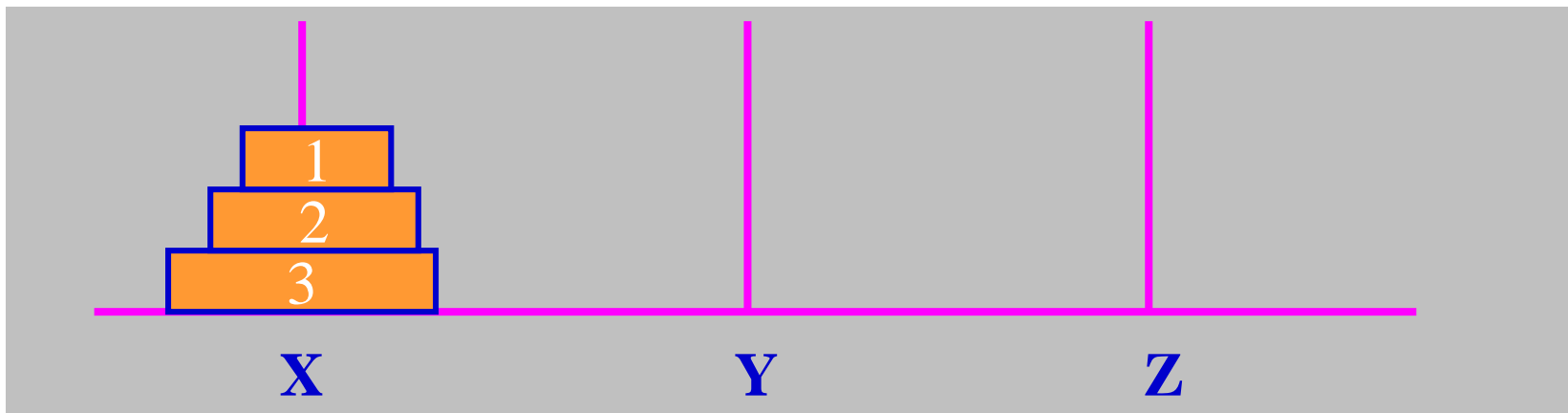
有了问题的递归定义，很容易写出对应的递归算法：

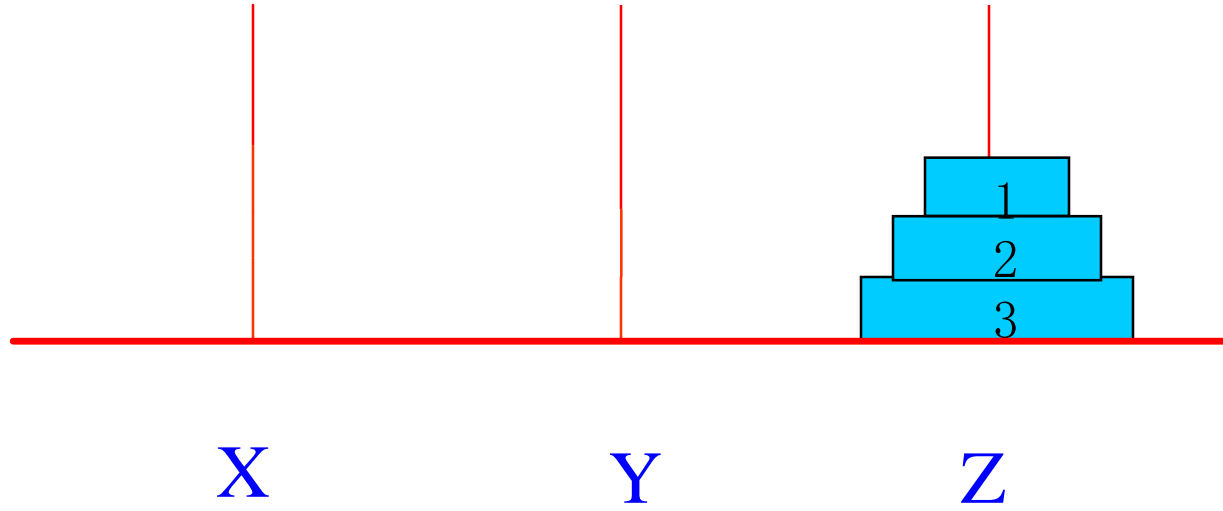
```
int fact (int n) {  
    //算法功能是求解并返回 $n$ 的阶乘  
    if (n==1) return (1) ;  
    else return (n*fact (n-1)) ;  
}
```

例2. 编写求解Hanoi塔问题的递归算法

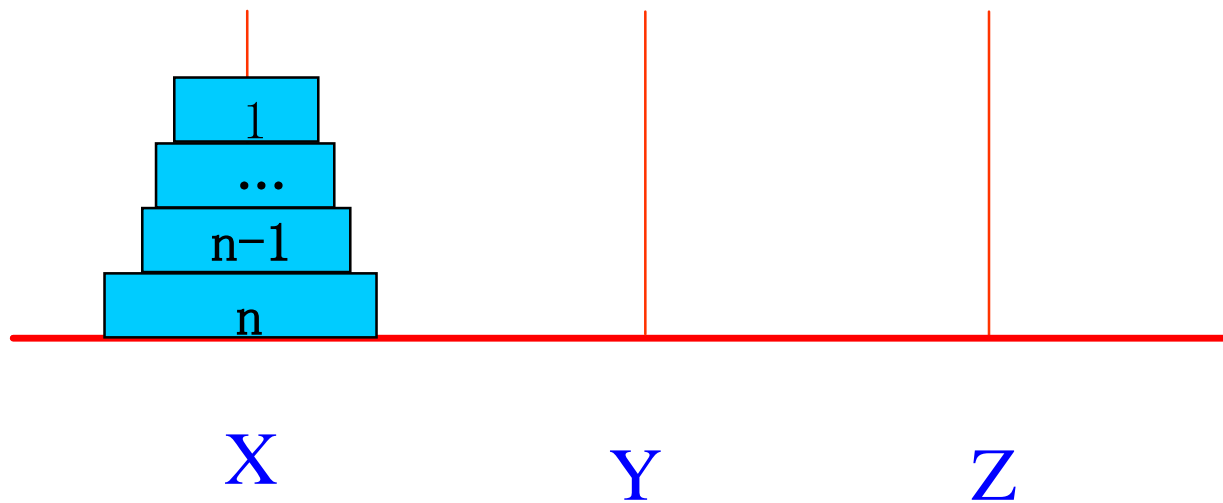
有三个各为X, Y, Z的塔座, 在X上有n个大小不同, 依小到大编号为1, 2...n的圆盘。现要求将X上的n个圆盘移至Z上, 并仍以同样顺序叠放, 圆盘移动时必须遵守下列原则:

- 1) 每次移动一个盘子;
- 2) 圆盘可以放在X, Y, Z中的任一塔座上;
- 3) 任何时刻都不能将较大的圆盘压放在较小圆盘之上;



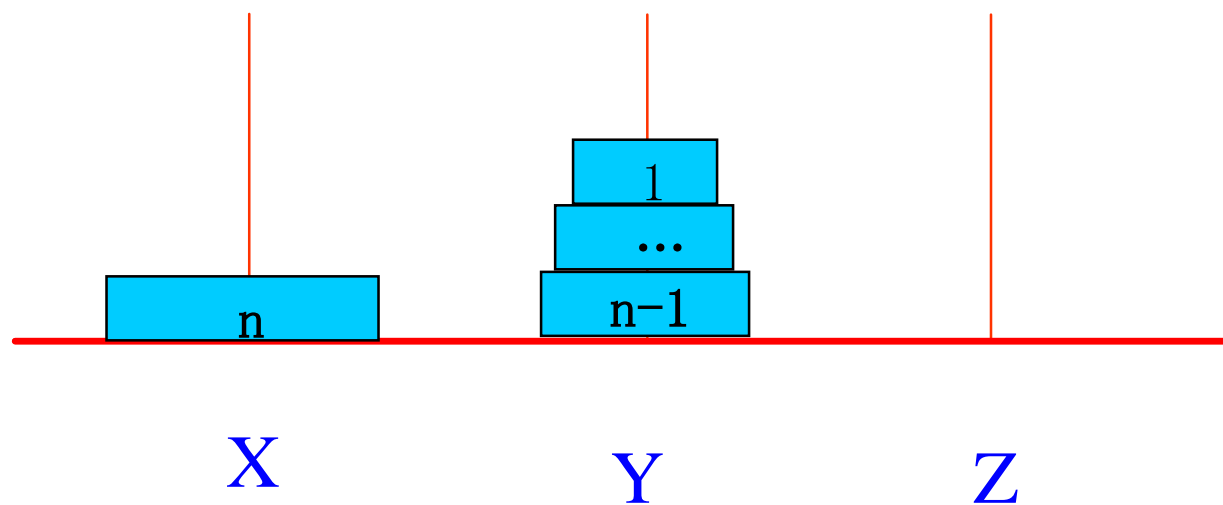


n $X \xrightarrow{Y} Z$



n $X \xrightarrow{Y} Z$

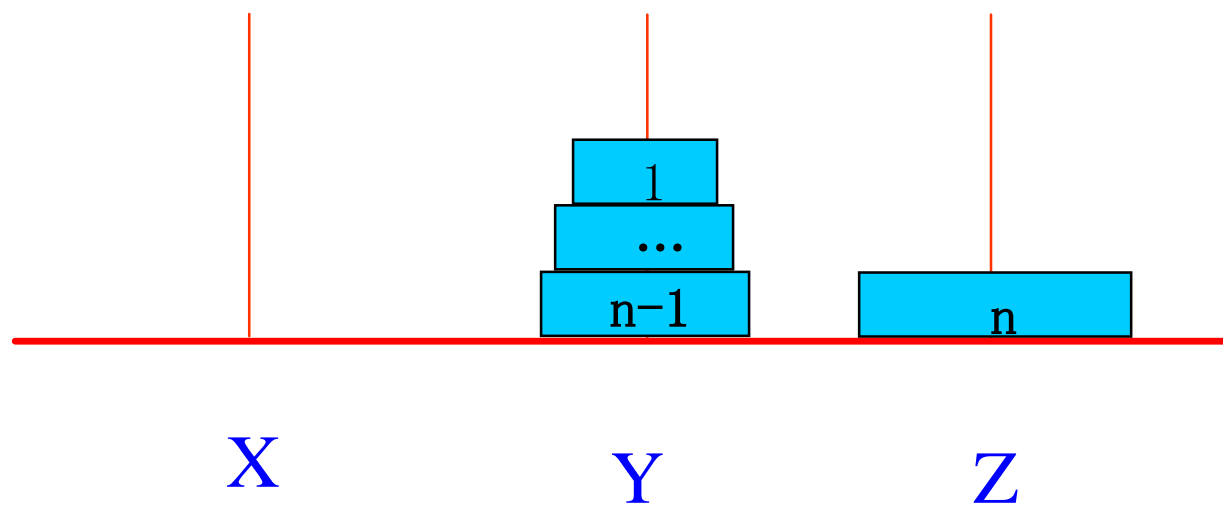
(1) $n-1$ $X \xrightarrow{Z} Y$



n X \xrightarrow{Y} Z

(1) n-1 X \xrightarrow{Z} Y

(2) 1 X $\xrightarrow{\quad}$ Z

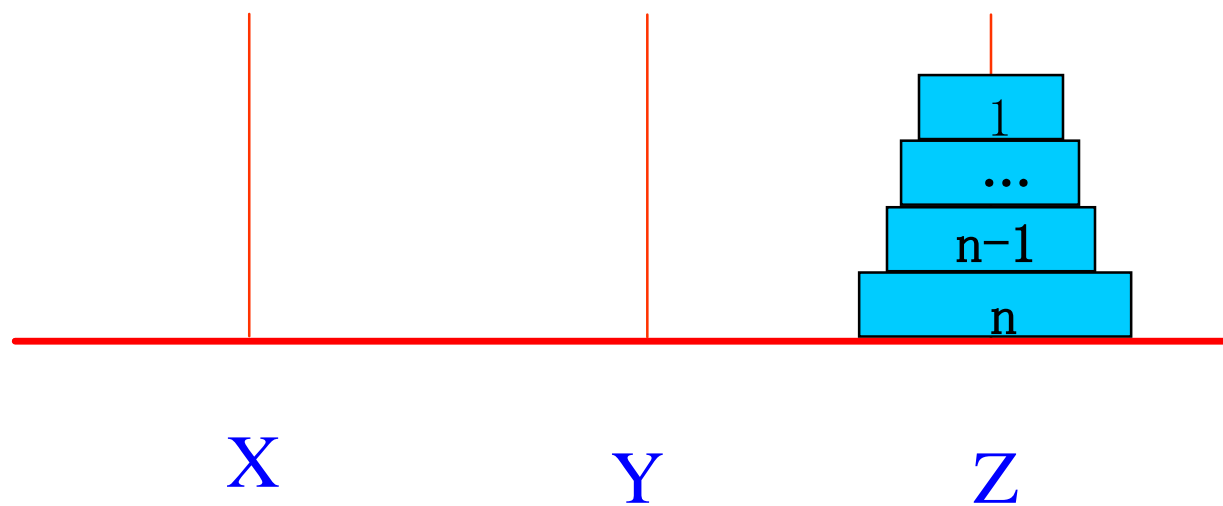


n $X \xrightarrow{Y} Z$

(1) $n-1$ $X \xrightarrow{Z} Y$

(2) 1 $X \xrightarrow{\quad} Z$

(3) $n-1$ $Y \xrightarrow{X} Z$



首先给出求解Hanoi塔问题的递归定义

基本项： $n=1$ 时，将1号圆盘从X移至Z；

递归项： $n>1$ 时，

将X上1— $n-1$ 号圆盘移至Y；

将X上的 n 号圆盘从X移至Z；

将Y上1— $n-1$ 号圆盘从Y移至Z；

将规模为 n 的问题的求解分解为规模为 $n-1$ 的问题的求解

有了问题的递归定义，很容易写出对应的递归算法：

```
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(“%d Move disk %di 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移动到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
```

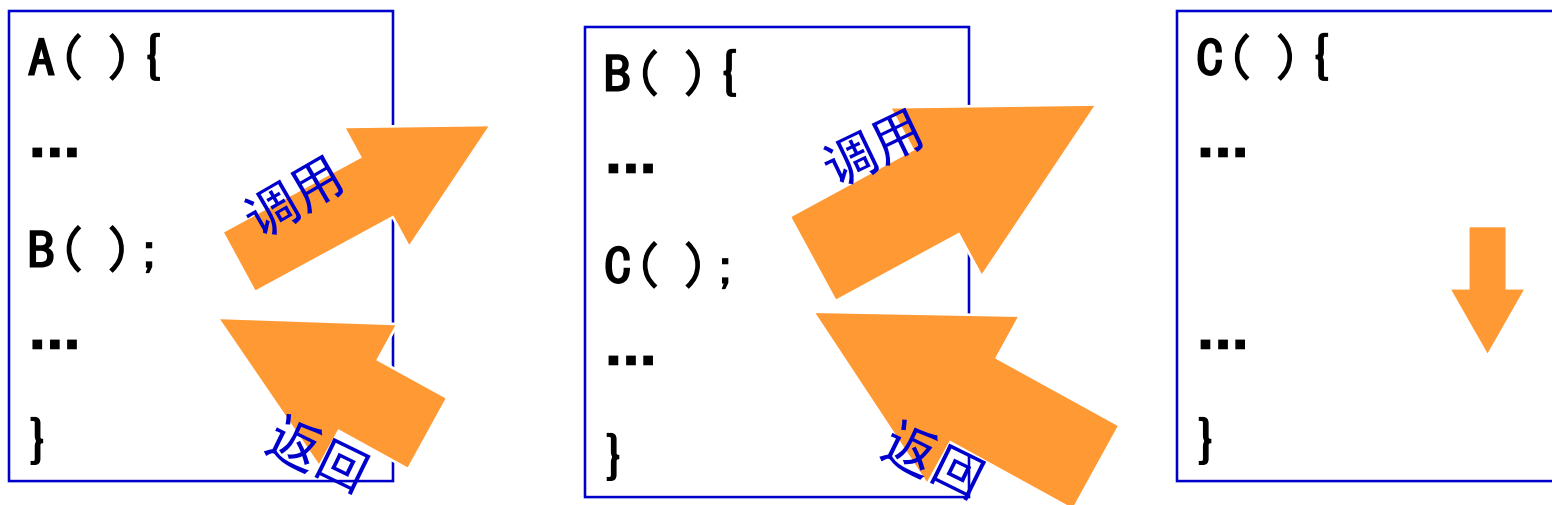
```
    }
```

```
}
```

算法3.5 Hanci塔问题

3 递归函数的实现

在递归函数的执行中，需多次自己调用自己，递归函数是如何执行的？先看一般函数的调用机制如何实现的。



函数调用顺序 A → B → C

函数返回顺序 C → B → A

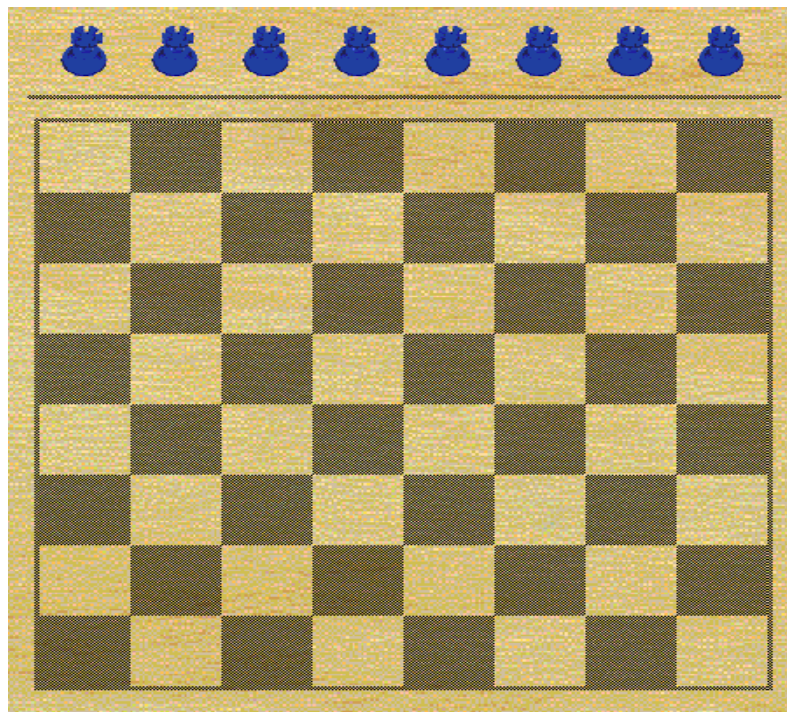
后调用的函数先返回

函数调用机制可通过栈来实现

计算机正是利用栈来实现函数的调用和返回的，函数之间的信息传递和控制转移通过“栈”来实现

八皇后问题

设在初始状态下在国际象棋棋盘上没有任何棋子(皇后)。然后顺序在第1行, 第2行, ...。第8行上布放棋子。在每一行中有8个可选择位置, 但在任一时刻, 棋盘的合法布局都必须满足3个限制条件, 即**任何两个棋子不得放在棋盘上的同一行、或者同一列、或者同一斜线上**。试编写一个算法, 求解并输出此问题的所有合法布局。



背包问题

设有一个背包可以放入的物品的重量为 s ，现有 n 件物品，重量分别为 $w[1]$ ， $w[2]$ ， \dots ， $w[n]$ 。问能否从这 n 件物品中选择若干件放入此背包中，使得放入的重量之和正好为 s 。

如果存在一种符合上述要求的选择，则称此背包问题有解（或称其解为真）；否则称此背包问题无解（或称其解为假）。试设计求解背包问题的算法

上面看到：栈结构后进先出的特征在程序设计中的应用，栈在实现函数递归调用中的作用；以及如何编写递归算法（递归函数）。

在后面的章节中，还将利用递归函数实现树和图的基本操作，这里同学们要好好理解如何编写递归算法（递归函数）。

小 结

1. 栈是限定仅能在表尾一端进行插入、删除操作的线性表；
2. 栈的元素具有后进先出的特点；
3. 栈顶元素的位置由一个称为栈顶指针的变量指示，进栈、出栈操作要修改栈顶指针；

第三章 习题一

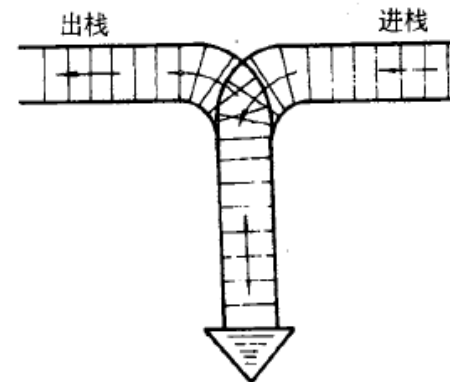
1、 P21-24 3. 1, 3. 4, 3. 17

P21.

3.1 若按教科书3.1.1节中图3.1(b)所示铁道进行车厢调度(注意:两侧铁道均为单向行驶道), 请回答:

(1) 如果进站的车厢序列为123, 则可能得到的出站车厢序列是什么?

(2) 如果进站的车厢序列为123456, 则能否得到435612和135426的出站序列, 并请说明为什么不能得到或者如何得到(即写出以'S'表示进栈和以'X'表示出栈的栈操作序列)。



3.4 简述以下算法的功能

（其中栈的元素类型**SElemType**为int）：

1.

```
status algo1 (Stack S){
```

```
    int i, n, A[255];
```

```
    n=0;
```

```
    while(!EmptyStack(S)) {n++; Pop(S, A[n]); }
```

```
    for(i=1; i<=n; i++) Push(S, A[i]);
```

```
    }
```

2.

```
status algo2(Stack S, int e){  
    Stack T; int d;  
    InitStack(T);  
    while(!EmptyStack(S)){  
        pop(S, d);  
        if (d!=e) Push( T, d);  
    }  
    while(!EmptyStack(T))  
    { Pop(T, d);  
      Push( S, d);  
    }  
}
```

3. 17. 试写一个算法，判断依次读入的一个以@为结束符的字母序列，是否为形如 ‘序列1&序列2’ 模式的字符序列。其中序列1和序列2中都不含字符‘&’，且序列2 是序列1的逆序列。例如，‘a+b&b+a’ 是属该模式的字符序列，而 ‘1+3&3-1’ 则不是。

[提示]:

1. 边读边入栈，直到&

2. 边读边出栈边比较，直到.....

队 列

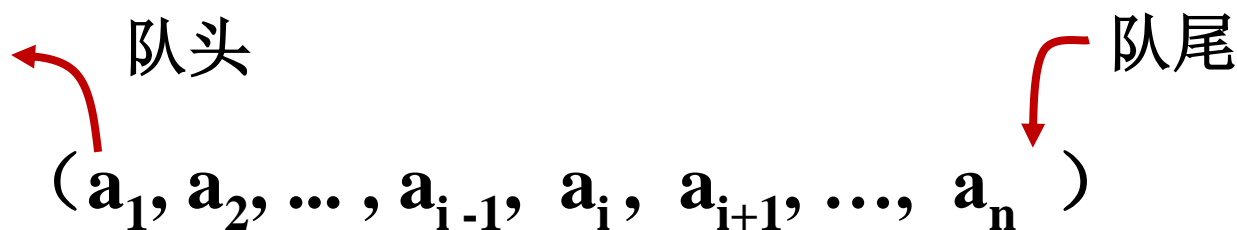
3. 4 队列的定义和链队列

3. 5 循环队列

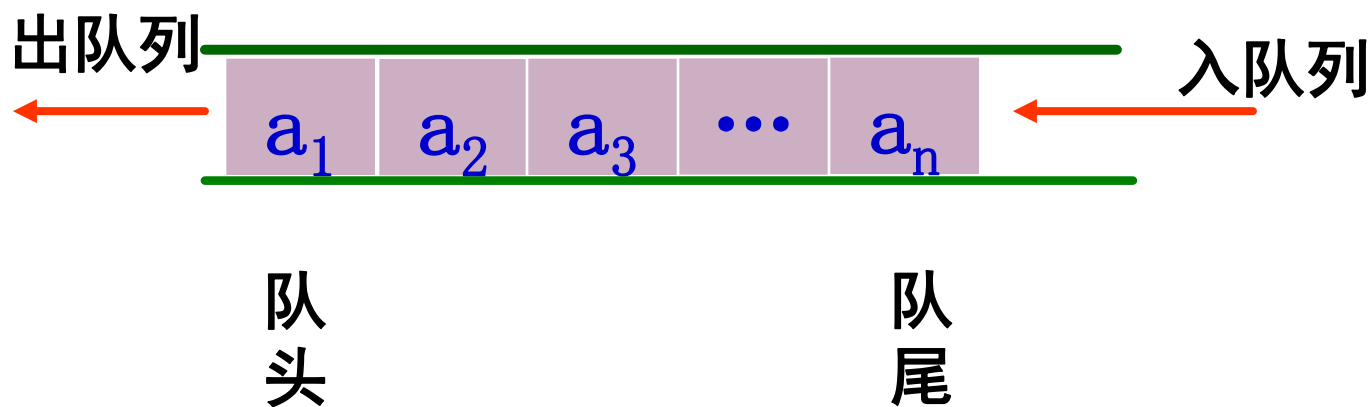
3.4 队列的定义和链队列

1. 队列的定义

队列 (Queue) : 只允许在表的一端进行插入, 而在另一端进行删除的线性表。允许插入的一端称为**队尾 (rear)**, 允许删除的一端称为**队头 (front)**。



- 当队列中没有元素时称为**空队列**。在空队列中依次加入元素 a_1, a_2, \dots, a_n 之后， a_1 是队头元素， a_n 是队尾元素。
- 队列也称为**先进先出(First In First Out)**线性表。



队列的基本操作:

- ◆ 初始化队列 `InitQueue(&Q)`: 构造一个空队列Q。
- ◆ 销毁队列 `DestroyQueue (&Q)`: 释放队列Q占用的内存空间。
- ◆ 清空队列 `ClearQueue (&Q)`: 将队列Q重置为空。
- ◆ 判断队列是否为空 `QueueEmpty (S)`: 若队列Q为空, 则返回TRUE, 否则FALSE。
- ◆ 求队列的长度 `QueueLength (Q)`: 返回队列Q中元素个数。

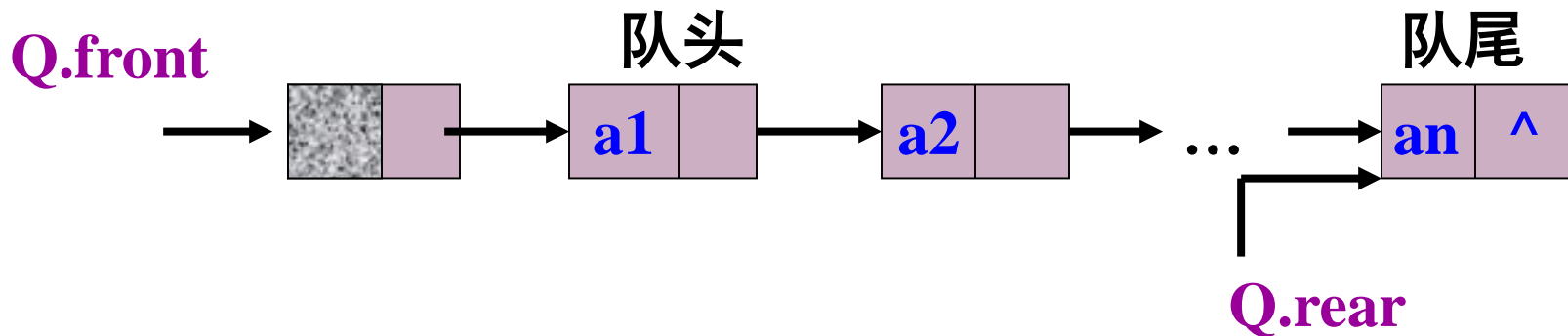
- ◆ **取队头元素** `GetHead(Q, &e)`: 用e返回Q的队头元素。
- ◆ **入队** `EnQueue(&Q, e)`: 插入元素e为新的队尾元素。
- ◆ **出队** `DeQueue(&Q, &e)`: 删除Q的队头元素，并用e返回其值。
- ◆ **遍历队列** `QueueTraverse(Q)`: 从队头到队尾依次对Q的每个元素进行访问。

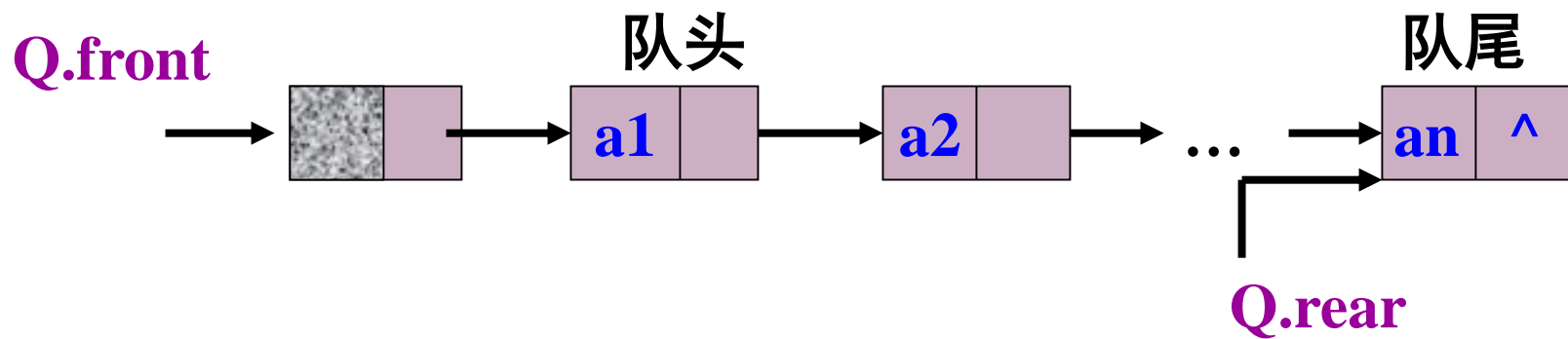
队列的存储结构

- (1). **链队列**—队列的链式表示
- (2). **循环队列**—队列的顺序表示

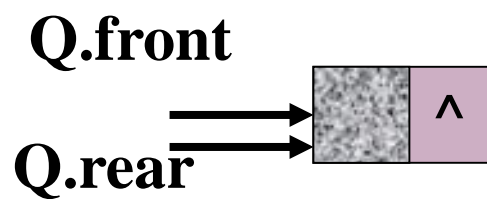
2 链队列—队列的链式表示和实现

用链表表示的队列简称为**链队列**。它是限制仅在**表头删除**和**表尾插入**的单链表。





非空链队列

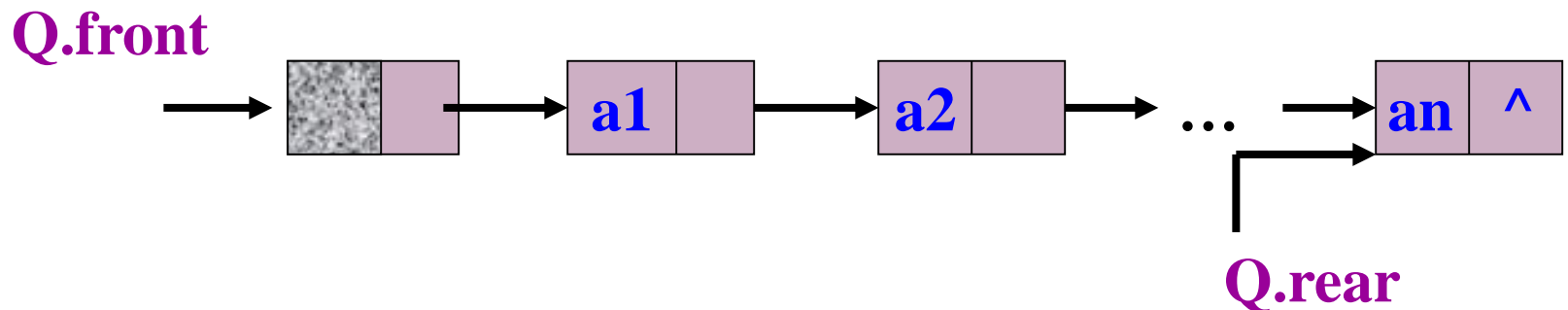


空队列

链队列的类型声明如下：

```
typedef struct QNode{ //链队列结点的类型
    QElemType data;
    struct QNode *next;
}QNode,*QueuePtr;
```

```
typedef struct{ //链队列的类型
    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;
```



链队列基本操作的实现

1. 初始化 InitQueue (&Q)

```
Status InitQueue(LinkQueue &Q)
```

```
{ //构造一个空队列Q
```

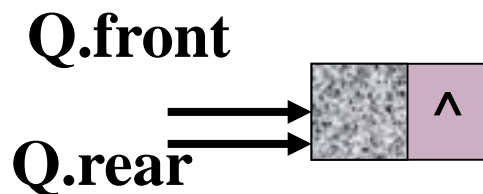
```
    Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
```

```
    //生成新结点作为头结点，队头和队尾指针都指向此结点
```

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

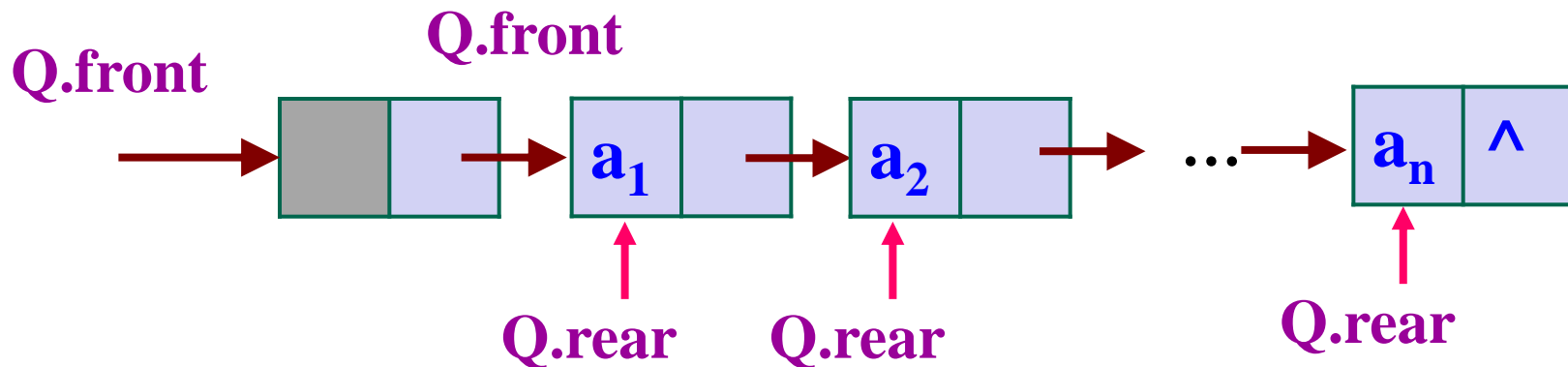
```
    return OK;
```

```
}
```



2. 销毁队列 DestroyQueue(&Q)

```
Status DestroyQueue_L(LinkQueue &Q)
{ //销毁队列Q
  while(Q.front)
  {
    Q.rear=Q.front->next;
    free(Q.front);
    Q.front=Q.rear
  }
  return OK;
}
```



3. 入队 EnQueue(&Q, e)

```
void EnQueue(LinkQueue &Q, QElemType e)
```

```
{//插入元素e为Q的新的队尾元素
```

```
    p=(QueuePtr)malloc(sizeof(QNode)); //为e分配结点空间
```

```
    p->data=e;    //将新结点数据域置为e
```

```
    p->next=NULL;
```

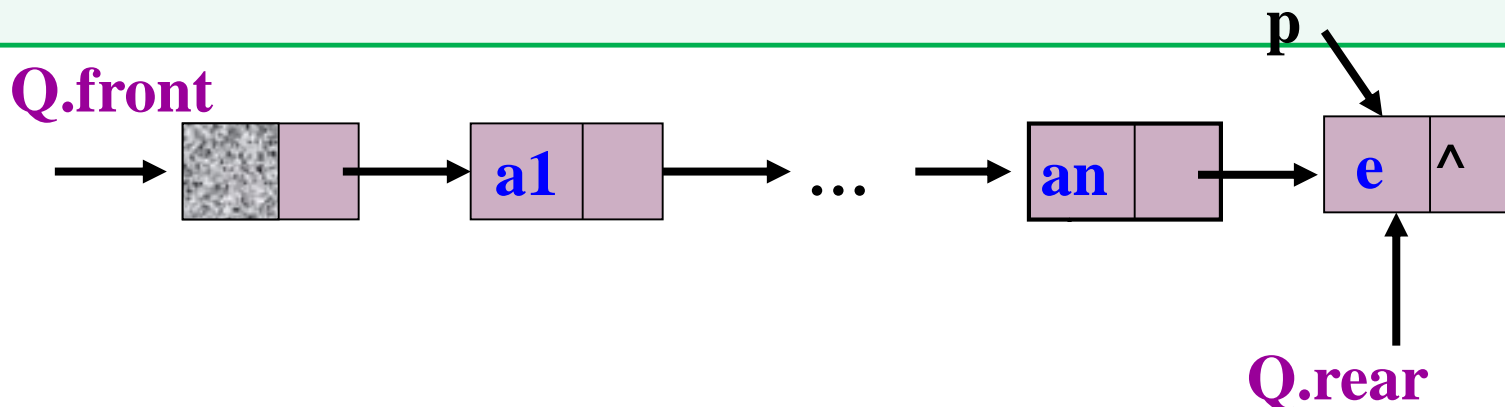
```
    Q.rear->next=p;
```

//将新结点插入到队尾

```
    Q.rear=p;
```

//修改队尾指针

```
}
```



4. 出队 DeQueue(&Q, &e)

Status DeQueue(LinkQueue &Q, QElemType &e)

{//删除Q的队头元素,用e返回其值

if(Q.front==Q.rear)return ERROR; //若队列空, 返回ERROR

p=Q.front->next; // p指向队头元素结点

e=p->data;

Q.front->next=p->next; // 修改链队列头结点指针

if(Q.rear==p) Q.rear=Q.front;

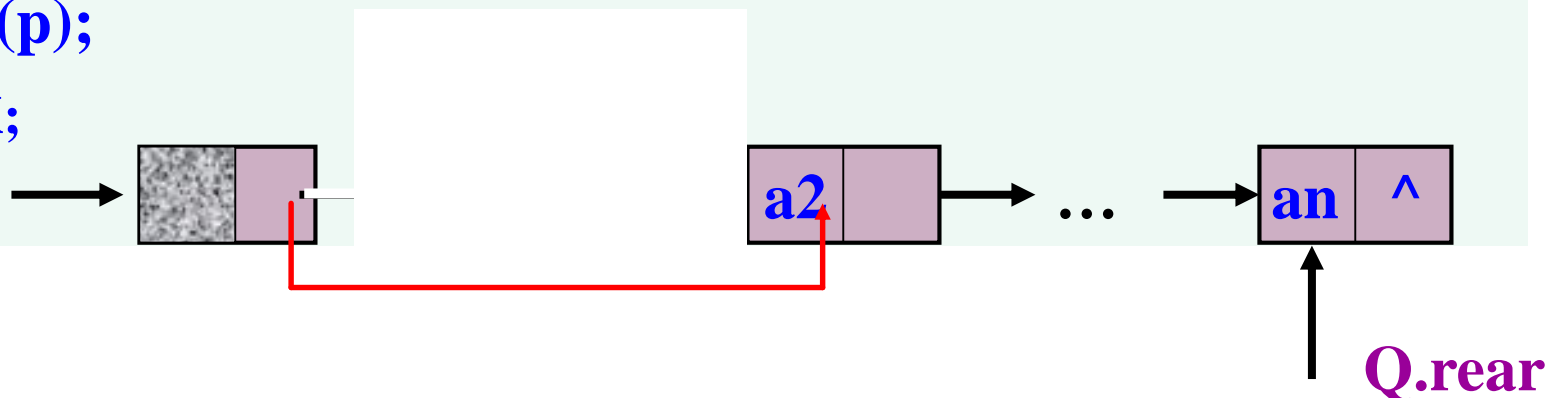
// 对于链队列只有一个元素结点的情况要同时修改队尾指针

free(p);

return OK;

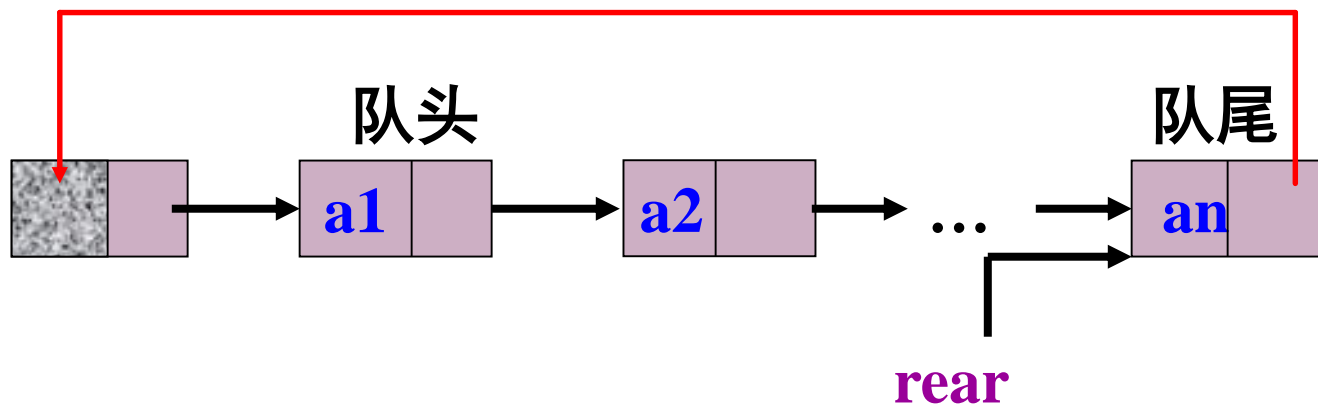
}

Q.front



思考题：

假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点（注意不设头指针），相应的队列的操作，如初始化、入队、出队等如何来具体实现？

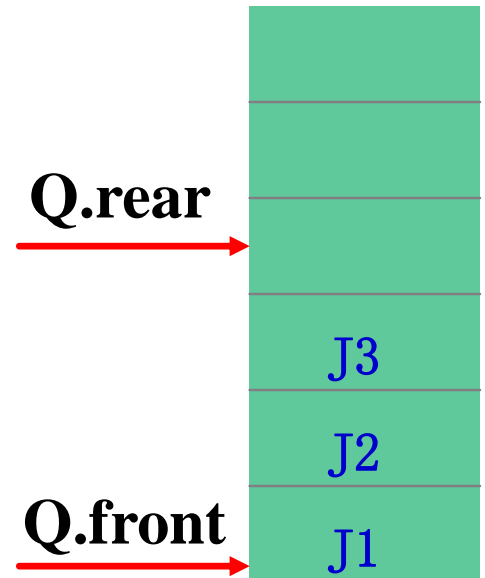


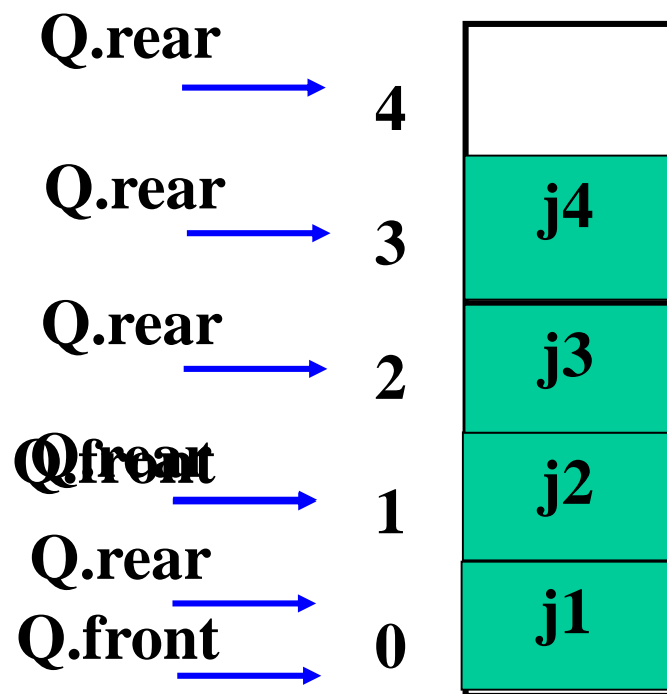
```
typedef struct QNode{ //链队列结点的类型
    QElemType data;
    struct QNode *next;
}QNode,*QueuePtr;
```

4.5 循环队列—队列的顺序表示和实现

队列的顺序存储结构称为顺序队列，是用一组地址连续的存储单元依次存放从队列头到队列尾的元素。

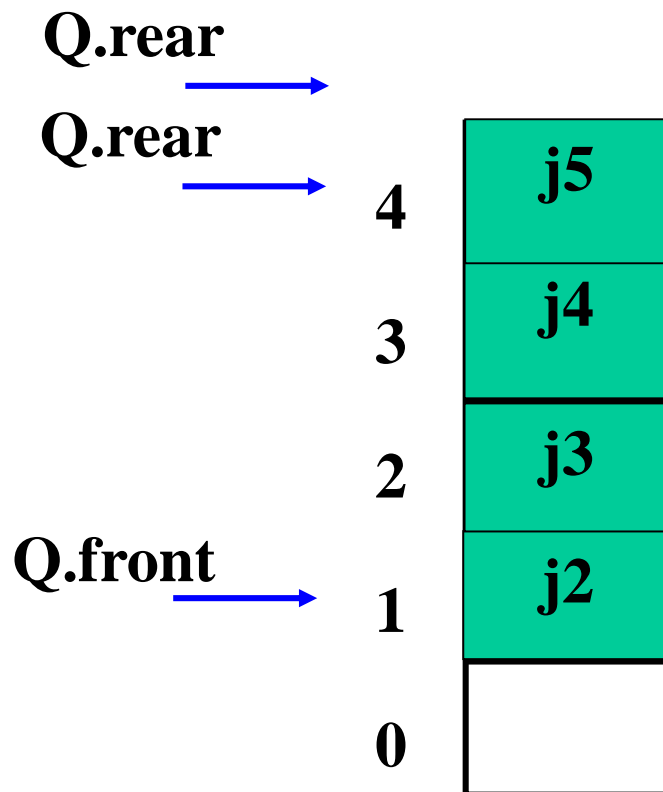
由于队列的队头和队尾元素的位置是变化的，因而要设两个指针`front`和`rear`分别指示队头和队尾元素的位置。





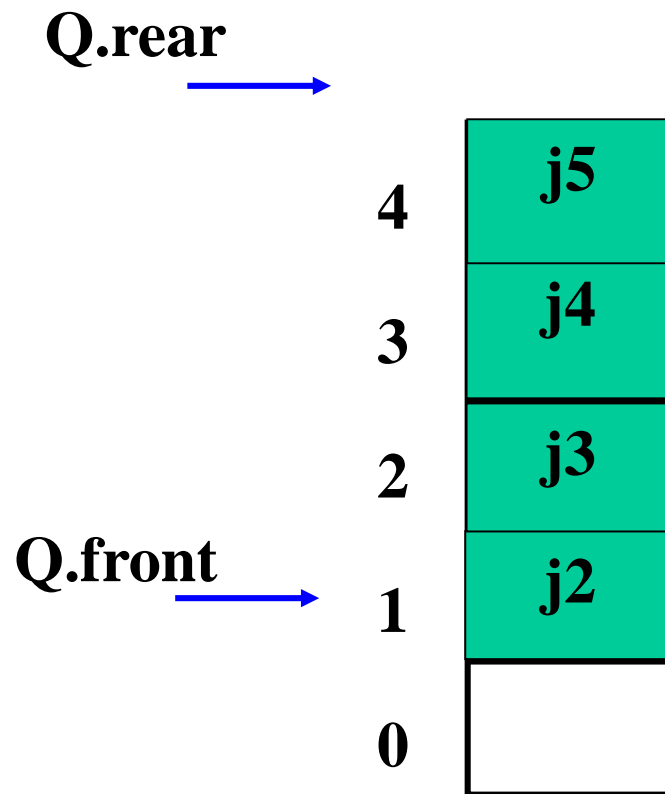
顺序队列中头、尾指针和元素之间的关系

- 初始化建空队列时，令 $front=rear=0$ 。
- 每当插入新的队列尾元素时，尾指针加1；
- 每当删除队列头元素时，头指针加1。



顺序队列中头、尾指针和元素之间的关系

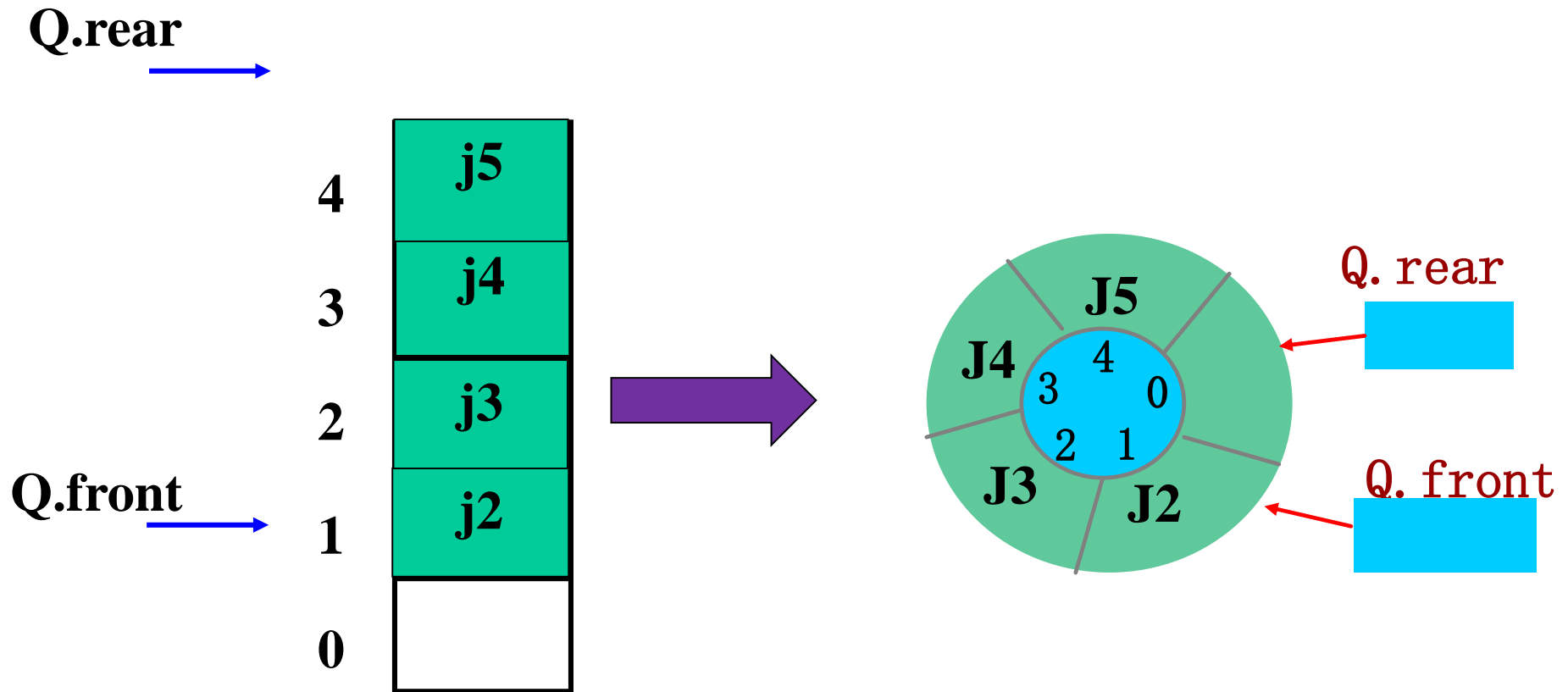
在非空队列里，头指针始终指向队头元素，而尾指针始终指向队尾元素的下一位置。



顺序队列中头、尾指针和元素之间的关系

假溢出：从图示中可看出，当J5 入队后，队尾指针Q. rear越界，不可能再插入新的队尾元素，但是另一方面，队列的实际可用空间并未占满。

解决办法：将顺序队列设想为首尾相连的环状空间，如图，当Q.rear值超出队列空间的最大位置时，令Q.rear= 0，使队列空间能“循环”使用。这种循环使用空间的队列称为循环队列。



这种循环意义下的加1操作可以描述为：

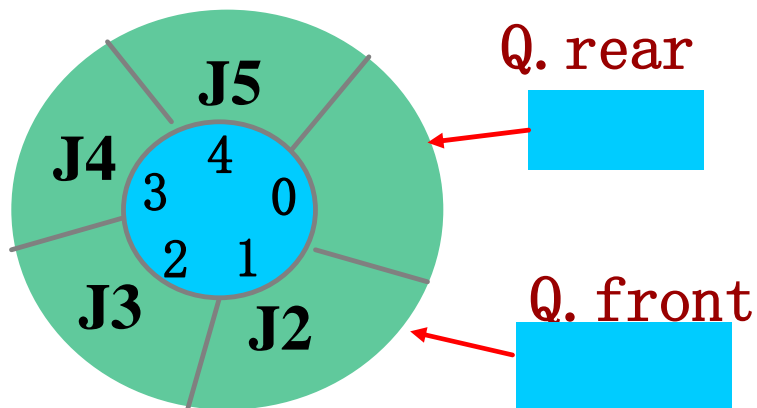
```
if (i+1 == MAXSIZE)
```

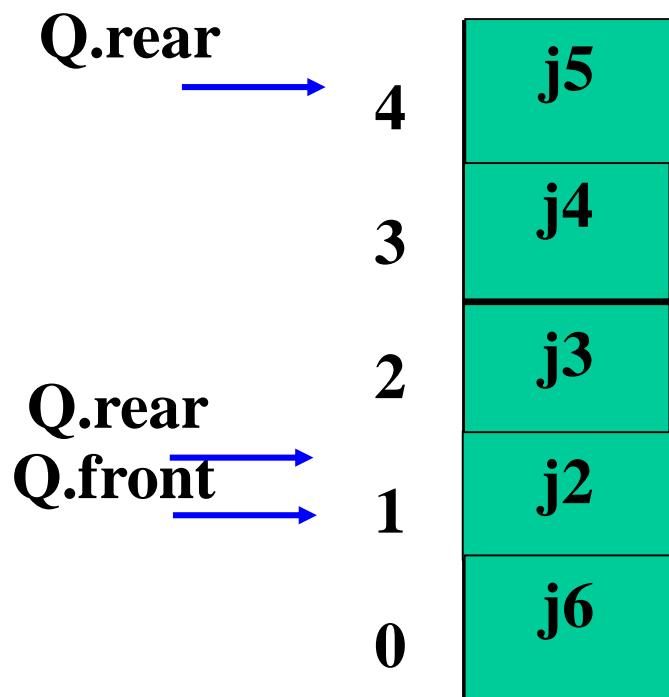
```
    i=0;
```

```
else
```

```
    i++;
```

利用模运算可简化为： $i = (i+1) \% \text{MAXSIZE}$



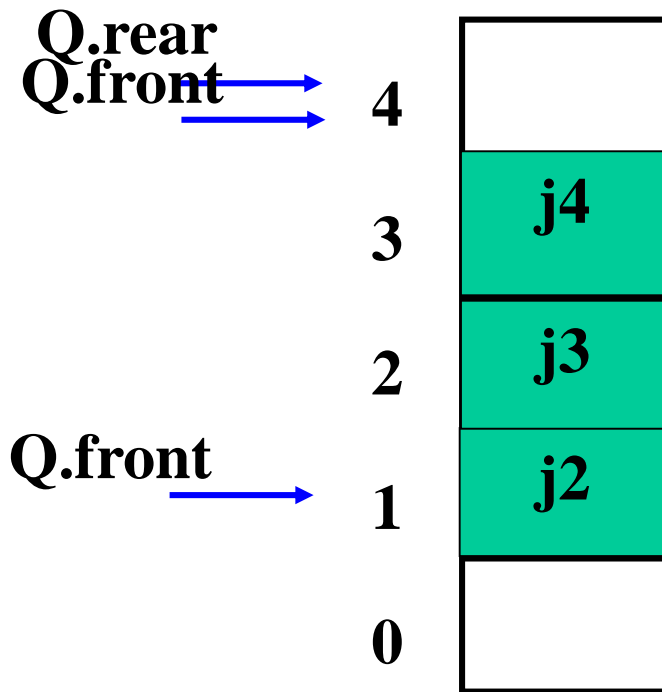


循环队列中头、尾指针和元素之间的关系

情况1:

(1). 初始情况, j2、j3、j4在队列中

(2). J5、J6入队后, **队满**: $Q.front == Q.rear$



情况2:

循环队列中头、尾指针和元素之间的关系

(1). 初始情况, j2、 j3、 j4在队列中

(2). J2、 J3、 J4出队后, 队空: $Q.front == Q.rear$

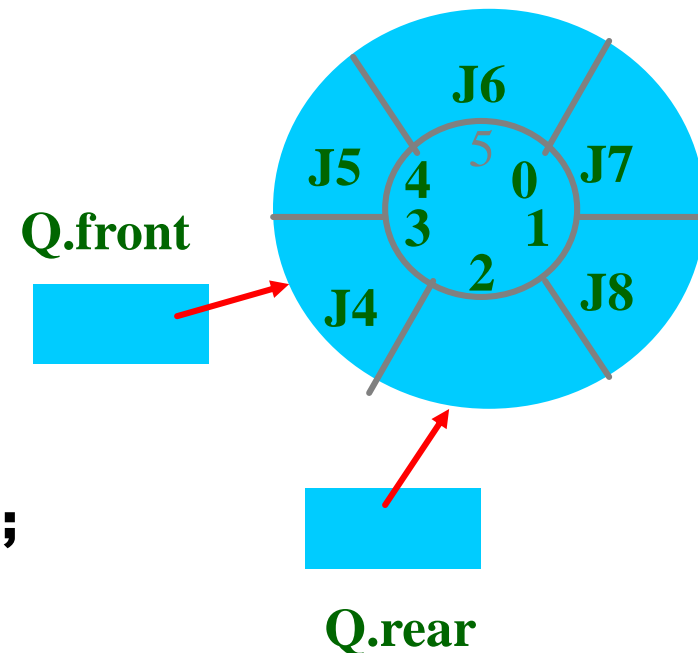
队空和队满时头尾指针均相等。因此，我们无法通过是否 $\text{front}=\text{rear}$ 来判断队列“空”还是“满”。

解决此问题的方法：

一、少用一个元素的空间。当尾指针在循环意义下加1后如等于头指针，则认为队满（注意：rear所指的单元始终为空）；

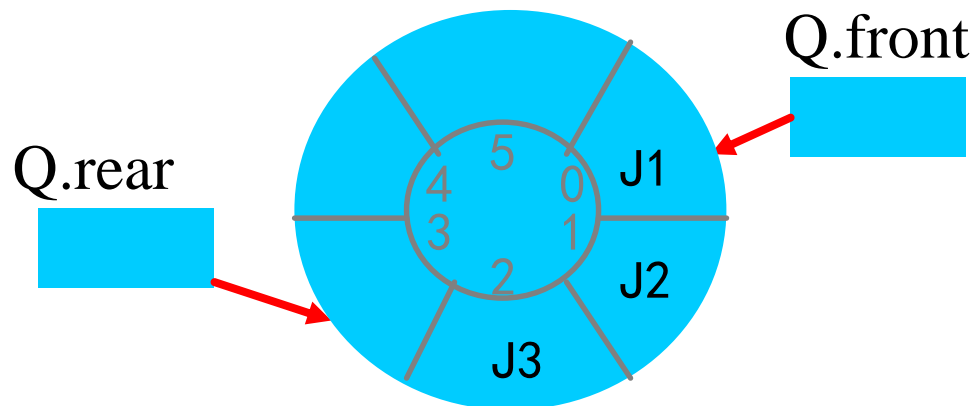
二、另设一个标志位以区分队空、队满；

三、使用一个计数器记录队列中元素的总数。



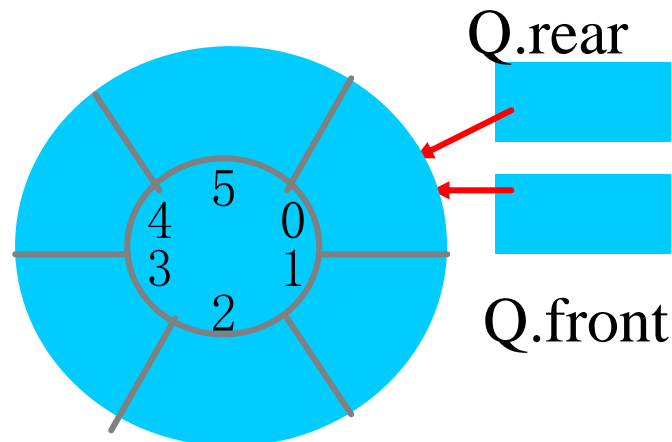
循环队列的表示：

```
# define MAXSIZE 100 //最大队列长度  
typedef struct {  
    QElemType *base; // 动态分配存储空间  
    int front; // 头指针，指示队列头元素的位置  
    int rear; // 尾指针，指示队列尾元素的下一个位置  
}SqQueue;
```



循环队列基本操作的实现

1. 初始化 InitQueue (&Q)



```
Status InitQueue_Sq(SqQueue &Q) {
```

```
    //构造一个空队列Q
```

```
    Q.base=(QElemType * )malloc (MAXQSIZE *sizeof (QElemType));
```

```
    //分配一个最大容量为MAXSIZE的数组空间
```

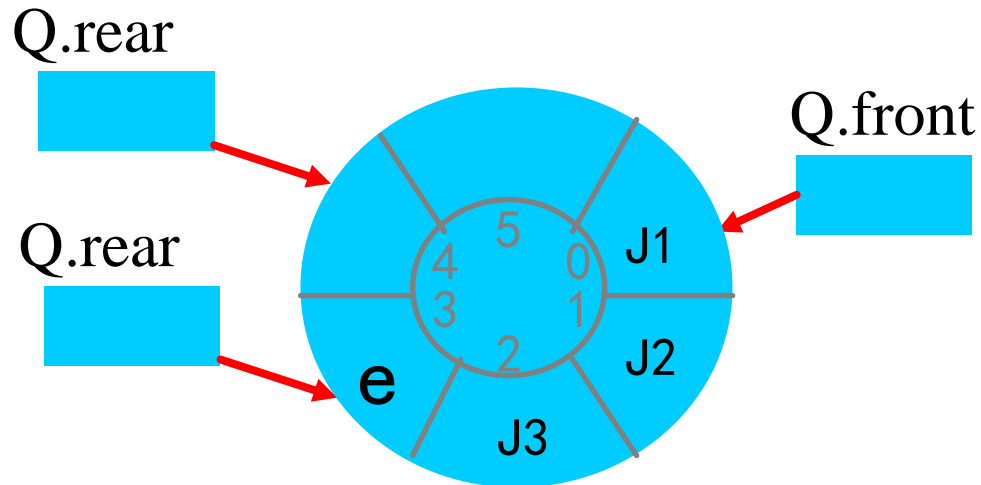
```
    if (!Q.base) exit (OVERFLOW); //存储分配失败
```

```
    Q.front = Q.rear=0; //队列为空，头指针和尾指针都为零
```

```
    return OK;
```

```
}
```

2. 入队 EnQueue (&Q, e)



Status EnQueue (SqQueue &Q, QElemType e)

if ((Q.rear+1)%MAXSIZE==Q.front) return ERROR ;

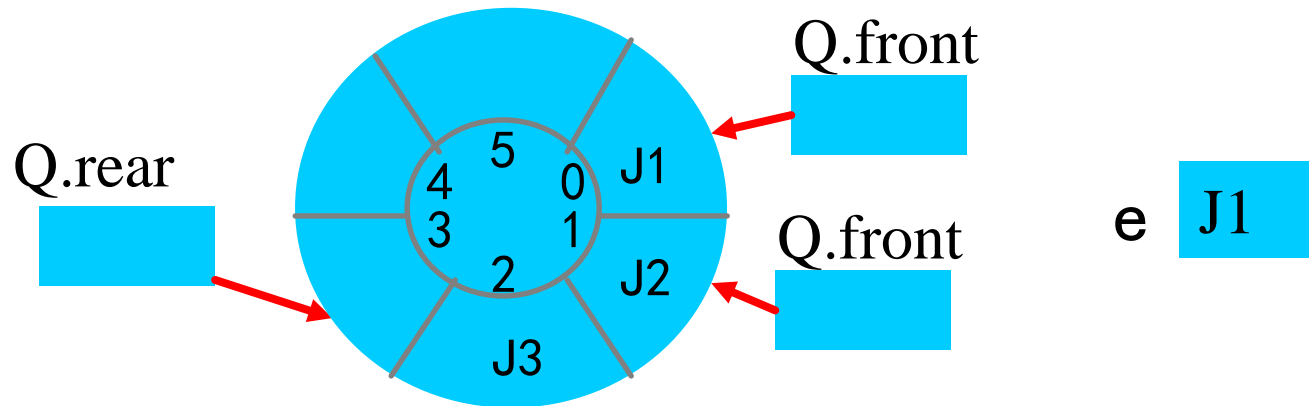
Q.base[Q.rear] = e ; // 将元素e插入队尾

Q.rear= (Q.rear+1)%MAXSIZE; // 修改队尾指针

return OK;

}

3. 出队 DeQueue(&Q, &e)



```
Status DeQueue (SqQueue &Q, QElemType &e )
```

```
//删除队头元素，用e返回其值
```

```
if ((Q.rear==Q.front) return ERROR ;
```

```
e =Q.base[Q.front] ;
```

```
Q.front= (Q.front+1)%MAXSIZE;      // 修改队头指针
```

```
return OK;
```

```
}
```

队列的应用

队列与栈一样，也是程序设计中经常使用的数据结构，凡是符合**先进先出**原则的数学模型，都可以用队列。例如：

在操作系统中用来

- 解决计算机主机与外设不匹配的问题
- 解决由于多用户引起的资源竞争问题

在实际应用中用来

- 模拟各种排队现象

广度优先搜索

判断题：

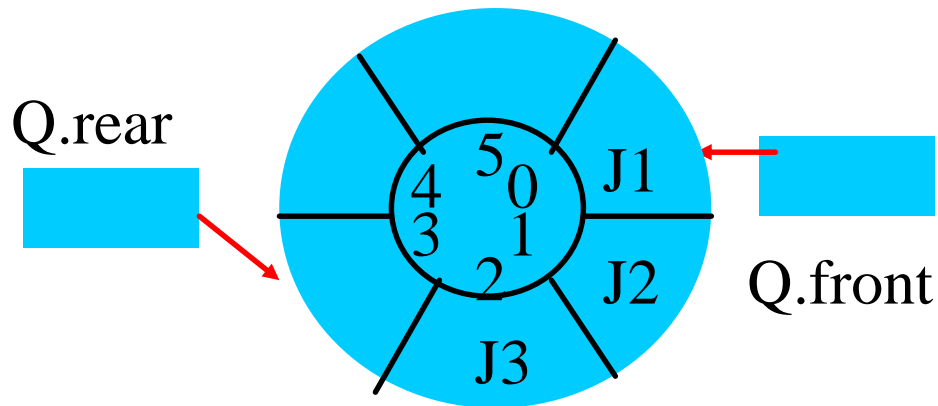
1.为解决计算机主机与打印机之间速度不匹配问题，通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是（ ）。

- A. 栈 B. 队列 C. 树 D. 图

第三章 习题二

1. P25 3. 29 3. 30

3. 29如果希望循环队列中的元素都能得到利用，则需设置一个标志域tag，并以tag的值为0或1来区分，尾指针和头指针值相同时的队列状态是“空”还是“满”。试编写与此结构相应的入队列和出队列的算法，并从时间和空间角度讨论设标志和不设标志这两种方法的使用范围(如当循环队列容量较小而队列中每个元素占的空间较多时，哪一种方法较好)。



3. 30假设将循环队列定义为：以域变量rear和length分别指示循环队列中队尾元素的位置和内含元素的个数。试给出此循环队列的队满条件，并写出相应的入队列和出队列的算法。

