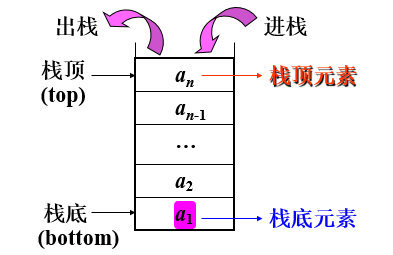
# 栈的定义及其ADT定义

## 栈（stack）：线性表

限定仅在表尾进行插入或删除操作。 后进先出 (LIFO结构)。



对于栈来说，表尾端称为栈顶，相应的，表头端称为栈底。不含元素的空表称为空栈。

## 栈的抽象数据类型的定义

**ADT Stack {**

数据对象：D＝{ ai | ai ∈ElemSet, i = 1, 2, ..., n, n≥0 }

数据关系：R1＝{ **<ai-1, ai >**| ai-1, ai∈D, i = 2, ..., n } 约定an 端为栈顶，a1 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。

**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。

**StackEmpty(S)**  
 初始条件：栈 S 已存在。

操作结果：若栈 S 为空栈，则返回 TRUE，否则 FALSE。

**StackLength(S)**   
 初始条件：栈 S 已存在。

操作结果：返回 S 的元素个数，即栈的长度。

**ClearStack(&S)**   
 初始条件：栈 S 已存在。

操作结果：将 S 清为空栈。

**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。

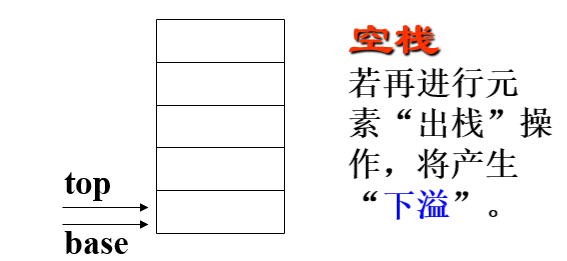
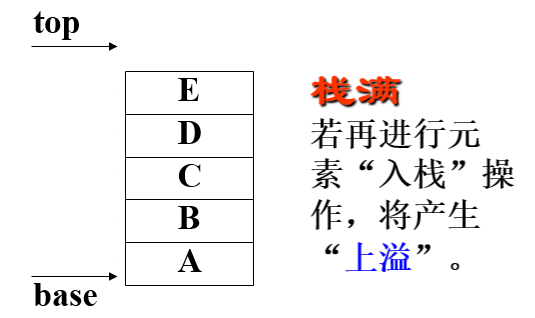
} **ADT Stack**

# 栈的表示和实现

## 顺序栈

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。

栈的两种状态：栈空、栈满

其中**base为栈底指针**，在顺序栈中，**它始终指向栈底的位置**，若base的值为NULL，则表明**栈结构不存在**。称top为栈顶指针，其初值指向栈底，**即top = base为栈空的标记**，每当插入新的栈顶元素的时候，指针top增加1，删除栈顶元素时，指针top减1.因此：top指示栈顶的位置，在非空顺序栈中栈顶指针指向栈顶元素的下一个位置。

### 顺序栈的定义

//顺序栈的定义

#define STACK\_INIT\_SIZE 100//栈存储空间的初识分配量

#define STACKINCREMENT 10//栈存储空间的分配增量

typedef struct

{

SElemType \*base;//栈底指针，它始终指向栈底位置

SElemType \*top;//栈顶指针

int stacksize;//当前分配的栈可使用的最大存储容量

}SqStack;

### 栈的基本操作在顺序栈中的实现

Status InitStack (SqStack &S) {

//构造一个空栈S

S.base = (SElemType \* )malloc(STACK\_INIT\_SIZE \* sizeof(SElemtype));

if (!S.base) exit (OVERFLOW);

S.top = S.base;

S.stacksize = STACK\_INIT\_SIZE;

return OK;

} // InitStack

Status GetTop (SqStack S, SElemType &e) {

//若栈不为空，则用e返回S的栈顶元素，并返回OK；否则返回error

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

e = \*(S.top – 1);

return OK;

} // GetTop

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) {

S.base = (SElemType \* )realloc(S.base,(S.stacksize + STACKINcrement) \* sizeof(SElemtype));

if (!S.base) exit (OVERFLOW);

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

S.stacksize += STACKINCREMENT; }

\* S.top ++ = e; //先赋值，之后再让指针往后指，实质就是\*(S.top) = e;S.top++;

return OK;

} // Push

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

//若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK，否则返回error

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

e = \* -- S.top; //先让指针往前指，之后再赋值，实质就是S.top--;e = \*(S.top);

return OK;

} // Pop

void DestroyStack(SqStack \*S)

{

//销毁栈

S->top = NULL;

S->stacksize = 0;

free(S->base);

}

Status ClearStack(SqStack &S)

{

//清空栈

S.top = S.base;

return OK;

}

Status StackEmpty(SqStack S)

{

//判断栈是否为空

if (S.top == S.base)

return ERROR;

else

return TRUE;

}

Status StackLength(SqStack S)

{

//求栈的长度

if (S.top == S.base)

return FALSE;

else

return (S.top - S.base);//也可以直接返回S.top - S.base

}

Status StackTraverse(SqStack S)

{

if (S.base == NULL)

return ERROR;

if (S.top == S.base)

printf("栈中没有元素……\n");

ElemType \*p;

p = S.top;

while (p > S.base)

{

p--;

printf("%d ",\*p);

}

return OK;

}

### 两种应用

应用一：如果将程序改为int top

**typedef** **struct**  //结构体定义

{

**int** \*elem;          //数据域

**int** top;    //顺序栈栈顶

**int** stacksize;     //顺序栈当前长度

}Stack;

**void** Gettop\_Stack(Stack &l, **int** e)

{

**if** (l.top == -1)

        Error("该栈为空栈！");

    e = l.elem[l.top];

    cout << "该顺序栈的栈顶元素是：";

    cout << e << endl;

}

**void** Increment\_Stack(Stack &l)

{

**int** \*newstack = **new** **int**[l.stacksize + STACK\_INCREMENT];

**if** (!newstack)

        Error("存储分配失败！");

**for** (**int** i = 0; i < l.top; i++)

    {

        newstack[i] = l.elem[i];

    }

    l.elem = newstack;

**delete**[] l.elem;

    l.stacksize += STACK\_INCREMENT;

}

**void** Push\_Stack(Stack &l, **int** e)

{

**if** (l.top == (l.stacksize - 1))

        Increment\_Stack(l);

    l.elem[++l.top] = e;

}

**void** Pop\_Stack(Stack &l, **int** e)

{

**if** (l.top == -1)

        Error("该顺序栈为空栈！");

    e = l.elem[l.top--];

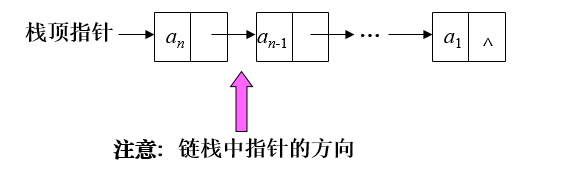
    cout << e << endl;

}

应用二：从后往前添加元素（需要改变base和top的指向）

## 链栈

### 链栈的定义



typedef struct SLNode

{

SElemType data;

struct SNode \*next;

}SLNode, SLinkList;

### 栈的基本操作在链栈中的实现

/\* 初始化栈\*/

SLinkList \* initStack()

{

SLinkList \* pNewNode = (SLinkList \*)malloc(sizeof(SLNode));

if (NULL != pNewNode)

{

pNewNode->data = 0;

pNewNode->next = NULL;

}

return pNewNode;

}

/\* 入栈操作 \*/

STACK\* pushStack(STACK\* pHead, DataType paraData)

{

STACK\* pNewNode = (STACK\*)malloc(sizeof(STACK));

if (NULL == pNewNode)

{

return NULL;

}

pNewNode->data = paraData;

pNewNode->pTop = pHead;

pHead = pNewNode;

return pHead;

}

/\* 出栈操作 \*/

STACK\* popStack(STACK\* pHead, DataType\* outData)

{

/\* 如果栈为空 \*/

if (NULL == pHead->pTop)

{

return NULL;

}

/\* 把数据传出去 \*/

\*outData = pHead->data;

STACK\* pTemp = pHead;

pHead = pHead->pTop;

/\* 把头删除 \*/

delete pTemp;

return pHead;

}

/\* 返回栈顶元素值 \*/

int getTopStack(STACK\* pHead, DataType\* outData)

{

/\* 如果栈为空 \*/

if (NULL == pHead->pTop)

{

return -1;

}

\*outData = pHead->data;

return 0;

}

/\* 遍历栈的操作 \*/

int visitStack(STACK\* pHead)

{

STACK\* pTemp = pHead;

/\* 判断栈是否为空 \*/

if (NULL == pHead->pTop)

{

printf("This stack is empty\n");

return -1;

}

while (NULL != pTemp->pTop)

{

printf("%d ", pTemp->data);

pTemp = pTemp->pTop;

}

printf("\n");

}

/\* 清空栈 \*/

STACK\* destoryStack(STACK\* pHead)

{

STACK\* pTemp = NULL;

while (NULL != pHead->pTop)

{

pTemp = pHead;

pHead = pHead->pTop;

delete pTemp;

}

return pHead;

}

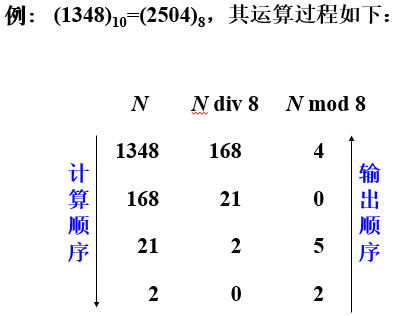
# 栈的应用、与递归的关系

## 数制转换

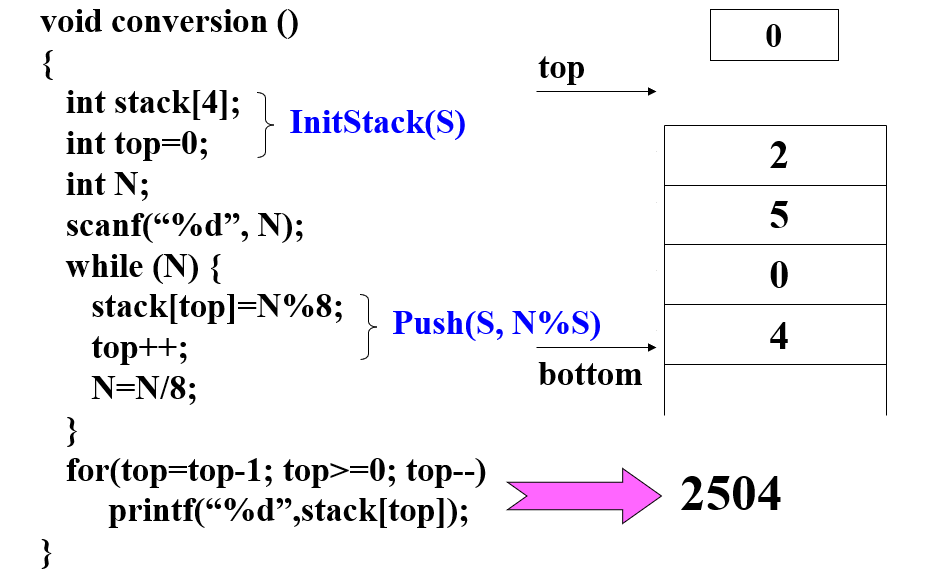
十进制数 *N* 和其他 *d*进制数 *M* 的转换是计算机实现 计算的基本问题，其解决方法很多，其中一个简单算法是 逐次除以基数 *d* 取余法，它基于下列原理：

***N* = (*N* div *d* )\**d* + *N* mod *d***

具体作法为：首先用*N*除以*d*，得到的余数是 *d* 进制数*M*的最低位*M*0， 接着以前一步得到的商作为被除数，再除以*d*，得到的余数是*d* 进制数*M*的次最低位*M*1，依次类推，直到商为0时得到的余数是*M*的最高位*Ms*（假定*M* 共有*s* +1 位）。



实现过程：



## 括号匹配的检验

假设表达式中允许括号嵌套，则检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。



过程如下：

1进栈，2进栈，3进栈，4是3所期待的那个人，因此3出栈，5进栈，6是5期待的那个人，因此5出栈，7是2期待的那个人，因此2出栈，8是1期待的那个人，因此1出栈

算法的设计思想：

1）**凡出现左括号，则进栈**；

2）凡出现右括号，首先**检查栈是否空**。**若栈空，则表明该“右括号”多余**；

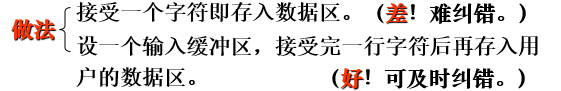
否则和栈顶元素比较，**若相匹配**，则“**左括号出栈**”，**否则表明不匹配**。

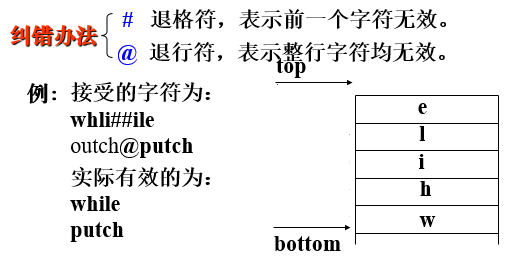
3）表达式检验结束时，**若栈空，则表明表达式中匹配正确**，否则表明“左括号”有多余的。

注意：**在算法开始和结束时，栈应该都是空的**。

## 行编辑程序

功能：接受用户从终端输入的数据并存入用户的数据区。





## 迷宫求解

求迷宫路径算法的基本思想：

（1）若当前位置“可通”，则纳入路径，继续前进；

（2）若当前位置“不可通”，则后退，换方向（按东南西北 的顺序）继续探索；

（3）若四周“均无通路”，则将当前位置从路径中删除出去

## 表达式求值（重要）

**先乘除，后加减；**

**运算规则**

**先括号内，后括号外；**

**从左算到右；**

例：求表达式 **4+2×3-10/5** 的值。

计算顺序为：4 + 2 \*3 = 4 + 6 – 10/5 = 10 – 10/5 = 10 – 2 = 8

为实现算符优先运算，可以使用两个工作栈，一个称作OPTR，用以寄存运算符；另一个称作OPND，用以寄存操作数或运算结果，算法的基本思想是：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 步骤 | OPTR栈 | OPND | 输入字符 | 主要操作 |
| 1 | **#** |  | **4+2×3-10/5#** |  |
| 2 | **#** | **4** | **+2×3-10/5#** | **Push 4** |
| 3 | **# +** | **4** | **2×3-10/5#** | **Push +** |
| 4 | **# +** | **4 2** | **×3-10/5#** | **Push 2** |
| 5 | **# + \*** | **4 2** | **3-10/5#** | **Push \*** |
| 6 | **# + \*** | **4 2 3** | **-10/5#** | **Push 3** |
| 7 | **# +** | **4 6** | **-10/5#** | **2\*3** |
| 8 | **#** | **10** | **-10/5#** | **4+6** |
| 9 | **# -** | **10** | **10/5#** | **Push -** |
| 10 | **# -** | **10 10** | **5#** | **Push 10** |
| 11 | **# - /** | **10 10** | **5#** | **Push /** |
| 12 | **# - /** | **10 10 5** | **#** | **Push 5** |
| 13 | **# -** | **10 2** | **#** | **10/5** |
| 14 | **#** | **8** | **#** | **10-2** |

## 地图四染色问题

同样利用了栈

## 栈与递归的实现

**递归：**一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称做递归函数。



当在一个函数的运行期间调用另一个函数时，在运行 该被调用函数之前，需先完成三件事：

1. 将实参等传递给被调用函数，保存返回地址（**入栈**）；
2. **为被调用函数的局部变量分配存储区**；
3. 将控制转移到被调用函数的入口。

从被调用函数返回调用函数之前，应该完成：

1. 保存被调函数的计算结果；
2. 释放被调函数的数据区；
3. 按被调函数保存的返回地址（**出栈**）将控制转移到调用函数。

**多个函数嵌套调用的规则是：后调用先返回。**

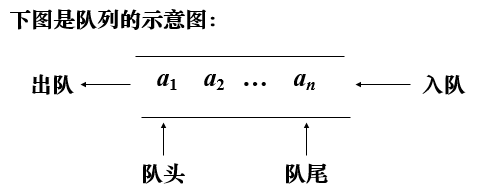
**此时的内存管理实行“栈式管理”**

# 队列的定义及其ADT定义

## 队列(queue) ：线性表

限定在表的一端插入（队尾）、另一端删除（对头）。

先进先出 (FIFO结构)。



当队列中没有元素时称为**空队列。**

## 队列的抽象数据类型的定义

**ADT Queue {**

**数据对象：**D＝{*ai*| *ai*∈ElemSet, *i* =1, 2, ..., *n*, *n*≥0}

**数据关系：**R1＝{ <*ai* -1,*ai* > | *ai* -1, *ai* ∈D, *i* =2, ..., *n*}

约定其中 *a*1 端为队列头，*an* 端为队列尾。

**基本操作：**

**InitQueue(&Q)**

操作结果：构造一个空队列 Q。

**DestroyQueue(&Q)**

初始条件：队列 Q 已存在。

操作结果：队列 Q 被销毁，不再存在。

**QueueEmpty(Q)**

初始条件：队列 Q 已存在。

操作结果：若 Q 为空队列，则返回 TRUE，

否则返回 FALSE。

**QueueLength(Q)**

初始条件：队列 Q 已存在。

操作结果：返回 Q 的元素个数，即队列的长度。

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

初始条件：Q 为非空队列。

操作结果：用 *e* 返回 Q 的队头元素。

**ClearQueue(&Q)**

初始条件：队列 Q 已存在。

操作结果：将 Q 清为空队列。

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

初始条件：队列 Q 已存在。

操作结果：插入元素 *e* 为 Q 的新的队尾元素。

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

初始条件：Q 为非空队列。

操作结果：删除 Q 的队头元素，并用 *e* 返回其值。

**} ADT Queue**

## 双端队列

**双端队列(double-ended queue**) **：线性表**

限定插入和删除在表的两端进行。（端点1、端点2）

先进先出 (FIFO结构)。



# 队列的表示和实现

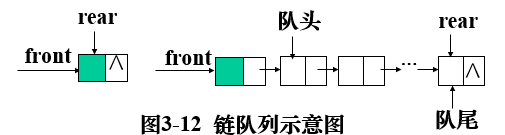
## 链队列——队列的链式表示和实现

链队列：用链表表示的队列。是限制仅在队**头删除和队尾插入**的单链表。

一个链队列由一个头指针和一个尾指针唯一确定。

（因为仅有头指针不便于在表尾做插入操作）。

为了操作的方便，也给链队列添加一个头结点，因此，空队列的判定条件是：**头指针和尾指针都指向头结点。**



typedef int QElemtype;

typedef struct QNode

{

QElemtype data;

struct QNode \*next;

} Qnode, \*QueuePtr; // 定义队列的结点

typedef struct

{

QueuePtr front; // 队头指针

QueuePtr rear; // 队尾指针

}LinkQueue;

## 队列的基本操作在链队列中的实现

**void** InitQueue(LinkQueue \*q) //初始化队列

{

    q->front = q->rear = (QuePtr)malloc(**sizeof**(QueNode)); //初始化队头与队尾的指针指向头结点

    q->front->next = NULL;

}

**void** EnQueue(LinkQueue \*q,ElemType e) //将元素e进队

{

    QuePtr temp = (QuePtr)malloc(**sizeof**(QueNode)); //创建新结点

**if**(temp) //如果内存分配成功

    {

        temp->data = e; //初始化新结点的数据为e

        temp->next = NULL; //队列只能从队尾插入所以下一个结点初始化为NULL

        q->rear->next = temp; //将队尾结点的指针指向新结点,如果新结点为第一个结点则q->rear->next相当于q->front->next

        q->rear = temp; //将指向队尾的指针指向新结点

    }

}

**void** DeQueue(LinkQueue \*q,ElemType \*e) //队头的结点出队,将出队的结点的元素存入\*e

{

**if**(q->front == q->rear) //队列为空

**return**;

    QuePtr temp = q->front->next; //初始化temp为要出队的结点的指针

**if**(q->front->next == q->rear) //如果要出队的结点为最后一个结点,使q->rear指向头结点防止出现悬空的指针

        q->rear = q->front;

    \*e = temp->data; //将出队的数据元素存入\*e

    q->front->next = temp->next; //使下一个结点成为队头,如果没有下一个结点则为NULL

    free(temp); //删除要出队的结点

}

**bool** IsEmpty(LinkQueue \*q) //判断队列是否为空

{

**if**(q->front == q->rear)

**return** **true**;

**return** **false**;

}

**int** GetQueueLength(LinkQueue \*q) //返回队列的长度

{

    QuePtr temp = q->front;

**int** i = 0;

**while**(temp != q->rear)

    {

        ++i;

        temp = temp->next;

    }

**return** i;

}

**void** Clear(LinkQueue \*q) //清空队列

{

    QuePtr temp = q->front->next;

**while**(temp)

    {

        QuePtr tp = temp;

        temp = temp->next;

        free(tp);

    }

    temp = q->front;

    q->front = q->rear = NULL;

    free(temp);

}

**void** Print(LinkQueue \*q) //打印队列的元素

{

**if**(q->front == q->rear)

**return**;

    QuePtr temp = q->front->next;

**while**(temp != q->rear)

    {

        printf("%d ",temp->data);

        temp = temp->next;

    }

    printf("%d",temp->data);

    printf("\n");

}

**void** TopQueue(LinkQueue \*q,ElemType \*e) //返回队头的结点元素存入\*e

{

**if**(q->front == q->rear)

**return**;

    \*e = q->front->next->data;

}

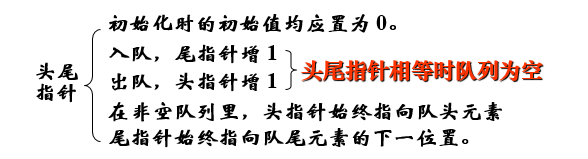
# 循环队列－队列的顺序表示和实现

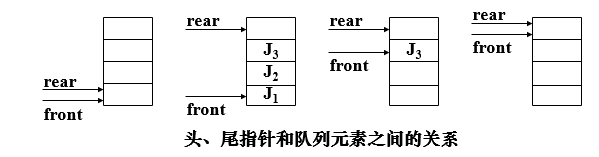
## 定义

是限制仅在表头删除和表尾插入的顺序表。

利用一组地址连续的存储单元依次存放队列中的数据元素。

因为：队头和队尾的位置是变化的，所以：设头、尾指针。

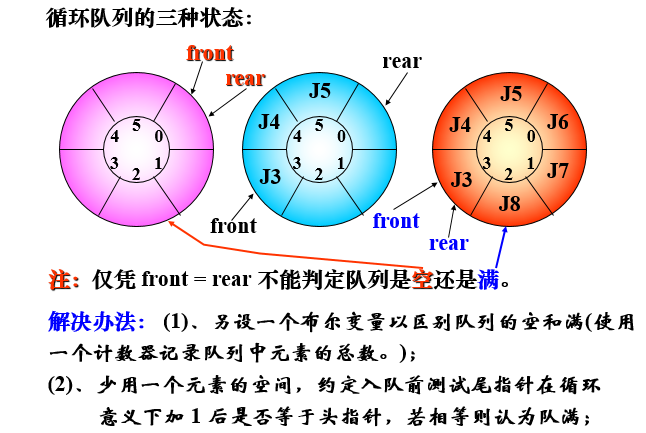




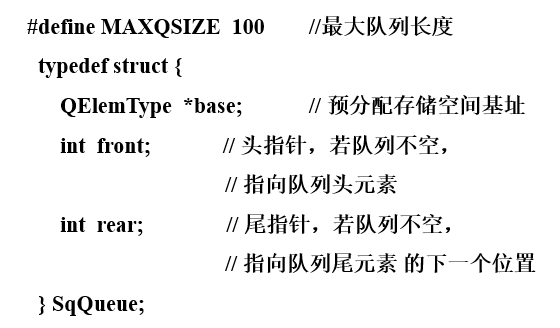
在顺序队列中，当尾指针已经指向了队列的最后一个位置的下一位置时，若再有元素入队，就会发生“溢出”

**“假溢出”——队列的存储空间未满，却发生了溢出。**





## 循环队列的顺序存储结构



**队列的初始化：**

**Status InitQueue (SqQueue &Q)**

**{ // 构造一个空队列Q**

**Q.base = (QElemType \*) malloc**

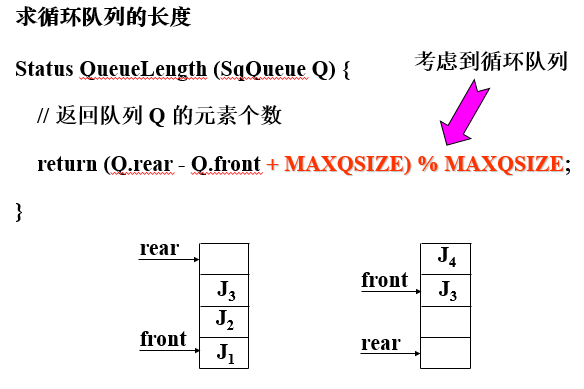
**(MAXQSIZE \*sizeof (QElemType));**

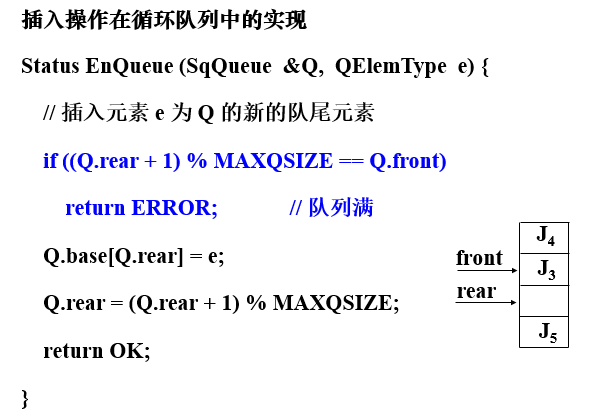
**if (!Q.base) exit (OVERFLOW); // 存储分配失败**

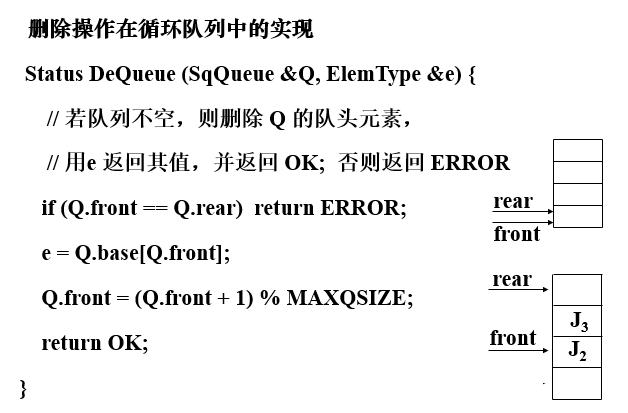
**Q.front = Q.rear = 0;**

**return OK;**

**}**







# 实验拓展：

1、 设有两个栈S1,S2都采用顺序栈方式，并且共享一个存储区[O..maxsize-1],为了尽量利用空间，减少溢出的可能，可采用栈顶相向，迎面增长的存储方式。试设计S1,S2有关入栈和出栈的操作算法。

#include<stdio.h>

#include<stdlib.h>

#define maxsize 100

typedef int SElemType;

typedef struct

{

SElemType \*base;

int top[2];

}SqStack;

//初始化

bool Init(SqStack \*L)

{

L->base = (SElemType \*)malloc(maxsize\*sizeof(SElemType));

if(!L->base)

return false;

L->top[0] = 0;

L->top[1] = maxsize-1;

return true;

}

//入栈

bool push(SqStack \*L,int i,SElemType e)

{

if(L->top[0] == L->top[1]-1)

return false;

if(i == 0)

{

L->base[L->top[0]] = e;

L->top[0]++;

}

if(i == 1)

{

L->base[L->top[1]] = e;

L->top[1]--;

}

return true;

}

//出栈

bool pop(SqStack \*L,int i,SElemType \*e)

{

if(L->top[0] == 0 && i == 0)

return false;

if(L->top[1] == maxsize-1 && i == 1)

return false;

if(i == 0)

{

L->top[0]--;

\*e = L->base[L->top[0]];

}

if(i == 1)

{

L->top[1]++;

\*e = L->base[L->top[1]];

}

return true;

}

void visit(SqStack \*L,int i)

{

int j = 0;

if(i ==1)

{

int a =maxsize -1;

for(j = a;j > L->top[1];j--)

printf("%d\n",L->base[j]);

}

if(i ==0)

{

for(j = 0;j<(L->top[0]);j++)

printf("%d\n",L->base[j]);

}

}

int main()

{

SqStack s;

SElemType a;

Init(&s);

push(&s,1,1);

push(&s,1,2);

pop(&s,1,&a);

visit(&s,1);

system("pause");

return 0;

}

2、编程：假设以数组Q[m]存放循环队列中的元素，同时以rear和length分别指示队列中的队尾位置和队列中所含元素的个数，试给出该循环队列的队空条件和队满条件，并写出相应的初始化、插入、删除元素的操、

#include<stdio.h>

#include<stdlib.h>

#define maxsize 10

typedef int QElemType;

typedef struct

{

QElemType \*base;

int rear;

int length;

}SqQueue;

//初识化

bool Init(SqQueue \*L)

{

L->base = (QElemType \*)malloc(maxsize\*sizeof(QElemType));

if(!L->base)

return false;

L->rear = -1;

L->length = 0;

return true;

}

//入队

bool EnQueue(SqQueue \*L,QElemType e)

{

if(L->length == maxsize)

return false;

L->rear = (L->rear+1)%maxsize;

L->base[L->rear] = e;

L->length++;

return true;

}

//出队

bool DeQueue(SqQueue \*L,QElemType \*e)

{

if(L->length == 0)

return false;

\*e = L->base[(L->rear-L->length+maxsize+1)%maxsize];

L->length --;

return true;

}

int main()

{

SqQueue q;

QElemType x;

Init(&q);

EnQueue(&q,666);

DeQueue(&q,&x);

printf("%d",x);

system("pause");

return 0;

}