数据结构——栈和队列

第三章　栈和队列

重点：栈和队列的基本特征，表示与实现

难点：递归、循环队列

第三章 栈和队列

3.1 栈

- 定义

  特殊的线性表：**操作受限**

  是限定仅在表尾进行插入或删除操作的线性表

  允许插入或删除的一端称为栈顶(top),另一端称为栈底(bottom)

- 逻辑特征

  后进先出(LIFO)

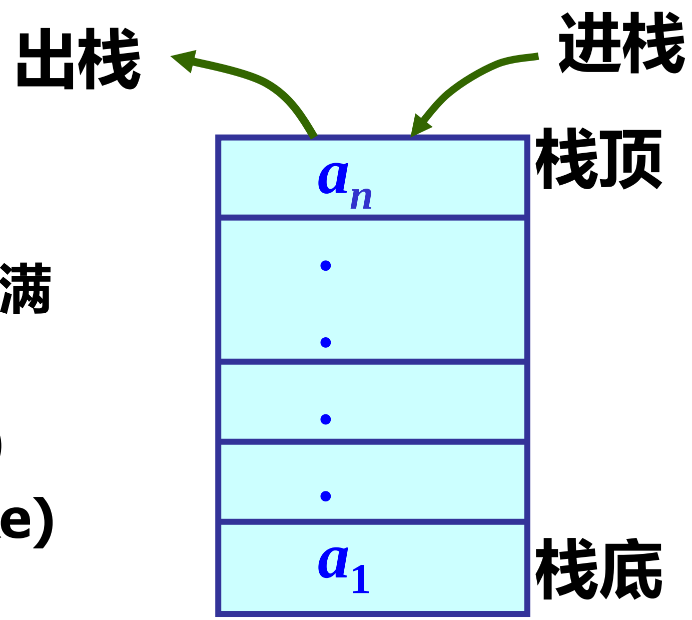
- **ADT Stack**

  初始化空栈、判断栈空、判断栈满

  取栈顶 vs. GetElem(L, i, &e)

  入栈 vs. ListInsert(&L, i, e)

  出栈 vs. ListDelete(&L, i, &e)



ADT Stack{

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

数据关系：$R=\{R1\}, R1=\{<a_{i-1},a_i> \mid a_{i-1},a_i \in D, i=2,3,\cdots,n\}$

基本操作：

InitStack(&S)

操作结果：构造一个空的栈 S

DestroyStack(&S)

初始条件：栈 S 已存在

操作结果：销毁栈 S

ClearStack(&S)

初始条件：栈 S 已存在

操作结果：将栈 S 重置为空栈

StackEmpty(S)

初始条件：栈 S 已存在

操作结果：若 S 为空栈，则返回 TRUE，否则返回 FALSE

StackLength(S)

初始条件：栈 S 已存在

操作结果：返回栈 S 中数据元素的个数

GetTop(S, &e)

GetElem(L, i, &e)

初始条件：栈 S 已存在且非空

操作结果：用 e 返回 S 中栈顶元素

Push(&S, e)

ListInsert(&L, i, 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)

ListDelete(&L, i, &e)

初始条件：栈 S 已存在且非空

操作结果：删除 S 的栈顶元素，并用 e 返回其值

StackTraverse(S, visit( ))

初始条件：栈 S 已存在且非空

操作结果：从栈底到栈顶依次对 S 的每个数据元素调用函数 visit( )。一旦 visit( )失败，则操作失败

}ADT Stack

3.1 栈–顺序栈-类型定义

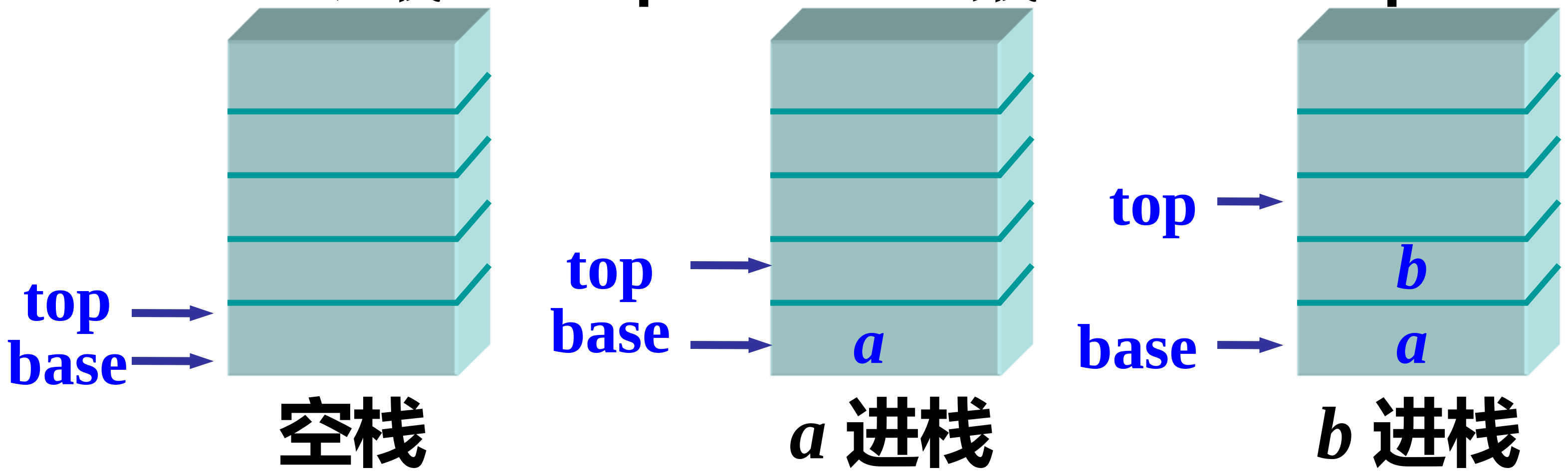
- <u>顺序栈</u>
  - 类型定义

    注意**top**的含义——约定

```
#define  STACK_INIT_SIZE  100// 存储空间的初始分配量
#define  STACKINCREMENT  10// 存储空间的分配增量
typedef  struct{
   ElemType      *base;           // 栈底指针
   ElemType      *top;    // 栈顶指针(栈顶元素的下一个位置)
   int           stacksize;        //  当前分配的存储容量
}SqStack;
```

3.1 栈—顺序栈-基本操作的实现

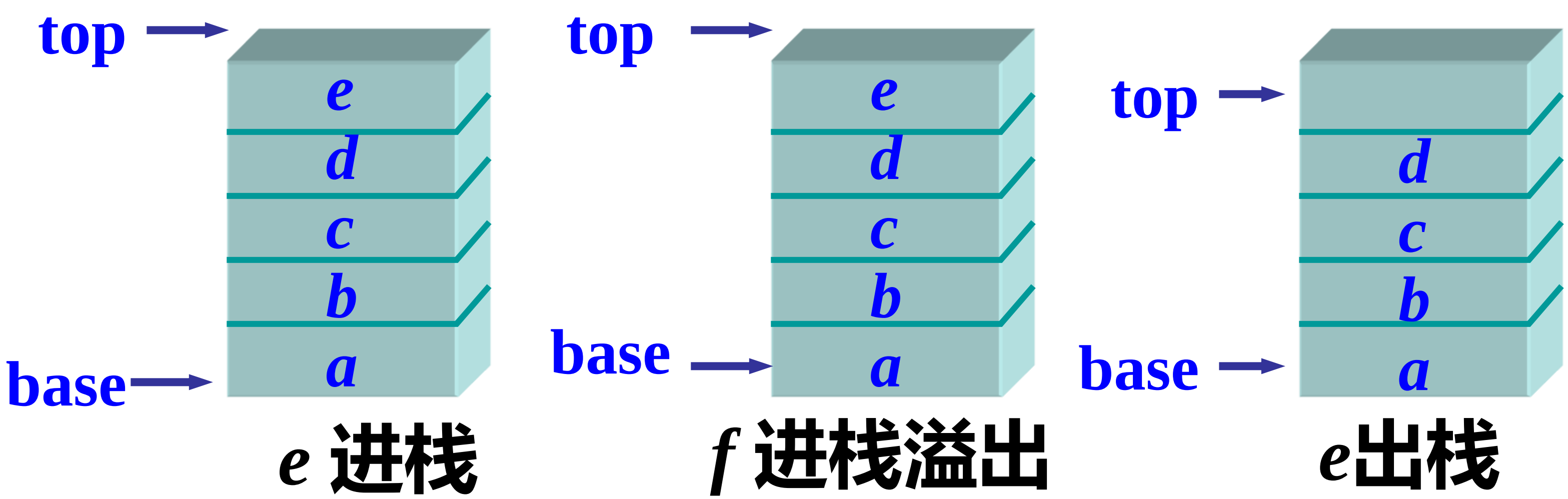
- 顺序栈
  - 基本操作的实现
    - 栈顶的初始化：S.top = S.base
    - 栈空：S.base == S.top
    - 栈满：S.top - S.base >= S.stacksize
    - 入栈：*S.top ++ = e，出栈：e = *--S.top



约定：top指向栈顶元素的下一个位置

3.1 栈–顺序栈-基本操作的实现

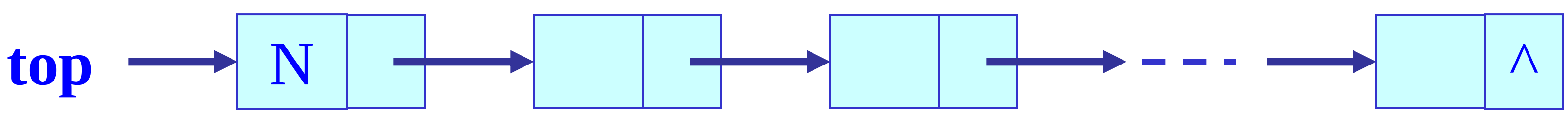
- 顺序栈



约定：top指向栈顶元素的下一个位置

3.1 栈–链栈的表示与实现

- <u>链栈</u>
  - **无栈满问题(除非分配不出内存), 空间可扩充**
  - **栈顶----链表的表头**
  - **可以不必引入头结点**



约定：top指向栈顶元素所在的结点

3.2 栈的应用举例

- 数制转换
- 括号匹配的检验
- 行编辑程序
- 迷宫求解
- 表达式求值

3.2 栈的应用举例-数制转换

- 数制转换: 将十进制数N转换成其他d进制数
  - 算法思想：N = (N div d)×d + N mod d

    1) 保存余数N%d

    2) N=N/d,

    3) 若N==0结束，否则继续1)。

    保存的余数从先到后依次表示转换后的d 进制数的低位到高位，而输出是由高位到低位的，因此必须定义先进后出的线性表——栈来保存；当全部的余数求出后，通过逐个出栈输出d进制数。

3.2 栈的应用举例-数制转换

- 数制转换
  - **算法**(算法3.1 P48)

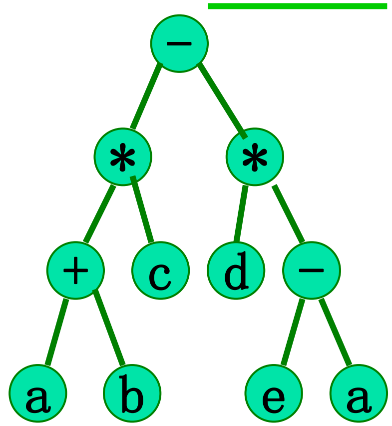
```
void conversion(int N, int d){
  InitStack(S);
  while(N) { Push(S, N%d); N=N/d; }
  while(!StackEmpty(S)) {
    Pop(S, e);
    printf(e);
  }
}
```

3.2 栈的应用举例-行编辑程序

- 行编辑程序(P49)
  - **处理规则**：遇‘#’退一格；遇‘@’退一行
  - **算法思想**：引入栈，保存终端输入的一行字符（逐行处理）；
    遇‘#’退一格——出栈一次
    遇‘@’退一行——清栈
  - **步骤**：
    1）初始化栈S
    2）读入字符ch
    3）ch!=EOF
        3.1) ch!=EOF && ch!=’\n’
            3.1.1）ch为‘#’：Pop(S, c), 转3.1.4)
            3.1.2）ch为‘@’： ClearStack (S) , 转3.1.4)
            3.1.3）ch为其他： Push (S, ch) , 转3.1.4)
            3.1.4）再读入字符ch，继续3.1)
        3.2) 处理完一行，清空栈
        3.3) 如ch!=EOF，读入字符ch，继续3)

3.2 栈的应用举例-表达式求值

- 表达式求值(P52)
  - **表达式的表示**
    - 中缀表达式 **(a+b)*c-d*(e-a)**
    - 前缀表达式 **-*+abc*d-ea** (波兰式)
    - 后缀表达式 **ab+c*dea-*-** (逆波兰式)



分支结点是算符

叶子结点是操作数

3.2 栈的应用举例-表达式求值

- **问题描述**
  - 只包含**+, -, *, /** 四个双目运算符，且算符本身不具有二义性；
  - 三个算术四则运算的规则
    - 先乘除,后加减
    - 从左算到右
    - 先括号内,后括号外

    ➔算符间的优先关系（算符的优先级和结合性）**(表3.1)**
  - **#:** 表达式的开始符和结束符
  - 只有 **'('==')'**，**'#'=='#'**；
  - 假设输入的是一个合法的表达式。

3.2 栈的应用举例-表达式求值

- **算法思想**

  输入： 中缀表达式串　　　**输出：** 表达式值

  - 引入**OPTR**和**OPND**两个栈
  - 初始**OPTR**有一元素‘**#**’，**OPND**为空
  - 读入一字符**c**

    c==‘#’：return(GetTop(OPND))

    c是非运算符：**Push(OPND,c)**

    c是运算符：**t=GetTop(OPTR)**，比较**t**和**c**的优先关系

t<c:	**Push(OPTR,c)**
t==c:	**Pop(OPTR, x)**
t>c:	**Pop(OPTR, theta); Pop(OPND, b);**
	Pop(OPND, a);
	x=Operate(a, theta, b);
	Push(OPND, x);

  - 继续读入字符处理。

3.2 栈的应用举例-表达式求值

- **输入**：前缀表达式串(波兰式)
  输出：表达式值
  例： **-*+abc*d-ea**
  - **遇到算符时，由于运算对象还未读到，故暂存**
  - **遇到运算对象时，需要判断当前最近读入的算符的运算对象是否齐全，若齐全则计算否则暂存**
  - **暂存的算符个数不固定，需要引入数据结构保存**
    - 针对上例：在进行第一个运算前，需暂存**-,***
    - 数据结构的操作特点：后进先出**-→**栈
  - **暂存的运算对象个数不固定，需要引入数据结构保存**
    - 针对上例：在进行第一个**+**运算前，需暂存**a,b,c**
    - 数据结构的操作特点：后进先出**-→**栈
  - **不需要比较算符的优先级和结合性，只要算符的运算对象已经齐全，即可计算！**

3.2 栈的应用举例-表达式求值

```
OpndType EvaluatePostExpr(){ //假设表达式是合法的
   InitStack(OPND);  c=getchar();
   while (c!='#') {
     if ( !IsOP(c) )  Push(OPND, c) ;
     else{
       Pop(OPND, b); Pop(OPND, a);
       Push(OPND, doOp(a,c,b));
     }
     c=getchar();
   }
   result = GetTop(OPND);    DestroyStack(OPND);
   return result;
}
```

3.2 栈的应用举例-表达式求值

- **输入：** 后缀表达式串(逆波兰式)
  输出： 表达式值
- 例： **ab+c*dea-*-**
  - **遇到运算对象时，由于算符在后，故暂存该运算对象**
  - **遇到算符时，立即计算**
  - **暂存的运算对象个数不固定，需要引入数据结构保存**
    - 上例中：在进行第一个运算前，需暂存**a,b**
    - 数据结构的操作特点：后进先出**-→**操作数栈
  - **不需要比较算符的优先级和结合性**
  - **遇到算符时，直接从操作数栈中弹出所需的操作数进行计算；计算后再将计算结果入栈**

3.2 栈的应用举例-表达式转换

- 表达式的不同表示之间的相互转换
  - 逆波兰式→波兰式
    - 如： **ab+c*dea-*- → -*+abc*d-ea**
    - 共同点
      - 运算对象的次序一致
      - 不需要括号区分优先级和结合性
      - 某算符在所有算符中的次序决定了它的计算次序
    - 转换的思想
      - 参照逆波兰式的计算过程，将计算转换为待计算的算符和运算对象的串的拼接
  - 逆波兰式→中缀表达式
    - 如： **ab+c*dea-*- → (a+b)*c-d*(e-a)**

栈的概念运用

- **问题1：习题集3.6 P23**
  - 输入序列：1 2 3 … n
  - 输出序列：$p_1\ p_2\ p_3\ \dots\ p_n$
  - 证明：不存在$i<j<k$, 使得$p_j< p_k < p_i$
  - 证(反证法)：假设存在$i<j<k$, 使$p_j< p_k < p_i$

    ∵ $i<j<k$ ∴ p_i最先出栈，p_k最后出栈

    由假设知p_i是三个数中最大的

    ∵入栈序列是按值递增的

    ∴若某个值大的先出栈，栈中还有比它小的值，这些值将按值递减的次序依次输出

    即p_i先输出，p_j和p_k都比p_i小，p_k最后输出，p_k的值应该是最小的

    这与假设相矛盾。

3.3 栈与递归的实现

- 递归的定义
  - **递归(recursion)**：直接或间接地调用自身.
    - 递归的规则
    - 递归终止条件

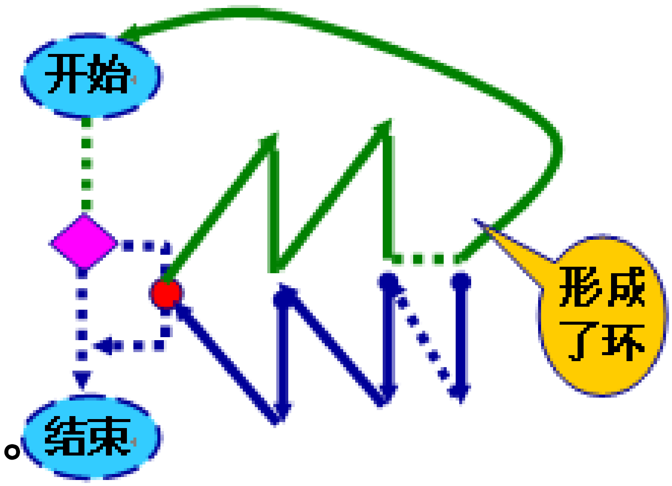
    如: **n! = (n-1)! *n**

    0! = 1
  - **递推**：从已知的初始条件出发，逐次递推出最后要求的值。

    如：**0!=1, 1!= 0!*1 = 1**

    2! = 1! *2 = 2

    ……



3.3 栈与递归的实现

<u>递归</u>

```
int  fact1(int n)
{
    if (n<0) return -1;
    else if (n==0) return 1;
    return n*fact1(n-1);
}
```

<u>递推</u>

```
int fact2(int n)
{
    if (n<0) return -1;
    fact=1; i=1;
    while (i<=n)
    {
        fact *= i; i++;
    }
}
```

3.3 栈与递归的实现

- **递归与递推的关系**

  递归算法的执行过程分递推与回归两个阶段。

  - 在递推阶段，把较复杂的问题（规模为n）的求解推到比原问题简单一些的问题（规模小于n）的求解。
  - 在回归阶段，当获得最简单的情况后，逐级返回，依次获得稍复杂问题的解。

  递推是递归的一个阶段，递归包含着递推。

3.3 栈与递归的实现

- **递归的对象：**一个对象部分地包含它自己，或用它自己给自己定义。**如某些数据结构的定义**

  线性表的另一种定义(归纳定义):

  - **基本步：若元素个数为0，则称为空表**
  - **归纳步：若元素个数大于0，则有且仅有一个第一个元素(表头)，剩余元素形成一个表(表尾)。**

- **递归的过程：**一个过程直接或间接地调用自己

  如：0的阶乘是1，n(n>0)的阶乘等于n乘上(n-1)的阶乘

3.3 栈与递归的实现

- 递归的应用
  - 递归定义：**如阶乘、Fibonacci数列等**
  - 数据结构：**如表、树、二叉树等**
  - 问题求解：**如Hanoi塔**

3.3 栈与递归的实现–阶乘函数

- 定义是递归的

$$n! = \begin{cases} 1, & 当 n = 0 时 \\ n * (n-1)!, & 当 n \geq 1 时 \end{cases}$$

求解阶乘函数的递归算法

```
long fact ( long n ) {
    if ( n == 0 ) return 1;          //递归结束条件
    else return n * fact (n-1);     //递归的规则
}
```

3.3 栈与递归的实现–阶乘函数

递归调用

参数传递

主程序 main( ): fact(4)

24

fact(4): 计算 4*fact(3)

6

fact(3): 计算 3*fact(2)

2

fact(2): 计算 2*fact(1)

1

fact(1): 计算 1*fact(0)

1

fact(0): 直接定值为 1

结果返回

回归求值

3.3 栈与递归的实现–数据结构

- 数据结构是递归的--表
  - 空表
  - 非空表: (表头元素+除表头元素以外的剩余元素)
  - **查找表L中是否有元素值e**

```
LinkList search(LinkList L, ElemType e)
// L为不带头结点的单向非循环链表
{
   if（L==NULL）return NULL;
   else if（L->data==e）return L;
   else return search(L->next, e);
}
```

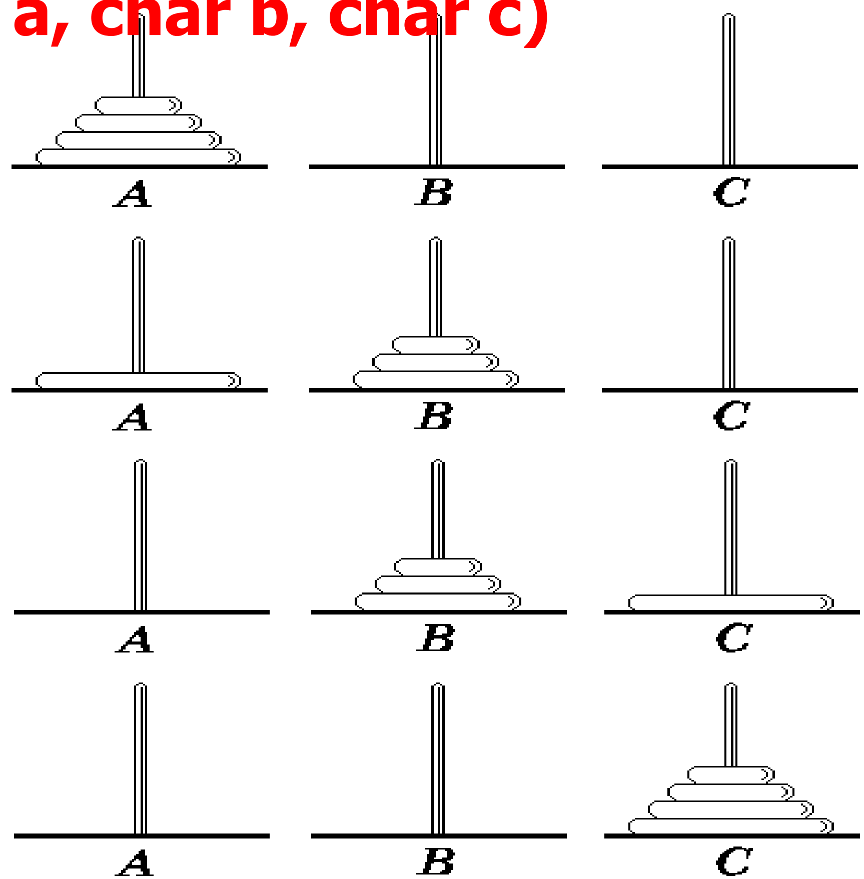
3.3 栈与递归的实现–Hanoi塔

- 问题求解是递归的—Hanoi塔

  void hanoi(int n, char a, char b, char c)

  n-圆盘数　**a-**源塔座

  b-中介塔座 **c-**目标塔座



- 搬动方法
  - **n=1，a->c**
  - **n>1:**

    hanoi(n-1, a, c, b)

    a->c

    hanoi(n-1, b, a, c)
- 注意

  用递归调用用的结果，不关注该结果如何获得的细节

3.3 栈与递归的实现–递归实现

- 调用函数与被调用函数---系统工作栈
  - 执行被调用函数前
    - 现场保护：**实在参数、返回地址**
    - 为被调用函数的局部变量分配存储区
    - 将控制转移到被调函数的入口
  - 从被调用函数返回调用函数前
    - 保存被调函数的计算结果
    - 释放被调函数的数据区
    - 现场恢复：**返回**

3.3 栈与递归的实现–递归实现

- 递归过程与递归工作栈
  实际的系统中，一般统一处理递归调用和非递归调用
- 递归工作栈
  - 活动记录(栈帧**stack frame)**
    实在参数、局部变量、上一层的返回地址
  - 每进入一层递归，产生一个新的工作记录入栈
  - 每退出一层递归，就从栈顶弹出一个工作记录
  - 当前执行层的工作记录必是栈顶记录
- 例子：**P57 hanoi(3,a,b,c)**

3.3 栈与递归的实现–递归/回溯

- 递归与回溯—N皇后问题
  - **回溯**是按照某种条件往前试探搜索,若前进中遭到失败,则回过头来另择通路继续搜索.
  - N皇后问题

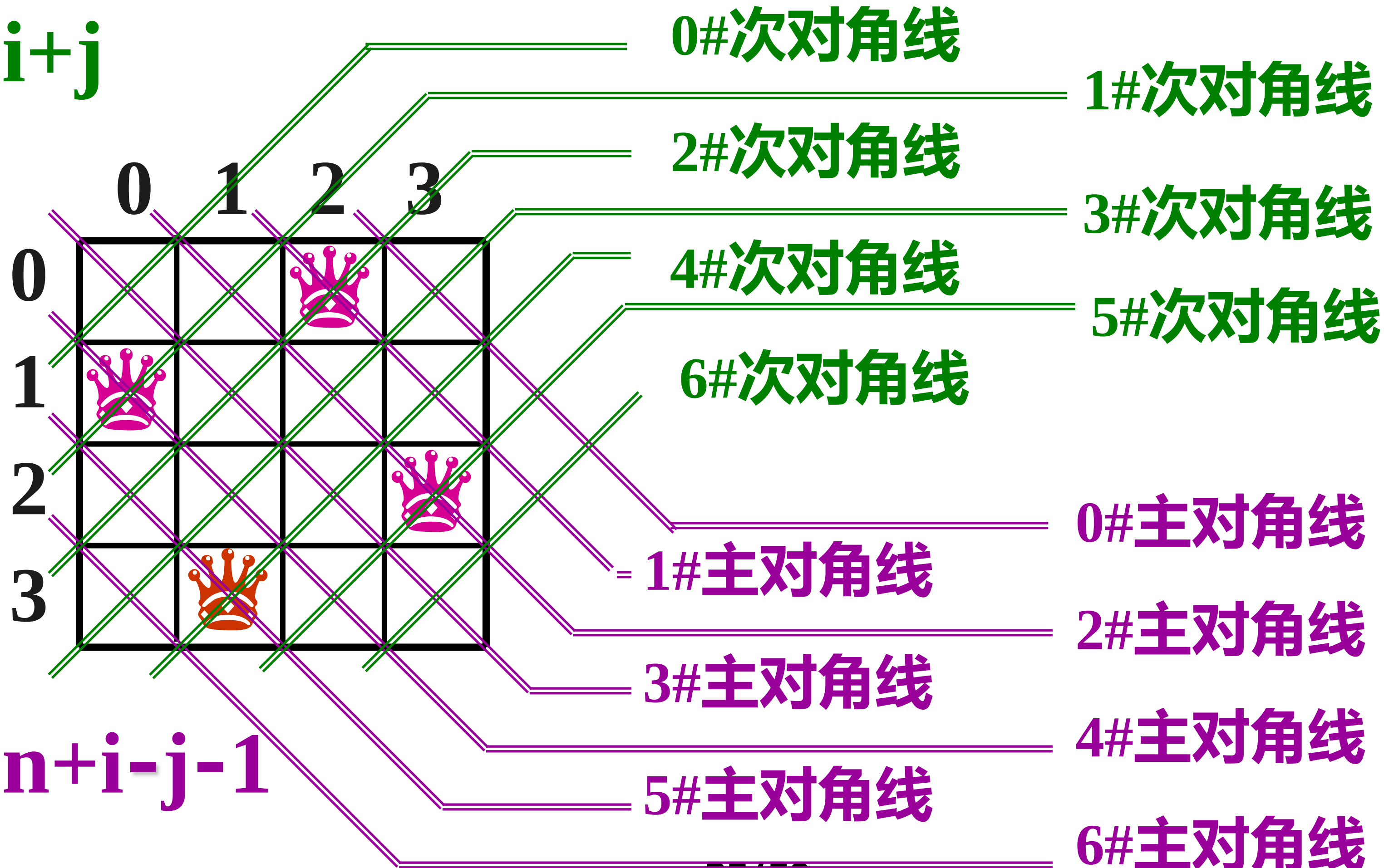
    在n行n列的国际象棋棋盘上，若两个皇后位于同一行、同一列或同一对角线上，则称为它们为互相攻击。

    n皇后问题是指:

    找到这 n 个皇后的互不攻击的布局

3.3 栈与递归的实现–N皇后问题

k = i+j



k = n+i-j-1

3.3 栈与递归的实现–N皇后问题

- 基本思路

  依次为每一行确定该行皇后的合法位置

  - **安放第 i 行皇后时，需要在列的方向从 0 到 n-1 试探 (j = 0, …, n-1)**
  - **在第 j 列安放一个皇后**
    - **如果在列、主对角线、次对角线方向有其它皇后，则出现攻击，撤消在第 j 列安放的皇后;**
    - **如果没有出现攻击，在第 j 列安放的皇后不动递归安放第 i+1行皇后**
  - **如果第 i 行不能安放皇后，则回溯到第 i-1 行，从下一个列(j+1)继续试探**

3.3 栈与递归的实现–N皇后问题

- 数据结构设计
  - **标识每一列是否已经安放了皇后**
    ——线性表，表长为**N**
  - **标识各条主对角线是否已经安放了皇后**
    ——线性表，表长为**2N-1**
  - **标识各条次对角线是否已经安放了皇后**
    ——线性表，表长为**2N-1**
  - **记录当前各行的皇后在第几列—布局状况**
    ——线性表，表长为**N**
- 存储结构的选择

表长固定，有随机存取的要求——顺序表

3.3 栈与递归的实现–N皇后问题

- 数据结构设计
  - **标识每一列是否已经安放了皇后**
    ——顺序表**col**，表长为**N**
  - **标识各条主对角线是否已经安放了皇后**
    ——顺序表**md**，表长为**2N-1**
  - **标识各条次对角线是否已经安放了皇后**
    ——顺序表**sd**，表长为**2N-1**
  - **记录当前各行的皇后在第几列—布局状况**
    ——顺序表**q**，表长为**N**

3.3 栈与递归的实现–N皇后问题

- 算法

```
     Queen( int i, int n ) {
       ( int j = 0;  j < n;  j++ ) {
        f ( 第 i 行第 j 列没有攻击 ) {
          在第 i 行第 j 列安放皇后;
          if ( i == n-1 )  输出一个布局;
          else Queen ( i+1, n );
         撤消第 i 行第 j 列的皇后;
        }
     }
   }
```

col[j]= 1;
md[n+i-j-1]=1;
sd[i+j]=1;
q[i] = j;

!col[j]
&& !md[n+i-j-1]
&& !sd[i+j]

输出q

col[j]= 0;
md[n+i-j-1]=0;
sd[i+j]=0;
q[i] = 0;

3.4 队列

- 定义

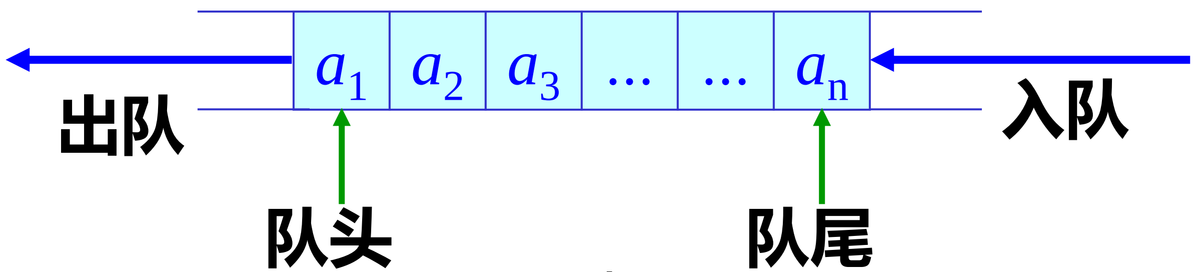
  特殊的线性表：操作受限

  只允许在一端进行插入，而在另一端删除元素

  允许插入的一端为队尾(rear),允许删除的一端为队头(head)

  - **双端队列：限定插入和删除操作在表的两端进行**

- 逻辑特征：**先进先出(FIFO)**
- **ADT Queue：初始化空队、入队、出队、判断队空、判断队满、取队头**



ADT Queue{

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

数据关系：$R=\{R1\}, R1=\{<a_{i-1},a_i> | a_{i-1},a_i \in D, i=2,3,\cdots,n\}$

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：销毁队列 Q

ClearQueue(&Q)

初始条件：队列 Q 已存在

操作结果：将队列 Q 重置为空队列

QueueEmpty(Q)

初始条件：队列 Q 已存在

操作结果：若 Q 为空队列，则返回 TRUE，否则返回 FALSE

QueueLength(Q)

初始条件：队列 Q 已存在

操作结果：返回队列 Q 中数据元素的个数

GetElem(L, i, &e)

GetHead(Q,&e)

初始条件：队列 Q 已存在且非空

操作结果：用 e 返回 Q 中队头元素

EnQueue(&Q, e)

ListInsert(&L, i, 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)

ListDelete(&L, i, &e)

初始条件：队列 Q 已存在且非空

操作结果：删除 Q 的队头元素，并用 e 返回其值

QueueTraverse(Q, visit())

初始条件：队列 Q 已存在且非空

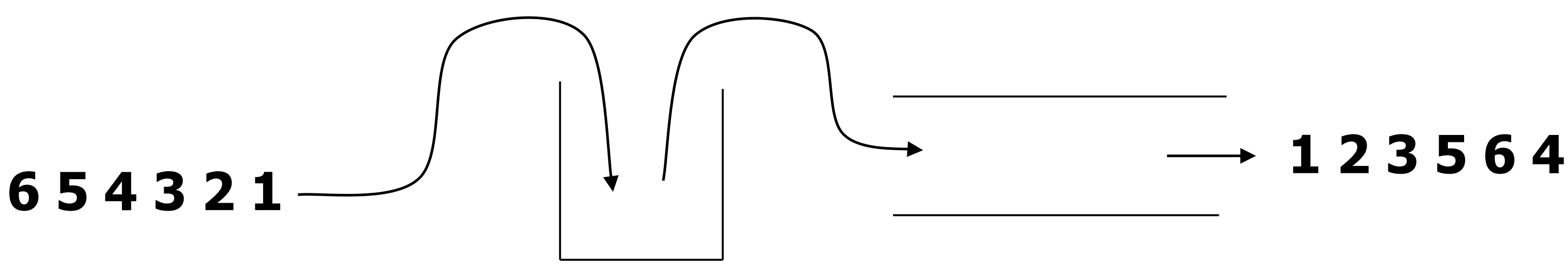
操作结果：从队头到队尾依次对 Q 的每个数据元素调用函数 visit()。一旦 visit()失败，则操作失败

}ADT Queue

栈与队列的概念运用

- 问题**2：设栈S和队列Q的初态均为空，将元素1, 2, 3, 4, 5, 6依次入栈S，一元素出栈后即进入队列Q，若这6个元素出队次序是4,6,5,3,2,1，则栈S至少应能容纳多少个元素?**

答案：**5**



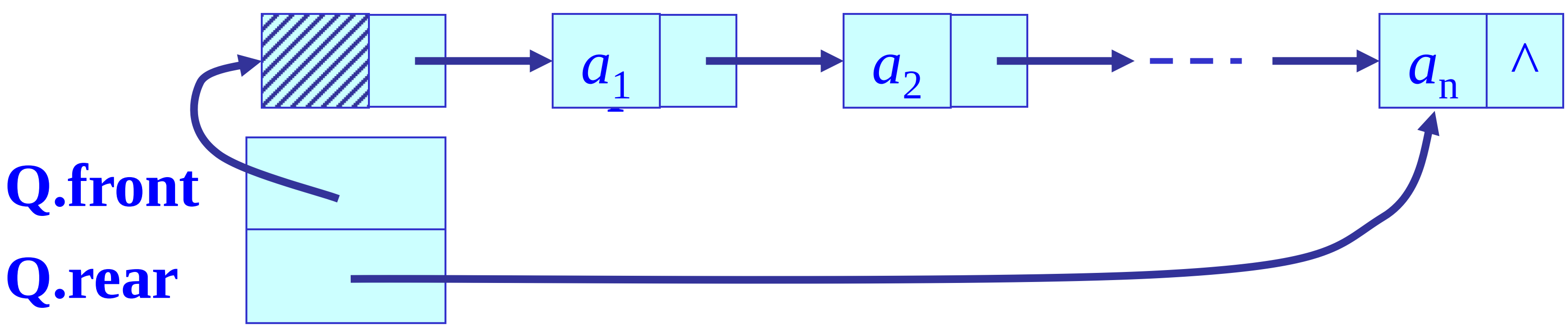
3.4 队列—链队列的表示与实现

- 链队列
  - **约定与类型定义：Q.front和Q.rear的含义**

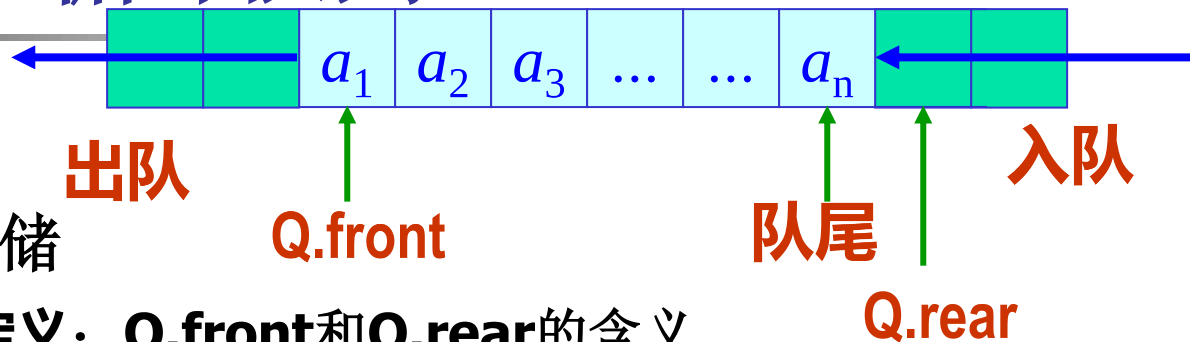
```
typedef struct QNode{
    ElemType        data;
    struct QNode    *next;
}QNode, *QueuePtr;
typedef struct {
    QueuePtr    front;  /* 队头指针，指向头元素*/
    QueuePtr    rear;   /* 队尾指针，指向队尾元素*/
}LinkQueue;
```

3.4 队列—链队列的表示与实现

- 链队列
  - 基本操作的实现
    - **无队满问题(除非分配不出内存), 空间可扩充**
    - **引入头结点(一定需要吗？)**



3.4 队列—循环队列



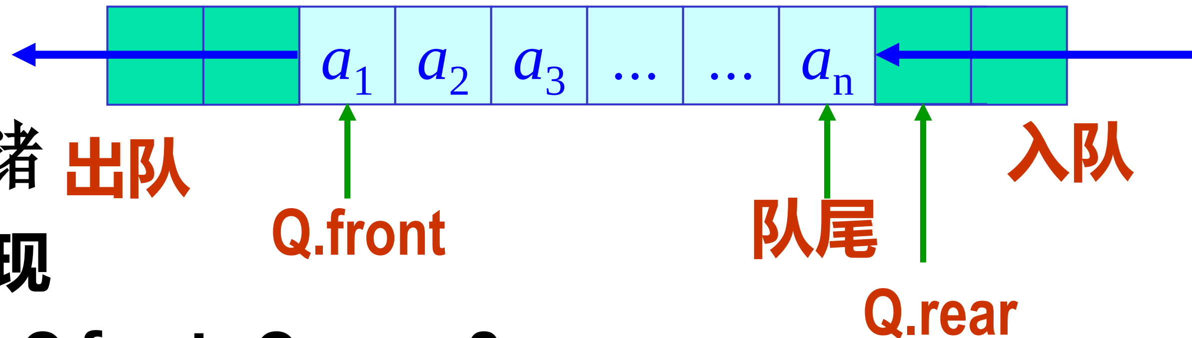
- 循环队列
  - 队列的顺序存储
    - **约定与类型定义：Q.front和Q.rear的含义**

```
#define MAXQSIZE   100        /* 最大队列长度 */
typedef struct{
        ElemType        *base;  /* 存储空间        */
        int       front;   /* 头指针，指向队列的头元素 */
        int       rear;    /* 尾指针，
                              指向队尾元素的下一个位置 */
}SqQueue;    /* 非增量式的空间分配 */
```

    - **删除：**避免大量的移动➔头指针增1

3.4 队列—循环队列

- 队列的顺序存储



  - **基本操作的实现**
    - 初始化空队：**Q.front=Q.rear=0;**
    - 入队：判断是否队满；非队满时，**Q.rear**位置放新插入的元素，**Q.rear++**
    - 出队：判断是否队空，非队空时，**Q.front**位置为待删除的元素，**Q.front++**
    - 队空条件：**Q.front == Q.rear**
    - 队满条件：**Q.rear == MAXQSIZE**

问题： 假上溢

3.4 队列—循环队列

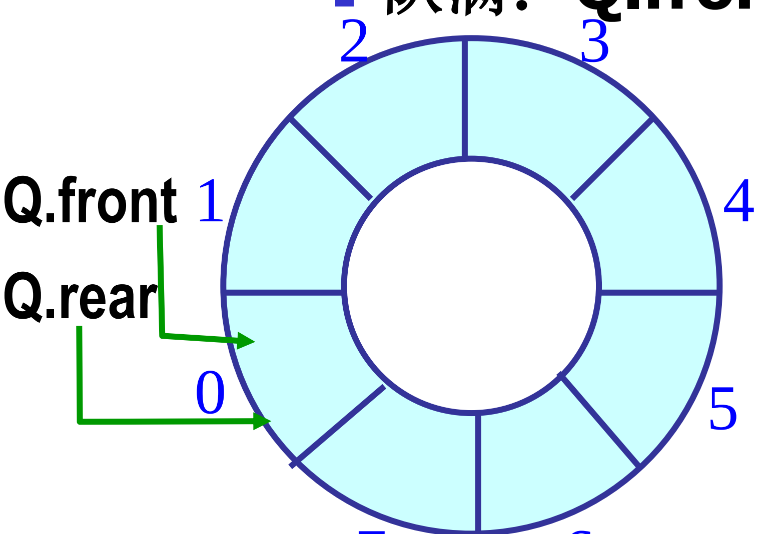
- 循环队列
  - **假上溢的解决办法**
    把顺序队列看成首尾相接的环**(钟表)-**循环队列
  - **基本操作的实现**
    - 入队：……，**Q.rear = (Q.rear+1)%MAXQSIZE**
    - 出队：……，**Q.front = (Q.front+1)%MAXQSIZE**
    - 队空条件：**Q.front == Q.rear**
      由于出队**Q.front**追上了**Q.rear**
    - 队满条件：**Q.front == Q.rear**
      由于入队**Q.rear**追上了**Q.front**

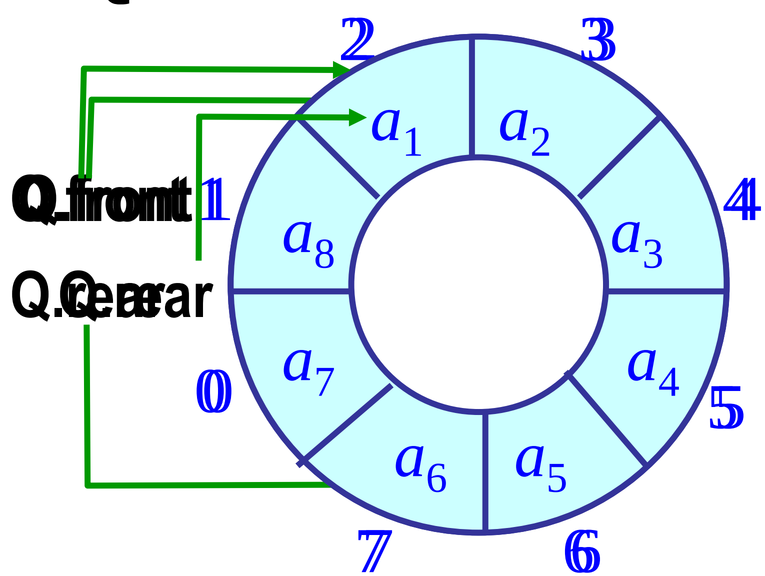
问题：队空和队满的判断条件一样

3.4 队列—循环队列

- 循环队列
  - 约定：**Q.front**和**Q.rear**(队尾的下一个)的含义
  - 队空：**Q.front==Q.rear**
  - 队满：**Q.front==Q.rear**

如何区分队空和队满?

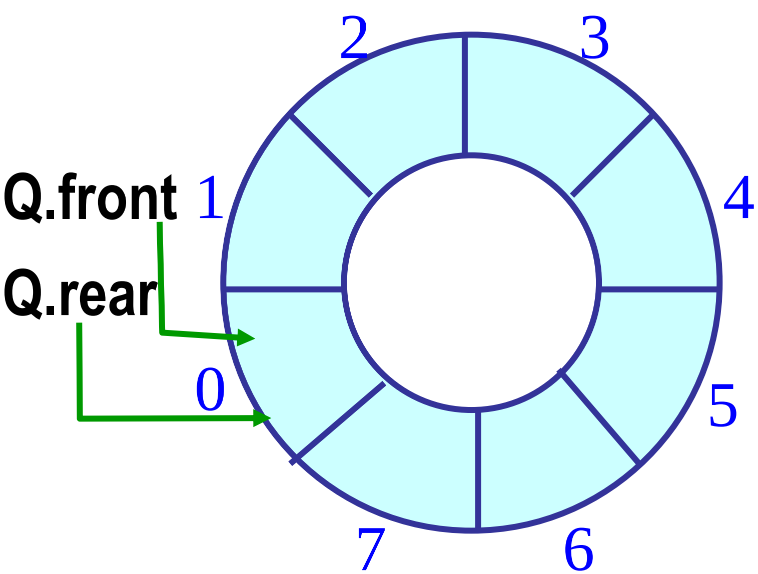


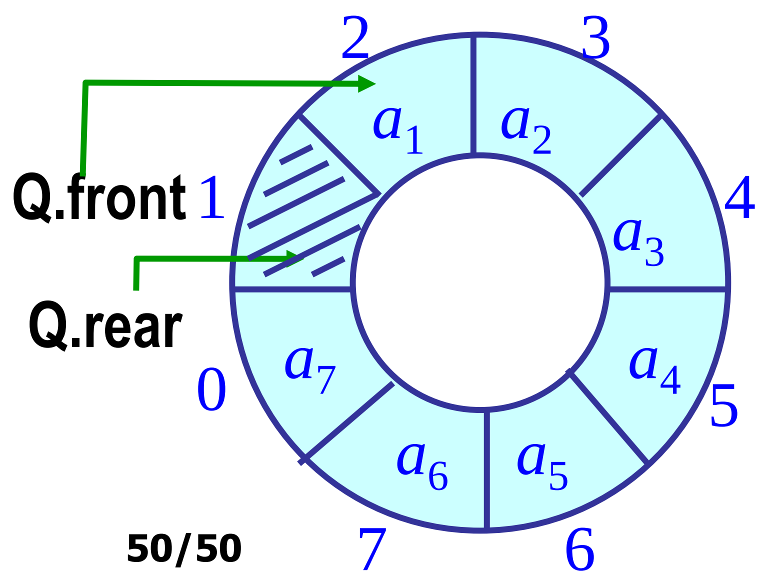


3.4 队列–循环队列

- 区分队空和队满的方法
  - 设标志位(上次的更新动作)：**0-创建/删除，1-插入**
  - 引入队列长度
  - 少用一个元素空间：
    插入前判断 **Q.front==(Q.rear+1)%MAXQSIZE**

有维护的代价





3.4 队列—循环队列

- 难点

连续的存储单元的上下界：[d1,d2]

  - 初始化空队：**Q.front = Q.rear = d1;**
  - 队空：**Q.front==Q.rear**
  - 队满：**Q.front==(Q.rear-d1+1)%(d2-d1+1)+d1**
  - 入队： 前提：队列不满

    Q.base[Q.rear] = e;
    Q.rear = (Q.rear-d1+1)%(d2-d1+1)+d1;
  - 出队：前提：队列不空

    e = Q.base[Q.front];
    Q.front = (Q. front-d1+1)%(d2-d1+1)+d1

3.5 离散事件模拟

- 问题描述 **P65**
  计算一天中客户在银行逗留的平均时间
- 数据结构设计
  - 四个窗口---- 四个队列
  - 客户
    - 到达时刻
    - 离开时刻(办理业务所需的时间，等待时间)
  - 事件表：事件驱动
    - 客户到达事件：入队(选择所含元素最少的队列)、产生离开事件插入到事件表中
    - 客户离开事件(区分是哪个窗口)
    - 事件：事件类型、事件发生时刻