第三章 栈和队列

第三章 栈和队列

3.1 栈

3.2 栈的应用

3.3 队列

3.4 栈在递归的实现

3.1 栈

- 线性表的概念

  定义 n（≥ 0）个数据元素的有限序列，记作 $(a_1, a_2, \cdots, a_n)$ 。

  a_i 是表中数据元素，n 是表长度。

  a_i 是 a_{i+1} 的直接前驱元素，

  a_i 是 a_{i-1} 的直接后继元素。

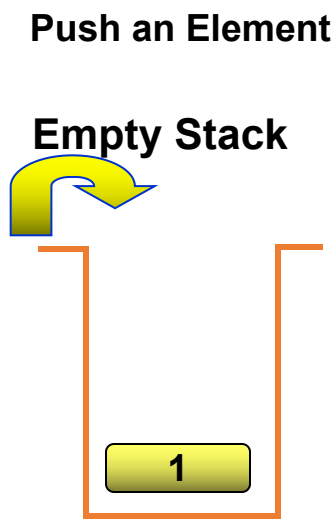
- 栈：栈是一种特殊的表。在该表中的插入和删除都限制在表的同一端进行。做插入和删除的一端称栈顶，另一端称栈底。

栈的定义

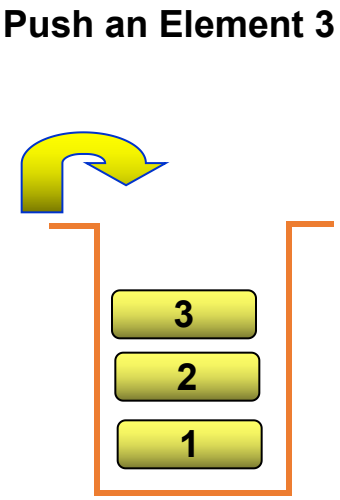
栈：插入和删除都限制在同一端的线性表。
栈顶、栈底、空栈

- 往栈中装入元素即为插入 称进栈
- 从栈中取出元素即为删除 称退栈
- 栈又称为“后进先出表”(LIFO表)

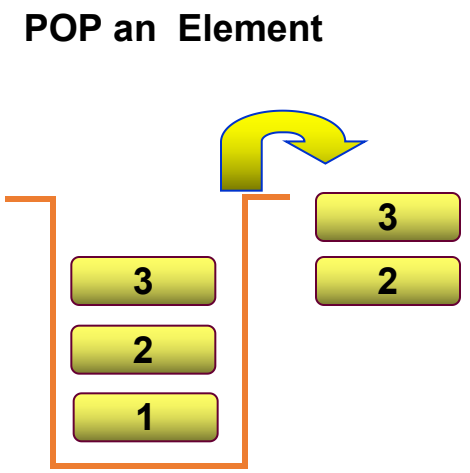
- PUSH：它是在堆栈顶部插入新元素的过程。



- PUSH：它是在堆栈顶部插入新元素的过程。



- POP：它是从堆栈顶部删除元素的过程。



进栈顺序：1—2—3

出栈顺序：3—2—1

特点：后进先出Last In First Out （LIFO）

线性表的基本操作：

InitList(&L)：初始化表。构造一个空的线性表L，分配内存空间。

DestroyList(&L)：销毁操作。销毁线性表，并释放线性表L所占用的内存空间。

ListInsert(&L,i,e)：插入操作。在表L中的第i个位置上插入指定元素e。

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

LocateElem(L,e)：按值查找操作。在表L中查找具有给定关键字值的元素。

GetElem(L,i)：按位查找操作。获取表L中第i个位置的元素的值。

其他常用操作：

Length(L)：求表长。返回线性表L的长度，即L中数据元素的个数。

PrintList(L)：输出操作。按前后顺序输出线性表L的所有元素值。

Empty(L)：判空操作。若L为空表，则返回true，否则返回false。

栈的基本操作

InitStack(SqStack &S) 构造一个空栈

GetTop(SqStack S,SElemType &e) 取栈顶元素

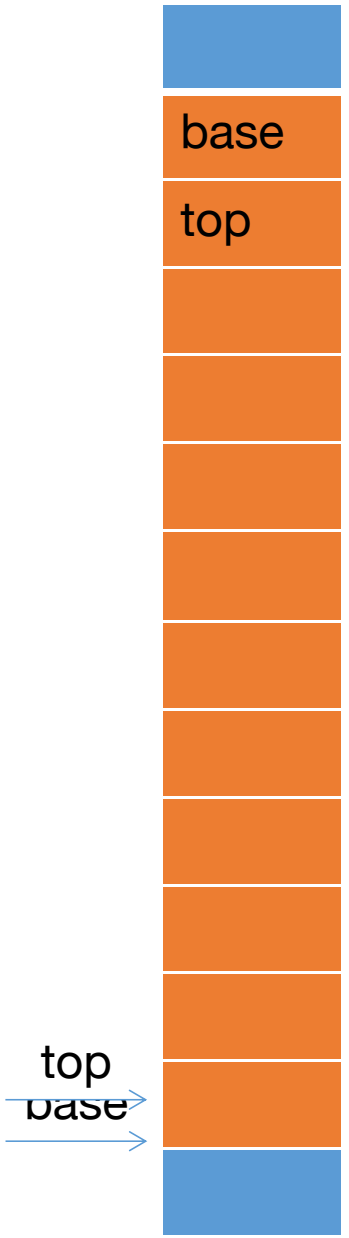
Push(SqStack &S,SElemType e) 进栈

Pop(SqStack &S,SElemType &e) 退栈

StackEmpty(SqStack S) 判栈空

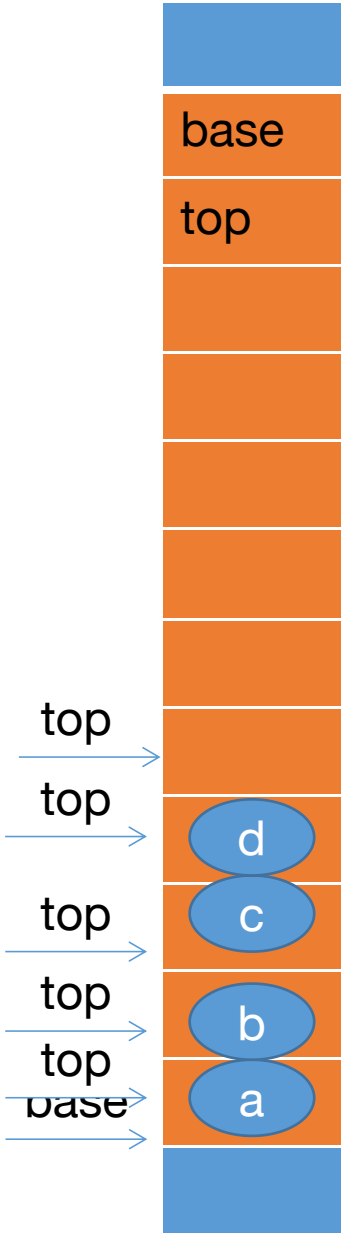
顺序栈的定义

```
#define STACK_INIT_SIZE      10
 typedef struct
       {  SElemType *base;  //栈底指针
          SElemType *top;    //栈顶指针
          int stacksize;       //当前已经分配的存储空间
        } SqStack;

     bool InitStack(SqStack &S)
      {
           S.base=(SElemType *) malloc(STACK_INIT_SIZE*sizeof(SElemType));
           if (!S.base)  return false;   //分配失败
           S.top=S.base;
           S.stacksize= STACK_INIT_SIZE;
           return true;
       }
```



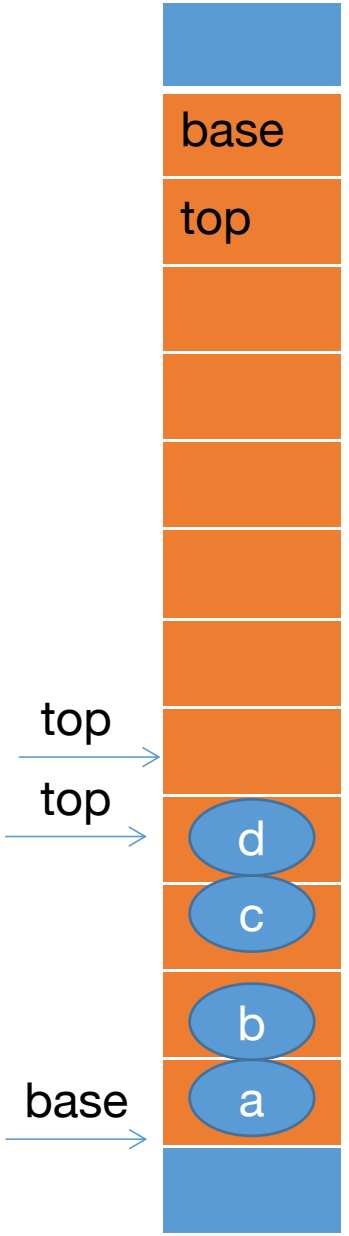
进栈操作

```
#define STACK_INIT_SIZE     10
#define STACKINCREMENT      10
 typedef struct
       {  SElemType *base;  //栈底指针
          SElemType *top;   //栈顶指针
          int stacksize;      //当前已经分配的存储空间
        } SqStack;

bool Push(SqStack &S,SElemType e)
 {
    if (S.top-S.base>=S.stacksize)  //栈满
    {
       S.base=(SElemType *)realloc(S.base,S.stacksize+STACKINCREMENT)*sizeof(SElemType));
       if (!S.base) return false;
       S.top=S.base+S.stacksize;
       S.stacksize+=STACKINCREMENT;
     }
    * S.top++ = e;                //元素存入栈顶
    return true;
}
```



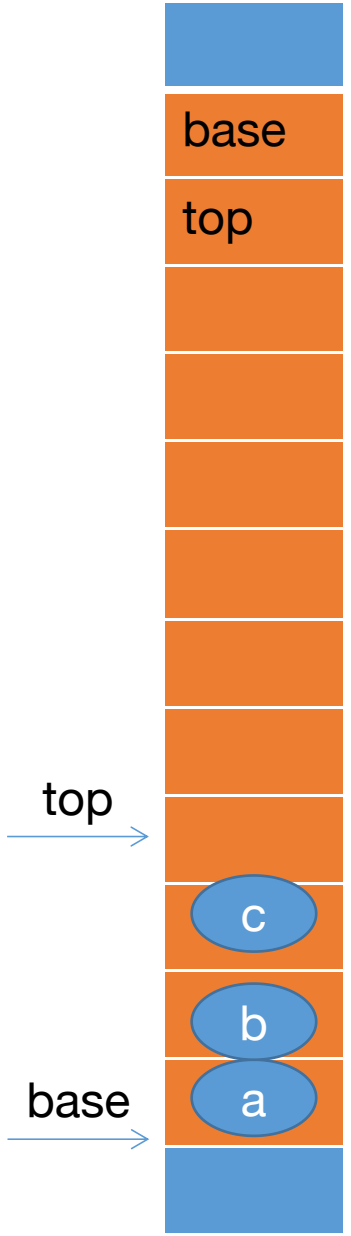
出栈操作

```
#define STACK_INIT_SIZE     10
#define STACKINCREMENT      10
typedef struct
    { SElemType *base; //栈底指针
      SElemType *top;   //栈顶指针
      int stacksize;     //当前已经分配的存储空间
    } SqStack;

bool Pop(SqStack &S,SElemType &e)
{
    if (S.top==S.base) return false;
    e=*--S.top;
    return true;
}
```



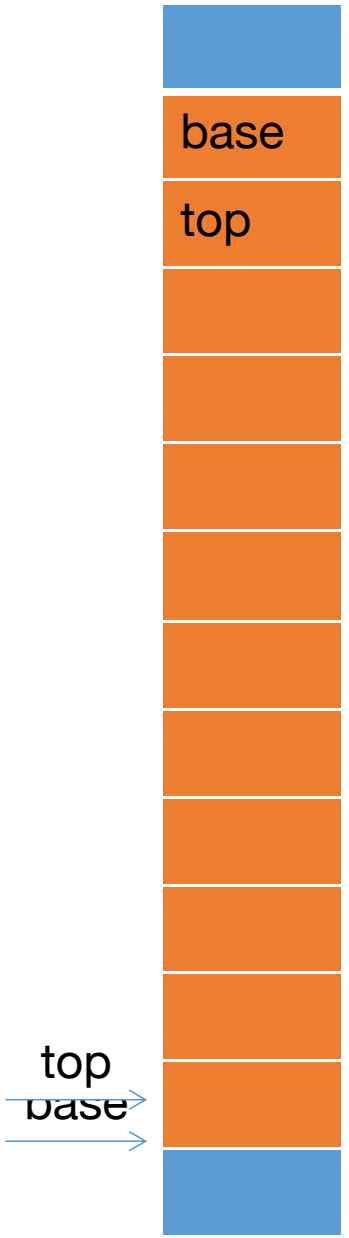
取栈顶元素

```
bool Pop(SqStack &S,SElemType &e)
{
    if (S.top==S.base) return false;
    e=*--S.top;
    return true;
}
bool GetTop(SqStack S,SElemType &e)
{
   if (S.top==S.base) return false;
   e=*(S.top-1);
   return true;
}
```



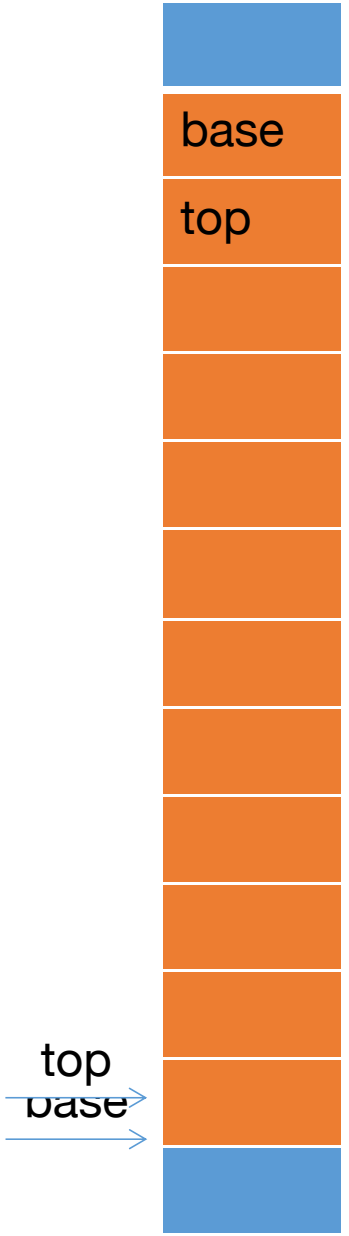
判空

```
#define STACK_INIT_SIZE      10
#define STACKINCREMENT       10
 typedef struct
        {  SElemType *base;  //栈底指针
           SElemType *top;   //栈顶指针
           int stacksize;      //当前已经分配的存储空间
         } SqStack;

bool StackEmpty(SqStack S)
{
   if (S.top==S.base) return false;
   return ture;
}
```

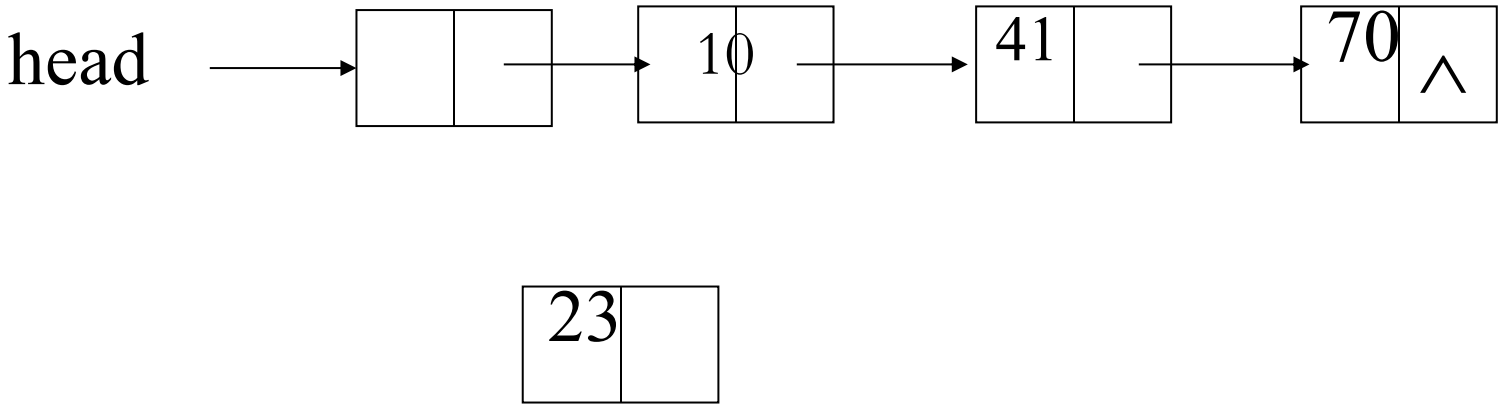


主函数测试

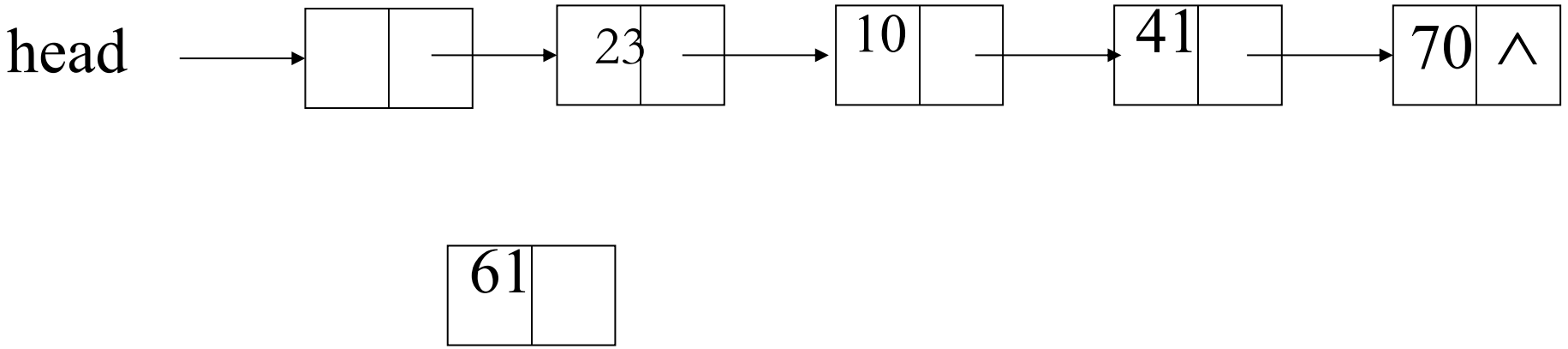
SStack.cpp



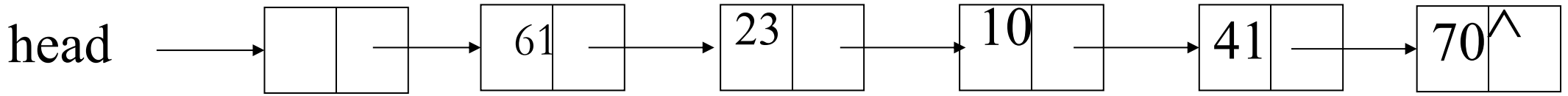
知识回顾：头插法建单链表



知识回顾：头插法建单链表



知识回顾：头插法建单链表



链栈的定义

```
typedef  struct  Snode {
        SElemType data;
        struct Snode *next;
    } *LinkStack;
```

1、假定利用数组a[n]顺序存储一个栈，用top表示栈顶指针，top==-1表示栈空，并已知栈未满，当元素x进栈时所执行的操作为（ ）。

A. a[--top]=x

B. a[top--]=x

C. a[++top]=x

D. a[top++]=x

2、向一个栈顶指针为top的链栈中插入一个x结点，则执行（ ）。

A. top->next=x

B. x->next=top->next;top->next=x

C. x->next=top;top=x

D. x->next=top;top=top->next

3、链栈执行Pop操作，并将出栈的元素存在x中应该执行（）。

A. x=top;top=top->next

B. x=top->data

C. top=top->next;x=top->data

D. x=top->data;top=top->next

4、3个不同元素依次进栈，能得到（）种不同的出栈序列。

A. 4

B. 5

C. 6

D. 7

5、设a、b、c、d、e、f以所给的次序进栈，若在进栈操作时，允许出栈操作，则下面得不到的序列为（）。

A. fedcba

B. bcafed

C. dcefba

D. cabdef

6、若a、b、c、d、e、f依次进栈，允许进栈、退栈操作交替进行，但不允许连续3次进行退栈操作，则不可能得到的出栈序列是（）。

A. dcebfa

B. cbdaef

C. bcaefd

D. afedcb

7、设栈S和队列Q的初始状态均为空，元素abcdefg依次进入栈S。若每个元素出栈后立即进入队列Q，且7个元素出队的顺序是bdcfeag，则栈S的容量至少是（）。

A. 1

B. 2

C. 3

D. 4

8、若一个栈的输入序列是P1,P2,…,Pn，其输出序列是1，2，3，…，n,若P3=1，则P1的值（ ）。

A. 可能是2

B.一定是2

C. 不可能是2

D.不可能是3

9、一个栈的入栈序列是1，2，3，…，其输出序列是n,P1,P2,…,Pn，若P2=3，则P3可能取值的个数是（ ）。

A. n-3

B.n-2

C. n-1

D.无法确定

10、元素a、b、c、d、e依次进入初始为空的栈中，若元素进栈后可停留、可出栈，直到所有元素都出栈，则在所有可能的出栈序列中，以元素d开头的序列个数是（ ）。

A. 3

B.4

C. 5

D.6

3.2 栈的应用
——括号匹配

```
void test() {
    int a[10][10];
    int x = 10*(20*(1+1)-(3-2));
    printf("加油! 奥利给! ";
}
```

Expected ')'
to match this '('

```
void test() {
    int a[10][10];
    int x = 10*(20*(1+1)-(3-2));
    printf("加油! 奥利给! ");
}
```

括号匹配问题

(((( ))))

① ② ③ ④④ ③ ② ①

最后出现的左括号最先被匹配（LIFO)

((( )) ( ))

① ② ③ ③ ② ④ ④ ①

每出现一个右括号，就"消耗"一个左括号

括号匹配问题

{ （ （ ） ） [ ] }

① ② ③ ③ ② ④ ④ ①

遇到左括号就入栈

遇到右括号，就"消用"一个左括号

括号匹配问题

{ （ （ ）] [ ]

① ② ③ ③ ②

遇到左括号就入栈

遇到右括号，就"消用"一个左括号

算法演示

{ （ （ ）] [ ]

① ② ③ ③ ②

遇到左括号就入栈

遇到右括号，就"消用"一个左括号

算法演示

{ （ （ ） ） }] （ ）

① ② ③ ③ ② ①

遇到左括号就入栈

遇到右括号，就"消用"一个左括号

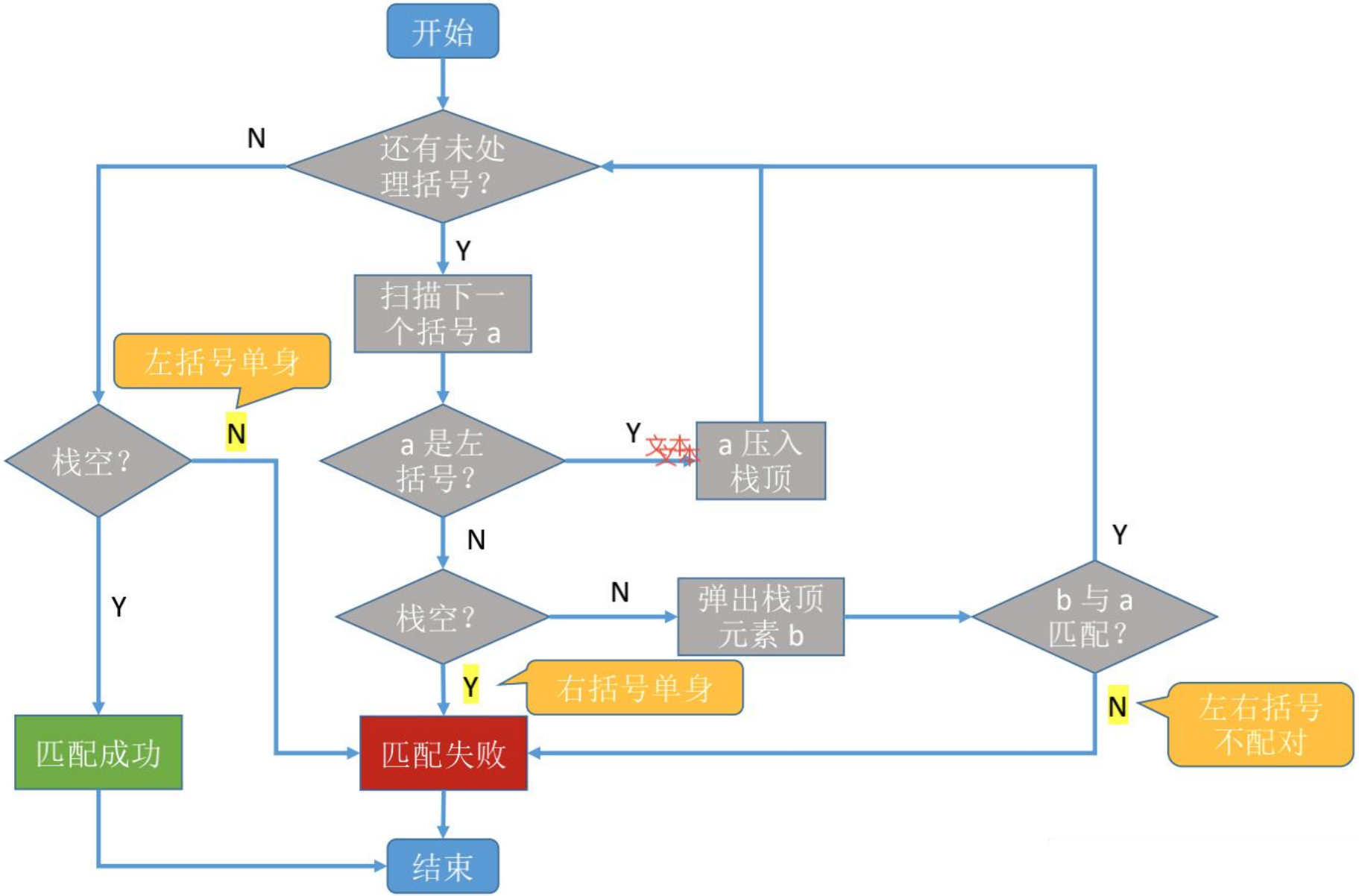
算法演示

{ { (( )) [ ] }

① ② ③ ④ ④ ③ ⑤ ⑤ ②

遇到左括号就入栈

遇到右括号，就"消用"一个左括号


开始
还有未处理括号？
N
Y
扫描下一个括号 a
左括号单身
栈空？
N
Y
a 是左括号？
Y
a 压入栈顶
N
栈空？
N
弹出栈顶元素 b
b 与 a 匹配？
Y
N
左右括号不配对
Y
右括号单身
匹配成功
匹配失败
结束

```
bool BracketCheck(SqStack S,char str[],int length)
{
    for(int i=0;i<length;i++)
    {
        if(str[i]=='('||str[i]=='['||str[i]=='{')
        {
            Push(S,str[i]);
        }
        else
        {
            if(StackEmpty(S))
                return false;

            char topElem;
            Pop(S,topElem);
            if(str[i]==')'&& topElem!='(')
                return false;
            if(str[i]==']'&& topElem!='[')
                return false;
            if(str[i]=='}'&& topElem!='{')
                return false;
        }
    }
    return StackEmpty(S);
}
```

3.2 栈的应用
——算数表达式

$$((15 \div (7 - (1 + 1))) \times 3) - (2 + (1 + 1))$$

③ ② ① ④ ⑦ ⑥ ⑤

$$15 \div 7 - 1 + 1 \times 3 - 2 + 1 + 1$$

① ② ④ ③ ⑤ ⑥ ⑦

由三个部分组成：操作数、运算符、界限符

波兰数学家的灵感

$((15 \div (7 - (1 + 1))) \times 3) - (2 + (1 + 1))$

可以不用界限符也能无歧义地表达运算顺序？？

Reverse Polish notation（逆波兰表达式 = 后缀表达式）
Polish notation（波兰表达式 = 前缀表达式）

中缀、后缀、前缀表达式

中缀表达式	后缀表达式	前缀表达式
运算符在两个操作数中间	运算符在两个操作数后面	运算符在两个操作数前面
a + b	a b +	+ a b
a + b - c	a b + c -	- + a b c
a+b-c*d	a b + c d * -	- + a b * c d

中缀表达式转后缀表达式（手算）

中缀转后缀的手算方法:

① 确定中缀表达式中各个运算符的运算顺序

② 选择下一个运算符，按照「左操作数 右操作数 运算符」的方式组合成一个新的操作数

③ 如果还有运算符没被处理，就继续 ②

$$((15 \div (7 - (1 + 1))) \times 3) - (2 + (1 + 1))$$

③ ② ① ④ ⑦ ⑥ ⑤

15 7 1 1 + - ÷ 3 × 2 1 1 + + -

①②③ ④ ⑤⑥⑦

中缀表达式转后缀表达式（手算）

中缀转后缀的手算方法：

① 确定中缀表达式中各个运算符的运算顺序

② 选择下一个运算符，按照「左操作数 右操作数 运算符」的方式组合成一个新的操作数

③ 如果还有运算符没被处理，就继续 ②

A+B*(C-D)-E/F

③ ② ① ⑤ ④

A B C D - * + E F / -

A+B*(C-D)-E/F

⑤ ③ ② ④ ①

A B C D - * E F / - +

“左优先”原则：只要左边的运算符能先计算，就优先算左边的，保证手算和机算结果相同

中缀表达式转后缀表达式（手算）

中缀转后缀的手算方法：

① 确定中缀表达式中各个运算符的运算顺序

② 选择下一个运算符，按照「左操作数 右操作数 运算符」的方式组合成一个新的操作数

③ 如果还有运算符没被处理，就继续 ②

“左优先”原则：只要左边的运算符能先计算，就优先算左边的

A + B - C * D / E + F

后缀表达式的计算（手算）

中缀表达式：((15 ÷ (7 − (1 + 1))) × 3) − (2 + (1 + 1))

③ ② ① ④ ⑦ ⑥ ⑤

后缀表达式：15 7 1 1 + - ÷ 3 × 2 1 1 + + -

①②③ ④ ⑤⑥⑦

从左往右扫描，每遇到一个运算符，就让运算符前面最近的两个操作数执行对应运算，合体为一个操作数

注意：两个操作数的左右顺序

后缀表达式的计算（手算）

后缀表达式的手算方法：

从左往右扫描，每遇到一个运算符，就让运算符前面最近的两个操作数执行对应运算，合体为一个操作数

注意：两个操作数的左右顺序

A + B - C * D / E + F

A B + C D * E / - F +

总结规律：LIFO

后缀表达式的计算（机算）

用栈实现后缀表达式的计算：

①从左往右扫描下一个元素，直到处理完所有元素

②若扫描到操作数则压入栈，并回到①；否则执行③

③若扫描到运算符，则弹出两个栈顶元素，执行相应运算，运算结果压回栈顶，回到①

A + B - C * D / E + F

A B + C D * E / - F +

后缀表达式的计算（机算）

用栈实现后缀表达式的计算:

①从左往右扫描下一个元素，直到处理完所有元素

②若扫描到操作数则压入栈，并回到①；否则执行③

③若扫描到运算符，则弹出两个栈顶元素，执行相应运算，运算结果压回栈顶，回到①

中缀表达式: $((15 \div (7 - (1 + 1))) \times 3) - (2 + (1 + 1))$

③ ② ① ④ ⑦ ⑥ ⑤

后缀表达式：15 7 1 1 + - ÷ 3 × 2 1 1 + + -

①②③ ④ ⑤⑥⑦

中缀表达式转前缀表达式（手算）

① 确定中缀表达式中各个运算符的运算顺序

② 选择下一个运算符，按照「运算符 左操作数 右操作数」的方式组合成一个新的操作数

③ 如果还有运算符没被处理，就继续 ②

“右优先”原则：只要右边的运算符能先计算，就优先算右边的

A + B * (C - D) – E / F

③ ② ① ⑤ ④

- + A * B - C D / E F

⑤ ③ ② ① ④

A + B * (C - D) – E / F

⑤ ③ ② ④ ①

+ A - * B - C D / E F

⑤ ④ ③ ② ①

中缀表达式转前缀表达式（手算）

((15 ÷ (7 – (1 + 1))) × 3) – (2 + (1 + 1))

③ ② ① ④ ⑦ ⑥ ⑤

中缀表达式:

15 7 1 1 + - ÷ 3 × 2 1 1 + + -

((15 ÷ (7 – (1 + 1))) × 3) – (2 + (1 + 1))

⑥ ④ ③ ⑤ ⑦ ② ①

- ÷ × 15 - 7 + 1 1 3 + 2 + 1 1

⑦⑥⑤ ④ ③ ② ①

前缀表达式的计算

用栈实现前缀表达式的计算:

①从右往左扫描下一个元素，直到处理完所有元素

②若扫描到操作数则压入栈，并回到①；否则执行③

③若扫描到运算符，则弹出两个栈顶元素，执行相应运算，运算结果压回栈顶，回到① 栈

注意：先出栈的是“左操作数”

((15 ÷ (7 − (1 + 1))) × 3) − (2 + (1 + 1))

⑥ ④ ③ ⑤ ⑦ ② ①

- ÷ × 15 - 7 + 1 1 3 + 2 + 1 1

⑦ ⑥ ⑤ ④ ③ ② ①

中缀表达式转后缀表达式（手算）

中缀转后缀的手算方法：

① 确定中缀表达式中各个运算符的运算顺序

② 选择下一个运算符，按照「左操作数 右操作数 运算符」的方式组合成一个新的操作数

③ 如果还有运算符没被处理，就继续 ②

“左优先”原则：只要左边的运算符能先计算，就优先算左边的

A + B - C * D / E + F

① ④ ② ③ ⑤

A B + C D * E / - F +

① ② ③④ ⑤

中缀表达式转后缀表达式（机算）

初始化一个栈，用于保存暂时还不能确定运算顺序的运算符。

从左到右处理各个元素，直到末尾。可能遇到三种情况：

① 遇到操作数。直接加入后缀表达式。

② 遇到界限符。遇到“(”直接入栈；遇到“)”则依次弹出栈内运算符并加入后缀表达式，直到弹出“(”为止。注意：“(”不加入后缀表达式。

③ 遇到运算符。依次弹出栈中优先级高于或等于当前运算符的所有运算符，并加入后缀表达式，若碰到“(”或栈空则停止。之后再把当前运算符入栈。

按上述方法处理完所有字符后，将栈中剩余运算符依次弹出，并加入后缀表达式。

A＋B-C*D/E+F

手算：

机算：

中缀表达式转后缀表达式（机算）

初始化一个栈，用于保存暂时还不能确定运算顺序的运算符。

从左到右处理各个元素，直到末尾。可能遇到三种情况：

① 遇到操作数。直接加入后缀表达式。

② 遇到界限符。遇到“(”直接入栈；遇到“)”则依次弹出栈内运算符并加入后缀表达式，直到弹出“(”为止。注意：“(”不加入后缀表达式。

③ 遇到运算符。依次弹出栈中优先级高于或等于当前运算符的所有运算符，并加入后缀表达式，若碰到“(”或栈空则停止。之后再把当前运算符入栈。

按上述方法处理完所有字符后，将栈中剩余运算符依次弹出，并加入后缀表达式。

A+B*(C-D)-E/F

手算：

机算：

中缀表达式转后缀表达式（机算）

初始化一个栈，用于保存暂时还不能确定运算顺序的运算符。

从左到右处理各个元素，直到末尾。可能遇到三种情况：

① 遇到操作数。直接加入后缀表达式。

② 遇到界限符。遇到“(”直接入栈；遇到“)”则依次弹出栈内运算符并加入后缀表达式，直到弹出“(”为止。注意：“(”不加入后缀表达式。

③ 遇到运算符。依次弹出栈中优先级高于或等于当前运算符的所有运算符，并加入后缀表达式，若碰到“(”或栈空则停止。之后再把当前运算符入栈。

按上述方法处理完所有字符后，将栈中剩余运算符依次弹出，并加入后缀表达式。

$((15 \div (7 - (1 + 1))) \times 3) - (2 + (1 + 1))$

手算：

机算：

中缀表达式的计算（用栈实现）

用栈实现中缀表达式的计算：

初始化两个栈，操作数栈和运算符栈

若扫描到操作数，压入操作数栈

若扫描到运算符或界限符，则按照“中缀转后缀”相同的逻辑压入运算符栈（期间也会弹出运算符，每当弹出一个运算符时，就需要再弹出两个操作数栈的栈顶元素并执行相应运算，运算结果再压回操作数栈）

后缀表达式的计算（机算）

①从左往右扫描下一个元素，直到处理完所有元素

②若扫描到操作数则压入栈，并回到①；否则执行③

③若扫描到运算符，则弹出两个栈顶元素，执行相应运算，运算结果压回栈顶，回到①

A + B - C * D / E + F

① ④ ② ③ ⑤

A B + C D * E / - F +

栈：用于存放当前暂时还不能确定运算次序的运算符

中缀表达式的计算（用栈实现）

两个算法的结合：中缀转后缀+后缀表达式求值

用栈实现中缀表达式的计算：

初始化两个栈，操作数栈和运算符栈

若扫描到操作数，压入操作数栈

若扫描到运算符或界限符，则按照“中缀转后缀”相同的逻辑压入运算符栈（期间也会弹出运算符，每当弹出一个运算符时，就需要再弹出两个操作数栈的栈顶元素并执行相应运算，运算结果再压回操作数栈）

A + B - C * D / E + F

中缀表达式的计算（用栈实现）

两个算法的结合：中缀转后缀+后缀表达式求值

用栈实现中缀表达式的计算：

初始化两个栈，操作数栈和运算符栈

若扫描到操作数，压入操作数栈

若扫描到运算符或界限符，则按照“中缀转后缀”相同的逻辑压入运算符栈（期间也会弹出运算符，每当弹出一个运算符时，就需要再弹出两个操作数栈的栈顶元素并执行相应运算，运算结果再压回操作数栈）

$((15 \div (7 - (1 + 1))) \times 3) - (2 + (1 + 1))$

3.3 队列

- 线性表的概念

  定义 n（≥ 0）个数据元素的有限序列，记作 $(a_1, a_2, \cdots, a_n)$。

  a_i 是表中数据元素，n 是表长度。

  a_i 是 a_{i+1} 的直接前驱元素，

  a_i 是 a_{i-1} 的直接后继元素。

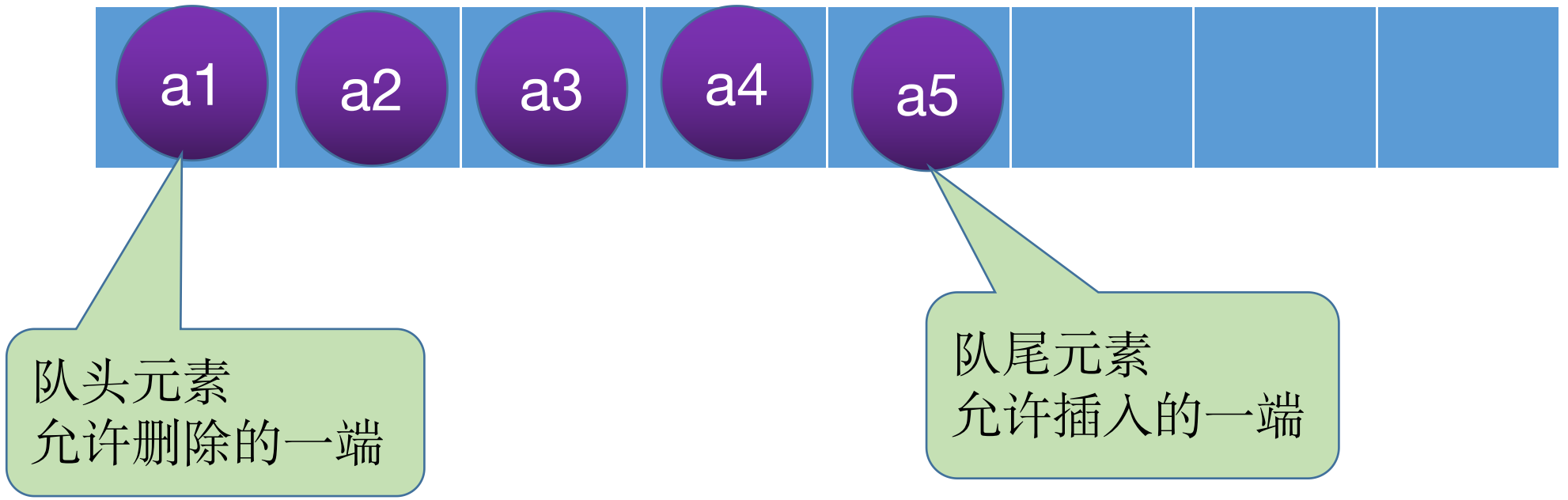
- 栈：栈是一种特殊的表。在该表中的插入和删除都限制在表的同一端进行。
- 队列（Queue）：是只允许在一端进行插入，在另一端删除的线性表。

3.3 队列

队列（Queue）：是只允许在一端进行插入，在另一端删除的线性表。

队列的定义

队列（Queue）：是只允许在一端进行插入，在另一端删除的



队列的特点：先进先出First In First Out（FIFO）

队列的基本操作

InitQueue(&Q)：初始化队列，构造一个空队列Q。

DestroyQueue(&Q)：销毁队列。销毁并释放队列Q所占用的内存空间。

EnQueue(&Q,x)：入队，若队列Q未满，将x加入，使之成为新的队尾。

DeQueue(&Q,&x)：出队，若队列Q非空，删除队头元素，并用x返回。

GetHead(Q,&x)：读队头元素，若队列Q非空，则将队头元素赋值给x。

其他常用操作：

QueueEmpty(Q)：判队列空，若队列Q为空返回true，否则返回false。

队列的顺序实现

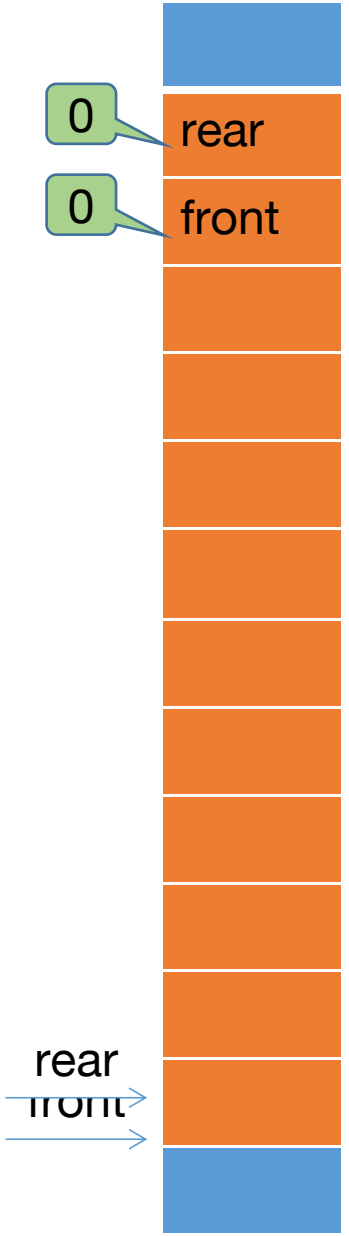
```
#define MaxSize 10
typedef struct{
ElemType data[MaxSize];
int front, rear;
}SqQueue;
```

队列的顺序实现

```
#define MaxSize 10
typedef struct{
ElemType data[MaxSize];
int front, rear;
}SqQueue;
```

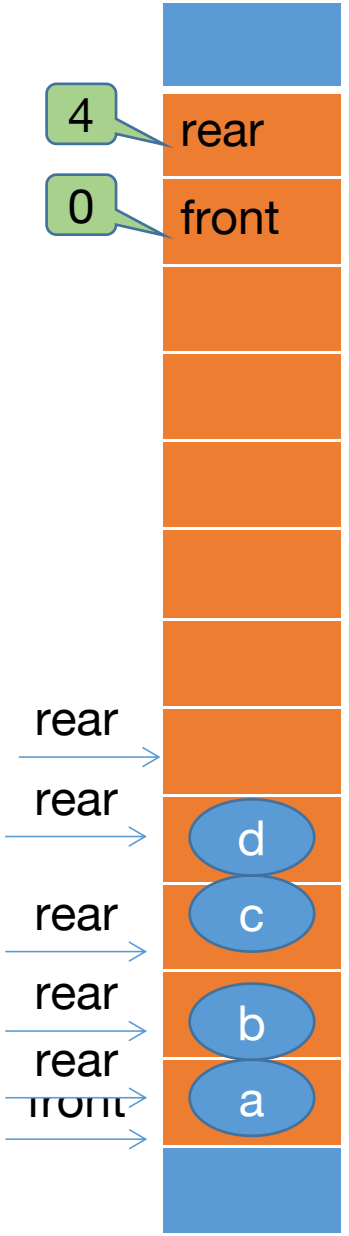
```
void InitQueue(SqQueue &Q){
    //初始时 队头、队尾指针指向
    Q.rear=Q.front=0;
}
```

```
//判断队列是否为空
bool QueueEmpty(SqQueue Q){
      if（Q.rear==Q.front）  //队空条件
            return true;
      else
            return false;
}
```



队列的顺序实现

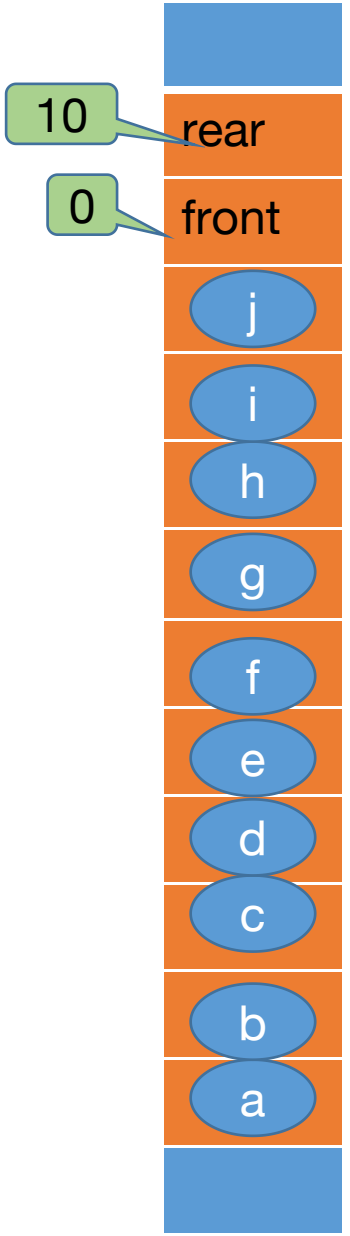
```
#define MaxSize 10
typedef struct{
ElemType data[MaxSize];
int front, rear;
}SqQueue;
booL Enueue(SqQueue &Q,ElemType x){
  if（队列已满){
        return false;//队满则报错
  Q.data[Q.rear]=x;//将x插入队尾
  Q.rear=Q.rear+1;//队尾指针后
  return true;
}
```



入队操作

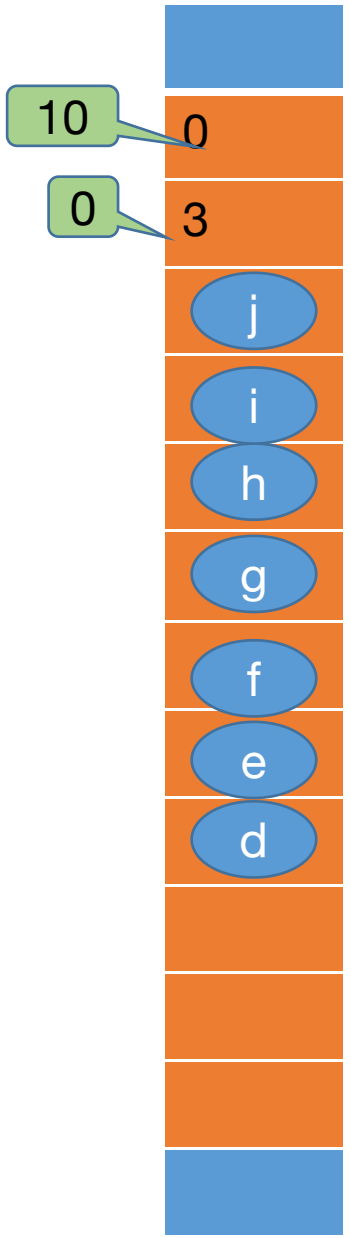
```
#define MaxSize 10
typedef struct{
ElemType data[MaxSize];
int front, rear;
}SqQueue;
```

队满条件？？？ rear==MaxSize？？？



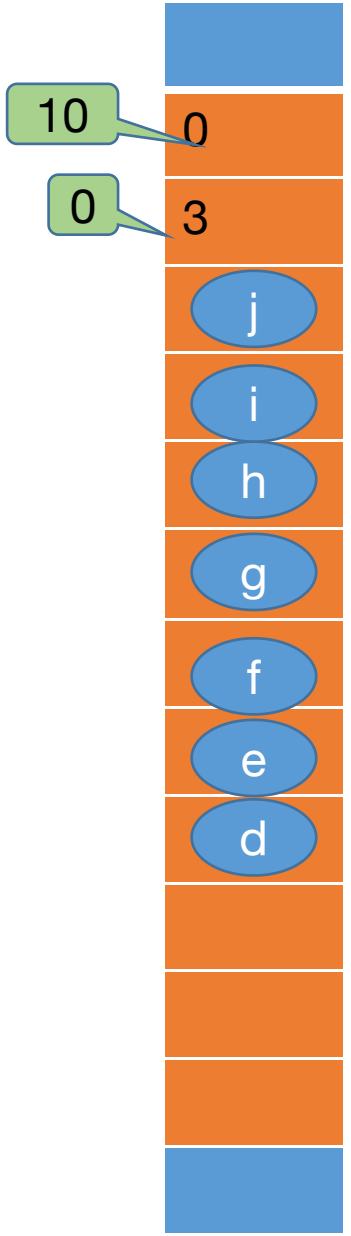
入队操作

```
#define MaxSize 10
typedef struct{
ElemType data[MaxSize];
int front, rear;
}SqQueue;

booL Enueue(SqQueue &Q,ElemType x){
  if（队列已满){
        return false;//队满则报错
  Q.data[Q.rear]=x;//将x插入队尾
  Q.rear=(Q.rear+1)%MaxSize;//队尾指针后
  return true;
}
```



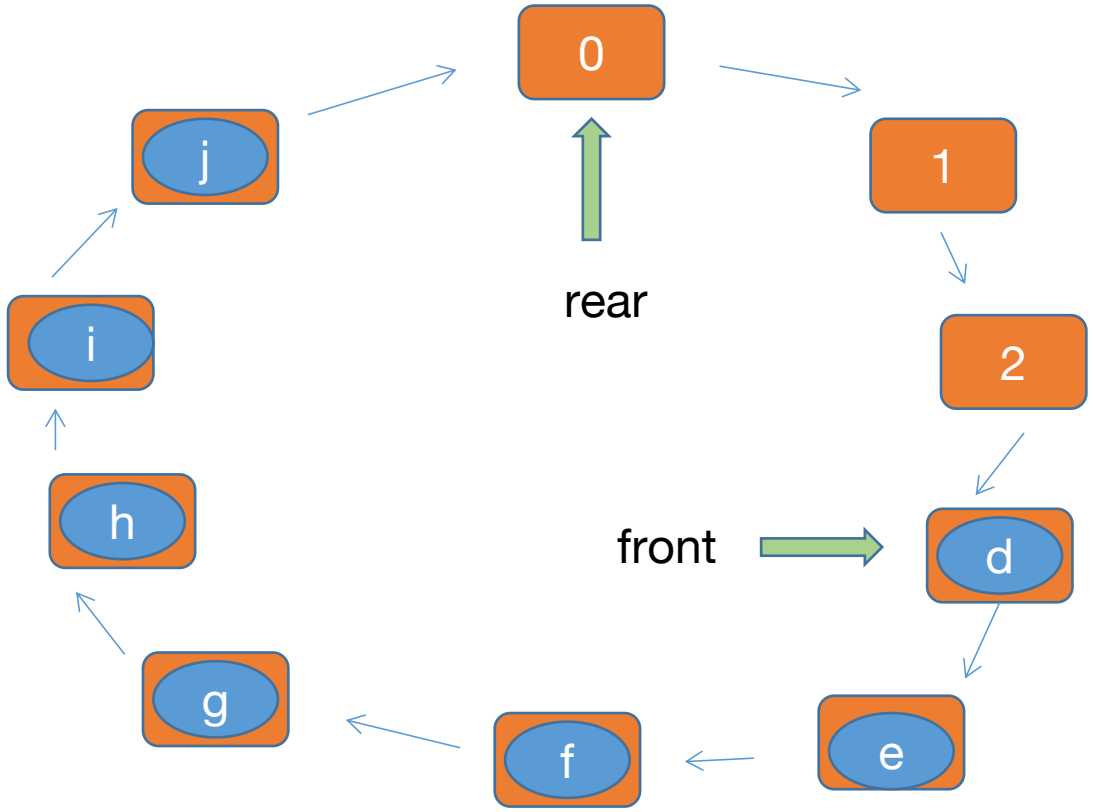
循环队列

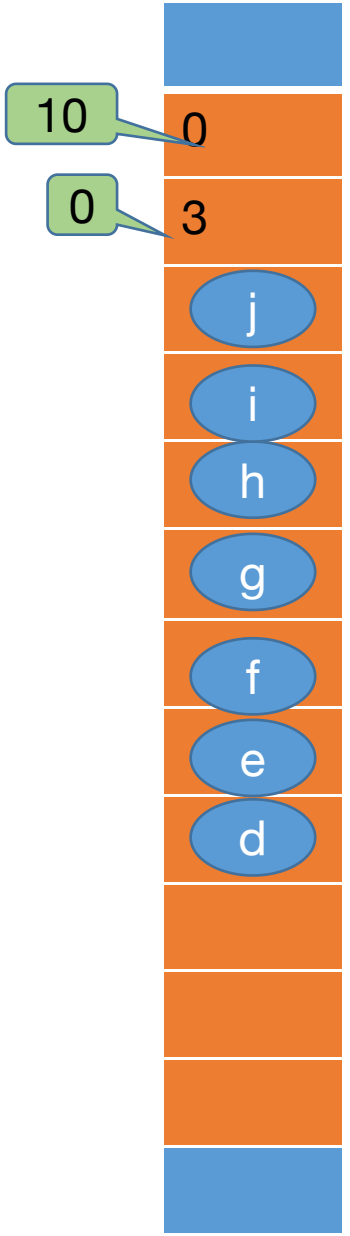
Q.data[Q.rear]=x;//新元素插入队尾
Q.rear=（Q.rear+1）%MaxSize;//队尾指针加I取模



循环队列

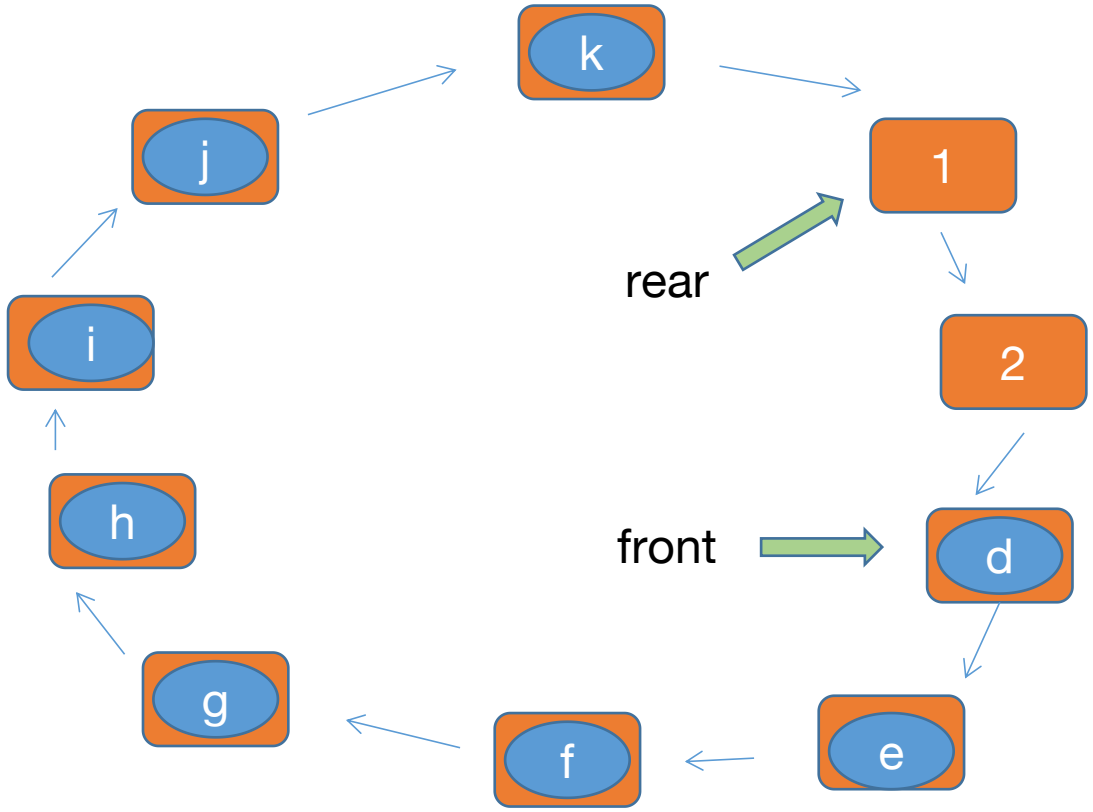
Q.data[Q.rear]=x;//新元素插入队尾
Q.rear=（Q.rear+1）%MaxSize;//队尾指针加I取模





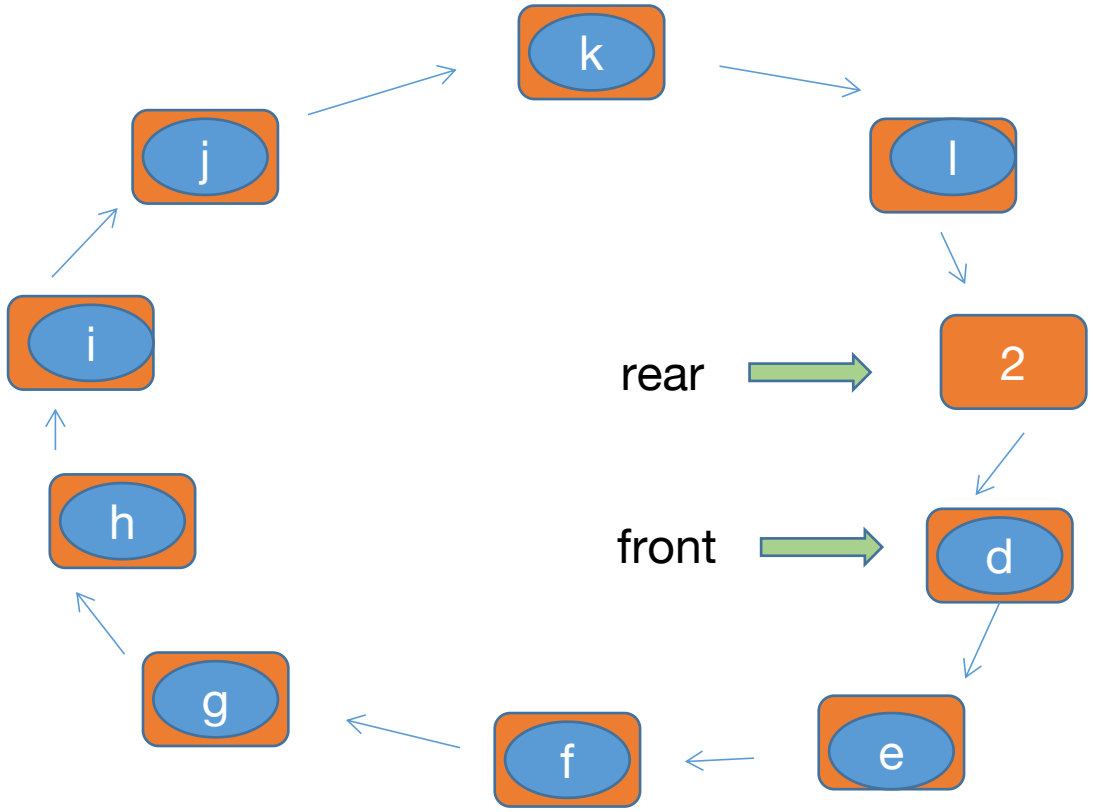
循环队列

Q.data[0.rear]=x;//新元素插入队尾
Q.rear=（Q.rear+1）%MaxSize;//队尾指针加I取模



循环队列

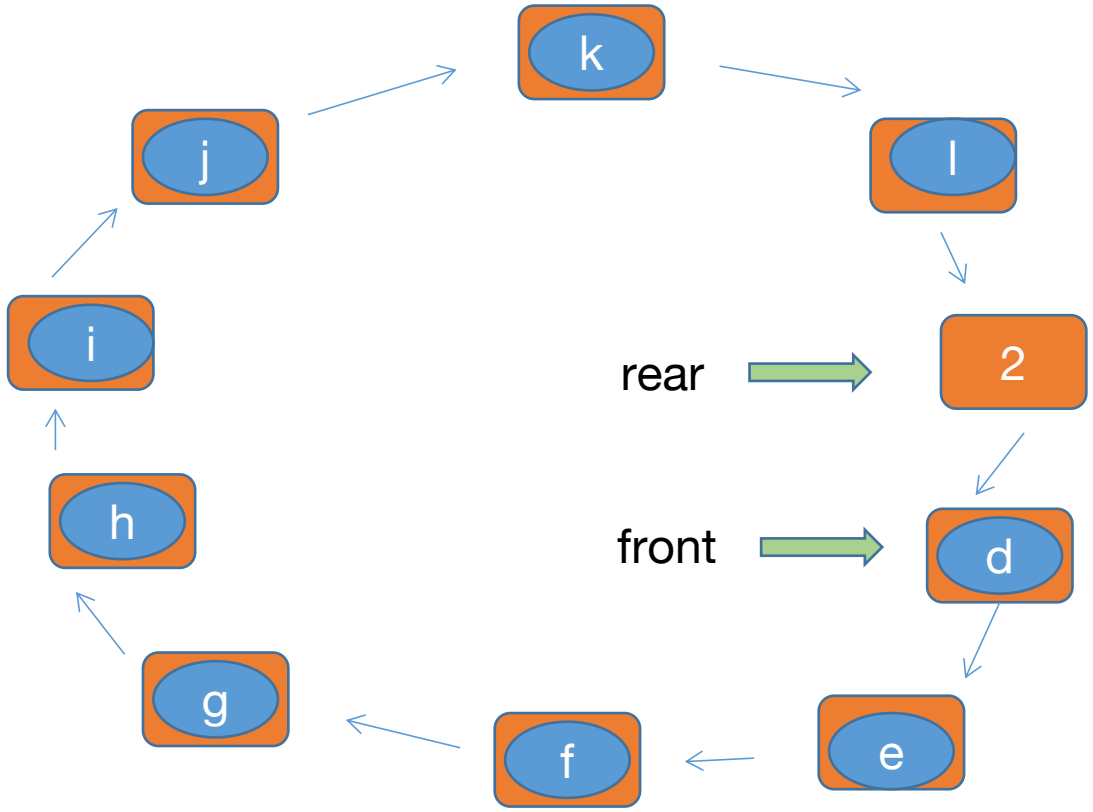
Q.data[0.rear]=x;//新元素插入队尾
Q.rear=（Q.rear+1）%MaxSize;//队尾指针加I取模




```
bool QueueEmpty(SqQueue Q){
    if（Q.rear==Q.front）;
            return true;
    else
            return false;
}
//入队
bool EnQueue(SqQueue &Q, ElemType x){
        if(队满？？)
                return false;
        Q.data[Q.rear]=x;
        Q.rear=(Q.rear+1)%MaxSize;
        return true;
//用模运算将存储空间在逻辑上变成了“环状”
}
```

循环队列

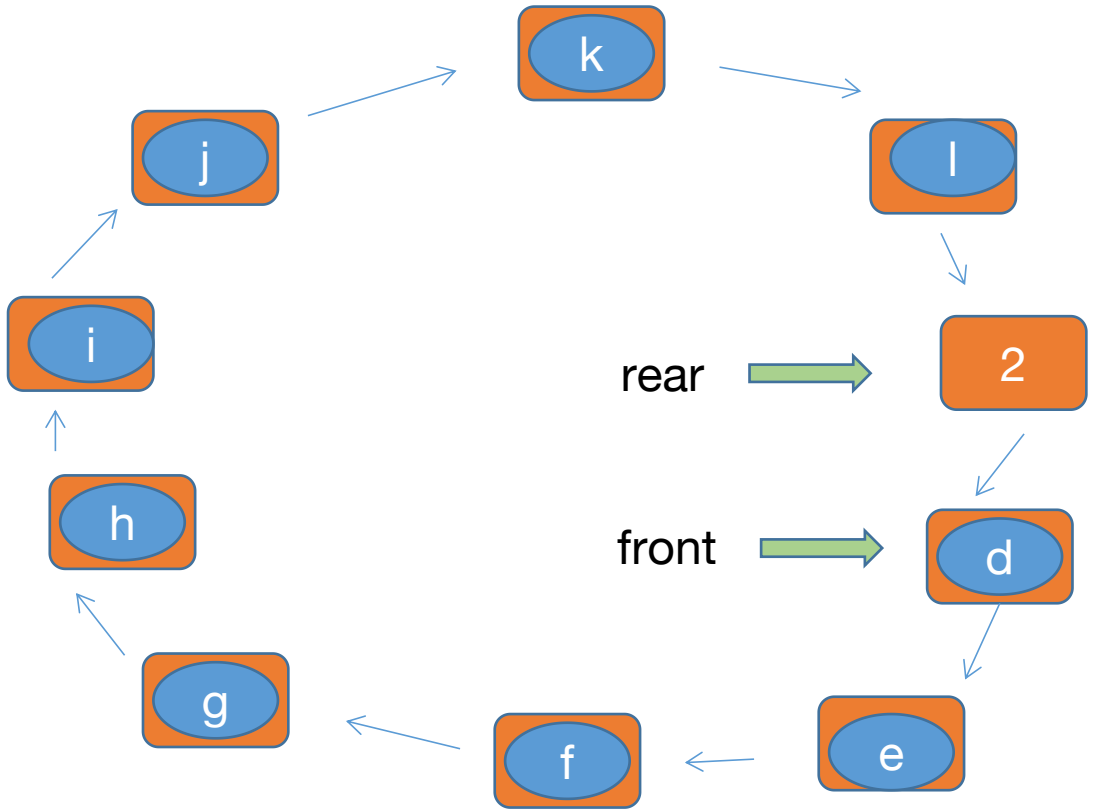
Q.data[0.rear]=x;//新元素插入队尾
Q.rear=（Q.rear+1）%MaxSize;//队尾指针加I取模




```
bool QueueEmpty(SqQueue Q){
    if（Q.rear==Q.front）;
            return true;
    else
            return false;
}
//入队
bool EnQueue(SqQueue &Q, ElemType x){
        if((Q.rear+1)%MaxSize==Q,front)
            return false;
        Q.data[Q.rear]=x;
        Q.rear=(Q.rear+1)%MaxSize;
        return true;
//用模运算将存储空间在逻辑上变成了“环状”
}
```

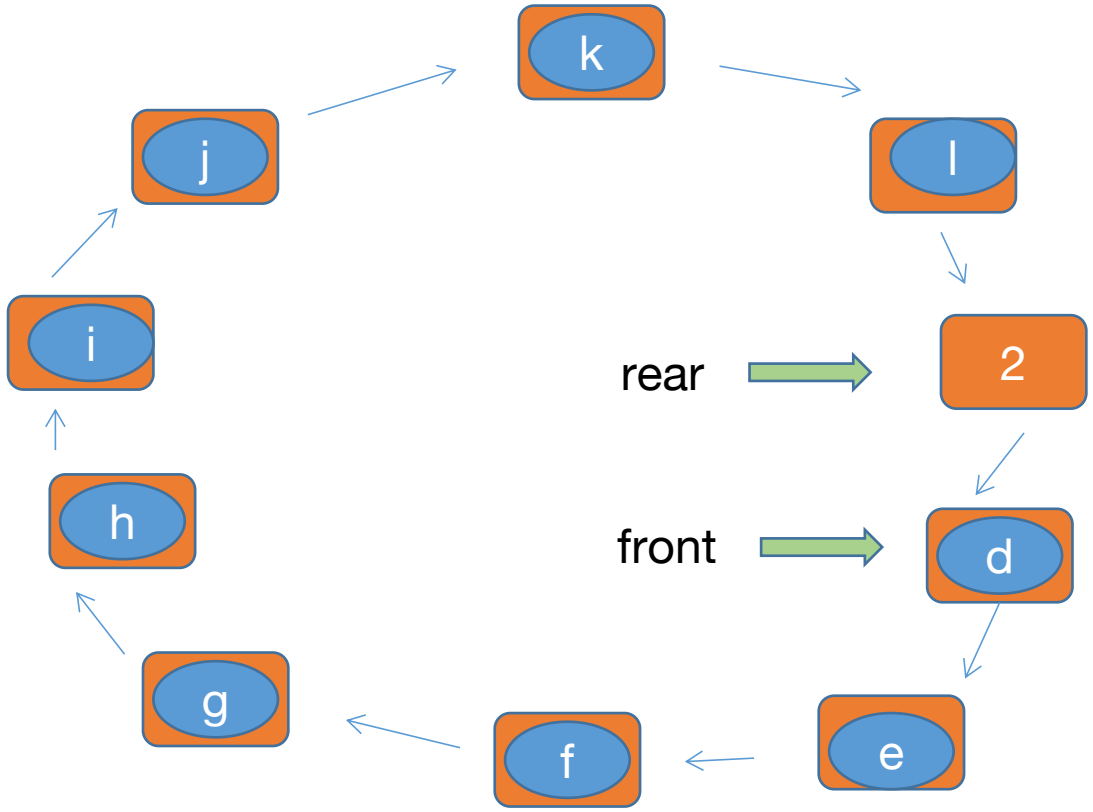
循环队列

——出队



循环队列

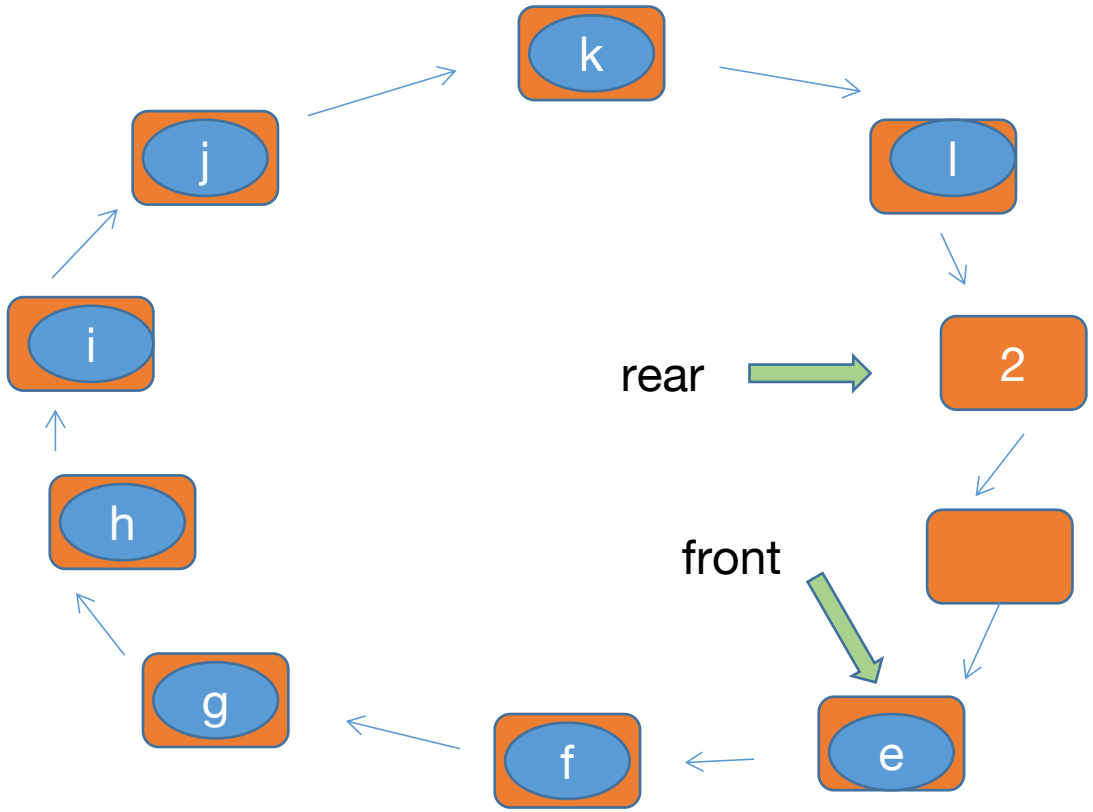
——获得队头元素




```
bool GetHead(SqQueue Q,ElemType&x){
    if(Q.rear==Q.front)
            return false; //队空则报错
     x=Q.data[Q.front];
     return true;
}
```

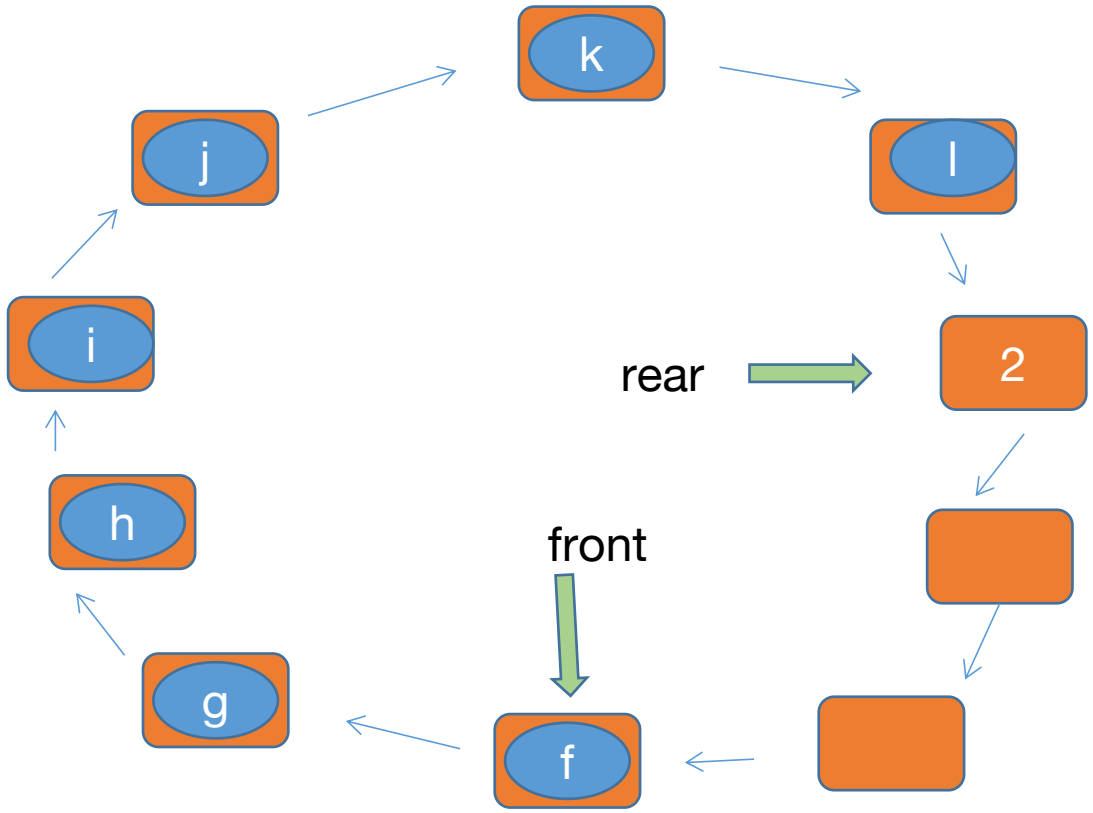
循环队列

——出队



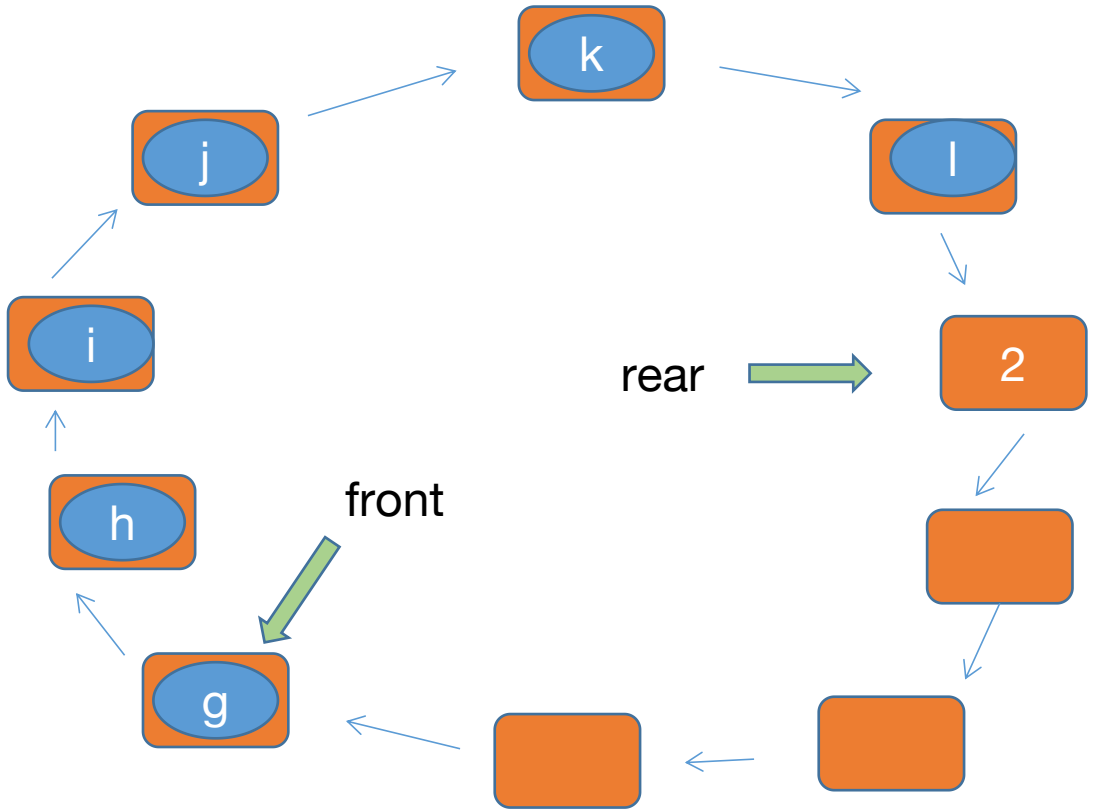
循环队列

——出队



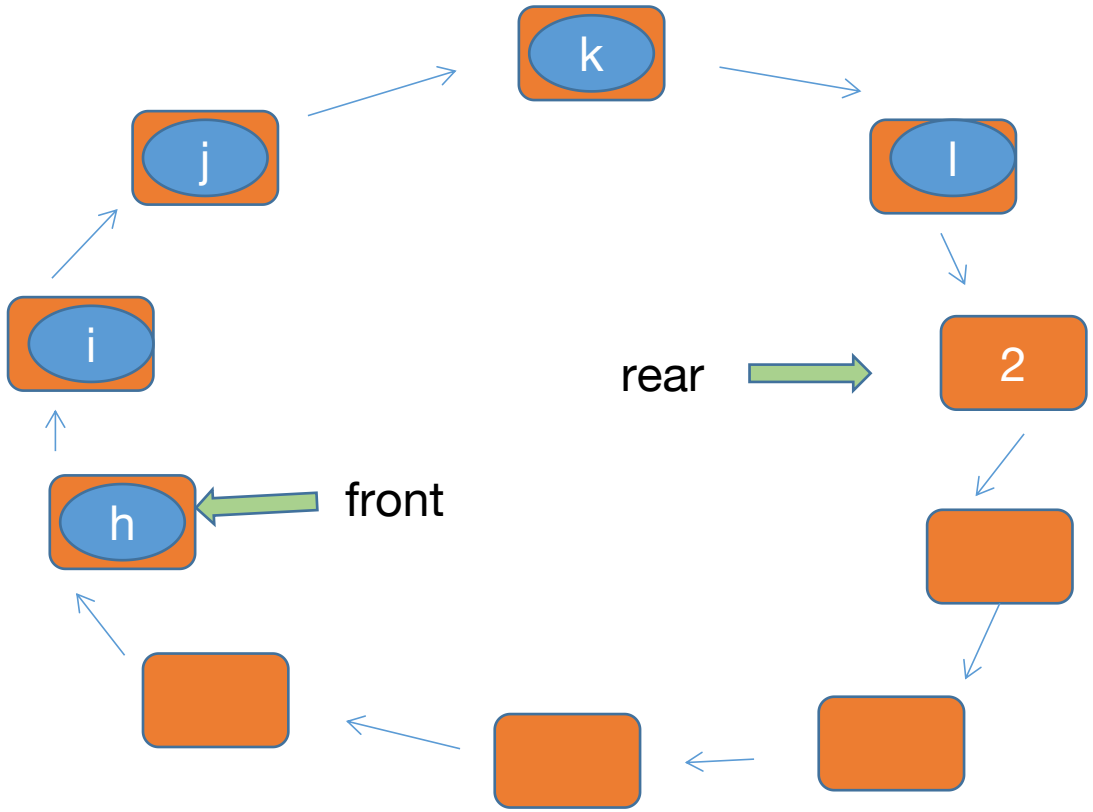
循环队列

——出队



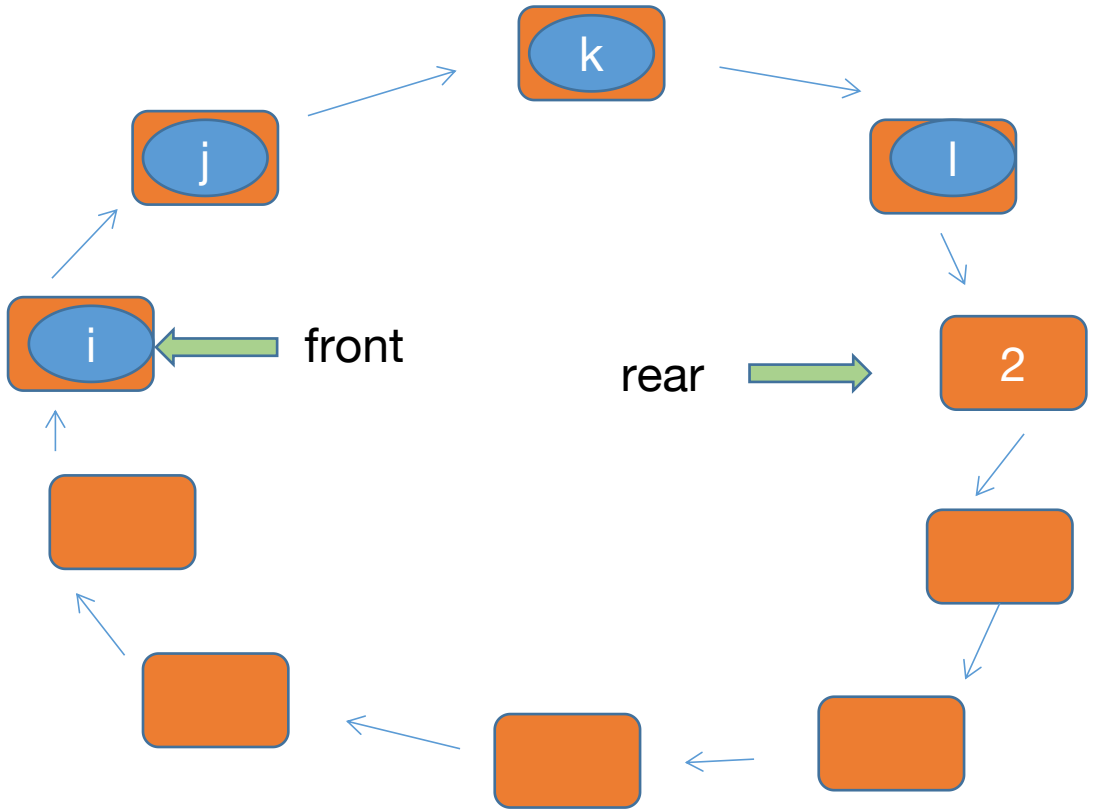
循环队列

——出队



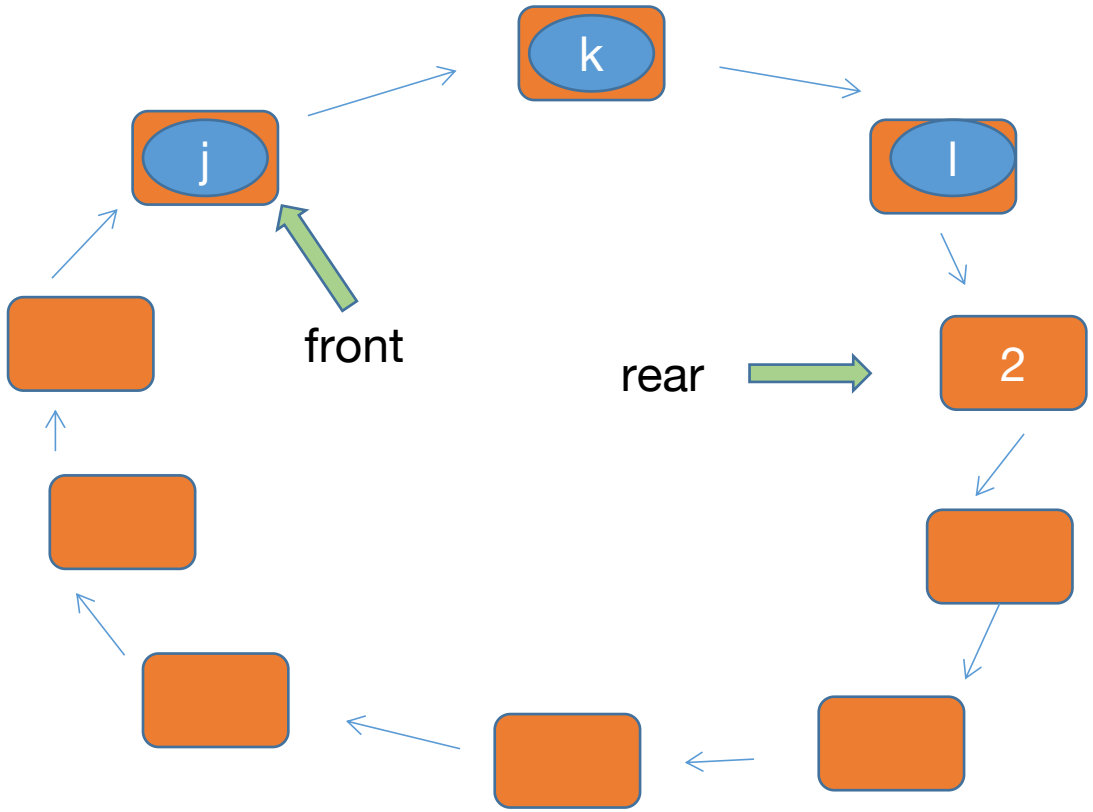
循环队列

——出队



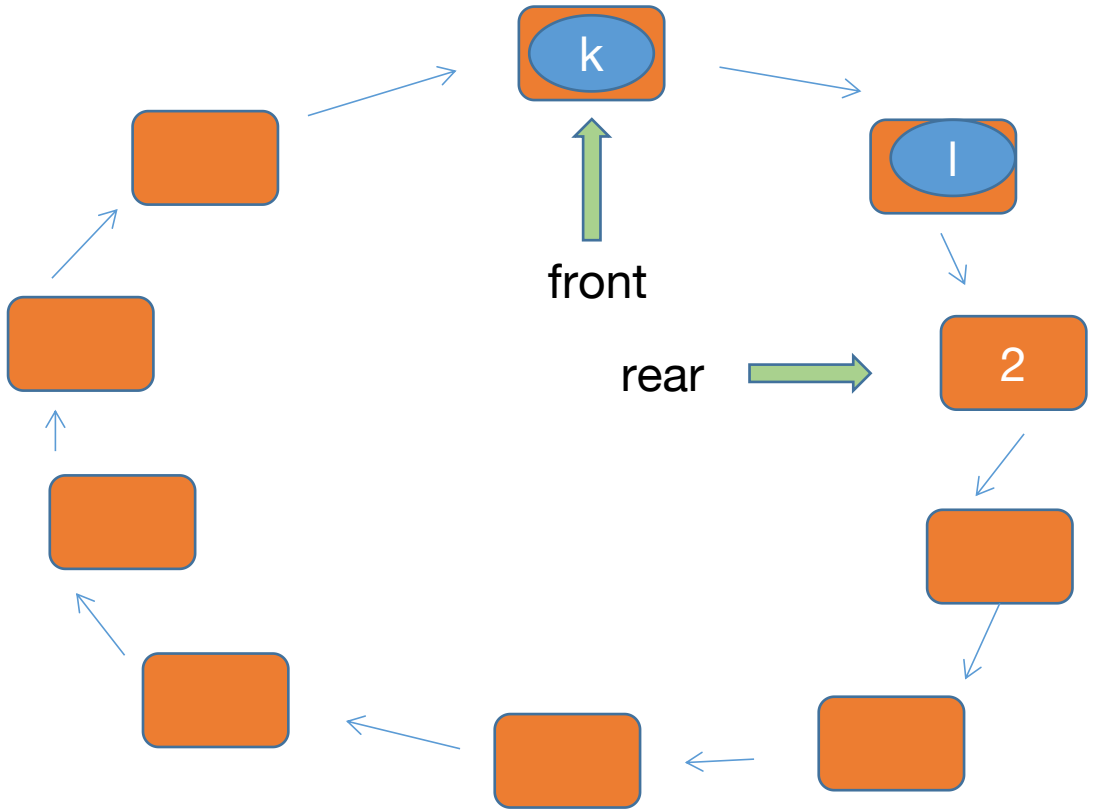
循环队列

——出队



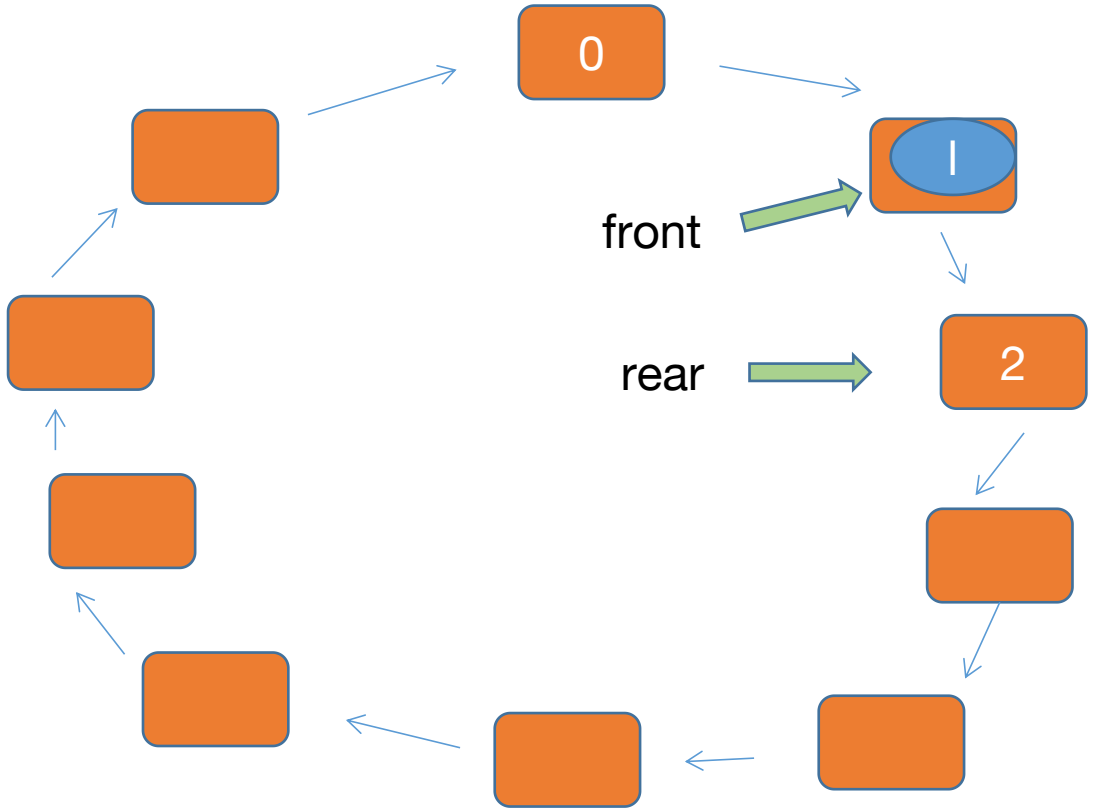
循环队列

——出队



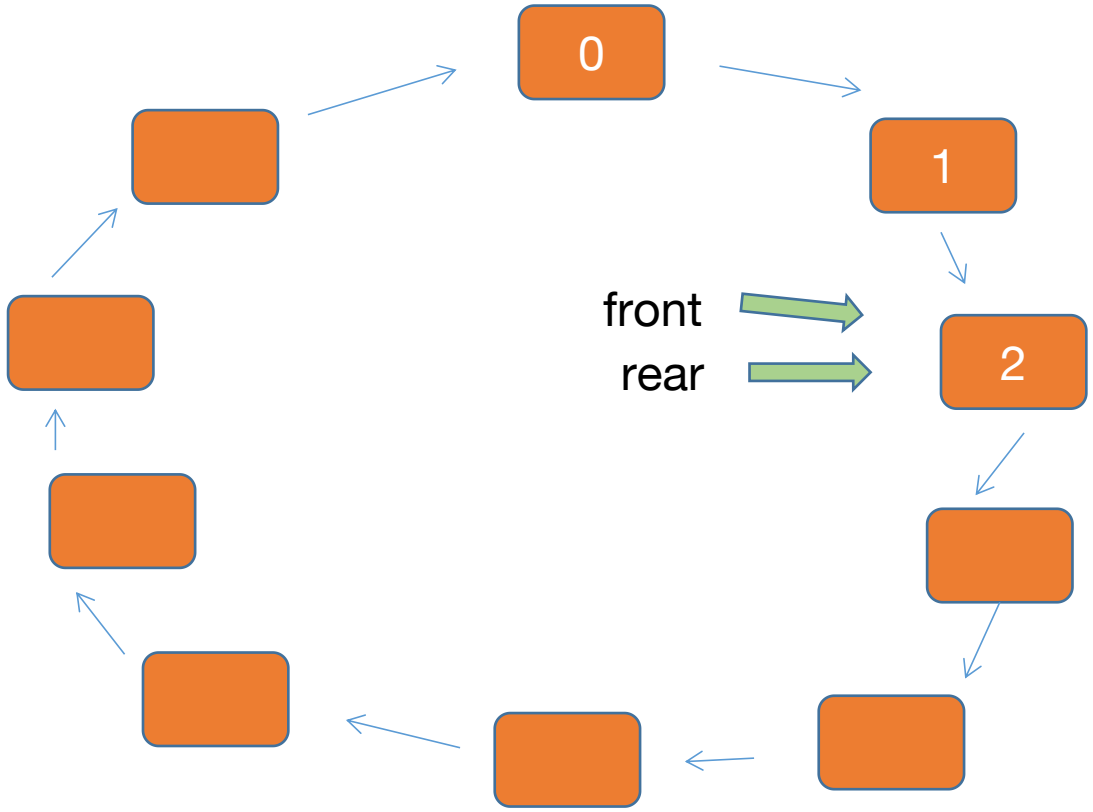
循环队列

——出队



循环队列

——出队



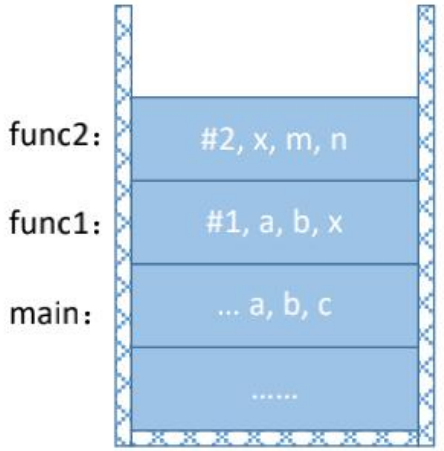

```
//出队（删除一个队头元素，并用x返回）bool
DeQueue(SqQueue &Q,ElemType &x){
        if(Q.rear==Q.front)
            return false;
        x=Q.data[Q.front];
        Q.front=(Q.front+1)%MaxSize;
        return true;
}
```

3.4 栈与递归的实现

```
void main(){
    int a, b, c;
    ...
    funcl(a,b);
#1  c=a+b;
    ...
}
```

```
void funcl(int a, int b){
    int x;
    ...
    func2(x);
#2  x=x+10086;
    ...
}
```

```
void func2(int x){
    int m,n;
    ...
}
```



函数调用的特点：最后被调用的函数最先执行结束（LIFO）

函数调用时，需要用一个栈存储：

① 调用返回地址

② 实参

③ 局部变量

函数调用背后的过程

StackRecursion.cpp

栈在递归中的应用

适合用“递归”算法解决：可以把原始问题转换为属性相同，但规模较小的问题

例1:计算正整数的阶乘 n!

$$\text{factorial}(n)=\begin{cases} n*\text{factorial}(n-1), & n>1 \\ 1, & n=1 \\ 1, & n=0 \end{cases}$$

递归体

递归出口

例2:求斐波那契数列

$$\text{Fib}(n)=\begin{cases} \text{Fib}(n-1)+\text{Fib}(n-2), & n>1 \\ 1, & n=1 \\ 0, & n=0 \end{cases}$$

栈在递归中的应用

例1:递归算法求阶乘

```
//计翼正整数n!
int factorial(int n){
    if(n==0||n==1)
          return 1;
     else
           return n*factorial(n-1);
}
int main(){
   int x;
   x=factorial(10);
   printf("%d",x);
   return 0;
}
```

递归调用时，函数调用栈可称为“递归工作栈”

每进入一层递归，就将递归调用所需信息压入栈顶

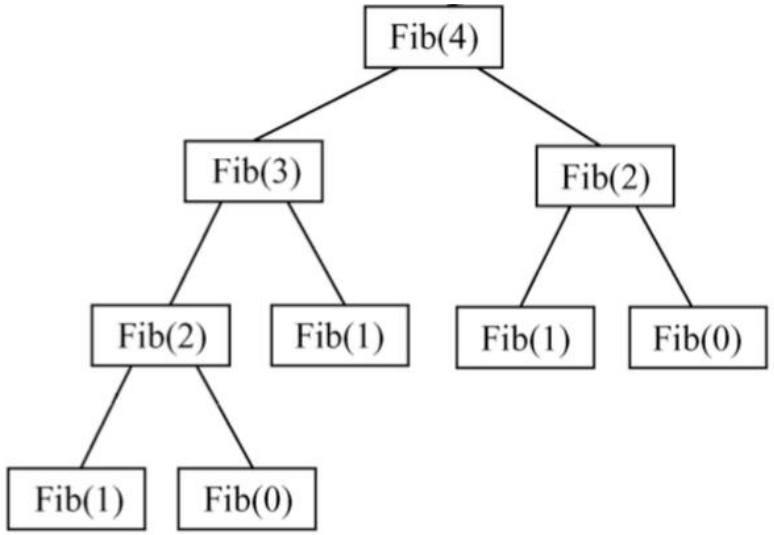
每退出一层递归，就从栈顶弹出相应信息

缺点：太多层递归可能会导致栈溢出

栈在递归中的应用

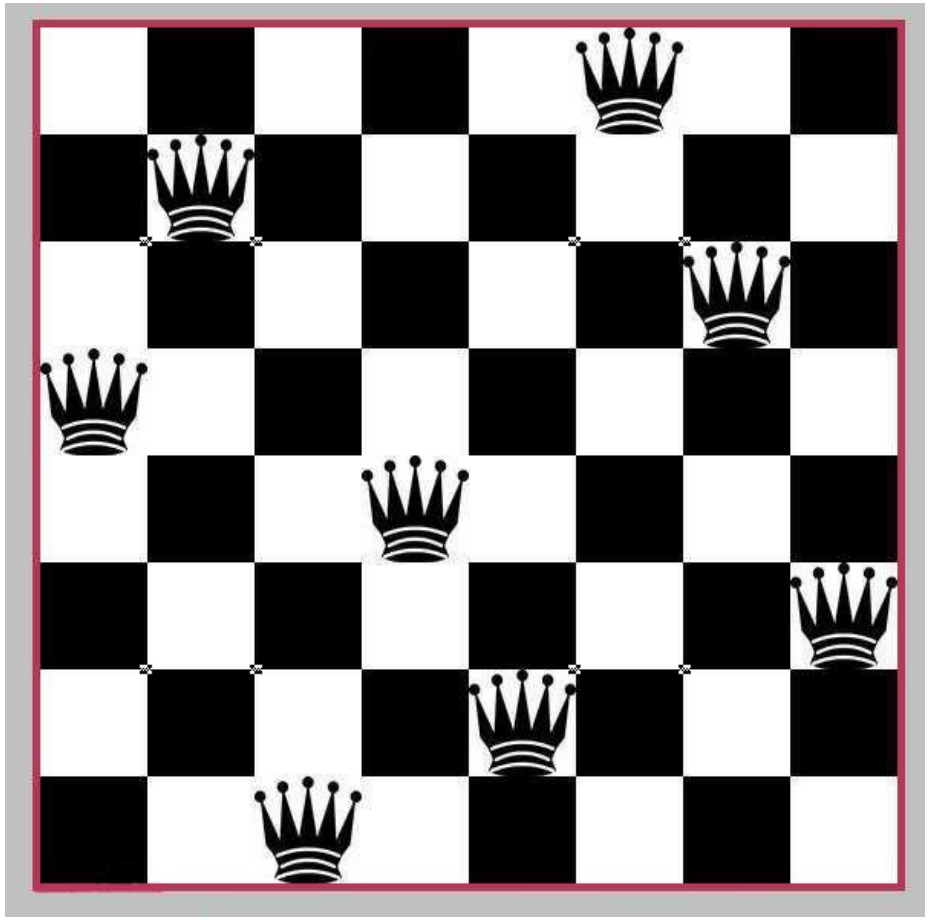
例1:递归求斐波那契数列

```
int Fib(int n){
    if(n==0)
        return 0;
    else if(n==1)
        return 1;
    else
        return Fib(n-1)+Fib(n-2);
}
int main(){
    int x;
    x=Fib(4);
    printf("%d",x);
    return 0;
}
```



缺点：可能包含很多次重复运算

栈的应用举例



栈的应用举例

自然语言处理:

今年十一颐和园游客人数同比下降一成。

今年 十一 颐和园 游客 人数 同比 下降 一成。