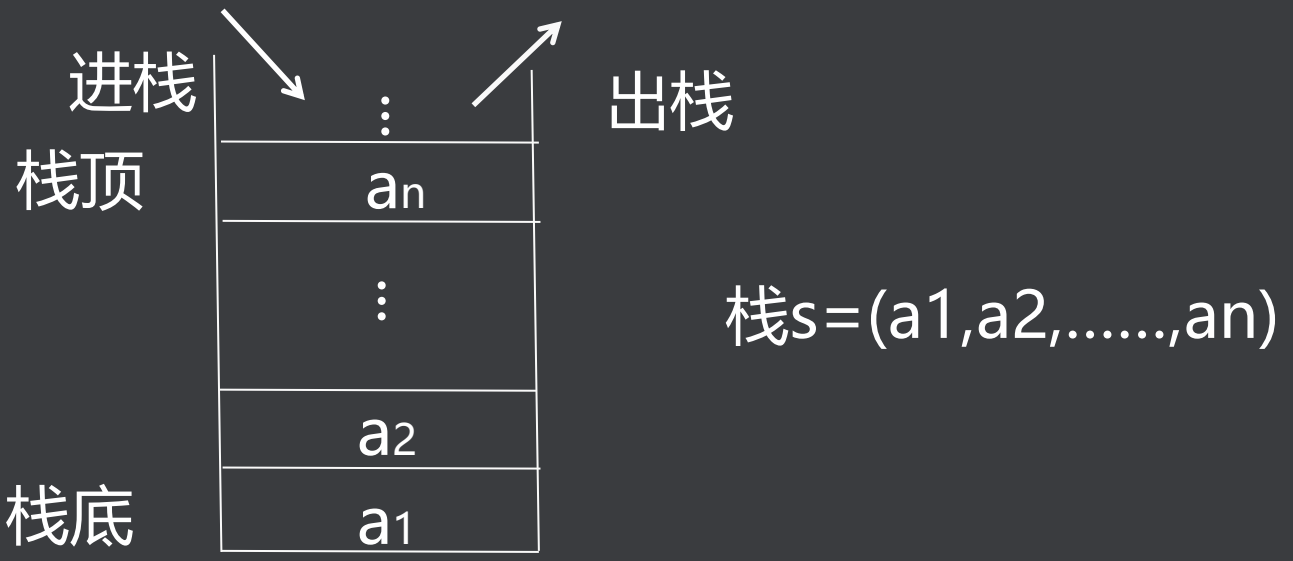
3.1

栈

栈

栈

栈是限制在一端进行插入操作和删除操作的线性表（俗称堆栈），允许进行操作的一端称为“栈顶”，另一固定端称为“栈底”，当栈中没有元素时称为“空栈”。特点：后进先出（LIFO）。



栈s=(a1,a2,.....,an)

栈

基本运算：

创建空栈： CreateStack (len)

清空栈： ClearStack (S)

判断是否栈空：EmptyStack (S)

判断是否栈满：FullStack (S)

元素进栈： PushStack (S, x)

元素出栈： PopStack (S)

取栈顶元素： GetTop (S)

栈

顺序栈：

它是顺序表的一种，具有顺序表同样的存储结构，由数组定义，配合用数组下标表示的栈顶指针top（相对指针）完成各种操作。

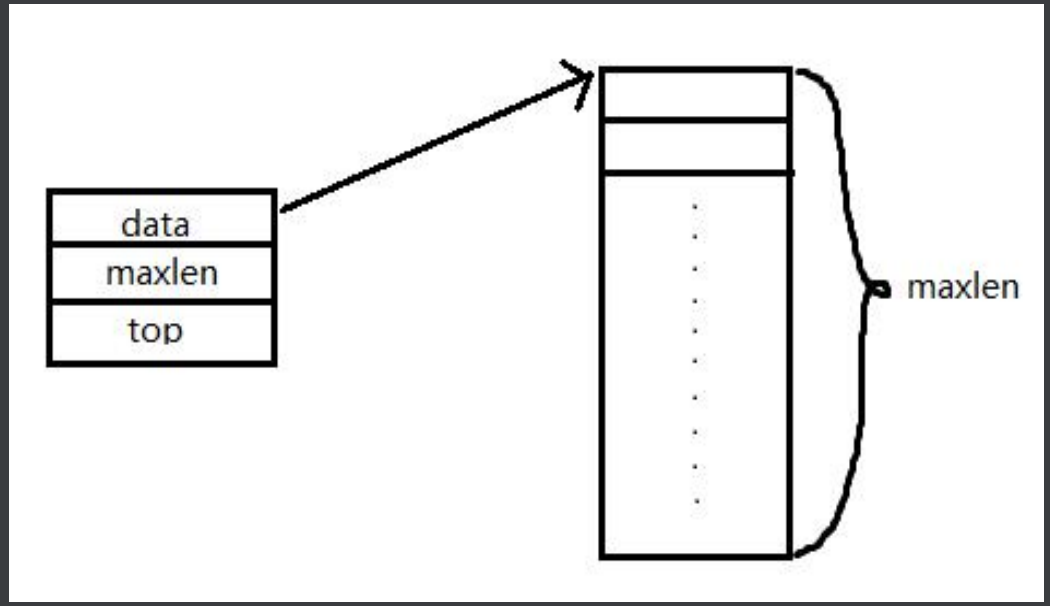
```
typedef  int  data_t ;          /*定义栈中数据元素的数据类型*/
typedef struct
{
      data_t  *data ;           /*用指针指向栈的存储空间*/
      int  maxlen;              /*当前栈的最大元素个数*/
      int  top ;                /*指示栈顶位置(数组下标)的变量*/
 } seqstack_t;                  /*顺序栈类型定义*/
```

栈

创建栈:

```
seqstack_t *CreateStack (int len)
{
        seqstack_t *ss;
        ss = (seqstack_t *)malloc(sizeof(seqstack_t));
        ss->data = (data_t *)malloc(sizeof(data_t) * len);
        ss->top = -1;
        ss->maxlen = len;

        return ss;
}
```



栈

清空栈:

```
ClearStack (seqstack_t  *s)
{
        s-> top = -1 ;
}
```

判断栈是否空：

```
int  EmptyStack (seqstack_t  *s)
{
        return   (s->top ==  -1  ?  1 : 0);
}
```

栈

进栈：

```
void  PushStack (seqstack_t  *s ,  data_t  x)
{     if  (s->top = = N - 1) {
          printf (  “overflow !\n”  ) ;
         return  ;
       }
       else  {
                 s->top ++  ;
                 s->data[s->top] = x ;
        }
        return  ;
}
```

栈

出栈：

```
datatype  PopStack(seqstack_t *s)
{
        s->top--;
    return  (s->data[s->top+1]);
}
```

取栈顶 元素:

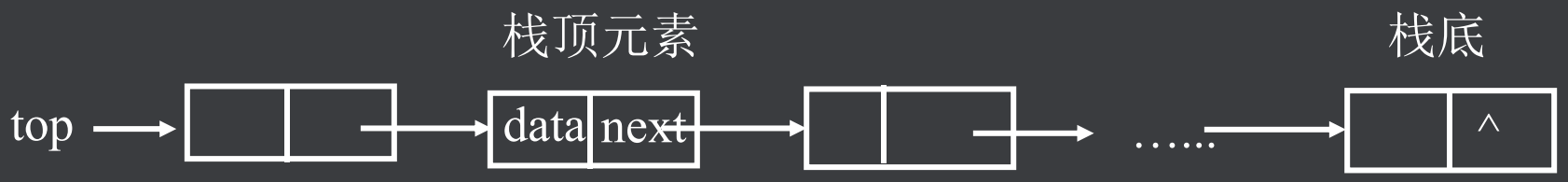
```
datatype  GetTop(seqstack_t  *s)
{
    return (s->data[s->top]);
}
```

栈

链式栈

插入操作和删除操作均在链表头部进行，链表尾部就是栈底，栈顶指针就是头指针。

```
typedef  int  data_t ;                /*定义栈中数据元素数据类型*/
typedef  struct  node_t
{
    data_t data ;              /*数据域*/
    struct  node_t  *next ;            /*链接指针域*/
} linkstack_t ;                /*链栈类型定义*/
```



栈

创建空栈：

```
linkstack_t  *CreateLinkstack() {
    linkstack_t  *top;
    top  =  (linkstack_t  *)malloc(sizeof(linkstack_t));
    top->next = NULL;
    return  top;
}
```

判断是否空栈：

```
int  EmptyStack (linkstack_t *top)
{
     return  (top->next  == NULL  ?  1  :  0);
}
```

栈

入栈：

```
void   PushStack(linkstack_t *top,  data_t  x)
{
      linkstack_t  *p ;            /*定义辅助指针*/
      p = (linkstack_t *)malloc ( sizeof (linkstack_t) ) ; /*指向新结点*/
      p->data = x ;   /*将数据存入新结点的数据域中*/
      p->next = top->next;
      top->next  =  p;     /*新结点插入原栈顶之前*/

      return;
}
```

栈

栈的应用举例

栈在表达式计算过程中的应用： 建立操作数栈和运算符栈。运算符有优先级。

规则：

- 自左至右扫描表达式，凡是遇到操作数一律进操作数栈。
- 当遇到运算符时，如果它的优先级比运算符栈栈顶元素的优先级高就进栈。反之，取出栈顶运算符和操作数栈栈顶的连续两个操作数进行运算，并将结果存入操作数栈，然后继续比较该运算符与栈顶运算符的优先级。
- 左括号一律进运算符栈，右括号一律不进运算符栈，取出运算符栈顶运算符和操作数栈顶的两个操作数进行运算，并将结果压入操作数栈，直到取出左括号为止。

栈

例如： 计算 4 + 8×2 - 3 ;

操作数栈： 4 8 2 | 4 16 |20 |20 3 |17

运算符栈： + × | + | | - |

例如： 计算 (4 + 8)×2 - 3 ;

操作数栈： 4 8 | 12 2 |24 3 |21

运算符栈： (+ |× | - |

3.2

队列

队列

队列

队列是限制在两端进行插入操作和删除操作的线性表，允许进行存入操作的一端称为“队尾”，允许进行删除操作的一端称为“队头”。当线性表中没有元素时，称为“空队”。

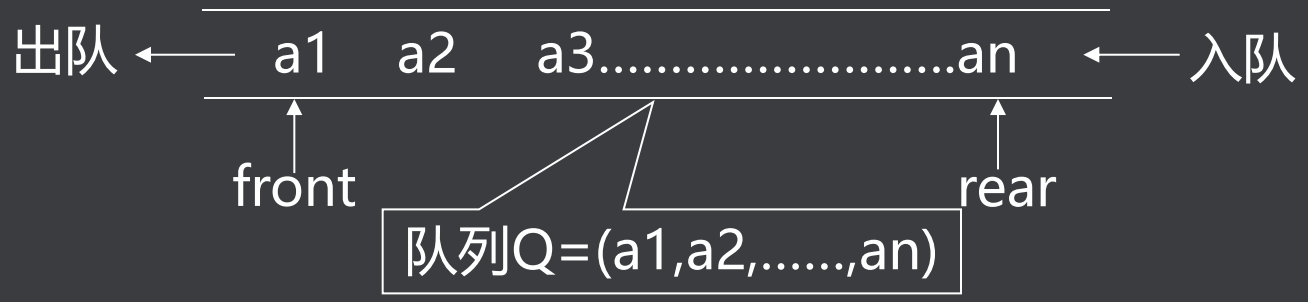
特点：先进先出（FIFO）。

基本操作：

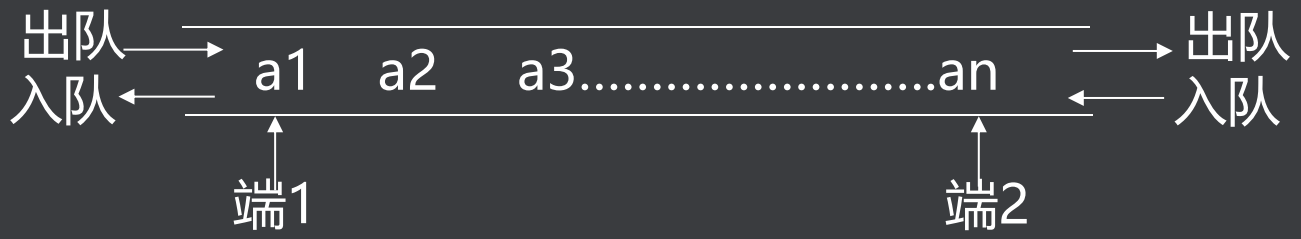
- 创建队列：CreateQueue ()
- 清空队列：ClearQueue (Q)
- 判断队列空：EmptyQueue(Q)
- 判断队列满：FullQueue(Q)
- 入队：EnQueue (Q , x)
- 出队：DeQueue(Q)

队列

图示：


出队
a1 a2 a3.....an
入队
front
rear
队列Q=(a1,a2,.....,an)

双端队列


出队
入队
a1 a2 a3.....an
出队
入队
端1
端2

队列

规定一： front指向队头元素的前一个位置; rear指向队尾元素所在位置。

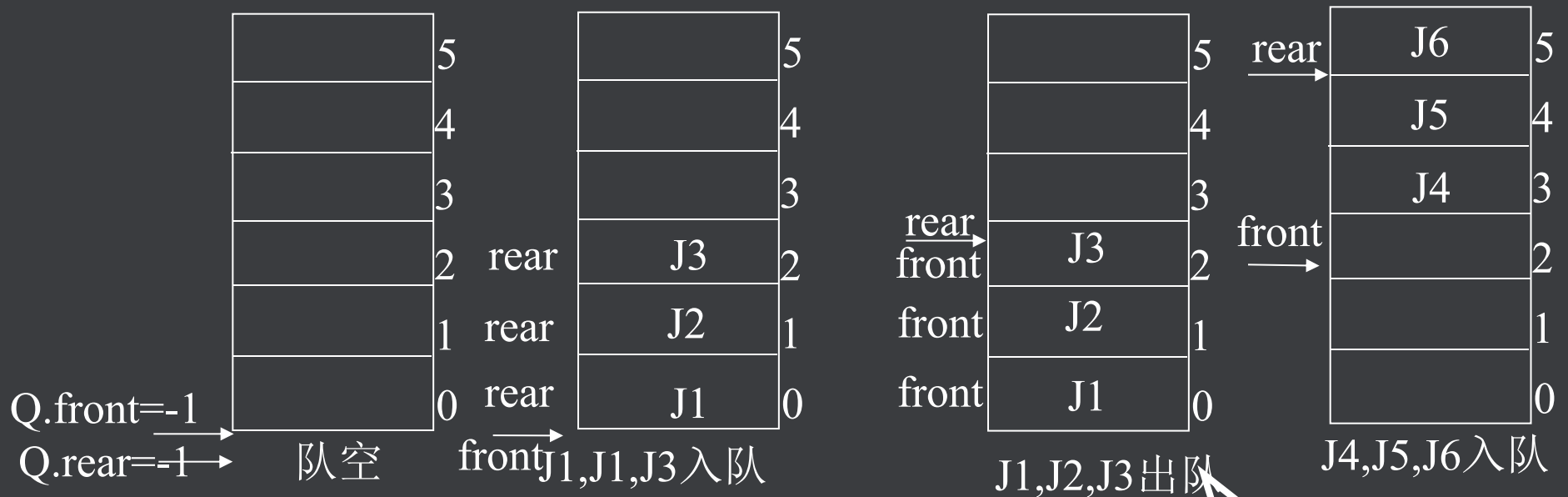
规定二：front指向队头元素的位置; rear指向队尾元素的下一个位置。

在队列操作过程中，为了提高效率，以调整指针代替队列元素的移动，并将数组作为循环队列的操作空间。

为区别空队和满队，满队元素个数比数组元素个数少一个。

队列

循环队列



设两个指针front,rear,约定：
Q.rear指示队尾元素；
Q.front指示队头元素前一个位置；
初值Q.front=Q.rear=-1

空队列条件： front==rear
入队列： sq[rear++]=x;
出队列： x=sq[front++];

队列

顺序队列：

它是顺序表的一种，具有顺序表同样的储存结构，由数组定义，配合用数组下标表示的队头指针和队尾指针完成各种操作。

```
typedef  int  data_t ;              /*定义队列中数据元素的数据类型*/
#define  N  64                  /*定义队列的容量*/
typedef  struct
{
    data_t          data[N] ;       /*用数组作为队列的储存空间*/
     int     front,rear ;     /*指示队头位置和队尾位置的指针*/
} sequeue_t ;                   /*顺序队列类型定义*/
```

队列

创建空队列：

```
sequeue_t  *CreateQueue ()
{
      sequeue_t  *sq = (sequeue_t *)malloc(sizeof(sequeue_t));
      sq->front = sq->rear = maxsize -1;
      return  sq;
}
```

判断队列空：

```
int   EmptyQueue (sequeue_t *sq)  {
      return (sq->front  = =  sq->rear) ;
}
```

队列

入队：将新数据元素x插入到队列的尾部。

```
void   EnQueue (sequeue_t *sq , data_t x)
{
      sq->data[sq->rear+1] = x ;
      sq->rear = (sq->rear + 1) % N ;
      return ;
}
```