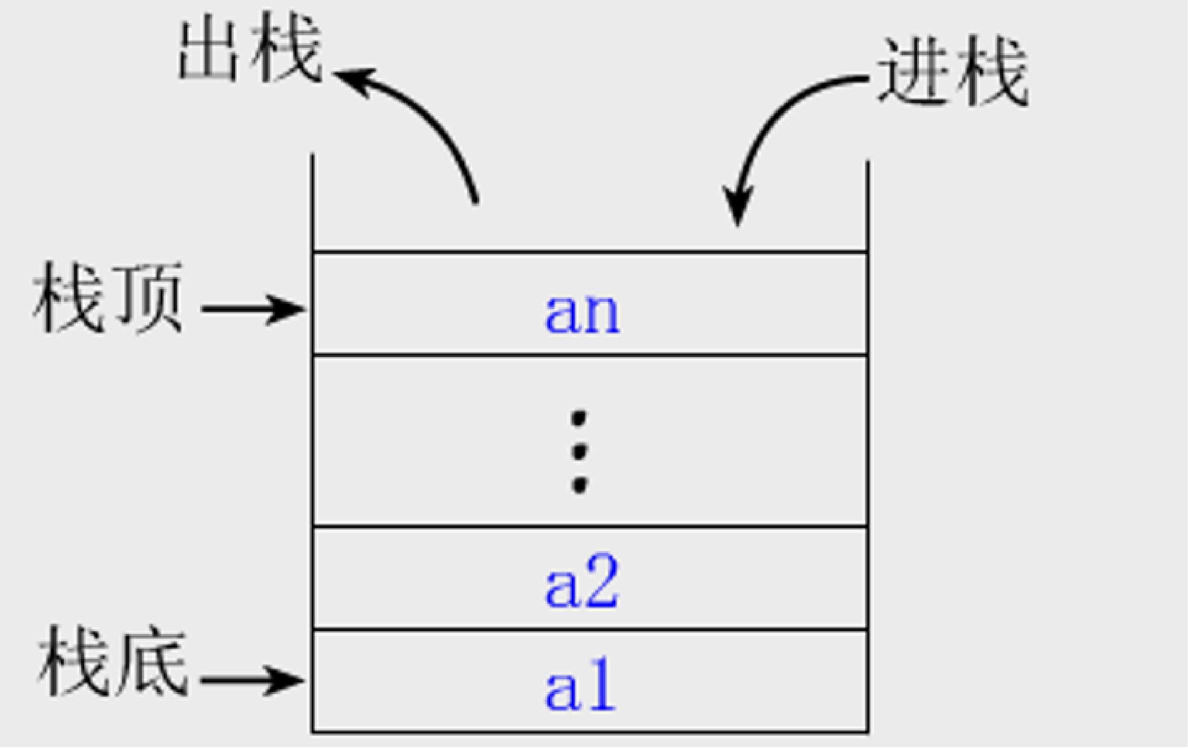
第三章 栈和队列

3.1 栈

3.1.1 栈的基本概念

> 栈是特殊的线性表：**只允许在一端进行插入或删除操作**，其逻辑结构与普通线性表相同。

1. 栈顶：允许进行插入和删除的一端 （最上面的为栈顶元素）。
2. 栈底：不允许进行插入和删除的一端 （最下面的为栈底元素）。
3. 空栈：不含任何元素的空表。
4. 特点：**后进先出**（后进栈的元素先出栈）、LIFO（Last In First Out）。
5. 缺点：**栈的大小不可变**，解决方法：共享栈。



> 栈的数学性质：n个不同元素进栈，出栈元素不同排列的个数是
>
> $$\frac{1}{n+1}C_{2n}^{n}$$

3.1.2. 栈的基本操作

1. `InitStack(&S)`：初始化栈。构造一个空栈 S，分配内存空间。
2. `DestroyStack(&S)`：销毁栈。销毁并释放栈 S 所占用的内存空间。
3. `Push(&S, x)`：进栈。若栈 S 未满，则将 x 加入使其成为新的栈顶元素。
4. `Pop(&S, &x)`：出栈。若栈 S 非空，则弹出（删除）栈顶元素，并用 x 返回。
5. `GetTop(S, &x)`：读取栈顶元素。若栈 S 非空，则用 x 返回栈顶元素。
6. `StackEmpty(S)`：判空。断一个栈 S 是否为空，若 S 为空，则返回 true，否则返回 false。

3.1.3 栈的顺序存储（顺序栈）

1、顺序栈的实现

```
typedef struct {
    int data[MaxSize];  // 存放栈中元素
    int top;            // 栈顶指针，记录栈顶坐标
}SqStack;
```

> 栈顶指针：S.top，初始时，设置S.top = -1（有的教材中会设置为0，规定top指针指向的是栈顶元素的下一存储单元）
>
> 进栈操作：栈不满时，栈顶指针先加1，在赋值给栈顶元素
>
> 出栈操作：栈非空时，先取栈顶元素值，再将栈顶指针减1
>
> 栈空条件：S.top == -1
>
> 栈满条件：S.top == MaxSize - 1

2、顺序栈的初始化

```
bool InitStack(SqStack &S) {
    S.top = -1;
    return true;
}
```

3、判断栈是否为空

```
bool StackEmpty(SqStack S) {
    if (S.top == -1) {
        return true;
    } else {
        return false;
    }
}
```

4、进栈

```
bool Push(SqStack &S, int x) {
    if (S.top == MaxSize - 1) {     // 栈满，报错
        cout << "栈满" << endl;
        return false;
    }
    S.data[++S.top] = x;     // 指针先加1，再进栈
    return true;
}
```

5、出栈

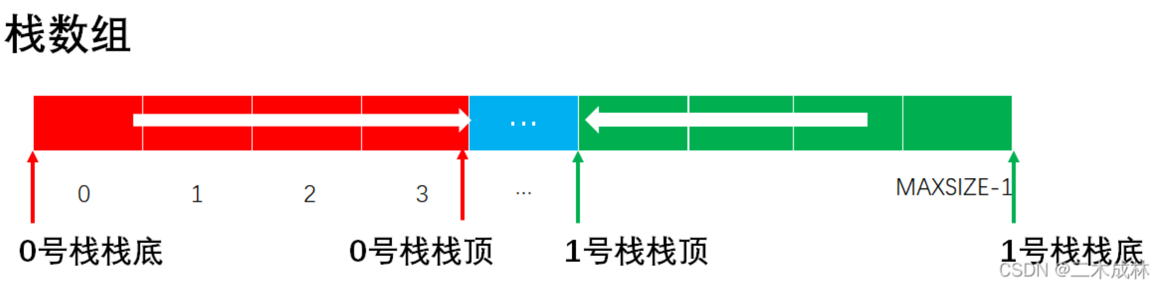
```
// 出栈
bool Pop(SqStack &S, int &x) {
    if (S.top == -1) {
        cout << "当前栈空，无法出栈" << endl;
        return false;
    }
    x = S.data[S.top--];    // 先让x记录栈顶元素，再让栈顶指针减1
    return true;
}
```

6、读取栈顶元素

```
int GetTop(SqStack S) {
    if (S.top == -1) {
        cout << "当前栈为空" << endl;
        return NULL;
    }
    return S.data[S.top];
}
```

3.1.4 共享栈

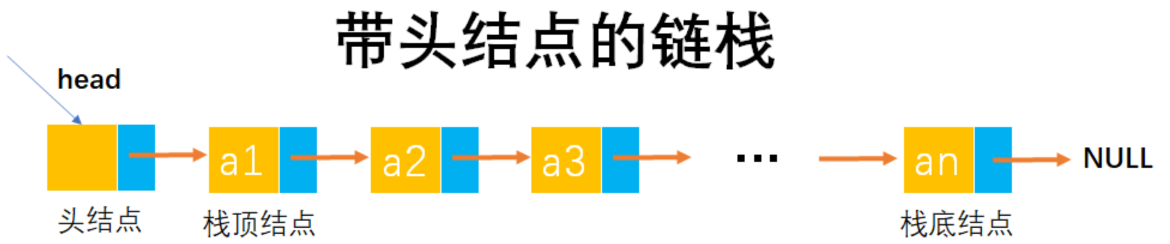
让两个顺序栈共享一个一维数组空间，将两个栈的栈底分别设置在共享空间的两端，两个栈顶同时向共享空间的中间延伸。

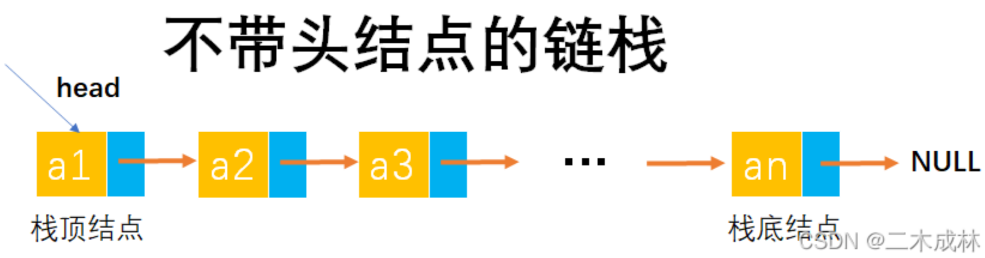



```
#define MaxSize 10         //定义栈中元素的最大个数
typedef struct{
    ElemType data[MaxSize];       //静态数组存放栈中元素
    int top0;                     //0号栈栈顶指针
    int top1;                     //1号栈栈顶指针
}ShStack;

// 初始化栈
void InitSqStack(ShStack &S){
    S.top0 = -1;
    S.top1 = MaxSize;
}
```

3.1.5 栈的链式存储（链栈）





采用链式存储的栈称为链栈。链栈的优点是便于多个栈共享存储空间和提高效率，且不存在栈满上溢的清空。通常采用单链表实现，并且规定所有操作都是在单链表的表头进行上的（因为头结点的 `next` 指针指向栈的栈顶结点）。

1、链栈的定义

其结构定义如下：

```
typedef struct LinkNode {
    int data;
    struct LinkNode *next;
} *LiStack;
```

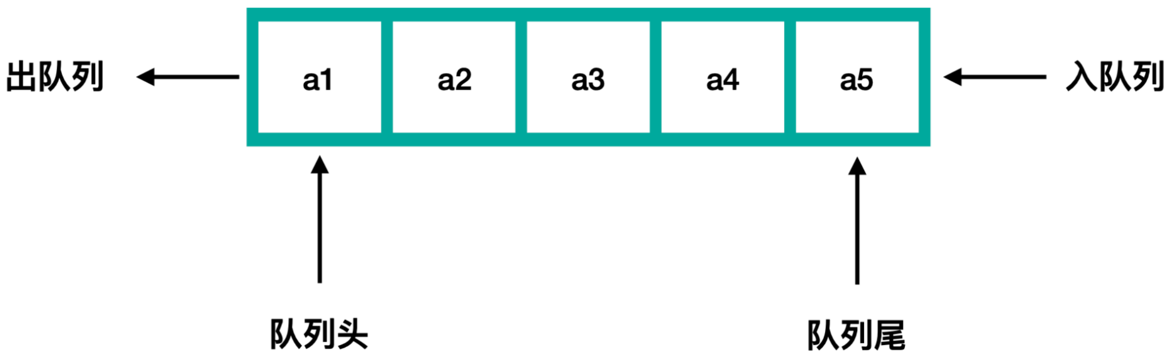
2、链栈的初始化

```
bool InitStack(LiStack &L) {
    L = (LinkNode *) malloc(sizeof(LinkNode));
    if(L == NULL){
        return false;
    }
    L->next = NULL;
    return true;
}
```

3.2 队列

3.2.1 队列的基本概念

> 队列是操作受限的线性表：只允许在一端进行插入 (入队)，另一端进行删除 (出队)。



队列的特性：**先进先出（FIFO, First In First Out)**

3.2.2 队列的顺序存储

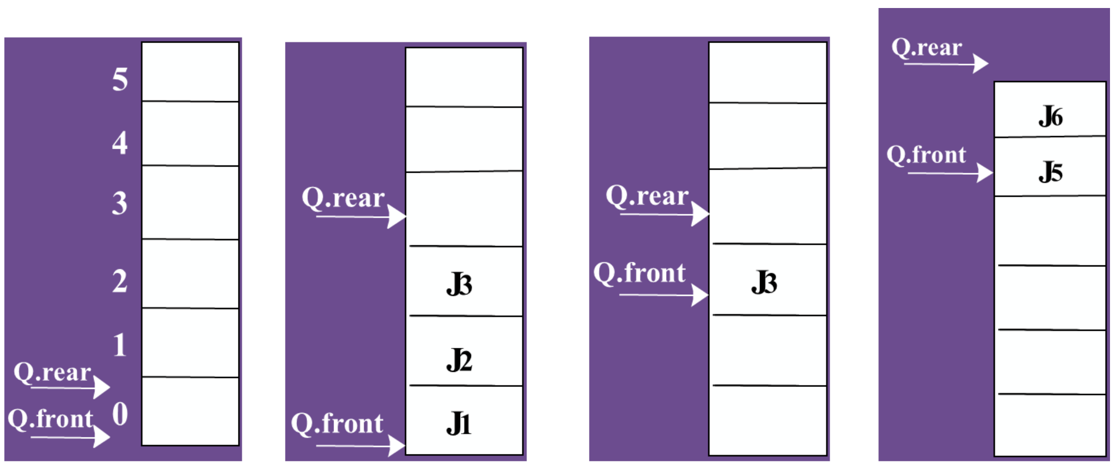
队列的顺序实现是指分配一块**连续的存储单元**存放队列中的元素，并附设两个指针：

> 队头指针front指向队头元素，
>
> 队尾指针rear指向队尾元素的下一个位置。

其代码定义如下：

```
typedef struct {
    int data[MaxSize];
    int front, rear;
}SqQueue;
```

其基本操作的文字描述如下：

> 初始状态：Q.front == Q.rear == 0
>
> 进队操作：队列不满时，先将值送到队尾，再将队尾指针加1
>
> 出队操作：队不空时，先取队头元素值，再将对头指针加1
>
> 
>

> 值得注意的是，Q.rear == MaxSize 不能作为队列满的条件，如上图右1所示，此时Q.rear已经等于MaxSize了，但是队列并没有满。data数组中仍然有能存放元素的其他位置，这是一种假溢出。

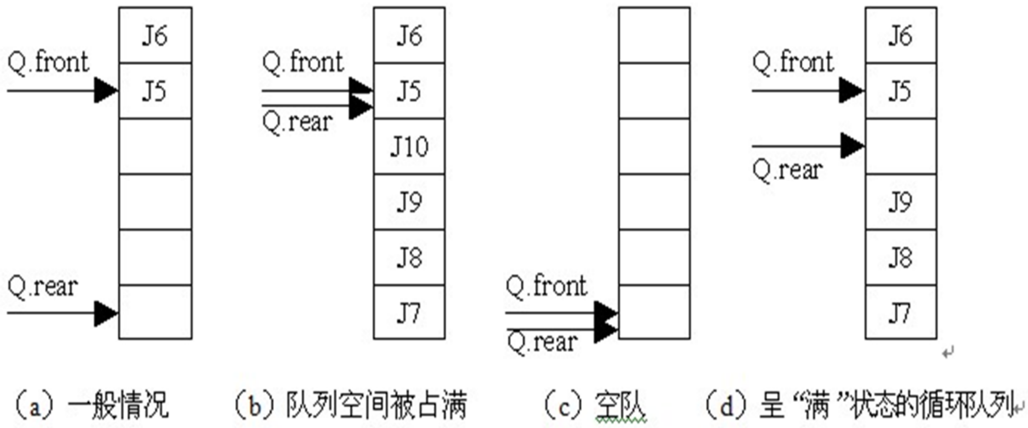


图 3.12 循环队列中头、尾指针和元素之间的关系

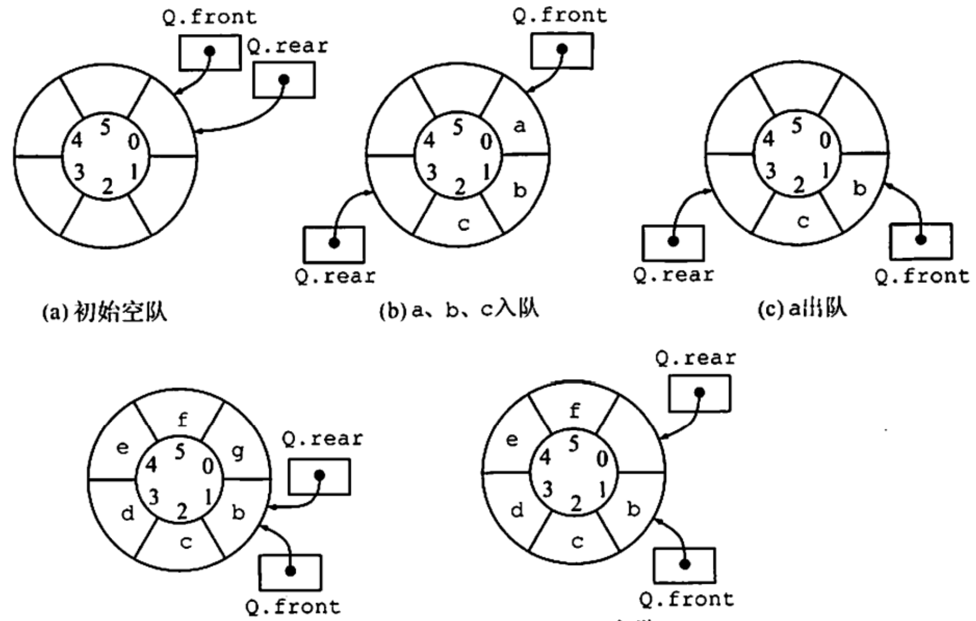
```
// 初始化顺序队
bool InitQueue(SqQueue &Q) {
    Q.front = Q.rear = 0;
}
```

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

3.2.3 循环队列

为了解决上述问题，提出了循环队列的概念。将顺序队列臆造为一个环状的空间，即把存储队列元素的表**从逻辑上视为一个环**，称为循环队列。当队首指针Q.front=MaxSize-1后，再前进一个位置就自动到0，这可以利用除法取余运算(%）来实现。

> 初始时：Q.front = Q.rear = 0
>
> 队首指针进1：Q.front = (Q.front + 1) % MaxSize
>
> 队尾指针进1：Q.rear = (Q.rear + 1) % MaxSize
>
> 队列长度：(Q.rear + MaxSize - Q.front) % MaxSize
>
> 出队入队时：指针都往顺时针方向进1



(a) 初始空队 (b) a、b、c入队 (c) a出队

(d1) d、e、f、g入队（无法判断队满还是队空） (d2) d、e、f入队（牺牲一个存储单元）

按照上述情况进行设计，队空和队满的条件都是：$Q.front == Q.rear$，这种情况下无法区分队空队满。

为了区分队空队满的情况，有以下三种处理方式：

（1）牺牲一个存储单元来区分队空或队满（或者增加辅助变量），这是一种普遍的方式，约定：队头指针在队尾指针的下一位置作为队满的标志，如上图d2所示。

> 此时：
>
> 队满条件：$(Q.rear + 1)\&MaxSize == Q.front$
>
> 队空条件：$Q.front == Q.rear$
>
> 队列中元素的个数：$(Q.rear - Q.front + MaxSize)$

（2）类型中增设表示元素个数的数据成员。这样，队空的条件就位Q.size == 0，队满的条件就是Q.size == MaxSize。这两种情况下都有$Q.front == Q.rear$.

```
typedef struct {
    int data[MaxSize];
    int front, rear;
    int size;
} SqQueue;
```

（3）类型中增设tag数据成员，用来区分是队满还是队空。tag等于0时，若因删除导致$Q.front == Q.rear$，则为队空。tag等于1时，若因插入导致$Q.front == Q.rear$，则为队满。

1、入队（循环队列）

```cpp
// 将x入队
bool EnQueue(SqQueue &Q, int x) {
    if ((Q.rear + 1) % MaxSize == Q.front) {    // 队满
        cout << "队满，无法插入" << endl;
        return false;
    }
    Q.data[Q.rear] = x;
    Q.rear = (Q.rear + 1) % MaxSize;
    return true;
}
```

2、出队（循环队列）

```cpp
// 出队，并将出队元素存储到x中
bool DeQueue(SqQueue &Q, int &x) {
    if (Q.rear == Q.front) {
        cout << "队空，无法出队" << endl;
        return false;
    }
    x = Q.data[Q.front];
    Q.front = (Q.front + 1) % MaxSize;
    return true;
}
```

3.2.4 队列的链式存储（链队）

队列的链式表示称为链队列，它实际上是一个同时带有队头指针和队尾指针的单链表。

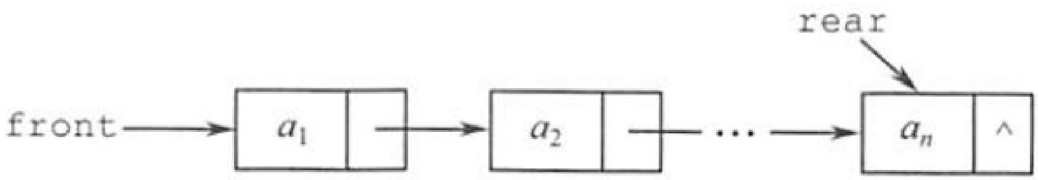


图 3.8　不带头结点的链式队列

> 当$Q.front == NULL$且$Q.rear == NULL$时，链队为空。

```cpp
typedef struct LinkNode {  // 链队结点
    int data;
    struct LinkNode *next;
} LinkNode;

typedef struct {              // 链式队列
    LinkNode *front, *rear;   // 头尾指针
} LinkQueue;
```

不带头结点的链队操作起来会比较麻烦，因此通常将链队设计成带头结点的单链表。

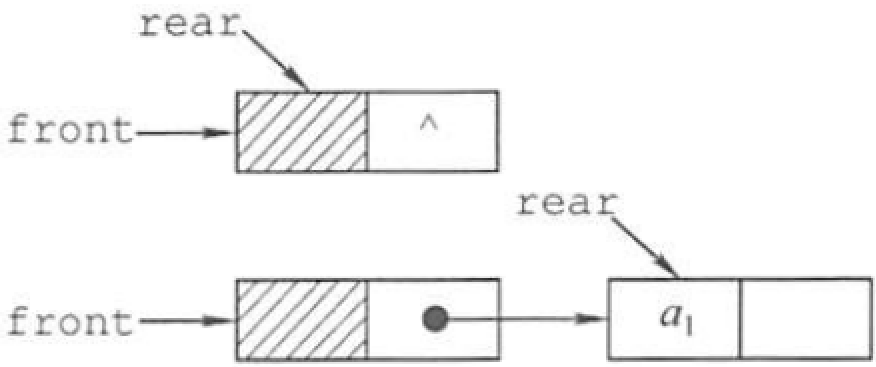


图 3.9 带头结点的链式队列

用单链表表示的链式队列特别适合于数据元素变动比较大的情形，而且不存在队列满且产生溢出的问题。另外，假如程序中要使用多个队列，与多个栈的情形一样，最好使用链式队列，这样就不会出现存储分配不合理和“溢出”的问题。

1、链队的初始化

a) 带头结点

```
bool InitQueue(LinkQueue &Q) {
    Q.front = Q.rear = (LinkNode *) malloc(sizeof(LinkNode));   // 建立头结点
    Q.front->next = NULL;   // 初始为空
}
```

b) 不带头节点

```
bool InitQueue(LinkQueue &Q) {
    Q.front = NULL;
    Q.rear = NULL;
}
```

2、判断链队是否为空

a) 带头结点

```
bool QueueEmpty(LinkQueue Q) {
    if (Q.front == Q.rear) {
        return true;
    } else {
        return false;
    }
}
```

b) 不带头结点

```
bool QueueEmpty(LinkQueue Q) {
    if (Q.front == NULL) {
        return true;
    } else {
        return false;
    }
}
```

3、入队

a) 带头结点

```
bool EnQueue(LinkQueue &Q, int x) {
    LinkNode *s = (LinkNode *) malloc(sizeof(LinkNode));
    s->data = x;
    s->next = NULL;
    Q.rear->next = s;
    Q.rear = s;
    return true;
}
```

b) 不带头节点

```
void EnQueue(LinkQueue &Q, int x) {
    LinkNode *s = (LinkNode *) malloc(sizeof(LinkNode));
    s->data = x;
    s->next = NULL;
    // 第一个元素入队时需要特别处理
    if (Q.front == NULL) {
        Q.front = s;
        Q.rear = s;
    } else {
        Q.rear->next = s;
        Q.rear = s;
    }
}
```

4、出队

a) 带头结点

```
bool DeQueue(LinkQueue &Q, int &x) {
    if (Q.front == Q.rear)
        return false;
    LinkNode *p = Q.front->next;
    x = p->data;
    Q.front->next = p->next;
    // 如果p是最后一个结点，则将队头指针也指向NULL
    if (Q.rear == p)
        Q.rear = Q.front;
    free(p);
    return true;
}
```

b) 不带头节点

```
bool DeQueue(LinkQueue &Q, int &x) {
    if (Q.front == NULL)
        return false;
    LinkNode *s = Q.front;
    x = s->data;
    if (Q.front == Q.rear) {
        Q.front = Q.rear = NULL;
    } else {
        Q.front = Q.front->next;
    }
    free(s);
    return true;
}
```

3.2.5 双端队列

1. 定义：
    1. 双端队列是允许从两端插入、两端删除的线性表。
    2. 如果只使用其中一端的插入、删除操作，则等同于栈。
    3. 输入受限的双端队列：允许一端插入，两端删除的线性表。
    4. 输出受限的双端队列：允许两端插入，一端删除的线性表。
2. 考点：判断输出序列的合法化

- 例：数据元素输入序列为 1，2，3，4，判断 4! = 24 个输出序列的合法性

> 栈中合法的序列，双端队列中一定也合法

栈	输入受限的双端队列	输出受限的双端队列
14个合法 $\frac{1}{n+1}C_{2n}^{n}$	只有 4213 和 4231 不合法	只有 4132 和 4231 不合法

3.3 栈和队列的应用

3.3.1 栈在括号匹配中的应用

> 括号匹配的规律：最后出现的左括号最先被匹配（LIFO，用栈来实现该特性是最优解）

> 每当出现一个右括号，就"消耗"一个左括号；这里的消耗就对应于出栈的过程，当我们遇到左括号就把它压入栈中，当我们遇到右括号的时候，就把栈顶的那个左括号弹出

【算法步骤】

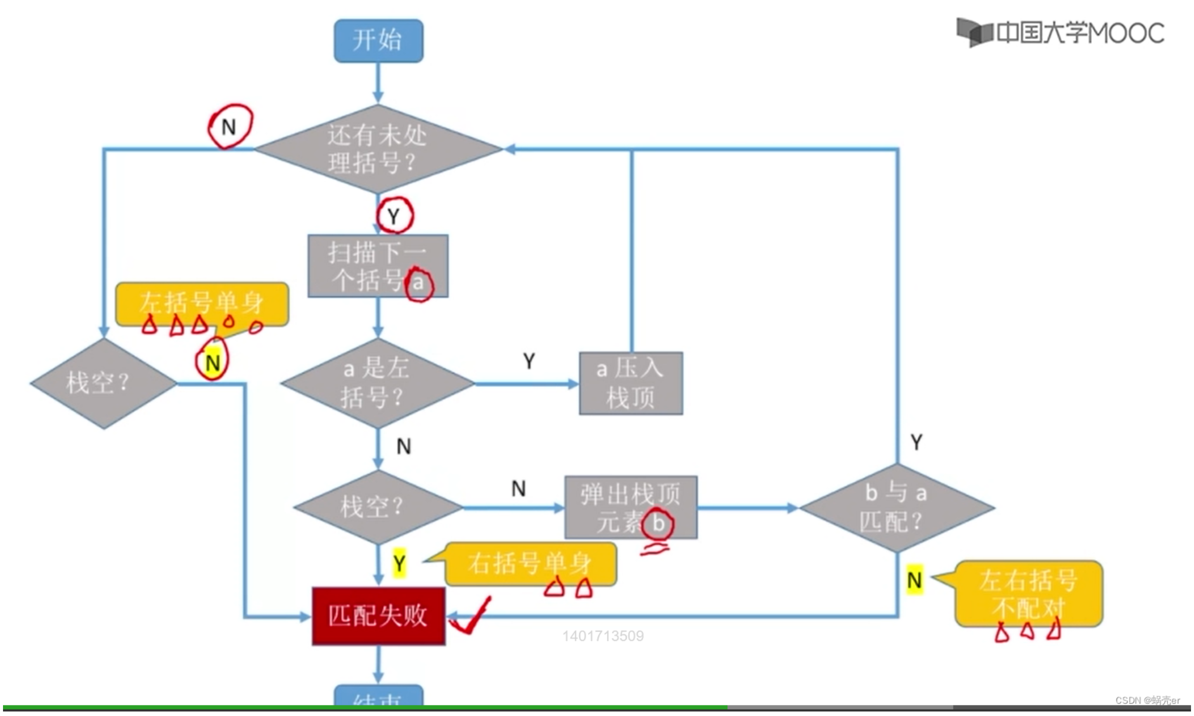
① 初始化一个空栈S。

② 设置一标记性变量flag，用来标记匹配结果以控制循环及返回结果，1表示正确匹配，0表示错误匹配，flag初值为1。

③ 扫描表达式，依次读入字符ch，如果表达式没有扫描完毕或flag非零，则循环执行以下操作：

- 若ch是左括号“[”或“(”，则将其压入栈；
- 若ch是右括号“)”，则根据当前栈顶元素的值分情况考虑：若栈非空且栈顶元素是“(”，则正确匹配，否则错误匹配，flag置为0；
- 若ch是右括号“]”，则根据当前栈顶元素的值分情况考虑：若栈非空且栈顶元素是“[”，则正确匹配，否则错误匹配，flag置为0。

④ 退出循环后，如果栈空且flag值为1，则匹配成功，返回true，否则返回false。



3.3.2 栈在表达式求值中的应用

> 前缀表达式（波兰表达式）：运算符在两个操作数的前面
>
> 中缀表达式：运算符在两个操作数中间
>
> 后缀表达式（逆波兰表达式）：运算符在两个操作数后

前缀表达式	中缀表达式	后缀表达式
+ 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

> 注意:
>
> 1、运算顺序是不唯一的，所有手算得到的后缀表达式也不唯一
>
> 2、得到的不唯一的后缀表达式客观上都是正确的，但是机算得到的结果只有一种
>
> 3、**为了保证手算机算结果相同，我们在手算时，要遵循"左优先原则"，只要左边的运算符能先计算，就计算左边的。**

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。

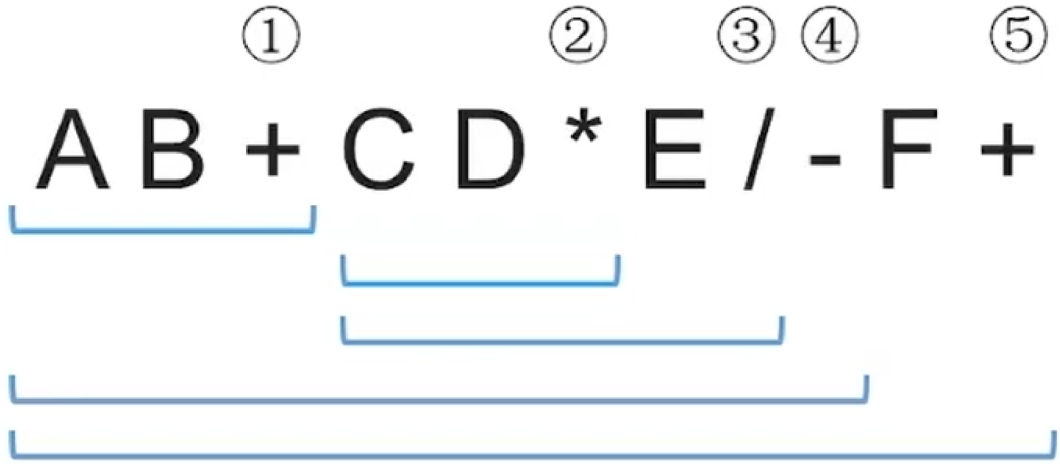
③遇到运算符。依次弹出栈中优先级高于或等于当前运算符的所有运算符，并加入后缀表达式，若碰到"("或栈空则停止。之后再把当前运算符入栈。

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。

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

> 注意两个操作数的操作顺序!



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

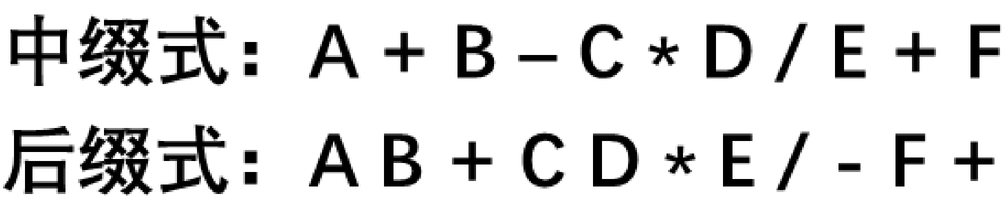
> 注意：先出栈的是 **右操作数**

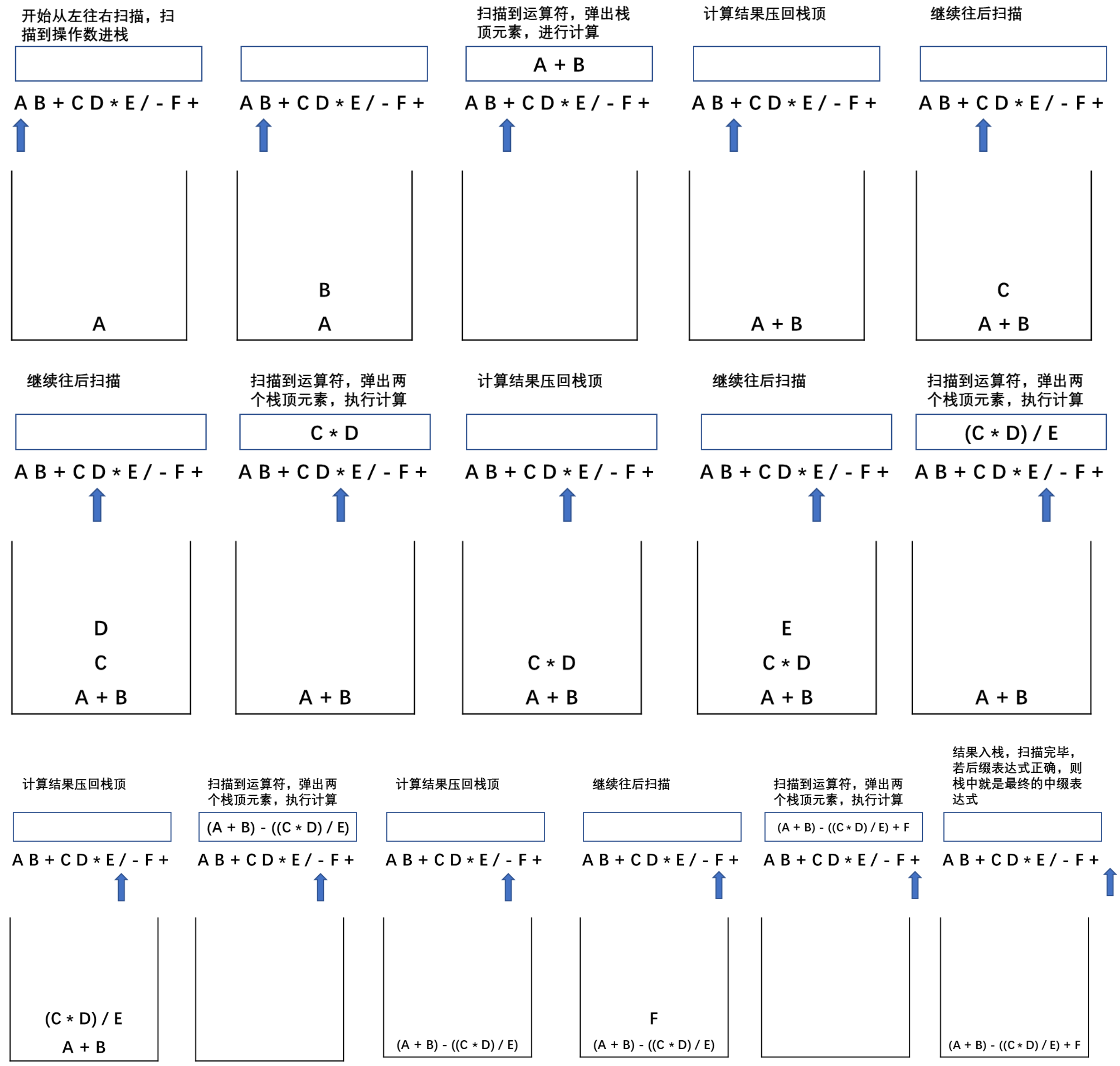
注意:先出栈的是“右操作数”

若表达式合法,则最后栈中只会留下一个元素，就是最终结果

以下是机算的图解:

例:





中缀表达式转前缀表达式

中缀转前缀的手算方法:

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「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

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

机算

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)

3.3.3 栈在递归中的应用

> 递归：将原始问题转化成属性相同，但规模更小的问题

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“函数调用栈” 存储：

> 调用返回地址
> 实参
> 局部变量

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：效率低，太多层递归可能会导致栈溢出；可能包含很多重复计算。

可以**自定义栈**将递归算法改造成非递归算法。

3.3.4 队列的应用

1. 树的层次遍历
2. 图的广度优先遍历
3. 操作系统中多个进程争抢着使用有限的系统资源时，先来先服务算法（First Come First Service）是是一种常用策略。

3.4 数组和特殊矩阵

3.4.1 数组的存储

> 除非题目特别说明，否则数组下标默认从0开始。

一维数组的存储：各数组元素大小相同，且物理上连续存放。以一维数组A[0...n-1]为例，其存储关系式为：

$$\text{数组元素}a[i]\text{的存储地址} = LOC + i * sizeof(ElemType)$$

其中，L是每个数组元素所占的存储单元。

多维数组的存储：

m行n列的二维数组`b[m][n]`中，若按照行优先存储：

$$b[i][j]\text{的存储地址} = LOC + (i * n + j) * sizeof(ElemType)$$

m行n列的二维数组`b[m][n]`中，若按照列优先存储：

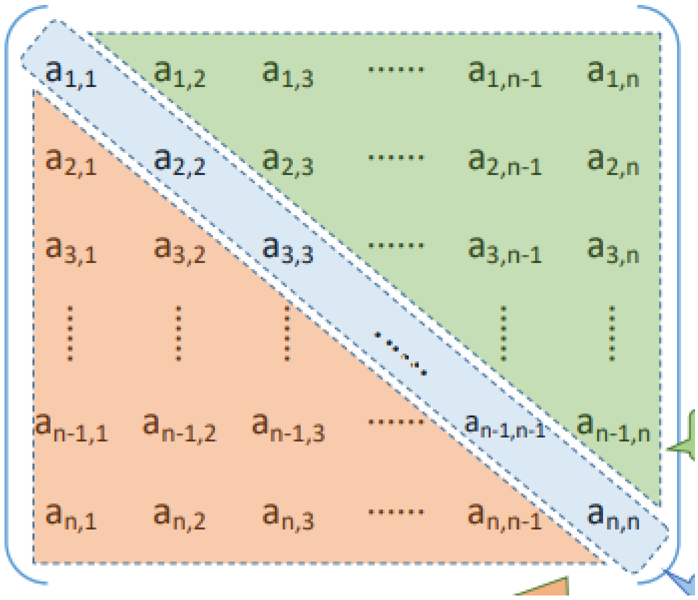
$$b[i][j]\text{的存储地址} = LOC + (j*m+i)*sizeof(ElemType)$$

3.4.2 特殊矩阵的压缩存储

1、对角矩阵

所交矩阵中存在着大量相同元素，若仍然采用二维数组存储，则会浪费几乎一半的空间。

> 解决策略：只存储主对角线元素+上三角区或下三角区的元素。



$$\text{其需要的一维数组大小} = 1+2+3+...+n = n(1+n)/2$$

对于具体的元素的查找，我们不能和二维数组一样直接使用下标进行查找，而是需要建立一个映射函数来在一维数组中进行查找，如：

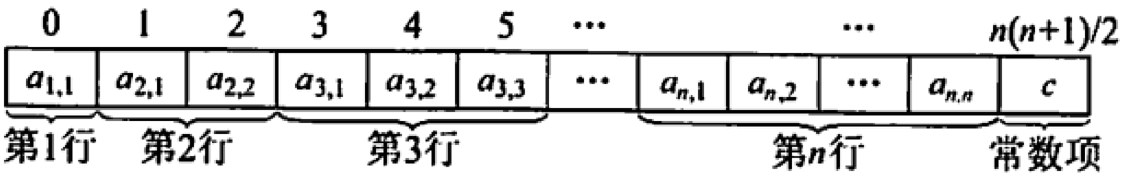
$$k=\begin{cases}\dfrac{i(i-1)}{2}+j-1, & i\ge j\ (\text{下三角区和主对角线元素})\\ \dfrac{j(j-1)}{2}+i-1, & i<j\ (\text{上三角区元素}a_{ij}=a_{ji})\end{cases}$$

2、三角矩阵

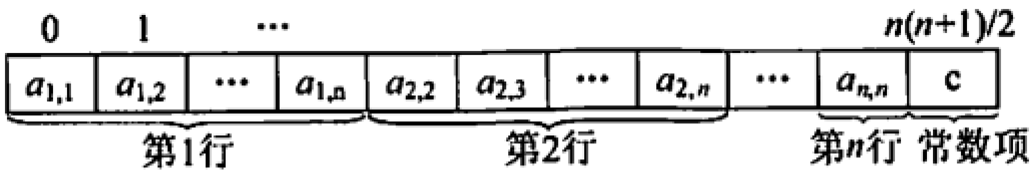
1. 下三角矩阵：除了主对角线和下三角区，其余的元素都相同。
2. 上三角矩阵：除了主对角线和上三角区，其余的元素都相同。

压缩存储策略：按行优先原则将主对角线+下三角区存入一维数组中，并在最后一个位置存储常量。

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j-1, & i\ge j\ (\text{下三角区和主对角线元素})\\ \dfrac{n(n+1)}{2}, & i<j\ (\text{上三角区元素})\end{cases}$$



$$k=\begin{cases}\dfrac{(i-1)(2n-i+2)}{2}+(j-i), & i\le j\text{（上三角区和主对角线元素）}\\[2ex] \dfrac{n(n+1)}{2}, & i>j\text{（下三角区元素）}\end{cases}$$



3、三对角矩阵

三对角矩阵 A 也可以采用压缩存储，将 3 条对角线上的元素按行优先方式存放在一维数组 B 中，且 $a_{1,1}$ 存放于 B[0] 中，其存储形式如图 3.25 所示。

由此可以计算矩阵 A 中 3 条对角线上的元素 $a_{i,j}$（$1\le i,j\le n$, $|i-j|\le 1$）在一维数组 B 中存放的下标为 $k=2i+j-3$。

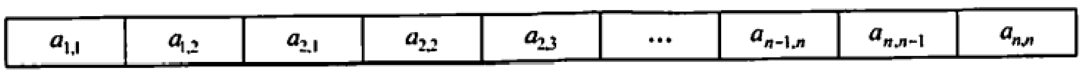
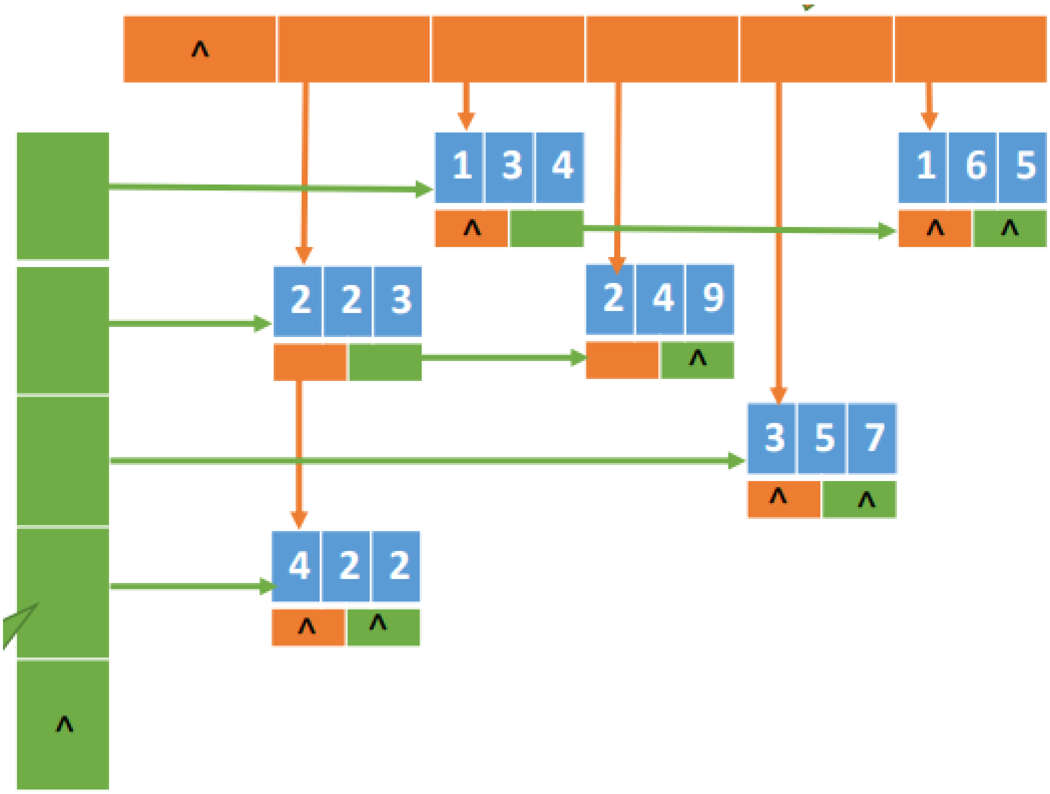


图 3.25 三对角矩阵的压缩存储

4、稀疏矩阵

1. 稀疏矩阵的非零元素远远少于矩阵元素的个数。压缩存储策略：
    1. 顺序存储：三元组 <行，列，值>
    2. 链式存储：十字链表法

i（行）	j（列）	v（值）
1	3	4
1	6	5
2	2	3
2	4	9
3	5	7
4	2	2



非零数据结点说明：