四.队列

- 只允许在表的一端(队尾)插入，表的另一端(队头)进行删除操作的线性表。
- 先进先出

队列的基本操作

- 创、销
  InitQueue(&Q)：初始化队列，构造一个空队列Q。
  ClearQueue(&Q)：销毁队列，并释放队列Q占用的内存空间。
- 增、删
  EnQueue(&Qx)：入队，若队列Q未满，则将x加入使之成为新的队尾。
  DeQueue(&Q&x)：出队，若队列Q非空，则删除队头元素，并用x返回。
- 查
  GetHead(Q&x)：读队头元素，若队列Q非空则用x返回队头元素。
- 其它常用操作
  QueueEmpty(Q)：判队列空，若队列Q为空返回true，否则返回false。

队列的顺序存储

```
//顺序队列的定义
#define MaxSize 10;      //定义队列中元素的最大个数

typedef struct{
    ElemType data[MaxSize];    //用静态数组存放队列元素
    int front, rear;           //队头指针和队尾指针
}SqQueue;

void test{
    SqQueue Q;                 //声明一个队列
}


// 顺序队列的初始化
void InitQueue(SqQueue &Q){
    // 初始化时，队头、队尾指针指向0
    // 队尾指针指向队尾元素后一个位置
    // 队头指针指向的是队头元素的数组下标
    Q.rear = Q.front = 0;
}

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

// 1判断队列是否已满(牺牲一个存储空间，将仅剩一个空位置的状态当作“满”状态)
(Q.rear+1)%MaxSize==Q.front//队尾指针的下一个位置是对头

// 2判断队列是否已满(设置一个入队或出队标志tag，每次删除操作成功时，都令tag=0;每次插入操作成功时，都令
front == rear && tag == 1
```

注：取余运算：(Q.rear+1)%MaxSize; 将存储空间在逻辑上变成了环状，将无限的整数域映射到有限的整数集合{0，1，2，...,MaxSize-1}上。队列元素个数：(rear+MaxSize-front)%MaxSize

```
// 入队
bool EnQueue(SqQueue &Q, ElemType x){
    // 如果队列已满直接返回
    if((Q.rear+1)%MaxSize == Q.front)   //牺牲一个单元区分队空和队满
        return false;
    Q.data[Q.rear] = x;
    Q.rear = (Q.rear+1)%MaxSize;
    return true;
}

// 出队
bool DeQueue(SqQueue &Q, ElemType &x){
    // 如果队列为空直接返回
    if(Q.rear == Q.front)
        return false;
    x = Q.data[Q.front];
    Q.front = (Q.front+1)%MaxSize;
    return true;
}

// 获取队头元素并存入x
bool GetHead(SqQueue &Q, ElemType &x){
    if(Q.rear == Q.front)
        return false;
    x = Q.data[Q.front];
    return true;
}
```

队列的链式存储

- 链队列的定义

```
// 链式队列结点
typedef struct LinkNode{
    ElemType data;
    struct LinkNode *next;
}LinkNode;

// 链式队列
typedef struct{
    LinkNode *front, *rear;// 头指针和尾指针
    // int count;    计数变量
}LinkQueue;
```

- 初始化队列（带头结点）

```
void InitQueue(LinkQueue &Q) {
    // 初始化时，front、rear都指向头结点
    Q.front = Q.rear = new LinkNode;
    Q.front->next = NULL;
}

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

- 入队、出队

```
//新元素入队
void EnQueue(LinkQueue &Q, ElemType x) {
    LinkNode *s = new LinkNode;
    s->data = x;
    s->next = nullptr;
    Q.rear->next = s;
    Q.rear = s;
}
//队头元素出队
bool DeQueue(LinkQueue &Q, ElemType &x) {
    if (Q.front == Q.rear)
        return false; //空队
    LinkNode *p = Q.front->next;
    x = p->data;//用变量x返回队头元素
    Q.front->next = p->next;//修改头结点的next指针
    if (Q.rear == p)
        Q.rear = Q.front;// 如果p是最后一个结点，则将rear指针也指向NULL
    delete p;//释放结点空间
    return true;
}
```

双端队列

- 双端队列是允许从两端插入、两端删除的线性表。
- 如果只使用其中一端的插入、删除操作，则等同于栈。
- 输入受限的双端队列：允许一端插入，两端删除的线性表。
- 输出受限的双端队列：允许两端插入，一端删除的线性表。

题型：判断输出序列的合法化

队列的应用

——1.树的层次遍历

- 若树非空，则根结点入队；
- 若队列非空，队头元素出队并访问，同时将该元素的孩子依次入队；
- 重复以上操作直至队尾为空；

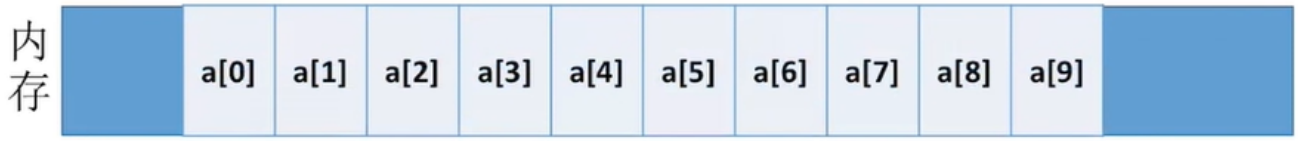
——2.图的广度优先遍历

- 找到与顶点相邻的所有顶点；
- 未被访问过顶点入队尾；
- 改顶点出队。

——3.在操作系统中的应用

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。

数组的存储



- 一维数组的存储：各数组元素大小相同，且物理上连续存放。设起始地址为LOC，则数组元素a[i]的存放地址 = LOC + i * sizeof(ElemType) (0≤i<10)

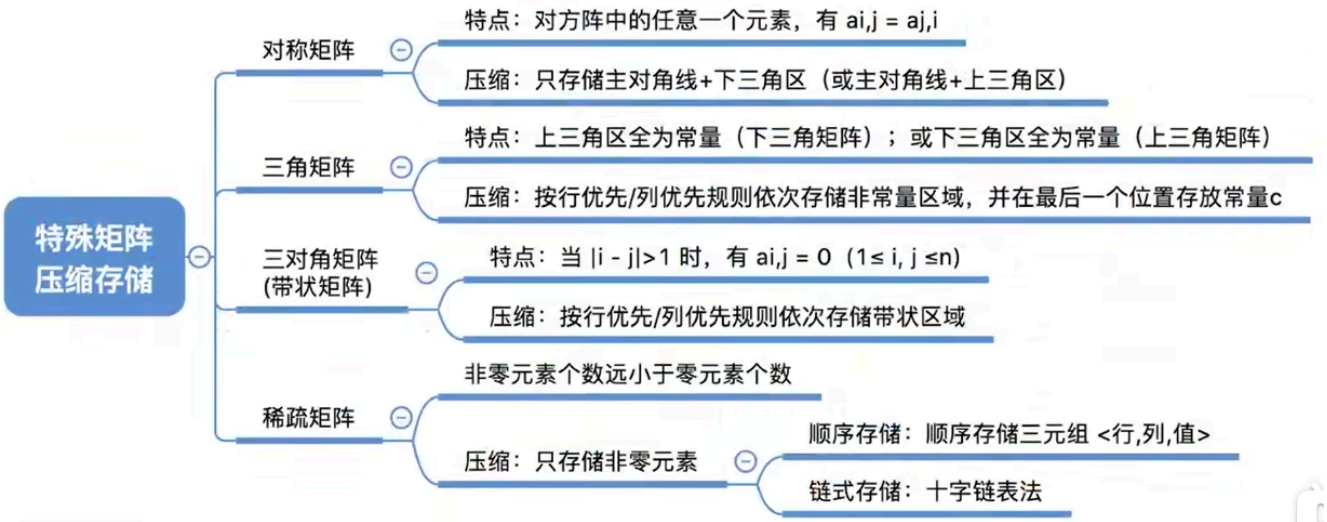
- 

  - M行N列的二维数组b[M][N]中，若按行优先存储，则b[i][j]的存储地址=LOC + （i*N+j)*sizeof(ElemType)
  - M行N列的二维数组b[M][N]中，若按列优先存储，则b[i][j]的存储地址=LOC + （j*M+i)*sizeof(ElemType)

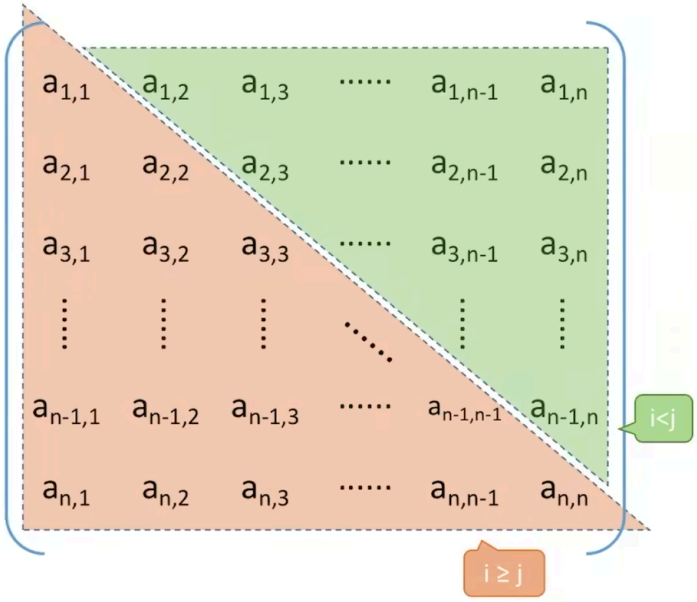
矩阵的压缩存储

- **题型**:
  - 矩阵的压缩存储需要多长的数组
  - 由矩阵行列号<i,j>推出对应的数组下标号k

- 由k推出<i,j>
- 存储上三角/下三角、行优先/列优先、矩阵元素的下标从0/1开始、数组下标从0/1开始



- **普通矩阵**可用二位数组存储，描述矩阵元素时，行、列号通常从1开始。
- **对称矩阵**的压缩存储
  - 若n阶方阵中任意一个元素$a_{i,j}$,都有$a_{i,j} = a_{j,i}$则该矩阵为对称矩阵
  - 压缩存储策略：只存储主对角线+下三角区
  - 数组大小：$\frac{(n+1)*n}{2}$
  - 实现一个映射函数：矩阵下标——>一维数组下标。



策略：只存储主对角线+下三角区

按行优先原则将各元素存入一维数组中。

B[0]	B[1]	B[2]	B[3]	….	B[$\frac{n(n+1)}{2}$-1]
$a_{1,1}$	$a_{2,1}$	$a_{2,2}$	$a_{3,1}$	……	$a_{n,n-1}$	$a_{n,n}$

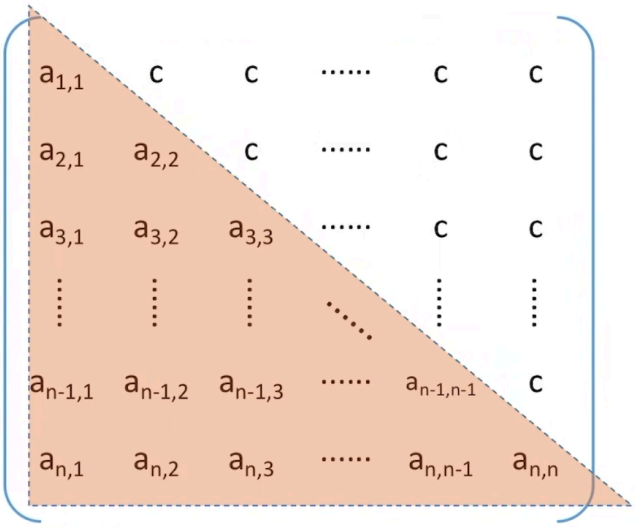
矩阵下标 → 一维数组下标

$a_{i,j}$ → B[k]

$a_{i,j} = a_{j,i}$（对称矩阵性质）

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

- **三角矩阵**的压缩存储
  - 下三角矩阵：除了主对角线和下三角区，其余的元素都相同为一常量。
    上三角矩阵：除了主对角线和上三角区，其余的元素都相同为一常量。
  - 数组大小：$\frac{(n-1)*n}{2} + 1$



下三角矩阵：除了主对角线和下三角区，其余的元素都相同

压缩存储策略：按行优先原则将橙色区元素存入一维数组中。并在最后一个位置存储常量c

矩阵下标 → 一维数组下标

$a_{i,j}$ （i≥j） → B[k]

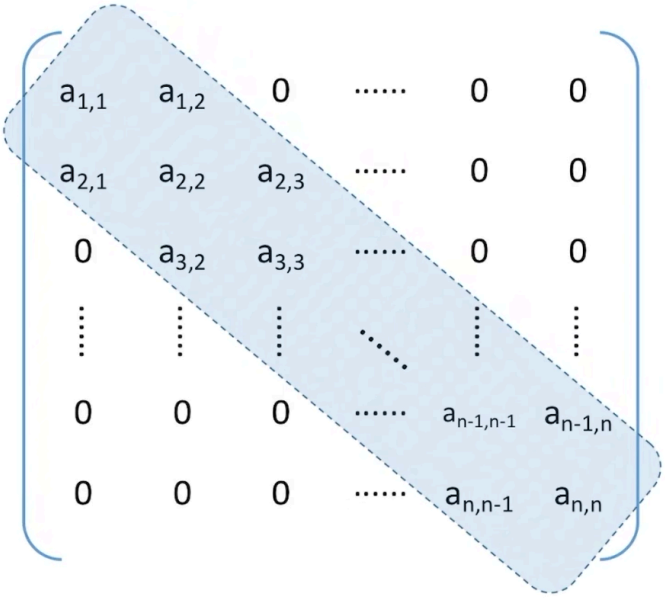
Key：按行优先的原则，$a_{i,j}$是第几个元素？

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

- **三对角矩阵**的压缩存储

  当$|i-j|>1$时，有$a_{i,j}$=0$(1\leqslant i,j\leqslant n)$

  - 数组大小：3n-2

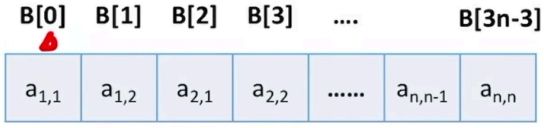


三对角矩阵，又称带状矩阵：

当 |i - j|>1 时，有 $a_{i,j}$ = 0 （1≤ i, j ≤n）

压缩存储策略：

按行优先（或列优先）原则，只存储带状部分



矩阵下标 → 一维数组下标

$a_{i,j}$ （|i-j|≤1） → B[k]

Key：按行优先的原则，$a_{i,j}$是第几个元素？

前i-1行共 3(i-1)-1 个元素

$a_{i,j}$是 i 行第 j-i+2 个元素

$a_{i,j}$是第 2i+j-2 个元素 → k = 2i+j-3

数组下标从0开始

- **稀疏矩阵**的压缩存储

  非零元素远远少于矩阵元素的个数

压缩存储策略：
顺序存储——三元组 <行，列，值>

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

（注：此处行、列标从1开始）