数组与广义表

数组、矩阵压缩存储与广义表

1. 数组的类型定义与基本操作

核心思想：数组是线性表的扩展，元素本身也可以是数据结构（如多维数组）。

```
// 数组ADT定义（伪代码）
typedef struct {
    ElemType *base;   // 数组基地址
    int dim;          // 维度
    int *bounds;      // 各维长度数组
    int *constants;   // 各维偏移量系数
} Array;

// 初始化数组
Status InitArray(Array *A, int dim, int *bounds) {
    // 计算元素总数 & 分配内存
    // 计算各维偏移量系数（ci = L * bi * ci+1）
    // 返回OK
}

// 存取元素
Status Value(Array A, ElemType *e, int *indices) {
    // 检查下标是否越界
    // 计算元素位置：LOC = base + Σ(indices[i] * constants[i])
    // 取元素值到e
}

// 修改元素
Status Assign(Array *A, ElemType e, int *indices) {
    // 同上计算位置
    // 将e写入该位置
}
```

---

2. 数组的顺序存储

核心公式（行优先存储）：
$LOC(j_1, j_2, \dots, j_n) = LOC(0, 0, \dots, 0) + \sum_{i=1}^{n}(c_i \times j_i)$
其中 $c_n = L$（元素大小），$c_{i-1} = b_i \times c_i$。

二维数组示例：

```
// 二维数组元素位置计算
int getPosition(int i, int j, int col) {
    return i * col + j;  // 行优先：LOC(i,j)=基址+(i*总列数+j)*元素大小
}
```

3. 矩阵压缩存储

(1) 对称矩阵

思想：只存储下三角元素（包括对角线）。

```
// 下三角矩阵元素位置（基1）
int getIndex(int i, int j) {
    if (i >= j)
        return i*(i-1)/2 + j-1;  // 下三角区
    else
        return j*(j-1)/2 + i-1;  // 上三角对称位置
}
```

(2) 三角矩阵

下三角矩阵公式：
$\mathrm{Loc}(a_{ij}) = \mathrm{Loc}(a_{11}) + \frac{i(i-1)}{2} + j - 1 \quad (i \geq j)$

```
// 下三角矩阵存储（含常量上三角）
void storeLowerTri(int mat[][N], int n, ElemType storage[]) {
    int k = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {  // 只存下三角
            storage[k++] = mat[i][j];
        }
    }
    storage[k] = CONSTANT;  // 上三角常量值
}
```

(3) 三对角矩阵

元素位置公式（基1）：
$\mathrm{Loc}(a_{ij}) = \mathrm{Loc}(a_{11}) + 2(i-1) + (j-1)$

```
// 三对角矩阵存取
int getTriDiagIndex(int i, int j) {
    return 2*(i-1) + (j-1);  // 每行最多3元素
}
```

---

4. 稀疏矩阵压缩存储

(1) 三元组顺序表

```
typedef struct {
    int i, j;     // 行号、列号
    ElemType e;   // 元素值
} Triple;

typedef struct {
    Triple data[MAXSIZE];  // 三元组数组
    int mu, nu, tu;        // 行数、列数、非零元数
```

```c
} TSMatrix;

// 转置矩阵算法（O(nu*tu)）
Status Transpose(TSMatrix M, TSMatrix *T) {
    T->mu = M.nu; T->nu = M.mu; T->tu = M.tu;
    int q = 0;  //三元组数组下标
    for (int col = 0; col < M.nu; col++) {      // 按列扫描
        for (int p = 0; p < M.tu; p++) {        // 遍历非零元
            if (M.data[p].j == col) {           // 找到该列元素
                T->data[q].i = M.data[p].j;      // 行列互换
                T->data[q].j = M.data[p].i;
                T->data[q].e = M.data[p].e;
                q++;
            }
        }
    }
    return OK;
}
```

(2) 快速转置（O(nu+tu)）

优化思想：预处理每列首个非零元位置。

```c
// 快速转置算法：优化稀疏矩阵转置操作
Status FastTranspose(TSMatrix M, TSMatrix *T) {
    // 1. 初始化转置矩阵T的基本信息
    T->mu = M.nu;  // 转置后行数 = 原列数
    T->nu = M.mu;  // 转置后列数 = 原行数
    T->tu = M.tu;  // 非零元素数量不变

    // 特殊情况：如果矩阵为空，直接返回
    if (M.tu == 0) return OK;

    // 2. 准备辅助数组
    int num[M.nu];   // 存储原矩阵每列的非零元素个数（转置后每行的元素个数）
    int cpot[M.nu];  // 存储原矩阵每列首个非零元素在转置矩阵中的存储位置

    // 初始化num数组为0
    for (int col = 0; col < M.nu; col++) {
        num[col] = 0;
    }

    // 3. 统计原矩阵每列的非零元素个数
    // 遍历原矩阵的所有非零元素
    for (int t = 0; t < M.tu; t++) {
        int col = M.data[t].j;  // 获取当前元素的列号
        num[col]++;             // 对应列的非零元素计数+1
    }

    // 4. 计算每列在转置矩阵中的起始位置
    cpot[0] = 0;  // 第0列的首个元素位置从0开始
    // 递推公式：当前列起始位置 = 前一列起始位置 + 前一列元素个数
    for (int col = 1; col < M.nu; col++) {
        cpot[col] = cpot[col-1] + num[col-1];
    }
```

```c

    // 5. 执行转置操作
    // 遍历原矩阵的每个非零元素
    for (int p = 0; p < M.tu; p++) {
        int col = M.data[p].j;   // 当前元素在原矩阵中的列号
        int q = cpot[col];       // 该元素在转置矩阵中的存储位置

        // 执行转置：行列互换，值不变
        T->data[q].i = M.data[p].j;  // 原列号 -> 转置行号
        T->data[q].j = M.data[p].i;  // 原行号 -> 转置列号
        T->data[q].e = M.data[p].e;  // 元素值保持不变

        // 更新该列的下一个存储位置
        cpot[col]++;//cpoy[]存储了每列首个元素的存储位置，后面又变掉了
    }

    return OK;  // 转置完成
}
```

(3) 十字链表（适合矩阵运算）

```c
// 十字链表节点定义
typedef struct OLNode {
    int i, j;                   // 节点在矩阵中的行号和列号
    ElemType e;                 // 节点存储的元素值
    struct OLNode *right;       // 指向同一行中下一个非零元素的指针
    struct OLNode *down;        // 指向同一列中下一个非零元素的指针
} OLNode;

// 十字链表结构定义
typedef struct {
    OLNode *rhead[MAXROW];    // 行头指针数组：每个元素指向该行第一个非零元素
    OLNode *chead[MAXCOL];    // 列头指针数组：每个元素指向该列第一个非零元素
    int mu;                   // 矩阵的行数
    int nu;                   // 矩阵的列数
    int tu;                   // 非零元素的总数
} CrossList;

// 向十字链表中插入新节点
void insertNode(CrossList *M, OLNode *newNode) {
    int row = newNode->i;  // 获取新节点的行号
    int col = newNode->j;  // 获取新节点的列号

    // ====== 行插入操作 ======
    // 目标：将新节点插入到行链表的正确位置（按列号从小到大排序）
    OLNode *p = M->rhead[row];  // 获取该行链表的头指针

    // 情况1：插入到行首（行链表为空 或 新节点列号最小）
    if (!p || p->j > col) {
        newNode->right = p;      // 新节点指向原行首节点
        M->rhead[row] = newNode; // 新节点成为行首
    }
    // 情况2：插入到行链表中间
    else {
        // 遍历行链表，找到插入位置的前驱节点
```

```c
        // 条件：下一个节点存在 且 下一个节点的列号小于新节点的列号
        while (p->right && p->right->j < col) {
            p = p->right;
        }
        // 插入新节点：新节点指向后节点，前驱节点指向新节点
        newNode->right = p->right;
        p->right = newNode;
    }

    // ====== 列插入操作 ======
    // 目标：将新节点插入到列链表的正确位置（按行号从小到大排序）
    OLNode *q = M->chead[col];  // 获取该列链表的头指针

    // 情况1：插入到列首（列链表为空 或 新节点行号最小）
    if (!q || q->i > row) {
        newNode->down = q;       // 新节点指向原列首节点
        M->chead[col] = newNode; // 新节点成为列首
    }
    // 情况2：插入到列链表中间
    else {
        // 遍历列链表，找到插入位置的前驱节点
        // 条件：下一个节点存在 且 下一个节点的行号小于新节点的行号
        while (q->down && q->down->i < row) {
            q = q->down;
        }
        // 插入新节点：新节点指向下节点，前驱节点指向新节点
        newNode->down = q->down;
        q->down = newNode;
    }

    // 更新非零元素计数
    M->tu++;
}
```

---

5. 广义表

(1) 定义与存储结构

头尾链表表示:

```c
typedef enum { ATOM, LIST } ElemTag;
typedef struct GLNode {
    ElemTag tag;  // 0=原子, 1=子表
    union {
        AtomType atom;          // 原子值
        struct {
            struct GLNode *hp;   // 表头指针
            struct GLNode *tp;   // 表尾指针
        } ptr;
    };
} *GList;

// 示例：广义表 D = (a, (b, c))
// 结构：D -> [LIST, hp->a, tp-> [LIST, hp->(b,c), tp=NULL]
```

(2) 递归算法

求深度:

```
int GListDepth(GList L) {
    if (!L) return 1;              // 空表深度=1
    if (L->tag == ATOM) return 0;  // 原子深度=0

    int maxDepth = 0;
    GList p = L;
    while (p) {
        int depth = GListDepth(p->ptr.hp);  // 递归求子表深度
        if (depth > maxDepth) maxDepth = depth;
        p = p->ptr.tp;  // 处理下一子表
    }
    return maxDepth + 1;  // 当前层深度=子表最大深度+1
}
```

复制广义表:

```
Status CopyGList(GList *T, GList L) {
    if (!L) { *T = NULL; return OK; }  // 复制空表

    *T = (GList)malloc(sizeof(GLNode));
    (*T)->tag = L->tag;

    if (L->tag == ATOM) {
        (*T)->atom = L->atom;  // 复制原子
    } else {
        CopyGList(&((*T)->ptr.hp), L->ptr.hp);  // 递归复制表头
        CopyGList(&((*T)->ptr.tp), L->ptr.tp);  // 递归复制表尾
    }
    return OK;
}
```

---

6. 递归算法设计思想

(1) 分治法

思想：将问题分解为相同类型的子问题（如汉诺塔）。

```
void hanoi(int n, char from, char via, char to) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", from, to);
    } else {
        hanoi(n-1, from, to, via);    // 将n-1个盘移到中转柱
        printf("Move disk %d from %c to %c\n", n, from, to);
        hanoi(n-1, via, from, to);    // 将n-1个盘移到目标柱
    }
}
```

(2) 回溯法

思想：试探性搜索，失败则回退（如N皇后问题）。

```
void solveNQueens(int row, int n, int board[]) {
    if (row == n) {
        printSolution(board);  // 找到解
        return;
    }
    for (int col = 0; col < n; col++) {
        if (isSafe(row, col, board)) {  // 检查位置
            board[row] = col;           // 放置皇后
            solveNQueens(row+1, n, board);  // 递归下一行
            board[row] = -1;            // 回溯（撤销选择）
        }
    }
}
```

关键点总结

1. **数组存储**：多维数组映射到一维内存，行优先公式最重要。
2. **矩阵压缩**：
    - 对称/三角矩阵：只存一半元素，下标转换是关键。
    - 稀疏矩阵：三元组适合运算，十字链表适合动态变化。
3. **广义表**：
    - 头尾链表存储灵活支持递归操作。
    - 递归算法注意终止条件（空表/原子）。
4. **递归设计**：
    - 分治法：汉诺塔、二叉树遍历。
    - 回溯法：N皇后、迷宫问题。
    - 避免重复计算（如斐波那契用动态规划优化）。

> 例题代码和详细注释已嵌入各节，所有伪代码均转为C风格。重点理解算法思想而非死记硬背语法！