第4章 扩展线性表—数组与广义表

#数据结构与算法

4.1 数组

4.1.1 数组的定义

数组是由n（n≥1）个相同类型的数据元素组成的有限序列，记作 $A=(a_0,a_1,\ldots,a_{n-1})$。

数组的特点：
1. **数据类型相同**：数组中的每个元素都属于同一种数据类型。
2. **连续存储**：数组元素在内存中通常是连续存放的。
3. **随机存取**：可以通过下标直接访问任何一个元素，时间复杂度为O(1)。

4.1.2 数组的基本操作

初始化：为数组分配内存空间并赋初值。
取值/访问：通过下标获取数组中某个位置的元素值，时间复杂度为O(1)。
赋值/修改：通过下标修改数组中某个位置的元素值，时间复杂度为O(1)。
插入：在数组的指定位置插入一个新的元素，后续元素需要后移，平均时间复杂度为O(n)。
删除：删除数组中指定位置的元素，后续元素需要前移，平均时间复杂度为O(n)。

4.1.3 数组的存储结构

- **一维数组**：存储地址 `LOC(a_i) = LOC(a_0) + i * L` ，其中L为每个元素占用的存储单元数。
- **二维数组** `A[m][n]` 推广到一般情况：
  以行为主序 (Row-major order)：`LOC(a_i,j) = LOC(a_0,0) + (i * n + j) * L`
  以列为主序 (Column-major order)：`LOC(a_i,j) = LOC(a_0,0) + (j * m + i) * L`

4.1.4 矩阵的压缩存储

- **压缩存储**是指为多个值相同的元素只分配一个存储空间，或者对零元素不分配空间的存储方式。
- **特殊矩阵**：指具有某种规律的矩阵，如对称矩阵、三角矩阵、对角矩阵等。
  - **对称矩阵**：可以只存储上三角或下三角部分的元素。
  - **三角矩阵**：可以只存储非零部分的元素。
- **稀疏矩阵**：指非零元素个数远远少于矩阵元素总数的矩阵（稀疏因子小于0.05）。
  - **存储方法**：三元组顺序表、十字链表等。

十字链表的实现

十字链表是稀疏矩阵的一种高效的链式存储结构。它能快速定位任意行或列的非零元素。

1. 节点结构

每个非零元素都表示为一个节点，包含五个域：

- `row`，`col`：元素所在的行和列。
- `value`：元素的值。
- `right`：指向同一行中下一个非零元素的指针。
- `down`：指向同一列中下一个非零元素的指针。

2. 矩阵结构

整个矩阵由一个结构体表示，包含：

- `rows`，`cols`：矩阵的行数和列数。
- `num`：非零元素的个数。
- `rhead`：行头指针数组，`rhead[i]` 指向第 `i` 行的第一个非零元素节点。
- `chead`：列头指针数组，`chead[j]` 指向第 `j` 列的第一个非零元素节点。

优点

快速存取：无论是按行还是按列遍历非零元素，都非常高效，时间复杂度仅与该行/列的非零元素个数有关。
空间高效：只为非零元素分配空间。

实现示例：十字链表

```
#include <iostream>
#include <vector>

// 十字链表节点
template<typename T>
struct OLNode {
    int row, col;
    T value;
    OLNode<T> *right, *down;
};

// 十字链表矩阵结构
template<typename T>
class CrossListMatrix {
public:
    CrossListMatrix(int rows, int cols)
        : m_rows(rows), m_cols(cols), m_num(0) {
        rhead.resize(rows, nullptr);
        chead.resize(cols, nullptr);
    }
```

```
~CrossListMatrix() {
    // 释放所有节点内存
    for (int i = 0; i < m_rows; ++i) {
        OLNode<T>* p = rhead[i];
        while (p) {
            OLNode<T>* next = p->right;
            delete p;
            p = next;
        }
    }
}

void insert(int row, int col, T value) {
    if (value == 0) return;

    OLNode<T>* newNode = new OLNode<T>{row, col, value, nullptr, nullptr};
    m_num++;

    // 在行中插入
    if (rhead[row] == nullptr || rhead[row]->col > col) {
        newNode->right = rhead[row];
        rhead[row] = newNode;
    } else {
        OLNode<T>* p = rhead[row];
        while (p->right && p->right->col < col) {
            p = p->right;
        }
        newNode->right = p->right;
        p->right = newNode;
    }

    // 在列中插入
    if (chead[col] == nullptr || chead[col]->row > row) {
        newNode->down = chead[col];
        chead[col] = newNode;
    } else {
        OLNode<T>* q = chead[col];
        while (q->down && q->down->row < row) {
            q = q->down;
        }
        newNode->down = q->down;
        q->down = newNode;
    }
```

```
    }

    void print() {
        std::cout << "Matrix (" << m_rows << "x" << m_cols << "), Non-zero elements: " << m_num <<
std::endl;
        for (int i = 0; i < m_rows; ++i) {
            OLNode<T>* p = rhead[i];
            while (p) {
                std::cout << "(" << p->row << ", " << p->col << "): " << p->value << std::endl;
                p = p->right;
            }
        }
    }

private:
    int m_rows, m_cols, m_num;
    std::vector<OLNode<T>*> rhead; // 行头指针
    std::vector<OLNode<T>*> chead; // 列头指针
};
```

4.2 扩展线性表—广义表

4.2.1 广义表的定义与性质

> 广义表L是由n>=0个元素组成的有序序列，记为$L=(a_1,\ldots,a_n)$

其中：

1. a_i 可以是**原子**（单个数据元素）或**广义表**（也称为子表）。
2. 广义表的定义是递归的。
3. **表头 (Head)**：若广义表不为空，则第一个元素 a_1 为表头。表头可以是原子或广义表。
4. **表尾 (Tail)**：除表头外，其余元素组成的广义表 $(a_2,\ldots,a_n)$ 为表尾。表尾一定是一个广义表。
5. **举例**：对于 $L=(a,(b,c),d)$，表头是 `a`，表尾是 `((b, c), d)`。

4.2.2 广义表的存储表示

广义表通常采用链式存储结构，最常用的是**头尾链表表示法**。

- 表节点分为两类：
  1. **原子节点**：标志位为0，用于存储原子值。
  2. **表节点**：标志位为1，包含两个指针域：`hp` 指向表头，`tp` 指向表尾。

4.2.3 广义表的操作

- **取表头** `Head(L)`：返回广义表 L 的第一个元素。
- **取表尾** `Tail(L)`：返回广义表 L 中除去表头后剩余元素组成的广义表。
- **求表的长度** `Length(L)`：返回广义表 L 中元素的个数（顶层）。
- **求表的深度** `Depth(L)`：返回广义表 L 中括号嵌套的最大层数。

C++ 实现示例：广义表（头尾链表）

```cpp
#include <iostream>

// 广义表节点类型定义
enum NodeType { ATOM, LIST };

// 广义表节点结构
struct GNode {
    NodeType tag; // 标志位: ATOM为原子, LIST为子表
    union {
        char atom; // 原子节点的值
        struct {
            GNode* hp; // 指向表头
            GNode* tp; // 指向表尾
        } ptr;
    };
};

// 取表头
GNode* GListHead(GNode* L) {
    if (!L || L->tag == ATOM) {
        return nullptr; // 空表或原子没有表头
    }
    return L->ptr.hp;
}

// 取表尾
GNode* GListTail(GNode* L) {
    if (!L || L->tag == ATOM) {
        return nullptr; // 空表或原子没有表尾
    }
    return L->ptr.tp;
}

// 求广义表的深度（递归实现）
```

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

    int max_depth = 0;
    for (GNode* p = L; p; p = p->ptr.tp) {
        int sub_depth = GListDepth(p->ptr.hp); // 递归求子表深度
        if (sub_depth > max_depth) {
            max_depth = sub_depth;
        }
    }
    return max_depth + 1;
}

// 求广义表的长度
int GListLength(GNode* L) {
    if (!L) return 0;
    int length = 0;
    GNode* p = L;
    while(p) {
        length++;
        p = p->ptr.tp;
    }
    return length;
}
```

上一章 第3章 受限线性表：栈，队列和串
下一章 第5章 树与二叉树