数据结构 02

2025 年 3 月 6 日

考纲内容：

- **栈和队列：** 栈的结构特性、基本操作及在顺序存储结构和链式存储结构上基本运算的实现，队列的结构特性、基本操作及在顺序存储结构和链式存储结构上基本运算的实现，栈和队列的基本应用。
- **数组和广义表：** 数组的基本概念和存储结构，广义表的定义和存储结构。

1 栈和队列

1.1 栈

1.1.1 栈的基本概念

栈（Stack）的定义：栈是一种线性表，但限定在表的一端（称为栈顶，top）进行插入和删除。另一端称为栈底（bottom）。

栈的特点是：**后进先出**（Last In First Out, LIFO）。

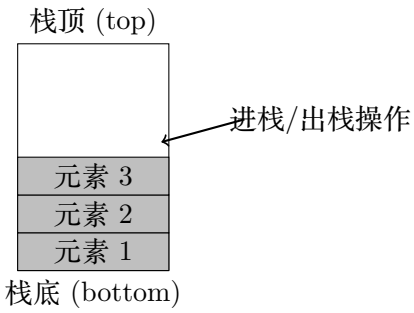

图 1: 栈的结构示意图

1.1.2 栈的基本操作

栈的基本操作包括：

- InitStack(&S)：初始化栈，构造一个空栈 S
- StackEmpty(S)：判断栈是否为空
- Push(&S, x)：进栈，将元素 x 压入栈 S
- Pop(&S, &x)：出栈，弹出栈顶元素并通过 x 返回
- GetTop(S, &x)：读栈顶元素，通过 x 返回栈顶元素（不改变栈）
- ClearStack(&S)：清空栈

1.1.3 栈的顺序存储结构

顺序栈通常使用一维数组和一个指向栈顶的指针来实现：

```
#define MaxSize 100  // 栈的最大容量
typedef struct {
    ElemType data[MaxSize];  // 存放栈中元素的数组
    int top;                 // 栈顶指针
} SqStack;
```

栈顶指针 top 的含义：

- 初始化时，top = -1
- 进栈操作：先 top++，再存入元素
- 出栈操作：先取出栈顶元素，再 top--

栈的基本操作实现：

1. 初始化

```
void InitStack(SqStack *S) {
    S->top = -1;
}
```

2. 判断栈空

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

3. 进栈操作

```
bool Push(SqStack *S, ElemType x) {
    if(S->top == MaxSize-1)  // 栈满
        return false;
    S->top++;
    S->data[S->top] = x;
    return true;
}
```

4. 出栈操作

```
bool Pop(SqStack *S, ElemType *x) {
    if(S->top == -1)  // 栈空
        return false;
    *x = S->data[S->top];
    S->top--;
    return true;
}
```

5. 获取栈顶元素

```
bool GetTop(SqStack S, ElemType *x) {
    if(S.top == -1)  // 栈空
        return false;
    *x = S.data[S.top];
    return true;
}
```

1.1.4 栈的链式存储结构

链栈通常使用单链表实现，且规定所有操作都在链表的头部进行：

```
typedef struct LinkNode {
    ElemType data;
    struct LinkNode *next;
} LinkNode, *LinkStack;
```

链栈的基本操作实现：

1. 初始化

```
void InitStack(LinkStack *S) {
    *S = NULL;
}
```

2. 判断栈空

```
bool StackEmpty(LinkStack S) {
    return S == NULL;
}
```

3. 进栈操作

```
void Push(LinkStack *S, ElemType x) {
    LinkNode *p = (LinkNode *)malloc(sizeof(LinkNode));
    p->data = x;
    p->next = *S;
    *S = p;
}
```

4. 出栈操作

```
bool Pop(LinkStack *S, ElemType *x) {
    if(*S == NULL)
        return false;
    LinkNode *p = *S;
    *x = p->data;
    *S = p->next;
    free(p);
```

```
    return true;
}
```

5. 获取栈顶元素

```
bool GetTop(LinkStack S, ElemType *x) {
    if(S == NULL)
        return false;
    *x = S->data;
    return true;
}
```

1.2 队列

1.2.1 队列的基本概念

队列（Queue）的定义：队列是一种线性表，限定在表的一端（称为队尾，rear）进行插入，在另一端（称为队头，front）进行删除。

队列的特点是：先进先出（First In First Out, FIFO）。

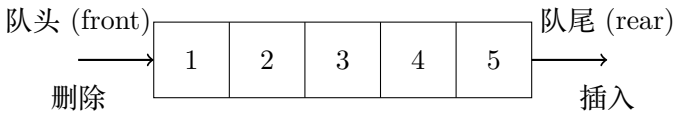

图 2: 队列的结构示意图

1.2.2 队列的基本操作

队列的基本操作包括：

- InitQueue(&Q)：初始化队列，构造一个空队列 Q
- QueueEmpty(Q)：判断队列是否为空
- EnQueue(&Q, x)：入队，将元素 x 加入队列 Q
- DeQueue(&Q, &x)：出队，删除队头元素并通过 x 返回
- GetHead(Q, &x)：读取队头元素，通过 x 返回队头元素（不改变队列）

1.2.3 队列的顺序存储结构

顺序队列通常使用一维数组和两个指针（队头指针 front 和队尾指针 rear）来实现：

```
#define MaxSize 100
typedef struct {
    ElemType data[MaxSize];
    int front, rear;
```

```
} SqQueue;
```

简单队列在多次入队出队操作后可能会出现”假溢出”现象（队列实际未满，但队尾指针已达数组尾部）。为解决这个问题，通常采用循环队列实现。

循环队列的特点：

- 队列元素在内存中的空间是环状的
- 初始时，front = rear = 0
- 入队：rear = (rear+1) % MaxSize
- 出队：front = (front+1) % MaxSize

判断队列空和队列满的条件：

- 队空：front == rear
- 队满：(rear+1) % MaxSize == front（牺牲一个存储单元）

循环队列的基本操作实现：

1. 初始化

```
void InitQueue(SqQueue *Q) {
    Q->front = Q->rear = 0;
}
```

2. 判断队空

```
bool QueueEmpty(SqQueue Q) {
    return Q.front == Q.rear;
}
```

3. 入队操作

```
bool EnQueue(SqQueue *Q, ElemType x) {
    if((Q->rear+1) % MaxSize == Q->front)  // 队满
        return false;
    Q->data[Q->rear] = x;
    Q->rear = (Q->rear+1) % MaxSize;
    return true;
}
```

4. 出队操作

```
bool DeQueue(SqQueue *Q, ElemType *x) {
    if(Q->front == Q->rear)  // 队空
        return false;
    *x = Q->data[Q->front];
    Q->front = (Q->front+1) % MaxSize;
    return true;
}
```

5. 获取队头元素

```
bool GetHead(SqQueue Q, ElemType *x) {
    if(Q.front == Q.rear)  // 队空
        return false;
    *x = Q.data[Q.front];
    return true;
}
```

1.2.4 队列的链式存储结构

链式队列通常用带头结点的单链表实现，front 指向队头结点，rear 指向队尾结点：

```
typedef struct LinkNode {
    ElemType data;
    struct LinkNode *next;
} LinkNode;

typedef struct {
    LinkNode *front, *rear;
} LinkQueue;
```

链式队列的基本操作实现：

1. 初始化（带头结点）

```
void InitQueue(LinkQueue *Q) {
    Q->front = Q->rear = (LinkNode *)malloc(sizeof(LinkNode));
    Q->front->next = NULL;
}
```

2. 判断队空

```
bool QueueEmpty(LinkQueue Q) {
    return Q.front == Q.rear;
}
```

3. 入队操作

```
void EnQueue(LinkQueue *Q, ElemType x) {
    LinkNode *p = (LinkNode *)malloc(sizeof(LinkNode));
    p->data = x;
    p->next = NULL;
    Q->rear->next = p;
    Q->rear = p;
}
```

4. 出队操作

```
bool DeQueue(LinkQueue *Q, ElemType *x) {
```

```
    if(Q->front == Q->rear)  // 队空
        return false;
    LinkNode *p = Q->front->next;
    *x = p->data;
    Q->front->next = p->next;
    if(Q->rear == p)  // 最后一个结点出队
        Q->rear = Q->front;
    free(p);
    return true;
}
```

1.3 栈和队列的应用

1.3.1 栈的应用

1. 括号匹配问题：检查括号是否匹配，例如”(()[()])” 是匹配的，而”([)]” 不匹配。
2. 表达式求值：利用栈实现中缀表达式转后缀表达式，再计算后缀表达式的值。
3. 递归实现：利用栈实现递归，避免使用系统栈。
4. 函数调用：程序中的函数调用使用栈来保存返回地址和局部变量。
5. 浏览器的前进/后退功能：利用两个栈来实现。

Example 1. 使用栈实现括号匹配的算法：

```
bool bracketMatch(char str[]) {
    Stack S;
    InitStack(&S);
    for(int i = 0; str[i] != '\0'; i++) {
        if(str[i] == '(' || str[i] == '[' || str[i] == '{') {
            Push(&S, str[i]);
        } else if(str[i] == ')' || str[i] == ']' || str[i] == '}') {
            if(StackEmpty(S))
                return false;
            char topChar;
            Pop(&S, &topChar);
            if(str[i] == ')' && topChar != '(')
                return false;
            if(str[i] == ']' && topChar != '[')
                return false;
            if(str[i] == '}' && topChar != '{')
                return false;
        }
    }
    return StackEmpty(S);
}
```

1.3.2 队列的应用

1. 层次遍历：广度优先搜索（BFS）算法中使用队列。
2. 消息缓冲区：在消息传递系统中作为消息的缓冲区。
3. 打印机任务队列：管理打印任务。
4. 操作系统中的作业调度：先来先服务（FCFS）调度算法。
5. 计算机网络中的数据包队列：路由器中的数据包队列。

Example 2. 使用队列实现二叉树层次遍历的算法：

```
void levelOrder(BiTree T) {
    Queue Q;
    InitQueue(&Q);
    if(T != NULL)
        EnQueue(&Q, T);
    while(!QueueEmpty(Q)) {
        BiTree p;
        DeQueue(&Q, &p);
        printf("%d␣", p->data);
        if(p->lchild != NULL)
            EnQueue(&Q, p->lchild);
        if(p->rchild != NULL)
            EnQueue(&Q, p->rchild);
    }
}
```

2 数组和广义表

2.1 数组

2.1.1 数组的基本概念

数组（Array）的定义：数组是由 n 个相同类型的数据元素构成的有限序列，每个数据元素称为一个数组元素。在内存中，数组占用一块连续的存储空间。

数组的特点：

- 数组中的所有元素具有相同的数据类型
- 数组元素在内存中按照一定的顺序连续存储
- 可以通过下标（索引）直接访问任何一个数组元素

一维数组可表示为 $A[0:n-1]$ 或 $A[1:n]$，其中 $A[i]$ 表示第 i 个元素。

2.1.2 数组的存储结构

数组的基本存储策略：随机存取（Random Access），通过公式计算元素的存储位置。

一维数组的存储　假设一维数组 $A[0:n-1]$ 的起始地址为 $base$，每个元素占用 $size$ 个存储单元，则数组元素 $A[i]$ 的存储地址计算公式为：

$$Loc(A[i]) = base + i \times size \tag{1}$$

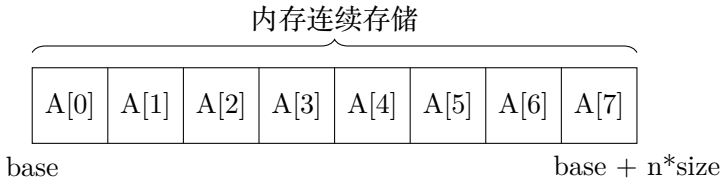

图 3: 一维数组的存储结构示意图

多维数组的存储　对于二维数组 $A[0:m-1][0:n-1]$，有两种常见的存储方式：

行优先存储：先存储第 0 行，再存储第 1 行，依此类推。

$$Loc(A[i][j]) = base + (i \times n + j) \times size \tag{2}$$

列优先存储：先存储第 0 列，再存储第 1 列，依此类推。

$$Loc(A[i][j]) = base + (j \times m + i) \times size \tag{3}$$

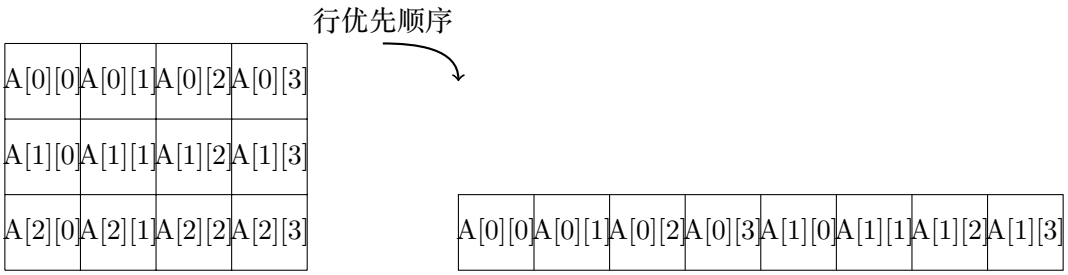

图 4: 二维数组的行优先存储示意图

对于更高维的数组，存储地址计算公式可以类推得到。

2.1.3　特殊矩阵的压缩存储

为了节省存储空间，对于一些特殊矩阵，可以采用压缩存储的方式。

对称矩阵　对称矩阵(Symmetric Matrix)：若对于 n 阶方阵 A 中的任意元素 $a_{ij} = a_{ji}\ (i, j = 1, 2, ..., n)$，则称 A 为对称矩阵。

对称矩阵只需存储主对角线和下（或上）三角区域的元素，共需存储 $\frac{n(n+1)}{2}$ 个元素。

一维数组存储映射公式：

$$k = \begin{cases} \frac{i(i-1)}{2} + j - 1, & \text{当} i \geq j \text{(存储下三角)} \\ \frac{j(j-1)}{2} + i - 1, & \text{当} i < j \text{(存储上三角)} \end{cases} \tag{4}$$

三角矩阵　三角矩阵（Triangular Matrix）：上三角矩阵的下三角部分（不含主对角线）的元素全为同一常数，下三角矩阵的上三角部分（不含主对角线）的元素全为同一常数。

对角矩阵　对角矩阵（Diagonal Matrix）：主对角线两侧各有不超过 s 条对角线上的元素可能非零，其余元素全为 0 的矩阵，也称为带状矩阵。

稀疏矩阵　稀疏矩阵（Sparse Matrix）：矩阵中非零元素的个数远远少于矩阵元素总数，并且非零元素的分布没有规律。

通常采用三元组表示法存储稀疏矩阵：

```
typedef struct {
    int row;      // 行号
    int col;      // 列号
    ElemType val; // 值
} Triple;

typedef struct {
    Triple data[MAXSIZE+1]; // 非零元素的三元组表
    int rows, cols;         // 矩阵的行数、列数
    int nums;               // 非零元素的个数
} SparseMatrix;
```

2.1.4　矩阵的基本运算

矩阵的转置　矩阵转置是将矩阵 A 的行与列互换得到矩阵 B，即 $b_{ij} = a_{ji}$。

普通矩阵转置算法：

```
void TransposeMatrix(SparseMatrix A, SparseMatrix *B) {
    B->rows = A.cols; B->cols = A.rows; B->nums = A.nums;
    if(B->nums > 0) {
        int q = 1;
        for(int col = 0; col < A.cols; col++) {
            for(int p = 1; p <= A.nums; p++) {
                if(A.data[p].col == col) {
                    B->data[q].row = A.data[p].col;
                    B->data[q].col = A.data[p].row;
                    B->data[q].val = A.data[p].val;
                    q++;
                }
            }
        }
    }
}
```

矩阵的乘法　两个矩阵相乘，左矩阵的列数必须等于右矩阵的行数。

普通矩阵乘法算法：

```
void MultiplyMatrix(int A[][N], int B[][P], int C[][P], int m, int n, int p) {
    for(int i = 0; i < m; i++) {
        for(int j = 0; j < p; j++) {
            C[i][j] = 0;
            for(int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

2.2　广义表

2.2.1　广义表的定义

广义表（Generalized List）的定义：广义表是一种非线性的数据结构，其元素可以是原子（不可分割的数据元素）或者是广义表。

用 LS 表示一个广义表，n 是表的长度，则其形式定义为：

$$LS = (a_1, a_2, \ldots, a_n) \tag{5}$$

其中，a_i 可以是原子，也可以是广义表。如果 a_i 是广义表，则称为 LS 的子表。

广义表的基本特点：

- 广义表的元素可以是原子，也可以是广义表
- 广义表可以为空表，此时 $n = 0$
- 广义表可以嵌套定义
- 广义表的长度是指第一层元素的个数
- 广义表的深度是指嵌套的最大层数

Example 3. 广义表示例：

1. $A = ()$：空表，长度为 *0*，深度为 *1*
2. $B = (a, b, c)$：长度为 *3*，深度为 *1*
3. $C = (a, (b, c), d)$：长度为 *3*，深度为 *2*
4. $D = (a, (b, (c, d), e), f)$：长度为 *3*，深度为 *3*
5. $E = ((a, b), (c, d), (e, f))$：长度为 *3*，深度为 *2*

2.2.2 广义表的存储结构

头尾链表存储表示 每个结点包含三个域：

- 标志域（Tag）：区分原子结点和表结点
- 结点值域：
  - 对于原子结点，值域存储原子的值
  - 对于表结点，值域是指向子表的第一个元素的指针（Head）
- 指针域（Tail）：指向当前结点所在表中的下一个元素

```
typedef enum {ATOM, LIST} ElemTag;  // ATOM=0：原子，LIST=1：子表
typedef struct GLNode {
    ElemTag tag;               // 标志域
    union {                    // 值域
        AtomType atom;         // tag=ATOM时，存储原子值
        struct {               // tag=LIST时
            struct GLNode *hp; // 指向表头
            struct GLNode *tp; // 指向表尾
        } ptr;
    };
} GLNode, *GList;
```

扩展线性链表存储表示 每个结点有三个域：

- 标志域（Tag）：区分原子结点和表结点
- 数据域：
  - 对于原子结点，数据域存储原子的值
  - 对于表结点，数据域是指向子表的指针
- 指针域（Next）：指向下一个元素

```
typedef enum {ATOM, LIST} ElemTag;  // ATOM=0：原子，LIST=1：子表
typedef struct GLNode {
    ElemTag tag;                   // 标志域
    union {                        // 值域
        AtomType atom;             // tag=ATOM时，存储原子值
        struct GLNode *sublist;    // tag=LIST时，指向子表
    } val;
    struct GLNode *next;           // 指向下一个元素
} GLNode, *GList;
```

2.2.3 广义表的基本操作

广义表的基本操作包括:

- 求广义表的长度（第一层元素的个数）
- 求广义表的深度（嵌套的最大层数）
- 复制广义表
- 创建广义表

求广义表的深度

```
int GListDepth(GList 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 = 0;
    int dep;
    GList p = L->val.sublist;  // p指向子表

    while(p) {
        dep = GListDepth(p);
        if(dep > max) max = dep;
        p = p->next;
    }

    return max + 1;  // 当前层深度+1
}
```

复制广义表

```
void CopyGList(GList *T, GList L) {
    *T = NULL;
    if(!L) return;

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

    if(L->tag == ATOM) {
        (*T)->val.atom = L->val.atom;
    } else {
        CopyGList(&((*T)->val.sublist), L->val.sublist);
    }

    CopyGList(&((*T)->next), L->next);
}
```