数据结构 01

2025 年 2 月 26 日

1 绪论

1.1 引言

数据结构是计算机科学中的一门基础课程，它研究数据的组织方式及其操作，对于程序设计、算法设计和计算机系统设计都有着重要的影响。本章将介绍数据结构的基本概念，包括数据、数据元素、数据结构、数据类型、抽象数据类型等，同时也会讨论算法及其分析方法。

1.2 数据及相关概念

1.2.1 数据的概念

- **数据**（Data）：是对客观事物的符号表示，在计算机科学中，是指所有能输入到计算机中并被计算机程序处理的符号的总称。
- 数据是计算机程序加工的原料，是信息的载体。
- 数据可以是数值型（如 1, 2, 3.14 等）、文本型（如"Hello"）、图形型、音频型等多种形式。

Example 1. 学生信息系统中的学生姓名、学号、成绩等都是数据。天气预报系统中的温度、湿度、气压等也是数据。

1.2.2 数据元素的概念

- **数据元素**（Data Element）：是数据的基本单位，通常作为一个整体进行考虑。
- 也称为记录（Record）、节点（Node）或实体（Entity）。
- 一个数据元素可由若干个数据项（Data Item）组成。

Example 2. 在学生信息系统中，每个学生的完整信息（包括姓名、学号、成绩等）构成一个数据元素。在图书管理系统中，每本书的信息（包括书名、作者、*ISBN* 等）是一个数据元素。

1.2.3 数据项的概念

- **数据项**（Data Item）：是数据的最小单位，是不可分割的基本单位。
- 数据项是组成数据元素的基本单位。

Example 3. 学生信息中的姓名是一个数据项，学号是一个数据项，成绩是一个数据项。

1.3 数据结构

1.3.1 数据结构的定义

- **数据结构**（Data Structure）：是相互之间存在一种或多种特定关系的数据元素的集合。
- 数据结构包括三个方面：
    1. 数据的逻辑结构
    2. 数据的存储结构（物理结构）
    3. 数据的操作

1.3.2 数据的逻辑结构

- **逻辑结构**（Logical Structure）：指数据元素之间的逻辑关系，与数据的存储无关。
- 主要分为以下四类：
  1. **线性结构**：元素之间是一对一的关系。如线性表、栈、队列等。
  2. **树形结构**：元素之间是一对多的关系。如树、二叉树等。
  3. **图形结构**：元素之间是多对多的关系。如图、网络等。
  4. **集合结构**：元素之间除了同属一个集合外，没有其他关系。

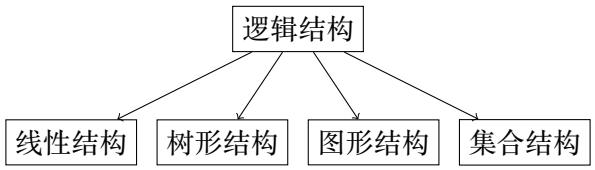


图 1: 数据的逻辑结构分类

1.3.3 数据的存储结构

- **存储结构**（Storage Structure）：指数据在计算机中的表示（又称物理结构）。
- 存储结构是逻辑结构在计算机中的映射，通过存储单元之间的邻接关系来反映数据元素之间的逻辑关系。
- 主要分为以下四类：
  1. **顺序存储结构**：使用一组地址连续的存储单元依次存储数据元素。
  2. **链式存储结构**：使用一组任意的存储单元存储数据元素，每个元素还存储指向相关元素的指针。
  3. **索引存储结构**：在存储数据的同时，建立附加的索引表，以便快速访问。
  4. **散列存储结构**：根据数据元素的关键字直接计算其存储地址。

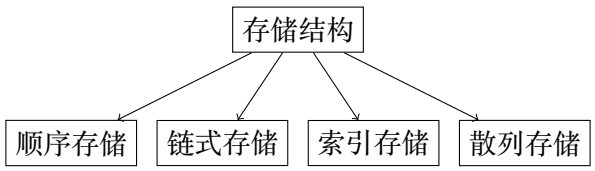


图 2: 数据的存储结构分类

1.4 数据类型与抽象数据类型

1.4.1 数据类型

- **数据类型**（Data Type）：是一组性质相同的值的集合及定义在此集合上的一组操作的总称。
- 数据类型可以分为两类：
  1. **原子类型**（基本类型）：不可再分的类型，如整数、实数、字符等。
  2. **结构类型**：由多个类型组合而成，如数组、结构体等。
- 在编程语言中，声明变量时必须指定数据类型，如 int、float、char 等。

1.4.2 抽象数据类型

- **抽象数据类型**（Abstract Data Type, ADT）：是一个数学模型及定义在该模型上的一组操作，它与存储结构的实现无关。
- ADT 由两部分组成：
  1. 数据对象的抽象描述
  2. 对数据对象的操作的抽象描述
- ADT 的表示方法：

```
ADT 抽象数据类型名 {
    数据对象：<数据对象的定义>
    数据关系：<数据关系的定义>
    基本操作：<基本操作的定义>
}
```

Example 4 (抽象数据类型示例)**.** 以”线性表”为例，其 *ADT* 可表示为：

```
ADT List {
    数据对象：
        D = {a_i | a_i   ElemSet, i = 1, 2, ..., n, n   0}
    数据关系：
        R = {<a_i, a_{i+1}> | a_i, a_{i+1}   D, i = 1, 2, ..., n-1}
    基本操作：
        InitList(&L)   // 初始化线性表
        Length(L)      // 求表长
        GetElem(L, i, &e)  // 获取第i个元素
        LocateElem(L, e)   // 查找元素e的位置
        Insert(&L, i, e)   // 在位置i插入元素e
        Delete(&L, i, &e)  // 删除位置i的元素
        ...
}
```

1.5 算法及其分析

1.5.1 算法的概念

- **算法**（Algorithm）：是解决特定问题的一系列操作的有限序列。
- 算法的五个基本特性：
  1. **有穷性**：算法必须在有限步骤内结束。
  2. **确定性**：算法的每一步骤必须有确定的含义，不能有歧义。
  3. **可行性**：算法的每一步操作都必须是可行的，即能够通过已经实现的基本操作执行有限次来实现。
  4. **输入**：算法可以有零个或多个输入。
  5. **输出**：算法必须有一个或多个输出。

1.5.2 算法的描述

算法可以通过多种方式描述：

1. **自然语言**：使用日常语言描述算法步骤。
2. **流程图**：使用图形符号表示算法的流程。
3. **伪代码**：介于自然语言和程序设计语言之间的描述方式。
4. **程序设计语言**：如 C、Java 等。

Example 5 (简单算法示例：顺序查找)**.** 伪代码描述：

```
Algorithm SequentialSearch(A[0...n-1], key):
    i = 0
    while i < n and A[i] != key do
        i = i + 1
    if i < n then
        return i  // 找到元素，返回位置
    else
        return -1 // 未找到元素，返回-1
```

1.5.3 算法的分析

算法分析的两个主要方面：

1. **正确性分析**：证明算法是否能正确地解决问题。
2. **效率分析**：评估算法的时间复杂度和空间复杂度。
    - **时间复杂度**：算法执行所需的时间。
    - **空间复杂度**：算法执行所需的存储空间。

渐进符号 用于表示算法时间复杂度的常用符号：

- **O(大 O)**：表示上界，如 $O(n^2)$ 表示算法的执行时间不超过 n^2 的常数倍。
- **Ω(大 Omega)**：表示下界，如 $\Omega(n)$ 表示算法的执行时间至少是 n 的常数倍。
- **Θ(大 Theta)**：表示确界，如 $\Theta(n)$ 表示算法的执行时间恰好是 n 的常数倍。

常见的时间复杂度 按照效率从高到低排序：

- $O(1)$：常数时间，与输入规模无关。
- $O(\log n)$：对数时间，如二分查找。
- $O(n)$：线性时间，如顺序查找。
- $O(n \log n)$：线性对数时间，如归并排序、快速排序。
- $O(n^2)$：平方时间，如冒泡排序、插入排序。
- $O(n^3)$：立方时间，如某些矩阵运算。
- $O(2^n)$：指数时间，如穷举法。

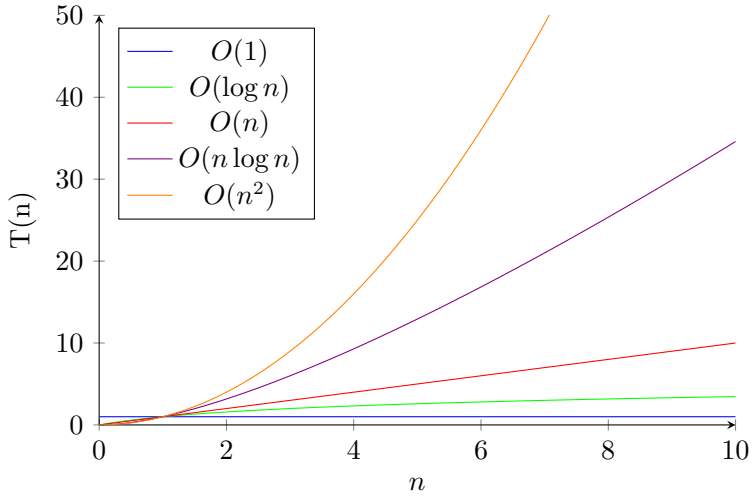


图 3: 常见时间复杂度函数的增长趋势

算法效率分析实例

Example 6 (顺序查找算法的时间复杂度分析)**.** 顺序查找算法的最坏情况：要查找的元素在数组的最后或不存在，需要比较 n 次，时间复杂度为 $O(n)$。最好情况：要查找的元素在数组的第一个位置，只需比较 1 次，时间复杂度为 $O(1)$。平均情况：假设查找任意位置的概率相等，则平均需要比较 $(n+1)/2$ 次，时间复杂度仍为 $O(n)$。

1.6 数据结构的应用

数据结构在计算机科学和实际应用中具有广泛的应用：

- **数据库系统**：利用各种数据结构组织和存储数据，实现高效的数据管理。
- **操作系统**：利用队列管理进程，利用树结构管理文件系统等。
- **编译器**：利用栈实现表达式求值，利用树结构表示语法分析等。
- **搜索引擎**：利用倒排索引等数据结构实现高效的信息检索。
- **图形处理**：利用图和树等结构描述图像和几何对象的关系。
- **人工智能**：利用各种数据结构表示知识和实现搜索算法。

1.7 本章小结

本章介绍了数据结构的基本概念，包括数据、数据元素、数据项、数据的逻辑结构和存储结构、数据类型和抽象数据类型等。同时，也讨论了算法的概念、算法的描述方法以及算法分析的基本方法，特别是时间复杂度和空间复杂度的概念和分析方法。这些基础知识是学习后续各种具体数据结构和算法的必要前提。

2 线性表

2.1 线性表的基本概念

2.1.1 线性表的定义

- **线性表**（Linear List）是具有相同数据类型的 n 个数据元素的有限序列。

- 其中，n 为表长度，当 n=0 时称为空表。
- 线性表中元素的位序（Position）是从 1 开始的。
- 若用 L 表示线性表，则记为：$L = (a_1, a_2, \ldots, a_n)$
- 线性表中元素的特点：
    1. 除第一个元素外，每个元素有且仅有一个直接前驱。
    2. 除最后一个元素外，每个元素有且仅有一个直接后继。

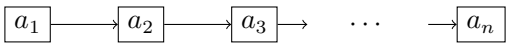


图 4: 线性表的逻辑结构

2.1.2 线性表的抽象数据类型描述

线性表的抽象数据类型（ADT）描述如下：

```
ADT LinearList {
    数据对象:
        D = {a_i | a_i   ElementSet, i = 1, 2, ..., n, n   0}
    数据关系:
        R = {<a_i, a_{i+1}> | a_i, a_{i+1}   D, i = 1, 2, ..., n-1}
    基本操作:
        InitList(&L)            // 初始化线性表
        DestroyList(&L)         // 销毁线性表
        ClearList(&L)           // 清空线性表
        ListEmpty(L)            // 判断线性表是否为空
        ListLength(L)           // 获取线性表长度
        GetElem(L, i, &e)       // 获取第i个元素
        LocateElem(L, e, compare)  // 查找元素e
        PriorElem(L, cur_e, &pre_e)  // 获取元素cur_e的前驱
        NextElem(L, cur_e, &next_e)  // 获取元素cur_e的后继
        ListInsert(&L, i, e)    // 在第i个位置插入元素e
        ListDelete(&L, i, &e)   // 删除第i个元素
        ListTraverse(L, visit)  // 遍历线性表
}
```

2.2 线性表的顺序表示

2.2.1 顺序表的定义

- 顺序表是用一组地址连续的存储单元依次存储线性表中的数据元素，使得逻辑上相邻的两个元素在物理位置上也相邻。
- 顺序表是线性表的顺序存储结构。
- 优点：

1. 随机访问特性：可以在 $O(1)$ 时间内访问任意位置的元素。
2. 无须为表中元素之间的逻辑关系增加额外的存储空间。

- 缺点：

1. 插入和删除操作需要移动大量元素，效率较低。
2. 存储空间需要预先分配，可能会出现内存浪费或溢出的情况。

2.2.2 顺序表的存储结构

静态分配 在 C 语言中，可以用数组来实现顺序表的静态分配：

```
#define MAXSIZE 100  // 顺序表的最大长度

typedef struct {
    ElemType data[MAXSIZE];  // 数组，存储数据元素
    int length;              // 当前长度
} SqList;
```

动态分配 在 C 语言中，可以用指针和动态内存分配来实现顺序表的动态分配：

```
typedef struct {
    ElemType *data;  // 指向动态分配数组的指针
    int maxSize;     // 顺序表的最大容量
    int length;      // 顺序表的当前长度
} SqList;
```

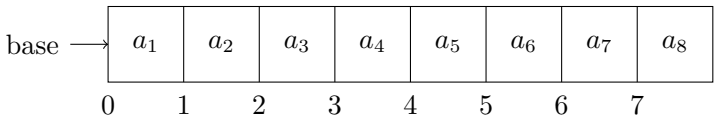


图 5: 顺序表的存储结构

2.2.3 顺序表的基本运算

顺序表的初始化

```
Status InitList_Sq(SqList &L) {
    // 为顺序表分配一个大小为MAXSIZE的数组空间
    L.data = new ElemType[MAXSIZE];
    if (!L.data) return ERROR;  // 存储分配失败
    L.length = 0;               // 空表长度为0
    L.maxSize = MAXSIZE;        // 最大长度为MAXSIZE
    return OK;
}
```

顺序表的取值 根据位置 i 获取相应位置数据元素 e 的值，时间复杂度为 $O(1)$。

```
Status GetElem_Sq(SqList L, int i, ElemType &e) {
    if (i < 1 || i > L.length) return ERROR;  // i值不合法
    e = L.data[i-1];                          // 第i-1个单元存储第i个元素
    return OK;
}
```

顺序表的查找 在顺序表 L 中查找值为 e 的元素，若找到，则返回其位序；否则，返回 0。

```
int LocateElem_Sq(SqList L, ElemType e) {
    for (int i = 0; i < L.length; i++)
        if (L.data[i] == e) return i+1;  // 返回位序
    return 0;  // 查找失败，返回0
}
```

顺序查找的平均时间复杂度为 $O(n)$。

顺序表的插入 在顺序表 L 的第 i 个位置插入新元素 e。

```
Status ListInsert_Sq(SqList &L, int i, ElemType e) {
    if (i < 1 || i > L.length + 1) return ERROR;  // i值不合法
    if (L.length >= L.maxSize) return ERROR;      // 当前存储空间已满

    // 将第i个位置后的元素向后移动
    for (int j = L.length; j >= i; j--)
        L.data[j] = L.data[j-1];

    L.data[i-1] = e;  // 插入新元素
    L.length++;       // 表长增加1
    return OK;
}
```

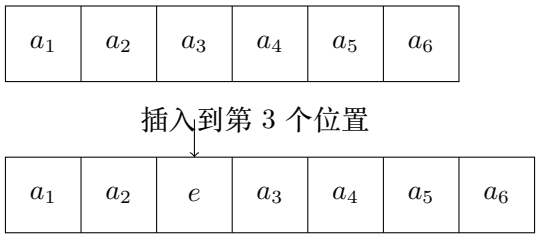


图 6: 顺序表的插入操作示意图

顺序表插入操作的时间复杂度分析：

- 最好情况：插入到表尾，无需移动元素，时间复杂度为 $O(1)$。
- 最坏情况：插入到表头，需要移动所有元素，时间复杂度为 $O(n)$。
- 平均情况：假设插入位置均匀分布，平均需要移动 $n/2$ 个元素，时间复杂度为 $O(n)$。

顺序表的删除　删除顺序表 L 中第 i 个元素，并用 e 返回其值。

```
Status ListDelete_Sq(SqList &L, int i, ElemType &e) {
    if (i < 1 || i > L.length) return ERROR;  // i值不合法

    e = L.data[i-1];  // 保存被删除元素

    // 将第i个位置后的元素向前移动
    for (int j = i; j < L.length; j++)
        L.data[j-1] = L.data[j];

    L.length--;       // 表长减少1
    return OK;
}
```

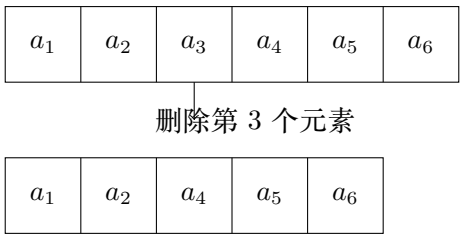


图 7: 顺序表的删除操作示意图

顺序表删除操作的时间复杂度分析：

- 最好情况：删除表尾元素，无需移动元素，时间复杂度为 $O(1)$。
- 最坏情况：删除表头元素，需要移动所有元素，时间复杂度为 $O(n)$。
- 平均情况：假设删除位置均匀分布，平均需要移动 $(n-1)/2$ 个元素，时间复杂度为 $O(n)$。

顺序表的排序　常用的排序算法包括冒泡排序、选择排序、插入排序、希尔排序、快速排序、堆排序、归并排序等。

以冒泡排序为例：

```
void BubbleSort(SqList &L) {
    for (int i = 0; i < L.length - 1; i++) {
        bool flag = false;  // 标记本轮是否有交换
        for (int j = 0; j < L.length - 1 - i; j++) {
            if (L.data[j] > L.data[j+1]) {  // 相邻元素比较
                // 交换元素
                ElemType temp = L.data[j];
                L.data[j] = L.data[j+1];
                L.data[j+1] = temp;
                flag = true;  // 本轮有交换
            }
        }
        if (!flag) break;  // 本轮无交换，说明已经有序
```

```
    }
}
```

2.3 线性表的链式表示

2.3.1 链表的基本概念

- 链表是线性表的链式存储结构，它使用一组任意的存储单元来存储线性表中的数据元素。
- 链表中的每个元素（称为结点）在存储数据的同时，还存储指向下一个结点的指针（或引用）。
- 链表不要求逻辑上相邻的元素在物理位置上也相邻。
- 优点：
  1. 插入和删除操作不需要移动元素，时间复杂度为 $O(1)$（不考虑查找时间）。
  2. 充分利用计算机内存空间，不会出现内存浪费。
- 缺点：
  1. 不支持随机访问，访问特定位置的元素需要 $O(n)$ 的时间复杂度。
  2. 需要额外的存储空间来存储指针信息。

2.3.2 单链表

单链表的节点结构 在 C 语言中，单链表的节点结构如下：

```
typedef struct LNode {
    ElemType data;          // 数据域
    struct LNode *next;     // 指针域，指向下一个节点
} LNode, *LinkList;         // LinkList为指向LNode的指针类型
```

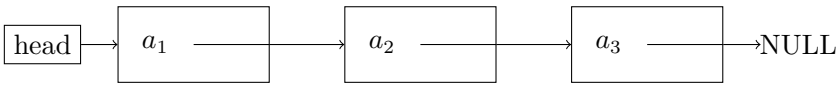


图 8: 单链表示意图

单链表的创建方式 1. 头插法（逆序建立）

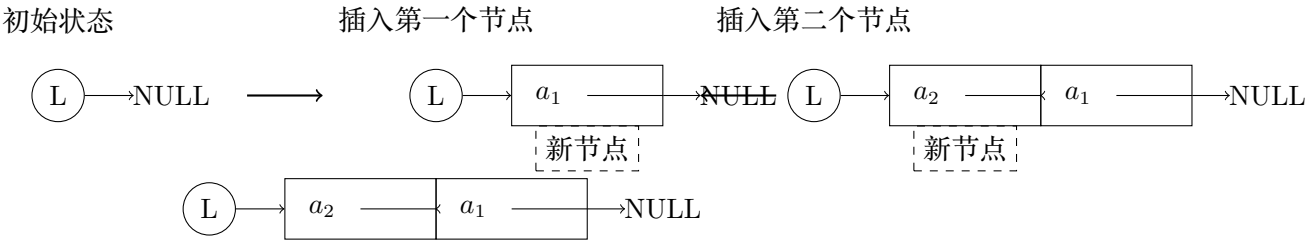
```
LinkList CreateList_H() {
    LinkList L = (LinkList)malloc(sizeof(LNode));  // 创建头节点
    L->next = NULL;                                // 初始为空链表

    int n;
    printf("请输入元素个数：");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        LinkList p = (LinkList)malloc(sizeof(LNode));  // 创建新节点
```

```
        scanf("%d", &(p->data));                          // 输入元素值
        p->next = L->next;   // 将新节点插入表头
        L->next = p;
    }

    return L;
}
```



头插法特点: 新节点总是插入到链表头部，导致元素顺序与输入顺序相反

图 9: 单链表头插法示意图（逆序建立）

2. 尾插法（正序建立）

```
LinkList CreateList_T() {
    LinkList L = (LinkList)malloc(sizeof(LNode));  // 创建头节点
    L->next = NULL;                                // 初始为空链表
    LNode *r = L;                                  // r指向尾节点，初始时指向头节点

    int n;
    printf("请输入元素个数: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        LinkList p = (LinkList)malloc(sizeof(LNode));  // 创建新节点
        scanf("%d", &(p->data));                       // 输入元素值
        p->next = NULL;      // 新节点暂时是尾节点
        r->next = p;         // 将新节点插入到尾节点之后
        r = p;               // r指向新的尾节点
    }

    return L;
}
```

单链表的基本操作 1. 获取链表长度

```
int GetLength(LinkList L) {
    int length = 0;
```

```
    LinkList p = L->next;  // 从第一个节点开始

    while (p) {
        length++;
        p = p->next;
    }

    return length;
}
```

2. 按位置查找节点

```
LNode* GetElem(LinkList L, int i) {
    if (i < 0) return NULL;  // i不合法

    int j = 0;
    LinkList p = L;  // 从头节点开始，j=0表示头节点

    while (p && j < i) {
        p = p->next;
        j++;
    }

    return p;  // 返回第i个节点
}
```

3. 按值查找节点

```
LNode* LocateElem(LinkList L, ElemType e) {
    LinkList p = L->next;  // 从第一个节点开始

    while (p && p->data != e)
        p = p->next;

    return p;  // 找到返回节点指针，否则返回NULL
}
```

4. 插入节点在第 i 个位置插入值为 e 的新节点。

```
Status ListInsert(LinkList &L, int i, ElemType e) {
    LinkList p = GetElem(L, i-1);  // 查找第i-1个节点
    if (!p) return ERROR;          // i-1位置不存在

    LinkList s = (LinkList)malloc(sizeof(LNode));  // 创建新节点
    s->data = e;                                   // 赋值

    s->next = p->next;  // 新节点指向原第i个节点
```

```
    p->next = s;          // 第i-1个节点指向新节点

    return OK;
}
```

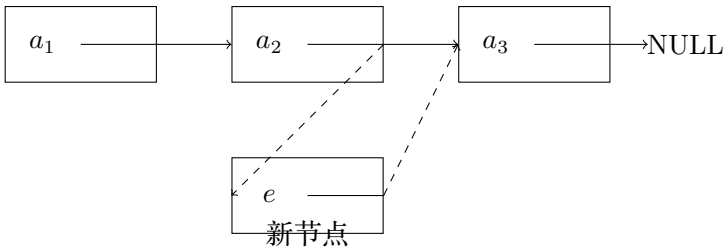


图 10: 单链表的插入操作

5. 删除节点删除第 i 个节点。

```
Status ListDelete(LinkList &L, int i, ElemType &e) {
    LinkList p = GetElem(L, i-1);  // 查找第i-1个节点
    if (!p || !p->next) return ERROR;  // i-1位置不存在或第i个节点不存在

    LinkList q = p->next;     // q指向待删除节点
    e = q->data;              // 保存被删除节点的值

    p->next = q->next;        // 第i-1个节点指向第i+1个节点
    free(q);                  // 释放被删除节点的空间

    return OK;
}
```

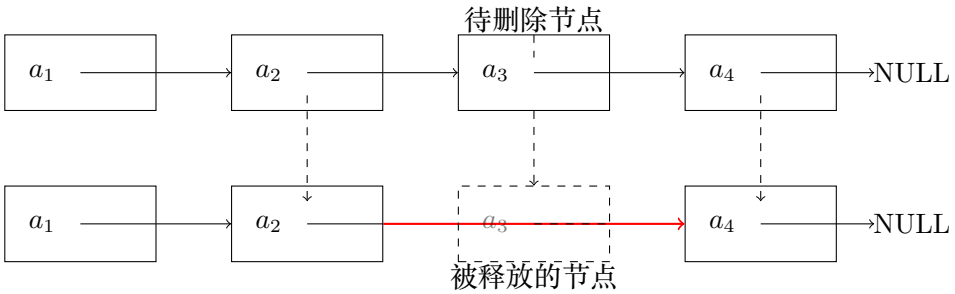


图 11: 单链表的删除操作示意图

2.3.3 循环链表

循环链表是一种特殊的链表，其中最后一个节点的指针不是 NULL，而是指向链表中的第一个节点或头节点，形成一个环。

单循环链表 单循环链表是单链表的一种变形，其最后一个节点的 next 指针指向头节点，形成一个环。

单循环链表的优点：

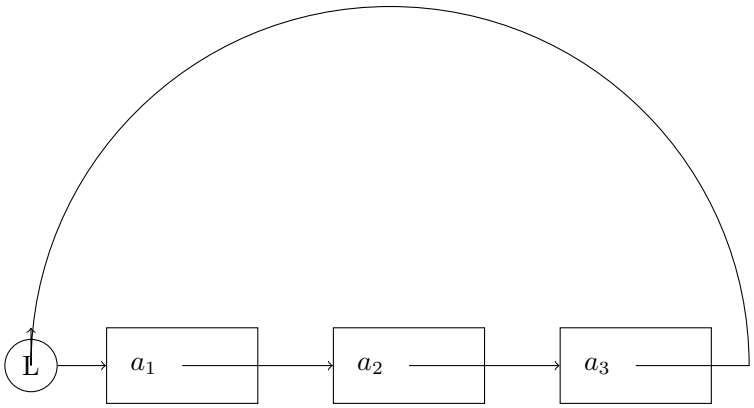


图 12: 单循环链表示意图

- 从表中任一节点出发都能遍历整个链表
- 适合于需要循环处理的数据结构，如约瑟夫问题
- 在尾节点插入和删除操作更方便

循环链表的基本操作 1. 初始化循环链表

```
Status InitList(LinkList &L) {
    L = (LinkList)malloc(sizeof(LNode));  // 创建头节点
    if (!L) return ERROR;                 // 内存分配失败
    L->next = L;                          // 头节点指向自己，形成环
    return OK;
}
```

2. 判断循环链表是否为空

```
Status ListEmpty(LinkList L) {
    if (L->next == L)  // 如果头节点指向自己，表示为空
        return TRUE;
    else
        return FALSE;
}
```

3. 在循环链表尾部插入节点

```
Status InsertLast(LinkList &L, ElemType e) {
    LinkList p = L;

    // 查找尾节点
    while (p->next != L)
        p = p->next;

    // 创建新节点
    LinkList s = (LinkList)malloc(sizeof(LNode));
    if (!s) return ERROR;
```

```
    s->data = e;

    // 插入到尾部
    s->next = L;      // 新节点指向头节点
    p->next = s;      // 原尾节点指向新节点

    return OK;
}
```

约瑟夫问题的应用实例 约瑟夫问题（约瑟夫环）：n 个人围成一圈，从第 k 个人开始报数，数到 m 的人出列，然后从下一个人开始重新报数，继续数到 m 的人出列，以此类推，直到所有人都出列，求出列顺序。

```
void Joseph(int n, int k, int m) {
    // 创建包含n个节点的循环链表
    LinkList L = (LinkList)malloc(sizeof(LNode));
    LinkList p = L;

    // 创建n个节点
    for (int i = 1; i <= n; i++) {
        LinkList s = (LinkList)malloc(sizeof(LNode));
        s->data = i;  // 节点数据为人的编号
        p->next = s;
        p = s;
    }
    p->next = L->next;  // 最后一个节点指向第一个节点，形成循环

    // 找到第k个人
    p = L->next;
    for (int i = 1; i < k; i++)
        p = p->next;

    // 依次找到第m个人并"出列"
    LinkList q;
    while (p->next != p) {  // 当只剩一个节点时结束
        // 找到第m-1个节点
        for (int i = 1; i < m-1; i++)
            p = p->next;

        q = p->next;  // q指向第m个节点，即要删除的节点
        printf("%d ", q->data);  // 输出出列人的编号
        p->next = q->next;  // 删除q节点
        free(q);
        p = p->next;  // p指向删除节点的下一个节点
    }
```

```
    printf("%d\n", p->data);  // 输出最后一个人的编号
    free(p);
}
```

2.3.4 双向链表

双向链表（Double Linked List）是一种更复杂的链表，它的每个节点有两个指针域，一个指向前驱节点，一个指向后继节点。

双向链表的节点结构

```
typedef struct DuLNode {
    ElemType data;           // 数据域
    struct DuLNode *prior;   // 前驱指针
    struct DuLNode *next;    // 后继指针
} DuLNode, *DuLinkList;
```

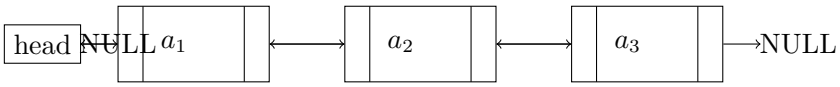


图 13: 双向链表示意图

双向链表的初始化

```
Status InitDuList(DuLinkList &L) {
    L = (DuLinkList)malloc(sizeof(DuLNode));
    if (!L) return ERROR;

    L->prior = NULL;
    L->next = NULL;

    return OK;
}
```

双向链表的插入操作 在双向链表中第 i 个位置之前插入元素 e。

```
Status ListInsert_DuL(DuLinkList &L, int i, ElemType e) {
    // 查找第i个节点
    DuLinkList p = L;
    int j = 0;

    while (p && j < i) {
        p = p->next;
        j++;
    }

    if (!p || j > i) return ERROR;  // i值不合法
```

```
    // 创建新节点
    DuLinkList s = (DuLinkList)malloc(sizeof(DuLNode));
    if (!s) return ERROR;
    s->data = e;

    // 插入操作
    s->prior = p->prior;  // 设置新节点的前驱
    s->next = p;          // 设置新节点的后继
    p->prior->next = s;   // 设置前驱节点的后继
    p->prior = s;         // 设置p的前驱

    return OK;
}
```

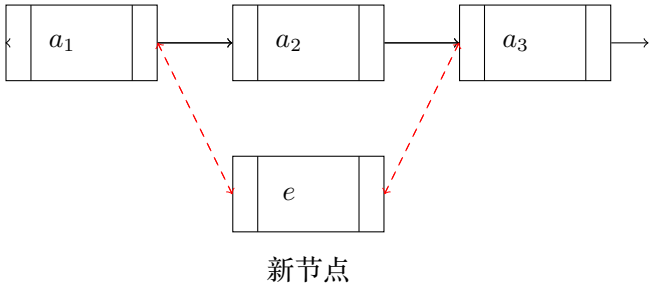


图 14: 双向链表的插入操作示意图

双向链表的删除操作　删除双向链表中的第 i 个节点。

```
Status ListDelete_DuL(DuLinkList &L, int i, ElemType &e) {
    // 查找第i个节点
    DuLinkList p = L;
    int j = 0;

    while (p && j < i) {
        p = p->next;
        j++;
    }

    if (!p || j > i) return ERROR;  // i值不合法

    e = p->data;  // 保存被删除节点的值

    // 删除操作
    p->prior->next = p->next;  // 前驱节点的后继指向后继节点

    if (p->next)  // 若p不是最后一个节点
```

```
        p->next->prior = p->prior;  // 后继节点的前驱指向前驱节点

    free(p);  // 释放节点

    return OK;
}
```

双向循环链表　双向循环链表是结合了循环链表和双向链表的特点，其中第一个节点的前驱指向最后一个节点，最后一个节点的后继指向第一个节点。

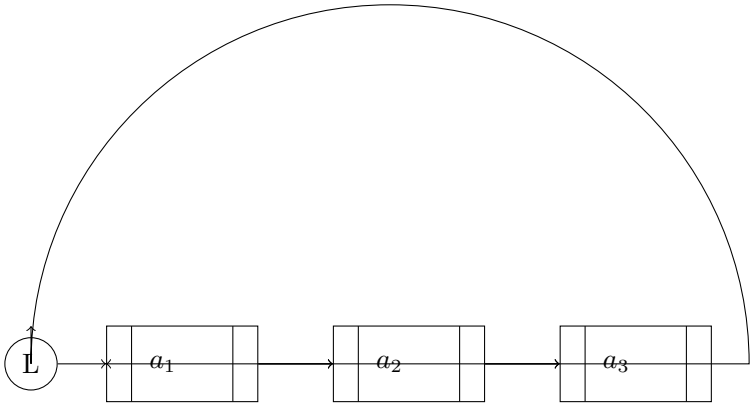


图 15: 双向循环链表示意图

双向循环链表的优点：

- 可以双向遍历，提高了访问效率
- 可以从任意节点开始遍历整个链表
- 某些特定操作（如删除给定节点）不需要再查找其前驱节点

2.4　线性表各种实现方式的比较

操作	顺序表	单链表	循环链表	双向链表
按位查找	$O(1)$	$O(n)$	$O(n)$	$O(n)$
按值查找	$O(n)$	$O(n)$	$O(n)$	$O(n)$
插入操作	$O(n)$	$O(1)^*$	$O(1)^*$	$O(1)^*$
删除操作	$O(n)$	$O(1)^*$	$O(1)^*$	$O(1)^*$
空间效率	高	低	低	最低

表 1: 线性表各种实现方式的时间复杂度比较 (* 不考虑查找时间)

存储空间分析

- **顺序表**：只需要存储数据元素本身和少量的管理信息（如表长）。
- **单链表**：每个节点除了数据域外，还需要一个指针域，额外空间开销约为数据空间的一倍。
- **双向链表**：每个节点需要两个指针域，额外空间开销约为数据空间的两倍。

适用场合分析

- **顺序表适用场合**：
  1. 表长变化不大，或可预先确定表长上限
  2. 经常按位序访问数据
  3. 数据元素占用存储空间较小
  4. 对存储空间利用率要求较高
- **单链表适用场合**：
  1. 表长变化较大或无法预知
  2. 经常插入和删除数据
  3. 不需要按位置随机访问
- **循环链表适用场合**：
  1. 需要循环遍历数据的场合
  2. 需要频繁在首尾节点间切换的场合
  3. 适用于约瑟夫环等问题
- **双向链表适用场合**：
  1. 需要双向遍历数据的场合
  2. 需要查找给定节点的前驱节点
  3. 经常在两个方向上移动指针的场合

2.5 线性表的应用实例

2.5.1 多项式表示

多项式 $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ 可以用线性表表示：

顺序表表示多项式 可以用一维数组按照指数递增的顺序存储各项系数。这种方式简单，但对于稀疏多项式（大多数系数为 0）会浪费空间。

链表表示多项式 对于稀疏多项式，更适合用链表表示，每个节点存储一个非零项的系数和指数。

```
typedef struct PolyNode {
    float coef;             // 系数
    int expon;              // 指数
    struct PolyNode *next;  // 指向下一项的指针
} PolyNode, *PolyList;
```

多项式的加法 利用链表表示的多项式，可以实现高效的加法操作：

```
PolyList PolyAdd(PolyList PA, PolyList PB) {
    PolyList PC = (PolyList)malloc(sizeof(PolyNode));  // 创建结果多项式的头节点
    PC->next = NULL;
    PolyList p = PA->next, q = PB->next, r = PC;  // p、q分别指向PA、PB的第一个节点，r指向PC的尾节点

    while (p && q) {
        if (p->expon == q->expon) {  // 指数相等，系数相加
            float sum = p->coef + q->coef;
            if (sum != 0.0) {  // 和不为0时才创建新节点
                PolyList s = (PolyList)malloc(sizeof(PolyNode));
                s->coef = sum;
                s->expon = p->expon;
                s->next = NULL;
                r->next = s;
                r = s;
            }
            p = p->next;
            q = q->next;
        } else if (p->expon < q->expon) {  // 将指数较小的项插入结果多项式
            PolyList s = (PolyList)malloc(sizeof(PolyNode));
            s->coef = p->coef;
            s->expon = p->expon;
            s->next = NULL;
            r->next = s;
            r = s;
            p = p->next;
        } else {  // p->expon > q->expon
            PolyList s = (PolyList)malloc(sizeof(PolyNode));
            s->coef = q->coef;
            s->expon = q->expon;
            s->next = NULL;
            r->next = s;
            r = s;
            q = q->next;
        }
    }

    // 将剩余项复制到结果多项式
    while (p) {
        PolyList s = (PolyList)malloc(sizeof(PolyNode));
        s->coef = p->coef;
        s->expon = p->expon;
        s->next = NULL;
```

```
        r->next = s;
        r = s;
        p = p->next;
    }

    while (q) {
        PolyList s = (PolyList)malloc(sizeof(PolyNode));
        s->coef = q->coef;
        s->expon = q->expon;
        s->next = NULL;
        r->next = s;
        r = s;
        q = q->next;
    }

    return PC;
}
```

2.6 线性表的实际应用

2.6.1 线性表在数据库中的应用

关系型数据库中的表本质上是由记录组成的线性表，每条记录可视为线性表中的一个元素。数据库索引的实现也依赖于各种线性表结构，如 B 树、B+ 树等。

2.6.2 线性表在操作系统中的应用

操作系统中的进程调度队列、内存管理的空闲块链表等都是线性表的典型应用。

2.6.3 线性表在编译器设计中的应用

编译器中的符号表、词法分析时的标识符表等都可以使用线性表实现。

2.7 线性表的高级应用

2.7.1 静态链表

静态链表是借助数组来描述线性表的链式存储，它既具有顺序表随机访问的特点，又具有链表插入删除方便的特点。

```
// 静态链表结构定义
#define MAXSIZE 1000
typedef struct {
    ElemType data;
    int next;    // 下一个元素的数组下标
} SLinkList[MAXSIZE];

// 初始化备用链表
```

```
void InitSpace(SLinkList &space) {
    for (int i = 0; i < MAXSIZE - 1; i++) {
        space[i].next = i + 1;
    }
    space[MAXSIZE - 1].next = 0;  // 目前静态链表为空，最后一个元素的next为0
}

// 分配结点
int Malloc_SL(SLinkList &space) {
    int i = space[0].next;
    if (i)
        space[0].next = space[i].next;
    return i;
}

// 回收结点
void Free_SL(SLinkList &space, int k) {
    space[k].next = space[0].next;
    space[0].next = k;
}
```

2.7.2 线性表的应用——文本编辑器

文本编辑器可以将文本的每一行作为线性表中的一个元素，使用双向链表实现可在 O(1) 时间内实现向上、向下翻页等操作。对于每一行文本，可再用一个线性表表示其中的字符。

```
// 文本编辑器的数据结构（伪代码）
typedef struct Line {
    char *text;           // 该行的文本内容
    int length;           // 该行的长度
    struct Line *prev;    // 指向前一行
    struct Line *next;    // 指向后一行
} Line, *TextEditor;

// 在当前行后插入新行
Status InsertLine(TextEditor &T, Line *current, char *text) {
    Line *p = (Line *)malloc(sizeof(Line));
    if (!p) return ERROR;

    p->text = strdup(text);  // 复制文本
    p->length = strlen(text);

    p->prev = current;
    p->next = current->next;
```

```
    if (current->next)
        current->next->prev = p;
    current->next = p;

    return OK;
}
```