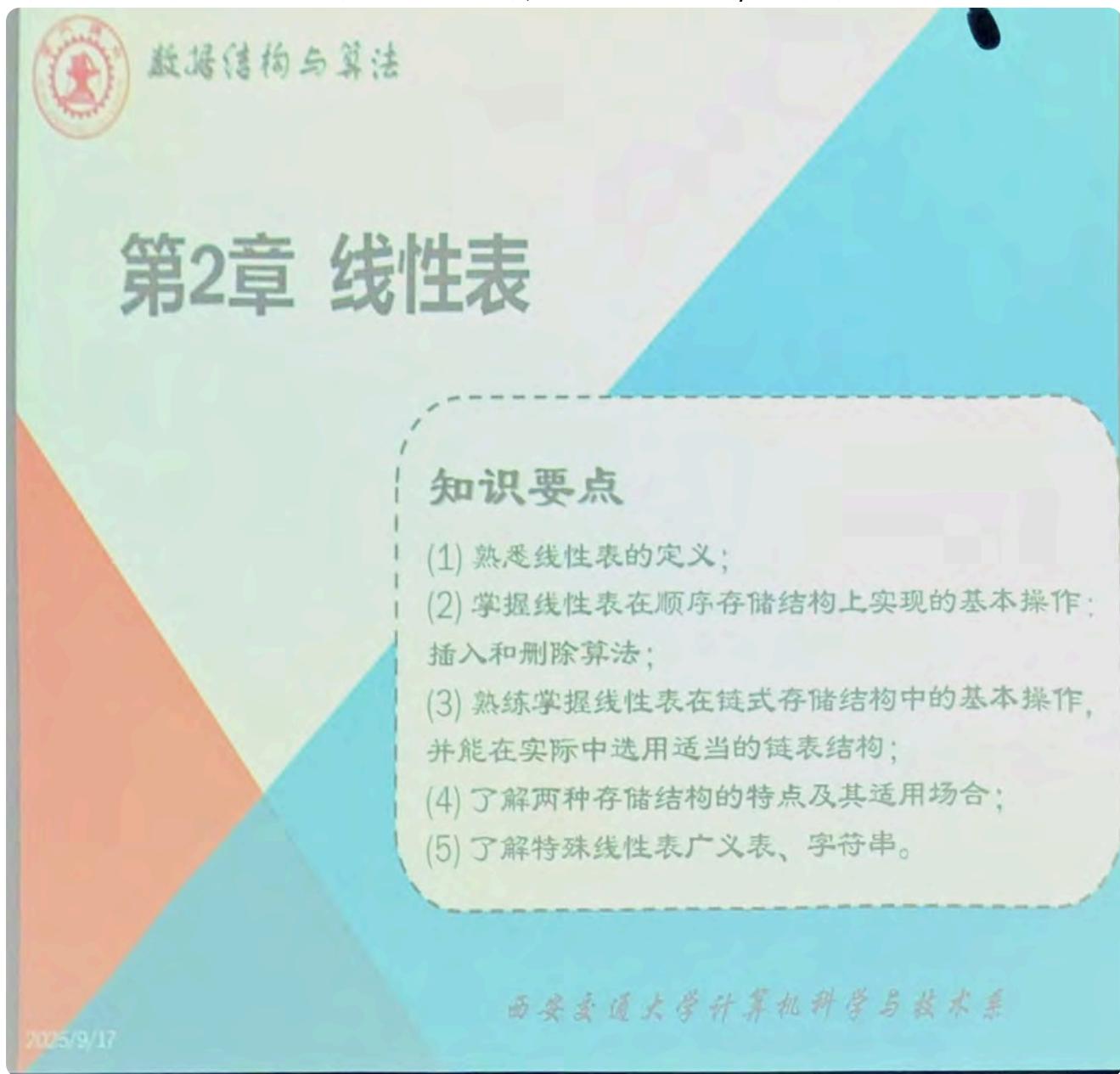


# 第2章 线性表

#数据结构与算法

## 概述

线性表是数据结构的基础，是一种线性结构。本章重点掌握：线性表的定义、两种存储结构（顺序与链式）、基本操作（插入、删除等），以及应用如队列/栈（★）。图像参考：



### 需要掌握的内容：

- 线性表的定义。
- 两种存储结构（顺序存储 vs. 链式存储）。
- 线性表的若干操作（插入、删除、查找等）。
- 队列 (Queue) / 栈 (Stack) 的基本概念（★）。

## 2.1 线性表的定义

- **定义**: 线性表是由**长度为 n** 的一组**节点** (数据元素) 组成的**有限序列**。
  - 形式:  $L = (a_1, a_2, \dots, a_n)$ , 其中  $n \geq 0$  ( $n=0$  时为空表)。
  - **长度**: 数据元素的数目  $n$ 。
- **形式化描述**: 长度为  $n$  的线性表是一种数据结构  $L = (D, R)$ , 其中:
  - $D$  是数据元素集 ( $\{a_1, a_2, \dots, a_n\}$ )。
  - $R$  是关系集, 定义  $D$  中元素的顺序关系 ( $a_i$  是  $a_{i-1}$  的后继,  $a_{i+1}$  是  $a_i$  的前驱)。
- **特点**: 元素间是一对一的逻辑关系 (除首尾元素外, 每个元素有唯一前驱和后继)。

## 2.2 线性表的顺序存储结构

### 2.2.1 顺序存储结构

- **定义**: 用一组**连续的存储单元**存储线性表数据, 相邻元素的**物理位置相邻** (以空间换时间)。
- **优点**: 随机访问高效 ( $O(1)$  通过下标)。
- **缺点**: 插入/删除需移动元素 ( $O(n)$  时间), 大小固定 (易溢出)。
- **顺序表**: 采用顺序存储的线性表称为**顺序表 (Sequential List)**。

- 示意图：



## 线性表顺序存储结构的定义：

```
Define MaxLen <线性表可能的最大长度>
ElemType elements[MaxLen] ; // 数据元素值
```

```
Typedef struct sequenlist{
    ElemType elements[MaxLen] ; // 数据元素值
    int last; // 线性表中当前的元素个数
} SqList;
```

## 2.2.2 数据存储结构的实现

顺序表的典型操作包括：

- 插入操作：**在位置  $i$  插入元素，需后移元素（时间  $O(n)$ ）。
  - 追加操作：**在表尾添加元素（时间  $O(1)$ ，若有空间）。
  - 删除操作：**删除位置  $i$  的元素，前移元素填补（时间  $O(n)$ ）。
- 示例：**已知一个有序顺序表  $L$ ，编写算法删除重复数据元素。
    - 要求：**时间复杂度  $O(n)$ ，空间复杂度  $O(1)$ （原地修改）。
    - 思路：**单指针遍历，从第二个元素开始比较；相同则跳过，不同则复制并前进。

```
// 假设 SeqList 是顺序表结构体，包含 elements[] 数组和 length
int DeleteMoreElement(SeqList &L) {
    if (L.length <= 1) return 0; // 空表或单元素无需处理
    int cnt = 0; // 有效元素计数（从0开始）
    for (int k = 1; k < L.length; k++) { // 从第二个元素开始
        if (L.elements[k] != L.elements[cnt]) {
            L.elements[cnt + 1] = L.elements[k];
            cnt++;
        }
    }
    L.length = cnt;
}
```

```

if (L.elements[cnt] != L.elements[k]) {
    cnt++; // 移动有效位置
    L.elements[cnt] = L.elements[k]; // 复制新元素
}
}

L.length = cnt + 1; // 更新长度
return 0; // 或返回删除的重复数
}

```

- **分析：**遍历一次  $O(n)$ ，无额外空间。

## 2.3 线性表的链式存储结构

### 2.3.1 单链表

- **定义：**每个节点包含**数据域 (data)** 和**指针域 (next)**，用指针连接非连续节点（以时间换空间）。
- **链表分类：**详见 静态链表与动态链表。
- **优点：**动态插入/删除高效 ( $O(1)$ )，若有指针)。
- **缺点：**访问需顺序遍历 ( $O(n)$ )，无随机访问。
- **示意图：**

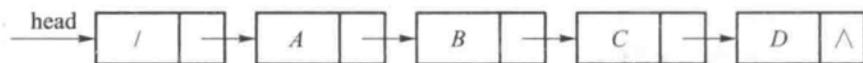


图 2-7 单链表结构示例图



图 2-8 空单链表结构示例图

#### 算法 2.6 单链表的结点类定义

```

template < class Elemt >
class Link {
public:
    Elemt element; // 当前结点的数据元素
    Link * next; // 指向下一结点的指针
    Link( const Elemt& item, Link * nextval = NULL ) { element = item; next = nextval; }
    Link( Link * nextval = NULL ) { next = nextval; }
};

```

截屏



## 数据结构与算法

### 单链表上所实现的基本操作：1. 链表的建立算法（一）

```

LinkList *CreateQueenLList()
{ struct node *p, *s, *head; // head为单链表的头
    head = NULL; p = head;      // p指向链表的最后一个结点
    scanf(ch);
    while(ch != 输入结束字符){
        if(new(s, 1) == NULL){
            printf("没有空间！");
            return( NULL );
        }
        s->data = ch;
        if( p == NULL) head = s;
        else p->next = s;
        p = s;
        scanf(ch);
    }
    p->next = NULL;
    return( head );
}

```



## 数据结构与算法

### 单链表上所实现的基本操作：1. 链表的建立算法（二）

```

LinkList *CreateStackLList()
{ head = NULL; // 建立一个头指针为head的堆栈式单链表
    scanf(ch);
    while(ch != 输入结束字符){
        if(new(s, 1) == NULL){
            printf("没有足够的空间！");
            return( NULL );
        }
        s->data = ch;
        s->next = head;
        head = s;
        scanf(ch);
    }
    return( head );
}

```

**注意：**头指针 (head) 和工作指针 (p) 不算附加空间，已包含在链表中。

- **插入/删除效率比较：**

- **队列算法（第一个方法）：**从头遍历，适合后插。
- **栈算法（第二个方法）：**从尾遍历，效率更高 ( $O(1)$  尾插)。

## 2.3.2 双向链表

- **定义**: 每个节点包含**两个指针域** (prev 和 next) 和**一个数据域 (data)**:
- **优点**: 支持双向遍历，插入/删除更灵活（无需遍历到头）。
- **缺点**: 每个节点多一个指针，空间开销大。
- **基本操作**:
  - 插入：调整前后节点的指针。
  - 删除：断开前后链接，释放节点。
- **作业实现**: 双向链表的基本代码（C++ 示例，使用结构体）。

```
#include <iostream>
using namespace std;

typedef struct DNode {
    int data;
    struct DNode *prev; // 指向前驱
    struct DNode *next; // 指向后继
} DNode, *DLinkedList;

// 初始化空链表
DLinkedList InitList() {
    DLinkedList head = new DNode;
    head->prev = head->next = nullptr;
    head->data = 0; // 头节点数据无关
    return head;
}

// 在位置 i 插入元素 e
int InsertList(DLinkedList L, int i, int e) {
    DNode *p = L;
    int j = 0;
    while (p != nullptr && j < i) {
        p = p->next;
        j++;
    }
    if (j != i || p == nullptr) return 0; // 位置无效

    DNode *s = new DNode;
    s->data = e;
    s->next = p;
    s->prev = p->prev;
    p->prev->next = s;
    p->prev = s;
}
```

```

    return 1;
}

// 删除位置 i 的元素，返回其值
int DeleteList(DLinkList L, int i, int &e) {
    DNode *p = L;
    int j = 0;
    while (p != nullptr && j < i) {
        p = p->next;
        j++;
    }
    if (j != i || p == nullptr) return 0;

    e = p->data;
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    return 1;
}

```

- **分析：**时间  $O(n)$ （需遍历到  $i$ ），空间  $O(1)$  附加（不计节点本身）。

## 2.4 线性表应用举例

### 2.4.1 一元多项式的表示

- **问题：**表示和运算一元多项式（如  $3x^2 + 2x + 1$ ）。
- **为什么用链表：**多项式项稀疏，数组会浪费空间 ( $O(n)$  项 vs. 高次)。
- **存储结构：**单链表，每个节点存指数、系数（按指数降序）。
- **解题思路**（参考 PPT）：
  1. 遍历链表 B 的每个项。
  2. 在链表 A 中查找相同指数项（线性查找  $O(n)$ ）。
  3. 未找到：创建新节点插入（按指数有序）。
  4. 找到：系数相加；若结果为 0，删除节点并释放空间；否则修改系数。
  5. 合并后，按指数排序（可选，插入时维护）。
- **时间复杂度：** $O(m * n)$ ， $m$  和  $n$  为两多项式项数。

### 2.4.2 商品链更新

- **问题：**模拟商品列表（如电商），支持插入新商品、更新价格、删除过期商品。
- **存储结构：**双向链表，便于前后导航。

- **示例操作：**
  - **插入：**在指定位置添加商品节点 (`{id, name, price}`)。
  - **更新：**遍历查找 `id`，修改 `price` ( $O(n)$ )。
  - **删除：**移除过期节点，双向指针快速链接前后。
- **优点：**动态调整列表，无需重分配内存。
- **代码片段（基于双向链表）：**

```
// 假设商品节点
typedef struct Goods {
    int id;
    string name;
    double price;
    struct Goods *prev, *next;
} GoodsNode;

// 更新价格示例
void UpdatePrice(DLinkList L, int id, double newPrice) {
    DNode *p = L->next; // 跳过头
    while (p != nullptr) {
        if (((GoodsNode*)p)->id == id) {
            ((GoodsNode*)p)->price = newPrice;
            return;
        }
        p = p->next;
    }
}
```

## 章节总结

- 顺序存储适合访问密集，链式存储适合修改密集。队列/栈是线性表的特化（后进先出/先进先出）。

[上一章 第1章 基础知识](#)

[下一章 第3章 受限线性表：栈，队列和串](#)