

巍巍交大 百年书香
www.jiaodapress.com.cn
bookinfo@sjtu.edu.cn



策划编辑 李 勇
责任编辑 胡思佳
封面设计 刘文东

数据结构 (C语言版)

数据结构 (C语言版)

主 编 程海军 王晓东 李静宜



上海交通大学出版社
SHANGHAI JIAO TONG UNIVERSITY PRESS



扫描二维码
关注上海交通大学出版社
官方微信

ISBN 978-7-313-33744-3



9 787313 337443 >

定价:65.00元

免费提供
精品教学资料包
服务热线: 400-615-1233
www.xinsijiaocai.com



河南省“十四五”普通高等教育规划教材
河南省本科高校新工科新形态教材



数据结构 (C语言版)

主 编 程海军 王晓东 李静宜



上海交通大学出版社
SHANGHAI JIAO TONG UNIVERSITY PRESS



河南省“十四五”普通高等教育规划教材

河南省本科高校新工科新形态教材



数据结构 (C语言版)

主编 程海军 王晓东 李静宜



上海交通大学出版社
SHANGHAI JIAO TONG UNIVERSITY PRESS

内容提要

本书是河南省本科高校新工科新形态教材、河南省“十四五”普通高等教育规划教材。本书以 C 语言为编程载体,系统讲解数据结构的基本原理、典型结构与核心算法,将抽象的理论知识与实际应用紧密结合,帮助学生在解决真实场景问题的过程中深入理解并掌握数据结构的设计与实现方法。本书包含绪论和 7 个项目,内容涉及顺序表和链表、栈和队列、串和数组、树和二叉树、图、查找、排序。每个项目包含若干任务,每个任务设置学习目标、任务背景、任务描述、设计思路、相关知识、任务实现、动手实践环节,并配有丰富的习题。

本书既适合作为高等院校数据结构课程的教材,也可作为软件开发人员的自学与参考用书。

图书在版编目 (CIP) 数据

数据结构 : C 语言版 / 程海军, 王晓东, 李静宜主
编. -- 上海 : 上海交通大学出版社, 2025. 10.
ISBN 978-7-313-33744-3
I. TP311.12; TP312.8
中国国家版本馆 CIP 数据核字第 2025CA6019 号

数据结构(C 语言版)
SHUJU JIEGOU(C YUYAN BAN)

主 编:程海军 王晓东 李静宜	地 址:上海市番禺路 951 号
出版发行:上海交通大学出版社	电 话:021-64071208
邮政编码:200030	经 销:全国新华书店
印 制:三河市龙大印装有限公司	印 张:19
开 本:850 mm×1 168 mm 1/16	印 次:2025 年 10 月第 1 次印刷
字 数:535 千字	电子书号:ISBN 978-7-89564-508-0
版 次:2025 年 10 月第 1 版	
书 号:ISBN 978-7-313-33744-3	
定 价:65.00 元	

版权所有 侵权必究
告读者:如发现本书有印装质量问题请与印刷厂质量科联系
联系电话:0316-3655788



前言

在新工科蓬勃发展、多学科交叉融合的当下，具有创新实践能力以及对产业发展需求的快速响应能力成为对新工科人才的基本要求。数据结构作为计算机科学与技术专业的核心基础课程，在培育学生计算思维、算法设计能力以及解决复杂工程问题的能力等方面，发挥着不可替代的作用。在新工科背景下，人工智能、大数据、物联网等众多新兴领域蓬勃发展，这些领域的海量数据需要高效的数据组织、存储和管理，而这正是数据结构课程的核心内容。

为了更好地发挥数据结构在新工科人才培养中的作用，打破传统学科界限，培养学生跨学科解决问题的能力，我们编写了本书。在编写本书时，编者注重知识体系的系统性和实用性，改变单一的理论讲解，结合实际应用场景，以项目、任务驱动的方式让学生在解决实际问题的过程中掌握和应用相关知识，全方位、深层次地培育学生的创新思维能力和科学素养。同时，顺应新形态教材的要求，结合传统纸质教材和数字化资源，打造精美的教学课件、完整的代码工程文件、配套的微课讲解视频等，为学生提供多样化的学习资源，充分满足学生不同学习风格与进度的需求。

本书按照数据结构的主要知识体系，将教材编排为绪论和 7 个项目，分别对应数据结构的基本概念、顺序表和链表、栈和队列、串和数组、树和二叉树、图、查找、排序。在每个项目的编写架构中，根据知识体系设置了 1 ~ 2 个实际应用任务。在每个任务中，通过学习目标和具体的任务背景，明确教学重难点和教学目标，生动还原实际问题情境；借助任务描述和设计思路，将实际任务抽象为数据结构问题，并结合数据结构的知识分析解决方案；通过相关知识与任务实现，补充理论知识，分步骤还原任务求解的源代码；设置动手实践和习题，围绕基础知识巩固和能力提高两方面，帮助学生扎实掌握基础知识。这种多环节的项目式设计能让学生沉浸式学习数据结构知识，同时通过模拟实际工程问题的解决流程，提升知识融会贯通的能力和综合实践能力。



本书配备了丰富完备的教学资源，包括教学课件、习题答案、任务源代码、测试用例、微课视频等。其中，源代码和测试用例使用C语言编写，在Dev-C++环境下调试运行。并且，源代码、测试用例、微课视频都以二维码的形式插入文中，在讲授或学习过程中，师生可借助包含编程思路和源代码细节的微课视频，提高知识理解和实际操作的能力。

本书由新乡工程学院程海军、王晓东和李静宜担任主编，新乡工程学院衡欣、吴亚萍、冯歌、王敏、吕娜和司毅洋参与编写工作。

本书在成书过程中虽几经修改，但限于编者的学识水平与能力，书中难免存在不足之处，敬请读者批评指正。

编 者

目录

1

绪论

0.1 数据结构的研究内容	2	0.3 算法和算法分析	13
0.2 基本概念和术语	4	1. 算法的定义	14
1. 数据、数据元素、数据项和数据对象	4	2. 算法的特性	14
2. 数据结构	4	3. 算法设计的要求	14
3. 数据类型	7	4. 算法的时间复杂度	15
4. 抽象数据类型	8	5. 算法的空间复杂度	19
5. 抽象数据类型的描述	8	0.4 小结	20
6. 抽象数据类型的表示与实现	9	0.5 习题	21

项目 1

顺序表和链表

23

任务 1 顺序表实现图书管理系统	24	任务 2 链表实现考试报名管理系统	38
学习目标	24	学习目标	38
任务背景	24	任务背景	38
任务描述	24	任务描述	38
设计思路	25	设计思路	39
相关知识	25	相关知识	39
1. 线性表的定义	25	1. 单链表	39
2. 顺序表的存储表示	26	2. 循环链表	50
3. 顺序表的基本操作	27	3. 双向链表	51
任务实现	31	任务实现	53
1. 顺序表的存储表示	31	1. 链表的存储表示	53
2. 顺序表的主要操作	31	2. 链表的主要操作	54
3. 顺序表的特点	34	3. 链表的特点	56
动手实践	35	动手实践	57
【实践任务】	35	【实践任务】	57
【算法步骤】	35	【算法步骤】	57
【算法实现】	35	【算法实现】	57
【算法分析】	36	【算法分析】	60
习题	36	习题	60
一、选择题	36	一、选择题	60
二、综合题	37	二、综合题	61



项目 2

栈和队列

62

任务 1 浏览器历史记录管理系统

63

学习目标	63
任务背景	63
任务描述	63
设计思路	64
相关知识	64
1. 栈的定义	64
2. 栈的顺序存储和基本操作	65
3. 栈的链式存储和基本操作	69
任务实现	74
1. 数据结构的定义 1	74
2. 主要操作实现	74
3. 主函数逻辑 1	78
动手实践	78
【实践任务】	78
【算法步骤】	78
【算法实现】	78
【算法分析】	79
习题	79
一、选择题	79
二、综合题	80

任务 2 餐厅排队叫号系统

81

学习目标	81
任务背景	81
任务描述	81
设计思路	82
相关知识	82
1. 队列的定义	82
2. 循环队列的存储和基本操作	83
3. 链式队列的存储和基本操作	88
任务实现	93
1. 数据结构的定义 2	93
2. 基本操作实现	93
3. 主函数逻辑 2	95
动手实践	96
【实践任务】	96
【算法步骤】	96
【算法实现】	96
【算法分析】	98
习题	98
一、选择题	98
二、综合题	99

项目 3

串和数组

100

任务 1 简易记事本

101

学习目标	101
任务背景	101
任务描述	101
设计思路	102
相关知识	102
1. 串的定义	102
2. 串的存储结构	105
3. 串的基本操作	106
4. 模式匹配算法	110
任务实现	116
1. 定义串的顺序存储结构	116
2. 记事本的主要操作	116
3. 串的特点	119
动手实践	119
【实践任务】	120
【算法步骤】	120
【算法实现】	120
【算法分析】	121
习题	121
一、选择题	121

任务 2 特殊矩阵的压缩存储

122

学习目标	123
任务背景	123
任务描述	123
设计思路	124
相关知识	124
1. 数组的存储计算	124
2. 特殊矩阵的压缩存储实现	126
任务实现	129
1. 优点	138
2. 缺点	138
动手实践	139
【实践任务】	139
【算法步骤】	139
【算法实现】	139
【算法分析】	141
习题	141
一、选择题	141
二、综合题	142

项目 4

树和二叉树

143

任务 1 构建家谱树	144	习题	166
学习目标	144	一、选择题	166
任务背景	144	二、综合题	167
任务描述	144	任务 2 哈夫曼编码	168
设计思路	145	学习目标	168
相关知识	145	任务背景	168
1. 树和二叉树的定义与性质	145	任务描述	168
2. 树和二叉树的存储结构	152	设计思路	168
3. 树与递归	156	相关知识	169
4. 树的遍历和二叉树的遍历与基本操作	158	1. 哈夫曼树的定义	169
任务实现	163	2. 哈夫曼树和哈夫曼编码的构造过程	170
1. 家谱树的存储表示	163	任务实现	176
2. 家谱树的主要操作	163	动手实践	176
动手实践	165	【实践任务】	177
【实践任务】	165	【算法步骤】	177
【算法步骤】	166	【算法实现】	177
【算法实现】	166	【算法分析】	177
【算法分析】	166	习题	177
		一、选择题	177
		二、综合题	178

项目 5

图

179

任务 1 公路交通网规划	180	任务 2 旅游景点路径规划	204
学习目标	180	学习目标	204
任务背景	180	任务背景	204
任务描述	180	任务描述	205
设计思路	181	设计思路	205
相关知识	181	相关知识	206
1. 图的定义和存储结构	181	1. 迪杰斯特拉算法	206
2. 图的遍历	189	2. 弗洛伊德算法	209
3. 最小生成树	192	任务实现	211
任务实现	196	1. 景点及路线距离的存储表示	211
1. 城市公路线路和辅助结构的存储表示	196	2. 初始化图结构	212
2. 初始化图结构	197	3. 计算各景点间的最短路径	212
3. 选择修建公路	198	4. 全排列景点并计算最短路径	213
动手实践	200	动手实践	214
【实践任务】	200	【实践任务】	214
【算法步骤】	200	【算法步骤】	214
【算法实现】	200	【算法实现】	214
【算法分析】	202	【算法分析】	216
习题	202	习题	216
一、选择题	202	一、选择题	216
二、综合题	203	二、综合题	217



项目 6

查找

218

任务 1 学生信息检索系统

219

学习目标	219
任务背景	219
任务描述	219
设计思路	220
相关知识	220
1. 查找的常用术语	220
2. 顺序查找	221
3. 折半查找	223
4. 分块查找	226
5. 散列查找	228
任务实现	232
1. 预置学生数据集	232
2. 多维度查询	234
动手实践	237
【实践任务】	237
【算法步骤】	237
【算法实现】	237
习题	238
一、选择题	238

二、综合题

239

任务 2 储蓄账户查询系统

240

学习目标	240
任务背景	240
任务描述	240
设计思路	241
相关知识	241
1. 二叉排序树	241
2. 平衡二叉树	245
3. B- 树和 B+ 树	247
任务实现	252
1. 定义结构体	252
2. 存储账户查询系统的各操作函数	252
动手实践	255
【实践任务】	255
【算法步骤】	256
【算法实现】	256
习题	256
一、选择题	256
二、综合题	257

项目 7

排序

259

任务 学生成绩排序系统

260

学习目标	260
任务背景	260
任务描述	260
设计思路	261
相关知识	261
1. 与排序相关的概念	261
2. 直接插入排序	262
3. 折半插入排序	265
4. 希尔排序	266
5. 冒泡排序	268
6. 快速排序	270
7. 简单选择排序	273

8. 堆排序	274
9. 归并排序	278
10. 基数排序	281
任务实现	284
1. 学生成绩排序系统的存储表示	284
2. 学生成绩排序系统的主要操作	285
动手实践	292
【实践任务】	292
【算法步骤】	292
【算法实现】	292
习题	293
一、选择题	293
二、综合题	294

参考文献

296



绪 论



早期计算机的核心任务是高效地处理数值数据。然而，随着计算机技术的飞速发展和应用领域的不断拓展，现代计算机的应用场景早已超越了数值计算的范畴。如今，计算机更多地被应用于处理字符、表格、图像、音频、视频等具有丰富结构的非数值数据。这些数据不仅在数量上呈爆炸式增长，而且在内容上也更加复杂多样。它们之间存在着各种各样的内在联系，如何有效地组织和处理这些数据，成为计算机科学中一个至关重要的问题。数据结构正是为解决这一问题而诞生的学科。本绪论简要介绍有关数据结构的基本概念和算法分析方法。

0.1 数据结构的研究内容

在计算机主要用于数值计算的早期，解决这类问题通常需要经过一系列明确的步骤：首先，从具体问题中抽象出数学模型；接着，设计一个用于求解该数学模型的算法；最后，编写程序并进行测试与调试，直至问题得到解决。在这个过程中，建立数学模型的核心在于分析问题，提取出操作的对象及这些对象之间的关系，并用数学语言进行描述，即建立相应的数学方程。例如，在利用计算机进行桥梁结构应力分析时，需要求解一组线性方程组；而在计算物体的自由落体运动时，其数学模型是一个简单的二阶常微分方程。求解这些数学方程的算法属于计算数学的研究范畴，常见的算法包括牛顿迭代法、欧拉法、有限差分法等。

然而，数据结构（data structure）主要关注的是非数值计算问题。与数值计算不同，非数值计算问题通常无法通过数学方程建立数学模型。下面通过三个实例来说明这一点。

【例 0-1】员工工资管理系统。

在一家大型企业中，人力资源部门使用计算机对全体员工的工资信息进行统一管理。员工的基本信息包括员工编号、姓名、部门、职位、基本工资、奖金和总工资等，如表 0-1 所示。每个员工的基本信息按照不同的员工编号，依次存放在员工工资信息表中，根据需要对这张表进行查找、更新和统计操作。每个员工的基本信息记录按员工编号排列，形成了员工工资信息记录的线性序列，呈现一种线性关系。

表 0-1 员工工资信息表

员 工 编 号	姓 名	部 门	职 位	基 本 工 资	奖 金	总 工 资
1001	张伟	技术部	软件工程师	8 000	2 000	10 000
1002	李娜	市场部	市场专员	6 000	1 500	7 500
1003	王强	人力资源部	招聘专员	7 000	1 000	8 000
1004	赵敏	财务部	会计	7 500	1 200	8 700

诸如此类的线性表结构还有医院的患者病历管理系统、超市的销售记录管理系统等。在这类问题中，计算机处理的对象是各种表，元素之间存在简单的一对一的线性关系。这种数学模型被称为线性的数据结构。

【例 0-2】企业项目管理问题。

在现代企业中，项目管理是确保业务顺利推进的关键环节。一个大型项目通常由多个子任务组成，而这些子任务又可能进一步细分为更小的任务单元。为了高效管理和监控项目进度，企业需要结构化地存储和管理项目的任务。

以一个软件开发项目为例，项目的初始状态是一个整体目标。例如，开发一款移动应用。随着项目的推进，这个整体目标会被分解为多个子任务，如需求分析、界面设计、后端开发和测试。每个子任务又可以进一步细分为更具体的小任务，如需求分析可以细分为市场调研、用户访谈和功能规划。整个项目结构呈现出一种层次化的树结构。

在这棵树（tree）中，根结点代表整个项目的初始目标，而每个分支结点代表一个子任务，叶结点则代表具体的任务单元。从根结点到某个叶结点的路径，代表了项目分解的一个具体路径。

项目管理问题的数学模型就是如何用树结构表示项目的任务分解和层级关系，算法则是任务分配的策略。诸如此类的树结构还包括课程学习计划、家庭族谱等。在这类问题中，计算机处理的对象是树结构，元素之间是一对多的层次关系，施加于对象上的操作包括查找、插入、删除和更新等。这种数学模型被称为树的数据结构。

【例 0-3】社交网络中的最短关系链问题。

在社交网络中，用户之间的关系构成了一个复杂的网络。每个用户可以与其他用户建立联系，形成朋友关系。假设要找到两个用户之间的最短关系链，即通过最少的中间用户将两个人联系起来。例如，用户 A 和用户 B 之间可能没有直接的联系，但通过若干个中间用户，他们可能间接相连。这种问题可以抽象为图的最短路径问题。

如图 0-1 所示，图中的顶点代表用户，无向边代表两个用户之间的朋友关系。边上的权值可以表示关系的紧密程度（权值越小表示关系越紧密）。求解用户 A 到用户 B 的最短关系链，就是要在无向图中找到从 A 点（起点）到 B 点（终点）的多条路径中，各边权值之和最小的路径，即最短路径。

社交网络中最短关系链问题的数学模型就是图结构，算法是求解两点之间的最短路径。诸如此类的图结构还包括物流配送网络、电力传输网络等。在这类问题中，元素之间是多对多的网状关系，施加于对象上的操作包括查找、插入、删除及更新等。这种数学模型被称为图的数据结构。

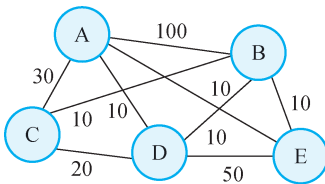


图 0-1 最短路径

通过对上述三个实例的分析可以发现，非数值计算问题的数学模型已经从传统的数学方程转变为更加灵活多样的数据结构，如线性表、树和图等。这些数据结构能够有效地组织和表达数据（data）之间的复杂关系，为解决实际问题提供了强有力的工具。因此，简单地说，数据结构是一门专注于研究非数值计算程序设计中的操作对象，以及这些对象之间的关系和对这些对象操作的学科。

20 世纪 60 年代初期，数据结构的相关内容还分散在操作系统、编译原理等课程中。然而，随着计算机应用的不断拓展，尤其是对复杂数据处理需求的增加，数据结构逐渐成为独立的研



究领域。1968 年, 数据结构作为一门独立课程被正式列入美国部分大学计算机科学系的教学计划。同年, 著名计算机科学家克努特 (D.E.Knuth) 教授发表了《计算机程序设计艺术》, 其第一卷为《基本算法》, 这是第一本系统阐述数据结构基本内容的著作。克努特 (Knuth) 的这部著作不仅奠定了数据结构的理论基础, 还推动了相关研究的深入发展。随后, 随着大型程序和大规模文件系统的出现, 结构化程序设计成为程序设计方法学的主要研究方向。瑞士计算机科学家沃斯 (Niklaus Wirth) 提出了著名的观点 “算法 + 数据结构 = 程序”, 并以此为名出版了相关著作。这一观点深刻地揭示了程序设计的本质, 即通过合理选择数据结构并设计高效的算法来解决问题。

数据结构是计算机科学中一门极具综合性的专业基础课程。其研究内容不仅与计算机硬件密切相关, 尤其是在编码理论、存储装置及数据存取方法等领域, 还与计算机软件的研究深度交融。无论是编译程序还是操作系统, 都不可避免地涉及数据元素 (data element) 在存储器中的分配问题。此外, 在信息检索领域, 如何高效地组织数据以优化查找和存取效率, 也是数据结构研究的重要方向。因此, 数据结构可以被视为一门处于数学、计算机硬件和软件三者交汇点的核心课程。

关于数据结构的研究仍在持续演进。一方面, 针对各专业领域中特定问题的数据结构不断涌现, 为解决复杂场景下的数据组织与处理问题提供了新的思路和方法; 另一方面, 从抽象数据类型角度来探讨数据结构, 正逐渐成为一种新的研究趋势, 并受到越来越多的关注和重视。

◆ 0.2 基本概念和术语

下列概念和术语将在后面的内容中多次出现, 本节先对这些概念和术语赋予确定的含义。

1. 数据、数据元素、数据项和数据对象

数据是客观事物的符号表示, 是所有能够输入计算机中并被程序处理的符号的总称。数学计算中用到的整数和实数, 文本编辑中用到的字符串, 气象数据中的温度和湿度记录, 金融交易中的股票价格和交易时间, 社交媒体中的用户信息和评论内容, 以及物联网设备采集的传感器数据等, 这些信息都通过特定的编码方式被计算机识别和处理。

数据元素是数据的基本单位, 通常在计算机中被视为一个整体进行操作和处理。在某些情况下, 数据元素也被称为元素、记录或实体。数据元素的作用是完整地描述一个具体对象的全部信息。例如, 一个学生的基本信息记录、一个文件系统中的文件目录、一个电商订单中的商品详情或一个社交网络用户的基本资料等, 都可以被视为数据元素。

数据项 (data item) 是组成数据元素的、有独立含义的、不可分割的最小单位。例如, 学生基本信息表中的学号、姓名、性别等都是数据项。

数据对象 (data object) 是性质相同的数据元素的集合, 是数据的一个子集。例如, 自然数数据对象可以表示为集合 $N = \{0, 1, 2, 3, \dots\}$, 小写英文字母字符数据对象可以表示为集合 $C = \{'a', 'b', \dots, 'z'\}$, 而员工基本信息表也可以被视为一个数据对象。由此可见, 无论是无限集合 (如自然数集)、有限集合 (如小写英文字母字符集), 还是由多个数据项组成的复合数据元素集合 (如员工基本信息表), 只要集合中的元素性质相同, 都可以被定义为一个数据对象。

2. 数据结构

数据结构是由一组数据元素及这些元素之间存在的特定关系所组成的集合。简而言之, 数据

结构是具有特定组织形式的数据元素的集合，其中，组织形式即指数据元素之间的关系。数据结构包括逻辑结构和存储结构两个层次。

1) 逻辑结构

数据的逻辑结构是从逻辑关系上描述数据的一种抽象方式，独立于数据的存储形式，不依赖于具体的计算机硬件或软件环境。因此，数据的逻辑结构可以被视为从具体问题中抽象出来的数学模型。

数据的逻辑结构由两个基本要素构成：数据元素和数据元素之间的关系。数据元素是逻辑结构的基本单位，而关系则描述了数据元素之间的逻辑联系。

(1) 逻辑结构的分类。根据数据元素之间关系的不同特性，通常可以将逻辑结构分为以下四类基本结构，如图 0-2 所示。它们的复杂程度依次递进。

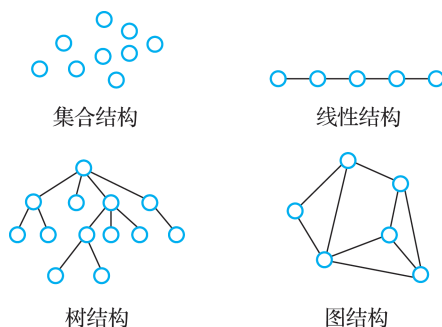


图 0-2 四类基本逻辑结构关系

①集合结构：数据元素除了共同属于一个集合这一基本关系外，元素之间不存在其他任何特定的先后、层次或关联关系，如整数集合、字符集合等。

②线性结构：数据元素之间存在一对一的关系。例如，按照员工入职时间的先后顺序排列公司员工信息，形成的结构即为线性结构。

③树结构：树结构中的数据元素之间具有明显的一对多的层级关系。就像学校内部的组织结构，学校下面是教学院系、行政处室及一些研究所。

④图结构：也称为网状结构，其数据元素之间呈现复杂的多对多关系。例如，在社交平台中，用户之间的关注关系是任意的，任意两个用户都可以相互关注，这种错综复杂的关系形成了图结构或网状结构。

(2) 线性结构和非线性结构。在常见的数据结构中，线性结构和非线性结构是两种主要的分类。

①线性结构。线性结构主要包括以下几种形式。

a. 线性表：典型的线性结构。例如，公司员工信息表，其中的数据元素呈线性排列，每个元素（除首尾元素外）都有一个直接前驱和一个直接后继。

b. 栈和队列：具有特殊操作限制的线性表。在栈中，数据操作遵循后进先出（LIFO）原则，所有操作都在表的一端进行；而在队列中，数据操作遵循先进先出（FIFO）原则，数据的插入在一端进行，删除在另一端进行。

c. 字符串：一种特殊的线性表，其数据元素仅由单个字符组成。例如，一个文本文件中的



内容。

d. 数组（array）：线性表的推广形式，其数据元素本身可以是一个线性表。例如，二维数组可以看作每个元素为一维数组的线性表。

e. 广义表：线性表的另一种推广形式，其数据元素既可以是单个元素，也可以是线性表，具有更灵活的结构。

②非线性结构。非线性结构则包括以下几种。

a. 树：具有多个分支的层次结构。例如，公司的组织架构图，其中每个结点可以有多个子结点，但只有一个父结点。

b. 二叉树：一种特殊的树结构，每个结点最多有两个子结点，常用于数据的快速查找和存储。

c. 有向图：一种图结构，其中的边是有方向的，表示从一个顶点指向另一个顶点的关系。例如，网页之间的链接关系。

d. 无向图：另一种图结构，其中的边是无方向的，表示两个顶点之间的直接连接。例如，社交网络中人与人之间的关系。

这两种结构在数据存储和处理中各有优势，适用于不同的应用场景。这些逻辑结构可以用一个层次图来描述，如图 0-3 所示。

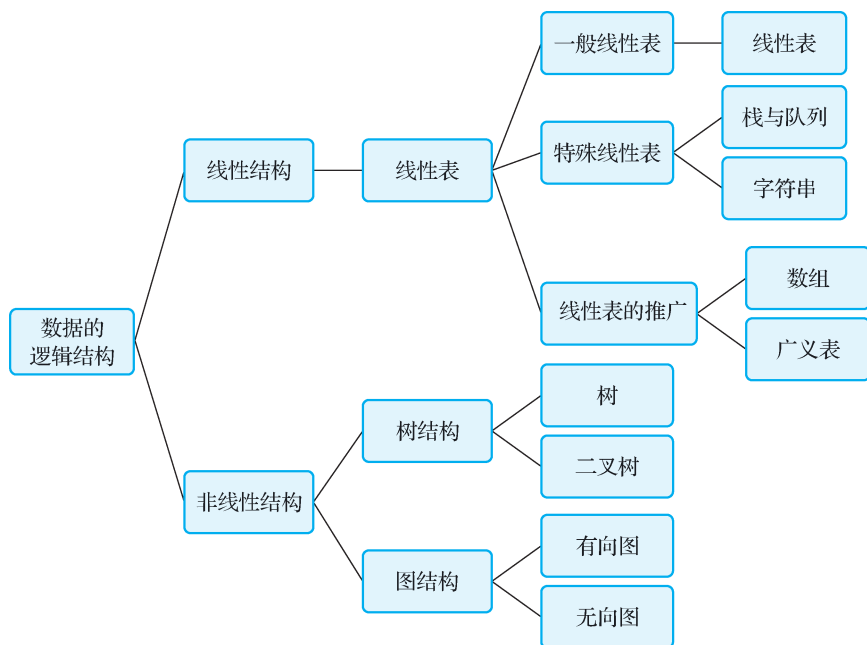


图 0-3 数据的逻辑结构

2) 存储结构

数据对象在计算机中的存储表示称为数据的存储结构，也称为物理结构。在将数据对象存储到计算机中时，通常需要同时存储数据元素本身及数据元素之间的逻辑关系。数据元素在计算机中通常用一个结点来表示。数据元素的存储结构主要有两种基本形式：顺序存储结构和链式存储结构。

（1）顺序存储是把数据元素存放在一块地址连续的存储单元中，其数据间的逻辑关系和物理

关系一致。它通常借助程序设计语言中的数组类型来实现。例如，假设有一个员工考勤信息表，每条记录（结点）占用 100 个存储单元，数据从地址 0 开始，按低地址到高地址的顺序存储，其顺序存储结构如表 0-2 所示。

表 0-2 顺序存储结构

地 址	员 工 编 号	部 门	考 勤 日 期	考 勤 状 态
0	1001	技术部	2024-02-01	正常
100	1002	市场部	2024-02-01	迟到
200	1003	人力资源部	2024-02-01	正常
300	1004	财务部	2024-02-01	请假

（2）顺序存储结构要求所有元素存放在连续的存储空间中，而链式存储结构则不需要占用整块连续的存储空间。为了表示结点之间的关系，链式存储结构需要为每个结点附加一个指针字段，用于存放后继结点的存储地址。链式存储结构通常借助程序设计语言中的指针类型来实现。

例如，对于上述员工考勤信息表，为每个结点附加一个后继结点地址字段，用于存放后继结点的首地址，其链式存储结构如表 0-3 所示。

表 0-3 链式存储结构

地 址	员 工 编 号	部 门	考 勤 日 期	考 勤 状 态	后继结点的首地址
0	1001	技术部	2024-02-01	正常	200
100	1004	财务部	2024-02-01	请假	A
200	1002	市场部	2024-02-01	迟到	300
300	1003	人力资源部	2024-02-01	正常	100

为了更直观地表示链式存储结构，可以通过图示来展示。例如，员工考勤信息表的链式存储结构可以用图 0-4 所示的方式表示。

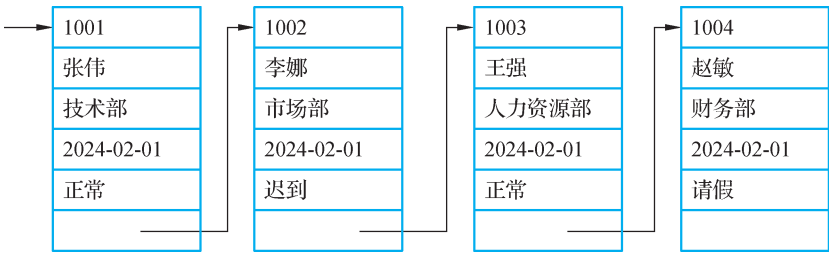


图 0-4 链式存储结构

3. 数据类型

数据类型是高级程序设计语言中的一个重要概念，涵盖了数据的取值范围及对数据可以执行的操作。在程序设计中，数据类型定义了程序中对象的属性，确保每个变量、常量或表达式的结果都归属于某种明确的数据类型。一方面，程序设计语言中的每个数据项都归属于某种数据类型，这些类型明确或隐式地规定了数据的取值范围、存储方式及允许的操作。数据类型实际上是



程序设计中已经实现的数据结构。另一方面,在程序设计过程中,当需要引入新的数据结构时,通常会借助编程语言提供的数据类型来描述数据的存储结构。

在客观世界中,每个数据元素都应具备自身的取值范围和允许的操作。数据类型可以被理解为一个值的集合及定义在这个集合上的一组操作的总称。例如,在 C 语言中,整型可以进行加、减、乘、除和取余数等操作,而实型则不能进行取余数操作。根据值的不同特性,数据类型可以分为不可分解的基本类型和可分解的构造类型。C 语言中的整型、实型、字符型属于基本数据类型;而数组、结构体和共用体则属于构造数据类型。

抽象是提取实际问题本质的过程。在计算机中,数据以二进制形式存储,而在汇编语言中,数据通常以十进制形式表示。这种十进制表示是对二进制数据的抽象,使得编程人员可以直接使用,而无须关注底层的实现细节。在高级语言中,数据的抽象层次更高,引入了数据类型(如整型、浮点型、字符型等),并基于这些基本类型构造出更复杂的抽象数据类型,如线性表、栈、队列、树和图等。

4. 抽象数据类型

抽象数据类型 (abstract data type, ADT) 是对具有某种逻辑关系的数据类型进行描述,并在该类型上定义的一组操作。ADT 主要关注数据的逻辑特性,而与数据在计算机内部的具体表示无关。例如,计算机中的整数数据类型就是一个典型的 ADT。尽管不同处理器的实现方法可能不同,但其逻辑特性保持一致,如加、减、乘、除等运算规则是相同的。

ADT 通常是用户定义的,用于表示特定应用问题的数据模型。它通常由基本数据类型组成,并包含一组相关的服务操作。例如,线性表、栈、队列、串、树、图等结构,都是不同的 ADT 实例。

以建造高楼为例,如果直接使用砖块、水泥和沙子来建造,不仅耗时长,而且建造的高度和规模也会受到限制。但如果使用公司提供的标准化水泥预制板,不仅可以高效、安全地完成高楼建设,还能大大减少接缝量,降低建造的复杂性。同样,ADT 是大型软件构建中的模块化方法。线性表、栈、队列、串、树、图等 ADT 就像是设计大型软件的水泥预制板,利用这些 ADT 可以安全、快速且方便地设计出功能复杂的大型软件。

ADT 是对象的数据模型,定义了数据对象、数据对象中各数据元素之间的关系,以及对数据元素的操作。例如, C++ 中的类就是一个 ADT,包括用户自定义类型的定义及在该类型上的一组操作。

ADT 体现了程序设计中的问题分解、抽象和信息隐藏特性。它将实际问题分解为多个小规模、易于处理的子问题,然后建立一个计算机能够处理的数据模型。每个功能模块的实现细节被封装为独立单元,隐藏了具体的实现过程。

这类似于人们日常生活中的盖房子过程。盖房子可以分解为几个小任务:一方面,需要工程技术人员提供设计图纸;另一方面,建筑工人根据图纸打地基、建造房屋,之后还需要装修工人进行装修。工程技术人员无须了解打地基和建造房屋的具体过程,装修工人也不需要知道如何绘制图纸和建造房屋。这种分工协作的方式正是 ADT 中问题分解和信息隐藏的体现。

5. 抽象数据类型的描述

抽象数据类型可以通过一个三元组来表示,即 $ADT = (D, S, P)$ 。其中, D 是数据对象的集合,表示 ADT 所包含的所有数据元素。 S 是定义在数据对象集合 D 上的关系集合,描述了数据元素之



间的逻辑关系。 P 是对数据对象集合 D 进行的基本操作集合，定义了可以在数据对象上执行的操作。

这种表示方法清晰地定义了抽象数据类型的三个核心组成部分：数据元素、数据元素之间的关系及对数据元素的操作。

抽象数据类型的定义格式如下。

```
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>
}ADT 抽象数据类型名
```

6. 抽象数据类型的表示与实现

本书采用的类C语言基于C语言的核心子集进行了扩充和修改，增强了语言的描述功能。以下是其主要特性。

(1) 预定义常量及类型。

```
#define TRUE 1
#define FALSE 0
```

(2) 数据结构的存储结构通过typedef定义。

```
typedef 数据类型 别名;
// 使用示例
typedef int ElemType;
```

(3) 基本操作以函数形式描述。

```
函数类型 函数名 (函数参数表)
{
    语句序列
}
```

示例函数如下。

```
int ExampleFunction(int param)
{
    if (param > 0) {
        return TRUE;
    } else {
        return FALSE;
    }
}
```

(4) 内存动态分配与释放。使用 malloc 和 free 来动态分配和释放内存。

```
#include <stdlib.h> // 包含 malloc 和 free 的头文件
// 分配空间
ElemType *pointer = malloc(sizeof(ElemType));
// 释放空间
```




```
free(pointer);
```

(5) 赋值语句。

```
// 简单赋值
```

```
变量名 = 表达式;
```

```
// 结构赋值
```

```
structName1 = structName2; // 假设 structName1 和 structName2 是同一结构体类型
```

```
// 条件赋值
```

```
变量名 = (条件表达式)? 表达式 T: 表达式 F;
```

```
// 交换赋值
```

```
// 使用临时变量
```

```
ElemType temp = 变量名 1;
```

```
变量名 1 = 变量名 2;
```

```
变量名 2 = temp;
```

(6) 选择语句。

```
// 条件语句 1
```

```
if(表达式) 语句;
```

```
// 条件语句 2
```

```
if(表达式) 语句;
```

```
else 语句;
```

```
// 开关语句
```

```
switch(表达式)
```

```
{
```

```
    case 值 1: 语句序列 1; break;
```

```
    case 值 2: 语句序列 2; break;
```

```
    :
```

```
    case 值 n: 语句序列 n; break;
```

```
    default: 语句序列 n+1;
```

```
}
```

(7) 循环语句。

```
// for 语句
```

```
for(表达式1; 条件; 表达式2) 语句;
```

```
// while 语句
```

```
while(条件) 语句;
```

```
// do-while 语句:
```

```
do {
```

```
    语句序列;
```

```
} while(条件);
```

(8) 结束语句。

```
// 函数结束语句
```

```
return 表达;
```



```
return;  
// 循环或 case 结束语句  
break;  
// 异常结束语句  
exit( 异常代码 );
```

(9) 输入 / 输出语句。

```
// 输入 / 输出语句采用 C 语言流式输入 / 输出  
// 输入语句  
scanf("格式控制字符串", 输入变量地址列表);  
// 输出语句  
printf("格式控制字符串", 输出变量列表);
```

(10) 基本函数。

```
// 求最大值  
Max( 表达式 1, ..., 表达式 n)  
// 求最小值  
Min( 表达式 1, ..., 表达式 n)
```

下面以日期为例，给出一个完整的抽象数据类型的定义、表示和实现。

```
// 定义部分
```

```
ADT Date {
```

 数据对象: $D = \{year, month, day \mid year \text{ 为年份, } month \text{ 为 } 1 \text{ 到 } 12 \text{ 的整数, } day \text{ 为 } 1 \text{ 到 } 31 \text{ 的整数}\}$

 数据关系: $R = \{<year, month, day>\}$

 基本操作:

```
        Create(&D, year, month, day)
```

 操作结果: 构造一个日期 D, 其年份为 year, 月份为 month, 日期为 day。

```
        GetYear(D)
```

 初始条件: 日期 D 已存在。

 操作结果: 返回日期 D 的年份。

```
        GetMonth(D)
```

 初始条件: 日期 D 已存在。

 操作结果: 返回日期 D 的月份。

```
        GetDay(D)
```

 初始条件: 日期 D 已存在。

 操作结果: 返回日期 D 的日期。

```
        IsLeapYear(D)
```

 初始条件: 日期 D 已存在。

 操作结果: 如果日期 D 所在的年份是闰年, 则返回 1, 否则返回 0。

```
        AddDays(D, days)
```

 初始条件: 日期 D 已存在, days 为正整数。

 操作结果: 返回日期 D 加上 days 天后的日期。



```
}
```

```
ADT Date
```

在后面的项目中，每定义一个新的数据结构，都先用这种定义方式给出其抽象数据类型的定义，对于数据结构的表示方法，则根据不同的存储结构相应给出，同时用类 C 语言给出主要操作的实现方法。为了让读者对抽象数据类型有一个完整、正确的理解，下面给出日期的存储表示和相应操作的具体实现过程。

```
// 表示部分
```

```
typedef struct {
```

```
    int year;    // 年份
```

```
    int month;   // 月份
```

```
    int day;     // 日期
```

```
} Date;
```

```
// 实现部分
```

```
#include <stdio.h>
```

```
// 构造一个日期
```

```
void Create(Date *D, int year, int month, int day) {
```

```
    D->year = year;
```

```
    D->month = month;
```

```
    D->day = day;
```

```
}
```

```
// 获取日期的年份
```

```
int GetYear(Date D) {
```

```
    return D.year;
```

```
}
```

```
// 获取日期的月份
```

```
int GetMonth(Date D) {
```

```
    return D.month;
```

```
}
```

```
// 获取日期的日期
```

```
int GetDay(Date D) {
```

```
    return D.day;
```

```
}
```

```
// 判断是否为闰年
```

```
int IsLeapYear(Date D) {
```

```
    return (D.year % 4 == 0 && D.year % 100 != 0) || (D.year % 400  
== 0);
```



```
}
// 获取每个月的天数
int GetDaysInMonth(Date D) {
    int daysInMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
30, 31};
    if(D.month == 2 && IsLeapYear(D)) {
        return 29;
    }
    return daysInMonth[D.month - 1];
}
// 日期加天数
Date AddDays(Date D, int days) {
    while(days > 0) {
        int daysInMonth = GetDaysInMonth(D);
        if(D.day + days <= daysInMonth) {
            D.day += days;
            days = 0;
        } else {
            days -= (daysInMonth - D.day + 1);
            D.day = 1;
            if (D.month == 12) {
                D.month = 1;
                D.year++;
            } else {
                D.month++;
            }
        }
    }
    return D;
}
```

通过这个日期抽象数据类型的例子，读者可以更清晰地理解抽象数据类型的概念及如何通过数据结构和操作函数实现其功能。

◆ 0.3 算法和算法分析

在数据类型构建完成后，需要对其进行操作，通过构建一系列运算组成程序。而这些运算的构建方式极为关键，因为它直接影响着计算机程序的运行效率。怎样构建一个优质的运算集合，正是算法研究的核心问题。在数据结构的研究领域中，算法问题无处不在，算法与数据在计算过程中的组织形式紧密相连。当我们想要实现特定的操作时，往往需要经过深思熟虑去设计与之适配的算法。可以说，算法是深入研究和透彻理解数据结构的关键钥匙。



1. 算法的定义

算法是一种明确的解决问题的步骤序列,通过一系列有序的操作来完成特定任务。每个操作都执行一个特定的功能,这些操作组合起来就能解决一个具体问题或一类问题。例如,求 n 个数的平均值的问题,其算法描述如下。

- (1) 定义一个变量存放 n 个数的总和,并赋初值 0 ($\text{sum} = 0$)。
- (2) 把 n 个数依次加到 sum 中 (假设 n 个数存放在数组 a 中, $\text{for}(j=0; j < n; j++) \text{sum} += a[j]$)。
- (3) 计算平均值并存放到变量 average 中 ($\text{average} = \text{sum} / n$)。

以上算法中,括号里是 C 语言描述。

算法可以通过多种方式来描述,常见的包括自然语言描述、伪代码 (或类语言) 描述、程序流程图及程序设计语言描述,具体如下。

(1) 自然语言描述:可以使用汉语、英语等文字来详细阐述算法的逻辑和步骤。这种方式通俗易懂,但可能不够精确,且容易因描述模糊而产生歧义。

(2) 伪代码描述:伪代码是一种类似于程序设计语言的形式,使用简洁的语句来描述算法的逻辑结构,但不具备可直接运行的特性。它介于自然语言和程序设计语言之间,便于理解和转换为具体的编程语言。

(3) 程序流程图描述:程序流程图通过图形化的方式展示算法的执行流程,具有直观、形象的优点。然而,它不易直接转化为可运行的程序代码,通常需要进行进一步转换为程序设计语言。

(4) 程序设计语言描述:使用具体的编程语言 (如 C、C++、Java 等) 来实现算法的方式。这种描述可以直接在计算机上运行,具有明确性和可执行性,但可能因语言特性而显得较为复杂。

2. 算法的特性

算法具备以下五个关键特性。

(1) 有穷性:算法必须在有限的步骤后终止,不会陷入无限循环。并且,算法的每一步都应在合理的时间内完成,确保算法能够在可接受的时间范围内结束。

(2) 确定性:算法的每一步骤都应具有明确的含义,不存在模糊或二义性。在给定的条件下,算法只能按照唯一的路径执行,相同的输入必然产生唯一的输出结果,确保输出结果的确定性。

(3) 可行性:算法中的每一步操作都必须是切实可行的,即每一步都能通过有限次的操作完成,确保算法的每一步都能在实际环境中执行。

(4) 输入:算法可以有零个或多个输入,输入是算法执行的初始条件或数据。

(5) 输出:算法至少有一个或多个输出,输出可以是打印结果,也可以是返回一个或多个值,输出是算法执行的结果。

3. 算法设计的要求

一个好的算法应该具备以下目标。

(1) 正确性:算法在合法的输入数据下,能够在有限的时间内得出正确的结果。这是算法最基本的要求,也是衡量算法是否可用的核心标准。



(2) 可读性：一个优秀的算法不仅应具备良好的机器可执行性，更应便于人类理解和交流。可读性强的算法能够帮助开发者更好地理解其逻辑，便于后续的维护和优化；反之，难以理解的算法容易隐藏错误，且难以调试和修改。

(3) 健壮性：当输入数据非法或不符合预期时，算法能够做出合理的响应或进行适当的处理，而不是产生错误或无法理解的输出结果。健壮性体现了算法对异常情况的处理能力，是确保程序稳定运行的重要特性。

(4) 高效性和低存储量：算法的效率主要体现在其执行时间上。针对同一问题，若存在多种算法可供选择，那么执行时间较短的算法通常被认为效率更高，而执行时间较长的算法效率则相对较低。此外，算法在执行过程中对存储空间的需求也是一个重要的考量因素。设计算法时，应尽量选择那些既高效又对存储量需求较低的算法，以优化资源利用并提升整体性能。

4. 算法的时间复杂度

分析算法效率的主要目的是判断算法在实际应用中的可行性，并在解决同一问题的多种算法中，通过比较时间和空间性能来选择更优的算法。

1) 两种算法时间性能分析方法

衡量算法效率的方法通常分为两类：事前分析估算法和事后统计法。事后统计法需要先将算法实现为程序，然后通过运行来测量其时间和空间开销。然而，这种方法存在明显缺陷：首先，它要求算法必须被转换为可执行的程序；其次，测量结果会受到计算机软硬件环境等因素的影响，从而难以准确反映算法本身的优劣。因此，人们更倾向于采用事前分析估算法，通过计算算法的渐近复杂度来评估其效率。这种方法能够更客观地反映算法在大规模数据处理时的性能表现，而不依赖具体的硬件环境。

2) 问题规模和语句频度

在分析算法的时间代价时，问题规模是最重要的因素之一，而与计算机的软硬件环境无关。问题规模反映了算法处理问题的输入量，是问题大小的本质体现，通常用整数 n 表示。对于不同的问题，问题规模 n 的含义也不同。例如，在排序算法中， n 表示待排序记录的数量；在矩阵运算中， n 表示矩阵的阶数；在多项式运算中， n 表示多项式的项数；在集合运算中， n 表示集合中元素的个数；在树的运算中， n 表示树的结点数；在图的运算中， n 表示图的顶点数或边数。一般来说，问题规模 n 越大，算法的执行时间越长。

算法的执行时间大致等于其所有语句执行时间的总和，而语句的执行时间则是该语句的重复执行次数与执行一次所需时间的乘积。语句的重复执行次数称为语句频度。

由于语句的执行需要经过源代码编译、目标代码生成及运行等多个步骤，语句执行一次的实际时间与机器的软硬件环境（如机器性能、编译器质量等）密切相关。因此，算法分析的目标并非精确计算算法的实际执行时间，而是通过估算语句的执行次数来推断算法的时间复杂度。

假设每条语句执行一次所需的时间为单位时间，则算法的执行时间可以用算法中所有语句频度之和来衡量。

【例 0-4】线性查找算法：在一个长度为 n 的数组中查找某个目标值 `target`。

```
int LinearSearch(int arr[],int n,int target)
{
```




```
for(int i = 0; i < n; i++) {           // 遍历数组
    if (arr[i] == target) {           // 比较当前元素与目标值
        return i;                     // 找到目标值, 返回索引
    }
}
return -1;                             // 未找到目标值, 返回 -1
}
```

该算法中所有语句频度之和是数组长度 n 的函数, 记为 $f(n)$ 。换句话说, 上例算法的执行时间与 $f(n)$ 成正比。具体计算可得 $f(n) = n$ 。

3) 算法的时间复杂度定义

对于简单的算法 (如例 0-4), 可以直接计算出所有语句的频度。但对于稍微复杂一些的算法, 通常是比较困难的, 即便能够给出, 也可能是一个非常复杂的函数。因此, 为了客观地反映一个算法的执行时间, 可以只用算法中基本语句的执行次数来度量算法的工作量。所谓基本语句, 指的是算法中重复执行次数与算法执行时间成正比的语句, 它对算法运行时间的贡献最大。通常, 算法的执行时间是随问题规模增大而增长的, 因此, 对算法的评价通常只需考虑其随问题规模增大的趋势。

在这种情况下, 只需要考虑当问题规模充分大时, 算法中基本语句的执行次数在渐近意义下的阶。例如, 在例 0-4 的线性查找算法中, 当 n 趋向无穷大时, 显然有 $f(n) \approx n$, 即当 n 充分大时, $f(n)$ 和 n 之比是一个不等于零的常数。因此, $f(n)$ 和 n 是同阶的, 或者说 $f(n)$ 和 n 的数量级相同。在这里, 用 O 来表示数量级, 记作 $T(n) = O(f(n)) = O(n)$ 。由此, 可以给出算法时间复杂度的定义: 一般情况下, 算法中基本语句重复执行的次数是问题规模 n 的某个函数 $f(n)$, 算法的时间量度记作 $T(n) = O(f(n))$ 。

它表示随着问题规模 n 的增大, 算法执行时间的增长率与 $f(n)$ 的增长率相同, 称为算法的渐近时间复杂度, 简称时间复杂度。

数学符号 O 的严格定义: 若 $T(n)$ 和 $f(n)$ 是定义在正整数集合上的两个函数, 则 $T(n) = O(f(n))$ 表示存在正的常数 C 和 n_0 , 使得当 $n \geq n_0$ 时, 满足 $0 \leq T(n) \leq Cf(n)$ 。

该定义说明函数 $T(n)$ 和 $f(n)$ 具有相同的生长趋势, 并且 $T(n)$ 的增长至多趋向于函数 $f(n)$ 的增长。符号 O 用来描述增长率的上限, 表示当问题规模 $n > n_0$ 时, 算法的执行时间不会超过 $f(n)$ 。

4) 算法的时间复杂度分析举例

分析算法时间复杂度的基本方法: 找出所有语句中频度最高的语句作为基本语句, 计算该语句的频度得到问题规模 n 的某个函数 $f(n)$, 并用符号 O 表示其数量级。具体计算数量级时, 可以遵循以下定理。

定理 0.1: 若 $f(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$ 是一个 m 次多项式, 则 $T(n) = O(n^m)$ 。

定理 0.1 说明, 在计算算法时间复杂度时, 可以忽略所有低次幂项和最高次幂项的系数。这样可以简化算法分析, 同时也体现了增长率的含义。

如果算法可以用递归方法描述, 则其时间复杂度通常可以用递归方程表示, 此时将涉及递归



方程的求解问题。

以下通过具体例子说明如何求非递归算法的时间复杂度。

【例 0-5】常量阶示例。

```
{x++;s = 0;}
```

两条语句的频度均为 1，算法的执行时间是一个与问题规模 n 无关的常数。因此，算法的时间复杂度为 $T(n) = O(1)$ ，称为常量阶。

实际上，如果算法的执行时间不随问题规模 n 的增大而增长，算法中语句的频度就是一个常数。即使这个常数很大，算法的时间复杂度仍然是 $O(1)$ 。例如，上述程序改为以下形式。

```
for(j = 0; j < 50000; j++) {x++;s = 0;}
```

算法的时间复杂度仍然是 $O(1)$ 。

【例 0-6】线性阶示例。

```
for(k = 0; k < 3n; k++) {x++;s = 0;}
```

循环体内的两条基本语句的频度均为 $f(n) = 3n$ ，因此，算法的时间复杂度为 $T(n) = O(n)$ ，称为线性阶。虽然循环次数是 $3n$ ，但根据定理 0.1，常数系数可以忽略，最终复杂度仍为 $O(n)$ 。

【例 0-7】平方阶示例。

```
① x = 0;
② for(k = 1; k <= 3n; k++)
③   x++;
④ for(i = 1; i <= n; i++)
⑤   for(j = 1; j <= 5n; j++)
⑥     y++;
```

对于循环语句，只需考虑循环体中语句的执行次数。在上述程序段中，频度最大的语句是⑥，其频度为 $f(n) = n \times 5n = 5n^2$ 。因此，该算法的时间复杂度为 $T(n) = O(n^2)$ ，称为平方阶。

在多数情况下，当有若干个循环语句时，算法的时间复杂度通常由最深层循环内的基本语句的频度 $f(n)$ 决定。

【例 0-8】立方阶示例。

```
① x=1;
② for(i = 1; i <= n; i++)
③   for(j = 1; j <= i; j++)
④     for(k = 1; k <= j; k++)
⑤       x++;
```

显而易见，该程序段中频度最大的语句是⑤。这条最深层循环内的基本语句的频度依赖于各层循环变量的取值。从内到外分析，语句⑤的执行次数如下。

$$\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j 1 = \sum_{i=1}^n \sum_{j=1}^i j = \sum_{i=1}^n \frac{i(i+1)}{2} = \frac{\left[\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} \right]}{2} \quad (0-1)$$

因此，该算法的时间复杂度为 $T(n) = O(n^3)$ ，称为立方阶。



【例 0-9】对数阶示例。

```
for(j = 1; j <= n; j = 2j){x++; s = 0;}
```

设循环体内两条基本语句的频度为 $f(n)$ 。循环每次将 j 的值加倍,直到 j 大于 n 。假设循环执行了 k 次,则当 $2^k > n$ 时,循环结束。因此,循环次数 k 满足 $2^k > n$,取对数得 $k > \log_2 n$ 。因此,循环次数 k 与 $\log_2 n$ 成正比。

循环体内的两条语句“ $x++$ ”和“ $s = 0$ ”每次循环都会执行一次,因此它们的频度为 k 。因为 k 与 $\log_2 n$ 成正比,所以循环体内语句的频度 $f(n)$ 为 $O(\log_2 n)$ 。

由于循环体内语句的频度为 $O(\log_2 n)$,因此,算法的时间复杂度为 $T(n) = O(\log_2 n)$,称为对数阶。

常见的时间复杂度按数量级递增排列依次为常量阶 $O(1)$ 、对数阶 $O(\log_2 n)$ 、线性阶 $O(n)$ 、线性对数阶 $O(n \log_2 n)$ 、平方阶 $O(n^2)$ 、立方阶 $O(n^3)$ 、 k 次方阶 $O(n^k)$ 、指数阶 $O(2^n)$ 等。

图 0-5 展示了不同时间复杂度随问题规模增长的变化趋势。通常情况下,随着输入规模 n 的增大,那些增长速度较慢的算法 [时间复杂度函数 $T(n)$ 增长较慢的算法] 被认为是更优的。显然,时间复杂度为指数级的算法 [如 $O(2^n)$] 效率非常低,当 n 的值稍微增大时,这些算法就变得不切实际,难以应用。因此,应该尽可能选择那些时间复杂度为多项式级别的算法 [如 $O(n)$],而避免使用那些效率低下的指数级算法。

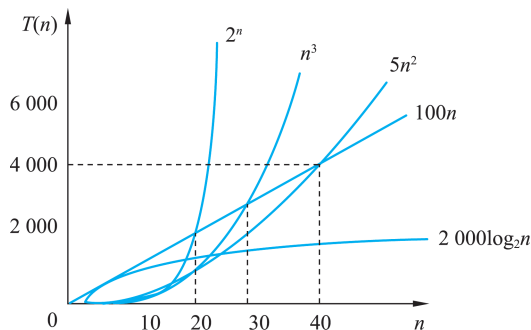


图 0-5 常见函数的增长率

5) 最好、最坏和平均时间复杂度

对于某些问题的算法,其基本语句的频度不仅与问题的规模相关,还依赖于其他因素。

在分析算法效率时,时间复杂度是一个重要的指标,反映了算法执行时间与问题规模之间的关系。然而,对于某些算法,其时间复杂度不仅取决于问题的规模,还受到输入数据的特征和其他因素的影响。

【例 0-10】顺序查找算法。

假设有一个一维数组 a ,我们需要在其中查找一个值等于 e 的元素,并返回该元素的位置。如果找到,则返回其位置 (从 1 开始计数);如果未找到,则返回 0。算法的伪代码如下。

```
① for(i=0; i<n; i++)
②   if(a[i] == e) return i + 1;    // 返回元素的位置 (从 1 开始计数)
③ return 0;                       // 如果未找到,返回 0
```



在这个算法中，语句②的执行次数（频度）不仅与数组的规模 i 有关，还与数组 a 中的元素值及目标值 e 的具体位置有关。假设数组中一定存在值等于 e 的元素，那么算法的执行效率将取决于目标元素在数组中的位置。

最好情况：如果目标值 e 恰好是数组的第一个元素，那么无论数组的规模有多大，if 语句仅执行一次。此时，语句②的频度为 $f(n)=1$ ，时间复杂度为 $O(1)$ 。

最坏情况：如果目标值 e 是数组的最后一个元素，那么 if 语句需要执行 i 次。此时，语句②的频度为 $f(i)=i$ ，时间复杂度为 $O(i)$ 。

平均情况：对于一个算法来说，需要考虑各种可能出现的情况及每一种情况出现的概率。一般情况下，可假设查找的元素在数组中所有位置上出现的可能性均相同，则可取语句②的频度在最好情况与最坏情况下的平均值，即 $f(n)=n^2$ ，作为它的度量。

这个例子表明，算法的时间复杂度不仅取决于问题的规模（如数组的长度 n ），还受到输入数据的特征（如目标值的位置）的影响。类似地，某些排序算法的执行时间也与待排序数据的初始状态密切相关。因此，有时会对算法有最好、最坏及平均时间复杂度的评价。

算法在最好情况下的时间复杂度为最好时间复杂度，指的是算法计算量可能达到的最小值；算法在最坏情况下的时间复杂度为最坏时间复杂度，指的是算法计算量可能达到的最大值；算法的平均时间复杂度是指算法在所有可能情况下，按照输入实例以等概率出现时，算法计算量的加权平均值。

对算法时间复杂度的度量，人们更关心的是最坏情况下和平均情况下的时间复杂度。然而在很多情况下，算法的平均时间复杂度难以确定。因此，通常只讨论算法在最坏情况下的时间复杂度，即分析在最坏情况下算法执行时间的上界。在后面内容中讨论的时间复杂度，除特别指明外，均指最坏情况下的时间复杂度。

5. 算法的空间复杂度

与算法的时间复杂度类似，算法的空间复杂度（space complexity）是衡量算法在运行过程中所需存储空间的量度。空间复杂度通常表示为问题规模 n 的函数，记作 $S(n)=O(f(n))$ 。

当一个程序在计算机上运行时，除了需要存储指令、常量、变量和输入数据外，还需要额外的存储空间来支持数据操作。这些额外的存储空间被称为辅助存储空间。对于输入数据本身所占用的存储空间，通常与问题本身的规模相关，而与算法的设计无关。因此，在分析算法的空间复杂度时，我们主要关注算法在实现过程中所需的辅助存储空间。

如果一个算法在执行过程中所需的辅助存储空间相对于输入数据量是一个常数，则称该算法为原地工作算法，其空间复杂度为 $O(1)$ 。然而，有些算法所需的临时存储空间会随着问题规模 n 的增大而增加。

为了更直观地理解算法的空间复杂度，下面通过一个简单示例来说明。

【例 0-11】数组求和。

假设需要计算一个一维数组 a 中所有元素的和，并将结果存储在一个变量中。一种方法是使用一个额外的变量 sum 来存储数组元素的和：

```
int sum = 0;
for(int j = 0; j < 5n; j++) {sum += a[j];}
```



在这个算法中,我们仅使用了一个额外的变量 `sum` 来存储数组元素的和。这个变量的大小与数组的规模 n 无关,因此算法的空间复杂度为 $O(1)$ 。

另一种方法是先将数组中的元素复制到一个辅助数组 `b` 中,然后对辅助数组进行求和操作。

```
int b[n];
for(int i = 0; i < n; i++) {b[i] = a[i];}
int sum = 0;
for(int i = 0; i < n; i++) {sum += b[i];}
```

在这个算法中,我们使用了一个大小为 n 的辅助数组 `b`,因此算法的空间复杂度为 $O(n)$ 。

通过对比上述两个算法,可以看到,第一种方法通过直接在原数组上操作,仅使用了一个额外变量,因此空间复杂度较低 [$O(1)$]。而第二种方法需要额外的存储空间来存放辅助数组,导致空间复杂度较高 [$O(n)$]。

在实际应用中,算法的时间复杂度和空间复杂度往往是相互影响的。当追求更优的时间复杂度时,可能会导致占用更多的存储空间,从而使空间复杂度变差;反之,优化空间复杂度可能会增加时间复杂度。

然而,在大多数情况下,由于现代计算机的存储空间相对较为充足,人们通常更关注算法的时间复杂度,将其作为衡量算法优劣的主要指标。不过,在资源受限的环境(如嵌入式系统)中,空间复杂度的重要性也不容忽视。设计算法时需要根据具体应用场景权衡时间复杂度和空间复杂度,以达到最优的性能表现。

◆ 0.4 小结

本绪论主要介绍了数据结构的基本概念、术语,以及算法和时间复杂度分析的相关内容,具体如下。

数据结构是一门专注于研究非数值计算程序设计中的操作对象、对象之间的关系及对它们操作的学科。

数据结构涵盖两个层次:数据的逻辑结构和存储结构。同一种逻辑结构,若采用不同的存储方式,会形成不同的存储结构。

逻辑结构是从具体问题中抽象出的数学模型,从逻辑关系上描述数据,与数据存储无关。按照数据元素间关系的不同特点,通常分为四类基本逻辑结构:集合结构、线性结构、树结构和图结构。

存储结构是逻辑结构在计算机中的存储表现形式,主要有顺序存储结构和链式存储结构两种。

抽象数据类型是由用户定义的,用于表示应用问题的数学模型,以及定义在该模型上的一组操作的总称。它具体包含三部分:数据元素、数据元素之间的关系集合,以及对数据之间的基本操作。

算法是为了处理某类问题而制定的一个有限长的操作序列,具有五个特性:有穷性、确定性、可行性、输入和输出。评价一个算法的优劣通常从正确性、可读性、健壮性、高效性和低存储量四个方面进行考量。



算法分析主要关注算法的时间复杂度和空间复杂度，以此来评估算法的时间和空间效率。在实际应用中，由于运算空间通常较为充裕，一般将算法的时间复杂度作为分析的重点。算法执行时间的增长趋势与函数 $f(n)$ 的增长趋势相同，这种关系称为算法的渐近时间复杂度，记作 $T(n) = O(f(n))$ ，简称时间复杂度。

通过本绪论的学习，读者需要掌握与数据结构相关的基本概念，如数据、数据元素、数据项、数据对象、逻辑结构、存储结构等；重点理解数据结构所包含的两个层次的具体含义及其相互关系；了解抽象数据类型的定义、表示和实现方法；熟悉算法的特性及评价标准；重点掌握算法时间复杂度的分析方法。



0.5 习 题

一、选择题

1. 算法分析的目的是 ()。
 - A. 找出数据结构的合理性
 - B. 研究算法中输入与输出的关系
 - C. 分析算法的效率以求改进
 - D. 分析算法的易懂性和文档性
2. 在数据结构中，与所使用的计算机无关的是数据的 () 结构。
 - A. 存储
 - B. 物理
 - C. 逻辑
 - D. 物理和存储
3. 算法分析的两个主要方面是指 ()。
 - A. 空间复杂度和时间复杂度
 - B. 正确性和简明性
 - C. 可读性和健壮性
 - D. 数据复杂性和程序复杂性
4. 对一个算法的评价，不包括 () 方面的内容。
 - A. 健壮性
 - B. 并行性
 - C. 正确性
 - D. 可读性
5. 在数据结构中，从存储方式上可以把数据结构分成 ()。
 - A. 动态结构和静态结构
 - B. 顺序结构和链式结构
 - C. 线性结构和非线性结构
 - D. 内部结构和外部结构
6. 以下数据结构中，() 是线性结构。
 - A. 图
 - B. 二叉树
 - C. 队列
 - D. 广义表
7. 以下关于算法的描述，正确的是 ()。
 - A. 算法的时间复杂度与计算机性能有关
 - B. 算法的空间复杂度与数据存储方式无关
 - C. 算法的效率取决于问题规模和数据的初始状态
 - D. 算法必须在有限步内结束
8. 以下关于数据元素的描述，正确的是 ()。
 - A. 数据元素是数据的最小单位
 - B. 数据元素是数据的基本单位
 - C. 数据元素必须包含多个数据项
 - D. 数据元素是数据结构的基本单位



9. 设 n 是描述问题规模的非负整数, 以下程序片段的时间复杂度是 ()。

```
x=2;
while (x<n/2)
x=2*x;
```

- A. $O(\log_2 n)$ B. $O(n)$ C. $O(n \log_2 n)$ D. $O(n^2)$

10. 求整数 $n(n \geq 0)$ 阶乘的算法如下, 其时间复杂度是 ()。

```
int fact(int n)
{
    if (n <= 1) return 1;
    return n*fact(n-1);
}
```

- A. $O(\log_2 n)$ B. $O(n)$ C. $O(n \log_2 n)$ D. $O(n^2)$

二、综合题

1. 简述下列概念: 数据类型、抽象数据类型、线性结构、非线性结构、算法、算法的时间复杂度、算法的空间复杂度。
2. 试举一个数据结构的例子, 叙述其逻辑结构和存储结构两个层次的含义及相互关系。
3. 简述存储结构的两种基本实现方式, 并比较它们的优缺点。
4. 在逻辑结构的四种基本类型中, 哪一种结构允许每个元素有多个直接前驱和多个直接后继?
5. 简述数据结构中线性结构和非线性结构的主要区别。



项目 1

顺序表和链表



任务 1 顺序表实现图书管理系统



学习目标

- 理解线性表的定义和特点。
- 理解线性表的顺序存储表示，掌握其基本特征和优缺点。
- 掌握顺序表的基本操作，包括顺序表的创建、查找、插入、删除和输出等。
- 能够明确顺序表的使用场景，并学会使用顺序表解决实际问题。
- 增强逻辑思维与系统思维意识，提高创新思维与解决问题的能力。



任务背景

在数字化时代，图书馆作为文化与知识传播的重要枢纽，也迫切需要借助新技术进行升级。传统以人工进行图书信息管理的方式难以适应信息数据量的指数级增长、读者需求高度个性化及多种信息融合处理的现状。图书管理系统可以实现高效、精准、便捷的管理需求。例如，图书的借阅登记、归还处理、库存盘点及信息查询等工作。

数据结构作为计算机科学与技术的核心基础，其中线性表结构为构建高效的图书管理系统提供了有力的工具。顺序表能够将图书的各种信息（如书名、书号、作者、价格、出版社、出版年份、馆藏位置等）按照一定的顺序组织起来，形成一个有序的信息序列，如同图书馆书架上的图书按照特定分类排列一样。顺序表则进一步为这些信息在计算机内存中的存储提供了一种连续的、紧密的存储方式，便于管理者快速地访问和获取图书信息，例如，能够快速定位到书架上具体位置的某本图书。利用线性表和顺序表的数据结构特性，可以初步设计一个图书管理系统，实现图书的借阅、归还等管理，提升图书馆的管理水平和服务效能，更好地满足读者对于图书资源的获取需求，促进知识的传播与共享。同时引导读者适应大数据时代信息的快速变化、数据激增现状，完善图书管理系统。



任务描述

某大学要对图书馆的书籍进行更新，须设计一个图书管理系统实现以下功能：图书的录入、查询、插入、删除、修改和输出等功能，系统功能图如图 1-1 所示。

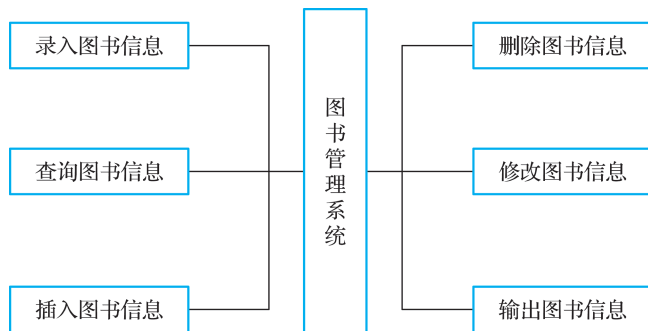


图 1-1 图书管理系统功能图



- (1) 录入：将所有书籍信息录入图书管理系统。
- (2) 查询：可以根据书名或书号找到指定图书所在书架位置。
- (3) 插入：上架新购买的图书，并按照书名的字典顺序排列。
- (4) 删除：可以根据书名或书号下架指定图书。
- (5) 修改：可以根据书名或书号修改指定图书的相关信息，如价格。
- (6) 输出：可以输出系统中所有图书的信息。

设计思路

本任务利用线性表的数据结构存储图书信息，根据图书表的特点将其抽象成一个线性表，每本图书作为线性表中的一个元素。

- (1) 定义“图书”结构体，包括“书名”“书号”“作者”和“价格”等信息。
- (2) 在数组存储的顺序存储结构基础上，自定义函数实现增、删、查、改功能。
- (3) 通过主函数调用自定义函数，输入数据，完成图书管理系统的顺序表。

相关知识

1. 线性表的定义

线性表是一种有序的集合，其元素具有一对一的关系，并且除了第一个元素和最后一个元素之外，每个元素都有且仅有一个前驱元素和一个后继元素。线性表可以是空表，也可以包含有限个元素。

线性结构表达式为 $(a_1, a_2, a_3, \dots, a_n)$ ，其中， a_i ($i=1, 2, \dots, n$) 表示线性表中的第 i 个数据元素， n 表示线性表的长度。通过表达式清晰地展示了线性表中数据元素的排列顺序和它们之间的线性关系，线性表的逻辑结构示意图如图 1-2 所示。

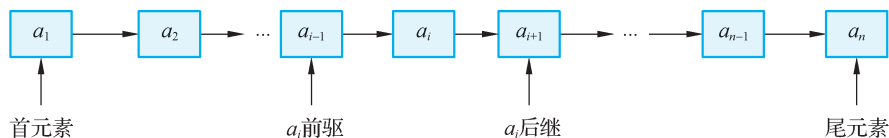


图 1-2 线性表的逻辑结构示意图

线性结构表达式的特点如下。

- (1) 有序性。线性结构表达式中的数据元素是按照一定的顺序排列的，即存在唯一的第一个元素 (a_1) 和最后一个元素 (a_n)，且每个元素 (除了第一个和最后一个) 都有且仅有一个直接前驱和一个直接后继。
- (2) 有限性。线性结构表达式中的数据元素数量是有限的，即存在一个确定的长度 n 。
- (3) 同类性。线性结构表达式中的数据元素通常具有相同的类型或特性，这样可以对它们进行统一处理和操作。

线性表基本操作的抽象定义如下。

ADT List



```
{  
    数据对象:  $D = \{a_i | a_i \in \text{ElemSet}, i = 1, 2, \dots, n, n \geq 0\}$   
    数据关系:  $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i = 2, \dots, n \}$   
    基本操作:  
    InitList(&L)  
    构造一个空的线性表 L。  
    DestroyList(&L)  
    若线性表 L 存在, 则将线性表 L 销毁。  
    ClearList(&L)  
    若线性表 L 已存在, 则将 L 重置为空表。  
    ListEmpty (L)  
    若线性表 L 已存在, 且 L 为空表, 则返回 true, 否则返回 false。  
    ListLength (L)  
    若线性表 L 已存在, 则返回 L 中数据元素的个数。  
    GetElem(L, i, &e)  
    若线性表 L 已存在, 且  $1 \leq i \leq \text{ListLength}(L)$ , 则用 e 返回 L 中第 i 个数据元素的值。  
    LocateElem(L, e)  
    若线性表 L 已存在, 则返回 L 中第 1 个值与 e 相同的元素在 L 中的位置。若这样的数据元素不存在, 则返回值为 0。  
    PriorElem(L, cur_e, &pre_e)  
    若线性表 L 已存在, 若 cur_e 是 L 的数据元素, 且不是第一个, 则用 pre_e 返回其前驱, 否则操作失败, pre_e 无定义。  
    NextElem(L, cur_e, &next_e)  
    若线性表 L 已存在, 若 cur_e 是 L 的数据元素, 且不是最后一个, 则用 next_e 返回其后继, 否则操作失败, next_e 无定义。  
    ListInsert(&L, i, e)  
    若线性表 L 已存在, 且  $1 \leq i \leq \text{ListLength}(L) + 1$ , 则在 L 中第 i 个位置之前插入新的数据元素 e, L 的长度加 1。  
    ListDelete(&L, i)  
    若线性表 L 已存在且非空, 且  $1 \leq i \leq \text{ListLength}(L)$ , 则删除 L 的第 i 个数据元素, L 的长度减 1。  
    TraverseList(L)  
    若线性表 L 已存在, 则对线性表 L 进行遍历, 在遍历过程中对 L 的每个结点访问一次。  
}ADT List
```

2. 顺序表的存储表示

作为线性表的一种顺序存储结构, 顺序表利用一组地址连续的存储单元来依次保存线性表中的各个元素。这种存储方式确保了逻辑上相邻的元素在物理位置上也保持相邻, 从而通过物理存储的相邻关系来反映元素之间的逻辑相邻关系。

(1) 存储单元。顺序表将元素依次存放在连续的存储单元中, 这些单元在物理上是相邻的, 有助于快速访问相邻元素。

（2）存储结构。顺序表采用线性存储结构，元素按照其在线性表中的顺序进行存储。每个元素占用固定的存储单元长度，因此，只要知道起始元素的地址，就可以通过简单的计算来定位任意元素的地址。计算公式如式（1-1）所示。

$$\text{LOC}(a_i) = \text{LOC}(a_1) + (i - 1) * L$$

(1-1)

其中， $\text{LOC}(a_i)$ 表示第 i 个元素的地址， $\text{LOC}(a_1)$ 表示第一个元素的地址， L 是元素所占存储单元的长度， i 是元素的位序（从 1 开始），顺序表的存储结构如图 1-3 所示。

存储内容	存储地址
元素1	$\text{LOC}(a_1)$
元素2	$\text{LOC}(a_1) + L$
.....	...
元素 <i>i</i>	$\text{LOC}(a_1) + (i - 1) * L$
.....	...
元素 <i>n</i>	$\text{LOC}(a_1) + (n - 1) * L$

图 1-3 顺序表存储结构示意图

（3）动态分配。虽然顺序表在创建时通常需要指定长度（数组的大小），但可以通过动态存储分配（如 C 语言中的 malloc 函数）在程序执行过程中根据需要动态地分配或调整存储空间。当数据空间不足时，可以分配更大的存储空间来替换原有的空间，从而实现存储空间的动态扩展。

3. 顺序表的基本操作

```
// 定义顺序表的结构体
#define MAXSIZE 100 // 最大长度
typedef struct
{
    ElemType *elem; // 指向数据元素的基地址
    int len; // 顺序表的当前长度
}SqList;
```

在以上语句中，typedef 关键字用于为已有的数据类型创建一个新的别名。在这里，定义了一个新的结构体类型，并为它起了一个别名 SqList。在后续的代码中，可以直接使用 SqList 来声明该结构体类型的变量。

（1）初始化。顺序表的初始化操作就是构造一个空的顺序表。

【算法 1.1】顺序表的初始化。

算法步骤如下。

- ①为顺序表 L 动态分配一个预定义大小的数组空间，使数组指针 elem 指向这段空间的基地址。若分配失败，则返回 -1。
- ②将顺序表的当前长度设为 0。
- ③返回 1，表示初始化成功。

算法实现如下。

```
int InitList(SqList *L)
```




```
{ // 构造一个空的顺序表 L
    // 为顺序表分配一个最多存储 MAXSIZE 个元素的数组空间
    L->elem = (ElemType*)malloc(MAXSIZE*(sizeof(ElemType)));
    if(!L->elem) return -1; // 存储空间分配失败, 返回失败标识 -1
    L->len = 0; // 当前顺序表中无元素, 长度为 0
    return 1; // 完成初始化操作, 返回成功标识 1
}
```

(2) 插入。顺序表的插入操作就是在表的第 i 个位置插入一个新的数据元素 e , 使得长度为 n 的顺序表变成长度为 $n+1$ 的顺序表。

【算法 1.2】顺序表的插入。

算法步骤如下。

①判断插入位置 i 是否合法。 i 值的合法范围是 $1 \leq i \leq n+1$ (n 是顺序表的长度), 若插入位置不合法, 则返回 -1。

②判断顺序表的存储空间是否已满。若满, 则返回 -2。

③若 $i=n+1$, 则无须移动元素; 否则, 将第 n 个至第 i 个位置的元素依次向后移动一个位置, 空出第 i 个位置。

④将要插入的新元素 e 放在第 i 个位置上。

⑤顺序表长度加 1。

⑥返回插入成功标识 1。

算法实现如下。

```
int InsertList(SqList *L, int i, ElemType e)
{ // 在顺序表的第 i 个位置插入一个新元素 e, i 值的合法范围是  $1 \leq i \leq L->len+1$ 
  if((i < 1) || (i > L->len + 1)) return -1; // 返回值 -1, 表示插入位置不合法
  if(L->len == MAXSIZE) return -2; // 返回值 -2, 表示存储空间已满
  for(j = L->len - 1; j >= i - 1; j--)
      L->elem[j+1] = L->elem[j]; // 将第 n 个至第 i 个位置的元素依次后移
  L->elem[i - 1] = e; // 将新元素 e 放入第 i 个位置
  ++L->len; // 顺序表长度加 1
  return 1; // 完成插入操作, 返回成功标识 1
}
```

算法分析如下。

①上述算法没有处理表的动态扩充, 因此, 当表长已经达到预设的最大空间时, 则不能再插入元素。

②数据元素 a_{i-1} 和 a_i 之间的逻辑关系发生了变化。由于顺序表中逻辑上相邻的数据元素在物理位置上也是相邻的, 因此, 当插入的位置不是 $n+1$ 时, 必须移动元素来留出插入位置 i , 并反映这个逻辑关系的变化。

③本算法的执行时间主要耗费在移动元素上, 而移动元素的耗时取决于插入元素的位置。假设 p_i 是在顺序表的第 i 个位置插入一个新元素的概率, E_{insert} 是在长度为 n 的顺序表中插入一个元



素需要移动元素的平均次数，则平均移动次数的公式如式（1-2）所示。

$$E_{\text{insert}} = \sum_{i=1}^{n+1} p_i (n - i + 1) \quad (1-2)$$

假定在顺序表的任何位置上插入元素都是等概率的，则 p_i 的值如式（1-3）所示。

$$p_i = \frac{1}{n+1} \quad (1-3)$$

利用式（1-3）简化式（1-2），得到

$$E_{\text{insert}} = \frac{n}{2} \quad (1-4)$$

因此，顺序表插入算法的平均时间复杂度为 $O(n)$ 。

（3）取值。顺序表的取值操作就是根据指定取值位置 i ，获取对应数据元素的值。由于顺序表具有随机存取的特点，因此可以通过数组下标 $i-1$ 得到 $\text{elem}[i-1]$ 位置存储的第 i 个数据元素。

【算法 1.3】 顺序表的取值。

算法步骤如下。

判断指定的取值位置 i 是否合法。 i 值的合法范围是 $1 \leq i \leq \text{L.len}$ ，若取值位置不合法，则返回 -1；否则，将第 i 个数据元素 $\text{L.elem}[i-1]$ 赋给参数 e ，通过 e 返回第 i 个数据元素的值。

算法实现如下。

```
int GetElem(SqList L, int i, ElemType *e)
{
    // 获取顺序表第 i 个位置的元素值，并将元素值通过 e 返回
    if (i < 1 || i > L.len) return -1; // i 值不合法，返回失败标识 -1
    *e = L.elem[i - 1];                // elem[i-1] 单元存储第 i 个数据元素
    return 1;                          // 完成取值操作，返回成功标识 1
}
```

算法分析如下。

顺序表取值算法的时间复杂度为 $O(1)$ 。

（4）查找。顺序表的查找操作就是查找顺序表中第 1 个与指定元素值 e 相等的元素。若查找成功，则返回该元素在顺序表中的位置序号；若查找失败，则返回 -1。

【算法 1.4】 顺序表的查找。

算法步骤如下。

①从第一个元素开始，依次比较顺序表中的元素和指定元素值 e 。若找到与 e 值相等的元素 $\text{L.elem}[i]$ ，则查找成功，返回该元素的位置序号 $i+1$ 。

②若查遍整个顺序表都没有找到与 e 相等的元素，则查找失败，返回 -1。

算法实现如下。

```
int SearchElem(SqList L, ElemType e)
{
    // 在顺序表 L 中查找值为 e 的数据元素，返回其序号
    for (i = 0; i < L.len; i++)
    {
        if (L.elem[i] == e) return i + 1; // 查找成功，返回序号 i+1
    }
    return -1;                          // 查找失败，返回 -1
}
```



}

算法分析如下。

本算法的执行时间主要耗费在顺序表元素与被查元素的比较次数上,而比较次数取决于被查元素在顺序表中的位置。假设 ASL 是在顺序表中找到被查元素所经历的平均查找次数, p_i 是被查元素在顺序表中位置序号为 i 的概率, c_i 是被查元素在顺序表第 i 个位置时,找到它需要进行的比较次数,如式 (1-5) 所示。

$$ASL = \sum_{i=1}^n p_i c_i \quad (1-5)$$

假定被查元素出现在顺序表的任何位置都是等概率的,则 p_i 的值如式 (1-6) 所示。

$$p_i = \frac{1}{n} \quad (1-6)$$

一般情况下,被查元素出现在顺序表第 i 个位置,需要比较 i 次才能找到,故 c_i 等于 i 。简化式 (1-5),得到

$$ASL = \frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2} \quad (1-7)$$

因此,顺序表按值查找算法的平均时间复杂度为 $O(n)$ 。

(5) 删除。顺序表的删除操作就是删除顺序表中第 i 个元素,长度为 n 的顺序表会变成长度为 $n-1$ 的顺序表。

【算法 1.5】 顺序表的删除。

算法步骤如下。

①判断删除位置 i 是否合法。 i 值的合法范围是 $1 \leq i \leq n$ (n 是顺序表的长度),若删除位置不合法,则返回 -1。

②若 $i=n$,则无须移动元素;否则,将第 $i+1$ 个至第 n 个位置的元素依次向前移动一个位置。

③顺序表长度减 1。

④返回删除成功标识 1。

算法实现如下。

```
int DeleteElem(SqList *L, int i)
{    // 删除顺序表 L 中的第 i 个元素, i 值的合法范围是  $1 \leq i \leq L->len$ 
    if((i < 1) || (i > L->len)) return -1;    // i 值不合法, 返回失败标识 -1
    for(j = i; j <= L-> len - 1; j++)
        L->elem[j - 1] = L->elem[j];    // 将第 i+1 个至第 n 个位置的元素依次前移
    --L->len;    // 删除第 i 个元素后, 顺序表长度减 1
    return 1;    // 完成删除操作, 返回成功标识 1
}
```

}

算法分析如下。

①数据元素 a_{i-1} 、 a_i 和 a_{i+1} 之间的逻辑关系发生了变化。由于顺序表中逻辑上相邻的数据元素在物理位置上也是相邻的,因此,当删除的位置不是 n 时,必须移动元素来反映逻辑关系的变化。



②本算法的执行时间主要耗费在移动元素上，而移动元素的耗时取决于删除元素的位置。假设 p_i 是删除顺序表第 i 个元素的概率， E_{delete} 是在长度为 n 的顺序表中删除一个元素需要移动元素的平均次数，则计算方法如式 (1-8) 所示。

$$E_{\text{delete}} = \sum_{i=1}^n p_i (n-i) \quad (1-8)$$

假定删除顺序表任意位置上的元素都是等概率的，则将式 (1-6) 代入式 (1-8)，结果如式 (1-9) 所示。

$$E_{\text{delete}} = \frac{n-1}{2} \quad (1-9)$$

因此，顺序表删除算法的平均时间复杂度为 $O(n)$ 。

任务实现

1. 顺序表的存储表示

图书管理系统使用两个结构体 Book 和 SqList 来表示图书和图书列表（顺序表）。Book 结构体包含图书的 ID、书名、作者和价格。SqList 结构体包含一个 Book 类型的数组 books 和一个整数 size，用于存储图书信息和记录当前存储的图书数量。图书表的具体存储结构的定义如下。

```
#define MAX_BOOKS 100           // 最大图书数量
#define MAX_NAME 50             // 最大书名长度
#define MAX_AUTHOR 50           // 最大作者名长度
// 图书信息结构
typedef struct
{
    int id;                      // 图书 id
    char name[MAX_NAME];         // 书名
    char author[MAX_AUTHOR];     // 作者
    double price;                // 价格
} Book;
// 图书表的顺序结构
typedef struct
{
    Book books[MAX_BOOKS];       // 存储图书的数组
    int size;                    // 当前图书数量
} SqList;
```

2. 顺序表的主要操作

在图书管理系统中，对顺序表的主要操作包括初始化顺序表、添加图书、查找图书、删除图书、修改图书和显示所有图书信息。



视频

项目1任务1任务
实现讲解



1) 初始化顺序表

初始化顺序表的操作将图书列表的当前大小设置为 0，表示图书列表为空。具体实现如下。

```
void initSqList(SqList *list)
{
    list->size = 0;
}
```

2) 添加图书

添加图书操作用于将一本新图书添加到顺序表中，并保持顺序表中图书按照书名的字典顺序排列。如果顺序表已满，则返回错误代码 -1。具体实现如下。

```
int insertBook(SqList *list, Book book)
{
    if(list->size >= MAX_BOOKS)
    {
        return -1;                // 顺序表已满
    }
    int i;
    // 查找插入位置
    for(i = list->size; i > 0 && strcmp(list->books[i - 1].name,
book.name) > 0; i--);
    // 将元素向后移动以为新元素腾出空间
    for(int j = list->size; j > i; j--)
    {
        list->books[j] = list->books[j-1];
    }
    // 插入新元素
    list->books[i] = book;
    list->size++;
    return 0;                    // 添加成功
}
```

该操作需要找到正确的插入位置，这可能需要遍历整个顺序表，时间复杂度为 $O(n)$ ，其中， n 是顺序表中图书的数量。找到位置后，需要将其后元素向后移动，这同样需要 $O(n)$ 的时间。因此，插入操作的整体时间复杂度为 $O(n)$ 。

3) 查找图书

查找图书操作用于在 SqList 顺序表中查找具有特定 ID 的图书，并返回该图书在数组中的位置索引。如果遍历完整个数组都没有找到，则返回 -1，表示图书不存在。具体实现如下。

```
int findBookById(SqList *list, int id)
{
    for(int i = 0; i < list->size; ++i)
    {
```



```
        if(list->books[i].id == id)
        {
            return i;                // 返回找到的位置
        }
    }
    return -1;                        // 未找到, 返回 -1
}
```

该操作需要遍历顺序表中的所有图书, 直到找到匹配的 ID 或遍历完整整个数组。在最坏的情况下, 如果图书 ID 不在表中或位于最后, 需要遍历 n 个元素, 因此时间复杂度为 $O(n)$ 。

4) 删除图书

删除图书操作用于从顺序表中删除具有特定 ID 的图书。如果没有找到对应图书, 则返回错误代码 -1。具体实现如下。

```
int deleteBookById(SqlList *list,int id)
{
    int index = findBookById(list,id);
    if(index == -1)
    {
        return -1; // 未找到
    }
    for(int i = index;i < list->size - 1;++i)
    {
        list->books[i] = list->books[i + 1];
    }
    list->size--;
    return 0; // 删除成功
}
```

该操作查找图书位置的时间复杂度为 $O(n)$, 将找到的图书之后的所有图书向前移动一位, 这也需要 $O(n)$ 的时间。因此, 删除操作的整体时间复杂度为 $O(n)$ 。

5) 修改图书

修改图书操作用于修改 SqlList 顺序表中特定 ID 图书的价格。如果没有找到对应图书, 则返回 -1。具体实现如下。

```
int editBookPriceById(SqlList *list,int id)
{
    int index = findBookById(list,id);
    if (index != -1)
    {
        double newPrice;
        printf(" 输入新的价格 :");
        scanf("%lf",&newPrice);
        list->books[index].price = newPrice;    // 修改价格
    }
}
```




```
        printf(" 图书价格已更新。\\n");  
        return 0;  
    }  
    return -1;  
}
```

该操作同样需要查找图书,其时间复杂度为 $O(n)$ 。一旦找到图书,修改价格是一个常数时间操作。因此,修改操作的整体时间复杂度为 $O(n)$ 。

6) 显示所有图书信息

显示所有图书信息操作用于打印 SqList 顺序表中所有图书的详细信息。具体实现如下。

```
void displayBooks(SqList *list)  
{  
    printf("%-10s %-20s %-15s %-10s\\n", "ID", " 书名 ", " 作者 ", " 价格 ");  
    for(int i = 0;i < list->size;++i)  
    {  
        printf("%-10d %-20s %-15s %-10.2f\\n",list->books[i].id,list->  
books[i].name,list->books[i].author,list->books[i].price);  
    }  
}
```

该操作需要遍历顺序表中的所有图书,这需要 $O(n)$ 的时间,其中, n 是顺序表中图书的数量。因此,显示操作的时间复杂度为 $O(n)$ 。

图书管理系统的完整代码参考以下二维码。



图文

顺序表实现图书管理系统的完整代码

综合以上分析,虽然顺序表的直接索引访问特性使其在访问元素时极为高效,但在具体应用中,往往需要根据某个属性查找表中元素(如根据 ID 查找图书),这样的查找操作仍需线性搜索。为了提升查找效率,可以为顺序表创建辅助索引。

3. 顺序表的特点

1) 优点

(1) 随机存取优势:顺序表支持通过下标直接访问任意元素,一旦知道所需元素的下标位置,可立即访问该元素,无须遍历整个数据结构,因此查找和存取效率较高,时间复杂度通常为 $O(1)$ 。

(2) 缓存友好特性:由于顺序表在物理存储上是连续的,可以充分利用 CPU 的缓存机制来提高访问速度,当 CPU 访问顺序表中的某个元素时,将相邻元素加载到缓存中(这称为缓存行填充或预取)。当后续需要访问这些相邻元素时,可直接从缓存中获取,无须再次访问内存,从而进一步提高访问速度。



顺序表因其存储密度大、随机访问效率高的特点，在一定程度上满足了图书管理系统中频繁查询操作的需求。在实现顺序表的基本操作方面，如创建顺序表，能够根据预估的图书数量合理分配存储空间，并初始化相关数据结构；查找操作通过遍历或优化算法（如二分查找，若图书信息按特定顺序存储）实现了快速定位目标图书。

2) 缺点

(1) 插入和删除效率低：在顺序表中插入或删除元素时，可能需要移动大量的元素以保持存储的连续性，这会导致操作效率较低。

(2) 空间浪费：顺序表在声明时通常需要指定一个固定的长度。若长度过大，则会造成存储空间的浪费（碎片问题）；反之，则可能引发数组越界错误或需要频繁地进行动态分配和释放操作。

在图书管理系统中，当有新书入库需要插入顺序表中或有图书下架需要删除对应元素时，都可能引发大量元素的移动操作，这不仅消耗系统资源，还会在一定程度上影响系统的运行效率，尤其是在图书数据量较大且借阅、归还操作频繁时，这种影响更为显著。此外，顺序表需要预先分配较大的存储空间，在实际使用中可能会出现存储空间利用率不高的情况，尤其是在图书馆图书数量增长难以准确预估时，可能导致空间浪费或后期频繁扩容操作，影响系统运行效率。

针对顺序表的这些问题，后续可考虑对系统进行改进与优化。例如，研究更适合频繁插入和删除操作的数据结构，如链表或结合顺序表与链表优点的数据结构，来代替单纯的顺序表，以提高系统在动态数据处理方面的性能。

因此，顺序表适用于以下两种情况：数据量较小且规模可预估的情况；较少进行插入或删除操作，需要频繁随机访问数据元素的情况。



动手实践

顺序表作为一种基础的线性表实现方式，以其元素的连续存储和高效的随机访问能力，在数据结构领域中占据着重要地位。接下来通过一个实践任务——合并两个有序顺序表，来加深对顺序表概念和基本操作的理解。

【实践任务】

本任务要求将两个已经按非递减顺序排列的顺序表 L_a 和 L_b 合并成一个新的顺序表 L_c 。在合并过程中， L_c 中的元素也必须保持非递减顺序。此任务涉及顺序表的内存分配、元素比较与复制等操作，是顺序表基本操作的直接应用。

【算法步骤】

- ① 设置索引 i 、 j 、 k 用于遍历 L_a 、 L_b 和 L_c 。
 - ② 为 L_c 分配足够的内存空间以存储来自 L_a 和 L_b 的所有元素。
 - ③ 使用双指针技术，比较 L_a 和 L_b 中对应位置的元素，将较小者复制到 L_c 中，同时移动指针。
 - ④ 在 L_a 或 L_b 的元素全部被复制到 L_c 后，将另一个顺序表中剩余的元素复制到 L_c 的末尾。
- 所有元素复制完成后，合并过程结束。

【算法实现】

```
int MergeList(Sqlist La,Sqlist Lb,Sqlist *Lc)
```



视频

项目1任务1动手
实践讲解



```
{  
    int i = 0, j = 0, k = 0;  
    Lc->length = La.length + Lb.length;  
    Lc->elem = (ElemType*)malloc(Lc->length * sizeof(ElemType));  
    if(Lc->elem == NULL) return -1;          // 存储分配失败  
    while(i < La.length && j < Lb.length)  
    {  
        if(La.elem[i] <= Lb.elem[j]) Lc->elem[k++] = La.elem[i++];  
        else Lc->elem[k++] = Lb.elem[j++];  
    }  
    while(i < La.length) Lc->elem[k++] = La.elem[i++];  
    while(j < Lb.length) Lc->elem[k++] = Lb.elem[j++];  
    return 1;                                // 合并成功  
}
```

顺序表合并的完整代码参考以下二维码。



图文

顺序表合并的完整代码

【算法分析】

该算法需要遍历两个顺序表的所有元素，因此算法的时间复杂度为 $O(m+n)$ ，其中， m 和 n 分别是 La 和 Lb 的长度。此外，该算法需要创建一个额外的顺序表来存储合并后的结果，其大小为两个输入顺序表长度之和，因此算法的空间复杂度为 $O(m+n)$ 。



习 题

一、选择题

- 关于线性表的描述，() 是正确的。
 - 线性表中的元素可以是不同类型的数据
 - 线性表只能通过顺序存储结构来表示
 - 在线性表中，每个元素都有一个前驱和一个后继
 - 线性表的长度是指线性表中元素的个数
- 在一个含有 n 个元素的顺序表中，在第 i 个位置插入一个新元素的时间复杂度为 ()。
 - $O(1)$
 - $O(n)$
 - $O(\log \geq n)$
 - $O(n^2)$
- 对于一个使用数组实现的顺序表，当需要将顺序表的容量扩大时，通常采用的方法是 ()。
 - 将原数组复制到一个新的、更大的数组中
 - 直接修改原数组的大小



- C. 创建一个新的数组, 并仅复制需要保留的部分元素
D. 删除不需要的元素以腾出空间
4. 线性表的顺序存储结构的一个主要缺点是 ()。
- A. 不便于随机访问元素
B. 插入和删除操作效率低
C. 存储密度太低
D. 无法根据下标直接定位元素
5. () 不是线性表的基本运算。
- A. 初始化空表
B. 查找指定元素的位置
C. 对线性表排序
D. 获取线性表的长度
6. 对于顺序存储的长度为 N 的线性表, 以下说法正确的是 ()。
- A. 删除第一个元素和插入最后一个元素的时间复杂度分别为 $O(1)$ 和 $O(N)$
B. 访问第 i ($1 \leq i \leq N$) 个结点和求第 i ($2 \leq i \leq N$) 个结点直接前驱的算法时间复杂度均为 $O(1)$
C. 顺序存储方式的优点是存储密度大, 且插入、删除运算效率高
D. 访问结点和增加结点的时间复杂度分别为 $O(N)$ 和 $O(1)$
7. 在包含 n 个数据元素的顺序表中, () 的时间复杂度为 $O(1)$ 。
- A. 将 n 个元素按升序排序
B. 访问第 i ($1 \leq i \leq n$) 个数据元素
C. 在位序 i ($1 \leq i \leq n+1$) 处插入一个新结点
D. 删除位序 i ($1 \leq i \leq n$) 处的结点
8. 顺序表第一个元素的存储地址是 1100, 每个元素的长度为 2, 则第 5 个元素的地址是 ()。
- A. 1110
B. 1108
C. 1100
D. 1120
9. 下列对顺序存储的有序表 (长度为 n) 实现给定操作的算法中, 平均时间复杂度为 $O(1)$ 的是 ()。
- A. 查找包含指定值元素的算法
B. 插入包含指定值元素的算法
C. 删除第 i ($1 \leq i \leq n$) 个元素的算法
D. 获取第 i ($1 \leq i \leq n$) 个元素的算法

二、综合题

- 给定一个顺序表 L , 编写一个函数统计其中值等于给定值 x 的元素个数。
- 设计一个算法, 实现将顺序有序表中的重复元素删除, 并将剩余的不重复元素按照有序顺序存储在原分配空间。
- 给定一个顺序表 L , 找出表中 m 最大元素并删除, 分析算法的时间复杂度和空间复杂度。
- 设将 n ($n > 1$) 个整数存放于一维数组 R 中。试设计一个在时间和空间两方面都尽可能高效的算法, 将 R 中保存的序列循环左移 p ($0 < p < n$) 个位置, 即将 R 中的数据序列由 $(x_0, x_1, \dots, x_{n-1})$ 变换为 $(x_p, x_{p+1}, \dots, x_{n-1}, x_0, x_1, \dots, x_{p-1})$ 。要求:
(1) 给出算法的基本设计思想。



(2) 根据设计思想,采用 C、C++ 或 Java 语言描述算法,在关键处给出注释。

(3) 说明算法的时间复杂度和空间复杂度。

任务 2 链表实现考试报名管理系统



学习目标

- 理解链表的定义和特点。
- 掌握链表的存储表示及其基本特征和优缺点。
- 掌握链表的基本操作,包括链表的创建、查找、插入、删除和输出等。
- 能够明确链表的使用场景,并学会使用链表解决实际问题。
- 增强逻辑思维与系统思维意识,提高创新思维与解决问题的能力。



任务背景

在当今数字化时代,教育领域的信息化建设不断推进,考试作为教育评价的重要手段,其报名管理方式也亟待升级。随着各类考试的社会化,报名人数呈现爆发式增长,传统的手工报名方式在效率方面存在提升空间,在实际操作过程中也可能会出现信息录入不够精准、信息查询存在一定难度等情况,在很大程度上难以充分契合现代考试管理的多元化需求。

考试报名管理系统作为教育信息化的重要组成部分,需要实现考生信息的快速录入、查询、修改和删除等功能。链表作为一种动态数据结构,能够灵活地应对报名人数的动态变化,为考试报名管理提供高效的支持。与传统的静态数据结构相比,链表具有动态分配内存、插入和删除操作高效等优点,特别适合处理报名人数不确定且频繁变动的场景。

在考试报名过程中,考生信息的管理至关重要。考生信息通常包括姓名、身份证号、报考科目、联系方式、报名时间等。这些信息需要被有序地组织和存储,以便在报名高峰期能够快速响应考生的操作请求。链表通过结点的连接方式,可以方便地在任意位置插入或删除考生信息,而无需像数组那样进行大规模的数据移动,这大大提高了系统的运行效率。

此外,考试报名管理系统还需要具备良好的扩展性和可维护性。链表的结构简单,易于实现,且在面对报名人数的大幅波动时,能够灵活地调整内存分配,避免资源浪费。同时,链表的这种动态特性也使得系统在后续的升级和功能扩展中更加便捷,能够更好地适应教育考试领域的不断发展和变化。

借助链表的数据结构特性来设计考试报名管理系统,可以在一定程度上弥补传统手工报名方式的不足,进而提升考试报名管理工作的效率与精准度,为考生带来更为贴心、高效的报名服务体验。



任务描述

某学校需要设计一个考试报名管理系统,以实现以下功能:考试报名信息的录入、查询、插入、删除、修改和输出等,系统功能图如图 1-4 所示。

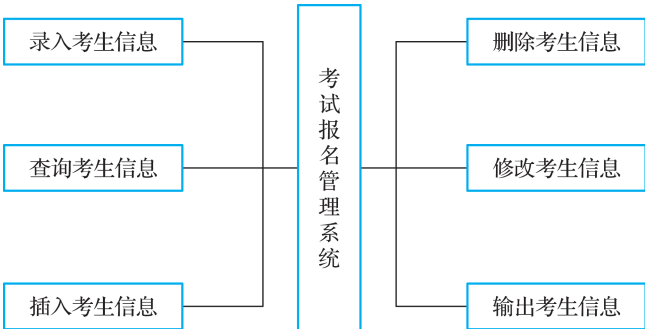


图 1-4 系统功能图

- (1) 录入：将考生信息录入系统。
- (2) 查询：根据考生姓名或报名号查询考生信息。
- (3) 插入：新增考生信息并插入链表中。
- (4) 删除：根据考生姓名或报名号删除考生信息。
- (5) 修改：根据考生姓名或报名号修改考生信息。
- (6) 输出：显示系统中所有考生信息。

设计思路

- 本任务利用链表的数据结构存储考生信息，将每个考生信息抽象为链表中的一个结点。
- (1) 定义“考生”结构体，包括“姓名”“报名号”“考试科目”等信息。
 - (2) 通过链表的基本操作实现增、删、查、改等功能。
 - (3) 通过主函数调用这些操作，完成考试报名管理系统。

相关知识

1. 单链表

线性表的链式存储结构是指利用一组任意的存储单元来存放线性表中的数据元素。这些存储单元既可以是连续的，也可以是不连续的。为了体现数据元素之间的逻辑顺序关系，除了要存储数据元素本身的信息之外，还需要额外存储一个指示其直接后继元素存储位置的信息。这种由数据元素信息和后继元素地址信息共同构成的存储单元被称为结点。每个结点包含两个部分：数据域和指针域。数据域用于存储数据元素的具体内容，而指针域则存储该数据元素直接后继元素的存储地址。单链表结点结构如图 1-5 所示。



图 1-5 单链表结点结构

1) 单链表存储原理

存储数据元素信息值的部分称为数据域（data），而存储其后继结点地址信息的部分称为指针域（next）。因此，一个含有 n 个元素的线性表可以通过每个结点（node）指针域的联系形成一个链表。每个结点中只有一个指向后继结点的指针，因此这种链表也被称为单链表或线性链表。

作为线性表的一种存储结构，结点之间的逻辑结构相对于每个结点的实际内存地址显得更为重要，因此，通常单链表习惯用图 1-6 所示的形式来表示。

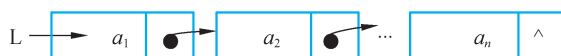


图 1-6 单链表示意图

2) 单链表数据类型定义

单链表是由多个结点组成的数据结构。线性表的单链表存储结构可以按照以下方式定义。

```
typedef int ElemType;           // 定义元素类型为 int
typedef struct LNode
{
    ElemType data;              // 存储数据元素的值
    struct LNode* next;         // 指向下一个结点的指针
} LNode, *LinkList;
```

其中, LNode 为链表的结点类型, LinkList 为指向链表结点的指针类型。如果有定义 “LinkList L;”, 则定义了一个链表, L 指向该链表的第一个结点。它和下面的定义含义相同。

```
LinkList *L;
```

对于不带头结点的链表, 如果链表为空, 则有 “L=NULL;”; 对于带头结点的链表, 如果链表为空, 则 L->next=NULL。

3) 带头结点的单链表

在单链表的前端添加一个特殊的结点 H, 这个结点的类型与数据结点相同, 但其数据域不存储有效数据。链表的头指针变量 L 指向该结点 H, 该结点称为头结点。引入头结点后, 即使链表为空, 头指针 L 也不会为空, 它始终指向头结点的内存地址。头结点可以被视为链表最前面结点的直接前驱, 这样, 链表最前面的结点就有了直接前驱。因此, 在进行插入和删除操作时, 就无须对头结点进行特殊处理。头结点的引入使得空表和非空表的处理方式统一, 简化了操作, 使其更加简洁易懂和方便。

头结点的特点在于其数据域没有定义, 而其指针域则存储链表中第一个数据结点的物理地址。图 1-7 展示了带头结点的单链表在空表和非空表状态下的逻辑结构。

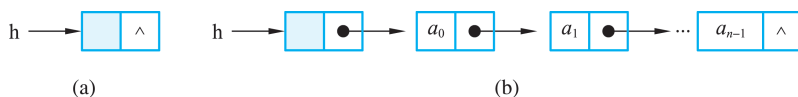


图 1-7 带头结点的单链表

(a) 带头结点的空链表; (b) 带头结点的非空链表

【算法 1.6】带头结点的单链表初始化。

算法步骤如下。

①初始化链表头指针, 分配一个头结点, 其数据域为空, 指针域也初始化为 NULL, 表示链表为空。

②检查头结点是否成功分配内存。

③返回创建好的带头结点的单链表的头指针。

算法实现如下。

```
LinkList Init_LinkList()
{
```



```
LinkList L; // 指向头结点的指针
L = (LNode *)malloc(sizeof(LNode)); // 分配一个头结点
// 检查内存分配是否成功
if(!L) {
    printf("内存分配失败。\\n");
    return NULL; // 分配失败则返回 NULL
}
// 初始化头结点的指针域为 NULL
L->next = NULL; // 头结点的指针域为空
// 返回头指针
return L;
}
```

算法 1.6 只创建了一个结点，所以时间复杂度为 $O(1)$ 。

4) 计算单链表的长度

计算单链表的长度需要遍历整个链表，从第一个结点到最后一个结点，并使用计数器统计其中的元素个数。在实现算法时，我们使用一个指针变量 p 来遍历链表，同时使用一个计数器变量 i 来记录结点的数量。每当 p 指向的结点存在下一个结点时， p 就向后移动到下一个结点，同时计数器 i 加 1，直到 p 指向链表的最后一个结点为止。链表的创建可以有两种形式：带头结点的和不带头结点的，我们需要分别为这两种情况设计算法。需要注意的是，在统计链表长度时，头结点不应计入。

对于带头结点的链表，头结点不存储实际的数据元素，因此在遍历时从头结点的下一个结点开始计数。而对于不带头结点的链表，由于链表的第一个结点就直接存储数据，所以在判断链表是否为空时需要进行特别处理：如果链表为空（头指针 L 为 $NULL$ ），则直接返回长度为 0；如果链表不为空，则从第一个结点开始遍历并计数。这样，在编写后续算法时，如果没有特别说明，通常假定链表是带头结点的，以简化处理过程。

无论是哪种类型的链表，计算长度的算法的时间复杂度均为 $O(n)$ ，其中， n 是链表中数据元素的个数，因为需要遍历每一个元素一次来计算长度。

【算法 1.7】求带头结点的单链表表长。

算法步骤如下。

- ①初始化计数器变量 i 为 0，用于统计链表长度。
- ②将指针变量 p 初始化为头结点 L 。
- ③遍历链表，从第一个实际数据结点到最后一个结点，每遍历一个结点，计数器 i 加 1。
- ④返回计数器 i 的值，即链表长度。

算法实现如下。

```
int Length_LinkList1(LinkList L)
{
```



```
    LNode *p = L->next;    // p 指向第一个实际数据结点
    int i = 0;              // 初始化计数器变量 i
    // 遍历链表, 直到 p 指向最后一个结点
    while(p != NULL) {
        p = p->next;
        i++;
    }
    // 返回链表长度
    return i;
}
```

通过观察算法 1.7, 我们可以发现, 对于带头结点的链表, 即使链表为空, 也无须进行特殊处理。因此, 为了简化算法实现, 在后续的算法设计中, 如果没有特别指明, 默认所操作的单链表都是带头结点的。

算法 1.7 的时间消耗直接取决于链表中数据元素的数量。这意味着, 它的执行时间与链表的长度成正比, 所以这个算法的时间复杂度是 $O(n)$, 其中, n 代表链表中元素的总数。

5) 查找操作

查找操作一般分为按序号查找和按值查找两种情况。

(1) 按序号查找: 从单链表的最前面的结点开始, 判断当前结点是否为第 i 个结点, 若是则返回该结点的地址指针, 否则沿着指针域的指向依次向下寻找, 直至表结束为止。若没有找到第 i 个结点, 则返回空。

【算法 1.8】单链表的按序号查找。

算法步骤如下。

- ①初始化指针 p 指向链表的第一个数据结点。
- ②初始化计数器 j 为 0。
- ③遍历链表, 直到找到第 i 个结点或到达链表末尾。
- ④若找到第 i 个结点, 则返回该结点的地址, 否则返回 NULL。

算法实现如下。

```
// 在带头结点的单链表 L 中查找第 i 个元素
LNode *Get_LinkList(LinkList L, int i)
{
    LNode *p = L->next;    // p 指向单链表 L 的第一个数据结点
    int j = 0;              // 初始化计数器 j
    // 遍历链表直到找到第 i 个结点或到达链表末尾
    while(p != NULL && j < i) {
        p = p->next;
        j++;
    }
```



```
}  
// 若找到第 i 个结点, 则返回该结点, 否则返回 NULL  
if(j == i) return p;  
else return NULL;  
}
```

(2) 按值查找: 从单链表最前面的结点开始, 如果当前结点数据域的值与给定参数 e 相等, 则返回该结点在单链表中的位序; 否则沿着指针域依次向下寻找, 直到表结束为止。若表中没有符合要求的结点, 则返回 -1 。

【算法 1.9】单链表的按值查找。

算法步骤如下。

- ①初始化指针 p 指向链表的第一个数据结点。
- ②初始化计数器 i 为 0。
- ③遍历链表, 直到找到与给定值相同的结点或到达链表末尾。
- ④若找到与给定值相同的结点, 则返回该结点的位序, 否则返回 -1 。

算法实现如下。

```
// 在带头结点的单链表 L 中查找与给定值 e 相同的元素的位序  
int Locate_LinkList(LinkList L, ElemType e)  
{  
    LNode *p = L->next; // p 指向单链表 L 的第一个数据结点  
    int i = 0;           // 初始化计数器 i  
    // 遍历链表直到找到与给定值相同的结点或到达链表末尾  
    while (p != NULL && p->data != e) {  
        p = p->next;  
        i++;  
    }  
    // 若找到与给定值相同的结点, 则返回该结点的位序, 否则返回 -1  
    if (p != NULL && p->data == e) return i;  
    else return -1;  
}
```

算法 1.8 和算法 1.9 的时间复杂度均为 $O(n)$, 其中, n 是链表中数据元素的个数。这是因为在最坏情况下, 需要遍历整个链表来找到指定序号或值的结点。

6) 删除操作

单链表的删除操作涉及移除链表中指定位置的结点。具体来说, 如果要删除第 i 个结点 p , 可以通过将 p 的直接前驱结点 q (第 $i-1$ 个结点) 的指针指向 p 的后继结点来实现, 然后释放结点 p 所占用的存储空间。要删除结点 p , 首先需要找到它的直接前驱结点 q , 接着修改 q 的指针以绕过 p 。具体的指针修改步骤如下。



(1) 从链表中移除结点 p 。为此, 将 p 的直接前驱结点 q 的 `next` 指针指向 p 的后继结点 ($p \rightarrow next$)。这样, q 就直接连接到了 p 的后继结点, 从而绕过了 p 。

(2) 释放结点 p 所占用的存储空间。为此, 使用 `free(p)` 函数来释放 p 。这样, p 就从内存中被移除了。

根据查找算法, 找到结点 p 的直接前驱结点的时间复杂度为 $O(n)$ 。如果需要删除的是结点 q 的直接后继结点 (假设该后继结点存在), 可以通过以下步骤完成。

(1) 定义并指向后继结点。定义一个指针 p , 并将其指向结点 q 的直接后继结点 ($q \rightarrow next$)。这样, p 就指向了要删除的结点。

(2) 将 q 的 `next` 指针指向 p 的后继结点。将 q 的 `next` 指针指向 p 的后继结点 ($p \rightarrow next$), 这样, q 就直接连接到了 p 的后继结点, 从而绕过了 p 。

(3) 释放结点 p 所占用的存储空间。使用 `free(p)` 函数来释放 p 。这样, p 就从内存中被移除了。

上述操作的时间复杂度为 $O(1)$ 。

【算法 1.10】单链表删除。

算法步骤如下。

① 初始化两个指针 p 和 q , 以及一个计数器 j 。 p 指向头结点, j 初始化为 0 (表示头结点的位置)。

② 遍历链表, p 移动到待删除结点的前一个结点 (第 $i-1$ 个结点), 同时 j 记录当前位置。

③ 验证位置 i 是否有效, 即检查 p 的下一个结点是否存在且 j 小于 $i-1$ 。如果位置有效, 保存待删除结点 q 的值到 e , 然后调整 p 的 `next` 指针以跳过 q , 最后释放 q 所占用的内存。

④ 返回操作结果, 若成功, 则返回 1, 若失败 (如位置无效或链表中没有第 i 个结点), 则返回 -1。

算法实现如下。

// 在带头结点的单链表 L 中删除第 i 个位置的结点, 并返回该结点的值

```
int Delete_LinkList(LinkList L, int i, ElemType *e)
{
    LNode *p = L;           // p 指向头结点
    LNode *q = NULL;        // q 用于指向要删除的结点
    int j = 0;              // 初始化计数器
    if(i < 1) {
        return -1;          // 删除位置不合理
    }
    // 寻找第 i 个结点的前驱结点
    while(p != NULL && j < i - 1) {
        p = p->next;
        j++;
    }
    // 检查是否找到了第 i 个结点的前驱结点
```



```
if (p == NULL || p->next == NULL) {  
    return -1;        // 链表中没有第 i 个结点  
}  
  
q = p->next;          // q 指向要删除的结点  
*e = q->data;          // 保存要删除结点的数据  
p->next = q->next;     // 绕过 q, 将其从链表中删除  
free(q);              // 释放 q 所占用的内存  
return 1;              // 删除成功  
}
```

7) 插入操作

在链式存储结构中,插入结点的操作与顺序存储结构不同。插入新结点不需要移动其他数据元素,仅需要调整相关结点的指针。插入操作的关键在于确定插入位置,这通常涉及查找操作,可能需要找到第 i 个结点、具有特定值的结点或满足其他条件的结点。找到位置后,需要确定是将新结点插入找到结点的前面还是后面。

插入操作的两种情况如下。

(1) 在某结点之后插入结点。若要在结点 p 之后插入新结点,操作相对简单。假设 p 指向链表中的某个结点, s 指向包含待插入值 e 的新结点。将 s 插入 p 后面,操作步骤如下。

①将新结点 s 的 $next$ 指针指向 p 的下一个结点。首先,需要让新结点 s 接入链表。为此,将 s 的 $next$ 指针指向 p 当前指向的下一个结点 ($p \rightarrow next$)。这样,新结点 s 就与 p 的后续链表部分连接起来了。

②将 p 的 $next$ 指针指向新结点 s 。需要将新结点 s 插入 p 之后。为此,将 p 的 $next$ 指针指向新结点 s 。这样, p 就与新结点 s 连接起来了。

重要的是,这两条指令的顺序不能颠倒,否则可能导致链表断裂,无法正确完成插入操作。

(2) 在某结点之前插入结点。若要在结点 p 之前插入新结点,首先需要找到 p 的直接前驱结点 pre 的地址。因为插入 p 之前,实际上是在 pre 之后插入新结点。假设 p 指向链表中的某个结点, s 指向待插入的新结点 e 。首先找到 p 的直接前驱结点 q ,然后在 q 之后插入 s 。假设链表的头指针为 L ,插入结点的步骤如下。

①定义一个指针 q ,并将其初始化为指向链表的头指针 L 。此时 q 指向链表的第一个结点(头结点)。

②使用一个循环,从头结点开始逐个遍历链表的结点。在循环中,检查 $q \rightarrow next$ 是否等于目标结点 p 。如果 $q \rightarrow next$ 不等于 p ,则将 q 移动到下一个结点 ($q = q \rightarrow next$)。重复上述步骤,直到找到满足 $q \rightarrow next == p$ 的结点 q 。此时, q 就是 p 的直接前驱结点。

③插入新结点。将新结点 s 的 $next$ 指针指向 $q \rightarrow next$ (p)。将 $q \rightarrow next$ 指向新结点 s 。通过这两步操作,新结点 s 被插入 q 和 p 之间,即在 p 之前插入了新结点。

④完成插入。经过上述步骤,新结点 s 已经被成功插入链表中,且插入位置是目标结点 p 之前。



在链式存储结构中,插入操作的时间复杂度:在某结点之后插入结点的,操作是常数时间复杂度 $O(1)$;在某结点之前插入结点的,操作时间复杂度为 $O(n)$,因为需要遍历链表找到前驱结点。

【算法 1.11】单链表的插入。

算法步骤如下。

- ①定义指针 p 和 s ,分别指向当前结点和待插入的新结点。
- ②初始化指针 p ,指向链表的头指针 L ,初始化 j 为 0,用于计数。
- ③使用 `while` 循环遍历链表,寻找第 $i-1$ 个结点,即目标结点 p 的直接前驱结点 q 。
- ④如果未找到目标结点 p (p 为 `NULL` 或 j 不满足条件),则返回 `-1`,表示插入失败。
- ⑤分配内存生成新结点 s ,并将新结点 s 的数据域设置为 e 。
- ⑥将新结点 s 的 `next` 指针指向 $q \rightarrow \text{next}$ (目标结点 p),然后将 $q \rightarrow \text{next}$ 指向新结点 s ,完成插入操作。
- ⑦返回 1,表示插入成功。

算法实现如下。

```
int Insert_LinkList(LinkList L,int i,ElemType e)
{
    LNode *p, *s;
    int j;
    p = L; j = 0;
    while(p && j < i - 1) { p = p->next; j++; } // 寻找第 i-1 个结点
    if (!p || j > i - 1) return -1; // i 值不合法
    s = (LNode *)malloc(sizeof(LNode)); // 生成新结点
    s->data = e;
    s->next = p->next; p->next = s; // 实现插入
    return 1;
}
```

由于查找任何一个结点的平均时间复杂度都是 $O(n)$,因此查找某个结点或它的前驱结点的平均时间复杂度也都是 $O(n)$,即在不知道结点及其前驱结点地址的情况下,插入一个结点的时间复杂度为 $O(n)$ 。

8) 建立单链表

由于单链表是一种动态结构,每个链表占用的空间不需要预先分配,而是由系统根据需要动态生成。因此,建立单链表的过程就是一个动态生成链表的过程,即从“空表”的初始状态起,依次建立各元素结点,并将其逐个插入链表。根据线性表元素的输入顺序与单链表中结点的顺序是否相同分,单链表的建立可以采用以下两种方法。

(1) 头插入法建立单链表。链表中的每个结点占用的存储空间都不需要预先分配,一般是在运行时系统根据需求实时生成的。因此建立单链表都是从空表开始的,头插入法建立单链表也称为逆序建表。其算法思想是建立单链表时每读入一个数据元素则向系统申请一个结点空间,然后



插在链表的头部。图 1-8 所示为线性表 (34,49,27,66,59) 对应的单链表建立过程,因为是在链表的头部插入,所以读入数据的顺序和线性表中各数据元素的逻辑顺序是相反的。

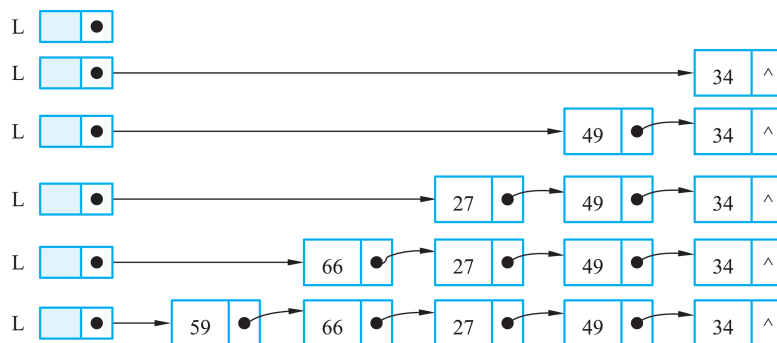


图 1-8 头插入法建立单链表示意图

【算法 1.12】头插入法建立带头结点的单链表。

算法步骤如下。

- ①初始化链表: 创建一个头结点, 将其 next 指针设置为 NULL。
- ②遍历数据元素: 依次读取线性表 (如 [34, 49, 27, 66, 59]) 中的每个数据元素。
- ③插入新结点: 对于每个读取的数据元素, 创建一个新结点, 并将其插入链表的头部 (头结点的 next 指针指向的位置)。每次插入后, 新结点成为链表的第一个结点。
- ④返回链表: 返回创建好的带头结点的单链表。

算法实现如下。

```

LNode* CreateLinkedListWithHeadInst(int arr[],int n)
{
    LNode* L;          // 头结点
    LNode* newNode; // 用于创建新结点
    // 初始化链表
    L = (LNode*)malloc(sizeof(LNode));
    if(!L) {
        printf("内存分配失败, 无法创建头结点\n");
        return NULL;
    }
    L->next = NULL; // 初始化头结点的 next 指针为 NULL
    // 遍历数据元素
    for(int i = 0;i < n;i++) {
        // 创建新结点
        newNode = (LNode*)malloc(sizeof(LNode));
        if(!newNode) {
            printf("内存分配失败, 无法创建新结点\n");
            return NULL;
        }
    }
}

```



```

// 设置新结点的数据
newNode->data = arr[i];
newNode->next = NULL;
// 插入新结点: 将新结点插入链表头部
newNode->next = L->next; // 新结点指向当前头结点的下一个结点
L->next = newNode;      // 头结点的 next 指针指向新结点
}
// 返回链表
return L;
}

```

(2) 尾插入法建立单链表。尽管头插入法构建单链表的过程相对直观,但这种方法会导致输入数据的顺序与链表中元素的排列顺序相反,这可能会给用户带来一定的困扰。为了解决这个问题,可以采用尾插入法来构建单链表,这样可以使输入的数据顺序与链表中元素的顺序保持一致。在尾插入法中,每次新结点的添加都是位于链表的末尾。由于单链表通常使用头指针来标识,如果要定位到链表的尾部,通常需要从头指针开始遍历整个链表。这种方法在每次插入操作时都会消耗大量的时间在重复的遍历上。为了优化算法并减少运行时间,我们引入了一个尾指针 r ,它始终指向链表的最后一个结点。这样,新结点就可以直接被添加到链表的末尾,从而将插入操作的时间复杂度降低到常数级别。图 1-9 所示为通过在链表尾部插入结点来构建链表的过程。

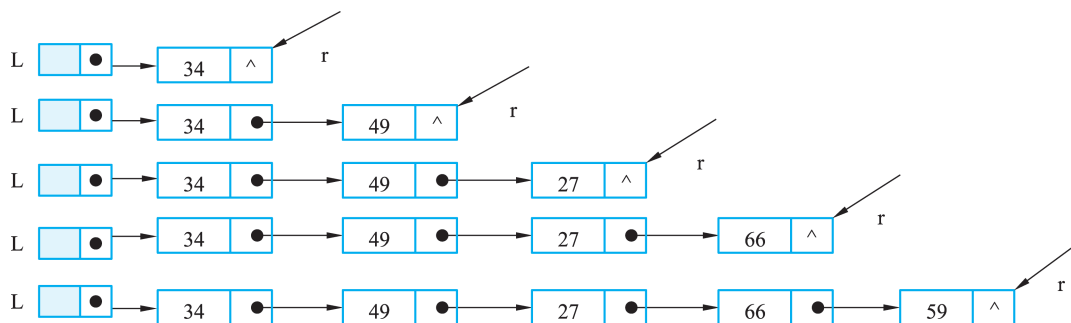


图 1-9 尾插入法建立单链表示意图

【算法 1.13】尾插入法建立单链表。

算法步骤如下。

- ①初始化头结点: 分配内存空间给头结点; 将头结点的 `next` 指针初始化为 `NULL`。
- ②创建尾指针: 初始化一个尾指针 `tail`, 初始时指向头结点。
- ③遍历数据: 遍历数组中的每个元素。对于每个元素来说, 可以
 - a. 创建一个新结点。
 - b. 设置新结点的数据为当前数组元素。
 - c. 将新结点的 `next` 指针设置为 `NULL`。
 - d. 将新结点插入尾指针的后面。
 - e. 更新尾指针, 使其指向新插入的结点。
- ④返回链表头结点。

算法实现如下。



```
ListNode* CreateLinkedListWithTailInst(int arr[],int n)
{
    ListNode* L; // 头结点
    ListNode* tail; // 尾指针
    ListNode* newNode; // 用于创建新结点
    // 初始化链表
    L = (ListNode*)malloc(sizeof(ListNode)); // 分配头结点空间
    if(!L) {
        printf("内存分配失败,无法创建头结点\n");
        return NULL;
    }
    L->next = NULL; // 初始化头结点的 next 指针为 NULL
    tail = L; // 尾指针初始指向头结点
    // 遍历数据元素
    for(int i = 0;i < n;i++) {
        // 创建新结点
        newNode = (ListNode*)malloc(sizeof(ListNode));
        if(!newNode) {
            printf("内存分配失败,无法创建新结点\n");
            return NULL;
        }
        // 设置新结点的数据
        newNode->data = arr[i];
        newNode->next = NULL;
        // 将新结点插入尾指针的后面
        tail->next = newNode;
        // 更新尾指针
        tail = newNode;
    }
    // 返回链表
    return L;
}
```

算法 1.12 和算法 1.13 耗费的时间与所建立的线性表中数据元素的数量有关,其时间复杂度为 $O(n)$ 。

在算法 1.13 中,我们对链表尾端结点的插入操作有一些特别的处理逻辑。当我们要插入链表的第一个有效数据结点时,虽然链表存在头结点(头结点不存储实际数据,只是作为辅助结点),但此时头结点的下一个指针指向 NULL,相当于链表还没有实际的数据结点。在这种情况下,新插入的有效数据结点将成为链表的第一个实际数据结点,需要将它的地址赋值给头结点的 next 指针域。而对于后续插入的其他有效数据结点,由于此时头结点的 next 指针已经指向了一个有效数据结点(链表已经有了部分结构),它们都有前驱结点(头结点或之前已插入的有效数据结点),



所以只需将新结点的地址放入前驱结点的 `next` 指针域即可。因此,在算法实现过程中,必须对这种情况的结点插入操作进行适当的处理。

2. 循环链表

在链表的数据结构中,头指针扮演着至关重要的角色。如果失去了头指针,我们将无法定位到链表,进而无法执行插入、查找、删除等操作。然而,存在一种方法可以使链表中的任何一个结点都具备类似头结点的功能,那就是将链表尾结点的指针域指向头结点,从而形成一个环状结构,即单向循环链表,如图 1-10 所示。

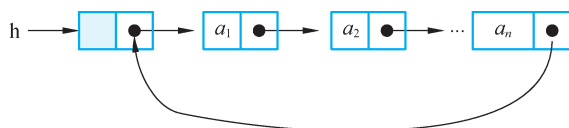


图 1-10 带头结点的单向循环链表

单向循环链表的操作与普通单链表基本相同,也需要从头指针开始进行操作,查找、插入、删除等算法也相似,只是在遍历(如查找)操作中,在单链表中,判别条件为 $p \neq \text{NULL}$ 或 $p \rightarrow \text{next} \neq \text{NULL}$,而循环单链表的判断条件为 $p \neq L$ 或 $p \rightarrow \text{next} \neq L$ 。

单向循环链表的一个显著特点是,可以从任意一个结点开始遍历整个链表,这是普通单链表所不具备的。正因如此,在单向循环链表的操作中,甚至可以不设置头指针,而只设置一个尾指针,因为尾指针后移一步就是链表头结点的指针。实际上,拥有尾指针也就意味着我们已经知道了头指针的位置。

例如,在连接两个单循环链表 L_1 和 L_2 的操作中,即将 L_2 的第一个数据元素结点连接到 L_1 的尾结点之后,如果两个单循环链表都通过头指针 h_1 、 h_2 来指示头结点,则需要从头结点开始依次搜索整个链表来找到单循环链表 L_1 的尾结点,再完成相关连接工作,其时间复杂度为 $O(n)$ 。而如果两个单循环链表是通过尾指针 r_1 、 r_2 来标识的,则时间复杂度可以优化为 $O(1)$ 。

连接的具体步骤如下。

- ①定义一个指针 q ,并将其指向 L_1 的头结点 ($r_1 \rightarrow \text{next}$)。这样, q 保存了 L_1 的头结点指针。
- ②将 L_1 的尾结点 (r_1) 的 `next` 指针指向 L_2 的第二个结点 ($r_2 \rightarrow \text{next} \rightarrow \text{next}$)。这样, L_1 的尾结点直接连接到 L_2 的第一个数据结点的后继结点,实现了 L_1 和 L_2 的连接。
- ③释放 L_2 的头结点 ($r_2 \rightarrow \text{next}$)。由于 L_2 的头结点在连接后不再需要,可以释放其占用的内存空间。
- ④将 L_2 的尾结点 (r_2) 的 `next` 指针指向 L_1 的头结点 (q)。这样,整个链表形成了一个大的循环链表。

两个单向循环链表的连接过程如图 1-11 所示。

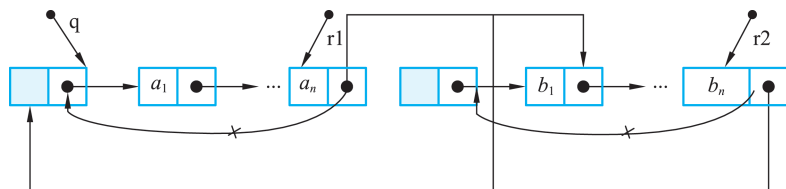


图 1-11 两个尾指针标识的单向循环链表的连接



3. 双向链表

单向循环链表在某些操作方面比单向链表更便利,但也存在一些局限性。单向链表的特点是每个结点仅包含一个指向其后继结点的指针。这种设计使得查找后继结点非常直接和快速,因为可以直接通过指针访问;然而,查找前驱结点则需要从链表的头结点开始,逐个结点遍历直到找到目标结点。查找后继结点的时间复杂度为 $O(1)$,而查找前驱结点的平均时间复杂度为 $O(n)$,这取决于结点在链表中的位置。

为了使查找前驱结点和后继结点同样方便,可以考虑在结点中增加一个指向其前驱结点的指针,这样就形成了双向链表。

双向链表的存储结构描述如下。

双向链表的结点结构包含三个部分:数据域 `data`、指向前驱结点的指针 `prior` 及指向后继结点的指针 `next`。其结构定义如下。

```
typedef struct DuLNode
{
    ElemType data;
    struct DuLNode *prior;
    struct DuLNode *next;
} DuLNode, *DuLinkList;
```

在双向链表中,如果 `p` 是指向表中某一个结点的指针,则有 `p->next->prior == p->prior->next == p`。

带头结点的双向循环链表结合了双向链表和循环链表的优势。

(1) 双向循环链表的构建。为统一删除链头、链中、链尾结点的算法,双向链表可增加头结点。若对前向单链表也增加头结点,使双向链表的前向和后向单链表共用该头结点且首尾相接,就能构成双向循环链表,统一删除算法,还具备循环链表便于从尾部访问头部的优点。

(2) 双向链表与单向链表操作对比。

①相同操作:查找第 i 个结点、查找满足特定性质的结点及遍历链表等仅涉及单方向指针的操作,双向链表和单链表实现方式基本相同。

②插入与删除操作:在单链表中删除结点时,仅知道结点地址 `p` 是不够的,还需查找其直接前驱结点;插入时,要先查找插入位置,若插入 `p` 结点前,还需查找 `p` 的前驱,这会增加遍历的时间。为提高效率,查找结点时通常要同时记录其前驱地址。

在双向链表中每个结点有指向前驱和后继的指针,知道结点地址就能获取其前后结点地址。插入、删除操作只需修改指针链,无须额外查找前驱,降低了时间消耗。不过,若不知道插入位置或要删除的结点地址,仍需遍历链表查找目标结点。

在依据特定条件找到结点地址后,即可在双向链表中顺利进行插入或删除操作。

【算法 1.14】双向循环链表的插入。

算法步骤如下。

①为要插入的数据创建一个新的双向链表结点 `s`,并将给定的数据 `e` 存入该结点的数据域。

②找到插入位置,这里假设插入位置为结点 `p` 之后。

③调整指针,将新结点 `s` 插入结点 `p` 之后,需要同时更新新结点 `s` 的前驱和后继指针及相关



结点 (结点 p 和 p 的后继结点) 的前驱和后继指针。

④插入操作完成。

算法实现如下。

// 在双向循环链表 L 中, 将数据 e 插入结点 p 之后

```
void inselem(DuLinkList L, ElemType e, DuLNode *p)
{
    // 为新结点分配内存空间
    DuLNode *s = (DuLNode *)malloc(sizeof(DuLNode));
    // 将给定的数据  $e$  存入新结点的数据域
    s->data = e;
    // 找到  $p$  的后继结点  $r$ 
    DuLNode *r = p->next;
    // 设置新结点  $s$  的后继指针指向  $r$ 
    s->next = r;
    // 设置  $r$  的前驱指针指向  $s$ 
    r->prior = s;
    // 插在结点  $p$  之后
    // 设置  $p$  的后继指针指向  $s$ 
    p->next = s;
    // 设置  $s$  的前驱指针指向  $p$ 
    s->prior = p;
}
```

【算法 1.15】双向循环链表的删除。

算法步骤如下。

①找到要删除的结点 p 。

②找到结点 p 的前驱结点 q 和后继结点 r 。

③调整指针, 将结点 p 从链表中删除, 需要同时更新 q 和 r 的指针。

④释放被删除结点 p 的内存空间。

⑤删除操作完成。

算法实现如下。

// 在双向循环链表 L 中, 删除地址为 p 的结点

```
int delclm(DuLinkList L, DuLNode *p)
{
    // 检查链表是否为空 (头结点的 next 指向自身)
    if (L->next == L) {
        printf("链表为空, 无法删除结点。\\n");
        return 0;
    }
}
```



```
// 检查 p 是否为空或是否为头结点（头结点不能被删除）
if(p == NULL || p == L) {
    printf("无效的删除操作：结点为空或试图删除头结点。\\n");
    return 0;
}
// 找到 p 的前驱结点 q 和后继结点 r
DuLNode *q = p->prior;    // p 的前驱结点
DuLNode *r = p->next;     // p 的后继结点
// 更新 q 和 r 的指针，将 p 结点从链表中移除
q->next = r;              // q 的后继指向 r
r->prior = q;             // r 的前驱指向 q
// 释放 p 结点的内存空间
free(p);
printf("结点删除成功。\\n");
return 1;
}
```

任务实现

1. 链表的存储表示

考试报名管理系统使用两个结构体 Candidate 和 LinkedList 来表示考生和考生表。利用链表的数据结构存储考生信息，将每个考生信息抽象为链表中的一个结点。首先定义“考生”结构体，包括姓名、报名号、考试科目等信息；然后通过链表的基本操作实现增、删、查、改等功能；最后通过主函数调用这些操作，完成考试报名管理系统。

```
#define MAX_NAME 50
#define MAX_SUBJECT 50
// 考生信息结构体
typedef struct Node
{
    char name[MAX_NAME];    // 姓名
    int regNumber;          // 报名号
    char subject[MAX_SUBJECT]; // 考试科目
    struct Node* next;      // 指向下一个结点的指针
} Candidate;
// 链表结构体
typedef struct
{
    Candidate *head; // 链表头指针
} LinkedList;
```



视频

项目1任务2任务
实现讲解



2. 链表的主要操作

在考试报名管理系统中,对链表的主要操作包括初始化链表、录入考生信息、查询考生信息、删除考生信息、修改考生信息、输出所有考生信息。

1) 初始化链表

初始化链表,设置头指针为 NULL,表示链表为空。

```
void initLinkedList(LinkedList* list)
{
    list->head = NULL;
}
```

2) 录入考生信息

将考生信息录入系统,并插入链表中。

```
int insertCandidate(LinkedList *list,char name[],int regNumber,
char subject[])
{
    Candidate *newCandidate = (Candidate*)
malloc(sizeof(Candidate));
    if(!newCandidate) {
        printf("内存分配失败.\n");
        return -1;
    }
    strcpy(newCandidate->name,name);
    newCandidate->regNumber = regNumber;
    strcpy(newCandidate->subject,subject);
    newCandidate->next = list->head;
    list->head = newCandidate;
    return 0;
}
```

3) 查询考生信息

根据考生姓名或报名号查询考生信息。

```
Candidate *findCandidate(LinkedList *list,char name[],int regNumber)
{
    Candidate *current = list->head;
    while(current != NULL) {
        if ((strcmp(current->name, name) == 0) || (current->
regNumber == regNumber)) {
            return current;
        }
        current = current->next;
    }
}
```



```
return NULL;
```

```
}
```

4) 删除考生信息

根据考生姓名或报名号删除考生信息。

```
int deleteCandidate(LinkedList* list, char name[], int regNumber)
{
    Candidate *current = list->head;
    Candidate *prev = NULL;
    while (current != NULL) {
        if((strcmp(current->name, name) == 0) || (current->
regNumber == regNumber)) {
            if(prev == NULL) { // 删除头结点
                list->head = current->next;
            } else { // 删除中间或尾结点
                prev->next = current->next;
            }
            free(current);
            return 0;
        }
        prev = current;
        current = current->next;
    }
    return -1;
}
```

5) 修改考生信息

根据考生姓名或报名号修改考生信息。

```
int editCandidate(LinkedList *list, char name[], int regNumber, char
newSubject[])
{
    Candidate *candidate = findCandidate(list, name, regNumber);
    if (candidate != NULL) {
        strcpy(candidate->subject, newSubject);
        printf(" 考生信息已更新。\\n");
        return 0;
    }
    return -1;
}
```

6) 输出所有考生信息

显示系统中所有考生信息。



图文

链表实现考试报名管
理系统的完整代码



```
void displayCandidates(LinkedList *list)
{
    Candidate *current = list->head;
    printf("%-20s %-10s %-20s\n", "姓名", "报名号", "考试科目");
    while (current != NULL) {
        printf("%-20s %-10d %-20s\n", current->name, current->
regNumber, current->subject);
        current = current->next;
    }
}
```

3. 链表的特点

1) 优点

(1) 动态存储优势：链表是一种动态数据结构，能够根据实际需求动态分配和释放内存。在考试报名管理系统中，考生数量可能不断变化，链表可以灵活地插入或删除结点，而无须预先分配固定大小的存储空间。这种动态性使得链表能够高效地处理不确定数量的数据，避免了顺序表中常见的空间浪费或频繁扩容问题。

(2) 插入和删除效率高：链表在插入和删除操作时，只需修改相邻结点的指针，而无须移动大量元素。这使得插入和删除操作的时间复杂度为 $O(1)$ （假设已经找到了插入或删除位置）。在考试报名管理系统中，考生信息的新增和删除操作较为频繁，链表能够高效地完成这些操作，而不会因数据量的增加而显著降低性能。

(3) 易于扩展和维护：链表的结构简单，每个结点独立存储数据和指向下一个结点的指针。这种结构使得链表易于扩展和维护，开发者可以方便地实现各种操作（如插入、删除、查找等），而无须对整个数据结构进行复杂的调整。此外，链表的结点可以分散存储在内存中的不同位置，不会受到连续存储空间的限制。

(4) 适合动态数据处理：在考试报名管理系统中，考生信息的增、删、改、查操作频繁且动态性强。链表的动态特性使其能够很好地适应这种场景，尤其是在考生数量变化较大时，链表能够灵活地调整存储结构，而不会像顺序表那样因固定大小而受限。

2) 缺点

(1) 访问效率低：链表不支持随机访问，只能从头结点开始逐个遍历结点来查找目标数据。这意味着查找操作的时间复杂度为 $O(n)$ ，在数据量较大时，访问效率较低。在考试报名管理系统中，如果需要频繁地根据考生姓名或报名号查询考生信息，链表的查找效率可能会成为瓶颈。

(2) 额外的存储开销：链表的每个结点除了存储实际数据外，还需要额外存储指向下一个结点的指针（或前驱和后继指针）。这导致链表的存储密度较低，相比顺序表，链表需要占用更多的内存空间来存储相同数量的数据。

(3) 缓存不友好：由于链表的结点在内存中是分散存储的，无法充分利用 CPU 的缓存机制。当访问链表中的结点时，CPU 缓存无法像顺序表那样预加载相邻元素，这可能导致频繁的缓存未命中，从而降低访问速度。在处理大量数据时，这种缓存不友好的特性可能会进一步影响系统的性能。

(4) 操作复杂性：尽管链表的插入和删除操作效率高，但在实现这些操作时需要仔细处理指



针的指向，避免出现指针错误或内存泄漏等问题。此外，链表的某些操作（如逆序、排序等）实现起来相对复杂，需要额外的逻辑支持。

基于链表的这些特点，链表适用于以下两种情况：数据量不确定且动态变化频繁的场景。例如，在考试报名管理系统中，考生数量难以预估且增删操作频繁，链表能够灵活地适应这种动态变化；插入和删除操作频繁但查询操作较少的场景，链表在插入和删除操作上的高效性使其适合处理动态数据，但在需要频繁随机访问数据时，链表的性能可能不如顺序表。

针对链表的缺点，后续可以考虑以下优化措施：结合其他数据结构。例如，使用哈希表来优化查找操作，或者结合顺序表和链表的优点，设计一种混合数据结构，以提高系统的整体性能；优化链表操作，通过双向链表或循环链表等变体，提高某些操作的效率；或者在链表中引入索引机制，减少查找时间；缓存优化，通过合理设计链表的存储方式，尽量减少缓存未命中次数。例如，将频繁访问的结点存储在连续的内存空间中。

通过以上总结，可以更全面地理解链表在考试报名管理系统中的适用性和局限性，并为后续的优化和改进提供参考。



动手实践

链表作为一种灵活的线性表实现方式，以其动态存储和高效的插入、删除操作能力，在数据结构领域中占据重要地位。接下来通过一个实践任务——合并两个有序链表，来加深对链表概念和基本操作的理解。

【实践任务】

本任务要求将两个已经按非递减顺序排列的单链表 La 和 Lb 合并成一个新的有序链表 Lc。在合并过程中，Lc 中的元素也必须保持非递减顺序。此任务涉及链表的结点操作、元素比较与链接等操作，是链表基本操作的直接应用。

【算法步骤】

- ①初始化链表：创建一个新的空链表 Lc。
- ②设置指针：使用指针 pa 和 pb 分别指向链表 La 和 Lb 的第一个结点，同时设置指针 pc 指向链表 Lc 的尾部。
- ③比较与链接：使用循环比较 La 和 Lb 当前结点的值，将较小的结点链接到 Lc 的尾部，并移动对应的指针。
- ④处理剩余结点：当其中一个链表的结点全部处理完毕后，将另一个链表的剩余部分直接链接到 Lc 的尾部。
- ⑤完成合并：所有结点处理完毕后，合并过程结束。

【算法实现】

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// 定义链表结点结构体
typedef struct Candidate
{
```



视频

项目1任务2动手
实践讲解



```
    char name[50];
    int regNumber;
    char subject[50];
    struct Candidate *next;
} Candidate, *LinkedList;
// 初始化链表
void initLinkedList(LinkedList *L)
{
    *L = NULL;
}
// 创建新结点
Candidate* createNode(char *name,int regNumber,char *subject)
{
    Candidate *newNode = (Candidate*)malloc(sizeof(Candidate));
    if(!newNode) {
        return NULL;
    }
    strcpy(newNode->name,name);
    newNode->regNumber = regNumber;
    strcpy(newNode->subject,subject);
    newNode->next = NULL;
    return newNode;
}
// 合并两个有序链表
int MergeList(LinkedList La,LinkedList Lb,LinkedList *Lc)
{
    // 初始化 Lc
    initLinkedList(Lc);
    // 设置指针
    Candidate *pa = La;
    Candidate *pb = Lb;
    Candidate *pc = NULL; // 指向 Lc 的尾部结点
    // 比较并链接结点
    while(pa != NULL && pb != NULL) {
        Candidate* newNode = (Candidate*)malloc(sizeof(Candidate));
        if(strcmp(pa->name, pb->name) <= 0) { // 按姓名升序比较
            strcpy(newNode->name,pa->name);
            newNode->regNumber = pa->regNumber;
            strcpy(newNode->subject,pa->subject);
```



```
        pa = pa->next;
    } else {
        strcpy(newNode->name,pb->name);
        newNode->regNumber = pb->regNumber;
        strcpy(newNode->subject,pb->subject);
        pb = pb->next;
    }
    // 将新结点链接到 Lc
    if(*Lc == NULL) {
        *Lc = newNode;
    } else {
        pc->next = newNode;
    }
    pc = newNode;
}
// 处理剩余结点
while(pa != NULL) {
    pc->next = pa;
    pc = pa;
    pa = pa->next;
}
while(pb != NULL) {
    pc->next = pb;
    pc = pb;
    pb = pb->next;
}
pc->next = NULL;    // 确保新链表的尾部指针为 NULL
return 0;           // 合并成功
}
// 打印链表
void printList(LinkedList L)
{
    Candidate *current = L;
    while(current != NULL) {
        printf("Name: %s,RegNumber: %d,Subject: %s\n",current->
name,current->regNumber,current->subject);
        current = current->next;
    }
}
```



```
// 释放链表内存
void freeList(LinkedList *L)
{
    Candidate *current = *L;
    while(current != NULL) {
        Candidate *next = current->next;
        free(current);
        current = next;
    }
    *L = NULL;
}
```



图文

合并两个有序链表的
完整代码

【算法分析】

时间复杂度：该算法需要遍历两个链表的所有结点，因此算法的时间复杂度为 $O(m+n)$ ，其中， m 和 n 分别是链表 La 和 Lb 的长度。

空间复杂度：该算法需要创建一个新的链表 Lc 来存储合并后的结果，其大小为两个输入链表长度之和，因此算法的空间复杂度为 $O(m+n)$ 。此外，算法中还涉及动态分配新结点的内存，但这些内存最终会被释放。



习 题

一、选择题

- 关于链式存储的线性表，下列说法正确的是 ()。
 - 链式存储的线性表不支持随机访问
 - 链式存储的线性表需要连续的存储空间
 - 链式存储的线性表插入和删除操作效率较低
 - 链式存储的线性表无法动态扩展存储空间
- 在一个单链表中，删除一个结点的时间复杂度为 ()。
 - $O(1)$
 - $O(n)$
 - $O(\log \geq n)$
 - $O(n^2)$
- 对于链式存储的线性表，查找一个值为 e 的元素的时间复杂度为 ()。
 - $O(1)$
 - $O(n)$
 - $O(\log \geq n)$
 - $O(n^2)$
- 链式存储结构的主要优点是 ()。
 - 插入和删除操作效率高
 - 随机访问效率高
 - 存储密度大
 - 不需要额外的存储空间
- 链式存储结构的一个主要缺点是 ()。
 - 不支持动态存储
 - 需要额外的指针存储空间
 - 插入操作复杂
 - 删除操作复杂
- 下列 () 不是链式存储线性表的基本运算。
 - 初始化空表
 - 查找指定元素的位置

- ## 二、综合题

- 61