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

丛书策划　张荣昌
责任编辑　王　清　孟海江
封面设计

大数据、云计算、人工智能、信息安全人才培养丛书
“互联网+”新形态一体化教材

大数据
大数据基础
数据可视化
数据清洗与治理
Hadoop应用与开发
数据挖掘基础
SEO搜索引擎优化
MySQL数据库
R语言程序设计
Go语言程序设计

云计算
云计算基础
虚拟化与容器
云安全运维
网络工程与组网技术
现代通信技术
路由交换技术
无线网络技术
现代网络SDN技术
数据网组建与维护
局域网组建与维护
云数据中心架构与SDN技术

人工智能
人工智能基础
物联网基础
深度学习
机器学习

信息安全
信息安全基础
Linux服务器安全高级运维
Web安全与防御
防火墙技术与应用
计算机病毒与防范
数据存储与恢复
密码学基础
计算机网络安全运维
网络设备配置与综合实战
无线网络安全技术
VPN虚拟专用网安全
终端数据存储与恢复
工控安全
渗透测试
恶意代码分析
网络空间安全态势感知
数据库安全技术
网络安全协议
企业级数据安全与灾备管理
信息安全法律法规
终端数据安全及防泄密

专业基础
Linux操作系统基础
计算机网络基础
Ubuntu服务器管理
Windows Server 2016配置与管理
数据结构
Python程序设计
Java程序设计
C语言程序设计
C#程序设计
Android程序设计
XML基础教程
JavaScript基础教程
Web前端开发
OpenStack应用与开发
● 数据结构与算法
静态网页设计与制作
HTML5与JavaScript程序设计
数据库设计与应用
数据库应用基础
UML建模与设计模式
ERP原理与应用
综合布线

教材配套
精品教学资源包
在线课程　素材源码　教学课件　习题答案

扫描二维码
关注上海交通大学出版社
官方微信

ISBN 978-7-313-26988-1
9 787313 269881 >
定价：56.00元

“互联网+”新形态一体化精品教材

数据结构与算法

主编◎黄龙军

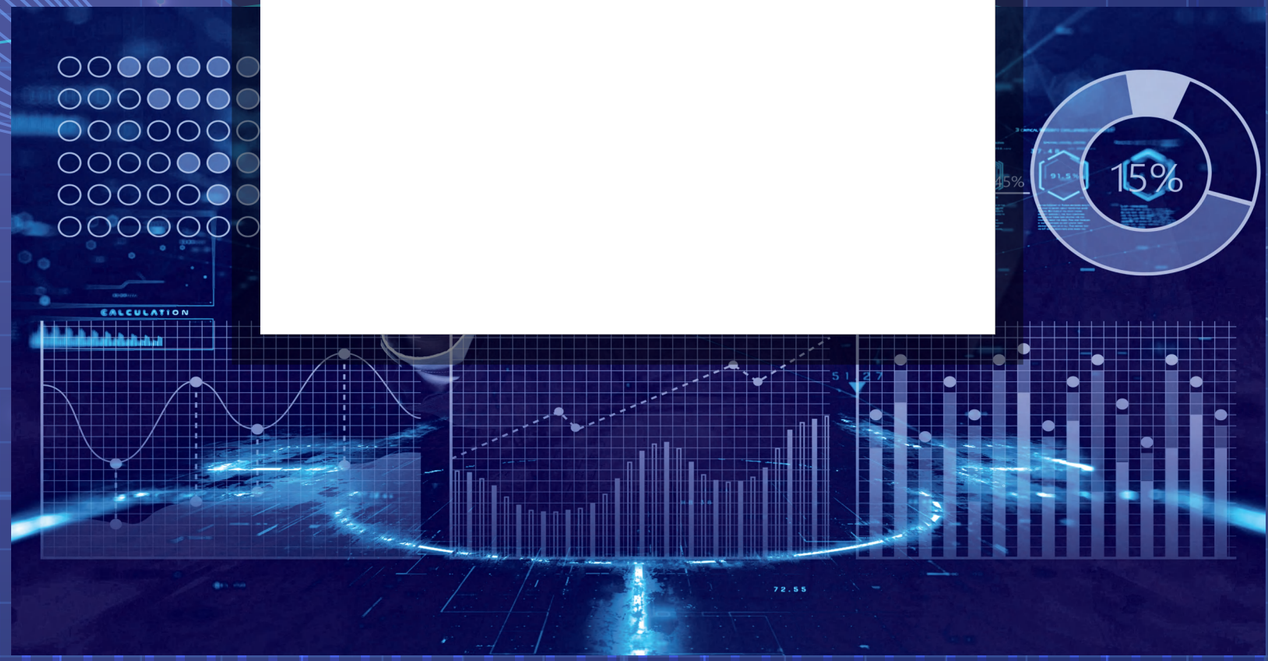
上海交通大学出版社

大数据、云计算、人工智能、信息安全人才培养丛书
浙江省线上一流课程“数据结构”配套教材
“互联网+”新形态一体化教材

数据结构与算法

SHUJU JIEGOU YU SUANFA

主编◎黄龙军



上海交通大學出版社
SHANGHAI JIAO TONG UNIVERSITY PRESS

大数据、云计算、人工智能、信息安全人才培养丛书
浙江省线上一流课程“数据结构”配套教材
“互联网+”新形态一体化教材

数据结构与算法

SHUJU JIEGOU YU SUANFA

主编◎黄龙军

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

内容提要

本书介绍数据结构和算法相关知识，引入部分 C++ 标准模板库（standard template library，STL）和程序设计竞赛知识，培养学生的计算思维、分析与解决具体问题的能力及创新能力。本书包括绪论、线性表、栈与队列、其他线性结构、树、图、查找、排序 8 章内容，重点介绍的内容主要包括线性表、栈、队列、二叉树和图等数据结构的概念、逻辑结构、存储结构及相应算法，二分查找、二叉排序树和哈希查找等查找算法，插入排序、快速排序、堆排序和二路归并排序等排序算法。

本书力求把数据结构的抽象理论知识和算法实现讲得通俗易懂，以 C++ 语言描述算法，注重问题求解，可作为高等院校计算机类、电子信息类、数据科学与大数据技术及应用统计学等相关专业的教材，也可作为程序设计类竞赛和信息学竞赛的参赛者、数据结构与算法的教师或自学者的参考用书。

图书在版编目（CIP）数据

数据结构与算法 / 黄龙军主编 . —上海：上海交通大学出版社，2022.7（2023.2 重印）

ISBN 978-7-313-26988-1

Ⅰ. ①数… Ⅱ. ①黄… Ⅲ. ①数据结构—高等学校—教材 ②算法分析—高等学校—教材 ③ C++ 语言—数据结构—高等学校—教材 Ⅳ. ① TP311.12 ② TP301.6

中国版本图书馆 CIP 数据核字（2022）第 108164 号

数据结构与算法
SHUJU JIEGOU YU SUANFA

主　　编：黄龙军
出版发行：上海交通大学出版社
邮政编码：200030
印　　制：北京荣玉印刷有限公司
开　　本：889 mm × 1194 mm　1/16
字　　数：393 千字
版　　次：2022 年 7 月第 1 版
书　　号：ISBN 978-7-313-26988-1
定　　价：56.00 元

地　　址：上海市番禺路 951 号
电　　话：6407 1208

经　　销：全国新华书店
印　　张：15.5

印　　次：2023 年 2 月第 2 次印刷

编写委员会

主　编　黄龙军

副主编　范立新　唐开山　胡珂立

编　委　冯　晟　李　平　张　骏

在线课程学习指南

本书是浙江省线上一流课程“数据结构”的配套教材，读者可在浙江省高等学校在线开放课程共享平台（www.zjooc.cn）在线学习该课程。

一、搜索课程

进入浙江省高等学校在线开放课程共享平台（www.zjooc.cn），登录平台账号（浙江省内高校在校学生请联系老师获取平台账号，其他读者可按照社会学员注册），在右上方的搜索栏中输入“数据结构”，单击搜索。在搜索结果中选择对应课程（授课教师：黄龙军）。

二、在线学习

选择对应课程后进入课程界面，单击“立即参加”即可进行学习。

前言

PREFACE

数据结构课程的理论性和实践性都很强。数据结构课程的理论知识比较抽象，有些同学在理解和掌握这些理论知识方面存在一定的困难。对于更多的同学而言，在算法实现方面存在较大的困难。在数据结构的教材中，有些理论知识艰深，有些算法描述抽象，有些未提供在线编程，从而导致同学们存在畏难情绪、理论知识掌握不够扎实、算法设计与实现能力不足。

为此，我们在教学过程中简化理论知识，细化代码实现，力争把数据结构的理论知识和算法实现讲得通俗易懂，并通过在线编程提升同学们的实践能力，从而培养他们的计算思维、分析与解决具体问题的能力及创新能力。基于此，我们总结多年数据结构课程的教学探索与实践，整合绍兴市精品在线开放课程“数据结构”建设资源，结合程序设计类竞赛编写了本书。

本书共 8 章，第 1 章是绪论；第 2 章至第 6 章介绍常用的数据结构，即线性结构、树结构和图结构，主要包括线性表、栈、队列、二叉树和图等数据结构的概念、逻辑结构、存储结构及相应算法；第 7 章介绍顺序查找、二分查找、二叉排序树和哈希查找等查找算法；第 8 章介绍插入排序、简单选择排序、冒泡排序、快速排序、堆排序和二路归并排序等排序算法。另外，本书中介绍了 STL 之 vector、list、stack、queue、set 和 map 等容器及若干算法。使用本书的教师可以酌情选择教学内容（带 * 的内容为选讲），或者对书中内容进行适当补充和拓展。

本书与在线测评系统（online judge，OJ）相结合，简化理论知识的讲解，注重运用知识求解 OJ 具体问题。本书中的编程例题、习题主要来自 OJ。在编写本书的过程中，编者参阅了一些数据结构和程序设计方面的著作，深受启发，书中部分内容及部分例题和习题参考或改编自这些著作及其网络资源，在这里对这些著作的作者及相关人员表示衷心的感谢！书中大部分编程例题和习题来自绍兴文理学院 OJ（happyLearn Online Judge，HLOJ，网址：http://acm.usx.edu.cn），这离不开绍兴文理学院数据结构课程组教师历年来的辛勤工作，在此表示由衷的感谢！书中部分编程例题和习题来源于程序设计类实验辅助教学平台（programming teaching assistant，PTA，网址：https://pintia.cn/），在此对陈越教授和刘春英教授以及其他相关的老师和同学们一并表示衷心的感谢！

为便于读者编程练习，书中的编程习题及“在线题目求解”节中的例题都标注了出处；另外，我们在 PTA 组建了数据结构编程练习的题目集，包含书中编程方面的例题、习题及若干拓展编程题。使用本教材的教师可联系编者获取该题目集的分享码方便自建题目集供学生练习，其他读者可联系编者提供 PTA 注册邮箱获得该题目集的实践资格。编者联系邮箱：hlj_jlh@163.com。此外，本书还配有服

务于本书的教学课件和习题答案，有需要者可致电 13810412048 或发邮件至 2393867076@qq.com。

本书落实立德树人根本任务，贯彻《高等学校课程思政建设指导纲要》精神，通过“思政链接”模块，将专业知识与思政教育有机结合，推动价值引领、知识传授和能力培养紧密结合。

编者在编写时力图把抽象的理论知识写得通俗易懂，把算法实现写得易于理解。然而受限于编者的能力和水平，书中难免存在错误和不足之处，恳请读者批评指正。

编　者

2021 年 11 月

目录

CONTENTS

第 1 章　绪论 …… 1

1.1　数据结构的研究内容 …… 2
1.2　数据结构的基本概念与结构 …… 3
1.3　算法与算法分析 …… 4
1.3.1 算法基础知识 …… 4
1.3.2 算法的时间复杂度分析 …… 5
1.3.3 算法的空间复杂度分析 …… 7
1.4　在线题目求解 …… 8

第 2 章　线性表 …… 13

2.1　线性表基础 …… 14
2.2　顺序表 …… 14
2.2.1 顺序表基础 …… 14
2.2.2 顺序表的定义与实现 …… 15
2.3　STL 之 vector …… 19
2.4　链表 …… 21
2.4.1 链表概述 …… 21
2.4.2 单链表基本操作及其实现 …… 22
2.4.3 其他形式的链表简介 …… 28
2.5　STL 之 list …… 28
2.6　在线题目求解 …… 32

第 3 章　栈与队列 …… 45

3.1　栈基础 …… 46
3.1.1 顺序栈 …… 46
3.1.2 链栈 …… 48
3.2　STL 之 stack …… 50
3.3　栈应用举例 …… 51
3.4　队列基础 …… 58
3.4.1 循环队列 …… 58
3.4.2 链式队列 …… 60
3.5　STL 之 queue …… 62
3.6　队列应用举例 …… 63
3.7　在线题目求解 …… 65

第 4 章　其他线性结构 …… 79

4.1　串 …… 80
4.1.1 串的基础知识 …… 80
4.1.2 STL 之 string …… 80
4.1.3 串的模式匹配算法 …… 82
4.2　数组 …… 84
4.2.1 一维数组 …… 84
4.2.2 二维数组 …… 85
4.3　广义表 …… 86
4.4　在线题目求解 …… 87

第 5 章　树 …… 95

5.1　树的概述 …… 96
5.2　二叉树基础 …… 97
5.2.1 定义与术语 …… 97
5.2.2 二叉树的性质 …… 98
5.2.3 二叉树的存储结构 …… 100
5.3　二叉树的遍历 …… 102
5.3.1 遍历基础知识 …… 102
5.3.2 遍历算法实现 …… 103
5.4　二叉树遍历算法的应用 …… 104
5.5　哈夫曼树与哈夫曼编码 …… 108
5.5.1 哈夫曼树基础知识 …… 108
5.5.2 哈夫曼树及其编码的算法实现 …… 110
5.6　树与森林 …… 112
5.6.1 树的存储表示 …… 112

5.6.2 二叉树与森林之间的转换 …… 113
5.6.3 树与森林的遍历 …… 114
5.7 并查集 ……115
5.8 在线题目求解 ……117

第 6 章 图 …… 131

6.1 图的概述 ……132
6.1.1 图的定义 …… 132
6.1.2 图的基本术语 …… 132
6.2 图的存储结构 ……134
6.2.1 邻接矩阵 …… 134
6.2.2 邻接表 …… 137
6.2.3 链式前向星 …… 139
6.3 图的遍历 ……141
6.3.1 图的深度优先搜索 …… 141
6.3.2 图的广度优先遍历 …… 142
6.4 图的最小生成树 ……144
6.4.1 Prim 算法 …… 144
6.4.2 Kruskal 算法 …… 148
6.5 最短路径 ……150
6.5.1 Dijkstra 算法 …… 150
6.5.2 Floyd 算法 …… 152
6.6 拓扑排序 ……154
6.6.1 拓扑排序基础 …… 154
6.6.2 拓扑排序算法实现 …… 155
6.7 在线题目求解 ……156

第 7 章 查找 …… 177

7.1 查找的概念与术语 ……178
7.2 顺序查找与二分查找 ……178
7.2.1 顺序查找 …… 179
7.2.2 二分查找 …… 180
7.3 二叉排序树 ……181
7.3.1 二叉排序树基础 …… 181
7.3.2 二叉排序树的查找算法与插入算法 … 182
7.3.3 二叉排序树的删除算法 …… 184
7.3.4 二叉排序树的查找性能分析 …… 187
7.4 平衡二叉树 ……187
7.4.1 平衡二叉树基础 …… 187
7.4.2 构造平衡二叉树的一般方法 …… 188
7.5 哈希查找 ……190
7.5.1 哈希查找基础 …… 190
7.5.2 哈希查找之处理冲突 …… 192
7.5.3 哈希查找的算法实现 …… 195
7.6 STL 之 set 与 map ……197
7.6.1 STL 之 set …… 198
7.6.2 STL 之 map …… 199
7.7 在线题目求解 ……200

第 8 章 排序 …… 211

8.1 排序概述 ……212
8.1.1 排序的定义与术语 …… 212
8.1.2 排序的分类 …… 212
8.2 插入排序 ……213
8.2.1 直接插入排序 …… 213
8.2.2 折半插入排序 …… 215
8.2.3 希尔排序 …… 215
8.3 简单选择排序和冒泡排序 ……217
8.3.1 简单选择排序 …… 217
8.3.2 冒泡排序 …… 218
8.4 快速排序 ……219
8.4.1 快速排序基础 …… 219
8.4.2 快速排序算法实现 …… 221
8.5 堆排序 ……222
8.5.1 堆排序基础 …… 222
8.5.2 堆排序算法实现 …… 226
8.6 二路归并排序 ……227
8.6.1 二路归并排序基础 …… 227
8.6.2 二路归并排序算法实现 …… 229
8.7 STL 与排序 ……230
8.7.1 STL 之 sort …… 230
8.7.2 STL 之 nth_element …… 232
8.8 在线题目求解 ……233

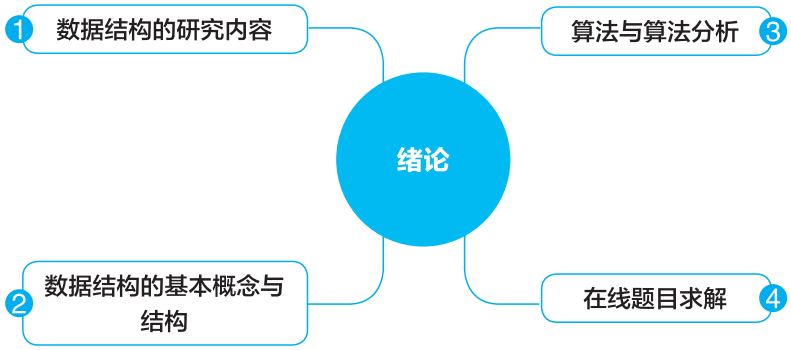
参考文献 …… 239

第 1 章 绪 论

学习目标

1. 记忆和理解数据结构的基本概念和术语。
2. 记忆和理解算法的含义及其特性。
3. 初步学会算法分析的方法。
4. 理解如何更好地提高算法的时间、空间效率。
5. 树立学好专业知识、助力中国梦的信念。
6. 获取仰望星空、探索创新的精神动力。

知识导图



导言

数据结构是介于数学、计算机硬件和计算机软件三者之间的一门核心课程，是诸多高校计算机相关专业考研、考博的常考课程，是程序设计竞赛的必备课程，也是计算机类专业及其他相关专业的必修课程。学好数据结构对于这些专业的学生而言，至关重要。

思政链接：
中国梦

通过学习数据结构课程，能够分析研究计算机加工对象的特性，获得其逻辑结构，根据需求选择合适的存储结构及其相应的算法；学习一些常用算法，如查找和排序；进行复杂程序设计的训练；初步掌握算法的时间、空间效率分析技术。

本章介绍数据结构的研究内容、数据结构的基本概念与结构、算法与算法分析等方面的内容，重点关注数据元素、数据项、数据结构、逻辑结构、存储结构、算法、时间复杂度、空间复杂度等概念与结构，算法的特性，算法的时间复杂度和空间复杂度分析方法。最后，通过在线题目的求解，表明有效提升算法性能的必要性。

1.1 数据结构的研究内容

思政链接：
沃斯

数据结构主要研究非数值计算问题，重点考虑如何合理地组织数据，如何高效地处理数据。

著名的计算机科学家、图灵奖得主尼古拉斯·沃思（Niklaus Wirth）曾提出一个公式：

$$\text{算法} + \text{数据结构} = \text{程序}$$

其中，算法是处理问题的策略，数据结构是问题的数学模型及其数据组织形式，程序是为实现特定目标或解决特定问题而用程序设计语言编写的指令序列。

数值计算问题的数学模型是数学方程，而非数值计算问题则无法用数学方程建立数学模型。对于非数值计算程序设计问题，首先要考虑操作对象在计算机中的组织方式。下面举例说明。

例 1.1.1 学生信息管理问题

对于学生信息管理系统中的学生信息，可以采用二维表的形式组织，如表 1-1 所示。

表 1-1　学生信息表

学号	姓名	专业
21145141	张无忌	计算机科学与技术
21539141	令狐冲	网络工程
21145241	郭靖	计算机科学与技术
21536141	杨过	数据科学与大数据

在学生信息表中，除了第一个和最后一个学生外，其他每个学生的前一个位置和后一位置都有且仅有一个学生，是一种“一对一”的线性关系，可视为线性表。本例的数学模型为线性表。

例 1.1.2 对弈问题

在对弈问题中，前一阶段的一种局势可以演变为后一阶段的若干种局势。例如，图 1-1 所示的三子棋局势变化中，上一步的一种局势可能产生下一步的多种局势。可见，对弈问题的局势变化是一种

“一对多”的关系，可用树结构表达。在树结构中，一个父节点可能有多个子节点，而每个节点最多只有一个父节点。本例的数学模型为树结构。

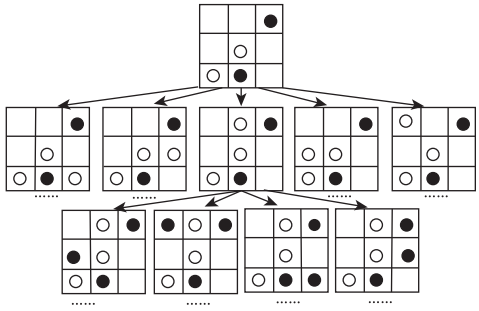


图 1-1　三子棋局势变化示意图

例 1.1.3 最短路径问题

在自驾旅游规划时，若希望从始发城市到目的城市的行车距离尽可能短，则该如何安排驾车路线呢？例如，图 1-2 所示把 5 个城市抽象为 5 个顶点 1、2、3、4、5，任意一个城市都可能与其他多个城市之间有直达道路，即任意两个顶点之间是一种“多对多”的关系，这是一种图结构。设始发城市为 1，目的城市为 2，则从 1 到 2 有 1 → 2、1 → 4 → 2、1 → 5 → 3 → 2、1 → 4 → 3 → 2 等多条路线可走，行车距离分别为 90、80、70、60，因此最佳路线是距离为 60 的 1 → 4 → 3 → 2，这是图结构中的一个典型应用问题——最短路径问题。本例的数学模型为图结构。

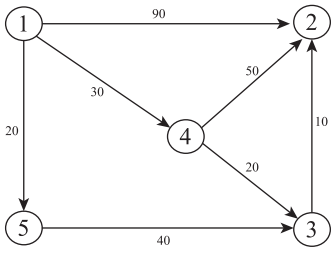


图 1-2　最短路径问题

从以上三个例子可见，非数值计算问题的数学模型是线性表、树结构和图结构等。因此，数据结构主要研究非数值计算程序设计问题中的操作对象以及它们之间的关系和操作。

1.2 数据结构的基本概念与结构

1. 基本概念

数据（data）是客观事物的符号表示，是所有能输入计算机中并能被计算机程序处理的符号的总称。数据可分为数值型数据（如整数、实数等）和非数值型数据（如字符串、图形、图像、声音和动画等）。

数据元素（data element）是数据的基本单位，也称元素或记录，在计算机中通常作为一个整体考虑。例如，表 1-1 中的每条学生记录都是一个数据元素。

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，表 1-1 中的学生学号、姓名和专业都是数据项。

数据对象（data object）是相同特性的数据元素的集合，是数据的一个子集。

例如，表 1-1 的学生信息表及英文字母子集 S={'A', 'B', 'C', 'D', 'E', 'F'} 都是数据对象。

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。或者说，数据结构是带“结构”的数据元素的集合，其中，“结构”是指数据元素之间存在的关系。

数据的逻辑结构、存储结构及运算是数据结构的三要素。

2. 逻辑结构

逻辑结构是指数据元素间抽象化的相互关系，与数据的存储无关，独立于计算机，是从具体问题抽象出来的数学模型。

对于逻辑结构而言，数据结构可归结为线性结构和非线性结构，其中，非线性结构主要包括树结构和图结构，如图 1-3 所示。

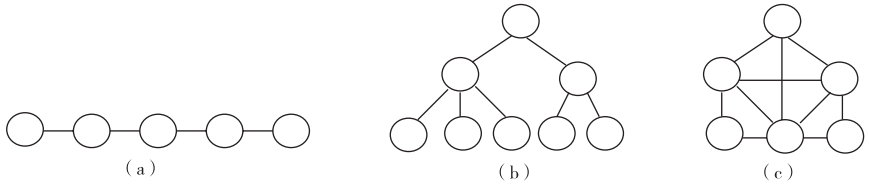


图 1-3　常见的数据结构

(a) 线性结构；(b) 树结构；(c) 图结构

3. 存储结构

存储结构也称物理结构，是数据元素及其关系在计算机存储器中的存储方式。存储结构可分为顺序存储结构和链式存储结构。

顺序存储结构借助元素在存储器中的相对位置来表示数据元素间的逻辑关系，一般采用数组表示，如图 1-4 所示。图中，元素 3 与元素 1 间隔了两个元素位置。

图 1-4　顺序存储结构

链式存储结构借助指示元素存储地址的指针来表示数据元素间的逻辑关系，一般采用链表表示。如图 1-5 所示是带头节点的单链表，头节点的地址设为 5000，存放在头指针 *head* 中；4 个元素相应节点的地址分别假设为 2000、3000、4000、1000。可见，后一个元素对应节点的地址存储在前一个节点的指针域，即前一个节点的指针域指向后一个节点。

图 1-5　链式存储结构

1.3　算法与算法分析

1.3.1 算法基础知识

1. 算法的定义及特性

算法是为解决某类问题而规定的一个有限长的操作序列（步骤）。一个算法必须满足有穷性、确定性、可行性、有输入、有输出 5 个特性。

（1）有穷性是指算法的步骤是有限的，且每个步骤都能在有限时间内完成。

（2）确定性是指算法的每个步骤都有确切规定，不会产生二义性。

（3）可行性是指算法中的每个步骤都是可行的，不会无法实现。

（4）有输入是指一个算法有 0 个或多个输入。因一个算法通常以一个函数来描述，故算法的输入往往以形参表示，在被调用时从主调函数获得输入值。0 个输入的算法通常包含一些初始化语句。

（5）有输出是指一个算法有 1 个或多个输出，表示算法进行信息加工后得到的结果。因一个算法通常以一个函数来描述，故算法的输出经常以函数的返回值或引用类型的形参表示。

2. 算法的评价

设计和评价算法时，通常应考虑达到正确性、可读性、健壮性、高效性等目标。

（1）算法的正确性是指算法能在有限运行时间内得到正确的结果。算法的正确性通常不易证明，可以通过精心设定测试数据的方法来表明算法的正确性。在程序设计竞赛中，经常使用这种方法。

（2）算法的可读性是指算法容易为人阅读和理解。可以通过添加适当的注释语句并合理缩排代码来提升算法的可读性。

（3）算法的健壮性是指算法在遇到非法输入时也能正常结束而不会崩溃。可以通过对非法输入进行特判来提高算法的健壮性。

（4）算法的高效性包括时间高效性和空间高效性，时间高效性是指算法执行效率高，可用时间复杂度衡量；空间高效性是指算法所占存储空间节省，可用空间复杂度衡量。

时间复杂度和空间复杂度是衡量算法优劣的两个主要指标。

1.3.2 算法的时间复杂度分析

1. 基础知识

算法的时间效率度量涉及两个相关术语，即问题规模和语句频度。

问题规模是指算法求解问题输入量的多少，是问题大小的本质表示，一般用整数 n 表示。当然，输入量也可能不仅仅与一个整数相关，这种情况下可用多个整数（如 n、m 等）表示问题规模。

语句频度是指一条语句的重复执行次数。一个算法的执行时间可用该算法中所有语句的频度之和来度量。

简单起见，可用基本语句频度衡量一个算法的时间复杂度。基本语句是指语句频度最大的语句。若计算基本语句的频度得到问题规模 n 的函数 $f(n)$，则算法的（渐近）时间复杂度 $T(n)$ 如式 (1-1) 所示。

$$T(n)=\mathrm{O}(f(n)) \tag{1-1}$$

式 (1-1) 表示随着问题规模 n 的增长，算法执行时间的增长率和函数 $f(n)$ 的增长率相同。

因此，分析一个算法的时间复杂度时，可先找出其中的基本语句，再取其频度函数 $f(n)$ 的数量级（仅需最高次幂项并忽略其系数）用符号 O 表示即可。

常见的时间复杂度依数量级递增（时间效率递减）有常数阶 $\mathrm{O}(1)$、对数阶 $\mathrm{O}(\log_2 n)$、线性阶 $\mathrm{O}(n)$、线性对数阶 $\mathrm{O}(n\log_2 n)$、平方阶 $\mathrm{O}(n^2)$、立方阶 $\mathrm{O}(n^3)$、k 次方阶 $\mathrm{O}(n^k)$、指数阶 $\mathrm{O}(2^n)$ 等。其中，$\mathrm{O}(1)$ 表示与问题规模无关。

2. 时间复杂度分析

例 1.3.1 分析两个 n 阶方阵相乘算法的时间复杂度

两个 n 阶方阵相乘的算法 mult 如下，请分析其时间复杂度。

```
//以二维数组存储矩阵元素，c 为 a 和 b 的乘积，N、n 分别方阵的最大大小和实际大小
void mult(int a[ ][N], int b[][N], int c[ ][N], int n ) {
    for (int i=1; i<=n; i++){
        for (int j=1; j<=n; j++) {
            c[i][j]=0;
            for (int k=1; k<=n; k++) {
                c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
}
```

解析：

找出基本语句为“c[i][j] += a[i][k]*b[k][j];”，计算其频度为 n^3，因此算法 mult 的时间复杂度为 $O(n^3)$。

例 1.3.2 分析选择排序的时间复杂度

选择排序算法 select_sort 如下，请分析其时间复杂度。

```
//n 个数据存放在 a 数组中，按升序排序
void select_sort(int a[ ], int n) {
    for (int i=0; i<n-1; ++i) {
        j=i;
        for (int k=i+1; k<n; ++k) {
            if (a[k]<a[j]) j = k;
        }
        swap(a[j], a[i]);
    }
}
```

解析：

找出基本语句为“if (a[k] < a[j]) j = k;”，其频度计算如式 (1-2) 所示。

$$f(n)=\sum_{i=0}^{n-2}(n-1-i)=\frac{1}{2}n^2-\frac{1}{2}n \tag{1-2}$$

取其最高次幂并忽略系数得到 n^2，因此算法 select_sort 的时间复杂度为 $O(n^2)$。

例 1.3.3 分析以下代码段的时间复杂度

有代码段如下，请分析其时间复杂度。

```
int i=1;
while (i<=n) {
    i*=2;
}
```

解析：

找出基本语句为“i*=2;”，先列出循环次数与 i 之间的对应关系，如图 1-6 所示。

循环次数	1	2	3	…	k
i 值	2	4	8	…	2^k

图 1-6　循环次数与 i 之间的对应关系

再令 $2^k=n$，有 $k=\log_2 n$，即基本语句的频度为 $\log_2 n$，因此该代码段的时间复杂度为 $O(\log_2 n)$。

1.3.3 算法的空间复杂度分析

1. 基础知识

算法的（渐近）空间复杂度 $S(n)$ 如式 (1-3) 所示。

$$S(n)=O(g(n)) \tag{1-3}$$

式 (1-3) 表示随着问题规模 n 的增大，算法运行所需存储量的增长率与某个函数 $g(n)$ 的增长率相同。

算法的存储量包括输入数据所占空间、程序本身所占空间和辅助存储空间，因前两者相对固定，算法的空间复杂度分析通常仅需考虑辅助存储空间即可。

若算法所需的辅助存储空间相对于输入数据量来说是常数，则称该算法为原地工作，相应的空间复杂度表示为 O(1)。

常见空间复杂度有 O(1)、$O(\log_2 n)$、$O(n)$ 等。

2. 空间复杂度分析

例 1.3.4 分析逆置数组元素的空间复杂度

有代码如下，分析算法 reverse1 和 reverse2 的空间复杂度。

```
// 算法 reverse1
void reverse1(int a[ ], int n) {
    int *b=new int [n];
    for (int i=n-1; i>=0; i--) {
        b[n-1-i]=a[i];
    }
    for (int i=0; i<n; i++) {
        a[i]=b[i];
    }
}
// 算法 reverse2
void reverse2(int a[ ], int n) {
    for (int i=0;  i<n/2; i++) {
        int t=a[i];
        a[i]=a[n-1-i];
        a[n-1-i]=t;
     }
}
```

解析：

算法 reverse1 使用一个长度为 n 的辅助数组 b，因此空间复杂度为 $O(n)$。

算法 reverse2 仅使用一个辅助变量 t，因此空间复杂度为 O(1)。

1.4 在线题目求解

例 1.4.1 最大子列和问题 ①

给定 K 个整数组成的序列 $\{N_1, N_2, \cdots, N_K\}$，“连续子列”被定义为 $\{N_i, N_{i+1}, \cdots, N_j\}$，其中 $1 \leqslant i \leqslant j \leqslant K$。“最大子列和”则被定义为所有连续子列元素的和中最大者。例如，给定序列 {–2, 11, –4, 13, –5, –2}，其连续子列 { 11, –4, 13 } 有最大的和 20。现要求编写程序，计算给定整数序列的最大子列和。

输入：

输入第 1 行给出正整数 K ($\leqslant$ 100 000)；第 2 行给出 K 个整数，其间以空格分隔。

输出：

在一行中输出最大子列和。如果序列中所有整数皆为负数，则输出 0。

输入样例：

```
6
-2 11 -4 13 -5 -2
```

输出样例：

```
20
```

解析：

最大子列和也称为最大子段和或最大子序列和，暴力求解方法是穷举所有以 i 为起始位置，以 j 为终止位置的子序列，并求得其中的最大和值。

算法 maxSum1 采用三重循环求最大子段和，以 O(n) 的时间复杂度求一个子序列和，总的时间复杂度为 O(n^3)。具体代码如下：

```
int maxSum1(int a[], int n) {                    // 三重循环，时间复杂度为立方阶
    int max=0;                                   // 最大和
    for(int i=0; i<n; i++) {
        for(int j=i; j<n; j++) {
            int sum=0;                           // a[i]~a[j] 的和
            for(int k=i; k<=j; k++) {            // O(n) 时间求一个子序列和
                sum+=a[k];
            }
            if (sum>max) {
                max=sum;
            }
        }
    }
    return max;
}
```

若采用算法 maxSum1 求解本例，则在 PTA 提交代码将得到超时反馈。因此，需考虑改进算法，

① 资料来源：PTA (https://pintia.cn/)，作者：浙江大学 DS 课程组。

提高时间效率。

考虑到若已经求得下标范围为 $[i, j-1]$ 的和为 *sum*，则下标范围为 $[i, j]$ 的和等于 *sum* 加上下标为 *j* 的元素值，则可以 O(1) 的时间复杂度求一个子序列和，如此可用二重循环求最大子段和，如算法 maxSum2 所示，总的时间复杂度为 $O(n^2)$。具体代码如下：

```
int maxSum2(int a[], int n) {                    // 二重循环，时间复杂度为平方阶
    int max=0;                                   // 最大和
    for(int i=0; i<n; i++) {
        int sum=0;                               // a[i]~a[j]的和
        for(int j=i; j<n; j++) {
            sum+=a[j];                           // O(1)时间求一个子序列和
            if (sum>max) {
                max=sum;
            }
        }
    }
    return max;
}
```

实际上，最大子段和的时间复杂度可降为 $O(n)$，算法如 maxSum3 所示。可以这样考虑：若一个子序列的和为负，则该子序列不可能是最大连续子序列的一部分，因为可以通过不包含它来得到一个更大和的连续子序列，所以可在一重循环中求子段和，一旦当前和为负值，则令当前和为 0；否则比较当前和与当前最大和，使当前最大和为两者中的大者。具体代码如下：

```
int maxSum3(int a[], int n) {                    // 一重循环，时间复杂度为线性阶
    int sum=0, max=0;                            // sum是当前和，max是当前最大和
    for(int i=0; i<n; i++) {
        sum+=a[i];
        if (sum<0) {
            sum=0;
        }
        else if (sum>max) {                      // 当前和大于当前最大和
            max=sum;
        }
    }
    return max;
}
```

可以调用以上算法尝试求解本例。

```
#include<bits/stdc++.h>
using namespace std;

int maxSum1(int a[], int n);          // 立方阶的算法
int maxSum2(int a[], int n);          // 平方阶的算法
int maxSum3(int a[], int n);          // 线性阶的算法
void run() {
    int n;
```

```
    cin>>n;
    int *a=new int [n];                    // 申请动态数组
    for(int i=0; i<n; i++) {
        cin>>a[i];
    }
    int result=maxSum1(a, n);              // 调用 maxSum1 进行测试
    //int result=maxSum2(a, n);            // 调用 maxSum2 进行测试
    //int result=maxSum3(a, n);            // 调用 maxSum3 进行测试
    cout << result << endl;
    delete [] a;
}

int main() {
    run();
    return 0;
}
```

对于同一个题目，不同求解方法的效率可能差别很大。如何有效提高算法的时间效率和空间效率是学习数据结构需要重点关注的问题。

习题

一、选择题

1. 数据结构的三要素指的是（　　）。

A. 逻辑结构、存储结构及数据的运算　　B. 线性结构、树结构及图结构

C. 数据、数据元素及数据项　　D. 数据元素、数据关系和数据操作

2. 以下说法正确的是（　　）。

A. 数据元素是数据的最小单位　　B. 数据项是数据的基本单位

C. 数据结构是带有结构的数据元素的集合　　D. 数据对象是同类型数据项的集合

3. 以下数据结构中，（　　）是非线性结构。

A. 树　　B. 字符串　　C. 队列　　D. 栈

4. 以下不属于算法的特性的是（　　）。

A. 有 0 个或多个输入　　B. 至少有 1 个输出　　C. 正确性　　D. 有穷性

5. 下述程序段的时间复杂度为（　　）。

```
i=1;
while(i<=n){
    i=i*2;
}
```

A. $O(\log_2 n)$　　B. $O(n)$　　C. $O(\sqrt{n})$　　D. $O(n^2)$

6. 简单选择排序算法的时间复杂度是（　　）。

A. $O(n^3)$　　B. $O(n^2)$　　C. $O(n)$　　D. $O(1)$

7. 一维数组的逆置算法中，可以达到的最好空间复杂度是（　　）。

A. $O(1)$　　B. $O(\log_2 n)$　　C. $O(n)$　　D. $O(n^2)$

8. 以下空间复杂度中最好的是（　　）。

A. O(n^2)　　B. O(n)　　C. O($\sqrt{n}$)　　D. O(1)

9. 下述程序段的时间复杂度为（　　）。

```
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        a[i][j]=0;
```

A. O(n)　　B. O(mn)　　C. O(m^2)　　D. O(n^2)

10. 下述程序段的时间复杂度为（　　）。

```
int m=100, n=200;
while(n>0) {
    if(m>100) m-=10,n--;
    else m++;
}
```

A. O(mn)　　B. O(n)　　C. O(m)　　D. O(1)

11. 某算法按顺序执行程序段 1 和程序段 2，假设程序段 1 的执行次数为 $3n^2$，程序段 2 的执行次数为 $0.02n^3$，则该算法的时间复杂度为（　　）。

A. O(n)　　B. O(n^2)　　C. O(n^3)　　D. O(1)

12. 下述程序段的时间复杂度为（　　）。

```
a=n;
b=0;
while(a>=b*b) b++;
```

A. O($\log_2 n$)　　B. O(n)　　C. O($\sqrt{n}$)　　D. O(n^2)

13. 算法优劣分析的两个主要指标是（　　）。

A. 空间复杂度和时间复杂度　　B. 正确性和简单性

C. 可读性和可维护性　　D. 数据复杂性和程序复杂性

14. 执行下面程序段时，执行 S 语句的次数为（　　）。

```
for(int i=1; i<=n; i++)
    for(int j=1;j<=n; j++)
        S;
```

A. n^2　　B. $n^2/2$　　C. $n(n+1)$　　D. $n(n+1)/2$

15. 下述程序段的时间复杂度为（　　）。

```
for (i=0; i<n-1; i++)
    for (j=0; j<n-1-i; j++)
        t=a[j], a[j]=a[j+1], a[j+1]=t;
```

A. O(1)　　B. O(n)　　C. O(n^2)　　D. O(n^3)

16. 第 15 小题中的程序段的空间复杂度为（　　）。

A. O(1)　　B. O(n)　　C. O(n^2)　　D. O(n^3)

二、编程题

寻找第 k 小的数（HLOJ 9500）

给定若干整数，请设计一个高效的算法，确定第 k 小的数。

输入：

测试数据有多组，处理到文件尾。每组测试数据的第 1 行输入两个整数 n，k（$1 \leqslant k \leqslant n \leqslant 1\,000\,000$）。第 2 行输入 n 个整数，每个数据的取值范围为 0~1 000 000。

输出：

对于每组测试，输出第 k 小的数。

输入样例：

```
9 3
1 2 3 4 5 6 9 8 7
```

输出样例：

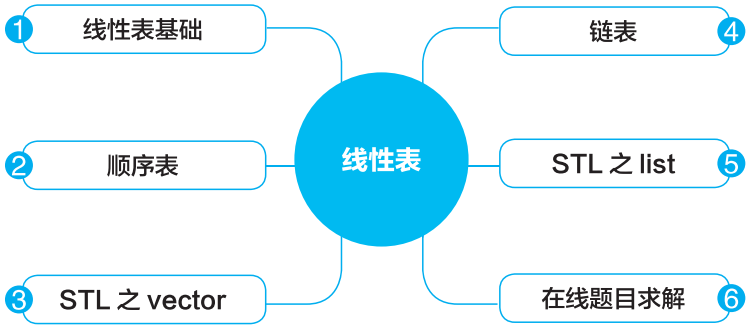
```
3
```

第 2 章 线性表

学习目标

1. 记忆和理解线性表、顺序表和链表的基础知识。
2. 掌握顺序表、链表的实现。
3. 学会运用顺序表、链表求解具体问题。
4. 学会运用 STL 之 vector、list 求解具体问题。
5. 树立攻坚克难、勇于挑战的信念。
6. 获取脚踏实地、精诚合作的精神动力。

知识导图



导言

线性表是最简单、最基本的一种线性数据结构。线性表可以采用顺序存储和链式存储，这两种存储方式同样适用于其他线性结构和非线性结构。因此，对应顺序存储的顺序表和对应链式存储的链表是学习其他数据结构及相关算法的重要基础，需熟练掌握。可以说，链表没学好是很多人学不好数据结构课程的主要原因。因此，熟练掌握链表至关重要。本章介绍线性表基础知识、顺序表、链表等方面的内容，重点关注顺序表和链表的实现、顺序表及链表的运用。另外，本章还介绍了 C++ 标准模板库（standard template library，STL）中的向量 vector，双向链表 list。最后，介绍运用顺序表和链表是如何求解具体问题的。

思政链接：佩利

2.1 线性表基础

线性表 $(a_1, a_2, \cdots, a_i, \cdots, a_n)$ 是一种最简单的线性结构，是 n 个特性相同的数据元素构成的有限序列；其中，n 称为线性表的表长；n=0 的线性表称为空表；i 称为数据元素 a_i 的位序。

线性表 $(a_1, a_2, \cdots, a_i, \cdots, a_n)$ 具有以下基本特征：

（1）存在唯一的一个“首元素”（第一个元素），即 a_1。

（2）存在唯一的一个“尾元素”（最后一个元素），即 a_n。

（3）除首元素之外，其他元素均仅有一个前驱，如 a_2 的前驱为 a_1，a_n 的前驱为 a_{n-1}。

（4）除尾元素之外，其他元素均仅有一个后继，如 a_1 的后继为 a_2，a_{n-1} 的后继为 a_n。

例如，6 个英文字母可构成线性表 (A, B, C, D, E, F)，5 个整数可构成线性表 $(1, 2, 3, 4, 5)$。

线性表具有创建、遍历、插入、删除、修改和查询等基本操作。线性表既可采用顺序存储结构，又可采用链式存储结构，相应的线性表分别称为顺序表和链表。

2.2 顺序表

2.2.1 顺序表基础

顺序表是采用顺序存储结构的线性表，是用一组地址连续的存储单元依次存放线性表中的数据元素。

例如，线性表 $(a_1, a_2, \cdots, a_i, \cdots, a_n)$ 的顺序存储示意图如图 2-1 所示。

a_1	a_2	...	a_{i-1}	a_i	a_{i+1}	...	a_{n-1}	a_n

图 2-1　线性表的顺序存储示意图

在顺序表中，以“存储位置相邻”表示有序对 $<a_i-1, a_i>$，则 a_i 的存储地址 LOC(a_i) 按式 (2-1) 计算。

$$\mathrm{LOC}(a_i) = \mathrm{LOC}(a_{i-1}) + C \tag{2-1}$$

其中，C 表示一个数据元素所占的存储量。首元素 a_1 的地址 LOC(a_1) 称为线性表的起始地址或基地址。实际上，线性表中的所有其他数据元素的存储位置均取决于首元素 a_1 的存储位置，具体计算如式 (2-2) 所示。

$$\mathrm{LOC}(a_i) = \mathrm{LOC}(a_1) + (i-1) \times C \tag{2-2}$$

2.2.2 顺序表的定义与实现

这里把顺序表类型定义为一个结构体，该结构体中包含数据成员（属性）和成员函数（方法），具体如下：

```
const int N=80;                          // 顺序表的最大容量为 N
typedef int ElemType;                    // 为 int 类型取别名 ElemType
struct SqList {
    ElemType *elem;                      // 存储空间基址
    int length;                          // 当前长度
    void Init( );                        // 初始化
    void Clear();                        // 清除线性表
    int Locate(ElemType e);              // 查找
    void InsertBefore(int i, ElemType e);// 插入元素
    void Delete(int i, ElemType &e);     // 删除元素
    bool Empty();                        // 判断空表
    void Traverse();                     // 遍历
};
```

结构体类型 SqList 中的数据成员 *elem* 是一个 ElemType 类型的指针，用于存放动态数组的首地址；数据成员 *length* 表示当前顺序表中的元素个数（长度）。结构体类型 SqList 中的一个成员函数对应顺序表的一个基本操作。后文中也经常使用类似的表达，即一个基本操作用一个成员函数描述。可以通过把数据成员和成员函数构成一个整体来实现封装性，对于调用基本操作完成相应功能的用户而言，只需知道基本操作名和参数即可，而不必关心基本操作的具体实现。

用 typedef 把已有类型 int 取别名为 ElemType，从而提升代码的可维护性。例如，当顺序表的数据元素类型由 int 改为 char 类型时，仅需把该语句中的 int 改为 char 即可，而基本操作无需修改。

下面讨论顺序表基本操作的具体实现。

1. 初始化

顺序表的初始化操作申请一个动态数组并用数据成员 *elem* 指向，并置数据成员 *length* 为 0，具体代码如下：

```
// 初始化顺序表
void SqList::Init ( ) {                  // 构造空的顺序表
    elem = new ElemType [N];             // 创建动态数组，首地址存放在 elem 中
    length = 0;
}
```

Init 算法用 new 运算符创建的长度为 *N* 的动态数组，把该数组的首地址存放在指针变量 *elem*（可视为数组名）中，并置表长为 0（表示空表）。显然，此算法的时间复杂度为 O(1)。

:: 是类属运算符，表明其后的函数（此处为 Init）属于其前的结构体类型（此处为 SqList）。

2. 查找

在顺序表中查找元素 *e* 时，从头开始依次把 *e* 与顺序表中元素进行比较，若相等则返回其位序，否则继续比较下一个元素；若最终都没有与 *e* 相等的元素，则返回 0。顺序表的查找算法 Locate 具体实现如下。

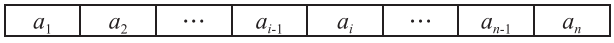
```
// 在顺序表中查找第一个满足判定条件的数据元素，若存在，则返回它的位序，否则返回 0
int SqList:: Locate (ElemType e) {                    // 查找
    int i = 1;                                        // i 的初值为第一个元素的位序（1）
    while (i <= length && elem[i-1] != e){
        i++;
    }
    if (i <= length)  return i;
    else  return 0;
}
```

设等于 e 的元素在最后一个位置，则需要循环 n 次才能找到，因此 Locate 算法的最坏时间复杂度为 O(n)。考虑 e 在各个位置出现的概率相同，则相应循环次数分别为 1,2,3,…，n，平均循环次数等于 $\frac{1}{n}\times\frac{n(n+1)}{2}=\frac{n+1}{2}$，因此 Locate 算法的平均时间复杂度也为 O(n)。

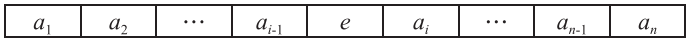
3. 插入

顺序表操作 InsertBefore (i, e) 实现在顺序表的第 i 个元素之前插入元素 e。

在第 i 个元素之前插入元素 e，将使线性表从 (a_1, …, a_{i-1}, a_i, …, a_n) 改变为 (a_1, …, a_{i-1}, e, a_i, …, a_n)，即从一个有序对 <a_{i-1}, a_i> 变为两个有序对 <a_{i-1}, e>、<e, a_i>，且表长增加了 1。插入前后的示意图分别如图 2-2(a)(b) 所示。



(a)



(b)

图 2-2　顺序表的插入操作示意图

(a) 插入前；(b) 插入后

为了在第 i 个元素之前插入元素 e，需把第 i 个位置空出来，即把 a_n，a_{n-1}，…，a_i 依次后移一个位置，再把 e 存放到第 i 个位置，最后表长 *length* 增加了 1。顺序表的插入算法 InsertBefore 具体实现如下：

```
// 在顺序表的第 i 个元素（下标为 i-1）之前插入元素 e
void SqList:: InsertBefore( int i, ElemType e) {
    if (i < 1 || i > length+1) return ;               // 若插入位置非法，则不做插入操作
    for (int j = length-1; j >= i-1;  j--)
        elem[j+1]= elem[j];                           // 插入位置及之后的元素后挪
    elem[i-1]= e;                                     // 插入元素 e
    length ++;                                        // 表长增 1
}
```

考虑移动元素的平均情况，假设在第 i 个元素之前插入的概率为 p_i，则在长度为 n 的线性表中插入一个元素所需移动元素次数的期望值（平均次数）如式 (2-3) 所示。

$$E_{is}=\sum_{i=1}^{n+1}p_i(n-i+1) \tag{2-3}$$

若假定在线性表中任何一个位置 i（$1 \leqslant i \leqslant n+1$）上进行插入的概率都是相等的，则移动元素的期望值如式 (2-4) 所示。

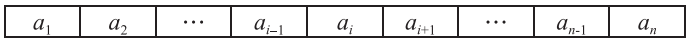
$$E_{is}=\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)=\frac{n}{2} \tag{2-4}$$

因此，InsertBefore 算法的平均时间复杂度为 O(n)。InsertBefore 算法的最坏情况是插入元素到第一个元素之前，时间复杂度也为 O(n)。

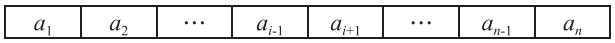
4. 删除

顺序表操作 Delete (i, &e) 实现删除顺序表中的第 i 个元素，并通过引用参数 e 返回该元素。

设删除元素 a_i，则线性表从 (a_1, …, a_{i-1}, a_i, a_{i+1}, …, a_n) 改变为 (a_1, …, a_{i-1}, a_{i+1}, …, a_n)，即从两个有序对 <a_{i-1}, a_i>, <a_i, a_{i+1}> 变为一个有序对 <a_{i-1}, a_{i+1}>，且表长 *length* 减 1。顺序表删除前后的示意图如图 2-3(a)(b) 所示。



(a)



(b)

图 2-3　顺序表的删除前后的示意图

(a) 删除前；(b) 删除后

可见，若要删除元素 a_i，则可通过把其后的元素 a_{i+1}, a_{i+2}, …, a_{n-1}, a_n 逐个前移覆盖原有元素。删除算法 Delete 具体实现如下：

```
// 顺序表中输出第 i 个元素，通过引用参数 e 返回该元素
void SqList:: Delete (int i, ElemType &e) {
    if (i < 1|| i > length)  return;          // 待删位置非法，则返回
    e = elem[i-1];                            // 被删除元素的值赋给 e
    // 被删除元素之后的元素左移
    for (int j=i; j <= length-1; j++)  elem[j-1]=elem[j];
    length--;                                 // 表长减 1
}
```

待删元素通过引用参数 e 返回，若删除之后无须再用到该元素，则可省略引用参数 e。

考虑移动元素的平均情况，假设删除第 i 个元素的概率为 q_i，则在长度为 n 的线性表中删除一个元素所需移动元素次数的期望值（平均次数）如式 (2-5) 所示。

$$E_{dl}=\sum_{i=1}^{n}q_i(n-i) \tag{2-5}$$

若假定在线性表中任何一个位置上进行删除的概率都是相等的，则移动元素的期望值如式 (2-6) 所示。

$$E_{dl}=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2} \tag{2-6}$$

因此，Delete 算法的平均时间复杂度为 O(n)。Delete 算法的最坏情况是删除第一个元素，时间复杂度也为 O(n)。

5. 其他操作的算法实现

顺序表的其他操作比较简单，下面直接给出它们的算法实现：

```
// 清空顺序表，时间复杂度为 O(1)
void SqList::Clear() {
    delete [] elem;
    length=0;
}
// 判断是否空顺序表，时间复杂度为 O(1)
bool SqList::Empty() {
    return length==0;
}
// 遍历顺序表，时间复杂度为 O(n)
void SqList::Traverse() {
    for (int i=0; i < length; i++) {
        if (i>0) cout<<" ";
        cout << elem[i];
    }
    cout<<endl;
}
```

6. 顺序表基本算法的调用

顺序表基本操作的算法实现之后，就可以调用这些算法完成相应功能。下面给出简单的调用示例：

```
#include <iostream>
using namespace std;
int main(){
    SqList L;                                   // 定义顺序表
    L.Init();                                   // 初始化顺序表
    int n;
    cin>>n;
    for(int i=0;i<n;i++){                       // 尾插创建顺序表，时间复杂度为平方阶
        ElemType t;
        cin>>t;
        L.InsertBefore(i+1, t);        // 在顺序表中插入元素
    }
    L.Traverse();                               // 遍历顺序表
    cout<<L.Empty()<<endl;                      // 顺序表判空
    ElemType m;
    cin>>m;
    cout<<L.Locate(m)<<endl;                    // 查找顺序表
    L.Delete(3, m);                             // 删除顺序表中的第 3 个元素
    L.Traverse();                               // 遍历顺序表
    L.Clear();                                  // 清空顺序表
    return 0;
}
```

2.3 STL 之 vector

1. 基础知识

STL 之 vector（向量）可用于表示顺序表，头文件是 vector。定义向量时，可同时指定其长度。例如：

```
int n;
cin>>n;
vetcor <int> a(n);
```

语句“ vetcor <int> a(n); ”是 vector 的实例化，即在 < > 中指定 *a* 的每个元素的类型都是 int，*a* 后面小括号中的 *n* 表明该向量的长度为 *n*。

vetcor 部分常用成员函数如表 2-1 所示，其中，参数 val 的类型同向量中的元素，可以是 int、char、double 等基本数据类型、string 类型及结构体类型等，pos 是迭代器（iterator）类型（类似指针类型）参数，*n* 是整型参数。设一维向量名为 *a*，其中的元素为 1、3、5、7。

表 2-1　vector 部分常用成员函数

成员函数	含义与示例
begin()	指向首元素的迭代器。例如，a.begin() 指向元素 1
end()	指向尾元素后一位置的迭代器。例如，a.end() 指向元素 7 之后的位置
clear()	清空向量。例如，a.clear() 清空向量 a
empty()	向量判空，若为空，则返回 true，否则返回 false。例如，a.clear() 之后 a.empty() 为 true
erase(pos)	删除迭代器 pos 所指元素并返回下一元素的迭代器。例如，a.earse(a.begin()) 将删除 a 的首元素 1 并返回指向元素 3 的迭代器
insert(pos, val)	在迭代器 pos 所指位置插入一个值为 val 的元素并返回其迭代器 例如，a.insert(a.begin(), 9) 将把 9 插入 a 的头部并返回指向该元素的迭代器
pop_back()	删除向量的尾元素。例如，a.pop_back() 将删除元素 7
push_back(val)	将 val 插入向量的最后位置。例如，a.push_back(9) 将把元素 9 插入 a 的最后位置
size()	vector 的大小（一维 vector 的元素个数或二维 vector 的行数、列数）。例如，a.size() 为 4
resize(n, val)	重置 vector 大小为 *n*，若 *n* 小于向量原来的大小（设为 *m*），则保留前面 *n* 个元素及其值，否则前 *m* 个元素保留原值，后 *n*–*m* 个元素置值为 val（该参数默认的参数值为 0）。例如，a.resize(10) 将 a 的长度重定义为 10，且前 4 个元素保留原值为 1、3、5、7，后 6 个元素为 0

2. 一维向量

例 2.3.1 一维 vector 使用方法

具体代码如下：

```
#include <iostream>
#include <vector>
using namespace std;
void prtVector(vector<int> a);
int main() {
```

```
    int n,i;
    cin>>n;
    vector <int> v(n);                          // 长度为 n 的一维向量，实现动态数组
    for(i=0; i<v.size(); i++) v[i]=i+1;
    v.push_back(123);                           // 在尾部插入
    v.insert(v.begin(), 456);                   // 在头部插入
    v.erase(v.begin()+1);                       // 删除第二个元素
    prtVector(v);                               // 输出向量元素
    v.clear();                                  // 清空向量
    v.resize(n*2);                              // 重新定义长度
    cout<<v.size()<<endl;
    return 0;
}
// 输出向量中所有元素，每两个数据之间间隔一个空格
void prtVector(vector<int> a) {
    for(int i=0; i<a.size(); i++) {
        if (i>0) cout<<" ";
        cout<<a[i];
    }
    cout<<endl;
}
```

读者可自行编程测试 vector 常用成员函数的使用。

3. 二维向量

例 2.3.2 二维 vector 简单示例

具体代码如下：

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int i;
    int r,c;
    cin>>r>>c;
    // 下面的语句直接定义 r 行 c 列的二维动态数组
    vector < vector<int> > tv(r, vector<int> (c));
    for(i=0; i<r; i++) {
        for(int j=0; j<c; j++)
            cin>>tv[i][j];
    }
    for(i=0; i<r; i++) {
        for(int j=0; j<c; j++) {
            if (j>0) cout << " ";
            cout << tv[i][j];
        }
        cout << endl;
    }
    tv.clear();
    return 0;
}
```

语句“vector < vector<int> > tv (r, vector<int> (c));”定义一个 *r* 行 *c* 列的二维向量，注意“>>”中两个 > 之间有空格，否则可能会被编译器理解为运算符“>>”而出错；定义二维向量还可以采用如下写法：

```
vector <vector <int> > tv;                    // 定义二维向量
tv.resize(r);                                 // 重置第一维长度（行数）为 r
for(i=0; i<r; i++) tv[i].resize(c);           // 重置第二维长度（列数）为 c
```

实际上，二维 vector 的每行的列数可以不一样，这样使用起来更加灵活，例如：

```
for(i=0; i<r; i++) tv[i].resize(i+1);         // 定义了一个下三角矩阵
```

因此，可用第二维长度不一的二维向量模拟实现图的邻接表存储结构。

可以用 *tv*.size() 得到 *tv* 的行数，用 *tv*[*i*].size() 得到第 *i* 行的列数，例如，以下代码输出各行的列数：

```
for(i=0; i<r; i++) cout<<tv[i].size()<<endl;
```

2.4 链表

2.4.1 链表概述

链表用一组地址任意的存储单元存放线性表中的数据元素。链表可认为是以“节点的序列”表示的线性表。链表中的一个节点包含数据域和指针域两部分，其中，数据域存放数据元素，而指针域存放其他节点的地址。

设节点包含一个整型数据域 *data* 和一个指针域 *next*，则节点的结构体类型声明如下：

```
struct LNode {                                // 节点结构体类型
    int data;                                 // 数据域
    LNode *next;                              // 指针域，用于存放下一个节点的地址
};
```

其中，数据成员 *next* 是一个 LNode* 类型的指针，即用来存放下一个节点（LNode 类型变量）的地址。

这里主要介绍带头节点（该节点的数据域不存放有效数据）的单链表。例如，共有 4 个数据节点（数据域存放有效数据）的带头节点的单链表示意图如图 2-4 所示。

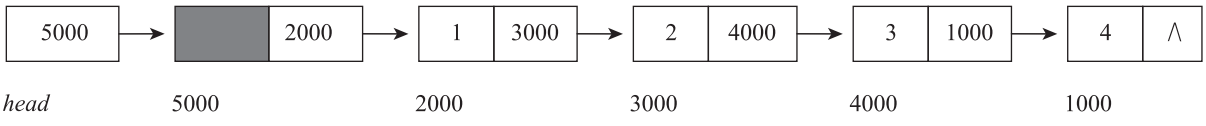


图 2-4　共有 4 个数据节点的带头节点的单链表示意图

在图 2-4 中，数据域为灰色（值为某个随机数）的节点是头节点，地址设为 5000，存放在指针变量 *head* 中，则 *head* 指向头节点，称为头指针；其余节点的地址分别设为 2000、3000、4000、1000。因地址指向该地址所在的存储单元，故后继（后一个节点）的地址存放在前驱（前一个节点）的指针域时，前驱的指针域就指向后继，从而构成一个链表。在带头节点的单链表中，第一个数据节点的前趋是头节点，最后一个数据节点没有后继，其指针域的值为空指针 NULL（链表结束标志），图中以“∧”

表示。

在单链表中，前一节点的指针域指向后一节点，只能通过前一节点才能找到后一节点。因此，单链表的访问规则是“从头开始、顺序访问”，即通过指向头节点的头指针，逐个节点依次访问。

在顺序表中插入、删除元素时，需要大量移动数据元素，时间效率较低。而在单链表中，若在某个节点之后插入或删除节点，只需简单修改节点指针域的指向而不必大量移动元素，因此插入和删除操作频繁时宜用链表结构。

思政链接：巴克斯

画示意图是学习链表的一个重要方法，读者可通过画图理解链表的基本操作。

2.4.2 单链表基本操作及其实现

节点类型 LNode 如前述，把链表的基本操作和头指针构成为一个整体，定义带头节点的链表的结构体类型 LinkList 的具体代码如下：

```
typedef int ElemType;                          // 定义 ElemType 为 int 类型的别名
struct LinkList{                               // 声明链表结构体类型
    LNode *head;                               // 头指针，指向头节点
    void Init();                               // 初始化
    void Clear();                              // 清空链表
    void Create(int n);                        // 建立含 n 个节点的单链表
    int Locate(ElemType e);                    // 查找
    void InsertBefore(int i, ElemType e);      // 插入元素
    void Delete(int i, ElemType &e);           // 删除元素
    void Traverse();                           // 遍历
    bool Empty();                              // 判断空链表
};
```

单链表基本操作的具体实现如下。

1. 初始化（创建空链表）

具体代码如下：

```
// 创建节点并用头指针指向，并置其指针域为空指针
void LinkList::Init() {
    head = new LNode;
    head->next = NULL;
}
```

Init 算法的时间复杂度为 O(1)。

2. 判断是否空链表

具体代码如下：

```
// 若头指针所指节点的指针域为空指针则返回 true，否则返回 false
bool LinkList::Empty() {
    return head->next == NULL;
}
```

Empty 算法的时间复杂度为 O(1)。

3. 插入节点

插入算法 InsertBefore(*i*, *e*) 实现的是在头指针为 *head* 的单链表的第 *i* 个节点之前插入数据域值为 *e* 的节点。首先需要从头开始找到第 *i*–1 个节点（设由 *p* 指向），然后在其后插入新节点（设由 *q* 指向）。设 *i*=3、*e*=5，则第 3 个节点插入前后的示意图如图 2-5(a)(b) 所示。

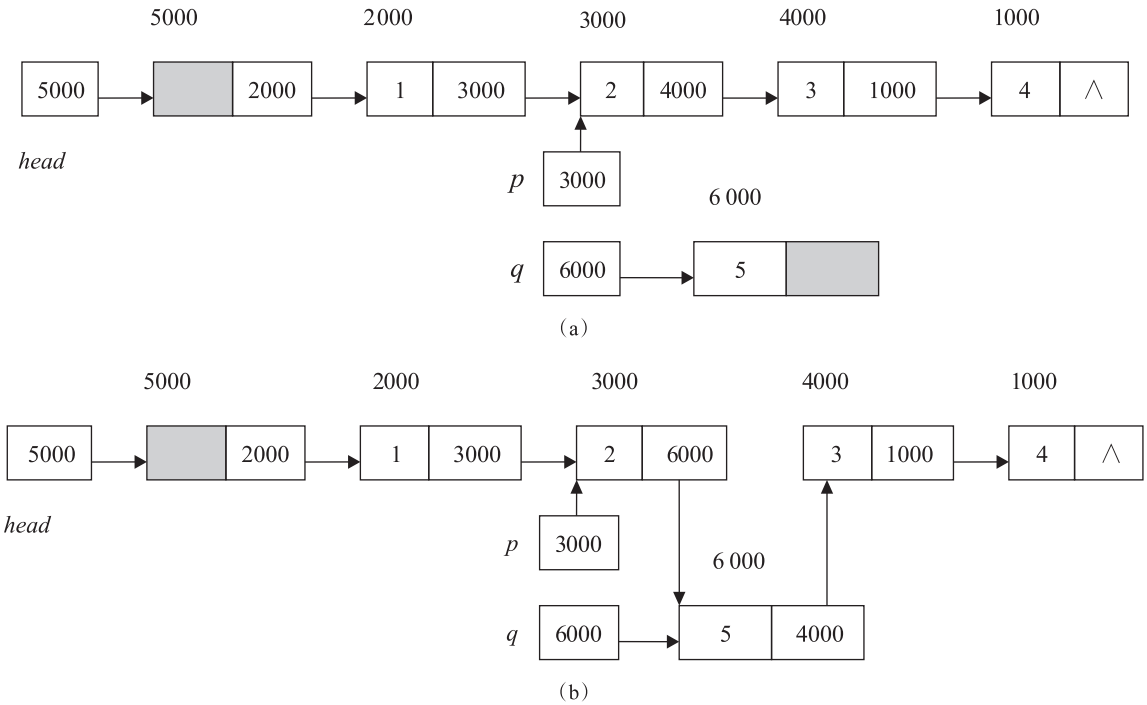


图 2-5　第 3 个节点插入前后的示意图

（a）插入前；（b）插入后

插入算法 InsertBefore 的具体代码如下：

```
//在带头节点的单链表的第 i（1<=i<=n+1）个节点之前插入数据域值为 e 的节点
void LinkList::InsertBefore(int i, ElemType e){
    LNode* p = head;
    while (p && i>1){                    // 找第 i-1 个节点，用 p 指向该节点
        p=p->next;
        i--;
    }
    if (p==NULL || i <1)  return;        // 若 i 超出范围，则插入位置非法
    LNode* q = new LNode;                // 生成新节点，用 q 指向该节点
    q->data = e;
    q->next = p->next;                   // 新节点链到第 i 个节点之前
    p->next = q;                         // 新节点链到第 i-1 个节点之后
}
```

InsertBefore 算法包含了查找第 *i*–1 个节点的过程，因此时间复杂度为 O(*n*)。若不做查找操作，则仅需语句“q->next = p->next; p->next = q;”就可完成插入操作，此时的时间复杂度为 O(1)。

4. 删除节点

删除算法 Delete (*i*, &*e*) 实现的是在头指针为 *head* 的单链表中删除第 *i* 个节点，并通过引用参数 *e* 返回其数据域值。首先需要从头开始找到该节点的前驱节点（设由 *q* 指向），则待删节点由 *p=q->next*

指向，然后只要把 *p* 所指节点的后继的地址 *p->next* 放到 *q* 所指节点的指针域 *q->next* 即可完成删除操作。设 *i* 为 3，则删除第 3 个节点前后的示意图如图 2-6(a)(b) 所示，释放所删节点的空间之后的示意图如图 2-6(c) 所示。

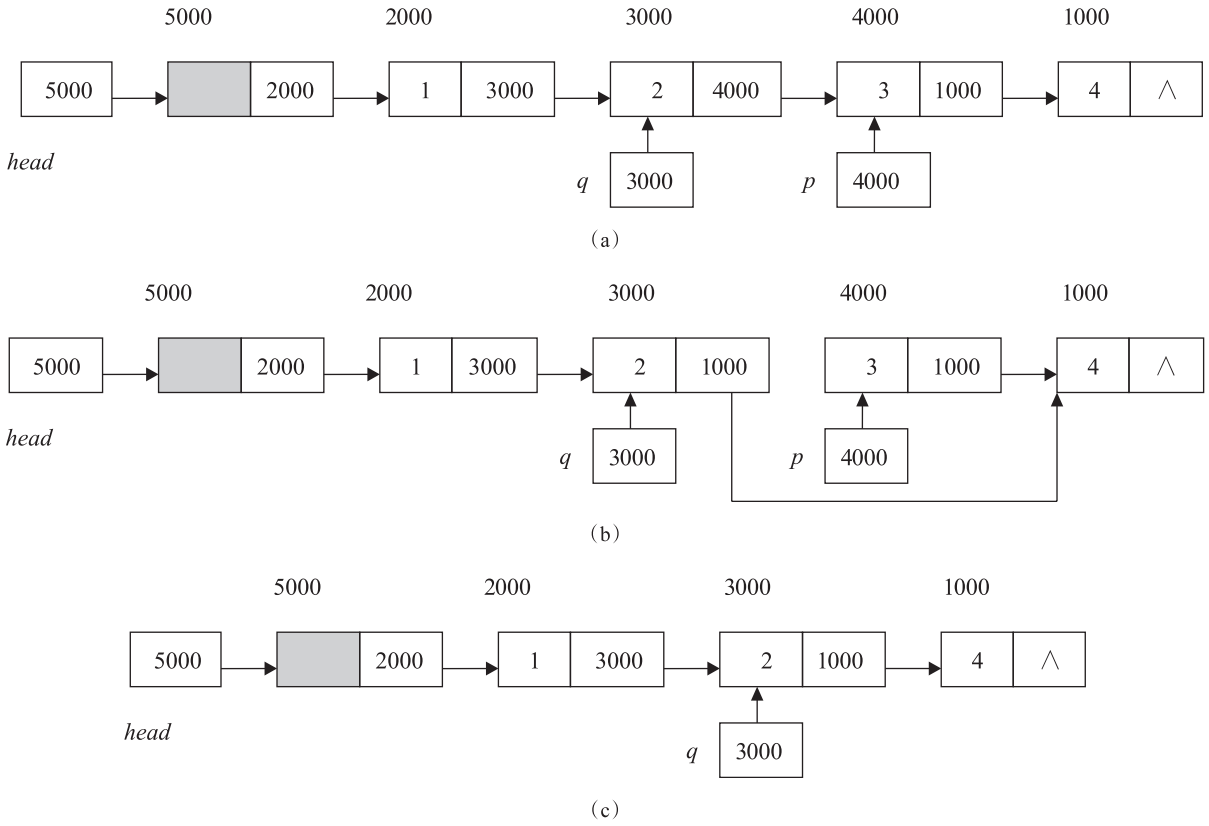


图 2-6　删除第 3 个节点前后的示意图

（a）删除前；（b）删除后；（c）释放所删节点的空间

删除算法 Delete 的具体代码如下：

```
// 删除带头节点的单链表中第 i（1<=i<=n）个节点，并用引用参数 e 返回其数据域值
void LinkList::Delete (int i, ElemType &e) {
    LNode* q = head;
    int j = 0;
    // 找到第 i-1 个节点，用 q 指向该节点，则 q->next 指向第 i 个节点
    while (q->next && j < i-1){
        q = q->next;
        j++;
    }
    if  (!(q->next) || j > i-1)  return;          // i 超出范围，删除位置不合理
    // 删除并释放节点
    LNode* p = q->next;
    q->next = p->next;
    e = p->data;
    delete p;
}
```

Delete 算法包含了查找第 *i*–1 个节点的过程，因此时间复杂度为 O(*n*)。若不做查找操作，也不考虑释

放已删节点所占空间，则仅需语句“*q->next = q->next->next;*”就可完成删除操作，此时的时间复杂度为 O(1)。

5. 清空链表

Clear 算法实现的是清空链表，可以通过逐个删除第一个数据节点的方法来完成，具体代码如下：

```
// 将单链表重新置为一个空表
void LinkList::Clear( ) {
    while (head->next) {
        LNode*p=head->next;
        head->next=p->next;
        delete p;
    }
}
```

Clear 算法的时间复杂度为 O(*n*)。

6. 查找节点

算法 Locate(*e*) 实现的是在链表中查找数据域值为 *e* 的节点，若找到，返回该节点的位序，否则返回 0。只需从头开始、顺序查找，即逐个比较待查找的 *e* 是否等于当前节点数据域的值。

查找算法 Locate 的具体代码如下：

```
// 在单链表中查找数据域值为 e 的节点，若找到，返回指向该节点的位序，否则返回 0
int LinkList::Locate(ElemType e) {
    LNode *p=head->next;
    int i=1;
    while(p) {
        if (e==p->data) break;
        i++;
        p=p->next;
    }
    if (!p) return 0;
    else return i;
}
```

Locate 算法的时间复杂度为 O(*n*)。

7. 创建链表

链表是一个动态的结构，它不需要预先分配空间，因此创建链表是一个“逐个插入”节点的过程。建立带头节点的单链表常用两种思想：

（1）尾插法：新节点插入尾节点之后，所得链表称为顺序链表。

（2）头插法：新节点插入头节点之后，第一个数据节点之前，所得链表称为逆序链表。

算法 Create(*n*) 采用头插法创建带头节点的单链表。以建立如图 2-4 所示的带头节点的单链表为例，数据输入顺序为 4、3、2、1。创建逆序链表的过程有如下三个步骤。

第一，建立一个带头节点的空链表，如图 2-7 所示，可以直接调用初始化算法 Init 实现。

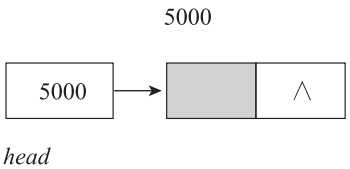


图 2-7　带头节点的空链表

第二，申请新节点由指针 *q* 指向，输入数据，并链接到头节点之后、第一个数据节点（由 *head->next* 指向，第一次为 NULL）之前，示意图如图 2-8 所示（输入数据 4）。

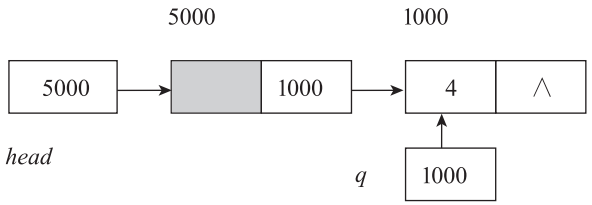


图 2-8　插入第 1 个数据构成的节点

第三，重复第二步，直到所有数据节点都插入链表中，如图 2-9 所示。

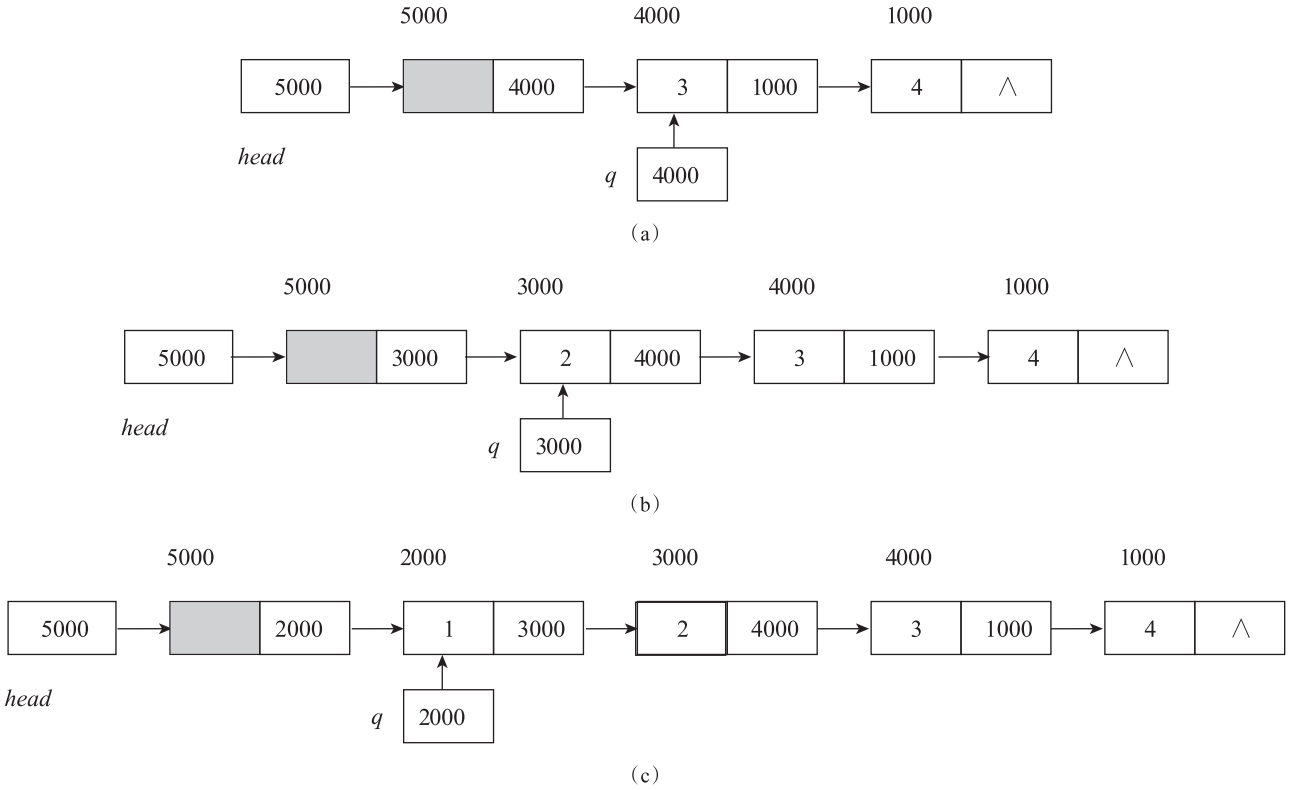


图 2-9　分别插入第 2、3、4 个数据构成的节点

（a）插入第 2 个数据构成的节点；（b）插入第 3 个数据构成的节点；（c）插入第 4 个数据构成的节点

创建逆序链表的 Create 算法具体代码如下：

```
//创建包含 n 个元素的逆序链表
void LinkList:: Create(int n) {
    Init();
    for (int i=n; i>0; i--) {
        LNode* q=new LNode;
        cin>>q->data;
        q->next=head->next;
        head->next=q;
    }
}
```

Create 算法的时间复杂度为 O(n)。

创建逆序链表也可以逐个输入数据暂存到变量 e 中，并调用 InsertBefore(1, e) 逐个插入链表的第一个数据节点之前，此时的时间复杂度也为 O(n)。

顺序链表的创建过程和代码实现留给读者自行思考和实现。

8. 遍历链表

遍历链表的 Traverse 算法根据“从头开始，顺序访问”的单链表访问规则，用一个指针变量 p 一开始指向头节点之后的节点，即第一个数据节点，在链表还未结束时不断输出 p 所指节点的数据域的值并使 p 指向下一个节点。另外，采用计数器的方法控制每两个数据之间留一个空格。

链表遍历算法 Traverse 具体代码如下：

```
//遍历单链表
void LinkList::Traverse(){
    LNode* p=head->next;
    int cnt=0;
    while (p) {
        cnt++;
        if (cnt>1) cout << " ";
        cout<<p->data;
        p=p->next;
    }
    cout << endl;
}
```

Traverse 算法的时间复杂度为 O(n)。

9. 调用链表基本操作

实现单链表的基本操作之后，可以调用这些基本操作以实现相应功能。具体代码如下：

```
#include<iostream>
using namespace std;
int main(){
    int m, n, t;
    LinkList L;                                    // 定义链表
    L.Init();                                      // 初始化链表
    cin>>n;
    //调用 InsertBefore 算法创建顺序链表，时间复杂度为平方阶
    for(int i=0;i<n;i++){
        cin>>t;
        L.InsertBefore(i+1,t);                     // 插入节点
    }
    L.Traverse();                                  // 遍历链表
    cout<<L.Locate(5)<<endl;                       // 查找节点
    L.Delete(5, m);                                // 删除节点
    L.Traverse();                                  // 遍历链表
    L.Clear();                                     // 清空链表
    return 0;
}
```

2.4.3 其他形式的链表简介

本节介绍的其他形式的链表包括循环单链表和双向链表。

1. 循环单链表

循环单链表是最后一个节点的指针域的指针又指回头节点的链表。如图 2-10 所示的是包含一个数据域和一个指针域的循环单链表。

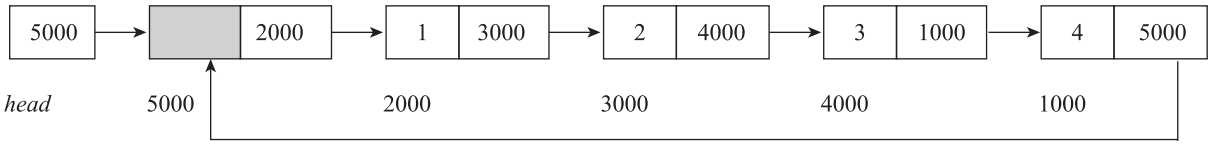


图 2-10　包含一个数据域和一个指针域的循环单链表

循环单链表和单链表的差别仅在于：判别链表中最后一个节点的条件不再是“后继是否为空”，而是“后继是否为头节点”。基本操作不再赘述，留给读者自行实现。

2. 双向链表

实际上，链表的每个节点也可以包含多个数据域或多个指针域。双向链表包含两个指针域，其中一个指向前驱，另一个指向后继。如图 2-11 所示的是包含一个数据域和两个指针域（分别指向前驱和后继）的不带头节点的双向链表。

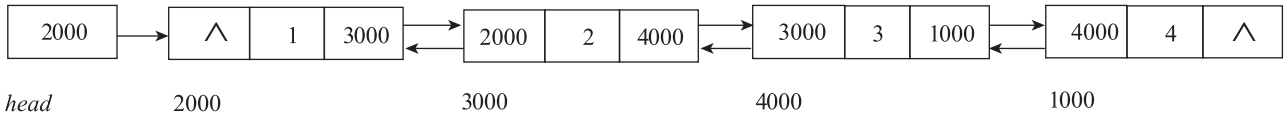


图 2-11　包含一个数据域和两个指针域的不带头节点的双向链表

设双向链表中的指向前驱的指针域为 *prior*，指向后继的指针域为 *next*，则在 *p* 所指节点之后插入一个 *q* 所指的新节点的语句序列如下：

```
q->next=p->next;
p->next->prior=q;
p->next=q;
q->prior=p;
```

删除 *p* 所指节点的后继的语句序列（不考虑释放所删节点的空间）如下：

```
p->next->next->prior=p;
p->next=p->next->next;
```

双向链表的其他操作不再赘述，读者可自行思考和实现。

2.5 STL 之 list

1. 基础知识

STL 之 list 对应双向链表，相当于每个节点既有后继指针，又有前驱指针，则 list 可以从前往后访问，也可以从后往前访问。这里仅讨论从前往后访问 list。

使用 list 首先需要包含头文件 list。定义数据元素类型为整型 list 的代码如下：

```
list <int> myList;                                    // 用 int 类型实例化 list
```

list 的部分常用成员函数如表 2-2 所示。

表 2-2　list 的部分常用成员函数

成员函数	含义
begin()	指向首元素的迭代器
end()	指向尾元素后一个位置的迭代器
clear()	清空链表
empty()	链表判空
size()	链表长度
push_back(val)	将 val 链接到链表的表尾
push_front(val)	将 val 链接到链表的表头
reverse()	逆置链表
merge(lst[,cmp])	归并有序链表，将有序链表 lst 有序地归并到原链表（需有序）中。默认按“小于”比较，可通过指定第二个参数 cmp（比较函数）表明比较规则
sort([cmp])	链表排序，默认按非递减序排序，可通过指定参数 cmp（比较函数）表明比较规则
unique()	去除链表中的重复元素，链表需先排序
erase(pos)	删除迭代器 pos 所指元素，返回 pos 所指元素的后继的迭代器
erase(start, end)	删除迭代器区间 [start，end) 之间的元素
insert(pos,val)	在迭代器 pos 所指位置插入一个值为 val 的元素并返回其迭代器
pop_back()	删除链表的尾元素
pop_front()	删除链表的首元素
remove(val)	删除链表中所有值为 val 的元素

2. 应用举例

例 2.5.1 建立顺序链表（HLOJ 9504）

先输入一个整数 n，再输入 n 个整数，按照输入的顺序建立链表，并遍历所建立的链表，输出这些数据。

例如，输入：5 1 2 3 4 5，输出：5 4 3 2 1。要求用 STL 之 list 实现。

可以把输入的数据不断用 push_back 函数插入表尾。输出链表则通过迭代器（类似于指针）从头开始顺序访问，具体代码如下：

```
#include <iostream>
#include <list>                                   // 包含 list 头文件
using namespace std;
list<int> createList(int n) {                     // 建立顺序链表
    list<int> myList;
```

```
    int t;
    for(int i=0; i<n; i++) {
        cin>>t;
        myList.push_back(t);                    // 元素 t 链接到表尾
    }
    return myList;
}
void prtList(list<int> myList) {                // 输出链表，每两个数据之间留一个空格
    list<int>::iterator it;                     // list<int>::iterator 类型迭代器
    for(it=myList.begin(); it!=myList.end(); it++) {
        if (it!=myList.begin())                 // 若 it 所指的不是第一个数据
            cout<<" ";                          // 则先输出一个空格
        cout<<*it;                              // 输出 it 所指的数据
    }
    cout<<endl;
}
int main() {
    int n;
    cin>>n;                                     // 输入数据个数
    list<int> lst=createList(n);                // 建立顺序链表
    prtList(lst);
    return 0;
}
```

注意，STL 之迭代器没有重载小于号，因此不能用条件 it<myList.end() 表示链表没有结束，而应用 it!=myList.end() 表达。如果要建立逆序链表，只需把语句“myList.push_back(t);”改为“myList.push_front(t);”即可。

注意，在用成员函数 erase 删除 myList 中迭代器 it 所指元素时，最好不要写成“myList.erase(it);”，因为此语句已使迭代器 it 销毁，若需要用 it 继续指向后续元素，需使 it 重新有明确指向，即写成“it=myList.erase(it);”或“myList.erase(it++);”。

读者可自行编程测试 list 常用成员函数的使用。

3. 运用 list 实现大整数加法

例 2.5.2 大整数 A+B（HLOJ 9515）

给定非负整数 A 和 B，请计算 A+B 的值。要求采用链表或 list 完成。

例如，输入：

2222222222222222 58944444444444444444

则输出：

58946666666666666666

采用链表或 list 实现两个大整数加法的基本思路：两个非负整数作为字符串输入，并用输入的字符串创建逆序链表，再根据“右对齐、逐位相加”的方法进行运算。由于创建的是逆序链表，因此按从头开始、顺序访问即可实现右对齐逐位相加。另外，在进位处理方面，可以用一个整型变量表示，其初值一开始设为 0，在计算过程中把其加到当前和中，并不断更新为最新的进位。

这里给出使用 list 的具体实现代码，读者可参考此代码写出使用自定义链表求解本例的代码。

```
// 利用 list 实现非负整数加法，两个整数以链表形式存储在 La 和 Lb 中，结果存放在 Lc 中。
#include <iostream>
#include <list>
#include <string>
using namespace std;
// 利用 list 实现非负整数加法，两个整数以链表形式存储在 La 和 Lb 中，结果存放在 Lc 中。
list<int> bigAddList(list<int> La, list<int> Lb) {
    list<int> Lc;
    list<int> ::iterator it1,it2;
    it1=La.begin();
    it2=Lb.begin();
    int carry=0;                                    // 保存进位
    // 当两个链表没有处理完毕时，逐位相加
    while(it1!=La.end() || it2!=Lb.end()) {
        int c=carry;                                // c 暂存当前和
        if(it1!=La.end()) {                         // 若第一个链表未结束，则把相应数位加到 c 中
            c = c + *it1;
            it1++;
        }
        if(it2!=Lb.end()) {                         // 若第二个链表未结束，则把相应数位加到 c 中
            c = c + *it2;
            it2++;
        }
        carry=c/10;                                 // 求得新的进位
        Lc.push_front(c%10);                        // 相加得到的当前和的余数插入结果链表头部
    }
    if(carry>0) Lc.push_front(carry);// 处理最后的进位
    return Lc;
}

int main() {
    string s,t ;
    cin>>s>>t;                                      // 非负整数作为字符串输入
    list <int> La, Lb, Lc;
    for(int i=0; i<s.size(); i++) {                 // 用第一个字符串创建逆序链表
        La.push_front(s[i]-' 0' );
    }
    for(int i=0; i<t.size(); i++) {                 // 用第二个字符串创建逆序链表
        Lb.push_front(t[i]-' 0' );
    }
    Lc=bigAddList(La, Lb);                          // 调用 bigAddList 求解两个非负整数相加
    // 遍历输出结果
    for (list<int>::iterator it = Lc.begin(); it != Lc.end(); it++) {
        cout<<*it;
    }
    cout<<endl;
    return 0;
}
```

设两个整数中最长的整数有 n 位，则 bigAddList 算法的时间复杂度为 O(n)。

若把 bigAddList 函数中的 push_front 改为 push_back，则要如何修改以上代码？读者可以自行思考并实现。

2.6 在线题目求解

例 2.6.1 顺序表的就地逆置（HLOJ 9503）

试写一算法，实现顺序表的就地逆置，即利用原表的存储空间将线性表 $(a_1, a_2, \cdots, a_n)$ 逆置为 $(a_n, a_{n-1}, \cdots, a_1)$。

输入：

测试数据有多组，处理到文件尾。每组测试数据在一行上输入数据个数 n（$n \leqslant 15$）及 n 个整数。

输出：

对于每组测试，将顺序表中的元素逆置存储于原表后输出。每两个整数间留一个空格。

输入样例：

```
12 32 22 88 25 75 29 5 41 89 83 24 1
7 36 83 39 86 62 89 24
```

输出样例：

```
1 24 83 89 41 5 29 75 25 88 22 32
24 89 62 86 39 83 36
```

解析：

实现顺序表的就地逆置的一种算法思想是以中间位置为界，逐个交换左右对称位置上的元素。顺序表的初始化、创建、遍历的实现与前述类似。具体代码如下：

```
#include<iostream>
using namespace std;
typedef int ElemType;
struct SqList {
    ElemType *elem;
    int length;
    void Init(int n);
    void Create(int n);
    void Traverse();
};
// 逆置顺序表，因需把逆置之后的结果返回，故使用引用参数
void reverseSqList(SqList &sl) {
    for(int i=0;i<sl.length/2;i++) {
        swap(sl.elem[i], sl.elem[sl.length-1-i]);
    }
}
int main() {
    int n;
```

```
    while(cin>>n) {
        SqList sl;
        sl.Create(n);
        reverseSqList(sl);
        sl.Traverse();
    }
    return 0;
}
// 遍历顺序表
void SqList::Traverse() {
    for(ElemType *p=elem; p<elem+length; p++) {
        if(p!=elem) cout<<" ";
        cout<<*p;
    }
    cout<<endl;
}
// 初始化顺序表
void SqList::Init(int n) {
    elem=new ElemType[n];
    length=0;
}
// 建立顺序表
void SqList::Create(int n) {
    Init(n);
    for(ElemType *p=elem; p<elem+n; p++) {
        cin>>*p;
        length++;
    }
}
```

以上这种代码的编写方式是为了便于读者体会如何以“数据结构方式”（先定义数据结构并实现基本操作，再调用已实现的基本操作求解问题）编写程序。而在比赛时，一般采用更简洁的代码编写方式。

例 2.6.2 合并升序单链表（HLOJ 9506）

各依次输入递增有序若干个不超过 100 的整数，分别建立两个单链表，将这两个递增的有序单链表合并为一个递增的有序链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间；且合并后的单链表中不允许包含重复的数据。然后输出合并后的单链表。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据首先在第一行输入数据个数 n；再在第二行和第三行分别输入 n 个依次递增有序的不超过 100 的整数。

输出：

对于每组测试，输出合并后的单链表，每两个数据之间留一个空格。

输入样例：

```
1
15
14 16 19 22 23 31 33 67 68 75 76 78 81 91 95
23 25 26 37 38 39 46 52 55 68 75 78 79 86 92
```

输出样例：

```
14 16 19 22 23 25 26 31 33 37 38 39 46 52 55 67 68 75 76 78 79 81 86 91 92 95
```

解析：

合并升序单链表的一种算法思想是从头开始分别用一个指针指向两个链表中的当前元素，然后逐个比较两个链表中的当前元素，把其中的小者链接到结果链表的尾部。若题目要求不能包含重复的元素，则在两个当前元素相等时，把其中一个链接到结果链表中的同时需跳过另一个链表中的相同元素。具体代码如下：

```
#include<iostream>
using namespace std;
typedef int ElemType;
struct LNode {
    ElemType data;                          // 数据域
    LNode* next;                            // 指针域
};
struct LinkList {
    LNode *head;                            // 头指针（带头节点）
    void Create(int n);                     // 建立含 n 个节点的单链表
    void Traverse();                        // 遍历，并输出内容
};
// 合并升序链表
void mergeList(LinkList &La,LinkList &Lb,LinkList &Lc) {
    LNode *pa,*pb,*pc;
    pa=La.head->next;                       // pa 指向第一个链表 La 的首元素
    pb=Lb.head->next;                       // pb 指向第二个链表 Lb 的首元素
    pc=Lc.head=La.head;                     // 用 La 的头节点作为 Lc 的头节点
    while(pa && pb) {                       // 当两个链表都没有结束时进行比较并处理
        if(pa->data<pb->data) {             // pa 所指元素小于 pb 所指元素
            pc->next=pa;                    // pa 所指节点链接到结果链表的最后
            pc=pa;                          // pc 指向结果链表中新的尾节点
            pa=pa->next;                    // pa 指向第一个链表 La 的下一个节点
        }
        else if(pa->data>pb->data) {        // pa 所指元素大于 pb 所指元素
            pc->next=pb;
            pc=pb;
            pb=pb->next;
        }
        else {                              // 相等时取 La 的元素，删除 Lb 的元素
            pc->next=pa;
            pc=pa;
            pa=pa->next;
            LNode *q=pb->next;
            delete pb;
            pb=q;
        }
    }
    pc->next = pa ? pa : pb;                // 插入 La 或 Lb 中的剩余节点
```

```
    delete Lb.head;                                    // 释放 Lb 的头节点
}
int main() {
    int T;
    cin>>T;
    while(T--) {
        LinkList a,b,c;
        int n;
        cin>>n;
        a.Create(n);
        b.Create(n);
        mergeList(a,b,c);
        c.Traverse();
    }
    return 0;
}
// 创建带头节点的单链表
void LinkList::Create(int n) {
    head=new LNode;                                    // 创建头节点
    head->next=NULL;                                   // 头节点的指针域置为空指针
    LNode *q=head;                                     // q 为尾指针，指向尾节点
    for (int i=0; i<n; i++) {
        LNode* p=new LNode;
        cin>>p->data;                                  // 输入元素值
        p->next=NULL;
        q->next=p;                                     // 新节点链接到尾节点之后
        q=p;                                           // q 指向新的尾节点
    }
}
// 遍历链表
void LinkList::Traverse() {
    LNode* p=head->next;
    int cnt=0;
    while (p) {
        cnt++;
        if (cnt>1) cout << " ";
        cout<<p->data;
        p=p->next;
    }
    cout << endl;
}
```

本题也可以采用 list 实现，可用 list 的成员函数 merge 合并升序链表，再用 list 的成员函数 unique 去重，读者可自行尝试实现。

例 2.6.3 单链表中重复元素的删除（HLOJ 9513）

按照数据输入的顺序建立一个单链表，并将单链表中重复的元素删除（值相同的元素只保留最早输入的那一个）。

输入：

测试数据有多组，处理到文件尾。对于每组测试，第一行输入元素个数 n；第二行输入 n 个整数。

输出：

对于每组测试，输出两行，第一行输出删除重复元素后的单链表元素个数；第二行输出删除重复元素后的单链表。

输入样例：

```
10
21 30 14 55 32 63 11 30 55 30
```

输出样例：

```
7
21 30 14 55 32 63 11
```

解析：

链表中删除重复元素的一种算法思想是遍历链表，对每个数据节点取数据域值与其后节点相比较，若相等，则删除其后节点。为便于删除操作，可设一个指针指向待删节点的前驱节点。具体代码如下：

```
#include<iostream>
using namespace std;
typedef int ElemType;
struct LNode {
    ElemType data;                          // 数据域
    LNode* next;                            // 指针域
};
struct LinkList {
    LNode *head;                            // 头指针（带头节点）
    void Create(int n);                     // 建立含 n 个节点的单链表
    void Traverse();                        // 遍历，并输出内容
};
void deleteSame(LinkList La, int &n) {
    LNode *p=La.head->next;                 // p 指向第一个数据节点
    while(p) {                              // 当链表未结束时重复处理
        LNode *q=p, *r=p->next;             // r 指向待删节点，q 指向其前驱节点
        while(r) {
            if (r->data==p->data) {         // 若 r 所指元素等于 p 所指元素，则删除 r 所指节点
                LNode* s=r;
                q->next=r->next;
                r=r->next;
                delete s;
                n--;
            }
            else {                          // 若 r 所指元素不等于 p 所指元素，则 q、r 往下走
                q=r;
                r=r->next;
            }
        }
```

```
            p=p->next;                          // p 指向下一个节点
        }
    }
    int main() {
        int n;
        while(cin>>n) {
            LinkList lst;
            lst.Create(n);
            deleteSame(lst,n);
            cout<<n<<endl;
            lst.Traverse();
        }
        return 0;
    }
    // 建立带头节点的单链表
    void LinkList::Create(int n) {
        head=new LNode;
        head->next=NULL;                        // 先建立一个带头节点的单链表
        LNode* q=head;                          // 尾指针
        for (int i=0; i<n; i++) {
            LNode* p=new LNode;
            cin>>p->data;                       // 输入元素值
            p->next=NULL;
            q->next=p;                          // 链接到尾指针 q 之后
            q=p;
        }
    }
    // 遍历链表
    void LinkList::Traverse() {
        LNode* p=head->next;
        int cnt=0;
        while (p) {
            cnt++;
            if (cnt>1) cout << " ";
            cout<<p->data;
            p=p->next;
        }
        cout << endl;
    }
```

注意，在 DeleteSame 函数中，在删除 r 所指节点后，r 接着指向下一个节点，但其前驱指针 q 不能同时往下指，请读者思考缘由，建议画图进一步理解。

本题也可以采用 list 实现，删除元素可用 list 的 erase 成员函数，读者可自行尝试实现。

习题

一、选择题

1. 等概率情况下，在表长为 n 的顺序表中插入一个元素所需移动的元素平均个数为（　　）。

A.$(n-1)/2$　　B.$n/2$　　C.$(n+1)/2$　　D.$n-i$

2. 假设某个带头节点的单链表的头指针为 *head*，则判定该表为空表的条件是（　　）。

A. *head*==NULL　　B. *head->next*==NULL

C. *head*!=NULL　　D. *head->next==head*

3. 在不带头节点的单链表中，某个节点的指针域指向该节点的（　　）。

A. 直接前趋　　B. 直接后继　　C. 开始节点　　D. 尾端节点

4. 线性表 L 在（　　）情况下适用于使用链式结构实现

A. 需不断对 L 进行删除插入　　B. 需经常修改 L 中的节点值

C. L 中含有大量的节点　　D. L 中含有少量的节点

5. 单链表的节点指针域为 *next*，其头节点由指针 *head* 指向，则删除第一个数据节点（由指针 p 指向）的语句序列为（　　）。

A. *p->next = head->next*;　　B. *head->next = p*;

C. *p = head->next*;　　D. *head->next = p->next*;

6. 创建一个包括 n 个节点的有序单链表的算法的时间复杂度是（　　）。

A. O(1)　　B. O(n)　　C. O(n^2)　　D. O($n\log_2 n$)

7. 单链表的指针域为 *next*，其头节点由指针 *head* 指向，则把指针 p 指向的节点链接到头节点之后的语句序列为（　　）。

A. *p->next=head->next*; *head->next=p*;　　B. *head->next=p*; *p->next=head->next*;

C. *head->next = p->next*; *p= head->next*;　　D. *p->next=head*; *head=p*;

8.（　　）结构中的节点最多只有一个前驱和一个后继。

A. 二叉树　　B. 图　　C. 线性表　　D. 以上都不是

9. 设两个单链表不带头节点且只提供头指针，则将长度为 n 的单链表链接在长度为 m 的单链表之后的算法的时间复杂度为（　　）。

A. O(1)　　B. O(n)　　C. O(m)　　D. O($m+n$)

10. 线性表 $L=(a_1, a_2, \cdots, a_n)$，下列说法正确的是（　　）。

A. 每个元素都有一个直接前驱和一个直接后继

B. 表中至少有一个元素

C. 表中元素要有序

D. 除第一个和最后一个元素外，其他元素都有且仅有一个直接前驱和一个直接后继

11. 在 n 个数据元素的顺序表中，算法时间复杂度为 O(1) 的操作是（　　）。

（1）访问第 i 个节点（$1 \leqslant i \leqslant n$）。

（2）求第 i 个节点的直接前驱（$2 \leqslant i \leqslant n$）。

（3）求第 i 个节点的直接后继（$1 \leqslant i \leqslant n-1$）。

（4）在第 i 个节点后插入一个新节点（$1 \leqslant i \leqslant n$）。

（5）删除第 i 个节点（$1 \leqslant i \leqslant n$）。

（6）排序。

A.（1）（2）（3）（4）（5）　　B.（1）（2）（3）

C.（4）（5）　　D.（6）

12. 在 n 个数据元素的单链表（仅有头指针）中，算法时间复杂度为 O(1) 的操作是（　　）。

（1）访问第 i 个节点（$1 \leqslant i \leqslant n$）。

（2）求第 i 个节点的直接前驱（$2 \leqslant i \leqslant n$）。

（3）求第 i 个节点的直接后继（$1 \leqslant i \leqslant n$-1）。

（4）在第 i 个节点后插入一个新节点（$1 \leqslant i \leqslant n$）。

（5）删除第 i 个节点（$1 \leqslant i \leqslant n$）。

（6）排序。

A.（1）（2）（3）　　B.（4）（5）　　C.（6）　　D. 以上都不是

13. 线性表 $(a_1, a_2, \cdots, a_n)$ 以顺序存储结构存储时，访问第 i 位置元素的时间复杂度为（　　）。

A. O(1)　　B. O(n)　　C. O(i)　　D. O(i–1)

14. 线性表 $(a_1, a_2, \cdots, a_n)$ 以仅有头节点的单链表存储时，访问第 i 位置元素的时间复杂度为（　　）。

A. O(1)　　B. O(n)　　C. O(i)　　D. O(i–1)

15. 不带头节点的单链表（头指针为 *head*）为空的判定条件是（　　）。

A. *head*==NULL　　B. *head->next*==NULL

C. *head->next=head*　　D. *head*!=NULL

16. 带头节点的循环单链表（头指针为 *head*）为空的判定条件是（　　）。

A. *head*==NULL　　B. *head->next*==NULL

C. *head->next==head*　　D. *head*!=NULL

17. 创建一个包括 n 个节点的单链表的时间复杂度是（　　）。

A. O(1)　　B. O(n)　　C. O(n^2)　　D. O($n\log_2 n$)

18. 在单链表中，指针域为 *next*，要将 q 所指节点链接到 p 所指节点之后，其语句序列应为（　　）。

A. *q->next=p->next*; *p->next=q*;　　B. *p->next=q*; *q->next=p->next*;

C. *q->next=p*+1; *p->next=q*;　　D. *p->next=q*; *q->next=p*;

19. 在双向链表中，前驱指针为 *prior*，后继指针为 *next*，在 p 指针所指的节点后插入 q 所指向的新节点，其语句序列是（　　）。

A. *q->prior=p*; *q->next=p->next*; *p->next=q*; *p->next->prior=q*;

B. *q->prior=p*; *q->next=p->next*; *p->next->prior=q*; *p->next=q;*

C. *p->next=q*; p-*>next->prior=q*; *q->prior=p*; *q->next=p->next*;

D. *p->next=q*; *q->prior=p*; *p->next->prior=q*; *q->next=q*;

20. 在双向链表中，前驱指针为 *prior*，后继指针为 *next*，删除 p 所指的节点的语句序列为（　　）。

A. *p->prior->next=p*; *p->prior=p->prior->prior*;

B. *p->prior*=p-*>next->next*; *p->next=p->prior->prior*;

C. *p->next->prior=p->prior*; *p->prior->next=p->next*;

D. *p->next=p->next->next*; *p->next->prior=p*;

二、应用题

1. 在如图 2-12 所示的静态链表中数组 A 中链接存储了一个线性表，表头指针为 A[0].*next*，试写出该线性表。

下标	0	1	2	3	4	5	6	7
next	3	5	7	2	0	4		1
data		60	50	78	90	34		40

图 2-12　静态链表

2. 对于带头节点的单循环链表，要求画出插入、删除节点的示意图，并写出相应的操作语句。

三、编程题

1. 顺序表中插入元素（HLOJ 9501）

设顺序表中的数据元素递增有序。请写一算法，将 x 插入顺序表的适当位置上，以保持该表的有序性。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据先在第一行输入数据个数 n（$n \leqslant 15$）及 n 个递增有序的整数；再在第二行输入一个整数 x。

输出：

对于每组测试，将插入 x 后保持该顺序表递增有序的顺序表中的整数输出，每两个整数间留一个空格。

输入样例：

```
2
10 124 392 410 487 757 940 989 3633 4228 4977
567
14 113 238 307 396 487 536 616 798 834 1697 2498 2640 2665 3271
100
```

输出样例：

```
124 392 410 487 567 757 940 989 3633 4228 4977
100 113 238 307 396 487 536 616 798 834 1697 2498 2640 2665 3271
```

2. 顺序表中删除元素（HLOJ 9502）

输入若干个不超过 100 的整数，存储到顺序表中，请写一算法，删除顺序表中所有值为 *item* 的数据元素，然后保持原数据顺序输出。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据先在第一行输入数据个数 n（$n \leqslant 40$）及 n 个不超过 100 的整数；再在第二行输入一个整数，表示要删除的的数据元素 *item*。

输出：

对于每组测试，保持原数据顺序输出删除顺序表中所有值为 *item* 后的数据元素，每两个数据之间留一个空格，若删除所有值为 *item* 的数据元素后为空表，则输出“empty”。引号不必输出。

输入样例：

```
2
30 38 76 77 53 38 38 80 16 79 38 48 50 59 88 15 47 15 95 39 38 46 73 52 77 26 28 31 98 60 26
```

```
38
10 1 1 1 1 1 1 1 1 1
1
```

输出样例：

```
76 77 53 80 16 79 48 50 59 88 15 47 15 95 39 46 73 52 77 26 28 31 98 60 26
empty
```

3. 两个有序单链表求差集（HLOJ 9507）

依次输入递增有序若干个不超过 100 的整数，分别建立两个递增有序单链表，分别表示集合 A 和集合 B。设计算法求出两个集合 A 和 B 的差集（仅由在集合 A 中出现而不在集合 B 中出现的元素所构成的集合），并存放于 A 链表中。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间。然后输出结果链表。测试数据保证结果链表至少存在一个元素。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据先在第一行输入数据个数 n 及 n 个依次递增有序的不超过 100 的整数，再在第二行输入数据个数 m 及 m 个依次递增有序的不超过 100 的整数。

输出：

对于每组测试，输出集合 A 与集合 B 的差集的单链表，每两个数据之间留一个空格。

输入样例：

```
1
11 10 14 23 25 26 31 34 42 51 65 90
10 10 41 42 46 51 58 59 60 68 97
```

输出样例：

```
14 23 25 26 31 34 65 90
```

4. 拆分单链表（HLOJ 9508）

输入若干个绝对值不超过 100 的整数，建立单链表 A，设计算法将单链表 A 分解为两个具有相同结构的链表 B、C，其中链表 B 的节点为链表 A 中值小于零的节点，而链表 C 的节点为链表 A 中值大于零的节点（链表 A 的元素类型为整型，要求链表 B、C 表利用链表 A 的节点，不另外占用其他的存储空间，若采用带头节点的单链表实现则允许再申请一个头节点）。然后分两行按原数据顺序输出链表 B、C。测试数据保证每个结果链表至少存在一个元素。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据在一行上输入数据个数 n 及 n 个不含整数 0 且绝对值不超过 100 的整数。

输出：

对于每组测试，分两行按原数据顺序输出链表 B、C，每行的每两个数据之间留一个空格。

输入样例：

```
1
10 49 53 -26 79 -69 -69 18 -96 -11 68
```

输出样例：

```
-26 -69 -69 -96 -11
49 53 79 18 68
```

5. 链表的逆置（HLOJ 9509）

输入若干个不超过 100 的整数，建立单链表，然后将链表中所有节点的链接方向逆置，要求仍利用原表的存储空间。输出逆置后的单链表。

输入：

首先输入一个整数 *T*，表示测试数据的组数，然后是 *T* 组测试数据。每组测试数据在一行上输入数据个数 *n* 及 *n* 个不超过 100 的整数。

输出：

对于每组测试，输出逆置后的单链表，每两个数据之间留一个空格。

输入样例：

```
1
11 55 50 45 40 35 30 25 20 15 10 5
```

输出样例：

```
5 10 15 20 25 30 35 40 45 50 55
```

6. 在单链表中删除 *mink* ~ *maxk* 的元素（HLOJ 9510）

输入若干个值不超过 1 000 的整数，建立递增有序的单链表，设计一个高效的算法，删除表中所有值大于 *mink* 且小于 *maxk* 的元素（若表中存在这样的元素），*mink* 和 *maxk* 可以与表中的元素相同，也可以不同。

输入：

首先输入一个整数 *T*，表示测试数据的组数，然后是 *T* 组测试数据。每组测试数据首先在第一行上输入数据个数 *n* 及 *n* 个递增有序的整数，存储在带头节点的单链表中；然后在下一行中输入整数 *mink*、*maxk*（*mink*<*maxk*），表示要删除表中所有值大于 *mink* 且小于 *maxk* 的元素。

输出：

对于每组测试，输出删除表中所有值大于 *mink* 且小于 *maxk* 的元素后单链表中的元素，每两个元素之间留一个空格。

输入样例：

```
1
9 8 125 251 351 467 492 508 731 742
100 300
```

输出样例：

```
8 351 467 492 508 731 742
```

7. 单链表中确定值最大的节点（HLOJ 9512）

输入若干个不超过 100 的整数，建立单链表，然后通过一趟遍历在单链表中确定值最大的节点。输出该节点的值及其序号。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据在一行上输入数据个数 n 及 n 个不超过 100 的整数。

输出：

对于每组测试，输出单链表中值最大的节点的值和该节点的序号。输出如下：

“max=d_{max} num=d_{num}”

其中，d_{max} 表示最大的节点的值，d_{num} 表示最大的节点的值所在节点的序号。若有多个相同的最大值，则以首次出现的为准。

输入样例：

```
1
30 85 97 43 70 69 29 77 22 64 25 55 39 95 69 99 61 97 69 59 12 88 55 75 66 13 75 36 85 67 69
```

输出样例：

```
max=99 num=15
```

8. 约瑟夫环（HLOJ 9514）

约瑟夫问题的一种描述是：编号为 1, 2, …, n 的 n 个人按顺时针方向围坐一圈，每人持有一个密码（正整数）。一开始以 6 作为报数上限值 m，从第一个人开始按顺时针方向自 1 开始顺序报数，报到 m 时停止报数。报 m 的人出列，将他的密码作为新的 m 值，再从他在顺时针方向上的下一个人开始重新从 1 报数，如此下去，直至所有人全部出列。要求用单向循环链表存储结构模拟约瑟夫环，并输出出列顺序。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据第一行输入 n，第二行输入 n 个不过 20 的正整数，依次表示这 n 个人所持有的密码。

输出：

对于每组测试，按照出列的顺序输出各人的编号。每两个编号之间留一个空格。

输入样例：

```
2
7
3 1 7 2 4 8 4
10
6 9 6 14 9 7 9 13 14 2
```

输出样例：

```
6 1 4 7 2 3 5
6 3 10 2 7 4 8 9 5 1
```

9. 大斐波数（HDOJ 1715）

斐波那契（Fibonacci）数列的定义：$f(1)=f(2)=1$，$f(n)=f(n-1)+f(n-2)$，（$n \geqslant 3$）；请计算 Fibonacci 的第 n 项。要求用链表或 STL 之 list 实现。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据输入一个整数 n（$1 \leqslant n \leqslant 1000$）。

输出：

对于每组测试，输出 Fibonacci 的第 n 项 $f(n)$。

输入样例：

```
2
3
105
```

输出样例：

```
2
392841376460687116 5730
```