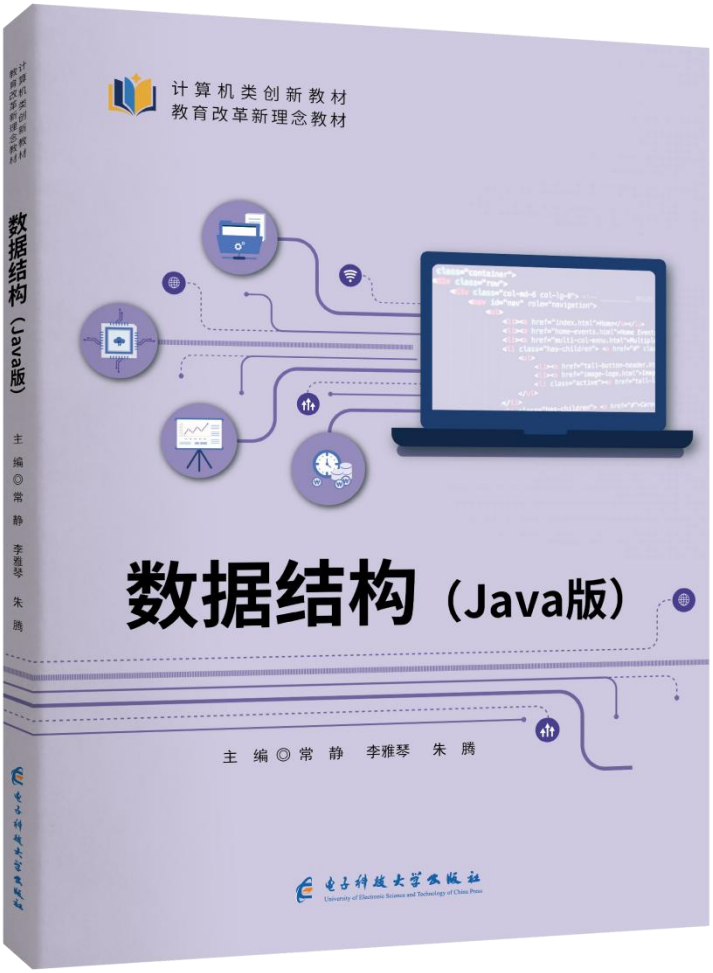
数据结构（JAVA）



类目：计算机类
书名：数据结构（JAVA 版）
主编：常静 李雅琴 朱腾
出版社：电子科技大学出版社
开本：大 16 开
书号：978-7-5770-1342-8
使用层次：通用
出版时间：2024 年 12 月
定价：49.80 元
印刷方式：双色
是否有资源：有

数据结构（JAVA版）

策划编辑：万晓桐
责任编辑：万晓桐
封面设计：旗语书装

ISBN 978-7-5770-1342-8
定价：49.80元

电子科技大学出版社

计算机类创新教材
教育改革新理念教材

数据结构（JAVA版）

主编◎常　静　李雅琴　朱　腾

电子科技大学出版社

计算机类创新教材
教育改革新理念教材

数据结构（JAVA版）

主编◎常　静　李雅琴　朱　腾

电子科技大学出版社
University of Electronic Science and Technology of China Press

计算机类创新教材
教育改革新理念教材

数据结构（Java 版）

主　审◎王　惠

主　编◎常　静　李雅琴　朱　腾

副主编◎赵大志　王　珏　夏保丽

王　睿　闫文静　程文杰

曹　非

电子科技大学出版社
University of Electronic Science and Technology of China Press

·成都·

图书在版编目（CIP）数据

数据结构：Java 版/常静，李雅琴，朱腾主编. 成都：成都电子科大出版社，2024. 12. -- ISBN 978-7-5770-1342-8

Ⅰ. TP311. 12；TP312. 8

中国国家版本馆 CIP 数据核字第 2025NN9666 号

数据结构（Java 版）

SHUJU JIEGOU（Java BAN）

常　静　李雅琴　朱　腾　主编

策划编辑　吴艳玲　万晓桐
责任编辑　兰　凯
责任校对　万晓桐
责任印制　梁　硕

出版发行　电子科技大学出版社
　　成都市一环路东一段 159 号电子信息产业大厦九楼　邮编 610051
主　　页　www. uestcp. com. cn
服务电话　028-83203399
邮购电话　028-83201495

印　　刷　涿州汇美亿浓印刷有限公司
成品尺寸　210 mm×285 mm
印　　张　15
字　　数　475 千字
版　　次　2024 年 12 月第 1 版
印　　次　2024 年 12 月第 1 次印刷
书　　号　ISBN 978-7-5770-1342-8
定　　价　49. 80 元

PREFACE
前　言

党的二十大报告中指出，教育、科技、人才是全面建设社会主义现代化国家的基础性、战略性支撑。必须坚持科技是第一生产力、人才是第一资源、创新是第一动力，深入实施科教兴国战略、人才强国战略、创新驱动发展战略。这三大战略共同服务于创新型国家的建设。

数据结构作为计算机专业及相关专业的一门必修课程，具有举足轻重的地位。它是一门集技术性、理论性和实践性于一体的课程。在学习本课程时，需要灵活运用数据结构和算法知识去解决实际问题。作为应用得最广泛的程序设计语言之一，Java 具有很好的封装性，易实现，因此本书选择 Java 作为开发语言。通过学习本书内容，读者既能加深对数据结构基本概念的理解和认识，又能提高对各种数据结构进行运算分析、设计的能力。

在选材与编排上，本书全面、系统地介绍了各种常见的数据结构及其存储表示，并讨论了数据结构的基本操作和实际算法。全书共 9 章，包括绪论、线性表、栈和队列、串、数组和广义表、树、图、查找和排序技术等内容。本书具有下面几个突出特色。

（1）内容精练，由浅入深、循序渐进。

本书各章都从基本概念入手，逐步介绍其特点和基本操作的实现，把重点放在基础知识的介绍上，缩减难度较大的内容，使理论叙述简洁明了、重点突出、详略得当。

（2）应用实例丰富、完整。

本书通过丰富的应用实例和源代码使理论和应用紧密结合，增强学生的理解能力，锻炼学生的程序设计思维，并且本书的代码有详细明了的注释，易于学生阅读。

（3）每章后面附有小结、实训和课后练习，便于学习、总结和提高。

本书结合学生的学习实际选择难度适中、逻辑合理，适于初学者和进阶者开拓思路、深入了解数据结构的使用方法和解题技巧，并附有详细的解答过程和注意要点，达到通俗易懂、由浅入深的效果，培养读者迁移知识的能力。

（4）采用 Java 的泛型方法来体现方法的通用性。

本书采用面向对象的观点讨论数据结构技术，先将抽象数据类型定义成接口，再结合具体的存储结构加以实现，并以各实现类为线索对类中各种操作的实现方法加以说明。

（5）图文并茂，便于学生直观地理解数据结构与算法。

本书通过图表的方式对数据结构及相应操作进行简单直接的描述，使内容更加浅显易懂。

本书参考了一些已正式发表或出版的文献以及非正式出版的网络资源，由于篇幅所限和其他原因，书末仅列出了部分参考资料，无法一一注明全部参考资料的来源和作者的具体姓名，在此一并向各位作者表示感谢！如果涉及版权问题，希望各读者告知编者处理。

由于数据结构本身还在探索之中，加上编者的水平和能力有限，本书难免有疏漏之处，恳请各位同人和广大读者给予批评指正，也希望各位能将实践过程中的经验和心得与我们交流。

编　者

2024 年 12 月

目　录 CONTENTS

第 1 章　绪　　论

学习目标

◎知识目标

1. 掌握数据结构的定义。
2. 了解数据结构包含的逻辑结构、存储结构和运算三个方面的相互关系。
3. 掌握算法的定义及其特性。
4. 了解影响算法效率的因素。

◎能力目标

1. 提升编程能力、培养逻辑思维能力。
2. 提高算法效率以及增强问题解决能力。

◎思政目标

培养学生的科学探索精神、家国情怀、团队协作精神和正确的价值观，为他们个人的全面发展打下坚实的基础。

1.1　数据结构概述

1.1.1　数据结构的定义

用计算机解决一个具体的问题大致需要经过以下几个步骤。

(1)分析问题，确定数据模型。

(2)设计相应的算法。

(3)编写程序，运行并调试程序直至得到正确的结果。

寻求数学模型的实质是先分析问题，从中提取操作的对象，并找出这些操作对象之间的关系；然后用数学语言加以描述。有些问题的数据模型可以用具体的数学方程式来表示，但更多的实际问题是无法用数学方程式来表示的，这就需要从数据入手来分析并得到解决问题的方法。

数据是描述客观事物的数、字符以及所有能输入计算机并被计算机程序处理的符号的集合。例如，在日常生活中使用的各种文字、数字和特定符号都是数据。从计算机的角度看，数据是所有能被输入计算机且能被计算机处理的符号的集合。它是计算机操作的对象的总称，也是计算机处理的信息的某种特定的符号表示形式（例如，A 班学生数据包含了该班全体学生的记录）。

通常以数据元素作为数据的基本单位（例如，A 班中的每个学生记录都是一个数据元素），也就是说数据元素是组成数据的有一定意义的基本单位，在计算机中通常作为整体处理，在有些情况下，数据元素也称为“元素”“节点”“记录”等。有时候，一个数据元素可以由若干个数据项组成。数据项是具有独立含义的数据最小单位，也称为“域”[例如，A 班中的每个数据元素（学生记录）是由学号、姓名、性别和班号等数据项组成的]。

数据对象是性质相同的有限个数据元素的集合，它是数据的一个子集，例如大写字母数据对象是集合 $C=\{'A','B','C',\cdots,'Z'\}$；1～100 的整数数据对象是集合 $N=\{1,2,\cdots,100\}$。在默认情况下，数据结构中的数据都是指的数据对象。

数据结构是指所涉及的数据元素的集合以及数据元素之间的关系，由数据元素之间的关系构成结构，因此可把数据结构看成是带结构的数据元素的集合。数据结构包括如下几个方面。

（1）数据元素之间的逻辑关系，即数据的逻辑结构，它是数据结构在用户面前呈现的形式。

（2）数据元素及其关系在计算机存储器中的存储方式，即数据的存储结构，也称为“数据的物理结构”。

（3）施加在该数据上的操作，即数据的运算。

数据的逻辑结构是从逻辑关系（主要是指数据元素的相邻关系）上描述数据的，它与数据的存储无关，是独立于计算机的，因此数据的逻辑结构可以看作是从具体问题抽象出来的数学模型。

数据的存储结构是逻辑结构用计算机语言的实现或在计算机中的表示（也称为“映射”），也就是逻辑结构在计算机中的存储方式，它是依赖于计算机语言的，一般在高级语言（例如 Java、C/C++等语言）的层次上来讨论存储结构。

数据的运算是定义在数据的逻辑结构之上的，每种逻辑结构都有一组相应的运算，最常用的运算有检索、插入、删除、更新、排序等。数据的运算最终需要在对应的存储结构上用算法实现。

因此，数据结构是一门讨论“描述现实世界实体的数学模型（通常为非数值计算）及其之上的运算在计算机中如何表示和实现”的学科。

1.1.2 数据的逻辑结构

讨论数据结构的目的是用计算机求解问题，而分析并弄清数据的逻辑结构是求解问题的基础，也是求解问题的第一步。

数据的逻辑结构是面向用户的，它反映数据元素之间的逻辑关系，而不是物理关系。其是独立于计算机的。

1. 逻辑结构的表示

由于数据的逻辑结构是面向用户的，因此可以采用表格、图等用户容易理解的形式表示。下面通过几个实例加以说明。

【例 1-1】 采用表格形式给出一个高等数学成绩表，表中的数据元素是学生成绩记录，每个数据元素由三个数据项（学号、姓名和分数）组成。

解：一个用表格表示的高等数学成绩表（见表 1-1 所列），表中的每一行对应一个学生记录。

表 1-1　高等数学成绩表

学号	姓名	分数/分
2018001	王×	90
2018010	刘×	62
2018006	陈×	54
2018009	张×	95
2018007	许×	76
2018012	李×萍	88
2018005	李×英	82

【例 1-2】　下图为某大学组织结构图，其大学下设若干个学院和若干个处，每个学院下设若干个系，每个处下设若干个科或办公室。

解：某大学组织结构如图 1-1 所示，该图中的每个矩形框为一个节点，对应一个单位名称。

【例 1-3】　下面是全国部分城市交通线路图。

解：全国部分城市交通线路如图 1-2 所示，图中的每个节点表示一个城市。

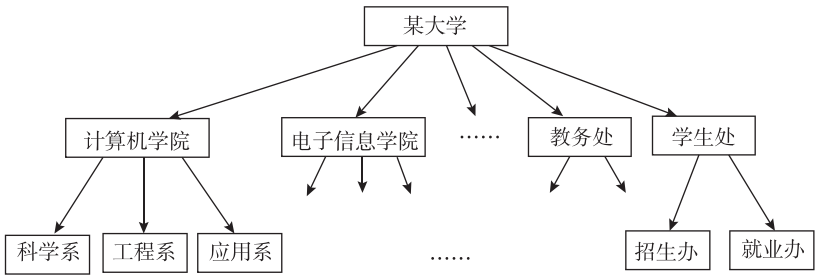


图 1-1　某大学组织结构图

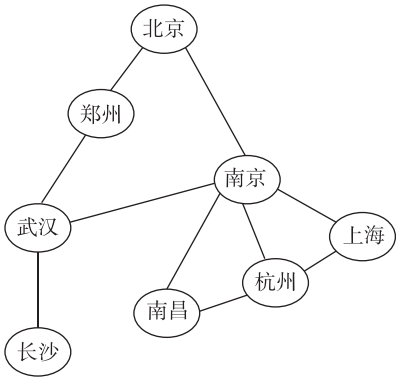


图 1-2　全国部分城市交通线路图

上面几个实例都是数据逻辑结构的表示形式，从中可以看出，数据逻辑结构主要是从数据元素之间的相邻关系来考虑的，数据结构课程中主要讨论这种相邻关系，在实际应用中很容易将其推广到其他关系。

实际上同一个数据逻辑结构可以采用多种方式来表示，假设用学号表示一个成绩记录，高等数学成绩表的逻辑结构也可以用图 1-3 表示。

图 1-3　高等数学成绩表的另一种逻辑结构表示

为了更通用地描述数据的逻辑结构，通常采用二元组表示数据的逻辑结构，一个二元组如下：

$$B=(D,R)$$

式中：B 是一种逻辑数据结构；D 是数据元素的集合，在 D 上数据元素之间可能存在多种关系；R 是所有关系的集合，即

$$D=\{d_i \mid 0\leqslant i\leqslant n-1, n\geqslant 0\}$$

$$R=\{r_j \mid 1\leqslant j\leqslant m, m\geqslant 0\}$$

式中：d_i 表示集合 D 中的第 $i(0\leqslant i\leqslant n-1)$ 个数据元素（或节点）；n 为 D 中数据元素的个数，特别地，若 $n=0$，则 D 是一个空集，此时 B 也就无结构可言；$r_j(1\leqslant j\leqslant m)$ 表示集合 R 中的第 j 个关系；m 为 R 中关系的个数，特别地，若 $m=0$，则 R 是一个空集，表明集合 D 中的数据元素间不存在任何关系，彼此是独立的，这与数学中

集合的概念是一致的。

说明：为了方便起见，数据结构中元素的逻辑序号统一从0开始。

R 中的某个关系 $r_j(1\leqslant j\leqslant m)$ 是序偶的集合，对于 r_j 中的任一序偶$<x,y>(x,y\in D)$，把 x 叫作序偶的第一元素，把 y 叫作序偶的第二元素，又称“序偶的第一元素为第二元素的前驱元素”，称“第二元素为第一元素的后继元素”。例如在$<x,y>$的序偶中，x 为 y 的前驱元素，而 y 为 x 的后继元素。

若某个元素没有前驱元素，则称该元素为开始元素；若某个元素没有后继元素，则称该元素为终端元素。

对于对称序偶，满足这样的条件：若$<x,y>\in r(r\in R)$，则$<y,x>\in r(x,y\in D)$，可用圆括号代替尖括号，即$(x,y)\in r$。

对于 D 中的每个数据元素，通常用一个关键字来唯一标识，例如高等数学成绩表中学生成绩记录的关键字为学号。前面三个示例均可以采用二元组来表示其逻辑结构。

【例1-4】 采用二元组表示前面三个例子的逻辑结构。

解：高等数学成绩表（假设学号为关键字）的二元组表示如下。

$B_1=(D,R)$

$D=\{2018001,2018010,2018006,2018009,2018007,2018012,2018005\}$

$R=\{r_1\}$　　//表示只有一种逻辑关系

$r_1=\{<2018001,2018010>,<2018010,2018006>,<2018006,2018009>,$
$<2018009,2018007>,<2018007,2018012>,<2018012,2018005>\}$

某高校组织结构（假设单位名为关键字）的二元组表示如下。

$B_2=(D,R)$

$D=\{$某大学，计算机学院，电子信息学院，…，教务处，学生处，…，科学系，工程系，应用系，…，招生办，就业办$\}$

$R=\{r_1\}$

$r_1=\{<$某大学，计算机学院$>,<$某大学，电子信息学院$>,\cdots,<$某大学，教务处$>,$
$<$某大学，学生处$>,\cdots,<$计算机学院，科学系$>,<$计算机学院，工程系$>,$
$<$计算机学院，应用系$>,\cdots,<$学生处，招生办$>,<$学生处，就业办$>\}$

全国部分城市交通图（假设城市名为关键字）的二元组表示如下。

$B_3=(D,R)$

$D=\{$北京，郑州，武汉，长沙，南京，南昌，杭州，上海$\}$

$R=\{r_1\}$

$r_1=\{($北京，郑州$),($北京，南京$),($郑州，武汉$),($武汉，南京$),($武汉，长沙$),($南京，南昌$),$
$($南京，上海$),($南京，杭州$),($南昌，杭州$),($南昌，上海$)\}$

2. 逻辑结构的类型

在现实生活中，数据呈现不同类型的逻辑结构，归纳起来数据的逻辑结构主要分为以下几种类型。

（1）集合：结构中的数据元素之间除了“同属于一个集合”的关系外没有其他关系，与数学中集合的概念相同。

（2）线性结构：若结构是非空的，则有且仅有一个开始元素和终端元素，并且所有元素最多只有一个前驱元素和一个后继元素。

在例1-1的高等数学成绩表中，每一行为一个学生成绩记录（或成绩元素），其逻辑结构特性是只有一个开始记录（即姓名为王×的记录）和一个终端记录（也称为“尾记录”，即姓名为李×英的记录），其余每个记录只有一个前驱记录和一个后继记录，也就是说记录之间存在一对一的关系，其逻辑结构特性为线性结构。

（3）树形结构：若结构是非空的，则有且仅有一个元素为开始元素（也称为“根节点”），可以有多个终端

元素,每个元素有零个或多个后继元素,除开始元素外每个元素有且仅有一个前驱元素。

例1-2中某大学组织结构图的逻辑结构特性是只有一个开始节点(即大学名称节点),有若干个终端节点(例如科学系等),每个节点有零个或多个下级节点,也就是说节点之间存在一对多的关系,其逻辑结构特性为树形结构。

(4)图形结构:若结构是非空的,则每个元素可以有多个前驱元素和多个后继元素。

例1-3中全国部分城市交通线路图的逻辑结构特性是每个节点和一个或多个节点相连,也就是说节点之间存在多对多的关系,其逻辑结构特性为图形结构。

1.1.3 数据的存储结构

数据的存储结构是逻辑结构用计算机语言的实现或在计算机中的表示(亦称为"映象"),也就是逻辑结构在计算机中的存储方式,它是依赖于计算机语言的。数据元素之间的关系在计算机中有两种表示方式:顺序映象和非顺序映象。归纳起来,数据的存储结构在计算机中有以下四种。

1. 顺序存储结构

顺序存储结构是把逻辑上相邻的节点存储在物理位置相邻的存储单元里,节点之间的逻辑关系由存储单元的邻接关系来体现。由此得到的存储结构称为顺序存储结构,通常顺序存储结构是借助于计算机程序设计语言的数组来描述的。

顺序存储结构的主要优点是节省存储空间,因为分配给数据的存储单元全部用于存放节点的数据,节点之间的逻辑关系没有占用额外的存储空间。采用这种结构时,可实现对节点的随机存取。然而顺序存储结构的主要缺点是不便于修改,在对节点进行插入、删除运算时,可能要移动一系列的节点。

2. 链式存储结构

链式存储结构不要求逻辑上相邻的节点在物理位置上也相邻,节点间的逻辑关系是由附加的"指针"字段表示的。由此得到的存储结构称为链式存储结构。

链式存储结构的优点是便于修改,在对节点进行插入、删除操作时,仅需要修改相应节点的"指针域",不必移动节点。但与顺序存储结构相比,链式存储结构的存储空间利用率较低,因为分配给数据的存储单元有一部分被用来存储节点间的逻辑关系了。另外,由于逻辑上相邻的节点在物理位置上并不一定相邻,所以不能对节点进行随机访问操作。

3. 索引存储结构

索引存储结构通常在存储节点信息的同时建立附加的索引表。索引表的每一项称为索引项,索引项的一般形式是(关键字,地址),关键字标识唯一的节点,地址作为指向节点的指针。这种带有索引表的存储结构可以大大提高数据查找的速度。

线性结构采用索引存储结构后,可以对节点进行随机访问。在对节点进行插入、删除运算时,只需移动存储在索引表中对应节点的存储地址,而不必移动存放在节点表中节点的数据,所以仍保持较高的数据修改效率。索引存储结构的缺点是增加了索引表及存储空间开销。

4. 哈希存储结构

哈希存储结构的基本思想是根据节点的关键字通过哈希函数直接计算出一个值,并将这个值作为该节点的存储地址。

哈希存储结构的优点是查找速度快,只要给出待查找节点的关键字,就可以计算出该节点的存储地址。与前三种存储结构不同的是,哈希存储结构只存储节点的数据,不存储节点之间的逻辑关系。哈希存储结构一般适用于要求对数据能够进行查找和插入的场景。

1.2 算法的描述和算法分析

1.2.1 算法的描述

算法是对特定问题求解步骤的一种描述，它是指令的有限序列，其中，每条指令表示一个或多个操作。一个算法具有以下五个重要的特性。

1. 有穷性

一个算法必须总是在执行有穷步（对任何合法的输入值）之后，且每一步都可在有穷时间内完成。即一个算法对于任意一组合法输入值，在执行有穷步之后一定能够结束。

2. 确定性

算法中每一条指令都必须有确切的含义，使读者在理解时不会产生二义性。同时，在任何条件下，算法都只有一条执行路径，即对于相同的输入只能得出相同的结果。

3. 可行性

算法中所描述的操作必须足够基本，且都可以通过已经实现的基本操作执行有限次来实现。

4. 输入

作为算法加工对象的量值，一个算法有 0 个或多个输入。有的量值需要在算法执行过程中输入，而有的算法表面上没有输入，实际上已经被嵌入算法中。

5. 输出

输出是一组与“输入”有确定对应关系的量值，是算法进行信息加工后所得到的产物，一个算法可以有一个或多个输出。

设计一个算法时，它应该满足以下四个要求。

（1）正确性。要求算法能够正确地实现预先规定的功能和满足性能要求，这是最重要也是最基本的标准。目前，大多数算法是用自然语言描述需求的，它至少应包括输入、输出和加工处理等的明确无歧义的描述。设计或选择算法应当能正确地反映这种需求。

（2）可读性。算法应当易于理解，可读性强。为了达到这个要求，算法必须做到逻辑清晰、简单并且结构化。晦涩难懂的程序容易隐藏较多错误，难以调试和修改。

（3）健壮性。算法要求具有良好的容错性，能够提供异常处理，能够对输入进行检查。不经常产生异常中断或死机现象。例如，一个求矩形面积的算法，当输入的坐标集合不能构成一个矩形时，不应继续计算，而应当报告输入错误。同时，处理错误的方法是返回一个表示错误的值，而不是输出错误信息或直接异常中断。

（4）效率与低存储量要求。通常算法的效率主要指算法的执行时间。对于同一个问题，如果能用多种算法进行求解，那么执行时间短的算法效率高。算法存储量指的是在算法执行过程中所需的存储空间。效率与低存储量要求这两者都与问题的规模有关。例如，求 100 个学生的平均成绩和求 10 000 个学生的平均成绩在时间和空间的成本上必然是存在差异的。

1.2.2 影响算法效率的因素

当一个算法用高级语言实现以后，在计算机上运行时所消耗的时间与很多因素有关，主要因素列举如下。

①依据算法所选择的具体策略。

②问题的规模,如求 100 以内还是 1 000 以内的素数。

③编写程序的语言,对于同一个算法,实现语言的级别越高,执行效率往往越低。

④编译程序所产生的计算机代码的质量。

⑤计算机执行指令的速度。

很显然,一个算法用不同的策略实现,或用不同的语言实现,或在不同的计算机上执行,它所耗费的时间是不一样的,因而效率均不相同。由此可知,使用一个绝对的时间单位去衡量一个算法的效率是不准确的。在上述五个因素当中,最后三个均与具体的计算机有关,抛开这些与计算机硬件、软件有关的因素,仅考虑算法本身的效率,可以认为一个特定算法的"执行工作量"只依赖于问题的规模,换而言之是问题的规模的函数。

1.2.3 算法效率的评价

一个算法是由控制结构(顺序、分支和循环)和原操作(指固有数据类型的操作)构成的,算法的执行时间取决于二者的综合结果。为了便于比较同一问题的不同算法,通常从算法中选取一种对于所研究的问题来说是基本运算的原操作,算法执行的时间大致为基本运算所需的时间与其运行次数(一条语句的运行次数称为语句频度)的乘积。

显然,在一个算法中,执行的基本运算次数越少,其执行时间就相对越少;执行基本运算的次数越多,其运行时间也相对越多。换言之,一个算法的执行时间可以看成基本运算执行的次数。

算法基本运算次数 $T(n)$ 是问题规模 n 的某个函数 $f(n)$,记作

$$T(n)=O(f(n))$$

式中,记号"O"读作"大欧"(值数量级),它表示随问题规模 n 的增大,算法执行时间的增长和 $f(n)$ 的增长率相同,称为算法的时间复杂度。

"O"的形式定义为:若 $f(n)$ 是正整数 n 的一个函数,则 $T(n)=O(f(n))$ 表示存在一个正的常数 M,使得当 $n\geqslant n_0$ 时都满足 $|T(n)|\leqslant M|f(n)|$,也就是只求出 $T(n)$ 的最高阶,忽略其低阶项和常数,这样既可以简化计算,又可以较为客观地反映当 n 很大时算法的效率。

一个没有循环的算法中基本运算次数与问题规模 n 无关,记为 $O(1)$,也称"常数阶"。一个只有一重循环的算法中基本次数与问题规模 n 的增长呈线性增大关系,记为 $O(n)$,也称"线性阶"。例如,以下 3 个程序段:

```
(a){ ++x; s = 0; }
(b)for(i = 1; i <= n; i ++ ){ ++x; s += x; }
(c)for(j = 1; j <= n; j ++ )
      for(k = 1; k < =n; k ++) { ++x; s += x; }
```

含基本操作"++x"的语句频度分别为 1、n 和 n^2,则这 3 个程序段的时间复杂度分别为 $O(1)$、$O(n)$ 和 $O(n^2)$,分别称为常量阶、线性阶和平方阶。各种不同数量级对应的值存在如下关系:

$$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)<O(n!)$$

【例 1-5】 分析以下算法的时间复杂度。

```
void fun(int a[ ],int n,int k)
{
    int i;
    i= 0;                          //语句(1)
    while(i < n && a[i] ! = k)     //语句(2)
        i ++;                      //语句(3)
    return (i);                    //语句(4)
}
```

解　该算法完成在一维数组 $a[n]$ 中查找给定值 k 的功能。语句(3)的频度不仅与问题规模 n 有关，还与输入实例中 a 的各个元素取值是否等于 k 的取值有关，即与输入实例的初始状态有关。若 a 中没有与 k 相等的元素，则语句(3)的频度为 n；若 a 中的第一个元素 $a[0]$ 等于 k，则语句(3)的频度是常数0。在这种情况下，可用最坏情况下的时间复杂度作为算法的时间复杂度。这样做的原因是，最坏情况下的时间复杂度是在任何输入实例里运行时间的上界。

有时也可以选择以算法的平均时间复杂度作为讨论的目标。所谓平均时间复杂度，是指所有可能的输入实例在以等概率出现的情况下算法的期望运行时间与问题规模 n 的数量级的关系。例1-4中，k 出现在任何位置的概率相同，都为 $1/n$，则语句(3)的平均时间复杂度为

$$(0+1+2+\cdots+(n-1))/n=(n-1)/2$$

它决定此程序的平均时间复杂度的数量级为 $O(n)$。

【例1-6】　分析以下算法的时间复杂度。

```
float RSum(float list [ ], int n)
{
        count ++;
    if(n){
        count ++;
        return RSum (list, n-1) + list[n-1];
    }
    count ++;
    return 0;
}
```

解　该算法是求数组元素之和的递归程序。为了确定这一递归程序的程序步，首先考虑当 $n=0$ 时的情况。显然，当 $n=0$ 时，程序只执行 if 条件判断语句和第二条 return 语句，所需程序步数为2。当 $n>0$ 时，程序在执行 if 条件判断语句后，将执行第一条 return 语句。此 return 语句不是简单返回，而是在调用函数 RSum(list,$n-1$)后，再执行一次加法运算后返回。

设 RSum(list,n)的程序步为 $T(n)$，则 RSum(list,$n-1$)为 $T(n-1)$，那么，当 $n>0$ 时，$T(n)=T(n-1)+2$。于是有

$$T(n)=\begin{cases}2 & n=0\\ T(n-1)+2 & n>0\end{cases}$$

这是一个递推关系式，它可以通过转换成如下公式来计算

$$\begin{aligned}T(n)&=2+T(n-1)=2+2+T(n-2)\\&=23+T(n-3)\\&\vdots\\&=2n+T(0)\\&=2n+2\end{aligned}$$

根据上述结果可知，该程序的时间复杂度为线性阶，即 $O(n)$。

1.2.4　算法的存储空间需求

一个算法的空间复杂度是指算法运行所需的存储空间。算法运行所需的存储空间包括如下两个部分。

1. 固定空间需求

固定空间所处理数据的规模大小和个数无关，换言之，与问题实例的特征无关，主要包括程序代码、常

量、简单变量、定长成分的结构变量所占的空间。

2. 可变空间需求

可变空间大小与算法在某次执行中处理的特定数据的规模有关。例如,分别包含100个元素的两个数组相加,与分别包含10个元素的两个数组相加,所需的存储空间显然是不同的。这部分存储空间包括数据元素所占的空间,以及算法执行所需的额外空间,例如,运行递归算法所需的系统栈空间。

在对算法进行存储空间分析时,只考察辅助变量所占空间,所以空间复杂度是对一个算法在运行过程中临时占用的存储空间大小的度量,一般也作为问题规模 n 的函数,以数量级的形式给出,记作

$$S(n)=O(g(n))$$

若所需额外空间相对于输入数据量来说是常数,则称此算法为原地工作或就地工作;若所需存储空间依赖特定的输入,则通常按最坏情况考虑。

【例1-7】 分析例1-4算法的空间复杂度。

解 对于例1-4的算法,只定义了一个辅助变量 i,临时存储空间大小与问题规模 n 无关,所以空间复杂度为 $O(1)$。

【例1-8】 有如下算法,求其空间复杂度。

```
void fun(int a[], int n,int k)
{
    int i;
    if(k == n - 1) {
        for(i=0;i<n;i++)
            System. out. println(a[i]);
    }
    else
    {
        for(i=k;i<n;i ++)
            a[i]=a[i]+i * i;
        fun(a, n, k+1);
    }
}
```

设 fun(a,n,k)的临时空间大小为 $S(k)$,其中定义了一个辅助变量 k,并有

$$S(k)=\begin{cases}1 & k=n-1\\ 1+S(k+1) & \text{其他}\end{cases}$$

计算 fun(a,n,0)所需的空间为 $S(0)$,则

$$\begin{aligned}S(0)&=1+S(1)=1+1+S(2)\\&=1+1+\cdots+1+S(n-1)\\&=\underbrace{1+1+\cdots+1}_{n\text{个}1}=O(n)\end{aligned}$$

1.3 数据结构的目标

从数据结构的角度看,一个求解问题可以通过抽象数据类型的方法来描述,也就是说抽象数据类型对一个求解问题从逻辑上进行了准确的定义,所以抽象数据类型由数据的逻辑结构和抽象运算两部分组成。

接下来用计算机解决这个问题。首先要设计其存储结构，然后在存储结构上设计实现抽象运算的算法。一种数据的逻辑结构可以映射成多种存储结构，抽象运算在不同的存储结构上实现可以对应多种算法，而且在同一种存储结构上实现也可能有多种算法，同一问题的这么多算法哪一种更好呢？好的算法的评价标准是什么呢？

算法的评价标准就是算法占用计算机资源的多少，占用计算机资源越多的算法越坏，反之占用计算机资源越少的算法越好。这是通过算法的时间复杂度和空间复杂度分析来完成的，所以设计好算法的过程如图1-4所示。

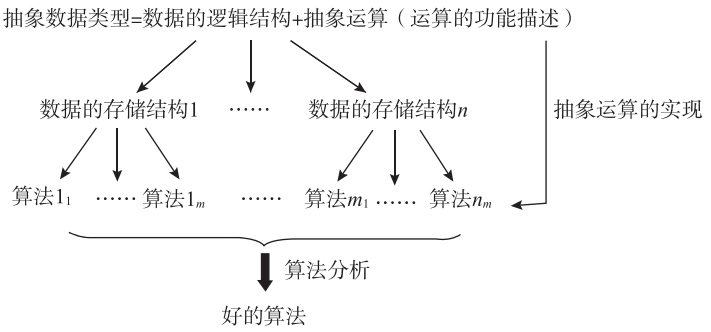


图1-4　设计好算法的过程

在采用Java面向对象的程序设计语言实现抽象数据类型时，通常将抽象数据类型设计成一个Java类，采用类的数据变量表示数据的存储结构，将抽象运算通过类的公有方法实现，如图1-5所示。

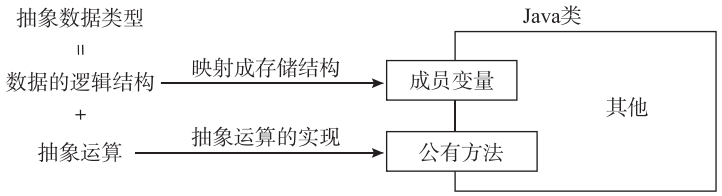


图1-5　用Java类实现抽象数据类型

从中看到算法设计分为三个步骤，即通过抽象数据类型进行问题定义，设计存储结构和设计算法。这三个步骤是不是独立的呢？结论为不是独立的，因为不可能设计出一大堆算法后再从中找出一个好的算法，也就是说必须以设计好算法为目标来设计存储结构。由于数据存储结构会影响算法的好坏，所以设计存储结构是关键的一步，在选择存储结构时需要考虑其对算法的影响。存储结构对算法的影响主要有以下两个方面。

(1)存储结构的存储能力：如果存储结构的存储能力强、存储的信息多，算法会方便设计；反之，过于简单的存储结构可能要设计一套比较复杂的算法，往往存储能力是与所使用的空间大小成正比的。

(2)存储结构应与所选择的算法相适应：存储结构是实现算法的基础，也会影响算法的设计，其选择要充分考虑算法的各种操作，并与算法的操作相适应。

除此之外，程序人员还需要具有基本的算法分析能力，能够熟练判别“好”算法和“坏”算法。

总之数据结构的目标就是针对求解问题设计好的算法。为了达到这一目标，程序人员不仅要具备较好的编程能力，还需要掌握各种常用的数据结构，例如线性表、栈和队列、二叉树和图等，这些是在后面各章中要学习的内容。

【例1-8】 设计一个完整的程序实现例1-6的抽象数据类型，并用相关数据进行测试。

解　设计求解本例程序的过程如下。

□　问题描述

见例1-6中的抽象数据类型Set和TwoSet。

□　设计存储结构

用一个E类型的数组存放集合，用一个整型变量size表示该数组中实际元素的个数，为此设计一个泛型类。这样的集合泛型中数据变量如下：

```
class Set<E>                                    //集合泛型类
```

```
{final int MaxSize=100;                          //集合中最多元素个数
    E[ ]data;                                    //存放集合元素
    int size;
    public Set( )                                //构造方法
    {data=(E[ ])new Object[MaxSize];             //强制转换为 E 类型数组
        size=0;
    }
    …
}
```

这样 Set 泛型类的一个对象存储一个集合。

TwoSet 泛型类中不含有任何数据成员,所处理的集合通过成员方法的形参提供。

□　设计运算算法

在集合类中创建一个空集合通过构造方法实现,Set 泛型类包含以下基本运算方法。

- int getsize( ):返回集合的长度。
- E get(int *i*):返回集合的第 *i* 个元素。
- boolean IsIn(E *e*):判断 *e* 是否在集合中。
- boolean add(E *e*):将元素 *e* 添加到集合中。
- boolean delete(E *e*):从集合中删除元素 *e*。
- void display( ):输出集合中的元素。

这些成员方法的实现如下:

```
public int getsize( )                            //返回集合的长度
{return size;   }
public E get(int i)                              //返回集合的第 i 个元素
{return(E) data [i];}
public boolean IsIn(E e)                         //判断 e 是否在集合中
{for(int i=0;i<size; i++)
        if (data[i]==e)
            return true;
    return false;
}
public boolean add(E e)                          //将元素 e 添加到集合中
{if (IsIn(e)) return false;                      //元素已在集合中返回 false
    else                                         //否则插入末尾并返回 true
    { data[size]=e;
        size++;
        return true;
}
}
public boolean delete(E e)                       //从集合中删除元素 e
{int i=0;
    while(i<size&&data[i]!=e)i++;
```

```
        if(i>=size)    return false;                    //未找到元素 e 返回 false
        for(int j =i+1;j<size;j++)
            data[j-1]=data[j];
        size--;
        return true;                                    //成功删除元素 e 返回 true
    }
    public void display()                               //输出集合中的元素
    {for(int i=0;i<size;i++)
        { if(i==0)System. out. print(data[i]);
            else System. out. print(""+data[i]);
        }
        System. out. println();
    }
```

TwoSet 泛型类的基本运算方法如下。

- Union(Set<E> s1,Set<E>s2):求 s3=s1∪s2。
- Intersection(Set<E>s1, Set<E>s2):求 s3=s1∩s2。
- Difference(Set<E> s1, Set<E>s2):求 s3=s1-s2。

这些成员方法的实现如下：

```
    public Set<E> Union(Set<E>s1, Set<E>s2)             //求 s3=s1∪s2
    {Set<E>s3=new Set<E>();
        for(int i=0;i<s1. getsize();i++)                //将集合 s1 中的所有元素复制到 s3 中
            s3. add(s1. get(i));
        for(int i=0;i<s2. getsize();i++)                //将 s2 中不在 s1 中出现的元素复制到 s3 中
            if(! s1. IsIn(s2. get(i)))
                    s3. add(s2. get(i));
        return s3;                                      //返回 s3
    }
    public Set<E> Intersection(Set<E>s1, Set<E>s2)      //求  s3=s1∩s2
    {Set<E>s3=new Set<E>();
        for(int i=0;i<s1. getsize();i++)                //将 s1 中出现在 s2 中的元素复制到 s3 中
            if(s2. IsIn(s1. get(i)))
                s3. add(s1. get(i));
        return s3;                                      //返回 s3
    }
    public Set<E>Difference(Set<E>s1, Set<E>s2)         //求 s3=s1-s2
    {Set<E>s3=new Set<E>();
        for(int i=0;i<s1. getsize();i++)                //将 s1 中不在 s2 中的元素复制到 s3 中
            if(! s2. IsIn(s1. get(i)))
                s3. add(s1. get(i));
        return s3;                                      //返回 s3
    }
```

□　设计主函数

在所有基本运算设计好之后，为了求两个集合{1,4,2,6,8}和{2,5,3,6}的并集、交集和差集，设计包含主函数的类如下：

```
public class Exam1_13
{public static void main(String[ ] args)
    {Set<Integer>s1. s2,s3,s4. s5;                    //建立 Set<Integer>的 5 个对象
        TwoSet<Integer>t=new TwoSet<Integer>( );      //建立 TwoSet<Integer>的 1 个对象
        s1=new Set<Integer>( );
        s1. add(1);
        s1. add(4);
        s1. add(2);
        s1. add(6);
        s1. add(8);
        System. out. print("集合 s1:");s1. display( );
        s2=new Set<Integer>( );
        s2. add(2);
        s2. add(5);
        s2. add(3);
        s2. add(6);
        System. out. print("集合 s2:");s2. display( );
        System. out. println("集合 s1 和 s2 的并集->s3");
        s3=t. Union(s1, s2);
        System. out. print("集合 s3:");s3. display( );
        System. out. println("集合 s1 和 s2 的差集->s4");
        s4=t. Difference(s1, s2);
        System. out. print("集合 s4:");s4. display( );
        System. out. println("集合 s1 和 s2 的交集->s5");
        s5=t. Intersection(s1, s2);
        System. out. print("集合 s5:");s5. display( );
    }
}
```

□　执行结果

上述程序的执行结果如下：

```
集合 s1:1 4 2 6 8
集合 s2:2 5 3 6
集合 s1 和 s2 的并集->s3
集合 s3:1 4 2 6 8 5 3
集合 s1 和 s2 的差集->s4
集合 s4:1 4 8
集合 s1 和 s2 的交集->s5
集合 s5:2 6
```

本章小结

本章主要介绍了数据结构的一些基本概念和算法的描述及分析方法。数据结构是指数据及其之间的相互关系，可以看作是相互之间存在一种或多种特定关系的数据元素的集合。因此，可以把数据结构看成带结构的数据元素的集合。数据结构包括：数据元素之间的逻辑关系，即数据的逻辑结构；数据元素及其关系在计算机存储中的存储方式，即数据的存储结构。数据的逻辑结构主要分为集合、线性结构、树形结构、图形结构。数据的存储结构包括顺序存储结构、链式存储结构、索引存储结构和哈希存储结构。

算法是对特定问题求解步骤的一种描述，它是指令的有限序列，其中，每条指令表示一个或多个操作；一个算法具有以下五个重要的特性：有穷性、确定性、可行性、输入和输出。一个算法用高级语言实现以后，在计算机上运行时所消耗的时间与很多因素有关：依据算法所选择的具体策略、问题的规模、书写程序的语言、机器代码的质量、机器执行指令的速度。可以认为一个特定算法的“执行工作量”只依赖于问题的规模。从时间和空间两方面来衡量一个算法效率，一个算法的执行时间可以看成基本运算执行的次数，一个算法的空间复杂度是指算法运行所需的存储空间。

在掌握以上概念的同时，读者应当重点掌握从时间和空间两方面评价一个算法好坏的方法，以便在自己设计一个算法时，能够达到效率上的最优化。

本章实训

皇后问题

一、问题描述

在一个 8×8 的棋盘里放置 8 个皇后，要求每个皇后两两之间不冲突，即在每一横列、竖列和斜列只有一个皇后。

二、提示与分析

数据表示：用一个 8 位的八进制数表示棋盘上皇后的位置。

例如 45615353 表示：

第 0 列皇后在第 4 个位置

第 1 列皇后在第 5 个位置

第 2 列皇后在第 6 个位置

⋮

第 7 列皇后在第 3 个位置

课后练习

1. 什么是数据结构？有关数据结构的讨论涉及哪三个方面？
2. 简述逻辑结构与存储结构的关系。
3. 简述数据结构中运算描述和运算实现的异同。
4. 简述数据结构和数据类型两个概念之间的区别。
5. 什么是算法？算法的五个特征是什么？
6. 存储结构对算法的影响主要是什么？

第 2 章　线　性　表

学习目标

◎知识目标

1. 了解线性表的基本概念及抽象数据类型的描述。
2. 掌握线性表的顺序存储和实现。
3. 掌握线性表的链式存储和实现。
4. 掌握广义表的概念及广义表的存储。
5. 了解顺序表与链表的优、缺点。

◎能力目标

1. 能够掌握基本的存储方式和运算。
2. 能够在实际问题中选择和应用线性表。

◎思政目标

通过学习线性表，学生可以更深入地了解计算机科学的基本原理和方法，从而增强计算机科学素养。

2.1　线性表及其基本操作

2.1.1　线性表的基本概念

线性表(linear list)是其组成元素间具有线性关系的一种线性结构，是由 n 个具有相同数据类型的数据元素 a_0、a_1、…、a_{n-1} 构成的有限序列，一般表示为

$$(a_0, a_1, \cdots a_i, a_{i+1}, \cdots, a_{n-1})$$

式中，数据元素 a_i 可以是字母、整数、浮点数、对象或其他更复杂的信息；i 代表数据元素在线性表中的位序号（$0 \leqslant i < n$）；n 是线性表的元素个数，称为线性表的长度，当 $n=0$ 时，线性表为空表。

例如：英文字母表（A，B，…，Z）是一个表长为 26 的线性表。又如见表 2-1 所列的书籍信息表，这个信息表中的所有记录序列构成了一个线性表，线性表中的每个数据元素都是由书名、作者、出版社、价格 4 个数据项构成的记录。

表 2-1　书籍信息表

书名	作者	出版社	价格
软件工程实用教程	吕云翔	清华大学出版社	49.00
…	…	…	…

线性表中的数据元素具有线性的“一对一”的逻辑关系，是与位置有关的，即第 i 个元素 a_i 处于第 $i-1$ 个元素 a_{i-1} 的后面和第 $i+1$ 个元素 a_{i+1} 的前面。这种位置上的有序性就是一种线性关系，可以用二元组表示为 $L=(D,R)$，其中有以下关系：

$$D=\{a_i \mid 0 \leqslant i < n\}$$

$$R=\{r\}$$

$$r=\{<a_i,a_{i+1}> \mid 0 \leqslant i < n-1\}$$

对应的逻辑结构如图 2-1 所示。

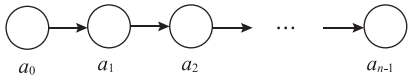


图 2-1　线性表的逻辑结构图

在线性表（$a_0,a_1,\cdots,a_{n-1}$）中，a_0 为开始节点，没有前驱元素；a_{n-1} 为终端节点，没有后继元素。除开始节点和终端节点外，每个数据元素 a_i 都有且仅有一个前驱元素和后继元素。

2.1.2　抽象数据类型描述

线性表的抽象数据 Java 接口描述如下：

```
1public interface IList{
2public void clear();                          //将线性表置为空表
3public boolean isEmpty();                     //判断线性表是否为空表
4public int length();                          //返回线性表的长度
5public Object get(int i);                     //读取并返回线性表中的第 i 个数据元素
6public void insert(int i,Object x);           //插入 x 作为第 i 个元素
7public void remove(int i);                    //删除第 i 个元素
8public int indexOf(Object x);                 //返回元素 x 首次出现的位序号
9public void display();                        //输出线性表中各个数据元素的值
10 }
```

【例 2-1】　有线性表 $A=(1,2,3,4,5,6,7)$，求 length()、isEmpty()、get(3)、indexOf(4)、display()、insert(2,7) 和 remove(4) 等基本运算的执行结果。

解

```
length()=7;
isEmpty()返回 false;
get(3)返回 4;
indexOf(4)返回 3;
```

display( )输出 1,2,3,4,5,6,7;
insert(2,7)执行后线性表 A 变为 1,2,7,3,4,5,6,7;
remove(4)执行后线性表 A 变为 1,2,3,4,6,7。

2.2　线性表的顺序存储和实现

线性表中基本运算的具体实现依赖于所采用的对数据的存储方式。本节将详细讨论线性表的顺序存储与其基本运算的实现。

2.2.1　顺序表

线性表的顺序存储是指在内存中用地址连续的一块存储空间顺序存放线性表的各元素,采用这种存储形式的线性表称为顺序表。因为内存中的地址空间是线性的,故而用物理上的相邻实现数据元素之间的逻辑相邻关系,既简单又自然。线性表的顺序存储如图 2-2 所示。

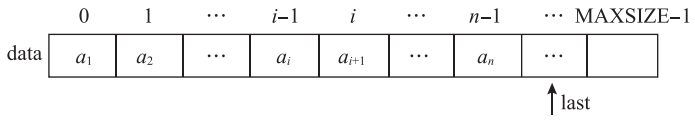


图 2-2　线性表的顺序存储

设 a_1 的存储地址为 Loc(a_1),每个数据元素占 d 个存储单元,则第 i 个数据元素的地址为

$$Loc(a_i) = Loc(a_1) + (i-1) \times d \qquad (1 \leqslant i \leqslant n) \qquad (2-1)$$

这就是说,只要知道顺序表的首地址和每个数据元素所占地址单元的个数就可求出第 i 个数据元素的地址,即顺序表具有按数据元素的序号存取的特点。

在程序设计语言中,一维数组在内存中占用的存储空间就是一组连续的存储区域,因此,用一维数组来表示顺序表的数据存储区域是再合适不过的。考虑到线性表的运算有插入、删除等,即表长是可变的,因此,数组的容量需设计得足够大。设用 data[MAXSIZE]来表示数组,其中,MAXSIZE 是一个根据实际问题定义的足够大的整数,线性表中的数据从 data[0]开始依次存放,但当前线性表中的实际元素个数可能未达到 MAXSIZE 个,因此需用一个变量 last 来记录当前线性表中最后一个元素在数组中的位置,即 last 起一个指示作用,始终指向线性表中最后一个元素的位置,因此,表空时 last=-1。这种存储思想的具体描述可以是多样的。例如,可以是

```
DataType[ ]data;
int last;
```

这样表示的顺序表如图 2-2 所示。表长为 last+1,第 1 个到第 n 个数据元素分别存放在 data[0]到 data[last]中。这样使用简单方便,但有时不便管理,信息的隐蔽性不好。在 Java 语言中可以定义一个顺序表类 SeqList,将数据存储区 data、位置 last 与顺序表中的基本运算封装在一起,作为对抽象数据类型接口 IList 的实现。

```
class SeqList implements IList{
    private static final int MAXSIZE=100;          //定义数组的最大容量
    private int last;                              //用于存储顺序表最后一个元素的存储位置
    private DataType[ ]data;                       //顺序表的存储空间
    SeqList( ){//构造函数,调用顺序表初始函数,建立存储空间为 MAXSIZE 的空表
        init( );
```

```
    };
    @Override
    public void init(){…}
    @Override
    public int length(){…}                          //求顺序表的长度
    @Override
    public DataType get(int i){…}                   //获取顺序表第 i 个元素
    @Override
    public int locate (DataType x)  {…}             //在顺序表中查找元素 x
    @Override
    public int insert (int i,DataType x){…}         //将元素 x 插入顺序表中作为第 i 个元素
    @Override
    public int delete (int i){…}                    //删除顺序表中的第 i 个元素
    …                                               //其他成员函数

}
```

在如图 2-3 所示的顺序表中，L 是一个引用类型的变量，是顺序表类 SeqList 的一个实例，通过“SeqListL=newSeqList();”操作来获得顺序表的存储空间，L 代表着一个具体的顺序表。

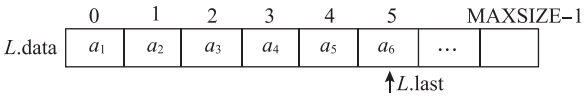


图 2-3　顺序表的存储表示方式

线性表的表长表示为 $L.last+1$。

线性表中数据元素顺序存储的基址为 $L.data$。

线性表中数据元素的存储或表示为 $L.data[0]\sim L.data[L.last]$。

2.2.2　顺序表上基本运算的实现

1. 顺序表的初始化

顺序表的初始化即构造一个空表，首先动态分配存储空间，然后将 last 置为-1，表示表中没有数据元素。

【算法 2-1】　顺序表的初始化。

```
public void init (){
    data=new DataType[MAXSIZE];
    last=-1;
}
```

2. 求顺序表的长度

由于在顺序表类的定义中，last 代表顺序表中最后一个元素的存储下标，而 Java 语言数组的下标是从 0 开始的，因此顺序表中的元素个数为 last+1。

【算法 2-2】　求顺序表的长度。

```
public int length(){                                //求顺序表的长度
    return last+1;
}
```

3. 插入运算

线性表的插入是指在表的第 i 个位置上插入一个值为 x 的新元素，插入后使原表长 n 变为 $n+1$。

插入前：$(a_1,a_2,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n)$。

插入后：$(a_1,a_2,\cdots,a_{i-1},x,a_i,a_{i+1},\cdots,a_n)$，其中 $1\leqslant i\leqslant n+1$。

插入过程如图 2-4 所示。

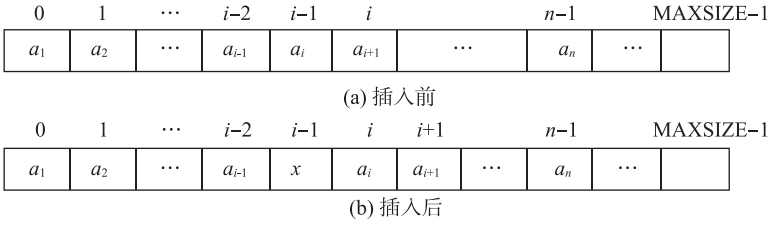


图 2-4　顺序表中的插入过程

顺序表上完成这一运算需通过以下几个步骤进行。

(1)将 $a_i\sim a_n$ 顺序向后移动，为新元素让出位置。

(2)将 x 置入空出的第 i 个位置。

(3)修改 last 的位置(相当于修改表长)，使之仍指向最后一个元素。

【算法 2-3】　顺序表的插入运算。

```
public boolean insert(int i,DataType x)throws Exception{   //将元素 x 插入顺序表中作为第 i 个元素
    int j;
    if (last==MAXSIZE-1)                                   //表空间已满,不能插入
    {
        throw new Exception("表空间已满");
    }
    if(i<1||i>last+2)                                      //检查插入位置的正确性
    {
        throw new Exception("插入位置错");
    }
    for(j=last;j>=i-1;j--)
        data[j+1]=data[j];                                 //节点移动
    data[i-1]=x;                                           //新元素插入
    last++;                                                //last 仍指向最后元素
    return true;                                           //插入成功,返回
}
```

性能分析：顺序表的插入运算，时间主要消耗在数据的移动上，在第 i 个位置上插入 x，从 $a_i\sim a_n$ 都要向后移动一个位置，共需要移动 $n-i+1$ 个元素，i 的取值范围为 $1\leqslant i\leqslant n+1$，即有 $n+1$ 个位置可以插入。设在第 i 个位置上做插入的概率为 p_i，则平均移动数据元素的次数为

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

设 $p_i=\dfrac{1}{n+1}$，即为在等概率的情况下，则

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

这说明顺序表的插入运算需移动表中一半的数据元素，算法的时间复杂度为 $O(n)$。

本算法应注意以下几个问题。

(1)顺序表中数据区域有 MAXSIZE 个存储单元，所以在向顺序表中做插入时先检查表空间是否满了，在表满的情况下不能再做插入，否则将产生溢出错误。

(2)要检验插入位置的有效性，这里 i 的有效范围是 $1 \leqslant i \leqslant n+1$，其中 n 为原表长。

(3)注意数据的移动方向。

4. 删除运算

线性表的删除运算是指将表中第 i 个元素从线性表中去掉，删除后使原表长 n 变为 $n-1$。

删除前：$(a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$。

删除后：$(a_1, a_2, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n)$，其中 $1 \leqslant i \leqslant n$。

删除过程如图 2-5 所示。

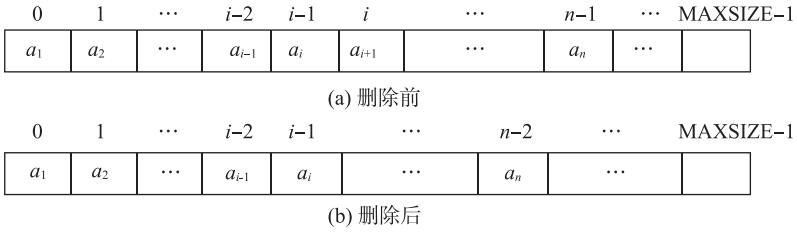


图 2-5　顺序表的删除过程

顺序表的删除运算需要通过以下几个步骤。

(1)将 $a_{i+1} \sim a_n$ 顺序向前移动。

(2)修改 last 的位置(相当于修改表长)，使之仍指向最后一个元素。

【算法 2-4】　顺序表的删除运算。

```
public boolean delete (int i)throws Exception{          //删除顺序表中的第 i 个元素
    int j;
    if(i<1||i>last+1)                                   //检查是否为空表及删除位置的合法性
    {
            throw new Exception("不存在第 i 个元素");
    }
    for(j=i;j<=last;j++)
        data[j-1]=data[j];                              //向上移动
    last--;
    return true;                                        //删除成功
}
```

性能分析：与插入运算相同，删除运算的时间主要消耗在移动表中元素上，删除第 i 个元素时，其后的元素 $a_{i+1} \sim a_n$ 都要向前移动一个位置，共移动了 $n-i$ 个元素，所以平均移动数据元素的次数为

$$E_{de} = \sum_{i=1}^{n} p_i(n - i) \tag{2-4}$$

在等概率情况下，$p_i = \frac{1}{n}$，则

$$E_{de} = \sum_{i=1}^{n} p_i(n - 1) = \frac{1}{n}\sum_{i=1}^{n+1}(n - i) = \frac{n - 1}{2} \tag{2-5}$$

这说明在顺序表上做删除运算时大约需要移动表中一半的元素,显然该算法的时间复杂度为 $O(n)$。

删除算法中需注意以下几个问题。

(1)删除第 i 个元素,i 的取值为 $1 \leqslant i \leqslant n$,否则第 i 个元素不存在,因此,要检查删除位置的有效性。

(2)当表空时不能做删除运算,因表空时 last 的值为-1,条件(i<<1||i>last+1)也包括了对表空的检查。

(3)删除 a_i 之后,该数据就不存在了,如果需要此数据,先取出 a_i,再做删除运算。

5. 按值查找

线性表中的按值查找是指在线性表中查找与给定值 x 相等的数据元素。在顺序表中完成该运算最简单的方法是:从第一个元素 a_1 起依次和 x 做比较,直到找到一个与 x 相等的数据元素,返回它在顺序表中的存储下标或序号(二者差 1,即序号=下标+1);若查遍整个表都没有找到与 x 相等的元素,则返回-1。

【算法 2-5】 顺序表的查找运算。

```
public int locate (DataType x) {                    //在顺序表中查找元素 x
    int i=0;
    while(i<=last&&data[i].equals (x))
        i++;
    if(i>last)
        return-1;                                   //查找失败,返回-1
    else
        return i;                                   //查找成功,返回的是存储位置
}
```

查找算法主要是给定值 x 与表中元素做比较。显然比较的次数与 x 在表中的位置有关,也与表长有关。当 $a_1=x$ 时,比较一次即成功;当 $a_n=x$ 时,比较 n 次才成功。在等概率情况下,查找成功的平均比较次数为

$$E_{lo}=\frac{1}{n}\sum_{i=1}^{n} i=\frac{n+1}{2} \tag{2-6}$$

在查找失败的情况下,需要比较 n 次。显然,按值查找算法的时间性复杂度为 $O(n)$。

2.2.3 顺序表应用举例

【例 2-1】 将顺序表$(a_1,a_2,\cdots,a_n)$重新排列为以 a_1 为界的两部分:a_1 前面的值均比 a_1 小,a_1 后面的值都比 a_1 大(这里假设数据元素的类型具有可比性,不妨设为整型)。顺序表划分前、后如图 2-6 所示。这一操作称为划分,a_1 称为基准。

25	30	20	60	10	35	15

(a) 划分前

15	10	20	25	30	60	35

(b) 划分后

图 2-6 顺序表的划分

划分的方法有多种,下面介绍的划分算法思路简单,但性能较差。

基本思路:从第二个元素开始到最后一个元素逐一向后扫描。

(1)当前数据元素 a_i 比 a_1 大时,表明它已经在 a_1 的后面,不必改变它与 a_1 之间的位置,需继续比较下一个。

(2)若前数据元素比 a_1 小,说明它应该在 a_1 的前面,此时先将它上面的元素都依次向后移动一个位置,然后将它置入最前方。

【算法 2-6】 划分算法。

```
public void part( )  {
        int i,j;
        DataType x,y;
        x=data[0];                                  //将基准置入 x 中
        for(i=1;i<=last;i++)
            if (data[i]. compareTo (x)<0)           //当前元素小于基准
            {
                y=data[i];
                for(j=i-1;j>=0;j--)                 //移动
                {
                    data[j+1]=data[j];
                    data[0]=y;
                }
            }
}
```

本算法中有两重循环，外循环执行 $n-1$ 次，内循环中移动元素的次数与当前数据的大小有关，当第 i 个元素小于 a_1 时，要移动它前面的 $i-1$ 个元素，再加上当前节点的保存及置入，所以移动 $i-1+2$ 次。在最坏情况下，a_1 后面的节点都小于 a_1，故总的移动次数为

$$\sum_{i=2}^{n}(i-1+2)=\sum_{i=2}^{n}(i+1)=\frac{(n-1)\times(n+4)}{2} \tag{2-7}$$

也就是说，最坏情况下移动数据的时间复杂度为 $O(n^2)$。

【例 2-2】 有顺序表 A 和 B，其元素均按从小到大的升序排列，编写一个算法将它们合并成一个新的顺序表 C，要求 C 的元素也从小到大按升序排列。

算法思路：依次扫描顺序表 A 和 B 的元素，比较当前元素的值，将较小值的元素赋给 C，如此直到一个线性表扫描完毕，并将未扫描完的那个顺序表中的余下部分赋给 C 即可。C 的大小要能够容纳 A、B 两个线性表相加的长度。

【算法 2-7】 顺序表的合并。

```
public static void merge(SeqList A,SeqList B,SeqList C)throws Exception  {
        int i,j;
        i=j=0;
        while(i<A. length( )&&j<B. length( )){          //将 A 和 B 的当前元素较小者复制到 C
            DataType oa=A. get (i),ob=B. get(j);
            if(oa. compareTo (ob)<0) {
                C. insert(C. length( )+1,oa);
                i++;
            }else {
                C. insert(C. length ( )+1,ob);
                j++;
            }
```

```
        }
        while{i<A. length( ) ){                          //将 A 中剩余元素复制到表 C
            C. insert (C. length( )+1,A.  get(i) );
            i++;
        }
        while(j<B. length( ) ){                          //将 B 中剩余元素复制到表 C
            C. insert( C. length( )+1,B. get(j) );
            j++;
        }
    }
```

该算法的时间复杂度是 $O(m+n)$,其中 m 是 A 的表长,n 是 B 的表长。

2.3　线性表的链式存储和实现

采用链式存储方式存储的线性表称为链表,链表是用若干地址分散的存储单元存储数据元素,逻辑上相邻的数据元素在物理位置上不一定相邻,必须采用附加信息表示数据元素之间的逻辑关系,因此链表的每一个节点不仅包含元素本身的信息-数据域,而且包含元素之间逻辑关系的信息,即逻辑上相邻节点地址的指针域。

2.3.1　单链表

单链表是指节点中只包含一个指针域的链表,指针域中储存着指向后继节点的指针。单链表的头指针是线性表的起始地址,是线性表中第一个数据元素的存储地址,可作为单链表的唯一标识。单链表的尾节点没有后继节点,所以其指针域值为 null。

为了使操作简便,在第一个节点之前增加头节点,单链表的头指针指向头节点,头节点的数据域不存放任何数据,指针域存放指向第一个节点的指针。空单链表的头指针 head 为 null。图 2-7 为不带头节点的单链表的存储示意图,图 2-8 为带头节点的单链表的存储示意图。

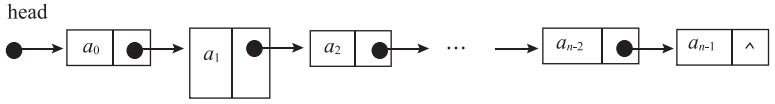


图 2-7　不带头节点的单链表

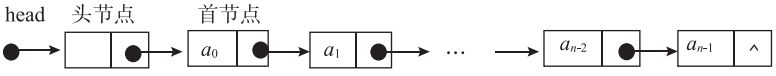


图 2-8　带头节点的单链表

单链表的节点的存储空间是在插入和删除过程中动态申请和释放的,不需要预先分配,从而避免了顺序表因存储空间不足需要扩充空间和复制元素的过程,避免了顺序表因容量过大造成内存资源浪费的问题,提高了运行效率和存储空间的利用率。

1. 节点类描述

```
1package ch02;
2public class Node{
3public Object data;                              //存放节点数据值
```

```
4public Node next;                          //存放后继节点
5                                           //无参构造函数
6public Node( ){
7this(null,null);
8   }
9                                           //只有节点值的构造函数
10public Node(Object data){
11this(data,null);
12   }
13                                          //带有节点值和后继节点的构造函数
14public Node(Object data,Node next){
15this. data=data;
16this. next=next;
17 }
18 }
```

2. 单链表类描述

```
1package ch02;
2public class LinkList implements IList{
3public Node head;                          //单链表的头指针
4                                           //构造函数初始化头节点
5public LinkList( ){
6head=new Node( );
7   }
8                                           //构造函数构造长度为 n 的单链表
9public LinkList(int n,boolean Order){
10this( );
11if(Order)
12create1(n);
13else
14create2(n);
15   }
16                                          //用尾插法顺序建立单链表
17public void create1(int n){
18
19   }
20                                          //用头插法逆序建立单链表
21public void create2(int n){
22
23   }
24                                          //将链表置空
25public void clear( ){
```

```
26head. data=null;
27head. next=null;
28   }
29                                                    //判断链表是否为空
30public boolean isEmpty( ){
31return head.  next= =null;
32   }
33                                                    //返回链表长度
34public int length( ){
35Node p=head.  next;
36int length=0;
37while(p!  =null){
38p=p. next;
39length++;
40       }
41return length;
42   }
43                                                    //读取并返回第 i 个位置的数据元素
44public Object get(int i)throws Exception{
45   }
46                                                    //插入 x 作为第 i 个元素
47public void insert(int i,Object x)throws Exception{
48
49 }
50                                                    //删除第 i 个元素
51public void remove(int i)throws Exception{
52
53   }
54                                                    //返回元素 x 首次出现的位序号
55public int indexOf(Object x){
56
57   }
58public void display( ){
59Node p=head. next;
60while(p!  = null){
61System. out. print(p. data+"");
62p=p. next;
63       }
64
65 }
66 }
```

2.3.2 单链表的基本操作实现

1. 查找操作

(1)位序查找 get(i)是返回长度为 n 的单链表中第 i 个节点的数据域的值,其中 $0 \leqslant i \leqslant n-1$。由于单链表的存储空间不连续,因此必须从头节点开始沿着后继节点依次进行查找。

【算法 2.4】 位序查找算法。

```
1public Object get(int i)throws Exception {
2Node p=head. next;                    //p 指向单链表的首节点
3int j;
4                                      //从首节点开始向后查找,直到 p 指向第 i 个节点或者 p 为 null
5for(j=0;j<i&&p! =null;j++){
6p=p. next;
7      }
8if(j>i||p==null)                      //i 不合法时抛出异常
9throw new Exception("第"+i+"个数据元素不存在");
10return p. data;
11   }
```

(2)查找操作 indexOf(x)是在长度为 n 的单链表中寻找初次出现的数据域值为 x 的数据元素的位置。

其主要步骤为将 x 与单链表中的每一个数据元素的数据域进行比较,若相等,则返回该数据元素在单链表中的位置;若比较结束未找到等值的数据元素,返回-1。

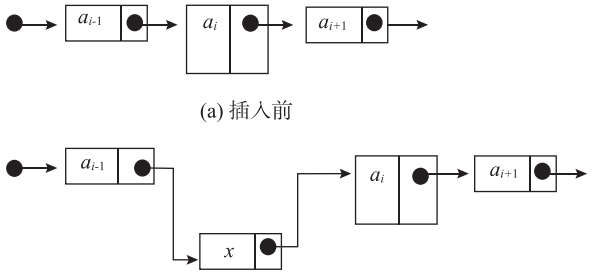
【算法 2-8】 按值查找。

```
1public int indexOf(Object x) {
2Node p=head. next;
3int j=0;
4while(p! =null&&! p. data. equals(x)){
5p=p. next;
6j++;
7      }
8if(p! =null)
9return j;
10else
11return-1;
12   }
```

2. 插入操作

插入操作 insert(i,x)是在长度为 n 的单链表的第 i 个节点之前插入数据域值为 x 的新节点,其中,$0 \leqslant i \leqslant n$,当 $i=0$ 时在表头插入,当 $i=n$ 时在表尾插入。

与顺序表相比,单链表不需要移动一批数据元素,而只需要改变节点的指针域,改变有序对,即可实现数据元素的插入,即<a_{i-1},a_i>转变为<a_{i-1},x>和<x,a_i>,如图 2-9 所示。



(a) 插入前

(b) 插入后

图 2-9 单链表上的插入

插入操作的主要步骤如下：

(1)查找到插入位置的前驱节点，即第 $i-1$ 个节点。

(2)创建数据域值为 x 的新节点。

(3)修改前驱节点的指针域为指向新节点的指针，新节点的指针域为指向原第 i 个节点的指针。

【算法 2-9】 带头节点的单链表的插入操作。

```
1public void insert(int i,Object x)throws Exception{
2Node p=head;
3int j=-1;
4                                          //寻找第 i 个节点的前驱
5while(p! =null&&j<i-1){
6p=p. next;
7j++;
8       }
9if(j>i-1||p==null)                        //i 不合法时抛出异常
10throw new Exception("插入位置不合法");
11Node s=new Node(x);
12s. next=p. next;
13p. next=s;
14 }
```

【算法 2-10】 不带头节点的单链表的插入操作。

```
1public voidinsert(int i,Object x)throws Exception{
2Node p=head;
3int j=0;
4                                          //寻找第 i 个节点的前驱
5while(p! =null&&j<i-1){
6p=p. next;
7j++;
8       }
9if(j>i-1||p==null)                        //i 不合法时抛出异常
10throw new Exception("插入位置不合法");
11Node s=new Node(x);
12if(i==0){
13s. next=head;
14head=s;
15       }
16else{
17s. next=p. next;
18p. next=s;
19       }
20   }
```

分析以上代码可以发现，由于链式存储采用的是动态存储分配空间，所以在进行插入操作之前不需要判断存储空间是否已满。

在带头节点的单链表上进行插入操作时，无论插入位置是表头、表尾还是表中，操作语句都是一致的；但是在不带头节点的单链表上进行插入操作时，在表头插入和在其他位置插入新节点的语句是不同的，需要分两种情况进行处理。本章之后的例题代码均是基于带头节点的单链表类实现。

3. 删除操作

删除操作 remove(i,x)是将长度为 n 的单链表的第 i 个节点删除，其中 $0 \leqslant i \leqslant n-1$。

与顺序表相比，单链表不需要移动一批数据元素，而只需要改变节点的指针域，实现有序对的改变，即可删除节点，即$<a_{i-1},a_i>$和$<a_i,a_{i+1}>$转变为$<a_{i-1},a_{i+1}>$，如图 2-10 所示。

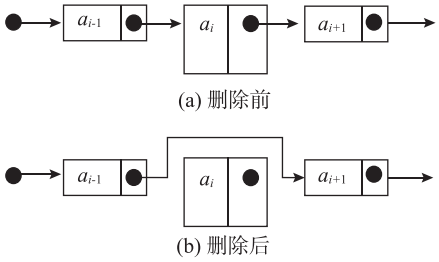


图 2-10　单链表的删除操作

其主要步骤如下。

(1)判断单链表是否为空。

(2)查找待删除节点的前驱节点。

(3)修改前驱节点的指针域为待删除节点的指针域。

【算法 2-11】　删除操作。

```
1public void remove(int i)throws Exception{
2Node p=head;
3int j=-1;
4                                          //寻找第 i 个节点的前驱节点
5while(p!  =null&&j<i-1){
6p=p. next;
7j++;
8       }
9if(j>i-1||p. next==null)                  //i 不合法时抛出异常
10throw new Exception("删除位置不合法");
11p. next=p. next. next;
12
13 }
```

4. 单链表的建立操作

(1)头插法

将新节点插入单链表的表头，读入的数据顺序与节点顺序相反。

【算法 2-12】　头插法。

```
1public void create1(int n)throws Exception{
2Scanner sc=new Scanner(System. in);
3for(int i=0;i<n;i++){
4insert(0,sc. next());
5       }
6  }
```

(2)尾插法

将新节点插入单链表的表尾，读入的数据顺序与节点顺序相同。

【算法 2-13】 尾插法。

```
1public void create2(int n)throws Exception{
2Scanner sc=new Scanner(System.in);
3for(int i=0;i<n;i++){
4insert(length(),sc.next());
5       }
6   }
```

【例 2-14】 编程实现将数组中的元素构建成一个有序的单链表。

```
public static void main(String[]args){
        Scanner sc=new Scanner(System.in);                    //构造用于输入的对象
        int n,val;
        n=sc.nextInt();                                       //n 为需要输入的数组元素的个数
        LinkList l=new LinkList();                            //声明单链表
        for(int i=0;i<n;i++){
            Node q=l.head;
            val=sc.nextInt();
            Node p=new Node(val);
while(q!=null&&q.next!=null&&Integer.valueOf(q.next.data.toString())<val){
            q=q.next;
        }
        p.next=q.next;                                        //进行插入操作
        q.next=p;
    }
    Node k=l.head;
    for(int i=0;i<n;i++){                                     //输出有序的单链表
        k=k.next;
        System.out.println(k.data);
    }
}
```

2.3.3 其他链表

1. 循环链表

循环链表与单链表的结构相似,只是将链表的首尾相连,即尾节点的指针域为指向头节点的指针,从而形成一个环状的链表。

循环链表与单链表的操作算法基本一致,判定循环链表中的某个节点是否为尾节点的条件不是它的后继节点为空,而是它的后继节点是否为头节点。

在实现循环链表时可用头指针或尾指针或二者同时使用来标识循环链表,通常使用尾指针进行标识,可简化某些操作。

2. 双向链表

双向链表的节点具有两个指针域,一个指针指向前驱节点,一个指针指向后继节点。这样使得查找某个节点的前驱节点不需要从表头开始顺着链表依次进行查找,从而减小时间复杂度。

（1）节点类描述

```
1  package ch02;
2  public class DuLNode{
3  public Object data;                    //存放数据值的数据域
4  public DuLNode prior;                  //存放指向前驱节点的指针
5  public DuLNode next;                   //存放指向后继节点的指针
6  public DuLNode( ){
7          this(null);
8  }
9  public DuLNode(Object data){
10           this. data=data;
11           this. prior=null;
12           this. next=null;
13 }
14 }
```

（2）双向链表的基本操作实现

双向链表与单链表的不同之处主要在于进行插入和删除操作时每个节点需要修改两个指针域。

【算法 2-15】 插入操作。

```
1  public void insert(int i,Object x)throws Exception{
2          DuLNode p=head;
3          int j=-1;
4                                          //寻找插入位置 i
5          while(p! =null&&j<i){
6                p=p. next;
7                j++;
8          }
9          if(j>i||p==null)                //i 不合法时抛出异常
10           throw new Exception("插入位置不合法");
11         DuLNode s=new DuLNode(x);
12         p. prior. next=s;
13         s. next=p;
14         s. prior=p. prior;
15         p. prior=s;
16 }
```

【算法 2-16】 删除操作。

```
1  public void remove(int i)throws Exception{
2          DuLNode p=head;
3          int j=-1;
4          while(p! =null&&j<i){
5                p=p. next;
6                j++;
```

```
7          }
8          if(j>i||p==null)                    //i 不合法时抛出异常
9              throw new Exception("删除位置不合法");
10         p.prior.next=p.next;
11         p.next.prior=p.prior;
12     }
```

2.3.4　链表应用举例

【例 2-3】　已知单链表 H 如图 2-11(a)所示，编写一算法将其逆置，即实现图 2-11(b)所示的操作。

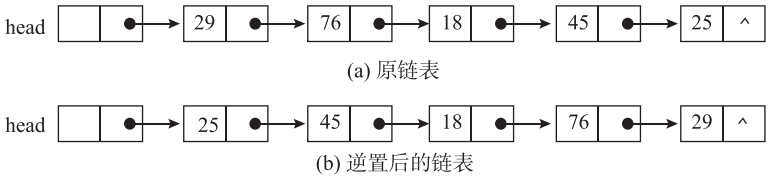


图 2-11　单链表的逆置

算法思路：依次取原链表中的每个节点，将其作为第一个节点插入新链表中，引用变量 p 用来指向原表中当前节点。p 为空时结束。

【算法 2-17】　单链表逆置。

```
public void reverse ( ) {
        LinkNode p,q;
        p=head.getNext( );                          //指向第一个数据节点
        head.setNext(null);;                        //将原链表置为空表
        while(p!=null) {
            q=p;
            p=p.getNext( );
            q.setNext(head.getNext( ));             //将当前节点插入头节点的后面
            head.setNext(q);
        }
}
```

该算法只是对链表顺序扫描一遍即完成了逆置，所以时间复杂度为 $O(n)$。

【例 2-4】　已知单链表 L 如图 2-12(a)所示，编写一个算法，删除其重复节点，即实现如图 2-23(b)所示的操作。

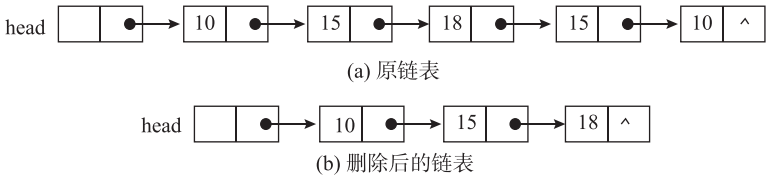


图 2-12　删除重复节点

算法思路：将引用变量 p 指向第一个数据节点，从它的后继节点开始到表结束，找与其值相同的节点并删除之，p 指向最后节点时算法结束。

【算法 2-18】 在单链表中删除重复节点。

```
public void pure( ) {
    LinkNode p, q;
    p=head. getNext( );                                    //p 指向第一个节点
    if(p==null)
        return;
    while (p. getNext( )! =null) {
        q=p;
        while (q. getNext( )! =null)                       //从 p 的后继开始找重复节点
        {
            if(q. getNext( ). getData( ). equals (p. getData( )))
                q. setNext (q. getNext( ). getNext( ));
                                                           //找到重复节点,删除 q. next 所指的节点
            else
                q=q. getNext( );
        }       //while(q. next)
        p=p. getNext( );                                   //p 指向下一个节点,继续
    }
}
```

该算法的时间复杂度为 $O(n^2)$。

【例 2-5】 设有两个单链表 A、B,其中,元素递增有序,编写算法将单链表 A、B 合并成一个按元素值递减(允许有相同值)有序的链表 C。要求用单链表 A、B 中的原节点形成,不能重新申请节点。

算法思路:利用单链表 A、B 有序的特点,依次进行比较,将当前值较小者摘下,插入 C 表的头部,得到的 C 表则为递减有序的。

【算法 2-19】 单链表的合并。

```
public static LinkList merge (LinkList A,LinkList B) {
                                                           //设 A、B 均为带头节点的单链表
    LinkList C=new LinkList( );
    LinkNode p,q,s;
    p=A. getHead( ). getNext( );
    q=B. getHead( ). getNext( );
    while(p! =null && q! =null)  {
        if(p. getData( ). compareTo (q. getData( ))<0) {
            s=p;
            p=p. getNext ( );
        }else {
            s=q;
            q=q. getNext ( );
        }                                                  //从原 A、B 表上摘下较小者
        s. setNext(C. getHead( ). getNext( ));             //插入 C 表的头部,C 将是反序的
        C. getHead( ). setNext (s);
```

```
        }
        if(p==null)
            p=q;
        while(p!=null)                          //将剩余的节点一个个摘下,插入 C 表的头部
        {
            s=p;
            p=p.getNext();
            s.setNext(C.getHead().getNext());
            C.getHead().setNext(s);
        }
        return C;
    }
```

该算法的时间复杂度为 $O(m+n)$。

【例 2-6】　设计算法将采用单循环链表 L 存储的线性表,转换为用双向循环链表 H 存储。单循环链表 L 和双向循环链表 H 分别如图 2-13(a)和图 2-13(b)所示。

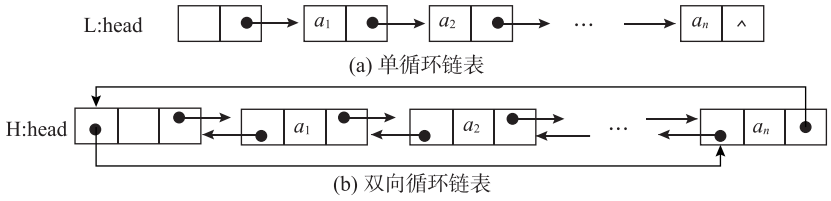


图 2-13　单循环链表和双向循环链表

算法思路:首先建立空的双向循环链表 H,然后从单循环链表 L 的第一个节点开始,逐个读取数据元素,生成双向链表的节点 s,再将 s 插入双向循环链表 H 中。

【算法 2-20】　将单循环链表转换为双向循环链表。

```
public static DLinkList Create_DlinkList (LinkList L){
    DLinkList H;
    DLinkNode s;
    LinkNode p;
    H=new DLinkList();                          //生成初始双向循环链表对象类 H
    p=L.getHead().getNext();                    //p 指向 L 的第一个元素
    while(p!=null) {
        s=new DLinkNode (p.getData());          //申请、填装节点 s
        s.setNext(H.getRear().getNext());
        s.setPrior(H.getRear());
        H.getRear().setNext (s);
        H.getHead().setPrior (s);               //将节点 s 插入 H 的链表尾
        H.setRear (s);                          //修改 H 的尾位置
        p=p.getNext();
    }
    return H;
}
```

该算法的时间复杂度为 $O(n)$。

2.4 顺序表与链表的比较

前面介绍了线性表的逻辑结构及它的两种存储结构：顺序表和链表。表 2-2 列出了用这两种存储方式实现线性表的优、缺点。

表 2-2 顺序表和链表的比较

存储结构	优点	缺点
顺序表	（1）方法简单，各种高级语言中都有数组类型，容易实现； （2）不用为表示节点间的逻辑关系而增加额外的存储开销。逻辑相邻与存储相邻一致； （3）顺序表具有按元素序号随机访问的特点	（1）顺序表在做插入、删除操作时，平均移动大约表中一半的元素，因此对于 n 较大的顺序表，插入、删除的效率低； （2）需要预先分配足够大的存储空间，如预先分配过大，可能会导致顺序表后部大量闲置；如预先分配过小，又会造成溢出
链表	（1）在链表中做插入、删除操作时，不需要多次移动元素，因此对于 n 较大的链表，插入和删除的效率高； （2）不需要预先分配足够大的存储空间，不会出现如下情况：预先分配过大，可能会导致链表后部大量闲置；预先分配过小，又可能会造成溢出	（1）需要为存储节点间的逻辑关系而增加额外的存储开销； （2）链表不具有按元素序号随机访问的特点

在实际中，怎样选取存储结构呢？通常有以下几方面的考虑。

（1）基于存储的考虑。顺序表的存储空间是静态分配的，在程序执行之前必须明确规定它的存储规模。也就是说，事先对“MAXSIZE”要有合适的设定，过大造成浪费，过小造成溢出。可见，对线性表的长度或存储规模难以估计时，不宜采用顺序表。链表不用事先估计存储规模，但链表的存储密度较低。存储密度是指一个节点中数据元素所占的存储单元和整个节点所占的存储单元之比。显然，链式存储结构的存储密度是小于 1 的。

（2）基于运算的考虑。在顺序表中按序号访问 a_i 的时间复杂度为 $O(1)$，而在链表中按序号访问 a_i 的时间复杂度为 $O(n)$。所以，如果经常做的运算是按序号访问数据元素，显然顺序表优于链表。而在顺序表中做插入、删除时平均移动表中一半的元素，当数据元素的信息量较大且表较长时，这一点是不应被忽视的。在链表中做插入和删除时，虽然也要找插入位置，但主要是比较操作，从这个角度考虑显然链表优于顺序表。

（3）基于环境的考虑。顺序表容易实现，任何高级语言中都有数组类型，而链表的操作是基于位置的，相对来讲，前者简单些，这也是用户考虑的一个因素。

总之，两种存储结构各有长短，选择哪一种由实际问题中的主要因素决定。通常“较稳定”的线性表选择顺序存储，而频繁进行插入和删除等“动态性”较强的线性表宜选择链式存储。

本章小结

本章首先介绍了线性表的逻辑结构，然后介绍了线性表的存储结构分为顺序存储结构和链式存储结构两种。线性表的顺序存储结构就是将线性表中的所有元素按照其逻辑结构顺序依次存储在计算机的一个