第1章　绪　论

一、单元概述

著名的瑞士计算机学家尼古拉斯·沃斯(Niklaus Wirth)曾用“程序＝数据结构＋算法”这样简单的公式,揭示了程序的本质。可见数据结构和算法在计算机相关理论和技术中的重要地位。

本教材介绍基本的数据结构及相关算法,这些是计算机及相关专业人员必须掌握的专业基础知识。本章作为全教材的绪论,介绍数据逻辑结构、数据存储结构以及算法的相关概念,并讲解评价算法时间效率、空间效率的方法。

二、教学重点与难点

重点:理解数据结构和算法的概念,以及本课程的研究内容;理解数据元素和数据项的概念,弄清二者之间差异;理解逻辑结构和存储结构的区别;理解4种逻辑结构;掌握算法时间复杂度和空间复杂度的分析方法。

难点:多重循环代码的时间复杂度分析。

1.1 数据、数据元素、数据项

本节我们对后续章节中经常出现的几个术语赋予明确的含义,以免出现混淆。

数据是对客观事物的符号表示,是指能够输入至计算机中并能够被计算机处理的符号的总称。例如,一个矩阵运算程序的处理对象是矩阵中的整数或实数;一个高级语言编译程序的处理对象是源文件中的字符串;一个人脸识别程序的处理对象是数字图像;一个语音识别程序的处理对象是数字音频。这些整数、实数、字符串、图像以及音频经过编码后,都可归为数据的范畴。

数据元素是数据结构课程中,数据处理的基本单位。一个数据元素可能体现为一个整数、一个实数这样的简单形式,也可能由若干**数据项**复合构成。例如,一个与学生信息管理相关的程序中,可能将一条学生基本信息的记录当作数据元素,此时学生基本信息中的每一项(如学号、姓名等)为一个数据项。后续章节在介绍数据结构和算法时,大部分情况下,数据处理涉及的基本单位是数据元素,而不是数据项。同时,在讨论数据结构特性或算法工作原理时,一般不考虑数据元素的具体数据类型,数据元素的类型采用抽象数据类型 ADT(Abstract Data Type)表示。例如,介绍冒泡排序算法时,我们只关心算法的工作原理,而不关心算法是在为一组整数排序,还是在为一组学生信息排序。

1.2 什么是数据结构

数据结构是指相互之间存在一种或多种关系的数据元素的集合和操作。它指的是数据元素之间的相互关系,即数据的组织形式。这种组织形式就是数据的逻辑结构。在计算机实际处理数据的过程中,我们必须考虑数据应以什么方式进行存储能使之体现数据之间的关系。数据在计算机中的存储方式,就是数据的存储结构。除此之外,在数据的处理过程中,还会出现数据的删除、插入、查找等操作,因此我们还应该考虑数据处理的方式,即算法。综上所述,按某种关系组织起来的一批数据,以一定的存储方式把它们存储到计算机的存储器中,并在这些数据上定义一个运算集合,这就是数据结构。

数据结构作为一门学科主要研究数据的各种逻辑结构和存储结构,以及对数据的各种操作。因此,主要有三个方面的内容:数据的逻辑结构;数据的物理存储结构;对数据的操作(算法)。通常,算法的设计取决于数据的逻辑结构,算法的实现取决于数据的物理存储结构。

1.2.1 数据的逻辑结构

数据的逻辑结构是从具体问题抽象出来的数学模型,逻辑结构描述数据元素之间的逻辑关系和操作,与数据的存储无关。根据数据元素之间关系的不同特性,通常有下列四类基本的结构:

1. 集合结构

结构中的数据元素之间没有关系，同属一个集合。集合结构往往可由有序的、无重复元素的线性结构或树结构代替，故本教材不做详细讨论，如图 1.1 所示。

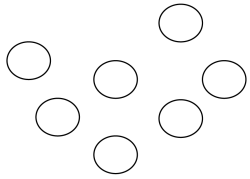

图 1.1 集合结构

2. 线性结构

在这种结构中，除第一个结点外，其他结点都有唯一一个直接前驱，除最后一个结点外，其他各结点有唯一的直接后继。数据元素之间是一对一的关系，如图 1.2 所示。

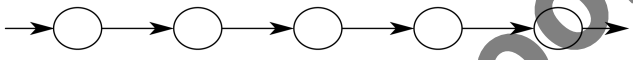

图 1.2 线性结构

例如，建立一张按学号排列的学生成绩表（包括学号、姓名和成绩），如下所示：

2017101	zhangmin	97.00
2017102	wangbin	77.00
2017103	liulu	82.00

3. 树状结构

在这种结构中，除了一个根结点外，各结点有唯一的前驱，但所有结点都可以有多个后继。数据元素之间是一对多的关系，如图 1.3 所示。

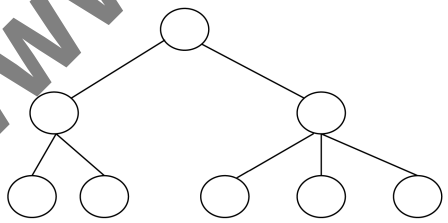

图 1.3 树状结构

例如，在对弈问题中，计算机操作的对象是对弈过程中可能出现的棋盘状态，这里称为格局。如图 1.4 所示，对弈过程中出现的格局之间的关系不是线性的，而是一棵“树”，这种数据结构称为树状数据结构。

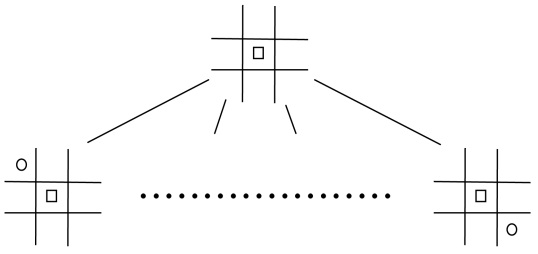

图 1.4 格局

4. 图状结构

在这种结构中,各结点可以有多个前驱或多个后继。数据元素之间是多对多的关系。如图 1.5 所示。

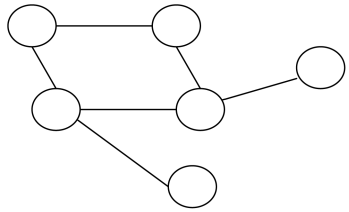

图 1.5 图状结构

例如,一组微信用户之间的“好友”关系,可以表示为图状结构。

1.2.2 数据的存储结构

逻辑结构在计算机中的实现称为数据的存储结构,简称为存储结构。存储结构包括数据元素本身的表示,也包括数据元素之间关系的表示。存储结构通常包括顺序和链式两种存储结构。顺序存储结构是借助数据元素在存储器中的相对位置来表示元素之间的逻辑关系,而链式存储结构是借助表示存储器地址的指针来表示数据元素之间的逻辑关系。在后续的章节介绍逻辑结构的实现时,读者将看到顺序和链式两种存储结构以及两者相结合的实际应用。

1.3 算法

算法(Algorithm)是指解题方案的准确而完整的描述,是一系列解决问题的清晰指令,每一条指令对应一个或多个操作,按照算法要求去执行这些操作,便能解决这个特定问题。也就是说,算法代表着用系统的方法描述解决问题的策略机制,能够对一定规范的输入,在有限时间内获得所要求的输出。算法需要满足以下 5 个性质:

(1)有穷性(Finiteness):算法必须能在执行有限个步骤之后终止;

(2)确切性(Definiteness):算法的每一步骤必须有确切的定义;

(3)输入项(Input):一个算法有 0 个或多个输入,以刻画运算对象的初始情况,所谓 0 个输入是指算法本身定义了初始条件;

(4)输出项(Output):一个算法有 1 个或多个输出,以反映对输入数据加工后的结果,没有输出的算法是毫无意义的;

(5)可行性(Effectiveness):算法中执行的任何计算步骤都是可以被分解为基本的可执行的操作步,即每个计算步都可以在有限时间内完成(也称之为有效性)。

描述算法可以采用自然语言、流程图以及伪代码等多种形式。在保证正确性的前提下,一个算法的优劣可以用时间复杂度与空间复杂度来衡量。

一般来说,算法中基本操作的执行次数与问题规模 n 存在函数 $f(n)$ 关系,记作:

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

随着问题规模 n 的增大，算法执行时间的增长率与 $f(n)$ 的增长率正相关，称作算法的渐进时间复杂度(Asymptotic Time Complexity)，简称时间复杂度。例如，在下列三个程序段中：

```
(a)++x;
(b)for(i=0; i<n ;++i) ++x;
(c)for(i=0; i<n; ++i)
       for(j=0; j<=i ;++j)
          ++x;
```

如果将“x 增 1”看成基本操作，上面代码的时间复杂度分别为 $O(1)$、$O(n)$ 和 $O(n^2)$。虽然，代码(c)执行次数为 $n(n+1)/2$，不是正好的 n^2 次，但 $n(n+1)/2$ 与 n^2 成正比，所以记录算法复杂度时，仍记作 $O(n^2)$。

注意，上述的算法时间复杂度是在算法执行之前对算法时间效率的一种估计方法，而不是算法实际的执行时间。因为，算法实际的执行时间不仅受到问题规模大小的影响，机器速度、采用哪种高级语言编写等因素也能够干扰实际执行时间。评价算法时间效率时，应该将除了问题规模以外的因素刨除。

常见的时间复杂度包括：常数阶 $O(1)$、对数阶 $O(\log n)$、线性阶 $O(n)$、平方阶 $O(n^2)$、立方阶 $O(n^3)$、指数阶 $O(2^n)$，等等。如图 1.6 所示，常数阶的算法优于对数阶算法，对数阶算法优于线性阶算法等多项式阶算法，而多项式阶算法优于指数阶算法。实际应用中，指数阶算法往往不被采用。

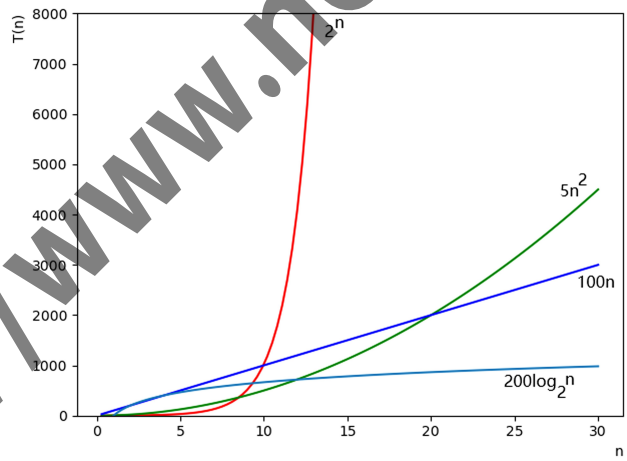

图 1.6　各种数量级的时间复杂度

算法的空间复杂度是指算法需要消耗的内存空间。其计算和表示方法与时间复杂度类似，一般都用复杂度的渐近性来表示。同时间复杂度相比，空间复杂度的分析要简单得多。

1.4　为什么要学习数据结构

数据结构是计算机相关专业最重要的专业基础课程之一，在计算机、互联网等领域有着十分重要作用。因此，有必要在介绍第一个数据结构之前，站在应用的角度，为读者简单地

解释一下“为什么要学习数据结构?”。这里应用领域选择了人们熟识的“搜索引擎”。但这仅是数据结构诸多应用领域中的一个。

在当今的互联网世界中,搜索引擎是最重要的入口之一,人们可以通过搜索引擎快速找到自己要访问的资源。那么一个搜索引擎的实现都用到了哪些重要的技术呢?我们首先考虑搜索引擎能够实现哪些功能。

第一,搜索引擎要能够识别用户的输入,根据输入快速找到与之相关的主题。这里涉及两个重要知识点,一是把用户输入的内容进行合理分解,提取出合适的主题,例如,用户如果输入“大连东软信息学院人才培养方案”,除了直接把整个名称作为整体进行查询之外,还可以把这个内容分解成“大连东软信息学院”以及“人才培养方案”,可以分别按照这两个主题进行查询,进一步还可以分解出“东软”“培养方案”“大连”“学院”“人才”等更加细粒度的主题,根据不同的重要程度进行查询,此处会用到自然语言处理中的分词技术,并且是结合以往大量用户历史输入的信息对分词结果进行合理排序,然后依据不同的权重进行分别的查询。第二个重要知识点是,当给定某个主题后,如何在后台数据库中快速查找到对应结果。如果数据量在千万级甚至上亿级,在设计合理的前提下,传统关系数据库中可以在一秒之内快速返回查询结果,但如果是互联网中的海量数据,采用原有技术就很难在一秒之内返回结果了,需要采用分布式集群管理数据库的索引,并且采用利于快速查询的数据结构来构建索引。高效的排序和查找算法必不可少。

第二,搜索引擎要事先存储大量网上资源的地址,以便在用户点击某个查找结果时能够跳转到对应页面中。互联网中的信息时刻都在更新,搜索引擎会释放很多数量的“爬虫”到互联网中不断爬取最新的数据。爬取数据时,可以将互联网中的网页看成一个图中的结点,把网页中的超链接看成一条有向边,这样就把整个互联网看成了一幅巨型的有向图,爬虫问题就转换成为在有向图中的搜索问题。

第三,搜索引擎要具有一定的“智能性”,当用户输入有误时,可以给出提醒或者自动更正。例如在搜索引擎中输入“Mississippi”,这个单词中字母 s 和 p 出现了多次,很容易弄错,一旦用户输入的内容跟实际单词相比有一定误差时,例如输入成“Missisippi”,系统将能够计算这两个单词的相关度,如果相关度高于一定的阈值,就会认为用户输入时可能出现错误,将给出与用户输入单词相关程度达到阈值以上的其他单词的提示,提醒用户可能输入错误。此处将会对字符串进行匹配,并计算相关度。

第四,搜索引擎要能够进行合适的推荐,当用户输入一部分内容时,根据其他用户以往输入的相关词语的频度,通过下拉框提醒用户可能的后续输入内容,方便用户快速录入。实现这部分功能需要先期记录大量用户的历史输入情况,并且能够在用户输入的过程中实时进行匹配,快速从历史记录中找到相似程度最高(需要合理计算相似度)的若干条,提供给用户选择。此处涉及海量历史输入数据的存储,从海量历史输入中快速检索关键词,计算两个词语的关联程度等技术。

第五,搜索引擎要有一定的“记忆性”,用户输入过的内容会记录下来,下次用户输入相同内容的前几个字时,就可以通过下拉框显示后续内容,便于用户快速录入。此处实现相对简单,因为只关心用户自身的历史输入情况,这些数据量相对于互联网来说是很小的,通常存储在自己的个人电脑或手机中(使用搜索引擎的设备),然后从中进行快速查询和匹配,相

似度超过阈值的将会给出提示。

通过学习本教材各部分内容，将能够部分解决以上提到的问题，至少能够给出一些较为合理的设计和实现方案。

上面仅仅是通过一个实例说明数据结构在计算机程序设计中的作用，事实上很多系统软件在设计和开发时都会用到各种各样的数据结构，尤其是操作系统、编译系统、数据库管理系统等系统软件，这些软件往往对于性能、兼容性、健壮性等要求极高。相对而言，一些应用软件中可以直接使用开发框架中特定数据结构的现成实现，对于掌握数据结构内部具体实现以及性能分析的要求不是那么高，但如果掌握了相关数据结构的基本原理，可以在多种备选方案中选择较优的实现方式，也有利于应用软件的性能、稳定性和兼容性等指标的实现。

习 题

一、选择题

1. 数据结构课程中，讨论计算机数据处理的基本单位是(　　)。

A. 字节　　B. 位　　C. 数据项　　D. 数据元素

2. 数据的逻辑结构不包括(　　)。

A. 线性结构　　B. 树状结构　　C. 总线结构　　D. 图状结构

3. 语句 for(i=1; i<n; i*=2) ++x; 的时间复杂度为(　　)。

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

二、判断题

1. 算法的时间复杂度与机型有关。(　　)

2. 算法的空间复杂度与采用何种语言有关。(　　)

部分习题解析

选择题第3题，代码 for(i=1; i<n; i*=2) ++x;中，循环体中没有修改循环变量的语句，也不存在内层循环，因此只需分析 for 语句部分。循环变量 i 从1开始，当 i 不小于 n 时循环终止，变量 i 在每轮循环中乘等2，i 值以指数级的增长速度接近 n。所以，答案选B。

第 2 章　线性表

一、单元概述

现实世界中很多信息可以通过“列表”的形式展现。生活中常见的“超市商品价目表”“饭店菜单”“航班时刻表”等等，采用带有一定次序的“列表”形式展现，既简单又直观，方便人们查询和统计。这种“列表”如何在计算机中存储，如何向“列表”中插入或删除“列表项”，如何存储可方便插入或删除“列表项”，如何存储更加便于查询等，就是本章要回答的问题。

线性表是信息的一种表示形式，是所有数据结构中最常用、最简单的一种数据结构。本章主要介绍线性表的逻辑结构、顺序存储及实现、链式存储及实现，还介绍了应用实例。

二、教学重点与难点

重点：理解顺序表和链表的特点以及二者的应用场合；掌握顺序表、链表的实现方法。

难点：实现链表结构时，指针操作较多，需要特别注意特殊情况处理，例如，指针指向空地址、指针指向链表的头、指针指向链表的尾等等。

2.1　线性表的基本定义

线性表是最简单、最基本、最常用的数据结构。例如，表示一个平面上的多边形，可以使用多边形各个顶点坐标作为数据元素构成的线性表；表示手机中的一组通话记录，可以使用单条通话记录作为数据元素构成的线性表，等等。

一个线性表是由 n 个元素构成的有限序列($n \geqslant 0$)。$n=0$ 时，线性表称为空表；当 $n>0$ 时，线性表表示为$(a_1, a_2, a_3, \cdots, a_n)$，其中 a_1 称为线性表的第一个元素，a_n 称为线性表的最后一个元素。元素 a_{i-1} 称为元素 a_i 的前驱($1<i \leqslant n$)，元素 a_{j+1} 称为元素 a_j 的后继($1 \leqslant j<n$)。显然，除了第一个元素没有前驱、最后一个元素没有后继之外，线性表中的其他元素都有且仅有一个元素是它的前驱，有且仅有一个元素是它的后继。

对线性表经常进行的操作包括插入元素、删除元素等。这里使用 List 表示线性表类型，ADT 表示线性表中存放数据元素的类型，下面使用 C 语言的函数原型来描述线性表的一些常见操作：

(1)List * init(List * lp);//初始化，将一个 List 变量初始化为空的线性表。

(2)List * clear(List * lp);//清空，删除线性表中全部元素，逻辑上清空线性表。

(3)void destroy(List * lp);//销毁，不再使用该表前需要进行的操作，注意和清空区别。

(4)_Bool empty(const List * lp);//判断是否为空表。

(5)unsigned length(const List * lp);//求线性表长度。

(6)List * insert(List * lp, ADT data,int i);//在线性表第 i 个元素之前插入元素 data。

(7)List * erase(List * lp, int i);//删除线性表第 i 个元素。

(8)int remove(List * lp, ADT data);//删除线性表中所有值等于 data 的元素。

如果使用 C 语言以外的程序设计语言描述线性表结构，对于以上列举的操作，可能出现不同的展现形式。例如，使用 C++语言描述，init 操作将被实现为线性表类型的构造函数，destroy 操作将被实现为线性表类型的析构函数；使用 Java 语言描述，可能不需要实现 destroy 操作，等等。

对线性表进行的常见操作不仅限于上面列举的 8 种。根据线性表应用场合不同，可能需要对线性表进行其他的一些操作，例如，排序、翻转、与另一个线性表合并，等等。另外，如果采用不同的存储结构来实现线性表，在实现线性表的一些操作时，可能会根据存储结构的特点，使用不同的函数原型。接下来的 2.2 节和 2.3 节，将分别介绍线性表的顺序存储和链式存储。

2.2　线性表的顺序表示和实现

本节介绍线性表的顺序表示方法，并给出 C 语言实现。顺序表示强调数据元素占用连

续的存储空间,将其用于线性表结构的实现,十分简单、直观。

2.2.1 顺序表的基本概念和特点

线性表的顺序表示是指用一组连续的存储单元依次存储线性表的数据元素。假设线性表中每个数据元素占用 l 个存储单元,第 i 个元素的地址是 $LOC(a_i)$,那么第 $i+1$ 个元素的地址 $LOC(a_{i+1})$ 等于 $LOC(a_i)+l$。

假设 n 个元素的线性表 $(a_1,a_2,\cdots,a_n)$ 采用顺序表示,表中首个元素的地址 $LOC(a_1)$ 称为该线性表的基地址,那么,第 i 个数据元素的地址 $LOC(a_i)$ 等于 $LOC(a_1)+(i-1)\times l$。

由于本教材采用 C 语言实现各种数据结构和算法,C 语言数组下标从 0 开始计算。我们总是将数据结构中的"第 1 个"元素存放在数组下标为 0 的存储单元中。为了减少描述过程中的麻烦,本教材后续章节中,在描述逻辑结构时,序号也从 0 开始计算,即逻辑结构中的首个元素不再称为"第 1 个"元素,而是称为"第 0 个"元素。例如,n 个元素的线性表不再表示为 $(a_1,a_2,\cdots,a_n)$,而是表示为 $(a_0,a_1,\cdots,a_{n-1})$。在这种情况下,线性表 $(a_0,a_1,\cdots,a_{n-1})$ 的基地址为 $LOC(a_0)$,第 i 个数据元素的地址 $LOC(a_i)$ 等于 $LOC(a_0)+i\times l$。

线性表的这种计算机表示称为线性表的顺序存储结构,通常称这种存储结构的线性表为顺序表,如图 2.1 所示。

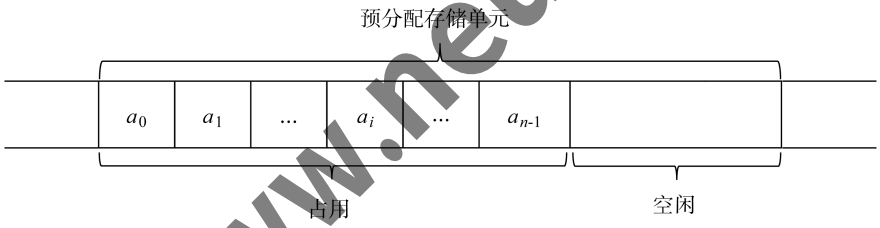

图 2.1 顺序表存储示意图

由于顺序表中的元素是连续存储的,只要确定了表的基地址,表中的任一数据元素都是可以随机存取的。存取顺序表中任一元素 a_i 的时间复杂度为 $O(1)$。

考虑向顺序表插入数据元素和从顺序表中删除数据元素的情况。向 n 个元素的线性表 $(a_0,a_1,\cdots,a_i,\cdots,a_{n-1})$ 第 i 个元素之前的位置上插入新元素 b,线性表变为 $(a_0,a_1,\cdots,b,a_i,\cdots,a_{n-1})$。如果线性表采用顺序存储,如图 2.1 所示,需要将顺序表中,编号从第 i 至第 $n-1$ 位置上的 $n-i$ 个元素向后移动 1 个存储单元。n 个元素的顺序表有 $n+1$ 个可能的插入位置,向这些位置插入元素依次需要移动 n、$n-1$、…、1、0 个元素。假设在表的任何位置插入元素的概率是相等的,在表长为 n 的情况下,插入 1 个元素平均要移动 $n/2$ 个元素,时间复杂度为 $O(n)$。类似的,删除顺序表中的第 i 个元素,需要将编号从第 $i+1$ 至 $n-1$ 位置上的 $n-i-1$ 个元素向前移动 1 个存储单元。n 个元素的顺序表有 n 个可能的删除位置,从这些位置删除元素依次需要移动 $n-1$、$n-2$、…、1、0 个元素。假设从表的任何位置删除元素的概率是相等的,在表长为 n 的情况下,删除 1 个元素平均要移动 $(n-1)/2$ 个元素,时间复杂度为 $O(n)$。

通过对顺序表特点的分析,可以看出,顺序表适合应用在频繁访问元素,但很少插入、删除元素或仅在尾部插入、删除元素的场合。

2.2.2　顺序表的实现

本小节使用 C 语言中的动态内存实现一个可自动增长的顺序表结构以及顺序表的基本操作。

1. 类型定义

```
typedef int ADT;
typedef struct
{
    unsigned size;//元素个数
    unsigned capacity;//当前已分配存储单元个数
    ADT * array;//指向存储单元的指针(基地址)
}SqList;
```

代码中首先定义类型 ADT(Abstract Data Type),表示顺序表存放数据元素的类型,在本教材的其他部分,“ADT”的含义也是如此,即表示数据结构中数据元素的数据类型。由于 C 语言不支持泛型程序设计,为简便起见本节以 int 为例。

类型 SqList 表示顺序表类型。更严格的讲,SqList 应称为顺序表结构的管理结构。其中,结构体分量 size 用于表示顺序表当前存放元素的个数,结构体分量 capacity 用于表示当前顺序表最多存放元素的个数(存储单元的长度)。如果将一个教室比作顺序表,教室能容纳的人数就是这个教室的“capacity”,而当前教室中实际的人数就是“size”。结构体分量 array 是指向存储单元的指针,那里是实际存放数据元素的地方,这就是将 SqList 称为顺序表管理结构的原因。另外,此处使用“array”一词,是强调指针指向的存储单元是连续的。

2. 初始化、清空和销毁操作

(1)初始化操作

```
#include<malloc.h>
SqList * init_sq(SqList * p,unsigned c)
{
    p->size=0;
    p->capacity=c?c:8;
    p->array=(ADT * )malloc(sizeof(ADT) * p->capacity);
    return p;
}
```

使用类型 SqList 定义的变量不能称之为顺序表,例如 SqList s,此时变量 s 的各个分量 s.size、s.capacity 和 s.array 未经初始化,其中存放的数据毫无意义,同时存放数据元素的存储单元也未指定。因此,变量 s 只有经过初始化操作后,才能称为顺序表,才能进行其他操作。

函数 init_sq 完成顺序表初始化功能,初始化后的顺序表为空表。参数 p 表示待初始化 SqList 变量的地址,参数 c 表示希望为顺序表预分配存储单元的个数,显然 c 应该是正整数(代码中对 c 为 0 的情况进行了简单处理)。函数返回初始化后顺序表变量的地址,含义是“初始化顺序表操作后得到的结果就是这个被初始化的顺序表”。代码中将 size 初始为 0,表

示空表;将 capacity 初始为参数 c 的值;使用 malloc 函数分配能够存放 capacity 个 ADT 类型数据元素的堆区内存,首地址保存在 array 中。此处未考虑 malloc 函数分配内存失败的情况,更加严谨的做法是,在 malloc 失败后(malloc 函数返回地址 0 表示失败),也为函数 init_sq 返回地址 0,表示初始化操作失败。即在

```
return p;
```

之前增加语句

```
if(!p->array) return (void*)0;
```

这里表达式"(void*)0"强调返回的是地址 0,而不是整数 0,当然可直接写成"0"或宏定义"NULL"(写成 NULL 需要包含定义该宏的头文件)。

(2)清空操作

```
SqList* clear_sq(SqList* p)
{
    p->size=0;
    return p;
}
```

函数 clear_sq 完成逻辑上清空一个顺序表的功能。这里只需将 size 赋值为 0 即可。函数最后返回清空后的顺序表的地址,清空后的顺序表仍然可以继续使用。这就像教室里面的人走光了,教室还是那个教室,仍然可以再进人。

(3)销毁操作

```
void destroy_sq(SqList* p)
{
    free(p->array);
}
```

由于本节实现的顺序表存放数据元素使用的是堆区内存(参见函数 init_sq 的代码),因此,为避免内存泄漏,应该在顺序表使用完毕后,SqList 类型变量释放之前,释放存储数据元素的堆区内存。函数 destroy_sq 完成这一功能,我们称为顺序表的销毁操作。注意,函数 destroy_sq 没有像初始化和清空操作一样,返回所操作顺序表的地址,这是因为经过此操作处理后的 SqList 类型变量已经不是顺序表了,在逻辑上顺序表已经不存在了(被销毁了)。

3. 读写元素操作

读取下标编号为 i 的元素的代码如下,读操作函数 geti_sq 将读到的元素值写入参数 dp 指向的空间中。

```
const SqList* geti_sq(const SqList* p, unsigned i, ADT* dp)
{
    if(i>=p->size)
        return (void*)0;
    *dp=p->array[i];
    return p;
}
```

修改下标编号为 i 的元素的代码如下,写操作函数 seti_sq 使用参数 d 的值覆盖顺序表第 i 个元素值。

```
SqList * seti_sq(SqList * p,unsigned i,ADT d)
{
    if(i>=p->size)
        return (void * )0;
    p->array[i]=d;
    return p;
}
```

虽然在 C 语言中,永远可以使用“s.array[i]”的形式访问顺序表 s 中的第 i 个数据元素。这里仍然提供了 geti_sq 和 seti_sq 两个函数分别用于顺序表元素的安全读写。两个函数在成功操作后,返回所操作顺序表的地址,含义是“对顺序表进行读或写元素操作后,顺序表还是那个顺序表”。函数中,首先对表示元素索引号的参数 i 进行了验证,如果 i 不在合理范围之内,函数返回地址 0。

4. 空间扩展操作

```
#include<string.h>
SqList * extend_sq(SqList * p)
{
    ADT * np=(ADT * )malloc(sizeof(ADT) * p->capacity * 2);
    memcpy(np,p->array,sizeof(ADT) * p->size);
    free(p->array);
    p->array=np;
    p->capacity * =2;
    return p;
}
```

空间扩展操作是向顺序表插入元素操作的辅助操作,如果顺序表预先分配的存储单元用尽,应该对顺序表进行空间扩展。函数 extend_sq 首先分配了 2 倍于原有空间的存储单元;之后使用 memcpy 函数复制顺序表中的数据元素至新空间,释放原有空间内存;最后调整 array 和 capacity 为正确的值。

5. 尾部插入元素和删除元素操作

对于顺序表来说,在尾部插入和删除元素是常见的操作,其代码如下。

```
SqList * push_back_sq(SqList * p, ADT d)
{
    if(p->capacity==p->size)
        extend_sq(p);
    p->array[p->size++]=d;
    return p;
}
SqList * pop_back_sq(SqList * p)
{
    if(p->size==0)
        return (void * )0;
    --p->size;
```

```
    return p;
}
```

如果不考虑空间扩展操作,在顺序表的尾部进行元素插入和删除操作,算法时间复杂度均为$O(1)$。函数 push_back_sq 和 pop_back_sq 的返回值均为成功操作后的顺序表,特别注意的是,当顺序表为空时,删除操作失败,函数 pop_back_sq 返回地址 0 表示这种情况。

6. 任意位置插入元素和删除元素操作

(1)在指定位置插入元素操作的代码

```
SqList * insert_sq(SqList * p, unsigned i, ADT d)
{
    if(i>p->size)
        return (void * )0;
    if(p->size==p->capacity)
        extend_sq(p);
    memmove(p->array+i+1,p->array+i,sizeof(ADT) * (p->size-i));
    p->array[i]=d;
    ++p->size;
    return p;
}
```

函数 insert_sq 的功能是向 p 指向的顺序表的第 i 个位置上插入值 d,成功插入返回 p,失败返回地址 0。插入失败的原因是由于参数 i 超出了合理范围。对于一个拥有 n 个元素的顺序表,合理的插入位置有 $n+1$ 个,参数 i 合理的取值是 0 至 n。

(2)删除指定位置元素的代码

```
SqList * erase_sq(SqList * p, unsigned i)
{
    if(i>=p->size)
        return (void * )0;
    memmove(p->array+i,p->array+i+1,sizeof(ADT) * (p->size-i-1));
    --p->size;
    return p;
}
```

函数 erase_sq 的功能是删除 p 指向的顺序表的第 i 个位置上的元素,成功删除返回 p,失败返回地址 0。删除失败的原因是由于参数 i 超出了合理范围。对于一个拥有 n 个元素的顺序表,合理的删除位置有 n 个,参数 i 合理的取值是 0 至 $n-1$。

在顺序表的任意位置插入(删除)元素,会造成操作位置之后的元素向后(向前)移动。为完成元素移动,这里使用了 C 语言函数 memmove,而不是 memcpy。函数 memcpy 复制内存数据时,不考虑数据目的地址和源地址重叠的情况。当元素目的地址和源地址重叠时,向后移动数据,应该从后向前逐一复制元素。向前移动数据,应该从前向后逐一复制元素。只有如此操作,才能保证数据被覆盖(写)之前被读取。读者可以自编循环语句代替函数 memmove,完成元素移动操作,进而体会 memmove 和 memcpy 的细微差别。

7. 移除元素操作

```
/*
unsigned remove_sq(SqList * p, ADT d)
{
    unsigned count,r,w;
    count=r=w=0;
    for(;r<p->size;++r)
        if(p->array[r]==d)
            ++count;
        else
            p->array[w++]=p->array[r];
    p->size-=count;
    return count;
}
*/
```

注释的函数 remove_sq 可用于删除顺序表中所有值等于参数 d 的元素，函数返回值为删除元素的个数。我们将此操作称为“移除”。代码中的变量 r 和 w 分别跟踪读写位置，当读到的元素不等于 d 时，将发生写操作，每写一个元素 w 便向后移动一个位置，不论写操作是否发生，当扫描过顺序表的全部元素后，调整变量 size，算法终止。

```
unsigned remove_if_sq(SqList * p, _Bool(*con)(ADT))
{
    unsigned count,r,w;
    count=r=w=0;
    for(;r<p->size;++r)
        if(con(p->array[r]))
            ++count;
        else
            p->array[w++]=p->array[r];
    p->size-=count;
    return count;
}
```

函数 remove_if_sq 用于删除顺序表中所有满足某个条件的元素。条件使用参数 con 表示，con 是一个一元判定函数的指针。将顺序表元素当作参数调用 con 指向的函数，如果返回值为真，表示满足条件(需要删除)，返回值为假，表示不满足条件(不需要删除)。显然，函数 remove_sq 表示的操作是函数 remove_if_sq 表示操作的特例。这里，函数 remove_if_sq 的代码复制于注释的函数 remove_sq 的代码，仅仅是将原来使用“相等”判定是否删除的语句，改成了使用函数指针所表示的函数来判定是否删除。

```
unsigned remove_sq(SqList * p, ADT d)
{
    _Bool equal(ADT v)
    {
```

```
        return v==d;
    }
    return remove_if_sq(p, equal);
}
```

未注释的 remove_sq 函数体内,提供了函数 equal 用于判断参数 v 是否等于外层函数参数 d。后面的代码是使用 equal 作为参数调用函数 remove_if_sq。

8. 测试代码与总结

```
#include<stdio.h>
void print(const SqList * p)
{
    int i;
    printf("size= %d,capacity= %d,elements:",p->size,p->capacity);
    for(i=0; i<p->size; ++i)
    {
        int x;
        geti_sq(p,i,&x);
        printf("%d ",x);
    }
    printf("\n");
}
_Bool odd(int n)
{
    return n&1;
}
int main(void)
{
    int i;
    SqList sq;
    init_sq(&sq,4);
    for(i=1; i<=10; ++i)
    {
        push_back_sq(&sq,i);
        print(&sq);
    }
    for(i=0; i<6; ++i)
    {
        pop_back_sq(&sq);
        print(&sq);
    }
    seti_sq(&sq,1,100);
    print(&sq);
    insert_sq(&sq,1,200);
```

```
    print(&sq);
    erase_sq(&sq,2);
    print(&sq);
    remove_if_sq(&sq,odd);
    print(&sq);
    remove_sq(&sq,200);
    print(&sq);
    clear_sq(&sq);
    print(&sq);
    destroy_sq(&sq);
    return 0;
}
```

显示结果：

```
size=1,capacity=4,elements:1
size=2,capacity=4,elements:1 2
size=3,capacity=4,elements:1 2 3
size=4,capacity=4,elements:1 2 3 4
size=5,capacity=8,elements:1 2 3 4 5
size=6,capacity=8,elements:1 2 3 4 5 6
size=7,capacity=8,elements:1 2 3 4 5 6 7
size=8,capacity=8,elements:1 2 3 4 5 6 7 8
size=9,capacity=16,elements:1 2 3 4 5 6 7 8 9
size=10,capacity=16,elements:1 2 3 4 5 6 7 8 9 10
size=9,capacity=16,elements:1 2 3 4 5 6 7 8 9
size=8,capacity=16,elements:1 2 3 4 5 6 7 8
size=7,capacity=16,elements:1 2 3 4 5 6 7
size=6,capacity=16,elements:1 2 3 4 5 6
size=5,capacity=16,elements:1 2 3 4 5
size=4,capacity=16,elements:1 2 3 4
size=4,capacity=16,elements:1 100 3 4
size=5,capacity=16,elements:1 200 100 3 4
size=4,capacity=16,elements:1 200 3 4
size=2,capacity=16,elements:200 4
size=1,capacity=16,elements:4
size=0,capacity=16,elements:
```

正确输入本节提供的所有代码并包含适当的头文件，编译、运行程序，可得到如上的显示结果。

顺序表结构在真实的项目开发中十分常用，许多高级语言的模板库都提供了顺序表的实现，例如，C++中的 vector、deque(C++中的 deque 不是严格意义上的顺序表，但提供顺序表常用的一些操作)，Java 中的 ArrayList、Vector 等。顺序表结构适用于那些频繁进行随机访问元素、插入删除元素操作仅在尾部进行的场合。另外，当预留存储单元占满后，如果

采用重新分配内存代替原内存的方式来扩展空间(本节代码就是这样处理的),为避免大量数据的频繁移动,应在创建顺序表结构时预留合适的存储单元。

2.3 线性表的链式表示和实现

线性表的顺序存储结构的特点是逻辑上相邻的两个数据元素在物理存储位置上也是相邻的,因此仅通过简单的地址计算,便能随机访问表中的任意元素。但是,从另一个角度出发,这一特点也造成顺序表的弱点:在任意位置插入或删除元素时,需要移动大量元素。本节讨论线性表的另一种重要的表示方法——链式存储结构,由于它不需要逻辑上相邻的元素在物理存储位置上也相邻,因此它没有顺序表插入删除元素时需要移动元素的弱点,但也失去了顺序表支持随机访问的优点。

2.3.1 链表的基本概念

线性表的链式存储结构是用一组任意的存储单元存储线性表的数据元素,这些存储单元可以是连续的,也可以是不连续的。为了表示线性表元素 a_i 与其直接后继元素 a_{i+1} 的逻辑关系,存储数据元素时,除了数据元素本身信息外,还需要存储直接后继元素的位置信息。这两部分信息组成的数据元素 a_i 的存储映像,被称为结点。结点分为两部分,存放数据元素信息的称为数据域,存放直接后继元素位置信息的称为指针域。采用链式存储结构表示的 n 个元素的线性表(a_0,a_1,…,a_{n-1}),就是由 n 个结点链成一条链表。

结点指针域仅存放后继元素位置信息时,链表称为单向链表,简称单链表;结点指针域同时存放前驱元素位置信息和后继元素位置信息时,链表称为双向链表,简称双链表。图 2.2 中,(a)是一个单链表的部分结点,(b)是一个双链表的部分结点。

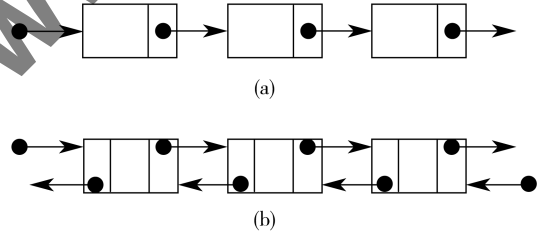

图 2.2 单链表和双链表

若使用链式结构表示线性表,仅仅描述结点结构是不够的,还需要描述链表从哪个结点开始,到哪个结点结束。我们将线性表首个元素 a_0 对应的链表结点称为首元结点,很显然,如果已知首元结点的地址,不仅可以访问首元结点,沿着各个结点指针域存放的后继结点位置信息,可以依次访问链表全部的结点。为记录首元结点地址,至少需要一个指针变量,我们将这个记录首元结点地址的指针变量称为链表的头指针,如果头指针为空(存放 0 地址),则表示空链表。有些情况下,为了方便链表的某些操作,会在首元结点之前附设一个数据域为空,指针域存放首元结点位置信息的结点,该结点称为头结点。图 2.3 使用单链表展示了头指针和头结点的差别,(a)是头指针的示意图,(b)是头结点的示意图。双链表的情况与单

链表类似。

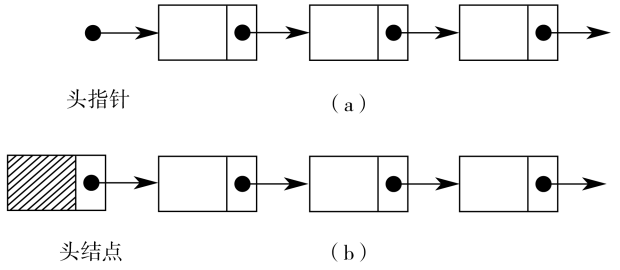

图 2.3　头指针和头结点

表示链表的尾部虽然可以像表示链表头部一样采用一个尾指针存放尾结点地址，但在链表结构中，尾指针并不是像头指针(或头结点)一样必须存在。由于线性表的最后一个元素不存在后继元素，因此链表的最后一个结点就不存在后继结点，可以通过将最后一个结点的指针域设置为特殊值的方法来表示链表到此结束。根据特殊值的含义不同，通常有两种表示链表尾部的方法，一种称为线性链表(非循环链表)，一种称为循环链表。对于非循环链表来说，如果指针域存放内存地址，最后一个结点指针域的特殊值可设置为 NULL(地址 0)，如果指针域存放数组下标(这种情况可能出现在使用连续存储空间存放链表结点的情况)，最后一个结点指针域的特殊值可选择－1 等非法数组下标。对于循环链表来说，最后一个结点指针域的特殊值可设置为首元结点或头结点(若存在头结点)的位置信息，即将首元结点或头结点当作最后一个结点的后继结点。另外，双链表结点的指针域除了后继结点位置信息外，还包括其前驱结点的位置信息，此时，首元结点(无头结点)或头结点(有头结点)的前驱指针应设置为空(非循环链表)，或令其指向尾结点(循环链表)。图 2.4 使用不带头结点的单链表展示了非循环链表和循环链表的差别，(a)是非循环单链表的示意图，(b)是循环单链表的示意图。

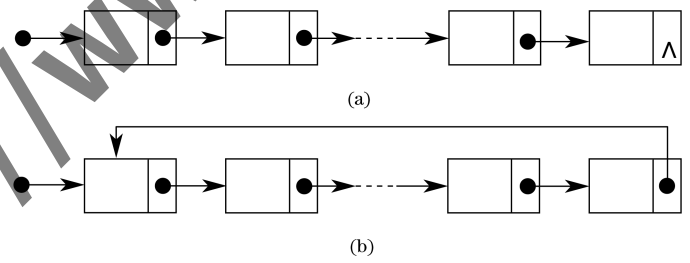

图 2.4　非循环单链表和循环单链表

2.3.2　链表的实现

本小节使用 C 语言实现一个带头尾指针、采用不连续存储单元的单向链表，并实现该链表的基本操作。在实现链表操作的过程中，将会分析链表结构的特点。

1. 类型定义及创建新结点函数

```
typedef int ADT;
typedef struct N
{
```

```
    ADT data;
    struct N * next;
}Node;
```

类型 Node 表示单链表结点。其中,结构体分量 data 表示数据域,结构体分量 next 表示指针域,指针域用于存放后继结点的内存地址。

```
typedef struct
{
    Node * head, * tail;
    unsigned size;
}List;
```

类型 List 表示单链表。更严格的讲,List 应称为单链表的管理结构。其中,结构体分量 size 用于表示单链表当前存放元素的个数,即链表长度。结构体分量 head 和 tail 分别存放链表首元结点和尾结点的内存地址。当链表为空时,head 和 tail 存放地址 0。

```
#include<malloc.h>
Node * make_node(ADT d)
{
    Node * np=malloc(sizeof(Node));
    if(!np)
        return (void * )0;
    np->data=d;
    np->next=(void * )0;
    return np;
}
```

函数 make_node 用于产生一个新结点。首先,使用 malloc 函数申请恰好存放一个链表结点的堆区内存;内存申请成功后,将结点数据域 data 设置为参数 d 的取值,将指针域设置为地址 0(该结点暂时不属于任何链表);最后,返回这个新结点的地址。此函数将被后续相关函数调用。

2. 初始化、清空和销毁操作

(1)初始化操作

```
List * init_list(List * lp)
{
    lp->head=lp->tail=(void * )0;
    lp->size=0;
    return lp;
}
```

函数 init_list 负责将参数 lp 指向的 List 类型变量初始化为空链表,具体做法是将 head 和 tail 都指向地址 0,表示链表长度的变量 size 也设置为 0,最后返回设置好的空链表的地址。

(2)清空操作

```
List * clear_list(List * lp)
{
```

```
    while(lp->head)
    {
        Node * del=lp->head;
        lp->head=del->next;
        free(del);
    }
    lp->tail=(void * )0;
    lp->size=0;
    return lp;
}
```

函数 clear_list 的作用是将一个链表清空。首先利用循环语句逐一释放链表结点占用的堆区内存,然后将 head、tail 置为地址 0,将 size 置为数字 0,最后返回设置好的空链表的地址。

(3)销毁操作

```
void destroy_list(List * lp)
{
    clear_list(lp);
}
```

函数 destroy_list 负责销毁一个链表,其中调用了函数 clear_list。虽然,函数 destroy_list 与函数 clear_list 功能相似。但是,函数 destroy_list 的作用是销毁链表(释放链表结点占用内存),一个链表被销毁后应该不再被使用,因此,函数 destroy_list 的类型为 void。而函数 clear_list 的作用是清空链表,链表被清空后仍是一个可用的链表,因此,函数 clear_list 的类型为 List * ,表示链表的地址。本例中,链表结点的内存是在进行插入操作时申请的,并且每次仅申请一个结点的内存。所以,本例的清空操作和销毁操作拥有类似的代码。然而,如果链表结点内存采用其他申请方式,清空操作和销毁操作则可能使用不同的代码。

3. 在两端插入元素和删除元素操作

(1)在链表头部插入元素的代码

```
List * push_front_list(List * lp, ADT d)
{
    Node * new_node=make_node(d);
    if(!new_node)
        return (void * )0;
    new_node->next=lp->head;
    lp->head=new_node;
    if(lp->tail==(void * )0)
        lp->tail=lp->head;
    ++lp->size;
    return lp;
}
```

函数 push_front_list 用于在参数 lp 指向的链表的头部插入数据 d。函数中首先调用 make_node 函数创建了新结点,新结点的地址存入临时变量 new_node;然后让新结点的指

针域指向链表原来的首元结点,链表头指针指向新结点;判断如果链表尾指针为空(表示原来的链表是空链表),此时新结点即是首元结点,又是尾结点,所以需要使尾指针也要指向新结点;最后调整表示链表长度的变量。注意,本函数的第 2 个参数 d 表示要插入的数据,而不是要插入的结点。因为,对于使用链表结构的程序员来说,所使用的是存放数据的容器,而不是存放结点的容器,他不需要知道链表的内部结构。

(2)在链表尾部插入元素的代码

```
List * push_back_list(List * lp, ADT d)
{
    Node * new_node=make_node(d);
    if(!new_node)
        return (void * )0;
    if(lp->tail)
        lp->tail->next=new_node;
    else
        lp->head=new_node;
    lp->tail=new_node;
    ++lp->size;
}
```

函数 push_back_list 用于在参数 lp 指向的链表的尾部插入数据 d。与函数 push_front_list 类似,需要根据参数 d 创建新结点;判断如果原来存在尾结点(链表非空),则调整原来尾结点的指针域指向新结点,否则令头指针指向新结点(新结点既是尾结点,又是首元结点);最后,调整链表尾指针和表示长度的变量。注意,如果类型 List 中没有使用变量 tail 表示尾指针,实现在链表尾部插入新数据的操作将进行一个接近 size 次的循环,从链表头部开始,搜索链表尾结点,以便将新结点插入在原来尾结点之后。

(3)删除链表头部元素的代码

```
List * pop_front_list(List * lp)
{
    if(lp->size==0)
        return (void * )0;
    Node * del=lp->head;
    lp->head=del->next;
    free(del);
    --lp->size;
    if(lp->size==0)
        lp->tail=(void * )0;
    return lp;
}
```

函数 pop_front_list 用于删除参数 lp 指向链表中的首元结点。当链表为空时,函数返回地址 0,表示删除操作失败。删除单向链表的首元结点操作十分简单,先定义临时指针指向链表的首元结点,链表头指针指向原来首元结点的下一个结点,然后使用 free 函数释放原

有首元结点，并调整变量 size。值得注意的是，如果原有链表仅有一个结点时，删除唯一结点后需要将尾指针赋值为地址 0。

(4)删除链表尾部元素的代码

```
List * pop_back_list_bad(List * lp)
{
    if(lp->size==0)
        return (void * )0;
    Node * del=lp->tail;
    Node * new_tail=lp->head;
    if(lp->head==lp->tail)
        lp->tail=lp->head=(void * )0;
    else
    {
        while(new_tail->next!=lp->tail)
            new_tail=new_tail->next;
        new_tail->next=(void * )0;
        lp->tail=new_tail;
    }
    free(del);
    --lp->size;
    return lp;
}
```

函数 pop_back_list_bad 用于删除参数 lp 指向链表中的尾结点。函数在判断链表非空后，先定义临时指针指向要删除的结点，同时定义另一个临时指针指向链表的头；然后，判断链表是否仅有 1 个结点，是则令头指针和尾指针指向地址 0(唯一的结点将被删除)，否则寻找链表原有尾结点的前一个结点，并调整它的指针域为地址 0，同时令尾指针指向这个新的尾结点；最后，释放原有尾结点占用的内存，并调整成员 size。

之所以在 pop_back_list_bad 函数的名字中加入 bad，是因为它是一个糟糕的函数。函数为了寻找尾结点的前一个结点，从链表的头部开始向后搜索至链表的倒数第二个结点。其他 3 个函数 push_front_list、push_back_list 和 pop_front_list 所使用的算法与链表长度无关，而 pop_back_list_bad 函数操作的链表越长，该函数执行的越慢。实际上，解决这个问题的根本方法是修改 Node 结构，在其中添加指向前驱结点的指针，使单向链表升级为双向链表。编写双向链表的难度不比编写单向链表的难度大，双向链表仅比单向链表每个结点多付出一个指针的存储空间，却能使很多操作变得简单。读者可在学习完本章全部内容后，将例程中实现的单向链表改造为双向链表。

4. 在链表中查找

```
Node * find_in_list(const List * lp, ADT d)
{
    Node * cur=lp->head;
    while(cur&&cur->data!=d)
        cur=cur->next;
```

```
    return cur;
}
```

函数 find_in_list 用于在参数 lp 指向的链表中,查找首个值等于参数 d 的元素。并返回结点的首地址,返回的地址可供使用者修改结点信息。如果参数 lp 指向的链表中不存在数据域与参数 d 等值的结点,函数返回地址 0。代码中定义结点指针 cur,并初始指向链表头,然后循环查找,当搜索至链表尾部或者找到等值元素时,循环停止,函数返回当前指针 cur。返回时,如果找到,cur 恰好指向要找的结点,否则 cur 的取值为地址 0。

5. 在指定位置插入和删除元素操作

(1)在指定位置插入元素的代码

```
List *   insert_after_list(List * lp, ADT d, Node * pos)
{
    if(pos==(void *)0||lp->size==0)
        return push_front_list(lp,d);
    if(pos==lp->tail)
        return push_back_list(lp,d);
    Node * new_node=make_node(d);
    if(!new_node)
        return (void *)0;
    new_node->next=pos->next;
    pos->next=new_node;
    ++lp->size;
    return lp;
}
```

函数 insert_after_list 用于在参数 lp 指向的链表中,参数 pos 指向的位置之后,插入元素 d。考虑现有 n 个元素的线性表,合法的插入位置有 $n+1$ 个,在现有元素的后面插入新元素的表示方法,只能表示 n 个位置,即插入首个元素之前的那个插入位置无法表示成哪个元素之后。因此,函数 insert_after_list 将在参数 pos 为地址 0 或者链表为空的情况下,将新元素放在链表头部。

为一个单向链表插入新结点时,需要修改插入位置前一个结点的指针域,因此在给定插入位置后,将新元素创建的新结点放在指定位置之后的操作比较方便,算法时间复杂度为 $O(1)$。但是,那样处理不符合正常的操作习惯,正常的插入操作 insert,一般要将待插入元素放置在指定位置 pos 之上(逻辑上原位置的元素向后移,相当于插入在原位置元素之前),而不是插入在指定位置 pos 之后。为此,本例另外提供了函数 insert_list1 用于正常的插入操作。

```
List *   insert_list1(List * lp, ADT d, Node * pos)
{
    if(pos==(void *)0||lp->size==0)
        return push_back_list(lp,d);
    if(pos==lp->head)
        return push_front_list(lp,d);
    Node * new_node=make_node(d);
```

```
    if(!new_node)
        return (void * )0;
    Node * pre=lp->head;
    while(pre&&pre->next!=pos)
        pre=pre->next;
    if(pre)
    {
        new_node->next=pos;
        pre->next=new_node;
        ++lp->size;
    }
    return lp;
}
```

函数 insert_list1 需要将元素 d 插入在指定位置 pos 之前。有 n 个元素的线性表，合法的插入位置有 $n+1$ 个，插入最后一个元素之后的那个插入位置无法表示成哪个元素之前。因此，函数 insert_list1 将在参数 pos 为地址 0 或者链表为空的情况下，将新元素放在链表尾部。

函数 insert_list1 中定义的指针 pre 用于寻找并保存 pos 位置之前的那个结点的地址，新结点将被插入在 pre 之后，pos 之前。为了寻找 pos 位置之前结点的地址，虽然不需要移动任何元素，函数仍付出了 $O(n)$ 的算法时间复杂度（n 为链表长度）。为了将算法时间复杂度变为 $O(1)$，本例又提供了函数 insert_list2。

```
List *  insert_list2(List * lp, ADT d, Node * pos)
{
    if(pos==(void * )0||lp->size==0)
        return push_back_list(lp,d);
    if(pos==lp->head)
        return push_front_list(lp,d);
    Node * new_node=make_node(pos->data);//注意
    if(!new_node)
        return (void * )0;
    pos->data=d;
    new_node->next=pos->next;
    pos->next=new_node;
    ++lp->size;
    if(pos==lp->tail)
        lp->tail=new_node;
    return lp;
}
```

函数 insert_list2 创建新结点时，新结点的数据域存放插入位置上结点的数据域（代码中有注释“//注意”），然后将插入位置上结点的数据域的值修改为待插入的新元素 d，最后将新结点放在插入位置之后。从逻辑上看，新元素插入在指定位置之前；而从物理上看，新结点插入在指定位置之后。例如在 1->2->3 组成的单链表的 2 之前插入 4，创建的新结点存

放2而不是4,再将元素2修改为4,此时的链表变成1->4->3,最后将新结点2放在4之后,形成1->4->2->3。值得注意的是,当插入位置是链表尾部时,虽然逻辑上链表的尾部没有变化,但是物理上出现了新的尾部,此时需要调整链表的尾指针。

本例提供了三个用于在指定位置插入元素的函数,三个函数都有其缺陷。函数insert_after_list虽然算法时间复杂度为$O(1)$,但此函数将新元素插入在指定位置之后,这与正常的操作习惯不符;函数insert_list1虽然完成了指定位置之前的插入操作,但其算法时间复杂度为$O(n)$;函数insert_list2虽然在$O(1)$时间复杂度的情况下,完成了指定位置之前的插入操作,但函数中出现了修改数据域内容的操作,如果链表存放的数据元素占用内存较大,对其内容进行修改付出的代价可能高于函数insert_list1中搜索pos之前位置的代价。总而言之,在单向链表任意位置插入元素的操作不论如何处理都存在一些弊端,解决这个问题的根本方法就是使用双向链表。

(2)删除操作

删除操作与插入操作类似,下面提供的三个完成删除操作的函数也存在类似问题。

```
List * erase_after_list(List * lp,Node * pos)
{
    if(pos==lp->tail||lp->size==0)
        return (void * )0;
    if(pos==(void * )0)
        return pop_front_list(lp);
    Node * del=pos->next;
    pos->next=del->next;
    --lp->size;
    if(del==lp->tail)
        lp->tail=pos;
    free(del);
    return lp;
}
```

函数erase_after_list用于删除lp指向链表中,pos指向位置之后的结点(注意不是删除pos指向的结点)。当链表为空或者pos等于链表尾(链表尾结点不存在后继结点)时,函数返回地址0,表示删除操作失败。当参数pos等于地址0时,调用函数pop_front_list删除链表首元结点,这样处理是因为首元结点不能表示为任何结点之后。另外,需要注意的是,当删除结点为尾结点时,需要调整尾指针tail的位置。

```
List * erase_list1(List * lp,Node * pos)
{
    if(pos==(void * )0||lp->size==0)
        return (void * )0;
    if(pos==lp->head)
        return pop_front_list(lp);
    Node * pre=lp->head;
    while(pre&&pre->next!=pos)
```

```
        pre=pre->next;
    if(pre)
    {
        pre->next=pos->next;
        if(pos==lp->tail)
            lp->tail=pre;
        --lp->size;
        free(pos);
    }
    return lp;
}
List * erase_list2(List * lp,Node * pos)
{
    if(pos==(void * )0||lp->size==0)
        return (void * )0;
    if(pos==lp->head)
        return pop_front_list(lp);
    if(pos==lp->tail)
        return pop_back_list_bad(lp);
    Node * del=pos->next;
    pos->data=del->data;
    pos->next=del->next;
    --lp->size;
    if(del==lp->tail)
        lp->tail=pos;
    free(del);
    return lp;
}
```

函数 erase_list1 和 erase_list2 都能完成删除参数 lp 指向链表中 pos 位置上元素的操作，这要比函数 erase_after_list 删除指定位置之后的元素更加合理。函数 erase_list1 为了搜索 pos 之前的结点，付出了 $O(n)$ 的时间复杂度；函数 erase_list2 的算法时间复杂度为 $O(1)$，但该函数是将 pos 之后的元素复制到 pos 位置后，删除了 pos 之后的结点，例如从 1->2->3->4 组成的链表中删除 2，首先将链表修改为 1->3->3->4，然后删除了第二个 3，链表变为 1->3->4。

6. 移除元素操作

```
int remove_if_list(List * lp,_Bool ( * con)(ADT))
{
    int count=0;
    while(lp->size&&con(lp->head->data))
    {
        ++count;
```

```
            pop_front_list(lp);
        }
        if(lp->size>1)
        {
            Node * pre=lp->head;
            while(pre->next)
            {
                if(con(pre->next->data))
                {
                    Node * del=pre->next;
                    pre->next=del->next;
                    free(del);
                    --lp->size;
                    ++count;
                    if(pre->next==0)
                        lp->tail=pre;
                }
                else
                    pre=pre->next;
            }
        }
        return count;
    }
```

函数 remove_if_list 用于删除链表中所有满足某个条件的元素。条件使用参数 con 表示,con 是一个一元判定函数的指针。将链表结点数据域当作参数调用 con 指向的函数,如果返回值为真,表示满足条件(需要删除),返回值为假,表示不满足条件(不需要删除)。函数的返回值表示删除元素的个数,代码中的变量 count 用于删除元素个数的统计。

考虑删除单向链表中的结点需要修改前驱结点的指针域,链表的首元结点没有前驱结点。函数 remove_if_list 中的第一个 while 循环,就是为了处理要删除的结点是首元结点的情况。之所以使用循环,是因为删除一个首元结点后的新首元结点还有可能需要删除。

while 循环结束后,如果链表中存在多于一个的元素,说明删除操作需要继续。临时的结点指针 pre 初始指向链表的首元结点,用于从链表首元结点的下一个结点开始向后搜索要删除的结点,直至搜索到链表尾。注意,使用 pre 搜索要删除结点时,每次检测 pre 指向结点的下一个结点的数据域是否满足 con 表示的条件,而不是使用 pre 指向结点的数据域。当发现需要删除的结点时,使用另一个临时指针 del 指向要删除的结点,pre 的指针域指向要删除结点的下一个结点,如果刚刚删除的结点是链表尾结点,还需要调整尾指针 tail。

7. 测试代码与总结

```
#include<stdio.h>
void print_list(const List * lp)
{
```

```
    Node * cur=lp->head;
    if(cur)
    {
        printf("链表存放 %d 个数据:%d",lp->size,cur->data);
        cur=cur->next;
        while(cur)
        {
            printf("-> %d",cur->data);
            cur=cur->next;
        }
        printf("\n");
    }
    else
        printf("空链表\n");
}
_Bool less10(int n)
{
    return n<10;
}
int main(void)
{
    int i;
    List list;
    init_list(&list);
    push_front_list(&list,1);
    push_front_list(&list,2);
    print_list(&list);
    push_back_list(&list,3);
    push_back_list(&list,4);
    print_list(&list);
    pop_front_list(&list);
    pop_back_list_bad(&list);
    print_list(&list);
    insert_after_list(&list,5,find_in_list(&list,3));
    insert_list1(&list,4,find_in_list(&list,5));
    insert_list2(&list,2,find_in_list(&list,3));
    print_list(&list);
    erase_after_list(&list,find_in_list(&list,4));
    erase_list1(&list,find_in_list(&list,3));
    erase_list2(&list,find_in_list(&list,4));
    print_list(&list);
    remove_if_list(&list,less10);
```

```
    print_list(&list);
    destroy_list(&list);
    return 0;
}
```

显示结果：

```
链表存放 2 个数据:2->1
链表存放 4 个数据:2->1->3->4
链表存放 2 个数据:1->3
链表存放 5 个数据:1->2->3->4->5
链表存放 2 个数据:1->2
空链表
```

正确输入本节提供的所有代码并包含适当的头文件，编译、运行程序，可得到如上的显示结果。

由于链表结构不支持随机访问，本例未提供类似于 geti、seti 那样访问线性表某个位置上元素的函数，即使提供此类函数，算法时间复杂度也不属于常数级，而与链表长度相关。如果需要频繁地随机访问线性表元素，应该选择顺序表，而不是链表。相对于顺序表，链表的优势主要包括两点：一是，动态扩展的时间效率和空间效率高，即不存在数据移动以及存储单元空闲的现象；二是，任何链表在给定位置上插入、删除数据时，都不会发生数据移动，特别是双向链表，结点的插入或删除操作的时间复杂度一定为 $O(1)$。

与顺序表一样，链表结构在真实的项目开发中十分常用，许多高级语言的模板库都提供了链表的实现(一般为双向链表)，例如，C++中的 list，Java 中的 LinkedList 等。链表适合应用在数据规模不确定或者需要频繁进行插入、删除元素的场合。

2.4 线性表的应用

线性表在工程中的应用几乎是无处不在的，本节提供的编程实例是使用基数排序算法为无符号整型数组排升序。之所以选择此例是为了示范在此应用中如何选择合适的线性表作为排序的辅助结构——“桶”。本节的重点是线性表的应用，而不是排序算法本身，关于排序算法后续章节会有详细介绍。

本节要实现的排序算法称为“基数排序”，属于分配式排序。以基数 10 为例，排序时需要借助 10 个线性表作为“桶”，然后将待排序数组中的数字根据相应规则放入桶中，再从桶中收回，重复“放入”“收回”若干次后，原数组排好升序。具体地说，第 1 次将待排序数组中的元素根据其个位的值放入对应桶中，例如 15 放入 5 号桶、150 放入 0 号桶等等，然后按照 0 到 9 的顺序把桶中的元素收回到一维数组中；第 2 次，根据待排序数组元素十位的值将其放入对应桶中，15 放入 1 号桶、150 放入 5 号桶等等，然后收回；第 3 次百位；第 4 次千位……直至处理过的位数等于待排序数组中所有元素的最高位，这时说明待排序数组已经排好升序。

根据上述排序算法，我们需要选择合适的线性结构作为“桶”。首先，待排序数组的长度