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。本章作为全书的绪论,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、单元教学目标

(1)描述和解释数据结构中的基本概念,理解数据结构、程序和编程语言之间的关系(二级,领会)。

(2)举例说明四类基本的逻辑关系(三级,应用)。

(3)设计实验,辨别 Python 中的几种数据结构属于哪种数据的存储结构(五级,设计)。

(4)分析数据的抽象数据类型类结构并给出代码描述(四级,分析)。

(5)分析算法的时间、空间复杂度(四级,分析)。

(6)提高要求:分析 Python 中的常用数据类型的数据存储结构和内存管理(四级,分析)。

三、单元重点与难点

重点:

(1)描述和解释数据结构中的基本概念,理解数据结构、程序和编程语言之间的关系。

(2)举例说明四类基本的逻辑关系,辨别具体应用时使用哪种数据结构。

(3)分析算法的时间、空间复杂度。

难点:

分析算法的时间、空间复杂度。

四、单元预习目标

理解并能够说出数据结构中的基本概念;理解数据结构、程序和编程语言之间的关系;能举例说明四类基本的逻辑关系;理解时间、空间复杂度的概念。

http://www.neubooks.com

预习测试

1. 算法的时间复杂度是关于(　　)的函数。

A. 问题的规模　　B. 变量

C. 问题的难度　　D. A和B

2. 算法能正确地实现预定功能的特性为算法的(　　)。

A. 正确性　　B. 易读性

C. 健壮性　　D. 高效性

3. 数据的物理结构主要包含(　　)这几种结构。

A. 顺序结构和链表结构　　B. 线性结构和非线性结构

C. 动态结构和静态结构　　D. 集合、线性结构、树形结构、图形结构

4. 数据在计算机内存中的表示是指(　　)。

A. 数据的存储结构　　B. 数据结构

C. 数据的逻辑结构　　D. 数据元素之间的关系

5. 数据结构课程中,数据的基本单位是(　　)。

A. 字节　　B. 位

C. 数据项　　D. 数据元素

6. 对算法效率的度量是(　　)。

A. 正确度和简明度　　B. 正确度和简明度

C. 高的速度和正确度　　D. 时间复杂度和空间复杂度

7. 计算机所处理的数据一般具备某种内在联系,这是指(　　)。

A. 数据和数据之间存在某种关系　　B. 元素和元素之间存在某种关系

C. 元素内部存在某种关系　　D. 数据项和数据项之间存在某种关系

8. 表示学生会的组织结构可以使用(　　)这种逻辑结构。

A. 集合　　B. 线性结构

C. 树形结构　　D. 图状结构

9. 数据结构课程中,数据的最小单位是(　　)。

A. 字节　　B. 位

C. 数据项　　D. 数据元素

10. 描述亲友关系可以使用(　　)这种逻辑结构。

A. 集合　　B. 线性结构

C. 树形结构　　D. 图状结构

http://www.neubooks.com

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。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,数据结构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:

http://www.neubooks.com

(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。

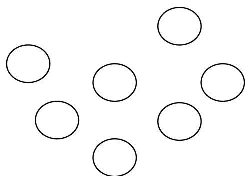


图1.1 集合结构

(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,其他各结点都有唯一的直接后继。数据元素之间是一对一的关系,如图1.2所示。

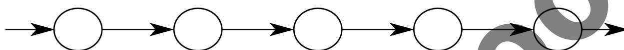


图1.2 线性结构

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。

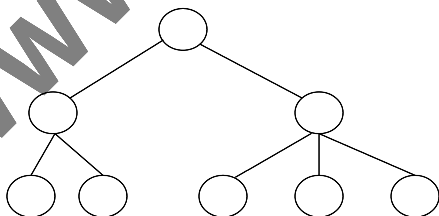


图1.3 树状结构

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。

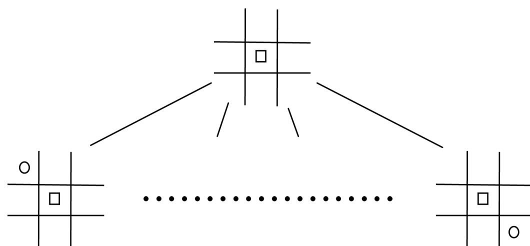


图1.4 格局

http://www.neubooks.com

(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。

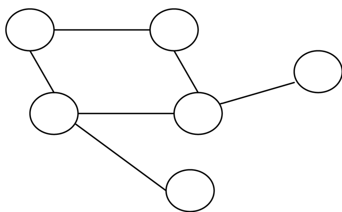


图 1.5　图状结构

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。算法需要满足以下五个性质:

(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):算法的每一个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:

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

http://www.neubooks.com

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+=1;
(b)for i in range(0,n):
        x+=1
(c)for i in range(0,n):
        for j in range(0,i+1):
            x+=1
```

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。

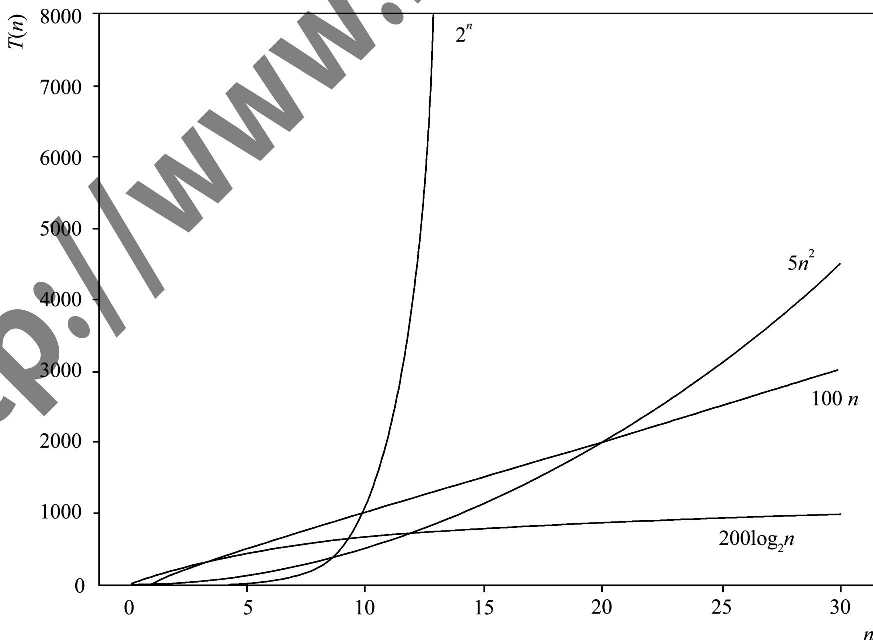


图 1.6 各种数量级的时间复杂度

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。

http://www.neubooks.com

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的作用。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。

(1)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"和"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。

(2)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。

(3)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和计算相关度。

(4)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

http://www.neubooks.com

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。

(5)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。

课后作业

一、选择题

1. 函数 *f*1 的时间复杂度为(　　),空间复杂度为(　　)。

```
def f1(lst):
    for i in range(len(lst)//2):
        lst[i],lst[len(lst)-i-1] = lst[len(lst)-i-1], lst[i]
```

A. $O(1)$　　　　B. $O(\log n)$

C. $O(n)$　　　　D. $O(n^2)$

2. 函数 fib 的时间复杂度为(　　),空间复杂度为(　　)。

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

A. $O(2^n)$　　　　B. $O(\log n)$

C. $O(n)$　　　　D. $O(n^2)$

http://www.neubooks.com

3.下面语句段的时间复杂度为(　　),空间复杂度为(　　)。

```
i = 0
c = 0
while i < n:
    i *= 2
    c += 1
```

A. $O(1)$　　B. $O(\log n)$

C. $O(n)$　　D. $O(n^2)$

二、分析题

1.分析 Python 中 list 的存储结构。

2.分析 Python 中 dict 的存储结构。

3.分析 Python 中 set 的存储结构。

三、简答题

用 Python 中的数据类型为每种存储结构举例。

http://www.neubooks.com

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,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、索引存储及实现、链式存储及实现,还介绍了应用实例。

二、单元学习目标

(1)描述和解释线性表、顺序表、链表的概念,以及顺序表、单链表、双链表、循环单链表的操作,各种操作算法步骤(二级,领会)。

(2)举例说明顺序表和链表的应用(三级,应用)。

(3)能够分析给出的应用场景,设计适当的抽象类型,应用数据结构,并给出代码实现(五级,综合)。

(4)提高要求:设计循环双链表的操作算法(五级,综合)。

三、单元重点与难点

重点:

(1)描述和解释线性表、顺序表、链表的概念。

(2)顺序表、链表操作和 Python 实现。

(3)顺序表、链表应用。

难点:

链表的操作和 Python 实现。

四、单元预习目标

自主学习掌握线性表、顺序表、链表的概念,能够将它们的应用进行举例;掌握顺序表、单链表可执行的操作及其算法实现。

http://www.neubooks.com

预习测试

1. 关于线性表下面说法正确的是(　　)。

A. 每个元素都有一个直接前驱和一个直接后继

B. 线性表中至少要有一个元素

C. 表中诸元素的排列顺序必须是由小到大或由大到小

D. 除第一个和最后一个元素,其余每个元素都有一个且仅有一个直接前驱和直接后继

2. 在线性表的下列运算中,不改变数据元素之间结构关系的运算是(　　)。

A. 插入　　B. 删除

C. 排序　　D. 定位

3. 线性表是具有 n 个(　　)的有限序列($n>0$)。

A. 字符串　　B. 整数

C. 数据元素　　D. 数据项

4. 在一个长度为 n 的顺序表中,在第 i 个元素($0\leqslant i\leqslant n$)之前插入一个新元素时需向后移动(　　)个元素。

A. $n-i$　　B. $n-i+1$

C. $n-i-1$　　D. i

5. 若某线性表中最常用的操作是取第 i 个元素和找第 i 个元素的前趋元素,则采用(　　)存储方式最节省时间。

A. 单链表　　B. 双链表

C. 单向循环链表　　D. 顺序表/索引表

6. 对于索引存储的线性表,访问结点和增加、删除结点的时间复杂度为(　　)。

A. $O(n)$　$O(n)$　　B. $O(n)$　$O(1)$

C. $O(1)$　$O(n)$　　D. $O(1)$　$O(1)$

7. 若某线性表最常用的操作是存取任一指定序号的元素和在最后进行插入和删除运算,则利用(　　)存储方式最节省时间。

A. 顺序/索引表　　B. 双链表

C. 带头结点的双循环链表　　D. 单循环链表

8. 线性表长度为 n,以链接方式存储时访问第 i 位置元素的时间复杂度为(　　)。

A. $O(i)$　　B. $O(1)$

C. $O(n)$　　D. $O(i-1)$

9. 清空索引表时执行的操作是(　　)。

A. 将索引表中所有元素值设置为 0　　B. 将数据表中所有元素值设置为 None

C. 删除索引表中所有元素　　D. 记录索引中当前元素个数为 0

10. 线性表若采用链式存储结构时,要求内存中可用存储单元的地址(　　)。

A. 必须是连续的　　B. 部分地址必须是连续的

C. 一定是不连续的　　D. 连续或不连续都可以

http://www.neubooks.com

11. 以下说法正确的有(　　)。

A. 单链表从任何一个结点出发,都能访问到所有结点。

B. 对线性表中的数据元素只能进行访问,不能进行插入和删除操作。

C. 线性表的长度 n 就是表中数据元素的个数,当 $n=0$ 时,称为空表。

D. 线性表中的每个结点都至少有一个前驱结点和后继结点。

12. 链表不具有的特点是(　　)。

A. 插入、删除不需要移动元素　　B. 可随机访问任一元素

C. 不必事先估计存储空间　　D. 所需空间与线性长度成正比

13. 当(　　)时需要为单链表记录当前元素个数。

A. 应用中会频繁统计元素个数　　B. 应用中从不统计元素个数

C. 应用中经常访问尾节点　　D. 应用中经常访问头结点

14. 当(　　)时需要为单链表记录指向尾节点的指针。

A. 应用中频繁向链表尾部插入元素　　B. 应用中频繁删除链表尾部的元素

C. 应用中频繁向链表尾部之前插入元素　　D. 以上都对

15. 对于带头结点的单链表其优势在于(　　)。

A. 可以降低算法的时间复杂度　　B. 可以在某些情况下简化算法

C. 可以降低算法的空间复杂度　　D. 以上都对

16. 对于单链表,删除一个指定元素时,一般应找到该元素的(　　)。

A. 前驱　　B. 后继

C. 本身　　D. 以上都对

17. 在遍历不带头结点的单链表,循环继续的条件应该是(　　),循环条件初始化为p=head。

A. p is not None　　B. not p

C. p. Next　　D. not p. Next

18. 链表不具有的特点是(　　)。

A. 插入、删除不需要移动元素　　B. 可随机访问任一元素

C. 不必事先估计存储空间　　D. 所需空间与线性长度成正比

19. 不带头结点的单链表头指针为 head,链表为空的判定条件是(　　)。

A. head is None　　B. head. next is None

C. head. next==NULL　　D. head==NULL

20. 在一个单链表中,结点指针域为 next,已知 q 所指结点是 p 所指结点的前驱结点,若在 q 和 p 之间插入 s 结点,则执行语句(　　)。

A. s. next=p. next
　p. next=s. next

B. p. next=s
　s. next=p

C. q. next=s
　s. next=p

D. p. next=s
　s. next=q

21. 在一个单链表中,结点指针域为 next,若 p 所指结点不是最后结点,在 p 之后插入 s

http://www.neubooks.com

结点，则应执行语句(　　)。

A. s. next＝p
　p. next＝s

B. s. next＝p. next
　p. next＝s

C. s. next＝p. next
　p＝s

D. p. next＝s
　s. next＝p

22. 在一个单链表中，结点指针域为 next，p 指向的结点不是尾结点，若删除 p 所指结点的后续结点，则应执行语句(　　)。

A. p. next＝p. next. next

B. p＝p. next p. next＝p. next. next

C. p. next＝p. next

D. p＝p. next. next

23. 若某线性表最常用的操作是存取任一指定序号的元素和在最后进行插入和删除运算，则利用(　　)存储方式最节省时间。

A. 顺序表或索引表

B. 双链表

C. 带头结点的双循环链表

D. 单循环链表

http://www.neubooks.com

2.1 线性表的基本定义

线性表是最简单、最基本、程序设计中最常用的数据结构。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。

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,并给出抽象类型的定义见表 2.1。

表 2.1 List 的 Python 抽象类型定义

成员方法	描述
init(self)	构造函数,List 对象初始化为空的 list
del(self)	析构函数,销毁 List 对象
clear(self)	清空 List 对象,使其中不包含任何元素
empty(self)	判断是否为空表
length(self)	求线性表长度
insert(self,i,value)	在线性表第 i 个元素之前插入元素 value
erase(self,i)	删除线性表第 i 个元素
remove(self,value)	删除线性表中所有值等于 value 的元素

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。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索引存储和链式存储。

2.2 线性表的索引表示和实现

本节介绍线性表的索引表示方法,并给出 Python 语言实现。索引表示强调数据元素的索引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。

http://www.neubooks.com

2.2.1　索引表的基本概念和特点

线性表的索引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引用。假设线性表中对每个数据元素的引用占用 l 个存储单元，第 i 个元素引用的地址是 $\mathrm{LOC}(a_i)$，那么第 $i+1$ 个元素引用的地址 $\mathrm{LOC}(a_{i+1})$ 等于 $\mathrm{LOC}(a_i)+l$。

假设 n 个元素的线性表采用 $(a_1, a_2, a_3, \cdots, a_n)$ 表示，表中首个元素引用的地址 $\mathrm{LOC}(a_1)$ 称为该线性表的基地址，那么，第 i 个数据元素引用的地址 $\mathrm{LOC}(a_i)$ 等于 $\mathrm{LOC}(a_1)+(i-1)\times l$。

由于本教材采用 Python 语言实现各种数据结构和算法，Python 语言 list 下标从 0 开始计算。我们总是将数据结构中的"第 1 个"元素引用存放在数组下标为 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 $(a_1, a_2, a_3, \cdots, a_n)$，而是表示为 $(a_0, a_1, a_2, \cdots, a_{n-1})$。在这种情况下，线性表 $(a_0, a_1, a_2, \cdots, a_{n-1})$ 的基地址为 $\mathrm{LOC}(a_0)$，第 i 个数据元素引用的地址 $\mathrm{LOC}(a_i)$ 等于 $\mathrm{LOC}(a_0)+i\times l$。

线性表的这种计算机表示称为线性表的索引存储结构，通常称这种存储结构的线性表为索引表，如图 2.1 所示。

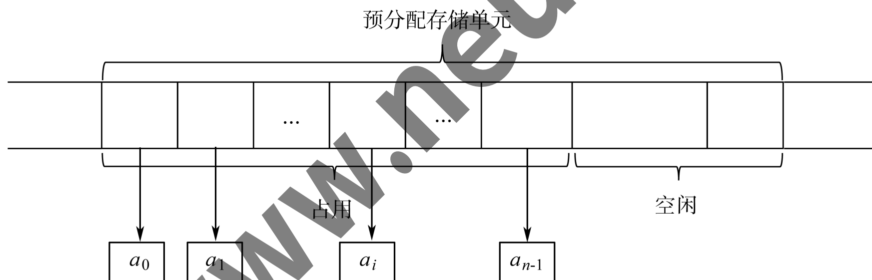


图 2.1　索引表存储示意图

由于索引表中的元素引用是连续存储的，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。存取索引表中任一元素 a_i 的时间复杂度为 $O(1)$。

考虑向索引表插入数据元素和从索引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 $(a_0, a_1, \cdots b, a_i, \cdots, a_{n-1})$。如果线性表采用索引存储，如图 2.1 所示，需要将索引表中，编号从第 i 至第 $n-1$ 位置上的 $n-i$ 个元素引用向后移动 1 个存储单元。n 个元素的索引表有 $n+1$ 个可能的插入位置，向这些位置插入元素依次需要移动 $n, n-1, \cdots, 1, 0$ 个元素引用。假设在表的任意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 $n/2$ 个元素引用，时间复杂度为 $O(n)$。类似的，删除索引表中的第 i 个元素，需要将编号从第 $i+1$ 至 $n-1$ 位置上的 $n-i-1$ 个元素引用向前移动 1 个存储单元。n 个元素的索引表有 n 个可能的删除位置，从这些位置删除元素引用依次需要移动 $n-1, n-2, \cdots, 1, 0$ 个元素。假设从表的任意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 $O(n)$。

通过对索引表特点的分析，可以看出，索引表适合应用在频繁访问元素，但很少插入、删

http://www.neubooks.com

除元素或仅在尾部插入、删除元素的场合。

2.2.2 索引表的实现

本小节使用 Python 语言中的 list 实现一个可自动增长的索引表结构及其基本操作。定义类 List 实现索引表，成员定义同表 2.1。为帮助读者理解索引表操作的原理，其他的插入删除操作只调用 list 的 append()和 pop(-1)方法实现，不调用 list 其他的内置方法。

(1)初始化、清空和销毁操作

首先看初始化操作。

```
def _init_(self):
    self._lst = []
```

使用构造函数_init_完成初始化功能，初始化后的索引表为空表。

接下来看清空操作。

```
def clear(self):
    while len(self._lst):
        self._lst.pop(-1)
```

函数 clear 完成逻辑上清空一个索引表的功能。len(self._lst)非零，表示索引表中还有元素，调用 pop(-1)删掉索引表中的最后一个元素，删掉了元素同时也删掉索引表中对元素的引用。

再来看销毁操作。

```
def _del_(self):
    self.clear()
    del self._lst
```

析构函数中销毁当前的 List 对象。首先调用 clear 清空索引表，然后删掉索引表本身。当用户要删除 List 对象，就会调用析构函数，释放每个元素之后释放掉 list 结构本身。

(2)读写元素操作

读取下标编号为 i 的元素的代码如下，重载读_getitem_函数实现读元素操作，如果给定的编号在合法范围 0 到元素个数-1 之间，就返回元素的值；否则返回 None。

```
def _getitem_(self, i):
    if index >=0 and index < len(self._lst):
        return self._lst[i]
    else:
        return None
```

修改下标编号为 i 的元素的代码如下，写操作函数_setitem_使用参数 value 的值覆盖线性表第 i 个元素值，成功返回 True；i 的范围非法则返回 False。

```
def _setitem_(self, i, value):
    if index >= 0 and index < len(self._lst):
        self._lst[index] = value
        return True
    else:
        return False
```

http://www.neubooks.com

虽然在 Python 语言中，永远可以使用“lst[i]”的形式访问索引表 lst 中的第 i 个数据元素。这里仍然为自定义的类提供了_getitem_和_setitem_函数，分别用于索引表元素的安全读写，实现属性的索引式存取。

(3)空间扩展

空间扩展操作是向索引表插入元素操作的辅助操作，如果索引表预先分配的存储单元用尽，应该对索引表进行空间扩展。Python 中的 list 底层使用下面的 C 语言代码实现的。

```
typedef struct {
  PyObject_VAR_HEAD
  PyObject * * ob_item;
  Py_ssize_t allocated;
}PyListObject;
```

根据 Python 的官方说明，在建立空表(或者很小的表)时，系统分配一块能容纳 8 个元素索引的存储区；在执行插入操作时，如果元素存储区满就将存储区容量扩展到原来的 4 倍。但如果此时的表已经很大(目前的阈值为 50000)，则改变策略，采用加一倍的方法。引入这种改变策略的方式，是为了避免出现过多空闲的存储位置。

(4)尾部插入元素和删除元素操作

对于索引表来说，在尾部插入和删除元素是常见的操作，其代码如下。调用 append，可以实现追加的功能。

```
def push_back(self,value):
    self._lst.append(value)
```

删除尾部元素的代码如下。采用 pop 方法，可以指定删除元素的索引，如果删掉最后一个元素，参数为－1。

```
def pop_back(self):
if len(self._lst):
    return self._lst.pop(－1)
else:
    return None
```

如果不考虑空间扩展操作，在索引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 pop_back 当索引表为空时，删除操作失败，返回 False；索引表非空就删掉最后一个元素，返回 True。

(5)任意位置插入元素和删除元素操作

下面是在指定位置插入元素操作的代码。

```
def insert(self,  i,  data):
    if i >=0 and i <= len(self._lst):
        self._lst.append(0)
        for j in range(len(self._lst)-1, i, -1):
            self._lst[j] = self._lst[j-1]
        self._lst[i] = data
        return True
    else:
```

http://www.neubooks.com

```
            return False
```

函数 insert 的功能是向索引表的第 i 个位置上插入值 value，成功插入返回 True，失败返回 False。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 个元素的索引表，合理的插入位置有 $n+1$ 个，参数 i 合理的取值是 $0\sim n$。当参数 i 取合理值，向索引表尾部增加一个元素，从后向前，将索引表中下标范围在 $n-1$ 至 i 的所有元素的索引都向后移动一位，然后把新元素放在 i 号位置上。

以下是删除指定位置元素的代码。

```
def erase(self, i):
    if i>=0 and i < len(self._lst):
        for j in range(i,len(self._lst)-1):
            self._lst[j]=self._lst[j+1]
        self._lst.pop(-1)
        return True
    else:
        return False
```

函数 erase 的功能是删除当前索引表的第 i 个位置上的元素，成功删除返回 True，失败返回 False。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 个元素的索引表，合理的删除位置有 n 个，参数 i 合理的取值是 $0\sim n-1$。删除 i 号元素，将 $i+1$ 至 $n-1$ 号索引全部向前移动一位，再删掉最后一个元素。

在索引表的任意位置插入(删除)元素，会造成操作位置之后的元素索引向后(向前)移动。

(6)移除元素操作

代码如下：

```
def remove(self, data):
    r = w = count = 0
    while r < len(self._lst):
        if self._lst[r] == data:
            count += 1
        else:
            self._lst[w] = self._lst[r]
            w += 1
        r += 1
    for i in range(count):
        self._lst.pop(-1)
    return count
```

函数 remove 可用于删除索引表中所有值等于参数 data 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 分别代表读写元素索引位置，如果当前元素值与 data 不相等，发生写操作，将 r 对应元素引用复制到 w 对应位置，w 和 r 同时递增；否则计数增加，r 递增，w 不变。最后删掉索引表尾部的 count 个元素。

http://www.neubooks.com

(7)测试代码与总结

```
class List:
    … …
    def output(self):
        print("size:{}".format(len(self._lst)),"elements:", end='')
        for i in self._lst:
            print(i, end='')
        print()
def odd(n):
    return n % 2 == 0

if _name_ == '_main_':
    lst = List()
    for i in range(10):
        lst.push_back(i+1)
        lst.output()
    for i in range(6):
        lst.pop_back()
        lst.output()
    lst[1]=100
    lst.output()
    lst.insert(1, 200)
    lst.output()
    lst.erase(2)
    lst.output()
    lst.remove_if(odd)
    lst.output()
    lst.remove(200)
    lst.output()
    lst.clear()
    lst.output()
    del lst
```

显示结果：

```
size:1 elements: 1
size:2 elements: 1 2
size:3 elements: 1 2 3
size:4 elements: 1 2 3 4
size:5 elements: 1 2 3 4 5
size:6 elements: 1 2 3 4 5 6
size:7 elements: 1 2 3 4 5 6 7
size:8 elements: 1 2 3 4 5 6 7 8
size:9 elements: 1 2 3 4 5 6 7 8 9
```

http://www.neubooks.com

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

正确输入本节提供的所有代码后编译、运行程序,可得到如上的显示结果。

线性表结构在真实的项目开发中十分常用,Python中提供基于索引结构的线性表实现,有些高级语言提供顺序结构表的实现,相关算法的思路与索引表类似。顺序表和索引表结构都适用于那些频繁进行随机访问元素、插入和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,为避免大量数据的频繁移动,应在创建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

线性表的索引存储结构的特点是逻辑上相邻的两个数据元素的引用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,这一特点也铸成索引表的弱点: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,因此它没有索引表插入或删除元素时需要移动元素的弱点,但也失去了索引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,除了数据元素本身的信息外,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引用域。采用链式存储结构表示的 n 个元素的线性表 $(a_0, a_1, a_2, \cdots, a_{n-1})$,就是由 n 个结点链成一条链表。

结点引用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;结点引用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。如图

http://www.neubooks.com

2.2(a)所示是一个单链表的部分结点,如图 2.2(b)所示是一个双链表的部分结点。

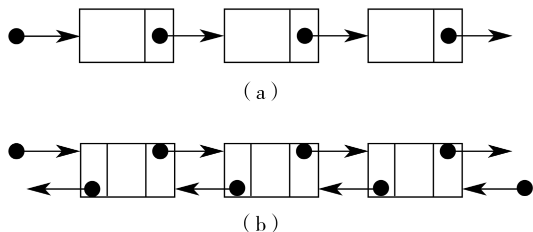


图 2.2　单链表和双链表

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,沿着各个结点引用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,至少需要一个引用变量,我们将这个记录首元结点地址的引用变量称为链表的头引用,如果头引用为空(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,引用域存放首元结点位置信息的结点,该结点称为头结点。如图 2.3 所示使用单链表展示了头引用和头结点的差别,图 2.3(a)是头引用的示意图,图 2.3(b)是头结点的示意图。双链表的情况与单链表类似。

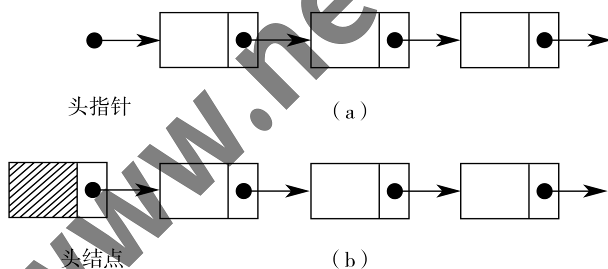
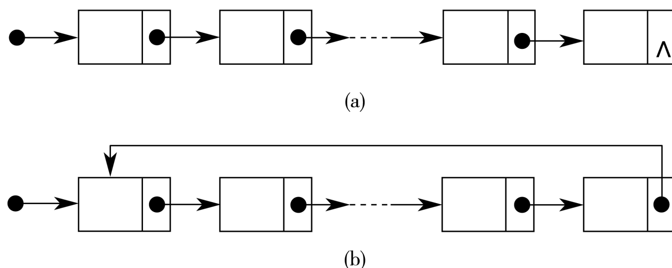


图 2.3　头引用和头结点

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引用存放尾结点地址,但在链表结构中,尾引用并不是像头引用(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引用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,如果引用域存放内存地址,最后一个结点引用域的特殊值可设置为 NULL(地址 0),如果引用域存放数组下标(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),最后一个结点引用域的特殊值可选择－1 等非法数组下标。对于循环链表来说,最后一个结点引用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,双链表结点的引用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)的前驱引用应设置为空(非循环链表),或令其指向尾结点(循环链表)。如图 2.4 所示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,图 2.4(a)是非循环单链表的

http://www.neubooks.com

示意图,图2.4(b)是循环单链表的示意图。



(a)

(b)

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Python语言实现一个带头尾引用、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)结点类型定义及创建和销毁

```
class Node:
    def _init_(self, data=None):
        self._data = data
        self._next = next_node
    def _del_(self):
        if self._data:
            del self._data
        if self._next:
            del self._next

    def _get_next(self):
        return self._next
    def _set_next(self, next_node):
        self._next = next_node
    Next = property(_get_next, _set_next)

    def _get_data(self):
        return self._data
    def _set_data(self, data):
        self._data = data
    Data = property(_get_data, _set_data)
```

类型Node表示单链表结点。其中,字段_data表示数据域,_next表示引用域,引用域用于存放对后继结点对象的引用。_init_()是构造函数,根据给定的参数创建新结点,将结点数据域data设置为参数data的取值(如果没有给出该参数值,则默认为None),将引用域设置为None(该结点暂时不属于任何链表);_del_()是析构函数,可以销毁当前结点对象。

http://www.neubooks.com

(2)初始化、清空和销毁操作

```
class LinkedList:
    def _init_(self):
        self._head = None
        self._rear = None
        self._size = 0
    def _get_size(self):
        return self._size
    Size = property(_get_size)
```

LinkedList 定义了一个单链表,包括_head、_rear 和_size 三个字段,分别表示链表的表头引用、表尾引用和元素个数。函数_init_()是初始化函数,否则创建一个空链表,具体做法是将 head 和 rear 都设置为 None,表示链表长度的变量_size 也设置为 0。Size 是类 LinkedList 的只读属性,获取链表中元素个数。

```
def clear(self):
    while self._head:
        p = self._head
        self._head = p.Next
        del p
    self._head = self._rear = None
    self._size = 0
```

clear 是类 LinkedList 的成员方法,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内存,然后将_head、_rear 置为 None,将_size 置为 0。

```
def _del_(self):
    self.clear()
```

del()是析构函数,负责销毁一个链表,其中调用了函数 clear()。该函数在用户 delLinkedList 对象时会自动调用,对象被销毁后不能再被使用。

(3)在两端插入元素和删除元素操作

首先是在链表头部插入元素的代码。

```
def push_front(self, data):
    node = Node(data)
    node.Next = self._head
    self._head = node
    if not node.Next:
        self._rear = node
    self._size += 1
```

LinkedList 的成员函数 push_front()用于在当前链表对象的头部插入数据 data。函数中首先创建新结点,对新结点的引用存入临时变量 node;然后让新结点的引用域指向链表原来的首元结点,链表头引用指向新结点,即将新结点插入链表的头部;判断如果链表尾引用为空(表示原来的链表是空链表),此时新结点既是首元结点,又是尾结点,所以需要使尾引用也要指向新结点;最后调整表示链表长度的变量。注意,本函数的第 2 个参数 data 表示要

http://www.neubooks.com

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。

接下来是在链表尾部插入元素的代码。

```
def push_back(self, data):
    node = Node(data)
    if self._rear:
        self._rear.Next = node
    else:
        self._head = node
    self._rear = node
    self._size += 1
```

成员函数 push_back()用于在当前链表对象的尾部插入数据 data。与函数push_front()类似,需要根据参数 data 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引用域指向新结点,否则令头引用指向新结点(新结点既是尾结点,又是首元结点);最后,调整链表尾引用和表示长度的变量。注意,如果类型 List 中没有成员变量_rear 表示尾引用,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。

然后是删除链表头部元素的代码。

```
def pop_front(self):
    if not self._head:
        return False
    else:
        node_del = self._head
        self._head = node_del.Next
        del node_del
        self._size -= 1
        if not self._size:
            self._rear = None
        return True
```

成员函数 pop_front()用于删除当前链表中对象的首元结点。当链表为空时,函数返回False,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,先定义临时引用指向链表的首元结点,链表头引用指向原来首元结点的下一个结点,然后删除原有首元结点,并调整变量 size。值得注意的是,如果原有链表仅有一个结点,则删除唯一结点后需要将尾引用赋值为 None。

最后是删除链表尾部元素的代码。

```
def pop_back_bad(self):
    if not self._size:
        return False
    node_del = self._rear
    node_new_rear = self._head
```

http://www.neubooks.com

```
        if node_del == node_new_rear:
            self._rear = self._head = None
        else:
            while node_new_rear.Next != node_del:
                node_new_rear = node_new_rear.Next
            node_new_rear.Next = None
            self._rear = node_new_rear
        del node_del
        self._size -= 1
        return True
```

成员函数 pop_back_bad()用于删除当前链表对象中的尾结点。链表为空时，返回 False，表示删除失败；函数在判断链表非空后，先定义临时引用指向要删除的结点，同时定义另一个临时引用指向链表的头；然后，判断链表是否仅有 1 个结点，若是，则令头引用和尾引用设置为 None（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，并调整它的引用域为 None，同时令尾引用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_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。其他三个函数 push_front()、push_back()和 pop_front()所使用的算法与链表长度无关，而 pop_back_bad()函数操作的链表越长，该函数执行得越慢。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引用，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引用的存储空间，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

```
    def search(self, data):
        cur = self._head
        while cur and cur.Data != data:
            cur = cur.Next
        return cur
```

成员函数 serach()用于在当前链表对象中，查找首个值等于参数 data 的元素。并返回结点对象的引用，返回结点引用可供使用者修改结点信息。如果当前链表对象中不存在数据域与参数 data 等值的结点，函数返回 None。代码中定义结点引用 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，函数返回当前引用 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 None。

（5）在指定位置插入和删除元素操作

首先是在指定位置插入元素的代码。

```
    def insert_after(self, data, node):
        if not node or self._size == 0:
            self.push_front(data)
```

http://www.neubooks.com

```
        elif node == self._rear:
            self.push_back(data)
        else:
            new_node = Node(data)
            new_node.Next = node.Next
            node.Next = new_node
            self._size += 1
```

成员函数 insert_after()用于在当前链表对象中,参数 node 引用的结点之后,插入元素 data。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,即插入到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()将在参数 node 为 None 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引用域,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,这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结点之上(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),而不是插入在指定结点之后。为此,本例另外提供了函数 insert_1 用于正常的插入操作。

```
    def insert_1(self, data, node):
        if not node or not self._size:
            self.push_back(data)
        if node == self._head:
            self.push_front(data)
        new_node = Node(data)
        node_pre = self._head
        while node_pre and node_pre.Next != node:
            node_pre = node_pre.Next
        if node_pre:
            new_node.Next = node
            node_pre.Next = new_node
            self._size += 1
```

成员函数 insert_1()需要将元素 data 插入在指定结点 node 之前。有 n 个元素的线性表,合法的插入位置有 $n+1$ 个,插入到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_1 将在参数 node 为 None 或者链表为空的情况下,将新元素放在链表尾部。

函数 insert_1 中定义的引用 node_pre 用于寻找并保存 node 位置之前的那个结点,新结点将被插入在 node_pre 之后,node 之前。为了寻找 node 之前的结点,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_2。

```
    def insert_2(self, data, node):
        if not node or not self._size:
```

http://www.neubooks.com

```
            self.push_back(data)
        if node == self._head:
            self.push_front(data)
        new_node = Node(node.Data) #注意
        node.Data = data
        new_node.Next = node.Next
        node.Next = new_node
        self._size += 1
        if node == self._rear:
            self._rear = new_node
```

成员函数 insert_2()创建新结点时，新结点的数据域存放插入 node 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 data，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引用。

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()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_1 虽然完成了在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_2 虽然在 $O(1)$ 时间复杂度的情况下，完成了在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_1 中搜索 node 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。

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。

```
def erase_after(self, node):
    if node == self._rear or not self._size:
        return False
    if not node:
        return self.pop_front()
    node_del = node.Next
    node.Next = node_del.Next
    self._size -= 1
    if node_del == self._rear:
        self._rear = node
    del node_del
    return True
```

成员函数 erase_after()用于删除当前链表中 node 之后的结点（注意不是删除 node 指向的结点）。当链表为空或者 node 等于链表尾（链表尾结点不存在后继结点）时，函数返回

http://www.neubooks.com

False,表示删除操作失败。当参数 node 等于 None 时,调用函数 pop_front 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,需要调整尾引用尾结点的位置。

```
def erase_1(self, node):
    if not node or not self._size:
        return False
    if node == self._head:
        return self.pop_front()
    node_pre = self._head
    while node_pre and node_pre.Next != node:
        node_pre = node_pre.Next
    if node_pre:
        node_pre.Next = node.Next
        if node == self._rear:
            self._rear = node_pre
        self._size -= 1
        del node
    return True
def erase_2(self, node):
    if not node or not self._size:
        return False
    if node == self._head:
        return self.pop_front()
    if node == self._rear:
        return  self.pop_back_bad()
    node_del = node.Next
    node.Data = node_del.Data
    node.Next = node_del.Next
    self._size -= 1
    if node_del == self._rear:
        self._rear = node
    del node_del
    return True
```

成员函数 erase_1()和 erase_2()都能完成删除当前链表对象中 node 上元素的操作,这要比函数 erase_after 删除指定位置之后的元素更加合理。函数 erase_1 为了搜索 node 之前的结点,付出了 $O(n)$的时间复杂度;函数 erase_2 的算法时间复杂度为 $O(1)$,但该函数是将 node 之后的元素复制到 node 位置后,删除了 pop 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。

http://www.neubooks.com

(6)移除元素操作

```
def remove_if(self, con):
    count = 0
    while  self._size and con(self._head.Data):
        count += 1
        self.pop_front()
    if self._size > 1:
        node_pre = self._head
        while node_pre:
            if con(node_pre.Next.Data):
                node_del = node_pre.Next
                node_pre.Next = node_del.Next
                del node_del
                self._size -= 1
                count += 1
                if not node_pre.Next:
                    self._rear = node_pre
                else:
                    node_pre = node_pre.Next
    return count
```

成员函数 remove_if()用于删除当前链表对象中所有满足某个条件的元素。条件用参数 con 表示,con 是一个一元判定函数的引用。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 用于统计删除元素的个数。

考虑删除单向链表中的结点需要修改前驱结点的引用域,链表的首元结点没有前驱结点。函数 remove_if()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。临时的结点引用 node_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 node_pre 搜索要删除的结点时,每次检测 node_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 node_pre 指向结点的数据域。当发现需要删除的结点时,使用另一个临时引用 node_del 指向要删除的结点,node_pre 的引用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,还需要调整尾引用_rear。

(7)测试代码与总结

```
def output(self):
    cur = self._head
    if cur:
        print("链表存放{0}个元素:{1}".format(\
              self._size,cur.Data), end='')
```

http://www.neubooks.com

```
        cur = cur.Next
        while cur:
            print("->%d" % cur.Data, end=")
            cur = cur.Next
        print()
    else:
        print("空链表")
less_than_10(data):
    return  data < 10
def main():
    lst = LinkedList()
    lst.push_front(1)
    lst.push_front(2)
    lst.output()
    lst.push_back(3)
    lst.push_back(4)
    lst.output()
    lst.pop_front()
    lst.pop_back_bad()
    lst.output()
    lst.insert_after(5,lst.search(3))
    lst.insert_1(4,lst.search(5))
    lst.insert_2(2,lst.search(3))
        lst.output()
    lst.erase_after(lst.search(4))
    lst.erase_1(lst.search(3))
    lst.erase_2(lst.search(4))
    lst.output()
    lst.remove_if(less_than_10)
    lst.output()
    del lst

if __name__ == '__main__':
    main()
```

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，编译、运行程序，可得到如上的显示结果。

http://www.neubooks.com