



普通高中教科书

信息技术

选择性必修1 数据与数据结构

 华东师范大学出版社

普通高中教科书

信息技术

选择性必修 1

数据与数据结构



华东师范大学出版社

· 上海 ·

总 主 编：李晓明

副总主编：赵 健

本册主编：李晓明

编写人员(按姓氏笔画排序)：

白晓琦 李晓明 周 刚 顾秋辉

责任编辑：竺 笑

美术设计：储 平

普通高中教科书 信息技术 选择性必修 1 数据与数据结构

上海市中小学(幼儿园)课程改革委员会组织编写

出版发行 华东师范大学出版社(上海市中山北路 3663 号)

印 刷 上海昌鑫龙印务有限公司

版 次 2021 年 3 月第 1 版

印 次 2021 年 3 月第 1 次

开 本 890 毫米×1240 毫米 1/16

印 张 9.75

字 数 174 千字

书 号 ISBN 978-7-5760-0549-3

定 价 12.10 元

版权所有·未经许可不得采用任何方式擅自复制或使用本产品任何部分·违者必究

如发现内容质量问题,请拨打电话 021-60821714

如发现印、装质量问题,影响阅读,请与华东师范大学出版社联系。电话:021-60821711

全国物价举报电话:12315

声明 按照《中华人民共和国著作权法》第二十五条有关规定,我们已尽量寻找著作权人支付报酬。著作权人如有关于支付报酬事宜可及时与出版社联系。

本册教材图片提供信息:

本册教材中的部分图片由全景网、视觉中国等图片网站提供。

致同学们

亲爱的同学们：

数据与数据结构，是计算机科学，尤其是程序设计中的两个核心概念。数据类型的丰富性，让我们得以方便地表达复杂的事物；数据结构的精巧性，让我们得以高效地实现优美的算法。

按照《普通高中信息技术课程标准(2017 版)》的规定，“数据与数据结构”是一个选择性必修模块。本册适合于那些将来有可能选择计算机科学、人工智能、大数据和物联网等相关专业的同学学习。该模块的学习有助于同学们进一步提高对这些专业的兴趣进而坚定选择这些专业的信心。

另一方面，对“数据与数据结构”基本知识的理解和适当掌握，也有超越应试的意义。因为它是计算思维的一部分，在基于互联网和大数据之上的信息社会中，计算无时无刻不在，对其核心概念与过程的熟悉和理解，不仅有助于同学们对信息社会运行的理解，也有助于同学们从计算思维的视角提升工作效率。

贯穿本册的思路是，力争不仅要讲“是什么”和“如何做”，还要讲“为什么”。好比汽车，我们可以只是驾驶着它解决通行，也可以打开前盖看看它里面的部件及其之间的关系，了解它是怎么工作的。同学们会看到，一旦开始学习，就会接触一些术语，如字符串、数组、线性表、堆栈、队列、二叉树、图及哈希表等。本册将不满足于仅从概念上解释它们是什么，还要试图解释它们为什么会出现用计算思维解决问题的情境中。

学习数据与数据结构，编程实践是必须的，编程是手段而不是目的。本册既借助 Python 的一些高级功能来为实现应用程序提供方便，同时也兼顾数据类型和数据结构细节的剖析。

目前，大数据和人工智能的广泛应用，既拓展了数据与数据结构

知识的用武之地,也展现了数据与数据结构知识创新的空间。相信同学们通过本课程的学习,能够在相关认知和能力方面都有显著提高,为将来的不断进步奠定基础。

编 者

目 录

第一章 引言 ...1

项目主题 随时提供有序的报名信息 ...3

第一节 数据 ...4

第二节 数据结构 ...10

第二章 数据类型 ...15

项目主题 用数据描述身边的同学 ...17

第一节 基本类型 ...18

第二节 常用类型 ...24

第三节 组合类型 ...33

第四节 抽象类型 ...36

第三章 基础数据结构 ... 41

项目主题 送餐机器人的行走路线 ... 43

第一节 线性表 ... 44

第二节 栈 ... 61

第三节 队列 ... 69

第四章 常用数据结构 ... 77

项目主题 心算游戏好帮手 ... 79

第一节 二叉树 ... 80

第二节 图* ... 94

第三节 哈希表* ... 101

第五章 数据结构应用 ... 109

项目主题 人脸识别系统中的信息管理 ... 111

第一节 排序 ... 112

第二节 查找 ... 122

附录 参考程序 ... 131

后记 ... 145

第一章

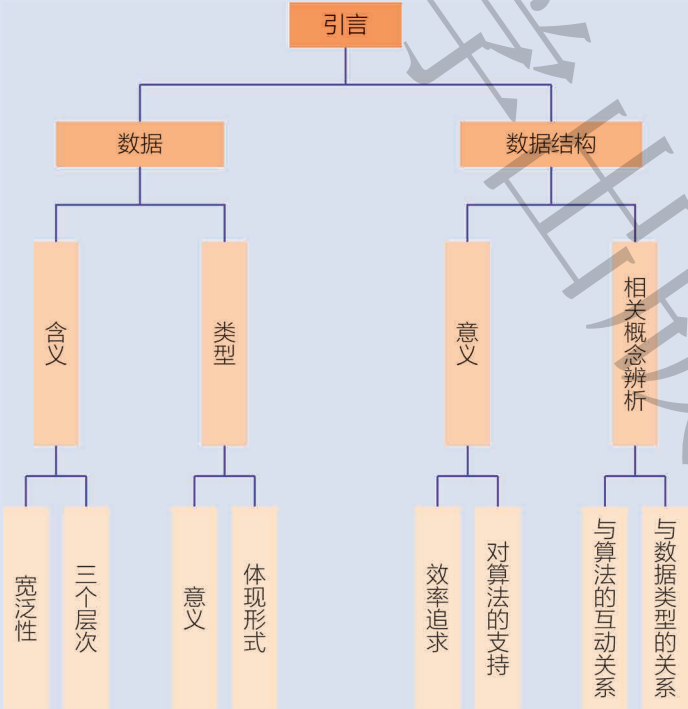
引言

本章学习目标

- 从计算机程序的角度理解数据的含义。
- 初步理解数据类型的概念及意义。
- 初步理解数据结构的概念及意义。
- 了解数据类型与数据结构的关系。

本章将从计算机科学的视角,开启关于数据类型和数据结构的讨论。数据类型是程序设计语言中的基本概念之一。数据类型的引入帮助我们在程序设计时更为方便、准确地描述和处理各种数据。数据结构是一个与算法紧密关联的概念,合理的数据结构搭配合适的算法能够有效地提高程序的运行效率。数据结构中的“数据”,以 0 和 1 的组合形式存储于计算机的存储单元中,其代表的具体含义则取决于该数据的数据类型。数据结构中的“结构”,是本课程学习的主要内容,在程序中它既服务于算法,也可能随着算法的运行而变化。数据结构并不神秘,从同学们开始学习程序设计的那一刻起,它就已经悄无声息地存在于代码的字里行间了。

本章知识结构



项目主题	随时提供有序的报名信息
------	-------------

项 · 目 · 情 · 境

设想同学们要参加一个活动,他们按照随机的顺序来到报名点,工作人员一一记录他们的信息,包括学号、姓名和性别等。不时会有负责人来了解报名的情况,除了想知道已有多少人报名外,还希望立刻看到一个按某种顺序(学号或姓氏笔画等)展现的已报名人的名册。如果你是该工作人员,会怎么做呢?

项 · 目 · 任 · 务

任务 1	任务 2	任务 3	任务 4
通过讨论项目情境中的要求,理解如果不用程序而是就在纸上做报名登记,可能会有哪些方式,分别会带来哪些困难(这里假设潜在报名的人会很多)。	假设每个报名者的信息包括(学号、姓名、性别)三项内容,通过考察 Python提供的丰富的语言机制,设计一个数据类型来表示单个学生的信息,并对你的设计选择进行评估。	设计一个数据结构来记录“当前已报名学生”。假设你预先不知道会有多少人来报名,那么你的数据结构应该是随实际报名学生数动态变化的。设计中会有哪些考量,你做取舍的理由何在?	实现一个程序,它接收随机而来的报名者信息和要看报名情况的指令,一旦后者出现,就按学号顺序直接打印出当前已报名者的名单。要求程序中不得调用排序函数。

第一节 数据

数据是信息技术中最常用的术语之一,其内涵和外延都十分丰富。作为对本课程后续内容的引导,本节首先带领同学们从多视角初步了解数据的基本概念,同时明确本课程中数据概念的主要视角,并通过几个例子初步探讨数据类型的意义。

一、进一步认识数据

本课程的必修1《数据与计算》中已经介绍过数据这一含义宽泛的概念。为了有效地讨论与数据相关的问题,人们对数据在不同语境下所表示的含义达成一定的共识。

1. 宽泛性

日常生活中,经常会接触到诸如“国民经济运行的主要数据”“人口统计数据”“中学生健康数据”等各种各样的“数据”。这里的“数据”可能指一个表,也可能指几张图,它们是相应背景下某种状态的直接反映。本课程必修模块中讨论过数据意义、价值等方面的问题,这些问题中提到的“数据”存在于日常生活中的综合应用层面,可能与计算机处理没有任何关系。

对于普通电脑用户而言,最常接触的是电脑桌面上或文件夹中的各种数据文件,例如一张图片、一个电子表格或一个视频文件等。它们通常由某些特定的应用程序生成,需要特定的应用程序才能打开和处理。因此,可以说它们是应用程序的输入或输出数据,其效用只能借助于计算机的处理才能体现出来。

对于计算机编程人员或者说计算机应用开发人员而言,除了要理解上述程序的输入输出数据,还要理解程序内部的数据,具体而言就是每一个常量或每一个变量代表什么,计算机能对它做些什么。而这些程序内部的数据在应用层面不一定有可以解释的含义。

2. 层次观

从信息技术的角度看,以上三种情况由表及里地体现了数据作为

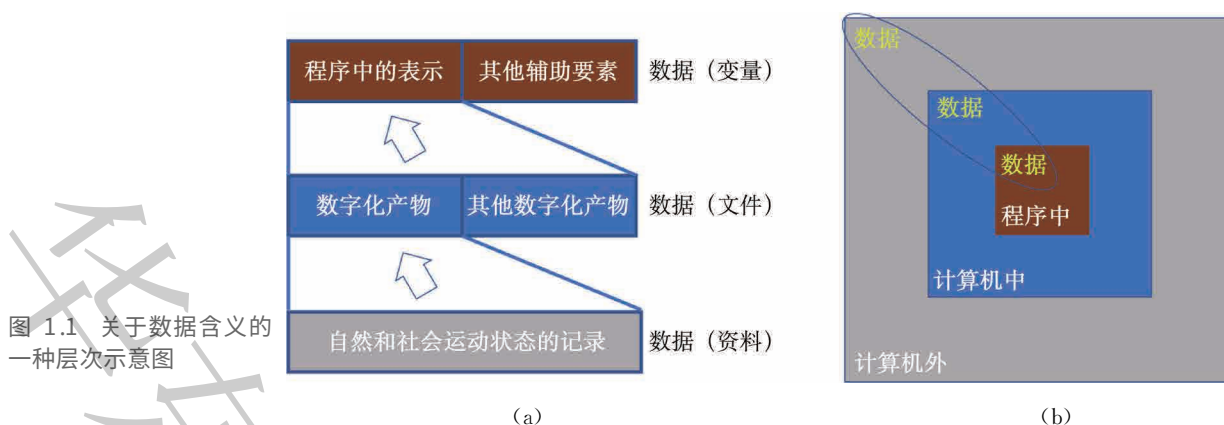


图 1.1 关于数据含义的一种层次示意图

一个概念的三个层面。如图 1.1(a)所示,将它看作上中下三层。普通泛指的数据对应下层的自然与社会状态的记录。为了用计算机处理它们,通常需要将它们数字化,成为计算机中的数据文件。但计算机中可以称为数据的文件不一定是那些记录,例如本书的电子版。因此就有了中间层的“其他数字化产物”部分。进一步地,当用计算机程序来处理那些文件时,除了其中的内容要变成程序内部的数据表示外,程序中一般都还需要一些额外的辅助数据才能实现目标。因此就有了上层的“其他辅助要素”部分。这样一种观念也可以由图 1.1(b)来简略表达,即可以从“程序中”“计算机中”和“计算机外”三个层面来谈论数据。这种概念上的分层有利于我们在讨论问题时,根据不同语境及不同的服务目的,理解“数据”这一术语的不同含义。例如,当谈论数据的价值时,涉及的主要是数据第一层面的含义。这在前面两个必修模块中已有充分讨论。当谈论两个功能相似但品牌不同的应用程序的兼容性时,则可能会涉及数据第二层面的含义。^①在本模块中,我们主要关心数据第三层面的含义,即计算机程序具体处理的数据。

这样划分数据概念虽然不够完备(即不一定每一个现实情境下的数据都能恰好归类于其中),但它能使我们讨论数据问题时保持适当的针对性和相关性。

二、数据类型

在计算机科学和程序设计中,数据类型是数据的一种属性。通过对该属性的声明,编程人员让编译器或解释器得知他使用相关数据的意图。

^① 例如,有的视频播放器能读入多种格式的视频文件并正常播放,有的则只能播放单一格式的视频文件。前者被认为兼容性强。又如,某些网站只能使用特定的浏览器访问,也属于这类问题。此时网页即浏览器程序的输入数据。

1. 计算机硬件不懂编码

计算机硬件最核心的部分是 CPU 和存储器。存储器由若干能够存放固定字长数据的单元构成,CPU 按照程序指令的要求将其中的数据(由 0 和 1 构成)读出,执行某种操作后,再将数据写回去。至于所处理的二进制数据代表什么,可不可以做某种操作,计算机硬件一般是不知道的。例如,假设我们知道有两个存储单元存放的数据是 ASCII 编码的字符“#”和“?”(即 00100011 和 00111111),但不小心让计算机对它们做加法了,即 $00100011 + 00111111$,CPU 经过处理给出 01100010。如果将结果看成 ASCII 编码即表示字符“b”。一般来看,这个操作没有任何意义。而“#”+“?”=“b”是让人莫名其妙的。这个例子说明计算机可能会做出无意义的操作。而在另一些情况下,这种“不理解”二进制代码含义的情况则有可能导致计算机做出错误的操作,造成损失。

在计算机技术发展的早期,程序员只能使用机器语言编程,他们必须非常熟悉各个二进制串的含义,十分小心地编写程序,避免让计算机做“不该做”的操作。受此限制,软件生产效率低下,难以编出大规模复杂的程序,阻碍了计算机在社会经济生活中的普及应用。

2. 程序语言应运而生

高级程序设计语言,例如 Fortran、C、Java、Python 等的诞生及其编译技术的发展,为解决这个瓶颈问题建立了一条基本途径。程序设计语言要求程序员指出数据的类型(数值型、逻辑型、字符型等),编译器根据不同数据类型的语义,检查是否有“不合法”的操作,若发现,则在程序调试阶段就报告给程序员,从而避免程序运行时出错。例如在 Python 中,如果运行如下程序:

```
a = '1'
b = '2'
c = a * b
```

就会产生一个类型错误(TypeError),告知 a 和 b 是两个字符,不能相乘。看起来简单,实际上要做好各种类型错误的检查是很不容易的,

这是现代编译器的一个重要功能。

探究活动

在 Python 中编一个会产生“Type Error”的小程序。

数据类型,作为程序设计语言中的一个重要概念,也是不断发展的,以给程序员提供更有效的方式向计算机表达问题求解算法,从而提高程序开发的效率。所谓“更有效”的含义就是表达层次更高。例如一个将 100 个数求和的程序,原则上可以用 100 个标量^①来表示那些数(需要 100 个变量名),在程序中要写 99 次加法,非常麻烦。如果使用数组类型,只需一个变量名,用下标来指示不同的数,可将 99 次加法蕴含在循环语句中,呈现为一行加法指令,使表达水平大大提高。

3. 数据类型让编程高效少错

高级程序设计语言中各种数据类型的有效运用,离不开编译技术的配合。在高级程序设计语言发展过程中,有时会发生程序语言中规定了某种功能,但某一版本的编译器“不支持”的情况,经过一段时间后,会出现新版编译器实现对该功能的支持。例如,当赋值语句两边类型不相同时,Python2 会做一些缺省的处理,而 Python3 则要求程序员在程序中明确表达类型转换,以避免因疏忽造成的错误。例如下面这段程序,在 Python2 和 Python3 中的执行结果是不同的:

```
a1 = 3; a2 = 3.0
b = 2
print (a1/b, a2/b)
```

Python2 的执行结果是: 1 1.5, Python3 的执行结果是: 1.5 1.5。也就是说,Python2 如果看到两个操作数都是整数,其计算结果就自动取整(此操作称为“缺省”操作)。这种操作看似方便,实则往往会带来潜在的错误。在 Python3 中,如果确实想将计算结果取

^① 在程序设计中,变量是一个上位概念,可以是标量(scalar),可以是数组(array),还可以是其他数据类型。标量是包含一个值的变量,而数组则可以包含多个值。

整,则需要使用 `print(int(a1/b))` 或者 `print(a1//b)` 语句来达到目的。

数据类型同时也是其他某些学科领域中的基础概念。在统计学科中,有类别数据和数值数据之分,例如性别就是一种类别数据,它可以取值“男”或“女”。计算机做统计数据处理时,需要在计算机和统计这两个学科不同数据类型体系之间建立起某种方便的对应,如用整数值 0 和 1 分别表示统计类别数据值“男”和“女”。

4. 文件扩展名的意义

与数据类型相关的一个概念是数据格式,它是数据文件的一种属性,常常通过文件扩展名来表达,告知操作系统应该启动哪一个应用程序对它进行处理。

广义地讲,数据类型和数据格式都是对本课程的必修模块中学过的“编码”概念的应用,即通过命名,对 0、1 串的语义作出规定。数据类型针对程序内部的数据,数据格式则主要针对程序的输入输出数据。

首先请通过如下实际操作案例,体会数据格式的含义及其意义。

打开电脑中的一个目录,可看到如图 1.2 所示的显示画面。

此目录中包含 9 个文件,不同的图标分别代表其中的一个文件。用鼠标双击其中一个图标,会启动相应的应用程序,并可进行某些操作,如画图、编辑文档、看视频等。

作为普通电脑用户,对这些常规操作似乎已经习惯成自然,通常不会去想这种过程为何能够发生。现在请尝试回答以下三个问题:(1)为什么点击 A.docx 的图标,文本编辑软件会被启动,点击另一个文件 D.mp4,视频播放器就会被启动呢?(2)如果修改文件名,将 A.docx 改成 X.docx,再次点击图标,结果会发生改变吗?如果将 A.docx 改成 A.mp4 呢?(3)C.png 和 G.jpg 的文件名和扩展名都不相同,分别点击它们,会成功启动同一个应用程序吗?为什么?

文件内容不变,只是扩展名变了,会导致计算机的不同行为。这样的问题与数据格式直接相关。计算机需要用户正确地告诉它数据文件的类型,方能正常有效地工作。图 1.2 中的 9 个文件,涉及 8 种

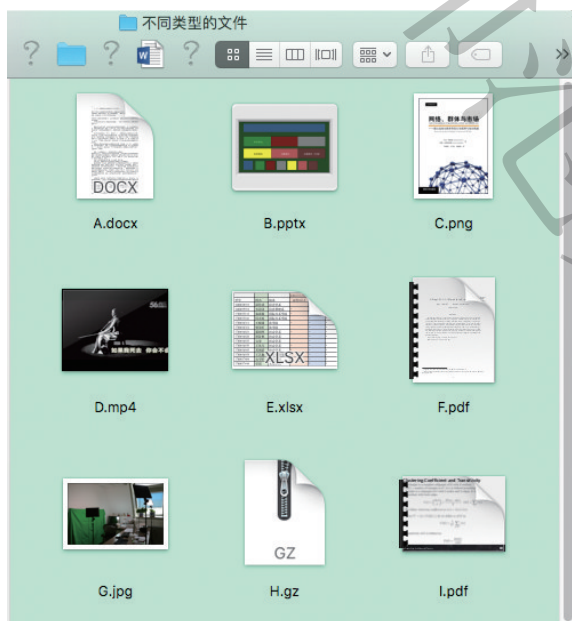


图 1.2 不同类型的文件

数据格式,不同数据格式以不同文件扩展名的形式标识。使用计算机打开文件,有时会出现所谓“乱码”的情况。其实,不一定真是文件的“码”乱了,而是打开它的程序不对。动手试一试,我们能从操作中得到实际的体验。

体 验 思 考

课堂上(或课后)实际动手体验本章第一节第二部分中的例子,回答所提出的问题,包括讨论为什么修改文件名很顺利,而修改文件扩展名可能会被警告。

第二节 数据结构

不同于数据类型是多个学科都可能有的概念(含义有别),数据结构是计算机学科的特有概念。数据结构是关于数据之间关系的描述,及其形成与访问的方式。关于数据结构的知识是计算机学科的基础知识,运用数据结构的能力是开发高水平程序的要求。本节将从总体上讨论数据结构的意义及其与数据类型的关系,一些典型的数据结构与应用则是本书要讲述的主要内容,将在第三、四、五章详细介绍。

一、数据结构的意义

数据结构这一概念,是随着程序设计技术的发展,为弥合计算机硬件的简单性和程序应用的复杂性之间的鸿沟^①,逐步形成并发展起来的。请通过以下应用实例,初步体会上文所述。

数据结构的选择影响程序执行的效率

有一程序,不断接收输入数据(即它面对的是一个“数据流”),针对每一个新到数据(严格讲应称为数据元素),均需判断是否曾经接收过。若是,则抛弃;若否,则保留。^②如何提高此判断的效率?下面以输入数据流 4, 3, 5, 1, 4, 2, 3, 1 为例,分析两种不同处理方法的效率。

方法一,采用简单的线性存储结构^③,即存储空间中的存储单元是顺序编号的 0, 1, 2, 3, ..., $n-1$,处理过程可描述如下:

每收到一个数,就将其依次与空间中已有的数进行比较,发现有相等的,就停止;若一直到最后还没有相等的,就把这个新数放在后面。

所给输入数据流处理后在存储空间中留下的结果为 4, 3, 5, 1, 2。如果用比较次数作为衡量效率的指标,针对原输入数据流中的 8 个数据,每一个输入数据需用的比较次数见表 1.1,总比较次数为 17。

① 这里说的“简单”和“复杂”,是就语义而言的。无论是做科学计算、进行文字处理还是上网冲浪,这些复杂的程序应用,在计算机硬件中化为简单的活动,即在二值(0、1)逻辑上的高速变换。

② 这个例子听起来简单,但它是许多实际应用的抽象,如互联网路由器就需要有类似的功能。

③ 实际情况中采用此结构的有计算机硬件的存储器、程序中用的一维数组等。

表 1.1 简单线性结构下的比较次数

数据	4	3	5	1	4	2	3	1
比较次数	0	1	2	3	1	4	2	4

方法二,想象有一种非线性的存储结构,每当收到一个新的数,就在该结构的引导下进行判断,必要的话,也对该结构进行动态更新,以支持对后续数据的判断。图 1.3 以所给输入数据流展示该结构的变化过程,图中所含 8 个线框中的内容,顺序对应 8 个数依次到来经判断处理后的结果,虚线框内表示此次接收到的是已有数据,仍需做判断,但对整个留存结果无影响。

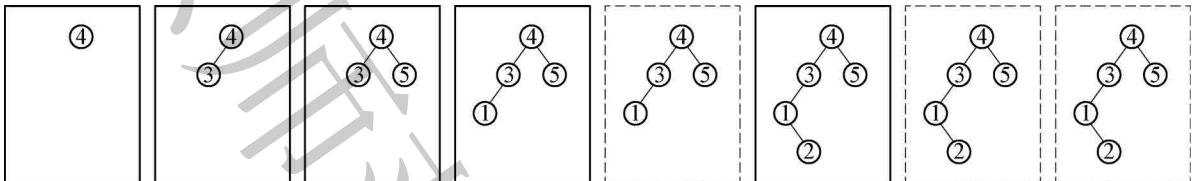


图 1.3 非线性存储结构的变化过程

此结构称为“二叉树”,它由一些“节点”和“边”构成,顶端的节点称为根节点,根节点左下方整体称为“左子树”,右下方整体称为“右子树”。^①基于此结构,具体操作如下:

当收到一个新的数(x),首先将此数和树根相比较,如果相等,就停止;如果较小,就与左子树比较,若左边没有子树,就将 x 放在左下,成为左子树的根;如果较大,就与右子树比较,若右边没有子树,就将 x 放在右下,成为右子树的根。其中“与左子树比较”和“与右子树比较”,是一个递归的过程^②,即把左、右子树看成二叉树,照样进行。

表 1.2 呈现了每一个输入数据所用的比较次数^③,总比较次数为 13。

表 1.2 二叉树结构下的比较次数

数据	4	3	5	1	4	2	3	1
比较次数	0	1	1	2	1	3	2	3

① 每个节点可能左右两个子树都有,也可能只有一个,甚至没有。没有子树的节点称为叶节点。
② 后面要学习的一个概念,此处可以比拟俄罗斯套娃,各层形态相同,层层嵌套,逐层缩小。
③ 鉴于此处示例数据不宜过多,仅假设了用 1 次比较就能给出 $<$ 、 $=$ 、 $>$ 的判断(在计算机中实际需要 2 次),从而得到总次数 $13 < 17$ 。在本书的第五章第二节将会看到,当数据量足够大,即便上述判断算 2 次比较,二叉树优于线性查找的基本结论仍然成立。

通过采用不同的数据组织方式来提高计算的效率是我们研究的重点。计算机科学家发明了诸如二叉树这样的结构,并通过软件的方式体现在硬件的线性存储空间上。几十年来,人们根据不同的计算需求发明了多种数据组织的形态,以支持高效软件的开发,它们统称为数据结构,这正是本模块学习的重点。二叉树是其中颇具代表性的,在本书多处都会涉及。

上述例子让我们获得了对数据结构的直观印象,即数据结构表达了程序数据之间的某种关系。特别需要注意的是,图 1.3 中数据之间的关系,是由算法的要求派生出来的,反过来又支持算法的执行。一般而言,数据结构中的数据,不仅包括程序的输入数据,往往还包括程序运行过程中产生的中间结果数据或控制数据。由数据结构表达出来的关系,通常是根据算法的需要形成的关系,而不仅是那些数据之间的某些天然关系。

作业练习

参照本章第二节第一部分中的例子,对 16 个数的序列: 5, 3, 5, 4, 2, 1, 6, 8, 3, 3, 4, 1, 7, 8, 4, 2, 给出类似于表 1.1 和表 1.2 的结果。

二、数据结构与算法

计算是操作步骤的执行,算法是操作步骤的描述,即先做什么、后做什么,当某个条件出现时该做什么等;操作是作用在数据上的,数据结构是数据的组织形态,体现数据元素之间的关系,以支持算法的高效执行。

一些情况下,应用数据之间有一种天然的关系,计算机中恰好也有一种对应的数据结构^①,算法设计者依据该结构来编排操作序列,按照计算任务的需要,对该结构上的数据进行比较、交换、更新等操作。整个过程,数据结构不发生变化。

另一些情况下,数据之间的结构关系是在算法过程中形成的。图 1.3 所示就是一个例子。在此例子中,算法设计者事先确定要采用

^① 例如每小时测一次气温,一天得到 24 个数字,依时间顺序组成一列。在计算机中就自然地对应线性存储结构。

二叉树结构,而所设计的操作序列一边在数据基础上构建一棵具体的二叉树,一边依托已构成的部分二叉树结构完成计算任务所要求的数据处理操作。这里不妨再仔细体会一下该示例所展示的意象,按照这种思路编写的程序,是不是有种“边执行边为自己创造条件”的意味?

数据结构与算法相辅相成

上述两种情况,不妨称前者为静态,后者为动态,都是计算机应用中的常见情况,但后者更能体现数据结构应用的精髓和算法与数据结构的精巧互动。当然,也有许多场合,在同一个算法中既用到静态数据结构,也用到动态数据结构。

瑞士计算机科学家,Pascal 等编程语言发明人,1984 年图灵奖得主,尼古拉斯·沃斯(Niklaus Wirth)教授在 1975 年曾给出“算法 + 数据结构 = 程序”的著名关系式,是对数据结构与算法关系的一个精辟诠释。这样一种见解,在动态数据结构的运用中显得格外精彩。

算法 + 数据结构 = 程序

三、数据结构与数据类型

本章第一节中,我们聚焦的是数据类型的含义和意义。数据类型是程序语言中的概念,使用它有助于提高程序开发的效率。前文中讨论的数据结构是程序设计中的概念,与算法一起,旨在提高程序运行效率。^①尽管有如此区别,在计算机科学的实践中它们也有密切的联系,常常体现在同一个数据对象的不同侧面或层面。

1. 数据结构中的元素具有数据类型

对这种情形可以先做个类比。我们说某人是孔子的第七十代孙子,那么“孔氏家族成员”就是他的一个标签,也可以说他是“孔氏家族成员”概念下的一个实例。而按照世系,所有孔氏家族成员构成了“孔子家谱”。进一步地讲,也有“孟子家谱”“朱子家谱”,等等,它们又都是“家谱”这个概念的实例。于是,单就孔子家谱而言,既可以说它是

^① 严格讲,合适数据类型的采用也可能提高程序运行效率,良好数据结构的采用也可能提高程序开发的效率,但它们的作用重点有所区别。

一个包含了许多成员的家谱,也可以说它是家谱的一个例子。

这样,一方面虽然各种数据结构其概念本身是抽象的(或者说是逻辑性的),但在程序实现时总是具象的,由若干数据元素和它们之间的关系构成。而数据元素总会有一定的数据类型,同一种数据结构在不同的应用中其数据元素可能是不同的数据类型,例如本节第一部分中的那个例子,数据可以是整数,也可以是实数,但数据结构不变。

2. 数据类型可以通过数据结构实现

另一方面,如我们要在第二章重点学习的,现代高级程序设计语言除了提供我们在《数据与计算》模块中已经熟悉的数值型、字符型等基本数据类型外,还支持程序员定义比较复杂的数据类型。例如表达个人信息的数据元素,可以定义为由一个字符串(姓名),一个数字(年龄)和一个字符(性别)构成的数据类型(可命名为“Person”)。这样的数据元素也可以看成是一个数据结构,那么我们就可以说此处基于数据结构定义了一个数据类型。

更复杂一些,还可以基于诸如线性表、二叉树等数据结构定义数据类型。在这样的数据类型上的操作与在相应数据结构上支持的操作相对应。以基于线性表的数据结构为例,可以做插入、删除、读取第*i*个元素等操作。

鉴于上述数据类型和数据结构概念在程序设计应用中相互交织的情况,对一个具体的变量来说,在一些适用场合中,可能存在双重身份,例如既可以说它是具有线性表类型的数据,也可以说它是具有线性表特性的数据结构。有些概念,在运用上也具有两重性,例如字符串和数组,在编程中主要被看作数据类型,但因为它们也体现了其中数据元素间的关系,有时人们也称它们为数据结构。这样的现象对初学者可能会造成一些困扰,只要深入学习并实践,就能感受到其正是计算机科学的魅力之一。

项目实践

按照本章项目要求,分组讨论实现方案,针对所给出的四项任务,完成项目报告。项目完成的测试将分成两个部分。第一,运行程序,看对符合要求的输入是否能给出所要求的输出;第二,审读程序,看在接收到“请打印当前已报名学生名单”指示后是否能立刻直接打印当前报名表的内容。



第二章

数据类型

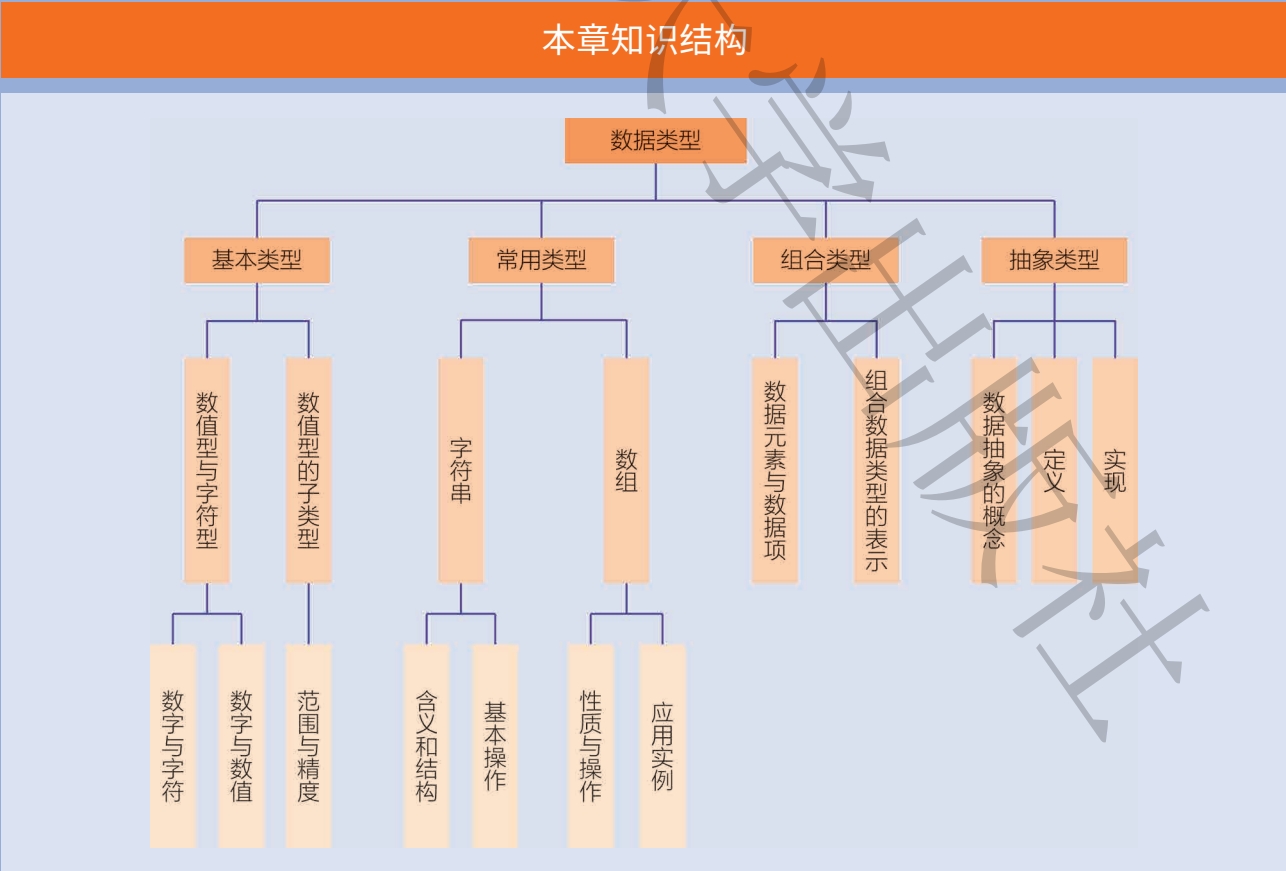
本章学习目标

- 了解计算机程序中涉及到的基本数据类型和常用数据类型。
 - 能够选择不同数据类型来描述客观事物,并运用于程序设计。
 - 初步理解抽象数据类型的含义,会定义简单的抽象数据类型,并将其应用于计算任务中。
-

现实世界中,一方面万物丰富多彩,另一方面同一事物常常呈现多侧面和多层次的特征。数据作为记录、描述和表示事物的一种基本工具,相应地也存在多种类型。

在应用层面上,有文本数据、图像数据、音视频数据、数值型数据等多种数据类型。服务于计算的,有数值型、逻辑型、字符型等基础数据类型。而为了有效地完成计算任务,在基础数据类型的基础上人们发展出了字符串和数组等常用数据类型。特别是为了能够有效应对各种应用的复杂需求,计算机程序设计语言通常会提供机制,让编程人员可以自行定义组合数据类型。抽象数据类型是自定义数据类型的高级形式,在编程实践中充分体现为面向对象程序设计。

本章主要学习服务于计算的数据类型的相关知识。理解数据类型,包括它们的含义、适用的操作、对算法的影响、对存储的需求和对应用的意义,以及相互的模拟与转换等,是理解计算的一个重要基础。



项·目·情·境

进入高中阶段后,经过一段时间学习和相处,同学们之间都有了一定了解。为了进一步提升班级的凝聚力,高一(5)班召开了一次主题班会——“相处了半学期,谈一谈身边的同学”。在相互了解和介绍的过程中,有的同学想到了运用数据的形式进行描述,让同学们的特点更直观具体。同学们组成研究小组,制订研究方案,从比较数字、数值和数据的概念和实例入手,首先描述身边同学的属性及其属性的值,然后,比较具有相同属性的同学,计算相似度,最后通过编程实现数据的描述,并评估结果。

通过该项目,认知数字、数值和数据的关系,探究数据的要素和数据的组织结构,进而体验数据抽象的一般方法。请你以相同的方式描述一下你身边的同学。

项·目·任·务

任务 1

通过比较数字、数值和数据的概念,找出反映身边同学典型特征的属性及其值的范围,归纳出数字化简略描述。体验数据抽象的必要性。

任务 2

提出计算两个同学之间相似度的依据,分析同学属性中数据元素和数据项的关系,画出示意图。思考存储的效率问题,感受数据的逻辑关系对现实描述的重要性。

任务 3

认识、比较和选择合适的数据组织方式,设计抽象数据类型用以概括出同学属性集合,感受数据抽象的过程。

任务 4

完成一份《“谈一谈身边同学”数据描述》的设计方案,编写程序实现相似度计算,举例说明数据组织方式起到的作用。制作一份“数据描述”演示文稿。

第一节 基本类型

计算机程序设计语言大都预设了一些用于表达数据类型的关键字,例如 byte、integer/int、real/float、char、boolean 等,这让程序员能直接指明变量所属的基本数据类型。^① 这些数据类型,通常与计算机的硬件资源,如存储单元(字节、字)或 CPU 处理能力(定点、浮点),有直接的对应关系。因此,程序对基本类型变量的操作效率最高。

一、数值型与字符型

数目(number)与文字符号(character, word, symbol)不仅是记载人类文明,还是促进人类文明交流互鉴的两种最基本方式。数目让我们能够描述距离的远近、时间的长短、规模的大小;文字符号让我们能够描述形态、状态、情感、思想和信念。

数目通过数字(digit,也称“数目字”)来表示,例如一、二、三、四……十、百、千……;1, 2, 3, …, 9 和 0; I, II, III, IV, …, X, C, M, …。它们发端于不同的文明,追求同样的目的(即表示数目)。经过历史长河的洗礼,阿拉伯数字成为主流,其根本原因是基于它形成的进位制能够表达任意大的数目。

进位制思想的一个典型运用——二进制(binary system),则成为当下信息社会的一个基础。

二进制,只需要两个数字——0 和 1,就能表示任意大的数目,以至于我们现在讲的数字主要指的就是它们。^② 而且,0 和 1 不仅能表示数目,还能用于对文字符号进行编码,常见编码系统如 ASCII 码、汉字国标码 GB1830-2005、UTF-8 等,从而构成了信息处理的数据基础。这些内容我们在本课程必修模块的学习中已多有认识。

数目与文字的一个重要区别,是前者具有“数值”。每个数对应数轴上的一个点,从而具有自然的大小关系,可以对它们有意义地进行算术运算。后者则没有这种特性,它们更强调灵活组合(例如构成任意字符串),以表达现实世界中无比丰富的数值无法刻画的意义。

计算机作为能够处理现实社会各种信息的通用工具,必然要能够

^① 著名的程序设计语言 Fortran、C 和 Java 都要求程序员声明变量类型。不过本课程主要用到的 Python 没有这样的要求,这虽然为程序员带来了方便,但损失了程序执行的效率。计算机硬件性能的逐步提高,以及多数程序对计算的效率并无很高要求,是 Python 这样的语言近年来得到极大普及的重要原因。

^② 例如作为信息社会的基础,万物数字化(digitization)就是指将事物在最基础的层面都用 0 和 1 编码,而不是用 0~9 编码。

有效地表示数目与文字这两个基本概念。在程序设计语言中,与数目和文字相对应的就是数值型和字符型两种基本数据类型。在数据类型的区别下,程序中语句 $x = 1$ 和 $x = '1'$ 的含义是不一样的,前者是将变量 x 赋值为数值 1,后者是将变量 x 赋值为字符 1。按照二进制编码,数值 1 的二进制码是 00000001,字符 1 则是 00110001。另一方面,同一个二进制码在不同类型中有可能表示不同的含义。例如二进制码串“01000001”既可以表示十进制整数 65,也可以表示 ASCII 码表中对应的字符 A。

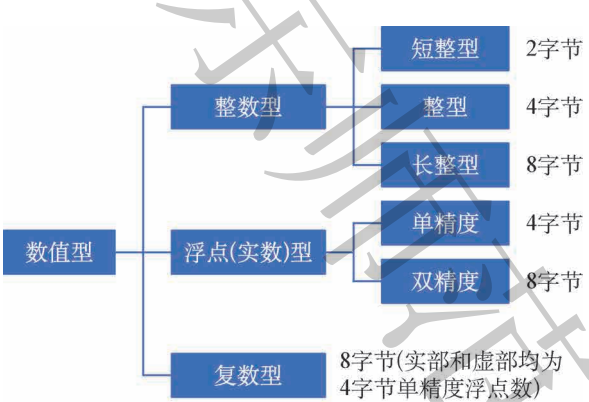


图 2.1 常见子类型关系示意图

在计算机程序语言中,为了应对各种数值计算的需求,按数据表示范围和精度要求的不同(对应存储需求与编码方式的不同),数值型数据通常又分为多种不同的子类型,图 2.1 是常见子类型关系示意图^①,其中也标注了相应的存储需求。

关于字符型数据,我们需要了解的基础概念首先是字符集,然后是编码方案。字符集规定了其所要表示的字符。一般来说,字符集的大小决定了编码所需要的位数。

一般地,多数程序设计语言默认字符型数据按照 ASCII 编码方案定义表示。在本课程的必修模块中我们已经学过,ASCII 码的字符一共有 128 个,7 位编码,每个编码占用 1 个字节内存。

随着计算机技术的发展和应用的普及,各民族语言文字出现在计算机应用中的需求越来越强烈,ASCII 的编码容量不能解决这一问题。于是各国逐步规定了自己的字符集和编码方案,在中国就有了 GB1830 - 2005 等汉字编码字符集。

由于不同的字符集可能包含一些相同的字符(例如都含有阿拉伯数字),同一个字符在不同的编码方案下可能有不同的编码,这就给计算机程序的处理造成了困扰,不时有某某程序不支持某某编码的情况发生。例如,Python2 不支持汉字,Python3 支持。这是什么意思呢?简单说,就是在 Python3 中,可以有语句 $x = '发展进步'$,但在 Python2 中应用此语句则会报错。

为了克服编码不一致的矛盾,计算机产业界定义了一个大而全的 Unicode(万国码)字符集,包含世界上大多数语言文字,其中一种编码方案是 UTF - 8,它是目前采用最广泛的,也是 Python3 缺省采用的编码方案。

① 不同程序设计语言的具体情况会有些差别。

第二章第一节第一部分结束时,提到 Python2和 Python3在处理汉字上的差别。尝试在两种环境下分别执行语句 `print('发展进步')`,观察出现的情况。进而,在 Python2中执行:

```
# coding = utf-8  
print('发展进步')
```

观察并回答: 执行结果与前面的有什么不同? 为什么?

二、数值型数据的实现

数值型数据的范围和精度,是关系到其存储和计算的两个基本问题。图 2.1 中指出了不同子类型所需的字节数,本质上,也就给出了相应子类型所能表示的数值的个数。例如,短整型最多能表示 2^{16} 个数,整型则能表示 2^{32} 个数,单精度浮点型也最多能表示 2^{32} 个数,等等。

从数轴上看,所谓“范围”,就是所能表示的最小数和最大数所定义的区间。所谓“精度”,可以理解为在一定范围所能做的最小区分。以 4 位十进制正数为例,若表示的是整数,则范围是 $[0, 9999]$,精度为 1;若表示的是 2 位小数,则范围是 $[0.00, 99.99]$,精度为 1×10^{-2} ,要高得多。注意,这两种情况所占的存储位数是一样的。也就是说,在同样位数即存储空间资源限制下,可以通过假定小数点的位置,调节范围和精度上的某种平衡。

在计算机中,整数型数据的范围依照其位数,以 0 为中心对称分布,包含其中的每一个整数。而由于 0 也是一个数,设计上就令正数比负数少一个,于是短整型的表示范围是 $-32768 \sim 32767$,整型则是 $-2^{31} \sim (2^{31} - 1)$ 。计算机如何区分正负呢? 通常用最高的那个二进制位,0 表示正,1 表示负。

例 2.1 整数型数据范围: 计算机硬件在做定长整数的操作时,并不会检查是否达到相应范围的边界从而不该再有增量,而是将所有的数看成是一个“线性循环”,0 后面是 1,1 后面是 2……,最大的数(正的)后面是最小的数(负的),等等。例如在和计算机硬件最接近的 C 语言中,若 `x` 是一个短整型变量,执行如下代码:

```
x = 32767; //所能表示范围中最大的数
x = x + 1;
printf("%d\n", x);
```

看到的输出会是 - 32768,即所能表示范围中最小的数。这一点在实际应用中应多加注意。有些语言,例如 VB,则会对类似程序段的执行报错,那是因为语言本身提供了越界检查功能。

日常生活中说的“小数”,在计算机中称为浮点类型数据。为什么用“浮点”这个有点奇怪的名称呢?下面我们会看到一种巧妙的格式设计,它使得小数点的位置可以“浮动”(不同于上面讲的十进制例子,明确规定小数点后是 2 位),从而使范围和精度达到更好的平衡。这样一种设计,本质上就是对可用的二进制位做一个合适的使用安排和解释。下面我们以 8 个二进制位为例,解释浮点数表示的基本原理。为简单起见,其中提到的一些“规则”与计算机中实际采用的不尽相同,但不妨碍我们对核心原理的理解。



图 2.2 在一种浮点数格式下 $\frac{11}{4}$ 的存储

例 2.2 数据存储时可用的二进制位被分为三个部分:符号(S),指数(E)和尾数(M)。图 2.2 展示的例子中,S 占最左边第 1 位;E 占第 2~4 位;M 则占第 5~8 位。

按照现在给的二进制位的值,这个图中所表达的是什么数(x)呢?

$S=0$,表示 x 为正数。 M 部分字面上是 1011,我们规定将它看成最左边隐含一个小数点的小数,即 $M = (0.1011)_2$ 。 E 部分则指出为得到所表达的数 x ,需要在 M 的基础上移动小数点的位数。不妨假设 E 中的第 1 位也表示其正(0)负(1),正则右移,负则左移。现在, $E = +2$,于是 $x = (10.11)_2$,换算成十进制为 $2 + 0 + \frac{1}{2} + \frac{1}{4} = \frac{11}{4}$,即 2.75。

探 究 活 动

假设某数与图 2.2 中数据具有相同的浮点数格式,但其空间中存储的内容为 11100101。试给出它所表示的十进制数。注意,此例中 E 对应的是 110(即十进制的 - 2),小数点应左移 2 位,也就相当于尾数 M 右移 2 位,即在它的前面补上 2 个 0。这就是“浮点”的含义。

下面我们来讨论范围。在如图 2.2 所示格式下,能表示的最小数

和最大数是多少呢? 从数轴的角度思考会比较容易: 使表示的数“尽量负”和“尽量正”, 即可得到 10111111 和 00111111 , 也就是 $(-111.1)_2$ 和 $(111.1)_2$, 转换成十进制则为 $-\frac{15}{2}$ 和 $\frac{15}{2}$ 。

浮点数的精度对应所能表达的有效位的个数, 由尾数的最低位确定, 此处就是 $2^{-4} = \frac{1}{16}$ 。

探究活动

假设用 16 个二进制位表示浮点数, E 用 6 位, M 用 8 位, 其他规定与图 2.2 中的格式相同。试求它能表示的绝对值最大和最小的数。

上面, 我们用一个简化的例子讨论了浮点数格式的要点。一般来说, E 的位数越多, 表示的范围越大, M 的位数越多, 精度越高。在实际应用中, 浮点数采用 32 位或 64 位表示, 并按照一定的规则划分 E 和 M 所用的位数, 分别称为单精浮点数和双精浮点数, 后者的范围更大、精度更高。

例 2.3 浮点数的精度问题对实际编程的影响: 由于二进制位是离散的, 一个 N 位浮点数格式最多只能表示 2^N 个数, 我们应该得到的一个重要认识是, 虽然浮点数可以表示小数, 但并不是在浮点数表示范围内的所有小数都可以得到表示。这不仅因为在一个区间内的实数(甚至有理数)是无穷多的, 还因为浮点数在数轴上的分布并不均匀。这是浮点数与整型数的一个基本区别。由此造成许多十进制中很简单的小数(如 0.1 和 0.2)在计算机中只能有近似值。

如在 Python 交互式环境下尝试如下语句:

```
>>> x = 0.1 + 0.2
>>> x == 0.3
False
>>> x
0.30000000000000004
```

为规避这类问题, 在必须判断两个浮点数(变量)是否相等时, 在 Python 中可以使用语句 `round(x[, d])` 对 x 四舍五入, d 是小数保留的位数, 当然 d 的选择应适当。此处四舍五入的语句如下:

```
>>> round(0.1 + 0.2, 1)
0.3
```

例 2.4 复数类型：有些语言(包括 Python)也支持复数类型,在 Python 中复数表示为 $a + bj$ 形式,例如 $3 + 2j$, $6 - 4j$ 等。而对于一个复数类型变量 z ,可通过 $z.real$ 获得实部,通过 $z.imag$ 获得虚部,它们均为单精浮点数。如在 Python 运行环境下:

```
>>> z = 4 + 3j
>>> z.imag
3.0
>>> z.real
4.0
>>> z = z + (2 - 1j)
>>> z
(6 + 2j)
```

作业练习

试利用 Python 的复数类型写一个求解一元二次方程的程序,即对任何实数 a 、 b 和 c 都能按照求根公式给出方程 $ax^2 + bx + c = 0$ 的两个根。(提示: Python 有支持复数运算的函数库 `cmath`, 其中包含求平方根的函数 `sqrt`)

第二节 常用类型

在基本数据类型之上,程序设计语言一般会提供关键字,让程序员能表达比较高层的常用数据类型,典型的有字符串(string)和数组(array)。顾名思义,前者可以看成由若干字符“串接”得到的整体,后者则是由若干同类型数据元素构成的多维数据体。它们不仅在逻辑上意味着是一个连续的数据体,从而在程序中可以直接引用其中的单个元素或“片段”,而且在物理上也通常存放于内存的一段连续空间中,因此有较高的访问效率。这是字符串和数组的共同特点。

一、字符串

1. 什么是字符串

字符串是由 0 个或多个字符组成的有限序列。一般使用顺序存储结构^①,末尾以某个特殊符号(如'\0')表示结束,但此符号不作为字符串的内容,也不计入字符串的长度,图 2.3 是字符串“Data Type”顺序存储的例子。

	长度为 9									结束标记	
串值	D	a	t	a		T	y	p	e	\0	空闲空间
空间下标	0	1	2	3	4	5	6	7	8	9	

图 2.3 字符串存储示意图

字符串通常用 $s = "a_0 a_1 a_2 \dots a_{n-1}"$ 表示,其中 s 是串的名称, n 是串的长度,每个 $a_i (0 \leq i < n)$ 代表一个字符, i 就是该字符的下标,表示该字符在串中的位置,串中所含字符的个数就是串长。含有零个字符的串称为空串,用 $s = ""$ 来表示,空串不等于空白字符构成的串。通常把一个串中任意个连续字符组成的子序列(含空串)称为该串的子串,

^① 原则上讲,字符串也可采用链式存储(如同第三章将学习的线性表),但除了在连接串与串以及子串替换操作时较方便,总的来说不如顺序存储有效,在程序语言中很少用到。

相应地,包含子串的串称为主串。例如,"D"、"Da"、" "、"pe"等都是"Data Type"的子串。

2. 字符串的实现

字符串是 Python 中最常用的数据类型,可以使用引号('或")来创建。

例 2.5 字符串的创建与子串操作

```
>>> str1 = "Data Type"
>>> str2 = "python"
>>> str1
'Data Type'
>>> str2
'python'
```

可直接使用下标的方式访问字符串中的字符或子串,但下标越界会导致报错。如下所示:

```
>>> str1 = "Data Type"
>>> str1[0]
'D'
>>> str1[0: 4]
'Data'
>>> str1[5: ]
'Type'
>>> str1[11]
IndexError: string index out of range
```

体 验 思 考

- 1 设已有字符串 `s="I am a Python learner."`,请用两种办法取出其中的 "am" 子串。
- 2 数字重复出现次数的统计: 随机生成 500个三位正整数,升序输出所有的不同的数字及每个数字重复的次数。

3. 字符串的比较

有时需要对字符串进行排序。两个字符串之间的先后顺序(或者说大小)是如何确定的呢? 不同的程序设计语言规定不尽相同。在 Python 中, 串的比较是通过组成串的字符的 ASCII 编码大小的比较来进行的(即把字符的 ASCII 编码看成是十六进制正整数)。例如字符'a'的编码是 61H, 'A'是 41H, 那么就规定'a'>'A'。对于两个字符串, 先比较第一个字符, 如果第一个相同的话, 再比较第二个, 依此类推, 一旦发现两串对应位置上有一个字符比另一个大, 就规定大字符所在的字符串大; 如果一个字符串已经没有字符可比了, 而另一个还有, 则后者大。当且仅当两个串的长度相等并且各个对应位置上的字符均相同时, 则称两个串相等。如下所示:

```
>>> str1 = "admin"
>>> str2 = "administrator"
>>> str3 = "advise"
>>> str1>str2
False
>>> str2>str3
False
>>> str3>str2>str1
True
```

4. 字符串的基本操作

Python 中针对字符串对象提供的一些常用操作方法如表 2.1 中所示。

表 2.1 Python中字符串常用操作方法

针对长度为 n的字符串 s	描述
<code>s[i:j:k]</code>	切片: 提取对应的部分作为一个子串, 下边界 j并不包含在内, k为递增步长
<code>s.count(s1)</code>	返回子串 s1在字符串中出现的次数

(续 表)

针对长度为 n 的字符串 s	描述
s.index(s1)	如果 s 中包含子串 s1, 返回子串 s1 第一个字符所在位置, 如果没有, ValueError
s.find(s1)	如果 s 中包含子串 s1, 返回子串 s1 第一次出现时第一个字符所在的位置, 如果没有, 返回 - 1
s.replace(s1, s2)	在 s 中用另一个字符串 s2 替换指定子串 s1
s.split(x)	通过指定的分隔符 x 将字符串 s 拆分为一组子串
s.lower(), s.upper()	将字符串 s 中字符转换为小写或大写

利用这些方法, 我们可以轻松满足一些看起来复杂的需求。

例 2.6 给出特定字符在一个字符串中的出现次数, 程序如下。

```
str = "Algorithm + Data Structures = Programs";
sub = "a";
print "str.count(sub): ", str.count(sub)
```

以上实例输出结果如下:

```
str.count(sub): 3
```

例 2.7 从 Python 字符串中提取数字字符。

```
def is_num(uchar): # 判断一个 unicode 编码的字符是否是数字字符
    if uchar >= u'\u0030' and uchar <= u'\u0039': # 数字字符的范围
        return True
    else:
        return False
def new_str(content):
    content_str = ''
    for i in content: # 逐个检查字符串中的字符
        if is_num(i):
            content_str = content_str + i + ","
    return content_str
basic = "沪 FD-7759"
basic_str = new_str(basic)
print (basic_str)
运行结果: 7, 7, 5, 9,
```

恺撒密码又叫移位密码,即将原码中的字母按照指定的偏移量(基于字母表)做移位,形成密码。例如,当偏移量是 3 的时候,所有的字母 A 将被替换成 D, B 将变成 E,依此类推, X 将变成 A, Y 将变成 B, Z 将变成 C。由此可见,偏移量(也称位数)就是恺撒密码加密和解密的密钥。恺撒密码作为一种最为古老的对称加密方法,在古罗马的时候就已经很流行了。请编程实现恺撒加密和解密,如输入字符串 "he lb"和偏移量 4,输出加密后的字符串 "lpps"。

二、数组

1. 什么是数组

现实生活和工作中,人们经常要对一批同类型数据进行某种计算。例如求一个篮球队 20 个队员身高的平均值,在计算机中若用 20 个标量 $a, b, c, \dots$ 来分别表示他们的身高,理论上可行,但实际上编程会很麻烦。于是编程语言提供了一种称为数组(array)的数据类型,用一个符号名来统一表示这一组数据,而下标(index)的方式区别其中的数据元素,如 $h[0], h[1], \dots, h[19]$ 。^①结合循环语句的使用,求平均值问题的程序就可以写得十分简洁:

```
sum = 0
for i in range(20):
    sum = sum + h[i]
average = sum/20
```

有时候,人们关心的数据呈现为阵列的形式,例如一幅 1024×768 大小的黑白图像,其每个像素的灰度值自然地排成一个二维阵列,做图像处理就要涉及一个像素与周围像素之间的关系。如果是一幅彩色图像,每个像素则由红、绿、蓝(RGB)3 种颜色表示,每种颜色分别对应一个二进制数,于是涉及的数值有 $1024 \times 768 \times 3$ 个,我们可以在脑海中将其想象为一个三维的数据体。

^① 有些编程语言,例如 Fortran,数组的下标从 1 开始。

这些数据体在程序设计中对应二维数组和三维数组中。数组可用一个变量名表示(比如 pic_bw, pic_color),数组中的不同元素则用二维和三维下标区别,如 pic_bw[0, 0], pic_bw[456, 1023]; pic_color[0, 0, 0], pic_color[1, 321, 2]^①等,第一个下标通常称为“行下标”,第二个称为“列下标”。

2. 数组的性质

数组作为一种数据类型,具有以下最主要的特性:一是其中的数据元素具有相同基础类型,二是在程序中一旦声明,其规模就不再变化。^②这就让计算机在内存中能够预留出一块连续的地址空间存放数组,这是数组访问高效的基本原因。我们来看其中的道理。

由于数组元素有相同的基础类型(整型,浮点等),它们需要的存储空间一样,设每个元素均占 m 个字节。那么,一个 n 元一维数组 A 就要用到 $n \times m$ 个字节的空間。设 $L(A)$ 为 A 的起始地址,也就是第一个元素(即 $A[0]$)的地址,那么 $A[i]$ 的地址(以字节为单位)就是:

$$L[A] + m * i$$

由于访问内存每个地址上的数据所需时间是相等的(所谓“随机访问”),那么访问数组的任何一个元素都只需要先做一个如上的地址计算,然后访问内存,也就是对每个元素的“访问时间相等”。

如果 A 是一个 $n_1 \times n_2$ 的二维数组,那么它就需要 $n_1 \times n_2 \times m$ 字节的空間。如何确定它的元素 $A[i, j]$ 的地址呢? 通常,二维数组在

内存空间的存放采用所谓“行优先”方式,即先放第 1 行的元素,然后第 2 行……,如图 2.4 所示(假设每行有 4 个元素)。

那么,对于声明为 n_1 行, n_2 列的数组 A,其元素 $A[i, j]$ 的地址就是:

$$L(A) + (i * n_2 + j) * m$$

也能通过一个简单的计算确定。如果 $m = 4$,存储访问以字为单位(1 个字 = 4 个字节),上面的公式就是 $L(A) + i * n_2 + j$ 。

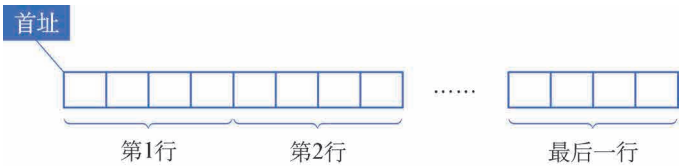


图 2.4 二维数组的行优先存放示意图

① 有的语言不是像这样把下标都放在一个[]中,而是分别用[]表示不同的维度,例如 pic_bw[0][0]。
② 有的程序设计语言有“动态数组”的概念。

绝大多数编程语言都提供对数组的直接支持,但有一个例外,那就是本课程中采用的 Python。Python 有一个更一般、更灵活的概念——列表,可以体现数组的逻辑功能(例如“列表的列表”可看成二维数组,可以 $A[i][j]$ 的方式访问其元素),但由于要支持灵活性,在数据规模较大时就没有其他语言中的数组效率高。为了支持高性能的大规模数据处理,研究人员采用封装的方式在 Python 的基础上实现了包含数组概念在内的程序库——numpy,可通过 import 语句加载调用。

Numpy 中的数组不如 Python 语言本身的列表灵活,例如它在程序运行中不能改变大小(从而也就不能做列表可做的插入和删除之类的操作),但在数据量很大时比列表的访问效率要高。

如图 2.5 所示,这两个程序在逻辑上都是对一块 $row \times col$ 大小的空间中的每个元素做一次访问,形式上完全一致。当 row 和 col 比较小时,由于一些其他因素,图 2.5(b)中程序可能会比图 2.5(a)中程序用的时间短,但当它们比较大,例如 row 和 col 分别都超过 15000 时,图 2.5(a)中程序的效率优势就会体现出来。

```
1 import numpy as np
2 import time
3 row = eval(input("input number of rows: "))
4 col = eval(input("input number of cols: "))
5 a = np.zeros((row,col),dtype=np.int32)
6 start_time = time.time()
7 for j in range(col):
8     for i in range(row):
9         a[i,j] = i + j
10 print('Finished, and time spent: ',time.time()-start_time)
11 exit()
```

(a) 数组访问

```
1 import time
2 row = eval(input("input number of rows: "))
3 col = eval(input("input number of cols: "))
4 a = [[0 for j in range(col)] for i in range(row)]
5 start_time = time.time()
6 for j in range(col):
7     for i in range(row):
8         a[i][j] = i + j
9 print('Finished, and time spent: ',time.time()-start_time)
10 exit()
```

(b) 列表访问

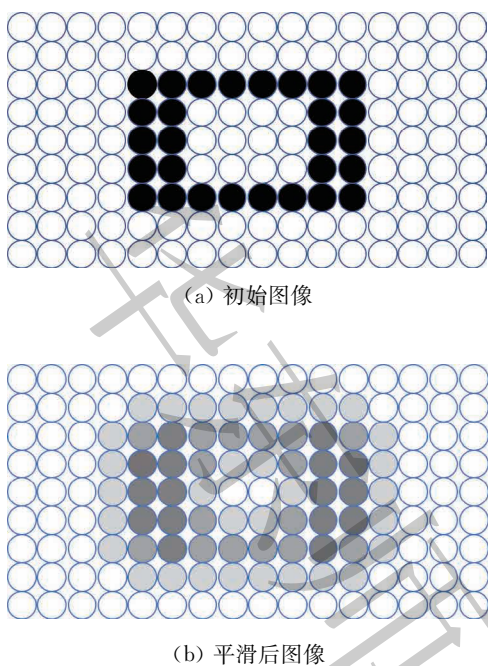
图 2.5 列表和数组访问的性能比较

体 验 思 考

设有一个 5 行 3 列的数组 A,数组元素为整型,存储访问以字为单位, $A[0][0]$ 的地址是 0AH,试分别给出 $A[2][2]$ 和 $A[4][1]$ 的地址。

3. 数组、循环与迭代

结合程序语言中的循环语句,数组在科学和工程计算中有十分广



灰度值示意：黑 = 1, 白 = 0, 深灰 = 0.75, 中灰 = 0.5, 浅灰 = 0.25

图 2.6 图像平滑计算示意图

泛的应用。这类应用又和计算机科学中的一个常用概念——迭代有着密切的联系。下面我们看一个简单的例子：灰度图像的平滑。为了简单起见,我们以一个 9×16 的图像矩阵为处理对象。

如图 2.6(a)所示是初始图像,只有黑、白两种像素,可用一个 9×16 数组表示。所谓图像的灰度平滑,一个简单的方法就是用像素点 $[i, j]$ 周围 4 个像素点的灰度值的平均值取代原灰度值,程序为两重循环结构,循环内进行如下操作:

$$\text{image}[i, j] = (\text{image}[i - 1, j] + \text{image}[i + 1, j] + \text{image}[i, j - 1] + \text{image}[i, j + 1]) / 4$$

在图 2.6(a)的基础上对每个非边沿像素点执行一次上述操作,可得到如图 2.6(b)所示结果。我们能观察到,图 2.6(b)与图 2.6(a)中图像的形貌相似,但整体显得柔和许多。如果在图 2.6(b)的基础上对每个节点再做一次上述操作,也称为再做一次迭代,就能看到灰度进一步平滑的效果。

如果反复对初始灰度图像进行平滑操作,图像最终会达到如下状态:每个像素点的值不再因平滑操作而改变。在程序中则需增加一重循环来控制操作次数或限定其结束条件。

这个例子综合展示了计算机科学中的三个概念:数组、循环与迭代。迭代就是重复做相同的操作,操作对象往往是数组中的不同元素,或者同一个数据元素的不同取值。为实现迭代,采用循环语句是最自然有效的方式。在循环语句的控制下,数组元素不仅会依序得到处理,而且会得到反复处理。图 2.7 从应用目的的角度展示了数组、循环和迭代的关系。

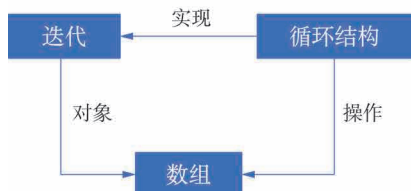


图 2.7 数组、循环、迭代概念应用关系示意图

对于此示意图,有两点值得一提。第一,数组操作不一定是迭代,例如在前面讨论的求身高均值的问题中,数组只参与计算,本身并不更新。第二,迭代过程不一定总涉及数组,例如人们有时需要根据一个函数 $f(x)$ 求一个 x 值,满足 $x = f(x)$ 。一种做法就是预先猜一个初值 x_0 ,代入函数得 $x_1 = f(x_0)$,如果 x_1 和 x_0 足够接近,就认为得到了所求解,否则将 x_1 代入 f 得 $x_2 = f(x_1)$ ……。当 x_n 与 $f(x_{n-1})$ 足够接近

近,则称该迭代过程“收敛”了。事实上,我们前面所做图像平滑的操作,当反复执行的时候,最终就会收敛。

探究活动

写一个程序,用数组存放图像数据,通过循环结构,实现正文中提到的灰度图像平滑应用,并输出第 1、第 2 和第 3 次迭代的结果,观察所发生的变化。

除了上述这类直接的应用外,由于数组在逻辑上体现的是一个多维连续数据空间(通过下标访问),也常常方便地用来模拟(或称实现)其他的一些数据结构。例如,一维数组是线性表的顺序实现的自然选择,而用二维数组能够方便地实现二叉树结构等,这在后面的章节中会多次看到。

在这个意义上,Python 的列表也可以用来模仿数组实现其他数据结构,因为它在逻辑数据空间上和数组有相似的含义。这里的实现指的是不用列表自带的操作,而是通过下标的运用来体现数据元素之间的联系。这种做法,在开发应用程序的时候没有必要采用,但在学习数据结构机理的时候很有意义。

在本模块的学习中,我们会涉及数组、列表、线性表、线性表的顺序表实现、线性表的链表实现等相互关联的概念。数组和列表是程序设计语言中的两个含义相近的数据类型,其中列表在大多数语言中没有,但在 Python 语言中很重要;同时,Python 中没有数组,但在大多数语言中都有。线性表,则是数据结构中的概念,通过插入和删除操作,支持一些算法逻辑的表达与执行。在编程实践中为了实现线性表,可以采用数组或者列表,无论是顺序表实现还是链表实现。这样一些说法,从现在开始到学完第四章后就会逐渐清楚。

作业练习

用数组实现大整数的乘法。使用两个一维数组分别记录两个十进制大整数(数组的每个元素对应一位数),并做这两个大整数的乘法。(注:由于 Python 中的整数可以任意大,实际应用中没有必要在 Python 中专门实现大整数的乘法。此活动的目的是熟悉数组的操作。)

第三节 组合类型

有时候,程序要处理的数据所代表的对象有多种属性,本章项目中涉及的对象“学生”就是一个例子。表达此类似对象的数据元素常常包含不同类型的数据项,或者说该数据元素由不同类型的数据项构成。为支持这种概念的表达,一些程序设计语言中会提供方法,让程序员可以定义组合数据类型,以便于对相关数据对象进行定义和批量处理。组合数据类型也可以成为数组类型的基础类型。

一、数据元素与数据项

请通过以下实例了解讨论数据时可能用到的几个术语。

表 2.2 学生体格数据表

姓名	学号	性别	身高(米)	体重(千克)
李萌	0115	女	1.60	50.0
张简	0327	男	1.75	60.8
王欣	0732	男	1.63	54.5
.....				

若某校学生体格状况的数据以表格形式呈现,如表 2.2 所示。这样的数据有时候也称为一个“数据集”,可能存放在一个文件中,是某个电子表格处理软件的输入或输出。此数据集中的每一个“数据元素”(有时简称“元素”),对应表格中的每一行,而每个数据元素中每一列对应的称为“数据项”。我们说数据元素是数据(集)的基本单位,可由若干个数据项组成,数据项则是最小的可处理单位。

上述概念一般情况下有可能被混合使用。例如可能将“数据集”简称为“数据”;当一个数据元素只有一个数据项的时候,也就不区分“数据元素”与“数据项”了;甚至有时候会用“数据”代指“数据元素”或“数据项”,如人们常说的“这个数据……”,往往指的是某个具体的可操作的数据元素或数据项。

特别值得一提的是,如表 2.2 中所显现的,数据元素的数据项不一定是同类型的。用第二章第一节的基本数据类型来描述的话,姓名是字符串,身高和体重是浮点型(可以进行求平均值等统计运算),性别常常用整型数对应,学号可以是字符串也可以是整型,这取决于

否强调其数值含义。

这样的数据元素在计算机程序中如何表示？此内容正是本节的中心话题——组合数据类型。

体 验 思 考

以药材、药柜和药方为例,说明数据元素、数据项和数据存储之间的关系。

探 究 活 动

试考虑一种数据表示的需求,要求用到 Python 的列表的列表的列表,即三重嵌套列表。

二、组合数据类型的表示

现代程序设计语言通常会提供某种方式,让程序员描述含有不同类型数据项的数据元素,但不同的语言所提供的方式可能有很大差异。例如 C 语言采用 struct 关键字,举例如下:

```
struct student
{
    char name[20];
    int id;
    float weight;
};

int main()
{
    struct student s[99];
    s[1].name = "王小明"
    s[1].id = 103
    s[1].weight = 49.8
}
```

此处定义了 100 个(0~99)有三个数据项(name、id、weight)的数据元素(student),每一个数据元素的各个数据项均可以单独访问(在 main 中通过直接引用数据项示意)。我们特别注意到,100 个数据元素存放在一个数组(s)中。于是也可以说,s 是一个元素类型为 student 的数组。

Python 中没有这种“struct”,但有多种表示数据项类型混合的数据元素的方法,列表是其中之一。下例中即采用列表,读入两个学生的数据(每个学生的数据包含三项内容,name、id 和 weight),并做出了简单的处理。

```
student = list()
for i in range(2):
    name, id, weight = eval(input('Enter next student info: '))
    student.append([name, id, weight])
print('names: ', student[0][0], student[1][0])
print('ids: ', student[0][1], student[1][1])
print('average of weights: ', (student[0][2] + student[1][2])/2)
```

类似于前面的 C 程序,从宏观上看这是一个列表,其中每一个元素对应一个学生的相关数据,而每一个元素也是一个列表,包含有三个不同类型的数据项。这种不同数据类型多层“嵌套”的能力让我们能构造出一个可表达极其丰富数据对象的空间。通过上例我们能看到,程序也是可以直接访问列表中每一个元素的每一个数据项的。

作业练习

结合本章项目情境,设计一个描述学生的数据(元素)表示方案,即需要哪些数据项、各数据项分别是什么类型、可以取哪些值,并完成计算两个数据元素相似度的函数。

第四节 抽象类型

数据类型概念的内涵有两个方面,一是存储,二是操作。前者指相关数据在计算机存储空间中的体现形式,后者指一组特定的(也称为合法的)操作。对于常用数据类型,其操作为人们熟知,例如对于数值型数据,可以做四则运算等基本操作。一般来讲,一个操作可能是若干基本操作的组合,将操作和对应的数据一起以函数或方法的方式封装起来,形成用户定义的抽象数据类型,在使用中能带来许多好处。

一、数据抽象

抽象是人们思考问题的一种基本方式,也是一个含义丰富的概念。例如我们说“科学”是一个抽象的概念,相应有许多具体实例,牛顿第二定律、门捷列夫元素周期表,等等。我们还可以说“动物”是一个比较抽象的概念,猫、狗则是相对具体的,而波斯猫、狼狗则是相对更加具体的,等等。作为一种思维方式,抽象也是一个过程,提炼事物的本质和主要方面,舍去非本质和次要的。

作为计算机问题求解的一个基本方法,抽象主要体现在两个方面。第一,追求问题求解的一般化。例如我们在第一章举的数据流检测例子,虽然只是用 8 个数据解释了两种方法的用法,但我们能意识到它们适合任意 n 个数据。因此,在讨论问题的时候我们虽然都是从有限规模的例子开始,但追求的往往是对任意规模的解法。第二,对于计算机提供的各种功能或服务,追求使用和实现的分离,实现者为使用者提供“抽象的”界面或接口。例如我们在电脑桌面上点击一个图标,相关程序就会被启动,一般用户根本不用管那程序怎么就被启动了,尽管背后实现的是一个相当复杂的过程。

在这一节,我们讨论的数据抽象属于上述第二种抽象的范畴。一般地讲,就是让处理程序只关心数据对象的使用性质,而不用管那些性质是如何实现的。

体 验 思 考

在 Python 中执行如下程序：


```
a = 99999
for i in range(5):
    a = a * a
print(a)
```

你会看到什么？你看到打印出来一个巨大的整数。记得在本章第一节讲基本数据类型的时候，我们讲了计算机中的整数表示范围受 32 位字长的限制，最大的数是 $2^{31} - 1 = 2147483647$ 。现在为什么能有那么大的数呢？这是因为 Python 实现了一种特殊的整数数据类型。该数据类型会根据需要采用多个存储单元来表示整数，理论上可以任意大。我们作为使用者，不用管 Python 是怎么做到的，只管按照整数运算的规则使用就好。于是我们可以说这个数据类型是抽象的。

基于上面的例子，有些同学可能会认为，“抽象数据类型”的实现主要是研发 Python 等编程语言的人需要关注的内容，我们普通编程人员只需要理解使用规则，会用即可。这种想法部分正确。

正确的是，研发程序语言的计算机科学家确实需要对其特别关注，他们需要思考要向用户（我们这些普通编程人员）提供哪些类型，如何保证接口的简单清晰（从而让我们好理解），如何让实现正确高效（从而我们用起来顺畅）。那是相当具有挑战性的任务。

同时，现代编程语言常常会提供机制，让我们可以根据应用需要，自行定义抽象数据类型，实现所谓用户自定义的数据类型。Python 中的类（class）就是这样一种机制。

程序设计语言提供这种机制具有重大意义。一方面，有些人可以专门开发有用的数据类型，形成所谓“类库”，供另一些人使用，从而极大地繁荣了程序语言应用的生态。Python 中 import 语句的执行，通常就是导入他人先前已经编好的类库。另一方面，采用类来组织代码，可以实现所谓“数据的封装”，限定其中的数据单元只能被规定的操作访问，阻止误操作，这对于编写大型复杂的软件十分重要。

二、抽象数据类型的定义与实现

什么是类？我们可以理解为类是可自定义的某些具有相同特征和行为的事物的抽象，而不是具体的实体。比如，人的概念就是对所有人类个体共同特征的抽象。

现在，我们首先通过一个例子展示 Python 中类的定义与使用的最简单情形，然后通过另一个例子来展示类的机制中所体现的数据封装的含义。

1. Python 类的定义与使用

我们前面讲到,数据类型的内涵是数据的特征及其操作,同样地,在 Python 中,类由属性和方法(即操作、函数)组成,其中属性用于表示该类本身所包含的各种特征,方法则用于表达该类的实体行为或功能实现。如,“人”包含的特征有姓名、年龄、性别等,方法有说话、动作、思考等。下面的程序定义了一个 student 类。

```
class student:
    # 定义了类的三个属性
    def __init__(self, name, age, school):
        self.name = name
        self.age = age
        self.school = school
    # 定义了一个 say 方法
    def say(self):
        print('%s 说: 我今年%s 岁, 来自%s.'%(self.name,
        self.age, self.school))
```

其中,“__init__”是规定用来描述属性的。此例中有三个属性,可以根据需要增减。容易引起初学者困惑的是“self”的使用,它与类的实例化机制有关。通俗地讲,它表达的意思是“这一个实例的……”。如果你愿意,也可以用不同的词来代替,例如“this”“this_obj”等,但必须保持一致,必须出现在参数表的第一个位置,且在类定义的后面凡涉及引用属性变量和方法的时候都要用它作为前缀。

在这个类中,我们只定义了一个方法,say(self)。在实际运用中可以根据需要定义任意多个。本书后面的数据结构部分将给出一些其他例子。这里需要说明的是,Python 类的使用是一个内容十分丰富的课题,但不是本课程学习的重点。本课程中,我们仅根据讨论数据结构的需要用到了它的一些基本功能。

一旦有了某个类的定义,使用时首先需要创建一个以它为类型的变量,也称为对象或实例,然后就可访问该变量的属性,调用其方法,示例如下:

```
person1 = student('王小明', 15, '新星中学'); # 用三个属性值创建了一个数据对象
person1.say() # 调用这个数据对象的方法
```

2. 数据封装的一个实例

假设我们需要一种数据类型,它包含三个整数,不妨称为“Triple”。我们只关心三个数中哪一个最大,于是在该数据类型上只有一个方法,不妨称为“win()”。那么,作为抽象数据类型,可如下描述:

```
ADT Triple
    data object: {data1, data2, data3}, a set of three integers
    data operation: win(), return max {data1, data2, data3}
endADT
```

在 Python 中用类实现如下:

```
class Triple():
    def __init__(this, data1, data2, data3):
        this.data = [data1, data2, data3]
    def win(this):
        if this.data[0] >= this.data[1]:
            if this.data[0] >= this.data[2]:
                return(this.data[0])
            else:
                return(this.data[2])
        if this.data[1] >= this.data[2]:
            return(this.data[1])
        else:
            return(this.data[2])
```

于是,用户可在程序中应用如下:

```
a, b, c = eval(input('a, b, c: '))
x = Triple(a, b, c)
print (x.win())
```

用户不仅不用关心 Triple 是怎么实现的,而且即使已经知道了 Triple 是怎么实现的,也只能用 win()来访问一个具有 Triple 类型的数据 x。其他操作会被 Python 认为是非法的。假设用户已知 Triple 是用一个列表实现的,在创建了具有 Triple 类型的数据对象 x 后,用

户试图在程序中访问 `x[0]`, Python 就会报错。这就叫数据被“封装了”, 相当于对数据提供了一种保护, 只允许做规定的操作。

当然, 有时候也需要访问“内部数据”, 例如希望定义如下将两个 Triple 相加的函数:

```
def add_triple(x, y):  
    z = Triple(0, 0, 0)  
    z[0] = x[0] + y[0]  
    z[1] = x[1] + y[1]  
    z[2] = x[2] + y[2]  
    return(z)
```

此函数需要访问 Triple 中的单个数据。程序语言一般也提供相应的支持, 在 Python 中可在类的定义中用 `__setitem__()` 和 `__getitem__()` 来指出这种可能。

探究活动

修改正文中的 Triple 实现, 让下面的程序能正确执行。

```
a, b, c = eval(input('a, b, c: '))  
x = Triple(a, b, c)  
a, b, c = eval(input('a, b, c: '))  
y = Triple(a, b, c)  
z = add_triple(x, y)  
print z.win()
```

项目实践

讨论本章项目情境中数据(各属性)的关系和逻辑结构, 用 ADT(即抽象数据类型)的方法表示它们。收集数据(属性值), 编程实现数据的存储和相似度计算。讨论结果准确性的评价标准, 并根据需要调整方案。

作业练习

用 Python 语言的类结构来定义“复数”的抽象数据类型。

```
class ComplexA(object):
```



第三章

基础数据结构

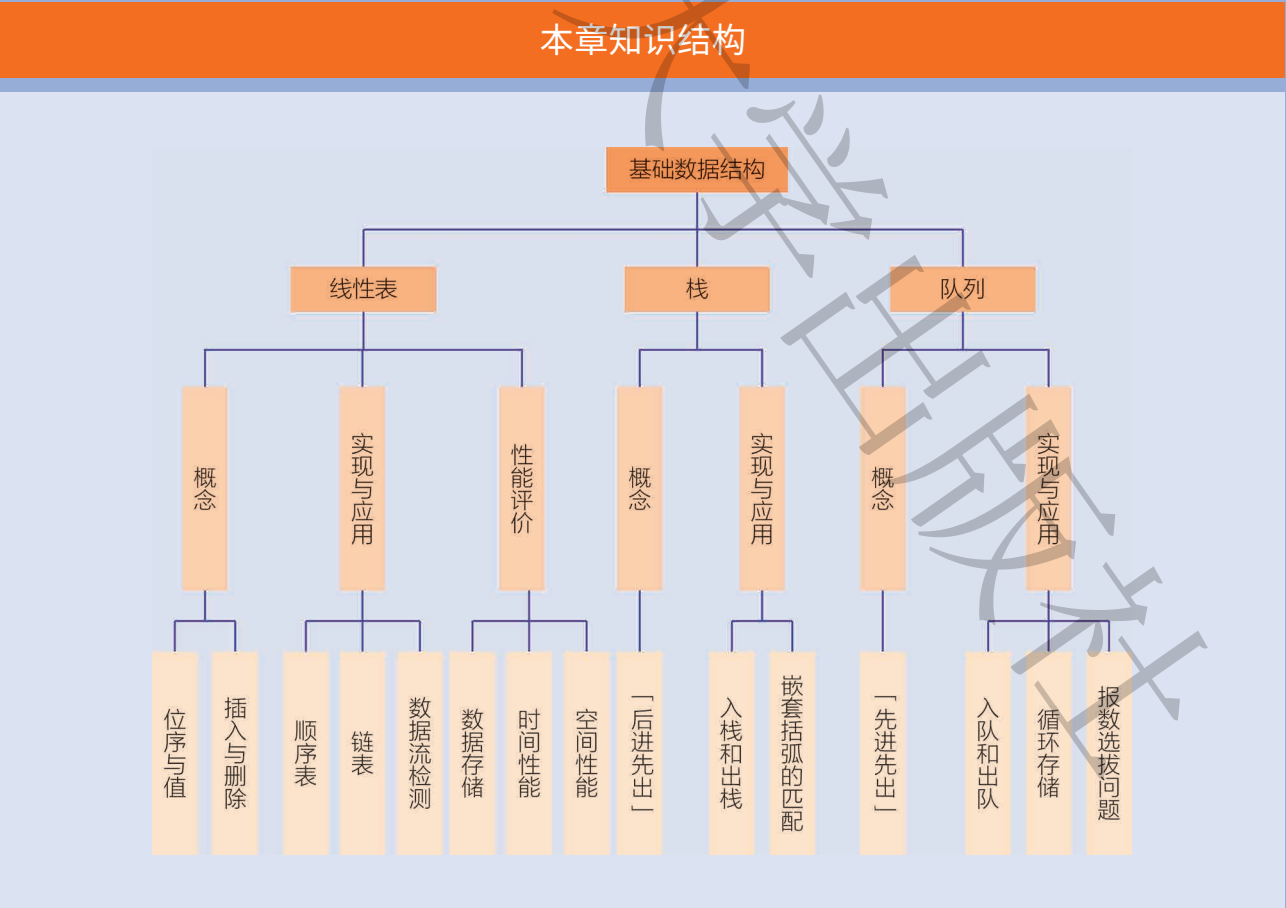
本章学习目标

- 理解线性表、栈、队列三种数据结构的含义、基本操作、应用场景。
 - 掌握实现上述三种数据结构的一种方法,了解不同实现方法的差异。
 - 能够将它们应用于本章项目任务中。
-

数据结构,是表达数据及其之间关系的一个概念。只有在恰当的数据结构支持下,计算机算法才能够实现为高效运行的程序。数据结构中的“数据”,可能直接对应程序输入的原始数据,也可能是程序运行中生成的中间数据。数据结构中表达的关系,可能是原始数据中呈现的自然关系,也可能是算法逻辑所要求的关系。

如在本书引言中指出的,数据结构和数据类型有着密切的关系。与数据类型类似,一种数据结构通常都对应有一组可以施行于其上的操作。就好比对于数值型数据,可以做加减乘除四则运算;对于字符串,可以提取其中的子串等。因此,也可以(但不必须)基于数据结构定义抽象数据类型。通过这两个概念的交织与嵌套,我们得以描述各种复杂的事物,进而让计算机也能够处理它们。

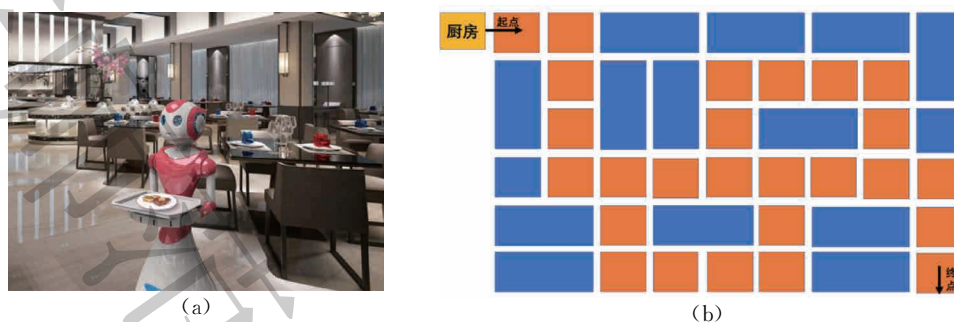
作为数据结构学习的开始,本章学习线性表、栈和队列三种基本数据结构。数据元素之间的结构性关系,对数据元素的访问方式,数据结构本身在特定操作下的动态变化,尤其是与算法互动的基本方式,这些内容是在学习中需要着意体会的。



项·目·情·境

随着信息时代的到来,越来越多的智能机器人进入了我们的工作和生活中。看!不少餐厅开始使用送餐机器人(如图 3.1(a)),这些可爱的机器人在厨房和各个餐桌间来回穿梭,为客人配送菜品。它们是如何做到高效、准确地在餐厅之中行走的呢?

图 3.1 机器人送餐示意图



餐厅通常是由餐桌和通道组成的,于是每一个餐厅都有一个类似于图 3.1(b)所示的平面图。其中橙色表示机器人可以到达的地方,蓝色表示餐桌或其他不能通过的设施。本项目的目标为,给定任意一个类似于图 3.1(b)的平面图,以及一个起始点(例如图中左上角厨房处)和一个终点(例如右下角餐桌处),为机器人规划出一条最佳的行走路线。

项·目·任·务

任务 1

分析项目中的关键步骤,寻找特征,建立数据模型。

任务 2

设计不同的方案,选择合适的数据结构进行组织和存储。

任务 3

选择其中一种方案编程实现,显示一条从任意给定起点到终点的路线,

任务 4

归纳总结机器人送餐方案,探究列举在生活中更广泛的应用。

第一节 线性表

线性表是一种最基础的数据结构,是指在位置上有前后顺序(线性序)的一组数据元素,从而可以界定出第一个数据元素、第二个数据元素……以及最后一个数据元素。程序可以在线性表的任意位置执行插入和删除数据元素的操作,导致其长度的变化。线性表的主要应用体现在其中数据元素的位序与它们值序的关系上。

一、什么是线性表

反映现实世界的数据常常具有一种天然的顺序。例如,每小时测量一次气温,将得到的 24 个数按照顺序写出来最有意义。又如,若有人告诉你下面这些数是 2003 年到 2018 年参加全国高考的人数(单位:万):

(613, 729, 877, 950, 1010, 1050, 1020, 946, 933,
915, 912, 939, 942, 940, 940, 975)

你会很自然地认为它们是按照年份顺序排列的。

线性表,就是表达这类数据的数据结构概念,一般地记为:

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

其中, a_0 称为第一个元素, a_1 为第二个元素……^①;对于 $n > i > 0$,称 a_i 为 a_{i-1} 的后继, a_{i-1} 为 a_i 的前序; n 称为线性表的长度,即其中元素的个数。

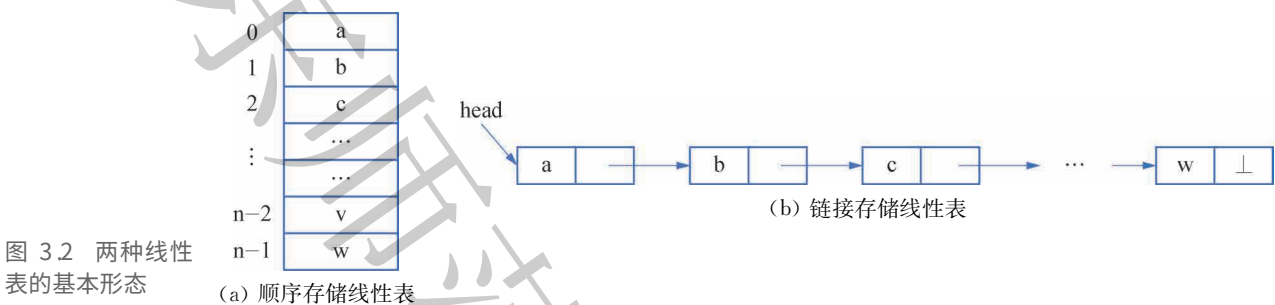
线性表不仅能静态地存放这类数据,还要能支持对这类数据的操作。比如一个班级的学生名册是按照姓氏笔画排序的,学期中转来一个新同学,要把他的名字放入名册中的适当位置(保持整体按姓氏笔画排序的规则),很可能需要把有些同学的名字往后挪一个位置,中间空出一个位置给他。反过来,若一个同学转走了,也需要对名册做出调整。这样的情况,在线性表相关操作上分别称为“插入”和“删除”。为了支持高效的程序设计,线性表上还可以有其他操作,但这两个是最基本、最具标志性的操作,它们的共同点是改变了线性表的长度(即线性表数据元素的个数),具有动态性。

^① 有同学可能好奇,第一个元素的下标怎么不从 1 开始呢? 这是计算机科学中的一个有趣现象,在有些场合,一组数据元素的第一个的确也是从 1 开始编号的,但在许多场合(尤其在现代程序设计中)则是从 0 开始。为什么呢? 当你们学到后面队列一节的时候应该能得到答案。这里,我们不妨先认识这种现象。

二、线性表的实现与应用

1. 线性表实现的基本考虑

一般而言,实现一种数据结构会涉及两个方面,一是如何存储它所包含的数据元素集合,二是如何支持在它上面定义的操作。线性表有两种实现方式:顺序存储实现和链接存储实现。它们的基本形态如图 3.2 所示。



顺序存储实现(简称顺序表)指的是让线性表中的数据元素按顺序存储在计算机中的一片连续空间中,让逻辑上相邻的元素在物理位置上也相邻。这样做的最大优点是在数据元素具有相同类型(因而大小一样)的情形下,对任意元素的访问时间相同,类似于在第二章第二节中学习过的数组,即任意元素的地址都可以通过一个统一的公式(首元素地址加上相应偏移量)得到。

链接存储实现(简称链表)即逻辑上相邻的数据元素的物理位置不一定是相邻的,每一个数据元素附着一个指向下一个数据元素位置的指针,从而体现它们之间的先后关系。链表通过一个头指针(head)进入,要访问链表中的第 i 个元素,需要从头开始,一个元素接一个元素地向尾部方向走,直到遇到第 $i-1$ 个元素。一个特殊符号用在最后一个数据元素的指针部分,标志不再有后继。^①后面会看到,与顺序表相比,链表的一个优点是“插入”和“删除”操作效率较高。

体 验 思 考

找 n 个学生(例如 $n=10$)排成一排站到讲台前,给每人胸前贴上一个号码(1, 2, ..., n , 可以不按顺序)。然后给其中的 $n-1$ 个人每人手里再发一个号码(也是 1, 2, ..., n 中之一),代表“下一个”。最后要

^① 图 3.2 中用的是“ $\perp$ ”,在程序中则可用某个肯定不会是有效地址的任意符号。

求学生们按照手上号码确定的“下一个”关系重新排队(可以计时),即给出对应的线性表。(注意:给学生发的号码与学生胸前号码的对应关系要事先设计好,避免出现“循环”。)

为支持某些应用,链表也可以是“双向”的,即每个数据元素附着两个指针,一个指向后继,一个指向前序。链表也可能是“循环”的,即最后一个元素回过来指向头元素。它们的形态如图 3.3 所示。^①在循环链表中,每个元素都可以作为头元素,哪个元素为头元素,由应用的需要来决定。

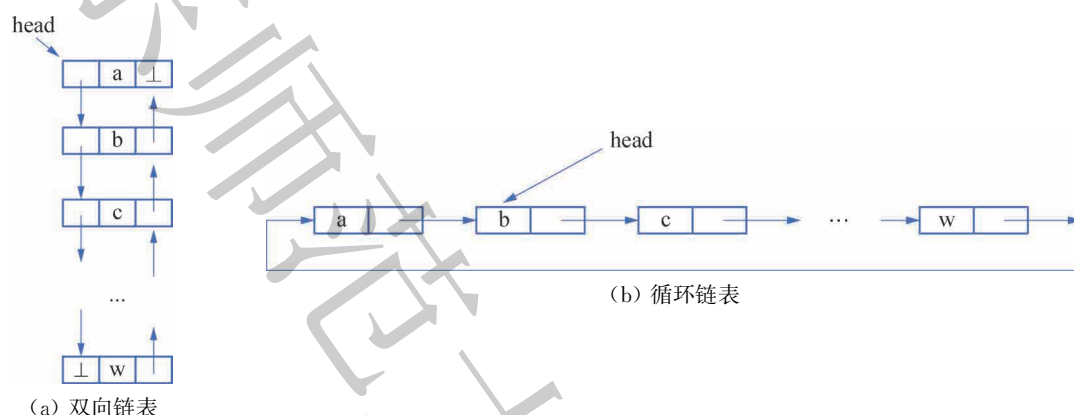


图 3.3 双向链表与循环链表

下面从使用者的角度来考虑一个线性表数据结构应该提供哪些基本操作:

表 3.1 线性表的基本操作

初始化	建立一个空的线性表
表清空	无论线性表是空表还是非空表,都将线性表置为空表
判空表	若线性表为空表,返回 True,否则返回 False
求表长	返回线性表中数据元素的个数
取元素	若 $0 \leq i \leq \text{length}() - 1$, 其函数值为线性表的第 $i+1$ 个数据元素,否则为空元素 NULL
查找	给定值 x ,若线性表中存在一个或多个其值与 x 相等的数据元素,则返回这些元素的位序的最小值,否则返回 -1
插入	在线性表的第 $i+1$ 个位置之前插入一个新的数据元素,将长度为 n 的线性表 $(a_0, a_1, \dots, a_{i-1}, a_i, \dots, a_{n-1})$ 变成长度为 $n+1$ 的线性表 $(a_0, a_1, \dots, a_{i-1}, x, a_i, \dots, a_{n-1})$ 参数 i 的合法取值范围是 $0 \leq i \leq n$,当然需要注意两个特例,当 $i=0$ 时, $(a_0, a_1, \dots, a_{n-1})$ 变成 $(x, a_0, a_1, \dots, a_{n-1})$ 当 $i=n$ 时, $(a_0, a_1, \dots, a_{n-1})$ 变成 $(a_0, a_1, \dots, a_{n-1}, x)$

^① 还可以有“双向循环链表”,想一想怎么修改图 3.3(a),让它表达这个概念。

(续 表)

删除	删除线性表的第 $i+1$ 个数据元素,使长度为 n 的线性表 $(a_0, a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1})$ 变为长度为 $n-1$ 的线性表 $(a_0, a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_{n-1})$ 。参数 i 的合法取值范围是 $0 \leq i \leq n-1$,当然需要注意两个特例,当 $i=0$ 时, $(a_0, a_1, \dots, a_{n-1})$ 变成 $(a_1, \dots, a_{n-1})$,当 $i=n-1$ 时, $(a_0, a_1, \dots, a_{n-2}, a_{n-1})$ 变成 $(a_0, a_1, \dots, a_{n-2})$
----	---

2. 顺序表的实现

存储容量	8
当前元素个数	3
0	1122
1	234
2	3312
3	
4	
5	
6	
7	

图 3.4 顺序表示例

在建立一个顺序表时,一种可能是按建立时确定的数据元素个数分配存储,此法适合创建数据元素个数不变的顺序表,Python 中的元组 (tuple) 就是这样的。如果考虑到顺序表要能够加入和删除数据元素,那么其长度(数据元素的个数)可能会变化,就必须区分顺序表中当前数据元素的个数和数据元素存储区的容量。在建立这种顺序表时,一般是分配一块足以容纳当前需要记录的数据元素的存储块,还应该保留一些空位,以满足增加数据元素的需要。如图 3.4 所示,这是一个具有可容纳 8 个数据元素的存储区,且当前存放了 3 个数据元素的顺序表。

下面讨论顺序表的查找、插入和删除等操作的实现方法。

(1) 创建空的顺序表。用 Python 定义顺序表如下:

```
class SeqList:
    def __init__(self, max = 10):
        self.max = max # 存储容量,默认顺序表最多容纳
        10 个元素
        self.num = 0 # 当前数据元素的个数
        self.data = [None] * self.max # 初始化顺序表,创
        建 max 个元素大小的列表
```

(2) 顺序表清空操作。需要将当前数据元素的个数设置为 0,并且将预定义的数据元素的内容都赋值为 None。

```
def clear(self):
    self.__init__();
```

(3) 判断顺序表是否为空。只需要判断当前数据元素的个数是否为 0。

```
def isEmpty(self):
    return self.num == 0
```

(4) 求顺序表长度。只需返回当前数据元素的个数即可。

```
def length(self):
    return self.num
```

(5) 根据索引获取顺序表数据元素。线性表的基本特征是要能通过索引访问数据(无论是顺序实现还是链接实现)。

```
def getItem(self,i):
    if i < 0 or i >= self.num:
        print("索引越界");return
    return self.data[i]
```

(6) 查找操作。求线性表中值等于给定值 x 的数据元素的位置的最小值,当不存在值为 x 的数据元素时返回 -1 。基本实现思路是:从前向后依次比较各数据元素的值是否等于 x ,直到找到值等于 x 的数据元素或顺序表结束为止。

```
def getLoc(self, x):
    for i in range(self.num):
        if self.data[i] == x:
            return i
    return -1
```

(7) 插入操作。在线性表的第 $i+1$ 个位置之前插入一个新的数据元素 x ,使 $(a_0, a_1, \dots, a_{i-1}, a_i, \dots, a_{n-1})$ 变成了 $(a_0, a_1, \dots, a_{i-1}, x, a_i, \dots, a_{n-1})$,数据元素 a_{i-1} 、 a_i 之间的逻辑关系因此发生了变化。

在线性表的顺序存储结构中,由于逻辑上相邻的数据元素在物理位置上也是相邻的,因此,除非 $i = n$,否则必须移动数据元素才能反映这个逻辑关系的变化。

例如,图 3.5 表示一个顺序表在进行插入操作的前后,其数据元素在存储空间中的位置变化。为了在线性表的第 4 个元素和第 5 个元素之间插入一个值为 18 的数据元素,则需将第 5 个元素到第 8 个元素依次往后移动一个位置。一般来说,在插入操作中移动的时候,需要先将最后一个数据元素向后移动一个位置,然后再将倒数第二个

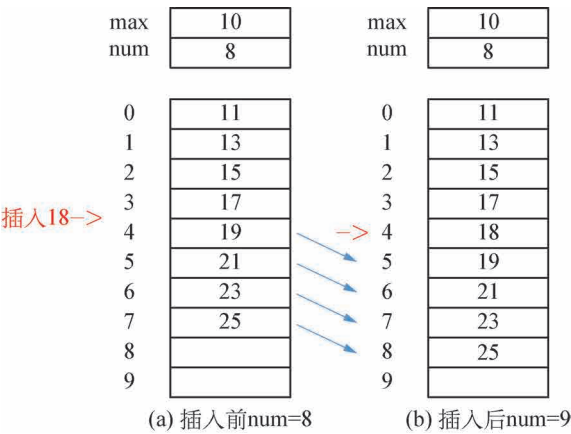


图 3.5 顺序表插入前后的情况

数据元素向后移动一个位置,依此类推。

```
def insert(self, i, value):
    if self.num >= self.max:
        print("顺序表预留空间已满!"); return
    if i < 0 or i > self.num:
        print("索引越界"); return
    for j in range(self.num, i, -1):
        self.data[j] = self.data[j - 1]
    self.data[i] = value; self.num += 1
```

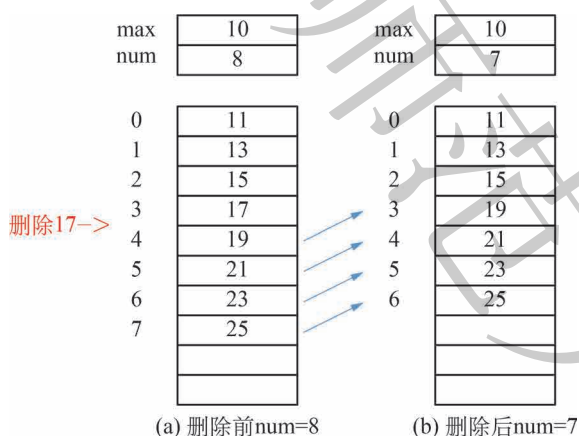


图 3.6 顺序表删除前后的情况

(8) 删除操作。在线性表里删除第 $i+1$ 个数据元素,使 $(a_0, a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1})$ 变成了 $(a_0, a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_{n-1})$, 数据元素 a_{i-1} 、 a_i 、 a_{i+1} 之间的逻辑关系因此发生了变化。对于顺序存储实现而言,为了反映这个变化,同样需要移动数据元素。如图 3.6 所示,为了删除第 4 个数据元素,必须将第 5 个元素到第 8 个元素都依次往前移动一个位置。

```
def delete(self, i):
    if i < 0 or i >= self.num:
        print("索引越界"); return
    for j in range(i, self.num):
        self.data[j] = self.data[j + 1]
    self.num -= 1
```

3. 链表的实现

接下来介绍单链表上的操作实现,先用 Python 定义一个简单的单链表节点类,只包含两个数据域,一个用于存放节点数据,另一个用于指向下一个节点的链接。

```
class ListNode:
    def __init__(self, data, next_ = None):
        self.data = data    # 数据域
        self.next = next_  # 链接域
```

这个类里只有一个初始化方法,它给对象的两个域赋值。方法的第二个参数用名字 `next_`,是为了避免与 Python 标准函数 `next` 重名。这也是 Python 程序中命名的一个惯例。为了方便使用,第二个参数 (`next_`)还提供了默认值 (`None`)。这个 `ListNode` 类会在下面的链表操作中被反复用到。

(1) 创建空链表。只需要把相应的表头变量设置为空链接。在 Python 语言中将其设置为 `None`,在其他语言里也有惯用值,例如有的语言里用 0 作为这个特殊值。

```
class LinkedList:    # 单链表类
    def __init__(self):    # 创建空链表
        self.head = None
```

(2) 链表清空操作。丢弃这个链表里的所有节点。这个操作的实现与具体的语言环境有关。在一些语言(如 C 语言)里,需要通过明确的操作释放一个个节点所用的存储空间。在 Python 里只需要将链表指针赋值为 `None`,即可抛弃链表原有的所有节点,Python 解释器的存储管理系统会自动回收不用的存储空间。

```
def clear(self):    # 清空操作(清空链表)
    self.__init__()
```

(3) 判断链表是否为空。将表头变量的值与空 (`None`) 比较。

```
def isEmpty(self):    # 判断表是否为空
    return self.head is None
```

另外,链表的长度,或者说其中节点的个数,也可以作为链表是否为空的指示。

(4) 求链表的长度。由于单链表只有一个方向的链接,开始情况下只有表头变量在掌握中,所以对表的内容的一切检查都只能从表头变量开始,沿着表中链接逐步进行。人们形象地将这种过程称为链表的扫描。设置变量 `p` 指向表头,再设置一个变量 `cnt` 作计数器,`cnt` 初始值为 0。循环判断:如果 `p` 不为 `None`,则 `cnt` 变量增加 1,将 `p.next` 赋给 `p`。

```
def length(self):    # 求表的长度的函数
    p = self.head; cnt = 0
```

```
while p is not None:
    cnt += 1; p = p.next
return cnt
```

小技巧：从表头节点开始,通过工作指针的不断后移而走过单链表的每个节点的扫描技术是一种常用技术,在许多算法中都要用到。当然,为了避免扫描带来的开销,我们可以在类里添加一个存储长度的变量,在对链表进行相应的操作(如插入元素、删除元素等)时对这个变量进行维护,则可以通过这个变量直接返回表的长度。

(5) 取数据元素。读取链表的第 $i+1$ 个数据元素的值。在单链表中无法直接获得第 $i+1$ 个数据元素的值,只能从表头指针出发,通过链接域往下搜索,直到找到第 $i+1$ 个节点为止。可见单链表是非顺序存取的存储结构。

```
def getItem(self,i): # 获取单链表指定位置的数据
    if i < 0 or i >= self.length():
        print("索引越界");return
    p = self.head;cnt = 0
    while cnt != i:
        p = p.next;cnt += 1
    return p.data
```

(6) 查找操作函数 `getLoc(self,x)` 在单链表的实现与顺序表上的实现相似。在单链表中从前往后查找第一个数据域值为 x 的节点。若找到,则返回该位置,否则函数返回 -1 。

```
def getLoc(self,x): #按值查找元素位置,如果没找到则返回-1
    index = 0
    p = self.head
    while p is not None:
        if p.data == x:
            return index
        p = p.next;index += 1
    print("%s 没有找到" % str(x))
    return -1
```

(7) 链表中的插入。让新节点成为下标为 i 的节点,包括两种情

况, $i=0$ (即插入到表头) 或其他。如果 $i=0$, 即让表头变量指向新节点, 让新节点指向表头变量原先指向的节点 (包括可能的 None)。如果 $i>0$, 则在链表上找到下标为 $i-1$ 的节点 p , 将 p 的 `next` 赋给新节点的 `next`, 将新节点赋给 p 的 `next`。

```
def insert(self, i, x):
    if i == 0:  # 包括当前是空表的情况
        self.head = ListNode(x, self.head); return True
    if i < 0 or i > self.length():
        print("索引越界"); return False
    p = self.head; cnt = 0;
    while cnt < i - 1:
        p = p.next; cnt += 1
    p.next = ListNode(x, p.next); return True
```

(8) 链表中元素的删除。在线性表中删除下标为 i 的数据元素。可以这样操作: 在链表上找到下标为 $i-1$ 的节点 p (如果 i 为 0, 则将表头指针指向表头指针的 `next`), 再将节点 p 的 `next` 指向节点 p 的 `next` 的 `next`。

```
def delete(self, i):  # 删除一个节点
    if i < 0 or i > = self.length():
        print("索引越界"); return
    if i == 0:  # 要注意删除第一个节点的情况
        self.head = self.head.next; return
    else:
        p = self.head; cnt = 0
        while cnt < i - 1:
            p = p.next; cnt += 1
        p.next = p.next.next
```

上面讲解了单链表上的操作实现, 循环链表的操作实现作为课后作业, 双向链表的操作实现在本模块中则不再专门讨论。

体 验 思 考

将上述关于线性表的两种实现方式整理成程序, 并添加如下代码后运行:

<pre> L= SeqList(20) for i in range(12): L.insert(2*%(L.length()+1) i+2) L.delete(5);L.delete(3) for i in range(10): print(L.getItem(i)) </pre>	<pre> L= LinkedList() for i in range(12): L.insert(2*%(L.length()+1) i+2) L.delete(5);L.delete(3) for i in range(10): print(L.getItem(i)) </pre>
---	--

对比这两段程序的差别,探究程序执行的过程,观察程序执行的结果(可根据需要插入打印当前表长的语句),体会“用户不用关心实现细节”这样一个十分重要的观念。

4. 线性表在数据流检测问题中的应用

下面,回到第一章第二节中数据流检测的例子,来看我们上面定义类如何用于完成数据流检测的任务。为此,假设现在已将上面的类定义代码整理好,放到了一个名为 list.py 的文件中。下面是一个数据流检测的程序,我们用一个列表 datastream 模拟数据流的到来,通过 pop() 函数一个个弹出数据供检测。

```

import list as L
datastream = [2,3,6,1,3,8,1,2,4,5]
print('数据流: ',datastream)

linear_list = L.SeqList()
while datastream:
    x = datastream.pop(0);no = True
    n = linear_list.length()
    for i in range(n):
        if x == linear_list.getItem(i):
            print('先前看到过这个元素!',x)
            no = False;break
        if x > linear_list.getItem(i):
            linear_list.insert(i,x)
            no = False;break
    if no:

```

```
linear_list.insert(n, x)
print('结束时线性表中的元素: ', end = ' ')
for i in range(linear_list.length()):
    print(linear_list.getItem(i), end = ' ')
print()
```

在学过了第一章的例子后,我们应该较容易理解这个程序的逻辑。其中一点不同是,在第一章的例子中,新数据总是被放到最后,而这里则是把它插入到适当的位置,从而始终保持着—个有序的数据列表。这里需要特别关注的是第 4 行代码, `linear_list = L. SeqList()`, 它基于前面实现的类,创建了一个由顺序存储实现的线性表。有了前面的“体验思考”活动,同学们此时应该能意识到,将这一条语句改成 `linear_list = L. LinkedList()`,其他都不变,程序运行会得到相同结果,尽管此时改换成了链表实现的线性表。

上面,我们学习的是如何通过 Python 类的方式实现线性表,然后在应用程序中使用。虽然聚焦的是一种数据结构的实现,但同时也感受到了数据类型的作用,即我们可以通过诸如 `x = SeqList()` 和 `x = LinkedList()` 的方式,创建具有所定义顺序表或链表类型的变量,然后对 `x` 做所允许的操作。在这种情况下,我们看到,应用程序(数据流检测)和实现数据结构的程序(类的定义)是分开完成的。

5. 线性表功能的数组实现与应用*

在另一些场合,人们有可能选择在应用程序中直接实现必要的数据结构,而不是专门定义对应数据结构的类,让它和算法一起获得较高的程序执行效率。作为数据结构的学习者,理解这是怎么回事是很有意义的。下面我们继续通过数据流检测的例子,看顺序表和链表的概念是如何体现在应用程序的字里行间的。

先看顺序表的实现。

在讨论程序流程细节之前,先给出其中的几个要点:

(1) 在编程实践中,由于—维数组与顺序表在特征上十分相似,用数组来实现顺序表是常用的方式。它们的主要不同在于作为数据类型的数组通常一旦定义后其大小就不能改变,而作为数据结构的线性表则能够随程序的运行动态改变大小。因此,用数组来实现顺序表时通常要根据应用情况在初始时设定—个较大的数组,允许线性表在其中消涨。

(2) 如何体现“数据流”? Python 的列表(list)所具有的一种功能此

时可用来模拟“数据流”的行为。将测试数据放入一个列表中,通过 `list.pop(0)` 一个个弹出数据至检测程序,直到列表为空,标志数据流结束。

(3) 顺序表用来存放曾出现过的数据元素,起初为空。对每一个待检测的数(x),有三种结果: ①发现表中已存在;②不在表中,但表中存在比它小的元素;③不在表中,表中不存在比它小的元素。对于①中的情况,不做任何操作,接着取数据流中的下一个元素;对于②中的情况,做插入操作;对于③中的情况,将数(x)放至表末。

顺序表实现数据流检测的流程图如图 3.7 所示,完整示例程序见附录程序 3.1。

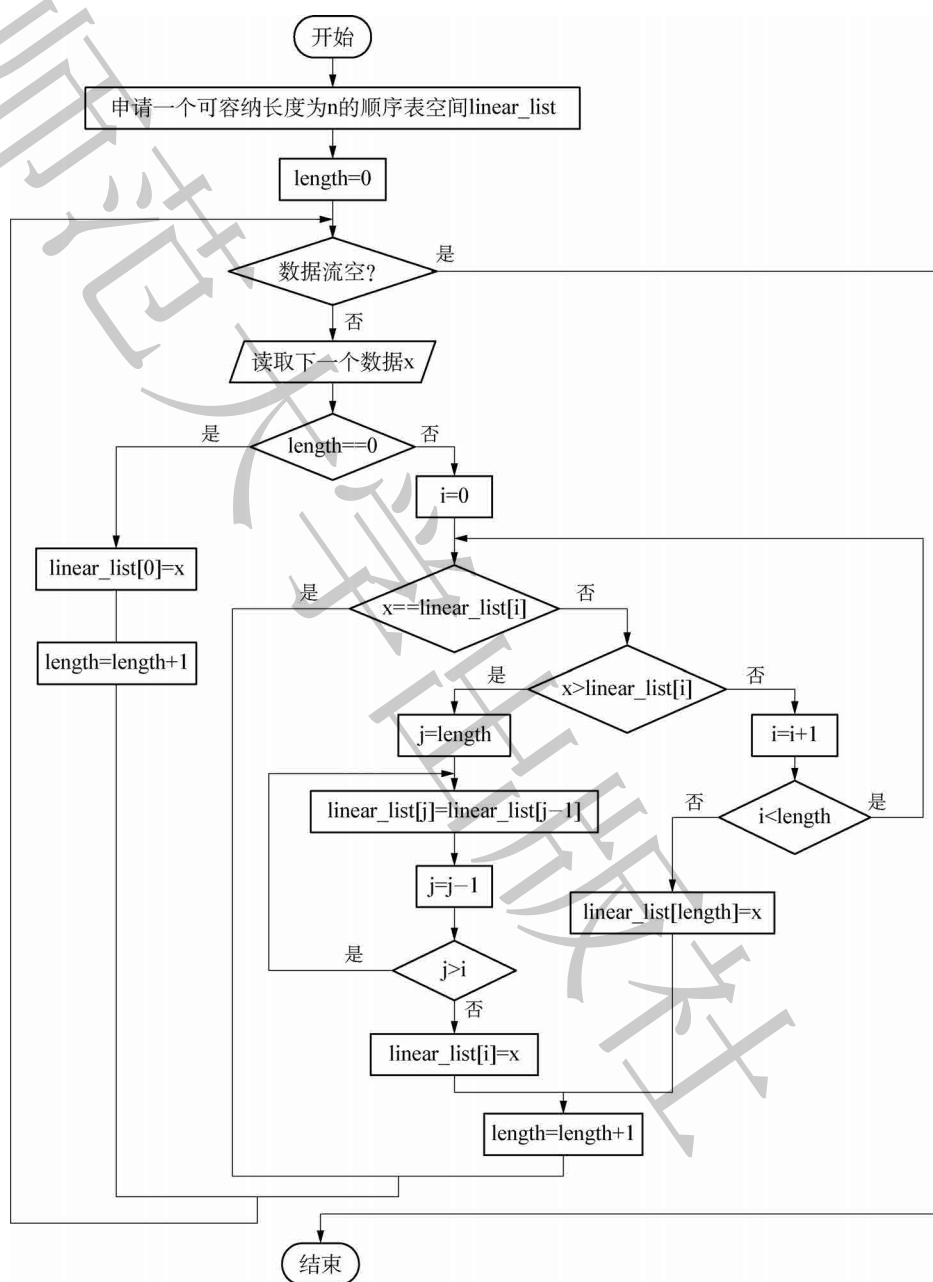


图 3.7 顺序表实现数据流检测

其中,开始后“申请一个可容纳长度为 n 的顺序表空间”比较方便用数组实现, n 的设置取决于对数据流中有多少不相同数据的估计。一般来说,在程序运行中需要检查是否超过这个限制。这里为突出重点,对此未作考虑,在队列一节的学习中对容量问题还会有深入讨论。

体 验 思 考

试用不同的测试数据运行附录程序 3.1。在老师的指导下在程序中插入适当的 `print` 语句,观察程序运行的过程。请思考,若在附录程序 3.1 的第 12 和 13 行之间插入一个 `print` 语句,此语句将被执行多少次?

接下来介绍数据流检测的链表实现方式,如图 3.9 所示,完整示例程序见附录程序 3.2。

先讨论几个要点:

(1) 链表实现要解决的第一个问题是数据元素节点的表示。它要既包含数据元素值,也包含指出下一元素“地址”的“指针”。对于这种要求,二维数组就是一种直观的应对方式。令二维数组包含两列,一列用于存放数据值,另一列用于存放地址。数组的每一行对应一个数据元素节点。这样定义的数组如同一个以数据元素节点大小为单位的线性存储空间。看以下示例。假设有一个含 8 个元素的线性表: 3, 5, 2, 4, 3, 1, 1, 6。图 3.8(a)即用二维数组实现的链表,其中第一列为数据,第二列为该数据后继的“地址”,也就是数组中对应行的下标。据图 3.8(a),从头元素(此例中的行下标为 0)开始,“顺藤摸瓜”,就能得到图 3.8(b)所示展开,正是所需之链表形式的线性表。^①

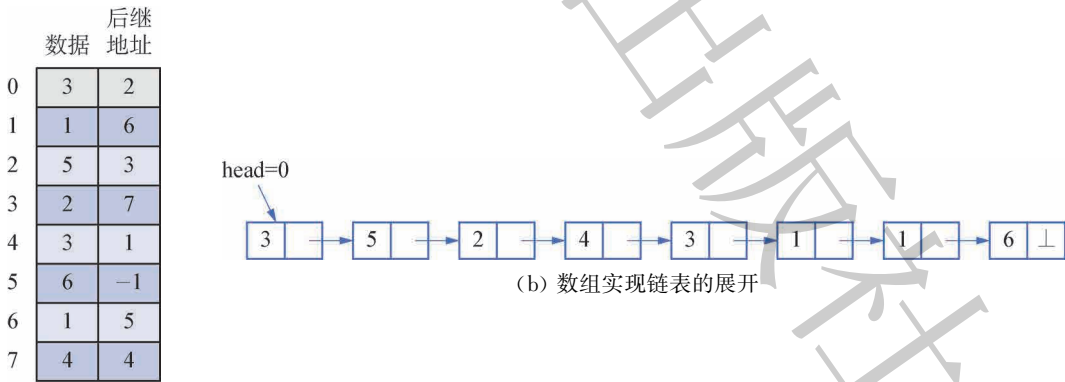


图 3.8 用二维数组实现链表
(a) 用数组实现链表

^① 细心的同学也许能意识到,若给定图 3.8(a),就能够唯一地展开成图 3.8(b)所示形式;但若给出图 3.8(b),反推能得到多种形如图 3.8(a)的数组。当然,从这些数组都只能得到同一个如图 3.8(b)所示展开。

(2) 每当读取一个新数据(x),就要启用该数组中尚未用到的一行(数据元素空间),令 x 为其数据值,并令其指针指向插入点的数据元素,还要调整其他相关数据元素的指针,保持其值序关系。尽管所定义的数组整体上是一个“连续的空间”,但由于数据元素的插入(而不总是接在数组末尾),链表上相邻的节点在数组中就可能不相邻。

(3) 几个重要变量。Linked_list_head: 总是指向链表头。memory: 数据元素集合空间。new: 下一个可用数据元素空间。附录程序 3.2 的宏观流程与顺序表实现(程序 3.1)的情形非常相似。

链表实现数据流检测的流程图如图 3.9 所示。

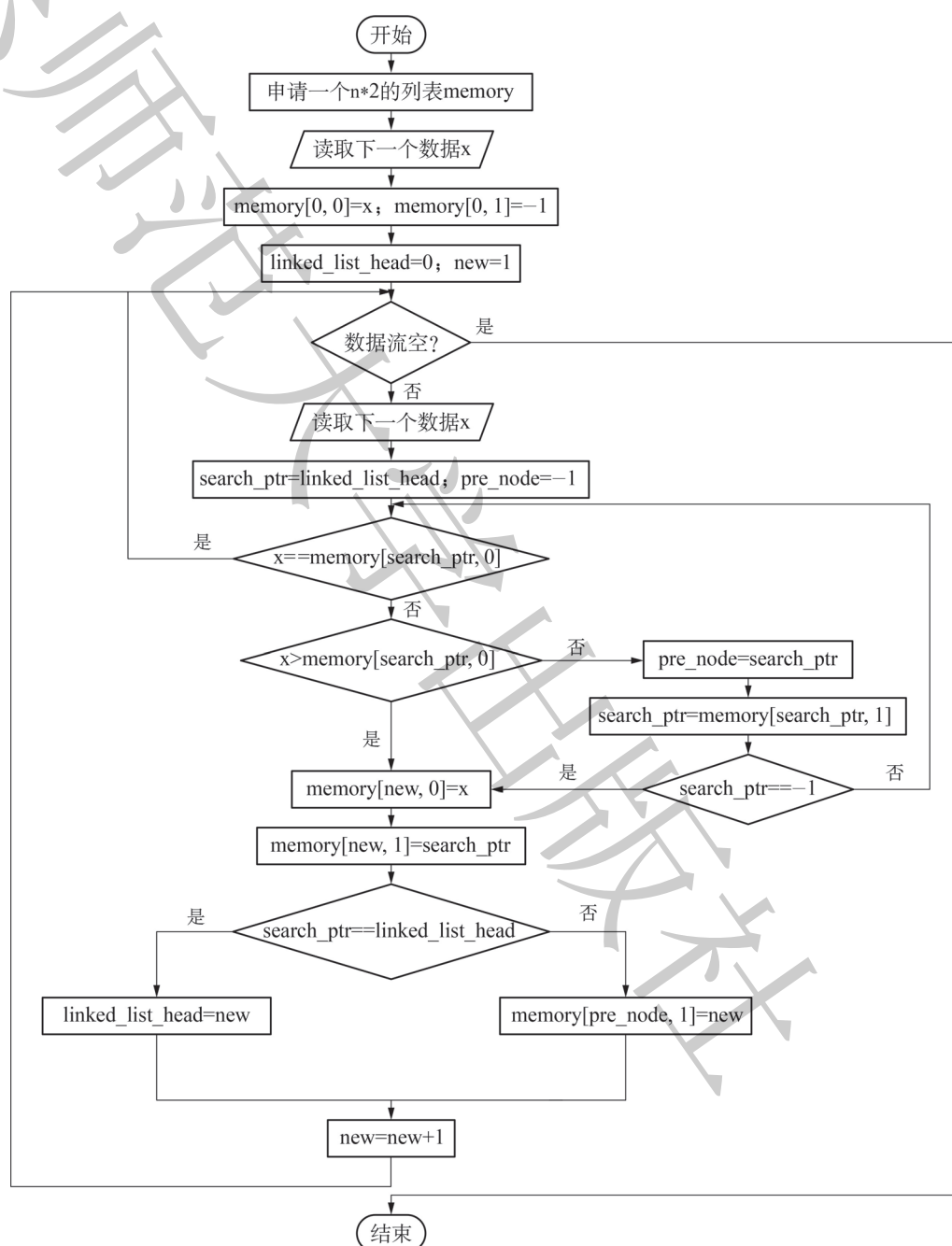


图 3.9 链表实现数据流检测

做一次附录程序 3.2 的人工过程模拟。可以采用第一章第二节第一部分中的例子数据流数据,列表表示 memory 中内容的变化。

三、线性表的性能分析

前面介绍了线性表的两种实现方式,在实际使用中使用哪种方式更合适呢? 对这个问题不能一概而论,下面从三个角度来分析这两种实现方式的优缺点。

1. 数据存储方式

(1) 顺序表的数据元素,按序存放在一段连续的空间中,让程序员能通过下标直接访问每一个元素。

(2) 链表的数据元素,存放在一个个包括链接域的数据单元中,数据元素在线性表中的顺序由链接关系体现,对它们的访问只能是“顺藤摸瓜”式的,不能通过下标直接访问。

2. 时间性能

为了表示数据结构在算法应用中的时间效率,计算机科学家发明了一种所谓“大 O 记法”。其基本含义是,如果待求解问题的规模为 n (不妨认为是输入数据的个数),程序执行主要操作的次数不超过 $k \times f(n)$,其中 k 为任意常数,就说该程序(算法)的时间效率是 $O(f(n))$ 。这样,对线性表的时间效率就可以有以下说法。

(1) 对于查找:

按位序查找时,顺序表可以随机访问,访问任一元素的时间相同,于是可称时间复杂度为 $O(1)$,即“常数时间”;而链表不支持随机访问,平均需要 $O(n)$ 。

按值查找时,如顺序表无序,顺序表和链表的时间复杂度都为 $O(n)$,但是当顺序表有序时,可以采用对分查找(见第五章),将时间复杂度降为 $O(\log_2 n)$ 。

(2) 对于插入和删除：
顺序表平均需要移动表长一半的数据元素，时间为 $O(n)$ 。
单链表在得到某位置的指针后，插入和删除时间仅为 $O(1)$ 。

3. 空间性能

- (1) 顺序表需要预分配存储空间，分配多了浪费，分配少了则容易发生上溢，而上溢的处理在计算机内的开销较大。
- (2) 单链表不需要预先分配存储空间，只要有，就可以分配，元素个数不受限制，但每个数据单元都有一个附加的链接域，会消耗存储空间。

综上，在实际应用中对线性表的两种实现方式的选择可有如下一般原则。如果线性表需要频繁查找且很少进行插入和删除操作，宜采用顺序表。如果需要频繁插入和删除，宜采用单链表结构。当线性表中的元素个数变化较大或者数量难以预判时，采用单链表结构可能更有利。而如果事先知道线性表的长度，如 1 年有 12 个月，一周有 7 天，使用顺序表会更有效率。

总之，顺序表和单链表结构各有优缺点，需要根据实际情况进行选用。特别值得一提的是，与顺序表相关的一个概念是第二章第二节介绍的字符串。由于其基础元素是按一定规范编码的字符，具有相同的存储需求，且所涉及操作大多为读取或片段读取，没有插入和删除，于是在内存中通常采用顺序存储，与顺序表十分相似。鉴于此，有些场合中会把字符串归为线性表一并讨论。

项目实践

餐厅中有餐桌和通道，分析这些特征，抽象餐厅的布局并建立成可处理的地图数据模型。在机器人沿着通道前行过程中，建立各方向移动的数据模型。

作业练习

1 假设有一个含 8 个元素的线性表：2, 3, 6, 8, 1, 7, 4, 5。设计一个用二维数组实现的链表，并画出其展开形式。

2 线性表的核心操作是“插入”与“删除”。在本节的数据流检测例子中,展现了插入操作的意义和实现的方式,但没有涉及线性表的删除操作。下面“报数选拔问题”的求解则是体验在线性表中做删除操作的经典例子。

报数选拔问题: 设想 n 个同学坐成一圈, 按照一个事先确定的报数最大值 m , 从任意一个指定的同学开始, 沿逆时针方向(可循环)报数: 1, 2, ..., m , 第 m 个同学出列; 然后从他右边的那位同学开始报数: 1, 2, ..., m , 也是第 m 个同学出列; 如此下去, 直到剩下最后一个同学。

以图 3.10 中的设定为例, 有 7 名同学(A, B, C, D, E, F, G)围坐一圈, 假设 $m=3$ 从 C 开始。上述报数选拔的出列顺序就是: A, E, B, D, F, C, 最后剩下 G。此过程可以用一个循环链表来实现, 其中的“出列”行为对应于在链表中做“删除”操作。试完成此程序设计。

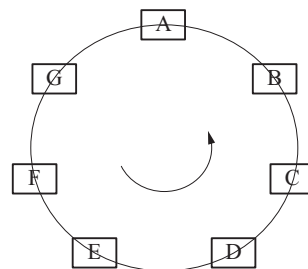


图 3.10 报数选拔问题示意图

第二节 栈

随着使用计算机求解问题的经验不断丰富,人们一定会意识到,在程序运行过程中常常需要依次“记住”一些中间状态,然后以一种相反的顺序再对这些状态一一进行处理,这被称为“后进先出”。支持这种需求的数据结构称为“栈”。所谓“后进先出”的概念,体现在生活中就如同往收纳箱里堆放书,放的时候从下往上叠放,用的时候则需将上面的书先拿出后才能取出下面的书。

一、什么是栈

第三章第一节中我们学习了线性表,它的基本特点是其中的数据元素具有线性序(即元素的位置是依次排列的)。而线性表的动态变化,是通过在该序中的任意位置上做插入和删除操作来实现的。

有时候,程序的逻辑不需要在任意位置上添加和删除操作,只需要在线性表一端操作即可。人们将这样的线性表称为栈。同时,将对栈元素的添加和删除操作分别命名为 push(入栈,压栈)和 pop(出栈,弹栈)。下面以将十进制数转换为八进制数为例,体会栈与程序逻辑的自然配合。

例 3.1 数制转换:按照数制转换算法,要将十进制数 N 转换成八进制数,需要不断用 8 除 N 及其商,保留余数,直至商为 0。这里的重点是,除法运算后需将保留的余数从高位到低位写成八进制数,次序恰好与保留时的次序相反(即保留的第一个余数是最低位,最后一个余数是最高位)。也就是说,可以从一个空线性表开始,每得到一个余数,就添加到其尾部,最后输出时从尾部开始,输出一位删除一位,直到线性表为空。利用 Python 列表的功能,可以写出如下程序^①:

```
n = int(input())
stack = []
while n != 0:
    r = n % 8
    n = (n - r) // 8
    stack.append(r)           # 添加,也就相当于压栈(push)
```

^① 其中用到了 Python 中在线性表上定义的操作,append()相当于添加,pop()相当于删除。这些操作会在下文中详细讲解。

```
while stack:
```

```
    print(stack.pop())
```

```
    # 倒序出栈(pop)
```

运行这个程序,如果你输入 234,将得到输出 352,即 $(352)_8$ 。而且,你应该可以意识到,改变输入(即变量 n 的值),会得到不同的输出,尤其是输出数字的个数可能会不同^①,意味着线性表(此处即

栈)stack 的大小是按需自动调整的。由此例可知,虽然栈的操作只是发生在线性表的一端,但通过栈操作可以轻松改变数据元素的顺序。

对于栈,进行插入和删除的一端称为栈顶,另一端称为栈底。在栈顶插入元素叫做入栈(压栈);删除栈顶元素叫做出栈(弹栈)。因为最先出栈的元素是最后入栈的,所以栈又称为后进先出(last in first out)的线性表,简称 LIFO 结构(如图 3.11 所示)。本书关于栈的图示,如果纵向表示,约定上面为栈顶,下面为栈底;如果横向表示,则约定右边为栈顶,左边为栈底。

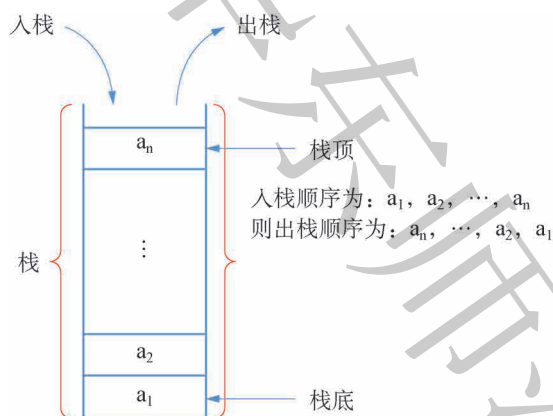


图 3.11 栈示意图

二、栈的基本操作

出栈和入栈是栈的基本操作。上一节的例子告诉我们,这样的操作能够将一个数据元素序列倒序排列。事实上,通过栈操作改变元素的顺序有许多有趣的应用。

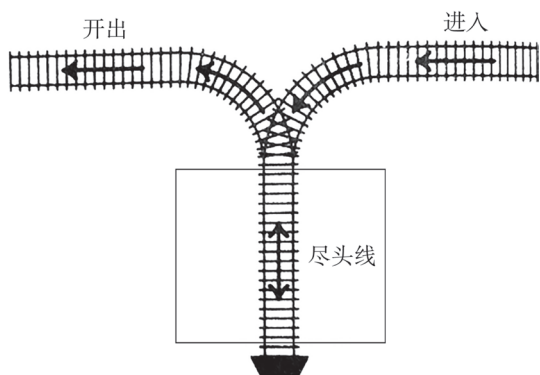


图 3.12 列车调度示意图

例 3.2 列车调度:在列车调度中,常常需要利用铁路上的一段“尽头线”将车厢进行重新编组,如图 3.12 所示。

初始的列车从右边来,按照一定的步骤,将车厢开进和驶出(往左)尽头线,最终形成一个不同的车厢序列。现在假设一列火车有四节车厢,顺序编号为 1, 2, 3, 4,从右边来。现在希望能以 1, 4, 3, 2 顺序从左边出去(即改变了原序列顺序)。如何实现?我们意识到,这里的尽头线相当于一个栈,车厢只能后进先出。下面即为一个有效操作序列。

^① 将程序中的 `n = int(input())` 语句替换为 `n = input()`,可尝试不同的数字。

1 进入尽头线
1 出尽头线
2 进入尽头线
3 进入尽头线
4 进入尽头线
4 出尽头线
3 出尽头线
2 出尽头线

上面这个例子,相当于通过基本栈操作(push 和 pop)的交织安排,改变了输入数据序列的顺序(不只是如例 3.1 中数制转换那样的简单反序)。那么,是不是任何序列都可以通过栈操作得到呢?不是的。在上例中,出栈序列 1, 4, 2, 3 就无法得到。在整个过程中,数据可以分成三种状态:栈前、栈中、栈后。当某个数字出栈了,说明比它小的数字要么已经出栈了,要么还在栈中,不能处于栈前状态,并且栈中的顺序肯定是从大到小(从栈顶往栈底看),因为入栈是按从小到大的顺序操作的。如出栈 4,要么 1、2、3 已经在 4 之前出栈了,要么还在栈中(假如 2、3 在栈中,从栈顶往栈底看依次为 3、2,那么后面的输出只能是 3、2,而不能是 2、3),不能处于栈前的状态。可以得出,在上例中如果某个数字要出栈,那么当前在栈中的数字都必须小于它。

推广到一般的规律,已知栈的输入序列是 $1, 2, 3, \dots, n$, 输出序列是 $a_1, a_2, \dots, a_i, \dots, a_n$ 。然后任选一个数 a_i , 并筛选 a_i 到 a_n 之间所有不大于 a_i 的元素,则它们一定是按照从大到小的顺序排列的(“从大到小”不一定紧紧相邻,只代表它们的相对位置关系,比如 $\dots, 10, \dots, 7, \dots, 3, \dots, 1, \dots$)。

除了入栈和出栈两个标志性基本操作外,栈在实际运用中还需要几个必要(或者方便)的操作。其中一个重要操作是判断栈中是否有元素(即栈是否为空),在例 3.1 中已经出现过(程序的第 7 行)。其他操作会在下文中出现。

体 验 思 考

序列 1 2 3 共有 6 种可能的排列: 1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1。根据第三章第二节内容可知,利用栈操作,不可能从 1 2 3 得到 3 1 2 但其他 5 种都能得到,例如,执行操作 push push pop pop 将实现 1 2 3 $\rightarrow$ 1 3 2。试分别给出从 1 2 3 到其他 4 个序列的栈操作过程。

三、栈的实现

当我们说实现一种数据结构时,可能有两方面的含义。一是在应用程序的“字里行间”通过安排一块存储区,以及必要的“管理性”变量,直接实现该数据结构对程序逻辑的支持。此时,对数据结构的操作体现为在程序中对那些变量的更新。二是通过设计相关的函数(操作、方法),让对数据结构的操作体现为对函数的调用。第三章第二节例 3.1 中的程序,就是利用 Python 内部已经实现的函数表示的。

下面先通过改写例 3.1 中的程序,即实现数制转换,但不使用 Python 提供的对列表可执行的 append 和 pop 功能,了解栈的思想是如何直接体现在程序中的,然后学习通过定义函数实现栈的方法,程序如下:

```
stack_size = 5
stack = [0 for i in range(stack_size)]
top = -1
n = int(input()) # 可以是例 3.1 中用的 234,也可以是其他数据
while n != 0:
    r = n % 8
    n = (n - r) // 8
    if top < stack_size - 1:
        top = top + 1
        stack[top] = r # 入栈
    else:
        print('The stack is already full !')
        exit()
while top >= 0:
    print(stack[top]) # 出栈
    top = top - 1
```

观察这个程序,第 1~3 行获得一个用于栈的存储空间(stack),其中的 stack_size 是容量,即最多可能容纳的元素个数(这里用了 5),用 top = -1 表示空栈。程序的 5~13 行是不断求余数并入栈的过程,并判断了是否出现栈满(如果输入数据太大以至于 5 位不够)的错误。

14~16 行是出栈过程,倒序输出前面求得的余数。这里,对于栈空的判断隐含在 while 语句中,即 $top < 0$ 意味着栈空,程序结束。由此可知,在第 3 行将 top 初始化为 -1,以及第 9 行和第 10 行顺序不能颠倒是多么重要,即 $top = -1$ 表示空栈,当栈不空的时候,top 总是指向栈顶元素。^①

探究活动

在第三章第二节第三部分第一段程序,即通过定义函数实现栈的功能的程序中第 8 行处判断条件是 $top < stack_size - 1$ 。为什么要有“- 1”呢? 试把它改为 $top < stack_size$,并提供一个输入数据,使程序在输入此数据时出错。

与第三章第二节例 3.1 中的程序相比,上面改写的例 3.1 程序要复杂不少,但它体现了栈操作的所有细节,push 和 pop 操作隐含在对栈顶指示变量 top 的更新中。其中一个重要的方面,就是在出栈时需要判断栈是否空(empty),在入栈时需要判断栈是否满(full)。

像这种将栈(以及其他一些数据结构)操作直接“嵌入”应用程序中,会使程序逻辑显得比较凌乱并容易出错,因此在实践中常常将它们用函数或者类的方式“封装”起来,应用程序只需调用函数或方法,而不需关心函数或方法内部的操作细节。

下面看一种用类实现的方式。目标是设计一个 Stack 类,针对数制转换程序,只需 push, pop, empty 这三个方法。基于这个类,编写的数制转换程序和例 3.1 中程序几乎完全一样,即:

```
n = int(input())
stack = Stack()
while n != 0:
    r = n % 8
    n = (n - r) // 8
    stack.push(r)
while not stack.empty():
    print(stack.pop())
```

^① 也可以让 top 总是指向“下一个进来的元素”所要占用的数据单元。如果那样做,此程序需要在好几个地方做修改才行。

此程序中的 `stack` 是一个自定义类 (`Stack`) 的对象, 而不是 Python 中的列表。在类 `Stack` 的定义中, 对于入栈操作, 使用更有针对性的 `push` 作为函数的名称, 而不是用 Python 列表中自带的, 为和较通用的 `insert` 相区别的 `append` 名称进行命名。下面就是这样一个 `Stack` 类的定义。其中, `num` 变量相当于本节中通过定义函数实现栈的功能程序中的 `top` 变量, 它总是指向“下一个进来的元素”所要占据的数据单元。

```
class Stack:
    def __init__(self, max=50):
        self.max=max
        self.num=0
        self.data=[None] * self.max
    def push(self, value):
        if self.num >= self.max:
            print('The stack is already full!')
            return()
        self.data[self.num]=value
        self.num=self.num+1
        return()
    def pop(self):
        if self.num <= 0:
            print('The stack is already empty!')
            return()
        value=self.data[self.num-1]
        self.num=self.num-1
        return(value)
    def empty(self):
        return(self.num==0)
```

四、栈的应用

编写程序时, 常常会用到一些带括号的计算表达式。有时候表达式很长, 括号套括号, 写了左括号忘了在适当的地方加右括号, 或者反过来, 右括号写了但没有匹配的左括号。这样的情况并不少

见。因此,自动判断一个表达式中的括号是否正确匹配是程序开发环境通常都支持的一种功能。下面来看这个功能如何通过栈轻松实现。

例 3.3 括号匹配:假设有一个表达式由括号()嵌套组成,并且嵌套顺序是任意的,但左右括号必须一一匹配,例如((()((()()))这样的格式是正确的,(((()((()这样的格式是不正确的。现有表达式 $(6+(2+4)*(3/(1+2)-(3+2)))$,请编写一个程序,判断表达式括号匹配是否正确。

分析设计思路:把表达式中的每个字符抽象成一个数据元素,数据元素采用栈结构来存储。对于表达式 $(6+(2+4)*(3/(1+2)-(3+2)))$,提取括号得到 $((()((()()))$,要求左右括号必须匹配,所以从左往右依次每出现一个右括号都可以匹配最近出现的左括号,匹配后消除这一对括号。在此过程中,先出现的左括号,需等后面的左括号匹配完成才能进行匹配,因此满足后进先出的原则,可以使用栈来实现,此处实质是用栈暂存左括号。具体步骤如下:

- (1) 建立一个栈,用来存储尚未匹配的左括号。
- (2) 遍历字符串,遇到左括号则入栈,遇到右括号则出栈一个左括号进行匹配。
- (3) 重复步骤(2)的过程中,若栈空但遇到了右括号,说明缺少左括号,不匹配,结束。
- (4) 若字符串遍历结束时栈不为空,说明缺少右括号,不匹配,结束。

剔除 $(6+(2+4)*(3/(1+2)-(3+2)))$ 中和括号无关的字符,得到 $((()((()()))$,以此为例模拟栈的变化过程,如图 3.13 所示。

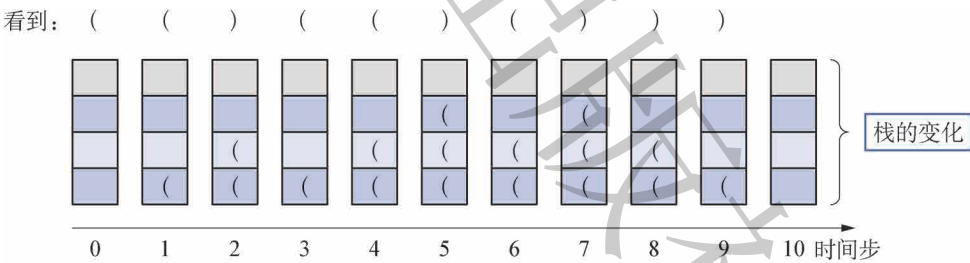


图 3.13 利用栈操作检测表达式括号匹配过程示意图

从以上过程可以看出,在输入符号串结束后恰好栈空,说明 $(6+(2+4)*(3/(1+2)-(3+2)))$ 这样一个表达式左右括号是匹配的。完整的程序见附录程序 3.3。

算法复杂度分析:如果表达式字符串长度为 n ,需要依次遍历整个字符串,因此时间复杂度为 $O(n)$ 。极端情况下,字符串前半半入

栈,后一半依次和出栈元素比较,使用栈空间大小是 $n/2$,因此空间复杂度也为 $O(n)$ 。

项目实践

从厨房到餐桌往往有多条路径。从通用性角度考虑,找出任意一条从起点到终点的路径,设计方案并编程实现。

作业练习

1 完成一个程序,令其输入是从 $1 \sim n$ 的 n 个数字的任意排列,功能是检查输入的排列是否能在序列 $1, 2, \dots, n$ 的基础上通过栈操作得到。如果能,就给出栈操作序列(例如 `push, push, pop, push, pop, ...`);如果不能,则指出出现失败的位置。进一步思考:上述情况隐含地假设了栈是无穷大的,当加入栈的大小限制条件时,例如 m ,情况会有什么变化?

2 多括号匹配:假设有一个表达式是由三种括号 `() [] {}` 嵌套组成的,并且嵌套顺序是任意的,但左右括号必须一一匹配,例如 `[]((() )]` 这样的格式是正确的, `[]((() )]` 这样的格式是不正确的。请编写一个程序,判断一个表达式字符串 `expr= "[ 1+ 2 ] * [ 3 / 1- 2 * ( 3+ 2 ] ]` 的括号匹配是否正确。

第三节 队列

计算机程序在运行过程中,常常会涉及一些“子任务”,它们在程序开始运行的时候并不明了,往往根据数据的情况,在运行过程中逐步出现,尤其是在解决一个子任务的时候可能又出现一批新的子任务。记住那些子任务,按照出现的次序一一解决,就是算法逻辑要面对的问题。支持这种需求的数据结构称为“队列”。队列在生活中好比在售票窗前排队买票的队伍,先到的人先买,买完后离开队伍,后到的人排到队伍的后面等待。

一、什么是队列

队列(queue)作为一种数据结构,和上一节学习的栈一样,也可以看成一种特殊的线性表。其特殊之处在于对它的操作只发生在头(head)和尾(tail)两端^①,且在头端只做删除,在尾端只做插入(追加)。为方便起见,人们为这两个操作起了形象的名称:“出队”(或“离队”)和“入队”(或“进队”)。数据元素出队后,其后继元素就成为新的队头元素;数据元素进队后就成为新的队尾元素。

队列在计算机系统工作的过程中有多种用途。一是如本节导言中提到的,在一个程序执行过程中管理子任务(让待完成的新任务入队,已完成的任务出队)。这种情形典型地体现在第四章的广度优先搜索中。二是可用资源的动态管理,例如计算机内存由操作系统进行动态分配时,一个程序开始执行时要向操作系统申请一定的内存,执行过程中也可能需要追加申请,而在结束时要把它占用的内存释放给操作系统,一个“当前可用内存空间队列”就是有效管理这个过程的数据结构。三是在多个程序之间保证必要的协调。例如,在键盘上输入字符,屏幕上就会显示相应的字符。这个过程中,实际上使用了两个程序:分别负责接收键盘输入和在屏幕上显示内容。这两个程序之间就有一个队列,前者不断将接收到的输入字符加入队尾,后者不断把字符从队头取出,从而保证了在屏幕上显示的顺序与键入的顺序一致,也在两个程序运行速度不一样时起到缓冲作用。队列的这种保证数据出入顺序一致的性质也被形象地称为“先进先出”(first in first out,简称 FIFO)。

通常,队列的图示如图 3.14 所示(其中前后继关系的实现可以采用顺序表方式,也可以采用链表方式)。如无特别交待,本书中所有队

^① 有时也称“前”(front)和“后”(rear)。

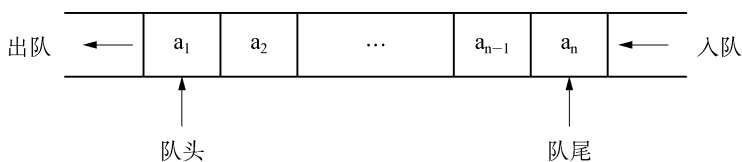


图 3.14 队列示意图

列图示约定队头在左边。

队列除了上述三个方面的应用外,也可以直接用于求解一些具体问题。下面是一个有趣的例子。

例 3.4 编码解密: 已知有一串加密过的微信号为“43532469”, 解密的规则是: 首先将第 1 个数删除,紧接着将第 2 个数放到这串数的末尾,再将第 3 个数删除并将第 4 个数放到这串数的末尾,再将第 5 个数删除……直到将最后一个数删除。把这些删除的数连在一起就是解密后的微信号。解密过程模拟如下:

- (1) 起始时这串数为 43532469。
- (2) 删除 4,并将 3 放到尾部,这串数更新为 5324693。
- (3) 删除 5,并将 3 放到尾部,这串数更新为 246933。
- (4) 删除 2,并将 4 放到尾部,这串数更新为 69334。
- (5) 删除 6,并将 9 放到尾部,这串数更新为 3349。
- (6) 删除 3,并将 3 放到尾部,这串数更新为 493。
- (7) 删除 4,并将 9 放到尾部,这串数更新为 39。
- (8) 删除 3,并将 9 放到尾部,这串数更新为 9。
- (9) 最后删除 9。

将数按被删除的顺序排列得到 45263439,此即解密得到的微信号。以上解密过程实际上就是将这些数组织为一个队列,每次从最前面拿去 2 个,第 1 个数出队后保存到一个字符串变量中,第 2 个数入队放到尾部。重复出队和入队就实现了对数据的重新组织。

二、队列的实现

如前所述,队列可以看成一种特殊的线性表,因此,队列也有顺序存储和链表存储两种方式。下面以完成一个框架性程序为目标,讨论队列的一种实现方案。

例 3.5 队列操作: 完成一个程序,它针对一个初始为空、容量为 n 的队列,取任意数字和字符‘x’交织的序列为输入,其中数字表示入队,字符‘x’表示出队。入队时若发现队列已满,就发出警告,且放弃当

前输入元素；当出队时发现队列为空，也发出相应警告，否则就让头元素出队。无论当前队列满还是空，都进入下一个输入元素，直至结束。例如，若 $n = 4$ ，对以下输入

1, 2, x, 4, x, x, x, 2, x, x, 3, 5, 1, x, 3, 5, 7, x

程序应该给出的输出是：1, 2, 4, 'empty!', 2, 'empty', 3, 'full!', 5；且最后队列中留下的是 1, 3, 5。（可尝试用不同输入得出相应的输出。）

体 验 思 考

有一个容量为 4 的队列，试设计一个长度为 10 的入队和出队混合的操作（分别用 in 和 out 表示）序列，在执行这些操作时，恰好出现 3 次“队列满”警告和 2 次“队列空”警告（次序不限）。

针对队列的实现，既可以如同本章第一节中实现线性表那样，直接将相关代码嵌入到应用中，也可以采用本章第二节实现栈的方式，构造一个函数或定义一个类。这里我们用一个函数来实现队列^①，通过参数指出要对它执行的操作。程序的基本语句如下（设用 input_sequence 模拟输入序列）：

```
while input_sequence:
    e = input_sequence.pop(0) # 此处借用 Python 的功能做模拟
    if e == 'x':
        print(queue('out', h)) # h 在此只是占位符
    else:
        queue('in', e)
exit()
```

其中，用 'out' 调用，返回值可能是出队元素值，也可能是警告 'empty!'；用 'in' 调用，若出现队列满，则在函数内部执行一条警告语句。当然，也可以采用不同的方式，能体现完整的队列管理即可。

函数 queue 的实现要点如下：

(1) 选择存储方案。可以考虑使用一个 n 元一维数组，即采用顺

^① 如果熟悉面向对象程序设计，也可以考虑采用定义一个类的方式实现。

序存储,并用两个代表数组下标的变量(head 和 tail),当队列非空时,两变量分别指向队头和队尾,如图 3.14 所示。此时,出队和入队操作将导致 head 和 tail 的“增量性”变化。如果采用链接存储方案,自然也会有 head 和 tail,但它们的变化是随着数据元素节点的链指针而变,而不是增量性的。

(2) 提高执行效率。需注意,队列在应用中,数据元素的“出队”导致队列长度减 1。如果仿照一般线性表的“删除”操作,就要把后面所有元素都向前移一位,这样做很像日常生活中排队买东西,第一个人买完了离开,后面的往前移,原先第二个人现在变成第一个人。这样做效率很低,因为后面每个人都要动!由于队列的操作只发生在两端的特殊性,我们可以换一个视角:让 head 和 tail 变化,而不是将后面所有元素前移,即在每一个出队操作后做 $\text{head} \leftarrow \text{head} + 1$;在每一个入队操作后做 $\text{tail} \leftarrow \text{tail} + 1$ 。

(3) 队列溢出情况。作为实现队列存储单元的数组,一旦设定了大小 n ,它的有效下标只能是 $0, 1, \dots, n-1$ 。若如上做 head 和 tail 的更新,很容易造成队列的实际长度虽然没有超过设定的存储容量,但已经不允许有新元素入队了。设 $n=4$,操作如下序列:

1, 2, 5, 8, x, x, 9, 6, x, 7

在前 6 个操作完成后,可看到如图 3.15 所示情形。

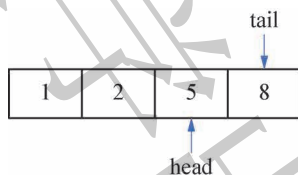


图 3.15 队列操作的影响

此时,“9”要入队,该怎么办?把 5 和 8 向前移两个位置腾出空来?理论上可以,但此方法并不好,低效,且烦琐。

这里反映的正是队列管理的核心问题:如何用一种统一的方式,在队列的实际长度还没有达到设定存储容量的情况下,保证队列操作的高效执行。

(4) 解决溢出问题。图 3.15 所示的数组中,1 和 2 实际上已经没

① 这里简单清晰的表达形式 $(x+1)\%n$,是与 n 个元素按照 $0, 1, 2, \dots, n-1$ 编号,而不是按照 $1, 2, \dots, n$ 编号直接相关的。如果按照 $1, 2, \dots, n$ 编号,要令 1 是 n 的后继,则要写成 $(x-1)\%n+1$,形式过于复杂。这也是许多程序设计语言的数组下标从 0 开始而不是从 1 开始的基本原因,即便于与模运算完美结合。

有用了,可以把 9 放到 1 的位置,后面的 6 则可以放到 2 的位置,接着让 5 出队,head 指向 8,后面的 7 入队则可以放到原来 5 的位置。这就是处理此问题的基本思路,将一维数组看成一个“循环线性表”,除了下标序列 $0, 1, 2, \dots, n-1$ 表达的后继关系外,也令 0 是 $n-1$ 的后继。体现在 Python 中即: $(x+1)\%n$ ^①,表达的是在“以 n 为长度的循环”意义下的“ x 的下一个”,即 x 的后继。在队列管理中,相应操作为当一个元素出队后,做 $\text{head} \leftarrow (\text{head} + 1)\%n$,当一个元素入队后,做 $\text{tail} \leftarrow (\text{tail} + 1)\%n$ 。

(5) 判断队满队空。此时出现一个新的问题: head 和 tail 可能是 $0 \sim n-1$ 之间的任何数,且相对大小关系是不确定的(即有时可能 $\text{head} < \text{tail}$,有时可能 $\text{head} > \text{tail}$),如何判断队列的“空”和“满”? 有多种办法,其中最简单的,是用一个附加变量作为计数器(counter),记录当前队列中元素的个数,若 counter 小于 n 时,才可以进行入队操作,若 counter 大于 0 时,才可以进行出队操作。根据入队和出队操作的成功执行情况,counter 相应做 $+1$ 和 -1 调整。

基于数组实现队列的程序框图如图 3.16 所示。完整的示例程序见附录程序 3.4。

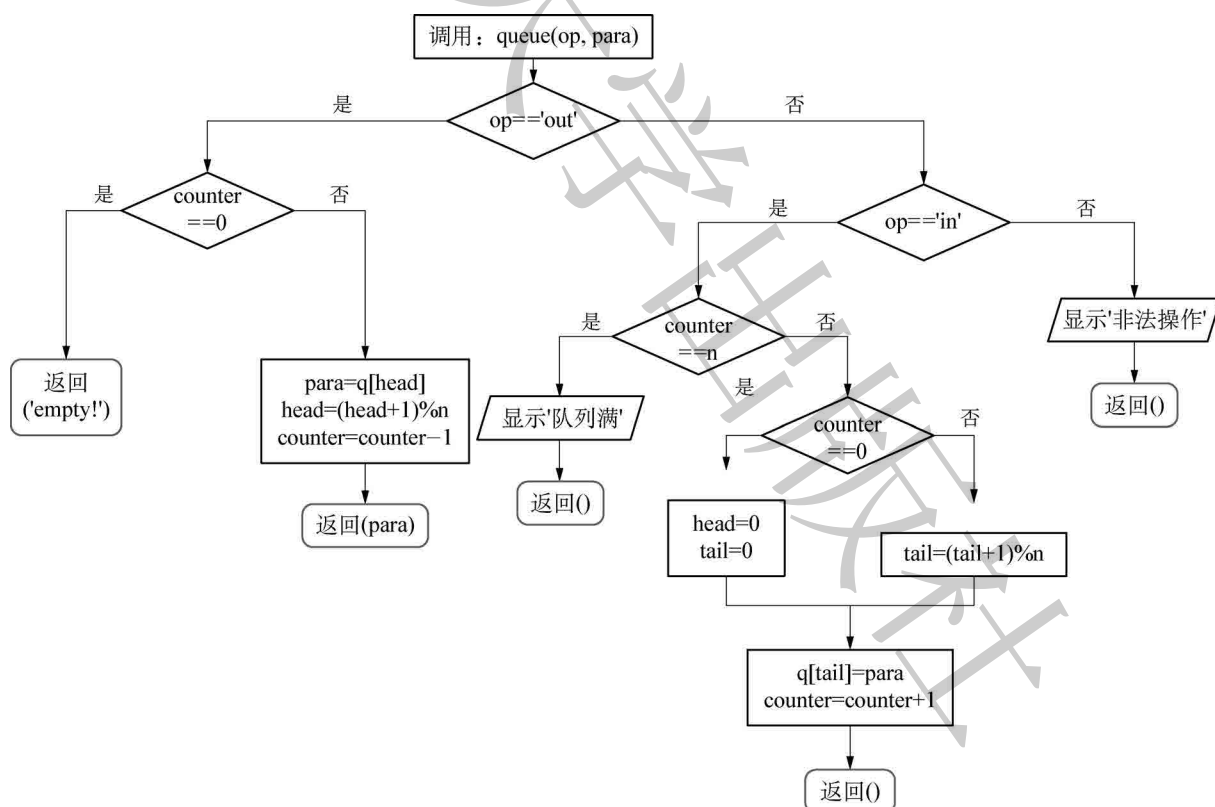


图 3.16 基于数组实现队列的程序框图

其中,所采用的数组为 q , n 是它的大小,即队列的容量;参数 $para$ 在 op 为 'in' 时是输入参数,在 op 为 'out' 时是输出参数。初始化 $counter = 0$,在该条件下若发生 'in' 操作,就将 $head$ 和 $tail$ 初始化为 0 到 $n - 1$ 之间的任何值,表示队列从那里开始(框图中给的是 0)。

探究活动

本章第三节第一点中介绍了队列在计算机系统中三个方面的应用。其中第三个方面所表达的意向在信息社会中是有普遍意义的,即不同程序(甚至可能在不同的计算机中)之间通过队列来协调工作。试讨论队列的一些应用场景。

三、队列的应用

队列除了在计算机系统中有广泛的应用外,也可以用来方便地支持一些具体的问题求解,例 3.4 的编码解密只是其中一个例子。下面讨论本章第一节曾经提到过的“报数选拔问题”如何在队列的支持下方便地解决。

例 3.6 报数选拔问题:这是一个有趣的数学游戏,游戏规则如下:

N 个人围成一个圈,编号从 1 开始,依次到 N ;
从编号为 1 的人开始报数,数到 m 的那个人出列;
他的下一个人又从 1 开始报数,数到 m 的那个人又出列;
以此规律重复下去,直到所有的人都出列。
假设共有 7 人,数到 3 的人出列,问最后留下的是几号。

(1) 设计思路。

这个游戏实质上是一个循环报数过程,可用一个队列的操作来模拟这个游戏过程。

- ① 建立一个 N 个元素的队列。
- ② 为了让数到 m 的人出列,需要设置一个计数器 i , i 的初始值为 0 。
- ③ 计数器 i 累加 1 ,队头元素出队,并判断 i 是否等于 m ,如果 i 不等于 m ,则将此元素添加到队尾;否则,计数器重置为 0 。
- ④ 重复步骤③,直到队列中只剩最后一个数据。

(2) 编写程序。

在程序设计中可以通过数组的方式来建立一个队列。在 Python 中,也可以使用列表来模拟队列,但在出队时,队头删除元素会涉及列表中大量后继元素的移动,效率较低。因此一般采用 Python 自带的标准队列库——queue,通过 import queue 导入后,使用 que = queue() 可以创建一个空队列。Python 队列库提供了以下一些常用的方法: put(x) 为入队; get() 为出队; empty() 判断队列是否空; qsize() 返回队列的长度。一个完整示例程序如下所示。

```
import queue                # 导入队列库
n=7;m=3
que=queue.Queue()          # 构建空队列
for i in range(1,n+1):
    que.put(i)              # 1 到 n 个编号入队
i=0
while que.qsize()>1:       # 队列中数据元素个数大于 1 时
    循环
    i=i+1
    temp = que.get()        # 左边队头数据出队并存储
    到 temp
    if i!=m:
        que.put(temp)      # 将刚出队的数据添加到队尾
    else:
        i=0                # 重新开始
print(que.get())
```

算法复杂度分析:如果有 n 个人,构建队列的 for 循环执行了 n 次,依次出队的 while 循环也执行了 n 次,因为第一个循环结束后,再开始第二个循环,所以时间复杂度为 $O(n)$ 。 n 个元素全部入队,使用队空间大小是 n ,因此空间复杂度也为 $O(n)$ 。

项目实践

送餐时,从厨房到餐桌往往有多条路径。从用时最少角度考虑,机器人要按照最短路径进行送餐,设计方案并编程实现。

作业练习

1 对于任意给定的一对正整数 a 、 b , 假设可以在 a 基础上进行加 1、减 1 或乘 2 的操作。那么, 对 a 最少进行多少次操作后, 能得到 b ? 例如: $a=3$, $b=11$, 可以通过 $3 \times 2 \times 2 - 1$, 3 次操作得到 11。 $a=5$, $b=8$, 可以通过 $(5-1) \times 2$, 2 次操作得到 8。试给出解决这个问题的一般思路, 并编程实现。

2 在本章第三节第二点讨论的队列实现中, 用到计数器变量 (counter) 来指示队列中元素的个数, 并通过对其值的判断, 决定出队和入队操作是否可能。事实上, 也可省去这个变量, 直接通过队列头指针 head 和尾指针 tail 之间的关系来做相应判断。试讨论总结出相关规律, 并尝试编程实现。

3 在本章第一节末, 提到了一种资源管理问题。抽象来讲, 设有 n 个单位的资源, $r_1, r_2, \dots, r_n$, 程序 (资源管理器) 面对一个请求 (req) 和释放 (rel) 序列, 如:

$req_i, rel_j, \dots, req_k, req_l, \dots, rel_m, \dots, rel_n, \dots$

其中 req 表示请求, 可被任一单位的资源满足, rel_i 表示释放, 会指出释放哪一个 (i), 且不一定按照请求发生的顺序释放。于是, 在请求和释放序列的作用下, 我们看到两个队列“当前使用队列”(USE) 和“当前可用队列”(AVL) 的动态变化——“此消彼长”。资源是不连续存储的, 在这种情形, 队列的实现就需要使用链表。试用一个二维数组来表示资源集合 $r_1, r_2, \dots, r_n$, 每行代表一个资源, 其内容包括指向相关队列下一个资源的指针。试讨论实现这样一个资源管理器所涉及的问题, 并尝试编程实现。

第四章

常用数据结构

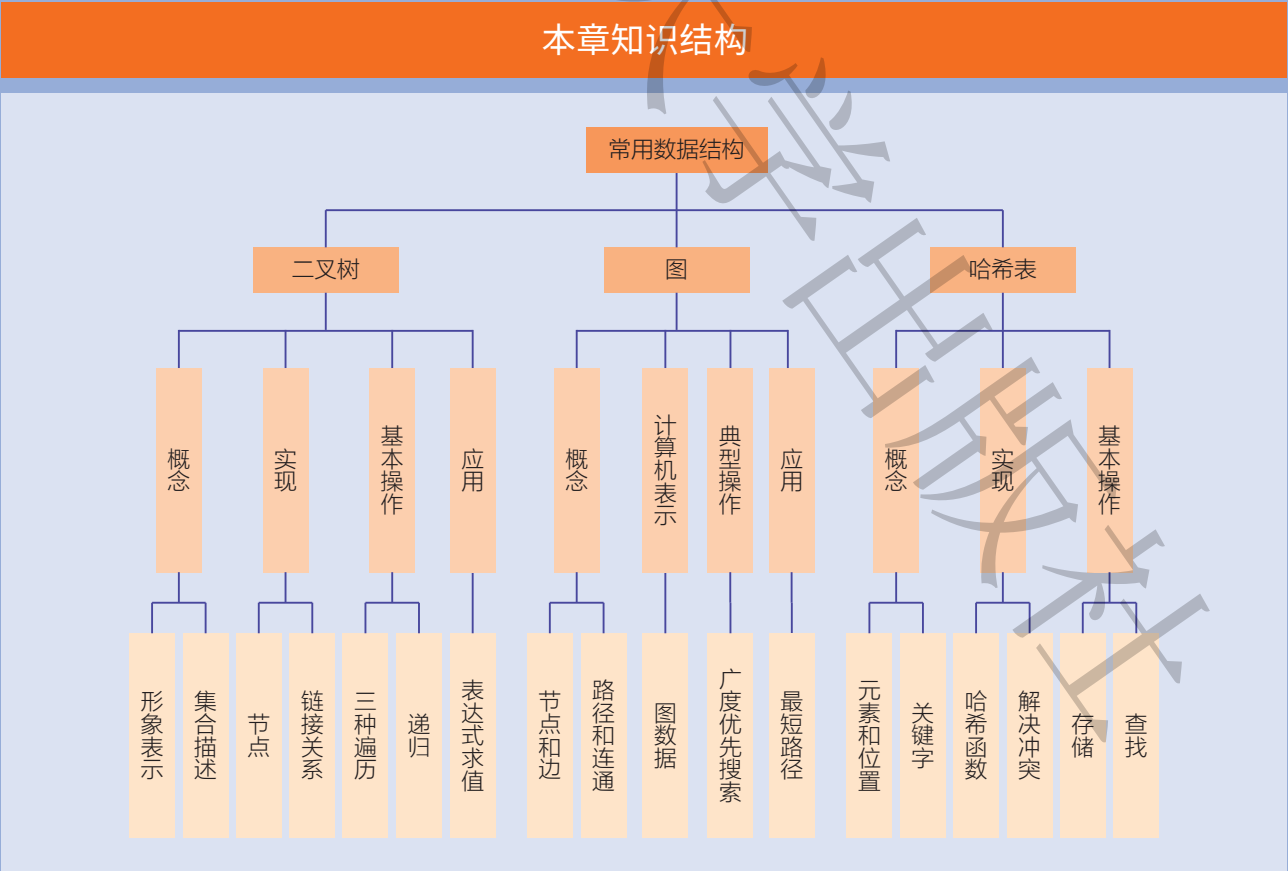
本章学习目标

- 理解二叉树的概念、基本操作和应用场景；掌握一种实现二叉树的方法。
- 初步理解图的基本概念及其在计算机中的表示；掌握一种遍历方法；了解图在表达网络关系信息中的广泛应用。
- 理解哈希表的概念；实现建表和编写基于哈希表的动态查找程序。

二叉树是一种非常重要的常用数据结构,在计算机科学的各个领域都有它的身影。如果说有些数据结构倾向于表达程序输入数据中的自然关系,另一些数据结构倾向于表达算法逻辑需求的关系,那么二叉树属于后者。理解二叉树的概念,掌握二叉树的构建与操作方法及其意义,是从事高水平程序设计工作的一项基本功,也是本模块学习的重点。

图作为一种数据结构,反映的是一个集合中元素之间的二元关系。它最突出的运用是作为各种现实网络的一种抽象,包括互联网、人际关系网、疾病传播网,等等。根据不同的应用需求,图在程序中可有不同的表示。“遍历”是图的基本操作,是许多应用算法的核心。

哈希表是另一种常用数据结构,在大数据时代其作用更为凸显。与二叉树和图等数据结构相比,哈希表的一个突出特点在于其数据元素在结构中的位置是通过“计算”来确定的。



项 · 目 · 情 · 境

邻里小学要搞一个数学心算游戏活动。竞赛采用对抗方式,甲、乙双方各派出一名同学,甲方同学即兴报出一个长度不限,包含若干加、减和乘法的算式(例如 $2+3\times 4-5+1\times 6\times 7-9$),乙方同学仔细听着,待甲方同学报完后要立刻给出答案(例如 42)。然后交替角色。

他们遇到的问题是:在比赛过程中,如何在每一轮立刻得到正确答案,用以评判答方同学给出的结果是否正确?他们思考无果,于是来请你帮忙了。你会如何解决这个问题呢?

项 · 目 · 任 · 务

任务 1	任务 2	任务 3
分析这个问题中的挑战有哪些,例如为什么手机上的计算器不能很好地解决问题。	针对任务 1 中的挑战分析,设计一个用计算机程序解决的方案。	结合本章学习的知识,编程实现你的方案。

第一节 二叉树

二叉树是一种非常重要的数据结构,在计算机科学许多不同领域的算法中都会用到,第一章第二节中已介绍过一个应用实例,后面还会多次用到。二叉树的重要性不仅在于它能够有效地体现许多不同问题的计算过程,还在于其高度通常与其节点个数呈对数关系的特点,成为许多算法得以高效实现的基础。

一、什么是二叉树

在第一章第二节的例子中,我们展示了二叉树在数据流检测应用中的效率高于线性表的情况。从本节开始将详细介绍二叉树相关内容。观察图 4.1,其中有 5 棵二叉树。它们有什么共同特征?

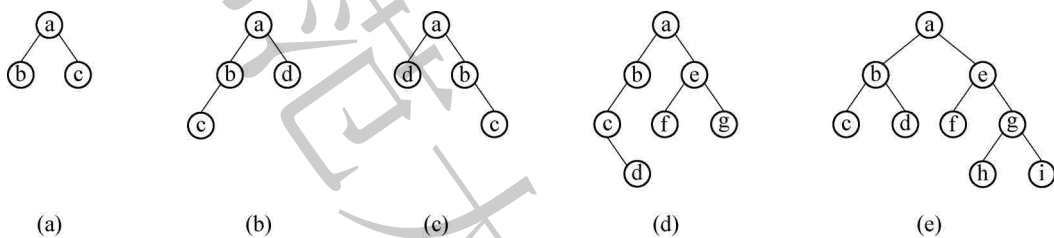


图 4.1 若干二叉树的图示

对比第三章学习过的三种数据结构(线性表、栈和队列)的特征,它们有什么显著不同? 这里看到的结构是“非线性”的,即其中的数据元素(小圆圈表示)之间没有唯一的前导和后继关系。

采用数学中的集合语言,二叉树可有如下严格的定义:

二叉树是一个集合,可以为空;如果不为空,则含有三个元素,“根节点”“左子树”和“右子树”。左、右子树也是二叉树。

上述定义的最后一句话值得特别注意,它表明这是一个递归的定义。按照以上定义并采用定义中的顺序^①,图 4.1(a)中的二叉树可表示为:

$$\{a, \{b, \{\}, \{\}\}, \{c, \{\}, \{\}\}\}$$

而图 4.1(b)中的二叉树可表示为:

$$\{a, \{b, \{c, \{\}, \{\}\}, \{\}\}, \{d, \{\}, \{\}\}\}$$

^① 其中三个元素的顺序也可以有两种变化。记当前的顺序为{根节点,左子树,右子树},那么也可以表示为{左子树,根节点,右子树}或{左子树,右子树,根节点}。而一旦规定表示顺序,其中所有嵌套的子树都必须采用同样的顺序表示。这三种顺序有专门的名字,即“先(根)序”“中(根)序”和“后(根)序”。

按照集合定义,图 4.1(c)(d)(e)的二叉树集合应该如何表示?

这种表示方法看起来相当烦琐,但计算机处理起来却十分轻松。^①据此观察图 4.1 中二叉树的图示可知,根节点在上,左子树在左下,右子树在右下,用“边”(edge)示意关联。由定义知,一个节点(node)可以有 0 个、1 个或者 2 个非空子树。子树的根节点通常称为其上一级的“子节点”,在需要区分左右时,可称为“左子节点”和“右子节点”;反过来其上一级称为它的“父节点”。

另外,在二叉树中,通常把没有子节点的节点称为“叶子节点”。从根节点到树中某节点所经路径上的边数称为该节点的层次,根节点为第 0 层,依次往下递增。所有节点中层次最大的值称为树的高度。二叉树在应用中有一些重要的特例:满(full)二叉树(如图 4.2(a)),完全(complete)二叉树(如图 4.2(b)),完美(perfect)二叉树(如图 4.2(c))。

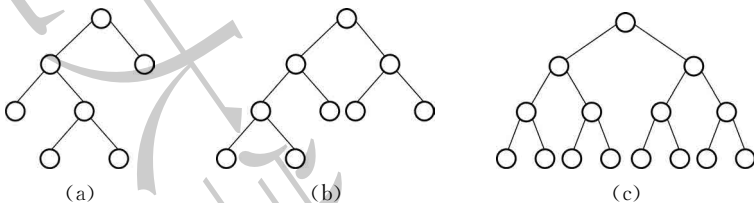


图 4.2 各种类型的二叉树

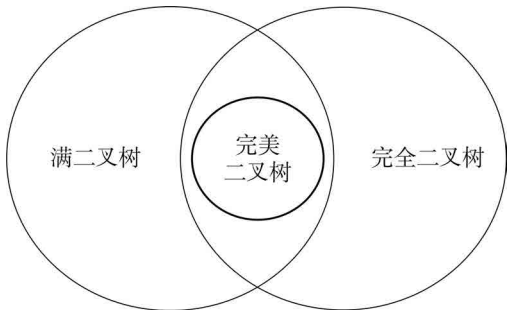


图 4.3 三种类型二叉树的关系图

完美二叉树的特点是,对于每一个层次,第 i 层上的节点数为 2^i 。满二叉树的特点是,一个节点若有子节点,则一定是左、右子节点都有。完全二叉树则看成是完美二叉树删去了最下层右边若干节点的结果。满二叉树、完全二叉树、完美二叉树之间的关系可以用图 4.3 表示。

关于二叉树的基本概念,还有以下内容需要关注。首先,我们注意到上文中二叉树的定义是“递归的”,即在定义中也包含了二叉树自身。但通过“可以为空”以及“不空”则有“根节点”等的表述,避免了简单的“同义反复”,体现了一种严谨的逻辑。这是一种计算机科学中常见的表达方式,不仅在定义中常见,在编程中也有广泛应用,即“递归调用”——一个函数内部

^① 请结合栈的应用一节中刚学过的括号匹配,思考计算机的处理方法。

包含对该函数自身的调用。当然,对应于定义中特别指出有空集合,递归调用的函数中也需要有对终止条件的处理,否则调用就真的“没完没了”了。后面我们会看到一个具体的例子。

探究活动

本节中已简略定义过完美二叉树,这里做进一步讨论。如图 4.4 所示。你能看出它们为何完美吗?现通过完美二叉树介绍与二叉树相关的另一些术语,进一步体会二叉树的内涵。首先,从根节点到某节点 x 的“距离”,指的是从根节点开始沿着边往下,到达节点 x 需要经过的边的条数。一棵二叉树的“高度”,就定义为树中最远节点的距离。这样,图 4.4 a 中二叉树的高度为 2 图 4.4 b 中二叉树的高度为 3 图 4.4 c 中二叉树的高度为 4。有时会说某个节点位于二叉树的第几“层”,层数就是它与根节点的距离。根节点在第 0 层。

由上可知,在任何二叉树中,第 i 层最多有 2^i 个节点。完美二叉树就是每一层恰好有 2^i 个节点的二叉树。试想一下:高度为 h 的完美二叉树共有多少节点?

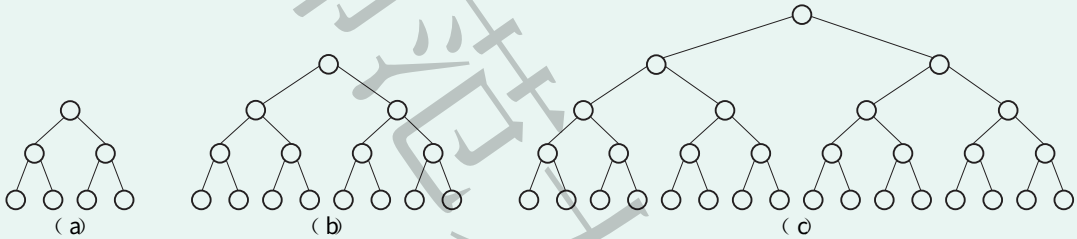


图 4.4 完美二叉树

二叉树特别强调区分子树的左右,我们可以对二叉树的节点按照从上到下、从左到右的顺序做一个自然的顺序编号(相当于一种遍历)。而节点也可以根据层序编号,被称为第 i 层的第 j 个节点(统一起见, j 从 0 开始)。这样,对于高度为 2 的完美二叉树,就有如下顺序编号和层序编号的对照表:

顺序	0	1	2	3	4	5	6	...
层序	(0, 0)	(1, 0)	(1, 1)	(2, 0)	(2, 1)	(2, 2)	(2, 3)	...

请尝试建立顺序编号和层序编号之间的关系。
给定一个顺序号 k 如何得到层序号 (i, j) ? 反过来呢?

二、二叉树的实现

对于图 4.1(d)所示的二叉树,它的集合定义表示为:

$\{a, \{b, \{c, \{\}, \{d, \{\}, \{\}\}\}, \{\}, \{e, \{f, \{\}, \{\}\}, \{g, \{\}, \{\}\}\}\}$

在程序中该如何体现呢? 最常见的做法是在程序中实现若干个含有

一个数据元素和两个链指针的“节点”，让它们的指向关系符合所需表达的树的定义。

下面尝试对于任何符合二叉树定义的集合(例如上式)，给出其程序实现。

	数据	左链	右链
0	a	1	4
1	b	2	-1
2	c	-1	3
3	d	-1	-1
4	e	5	6
5	f	-1	-1
6	g	-1	-1

(a)

	数据	左链	右链
0	b	3	-1
1	e	6	5
2	a	0	1
3	c	-1	4
4	d	-1	-1
5	g	-1	-1
6	f	-1	-1

(b)

图 4.5 二叉树的程序内部表示例

与第三章第一节中关于线性表的链接实现类似，也可以采用 Python 类或数组方式来实现二叉树。我们将此类实现方式放到后面和递归问题一起讨论，这里先看采用数组的方法。类似于线性表实现的链表方法，用二维数组的行表示节点，此时数组要有三列，第 1 列用于存放数据元素，第 2 列用于存放指向左子树的指针，第 3 列用于存放指向右子树的指针。这样，图 4.5(a)和(b)所示的数组，均为图 4.1(d)的一种程序内部表示，表中的“-1”表示对应的子树为空，类似于链表中的最后一个元素“⊥”。

根据上面的集合表达式 $\{a, \{b, \{c, \{\}, \{d, \{\}, \{\}\}\}, \{\}, \{e, \{f, \{\}, \{\}\}, \{g, \{\}, \{\}\}\}\}$ ，可知 a 为根节点，进而看出图 4.5(a)和图 4.5(b)的区别。图 4.5(a)中，根节点在第 0 行；图 4.5(b)中，根节点在第 2 行。一般来说，在二叉树的应用中，需要一个变量指向根节点所在的位置。

体 验 思 考

对图 4.1(d)所示的二叉树，给出两种不同于图 4.5(a)和(b)所示的表示，对应数据元素在数组中的序分别为 a, c, f, b, g, d, e 和 f, g, c, d, e, b, a。

例 4.1 根据集合生成二叉树：给定集合表达式 $\{a, \{b, \{c, \{\}, \{d, \{\}, \{\}\}\}, \{\}, \{e, \{f, \{\}, \{\}\}, \{g, \{\}, \{\}\}\}\}$ ，设计程序自动得到形如图 4.5 中的数组。

要生成二叉树，需确定节点和左、右子树。首先，分析一下这个集合，该集合由左括号、字母、逗号、右括号组成；进一步找寻规律，发现只要遇到“{”就表示一个树的开始，遇到“}”则表示一个树的结束。因此可以通过依次读取集合表达式的字符来建立树，再用一个栈来保存过程中生成的节点，当读取到“{”时，用“{”后面的字母生成节点并入栈，当读取到“}”时，出栈一个数据元素，此时的栈顶即为出栈元素的父节点，把出栈元素的地址填写到父节点的链域，也就是数组的第 2 列或第 3 列。如果第 2 列为 0，就填入第 2 列，否则就填入第 3 列。但是对于“{”之后紧接着“}”这种情况，不用生成新的节点，只需给父节点的对应

链域赋值 -1, 表示该父节点对应链域后面无节点。程序流程如下。

(1) 产生 100 行 3 列的二维数组。

(2) 从左到右依次读取表达式。

(3) 当读取到“{”时, 若后一个字符不是“}”而是一个字母, 则用字母生成一个节点并进栈; 否则, 将 -1 赋值给栈顶节点的对应链域, 并读入下一个字符。

(4) 当遇到“}”时, 如果前一个字符不是“{”, 则出栈, 并将出栈元素地址赋值给栈顶元素的对应链域(即每当出栈时, 出栈元素一定为栈顶元素的子节点, 应将其挂到栈顶元素的子节点位置上), 这样以栈顶元素为根的树(子树)构造完毕。否则, 读入下一个字符。

程序流程图如图 4.6 所示。

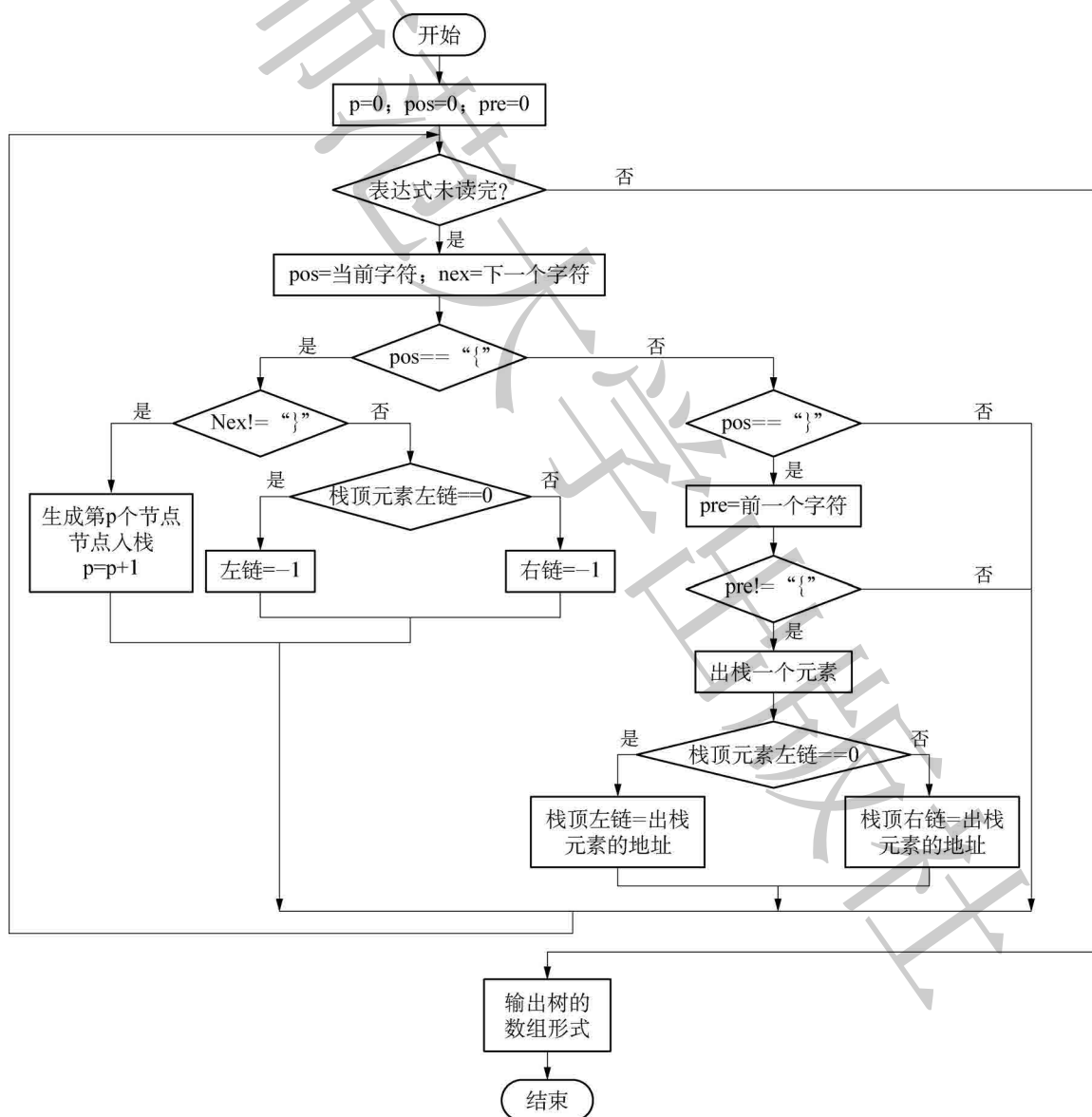


图 4.6 根据集合建立二叉树流程图

完整参考程序见附录程序 4.1。

上述过程根据二叉树的定义将其存储在计算机内部。虽然以数组形式存放,但我们能从数组元素之间的关系中体会到二叉树的逻辑结构。

三、二叉树、遍历与递归

二叉树得到广泛应用的原因之一是它与许多算法的逻辑完美匹配。针对二叉树的操作,除了根据算法的需求在适当的地方做节点的添加、插入和删除外,还有一类具有标志性的操作——二叉树的遍历,即按照一定的顺序访问二叉树的所有节点,使得每个节点被访问一次,而且只访问一次,等价于将二叉树的节点做线性排列。遍历二叉树是许多实际应用得以实现的基础,后文即将介绍的表达式求值就是一个典型例子。

给定一棵二叉树,根据对其根节点的访问次序,有三种主要的遍历方式,如表 4.1 所示。

表 4.1 二叉树的主要遍历方式

名称	访问顺序规则
先根序 (简称先序)	先访问根节点,然后访问左子树,最后访问右子树;递归遍历
中根序 (简称中序)	先访问左子树,然后访问根节点,最后访问右子树;递归遍历
后根序 (简称后序)	先访问左子树,然后访问右子树,最后访问根节点;递归遍历

例如,图 4.7(a)是表示一个算术表达式的二叉树,三种方式的遍历结果如图 4.7(b)中所示。^①

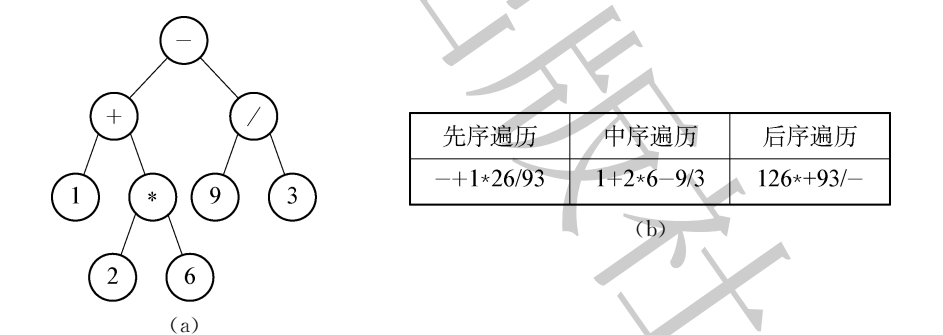


图 4.7 一棵表达式二叉树

① 观察三种遍历结果,哪一种比较“自然”?

回顾前面关于二叉树的集合定义,可发现表 4.1 的内容与二叉树集合定义中三个元素的次序有直接对应关系。现在,我们要写出一个程序,对于任意如图 4.5 中数组表示的二叉树,得到它的遍历结果。这里只具体讨论先序遍历,另外两种遍历可作拓展练习。

先序遍历从根开始,按照访问顺序规则顺着一条路径尽可能向前探索,直到检查完叶子节点后,返回上一个节点,继续沿另一路径向前探索,如此反复,直到所有节点都访问到。这是一个顺着路径不断探索、不断回头的探索过程,与栈基本操作的特点非常吻合,所以可通过入栈、出栈的方法实现。按照先序方式分析入栈出栈过程如下。

- (1) 先创建一个空栈,令根节点入栈。
- (2) 当栈不空时,重复下面操作,否则结束。
- (3) 出栈一个元素赋值给 temp,并显示该元素数据。
- (4) 如果 temp 右链不等于 -1,则右链对应的节点入栈。
- (5) 如果 temp 左链不等于 -1,则左链对应的节点入栈。

基于以上表述,程序流程图如图 4.8 所示。

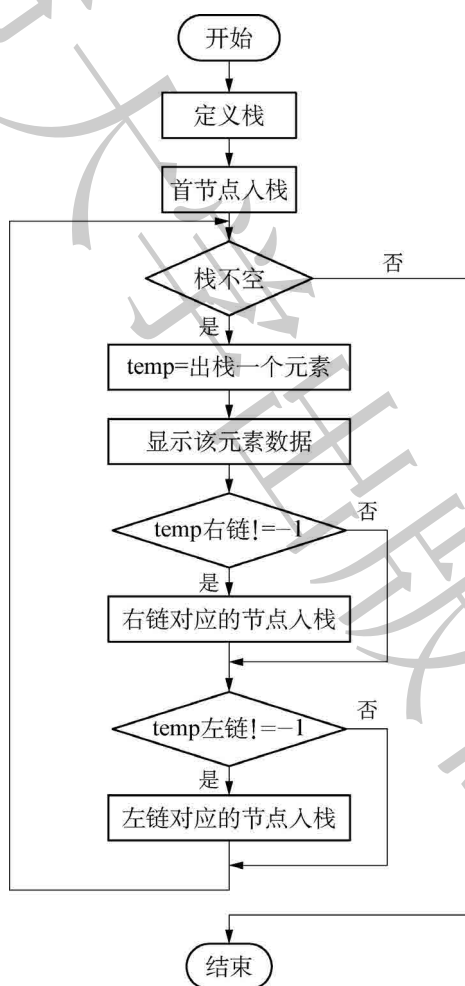


图 4.8 先序遍历流程图

程序如下所示。

```
stack1 = list() # 建立一个栈,存放各节点
stack1.append(memory[0]) # memory 及 change 定义见附录程序 4.1
while stack1:
    temp = stack1.pop() # 出栈
    print(change[temp[0]], end=" ")
    if temp[2] != -1:
        stack1.append(memory[temp[2]]) # 右子树入栈
    if temp[1] != -1:
        stack1.append(memory[temp[1]]) # 左子树入栈
```

下面我们回到表 4.1,重新审视三种遍历规则,会发现描述中都有“递归遍历”字样。递归是计算机科学中的一个基本且重要的概念,其含义我们在本教材第一章第二节的例子中已有所触及,在本节前面关于二叉树的定义描述部分中也有所体现。递归,通俗讲就是将规模较大的事物不断用规模较小但性质相同的事物表达,而当规模足够小的时候,情况变得简单明了。一般来说,我们可以谈论定义的递归和过程的递归。而一个递归定义的概念常常也容易用递归的算法予以实现。二叉树及其遍历就是一类经典的例子。

接下来,我们将通过定义二叉树类,让它包含树的创建和三种遍历方法,展示递归的威力。我们将看到,与前面用数组模拟创建二叉树的方式相比,采用递归调用,在一个递归定义二叉树的集合上构建出一棵程序可用的二叉树,程序代码会多么简洁。

下面是二叉树类定义的一部分。类似于第三章链表类的定义,首先有一个节点类的定义。此节点类定义较简单,仅含三个数据域,适应树节点有两个链接的要求。而二叉树类中只有一个方法 create,它基于前面介绍的二叉树的集合定义(实际使用 Python 的元组以保证顺序的要求),生成一棵程序可用的二叉树。我们可以清楚地看到,create 方法中也调用了 create,但树的规模变小了,这就是递归。整个程序结构与集合定义结构完全对应。

```
class TreeNode:
    def __init__(self, data, left = None, right = None):
        self.data = data
        self.left = left
```



```
self.right = right
```

```
class BinaryTree:
```

```
    def __init__(self):
```

```
        self.head = None
```

```
    def create(self, triple):
```

```
        root = None
```

```
        if triple: # not empty
```

```
            root = TreeNode(triple[0], None, None)
```

```
            if self.head == None: # 将要返回的整棵树的根
```

```
                self.head = root
```

```
            print(root.data, end = ' ')
```

```
            root.left = self.create(triple[1])
```

```
            root.right = self.create(triple[2])
```

```
        return root
```

也就是说,有了这个类定义,下面程序段的执行就完成了对应于前面图 4.1(d)(为方便起见,这里用数字替换了字母)二叉树的构建。

```
t = BinaryTree() # 实例化一个二叉树变量 t, 初始为空
tree = (1, (2, (3, ()), (4, ())), ()), (5, (6, ())), (7, ()), ()))
t.create(tree) # 基于 tree 的集合定义, 程序建树, 返回树根指针
```

怎么知道这棵树构造是否正确呢? 遍历就是一种检验的方法。根据表 4.1 中的规则,我们能够得知图 4.1(d)所示的三种遍历结果分别如下。

先根序: 1, 2, 3, 4, 5, 6, 7;

中根序: 3, 4, 2, 1, 6, 5, 7;

后根序: 4, 3, 2, 6, 7, 5, 1。

下面给出了中根序和后根序遍历方法的实现,是不是也观察到了递归调用呢?

```
t.midroot_traversal(t.head)    # 中根序遍历
print()
t.postroot_traversal(t.head)   # 后根序遍历
print()
```

将它们结合到前面类定义的代码中。在创建了树之后,如下代码将给出遍历的结果。

```
def midroot_traversal(self, root):
    if root == None:
        return
    else:
        self.midroot_traversal(root.left)
        print(root.data, end=' ')
        self.midroot_traversal(root.right)
    return

def postroot_traversal(self, root):
    if root == None:
        return
    else:
        self.postroot_traversal(root.left)
        self.postroot_traversal(root.right)
        print(root.data, end=' ')
    return
```

探 究 活 动

将上述零散的代码整理成适当的 Python 程序文件,并补充一个先根序遍历方法 `preroot_traversal(root)` 让 `t.preroot_traversal(t.head)` 语句的执行也能得到正确结果。

四、二叉树的应用

二叉树在计算机科学中有许多应用。第一章中提到的数据流检

测是信息查找应用的一个例子。关于查找,我们在第五章还会专门学习。这里介绍另一个典型应用:算术表达式的求值。

一个算术表达式通常由若干个具有不同优先级的运算符和运算数组成,这种形式用二叉树能很好地表示出来,可以将运算符放在内部节点位置,把运算数放在叶子节点位置,通过不同的节点层次,表达运算符的优先顺序,按照这种思想构成的二叉树,也称为“表达式二叉树”。

例 4.2 表达式二叉树:编写一个程序,输入一个任意长度但只含四则算术运算(+, -, *, /)和一位正整数(不包含括号)的算术表达式,首先按照先乘除后加减的原则构造表达式二叉树,然后由二叉树计算该表达式的值。如表达式“ $1 + 2 * 6 - 9 / 3$ ”,生成后的二叉树如下图所示,计算出的值为 10。

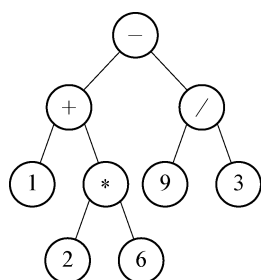


图 4.9 表达式二叉树

从图 4.9 可知,表达式二叉树的特点:运算数都是叶子节点;运算符都是内部节点;优先运算的运算符都在树下方,而相对优先级较低的减法则最后运算。因此可以换一个角度来思考,采用从叶子往根的顺序进行树的构造。归结对任意一个表达式构造树的算法如下。

(1) 读取第一个运算数先作为表达式树的根。

(2) 读取第二个运算符插入并变为根节点,原根节点变为新节点的左子节点。

(3) 读取运算数时,生成一个节点,并沿着根节点右链插入最右端。

(4) 读取运算符时,生成一个节点,之后跟根节点运算符进行对比,分两种情况进行处理:

当优先级不高时,新节点作为根节点,原表达式树变为新节点的左子树(如下文表达式“ $1 + 2 * 6 - 9 / 3$ ”二叉树构造过程中的步骤(6))。

当优先级较高时,新节点作为根节点右儿子,原根节点右子树变为新节点的左子树。(如下文表达式“ $1 + 2 * 6 - 9 - 3$ ”二叉树构造过程中的步骤(4))。

(5) 重复第(3)步和第(4)步,直到表达式全部读取完。

据此解析对表达式“ $1 + 2 * 6 - 9 / 3$ ”构造二叉树的过程。

(1) 首先读取表达式的第一个字符“1”,由于表达式树目前还是一棵空树,所以 1 作为根节点,如图 4.10(a)所示。

(2) 读取第二个字符“+”,此时表达式树根节点为数字,需要将新节点作为根节点,原根节点作为新根节点的左子节点。如图 4.10

(b)所示。

(3) 读取第三个字符“2”，判断为数字，则沿着根节点右链插入最右端，如图 4. 10(c)所示。

(4) 读取第四个字符“*”，判断是运算符，则与根节点“+”比较优先级，“*”的优先级高则插入作为“+”节点的右子节点，而“+”节点原来的右子节点则变为“*”节点的左子树，如图 4. 10(d)所示。

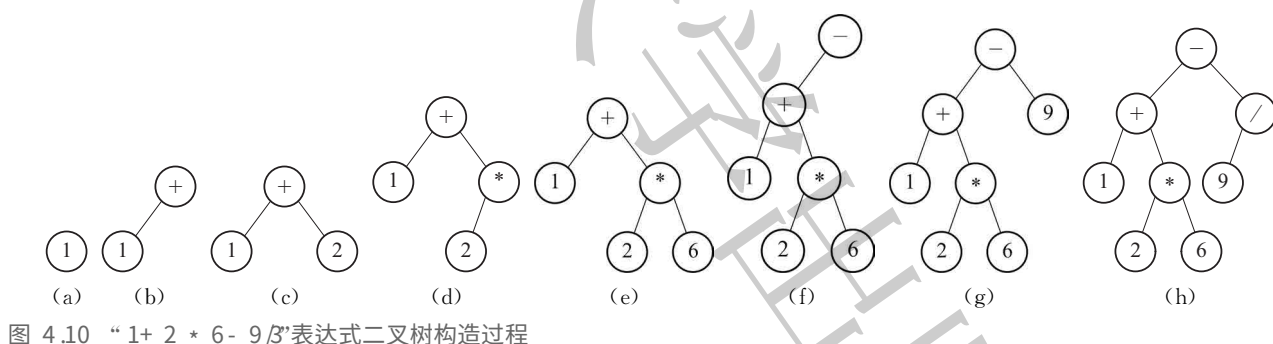
(5) 读取第五个字符“6”，判断为数字，则沿着根节点右链插入最右端，如图 4. 10(e)所示。

(6) 读取第六个字符“-”，“-”与根节点“+”比较优先级，优先级相等，则“-”节点作为根节点，原表达式树则变为“-”节点的左子树，如图 4. 10(f)所示。

(7) 读取第七个字符“9”，判断为数字，则沿着根节点右链插入最右端，如图 4. 10(g)所示。

(8) 读取第八个字符“/”，与根节点“-”比较运算符的优先级，优先级高则作为根节点“-”的右儿子，原根节点右子树“9”则变为新节点的左子树，如图 4. 10(h)所示。

(9) 读取第九个字符“3”，判断为数字，则沿着根节点右链插入最右端，如图 4. 9 所示。至此，运算表达式已全部遍历，一棵表达式二叉树就建立完成了。



下面分析通过表达式二叉树计算表达式值的过程：观察图 4. 9，从上往下看，要得到根节点“-”号的结果必须先计算出左子树“+”号和右子树“/”号的结果。而要想得到“+”号的结果，又必须先计算其右子树“*”号的结果。“*”号左、右子节点是数字，可以直接计算， $2 \times 6 = 12$ 。接下来计算“+”号的结果， $1 + 12 = 13$ ，即根节点的左子树结果为 13。“/”号左、右子节点是数字，可以直接计算， $9 / 3 = 3$ 。于是，根节点的右子树结果为 3。最后计算根节点“-”号的结果， $13 - 3 = 10$ ，得出该表达式的值为 10。这个过程其实是对二叉树进行后序遍历，并在遍历过程中计算出表达式的值。

除了用数组的方式之外,还可以通过创建类的方式来描述树。在类中包含一个数据域和两个地址域,一个地址域表示左子节点,另一个地址域表示右子节点,这样就可以用类来表示节点和下层节点之间的关系。在 Python 中常用定义类的方式来实现一个节点。节点定义代码如下。

```
class TreeNode:
    def __init__(self, data, left = None, right = None):
        self.data = data
        self.left = left
        self.right = right
```

程序流程图如图 4.11 所示,参考程序见附录程序 4.2。

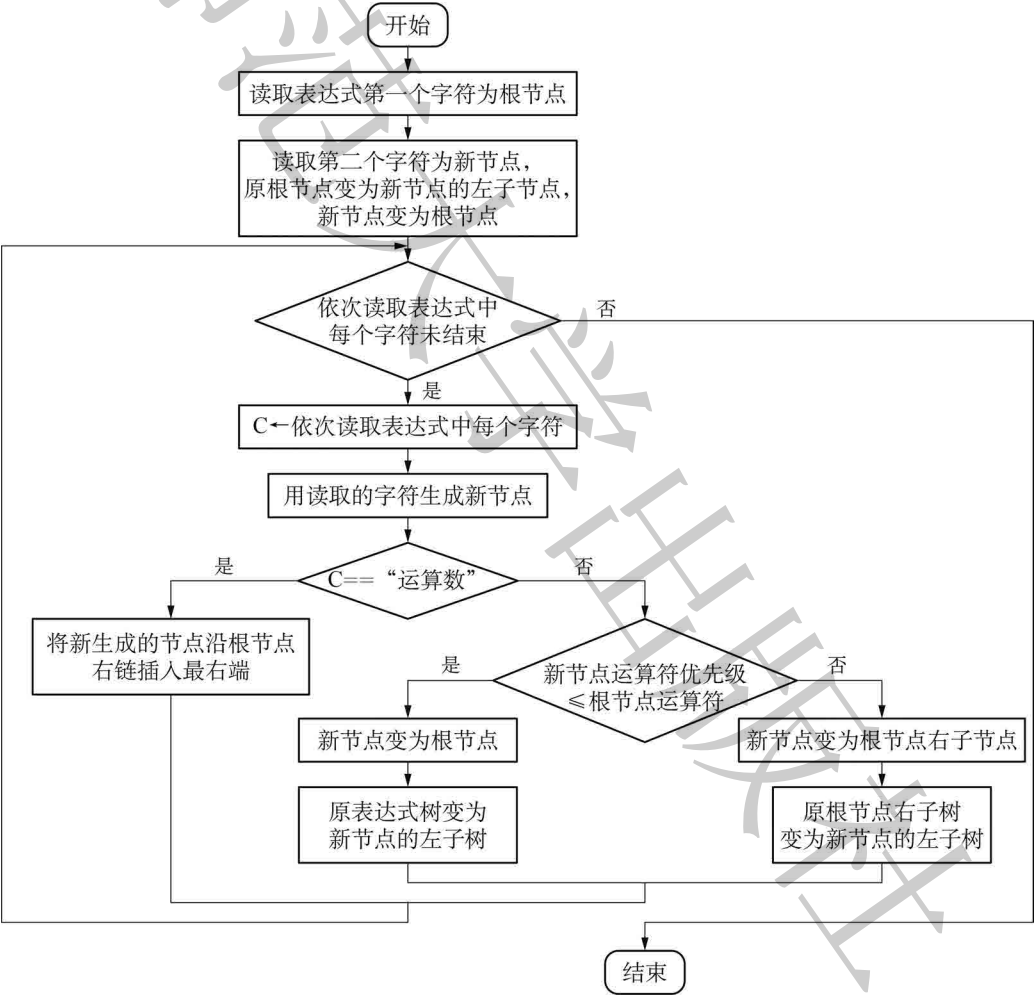


图 4.11 构造算术表达式二叉树的流程图

项目实践

按照本章项目要求,分组讨论实现方案,针对所给出的三项任务,完成项目报告。在项目中可依次读取文件中的表达式,然后采用二叉树的结构来存储表达式,通过树的合并操作完成表达式计算,从而得出总价,请参考例 4.2 编程实现。

作业练习

- 1 本章第一节中提到了关于二叉树种类或特征性质的一些术语概念,如满二叉树、完全二叉树、高度、叶子节点等,试做归纳总结。
- 2 针对一个二叉树具体例子,做三种遍历。
- 3 写出二叉树的中序和后序遍历程序。
- 4 完善表达式求值程序(允许多位整数,允许括号和负数,能判断除数为 0 的错误)。

第二节 图*

图作为一种重要的数据结构,是现实世界中许多事物(例如社交网络、交通网络等)的一种抽象模型,本章第一节中的二叉树也是一种特殊的图结构。图的一些基础算法常常有很广泛的应用,本节重点讨论的广度优先搜索算法就是其中之一。

一、什么是图

“图”对应的英文单词是“graph”,是数学和计算机科学中的一个概念^①,是现实生活中许多事物和状态的一种抽象,因其应用的普遍性产生了一个专门的数学分支——图论。例如,分析一个班级同学的姓名中出现相同字的情况。一种分析的角度是,若某两个同学的姓名中有相同字,就认为他们两人有关系。在此取某一个班级中七名同学,将他们的姓名写成如下集合:

$V = \{\text{张晓光, 李鹏程, 张晨, 王晓蓓, 谢李晨曦, 林宇聪, 周晓宇}\}$

按照上述关系的含义,两两之间的关系用集合表示为:

$E = \{(\text{张晓光, 张晨}), (\text{张晓光, 王晓蓓}), (\text{张晓光, 周晓宇}), (\text{张晨, 谢李晨曦}), (\text{李鹏程, 谢李晨曦}), (\text{王晓蓓, 周晓宇}), (\text{林宇聪, 周晓宇})\}$

在图论的意义上,以上集合定义了一个图,不妨称为“姓名关联图”,其中同学姓名的集合 V 称为“节点”(或“顶点”)集(vertex, node),关系集合 E 称为“边”集(edge, arc)。为了帮助理解,可将这样的图“画”出来,如图 4.12(a)所示,这样的图能形象地表示出“节点”和“边”。

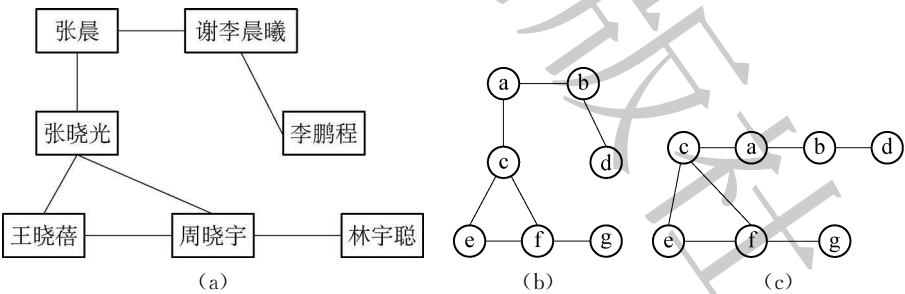


图 4.12 图的示例

^① 不同于日常所说的“图”。日常所说的“图”具有较强的画面感,对应的英文单词是“picture”。

仿照图 4.12 a), 给出现有班级同学的“姓名关联图”。在这个过程中, 大家也许会发现, 你所得到的图与图 4.12 a) 所示有一个重要的不同, 即有些节点之间不存在由边构成的路径。这样的图称为“非连通图”, 而图 4.12 a) 所示的图形称为“连通图”。

有时候, 我们只关心图的结构, 而不关心节点的具体含义, 于是可根据图 4.12(a) 画出图 4.12(b), 其中的字母(或数字)是为了讨论方便赋予节点的代号(或称标签, label)。图 4.12(b) 这种抽象的表达, 不必在形貌上和图 4.12(a) 保持一致, 也可以画成图 4.12(c) 的样子, 保持节点之间的关系正确即可。

现实世界中的许多事物和现象都可以抽象为图。例如, 以城市作为节点, 以两个城市之间是否有直航作为关系, 就可以定义出一个直航关系图; 以国家为节点, 以两个国家之间是否有贸易往来为边, 就可以定义出一个国家贸易关系图; 而按照人们的通讯录名单, 自然能定义出一个社交网络图……

一般情况下, 当讨论图的相关问题时, 人们习惯说“图 $G(V, E)$ ”, 其中 V 表示节点集合, E 表示边集。不过字母的采用并没有强制性, 也可以说“图 $D(X, L)$ ”, 括号中的第一个字母表示节点集, 第二个表示边集。

一个图, 除了其节点集(以及节点个数)和边集(以及边的条数)外, 根据不同的背景应用需求, 人们还常讨论它的其他结构性特征。例如在研究表示社交网络的图时, 就关心节点的度数, 即与节点相关联的边的数目, 以图 4.12(b) 为例, 其节点度数对照关系如表 4.2 所示。

表 4.2 节点度数示例

节点	a	b	c	d	e	f	g
度数	2	2	3	1	2	3	1

对于现实社交网络, “度数”常常意味着“人气”。在研究表示航空直航关系的图时, 往往关心两个节点之间的最短路径, 以图 4.12(b) 为例, a 和 g 之间存在路径 $a-c-e-f-g$, 长度为 4, 仔细观察发现, 此

路径长度不是最短的。它们之间的最短路径是 $a - c - f - g$ ，其长度为 3。两个节点间最短路径的长度也称为它们在图中的距离。^①可以想到，在现实航空直航关系图中，两个城市节点之间的距离意味着从一个城市飞到另一个城市需要搭乘的最少航班数。

探究活动

列举现实生活中可以用图来表示的事物和现象。讨论梳理图的相关概念：路径，距离，连通。

二、图的实现

为了满足应用的需求，确定某两个节点之间最短路径的长度，当图的规模稍微大一点时，仅凭目测就很困难了，而计算机能够很高效地解决这个问题，前提是我们要把所关心的图“告诉”它，如同上一节学习二叉树，要把二叉树的定义转变为计算机内部的表示一样。由上一节可知，图是由两个集合定义的，它可以有多种计算机表示方法，这里介绍最直接的一种：用一个一维数组表示节点集合，用一个二维数组表示边集合^②，其中每一行表示一条边，两列分别对应边的两个端点。这样，图 4.12(b) 中的图就可以表示为如图 4.13(a) 所示形式：

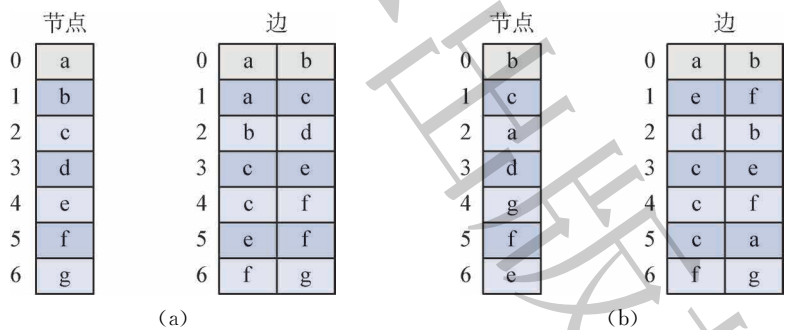


图 4.13 图的两计算机表示方法

① 注意，这里的“距离”是两个节点之间最短路径上边的条数，完全是一个结构意义上的抽象概念。在实际应用中，有时候会给图的边赋予一个权重，例如直航线的实际距离，在那种情况下谈距离常常就会把边上的权重加起来。

② 在 Python 中则可以方便地用两个列表表示，一个列表的元素为节点，另一个列表表示边，其元素由两个节点构成。

图 4.13(a)中,节点个数和边的条数正好相等,都是 7。一般来说,它们不一定是相等的。同时,在这两个数组中,元素的顺序并不重要;在表示边的数组中,每一条边两个端点的先后(即在哪一列)也不重要。于是,图 4.13(b)也可以看作同一个图的计算机表示。这种表示的结果(以及其他图表示方法的结果),在当今大数据时代被称为“图数据”。

基于图数据,我们能做许多有意义的计算和分析。先从一个简单的(也是基本的)计算入手:计算每一个节点的度数,即编写程序使其能由如图 4.13(a)或(b)所示数据产生表 4.2 所示的结果。

体验思考

根据一个图的定义,给出它的计算机表示。

如果不特别考虑效率,程序设计可以很简单:依次以每个节点为观察点,看边集中有哪些元素涉及该节点,凡涉及,即令相应节点度数加一。程序示例如下。

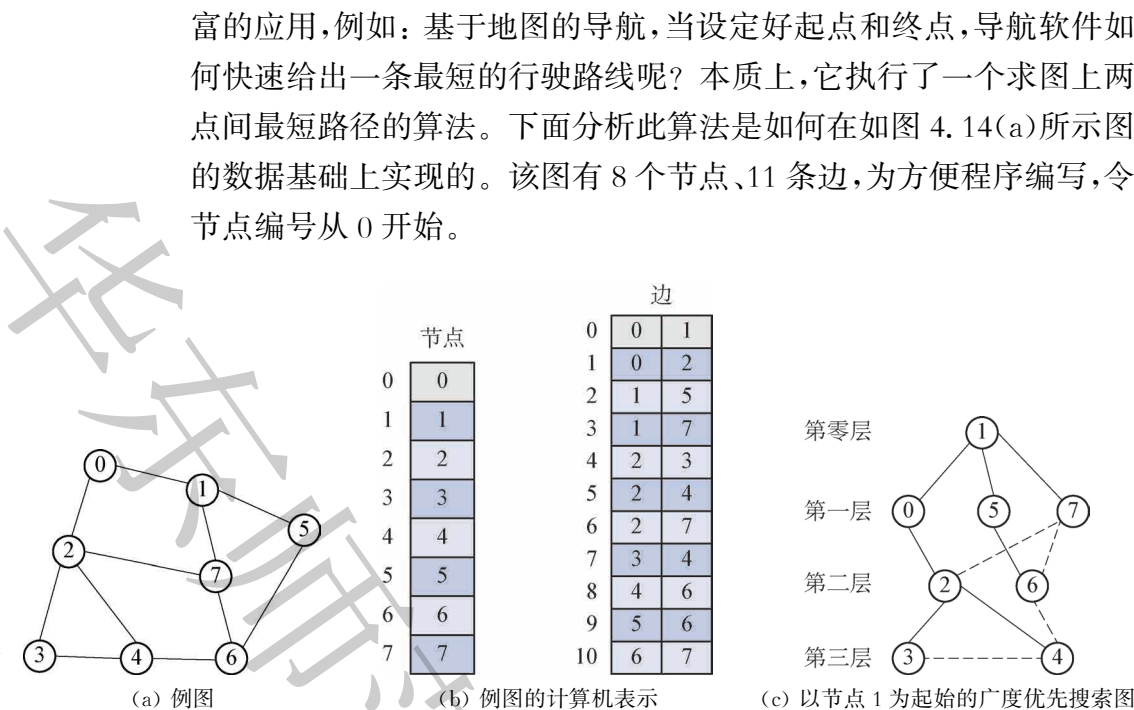
```
node = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
edge = [['a', 'b'], ['a', 'c'], ['b', 'd'], ['c', 'e'], ['c', 'f'], ['e', 'f'], ['f', 'g']]
n = len(node)
e = len(edge)
degree = list(0 for i in range(n))
for i in range(n):
    for j in range(e):
        if edge[j][0] == node[i] or edge[j][1] == node[i]:
            degree[i] = degree[i] + 1
print(node)
print(degree)
```

其中稍值得注意的是第 8 行,条件语句的判断需要包括两个端点。

三、图的应用

由于许多不同的事物都可以抽象为图,因此图在现实中有非常丰

图 4.14 求两节点间最短路径例图



算法除了需要如图 4.14(b)所示数据外,还需要用户给出代表起点和终点的两个节点的编号,不妨称为 s 和 t ,然后由算法来确定一条从 s 到 t 的最短路径。

基本思路如下:从 s 开始,列出所有与其距离为 1 的节点(可直接从边集中得到);然后,列出所有与 s 距离为 2 的节点,这就要借助那些和它距离为 1 的节点在边集中发现;然后是距离为 3 的节点……这种做法在计算机科学中称为“广度优先搜索”(BFS: breadth first search)。这种方法具有十分广泛的应用,其效果相当于把一个图的其他节点相对于指定的起始节点做分层,图 4.14(c)就是以节点 1 为起始节点得到的广度优先搜索图。图中实线和虚线的意义会在下文介绍。

据此操作后,每个节点都会落在某一层,通常将起始节点所在层称为“第 0 层”,往下依次为“第 1 层”“第 2 层”……一个节点所在层数,就是它与初始节点之间的距离。^①如果目标节点 t 落在了第 i 层,我们就不仅知道了它与 s 的距离为 i (最短路径的长度),而且循着上述广度优先搜索过程所涉及的边(实线)往上,可回溯出一条最短路径来。

^① 此处我们又看到了一种序数从 0 开始的好处。如果从 1 开始,就要说一个节点所在层数减 1,就是它与初始节点间的距离,是不是较烦琐?

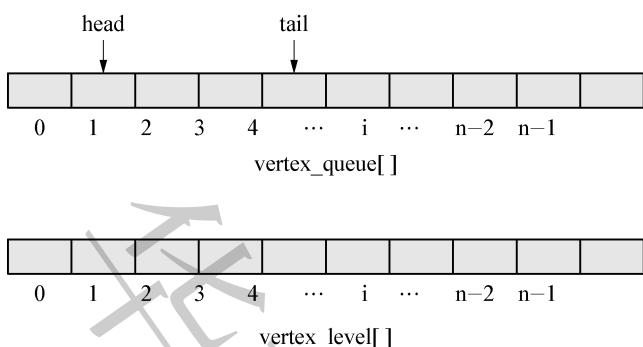


图 4.15 支持广度优先搜索的数据结构

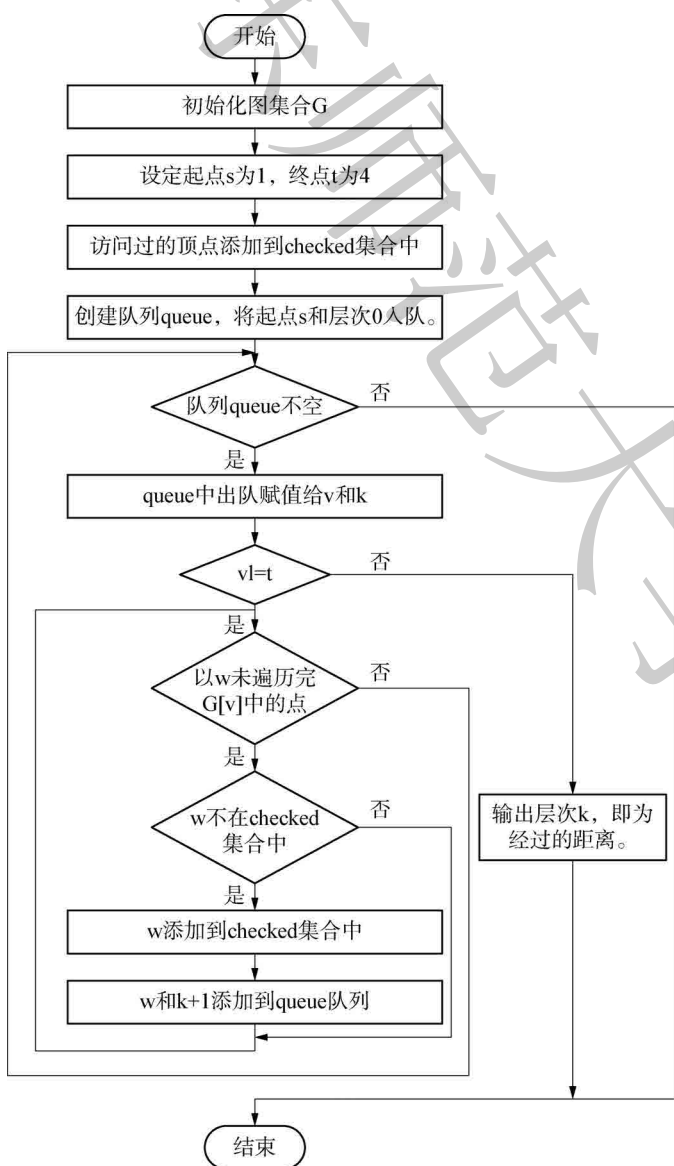


图 4.16 广度优先搜索流程图

在搜索过程中,如果某一层($i-1$)的节点已被搜索过,但它们下一层(i)的节点还没有被搜索,那么就要把它们记录下来,再经由它们发现下一层的节点,这是程序实现的关键。队列就是支持完成这个任务的最佳“数据结构”!同时,也需要记住那些已经发现的节点分别所在的层次。线性表(顺序表)就是一个自然的选择。图 4.15 即为一个示意,上面是队列,下面是线性表。初始,将 s 入队。从队列的 $head$ 所指向的节点开始,根据图的边集,将直接相连(即距离 + 1)且尚未入队的那些节点入队,并在线性表对应节点处记录所在层次(当前节点的层次 + 1)。如何判断“尚未入队”? 一个简单办法就是查看线性表中的记录。如果已经入队,则不用重复操作,这也就是图 4.14(c)中的虚线边所表示的意义。

下面以图 4.14(a)和(b), $s=1$, $t=4$ 为例,模拟这个过程。其中队列和线性表中数据的变化如图 4.17 所示。^①具体细节与图 4.14(b)中边的顺序和查找方式有关(这里采用的是每次从上往下扫描一遍的简单方式)。算法流程见图 4.16,程序实现见附录程序 4.3。

这个过程的结果条件是队列为空,也就是完成了从 $s=1$ 开始的全部节点的广度优先搜索。从中我们不仅看到 $t=4$ 位于第 3 层,也看到所有节点相对于 s 的层次。的确,广度优先搜索是发现一个节点分别到其他所有节点最短路径的一种算法。

① 为编程方便起见,这里让线性表的下标与节点编号一致,即 $vertex_level[i]$ 的值是元素 i 到元素 1 的距离。

队列	线性表							
1	—	0	—	—	—	—	—	—
0, 5, 7	1	0	—	—	—	1	—	1
5, 7, 2	1	0	2	—	—	1	—	1
7, 2, 6	1	0	2	—	—	1	2	1
2, 6	1	0	2	—	—	1	2	1
6, 3, 4	1	0	2	3	3	1	2	1

图 4.17 广度优先搜索过程中队列与线性表的变化情况

这里值得一提的是,从图 4.17 中的信息,虽然知道所有节点相对于节点 1 的距离,即最短路径长度,但并没有从节点 1 到其他节点最短路径的构成信息,即没有给出从节点 1 到节点 4 有一条如图 4.14(c)所示的最短路径 1-0-2-4。弥补这个不足的方法之一是在记录一个节点的层次信息的同时,也记录下它的父节点,从而建立起回溯的线索。

项目实践

第三章关于机器人送餐的项目有多种实现方案,除了前面讨论过的项目实践方案外,还可以采用沿着地面贴轨迹线,或者在餐桌上放置信号发射源等多种方案,请结合不同应用场景,设计和分析尽可能多的方案,归纳总结特点,并探究类似机器人送餐新方式还可以在哪些场景中有所应用。

作业练习

1 仔细品读本节第二部分中的程序,会发现其效率很低,边集合被重复访问过多次(n 次)。事实上,可以将基本观察点放在边集合上,只需将它扫描一次,就能得到所有节点的度数。试写出程序。

2 在 BFS 示例程序中,只是给出了起始节点和终止节点之间的距离,并没有给出最短路径,但相关信息都记录在算法所用的数据结构中。试修改该程序(或自己写一个),不仅给出距离,也给出最短路径(注意,在广度优先搜索的层次结果中,任何一个节点只有唯一的一条由实线标识的路径往上到达起始节点,这就是最短路径)。

第三节 哈希表*

哈希表已有几十年历史,特别适合大规模、多重复数据的记录和查找,在当今大数据时代,它的地位越发凸出。在本书引言部分的数据流例子中,比较了线性表和二叉树实现的效率。本节,将研究哈希表在该例子上的应用。

一、什么是哈希表

在网站上注册账号时,填好用户名后,系统都会判断用户名是否已被使用,如果已被使用,系统就会提示该用户名已被注册。如何检测用户名是否被使用,最简单的方法就是与存在的用户名逐个比对。如果用户名有很多,查找效率就会很低;有没有更好的方法呢?通常情况下,使用哈希表这种数据结构来组织账号数据的存储,其查找速度会非常快。

哈希表(Hash table,也叫散列表),是在数据元素的关键字和数据元素的存放位置之间建立起某种对应关系,以实现根据关键字(key)直接访问数据元素存储位置,完成数据元素查找的一种数据结构。建立这种对应关系的函数称为哈希函数(Hash function),这种对应关系称为映射。利用哈希函数建立的关键字对应存储表,称为哈希表,如表 4.3 所示。当存储数据元素时,先通过哈希函数计算出一个地址,并将数据元素存入该地址;当查找数据元素时,通过同样的哈希函数计算出数据元素的地址,并按此地址访问该数据。

表 4.3 哈希表示例

地址 [H(key)]	0	1	2	3	4	5	6	7	8	9	10	11	12
关键字 (key)	65		67			70		33	99	48			12

例 4.3 哈希表存储数据流:回到本书引言中的输入数据流例子,其中按序有 8 个关键字值(key): 4, 3, 5, 1, 4, 2, 3, 1。请分析如何用哈希表的结构来实现。

学习了第二章中数组知识后,我们知道利用数组下标进行查找非常快,因为数组在程序数据空间中是连续存放的,根据数组的起始地址和数组元素的下标值就可以直接定位每一个数组元素的存储位置,

如表 4.4 所示,数组中 $a[6]$ 的下标为 6,它的元素值为 55,因此根据下标能快速找到数组中所对应的数据。

表 4.4 数组 a

下标	0	1	2	3	4	5	6	7	N - 1
数据							55		...

哈希表的设计就是基于数组快速定位数据元素所在位置的思想,只要想办法把不同的关键字值(key)转化为 $0 \sim N - 1$ 的数组下标就可以像数组一样进行高速存储和查找。将关键字值(key)转化为数组下标的函数,就是哈希函数。通过哈希函数计算一个数据元素在数组中应该存储的位置,可用数学关系式表示为:

$$p = H(\text{key})$$

其中,H 为哈希函数,key 为关键字,p 为存储这个关键字的数组元素的下标。通过这个函数来组织存储并进行查找,如果 $H(55) = 6$,那么这个关键字就存储在数组下标为 6 的位置,如表 4.5 所示。

表 4.5 位置下标 6 中关键字为 55 的数组

下标	0	1	2	3	4	5	6	N - 1
数据	Key0	Key1	Key2	Key3	Key4	Key5	55	...

因此,构造哈希表的关键是确定哈希函数,对于一个特定 key 的哈希函数,每次计算出的下标必须一样,而且范围不能超过给定的数组长度。

根据上面的讨论,对于输入的含 8 个元素的数据流,可以设定哈希函数为 $p = x$,x 为元素数据的值。选择表长为 7,那么建立的哈希表如表 4.6 所示。

表 4.6 存有 5 个数据的哈希表

下标	0	1	2	3	4	5	6
数据		1	2	3	4	5	

在存储时,通过哈希函数 $p = x$,就能定位到 x 元素所在的位置 p,之后通过一次比较就能查找到数据的存储位置。比较的次数如表 4.7 所示,总共为 8 次。

表 4.7 比较次数统计表

数据	4	3	5	1	4	2	3	1
比较次数	1	1	1	1	1	1	1	1

讨论一种常见的情况：如果数据流中再增加一个数据，比如再输入一个数据 55，用目前的 $p = x$ 哈希函数来存储，计算出来的地址将是 55，远远超过了表长，可以扩大哈希表长度，但扩大长度会产生不必要的空间浪费。通常采用改变哈希函数的方法解决此类问题，如采用 $p = x \% 7$ (用素数)，也就是取余数作为地址，即可以把 55 存储在下标为 6 的位置，如表 4.8 所示。此方法叫除留余数法。用除留余数法构造哈希函数在后面会重点介绍。

表 4.8 用除留余数法构造的哈希表

下标	0	1	2	3	4	5	6
数据		1	2	3	4	5	55

随着输入数据的增多，有时候难免会出现不同的关键字(key)对应同一下标的状况。如 55 和 34，它们除以 7 的余数都是 6，储存位置即下标出现冲突。这时，可以应用链表法，帮助有冲突的数据定义一个新的位置。具体方法将在下一部分详细介绍。

二、哈希表的实现

构造出读取数据时间最短的哈希函数是建立哈希表的关键。同时，还需要采用合适的方法处理产生的地址冲突。哈希数据结构的性能主要取决于哈希函数、哈希表的大小、冲突处理方法等三个因素。

1. 构造哈希函数

下面介绍两种常用的函数构造方法：直接定址法和除留余数法。

(1) 直接定址法

直接定址法就是以数据元素关键字 key 本身或它的线性函数作为其哈希地址，即 $H(key) = key$ 或 $H(key) = a * key + b$ ；(其中 a，b 是常数)。

例 4.4 建立人口统计哈希表：有一个人口统计表，记录了某地区从 1 岁到 100 岁各年龄的人口数目，如表 4.9 所示，建立哈希表，实现按年龄进行快速查找。

表 4.9 人口统计表

年龄	1	2	...	99	100
人数	980	800	...	20	12

将年龄作为关键字 (key)，哈希函数可以取关键字本身，即 $H(\text{key}) = \text{key}$ ，这样当需要查找某一年龄的人数时，直接查找相应的项即可，建立的哈希表如表 4.10 所示。

表 4.10 人口统计哈希表

地址 (key)	1	2	...	99	100
年龄 (key)	1	2	...	99	100
人数 (Value)	980	800	...	20	12

如查找 99 岁的老人数，则直接读出第 99 项即可。

例 4.5 出生年份统计的哈希表：有一个按出生年份统计的人口表，记录了某地区从 1949 年到 2019 年每年出生的人口数，如表 4.11 所示，现要求建立哈希表，实现按出生年份快速查找当年出生人口数。

表 4.11 按出生年份统计的人口表

出生年份	1949	1950	...	2018	2019
人数	980	800	...	490	300

如果以出生年份作为关键字，用出生年份减去 1949 来作为地址，此时 $H(\text{key}) = \text{key} - 1949$ 。建立的哈希表如表 4.12 所示。

表 4.12 按出生年份统计的哈希表

地址 (key - 1949)	0	1		69	70
出生年份 (key)	1949	1950		2018	2019
人数 (Value)	980	800		490	300

这种哈希函数很简单，且不同的关键字不会产生冲突，但可以看

出这是一种较为特殊的哈希函数,要求关键字是连续的。实际生活中,关键字的元素很少是连续的。用该方法产生的哈希表会导致空间大量浪费,因此这种方法适应性并不强。

(2) 除留余数法

除留余数法就是将数据除以某一个常数后,取余数作为地址。这是一种最常用的构造哈希函数的方法。可以用下式表示:

$$H(key) = key \% m$$

其中,m 表示哈希表长,%为取余运算,key 为关键字。

如已知数据元素为 18, 75, 60, 43, 55, 90, 48,表长 $m = 11$ 。令哈希函数为 $H(key) = key \% m$,则 $H(18) = 18 \% 11 = 7$, $H(75) = 75 \% 11 = 9$, $H(60) = 60 \% 11 = 5$, $H(43) = 43 \% 11 = 10$, $H(55) = 55 \% 11 = 0$, $H(90) = 90 \% 11 = 2$, $H(48) = 48 \% 11 = 4$ 。所建立的哈希表如表 4.13 所示。

表 4.13 用除留余数法建立哈希表

地址	0	1	2	3	4	5	6	7	8	9	10
数据	55		90		48	60		18		75	43

人们的经验表明,除留余数法的表长 m 最好取 $1.1n \sim 1.7n$ 之间的一个素数(n 为要查找的数据元素个数),通过这种方法构造的哈希函数可以减少冲突,但不可能完全避免冲突。上例中如果增加一个数据元素 54,则 $H(54) = 54 \% 11 = 10$,和数据元素 43 所在的地址就会发生冲突。因此,解决冲突是哈希法需要面对的一个关键问题。

体 验 思 考

作为某蔬菜店的一名售货员,当顾客来买东西时,需要根据价目表告知顾客货品的价格。如果价目表的内容不是按字母顺序排列的,如表 4.14所示,你可能会为找出一种商品(如洋葱)的价格而浏览一整行,花费时间较长。现在请以表中 9种蔬菜为例画出一个哈希表(将品名中各汉字所对应的 GB 2312—1980 国标码相加,所得的整数作为哈希表的关键字,表长取为 11)来实现快速查找。

表 4.14 品名价格对照表

品名	土豆	胡萝卜	卷心菜	豆荚	番茄	茄子	辣椒	玉米	洋葱
价格	1.4	2.1	0.6	2.5	4.1	2.5	4.7	4.2	1.4

2. 哈希表冲突解决

创建哈希表和查找哈希表都会遇到冲突,两种情况下解决冲突的方法需要保持一致。下面以创建哈希表为例,阐明解决冲突的一种常用方法——链表法。

采用链表法解决冲突就是将哈希表设计成为数组加链表的组合形式,数组是哈希表的主体,链表则主要为解决哈希冲突而存在。此方法将所有哈希地址相同的不同关键字数据元素都链接在同一个链表中。

例 4.6 建立哈希表:已知一组关键字 19, 14, 23, 1, 68, 20, 27, 55, 11, 9, 哈希函数采用 $H(\text{key}) = \text{key} \% 13$, 用链表法处理冲突。编程构造这样一个哈希表,功能包括添加、查找、删除、显示整个哈希表。

程序分析如下:

(1) 定义节点类。

```
class Node:
    def __init__(self, key):
        self.key = key
        self.next = None
```

(2) 定义创建哈希表的类并初始化这个类。由节点组成数组,作为哈希表的主体。

```
class CreateTable:
    def __init__(self, indexbox):
        self.indextable = [Node] * indexbox
        for i in range(indexbox):
            self.indextable[i] = Node(-1)
```

(3) 在 CreateTable 中,要添加一个数据,可以关键字为内容生成一个节点,利用哈希函数 $\text{key} \% 13$,产生相对应的哈希地址。如果该地址没有内容,就将生成的节点放到链表头部;如果该地址有内容,就添加到该哈希地址的链表末尾。

(4) 在 CreateTable 中,要查找一个数据,只需将它经过哈希函数 $\text{key} \% 13$ 处理得到对应的哈希地址,然后直接到该地址中查找,如果

没有找到,表示数据不存在。

(5) 在 CreateTable 中,要删除一个数据,首先需在哈希表中找到该数据节点并删除,然后在该节点的后序节点和前序节点之间建立链接即可。

(6) 在 CreateTable 中,从哈希表的第一个哈希地址处开始,依次显示各个链表,结构如图 4.18 所示:

参考程序见附录程序 4.4。

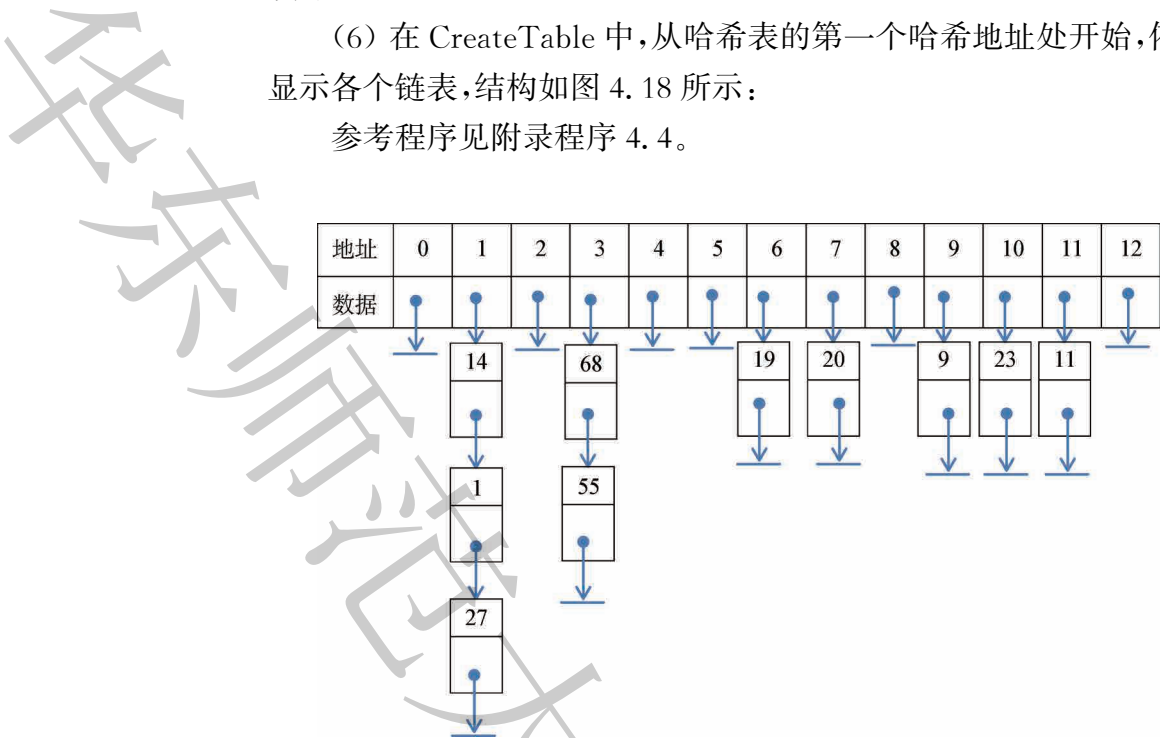


图 4.18 哈希表结构图示例

三、哈希表的基本操作

在大数据时代,人们每天进行的网络搜索,以及各类电商平台的促销活动,都需要处理亿万级别的海量数据。建立哈希表是提高大数据处理效率的一种有效方法。

哈希表有储存数据元素和读取数据元素两个基本操作。数据存储在哪个地址,从哪个地址读取均需通过哈希函数完成。当存储数据元素时,通过哈希函数将关键字(key)转换为数组的下标,然后在该下标所在位置存储元素(key, value)。当进行查询的时候,需要再次使用哈希函数将关键字(key)转换为对应的数组下标,并定位到该空间获取数据元素值(key, value),这样,就可以充分利用数组的快速定位进行数据的存储和查找。因此,哈希表的查询速度非常快。哈希表存储数据的工作流程如图 4.19 所示,查找数据的工作流程如图 4.20 所示。

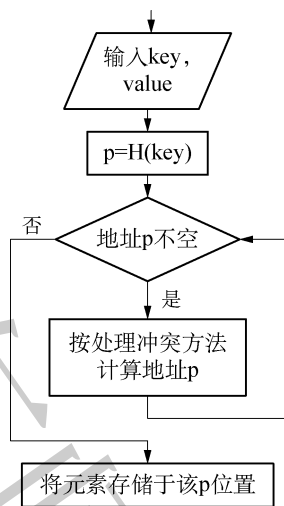


图 4.19 存储流程图

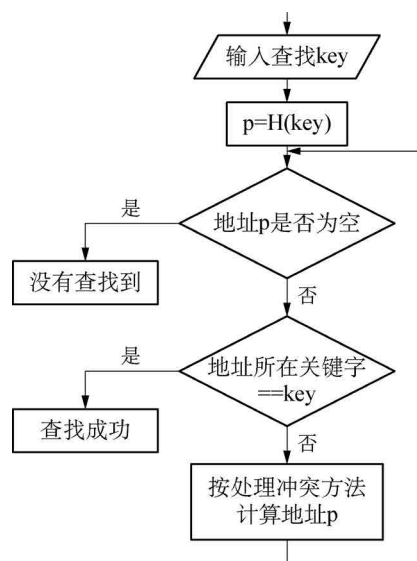


图 4.20 查找流程图

存储过程说明如下：

第一步获取数据元素的关键字 key ，计算其哈希函数值（地址）。若该地址对应的存储空间未被占用，则将该元素存入此地址；否则执行第二步解决冲突。

第二步根据选择的冲突处理方法，计算 key 的下一个存储地址。

若下一个存储地址仍被占用，则重复执行第二步，直到找到能用的存储地址为止。

查找过程说明如下：

第一步输入需查找的 key ，计算其哈希函数值（地址）。若该地址对应的存储地址为空则说明查找失败。否则，执行第二步关键字判断。

第二步取该地址数据元素的关键字 key 与输入的 key 判断，若相同则查找成功，否则根据选择的冲突处理方法，计算关键字 key 的下一个存储地址，之后重复第一步中的地址是否为空的判断。

作业练习

已知一组关键字序列为 25, 51, 8, 22, 26, 67, 11, 16, 54, 41, 其哈希地址空间为 $0, \dots, 12$ ，若哈希函数定义为 $H(key) = key \% 13$ ，采用链表法处理冲突，请画出其对应的哈希表并编程实现。



第五章

数据结构应用

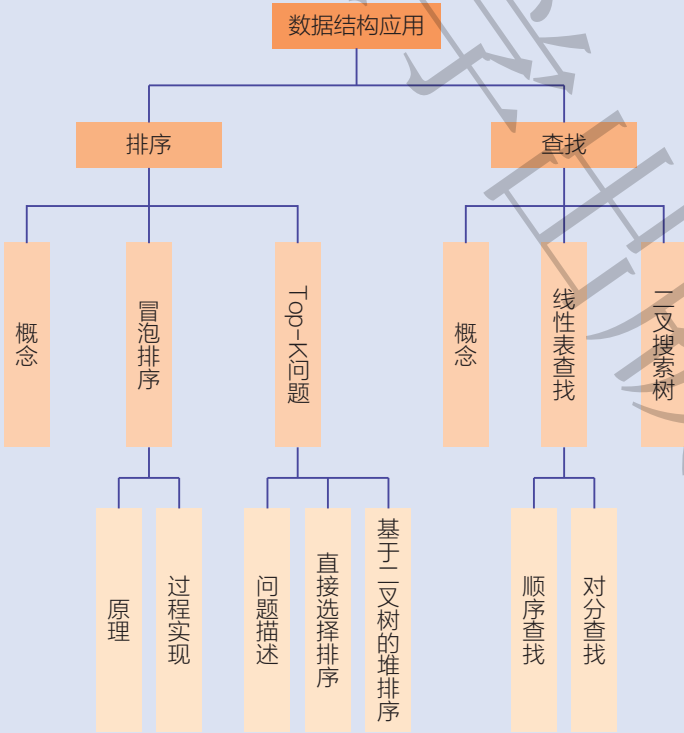
本章学习目标

- 理解“Top-K”问题的需求,体会以普通排序算法为基础来求解“Top-K”问题的不足之处。
- 理解大顶堆、小顶堆的概念及其与二叉树的关系。掌握建立大顶堆、小顶堆的方法,并通过其实现一种高效的“Top-K”算法。
- 知道节点数为 n 时,最矮二叉树的高度约为 $\lg_2 n$ 。理解基于二叉树的查找效率与树高成正比。
- 理解“平衡二叉树”的概念,通过实现一种平衡二叉树,实现 $O(\lg_2 n)$ 效率的查找。

通过前面章节的学习,我们已经熟悉了一些常用数据结构的基本形态,并结合应用实例的讨论和练习,对其在计算机程序中的基本存在形式及作用有了较深入的理解。本章将通过两个例子,展示如何在数据结构(主要是二叉树)的基本形态之上附着新的要求,从而体现出特定的含义(语义),以支持相关应用的需求。在一定意义上,可以说它们是数据结构(主要为二叉树)的“高级应用”。

这两个例子一个是排序问题的扩展,另一个与查找问题有关。排序与查找,在本模块的先导模块“数据与计算”中曾经介绍过,在本书前面章节(例如第一章中)也多有涉及。这里要强调的是,当有某些特殊要求时,数据结构的基本形态可能会应对乏力,有些时候,可以通过对数据结构进行针对性调整(增强),形成十分高效的算法。在本章的学习中,我们将通过算法与数据结构的配合、协调,体会不同程序实现的效果,从而领会数据结构在优化程序设计和提高程序执行效率中的作用。

本章知识结构



项 · 目 · 情 · 境

某单位拟通过人脸识别系统进行进出人员管理。每来一个人,系统基于来人的面部图像生成一个编码,与已经来过的人员编码进行比对,若成功,则标记“来过”,若不成功,则标记“新来”且将新的编码和图像等相关数据添加到系统中。现在遇到的问题是,系统的容量有限(假设只能最多存放 n 个人的数据),但来来往往的人可能会有许多(且一个人可以多次来往),数量会大大超过 n 。于是可能有如下情况:系统容量已满,此时来了一个人,发现他并不是已记录的 n 个人之一。此时应如何处理?(假设不拒绝进入)

项 · 目 · 任 · 务

任务 1	任务 2	任务 3
分析情境中带来的问题,梳理该管理系统要实现的功能。应意识到在系统容量有限的条件下,被标记“新来”的人不一定是第一次到来。	设计一个方案,实现新生成编码与已有编码集合元素的快速比对。若比对不成功,则将该新编码存储起来,以参与后续比对。在存储新编码的时候,若发现系统容量已满,则从系统中选择一个元素删除,以给新编码腾出空间。	基于你的方案完成一个程序,要求(1)争取有较高执行效率(即判定一个编码是否“新来”所用的比较次数尽量少)。(2)要争取在总体上有较高命中率(即尽量少误报“新来”)。

第一节 排序

通过本课程的《数据与计算》模块中的知识延伸,同学们也许已了解选择排序算法。人们还发明了其他排序算法,以追求更高的效率。同时,在许多应用场合,并不需要将全部数据都排序,只需要找出其中前 k 个元素,即所谓“Top-K”。本节将在进一步学习排序算法的基础上,讨论如何利用“堆”(一种特殊的二叉树)高效地从 n 个数据中找出 Top-K 来。

体验思考

请列举排序操作和 Top K 操作的应用场合。

一、排序的概念

排序指将一些数据元素根据特定的序关系排列成递增或递减的顺序(也称“升序”或“降序”)。例如有 10 个数据元素:

89, 71, 63, 78, 71, 85, 93, 45, 89, 97

将它们按升序排列,得到:

45, 63, 71, 71, 78, 85, 89, 89, 93, 97

需要特别注意的是,待排序的数据元素是可以重复的,而且重复元素应该保留在结果中。

数值型数据有一种自然的序关系,即它们之间的相对大小关系,易于对其进行排序,但对非数值型数据也可以进行排序。在第二章第二节,我们曾经定义过字符串之间的一种序关系,基于它就可以对字符串进行排序。不过,就学习排序方法而言,讨论数值型数据就够了,因此为简单起见,所有使用的例子中的数据元素都是数值型的。同时,由于升序和降序从方法上没有本质区别,若不特别说明,都假定讨论的是升序。一般而言,对线性表

$a_0, a_1, \dots, a_i, \dots, a_{n-1}$

中的数据排序,就是要对其中的数据重新排列

$a'_0, a'_1, \dots, a'_i, \dots, a'_{n-1}$

使其满足,若 $i < j$, 则 $a'_i \leq a'_j$ 。

二、冒泡排序

排序操作是许多计算机应用的核心操作,其效率的高低对应用影响很大,人们在过去几十年里不断研究改进,产生了若干经典的排序算法,诸如选择排序、冒泡排序、快速排序、归并排序,等等。在《数据与计算》模块的知识延伸中,同学们可能已经初步了解了选择排序。本节将首先学习另一个经典的排序算法——冒泡排序,在此之后,我们将考虑一个更一般的问题:从 n 个元素中选出最大(或最小)的 k 个元素,即所谓“Top-K”问题。^①着重分析在不同数据结构支持下的性能差异。

冒泡排序(bubble sort)是一种基于交换的排序,它的基本思想是:从左往右比较相邻的元素,如果左边的比右边的大,就交换位置;如此继续,直到最右边尚未排序的元素。这称为排序过程的“一轮”。第一轮结束,最大的元素被移至最右边,重复以上步骤 $n-1$ 轮,每次少考虑一个数,所有元素就被按照从小到大顺序排列好了。

例 5.1 冒泡排序:有 8 个数:89, 71, 63, 78, 85, 93, 45, 97, 要求按从小到大排序,以下是排序的过程:

第一轮:

初始值:89, 71, 63, 78, 85, 93, 45, 97

第一次比较,89 比 71 大,彼此交换,得到:71, 89, 63, 78, 85, 93, 45, 97

第二次比较,89 比 63 大,彼此交换,得到:71, 63, 89, 78, 85, 93, 45, 97

第三次比较,89 比 78 大,彼此交换,得到:71, 63, 78, 89, 85, 93, 45, 97

第四次比较,89 比 85 大,彼此交换,得到:71, 63, 78, 85, 89, 93, 45, 97

第五次比较,89 比 93 小,不用交换,得到:71, 63, 78, 85, 89, 93, 45, 97

第六次比较,93 比 45 大,彼此交换,得到:71, 63, 78, 85, 89, 45, 93, 97

第七次比较,93 比 97 小,不用交换,得到:71, 63, 78, 85, 89,

^① n 个元素的排序问题是“Top-K”问题的一个特例,即 $K=n$ 时的情形。

45, 93, 97

经过第一轮比较后,8个数中最大的数 97 已经移至最右,不用再参加以后的比较,下一轮只需比较前 7 个数,即每一轮少比较一个元素,依此类推。

第二轮,经过六次比较,得到: 63, 71, 78, 85, 45, 89, 93, 97

第三轮,经过五次比较,得到: 63, 71, 78, 45, 85, 89, 93, 97

第四轮,经过四次比较,得到: 63, 71, 45, 78, 85, 89, 93, 97

第五轮,经过三次比较,得到: 63, 45, 71, 78, 85, 89, 93, 97

第六轮,经过两次比较,得到: 45, 63, 71, 78, 85, 89, 93, 97

第七轮,经过一次比较,得到: 45, 63, 71, 78, 85, 89, 93, 97

这样就完成了排序。实现这个过程的一个示例程序如下。

```
for i in range(len(lst)-1):    # 这个循环负责设置冒泡排序
    进行的次数
    for j in range(len(lst)-i-1):    # j 为列表下标,比较到
        最右边尚未排序的元素结束
        if lst[j]>lst[j+1]:    # 这里是比较相邻的两个元素
            lst[j], lst[j+1]=lst[j+1], lst[j]    # 做交换
```

如果有 n 个元素,能看到程序中的 if 语句要执行 $(n-1) + (n-2) + \dots + 1 = n(n-1)/2$ 次,于是可以说冒泡排序的时间复杂度是 $O(n^2)$ 。

对比冒泡排序过程中可能有大量的交换操作,选择排序是基于定位(即每一轮的任务是确定下一个数在线性表中的位置,只做一次交换)展开的,但两种方法殊途同归,最后都能得到一个有序数据表。

探究活动

观察冒泡排序的过程,设计一个从未位开始进行比较的程序。总结冒泡排序的特点,理解“冒泡”的内涵。

三、Top-K 问题及其挑战

前面提到,在许多应用场合,不需要将整个数列(假设 n 个数)完

全排序,只需要找到前 k 个元素就可以了。如何做到呢? 一个简单的方法是,利用排序算法,将 n 个数排好序,然后取前 k 个。这样做结果当然正确,但效率不高,无论用选择排序还是冒泡排序,都要进行 $O(n^2)$ 次比较操作。

改进方法是,如果用选择排序,只需要选出前 k 个元素就好,如果用冒泡排序,只需要控制运行 k 轮即可,这样只需执行 $O(n \cdot k)$ 次比较操作。如果 k 不是很大,用这种方法就比较理想。但在很多大数据实际应用中, n 可能上亿, k 可能上千,因此 $n \cdot k$ 会很大。还有没有什么办法能提高效率呢?

下面的目标,就是要看如何利用一种更高级的数据结构,让求解 Top-K 问题的效率提高到 $O(n + k \cdot \log_2 n)$,这和上面的 $O(n^2)$ 或 $O(k \cdot n)$ 相比,就是一个很大的进步。

无论是在《数据与计算》模块中提到的选择排序,还是上面的冒泡排序,它们都是基于线性表的,都是让一个“比较”操作过程沿线性表一轮轮反复进行,在必要的时候进行元素的位置交换。我们可以注意到,每一轮排序都会产生一些比较结果信息,但它们在后续的排序中却完全没有用到,这是一种“浪费”。能不能减少这种浪费,从而提高排序的效率呢?

四、用一种特别的二叉树解决问题

利用已有比较信息提高效率的认识基础是:如果已知 $x > y$ 且 $y > z$,则必有 $x > z$,所以就不需要再对 x 和 z 进行比较了。但如何记住并利用这样的信息呢? 通过第四章二叉树结构的学习,我们会发现利用二叉树能从多个角度很自然地将这样一种传递性表达出来。一种表达方式如同在第一章第二节中所看到的,根节点上的值比左子树的大,比右子树的小,这就意味着右子树上的值都比左子树上的值大。另外一种就是我们这里要采用的,让根节点上的值比左右子树节点上的值都大,于是,从根节点到任何叶子节点,其路径上节点的值都是有序的。

我们需要用到前面简略提到过的一种有特别要求的二叉树——完全二叉树。为了进一步理解完全二叉树的含义,请回顾一下第四章第一节的学生活动部分中关于“完美二叉树”的描述。完全二叉树就是将某个完美二叉树的叶子节点从右边删去若干节点后剩下的二叉树。它会有这样的特点:(1)除了最后一层的节点可能例外,所有其他层的节点都是满的(即第 0 层有 1 个节点,第 1 层有 2 个节点,……,第 i 层有 2^i 个节点);(2)如果最后一层不满,则那些节点优先靠左。

图 5.1 中,(a)和(b)是完全二叉树,(c)和(d)不是。

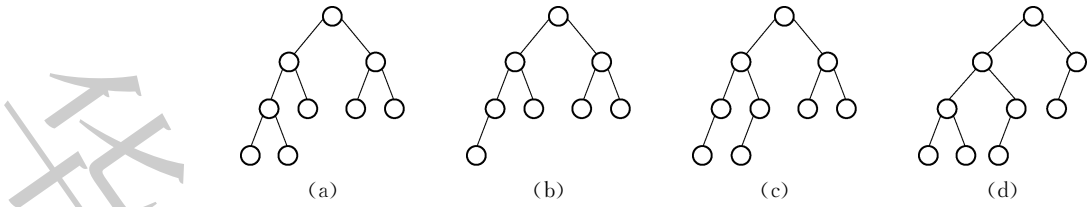


图 5.1 完全二叉树辨识

体 验 思 考

若一棵完全二叉树的高度为 h , 那么它的节点个数的范围是多少? 换个角度问这个问题就是: 一共有多少棵高度为 h 的完全二叉树?

这样的二叉树有一个很有价值的性质, 即树高(h)与节点数(n)成对数关系($h \approx \log_2 n$)。这种性质是它能产生高效率的关键。为了能够保存已有信息, 考虑构造一种完全二叉树: 它的任意节点内所存储的数据不大于(或不小于)其子节点(如果存在)内的数据。满足如上性质的完全二叉树称为“堆”(heap)。

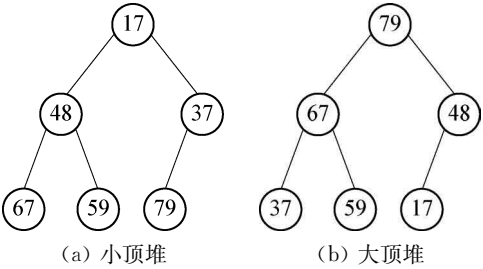


图 5.2 小顶堆和大顶堆

如果节点内的数据不大于其子节点内的数据, 构造出来的堆就称为“小顶堆”(min heap), 如图 5.2(a)所示; 如果节点内的数据不小于其子节点内的数据, 构造出来的堆就称为“大顶堆”(max heap), 如图 5.2(b)所示。

一棵完全二叉树的信息可以通过顺序存储方式自然地存入一个顺序表内, 所以一个堆也可以存入一个顺序表内, 并且通过下标可以找到任一节点的父节点或子节点。在接下来的讨论中所提到的堆, 就是指存储在顺序表内的一棵完全二叉树。表 5.1 就是用顺序表存储图 5.2(a)中的堆时所呈现的顺序表的情况。

表 5.1 顺序表存储

0	1	2	3	4	5
17	48	37	67	59	79

堆有如下性质：

(1) 在一个堆中从根节点到任何一个叶子节点的路径上,各节点里所存储的数据按规定的优先次序存放。

(2) 堆中最优先的元素必定位于二叉树的根节点里。

(3) 在一个堆的最后加上一个数据元素,整个结构仍然是一个完全二叉树,但并不一定是一个堆。

(4) 一个堆去掉堆顶,其余元素形成两个子堆。

(5) 去掉一个堆中最后的数据元素,剩下的数据元素仍构成一个堆。

堆排序的原理如下:首先,把待排序的数据构造成大顶堆(以下提到的堆均为大顶堆的简写),之后通过从堆中不断选出最大元素即可完成排序。所以整个堆排序分成两个主要步骤:第一,将初始完全二叉树调整成堆;第二,取出堆中最大元素并将其重新调整成堆。

先考虑第二个步骤的实现。由大顶堆的性质知最大的元素就在堆顶,所以每次取出堆顶元素。为了每次都能在堆顶取出最大元素,需将剩下的数据元素恢复成堆,为此要先恢复成一棵完全二叉树。于是,我们先将原来堆的最后一个元素 y 替换到当前堆顶元素的位置,恢复成一棵完全二叉树,并且 y 的左子树和右子树都分别保持为一个堆。图 5.3(a)到(b)表示出了这种变化。然后我们需要将现在的树根元素通过一种称为向下筛选的操作,调整到树中的适当位置,实现完全二叉树到堆的转变。所谓“向下筛选”,意指此时虽然 y 在树根的位置,但它不一定是最大的,需要让它沿着二叉树往下沉,不断和较大元素交换位置,直到不再可能,操作如下:

(1) 判断 y 是否为叶节点,如果是,操作结束。

(2) 比较 y , y 的左子节点,以及 y 的右子节点的大小。

(3) 如果 y 最大,操作结束。

(4) 如果 y 不是最大的,就把它与最大的做位置交换,回到(1)。

图 5.3(b)到(c)到(d)是这个过程的一个示例。

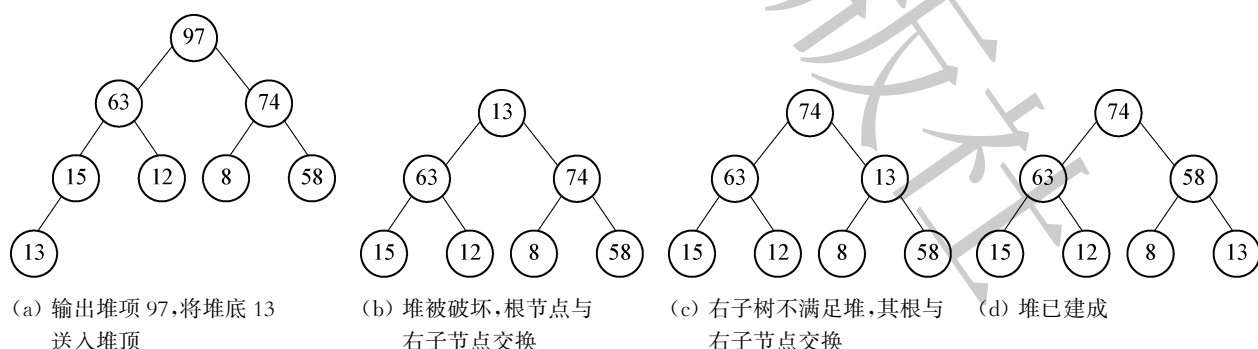


图 5.3 自堆顶到叶子的调整过程

再考虑第一个步骤的实现,将完全二叉树中无序的数据元素调整为堆。做法是从后往前,逐个考察数据元素,看以它为子树根的完全二叉树是否已经是堆,若是,则继续到下一个元素,若不是,则做一次以当前元素为顶的向下筛选。

这里,从后往前的顺序很关键,它保证了当我们考察一个元素的时候,它的两个子树都已经是堆,从而在判断一个子树是否为堆的时候只比较树根和左右子节点就可以了。

例 5.2 堆排序: 以下用顺序表 23, 7, 90, 65, 98, 11, 46 为例,具体解释一下堆排序的过程。图 5.4(a)是将输入序列按层次建立的一棵完全二叉树。

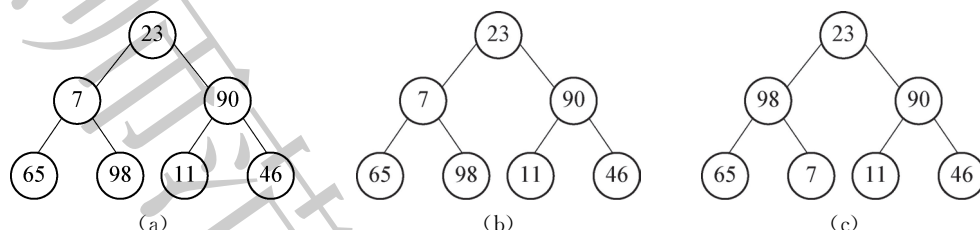


图 5.4 堆排序过程示例(一)

(1) 构造堆结构: 首先进行第一阶段,通过调整数据建成一个大顶堆,按照层次从下往上,一层节点做完再做另一层,从而保证每考虑一个节点的时候,它的左、右子树都已经是大顶堆了。图 5.4(b)是数据元素“90”试图进行向下筛选的结果,由于下面两个都小,它就保留在原处。然后是数据元素“7”往下筛选,与较大的“98”交换了位置,见图 5.4(c)。

接下来是上一层的“23”,向下筛选。注意,筛选过程一是要选择较大的,二是要尽可能向下,于是我们看到图 5.5(a)中的“23”到了最左下端,“98”和“65”则向上移动了一层,于是调整成为一个大顶堆。顶层的“98”就是最大的。这个过程的效率如何? 假设树高为 h (节点数 n 不超过 $2^{h+1} - 1$),位于第 i 层的节点向下筛选过程最多用到 $h - i$ 次比较操作,而第 i 层共有 2^i 个节点,于是总的操作次数估计为 $\sum_{i=0}^{h-1} (h - i) \cdot 2^i$, 其求和的值约为 n 。^① 最终构造的大顶堆如图 5.5(a)所示。

① 根据 $\sum_{i=0}^m 2^i = 2^{m+1} - 1$, $\sum_{i=0}^m i \cdot 2^i = (m - 1) \cdot 2^{m+1} + 2$, 不难得出此结论。

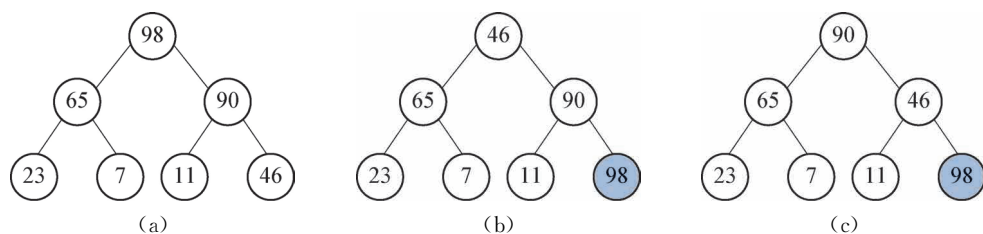


图 5.5 堆排序过程示例(续一)

(2) 读取堆元素：有了这个大顶堆，就可以开始一个个取出最大元素了。基本做法就是每次把堆顶元素(x)取出，和最后一个元素(y)交换位置，然后让y向下筛选，调整成一个新的堆(注意调整的时候不涉及已经选出的元素)。图 5.5(b)就是第一个数交换的结果，图 5.5(c)则是新的调整结果。(注：图 5.5、图 5.6、图 5.7、图 5.8 中带底纹的节点是已读取出的数据，不参与下一轮的堆调整过程。)

图 5.6(a)是第二个数取出后交换的结果，图 5.6(b)则是新的调整结果。如此继续进行，如图 5.6(c)、图 5.7(a)、图 5.7(b)、图 5.7(c)、图 5.8(a)、图 5.8(b)所示，最后到了图 5.8(c)。我们看到，最后一个，也就是最小一个数据出现在了堆顶。

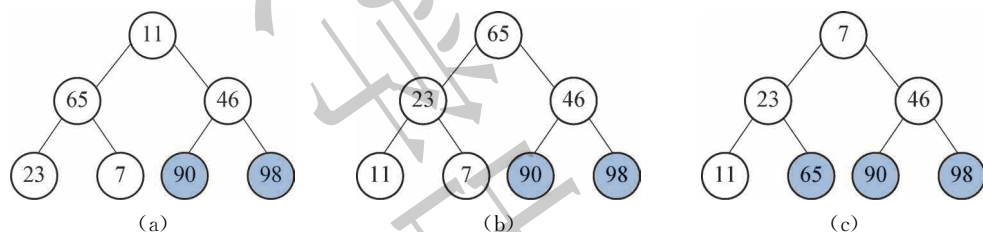


图 5.6 堆排序过程示例(续二)

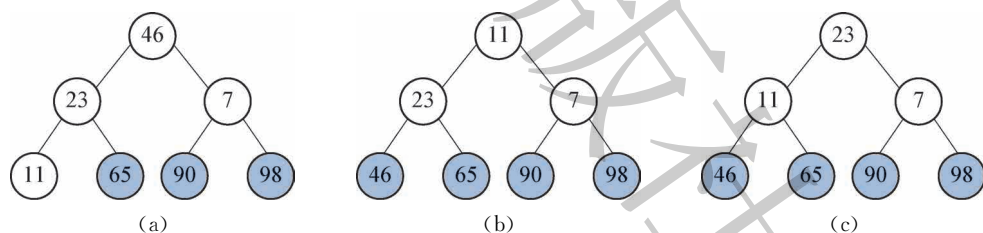


图 5.7 堆排序过程示例(续三)

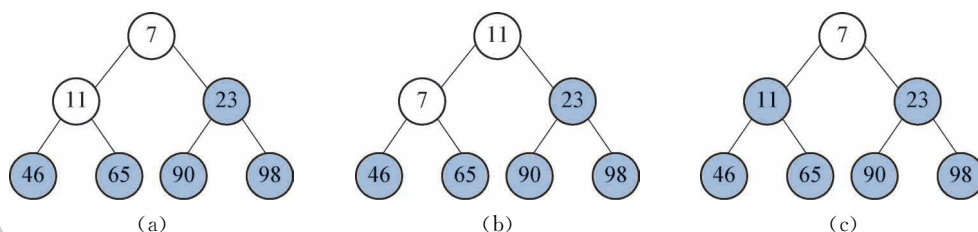


图 5.8 堆排序过程示例(续四)

这个过程的效率如何呢? 每取出一个元素后进行堆调整,最多需要 $\log_2 n$ 次操作,若要取最大的 k 个元素,也就是 $k \cdot \log_2 n$ 。如果考虑建堆的时间,整个效率就是 $n + k \cdot \log_2 n$ 。这要比前面提到的 n^2 或 $k \cdot n$ 更有效率。

(3) 编写堆排序程序: 理解了这个算法过程,如何用程序实现呢? 编程时,要充分利用完全二叉树的性质。首先,当我们把一棵完全二叉树的节点数据,依照层序遍历的结果放到一个顺序表中之后,每一个非叶子节点的位置和它的子节点的位置有一个很简单的对应关系,即若一个节点在顺序表中的下标为 i ,则它的左子节点的下标为 $2 \cdot i + 1$,右子节点下标为 $2 \cdot i + 2$ 。表 5.2 就是图 5.8(c)小顶堆在顺序表中按照层序遍历放置的结果,我们能验证上面所表述的性质。

表 5.2 图 5.8(c)小顶堆的顺序存储示意

0	1	2	3	4	5	6
7	11	23	46	65	90	98

另外一个让编程方便的性质是,完全二叉树的叶子节点数要么比非叶子节点多一个,要么与其相等,这意味着若完全二叉树有 n 个节点,那么 Python 的向下取整除法操作“ $n//2$ ”结果就是其非叶子节点的个数。也就是说,当给定一个以层序存储在顺序表中的完全二叉树时,就能清楚地知道其所有非叶子节点恰好就在 $\text{range}(n//2)$ 中。建堆的时候,从后往前,一个个考察非叶子节点;调整堆的时候,只需对根节点(下标为 0)做向下筛选即可。

附录中的示例程序 5.1,就是基于完全二叉树的顺序存储,在起初建堆过程以及后来堆调整的过程中,不断尝试调整非叶子节点和其子节点之间的位置,以实现向下筛选的操作。该程序做的是完整的排序。如果只关心 Top-K,那么只需让调整堆的次数为 k 就可以了。

探究活动

- 1 尝试对文中提到的完全二叉树在层序遍历结果中的节点与其儿子节点的下标关系给予证明。
- 2 尝试对文中提到的完全二叉树非叶子节点与叶子节点个数的关系给予一般的证明。

作业练习

- 1 对一组给定的数据：35, 78, 41, 99, 23, 76, 56, 88, 21, 97, 45, 分别写出冒泡排序和堆排序算法在处理上述数据时各轮的结果。
- 2 有 6 位裁判为运动员评分, 给出的分数分别为 53, 50, 68, 51, 64, 62。采用直接选择排序算法对其进行排序, 若完成第一轮时的结果为：68, 50, 53, 51, 64, 62, 则完成第二轮时的结果是()。
A 68, 50, 53, 51, 64, 62
B 68, 64, 62, 53, 50, 51
C 68, 64, 53, 51, 50, 62
D 68, 64, 62, 51, 50, 53

第二节 查找

在本书第一章中,我们看到一个与“查找”有关的例子:数据不断到来,对于每一个新数据(x),我们要在已接收到的数据中查看 x 是否存在,结论是采用二叉树结构要比采用线性表效率高。作为本书的最后一节,我们将回到那个例子,对其做进一步的讨论,分析不同的数据结构在同一问题上表现的差异,以及在同一个数据结构上不同算法的性能差别。

一、查找的概念

很多人都有在一堆学习材料中试图翻找某一份材料的体验。无论是在日常生活还是在计算机应用中,查找(search,或称搜索)都是一件十分普遍的事情。

有些同学平时喜欢整理,习惯将自己的东西分门别类收拾得整整齐齐,查找时很容易缩小范围,从而效率较高。而不喜欢收拾的同学,认为平时的整理也需要花费时间,光追求查找的高效率总体上不一定最优。这和计算机应用中的情形完全一致。如果所关心的数据集是静态的,一旦形成就不再变化,将数据元素一次性排好序就能最有效地支持反复多次查找。如果数据集本身在应用过程中会不断变化,添加和删除操作频繁,而并不经常进行查找,是否应维持一个有序数据集以支持查找就是一个很值得权衡的问题。

由于数据集中的数据元素可能有多个数据项,和上一节中学过的排序一样,在计算机中的查找常常是针对某一个特定的数据项而言的,不同的应用场合可能不一样。例如学生注册信息,包括姓名、学号、年级等数据项,有时可能需要根据姓名来查找,有时候则需要根据学号来查找。查找所针对的数据项称为“键”(key)。查找就是根据给定的一个键值,在数据集中确定是否存在键值等于给定值的数据元素的操作。这种认识在实际应用中是很有意义的,在有些场合,可能需要确定有多少键值等于给定值的数据元素,因为两个不同的数据元素可能包含相同的数据项。

不过,在讨论查找算法的时候,通常会简单地认为数据元素就一个数据项,也就不再特别强调键的概念,这样便于抓住核心思想。

下文中我们会首先讨论在静态数据集情形下的两种不同的查找算法,即顺序查找与对分查找。从数据结构的角度讲,使用线性表能

对它们进行最自然的表达;从编程实现的角度,数组或列表则是最自然也最高效的运用。

然后分析动态数据集的情况,例如第一章中数据流检测问题。我们将看到线性表在这种场合可能效率不高(无论是顺序查找还是对分查找),而“二叉搜索树”提供了对这类问题的一种新视角,能够提高效率。

二、线性表上的查找

1. 顺序查找

一个线性表($a_0, a_1, \dots, a_{n-1}$)中的元素,不仅有简单的前后关系,而且可以通过下标直接访问。当给定待查找的值 x ,自然会想到从头到尾一个个比较(x 与 a_i 是否相等),相等,则查找成功,如果整个表都被查遍了,仍未找到所需的数据元素,则查找失败。这就叫顺序查找(sequential search)。程序示例如下。

```
def sequentialSearch(list, key):  
    pos = 0  
    while pos < len(list):  
        if list[pos] == key:  
            return pos  
        else:  
            pos = pos + 1  
    return -1
```

不仅顺序存储的线性表能够有效支持这种方法,链接存储的线性表也可以。因为查找过程顺着链向下走,一次一步,每一步都有用,于是效率也高。观察程序中的循环,我们能看出,这种方式,每一次查找,最少需要执行 1 次比较,最多执行 n 次比较,平均情况为 $\frac{n}{2}$ 次。

2. 对分查找

如果仅仅是把学习资料随意摆放在书架上,按前面讲的逐个顺序查找,要找到一本书还是比较困难的。但如果事先整理一下,将图书

按照书名的拼音排序放置,要找到某一本书就相对容易了。

如果顺序存储的线性表中的各数据元素是按其关键字的大小顺序(以下假定按从小到大的顺序)排列,则该线性表为有序表。这时对它的查找可以采用对分查找(binary search)法。

对分查找的基本思路是:先取表的中间位置数据元素的关键字与给定值进行比较,如果两者相等,则查找成功;如果给定值比该数据元素的关键字大,则所查找的数据元素只可能在表的后半部分,否则就只可能在表的前半部分;然后在这半部分中再取中间位置数据元素的关键字进行比较,又可舍去这半部分中的一半;如此反复进行,直到查找成功或确定查找不到为止。

采用对分查找需满足两个条件:线性表必须采用顺序结构(否则效率很低);表中数据元素按关键字有序排列。

在对分查找算法中需要用到三个变量:low、high 和 mid,分别用来指示被查找的那一部分表的表头、表尾和中间位置。其中 $mid = (low + high) // 2$ 。

对分查找的过程如下:

假设 key 为给定值,序列 a 存放了 n 个已按升序排列的数据元素(假设关键字即为数据元素的值)。在使用对分查找时,把查找范围 [low, high] 的中间位置上的数据 a[mid] 与给定值 key 进行比较,有如下三种情况:

(1) $a[mid] > key$, 由 a 中数据元素的递增性,可以确定 [mid, high] 内不可能存在值为 key 的数据元素,必须在新的范围 [low, mid-1] 中继续查找。

(2) $a[mid] = key$, 则找到了所需的数据元素。

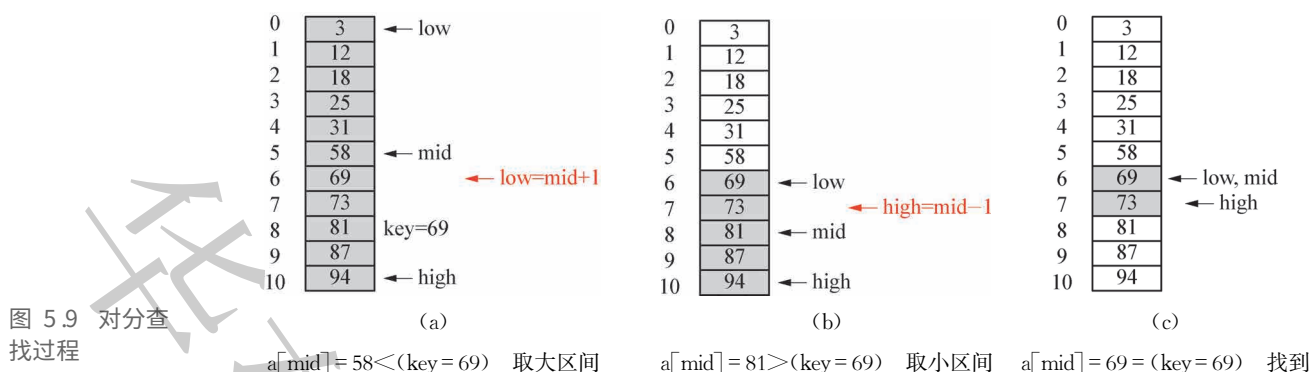
(3) $a[mid] < key$, 同样根据数据元素的递增性,必须新的范围 [mid+1, high] 中继续查找。

这样,除了出现情况(2),在通过一次比较后,新的查找范围将不超过上次查找范围的一半。

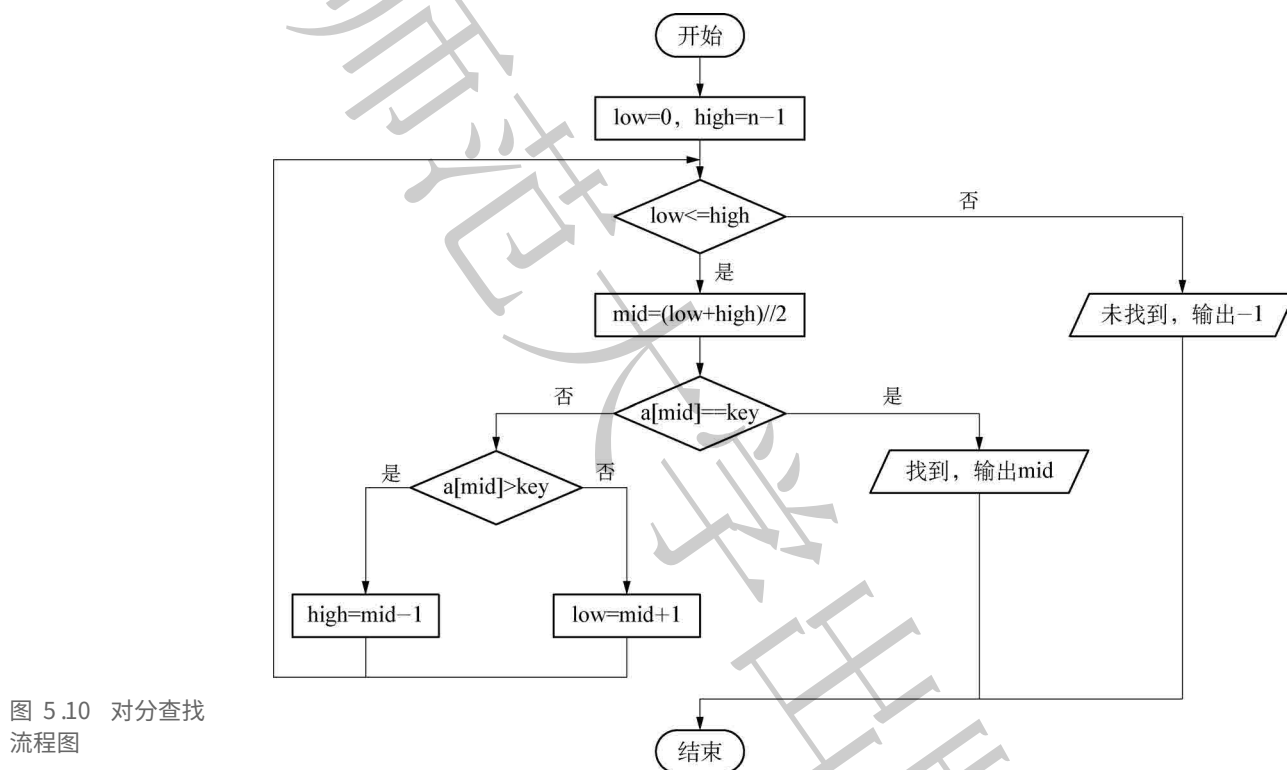
例 5.3 对分查找:已知含 11 个数据元素的有序表(关键字即为数据元素的值)如下所示:

3, 12, 18, 25, 31, 58, 69, 73, 81, 87, 94

对于给定值 key = 69 的查找过程如图 5.9 所示。



结合上面的思路分析,对分查找算法的核心流程如图 5.10 所示。



对分查找程序如下所示:

```

def binarySearch(list, key):
    low = 0
    high = len(list) - 1
    while (low <= high):
        mid = (low + high) // 2
        if list[mid] == key:

```

```
        return mid
    elif list[mid] > key:
        high = mid - 1
    else:
        low = mid + 1
    return -1
```

体验思考

校运会上报名参加跳远的选手的编号有：121、138、163、171、192、201、213、224、251、267。采用对分查找，查找编号为 251 的学生信息，写出依次被访问的编号。

探究活动

利用递归的思路实现对分查找。

假设 key 为给定值，序列 a 存放了 n 个已按升序排列的数据元素（假设关键字即为数据元素的值），查找范围是 $[bw, high]$ 。

- （1）确定该区间的中间位置 mid 。
- （2）将给定值 key 与 $a[mid]$ 进行比较，有如下三种情况：
 - ① $a[mid] = key$ ，则返回此位置 mid 。
 - ② $a[mid] > key$ ，则在新的范围 $[bw, mid - 1]$ 中进行递归调用查找。
 - ③ $a[mid] < key$ ，则在新的范围 $[mid + 1, high]$ 中进行递归调用查找。

三、二叉搜索树

类似于研究排序问题时学过的堆，二叉搜索树（binary search tree, BST）也是一种特殊二叉树。二叉搜索树没有完全二叉树那样的结构性要求，但对节点中的数据关系有以下要求。二叉搜索树可以是一棵空树，或者是具有下列性质的二叉树：若左子树不空，则左子树所有节点的值均小于根节点的值；若右子树不空，则右子树所有节点的值均大于根节点的值；左、右子树也分别为二叉搜索树；没有键值相等的节点。如图 5.11 所示即为一棵二叉搜索树。当节点数为 n ，二叉树最矮可达 $\log_2 n$ ，对应搜索复杂性为 $O(\log_2 n)$ ，这是我们追求的效

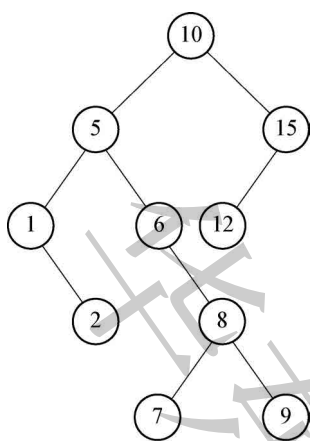


图 5.11 二叉搜索树图

率目标。

二叉搜索树最关键的操作是数据项的插入和删除。而这两项操作通常都需要确定某个元素的位置,所以先考虑如何在二叉搜索树中查找一个数据。

查找数据的实现方法:

(1) 将要查找的数据定义为 `key`,并设 `cur` 为根节点,`cpar` 为 `cur` 的父节点。

(2) 如果 `cur` 是空节点,则查找失败,树中不存在这样的 `key`。将 `key` 和 `cur.data` 比较,如果 `key` 等于 `cur.data`,查找成功;如果 `key` 大于 `cur.data`,则执行步骤(3)操作;如果 `key` 小于 `cur.data`,则执行步骤(4)操作。

(3) 将 `cur` 赋值为 `cur` 的右儿子节点。再次执行步骤(2)操作。

(4) 将 `cur` 赋值为 `cur` 的左儿子节点。再次执行步骤(2)操作。

插入数据的实现方法:

由于二叉搜索树一般不支持树中存在相同的元素,所以若插入的元素和树中某个元素相同,则不进行任何操作。插入数据本质上就是找到某个节点,然后在该节点下面添加新数据。

(1) 新建节点 `new`,使其数据为需要插入的 `key`,左、右子节点为空。

(2) 如果根节点为空,直接将 `new` 变成根节点,操作结束。

(3) 将 `cur` 赋值为根节点,`cpar` 赋值为 `None`。

(4) 判断 `cur` 是否为空。如果 `cur` 为空,用 `new` 代替 `cur`,并修改 `cpar` 的子节点数据。否则,比较 `cur.data` 和 `new`。如果 `new` 等于 `cur.data`,则操作结束;如果 `new` 大于 `cur.data`,则执行步骤(5)操作;如果 `new` 小于 `cur.data`,则执行步骤(6)操作。

(5) 将 `cpar` 赋值为 `cur`,`cur` 赋值为 `cur` 的右儿子节点。再次执行步骤(4)操作。

(6) 将 `cpar` 赋值为 `cur`,`cur` 赋值为 `cur` 的左儿子节点。再次执行步骤(4)操作。

删除数据的实现方法:

删除节点相对复杂一点,首先需要定位该节点;然后,删掉该节点;最后,重新整理数据元素,使删除节点后的新树仍然是二叉搜索树。

删除的节点在不同的位置对于树的影响各不相同,因此需分情况讨论删除的操作。

(1) 设需要删除的元素的数据为 `key`, 在树中进行查找, 如果没有相应节点, 操作结束; 否则, 设 `cur` 为满足要求的节点。

(2) 如果 `cur` 没有左子树, 则执行步骤(3)操作; 否则, 执行步骤(4)操作。

(3) 将 `cur` 用 `cur.right` 代替, 操作结束。

(4) 设 `cur` 的左子树最右下方的节点为 `mostright`, 将 `cur.right` 变成 `mostright` 的右儿子节点, 将 `cur` 替换成 `cur.left`。

附录程序 5.2 是一个示例程序。该程序除了上述基本操作外, 也包含通过一系列插入操作构建一棵二叉搜索树, 然后对该树做中序遍历, 从而得到一个增序数据集的过程。

体 验 思 考

将序列 46, 78, 36, 90, 70, 45, 120, 13, 65, 52, 23 依次插入一棵开始为空的二叉搜索树, 然后画出这个二叉搜索树。

四、进一步挑战 *

有的同学可能会发现, 插入和删除某些数据元素时, 有可能会導致二叉搜索树“畸形”, 严重偏向某一边的子树, 最严重的时候(例如插入的元素序列本身已经是有序的)就像是一个线性表。这将导致查找操作的效率很低, 甚至达到了 $O(n)$ 的复杂度。这种情况与创建二叉搜索树的目的, 即希望操作复杂度只有 $O(\log_2 n)$ 不相符。

为了避免这一情况, 在每次插入节点后, 如果发现“情况不妙”, 就要对树进行某种调整, 在保持二叉搜索树性质的同时, 还使得每一个节点左子树的高度和右子树的高度之差不超过 1, 满足上述条件的二叉树被称为“平衡二叉树”。不妨回到第一章第二节中的例子, 有数据流 4, 3, 5, 1, 4, 2, 3, 1, 随着数据的一个个到来, 一棵二叉搜索树建立起来, 图 1.3 展示了这个过程。观察这个过程, 我们会感到前面都比较理想, 但输入第 6 个数据“2”之后, 按照规则, 它被插入成为“1”的右儿子, 这样一来这棵二叉树就“失衡”了。如图 5.12(a) 所示。

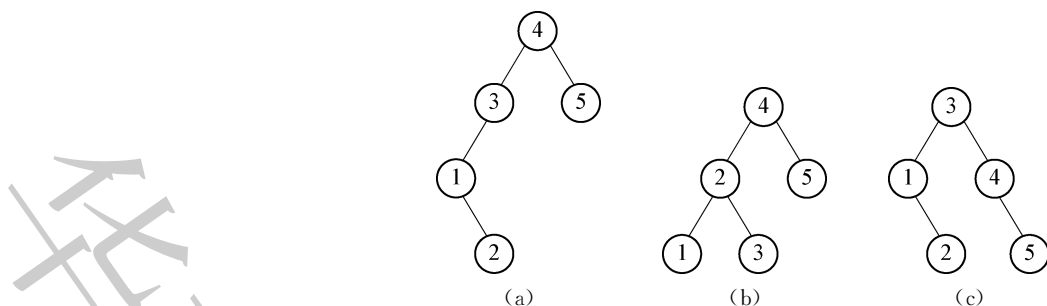


图 5.12 二叉搜索树的平衡操作

这里的失衡表现在以节点 3 为根的子树,左右两边子树的高度差超过了 1。实际上,以 3 为根的子树上的节点可以做一下位置调整,在保证二叉搜索树的基本性质的同时也保持其平衡状态,如图 5.12(b)所示。而在图 5.12(a)中从节点 4 的角度看,2 的加入也使树失衡了,可以节点 4 为“轴”做一次调整,得到图 5.12(c),在保证二叉搜索树基本性质的同时也恢复了其平衡状态。如何在发现失衡后以不大的代价进行此类调整,在 50 多年前已是人们面临的一个挑战。下面通过一个例子来看一种方法(该方法包括对因插入和删除两个操作导致的失衡所进行的调整,这里我们只讨论插入)。

一棵平衡二叉树,在插入一个新节点后,每一个节点将具有如下五种状态之一:

两个子树的高度一样;称为完全平衡,用 0 表示。

左子树高度比右子树高 1;称为左沉(但依然平衡),用 -1 表示。

右子树高度比左子树高 1;称为右沉(但依然平衡),用 $+1$ 表示。

左子树高度比右子树高 2;称为左失衡,用 -2 表示。

右子树高度比左子树高 2;称为右失衡,用 $+2$ 表示。

以图 5.12(a)为例,对应节点 1、2、3、4、5,就有 $+1$ 、 0 、 -2 、 -2 、 0 。而在图 5.12(b)中,节点 4 的状态为 -1 ,其他节点都是 0。显然,插入一个节点后,若每个节点都处于 0 、 -1 或 $+1$ 状态,就不需要做任何事情。因此,调整发生在有节点的状态变成 -2 或 $+2$ 时。

如果一个节点(X)的状态变成了 -2 ($+2$ 的情况对称,从略),我们考虑两种情况,恰好对应图 5.12(a)中的节点 4 和 3,它们都是因为 2 的插入变成左失衡状态,但情况不同。

(1) 若 X 失衡的原因是因为其左子树的左子树上插入了一个节点,此时 X 对应于图 5.12(a)中的节点 4,则令 X 的左儿子(Y)为根,令 X 为 Y 的右儿子,同时令 Y 原先的右子树为 X 的左子树。这个操作称为“右旋”。图 5.12(a)右旋后即得到图 5.12(c)。

(2) 若 X 失衡的原因是其左子树的右子树上插入了一个节点,此时 X 对应于图 5.12(a) 中的节点 3,那就先以左儿子(Y)为轴做一个“左旋”,让它的右儿子“上位”(成为这棵子树的根节点,即占据原来节点 3 的位置),接着再以 X 为轴做一个右旋。图 5.12(a)据此操作后即得到图 5.12(b)。

按照这种方法得到的二叉搜索树(平衡二叉树)称为“AVL 树”^①,也是最早(1962 年)被发明出来的一种平衡二叉树。由于高效查找操作在计算机应用中的普遍性和重要性,人们还发明了其他一些平衡二叉树。它们的共同特点是树高为 $O(\log_2 n)$ 。^②由于这条性质的保证,上述的极端情况不再出现,查找数据的最坏复杂度仍然保持为 $\log_2 n$ 。

项目实践

按照本章的项目要求,分组讨论实现方案,可自行考虑用什么数据结构(线性表,二叉树,哈希表等)来组织 n 个元素以及相关的替换算法,针对所给出的三项任务,完成项目报告及程序实现。做验收评比的时候,可通过调整 n 、 m 、 N ,以及输入序列中元素的分布,观察程序性能对不同数据集的响应情况。

作业练习

- 1 判断以下说法的对错:
 - () 顺序查找时,被查找的数据必须有序;
 - () 对分查找时,被查找的数据不一定有序;
 - () 顺序查找总能找到要查找的关键字;
 - () 一般情况下,对分查找的效率较高。
- 2 设计一个用单链表作存储结构的顺序查找算法。
- 3 现有一个包含 7 个元素的序列 A,元素依次为 68、149、273、321、475、591、666。若采用对分查找法在该数组中查找数据 666,需要查找几次?
- 4 为数列 10、9、8、4、11、5、3、2、6、7 按照给出的顺序建立一棵二叉搜索树,然后试图查找 1、7、11、2 等数字,并给出这些数字的查找次数。

① AVL 由两位发明人 Georgy Adelson-Velsky 和 Evgenii Landis 的姓氏首字母组合而成。

② 这一点并不显然,但可以证明。

附录 参考程序

程序 3.1: 顺序表实现数据流检测

```
import numpy as np
datastream = [2,3,6,1,3,8,1,2,4,5]
print ('data stream:',datastream)
linear_list = np.zeros((100))
length = 0
while datastream:
    x = datastream.pop(0); no = True
    for i in range(length):
        if x == linear_list[i]:
            print (x,'Found it !')
            no = False; break
        if x > linear_list[i]:
            for j in range(length,i,-1):
                linear_list[j] = linear_list[j-1]
            linear_list[i] = x; length = length + 1
            no = False; break
    if no:
        linear_list[length] = x; length = length + 1
print ('final list:',linear_list[range(length)])
exit()
```

补充说明：

程序中,datastream 是测试数据,可由其他数据任意替换。Linear_list 是一个一维数组,其中的 100 是元素个数,也可以根据需要改变。插入操作发生在 12~15 行,特别值得关注的是 13~14 行的 for 循环,这是顺序表效率损失所在。

程序 3.2：链表实现数据流检测

```
import numpy as np
memory = np.zeros((100,2),dtype=np.int32)
datastream = [2,3,6,1,3,8,1,2,4,5]
print ('data stream:',datastream)
x = datastream.pop(0)
memory[0,0] = x; memory[0,1] = -1
linked_list_head = 0; new = 1
while datastream:
    x = datastream.pop(0); no = True
    search_ptr = linked_list_head; pre_node = -1
    while search_ptr != -1:
        if x == memory[search_ptr,0]:
            print (x,'Found it !')
            no = False; break
        if x > memory[search_ptr,0]:
            memory[new,0] = x
            memory[new,1] = search_ptr
            if search_ptr == linked_list_head:
                linked_list_head = new
            else:
                memory[pre_node,1] = new
            new = new + 1
            no = False; break
        else:
            pre_node = search_ptr
            search_ptr = memory[search_ptr,1]
    if no:
        memory[new,0] = x; memory[new,1] = -1
```

```

memory[pre_node,1] = new
new = new + 1
while linked_list_head != -1:
    print (memory[linked_list_head,0]),
    linked_list_head = memory[linked_list_head,1]
exit()

```

补充说明：

在课文中以及图 3.1 中,都没有提到程序中的变量 `pre_node`,它的作用是伴随 `search_ptr`,保证插入(以及添加表尾)操作的正确完成。从程序中可以看到,当在第 17 行发现一个较小的元素(`if` 语句条件成立)时,做插入操作(第 17~23 行),只涉及几个指针的调整,而不再有程序 3.1 中对应处的 `for` 循环。

程序 3.3：符号匹配

```

expr="(6+(2+4)*(3/(1+2)-(3+2)))" #确定表达式
Py_stack=list() #创建一个栈
for c in expr: #遍历expr中的字符
    if c=="(":
        Py_stack.append(c) #将左括号入栈
    if c==")":
        if not Py_stack: #如果是空栈
            print("缺左括号")
            break #循环提前终止,不执行循环后的else语句块
        else:
            Py_stack.pop() #出栈
    else: #循环正常终止,执行循环后的else语句块
        if not Py_stack: #如果是空栈
            print("匹配")
        else:
            print("缺右括号")

```

程序 3.4：基于数组的队列实现（包括实现队列的函数和测试调用主程序）

```
import numpy as np
n = 4
q = np.zeros((n),dtype=np.int32)
head = 123; tail = 456; counter = 0
def queue(op,para):
    global q, head, tail, counter
    if op == 'out':
        if counter == 0:
            return('empty!')
        else:
            para = q[head]
            head = (head+1)%n
            counter = counter - 1
            return(para)
    elif op == 'in':
        if counter == n:
            print('full!'),
            return()
        elif counter == 0:
            head = 0
            tail = 0
        else:
            tail = (tail+1)%n
        q[tail] = para
        counter = counter + 1
        return()
    else:
        print( 'invalid operator')
    return()
```

```

x = 'x'
h = 0
test = [1,2,x,4,x,x,x,2,x,x,3,5,1,x,3,5,7,x]
while test:
    e = test.pop(0)
    if e == x:
        print( queue('out',h)),
    else:
        queue('in',e)
print( "")
while counter:
    print( q[head]),
    head = (head+1)%n
    counter = counter - 1
exit()

```

注：观察该段代码的主程序部分(从第 30 到第 44 行)，我们看到了数据结构的实现和应用之间的“隔离”。主程序只管按照规定的接口做调用，而不用管该数据结构是如何实现的。也就是说，我们完全可以替换一个不同的 `def queue(op, para)` 代码，其中队列的实现采用链表方式，而不影响该主程序的执行。这是与第三章第一节中的线性表实现不同的。

程序 4.1：根据集合建立二叉树

```

import numpy as np
memory=np.zeros((100,3),dtype=np.int32) #100行3列数组
expre="{a,{b,{c,{},{d,{},{}}},{},{e,{f,{},{},{g,{},{}}}}}"
change=list()
stack = list()
p = 0

```



```

for i in range(len(expre) - 1):
    pos = expre[i]          #读取当前字符
    nex = expre[i + 1]      #读取下一个字符
    if pos == "{":          #当前字符为{
        if nex != "}":
            memory[p][0] = p    #生成节点
            stack.append(memory[p]) #节点入栈
            p = p + 1          #指针位置下移
            change.append(nex)
        else:
            if stack[-1][1] == 0:
                stack[-1][1] = -1    #将-1赋值给栈顶元素的左链
            else:
                stack[-1][2] = -1    #将-1赋值给栈顶元素的右链
    elif pos == "}":        #当前字符为}
        pre = expre[i - 1]    #读取前一个字符
        if pre != "{":
            aa = stack.pop()    #出栈一个节点元素
            if stack[-1][1] == 0:
                stack[-1][1] = aa[0] #将出栈元素地址给父节点左链
            else:
                stack[-1][2] = aa[0] #将出栈元素地址给父节点右链
for k in range(p):
    print(change[k],memory[k][1],memory[k][2])

```

程序 4.2：表达式二叉树建立和求值

```

class TreeNode:
    def __init__(self,data,left=None,right=None):
        self.data=data
        self.left=left
        self.right=right
def CreateTree(expre):

```

```

root=None
for c in expre:
    node = TreeNode(c)
    if root == None:
        root = node
    elif root.data.isdigit(): #如果根节点是数字
        node.left = root
        root = node
    elif node.data.isdigit(): #如果当前节点是数字
        tempNode = root #当根节点沿右路插入最右边成为右儿子
        while (tempNode.right != None):
            tempNode = tempNode.right
        tempNode.right = node
    else:
        if (GetPriority(node.data) <= GetPriority(root.data)):
            node.left = root
            root = node
        else:
            node.left = root.right
            root.right = node
return root
def calculate(root): #后序遍历函数求值
    if root==None:
        return
    else:
        calculate(root.left)
        calculate(root.right)
        print(root.data, end=" ")
        if root.data!="+" and root.data!="-" and root.data!="*" and root.data!="/" and root.data!="(":
            stackcal.append(int(root.data))
        else:
            x=stackcal.pop()
            y=stackcal.pop()
            result=0
            if root.data=="+":
                result=y+x
            elif root.data=="-":

```

```

        result=y-x
    elif root.data=="*":
        result=y*x
    elif root.data=="/":
        result=y/x
    stackcal.append(result)
def GetPriority(c):
    if c in ["+", "-"]:
        return 0
    if c in ["*", "/"]:
        return 1
if __name__=="__main__":    #主程序段
    expression="1+2*6-9/3"
    tree=CreateTree(expression)
    stackcal=list()
    calculate(tree)
    print(stackcal)

```

程序 4.3：广度优先搜索

```

import queue
a,b,c,d,e,f,g,h=range(8)    #对节点进行从0到7编号
edge=[[a,b],[a,c],[b,f],[b,h],[c,d],[c,e],[c,h],[d,e],[e,g],[f,g],[g,h]]    #边的集合
G=[]
t=list()
for v in range(8):          #将边集合转成邻接表
    for x in edge:
        if v in x:
            t=list(set(t+x))
    t.remove(v)
    G.append(t)
    t=list()

```

```

s=1;t=4          #设定起点和终点
checked = set()   #保存访问过的顶点
checked.add(s)
queue = queue.Queue()
queue.put((s, 0)) #把起始顶点和层次 0 入队
while not queue.empty():
    v, k = queue.get() #出队一个顶点和层次
    if v == t:
        print("经过的距离为: ", k)
        break
    for w in G[v]:
        if w not in checked:
            checked.add(w) #添加到已访问过的集合中
            queue.put((w, k + 1)) #顶点和层次入队

```

程序 4.4：建立哈希表

```

class Node:
    def __init__(self, key):
        self.key = key
        self.next = None
class CreateTable:
    def __init__(self, indexbox):
        self.indextable = [Node] * indexbox
        for i in range(indexbox):
            self.indextable[i] = Node(-1)
    def add(self, key): #添加数据元素，建立哈希表子程序
        newnode = Node(key)
        myhash = key % 13
        current = self.indextable[myhash]
        if current.next == None:
            self.indextable[myhash].next = newnode

```

```

else:
    while current.next!=None:
        current=current.next
    current.next=newnode
def get(self,key):
    i=0
    myhash=key%13
    ptr=self.indextable[myhash].next
    while ptr!=None:
        i=i+1
        if ptr.key==key:
            return "找了" + str(i) + "次"
        else:
            ptr=ptr.next
    return "未找到！"
def remove(self,key):
    i=0
    myhash=key%13
    pre=self.indextable[myhash]
    ptr = self.indextable[myhash].next
    while ptr!=None:
        i=i+1
        if ptr.key==key:
            if ptr.next==None:
                pre.next=None
            else:
                pre.next=ptr.next
            return "找了" + str(i) + "次"
        else:
            pre = ptr
            ptr=ptr.next
    return "未找到！"
def show_table(self,sum):
    for i in range(sum):
        head=self.indextable[i].next
        print(' %2d:\t' %i,end="")
        while head!=None:

```

```

        print("[%2d]-' %head.key,end='')
        head=head.next
    print()
if __name__=="__main__":    #主程序段
    indexbox=13              #哈希表长度
    data=[19, 14, 23, 1, 68, 20, 27, 55, 11, 9]
    ha=CreateTable(indexbox)
    for i in data:
        ha.add(i)
    print(ha.get(23))
    ha.show_table(indexbox)

```

程序 5.1: 堆排序

```

def siftdown(lst, i, size):
    lchild = 2 * i + 1    #确定i的左儿子的位置
    rchild = 2 * i + 2    #确定i的右儿子的位置
    max = i
    if i < size // 2:      #判断i不是叶节点
        if lchild < size and lst[lchild] > lst[max]: #判断左儿子最大
            max = lchild                                #记录左儿子位置
        if rchild < size and lst[rchild] > lst[max]: #判断右儿子最大
            max = rchild                                #记录右儿子位置
        if max != i:                                     #判断最大的不是i
            lst[max], lst[i] = lst[i], lst[max]
            siftdown(lst, max, size)    #递归向下
def build_heap(lst):      #初始建堆
    size = len(lst)
    for i in range((size//2)-1, -1, -1): #最后一个起对非叶节点进行向下筛选
        siftdown(lst, i, size)
def chooseandrebuild(lst, i):    #选择并重建堆

```



```

lst[0], lst[i] = lst[i], lst[0]      #交换堆顶与堆中最后一个元素
siftdown(lst, 0, i)                 #对新二叉树的根进行向下筛选
def heap_sort(lst):                  #主程序,对lst进行堆排序
    size = len(lst)
    build_heap(lst)                  #初始建堆
    for i in range(size-1, -1, -1):  #从后向前依次选择和重建堆
        chooseandbuild(lst, i)
if __name__ == "__main__":          #主程序段
    lst = [23,7,90,65,98,11,46,13,14,6,7,11,22]
    heap_sort(lst)
    print(lst)

```

程序 5.2：二叉搜索树

```

class TreeNode: #首先为树的节点定义一个类:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
class BSTree: #然后为二叉搜索树定义一个类:
    def __init__(self):
        self.root = None
    def search(self, key): #查找程序:
        cur = self.root
        while cur:
            if key == cur.data:
                return cur
            elif key > cur.data:
                cur = cur.right
            else:
                cur = cur.left
        return None

```

```
def insert(self, key): # 插入程序:
```

```
    new = TreeNode(key)
```

```
    if not self.root:
```

```
        self.root = new
```

```
    return
```

```
    cur = self.root
```

```
    cpar = None
```

```
    while cur:
```

```
        if cur.data == key:
```

```
            return
```

```
        elif cur.data < key:
```

```
            cpar = cur
```

```
            cur = cur.right
```

```
        else:
```

```
            cpar = cur
```

```
            cur = cur.left
```

```
    if cpar.data > key:
```

```
        cpar.left = new
```

```
    else:
```

```
        cpar.right = new
```

```
def delete(self, key): # 删除程序:
```

```
    cur, cpar = self.root, None
```

```
    while cur and key != cur.data:
```

```
        cpar = cur
```

```
        if key < cur.data:
```

```
            cur = cur.left
```

```
        else:
```

```
            cur = cur.right
```

```
    if not cur:
```

```
        return
```

```
    if cur.left is None:
```

```
        if cpar is None:
```

```
            self.root = cur.right
```

```
        elif cur == cpar.left:
```

```
            cpar.left = cur.right
```

```
        elif cur == cpar.right:
```

```
            cpar.right = cur.right
```

```
    else:
```

```

    mostright = cur.left
    while mostright.right:
        mostright = mostright.right
    mostright.right = cur.right
    if cpar is None:
        self.root = cur.left
    elif cpar.left == cur:
        cpar.left = cur.left
    else:
        cpar.right = cur.left

def InTvs(root):          #递归实现中序遍历
    if root is None:
        return
    else:
        InTvs(root.left)
        print(root.data, end=" ")
        InTvs(root.right)

if __name__ == "__main__":    #主程序段
    Py_tree=BSTree()          #建立一棵空的二叉搜索树
    data=[10,5,15,1,6,8,12,2,7,9]
    for i in data:
        Py_tree.insert(i)      #将数据依次插入建立二叉搜索树
    InTvs(Py_tree.root)         #中序遍历显示二叉搜索树

```

后 记

本册教科书依据教育部《普通高中信息技术课程标准(2017 年版 2020 年修订)》编写,并经国家教材委员会专家委员会审核通过。全体编写人员认真领会国家基础教育改革精神,精心研究当代信息社会的人才培养要求,广泛调研上海及各地高中信息技术教育的现状和挑战,深入了解高中学生的学习需求,并汲取了上海市《普通高中信息科技(试用本)》的编写经验。

编写过程中,上海市中小学(幼儿园)课程改革委员会专家工作委员会,上海市教育委员会教学研究室,上海市课程方案教育教学研究基地、上海市心理教育教学研究基地、上海市基础教育教材建设研究基地、上海市信息科技教育教学研究基地(上海高校“立德树人”人文社会科学重点研究基地)及基地所在单位华东师范大学等单位给予了大力支持,在此表示感谢!

本册教科书出版之前,我们已通过多种渠道与教科书选用作品(包括照片、画作)的作者进行了联系,得到了他们的大力支持。对此,我们衷心地表示感谢! 恳请尚未联系到的作者与我们联系,以便出版社及时支付相关稿酬。

我们真诚地希望广大教师、学生及家长在使用本册教科书的过程中提出宝贵意见。我们将集思广益,不断修订,使教科书趋于完善。

编 者



致力于用榜样的力量提升学生成绩的共享家教平台

中国家庭教育学会荣誉会员单位

985/211 大学生 1对1 上门辅导

找家教就像叫“代驾”一样简单
家长们都在偷偷用的家教预约神器

记得拍照留存哦



扫码关注 预约上门

关注送200元优惠券

小初高全科辅导

学霸云集任您挑

学历真实可担保



与优秀大学生同行，激发孩子无限潜能



微信搜索公众号：365优教网

咨询热线：4000-711-365

YOUJ 优教

既是找老师，更是找榜样

家教老师全国招募中