

2019年 数据结构 考研复习指导

王道论坛·组编



微信: wangdaoluntan

关注王道公众号:

- ✓ 在线交流, 实时发布勘误
- ✓ 获取 C 语言基础讲课视频
- ✓ 获取本书配套辅导及习题讲解视频
- ✓ 了解程序员的学习与发展
- ✓ 了解王道训练营的开班情况

王道考研系列

2019 年数据结构 考研复习指导

Review Guide of Data Structures Entrance Exam

王道论坛 组编

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书是计算机专业研究生入学考试“数据结构”课程的复习用书,内容包括绪论、线性表、栈和队列、树与二叉树、图、查找、排序等。全书严格按照最新计算机统考大纲的数据结构部分,对大纲所涉及的知识点进行集中梳理,力求内容精炼、重点突出、深入浅出。本书精选各名校的历年考研真题,并给出详细的解题思路,力求实现讲练结合、灵活掌握、举一反三的功效。通过本书“书本+在线”的学习方式和网上答疑方式,读者在使用本书时遇到的任何疑难点,都可以在王道论坛发帖与道友互动。

本书可作为考生参加计算机专业研究生入学考试的备考复习用书,也可作为计算机专业的学生学习数据结构课程的辅导用书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

2019 年数据结构考研复习指导 / 王道论坛组编. —北京: 电子工业出版社, 2018.4

ISBN 978-7-121-33708-6

I. ①2… II. ①王… III. ①数据结构—研究生—入学考试—自学参考资料 IV. ①TP311.12

中国版本图书馆 CIP 数据核字 (2018) 第 030442 号

策划编辑: 谭海平

责任编辑: 裴 杰

印 刷: 保定市画美凯印刷有限公司

装 订: 保定市画美凯印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1 092 1/16 印张: 21.5 字数: 632.9 千字

版 次: 2018 年 4 月第 1 版

印 次: 2018 年 4 月第 1 次印刷

定 价: 49.50 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: (010) 88254386。

本书编委会

总 主 编:

赵 霖

副 主 编:

罗 乐 李福龙 夏俭磊 王 彪 甘海波 邢 勇 刘向阳
刘 莹 蔡飞飞

编 委:

蔡飞飞	曹鑫瑞	陈佳楠	陈 骞	陈敬坤	陈明惠	陈仕理
陈益康	陈振高	单飞燕	丁 月	杜 祎	樊庆艳	樊绪新
樊志洪	甘海波	高艾华	高 成	高 峰	官水旺	郭春阳
郭 丹	郭慧丰	韩 鼎	何成伟	何建荣	何 荧	胡关虎
胡 凯	胡宇成	黄 敏	黄冉东	黄松林	黄 伟	姬玉柱
贾欣欣	兰善伟	李福龙	李 好	李昊迪	李鹤群	李红军
李 康	李 敏	李维娜	李潇奕	李玉琴	李芝峰	刘兵兵
刘 聪	刘 飞	刘红雨	刘 立	刘 浏	刘 蒙	刘鹏程
刘 胜	刘晟源	刘 伟	刘向阳	刘雪飞	刘亚男	刘 岩
刘 莹	刘正学	陆晓宇	罗立志	骆 彬	律岩杰	马金晶
马林浩	马原龙	孟 倩	欧 勇	彭先强	彭晓娟	乔 晶
邱 从	全文令	邵 云	沈学东	石 帅	史魁洋	史鹏宙
宋 欢	宋景凯	宋双志	谭 健	汤晓静	唐 溧	汪道亮
汪 潭	王宝童	王 彪	王德强	王 法	王怀宇	王 浪
王鹏程	王 琦	王晓政	王雨婷	王中义	卫 健	吴福怀
吴海波	吴丽林	吴 楠	夏俭磊	相 洋	肖 隆	肖任远
邢 超	邢林林	邢 勇	杨 灿	杨 培	杨睿琦	姚超金
姚泓斌	姚 佳	余婷婷	余 勇	袁耀强	曾吉斌	曾全站
张俊杰	张敏仪	张 帅	张显君	张晓函	赵 冬	赵 军
赵 康	赵 霖	赵淑芳	赵淑芬	赵思成	赵文博	赵一嘉
郑昌华	郑德原	郑 文	郑肖雄	周广露	周 羽	朱大龙
朱晓宁	邹亚劼					

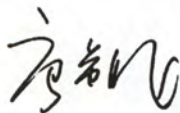
序 言

当前,随着我国经济和科技高速发展,特别是计算机科学突飞猛进的发展,对计算机相关人才,尤其是中高端人才的需求也将不断增长。硕士研究生入学考试可视为人生的第二次大考试,它是改变命运、实现自我理想的又一次机会,而计算机专业一直是高校考研的热门专业之一。

自计算机专业研究生入学考试实行统一命题以来,初试科目包含了最重要的四门基础课程(数据结构、计算机组成原理、操作系统、计算机网络),很多学生普遍反映找不到方向,复习也无从下手。倘若有一本能够指导考生如何复习的好书,必将对考生的帮助匪浅。我的学生风华他们策划和编写了这一系列的计算机专业考研辅导书,重点突出,层次分明。他们结合了自身的复习经验、理解深度以及对大纲把握程度的体会,对考生而言是很有启发和指导意义的。

计算机这门学科,任何机械式的死记硬背都是收效甚微的。在全面深入复习之后,首先对诸多知识点分清主次,并结合做题,灵活运用所掌握的知识点,再选择一些高质量的模拟试题来检测自己理解和掌握的程度,查漏补缺。这符合我执教 40 余年来一直坚持“教材—习题集—试题库”的教学体系。

从风华他们策划并组建编写团队到初稿成形,直至最后定稿,我能体会到风华和他的团队确实倾注了大量的精力。这套书的出版一定会受到广大考研学生的欢迎,它会使你在考研的路上得到强有力的帮助。



前言

由王道论坛 (www.cskaoyan.com) 组织名校状元级选手, 编写了 4 本单科辅导书。单科书是基于王道指导全书的二代作品, 不论是编排方式, 还是内容质量都较前一版本的王道书有了较大的提升。这套书也参考了同类优秀的教材和辅导书, 更是结合了高分选手们自己的复习经验。无论是对考点的讲解, 还是对习题的选择和解析, 都结合了他们对专业课复习的独特见解。2018 年, 我们继续推出“王道考研系列”单科书, 一共 4 本:

- 《2019 年数据结构考研复习指导》
- 《2019 年操作系统考研复习指导》
- 《2019 年计算机组成原理考研复习指导》
- 《2019 年计算机网络考研复习指导》

每一版, 我们不仅会修正之前发现的全部错误, 还会对考点讲解做出尽可能的优化, 也重新审视读者对本书的反馈, 针对大家提出的疑问和建议对本书做出针对性的优化; 此外还重新筛选了部分习题, 尤其是对习题的解析做出了更好的改进。

当然, 深入掌握专业课内容没有捷径, 考生也不应怀有任何侥幸心理, 扎扎实实打好基础、踏踏实实做题巩固, 最后灵活致用才是高分的保障。我们只希望这套书能够指导大家复习考研, 但学习还是得靠自己, 高分不是建立在任何空中楼阁之上的。对于一个想继续在计算机专业领域深造的考生来说, 认真学习并扎实掌握这 4 门计算机专业中最基础的专业课, 是最基本的前提。

“王道考研系列”是计算机考生口碑相传的辅导书, 自出版以来在同类书中的销量始终遥遥领先。有这么多的成功学长, 我相信只要考生合理地利用好本书、并采用科学的复习方法, 一定会收获属于自己的那份回报。

读者在复习时遇到的疑问, 可以通过读者 QQ 群 (群号: 346165686) 交流。从 2019 版起, 我们也将陆续为本书制作配套辅导及习题讲解视频, 请读者关注王道论坛公众号 (微信号: wangdaoluntan), 里面会及时推送勘误及配套视频的相关情况。

“不打广告、不发证书、不包就业, 专注于培养有态度、有能力的高级码农”王道程序员训练营是王道团队举办的线下高强度编程集训。打下扎实的编程和算法基本功, 培养程序员式的学习能力和学习方法, 期待有梦想、有追求的你加入!

予人玫瑰, 手有余香, 王道论坛伴你一路同行!

风华漫舞

致 读 者

——王道单科使用方法的道友建议

我是二战考生，第一次考研成绩 333 分（专业代码 408，成绩 81 分），痛定思痛后决心再战。潜心复习了半年后终于以 392 分（专业代码 408，成绩 124 分）考入上海交通大学计算机系，这半年里我的专业课成绩提高了 43 分，成了提分主力。从不达线到比较满意的成绩；从闷头乱撞到有了自己明确的复习思路，我想这也是为什么风华哥从诸多高分选手中选我给大家介绍经验的一个原因吧。

整个专业课的复习是围绕王道材料展开的，从一遍、两遍、三遍看单科书的积累提升，到做 8 套模拟题时的强化巩固，再到看思路分析时的醍醐灌顶。王道书能两次押中原题固然有运气成分，但这也从侧面说明他们的编写思路和选题方向与真题很接近。

下面说说我的具体复习过程：

每天划给专业课的时间是 3~4 小时。第一遍细看课本，看完一章做一章单科书（红笔标注错题），这一遍共持续 2 个月。第二遍主攻单科书（红笔标注重难点），辅看课本。第二遍看单科书和课本的速度快了很多，但感觉收获更多，常有温故知新的感觉，理解更深刻（风华注，建议这里再速看第三遍，特别针对错题和重难点。模拟题完后再跳看第四遍）。

以上是打基础阶段，注意单科书和课本我仔细精读了两遍，弄懂每个知识点和习题。大概 11 月上旬开始做模拟题和思路分析，期间遇到不熟悉的地方不断回头查阅单科书和课本。8 套模拟题的考点覆盖得很全面，所以大家做题时如果忘记了某个知识点，千万不要慌张，赶紧回去看这个知识盲点，最后的模拟就是查漏补缺。模拟题一定要严格按考试时间去做（14:00~17:00），注意应试技巧，做完试题后再回头研究错题。算法题的最优解法不太好想，如果实在没思路，建议直接“暴力”解决，结果正确也能有 10 分，总比苦拼出 15 分来而将后面比较好拿分的题耽误了好（这是我第一年的切身教训！）。最后剩了几天看标注的错题，第三遍跳看单科书，考前一夜浏览完网络，踏实地睡着了……

考完专业课，走出考场终于长舒一口气，考试情况也胸中有数。回想这半年的复习，耐住了寂寞和诱惑，雨雪风霜从未间断跑去自习，考研这人生一站终归没有辜负我的用心良苦。佛教徒说世间万物生来平等，都要落入春华秋实的代谢中去，辩证唯物主义认为事物作为过程存在，凡是存在的终归要结束，你不去为活得多姿多彩拼搏，真到了和青春说再见时你是否会可惜虚枉了青春？风华哥说过我们都是“屌丝”，我们正在逆袭，你呢？

感谢风华大哥的信任，给我这个机会分享专业课复习经验给大家，作为一个铁杆道友在王道受益匪浅，也借此机会回报王道论坛。祝大家金榜题名！

ccg1990@SJTU

王道程序员训练营

经常有人问我们：“为什么不做考研培训？这个市场很大”？

这里，算作一个简短的回答吧。王道尊重的不是考研这个行当，而是考研学生的精神，他们的梦想，仅此而已。考研可能是部分 CS 学生实现梦想的阶段，但考研学习的内容，对 CSer 的职业生涯毕竟没有太多的帮助和意义。对于计算机专业的学生，编程基本功和学习能力才是受用终生的资本，决定了未来在技术道路上能走多远。而王道团队也只会专注于计算机这个领域，从名校考研、到编程集训。

从 2008 年初创办至今，王道团队，经历了从本科到考研成功，从硕士到社会历练，积累了不少经验，但也走过不少弯路，而希望大家也尽量少走我们所走过的弯路。

计算机是一个靠能力吃饭的专业。和很多现在的你们一样，当年的我们也经历过本科时的迷茫，而无非是自觉能力太弱，以致底气不足。学历只是敲门砖，同样是名校硕士，有人走上正确的方向，如鱼得水，成为 Offer 帝；有人却始终难入“编程与算法之门”，始终与好 Offer 无缘，再一次体会就业之痛，最后只能“将就”签约。即便是名校硕士，Offer 成色也是三六九等。考研高分 \neq Offer 高薪，我们也更欣赏技术上的牛人。

考研结束后的日子，或许是一段难得的提升编程能力的连续完整时光，趁着还有时间，也该去弥补本科期间应掌握的能力，也是追赶与那些大牛们的差距的时候了。

把参加训练营看做是一次对自己的投资，投资自身、投资未来才是最好的投资。

你将从王道训练营获得

编程能力的迅速提升，结合项目实战，逐步打下坚实的编程基础，培养积极、主动的学习能力。动手编程为驱动的教学模式，解决你在编程、算法思维上的不足。也是为未来的深入学习提供方向指导，掌握编程的学习方法，引导进入“编程与算法之门”。

道友们在训练营里从菜鸟逐步成长，训练营中目前已有不少研究生道友陆续拿到百度、腾讯、阿里、微博、网易等一线互联网公司的 Offer。这就是竞争力！

正如八期道友孟亮所言：“来了你就发现，这里无关程序员以外的任何东西，这是一个过程，一个对自己认真，对自己负责的过程”。

.....

王道训练营的优势

这里都是王道道友，他们信任王道，乐于分享与交流，纯粹。

因为都是忠实的王道道友，都曾经经历过考研……训练营的住宿、生活都在一起，其乐融融，很快大家也将成为互帮互助的好朋友！相互学习对方的优点。

本科+硕士的生源。考研也绝非人生唯一的出路，给自己换一条路走，去职场上好好发

展或许会更好。考上研究生也并不意味着高枕无忧，人生的道路还很漫长。

王道团队皆具有扎实的编程基本功，他们用自己的态度、思维去影响训练营的道友，尽可能引导他们走上正确的发展方向……是对道友信任的回报，也是一种责任！

王道训练营只是一个平台，网罗王道论坛上有梦想、有态度的 CS 屌丝。并为他们的梦想提供土壤和圈子。始终相信那句“物竞天择 适者生存”，这里的生存并不是简简单单的活下来，而是活得有价值、活得有态度！

王道训练营的参与条件

1. 面向就业

面临就业，但编程能力偏弱的计算机相关专业学生。

大学酱油模式渡过，投简历如石沉大海，好不容易有次面试机会，又由于基础薄弱、编程太少，以至于面试时有口无言，面试结果可想而知。开始偿债吧，再不抓住当下，未来或将持续迷茫，逝去了的青春是无法复返的，这个世界上后悔药是确定没有的。

眼光和视野放长远一点吧，在这个充分竞争的技术领域，当前的能力决定了你能找一份怎样的工作，踏实的态度和学习的能力决定了你未来能走多远。

王道集训营（C/C++或 JAVA 方向）的费用通常只有市面培训机构的一半左右，且费用四年未涨（业界良心），极大降低了参加道友的学习成本。

2. 面向硕士

提升能力，刚考上计算机相关专业的准研究生或在读研究生。

名校研究生已没有什么可以值得骄傲的资本，我们身边所看到的都是名校硕士。同为名校，为什么有人能轻松拿到百度、腾讯、阿里、微软等 Offer，年薪 15~30 万元人民币，发展前景甚好；有人却只能拿 8~12 万元人民币年薪的 Offer，在房价/物价高飞的年代，这点收入也只能月光吧。家中父母可能因有名校硕士的孩子而骄傲，可不知孩子其实在外面过得很辛苦。

来王道训练营的一些要求

- 王道是开放式网络（同类机构没有谁会开放网络的），有利有弊，我们是希望培养会积极主动学习的人，着重培养他们独立解决问题的能力，需擅于利用网络。

- 保持内在的激情和踏实态度，不需要“打鸡血”。第一次打鸡血能坚持 1 个月、第二次能坚持半个月、第三次只能坚持一周，但试想读研或工作后谁会给你“打鸡血”。

- 不要总是期待老师灌输得更多，应逐步摆脱对老师的依赖，培养积极主动的钻研能力、独立解决问题的能力，到后期更应主动去钻研、主动去解决问题。

- 随遇而安，不轻浮，保持虚心和踏实的态度，多独立思考，也要多交流。

- 坚持做笔记，多向身边优秀的道友学习，少说多做，沉浸在代码的世界中。

最后，我们并不太看重眼前的基础。始终相信：眼前哪怕基础弱一些，但只要踏踏实实努力做好，短期内也是完全可以追赶的。这也是往期集训营道友的切身经验。

目 录

第1章 绪论	1
1.1 数据结构的基本概念	1
1.1.1 基本概念和术语	1
1.1.2 数据结构的三要素	2
1.1.3 本节试题精选	3
1.1.4 答案与解析	4
1.2 算法和算法评价	5
1.2.1 算法的基本概念	5
1.2.2 算法效率的度量	5
1.2.3 本节试题精选	6
1.2.4 答案与解析	9
归纳总结	10
思维拓展	11
第2章 线性表	12
2.1 线性表的定义和基本操作	12
2.1.1 线性表的定义	12
2.1.2 线性表的基本操作	13
2.1.3 本节试题精选	13
2.1.4 答案与解析	13
2.2 线性表的顺序表示	14
2.2.1 顺序表的定义	14
2.2.2 顺序表上基本操作的实现	15
2.2.3 本节试题精选	17
2.2.4 答案与解析	19
2.3 线性表的链式表示	26
2.3.1 单链表的定义	26
2.3.2 单链表上基本操作的实现	27
2.3.3 双链表	30
2.3.4 循环链表	32
2.3.5 静态链表	32
2.3.6 顺序表和链表的比较	33

2.3.7	本节试题精选	34
2.3.8	答案与解析	39
	归纳总结	56
	思维拓展	57
第3章	栈和队列	58
3.1	栈	58
3.1.1	栈的基本概念	58
3.1.2	栈的顺序存储结构	59
3.1.3	栈的链式存储结构	61
3.1.4	本节试题精选	61
3.1.5	答案与解析	64
3.2	队列	71
3.2.1	队列的基本概念	71
3.2.2	队列的顺序存储结构	71
3.2.3	队列的链式存储结构	74
3.2.4	双端队列	75
3.2.5	本节试题精选	77
3.2.6	答案与解析	79
3.3	栈和队列的应用	82
3.3.1	栈在括号匹配中的应用	83
3.3.2	栈在表达式求值中的应用	83
3.3.3	栈在递归中的应用	84
3.3.4	队列在层次遍历中的应用	85
3.3.5	队列在计算机系统中的应用	86
3.3.6	本节试题精选	86
3.3.7	答案与解析	88
3.4	特殊矩阵的压缩存储	93
3.4.1	数组的定义	94
3.4.2	数组的存储结构	94
3.4.3	矩阵的压缩存储	94
3.4.4	稀疏矩阵	96
3.4.5	本节试题精选	97
3.4.6	答案与解析	98
	归纳总结	99
	思维拓展	100
第4章	树与二叉树	101
4.1	树的基本概念	101
4.1.1	树的定义	101

4.1.2	基本术语	102
4.1.3	树的性质	103
4.1.4	本节试题精选	103
4.1.5	答案与解析	104
4.2	二叉树的概念	105
4.2.1	二叉树的定义及其主要特性	105
4.2.2	二叉树的存储结构	107
4.2.3	本节试题精选	108
4.2.4	答案与解析	110
4.3	二叉树的遍历和线索二叉树	114
4.3.1	二叉树的遍历	114
4.3.2	线索二叉树	117
4.3.3	本节试题精选	120
4.3.4	答案与解析	124
4.4	树、森林	142
4.4.1	树的存储结构	142
4.4.2	树、森林与二叉树的转换	143
4.4.3	树和森林的遍历	144
4.4.4	树的应用——并查集	145
4.4.5	本节试题精选	146
4.4.6	答案与解析	148
4.5	树与二叉树的应用	153
4.5.1	二叉排序树	153
4.5.2	平衡二叉树 (Balanced Binary Tree)	156
4.5.3	哈夫曼 (Huffman) 树和哈夫曼编码	159
4.5.4	本节试题精选	161
4.5.5	答案与解析	165
	归纳总结	174
	思维拓展	176
第5章	图	177
5.1	图的基本概念	178
5.1.1	图的定义	178
5.1.2	本节试题精选	181
5.1.3	答案与解析	182
5.2	图的存储及基本操作	184
5.2.1	邻接矩阵法	184
5.2.2	邻接表法	186
5.2.3	十字链表	187
5.2.4	邻接多重表	188

5.2.5	图的基本操作	189
5.2.6	本节试题精选	190
5.2.7	答案与解析	192
5.3	图的遍历	195
5.3.1	广度优先搜索 (Breadth-First-Search, BFS)	195
5.3.2	深度优先搜索 (Depth-First-Search, DFS)	197
5.3.3	图的遍历与图的连通性	199
5.3.4	本节试题精选	199
5.3.5	答案与解析	201
5.4	图的应用	206
5.4.1	最小生成树 (Minimum-Spanning-Tree, MST)	207
5.4.2	最短路径	209
5.4.3	拓扑排序	212
5.4.4	关键路径	213
5.4.5	本节试题精选	215
5.4.6	答案与解析	221
	归纳总结	232
	思维拓展	233
第6章	查找	234
6.1	查找的基本概念	234
6.2	顺序查找和折半查找	235
6.2.1	顺序查找	235
6.2.2	折半查找	237
6.2.3	分块查找	238
6.2.4	本节试题精选	239
6.2.5	答案与解析	242
6.3	B树和B+树	248
6.3.1	B树及其基本操作	248
6.3.2	B+树基本概念	251
6.3.3	本节试题精选	252
6.3.4	答案与解析	254
6.4	散列 (HASH) 表	257
6.4.1	散列表的基本概念	257
6.4.2	散列函数的构造方法	258
6.4.3	处理冲突的方法	259
6.4.4	散列查找及性能分析	260
6.4.5	本节试题精选	260
6.4.6	答案与解析	263
6.5	字符串模式匹配	268

6.5.1 简单的模式匹配算法	268
6.5.2 改进的模式匹配算法——KMP 算法	269
6.5.3 本节试题精选	273
6.5.4 答案与解析	274
归纳总结	280
思维拓展	281
第 7 章 排序	282
7.1 排序的基本概念	283
7.1.1 排序的定义	283
7.1.2 本节试题精选	283
7.1.3 答案与解析	284
7.2 插入排序	284
7.2.1 直接插入排序	284
7.2.2 折半插入排序	285
7.2.3 希尔排序	286
7.2.4 本节试题精选	287
7.2.5 答案与解析	288
7.3 交换排序	290
7.3.1 冒泡排序	290
7.3.2 快速排序	291
7.3.3 本节试题精选	293
7.3.4 答案与解析	295
7.4 选择排序	300
7.4.1 简单选择排序	300
7.4.2 堆排序	301
7.4.3 本节试题精选	303
7.4.4 答案与解析	305
7.5 归并排序和基数排序	308
7.5.1 归并排序	308
7.5.2 基数排序	310
7.5.3 本节试题精选	310
7.5.4 答案与解析	312
7.6 各种内部排序算法的比较及应用	314
7.6.1 内部排序算法的比较	314
7.6.2 内部排序算法的应用	315
7.6.3 本节试题精选	316
7.6.4 答案与解析	317
7.7 外部排序	320
7.7.1 外部排序的基本概念	320

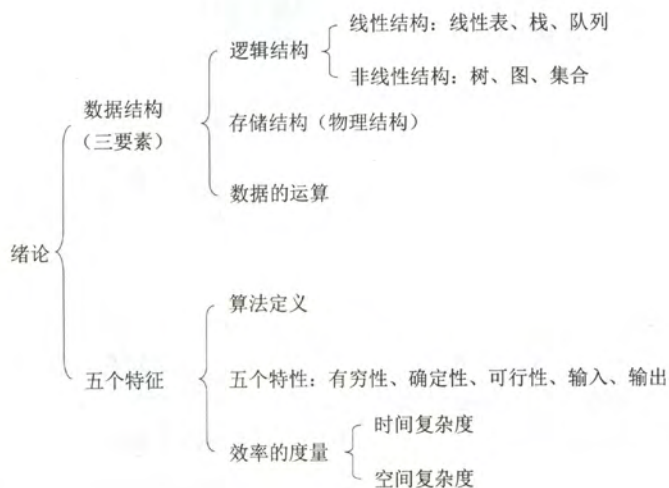
7.7.2 外部排序的方法·····	320
7.7.3 多路平衡归并与败者树·····	321
7.7.4 置换-选择排序(生成初始归并段)·····	322
7.7.5 最佳归并树·····	323
7.7.6 本节试题精选·····	324
7.7.7 答案与解析·····	325
归纳总结·····	328
思维拓展·····	329
参考文献·····	330

第1章 绪 论

【复习要点】

- (一) 数据结构相关的概念和术语
- (二) 数据结构的三要素：逻辑结构、物理结构和数据运算
- (三) 算法的时间复杂度和空间复杂度的分析与计算

【知识框架】



【复习提示】

本章内容并不在考研大纲中，它是数据结构的一个概述。但读者千万不要忽视本章，更应该通过对本章的学习初步了解数据结构的基本内容和基本方法。**分析算法的时间复杂度和空间复杂度是本章的重点**，属必考内容，一定要熟练掌握，每年都会结合算法设计题而出现。还多次直接以选择题的形式考查了时间复杂度的计算。掌握本章的基本概念，对后续章节的学习以及整个数据结构课程的理解是非常重要的。

1.1 数据结构的基本概念

1.1.1 基本概念和术语

1. 数据

数据是信息的载体，是描述客观事物属性的数、字符以及所有能输入到计算机中并被计算机程序识别和处理的符号的集合。

2. 数据元素

数据元素是数据的基本单位，通常作为一个整体进行考虑和处理。一个数据元素可由若干个数据项组成，数据项是构成数据元素的不可分割的最小单位。例如，学生记录就是一个数据元素，它由学号、姓名、性别等数据项组成。

注意：不要混淆数据、数据元素、数据项之间的概念，也要注意和数据库中的相关术语进行区别：如数据记录、数据字段等概念。

3. 数据对象

数据对象是具有相同性质的数据元素的集合，是数据的一个子集。例如，整数数据对象是集合 $N=\{0, \pm 1, \pm 2, \dots\}$ 。

4. 数据类型

数据类型是一个值的集合和定义在此集合上一组操作的总称。

- 1) 原子类型：其值不可再分的数据类型。
- 2) 结构类型：其值可以再分解为若干成分（分量）的数据类型。
- 3) 抽象数据类型：抽象数据组织和与之相关的操作。

5. 抽象数据类型

抽象数据类型（ADT）是指一个数学模型以及定义在该模型上的一组操作。抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关，即不论其内部结构如何变化，只要它的数学特性不变，都不影响其外部的使用。通常用（数据对象、数据关系、基本操作集）这样的三元组来表示抽象数据类型。

6. 数据结构

在任何问题中，数据元素都不是孤立存在的，而是在它们之间存在着某种关系，这种数据元素相互之间的关系称为结构（Structure）。数据结构是相互之间存在一种或多种特定关系的数据元素的集合。数据结构包括三方面的内容：逻辑结构、存储结构和数据的运算。数据的逻辑结构和存储结构是密不可分的两个方面，一个算法的设计取决于所选定的逻辑结构，而算法的实现依赖于所采用的存储结构^①。

1.1.2 数据结构的三要素

1. 数据的逻辑结构

逻辑结构是指数据元素之间的逻辑关系，即从逻辑关系上描述数据。它与数据的存储无关，是独立于计算机的。数据的逻辑结构分为线性结构和非线性结构，线性表是典型的线性结构；集合、树和图是典型的非线性结构。数据的逻辑结构分类见图 1-1。

集合 结构中的数据元素之间除了“同属于一个集合”的关系外，别无其他关系。

线性结构 结构中的数据元素之间只存在一对一的关系。

树形结构 结构中的数据元素之间存在一对多的关系。

图状结构或网状结构 结构中的数据元素之间存在多对多的关系。

^① 读者应通过后续章节的学习，逐步理解设计与实现的概念与区别。

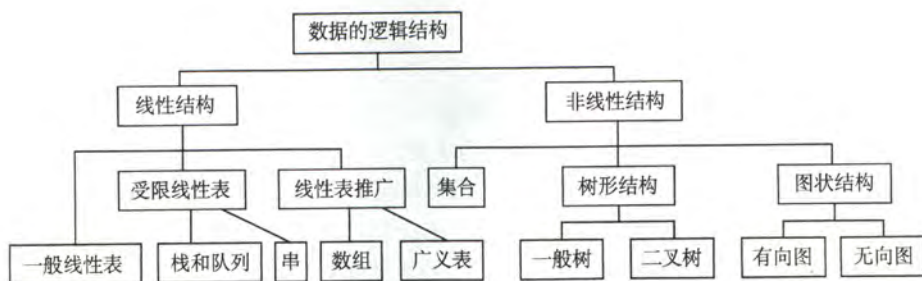


图 1-1 数据的逻辑结构分类图

2. 数据的存储结构

存储结构是指数据结构在计算机中的表示（又称映像），也称**物理结构**。它包括数据元素的表示和关系的表示。数据的存储结构是逻辑结构用计算机语言的实现，它依赖于计算机语言。数据的存储结构主要有：顺序存储、链式存储、索引存储和散列存储。

1) 顺序存储：把逻辑上相邻的元素存储在物理位置上也相邻的存储单元里，元素之间的关系由存储单元的邻接关系来体现。其优点是可实现随机存取，每个元素占用最少的存储空间；缺点是只能使用相邻的一整块存储单元，因此可能产生较多的外部碎片。

2) 链式存储：不要求逻辑上相邻的元素在物理位置上也相邻，借助指示元素存储地址的指针表示元素之间的逻辑关系。其优点是不会出现碎片现象，充分利用所有存储单元；缺点是每个元素因存储指针而占用额外的存储空间，并且只能实现顺序存取。

3) 索引存储：在存储元素信息的同时，还建立附加的索引表。索引表中的每一项称为索引项，索引项的一般形式是：（关键字，地址）。其优点是检索速度快；缺点是增加了附加的索引表，会占用较多的存储空间。另外，在增加和删除数据时要修改索引表，因而会花费较多的时间。

4) 散列存储：根据元素的关键字直接计算出该元素的存储地址，又称为 Hash 存储。其优点是检索、增加和删除结点的操作都很快；缺点是如果散列函数不好可能出现元素存储单元的冲突，而解决冲突会增加时间和空间开销。

3. 数据的运算

施加在数据上的运算包括运算的定义和实现。运算的定义是针对逻辑结构的，指出运算的功能；运算的实现是针对存储结构的，指出运算的具体操作步骤。

1.1.3 本节试题精选

一、单项选择题

1. 可以用（ ）定义一个完整的数据结构。
A. 数据元素 B. 数据对象 C. 数据关系 D. 抽象数据类型
2. 以下数据结构中，（ ）是非线性数据结构。
A. 树 B. 字符串 C. 队列 D. 栈
3. 以下属于逻辑结构的是（ ）。
A. 顺序表 B. 哈希表 C. 有序表 D. 单链表
4. 以下与数据的存储结构无关的术语是（ ）。
A. 循环队列 B. 链表 C. 哈希表 D. 栈
5. 以下关于数据结构的说法中，正确的是（ ）。

- A. 数据的逻辑结构独立于其存储结构
 - B. 数据的存储结构独立于其逻辑结构
 - C. 数据的逻辑结构唯一决定了其存储结构
 - D. 数据结构仅由其逻辑结构和存储结构决定
6. 在存储数据时, 通常不仅要存储各数据元素的值, 而且还要存储 ()。
- A. 数据的操作方法
 - B. 数据元素的类型
 - C. 数据元素之间的关系
 - D. 数据的存取方法
7. 链式存储设计时, 结点内的存储单元地址 ()。
- A. 一定连续
 - B. 一定不连续
 - C. 不一定连续
 - D. 部分连续, 部分不连续

二、综合应用题

1. 对于两种不同的数据结构, 逻辑结构或物理结构一定不相同吗?
2. 试举一例, 说明对相同的逻辑结构, 同一种运算在不同的存储方式下实现, 其运算效率不同。

1.1.4 答案与解析

一、单项选择题

1. D

抽象数据类型 (ADT) 描述了数据的逻辑结构和抽象运算, 通常用 (数据对象, 数据关系, 基本操作集) 这样的三元组来表示, 从而构成了一个完整的数据结构定义。

2. A

树和图是典型的非线性数据结构, 其他选项都属于线性数据结构。

3. C

顺序表、哈希表和单链表表示几种数据结构, 既描述逻辑结构, 也描述存储结构和数据运算。而有序表是指关键字有序的线性表, 可以链式存储也可以顺序存储, 仅描述了元素之间的逻辑关系, 故它属于逻辑结构。

4. D

数据的存储结构有顺序存储、链式存储、索引存储和散列存储。循环队列 (易错点) 是用顺序表表示的队列 (参见本书 3.2.2 节), 是一种数据结构。栈是一种抽象数据类型, 可采用顺序存储或链式存储, 只表示逻辑结构。

5. A

数据的逻辑结构是以面向实际问题的角度出发的, 只采用抽象表达方式, 独立于存储结构, 数据的存储方式有多种不同的选择; 而数据的存储结构是逻辑结构在计算机上的映射, 它不能独立于逻辑结构而存在。数据结构包括三个要素, 缺一不可。

6. C

在存储数据时, 不仅要存储数据元素的值, 还要存储数据元素之间的关系。

7. A

链式存储设计时, 各个不同结点的存储空间可以不连续, 但是结点内的存储单元地址则必须连续。

二、综合应用题

1. 解答:

应该注意到,数据的运算也是数据结构的一个重要方面。

对于两种不同的数据结构,它们的逻辑结构和物理结构完全有可能相同。比如二叉树和二叉排序树,二叉排序树可以采用二叉树的逻辑表示和存储方式,前者通常用来表示层次关系,而后者通常用于排序和查找。虽然它们的运算都有建树、插入结点、删除结点和查找结点等,但是对于二叉树和二叉排序树,这些运算的定义是不同的,以查找结点为例,二叉树的时间复杂度为 $O(n)$,而二叉排序树的时间复杂度为 $O(\log_2 n)$ 。

2. 解答:

线性表既可以用顺序存储方式实现,也可以用链式存储方式实现。在顺序存储方式下,在线性表中插入和删除元素,平均要移动近一半的元素,时间复杂度为 $O(n)$;而在链式存储方式下,插入和删除的时间复杂度都是 $O(1)$ 。

1.2 算法和算法评价

1.2.1 算法的基本概念

算法 (Algorithm) 是对特定问题求解步骤的一种描述,它是指令的有限序列,其中每一条指令表示一个或多个操作。此外,一个算法还具有下列 5 个重要特性。

1) **有穷性** 一个算法必须总是 (对任何合法的输入值) 在执行有穷步之后结束,且每一步都可在有穷时间内完成。

2) **确定性** 算法中每一条指令必须有确切的含义,读者理解时不会产生二义性。即对于相同的输入只能得出相同的输出。

3) **可行性** 一个算法是可行的,即算法中描述的操作都是可以通过已经实现的基本运算执行有限次来实现的。

4) **输入** 一个算法有零个或多个的输入,这些输入取自于某个特定的对象的集合。

5) **输出** 一个算法有一个或多个的输出,这些输出是同输入有着某种特定关系的量。

通常设计一个“好”的算法应考虑达到以下目标。

1) **正确性**: 算法应当能够正确地解决求解问题。

2) **可读性**: 算法应当具有良好的可读性,以助于人们理解。

3) **健壮性**: 当输入非法数据时,算法也能适当地做出反应或进行处理,而不会产生莫名其妙的输出结果。

4) **效率与低存储量需求**: 效率是指算法执行的时间,存储量需求是指算法执行过程中所需要的最大存储空间,这两者都与问题的规模有关。

1.2.2 算法效率的度量

算法效率的度量是通过时间复杂度和空间复杂度来描述的。

1. 时间复杂度

一个语句的频度是指该语句在算法中被重复执行的次数。算法中所有语句的频度之和记作

$T(n)$, 它是该算法问题规模 n 的函数, 时间复杂度主要分析 $T(n)$ 的数量级。算法中的基本运算 (最深层循环内的语句) 的频度与 $T(n)$ 同数量级, 所以通常采用算法中基本运算的频度 $f(n)$ 来分析算法的时间复杂度^①。因此, 算法的时间复杂度记为:

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

上式中“ O ”的含义是 $T(n)$ 的数量级, 其严格的数学定义是: 若 $T(n)$ 和 $f(n)$ 是定义在正整数集合上的两个函数, 则存在正常数 C 和 n_0 , 使得当 $n \geq n_0$ 时, 都满足 $0 \leq T(n) \leq C \times f(n)$ 。

算法的时间复杂度不仅依赖于问题的规模 n , 也取决于待输入数据的性质 (如输入数据元素的初始状态)。

例如 在数组 $A[0 \cdots n-1]$ 中, 查找给定值 k 的算法大致如下:

```
(1) i=n-1;
(2) while(i>=0 && (A[i]!=k))
(3)    i--;
(4) return i;
```

此算法中的语句 (3) (基本运算) 的频度不仅与问题规模 n 有关, 还与输入实例中 A 的各元素取值及 k 的取值有关:

- ① 若 A 中没有与 k 相等的元素, 则语句 (3) 的频度 $f(n)=n$ 。
- ② 若 A 的最后一个元素等于 k , 则语句 (3) 的频度 $f(n)$ 是常数 0。

最坏时间复杂度是指在最坏情况下, 算法的时间复杂度。

平均时间复杂度是指所有可能输入实例在等概率出现的情况下, 算法的期望运行时间。

最好时间复杂度是指在最好情况下, 算法的时间复杂度。

一般总是考虑在最坏情况下的时间复杂度, 以保证算法的运行时间不会比它更长。

在分析一个程序的时间复杂性时, 有以下两条规则:

a) 加法规则

$$T(n)=T_1(n)+T_2(n)=O(f(n))+O(g(n))=O(\max(f(n), g(n)))$$

b) 乘法规则

$$T(n)=T_1(n) \times T_2(n)=O(f(n)) \times O(g(n))=O(f(n) \times g(n))$$

常见的渐近时间复杂度有:

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

2. 空间复杂度

算法的空间复杂度 $S(n)$, 定义为该算法所耗费的存储空间, 它是问题规模 n 的函数。渐近空间复杂度也常简称为空间复杂度, 记作 $S(n)=O(g(n))$ 。

一个上机程序除了需要存储空间来存放本身所用指令、常数、变量和输入数据外, 也需要一些对数据进行操作的工作单元和存储一些为实现计算所需信息的辅助空间, 若输入数据所占空间只取决于问题本身, 和算法无关, 则只需分析除输入和程序之外的额外空间。

算法原地工作是指算法所需辅助空间是常量, 即 $O(1)$ 。

1.2.3 本节试题精选

一、单项选择题

1. 一个算法应该是 ()。

^① 取 $f(n)$ 中随 n 增长最快的项将其系数置为 1 作为时间复杂度的度量。例如, $f(n)=a \cdot n^3+b \cdot n^2+c \cdot n$, 则其时间复杂度为 $O(n^3)$ 。

- A. 程序
C. 要满足五个基本特性
2. 某算法的时间复杂度为 $O(n^2)$, 表明该算法的 ()。
A. 问题规模是 n^2
C. 执行时间与 n^2 成正比
3. 以下算法的时间复杂度为 ()。

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

- A. $O(n)$ B. $O(n^2)$ C. $O(n\log_2 n)$ D. $O(\log_2 n)$
4. 【2011 年计算机联考真题】

设 n 是描述问题规模的非负整数, 下面程序片段的时间复杂度是 ()。

```
x=2;  
while(x<n/2)  
    x=2*x;
```

- A. $O(\log_2 n)$ B. $O(n)$ C. $O(n\log_2 n)$ D. $O(n^2)$
5. 【2012 年计算机联考真题】

求整数 n ($n \geq 0$) 阶乘的算法如下, 其时间复杂度是 ()。

```
int fact(int n){  
    if(n<=1) return 1;  
    return n*fact(n-1);  
}
```

- A. $O(\log_2 n)$ B. $O(n)$ C. $O(n\log_2 n)$ D. $O(n^2)$
6. 【2013 年计算机联考真题】

已知两个长度分别为 m 和 n 的升序链表, 若将它们合并为一个长度为 $m+n$ 的降序链表, 则最坏情况下的时间复杂度是 ()。

- A. $O(n)$ B. $O(m \times n)$ C. $O(\min(m, n))$ D. $O(\max(m, n))$
7. 【2014 年计算机联考真题】

下列程序段的时间复杂度是 ()。

```
count=0;  
for(k=1;k<=n;k*=2)  
    for(j=1;j<=n;j++)  
        count++;
```

- A. $O(\log_2 n)$ B. $O(n)$ C. $O(n\log_2 n)$ D. $O(n^2)$
8. 【2017 年计算机联考真题】

下列函数的时间复杂度是 ()。

```
int func(int n){  
    int i=0, sum=0;  
    while(sum < n) sum += ++i;  
    return i;  
}
```

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

9. 有以下算法, 其时间复杂度为 ()。

```
void fun(int n){
    int i=0;
    while(i*i*i<=n)
        i++;
}
```

- A. $O(n)$ B. $O(n \log n)$ C. $O(\sqrt[3]{n})$ D. $O(\sqrt{n})$
10. 程序段

```
for(i=n-1; i>1; i--)
    for(j=1; j<i; j++)
        if(A[j]>A[j+1])
```

A[j]与A[j+1]对换;

- 其中 n 为正整数, 则最后一行的语句频度在最坏情况下是 ()。

- A. $O(n)$ B. $O(n \log n)$ C. $O(n^3)$ D. $O(n^2)$
11. 以下算法中加下划线语句的执行次数为 ()。

```
int m=0, i, j;
for(i=1; i<=n; i++)
    for(j=1; j<=2*i; j++)
        m++;
```

- A. $n(n+1)$ B. n C. $n+1$ D. n^2
12. 下面说法错误的是 ()。

- I. 算法原地工作的含义是指不需要任何额外的辅助空间
 II. 在相同的规模 n 下, 复杂度 $O(n)$ 的算法在时间上总是优于复杂度 $O(2^n)$ 的算法
 III. 所谓时间复杂度是指最坏情况下, 估算算法执行时间的一个上界
 IV. 同一个算法, 实现语言的级别越高, 执行效率就越低

- A. I B. I, II C. I, IV D. III

二、综合应用题

1. 一个算法所需时间由下述递归方程表示, 试求出该算法的时间复杂度的级别 (或阶)。

$$T(n) = \begin{cases} 1 & \text{若 } n=1 \\ 2T(n/2) + n & \text{若 } n>1 \end{cases}$$

式中, n 是问题的规模, 为简单起见, 设 n 是 2 的整数次幂。

2. 分析以下各程序段, 求出算法的时间复杂度。

① $i=1; k=0;$

```
while(i<n-1){
    k=k+10*i;
    i++;
}
```

② $y=0;$

```
while((y+1)*(y+1)<=n)
    y=y+1;
```

③ $\text{for}(i=1; i<=n; i++)$

```
    for(j=1; j<=i; j++)
        for(k=1; k<=j; k++)
            x++;
```

④ $\text{for}(i=0; i<n; i++)$

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

1.2.4 答案与解析

一、单项选择题

1. B

程序不一定满足有穷性,如死循环、操作系统等,而算法必须有穷。算法代表了对问题求解步骤的描述,而程序则是算法在计算机上的特定的实现。

2. C

时间复杂度为 $O(n^2)$,说明算法的时间复杂度 $T(n)$ 满足 $T(n) \leq c \times n^2$ (c 为比例常数),即 $T(n)=O(n^2)$,时间复杂度 $T(n)$ 是问题规模 n 的函数,其问题规模仍然是 n 而不是 n^2 。

3. D

基本运算是 $i=i*2$,设其执行次数为 t ,则 $2^t \leq n$,即 $t \leq \log_2 n$,则该算法的时间复杂度 $T(n)=O(\log_2 n)$ 。

更加直观、快速的解题方法:要计算语句 $i=i*2$ 的执行次数(由于每执行一次 i 乘 2),其中判断条件可理解为 $2^t=n$,即 $t=\log_2 n$,则 $T(n)=O(\log_2 n)$ 。(注:可将此方法应用到该章节的第 4、8 题。考生应灵活应用)

4. A

在程序中,执行频率最高的语句为“ $x=2*x$ ”。设该语句共执行了 t 次,则 $2^{t+1} < n/2$,故 $\log_2(n/2)-1=\log_2 n-2$,得 $T(n)=O(\log_2 n)$ 。

5. B

本题是求阶乘 $n!$ 的递归代码,即 $n*(n-1)*\dots*1$,共执行 n 次乘法操作,故 $T(n)=O(n)$ 。

6. D

两个升序链表合并,两两比较表中元素,每比较一次确定一个元素的链接位置(取较小元素,头插法)。当一个链表比较结束后,将另一个链表的剩余元素插入即可。最坏的情况是两个链表中的元素依次进行比较,时间复杂度为 $O(\max(m, n))$ 。

7. C

内层循环条件 $j \leq n$ 与外层循环的变量无关,每次循环 j 自增 1,每次内层循环都执行 n 次。外层循环条件为 $k \leq n$,增量定义为 $k*=2$,可知循环次数为 $2^k \leq n$,即 $k \leq \log_2 n$ 。所以内层循环的时间复杂度是 $O(n)$,外层循环的时间复杂度是 $O(\log_2 n)$ 。对于嵌套循环,根据乘法规则可知,该段程序的时间复杂度 $T(n)=T_1(n)*T_2(n)=O(n)*O(\log_2 n)=O(n \log_2 n)$ 。

8. B

$\text{sum} += ++i$; 相当于 $++i$; $\text{sum} = \text{sum} + i$ 。进行到第 k 趟循环, $\text{sum} = (1+k)*k/2$ 。显然需要进行 $O(n^{1/2})$ 趟循环,因此这也是该函数的时间复杂度。

9. C

基本运算是 $i++$,设其执行次数为 t ,则有, $t \times t \times t \leq n$,即 $t^3 \leq n$ 。故有, $t \leq \sqrt[3]{n}$,则 $T(n)=O(\sqrt[3]{n})$ 。

10. D

当所有相邻元素都为逆序时,则最后一行的语句每次都会执行。此时,

$$T(n) = \sum_{i=2}^{n-1} \sum_{j=1}^{i-1} 1 = \sum_{i=2}^{n-1} i - 1 = (n-2)(n-1)/2 = O(n^2)$$

所以在最坏情况下的该语句频度是 $O(n^2)$ 。

11. A

$m++$ 语句的执行次数为

$$\sum_{i=1}^n \sum_{j=1}^{2i} 1 = \sum_{i=1}^n 2i = 2 \sum_{i=1}^n i = n(n+1)$$

12. A

I, 算法原地工作是指算法所需的辅助空间是常量。II, 题中是指算法的时间复杂度, 不要想当然认为是程序(该算法的实现)的具体执行时间, 而赋予 n 一个特殊的值。时间复杂度为 $O(n)$ 的算法, 必然总是优于时间复杂度为 $O(2^n)$ 的算法。III, 时间复杂度总是考虑在最坏情况下的时间复杂度, 以保证算法的运行时间不会比它更长。IV 为严蔚敏教材的原话, 该问题在论坛讨论过多年, 对于这种在语言层次上的效率问题, 建议不要以特例程序来解释其优劣, 该结论肯定没有任何错误的, 严老师这么严谨的一个老师, 也不可能说这么多年, 把这么明显和确定的一个“错误”放在她的教材上。

二、综合应用题

1. 解答:

时间复杂度为 $O(n \log_2 n)$ 。

设 $n=2^k (k \geq 0)$, 根据题目所给定义, 有 $T(2^k) = 2T(2^{k-1}) + 2^k = 2^2 T(2^{k-2}) + 2 \times 2^k$, 由此, 可得一般递推公式 $T(2^k) = 2^i T(2^{k-i}) + i \times 2^k$, 进而, 可得 $T(2^k) = 2^k T(2^0) + k \times 2^k = (k+1)2^k$, 即 $T(n) = 2^{\log_2 n} + \log_2 n \cdot n = n(\log_2 n + 1)$, 即为 $O(n \log_2 n)$ 。

2. 解答:

① 基本语句是 $k=k+10*i$, 共执行了 $n-2$ 次, 所以 $T(n)=O(n)$ 。

② 设循环体共执行 t 次, 每循环一次, 循环变量 y 加 1, 最终 $t=y$ 。故 $t^2 \leq n$, 解得 $T(n)=O(n^{1/2})$ 。

③ $x++$ 是基本语句, $T(n)=O\left(\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j 1\right)=O\left(\frac{1}{6}n^3\right)=O(n^3)$ 。

④ $a[i][j]=0$ 是基本语句, 内循环执行 m 次, 外循环执行 n 次, 共执行了 $m*n$ 次, 所以 $T(m, n)=O(m*n)$ 。

归纳总结

本章的重点是分析程序的时间复杂度。一定要掌握分析时间复杂度的方法和步骤, 很多同学在做题时一眼就能看出程序的时间复杂度, 但就是无法将其推导过程规范地表述出来。为此, 编者查阅众多资料, 总结出此类题型的两种形式, 供大家参考:

一、循环主体中的变量参与循环条件的判断

此类题应该找出主体语句中与 $T(n)$ 成正比的循环变量, 将之带入条件中进行计算。

例如:

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

```
2. int y=5;
   while((y+1)*(y+1)<n)
       y=y+1;
```

例 1 中, i 乘以 2 的次数正是主体语句的执行次数 t , 因此有 $2^t \leq n$, 取对数后, 即得 $t \leq \log_2 n$, 则 $T(n)=O(\log_2 n)$ 。

例 2 中, y 加 1 的次数恰好与 $T(n)$ 成正比, 则记 t 为该程序的执行次数并令 $t=y-5$, 即 $y=t+5$, $(t+5+1)*(t+5+1)<n$, 得出 $t < \sqrt{n}-6$, 即 $T(n)=O(\sqrt{n})$ 。

二、循环主体中的变量与循环条件无关

此类题可采用数学归纳法或直接累计循环次数。多层循环时从内到外分析，忽略单步语句、条件判断语句，只关注主体语句的执行次数。此类问题又可分为递归程序和非递归程序：

- 递归程序一般使用公式进行递推。例如习题 5 的时间复杂度分析如下：

$$T(n)=1+T(n-1)=1+1+T(n-2)=\cdots=n-1+T(1)$$

$$\text{即 } T(n)=O(n)。$$

- 非递归程序比较简单，可以直接累计次数。例如习题 8、9 等。

思维拓展

求解斐波那契数列 $F(n)=\begin{cases} 1, & n=0,1 \\ F(n-1)+F(n-2), & n>1 \end{cases}$ 有两种常用的算法：递归算法和非递归算法。

试分别分析两种算法的时间复杂度。

（提示：请结合归纳总结中的两种方法进行解答。）

第2章 线性表

【考纲内容】

(一) 线性表的定义和基本操作

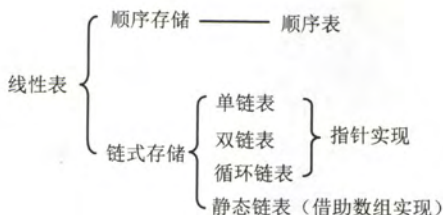
(二) 线性表的实现

1. 顺序存储

2. 链式存储

3. 线性表的应用

【知识框架】



【复习提示】

线性表是考研命题的重中之重。除 2014 年是关于树的算法设计题之外，2009 年到 2013 年的算法设计题都是基于线性表（顺序表或单链表）的，这类算法题实现起来比较容易而且代码量较少，但却要求具有最优的性能（时间复杂度和空间复杂度），才能获得满分。因此，应牢固掌握线性表的各种基本操作（基于两种存储结构），读者在平时的学习中应多注重积累和培养动手能力。另外，需要提醒的是：算法最重要的是思想！在考场上的时间有限，在试卷上不一定要求代码具有实际的可执行性，因此应尽力表达出算法的思想和步骤，不必过于拘泥于每一个细节。考纲把 Java 从算法设计题中的备选语言中删去，也就是说算法题只能用 C/C++ 来完成。

2.1 线性表的定义和基本操作

2.1.1 线性表的定义

线性表是具有相同数据类型的 n ($n \geq 0$) 个数据元素的有限序列。其中 n 为表长，当 $n=0$ 时该线性表是一个空表。若用 L 命名线性表，则其一般表示如下：

$$L=(a_1, a_2, \dots, a_i, a_{i+1}, \dots, a_n)$$

其中， a_1 是唯一的“第一个”数据元素，又称为表头元素； a_n 是唯一的“最后一个”数据元素，又称为表尾元素。除第一个元素外，每个元素有且仅有一个直接前驱。除最后一个元素外，每个元素有且仅有一个直接后继。以上就是线性表的逻辑特性，这种线性有序的逻辑结构正是线性表

名字的由来。

由此,我们得出线性表的特点如下:

表中元素的个数有限。

表中元素具有逻辑上的顺序性,在序列中各元素排序有其先后次序。

表中元素都是数据元素,每一个元素都是单个元素。

表中元素的数据类型都相同。这意味着每一个元素占有相同大小的存储空间。

表中元素具有抽象性。即仅讨论元素间的逻辑关系,不考虑元素究竟表示什么内容。

注意:线性表是一种逻辑结构,表示元素之间一对一的相邻关系。顺序表和链表是指存储结构,两者属于不同层面的概念,因此不要将其混淆。

2.1.2 线性表的基本操作

一个数据结构的基本操作是指其最核心、最基本的操作。其他较复杂的操作可以通过调用其基本操作来实现。线性表的主要操作如下:

InitList(&L): 初始化表。构造一个空的线性表。

Length(L): 求表长。返回线性表 L 的长度,即 L 中数据元素的个数。

LocateElem(L, e): 按值查找操作。在表 L 中查找具有给定关键字值的元素。

GetElem(L, i): 按位查找操作。获取表 L 中第 i 个位置的元素的值。

ListInsert(&L, i, e): 插入操作。在表 L 中第 i 个位置上插入指定元素 e。

ListDelete(&L, i, &e): 删除操作。删除表 L 中第 i 个位置的元素,并用 e 返回删除元素的值。

PrintList(L): 输出操作。按前后顺序输出线性表 L 的所有元素值。

Empty(L): 判空操作。若 L 为空表,则返回 true,否则返回 false。

DestroyList(&L): 销毁操作。销毁线性表,并释放线性表 L 所占用的内存空间。

注意:①基本操作的实现取决于采用哪一种存储结构,存储结构不同,算法的实现也不同。②'&'表示 C++ 中的引用。如果传入的变量是指针型的变量,且在函数体内要对传入的指针进行改变,则将用到指针变量的引用型。在 C 中采用指针的指针也可达到同样的效果。

2.1.3 本节试题精选

一、单项选择题

- 线性表是具有 n 个 () 的有限序列。
A. 数据表
B. 字符
C. 数据元素
D. 数据项
- 以下 () 是一个线性表。
A. 由 n 个实数组成的集合
B. 由 100 个字符组成的序列
C. 所有整数组成的序列
D. 邻接表
- 在线性表中,除了开始元素外,每个元素 ()。
A. 只有唯一的前趋元素
B. 只有唯一的后继元素
C. 有多个前趋元素
D. 有多个后继元素

2.1.4 答案与解析

一、单项选择题

- C

线性表是由具有相同数据类型的有限个数据元素组成的，数据元素是由数据项组成的。

2. B

线性表的定义要求为有限序列，而 C 中序列的元素个数是无穷多个，C 错误；而 A 中指定的是集合，集合中各元素没有前后驱关系，A 错误；D 是属于存储结构，线性表是一种逻辑结构，不要将二者混为一谈。只有 B 满足线性表定义的条件。

3. A

线性表中，除了最后一个元素外，每个元素只有唯一的后继元素。

2.2 线性表的顺序表示

2.2.1 顺序表的定义

线性表的顺序存储又称为**顺序表**。它是用一组地址连续的存储单元，依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻。第 1 个元素存储在线性表的起始位置，第 i 个元素的存储位置后面紧接着存储的是第 $i+1$ 个元素。因此，顺序表的特点是表中元素的逻辑顺序与其物理顺序相同。

假设线性表 L 存储的起始位置为 $LOC(A)$ ， $sizeof(ElemType)$ 是每个数据元素所占用存储空间的大小，则表 L 所对应的顺序存储如图 2-1 所示。

数组下标	顺序表	内存地址
0	a_1	$LOC(A)$
1	a_2	$LOC(A)+sizeof(ElemType)$
	\vdots	
$i-1$	a_i	$LOC(A)+(i-1) \times sizeof(ElemType)$
	\vdots	
$n-1$	a_n	$LOC(A)+(n-1) \times sizeof(ElemType)$
	\vdots	
$MaxSize-1$	\vdots	$LOC(A)+(MaxSize-1) \times sizeof(ElemType)$

图 2-1 线性表的顺序存储结构

注意：线性表中元素的位序是从 1 开始的，而数组中元素的下标是从 0 开始的。

假定线性表的元素类型为 $ElemType$ ，线性表的顺序存储类型描述为

```
#define MaxSize 50           //定义线性表的最大长度
typedef struct{
    ElemType data[MaxSize];   //顺序表的元素
    int length;               //顺序表的当前长度
}SqList;                     //顺序表的类型定义
```

一维数组可以是**静态分配**的，也可以是**动态分配**的。在静态分配时，由于数组的大小和空间事先已经固定，一旦空间占满，再加入新的数据将产生溢出，就会导致程序崩溃。

而动态分配时，存储数组的空间是在程序执行过程中通过动态存储分配语句分配的，一旦数据空间占满，可以另外开辟一块更大的存储空间，用以替换原来的存储空间，从而达到扩充存储数组空间的目的，而不需要一次性地划分所有所需空间给线性表。

```
#define InitSize 100         //表长度的初始定义
typedef struct{
    ElemType *data;          //指示动态分配数组的指针
```

```
int MaxSize,length;           //数组的最大容量和当前个数
} SeqList;                    //动态分配数组顺序表的类型定义
```

C 的初始动态分配语句为

```
L.data=(ElemType*)malloc(sizeof(ElemType)*InitSize);
```

C++的初始动态分配语句为

```
L.data=new ElemType[InitSize];
```

注意：动态分配并不是链式存储，同样还是属于顺序存储结构，其物理结构没有变化，依然是随机存取方式，只是分配的空间大小可以在运行时决定。

顺序表最主要的特点是随机访问，即通过首地址和元素序号可以在 $O(1)$ 的时间内找到指定的元素。

顺序表的存储密度高，每个结点只存储数据元素。

顺序表逻辑上相邻的元素物理上也相邻，所以插入和删除操作需要移动大量元素。

2.2.2 顺序表上基本操作的实现

我们在这里仅给出顺序表的插入操作、删除操作和按值查找的算法，其他基本操作的算法相对都比较简单，请读者自行思考。

(1) 插入操作

在顺序表 L 的第 i ($1 \leq i \leq L.length+1$) 个位置插入新元素 e 。如果 i 的输入不合法，则返回 `false`，表示插入失败；否则，将顺序表的第 i 个元素以及其后的所有元素右移一个位置，腾出一个空位置插入新元素 e ，顺序表长度增加 1，插入成功，返回 `true`。

```
bool ListInsert(SqList &L,int i,ElemType e){
//本算法实现将元素e插入到顺序表L中第i个位置
    if(i<1||i>L.length+1)           //判断i的范围是否有效
        return false;
    if(L.length>=MaxSize)             //当前存储空间已满，不能插入
        return false;
    for(int j=L.length;j>=i;j--)      //将第i个元素及之后的元素后移
        L.data[j]=L.data[j-1];
    L.data[i-1]=e;                    //在位置i处放入e
    L.length++;                       //线性表长度加1
    return true;
}
```

注意：区别顺序表的位序和数组下标。理解为什么判断插入位置是否合法时 `if` 语句中用 `length+1`，而移动元素的 `for` 语句中只用 `length`？

最好情况：在表尾插入（即 $i=n+1$ ），元素后移语句将不执行，时间复杂度为 $O(1)$ 。

最坏情况：在表头插入（即 $i=1$ ），元素后移语句将执行 n 次，时间复杂度为 $O(n)$ 。

平均情况：假设 p_i ($p_i=1/(n+1)$) 是在第 i 个位置上插入一个结点的概率，则在长度为 n 的线性表中插入一个结点时所需移动结点的平均次数为

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

因此，线性表插入算法的平均时间复杂度为 $O(n)$ 。

(2) 删除操作

删除顺序表 L 中第 i ($1 \leq i \leq L.length$) 个位置的元素，成功则返回 `true`，并将被删除的元素用引用变量 e 返回，否则返回 `false`。


```

bool ListDelete(SqList &L, int i, Elemtype &e){
//本算法实现删除顺序表 L 中第 i 个位置的元素
    if(i<1||i>L.length)                //判断 i 的范围是否有效
        return false;
    e=L.data[i-1];                        //将被删除的元素赋值给 e
    for(int j=i; j<L.length; j++)        //将第 i 个位置之后的元素前移
        L.data[j-1]=L.data[j];
    L.length--;                          //线性表长度减 1
    return true;
}

```

最好情况：删除表尾元素（即 $i=n$ ），无须移动元素，时间复杂度为 $O(1)$ 。

最坏情况：删除表头元素（即 $i=1$ ），需要移动除第一个元素外的所有元素，时间复杂度为 $O(n)$ 。

平均情况：假设 $p_i (p_i=1/n)$ 是删除第 i 个位置上结点的概率，则在长度为 n 的线性表中删除一个结点时所需移动结点的平均次数为

$$\sum_{i=1}^n p_i (n-i) = \sum_{i=1}^n \frac{1}{n} (n-i) = \frac{1}{n} \sum_{i=1}^n (n-i) = \frac{1}{n} \frac{n(n-1)}{2} = \frac{n-1}{2}$$

因此，线性表删除算法的平均时间复杂度为 $O(n)$ 。

如图 2-2 所示为一个顺序表在进行插入和删除操作的前、后状态，其数据元素在存储空间中的位置变化以及表长的变化。在图 2-2 (a) 中，元素移动是从后往前依次后移一个位置，在图 2-2 (b) 中，元素移动是从前往后依次前移一个位置。

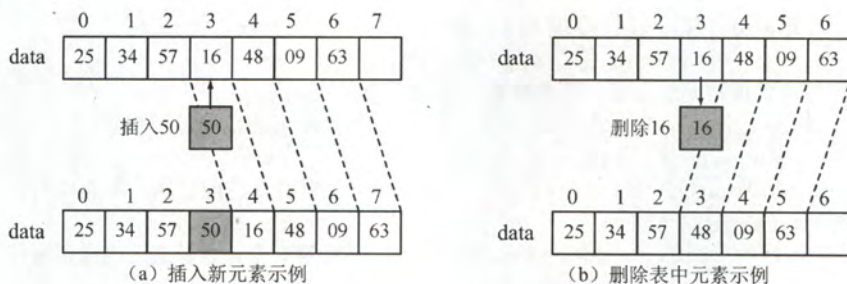


图 2-2 顺序表的插入和删除

(3) 按值查找（顺序查找）

在顺序表 L 中查找第一个元素值等于 e 的元素，并返回其位序。

```

int LocateElem(SqList L, ElemType e){
//本算法实现查找顺序表中值为 e 的元素，如果查找成功，返回元素位序，否则返回 0
    int i;
    for(i=0; i<L.length; i++)
        if(L.data[i]==e)
            return i+1;                //下标为 i 的元素值等于 e，返回其位序 i+1
    return 0;                          //退出循环，说明查找失败
}

```

最好情况：查找的元素就在表头，仅需比较一次，时间复杂度为 $O(1)$ 。

最坏情况：查找的元素在表尾（或不存在）时，需要比较 n 次，时间复杂度为 $O(n)$ 。

平均情况：假设 $p_i (p_i=1/n)$ 是查找的元素在第 i ($1 \leq i \leq L.length$) 个位置上的概率，则在长度为 n 的线性表中查找值为 e 的元素所需比较的平均次数为

$$\sum_{i=1}^n p_i \times i = \sum_{i=1}^n \frac{1}{n} \times i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

因此,线性表按值查找算法的平均时间复杂度为 $O(n)$ 。

2.2.3 本节试题精选

一、单项选择题

1. 下述哪一条是顺序存储结构的优点 ()。
 - A. 存储密度大
 - B. 插入运算方便
 - C. 删除运算方便
 - D. 方便地运用于各种逻辑结构的存储表示
2. 线性表的顺序存储结构是一种 ()。
 - A. 随机存取的存储结构
 - B. 顺序存取的存储结构
 - C. 索引存取的存储结构
 - D. 散列存取的存储结构
3. 一个顺序表所占用的存储空间大小与 () 无关。
 - A. 表的长度
 - B. 元素的存放顺序
 - C. 元素的类型
 - D. 元素中各字段的类型
4. 若线性表最常用的操作是存取第 i 个元素及其前驱和后继元素的值,为了提高效率,应采用 () 的存储方式。
 - A. 单链表
 - B. 双向链表
 - C. 单循环链表
 - D. 顺序表
5. 一个线性表最常用的操作是存取任一指定序号的元素和在最后进行插入删除操作,则利用 () 存储方式可以节省时间。
 - A. 顺序表
 - B. 双链表
 - C. 带头结点的双循环链表
 - D. 单循环链表
6. 在 n 个元素的线性表的数组表示中,以下时间复杂度为 $O(1)$ 的操作是 ()。
 - I. 访问第 i 个结点 ($1 \leq i \leq n$) 和求第 i 个结点的直接前驱 ($2 \leq i \leq n$)
 - II. 在最后一个结点后插入一个新的结点
 - III. 删除第 1 个结点
 - IV. 在第 i 个结点后插入一个结点 ($1 \leq i \leq n$)
 - A. I
 - B. II、III
 - C. I、II
 - D. I、II、III
7. 设线性表有 n 个元素,严格说来,以下操作中, () 在顺序表上实现要比链表上实现的效率高。
 - I. 输出第 i ($1 \leq i \leq n$) 个元素值
 - II. 交换第 3 个元素与第 4 个元素的值
 - III. 顺序输出这 n 个元素的值
 - A. I
 - B. I、III
 - C. I、II
 - D. II、III
8. 在一个长度为 n 的顺序表中删除第 i 个元素 ($1 \leq i \leq n$) 时,需向前移动 () 个元素。
 - A. n
 - B. $i-1$
 - C. $n-i$
 - D. $n-i+1$
9. 对于顺序表,访问第 i 个位置的元素和在第 i 个位置插入一个元素的时间复杂度为 ()。
 - A. $O(n), O(n)$
 - B. $O(n), O(1)$
 - C. $O(1), O(n)$
 - D. $O(1), O(1)$
10. 若长度为 n 的非空线性表采用顺序存储结构,在表的第 i 个位置插入一个数据元素, i 的合法值应该是 ()。
 - A. $1 \leq i \leq n$
 - B. $1 \leq i \leq n+1$
 - C. $0 \leq i \leq n-1$
 - D. $0 \leq i \leq n$

二、综合应用题

1. 从顺序表中删除具有最小值的元素(假设唯一)并由函数返回被删元素的值。空出的位置由最后一个元素填补,若顺序表为空则显示出错信息并退出运行。

2. 设计一个高效的算法,将顺序表的所有元素逆置,要求算法的空间复杂度为 $O(1)$ 。

3. 长度为 n 的顺序表 L ,编写一个时间复杂度为 $O(n)$ 、空间复杂度为 $O(1)$ 的算法,该算法删除线性表中所有值为 x 的数据元素。

4. 从有序顺序表中删除其值在给定值 s 与 t 之间(要求 $s < t$)的所有元素,如果 s 或 t 不合理或者顺序表为空则显示出错信息并退出运行。

5. 从顺序表中删除其值在给定值 s 与 t 之间(包含 s 和 t , 要求 $s < t$)的所有元素,如果 s 或 t 不合理或者顺序表为空则显示出错信息并退出运行。

6. 从有序顺序表中删除所有其值重复的元素,使表中所有元素的值均不同。

7. 将两个有序顺序表合并成一个新的有序顺序表,并由函数返回结果顺序表。

8. 已知在一维数组 $A[m+n]$ 中依次存放着两个线性表 $(a_1, a_2, a_3, \dots, a_m)$ 和 $(b_1, b_2, b_3, \dots, b_n)$ 。试编写一个函数,将数组中两个顺序表的位置互换,即将 $(b_1, b_2, b_3, \dots, b_n)$ 放在 $(a_1, a_2, a_3, \dots, a_m)$ 的前面。

9. 线性表 $(a_1, a_2, a_3, \dots, a_n)$ 中元素递增有序且按顺序存储于计算机内。要求设计一算法完成用最少时间在表中查找数值为 x 的元素,若找到将其与后继元素位置相交换,若找不到将其插入表中并使表中元素仍递增有序。

10. 【2010年计算机联考真题】

设将 n ($n > 1$) 个整数存放于一维数组 R 中。试设计一个在时间和空间两方面都尽可能高效的算法。将 R 中保存的序列循环左移 p ($0 < p < n$) 个位置,即将 R 中的数据由 $(X_0, X_1, \dots, X_{n-1})$ 变换为 $(X_p, X_{p+1}, \dots, X_{n-1}, X_0, X_1, \dots, X_{p-1})$ 。要求:

(1) 给出算法的基本设计思想。

(2) 根据设计思想,采用 C 或 C++ 或 Java 语言描述算法,关键之处给出注释。

(3) 说明你所设计算法的时间复杂度和空间复杂度。

11. 【2011年计算机联考真题】

一个长度为 L ($L \geq 1$) 的升序序列 S , 处在第 $\lceil L/2 \rceil$ 个位置的数称为 S 的中位数。例如,若序列 $S_1 = (11, 13, 15, 17, 19)$, 则 S_1 的中位数是 15, 两个序列的中位数是含它们所有元素的升序序列的中位数。例如,若 $S_2 = (2, 4, 6, 8, 20)$, 则 S_1 和 S_2 的中位数是 11。现在有两个等长升序序列 A 和 B , 试设计一个在时间和空间两方面都尽可能高效的算法,找出两个序列 A 和 B 的中位数。要求:

(1) 给出算法的基本设计思想。

(2) 根据设计思想,采用 C 或 C++ 或 Java 语言描述算法,关键之处给出注释。

(3) 说明你所设计算法的时间复杂度和空间复杂度。

12. 【2013年计算机联考真题】

已知一个整数序列 $A = (a_0, a_1, \dots, a_{n-1})$, 其中 $0 \leq a_i < n$ ($0 \leq i < n$)。若存在 $a_{p_1} = a_{p_2} = \dots = a_{p_m} = x$ 且 $m > n/2$ ($0 \leq p_k < n, 1 \leq k \leq m$), 则称 x 为 A 的主元素。例如 $A = (0, 5, 5, 3, 5, 7, 5, 5)$, 则 5 为主元素; 又如 $A = (0, 5, 5, 3, 5, 1, 5, 7)$, 则 A 中没有主元素。假设 A 中的 n 个元素保存在一个一维数组中,请设计一个尽可能高效的算法,找出 A 的主元素。若存在主元素,则输出该元素;否则输出 -1。要求:

(1) 给出算法的基本设计思想。

(2) 根据设计思想,采用 C 或 C++ 或 Java 语言描述算法,关键之处给出注释。

(3) 说明你所设计算法的时间复杂度和空间复杂度。

2.2.4 答案与解析

一、单项选择题

1. A

顺序表不像链表要在结点中存放指针域,因此存储密度较大,A正确。B和C是链表的优点。D是错误的,比如对于树形结构,顺序表显然不如链表表示起来方便。

2. A

本题容易误选B。顺序表是一种支持随机存取的顺序存储结构,根据起始地址加上元素的序号,可以很方便地访问到任一元素,即随机存取的概念。注意,顺序存取是一种读写方式,不是存储方式,有别于顺序存储。

3. B

顺序表所占存储空间=表长 \times sizeof(元素的类型),元素的类型显然会影响到存储空间的大小。对于同一类型的顺序表,表越长,则所占存储空间就越大。

4. D

题干实际要求能够最快存取第 $i-1$ 、 i 和 $i+1$ 个元素值。A、B、C都只能从头结点依次顺序查找,时间复杂度为 $O(n)$,只有顺序表可以随机存取,时间复杂度为 $O(1)$ 。

5. A

只有顺序表具有随机存取的优点,且在最后进行插入和删除操作不需要移动任何元素。

6. C

I中,时间复杂度显然为 $O(1)$;II中,在最后一个结点插入一个新结点不需要移动元素,故也为 $O(1)$;III中,第一个结点后的结点需要依次前移,时间复杂度为 $O(n)$;IV中,需要移动 $n-i$ 个结点,时间复杂度为 $O(n)$ 。

7. C

顺序输出这 n 个元素的值,都要依次访问每个元素,故时间复杂度相同。

8. C

需要将 $a_{i+1} \sim a_n$ 元素前移一位,共移动 $n-(i+1)+1=n-i$ 个元素。

9. C

顺序表中,第 i 个元素的物理地址可以通过起始地址和序号直接计算出,即可在 $O(1)$ 时间内访问。在第 i 个位置插入一个元素,需要移动 $n-i+1$ 个元素,时间复杂度为 $O(n)$ 。

10. B

表元素序号从1开始,而在第 $n+1$ 个位置插入相当于在表尾追加。

二、综合应用题

1. 解答:

算法思想:搜索整个顺序表,查找最小值元素并记住其位置,搜索结束后用最后一个元素填补空出的原最小值元素位置。

本题代码如下:

```
bool Del_Min(sqList &L, ElemType &value) {
    //删除顺序表L中最小值元素结点,并通过引用型参数value返回其值
    //如果删除成功,返回true;否则,返回false
    if (L.length == 0)
        return false;                //表空,中止操作返回
```



```

value=L.data[0];
int pos=0;
for(int i=1;i<L.length;i++){
    if(L.data[i]<value){
        value=L.data[i];
        pos=i;
    }
}
L.data[pos]=L.data[L.length-1];
L.length--;
return true;

```

//假定 0 号元素的值最小
//循环, 寻找具有最小值的元素
//让 value 记忆当前具有最小值的元素
//空出的位置由最后一个元素填补
//此时, value 即为最小值

注意: 本题也可以用函数返回值返回, 两者的区别是: 函数返回值只能返回一个值, 而参数返回 (引用传参) 则可以返回多个值。

2. 解答:

算法思想: 扫描顺序表 L 的前半部分元素, 对于元素 L.data[i] ($0 \leq i < L.length/2$), 将其余后半部分对应元素 L.data[L.length-i-1] 进行交换。

本题代码如下:

```

void Reverse(SqList &L){
    Elemtype temp; //辅助变量
    for(i=0;i<L.length/2;i++){
        temp=L.data[i]; //交换 L.data[i] 与 L.data[L.length-i-1]
        L.data[i]=L.data[L.length-i-1];
        L.data[L.length-i-1]=temp;
    }
}

```

3. 解答:

解法一: 用 k 记录顺序表 L 中不等于 x 的元素个数 (即需要保存的元素个数), 边扫描 L 边统计 k, 并将不等于 x 的元素向前放置 k 位置上, 最后修改 L 的长度。

本题代码如下:

```

void del_x_1(SqList &L, Elemtype x){
    //本算法实现删除顺序表 L 中所有值为 x 的数据元素
    int k=0; //记录值不等于 x 的元素个数
    for(i=0;i<L.length;i++){
        if(L.data[i]!=x){
            L.data[k]=L.data[i];
            k++; //不等于 x 的元素增 1
        }
    }
    L.length=k; //顺序表 L 的长度等于 k
}

```

解法二: 用 k 记录顺序表 L 中等于 x 的元素个数, 边扫描 L 边统计 k, 并将不等于 x 的元素前移 k 个位置, 最后修改 L 的长度。

本题代码如下:

```

void del_x_2(SqList &L, Elemtype x){
    int k=0, i=0; //k 记录值等于 x 的元素个数
    while(i<L.length){
        if(L.data[i]==x)
            k++;
        else

```

```

        L.data[i-k]=L.data[i]; //当前元素前移 k 个位置
        i++;
    }
    L.length=L.length-k;          //顺序表 L 的长度递减
}

```

此外,本题还可以考虑设头、尾两个指针($i=1, j=n$),从两端向中间移动,凡遇到最左端值 x 的元素时,直接将最右端值非 x 的元素左移至值为 x 的数据元素位置,直到两指针相遇。但这种方法会改变原表中元素的相对位置。

4. 解答:

注意:本题与上一题的区别,因为是有序表,所以删除的元素必然是相连的整体。

算法思想:先寻找值大于等于 s 的第一个元素(第一个删除的元素),然后寻找值 $> t$ 的第一个元素(最后一个删除的元素的下一个元素),要将这段元素删除,则只需直接将后面的元素前移即可。

本题代码如下:

```

bool Del_s_t2(SqList &L, ElemType s, ElemType t) {
    //删除有序顺序表 L 中值在给定值 s 与 t 之间的所有元素
    int i, j;
    if(s>t || L.length==0)
        return false;
    for(i=0; i<L.length&&L.data[i]<s; i++); //寻找值 ≥ s 的第一个元素
    if(i>=L.length)
        return false; //所有元素值均小于 s, 则返回
    for(j=i; j<L.length&&L.data[j]<=t; j++); //寻找值 > t 的第一个元素
    for(; j<L.length; i++, j++)
        L.data[i]=L.data[j]; //前移, 填补被删元素位置
    L.length=i+1;
    return true;
}

```

5. 解答:

算法思想:从前向后扫描顺序表 L ,用 k 记录下元素值在 s 到 t 之间元素的个数(初始时 $k=0$)。对于当前扫描的元素,若其值不在 s 到 t 之间,则前移 k 个位置;否则执行 $k++$ 。由于这样每个不在 s 到 t 之间的元素仅移动一次,所以算法效率高。

本题代码如下:

```

bool Del_s_t(SqList &L, ElemType s, ElemType t) {
    //删除顺序表 L 中值在给定值 s 与 t 之间 (要求 s<t) 的所有元素
    int i, k=0;
    if(L.length==0 || s>=t)
        return false; //线性表为空或 s、t 不合法, 返回
    for(i=0; i<L.length; i++) {
        if(L.data[i]>=s&&L.data[i]<=t)
            k++;
        else
            L.data[i-k]=L.data[i]; //当前元素前移 k 个位置
    } //for
    L.length-=k; //长度减小
    return true;
}

```

注意:本题也可以从后向前扫描顺序表,每遇到一个值在 s 到 t 之间的元素,则删除该元素,

其后的所有元素全部前移。但移动次数远大于前者，效率不够高。

6. 解答:

算法思想: 注意是有序顺序表, 值相同的元素一定在连续的位置上, 用类似于直接插入排序的思想, 初始时将第一个元素看做非重复的有序表。之后依次判断后面的元素是否与前面非重复有序表的最后一个元素相同, 如果相同则继续向后判断, 如果不同则插入到前面的非重复有序表的最后, 直至判断到表尾为止。

本题代码如下:

```
bool Delete_Same(SeqList& L){
    if(L.length==0)
        return false;
    int i,j; //i 存储第一个不相同的元素, j 工作指针
    for(i=0,j=1;j<L.length;j++)
        if(L.data[i]!=L.data[j]) //查找下一个与上个元素值不同的元素
            L.data[++i]=L.data[j]; //找到后, 则将元素前移
    L.length=i+1;
    return true;
}
```

对于本题的算法, 请读者以序列 1,2,2,2,3,3,3,4,4,5 来手动模拟算法的执行过程, 在模拟的过程中要标注 i 和 j 所指示的元素。

思考: 如果将本题的有序表改为无序表, 你能想到时间复杂度为 $O(n)$ 的方法吗?

(提示: 使用 Hash 表)

7. 解答:

算法思想: 首先, 按顺序不断取下两个顺序表表头较小的结点存入新的顺序表中。然后, 看哪个表还有剩余, 将剩下的部分加到新的顺序表后面。

```
本题代码如下: bool Merge(SeqList A, SeqList B, SeqList &C){
    //合并有序顺序表 A 与 B 成为一个新的有序顺序表 C
    if(A.length+B.length>C.maxSize) //大于顺序表的最大长度
        return false;
    int i=0,j=0,k=0;
    while(i<A.length&&j<B.length){ //循环, 两两比较, 小者存入结果表
        if(A.data[i]<=B.data[j])
            C.data[k++]=A.data[i++];
        else
            C.data[k++]=B.data[j++];
    }
    while(i<A.length) //还剩一个没有比较完的顺序表
        C.data[k++]=A.data[i++];
    while(j<B.length)
        C.data[k++]=B.data[j++];
    C.length=k+1;
    return true;
}
```

注, 本算法的方法非常典型, 需牢固掌握。

8. 解答:

算法思想: 首先将数组 $A[m+n]$ 中的全部元素 $(a_1, a_2, a_3, \dots, a_m, b_1, b_2, b_3, \dots, b_n)$ 原地逆置为 $(b_n, b_{n-1}, b_{n-2}, \dots, b_1, a_m, a_{m-1}, a_{m-2}, \dots, a_1)$, 再对前 n 个元素和后 m 个元素分别使用逆置算法, 就可以得到 $(b_1, b_2, b_3, \dots, b_n, a_1, a_2, a_3, \dots, a_m)$, 从而实现顺序表的位置互换。

本题代码如下:

```
typedef int DataType;
void Reverse(DataType A[],int left,int right,int arraySize){
//逆转 (aleft,aleft+1,aleft+2...,aright) 为 (aright,aright-1,...,aleft)
    if(left>=right||right>=arraySize)
        return;
    int mid=(left+right)/2;
    for(int i=0;i<=mid-left;i++){
        DataType temp=A[left+i];
        A[left+i]=A[right-i];
        A[right-i]=temp;
    }
}

void Exchange(DataType A[],int m,int n,int arraySize){
/*数组 A[m+n]中, 从0到m-1存放顺序表(a1,a2,a3,...,am), 从m到m+n-1存放顺序表
(b1,b2,b3,...,bn), 算法将这两个表的位置互换*/
    Reverse(A,0,m+n-1,arraySize);
    Reverse(A,0,n-1,arraySize);
    Reverse(A,n,m+n-1,arraySize);
}
```

9. 解答:

算法思想: 顺序存储的线性表递增有序, 可以顺序查找, 也可折半查找。题目要求“用最少的时间在表中查找数值为 x 的元素”, 这里应使用折半查找法。

本题代码如下:

```
void SearchExchangeInsert(ElemType A[],ElemType x){
    int low=0,high=n-1,mid;           //low和high指向顺序表下界和上界的下标
    while(low<=high){
        mid=(low+high)/2;              //找中间位置
        if(A[mid]==x) break;           //找到x, 退出while循环
        else if(A[mid]<x) low=mid+1;    //到中点mid的右半部去查
        else high=mid-1;               //到中点mid的左半部去查
    } //下面两个if语句只会执行一个
    if(A[mid]==x&&mid!=n-1){           //若最后一个元素与x相等, 则不存在与其后
        //继交换的操作
        t=A[mid]; A[mid]=A[mid+1]; A[mid+1]=t;
    }
    if(low>high){                      //查找失败, 插入数据元素x
        for(i=n-1;i>high;i--) A[i+1]=A[i]; //后移元素
        A[i+1]=x;                      //插入x
    } //结束插入
}
```

本题的算法也可以写成三个函数: 查找函数、交换后继函数与插入函数。写成三个函数的优点是逻辑清晰、易读。

10. 解答:

(1) 算法的基本设计思想:

可以将这个问题看做是把数组 ab 转换成数组 ba (a 代表数组的前 p 个元素, b 代表数组中余下的 $n-p$ 个元素), 先将 a 逆置得到 $a^{-1}b$, 再将 b 逆置得到 $a^{-1}b^{-1}$, 最后将整个 $a^{-1}b^{-1}$ 逆置得到 $(a^{-1}b^{-1})^{-1}=ba$ 。设 Reverse 函数执行将数组元素逆置的操作, 对 $abcdefgh$ 向左循环移动 3 ($p=3$) 个位置的过程如下:

Reverse(0, p-1)得到 cbadefgh;

Reverse(p, n-1)得到 cbahgfed;

Reverse(0, n-1)得到 defghabc;

注: Reverse 中, 两个参数分别表示数组中待转换元素的始末位置。

(2) 使用 C 语言描述算法如下:

```
void Reverse(int R[],int from,int to) {
    int i,temp;
    for(i=0;i<(to-from+1)/2;i++)
    {   temp=R[from+i];R[from+i]=R[to-i];R[to-i]=temp;}
} //Reverse

void Converse(int R[],int n,int p){
    Reverse(R,0,p-1);
    Reverse(R,p,n-1);
    Reverse(R,0,n-1);
}
```

(3) 上述算法中三个 Reverse 函数的时间复杂度分别为 $O(p/2)$ 、 $O((n-p)/2)$ 和 $O(n/2)$, 故所设计的算法的时间复杂度为 $O(n)$, 空间复杂度为 $O(1)$ 。

另解, 借助辅助数组来实现:

算法思想: 创建大小为 p 的辅助数组 S , 将 R 中前 p 个整数依次暂存在 S 中, 同时将 R 中后 $n-p$ 个整数左移, 然后将 S 中暂存的 p 个数依次放回到 R 中的后续单元。

时间复杂度为 $O(n)$, 空间复杂度为 $O(p)$ 。

11. 解答:

(1) 算法的基本设计思想如下:

分别求两个升序序列 A 、 B 的中位数, 设为 a 和 b , 求序列 A 、 B 的中位数过程如下:

① 若 $a=b$, 则 a 或 b 即为所求中位数, 算法结束。

② 若 $a < b$, 则舍弃序列 A 中较小的一半, 同时舍弃序列 B 中较大的一半, 要求两次舍弃的长度相等。

③ 若 $a > b$, 则舍弃序列 A 中较大的一半, 同时舍弃序列 B 中较小的一半, 要求两次舍弃的长度相等。

在保留的两个升序序列中, 重复过程 1)、2)、3), 直到两个序列中均只含一个元素时为止, 较小者即为所求的中位数。

(2) 本题代码如下:

```
int M_Search(int A[],int B[],int n){
    int s1=0,d1=n-1,m1,s2=0,d2=n-1,m2;
    //分别表示序列 A 和 B 的首位数、末位数和中位数
    while(s1!=d1||s2!=d2){
        m1=(s1+d1)/2;
        m2=(s2+d2)/2;
        if(A[m1]==B[m2])           //满足条件 1)
            return A[m1];
        if(A[m1]<B[m2]){           //满足条件 2)
            if((s1+d1)%2==0){     //若元素个数为奇数
                s1=m1;           //舍弃 A 中间点以前的部分且保留中间点
                d2=m2;           //舍弃 B 中间点以后的部分且保留中间点
            }
            else{                 //元素个数为偶数
```

```

        s1=m1+1;           //舍弃A中间点及中间点以前部分
        d2=m2;             //舍弃B中间点以后部分且保留中间点
    }
}
else{                      //满足条件3)
    if((s2+d2)%2==0){      //若元素个数为奇数
        d1=m1;             //舍弃A中间点以后的部分且保留中间点
        s2=m2;             //舍弃B中间点以前的部分且保留中间点
    }
    else{                  //元素个数为偶数
        d1=m1;             //舍弃A中间点以后部分且保留中间点
        s2=m2+1;          //舍弃B中间点及中间点以前部分
    }
}
}
return A[s1]<B[s2]? A[s1]:B[s2];
}

```

(3) 算法的时间复杂度为 $O(\log_2 n)$, 空间复杂度为 $O(1)$ 。

12. 解答:

(1) 给出算法的基本设计思想:

算法的策略是从前向后扫描数组元素, 标记出一个可能成为主元素的元素 Num。然后重新计数, 确认 Num 是否是主元素。

算法可分为以下两步:

① 选取候选的主元素: 依次扫描所给数组中的每个整数, 将第一个遇到的整数 Num 保存到 c 中, 记录 Num 的出现次数为 1; 若遇到的下一个整数仍等于 Num, 则计数加 1, 否则计数减 1; 当计数减到 0 时, 将遇到的下一个整数保存到 c 中, 计数重新记为 1, 开始新一轮计数, 即从当前位置开始重复上述过程, 直到扫描完全部数组元素。

② 判断 c 中元素是否是真正的主元素: 再次扫描该数组, 统计 c 中元素出现的次数, 若大于 $n/2$, 则为主元素; 否则, 序列中不存在主元素。

(2) 算法实现:

```

int Majority(int A[],int n){
    int i,c,count=1;        //c 用来保存候选主元素, count 用来计数
    c=A[0];                 //设置 A[0] 为候选主元素
    for(i=1;i<n;i++){       //查找候选主元素
        if(A[i]==c)
            count++;         //对 A 中的候选主元素计数
        else
            if(count>0)      //处理不是候选主元素的情况
                count--;
            else{             //更换候选主元素, 重新计数
                c=A[i];
                count=1;
            }
    }
    if(count>0)
        for(i=count=0;i<n;i++) //统计候选主元素的实际出现次数
            if(A[i]==c)
                count++;
}

```



```

    if(count>n/2) return c;    //确认候选主元素
    else return -1;           //不存在主元素
}

```

(3) 实现的程序的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

【说明】本题如果采用先排序再统计的方法（时间复杂度可为 $O(n\log_2 n)$ ），只要解答正确，最高可拿 11 分。即便是写出 $O(n^2)$ 的算法，最高也能拿 10 分，因此对于统考算法题，去花费大量时间去思考最优解法是得不偿失的。

2.3 线性表的链式表示

由于顺序表的插入、删除操作需要移动大量的元素，影响了运行效率，由此引入了线性表的链式存储。链式存储线性表时，不需要使用地址连续的存储单元，即它不要求逻辑上相邻的两个元素在物理位置上也相邻，它是通过“链”建立起数据元素之间的逻辑关系，因此，对线性表的插入、删除不需要移动元素，而只需要修改指针。

2.3.1 单链表的定义

线性表的链式存储又称为**单链表**，它是指通过一组任意的存储单元来存储线性表中的数据元素。为了建立起数据元素之间的线性关系，对每个链表结点，除了存放元素自身的信息之外，还需要存放一个指向其后继的指针。单链表结点结构如图 2-3 所示，其中，data 为数据域，存放数据元素；next 为指针域，存放其后继结点的地址。

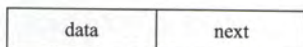


图 2-3 单链表结点结构

单链表中结点类型的描述如下：

```

typedef struct LNode{           //定义单链表结点类型
    ElemType data;              //数据域
    struct LNode *next;         //指针域
}LNode, *LinkList;

```

利用单链表可以解决顺序表需要大量的连续存储空间缺点，但是单链表附加指针域，也存在浪费存储空间的缺点。由于单链表的元素是离散地分布在存储空间中的，所以单链表是**非随机存取**的存储结构，即不能直接找到表中某个特定的结点。查找某个特定的结点时，需要从表头开始遍历，依次查找。

通常用“**头指针**”来标识一个单链表，如单链表 L，头指针为“NULL”时则表示一个空表。此外，为了操作上的方便，在单链表第一个结点之前附加一个结点，称为**头结点**。头结点的数据域可以不设任何信息，也可以记录表长等相关信息。头结点的指针域指向线性表的第一个元素结点，如图 2-4 所示。

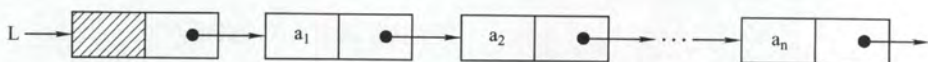


图 2-4 带头结点的单链表

头结点和头指针的区分：不管带不带头结点，头指针始终指向链表的第一个结点，而头结点是带头结点链表中的第一个结点，结点内通常不存储信息。

引入头结点后，可以带来两个优点：

- ① 由于开始结点的位置被存放在头结点的指针域中，所以在链表的第一个位置上的操作和

在表的其他位置上的操作一致, 无须进行特殊处理。

② 无论链表是否为空, 其头指针是指向头结点的非空指针 (空表中头结点的指针域为空), 因此空表和非空表的处理也就统一了。

2.3.2 单链表上基本操作的实现

1. 采用头插法建立单链表

该方法从一个空表开始, 生成新结点, 并将读取到的数据存放到新结点的数据域中, 然后将新结点插入到当前链表的表头, 即头结点之后, 如图 2-5 所示。

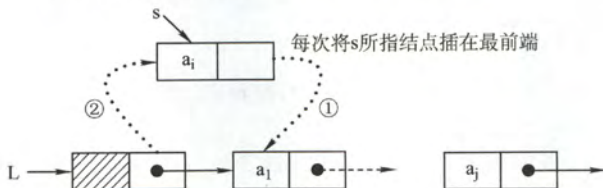


图 2-5 头插法建立单链表

头插法建立单链表的算法如下:

```
LinkList CreatList1(LinkList &L){
//从表尾到表头逆向建立单链表 L, 每次均在头结点之后插入元素
    LNode *s; int x;
    L=(LinkList)malloc(sizeof(LNode)); //创建头结点
    L->next=NULL; //初始为空链表
    scanf("%d",&x); //输入结点的值
    while(x!=9999){ //输入 9999 表示结束
        s=(LNode*)malloc(sizeof(LNode)); //创建新结点
        s->data=x;
        s->next=L->next;
        L->next=s; //将新结点插入表中, L 为头指针
        scanf("%d",&x);
    } //while 结束
    return L;
}
```

采用头插法建立单链表, 读入数据的顺序与生成的链表中元素的顺序是相反的。每个结点插入的时间为 $O(1)$, 设单链表长为 n , 则总的时间复杂度为 $O(n)$ 。

2. 采用尾插法建立单链表

头插法建立单链表的算法虽然简单, 但生成的链表中结点的次序和输入数据的顺序不一致。若希望两者次序一致, 可采用尾插法。该方法是将新结点插入到当前链表的表尾上, 为此必须增加一个尾指针 r , 使其始终指向当前链表的尾结点, 如图 2-6 所示。

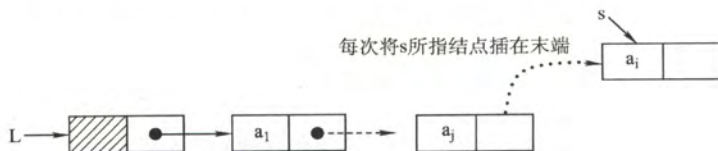


图 2-6 尾插法建立单链表

尾插法建立单链表的算法如下:


```

LinkedList CreatList2(LinkedList &L){
    //从表头到表尾正向建立单链表 L, 每次均在表尾插入元素
    int x; //设元素类型为整型
    L=(LinkedList)malloc(sizeof(LNode));
    LNode *s,*r=L; //r 为表尾指针
    scanf("%d",&x); //输入结点的值
    while(x!=9999){ //输入 9999 表示结束
        s=(LNode *)malloc(sizeof(LNode));
        s->data=x;
        r->next=s;
        r=s; //r 指向新的表尾结点
        scanf("%d",&x);
    }
    r->next=NULL; //尾结点指针置空
    return L;
}

```

因为附设了一个指向表尾结点的指针, 故时间复杂度和头插法相同。

3. 按序号查找结点值

在单链表中从第一个结点出发, 顺指针 next 域逐个往下搜索, 直到找到第 i 个结点为止, 否则返回最后一个结点指针域 NULL。

按序号查找结点值的算法如下:

```

LNode *GetElem(LinkedList L,int i){
    //本算法取出单链表 L (带头结点) 中第 i 个位置的结点指针
    int j=1; //计数, 初始为 1
    LNode *p=L->next; //头结点指针赋给 p
    if(i==0)
        return L; //若 i 等于 0, 则返回头结点
    if(i<1)
        return NULL; //若 i 无效, 则返回 NULL
    while(p&&j<i){ //从第 1 个结点开始找, 查找第 i 个结点
        p=p->next;
        j++;
    }
    return p; //返回第 i 个结点的指针, 如果 i 大于表长, p=NULL, 直接返回 p 即可
}

```

按序号查找操作的时间复杂度为 $O(n)$ 。

4. 按值查找表结点

从单链表第一个结点开始, 由前往后依次比较表中各结点数据域的值, 若某结点数据域的值等于给定值 e , 则返回该结点的指针; 若整个单链表中没有这样的结点, 则返回 NULL。

按值查找结点的算法如下:

```

LNode *LocateElem(LinkedList L,ElemType e){
    //本算法查找单链表 L (带头结点) 中数据域值等于 e 的结点指针, 否则返回 NULL
    LNode *p=L->next;
    while(p!=NULL&&p->data!=e) //从第 1 个结点开始查找 data 域为 e 的结点
        p=p->next;
    return p; //找到后返回该结点指针, 否则返回 NULL
}

```

按值查找操作的时间复杂度为 $O(n)$ 。

5. 插入结点操作

插入操作是将值为 x 的新结点插入到单链表的第 i 个位置上。先检查插入位置的合法性，然后找到待插入位置的前驱结点，即第 $i-1$ 个结点，再在其后插入新结点。

算法首先调用 (3) 中的按序号查找算法 $\text{GetElem}(L, i-1)$ ，查找第 $i-1$ 个结点。假设返回的第 $i-1$ 个结点为 $*p$ ，然后令新结点 $*s$ 的指针域指向 $*p$ 的后继结点，再令结点 $*p$ 的指针域指向新插入的结点 $*s$ 。其操作过程如图 2-7 所示。

实现插入结点的代码片段如下：

```
① p=GetElem(L, i-1);           // 查找插入位置的前驱结点
② s->next=p->next;             // 图 2-7 中操作步骤 1
③ p->next=s;                   // 图 2-7 中操作步骤 2
```

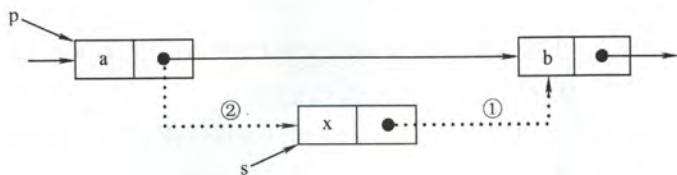


图 2-7 单链表的插入操作

算法中，语句②③的顺序不能颠倒，否则，当先执行 $p \rightarrow next = s$ 后，指向其原后继的指针就不存在了，再执行 $s \rightarrow next = p \rightarrow next$ 时，相当于执行了 $s \rightarrow next = s$ ，显然是错误的。本算法主要的时间开销在于查找第 $i-1$ 个元素，时间复杂度为 $O(n)$ 。若是在给定的结点后面插入新结点，则时间复杂度仅为 $O(1)$ 。

扩展：对某一结点进行前插操作

前插操作是指在某结点的前面插入一个新结点，后插操作的定义刚好与之相反，在单链表插入算法中，通常都是采用后插操作的。

以上面的算法为例，首先调用函数 $\text{GetElem}()$ 找到第 $i-1$ 个结点，即待插入结点的前驱结点后，再对其执行后插操作。由此可知，对结点的前插操作均可以转化为后插操作，前提是从单链表的头结点开始顺序查找找到其前驱结点，时间复杂度为 $O(n)$ 。

此外，可以采用另一种方式将其转化为后插操作来实现，设待插入结点为 $*s$ ，将 $*s$ 插入到 $*p$ 的前面。我们仍然将 $*s$ 插入到 $*p$ 的后面，然后将 $p \rightarrow data$ 与 $s \rightarrow data$ 交换即可，这样既满足了逻辑关系，又能使得时间复杂度为 $O(1)$ 。算法的代码片段如下：

```
// 将*s结点插入到*p之前的主要代码片段
s->next=p->next;           // 修改指针域，不能颠倒
p->next=s;
temp=p->data;              // 交换数据域部分
p->data=s->data;
s->data=temp;
```

6. 删除结点操作

删除操作是将单链表的第 i 个结点删除。先检查删除位置的合法性，然后查找表中第 $i-1$ 个结点，即被删结点的前驱结点，再将其删除。其操作过程如图 2-8 所示。

假设结点 $*p$ 为找到的被删结点的前驱结点，为了实现这一操作后的逻辑关系的变化，仅需修改 $*p$ 的指针域，即将 $*p$ 的指针域 $next$ 指向 $*q$ 的下一结点。

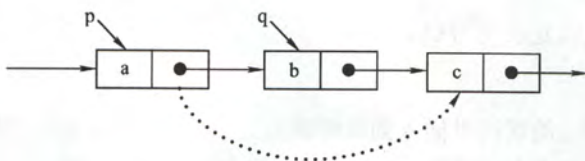


图 2-8 单链表结点的删除

实现删除结点的代码片段如下:

```
p=GetElem(L,i-1);           //查找删除位置的前驱结点
q=p->next;                   //令 q 指向被删除结点
p->next=q->next               //将*q 结点从链中“断开”
free(q);                     //释放结点的存储空间
```

和插入算法一样,该算法的主要时间也是耗费在查找操作上,时间复杂度为 $O(n)$ 。

扩展:删除结点*p

要实现删除某一个给定结点*p,通常的做法是先从链表的头结点开始顺序找到其前驱结点,然后再执行删除操作即可,算法的时间复杂度为 $O(n)$ 。

其实,删除结点*p的操作可以用删除*p的后继结点操作来实现,实质就是将其后继结点的值赋予其自身,然后删除后继结点,也能使得时间复杂度为 $O(1)$ 。

实现上述操作的代码片段如下:

```
q=p->next;                   //令 q 指向*p 的后继结点
p->data=p->next->data;       //和后继结点交换数据域
p->next=q->next;             //将*q 结点从链中“断开”
free(q);                     //释放后继结点的存储空间
```

7. 求表长操作

求表长操作就是计算单链表中数据结点(不含头结点)的个数,需要从第一个结点开始顺序依次访问表中的每一个结点,为此需要设置一个计数器变量,每访问一个结点,计数器加 1,直到访问到空结点为止。算法的时间复杂度为 $O(n)$ 。

需要注意的是,因为单链表的长度是不包括头结点的,因此,不带头结点和带头结点的单链表在求表长操作上会略有不同。对不带头结点的单链表,当表为空时,要单独处理。

单链表是整个链表的基础,读者一定要熟练掌握单链表的基本操作算法,在设计算法时,建议先通过图示的方法理清算法的思路,然后再进行算法的编写。

2.3.3 双链表

单链表结点中只有一个指向其后继的指针,这使得单链表只能从头结点依次顺序地向后遍历。若要访问某个结点的前驱结点(插入、删除操作时),只能从头开始遍历,访问后继结点的时间复杂度为 $O(1)$,访问前驱结点的时间复杂度为 $O(n)$ 。

为了克服单链表的上述缺点,引入了双链表,双链表结点中有两个指针 prior 和 next,分别指向其前驱结点和后继结点,如图 2-9 所示。

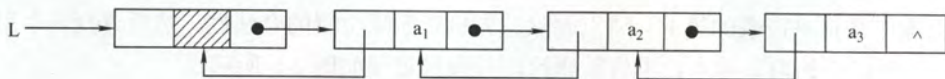


图 2-9 双链表示意图

双链表中结点类型的描述如下:

```
typedef struct DNode{           //定义双链表结点类型
    ElemType data;             //数据域
    struct DNode *prior,*next; //前驱和后继指针
}DNode, *DLinklist;
```

双链表仅仅是在单链表的结点中增加了一个指向其前驱的 *prior* 指针, 因此, 在双链表中执行按值查找和按位查找的操作和单链表相同。但双链表在插入和删除操作的实现上, 和单链表有着较大的不同。这是因为“链”变化时也需要对 *prior* 指针做出修改, 其关键在于保证在修改的过程中不断链。此外, 双链表可以很方便地找到其前驱结点, 因此, 插入、删除结点算法的时间复杂度仅为 $O(1)$ 。

1. 双链表的插入操作

在双链表中 *p* 所指的结点之后插入结点 *s*, 其指针的变化过程如图 2-10 所示。

插入操作的代码片段如下:

```
① s->next=p->next;           //将结点*s 插入到结点*p 之后
② p->next->prior=s;
③ s->prior=p;
④ p->next=s;
```

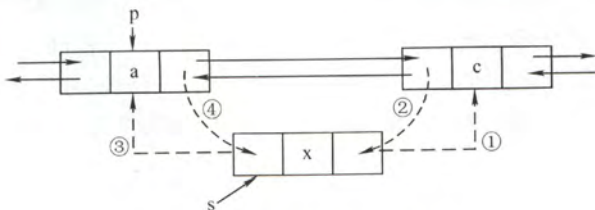


图 2-10 双链表插入结点过程

上述代码的语句顺序不是唯一的, 但也不是任意的, ①②两步必须在④步之前, 否则 *p* 的后继结点的指针就丢掉了, 导致插入失败。为了加深理解, 读者可以在纸上画出示意图。若问题改成要求在结点 *p* 之前插入结点 *s*, 请读者思考具体的操作步骤。

2. 双链表的删除操作

删除双链表中结点 *p* 的后继结点 *q*, 其指针的变化过程如图 2-11 所示。

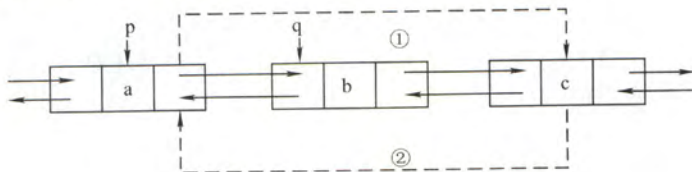


图 2-11 双链表删除结点过程

删除操作的代码片段如下:

```
p->next=q->next;           //图 2-11 中步骤①
q->next->prior=p;          //图 2-11 中步骤②
free(q);                   //释放结点空间
```

若问题改成要求删除结点 *q* 的前驱结点 *p*, 请读者思考具体的操作步骤。

建立双链表的操作中,也可以采用如同单链表的头插法和尾插法,但是在操作上需要注意指针的变化和单链表有所不同。

2.3.4 循环链表

1. 循环单链表

循环单链表和单链表的区别在于,表中最后一个结点的指针不是 NULL,而改为指向头结点,从而整个链表形成一个环,如图 2-12 所示。

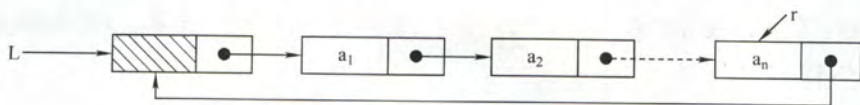


图 2-12 循环单链表

在循环单链表中,表尾结点*r的 next 域指向 L,故表中没有指针域为 NULL 的结点,因此,循环单链表的判空条件不是头结点的指针是否为空,而是它是否等于头指针。

循环单链表的插入、删除算法与单链表的几乎一样,所不同的是如果操作是在表尾进行,则执行的操作不相同,以让单链表继续保持循环的性质。当然,正是因为循环单链表是一个“环”,因此,在任何一个位置上的插入和删除操作都是等价的,无须判断是否是表尾。

在单链表中只能从表头结点开始往后顺序遍历整个链表,而循环单链表可以从表中的任意结点开始遍历整个链表。有时对单链表常做的操作是在表头和表尾进行的,此时可对循环单链表不设头指针而仅设尾指针,从而使得操作效率更高。其原因是若设的是头指针,对表尾进行操作需要 $O(n)$ 的时间复杂度,而如果设的是尾指针 r , $r \rightarrow \text{next}$ 即为头指针,对于表头与表尾进行操作都只需要 $O(1)$ 的时间复杂度。

2. 循环双链表

由循环单链表的定义不难推出循环双链表,不同的是在循环双链表中,头结点的 prior 指针还要指向表尾结点,如图 2-13 所示。

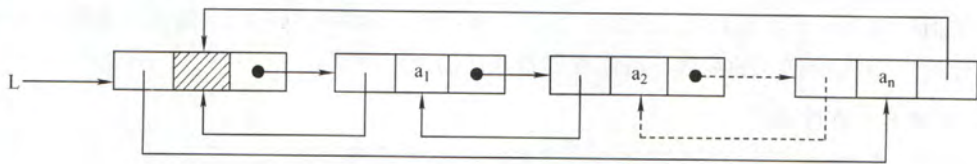


图 2-13 循环双链表

在循环双链表 L 中,某结点*p为尾结点时, $p \rightarrow \text{next} = L$;当循环双链表为空表时,其头结点的 prior 域和 next 域都等于 L。

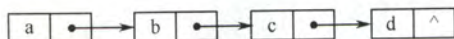
2.3.5 静态链表

静态链表是借助数组来描述线性表的链式存储结构,结点也有数据域 data 和指针域 next,与前面所讲的链表中的指针不同的是,这里的指针是结点的相对地址(数组下标),又称为游标。和顺序表一样,静态链表也要预先分配一块连续的内存空间。

静态链表和单链表的对应关系如图 2-14 所示。

0		2
1	b	6
2	a	1
3	d	-1
4		
5		
6	c	3

(a)静态链表表示例



(b)静态链表对应的单链表

图 2-14 静态链表存储示意图

静态链表结构类型的描述如下:

```

#define MaxSize 50          //静态链表的最大长度
typedef struct {             //静态链表结构类型的定义
    ElemType data;          //存储数据元素
    int next;                //下一个元素的数组下标
} SLinkList[MaxSize];
  
```

静态链表以 $\text{next} = -1$ 作为其结束的标志。静态链表的插入、删除操作与动态链表相同, 只需要修改指针, 而不需要移动元素。总体来说, 静态链表没有单链表使用起来方便, 但是在一些不支持指针的高级语言 (如 Basic) 中, 这又是一种非常巧妙的设计方法。

2.3.6 顺序表和链表的比较

1. 存取方式

顺序表可以顺序存取, 也可以随机存取, 链表只能从表头顺序存取元素。

2. 逻辑结构与物理结构

采用顺序存储时, 逻辑上相邻的元素, 其对应的物理存储位置也相邻。而采用链式存储时, 逻辑上相邻的元素, 其物理存储位置则不一定相邻, 其对应的逻辑关系是通过指针链接来表示的。这里请读者注意区别存取方式和存储方式。

3. 查找、插入和删除操作

对于按值查找, 当顺序表在无序的情况下, 两者的时间复杂度均为 $O(n)$; 而当顺序表有序时, 可采用折半查找, 此时时间复杂度为 $O(\log_2 n)$ 。

对于按序号查找, 顺序表支持随机访问, 时间复杂度仅为 $O(1)$, 而链表的平均时间复杂度为 $O(n)$ 。顺序表的插入、删除操作, 平均需要移动半个表长的元素。链表的插入、删除操作, 只需要修改相关结点的指针域即可。由于链表每个结点带有指针域, 因而在存储空间上比顺序存储要付出较大的代价, 存储密度不够大。

4. 空间分配

顺序存储在静态存储分配情形下, 一旦存储空间装满就不能扩充, 如果再加入新元素将出现内存溢出, 需要预先分配足够大的存储空间。预先分配过大, 可能会导致顺序表后部大量闲置; 预先分配过小, 又会造成溢出。动态存储分配虽然存储空间可以扩充, 但需要移动大量元素, 导致操作效率降低, 而且若内存中没有更大块的连续存储空间将导致分配失败。链式存储的结点空间只在需要的时候申请分配, 只要内存有空间就可以分配, 操作灵活、高效。

在实际中应该怎样选取存储结构呢?

1. 基于存储的考虑

对线性表的长度或存储规模难以估计时,不宜采用顺序表;链表不用事先估计存储规模,但链表的存储密度较低,显然链式存储结构的存储密度是小于1的。

2. 基于运算的考虑

在顺序表中按序号访问 a_i 的时间复杂度为 $O(1)$,而链表中按序号访问的时间复杂度为 $O(n)$,所以如果经常做的运算是按序号访问数据元素,显然顺序表优于链表。

在顺序表中做插入、删除操作时,平均移动表中一半的元素,当数据元素的信息量较大且表较长时,这一点是不应忽视的;在链表中做插入、删除操作时,虽然也要找插入位置,但操作主要是比较操作,从这个角度考虑显然后者优于前者。

3. 基于环境的考虑

顺序表容易实现,任何高级语言中都有数组类型;链表的操作是基于指针的,相对来讲,前者实现较为简单,这也是用户考虑的一个因素。

总之,两种存储结构各有长短,选择哪一种由实际问题的主要因素决定。通常较稳定的线性表选择顺序存储,而频繁做插入、删除操作的线性表(即动态性较强)宜选择链式存储。

注意:只有熟练掌握顺序存储和链式存储,才能深刻理解它们各自的优缺点。

2.3.7 本节试题精选

一、单项选择题

1. 关于线性表的顺序存储结构和链式存储结构的描述中,正确的是()。

- I. 线性表的顺序存储结构优于其链式存储结构
- II. 链式存储结构比顺序存储结构能更方便地表示各种逻辑结构
- III. 如频繁使用插入和删除结点操作,顺序存储结构更优于链式存储结构
- IV. 顺序存储结构和链式存储结构都可以进行顺序存取

A. I、II、III B. II、IV C. II、III D. III、IV

2. 对于一个线性表既要求能够进行较快速地插入和删除,又要求存储结构能反映数据之间的逻辑关系,则应该用()。

- A. 顺序存储方式
- B. 链式存储方式
- C. 散列存储方式
- D. 以上均可以

3. 对于顺序存储的线性表,其算法的时间复杂度为 $O(1)$ 的运算应该是()

- A. 将 n 个元素从小到大排序
- B. 删除第 i 个元素 ($1 \leq i \leq n$)
- C. 改变第 i 个元素的值 ($1 \leq i \leq n$)
- D. 在第 i 个元素后插入一个新元素 ($1 \leq i \leq n$)

4. 下列关于线性表说法正确的是()。

- I. 顺序存储方式只能用于存储线性结构
- II. 取线性表的第 i 个元素的时间与 i 的大小有关
- III. 静态链表需要分配较大的连续空间,插入和删除不需要移动元素
- IV. 在一个长度为 n 的有序单链表中插入一个新结点并仍保持有序的时间复杂度为 $O(n)$
- V. 若用单链表来表示队列,则应该选用带尾指针的循环链表

A. I、II B. I、III、IV、V

D. $p \rightarrow \text{prior} \rightarrow \text{next} = q; q \rightarrow \text{next} = p; q \rightarrow \text{prior} = p \rightarrow \text{prior}; p \rightarrow \text{prior} = q;$

15. 在双向链表存储结构中, 删除 p 所指的结点时必须修改指针 ()。

A. $p \rightarrow \text{llink} \rightarrow \text{rlink} = p \rightarrow \text{rlink}; p \rightarrow \text{rlink} \rightarrow \text{llink} = p \rightarrow \text{llink};$

B. $p \rightarrow \text{llink} = p \rightarrow \text{llink} \rightarrow \text{llink}; p \rightarrow \text{llink} \rightarrow \text{rlink} = p;$

C. $p \rightarrow \text{rlink} \rightarrow \text{llink} = p; p \rightarrow \text{rlink} = p \rightarrow \text{rlink} \rightarrow \text{rlink};$

D. $p \rightarrow \text{rlink} = p \rightarrow \text{llink} \rightarrow \text{llink}; p \rightarrow \text{llink} = p \rightarrow \text{rlink} \rightarrow \text{rlink};$

16. 【2016 年计算机联考真题】

已知一个带有表头结点的双向循环链表 L , 结点结构为

prev	data	next
------	------	------

, 其中, prev 和 next 分别是指向其直接前驱和直接后继结点的指针。现要删除指针 p 所指的结点, 正确的语句序列是 ()。

A. $p \rightarrow \text{next} \rightarrow \text{prev} = p \rightarrow \text{prev}; p \rightarrow \text{prev} \rightarrow \text{next} = p \rightarrow \text{prev}; \text{free}(p);$

B. $p \rightarrow \text{next} \rightarrow \text{prev} = p \rightarrow \text{next}; p \rightarrow \text{prev} \rightarrow \text{next} = p \rightarrow \text{next}; \text{free}(p);$

C. $p \rightarrow \text{next} \rightarrow \text{prev} = p \rightarrow \text{next}; p \rightarrow \text{prev} \rightarrow \text{next} = p \rightarrow \text{prev}; \text{free}(p);$

D. $p \rightarrow \text{next} \rightarrow \text{prev} = p \rightarrow \text{prev}; p \rightarrow \text{prev} \rightarrow \text{next} = p \rightarrow \text{next}; \text{free}(p);$

17. 在长度为 n 的有序单链表中插入一个新结点, 并仍然保持有序的时间复杂度是 ()。

A. $O(1)$

B. $O(n)$

C. $O(n^2)$

D. $O(n \log_2 n)$

18. 与单链表相比, 双链表的优点之一是 ()。

A. 插入、删除操作更方便

B. 可以进行随机访问

C. 可以省略表头指针或表尾指针

D. 访问前后相邻结点更灵活

19. 带头结点的双循环链表 L 为空的条件是 ()。

A. $L \rightarrow \text{prior} = L \&\& L \rightarrow \text{next} = \text{NULL}$

B. $L \rightarrow \text{prior} = \text{NULL} \&\& L \rightarrow \text{next} = \text{NULL}$

C. $L \rightarrow \text{prior} = \text{NULL} \&\& L \rightarrow \text{next} = L$

D. $L \rightarrow \text{prior} = L \&\& L \rightarrow \text{next} = L$

20. 一个链表最常用的操作是在末尾插入结点和删除结点, 则选用 () 最节省时间。

A. 带头结点的双循环链表

B. 单循环链表

C. 带尾指针的单循环链表

D. 单链表

21. 设对 $n(n > 1)$ 个元素的线性表的运算只有 4 种: 删除第一个元素; 删除最后一个元素; 在第一个元素之前插入新元素; 在最后一个元素之后插入新元素, 则最好使用 ()。

A. 只有尾结点指针没有头结点指针的循环单链表

B. 只有尾结点指针没有头结点指针的非循环双链表

C. 只有头结点指针没有尾结点指针的循环双链表

D. 既有头结点指针也有尾结点指针的循环单链表

22. 一个链表最常用的操作是在最后一个元素之后插入一个元素和删除第一个元素, 则选用 () 最节省时间。

A. 不带头结点的单循环链表

B. 双链表

C. 不带头结点且有尾指针的单循环链表

D. 单链表

23. 静态链表中指针表示的是 ()。

A. 下一元素的地址

B. 内存地址

C. 下一个元素在数组中的位置

D. 左链或右链指向的元素的地址

24. 【2016 年计算机联考真题】

已知表头元素为 c 的单链表在内存中的存储状态如下表所示。

地址	元素	链接地址
1000H	a	1010H
1004H	b	100CH
1008H	c	1000H
100CH	d	NULL
1010H	e	1004H
1014H		

现将f存放于1014H处并插入到单链表中,若f在逻辑上位于a和e之间,则a,e,f的“链接地址”依次是()。

- A. 1010H, 1014H, 1004H B. 1010H, 1004H, 1014H
C. 1014H, 1010H, 1004H D. 1014H, 1004H, 1010H
25. 需要分配较大的空间,插入和删除不需要移动元素的线性表,其存储结构为()。
A. 单链表 B. 静态链表 C. 顺序表 D. 双链表

二、综合应用题

- 设计一个递归算法,删除不带头结点的单链表L中所有值为x的结点。
- 在带头结点的单链表L中,删除所有值为x的结点,并释放其空间,假设值为x的结点不唯一,试编写算法以实现上述操作。
- 设L为带头结点的单链表,编写算法实现从尾到头反向输出每个结点的值。
- 试编写在带头结点的单链表L中删除一个最小值结点的高效算法(假设最小值结点是唯一的)。
- 试编写算法将带头结点的单链表就地逆置,所谓“就地”是指辅助空间复杂度为 $O(1)$ 。
- 有一个带头结点的单链表L,设计一个算法使其元素递增有序。
- 设在一个带头结点的单链表中所有元素结点的数据值无序,试编写一个函数,删除表中所有介于给定的两个值(作为函数参数给出)之间的元素的元素(若存在)。
- 给定两个单链表,编写算法找出两个链表的公共结点。
- 给定一个带头结点的单链表,设head为头指针,结点的结构为(data, next),data为整型元素,next为指针,试写出算法:按递增次序输出单链表中各结点的数据元素,并释放结点所占的存储空间(要求:不允许使用数组作为辅助空间)。
- 将一个带头结点的单链表A分解为两个带头结点的单链表A和B,使得A表中含有原表中序号为奇数的元素,而B表中含有原表中序号为偶数的元素,且保持其相对顺序不变。
- 设 $C=\{a_1, b_1, a_2, b_2, \dots, a_n, b_n\}$ 为线性表,采用带头结点的hc单链表存放,设计一个就地算法,将其拆分为两个线性表,使得
 $A=\{a_1, a_2, \dots, a_n\}$, $B=\{b_n, \dots, b_2, b_1\}$
- 在一个递增有序的线性表中,有数值相同的元素存在。若存储方式为单链表,设计算法去掉数值相同的元素,使表中不再有重复的元素。例如(7, 10, 10, 21, 30, 42, 42, 42, 51, 70)将变作(7, 10, 21, 30, 42, 51, 70)。
- 假设有两个按元素值递增次序排列的线性表,均以单链表形式存储。请编写算法将这两个单链表归并为一个按元素值递减次序排列的单链表,并要求利用原来两个单链表的结点存放归并后的单链表。
- 设A和B是两个单链表(带头结点),其中元素递增有序。设计一个算法从A和B中公共元素产生单链表C,要求不破坏A、B的结点。
- 已知两个链表A和B分别表示两个集合,其元素递增排列。编制函数,求A与B的交

集, 并存放于 A 链表中。

16. 两个整数序列 $A=a_1, a_2, a_3, \dots, a_m$ 和 $B=b_1, b_2, b_3, \dots, b_n$ 已经存入两个单链表中, 设计一个算法, 判断序列 B 是否是序列 A 的连续子序列。

17. 设计一个算法用于判断带头结点的循环双链表是否对称。

18. 有两个循环单链表, 链表头指针分别为 h_1 和 h_2 , 编写一个函数将链表 h_2 链接到链表 h_1 之后, 要求链接后的链表仍保持循环链表形式。

19. 设有一个带头结点的循环单链表, 其结点值均为正整数。设计一个算法, 反复找出单链表中结点值最小的结点并输出, 然后将该结点从中删除, 直到单链表空为止, 再删除表头结点。

20. 设头指针为 L 的带有表头结点的非循环双向链表, 其每个结点中除有 $pred$ (前驱指针)、 $data$ (数据) 和 $next$ (后继指针) 域外, 还有一个访问频度域 $freq$ 。在链表被启用前, 其值均初始化为零。每当在链表中进行一次 $Locate(L, x)$ 运算时, 令元素值为 x 的结点中 $freq$ 域的值增 1, 并使此链表中结点保持按访问频度非增 (递减) 的顺序排列, 同时最近访问的结点排在频度相同的结点的前面, 以便使频繁访问的结点总是靠近表头。试编写符合上述要求的 $Locate(L, x)$ 运算的算法, 该运算为函数过程, 返回找到结点的地址, 类型为指针型。

21. 【2009 年计算机联考真题】

已知一个带有表头结点的单链表, 结点结构为

data	link
------	------

假设该链表只给出了头指针 $list$ 。在不改变链表的前提下, 请设计一个尽可能高效的算法, 查找链表中倒数第 k 个位置上的结点 (k 为正整数)。若查找成功, 算法输出该结点的 $data$ 域的值, 并返回 1; 否则, 只返回 0。要求:

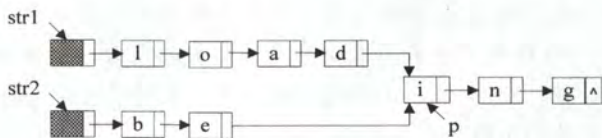
1) 描述算法的基本设计思想。

2) 描述算法的详细实现步骤。

3) 根据设计思想和实现步骤, 采用程序设计语言描述算法 (使用 C、C++ 或 Java 语言实现), 关键之处请给出简要注释。

22. 【2012 年计算机联考真题】

假定采用带头结点的单链表保存单词, 当两个单词有相同的后缀时, 则可共享相同的后缀存储空间, 例如, “loading” 和 “being” 的存储映像如下图所示。



设 str_1 和 str_2 分别指向两个单词所在单链表的头结点, 链表结点结构为

data	next
------	------

, 请设计一个时间上尽可能高效的算法, 找出由 str_1 和 str_2 所指向两个链表共同后缀的起始位置 (如图中字符 i 所在结点的位置 p)。要求:

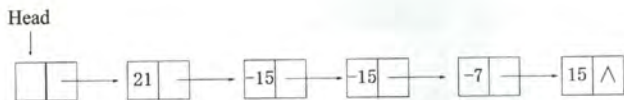
1) 给出算法的基本设计思想。

2) 根据设计思想, 采用 C 或 C++ 或 Java 语言描述算法, 关键之处给出注释。

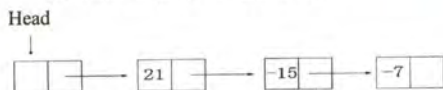
3) 说明你所设计算法的时间复杂度。

23. 【2015 年计算机联考真题】

用单链表保存 m 个整数, 结点的结构为: $[data][link]$, 且 $|data| \leq n$ (n 为正整数)。现要求设计一个时间复杂度尽可能高效的算法, 对于链表中 $data$ 的绝对值相等的结点, 仅保留第一次出现的结点而删除其余绝对值相等的结点。例如, 若给定的单链表 $head$ 如下:



则删除结点后的 head 为:



要求:

- 1) 给出算法的基本设计思想。
- 2) 使用 C 或 C++ 语言, 给出单链表结点的数据类型定义。
- 3) 根据设计思想, 采用 C 或 C++ 语言描述算法, 关键之处给出注释。
- 4) 说明你所设计算法的时间复杂度和空间复杂度。

2.3.8 答案与解析

一、单项选择题

1. B

两种存储结构有不同的适用场合, 不能简单地谁好谁坏, I 错误。链式存储用指针表示逻辑结构, 而指针的设置是任意的, 故可以很方便地表示各种逻辑结构; 顺序存储则只能用物理上的邻接关系来表示逻辑结构, II 正确。在顺序存储中, 插入和删除结点需要大量的移动元素, 效率较低, III 的描述刚好相反。顺序存储结构既可以随机存取也能顺序存取, 而链式结构只能进行顺序存取, IV 正确。

2. B

顺序存储的插入和删除时间复杂度都是 $O(n)$, 查找最优的时间复杂度是 $O(\log_2 n)$ 。要求较快地插入和删除, 排除 A、D。散列存储通过散列函数映射到物理空间, 不能反映数据之间的逻辑关系, 排除 C。链式存储中的静态链表满足这一要求。链式存储的插入和删除时间复杂度是 $O(1)$, 查找时间复杂度是 $O(n)$ 。

3. C

对 n 个元素进行排序的时间复杂度最快也要 $O(n)$ (初始有序), 通常是 $O(n \log_2 n)$ 或 $O(n^2)$ 。在顺序表中删除第 i 个元素, 或在第 i 个元素之后插入一个新元素, 如想保持其他元素原来的顺序, 时间复杂度为 $O(n)$, 因此 A、B、D 均错误。在顺序存储的线性表中更改第 i 个元素的值, 直接用下标访问并重新赋值即可, 时间复杂度为 $O(1)$ 。

4. D

顺序存储方式同样适合图和树的存储, 如满二叉树的顺序存储, I 错误。若线性表采用顺序存储方式, 则取其第 i 个元素的时间与 i 的大小无关, II 错误。III 是静态链表的特有性质。单链表只能顺序查找插入位置, 时间复杂度为 $O(n)$, 若为顺序表, 可采用折半查找, 时间复杂度可降至 $O(\log_2 n)$, IV 正确。队列需要在表头删除元素, 表尾插入元素, 故采用带尾指针的循环单链表较为方便, 插入和删除的时间复杂度都是 $O(1)$, V 正确。

5. A

A 中对于单链表和顺序表上实现的时间复杂度都为 $O(n)$, 但后者要移动很多元素, 所以在单链表上实现效率更高。B 和 D 效率刚好相反, C 无区别。

6. C

s 插入后, q 成为 s 的前驱, 而 p 成为 s 的后继, 选项 C 满足此条件。

7. D

若先建立链表, 然后依次直接插入建立有序表, 则每插入一个元素就需遍历链表寻找插入位置, 此即链表的插入排序, 时间复杂度为 $O(n^2)$ 。若先将数组排好序, 然后建立链表, 建立链表的时间复杂度为 $O(n)$, 而数组排序的最少时间复杂度为 $O(n\log_2 n)$, 故时间复杂度为 $O(n\log_2 n)$ 。本题问最低时间复杂度, 故选 D。

8. C

先遍历长度为 m 的单链表, 找到这个长度为 m 的单链表的尾结点, 然后将其 next 域置为另一个单链表的首结点, 其时间复杂度为 $O(m)$ 。

9. C

单链表设置头结点的目的是为了便于运算的实现, 主要好处体现在: 第一, 有头结点后, 插入和删除数据元素的算法统一了, 不再需要判断是否在第一个元素之前插入或删除第一个元素; 第二, 不论链表是否为空, 其头指针是指向头结点的非空指针, 链表的头指针不变, 因此空表和非空表的处理也就统一了。

10. B

删除单链表的最后一个结点需要置其前驱结点的指针域为 NULL, 故需要的时间为 $O(n)$, 与表长有关。其他操作均与表长无关, 读者可以自行模拟。

11. B, A

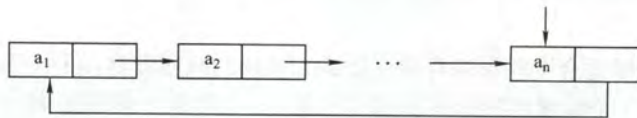
在带头结点的单链表中, 头指针 head 指向头结点, 头结点的 next 域指向第 1 个元素结点, head->next==NULL 表示该单链表为空。在不带头结点的单链表中, head 直接指向第 1 个元素结点, head==NULL 表示该单链表为空。

12. C

双链表能很方便地访问前驱和后继, 故删除和插入数据较为方便。A 显然错误。B 表中第一个元素和最后一个元素不满足题设要求。D 未考虑顺序存储的情况。B、C、D 在删除尾结点时, 都需要先查找其前驱结点, 时间复杂度为 $O(n)$ 。

13. D

在最后一个元素之后插入元素, 需要先找到最后一个元素, 故 A、B 和 C 的时间复杂度均为 $O(n)$ 。B 因为没有特殊的指针指示头结点或尾结点, 故需从某一结点向固定的方向顺序依次搜索插入和删除的位置, 时间复杂度也为 $O(n)$ 。D 的两种算法的时间复杂度都是 $O(1)$, 如下图所示。



14. D

为了在 p 之前插入结点 q, 可以将 p 的前一个结点的 next 域指向 q, 将 q 的 next 域指向 p, 将 q 的 prior 域指向 p 的前一个结点, 将 p 的 prior 域指向 q。仅 D 满足条件。

15. A

与上一题的分析基本类似, 只不过这里是删除一个结点, 注意将 p 的前、后两结点链接起来。关键是要保证在链表的修改过程中不断链!

注, 请读者仔细对比上述两题, 弄清双链表的插入和删除的方法。

16. D

此类题的思路万变不离其宗，无论是链表的插入还是删除都必须保证不断链。

17. B

假设单链表递增有序（递减的情况同理），在插入数据为 x 的结点之前，先要在单链表中找到第一个大于 x 的结点的直接前驱 p ，在 p 之后插入该结点。查找过程的时间复杂度为 $O(n)$ ，插入过程的时间复杂度为 $O(1)$ ，因此时间复杂度为 $O(n)$ 。

18. D

在双链表中可以快速访问任何一个结点的前后相邻结点，而在单链表中只能快速访问任何一个结点的后继结点。

19. D

循环双链表 L 判空的条件是头结点（头指针）的 $prior$ 和 $next$ 域都指向它自身。

20. A

在链表的末尾插入和删除一个结点时，需要修改其相邻结点的指针域。而寻找尾结点以及尾结点的前驱结点，只有带头结点的双循环链表所需要的时间最少。

21. C

对于 A，删除尾结点 $*p$ 时，需要找到 $*p$ 的前一个结点，时间复杂度为 $O(n)$ 。对于 B，删除首结点 $*p$ 时，需要找到 $*p$ 结点，这里没有直接给出头结点指针，而通过尾结点的 $prior$ 指针找到 $*p$ 结点的时间复杂度为 $O(n)$ 。对于 D，删除尾结点 $*p$ 时，需要找到 $*p$ 的前一个结点，时间复杂度为 $O(n)$ 。对于 C，执行这 4 种算法的时间复杂度均为 $O(1)$ 。

22. C

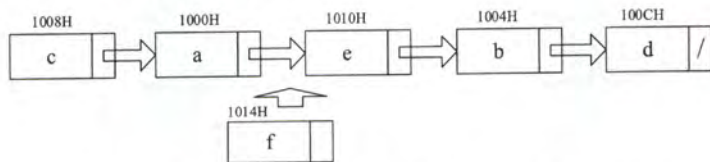
对于 A，在最后一个元素之后插入一个元素的情况与普通单链表相同，时间复杂度为 $O(n)$ ；而删除表中第一个元素时，为保持单循环链表的性质（尾结点的指针指向第一个结点），需要首先遍历整个链表找到尾结点，然后再执行删除操作，时间复杂度也为 $O(n)$ 。对于 B，双链表的情况与单链表的相同，一个是 $O(n)$ ，一个是 $O(1)$ 。对于 C，与 A 的分析对比，因为已经知道了尾结点的位置，省去了遍历链表的过程，因此插入和删除的时间复杂度均为 $O(1)$ 。对于 D，要在最后一个元素之后插入一个元素，需要遍历整个链表才能找到插入位置，时间复杂度为 $O(n)$ ；删除第一个元素的时间复杂度为 $O(1)$ 。

23. C

静态链表中的指针又称游标，指示下一个元素在数组中的下标。

24. D

根据存储状态，单链表的结构如下图所示：



其中“链接地址”是指结点 $next$ 所指的内存地址。当结点 f 插入后， a 指向 f ， f 指向 e ， e 指向 b 。显然 a 、 e 和 f 的“链接地址”分别是 f 、 b 和 e 的内存地址，即 $1014H$ 、 $1004H$ 和 $1010H$ 。

25. B

静态链表用数组表示，因此需要预先分配较大的连续空间，静态链表同时还具有一般链表的特点，即插入和删除不需要移动元素。

二、综合应用题

1. 解答:

设 $f(L, x)$ 的功能是删除以 L 为首结点指针的单链表中所有值等于 x 的结点, 则显然有 $f(L \rightarrow next, x)$ 的功能是删除以 $L \rightarrow next$ 为首结点指针的单链表中所有值等于 x 的结点。由此, 可以推出递归模型如下:

终止条件: $f(L, x) \equiv$ 不做任何事情;	若 L 为空表
递归主体: $f(L, x) \equiv$ 删除 $*L$ 结点; $f(L \rightarrow next, x)$;	若 $L \rightarrow data == x$
$f(L, x) \equiv f(L \rightarrow next, x)$;	其他情况

本题代码如下:

```
void Del_X_3(Linklist &L, ElemType x) {
    // 递归实现在单链表 L 中删除值为 x 的结点
    LNode *p;                // p 指向待删除结点
    if (L == NULL)            // 递归出口
        return;
    if (L->data == x) {        // 若 L 所指结点的值为 x
        p = L;                // 删除 *L, 并让 L 指向下一结点
        L = L->next;
        free(p);
        Del_X_3(L, x);        // 递归调用
    }
    else                       // 若 L 所指结点的值不为 x
        Del_X_3(L->next, x);  // 递归调用
}
```

算法需要借助一个递归工作栈, 深度为 $O(n)$, 时间复杂度为 $O(n)$ 。有读者认为直接 `free` 掉 p 结点会造成断链, 实际上因为 L 为引用, 是直接对原链表进行操作, 因此不会断链。

2. 解答:

解法一: 用 p 从头至尾扫描单链表, pre 指向 $*p$ 结点的前驱。若 p 所指结点的值为 x , 则删除, 并让 p 移向下一个结点, 否则让 pre 、 p 指针同步后移一个结点。

本题代码如下:

```
void Del_X_1(Linklist &L, ElemType x) {
    // L 为带头结点的单链表, 本算法删除 L 中所有值为 x 的结点
    LNode *p = L->next, *pre = L, *q; // 置 p 和 pre 的初始值
    while (p != NULL) {
        if (p->data == x) {
            q = p;                // q 指向该结点
            p = p->next;
            pre->next = p;        // 删除 *q 结点
            free(q);              // 释放 *q 结点的空间
        }
        else {
            pre = p;              // 否则, pre 和 p 同步后移
            p = p->next;
        } // else
    } // while
}
```

本算法是在无序单链表中删除满足某种条件的所有结点, 这里的条件是结点的值为 x 。实际上, 这个条件是可以任意指定的, 只要修改 `if` 条件即可, 比如, 我们要求删除值介于 `mink` 和 `maxk`

之间的所有结点, 则只需将 if 语句修改为 `if(p->data>mink && p->data<maxk)`。

解法二: 采用尾插法建立单链表。用 `p` 指针扫描 `L` 的所有结点, 当其值不为 `x` 时将其链接到 `L` 之后, 否则将其释放。

本题代码如下:

```
void Del_X_2(Linklist &L, ElemType x) {
    //L 为带头结点的单链表, 本算法删除 L 中所有值为 x 的结点
    LNode *p=L->next, *r=L, *q;    //r 指向尾结点, 其初值为头结点
    while(p!=NULL) {
        if(p->data!=x) {            // *p 结点值不为 x 时将其链接到 L 尾部
            r->next=p;
            r=p;
            p=p->next;              //继续扫描
        }
        else {                      // *p 结点值为 x 时将其释放
            q=p;
            p=p->next;              //继续扫描
            free(q);                //释放空间
        }
    }
    r->next=NULL;                  //插入结束后置尾结点指针为 NULL
}
```

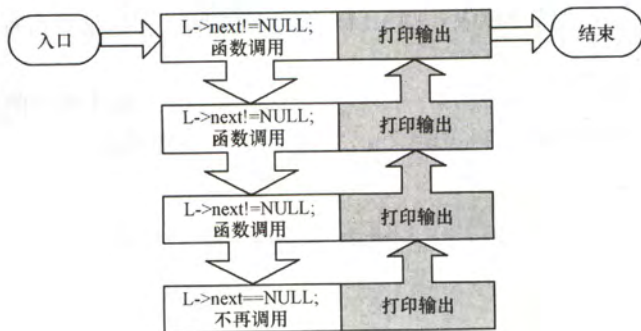
上述两个算法扫描一遍链表, 时间复杂度为 $O(n)$, 空间复杂度为 $O(1)$ 。

3. 解答:

考虑到从头到尾输出比较简单, 本题的思路很自然地联系到借助上题链表逆置的方法来实现, 改变链表的方向, 然后就可以从头到尾实现反向输出了。

此外, 本题还可以借助一个栈来实现, 每经过一个结点时, 将该结点放入栈中。在遍历完整个链表后, 再从栈顶开始输出结点值即可。这种实现的方法, 请读者在学习完第 3 章后再自行思考 (实现时可直接使用栈的基本操作函数)。

既然能用栈的思想解决, 也就很自然地联想到了用递归来实现, 每当访问一个结点时, 先递归输出它后面的结点, 再输出该结点自身, 这样链表就反向输出了, 如下图所示:



本题代码如下:

```
void R_Print(LinkList L) {
    //从尾到头输出单链表 L 中每个结点的值
    if (L->next!=NULL) {
```



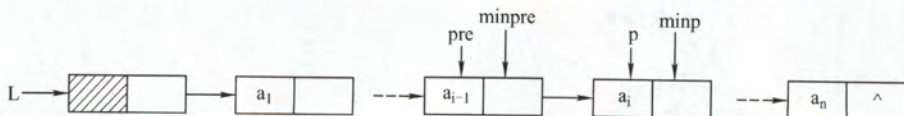
```

        R_Print(L->next);        //递归
    } //if
    print(L->data);              //输出函数
}

```

4. 解答:

算法思想: 用 p 从头至尾扫描单链表, pre 指向* p 结点的前驱, 用 $minp$ 保存值最小的结点指针 (初值为 p), $minpre$ 指向* $minp$ 结点的前驱 (初值为 pre)。一边扫描, 一边比较, 若 $p \rightarrow data$ 小于 $minp \rightarrow data$, 则将 p 、 pre 分别赋值给 $minp$ 、 $minpre$, 如下图所示。当 p 扫描完毕, $minp$ 指向最小值结点, $minpre$ 指向最小值结点的前驱结点, 再将 $minp$ 所指结点删除即可。



本题代码如下:

```

LinkedList Delete_Min(LinkedList &L){
    //L 是带头结点的单链表, 本算法删除其最小值结点
    LNode *pre=L, *p=pre->next; //p 为工作指针, pre 指向其前驱
    LNode *minpre=pre, *minp=p; //保存最小值结点及其前驱
    while(p!=NULL){
        if(p->data<minp->data){
            minp=p;                //找到比之前找到的最小值结点更小的结点
            minpre=pre;
        }
        pre=p;                    //继续扫描下一个结点
        p=p->next;
    }
    minpre->next=minp->next;      //删除最小值结点
    free(minp);
    return L;
}

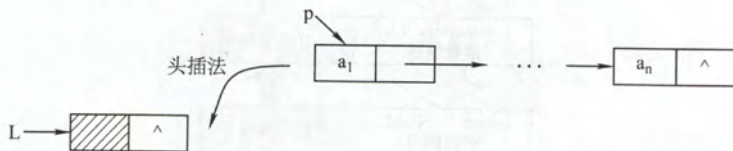
```

算法需要从头至尾扫描链表, 时间复杂度为 $O(n)$, 空间复杂度为 $O(1)$ 。

若本题改为不带头结点的单链表, 则实现上会有所不同, 留给读者自行思考。

5. 解答:

解法一: 将头结点摘下, 然后从第一结点开始, 依次前插入到头结点的后面 (头插法建立单链表), 直到最后一个结点为止, 则实现了链表的逆置, 如下图所示。



本题代码如下:

```

LinkedList Reverse_1(LinkedList L){
    //L 是带头结点的单链表, 本算法将 L 就地逆置
}

```

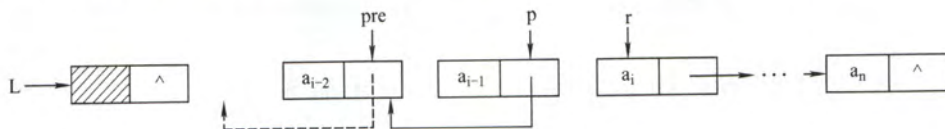
```

LNode *p, *r;           //p 为工作指针, r 为 p 的后继, 以防断链
p=L->next;              //从第一个元素结点开始
L->next=NULL;           //先将头结点 L 的 next 域置为 NULL
while(p!=NULL){         //依次将元素结点摘下
    r=p->next;           //暂存 p 的后继
    p->next=L->next;     //将 p 结点插入到头结点之后
    L->next=p;
    p=r;
}
return L;

```

解法二：大部分辅导书都只介绍解法一，这对读者的理解和思维是不利的。为了将调整指针这个复杂的过程分析清楚，我们借助图形来进行直观的分析。

假设 pre 、 p 和 r 指向 3 个相邻的结点，如下图所示。假设经过若干操作， $*pre$ 之前的结点的指针都已调整完毕，它们的 $next$ 都指向其原前驱结点。现在令 $*p$ 结点的 $next$ 域指向 $*pre$ 结点，注意到一旦调整指针的指向后， $*p$ 的后继结点的链就断开了，为此需要用 r 来指向原 $*p$ 的后继结点。处理时需要注意两点：一是在处理第一个结点时，应将其 $next$ 域置为 $NULL$ ，而不是指向头结点（因为它将作为新表的尾结点）；二是在处理完最后一个结点后，需要将头结点的指针指向它。



本题代码如下：

```

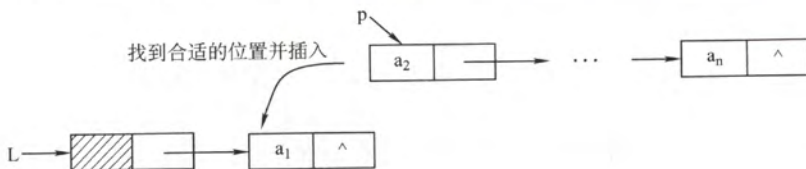
LinkList Reverse_2(LinkList L){
    //依次遍历线性表 L, 并将结点指针反转
    LNode *pre, *p=L->next, *r=p->next;
    p->next=NULL;           //处理第一个结点
    while(r!=NULL){         //r 为空, 则说明 p 为最后一个结点
        pre=p;              //依次继续遍历
        p=r;
        r=r->next;
        p->next=pre;        //指针反转
    }
    L->next=p;              //处理最后一个结点
    return L;
}

```

上述两个算法的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

6. 解答：

算法思想：采用直接插入排序算法的思想，先构成只含一个数据结点的有序单链表，然后依次扫描单链表中剩下的结点 $*p$ （直至 $p=NULL$ 为止），在有序表中通过比较查找插入 $*p$ 的前驱结点 $*pre$ ，然后将 $*p$ 插入到 $*pre$ 之后，如下图所示。



本题代码如下:

```
void Sort(LinkList &L){
    //本算法实现将单链表L的结点重排,使其递增有序
    LNode *p=L->next,*pre;
    LNode *r=p->next;           //r 保持*p 后继结点指针,以保证不断链
    p->next=NULL;                //构造只含一个数据结点的有序表
    p=r;
    while(p!=NULL){
        r=p->next;               //保存*p 的后继结点指针
        pre=L;
        while(pre->next!=NULL&&pre->next->data<p->data)
            pre=pre->next;       //在有序表中查找插入*p 的前驱结点*pre
        p->next=pre->next;        //将*p 插入到*pre 之后
        pre->next=p;
        p=r;                     //扫描原单链表中剩下的结点
    }
}
```

细心的读者显然发现该算法的时间复杂度为 $O(n^2)$, 为达到最佳的时间性能, 可先将链表的数据复制到数组里, 再采用时间复杂度为 $O(n\log_2 n)$ 的排序算法进行排序, 然后将数组元素依次插入到链表中, 此时的时间复杂度为 $O(n\log_2 n)$, 显然是以空间换时间的策略。

7. 解答:

因为链表是无序的, 所以只能逐个结点进行检查, 执行删除。

本题代码如下:

```
void RangeDelete(LinkList &L,int min,int max){
    LNode *pr=L,*p=L->link;      //p 是检测指针, pr 是其前驱
    while(p!=NULL)
        if(p->data>min&&p->data<max){ //寻找到被删结点, 删除
            pr->link=p->link;
            free(p);
            p=pr->link;
        }
        else{                      //否则继续寻找被删结点
            pr=p;
            p=p->link;
        }
    }
}
```

8. 解答:

两个单链表有公共结点, 也就是说两个链表从某一结点开始, 它们的 `next` 都指向同一个结点。由于每个单链表结点只有一个 `next` 域, 因此从第一个公共结点开始, 之后它们所有的结点都是重合的, 不可能再出现分叉。所以, 两个有公共结点而部分重合的单链表, 拓扑形状看起来像 Y, 而不可能像 X。

本题极易联想到“蛮”方法: 在第一个链表上顺序遍历每个结点, 每遍历一个结点, 在第二个链表上顺序遍历所有结点, 如果找到两个相同的结点, 于是就找到了它们的公共结点。显然, 该算法的时间复杂度为 $O(len1 * len2)$ 。

接下来我们试着去寻找一个线性时间复杂度的算法。先把问题简化: 如何判断两个单向链表有没有公共结点? 应注意到这样一个事实: 如果两个链表有一个公共结点, 那么该公共结点之后的所有结点都是重合的, 即它们的最后一个结点必然是重合的。因此, 我们判断两个链表是不是

有重合的部分,只要分别遍历两个链表到最后一个结点。如果两个尾结点是一样的,说明它们有公共结点,否则两个链表没有公共的结点。

然而,在上面的思路中,顺序遍历两个链表到尾结点的时候,并不能保证在两个链表上同时到达尾结点。这是因为两个链表长度不一定一样。但假设一个链表比另一个长 k 个结点,我们先在长的链表上遍历 k 个结点,之后再同步遍历,此时我们就能保证同时到达最后一个结点了。由于两个链表从第一个公共结点开始到链表的尾结点,这一部分是重合的。因此,它们肯定也是同时到达第一公共结点的。于是在遍历中,第一个相同的结点就是第一个公共的结点。

在这个思路中,我们先要分别遍历两个链表得到它们的长度,并求出两个长度之差。在长的链表上先遍历长度之差个结点之后,再同步遍历两个链表,直到找到相同的结点,或者一直到链表结束。此时,该方法的时间复杂度为 $O(\text{len1}+\text{len2})$ 。

本题代码如下:

```
LinkList Search_1st_Common(LinkList L1,LinkList L2){
//本算法实现在线性的时间内找到两个单链表的第一个公共结点
    int len1=Length(L1),len2=Length(L2);    //计算两个链表的表长
    LinkList longList,shortList;            //分别指向表长较长和较短的链表
    if(len1>len2){                            //L1 表长较长
        longList=L1->next; shortList=L2->next;

        dist=len1-len2;                        //表长之差
    }
    else{                                    //L2 表长较长
        longList=L2->next; shortList=L1->next;
        dist=len2-len1;                        //表长之差
    }
    while(dist-->0)                          //表长的链表先遍历到第 dist 个结
                                                //点,然后同步
        longList=longList->next;
    while(longList!=NULL){                    //同步寻找共同结点
        if(longList==shortList)              //找到第一个公共结点
            return longList;
        else{                                //继续同步寻找
            longList=longList->next;
            shortList=shortList->next;
        }
    }
    return NULL;
}
```

9. 解答:

算法思想:对链表进行遍历,在每趟遍历中查找出整个链表的最小值元素,输出并释放结点所占空间;再查找次小值元素,输出并释放空间,如此下去,直至链表为空,最后释放头结点所占存储空间,该算法的时间复杂度为 $O(n^2)$ 。

```
void Min_Delete(LinkList &head){
//head 是带头结点的单链表的头指针,本算法按递增顺序输出单链表中的数据元素
    while(head->next!=NULL){                //循环到仅剩头结点
        pre=head;                            //pre 为元素最小值结点的前驱结点的指针
        p=pre->next;                          //p 为工作指针
        while(p->next!=NULL){
            if(p->next->data<pre->next->data)
```



```

        pre=p;                                //记住当前最小值结点的前驱
        p=p->next;
    }
    print(pre->next->data);                    //输出元素最小值结点的数据
    u=pre->next;                                //删除元素值最小的结点, 释放结点空间
    pre->next=u->next;
    free(u);
} //while
free(head);                                //释放头结点
}

```

若题设不限制数组辅助空间的使用, 则可先将链表的数据复制在数组里, 再采用时间复杂度为 $O(n\log_2 n)$ 的排序算法进行排序, 然后将数组元素输出, 时间复杂度为 $O(n\log_2 n)$ 。

10. 解答:

算法思想: 设置一个访问序号变量 (初值为 0), 每访问一个结点序号自动加 1, 然后根据序号的奇偶性将结点插入到 A 表或 B 表中。重复以上操作直到表尾。

```

LinkedList DisCreat_1(LinkedList &A){
//将表 A 中结点按序号的奇偶性分解到表 A 或表 B 中
    i=0;                                        //i 记录表 A 中结点的序号
    B=(LinkedList)malloc(sizeof(LNode));      //创建 B 表表头
    B->next=NULL;                              //B 表的初始化
    LNode *ra=A, *rb=B;                       //ra 和 rb 将分别指向将创建的 A 表和 B
                                              //表的尾结点
    p=A->next;                                //p 为链表工作指针, 指向待分解的结点
    A->next=NULL;                              //置空新的 A 表
    while(p!=NULL){
        i++;                                  //序号加 1
        if(i%2==0){                          //处理序号为偶数的链表结点
            rb->next=p;                       //在 B 表尾插入新结点
            rb=p;                             //rb 指向新的尾结点
        }
        else{                                //处理原序号为奇数的结点
            ra->next=p;                       //在 A 表尾插入新结点
            ra=p;
        }
        p=p->next;                            //将 p 恢复为指向新的待处理结点
    }                                          //while 结束
    ra->next=NULL;
    rb->next=NULL;
    return B;
}

```

为了保持原来结点中的顺序, 本题采用尾插法建立单链表。此外, 本算法完全可以不用设置序号变量。while 循环中的代码改为将结点插入到表 A 中和将下一结点插入到表 B 中, 这样 while 中第一处理的结点就是奇数号结点, 第二处理的结点就是偶数号结点。

11. 解答:

算法思想: 采用上题的思路, 不设序号变量。二者的差别仅在于对 B 表的建立不采用尾插法, 而是采用头插法。

```

LinkedList DisCreat_2(LinkedList &A){
    LinkedList B=(LinkedList)malloc(sizeof(LNode)); //创建 B 表表头

```



```

B->next=NULL;           //B 表的初始化
LNode *p=A->next,*q;     //p 为工作指针
LNode *ra=A;             //ra 始终指向 A 的尾结点
while(p!=NULL){
    ra->next=p; ra=p;     //将*p 链到 A 的表尾
    p=p->next;
    q=p->next;            //头插后, *p 将断链, 因此用 q 记忆*p 的后继
    p->next=B->next;      //将*p 插入到 B 的前端
    B->next=p;
    p=q;
}
ra->next=NULL;           //A 尾结点的 next 域置空
return B;
}

```

该算法特别需要注意的是, 采用头插法插入结点后, *p 的指针域已改变, 如果不设变量保存其后继结点会引起断链, 从而导致算法出错。

12. 解答:

算法思想: 由于是有序表, 所有相同值域的结点都是相邻的。用 p 扫描递增单链表 L, 若 *p 结点的值域等于其后继结点的值域, 则删除后者, 否则 p 移向下一个结点。

```

void Del_Same(LinkList &L){
//L 是递增有序的单链表, 本算法删除表中数值相同的元素
    LNode *p=L->next,*q;      //p 为扫描工作指针
    if(p==NULL)
        return;
    while(p->next!=NULL){
        q=p->next;            //q 指向*p 的后继结点
        if(p->data==q->data){  //找到重复值的结点
            p->next=q->next;    //释放*q 结点
            free(q);           //释放相同元素值的结点
        }
        else
            p=p->next;
    }
}

```

本算法的时间复杂度为 $O(n)$, 空间复杂度为 $O(1)$ 。

本题也可采用尾插法, 将头结点摘下, 然后从第一结点开始, 依次与已经插入节点的链表的最后一个节点比较, 若不等则直接插入, 否则将当前遍历的节点删除并处理下一个节点, 直到最后一个节点为止。

13. 解答:

算法思想: 两个链表已经按元素值递增次序排序, 将其合并时, 均从第一个结点起进行比较, 将小的结点链入链表中, 同时后移工作指针。该问题要求结果链表按元素值递减次序排列, 故新链表的建立应该采用头插法。比较结束后, 可能会有一个链表非空, 此时用头插法将剩下的结点依次插入新链表中即可。

```

void MergeList(LinkList &La, LinkList &Lb){
//合并两个递增有序链表(带头结点), 并使合并后的链表递减排列
    LNode *r,*pa=La->next,*pb=Lb->next; //分别是表 La 和 Lb 的工作指针
    La->next=NULL;                       //La 作为结果链表的头指针, 先将结果

```



```

while(pa&&pb)
    if(pa->data<=pb->data){
        r=pa->next;
        pa->next=La->next;
        La->next=pa;

        pa=r;
    }
    else{
        r=pb->next;
        pb->next=La->next;
        La->next=pb;

        pb=r;
    }
if(pa)
    pb=pa;

while(pb){
    r=pb->next;
    pb->next=La->next;
    La->next=pb;
    pb=r;
}
free(Lb);
}

```

链表初始化为空
//当两链表均不为空时，循环
//r 暂存 pa 的后继结点指针
//将 pa 结点链于结果表中，同时逆置（头插法）
//恢复 pa 为当前待比较结点
//r 暂存 pb 的后继结点指针
//将 pb 结点链于结果表中，同时逆置（头插法）
//恢复 pb 为当前待比较结点
//通常情况下会剩一个链表非空，处理剩下的部分
//处理剩下的一个非空链表
//依次插入到 La 中（头插法）

14. 解答:

算法思想: 表 A、B 都有序, 可从第一个元素起依次比较 A、B 两表的元素, 若元素值不等, 则值小的指针往后移, 若元素值相等, 则创建一个值等于两结点的元素值的新的结点, 使用尾插法插入到新的链表中, 并两个原表指针后移一位, 直到其中一个链表遍历到表尾。

```

void Get_Common(LinkList A, LinkList B){
    //本算法产生单链表 A 和 B 的公共元素的单链表 C
    LNode *p=A->next, *q=B->next, *r, *s;
    LinkList C=(LinkList)malloc(sizeof(LNode)); //建立表 C
    r=C;
    while(p!=NULL&&q!=NULL){
        if(p->data<q->data)
            p=p->next;
        else if(p->data>q->data)
            q=q->next;
        else{
            s=(LNode*)malloc(sizeof(LNode));
            s->data=p->data;
            r->next=s;
            r=s;
            p=p->next;
            q=q->next;
        }
    }
}

```

//r 始终指向 C 的尾结点
//循环跳出条件
//若 A 的当前元素较小, 后移指针
//若 B 的当前元素较小, 后移指针
//找到公共元素结点
//复制产生结点*s
//将*s 链接到 C 上(尾插法)
//表 A 和 B 继续向后扫描


```
    }  
    r->next=NULL;           //置C尾结点指针为空  
}
```

15. 解答:

算法思想: 采用归并的思想, 设置两个工作指针 **pa** 和 **pb**, 对两个链表进行归并扫描, 只有同时出现在两集中的元素才链接到结果表中且仅保留一个, 其他的结点全部释放。当一个链表遍历完毕后, 释放另一个表中剩下的全部结点。

```
LinkList Union(LinkList &la, LinkList &lb){  
    pa=la->next;           //设工作指针分别为 pa 和 pb  
    pb=lb->next;  
    pc=la;                 //结果表中当前合并结点的前驱指针  
    while(pa&&pb){  
        if(pa->data==pb->data){ //交集并入结果表中  
            pc->next=pa;       //A 中结点链接到结果表  
            pc=pa;  
            pa=pa->next;  
            u=pb;              //B 中结点释放  
            pb=pb->next;  
            free(u);  
        }  
        else if(pa->data<pb->data){ //若 A 中当前结点值小于 B 中当前结点值  
            u=pa;  
            pa=pa->next;       //后移指针  
            free(u);          //释放 A 中当前结点  
        }  
        else{               //若 B 中当前结点值小于 A 中当前结点值  
            u=pb;  
            pb=pb->next;       //后移指针  
            free(u);          //释放 B 中当前结点  
        }  
    } //while 结束  
    while(pa){              //B 已遍历完, A 未完  
        u=pa;  
        pa=pa->next;  
        free(u);            //释放 A 中剩余结点  
    }  
    while(pb){              //A 已遍历完, B 未完  
        u=pb;  
        pb=pb->next;  
        free(u);            //释放 B 中剩余结点  
    }  
    pc->next=NULL;          //置结果链表尾指针为 NULL  
    free(lb);               //释放 B 表的头结点  
    return la;  
}
```


链表归并类型的题在各学校历年真题中出现的频率很高，故应扎实掌握解决此类问题的思想。该算法的时间复杂度为 $O(\text{len1}+\text{len2})$ ，空间复杂度为 $O(1)$ 。

16. 解答:

算法思想：因为两个整数序列已存入两个链表中，操作从两个链表的第一个结点开始，若对应数据相等，则后移指针；若对应数据不等，则 A 链表从上次开始比较结点的后继开始，B 链表仍从第一个结点开始比较，直到 B 链表到尾表示匹配成功。A 链表到尾而 B 链表未到尾表示失败。操作中应记住 A 链表每次的开始结点，以便下趟匹配时好从其后继开始。

```
int Pattern(LinkList A, LinkList B) {
    //A 和 B 分别是数据域为整数的单链表，本算法判断 B 是否是 A 的子序列
    LNode *p=A;                //p 为 A 链表的工作指针，本题假定 A 和 B 均无头结点
    LNode *pre=p;               //pre 记住每趟比较中 A 链表的开始结点
    LNode *q=B;                 //q 是 B 链表的工作指针
    while(p && q)
        if(p->data==q->data) { //结点值相同
            p=p->next;
            q=q->next;
        }
        else {
            pre=pre->next;
            p=pre;                //A 链表新的开始比较结点
            q=B;                 //q 从 B 链表第一个结点开始
        }
    if(q==NULL)                 //B 已经比较结束
        return 1;               //说明 B 是 A 的子序列
    else
        return 0;               //B 不是 A 的子序列
}
```

注：该题其实是字符串模式匹配的链式表示形式，读者应该结合第 6 章的“字符串模式匹配”小节重新考虑能否优化该算法。

17. 解答:

算法思想：让 p 从左向右扫描，q 从右向左扫描，直到它们指向同一结点 ($p==q$ ，当循环双链表中结点数为奇数时) 或相邻 ($p->\text{next}=q$ 或 $q->\text{prior}=p$ ，当循环双链表中结点数为偶数时) 为止，若它们所指结点值相同，则继续进行下去，否则返回 0。若比较全部相等，则返回 1。

```
int Symmetry(DLinkList L) {
    //本算法从两头扫描循环双链表，以判断链表是否对称
    DNode *p=L->next, *q=L->prior; //两头工作指针
    while(p!=q && q->next!=p)      //循环跳出条件
        if(p->data==q->data) {     //所指结点值相同则继续比较
            p=p->next;
            q=q->prior;
        }
        else                        //否则，返回 0
            return 0;
    return 1;                       //比较结束后返回 1
}
```

18. 解答:

算法思想：先找到两个链表的尾指针，将第一个链表的尾指针与第二个链表的头结点链接起

来,再使之成为循环的。

```

LinkList Link(LinkList &h1, LinkList &h2) {
//将循环链表 h2 链接到循环链表 h1 之后,使之仍保持循环链表的形式
    LNode *p, *q;                //分别指向两个链表的尾结点
    p=h1;
    while(p->next!=h1)            //寻找 h1 的尾结点
        p=p->next;
    q=h2;
    while(q->next!=h2)            //寻找 h2 的尾结点
        q=q->next;
    p->next=h2;                  //将 h2 链接到 h1 之后
    q->next=h1;                  //令 h2 的尾结点指向 h1
    return h1;
}

```

19. 解答:

对于循环单链表 L,在不空时循环:每循环一次查找一个最小结点(由 minp 指向最小值结点, minpre 指向其前驱结点)并删除它。最后释放头结点。

```

void Del_All(LinkList &L) {
//本算法实现每次删除循环单链表中的最小元素,直到链表空为止
    LNode *p, *pre, *minp, *minpre;
    while(L->next!=L) {          //表不空,循环
        p=L->next; pre=L;        //p 为工作指针, pre 指向其前驱
        minp=p; minpre=pre;      //minp 指向最小值结点
        while(p!=L) {           //循环一趟,查找最小值结点
            if(p->data<minp->data) {
                minp=p;          //找到值更小的结点
                minpre=pre;
            }
            pre=p;                //查找下一个结点
            p=p->next;
        }
        printf("%d", minp->data) //输出最小值结点元素
        minpre->next=minp->next; //最小值结点从表中“断”开
        free(minp);              //释放空间
    }
    free(L);                     //释放头结点
}

```

20. 解答:

此题主要考查双链表的查找、删除和插入算法。

算法思想:首先在双向链表中查找数据值为 x 的结点,查到后,将结点从链表上摘下,然后再顺着结点的前驱链查找该结点的插入位置。(频度递减,且排在同频度的第一个,即向前找到第一个比它频度大的结点,插入位置为该结点之后),并插入到该位置。

```

DLinkList Locate(DLinkList &L, ElemType x) {
//本算法先查找数据 x, 查找成功时结点的访问频度域增 1
//最后将该结点按频度递减插入链表中适当位置(同频度最近访问的在前面)
    DNode *p=L->next, *q;        //p 为工作指针, q 为 p 的前驱, 用于查找插入位置
    while(p&& p->data!=x)

```



```

        p=p->next;                //查找值为x的结点
    if(!p){
        printf("不存在值为x的结点\n");
        exit(0);
    }
    else{
        p->freq++;                //令元素值为x的结点的freq域加1
        p->next->pred=p->pred;
        p->pred->next=p->next;    //将p结点从链表上摘下
        q=p->pred;                //以下查找p结点的插入位置
        while(q!=L&&q->freq<=p->freq)
            q=q->pred;
        p->next=q->next;
        q->next->pred=p;          //将p结点插入，一定是排在同频率的第一个
        p->pred=q;
        q->next=p;
    }
    return p;                    //返回值为x的结点的指针
}

```

21. 解答:

1) 算法的基本设计思想如下:

问题的关键是设计一个尽可能高效的算法,通过链表的一趟遍历,找到倒数第 k 个结点的位置。算法的基本设计思想是:定义两个指针变量 p 和 q ,初始时均指向头结点的下一个结点(链表的第一个结点), p 指针沿链表移动;当 p 指针移动到第 k 个结点时, q 指针开始与 p 指针同步移动;当 p 指针移动到最后一个结点时, q 指针所指示结点为倒数第 k 个结点。以上过程对链表仅进行一遍扫描。

2) 算法的详细实现步骤如下:

① $\text{count}=0$, p 和 q 指向链表表头结点的下一个结点。

② 若 p 为空,转⑤。

③ 若 count 等于 k ,则 q 指向下一个结点;否则, $\text{count}=\text{count}+1$ 。

④ p 指向下一个结点,转②。

⑤ 若 count 等于 k ,则查找成功,输出该结点的 data 域的值,返回1;否则,说明 k 值超过了线性表的长度,查找失败,返回0。

⑥ 算法结束。

3) 算法实现

```

typedef int ElemType;           //链表数据的类型定义
typedef struct LNode{           //链表结点的结构定义
    ElemType data;              //结点数据
    struct LNode *link;         //结点链接指针
}LNode, *LinkList;
int Search_k(LinkList list,int k){
    //查找链表list倒数第k个结点,并输出该结点data域的值
    LNode *p=list->link,*q=list->link; //指针p、q指示第一个结点
    int count=0;
    while(p!=NULL){              //遍历链表直到最后一个结点
        if (count<k) count++;     //计数,若count<k只移动p
        else q=q->link;
    }
}

```



```

        p=p->link;                //之后让 p、q 同步移动
    }//while
    if(count<k)
        return 0;                //查找失败返回 0
    else {
        printf("%d",q->data);
        return 1;                //否则打印并返回 1
    }
} //Search_k

```

【评分说明】：若所给出的算法采用一遍扫描方式就能得到正确结果，可给满分 15 分；若采用两遍或多遍扫描才能得到正确结果的，最高分为 10 分。若采用递归算法得到正确结果的，最高给 10 分；若实现算法的空间复杂度过高（使用了大小与 k 有关的辅助数组），但结果正确，最高给 10 分。

22. 解答：

本题的结构体是单链表，采用双指针法。用指针 p 、 q 分别扫描 $str1$ 和 $str2$ ，当 p 、 q 指向同一个地址时，即找到共同后缀的起始位置。（注：该题与第 8 题的思路一样）

1) 算法的基本设计思想如下：

① 分别求出 $str1$ 和 $str2$ 所指的链表的长度 m 和 n 。

② 将两个链表以表尾对齐：令指针 p 、 q 分别指向 $str1$ 和 $str2$ 的头结点，若 $m \geq n$ ，则指针 p 先走，使 p 指向链表中的第 $m-n+1$ 个结点；若 $m < n$ ，则使 q 指向链表中的第 $n-m+1$ 个结点，即使指针 p 和 q 所指的结点到表尾的长度相等。

③ 反复将指针 p 和 q 同步向后移动，当 p 、 q 指向同一位置时停止，即为共同后缀的起始位置，算法结束。

2) 本题代码如下：

```

typedef struct Node{
    char data;
    struct Node *next;
}SNode;
/*求链表长度的函数*/
int listlen(SNode *head){
    int len=0;
    while(head->next!=NULL){
        len++;
        head=head->next;
    }
    return len;
}
/*找出共同后缀的起始地址*/
SNode* find_addr(SNode *str1,SNode *str2){
    int m,n;
    SNode *p,*q;
    m=listlen(str1);        //求 str1 的长度
    n=listlen(str2);        //求 str2 的长度
    for(p=str1;m>n;m--)      //若 m>n, 使 p 指向链表中的第 m-n+1 个结点
        p=p->next;
    for(q=str2;m<n;n--)      //若 m<n, 使 q 指向链表中的第 n-m+1 个结点
        q=q->next;
}

```



```

while(p->next!=NULL&& p->next!=q->next){    //将指针 p 和 q 同步向后移动
    p=p->next;
    q=q->next;
} //while
return p->next;    //返回共同后缀的起始地址
}

```

23. 解答:

1) 算法的基本设计思想

- 算法的核心思想是用空间换时间。使用辅助数组记录链表中已出现的数值，从而只需对链表进行一趟扫描。
- 因为 $|data| \leq n$ ，故辅助数组 q 的大小为 $n+1$ ，各元素的初值均为 0。依次扫描链表中的各结点，同时检查 $q[|data|]$ 的值，如果为 0，则保留该结点，并令 $q[|data|]=1$ ；否则，将该结点从链表中删除。

2) 使用 C 语言描述的单链表结点的数据类型定义

```

typedef struct node {
    int      data;
    struct node  *link;
}NODE;
typedef NODE *PNODE;

```

3) 算法实现

```

void func (PNODE h,int n)
{
    PNODE p=h,r;
    int *q,m;
    q=(int *)malloc(sizeof(int)*(n+1)); //申请 n+1 个位置的辅助空间
    for(int i=0;i<n+1;i++)              //数组元素初值置 0
        *(q+i)=0;
    while(p->link!=NULL)
    {
        m=p->link->data>0? p->link->data:-p->link->data;
        if(*(q+m)==0)                    //判断该结点的 data 是否已出现过
        {
            *(q+m)=1;                  //首次出现
            p=p->link;                  //保留
        }
        else                             //重复出现
        {
            r=p->link;                  //删除
            p->link=r->link;
            free(r);
        }
    }
    free(q);
}

```

4) 参考答案所给算法的时间复杂度为 $O(m)$ ，空间复杂度为 $O(n)$ 。

归纳总结

本章是算法设计题的重点考查章节，因为线性表的算法题代码量一般都比较少，又具有一定的算法设计技巧，适合笔试考查。考研题中经常以三段式的结构命题：

在给出题目背景和要求的情况下：

- ① 给出算法的基本设计思想；
- ② 采用 C 或 C++ 语言描述算法，并给出注释；
- ③ 分析所设计算法的时间复杂度和空间复杂度。

这些题目形式类似，但算法具体的设计思想千变万化，难以从一而定。因此读者一定要勤加练习，反复咀嚼本章的练习题，采用多种方法进行设计并比较它们的复杂度，逐渐熟悉各类题型的思考角度和最佳思路。在这里，编者列出几种常用的算法设计技巧，仅供参考：对于链表，经常采用的方法有头插法、尾插法、逆置法、归并法、双指针法等，对具体问题需要灵活变通；对于顺序表，由于可以直接存取，经常结合排序和查找的几种算法设计思路进行设计，如归并排序、二分查找等。

提示：算法设计题，如果能写出数据结构类型定义、正确的算法思想都会至少给一半的分数，如果能用伪代码写出自然更好，比较复杂的地方可以直接用文字表达。

思维拓展

一个长度为 N 的整型数组 $A[1 \cdots N]$ ，给定整数 X ，请设计一个时间复杂度不超过 $O(n \log_2 n)$ 的算法，查找出这个数组中所有两两之和等于 X 的整数对（每个元素只输出一次）。

（提示：本题若想到排序，问题便迎刃而解。先用一种时间复杂度为 $O(n \log_2 n)$ 的排序算法将 $A[1 \cdots N]$ 从小到大排序，可以用快速排序（或二路归并等），然后分别从数组的小端（ $i=1$ ）和大端（ $j=N$ ）开始查找：若 $A[i]+A[j]<X$ ， $i++$ ；若 $A[i]+A[j]>X$ ， $j--$ ；否则输出 $A[i]$ 、 $A[j]$ ，然后 $i++$ ， $j--$ ；直到 $i>j$ 停止）

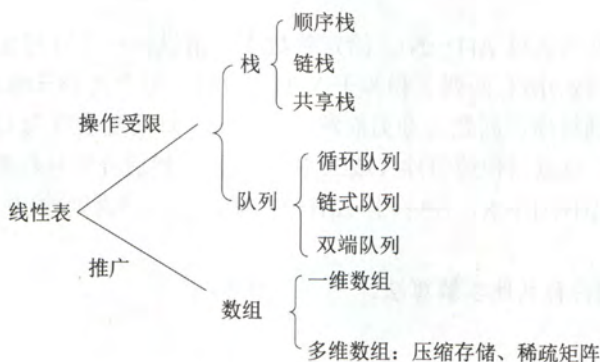
请读者思考本题是否有其他求解算法。

第3章 栈和队列

【考纲内容】

- (一) 栈和队列的基本概念
- (二) 栈和队列的顺序存储结构
- (三) 栈和队列的链式存储结构
- (四) 栈和队列的应用
- (五) 特殊矩阵的压缩存储

【知识框架】



【复习提示】

本章内容通常以选择题的形式考查，题目不算难，但命题的形式比较灵活。其中栈（出入栈的过程、出栈序列的合法性）和队列的操作及其特征是重中之重，属每年必考内容。由于它们均是线性表的应用和推广，很容易出现在算法设计题中。此外，栈和队列的顺序存储结构、链式存储结构及其特点、双端队列的特点、栈和队列的常见应用，以及数组和特殊矩阵的压缩存储都是读者必须掌握的内容。

3.1 栈

3.1.1 栈的基本概念

1. 栈的定义

栈 (Stack)：只允许在一端进行插入或删除操作的线性表。首先栈是一种线性表，但是限定这种线性表只能在某一端进行插入和删除操作，如图 3-1 所示。

栈顶 (Top)：线性表允许进行插入和删除的那一端。

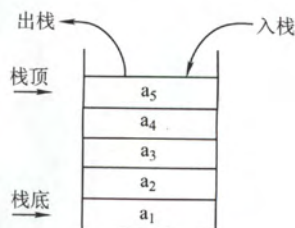


图 3-1 栈的示意图

栈底 (Bottom): 固定的, 不允许进行插入和删除的另一端。

空栈: 不含任何元素的空表。

假设某个栈 $S = (a_1, a_2, a_3, a_4, a_5)$, 如图 3-1 所示, 则 a_1 为栈底元素, a_5 为栈顶元素。由于栈只能在栈顶进行插入和删除操作, 故进栈次序依次为 a_1, a_2, a_3, a_4, a_5 , 而出栈次序为 a_5, a_4, a_3, a_2, a_1 。由此可见, 栈的一个明显的操作特性可以概括为后进先出 (Last In First Out, LIFO), 故又称为后进先出的线性表。

注意: 我们每接触到一种新的数据结构类型, 都应该分别从其逻辑结构、存储结构和对数据的运算三个方面着手, 以加深对定义的理解。

2. 栈的基本操作

各种辅导书中给出的基本操作的名称不尽相同, 但所表达的意思大致是一样的。这里我们以严蔚敏编写的教材为准给出栈的基本操作, 希望读者能熟记下面的基本操作:

InitStack(&S): 初始化一个空栈 S。

StackEmpty(S): 判断一个栈是否为空, 若栈 S 为空返回 true, 否则返回 false。

Push(&S, x): 进栈, 若栈 S 未满, 将 x 加入使之成为新栈顶。

Pop(&S, &x): 出栈, 若栈 S 非空, 弹出栈顶元素, 并用 x 返回。

GetTop(S, &x): 读栈顶元素, 若栈 S 非空, 用 x 返回栈顶元素。

ClearStack(&S): 销毁栈, 并释放栈 S 占用的存储空间。(注: 符号 '&' 是 C++ 特有的, 用来表示引用调用, 有的书上采用 C 语言中的指针类型 '*', 也可以达到传址的目的。)

在解答算法题时, 若题干没有做出限制, 可以直接使用这些基本的操作函数。

3.1.2 栈的顺序存储结构

1. 顺序栈的实现

栈的顺序存储称为顺序栈, 它是利用一组地址连续的存储单元存放自栈底到栈顶的数据元素, 同时附设一个指针 (top) 指示当前栈顶的位置。

栈的顺序存储类型可描述为

```
#define MaxSize 50           //定义栈中元素的最大个数
typedef struct{
    Elemtype data[MaxSize];   //存放栈中元素
    int top;                  //栈顶指针
} SqStack;
```

栈顶指针: S.top, 初始时设置 S.top=-1; 栈顶元素: S.data[S.top]。

进栈操作: 栈不满时, 栈顶指针先加 1, 再送值到栈顶元素。

出栈操作: 栈非空时, 先取栈顶元素值, 再将栈顶指针减 1。

栈空条件: S.top== -1; 栈满条件: S.top==MaxSize-1; 栈长: S.top+1。

由于顺序栈的入栈操作受数组上界的约束, 当对栈的最大使用空间估计不足时, 有可能发生栈上溢, 此时应及时向用户报告消息, 以便及时处理, 避免出错。

对于栈和后面提到的队列的判空和判满条件会因为实际给的条件不同而变化, 以上提到的方法以及下面给出的代码实现只是在栈顶指针设定的条件下相应的方法, 而其他情况需要具体问题具体分析。

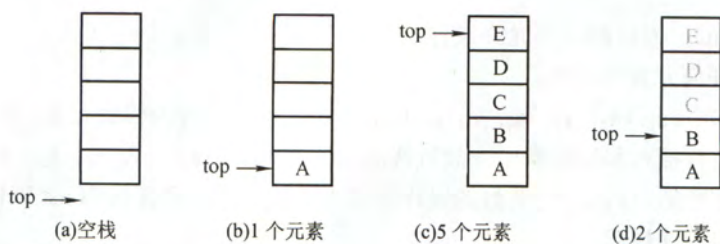


图 3-2 栈顶指针和栈中元素之间的关系

2. 顺序栈的基本运算

栈操作的示意图如图 3-2 所示, 图 3-2 (a) 是空栈, 图 3-2 (c) 是 A、B、C、D、E 共 5 个元素依次入栈后的结果, 图 3-2 (d) 是在图 3-2 (c) 之后 E、D、C 相继出栈, 此时栈中还有 2 个元素, 或许最近出栈的元素 C、D、E 仍在原先的单元存储着, 但 top 指针已经指向了新的栈顶, 则元素 C、D、E 已不在栈中了, 读者应通过该示意图深刻理解栈顶指针的作用。

下面是顺序栈上常用的基本运算的实现。

(1) 初始化

```
void InitStack(&S){
    s.top=-1;                //初始化栈顶指针
}
```

(2) 判栈空

```
bool StackEmpty(S){
    if(s.top==-1)            //栈空
        return true;
    else                    //不空
        return false;
}
```

(3) 进栈

```
bool Push(SqStack &S,ElemType x){
    if(S.top==MaxSize-1)    //栈满, 报错
        return false;
    S.data[++S.top]=x;      //指针先加 1, 再入栈
    return true;
}
```

(4) 出栈

```
bool Pop(SqStack &S,ElemType &x){
    if(S.top==-1)          //栈空, 报错
        return false;
    x=S.data[S.top--];     //先出栈, 指针再减 1
    return true;
}
```

(5) 读栈顶元素

```
bool GetTop(SqStack S,ElemType &x){
    if(S.top==-1)          //栈空, 报错
        return false;
    x=S.data[S.top];       //x 记录栈顶元素
    return true;
}
```

注意：这里栈顶指针指向的就是栈顶元素，所以进栈时的操作是 $S.data[++S.top]=x$ ；出栈时的操作是 $x=S.data[S.top--]$ 。如果栈顶指针初始化为 $S.top=0$ ，即栈顶指针指向栈顶元素的下一个位置，则入栈操作变为 $S.data[S.top++]=x$ ；出栈操作变为 $x=S.data[--S.top]$ 。相应的栈空、栈满条件也会发生变化。请读者仔细体会其中的不同之处，做题时也应灵活应变。

3. 共享栈

利用栈底位置相对不变的特性，可以让两个顺序栈共享一个一维数据空间，将两个栈的栈底分别设置在共享空间的两端，两个栈顶向共享空间的中间延伸，如图 3-3 所示。

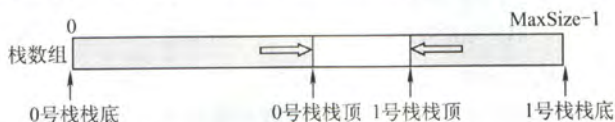


图 3-3 两个顺序栈共享存储空间

两个栈的栈顶指针都指向栈顶元素， $top0=-1$ 时 0 号栈为空， $top1=MaxSize$ 时 1 号栈为空；仅当两个栈顶指针相邻（ $top1-top0=1$ ）时，判断为栈满。当 0 号栈进栈时 $top0$ 先加 1 再赋值，1 号栈进栈时 $top1$ 先减 1 再赋值；出栈时则刚好相反。

共享栈是为了更有效地利用存储空间，两个栈的空间相互调节，只有在整个存储空间被占满时才发生上溢。其存取数据的时间复杂度均为 $O(1)$ ，所以对存取效率没有什么影响。

3.1.3 栈的链式存储结构

采用链式存储的栈称为链栈，链栈的优点是便于多个栈共享存储空间和提高其效率，且不存在栈满上溢的情况。通常采用单链表实现，并规定所有操作都是在单链表的表头进行的。这里规定链栈没有头结点， $Lhead$ 指向栈顶元素，如图 3-4 所示。



图 3-4 栈的链式存储

栈的链式存储类型可描述为

```
typedef struct Linknode{
    ElemType data;           //数据域
    struct Linknode *next;   //指针域
} *LiStack;                 //栈类型定义
```

采用链式存储，便于结点的插入与删除。链栈的操作与链表类似，在此不做详细讨论。读者需要注意的是，对于带头结点和不带头结点的链栈，在具体的实现方面有所不同。

3.1.4 本节试题精选

一、单项选择题

1. 栈和队列具有相同的（ ）。
A. 抽象数据类型 B. 逻辑结构 C. 存储结构 D. 运算
2. 栈是（ ）。
A. 顺序存储的线性结构 B. 链式存储的非线性结构
C. 限制存取点的线性结构 D. 限制存储点的非线性结构

3. () 不是栈的基本操作。

- A. 删除栈顶元素 B. 删除栈底元素
C. 判断栈是否为空 D. 将栈置为空栈

4. 假定利用数组 $a[n]$ 顺序存储一个栈, 用 top 表示栈顶指针, $top=-1$ 表示栈空, 并已知栈未满, 当元素 x 进栈时所执行的操作为 ()。

- A. $a[-top]=x$ B. $a[top-]=x$ C. $a[++top]=x$ D. $a[top++]=x$

5. 设有一个空栈, 栈顶指针为 $1000H$, 每个元素需要 1 个存储单元, 在执行 Push、Push、Pop、Push、Pop、Push、Pop、Push 操作后, 栈顶指针的值为 ()。

- A. $1002H$ B. $1003H$ C. $1004H$ D. $1005H$

6. 和顺序栈相比, 链栈有一个比较明显的优势是 ()。

- A. 通常不会出现栈满的情况 B. 通常不会出现栈空的情况
C. 插入操作更容易实现 D. 删除操作更容易实现

7. 设链表不带头结点且所有操作均在表头进行, 则下列最不适合作为链栈的是 ()。

- A. 只有表头结点指针, 没有表尾指针的双向循环链表
B. 只有表尾结点指针, 没有表头指针的双向循环链表
C. 只有表头结点指针, 没有表尾指针的单向循环链表
D. 只有表尾结点指针, 没有表头指针的单向循环链表

8. 向一个栈顶指针为 top 的链栈中插入一个 x 结点, 则执行 ()。

- A. $top \rightarrow next = x$ B. $x \rightarrow next = top \rightarrow next; top \rightarrow next = x$
C. $x \rightarrow next = top; top = x$ D. $x \rightarrow next = top, top = top \rightarrow next$

9. 链栈执行 Pop 操作, 并将出栈的元素存在 x 中应该执行 ()。

- A. $x = top; top = top \rightarrow next$ B. $x = top \rightarrow data$
C. $top = top \rightarrow next; x = top \rightarrow data$ D. $x = top \rightarrow data; top = top \rightarrow next$

10. 经过以下栈的操作后, 变量 x 的值为 ()。

InitStack(st); Push(st, a); Push(st, b); Pop(st, x); Top(st, x);

- A. a B. b C. NULL D. FALSE

11. 3 个不同元素依次进栈, 能得到 () 种不同的出栈序列。

- A. 4 B. 5 C. 6 D. 7

12. 设 a 、 b 、 c 、 d 、 e 、 f 以所给的次序进栈, 若在进栈操作时, 允许出栈操作, 则下面得不到的序列为 ()。

- A. $fedcba$ B. $bcafed$ C. $dcefb a$ D. $cabdef$

13. 用 S 表示进栈操作, 用 X 表示出栈操作, 若元素的进栈顺序是 1234 , 为了得到 1342 的出栈顺序, 相应的 S 和 X 的操作序列为 ()。

- A. $SXSXSXX$ B. $SSXSXSXX$ C. $SXSXSXSX$ D. $SXSXSXSX$

14. 【2010 年计算机联考真题】

若元素 a 、 b 、 c 、 d 、 e 、 f 依次进栈, 允许进栈、退栈操作交替进行, 但不允许连续 3 次进行退栈操作, 则不可能得到的出栈序列是 ()。

- A. $dcebfa$ B. $cbdaef$ C. $bcaefd$ D. $afedcb$

15. 设栈 S 和队列 Q 的初始状态为空, 元素 e_1 、 e_2 、 e_3 、 e_4 、 e_5 和 e_6 依次通过栈 S , 一个元素出栈后即进队列 Q , 若 6 个元素出栈的序列是 e_2 、 e_4 、 e_3 、 e_6 、 e_5 、 e_1 , 则栈 S 的容量至少应该是 ()。

A. 6

B. 4

C. 3

D. 2

16. 【2009年计算机联考真题】

设栈S和队列Q的初始状态均为空,元素abcdefg依次进入栈S。若每个元素出栈后立即进入队列Q,且7个元素出队的顺序是bdcfeag,则栈S的容量至少是()。

A. 1

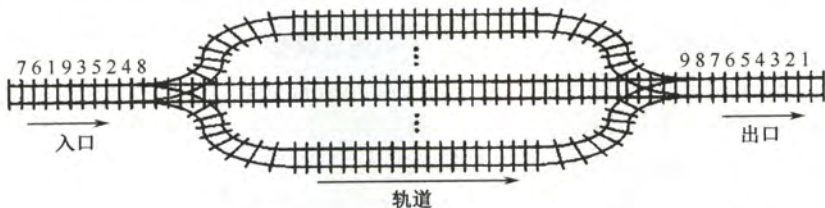
B. 2

C. 3

D. 4

17. 【2016年计算机联考真题】

设有如下图所示的火车车轨,入口到出口之间有 n 条轨道,列车的行进方向均为从左至右,列车可驶入任意一条轨道。现有编号为1~9的9列列车,驶入的次序依次是8,4,2,5,3,9,1,6,7。若期望驶出的次序依次为1~9,则 n 至少是()。



A. 2

B. 3

C. 4

D. 5

18. 若一个栈的输入序列是1, 2, 3, ..., n , 输出序列的第一个元素是 n , 则第 i 个输出元素是()。

A. 不确定

B. $n-i$ C. $n-i-1$ D. $n-i+1$

19. 一个栈的输入序列为1, 2, 3, ..., n , 输出序列的第一个元素是 i , 则第 j 个输出元素是()。

A. $i-j-1$ B. $i-j$ C. $j-i+1$

D. 不确定

20. 某栈的输入序列为a, b, c, d, 下面的4个序列中, 不可能是它的输出序列的是()。

A. a,b,c,d

B. c,b,d,a

C. d,c,a,b

D. a,c,b,d

21. 若一个栈的输入序列是 P_1, P_2, \dots, P_n , 其输出序列是1, 2, 3, ..., n , 若 $P_3=1$, 则 P_1 的值()。

A. 可能是2

B. 一定是2

C. 不可能是2

D. 不可能是3

22. 若已知一个栈的入栈序列是1, 2, 3, 4。其出栈序列为 P_1, P_2, P_3, P_4 , 则 P_2, P_4 不可能是()。

A. 2, 4

B. 2, 1

C. 4, 3

D. 3, 4

23. 【2013年计算机联考真题】

一个栈的入栈序列为1, 2, 3, ..., n , 其出栈序列是 $p_1, p_2, p_3, \dots, p_n$ 。若 $p_2=3$, 则 p_3 可能取值的个数是()。

A. $n-3$ B. $n-2$ C. $n-1$

D. 无法确定

24. 设栈的初始状态为空, 当字符序列“n1_”作为栈的输入时, 输出长度为3, 且可用做C语言标识符的序列有()个。

A. 4

B. 5

C. 3

D. 6

25. 【2011年计算机联考真题】

元素a, b, c, d, e依次进入初始为空的栈中, 若元素进栈后可停留、可出栈, 直到所有元素都出栈, 则在所有可能的出栈序列中, 以元素d开头的序列个数是()。

A. 3

B. 4

C. 5

D. 6

26. 采用共享栈的好处是 ()。
- A. 减少存取时间, 降低发生上溢的可能
B. 节省存储空间, 降低发生上溢的可能
C. 减少存取时间, 降低发生下溢的可能
D. 节省存储空间, 降低发生下溢的可能

27. 【2017年计算机联考真题】

下列关于栈的叙述中, 错误的是 ()。

- I. 采用非递归方式重写递归程序时必须使用栈
II. 函数调用时, 系统要用栈保存必要的信息
III. 只要确定了入栈次序, 即可确定出栈次序
IV. 栈是一种受限的线性表, 允许在其两端进行操作

- A. 仅 I B. 仅 I、II、III C. 仅 I、III、IV D. 仅 II、III、IV

28. 设有一个顺序共享栈 $\text{Share}[0: n-1]$, 其中第一个栈顶指针 top1 的初值为 -1, 第二个栈顶指针 top2 的初值为 n , 则判断共享栈满的条件是 ()。

- A. $\text{top2} - \text{top1} = 1$ B. $\text{top1} - \text{top2} = 1$ C. $\text{top1} = \text{top2}$ D. 以上都不对

二、综合应用题

1. 有 5 个元素, 其入栈次序为 A、B、C、D、E, 在各种可能的出栈次序中, 第一个出栈元素为 C 且第二个出栈元素为 D 的出栈序列有哪些?

2. 若元素的进栈序列为 A、B、C、D、E, 运用栈操作, 能否得到出栈序列 B、C、A、E、D 和 D、B、A、C、E? 为什么?

3. 假设以 I 和 O 分别表示入栈和出栈操作。栈的初态和终态均为空, 入栈和出栈的操作序列可表示为仅由 I 和 O 组成的序列, 可以操作的序列称为合法序列, 否则称为非法序列。

1) 下面所示的序列中哪些是合法的?

- A. IOIOIOIO B. IOOIOIOIO C. IIIIOIOIO D. IIIOOIOIO

2) 通过对 1) 的分析, 写出一个算法, 判定所给的操作序列是否合法。若合法, 返回 true, 否则返回 false (假定被判定的操作序列已存入一维数组中)。

4. 设单链表的表头指针为 L, 结点结构由 data 和 next 两个域构成, 其中 data 域为字符型。试设计算法判断该链表的全部 n 个字符是否中心对称。例如 xyx、xyyx 都是中心对称。

5. 设有两个栈 s_1 、 s_2 都采用顺序栈方式, 并且共享一个存储区 $[0, \dots, \text{maxsize}-1]$, 为了尽量利用空间, 减少溢出的可能, 可采用栈顶相向、迎面增长的存储方式。试设计 s_1 、 s_2 有关入栈和出栈的操作算法。

3.1.5 答案与解析

一、单项选择题

1. B

线性表、栈和队列的逻辑结构都是相同的, 都属于线性结构, 只是它们对数据的运算不同, 从而表现出不同的特点。

2. C

首先栈是一种线性表, 所以选项 B、D 错。按存储结构的不同可分为顺序栈和链栈, 但不可以把栈局限在某种存储结构上, 所以选项 A 错。栈和队列都是限制存取点的线性结构。

3. B

基本操作是指该结构最核心、最基本的运算,其他较复杂的操作可以通过基本操作实现。删除栈底元素不属于栈的基本运算,但它可以通过调用栈的基本运算求得。

4. C

入栈指针 top 加 1,初始时 top 为 -1,则第 1 个元素入栈后, top 为 0,即指向栈顶元素,故入栈时应先将指针 top 加 1,再将元素入栈,只有选项 C 符合题意。

5. A

每个元素需要 1 个存储单元,所以每入栈一次 top 加 1,出栈一次 top 减 1。指针 top 的值依次为 1001H, 1002H, 1001H, 1002H, 1001H, 1002H, 1001H, 1002H。

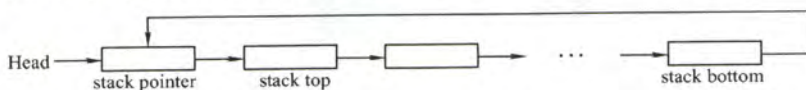
6. A

顺序栈采用数组存储,数组的大小是固定的,不能动态地分配大小。和顺序栈相比,链栈的最大优势在于它可以动态地分配存储空间,所以答案为选项 A。

7. C

通常栈的插入和删除在表头进行。对于选项 C,插入和删除一个结点后,仍需将其变为循环单链表,因此需要找到其尾结点,时间复杂度为 $O(n)$ 。

若不做题干中的限制,则栈顶可取表头(带头结点的链表)或第二个结点(不带头结点的链表),栈指针的位置取头结点(带头结点的链表)或表头(不带头结点的链表)。



8. C

链栈采用不带头结点的单链表表示时,进栈操作在首部插入一个结点 x (即 $x \rightarrow next = top$),插入完后需将 top 指向该插入的结点 x 。请思考当链栈存在头结点时的情况。

9. D

这里假设栈顶指针指向的是栈顶元素,所以选 D;而 A 中首先将 top 指针赋给了 x ,错误;B 中没有修改 top 指针的值;C 为 top 指针指向栈顶元素的上一个元素时的答案。

10. A

执行前 3 句后,栈 st 内的值为 a, b ,其中 b 为栈顶元素,执行第 4 句后, x 的值为 b ,执行最后一句后 x 的值为 a 。

11. B

对于 n 个不同元素进栈,出栈序列的个数为

$$\frac{1}{n+1} C_{2n}^n = \frac{1}{n+1} \frac{(2n)!}{n! \times n!} = \frac{6 \times 5 \times 4}{4 \times 3 \times 2 \times 1} = 5$$

上述的公式叫做卡特兰 (Catalan) 数,可采用数学归纳法证明,有兴趣的读者可以参考组合数学的教材(知道该公式即可)。考题中给出的 n 值不会很大,可以根据栈的特点,若 X_i 已经出栈,则 X_i 前面的尚未出栈的元素一定逆置有序地出栈,因此可以采取列举的方法。如 a, b, c 依次进栈的出栈序列有 abc, acb, bac, bca, cba 。另外,在一些考题中可能会问符合某个特定条件的出栈序列有多少种,比如此题中的问以 b 开头的出栈序列有几种,这种类型的题目一般都使用穷举法。

12. D

根据栈“先进后出”的特点,并且在进栈操作的同时允许出栈操作,显然,答案 D 中 c 最先出栈,则此时栈内必定为 a 和 b ,但由于 a 先于 b 进栈,故要晚出栈。对于某个出栈的元素,在

它之前进栈却晚出栈的元素必定是按逆序出栈的，其余答案均是可能出现的情况。

此题也可以采用将各序列逐个代入，以确定是否有对应的进出栈序列（类似下题）。

13. D

采用排除法，选项 A、B、C 得到的出栈序列分别为 1243、3241、1324。由 1234 得到 1342 的进出栈序列为：1 进，1 出，2 进，3 进，3 出，4 进，4 出，2 出，故选 D。

14. D

选项 A 可由 a 进，b 进，c 进，d 进，d 出，c 出，e 进，e 出，b 出，f 进，f 出，a 出得到；选项 B 可由 a 进，b 进，c 进，c 出，b 出，d 进，d 出，a 出，e 进，e 出，f 进，f 出得到；选项 C 可由 a 进，b 进，b 出，c 进，c 出，a 出，d 进，e 进，e 出，f 进，f 出，d 出得到；选项 D 可由 a 进，a 出，b 进，c 进，d 进，e 进，f 进，f 出，e 出，d 出，c 出，b 出得到，但要求不允许连续 3 次退栈操作，故选 D。

15. C

本题将队列和栈结合起来，由于队列“先进先出”的特性，所以出队的序列与进队的序列是相同的，从而可以得到出栈的次序为 e2、e4、e3、e6、e5、e1；再由栈“先进后出”的特性，进栈的次序为 e1、e2、e3、e4、e5、e6，可见，排在某个元素之后出栈却排在它之前进栈的元素个数，表示之前栈内的元素个数。因此，e4 进栈后，e1 和 e3 在栈中；而后 e4 和 e3 出栈；e6 进栈后，e5 和 e1 也在栈中。因此，栈的最小容量为 3。

16. C

时刻注意栈的特点是先进后出，下表是出入栈的详细过程。

序号	说明	栈内	栈外	序号	说明	栈内	栈外
1	a 入栈	a		8	e 入栈	ae	bdc
2	b 入栈	ab		9	f 入栈	acf	bdc
3	b 出栈	a	b	10	f 出栈	ae	bdcf
4	c 入栈	ac	b	11	e 出栈	a	bdcfe
5	d 入栈	acd	b	12	a 出栈		bdcfea
6	d 出栈	ac	bd	13	g 入栈	g	bdcfea
7	c 出栈	a	bdc	14	g 出栈		bdcfeag

栈内的最大深度为 3，故栈 S 的容量至少是 3。

另解：由于元素的出队顺序和入队顺序是一样的，所以，元素的出栈顺序也就是 b、d、c、f、e、a、g，所以，元素的入栈出栈顺序为 Push(S,a)，Push(S,b)，Pop(S,b)，Push(S,c)，Push(S,d)，Pop(S,d)，Pop(S,c)，Push(S,e)，Push(S,f)，Pop(S,f)，Pop(S,e)，Pop(S,a)，Push(S,g)，Pop(S,g)。假设初始所需容量为 0，每做一次 Push 操作进行一次加 1 操作，每做一次 Pop 进行一次减 1 操作，记录容量的最大值为 3，所以答案选 C。

17. C

在确保队列先进先出原则的前提下。根据题意具体分析：入队顺序为 8、4、2、5、3、9、1、6、7，出队顺序为 1~9。入口和出口之间有多条队列（n 条轨道），且每个队列（轨道）可容纳多个元素（多列列车）。如此分析：显然先入队的元素必须小于后入队的元素（否则，如果 8 和 4 入同一队列，8 在前 4 在后，那么出队时只能是 8 在前 4 在后），这样 8 入队列 1，4 入队列 2，2 入队列 3，5 入队列 2（按照前面的原则“大的元素在小的元素后面”也可以将 5 入队列 3，但这时剩下的元素 3 就必须放到一个新的队列里面，无法确保“至少”，本应该将 5 入队列 2，再将 3 入队列 3，不增加新队列的情况下，可以满足题意“至少”的要求），3 入队列 3，9 入队列 1，

这时共占了3个队列，后面还有元素1，直接再占用一个新的队列4，1从队列4出队后，剩下的元素6和7或者入队到队列2或者入队到队列3（为简单起见我们不妨设n个队列的序号分别为1、2、…、n），这样就可以满足题目的要求。综上，共占用了4个队列。当然还有其他的入队出队的情况，请考生们自己推演。但要确保满足：1）队列中后面的元素大于前面的元素；2）确保占用最少（即满足题目中的“至少”）的队列。

18. D

第n个元素先出栈，说明前n-1个元素都已经按顺序入栈，由“先进后出”的特点可知，此时的输出序列一定是输入序列的逆序，故答案选D。

19. D

当第i个元素第一个出栈时，则i之前的元素可以依次排在i之后出栈，但剩余的元素可以在此时进栈并且也会排在i之前的元素出栈，所以，第j个出栈的元素是不确定的。

20. C

对于A，可能的顺序是a入，a出，b入，b出，c入，c出，d入，d出。对于B，可能的顺序是a入，b入，c入，c出，b出，d入，d出，a出。对于D，可能的顺序是a入，a出，b入，c入，c出，b出，d入，d出。C没有对应的序列。

另解：若出栈序列的第一个元素为d，则出栈序列只能是dcba。该思想通常也适用于出栈序列的局部分析：如12345入栈，问出栈序列34152是否正确？如何分析？若第一个出栈元素是3，则此时12必停留在栈中，它们出栈的相对顺序只能是21，故34152错误。

21. C

入栈序列是 P_1, P_2, \dots, P_n 。由于 $P_3=1$ ，即 P_1, P_2, P_3 连续入栈后，第一个出栈元素是 P_3 ，说明 P_1, P_2 已经按序进栈，根据先进后出的特点可知， P_2 必定在 P_1 之前出栈，而第二个出栈元素是2，而此时 P_1 必定不是栈顶元素，因为 P_2 尚未出栈。因此 P_1 的值不可能是2。

22. C

对于A，可能的顺序是1入栈，1出栈，2入栈，2出栈，3入栈，3出栈，4入栈，4出栈。对于B，可能的顺序是1入栈，2入栈，3入栈，3出栈，2出栈，4入栈，4出栈，1出栈。对于D，可能的顺序是1入栈，1出栈，2入栈，3入栈，3出栈，2出栈，4入栈，4出栈。C则没有对应的序列。

23. C

显然，3之后的4,5,⋯,n都是 P_3 可取的数（一直进栈直到该数入栈后马上出栈）。接下来分析1和2： P_1 可以是3之前入栈的数（可能是1或2）也可以是4，当 $P_1=1$ 时， P_3 可取2；当 $P_1=2$ 时， P_3 可取1；当 $P_1=4$ 时， P_3 可取除1、3、4之外的所有数；故 P_3 可能取除3之外的所有数，个数为n-1。

24. C

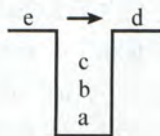
标识符的第一个字符必须是大小写英文字母或者下划线，而不能是数字。按照上述规定n1_三个字符符合规定的标识符有n1_、n_1、_1n、_n1四种形式。第一种：n进栈再出栈，1进栈再出栈，_进栈再出栈。第二种：n进栈再出栈，1进栈，_进栈，_出栈，1出栈。第三种：n进栈，1进栈，_进栈，_出栈，1出栈，n出栈。而根据栈的操作特性，_n1这种情况不可能出现，故选C。

25. B

出栈顺序必为d_c_b_a_，e的顺序不定，在任意一个“_”上都有可能。

另解: d 首先出栈, 则 abc 停留在栈中, 此时栈的状态如左图所示。

此时可以有如下 4 种操作: ① e 进栈后出栈, 则出栈序列为 decba; ② c 出栈, e 进栈后出栈, 出栈序列为 dceba; ③ cb 出栈, e 进栈后出栈, 出栈序列为 dcbea; ④ cba 出栈, e 进栈后出栈, 出栈序列为 dcbae。



第 25 题图

26. B

存取栈中的元素都只需要 $O(1)$ 的时间, 所以, 减少存取时间无从谈起。

另外, 栈的插入和删除操作都是在栈顶进行的, 只可能发生上溢 (栈顶指针超出了最大范围), 因此本题答案为 B。

27. C

I 的反例: 计算斐波拉契数列迭代实现只需要一个循环即可实现。III 的反例: 入栈序列为 1、2, 进行如下操作 PUSH、PUSH、POP、POP, 出栈次序为 2、1; 进行如下操作 PUSH、POP、PUSH、POP, 出栈次序为 1、2。IV, 栈是一种受限的线性表, 只允许在一端进行操作。II 正确。

28. A

这种情况就是前面我们所描述的, 详细内容请参见本节考点精析部分对共享栈的讲解。另外, 读者可以思考若 top1 的初值为 0, top2 的初值为 $n-1$ 时栈满的条件。

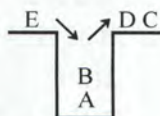
注意: 栈顶、队头与队尾的指针的定义是不唯一的, 读者务必要仔细审题。

二、综合应用题

1. 解答:

CD 出栈后的状态如右图所示。

此时有如下 3 种操作: ① E 进栈后出栈, 则出栈序列为 CDEBA; ② B 出栈, E 进栈后出栈, 出栈序列为 CDBEA; ③ B 出栈, A 出栈, E 进栈后出栈, 出栈序列为 CDBAE。



第 1 题图

所以, 以 CD 开头的出栈序列有: CDEBA、CDBEA、CDBAE 三种。

2. 解答:

能得到出栈序列 BCAED。可由 A 进, B 进, B 出, C 进, C 出, A 出, D 进, E 进, E 出, D 出得到。不能得到出栈序列 DBACE。若出栈序列以 D 开头, 说明在 D 之前的入栈元素是 A、B 和 C, 三个元素中 C 是栈顶元素, B 和 A 不可能早于 C 出栈, 故不可能得到出栈序列 DBACE。

3. 解答:

1) A、D 合法, 而 B、C 不合法。在 B 中, 先入栈 1 次, 再连续出栈 2 次, 错误。在 C 中, 入栈和出栈次数不一致, 会导致最终的栈不空。A、D 均为合法序列, 请自行模拟。注意: 在操作过程中, 入栈次数一定大于或等于出栈次数; 结束时, 栈一定为空。

2) 设被判定的操作序列已存入一维数组 A 中。算法的基本设计思想: 依次逐一扫入栈出栈序列 (即由 “I” 和 “O” 组成的字符串), 每扫描至任一位置均需检查出栈次数 (即 “O” 的个数) 是否小于入栈次数 (“I” 的个数), 若大于则为非法序列。扫描结束后, 再判断入栈和出栈次数是否相等, 若不相等则不合题意, 为非法序列。

```
int Judge(char A[ ]){
//判断字符串 A 中的输入输出序列是否是合法序列。如是, 返回 true, 否则返回 false
    int i=0;
    int j=k=0;
    while(A[i]!='\0'){
        switch(A[i]){
            case 'I': j++; break; //i 为下标, j 和 k 分别为字母 I 和 O 的个数
            case 'O': k++; break; //未到字符串末尾
        }
    }
    if(j==k) return true;
    else return false;
}
```



```

        case 'O': k++;
        if(k>j){printf("序列非法\n"); exit(0); }
    }
    i++; //不论A[i]是"I"或"O", 指针i均后移
} //while
if(j!=k){
    printf("序列非法\n");
    return false;
}
else{
    printf("序列合法\n");
    return true;
}
}

```

另解：入栈后，栈内元素个数加 1；出栈后，栈内元素个数减 1，因此，可以将判定一组出入栈序列是否合法，转化为一组+1、-1 组成的序列，它的任意前缀子序列的累加和不小于 0（每次出栈或入栈操作后判断），则合法；否则，非法。

4. 解答：

算法思想：使用栈来判断链表中的数据是否中心对称。将链表的前一半元素依次进栈。在处理链表的后一半元素时，当访问到链表的一个元素后，就从栈中弹出一个元素，两个元素比较，若相等，则将链表中下一个元素与栈中再弹出的元素比较，直至链表到尾。这时若栈是空栈，则得出链表中心对称的结论；否则，当链表中的一个元素与栈中弹出元素不等时，结论为链表非中心对称，结束算法的执行。

```

int dc(LinkList L,int n){
//L 是带头结点的 n 个元素单链表，本算法判断链表是否是中心对称
    int i;
    char s[n/2]; // s 字符栈
    p=L->next; //p 是链表的工作指针，指向待处理的当前元素
    for(i=0;i<n/2;i++){ //链表前一半元素进栈
        s[i]=p->data;
        p=p->next;
    }
    i--; //恢复最后的 i 值
    if(n%2==1) //若 n 是奇数，后移过中心结点
        p=p->next;
    while(p!=NULL&& s[i]==p->data){ //检测是否中心对称
        i--; //i 充当栈顶指针
        p=p->next;
    }
    if(i==--1) //栈为空栈
        return 1; //链表中心对称
    else
        return 0; //链表不中心对称
}

```

算法先将“链表的前一半”元素（字符）进栈。当 n 为偶数时，前一半和后一半的个数相同；当 n 为奇数时，链表中心结点字符不必比较，移动链表指针到下一字符开始比较。比较过程中遇到不相等时，立即退出 while 循环，不再进行比较。

本题也可以先将单链表中的元素全部入栈，然后再次扫描单链表 L 并比较，直到比较到单链

表 L 尾为止,但是算法需要两次扫描单链表 L,效率不及上述算法高。

5. 解答:

两个栈共享向量空间,将两个栈的栈底设在向量两端,初始时,s1 栈顶指针为-1,s2 栈顶指针为 maxsize。两个栈顶指针相邻时为栈满。两个栈顶相向、迎面增长,栈顶指针指向栈顶元素。

```
#define maxsize 100           //两个栈共享顺序存储空间所能达到的最多元素数,
                                //初始化为 100
#define elemtp int           //假设元素类型为整型
typedef struct{
    elemtp stack[maxsize];    //栈空间
    int top[2];               //top 为两个栈顶指针
}stk;
stk s;                        //s 是如上定义的结构类型变量,为全局变量
```

本题的关键在于,两个栈入栈和退栈时的栈顶指针的计算。s1 栈是通常意义下的栈;而 s2 栈入栈操作时,其栈顶指针左移(减 1),退栈时,栈顶指针右移(加 1)。

此外,对于所有栈的操作,都要注意“入栈判满、出栈判空”的检查。

(1) 入栈操作

```
int push(int i, elemtp x){
    //入栈操作。i 为栈号, i=0 表示左边的 s1 栈, i=1 表示右边的 s2 栈, x 是入栈元素
    //入栈成功返回 1, 否则返回 0
    if(i<0||i>1){
        printf("栈号输入不对");
        exit(0);
    }
    if(s.top[1]-s.top[0]==1){
        printf("栈已满\n");
        return 0;
    }
    switch(i){
        case 0: s.stack[++s.top[0]]=x; return 1; break;
        case 1: s.stack[--s.top[1]]=x; return 1;
    }
}
```

(2) 退栈操作

```
elemtp pop(int i){
    //退栈算法。i 代表栈号, i=0 时为 s1 栈, i=1 时为 s2 栈
    //退栈成功返回退栈元素, 否则返回-1
    if(i<0||i>1){
        printf("栈号输入错误\n");
        exit(0);
    }
    switch(i){
        case 0:
            if(s.top[0]==-1){
                printf("栈空\n");
                return -1;
            }
        else
```

```

        return s.stack[s.top[0]--];
    case 1:
        if(s.top[1]==maxsize){
            printf("栈空\n");
            return -1;
        }
        else
            return s.stack[s.top[1]++];
    } //switch
}

```

3.2 队列

3.2.1 队列的基本概念

1. 队列的定义

队列 (Queue): 队列简称队, 也是一种操作受限的线性表, 只允许在表的一端进行插入, 而在表的另一端进行删除。向队列中插入元素称为入队或进队; 删除元素称为出队或离队。这和我们日常生活中的排队是一致的, 最早排队的也是最早离队的。

其操作的特性是先进先出 (First In First Out, FIFO), 故又称为先进先出的线性表, 如图 3-5 所示。

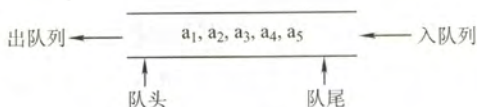


图 3-5 队列的示意图

队头 (Front): 允许删除的一端, 又称为队首。

队尾 (Rear): 允许插入的一端。

空队列: 不含任何元素的空表。

2. 队列常见的基本操作

InitQueue(&Q): 初始化队列, 构造一个空队列 Q。

QueueEmpty(Q): 判队列空, 若队列 Q 为空返回 true, 否则返回 false。

EnQueue(&Q, x): 入队, 若队列 Q 未满, 将 x 加入, 使之成为新的队尾。

DeQueue(&Q, &x): 出队, 若队列 Q 非空, 删除队头元素, 并用 x 返回。

GetHead(Q, &x): 读队头元素, 若队列 Q 非空, 则将队头元素赋值给 x。

需要注意的是, 队列是操作受限的线性表, 所以, 不是任何对线性表的操作都可以作为队列的操作。比如, 不可以随便读取队列中间的某个数据。

3.2.2 队列的顺序存储结构

1. 队列的顺序存储

队列的顺序实现是指分配一块连续的存储单元存放队列中的元素, 并附设两个指针 front 和 rear 分别指示队头元素和队尾元素的位置。设队头指针指向队头元素, 队尾指针指向队尾元素的下一个位置 (也可以让 rear 指向队尾元素, front 指向队头元素的前一个位置, 对于这种设置方法, 请读者以图 3-6 为例思考出队和入队后这两个指针的变化)。

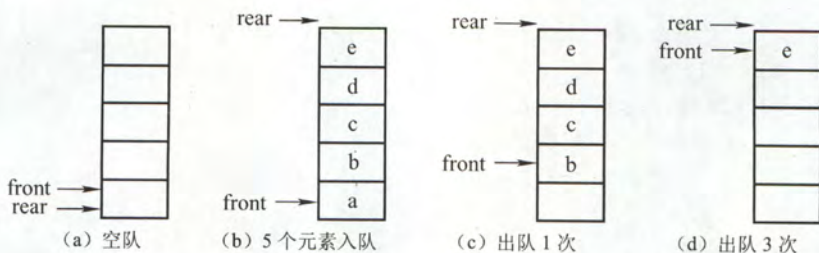


图 3-6 队列的操作

队列的顺序存储类型可描述为

```
#define MaxSize 50                //定义队列中元素的最大个数
typedef struct{
    ElemType data[MaxSize];        //存放队列元素
    int front, rear;               //队头指针和队尾指针
} SqQueue;
```

初始状态（队空条件）： $Q.front = Q.rear = 0$ 。

进队操作：队不满时，先送值到队尾元素，再将队尾指针加 1。

出队操作：队不空时，先取队头元素值，再将队头指针加 1。

如图 3-6 (a) 所示为队列的初始状态，有 $Q.front = Q.rear = 0$ 成立，该条件可以作为队列判空的条件。但能否用 $Q.rear = \text{MaxSize}$ 作为队列满的条件呢？显然不能，图 3-6 (d) 中，队列中仅有 1 个元素，但仍满足该条件。这时入队出现“上溢出”，但这种溢出并不是真正的溢出，在 data 数组中依然存在可以存放元素的空位置，所以是一种“假溢出”。

2. 循环队列

前面已指出了顺序队列的缺点，这里我们引出循环队列的概念。将顺序队列臆造为一个环状的空间，即把存储队列元素的表从逻辑上看成一个环，称为循环队列。当队首指针 $Q.front = \text{MaxSize} - 1$ 后，再前进一个位置就自动到 0，这可以利用除法取余运算 (%) 来实现。

初始时： $Q.front = Q.rear = 0$

队首指针进 1： $Q.front = (Q.front + 1) \% \text{MaxSize}$

队尾指针进 1： $Q.rear = (Q.rear + 1) \% \text{MaxSize}$

队列长度： $(Q.rear + \text{MaxSize} - Q.front) \% \text{MaxSize}$

出队入队时：指针都按顺时针方向进 1（如图 3-7 所示）。

那么，循环队列队空和队满的判断条件是什么呢？显然，队空的条件是 $Q.front = Q.rear$ 。如果入队元素的速度快于出队元素的速度，队尾指针很快就赶上了队首指针，如图 3-7 (d1) 所示，此时可以看出队满时也有 $Q.front = Q.rear$ 。循环队列出入队示意图如图 3-7 所示。

为了区分队空还是队满的情况，有三种处理方式：

1) 牺牲一个单元来区分队空和队满，入队时少用一个队列单元，这是一种较为普遍的做法，约定以“队头指针在队尾指针的下一位置作为队满的标志”，如图 3-7 (d2) 所示。

队满条件为： $(Q.rear + 1) \% \text{MaxSize} = Q.front$ 。

队空条件仍为： $Q.front = Q.rear$ 。

队列中元素的个数： $(Q.rear - Q.front + \text{MaxSize}) \% \text{MaxSize}$

2) 类型中增设表示元素个数的数据成员。这样，则队空的条件为 $Q.size = 0$ ；队满的条件为

$Q.size == \text{MaxSize}$ 。这两种情况都有 $Q.front == Q.rear$ 。

3) 类型中增设 tag 数据成员, 以区分是队满还是队空。tag 等于 0 的情况下, 若因删除导致 $Q.front == Q.rear$ 则为队空; tag 等于 1 的情况下, 若因插入导致 $Q.front == Q.rear$ 则为队满。

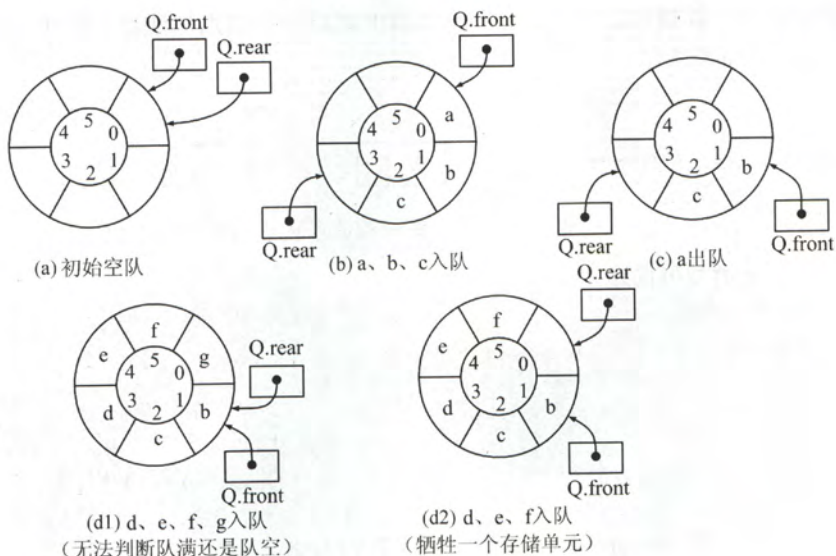


图 3-7 循环队列出入队示意图

3. 循环队列的操作

(1) 初始化

```
void InitQueue(&Q) {
    Q.rear = Q.front = 0;           // 初始化队首、队尾指针
}
```

(2) 判队空

```
bool isEmpty(Q) {
    if (Q.rear == Q.front) return true; // 队空条件
    else return false;
}
```

(3) 入队

```
bool EnQueue(SqQueue &Q, ElemType x) {
    if ((Q.rear + 1) % MaxSize == Q.front) return false; // 队满
    Q.data[Q.rear] = x;
    Q.rear = (Q.rear + 1) % MaxSize; // 队尾指针加 1 取模
    return true;
}
```

(4) 出队

```
bool DeQueue(SqQueue &Q, ElemType &x) {
    if (Q.rear == Q.front) return false; // 队空, 报错
    x = Q.data[Q.front];
    Q.front = (Q.front + 1) % MaxSize; // 队头指针加 1 取模
    return true;
}
```


3.2.3 队列的链式存储结构

1. 队列的链式存储

队列的链式表示称为链队列，它实际上是一个同时带有队头指针和队尾指针的单链表。头指针指向队头结点，尾指针指向队尾结点，即单链表的最后一个结点（注意与顺序存储的不同）。队列的链式存储如图 3-8 所示。

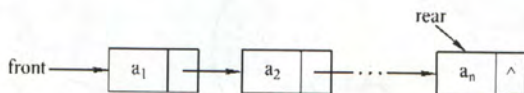


图 3-8 不带头结点的链式队列

队列的链式存储类型可描述为

```
typedef struct{                               //链式队列结点
    ElemType data;
    struct LinkNode *next;
}LinkNode;
typedef struct{                               //链式队列
    LinkNode *front,*rear;                   //队列的队头和队尾指针
}LinkQueue;
```

当 $Q.front == NULL$ 且 $Q.rear == NULL$ 时，链式队列为空。

出队时，首先判断队是否为空，若不为空，则取出队头元素，将其从链表中摘除，并让 $Q.front$ 指向下一个结点（若该结点为最后一个结点，则置 $Q.front$ 和 $Q.rear$ 都为 $NULL$ ）。入队时，建立一个新结点，将新结点插入到链表的尾部，并改让 $Q.rear$ 指向这个新插入的结点（若原队列为空队，则令 $Q.front$ 也指向该结点）。

不难看出，不设头结点的链式队列在操作上往往比较麻烦，因此，通常将链式队列设计成一个带头结点的单链表，这样插入和删除操作就统一了，如图 3-9 所示。

用单链表表示的链式队列特别适合于数据元素变动比较大的情形，而且不存在队列满且产生溢出的问题。另外，假如程序中要使用多个队列，与多个栈的情形一样，最好使用链式队列，这样就不会出现存储分配不合理和“溢出”的问题。

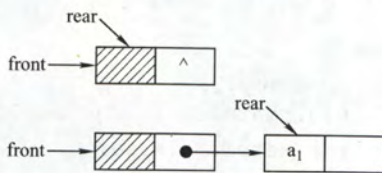


图 3-9 带头结点的链式队列

2. 链式队列的基本操作

(1) 初始化

```
void InitQueue(LinkQueue &Q){
    Q.front=Q.rear=(LinkNode*)malloc(sizeof(LinkNode)); //建立头结点
    Q.front->next=NULL; //初始为空
}
```

(2) 判队空

```
bool IsEmpty(LinkQueue Q){
    if(Q.front==Q.rear) return true;
    else return false;
}
```

(3) 入队

```
void EnQueue(LinkQueue &Q,ElemType x){
```

```

s=(LinkNode *)malloc(sizeof(LinkNode));
s->data=x; s->next=NULL;      //创建新结点, 插入到链尾
Q.rear->next=s;
Q.rear=s;
}

```

(4) 出队

```

bool DeQueue(LinkQueue &Q, ElemType &x){
    if(Q.front==Q.rear) return false; //空队
    p=Q.front->next;
    x=p->data;
    Q.front->next=p->next;
    if(Q.rear==p)
        Q.rear=Q.front;      //若原队列中只有一个结点, 删除后变空
    free(p);
    return true;
}

```

3.2.4 双端队列

双端队列是指允许两端都可以进行入队和出队操作的队列, 如图 3-10 所示。其元素的逻辑结构仍是线性结构。将队列的两端分别称为前端和后端, 两端都可以入队和出队。

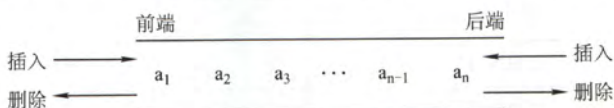


图 3-10 双端队列

在双端队列进队时: 前端进的元素排列在队列中后端进的元素的前面, 后端进的元素排列在队列中前端进的元素的后面。在双端队列出队时: 无论前端还是后端出队, 先出的元素排列在后出的元素的前面。思考: 如何由入队序列 a,b,c,d 得到出队序列 d,c,a,b?

输出受限的双端队列: 允许在一端进行插入和删除, 但在另一端只允许插入的双端队列称为输出受限的双端队列, 如图 3-11 所示。

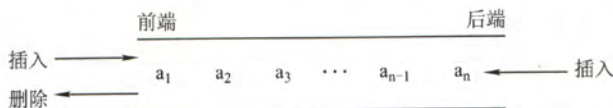


图 3-11 输出受限的双端队列

输入受限的双端队列: 允许在一端进行插入和删除, 但在另一端只允许删除的双端队列称为输入受限的双端队列, 如图 3-12 所示。而如果限定双端队列从某个端点插入的元素只能从该端点删除, 则该双端队列就蜕变为两个栈底相邻接的栈了。

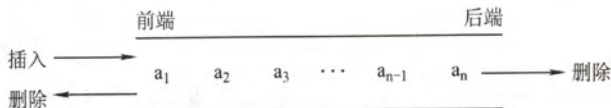


图 3-12 输入受限的双端队列

例 设有一个双端队列, 输入序列为 1,2,3,4, 试分别求出以下条件的输出序列。

- (1) 能由输入受限的双端队列得到,但不能由输出受限的双端队列得到的输出序列。
- (2) 能由输出受限的双端队列得到,但不能由输入受限的双端队列得到的输出序列。
- (3) 既不能由输入受限的双端队列得到,也不能由输出受限的双端队列得到的输出序列。

解答:



图 3-13 输入受限的双端队列



图 3-14 输出受限的双端队列

先看输入受限的双端队列,如图 3-13 所示。假设 end1 端输入 1, 2, 3, 4, 那么 end2 端的输出相当于队列的输出: 1, 2, 3, 4; 而 end1 端的输出相当于栈的输出, $n=4$ 时仅通过 end1 端有 14 种输出序列(由前面提到的 **Catalan** 公式计算得出), 仅通过 end1 端不能得到的输出序列有 $4!-14=10$ 种, 它们是:

1, 4, 2, 3 2, 4, 1, 3 3, 4, 1, 2 3, 1, 4, 2 3, 1, 2, 4
4, 3, 1, 2 4, 1, 3, 2 4, 2, 3, 1 4, 2, 1, 3 4, 1, 2, 3

通过 end1 和 end2 端混合输出, 可以输出这 10 种中的 8 种, 参看下表。其中, SL、XL 分别代表 end1 端的进队和出队, XR 代表 end2 端的出队。

输出序列	进队出队顺序	输出序列	进队出队顺序
1, 4, 2, 3	SLXRSLSLXLXRXR	3, 1, 2, 4	SLSLSRXLXRSRXXR
2, 4, 1, 3	SLSLXLSLSLXLXRXR	4, 1, 2, 3	SLSLSLXLXLXRXRXR
3, 4, 1, 2	SLSLSLXLXLXRXR	4, 1, 3, 2	SLSLSLXLXLXRXLXR
3, 1, 4, 2	SLSLSLXLXRSXLXR	4, 3, 1, 2	SLSLSLXLXLXRXR

剩下两种是不能通过输入受限的双端队列输出的, 即 4, 2, 3, 1 和 4, 2, 1, 3。

再看输出受限的双端队列, 如图 3-14 所示。假设 end1 端和 end2 端都能输入, 仅 end2 端可以输出。如果都从 end2 端输入, 从 end2 端输出, 就是一个栈了。当输入序列为 1, 2, 3, 4 时, 输出序列有 14 种。对于其他 10 种不能得到的输出序列, 通过交替从 end1 和 end2 端输入, 还可以输出其中 8 种。设 SL 代表 end1 端的输入, SR、XR 分别代表 end2 端的输入和输出, 则可能的输出序列及进队和出队顺序见下表。

输出序列	进队出队顺序	输出序列	进队出队顺序
1, 4, 2, 3	SLXRSLSLSRXXRXXR	3, 1, 2, 4	SLSLSRXXRSLXRXR
2, 4, 1, 3	SLSRXRSRXXRXXR	4, 1, 2, 3	SLSLSLXXRXXRXXR
3, 4, 1, 2	SLSLSRXRSRXXRXXR	4, 2, 1, 3	SLSRSLRXRXXRXXR
3, 1, 4, 2	SLSLSRXXRSRXXR	4, 3, 1, 2	SLSLSRXXRXXRXXR

通过输出受限的双端队列不能得到的两种输出序列是 4, 1, 3, 2 和 4, 2, 3, 1。

综上所述:

- 1) 能由输入受限的双端队列得到, 但不能由输出受限的双端队列得到的是 4, 1, 3, 2。
- 2) 能由输出受限的双端队列得到, 但不能由输入受限的双端队列得到的是 4, 2, 1, 3。
- 3) 既不能由输入受限的双端队列得到, 又不能由输出受限的双端队列得到的是 4, 2, 3, 1。

3.2.5 本节试题精选

一、单项选择题

- 栈和队列的主要区别在于 ()。
 - 它们的逻辑结构不一样
 - 它们的存储结构不一样
 - 所包含的元素不一样
 - 插入、删除操作的限定不一样
- 队列的“先进先出”特性是指 ()。
 - 最后插入队列中的元素总是最后被删除
 - 当同时进行插入、删除操作时，总是插入操作优先
 - 每当有删除操作时，总要先做一次插入操作
 - 每次从队列中删除的总是最早插入的元素
 - I
 - I 和 IV
 - II 和 III
 - IV
- 允许对队列进行的操作有 ()。
 - 对队列中的元素排序
 - 取出最近进队的元素
 - 在队列元素之间插入元素
 - 删除队头元素
- 一个队列的入队顺序是 1、2、3、4，则出队的输出顺序是 ()。
 - 4、3、2、1
 - 1、2、3、4
 - 1、4、3、2
 - 3、2、4、1
- 循环队列存储在数组 $A[0 \dots n]$ 中，则入队时的操作为 ()。
 - $rear=rear+1$
 - $rear=(rear+1) \bmod (n-1)$
 - $rear=(rear+1) \bmod n$
 - $rear=(rear+1) \bmod (n+1)$
- 已知循环队列的存储空间为数组 $A[21]$ ，front 指向队头元素的前一个位置，rear 指向队尾元素，假设当前 front 和 rear 的值分别为 8 和 3，则该队列的长度为 ()。
 - 5
 - 6
 - 16
 - 17
- 若用数组 $A[0..5]$ 来实现循环队列，且当前 rear 和 front 的值分别为 1 和 5，当从队列中删除一个元素，再加入两个元素后，rear 和 front 的值分别为 ()。
 - 3 和 4
 - 3 和 0
 - 5 和 0
 - 5 和 1
- 【2011 年计算机联考真题】

已知循环队列存储在一维数组 $A[0 \dots n-1]$ 中，且队列非空时 front 和 rear 分别指向队头元素和队尾元素。若初始时队列为空，且要求第一个进入队列的元素存储在 $A[0]$ 处，则初始时 front 和 rear 的值分别是 ()。

 - 0, 0
 - 0, $n-1$
 - $n-1$, 0
 - $n-1$, $n-1$
- 假设一个循环队列 $Q[\text{MaxSize}]$ 的队头指针为 front，队尾指针为 rear，队列的最大容量为 MaxSize，除此之外，该队列再没有其他数据成员，则判断该队的列满条件是 ()。
 - $Q.\text{front}=Q.\text{rear}$
 - $Q.\text{front}+Q.\text{rear} \geq \text{MaxSize}$
 - $Q.\text{front}=(Q.\text{rear}+1) \% \text{MaxSize}$
 - $Q.\text{rear}=(Q.\text{front}+1) \% \text{MaxSize}$
- 【2014 年计算机联考真题】

循环队列放在一维数组 $A[0 \dots M-1]$ 中，end1 指向队头元素，end2 指向队尾元素的后一个位置。假设队列两端均可进行入队和出队操作，队列中最多能容纳 $M-1$ 个元素。初始时为空。下列判断队空和队满的条件中，正确的是 ()。

 - 队空: $\text{end1} = \text{end2}$;
 - 队满: $\text{end1} = (\text{end2}+1) \% M$
 - 队空: $\text{end1} = \text{end2}$;
 - 队满: $\text{end2} = (\text{end1}+1) \% (M-1)$

- C. 队空: $\text{end2} = (\text{end1} + 1) \bmod M$; 队满: $\text{end1} = (\text{end2} + 1) \bmod M$
 D. 队空: $\text{end1} = (\text{end2} + 1) \bmod M$; 队满: $\text{end2} = (\text{end1} + 1) \bmod (M - 1)$
11. 最适合用做链队的链表是 ()。
 A. 带队首指针和队尾指针的循环单链表
 B. 带队首指针和队尾指针的非循环单链表
 C. 只带队首指针的非循环单链表
 D. 只带队首指针的循环单链表
12. 最不适合用做链式队列的链表是 ()。
 A. 只带队首指针的非循环双链表
 B. 只带队首指针的循环双链表
 C. 只带队尾指针的循环双链表
 D. 只带队尾指针的循环单链表
13. 在用单链表实现队列时, 队头在链表的 () 位置。
 A. 链头
 B. 链尾
 C. 链中
 D. 以上都可以
14. 用链式存储方式的队列进行删除操作时需要 ()。
 A. 仅修改头指针
 B. 仅修改尾指针
 C. 头尾指针都要修改
 D. 头尾指针可能都要修改
15. 在一个链队列中, 假设队头指针为 front, 队尾指针为 rear, x 所指向的元素需要入队, 则需要执行的操作为 ()。
 A. $\text{front} = x, \text{front} = \text{front} \rightarrow \text{next}$
 B. $x \rightarrow \text{next} = \text{front} \rightarrow \text{next}, \text{front} = x$
 C. $\text{rear} \rightarrow \text{next} = x, \text{rear} = x$
 D. $\text{rear} \rightarrow \text{next} = x, x \rightarrow \text{next} = \text{null}, \text{rear} = x$
16. 假设循环单链表表示的队列长度为 n, 队头固定在链表表尾, 若只设头指针, 则进队操作的时间复杂度为 ()。
 A. $O(n)$
 B. $O(1)$
 C. $O(n^2)$
 D. $O(n \log_2 n)$
17. 若以 1、2、3、4 作为双端队列的输入序列, 则既不能由输入受限的双端队列得到, 也不能由输出受限的双端队列得到的输出序列是 ()。
 A. 1,2,3,4
 B. 4,1,3,2
 C. 4,2,3,1
 D. 4,2,1,3
18. 【2010 年计算机联考真题】
 某队列允许在其两端进行入队操作, 但仅允许在一端进行出队操作。若元素 a、b、c、d、e 依次入此队列后再进行出队操作, 则不可能得到的出队序列是 ()。
 A. bacde
 B. dbace
 C. dbcae
 D. ecbad

二、综合应用题

1. 如果希望循环队列中的元素都能得到利用, 则需设置一个标志域 tag, 并以 tag 的值为 0 或 1 来区分队头指针 front 和队尾指针 rear 相同时的队列状态是“空”还是“满”。试编写与此结构相应的入队和出队算法。

2. Q 是一个队列, S 是一个空栈, 实现将队列中的元素逆置的算法。

3. 利用两个栈 S1、S2 来模拟一个队列, 已知栈的 4 个运算定义如下:

Push(S, x);	// 元素 x 入栈 S
Pop(S, x);	// S 出栈并将出栈的值赋给 x
StackEmpty(S);	// 判断栈是否为空
StackOverflow(S);	// 判断栈是否满

那么如何利用栈的运算来实现该队列的 3 个运算 (形参由读者根据要求自己设计):

Enqueue;	// 将元素 x 入队
Dequeue;	// 出队, 并将出队元素存储在 x 中

`QueueEmpty;`

//判断队列是否为空

3.2.6 答案与解析

一、单项选择题

1. D

栈和队列的逻辑结构都是线性表，它们的存储结构可能是顺序的，也可能是链式的，但不能说是它们的主要区别，C的道理也是一样，只有D才是栈和队列的本质区别。不管是顺序存储还是链式存储，栈和队列都只能顺序存取，而向量数组是直接（随机）存取。

2. B

队列的先进先出特性表现在：先进队列的元素先出队列，后进队列的元素后出队列，进队列对应的就是插入操作，出队列对应的就是删除操作。I和IV均正确。

3. D

删除队头元素即出队，是队列的基本操作之一，故选D。

4. B

队列的入队顺序和出队顺序是一致的，这是和栈不同的。

5. D

这道题需要注意的是，由于数组的下标范围为 $0 \sim n$ ，所以数组的容量为 $n+1$ 。

循环队列中新元素入队时的操作是 $\text{rear} = (\text{rear} + 1) \text{ MOD } \text{maxsize}$ ，本题中 $\text{maxsize} = n+1$ 。因此入队操作应为 $\text{rear} = (\text{rear} + 1) \text{ MOD } (n + 1)$ 。

6. C

队列的长度为 $(\text{rear} - \text{front} + \text{maxsize}) \% \text{maxsize} = (\text{rear} - \text{front} + 21) \% 21 = 16$ 。这种情况和front指向当前元素，rear指向队尾元素的下一个元素的计算是相同的。

7. B

循环队列中，每删除一个元素，队首指针： $\text{front} = (\text{front} + 1) \% 6$ ，每插入一个元素，队尾指针： $\text{rear} = (\text{rear} + 1) \% 6$ 。上述操作后， $\text{front} = 0$ 、 $\text{rear} = 3$ 。

8. B

根据题意，第一个元素进入队列后存储在 $A[0]$ 处，此时front和rear值都为0。入队时由于要执行 $(\text{rear} + 1) \% n$ 操作，所以如果入队后指针指向0，则rear初值为 $n-1$ ，而由于第一个元素在 $A[0]$ 中，插入操作只改变rear指针，所以front为0不变。

9. C

既然不能附加任何其他数据成员，只能采用牺牲一个存储单元的方法来区分队空还是队满，约定以“队列头指针在队尾指针的下一位置作为队满的标志”，因此选C。选项A是判断队列是否空的条件，选项B和D都是干扰项。

10. A

end1指向队头元素，可知出队的操作是先从 $A[\text{end1}]$ 读数，然后end1再加1。end2指向队尾元素的后一个位置，可知入队操作是先存数到 $A[\text{end2}]$ ，然后end2再加1。若把 $A[0]$ 储存第一个元素，当队列初始时，入队操作是先把数据放到 $A[0]$ ，然后end2自增，即可知end2初值为0；而end1指向的是队头元素，队头元素在数组A中的下标为0，所以得知end1初值也为0，可知队空条件为 $\text{end1} = \text{end2}$ ；然后考虑队列满时，因为队列最多能容纳 $M-1$ 个元素，假设队列存储在下标为0到下标为 $M-2$ 的 $M-1$ 个区域，队头为 $A[0]$ ，队尾为 $A[M-2]$ ，此时队列满，考虑在这种情况下end1和end2的状态，end1指向队头元素，可知 $\text{end1} = 0$ ，end2指向队尾元素的后一

个位置, 可知 $\text{end2} = M - 2 + 1 = M - 1$, 所以可知队满的条件为 $\text{end1} = (\text{end2} + 1) \bmod M$, 选 A。

注意: 考虑这类具体问题时, 用一些特殊情况判断往往比直接思考问题能更快地得到答案, 并可以画出简单的草图以方便解题。

11. B

由于队列需在双端进行操作, 选项 C 和 D 的链表显然不太适合链队。选项 A 的链表在完成进队和出队后还要修改为循环的, 对于队列来讲这是多余的。对于选项 B, 由于有首指针, 适合删除首结点; 由于有尾指针, 适合在其后插入结点, 故选 B。

12. A

由于非循环双链表只带队首指针, 在执行入队操作时需要修改队尾结点的指针域, 而查找队尾结点需要 $O(n)$ 的时间。B、C 和 D 均可在 $O(1)$ 的时间内找到队首和队尾。

13. A

由于在队头做出队操作, 为了便于删除队头元素, 故总是选择链头作为队头。

14. D

队列用链表形式存储时, 删除元素时, 从队头删除, 一般情况下, 仅需修改头指针即可, 但如果此时队列中仅有一个元素, 则尾指针也需要被修改, 因为仅有一个元素时, 删除后队列为空, 需修改尾指针为 $\text{rear} = \text{front}$ 。

15. D

插入操作时, 先将结点 x 插入到链表尾部, 再让 rear 指向这个结点 x 。C 的做法不够严密, 因为是队尾, 所以队尾 $x \rightarrow \text{next}$ 必须置为空。

16. A

依题意, 进队操作是在队尾进行的, 即单链表表头。题中已明确说明只设有头指针, 也就是没有头结点和尾指针, 进队后, 循环单链表须保持循环的性质, 在只带头指针的循环单链表中寻找表尾结点的时间复杂度为 $O(n)$, 故进队的时间复杂度为 $O(n)$ 。

17. C

使用排除法。先看可由输入受限的双端队列产生的序列: 设右端输入受限, 1、2、3、4 依次左入, 则依次左出可得到 4、3、2、1, 排除 A; 右出、左出、右出、右出可得到 4、1、3、2, 排除 B; 再看可由输出受限的双端队列产生的序列: 设右端输出受限, 1、2、3、4 依次左入、左入、右入、左入, 依次左出可得到 4、2、1、3, 排除 D。

18. C

本题的队列实际上是一个输出受限的双端队列, 如图 3-11 所示。A 操作: a 左入 (或右入)、b 左入、c 右入、d 右入、e 右入。B 操作: a 左入 (或右入)、b 左入、c 右入、d 左入、e 右入。D 操作: a 左入 (或右入)、b 左入、c 左入、d 右入、e 左入。C 操作: a 左入 (或右入)、b 右入、因 d 未出, 此时只能进队, c 怎么进都不可能在 b 和 a 之间。

二、综合应用题

1. 解答:

在循环队列的类型结构中, 增设一个 tag 的整型变量, 进队时置 tag 为 1, 出队时置 tag 为 0 (因为只有入队操作可能导致队满, 也只有出队操作可能导致队空)。队列 Q 初始时, 置 $\text{tag} = 0$ 、 $\text{front} = \text{rear} = 0$ 。这样队列的 4 要素如下:

队空条件: $Q.\text{front} = Q.\text{rear}$ 且 $Q.\text{tag} = 0$ 。

队满条件: $Q.\text{front} = Q.\text{rear}$ 且 $Q.\text{tag} = 1$ 。

进队操作: $Q.\text{data}[Q.\text{rear}] = x$; $Q.\text{rear} = (Q.\text{rear} + 1) \% \text{MaxSize}$; $Q.\text{tag} = 1$ 。

出队操作: $x=Q.data[Q.front]$; $Q.front=(Q.front+1)\%MaxSize$; $Q.tag=0$ 。

1) 设“tag”法的循环队列入队算法:

```
int EnQueue1(SqQueue &Q, ElemType x){
    if (Q.front==Q.rear&&Q.tag==1)
        return 0;           //两个条件都满足时则队满
    Q.data[Q.rear]=x;
    Q.rear=(Q.rear+1)%MaxSize;
    Q.tag=1;                 //可能队满
    return 1;
}
```

2) 设“tag”法的循环队列出队算法:

```
int DeQueue1(SqQueue &Q, ElemType &x){
    if (Q.front==Q.rear&&Q.tag==0)
        return 0;           //两个条件都满足时则队空
    x=Q.data[Q.front];
    Q.front=(Q.front+1)%MaxSize;
    Q.tag=0;                 //可能队空
    return 1;
}
```

2. 解答:

本题主要考查大家对队列和栈的特性和操作的理解。只是对队列的一系列操作是不可能将其中的元素逆置的,而栈可以将入栈的元素逆序提取出来。所以,我们可以将队列中的元素逐个地出队列,入栈;全部入栈后再逐个出栈,入队列。

算法的实现如下:

```
void Inverser(Stack S, Queue Q){
    //本算法实现将队列中的元素逆置
    while(!QueueEmpty(Q)){
        x=DeQueue(Q);           //队列中全部元素依次出队
        Push(S, x);             //元素依次入栈
    }
    while(!StackEmpty(S)){
        Pop(S, x);               //栈中全部元素依次出栈
        EnQueue(Q, x);           //再入队
    }
}
```

3. 解答:

利用两个栈 S1 和 S2 来模拟一个队列,当需要向队列中插入一个元素时,用 S1 来存放已输入的元素,即 S1 执行入栈操作。当需要出队时,则对 S2 执行出栈操作。由于从栈中取出元素的顺序是原顺序的逆序,所以,必须先将 S1 中的所有元素全部出栈并入栈到 S2 中,再在 S2 中执行出栈操作,即可实现出队操作,而在执行此操作前必须判断 S2 是否为空,否则会导致顺序混乱。当栈 S1 和 S2 都为空时队列为空。

总结如下:

1) 对 S2 的出栈操作做出队,若 S2 为空,则先将 S1 中的所有元素送入 S2。

2) 对 S1 的入栈操作做出入队,若 S1 满,必须先保证 S2 为空,才能将 S1 中的元素全部插入 S2 中。

入队算法:


```

int EnQueue(Stack &S1, Stack &S2, ElemType e) {
    if (!StackOverflow(S1)) {
        Push(S1, e);
        return 1;
    }
    if (StackOverflow(S1) && !StackEmpty(S2)) {
        printf("队列满");
        return 0;
    }
    if (StackOverflow(S1) && StackEmpty(S2)) {
        while (!StackEmpty(S1)) {
            Pop(S1, x);
            Push(S2, x);
        }
    }
    Push(S1, e);
    return 1;
}

```

出队算法:

```

void DeQueue(Stack &S1, Stack &S2, ElemType &x) {
    if (!StackEmpty(S2)) {
        Pop(S2, x);
    }
    else if (StackEmpty(S1)) {
        printf("队列为空");
    }
    else {
        while (!StackEmpty(S1)) {
            Pop(S1, x);
            Push(S2, x);
        }
        Pop(S2, x);
    }
}

```

判断队列为空的算法:

```

int QueueEmpty(Stack S1, Stack S2) {
    if (StackEmpty(S1) && StackEmpty(S2))
        return 1;
    else
        return 0;
}

```

3.3 栈和队列的应用

要熟练掌握栈和队列, 必须学习栈和队列的应用, 把握其中的规律, 然后才能举一反三。接下来将简单介绍栈和队列的一些常见应用。

3.3.1 栈在括号匹配中的应用

假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序任意，即 $[()]$ 或 $[([)]]$ 等均为正确的格式， $[()]$ 或 $[()]$ 或 $([)]$ 均为不正确的格式。

考虑下列括号序列：

[([]	[])]
1	2	3	4	5	6	7	8

分析如下：

- 1) 计算机接收第1个括号“[”后，期待与之匹配的第8个括号“]”出现。
- 2) 获得了第2个括号“(”，此时第1个括号“[”暂时放在一边，而急迫期待与之匹配的第7个括号“)”的出现。
- 3) 获得了第3个括号“[”，此时第2个括号“(”暂时放在一边，而急迫期待与之匹配的第4个括号“]”的出现。第3个括号的期待得到满足，消解之后，第2个括号的期待匹配又成为当前最急迫的任务了。

4) 依此类推，可见，该处理过程与栈的思想吻合。

算法的思想如下：

- 1) 初始设置一个空栈，顺序读入括号。
- 2) 若是右括号，则或者使置于栈顶的最急迫期待得以消解，或者是不合法的情况（括号序列不匹配，退出程序）。
- 3) 若是左括号，则作为一个新的更急迫的期待压入栈中，自然使原有的在栈中的所有未消解的期待的急迫性降了一级。算法结束时，栈为空，否则括号序列不匹配。

3.3.2 栈在表达式求值中的应用

表达式求值是程序设计语言编译中一个最基本的问题，它的实现是栈应用的一个典型范例。中缀表达式不仅依赖运算符的优先级，而且还要处理括号。后缀表达式的运算符在操作数后面，在后缀表达式中已考虑了运算符的优先级，没有括号，只有操作数和运算符。中缀表达式 $A+B \times (C-D) - E/F$ 所对应的后缀表达式为 $ABCD - \times + EF / -$ 。

中缀表达式转化为后缀表达式的过程，见本章习题 3.3.6 第 11 题的解析，这里不再赘述。

读者也可以将后缀表达式与原运算式对应的表达式树（用来表示算术表达式的二元树，如图 3-15）的后序遍历进行比较，可以发现它们有异曲同工之妙。

通过后缀表示计算表达式值的过程为：

顺序扫描表达式的每一项，然后根据它的类型做如下相应操作：如果该项是操作数，则将其压入栈中；如果该项是操作符 $\langle op \rangle$ ，则连续从栈中退出两个操作数 Y 和 X ，形成运算指令 $X \langle op \rangle Y$ ，并将计算结果重新压入栈中。当表达式的所有项都扫描并处理完后，栈顶存放的就是最后的计算结果。

例如，后缀表达式 $ABCD - \times + EF / -$ 求值的过程需要 12 步，见表 3-1。

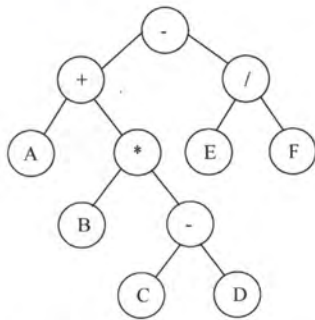


图 3-15 $A+B \times (C-D) - E/F$ 对应的表达式树

表 3-1 后缀表达式 ABCD-*+EF/-求值的过程

步	扫描项	项类型	动作	栈中内容
1			置空栈	空
2	A	操作数	进栈	A
3	B	操作数	进栈	AB
4	C	操作数	进栈	ABC
5	D	操作数	进栈	ABCD
6	-	操作符	D、C 退栈, 计算 C-D, 结果 R ₁ 进栈	ABR ₁
7	*	操作符	R ₁ 、B 退栈, 计算 B×R ₁ , 结果 R ₂ 进栈	AR ₂
8	+	操作符	R ₂ 、A 退栈, 计算 A+R ₂ , 结果 R ₃ 进栈	R ₃
9	E	操作数	进栈	R ₃ E
10	F	操作数	进栈	R ₃ EF
11	/	操作符	F、E 退栈, 计算 E/F, 结果 R ₄ 进栈	R ₃ R ₄
12	-	操作符	R ₄ 、R ₃ 退栈, 计算 R ₃ -R ₄ , 结果 R ₅ 进栈	R ₅

3.3.3 栈在递归中的应用

递归是一种重要的程序设计方法。简单地说, 如果在一个函数、过程或数据结构的定义中又应用了它自身, 那么这个函数、过程或数据结构称为是递归定义的, 简称递归。

它通常把一个大型的复杂问题, 层层转化为一个与原问题相似的规模较小的问题来求解, 递归策略只需少量的代码就可以描述出解题过程所需要的多次重复计算, 大大地减少了程序的代码量。但在通常情况下, 它的效率并不是太高。

以斐波那契数列为例, 其定义为

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

这就是递归的一个典型例子, 用程序来实现如下:

```
int Fib(n){                                     //斐波那契数列的实现
    if(n==0)
        return 0;                               //边界条件
    else if(n==1)
        return 1;                               //边界条件
    else
        return Fib(n-1)+Fib(n-2);               //递归表达式
}
```

必须注意递归模型不能是循环定义的, 其必须满足下面的两个条件:

- 递归表达式 (递归体)。
- 边界条件 (递归出口)。

递归的精髓在于能否将原始问题转换为属性相同但规模较小的问题。

在递归调用的过程中，系统为每一层的返回点、局部变量、传入实参等开辟了递归工作栈来进行数据存储，递归次数过多容易造成栈溢出等。而其效率不高的原因是递归调用过程中包含很多重复的计算。下面以 $n=5$ 为例，列出递归调用执行过程，如图 3-16 所示。

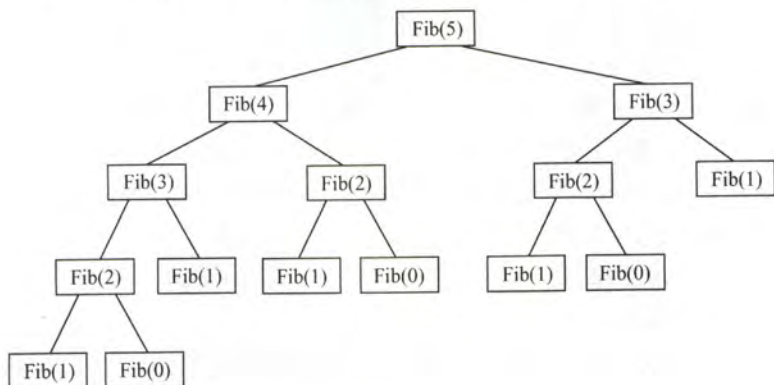


图 3-16 Fib(5)的递归执行过程

显然，在递归调用的过程中，Fib(3)被计算了 2 次，Fib(2)被计算了 3 次，Fib(1)被调用了 5 次，Fib(0)被调用了 3 次。所以，递归的效率低下，但优点是代码简单，容易理解。在第 4 章的树中利用了递归的思想，代码将会变得十分简单。通常情况下，初学者对于递归的调用过程很难理解，若读者想具体了解递归是如何实现的，可以参阅《编译原理》的相关内容。

可以将递归算法转换为非递归算法，通常需要借助栈来实现这种转换。

3.3.4 队列在层次遍历中的应用

在信息处理中有一大类问题需要逐层或逐行处理。这类问题的解决方法往往是在处理当前层或当前行时就对下一层或下一行做预处理，把处理顺序安排好，待当前层或当前行处理完毕，就可以处理下一层或下一行。使用队列是为了保存下一步的处理顺序。下面用二叉树（见图 3-17）层次遍历的例子，说明队列的应用。表 3-2 显示了层次遍历二叉树的过程。

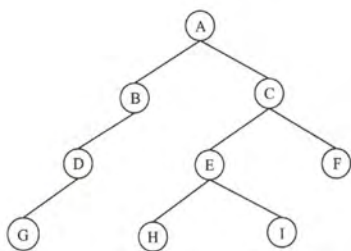


图 3-17 二叉树

表 3-2 层次遍历二叉树的过程

序	说明	队内	队外
1	A 入	A	
2	A 出, BC 入	BC	A
3	B 出, D 入	CD	AB
4	C 出, EF 入	DEF	ABC
5	D 出, G 入	EFG	ABCD
6	E 出, HI 入	FGHI	ABCDE
7	F 出	GHI	ABCDEF
8	GHI 出		ABCDEFGHI

该过程的简单描述如下：

① 根结点入队。

② 若队空（所有结点都已处理完毕），则结束遍历；否则重复③操作。

③ 队列中第一个结点出队，并访问之。若其有左孩子，则将左孩子入队；若其有右孩子，则将右孩子入队，返回②。

3.3.5 队列在计算机系统中的应用

队列在计算机系统中的应用非常广泛，以下仅从两个方面来简述队列在计算机系统中的作用：第一个方面是解决主机与外部设备之间速度不匹配的问题，第二个方面是解决由多用户引起的资源竞争问题。

对于第一个方面，仅以主机和打印机之间速度不匹配的问题为例作简要说明。主机输出数据给打印机打印，输出数据的速度比打印数据的速度要快得多，由于速度不匹配，若直接把输出的数据送给打印机打印显然是不行的。解决的方法是设置一个打印数据缓冲区，主机把要打印输出的数据依次写入到这个缓冲区中，写满后就暂停输出，转去做其他的事情。打印机就从缓冲区中按照先进先出的原则依次取出数据并打印，打印完后再向主机发出请求。主机接到请求后再向缓冲区写入打印数据。这样做既保证了打印数据的正确，又使主机提高了效率。由此可见，打印数据缓冲区中所存储的数据就是一个队列。

对于第二个方面，CPU（即中央处理器，它包括运算器和控制器）资源的竞争就是一个典型的例子。在一个带有多终端的计算机系统上，有多个用户需要 CPU 各自运行自己的程序，它们分别通过各自的终端向操作系统提出占用 CPU 的请求。操作系统通常按照每个请求在时间上的先后顺序，把它们排成一个队列，每次把 CPU 分配给队首请求的用户使用。当相应的程序运行结束或用完规定的时间间隔后，则令其出队，再把 CPU 分配给新的队首请求的用户使用。这样既满足了每个用户的请求，又使 CPU 能够正常运行。

3.3.6 本节试题精选

一、单项选择题

1. 栈的应用不包括（ ）。
A. 递归 B. 进制转换 C. 迷宫求解 D. 缓冲区
2. 表达式 $a*(b+c)-d$ 的后缀表达式是（ ）。
A. $abcd*+-$ B. $abc+*d-$ C. $abc*+d-$ D. $-+*abcd$
3. 下面（ ）用到了队列。
A. 括号匹配 B. 迷宫求解
C. 页面替换算法 D. 递归
4. 【2009 年计算机联考真题】

为解决计算机主机与打印机之间速度不匹配的问题，通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是（ ）。

- A. 栈 B. 队列 C. 树 D. 图
5. 利用栈求表达式的值时，设立运算数栈 OPEN。假设 OPEN 只有两个存储单元，则在下列表达式中，不会发生溢出的是（ ）。
A. $A-B*(C-D)$ B. $(A-B)*C-D$ C. $(A-B*C)-D$ D. $(A-B)*(C-D)$
 6. 执行完下列语句段后，i 的值为（ ）。

```
int f(int x){
    return ((x>0)? x*f(x-1):2);
}
int i;
i=f(f(1));
```

- A. 2 B. 4 C. 8 D. 无限递归
7. 一个问题的递归算法求解和其相对应的非递归算法求解, ()。
- A. 递归算法通常效率高一些 B. 非递归算法通常效率高一些
- C. 两者相同 D. 无法比较
8. 当执行函数时, 其局部变量的存储一般采用 () 进行存储。
- A. 树形结构 B. 静态链表 C. 栈结构 D. 队列结构
9. 执行 () 操作时, 需要使用队列作为辅助存储空间。
- A. 查找散列 (哈希) 表 B. 广度优先搜索图
- C. 前序 (根) 遍历二叉树 D. 深度优先搜索图
10. 下列说法中正确的是 ()。
- A. 消除递归不一定需要使用栈
- B. 对同一输入序列进行两组不同的合法入栈和出栈组合操作, 所得的输出序列也一定相同
- C. 通常使用队列来处理函数或过程调用
- D. 队列和栈都是运算受限的线性表, 只允许在表的两端进行运算
11. 【2012 年计算机联考真题】
- 已知操作符包括 '+', '-', '*', '/', '(', ')' 和 '^'。将中缀表达式 $a+b-a*((c+d)/e-f)+g$ 转换为等价的后缀表达式 $ab+acd+c/f-*g+$ 时, 用栈来存放暂时还不能确定运算次序的操作符。若栈初始时空, 则转换过程中同时保存在栈中的操作符的最大个数是 ()。
- A. 5 B. 7 C. 8 D. 11

12. 【2014 年计算机联考真题】

假设栈初始为空, 将中缀表达式 $a/b+(c*d-e*f)/g$ 转换为等价的后缀表达式的过程中, 当扫描到 f 时, 栈中的元素依次是 ()。

- A. $+(* -$ B. $+(- *$ C. $/+(* - *$ D. $/+ - *$

13. 【2015 年计算机联考真题】

已知程序如下:

```
int S(int n)
{ return (n<=0)?0:S(n-1)+n;}
void main()
{ cout<< S(1);}
```

程序运行时使用栈来保存调用过程的信息, 自栈底到栈顶保存的信息依次对应的是 ()。

- A. $main() \rightarrow S(1) \rightarrow S(0)$ B. $S(0) \rightarrow S(1) \rightarrow main()$
- C. $main() \rightarrow S(0) \rightarrow S(1)$ D. $S(1) \rightarrow S(0) \rightarrow main()$

二、综合应用题

1. 假设一个算术表达式中包含圆括号、方括号和花括号 3 种类型的括号, 编写一个算法来判别表达式中的括号是否配对, 以字符 "\0" 作为算术表达式的结束符。

2. 如图 3-18 所示铁道进行车厢调度 (注意: 两侧铁道均为单向行驶道, 火车调度站有一个用于调度的“栈道”), 火车调度站的入口处有 n 节硬座和软座车厢 (分别以 H 和 S 表示) 等待调度, 试编写算法, 输出对这 n 节车厢进行调度的操作 (即入栈或出栈操作) 序列, 以使所有的软座车厢都被调整到硬座车厢之前。

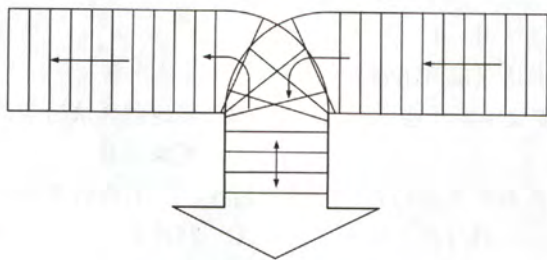


图 3-18 铁道进行车厢调度示意图

3. 利用一个栈实现以下递归函数的非递归计算:

$$P_n(x) = \begin{cases} 1 & n = 0 \\ 2x & n = 1 \\ 2xP_{n-1}(x) - 2(n-1)P_{n-2}(x) & n > 1 \end{cases}$$

4. 某汽车轮渡口, 过江渡船每次能载 10 辆车过江。过江车辆分为客车类和货车类, 上渡船有如下规定: 同类车先到先上船; 客车先于货车上渡船, 且每上 4 辆客车, 才允许放上一辆货车; 若等待客车不足 4 辆, 则以货车代替; 若无货车等待, 允许客车都上船。试设计一个算法模拟渡口管理。

3.3.7 答案与解析

一、单项选择题

1. D

缓冲区是用队列实现的, A、B、C 都是栈的典型应用。

2. B

后缀表达式中, 每一个计算符号均位于它两个操作数的直接后面, 按照这样的方式逐步根据计算的优先级将每个计算式进行变换, 即可得到后缀表达式。

另解: 将两个直接操作数用括号括起来, 再将操作符提出到括号后, 最后再去掉括号。如下: $(a * (b + c)) - d$, 提出操作符并去掉括号后, 可得后缀表达式为 $abc + * d -$ 。

3. C

页面替换算法中的 FIFO 用到了队列。其余的都只用到了栈。

4. B

在提取数据的时候必须保持原来数据的顺序, 所以缓冲区的特性是先进先出, 故排除 A、C、D, 答案只能是 B。

5. B

利用栈求表达式的值时, 可以分别设立运算符栈和运算数栈, 但其原理不变。选项 B 中 A 入栈, B 入栈, 计算得 R1, C 入栈, 计算得 R2, D 入栈, 计算得 R3, 由此得栈深为 2。A、C、D 依次计算得栈深为 4、3、3。因此选 B。

6. B

栈与递归有着紧密的联系。递归模型包括递归出口和递归体两个方面。递归出口是递归算法的出口, 也就是终止递归的条件。递归体是一个递推的关系式。根据题意有:

$$f(0) = 2;$$

$$f(1) = 1 * f(0) = 2;$$

$$f(f(1)) = f(2) = 2 * f(1) = 4;$$

即 $f(f(1)) = 4$ 。因此本题答案为 B。

7. B

通常情况下，递归算法在计算机实际执行的过程中包含很多的重复计算，所以效率会低。

8. C

调用函数时，系统会为调用者构造一个由参数表和返回地址组成的活动记录，并将记录压入系统提供的栈中，若被调用函数有局部变量，也要压入栈中。

9. B

图的广度优先搜索类似于树的层序遍历，同样需要借助于队列。

10. A

使用栈可以模拟递归的过程，以此来消除递归，但对于单向递归和尾递归而言，可以用迭代的方式来消除递归，A 对；不同的进栈和出栈组合操作，会产生许多不同的输出序列，B 错；通常使用栈来处理函数或过程调用，C 错；队列和栈都是操作受限的线性表，但只有队列允许在表的两端进行运算，而栈只允许在栈顶方向进行操作，D 错。

11. A

考查栈在中缀表达式转化为后缀表达式中的应用。将中缀表达式 $a+b-a*((c+d)/e-f)+g$ 转换为相应的后缀表达式，需要根据操作符 $\langle op \rangle$ 的优先级来进行栈的变化，我们用 isp 来表示当前扫描到的运算符 ch 的优先级，该运算符进栈后的优先级为 isp ，则运算符的优先级如下表所示： $(isp$ 是栈内优先 (in stack priority) 数， icp 是栈外优先 (in coming priority) 数。)

操作符	#	(*, /	+, -)
isp	0	1	5	3	6
icp	0	6	4	2	1

我们在表达式后面加上符号 ‘#’，表示表达式结束。具体转换过程如下：

步 骤	扫描项	项类型	动 作	栈内内容	输 出
0			‘#’ 进栈，读下一符号	#	
1	a	操作数	直接输出	#	a
2	+	操作符	$isp(‘#’) < icp(‘+’)$ ，进栈	#+	
3	b	操作数	直接输出	#+	b
4	-	操作符	$isp(‘+’) > icp(‘-’)$ ，退栈并输出	#	+
5			$isp(‘#’) < icp(‘-’)$ ，进栈	#-	
6	a	操作数	直接输出	#-	a
7	*	操作符	$isp(‘-’) < icp(‘*’)$ ，进栈	#-*	
8	(操作符	$isp(‘*’) < icp(‘(’)$ ，进栈	#-*(
9	(操作符	$isp(‘(’) < icp(‘(’)$ ，进栈	#-*((
10	c	操作数	直接输出	#-*((c
11	+	操作符	$isp(‘(’) < icp(‘+’)$ ，进栈	#-*((+	
12	d	操作数	直接输出	#-*((+	d
13)	操作符	$isp(‘+’) > icp(‘)’)，退栈并输出$	#-*((+
14			$isp(‘(’) = icp(‘)’)，直接退栈$	#-*(
15	/	操作符	$isp(‘(’) < icp(‘/’)$ ，进栈	#-*(/	
16	e	操作数	直接输出	#-*(/	e
17	-	操作符	$isp(‘/’) > icp(‘-’)$ ，退栈并输出	#-*(/

步 骤	扫描项	项类型	动 作	栈内内容	输 出
18			isp('(')<icp('-'), 进栈	#*(-	
19	f	操作数	直接输出	#*(-	f
20)	操作符	isp('-')>icp(')'), 退栈并输出	#*(-	-
21			isp('(')==icp(')'), 直接退栈	#-*	
22	+	操作符	isp('*')>icp('+'), 退栈并输出	#-	*
23			isp('-')>icp('+'), 退栈并输出	#	-
24			isp('#')<icp('+'), 进栈	#+	
25	g	操作数	直接输出	#+	g
26	#	操作符	isp('+')>icp('#'), 退栈并输出	#	+
27			isp('#')==icp('#'), 退栈, 结束		

即相应的后缀表达式为 $ab+acd+e/f-*-g+$ 。由上表可以看出, 第 11、12 步时栈中存放的操作符最多, 请注意题中明确表示了六种操作符, 而 '#' 不算, 即最大个数为 5。

12. B

将中缀表达式转换为后缀表达式的算法思想如下:

从左向右开始扫描中缀表达式;

遇到数字时, 加入后缀表达式;

遇到运算符时:

a. 若为 '(', 入栈;

b. 若为 ')', 则依次把栈中的运算符加入后缀表达式中, 直到出现 '(', 从栈中删除 '(';

c. 若为除括号外的其他运算符, 当其优先级高于除 '(' 以外的栈顶运算符时, 直接入栈。否则从栈顶开始, 依次弹出比当前处理的运算符优先级高和优先级相等的运算符, 直到一个比它优先级低的或者遇到了一个左括号为止。

当扫描的中缀表达式结束时, 栈中的所有运算符依次出栈加入后缀表达式。

待处理序列	栈	后缀表达式	当前扫描元素	动作
a/b+(c*d-e*f)/g			a	a 加入后缀表达式
/b+(c*d-e*f)/g		a	/	/入栈
b+(c*d-e*f)/g	/	a	b	b 加入后缀表达式
+(c*d-e*f)/g	/	ab	+	+优先级低于栈顶的/, 弹出/
+(c*d-e*f)/g		ab/	+	+入栈
(c*d-e*f)/g	+	ab/	((入栈
c*d-e*f)/g	+(ab/	c	c 加入后缀表达式
*d-e*f)/g	+(ab/c	*	栈顶为(, *入栈
d-e*f)/g	+(*	ab/c	d	d 加入后缀表达式
-e*f)/g	+(*	ab/cd	-	-优先级低于栈顶的*, 弹出*
-e*f)/g	+(ab/cd*	-	栈顶为(, -入栈
e*f)/g	+(ab/cd*	e	e 加入后缀表达式
*f)/g	+(ab/cd*e	*	*优先级高于栈顶的-, *入栈
f)/g	+(ab/cd*e	f	f 加入后缀表达式
/g	+(ab/cd*ef)	把栈中(之前的符号加入表达式
/g	+	ab/cd*ef*	/	/优先级高于栈顶的+, /入栈
g	+/	ab/cd*ef*	g	g 加入后缀表达式
	+/	ab/cd*ef*-g		扫描完毕, 运算符依次退栈加入表达式
		ab/cd*ef*-g/+		完成

由此可知,当扫描到 f 的时候,栈中的元素依次是+(-*, 选 B。

在此,再给出中缀表达式转换为前缀或后缀表达式的手工做法,以上面给出的中缀表达式为

例:

第一步:按照运算符的优先级对所有的运算单位加括号。

式子变成了: $((a/b)+(((c*d)-(e*f))/g))$

第二步:转换为前缀或后缀表达式。

前缀:把运算符移动到对应的括号前面,则变成了: $+/(ab)/(-(*cd)*(ef)g)$

把括号去掉: $+ab/-*cd*efg$ 前缀式子出现。

后缀:把运算符移动到对应的括号后面,则变成了: $((ab)/(((cd)*(ef)*-g)/)+$

把括号去掉: $ab/cd*cf*-g/+$ 后缀式子出现。

当题目要求直接求前缀或后缀表达式时,这种方法会比上一种快捷得多。

13. A

递归调用函数时,在系统栈里保存的函数信息需满足先进后出的特点,依次调用了 main()、S(1)、S(0),故栈底到栈顶的信息依次是 main()、S(1)、S(0)。

注意:在递归调用的过程中,系统为每一层的返回点、局部变量、传入实参等开辟了递归工作栈来进行数据存储。

二、综合应用题

1. 解答:

括号匹配是栈的一个典型应用,给出这道题是希望读者好好掌握栈的应用。算法的基本思想是扫描每个字符,遇到花、中、圆的左括号进栈,遇到花、中、圆的右括号时检查栈顶元素是否为相应的左括号,若是,退栈,否则配对错误。最后栈如果不为空也为错误。

```
bool BracketsCheck(char *str){
    InitStack(S);          //初始化栈
    int i=0;
    while(str[i]!='\0'){
        switch(str[i]){
            //左括号入栈
            case '(': Push(S,'('); break;
            case '[': push(S,'['); break;
            case '{': push(S,'{'); break;
            //遇到右括号,检测栈顶
            case ')': Pop(S,e);
                if(e!='(') return false;
                break;
            case ']': Pop(S,e);
                if(e!='[') return false;
                break;
            case '}': Pop(S,e);
                if(e!='{') return false;
                break;
            default:
                break;
        } //switch
        i++;
    } //while
}
```



```

if(!IsEmpty(S)){
    printf("括号不匹配\n");
    return false;
}
else {
    printf("括号匹配\n");
    return true;
}
}

```

2. 解答:

两侧的铁道均为单向行驶道,且两侧不相通。所有车辆都必须通过“栈道”进行调度。算法的基本设计思想:所有车厢依次前进并逐一检查,若为硬座车厢则入栈,等待最后调度。检查完后,所有的硬座车厢已全部入栈道,车道中的车厢均为软座车厢,此时将栈道的车厢调度出来,调整到软座车厢之后。算法的实现如下:

```

void Train_Arrange(char *train){
//用字符串 train 表示火车, H 表示硬座, S 表示软座
    char *p=train,*q=train,c;
    stack s;
    InitStack(s);           //初始化栈结构
    while(*p){
        if(*p=='H')
            Push(s,*p);     //把 H 存入栈中
        else
            *(q++)=*p;       //把 S 调到前部
        p++;
    }
    while(!StackEmpty(s)){
        Pop(s,c);
        *(q++)=c;           //把 H 接在后部
    }
}

```

3. 解答:

算法思想:设置一个栈用于保存 n 和对应的 $P_n(x)$ 值,栈中相邻元素的 $P_n(x)$ 有题中关系。然后边出栈边计算 $P_n(x)$,栈空后该值就计算出来了。算法的实现如下:

```

double p(int n,double x){
    struct stack{
        int no;           //保存 n
        double val;       //保存  $P_n(x)$  值
    }st[MaxSize];
    int top=-1,i;         //top 为栈 st 的下标值变量
    double fv1=1,fv2=2*x; //n=0,n=1 时的初值
    for(i=n;i>=2;i--){
        top++;
        st[top].no=i;
    }
    //入栈
    while(top>=0){
        st[top].val=2*x*fV2-2*(st[top].no-1)*fv1;
        fv1=fv2;
        fv2=st[top].val;
    }
}

```



```

        top--; //出栈
    }
    if(n==0){
        return fv1;
    }
    return fv2;
}

```

4. 解答:

算法思想: 假设数组 Q 的最大下标为 10, 恰好是每次载渡的最大量。假设客车的队列为 Q1, 货车的队列为 Q2。若 Q1 充足, 则每取 4 个 Q1 元素后再取一个 Q2 元素, 直到 Q 的长度为 10。若 Q1 不充足, 则直接用 Q2 补齐。算法的实现如下:

```

Queue q; //过江渡船载渡队列
Queue q1; //客车队列
Queue q2; //货车队列

void manager(){
    int i=0, j=0; //j 表示渡船上的总车辆数
    while(j<10){ //不足 10 辆时
        if(!QueueEmpty(q1)&&i<4){ //客车队列不空, 则未上足 4 辆
            DeQueue(q1, x); //从客车队列出队
            EnQueue(q, x); //客车上渡船
            i++; //客车数加 1
            j++; //渡船上的总车辆数加 1
        }
        else if(i==4&&!QueueEmpty(q2)){ //客车已上足 4 辆
            DeQueue(q2, x); //从货车队列出队
            EnQueue(q, x); //货车上渡船
            j++; //渡船上的总车辆数加 1
            i=0; //每上一辆货车, i 重新计数
        }
        else{ //其他情况 (客车队列空或货车队列空)
            while(j<10&&i<4&&!QueueEmpty(q2)){ //客车队列空
                DeQueue(q2, x); //从货车队列出队
                EnQueue(q, x); //货车上渡船
                i++; //i 计数, 当 i>4 时, 退出本循环
                j++; //渡船上的总车辆数加 1
            }
            i=0;
        }
        if(QueueEmpty(q1)&&QueueEmpty(q2))
            j=11; //如果货车和客车加起来不足 10 辆
    }
}

```

3.4 特殊矩阵的压缩存储

矩阵在计算机图形学、工程计算中占有举足轻重的地位。在数据结构中考虑的是如何用最小的内存空间来存储同样的一组数据。所以, 我们不需要研究矩阵及其运算等, 而把精力放在如何将矩阵更有效地存储在内存中, 并能方便地提取矩阵中的元素。

3.4.1 数组的定义

数组是由 n ($n \geq 1$) 个相同类型的数据元素构成的有限序列, 每个数据元素称为一个数组元素, 每个元素受 n 个线性关系的约束, 每个元素在 n 个线性关系中的序号称为该元素的下标, 并称该数组为 n 维数组。

数组与线性表的关系: 数组是线性表的推广。一维数组可以看做是一个线性表; 二维数组可以看做元素是线性表的线性表, 依此类推。数组一旦被定义, 它的维数和维界就不再改变。因此, 除了结构的初始化和销毁之外, 数组只有存取元素和修改元素的操作。

3.4.2 数组的存储结构

大多数计算机语言都提供了数组数据类型, 逻辑意义上的数组可以采用计算机语言中的数组数据类型进行存储, 一个数组的所有元素在内存中占用一段连续的存储空间。

以一维数组 $A[0 \cdots n-1]$ 为例, 其存储结构关系式为

$$\text{LOC}(a_i) = \text{LOC}(a_0) + (i) \times L \quad (0 \leq i < n)$$

其中, L 是每个数组元素所占存储单元。

对于多维数组, 有两种映射方法: 按行优先和按列优先。以二维数组为例, 按行优先存储的基本思想是: 先行后列, 先存储行号较小的元素, 行号相等先存储列号较小的元素。设二维数组的行下标与列下标的范围分别为 $[l_1, h_1]$ 与 $[l_2, h_2]$, 则存储结构关系式为

$$\text{LOC}(a_{i,j}) = \text{LOC}(a_{l_1,l_2}) + [(i-l_1) \times (h_2-l_2+1) + (j-l_2)] \times L$$

假设 l_1 和 l_2 的值均为 0, 则上式变为

$$\text{LOC}(a_{i,j}) = \text{LOC}(a_{0,0}) + [i \times (h_2+1) + j] \times L$$

例如, 对于数组 $A_{2 \times 3}$, 它按行优先方式在内存中的存储形式如图 3-19 所示。

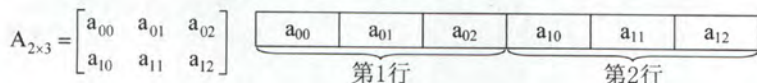


图 3-19 二维数组按行优先顺序存放

当以列优先方式存储时, 得出存储结构关系式为

$$\text{LOC}(a_{i,j}) = \text{LOC}(a_{l_1,l_2}) + [(j-l_2) \times (h_1-l_1+1) + (i-l_1)] \times L$$

例如, 对于数组 $A_{2 \times 3}$, 它按列优先方式在内存中的存储形式如图 3-20 所示。

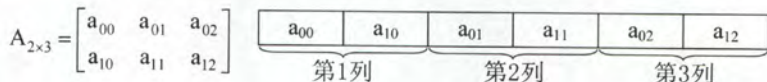


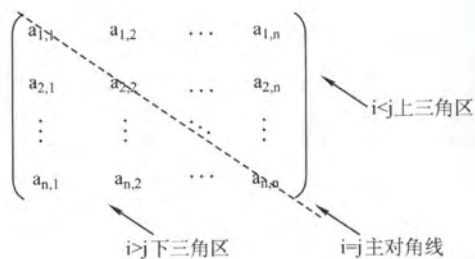
图 3-20 二维数组按列优先顺序存放

3.4.3 矩阵的压缩存储

压缩存储: 指为多个值相同的元素只分配一个存储空间, 对零元素不分配存储空间。其目的是为了节省存储空间。

特殊矩阵: 指具有许多相同矩阵元素或零元素, 并且这些相同矩阵元素或零元素的分布有一定规律性的矩阵。常见的特殊矩阵有对称矩阵、上(下)三角矩阵、对角矩阵等。

特殊矩阵的压缩存储方法: 找出特殊矩阵中值相同的矩阵元素的分布规律, 把那些呈现规律性分布的值相同的多个矩阵元素压缩存储到一个存储空间中。

图 3-21 n 阶方阵的划分

1. 对称矩阵

若一个 n 阶方阵 $A[1 \cdots n][1 \cdots n]$ 中的任一个元素 a_{ij} , 都有 $a_{ij} = a_{ji}$ ($1 \leq i, j \leq n$), 则称其为对称矩阵。对于一个 n 阶方阵, 其中元素可以划分为 3 个部分, 即上三角区、主对角线和下三角区, 如图 3-21 所示。

对于 n 阶对称矩阵, 上三角区所有元素和下三角区对应元素相同, 如果还采用二维数组存放, 就会浪费几乎一半的空间, 为此将对称矩阵 $A[1 \cdots n][1 \cdots n]$ 存放在一维数组 $B[n(n+1)/2]$ 中, 即元素 a_{ij} 存放在

b_k 中。只存放主对角线和下三角区的元素。

在数组 B 中, 位于元素 a_{ij} ($i \geq j$) 前面的元素个数为

第 1 行: 1 个元素 ($a_{1,1}$);

第 2 行: 2 个元素 ($a_{2,1}, a_{2,2}$);

.....

第 $i-1$ 行: $i-1$ 个元素 ($a_{i-1,1}, a_{i-1,2}, \dots, a_{i-1,i-1}$);

第 i 行: $j-1$ 个元素 ($a_{i,1}, a_{i,2}, \dots, a_{i,j-1}$)。

故, 元素 a_{ij} 在数组 B 中的下标 $k=1+2+\dots+(i-1)+j-1=i(i-1)/2+j-1$ (数组下标从 0 开始)。因此, 元素下标之间的对应关系如下:

$$k = \begin{cases} \frac{i(i-1)}{2} + j - 1 & \text{当 } i \geq j \text{ (下三角区和主对角线元素)} \\ \frac{j(j-1)}{2} + i - 1 & \text{当 } i < j \text{ (上三角区元素 } a_{ij} = a_{ji} \text{)} \end{cases}$$

当数组下标从 1 开始时, 可以采用同样的推导方法, 请读者自行思考。

2. 三角矩阵

下三角矩阵 (见图 3-23 (a)) 中, 上三角区的所有元素均为同一常量。其存储思想与对称矩阵类似, 不同之处在于存储完下三角区和主对角线上的元素之后, 紧接着存储对角线上方的常量一次, 故可以将下三角矩阵 $A[1 \cdots n][1 \cdots n]$ 压缩存储在 $B[n(n+1)/2+1]$ 中。

元素下标之间的对应关系为

$$k = \begin{cases} \frac{i(i-1)}{2} + j - 1 & \text{当 } i \geq j \text{ (下三角区和主对角线元素)} \\ \frac{n(n+1)}{2} & \text{当 } i < j \text{ (上三角区元素)} \end{cases}$$

下三角矩阵的在内存中的压缩存储形式如图 3-22 所示。

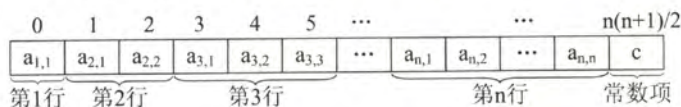


图 3-22 下三角矩阵的压缩存储

上三角矩阵 (见图 3-23 (b)) 中, 下三角区的所有元素均为同一常量。只需存储主对角线、上三角区上的元素和下三角区的常量一次, 可以将其压缩存储在 $B[n(n+1)/2+1]$ 中。

为

在数组 B 中, 位于元素 a_{ij} ($i \leq j$) 前面的元素个数

第 1 行: n 个元素

第 2 行: $n-1$ 个元素

.....

第 $i-1$ 行: $n-i+2$ 个元素

第 i 行: $j-i$ 个元素

故, 元素 a_{ij} 在数组 B 中的下标 $K=n+(n-1)+\cdots+(n-i+2)+(j-i+1)-1=(i-1)(2n-i+2)/2+(j-i)$ 。因此, 元素下标之间的对应关系如下:

$$k = \begin{cases} \frac{(i-1)(2n-i+2)}{2} + (j-i) & \text{当 } i \leq j \text{ (上三角区和主对角线元素)} \\ \frac{n(n+1)}{2} & \text{当 } i > j \text{ (下三角区元素)} \end{cases}$$

上三角矩阵的在内存中的压缩存储形式如图 3-24 所示。

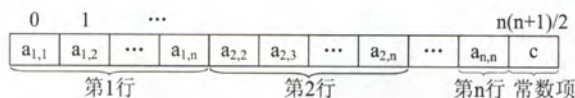


图 3-24 上三角矩阵的压缩存储

以上推导均假设数组的下标是从 0 开始, 若题设有具体要求, 则应该灵活应对。

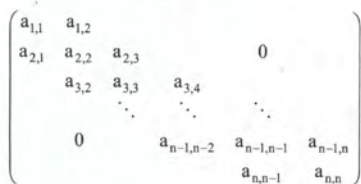


图 3-25 三对角矩阵 A

3. 三对角矩阵

对角矩阵也称为带状矩阵。对于 n 阶方阵 A 中的任一元素 a_{ij} , 当 $|i-j| > 1$ 时, 有 $a_{ij}=0$ ($1 \leq i, j \leq n$), 则称为三对角矩阵, 如图 3-25 所示。在三对角矩阵中, 所有非零元素都集中在以主对角线为中心的 3 条对角线的区域中, 其他区域的元素都为零。

三对角矩阵 A 也可以采用压缩存储, 将 3 条对角线上的元素按行优先方式存放在一维数组 B 中, 且 $a_{1,1}$ 存放于 B[0] 中, 其存储形式如图 3-26 所示。

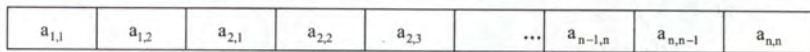


图 3-26 三对角矩阵的压缩存储

由此可以计算矩阵 A 中 3 条对角线上的元素 a_{ij} ($1 \leq i, j \leq n, |i-j| \leq 1$) 在一维数组 B 中存放的下标为 $k=2i+j-3$ 。

反之, 若已知三对角线矩阵中某元素 a_{ij} 在一维数组 B 中存放于第 k 个位置, 则可求得 $i=\lfloor (k+1)/3+1 \rfloor$, $j=k-2i+3$ 。例如, 当 $k=0$ 时, $i=\lfloor (0+1)/3+1 \rfloor=1$, $j=0-2 \times 1+3=1$, 存放的是 $a_{1,1}$; 当 $k=2$ 时, $i=\lfloor (2+1)/3+1 \rfloor=2$, $j=2-2 \times 2+3=1$, 存放的是 $a_{2,1}$; 当 $k=4$ 时, $i=\lfloor (4+1)/3+1 \rfloor=2$, $j=4-2 \times 2+3=3$, 存放的是 $a_{2,3}$ 。

3.4.4 稀疏矩阵

矩阵元素个数 s 相对于矩阵中非零元素个数 t 来说非常多, 即 $s \gg t$ 的矩阵称为稀疏矩阵。例如, 若一个矩阵的阶为 100×100 , 而该矩阵中只有少于 100 个非零元素。

如果采用常规的方法存储稀疏矩阵, 那将相当浪费存储空间, 因此仅存储非零元素。但通常零元素的分布没有规律, 所以, 仅存储非零元素的值是不够的, 还要存储它所在的行和列。因此, 将非零元素及其相应的行和列构成一个三元组 (行标, 列标, 值), 如图 3-27 所示, 然后再按照某种规律存储这些三元组。稀疏矩阵压缩存储后便失去了随机存取特性。

$$M = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 9 & 0 & 0 \\ 0 & 23 & 0 & 0 \end{bmatrix}$$
对应的三元组:

i	j	v
0	0	4
1	2	6
2	1	9
3	1	23

图 3-27 稀疏矩阵及其对应的三元组

3.4.5 本节试题精选

一、单项选择题

- 对特殊矩阵采用压缩存储的主要目的是 ()。
 - 表达变得简单
 - 对矩阵元素的存取变得简单
 - 去掉矩阵中的多余元素
 - 减少不必要的存储空间
- 对 n 阶对称矩阵压缩存储时, 需要表长为 () 的顺序表。
 - $n/2$
 - $n \times n/2$
 - $n(n+1)/2$
 - $n(n-1)/2$
- 设有一个 $n \times n$ 的对称矩阵 A , 将其下三角部分按行存放在一维数组 B 中, 而 $A[0][0]$ 存放于 $B[0]$ 中, 那么第 i 行的对角元素 $A[i][i]$ 存放于 B 中 () 处。
 - $(i+3)i/2$
 - $(i+1)i/2$
 - $(2n-i+1)i/2$
 - $(2n-i-1)i/2$
- 在一个二维数组 A 中, 假设每个数组元素的长度为 3 个存储单元, 行下标 i 为 $0 \sim 8$, 列下标 j 为 $0 \sim 9$, 从首地址 SA 开始连续存放。在这种情况下, 元素 $A[8][5]$ 的起始地址为 ()。
 - $SA+141$
 - $SA+144$
 - $SA+222$
 - $SA+255$
- 将一个 $A[1 \dots 100][1 \dots 100]$ 的三对角矩阵, 按行优先存入一维数组 $B[1..298]$ 中, A 中元素 $A[66][65]$ 在数组 B 中的位置 k 为 ()。
 - 198
 - 195
 - 197
 - 196
- 【2016 年计算机联考真题】

有一个 100 阶的三对角矩阵 M , 其元素 m_{ij} ($1 \leq i \leq 100, 1 \leq j \leq 100$) 按行优先依次压缩存入下标从 0 开始的一维数组 N 中。元素 $m_{30,30}$ 在 N 中的下标是 ()。

 - 86
 - 87
 - 88
 - 89
- 若将 n 阶上三角矩阵 A 按列优先级压缩存放在一维数组 $B[1 \dots n(n+1)/2+1]$ 中, 则存放到 $B[k]$ 中的非零元素 a_{ij} ($1 \leq i, j \leq n$) 的下标 i, j 与 k 的对应关系是 ()。
 - $i(i+1)/2+j$
 - $i(i-1)/2+j-1$
 - $j(j-1)/2+i$
 - $j(j-1)/2+i-1$
- 若将 n 阶下三角矩阵 A 按列优先顺序压缩存放在一维数组 $B[1 \dots n(n+1)/2+1]$ 中, 则存放到 $B[k]$ 中的非零元素 a_{ij} ($1 \leq i, j \leq n$) 的下标 i, j 与 k 的对应关系是 ()。
 - $(j-1)(2n-j+1)/2+i-j$
 - $(j-1)(2n-j+2)/2+i-j+1$
 - $(j-1)(2n-j+2)/2+i-j$
 - $(j-1)(2n-j+1)/2+i-j-1$
- 【2017 年计算机联考真题】

适用于压缩存储稀疏矩阵的两种存储结构是 ()。

- A. 三元组表和十字链表
C. 十字链表和二叉链表

- B. 三元组表和邻接矩阵
D. 邻接矩阵和十字链表

3.4.6 答案与解析

一、单项选择题

1. D

由于特殊矩阵中含有很多相同的元素或零元素，故可以给相同的元素只分配一次存储空间，从而达到节省内存的目的。

2. C

对于对称矩阵的压缩存储，只需存储其对应的上三角或者下三角矩阵即可，元素个数为 $n+(n-1)+(n-2)+\cdots+1=n(n+1)/2$ 。

3. A

此题要注意 3 个细节：矩阵的最小下标为 0；数组下标也是从 0 开始的；矩阵按行优先存在数组中。注意到此三点答案不难得到为 A。此外，本类题建议采用特殊值代入法求解，例如， $A[1][1]$ 对应的下标应为 2，代入后只有 A 满足条件。

技巧：对于特殊三角矩阵压缩存储的题，心中应有“平移”搬动的思想，并结合草图，这样会比较形象，在计算时再注意矩阵和数组的起始下标，就不容易出错。

4. D

二维数组计算地址（按行优先顺序）的公式为

$$LOC(i,j)=LOC(0,0)+(i \times m+j) \times L$$

其中， $LOC(0,0)=SA$ ，是数组存放的首地址； $L=3$ 是每个数组元素的长度； $m=9-0+1=10$ 是数组的列数。因此有 $LOC(8,5)=SA+(8 \times 10+5) \times 3=SA+255$ ，故选 D。

5. B

对于三对角矩阵，将 $A[1 \cdots n][1 \cdots n]$ 压缩至 $B[1 \cdots 3n-2]$ 时， a_{ij} 与 b_k 的对应关系为 $k=2i+j-2$ 。则 A 中的元素 $A[66][65]$ 在数组 B 中的位置 k 为 $2 \times 66+65-2=195$ 。

总结：矩阵的压缩存储首先应注意矩阵的形式（上三角、下三角、三对角等），其次注意下标是从 0 开始还是从 1 开始。

6. B

三对角矩阵如下图所示：

$$\begin{pmatrix} a_{1,1} & a_{1,2} & & & & \\ a_{2,1} & a_{2,2} & a_{2,3} & & & 0 \\ & a_{3,2} & a_{3,3} & a_{3,4} & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ & & & & a_{n,n-1} & a_{n,n} \end{pmatrix}$$

采用压缩存储，将 3 条对角线上的元素按行优先方式存放在一维数组 B 中，且 $a_{1,1}$ 存放于 $B[0]$ 中，其存储形式如下所示：

$a_{1,1}$	$a_{1,2}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	\dots	$a_{n-1,n}$	$a_{n,n-1}$	$a_{n,n}$
-----------	-----------	-----------	-----------	-----------	---------	-------------	-------------	-----------

可以计算矩阵 A 中 3 条对角线上的元素 a_{ij} ($1 \leq i, j \leq n$, $|i-j| \leq 1$) 在一维数组 B 中存放的下

标为 $k=2i+j-3$ 。

解法一：针对该题，仅需将数字逐一带到公式里面即可： $k=2*30+30-3=87$ ，结果为 87。

解法二：观察上图的三对角矩阵不难发现，第一行有两个元素，剩下的在元素 $m_{30,30}$ 所在行之前的 28 行（注意下标 $1 \leq i \leq 100$ 、 $1 \leq j \leq 100$ ）中每行都有 3 个元素，而 $m_{30,30}$ 之前仅有一个元素 $m_{30,29}$ ，那么不难发现元素 $m_{30,30}$ 在数组 N 中的下标是： $2+28*3+2-1=87$ 。

【注意】矩阵和数组的下标是从 0 或 1 开始的（如矩阵可能从 $a_{0,0}$ 或 $a_{1,1}$ 开始，数组可能从 $B[0]$ 或 $B[1]$ 开始），这时就需要适时调整计算方法（这个方法无非是针对上面提到的公式 $k=2*i+j-3$ 多计算 1 或少计算 1 的问题）。

7. C

由于按列优先存储，则元素 a_{ij} 前面有 $j-1$ 列，共有 $1+2+3+\cdots+j-1=j(j-1)/2$ 个元素，元素 a_{ij} 在第 j 列上是第 i 个元素，注意到数组 B 的下标是从 1 开始的，因此 $k=j(j-1)/2+i$ 。

8. B

由于按列优先的顺序，则元素 a_{ij} 的前面有 $j-1$ 列，共计 $n+(n-1)+\cdots+(n-j+2)=(j-1)(2n-j+2)/2$ 个元素，元素 a_{ij} 在第 j 列上是第 $i-j+1$ 个元素，注意到数组 B 的下标从 1 开始，因此， $k=(j-1)(2n-j+2)/2+i-j+1$ 。

9. A

三元组表的结点存储了行 row、列 col、值 value 三种信息，是主要用来存储稀疏矩阵的一种数据结构。十字链表将行单链表和列单链表结合起来存储稀疏矩阵。邻接矩阵空间复杂度达 $O(n^2)$ ，不适于存储稀疏矩阵。二叉链表又名左孩子右兄弟表示法，可用于表示树或森林。因此 A 正确。

归纳总结

本章所讲的几种数据结构类型是线性表的应用和推广，在考试中主要以选择题形式进行考查，但栈和队列也仍然有可能出现在算法设计题当中。很多读者看到课本上有好多个函数很恐惧，如果考到了栈或队列的大题，难道要把每个操作的函数都写出来吗？

其实，在考试中，栈或队列都是作为一个工具来解决其他问题的，我们可以把栈或队列的声明和操作写得很简单，不必分函数写出。以顺序栈的操作为例：

(1) 声明一个栈并初始化：

```
Elemtype stack[maxSize]; int top=-1; //两句话连声明带初始化都有了
```

(2) 元素进栈：

```
stack[++top]=x; //仅一句话即实现进栈操作。
```

(3) 元素 x 出栈：

```
x=stack[top--]; //双目运算符在变量之前表示“先运算后使用”，之后则相反
```

对于链式栈，同样只需定义一个结构体，然后从讲解中摘取必要的语句组合在自己的函数代码中即可。另外，在考研真题中链式栈出现的概率比顺序栈低得多，因此大家应该有所侧重，多训练顺序栈相关的题目。

思维拓展

设计一个栈，使它可以在 $O(1)$ 的时间复杂度内实现 push, pop 和 min 操作，所谓 min 操作就是得到栈中最小的元素。

(提示：使用双栈，两个栈是同步关系。主栈是普通栈，用来实现栈的基本操作 push 和 pop；辅助栈用来记录同步的最小值 min，如元素 x 进栈，则辅助栈 $\text{stack_min}[\text{top}++] = (x < \text{min}) ? x : \text{min}$ ；也就是说在每次 push 中，都将当前最小元素放到 stack_min 的栈顶。当在主栈中 pop 最小元素 y 时，stack_min 栈中相同位置的最小元素 y 也会随着 top-- 而出栈。那么 stack_min 的栈顶元素必然是 y 之前入栈的最小元素。本题是典型的以空间换时间的算法)

第4章 树与二叉树

【考纲内容】

(一) 树的基本概念

(二) 二叉树

二叉树的定义及其主要特征；二叉树的顺序存储结构和链式存储结构

二叉树的遍历；线索二叉树的基本概念和构造

(三) 树、森林

树的存储结构；森林与二叉树的转换；树和森林的遍历

(四) 树与二叉树的应用

二叉排序树；平衡二叉树；哈夫曼(Huffman)树和哈夫曼编码

【知识框架】



【复习提示】

本章内容通常以选择题的形式考查，但不排除会有算法题涉及树的遍历。树和二叉树的性质、遍历操作、转换、存储结构和操作特性等，满二叉树、完全二叉树、线索二叉树、哈夫曼树的定义和性质，二叉排序树和二叉平衡树的性质和操作等都是选择题必然会涉及的内容。本书 2014 版强调过树遍历算法的重要性，没能引起读者的足够重视，2014 年也是第一年考查非线性表的算法题。希望读者在复习专业课时不要抱有投机的想法。

4.1 树的基本概念

4.1.1 树的定义

树是 N ($N \geq 0$) 个结点的有限集合， $N=0$ 时，称为空树，这是一种特殊情况。在任意一棵非空树中应满足：

1) 有且仅有一个特定的称为根的结点。

2) 当 $N > 1$ 时, 其余结点可分为 m ($m > 0$) 个互不相交的有限集合 T_1, T_2, \dots, T_m , 其中每一个集合本身又是一棵树, 并且称为根结点的子树。

显然树的定义是递归的, 是一种递归的数据结构。树作为一种逻辑结构, 同时也是一种分层结构, 具有以下两个特点:

1) 树的根结点没有前驱结点, 除根结点之外的所有结点有且只有一个前驱结点。

2) 树中所有结点可以有零个或多个后继结点。

树适合于表示具有层次结构的数据。树中的某个结点 (除根结点外) 最多只和上一层的一个结点 (即其父结点) 有直接关系, 根结点没有直接上层结点, 因此在 n 个结点的树中有 $n-1$ 条边。而树中每个结点与其下一层的零个或多个结点 (即其子女结点) 有直接关系。

4.1.2 基本术语

下面结合图 4-1 中的树来说明一些基本术语和概念。

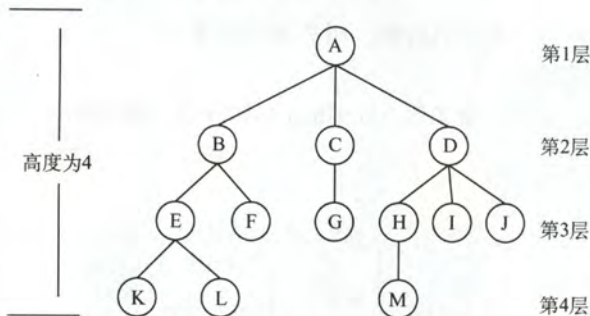


图 4-1 树的树形表示

1) 考虑结点 K。根 A 到结点 K 的唯一路径上的任意结点, 称为结点 K 的祖先结点。如结点 B 是结点 K 的祖先结点, 而结点 K 是结点 B 的子孙结点。路径上最接近结点 K 的结点 E 称为 K 的双亲结点, 而 K 为结点 E 的孩子结点。根 A 是树中唯一没有双亲的结点。有相同双亲的结点称为兄弟结点, 如结点 K 和结点 L 有相同的双亲结点 E, 即 K 和 L 为兄弟结点。

2) 树中一个结点的子结点数称为该结点的度, 树中结点的最大度数称为树的度。如结点 B 的度为 2, 结点 D 的度为 3, 树的度为 3。

3) 度大于 0 的结点称为分支结点 (又称非终端结点); 度为 0 (没有子女结点) 的结点称为叶子结点 (又称终端结点)。在分支结点中, 每个结点的分支数就是该结点的度。

4) 结点的深度、高度和层次。

结点的层次从树根开始定义, 根结点为第 1 层 (有些教材中将根结点定义为第 0 层), 它的子结点为第 2 层, 依此类推。

结点的深度是从根结点开始自顶向下逐层累加的。

结点的高度是从叶结点开始自底向上逐层累加的。

树的高度 (又称深度) 是树中结点的最大层数。图 4-1 中树的高度为 4。

5) 有序树和无序树: 树中结点的子树从左到右是有次序的, 不能交换, 这样的树叫做有序树, 有序树中, 一个结点其子结点按从左到右顺序出现是有关联的。反之则称为无序树。在图 4-1 中, 若将子结点位置互换, 则变成一棵不同的树。

6) 路径和路径长度: 树中两个结点之间的**路径**是由这两个结点之间所经过的结点序列构成的, 而**路径长度**是路径上所经过的边的个数。在图 4-1 中, 结点 A 和结点 K 的路径长度为 3, 中间经过结点 B 和结点 E。

注意: 由于树中的分支是有向的, 即从双亲结点指向孩子结点, 所以树中的路径是从上向下的, 同一双亲结点的两个孩子结点之间不存在路径。

7) 森林: 森林是 m ($m \geq 0$) 棵互不相交的树的集合。森林的概念与树的概念十分相近, 因为只要把树的根结点删去就成了森林。反之, 只要给 n 棵独立的树加上一个结点, 并把这 n 棵树作为该结点的子树, 则森林就变成了树。

注意: 上述概念无需刻意记忆, 根据实例理解即可。考研不大可能直接考查概念, 都是结合具体的题目考查。做题时, 遇到不熟悉的概念可以再翻书, 练习得多自然就记住了。

4.1.3 树的性质

树具有如下最基本的性质:

- 1) 树中的结点数等于所有结点的度数加 1。
- 2) 度为 m 的树中第 i 层上至多有 m^{i-1} 个结点 ($i \geq 1$)。
- 3) 高度为 h 的 m 叉树至多有 $(m^h - 1)/(m - 1)$ 个结点^①。
- 4) 具有 n 个结点的 m 叉树的最小高度为 $\lceil \log_m(n(m-1)+1) \rceil$ 。

4.1.4 本节试题精选

一、单项选择题

1. 树最适合用来表示 () 的数据。
A. 有序 B. 无序
C. 任意元素之间具有多种联系 D. 元素之间具有分支层次关系
2. 一棵有 n 个结点的树的所有结点的度数之和为 ()。
A. $n-1$ B. n C. $n+1$ D. $2n$
3. 树的路径长度是从树根到每一结点的路径长度的 ()。
A. 总和 B. 最小值 C. 最大值 D. 平均值
4. 对于一棵具有 n 个结点、度为 4 的树来说, ()。
A. 树的高度至多是 $n-3$ B. 树的高度至多是 $n-4$
C. 第 i 层上至多有 $4(i-1)$ 个结点 D. 至少在某一层上正好有 4 个结点
5. 度为 4、高度为 h 的树, 则 ()。
A. 至少有 $h+3$ 个结点 B. 至多有 $4h-1$ 个结点
C. 至多有 $4h$ 个结点 D. 至少有 $h+4$ 个结点
6. 假定一棵度为 3 的树中结点数为 50, 则其最小高度为 ()。
A. 3 B. 4 C. 5 D. 6
7. 【2010 年计算机联考真题】
在一棵度为 4 的树 T 中, 若有 20 个度为 4 的结点, 10 个度为 3 的结点, 1 个度为 2 的结点, 10 个度为 1 的结点, 则树 T 的叶结点个数是 ()。
A. 41 B. 82 C. 113 D. 122

① 推导公式: $S = m^{h-1} + m^{h-2} + m^{h-3} + \dots + m + 1 = (m^h - 1)/(m - 1)$

二、综合应用题

1. 含有 n 个结点的 3 叉树的最小高度是多少?
2. 已知一棵度为 4 的树中, 其度为 0、1、2、3 的结点数分别为 14、4、3、2, 求该树的结点总数 n 和度为 4 的结点个数, 并给出推导过程。
3. 已知一棵度为 m 的树中, 有 n_1 个度为 1 的结点, 有 n_2 个度为 2 的结点, ..., 有 n_m 个度为 m 的结点, 问该树有多少个叶子结点?

4.1.5 答案与解析

一、单项选择题

1. D

树是一种分层结构, 它特别适合组织那些具有分支层次关系的数据。

2. A

除根结点外, 其他每个结点都是某个结点的孩子结点, 因此树中所有结点的度数加 1 等于结点数, 所以所有结点的度数之和就等于总结点数减 1。这是个重要的结论, 做题时经常用到, 希望考生牢记。

3. A

树的路径长度是所有路径长度的总和, 树根到每一结点的路径的最大值应该是树的高度减 1。注意与 Huffman 树的带权路径长度相区别。

4. A

要使得具有 n 个结点、度为 4 的树的高度最大, 那么就要使得每一层的结点数尽可能的少, 类似第 4 题图所示的树, 除了最后一层外, 每一层的结点数是 1, 最终该树的高度为 $n-3$ 。树的度为 4 只能说明某一结点正好 (也是最多) 有 4 个子结点, D 错误。

5. A

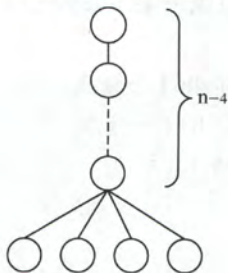
要使得度为 4、高度为 h 的树的总结点数最少, 那么要满足以下两个条件:

- ① 至少有一个结点有 4 个分支;
- ② 每一层的结点数目尽可能的少。

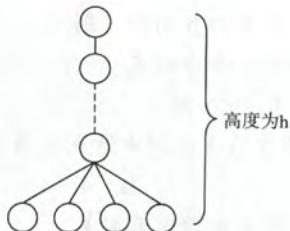
情况类似第 5 题图所示的树, 结点个数 $n = h + 3$ 。

要使得度为 4、高度为 h 的树的总结点数最多, 则应使每个非叶结点的度均为 4, 即为满树, 总结点个数最多为 $1 + 4 + 4^2 + \dots + 4^{h-1}$ 。

对于上面的两题, 应画出草图来求解, 一目了然。



第 4 题图



第 5 题图

6. C

要求满足条件的树, 那么该树是一棵完全三叉树。在度为 3 的完全三叉树中, 第 1 层有 1 个结点, 第 2 层有 $3^1=3$ 个结点, 第 3 层有 $3^2=9$ 个结点, 第 4 层有 $3^3=27$ 个结点, 这样结点和为

$1+3+9+27=40$ 个结点, 第 5 层结点数 $=50-40=10$ 个, 因此最小高度为 5。

7. B

设树中度为 i ($i=0, 1, 2, 3, 4$) 的结点数分别为 N_i , 树中结点总数为 N , 则 $N=$ 分支数 $+1$, 而分支数又等于树中各结点的度之和, 即 $N=1+N_1+2N_2+3N_3+4N_4=N_0+N_1+N_2+N_3+N_4$ 。依题意, $N_1+2N_2+3N_3+4N_4=10+2+30+80=122$, $N_1+N_2+N_3+N_4=10+1+10+20=41$, 可得出 $N_0=82$, 即树 T 的叶结点的个数是 82。

二、综合应用题

1. 解答:

要求含有 n 个结点的 3 叉树的最小高度, 那么满足条件的一定是一棵完全 3 叉树, 设含有 n 个结点的完全 3 叉树的高度为 h , 第 h 层至少有 1 个结点, 至多有 3^{h-1} 个结点。则有:

$$1+3^1+3^2+\cdots+3^{h-2}<n\leq 1+3^1+3^2+\cdots+3^{h-2}+3^{h-1}$$

$$\text{即 } (3^{h-1}-1)/2 < n \leq (3^h-1)/2$$

$$\text{得: } 3^{h-1} < 2n+1 \leq 3^h$$

$$\text{也即 } h < \log_3(2n+1)+1, h \geq \log_3(2n+1)$$

由于 h 只能为正整数, $h=\lceil \log_3(2n+1) \rceil$, 故这样的 3 次树最小高度是 $\lceil \log_3(2n+1) \rceil$ 。

2. 解答:

设树中度为 i ($i=0, 1, 2, 3, 4$) 的结点数为 N_i , 则结点总数 $N=N_0+N_1+N_2+N_3+N_4$, 即 $N=23+N_4$, 根据“树中所有结点的度数加 1 等于结点数”的结论, 有 $N=0+N_1+2N_2+3N_3+4N_4+1$, 即有 $N=17+4N_4$ 。

综合两式得: $N_4=2$, $N=25$ 。所以, 该树的结点总数为 25, 度为 4 的结点个数为 2。

3. 解答:

根据“树中所有结点的度数加 1 等于结点数”的结论, 有 $n=\sum_{i=0}^m i \times n_i = n_1+2 \times n_2+3 \times n_3+\cdots+m \times n_m+1$

$$\text{又有: } n=n_0+n_1+n_2+\cdots+n_m$$

$$\text{所以: } n_0=(n_1+2 \times n_2+3 \times n_3+\cdots+m \times n_m+1)-(n_1+n_2+\cdots+n_m)$$

$$=n_2+2 \times n_3+\cdots+(m-1) \times n_m+1=1+\sum_{i=2}^m (i-1)n_i$$

注意: 综合以上几题, 常用于求解树结点与度之间关系有:

$$\text{① 总结点数} = N_0+N_1+N_2+\cdots+N_m;$$

$$\text{② 总分支数} = 1 \times N_1+2 \times N_2+\cdots+m \times N_m \quad (\text{度为 } m \text{ 的结点引出 } m \text{ 条分支});$$

$$\text{③ 总结点数} = \text{总分支数} + 1。$$

这个性质经常在选择題中出现, 读者对于以上关系应当熟练掌握并灵活应用。

4.2 二叉树的概念

4.2.1 二叉树的定义及其主要特性

1. 二叉树的定义

二叉树是另一种树形结构, 其特点是每个结点至多只有两棵子树 (即二叉树中不存在度大于 2 的结点), 并且, 二叉树的子树有左右之分, 其次序不能任意颠倒。

与树相似, 二叉树也以递归的形式定义。二叉树是 n ($n \geq 0$) 个结点的有限集合:

① 或者为空二叉树, 即 $n=0$ 。

② 或者由一个根结点和两个互不相交的被称为根的左子树和右子树组成。左子树和右子树又分别是一棵二叉树。

二叉树是有序树，若将其左、右子树颠倒，就成为另一棵不同的二叉树。即使树中结点只有一棵子树，也要区分它是左子树还是右子树。二叉树的5种基本形态如图4-2所示。

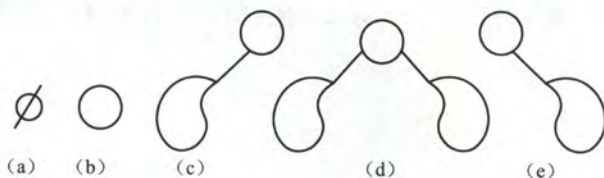


图4-2 二叉树的5种基本形态

二叉树与度为2的有序树的区别：

- ① 度为2的树至少有3个结点，而二叉树可以为空；
- ② 度为2的有序树的孩子结点的左右次序是相对于另一孩子结点而言的，如果某个结点只有一个孩子结点，这个孩子结点就无须区分其左右次序，而二叉树无论其孩子数是否为2，均需确定其左右次序，也就是说二叉树的结点次序不是相对于另一结点而言，而是确定的。

2. 几个特殊的二叉树

1) **满二叉树**：一棵高度为 h ，并且含有 $2^h - 1$ 个结点的二叉树称为满二叉树，即树中的每一层都含有最多的结点，如图4-3(a)所示。满二叉树的叶子结点都集中在二叉树的最下一层，并且除叶子结点之外的每个结点度数均为2。

可以对满二叉树按层序编号：约定编号从根结点（根结点编号为1）起，自上而下，自左向右。这样每个结点对应一个编号，对于编号为 i 的结点，如果有双亲，其双亲为 $\lfloor i/2 \rfloor$ ，如果有左孩子，则左孩子为 $2i$ ；如果有右孩子，则右孩子为 $2i+1$ 。

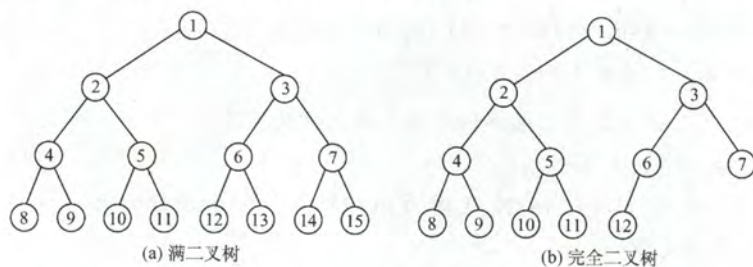


图4-3 两种特殊形态的二叉树

2) **完全二叉树**：设一个高度为 h ，有 n 个结点的二叉树，当且仅当其每一个结点都与高度为 h 的满二叉树中编号为 $1 \sim n$ 的结点一一对应时，称为完全二叉树，如图4-3(b)所示。这种树的特点如下：

- ① 若 $i \leq \lfloor n/2 \rfloor$ ，则结点 i 为分支结点，否则为叶子结点。
- ② 叶子结点只可能在层次最大的两层上出现。对于最大层次中的叶子结点，都依次排列在该层最左边的位置上。
- ③ 如果有度为1的结点，只可能有一个，且该结点只有左孩子而无右孩子（重要特征）。
- ④ 按层序编号后，一旦出现某结点（其编号为 i ）为叶子结点或只有左孩子，则编号大于 i 的结点均为叶子结点。
- ⑤ 若 n 为奇数，则每个分支结点都有左子女和右子女；若 n 为偶数，则编号最大的分支结

点(编号为 $n/2$)只有左子女,没有右子女,其余分支结点左右子女都有。

3) **二叉排序树**:一棵二叉树或者是空二叉树,或者是具有如下性质的二叉树:左子树上所有结点的关键字均小于根结点的关键字;右子树上的所有结点的关键字均大于根结点的关键字。左子树和右子树又各是一棵二叉排序树。

4) **平衡二叉树**:树上任一结点的左子树和右子树的深度之差不超过 1。

3. 二叉树的性质

1) 非空二叉树上叶子结点数等于度为 2 的结点数加 1, 即 $N_0=N_2+1$ 。

证明: 设度为 0、1 和 2 的结点个数分别为 N_0 、 N_1 和 N_2 , 结点总数 $N=N_0+N_1+N_2$ 。

再看二叉树中的分支数, 除根结点外, 其余结点都有一个分支进入, 设 B 为分支总数, 则 $N=B+1$ 。由于这些分支是由度为 1 或 2 的结点射出的, 所以又有 $B=N_1+2N_2$ 。

于是得: $N_0+N_1+N_2=N_1+2N_2+1$, 则 $N_0=N_2+1$ 。

注意: 该结论经常在选择題中用到, 希望考生牢记并灵活应用。

2) 非空二叉树上第 K 层上至多有 2^{K-1} 个结点 ($K \geq 1$)。

3) 高度为 H 的二叉树至多有 2^H-1 个结点 ($H \geq 1$)。

4) 对完全二叉树按从上到下、从左到右的顺序依次编号 1, 2, ..., N , 则有以下关系:

① 当 $i > 1$ 时, 结点 i 的双亲结点编号为 $\lfloor i/2 \rfloor$, 即当 i 为偶数时, 其双亲结点的编号为 $i/2$, 它是双亲结点的左孩子; 当 i 为奇数时, 其双亲结点的编号为 $(i-1)/2$, 它是双亲结点的右孩子。

② 当 $2i \leq N$ 时, 结点 i 的左孩子编号为 $2i$, 否则无左孩子。

③ 当 $2i+1 \leq N$ 时, 结点 i 的右孩子编号为 $2i+1$, 否则无右孩子。

④ 结点 i 所在层次(深度)为 $\lfloor \log_2 i \rfloor + 1$ 。

5) 具有 N 个 ($N > 0$) 结点的完全二叉树的高度为 $\lceil \log_2(N+1) \rceil$ 或 $\lfloor \log_2 N \rfloor + 1$ 。

4.2.2 二叉树的存储结构

1. 顺序存储结构

二叉树的顺序存储结构就是用一组地址连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素, 即将完全二叉树上编号为 i 的结点元素存储在某个数组下标为 $i-1$ 的分量中, 然后通过一些方法确定结点在逻辑上的父子和兄弟关系。

依据二叉树的性质, 完全二叉树和满二叉树采用顺序存储比较合适, 树中结点的序号可以唯一地反映出结点之间的逻辑关系, 这样既能最大可能地节省存储空间, 又可以利用数组元素的下标值确定结点在二叉树中的位置, 以及结点之间的关系。

但对于一般的二叉树, 为了让数组下标能反映二叉树中结点之间的逻辑关系, 只能添加一些并不存在的空结点让其每个结点与完全二叉树上的结点相对照, 再存储到一维数组的相应分量中。然而, 在最坏的情况下, 一个高度为 H 且只有 H 个结点的单支树却需要占据接近 2^H-1 个存储单元。二叉树的顺序存储结构如图 4-4 所示, 其中 0 表示并不存在的空结点。

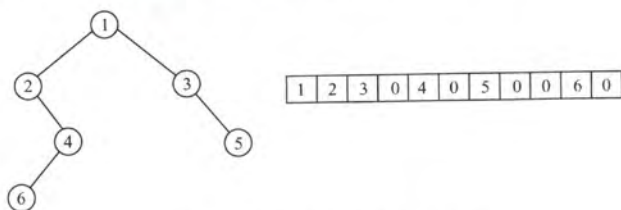


图 4-4 二叉树的顺序存储结构

注意：这种存储结构显然要从数组下标 1 开始存储树中的结点，如果从数组下标 0 开始存储，则不满足性质 4) 的描述，（比如结点 A 存储在 0 下标位置上时，则无法根据性质 4 来计算其孩子结点在数组中的位置），这是考生在书写程序的时候容易忽略的。

注意区别树的顺序存储结构与二叉树的顺序存储结构。在树的顺序存储结构中，数组下标代表结点的编号，下标上所存的内容指示了结点之间的关系。而在二叉树的顺序存储结构中，数组下标既代表了结点的编号，也指示了树中各结点之间的关系，这种关系借助完全二叉树的性质反应，当然，二叉树属于树，因此二叉树都可以用树的存储结构来存储，但是树却不都能用二叉树的存储结构来存储。

2. 链式存储结构

由于顺序存储对空间利用率较低，因此，一般二叉树都采用链式存储结构。链式结构是指用一个链表来存储一棵二叉树，二叉树中每一个结点用链表的一个链结点来存储。在二叉树中，结点结构通常包括若干数据域及若干个指针域。二叉链表至少包含 3 个域：数据域 **data**、左指针域 **lchild** 和右指针域 **rchild**，如图 4-5 所示。

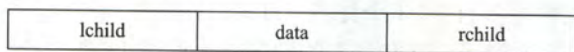


图 4-5 二叉树链式存储的结点结构

图 4-6 所示为常用的二叉链表的存储结构。而实际在不同的应用中，还可以增加某些指针域，如增加指向父结点的指针，则变为三叉链表的存储结构。

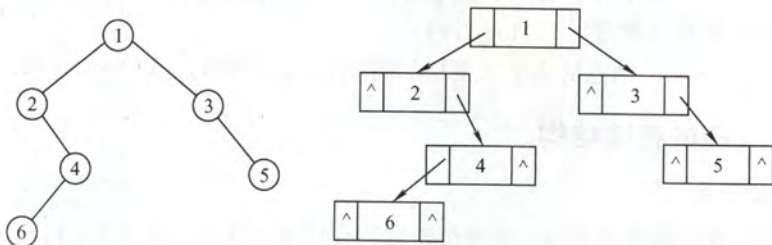


图 4-6 二叉链表的存储结构

二叉树的链式存储结构描述如下：

```
typedef struct BiTNode{
    ElemType data;                //数据域
    struct BiTNode *lchild,*rchild; //左、右孩子指针
}BiTNode,*BiTree;
```

使用不同的存储结构，实现二叉树操作的算法也会不同，因此，要根据实际应用的场合（二叉树的形态和需要进行的运算）来选择合适的存储结构。

容易验证，在含有 n 个结点的二叉链表中含有 $n+1$ 个空链域（重要结论，经常出现在选择题中）。在下一节中，我们将利用这些空链域来组成另一种链表结构——线索链表。

4.2.3 本节试题精选

一、单项选择题

1. 下列关于二叉树的说法中，正确的是（ ）。
 - A. 度为 2 的有序树就是二叉树
 - B. 含有 N 个结点的二叉树其高度为 $\lfloor \log_2 N \rfloor + 1$

- C. 在完全二叉树中, 若一个结点没有左孩子, 则它必是叶结点
 D. 在任意一棵非空二叉排序树中, 删除某结点后又将其插入, 则所得二叉排序树与删除前原二叉排序树相同
2. 以下说法中, 正确的是 ()。
- A. 在完全二叉树中, 叶子结点的双亲的左兄弟 (如果存在) 一定不是叶子结点
 B. 任何一棵二叉树, 叶子结点个数等于度为 2 的结点数减 1, 即 $N_0 = N_2 - 1$
 C. 完全二叉树不适合顺序存储结构, 只有满二叉树适合顺序存储结构
 D. 结点按完全二叉树层序编号的二叉树中, 第 i 个结点的左孩子的编号为 $2i$
3. 具有 10 个叶子结点的二叉树中有 () 个度为 2 的结点。
 A. 8 B. 9 C. 10 D. 11
4. 设高度为 h 的二叉树上只有度为 0 和度为 2 的结点, 则此类二叉树中所包含的结点数至少为 ()。
 A. h B. $2h-1$ C. $2h+1$ D. $h+1$
5. 假设一棵二叉树的结点个数为 50, 则它的最小高度是 ()。
 A. 4 B. 5 C. 6 D. 7
6. 设二叉树有 $2n$ 个结点, 且 $m < n$, 不可能存在 () 的结点。
 A. n 个度为 0 B. $2m$ 个度为 0 C. $2m$ 个度为 1 D. $2m$ 个度为 2
7. 一个具有 1025 个结点的二叉树的高 h 为 ()。
 A. 11 B. 10 C. $11 \sim 1025$ D. $10 \sim 1024$
8. 设二叉树只有度为 0 和 2 的结点, 其结点个数为 15, 则该二叉树的最大深度为 ()。
 A. 4 B. 5 C. 8 D. 9
9. 高度为 h 的完全二叉树最少有 () 个结点。
 A. 2^h B. 2^{h+1} C. 2^{h-1} D. $2^h - 1$
10. 已知一棵完全二叉树的第 6 层 (设根为第 1 层) 有 8 个叶结点, 则完全二叉树的结点数最少是 ()。
 A. 39 B. 52 C. 111 D. 119
11. 【2009 年计算机联考真题】
 已知一棵完全二叉树的第 6 层 (设根为第 1 层) 有 8 个叶结点, 则该完全二叉树的结点数最多是 ()。
 A. 39 B. 52 C. 111 D. 119
12. 若一棵深度为 6 的完全二叉树的第 6 层有 3 个叶子结点, 则该二叉树共有 () 个叶子结点。
 A. 17 B. 18 C. 19 D. 20
13. 一棵完全二叉树上有 1001 个结点, 其中叶结点的个数是 ()。
 A. 250 B. 500 C. 254 D. 501
14. 【2011 年计算机联考真题】
 若一棵完全二叉树有 768 个结点, 则该二叉树中叶结点的个数是 ()。
 A. 257 B. 258 C. 384 D. 385
15. 若一棵二叉树有 126 个结点, 在第 7 层 (根结点在第 1 层) 至多有 () 个结点。
 A. 32 B. 64 C. 63 D. 不存在第 7 层
16. 一棵有 124 个叶子结点的完全二叉树, 最多有 () 个结点。

- A. 247 B. 248 C. 249 D. 250
17. 一棵有 n 个结点的二叉树采用二叉链存储结构, 其中空指针数为 ()。
- A. n B. $n+1$ C. $n-1$ D. $2n$
18. 在一棵完全二叉树中, 其根的序号为 1, () 可判定序号为 p 和 q 的两个结点是否在同一层。
- A. $\lfloor \log_2 p \rfloor = \lfloor \log_2 q \rfloor$ B. $\log_2 p = \log_2 q$
 C. $\lfloor \log_2 p \rfloor + 1 = \lfloor \log_2 q \rfloor$ D. $\lfloor \log_2 p \rfloor = \lfloor \log_2 q \rfloor + 1$
19. 假定一棵三叉树的结点数为 50, 则它的最小高度为 ()。
- A. 3 B. 4 C. 5 D. 6
20. 已知一棵有 2011 个结点的树, 其叶结点个数是 116, 该树对应的二叉树中无右孩子的结点个数是 ()。
- A. 115 B. 116 C. 1895 D. 1896
21. 对于一棵满二叉树, 共有 n 个结点和 m 个叶子结点, 高度为 h , 则 ()。
- A. $n=h+m$ B. $h+m=2h$ C. $m=h-1$ D. $n=2^h-1$

二、综合应用题

- 在一棵完全二叉树中, 含有 N_0 个叶子结点, 当度为 1 的结点数为 1 时, 该树的高度是多少? 当度为 1 的结点数为 0 时, 该树的高度是多少?
- 一棵有 n 个结点的满二叉树有多少个分支结点和多少个叶子结点? 该满二叉树的高度是多少?
- 已知完全二叉树的第 9 层有 240 个结点, 则整个完全二叉树有多少个结点? 有多少个叶子结点?
- 一棵高度为 h 的满 m 叉树有如下性质: 根结点所在层次为第 1 层, 第 h 层上的结点都是叶结点, 其余各层上每个结点都有 m 棵非空子树, 如果按层次自顶向下, 同一层自左向右, 顺序从 1 开始对全部结点进行编号, 试问:
 - 各层的结点个数是多少?
 - 编号为 i 的结点的双亲结点 (若存在) 的编号是多少?
 - 编号为 i 的结点的第 k 个孩子结点 (若存在) 的编号是多少?
 - 编号为 i 的结点有右兄弟的条件是什么? 其右兄弟结点的编号是多少?
- 已知一棵二叉树按顺序存储结构进行存储, 设计一个算法, 求编号分别为 i 和 j 的两个结点的最近的公共祖先结点的值。

4.2.4 答案与解析

一、单项选择题

1. C

二叉树是有序树, 在二叉树中, 如果某个结点只有一个孩子结点, 这个孩子结点的左右次序是确定的; 而在度为 2 的有序树中, 如果某个结点只有一个孩子结点, 这个孩子结点就无须区分其左右次序。因此度为 2 的有序树不是二叉树, A 错误。选项 B 仅当是完全二叉树时才有意义, 对于任意一棵二叉树高度可能为 $\lfloor \log_2 N \rfloor + 1 \sim N$ 。在二叉排序树中插入结点时一定是插入在叶结点的位置, 故如果先删除分支结点再插入, 会导致二叉排序树的重构, 其结果就不再一样了, D 错误。根据完全二叉树的定义, 在完全二叉树中, 如果有度为 1 的结点, 只可能有一个, 且该结点

只有左孩子而无右孩子, 选项 C 正确。

2. A

在完全二叉树中, 叶子结点的双亲左兄弟的孩子一定在其前面 (且一定存在), 故双亲的左兄弟 (如果存在) 一定不是叶结点, A 正确。 N_0 应该等于 N_2+1 , B 错误。完全二叉树和满二叉树均可以采用顺序存储结构, C 错误。第 i 个结点的左孩子不一定存在, D 错误。

对于选项 B 的这种通用公式, 适用于所有的二叉树, 我们应立即联想到采用特殊值代入法验证, 如画一个只含 3 个结点的满二叉树的草图来验证是否满足条件。

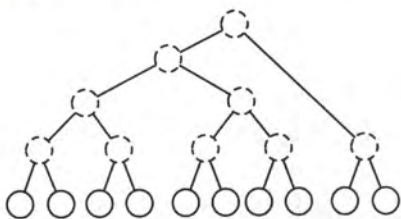
3. B

由二叉树的性质 $N_0=N_2+1$, 即 $N_2=N_0-1=10-1=9$ 。

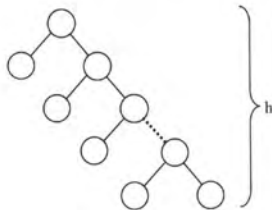
另解: 画出草图 (如第 3 题图)。首先画出 10 个叶结点, 然后每 2 个结点向上合并, 并构造一个新的度为 2 的分支结点, 直到构成如下图所示的二叉树, 其中度为 2 的分支结点数为 9。

4. B

结点最少的情况如第 4 题图所示。除根结点层只有 1 个结点外, 其他 $h-1$ 层均有两个结点, 结点总数 $=2(h-1)+1=2h-1$ 。



第 3 题图



第 4 题图

5. C

要求满足条件的树, 分析可知, 当这 50 个结点构成一棵完全二叉树时高度最小, $h=\lfloor \log_2 N \rfloor + 1 = \lfloor \log_2 50 \rfloor + 1 = 6$ 。

另解: 第 1 层最多有 1 个结点, 第 2 层最多有 2^1 个结点, 第 3 层最多有 2^2 个结点, 第 4 层最多有 2^3 个结点, 依次类推, 可以得到 h 最少为 6。

6. C

由二叉树性质 1 可知 $n_0=n_2+1$, 结点总数 $=2n=n_0+n_1+n_2=n_1+2n_2+1$, 则 $n_1=2(n-n_2)-1$, 所以 n_1 为奇数, 说明该二叉树中不可能有 $2m$ 个度为 1 的结点。

7. C

当二叉树为单支树时具有最大高度, 即每层上只有一个结点, 最大高度为 1025。而当树为完全二叉树时, 其高度最小, 最小高度为 $\lfloor \log_2 N \rfloor + 1 = 11$ 。

8. C

解题思路同第 4 题, 第一层有一个结点, 其余 $h-1$ 层上各有两个结点, 总结点数 $=1+2(h-1)=15$, $h=8$ 。建议画出草图。

9. C

高度为 h 的完全二叉树中, 第 1 层~第 $h-1$ 层构成一个高度为 $h-1$ 的满二叉树, 结点个数为 $2^{h-1}-1$ 。第 h 层至少有一个结点, 所以最少的结点个数 $=(2^{h-1}-1)+1=2^{h-1}$ 。

10. A

第 6 层有叶结点则说明完全二叉树的高度可能为 6 或 7, 显然树高为 6 时结点最少。若第 6 层上有 8 个叶结点, 则前 5 层为满二叉树, 故完全二叉树的结点个数最少为 $2^5-1+8=39$ 个结点。

11. C

第6层有叶结点则完全二叉树的高度可能为6或7,显然树高为7时结点最多。完全二叉树相比满二叉树只是在最下一层的右边缺少了部分叶结点,而最后一层之上是个满二叉树,并且只有最后两层上有叶结点。若第6层上有8个叶结点,则前6层为满二叉树,而第7层缺失了 $8 \times 2 = 16$ 个叶结点,故完全二叉树的结点个数最多为 $2^7 - 1 - 16 = 111$ 个结点。

12. A

深度为6的完全二叉树,第5层共有结点 $2^4 = 16$ 个。第6层最左边有3个叶子结点,其对应的双亲结点为第5层最左边两个结点,所以,第5层剩余的结点均为叶子结点,共有 $16 - 2 = 14$ 个,加上第6层3个,共17个叶子结点。

13. D

由完全二叉树的性质,最后一个分支结点的序号为 $\lfloor 1001/2 \rfloor = 500$,故叶子结点个数为501。

另解: $n = n_0 + n_1 + n_2 = n_0 + n_1 + (n_0 - 1) = 2n_0 + n_1 - 1$,因为 $n = 1001$,而在完全二叉树中, n_1 只能取0或者1。当 $n_1 = 1$ 时, n_0 为小数,不符合题意。所以 $n_1 = 0$,故 $n_0 = 501$ 。

14. C

求解过程与第13题类似。由完全二叉树的性质,最后一个分支结点的序号为 $\lfloor 768/2 \rfloor = 384$,故叶子结点的个数为 $768 - 384 = 384$ 。

另解: $n = n_0 + n_1 + n_2 = n_0 + n_1 + (n_0 - 1) = 2n_0 + n_1 - 1$,其中 $n = 768$,而在完全二叉树中, n_1 只能取0或者1,当 $n_1 = 0$ 时, n 为小数,不符合题意。所以 $n_1 = 1$,故 $n_0 = 384$ 。

15. C

要使二叉树在第7层达到最多的结点个数,其上面的6层必须是一个满二叉树,深度为6的满二叉树有 $63 (2^6 - 1)$ 个结点,故第7层最多有 $126 - 63 = 63$ 个结点。

16. B

在非空的二叉树当中,由度为0和2的结点数的关系 $N_0 = N_2 + 1$ 可知 $N_2 = 123$;总结点数 $N = N_0 + N_1 + N_2 = 247 + N_1$,其最大值为248(N_1 的取值为1或0,当 $N_1 = 1$ 时结点最多)。注意,由完全二叉树总结点数的奇偶性可以确定 N_1 的值,但不能根据 N_0 来确定的 N_1 值。

另解: $124 < 2^7 - 128$,故第8层没满,前7层为完全二叉树,由此可推算第8层有120个叶子结点,第7层的最右4个为叶子结点,而考虑最多的情况,这4个叶子结点中的最左边的可以有一个左孩子(不改变叶子结点数),因此,结点总数 $= 2^7 - 1 + 120 + 1 = 248$ 。

17. B

非空指针数=总分支数 $= n - 1$,空指针数 $= 2 \times$ 结点总数 $-$ 非空指针数 $= 2n - (n - 1) = n + 1$ 。

另解:在树中,1个指针对应1个分支, n 个结点的树共有 $n - 1$ 个分支,即 $n - 1$ 个非空指针,每个结点有2个指针域,故空指针数 $= 2n - (n - 1) = n + 1$ 。

18. A

由完全二叉树的性质,编号为 $i (i \geq 1)$ 的结点所在层次为 $\lfloor \log_2 i \rfloor + 1$,如果两个结点位于同一层,一定是 $\lfloor \log_2 p \rfloor + 1 = \lfloor \log_2 q \rfloor + 1$,因此有 $\lfloor \log_2 p \rfloor = \lfloor \log_2 q \rfloor$ 成立。

19. C

分析可知,满足条件的三叉树是完全三叉树,该树的第 $i (i \geq 1)$ 层最多有 3^{i-1} 个结点。设高度为 h ,则 $3^0 + 3^1 + \dots + 3^{h-1} = (3^h - 1)/2$ 是结点数上限,问题是求解 $50 \leq (3^h - 1)/2$ 的最小 h 值,即 $h \geq \log_3 101$,即 $h = \lceil \log_3 101 \rceil = 5$ 。

20. D

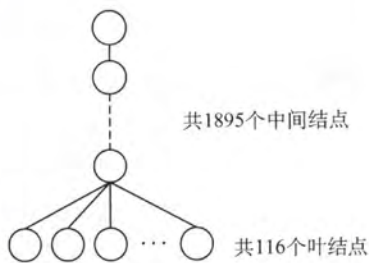
可以采用特殊情况法求解。可举如下特例:

如图,则对应的二叉树中仅有前 115 个结点有右孩子结点,其余 1896 个结点均无右孩子结点。

21. D

对于高度为 h 的满二叉树, $n=2^0+2^1+\cdots+2^{h-1}=2^h-1$, $m=2^{h-1}$, 故选 D。

另解:对于本题,也可以画出特殊的图形来检验选项。例如对于高度为 3 的满二叉树, $n=7$, $m=4$, $h=3$, 经检验,只有 D 正确。



第 20 题图

二、综合应用题

1. 解答:

在非空的二叉树中,由度为 0 和度为 2 的结点之间的关系 $N_0=N_2+1$, 可知 $N_2=N_0-1$ 。

则总结点数 $N=N_0+N_1+N_2=2N_0+N_1-1$ 。

①当 $N_1=1$ 时, $N=2N_0$, $h=\lceil \log_2(N+1) \rceil = \lceil \log_2(2N_0+1) \rceil$

②当 $N_1=0$ 时, $N=2N_0-1$, $h=\lceil \log_2(N+1) \rceil = \lceil \log_2(2N_0) \rceil = \lceil \log_2(N_0) \rceil + 1$

2. 解答:

满二叉树中 $n_1=0$, 由二叉树的性质 1 可知, $n_0=n_2+1$, 即 $n_2=n_0-1$, $n=n_0+n_1+n_2=2n_0-1$, 则 $n_0=(n+1)/2$ 。分支结点个数 $n_2=n-(n+1)/2=(n-1)/2$ 。高度为 h 的满二叉树的结点个数 $n=1+2^1+2^2+\cdots+2^{h-1}=2^h-1$, 即高度 $h=\log_2(n+1)$ 。

3. 解答:

在完全二叉树中,若第 9 层是满的,则结点数 $=2^{9-1}=256$, 而现在第 9 层只有 240 个结点,说明第 9 层未滿,是最后一层。其 1~8 层是满的,所以总的结点数 $=2^8-1+240=495$ 。

因为第 9 层是最后一层,所以第 9 层的结点都是叶子结点。且第 9 层的 240 个结点的双亲在第 8 层中,其双亲个数为 120,即第 8 层有 120 个分支结点,其余为叶子结点,所以第 8 层的叶子结点个数为 $2^{8-1}-120=8$ 。因此,总的叶子结点个数 $=8+240=248$ 个。

另解:总结点数 $N=N_0+N_1+N_2$, $N_2=N_0-1$, 则 $N=N_0+N_1+N_2=2N_0+N_1-1$ 。若 $N_1=1$, 则 $2N_0+N_1-1=2N_0=495$, 不符合;若 $N_1=0$, 则 $2N_0+N_1-1=2N_0-1=495$, 则 $N_0=248$ 。注意:对于本题,应理解完全二叉树中只有最底层结点是不满的,其他各层的结点是满的(即第 i 层有 2^{i-1} 个结点)。

4. 解答:

1) 第 1 层有 $m^0=1$ 个结点,第 2 层有 m^1 个结点,第 3 层有 m^2 个结点……一般地,第 i 层有 m^{i-1} 个结点($1 \leq i \leq h$)。

2) 在 m 叉树的情形下,结点 i 的第 1 个子女编号为 $j=(i-1) \times m+2$,反过来,结点 i 的双亲的编号是 $\lfloor (i-2)/m \rfloor + 1$,根结点没有双亲,所以要求 $i > 1$ 。

3) 因为结点 i 的第 1 个子女编号为 $(i-1) \times m+2$,若设该结点子女的序号为 $k=1,2,\cdots,m$,则第 k 个子女结点的编号为 $(i-1) \times m+k+1$ ($1 \leq k \leq m$)。

4) 当结点 i 不是其双亲的第 m 个子女时才有右兄弟。若设其双亲编号为 j ,可得 $j=\lfloor (i+m-2)/m \rfloor$,结点 j 的第 m 个子女的编号为 $(j-1) \times m+m+1=j \times m+1=\lfloor (i+m-2)/m \rfloor \times m+1$,所以当结点的编号 $i \leq \lfloor (i+m-2)/m \rfloor \times m$ 时才有右兄弟,右兄弟的编号为 $i+1$ 。或者,对于任一双亲结点 j ,其第 m 个子女结点的编号是 $j \times m+1$,故若不为第 m 的子女结点则 $(i-1) \% m \neq 0$ 。

5. 解答:

首先,必须明确二叉树中任意两个结点必然存在最近的公共祖先结点,最坏的情况下是根结点(两个结点分别在根结点的左右分支中),而且从最近的公共祖先结点到根结点的全部祖先结点都是公共的。由二叉树顺序存储的性质可知,任一结点 i 的双亲结点的编号为 $i/2$ 。求解 i 和 j 最近公共祖先结点的算法步骤如下(设从数组下标 1 开始存储):

1) 若 $i > j$, 则结点 i 所在层次大于或等于结点 j 所在层次。结点 i 的双亲结点为结点 $i/2$, 若 $i/2 = j$, 则结点 $i/2$ 是原结点 i 和结点 j 的最近公共祖先结点, 若 $i/2 \neq j$, 则令 $i = i/2$, 即以该结点 i 的双亲结点为起点, 采用递归的方法继续查找。

2) 若 $j > i$, 则结点 j 所在层次大于或等于结点 i 所在层次。结点 j 的双亲结点为结点 $j/2$, 若 $j/2 = i$, 则结点 $j/2$ 是原结点 i 和结点 j 的最近公共祖先结点, 若 $j/2 \neq i$, 则令 $j = j/2$ 。

重复上述过程, 直到找到它们最近的公共祖先结点为止。

本题代码如下:

```
ElemType Comm_Ancestor(SqTree T, int i, int j){
    //本算法在二叉树中查找结点 i 和结点 j 的最近公共祖先结点
    if (T[i] != '#' && T[j] != '#'){           //结点存在
        while (i != j){                          //两个编号不同时循环
            if (i > j)
                i = i/2;                          //向上找 i 的祖先
            else
                j = j/2;                          //向上找 j 的祖先
        }
        return T[i];
    }
}
```

由解题中算法的步骤描述可知, 本题也可以很容易地联想到采用递归的方法求解。

4.3 二叉树的遍历和线索二叉树

4.3.1 二叉树的遍历

所谓二叉树的遍历, 是指按某条搜索路径访问树中的每个结点, 使得每个结点均被访问一次, 而且仅被访问一次。

由二叉树的递归定义可知, 遍历一棵二叉树便要决定对根结点 N 、左子树 L 和右子树 R 的访问顺序。按照先遍历左子树再遍历右子树的原则, 常见的遍历次序有先序 (NLR)、中序 (LNR) 和后序 (LRN) 三种遍历算法。其中, 序指的是根结点在何时被访问。

1. 先序遍历 (PreOrder)

先序遍历的操作过程为:

如果二叉树为空, 什么也不做。否则:

- 1) 访问根结点;
- 2) 先序遍历左子树;
- 3) 先序遍历右子树。

对应的递归算法如下:

```
void PreOrder(BiTree T){
    if (T != NULL){
```

```

        visit(T);                //访问根结点
        PreOrder(T->lchild);      //递归遍历左子树
        PreOrder(T->rchild);      //递归遍历右子树
    }
}

```

对于图 4-4 所示的二叉树，先序遍历所得到的结点序列为：1 2 4 6 3 5。

2. 中序遍历 (InOrder)

中序遍历的操作过程为：

如果二叉树为空，什么也不做。否则：

- 1) 中序遍历左子树；
- 2) 访问根结点；
- 3) 中序遍历右子树。

对应的递归算法如下：

```

void InOrder(BiTree T){
    if(T!=NULL){
        InOrder(T->lchild);    //递归遍历左子树
        visit(T);              //访问根结点
        InOrder(T->rchild);    //递归遍历右子树
    }
}

```

对于图 4-4 所示的二叉树，中序遍历所得到的结点序列为：2 6 4 1 3 5。

3. 后序遍历 (PostOrder)

后序遍历的操作过程为：

如果二叉树为空，什么也不做。否则：

- 1) 后序遍历左子树；
- 2) 后序遍历右子树；
- 3) 访问根结点。

对应的递归算法如下：

```

void PostOrder(BiTree T){
    if(T!=NULL){
        PostOrder(T->lchild);    //递归遍历左子树
        PostOrder(T->rchild);    //递归遍历右子树
        visit(T);               //访问根结点
    }
}

```

对于图 4-4 所示的二叉树，后序遍历所得到的结点序列为：6 4 2 5 3 1。

三种遍历算法中递归遍历左、右子树的顺序都是固定的，只是访问根结点的顺序不同。不管采用哪种遍历算法，每个结点都访问一次且仅访问一次，故时间复杂度都是 $O(n)$ 。在递归遍历中，递归工作栈的栈深恰好为树的深度，所以在最坏的情况下，二叉树是有 n 个结点且深度为 n 的单支树，遍历算法的空间复杂度为 $O(n)$ 。

注意：以上三种遍历方式以及算法描述是简单易懂的，读者需要将它们作为模板来记忆，考研中的很多题目都是基于这 3 个模板而延伸出来的。

4. 递归算法和非递归算法的转换

可以借助栈，将二叉树的递归遍历算法转换为非递归算法，下面以中序遍历为例给出中序遍历的非递归算法。先扫描（并非访问）根结点的所有左结点并将它们一一进栈。然后出栈一个结点*p（显然结点*p没有左孩子结点或者左孩子结点均已访问过），则访问它。然后扫描该结点的右孩子结点，将其进栈，再扫描该右孩子结点的所有左结点并一一进栈，如此继续，直到栈空为止。

中序遍历的非递归算法如下：

```
void InOrder2(BiTree T){
//二叉树中序遍历的非递归算法，算法需要借助一个栈
InitStack(S); BiTree p=T;           //初始化栈；p是遍历指针
while(p||!IsEmpty(S)){               //栈不空或p不空时循环
    if(p){                             //根指针进栈，遍历左子树
        Push(S,p);                     //每遇到非空二叉树先向左走
        p=p->lchild;
    }
    else{                               //根指针退栈，访问根结点，遍历右子树
        Pop(S,p);visit(p);             //退栈，访问根结点
        p=p->rchild;                   //再向右子树走
    }
}
```

显然非递归算法的执行效率要高于递归算法。类似地可以得到先序遍历与后序遍历的非递归算法，其中后序遍历的非递归算法较复杂，留给读者思考。

5. 层次遍历

如图4-7所示为二叉树的层次遍历，即按照箭头所指方向，按照1、2、3、4的层次顺序，对二叉树中各个结点进行访问。

要进行层次遍历需要借助一个队列。先将二叉树根结点入队，然后出队，访问该结点，如果它有左子树，则将左子树根结点入队；如果它有右子树，则将右子树根结点入队。然后出队，对出队结点访问，如此反复，直到队列为空。

二叉树的层次遍历算法如下：

```
void LevelOrder(BiTree T){
    InitQueue(Q);                       //初始化辅助队列
    BiTree p;
    EnQueue(Q,T);                       //将根结点入队
    while(!IsEmpty(Q)){                 //队列不空循环
        DeQueue(Q,p);                   //队头元素出队
        visit(p);                       //访问当前p所指向结点
        if(p->lchild!=NULL)              //左子树不空，则左子树入队列
            EnQueue(Q,p->lchild);
        if(p->rchild!=NULL)              //右子树不空，则右子树入队列
            EnQueue(Q,p->rchild);
    }
}
```

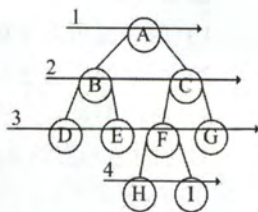


图 4-7 二叉树的层次遍历

对于上述二叉树层次遍历的算法,读者在复习过程当中应该将其作为一个模板,在熟练掌握其执行过程的基础上来记忆,并能够达到熟练默写的程度。这样才能将层次遍历模板应用于各种题目之中。

注意:遍历是二叉树各种操作的基础,可以在遍历的过程中对结点进行各种操作,例如,对于一棵已知树求结点的双亲、求结点的孩子结点、求二叉树的深度、求二叉树的叶子结点个数、判断两棵二叉树是否相等等。所有这些操作都建立在二叉树遍历的基础上,因此,必须掌握二叉树的各种遍历过程,并能灵活运用以解决各种问题。

6. 由遍历序列构造二叉树

由二叉树的先序序列和中序序列可以唯一地确定一棵二叉树,在先序遍历序列中,第一个结点一定是二叉树的根结点,而在中序遍历中,根结点必然将中序序列分割成两个子序列,前一个子序列就是根结点的左子树的中序序列,后一个子序列是根结点的右子树的中序序列。根据这两个子序列,在先序序列中找到对应的左子序列和右子序列。在先序序列中,左子序列的第一个结点是左子树的根结点,右子序列的第一个结点是右子树的根结点。如此递归地进行下去,便能唯一地确定这棵二叉树。

同理,由二叉树的后序序列和中序序列也可以唯一地确定一棵二叉树,因为后序序列的最后一个结点就如同先序序列的第一个结点,可以将中序序列分割成两个子序列,然后采用类似的方法递归地进行划分,就可以得到一棵二叉树。

由二叉树的层序序列和中序序列也可以唯一地确定一棵二叉树,实现方法留给读者思考。需要注意的是,如果只知道二叉树的先序序列和后序序列,则无法唯一确定一棵二叉树。

例如,求先序序列(ABCDEFGHI)和中序序列(BCAEDGHFI)所确定的二叉树。

首先,由先序序列可知A为二叉树的根结点。中序序列中A之前的BC为左子树的中序序列,EDGHFI为右子树的中序序列。然后由先序序列可知B是左子树的根结点,D是右子树的根结点。依此类推,就能将剩下的结点继续分解下去,最后得到的二叉树如图4-8(c)。

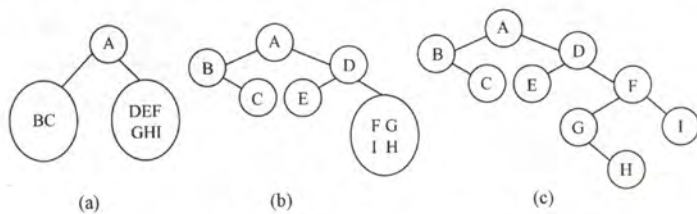


图4-8 一棵二叉树的构造过程

4.3.2 线索二叉树

1. 线索二叉树的基本概念

从上一节的介绍可知,遍历二叉树就是以一定的规则将二叉树中的结点排列成一个线性序列,从而得到二叉树结点的各种遍历序列。其实质就是对一个非线性结构进行线性化操作,使在这个访问序列中每一个结点(除第一个和最后一个)都有一个直接前驱和直接后继。

传统的链式存储仅能体现一种父子关系,不能直接得到结点在遍历中的前驱或后继。通过观察,我们发现在二叉链表表示的二叉树中存在大量的空指针,若利用这些空链域存放指向其直接前驱或后继的指针,则可以更方便地运用某些二叉树操作算法。引入线索二叉树是为了加快查找结点前驱和后继的速度。

前面提到,在有 N 个结点的二叉树中,有 $N+1$ 个空指针。这是因为每一个叶结点有 2 个空指针,而每一个度为 1 的结点有 1 个空指针,总的空指针数为 $2N_0+N_1$,又有 $N_0=N_2+1$,所以,总的空指针为 $N_0+N_1+N_2+1=N+1$ 。

在二叉树线索化时,通常规定:若无左子树,令 $lchild$ 指向其前驱结点;若无右子树,令 $rchild$ 指向其后继结点。如图 4-9 所示,还需要增加两个标志域表明当前指针域所指对象是指向左(右)子结点还是直接前驱(后继)。

ltag	lchild	data	rchild	rtag
------	--------	------	--------	------

图 4-9 线索二叉树的结点结构

其中标志域含义如下:

$ltag \begin{cases} 0 & lchild \text{域指示结点的左孩子} \\ 1 & lchild \text{域指示结点的前驱} \end{cases}$
 $rtag \begin{cases} 0 & rchild \text{域指示结点的右孩子} \\ 1 & rchild \text{域指示结点的后继} \end{cases}$

线索二叉树的存储结构描述如下:

```
typedef struct ThreadNode{
    ElemType data;           //数据元素
    struct ThreadNode *lchild,*rchild; //左、右孩子指针
    int ltag,rtag;           //左、右线索标志
}ThreadNode,*ThreadTree;
```

以这种结点结构构成的二叉链表作为二叉树的存储结构,叫做**线索链表**,其中指向结点前驱和后继的指针,叫做**线索**。加上线索的二叉树称为**线索二叉树**。对二叉树以某种次序遍历使其变为线索二叉树的过程叫做**线索化**。

2. 线索二叉树的构造

对二叉树的线索化,实质上就是遍历一次二叉树,只是在遍历的过程中,检查当前结点左、右指针域是否为空,若为空,将它们改为指向前驱结点或后继结点的线索。

以中序线索二叉树的建立为例,指针 pre 指向中序遍历时上一个刚刚访问过的结点,用它来表示各结点访问的前后关系,如图 4-10 所示。

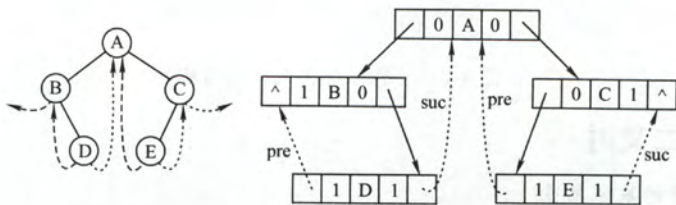


图 4-10 中序线索二叉树及其二叉链表表示

通过中序遍历对二叉树线索化的递归算法如下:

```
void InThread(ThreadTree &p, ThreadTree &pre){
    //中序遍历对二叉树线索化的递归算法
    if(p!=NULL){
        InThread(p->lchild, pre); //递归, 线索化左子树
        if(p->lchild==NULL){      //左子树为空, 建立前驱线索
            p->lchild=pre;
        }
    }
```

```

        p->ltag=1;
    }
    if(pre!=NULL&&pre->rchild==NULL){
        pre->rchild=p;           //建立前驱结点的后继线索
        pre->rtag=1;
    }
    pre=p;                       //标记当前结点成为刚刚访问过的结点
    InThread(p->rchild,pre);      //递归,线索化右子树
} //if(p!=NULL)
}

```

通过中序遍历建立中序线索二叉树的主过程算法如下:

```

void CreateInThread(ThreadTree T){
    ThreadTree pre=NULL;
    if(T!=NULL){                //非空二叉树,线索化
        InThread(T,pre);        //线索化二叉树
        pre->rchild=NULL;       //处理遍历的最后一个结点
        pre->rtag=1;
    }
}

```

有时为了方便,仿照线性表的存储结构,在二叉树的线索链表上也添加一个头结点(见图4-11),并令其 lchild 域的指针指向二叉树的根结点,其 rchild 域的指针指向中序遍历时访问的最后一个结点;反之,令二叉树中序序列中的第一个结点的 lchild 域的指针和最后一个结点 rchild 域的指针均指向头结点。这好比为二叉树建立了一个双向线索链表,既可以从第一个结点起顺后继进行遍历,也可从最后一个结点起顺前驱进行遍历。

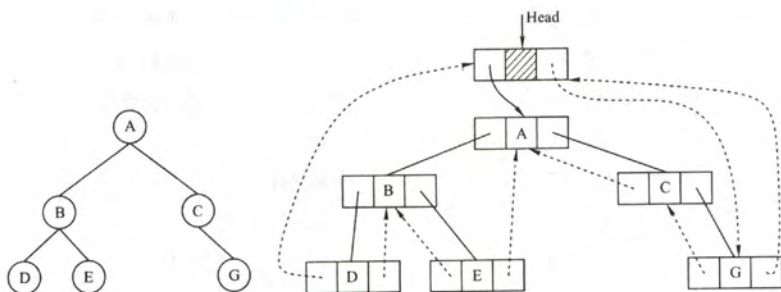


图 4-11 带头结点的中序线索二叉树

3. 线索二叉树的遍历

中序线索化二叉树主要是为访问运算服务的,这种遍历不再需要借助栈,因为它的结点中隐含了线索二叉树的前驱和后继信息。利用线索二叉树,可以实现二叉树遍历的非递归算法。不含头结点的线索二叉树的遍历算法如下:

1) 求中序线索二叉树中中序序列下的第一个结点:

```

ThreadNode *Firstnode(ThreadNode *p){
    while(p->ltag==0) p=p->lchild; //最左下结点(不一定是叶结点)
    return p;
}

```

2) 求中序线索二叉树中结点 p 在中序序列下的后继结点:

```

ThreadNode *Nextnode(ThreadNode *p){
    if(p->rtag==0) return Firstnode(p->rchild);
    else return p->rchild; //rtag==1 直接返回后继线索
}

```


请读者自行分析并完成求中序线索二叉树的最后一个结点和结点 p 前驱结点的运算。

3) 利用上面两个算法, 可以写出不含头结点的中序线索二叉树的中序遍历的算法:

```
void Inorder(ThreadNode *T){
    for(ThreadNode *p=Firstnode(T);p!=NULL; p=Nextnode(p))
        visit(p);
}
```

4.3.3 本节试题精选

一、单项选择题

- 在下列关于二叉树遍历的说法中, 正确的是 ()。
 - 若有一个结点是二叉树中某个子树的中序遍历结果序列的最后一个结点, 则它一定是该子树的前序遍历结果序列的最后一个结点
 - 若有一个结点是二叉树中某个子树的前序遍历结果序列的最后一个结点, 则它一定是该子树的中序遍历结果序列的最后一个结点
 - 若有一个叶子结点是二叉树中某个子树的中序遍历结果序列的最后一个结点, 则它一定是该子树的前序遍历结果序列的最后一个结点
 - 若有一个叶子结点是二叉树中某个子树的前序遍历结果序列的最后一个结点, 则它一定是该子树的中序遍历结果序列的最后一个结点
- 在任何一棵二叉树中, 如果结点 a 有左孩子 b、右孩子 c, 则在结点的先序序列、中序序列、后序序列中, ()。
 - 结点 b 一定在结点 a 的前面
 - 结点 a 一定在结点 c 的前面
 - 结点 b 一定在结点 c 的前面
 - 结点 a 一定在结点 b 的前面
- 设 n、m 为一棵二叉树上的两个结点, 在中序遍历时, n 在 m 前的条件是 ()。
 - n 在 m 右方
 - n 是 m 祖先
 - n 在 m 左方
 - n 是 m 子孙
- 设 n、m 为一棵二叉树上的两个结点, 在后序遍历时, n 在 m 前的条件是 ()。
 - n 在 m 右方
 - n 是 m 祖先
 - n 在 m 左方
 - n 是 m 子孙
- 在二叉树中有两个结点 m 和 n, 如果 m 是 n 的祖先, 使用 () 可以找到从 m 到 n 的路径。
 - 先序遍历
 - 中序遍历
 - 后序遍历
 - 层次遍历
- 在二叉树的前序序列、中序序列和后序序列中, 所有叶子结点的先后顺序 ()。
 - 都不相同
 - 完全相同
 - 前序和中序相同, 而与后序不同
 - 中序和后序相同, 而与前序不同
- 对二叉树的结点从 1 开始进行连续编号, 要求每个结点的编号大于其左、右孩子的编号, 同一结点的左、右孩子中, 其左孩子的编号小于其右孩子的编号, 可采用 () 次序的遍历实现编号。
 - 先序遍历
 - 中序遍历
 - 后序遍历
 - 层次遍历
- 前序为 A、B、C, 后序为 C、B、A 的二叉树共有 ()。
 - 1 棵
 - 2 棵
 - 3 棵
 - 4 棵
- 一棵非空的二叉树的先序遍历序列与后序遍历序列正好相反, 则该二叉树一定满足 ()。
 - 所有的结点均无左孩子
 - 所有的结点均无右孩子

- C. 只有一个叶结点 D. 是任意一棵二叉树

10. 设结点 X 和 Y 是二叉树中任意的两个结点。在该二叉树的先序遍历序列中 X 在 Y 之前, 而在其后序遍历序列中 X 在 Y 之后, 则 X 和 Y 的关系是 ()。

- A. X是Y的左兄弟 B. X是Y的右兄弟
C. X是Y的祖先 D. X是Y的后裔

11. 如果二叉树中结点的先序序列是...a...b..., 中序序列是...b...a..., 则 ()。

- A. 结点 a 和结点 b 分别在某结点的左子树和右子树中
B. 结点 b 在结点 a 的右子树中
C. 结点 b 在结点 a 的左子树中
D. 结点 a 和结点 b 分别在某结点的两棵非空子树中

12. 【2009 年计算机联考真题】

给定二叉树如图 4-12 所示。设 N 代表二叉树的根，L 代表根结点的左子树，R 代表根结点的右子树。若遍历后的结点序列是 3175624，则其遍历方式是 ()。

- A. LRN
B. NRL
C. RLN
D. RNL

13. 一棵二叉树的前序遍历序列为 1234567, 它的中序遍历序列可能是)。

- A. 3124567 B. 1234567 C. 4135627 D. 1463572

14. 【2011 年计算机联考真题】

若一棵二叉树的前序遍历序列和后序遍历序列分别为 1, 2, 3, 4 和 4, 3, 2, 1, 则该二叉树的中序遍历序列不会是 ()。

- A. 1, 2, 3, 4 B. 2, 3, 4, 1 C. 3, 2, 4, 1 D. 4, 3, 2, 1

15. 【2012 年计算机联考真题】

若一棵二叉树的前序遍历序列为 a, e, b, d, c, 后序遍历序列为 b, c, d, e, a, 则根结点的孩子结点 ()。

- A. 只有 e B. 有 e、b C. 有 e、c D. 无法确定

16. 【2017 年计算机联考真题】

某二叉树的树形如下图所示,其后序序列为 e,a,c,b,d,g,f, 树中与结点 a 同层的结点是()。

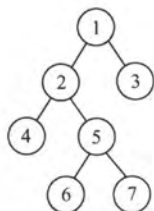
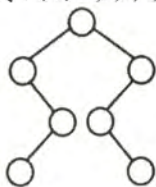


图 4-12 二叉树



- A. c B. d C. f D. g

17. 下列序列中, 不能唯一地确定一棵二叉树的是 ()。

- A. 层次序列和中序序列
B. 先序序列和中序序列
C. 后序序列和中序序列
D. 先序序列和后序序列

18. 已知一棵二叉树的后序序列为 DABEC, 中序序列为 DEBAC, 则先序序列为 ()。

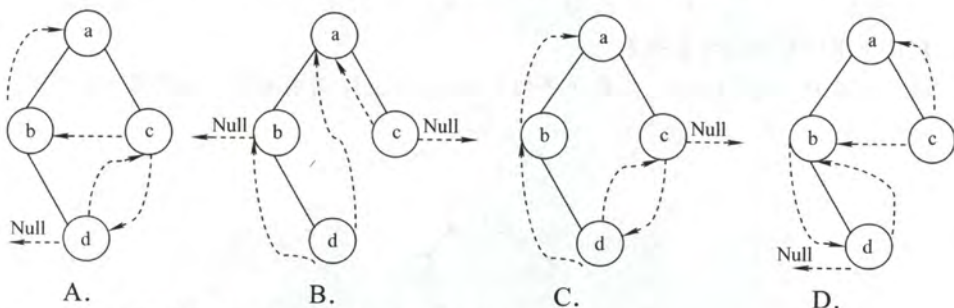
- A. ACBED B. DECAB C. DEABC D. CEDBA

19. 已知一棵二叉树的先序遍历结果为 ABCDEF, 中序遍历结果为 CBAEDF, 则后序遍历

的结果为 ()。

- A. CBEFDA B. FEDCBA C. CBEDFA D. 不确定
20. 已知一棵二叉树的层次序列为 ABCDEF, 中序序列为 BADCFE, 则先序序列为 ()。
- A. ACBEDF B. ABCDEF C. BDFECA D. FCEDBA
21. 【2017 年计算机联考真题】
- 要使一棵非空二叉树的先序序列与中序序列相同, 其所有非叶结点须满足的条件是 ()。
- A. 只有左子树 B. 只有右子树 C. 结点的度均为 1 D. 结点的度均为 2
22. 引入线索二叉树的目的是 ()。
- A. 加快查找结点的前驱或后继的速度 B. 为了能在二叉树中方便插入和删除
- C. 为了能方便找到双亲 D. 使二叉树的遍历结果唯一
23. 线索二叉树是一种 () 结构。
- A. 逻辑 B. 逻辑和存储 C. 物理 D. 线性
24. n 个结点的线索二叉树上含有的线索数为 ()。
- A. $2n$ B. $n-1$ C. $n+1$ D. n
25. 判断线索二叉树中 $*p$ 结点有右孩子结点的条件是 ()。
- A. $p!=\text{NULL}$ B. $p->\text{rchild}!=\text{NULL}$
- C. $p->\text{rtag}=0$ D. $p->\text{rtag}=1$
26. 一棵左子树为空的二叉树在先序线索化后, 其中空的链域的个数是 ()。
- A. 不确定 B. 0 个 C. 1 个 D. 2 个
27. 在线索二叉树中, 下列说法不正确的是 ()。
- A. 在中序线索树中, 若某结点有右孩子, 则其后继结点是它的右子树的最左下结点
- B. 在中序线索树中, 若某结点有左孩子, 则其前驱结点是它的左子树的最右下结点
- C. 线索二叉树是利用二叉树的 $n+1$ 个空指针来存放结点的前驱和后继信息的
- D. 每个结点通过线索都可以直接找到它的前驱和后继
28. 【2010 年计算机联考真题】

下列线索二叉树中 (用虚线表示线索), 符合后序线索树定义的是 ()。



29. 二叉树在线索化后, 仍不能有效求解的问题是 ()。
- A. 先序线索二叉树中求先序后继 B. 中序线索二叉树中求中序后继
- C. 中序线索二叉树中求中序前驱 D. 后序线索二叉树中求后序后继
30. 若 X 是二叉中序线索树中一个有左孩子的结点, 且 X 不为根, 则 X 的前驱为 ()。
- A. X 的双亲 B. X 的右子树中最左的结点
- C. X 的左子树中最右结点 D. X 的左子树中最右叶结点
31. () 的遍历仍需要栈的支持。

- A. 前序线索树 B. 中序线索树
C. 后序线索树 D. 所有线索树

32. 【2013 年计算机联考真题】

若 X 是后序线索二叉树中的叶结点, 且 X 存在左兄弟结点 Y, 则 X 的右线索指向的是()。

- A. X 的父结点
B. 以 Y 为根的子树的最左下结点
C. X 的左兄弟结点 Y
D. 以 Y 为根的子树的最右下结点

33. 【2014 年计算机联考真题】

若对如右图所示的二叉树进行中序线索化, 则结点 x 的左、右线索指向的结点分别是()。

- A. e、c B. e、a
C. d、c D. b、a

34. 【2015 年计算机联考真题】

先序序列为 a,b,c,d 的不同二叉树的个数是()。

- A. 13 B. 14 C. 15 D. 16

二、综合应用题

- 若某非空二叉树的先序序列和后序序列正好相反, 则该二叉树的形态是什么?
- 若某非空二叉树的先序序列和后序序列正好相同, 则该二叉树的形态是什么?
- 编写后序遍历二叉树的非递归算法。
- 试给出二叉树的自下而上、从右到左的层次遍历算法。
- 假设二叉树采用二叉链表存储结构, 设计一个非递归算法求二叉树的高度。
- 设一棵二叉树中各结点的值互不相同, 其先序遍历序列和中序遍历序列分别存于两个一维数组 A[1...n] 和 B[1...n] 中, 试编写算法建立该二叉树的二叉链表。
- 二叉树按二叉链表形式存储, 写一个判别给定二叉树是否是完全二叉树的算法。
- 假设二叉树采用二叉链表存储结构存储, 试设计一个算法, 计算一棵给定二叉树的所有双分支结点个数。
- 设树 B 是一棵采用链式结构存储的二叉树, 编写一个把树 B 中所有结点的左、右子树进行交换的函数。
- 假设二叉树采用二叉链存储结构存储, 设计一个算法, 求先序遍历序列中第 k ($1 \leq k \leq$ 二叉树中结点个数) 个结点的值。
- 已知二叉树以二叉链表存储, 编写算法完成: 对于树中每一个元素值为 x 的结点, 删去以它为根的子树, 并释放相应的空间。
- 在二叉树中查找值为 x 的结点, 试编写算法 (用 C 语言) 打印值为 x 的结点的所有祖先, 假设值为 x 的结点不多于一个。
- 设一棵二叉树的结点结构为 (LLINK, INFO, RLINK), ROOT 为指向该二叉树根结点的指针, p 和 q 分别为指向该二叉树中任意两个结点的指针, 试编写算法 ANCESTOR (ROOT, p, q, r), 该算法找到 p 和 q 的最近公共祖先结点 r。
- 假设二叉树采用二叉链表存储结构, 设计一个算法, 求非空二叉树 b 的宽度 (即具有结点数最多的那一层的结点个数)。
- 设有一棵满二叉树 (所有结点值均不同), 已知其先序序列为 pre, 设计一个算法求其后序序列 post。

16. 设计一个算法将二叉树的叶结点按从左到右的顺序连成一个单链表, 表头指针为 head。二叉树按二叉链表方式存储, 链接时用叶结点的右指针域来存放单链表指针。

17. 试设计判断两棵二叉树是否相似的算法。所谓二叉树 T1 和 T2 相似, 指的是 T1 和 T2 都是空的二叉树或都只有一个根结点; 或 T1 的左子树和 T2 的左子树是相似的且 T1 的右子树和 T2 的右子树是相似的。

18. 写出在中序线索二叉树里查找指定结点在后序的前驱结点的算法。

19. 【2014 年计算机联考真题】

二叉树的带权路径长度 (WPL) 是二叉树中所有叶结点的带权路径长度之和。给定一棵二叉树 T, 采用二叉链表存储, 结点结构为:

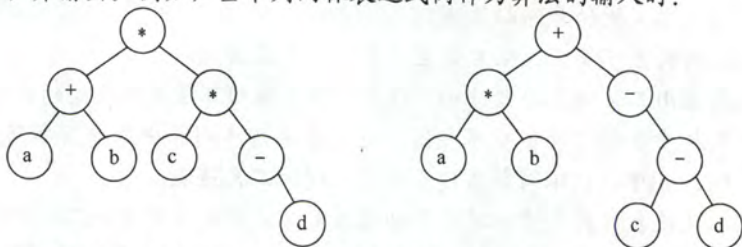
left	weight	right
------	--------	-------

其中叶结点的 weight 域保存该结点的非负权值。设 root 为指向 T 的根结点的指针, 请设计求 T 的 WPL 的算法, 要求:

- 1) 给出算法的基本设计思想;
- 2) 使用 C 或 C++ 语言, 给出二叉树结点的数据类型定义;
- 3) 根据设计思想, 采用 C 或 C++ 语言描述算法, 关键之处给出注释。

20. 【2017 年计算机联考真题】

请设计一个算法, 将给定的表达式树 (二叉树) 转换为等价的中缀表达式 (通过括号反映操作符的计算次序) 并输出。例如, 当下列两棵表达式树作为算法的输入时:



输出的等价中缀表达式分别为 $(a+b)*(c*(-d))$ 和 $(a*b)+(-(c-d))$ 。

二叉树结点定义如下:

```
typedef struct node{
    char data[10];           //存储操作数或操作符
    struct node *left, *right;
}BTree;
```

要求:

- 1) 给出算法的基本设计思想。
- 2) 根据设计思想, 采用 C 或 C++ 语言描述算法, 关键之处给出注释。

4.3.4 答案与解析

一、单项选择题

1. C

二叉树中序遍历的最后一个结点一定是从根开始沿右子女指针链走到底的结点, 设用 p 指示。若结点 p 不是叶子结点 (其左子树非空), 则前序遍历的最后一个结点在它的左子树中, A、B 错; 若结点 p 是叶子结点, 则前序与中序遍历的最后一个结点就是它, C 正确。若中序遍历的最后一个结点 p 不是叶子结点, 它还有一个左子女 q, 结点 q 是叶子结点, 那么结点 q 是前序遍历的最

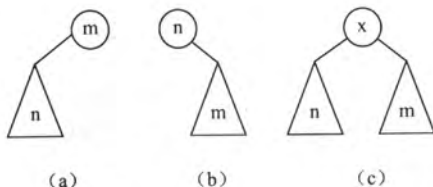
后一个结点,但不是中序遍历的最后一个结点,D错。

2. C

这3种遍历方式中无论哪种遍历方式,都是先遍历左子树,再遍历右子树,所以结点b一定在结点c的前面访问。

3. C

中序遍历时,先访问左子树,再访问根结点,后访问右子树。 n 在 m 前的3种可能性如下图所示,从中看出 n 总是在 m 的左方。



另解: 设 n 和 m 的最近公共祖先 p , 则有以下可能:

情形1, m 和 n 分别在 p 的左、右(右、左)分支上; 情形2, m 或 n 为 p 结点, 另一结点在 p 的分支上。只有 n 和 m 分别处于 p 的左、右分支上, m 为祖先结点且 n 位于 m 的左分支, n 为祖先结点且 m 位于 n 的右分支符合题意。

4. D

后序遍历的顺序是LRN, 如果 n 在 V 的左子树, m 在 N 的右子树, 则在后序遍历的过程当中 n 在 m 之前访问; 如果 n 是 m 的子孙, 设 m 在 N 的位置, 则 n 无论在 m 的左子树还是右子树, 在后序遍历的过程当中 n 都在 m 之前访问。其他都不可以。若C要成立, 要加上两个结点位于同一层。

5. C

在后序遍历退回时访问根结点, 就可以从下向上把从 n 到 m 的路径上的结点输出, 如果采用非递归的算法, 当后序遍历访问到 n 时, 栈中把从根到 n 的父指针的路径上的结点都记忆下来, 也可以找到从 m 到 n 的路径。

6. B

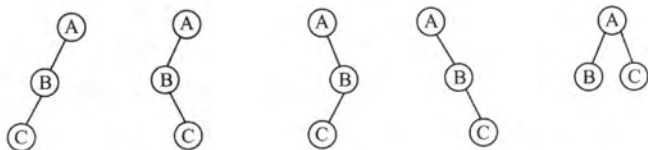
在3种遍历方式中, 访问左右子树的先后顺序是不变的, 只是访问根结点的顺序不同, 因此叶子结点的先后顺序完全相同。此外, 读者可以采用特殊值法, 画一个结点数为3的满二叉树, 采用三种遍历方式来验证答案的正确性。

7. C

对每个顶点从1开始按序编号, 要求结点编号大于其左、右孩子编号, 并且左孩子编号小于右孩子编号。编号越大说明遍历顺序越靠后, 因此, 三者遍历顺序为先左子树再右子树后根结点, 4个选项中仅后序遍历满足要求。

8. D

前序为A、B、C的不同二叉树共有5种, 其中后序为C、B、A的有4种(前4种), 都是单支树, 如下图所示。



9. C

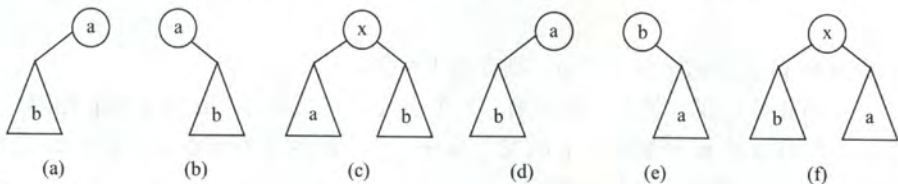
非空树的先序序列和后序序列相反,即“根左右”与“左右根”顺序相反,因此,树只有根结点,或者根结点只有左子树或右子树,依此类推,其子树有同样的性质。因此,树中所有非叶结点的度均为1,即二叉树仅有一个叶结点。

10. C

设二叉树的前序遍历顺序为 NLR, 后序遍历顺序为 LRN。根据题意, 在前序遍历序列中 X 在 Y 前, 在后序遍历序列中 X 在 Y 之后, 若设 X 在根结点的位置, Y 在其左子树或右子树中, 即满足要求。

11. C

先序序列是 $\cdots a \cdots b \cdots$, 则 a 和 b 结点的 3 种情况如下图 (a) ~ (c) 所示。中序序列是 $\cdots b \cdots a \cdots$, 则 a 和 b 结点的 3 种情况如下图 (d) ~ (f) 所示, 相同部分是 b 在 a 的左子树中。



12. D

分析遍历后的结点序列, 可以看出根结点是在中间被访问的, 而且右子树结点在左子树之前, 则遍历的方法是 RNL。本题考查的遍历方法并不是二叉树遍历的 3 种基本遍历方法, 对于考生而言, 重要的是掌握遍历的思想。

13. B

由题可知 1 为根结点。对于 A 选项, 3 应为 1 的左孩子, 先序序列应为 13..., 不符, 类似的选项 C 也错误。对于 B 选项, 当 2 为 1 的右孩子, 3 为 2 的右孩子..., 满足题意。对于 D 选项, 463572 应为 1 的右子树, 2 为 1 的右孩子, 46357 为 2 的左子树, 3 为 2 左孩子, 46 为 3 的左子树, 57 为 3 的右子树, 先序序列 4、6 应相连, 5、7 应相连, 不符。

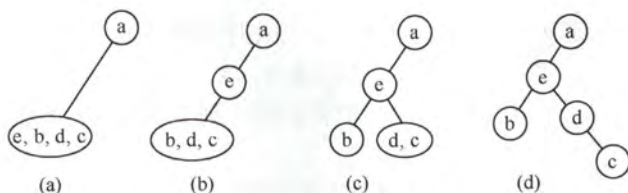
另解: 前序遍历时需要借助栈。前序序列和中序序列的关系相当于以前序序列为入栈次序, 中序序列为出栈次序。题中以 1234567 入栈; 选项 A, 第一个出栈的是 3, 故 1 不可能在 2 之前出栈, 错误。选项 C, 1 不可能在 3 之前出栈, 错误。选项 D, 6 第三个出栈此时栈顶元素是 5, 不是 3, 错误。故选 B。

14. C

前序序列为 LRN, 后序序列为 NLR, 由于前序序列和后序序列刚好相反, 故不可能存在一个结点同时有左右孩子, 即二叉树的高度为 4。1 为根结点, 由于根结点只能有左孩子 (或右孩子), 因此, 在中序序列中, 1 或在序列首或在序列尾, ABCD 皆满足要求。仅考虑以 1 的孩子结点 2 为根结点的子树, 它也只能有左孩子 (或右孩子), 因此, 在中序序列中, 2 或在序列首或在序列尾, ABD 皆满足要求, 故选 C。

15. A

前序序列和后序序列不能唯一确定一棵二叉树, 但可以确定二叉树中结点的祖先关系: 当两个结点的前序序列为 XY 与后序序列为 YX 时, 则 X 为 Y 的祖先。考虑前序序列 a,e,b,d,c、后序序列 b,c,d,e,a, 可知 a 为根结点, e 为 a 的孩子结点; 此外, a 的孩子结点的前序序列 e,b,d,c、后序序列 b,c,d,e, 可知 e 是 bcd 的祖先, 故根结点的孩子结点只有 e。故选 A。



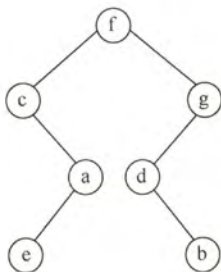
排除法：显然 a 为根结点，且确定 e 为 a 的孩子结点，排除 D。各种遍历算法中左右子树的遍历次序是固定的，若 b 也为 a 的孩子结点，则在前序序列和后序序列中 e 、 b 的相对次序应是不变的，故排除 B，同理排除 C。

特殊法：前序序列和后序序列对应着多棵不同的二叉树树形，我们只需画出满足该条件的任一棵二叉树即可，任意一棵二叉树必定满足正确选项的要求。

显然选 A，最终得到的二叉树满足题设中前序序列和后序序列的要求。

16. B

后序序列是先左子树，接着右子树，最后父结点，递归进行。根结点左子树的叶结点首先被访问，它是 e 。接下来是它的父结点 a ，然后是 a 的父节点 c 。接着访问根结点的右子树。它的叶结点 b 首先被访问，然后是 b 的父结点 d ，再者是 d 的父结点 g 。最后是根结点 f 。因此 d 与 a 同层，选项 B 正确。



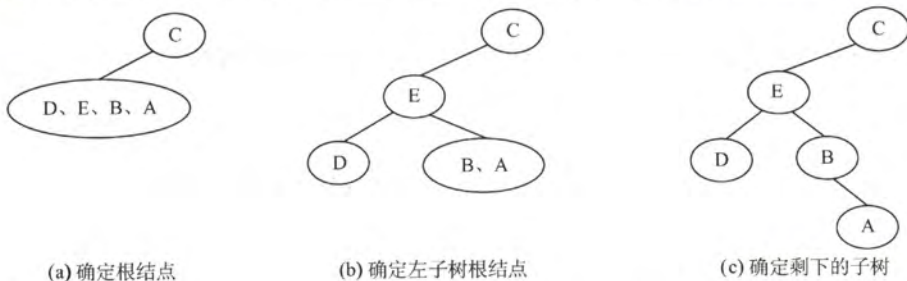
17. D

先序序列为 NLR，后序序列为 LRN，虽然可以唯一确定树的根结点，但是无法划分左、右子树，例如，先序序列为 AB，后序序列为 BA，则其对应的二叉树如下图所示。



18. D

根据后序序列与中序序列可构造出二叉树，如下图所示。由图可知先序序列为 CEDBA。



(a) 确定根结点

(b) 确定左子树根结点

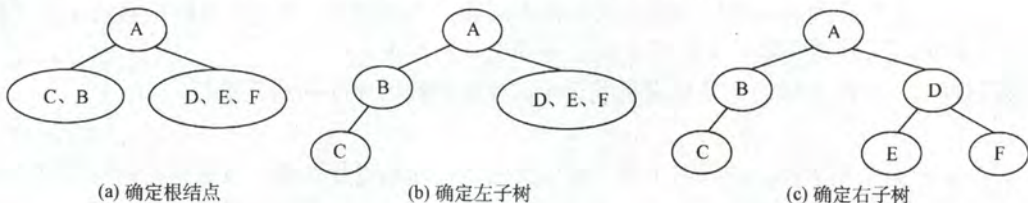
(c) 确定剩下的子树

19. A

对于这种遍历序列问题，先根据遍历的性质排除若干项，若还无法确定答案，再根据遍历结

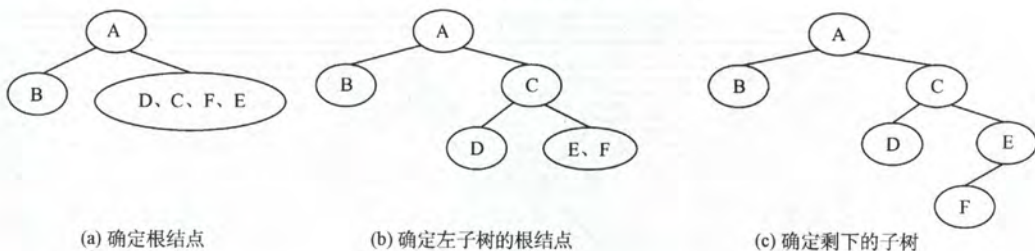
果得到二叉树,找到对应遍历序列。如本题中,已知先序和中序遍历结果,可知本树根结点为A,左子树有C和B,其余为右子树,则后序遍历结果中,A一定在最后,并且C和B一定在前面,排除答案B和D。又因先序中有DEF,中序中有EDF,则D为这个子树的根,所以D在后序中排在EF之后,故答案为A。

根据二叉树递归定义,确定二叉树,要分别找到根结点和左、右子树。因此,根据遍历结果,必定要确定根结点位置和如何划分左、右子树,才可以确定最终的二叉树。故仅有先序和后序遍历不能唯一确定一棵二叉树,而二者其中任一加上中序遍历都可以唯一确定一棵二叉树。如在本题中,根据先序和中序遍历的结果确定二叉树的过程如下图所示。



20. B

可构造出二叉树如下图所示。因此,先序序列为 ABCDEF。



21. B

先序序列是先父结点,接着左子树,然后右子树。中序序列是先左子树,接着父结点,然后右子树,递归进行。如果所有非叶结点只有右子树,先序序列和中序序列都是先父结点,然后右子树,递归进行,因此选项B正确。

22. A

线索是前驱结点和后继结点的指针,引入线索的目的是加快对二叉树的遍历。

23. C

二叉树是一种逻辑结构,但线索二叉树是加上线索后的链表结构,也就是说,它是二叉树在计算机内部的一种存储结构,所以是一种物理结构。

24. C

n 个结点共有链域指针 $2n$ 个,其中,除根结点外,每一个结点都被一个指针指向。剩余的链域建立线索,共 $2n - (n - 1) = n + 1$ 个线索。

25. C

线索二叉树中用 ltag/rtag 标识结点的左/右指针域是否为线索,当其值为 1 时,则对应指针域为线索,其值为 0 时,则为左/右孩子。

26. D

对左子树为空的二叉树进行先序线索化,根结点的左子树为空并且也没有前驱结点(先遍历根结点),先序遍历的最后一个元素为叶结点,左、右子树均为空且有前驱无后继结点,故线索

化后, 树中空链域有 2 个。

27. D

不是每个结点通过线索都可以直接找到它的前驱和后继。在先序线索二叉树中查找一个结点的先序后继很简单, 而查找先序前驱必须知道该结点的双亲结点。同样, 在后序线索二叉树中查找一个结点的后序前驱也很简单, 而查找后序后继也必须知道该结点的双亲结点, 二叉链表中没有存放双亲的指针。

28. D

题中所给二叉树的后序序列为 **dbca**。结点 **d** 无前驱和左子树, 左链域空, 无右子树, 右链域指向其后继结点 **b**; 结点 **b** 无左子树, 左链域指向其前驱结点 **d**; 结点 **c** 无左子树, 左链域指向其前驱结点 **b**, 无右子树, 右链域指向其后继结点 **a**。正确选项为 **D**。

29. D

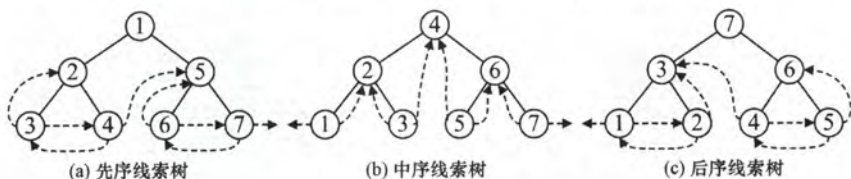
后序线索二叉树中不能有效解决求后序后继的问题。如右图所示, 结点 **E** 的右指针指向右孩子, 而在后序序列中 **E** 的后继结点为 **B**, 在查找 **E** 的后继时后序线索不能起到任何作用, 只能按常规方法来查找。

30. C

在二叉中序线索树中, 某结点若有左孩子, 则按照中序“左根右”的顺序, 该结点的前驱结点为左子树中最右的一个结点 (注意, 并不一定是最右叶子结点)。

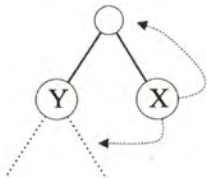
31. C

后序线索树遍历时, 最后访问根结点, 如果是从右孩子 **x** 返回访问父结点, 由于结点 **x** 的右孩子不一定为空 (右指针无法指向其后继), 因此通过指针可能无法遍历整棵树。如下图, 结点中的数字表示遍历的顺序, 图 (c) 中结点 6 的右指针指向其右孩子 5, 而不指向其后序后继结点 7, 因此, 后序遍历还需要栈的支持, 而图 (a) 和图 (b) 均可遍历。



32. A。

根据后序线索二叉树的定义, **X** 结点为叶子结点且有左兄弟, 那么这个结点为右孩子结点, 利用后序遍历的方式可知 **X** 结点的后序后继是其父结点, 即其右线索指向的是父结点。为了更加形象, 在解题的过程中可以画出如下草图。



33. D

线索二叉树的线索实际上指向的是相应遍历序列特定结点的前驱结点和后继结点, 所以先写出二叉树的中序遍历序列: **debxac**, 中序遍历中在 **x** 左边和右边的字符, 就是它在中序线索化的左、右线索, 即 **b**、**a**, 选 **D**。

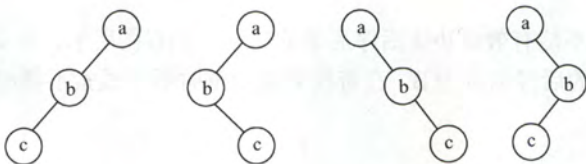
34. B

根据二叉树前序遍历和中序遍历的递归算法中递归工作栈的状态变化得出：前序序列和中序序列的关系相当于以前序序列为入栈次序，以中序序列为出栈次序。因为前序序列和中序序列可以唯一地确定一棵二叉树，所以题意相当于“以序列 a,b,c,d 为入栈次序，则出栈序列的个数为？”，对于 n 个不同元素进栈，出栈序列的个数为 $\frac{1}{n+1}C_{2n}^n = 14$ 。

二、综合应用题

1. 解答：

二叉树的先序序列是 NLR，后序序列是 LRN。要使 NLR=NRL（后序序列反序）成立，则 L 或 R 为空，这样的二叉树每层只有一个结点，即二叉树的形态是其高度等于结点个数。以 3 个结点 a、b、c 为例，其形态如下图所示。



2. 解答：

二叉树的先序序列是 NLR，后序序列是 LRN。要使 NLR=LRN 成立，则 L 和 R 均为空，所以满足条件的二叉树只有一个根结点。

3. 解答：

算法的思想：因为后序非递归遍历二叉树的顺序是先访问左子树，再访问右子树，最后访问根结点。当用堆栈来存储结点，必须分清返回根结点时，是从左子树返回的，还是从右子树返回的。所以，使用辅助指针 r，其指向最近访问过的结点。也可以在结点中增加一个标志域，记录是否已被访问。

```
void PostOrder(BiTree T){
    InitStack(S);
    p=T;
    r=NULL;
    while(p||!IsEmpty(S)){
        if(p){
            push(S,p);
            p=p->lchild;
        }
        else{
            GetTop(S,p);
            if(p->rchild&& p->rchild!=r){
                p=p->rchild;
                push(S,p);
                p=p->lchild;
            }
            else{
                pop(S,p);
                visit(p->data);
                r=p;
                p=NULL;
            }
        }
    }
}
```

```

    }//while
}

```

当访问一个结点*p时, 栈中结点恰好是*p结点的所有祖先。从栈底到栈顶结点再加上*p结点, 刚好构成从根结点到*p结点的一条路径。在很多算法设计中都利用了这一特性求解, 如求根结点到某结点的路径、求两个结点的最近公共祖先等都可以利用这个思路来实现。

4. 解答:

一般的二叉树层次遍历是自上而下、从左到右的遍历, 这里的遍历顺序恰好相反。算法思想: 利用原有的层次遍历算法, 出队的同时将各结点指针入栈, 在所有结点入栈后再从栈顶开始依次访问即是所求的算法。具体实现为

- (1) 把根结点入队列
- (2) 把一个元素出队列, 遍历这个元素
- (3) 依次把这个元素的右孩子、左孩子入队列
- (4) 若队列不空, 则跳到(2), 否则结束。

算法实现如下:

```

void InvertLevel(BiTree bt){
    Stack s; Queue Q;
    if(bt!=NULL){
        InitStack(s);           //栈初始化, 栈中存放二叉树结点的指针
        InitQueue(Q);           //队列初始化, 队列中存放二叉树结点的指针
        EnQueue(Q, bt);
        while(IsEmpty(Q)==false){ //从上而下层次遍历
            DeQueue(Q, p);
            Push(s, p);           //出队, 入栈
            if(p->lchild)
                EnQueue(Q, p->lchild); //若左子女不空, 则入队列
            if(p->rchild)
                EnQueue(Q, p->rchild); //若右子女不空, 则入队列
        }
        while(IsEmpty(s)==false){
            Pop(s, p);
            visit(p->data);
        } //自下而上、从右到左的层次遍历
    } //if 结束
}

```

5. 解答:

采用层次遍历的算法, 设置变量 level 记录当前结点所在层数, 设置变量 last 指向当前层最右结点, 每次层次遍历出队时与 last 指针比较, 若两者相等, 那么层数加 1, 并让 last 指向下一层最右结点, 至少遍历完成。level 的值即为二叉树的高度。

```

int Btdepth(BiTree T){
    //采用层次遍历的非递归方法求解二叉树的高度
    if(!T)
        return 0; //树空, 高度为 0
    int front=-1, rear=-1;
    int last=0, level=0; //last 指向下一层第一个结点的位置
    BiTree Q[MaxSize]; //设置队列 Q, 元素是二叉树结点指针且容量足够
    Q[++rear]=T; //将根结点入队
    BiTree p;

```



```

while(front<rear){
    p=Q[++front];
    if(p->lchild)
        Q[++rear]=p->lchild;
    if(p->rchild)
        Q[++rear]=p->rchild;
    if(front==last){
        level++;
        last=rear;
    }
}
return level;
}

```

//队不空, 则循环
//队列元素出队, 即正在访问的结点
//左孩子入队
//右孩子入队
//处理该层的最右结点
//层数增1
//last 指向下层

求某一层结点个数、每一层结点个数、树的最大宽度等都是采用与此题类似的思想。当然此题可编写递归算法, 其实现如下:

```

int Btdepth2(BiTree T){
    if(T==NULL)
        return 0;
    ldep=Btdepth(T->lchild);
    rdep=Btdepth(T->rchild);
    if(ldep>rdep)
        return ldep+1;
    else
        return rdep+1;
}

```

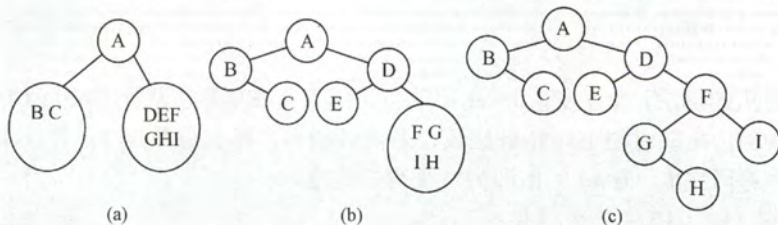
//空树, 高度为0
//左子树高度
//右子树高度
//树的高度为子树最大高度加根结点

6. 解答:

由先序序列和中序序列可以唯一确定一棵二叉树。算法的实现步骤如下:

- 1) 根据先序序列确定树的根结点。
- 2) 根据根结点在中序序列中划分出二叉树的左、右子树包含哪些结点。然后根据左、右子树结点在先序序列中的次序可以确定子树的根结点, 即回到步骤1)。

如此重复上述步骤, 直到每棵子树仅有一个结点(该子树的根结点)为止, 如下图所示。



```

BiTree PreInCreat (ElemType A[], ElemType B[], int l1, int h1, int l2, int h2){
    //l1, h1 为先序的第一和最后一个结点下标, l2, h2 为中序的第一和最后一个结点下标
    //初始调用时, l1=l2=1, h1=h2=n
    root=(BiTNode*)malloc(sizeof(BiTNode));
    root->data=A[l1];
    for(i=l2; B[i]!=root->data; i++);
    llen=i-l2;
    rlen=h2-i;
}

```

//建根结点
//根结点
//根结点在中序序列中的划分
//左子树长度
//右子树长度


```

        if (llen) //递归建立左子树
            root->lchild=PreInCreat(A,B,ll+1,ll+llen,l2,l2+llen-1);
        else //左子树为空
            root->lchild=NULL;
        if (rlen) //递归建立右子树
            root->rchild=PreInCreat(A,B,h1-rlen+1,h1,h2-rlen+1,h2);
        else //右子树为空
            root->rchild=NULL;
        return root; //返回根结点指针
    }

```

7. 解答:

根据完全二叉树的定义, 具有 n 个结点的完全二叉树与满二叉树中编号从 $1 \sim n$ 的结点一一对应。算法思想: 采用层次遍历的算法, 将所有结点加入队列 (包括空结点)。当遇到空结点时, 查看其后是否有非空结点。若有, 则二叉树不是完全二叉树。

```

bool IsComplete(BiTree T) {
    //本算法判断给定二叉树是否为完全二叉树
    InitQueue(Q);
    if (!T)
        return 1; //空树为满二叉树
    EnQueue(Q, T);
    while (!IsEmpty(Q)) {
        DeQueue(Q, p);
        if (p) { //结点非空, 将其左、右子树入队列
            EnQueue(Q, p->lchild);
            EnQueue(Q, p->rchild);
        }
        else //结点为空, 检查其后是否有非空结点
            while (!IsEmpty(Q)) {
                DeQueue(Q, p);
                if (p) //结点非空, 则二叉树为非完全二叉树
                    return 0;
            }
    }
    return 1;
}

```

8. 解答:

计算一棵二叉树 b 中所有双分支结点个数的递归模型 $f(b)$ 如下:

$f(b)=0$	若 $b=NULL$
$f(b)=f(b->lchild)+f(b->rchild)+1$	若 $*b$ 为双分支结点
$f(b)=f(b->lchild)+f(b->rchild)$	其他情况 ($*b$ 为单分支结点或叶子结点)

具体实现算法如下:

```

int DsonNodes(BiTree b) {
    if (b==NULL)
        return 0;
    else if (b->lchild!=NULL&& b->rchild!=NULL) //双分支结点
        return DsonNodes(b->lchild)+DsonNodes(b->rchild)+1;
    else
        return DsonNodes(b->lchild)+DsonNodes(b->rchild);
}

```


}

当然本题也可以设置一个全局变量 Num, 当每遍历到一个结点时, 判断每个结点是否为分支结点 (左、右结点都不为空, 注意是双分支), 若是则 Num++。

9. 解答:

采用递归算法实现交换二叉树的左、右子树, 首先交换 b 结点的左孩子的左、右子树, 然后交换 b 结点的右孩子的左、右子树, 最后交换 b 结点的左、右孩子, 当结点为空时递归结束 (后序遍历的思想)。算法实现如下:

```
void swap(BiTree b){
    //本算法递归地交换二叉树的左、右子树
    if(b){
        swap(b->lchild);           //递归地交换左子树
        swap(b->rchild);           //递归地交换右子树
        temp=b->lchild;            //交换左、右孩子结点
        b->lchild=b->rchild;
        b->rchild=temp;
    }
}
```

10. 解答:

设置一个全局变量 i 记录已访问过的结点的序号, 其初值是根结点在先序序列中的序号, 即为 1。当二叉树 b 为空时返回特殊字符 '#', 当 $i=k$ 时, 表示找到了满足条件的结点, 返回 b->data; 当 $i \neq k$ 时, 则递归地在左子树中查找, 若找到了则返回该值, 否则继续递归地在右子树中查找, 并返回其结果。实现算法如下:

```
int i=1;                               //遍历序号的全局变量
ElemType PreNode(BiTree b,int k){
    //本算法查找二叉树先序遍历序列中第 k 个结点的值
    if(b==NULL)                         //空结点, 则返回特殊字符
        return '#';
    if(i==k)                             //相等, 则当前结点即为第 k 个结点
        return b->data;
    i++;
    ch=PreNode(b->lchild,k);             //左子树中递归寻找
    if(ch!='#')                          //在左子树中, 则返回该值
        return ch;
    ch=PreNode(b->rchild,k);             //在右子树中递归寻找
    return ch;
}
```

本题实质上就是一个遍历算法的实现, 只不过用一个全局变量来记录访问的序号, 求其他遍历序列的第 k 个结点也采用相似的方法。二叉树的遍历算法可以引申出大量的算法题, 因此考生务必要熟练掌握二叉树的遍历算法。

11. 解答:

删除以元素值 x 为根的子树, 只要能删除其左、右子树, 就可以释放值为 x 的根结点, 因此宜采用后序遍历。算法思想: 删除值为 x 的结点, 意味着应将其父结点的左(右)子女指针置空, 用层次遍历易于找到某结点的父结点。本题要求删除树中每一个元素值为 x 的结点的子树, 因此要遍历完整棵二叉树。算法实现如下:

```
void DeleteXTree(BiTree bt){           //删除以 bt 为根的子树
    if(bt){
        DeleteXTree(bt->lchild);
    }
```



```

        DeleteXTree(bt->rchild);    //删除 bt 的左子树、右子树
        free(bt);                  //释放被删结点所占的存储空间
    }
}
//在二叉树上查找所有以 x 为元素值的结点, 并删除以其为根的子树
void Search(BiTree bt, ElemType x) {
    BiTree Q[];                    //Q 是存放二叉树结点指针的队列, 容量足够大
    if(bt) {
        if(bt->data==x) {           //若根结点值为 x, 则删除整棵树
            DeleteXTree(bt);
            exit(0);
        }
        Init Queue(Q);
        EnQueue(Q, bt);
        while(!IsEmpty(Q)) {
            DeQueue(Q, p);
            if(p->lchild)             //若左子女非空
                if(p->lchild->data==x) { //左子树符合则删除左子树
                    DeleteXTree(p->lchild);
                    p->lchild=NULL;
                }                     //父结点的左子女置空
            else
                EnQueue(Q, p->lchild); //左子树入队列
            if(p->rchild)             //若右子女非空
                if(p->rchild->data==x) { //右子女符合则删除右子树
                    DeleteXTree(p->rchild);
                    p->rchild=NULL;
                }                     //父结点的右子女置空
            else
                EnQueue(Q, p->rchild); //右子女入队列
        }
    }
}

```

12. 解答:

算法思想: 采用非递归后序遍历, 最后访问根结点, 当访问到值为 x 的结点时, 栈中所有元素均为该结点的祖先, 依次出栈打印即可。

```

typedef struct {
    BiTree t;
    int tag;
} stack; //tag=0 表示左子女被访问, tag=1 表示右子女被访问
void Search(BiTree bt, ElemType x) {
    //在二叉树 bt 中, 查找值为 x 的结点, 并打印其所有祖先
    stack s[]; //栈容量足够大
    top=0;
    while(bt!=NULL || top>0) {
        while(bt!=NULL && bt->data!=x) { //结点入栈
            s[++top].t=bt;
            s[top].tag=0;
            bt=bt->lchild; //沿左分支向下
        }
        if(bt->data==x) {
            printf("所查结点的所有祖先结点的值为:\n"); //找到 x
            for(i=1; i<=top; i++)

```



```

        printf("%d", s[i].t->data);           //输出祖先值后结束
        exit(1);
    }
    while(top!=0&& s[top].tag==1)
        top--;                               //退栈（空遍历）
    if(top!=0){
        s[top].tag=1;
        bt=s[top].t->rchild;                 //沿右分支向下遍历
    }
} //while(bt!=NULL||top>0)
}

```

因为查找的过程就是后序遍历的过程，因此使用的栈的深度不超过树的深度。

13. 解答：

后序遍历最后访问根结点，即在递归算法中，根是压在栈底的。本题要找 p 和 q 的最近公共祖先结点 r ，不失一般性，设 p 在 q 的左边。算法思想：采用后序非递归算法，栈中存放二叉树结点的指针，当访问到某结点时，栈中所有元素均为该结点的祖先。后序遍历必然先遍历到结点 p ，栈中元素均为 p 的祖先。先将栈复制到另一辅助栈中。继续遍历到结点 q 时，将栈中元素从栈顶开始逐个到辅助栈中去匹配，第一个匹配（即相等）的元素就是结点 p 和 q 的最近公共祖先。算法实现如下：

```

typedef struct{
    BiTree t;
    int tag; //tag=0 表示左子女已被访问，tag=1 表示右子女已被访问
}stack;
stack s[], s1[]; //栈，容量足够大
BiTree Ancestor(BiTree ROOT, BiTNode *p, BiTNode *q){
    //本算法求二叉树中 p 和 q 指向结点的最近公共结点
    top=0; bt=ROOT;
    while(bt!=NULL||top>0){
        while(bt!=NULL&&bt!=p&&bt!=q) //结点入栈
            while(bt!=NULL){
                s[++top].t=bt;
                s[top].tag=0;
                bt=bt->lchild;
            } //沿左分支向下
        while(top!=0&& s[top].tag==1){
            //假定 p 在 q 的左侧，遇到 p 时，栈中元素均为 p 的祖先
            if(s[top].t==p){
                for(i=1; i<=top; i++){
                    s1[i]=s[i];
                    topl=top;
                } //将栈 s 的元素转入辅助栈 s1 保存
            }
            if(s[top].t==q) //找到 q 结点
                for(i=top; i>0; i--){ //将栈中元素的树结点到 s1 中去匹配
                    for(j=top1; j>0; j--){
                        if(s1[j].t==s[i].t)
                            return s[i].t; //p 和 q 的最近公共祖先已找到
                    }
                }
            top--; //退栈
        } //while
        if(top!=0){
            s[top].tag=1;

```



```

        bt=s[top].t->rchild;
    }
    } //while
    return NULL;
}

```

//沿右分支向下遍历
//p 和 q 无公共祖先

14. 解答:

采用层次遍历的方法求出所有结点的层次, 并将所有结点和对应的层次放在一个队列中。然后通过扫描队列求出各层的结点总数, 最大的层结点总数即为二叉树的宽度。算法实现如下:

```

typedef struct{
    BiTree data[MaxSize];    //保存队列中的结点指针
    int level[MaxSize];      //保存 data 中相同下标结点的层次
    int front, rear;
}Qu;

int BTWidth(BiTree b){
    BiTree p;
    int k, max, i, n;
    Qu.front=Qu.rear=-1;    //队列为空
    Qu.rear++;
    Qu.data[Qu.rear]=b;     //根结点指针入队
    Qu.level[Qu.rear]=1;    //根结点层次为 1
    while(Qu.front<Qu.rear){
        Qu.front++;        //出队
        p=Qu.data[Qu.front]; //出队结点
        k=Qu.level[Qu.front]; //出队结点的层次
        if(p->lchild!=NULL){ //左孩子进队列
            Qu.rear++;
            Qu.data[Qu.rear]=p->lchild;
            Qu.level[Qu.rear]=k+1;
        }
        if(p->rchild!=NULL){ //右孩子进队列
            Qu.rear++;
            Qu.data[Qu.rear]=p->rchild;
            Qu.level[Qu.rear]=k+1;
        }
    } //while
    max=0; i=0;            //max 保存同一层最多的结点个数
    k=1;                    //k 表示从第一层开始查找
    while(i<=Qu.rear){     //i 扫描队中所有元素
        n=0;                //n 统计第 k 层的结点个数
        while(i<=Qu.rear&&Qu.level[i]==k){
            n++;
            i++;
        }
        k=Qu.level[i];
        if(n>max) max=n;    //保存最大的 n
    }
    return max;
}

```

注意: 本题队列中的结点, 在出队后仍需要保留在队列中以便求二叉树的宽度, 所以, 设置的队列采用非环形队列, 否则在出队后可能被其他结点覆盖, 无法再求二叉树的宽度。

15. 解答:

对一般二叉树,仅根据先序或后序序列,不能确定另一个遍历序列。但对满二叉树,任一结点的左、右子树均含有相等的结点数,同时,先序序列的第一个结点作为后序序列的最后一个结点,由此得到将先序序列 $pre[l1..h1]$ 转换为后序序列 $post[l2..h2]$ 的递归模型如下:

```
f(pre, l1, h1, post, l2, h2) = 不做任何事情          当  $h1 < l1$  时
f(pre, l1, h1, post, l2, h2) = post[h2] = pre[l1];      其他情况
    取中间位置  $half = (h1 - l1) / 2$ ;
    将  $pre[l1+1, l1+half]$  左子树转换为  $post[l2, l2+half-1]$ ,
    即  $f(pre, l1+1, l1+half, post, l2, l2+half-1)$ ;
    将  $pre[l1+half+1, h1]$ , 右子树转换为  $post[l2+half, h2-1]$ ,
    即  $f(pre, l1+half+1, h1, post, l2+half, h2-1)$ ;
```

其中, $post[h2] = pre[l1]$ 表示后序序列的最后一个结点(根结点)等于先序序列的第一个结点(根结点)。相应的算法实现如下:

```
void PreToPost (ElemType pre[], int l1, int h1, ElemType post[], int l2, int h2) {
    int half;
    if (h1 >= l1) {
        post[h2] = pre[l1];
        half = (h1 - l1) / 2;
        PreToPost (pre, l1+1, l1+half, post, l2, l2+half-1); // 转换左子树
        PreToPost (pre, l1+half+1, h1, post, l2+half, h2-1); // 转换右子树
    }
}
```

例如,有以下代码:

```
ElemType *pre = "ABCDEFGH";
ElemType post[MaxSize];
PreToPost (pre, 0, 6, post, 0, 6);
printf("后序序列: ");
for (int i = 0; i <= 6; i++)
    printf("%c", post[i]);
printf("\n");
```

执行结果如下:

后序序列: C D B F G E A

16. 解答:

通常我们所使用的先序、中序和后序遍历对于叶结点的访问都是从左到右的顺序,这里我们选择中序递归遍历。算法思想:设置前驱结点指针 pre , 初始为空。第一个叶结点由指针 $head$ 指向,遍历到叶结点时,就把它前驱的 $rchild$ 指针指向它,最后一个叶结点的 $rchild$ 为空。算法实现如下:

```
LinkedList head, pre = NULL; // 全局变量
LinkedList InOrder (BiTree bt) {
    if (bt) {
        InOrder (bt->lchild); // 中序遍历左子树
        if (bt->lchild == NULL && bt->rchild == NULL) // 叶结点
            if (pre == NULL) {
                head = bt;
                pre = bt;
            } // 处理第一个叶结点
    }
}
```

```

        else{
            pre->rchild=bt;
            pre=bt;
        }
        InOrder(bt->rchild);
        pre->rchild=NULL;
    }
    return head;
}

```

//将叶结点链入链表
//中序遍历右子树
//设置链表尾

上述算法的时间复杂度为 $O(n)$ ，辅助变量使用 head 和 pre，栈空间复杂度为 $O(n)$ 。

17. 解答:

本题采用递归的思想求解，若 T_1 和 T_2 都是空树，则相似；若有一个为空另一个不空，则必然不相似；否则递归地比较它们的左、右子树是否相似。递归函数的定义如下：

- a) $f(T_1, T_2)=1$ ；若 $T_1=T_2=NULL$ 。
- b) $f(T_1, T_2)=0$ ；若 T_1 和 T_2 之一为 NULL，另一个不为 NULL。
- c) $f(T_1, T_2)=f(T_1 \rightarrow lchild, T_2 \rightarrow lchild) \&\& f(T_1 \rightarrow rchild, T_2 \rightarrow rchild)$ ；若 T_1 和 T_2 均不为 NULL。

因此，算法实现如下：

```

int similar(BiTree T1, BiTree T2){
    //采用递归的算法判断两个二叉树是否相似
    int leftS, rightS;
    if (T1==NULL&&T2==NULL)           //两树皆空
        return 1;
    else if (T1==NULL||T2==NULL)       //只有一树为空
        return 0;
    else{                               //递归判断
        leftS=similar(T1->lchild, T2->lchild);
        rightS=similar(T1->rchild, T2->rchild);
        return leftS&&rightS;
    }
}

```

18. 解答:

算法思想：在后序序列中，若结点 p 有右子女，则右子女是其前驱，若无右子女而有左子女，则左子女是其前驱。若结点 p 左、右子女均无，设其中序左线索指向某祖先结点 f (p 是 f 右子树中按中序遍历的第一个结点)，若 f 有左子女，则其左子女是结点 p 在后序下的前驱；若 f 无左子女，则顺其前驱找双亲的双亲，一直找到双亲有左子女（这时左子女是 p 的前驱）。还有一种情况，若 p 是中序遍历的第一个结点，结点 p 在中序和后序下均无前驱。

算法的代码：

```

BiThrTree InPostPre(BiThrTree t, BiThrTree p){
    //在中序线索二叉树 t 中，求指定结点 p 在后序下的前驱结点 q
    BiThrTree q;
    if (p->rtag==0)           //若 p 有右子女，则右子女是其后序前驱
        q=p->rchild;
    else if (p->ltag==0)      //若 p 只有左子女，左子女是其后序前驱
        q=p->lchild;
    else if (p->lchild==NULL)
        q=NULL;              //p 是中序序列第一结点，无后序前驱
}

```



```

else{    //顺左线索向上找p的祖先,若存在,再找祖先的左子女
    while(p->ltag==1&& p->lchild!=NULL)
        p=p->lchild;
    if(p->ltag==0)
        q=p->lchild;    //p结点的祖先的左子女是其后续前驱
    else
        q=NULL;        //仅有单支树(p是叶子),已到根结点,p无后续前驱
}
return q;
}

```

19. 解答:

考查二叉树的带权路径长度, 二叉树的带权路径长度为每个叶结点的深度与权值之积的总和, 可以使用先序遍历或层次遍历解决问题(考生只需采用一种思路即可)。

1) 算法的基本设计思想:

① 基于先序递归遍历的算法思想是用一个 static 变量记录 wpl, 把每个结点的深度作为递归函数的一个参数传递, 算法步骤如下:

若该结点是叶结点, 那么变量 wpl 加上该结点的深度与权值之积;

若该结点是非叶结点, 那么若左子树不为空, 对左子树调用递归算法; 若右子树不为空, 对右子树调用递归算法, 深度参数均为本结点的深度参数加 1;

最后返回计算出的 wpl 即可。

② 基于层次遍历的算法思想是使用队列进行层次遍历, 并记录当前的层数,

当遍历到叶结点时, 累计 wpl;

当遍历到非叶结点时, 把该结点的子树加入队列;

当某结点为该层的最后一个结点时, 层数自增 1;

队列空时遍历结束, 返回 wpl。

2) 二叉树结点的数据类型定义如下:

```

typedef struct BiTNode{
    int weight;
    struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;

```

3) 算法的代码如下:

① 基于先序遍历的算法:

```

int WPL(BiTree root){
    return wpl_PreOrder(root,0);
}

int wpl_PreOrder(BiTree root,int deep){
    static int wpl=0;    //定义一个 static 变量存储 wpl
    if(root->lchild==NULL&&root->rchild==NULL) //若为叶结点, 累积 wpl
        wpl+=deep*root->weight;
    if(root->lchild!=NULL)    //若左子树不空, 对左子树递归遍历
        wpl_PreOrder(root->lchild,deep+1);
    if(root->rchild!=NULL)    //若右子树不空, 对右子树递归遍历
        wpl_PreOrder(root->rchild,deep+1);
    return wpl;
}

```

② 基于层次遍历的算法:

```

#define MaxSize 100    //设置队列的最大容量

```



```

int wpl_LevelOrder(BiTree root){
    BiTree q[MaxSize];           //声明队列, end1 为头指针, end2 为尾指针
    int end1, end2;               //队列最多容纳 MaxSize-1 个元素
    end1=end2=0;                 //头指针指向队头元素, 尾指针指向队尾的后一个元素
    int wpl=0, deep=0;           //初始化 wpl 和深度
    BiTree lastNode;              //lastNode 用来记录当前层的最后一个结点
    BiTree newlastNode;           //newlastNode 用来记录下一层的最后一个结点
    lastNode=root;               //lastNode 初始化为根结点
    newlastNode=NULL;            //newlastNode 初始化为空
    q[end2++]=root;              //根结点入队
    while(end1!=end2){           //层次遍历, 若队列不空则循环
        BiTree t=q[end1++];      //拿出队列中的头一个元素
        if(t->lchild==NULL&& t->rchild==NULL){
            wpl+=deep*t->weight;
        }                        //若为叶结点, 统计 wpl
        if(t->lchild!=NULL){      //若非叶结点把左结点入队
            q[end2++]=t->lchild;
            newlastNode=t->lchild;
        }                        //并设下一层的最后一个结点为该结点的左结点
        if(t->rchild!=NULL){      //处理叶结点
            q[end2++]=t->rchild;
            newlastNode=t->rchild;
        }
        if(t==lastNode){         //若该结点为本层最后一个结点, 更新 lastNode
            lastNode=newlastNode;
            deep+=1;              //层数加 1
        }
    }
    return wpl;                  //返回 wpl
}

```

注意: 当 static 关键字用于代码块内部的变量的声明时, 用于修改变量的存储类型, 从自动变量修改为静态变量, 但变量的链接属性和作用域不受影响。用这种方式声明的变量在程序执行之前创建, 并在程序的整个执行期间一直存在, 而不是每次在代码块开始执行时创建, 在代码块执行完毕后销毁。即保持局部变量内容的持久。静态局部变量的生存期虽然为整个源程序, 但是其作用域仍与局部变量相同, 即只能在定义该变量的函数内使用该变量。退出该函数后, 尽管该变量还继续存在, 但不能使用它。读者在该阶段对此只了解即可, 有兴趣的请查阅相关资料。

20. 解答:

1) 算法的基本设计思想

表达式树的中序序列加上必要的括号即为等价的中缀表达式。可以基于二叉树的中序遍历策略得到所需的表达式。

表达式树中分支结点所对应的子表达式的计算次序, 由该分支结点所处的位置决定。为得到正确的中缀表达式, 需要在生成遍历序列的同时, 在适当位置增加必要的括号。显然, 表达式的最外层(对应根结点)及操作数(对应叶结点)不需要添加括号。

2) 算法实现

将二叉树的中序遍历递归算法稍加改造即可得本题答案。除根结点和叶结点外, 遍历到其他结点时在遍历其左子树之前加上左括号, 在遍历完右子树后加上右括号。


```

void BtreeToE(BTree *root){
    BtreeToExp(root, 1);           //根的高度为 1
}
void BtreeToExp( BTree *root, int deep)
{
    if(root == NULL) return;       //空结点返回
    else if(root->left==NULL&&root->right==NULL) //若为叶结点
        printf("%s", root->data);   //输出操作数, 不加括号
    else{
        if(deep>1) printf("(");     //若有子表达式则加1层括号
        BtreeToExp(root->left, deep+1);
        printf("%s", root->data);   //输出操作符
        BtreeToExp(root->right, deep+1);
        if(deep>1) printf(")");    //若有子表达式则加1层括号
    }
}

```

4.4 树、森林

4.4.1 树的存储结构

树的存储方式有多种, 既可以采用顺序存储结构, 也可以采用链式存储结构, 但无论采用何种存储方式, 都要求能唯一地反映出树中各结点之间的逻辑关系, 这里介绍 3 种常用的存储结构。

1. 双亲表示法

这种存储方式采用一组连续空间来存储每个结点, 同时在每个结点中增设一个伪指针, 指示其双亲结点在数组中的位置。如图 4-13 所示, 根结点下标为 0, 其伪指针域为 -1。

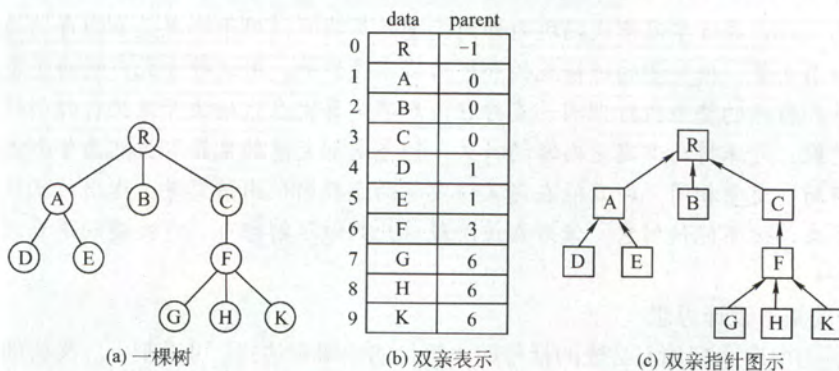


图 4-13 树的双亲表示法

双亲表示法的存储结构描述如下:

```

#define MAX_TREE_SIZE 100           //树中最多结点数
typedef struct                       //树的结点定义
{
    ElemType data;                  //数据元素
    int parent;                      //双亲位置域
}PTNode;

```

```

typedef struct{                                //树的类型定义
    PTNode nodes[MAX_TREE_SIZE];             //双亲表示
    int n;                                     //结点数
}PTree;

```

该存储结构利用了每个结点（根结点除外）只有唯一双亲的性质，可以很快得到每个结点的双亲结点，但是求结点的孩子时却需要遍历整个结构。

2. 孩子表示法

孩子表示法是将每个结点的孩子结点都用单链表链接起来形成一个线性结构，则 N 个结点就有 N 个孩子链表（叶子结点的孩子链表为空表），如图 4-14（a）所示。

对于这种存储方式寻找子女的操作非常直接，而寻找双亲的操作需要遍历 N 个结点中孩子链表指针域所指向的 N 个孩子链表。

3. 孩子兄弟表示法

孩子兄弟表示法又称为二叉树表示法，即以二叉链表作为树的存储结构。孩子兄弟表示法是使每个结点包括三部分内容：结点值、指向结点第一个孩子结点的指针和指向结点下一个兄弟结点的指针（沿此域可以找到结点的所有兄弟结点），如图 4-14（b）所示。

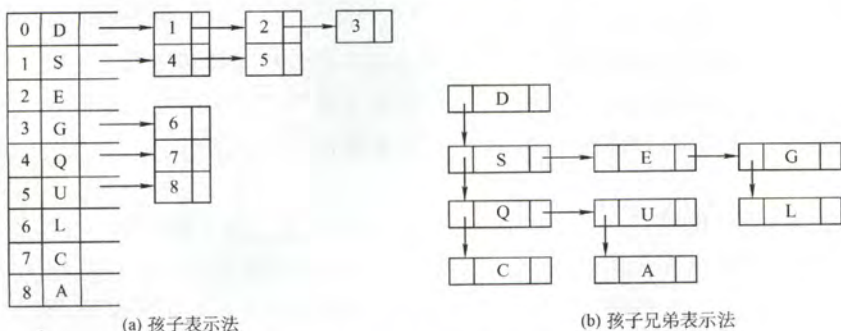


图 4-14 树的孩子表示法和孩子兄弟表示法

孩子兄弟表示法的存储结构描述如下：

```

typedef struct CSNode{
    ElemType data;                                //数据域
    struct CSNode *firstchild, *nextsibling;      //第一个孩子和右兄弟指针
}CSNode, *CSTree;

```

这种存储表示法比较灵活。其最大的优点就是可以方便地实现树转换为二叉树的操作，易于查找结点的孩子等，但缺点是从当前结点查找其双亲结点比较麻烦。如果为每个结点增设一个 `parent` 域指向其父结点，则查找结点的父结点也很方便。

4.4.2 树、森林与二叉树的转换

由于二叉树和树都可以用二叉链表作为存储结构，则以二叉链表作为媒介可以导出树与二叉树的一个对应关系，即给定一棵树，可以找到唯一的一棵二叉树与之对应。从物理结构上看，树的孩子兄弟表示法与二叉树的二叉链表表示法相同，即每个结点共有两个指针，分别指向结点第一个孩子和结点的下一兄弟结点，而二叉链表中使用双指针。因此，就可以用同一存储结构的不同解释将一棵树转换为二叉树。

树转换为二叉树的规则：每个结点左指针指向它的第一个孩子结点，右指针指向它在树中的

相邻兄弟结点,可表示为“左孩子右兄弟”。由于根结点没有兄弟,所以,由树转换而得的二叉树没有右子树,如图 4-15 所示。

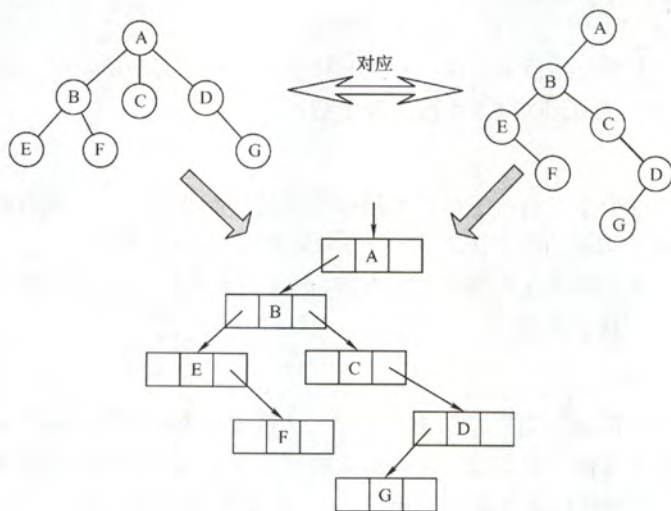


图 4-15 树转换为二叉树

将森林转换为二叉树的规则与树类似。先将森林中的每一棵树转换为二叉树,再将第一棵树的根作为转换后的二叉树的根,第一棵树的左子树作为转换后二叉树根的左子树,第二棵树作为转换后二叉树的右子树,第三棵树作为转换后二叉树根的右子树的右子树,依此类推,就可将森林转换为二叉树。

二叉树转换为森林的规则:若二叉树非空,则二叉树根及其左子树为第一棵树的二叉树形式,二叉树根的右子树又可以看做是一个由除第一棵树外的森林转换后的二叉树,应用同样的方法,直到最后产生一棵没有右子树的二叉树为止,这样就得到了原森林,如图 4-16 所示。二叉树转换为树的规则与此类似,二叉树转换为树或森林是唯一的。

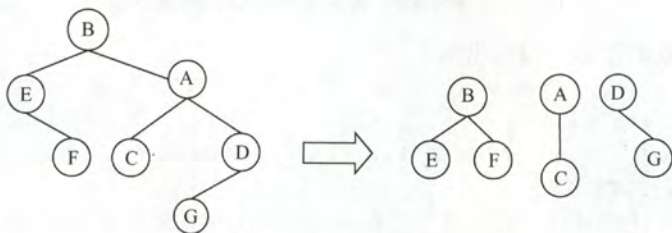


图 4-16 二叉树转换为森林

树转换成二叉树的画法:①在兄弟结点之间加一连线;②对每一个结点,只保留它与第一个子结点的连线,与其他子结点的连线全部抹掉;③以树根为轴心,顺时针旋转 45° 。

森林转换成二叉树的画法:①将每棵树的根相连;②将森林中的每棵树转换成相应的二叉树;③以第一棵树的根为轴心顺时针旋转 45° (结合 综合应用题 第 2 题模拟)。

4.4.3 树和森林的遍历

树的遍历操作是以某种方式访问树中每一个结点,且仅访问一次。树的遍历操作主要有先根遍历和后根遍历。

1) 先根遍历:若树非空,则先访问根结点,再按从左到右的顺序遍历根结点的每一棵子树。

其访问顺序与这棵树相应二叉树的先序遍历顺序相同。

2) 后根遍历: 若树非空, 则按从左到右的顺序遍历根结点的每一棵子树, 之后再访问根结点。其访问顺序与这棵树相应二叉树的中序遍历顺序相同 (结合图 4-15 自行模拟)。

另外, 树也有层次遍历, 与二叉树的层次遍历思想基本相同, 即按层序依次访问各结点。按照森林和树相互递归的定义, 可得到森林的两种遍历方法。

1) 先序遍历森林。若森林为非空, 则按如下规则进行遍历:

- 访问森林中第一棵树的根结点。
- 先序遍历第一棵树中根结点的子树森林。
- 先序遍历除去第一棵树之后剩余的树构成的森林。

2) 中序遍历森林。若森林为非空, 则按如下规则进行遍历:

- 中序遍历森林中第一棵树的根结点的子树森林。
- 访问第一棵树的根结点。
- 中序遍历除去第一棵树之后剩余的树构成的森林。

表 4-1 树和森林的遍历与
二叉树遍历的对应关系

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

树和森林的遍历: 可采用对应二叉树的遍历算法来实现, 见表 4-1。

4.4.4 树的应用——并查集

并查集是一种简单的集合表示, 它支持以下 3 种操作:

1) Union (S, Root1, Root2): 把集合 S 中的子集合 Root2 并入子集合 Root1 中。要求 Root1 和 Root2 互不相交, 否则不执行合并。

2) Find (S, x): 查找集合 S 中单元素 x 所在的子集合, 并返回该子集合的名字。

3) Initial (S): 将集合 S 中每一个元素都初始化为只有一个单元元素的子集合。

通常用树 (森林) 的双亲表示作为并查集的存储结构, 每个子集合以一棵树表示。所有表示子集合的树, 构成表示全集合的森林, 存放在双亲表示数组内。通常用数组元素的下标代表元素名, 根结点的下标代表子集合名, 根结点的双亲结点为负数。

例如, 若设有一个全集合为 $S=\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, 初始化时每个元素自成一个单元元素子集合, 每个子集合的数组值为 -1, 如图 4-17 所示。

经过一段时间的计算, 这些子集合合并成 3 个更大的子集合: $S_1=\{0, 6, 7, 8\}$, $S_2=\{1, 4, 9\}$, $S_3=\{2, 3, 5\}$, 此时并查集的树形表示和存储结构如图 4-18 所示。

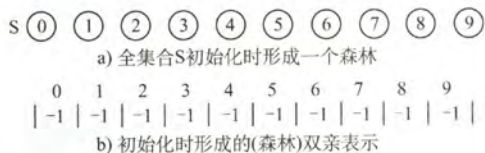


图 4-17 并查集的初始化

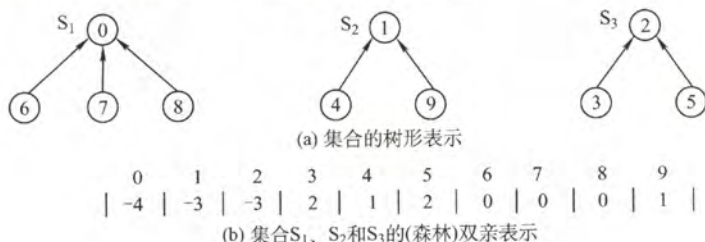


图 4-18 用树表示并查集

为了得到两个子集合的并, 只要将其中一个子集合根结点的双亲指针指向另一个集合的根结

点即可。因此, $S_1 \cup S_2$ 可以具有如图 4-19 所示的表示。

在采用树的双亲指针数组表示作为并查集的存储表示时, 集合元素的编号从 0 到 $\text{size}-1$ 。其中 size 是最大元素的个数。下面是并查集主要运算的实现。

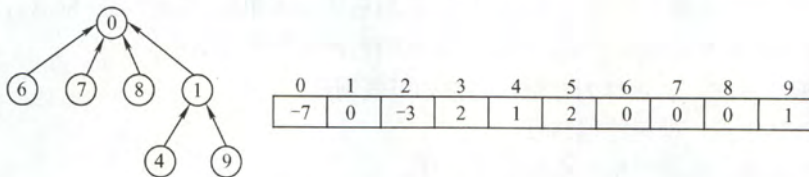


图 4-19 $S_1 \cup S_2$ 可能的表示方法

并查集的结构定义如下:

```
#define SIZE 100
int    UFsets[SIZE];           //集合元素数组(双亲指针数组)
```

并查集的初始化操作(S 即为并查集)。

```
void Initial(int S[]){
    for(int i=0;i<size;i++)    //每个自成单元素集合
        S[i]=-1;
}
```

Find 操作(函数在并查集 S 中查找并返回包含元素 x 的树的根)。

```
int Find(int S[],int x){
    while(S[x]>=0)            //循环寻找 x 的根
        x=S[x];
    return x;                 //根的 S[] 小于 0
}
```

Union 操作(函数求两个不相交子集的并集)。

```
void Union(int S[],int Root1,int Root2){
    //要求 Root1 与 Root2 是不同的, 且表示子集合的名字
    S[Root2]=Root1;          //将根 Root2 连接到另一根 Root1 下面
}
```

4.4.5 本节试题精选

一、单项选择题

- 下列关于树的说法中, 正确的是()。
 - 对于有 n 个结点的二叉树, 其高度为 $\log_2 n$
 - 完全二叉树中, 若一个结点没有左孩子, 则它必是叶结点
 - 高度为 $h(h > 0)$ 的完全二叉树对应的森林所含的树的个数一定是 h
 - 一棵树中的叶子数一定等于与其对应的二叉树的叶子数

A. I 和 III B. IV C. I 和 II D. II

- 利用二叉链表存储森林, 则根结点的右指针是()。

A. 指向最左兄弟 B. 指向最右兄弟 C. 一定为空 D. 不一定为空

- 【2009 年计算机联考真题】

将森林转换为对应的二叉树, 若在二叉树中, 结点 u 是结点 v 的父结点的父结点, 则在原来的森林中, u 和 v 可能具有的关系是()。

I. 父子关系 II. 兄弟关系 III. u 的父结点与 v 的父结点是兄弟关系

A. 只有 II B. I 和 II C. I 和 III D. I、II 和 III

4. 设森林 F 中有 3 棵树, 第一、第二、第三棵树的结点个数分别为 M_1 、 M_2 和 M_3 。与森林 F 对应的二叉树根结点的右子树上的结点个数是 ()。

- A. M_1 B. M_1+M_2 C. M_3 D. M_2+M_3

5. 设森林 F 对应的二叉树为 B, 它有 m 个结点, B 的根为 p , p 的右子树结点数为 n , 森林 F 中第一棵树的结点个数是 ()。

- A. $m-n$ B. $m-n-1$ C. $n+1$ D. 条件不足, 无法确定

6. 森林 $T=(T_1, T_2, \dots, T_m)$ 转化为二叉树 BT 的过程为若 $m=0$, 则 BT 为空, 若 $m \neq 0$, 则 ()。

- A. 将中间子树 T_{mid} ($mid=(1+m)/2$) 的根作为 BT 的根; 将 $(T_1, T_2, \dots, T_{mid-1})$ 转换为 BT 的左子树; 将 (T_{mid+1}, \dots, T_m) 转换为 BT 的右子树
B. 将子树 T_1 的根作为 BT 的根; 将 T_1 的子树森林转换成 BT 的左子树; 将 (T_2, T_3, \dots, T_m) 转换成 BT 的右子树
C. 将子树 T_1 的根作为 BT 的根; 将 T_1 的左子树森林转换成 BT 的左子树; 将 T_1 的右子树森林转换为 BT 的右子树; 其他依此类推
D. 将森林 T 的根作为 BT 的根; 将 (T_1, T_2, \dots, T_m) 转化为该根下的结点, 得到一棵树, 然后将这棵树再转化为二叉树 BT

7. 设 F 是一个森林, B 是由 F 变换来的二叉树。若 F 中有 n 个非终端结点, 则 B 中右指针域为空的结点有 () 个。

- A. $n-1$ B. n C. $n+1$ D. $n+2$

8. 【2011 年计算机联考真题】

已知一棵有 2011 个结点的树, 其叶结点个数为 116, 该树对应的二叉树中无右孩子的结点个数是 ()。

- A. 115 B. 116 C. 1895 D. 1896

9. 【2014 年计算机联考真题】

将森林 F 转换为对应的二叉树 T, F 中叶结点的个数等于 ()。

- A. T 中叶结点的个数 B. T 中度为 1 的结点个数
C. T 中左孩子指针为空的结点个数 D. T 中右孩子指针为空的结点个数

10. 如果 T_1 是由有序树 T 转换而来的二叉树, 那么 T 中结点的后根序列就是 T_1 中结点的 () 序列。

- A. 先序 B. 中序 C. 后序 D. 层序

11. 某二叉树结点的中序序列为 BDAECF, 后序序列为 DBEFCA, 则该二叉树对应的森林包括 () 棵树。

- A. 1 B. 2 C. 3 D. 4

12. 【2016 年计算机联考真题】

若森林 F 有 15 条边、25 个结点, 则 F 包含树的个数是 ()。

- A. 8 B. 9 C. 10 D. 11

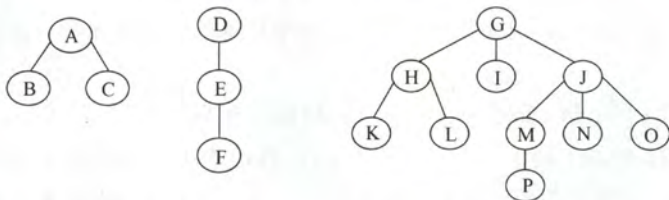
13. 设 X 是树 T 中的一个非根结点, B 是 T 所对应的二叉树。在 B 中, X 是其双亲结点的右孩子, 下列结论中正确的是 ()。

- A. 在树 T 中, X 是其双亲结点的第一个孩子
B. 在树 T 中, X 一定无右边兄弟
C. 在树 T 中, X 一定是叶子结点

D. 在树T中, X一定有左边兄弟

二、综合应用题

1. 给定一棵树的先根遍历序列和后根遍历序列, 能否唯一确定一棵树? 若能, 请举例说明; 若不能, 请给出反例。
2. 将下面一个由3棵树组成的森林转换为二叉树。



3. 已知某二叉树的先序序列和中序序列分别为 ABDEHCFIMGJKL 和 DBHEAIM FCGKLJ, 请画出这棵二叉树, 并画出二叉树对应的森林。

4. 【2016年计算机联考真题】

如果一棵非空 k ($k \geq 2$) 叉树 T 中每个非叶结点都有 k 个孩子, 则称 T 为正则 k 叉树。请回答下列问题并给出推导过程。

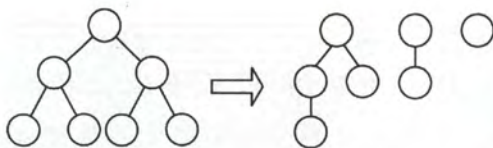
- (1) 若 T 有 m 个非叶结点, 则 T 中的叶结点有多少个?
- (2) 若 T 的高度为 h (单结点的树 $h=1$), 则 T 的结点数最多为多少个? 最少为多少个?
5. 编程求以孩子兄弟表示法存储的森林的叶子结点数。
6. 以孩子兄弟链表为存储结构, 请设计递归算法求树的深度。
7. 已知一棵树的层次序列以及每个结点的度, 编写一个算法构造此树的孩子-兄弟链表。

4.4.6 答案与解析

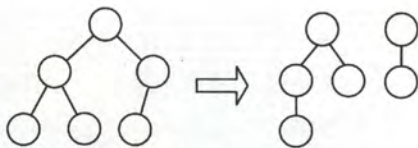
一、单项选择题

1. D

若 n 个结点的二叉树是一棵单支树, 其高度为 n 。完全二叉树中顶多存在一个度为 1 的结点且只有左孩子, 若不存在左孩子, 则一定也不存在右孩子, 因此必是叶结点, II 正确。只有满二叉树才具有 III 性质, 如下图所示:



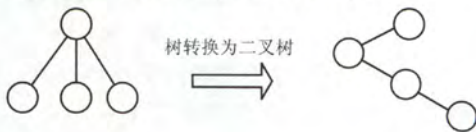
(a) 满二叉树转化为对应的森林



(b) 非满二叉树转化为对应的森林

在树转换为二叉树时, 若有几个叶子结点具有共同的双亲, 则转换成二叉树后只有一个叶子结点 (最右边的叶子结点), 如右图所示, IV 错误。

注意: 若树中的任两个叶子结点都不存在相同的双亲, 则树中的叶子数才有可能与其对应的二叉树中的叶子数相等。



2. D

森林与二叉树具有对应关系, 因此, 我们存储森林的时候应先将森林转换成二叉树, 转换的

方法就是“左孩子右兄弟”，与树不同的是，如果存在第二棵树，二叉链表的根结点的右指针指向的是森林中第二棵树的根结点。若此森林只有一棵树，那么根结点的右指针为空。因此，右指针可能为空也可能不为空。

3. B

森林与二叉树的转换规则为“左孩子右兄弟”。在最后生成的二叉树中，父子关系在对应的森林关系中可能是兄弟关系或原本就是父子关系。情形 I：若结点 v 是结点 u 的第二个孩子结点，在转换时，结点 v 就变成结点 u 第一个孩子的右孩子，符合要求。情形 II：结点 u 和 v 是兄弟结点的关系，但二者之中还有一个兄弟结点 k ，则转换后，结点 v 就变为结点 k 的右孩子，而结点 k 则是结点 u 的右孩子，符合要求。情形 III：结点 v 的父结点要么是原先的父结点或是兄弟结点。若结点 u 的父结点与 v 的父结点是兄弟关系，则转换之后，不可能出现结点 u 是结点 v 的父结点的父结点。

4. D

森林与二叉树的转换规则同样是“左孩子右兄弟”，不过与普通树不同，森林中每棵树是独立的，因此，先要将每棵树的根结点全部看成是兄弟结点的关系。所以，题中森林转换后，树 2 是作为树 1 根结点的右子树，树 3 是作为树 2 根结点的右子树。所以，森林 F 对应的二叉树根结点的右子树上的结点个数是 M_2+M_3 。

5. A

森林转换成二叉树时采用孩子兄弟表示法，根结点及其左子树为森林中的第一棵树。右子树为其他剩余的树。所以，第一棵树的结点个数为 $m-n$ 。

6. B

将森林中每棵树的根结点看成是兄弟结点的关系，再按照“左孩子右兄弟”的规则来进行转化。

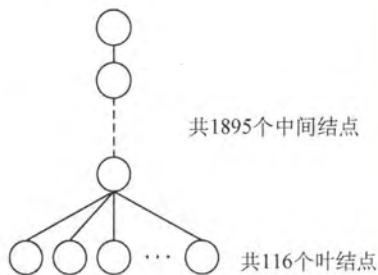
7. C

根据森林与二叉树转换规则“左孩子右兄弟”。二叉树 B 中右指针域为空代表该结点没有兄弟结点。森林中每棵树的根结点从第二个开始依次连接在前一棵树根的右孩子上，则最后一棵树根结点的右指针为空。另外，每一个非终端结点，它的所有孩子结点在转换之后，最后一个孩子的右指针也为空，故树 B 中右指针域为空的结点有 $n+1$ 个。

8. D

树转换为二叉树时，树中每一个分支结点的所有子结点中的最右子结点无右孩子，根结点转换后也没有右孩子，因此，对应的二叉树中无右孩子的结点个数=分支结点数+1=2011-116+1=1896。

通常本题应采用特殊法解，设题意中的树是如图所示的结构，则对应的二叉树中仅有前 115 个叶结点有右孩子，故无右孩子的结点个数=2011-115=1896。



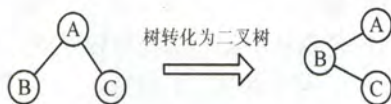
第 8 题图

9. C

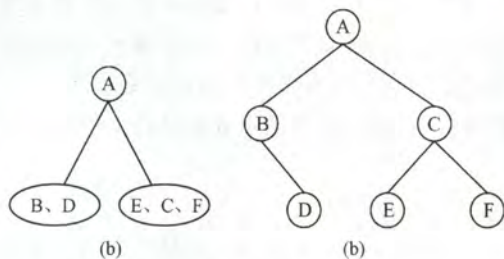
将森林转化为二叉树即相当于用孩子兄弟表示法表示森林。在变化过程中，原森林某结点的第一个孩子结点作为它的左子树，它的兄弟作为它的右子树。那么森林中的叶结点由于没有孩子结点，那么转化为二叉树时，该结点就没有左结点，所以 F 中叶结点的个数就等于 T 中左孩子指针为空的结点个数，选 C 。此题还可以通过一些特例来排除 A 、 B 、 D 选项。

10. B

有序树 T 转换成二叉树 T_1 时, T 的后根序列是对应 T_1 的中序序列还是后序序列呢(显然树的后根序列不可能对应二叉树的先序序列和层序序列)? 看第 10 题图所示的例子, 在树 T 中, 叶子结点 B 应最先访问, 在 T_1 中, B 的右兄弟 C 转换为它的右孩子, 如果对应 T_1 的后序序列, 则 C 应在 B 的前面访问, 所以 T 的后根序列不可能对应 T_1 的后序序列。



第 10 题图



第 11 题图

11. C

根据二叉树的前序遍历序列和中序遍历序列可以唯一确定一棵二叉树。

根据后序遍历序列, A 是二叉树的根结点。根据中序遍历序列, 则二叉树的形态一定如图第 11 题 (a) 所示。

对于 A 的左子树, 由后序遍历序列可知, 因为 B 比 D 后被访问, 因此, B 必为 D 的父结点, 又由中序遍历序列可知, D 是 B 的右儿子。对于

A 的右子树, 同理可确定结点 E 、 C 、 F 的关系。此二叉树的形态如图第 11 题 (b) 所示。

再根据二叉树与森林的对应关系。森林中树的棵数即为其对应二叉树中根结点 A 的兄弟数。可知此森林中有 3 棵树, 根结点分别为 A 、 C 和 F 。

12. C

解法一: 树有一个很重要的性质: 在 n 个结点的树中有 $n-1$ 条边, “那么对于每棵树, 其结点数比边数多 1”。题中的森林中的结点数比边数多 10 (即 $25-15=10$), 显然共有 10 棵树。

解法二: 若考生再仔细分析发现, 此题也是考察图的某些方面的性质: 生成树和生成森林。此时对于图的生成树有一个重要的性质: 若图中顶点数为 n , 则它的生成树含有 $n-1$ 条边。对比解法一中树的性质, 不难发现两种解法都利用到了“树中结点数比边数多 1”的性质, 接下来的分析如解法一。

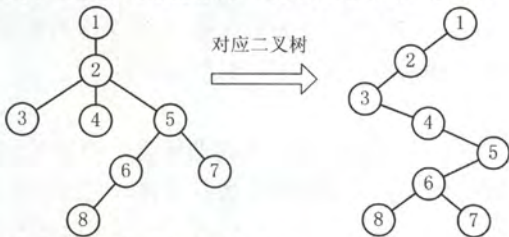
13. D

在二叉树 B 中, X 是其双亲的右孩子, 因此在树 T 中, X 必是其双亲结点的右兄弟, 换句话说, X 在树中必有左兄弟。

二、综合应用题

1. 解答:

一棵树的先根遍历结果与其对应二叉树的先序遍历结果相同, 树的后根遍历结果与其对应二叉树表示的中序遍历结果相同。由于二叉树的先序序列和中序序列能够唯一地确定这棵二叉树, 因此, 根据题目给出的条件, 利用树的先根遍历序列和后根遍历序列能够唯一地确定这棵树。例如, 对于下图所示的树, 对应二叉树的先序序列为 1, 2, 3, 4, 5, 6, 8, 7, 中序序列为 3, 4, 8, 6, 7, 5, 2, 1。原树的先根遍历序列为 1, 2, 3, 4, 5, 6, 8, 7, 后根遍历序列为 3, 4, 8, 6, 7, 5, 2, 1。

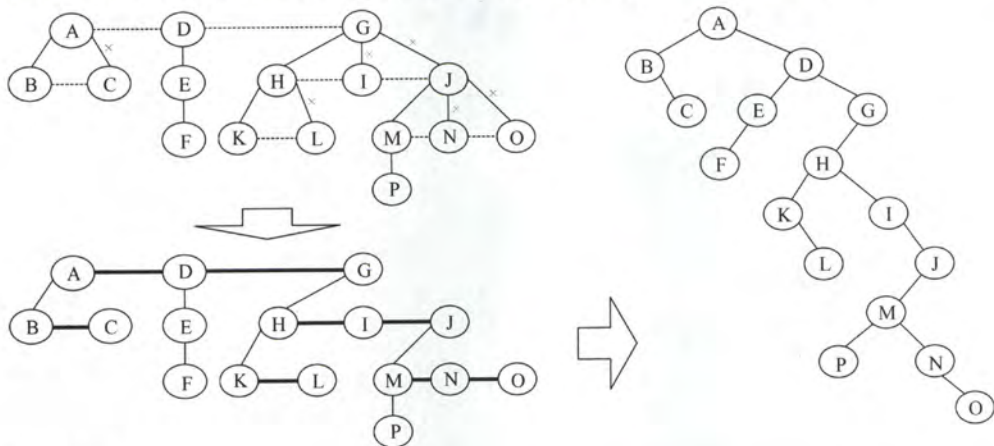


注意: 树的先根遍历、后根遍历与对应二叉树前序遍历、中序遍历的对应。

2. 解答:

根据树与二叉树“左孩子右兄弟”的转换规则, 将森林转换为二叉树的过程: ①将每棵树的

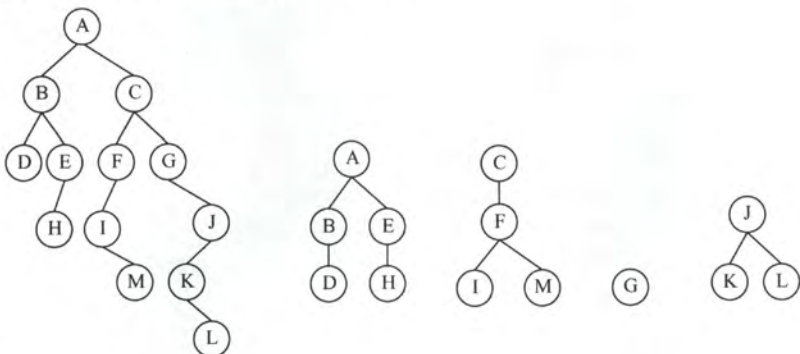
根结点也看成是兄弟关系,在兄弟结点之间加一连线。②对每一个结点,只保留它与第一个子结点的连线,与其他子结点的连线全部抹掉。③以树根为轴心,顺时针旋转 45° 。



3. 解答:

知道二叉树的先序和中序遍历后可以唯一确定这棵树的结构。然后把二叉树转换到树和森林的方式是,若结点 x 是双亲 y 的左孩子,则把 x 的右孩子,右孩子的右孩子……都与 y 用连线连起来,最后去掉所有双亲到右孩子的连线。

最后得到的二叉树及对应的森林如下图所示。



4. 解答:

1) 根据定义,正则 k 叉树中仅含有两类结点:叶结点(个数记为 n_0)和度为 k 的分支结点(个数记为 n_1)。树 T 中的结点总数 $n=n_0+n_k=n_0+m$ 。树中所含的边数 $e=n-1$, 这些边均为 m 个度为 k 的结点发出的,即 $e=m \times k$ 。整理得: $n_0+m=m \times k+1$, 故 $n_0=(k-1) \times m+1$ 。(3分)

2) 高度为 h 的正则 k 叉树 T 中,含最多结点的树形为:除第 h 层外,第 1 到第 $h-1$ 层的结点都是度为 k 的分支结点;而第 h 层均为叶结点,即树是“满”树。此时第 j ($1 \leq j \leq h$) 层结点数为 k^{j-1} , 结点总数 M_1 为:

$$M_1 = \sum_{j=1}^h k^{j-1} = \frac{k^h - 1}{k - 1}$$

含最少结点的正则 k 叉树的树形为:第 1 层只有根结点,第 2 到第 $h-1$ 层仅含 1 个分支结点和 $k-1$ 个叶结点,第 h 层有 k 个叶结点。即除根外第 2 到第 h 层中每层的结点数均为 k , 故 T 中所含结点总数 M_2 为:

$$M_2 = 1 + (h-1) \times k$$

5. 解答:

当森林(树)以孩子兄弟表示法存储时,若结点没有孩子($fch=null$),则它必是叶子,总的叶子结点个数是孩子子树(fch)上的叶子数和兄弟子树($nsib$)上叶结点个数之和。

算法的代码:

```
typedef struct node
{
    ElemType    data;           //数据域
    struct node *fch,*nsib;     //孩子与兄弟域
}*Tree;
int  Leaves(Tree t){           //计算以孩子兄弟表示法存储的森林的叶子数
    if(t==NULL)                //树空返回0
        return 0;
    if(t->fch==NULL)            //若结点无孩子,则该结点必是叶子
        return 1+Leaves(t->nsib); //返回叶子结点和其兄弟子树中的叶子结点数
    else                        //孩子子树和兄弟子树中叶子数之和
        return  Leaves(t->fch)+Leaves(t->nsib);
}
```

6. 解答:

由孩子兄弟链表表示的树,求高度的算法思想:采用递归算法,若树为空,高度为零;否则,高度为第一子女树高度加1和兄弟子树高度的大者。其非递归算法使用队列,逐层遍历树,取得树的高度。

```
int  Height(CSTree bt){
    //递归求以孩子兄弟链表表示的树的深度
    int hc,hs;
    if(bt==NULL)
        return 0;
    else{//否则,高度取子女高度+1和兄弟子树高度的大者
        hc=height(bt->firstchild); //第一子女树高
        hs=height(bt->nextsibling); //兄弟树高
        if(hc+1>hs)
            return hc+1;
        else
            return hs;
    }
}
```

7. 解答:

本题与树的层次序有关。可设立一个辅助数组 $pointer[]$, 存储新建树的各结点的地址, 再根据层次序列与每个结点的度, 逐个链接结点。算法的描述如下:

```
#define maxNodes 15
void createCSTree_Degree(CSTree&T,DataType e[],int degree[],int n){
    //根据树结点的层次序列 e[] 和各结点的度 degree[] 构造树的孩子-兄弟链表
    //参数 n 是树结点个数
    CSNode *pointer=new CSNode[maxNodes]; //判断 pointer[i] 为空的语句未写
    int i,j,d,k=0;
    for(i=0;i<n;i++){//初始化
        pointer[i]=new CSNode;           //判断 pointer[i] 为空的语句未写
        pointer[i]->data=e[i];
    }
```



```

        pointer[i]->lchild=pointer[i]->rsibling=NULL;
    }
    for(i=0;i<n;i++){
        d=degree[i];           //结点 i 的度数
        if(d){
            k++;                //k 为子女结点序号
            pointer[i]->lchild=pointer[k]; //建立 i 与子女 k 间的链接
            for(j=2;j<=d;j++){
                pointer[j-1]->rsibling=pointer[j];
            }
        }
        T=pointer[0];
        delete [] pointer;
    }
}

```

4.5 树与二叉树的应用

4.5.1 二叉排序树

1. 二叉排序树的定义

二叉排序树（简称 BST），也称为二叉查找树。二叉排序树或者是一棵空树，或者是一棵具有下列特性的非空二叉树：

- 1) 若左子树非空，则左子树上所有结点关键字值均小于根结点的关键字值。
- 2) 若右子树非空，则右子树上所有结点关键字值均大于根结点的关键字值。
- 3) 左、右子树本身也分别是一棵二叉排序树。

由此定义可知，二叉排序树是一个递归的数据结构，可以方便地使用递归算法对二叉排序树进行各种运算。图 4-20 所示为一棵二叉排序树。

根据二叉排序树的定义，有左子树结点值<根结点值<右子树结点值，所以，对二叉排序树进行中序遍历，可以得到一个递增的有序序列。例如，图 4-20 的二叉排序树的中序遍历序列为 123468。

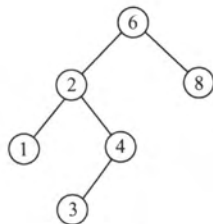


图 4-20 一棵二叉排序树

2. 二叉排序树的查找

二叉排序树的查找是从根结点开始，沿某一个分支逐层向下进行比较的过程。若二叉排序树非空，将给定值与根结点的关键字比较，若相等，则查找成功；若不等，则当根结点的关键字大于给定关键字值时，在根结点的左子树中查找，否则在根结点的右子树中查找。这显然是一个递归的过程。

二叉排序树的非递归查找算法：

```

BSTNode *BST_Search(BiTree T, ElemType key, BSTNode *&p){
    //查找函数返回指向关键字值为 key 的结点指针，若不存在，返回 NULL
    p=NULL; //p 指向被查找结点的双亲，用于插入和删除操作中
    while(T!=NULL&&key!=T->data){
        p=T;
        if(key<T->data) T=T->lchild;
        else T=T->rchild;
    }
}

```



```

    }
    return T;
}

```

例如,在图 4-20 中查找值为 4 的结点。首先 4 与根结点 6 比较。因为 4 小于 6,所以在根结点 6 的左子树中继续查找。因为 4 大于 2,所以在结点 2 的右子树中查找,查找成功。

同样,二叉排序树的查找也可以用递归算法实现,递归算法比较简单,但执行效率较低。具体的实现,留给读者思考。

3. 二叉排序树的插入

二叉排序树作为一种动态集合,其特点是树的结构通常不是一次生成的,而是在查找过程中,当树中不存在关键字等于给定值的结点时再进行插入。

由于二叉排序树是递归定义的,插入结点的过程是,若原二叉排序树为空,则直接插入结点;否则,若关键字 k 小于根结点关键字,则插入到左子树中,若关键字 k 大于根结点关键字,则插入到右子树中。

```

int BST_Insert(BiTree &T,KeyType k){
//在二叉排序树 T 中插入一个关键字为 k 的结点
    if(T==NULL){                //原树为空,新插入的记录为根结点
        T=(BiTree)malloc(sizeof(BSTNode));
        T->key=k;
        T->lchild=T->rchild=NULL;
        return 1;                //返回 1,表示成功
    }
    else if(k==T->key)            //树中存在相同关键字的结点
        return 0;
    else if(k<T->key)            //插入到 T 的左子树中
        return BST_Insert(T->lchild,k);
    else                          //插入到 T 的右子树中
        return BST_Insert(T->rchild,k);
}

```

由此可见,插入的新结点一定是某个叶结点。如图 4-21 所示,在一个二叉排序树先后依次插入结点 28 和结点 58,虚线表示的边是其查找的路径。

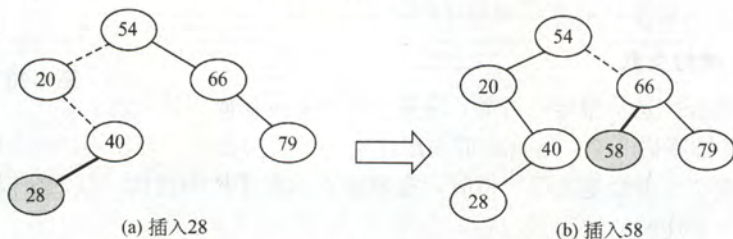


图 4-21 向二叉排序树中插入结点

4. 二叉排序树的构造

构造一棵二叉排序树就是依次输入数据元素,并将它们插入到二叉排序树中的适当位置上的过程。具体过程是,每读入一个元素,就建立一个新结点,若二叉排序树非空,则将新结点的值与根结点的值比较,如果小于根结点的值,则插入到左子树中,否则插入到右子树中;若二叉排序树为空,则新结点作为二叉排序树的根结点。

```

void Creat_BST(BiTree &T,KeyType str[],int n){

```

```

//用关键字数组 str[] 建立一个二叉排序树
T=NULL;           //初始时 bt 为空树
int i=0;
while(i<n){        //依次将每个元素插入
    BST_Insert(T,str[i]);
    i++;
}
}

```

5. 二叉排序树的删除

在二叉排序树中删除一个结点时,不能把以该结点为根的子树上的结点都删除,必须先把被删除结点从存储二叉排序树的链表上摘下,将因删除结点而断开的二叉链表重新链接起来,同时确保二叉排序树的性质不会丢失。

删除操作的实现过程按 3 种情况来处理:

- ① 如果被删除结点 z 是叶结点,则直接删除,不会破坏二叉排序树的性质。
- ② 若结点 z 只有一棵左子树或右子树,则让 z 的子树成为 z 父结点的子树,替代 z 的位置。
- ③ 若结点 z 有左、右两棵子树,则令 z 的直接后继(或直接前驱)替代 z ,然后从二叉排序树中删去这个直接后继(或直接前驱),这样就转换成了第一或第二种情况。如图 4-22 所示,在 3 种情况下分别删除结点 45、78、78。

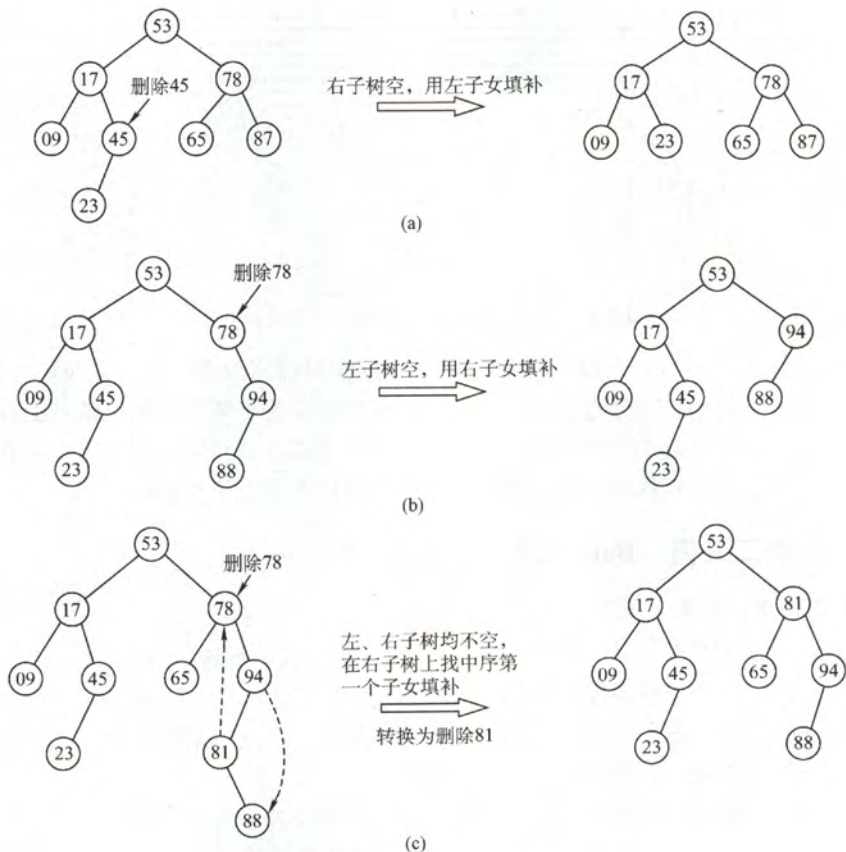


图 4-22 3 种情况下的删除过程

思考：若在二叉排序树中删除并插入某结点，得到的二叉排序树是否和原来的相同？

6. 二叉排序树的查找效率分析

对于高度为 H 的二叉排序树，其插入和删除操作的运行时间都是 $O(H)$ 。但在最坏的情况下，即构造二叉排序树的输入序列是有序的，则会形成一个倾斜的单支树，此时二叉排序树的性能显著变坏，树的高度也增加为元素个数 N ，如图 4-23 所示。

在等概率情况下，图 4-23 (a) 的查找成功的平均查找长度为

$$ASL_a = (1+2 \times 2 + 3 \times 4 + 4 \times 3) / 10 = 2.9$$

而图 4-23 (b) 的查找成功的平均查找长度为

$$ASL_b = (1+2+3+4+5+6+7+8+9+10) / 10 = 5.5$$

由上可知，二叉排序树查找算法的平均查找长度，主要取决于树的高度，即与二叉树的形态有关。如果二叉排序树是一个只有右（左）孩子的单支树（类似于有序的单链表），其平均查找长度和单链表相同，为 $O(n)$ 。如果二叉排序树的左、右子树的高度之差的绝对值不超过 1，这样的二叉排序树称为平衡二叉树。它的平均查找长度达到 $O(\log_2 n)$ 。

从查找过程看，二叉排序树与二分查找相似。就平均时间性能而言，二叉排序树上的查找和二分查找差不多。但二分查找的判定树唯一，而二叉排序树不唯一，相同的关键字其插入顺序不同可能生成不同的二叉排序树，如图 4-23 所示。

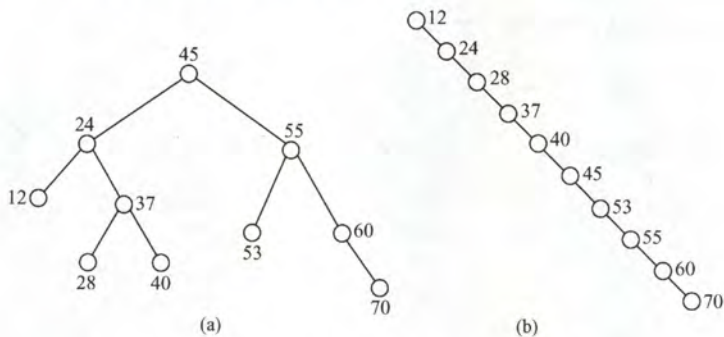


图 4-23 相同关键字组成的不同二叉排序树

就维护表的有序性而言，二叉排序树无须移动结点，只需修改指针即可完成插入和删除操作，平均执行时间为 $O(\log_2 n)$ 。二分查找的对象是有序顺序表，若有插入和删除结点的操作，所花的代价是 $O(n)$ 。当有序表是静态查找表时，宜用顺序表作为其存储结构，而采用二分查找实现其查找操作；若有序表是动态查找表，则应选择二叉排序树作为其逻辑结构。

4.5.2 平衡二叉树 (Balanced Binary Tree)

1. 平衡二叉树的定义

为了避免树的高度增长过快，降低二叉排序树的性能，我们规定在插入和删除二叉树结点时，要保证任意结点的左、右子树高度差的绝对值不超过 1，将这样的二叉树称为平衡二叉树，简称平衡树 (AVL 树)。定义结点左子树与右子树的高度差为该结点的平衡因子，则平衡二叉树结点的平衡因子的值只可能是 -1、0 或 1。

因此，平衡二叉树可定义为它或者是一棵空树，或者是具有下列性质的二叉树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的高度差的绝对值不超过 1。如图 4-24 (a) 所示是平衡二叉树，图 4-24 (b) 所示是不平衡的二叉树。结点中的值为该结点的平衡因子。

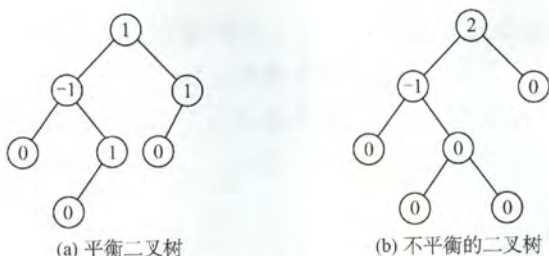


图 4-24 平衡二叉树和不平衡的二叉树

2. 平衡二叉树的插入

二叉排序树保证平衡的基本思想：每当在二叉排序树中插入（或删除）一个结点时，首先要检查其插入路径上的结点是否因为此次操作而导致了不平衡。如果导致了不平衡，则先找到插入路径上离插入结点最近的平衡因子绝对值大于 1 的结点 A，再对以 A 为根的子树，在保持二叉排序树特性的前提下，调整各结点的位置关系，使之重新达到平衡。

注意：每次调整的对象都是最小不平衡子树，即在插入路径上离插入结点最近的平衡因子的绝对值大于 1 的结点作为根的子树。如图 4-25 所示虚线框内为最小不平衡子树。

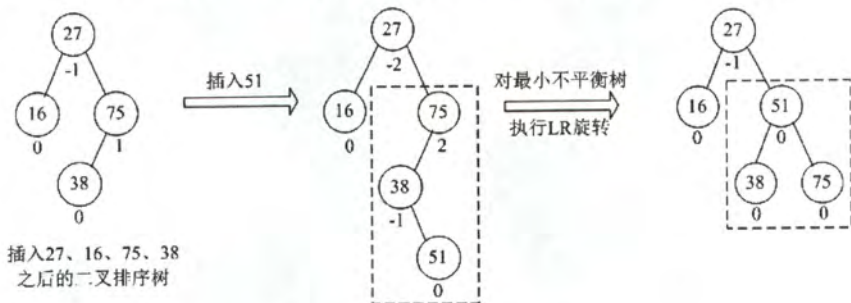


图 4-25 最小不平衡子树示意

平衡二叉树的插入过程前半部分与二叉排序树相同，但是在新结点插入后，如果造成了查找路径上某个结点不再平衡，需要做出相应的调整。一般可将失去平衡后进行调整的规律归纳为下列 4 种情况：

1) LL 平衡旋转（右单旋转）。由于在结点 A 的左孩子（L）的左子树（L）上插入了新结点，A 的平衡因子由 1 增至 2，导致以 A 为根的子树失去平衡，需要一次向右的旋转操作。将 A 的左孩子 B 向右上旋转代替 A 成为根结点，将 A 结点向右上旋转成为 B 的右子树的根结点，而 B 的原右子树则作为 A 结点的左子树。

如图 4-26 所示，结点旁的数值代表结点的平衡因子，而用方块表示相应结点的子树，下方数值代表该子树的高度。

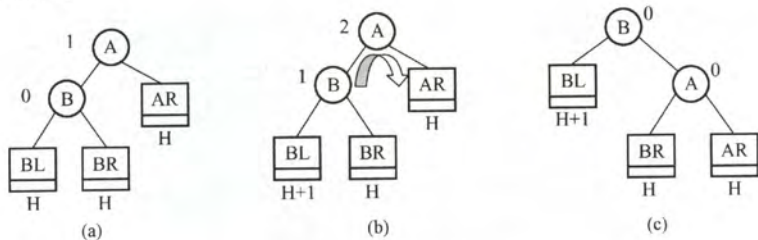


图 4-26 LL 平衡旋转

2) RR 平衡旋转(左单旋转)。由于在结点 A 的右孩子(R)的右子树(R)上插入了新结点, A 的平衡因子由-1 减至-2, 导致以 A 为根的子树失去平衡, 需要一次向左的旋转操作。将 A 的右孩子 B 向左上旋转代替 A 成为根结点, 将 A 结点向左下旋转成为 B 的左子树的根结点, 而 B 的原左子树则作为 A 结点的右子树, 如图 4-27 所示。

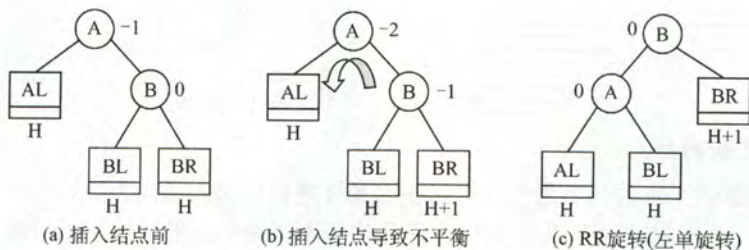


图 4-27 RR 平衡旋转

3) LR 平衡旋转(先左后右双旋转)。由于在 A 的左孩子(L)的右子树(R)上插入新结点, A 的平衡因子由 1 增至 2, 导致以 A 为根的子树失去平衡, 需要进行两次旋转操作, 先左旋转后右旋转。先将 A 结点的左孩子 B 的右子树的根结点 C 向左上旋转提升到 B 结点的位置, 然后再把该 C 结点向右上旋转提升到 A 结点的位置, 如图 4-28 所示。

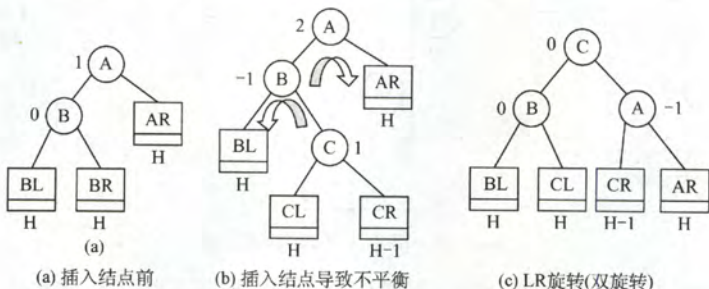


图 4-28 LR 平衡旋转

4) RL 平衡旋转(先右后左双旋转)。由于在 A 的右孩子(R)的左子树(L)上插入新结点, A 的平衡因子由-1 减至-2, 导致以 A 为根的子树失去平衡, 需要进行两次旋转操作, 先右旋转后左旋转。先将 A 结点的右孩子 B 的左子树的根结点 C 向右上旋转提升到 B 结点的位置, 然后再把该 C 结点向左上旋转提升到 A 结点的位置, 如图 4-29 所示。

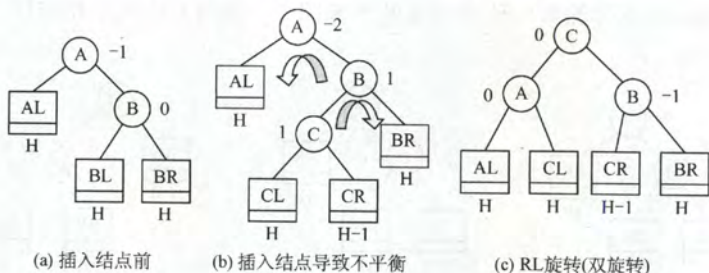


图 4-29 RL 平衡旋转

注意: LR 和 RL 旋转时, 究竟新结点插入在 C 的左子树还是右子树上, 不影响旋转过程,

而图 4-28 和图 4-29 中以插入 C 的左子树中为例。

3. 平衡二叉树的查找

在平衡二叉树上进行查找的过程和二叉排序树相同, 因此, 在查找的过程中和给定值进行比较的关键字个数不超过树的深度。假设以 N_h 表示深度为 h 的平衡树中含有的最少结点数。显然, $N_0=0$, $N_1=1$, $N_2=2$, 并且有 $N_h=N_{h-1}+N_{h-2}+1$ 。可以证明, 含有 n 个结点平衡二叉树的最大深度为 $O(\log_2 n)$, 因此, 平衡二叉树的平均查找长度为 $O(\log_2 n)$ 。如图 4-30 所示。

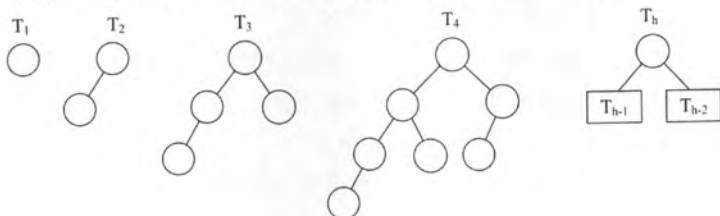


图 4-30 结点个数 n 最少的平衡二叉树

注意: 该结论可用于求解给定结点数的平衡二叉树的查找所需的最多比较次数 (或树的最大高度)。如, 问含有 12 个结点的平衡二叉树中查找某个结点的最多比较次数是?

4.5.3 哈夫曼 (Huffman) 树和哈夫曼编码

1. 哈夫曼树的定义

在许多实际应用中, 树中结点常常被赋予一个表示某种意义的数值, 称为该结点的权。从树根结点到任意结点的路径长度 (经过的边数) 与该结点上权值的乘积称为该结点的带权路径长度。树中所有叶结点的带权路径长度之和称为该树的带权路径长度, 记为

$$WPL = \sum_{i=1}^n w_i \times l_i$$

式中, w_i 是第 i 个叶结点所带的权值; l_i 是该叶结点到根结点的路径长度。

在含有 N 个带权叶子结点的二叉树中, 其中带权路径长度 (WPL) 最小的二叉树称为哈夫曼树, 也称为最优二叉树。

例如, 图 4-31 中的 3 棵二叉树, 都有 4 个叶子结点 a 、 b 、 c 、 d , 分别带权 7、5、2、4, 它们的带权路径长度分别为

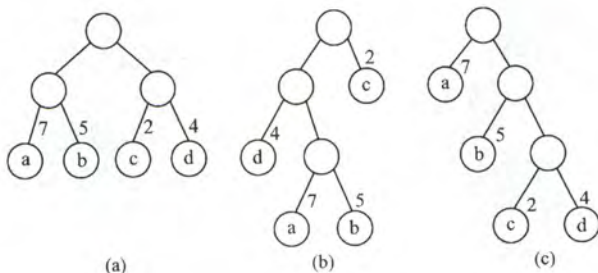


图 4-31 具有不同带权长度的二叉树

(a) $WPL=7 \times 2 + 5 \times 2 + 2 \times 2 + 4 \times 2 = 36$

(b) $WPL=7 \times 3 + 5 \times 3 + 2 \times 1 + 4 \times 2 = 46$

(c) $WPL=7 \times 1 + 5 \times 2 + 2 \times 3 + 4 \times 3 = 35$

其中图 4-31 (c) 中树的 WPL 最小。可以验证, 它恰为哈夫曼树。

2. 哈夫曼树的构造

给定 N 个权值分别为 w_1, w_2, \dots, w_N 的结点。通过哈夫曼算法可以构造出最优二叉树, 算法的描述如下:

- 1) 将这 N 个结点分别作为 N 棵仅含一个结点的二叉树, 构成森林 F 。
- 2) 构造一个新结点, 并从 F 中选取两棵根结点权值最小的树作为新结点的左、右子树, 并且将新结点的权值置为左、右子树上根结点的权值之和。
- 3) 从 F 中删除刚才选出的两棵树, 同时将新得到的树加入 F 中。
- 4) 重复步骤 2) 和 3), 直至 F 中只剩下一棵树为止。

从上述构造过程中可以看出哈夫曼树具有如下特点:

- 1) 每个初始结点最终都成为叶结点, 并且权值越小的结点到根结点的路径长度越大。
- 2) 构造过程中共新建了 $N-1$ 个结点 (双分支结点), 因此哈夫曼树中结点总数为 $2N-1$ 。
- 3) 每次构造都选择 2 棵树作为新结点的孩子, 因此哈夫曼树中不存在度为 1 的结点。

例如, 权值 $\{7, 5, 2, 4\}$ 的哈夫曼树的构造过程如图 4-32 所示。

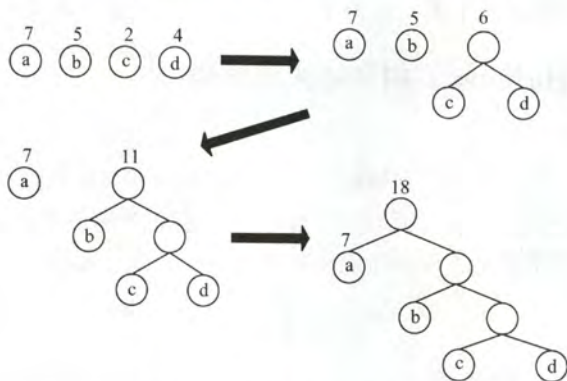


图 4-32 哈夫曼树的构造过程

3. 哈夫曼编码

对于待处理的一个字符串序列, 如果对每个字符用同样长度的二进制位来表示, 则称这种编码方式为固定长度编码。若允许对不同字符用不等长的二进制位表示, 则这种方式称为可变长度编码。可变长度编码比固定长度编码好得多, 其特点是对频率高的字符赋以短编码, 而对频率较低的字符则赋以较长一些的编码, 从而可以使字符平均编码长度减短, 起到压缩数据的效果。哈夫曼编码是一种被广泛应用而且非常有效的数据压缩编码。

如果没有一个编码是另一个编码的前缀, 则称这样的编码为前缀编码。如 0、101 和 100 是前缀编码。对前缀编码的解码也是很简单的, 因为没有一个码是其他码的前缀。所以, 可以识别出第一个编码, 将它翻译为原码, 再对余下的编码文件重复同样的解码操作。如 00101100 可被唯一地分析为 0、0、101 和 100。

由哈夫曼树得到哈夫曼编码是很自然的过程, 首先, 将每个出现的字符当做一个独立的结点, 其权值为它出现的频度 (或次数), 构造出对应的哈夫曼树。显然, 所有字符结点都出现在叶结点中。我们可以将字符的编码解释为从根至该字符的路径上边标记的序列, 其中边标记为 0 表示“转向左孩子”, 标记为 1 表示“转向右孩子”。图 4-33 所示为一个由哈夫曼树构造哈夫曼编码的示例, 矩形方块表示字符及其出现的次数。

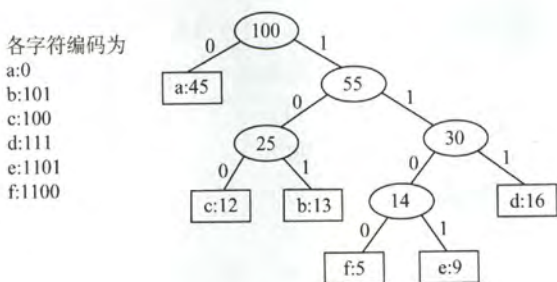


图 4-33 由哈夫曼树构造哈夫曼编码

这棵哈夫曼树的 WPL 为

$$\text{WPL} = 1 \times 45 + 3 \times (13 + 12 + 16) + 4 \times (5 + 9) = 224$$

此处的 WPL 可以看成是最终编码得到二进制编码的长度, 共 224 位。如果采用 3 位固定长度编码, 则得到的二进制编码长度为 300 位。哈夫曼编码共压缩了 25% 的数据。利用哈夫曼树可以设计出总长度最短的二进制前缀编码。

注意: 究竟 0 和 1 表示左子树还是右子树没有明确规定。因此, 左、右结点的顺序是任意的, 所以构造出的哈夫曼树并不唯一, 但是各哈夫曼树的带权路径长度相同且为最优。

4.5.4 本节试题精选

一、单项选择题

- 对于二叉排序树, 下面的说法 () 是正确的。
 - 二叉排序树是动态树表, 查找失败时插入新结点时, 会引起树的重新分裂和组合
 - 对二叉排序树进行层序遍历可得到有序序列
 - 用逐点插入法构造二叉排序树, 若先后插入的关键字有序, 二叉排序树的深度最大
 - 在二叉排序树中进行查找, 关键字的比较次数不超过结点数的 $1/2$
- 按 () 遍历二叉排序树得到的序列是一个有序序列。
 - 先序
 - 中序
 - 后序
 - 层次
- 在二叉排序树中进行查找的效率与 () 有关。
 - 二叉排序树的深度
 - 二叉排序树的结点的个数
 - 被查找结点的度
 - 二叉排序树的存储结构
- 在常用的描述二叉排序树的存储结构中, 关键字值最大的结点 ()。
 - 左指针一定为空
 - 右指针一定为空
 - 左右指针均为空
 - 左右指针均不为空
- 设二叉排序树中关键字由 1 到 1000 的整数构成, 现要查找关键字为 363 的结点, 下述关键字序列中, 不可能是在二叉排序树上查找的序列是 ()。
 - 2, 252, 401, 398, 330, 344, 397, 363
 - 924, 220, 911, 244, 898, 258, 362, 363
 - 925, 202, 911, 240, 912, 245, 363
 - 2, 399, 387, 219, 266, 382, 381, 278, 363
- 【2011 年计算机联考真题】

对于下列关键字序列, 不可能构成某二叉排序树中一条查找路径的序列是 ()。

 - 95, 22, 91, 24, 94, 71
 - 92, 20, 91, 34, 88, 35

C. 21, 89, 77, 29, 36, 38

D. 12, 25, 71, 68, 33, 34

7. 分别以下列序列构造二叉排序树, 与用其他3个序列所构造的结果不同的是 ()。

A. (100, 80, 90, 60, 120, 110, 130)

B. (100, 120, 110, 130, 80, 60, 90)

C. (100, 60, 80, 90, 120, 110, 130)

D. (100, 80, 60, 90, 120, 130, 110)

8. 从空树开始, 依次插入元素 52、26、14、32、71、60、93、58、24 和 41 后构成了一棵二叉排序树。在该树查找 60 要进行比较的次数为 ()。

A. 3

B. 4

C. 5

D. 6

9. 【2013 年计算机联考真题】

在任意一棵非空二叉排序树 T_1 中, 删除某结点 v 之后形成二叉排序树 T_2 , 再将 v 插入 T_2 形成二叉排序树 T_3 。下列关于 T_1 与 T_3 的叙述中, 正确的是 ()。

I. 若 v 是 T_1 的叶结点, 则 T_1 与 T_3 不同

II. 若 v 是 T_1 的叶结点, 则 T_1 与 T_3 相同

III. 若 v 不是 T_1 的叶结点, 则 T_1 与 T_3 不同

IV. 若 v 不是 T_1 的叶结点, 则 T_1 与 T_3 相同

A. 仅 I、III

B. 仅 I、IV

C. 仅 II、III

D. 仅 II、IV

10. 在含有 n 个结点的二叉排序树中查找某个关键字的结点时, 最多进行 () 次比较。

A. $n/2$

B. $\log_2 n$

C. $\log_2 n + 1$

D. n

11. 构造一棵具有 n 个结点的二叉排序树, 在最理想情况下的深度为 ()。

A. $n/2$

B. n

C. $\lfloor \log_2(n+1) \rfloor$

D. $\lceil \log_2(n+1) \rceil$

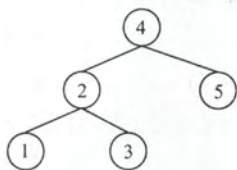
12. 不可能生成如第 12 题图所示的二叉排序树的关键字序列是 ()。

A. {4, 2, 1, 3, 5}

B. {4, 2, 5, 3, 1}

C. {4, 5, 2, 1, 3}

D. {4, 5, 1, 2, 3}



第 12 题图 一棵二叉排序树

13. 含有 20 个结点的平衡二叉树的最大深度为 ()。

A. 4

B. 5

C. 6

D. 7

14. 具有 5 层结点的 AVL 树至少有 () 个结点。

A. 10

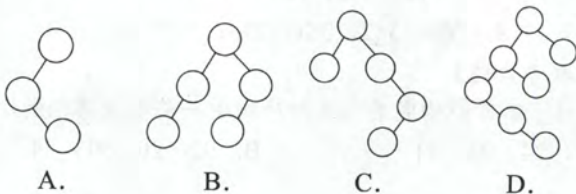
B. 12

C. 15

D. 17

15. 【2009 年计算机联考真题】

下列二叉排序树中, 满足平衡二叉树定义的是 ()。



16. 【2010年计算机联考真题】

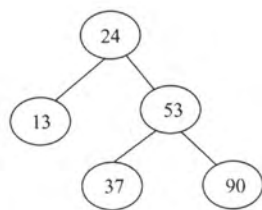
在如题 16 图所示的平衡二叉树中插入关键字 48 后得到一棵新平衡二叉树, 在新平衡二叉树中, 关键字 37 所在结点的左、右子结点中保存的关键字分别是 ()。

A. 13, 48

B. 24, 48

C. 24, 53

D. 24, 90



第 16 题图 一棵平衡二叉树

17. 【2012年计算机联考真题】

若平衡二叉树的高度为 6, 且所有非叶子结点的平衡因子均为 1, 则该平衡二叉树的结点总数为 ()。

A. 12

B. 20

C. 32

D. 33

18. 【2013年计算机联考真题】

若将关键字 1, 2, 3, 4, 5, 6, 7 依次插入到初始为空的平衡二叉树 T 中, 则 T 中平衡因子为 0 的分支结点的个数是 ()。

A. 0

B. 1

C. 2

D. 3

19. 在有 n 个叶子结点的哈夫曼树中, 非叶子结点的总数 ()。A. $n-1$ B. n C. $2n-1$ D. $2n$

20. 【2010年计算机联考真题】

对 n ($n \geq 2$) 个权值均不相同的字符构成哈夫曼树, 关于该树的叙述中, 错误的是 ()。

A. 该树一定是一棵完全二叉树

B. 树中一定没有度为 1 的结点

C. 树中两个权值最小的结点一定是兄弟结点

D. 树中任一非叶结点的权值一定不小于下一层任一结点的权值

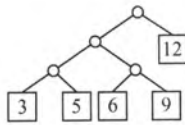
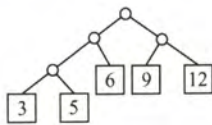
21. 给定整数集合 {3, 5, 6, 9, 12}, 与之对应的哈夫曼树是 ()。

A.

B.

C.

D.



22. 下列编码中 () 不是前缀码。

A. {00, 01, 10, 11}

B. {0, 1, 00, 11}

C. {0, 10, 110, 111}

D. {10, 110, 1110, 1111}

23. 【2014年计算机联考真题】

5 个字符有如下 4 种编码方案, 不是前缀编码的是 ()。

A. 01,0000,0001,001,1

B. 011,000,001,010,1

C. 000,001,010,011,100

D. 0,100,110,1110,1100

24. 设哈夫曼编码的长度不超过 4, 若已对两个字符编码为 1 和 01, 则还最多可对 () 个字符编码。

A. 2

B. 3

C. 4

D. 5

25. 一棵哈夫曼树共有 215 个结点, 对其进行哈夫曼编码, 共能得到 () 个不同的码字。

A. 107

B. 108

C. 214

D. 215

26. 对于哈夫曼树的说法错误的是 ()

- A. 对应一组权值构造出来的哈夫曼树一般不是唯一的
 B. 哈夫曼树具有最小的带权路径长度
 C. 哈夫曼树中没有度为 1 的结点
 D. 哈夫曼树中除了度为 1 的结点外, 还有度为 2 的结点和叶结点
27. 若度为 m 的哈夫曼树中, 叶子结点个数为 n , 则非叶子结点的个数为 ()。
 A. $n-1$ B. $\lfloor n/m \rfloor - 1$ C. $\lceil (n-1)/(m-1) \rceil$ D. $\lceil n/(m-1) \rceil - 1$

28. 【2015 年计算机联考真题】

现有一棵无重复关键字的平衡二叉树 (AVL 树), 对其进行中序遍历可得到一个降序序列。下列关于该平衡二叉树的叙述中, 正确的是 ()。

- A. 根结点的度一定为 2 B. 树中最小元素一定是叶结点
 C. 最后插入的元素一定是叶结点 D. 树中最大元素一定是无左子树
29. 【2015 年计算机联考真题】

下列选项给出的是从根分别到达两个叶结点路径上的权值序列, 能属于同一棵哈夫曼树的是 ()。

- A. 24,10,5 和 24,10,7 B. 24,10,5 和 24,12,7
 C. 24,10,10 和 24,14,11 D. 24,10,5 和 24,14,6

30. 【2017 年计算机联考真题】

已知字符集 $\{a,b,c,d,e,f,g,h\}$, 若各字符的哈夫曼编码依次是 0100, 10, 0000, 0101, 001, 011, 11, 0001, 则编码序列 0100011001001011110101 的译码结果是 ()。

- A. acgabfgh B. adbaggbb C. afbeagfd D. afeefgd

二、综合应用题

- 有一棵二叉排序树按先序遍历得到的序列为 (50, 38, 30, 45, 40, 48, 70, 60, 75, 80), 试画出该平衡二叉树, 并求出等概率下的查找成功和查找失败的平均查找长度。
- 按照序列 (40, 72, 38, 35, 67, 51, 90, 8, 55, 21) 建立一棵二叉排序树, 画出该树。并求出在等概率的情况下, 查找成功的平均查找长度。
- 依次把结点 (34, 23, 15, 98, 115, 28, 107) 插入到初始状态为空的平衡二叉排序树中, 使得在每次插入后保持该树仍然是平衡二叉树。请依次画出每次插入后所形成的平衡二叉排序树。
- 给定一个关键字集合 $\{25, 18, 34, 9, 14, 27, 42, 51, 38\}$, 假定查找各关键字的概率相同, 请画出其最佳二叉排序树。
- 画出一个二叉树, 使得它既满足大根堆的要求又满足二叉排序树的要求。
- 试编写一个算法, 判断给定的二叉树是否是二叉排序树。
- 设计一个算法, 求出指定结点在给定二叉排序树中的层次。
- 利用二叉树遍历的思想编写一个判断二叉树是否是平衡二叉树的算法。
- 设计一个算法, 求出给定二叉排序树中最小和最大的关键字。
- 设计一个算法, 从大到小输出二叉排序树中所有其值不小于 k 的关键字。
- 设给定权集 $w = \{5, 7, 2, 3, 6, 8, 9\}$, 试构造关于 w 的一棵哈夫曼树, 并求其加权路径长度 WPL。
- 编写一个递归算法, 在一棵有 n 个结点的随机建立起来的二叉排序树上查找第 k ($1 \leq k \leq n$) 小的元素, 并返回指向该结点的指针。要求算法的平均时间复杂度为 $O(\log_2 n)$ 。二叉排序树

的每个结点中除 data、lchild、rchild 等数据成员外, 增加一个 count 成员, 保存以该结点为根的子树上的结点个数。

13. 【2012 年计算机联考真题】

设有 6 个有序表 A、B、C、D、E、F, 分别含有 10、35、40、50、60 和 200 个数据元素, 各表中元素按升序排列。要求通过 5 次两两合并, 将 6 个表最终合并成 1 个升序表, 并在最坏情况下比较的总次数达到最小。请回答下列问题。

- 1) 给出完整的合并过程, 并求出最坏情况下比较的总次数。
- 2) 根据你的合并过程, 描述 $n(n \geq 2)$ 个不等长升序表的合并策略, 并说明理由。

4.5.5 答案与解析

一、单项选择题

1. C

二叉排序树插入新结点时不会引起树的分裂组合。对二叉排序树进行中序遍历可得到有序序列。当插入的关键字有序时, 二叉排序树会形成一个长链, 此时深度最大。在此种情况下进行查找, 有可能需要比较每个结点的关键字, 超过总结点数的 $1/2$ 。

2. B

由二叉排序树的定义不难得出中序遍历二叉树得到的序列是一个有序序列。

3. A

二叉排序树的查找路径是自顶向下的, 其平均查找长度主要取决于树的高度。

4. B

在二叉排序树的存储结构中, 每个结点由三部分构成, 其中左(或右)指针指向比结点的关键字值小(或大)的结点。关键字值最大的结点位于二叉排序树的最右位置上, 因此它的右指针一定为空(注意, 有可能不是叶子结点)。

5. C

在二叉排序树上查找时, 先与根结点值进行比较, 若相同, 则查找结束, 否则根据比较结果, 沿着左子树或右子树向下继续查找。根据二叉排序树定义, 有左子树结点值 \leq 根结点值 \leq 右子树结点值。C 序列中, 比较 911 关键字后, 应转向其左子树比较 240, 左子树中不应出现比 911 更大的数值, 但是 240 竟有一个右孩子结点值为 912, 所以不可能是正确的序列。

6. A

在二叉排序树中, 左子树结点值小于根结点, 右子树结点值大于根结点。在选项 A 中, 当查找到 91 后再向 24 查找, 说明这一条路径(左子树)之后查找的数都要比 91 小, 而后面却查找到了 94, 因此错误, 故选 A。

7. C

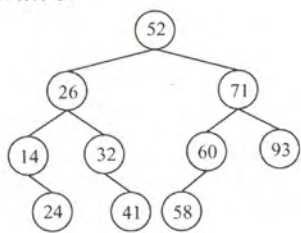
按照二叉排序树的构造方法, 不难得到 A、B、D 序列的构造结果相同。

8. A

以第一个元素为根结点, 依次将元素插入到树中, 生成的二叉排序树如第 8 题图所示。进行查找时, 先与根结点比较, 然后根据比较结果, 继续在左子树或者右子树上进行查找。比较的结点依次为 52、71、60。

9. C

在一棵二叉排序树中删除一个结点后再将此结点插入到二叉排



第 8 题图

序树中,如果删除的是叶子结点,那么插入结点后的二叉排序树与删除之前的相同。如果删除的不是叶子结点,那么在插入结点后的二叉排序树会发生变化,不完全相同。

10. D

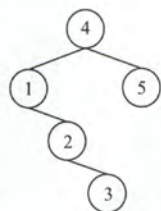
当输入序列是一个有序序列,构造的二叉排序树是一个单支树,当查找一个不存在的关键字值或最后一个结点的关键字值时,需要 n 次比较。

11. D

当二叉排序树的叶子结点全部都在相邻的两层内时,深度最小。理想情况是从第一层到倒数第二层为满二叉树。类比完全二叉树,可得深度为 $\lceil \log_2(n+1) \rceil$ 。

12. D

选项 D 中,插入 1 后,再插入 2,2 应作为 1 的右孩子结点,再插入 3,3 应作为 2 的右孩子结点。故选 D。选项 D 对应的二叉排序树如第 12 题图所示。



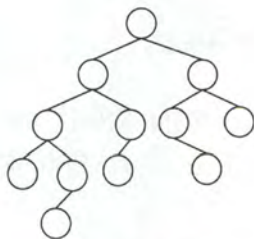
第12题图

13. C

平衡二叉树结点数的递推公式为 $N_0=0, N_1=1, N_2=2, N_h=1+N_{h-1}+N_{h-2}$ (h 为平衡二叉树高度, N_h 为构造此高度的平衡二叉树所需最少结点数)。通过递推公式可得,构造 5 层平衡二叉树至少需 12 个结点,构造 6 层至少需要 20 个。

14. B

设 N_h 表示高度为 h 的平衡二叉树中含有的最少结点数,则有 $N_1=1; N_2=2; N_h=N_{h-1}+N_{h-2}+1$, 由此求出 $N_5=12$, 对应的 AVL 树如第 14 题图所示。



第14题图

15. B

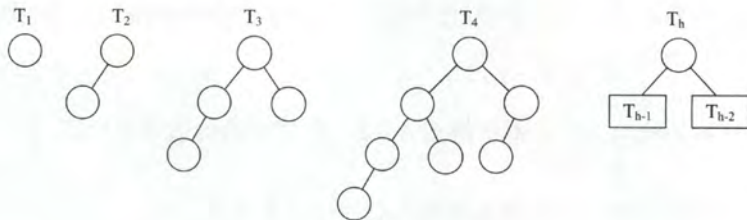
根据平衡二叉树的定义有,任意结点的左、右子树高度差的绝对值不超过 1。而其余 3 个答案均可以找到不满足条件的结点。

16. C

插入 48 以后,该二叉树根结点的平衡因子由 -1 变为 -2,失去平衡,需进行两次旋转(先右旋后左旋)操作。

17. B

所有非叶结点的平衡因子均为 1,即平衡二叉树满足平衡的最少结点情况,如第 17 题图所示。对于高度为 N 、左右子树的高度分别为 $N-1$ 和 $N-2$ 、所有非叶结点的平衡因子均为 1 的平衡二叉树,总结点数的公式为: $C_N=C_{N-1}+C_{N-2}+1, C_1=1, C_2=2, C_3=2+1+1=4$, 可推出 $C_6=20$ 。



第17题图

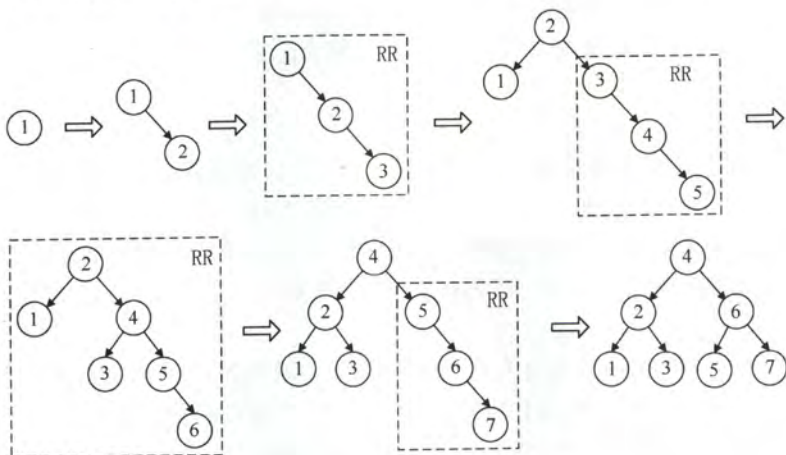
画图法:先画出 T_1 和 T_2 ; 然后新建一个根结点,连接 T_2 、 T_1 构成 T_3 ; 新建一个根结点,连接 T_3 、 T_2 构成 T_4 ; ……依此类推,直到画出 T_6 ,可知 T_6 的结点数为 20。

排除法:对于选项 A,高度为 6、结点数为 10 的树怎么也无法达到平衡。对于选项 C,结点

较多时,考虑较极端情形,即第6层只有最左叶子的完全二叉树刚好有32个结点,虽然满足平衡的条件,但显然再删去部分结点,依然不影响平衡,不是最少结点的情况。同理D错误。只能选B。

18. D。

利用7个关键字构建平衡二叉树T,平衡因子为0的分支结点个数为3,构建的平衡二叉树如下图所示。构造及调整的过程如下:



19. A

由哈夫曼树的构造过程可知,哈夫曼树中只有度为0和2的结点。在非空二叉树中有 $N_0 = N_2 + 1$, 故 $N_2 = n - 1$ 。

另解: n 个结点构造哈夫曼树需要 $n-1$ 次合并过程,每次合并新建一个分支结点,故选A。

20. A

哈夫曼树为带权路径长度最小的二叉树,不一定是完全二叉树。哈夫曼树中没有度为1的结点, B 正确。构造哈夫曼树时,最先选取两个权值最小的结点作为左、右子树构造一棵新的二叉树, C 正确。哈夫曼树中任一非叶结点P的权值为其左、右子树根结点权值之和,其权值不小于其左、右子树根结点的权值,可知哈夫曼树中任一非叶结点的权值一定不小于下一层任一结点的权值, D 正确。

21. C

先是3和5构造为一棵子树,其根权值为8,然后该子树与6构成一棵新子树,根权值为14,然后9与12构造为一棵子树,最后两棵子树共同构造为一棵哈夫曼树。

22. B

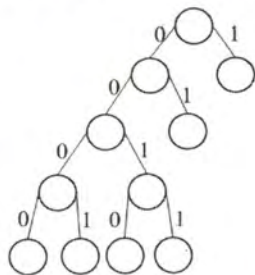
如果没有一个编码是另一个编码的前缀,称这样的编码为前缀编码。B选项中,0是00的前缀,1是11的前缀。

23. D

前缀编码的定义是在一个字符集中,任何一个字符的编码都不是另一个字符编码的前缀。选项D中编码110是编码1100的前缀,违反了前缀编码的规则,所以D不是前缀编码。

24. C

在哈夫曼编码中,一个编码不能是任何其他编码的前缀。3位编码可能是001,对应的4位编码只能是0000和0001。3位编码也可能是000,对应的4位编码只能是00010和00011。若全采用4位编码,可以为0000、



第24题图

0001、0010 和 0011。题中间的是最多，故选 C。

另解：若哈夫曼编码的长度只允许小于等于 4，则哈夫曼树的高度最高是 5，已知一个字符编码为 1，另一个字符编码是 01，这说明第二层和第三层各有一个叶子结点，为使得该树从第 3 层起能够对尽可能多的字符编码，余下的二叉树应该是满二叉树，如第 24 题图所示，底层可以有 4 个叶结点，最多可以再对 4 个字符编码。

25. B

根据上题结论，叶子结点数为 $(215+1)/2=108$ ，所以共有 108 个不同的码字。

另解：在哈夫曼树中只有度为 0 和 2 的结点，结点总数 $N=N_0+N_2$ ，且 $N_0=N_2+1$ ，由题知 $N=215$ ， $N_0=108$ 。

26. D

哈夫曼树通常是指带权路径长度达到最小的扩充二叉树，在其构造过程中每次选权的权值最小的两棵树，一棵作为左子树，一棵作为右子树，生成新的二叉树，新的二叉树根的权值应为其左右两棵子树根结点权值的和。至于谁做左子树，谁做右子树，没有限制，所以构造哈夫曼树是不唯一的。哈夫曼树只有度为 0 和 2 的结点，度为 0 的结点是外结点，带有权值，没有度为 1 的结点。

27. C

一棵度为 m 的哈夫曼树应只有度为 0 和 m 的结点，设度为 m 的结点有 N_m 个，度为 0 的结点有 N_0 个，又设结点总数为 N ， $N=N_0+N_m$ 。因有 N 个结点的哈夫曼树有 $N-1$ 条分支，则 $m \cdot N_m = N-1 = N_m + N_0 - 1$ ，整理得 $(m-1) \cdot N_m = N_0 - 1$ ， $N_m = (n-1)/(m-1)$ 。

28. D

只有两个结点的平衡二叉树的根结点的度为 1，A 错误。中序遍历后可以得到一个降序序列，树中最大元素一定无左子树（可能有右子树），因此不一定是叶结点，B 错误。最后插入的结点可能会导致平衡调整，而不一定是叶结点，C 错误。

29. D

在哈夫曼树中，左右孩子权值之和为父结点权值。仅以分析选项 A 为例：若两个 10 分别属于两棵不同的子树，根的权值不等于其孩子的权值和，不符；若两个 10 属同棵子树，其权值不等于其两个孩子（叶结点）的权值和，不符。B、C 选项的排除方法一样。

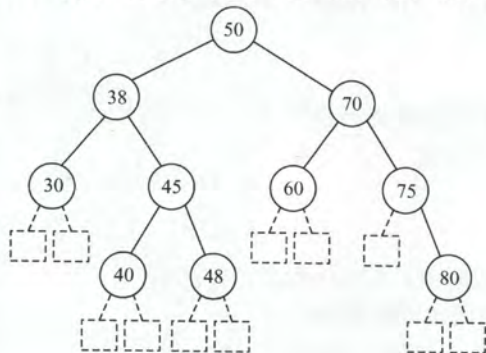
30. D

哈夫曼编码是前缀编码，各个编码的前缀不相同，因此直接拿编码序列与哈夫曼编码一一比对即可。序列可分割为 0100 011 001 001 011 11 0101，译码结果是 a f e e f g d。D 正确。

二、综合应用题

1. 解答：

先序序列为 (50, 38, 30, 45, 40, 48, 70, 60, 75, 80)，二叉树的中序序列是一个有序



序列，故为 (30, 38, 40, 45, 48, 50, 60, 70, 75, 80)，由先序序列和中序序列可以构造出对应的二叉树，如左图所示。

查找成功的平均查找长度为

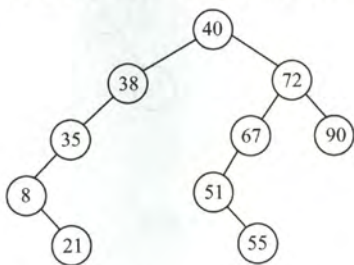
$$ASL = (1 \times 1 + 2 \times 2 + 3 \times 4 + 4 \times 3) / 10 = 2.9$$

图中的方块结点为虚构的查找失败结点，其查找路径为从根结点到其父结点（圆形结点）的结点序列，故对应的查找失败平均长度为

$$ASL = (3 \times 5 + 4 \times 6) / 11 = 39 / 11$$

2. 解答:

根据二叉排序树的定义, 该序列所对应的二叉排序树如下图所示。

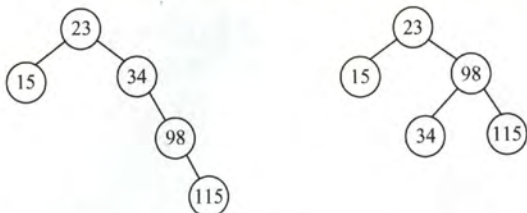


平均查找长度为: $ASL(1+2 \times 2+3 \times 3+4 \times 2+5 \times 2)/10=3.2$

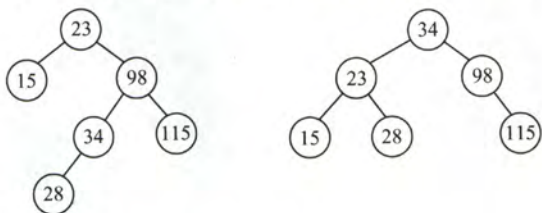
3. 解答:



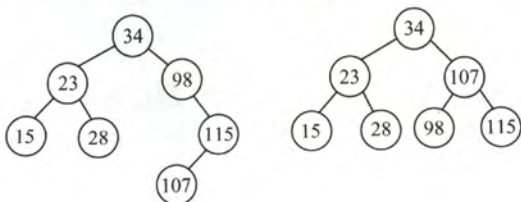
第一步: 插入结点 34、23、15 后, 需要根结点 34 的子树做 LL 调整。



第二步: 插入结点 98、115 后, 需要根结点 34 的子树做 RR 调整。



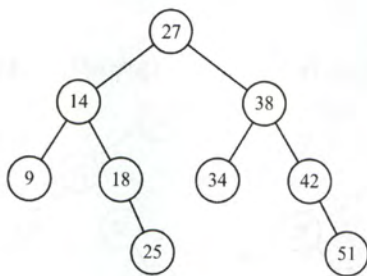
第三步: 插入结点 28 后, 需要根结点 23 的子树做 RL 调整。



第四步: 插入结点 107 后, 需要根结点 98 的子树做 RL 调整。

4. 解答:

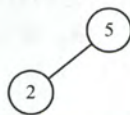
当各关键字的查找概率相等时, 最佳二叉排序树应是高度最小的二叉排序树。构造过程分两步走: 首先对各关键字按值从小到大排序, 然后仿照折半查找的判定树的构造方法构造二叉排序树。这样得到的就是最佳二叉排序树。



第4题图

5. 解答:

大根堆要求根结点的关键字值既大于或等于左子女的关键字值,又大于或等于右子女的关键字值。二叉排序树要求根结点的关键字值大于左子女的关键字值,同时小于右子女的关键字值。两者的交集是:根结点的关键字值大于左子女的关键字值。这意味着是一棵左斜单支树。但大根堆要求是完全二叉树,因此最后得到的只能是如第5题图所示的两个结点的二叉树。



第5题图

读者也可能会注意到,当只有一个结点时,显然是满足题意的,但我们不举一个结点的例子是为了体现出排序树与大根堆的区别。

6. 解答:

对二叉排序树来说,其中序遍历序列为一个递增有序序列。因此,对给定的二叉树进行中序遍历,如果始终能保持前一个值比后一个值小,则说明该二叉树是一棵二叉排序树。

```

KeyType predt=-32767;           //predt 为全局变量,保存当前结点中序前驱
                                //的值,初值为-∞

int JudgeBST(BiTree bt){
    int b1,b2;
    if(bt==NULL)                 //空树
        return 1;
    else{
        b1=JudgeBST(bt->lchild); //判断左子树是否是二叉排序树
        if(b1==0||predt>=bt->data) //若左子树返回值为0或前驱大于等于当前结点
            return 0;              //则不是二叉排序树
        predt=bt->data;            //保存当前结点的关键字
        b2=JudgeBST(bt->rchild);   //判断右子树
        return b2;                //返回右子树的结果
    }
}
  
```

7. 解答:

算法思想:设二叉树采用二叉链表存储结构。在二叉排序树中,查找一次就下降一层。因此,查找该结点所用的次数就是该结点在二叉排序树中的层次。采用二叉排序树非递归查找算法,用 n 保存查找层次,每查找一次, n 就加 1,直到找到相应结点。算法如下:

```

int level(BiTree bt,BSTNode *p){
    //本算法计算给定结点在二叉排序树中的层次
    int n=0;                      //统计查找次数
    BiTree t=bt;
    if(bt!=NULL){
        n++;
        while(t->data!=p->data){
  
```



```

        if(t->data<p->data)    //在左子树中查找
            t=t->rchild;
        else                  //在右子树中查找
            t=t->lchild;
        n++;                  //层次加1
    }
}
return n;
}

```

8. 解答:

设置二叉树的平衡标记 **balance**, 以标记返回二叉树 **bt** 是否为平衡二叉树, 若为平衡二叉树, 则返回 1, 否则返回 0; **h** 为二叉树 **bt** 的高度。采用后序遍历的递归算法:

1) 若 **bt** 为空, 则高度为 0, **balance**=1。

2) 若 **bt** 仅有根结点, 则高度为 1, **balance**=1。

3) 否则, 对 **bt** 的左、右子树执行递归运算, 返回左、右子树的高度和平衡标记, **bt** 的高度为最高子树的高度加 1。若左、右子树的高度差大于 1, 则 **balance**=0; 若左、右子树的高度差小于等于 1, 且左、右子树都平衡时, **balance**=1, 否则 **balance**=0。

```

void Judge_AVL(BiTree bt,int &balance,int &h){
//本算法判断一个给定的二叉树是否为平衡二叉树
    int bl=0,br=0,h1=0,hr=0;                //左、右子树的平衡标记和高度
    if(bt==NULL){                            //空树, 高度为 0
        h=0;
        balance=1;
    }
    else if(bt->lchild==NULL&&bt->rchild==NULL){ //仅有根结点, 则高度为 1
        h=1;
        balance=1;
    }
    else{
        Judge_AVL(bt->lchild,bl,h1);    //递归判断左子树
        Judge_AVL(bt->rchild,br,hr);    //递归判断右子树
        h=(h1>hr?h1:hr)+1;
        if(abs(h1-hr)<2)    //若子树高度差的绝对值<2, 则看左、右子树是否都平衡
            balance=bl&&br; //&&为逻辑与, 即左、右子树都平衡时, 二叉树平衡
        else
            balance=0;
    }
}

```

9. 解答:

在一棵二叉排序树中, 最左下结点即为关键字最小的结点, 最右下结点即为关键字最大的结点, 本算法只要找出这两个结点即可, 而不需要比较关键字。

```

KeyType MinKey(BSTNode *bt){
//求出二叉排序树中最小关键字结点
    while(bt->lchild!=NULL)
        bt=bt->lchild;
    return bt->data;
}

```



```

KeyType MaxKey(BSTNode *bt){
    //求出二叉排序树中最大关键字结点
    while(bt->rchild!=NULL)
        bt=bt->rchild;
    return bt->data;
}

```

10. 解答:

由二叉排序树的性质可知, 右子树中所有的结点值均大于根结点值, 左子树中所有的结点值均小于根结点值。为了从大到小输出, 先遍历右子树, 再访问根结点, 后遍历左子树。

```

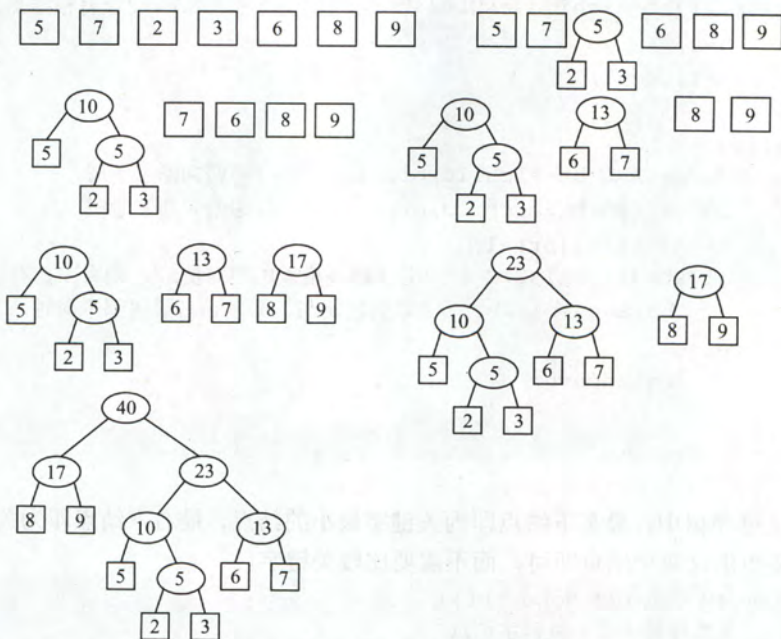
void OutPut(BSTNode *bt, KeyType k)
{
    //本算法从大到小输出二叉排序树中所有值不小于 k 的关键字
    if(bt==NULL)
        return ;
    if(bt->rchild!=NULL)
        OutPut(bt->rchild, k); //递归输出右子树结点
    if(bt->data>=k)
        printf("%d", bt->data); //只输出大于等于 k 的结点值
    if(bt->lchild!=NULL)
        OutPut(bt->lchild, k); //递归输出左子树的结点
}

```

本题也可采用中序遍历加辅助栈的方法实现。

11. 解答:

根据哈夫曼树的构造方法, 每次从森林中选取两个根结点值最小的树合并成一棵树, 将原先的两棵树作为左、右子树, 且新根结点的值为左、右孩子关键字之和。构造过程如下图所示。



由构造出的哈夫曼树可得 $WPL = (2+3) \times 4 + (5+6+7) \times 3 + (8+9) \times 2 = 108$ 。

注意: 哈夫曼树并不唯一, 但带权路径长度一定是相同的。

12. 解答:

设二叉排序树的根结点为*t, 根据结点存储的信息, 有以下几种情况:

• 若 $t \rightarrow lchild$ 为空, 情况如下:

- 1) 若 $t \rightarrow rchild$ 非空且 $k=1$, 则 *t 即为第 k 小的元素, 查找成功;
- 2) 若 $t \rightarrow rchild$ 非空且 $k \neq 1$, 则第 k 小的元素必在 *t 的右子树;

• 若 $t \rightarrow lchild$ 非空, 情况如下:

- 1) $t \rightarrow lchild \rightarrow count = k-1$, 则 *t 即为第 k 小的元素, 查找成功;
- 2) $t \rightarrow lchild \rightarrow count > k-1$, 则第 k 小的元素必在 *t 的左子树, 继续到 *t 的左子树中查找;
- 3) $t \rightarrow lchild \rightarrow count < k-1$, 则第 k 小的元素必在右子树, 继续搜索右子树, 寻找第

$k - (t \rightarrow lchild \rightarrow count + 1)$ 小的元素。

对左右子树的搜索采用相同的规则, 递归实现的算法描述如下:

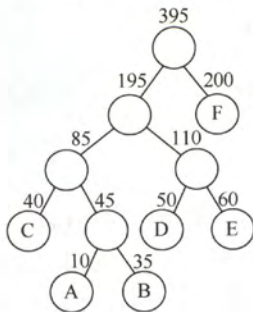
```
BSTNode *Search_Small(BSTNode*t, int k){
//在以 t 为根的子树上寻找第 k 小的元素, 返回其所在结点的指针。K 从 1 开始计算
//在树结点中增加一个 count 数据成员, 存储以该结点为根的子树的结点个数
    if(k<1||k>t->count) return NULL;
    if(t->lchild==NULL){
        if(k==1) return t;
        else return Search_Small(t->rchild, k-1);
    }
    else{
        if(t->lchild->count==k-1) return t;
        if(t->lchild->count>k-1) return Search_Small(t->lchild, k);
        if(t->lchild->count<k-1)
            return Search_Small(t->rchild, k-(t->lchild->count+1));
    }
}
```

最大查找长度取决于树的高度。由于二叉排序树是随机生成的, 其高度应该是 $O(\log_2 n)$, 算法的时间复杂度为 $O(\log_2 n)$ 。

13. 解答:

1) 由于最先合并的表中的元素, 在后续每次合并中都会再次参与比较, 因此求最小合并次数类似于求最小带权路径长度, 立即想到哈夫曼树。根据哈夫曼树的构造过程, 每次选择表集中长度最小的两个表进行合并。6 个表的合并顺序如第 13 题图所示。

根据第 13 题图中的哈夫曼树, 6 个序列的合并过程为:



第 13 题图

① 在表集合{10, 35, 40, 50, 60, 200}中, 选择表 A 与表 B 合并, 生成含 45 个元素的表 AB;

② 在表集合{40, 45, 50, 60, 200}中, 将表 AB 与表 C 合并, 生成含 85 个元素的表 ABC;

③ 在表集合{50, 60, 85, 200}中, 表 D 与表 E 合并, 生成含 110 个元素的表 DE;

④ 在表集合{85, 110, 200}中, 表 ABC 与表 DE 合并, 生成含 195 个元素的表 ABCDE;

⑤ 当前表集合{195, 200}, 表 ABCDE 与表 F 合并, 生成含 395 个元素的表 ABCDEF;

由于合并两个长度分别为 m 和 n 的有序表, 最坏情况下需要比较 $m+n-1$ 次, 故最坏情况下比较的总次数计算如下:

第 1 次合并: 最多比较次数 $=10+35-1=44$;

第 2 次合并: 最多比较次数 $=40+45-1=84$;

第 3 次合并: 最多比较次数 $=50+60-1=109$;

第 4 次合并: 最多比较次数 $=85+110-1=194$;

第 5 次合并: 最多比较次数 $=195+200-1=394$;

比较的总次数最多为: $44+84+109+194+394=825$ 。

2) 各表的合并策略是: 在对多个有序表进行两两合并时, 若表长不同, 则最坏情况下总的比较次数依赖于表的合并次序。可以借助于哈夫曼树的构造思想, 依次选择最短的两个表进行合并, 可以获得最坏情况下最佳的合并效率。

归纳总结

本章的内容较多, 其中二叉树是极其重要的查考点。关于二叉树的有关操作, 在 2014 年的统考中第一次打破传统, 出现了树的算法设计题, 需要引起读者的注意。

其中遍历是二叉树各种操作的基础, 可以在遍历过程中考查对结点的各种其他操作, 而且容易结合递归算法和利用栈或队列的非递归算法。读者需重点掌握各种遍历方法的代码书写, 并学会在遍历的基础上, 进行一些其他的相关操作。其中递归算法短小精悍, 出现的概率较大, 请读者们不要掉以轻心, 要做到对几种遍历方式的程序模板烂熟于心, 并结合一定数量的习题, 才可以在考试中快速地写出漂亮的代码。

(一) 遍历的递归程序

```
void Track(BiTree *p){
    if(p!=NULL){
        // (1)
        Track(p->lchild);
        // (2)
        Track(p->rchild);
        // (3)
    }
}
```

访问函数 visit() 位于 (1)、(2)、(3) 的位置, 分别对应于先序、中序、后序遍历。但对于具体题目来说, 设计算法的时候要灵活应用。请读者认真练习下面的例题。

例题: 设二叉树的存储结构为二叉链表, 编写有关二叉树的递归算法。

(1) 统计二叉树中度为 1 的结点个数。

(2) 统计二叉树中度为 2 的结点个数。

- (3) 统计二叉树中度为0的结点个数。
- (4) 统计二叉树的高度。
- (5) 统计二叉树的宽度。
- (6) 从二叉树中删去所有叶结点。
- (7) 计算指定结点*p所在的层次。
- (8) 计算二叉树中各结点中的最大元素的值。
- (9) 交换二叉树中每个结点的两个子女。
- (10) 以先序次序输出一颗二叉树中所有结点的数据值及结点所在层次。

(二) 非递归函数程序

先序、中序、后序遍历的非递归算法都是利用栈实现，层次遍历利用队列实现。中序遍历的非递归算法和层次遍历的实现参见4.3.1节。请读者自行实现先序遍历的非递归算法。

后序遍历的非递归算法最为麻烦，栈中不仅要保留指向结点的指针，还必须有一些额外的信息来表示结点的右结点是否访问过。在这里给出它的一种实现算法：

```
typedef struct{
    BTreeNode *p;      //p 是二叉树的结点的指针
    int rvisited;      //rvisited=1 代表 p 所指结点的右结点已被访问过
}SNode;               //栈中的结点定义

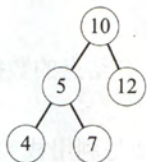
typedef struct{
    SNode Elem[maxsize];
    int top;
}SqStack;             //栈结构体

void PostOrder2(BiTree T){
    SNode sn;
    BTreeNode *pt=T;
    InitStack(S);      //从根结点开始，往左下方走，将路径上的每一个结点入栈
    while(T){
        Push(pt,0);    //push 到栈中两个信息：一是结点指针，一是其右结点是否被访问过
        pt=pt->lchild;
    }
    while(!S.IsEmpty()){ //只要栈 S 非空
        sn=S.getTop();    //sn 是栈顶结点
        if(sn.p->rchild==NULL||sn.rvisited){
            Pop(S,pt);
            visit(pt);
        }
        else{ //若它的右孩子存在且 rvisited 为 0，处理其右孩子
            sn.rvisited=1; //往左下方走到尽头，将路径上所有元素入栈
            pt=sn.p->rchild;
            while(pt!=NULL){
                Push(S,pt,0);
                pt=pt->lchild;
            }
        }
        //本轮结束
    }
    //while 结束
}
```


思维拓展

输入一个整数 `data` 和一棵二元树。从树的根结点开始往下访问一直到叶结点，所经过的所有结点形成一条路径。打印出路径和与 `data` 相等的所有路径。

例如：输入整数 22 和如下二元树。



则打印出两条路径：10, 12 和 10, 5, 7。

（提示：使用数组或栈保存访问的路径，并记录当前路径上所有元素的和 `sum`。如果当前结点为叶子结点，且当前结点值与 `sum` 的和等于 `data`，则满足条件，打印当前路径。然后递归返回到父结点，注意在递归返回之前要先减去当前结点元素的值。使用前序遍历操作的递归算法模板可以简化程序。）

第5章 图

【考纲内容】

(一) 图的基本概念

(二) 图的存储及基本操作

邻接矩阵法; 邻接表法; 邻接多重表; 十字链表

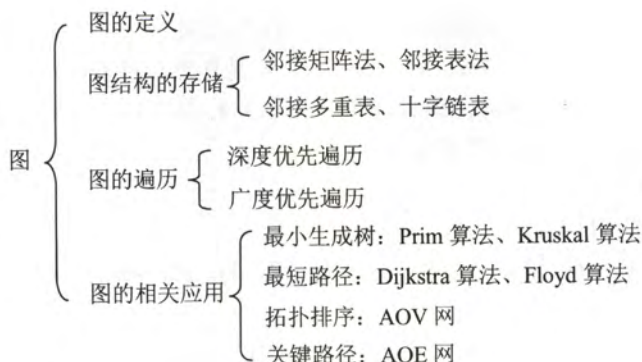
(三) 图的遍历

深度优先搜索; 广度优先搜索

(四) 图的基本应用

最小(代价)生成树; 最短路径; 拓扑排序; 关键路径

【知识框架】



【复习提示】

图算法的难度较大, 因此主要掌握深度优先搜索与广度优先搜索的程序设计, 其他内容以算法设计题形式出现的概率不大, 通常会以选择题和综合题的形式考查。应掌握图的基本概念及基本性质(度、路径长度、回路、路径等)、图的存储结构(邻接矩阵、邻接表、邻接多重表和十字链表)及其特性、存储结构之间的转化、基于存储结构上的遍历操作和各种应用(拓扑排序、最小生成树、最短路径和关键路径)等。图的相关算法较多易混, 但通常只要求掌握其基本思想和实现的步骤(能动手模拟), 而算法的具体实现则不是重点。读者应将这些算法按不同的方式分类, 对比记忆, 并用一定量的实例练习。

5.1 图的基本概念

5.1.1 图的定义

图 G 由顶点集 V 和边集 E 组成, 记为 $G=(V, E)$, 其中 $V(G)$ 表示图 G 中顶点的有限非空集; $E(G)$ 表示图 G 中顶点之间的关系 (边) 集合。若 $V=\{v_1, v_2, \dots, v_n\}$, 用 $|V|$ 表示图 G 中顶点的个数, 也称为图 G 的阶, $E=\{(u, v) | u \in V, v \in V\}$, 用 $|E|$ 表示图 G 中边的条数。

注意: 线性表可以是空表, 树可以是空树, 但图不可以是空图。就是说, 图中不能一个顶点也没有, 图的顶点集 V 一定非空, 但边集 E 可以为空, 此时图中只有顶点而没有边。

下面是图的一些基本概念及术语。

1. 有向图

若 E 是有向边 (也称为弧) 的有限集合时, 则图 G 为有向图。弧是顶点的有序对, 记为 $\langle v, w \rangle$, 其中 v, w 是顶点, v 称为弧尾, w 称为弧头, 称为从顶点 v 到顶点 w 的弧, 也称 v 邻接到 w , 或 w 邻接自 v 。

图 5-1 (a) 所示的有向图 G_1 可表示为

$$G_1=(V_1, E_1)$$

$$V_1=\{1, 2, 3\}$$

$$E_1=\{\langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle\}$$

2. 无向图

若 E 是无向边 (简称边) 的有限集合时, 则图 G 为无向图。边是顶点的无序对, 记为 (v, w) 或 (w, v) , 因为 $(v, w) = (w, v)$, 其中 v, w 是顶点。可以说顶点 w 和顶点 v 互为邻接点。边 (v, w) 依附于顶点 w 和 v , 或者说边 (v, w) 和顶点 v, w 相关联。

图 5-1 (b) 所示的无向图 G_2 可表示为

$$G_2=(V_2, E_2)$$

$$V_2=\{1, 2, 3, 4\}$$

$$E_2=\{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$$

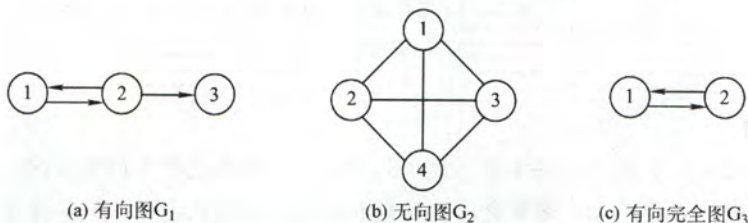


图 5-1 图的示例

3. 简单图

一个图 G 如果满足: ①不存在重复边; ②不存在顶点到自身的边, 则称图 G 为简单图。图 5-1 中 G_1 和 G_2 均为简单图。数据结构中仅讨论简单图。

4. 多重图

若图 G 中某两个结点之间的边数多于一条, 又允许顶点通过同一条边和自己关联, 则 G 为多重图。多重图的定义和简单图是相对的。

5. 完全图 (也称简单完全图)

在无向图中, 如果任意两个顶点之间都存在边, 则称该图为无向完全图。含有 n 个顶点的无向完全图有 $n(n-1)/2$ 条边。在有向图中, 如果任意两个顶点之间都存在方向相反的两条弧, 则称该图为有向完全图。含有 n 个顶点的有向完全图有 $n(n-1)$ 条有向边。图 5-1 中 G_2 为无向完全图, 而 G_3 为有向完全图。

6. 子图

设有两个图 $G=(V, E)$ 和 $G'=(V', E')$, 若 V' 是 V 的子集, 且 E' 是 E 的子集, 则称 G' 是 G 的子图。若有满足 $V(G')=V(G)$ 的子图 G' , 则为 G 的生成子图。图 5-1 中 G_3 为 G_1 的子图。

注意: 并非 V 和 E 的任何子集都能构成 G 的子图, 因为这样的子集可能不是图, 也就是说, E 的子集中的某些边关联的顶点可能不在这个 V 的子集中。

7. 连通、连通图和连通分量

在无向图中, 若从顶点 v 到顶点 w 有路径存在, 则称 v 和 w 是连通的。若图 G 中任意两个顶点都是连通的, 则称图 G 为连通图, 否则称为非连通图。无向图中的极大连通子图称为连通分量。如果一个图有 n 个顶点, 并且有小于 $n-1$ 条边, 则此图必是非连通图。如图 5-2 (a) 所示, 图 G_4 有 3 个连通分量, 如图 5-2 (b) 所示。

注意: 弄清连通、连通图、连通分量的概念非常重要。首先要区分极大连通子图和极小连通子图, 极大连通子图是无向图的连通分量, 极大即要求该连通子图包含其所有的边; 极小连通子图是既要保持图连通, 又要使得边数最少的子图。

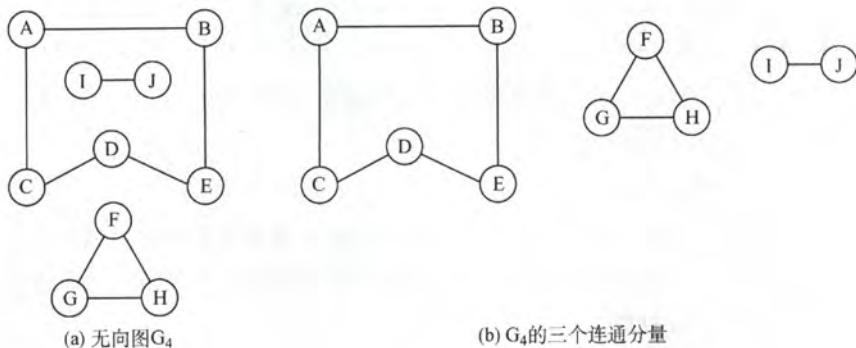


图 5-2 无向图及其连通分量

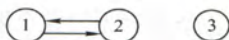
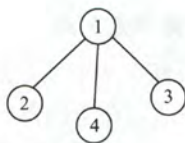
8. 强连通图、强连通分量

在有向图中, 若从顶点 v 到顶点 w 和从顶点 w 到顶点 v 之间都有路径, 则称这两个顶点是强连通的。若图中任何一对顶点都是强连通的, 则称此图为强连通图。有向图中的极大强连通子图称为有向图的强连通分量, 图 G_1 的强连通分量如图 5-3 所示。

注意: 强连通图、强连通分量只是针对有向图而言。一般在无向图中讨论连通性, 在有向图中考虑强连通性。

9. 生成树、生成森林

连通图的生成树是包含图中全部顶点的一个极小连通子图。若图中顶点数为 n , 则它的生成树含有 $n-1$ 条边。对于生成树而言, 若砍去它的一条边, 则会变成非连通图, 若加上一条边则会形成一个回路。在非连通图中, 连通分量的生成树构成了非连通图的生成森林。图 G_2 的一个生成树如图 5-4 所示。

图 5-3 图 G_1 的强连通分量图 5-4 图 G_2 的一个生成树

注意：包含无向图中全部顶点的极小连通子图，只有生成树满足条件，因为砍去生成树的任一条边，图将不再连通。

10. 顶点的度、入度和出度

图中每个顶点的度定义为以该顶点为一个端点的边的数目。

对于无向图，顶点 v 的度是指依附于该顶点的边的条数，记为 $TD(v)$ 。

在具有 n 个顶点 e 条边的无向图中，有 $\sum_{i=1}^n TD(v_i) = 2e$ 。即无向图的全部顶点的度之和等于边数的两倍，这是因为每条边和两个顶点相关联。

对于有向图，顶点 v 的度分为入度和出度，入度是以顶点 v 为终点的有向边的数目，记为 $ID(v)$ ；而出度是以顶点 v 为起点的有向边的数目，记为 $OD(v)$ 。顶点 v 的度等于其入度和出度之和，即 $TD(v) = ID(v) + OD(v)$ 。

在具有 n 个顶点 e 条边的有向图中，有 $\sum_{i=1}^n ID(v_i) = \sum_{i=1}^n OD(v_i) = e$ ，即有向图的全部顶点的入度之和与出度之和相等并且等于边数。这是因为每条有向边都有一个起点和终点。

11. 边的权和网

在一个图中，每条边都可以标上具有某种含义的数值，该数值称为该边的权值。这种边上带有权值的图称为带权图，也称作网。

12. 稠密图、稀疏图

边数很少的图称为稀疏图，反之，称为稠密图。稀疏和稠密本身是模糊的概念，稀疏图和稠密图常常是相对而言的。一般当图 G 满足 $|E| < |V| \cdot \log |V|$ 时，可以将 G 看成是稀疏图。

13. 路径、路径长度和回路

顶点 v_p 到顶点 v_q 之间的一条路径是指顶点序列 $v_p, v_{i1}, v_{i2}, \dots, v_{im}, v_q$ 。路径上边的数目称为路径长度。第一个顶点和最后一个顶点相同的路径称为回路或环。如果一个图有 n 个顶点，并且有大于 $n-1$ 条边，则此图一定有环。

14. 简单路径、简单回路

在路径序列中，顶点不重复出现的路径称为简单路径。除第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路称为简单回路。

15. 距离

从顶点 u 出发到顶点 v 的最短路径若存在，则此路径的长度称作从 u 到 v 的距离。若从 u 到 v 根本不存在路径，则记该距离为无穷 (∞)。

16. 有向树

有一个顶点的入度为 0，其余顶点的入度均为 1 的有向图称作有向树。

5.1.2 本节试题精选

一、单项选择题

1. 图中有关路径的定义是 ()。
A. 由顶点和相邻顶点序偶构成的边所形成的序列
B. 由不同顶点所形成的序列
C. 由不同边所形成的序列
D. 上述定义都不是
2. 一个有 n 个顶点和 n 条边的无向图一定是 ()。
A. 连通的 B. 不连通的 C. 无环的 D. 有环的
3. 如果从无向图的任意顶点出发进行一次深度优先搜索即可访问所有顶点, 则该图一定是 ()。
A. 强连通图 B. 连通图 C. 有回路 D. 一棵树
4. 【2011 年计算机联考真题】
下列关于图的叙述中, 正确的是 ()。
I. 回路是简单路径
II. 存储稀疏图, 用邻接矩阵比邻接表更省空间
III. 若有向图中存在拓扑序列, 则该图不存在回路
A. 仅 II B. 仅 I、II C. 仅 III D. 仅 I、III
5. 以下关于图的叙述中, 正确的是 ()。
A. 图与树的区别在于图的边数大于或等于顶点数
B. 假设有图 $G=(V, \{E\})$, 顶点集 $V' \subseteq V$, $E' \subseteq E$, 则 V' 和 $\{E'\}$ 构成 G 的子图
C. 无向图的连通分量指无向图中的极大连通子图
D. 图的遍历就是从图中某一顶点出发访遍图中其余顶点
6. 【2009 年计算机联考真题】
下列关于无向连通图特性的叙述中, 正确的是 ()。
I 所有顶点的度之和为偶数
II 边数大于顶点个数减 1
III 至少有一个顶点的度为 1
A. 只有 I B. 只有 II C. I 和 II D. I 和 III
7. 【2010 年计算机联考真题】
若无向图 $G=(V, E)$ 中含有 7 个顶点, 要保证图 G 在任何情况下都是连通的, 则需要的边数最少是 ()。
A. 6 B. 15 C. 16 D. 21
8. 以下关于图的叙述中, 正确的是 ()。
A. 强连通有向图的任何顶点到其他所有顶点都有弧
B. 图的任意顶点的入度等于出度
C. 有向完全图一定是强连通有向图
D. 有向图的边集的子集和顶点集的子集可构成原有有向图的子图
9. 【2013 年计算机联考真题】
设图的邻接矩阵 A 如下所示。各顶点的度依次是 ()。

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

- A. 1, 2, 1, 2 B. 2, 2, 1, 1 C. 3, 4, 2, 3 D. 4, 4, 2, 2
10. 一个有 28 条边的非连通无向图至少有 () 个顶点。
A. 7 B. 8 C. 9 D. 10
11. 对于一个有 n 个顶点的图: 如果是连通无向图, 其边的个数至少为 (); 如果是强连通有向图, 其边的个数至少为 ()。
A. $n-1, n$ B. $n-1, n(n-1)$ C. n, n D. $n, n(n-1)$
12. 无向图 G 有 23 条边, 度为 4 的顶点有 5 个, 度为 3 的顶点有 4 个, 其余都是度为 2 的顶点, 则图 G 最多有 () 个顶点。
A. 11 B. 12 C. 15 D. 16
13. 在有 n 个顶点的有向图中, 每个顶点的度最大可达 ()。
A. n B. $n-1$ C. $2n$ D. $2n-2$
14. 具有 6 个顶点的无向图, 当有 () 条边时能确保是一个连通图。
A. 8 B. 9 C. 10 D. 11
15. 【2017 年计算机联考真题】
已知无向图 G 含有 16 条边, 其中度为 4 的顶点个数为 3, 度为 3 的顶点个数为 4, 其他顶点的度均小于 3。图 G 所含的顶点个数至少是 ()。
A. 10 B. 11 C. 13 D. 15
16. 设有无向图 $G=(V, E)$ 和 $G'=(V', E')$, 如果 G' 是 G 的生成树, 则下列不正确的是 ()。
I. G' 为 G 的连通分量
II. G' 为 G 的无环子图
III. G' 为 G 的极小连通子图且 $V'=V$
A. I、II B. 只有 III C. II、III D. 只有 I
17. 如果具有 n 个顶点的图是一个环, 则它有 () 棵生成树。
A. n^2 B. n C. $n-1$ D. 1
18. 若一个具有 n 个顶点, e 条边的无向图是一个森林, 则该森林中必有 () 棵树。
A. n B. e C. $n-e$ D. 1

二、综合应用题

- 图 G 是一个非连通无向图, 共有 28 条边, 则该图至少有多少个顶点?
- 如何对无环有向图中的顶点号重新安排可使得该图的邻接矩阵中所有的 1 都集中到对角线以上?

5.1.3 答案与解析

一、单项选择题

1. A

参考路径的定义。

2. D

如果一个无向图有 n 个顶点和 $n-1$ 条边, 可以使它连通但没有环 (即生成树), 但再加一条

边,在不考虑重边的情形下,就必然会构成环。

3. B

强连通图是有向图,与题意矛盾,选项 A 错误;对无向连通图做一次深度优先搜索,可以访问到该连通图的所有顶点,选项 B 正确;有回路的无向图不一定是连通图,因为回路不一定包含图的所有结点,选项 C 错误;连通图可能是树,也可能存在环,选项 D 错误。

4. C

回路对应于路径,简单回路对应于简单路径,故 I 错误;稀疏图是边比较少的情況,此时用邻接矩阵必将浪费大量的空间,应选用邻接表,故 II 错误。存在回路的图不存在拓扑序列,III 正确。II 和 III 中所涉知识请参阅后面的内容。

5. C

图与树的区别是逻辑上的而不是边数的区别,图的边数也可能小于树的边数,故选项 A 错;若 E' 中的边对应的顶点不是 V' 的元素时, V' 和 $\{E'\}$ 无法构成图,故选项 B 错;无向图的极大连通子图称为连通分量,选项 C 正确;图的遍历要求每个结点只能被访问一次,且若图非连通,从某一顶点出发,无法访问到其他全部顶点,选项 D 的说法不准确。

6. A

无向连通图对应的生成树也是无向连通图,但此时边数等于顶点数减 1,故 II 错误。考虑一个无向连通图的顶点恰好构成一个回路的情况,此时,每个顶点的度都是 2,故 III 错误。在无向图中,所有顶点的度之和为边数的 2 倍,故 I 正确。

7. C

题干要求在“任何情况”下都是连通的,考虑最极端情形,即图 G 的 6 个顶点构成一个完全无向图,再加上一条边后,第 7 个顶点必然与此完全无向图构成一个连通图,所以最少边数 $=6 \times 5/2 + 1 = 16$ 。若边数 n 小于或等于 15 时,可以使这 n 条边仅连接图 G 中的某 6 个顶点,从而导致第 7 个顶点无法与这 6 个顶点构成连通图(不满足“任何情况”)。

8. C

强连通有向图的任何顶点到其他所有顶点都有路径,但未必有弧;无向图任意顶点的入度等于出度,但有向图未必满足;若边集中的某条边对应的某个顶点不在对应的顶点集中,则有向图的边集的子集和顶点集的子集无法构成子图。

9. C

邻接矩阵 A 为非对称矩阵,说明图是有向图,度为入度加出度之和。各顶点的度是矩阵中此结点对应的行(对应出度)和列(对应入度)的非零元素之和。

10. C

此题解题思路和第 7 题恰好相反。考查至少有多少个顶点,则考虑该非连通图最极端的情况,即由一个完全图加一个独立的顶点构成,因为此时若再加一条边则必然使图变成连通图。 $28 = n(n-1)/2 = 8 \times 7/2$ 条边的完全无向图中,总共有 8 个顶点构成,再加上 1 个不连通的顶点,共 9 个顶点。

11. A

对于连通无向图,边最少即构成一棵树的情形;对于强连通有向图,边最少即构成一个有向环的情形。

12. D

由于在具有 n 个顶点 e 条边的无向图中,有 $\sum_{i=1}^n TD(v_i) = 2e$,故可求得度为 2 的顶点数为 7 个,从而最多有 16 个顶点。

13. D

在有向图中, 顶点的度等于入度与出度之和。 n 个顶点的有向图中, 任一顶点最多还可以与其他 $n-1$ 个顶点有一对指向相反的边相连。

14. D

解题思路与第 7 题类似。5 个顶点构成一个完全无向图, 需要 10 条边; 再加上 1 条边后, 能保证第 6 个顶点必然与此完全无向图构成一个连通图, 故共需 11 条边。

15. B

无向图边数的两倍等于各顶点度数的总和。由于其他顶点的度均小于 3, 可以设它们的度都为 2, 设它们的数量是 x , 可列出这样的方程 $4*3+3*4+2*x=16*2$, 解得 $x=3$ 。 $4+3+3=11$, 选项 B 正确。

16. D

一个连通图的生成树是一个极小连通子图, 显然它是无环的, 故 II、III 正确。极大连通子图称为连通分量, G' 连通但非连通分量。这里再补充一下“极大连通子图”: 如果图本来就不是连通的, 那么每个子部分如果包含它本身的所有顶点和边, 那么它就是极大连通子图。

17. B

因为 n 个顶点构成的环共有 n 条边, 去掉其中任意一条便是一棵生成树, 所以共有 n 种情况。

18. C

n 个结点的树有 $n-1$ 条边, 假设森林中有 x 棵树, 将每棵树的根连到一个添加的结点, 则成为一棵树, 结点数是 $n+1$, 边数是 $e+x$, 从而可知 $x=n-e$ 。

另解: 设森林中有 x 棵树, 则再用 $x-1$ 条边就能把所有的树连接成一棵树, 此时有: 边数+1=顶点数, 即 $e+(x-1)+1=n$, 故 $x=n-e$ 。

二、综合应用题

1. 解答:

由于图 G 是一个非连通无向图, 在边数固定时, 顶点数最少的情况是该图由两个连通子图构成, 且其中之一只含一个顶点, 另一个为完全图。其中只含一个顶点的子图没有边, 另一个完全图的边数为 $n(n-1)/2=28$, 得 $n=8$ 。所以该图至少有 $1+8=9$ 个顶点。

2. 解答:

按各顶点的出度进行排序。 n 个顶点的有向图, 其顶点最大出度是 $n-1$, 最小出度为 0。这样排序后, 出度最大的顶点编号为 1, 出度最小的顶点编号为 n 。之后, 进行调整, 即只要存在弧 $\langle i, j \rangle$, 则不管顶点 j 的出度是否大于顶点 i 的出度, 都应把 i 编号在顶点 j 的编号之前, 因为只有 $i \geq j$, 弧 $\langle i, j \rangle$ 对应的 1 才能出现在邻接矩阵的上三角。

通过后面小节的学习, 采用拓扑排序并依次编号是一种更为简便的方法。

5.2 图的存储及基本操作

图的存储必须要完整、准确地反映顶点集和边集的信息。根据不同图的结构和算法, 可以采用不同的存储方式, 但不同的存储方式将对程序的效率产生相当大的影响, 因此, 所选的存储结构应适合于欲求解的问题。无论是无向图还是有向图, 主要的存储方式都有两种: 邻接矩阵和邻接表。前者属于图的顺序存储结构, 后者属于图的链接存储结构。

5.2.1 邻接矩阵法

所谓邻接矩阵存储, 就是用一个一维数组存储图中顶点的信息, 用一个二维数组存储图中边

的信息（即各顶点之间的邻接关系），存储顶点之间邻接关系的二维数组称为邻接矩阵。

结点数为 n 的图 $G=(V, E)$ 的邻接矩阵 A 是 $n \times n$ 的。将 G 的顶点编号为 v_1, v_2, \dots, v_n 。若 $(v_i, v_j) \in E$ ，则 $A[i][j]=1$ ，否则 $A[i][j]=0$ 。

$$A[i][j] = \begin{cases} 1 & \text{若}(v_i, v_j) \text{或} \langle v_i, v_j \rangle \text{是} E(G) \text{中的边} \\ 0 & \text{若}(v_i, v_j) \text{或} \langle v_i, v_j \rangle \text{不是} E(G) \text{中的边} \end{cases}$$

对于带权图而言，若顶点 v_i 和 v_j 之间有边相连，则邻接矩阵中对应项存放着该边对应的权值，若顶点 v_i 和 v_j 不相连，则用 ∞ 来代表这两个顶点之间不存在边。

$$A[i][j] = \begin{cases} w_{ij} & \text{若}(v_i, v_j) \text{或} \langle v_i, v_j \rangle \text{是} E(G) \text{中的边} \\ 0 \text{或} \infty & \text{若}(v_i, v_j) \text{或} \langle v_i, v_j \rangle \text{不是} E(G) \text{中的边} \end{cases}$$

有向图、无向图和网对应的邻接矩阵示例如图 5-5 所示。

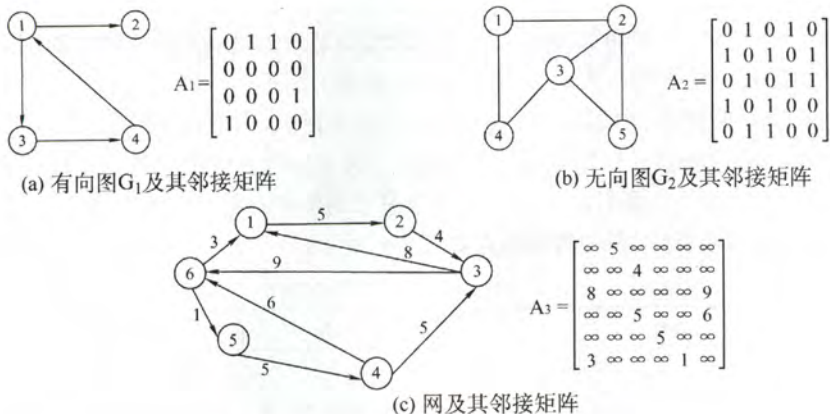


图 5-5 有向图、无向图及网的邻接矩阵

图的邻接矩阵存储结构定义如下：

```
#define MaxVertexNum 100 // 顶点数目的最大值
typedef char VertexType; // 顶点的数据类型
typedef int EdgeType; // 带权图中边上权值的数据类型
typedef struct {
    VertexType Vex[MaxVertexNum]; // 顶点表
    EdgeType Edge[MaxVertexNum][MaxVertexNum]; // 邻接矩阵，边表
    int vexnum, arcnum; // 图的当前顶点数和弧数
} MGraph;
```

注意：

① 在简单应用中，可直接用二维数组作为图的邻接矩阵（顶点信息等均可省略）。
 ② 当邻接矩阵中的元素仅表示相应的边是否存在时，EdgeType 可定义为值为 0 和 1 的枚举类型。

③ 无向图的邻接矩阵是对称矩阵，对规模特大的邻接矩阵可采用压缩存储。

④ 邻接矩阵表示法的空间复杂度为 $O(n^2)$ ，其中 n 为图的顶点数 $|V|$ 。

图的邻接矩阵存储表示法具有以下特点：

① 无向图的邻接矩阵一定是一个对称矩阵（并且唯一）。因此，在实际存储邻接矩阵时只需存储上（或下）三角矩阵的元素即可。

② 对于无向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非 ∞ 元素）的个数正好是第 i

个顶点的度 $TD(v_i)$ 。

③ 对于有向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非 ∞ 元素）的个数正好是第 i 个顶点的出度 $OD(v_i)$ （或入度 $ID(v_i)$ ）。

④ 用邻接矩阵法存储图，很容易确定图中任意两个顶点之间是否有边相连。但是，要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大。这是用邻接矩阵存储图的局限性。

⑤ 稠密图适合使用邻接矩阵的存储表示。

⑥ 设图 G 的邻接矩阵为 A ， A^n 的元素 $A^n[i][j]$ 等于由顶点 i 到顶点 j 的长度为 n 的路径的数目。该结论了解即可，证明方法请参考离散数学教材。

5.2.2 邻接表法

当一个图为稀疏图时，使用邻接矩阵表示法显然要浪费大量的存储空间。而图的邻接表法结合了顺序存储和链式存储方法，大大减少了这种不必要的浪费。

所谓邻接表就是对图 G 中的每个顶点 v_i 建立一个单链表，第 i 个单链表中的结点表示依附于顶点 v_i 的边（对于有向图则是以顶点 v_i 为尾的弧），这个单链表就称为顶点 v_i 的边表（对于有向图则称为出边表）。边表的头指针和顶点的数据信息采用顺序存储（称为顶点表），所以在邻接表中存在两种结点：顶点表结点和边表结点，如图 5-6 所示。

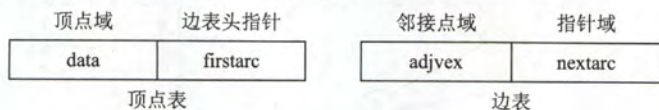


图 5-6 顶点表和边表结点结构

顶点表结点由顶点域（data）和指向第一条邻接边的指针（firstarc）构成，边表（邻接表）结点由邻接点域（adjvex）和指向下一条邻接边的指针域（nextarc）构成。

无向图和有向图的邻接表的实例分别如图 5-7 和图 5-8 所示。

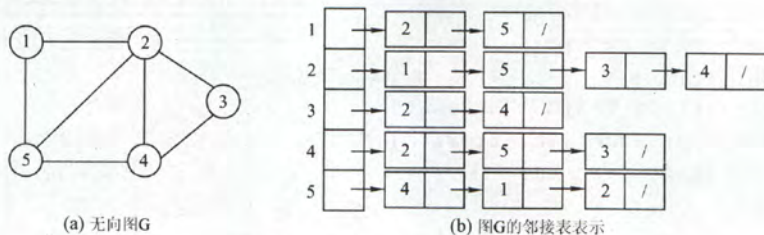


图 5-7 无向图邻接表法实例

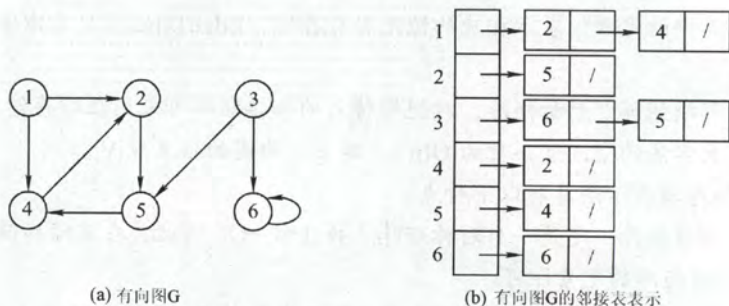


图 5-8 有向图邻接表法实例

图的邻接表存储结构定义如下:

```
#define MaxVertexNum 100           //图中顶点数目的最大值
typedef struct ArcNode{           //边表结点
    int adjvex;                   //该弧所指向的顶点的位置
    struct ArcNode *next;         //指向下一条弧的指针
    //InfoType info;             //网的边权值
}ArcNode;
typedef struct VNode{             //顶点表结点
    VertexType data;              //顶点信息
    ArcNode *first;              //指向第一条依附该顶点的弧的指针
}VNode, AdjList[MaxVertexNum];
typedef struct{
    AdjList vertices;             //邻接表
    int vexnum, arcnum;          //图的顶点数和弧数
} ALGraph;                       //ALGraph 是以邻接表存储的图类型
```

图的邻接表存储方法具有以下特点:

① 如果 G 为无向图, 则所需的存储空间为 $O(|V|+2|E|)$; 如果 G 为有向图, 则所需的存储空间为 $O(|V|+|E|)$ 。前者的倍数 2 是由于无向图中, 每条边在邻接表中出现了两次。

② 对于稀疏图, 采用邻接表表示将极大地节省存储空间。

③ 在邻接表中, 给定一顶点, 能很容易地找出它的所有邻边, 因为只需要读取它的邻接表就可以了。在邻接矩阵中, 相同的操作则需要扫描一行, 花费的时间为 $O(n)$ 。但是, 如果要确定给定的两个顶点间是否存在边, 则在邻接矩阵里可以立刻查到, 在邻接表中则需要在相应结点对应的边表中查找另一结点, 效率较低。

④ 在有向图的邻接表表示中, 求一个给定顶点的出度只需计算其邻接表中的结点个数即可; 但求其顶点的入度, 则需要遍历全部的邻接表。因此, 也有人采用逆邻接表的存储方式来加速求解给定顶点的入度。当然, 这实际上与邻接表存储方式是类似的。

⑤ 图的邻接表表示并不唯一, 这是因为在每个顶点对应的单链表中, 各边结点的链接次序可以是任意的, 取决于建立邻接表的算法以及边的输入次序。

5.2.3 十字链表

十字链表是有向图的一种链式存储结构。在十字链表中, 对应于有向图中的每条弧有一个结点, 对应于每个顶点也有一个结点。这些结点的结构如下:

弧结点					顶点结点		
tailvex	headvex	hlink	tlink	info	data	firstin	firstout

弧结点中有 5 个域: 其中尾域 (tailvex) 和头域 (headvex) 分别指示弧尾和弧头这两个顶点在图中的位置, 链域 hlink 指向弧头相同的下一条弧, 链域 tlink 指向弧尾相同的下一条弧, info 域指向该弧的相关信息。这样, 弧头相同的弧在同一个链表上, 弧尾相同的弧也在同一个链表上。

顶点结点中有 3 个域: data 域存放顶点相关的数据信息, 如顶点名称, firstin 和 firstout 两个域分别指向以该顶点为弧头或弧尾的第一个弧结点。

图 5-9 为有向图的十字链表表示法。注意, 顶点结点之间是顺序存储。

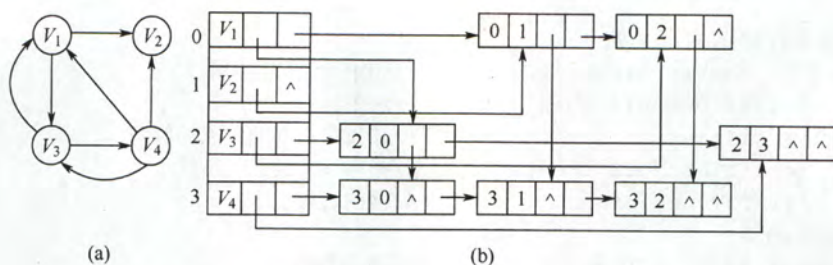


图 5-9 有向图的十字链表表示

图的十字链表存储结构定义如下:

```
#define MaxVertexNum 100           //图中顶点数目的最大值
typedef struct ArcNode{             //边表结点
    int tailvex, headvex;           //该弧的头尾结点
    struct ArcNode *hlink, *tlink;  //分别指向弧头相同和弧尾相同的结点
    //InfoType info;               //相关信息指针
}ArcNode;
typedef struct VNode{              //顶点表结点
    VertexType data;               //顶点信息
    ArcNode *firstin, *firstout;    //指向第一条入弧和出弧
}VNode;
typedef struct{
    VNode xlist[MaxVertexNum];     //邻接表
    int vexnum, arcnum;            //图的顶点数和弧数
} GLGraph;                         //GLGraph 是以十字邻接存储的图类型
```

在十字链表中,既容易找到 v_i 为尾的弧,也容易找到 v_i 为头的弧,因而容易求得顶点的出度和入度。

图的十字链表表示是不唯一的,但一个十字链表表示确定一个图。

5.2.4 邻接多重表

邻接多重表是无向图的另一种链式存储结构。

在邻接表中,容易求得顶点和边的各种信息,但在邻接表中求两个顶点之间是否存在边,或需要对边执行删除等操作时,需要分别在两个顶点的边表中遍历,效率较低。

与十字链表类似,在邻接多重表中,每一条边用一个结点表示,其结构如下图。

mark	ivex	ilink	ivex	jlink	info
------	------	-------	------	-------	------

其中,mark 为标志域,可用以标记该条边是否被搜索过;ivex 和 jvex 为该边依附的两个顶点在图中的位置;ilink 指向下一条依附于顶点 ivex 的边;jlink 指向下一条依附于顶点 jvex 的边,info 为指向和边相关的各种信息的指针域。

每一个顶点也用一個结点表示,它由如下所示的两个域组成。

data	firstedge
------	-----------

其中,data 域存储该顶点的相关信息,firstedge 域指示第一条依附于该顶点的边。

在邻接多重表中,所有依附于同一顶点的边串联在同一链表中,由于每条边依附于两个顶点,则每个边结点同时链接在两个链表中。

图 5-10 为无向图的邻接多重表表示法。注意，每条边只有一个结点。

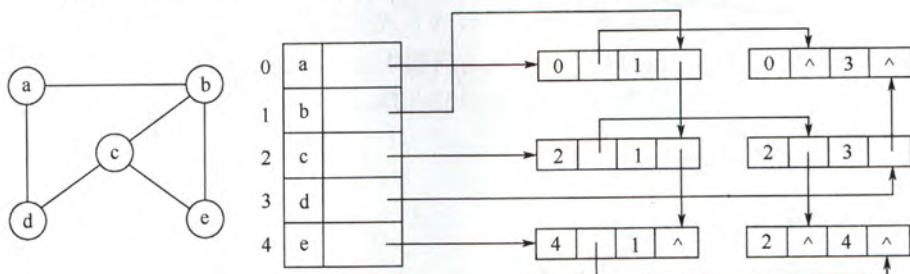


图 5-10 无向图的邻接多重表表示

图的邻接多重表存储结构定义如下：

```
#define MaxVertexNum 100 // 图中顶点数目的最大值
typedef struct ArcNode{ // 边表结点
    bool mark; // 访问标记
    int ivex, jvex; // 分别指向该弧的两个结点
    struct ArcNode *ilink, *jlink; // 分别指向两个顶点的下一条边
    //InfoType info; // 相关信息指针
}ArcNode;
typedef struct VNode{ // 顶点表结点
    VertexType data; // 顶点信息
    ArcNode *firstedge; // 指向第一条依附该顶点的边
}VNode;
typedef struct{
    VNode adjmulist[MaxVertexNum]; // 邻接表
    int vexnum, arcnum; // 图的顶点数和弧数
} AMLGraph; // AMLGraph 是以邻接多重表存储的图类型
```

5.2.5 图的基本操作

图的基本操作是独立于图的存储结构的。而对于不同的存储方式，操作算法的具体实现会有着不同的性能。请读者根据上述的存储方式，考虑如下具体算法如何实现，以及采用何种存储方式的算法效率会更高。

图的基本操作主要包括（仅抽象地考虑，故忽略掉各变量的类型）：

Adjacent(G,x,y): 判断图 G 是否存在边 $\langle x, y \rangle$ 或 (x, y) 。

Neighbors(G,x): 列出图 G 中与结点 x 邻接的边。

InsertVertex(G,x): 在图 G 中插入顶点 x。

DeleteVertex(G,x): 从图 G 中删除顶点 x。

AddEdge(G,x,y): 如果无向边 (x, y) 或有向边 $\langle x, y \rangle$ 不存在，则向图 G 中添加该边。

RemoveEdge(G,x,y): 如果无向边 (x, y) 或有向边 $\langle x, y \rangle$ 存在，则从图 G 中删除该边。

FirstNeighbor(G,x): 求图 G 中顶点 x 的第一个邻接点，若有则返回顶点号。若 x 没有邻接点或图中不存在 x，则返回 -1。

NextNeighbor(G,x,y): 假设图 G 中顶点 y 是顶点 x 的一个邻接点，返回除 y 之外顶点 x 的下一个邻接点的顶点号，若 y 是 x 的最后一个邻接点，则返回 -1。

Get_edge_value(G,x,y): 获取图 G 中边 (x, y) 或 <x, y> 对应的权值。

Set_edge_value(G,x,y,v): 设置图 G 中边 (x, y) 或 <x, y> 对应的权值为 v。

此外还有图的遍历算法: 按照某一种方式访问图中每一个顶点且仅访问一次。图的遍历算法包括深度优先遍历和广度优先遍历, 具体见下一节内容。

5.2.6 本节试题精选

一、单项选择题

- 关于图的存储结构, () 是错误的。
 - 使用邻接矩阵存储一个图时, 在不考虑压缩存储的情况下, 所占用的存储空间大小只与图中的顶点数有关, 与边数无关
 - 邻接表只用于有向图的存储, 邻接矩阵适用于有向图和无向图
 - 若一个有向图的邻接矩阵, 对角线以下元素为 0, 则该图的拓扑序列必定存在
 - 存储无向图的邻接矩阵是对称的, 故只需存储邻接矩阵的下 (或上) 三角部分即可
- 若图的邻接矩阵中主对角线上的元素皆为 0, 其余元素全为 1, 则可以断定该图一定 ()。
 - 是无向图
 - 是有向图
 - 是完全图
 - 不是带权图
- 在含有 n 个顶点和 e 条边的无向图的邻接矩阵中, 零元素的个数为 ()。
 - e
 - $2e$
 - $n^2 - e$
 - $n^2 - 2e$
- 带权有向图 G 用邻接矩阵存储, 则 v_i 的入度等于邻接矩阵中 ()。
 - 第 i 行非 ∞ 的元素个数
 - 第 i 列非 ∞ 的元素个数
 - 第 i 行非 ∞ 且非 0 的元素个数
 - 第 i 列非 ∞ 且非 0 的元素个数
- 一个有 n 个顶点的图用邻接矩阵 A 表示, 若图为有向图时, 顶点 v_i 的入度是 () ; 若图为无向图时, 顶点 v_i 的度是 ()。
 - $\sum_{i=1}^n A[i][j]$
 - $\sum_{j=1}^n A[j][i]$
 - $\sum_{i=1}^n A[j][i]$
 - $\sum_{j=1}^n A[j][i]$, 或 $\sum_{j=1}^n A[i][j]$
- 下列哪一种图的邻接矩阵是对称矩阵 ()。
 - 有向网
 - 无向网
 - AOV 网
 - AOE 网
- 从邻接矩阵 $A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ 可以看出, 该图共有 (①) 个顶点; 如果是有向图该图共有 (②) 条弧; 如果是无向图, 则共有 (③) 条边。
 - ①. A. 9 B. 3 C. 6 D. 1 E. 以上答案均不正确
 - ②. A. 5 B. 4 C. 3 D. 2 E. 以上答案均不正确
 - ③. A. 5 B. 4 C. 3 D. 2 E. 以上答案均不正确
- 以下关于图的存储结构的叙述中正确的是 ()。
 - 一个图的邻接矩阵表示唯一, 邻接表表示唯一
 - 一个图的邻接矩阵表示唯一, 邻接表表示不唯一
 - 一个图的邻接矩阵表示不唯一, 邻接表表示唯一
 - 一个图的邻接矩阵表示不唯一, 邻接表表示不唯一
- 用邻接表法存储图所用的空间大小 ()。
 - 与图的顶点数和边数有关
 - 只与图的边数有关

- C. 只与图的顶点数有关 D. 与边数的平方有关
10. 若邻接表中有奇数个边表结点, 则一定是 ()。
- A. 图中有奇数个结点 B. 图中有偶数个结点
C. 图为无向图 D. 图为有向图
11. 在有向图的邻接表存储结构中, 顶点 v 在边表中出现的次数是 ()。
- A. 顶点 v 的度 B. 顶点 v 的出度 C. 顶点 v 的入度 D. 依附于顶点 v 的边数
12. n 个顶点的无向图的邻接表最多有 () 个边表结点。
- A. n^2 B. $n(n-1)$ C. $n(n+1)$ D. $n(n-1)/2$
13. 假设有 n 个顶点 e 条边的有向图用邻接表表示, 则删除与某个顶点 v 相关的所有边的时间复杂度为 ()。
- A. $O(n)$ B. $O(e)$ C. $O(n+e)$ D. $O(ne)$
14. 对邻接表的叙述中, () 是正确的。
- A. 无向图的邻接表中, 第 i 个顶点的度为第 i 个链表中结点个数的两倍
B. 邻接表比邻接矩阵的操作更简便
C. 邻接矩阵比邻接表的操作更简便
D. 求有向图结点的度, 必须遍历整个邻接表
15. 邻接多重表是 () 的存储结构。
- A. 无向图 B. 有向图 C. 无向图和有向图 D. 都不是
16. 十字链表是 () 的存储结构。
- A. 无向图 B. 有向图 C. 无向图和有向图 D. 都不是

二、综合应用题

1. 已知带权有向图 G 的邻接矩阵如图 5-11 所示, 请画出该带权有向图 G 。

0	15	2	12	∞	∞	∞
∞	0	∞	∞	6	∞	∞
∞	∞	0	∞	8	4	∞
∞	∞	∞	0	∞	∞	3
∞	∞	∞	∞	0	∞	9
∞	∞	∞	5	∞	0	10
∞	4	∞	∞	∞	∞	0

图 5-11 图 G 的邻接矩阵

2. 设图 $G=(V, E)$ 以邻接表存储, 如图 5-12 所示。画出其邻接矩阵存储以及图 G 。

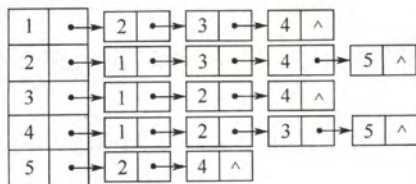
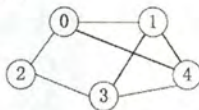


图 5-12 图 G 的邻接表

3. 对 n 个顶点的无向图和有向图, 分别采用邻接矩阵和邻接表表示时, 试问:
- 1) 如何判别图中有多少条边?
- 2) 如何判别任意两个顶点 i 和 j 是否有边相连?

- 3) 任意一个顶点的度是多少?
 4. 写出从图的邻接表表示转换成邻接矩阵表示的算法。
 5. 【2015年计算机联考真题】
 已知含有5个顶点的图G如下图所示。



请回答下列问题:

- 1) 写出图G的邻接矩阵A(行、列下标从0开始)。
 2) 求 A^2 , 矩阵 A^2 中位于0行3列元素值的含义是什么?
 3) 若已知具有 n ($n \geq 2$) 个顶点的图的邻接矩阵为B, 则 B^m ($2 \leq m \leq n$) 中非零元素的含义是什么?

5.2.7 答案与解析

一、单项选择题

1. B

n 个顶点的图, 若采用邻接矩阵表示, 不考虑压缩存储, 则存储空间大小为 $O(n^2)$, A 正确。邻接表可以用于存储无向图, 只是把每条边都看成是两条方向相反的有向边, 因此需要存储两次, B 错误。由于邻接矩阵中对角线以下的元素全为 0, 若存在 $\langle i, j \rangle$, 则必然 $i < j$, 由传递性可知图中路径的顶点编号是依次递增的, 假设存在环 $k \rightarrow \dots \rightarrow j \rightarrow k$, 由题设可知 $k < j < k$, 矛盾, 故不存在环, 拓扑序列必定存在, C 正确。选项 D 显然正确。

注意: 若邻接矩阵对角线以下(或以上)的元素全为 0, 则图中必然不存在环, 即拓扑序列一定存在(结合第 1 节综合题 2), 但这并不能说明拓扑序列是唯一的。

2. C

除主对角线上的元素, 其余元素全为 1, 则说明任意两个顶点之间都有边相连, 则该图一定是完全图。

3. D

无向图的邻接矩阵中, 矩阵大小为 n^2 , 非零元素的个数为 $2e$, 故零元素的个数为 $n^2 - 2e$ 。读者应掌握此题的变形, 当无向图变为有向图时, 能够求出零的个数和非零的个数。

4. D

有向图的邻接矩阵中, 0 和 ∞ 表示的都不是有向边, 而入度是由邻接矩阵的列中元素计算出来的; 出度是由邻接矩阵的行中元素计算出来的。

5. B、D

有向图的入度是其第 i 列的非 0 元素之和, 无向图的度是第 i 行或第 i 列中非 0 元素之和。

6. B

无向图的邻接矩阵存储中, 每条边存储两次, 且 $A[i][j] = A[j][i]$ 。

7. B、B、D

邻接矩阵的顶点数等于矩阵的行(列)数, 有向图的边数等于矩阵中非零元素的个数, 无向图的边数等于矩阵中非零元素个数的一半。

注意: 本题中所给的矩阵为对称矩阵, 若不是对称矩阵则必然不可能是无向图。

8. B

邻接矩阵表示唯一是因为图中边的信息在矩阵中有确定的位置,邻接表不唯一是因为邻接表的建立取决于读入边的顺序和边表中的插入算法。

9. A

邻接表存储时,顶点数 n 决定了顶点表的大小,边数 e 决定了边表的个数,且每条边存储两次,总存储空间为 $O(n+2e)$, 故选 A。而邻接矩阵只与图的顶点数有关,为 $O(n^2)$ 。

10. D

无向图采用邻接表表示时,每条边存储两次,所以其边表结点个数为偶数。题中边表结点为奇数个,故必然是有向图,且有奇数条边。

11. C

题中的边表是不包括顶点表的。因为任何顶点 u 对应的边表里存放的都是以 u 为起点的边所对应的另一个顶点 v 。从而 v 在边表中出现的次数也就是它的入度。

12. B

n 个顶点的无向图最多有 $n(n-1)/2$ 条边,每条边在邻接表中存储两次,所以边表结点最多为 $n(n-1)$ 个。

13. C

删除与某顶点 v 相关的所有边过程如下:先删除下标为 v 的顶点表结点的单链表,出边数最多为 $n-1$,对应时间复杂度为 $O(n)$,再扫描所有边表结点,删除所有的顶点 v 的入边,对应时间复杂度为 $O(e)$ 。故总的时间复杂度为 $O(n+e)$ 。

14. D

无向图的邻接表中,第 i 个顶点的度为第 i 个链表中结点数,故 A 错。邻接表和邻接矩阵对于不同的操作各有优势,B 和 C 都不准确。有向图结点的度包括出度和入度,对于出度,需要遍历顶点表结点所对应的边表;对于入度,则需要遍历剩下全部的边表,故 D 正确。

15. A

邻接多重表是无向图的存储结构,选 A。

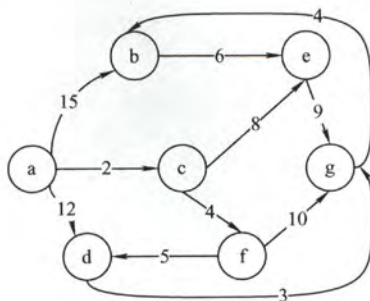
16. B

十字链表是有向图的存储结构,选 B。

二、综合应用题

1. 解答:

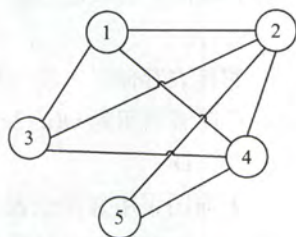
带权有向图 G 如下图所示。



2. 解答:

其邻接矩阵存储如下所示。

0	1	1	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	0	1
0	1	0	1	0



第 2 题图

在邻接表中，每条边存储了 2 次，在没有特殊指明的情况下，通常默认其为无向图（当然，无向图也可以看做是具有对边的有向图）。该邻接表对应的图 G 如第 2 题图所示。

3. 解答：

1) 对于邻接矩阵表示的无向图，边数等于矩阵中 1 的个数除以 2；对于邻接表表示的无向图，边数等于边结点的个数除以 2。对于邻接矩阵表示的有向图，边数等于矩阵中 1 的个数；对于邻接表表示的有向图，边数等于边结点的个数。

2) 对于邻接矩阵表示的无向图或有向图，任意两个顶点 i 和 j ，邻接矩阵 $\text{arcs}[i][j]$ 或 $\text{arcs}[j][i]$ 为 1 表示有边相连；否则为无边相连。对于邻接表表示的无向图或有向图，任意两个顶点 i 和 j ，若从顶点表结点 i 出发找到编号为 j 的边表结点或从顶点表结点 j 出发找到编号为 i 的边表结点，表示有边相连；否则为无边相连。

3) 对于邻接矩阵表示的无向图，顶点 i 的度等于第 i 行中 1 的个数；对于邻接矩阵表示的有向图，顶点 i 的出度等于第 i 行中 1 的个数；入度等于第 i 列中 1 的个数；度数等于它们之和。对于邻接表表示的无向图，顶点 i 的度等于顶点表结点 i 的单链表中边表结点的个数；对于邻接表表示的有向图，顶点 i 的出度等于顶点表结点 i 的单链表中边表结点的个数，顶点 i 的入度等于邻接表中所有编号为 i 的边表结点数；度数等于入度和出度之和。

4. 解答：

算法的基本思想：设图的顶点分别存储在 $v[n]$ 数组中。首先初始化邻接矩阵。遍历邻接表，在依次遍历顶点 $v[i]$ 的边链表时，修改邻接矩阵的第 i 行的元素值。若链表边结点的值为 j ，则置 $\text{arcs}[i][j]=1$ 。遍历完邻接表时，整个转换过程结束。此算法对于无向图、有向图均适用。

算法的实现如下：

```
void Convert(ALGraph &G,int arcs[M][N]){
    //此算法将邻接表方式表示的图 G 转换为邻接矩阵 arcs
    for(i=0;i<n;i++){           //依次遍历各顶点表结点为头的边链表
        p=(G->v[i]).firstarc;    //取出顶点 i 的第一条出边
        while(p!=NULL){          //遍历边链表
            arcs[i][p->data]=1;
            p=p->nextarc;         //取下一条出边
        }                        //while
    }                             //for
}
```

5. 解答：

考查图的邻接矩阵的性质。

1) 图 G 的邻接矩阵 A 如下：

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

2) A^2 如下:

$$A^2 = \begin{bmatrix} 3 & 1 & 0 & 3 & 1 \\ 1 & 3 & 2 & 1 & 2 \\ 0 & 2 & 2 & 0 & 2 \\ 3 & 1 & 0 & 3 & 1 \\ 1 & 2 & 2 & 1 & 3 \end{bmatrix}$$

0 行 3 列的元素值 3 表示从顶点 0 到顶点 3 之间长度为 2 的路径共有 3 条。

3) B^m ($2 \leq m \leq n$) 中位于 i 行 j 列 ($0 \leq i, j \leq n-1$) 的非零元素的含义是: 图中从顶点 i 到顶点 j 长度为 m 的路径条数。

5.3 图的遍历

图的遍历是指从图中的某一顶点出发, 按照某种搜索方法沿着图中的边对图中的所有顶点访问一次且仅访问一次。注意到树是一种特殊的图, 所以树的遍历实际上也可以看作是一种特殊的图的遍历。图的遍历是图的一种最基本的操作, 其他许多操作都建立在图的遍历操作基础之上。

图的遍历主要有两种算法: 广度优先搜索和深度优先搜索。包括广度优先搜索和深度优先搜索在内的几乎所有图的搜索算法都可以抽象为优先级搜索, 或最佳优先搜索。广度优先搜索会优先考虑最早被发现的顶点, 也就说离起点越近的顶点优先级越高。深度优先搜索会优先考虑最后被发现的顶点。广度优先搜索算法由 Edward F. Moore 在研究迷宫路径问题时发现; 深度优先搜索在 20 世纪 50 年代晚期获得广泛使用, 尤其是在人工智能方面。

5.3.1 广度优先搜索 (Breadth-First-Search, BFS)

广度优先搜索 (BFS) 类似于二叉树的层序遍历算法, 它的基本思想是: 首先访问起始顶点 v , 接着由 v 出发, 依次访问 v 的各个未访问过的邻接顶点 w_1, w_2, \dots, w_i , 然后再依次访问 w_1, w_2, \dots, w_i 的所有未被访问过的邻接顶点; 再从这些访问过的顶点出发, 再访问它们所有未被访问过的邻接顶点……依次类推, 直到图中所有顶点都被访问过为止。类似的思想还将应用于 Dijkstra 单源最短路径算法和 Prim 最小生成树算法。

广度优先搜索是一种分层的查找过程, 每向前走一步可能访问一批顶点, 不像深度优先搜索那样有往回退的情况, 因此它不是一个递归的算法。为了实现逐层的访问, 算法必须借助一个辅助队列, 以记忆正在访问的顶点的下一层顶点。

广度优先搜索算法的伪代码如下:

```
bool visited[MAX_VERTEX_NUM]; // 访问标记数组
void BFSTraverse(Graph G) {
```



```

//对图 G 进行广度优先遍历, 设访问函数为 visit()
for(i=0;i<G.vexnum;++i)
    visited[i]=FALSE;           //访问标记数组初始化
InitQueue(Q);                   //初始化辅助队列 Q
for(i=0;i<G.vexnum;++i)        //从 0 号顶点开始遍历
    if(!visited[i])             //对每个连通分量调用一次 BFS
        BFS(G,i);               //vi 未访问过, 从 vi 开始 BFS
}

void BFS(Graph G,int v){
//从顶点 v 出发, 广度优先遍历图 G, 算法借助一个辅助队列 Q
    visit(v);                   //访问初始顶点 v
    visited[v]=TRUE;            //对 v 做已访问标记
    Enqueue(Q,v);               //顶点 v 入队列
    while(!isEmpty(Q)){
        Dequeue(Q,v);           //顶点 v 出队列
        for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w))
            //检测 v 所有邻接点
            if(!visited[w]){     //w 为 v 的尚未访问的邻接点
                visit(w);        //访问顶点 w
                visited[w]=TRUE; //对 w 做已访问标记
                Enqueue(Q,w);     //顶点 w 入队列
            } //if
    } //while
}

```

辅助数组 `visited[]` 标志顶点是否被访问过, 它的初始状态为 `FALSE`。在图的遍历过程中, 一旦某一个顶点 v_i 被访问, 就立即置 `visited[i]` 为 `TRUE`, 防止它被多次访问。

下面通过实例, 演示广度优先搜索的过程, 给定图 G 如图 5-13 所示。

假设从 a 结点开始访问, a 先入队。此时队列非空, 取出队头元素 a , 由于 b 、 c 与 a 邻接且未被访问过, 于是依次访问 b 、 c , 并将 b 、 c 依次入队。队列非空, 取出队头元素 b , 依次访问与 b 邻接且未被访问的顶点 d 、 e , 并将 d 、 e 入队 (注意: a 与 b 也邻接, 但 a 已置访问标记, 故不再重复访问)。此时队列非空, 取出队头元素 c , 访问与 c 邻接且未被访问的顶点 f 、 g , 并将 f 、 g 入队。此时, 取出队头元素 d , 但与 d 邻接且未被访问的顶点为空, 故而不进行任何操作。继续取出队头元素 e , 将 h 入队列……当最后取出队头元素 h 后, 队列为空, 从而循环自动跳出。遍历结果为 `abcdefgh`。

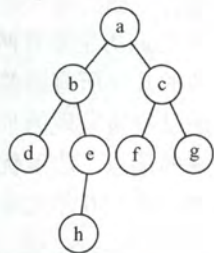


图 5-13 一个无向图 G

从上例不难看出, 图的广度优先搜索的过程与二叉树的层序遍历是完全一致的, 这也说明了图的广度优先搜索遍历算法是二叉树的层次遍历算法的扩展。

图的广度优先遍历还可以用在求一些问题的最优解上, 不过初试方面很难涉及, 有兴趣的同学可以参考《王道考研系列: 计算机考研——机试指南》。

1. BFS 算法的性能分析

无论是邻接表还是邻接矩阵的存储方式, BFS 算法都需要借助一个辅助队列 Q , n 个顶点均

需入队一次,在最坏的情况下,空间复杂度为 $O(|V|)$ 。

当采用邻接表存储方式时,每个顶点均需搜索一次(或入队一次),故时间复杂度为 $O(|V|)$,在搜索任一顶点的邻接点时,每条边至少访问一次,故时间复杂度为 $O(|E|)$,算法总的时间复杂度为 $O(|V|+|E|)$ 。当采用邻接矩阵存储方式时,查找每个顶点的邻接点所需时间为 $O(|V|)$,故算法总的时间复杂度为 $O(|V|^2)$ 。

2. BFS 算法求解单源最短路径问题

如果图 $G=(V, E)$ 为非带权图,定义从顶点 u 到顶点 v 的最短路径 $d(u, v)$ 为从 u 到 v 的任何路径中最少的边数;如果从 u 到 v 没有通路,则 $d(u, v)=\infty$ 。

使用 BFS,我们可以求解一个满足上述定义的非带权图的单源最短路径问题,这是由广度优先搜索总是按照距离由近到远来遍历图中每个顶点的性质决定的。

BFS 算法求解单源最短路径问题的算法如下:

```
void BFS_MIN_Distance(Graph G, int u) {
    //d[i]表示从u到i结点的最短路径
    for(i=0; i<G.vexnum; ++i)
        d[i]=∞; //初始化路径长度
    visited[u]=TRUE; d[u]=0;
    EnQueue(Q, u);
    while(!isEmpty(Q)) { //BFS 算法主过程
        DeQueue(Q, u); //队头元素u出队
        for(w=FirstNeighbor(G, u); w>=0; w=NextNeighbor(G, u, w))
            if(!visited[w]) { //w为u的尚未访问的邻接顶点
                visited[w]=TRUE; //设已访问标记
                d[w]=d[u]+1; //路径长度加1
                EnQueue(Q, w); //顶点w入队
            } //if
    } //while
}
```

3. 广度优先生成树

在广度遍历的过程中,我们可以得到一棵遍历树,称为广度优先生成树,如图 5-14 所示。需要注意的是,一给定图的邻接矩阵存储表示是唯一的,故其广度优先生成树也是唯一的,但由于邻接表存储表示不唯一,故其广度优先生成树也是不唯一的。

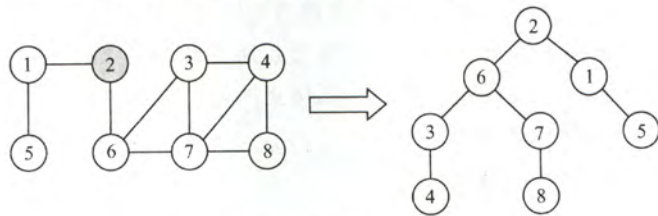


图 5-14 图的广度优先生成树

5.3.2 深度优先搜索 (Depth-First-Search, DFS)

与广度优先搜索不同,深度优先搜索 (DFS) 类似于树的先序遍历。正如其名称中所暗示的意思一样,这种搜索算法所遵循的搜索策略是尽可能“深”地搜索一个图。它的基本思想如下:

首先访问图中某一起始顶点 v ，然后由 v 出发，访问与 v 邻接且未被访问的任一顶点 w_1 ，再访问与 w_1 邻接且未被访问的任一顶点 w_2 ，……重复上述过程。当不能再继续向下访问时，依次退回到最近被访问的顶点，若它还有邻接顶点未被访问过，则从该点开始继续上述搜索过程，直到图中所有顶点均被访问过为止。

一般情况下，其递归形式的算法十分简洁。下面描述其算法过程。

```
bool visited[MAX_VERTEX_NUM]; //访问标记数组
void DFSTraverse(Graph G){
//对图 G 进行深度优先遍历，访问函数为 visit()
    for(v=0;v<G.vexnum;++v)
        visited[v]=FALSE; //初始化已访问标记数据
    for(v=0;v<G.vexnum;++v) //本代码中是从 v=0 开始遍历
        if(!visited[v])
            DFS(G,v);
}
void DFS(Graph G,int v){
//从顶点 v 出发，采用递归思想，深度优先遍历图 G
    visit(v); //访问顶点 v
    visited[v]=TRUE; //设已访问标记
    for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w))
        if(!visited[w]){ //w 为 u 的尚未访问的邻接顶点
            DFS(G,w);
        } //if
}
```

以图 5-13 所示的无向图为例，演示深度优先搜索的过程：

首先访问 a ，并置 a 已访问标记；然后访问与 a 邻接且未被访问的顶点 b ，置 b 已访问标记；然后访问与 b 邻接且未被访问的顶点 d ，置 d 已访问标记。此时 d 已没有未被访问过的邻接点，故返回上一个访问过的顶点 b ，访问与其邻接且未被访问的顶点 e ，置 e 访问标记……。依次类推，直到图中所有的顶点访问一次且仅访问一次。遍历结果为 $abdehcfg$ 。

注意：图的邻接矩阵表示是唯一的，但对于邻接表来说，如果边的输入次序不同，生成的邻接表也不同。因此，对于同样一个图，基于邻接矩阵的遍历所得到的 DFS 序列和 BFS 序列是唯一的，基于邻接表的遍历所得到的 DFS 序列和 BFS 序列是不唯一的。

1. DFS 算法的性能分析

DFS 算法是一个递归算法，需要借助一个递归工作栈，故它的空间复杂度为 $O(|V|)$ 。

遍历图的过程实质上是对每个顶点查找其邻接点的过程，其耗费的时间取决于所采用的存储结构。当以邻接矩阵表示时，查找每个顶点的邻接点所需时间为 $O(|V|)$ ，故总的时间复杂度为 $O(|V|^2)$ 。当以邻接表表示时，查找所有顶点的邻接点所需时间为 $O(|E|)$ ，访问顶点所需时间为 $O(|V|)$ ，此时，总的时间复杂度为 $O(|V|+|E|)$ 。

2. 深度优先的生成树和生成森林

与广度优先搜索一样，深度优先搜索也会产生一棵深度优先生成树。当然，这是有条件的，即对连通图调用 DFS 才可以产生深度优先生成树，否则产生的将是深度优先生成森林，如图 5-15 所示。和 BFS 类似，基于邻接表存储的深度优先生成树是不唯一的。



图 5-15 图的深度优先生成森林

5.3.3 图的遍历与图的连通性

图的遍历算法可以用来判断图的连通性。

对于无向图来说，如果无向图是连通的，则从任一结点出发，仅需一次遍历就能够访问图中所有顶点；如果无向图是非连通的，则从某一个顶点出发，一次遍历只能访问到该顶点所在连通分量的所有顶点，而对于图中其他连通分量的顶点，则无法通过这次遍历访问。对于有向图来说，若从初始点到图中的每个顶点都有路径，则能够访问到图中的所有顶点，否则不能访问到所有顶点。

故而在 `BFS_Traverse()` 或 `DFS_Traverse()` 中添加了第二个 `for` 循环，再选取初始点，继续进行遍历，以防止一次无法遍历图的所有顶点。对于无向图，上述两个函数调用 `BFS(G, i)` 或 `DFS(G, i)` 的次数等于该图的连通分量数；而对于有向图，则不是这样，因为一个连通的有向图分为强连通的和非强连通的，它的连通子图也分为强连通分量和非强连通分量，非强连通分量一次调用 `BFS(G, i)` 或 `DFS(G, i)` 无法访问到该连通分量的所有顶点，如图 5-16 所示。



图 5-16 有向图的非强连通分量

5.3.4 本节试题精选

一、单项选择题

- 下列关于广度优先算法的说法正确的是 ()。
 - 当各边的权值相等时，广度优先算法可以解决单源最短路径问题
 - 当各边的权值不等时，广度优先算法可用来解决单源最短路径问题
 - 广度优先遍历算法类似于树中的后序遍历算法
 - 实现图的广度优先算法时，使用的数据结构是队列

A. I、IV B. II、III、IV C. II、IV D. I、III、IV
- 对于一个非连通无向图 G ，采用深度优先遍历访问所有顶点，在 `DFS_Traverse` 函数（见考点讲解 DFS 部分）中调用 `DFS` 的次数正好等于 ()。

A. 顶点数 B. 边数 C. 连通分量数 D. 不确定
- 对一个有 n 个顶点 e 条边的图采用邻接表表示时，进行 DFS 遍历的时间复杂度为 ()，空间复杂度为 ()；进行 BFS 遍历的时间复杂度为 ()，空间复杂度为 ()。

A. $O(n)$ B. $O(e)$ C. $O(n+e)$ D. $O(1)$
- 对于有 n 个顶点 e 条边的图采用邻接矩阵表示时，进行 DFS 遍历的时间复杂度为 ()；进行 BFS 遍历的时间复杂度为 ()。

A. $O(n^2)$ B. $O(e)$ C. $O(n+e)$ D. $O(e^2)$
- 无向图 $G=(V, E)$ ，其中： $V=\{a, b, c, d, e, f\}$ ， $E=\{(a, b), (a, e), (a, c), (b, e), (c, f), (f, d), (e, d)\}$ ，对该图从 a 开始进行深度优先遍历，得到的顶点序列正确的是 ()。

A. a, b, e, c, d, f B. a, c, f, e, b, d
C. a, e, b, c, f, d D. a, e, d, f, c, b

6. 如图 5-17 所示, 在下面的 5 个序列中, 符合深度优先遍历的序列有多少个 ()。

1、aebfdc 2、acfdbe 3、aedfcb 4、aefdbc 5、aecfdb

A. 5

B. 4

C. 3

D. 2

7. 用邻接表存储的图的深度优先遍历算法类似于树的 (), 而其广度优先遍历算法类似于树的 ()。

A. 中序遍历

B. 先序遍历

C. 后序遍历

D. 按层次遍历

8. 一个有向图 G 的邻接表存储如图 5-18 所示, 从顶点 1 出发, 对图 G 调用深度优先遍历所得顶点序列是 (); 按广度优先遍历所得顶点序列是 ()。

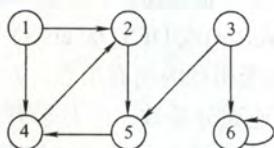
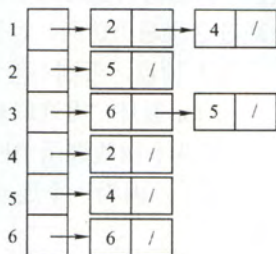


图 5-18 题 8 图

A. 125436

B. 124536

C. 124563

D. 362514

9. 无向图 $G=(V, E)$, 其中 $V=\{a, b, c, d, e, f\}$, $E=\{(a, b), (a, e), (a, c), (b, e), (c, f), (f, d), (e, d)\}$ 。对该图进行深度优先遍历, 下面不能得到的序列是 ()。

A. acfdeb

B. aebdfc

C. aedfcb

D. abecdff

10. 【2013 年计算机联考真题】

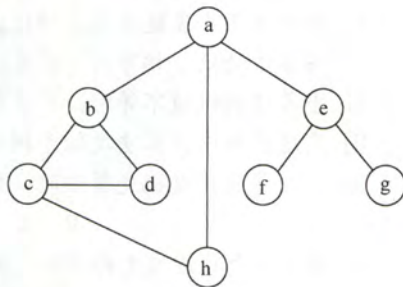
若对如下无向图进行遍历, 则下列选项中, 不是广度优先遍历序列的是 ()。

A. h, c, a, b, d, e, g, f

B. e, a, f, g, b, h, c, d

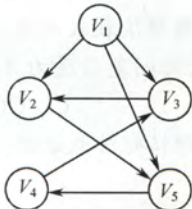
C. d, b, c, a, h, e, f, g

D. a, b, c, d, h, e, f, g



11. 【2016 年计算机联考真题】

下列选项中, 不是下图深度优先搜索序列的是 ()。



A. V_1, V_5, V_4, V_3, V_2

B. V_1, V_3, V_2, V_5, V_4

C. V_1, V_2, V_5, V_4, V_3

D. V_1, V_2, V_3, V_4, V_5

12. 判断有向图中是否存在回路, 除了可以利用拓扑排序外, 还可以利用 ()。

A. 求关键路径的方法

B. 求最短路径的 Dijkstra 算法

C. 深度优先遍历算法

D. 广度优先遍历算法

13. 使用 DFS 算法递归地遍历一个无环有向图,并在退出递归时输出相应顶点,这样得到的顶点序列是()。

A. 逆拓扑有序

B. 拓扑有序

C. 无序的

D. 都不是

14. 设无向图 $G=(V, E)$ 和 $G'=(V', E')$, 如果 G' 是 G 的生成树, 则下列说法中错误的是()。

A. G' 为 G 的子图B. G' 为 G 的连通分量C. G' 为 G 的极小连通子图且 $V=V'$ D. G' 是 G 的一个无环子图

15. 图的广度优先生成树的树高比深度优先生成树的树高()。

A. 小或相等

B. 小

C. 大或相等

D. 大

16. 【2010 年计算机联考真题】

对有 n 个顶点、 e 条边且使用邻接表存储的有向图进行广度优先遍历,其算法的时间复杂度是()。

A. $O(n)$ B. $O(e)$ C. $O(n+e)$ D. $O(n*e)$

17. 【2015 年计算机联考真题】

设有向图 $G=(V, E)$, 顶点集 $V=\{V_0, V_1, V_2, V_3\}$, 边集 $E=\{\langle V_0, V_1 \rangle, \langle V_0, V_2 \rangle, \langle V_0, V_3 \rangle, \langle V_1, V_3 \rangle\}$ 。若从顶点 V_0 开始对图进行深度优先遍历,则可能得到的不同遍历序列个数是()。

A. 2

B. 3

C. 4

D. 5

二、综合应用题

1. 图 $G=(V, E)$ 以邻接表存储,如图 5-19 所示,试画出图 G 的深度优先生成树和广度优先生成树(假设从结点 1 开始遍历)。

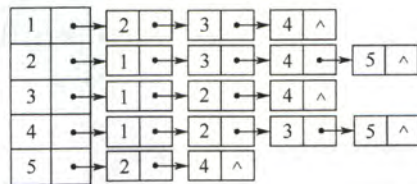


图 5-19 题 1 图

2. 试设计一个算法,判断一个无向图 G 是否为一棵树。若是一棵树,则算法返回 true,否则返回 false。

3. 写出图的深度优先搜索 DFS 算法的非递归算法(图采用邻接表形式)。

4. 分别采用基于深度优先遍历和广度优先遍历算法判别以邻接表方式存储的有向图中是否存在由顶点 v_i 到顶点 v_j 的路径($i \neq j$)。注意,算法中涉及的图的基本操作必须在此存储结构上实现。

5. 假设图用邻接表表示,设计一个算法,输出从顶点 V_i 到顶点 V_j 的所有简单路径。

5.3.5 答案与解析

一、单项选择题

1. A

广度优先搜索是以起始结点为中心,一层一层向外层扩展遍历图的顶点,因此无法考虑到边权值,只适合求边权值相等的图的单源最短路径。广度优先搜索相当于树的层序遍历,III错误。广度优先搜索需要用到队列,深度优先搜索需要用到栈,IV正确。

2. C

DFS (或 BFS) 可以用来计算图的连通分量数, 因为一次遍历必然能将一个连通图中的所有顶点都访问到, 而对于已被访问的顶点将不再调用 DFS, 故计算图的连通分量数正好是 DFSTraverse() 中 DFS 被调用的次数。

3. C、A、C、A

深度优先遍历时, 每个顶点表结点和每个边表结点均查找一次, 每个顶点递归调用一次, 需要借助一个递归工作栈; 而广度优先时也是每个顶点表结点和每个边表结点均查找一次, 每个顶点进入队列一次。故而都是选 C, A。

4. A、A

采用邻接矩阵表示时, 查找一个顶点所有出边的时间复杂度为 $O(n)$, 共有 n 个顶点, 故时间复杂度均为 $O(n^2)$ 。

5. D

画出草图后, 此类题可以根据边的邻接关系快速排除错误选项, 以 A 为例, 在遍历到 e 之后, 应该访问与 e 邻接但未被访问的结点, (e, c) 显然不在边集中。

6. D

仅 1 和 4 正确。以 2 为例, 遍历到 c 之后, 与 c 邻接且未被访问的结点为空集, 所以 a 的邻接点 b 或 e 入栈, 显然 2 不符合这种情况。

7. B、D

图的深度优先搜索类似与树的先根遍历, 是先访问结点, 再递归向外层结点遍历, 都采用回溯算法。图的广度优先搜索类似于树的层序遍历, 是一层一层向外层扩展遍历, 都需要采用队列来辅助算法的实现。

8. A、B

DFS 序列产生的路径为 $\langle 1, 2 \rangle$ 、 $\langle 2, 5 \rangle$ 、 $\langle 5, 4 \rangle$ 、 $\langle 3, 6 \rangle$; BFS 序列产生的路径为 $\langle 1, 2 \rangle$ 、 $\langle 1, 4 \rangle$ 、 $\langle 2, 5 \rangle$ 、 $\langle 3, 6 \rangle$ 。

9. D

画出 V 和 E 对应的图 G, 然后再根据搜索算法求解。

这里应该注意到: 为什么本题序列是不唯一的, 而上题却是唯一的呢?

因为上题给出了具体的存储结构, 此时就必须按照算法的过程来执行, 每个顶点的邻接点的顺序已固定, 但本题中每个顶点的邻接点的顺序是非固定的。

10. D

只要掌握 DFS 和 BFS 的遍历过程, 便能轻易解决。逐个代入, 手工模拟, 选项 D 是深度优先遍历, 而不是广度优先遍历。

11. D

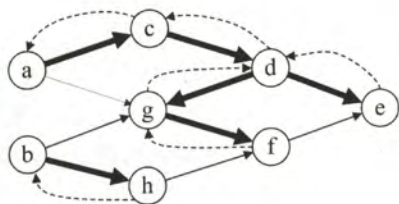
对于本题, 只需按深度优先遍历的策略进行遍历即可。对于选项 A: 先访问 V_1 , 然后访问与 V_1 邻接且未被访问的任一顶点 (满足的有 V_2 、 V_3 和 V_5), 此时访问 V_5 , 然后从 V_5 出发, 访问与 V_5 邻接且未被访问的任一顶点 (满足的只有 V_4), 然后从 V_4 出发, 访问与 V_4 邻接且未被访问的任一顶点 (满足的只有 V_3), 然后从 V_3 出发, 访问与 V_3 邻接且未被访问的任一顶点 (满足的只有 V_2), 结束遍历。选项 B 和 C 的分析方法与选项 A 相同, 不再赘述。对于选项 D, 首先访问 V_1 , 然后从 V_1 出发, 访问与 V_1 邻接且未被访问的任一顶点 (满足的有 V_2 、 V_3 和 V_5), 然后从 V_2 出发, 访问与 V_2 邻接且未被访问的任一顶点 (满足的只有 V_5), 按规则本应该访问 V_5 , 但选项 D 却访问 V_3 , D 错误。

12. C

利用深度优先遍历可以判断图 G 中是否存在回路。对于无向图来说,若深度优先遍历过程中遇到了回边则必定存在环;对于有向图来说,这条回边可能是指向深度优先森林中另一棵生成树上顶点的弧;但是,如果从有向图的某个顶点 v 出发进行深度优先遍历,若在 $\text{DFS}(v)$ 结束之前出现一条从顶点 u 到顶点 v 的回边,且 u 在生成树上是 v 的子孙,则有向图必定存在包含顶点 v 和顶点 u 的环。

13. A

对一个有向图做深度优先遍历,并未专门判断有向图是否有环(有向回路)存在,无论图中是否有环,都得到一个顶点序列。如果无环,在退出递归过程中输出的应是逆拓扑有序序列。对有向无环图利用深度优先搜索进行拓扑排序的例子如下:



第13题图

退出 DFS 栈的顺序为 $efgdcahb$, 此图的一个拓扑序列为 $bhacdgfe$ 。该方法的每一步均是先输出当前无后继的结点,即对每个结点 v ,先递归地求出 v 的每个后继的拓扑序列。对于一个 AOV 网,按此方法输出的序列是一个逆拓扑序列。

14. B

连通分量是无向图的极大连通子图,其中极大的含义是将依附于连通分量中顶点的所有边都加上,所以,连通分量中可能存在回路,这样就不是生成树了。

15. A

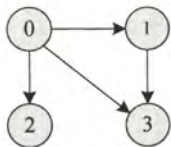
对于无向图的广度优先搜索生成树,起点到其他顶点的路径是图中对应的最短路径,也即是所有生成树中树高最小的。此外,深度优先总是尽可能“深”地搜索图,因此其路径也是尽可能的长,故深度优先生成树的树高总是大于或等于广度优先生成树的树高。

16. C

广度优先遍历需要借助队列实现。当采用邻接表存储方式时,在对图进行广度优先遍历时每个顶点均需入队一次(顶点表遍历),故时间复杂度为 $O(n)$,在搜索所有顶点的邻接点的过程中,每条边至少访问一次(出边表遍历),故时间复杂度为 $O(e)$,算法总的时间复杂度为 $O(n+e)$ 。

17. D

画出该有向图图形如下:

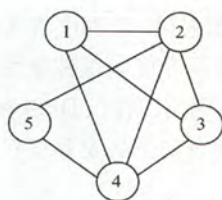


采用图的深度优先遍历,共5种可能: $\langle v_0, v_1, v_3, v_2 \rangle$, $\langle v_0, v_2, v_3, v_1 \rangle$, $\langle v_0, v_2, v_1, v_3 \rangle$, $\langle v_0, v_3, v_2, v_1 \rangle$, $\langle v_0, v_3, v_1, v_2 \rangle$,选D。

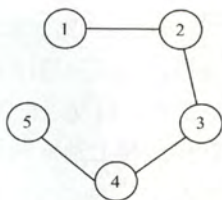
二、综合应用题

1. 解答:

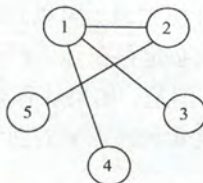
根据 G 的邻接表不难画出图(a)。



(a) 图G



(b) 深度优先生成树



(c) 广度优先生成树

1) 采用深度优先遍历

深度优先搜索总是尽可能“深”的搜索图,根据存储结构可知深度优先搜索的路径次序为(1, 2)、(2, 3)、(3, 4)、(4, 5),深度优先生成树如图(b)所示。需要注意的是,当存储结构固定时,生成树的树形也就固定了,比如不能先搜索(1, 3)。

2) 采用广度优先遍历

广度优先搜索总是尽可能“广”的搜索图,一层一层向外扩展,根据存储结构可知广度优先搜索的路径次序为(1, 2)、(1, 3)、(1, 4)、(2, 5),广度优先生成树如图(c)所示。

2. 解答:

一个无向图G是一棵树的条件是:G必须是无回路的连通图或者是有 $n-1$ 条边的连通图。这里采用后者作为判断条件。对连通的判定,可用能否遍历全部顶点来实现。可以采用深度优先搜索算法在遍历图的过程中统计可能访问到的顶点个数和边的条数,如果一次遍历就能访问到 n 个顶点和 $n-1$ 条边,则可断定此图是一棵树。

```
bool isTree(Graph& G){
    for(i=1;i<=G.vexnum;i++){
        visited[i]=FALSE;           //访问标记 visited[] 初始化
    }
    int Vnum=0,Enum=0;              //记录顶点数和边数
    DFS(G,1,Vnum,Enum,visited);
    if(Vnum==G.vexnum&&Enum==2*(G.vexnum-1))
        return true;                //符合树的条件
    else
        return false;               //不符合树的条件
}

void DFS(Graph& G,int v,int& Vnum,int& Enum,int visited[]){
    //深度优先遍历图G,统计访问过的顶点数和边数,通过Vnum和Enum返回
    visited[v]=TRUE;Vnum++;          //作访问标记,顶点计数
    int w=FirstNeighbor(G,v);        //取v的第一个邻接顶点
    while(w!=-1){                    //当邻接顶点存在
        Enum++;                       //边存在,边计数
        if(!visited[w])               //当该邻接顶点未访问过
            DFS(G,w,Vnum,Enum,visited);
        w=NextNeighbor(G,v,w);
    }
}
```

3. 解答:

在深度优先搜索的非递归算法中使用了一个栈S,记忆下一步可能访问的顶点,同时使用了一个访问标记数组visited[i],在visited[i]中记忆第i个顶点是否在栈内或曾经在栈内。若是,以后它不能再进栈。图采用邻接表形式,算法的实现如下:

```
void DFS_Non_RC(AGraph& G,int v){
    //从顶点v开始进行深度优先搜索,一次遍历一个连通分量的所有顶点
```



```

int w; //顶点序号
InitStack(S); //初始化栈 S
for(i=0;i<G.vexnum;i++)
    visited[i]=FALSE; //初始化 visited
Push(S,v);visited[v]=TRUE; //v 入栈并置 visited[v]
while(!IsEmpty(S)){
    k=Pop(S); //栈中退出一个顶点
    visit(k); //先访问, 再将其子结点入栈
    for(w=FirstNeighbor(G,k);w>=0;w=NextNeighbor(G,k,w))
        //k 所有邻接点
        if(!visited[w]){ //未进过栈的顶点进栈
            Push(S,w);
            visited[w]=true; //作标记, 以免再次入栈
        } //if
    } //while
} //DFS_Non_RC

```

注意: 由于使用了栈, 使得遍历的方式是从右端到左端的顺序进行的, 不同于常规的从左往右方式, 不过仍然是深度优先遍历, 读者可以用实例模拟验证。

4. 解答:

对于两个不同的遍历算法, 都是采用从顶点 v_i 出发, 依次遍历图中每个顶点, 直到搜索到顶点 v_j , 如果能够搜索到 v_j , 则说明存在由顶点 v_i 到顶点 v_j 的路径。

深度优先遍历算法的实现如下:

```

int visited[MAXSIZE]={0}; //访问标记数组
int Exist_Path_DFS(ALGraph G,int i,int j){
    int p; //顶点序号
    //深度优先判断有向图 G 中顶点  $v_i$  到顶点  $v_j$  是否有路径, 是则返回 1, 否则返回 0
    if(i==j)
        return 1; //i 就是 j
    else{
        visited[i]=1; //置访问标记
        for(p=FirstNeighbor(G,i);p>=0;p=NextNeighbor(G,i,p)){
            if(!visited[p]&&Exist_Path_DFS(G,p,j)) //递归检测邻接点
                return 1; //i 下游的顶点到 j 有路径
        } //for
    } //else
    return 0;
}

```

广度优先遍历算法的实现如下:

```

int visited[MAXSIZE]={0}; //访问标记数组
int Exist_Path_BFS(ALGraph G,int i,int j){
    //广度优先判断有向图 G 中顶点  $v_i$  到顶点  $v_j$  是否有路径, 是则返回 1, 否则返回 0
    InitQueue(Q); EnQueue(Q,i); //顶点 i 入队
    while(!IsEmpty(Q)){ //非空循环
        DeQueue(Q,u); //队头顶点出队
        visited[u]=1; //置访问标记
        for(p=FirstNeighbor(G,i);p>=0;p=NextNeighbor(G,i,p)){
            //检查所有邻接点
        }
    }
}

```



```

        k=p.adjvex;
        if(k==j)                //若 k==j, 则查找成功
            return 1;
        if(!visited[k])        //否则, 顶点 k 入队
            EnQueue(Q,k);
    } //for
} //while
return 0;
}

```

本题也可以这样解答, 调用以 i 为参数的 $\text{DFS}(G, i)$ 或 $\text{BFS}(G, i)$, 执行结束后判断 $\text{visited}[j]$ 是否为 TRUE, 如果是则说明 v_j 已被遍历, 图中必存在由 v_i 到 v_j 的路径。但此种解法每次都耗费最坏时间复杂度对应的的时间, 需要遍历与 V_i 连通的所有顶点。

5. 解答:

本题采用基于递归的深度优先遍历算法, 从结点 u 出发, 递归深度优先遍历图中结点, 当访问到结点 v 时, 则输出该搜索路径上的结点。为此, 设置一个 path 数组用于存放路径上的结点 (初始为空), d 表示路径长度 (初始为 -1)。查找从顶点 u 到 v 的简单路径过程, 说明如下 (假设查找函数名为 $\text{FindPath}()$):

1) $\text{FindPath}(G, u, v, \text{path}, d)$: $d++$; $\text{path}[d]=u$; 若找到 u 的未访问过的相邻结点 u_1 , 则继续下去, 否则置 $\text{visited}[u]=0$ 并返回。

2) $\text{FindPath}(G, u_1, v, \text{path}, d)$: $d++$; $\text{path}[d]=u_1$; 若找到 u_1 的未访问过的相邻结点 u_2 , 则继续下去, 否则置 $\text{visited}[u_1]=0$ 。

3) 依此类推, 继续上述递归过程, 直到 $U_i=V$, 则输出 path 。

算法的实现如下:

```

void FindPath(AGraph *G,int u,int v,int path[],int d){
    int w,i;
    ArcNode *p;
    d++;                                //路径长度增 1
    path[d]=u;                          //将当前顶点添加到路径中
    visited[u]=1;                       //置已访问标记
    if(u==v)                            //找到一条路径则输出
        print(path[]);                 //输出路径上的结点
    p=G->adjlist[u].firstarc;           //p 指向 v 的第一个相邻点
    while(p!=NULL){
        w=p->adjvex;                   //若顶点 w 未访问, 递归访问它
        if(visited[w]==0)
            FindPath(G,w,V,path,d);
        p=p->nextarc;                 //p 指向 v 的下一个相邻点
    }
    visited[u]=0;                      //恢复环境, 使该顶点可重新使用
}

```

5.4 图的应用

本节是历年考查的重点。图的应用主要包括: 最小生成 (代价) 树、最短路径、拓扑排序和

关键路径。一般而言，这部分内容直接以算法设计题形式考查的可能性很小，而更多的是结合图的实例来考查算法的具体执行过程，读者必须学会对给定的图手工模拟各个算法的执行过程。此外，还需掌握对于给定的模型建立相应的图去解决问题的方法。

5.4.1 最小生成树 (Minimum-Spanning-Tree, MST)

一个连通图的生成树是图的极小连通子图，它包含图中的所有顶点，并且只含尽可能少的边。这意味着对于生成树来说，若砍去它的一条边，就会使生成树变成非连通图；若给它增加一条边，就会形成图中的一条回路。

对于一个带权连通无向图 $G=(V, E)$ ，生成树不同，每棵树的权（即树中所有边上的权值之和）也可能不同。设 \mathcal{R} 为 G 的所有生成树的集合，若 T 为 \mathcal{R} 中边的权值之和最小的那棵生成树，则 T 称为 G 的最小生成树。

不难看出，最小生成树具有如下性质：

1) 最小生成树不是唯一的，即最小生成树的树形不唯一， \mathcal{R} 中可能有多个最小生成树。当图 G 中的各边权值互不相等时， G 的最小生成树是唯一的；若无向连通图 G 的边比顶点数少 1，即 G 本身就是一棵树时， G 的最小生成树就是它本身。

2) 最小生成树的边的权值之和总是唯一的，虽然最小生成树不唯一，但其对应的边的权值之和总是唯一的，而且是最小的。

3) 最小生成树的边数为顶点数减 1。

构造最小生成树有多种算法，但大多数算法都利用了最小生成树的下列性质：

假设 $G=(V, E)$ 是一个带权连通无向图， U 是顶点集 V 的一个非空子集。若 (u, v) 是一条具有最小权值的边，其中 $u \in U, v \in V-U$ ，则必存在一棵包含边 (u, v) 的最小生成树。

基于该性质的最小生成树算法主要有：Prim 算法和 Kruskal 算法，它们都基于贪心算法的策略。对这两种算法的掌握不应拘泥于其代码实现，而应掌握算法的本质含义和基本思想，并能够手工模拟算法的实现步骤。

下面介绍一个通用的最小生成树算法：

```

GENERIC_MST(G) {
    T=NULL;
    while T 未形成一棵生成树;
        do 找到一条最小代价边  $(u, v)$  并且加入  $T$  后不会产生回路;
            $T=T \cup (u, v)$ ;
}

```

通用的算法采用每次加入一条边来逐渐形成一棵生成树。下面介绍两种实现上述通用算法的途径：

1. 普里姆 (Prim) 算法

Prim 算法的执行非常类似于寻找图的最短路径的 Dijkstra 算法（见下一节）。假设 $N=\{V, E\}$ 是连通网， E_T 是 N 上最小生成树中边的集合。算法从 $V_T=\{u_0\}$ ($u_0 \in V$)， $E_T=\{\}$ 开始，重复执行下述操作：在所有 $u \in V_T, v \in V-V_T$ 的边 $(u, v) \in E$ 中找一条代价最小的边 (u_0, v_0) 并入集合 E_T ，同时将 v_0 并入 V_T ，直至 $V_T=V$ 为止。此时 E_T 中必有 $n-1$ 条边，则 $T=\{V_T, E_T\}$ 为 N 的最小生成树。图 5-20 所示为 Prim 算法构造最小生成树的过程。

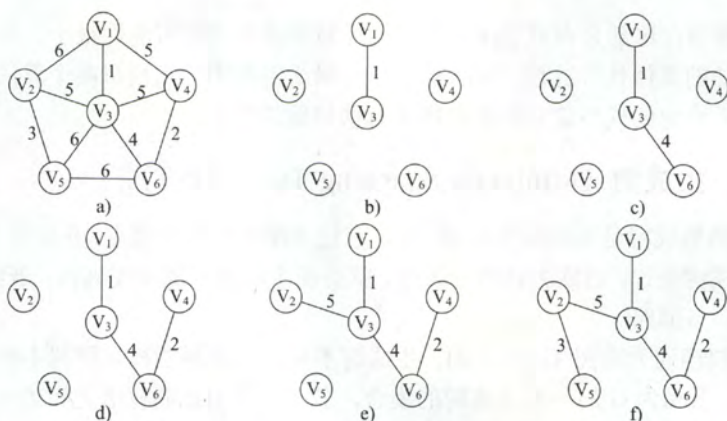


图 5-20 Prim 算法构造最小生成树的过程

Prim 算法的步骤如下:

初始化: 向空树 $T=(V_T, E_T)$ 中添加图 $G=(V, E)$ 的任一顶点 u_0 , 使 $V_T=\{u_0\}$, $E_T=\emptyset$ 。

循环 (重复下列操作至 $V_T=V$): 从图 G 中选择满足 $\{(u, v) | u \in V_T, v \in V-V_T\}$ 且具有最小权值的边 (u, v) , 并置 $V_T=V_T \cup \{v\}$, $E_T=E_T \cup \{(u, v)\}$ 。

Prim 算法的简单实现如下:

```
void Prim(G, T) {
    T = ∅; // 初始化空树
    U = {w}; // 添加任一顶点 w
    while ((V-U) != ∅) { // 若树中不含全部顶点
        设 (u, v) 是使  $u \in U$  与  $v \in (V-U)$ , 且权值最小的边;
        T = T ∪ {(u, v)}; // 边归入树
        U = U ∪ {v}; // 顶点归入树
    }
}
```

Prim 算法的时间复杂度为 $O(|V|^2)$, 不依赖于 $|E|$, 因此它适用于求解边稠密的图的最小生成树。虽然采用其他方法可以改进 Prim 算法的时间复杂度, 但增加了实现的复杂性。

2. 克鲁斯卡尔 (Kruskal) 算法

与 Prim 算法从顶点开始扩展最小生成树不同, Kruskal 算法是一种按权值的递增次序选择合适的边来构造最小生成树的方法。假设 $N=(V, E)$ 是连通网, 对应的最小生成树 $T=(V_T, E_T)$, Kruskal 算法的步骤如下:

初始化: $V_T=V$, $E_T=\emptyset$ 。即每个顶点构成一棵独立的树, T 此时是一个仅含 $|V|$ 个顶点的森林;

循环 (重复下列操作至 T 是一棵树): 按 G 的边的权值递增顺序依次从 $E-E_T$ 中选择一条边, 如果这条边加入 T 后不构成回路, 则将其加入 E_T , 否则舍弃, 直到 E_T 中含有 $n-1$ 条边。

Kruskal 算法的简单实现如下:

```
void Kruskal(V, T) {
    T = V; // 初始化树 T, 仅含顶点
    numS = n; // 连通分量数
    while (numS > 1) { // 如果连通分量数大于 1
        从 E 中取出权值最小的边 (v, u);
        if (v 和 u 属于 T 中不同的连通分量) {
            T = T ∪ {(v, u)}; // 将此边加入生成树中
            numS--; // 连通分量数减 1
        }
    }
}
```

根据图的相关性质,若一条边连接了两棵不同树中的顶点时,则对这两棵树来说,它必定是连通的,将这条边加入森林中,完成两棵树的合并,直到整个森林合并成一棵树。

对于图 5-21 (a) 所示的连通网,按照 Kruskal 算法构造最小生成树的过程如图 5-21 所示。在构造过程中,按照网中边的权值由小到大的顺序,不断选取当前未被选取的边集中权值最小的边。依据生成树的概念, n 个结点的生成树,有 $n-1$ 条边,故反复上述过程,直到选取了 $n-1$ 条边为止,就构成了一棵最小生成树。

通常在 Kruskal 算法中,采用堆(第 7 章)来存放边的集合,则每次选择最小权值的边只需 $O(\log|E|)$ 的时间。又生成树 T 中所有边可以看作一个等价类,每次添加新的边的过程类似于求解等价类的过程,由此可以采用并查集的数据结构来描述 T ,从而构造 T 的时间复杂度为 $O(|E|\log|E|)$,因此, Kruskal 算法适合于边稀疏而顶点较多的图。

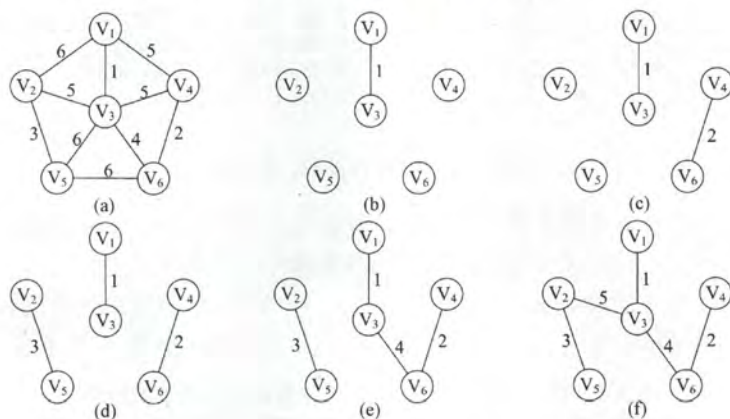


图 5-21 Kruskal 算法构造最小生成树的过程

5.4.2 最短路径

在 5.3 节所述的广度优先搜索查找最短路径只是对无权图而言,若图是带权图,则把从一个顶点 v_0 到图中其余任一个顶点 v_i 的一条路径(可能不止一条)上所经过边上的权值之和定义为该路径的带权路径长度,把带权路径长度最短的那条路径也称作最短路径。

求解最短路径的算法通常都依赖于一种性质,也就是两点之间的最短路径也包含了路径上其他顶点间的最短路径。带权有向图 G 的最短路径问题,一般可分为两类:一是单源最短路径,即求图中某一顶点到其他各顶点的最短路径,可通过经典的 Dijkstra 算法求解;二是求每一对顶点间的最短路径,可通过 Floyd-Warshall 算法来求解。

1. Dijkstra 算法求单源最短路径问题

求带权有向图中某个源点到其余各顶点的最短路径,最常用的是 Dijkstra 算法。该算法设置一个集合 S 记录已求得的最短路径的顶点,可用一个数组 $s[]$ 来实现,初始化为 0,当 $s[v_i]=1$ 时表示将顶点 v_i 放入 S 中,初始时把源点 v_0 放入 S 中。此外,在构造过程中还设置了两个辅助数组:

$\text{dist}[]$: 记录了从源点 v_0 到其他各顶点当前的最短路径长度, $\text{dist}[i]$ 初值为 $\text{arcs}[v_0][i]$ 。

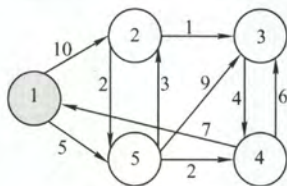
$\text{path}[]$: $\text{path}[i]$ 表示从源点到顶点 i 之间的最短路径的前驱结点,在算法结束时,可根据其值追溯得到源点 v_0 到顶点 v_i 的最短路径。

假设从顶点 0 出发,即 $v_0=0$,集合 S 最初只包含顶点 0,邻接矩阵 arcs 表示带权有向图, $\text{arcs}[i][j]$

表示有向边 $\langle i, j \rangle$ 的权值, 若不存在有向边 $\langle i, j \rangle$, 则 $\text{arcs}[i][j]$ 为 ∞ 。Dijkstra 算法的步骤如下(不考虑对 $\text{path}[]$ 的操作):

- 1) 初始化: 集合 S 初始为 $\{0\}$, $\text{dist}[]$ 的初始值 $\text{dist}[i] = \text{arcs}[0][i]$, $i=1, 2, \dots, n-1$ 。
- 2) 从顶点集合 $V-S$ 中选出 v_j , 满足 $\text{dist}[j] = \min\{\text{dist}[i] \mid v_i \in V-S\}$, v_j 就是当前求得的一条从 v_0 出发的最短路径的终点, 令 $S=S \cup \{j\}$ 。
- 3) 修改从 v_0 出发到集合 $V-S$ 上任一顶点 v_k 可达的最短路径长度: 如果 $\text{dist}[j] + \text{arcs}[j][k] < \text{dist}[k]$, 则令 $\text{dist}[k] = \text{dist}[j] + \text{arcs}[j][k]$ 。
- 4) 重复 2) ~ 3) 操作共 $n-1$ 次, 直到所有的顶点都包含在 S 中。

例如, 表 5-1 所示为应用 Dijkstra 算法对图 5-22 中的图从顶点 1 出发, 求其到其余顶点的最短路径。思考: Dijkstra 算法与 Prim 算法的相似之处?



每趟得到的最短路径为:

第1趟: $1 \rightarrow 5$, 路径距离为5

第2趟: $1 \rightarrow 5 \rightarrow 4$, 路径距离为7

第3趟: $1 \rightarrow 5 \rightarrow 2$, 路径距离为8

第4趟: $1 \rightarrow 5 \rightarrow 2 \rightarrow 3$, 路径距离为9

图 5-22 应用 Dijkstra 算法图

显然, Dijkstra 算法也是基于贪心策略的。算法的主要部分为一个双重循环, 外层循环内有两个并列的单层循环, 任取一个循环内的操作为基本操作, 则基本操作执行的总次数为双重循环执行的次数。若使用邻接矩阵表示, 它的时间复杂度为 $O(|V|^2)$ 。若使用带权的邻接表表示, 虽然修改 $\text{dist}[]$ 的时间可以减少, 但由于在 $\text{dist}[]$ 中选择最小分量的时间不变, 其时间复杂度仍为 $O(|V|^2)$ 。

表 5-1 从 v_1 到各终点的 dist 值和最短路径的求解过程

顶点	第 1 趟	第 2 趟	第 3 趟	第 4 趟
2	10 $v_1 \rightarrow v_2$	8 $v_1 \rightarrow v_5 \rightarrow v_2$	8 $v_1 \rightarrow v_5 \rightarrow v_2$	
3	∞	14 $v_1 \rightarrow v_5 \rightarrow v_3$	13 $v_1 \rightarrow v_5 \rightarrow v_4 \rightarrow v_3$	9 $v_1 \rightarrow v_5 \rightarrow v_2 \rightarrow v_3$
4	∞	7 $v_1 \rightarrow v_5 \rightarrow v_4$		
5	5 $v_1 \rightarrow v_5$			
集合 S	{1, 5}	{1, 5, 4}	{1, 5, 4, 2}	{1, 5, 4, 2, 3}

人们可能只希望找到从源点到某一个特定顶点的最短路径, 但是, 这个问题和求解源点到其他所有顶点的最短路径一样复杂, 其时间复杂度也为 $O(|V|^2)$ 。而如果要找出所有结点对之间的最短距离, 则需要对每个结点运行一次 Dijkstra 算法, 即时间复杂度为 $O(|V|^3)$ 。

值得注意的是, 如果边上带有负权值, Dijkstra 算法并不适用。若允许边上带有负权值, 有可能出现当与 S (已求得最短路径的顶点集, 归入 S 内的结点的最短路径不再变更) 内某点 (记为 a) 以负边相连的点 (记为 b) 确定其最短路径时, 它的最短路径长度加上这条负边的权值结果小于 a 原先确定的最短路径长度, 而此时 a 在 Dijkstra 算法下是无法更新的。例如, 对于图 5-23 所示的带权有向图, 利用 Dijkstra 算法不一定能得到正确的结果。

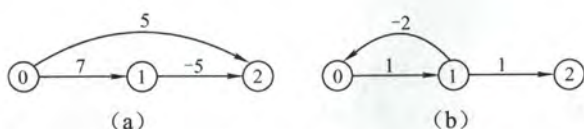


图 5-23 边上带有负权值的有向带权图

2. Floyd 算法求各顶点之间最短路径问题

求所有顶点之间的最短路径问题描述如下：已知一个各边权值均大于 0 的带权有向图，对每一对顶点 $v_i \neq v_j$ ，要求求出 v_i 与 v_j 之间的最短路径和最短路径长度。

Floyd 算法的基本思想是：递推产生一个 n 阶方阵序列 $A^{(-1)}, A^{(0)}, \dots, A^{(k)}, \dots, A^{(n-1)}$ ，其中 $A^{(k)}[i][j]$ 表示从顶点 v_i 到顶点 v_j 的路径长度， k 表示绕行第 k 个顶点的运算步骤。初始时，对于任意两个顶点 v_i 和 v_j ，若它们之间存在边，则以此边上的权值作为它们之间的最短路径长度；若它们之间不存在有向边，则以 ∞ 作为它们之间的最短路径长度。以后逐步尝试在原路径中加入顶点 $k(k=0, 1, \dots, n-1)$ 作为中间顶点。如果增加中间顶点后，得到的路径比原来的路径长度减少了，则以此新路径代替原路径。算法的描述如下：

定义一个 n 阶方阵序列： $A^{(-1)}, A^{(0)}, \dots, A^{(n-1)}$ ，其中：

$$A^{(-1)}[i][j] = \text{arcs}[i][j]$$

$$A^{(k)}[i][j] = \min\{A^{(k-1)}[i][j], A^{(k-1)}[i][k] + A^{(k-1)}[k][j]\}, k=0, 1, \dots, n-1$$

其中， $A^{(0)}[i][j]$ 是从顶点 v_i 到 v_j 、中间顶点是 v_0 的最短路径的长度， $A^{(k)}[i][j]$ 是从顶点 v_i 到 v_j 、中间顶点的序号不大于 k 的最短路径的长度。Floyd 算法是一个迭代的过程，每迭代一次，在从 v_i 到 v_j 的最短路径上就多考虑了一个顶点；经过 n 次迭代后所得到的 $A^{(n-1)}[i][j]$ 就是 v_i 到 v_j 的最短路径长度，即方阵 $A^{(n-1)}$ 中就保存了任意一对顶点之间的最短路径长度。

如图 5-24 所示为带权有向图 G 及其邻接矩阵，下面通过实例来说明 Floyd 算法的过程见表 5-2。

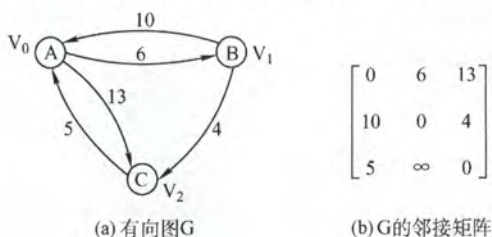
图 5-24 带权有向图 G 及其邻接矩阵

表 5-2 Floyd 算法的执行过程

dist	$\text{dist}^{(-1)}$			$\text{dist}^{(0)}$			$\text{dist}^{(1)}$			$\text{dist}^{(2)}$		
	V_0	V_1	V_2	V_0	V_1	V_2	V_0	V_1	V_2	V_0	V_1	V_2
V_0	0	6	13	0	6	13	0	6	10	0	6	10
V_1	10	0	4	10	0	4	10	0	4	9	0	4
V_2	5	∞	0	5	11	0	5	11	0	5	11	0

Floyd 算法的时间复杂度为 $O(|V|^3)$ 。不过由于其代码很紧凑，而且并不包含其他复杂的数据结构，因此隐含的常数系数是很小的，即使对于中等规模的输入来说，它仍然是相当有效的。

Floyd 算法允许图中有带负权值的边，但不允许有包含带负权值的边组成的回路。Floyd 算法同样也适用于带权无向图，因为带权无向图可以看做是有往返二重边的有向图，只要在顶点 v_i 和 v_j 之间存在无向边 (v_i, v_j) ，就可以看成是在这两个顶点之间存在权值相同的两条有向边 $\langle v_i, v_j \rangle$

和 $\langle v_j, v_i \rangle$ 。

也可以用单源最短路径算法来解决每对顶点之间最短路径问题。每一次运行时,轮流将一个顶点作为源点,并且若所有边权值均为非负时,可以采用上面提到的 Dijkstra 算法,其时间复杂度为 $O(|V|^2 \cdot |V|) = O(|V|^3)$ 。

5.4.3 拓扑排序

有向无环图: 一个有向图中不存在环,则称为有向无环图,简称 DAG 图。

AOV 网: 如果用 DAG 图表示一个工程,其顶点表示活动,用有向边 $\langle V_i, V_j \rangle$ 表示活动 V_i 必须先于活动 V_j 进行的这样一种关系,则将这种有向图称为顶点表示活动的网络,记为 AOV 网。在 AOV 网中,活动 V_i 是活动 V_j 的直接前驱,活动 V_j 是活动 V_i 的直接后继,这种前驱和后继关系具有传递性,且任何活动 V_i 不能以它自己作为自己的前驱或后继。

拓扑排序: 在图论中,由一个有向无环图的顶点组成的序列,当且仅当满足下列条件时,称为该图的一个拓扑排序。

- ① 每个顶点出现且只出现一次。
- ② 若顶点 A 在序列中排在顶点 B 的前面,则在图中不存在从顶点 B 到顶点 A 的路径。

或者定义为: 拓扑排序是对有向无环图的顶点的一种排序,它使得如果存在一条从顶点 A 到顶点 B 的路径,那么在排序中顶点 B 出现在顶点 A 的后面。每个 DAG 图都有一个或多个拓扑排序序列。

对一个 DAG 图进行拓扑排序的算法有很多,下面介绍比较常用的一种方法的步骤:

- ① 从 DAG 图中选择一个没有前驱的顶点并输出。
- ② 从图中删除该顶点和所有以它为起点的有向边。
- ③ 重复①和②直到当前的 DAG 图为空或当前图中不存在无前驱的顶点为止。而后一种情况说明有向图中必然存在环。

下面通过实例来说明拓扑排序过程,如图 5-25 所示。

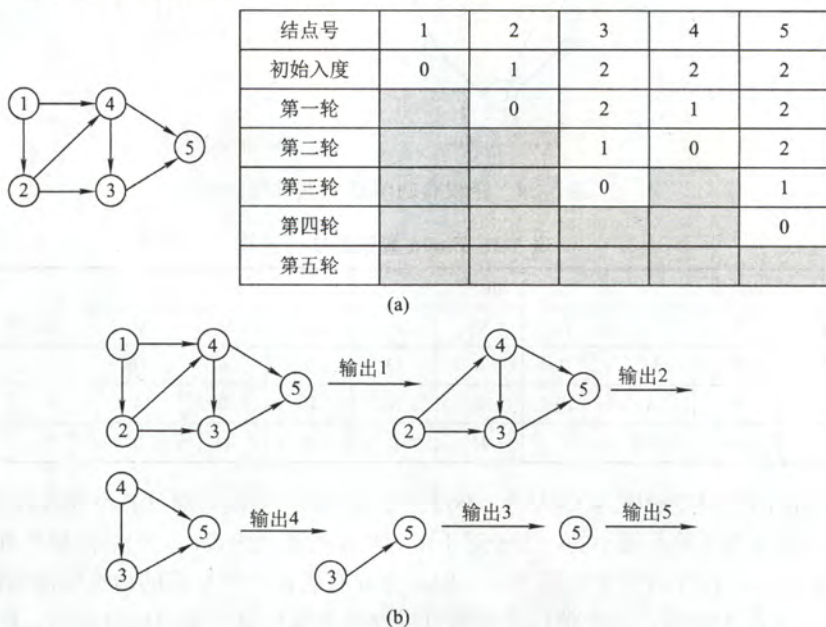


图 5-25 有向无环图的拓扑排序过程

于是,得到拓扑排序后的结果为 {1, 2, 4, 3, 5}。

拓扑排序算法的实现如下:

```
bool TopologicalSort(Graph G){
    //如果 G 存在拓扑序列, 返回 true; 否则返回 false, 这时 G 中存在环
    InitStack(S);           //初始化栈, 存储入度为 0 的顶点
    for(int i=0; i<G.vexnum; i++){
        if(indegree[i]==0)
            Push(S, i);      //将所有入度为 0 的顶点进栈
    }
    int count=0;           //计数, 记录当前已经输出的顶点数
    while(!IsEmpty(S)){    //栈不空, 则存在入度为 0 的顶点
        Pop(S, i);          //栈顶元素出栈
        print[count++]=i;    //输出顶点 i
        for(p=G.vertices[i].firstarc; p; p=p->nextarc){
            //将所有 i 指向的顶点的入度减 1, 并且将入度减为 0 的顶点压入栈 S
            v=p->adjvex;
            if(--indegree[v]==0)
                Push(S, v);  //入度为 0, 则入栈
        }
    }
    if(count<G.vexnum)
        return false;      //排序失败, 有向图中有回路
    else
        return true;       //拓扑排序成功
}
```

由于输出每个顶点的同时还要删除以它为起点的边, 故拓扑排序的时间复杂度为 $O(|V|+|E|)$ 。此外, 利用上一节的深度优先遍历也可以实现拓扑排序, 请读者仔细思考其原因及实现方法。后面的习题将会涉及该内容。

用拓扑排序算法处理 DAG 图时, 应该注意以下问题:

- ① 入度为零的顶点即没有前驱活动的, 或前驱活动都已经完成的顶点, 工程可以从这个顶点所代表的活动开始或继续。
- ② 如果一个顶点有多个直接后继, 则拓扑排序的结果通常不唯一; 但如果各个顶点已经排在一个线性有序的序列中, 每个顶点有唯一的前驱后继关系, 再作拓扑排序时, 则排序的结果是唯一的。
- ③ 由于 DAG 图中各顶点的地位平等, 每个顶点编号是人为的, 因此可以按照拓扑排序的结果重新安排顶点的序号, 生成的 DAG 图的新的邻接矩阵存储表示, 这种邻接矩阵可以是三角矩阵; 但是对于一般的图, 如果它的邻接矩阵是三角矩阵, 则存在拓扑序列; 反之则不一定成立。

5.4.4 关键路径

在带权有向图中, 以顶点表示事件, 有向边表示活动, 边上的权值表示完成该活动的开销 (如完成活动所需时间), 则称这种有向图为用边表示活动的网络, 简称为 **AOE 网**。

AOE 网具有以下两个性质: ① 只有在某顶点所代表的事件发生后, 从该顶点出发的各有向边所代表的活动才能开始; ② 只有在进入某一顶点的各有向边所代表的活动都已经结束时, 该顶点所代表的事件才能发生。

在 AOE 网中仅有一个入度为 0 的顶点, 称为开始顶点 (源点), 它表示整个工程的开始; 网中也仅存在一个出度为 0 的顶点, 称为结束顶点 (汇点), 它表示整个工程的结束。

在 AOE 网中,有些活动是可以并行进行的。从源点到汇点的有向路径可能有多条,并且这些路径长度可能不同。完成不同路径上的活动所需时间虽然不同,但是只有所有路径上的活动都完成了,整个工程才能算是结束了。因此,从源点到汇点的所有路径中,具有最大路径长度的路径称为**关键路径**。把关键路径上的活动称为**关键活动**。

完成整个工程的最短时间就是关键路径的长度,也就是关键路径上各活动花费开销的总和。这是因为关键活动影响了整个工程的时间,即如果关键活动不能按时完成的话,整个工程的完成时间就会延长。因此,只要找到了关键活动,就找到了关键路径,也就可以得出最短完成时间。

下面给出在寻找关键活动时所用到的几个参量定义:

1. 事件 v_k 的最早发生时间 $ve(k)$

它是指从开始顶点 V 到 V_k 的最长路径长度。事件的最早发生时间决定了所有从 V_k 开始的活动能够开工的最早时间。可用下面的递推公式来计算:

$$ve(\text{源点})=0$$

$$ve(k)=\text{Max}\{ve(j)+\text{Weight}(v_j, v_k)\}, \text{Weight}(v_j, v_k) \text{ 表示 } \langle v_j, v_k \rangle \text{ 上的权值}$$

注意:在计算 $ve(k)$ 时,是按从前往后的顺序来计算的。

2. 事件 v_k 的最迟发生时间 $vl(k)$

它是指在不推迟整个工程完成的前提下,即保证它所指向的事件 v_i 在 $ve(i)$ 时刻能够发生时,该事件最迟必须发生的时间。可用下面的递推公式来计算:

$$vl(\text{汇点})=ve(\text{汇点})$$

$$vl(j)=\text{Min}\{vl(k)-\text{Weight}(v_j, v_k)\}, \text{Weight}(v_j, v_k) \text{ 表示 } \langle v_j, v_k \rangle \text{ 的权值}$$

注意:在计算 $vl(j)$ 时,是按从后往前的顺序来计算的。

3. 活动 a_i 的最早开始时间 $e(i)$

它是指该活动的起点所表示的事件最早发生时间。如果边 $\langle v_k, v_j \rangle$ 表示活动 a_i , 则有 $e(i)=ve(k)$ 。

4. 活动 a_i 的最迟开始时间 $l(i)$

它是指该活动的终点所表示的事件最迟发生时间与该活动所需时间之差。如果边 $\langle v_k, v_j \rangle$ 表示活动 a_i , 则有 $l(i)=vl(j)-\text{Weight}(v_k, v_j)$ 。

5. 一个活动 a_i 的最迟开始时间 $l(i)$ 和其最早开始时间 $e(i)$ 的差额 $d(i)=l(i)-e(i)$

它是指该活动完成的时间余量,是在不增加完成整个工程所需的总时间的情况下,活动 a_i 可以拖延的时间。如果一个活动的时间余量为零时,说明该活动必须要如期完成,否则就会拖延完成整个工程的进度,所以称 $l(i)-e(i)=0$ 、即 $l(i)=e(i)$ 的活动 a_i 是关键活动。

求关键路径的算法步骤如下:

- (1) 求 AOE 网中所有事件的最早发生时间 $ve()$ 。
- (2) 求 AOE 网中所有事件的最迟发生时间 $vl()$ 。
- (3) 求 AOE 网中所有活动的最早开始时间 $e()$ 。
- (4) 求 AOE 网中所有活动的最迟开始时间 $l()$ 。
- (5) 求 AOE 网中所有活动的差额 $d()$, 找出所有 $d()=0$ 的活动构成关键路径。

图 5-26 所示为求解关键路径的过程。由此图我们可以得到这个 AOE 网的关键路径为 (v_1, v_3, v_4, v_6) 。

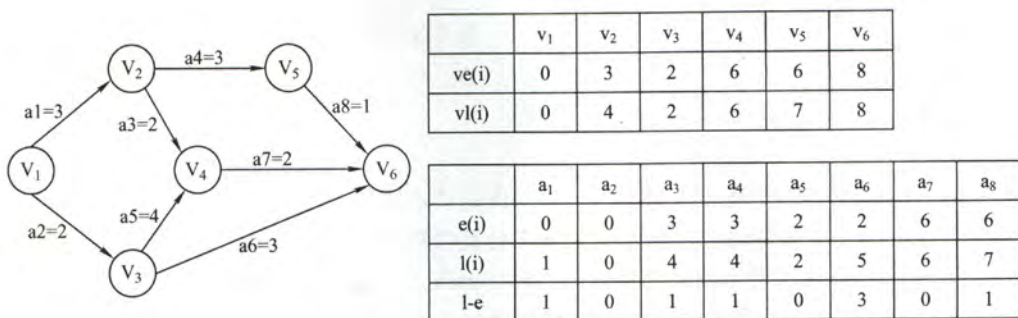


图 5-26 求解关键路径的过程

对于关键路径，我们需要注意以下几点：

1) 关键路径上的所有活动都是关键活动，它是决定整个工程的关键因素，因此可通过加快关键活动来缩短整个工程的工期。但也不能任意缩短关键活动，因为一旦缩短到一定的程度，该关键活动可能变成非关键活动了。

2) 网中的关键路径并不唯一。且对于有几条关键路径的网，只提高一条关键路径上的关键活动速度并不能缩短整个工程的工期，只有加快那些包括在所有关键路径上的关键活动才能达到缩短工期的目的。

5.4.5 本节试题精选

一、单项选择题

1. 【2010 年计算机联考真题】

对图 5-27 进行拓扑排序，可以得到不同的拓扑序列的个数是 ()。

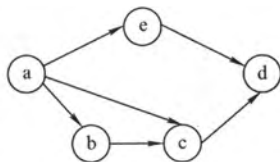


图 5-27 题 1 图

- A. 4
B. 3
C. 2
D. 1
2. 任何一个无向连通图的最小生成树 ()。
- A. 有一棵或多棵
B. 只有一棵
C. 一定有多棵
D. 可能不存在
3. 用 Prim 算法和 Kruskal 算法构造图的最小生成树，所得到的最小生成树 ()。
- A. 相同
B. 不相同
C. 可能相同，可能不同
D. 无法比较
4. 以下叙述中正确的是 ()。
- A. 只要无向连通图中没有权值相同的边，则其最小生成树唯一
B. 只要无向图中有权值相同的边，则其最小生成树一定不唯一
C. 从 n 个顶点的连通图中选取 $n-1$ 条权值最小的边，即可构成最小生成树
D. 设连通图 G 含有 n 个顶点，则含有 n 个顶点 $n-1$ 条边的子图一定是 G 的生成树

5. 【2012 年计算机联考真题】

下列关于最小生成树的叙述中，正确的是 ()。

- I. 最小生成树的代价唯一
II. 所有权值最小的边一定会出现在所有的最小生成树中
III. 使用普里姆 (Prim) 算法从不同顶点开始得到的最小生成树一定相同
IV. 使用普里姆算法和克鲁斯卡尔 (Kruskal) 算法得到的最小生成树总不相同

A. 仅 I

B. 仅 II

C. 仅 I、III

D. 仅 II、IV

6. 以下叙述正确的是 ()。

A. 最短路径一定是简单路径

B. Dijkstra 算法不适合求有回路的带权图的最短路径

C. Dijkstra 算法不适合求任意两个顶点的最短路径

D. Floyd 算法求两个顶点的最短路径时, $path_{k-1}$ 一定是 $path_k$ 的子集

7. 已知带权连通无向图 $G=(V, E)$, 其中 $V=\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$, $E=\{(v_1, v_2)10, (v_1, v_3)2, (v_3, v_4)2, (v_3, v_6)11, (v_2, v_5)1, (v_4, v_5)4, (v_4, v_6)6, (v_5, v_7)7, (v_6, v_7)3\}$ (注: 顶点偶对括号外的数据表示边上的权值), 从源点 v_1 到顶点 v_7 的最短路径上经过的顶点序列是 ()。

A. v_1, v_2, v_5, v_7 B. v_1, v_3, v_4, v_6, v_7 C. v_1, v_3, v_4, v_5, v_7 D. $v_1, v_2, v_5, v_4, v_6, v_7$

8. 【2012 年计算机联考真题】

对如图 5-28 有向带权图, 若采用迪杰斯特拉 (Dijkstra) 算法求从源点 a 到其他各顶点的最短路径, 则得到的第一条最短路径的目标顶点是 b , 第二条最短路径的目标顶点是 c , 后续得到的其余各最短路径的目标顶点依次是 ()。

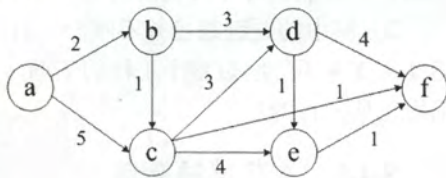
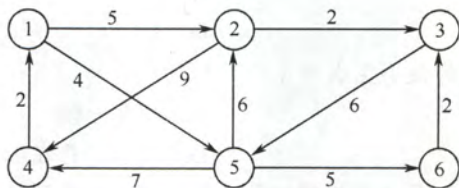
A. d, e, f B. e, d, f C. f, d, e D. f, e, d 

图 5-28 题 8 图

9. 【2016 年计算机联考真题】

使用迪杰斯特拉 (Dijkstra) 算法求下图中从顶点 1 到其他各顶点的最短路径, 依次得到的各最短路径的目标顶点是 ()。



A. 5, 2, 3, 4, 6

B. 5, 2, 3, 6, 4

C. 5, 2, 4, 3, 6

D. 5, 2, 6, 3, 4

10. 下面哪一方法可以判断出一个有向图是否有环 (回路) ()。

I. 深度优先遍历

II. 拓扑排序

III. 求最短路径

IV. 求关键路径

A. I、II、IV

B. I、III、IV

C. I、II、III

D. 全部可以

11. 【2016 年计算机联考真题】

若将 n 个顶点 e 条弧的有向图采用邻接表存储, 则拓扑排序算法的时间复杂度是 ()。

A. $O(n)$ B. $O(n+e)$ C. $O(n^2)$ D. $O(n \cdot e)$ 12. 在有向图 G 的拓扑序列中, 若顶点 v_i 在顶点 v_j 之前, 则下列情形不可能出现的是 ()。A. G 中有弧 $\langle v_i, v_j \rangle$ B. G 中有一条从 v_i 到 v_j 的路径C. G 中没有弧 $\langle v_i, v_j \rangle$ D. G 中有一条从 v_j 到 v_i 的路径

13. 若一个有向图的顶点不能排在一个拓扑序列中, 则可判定该有向图 ()。
- A. 是一个有根的有向图 B. 是一个强连通图
C. 含有多个入度为 0 的顶点 D. 含有顶点数目大于 1 的强连通分量
14. 以下关于拓扑排序的说法中错误的是 ()。
- I. 如果某有向图存在环路, 则该有向图一定不存在拓扑排序
II. 在拓扑排序算法中, 为暂存入度为零的顶点可以使用栈, 也可以使用队列
III. 若有向图的拓扑有序序列唯一, 则图中每个顶点的入度和出度最多为 1
- A. I、III B. II、III C. II D. III
15. 若一个有向图的顶点不能排成一个拓扑序列, 则判定该有向图 ()。
- A. 含有多个出度为 0 的顶点
B. 是个强连通图
C. 含有多个入度为 0 的顶点
D. 含有顶点数大于 1 的强连通分量
16. 【2012 年计算机联考真题】

对如图 5-29 所示的有向图进行拓扑排序, 得到的拓扑序列可能是 ()。

- A. 3,1,2,4,5,6 B. 3,1,2,4,6,5
C. 3,1,4,2,5,6 D. 3,1,4,2,6,5

17. 如图 5-30 所示有向图的所有拓扑序列共有 () 个。

- A. 4 B. 6 C. 5 D. 7

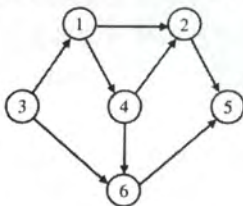


图 5-29 题 16 图

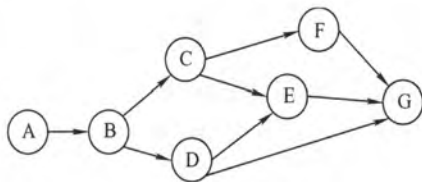


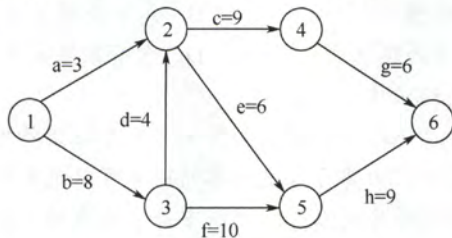
图 5-30 题 17 图

18. 若一个有向图具有有序的拓扑排序序列, 那么它的邻接矩阵必定为 ()。
- A. 对称 B. 稀疏 C. 三角 D. 一般
19. 下列关于图的说法中, 正确的是 ()。
- I. 有向图中顶点 V 的度等于其邻接矩阵中第 V 行中 1 的个数
II. 无向图的邻接矩阵一定是对称矩阵, 有向图的邻接矩阵一定是非对称矩阵
III. 在图 G 的最小生成树 G_1 中, 可能会有某条边的权值超过未选边的权值
IV. 如果有向无环图的拓扑序列唯一, 则可以唯一确定该图
- A. I、II 和 III B. III 和 IV C. III D. IV
20. 若某带权图为 $G=(V, E)$, 其中 $V=\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$, $E=\{<v_1, v_2>5, <v_1, v_3>6, <v_2, v_5>3, <v_3, v_5>6, <v_3, v_4>3, <v_4, v_5>3, <v_4, v_7>1, <v_4, v_8>4, <v_5, v_6>4, <v_5, v_7>2, <v_6, v_{10}>4, <v_7, v_9>5, <v_8, v_9>2, <v_9, v_{10}>2\}$ (注: 边括号外的数据表示边上的权值), 则 G 的关键路径的长度为 ()。
- A. 19 B. 20 C. 21 D. 22

21. 【2013 年计算机联考真题】

下列 AOE 网表示一项包含 8 个活动的工程。通过同时加快若干活动的进度可以缩短整个工

程的工期。下列选项中, 加快其进度就可以缩短工程工期的是 ()。



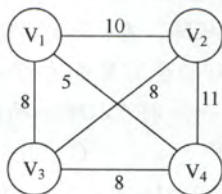
- A. c 和 e B. d 和 c C. f 和 d D. f 和 h
22. 下面关于求关键路径的说法中, 不正确的是 ()。
- A. 求关键路径是以拓扑排序为基础的
- B. 一个事件的最早发生时间同以该事件为始的弧的活动最早开始时间相同
- C. 一个事件的最迟发生时间为以该事件为尾的弧的活动最迟开始时间与该活动的持续时间的差
- D. 关键活动一定位于关键路径上
23. 下列关于关键路径的说法中, 正确的是 ()。
- I. 当改变网上某一关键路径上任一关键活动后, 必将产生不同的关键路径
- II. 在 AOE 图中, 关键路径上活动的时间延长多少, 整个工程的时间也就随之延长多少
- III. 缩短关键路径上任意一个关键活动的持续时间可缩短关键路径长度
- IV. 缩短所有关键路径上共有的任意一个关键活动的持续时间可缩短关键路径长度
- V. 缩短多条关键路径上共有的任意一个关键活动的持续时间可缩短关键路径长度
- A. II 和 V B. I、II 和 IV C. II 和 IV D. I 和 IV

24. 【2012 年计算机联考真题】

若用邻接矩阵存储有向图, 矩阵中主对角线以下的元素均为零, 则关于该图拓扑序列的结论是 ()。

- A. 存在, 且唯一 B. 存在, 且不唯一
- C. 存在, 可能不唯一 D. 无法确定是否存在
25. 【2015 年计算机联考真题】

求下面带权图的最小 (代价) 生成树时, 可能是克鲁斯卡 (Kruskal) 算法第 2 次选中但不是普里姆 (Prim) 算法 (从 V_4 开始) 第 2 次选中的边是 ()。



- A. (V_1, V_3) B. (V_1, V_4) C. (V_2, V_3) D. (V_3, V_4)

二、综合应用题

1. 下面有一种称为“破圈法”的求解最小生成树的方法:

所谓“破圈法”就是“任取一圈, 去掉圈上权最大的边”, 反复执行这一步骤, 直到没有圈为止。

试判断这种方法是否正确。如果正确, 请说明理由; 如果不正确, 举出反例。(注: 圈就是

回路。)

2. 已知有向图如图 5-31 所示:

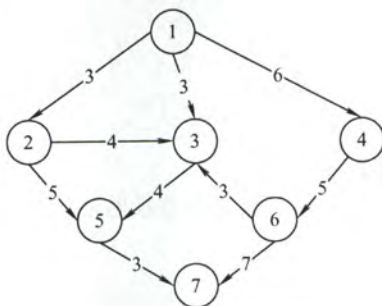


图 5-31 题 2 图

1) 写出该图的邻接矩阵表示并据此给出从顶点 1 出发的深度优先遍历序列。

2) 求该有向图的强连通分量数目。

3) 给出该图的两个拓扑序列。

4) 若将该图视为无向图, 分别用 Prim 算法和 Kruskal 算法求最小生成树。

3. 对图 5-32 所示的无向图, 按照 Dijkstra 算法, 写出从顶点 1 到其他各个顶点的最短路径和最短路径长度 (顺序不能颠倒)。

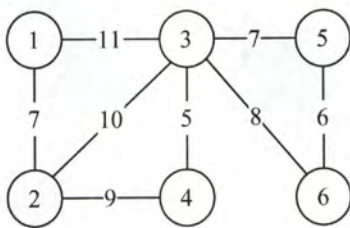


图 5-32 题 3 图

4. 【2009 年计算机联考真题】

带权图 (权值非负, 表示边连接的两顶点间的距离) 的最短路径问题是找出从初始顶点到目标顶点之间的一条最短路径。假设从初始顶点到目标顶点之间存在路径, 现有一种解决该问题的方法:

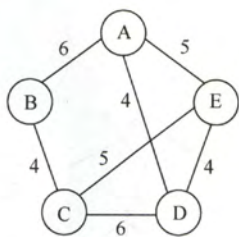
- ① 设最短路径初始时仅包含初始顶点, 令当前顶点 u 为初始顶点。
- ② 选择离 u 最近且尚未在最短路径中的一个顶点 v , 加入到最短路径中, 修改当前顶点 $u=v$ 。
- ③ 重复步骤②, 直到 u 是目标顶点时为止。

请问上述方法能否求得最短路径? 若该方法可行, 请证明; 否则, 请举例说明。

5. 【2017 年计算机联考真题】

使用 Prim (普里姆) 算法求带权连通图的最小 (代价) 生成树 (MST)。请回答下列问题。

- 1) 对下列图 G , 从顶点 A 开始求 G 的 MST, 依次给出按算法选出的边。
- 2) 图 G 的 MST 是唯一的吗?
- 3) 对任意的带权连通图, 满足什么条件时, 其 MST 是唯一的?



6. 【2011 年计算机联考真题】

已知有 6 个顶点（顶点编号为 0~5）的有向带权图 G，其邻接矩阵 A 为上三角矩阵，按行为主序（行优先）保存在如下的一维数组中。

4	6	∞	∞	∞	5	∞	∞	∞	4	3	∞	∞	3	3
---	---	----------	----------	----------	---	----------	----------	----------	---	---	----------	----------	---	---

要求：

- 1) 写出图 G 的邻接矩阵 A。
- 2) 画出有向带权图 G。
- 3) 求图 G 的关键路径，并计算该关键路径的长度。
7. 图 5-33 所示为一个用 AOE 网表示的工程。

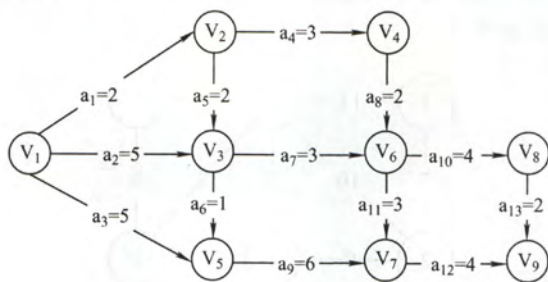


图 5-33 题 6 图

试回答：

- 1) 画出此图的邻接表表示。
- 2) 完成此工程，至少需要多少时间？
- 3) 指出关键路径。
- 4) 哪些活动加速可以缩短完成工程所需的时间？
8. 下表给出了某工程各工序之间的优先关系和各工序所需时间（其中“—”表示无先驱工序），请完成以下各题：
 - 1) 画出相应的 AOE 网。
 - 2) 列出各事件的最早发生时间和最迟发生时间。
 - 3) 求出关键路径并指明完成该工程所需最短时间。

工序代号	A	B	C	D	E	F	G	H
所需时间	3	2	2	3	4	3	2	1
先驱工序	—	—	A	A	B	A	C、E	D

9. 试说明利用 DFS 如何实现有向无环图拓扑排序。
10. 一连通无向图，边非负权值，问用 Dijkstra 最短路径算法能否给出一棵生成树，该树是

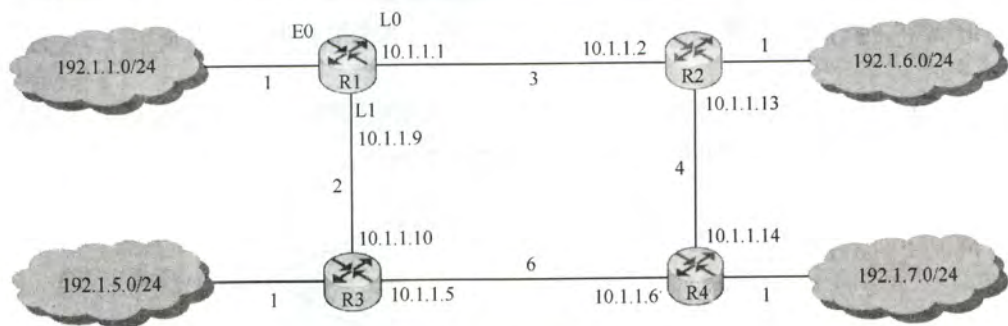
否一定是最小生成树? 说明理由。

11. 【2014 年计算机联考真题】

某网络中的路由器运行 OSPF 路由协议, 下表是路由器 R1 维护的主要链路状态信息(LSI), R1 构造的网络拓扑图是根据题下表及 R1 的接口名构造出来的网络拓扑。

题 11-1 表 R1 所维护的 LSI

		R1 的 LSI	R2 的 LSI	R3 的 LSI	R4 的 LSI	备 注
Router ID		10.1.1.1	10.1.1.2	10.1.1.5	10.1.1.6	标识路由器的 IP 地址
Link1	ID	10.1.1.2	10.1.1.1	10.1.1.6	10.1.1.5	所连路由器的 Router ID
	IP	10.1.1.1	10.1.1.2	10.1.1.5	10.1.1.6	Link1 的本地 IP 地址
	Metric	3	3	6	6	Link1 的费用
Link2	ID	10.1.1.5	10.1.1.6	10.1.1.1	10.1.1.2	所连路由器的 Router ID
	IP	10.1.1.9	10.1.1.13	10.1.1.10	10.1.1.14	Link2 的本地 IP 地址
	Metric	2	4	2	4	Link2 的费用
Net1	Prefix	192.1.1.0/24	192.1.6.0/24	192.1.5.0/24	192.1.7.0/24	直连网络 Net1 的网络前缀
	Metric	1	1	1	1	到达直连网络 Net1 的费用



R1 构造的网络拓扑

请回答下列问题。

- 1) 本题中的网络可抽象为数据结构中的哪种逻辑结构?
- 2) 针对表中的内容, 设计合理的链式存储结构, 以保存表中的链路状态信息(LSI)。要求给出链式存储结构的数据类型定义, 并画出对应表的链式存储结构示意图(示意图中可仅以 ID 标识结点)。
- 3) 按照迪杰斯特拉 (Dijkstra) 算法的策略, 依次给出 R1 到达子网 192.1.x.x 的最短路径及费用。

5.4.6 答案与解析

一、单项选择题

1. B

可以得到 3 种不同的拓扑序列 abcd, abcd, aebcd, 故选 B。

2. A

当无向连通图存在权值相同的多条边时, 最小生成树可能是不唯一的, 另外, 由于这是一个无向连通图, 故而最小生成树必定存在。从而选 A。

3. C

由于无向连通图的最小生成树不一定唯一，所以用不同算法生成的最小生成树可能不同，但当无向连通图的最小生成树唯一时，不同算法生成的最小生成树必定是相同的。

4. A

选项 A 显然正确；选项 B，如果无向图本身就是一棵树，则最小生成树就是它本身，这个时候就是唯一的；选项 C，选取的 $n-1$ 条边可能构成回路；选项 D，含有 n 个顶点 $n-1$ 条边的子图可能构成回路，也可能不连通。

5. A

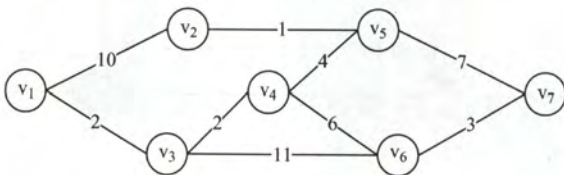
最小生成树的树形可能不唯一（这是因为可能存在权值相同的边），但是代价一定是唯一的，I 正确。如果权值最小的边有多条并且构成环状，则总有权值最小的边将不出现在某棵最小生成树中，II 错误。设 N 个结点构成环， $N-1$ 条边权值相等，则从不同的顶点开始普里姆算法会得到 $N-1$ 中不同的最小生成树，III 错误。当最小生成树唯一时（各边的权值不同），普里姆算法和克鲁斯卡尔算法得到的最小生成树相同，IV 错误。

6. A

选项 A 正确，见严蔚敏《数据结构》教材。Dijkstra 算法适合求解有回路的带权图的最短路径，也可以求任意两个顶点的最短路径，不适合求带负权值的最短路径问题。在用 Floyd 算法求两个顶点的最短路径时，当最短路径发生更改时， $path_{k-1}$ 就不是 $path_k$ 的子集。

7. B

题干内容所述的图 G 如下图所示。则 A、B、C、D 对应的路径长度分别为 18、13、15、24。应用 Dijkstra 算法不难求出最短路径为 $v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow v_6 \rightarrow v_7$ 。



8. C

从 a 到各顶点的最短路径的求解过程：

顶点	第 1 趟	第 2 趟	第 3 趟	第 4 趟	对 5 趟
b	(a,b) 2				
c	(a,c) 5	(a,b,c) 3			
d	∞	(a,b,d) 5	(a,b,d) 5	(a,b,d) 5	
e	∞	∞	(a,b,c,e) 7	(a,b,c,e) 7	(a,b,d,e) 6
f	∞	∞	(a,b,c,f) 4		
集合 S	{a,b}	{a,b,c}	{a,b,c,f}	{a,b,c,f,d}	{a,b,c,f,d,e}

后续目标顶点依次为 f,d,e,

本题也可用排除法：对于 A，若下一个顶点为 d，路径 a,b,d 的长度 5，而 a,b,c,f 的长度仅为 4，显然错误。同理可排除选项 B。将 f 加入集合 S 后，采用上述的方法也可排除选项 D。

9. B

根据 Dijkstra 算法，从顶点 1 到其余各顶点的最短路径如下表所示：

顶 点	第 1 趟	第 2 趟	第 3 趟	第 4 趟	第 5 趟
2	5 $v_1 \rightarrow v_2$	5 $v_1 \rightarrow v_2$			
3	∞	∞	7 $v_1 \rightarrow v_2 \rightarrow v_3$		
4	∞	11 $v_1 \rightarrow v_5 \rightarrow v_4$	11 $v_1 \rightarrow v_5 \rightarrow v_4$	11 $v_1 \rightarrow v_5 \rightarrow v_4$	11 $v_1 \rightarrow v_5 \rightarrow v_4$
5	4 $v_1 \rightarrow v_5$				
6	∞	9 $v_1 \rightarrow v_5 \rightarrow v_6$	9 $v_1 \rightarrow v_5 \rightarrow v_6$	9 $v_1 \rightarrow v_5 \rightarrow v_6$	
集合 S	{1, 5}	{1, 5, 2}	{1, 5, 2, 3}	{1, 5, 2, 3, 6}	{1, 5, 2, 3, 6, 4}

10. A

使用深度优先遍历, 如果从有向图上某个顶点 u 出发, 在 $\text{DFS}(u)$ 结束之前出现一条从顶点 v 到 u 的边, 由于 v 在生成树上是 u 的子孙, 则图中必定存在包含 u 和 v 的环, 因此深度优先遍历的方法可以检测出一个有向图是否有环。拓扑排序时, 当某顶点不为任何边的头时才能加入序列, 存在环路时环路中的顶点一直是某条边的头, 不能加入拓扑序列。也就是说, 还存在顶点但无法找到下一个可以加入拓扑序列的顶点, 则说明此图存在回路。最短路径是允许有环的。至于关键路径能不能判断一个图有环存在一些争议, 关键路径本身虽然不允许有环, 但求关键路径的算法本身无法判断是否有环, 判断是否有环的是关键路径的第一步——拓扑排序。所以这个问题的答案主要是看你从哪个角度出发看问题, 考生需要理解问题本身, 而真正统考的时候是不会涉及一些模棱两可的问题的。

11. B

采用邻接表作为 AOV 网的存储结构进行拓扑排序, 需要对 n 个顶点做进栈、出栈、输出各一次, 在处理 e 条边时, 需要检测这 n 个顶点的边链表的 e 个边结点, 总共需要的时间代价为 $O(n+e)$ 。若采用邻接矩阵作为 AOV 网的存储结构进行拓扑排序, 在处理 e 条边时需要对每一个顶点检测相应矩阵中的某一行, 寻找与它相关联的边, 以便对这些边的入度减 1, 需要的时间代价为 $O(n^2)$ 。

12. D

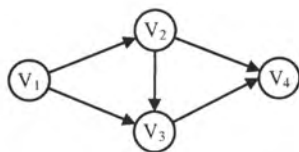
如果图 G 中存在一条 $v_j \rightarrow \dots \rightarrow v_i$ 的路径, 中间的“ \dots ”表示中间路径所经过的结点, 从而 v_j 的入度比 v_i 的入度更早地为 0, 从而拓扑序列中必然是先输出 v_j , 再输出 v_i , 这显然与题意矛盾。

13. D

若不存在拓扑排序, 则表示图中必定存在回路, 该回路构成一个强连通分量 (不然理解: 顶点数目大于 1 的强连通分量中必然存在回路)。

14. D

I 中, 对于一个存在环路的有向图, 使用拓扑排序算法运行后, 肯定会出现有环的子图, 在此环中就无法再找到入度为 0 的结点了, 拓扑排序也就进行不下去了。II 中, 注意, 如果两个结点之间不存在祖先或子孙关系的话, 则它们在拓扑序列中的关系是任意的 (即前后关系任意), 从而使用栈和队列都可以, 因为进栈或队列的



第 14 题图

都是入度为0的结点,此时入度为0的所有结点是没有关系的。III是难点,若拓扑有序序列唯一,则很自然地让人联系到一个线性的有向图(错误),第14题图的拓扑序列也是唯一的,但度却不满足条件。

15. D

一个有向图中的顶点不能排成一个拓扑序列,则表明其中存在一个顶点数目大于1的回路(环),该回路构成一个强连通分量。从而答案选D。

16. D

按照拓扑排序的算法,每次都选择入度为0的结点从图中删去,此图中一开始只有结点3的入度为0;删掉结点3后,只有结点1的入度为0;删掉结点1后,只有结点4的入度为0;删掉结点4后,结点2和结点6的入度都为0,此时选择删去不同的结点,会得出不同的拓扑序列,分别处理完毕后可知可能的拓扑序列为3,1,4,2,6,5和3,1,4,6,2,5,选D。

17. C

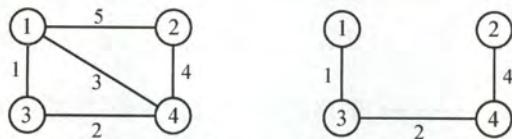
本图的拓扑排序序列有:ABCFDEG、ABCD FEG、ABCDEF G、ABDC FEG、ABDC EFG。读者应该能够把这一类经典习题的拓扑序列全部写出来。

18. C

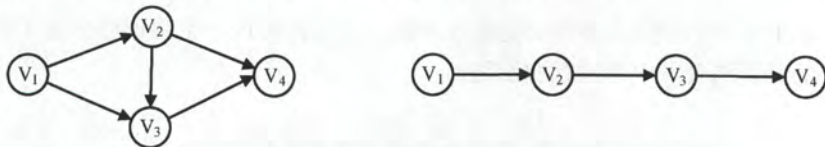
此题一直以来争议较大,是因为有些书中漏掉了“有序”二字,可以证明,对于有向图中顶点适当地编号,使其邻接矩阵为三角矩阵且主对角元全为零的充分必要条件是:该有向图可以进行拓扑排序。如果这个题目把“有序”二字去掉,显然应该选D。但此题题干中已经指出是“有序的拓扑序列”,从而应该选C。需要注意的是,如果一个有向图的邻接矩阵为三角矩阵(对角线上元素为0),则图中必不存在环,则其拓扑序列必然存在。

19. C

有向图的邻接矩阵的第V行中1的个数是顶点V的出度,而有向图中顶点的度为入度和出度之和,I错。无向图的邻接矩阵一定是对称矩阵,但当有向图中的任意两顶点之间若有边相连则是两条方向相反的有向边(无向图也可看成是有两条方向相反有向边的特殊有向图)时,有向图的邻接矩阵也是一个对称矩阵,II错。最小生成树中的n-1条边并不能保证是图中权值最小的n-1条边,因为权值最小的n-1条边并不一定能使图连通,如下图所示,左图的最小生成树如图所示,权值为3的边并不在其最小生成树中。

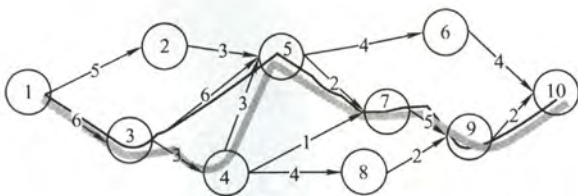


有向无环图的拓扑序列唯一,并不能唯一确定该图,如下图所示的两个有向无环图,拓扑序列都为 V_1, V_2, V_3, V_4 , IV错。【注:很多辅导书对该命题判断错误】



20. C

画出题目所表示的图如下,则可以得到关键路径的长度为21。图中所表示的两条路径都是关键路径。



21. C

找出 AOE 网的全部关键路径为 bdcg、bdeh 和 bfh。根据定义，只有关键路径上的活动时间同时减少时，才能缩短工期。选项 A、B 和 D 并不包含在所有的关键路径中，只有 C 包含，因此只有加快 f 和 d 的进度才能缩短工期（建议读者在图中检验）。

22. C

一个事件的最迟发生时间等于 $\text{Min}\{\text{以该事件为尾的弧的活动最迟开始时间, 最迟结束时间与该活动的持续时间的差}\}$ 。

23. C

如果改变的是所有关键路径上的公共活动，则不一定会产生不同的关键路径（延长必然不会导致，只有缩短才有可能导致）。根据关键路径的定义，可知 II 正确。关键路径是源点到终点的最长路径，只有所有关键路径的长度都缩短，整个图的关键路径才能有效缩短，但也不能任意缩短，一旦缩短到一定程度，该关键活动可能变成非关键活动了。

24. C

对角线以下元素均为零，表明只有顶点 i 到顶点 j ($i < j$) 可能有边，而顶点 j 到顶点 i 一定没有边，即有向图是一个无环图，因此一定存在拓扑序列。对于拓扑序列是否唯一，试举一例：设

有向图的邻接矩阵 $\begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ ，则存在两个拓扑序列，因此该图存在可能不唯一的拓扑序列。若

题目中说对角线以上均为 1，以下均为 0，则拓扑序列唯一。

25. C

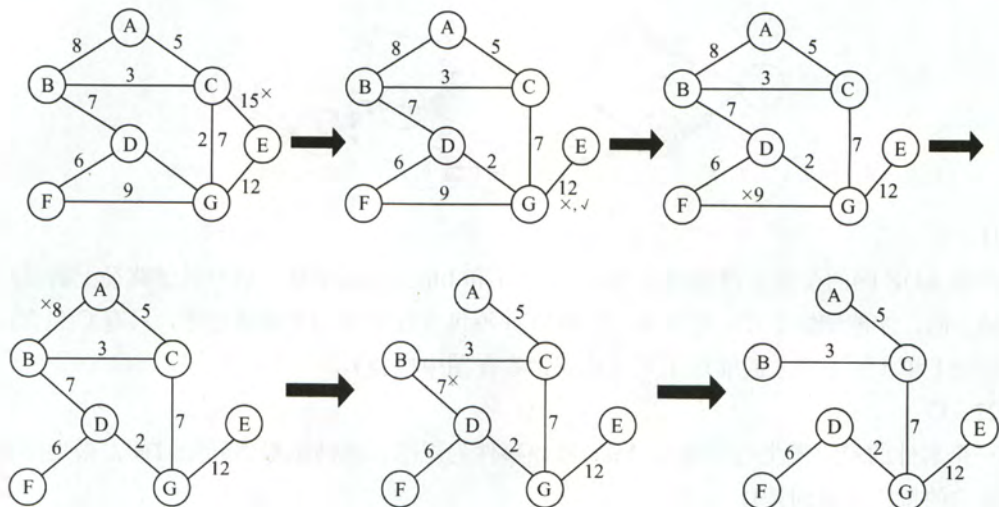
从 V_4 开始，Kruskal 算法选中的第一条边一定是权值最小的 (V_1, V_4) ，B 错误。由于 V_1 和 V_4 已经可达，第二条边含有 V_1 和 V_4 的权值为 8 的一定符合 Prim 算法，排除 A、D。

二、综合应用题

1. 解答：

这种方法是正确的。

由于经过“破圈法”之后，最终没有回路，故一定可以构造出一棵生成树。下面证明这棵生成树是最小生成树。记“破圈法”生成的树为 T ，假设 T 不是最小生成树，则必然存在最小生成树 T_0 ，使得它与 T 的公共边尽可能地多，则将 T_0 与 T 取并集，得到一个图，此图中必然将存在回路，由于“破圈法”的定义就是从回路中去除权最大的边，此时生成的 T 的权必然是最小的，这与原假设 T 不是最小生成树矛盾。从而 T 是最小生成树。下面通过实例来说明“破圈法”的过程：



2. 解答:

1) 该图的邻接矩阵如下所示:

$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \begin{pmatrix} 0 & 3 & 3 & 6 & \infty & \infty & \infty \\ \infty & 0 & 4 & \infty & 5 & \infty & \infty \\ \infty & \infty & 0 & \infty & 4 & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & 5 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & 3 & \infty & \infty & 0 & 7 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix} \end{matrix}$$

得到的深度优先遍历序列为: 1, 2, 3, 5, 7, 4, 6

2) 解题思路: 当某个顶点只有出弧没有入弧时, 其他顶点无法到达这个顶点, 不可能与其他顶点和边构成强连通分量 (这个单独的顶点构成一个强连通分量)。

[1] 顶点 1 无入弧构成第一个强连通分量。删除顶点 1 及所有以之为尾的弧。

[2] 顶点 2 无入弧构成一个强连通分量。删除顶点 2 及所有以之为尾的弧。

[3] ……

依次类推, 可以最后得到, 每一个顶点都是一个强连通分量, 故强连通分量数目为 7。

3) 该图的两个拓扑序列如下:

[1] 1, 2, 4, 6, 3, 5, 7

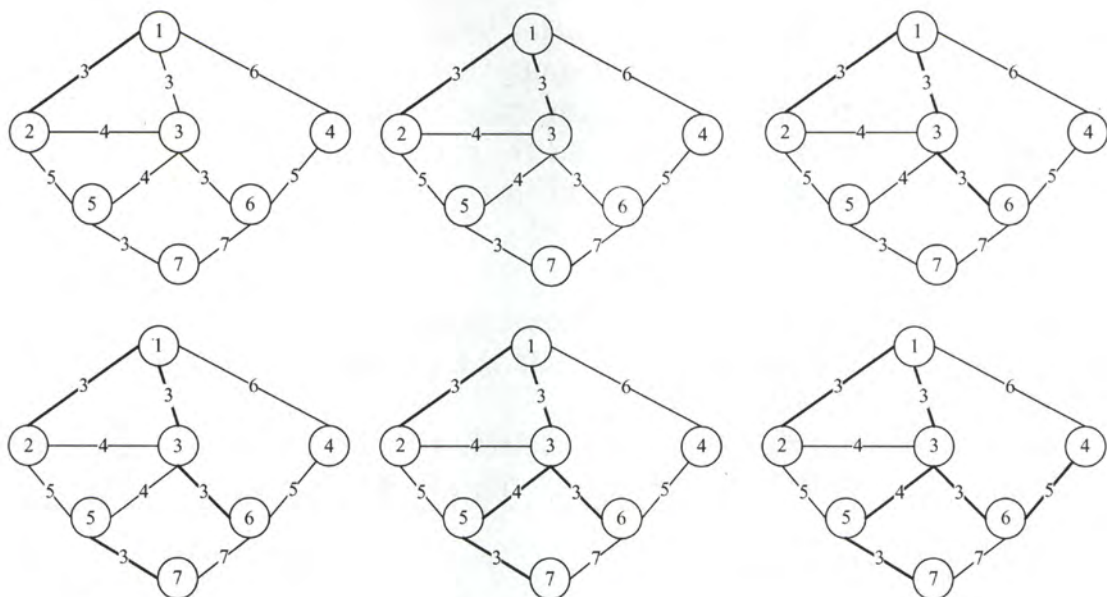
[2] 1, 4, 2, 6, 3, 5, 7

4) 若视该图为无向图:

用 Prim 算法生成最小生成树的过程如下:

1—2, 1—3, 3—6, 3—5, 5—7, 6—4 (图略)。

用 Kruskal 算法生成最小生成树的过程如下图所示。



3. 解答:

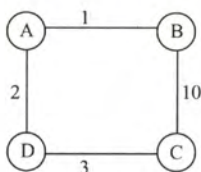
根据 Dijkstra 算法, 求从顶点 1 到其余各顶点的最短路径如下表所示:

顶点	从顶点 1 到各终点的 dist 值				
	第 1 趟	第 2 趟	第 3 趟	第 4 趟	
2	7 v_1, v_2				
3	11 v_1, v_3	11 v_1, v_3			
4	∞	16 v_1, v_2, v_4	16 v_1, v_2, v_4		
5	∞	∞	18 v_1, v_3, v_5	18 v_1, v_3, v_5	
6	∞	∞	19 v_1, v_3, v_6	19 v_1, v_3, v_6	19 v_1, v_3, v_6
集合 S	{1, 2}	{1, 2, 3}	{1, 2, 3, 4}	{1, 2, 3, 4, 5}	{1, 2, 3, 4, 5, 6}

4. 解答:

该方法不一定能(或不能)求得最短路径。

例如, 对于下图所示的带权图, 如果按照题中的原则, 从 A 到 C 的最短路径是 A->B->C, 事实上其最短路径是 A->D->C。



5. 解答:

1) Prim 算法属于贪心策略。算法从一个任意的顶点开始, 一直长大到覆盖图中所有顶点为止。算法每一步在连接树集合 S 中顶点和其他顶点的边中, 选择一条使得树的总权重增加最小的

边加入集合 S 。当算法终止时, S 就是最小生成树。

- ① S 中顶点为 A , 候选边为 (A,D) 、 (A,B) 、 (A,E) , 选择 (A,D) 加入 S 。
- ② S 中顶点为 A, D , 候选边为 (A,B) 、 (A,E) 、 (D,E) 、 (C,D) , 选择 (D,E) , 加入 S 。
- ③ S 中顶点为 A, D, E , 候选边为 (A,B) 、 (C,D) 、 (C,E) , 选择 (C,E) 加入 S 。
- ④ S 中顶点为 A, D, E, C , 候选边为 (A,B) 、 (B,C) , 选择 (B,C) 加入 S 。
- ⑤ S 就是最小生成树。

依次选出的边为:

$(A,D), (D,E), (C,E), (B,C)$

2) 图 G 的 MST 是唯一的。(2 分) 第一小题的最小生成树包括了图中权值最小的四条边, 其他边都比这四条边大, 所以此图的最小生成树唯一。

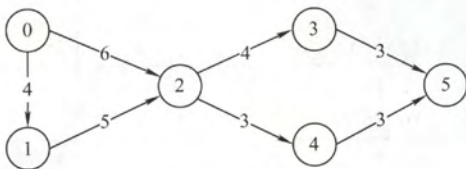
3) 当带权连通图的任意一个环中所包含的边的权值均不相同, 其 MST 是唯一的。此题不要求回答充分必要条件, 所以回答一个限制边权值的充分条件即可。

6. 解答:

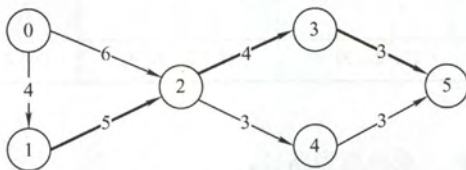
1) 图 G 的邻接矩阵 A 如下所示。

$$A = \begin{pmatrix} 0 & 4 & 6 & \infty & \infty & \infty \\ \infty & 0 & 5 & \infty & \infty & \infty \\ \infty & \infty & 0 & 4 & 3 & \infty \\ \infty & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & \infty & 0 & 3 \\ \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

2) 有向带权图 G 如下图所示。



3) 关键路径为 $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5$ (如下图所示粗线表示), 长度为 $4+5+4+3=16$ 。



注意: 读者务必熟练掌握根据邻接矩阵(邻接表)画出有向图(或无向图)的方法和计算其关键路径、最短路径及长度等的方法。

7. 解答:

1) 该图的邻接表表示如图 5-34 所示。

求关键路径的算法如下:

- ① 输入 e 条弧 $\langle j, k \rangle$, 建立 AOE 网的存储结构;
- ② 从源点 v_1 出发, 令 $ve(1)=0$, 求 $ve(j) \ 2 \leq j \leq n$;
- ③ 从汇点 v_n 出发, 令 $vl(n)=ve(n)$, 求 $vl(i) \ 1 \leq i \leq n-1$;
- ④ 根据各顶点的 ve 和 vl 值, 求每条弧 s (活动) 的最早开始时间 $e(s)$ 和最晚开始时间 $l(s)$,

其中 $e(s)=l(s)$ 为关键活动。

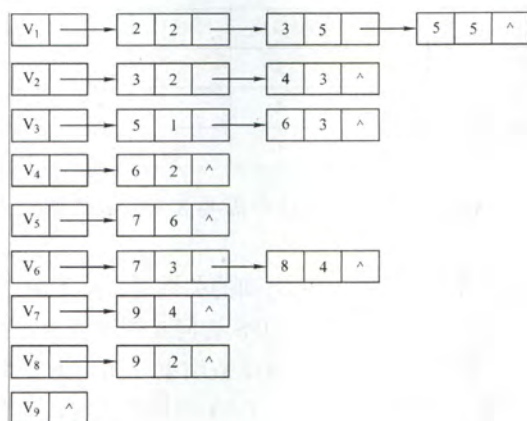


图 5-34 题 6 图

根据以上算法可以得到:

2) 至少需要时间 16。

3) 关键路径为: (V₁, V₃, V₅, V₇, V₉)。

4) 活动 a₂、a₆、a₉、a₁₂ 加速, 可以缩短工程所需的时间。

8. 解答:

1) 根据上表可以画出 AOE 网如图 5-35 所示。

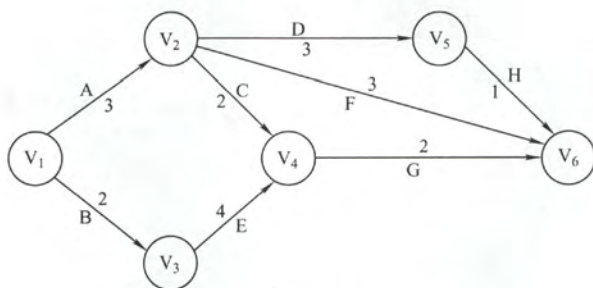


图 5-35 题 7 图

求解各事件和活动的最早发生时间和最迟发生时间公式分别如下:

① $ve(\text{源点})=0$, $ve(k)=\text{Max}\{ve(j)+\text{Weight}(v_j, v_k)\}$, $\text{Weight}(v_j, v_k)$ 表示从 v_j 指向 v_k 的弧的权值。

② $vl(\text{汇点})=ve(\text{汇点})$, $vl(j)=\text{Min}\{vl(k)-\text{Weight}(v_j, v_k)\}$, $\text{Weight}(v_j, v_k)$ 表示从 v_j 指向 v_k 的弧的权值。

③ 如果边 $\langle v_k, v_j \rangle$ 表示活动 a_i , 则有 $e(i)=ve(k)$ 。

④ 如果边 $\langle v_k, v_j \rangle$ 表示活动 a_i , $l(i)=vl(j)-\text{Weight}(v_k, v_j)$ 。

⑤ $d(i)=l(i)-e(i)$ 。

关键路径即由 $d(i)=0$ 的 i 构成。

2) 根据上述公式, 各事件的最早发生时间 ve 和最迟发生时间 vl 如下表:

	v_1	v_2	v_3	v_4	v_5	v_6
$ve(i)$	0	3	2	6	6	8
$vl(i)$	0	4	2	6	7	8

3) 根据上述公式, 各活动最早发生时间 e 、最迟发生时间 l 和时间余量 $d=l-e$ 如下表:

	A	B	C	D	E	F	G	H
$e(i)$	0	0	3	3	2	3	6	6
$l(i)$	1	0	4	4	2	5	6	7
$l(i)-e(i)$	1	0	1	1	0	2	0	1

所以关键路径为: B、E、G, 完成该工程最少需要 8 (单位依题意而定)。

9. 解答:

对于有向无环图 G 中任意结点 u 、 v , 它们之间的关系必然是下列三种之一:

1) 假设结点 u 是结点 v 的祖先, 则在调用 DFS 访问 u 的过程中, 必然会在这个过程结束之前递归地对 v 调用 DFS 访问, 也就是说 v 的 DFS 函数结束时间先于 u 的 DFS 结束时间。从而可以考虑在 DFS 调用过程中设定一个时间标记, 在 DFS 调用结束时, 对各结点计时。则祖先的结束时间必然大于子孙的结束时间。

2) 如果 u 是结点 v 的子孙, 则 v 为 u 的祖先, 按上述思路, v 的结束时间大于 u 的结束时间。

3) 如果 u 、 v 没有关系, 则 u 和 v 在拓扑序列的关系任意。

从而按结束时间从大到小, 就可以得到一个拓扑序列。

下面给出利用 DFS 求出各结点的结束时间, 至于拓扑序列, 将结束时间从大到小排序即可得到 (实际上和深度优先遍历算法完全相同, 只不过加入了 $time$ 变量)。

```
bool visited[MAX_VERTEX_NUM];           //访问标记数组
void DFSTraverse(Graph G){
    //对图 G 进行遍历, 访问函数为 visit()
    for(v=0;v<G.vexnum;++v)
        visited[v]=FALSE;               //初始化访问标记数组
    time=0;
    for(v=0;v<G.vexnum;++v)             //本代码中是从 v=0 开始遍历
        if(!visited[v]) DFS(G,v);
}
void DFS(Graph G,int v){
    visited[v]=TRUE;
    visit(v);
    for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w))
        if(!visited[w]){                //w 为 u 的尚未访问的邻接顶点
            DFS(G,w);
        }//if
    time=time+1;finishTime[v]=time;
}
```

10. 解答:

不一定。参考下面的对比后, 请考生自己举出反例。可以对比 Dijkstra 算法和 Prim 算法 (基于邻接矩阵 $G[N][N]$)。

不同之处即中间的两栏更新 Min 算法处, 其余地方一致。

```
bool closed[N]={false}; //将 closed 表初始置为 false
int Min[N]={INF};        /*对应 Dijkstra 中的从 start 点出发到其余各点的最短路径或加
                          /*入 Prim 算法中最小生成树的边。初始化的时候, 都置为正无穷。*/
closed[start]=true;Min[start]=0; /*表示从 start 结点出发执行 Dijkstra 算
/*法或 Prim 算法*/
```



```

for(int i=1;i<N;i++){ //执行 N-1 次, 即开始连接其余的 N-1 个结点
    int k=-1; //保存尚未求解出的结点中与起点距离最短的结点或者到已求出来的
               /*最小生成树中距离最小的那个结点*/
    for(int j=0;j<N;j++){
        if(!closed[j]&&(k==-1||Min[k]>Min[j])) k=j;
        closed[k]=true;
        //得到了 k, 这里考虑了图是连通的, 所以认为 k 一定是存在的, 而不加判定条件
        for(int j=0;j<N;j++){
            //Dijkstra 算法对应的更新 Min 算法
            if(Min[j]>Min[k]+G[k][j])
                Min[j]=Min[k]+G[k][j];
            //Prim 算法对应的更新 Min 算法
            if(Min[j]>G[k][j])
                Min[j]=G[k][j];
        } //for
    } //for
}

```

11. 解答:

考查具体模型中数据结构的应用。很多考生乍看以为是网络的题目, 其实该题本身并没有涉及太多的网络知识点, 只是应用了网络的模型, 实际考查的还是数据结构。)

图题中给出的是一个简单的网络拓扑图, 可以抽象为无向图。

2) 链式存储结构的如下图所示。

弧结点的两种基本形态

Flag=1	Next
ID	
IP	
Metric	

Flag=2	Next
Prefix	
Mask	
Metric	

表头结点
结构示意图

RouterID
LN_link
Next

其数据类型定义如下:

```

typedef struct{
    unsigned int ID, IP;
}LinkNode; //Link 的结构

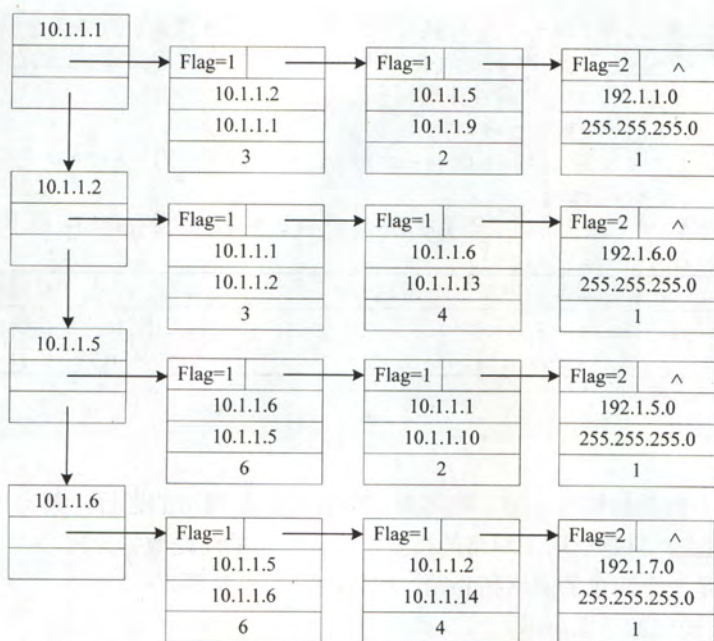
typedef struct{
    unsigned int Prefix, Mask;
}NetNode; //Net 的结构

typedef struct Node{
    int Flag; //Flag=1 为 Link; Flag=2 为 Net
    union{
        LinkNode Lnode;
        NetNode Nnode
    };
    unsigned int Metric;
    struct Node *next;
}ArcNode; //弧结点

typedef struct hNode{
    unsigned int RouterID;
    ArcNode *LN_link;
    Struct hNode *next;
}HNODE; //表头结点

```

对应表的链式存储结构示意图如下。



3) 计算结果如下表所示。

	目的网络	路径	代价(费用)
步骤 1	192.1.1.0/24	直接到达	1
步骤 2	192.1.5.0/24	R1→R3→192.1.5.0/24	3
步骤 3	192.1.6.0/24	R1→R2→192.1.6.0/24	4
步骤 4	192.1.7.0/24	R1→R2→R4→192.1.7.0/24	8

归纳总结

一、关于图的基本操作：

读者可能有疑问，本章很多程序对采用邻接表或者邻接矩阵的存储结构都适用，主要是因为图的基本操作函数中保持相同的参数和返回值，而封闭了内部实现细节。

例如：取 x 邻接顶点 y 的下一个邻接顶点的函数 $\text{NextNeighbor}(G, x, y)$

1. 用邻接矩阵作存储结构

```
int NextNeighbor(MGraph& G, int x, int y){
    if(x!=-1 && y!=-1){
        for(int col=y+1; col<G.vexnum; col++){
            if(G.Edge[x][col]>0 && G.Edge[x][col]<maxWeight)
                return col;
        }
        return -1;
    }
}
```

2. 用邻接表作存储结构

```
int NextNeighbor(ALGraph& G, int x, int y){
    if(x!=-1){
        ArcNode *p=G.vertices[x].first; //顶点x存在
        while(p!=NULL && p->data!=y) //对应边链表第一个边结点
            p=p->next; //寻找邻接顶点y
    }
}
```

```

        p=p->next;
        if(p!=NULL && p->next!=NULL)
            return p->next->data;           //返回下一个邻接顶点
    }
    return -1;
}

```

二、关于图的遍历、连通性、生成树、关键路径的几个要点:

1) 在执行图的遍历时, 由于图中可能存在回路, 且图的任一顶点都可能与其他顶点相连, 所以在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。因此, 需要设置一个辅助数组标 `visited[]` 记顶点是否已被访问过, 避免重复访问。

2) 深度优先搜索时利用回溯法对图遍历, 一般利用递归方法实现, 每当向前递归查找某一邻接结点之前, 必须判断该结点是否访问过。另外, 递归算法均可借助栈来实现非递归算法, 深度优先搜索也不例外, 具体程序在习题 5.3.4 的综合题第 3 题。

3) 广度优先搜索是一种分层的遍历过程, 每向前走一步可能访问一批顶点, 不像深度优先搜索那样有回退的情况。因此, 它不是一个递归的过程。

4) 一个给定的图的邻接矩阵表示是唯一的, 但对于邻接表来说, 如果边的输入先后次序不同, 生成的邻接表表示也不同。

5) 图的最小生成树首先必须是带权连通图, 其次要在 n 个顶点的图中选择 $n-1$ 条边将其连通, 使得其权值总和达到最小, 且不出现回路。

6) 加速某一关键活动不一定能缩短整个工程的工期, 因为 AOE 网中可能存在多条关键路径。可能存在一种特殊的关键活动称为“桥”, 位于所有关键路径上, 只有它加速, 才会缩短整个工期。

思维拓展

【网易有道笔试题】

求一个无向连通图的割点, 割点的定义是, 如果除去此节点和与其相关的边, 无向图不再连通, 描述算法。

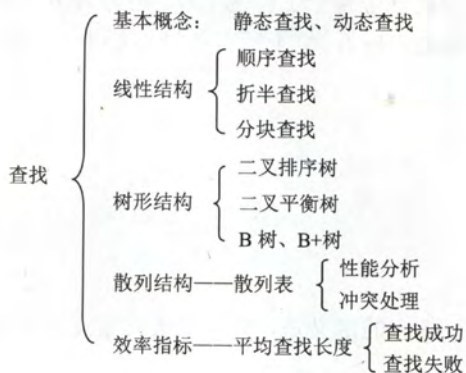
(提示: 要判断一个点是否为割点, 最简单最直接的方法是先把这个点和所有与它相关的边从图中去掉, 再用深搜或广搜来判断剩下的图的连通性即可, 此种方法适合判断给定结点是否为割点; 还有一种比较复杂的方法可以快速找出所有割点, 有兴趣的读者可自行上网查询相关资料。)

第6章 查 找

【考纲内容】

- (一) 查找的基本概念
- (二) 顺序查找法
- (三) 分块查找法
- (四) 折半查找法
- (五) B 树及其基本操作、B+树的基本概念
- (六) 散列 (Hash) 表
- (七) 字符串模式匹配
- (八) 查找算法的分析及应用

【知识框架】



【复习提示】

本章是考研命题的重点，特别是散列查找和折半查找，容易考计算题。对于散列查找，应掌握散列表的构造、冲突处理方法（各种方法的处理过程）、查找成功和查找失败的平均查找长度、散列查找的特征和性能分析。对于折半查找，读者应掌握折半查找的过程、构造判定树、分析查找成功和查找失败的平均查找长度等；另外折半查找只能基于顺序表，因此也比较容易结合顺序表出算法设计题。B 树和 B+树是本章的难点，考纲仅要求了解 B+树的基本概念和性质，而 B 树则要求掌握插入、删除和查找的操作过程，不要求掌握算法。字符串模式匹配部分为新增考点，考生需要重点掌握 next 数组的构造方法。

6.1 查找的基本概念

1) 查找：在数据集中寻找满足某种条件的数据元素的过程称为查找。查找的结果一般分为两种：查找成功，即在数据集中找到了满足条件的数据元素；另一种是查找失败。

2) 查找表（查找结构）：用于查找的数据集合称为查找表，它由同一类型的数据元素（或

记录)组成,可以是一个数组或链表等数据类型。对查找表经常进行的操作一般有四种:①查询某个特定的数据元素是否在查找表中;②检索满足条件的某个特定的数据元素的各种属性;③在查找表中插入一个数据元素;④从查找表中删除某个数据元素。

3) **静态查找表**:如果一个查找表的操作只涉及①和②的操作,则无需动态地修改查找表,此类查找表称为静态查找表。与此对应,需要动态地插入或删除的查找表则称为**动态查找表**。适合静态查找表的查找方法有:顺序查找、折半查找、散列查找等;适合动态查找表的查找方法有:二叉排序树的查找、散列查找等。二叉平衡树和B树都是二叉排序树的改进。二叉排序树与二叉平衡树已在第4章介绍过。

4) **关键字**:数据元素中唯一标识该元素的某个数据项的值,使用基于关键字的查找,查找结果应该是唯一的。比如由一个学生元素构成的数据集合,则学生元素中“学号”这一数据项的值唯一地标识一个学生。

5) **平均查找长度**:在查找的过程中,一次查找的长度是指需要比较的关键字次数,而平均查找长度则是所有查找过程中进行关键字的比较次数的平均值。其数学定义为

$$ASL = \sum_{i=1}^n P_i C_i$$

式中, n 是查找表的长度; P_i 是查找第 i 个数据元素的概率,一般认为每个数据元素的查找概率相等,即 $P_i = 1/n$; C_i 是找到第 i 个数据元素所需进行的比较次数。平均查找长度是衡量查找算法效率的最主要的指标。

6.2 顺序查找和折半查找

6.2.1 顺序查找

顺序查找,又称为线性查找,主要用于在线性表中进行查找。顺序查找通常分为对一般的无序线性表的顺序查找和对按关键字有序的顺序表的顺序查找。下面分别进行讨论。

1. 一般线性表的顺序查找

作为一种最直观的查找方法,其基本思想是从线性表的一端开始,逐个检查关键字是否满足给定的条件。若查找到某个元素的关键字满足给定条件,则查找成功,返回该元素在线性表中的位置;若已经查找到的另一端,还没有查找到符合给定条件的元素,则返回查找失败的信息。下面给出其算法,主要是为了说明其中引入的“哨兵”的作用。

```
typedef struct{           //查找表的数据结构
    ElemType *elem;       //元素存储空间基址,建表时按实际长度分配,0号单元留空
    int      TableLen;    //表的长度
}SSTable;

int Search_Seq(SSTable ST,ElemType key){
    //在顺序表ST中顺序查找关键字为key的元素。若找到则返回该元素在表中的位置
    ST.elem[0]=key;           //“哨兵”
    for(i=ST.TableLen;ST.elem[i]!=key;--i); //从后往前找
    return i; //若表中不存在关键字为key的元素,将查找到i为0时退出for循环
}
```

在上述算法中,将 $ST.elem[0]$ 称为“哨兵”。引入的目的是使得 `Search_Seq` 内的循环不必判断数组是否会越界,因为当满足 $i=0$ 时,循环一定会跳出。需要说明的是,在程序中引入“哨兵”并不是这个算法独有的。通过引入“哨兵”,可以避免很多不必要的判断语句,从而提高程

序效率。

对于有 n 个元素的表，给定值 key 与表中第 i 个元素的关键字相等，即定位第 i 个元素时，需进行 $n-i+1$ 次关键字的比较，即 $C_i=n-i+1$ 。查找成功时，顺序查找的平均长度为

$$ASL_{成功} = \sum_{i=1}^n P_i(n-i+1)$$

当每个元素的查找概率相等时，即 $P_i=1/n$ ，则有

$$ASL_{成功} = \sum_{i=1}^n P_i(n-i+1) = \frac{n+1}{2}$$

查找不成功时，与表中各关键字的比较次数显然是 $n+1$ 次，从而顺序查找不成功的平均查找长度为 $ASL_{不成功}=n+1$ 。

通常，查找表中记录的查找概率并不相等。若能预先得知每个记录的查找概率，则应先对记录的查找概率进行排序，使表中记录按查找概率由小至大重新排列。

综上所述，顺序查找的缺点是当 n 较大时，平均查找长度较大，效率低；优点是对数据元素的存储没有要求，顺序存储或链式存储皆可。对表中记录的有序性也没有要求，无论记录是否按关键码有序均可应用。同时还需注意，对线性的链表只能进行顺序查找。

2. 有序表的顺序查找

如果在查找之前就已经知道表是按关键字有序的，那么当查找失败时可以不用再比较到表的另一端就能返回查找失败的信息，这样能降低顺序查找失败的平均查找长度。

假设表 L 是按关键字从小到大排列的，查找的顺序是从前往后查找，待查找元素的关键字为 key ，当查找到第 i 个元素时，发现第 i 个元素对应的关键字小于 key ，但第 $i+1$ 个元素对应的关键字大于 key ，这时就可以返回查找失败的信息了，因为第 i 个元素之后的元素的关键字均大于 key ，所以表中不存在关键字为 key 的元素。

可以用如图 6-1 所示的判定树来描述有序顺序表的查找过程。树中的圆形结点表示有序顺序表中存在的元素；树中的矩形结点称为失败结点（注意，若有 n 个查找成功结点，则必相应的有 $n+1$ 个查找失败结点），它描述的是那些不在表中的数据值的集合。如果查找到失败结点，则说明查找不成功。

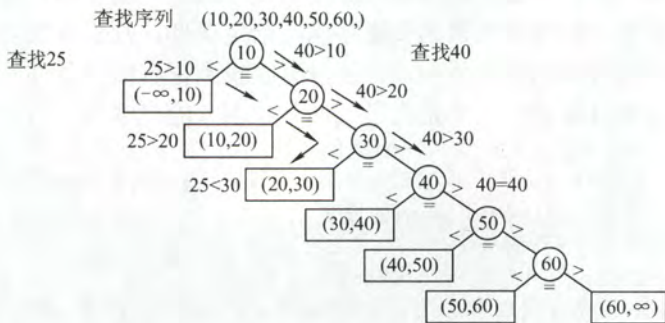


图 6-1 有序顺序表上的顺序查找判定树

在有序表的顺序查找中，查找成功的平均查找长度和一般线性表的顺序查找一样。查找失败时，查找指针一定走到了某个失败结点。这些失败结点是我们虚构的空结点，实际上是不存在的，所以到达失败结点所查找的长度等于它上面一个圆形结点的所在层数。查找不成功的平均查找长度在相等查找概率的情形下有

$$ASL_{不成功} = \sum_{j=1}^n q_j(l_j - 1) = \frac{1+2+\cdots+n+n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

式中, q_j 是到达第 j 个失败结点的概率, 在相等查找概率的情形下, 为 $1/(n+1)$; l_j 是第 j 个失败结点所在的层数。当 $n=6$ 时, $ASL_{不成功}=6/2+6/7=3.86$, 比一般的顺序查找算法好一些。

请注意, 有序表的顺序查找和后面的折半查找的思想是不一样的, 而且有序表的顺序查找中的线性表可以是链式存储结构的。

6.2.2 折半查找

折半查找, 又称为二分查找, 它仅适用于有序的顺序表。基本思路是: 首先将给定值 key 与表中中间位置元素的关键字比较, 若相等, 则查找成功, 返回该元素的存储位置; 若不等, 则所需查找的元素只能在中间元素以外的前半部分或后半部分中 (例如, 在查找表升序排列时, 若给定值 key 大于中间元素的关键字, 则所查找的元素只可能在后半部分)。然后在缩小的范围内继续进行同样的查找, 如此重复直到找到为止, 或者确定表中没有所需要查找的元素, 则查找不成功, 返回查找失败的信息。

算法如下:

```
int Binary_Search(SeqList L, ElemType key) {
    // 在有序表 L 中查找关键字为 key 的元素, 若存在则返回其位置, 不存在则返回 -1
    int low=0, high=L.TableLen-1, mid;
    while (low<=high) {
        mid=(low+high)/2;           // 取中间位置
        if (L.elem[mid]==key)
            return mid;             // 查找成功则返回所在位置
        else if (L.elem[mid]>key)
            high=mid-1;             // 从前半部分继续查找
        else
            low=mid+1;              // 从后半部分继续查找
    }
    return -1;
}
```

例如, 已知 11 个元素的有序表 {7, 10, 13, 16, 19, 29, 32, 33, 37, 41, 43}, 要查找值为 11 和 32 的元素, 指针 low 和 $high$ 分别指向表的下界和上界, mid 则指向表的中间位置 $\lfloor (low+high)/2 \rfloor$ 。以下说明查找 11 的过程 (查找 32 的过程请读者自行分析):

7	10	13	16	19	29	32	33	37	41	43
	↑ low				↑ mid				↑ high	

第一次查找, 将中间位置元素与 key 值比较。因为 $11 < 29$, 说明待查元素若存在, 必在 $[low, mid-1]$ 的范围内, 令指针 $high$ 指向 $mid-1$ 的位置, $high=mid-1=5$, 重新求得 $mid=(1+5)/2=3$, 第二次的查找范围为 $[1, 5]$ 。

7	10	13	16	19	29	32	33	37	41	43
	↑ low		↑ mid			↑ high				

第二次查找, 同样将中间位置元素与 key 值比较。因为 $11 < 13$, 说明待查元素若存在, 必在 $[low, mid-1]$ 的范围内, 令指针 $high$ 指向 $mid-1$ 的位置, $high=mid-1=2$, 重新求得 $mid=(1+2)/2=1$, 第三次的查找范围为 $[1, 2]$ 。

7 10 13 16 19 29 32 33 37 41 43
low ↑ ↑ high
mid ↑

第三次查找, 将中间位置元素与 key 值比较。因为 $11 > 7$, 说明待查元素若存在, 必在 $[mid+1, high]$ 的范围内。令 $low=mid+1=2$, $mid=(2+2)/2=2$, 第四次的查找范围为 $[2, 2]$ 。

7 10 13 16 19 29 32 33 37 41 43
low ↑ ↑ high
↑ mid

第四次查找, 此时子表只含有一个元素, 且 $10 \neq 11$, 故表中不存在待查元素。

折半查找的过程可用图 6-2 所示的二叉树来描述, 称为判定树。树中每个圆形结点表示一个记录, 结点中的值为该记录的关键字值; 树中最下面的叶结点都是方形的, 它表示查找不成功的情况。从判定树可以看出, 查找成功时的查找长度为从根结点到目的结点的路径上的结点数, 而查找不成功时的查找长度为从根结点到对应失败结点的父结点的路径上的结点数; 每个结点值均大于其左子结点值, 且均小于其右子结点值。若有序序列有 n 个元素, 则对应的判定树有 n 个圆形的非叶结点和 $n+1$ 个方形的叶结点。

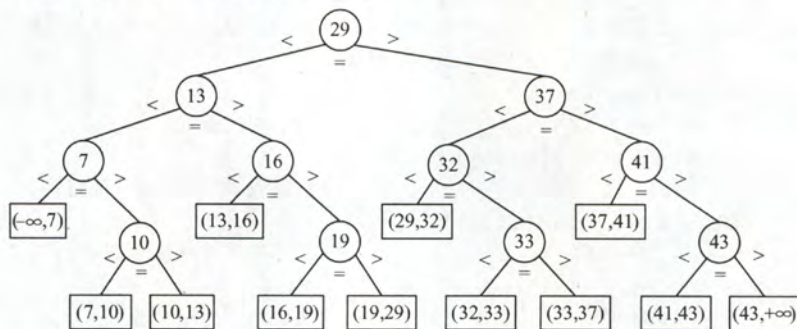


图 6-2 描述折半查找过程的判定树

由上述的分析可知, 用折半查找法查找到给定值的比较次数最多不会超过树的高度。在等概率查找时, 查找成功的平均查找长度为

$$ASL = \frac{1}{n} \sum_{i=1}^n l_i = \frac{1}{n} (1 \times 1 + 2 \times 2 + \cdots + h \times 2^{h-1}) = \frac{n+1}{n} \log_2(n+1) - 1 \approx \log_2(n+1) - 1$$

式中, h 是树的高度, 并且元素个数为 n 时树高 $h = \lceil \log_2(n+1) \rceil$ 。所以, 折半查找的时间复杂度为 $O(\log_2 n)$, 平均情况下比顺序查找的效率要高。

在图 6-2 所示的判定树中, 在等概率的情况下, 查找成功的 $ASL = (1 \times 1 + 2 \times 2 + 3 \times 4 + 4 \times 4) / 11 = 3$, 查找不成功的 $ASL = (3 \times 4 + 4 \times 8) / 12 = 11/3$ 。

因为折半查找需要方便地定位查找区域, 所以适合折半查找的存储结构必须具有随机存取的特性。因此, 该查找法仅适合于线性表的顺序存储结构, 不适合链式存储结构, 且要求元素按关键字有序排列。

6.2.3 分块查找

分块查找, 又称为索引顺序查找, 吸取了顺序查找和折半查找各自的优点, 既有动态结构, 又适于快速查找。

分块查找的基本思想：将查找表分为若干个子块。块内的元素可以无序，但块之间是有序的，即第一个块中的最大关键字小于第二个块中的所有记录的关键字，第二个块中的最大关键字小于第三个块中的所有记录的关键字，依次类推。再建立一个索引表，索引表中的每个元素含有各块的最大关键字和各块中第一个元素的地址，索引表按关键字有序排列。

分块查找的过程分为两步：第一步在索引表中确定待查记录所在的块，可以顺序查找或折半查找索引表；第二步在块内顺序查找。

例如，关键码集合为{88, 24, 72, 61, 21, 6, 32, 11, 8, 31, 22, 83, 78, 54}，按照关键码值为 24、54、78、88，分为四个块和索引表，如图 6-3 所示。

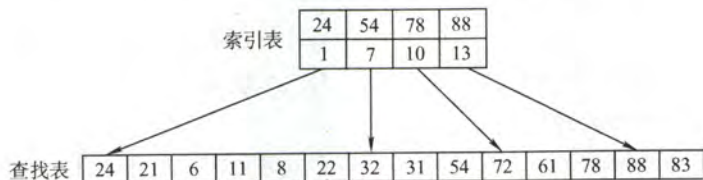


图 6-3 分块查找示意图

分块查找的平均查找长度为索引查找和块内查找的平均长度之和，设索引查找和块内查找的平均查找长度分别为 L_1 、 L_s ，则分块查找的平均查找长度为

$$ASL = L_1 + L_s$$

设将长度为 n 的查找表均匀的分为 b 块，每块有 s 个记录，在等概率的情况下，若在块内和索引表中均采用顺序查找，则平均查找长度为

$$ASL = L_1 + L_s = \frac{b+1}{2} + \frac{s+1}{2} = \frac{s^2 + 2s + n}{2s}$$

此时，若 $s = \sqrt{n}$ ，则平均查找长度取最小值： $\sqrt{n} + 1$ ；若对索引表采用折半查找时，则平均查找长度为

$$ASL = L_1 + L_s = \lceil \log_2(b+1) \rceil + \frac{s+1}{2}.$$

6.2.4 本节试题精选

一、单项选择题

- 顺序查找适合于存储结构为 () 的线性表。
 - 顺序存储结构或链式存储结构
 - 散列存储结构
 - 索引存储结构
 - 压缩存储结构
- 由 n 个数据元素组成的两个表：一个递增有序，一个无序。采用顺序查找算法，对有序表从头开始查找，发现当前元素已不小于待查元素时，停止查找，确定查找不成功，已知查找任一元素的概率是相同的，则在两种表中成功查找 ()。
 - 平均时间后者小
 - 平均时间两者相同
 - 平均时间前者小
 - 无法确定
- 对长度为 n 的有序单链表，若查找每个元素的概率相等，则顺序查找表中任一元素的查找成功的平均查找长度为 ()。
 - $n/2$
 - $(n+1)/2$
 - $(n-1)/2$
 - $n/4$
- 对长度为 3 的顺序表进行查找，若查找第一个元素的概率为 $1/2$ ，查找第二个元素的概率

为 $1/3$, 查找第三个元素的概率为 $1/6$, 则查找任一元素的平均查找长度为 ()。

- A. $5/3$ B. 2 C. $7/3$ D. $4/3$

5. 下列关于二分查找的叙述中, 正确的是 ()。

- A. 表必须有序, 表可以顺序方式存储, 也可以链表方式存储
B. 表必须有序且表中数据必须是整型, 实型或字符型
C. 表必须有序, 而且只能从小到大排列
D. 表必须有序, 且表只能以顺序方式存储

6. 当在一个顺序存储的有序线性表上查找一个数据时, 既可以采用折半查找, 也可以采用顺序查找, 但前者比后者的查找速度 ()。

- A. 必然快 B. 取决于表递增还是递减
C. 在大部分情况下要快 D. 不能确定

7. 折半查找过程所对应的判定树是一棵 ()。

- A. 最小生成树 B. 平衡二叉树 C. 完全二叉树 D. 满二叉树

8. 【2010 年计算机联考真题】

已知一个长度为 16 的顺序表 L, 其元素按关键字有序排列, 若采用折半查找法查找一个 L 中不存在的元素, 则关键字的比较次数最多是 ()。

- A. 4 B. 5 C. 6 D. 7

9. 折半查找和二叉排序树的时间性能 ()。

- A. 相同 B. 有时不相同 C. 完全不同 D. 无法比较

10. 在有 11 个元素的有序表 $A[1, 2, \dots, 11]$ 中进行折半查找 ($\lfloor (low+high)/2 \rfloor$), 查找元素 $A[11]$ 时, 被比较的元素下标依次是 ()。

- A. 6, 8, 10, 11 B. 6, 9, 10, 11 C. 6, 7, 9, 11 D. 6, 8, 9, 11

11. 已知一个有序表 (13, 18, 24, 35, 47, 50, 62, 83, 90, 115, 134), 当二分查找值为 90 的元素时, 查找成功的比较次数为 ()。

- A. 1 B. 2 C. 4 D. 6

12. 对表长为 n 的有序表进行折半查找, 其判定树的高度为 ()。

- A. $\lceil \log_2(n+1) \rceil$ B. $\lfloor \log_2(n+1) \rfloor - 1$ C. $\lceil \log_2 n \rceil$ D. $\lfloor \log_2 n \rfloor - 1$

13. 已知一个长度为 16 的顺序表, 其元素按关键字有序排列, 若采用折半查找查找一个不存在的元素, 则比较的次数至少是 (), 至多是 ()。

- A. 4 B. 5 C. 6 D. 7

14. 具有 12 个关键字的有序表中, 对每个关键字的查找概率相同, 折半查找查找成功的平均查找长度为 (), 折半查找查找失败的平均查找长度为 ()。

- A. $37/12$ B. $35/12$ C. $39/13$ D. $49/13$

15. 【2017 年计算机联考真题】

下列二叉树中, 可能成为折半查找判定树 (不含外部结点) 的是 ()。

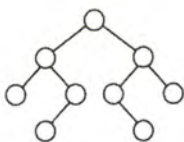
A.



B.



C.



D.



16. 当采用分块查找时,数据的组织方式为()。

- A. 数据分成若干块,每块内数据有序
- B. 数据分成若干块,每块内数据不必有序,但块间必须有序,每块内最大(或最小)的数据组成索引块
- C. 数据分成若干块,每块内数据有序,每块内最大(或最小)的数据组成索引块
- D. 数据分成若干块,每块(除最后一块外)中数据个数需相同

17. 对有 2500 个记录的索引顺序表(分块表)进行查找,最理想的块长为()。

- A. 50
- B. 125
- C. 500
- D. $\lceil \log_2 2500 \rceil$

18. 设顺序存储的某线性表共有 123 个元素,按分块查找的要求等分为 3 块。若对索引表采用顺序查找方法来确定子块,且在确定的子块中也采用顺序查找方法,则在等概率的情况下,分块查找成功的平均查找长度为()。

- A. 21
- B. 23
- C. 41
- D. 62

19. 为提高查找效率,对有 65025 个元素的有序顺序表建立索引顺序结构,在最好情况下查找到表中已有元素最多需要执行()次关键字比较。

- A. 10
- B. 14
- C. 16
- D. 21

20. 【2015 年计算机联考真题】

下列选项中,不能构成折半查找中关键字比较序列的是()。

- A. 500,200,450,180
- B. 500,450,200,180
- C. 180,500,200,450
- D. 180,200,500,450

21. 【2016 年计算机联考真题】

在有 n ($n > 1000$) 个元素的升序数组 A 中查找关键字 x 。查找算法的伪代码如下所示。

```
k=0;
while(k<n 且 A[k]<x) k=k+3;
if(k<n 且 A[k]==x) 查找成功;
else if(k-1<n 且 A[k-1]==x) 查找成功;
    else if(k-2<n 且 A[k-2]==x) 查找成功;
    else 查找失败;
```

本算法与折半查找算法相比,有可能具有更少比较次数的情形是()。

- A. 当 x 不在数组中
- B. 当 x 接近数组开头处
- C. 当 x 接近数组结尾处
- D. 当 x 位于数组中间位置

二、综合应用题

1. 若对有 n 个元素的有序顺序表和无序顺序表进行顺序查找,试就下列三种情况分别讨论两者在相等查找概率时的平均查找长度是否相同?

1) 查找失败。

2) 查找成功,且表中只有一个关键字等于给定值 k 的元素。

3) 查找成功,且表中有若干个关键字等于给定值 k 的元素,要求一次查找能找出所有元素。

2. 设有序顺序表中的元素依次为 017、094、154、170、275、503、509、512、553、612、677、765、897、908。

1) 试画出对其进行折半查找的判定树。

2) 若查找 275 或 684 的元素, 将依次与表中哪些元素比较?

3) 计算查找成功的平均查找长度和查找不成功的平均查找长度。

3. 类比二分查找算法, 设计 k 分查找算法 (k 为大于 2 的整数) 如下: 首先检查 n/k 处 (n 为查找表的长度) 的元素是否等于要搜索的值, 然后检查 $2n/k$ 处的元素……这样, 或者找到要查找的元素, 或者把集合缩小到原来的 $1/k$, 如果未找到要查找的元素, 则继续在得到的集合上进行 k 分查找; 如此进行, 直到找到要查找的元素或查找失败。试求, 查找成功和查找失败的时间复杂度。

4. 已知一个有序顺序表 $A[0 \dots 8N-1]$ 的表长为 $8N$, 并且表中没有关键字相同的数据元素。假设按如下所述的方法查找一个关键字值等于给定值 X 的数据元素: 先在 $A[7]$, $A[15]$, $A[23]$, \dots , $A[8K-1]$, \dots , $A[8N-1]$ 中进行顺序查找, 若查找成功, 则算法报告成功位置并返回; 若不成功, 当 $A[8K-1] < X < A[8 \times (K+1)-1]$ 时, 则可确定一个缩小的查找范围 $A[8K] \sim A[8 \times (K+1)-2]$, 然后可以在这个范围内执行折半查找。特殊情况: 若 $X > A[8N-1]$ 的关键字, 则查找失败。

1) 画出描述上述查找过程的判定树。

2) 计算相等查找概率下查找成功的平均查找长度。

5. 【2013 年计算机联考真题】

设包含 4 个数据元素的集合 $S = \{ \text{"do"}, \text{"for"}, \text{"repeat"}, \text{"while"} \}$, 各元素的查找概率依次为: $p_1=0.35$, $p_2=0.15$, $p_3=0.15$, $p_4=0.35$ 。将 S 保存在一个长度为 4 的顺序表中, 采用折半查找法, 查找成功时的平均查找长度为 2.2。请回答:

1) 若采用顺序存储结构保存 S , 且要求平均查找长度更短, 则元素应如何排列? 应使用何种查找方法? 查找成功时的平均查找长度是多少?

2) 若采用链式存储结构保存 S , 且要求平均查找长度更短, 则元素应如何排列? 应使用何种查找方法? 查找成功时的平均查找长度是多少?

6. 写出折半查找的递归算法。初始调用时, low 为 1, $high$ 为 $ST.length$ 。

7. 线性表中各结点的检索概率不等, 则可用如下策略提高顺序检索的效率: 若找到指定的结点, 将该结点和其前驱结点 (若存在) 交换, 使得经常被检索的结点尽量位于表的前端。试设计在顺序结构和链式结构的线性表上实现上述策略的顺序检索算法。

6.2.5 答案与解析

一、单项选择题

1. A

顺序查找是从表的一端开始向另一端查找。它不要求查找表具有随机存取的特性, 可以是顺序存储结构或链式存储结构。

2. B

对于顺序查找, 不管线性表是有序还是无序的, 成功查找第 1 个元素的比较次数为 1, 成功查找第 2 个元素的比较次数为 2, \dots , 依次类推, 即每个元素查找成功的比较次数只和其位置有关 (与是否有序无关), 因此查找成功的平均时间两者相同。

3. B

在有序单链表上做顺序查找, 查找成功的平均查找长度与在无序顺序表或有序顺序表上做顺序查找的平均查找长度相同, 都是 $(n+1)/2$ 。

4. A

在长度为3的顺序表中, 查找第1个元素的查找长度为1, 查找第2个元素的查找长度为2, 查找第3个元素的查找长度为3, 故有

$$ASL_{成功} = \frac{1}{2} \times 1 + \frac{1}{3} \times 2 + \frac{1}{6} \times 3 = \frac{5}{3}$$

5. D

二分查找通过下标来定位中间位置元素, 故而应采用顺序存储, 且二分查找能够进行的前提是查找表是有序的, 但具体是从大到小还是从大到小的顺序则不做要求。

6. D

折半查找的快体现在一般的情况下, 对于某些特殊的情况, 可能顺序查找会快于折半查找, 并不能绝对地说哪种查找方法快。例如, 查找一个含1000个元素的有序表中第一个元素, 则顺序查找的比较次数为1次, 而折半查找的比较次数却将近10次。

7. B

A显然排除。对于选项C, 考点精析示例中的判定树就不是完全二叉树。由选项C也可以排除选项D, 且满二叉树对结点数有要求。只可能选B。事实上, 由折半查找的定义不难看出, 每次把一个数组从中间结点分割时, 总是把数组分为结点数相差最多不超过1的两个子数组, 从而使得对应的判定树的两棵子树高度差绝对值不超过1, 所以应该是平衡二叉树。

8. B

折半查找法在查找不成功时和给定值进行关键字的比较次数最多为树的高度, 即 $\lfloor \log_2 n \rfloor + 1$ 或 $\lceil \log_2(n+1) \rceil$, 在本题中, $n=16$, 故比较次数最多为5。

注意: 在折半查找判定树中的方形结点是虚构的, 它并不计入比较的次数中。

9. B

折半查找的性能分析可以用二叉判定树来衡量, 平均查找长度和最大查找长度都是 $O(\log_2 n)$; 二叉排序树的查找性能与数据的输入顺序有关, 最好情况下的平均查找长度与折半查找相同, 但最坏情况时, 即形成单支树时, 其查找长度为 $O(n)$ 。

10. B

依据折半查找算法的思想, 第一次 $mid = \lfloor (1+11)/2 \rfloor = 6$, 第二次 $mid = \lfloor [(6+1)+11]/2 \rfloor = 9$, 第三次 $mid = \lfloor [(9+1)+11]/2 \rfloor = 10$, 第四次 $mid = 11$ 。

11. B

开始时low指向13, high指向134, mid指向50, 比较第一次 $90 > 50$, 所以将low指向62, high指向134, mid指向90, 第二次比较找到90。

12. A

对n个结点的判定树, 设结点总数 $n=2^h-1$, 则 $h = \lceil \log_2(n+1) \rceil$ 。

另解: 特殊值代入法。直接将 $n=1$ 和 $n=2$ 的情况代入, 仅有A满足要求。

13. A、B

对于此类题, 有两种做法: 一种是将查找过程中构成的判定树画出来, 然后最小的分支高度对应于最少的比较次数, 最大的分支高度对应于最多的比较次数, 当出现类似于长度为15的顺序表时, 判定树刚好是一棵满树, 此时, 最多比较次数与最少比较次数相等; 另一种方法是直接用公式求出最小的分支高度和最大分支高度, 从前面的讲解不难看出最大分支高度为 $H=$

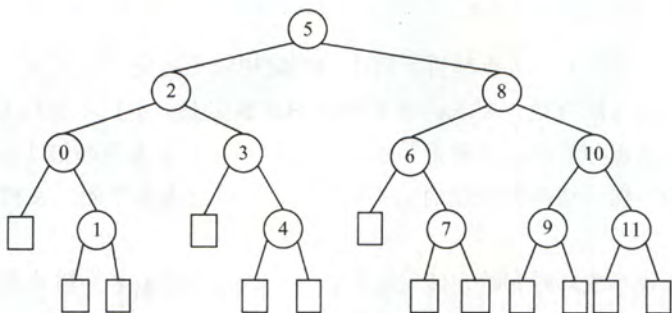
$\lceil \log_2(n+1) \rceil = 5$, 这对应的就是最多比较次数, 然后由于判定树不是一棵满树, 所以至少应该是 4 (由判定树的各分支高度最多相差 1 得出)。

值得注意的是, 若是求查找成功或查找失败的平均查找长度, 则需要画出判定树进行求解。此外, 对长度为 n 的有序表, 采用折半查找时, 查找成功和查找失败的最多比较次数相同, 均为 $\lceil \log_2(n+1) \rceil$, 最少比较次数也相同。

14. A、D

假设有有序表中元素为 $A[0 \cdots 11]$, 不难画出对它进行折半查找的判定树如下图所示, 圆圈是查找成功结点, 方形是虚构的查找失败结点。从而可以求出查找成功的 $ASL = (1+2 \times 2 + 3 \times 4 + 4 \times 5)/12 = 37/12$, 查找失败的 $ASL = (3 \times 3 + 4 \times 10)/13$ 。

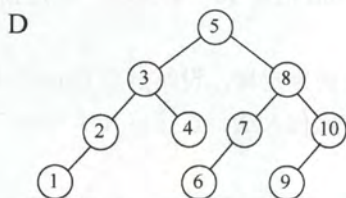
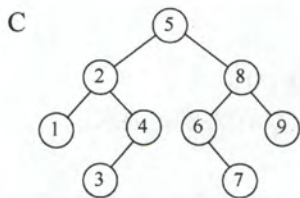
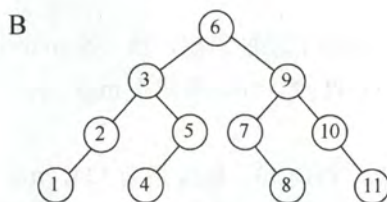
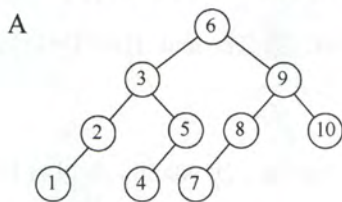
注意: 对于本类题目, 应先根据所给 n 的值, 画出如上图的折半查找判定树。另外, 查找失败结点的 ASL 不是图中的方形结点, 而是方形结点上一层的圆形结点。



15. A

折半查找判定树实际上是一棵二叉排序树, 它的中序序列是一个有序序列。可以在树结点上依次填上相应的元素, 符合折半查找规则的树即是所求。

B 选项 4、5 相加除二向上取整, 7、8 相加除二向下取整, 矛盾。C 选项, 3、4 相加除二向上取整, 6、7 相加除二向下取整, 矛盾。D 选项, 1、10 相加除二向下取整, 6、7 相加除二向上取整, 矛盾。A 选项符合折半查找规则, 正确。



16. B

通常情况下, 在分块查找的结构中, 不要求每个索引块中的元素个数都相等。

17. A

设块长为 b , 索引表包含 n/b 项, 索引表的 $ASL = (n/b+1)/2$, 块内的 $ASL = (b+1)/2$, 总的 $ASL =$

索引表的 $ASL + \text{块内的 } ASL = (b+n/b+2)/2$, 其中对于 $b+n/b$, 由均值不等式知 $b=n/b$ 时有最小值, 此时 $b=\sqrt{n}$ 。则最理想块长为 $\sqrt{2500}=50$ 。

18. B

根据公式 $ASL = L_1 + L_s = \frac{b+1}{2} + \frac{s+1}{2} = \frac{s^2 + 2s + n}{2s}$, 其中 $b=n/s$, $s=123/3$, $n=123$, 代入不

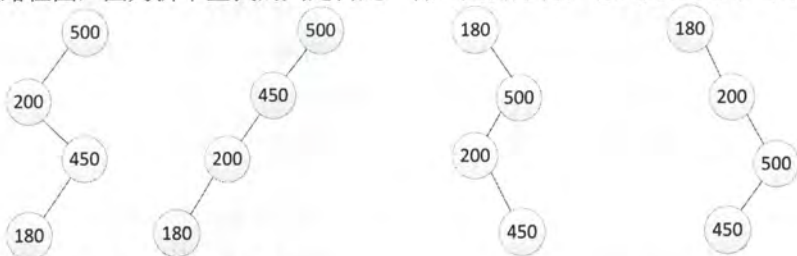
难得出 ASL 为 23。从而选 B。另一方面, 可根据穷举法来一步步模拟。对于 A 块里的元素, 查找过程的第一步是首先找到 A 块, 由于是顺序查找, 找到 A 块只需一步, 然后再在 A 块中顺序查找。因此, A 块内各元素查找长度分别为 2、3、4、...、42。对于 B 块, 采用类似的方法, 但是查找到 B 块要比查找到 A 块多一步, 因此, B 块内各元素查找长度为 3、4、5...43。同理, C 块中各个元素查找长度为 4、5、6、...、44。所以平均查找长度为 $(2+3+4+\dots+42+3+4+5+\dots+43+4+5+6+\dots+44)/123=23$ 。

19. C

为使查找效率最高, 则每个索引块的大小应该是 $\sqrt{65025}=255$, 为每一个块建立索引, 则索引表中索引项的个数为 255。若对索引项和索引块内部都采用折半查找, 则查找效率最高, 为 $\log_2(255+1)+\log_2(255+1)=16$ 。

20. A

画出查找路径图, 因为折半查找的判定树是一棵二叉排序树, 看其是否满足二叉排序树的要求。



很显然, 选项 A 的查找路径不满足。

21. B

送分题。该程序采用跳跃式的顺利查找法查找升序数组中的 x , 显然是 x 越靠前, 比较次数才会越少。

二、综合应用题

1. 解答:

1) 平均查找长度不同。因为有序顺序表查找到其关键字值比要查找值大的元素时就停止查找, 并报告失败信息, 不必查找到表尾; 而无序顺序表必须查找到表尾才能确定查找失败。

2) 平均查找长度相同。两者查找到表中元素的关键字值等于给定值时就停止查找。

3) 平均查找长度不同。有序顺序表中关键字相等的元素相继排列在一起, 只要查找到第一个就可以连续查找到其他关键字相同的元素。而无序顺序表必须查找全部表中元素才能确定相同关键字的元素都找了出来, 所需时间就不相同了。

2. 解答:

1) 判定树如图 6-4 所示。

2) 若查找 275, 依次与表中元素 509、154、275 进行比较, 共比较 3 次。若查找 684, 依次与表中元素 509、677、897、765 进行比较, 共比较 4 次。

3) 在查找成功时, 会找到图中某个圆形结点, 其平均查找长度为

$$ASL_{succ} = \frac{1}{14} \sum_{i=1}^{14} C_i = \frac{1}{14} (1 + 2 \times 2 + 3 \times 4 + 4 \times 7) = \frac{45}{14}$$

在查找失败时, 会找到图中某个方形结点, 但这些结点是虚构的, 最后一次的比较元素为其父结点 (圆形结点), 故其平均查找长度为

$$ASL_{unsucc} = \frac{1}{15} \sum_{i=0}^{15} C'_i = \frac{1}{15} (3 \times 1 + 4 \times 14) = \frac{59}{15}$$

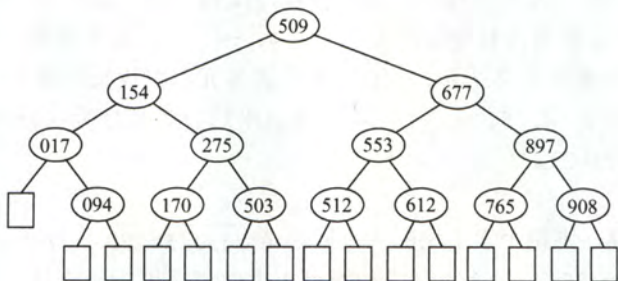


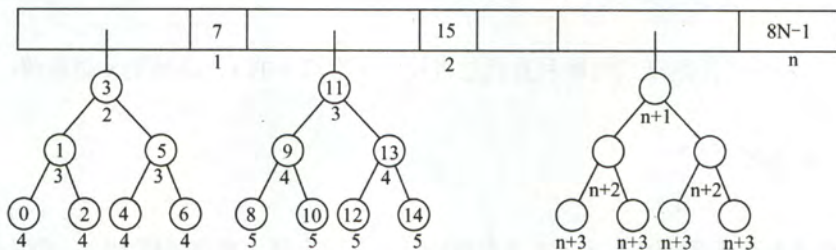
图 6-4

3. 解答:

与二分查找类似, k 分查找法可用 k 叉树来描述。 k 分查找法在查找成功时进行比较的关键字数最多不超过树的深度, 而具有 n 个结点的 k 叉树的深度为 $\lfloor \log_k n \rfloor + 1$, 所以 k 分查找法在查找成功时和给定值进行比较的关键字数至多为 $\lfloor \log_k n \rfloor + 1$, 即时间复杂度为 $O(\log_k n)$ 。同理, 查找不成功时, 和给定值进行比较的关键字数也至多为 $\lfloor \log_k n \rfloor + 1$, 故时间复杂度也为 $O(\log_k n)$ 。

4. 解答:

1) 先在 $A[7]$ 、 $A[15]$ 、 \dots 、 $A[8N-1]$ 内顺序查找, 再在区间内折半查找。则相应的判定树如下图所示。其中, 每一个关键字下的数字为其查找成功时的关键字比较次数。



2) 等查找概率下, 平均每个关键字的查找成功的概率为 $1/8n$; $0 \sim 7$ 之间的关键字, 顺序比较 1 次后, 进行折半查找 $2+3 \times 2+4 \times 4$; $8 \sim 15$ 之间的关键字, 是先顺序比较 2 次后, 再进入折半查找; 依次类推, $8(N-1) \sim 8N-1$ 之间的关键字, 是先顺序比较了 n 次, 再进入折半查找。如上图所示:

故查找成功的平均查找长度为

$$\begin{aligned} ASL_{succ} &= \frac{1}{8n} \sum_{i=0}^{8n-1} C_i = \frac{1}{8n} \left(\sum_{i=1}^n i + \sum_{i=2}^{n+1} i + 2 \sum_{i=3}^{n+2} i + 4 \sum_{i=4}^{n+3} i \right) \\ &= \frac{1}{8n} \left(\sum_{i=1}^n (i + (i+1) + 2(i+2) + 4(i+3)) \right) \\ &= \frac{1}{8n} \sum_{i=1}^n (8i + 17) = \frac{1}{n} \sum_{i=1}^n i + \frac{17}{8} = \frac{n+1}{2} + \frac{17}{8} \end{aligned}$$

5. 解答:

1) 折半查找要求元素有序顺序存储, 若各个元素的查找概率不同, 折半查找的性能不一定优于顺序查找。采用顺序查找时, 元素按其查找概率的降序排列时查找长度最小。

采用顺序存储结构, 数据元素按其查找概率降序排列。采用顺序查找方法。

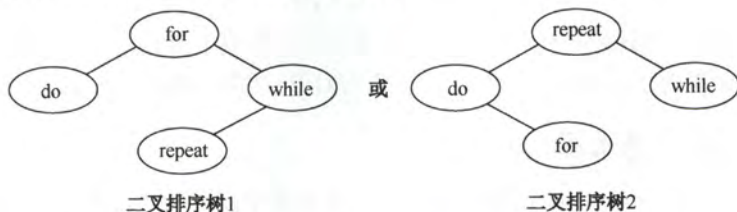
查找成功时的平均查找长度 $=0.35 \times 1 + 0.35 \times 2 + 0.15 \times 3 + 0.15 \times 4 = 2.1$ 。

此时, 显然查找长度比折半查找的更短。

2) 答案一: 采用链式存储结构时, 只能采用顺序查找, 其性能和顺序表一样, 类似于上题。数据元素按其查找概率降序排列, 构成单链表。采用顺序查找方法。

查找成功时的平均查找长度 $=0.35 \times 1 + 0.35 \times 2 + 0.15 \times 3 + 0.15 \times 4 = 2.1$ 。

答案二: 还可以构造成二叉排序树的形式。采用二叉链表的存储结构, 构造二叉排序树, 元素的存储方式见下图。采用二叉排序树的查找方法。



查找成功时的平均查找长度 $=0.15 \times 1 + 0.35 \times 2 + 0.35 \times 2 + 0.15 \times 3 = 2.0$ 。

6. 解答:

算法的基本思想: 根据查找的起始位置和终止位置, 将查找序列一分为二, 判断所查找的关键字在哪一部分, 然后用新的序列起始位置和终止位置递归求解。

算法的代码:

```

typedef struct{                                //查找表的数据结构
    ElemType *elem;                            //存储空间基址, 建表时按实际长度分配, 0号留空
    int length;                                //表的长度
} SSTable;

int BinSearchRec(SSTable ST, ElemType key, int low, int high){
//在有序表中递归折半查找其关键字为key的元素, 返回其在表中序号
    if(low > high)
        return 0;
    mid = (low + high) / 2;                    //取中间位置
    if(key > ST.elem[mid])                    //向后半部分查找
        Search(ST, key, mid + 1, high);
    else if(key < ST.elem[mid])              //向前半部分查找
        Search(ST, key, low, mid - 1);
    else                                     //查找成功
        return mid;
}
  
```

算法把规模为 n 的复杂问题经过多次递归调用转化为规模减半的子问题求解。时间复杂度为 $O(\log_2 n)$, 算法中用到了一个递归工作栈, 其规模与递归深度有关, 也是 $O(\log_2 n)$ 。

7. 解答:

算法的基本思想: 检索时可先从表头开始向后顺序扫描, 若找到指定的结点, 将该结点和其前趋结点 (若存在的话) 交换。采用顺序表存储结构的算法实现如下:

```

int SeqSrch(RcdType R[], ElemType k){
//顺序查找线性表, 找到后和其前面的元素交换
  
```



```

int i=0;
while((R[i].key!=k)&&(i<n))
    i++;
//从前向后顺序查找指定结点
if(i<n&&i>0){
    //若找到,则交换
    temp=R[i];R[i]=R[i-1];R[i-1]=temp;
    return --i;
    //交换成功,返回交换后的位置
} else return -1;
//交换失败

```

链表的实现方式请读者自行思考。注意：链表方式实现的基本思想与上述思想相似，但注意用链表实现时交换两个结点前需要保存指向前一结点的指针。

6.3 B 树和 B+树

考试大纲对 B 树和 B+树的要求各不相同，重点在于考查 B 树，不仅要求理解 B 树的基本特点，还要求掌握 B 树的建立、插入和删除操作，而对 B+树则只考查基本概念。

6.3.1 B 树及其基本操作

B 树，又称为多路平衡查找树，B 树中所有结点的孩子结点数的最大值称为 B 树的阶，通常用 m 表示。一棵 m 阶 B 树或为空树，或为满足如下特性的 m 叉树：

- 1) 树中每个结点至多有 m 棵子树（即至多含有 $m-1$ 个关键字）。
- 2) 若根结点不是终端结点，则至少有两棵子树。
- 3) 除根结点外的所有非叶结点至少有 $\lceil m/2 \rceil$ 棵子树（即至少含有 $\lceil m/2 \rceil - 1$ 个关键字）。
- 4) 所有非叶结点的结构如下：

n	P_0	K_1	P_1	K_2	P_2	\dots	K_n	P_n
-----	-------	-------	-------	-------	-------	---------	-------	-------

其中， K_i ($i=1, 2, \dots, n$) 为结点的关键字，且满足 $K_1 < K_2 < \dots < K_n$ ； P_i ($i=0, 1, \dots, n$) 为指向子树根结点的指针，且指针 P_{i-1} 所指子树中所有结点的关键字均小于 K_i ， P_i 所指子树中所有结点的关键字均大于 K_i ， n ($\lceil m/2 \rceil - 1 \leq n \leq m-1$) 为结点中关键字的个数。

5) 所有的叶结点都出现在同一层次上，并且不带信息（可以看做是外部结点或者类似于折半查找判定树的查找失败结点，实际上这些结点不存在，指向这些结点的指针为空）。

B 树是所有结点的平衡因子均等于 0 的多路查找树，如图 6-5 所示为一棵 3 阶 B 树。其中底层方形结点表示叶结点，在这些结点中没有存储任何信息。



图 6-5 3 阶 B 树示意图

1. B 树的高度（磁盘存取次数）

由下一节可知，B 树中的大部分操作所需的磁盘存取次数与 B 树的高度成正比。

下面来分析 B 树在不同的情况下的高度。当然，首先应该明确 B 树的高度不包括最后的不带任何信息的叶结点所处的那一层。（注：有些书上对于 B 树的高度定义中是包括了最后的那一层

的,本书中不包括这一层,所以希望读者在阅读的时候注意一下)。

如果 $n \geq 1$, 则对任意一棵包含 n 个关键字、高度为 h 、阶数为 m 的 B 树:

1) 因为 B 树中每个结点最多有 m 棵子树, $m-1$ 个关键字, 所以在一棵高度为 h 的 m 阶 B 树中关键字的个数应满足 $n \leq (m-1) * (1 + m + m^2 + \dots + m^{h-1}) = m^h - 1$, 因此有

$$h \geq \log_m(n+1)$$

2) 若让每个结点中的关键字个数达到最少, 则容纳同样多关键字的 B 树的高度可达到最大。由 B 树定义: 第一层至少有 1 个结点; 第二层至少有 2 个结点; 除根结点以外的每个非终端结点至少有 $\lceil m/2 \rceil$ 棵子树, 则第三层至少有 $2\lceil m/2 \rceil$ 个结点……第 $h+1$ 层至少有 $2(\lceil m/2 \rceil)^{h-1}$ 个结点, 注意到第 $h+1$ 层是不包含任何信息的叶结点。对于关键字个数为 n 的 B 树, 叶结点即查找不成功的结点为 $n+1$, 由此有 $n+1 \geq 2(\lceil m/2 \rceil)^{h-1}$, 即 $h \leq \log_{\lceil m/2 \rceil}((n+1)/2) + 1$ 。

例如, 假设一棵 3 阶 B 树, 共有 8 个关键字, 则其高度范围为 $2 \leq h \leq 3.17$ 。

2. B 树的查找

在 B 树上进行查找与二叉查找树很相似, 只是每个结点都是多个关键字的有序表, 在每个结点上所做的不是两路分支决定, 而是根据该结点的子树所做的多路分支决定。

B 树的查找包含两个基本操作: ①在 B 树中找结点; ②在结点内找关键字。由于 B 树常存储在磁盘上, 则前一个查找操作是在磁盘上进行的, 而后一个查找操作是在内存中进行的, 即在找到目标结点后, 先将结点中的信息读入内存, 然后再采用顺序查找法或折半查找法查找等于 K 的关键字。

在 B 树上上查找某个结点后, 先在有序表中进行查找, 若找到则查找成功, 否则按照对应的指针信息到所指的子树中去查找 (例如, 在图 6-5 中查找到第一层的第一个结点时, 若发现关键字大于 18 而小于 33, 则在这个结点上查找失败, 将根据 18 与 33 之间的指针到结点的第二个子树中继续查找)。当查找到叶结点时 (对应的指针为空指针), 则说明树中没有对应的关键字, 查找失败。

3. B 树的插入

与二叉查找树的插入操作相比, B 树的插入操作要复杂得多。在二叉查找树中, 仅需查找到需插入的终端结点的位置。但是, 在 B 树中找到插入的位置后, 并不能简单地将其添加到终端结点中去, 因为此时可能会导致整棵树不再满足 B 树中定义中的要求。将关键字 key 插入到 B 树的过程如下:

1) 定位: 利用前述的 B 树查找算法, 找出插入该关键字的最底层中某个非叶结点 (注意, B 树中的插入关键字一定是插入在最底层中的某个非叶结点内)。

2) 插入: 在 B 树中, 每个非失败结点的关键字个数都在 $[\lceil m/2 \rceil - 1, m-1]$ 之间。当插入后的结点关键字个数小于 m , 则可以直接插入; 插入后检查被插入结点内关键字的个数, 当插入后的结点关键字个数大于 $m-1$ 时, 则必须对结点进行分裂。

分裂的方法是: 取一个新结点, 将插入 key 后的原结点从中间位置将其中的关键字分为两部分, 左部分包含的关键字放在原结点中, 右部分包含的关键字放到新的结点中, 中间位置 ($\lceil m/2 \rceil$) 的结点插入到原结点的父结点中。若此时导致其父结点的关键字个数也超过了上限, 则继续进行这种分裂操作, 直至这个过程传到根结点为止, 这样导致 B 树高度增 1。

对于 $m=3$ 的 B 树, 所有结点中最多有 $m-1=2$ 个关键字, 若某结点中已有两个关键字时, 则结点已满, 如图 6-6 (a) 所示。当插入一个关键字 60 后, 结点内的关键字个数超出了 $m-1$, 如图 6-6 (b) 所示, 此时必须进行结点分裂, 分裂的结果如图 6-6 (c) 所示。

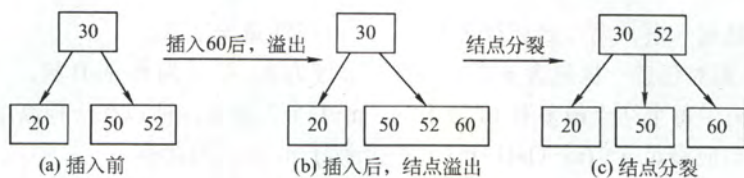


图 6-6 结点的“分裂”示意

4. B 树的删除

B 树中的删除操作与插入操作类似, 但要稍微复杂些, 要使得删除后的结点中的关键字个数 $\geq \lceil m/2 \rceil - 1$, 因此将涉及结点的“合并”问题。

当所删除的关键字 k 不在终端结点 (最底层非叶结点) 中时, 有下列几种情况:

- 1) 如果小于 k 的子树中关键字个数 $> \lceil m/2 \rceil - 1$, 则找出 k 的前驱值 k' , 并且用 k' 来取代 k , 再递归地删除 k' 即可。
- 2) 如果大于 k 的子树中关键字个数 $> \lceil m/2 \rceil - 1$, 则找出 k 的后继值 k' , 并且用 k' 来取代 k , 再递归地删除 k' 即可。
- 3) 如果前后两个子树中关键字个数均为 $\lceil m/2 \rceil - 1$, 则直接将两个子结点合并, 直接删除 k 即可。图 6-7 所示为某 4 阶 B 树的一部分。

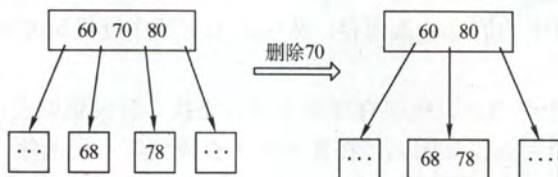


图 6-7 B 树中删除非终端结点关键字的合并

当被删除的关键字在终端结点 (最底层非叶结点) 中时, 有下列几种情况:

- 1) 直接删除关键字: 若被删除关键字所在结点的关键字个数 $> \lceil m/2 \rceil - 1$, 表明删除该关键字后仍满足 B 树的定义, 则直接删去该关键字。
- 2) 兄弟够借: 若被删除关键字所在结点删除前的关键字个数 $= \lceil m/2 \rceil - 1$, 且与此结点相邻的右 (左) 兄弟结点的关键字个数 $\geq \lceil m/2 \rceil$, 需要调整该结点、右 (左) 兄弟结点以及其双亲结点 (父子换位法), 以达到新的平衡, 如图 6-8 (a) 所示。

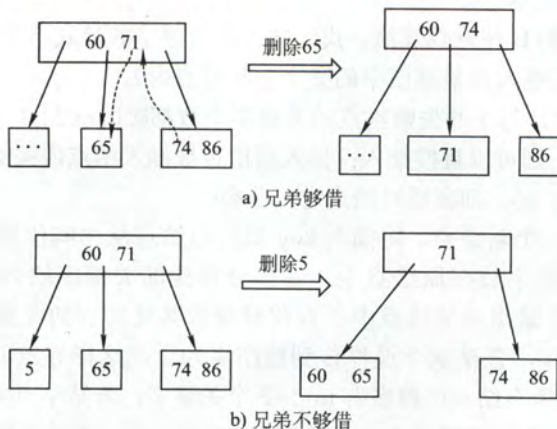


图 6-8 B 树中删除终端结点关键字的示意图

3) 兄弟不够借: 若被删除关键字所在结点删除前的关键字个数= $\lceil m/2 \rceil - 1$, 且此时与该结点相邻的右(左)兄弟结点的关键字个数= $\lceil m/2 \rceil - 1$, 则将关键字删除后与右(左)兄弟结点及双亲结点中的关键字进行合并, 如图 6-8 (b) 所示。

在合并的过程中, 双亲结点中的关键字个数会减少。若其双亲结点是根结点并且关键字个数减少至 0 (根结点关键字个数为 1 时, 有 2 棵子树), 则直接将根结点删除, 合并后的新结点成为根; 若双亲结点不是根结点, 且关键字个数减少到 $\lceil m/2 \rceil - 2$, 又要与它自己的兄弟结点进行调整或合并操作, 并重复上述步骤, 直至符合 B 树的要求为止。

6.3.2 B+树基本概念

B+树是应数据库所需而出现的一种 B 树的变形树。

一棵 m 阶的 B+树需满足下列条件:

- 1) 每个分支结点最多有 m 棵子树 (子结点)。
- 2) 非叶根结点至少有两棵子树, 其他每个分支结点至少有 $\lceil m/2 \rceil$ 棵子树。
- 3) 结点的子树个数与关键字个数相等。
- 4) 所有叶结点包含全部关键字及指向相应记录的指针, 而且叶结点中将关键字按大小顺序排列, 并且相邻叶结点按大小顺序相互链接起来。
- 5) 所有分支结点 (可看成是索引的索引) 中仅包含它的各个子结点 (即下一级的索引块) 中关键字的最大值及指向其子结点的指针。

m 阶的 B+树与 m 阶的 B 树的主要差异在于:

- 1) 在 B+树中, 具有 n 个关键字的结点只含有 n 棵子树, 即每个关键字对应一棵子树; 而在 B 树中, 具有 n 个关键字的结点含有 $(n+1)$ 棵子树。
- 2) 在 B+树中, 每个结点 (非根内部结点) 关键字个数 n 的范围是 $\lceil m/2 \rceil \leq n \leq m$ (根结点: $1 \leq n \leq m$), 在 B 树中, 每个结点 (非根内部结点) 关键字个数 n 的范围是 $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ (根结点: $1 \leq n \leq m - 1$)。

3) 在 B+树中, 叶结点包含信息, 所有非叶结点仅起到索引作用, 非叶结点中的每个索引项只含有对应子树的最大关键字和指向该子树的指针, 不含有该关键字对应记录的存储地址。

4) 在 B+树中, 叶结点包含了全部关键字, 即在非叶结点中出现的关键字也会出现在叶结点中; 而在 B 树中, 叶结点包含的关键字和其他结点包含的关键字是不重复的。

如图 6-9 所示为一棵 4 阶 B+树的示例。从图中可以看出, 分支结点的某个关键字是其子树中最大关键字的副本。通常在 B+树中有两个头指针: 一个指向根结点, 另一个指向关键字最小的叶结点。因此, 可以对 B+树进行两种查找运算: 一种是从最小关键字开始的顺序查找, 另一种是从根结点开始, 进行多路查找。

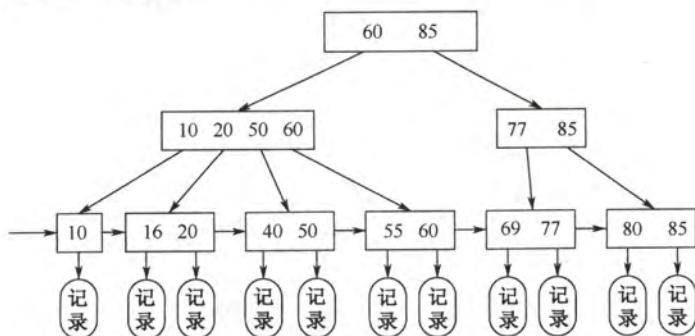


图 6-9 B+树结构示意图

B+树的查找、插入和删除操作和 B 树基本类似。只是在查找过程中, 如果非叶结点上的关键字值等于给定值时并不终止, 而是继续向下查找直到叶结点上的该关键字为止。所以, 在 B+ 树中查找, 无论查找成功与否, 每次查找都是一条从根结点到叶结点的路径。

6.3.3 本节试题精选

一、单项选择题

1. 图 6-10 所示是一棵 ()。

- A. 4 阶 B 树
- B. 4 阶 B+ 树
- C. 3 阶 B 树
- D. 3 阶 B+ 树

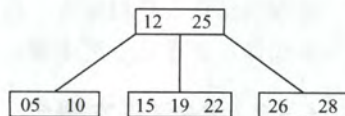


图 6-10 树

2. 下列关于 m 阶 B 树的说法错误的是 ()。

- A. 根结点至多有 m 棵子树
- B. 所有叶结点都在同一层次上
- C. 非叶结点至少有 $m/2$ (m 为偶数) 或 $(m+1)/2$ (m 为奇数) 棵子树
- D. 根结点中的数据是有序的

3. 以下关于 m 阶 B 树的说法中正确的是 ()。

- I. 每个结点至少有两棵非空子树
- II. 树中每个结点至多有 $m-1$ 个关键字
- III. 所有叶结点在同一层
- IV. 当插入一个元素引起 B 树结点分裂后, 树长高一层

- A. I、II
- B. II、III
- C. III、IV
- D. I、II、IV

4. 【2009 年计算机联考真题】

下列叙述中, 不符合 m 阶 B 树定义要求的是 ()。

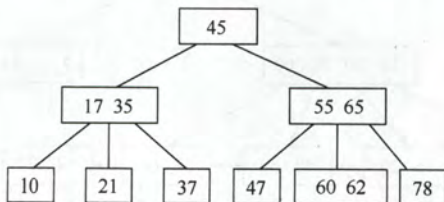
- A. 根结点最多有 m 棵子树
- B. 所有叶结点都在同一层上
- C. 各结点内关键字均升序或降序排列
- D. 叶结点之间通过指针链接

5. 当在一棵 m 阶 B 树中做插入操作时, 若一个结点中的关键字个数等于 (), 则必须分裂成两个结点, 当向一棵 m 阶的 B 树做删除操作时, 若一个结点中的关键字个数等于 (), 则可能需要同它的左兄弟或右兄弟结点合并成一个结点。

- A. $m, \lceil m/2 \rceil - 2$
- B. $m-1, \lceil m/2 \rceil - 1$
- C. $m+1, \lceil m/2 \rceil$
- D. $m/2, \lceil m/2 \rceil + 1$

6. 【2012 年计算机联考真题】

已知一棵 3 阶 B 树, 如下图所示。删除关键字 78 得到一棵新 B 树, 其最右叶结点中的关键字是 ()。



- A. 60
- B. 60, 62
- C. 62, 65
- D. 65

7. 具有 n 个关键字的 m 阶 B 树, 应有 () 个叶结点。

- A. $n+1$ B. $n-1$ C. mn D. $nm/2$
8. 高度为5的3阶B树至少有()个结点,至多有()个结点。
A. 32 B. 31 C. 120 D. 121
9. 【2013年计算机联考真题】
在一棵高度为2的5阶B树中,所含关键字的个数最少是()。
A. 5 B. 7 C. 8 D. 14
10. 【2014年计算机联考真题】
在一棵具有15个关键字的4阶B树中,含关键字的结点个数最多是()。
A. 5 B. 6 C. 10 D. 15
11. 含有 n 个非叶结点的 m 阶B树中至少包含()个关键字。
A. $n(m+1)$ B. n C. $n(\lceil m/2 \rceil - 1)$ D. $(n-1)(\lceil m/2 \rceil - 1) + 1$
12. 已知一棵5阶B树中共有53个关键字,则树的最大高度为(),最小高度为()。
A. 2 B. 3 C. 4 D. 5
13. 已知一棵3阶B树中有2047个关键字,则此B树的最大高度为(),最小高度为()。
A. 11 B. 10 C. 8 D. 7
14. 下列关于B树和B+树的叙述中,不正确的是()。
A. B树和B+树都能有效地支持顺序查找
B. B树和B+树都能有效地支持随机查找
C. B树和B+树都是平衡的多叉树
D. B树和B+树都可以用于文件索引结构
15. 【2016年计算机联考真题】
B+树不同于B树的特点之一是()。
A. 能支持顺序查找 B. 结点中含有关键字
C. 根结点至少有两个分支 D. 所有叶结点都在同一层上
16. 【2017年计算机联考真题】
下列应用中,适合使用B+树的是()。
A. 编译器中的词法分析 B. 关系数据库系统中的索引
C. 网络中的路由表快速查找 D. 操作系统的磁盘空闲块管理

二、综合应用题

1. 给定一组关键字{20, 30, 50, 52, 60, 68, 70}, 给出创建一棵3阶B树的过程。

2. 对如图6-11所示的3阶B树,依次执行下列操作,

画出各步操作的结果。

- 1) 插入90 2) 插入25 3) 插入45
4) 删除60 5) 删除80

3. 利用B树做文件索引时,若假设磁盘页块的大小

是4000字节(实际应是2的次幂,题目是为了计算方便),指示磁盘地址的指针需要5个字节。现有20 000 000个记录构成的文件,每个记录为200字节,其中包括关键字5个字节。

试问在此采用B树作索引的文件中,B树的阶数应为多少?假定文件数据部分未按关键字有序排列,则索引部分需要占用多少磁盘页块?

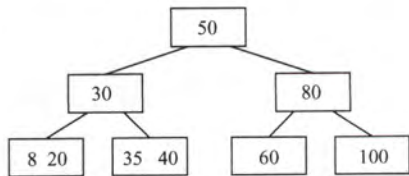


图 6-11

6.3.4 答案与解析

一、单项选择题

1. A

关键字数目比子树数目少 1, 所以不是 B+ 树, 而是 B 树。又有 m 阶 B 树结点关键字数最多为 $m-1$, 有一个结点关键字个数为 3, 所以, 不可能为 3 阶。

2. C

除根结点之外的所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树。对于根结点, 最多有 m 棵子树, 若其不是叶结点, 则至少有 2 棵子树。

3. B

每个非根的内部结点必须至少有 $\lceil m/2 \rceil$ 棵子树, 而根结点至少要有两棵子树, 所以 I 不正确。II、III 显然正确。对于 IV, 当插入一个元素引起 B 树结点分裂后, 只要从根结点到该元素插入位置的路径上至少有 1 个结点未滿, B 树就不会长高, 所以 IV 不正确。

4. D

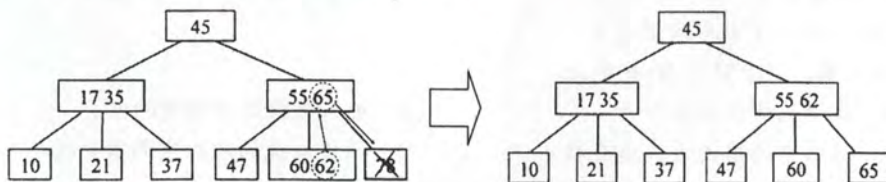
m 阶 B 树不要求将各叶结点之间用指针链接。选项 D 描述的实际上是 B+ 树。

5. A

由于 B 树中每个结点内关键字个数最多为 $m-1$, 所以当关键字个数大于 $m-1$ 时, 就应该分裂。而每个结点内关键字个数至少为 $\lceil m/2 \rceil - 1$ 个, 所以当关键字个数少于 $\lceil m/2 \rceil - 1$ 时, 应该与其他结点合并。若将本题题干改为 B+ 树, 请读者自行思考上述问题的解答。

6. D

对于图中所示的 3 阶 B 树, 被删关键字 78 所在结点在删除前的关键字个数 $= 1 = \lceil 3/2 \rceil - 1$, 且其左兄弟结点的关键字个数 $= 2 \geq \lceil 3/2 \rceil$, 属于“兄弟够借”的情况, 则需把该结点的左兄弟结点中最大的关键字上移到双亲结点中, 同时把双亲结点中大于上移关键字的关键字下移到要删除关键字的结点中, 这样就达到了新的平衡, 如下图所示。



7. A

B 树的叶结点对应查找失败的情况, 对有 n 个关键字的查找集合进行查找, 失败的可能性有 $n+1$ 种。

8. B、D

由 m 阶 B 树的性质可知, 根结点至少有 2 棵子树; 根结点之外的所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树, 则结点数最少的时候, 3 阶 B 树形状至少类似于一棵满二叉树, 也即高度为 5 的 B 树至少有 $2^5 - 1 = 31$ 个结点。又每个结点最多有 m 棵子树, 所以当结点数最多的时候, 3 阶 B 树形状类似于满三叉树, 结点数为 $(3^5 - 1)/2 = 121$ (请注意, 这里求的是结点数, 而不是关键字数, 如果是求关键字的数目, 则还应把每个结点中关键字个数的上下界确定出来)。

9. A

对于 5 阶 B 树, 根结点只有达到 5 个关键字时才能产生分裂, 成为高度为 2 的 B 树, 因此高度为 2 的 5 阶 B 树所含关键字的个数最少是 5。

注意：要与第1题相区别，论坛上有些同学对此题的理解存在偏差。第1题要根据图示给出阶数和属于哪种树。对于该题所述的5阶B树，不要误认为：“存在至少有一个含关键字的节点中的关键字达到4”才符合5阶B树的要求，因为5阶B树中的各个节点包含的关键字个数最少为2 ($\lceil 5/2 \rceil - 1 = 2$)，最多为4 ($5 - 1 = 4$)。当5阶B树各个节点包含的关键字个数为2时也满足5阶B树的要求，存在至少有一个含关键字的节点中的关键字达到4同样也符合5阶B树的要求。（此时若题目给定了关键字个数（如第10题），则可以计算出该树的含关键字节点个数将达到最多；若各节点的关键字个数达到4，则该树的总关键字个数一定的条件下可以计算出含关键字节点的最少的情况。这与“当树高一定的情况下，求含关键字的最多（或最少）个数（或相反“给定关键字个数求树高”如第12题）”的思路都是一样的。重中之重是对B树和B+树的定义及特性一定要透彻理解。如此，相信读者方可对第8、9、10、11、12和13题应该游刃有余）（对此存在疑问的同学请回看B树的结构及特性，不要将二者混淆）。

10. D

关键字数量不变，要求结点数量最多，那么即每个结点中含关键字的数量最少。根据4阶B树的定义，根结点最少含1个关键字，非根结点中至少含 $\lceil 4/2 \rceil - 1 = 1$ 个关键字，所以每个结点中，关键字数量最少都为1个，即每个结点都有2个分支，类似与排序二叉树，而15个结点正好可以构造一个4层的4阶B树，使得叶结点全在第四层，符合B树定义，因此选D。

11. D

除根结点外，m阶B树中的每个非叶结点至少有 $\lceil m/2 \rceil - 1$ 个关键字，根结点至少有一个关键字，所以总共包含的关键字最少个数 $= (n-1)(\lceil m/2 \rceil - 1) + 1$ 。

注意：由以上题目可知B树和B+树的定义和性质尤为重要，需要熟练掌握。

12. C、B

5阶B树中共有53个关键字，由最大高度公式 $H \leq \log_{\lceil m/2 \rceil} ((n+1)/2) + 1$ ，算出最大高度 $H \leq \log_3[(53+1)/2] + 1 = 4$ ，即最大高度为4；由最小高度公式 $h \geq \log_m(n+1)$ ，算数最小高度 $h \geq \log_5 54 = 2.5$ ，从而最小高度为3。

13. A、D

利用前面的公式最小高度 $h \geq \log_m(n+1)$ 和最大高度 $H \leq \log_{\lceil m/2 \rceil} [(n+1)/2] + 1$ ，易算出最大高度 $H \leq \log_2[(2047+1)/2] + 1 = 11$ ，最小高度 $h \geq \log_3 2048 = 6.9$ ，从而最小高度取7。（注意，有些辅导书对本题计算出的高度比这里给出的答案要多1，这是因为它们对B树的高度定义中把最底层不包含任何关键字的叶结点也算了进去。）

14. A

由于B+树的所有叶结点中包含了全部的关键字信息，且叶结点本身依关键字从小到大顺序链接，可以进行顺序查找，而B树不支持顺序查找。

15. A

由于B+树的所有叶结点中包含了全部的关键字信息，且叶结点本身依关键字从小到大顺序链接，可以进行顺序查找，而B树不支持顺序查找（只支持多路查找）。

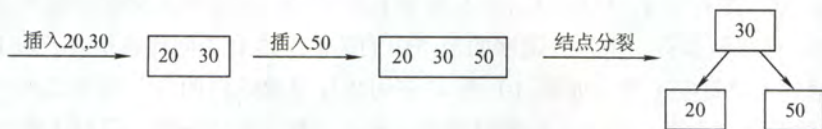
16. B

B+树是应文件系统所需而产生的B-树的变形，前者比后者更加适用于实际应用中的操作系统的文件索引和数据库索引，因为前者磁盘读写代价更低，查询效率更加稳定。编译器中的词法分析使用有穷自动机和语法树。网络中的路由表快速查找主要靠高速缓存、路由表压缩技术和快速查找算法。系统一般使用空闲空间链表管理磁盘空闲块。所以选项B正确。

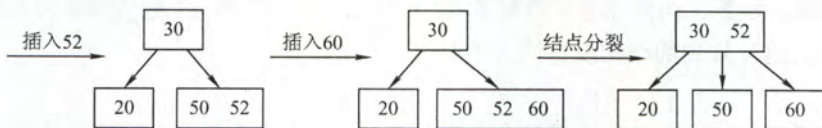
二、综合应用题

1. 解答:

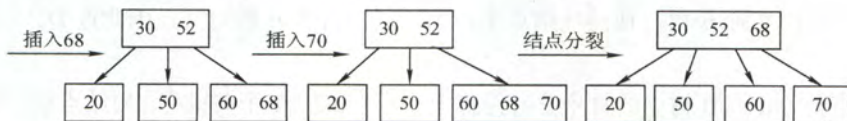
$m=3$, 则除根结点外, 非叶子结点关键字个数为 $1 \sim 2$ 。



首先插入 20, 30, 结点内关键字个数不超过 $\lceil m/2 \rceil = 2$, 不会引起分裂; 插入 50, 插入到 20、30 所在的结点内, 引起分裂, 结点内第 $\lceil m/2 \rceil$ 个关键字 30 上升为父结点。



插入 52, 插入到 50 所在的结点内, 不会引起分裂; 继续插入 60, 插入到 50、52 所在的结点内, 引起分裂, 52 上升到父结点中, 不会引起父结点的分裂。



插入 68, 插入到 60 所在的结点内, 不会引起分裂; 继续插入 70, 插入到 60、68 所在的结点内, 引起分裂, 68 上升为新的父结点, 68 上升到 30、52 所在的结点后, 会继续引起该结点的分裂, 故 52 上升为新的根结点。最后得到的 B 树如第 1 题图 4 所示。

2. 解答:

1) 插入 90: 将 90 插入到 100 所在的结点中, 由于插入 90 后这个结点中的元素个数不超过 $\lceil 3/2 \rceil = 2$, 不会引起结点的分裂, 插入后的 B 树如第 2 题图 1 所示。



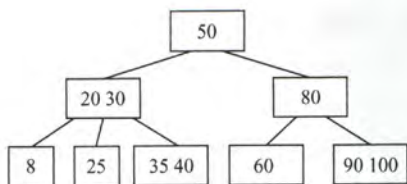
第 1 题图 4

第 2 题图 1

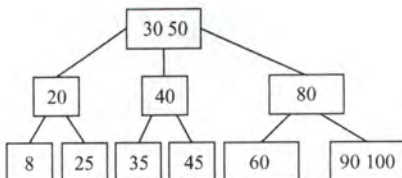
2) 插入 25: 将 25 插入到 8、20 所在的结点中, 插入后结点内的元素个数为 3, 引起分裂。故将结点内的中间元素 20 上升到父结点中, 此时父结点中元素个数为 2 (元素 20 和 30), 不会引起继续分裂, 插入 25 后的 B 树如第 2 题图 2 所示。

3) 插入 45: 将 45 插入到 35、40 所在的结点中, 引起分裂, 中间元素 40 上升到父结点 (20, 30 所在的结点) 中, 引起父结点分裂, 中间元素 30 上升到父结点 (50 所在的结点) 中, 两次分裂后的 B 树如第 2 题图 3 所示。

4) 删除 60: 删除 60 后, 其所在的结点元素为空, 从而导致借用右兄弟结点的元素, 调整后的 B 树如第 2 题图 4 所示。

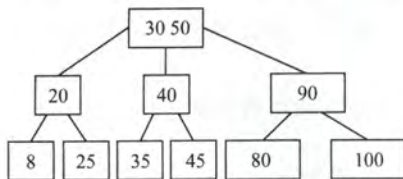


第2题图2

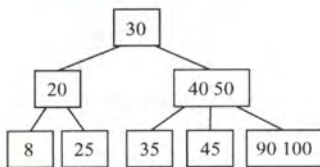


第2题图3

5) 删除 80: 删除 80 后, 导致 80 所在结点的父结点与其右兄弟结点合并, 这时父结点元素个数为 0, 再次对父结点进行调整。将 50 与 40 合并成一个新的结点, 则 90、100 所在结点为该结点的子结点。从而构造的 B 树如下图。注意, 这次调整的过程实际上包含多次调整过程, 希望读者对照考点讲解中的删除过程进行仔细思考。经过以上操作后得到的 B 树如第 2 题图 5 所示。



第2题图4



第2题图5

注意: B 树中结点的插入、删除操作 (特别是插入、删除后的结点分裂与合并) 是本节的重点, 也是难点, 请读者务必熟练掌握。

3. 解答:

根据 B 树的概念, 一个索引结点应适应操作系统一次读写的物理记录大小, 其大小应取不超过但最接近一个磁盘页块的大小。假设 B 树为 m 阶, 一个 B 树结点最多存放 $m-1$ 个关键字 (5 个字节) 和对应的记录地址 (5 个字节)、 m 个子树指针 (5 个字节) 和 1 个指示结点中实际关键字个数的整数 (2 个字节)。则有

$$(2 \times (m-1) + m) \times 5 + 2 \leq 4000$$

计算结果, $m \leq 267$ 。

一个索引结点最多可以存放 $m-1=266$ 个索引项, 最少可以存放 $\lceil m/2 \rceil - 1 = 133$ 个索引项。全部有 $n=20\,000\,000$ 个记录, 每个记录占用空间 200 个字节, 每个页块可以存放 $4\,000/200=20$ 个记录, 则全部记录分布在 $20\,000\,000/20=1\,000\,000$ 个页块中, 则最多需要占用 $1\,000\,000/133=7\,519$ 个磁盘页块作为 B 树索引, 最少需要占用 $1\,000\,000/266=3\,760$ 个磁盘页块作为 B 树索引。(注意 B 树与 B+树的不同, B 树所有对数据记录的索引项分布在各个层次的结点中, B+树所有对数据记录的索引项都在叶结点中)

6.4 散列 (Hash) 表

6.4.1 散列表的基本概念

在前面介绍的线性表和树表的查找中, 记录在表中的位置跟记录的关键字之间不存在确定关系, 因此, 在这些表中查找记录时需进行一系列的关键字比较。这一类查找方法是建立在“比较”的基础上, 查找的效率取决于比较的次数。

散列函数: 一个把查找表中的关键字映射成该关键字对应的地址的函数, 记为

$\text{Hash}(\text{key}) = \text{Addr}$ 。(这里的地址可以是数组下标、索引、或内存地址等)

散列函数可能会把两个或两个以上的不同关键字映射到同一地址,称这种情况为“冲突”,这些发生碰撞的不同关键字称为同义词。一方面,设计好的散列函数应尽量减少这样的冲突;另一方面,由于这样的冲突总是不可避免的,所以还要设计好处理冲突的方法。

散列表:是根据关键字而直接进行访问的数据结构。也就是说,散列表建立了关键字和存储地址之间的一种直接映射关系。

理想情况下,对散列表进行查找的时间复杂度为 $O(1)$,即与表中元素个数无关。下面将分别介绍常用的散列函数和处理冲突的方法。

6.4.2 散列函数的构造方法

在构造散列函数时,必须注意以下几点:

1) 散列函数的定义域必须包含全部需要存储的关键字,而值域的范围则依赖于散列表的大小或地址范围。

2) 散列函数计算出来的地址应该能等概率、均匀地分布在整个地址空间,从而减少冲突的发生。

3) 散列函数应尽量简单,能够在较短的时间内就计算出任一关键字对应的散列地址。

下面介绍常用的散列函数:

1. 直接定址法

直接取关键字的某个线性函数值为散列地址,散列函数为

$$H(\text{key}) = a \times \text{key} + b$$

式中, a 和 b 是常数。这种方法计算最简单,并且不会产生冲突。它适合关键字的分布基本连续的情况,若关键字分布不连续,空位较多,将造成存储空间的浪费。

2. 除留余数法

这是一种最简单、最常用的方法,假定散列表表长为 m ,取一个不大于 m 但最接近或等于 m 的质数 p ,利用以下公式把关键字转换成散列地址。散列函数为

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

除留余数法的关键是选好 p ,使得每一个关键字通过该函数转换后等概率地映射到散列空间上的任一地址,从而尽可能减少冲突的可能性。

3. 数字分析法

设关键字是 r 进制数(如十进制数),而 r 个数码在各位上出现的频率不一定相同,可能在某些位上分布均匀些,每种数码出现的机会均等;而在某些位上分布不均匀,只有某几种数码经常出现,则应选取数码分布较为均匀的若干位作为散列地址。这种方法适合于已知的关键字集合,如果更换了关键字,就需要重新构造新的散列函数。

4. 平方取中法

顾名思义,取关键字的平方值的中间几位作为散列地址。具体取多少位要看实际情况而定。这种方法得到的散列地址与关键字的每一位都有关系,使得散列地址分布比较均匀。适用于关键字的每一位取值都不够均匀或均小于散列地址所需的位数。

5. 折叠法

将关键字分割成位数相同的几部分(最后一部分的位数可以短一些),然后取这几部分的叠

加和作为散列地址,这种方法称为折叠法。关键字位数很多,而且关键字中每一位上数字分布大致均匀时,可以采用折叠法得到散列地址。

在不同的情况下,不同的散列函数会发挥出不同的性能,因此不能笼统地说哪种散列函数最好。在实际的选择中,采用何种构造散列函数的方法取决于关键字集合的情况,但是目标是为了使产生冲突的可能性尽量地降低。

6.4.3 处理冲突的方法

应该注意到,任何设计出来的散列函数都不可能绝对地避免冲突,为此,必须考虑在发生冲突时应该如何进行处理,即为产生冲突的关键字寻找下一个“空”的 Hash 地址。

假设已经选定散列函数 $H(\text{key})$, 下面用 H_i 表示发生冲突后第 i 次探测的散列地址。

1. 开放定址法

所谓开放定址法,指的是可存放新表项的空闲地址既向它的同义词表项开放,又向它的非同义词表项开放。其数学递推公式为

$$H_i = (H(\text{key}) + d_i) \% m$$

式中, $i=0, 1, 2, \dots, k$ ($k \leq m-1$); m 表示散列表表长; d_i 为增量序列。

当取定某一增量序列后,则对应的处理方法是确定的。通常有以下四种取法:

1) 线性探测法: 当 $d_i=0, 1, 2, \dots, m-1$, 称为线性探测法。这种方法的特点是: 冲突发生时, 顺序查看表中下一个单元(当探测到表尾地址 $m-1$ 时, 下一个探测地址是表首地址 0), 直到找出一个空闲单元(当表未填满时一定能找到一个空闲单元)或查遍全表。

线性探测法可能使第 i 个散列地址的同义词存入第 $i+1$ 个散列地址, 这样本应存入第 $i+1$ 个散列地址的元素就争夺第 $i+2$ 个散列地址的元素的地址……, 从而造成大量元素在相邻的散列地址上“聚集”(或堆积)起来, 大大降低了查找效率。

2) 平方探测法: 当 $d_i=0^2, 1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2$, 其中 $k \leq m/2$, 散列表长度 m 必须是一个可以表示成 $4k+3$ 的素数, 又称二次探测法。

平方探测法是一种较好的处理冲突的方法, 可以避免出现“堆积”问题, 它的缺点是不能探测到散列表上的所有单元, 但至少能探测到一半单元。

3) 再散列法: 当 $d_i=\text{Hash}_2(\text{Key})$, 又称为双散列法。需要使用两个散列函数, 当通过第一个散列函数 $H(\text{Key})$ 得到的地址发生冲突时, 则利用第二个散列函数 $\text{Hash}_2(\text{Key})$ 计算该关键字的地址增量。它的具体散列函数形式如下:

$$H_i = (H(\text{Key}) + i * \text{Hash}_2(\text{Key})) \% m$$

初始探测位置 $H_0 = H(\text{Key}) \% m$ 。 i 是冲突的次数, 初始为 0。在散列法中, 最多经过 $m-1$ 次探测会遍历表中所有位置, 回到 H_0 位置。

4) 伪随机序列法: 当 d_i 为伪随机数序列, 称为伪随机序列法。

注意: 在开放定址的情形下, 不能随便物理删除表中已有元素, 因为若删除元素将会截断其他具有相同散列地址的元素的查找地址。所以若想删除一个元素时, 给它做一个删除标记, 进行逻辑删除。但这样做的副作用是: 在执行多次删除后, 表面上看起来散列表很满, 实际上有许多位置没有利用, 因此需要定期维护散列表, 要把删除标记的元素物理删除。

2. 拉链法(链接法, chaining)

显然, 对于不同的关键字可能会通过散列函数映射到同一地址, 为了避免非同义词发生冲突, 可以把所有的同义词存储在一个线性链表中, 这个线性链表由其散列地址唯一标识。假设

散列地址为 i 的同义词链表的头指针存放在散列表的第 i 个单元中, 因而查找、插入和删除操作主要在同义词链中进行。拉链法适用于经常进行插入和删除的情况。

例如, 关键字序列为 {19, 14, 23, 01, 68, 20, 84, 27, 55, 11, 10, 79}, 散列函数 $H(\text{key}) = \text{key} \% 13$, 用拉链法处理冲突, 建立的表如图 6-12 所示。

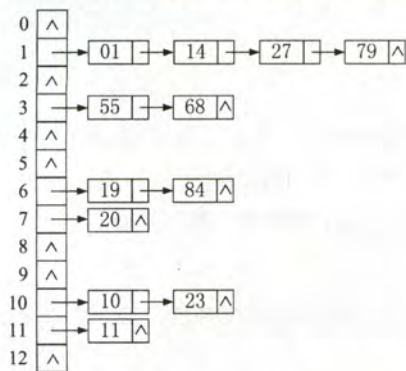


图 6-12 拉链法处理冲突的散列表

6.4.4 散列查找及性能分析

散列表的查找过程与构造散列表的过程基本一致。

对于一个给定的关键字 key , 根据散列函数可以计算出其散列地址, 执行步骤如下:

初始化: $\text{Addr} = \text{Hash}(\text{key})$;

① 检测查找表中地址为 Addr 的位置上是否有记录, 若没有记录, 返回查找失败; 若有记录, 比较它与 key 值, 若相等, 返回查找成功标志, 否则执行步骤②。

② 用给定的处理冲突方法计算“下一个散列地址”, 并把 Addr 置为此地址, 转入步骤①。

例如, 关键字序列 {19, 14, 23, 01, 68, 20, 84, 27, 55, 11, 10, 79} 按散列函数 $H(\text{key}) = \text{key} \% 13$ 和线性探测处理冲突构造所得的散列表 L 如图 6-13 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	01	68	27	55	19	20	84	79	23	11	10			

图 6-13 用线性探测法得到的散列表 L

给定值 84 的查找过程为: 首先求得散列地址 $H(84) = 6$, 因 $L[6]$ 不空且 $L[6] \neq 84$, 则找第一次冲突处理后的地址 $H_1 = (6+1) \% 16 = 7$, 而 $L[7]$ 不空且 $L[7] \neq 84$, 则找第二次冲突处理后的地址 $H_2 = (6+2) \% 16 = 8$, $L[8]$ 不空且 $L[8] = 84$, 查找成功, 返回记录在表中的序号 8。

给定值 38 的查找过程为: 先求散列地址 $H(38) = 12$, $L[12]$ 不空且 $L[12] \neq 38$, 则找下一地址 $H_1 = (12+1) \% 16 = 13$, 由于 $L[13]$ 是空记录, 故表中不存在关键字为 38 的记录。

散列表的查找效率取决于三个因素: 散列函数、处理冲突的方法和装填因子。

装填因子: 散列表的装填因子一般记为 α , 定义为一个表的装满程度, 即:

$$\alpha = \frac{\text{表中记录数 } n}{\text{散列表长度 } m}$$

散列表的平均查找长度依赖于散列表的装填因子 α , 而不直接依赖于 n 或 m 。直观地看, α 越大, 表示装填的记录越“满”, 发生冲突的可能性就越大, 反之发生冲突的可能性越小。

虽然散列表在关键字与记录的存储位置之间建立了直接映像, 但由于“冲突”的产生, 使得散列表的查找过程仍然是一个给定值和关键字进行比较的过程。因此, 仍需以平均查找长度作为衡量散列表的查找效率的度量。读者应能在给出了散列表的长度、元素个数, 以及散列函数和解决冲突方法后, 可以在求出散列表的基础上计算查找成功时的平均查找长度和查找不成功的平均查找长度。

6.4.5 本节试题精选

一、单项选择题

1. 只能在顺序存储结构上进行的查找方法是 ()。

- A. 顺序查找法 B. 折半查找法 C. 树型查找法 D. 散列查找法
2. 散列查找一般适用于()情况下的查找。
- A. 查找表为链表
B. 查找表为有序表
C. 关键字集合比地址集合大得多
D. 关键字集合与地址集合之间存在对应关系
3. 下列关于散列表的说法中, 正确是有()。
- I. 若散列表的填装因子 $\alpha < 1$, 则可避免碰撞的产生
II. 散列查找中不需要任何关键字的比较
III. 散列表在查找成功时平均查找长度与表长有关
IV. 若在散列表中删除一个元素, 不能简单地将该元素删除
- A. I 和 IV B. II 和 III C. II 和 III D. IV
4. 在开址法中散列到同一个地址而引起的“堆积”问题是由于()引起的。
- A. 同义词之间发生冲突
B. 非同义词之间发生冲突
C. 同义词之间或非同义词之间发生冲突
D. 散列表“溢出”
5. 下列关于散列冲突处理方法的说法中, 正确的有()。
- I. 采用再散列法处理冲突时不易产生聚集
II. 采用线性探测法处理冲突时, 所有同义词在散列表中一定相邻
III. 采用链地址法处理冲突时, 若限定在链首插入, 则插入任一个元素的时间是相同的
IV. 采用链地址法处理冲突易引起聚集现象
- A. I 和 III B. I、II 和 III C. III 和 IV D. I 和 IV
6. 设有一个含有 200 个表项的散列表, 用线性探测法解决冲突, 按关键字查询时找到一个表项的平均探测次数不超过 1.5, 则散列表项应能够容纳()个表项。(设查找成功的平均查找长度为 $ASL = [1 + 1/(1 - \alpha)]/2$, 其中 α 为填装因子)
- A. 400 B. 526 C. 624 D. 676
7. 假定有 K 个关键字互为同义词, 若用线性探测法把这 K 个关键字填入散列表中, 至少要进行()次探测。
- A. K-1 B. K C. K+1 D. $K(K+1)/2$
8. 对包含 n 个元素的散列表进行查找, 平均查找长度()。
- A. 为 $O(\log_2 n)$ B. 为 $O(1)$
C. 不直接依赖于 n D. 直接依赖于表长 m
9. 采用开放定址法解决冲突的散列查找中, 发生聚集的原因主要是()。
- A. 数据元素过多 B. 负载因子过大
C. 散列函数选择不当 D. 解决冲突的方法选择不当
10. 【2011 年计算机联考真题】
- 为提高散列 (Hash) 表的查找效率, 可以采取的正确措施是()。
- I. 增大装填 (载) 因子
II. 设计冲突 (碰撞) 少的散列函数
III. 处理冲突 (碰撞) 时避免产生聚集 (堆积) 现象

- A. 仅 I B. 仅 II C. 仅 I、II D. 仅 II、III

11. 【2014 年计算机联考真题】

用哈希（散列）方法处理冲突（碰撞）时可能出现堆积（聚集）现象，下列选项中，会受堆积现象直接影响的是（ ）。

- A. 存储效率 B. 散列函数
C. 装填(装载)因子 D. 平均查找长度

12. 一组记录的关键字为{19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79}，用链地址法构造散列表，散列函数为 $H(\text{key}) = \text{key} \bmod 13$ ，散列地址为 1 的链中有（ ）个记录。

- A. 1 B. 2 C. 3 D. 4

13. 在采用链地址法处理冲突所构成的散列表上查找某一关键字，则在查找成功的情况下，所探测的这些位置上的键值（ ）；若采用线性探测法，则（ ）。

- A. 一定都是同义词 B. 不一定都是同义词
C. 都相同 D. 一定都不是同义词

14. 若采用链地址法构造散列表，散列函数为 $H(\text{key}) = \text{key} \bmod 17$ ，则需（①）个链表。这些链的链首指针构成一个指针数组，数组的下标范围为（②）。

- ① A. 17 B. 13 C. 16 D. 任意
② A. 0 至 17 B. 1 至 17 C. 0 至 16 D. 1 至 16

15. 设散列表长 $m=14$ ，散列函数为 $H(\text{key}) = \text{key} \% 11$ ，表中仅有 4 个结点 $H(15)=4$ ， $H(38)=5$ ， $H(61)=6$ ， $H(84)=7$ ，若采用线性探测法处理冲突，则关键字为 49 的结点地址是（ ）。

- A. 8 B. 3 C. 5 D. 9

16. 将 10 个元素散列到 100 000 个单元的散列表中，则（ ）产生冲突。

- A. 一定会 B. 一定不会 C. 仍可能会 D. 不确定

二、综合应用题

1. 若要在散列表中删除一个记录，应如何操作？为什么？

2. 假定把关键字 key 散列到有 n 个表项(从 0 到 $n-1$ 编址)的散列表中。对于下面的每一个函数 $H(\text{key})$ (key 为整数)，这些函数能够当做散列函数吗？如果能够，它是一个好的散列函数吗？请说明理由。设函数 $\text{random}(n)$ 返回一个 0 到 $n-1$ 之间的随机整数（包括 0 与 $n-1$ 在内）。

- 1) $H(\text{key}) = \text{key}/n$ 。
2) $H(\text{key}) = 1$ 。
3) $H(\text{key}) = (\text{key} + \text{random}(n)) \% n$ 。
4) $H(\text{key}) = \text{key} \% p(n)$ ；其中 $p(n)$ 是不大于 n 的最大素数。

3. 使用散列函数 $H(\text{key}) = \text{key} \% 11$ ，把一个整数值转换成散列表下标，现要把数据{1, 13, 12, 34, 38, 33, 27, 22}依次插入到散列表中。

- 1) 使用线性探测法来构造散列表。
2) 使用链地址法构造散列表。

试针对这两种情况，分别确定查找成功所需的平均查找长度，以及查找不成功所需的平均查找长度。

4. 已知一组关键字为 {26, 36, 41, 38, 44, 15, 68, 12, 6, 51, 25}，用链地址法解决冲突，假设装填因子 $\alpha=0.75$ ，Hash 函数的形式为 $H(\text{key}) = \text{key} \% P$ ，回答以下问题：

- 1) 构造出 Hash 函数。
2) 分别计算出等概率情况下查找成功和查找失败的平均查找长度。（查找失败的计算中只将

与关键字的比较次数计算在内即可)

5. 【2010年计算机联考真题】

将关键字序列 (7、8、30、11、18、9、14) 散列存储到散列表中。散列表的存储空间是一个下标从 0 开始的一维数组, 散列函数为: $H(\text{key})=(\text{key} \times 3) \bmod 7$, 处理冲突采用线性探测再散列法, 要求装填 (载) 因子为 0.7。

(1) 请画出所构造的散列表。

(2) 分别计算等概率情况下, 查找成功和查找不成功的平均查找长度。

6. 设散列表为 HT[0..12], 即表的大小为 $m=13$ 。现采用双散列法解决冲突, 散列函数和再散列函数分别为:

$H_0(\text{key})=\text{key} \% 13$ 注: % 是求余数运算 ($=\bmod$)

$H_i=(H_{i-1}+\text{REV}(\text{key}+1)\%11+1) \% 13; \quad i=1, 2, 3, \dots, m-1$

其中, 函数 $\text{REV}(x)$ 表示颠倒 10 进制数 x 的各位, 如 $\text{REV}(37)=73$, $\text{REV}(7)=7$ 等。若插入的关键码序列为 (2,8,31,20,19,18,53,27)。请回答:

1) 画出插入这 8 个关键码后的散列表。

2) 计算查找成功的平均查找长度 ASL。

6.4.6 答案与解析

一、单项选择题

1. B

顺序查找可以是顺序存储或者链式存储; 折半查找只能是顺序存储且要求关键字有序; 树形查找法要求采用树的存储结构, 既可以采用顺序存储也可以采用链式存储; 散列查找中的链地址法解决冲突采用的是顺序存储与链式存储相结合的方式。

2. D

关键字集合与地址集合之间存在对应关系时, 通过散列函数表示这种关系, 这样, 查找以计算散列函数而不是比较的方式进行查找。

3. D

冲突 (碰撞) 是不可避免的, 与装填因子无关, 因此需要设计处理冲突的方法, I 错误。散列查找的思想是计算出散列地址来进行查找, 然后再比较关键字以确定是否查找成功, II 错误。散列查找成功的平均查找长度与装填因子有关, 与表长无关, III 错误。在开放定址的情形下, 不能随便删除散列表中某个元素, 否则可能会导致搜索路径被中断 (因此通常的做法是在要删除的地方做删除标记, 而不是直接删除), IV 错误。

4. C

在开址法中散列到同一个地址而产生的“堆积”问题, 是由于为解决同义词冲突的探查序列和非同义词之间不同的探查序列交织在一起, 导致关键字查询需要经过较长的探测距离, 降低了散列的效率。所以要选择好的处理冲突的方法来避免“堆积”。

5. A

利用再散列法处理冲突时, 按一定的距离, 跳跃式地寻找“下一个”空闲位置, 减少了发生聚集的可能, I 正确。散列地址 i 的关键字, 和为解决冲突形成的某次探测地址为 i 的关键字, 都争夺地址 $i, i+1, \dots$, 因此不一定相邻, II 错误。III 正确。同义词冲突不等于聚集, 链地址法处理冲突时将同义词放在同一个链表中, 不会引起聚集现象, IV 错误。

6. A

如果有 200 个表项要放入散列表, 采用线性探测法解决冲突, 限定查找成功的平均查找长度不超过 1.5, 则

$$ASL_{成功} = \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right) \leq 1.5 \Rightarrow \alpha = \frac{200}{m} \leq \frac{1}{2} \Rightarrow m \geq 400$$

7. D

由于 K 个关键字在依次填入的过程中, 只有第一个不会发生冲突, 探测的次数为 $(1+2+3+\cdots+K)=K(K+1)/2$, 即选 D。

8. C

在散列表中, 平均查找长度与装填因子 α 直接相关, 表的查找效率不直接依赖于表中已有表项个数 n 或表长 m 。若散列表中存放的记录全部是某个地址的同义词, 则平均查找长度为 $O(n)$, 而不是 $O(1)$ 。

9. D

聚集是因选取不当的处理冲突的方法, 而导致不同关键字的元素对同一散列地址进行争夺的现象。当采用线性再探测法时, 容易引发聚集现象。

10. D

散列表的查找效率取决于: 散列函数、处理冲突的方法和装填因子。显然, 冲突的产生概率与装填因子 (即表中记录数与表长之比) 的大小成正比, I 与题意相反。II 显然正确。采用合适的冲突处理方法可避免聚集现象, 也将提高查找效率, III 正确。如, 用拉链法处理冲突时不存在聚集现象, 用线性探测法处理冲突时易引起聚集现象。

11. D

产生堆积现象, 即产生了冲突, 它对存储效率、散列函数和装填因子均不会有影响, 而平均查找长度会因为堆积现象而增大, 选 D。

12. D

由散列函数计算可知 14, 1, 27, 79 散列后的地址都是 1, 所以有 4 个记录。

13. A, B

因为在链地址法中, 映射到同一地址的关键字都会链到与此地址相对应的链表上, 所以探测过程一定是在此链表上进行的, 从而这些位置上的关键字均为同义词; 但在线性探测法中, 出现两个同义关键字时, 会把该关键字对应地址的下一个地址也占用掉, 两个地址分别记为 $Addr$, $Addr+1$, 当查找一个满足 $H(key)=Addr+1$ 的关键字 key 时, 显然首次探测到的不是 key 的同义词。

14. A, C

H 的取值有 17 种可能, 对应到不同的链表中, 所以链表的个数应该是 17 个。由于 $H(key)$ 取值范围是 $0 \sim 16$, 所以数组下标为 $0 \sim 16$ 。

15. A

线性探测法的公式为 $H_i = (H(k) + d_i) \% m$, 其中 $d_i = 1, 2, 3, \dots, m-1$ 。 $H(49) = 49 \% 11 = 5$, 发生冲突; $H_1 = (H(49) + 1) \% 14 = 6$, 冲突; $H_2 = (H(49) + 2) \% 14 = 7$, 冲突; $H_3 = (H(49) + 3) \% 14 = 8$, 没有冲突。选 A。

16. C

由于散列函数的选取, 仍然有可能产生地址冲突, 冲突不能绝对地避免。

二、综合应用题

1. 解答:

在散列表中删除一个记录,在拉链法情况下可以物理地删除。但在开放定址法情况下,不能物理地删除,只能作删除标记。该地址可能是该记录的同义词查找路径上的地址,物理的删除就中断了查找路径。因为查找时碰到空地址就认为是查找失败。

2. 解答:

1) 不能当做散列函数,因为 key/n 可能大于 n , 这样就无法找到适合的位置。

2) 能够作为散列函数,但不是一个好的散列函数,因为所有关键字都映射到同一位置,造成大量的冲突机会。

3) 不能当做散列函数,因为该函数的返回值不确定,这样无法进行正常的查找。

4) 能够作为散列函数,是一个好的散列函数。

3. 分析:

由散列函数可知散列地址的范围为 $0 \sim 10$ 。

采用线性探测法构造散列表时,首先应计算出关键字对应的散列地址,然后检查散列表中对应的地址是否已经有元素。如果没有元素,则直接将该关键字放入散列表对应的地址中;如果有元素,则采用线性探测的方法查找下一个地址,从而决定该关键字的存放位置。

采用链地址法构造散列表时,在直接计算出关键字对应的散列地址后,将关键字结点插入到此散列地址所在链表中。

解答:

1) 线性探测法:

$H(1)=1$, 没有冲突,地址 1 存放关键字 1。 $H(13)=2$, 没有冲突,地址 2 存放关键字 13。 $H(12)=1$, 发生冲突,根据线性探测法: $H_1=2$, 发生冲突,继续探测 $H_2=3$, 没有冲突,于是 12 存放在地址为 3 的表项中。 $H(34)=1$, 发生冲突,根据线性探测法: $H_1=2$, 发生冲突, $H_2=3$, 发生冲突, $H_3=4$, 没有冲突,于是 34 存放在地址为 4 的表项中。

同理可以计算其他的数据存放情况,最后结果如下表所示。

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键字	33	1	13	12	34	38	27	22			
冲突次数	0	0	0	2	3	0	1	7			

下面计算平均查找长度:

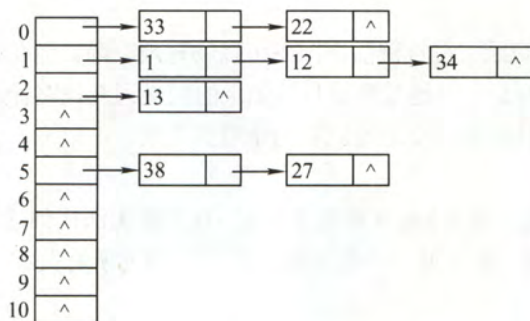
查找成功时,显然查找每个元素的概率都是 $1/8$,对于 33,由于冲突次数为 0,所以仅需 1 次比较便可查找成功;对于 22,由于计算出的地址为 0,但需要 8 次比较才能查找成功,所以 22 的查找长度为 8,其他元素的分析类似,因此:

$$ASL_{\text{成功}} = (1+1+1+3+4+1+2+8)/8 = 21/8$$

查找失败时,由于 $H(\text{key})=0 \sim 10$,故对每个位置查找的概率都是 $1/11$,对于计算出的地址为 0 的关键字 key_0 ,只有探测完 $0 \sim 8$ 号地址后才能确定该元素不在表中,比较次数为 9;对于计算出的地址为 1 的关键字 key_1 ,只有探测完 $1 \sim 8$ 号地址后才能确定该元素不在表中,比较次数为 8,以此类推。而对于计算出的地址为 8、9、10 的关键字,这些单元中没有存放元素,所以只需比较 1 次便可确定查找失败,因此:

$$ASL_{\text{失败}} = (9+8+7+6+5+4+3+2+1+1+1)/11 = 47/11$$

2) 链地址法构造的表如下:



在链地址表中查找成功时, 查找关键字为 33 的记录需进行 1 次比较, 查找关键字为 22 的记录需进行 2 次比较, 以此类推……, 因此:

$$ASL_{成功} = (1 \times 4 + 2 \times 3 + 3) / 8 = 13/8$$

查找失败时, 对于地址 0, 比较 3 次后确定元素不在表中 (空指针算 1 次), 所以其查找长度为 3; 对于地址 1, 其查找长度为 4; 地址 2 查找长度为 2, 以此类推……, 因此:

$$ASL_{失败} = (3 + 4 + 2 + 1 + 1 + 3 + 1 + 1 + 1 + 1 + 1) / 11 = 19/11$$

值得注意的是, 求查找失败的平均查找长度有两种观点: 其一, 认为比较到空结点才算失败, 所以比较次数等于冲突次数加 1; 其二, 认为只有与关键字的比较才算比较次数。

4. 分析:

由装填因子的计算公式 $\alpha = n/N$ (n 为关键字个数, N 为表长), 不难得出表长, 而根据 Hash 函数的选择要求, P 应该取不大于表长的最大素数, 从而可以确定 P 的大小, 也就构造出了 Hash 函数。这里采用的是链地址法解决冲突, 其两种情况下的平均查找长度的计算过程与上一题完全相似。

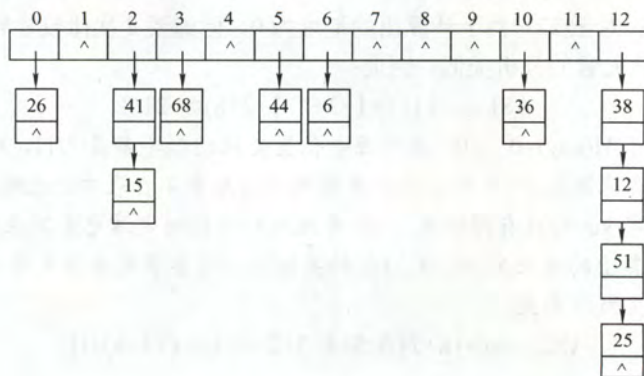
解答:

1) 由 $\alpha = n/N$ 得 $N = n/\alpha$, 由于 N 为整数, 故应该向上取整, 即 $N = \lceil n/\alpha \rceil = 15$, 从而 $P = 13$ 。因此散列函数为 $H(key) = key \% 13$ 。

2) 由 1) 求出的 Hash 函数, 计算各关键字对应的 Hash 地址如下表:

关键字	26	36	41	38	44	15	68	12	6	51	25
Hash 地址	0	10	2	12	5	2	3	12	6	12	12

由此构造的链地址法处理冲突的 Hash 表为



由上图不难计算

$$ASL_{成功} = (1*7 + 2*2 + 3*1 + 4*1) / 11 = 18/11$$

$$ASL_{失败} = (1+0+2+1+0+1+1+0+0+0+1+0+4) / 13 = 11/13$$

5. 解答:

1) 由装载因子 0.7, 数据总数为 7, 得一维数组大小为 $7/0.7=10$, 数组下标为 0~9。所构造的散列函数值如下所示:

key	7	8	30	11	18	9	14
H(key)	0	3	6	5	5	6	0

采用线性探测再散列法处理冲突, 所构造的散列表为:

地址	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	

2) 查找成功时, 在等概率情况下, 查找每个表中元素的概率是相等的, 因此, 是根据表中元素个数来计算平均查找长度, 各关键字的比较次数为:

key	7	8	30	11	18	9	14
次数	1	1	1	1	3	3	2

故, $ASL_{成功} = \text{查找次数} / \text{元素个数} = (1+2+1+1+1+3+3) / 7 = 12/7$

在计算查找失败时的平均查找长度时, 要特别注意防止思维定势, 在查找失败的情况下既不是根据表中元素个数, 也不是根据表长来计算平均查找长度的。

查找失败时, 在等概率情况下, 经过散列函数计算后只可能映射到表中的 0~6 位置, 且映射到 0~6 中任一位置的概率是相等的。因此, 是根据散列函数 (MOD 后面的数字) 来计算平均查找长度。在等概率情况下, 查找失败的比较次数为:

H(key)	0	1	2	3	4	5	6
次数	3	2	1	2	1	5	4

故, $ASL_{不成功} = \text{查找次数} / \text{散列后的地址个数} = (3+2+1+2+1+5+4) / 7 = 18/7$

6. 解答:

1) $H_0(2)=2$ 、 $H_0(8)=8$ 、 $H_0(31)=5$ 、 $H_0(20)=7$ 、 $H_0(19)=6$, 没有冲突。 $H_0(18)=5$, 发生冲突, $H_1(18)=(H_0(18)+REV(18+1)\%11+1)\%13=(5+3+1)\%13=9$, 没有冲突。 $H_0(53)=1$, 没有冲突。 $H_0(27)=1$, 发生冲突, $H_1(27)=(H_0(27)+REV(27+1)\%11+1)\%13=(1+5+1)\%13=7$, 发生冲突, $H_2(27)=(H_1(27)+REV(27+1)\%11+1)\%13=0$, 没有冲突。构造的散列表如下:

散列地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键字	27	53	2			31	19	20	8	18			
比较次数	3	1	1			1	1	1	1	2			

2) 由 1) 中散列表的构造过程, 各个关键字查找成功的比较次数如上表, 故:

$$ASL_{成功} = (3+1+1+1+1+1+1+2) / 8 = 11/8$$

6.5 字符串模式匹配

6.5.1 简单的模式匹配算法

串的模式匹配，是求第一个字符串（模式串）在第二个字符串（主串）中的位置。

下面给出一种简单的字符串模式匹配算法：从主串 S 指定的字符开始（一般为第一个）和模式串 T 的第一个字符比较，若相等，则继续逐个比较后续字符，直到 T 中的每个字符依次和 S 中的一个连续的字符序列相等，则称匹配成功；如果比较过程中有某对字符不相等，则从主串 S 的下一个字符起再重新和 T 的第一个字符比较。如果 S 中的字符都比完了仍然没有匹配成功，则称匹配不成功。代码如下：

```
int Index(SString S, SString T){
    int i=1, j=1;
    while(i<=S[0]&&j<=T[0]){
        if(S[i]==T[j])
            {++i; ++j;} //继续比较后继字符
        else
            {i=i-j+2; j=1;} //指针后退重新开始匹配
    }
    if(j>T[0]) return i-T[0];
    else return 0;
}
```

图 6-12 展示了模式 $T='abcac'$ 和主串 S 的匹配过程。



图 6-12 简单模式匹配算法的举例

简单模式匹配算法的最坏时间复杂度为 $O(n*m)$ ， n 、 m 分别为主串和模式串的长度。

6.5.2 改进的模式匹配算法——KMP 算法

KMP 算法可以在 $O(n+m)$ 的时间数量级上完成串的模式匹配操作。其改进在于：每当一趟匹配过程中出现字符比较不等时，不需回溯 i 指针，而是利用已经得到的“部分匹配”的结果将模式向右“滑动”尽可能远的一段距离后，继续进行比较。

回顾图 6-12 的匹配过程，在第三趟的匹配中，当 $i=7$ 、 $j=5$ 字符比较不等时，又从 $i=4$ 、 $j=1$ 重新开始比较。

然而，经仔细观察可发现，在 $i=4$ 和 $j=1$ ， $i=5$ 和 $j=1$ 以及 $i=6$ 和 $j=1$ 这 3 次比较都是不必进行的。因为从第三趟部分匹配的结果就可得出，主串中第 4、5 和 6 个字符必然是 'b'、'c' 和 'a'（即模式串中第 2、3、和 4 个字符）。因为模式中第一个字符是 a，因此它无需再和这 3 个字符进行比较，而仅需将模式向右滑动 3 个字符的位置继续进行 $i=7$ 、 $j=2$ 时的字符比较即可。同理，在第一趟匹配中出现字符不等时，仅需将模式向右移动两个字符的位置继续进行 $i=3$ 、 $j=1$ 时的字符比较。由此，在整个匹配的过程中， i 指针没有回溯，如图 6-14 所示。

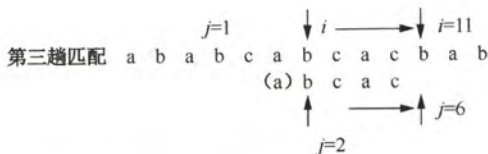


图 6-14 改进算法的匹配过程示例

KMP 算法的每趟比较过程让子串向后滑动一个合适的位置，让这个位置上的字符和主串中的那个字符比较，这个合适的位置与子串本身的结构有关。

下面来看一种更一般的情况。假设原始串为 S ，长度为 n ，模式串为 T ，长度为 m 。目前匹配到如下下划线的位置：

$$S_0, S_1, S_2, \dots, S_{i-j}, S_{i-j+1}, \dots, S_{i-1}, \underline{S_i}, S_{i+1}, \dots, S_{n-1}$$

$$T_0, T_1, \dots, T_{j-1}, \underline{T_j}, \dots, T_{m-1}$$

' $S_{i-j}S_{i-j+1}\dots S_{i-1}$ '和' $T_0T_1\dots T_{j-1}$ '的部分匹配成功，恰好到 S_i 和 T_j 的时候匹配失败，如果保持 i 不变，同时达到让模式串 T 相对原始串 S 右移的话，我们可以想办法更新 j 的值，找到一个最大的 k ，满足 ' $S_{i-k}S_{i-k+1}\dots S_{i-1} = T_0T_1\dots T_{k-1}$ '，使新的 $j=k$ ，然后让 S_i 和 T_j 进行匹配，假设新的 j 用 $\text{next}[j]$ 表示，即 $\text{next}[j]$ 表示当模式串匹配到 $T[j]$ 遇到失配时，在模式串中需要重新和主串匹配的位置。换言之， next 数组的求解实际是对每个位置找到最长的公共前缀。所以 next 数组的定义为：

$$\text{next}[j] = \begin{cases} 0 & \text{当 } j=1 \text{ 时} \\ \text{Max}\{k \mid 1 < k < j \text{ 且 } 'p_1\dots p_{k-1}' = 'p_{j-k+1}\dots p_{j-1}'\} & \text{当此集合不空时} \\ 1 & \text{其他情况} \end{cases}$$

求 next 函数值的算法如下：

```
void get_next(char T[], int next[]) {
    i = 1;
    next[1] = 0;
    j = 0;
    while (i <= T[0]) { // T[0] 用于保存字符串的长度
        if (j == 0 || T[i] == T[j]) {
            ++i; ++j; next[i] = j;
        }
        else {
            j = next[j];
        }
    } // while
}
```

下面介绍一个手工求解 next 数组的方法：

- (1) $\text{next}[1]=0$, $\text{next}[2]=1$ 。(next[0]不使用)
 (2) 后面求解每一位的 $\text{next}[j]$ 值时, 根据 j 的前一位进行比较, 令 $k=\text{next}[j-1]$;
 (3) 将 $S[j-1]$ 与 $S[k]$ 进行比较:
 a. 如果相等, 则该 $\text{next}[j]=k+1$;
 b. 如果不等, 令 $k=\text{next}[k]$, 若 k 不等于 0, 跳到(3); 若 k 等于 0, $\text{next}[j]=1$ 。

例如模式串 $S='abaabcac'$, 依照以上算法求 next 数组:

- (1) 设 $\text{next}[1]=0$, $\text{next}[2]=1$ 。

编号	1	2	3	4	5	6	7	8
S	a	b	a	a	b	c	a	c
next	0	1						

- (2) 当 $j=3$, 此时 $k=\text{next}[j-1]=\text{next}[2]=1$, 观察 $S[2]$ 与 $S[k](S[1])$ 是否相等, $S[2]=b$, $S[1]=a$, $S[2] \neq S[1]$, 此时 $k=\text{next}[k]=0$, 所以 $\text{next}[j]=1$ 。

↓ $j-1=2$

```

a   b   a   a   b   c   a   c
   a   b   a   a   b   c   a   c

```

↑ $k=1$

- (3) 当 $j=4$, 此时 $k=\text{next}[3]=1$, 观察 $S[3]$ 与 $S[k](S[1])$, $S[3]=a$, $S[1]=a$, $S[3]=S[1]$, 所以 $\text{next}[j]=k+1=1+1=2$ 。

↓ $j-1=3$

```

a   b   a   a   b   c   a   c
   a   b   a   a   b   c   a   c

```

↑ $k=1$

- (4) 当 $j=5$, 此时 $k=\text{next}[4]=2$, 观察 $S[4]$ 与 $S[k](S[2])$, $S[4]=a$, $S[2]=b$, $S[4] \neq S[2]$ 。

↓ $j-1=4$

```

a   b   a   a   b   c   a   c
   a   b   a   a   b   c   a   c

```

↑ $k=2$

此时 $k=\text{next}[k]=\text{next}[2]=1$, 继续比较 $S[4]$ 与 $S[k](S[1])$, $S[1]=a$, $S[4]=S[1]$, 所以 $\text{next}[j]=k+1=2$ 。

↓ $j-1=4$

```

a   b   a   a   b   c   a   c
   a   b   a   a   b   c   a   c

```

↑ $k=1$

- (5) 当 $j=6$, 此时 $k=\text{next}[5]=2$, 观察 $S[5]$ 与 $S[k](S[2])$, $S[5]=b$, $S[2]=b$, $S[5]=S[2]$, 所以 $\text{next}[j]=k+1=3$ 。

↓ $j-1=5$

```

a   b   a   a   b   c   a   c
   a   b   a   a   b   c   a   c

```

↑ $k=2$

- (6) 当 $j=7$, 此时 $k=\text{next}[6]=3$, 观察 $S[6]$ 与 $S[k](S[3])$, $S[6]=c$, $S[3]=a$, $S[6] \neq S[3]$ 。

↓ $j-1=6$

```

a   b   a   a   b   c   a   c
   a   b   a   a   b   c   a   c

```

↑ $k=3$

此时 $k=\text{next}[k]=1$, 继续比较 $S[6]$ 与 $S[k](S[1])$, $S[1]=a$, 所以 $S[6] \neq S[1]$ 。

↓ $j-1=6$

```

a   b   a   a   b   c   a   c
   a   b   a   a   b   c   a   c

```

↑ $k=1$

此时 $k=\text{next}[k]=0$, 所以 $\text{next}[j]=1$ 。

- (7) 当 $j=8$, 此时 $k=\text{next}[7]=1$, 观察 $S[7]$ 与 $S[k](S[1])$, $S[7]=a$, $S[1]=a$, $S[7]=S[1]$, 所以 $\text{next}[j]=k+1=2$ 。

$\downarrow j-1=7$
 a b a a b c a c
 a b a a b c a c
 $\uparrow k=1$

最后结果如下:

编号	1	2	3	4	5	6	7	8
S	a	b	a	a	b	c	a	c
next	0	1	1	2	2	3	1	2

与 next 数组的求解相比, KMP 的匹配算法就相对简单很多, 它在形式上与简单的模式匹配算法很相似。不同之处仅在于当匹配过程产生失配时, 指针 i 不变, 指针 j 退回到 $\text{next}[j]$ 的位置并重新进行比较, 并且当指针 j 为 0 时, 指针 i 和 j 同时加 1。即若主串的第 i 个位置和模式串的第一个字符不等, 应从主串的第 $i+1$ 个位置开始匹配。具体代码如下:

```
int KMP(char S[], char T[], int next[], int pos){
//利用模式串 T 的 next 函数求 T 在主串 S 中第 pos 个字符之后的位置的 KMP 算法。
//其中, T 非空,  $1 \leq \text{pos} \leq \text{strlen}(S)$ 
    i=pos;
    j=1;
    while(i<=S[0]&&j<=T[0]){
        if(j==0||S[i]==T[j]){
            ++i;
            ++j;
        }//if
        else
            j=next[j];
    }//while
    if(j > T[0])
        return i - T[0];
    else
        return 0;
}
```

下面用一个实例来说明 KMP 算法的匹配过程。

假设主串 $S='abcabaaabaabcac'$, 子串 $T='abaabcac'$, 其中子串的 next 数组在上面已经求出。匹配过程如下:

- (1) 初始化, $i=1, j=1$

$\downarrow i=1$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=1$

- (2) $S[1]=T[1]$, 匹配, i 和 j 自加 1

$\downarrow i=2$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=2$

- (3) $S[2]=T[2]$, 匹配, i 和 j 自加 1

- $\downarrow i=3$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=3$
 (4) $S[3] \neq T[3]$, 不匹配, $j=\text{next}[j]=\text{next}[3]=1$
 $\downarrow i=3$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=1$
 (5) $S[3] \neq T[1]$, 不匹配, 且 $j=\text{next}[j]=0$
 $\downarrow i=3$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=0$
 (6) $j=0$, i 和 j 自加 1
 $\downarrow i=4$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=1$
 (7) $S[4]=T[1]$, i 和 j 自加 1
 $\downarrow i=5$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=2$
 (8) $S[5]=T[2]$, i 和 j 自加 1
 $\downarrow i=6$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=3$
 (9) $S[6]=T[3]$, i 和 j 自加 1
 $\downarrow i=7$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=4$
 (10) $S[7]=T[4]$, i 和 j 自加 1
 $\downarrow i=8$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=5$
 (11) $S[8] \neq T[5]$, 不匹配, $j=\text{next}[j]=2$
 $\downarrow i=8$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=2$
 (12) $S[8] \neq T[2]$, 不匹配, $j=\text{next}[j]=1$
 $\downarrow i=8$
 a b c a b a a a b a a b c a c
 a b a a b c a c
 $\uparrow j=1$
 (13) 以下全部匹配, 过程略

↓ i=15

a	b	c	a	b	a	a	a	b	a	a	b	c	a	c
							a	b	a	a	b	c	a	c
							a	b	a	a	b	c	a	c
														↑ j=8

以上就是手工模拟 KMP 算法的过程。

尽管朴素的模式匹配的时间复杂度是 $O(m*n)$ ，KMP 算法的时间复杂度是 $O(m+n)$ 。但在一般情况下，朴素的模式匹配算法的实际执行时间近似 $O(m+n)$ ，因此至今仍然被采用。KMP 算法仅仅是在主串与子串有很多“部分匹配”时才显得比朴素的算法快得多，其主要优点是主串不回溯。

KMP 算法对于初学者来说有些不容易理解，读者可以尝试多读几遍本章内容，并参考一些其他教材的相关内容来巩固这个知识点。

6.5.3 本节试题精选

一、单项选择题

1. 设有两个串 S1 和 S2，求 S2 在 S1 中首次出现的位置的运算称作 ()。
A. 求子串 B. 判断是否相等 C. 模式匹配 D. 连接
2. KMP 算法的特点是在模式匹配时指示主串的指针 ()。
A. 不会变大 B. 不会变小 C. 都有可能 D. 无法判断
3. 设主串的长度为 n，子串的长度为 m，那么简单的模式匹配算法的时间复杂度为 ()，KMP 算法的时间复杂度为 ()。
A. $O(m)$ B. $O(n)$ C. $O(m*n)$ D. $O(m+n)$
4. 已知串 S='aaab'，其 next 数组值为 ()。
A. 0123 B. 0112 C. 0231 D. 1211
5. 串 'ababaaababaa' 的 next 数组值为 ()。
A. 01234567899 B. 012121111212 C. 011234223456 D. 0123012322345
6. 串 "ababaaababaa" 的 next 数组为 ()。
A. -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 8, 8
B. -1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1
C. -1, 0, 0, 1, 2, 3, 1, 1, 2, 3, 4, 5
D. -1, 0, 1, 2, -1, 0, 1, 2, 1, 1, 2, 3

7. 【2015 年计算机联考真题】

已知字符串 S 为 "abaabaabacacaabaabcc"，模式串 t 为 "abaabc"。采用 KMP 算法进行匹配，第一次出现“失配”($s[i] \neq t[j]$) 时， $i=j=5$ ，则下次开始匹配时，i 和 j 的值分别是 。

- A. $i=1, j=0$ B. $i=5, j=0$ C. $i=5, j=2$ D. $i=6, j=2$

二、综合应用题

1. 在字符串模式匹配的 KMP 算法中，求模式的 next 数组值的定义如下：

$$\text{next}[j] = \begin{cases} 0, & \text{当 } j=1 \text{ 时} \\ \text{Max}\{k \mid 1 < k < j \text{ 且 } 'p_1 \cdots p_{k-1}' = 'p_{j-k+1} \cdots p_{j-1}'\}, & \text{当此集合不空时} \\ 1, & \text{其他情况} \end{cases}$$

- 1) 当 $j=1$ 时，为什么要取 $\text{next}[1]=0$?
- 2) 为什么要取 $\text{max}\{K\}$ ，K 最大是多少?

3) 其他情况是什么情况, 为什么取 $\text{next}[j]=1$?

2. 设字符串 $S = \text{'aabaabaabaac'}$, $P = \text{'aabaac'}$

1) 给出 S 和 P 的 next 数组。

2) 若 S 作主串, P 作模式串, 试给出利用 BF 算法和 KMP 算法的匹配过程。

6.5.4 答案与解析

一、单项选择题

1. C

求子串操作是从字符串 s 中截取第 i 个字符开始后的长度 l 的子串, A 不选。 B 、 D 明显不对, 答案选 C 。

2. B

在 KMP 算法中, 模式匹配时主串不会回溯, 所以主串的指针不会变小。选 B 。

3. C、D

尽管实际应用中的一般情况下, 简单的模式匹配算法时间复杂度近似为 $O(m+n)$, 但是它的实际复杂度还是 $O(m*n)$, 选 C 。KMP 算法的时间复杂度为 $O(m+n)$, 选 D 。

4. A

(1) 设 $\text{next}[1]=0$, $\text{next}[2]=1$ 。

编号	1	2	3	4
S	a	a	a	b
next	0	1		

(2) 当 $j=3$, 此时 $k=\text{next}[j-1]=\text{next}[2]=1$, 观察 $S[2]$ 与 $S[k](S[1])$ 是否相等, $S[2]=a$, $S[1]=a$, $S[2]=S[1]$, 所以 $\text{next}[j]=k+1=2$ 。

$\downarrow j-1=2$
 a a a b
 a a a b
 $\uparrow k=1$

(3) 当 $j=4$, 此时 $k=\text{next}[j-1]=\text{next}[3]=2$, 观察 $S[3]$ 与 $S[k](S[2])$ 是否相等, $S[3]=a$, $S[2]=a$, $S[3]=S[2]$, 所以 $\text{next}[j]=k+1=3$ 。

$\downarrow j-1=3$
 a a a b
 a a a b
 $\uparrow k=2$

最后结果为:

编号	1	2	3	4
S	a	a	a	b
next	0	1	2	3

选 A 。

5. C

(1) 设 $\text{next}[1]=0$, $\text{next}[2]=1$ 。

编号	1	2	3	4	5	6	7	8	9	10	11	12
S	a	b	a	b	a	a	a	b	a	b	a	a
next	0	1										

(2) 当 $j=3$, 此时 $k=\text{next}[j-1]=\text{next}[2]=1$, 观察 $S[2]$ 与 $S[k](S[1])$ 是否相等, $S[2]=b$, $S[1]=a$, $S[2] \neq S[1]$, 此时 $k=\text{next}[k]=0$, 所以 $\text{next}[j]=1$ 。

↓ $j-1=2$

```
a  b  a  b  a  a  a  b  a  b  a  a
   a  b  a  b  a  a  a  b  a  b  a  a
```

↑ $k=1$

(3) 当 $j=4$, 此时 $k=\text{next}[j-1]=\text{next}[3]=1$, 观察 $S[3]$ 与 $S[k](S[1])$ 是否相等, $S[3]=a$, $S[1]=a$, $S[3]=S[1]$, 所以 $\text{next}[j]=k+1=2$ 。

↓ $j-1=3$

```
a  b  a  b  a  a  a  b  a  b  a  a
   a  b  a  b  a  a  a  b  a  b  a  a
```

↑ $k=1$

(4) 当 $j=5$, 此时 $k=\text{next}[j-1]=\text{next}[4]=2$, 观察 $S[4]$ 与 $S[k](S[2])$ 是否相等, $S[4]=b$, $S[2]=b$, $S[4]=S[2]$, 所以 $\text{next}[j]=k+1=3$ 。

↓ $j-1=4$

```
a  b  a  b  a  a  a  b  a  b  a  a
   a  b  a  b  a  a  a  b  a  b  a  a
```

↑ $k=2$

(5) 当 $j=6$, 此时 $k=\text{next}[j-1]=\text{next}[5]=3$, 观察 $S[5]$ 与 $S[k](S[3])$ 是否相等, $S[5]=a$, $S[3]=a$, $S[5]=S[3]$, 所以 $\text{next}[j]=k+1=4$ 。

↓ $j-1=5$

```
a  b  a  b  a  a  a  b  a  b  a  a
   a  b  a  b  a  a  a  b  a  b  a  a
```

↑ $k=3$

此时可知 next 数组的前 6 项为 011234, 只有 C 选项符合, 另外请读者自己手动算完整个 next 数组的值, 选 C。

读者作完该题后是否会想到: 如果考试中遇到这类题型, 难道要按照这么严格的步骤才能得出最后的结果? 这样“机械”的做法在考试中是极不推荐的! 首先, 用时较长; 其次, 万一在考试的紧张氛围中不小心把某个下标之类的写错, 该题基本可以宣布 **gameover** 了。如此费力不讨好的事劝读者少做为好, 但原理必须清楚。推荐读者用下面的方法:

首先, 按步骤来 (考试中不必这么详细)

编号	1	2	3	4	5	6	7	8	9	10	11	12
S	a	b	a	b	a	a	a	b	a	b	a	a
next	0	1										

其次, 在字母的下面标数字, 填写上表 (考试中也不需要如此详细)

该串长度为 1 时, 缺省为 $\text{next}[1]=0$;

该串长度为 2 时, 缺省为 $\text{next}[2]=1$;

注意: 求解 $\text{next}[j]$ 时应该根据 j 的前面的字符串进行比较得出:

该串长度为 3 时, 其前面的串为 “ab”, 因为 $\text{next}[2]=1$, 则在此基础上根据 “ $\text{next}[2]=1$ ” 后面中的 1, 观察 “ab” 中的前 “1” 个字符和后 “1” 个字符是否相同, 因为 a 和 b 不同, 所以 $\text{next}[3]=1$ (至于为什么是 1, 是由于在 $\text{next}[2]=1$ 之后的 $\text{next}[j](j>2)$ 的最小值都为 1, 记住即可。对此欲深入研究的读者请查阅相关的资料。限于篇幅, 此文不再赘述);

该串长度为 4 时, 其前面的串为 “aba”, 则在此基础上根据 “ $\text{next}[3]=1$ ” 后面中的 1, 观察 “aba” 中的前 “1” 个字符和后 “1” 个字符是否相同, 因为 a 和 a 相同, 所以 $\text{next}[4]$ 的值在此基础上应加 1, 即得 $\text{next}[4]=1+1=2$;

该串长度为 5 时,其前面的串为“abab”,则在此基础上根据“next[4]=2”后面中的 2,观察“abab”中的前“2”个字符和后“2”个字符是否相同,因为 ab 和 ab 相同,所以 next[5]的值在此基础上应加 1,即得 $\text{next}[5]=2+1=3$;

该串长度为 6 时,其前面的串为“ababa”,则在此基础上根据“next[5]=3”后面中的 3,观察“ababa”中的前“3”个字符和后“3”个字符是否相同,因为 aba 和 aba 相同,所以 next[6]的值在此基础上应加 1,即得 $\text{next}[6]=4$;

该串长度为 7 时,其前面的串为“ababaa”,则在此基础上根据“next[6]=4”后面中的 4,观察“ababaa”中的前“4”个字符和后“4”个字符是否相同,因为 abab 和 abaa 不同,所以 next[7]的值在此基础上应减 1(采取匹配“退而求其次”的办法),即 $\text{next}[7]=4-1=3$;则由于“ababaa”中的前“3”个字符 aba 和后“3”个字符 baa 不同,所以继续减 1: $3-1=2$,观察“ababaa”的前“2”个字符 ab 和后“2”个字符 aa 不同,所以继续减 1: $2-1=1$,观察“ababaa”的前“1”个字符 a 和后“1”个字符 a 相同,所以应在此基础上加 1,得 $\text{next}[7]=2$;

该串长度为 8 时,其前面的串为“ababaaa”,则在此基础上根据“next[7]=2”后面中的 2,观察“ababaaa”中的前“2”个字符和后“2”个字符是否相同,因为 ab 和 aa 不同,所以 next[8]的值在此基础上应减 1: $2-1=1$,观察“ababaaa”的前“1”个字符 a 和后“1”个字符 a,因为相同,所以在此基础上加 1: $1+1=2$,得 $\text{next}[8]=2$;

该串长度为 9 时,其前面的串为“ababaaab”,则在此基础上根据“next[8]=2”后面中的 2,观察“ababaaab”中的前“2”个字符和后“2”个字符是否相同,因为 ab 和 ab 相同,所以 next[9]的值在此基础上应加 1: $2+1=3$,即得 $\text{next}[9]=3$;

该串长度为 10 时,其前面的串为“ababaaaba”,则在此基础上根据“next[9]=3”后面中的 3,观察“ababaaaba”中的前“3”个字符和后“3”个字符是否相同,因为 aba 和 aba 相同,所以 next[10]的值在此基础上应加 1: $3+1=4$,即得 $\text{next}[10]=4$;

该串长度为 11 时,其前面的串为“ababaaabab”,则在此基础上根据“next[10]=4”后面中的 4,观察“ababaaabab”中的前“4”个字符和后“4”个字符是否相同,因为 abab 和 abab 相同,所以 next[11]的值在此基础上应加 1: $4+1=5$,即得 $\text{next}[11]=5$;

该串长度为 12 时,其前面的串为“ababaaababa”,则在此基础上根据“next[11]=5”后面中的 5,观察“ababaaababa”中的前“5”个字符和后“5”个字符是否相同,因为 ababa 和 ababa 相同,所以 next[12]的值在此基础上应加 1: $5+1=6$,即得 $\text{next}[12]=6$;

总结:设当前待求的为 $\text{next}[n]$ ($n>2$), $\text{next}[n-1]=m$;

- 1) 若“字符串*****”(共 $n-1$ 个字符)的前“ m ”个字符和后“ m ”个字符相同则 $\text{next}[n]=m+1$,转到 3);
- 2) 若不同:若 $m=1$,则 $\text{next}[n]=1$ 转到 3);否则,令 $m=m-1$,观察该字符串的前“ $m-1$ ”字符和后“ $m-1$ ”个字符,若相同则转到 1),否则继续 2);
- 3) 结束本次求解 $\text{next}[n]$ 的操作;

以上便是快速求解 next 数组的方法,望读者认真领会字里行间的精髓,掌握此方法后相信你会在较短的时间里解决“字符串的 next 数组”的很多难题。考试中比较常见的是:next 数组的下标是从 -1 开始的,如此仅需稍微地对下标修改即可,如下题(请沿用此方法快速求解,下面仅列出了大致步骤)。

6. C

做出模式串以及对应字符下标(注意下标是从 -1 开始的),如下表所示:

a	b	a	b	a	a	a	b	a	b	a	a
0	1	2	3	4	5	6	7	8	9	10	11

S 串长度为 0 时, $\text{next}[0] = -1$;
 S 串长度为 1 时, $\text{next}[1] = 0$;
 S 串长度为 2 时, S 串为 “ab”, $\text{next}[2] = 0$;
 S 串长度为 3 时, S 串为 “aba”, $\text{next}[3] = 1$;
 S 串长度为 4 时, S 串为 “abab”, $\text{next}[4] = 2$;
 S 串长度为 5 时, S 串为 “ababa”, $\text{next}[5] = 3$;
 S 串长度为 6 时, S 串为 “ababaa”, $\text{next}[6] = 1$;
 S 串长度为 7 时, S 串为 “ababaaa”, $\text{next}[7] = 1$;
 S 串长度为 8 时, S 串为 “ababaaab”, $\text{next}[8] = 2$;
 S 串长度为 9 时, S 串为 “ababaaaba”, $\text{next}[9] = 3$;
 S 串长度为 10 时, S 串为 “ababaaabab”, $\text{next}[10] = 4$;
 S 串长度为 11 时, S 串为 “ababaaababa”, $\text{next}[11] = 5$;
 7. C

由题中“失配 $s[i] \neq t[j]$ 时, $i=j=5$ ”, 可知题中的主串和模式串的位序都是从 0 开始的 (要注意灵活应变)。按照 next 数组生成算法, 对于 t 有:

编号	0	1	2	3	4	5
t	a	b	a	a	b	c
next	-1	0	0	1	1	2

依据 KMP 算法“当失配时, i 不变, j 回退到 $\text{next}[j]$ 的位置并重新比较”, 当失配 $s[i] \neq t[j]$ 时, $i=j=5$, 由上表不难得出 $\text{next}[j]=\text{next}[5]=2$ (位序从 0 开始)。从而最后结果应为: $i=5$ (i 保持不变), $j=2$ 。

二、综合应用题

1. 解答:

(1) 当模式串中第一个字符与主串中某字符比较不等 (失配) 时, $\text{next}[1]=0$ 表示模式串中已没有字符可与主串中当前字符 $s[i]$ 比较, 主串当前指针应后移至下一字符, 再和模式串中第一字符进行比较。

(2) 当主串第 i 个字符与模式串中第 j 个字符失配时, 若主串 i 不回溯, 则假定模式串第 k 个字符与主串第 i 个字符比较, k 值应满足条件 $1 < k < j$ 并且 ‘ $p_1 \cdots p_{k-1}$ ’ = ‘ $p_{j-k+1} \cdots p_{j-1}$ ’, 即 k 为模式串向后移动的距离, k 值有多个, 为了不使向右移动丢失可能的匹配, k 要取最大, 由于 $\max\{k\}$ 表示移动的最大距离, 所以取 $\max\{k\}$, k 的最大值为 $j-1$ 。

(3) 在上面两种情况外, 发生失配时, 主串指针 i 不回溯, 在最坏情况下, 模式串从第 1 个字符开始与主串第 i 个字符比较, 以便不致丢失可能的匹配。

2. 解答:

1) $S = \text{'aabaabaabaac'}$, $P = \text{'aabaac'}$, 按照 next 数组生成算法, 对于 S 有:

(1) 设 $\text{next}[1]=0$, $\text{next}[2]=1$ 。

编号	1	2	3	4	5	6	7	8	9	10	11	12
S	a	a	b	a	a	b	a	a	b	a	a	c
next	0	1										

(2) 当 $j=3$, 此时 $k=\text{next}[j-1]=\text{next}[2]=1$, 观察 $S[2]$ 与 $S[k](S[1])$ 是否相等, $S[2]=a$, $S[1]=a$, $S[2]=S[1]$, 所以 $\text{next}[j]=k+1=2$ 。

↓ j-1=2

```

a  a  b  a  a  b  a  a  b  a  a  c
a  a  b  a  a  b  a  a  b  a  a  c
↑ k=1

```

(3) 当 $j=4$, 此时 $k=\text{next}[j-1]=\text{next}[3]=2$, 观察 $S[3]$ 与 $S[k](S[2])$ 是否相等, $S[3]=b$, $S[2]=a$, $S[3] \neq S[2]$.

↓ j-1=3

```

a  a  b  a  a  b  a  a  b  a  a  c
a  a  b  a  a  b  a  a  b  a  a  c
↑ k=2

```

此时 $k=\text{next}[k]=1$, 观察 $S[3]$ 与 $S[k](S[1])$, $S[3]=b$, $S[1]=a$, $S[3] \neq S[1]$. $k=\text{next}[k]=0$, 因为 $k=0$, 所以 $\text{next}[j]=1$.

↓ j-1=3

```

a  a  b  a  a  b  a  a  b  a  a  c
a  a  b  a  a  b  a  a  b  a  a  c
↑ k=0

```

(4) 当 $j=5$, 此时 $k=\text{next}[j-1]=\text{next}[4]=1$, 观察 $S[4]$ 与 $S[k](S[1])$ 是否相等, $S[4]=a$, $S[1]=a$, $S[4]=S[1]$, 所以 $\text{next}[j]=k+1=2$.

↓ j-1=4

```

a  a  b  a  a  b  a  a  b  a  a  c
a  a  b  a  a  b  a  a  b  a  a  c
↑ k=1

```

(5) 当 $j=6$, 此时 $k=\text{next}[j-1]=\text{next}[5]=2$, 观察 $S[5]$ 与 $S[k](S[2])$ 是否相等, $S[5]=a$, $S[2]=a$, $S[5]=S[2]$, 所以 $\text{next}[j]=k+1=3$.

↓ j-1=5

```

a  a  b  a  a  b  a  a  b  a  a  c
a  a  b  a  a  b  a  a  b  a  a  c
↑ k=2

```

(6) 当 $j=7$, 此时 $k=\text{next}[j-1]=\text{next}[6]=3$, 观察 $S[6]$ 与 $S[k](S[3])$ 是否相等, $S[6]=b$, $S[3]=b$, $S[6]=S[3]$, 所以 $\text{next}[j]=k+1=4$.

↓ j-1=6

```

a  a  b  a  a  b  a  a  b  a  a  c
a  a  b  a  a  b  a  a  b  a  a  c
↑ k=3

```

(7) 当 $j=8$, 此时 $k=\text{next}[j-1]=\text{next}[7]=4$, 观察 $S[7]$ 与 $S[k](S[4])$ 是否相等, $S[7]=a$, $S[4]=a$, $S[7]=S[4]$, 所以 $\text{next}[j]=k+1=5$.

↓ j-1=7

```

a  a  b  a  a  b  a  a  b  a  a  c
a  a  b  a  a  b  a  a  b  a  a  c
↑ k=4

```

(8) 当 $j=9$, 此时 $k=\text{next}[j-1]=\text{next}[8]=5$, 观察 $S[8]$ 与 $S[k](S[5])$ 是否相等, $S[8]=a$, $S[5]=a$, $S[8]=S[5]$, 所以 $\text{next}[j]=k+1=6$.

↓ j-1=8

```

a  a  b  a  a  b  a  a  b  a  a  c
a  a  b  a  a  b  a  a  b  a  a  c
↑ k=5

```

(9) 当 $j=10$, 此时 $k=\text{next}[j-1]=\text{next}[9]=6$, 观察 $S[9]$ 与 $S[k](S[6])$ 是否相等, $S[9]=b$, $S[6]=b$, $S[9]=S[6]$, 所以 $\text{next}[j]=k+1=7$ 。

$\downarrow j-1=9$
 a a b a a b a a b a a c
 a a b a a b a a b a a c
 $\uparrow k=6$

(10) 当 $j=11$, 此时 $k=\text{next}[j-1]=\text{next}[10]=7$, 观察 $S[10]$ 与 $S[k](S[7])$ 是否相等, $S[10]=a$, $S[7]=a$, $S[10]=S[7]$, 所以 $\text{next}[j]=k+1=8$ 。

$\downarrow j-1=10$
 a a b a a b a a b a a c
 a a b a a b a a b a a c
 $\uparrow k=7$

(11) 当 $j=12$, 此时 $k=\text{next}[j-1]=\text{next}[11]=8$, 观察 $S[11]$ 与 $S[k](S[8])$ 是否相等, $S[11]=a$, $S[8]=a$, $S[11]=S[8]$, 所以 $\text{next}[j]=k+1=9$ 。

$\downarrow j-1=11$
 a a b a a b a a b a a c
 a a b a a b a a b a a c
 $\uparrow k=8$

最后结果为:

编号	1	2	3	4	5	6	7	8	9	10	11	12
S	a	a	b	a	a	b	a	a	b	a	a	c
next	0	1	2	1	2	3	4	5	6	7	8	9

对于 P 有:

(1) 设 $\text{next}[1]=0$, $\text{next}[2]=1$ 。

编号	1	2	3	4	5	6
S	a	a	b	a	a	c
next	0	1				

(2) 当 $j=3$, 此时 $k=\text{next}[j-1]=\text{next}[2]=1$, 观察 $S[2]$ 与 $S[k](S[1])$ 是否相等, $S[2]=a$, $S[1]=a$, $S[2]=S[1]$, 所以 $\text{next}[j]=k+1=2$ 。

$\downarrow j-1=2$
 a a b a a c
 a a b a a c
 $\uparrow k=1$

(3) 当 $j=4$, 此时 $k=\text{next}[j-1]=\text{next}[3]=2$, 观察 $S[3]$ 与 $S[k](S[2])$ 是否相等, $S[3]=b$, $S[2]=a$, $S[3] \neq S[2]$ 。

$\downarrow j-1=3$
 a a b a a c
 a a b a a c
 $\uparrow k=2$

此时 $k=\text{next}[k]=1$, 所观察 $S[3]$ 与 $S[k](S[1])$, $S[3]=b$, $S[1]=a$, $S[3] \neq S[1]$ 。 $k=\text{next}[k]=0$, 因为 $k=0$, 所以 $\text{next}[j]=1$ 。

$\downarrow j-1=3$
 a a b a a c
 a a b a a c
 $\uparrow k=0$

(4) 当 $j=5$, 此时 $k=\text{next}[j-1]=\text{next}[4]=1$, 观察 $S[4]$ 与 $S[k](S[1])$ 是否相等, $S[4]=a$, $S[1]=a$, $S[4]=S[1]$, 所以 $\text{next}[j]=k+1=2$ 。

↓ $j-1=4$

a	a	b	a	a	c
			a	a	b
				a	a
					c

↑ $k=1$

(5) 当 $j=6$, 此时 $k=\text{next}[j-1]=\text{next}[5]=2$, 观察 $S[5]$ 与 $S[k](S[2])$ 是否相等, $S[5]=a$, $S[2]=a$, $S[5]=S[2]$, 所以 $\text{next}[j]=k+1=3$ 。

↓ $j-1=5$

a	a	b	a	a	c
			a	a	b
				a	a
					c

↑ $k=2$

最后结果为:

编号	1	2	3	4	5	6
S	a	a	b	a	a	c
next	0	1	2	1	2	3

2)

利用 BF 算法的匹配过程:

第一趟匹配: aabaabaabaac

aabaac($i=6, j=6$)

第二趟匹配: aabaabaabaac

aa($i=3, j=2$)

第三趟匹配: aabaabaabaac

a($i=3, j=1$)

第四趟匹配: aabaabaabaac

aabaac($i=9, j=6$)

第五趟匹配: aabaabaabaac

aa($i=6, j=2$)

第六趟匹配: aabaabaabaac

a($i=6, j=1$)

第七趟匹配: aabaabaabaac

(成功) aabaac ($i=13, j=7$)

利用 KMP 算法的匹配过程:

第一趟匹配: aabaabaabaac

aabaac($i=6, j=6$)

第二趟匹配: aabaabaabaac

aabaac ($i=9, j=6$)

第三趟匹配: aabaabaabaac

(成功) aabaac

归纳总结

本章的核心考查点是求平均查找长度 (ASL), 来度量各种查找算法的性能, 2013 年联考 408 的综合题中就考到了。查找算法本身依托于查找结构, 查找结构又是由相同数据类型的记录或结点构成, 故最终落脚于数据结构类型的区别。

不管何种查找算法, 其平均查找长度的计算公式都是一样的。

查找成功的平均查找长度: $ASL_{\text{成功}} = \sum_{i=1}^n p_i * c_i$

查找失败的平均查找长度: $ASL_{\text{不成功}} = \sum_{j=0}^n q_j * c_j$

设一个查找集合中已有 n 个数据元素，每个元素的查找概率为 p_i ，查找成功的数据比较次数为 c_i ($i=1,2,\dots,n$)；而不在此集合中的数据元素分布在由该 n 个元素的间隔构成的 $n+1$ 个子集合内，每个子集合元素的查找概率为 q_j ，查找不成功的数据比较次数为 c_j ($j=0,1,\dots,n$)。那么对于某一特定查找算法的查找成功的 $ASL_{成功}$ 和查找失败的 $ASL_{不成功}$ 是综合考虑，还是分开考虑呢？

如果综合考虑，即 $\sum_{i=1}^n p_i + \sum_{j=0}^n q_j = 1$ ，若所有元素查找概率相等，则有 $p_i = q_j = \frac{1}{2n+1}$ ；如果分开考虑，即 $\sum_{i=1}^n p_i = 1$ ， $\sum_{j=0}^n q_j = 1$ ，若所有元素查找概率相等，则有 $p_i = \frac{1}{n}$ ， $q_j = \frac{1}{n+1}$ 。

虽然综合考虑更理想，但在实际应用中，多数是分开考虑的，因为对于查找不成功的情况，很多场合下没有明确给出，往往被忽略掉。不过读者仍要注意，这两种考虑的计算结果是不同的，考试中一定要仔细阅读题目要求，以免失误。

思维拓展

本章介绍了几种基本的查找算法，在实际中又会碰到怎样的查找问题呢？

题目：数组中有一个数字出现的次数超过了数组长度的一半，请找出这个数字。读者也许会想到先进行排序，在 $(n+1)/2$ 位置的数即为要找的数，这样最小时间复杂度为 $O(n \log_2 n)$ ；如果进行 Hash，数字的范围又未知。那么如何将时间复杂度控制在 $O(n)$ 内？

（提示：出现的次数超过数组长度的一半，表明这个数字出现的次数比其他数字出现的次数的总和还多。所以我们可以考虑每次删除两个不同的数，那么在剩下的数中，待找数字出现的次数仍然超过总数的一半。通过不断重复这个过程，不断排除掉其他的数字，最终剩下的都为同一个数字，即为要找的数字。）

第7章 排 序

【考纲内容】

(一) 排序的基本概念

(二) 插入排序

直接插入排序；折半插入排序；希尔排序 (shell sort)

(三) 交换排序

起泡排序 (bubble sort)；快速排序

(四) 选择排序

简单选择排序；堆排序

(五) 二路归并排序 (merge sort)

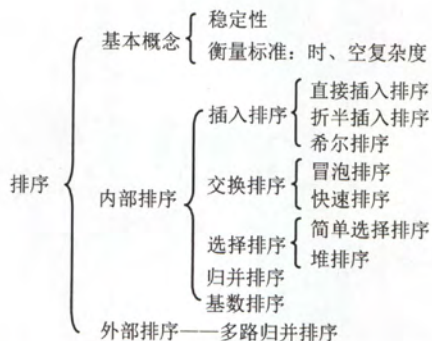
(六) 基数排序

(七) 外部排序

(八) 各种排序算法的比较

(九) 排序算法的应用

【知识框架】



【复习提示】

堆排序 (建堆、插入和调整)、快速排序 (划分、过程特征) 和归并排序 (归并路数、归并过程) 是本章重点。读者应深入掌握各种排序算法的思想、排序过程 (能动手模拟) 和特征 (初始态的影响、时空复杂度、稳定性、适用性等), 通常以选择题的形式考查不同算法之间的对比。此外, 对于一些常用排序算法的关键代码, 要达到熟练编写的程度; 看到某特定序列, 读者应具有选择最优排序算法 (根据排序算法特征) 的能力。

7.1 排序的基本概念

7.1.1 排序的定义

排序：就是重新排列表中的元素，使表中的元素满足按关键字递增或递减的过程。为了查找方便，通常要求计算机中的表是按关键字有序的。排序的确切定义如下：

输入： n 个记录 R_1, R_2, \dots, R_n ，对应的关键字为 k_1, k_2, \dots, k_n 。

输出：输入序列的一个重排 R_1', R_2', \dots, R_n' ，使得有 $k_1' \leq k_2' \leq \dots \leq k_n'$ （其中“ \leq ”可以换成其他的比较大小的符号）。

算法的稳定性：如果待排序表中有两个元素 R_i, R_j ，其对应的关键字 $key_i = key_j$ ，且在排序前 R_i 在 R_j 前面，如果使用某一排序算法排序后， R_i 仍然在 R_j 的前面，则称这个排序算法是稳定的，否则称排序算法是不稳定的。需要注意的是，算法是否具有稳定性并不能衡量一个算法的优劣，它主要是对算法的性质进行描述。

注意：对于不稳定的排序算法，只需举出一组关键字的实例，说明它的不稳定性即可。

在排序的过程中，根据数据元素是否完全在内存中，可将排序算法分为两类：**内部排序**是指在排序期间元素全部存放在内存中的排序；**外部排序**是指在排序期间元素无法全部同时存放在内存中，必须在排序的过程中根据要求不断地在内、外存之间移动的排序。

一般情况下，内部排序算法在执行过程中都要进行两种操作：比较和移动。通过比较两个关键字，确定对应的元素的前后关系，然后通过移动元素以达到有序。当然，并不是所有的内部排序算法都要基于比较操作，事实上，基数排序就不是基于比较的。

内部排序算法的性能取决于算法的时间复杂度和空间复杂度，而时间复杂度一般是由比较和移动的次数来决定的。

7.1.2 本节试题精选

一、单项选择题

1. 下述排序方法中，不属于内部排序方法的是（ ）。
A. 插入排序 B. 选择排序 C. 拓扑排序 D. 冒泡排序
2. 排序算法的稳定性是指（ ）。
A. 经过排序后，能使关键字相同的元素保持原顺序中的相对位置不变
B. 经过排序后，能使关键字相同的元素保持原顺序中的绝对位置不变
C. 排序算法的性能与被排序元素个数关系不大
D. 排序算法的性能与被排序元素的个数关系密切
3. 下列关于排序的叙述中，正确的是（ ）。
A. 稳定的排序方法优于不稳定的排序方法
B. 对同一线性表使用不同的排序方法进行排序，得到的排序结果可能不同
C. 排序方法都是在顺序表上实现的，在链表上无法实现排序方法
D. 在顺序表上实现的排序方法在链表上也可以实现
4. 对任意的 7 个关键字进行基于比较的排序，至少要进行（ ）次关键字之间的两两比较。

A. 13

B. 14

C. 15

D. 6

7.1.3 答案与解析

一、单项选择题

1. C

拓扑排序是将有向图中所有结点排成一个线性序列，虽然也是在内存中进行，但它不属于我们这里所提到的内部排序范畴，也不满足前面排序的定义。

2. A

注意，这里的绝对位置是指如果在排序前元素 R 在位置 i ，则绝对位置就是 i ，即排序后 R 的位置不发生变化，显然 B 是不对的。 C 、 D 与题目要求无关。

3. B

算法的稳定性与算法优劣无关， A 排除。使用链表也可以进行排序，只是有些排序算法不再适用，因为这个时候定位元素只能顺序逐链查找，如折半插入排序。

4. A

对于任意序列进行基于比较的排序，求最少的比较次数，应考虑在最坏的情况下。对任意 n 个关键字排序的比较次数至少为 $\lceil \log_2(n!) \rceil$ 。将 $n=7$ 代入公式，答案为 13。

上述公式证明如下（仅供有兴趣的同学参考）：在基于比较的排序方法中，每次比较两个关键字之后，仅出现两种可能的转移。假设整个排序过程至少需要作 t 次比较，则显然会有 2^t 种情况。由于 n 个记录共有 $n!$ 种不同排列，因而必须有 $n!$ 种不同的比较路径，于是有 $2^t \geq n!$ ，即 $t \geq \log_2(n!)$ 。考虑到 t 为整数，故为 $\lceil \log_2(n!) \rceil$ 。

注意要与 7.5.3 章节的选择题 7 区别。

7.2 插入排序

插入排序是一种简单直观的排序方法，其基本思想在于每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子序列中，直到全部记录插入完成。

由插入排序的思想可以引申出三个重要的排序算法：直接插入排序、折半插入排序和希尔排序。下面将依次进行讲解。

7.2.1 直接插入排序^①

从上面的插入排序思想中，不难得出一种最简单也最直观的直接插入排序算法。假设在排序过程中，待排序表 $L[1 \cdots n]$ 在某次排序过程中的某一时刻状态如下：

有序序列 $L[1 \cdots i-1]$	$L(i)$	无序序列 $L[i+1 \cdots n]$
------------------------	--------	------------------------

为了实现将元素 $L(i)$ 插入到已有序的子序列 $L[1 \cdots i-1]$ 中，我们需要执行以下操作（为避免混淆，下面用“ $L[]$ ”表示一个表，而用“ $L()$ ”表示一个元素）：

1) 查找出 $L(i)$ 在 $L[1 \cdots i-1]$ 中的插入位置 k 。

① 凡在本书中没有加以特殊说明，通常都是指递增有序序列。

2) 将 $L[k \cdots i-1]$ 中所有元素全部后移一个位置。

3) 将 $L(i)$ 复制到 $L(k)$ 。

为了实现对 $L[1 \cdots n]$ 的排序, 可以将 $L(2) \sim L(n)$ 依次插入到前面已排好序的子序列中, 初始假定 $L[1]$ 是一个已排好序的子序列。上述操作执行 $n-1$ 次就能得到一个有序的表。插入排序在实现上通常采用就地排序 (空间复杂度为 $O(1)$), 因而在从后向前的比较过程中, 需要反复把已排序元素逐步向后挪位, 为新元素提供插入空间。

下面是直接插入排序的代码, 再次用到了我们前面提到的“哨兵”(作用相同)。

```
void InsertSort(ElemType A[], int n) {
    int i, j;
    for (i = 2; i <= n; i++)           // 依次将 A[2] ~ A[n] 插入到前面已排序序列
        if (A[i].key < A[i-1].key) { // 若 A[i] 的关键码小于其前驱, 需将 A[i] 插入有序表
            A[0] = A[i];               // 复制为哨兵, A[0] 不存放元素
            for (j = i-1; A[0].key < A[j].key; --j) // 从后往前查找待插入位置
                A[j+1] = A[j];         // 向后挪位
            A[j+1] = A[0];             // 复制到插入位置
        }
}
```

直接插入排序算法的性能分析如下:

空间效率: 仅使用了常数个辅助单元, 因而空间复杂度为 $O(1)$ 。

时间效率: 在排序过程中, 向有序子表中逐个地插入元素的操作进行了 $n-1$ 趟, 每趟操作都分为比较关键字和移动元素, 而比较次数和移动次数取决于待排序表的初始状态。

在最好情况下, 表中元素已经有序, 此时每插入一个元素, 都只需比较一次而不用移动元素, 因而时间复杂度为 $O(n)$ 。

在最坏情况下, 表中元素顺序刚好与排序结果中元素顺序相反 (逆序) 时, 总的比较次数达到最大, 为 $\sum_{i=2}^n i$, 总的移动次数也达到最大, 为 $\sum_{i=2}^n (i+1)$ 。

平均情况下, 考虑待排序表中元素是随机的, 此时可以取上述最好与最坏情况的平均值作为平均情况下的时间复杂度, 总的比较次数与总的移动次数均约为 $n^2/4$ 。

由此, 直接插入排序算法的时间复杂度为 $O(n^2)$ 。虽然折半插入排序算法的时间复杂度也有 $O(n^2)$, 但对于数据量比较小的排序表, 折半插入排序往往能表现出很好的性能。

稳定性: 由于每次插入元素时总是从后向前先比较再移动, 所以不会出现相同元素相对位置发生变化的情况, 即直接插入排序是一个稳定的排序方法。

适用性: 直接插入排序算法适用于顺序存储和链式存储的线性表。当为链式存储时, 可以从前往后查找指定元素的位置。注意: 大部分排序算法都仅适用于顺序存储的线性表。

7.2.2 折半插入排序

从前面的直接插入排序算法中, 不难看出每趟插入的过程中, 都进行了两项工作: ①从前面的子表中查找出待插入元素应该被插入的位置; ②给插入位置腾出空间, 将待插入元素复制到表中的插入位置。注意到该算法中, 总是边比较边移动元素, 下面将比较和移动操作分离出来, 即先折半查找出元素的待插入位置, 然后再统一地移动待插入位置之后的所有元素。当排序表为顺序存储的线性表时, 可以对直接插入排序算法作如下改进:

由于是顺序存储的线性表, 所以查找有序子表时可以用折半查找来实现。在确定出待插入位置后, 就可以统一地向后移动元素了。算法如下:


```

void InsertSort(ElemType A[],int n){
    int i,j,low,high,mid;
    for(i=2;i<=n;i++){           //依次将 A[2]~A[n]插入到前面已排序序列
        A[0]=A[i];               //将 A[i]暂存到 A[0]
        low=1;high=i-1;          //设置折半查找的范围
        while(low<=high){        //折半查找(默认递增有序)
            mid=(low+high)/2;     //取中点
            if(A[mid].key>A[0].key) high=mid-1; //查找左半子表
            else low=mid+1;       //查找右半子表
        }
        for(j=i-1;j>=high+1;--j)
            A[j+1]=A[j];         //统一后移元素,空出插入位置
        A[high+1]=A[0];          //插入操作
    }
}

```

从上述算法中,不难看出折半插入排序仅仅是减少了比较元素的次数,约为 $O(n\log_2 n)$,该比较次数与待排序表的初始状态无关,仅取决于表中的元素个数 n ;而元素的移动次数没有改变,它依赖于待排序表的初始状态。因此,折半插入排序的时间复杂度仍为 $O(n^2)$ 。折半插入排序是一个稳定的排序方法。

7.2.3 希尔排序

从前面的讲解不难看出,直接插入排序算法适用于基本有序的排序表和数据量不大的排序表。基于这两点,1959年 D.L.Shell 提出了希尔排序,又称为缩小增量排序。

希尔排序的基本思想是:先将待排序表分割成若干个形如 $L[i, i+d, i+2d, \dots, i+kd]$ 的“特殊”子表,分别进行直接插入排序,当整个表中元素已呈“基本有序”时,再对全体记录进行一次直接插入排序。希尔排序的排序过程如下:

先取一个小于 n 的步长 d_1 ,把表中全部记录分成 d_1 个组,所有距离为 d_1 的倍数的记录放在同一个组中,在各组中进行直接插入排序;然后取第二个步长 $d_2 < d_1$,重复上述过程,直到所取到的 $d_i=1$,即所有记录已放在同一组中,再进行直接插入排序,由于此时已经具有较好的局部有序性,故可以很快得到最终结果。到目前为止,尚未求得一个最好的增量序列,希尔提出的方法是 $d_1=n/2$, $d_{i+1}=\lfloor d_i/2 \rfloor$,并且最后一个增量等于 1。

希尔排序算法如下:

```

void ShellSort (ElemType A[],int n){
    //对顺序表作希尔插入排序,本算法和直接插入排序相比,作了以下修改:
    //1.前后记录位置的增量是 dk,不是 1
    //2.r[0]只是暂存单元,不是哨兵,当 j<=0 时,插入位置已到
    for(dk=n/2;dk>=1;dk=dk/2)    //步长变化
        for(i=dk+1;i<=n;++i)
            if(A[i].key<A[i-dk].key){ //需将 A[i]插入有序增量子表
                A[0]=A[i];           //暂存在 A[0]
                for(j=i-dk;j>0&&A[0].key<A[j].key;j-=dk)
                    A[j+dk]=A[j];    //记录后移,查找插入的位置
                A[j+dk]=A[0];         //插入
            } //if
    }
}

```

希尔排序算法的性能分析如下:

空间效率：仅使用了常数个辅助单元，因而空间复杂度为 $O(1)$ 。

时间效率：由于希尔排序的时间复杂度依赖于增量序列的函数，这涉及数学上尚未解决的难题，所以其时间复杂度分析比较困难。当 n 在某个特定范围时，希尔排序的时间复杂度约为 $O(n^{1.3})$ 。在最坏情况下希尔排序的时间复杂度为 $O(n^2)$ 。

稳定性：当相同关键字的记录被划分到不同的子表时，可能会改变它们之间的相对次序，因此，希尔排序是一个不稳定的排序方法。例如，表 $L=\{3, 2, 2\}$ ，经过一趟排序后， $L=\{2, 2, 3\}$ ，最终排序序列也是 $L=\{2, 2, 3\}$ ，显然，2 与 2 的相对次序已经发生了变化。

适用性：希尔排序算法仅适用于当线性表为顺序存储的情况。

7.2.4 本节试题精选

一、单项选择题

- 对 5 个不同的数据元素进行直接插入排序，最多需要进行的比较次数是 ()。
A. 8 B. 10 C. 15 D. 25
- 在待排序的元素序列基本有序的前提下，效率最高的排序方法是 ()。
A. 直接插入排序 B. 简单选择排序 C. 快速排序 D. 归并排序
- 【2012 年计算机联考真题】
对同一待排序序列分别进行折半插入排序和直接插入排序，两者之间可能的不同之处是 ()。
A. 排序的总趟数 B. 元素的移动次数
C. 使用辅助空间的数量 D. 元素之间的比较次数
- 对有 n 个元素的顺序表采用直接插入排序算法进行排序，在最坏情况下所需的比较次数是 ()；在最好情况下所需的比较次数是 ()。
A. $n-1$ B. $n+1$ C. $n/2$ D. $n(n-1)/2$
- 数据序列 $\{8, 10, 13, 4, 6, 7, 22, 2, 3\}$ 只能是 () 的两趟排序后的结果。
A. 简单选择排序 B. 起泡排序 C. 直接插入排序 D. 堆排序
- 用直接插入排序算法对下列 4 个表进行 (从小到大) 排序，比较次数最少的是 ()。
A. 94, 32, 40, 90, 80, 46, 21, 69
B. 21, 32, 46, 40, 80, 69, 90, 94
C. 32, 40, 21, 46, 69, 94, 90, 80
D. 90, 69, 80, 46, 21, 32, 94, 40
- 在下列算法中，() 算法可能出现下列情况：在最后一趟开始之前，所有元素都不在最终位置上。
A. 堆排序 B. 冒泡排序 C. 直接插入排序 D. 快速排序
- 希尔排序属于 ()。
A. 插入排序 B. 交换排序 C. 选择排序 D. 归并排序
- 对序列 $\{15, 9, 7, 8, 20, -1, 4\}$ 用希尔排序方法排序，经一趟后序列变为 $\{15, -1, 4, 8, 20, 9, 7\}$ 则该次采用的增量是 ()。
A. 1 B. 4 C. 3 D. 2
- 【2009 年计算机联考真题】
用希尔排序方法对一个数据序列进行排序时，若第 1 趟排序结果为 9,1,4,13,7,8,20,23,15，则该趟排序采用的增量 (间隔) 可能是 ()。

- A. 2 B. 3 C. 4 D. 5

11. 若对于第9题中的序列, 经一趟排序后序列变成{9, 15, 7, 8, 20, -1, 4}, 则采用的是下列的()排序。

- A. 选择排序 B. 快速排序 C. 直接插入排序 D. 冒泡排序

12. 【2009年计算机联考真题】

若数据元素序列{11, 12, 13, 7, 8, 9, 23, 4, 5}是采用下列排序方法之一得到的第二趟排序后的结果, 则该排序算法只能是()。

- A. 冒泡排序 B. 插入排序 C. 选择排序 D. 二路归并排序

13. 对序列{98, 36, -9, 0, 47, 23, 1, 8, 10, 7}采用希尔排序, 下列序列()是增量为4的一趟排序结果。

- A. {10, 7, -9, 0, 47, 23, 1, 8, 98, 36}
B. {-9, 0, 36, 98, 1, 8, 23, 47, 7, 10}
C. {36, 98, -9, 0, 23, 47, 1, 8, 7, 10}
D. 以上都不对

14. 折半插入排序算法时间复杂度为()。

- A. $O(n)$ B. $O(n \log_2 n)$ C. $O(n^2)$ D. $O(n^3)$

15. 有些排序算法在每趟排序过程中, 都会有一个元素被放置到其最终位置上, 下列算法不会出现此种情况的是()。

- A. 希尔排序 B. 堆排序 C. 冒泡排序 D. 快速排序

16. 以下不稳定的排序算法是()。

- A. 冒泡排序 B. 直接插入排序 C. 希尔排序 D. 归并排序

17. 以下()算法是稳定的排序算法。

- A. 快速排序 B. 堆排序 C. 直接插入排序 D. 简单选择排序

18. 【2015年计算机联考真题】

希尔排序的组内排序采用的是()。

- A. 直接插入排序 B. 折半插入排序 C. 快速排序 D. 归并排序

二、综合应用题

1. 给出关键字序列{4, 5, 1, 2, 6, 3}的直接插入排序过程。

2. 给出关键字序列{50, 26, 38, 80, 70, 90, 8, 30, 40, 20}的希尔排序过程。(取增量序列为 $d=\{5, 3, 1\}$, 排序结果为从小到大排列)。

7.2.5 答案与解析

一、单项选择题

1. B

直接插入排序在最坏的情况下要做 $n(n-1)/2$ 次关键字的比较, 当 $n=5$ 时, 关键字的比较次数为 10 (未考虑与哨兵的比较)。

2. A

由于这里的序列是基本有序, 使用直接插入排序算法时间复杂度接近 $O(n)$ 。而使用其他算法时间复杂度均会大于 $O(n)$ 。

3. D

折半插入排序与直接插入排序都是将待插入元素插入前面的有序子表, 区别是: 确定当前记

录在前面有序子表中的位置时,直接插入排序是采用顺序查找法,而折半插入排序是采用折半查找法。排序的总趟数取决于元素个数 n ,两者都是 $n-1$ 趟。元素的移动次数都取决于初试序列,两者相同。使用辅助空间的数量也都是 $O(1)$ 。折半插入排序的比较次数与序列初态无关,为 $O(n\log_2 n)$;而直接插入排序的比较次数与序列初态有关,为 $O(n) \sim O(n^2)$ 。

4. D、A

当待排序表为反序时,直接插入排序需要进行 $n(n-1)/2$ 次比较(从前往后依次需要比较 1、2、...、 $n-1$ 次),当待排序表为正序时,只需进行 $n-1$ 次比较。

5. C

选项 A、B、D 的三种排序算法,在每趟排序后均会使一个记录存放在最终位置上。选项 C 在经过两趟排序后,原序列中的前 3 个元素会排成局部有序序列。

6. B

首先,越接近正序的序列,比较次数应该是越少的;而越接近逆序,则比较次数越多。不难得出 B 和 C 是比较接近正序的,然后再分别判断两个序列的比较次数,以 B 为例:第一趟,插入 32,比较 1 次;第二趟,插入 46,比较 1 次;第三趟,插入 40,由于 40 比 46 小但比 32 大,所以比较了 2 次;第四趟,插入 80,比较 1 次;第五趟,插入 69,比较 2 次;.....共比较 9 次。同理求出 C 的比较次数为 11 次。故选 B。

7. C

在直接插入排序中,如果待排序列中的最后一个元素应插入到表中第一个位置,则前面的有序子序列中的所有元素都不在最终位置上。

8. A

希尔排序是对直接插入排序算法改进后提出来的,本质上还是属于插入排序的范畴。

9. B

希尔排序将序列分成若干个组,记录只在组内进行交换。由观察可知,经过一趟后 9 和 -1 交换,7 和 4 交换,可知增量为 4。

10. B

首先,第二个元素为 1,是整个序列中的最小元素,所以可知该希尔排序为从小到大排序。然后考虑增量问题,若增量为 2,第 1+2 个元素 4 明显比第 1 个元素 9 要小, A 排除;若增量为 3,第 i 、 $i+3$ 、 $i+6$ 个元素都为有序序列($i=1,2,3$),符合希尔排序的定义;若增量为 4,第 1 个元素 9 比第 1+4 个元素 7 要大, C 排除;若增量为 5,第 1 个元素 9 比第 1+5 个元素 8 要大, D 排除,选 B。

11. C

前两个元素局部已经有序,很明显一趟直接插入排序算法有效。再排除其他算法即可。

12. B

由于冒泡和选择排序每一趟之后,总会有一个元素被放置在最终位置上,显然这里 {11, 12} 和 {4, 5} 所处位置并不是最终位置,故而不可能是冒泡和选择排序。二路归并算法经过第二趟后应该是每 4 个元素有序的,但 {11, 12, 13, 7} 并非有序,故而也不可能是二路归并排序。

13. A

增量为 4 意味着所有相距为 4 的记录构成一组,然后在组内进行直接插入排序,经观察,只有选项 A 满足要求。

14. C

虽然折半插入排序是对直接插入排序的改进,但它改进的只是比较的次数,而移动次数没有

发生变化,时间复杂度仍为 $O(n^2)$ 。

15. A

由于希尔排序是基于插入排序算法而提出的,它不一定在每趟排序过程后使某一元素放置到最终位置上。

16. C

特别注意,希尔排序虽然属于插入排序的范畴,但它是一个不稳定的排序方法。

17. C

此外还有希尔排序也是不稳定的排序方法,需要熟记。

18. A

希尔排序的思想是:先将待排元素序列分割成若干个子序列(由相隔某个“增量”的元素组成),分别进行直接插入排序,然后依次缩减增量再进行排序,待整个序列中的元素基本有序(增量足够小)时,再对全体元素进行一次直接插入排序。

二、综合应用题

1. 解答:

直接插入排序过程如下:

初始序列: 4, 5, 1, 2, 6, 3

第一趟: 4, 5, 1, 2, 6, 3 (将5插入到{4})

第二趟: 1, 4, 5, 2, 6, 3 (将1插入到{4, 5})

第三趟: 1, 2, 4, 5, 6, 3 (将2插入到{1, 4, 5})

第四趟: 1, 2, 4, 5, 6, 3 (将6插入到{1, 2, 4, 5})

第五趟: 1, 2, 3, 4, 5, 6 (将3插入到{1, 2, 4, 5, 6})

2. 解答:

原始序列: 50, 26, 38, 80, 70, 90, 8, 30, 40, 20

第一趟(增量5): 50, 8, 30, 40, 20, 90, 26, 38, 80, 70

第二趟(增量3): 26, 8, 30, 40, 20, 80, 50, 38, 90, 70

第二趟(增量1): 8, 20, 26, 30, 38, 40, 50, 70, 80, 90

7.3 交换排序

所谓交换,就是根据序列中两个元素关键字的比较结果来对换这两个记录在序列中的位置。基于交换的排序算法有很多,考研主要要求掌握冒泡排序和快速排序。其中冒泡排序算法比较简单,一般不会单独考查,重点是考查快速排序算法的相关内容。

7.3.1 冒泡排序

冒泡排序算法的基本思想是:假设待排序表长为 n ,从后往前(或从前往后)两两比较相邻元素的值,若为逆序(即 $A[i-1] > A[i]$),则交换它们,直到序列比较完。我们称它为一趟冒泡,结果将最小的元素交换到待排序列的第一个位置(关键字最小的元素如气泡一般逐渐往上“漂浮”直至“水面”,这就是冒泡排序名字的由来)。下一趟冒泡时,前一趟确定的最小元素不再参与比较,待排序列减少一个元素,每趟冒泡的结果把序列中的最小元素放到了序列的最终位置,……,

这样最多做 $n-1$ 趟冒泡就能把所有元素排好序。

```
void BubbleSort(ElemType A[],int n){
    //用冒泡排序法将序列 A 中的元素按从小到大排列
    for(i=0;i<n-1;i++){
        flag=false;                //表示本趟冒泡是否发生交换的标志
        for(j=n-1;j>i;j--)          //一趟冒泡过程
            if(A[j-1].key>A[j].key){ //若为逆序
                swap(A[j-1],A[j]); //交换
                flag=true;
            }
        if(flag==false)
            return ;                //本趟遍历后没有发生交换,说明表已经有序
    }
}
```

冒泡排序算法的性能分析如下:

空间效率: 仅使用了常数个辅助单元, 因而空间复杂度为 $O(1)$ 。

时间效率: 当初始序列有序时, 显然第一趟冒泡后 `flag` 依然为 `false` (本趟冒泡没有元素交换), 从而直接跳出循环, 比较次数为 $n-1$, 移动次数为 0, 从而最好情况下的时间复杂度为 $O(n)$; 当初始序列为逆序时, 需要进行 $n-1$ 趟排序, 第 i 趟排序要进行 $n-i$ 次关键字的比较, 而且每次比较都必须移动元素 3 次来交换元素位置。这种情况下,

$$\text{比较次数} = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}, \quad \text{移动次数} = \sum_{i=1}^{n-1} 3(n-i) = \frac{3n(n-1)}{2}$$

从而, 最坏情况下时间复杂度为 $O(n^2)$ 。其平均时间复杂度也为 $O(n^2)$ 。

稳定性: 由于当 $i > j$ 且 $A[i].key = A[j].key$ 时, 不会交换两个元素, 从而冒泡排序是一个稳定的排序方法。

注意: 冒泡排序中所产生的有序子序列一定是全局有序的 (不同于直接插入排序), 也就是说有序子序列中的所有元素的关键字一定小于或大于无序子序列中所有元素的关键字, 这样每一趟排序都会将一个元素放置到其最终的位置上。

7.3.2 快速排序

快速排序是对冒泡排序的一种改进。其基本思想是基于分治法的: 在待排序表 $L[1 \cdots n]$ 中任取一个元素 `pivot` 作为基准, 通过一趟排序将待排序表划分为独立的两部分 $L[1 \cdots k-1]$ 和 $L[k+1 \cdots n]$, 使得 $L[1 \cdots k-1]$ 中所有元素小于 `pivot`, $L[k+1 \cdots n]$ 中所有元素大于或等于 `pivot`, 则 `pivot` 放在了其最终位置 $L(k)$ 上, 这个过程称作一趟快速排序。而后分别递归地对两个子表重复上述过程, 直至每部分内只有一个元素或空为止, 即所有元素放在了其最终位置上。

首先假设划分算法已知, 记为 `Partition()`, 返回的是上述中的 k , 注意到 $L(k)$ 已经在最终的位置, 所以可以先对表进行划分, 而后对两个表调用同样的排序操作。因此可以递归地调用快速排序算法进行排序, 具体的程序结构如下:

```
void QuickSort(ElemType A[],int low,int high){
    if(low<high){                //递归跳出的条件
        //Partition()就是划分操作, 将表 A[low...high]划分为满足上述条件的两个子表
        int pivotpos=Partition(A,low,high); //划分
        QuickSort(A,low,pivotpos-1);        //依次对两个子表进行递归排序
    }
}
```



```
QuickSort(A,pivotpos+1,high);
```

从上面的代码也不难看出快速排序算法的关键在于划分操作，同时快速排序算法的性能也主要取决于划分操作的好坏。

从快速排序算法提出至今，有许多不同的划分操作版本，但考研通常所考查的快速排序的划分操作基本上以严版的教材为主。假设每次总是以当前表中第一个元素作为枢轴值（基准）对表进行划分，则必须将表中比枢轴值大的元素向右移动，比枢轴值小的元素向左移动，使得一趟 Partition() 操作之后，表中的元素被枢轴值一分为二。

```
int Partition(ElemType A[],int low,int high){
//严版教材中的划分算法（一趟排序过程）
    ElemType pivot=A[low]; //将当前表中第一个元素设为枢轴值，对表进行划分
    while(low<high){        //循环跳出条件
        while(low<high&&A[high]>=pivot) --high;
        A[low]=A[high];      //将比枢轴值小的元素移动到左端
        while(low<high&&A[low]<=pivot) ++low;
        A[high]=A[low];      //将比枢轴值大的元素移动到右端
    }
    A[low]=pivot;            //枢轴元素存放到最终位置
    return low;              //返回存放枢轴的最终位置
}
```

对算法的最好的理解方式就是手动地模拟一遍这些算法。

快速排序算法的性能分析如下：

空间效率：由于快速排序是递归的，需要借助一个递归工作栈来保存每一层递归调用的必要信息，其容量应与递归调用的最大深度一致。最好情况下为 $\lceil \log_2(n+1) \rceil$ ；最坏情况下，因为要进行 $n-1$ 次递归调用，所以栈的深度为 $O(n)$ ；平均情况下，栈的深度为 $O(\log_2 n)$ 。因而空间复杂度在最坏情况下为 $O(n)$ ，平均情况下为 $O(\log_2 n)$ 。

时间效率：快速排序的运行时间与划分是否对称有关，而后者又与具体使用的划分算法有关。快速排序的最坏情况发生在两个区域分别包含 $n-1$ 个元素和 0 个元素时，这种最大程度的不对称性若发生在每一层递归上，即对应于初始排序表基本有序或基本逆序时，就得到最坏情况下的时间复杂度为 $O(n^2)$ 。

有很多方法可以提高算法的效率。一种方法是当递归过程中划分得到的子序列的规模较小时不要再继续递归调用快速排序，可以直接采用直接插入排序算法进行后续的排序工作。另一种方法就是尽量选取一个可以将数据中分的枢轴元素。如从序列的头尾以及中间选取三个元素，再取这三个元素的中间值作为最终的枢轴元素；或者随机从当前表中选取枢轴元素，这样做使得最坏情况在实际排序中几乎不会发生。

在最理想的状态下，也即 Partition() 可能做到最平衡的划分中，得到的两个子问题的大小都不可能大于 $n/2$ ，在这种情况下，快速排序的运行速度将大大提升，此时，时间复杂度为 $O(n \log_2 n)$ 。好在快速排序平均情况下运行时间与其最佳情况下的运行时间很接近，而不是接近其最坏情况下的运行时间。快速排序是所有内部排序算法中平均性能最优的排序算法。

稳定性：在划分算法中，若右端区间存在两个关键字相同，且均小于基准值的记录，则在交换到左端区间后，它们的相对位置会发生变化，即快速排序是一个不稳定的排序方法。例如，表

$L=\{3, 2, 2\}$, 经过一趟排序后, $L=\{2, 2, 3\}$, 最终排序序列也是 $L=\{2, 2, 3\}$, 显然, 2 与 2 的相对次序已经发生了变化。

注意: 在快速排序算法中, 并不产生有序子序列, 但每一趟排序后将一个元素(基准元素)放到其最终的位置上。

7.3.3 本节试题精选

一、单项选择题

1. 对 n 个不同的元素利用冒泡法从小到大排序, 在 () 情况下元素交换的次数最多。

- A. 从大到小排列好的 B. 从小到大排列好的
C. 元素无序 D. 元素基本有序

2. 若用冒泡排序算法对序列 $\{10, 14, 26, 29, 41, 52\}$ 从大到小排序, 需进行 () 次比较。

- A. 3 B. 10 C. 15 D. 25

3. 【2010 年计算机联考真题】

对一组数据 $(2, 12, 16, 88, 5, 10)$ 进行排序, 若前 3 趟排序结果如下:

第一趟排序结果: $2, 12, 16, 5, 10, 88$

第二趟排序结果: $2, 12, 5, 10, 16, 88$

第三趟排序结果: $2, 5, 10, 12, 16, 88$

则采用的排序方法可能是 ()。

- A. 冒泡排序 B. 希尔排序 C. 归并排序 D. 基数排序

4. 【2011 年计算机联考真题】

为实现快速排序算法, 待排序序列宜采用的存储方式是 ()。

- A. 顺序存储 B. 散列存储 C. 链式存储 D. 索引存储

5. 用某种排序方法对线性表 $\{25, 84, 21, 47, 15, 27, 68, 35, 20\}$ 进行排序时, 元素序列的变化情况如下:

1) $25, 84, 21, 47, 15, 27, 68, 35, 20$

2) $20, 15, 21, 25, 47, 27, 68, 35, 84$

3) $15, 20, 21, 25, 35, 27, 47, 68, 84$

4) $15, 20, 21, 25, 27, 35, 47, 68, 84$

则所采用的排序方法是 ()。

- A. 选择排序 B. 插入排序 C. 二路归并排序 D. 快速排序

6. 一组记录的关键码为 $(46, 79, 56, 38, 40, 84)$, 则利用快速排序的方法, 以第一个记录为基准, 从小到大得到的一次划分结果为 ()。

- A. $(38, 40, 46, 56, 79, 84)$ B. $(40, 38, 46, 79, 56, 84)$
C. $(40, 38, 46, 56, 79, 84)$ D. $(40, 38, 46, 84, 56, 79)$

7. 快速排序算法在 () 情况下最不利于发挥其长处。

- A. 要排序的数据量太大 B. 要排序的数据中含有多个相同值
C. 要排序的数据个数为奇数 D. 要排序的数据已基本有序

8. 就平均性能而言, 目前最好的内排序方法是 ()。

- A. 冒泡排序 B. 直接插入排序 C. 希尔排序 D. 快速排序

9. 数据序列 $F=\{2, 1, 4, 9, 8, 10, 6, 20\}$ 只能是下列排序算法中的 () 的两趟排

序后的结果。

A. 快速排序 B. 冒泡排序 C. 选择排序 D. 插入排序

10. 对数据序列{8, 9, 10, 4, 5, 6, 20, 1, 2}采用冒泡排序(从后向前次序进行, 要求升序), 需要进行的趟数至少是()。

A. 3 B. 4 C. 5 D. 8

11. 对下列关键字序列用快排进行排序时, 速度最快的情形是(), 速度最慢的情形是()。

A. {21, 25, 5, 17, 9, 23, 30} B. {25, 23, 30, 17, 21, 5, 9}

C. {21, 9, 17, 30, 25, 23, 5} D. {5, 9, 17, 21, 23, 25, 30}

12. 对下列4个序列, 以第一个关键字为基准用快速排序算法进行排序, 在第一趟过程中移动记录次数最多的是()。

A. 92, 96, 88, 42, 30, 35, 110, 100

B. 92, 96, 100, 110, 42, 35, 30, 88

C. 100, 96, 92, 35, 30, 110, 88, 42

D. 42, 30, 35, 92, 100, 96, 88, 110

13. 下列序列中, ()可能是执行第一趟快速排序后所得到的序列。

I. {68, 11, 18, 69, 23, 93, 73} II. {68, 11, 69, 23, 18, 93, 73}

III. {93, 73, 68, 11, 69, 23, 18} IV. {68, 11, 69, 23, 18, 73, 93}

A. I、IV B. II、III C. III、IV D. 只有IV

14. 【2011年计算机联考真题】

下列选项中, 不可能是快速排序第2趟排序结果的是()。

A. 2,3,5,4,6,7,9

B. 2,7,5,6,4,3,9

C. 3,2,5,4,7,6,9

D. 4,2,3,5,7,6,9

15. 【2010年计算机联考真题】

采用递归方式对顺序表进行快速排序。下列关于递归次数的叙述中, 正确的是()。

A. 递归次数与初始数据的排列次序无关

B. 每次划分后, 先处理较长的分区可以减少递归次数

C. 每次划分后, 先处理较短的分区可以减少递归次数

D. 递归次数与每次划分后得到的分区的处理顺序无关

16. 对n个关键字进行快速排序, 最大递归深度为(), 最小递归深度为()。

A. 1

B. n

C. $\log_2 n$

D. $n\log_2 n$

二、综合应用题

1. 在使用非递归方法实现快速排序时, 通常要利用一个栈记忆待排序区间的两个端点。那么能否用队列来实现这个栈? 为什么?

2. 编写双向冒泡排序算法, 在正反两个方向交替进行扫描, 即第一趟把关键字最大的元素放在序列的最后面, 第二趟把关键字最小的元素放在序列的最前面。如此反复进行。

3. 已知线性表按顺序存储, 且每个元素都是不相同的整数型元素, 设计把所有奇数移动到所有偶数前边的算法(要求时间最少, 辅助空间最少)。

4. 试重新编写考点精析中的快速排序的划分算法, 使之每次选取的枢轴值都是随机地从当前子表中选择的。

5. 试编写一个算法, 使之能够在数组 $L[1 \dots n]$ 中找出第k小的元素(即从小到大排序后处于

第 k 个位置的元素)。

6. 【2016 年计算机联考真题】

已知由 n ($n \geq 2$) 个正整数构成的集合 $A = \{a_k | 0 \leq k < n\}$, 将其划分为两个不相交的子集 A_1 和 A_2 , 元素个数分别是 n_1 和 n_2 , A_1 和 A_2 中元素之和分别为 S_1 和 S_2 。设计一个尽可能高效的划分算法, 满足 $|n_1 - n_2|$ 最小且 $|S_1 - S_2|$ 最大。要求:

- 1) 给出算法的基本设计思想。
- 2) 根据设计思想, 采用 C 或 C++ 语言描述算法, 关键之处给出注释。
- 3) 说明你所设计算法的平均时间复杂度和空间复杂度。

7. 荷兰国旗问题: 设有一个仅由红、白、蓝三种颜色的条块组成的条块序列, 请编写一个时间复杂度为 $O(n)$ 的算法, 使得这些条块按红、白、蓝的顺序排好, 即排成荷兰国旗图案。

7.3.4 答案与解析

一、单项选择题

1. A

通常情况下, 冒泡排序最少进行 1 次冒泡, 最多进行 $n-1$ 次冒泡。若初始序列为逆序时, 则需进行 $n-1$ 次冒泡, 并且需要交换的次数最多。若初始序列为正序时, 则进行 1 次冒泡 (无交换) 就可以终止算法。

2. C

冒泡排序始终在调整“逆序”, 因此交换次数为排列中逆序的个数。对逆序序列进行冒泡排序, 每一个元素向后调整时都需要进行比较, 因此共需要比较 $5+4+3+2+1=15$ 次。

3. A

分别用其他 3 种排序算法执行数据, 归并排序第一趟结果为 (2, 12, 16, 88, 5, 10), 基数排序第一趟后结果为 (10, 2, 12, 5, 16, 88), 希尔排序显然是不符合的。只有冒泡排序符合条件。

另解: 由题干可以看出每一趟都产生一个最大的数排在后面, 可直接定位冒泡排序。

4. A

对绝大部分内部排序而言, 只适用于顺序存储结构。快速排序在排序的过程中, 既要从后向前查找, 也要从前向后查找, 因此宜采用顺序存储。

5. D

选择排序在每趟结束后可以确定一个元素的最终位置, 因此不对。插入排序, 第 i 趟后前 $i+1$ 个元素应该是有序的, 因此不对。第 2 趟 {20, 15} 和 {21, 25} 是反序的, 因此不是归并排序。快速排序每趟都将基准元素放在其最终位置, 然后以它为基准将序列划分为两个子序列。观察题中的排序过程, 可知是快速排序。

6. C

以 46 为基准元素, 首先从后向前扫描比 46 小的元素, 并与之进行交换, 而后再从前向后扫描比 46 大的元素并将 46 与该元素交换, 从而得到了 (40, 46, 56, 38, 79, 84)。而后继续重复从后向前扫描与从前向后扫描的操作, 直到 46 处于最终位置, 答案选 C。

7. D

当待排序数据为基本有序时, 每次选取第 n 个元素为基准时, 会导致划分区间分配不均匀, 不利于发挥快速排序算法的优势。相反的, 当待排序数据分布较为随机时, 基准元素能将序列划分为两个长度大致相等的序列, 这时才能发挥快速排序的优势。

8. D

这里问的是平均性能，A、B 的平均性能都会达到 $O(n^2)$ ，而希尔排序虽然大大地降低了直接插入排序的时间复杂度，但它的平均性能不如快速排序。另外，虽然有众多排序算法平均时间复杂度也是 $O(n\log_2 n)$ ，但快速排序算法的常数因子是最小的。

9. A

若为插入排序，则前三个元素应该是有序的，显然不对。而冒泡排序和选择排序经过两趟排序后应该有两个元素处于最终位置（最左/右端），无论是按从小到大还是从大到小排序，数据序列中都没有两个满足这样条件的元素。只可能选 A。

另解：先写出排好序的序列，并和题中的序列做对比。

题中序列：2 1 4 9 8 10 6 20

已排好序：1 2 4 6 8 9 10 20

在已排好序的序列中，和题中序列相同的有 4、8 和 20，最左和最右两个元素和题中序列不同，故不可能是冒泡排序、选择排序或插入排序。

10. C 从后向前“冒泡”的过程为，第 1 趟{1, 8, 9, 10, 4, 5, 6, 20, 2}，第 2 趟{1, 2, 8, 9, 10, 4, 5, 6, 20}，第 3 趟{1, 2, 4, 8, 9, 10, 5, 6, 20}，第 4 趟{1, 2, 4, 5, 8, 9, 10, 6, 20}，第 5 趟{1, 2, 4, 5, 6, 8, 9, 10, 20}，经过第 5 趟冒泡后，序列已经全局有序，故选 C。

11. A、D

由考点精析可知，当每次的枢轴都把表等分为长度相近的二个子表时，速度是最快的；当表本身已经有序或逆序时，速度最慢。选项 D 中的序列已按关键字排好序，因此它是最慢的，而 A 中第一趟枢轴值 21 将表划分为二个子表 {9, 17, 5} 和 {25, 23, 30}，而后对两个子表划分时，枢轴值再次地将它们等分，所以，该序列是快速排序最优的情况，速度最快。其他选项可以类似分析。

12. B

对各序列分别执行一趟快速排序，则可以做如下分析（以 A 为例）：由于枢轴值为 92，则 35 移动到第一个位置，96 移动到第六个位置，30 移动到第二个位置，再将枢轴值移动到 30 所在的单元，即第五个位置，所以 A 中序列移动次数为 4。同样可以分析出 B 中序列的移动次数为 8，C 序列移动次数为 4，D 序列移动次数为 2。

13. C

显然，如果按从小到大排序，则最终有序的序列是{11, 18, 23, 68, 69, 73, 93}；如果按从大到小排序，则最终有序的序列是{93, 73, 69, 68, 23, 18, 11}。对比可知 I，II 中没有处于最终位置的元素，故 I，II 都不可能。III 中 73 和 93 处于从大到小排序后的最终位置，而且 73 将序列分割成大于 73 和小于 73 的两部分，故而 III 是有可能的。IV 中 73 和 93 处于从小到大排列后最终位置，73 也满足把序列分割成大于 73 和小于 73 的两部分。

14. C

快排的阶段排序结果的特点是，第 i 趟完成时，会有 i 个以上的数出现在它最终将要出现的位置，即它左边的数都比它小，它右边的数都比它大。题目问第二趟排序的结果，即要找不存在 2 个这样的数的选项。A 选项中 2、3、6、7、9 均符合，所以 A 排除；B 选项中 2、9 均符合，所以 B 排除；D 选项中 5、9 均符合，所以 D 选项排除；最后看 C 选项，只有 9 一个数符合，所以 C 不可能是快速排序第二趟的结果。

15. D

递归次数与各元素的初始排列有关。如果每次划分后分区比较平衡，则递归次数少；如果分

区不平衡,递归次数多。递归次数与处理顺序是没有关系的。

16. B、C

快速排序过程构成一个递归树,递归深度即为递归树的高度。当枢轴值每次都子表等分时,此时递归树的高为 $\log_2 n$;当枢轴值每次都是子表的最大值或最小值时,此时递归树退化为单链表,树高为 n 。

二、综合应用题

1. 解答:

可以用队列来代替栈。在快速排序的过程中,通过一趟划分,可以把一个待排序区间分为两个子区间,然后分别对这两个子区间施行同样的划分。栈的作用是在处理一个子区间时,保存另一个子区间的上界和下界(排序过程中可能产生新的左、右子区间),待该区间处理完后再从栈中取出另一子区间的边界,对其进行处理。这个功能用队列也可以实现,只不过处理子区间的顺序有所变动而已。

2. 解答:

这种排序方法又称为“双向起泡”。奇数趟时,从前向后比较相邻元素的关键字,遇到逆序即交换,直到把序列中关键字最大的元素移动到序列尾部。偶数趟时,从后往前比较相邻元素的关键字,遇到逆序即交换,直到把序列中关键字最小的元素移动到序列前端。

```
void BubbleSort (ElemType A[],int n){
//双向起泡排序,交替进行正反两个方向的起泡过程
    int low=0,high=n-1;
    bool flag=true;                                //一趟冒泡后记录元素是否交换标志
    while(low<high&&flag){                          //循环跳出条件,当flag为false说明已没有逆序
        flag=false;                                //每趟初始置flag为false
        for(i=low;i<high;i++){                      //从前向后起泡
            if(a[i]>a[i+1]){                          //发生逆序
                swap(a[i],a[i+1]);                  //交换
                flag=true;                          //置flag
            }
        }
        high--;                                    //更新上界
        for(i=high;i>low;i--){                      //从后往前起泡
            if(a[i]<a[i-1]){                          //发生逆序
                swap(a[i],a[i-1]);                  //交换
                flag=true;                          //置flag
            }
        }
        low++;                                    //修改下界
    }
}
```

3. 解答:

本题可以采用基于快速排序的划分思想来设计算法,只需遍历一次即可,其时间复杂度为 $O(n)$,空间复杂度为 $O(1)$ 。假设表为 $L[1 \cdots n]$,基本思想是:先从前向后找到一个偶数元素 $L(i)$,再从后向前找到一个奇数元素 $L(j)$,将二者交换;重复上述过程直到 i 大于 j 。

算法的实现如下:

```
void move(ElemType A[],int len){
//对表A按奇偶进行一趟划分
    int i=0,j=len-1;    //i表示左端偶数元素的下标;j表示右端奇数元素的下标
```



```

while(i<j){
    while(i<j&&A[i]%2!=0) i++; //从前向后找到一个偶数元素
    while(i<j&&A[j]%2!=1) j--; //从后向前找到一个奇数元素
    if (i<j){
        Swap(A[i],A[j]); //交换这两个元素
        i++; j--;
    }
}
}

```

4. 解答:

这类题目比较简单, 为了方便起见, 可以直接先随机地求出枢轴的下标, 然后将枢轴值与 $A[\text{low}]$ 交换, 而后的思想就是和前面的划分算法一样了。

```

int Partition2(ElemType A[],int low,int high){
    int rand_Index=low+rand()%(high-low+1);
    Swap(A[rand_Index],A[low]); //将枢轴值交换到第一个元素
    ElemType pivot=A[low]; //置当前表中第一个元素为枢轴值
    int i=low; //使得表 A[low...i] 中的所有元素小于 pivot, 初始为空表
    for(int j=low+1;j<=high;j++) //从第二个元素开始寻找小于基准的元素
        if(A[j]<pivot) //找到后, 则交换到前面
            swap(A[++i],A[j]);
    swap(A[i],A[low]); //将基准元素插入到最终位置
    return i; //返回基准元素的位置
}

```

注意, 本题代码中的比较方法和考点精析中的两种方法均不相同, 请读者仔细模拟各种划分算法的执行步骤以做到真正掌握, 对于下题, 读者可以尝试采用这三种比较方法来分别解决。

5. 解答:

显然, 本题最直接的做法就是用排序算法对数组先进行从小到大的排序, 然后直接提取 $L(k)$ 便得到了第 k 小元素, 但其平均时间复杂度将达到 $O(n\log_2 n)$ 以上。此外, 还可以采用小顶堆的方法, 每次堆顶元素都是最小值元素, 时间复杂度为 $O(n+k\log_2 n)$ 。下面介绍一个更精彩的算法, 它是基于快速排序的划分操作的。

它的主要思想是:

从数组 $L[1\cdots n]$ 中选择枢轴 pivot (随机地或者直接取第一个) 进行和快速排序一样的划分操作后, 表 $L[1\cdots n]$ 被划分为 $L[1\cdots m-1]$ 和 $L[m+1\cdots n]$, 其中 $L(m)=\text{pivot}$ 。

讨论 m 与 k 的大小关系:

- 1) 当 $m=k$ 时, 显然 pivot 就是所要寻找的元素, 直接返回 pivot 即可。
- 2) 当 $m<k$ 时, 则所要寻找的元素一定落在 $L[m+1\cdots n]$ 中, 从而可以对 $L[m+1\cdots n]$ 递归地查找第 $k-m$ 小的元素。
- 3) 当 $m>k$ 时, 则所要寻找的元素一定落在 $L[1\cdots m-1]$ 中, 从而可以对 $L[1\cdots m-1]$ 递归地查找第 k 小的元素。

这个算法的时间复杂度在平均情况下可以达到 $O(n)$ 。而所占空间复杂度则取决于划分的方法。该算法的实现如下:

```

int kth_elem(int a[],int low,int high,int k){
    int pivot=a[low];
    int low_temp=low; //由于下面会修改 low 与 high, 在递归时又要用到它们
    int high_temp=high;

```



```

while(low<high){
    while(low<high&& a[high]>=pivot)
        --high;
    a[low]=a[high];
    while(low<high&& a[low]<=pivot)
        ++low;
    a[high]=a[low];
}
a[low]=pivot;
//上面即为快速排序中的划分算法
//以下就是本算法思想中所述的内容
if(low==k)                //由于与 k 相同, 直接返回 pivot 元素
    return a[low];
else if(low>k)             //在前一部分表中递归寻找
    return kth_elem(a, low_temp, low-1, k);
else                      //在后一部分表中递归寻找
    return kth_elem(a, low+1, high_temp, k-low);
}

```

6. 解答:

1) 算法的基本设计思想

由题意知, 将最小的 $\lfloor n/2 \rfloor$ 个元素放在 A_1 中, 其余的元素放在 A_2 中, 分组结果即可满足题目要求。仿照快速排序的思想, 基于枢轴将 n 个整数划分为两个子集。根据划分后枢轴所处的位置 i 分别处理:

①若 $i = \lfloor n/2 \rfloor$, 则分组完成, 算法结束;

②若 $i < \lfloor n/2 \rfloor$, 则枢轴及之前的所有元素均属于 A_1 , 继续对 i 之后的元素进行划分;

③若 $i > \lfloor n/2 \rfloor$, 则枢轴及之后的所有元素均属于 A_2 , 继续对 i 之前的元素进行划分;

基于该设计思想实现的算法, 毋须对全部元素进行全排序, 其平均时间复杂度是 $O(n)$, 空间复杂度是 $O(1)$ 。

2) 算法实现 (9 分)

```

int setPartition(int a[], int n){
    int pivotkey, low=0, low0=0, high=n-1, high0=n-1, flag=1, k=n/2, i;
    int s1=0, s2=0;
    while(flag) {
        pivotkey=a[low];                //选择枢轴
        while(low<high) {               //基于枢轴对数据进行划分
            while(low<high && a[high]>=pivotkey) --high;
            if(low!=high) a[low]=a[high];
            while(low<high && a[low]<=pivotkey) ++low;
            if(low!=high) a[high]=a[low];
        }                               //end of while(low<high)
        a[low]=pivotkey;
        if(low==k-1)                    //如果枢轴是第n/2小元素, 划分成功
            flag=0;
        else{                           //是否继续划分
            if(low<k-1){
                low0=++low;
            }
        }
    }
}

```



```

        high=high0;
    }
    else{
        high0--high;
        low=low0;
    }
}
}
for(i=0;i<k;i++) s1+=a[i];
for(i=k;i<n;i++) s2+=a[i];
return s2-s1;
}

```

3) 本答案给出的算法平均时间复杂度是 $O(n)$, 空间复杂度是 $O(1)$ 。

7. 解答:

算法的思想: 顺序扫描线性表, 将红色条块交换到线性表的最前面, 蓝色条块交换到线性表的最后面, 为此设立三个指针, 其中, j 为工作指针表示当前扫描的元素, i 以前的元素全部为红色, k 以后的元素全部为蓝色。根据 j 所指示元素的颜色, 决定将其交换到序列的前部或者尾部。初始时 $i=0$, $k=n-1$, 算法的实现如下:

```

typedef enum{RED,WHITE,BLUE} color; //设置枚举数组
void Flag_Arrange(color a[],int n){
    int i=0,j=0,k=n-1;
    while(j<=k)
        switch(a[j]){ //判断条块的颜色
            case RED: Swap(a[i],a[j]);i++;j++;break;
                //红色, 则和 i 交换
            case WHITE: j++;break;
            case BLUE: Swap(a[j],a[k]);k--;
                //蓝色, 则和 k 交换
                //这里没有 j++语句以防止交换后 a[j] 仍为蓝色的情况
        }
    }
}

```

如将元素值为正数、负数和零排序成前面都是负数, 接着是 0, 最后是正数, 也是用同样的方法。思考, 为什么 case RED 时不用考虑交换后 $a[j]$ 仍为红色, 而 case BLUE 却需要考虑交换后 $a[j]$ 仍为蓝色?

7.4 选择排序

选择排序的基本思想是: 每一趟 (例如第 i 趟) 在后面 $n-i+1$ ($i=1, 2, \dots, n-1$) 个待排序元素中选取关键字最小的元素, 作为有序子序列的第 i 个元素, 直到第 $n-1$ 趟做完, 待排序元素只剩下 1 个, 就不用再选了。选择排序中的堆排序算法是考查的重点。

7.4.1 简单选择排序

从上面选择排序的思想中可以很直观地得出简单选择排序算法的思想: 假设排序表为 $L[1 \dots n]$, 第 i 趟排序即从 $L[i \dots n]$ 中选择关键字最小的元素与 $L(i)$ 交换, 每一趟排序可以确定一个元素的最终位置, 这样经过 $n-1$ 趟排序就可以使得整个排序表有序。

下面是简单选择排序算法的伪代码:

```

void SelectSort(ElemType A[],int n){
    //对表A作简单选择排序，A[]从0开始存放元素
    for(i=0;i<n-1;i++){
        min=i;
        for(j=i+1;j<n;j++){
            if(A[j]<A[min]) min=j;
            if(min!=i) swap(A[i],A[min]);
        }
    }
}

```

简单选择排序算法的性能分析如下：

空间效率：仅使用常数个辅助单元，故而空间效率为 $O(1)$ 。

时间效率：从上述伪码中不难看出，简单选择排序过程中，元素移动的操作次数很少，不会超过 $3(n-1)$ 次，最好的情况是移动 0 次，此时对应的表已经有序；但元素间比较的次数与序列的初始状态无关，始终是 $n(n-1)/2$ 次，所以时间复杂度始终是 $O(n^2)$ 。

稳定性：在第 i 趟找到最小元素后，和第 i 个元素交换，可能会导致第 i 个元素与其含有相同关键字元素的相对位置发生改变。例如，表 $L=\{2, 2, 1\}$ ，经过一趟排序后， $L=\{1, 2, 2\}$ ，最终排序序列也是 $L=\{1, 2, 2\}$ ，显然，2 与 2 的相对次序已经发生了变化。因此，简单选择排序是一个不稳定的排序方法。

7.4.2 堆排序

堆排序是一种树形选择排序方法，它的特点是：在排序过程中，将 $L[1 \cdots n]$ 看成是一棵完全二叉树的顺序存储结构，利用完全二叉树中双亲结点和孩子结点之间的内在关系，在当前无序区中选择关键字最大（或最小）的元素。

堆的定义如下： n 个关键字序列 $L[1 \cdots n]$ 称为堆，当且仅当该序列满足：

① $L(i) \leq L(2i)$ 且 $L(i) \leq L(2i+1)$ 或 ② $L(i) \geq L(2i)$ 且 $L(i) \geq L(2i+1)$ ($1 \leq i \leq \lfloor n/2 \rfloor$)

满足第①种情况的堆称为**小根堆**（小顶堆），满足第②种情况的堆称为**大根堆**（大顶堆）。显然，在大根堆中，最大元素存放在根结点中，且对其任一非根结点，它的值小于或等于其双亲结点值。小根堆的定义刚好相反，根结点是最小元素。堆经常被用来实现优先级队列，优先级队列在操作系统的作业调度和其他领域有广泛的应用。图 7-1 所示为一个**大根堆**。

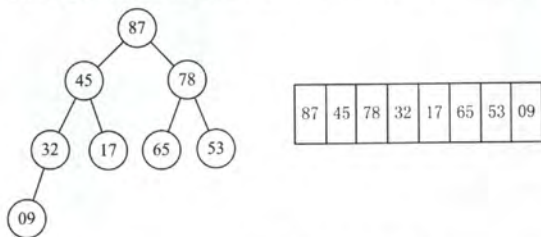


图 7-1 一个大根堆示意图

堆排序的关键是构造初始堆，对初始序列建堆，就是一个反复筛选的过程。 n 个结点的完全二叉树，最后一个结点是第 $\lfloor n/2 \rfloor$ 个结点的孩子。对第 $\lfloor n/2 \rfloor$ 个结点为根的子树筛选（对于大根堆：若根结点的关键字小于左右子女中关键字较大者，则交换），使该子树成为堆。之后向前依次对各结点 $(\lfloor n/2 \rfloor - 1 \sim 1)$ 为根的子树进行筛选，看该结点值是否大于其左右子结点的值，若不是，将左右子结点中较大值与之交换，交换后可能会破坏下一级的堆，于是继续采用上述方法构造下一级的堆，直到以该结点为根的子树构成堆为止。反复利用上述调整堆的方法建堆，直到根结点，调整过程的示例如图 7-2 所示。

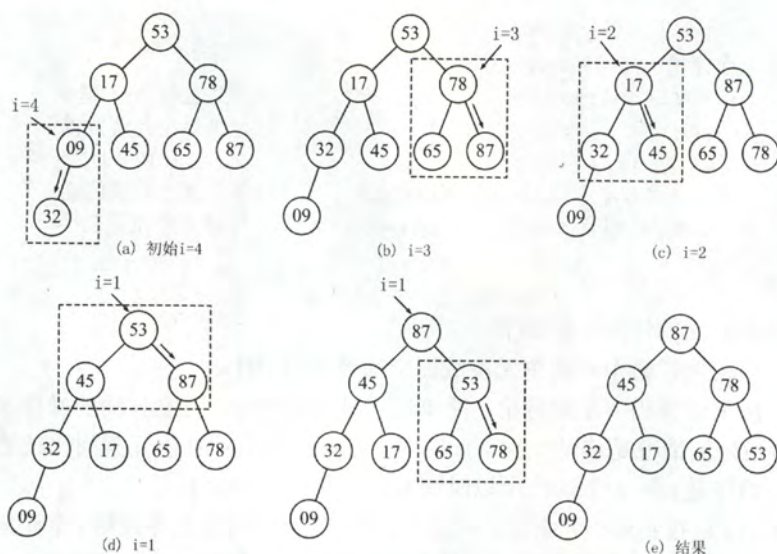


图 7-2 自下往上逐步调整为大根堆

下面是建立大根堆的算法:

```
void BuildMaxHeap(ElemType A[],int len){
    for(int i=len/2;i>0;i--) //从 i=[n/2]~1, 反复调整堆
        AdjustDown(A,i,len);
}

void AdjustDown(ElemType A[],int k,int len){
    //函数 AdjustDown 将元素 k 向下进行调整
    A[0]=A[k]; //A[0] 暂存
    for(i=2*k;i<=len;i*=2){ //沿 key 较大的子结点向下筛选
        if(i<len&&A[i]<A[i+1])
            i++; //取 key 较大的子结点的下标
        if(A[0]>=A[i]) break; //筛选结束
        else{
            A[k]=A[i]; //将 A[i] 调整到双亲结点上
            k=i; //修改 k 值, 以便继续向下筛选
        }
    }
    A[k]=A[0]; //被筛选结点的值放入最终位置
}
```

向下调整的时间与树高有关, 为 $O(h)$ 。建堆过程中每次向下调整时, 大部分结点的高度都较小。因此, 可以证明在元素个数为 n 的序列上建堆, 其时间复杂度为 $O(n)$, 这说明可以在线性时间内, 将一个无序数组建成一个大顶堆。

应用堆这种数据结构进行排序的思路很简单, 首先将存放在 $L[1..n]$ 中的 n 个元素建成初始堆, 由于堆本身的特点 (以大顶堆为例), 堆顶元素就是最大值。输出堆顶元素后, 通常将堆底元素送入堆顶, 此时根结点已不满足大顶堆的性质, 堆被破坏, 将堆顶元素向下调整使其继续保持大顶堆的性质, 再输出堆顶元素。如此重复, 直到堆中只剩下一个元素为止。

下面是堆排序算法:

```
void HeapSort(ElemType A[],int len){
    BuildMaxHeap(A,len); //初始建堆
    for(i=len;i>1;i--){ //n-1 趟的交换和建堆过程
```

```

        Swap(A[i],A[1]);          //输出堆顶元素(和堆底元素交换)
        AdjustDown(A,1,i-1);      //整理,把剩余的 i-1 个元素整理成堆
    }//for
}

```

同时,堆也支持删除和插入的操作。由于堆顶元素或为最大值或为最小值,删除堆顶元素时,先将堆的最后一个元素与堆顶元素交换,由于此时堆的性质被破坏,需对此时的根结点进行向下调整操作。对堆进行插入操作时,先将新结点放在堆的末端,再对这个新结点执行向上调整操作,大根堆的插入操作示例如图 7-3 所示。

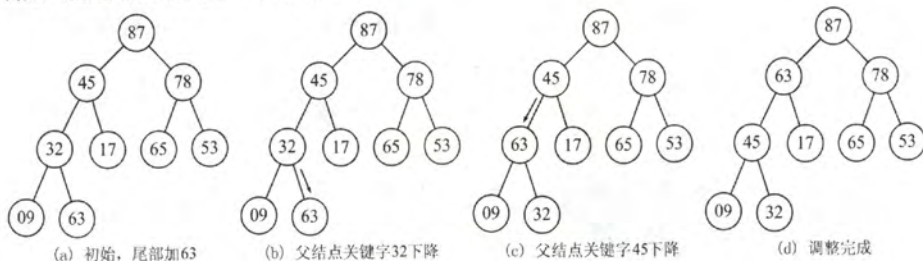


图 7-3 大根堆的插入操作示例

下面是向上调整堆的算法:

```

void AdjustUp(ElemType A[],int k){
    //参数 k 为向上调整的结点,也为堆的元素个数
    A[0]=A[k];
    int i=k/2; //若结点值大于双亲结点,则将双亲结点向下调,并继续向上比较
    while(i>0&&A[i]<A[0]){ //循环跳出条件
        A[k]=A[i];          //双亲结点下调
        k=i;
        i=k/2;              //继续向上比较
    }//while
    A[k]=A[0];              //复制到最终位置
}

```

堆排序算法的性能分析如下:

空间效率:仅使用了常数个辅助单元,所以空间复杂度为 $O(1)$ 。

时间效率:建堆时间为 $O(n)$,之后有 $n-1$ 次向下调整操作,每次调整的时间复杂度为 $O(h)$,故在最好、最坏和平均情况下,堆排序的时间复杂度为 $O(n\log_2 n)$ 。

稳定性:在进行筛选时,有可能把后面相同关键字的元素调整到前面,所以堆排序算法是一种不稳定的排序方法。例如,表 $L=\{1, 2, 2\}$,构造初始堆时,可能将 2 交换到堆顶,此时 $L=\{2, 1, 2\}$,最终排序序列为 $L=\{1, 2, 2\}$,显然,2 与 2 的相对次序已经发生了变化。

7.4.3 本节试题精选

一、单项选择题

- 在以下排序算法中,每次从未排序的记录中选取最小关键字的记录,加入到已排序记录的末尾,该排序方法是 ()。
A. 简单选择排序 B. 冒泡排序 C. 堆排序 D. 直接插入排序
- 简单选择排序算法的比较次数和移动次数分别为 ()。
A. $O(n)$, $O(\log_2 n)$ B. $O(\log_2 n)$, $O(n^2)$
C. $O(n^2)$, $O(n)$ D. $O(n\log_2 n)$, $O(n)$

3. 设线性表中每个元素有两个数据项 k_1 和 k_2 , 现对线性表按以下规则进行排序: 先看数据项 k_1 , k_1 值小的元素在前, 大的在后; 在 k_1 值相同的情况下, 再看 k_2 , k_2 值小的在前, 大的在后。满足这种要求的排序方法是 ()。

- A. 先按 k_1 进行直接插入排序, 再按 k_2 进行简单选择排序
- B. 先按 k_2 进行直接插入排序, 再按 k_1 进行简单选择排序
- C. 先按 k_1 进行简单选择排序, 再按 k_2 进行直接插入排序
- D. 先按 k_2 进行简单选择排序, 再按 k_1 进行直接插入排序

4. 如果只想得到 1000 个元素组成的序列中第 10 个最小元素之前的部分排序的序列, 用 () 方法最快。

- A. 冒泡排序
- B. 快速排序
- C. 希尔排序
- D. 堆排序

5. 下列 () 是一个堆。

- A. 19, 75, 34, 26, 97, 56
- B. 97, 26, 34, 75, 19, 56
- C. 19, 56, 26, 97, 34, 75
- D. 19, 34, 26, 97, 56, 75

6. 有一组数据 (15, 9, 7, 8, 20, -1, 7, 4), 用堆排序的筛选方法建立的初始小根堆为 ()。

- A. -1, 4, 8, 9, 20, 7, 15, 7
- B. -1, 7, 15, 7, 4, 8, 20, 9
- C. -1, 4, 7, 8, 20, 15, 7, 9
- D. A、B、C 均不对

7. 在含有 n 个关键字的小根堆中, 关键字最大的记录有可能存储在 ()。

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

8. 向具有 n 个结点的堆中插入一个新元素的时间复杂度为 (), 删除一个元素的时间复杂度为 ()。

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

9. 构建 n 个记录的初始堆, 其时间复杂度为 (); 对 n 个记录进行堆排序, 最坏情况下其时间复杂度为 ()。

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

10. 对关键字序列 {23, 17, 72, 60, 25, 8, 68, 71, 52} 进行堆排序, 输出两个最小关键字后的剩余堆是 ()。

- A. {23, 72, 60, 25, 68, 71, 52}
- B. {23, 25, 52, 60, 71, 72, 68}
- C. {71, 25, 23, 52, 60, 72, 68}
- D. {23, 25, 68, 52, 60, 72, 71}

11. 【2009 年计算机联考真题】

已知关键字序列 5, 8, 12, 19, 28, 20, 15, 22 是小根堆, 插入关键字 3, 调整后得到的小根堆是 ()。

- A. 3, 5, 12, 8, 28, 20, 15, 22, 19
- B. 3, 5, 12, 19, 20, 15, 22, 8, 28
- C. 3, 8, 12, 5, 20, 15, 22, 28, 19
- D. 3, 12, 5, 8, 28, 20, 15, 22, 19

12. 【2011 年计算机联考真题】

已知序列 25, 13, 10, 12, 9 是大根堆, 在序列尾部插入新元素 18, 将其再调整为大根堆,

调整过程中元素之间进行的比较次数是 ()。

- A. 1 B. 2 C. 4 D. 5

13. 下列四种排序方法中, 排序过程中的比较次数与序列初始状态无关的是 ()。

- A. 选择排序法 B. 插入排序法 C. 快速排序法 D. 冒泡排序法

14. 【2015 年计算机联考真题】

已知小根堆为 8,15,10,21,34,16,12, 删除关键字 8 之后需重建堆, 在此过程中, 关键字之间的比较次数是 ()。

- A. 1 B. 2 C. 3 D. 4

二、综合应用题

1. 指出堆和二叉排序树的区别?

2. 如果只想得到一个序列中第 k ($k \geq 5$) 个最小元素之前的部分排序序列, 最好采用什么排序方法?

3. 若有 n 个元素已构成一个小根堆, 那么如果增加一个元素 K_{n+1} , 请用文字简要说明你如何在 $\log_2 n$ 的时间内将其重新调整为一个堆?

4. 编写一个算法, 在基于单链表表示的待排序关键字序列上进行简单选择排序。

5. 试设计一个算法, 判断一个数据序列是否构成一个小根堆。

7.4.4 答案与解析

一、单项选择题

1. A

2. C

注意: 读者应对各种排序算法的思想、过程、特点等都熟练掌握。

3. D

首先应确定 k_1 、 k_2 的排序顺序, 若先排 k_1 再排 k_2 , 则排序结果不符合题意, 排除 A、C。再考虑算法的稳定性, 当 k_2 排好序后, 再对 k_1 排序, 若对 k_1 排序采用的算法是不稳定的, 则对于 k_1 相同、而 k_2 不同的元素可能会改变相对次序, 从而不一定能满足题干中“在 k_1 值相同的情况下, k_2 值小的在前, 大的在后”的条件。直接插入排序算法是稳定的, 而简单选择排序算法是不稳定的, 故只能选 D。

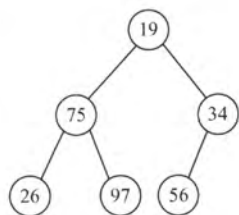
4. D

希尔排序和快速排序要等排序全部完成之后才能确定最小的 10 个元素。冒泡排序需要从后向前执行 10 趟冒泡才能得到 10 个最小的元素, 而堆排序只需要调整 10 次小根堆, 调整时间与树高成正比。显然堆排序所需时间更短。

通常, 取一大堆数据中的 k 个最大 (最小) 的元素, 都优先采用堆排序。

5. D

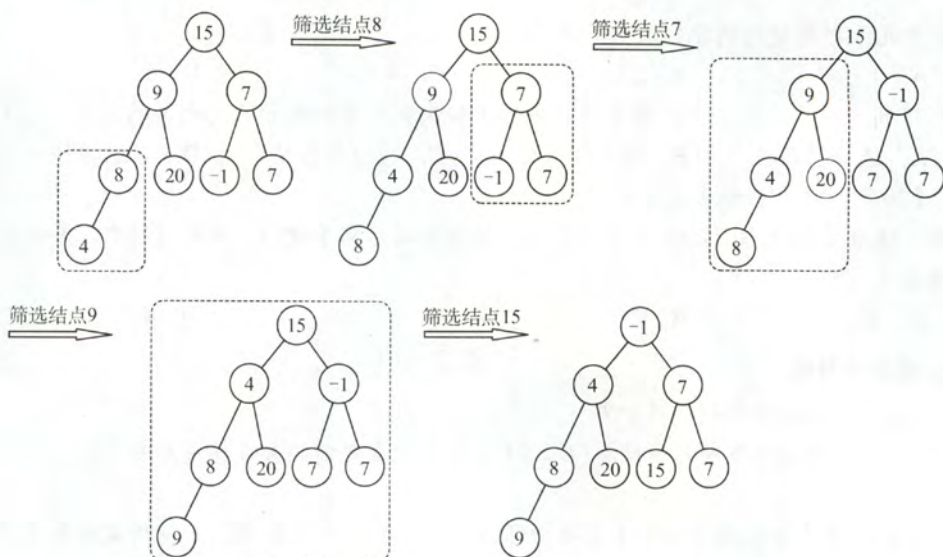
可以将每个选项中的序列表示成完全二叉树, 再看父结点与子结点的关系是否全部满足堆的定义。比如 A 选项中序列对应的完全二叉树如右图所示: 显然, 最小元素 19 在根结点, 所以可能是小根堆, 但 75 与 26 的关系却不满足小根堆的定义, 所以 A 选项中序列不是一个堆。其他选项采用类似的过程分析。



第 5 题图

6. C

从 $\lfloor n/2 \rfloor \sim 1$ 依次筛选堆的过程如下图所示, 显然选 C。



7. B

由于这是小根堆，所以关键字最大的记录一定存储在这个堆所对应的完全二叉树的叶子结点中，又二叉树中最后一个非叶子结点存储在 $\lfloor n/2 \rfloor$ 中，所以关键字最大记录存储范围在 $\lfloor n/2 \rfloor + 1 \sim n$ 中，所以应该选 B。

8. C、C

在向有 n 个元素的堆中插入一个新元素时，需要调用一个向上调整的算法，比较次数最多等于树的高度减 1，由于树的高度为 $\lfloor \log_2 n \rfloor + 1$ ，所以堆的向上调整算法的比较次数最多等于 $\lfloor \log_2 n \rfloor$ 。此处需要注意到，调整堆和建初始堆的时间复杂度是不一样的，读者可以仔细分析两个算法的具体执行过程。

9. A、D

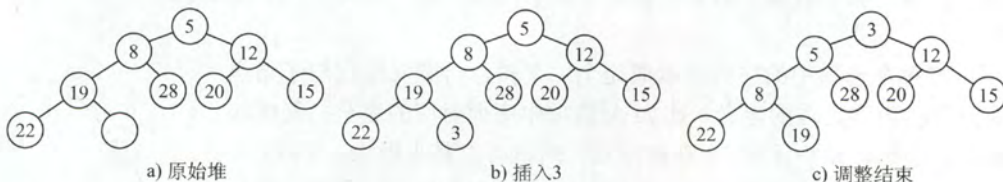
建堆过程中，向下调整的时间与树高 h 有关，为 $O(h)$ ，每次向下调整时，大部分结点的高度都较小。因此，可以证明在元素个数为 n 的序列上建堆，其时间复杂度为 $O(n)$ 。无论在最好情况还是在最坏情况下，堆排序的时间复杂度均为 $O(n \log_2 n)$ 。

10. D

筛选法初始建堆为{8, 17, 23, 52, 25, 72, 68, 71, 60}，输出 8 后重建的堆为{17, 25, 23, 52, 60, 72, 68, 71}，输出 17 后重建的堆为{23, 25, 68, 52, 60, 72, 71}。

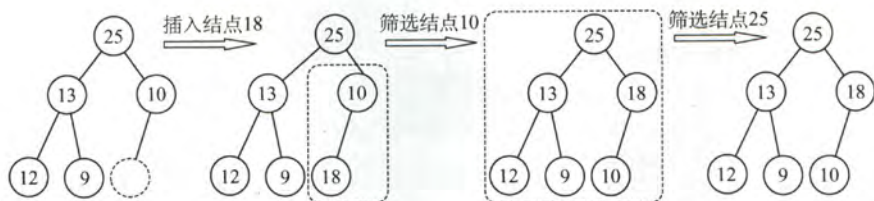
11. A

插入关键字 3 后堆变化过程如下：



12. B

首先 18 与 10 比较，交换位置，再与 25 比较，不交换位置。共比较了 2 次，调整的过程如下图所示。



13. A

选择排序算法的比较次数始终为 $n(n-1)/2$ ，与序列状态无关。

14. C

删除 8 后，将 12 移动到堆顶，第一次是 15 和 10 比较，第二次是 10 和 12 比较并交换，第三次还需比较 12 和 16，故比较次数为 3 次。

二、综合应用题

1. 解答：

以小根堆为例，堆的特点是双亲结点的关键字必然小于等于该孩子结点的关键字，而两个孩子结点的关键字没有次序规定。而二叉排序树中，每个双亲结点的关键字均大于左子树结点的关键字、均小于右子树结点的关键字，也就是说，每个双亲结点的左、右孩子关键字有次序关系。这样，当对两种树执行中序遍历后，二叉排序树会得到一个有序的序列，而堆则不一定能得到一个有序的序列。

2. 解答：

在基于比较的排序方法中：

插入排序、快速排序和归并排序，只有在将元素全部排完序后，才能得到前 k 小的元素序列，算法的效率不高。

冒泡排序、堆排序和简单选择排序可以，因为它们在一趟中都可以确定一个最小的元素。采用堆排序最合适，对于 n 个元素的序列，建立初始堆的时间不超过 $4n$ ，取得第 k 个最小元素之前的排序序列所花时间为 $k\log_2 n$ ，总时间为 $4n+k\log_2 n$ ，冒泡和简单选择排序完成此功能所花时间为 kn ，当 $k \geq 5$ 时，通过比较可以得出堆排序最优。

注意：从本题可以得出结论，只需得到前 k 小元素的顺序排列可采用的排序算法有冒泡排序、堆排序和简单选择排序。

3. 解答：

将 K_{n+1} 插入数组的第 $n+1$ 个位置（即作为一个树叶插入），然后将其与双亲比较，若它大于其双亲则停止调整，否则将 K_{n+1} 与其双亲交换，重复的将 K_{n+1} 与其新的双亲比较，算法终止于 K_{n+1} 大于等于其双亲或 K_{n+1} 本身已上升为根。

4. 解答：

算法的思想是：每趟在原始链表中摘下关键字最大的结点，把它插入到结果链表的最前端。由于在原始链表中摘下的关键字越来越小，在结果链表前端插入的关键字也越来越小，最后形成的结果链表中的结点将按关键字非递减的顺序有序链接。

单链表的定义如第 2 章所述，假设它不带表头结点。

```
void selectSort(LinkedList& L){
    //对不带表头结点的单链表 L 执行简单选择排序
    LinkNode *h=L,*p,*q,*r,*s;
    L=NULL;
    while(h!=NULL){
        //持续扫描原链表
```



```

p=s=h;q=r=NULL;
//指针 s 和 r 记忆最大结点和其前驱; p 为工作指针, q 为其前驱
while(p!=NULL){ //扫描原链表寻找最大结点 s
    if(p->data>s->data){s=p;r=q;} //找到更大的, 记忆它和它的前驱
    q=p;p=p->link; //继续寻找
}
if(s==h)
    h=h->link; //最大结点在原链表前端
else
    r->link=s->link; //最大结点在原链表表内
s->link=L;L=s; //结点 s 插入到结果链前端
}
}

```

5. 解答:

将顺序表 $L[1 \cdots n]$ 看成一个完全二叉树, 扫描所有分支结点, 当遇到孩子结点的关键字小于根结点关键字时返回 false, 扫描完后返回 true。算法的实现如下:

```

bool IsMinHeap(ElemType A[],int len){
    if(len%2==0){ //len 为偶数, 有一个单分支结点
        if(A[len/2]>A[len]) //判断单分支结点
            return false;
        for(i=len/2-1;i>=1;i--) //判断所有双分支结点
            if(A[i]>A[2*i]||A[i]>A[2*i+1])
                return false;
    }
    else{ //len 为奇数时, 没有单分支结点
        for(i=len/2;i>=1;i--) //判断所有双分支结点
            if(A[i]>A[2*i]||A[i]>A[2*i+1])
                return false;
    }
    return true;
}

```

7.5 归并排序和基数排序

7.5.1 归并排序

归并排序与上述基于交换、选择等排序的思想不一样,“归并”的含义是将两个或两个以上的有序表组合成一个新的有序表。假定待排序表含有 n 个记录,则可以看成是 n 个有序的子表,每个子表长度为 1,然后两两归并,得到 $\lceil n/2 \rceil$ 个长度为 2 或 1 的有序表;再两两归并,……如此重复,直到合并成一个长度为 n 的有序表为止,这种排序方法称为 2-路归并排序。如图 7-4 所示为 2 路归并排序的一个例子。

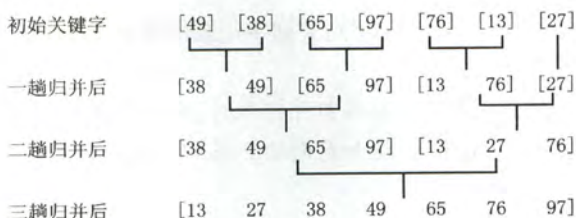


图 7-4 2-路归并排序示例

Merge()的功能是将前后相邻的两个有序表归并为一个有序表的算法。设两段有序表 $A[\text{low} \cdots \text{mid}]$ 、 $A[\text{mid}+1 \cdots \text{high}]$ 存放在同一顺序表中相邻的位置上，先将它们复制到辅助数组 B 中。每次从对应 B 中的两个段取出一个记录进行关键字的比较，将较小者放入 A 中，当数组 B 中有一段的下标超出其对应的表长时（即该段的所有元素已经完全复制到 A 中），将另一段中的剩余部分直接复制到 A 中。算法如下：

```
ElemType *B=(ElemType *)malloc((n+1)*sizeof(ElemType)); //辅助数组 B
void Merge(ElemType A[],int low,int mid,int high){
//表 A 的两段 A[low...mid]和 A[mid+1...high]各自有序，将它们合并成一个有序表
for(int k=low;k<=high;k++){
    B[k]=A[k]; //将 A 中所有元素复制到 B 中
}
for(i=low,j=mid+1,k=i;i<=mid&& j<=high;k++){
    if(B[i]<=B[j]) //比较 B 的左右两段中的元素
        A[k]=B[i++]; //将较小值复制到 A 中
    else
        A[k]=B[j++];
}
//for
while(i<=mid) A[k++]=B[i++]; //若第一个表未检测完，复制
while(j<=high) A[k++]=B[j++]; //若第二个表未检测完，复制
}
```

注意：上面代码中最后两个 while 循环只有一个会执行。

一趟归并排序的操作是，调用 $\lceil n/2h \rceil$ 次算法 merge() 将 $L[1 \cdots n]$ 中前后相邻且长度为 h 的有序段进行两两归并，得到前后相邻、长度为 $2h$ 的有序段，整个归并排序需进行 $\lceil \log_2 n \rceil$ 趟。

递归形式的 2-路归并排序算法是基于分治的，其过程如下：

分解：将含有 n 个元素的待排序表分成各含 $n/2$ 个元素的子表，采用 2-路归并排序算法对两个子表递归地进行排序；

合并：合并两个已排序的子表得到排序结果。

```
void MergeSort(ElemType A[],int low,int high){
    if(low<high){
        int mid=(low+high)/2; //从中间划分两个子序列
        MergeSort(A,low,mid); //对左侧子序列进行递归排序
        MergeSort(A,mid+1,high); //对右侧子序列进行递归排序
        Merge(A,low,mid,high); //归并
    }
}
```

2-路归并排序算法的性能分析如下：

空间效率：Merge() 操作中，辅助空间刚好要占用 n 个单元，所以归并排序的空间复杂度为 $O(n)$ 。

时间效率：每一趟归并的时间复杂度为 $O(n)$ ，共需进行 $\lceil \log_2 n \rceil$ 趟归并，所以算法的时间复杂度为 $O(n \log_2 n)$ 。

稳定性: 由于 Merge() 操作不会改变相同关键字记录的相对次序, 所以 2-路归并排序算法是一个稳定的排序方法。

注意: 一般而言, 对于 N 个元素进行 k -路归并排序时, 排序的趟数 m 满足 $k^m = N$, 从而 $m = \log_k N$, 又考虑到 m 为整数, 所以 $m = \lceil \log_k N \rceil$ 。这和前面的 2-路归并是一致的。

7.5.2 基数排序

基数排序是一种很特别的排序方法, 它不是基于比较进行排序的, 而是采用多关键字排序思想 (即基于关键字各位的大小进行排序的), 借助“分配”和“收集”两种操作对单逻辑关键字进行排序。基数排序又分为最高位优先 (MSD) 排序和最低位优先 (LSD) 排序。

以 r 为基数的最低位优先基数排序的过程: 假设线性表由结点序列 a_0, a_1, \dots, a_{n-1} 构成, 每个结点 a_j 的关键字由 d 元组 $(k_j^{d-1}, k_j^{d-2}, \dots, k_j^1, k_j^0)$ 组成, 其中 $0 \leq k_j^i \leq r-1$ ($0 \leq j < n, 0 \leq i \leq d-1$)。在排序过程中, 使用 r 个队列 Q_0, Q_1, \dots, Q_{r-1} 。排序过程如下:

对 $i=0, 1, \dots, d-1$, 依次做一次“分配”和“收集” (其实就是一次稳定的排序过程)。

分配: 开始时, 把 Q_0, Q_1, \dots, Q_{r-1} 各个队列置成空队列, 然后依次考察线性表中的每一个结点 a_j ($j=0, 1, \dots, n-1$), 如果 a_j 的关键字 $k_j^i = k$, 就把 a_j 放进 Q_k 队列中。

收集: 把 Q_0, Q_1, \dots, Q_{r-1} 各个队列中的结点依次首尾相接, 得到新的结点序列, 从而组成新的线性表。

图 7-5 所示为基数排序作用于一个由 7 个 3 位数组成的表的过程。最左端为输入, 其余各列表示了对各个不断递增的有效位连续排序后的情况, 阴影表示当前正被排序的数位。

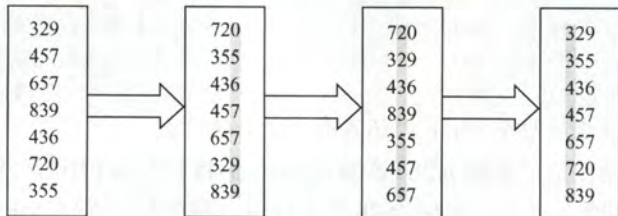


图 7-5 基数排序示意图

基数排序算法的性能分析如下:

空间效率: 一趟排序需要的辅助存储空间为 r (r 个队列), 但以后的排序中重复使用这些队列, 所以基数排序的空间复杂度为 $O(r)$ 。

时间效率: 基数排序需要进行 d 趟分配和收集, 一趟分配需要 $O(n)$, 一趟收集需要 $O(r)$, 所以基数排序的时间复杂度为 $O(d(n+r))$, 它与序列的初始状态无关。

稳定性: 对于基数排序算法而言, 很重要一点就是按位排序时必须是稳定的。因此, 这也保证了基数排序保持稳定性。

7.5.3 本节试题精选

一、单项选择题

- 以下排序方法中, () 在一趟结束后不一定能选出一个元素放在其最终位置上。
A. 简单选择排序 B. 冒泡排序 C. 归并排序 D. 堆排序
- 以下排序算法中, () 不需要进行关键字的比较。

- A. 快速排序 B. 归并排序 C. 基数排序 D. 堆排序
3. 在下列排序算法中, 平均情况下空间复杂度为 $O(n)$ 的是 (); 最坏情况下空间复杂度为 $O(n)$ 的是 ()。
- I. 希尔排序 II. 堆排序 III. 冒泡排序
IV. 归并排序 V. 快速排序 VI. 基数排序
- A. I、IV、VI B. II、V C. IV、V D. IV
4. 下列排序方法中, 排序过程中比较次数的数量级与序列初始状态无关的是 ()。
- A. 归并排序 B. 插入排序 C. 快速排序 D. 冒泡排序
5. 若对 27 个元素只进行三趟多路归并排序, 则选取的归并路数最少为 ()。
- A. 2 B. 3 C. 4 D. 5
6. 【2016 年计算机联考真题】
对 10TB 的数据文件进行排序, 应使用的方法是 ()。
- A. 希尔排序 B. 堆排序 C. 快速排序 D. 归并排序
7. 2-路归并排序中, 归并趟数的数量级是 ()。
- A. $O(n)$ B. $O(\log_2 n)$ C. $O(n \log_2 n)$ D. $O(n^2)$
8. 将两个各有 N 个元素的有序表合并成一个有序表, 最少的比较次数是 (), 最多的比较次数是 ()。
- A. N B. $2N-1$ C. $2N$ D. $N-1$
9. 【2017 年计算机联考真题】
在内部排序时, 若选择了归并排序而没有选择插入排序, 则可能的理由是 ()。
- I. 归并排序的程序代码更短 II. 归并排序的占用空间更少
III. 归并排序的运行效率更高
- A. 仅 II B. 仅 III C. 仅 I、II D. 仅 I、III
10. 一组经过第一趟 2-路归并排序后的记录的关键字为 {25, 50, 15, 35, 80, 85, 20, 40, 36, 70}, 其中包含 5 个长度为 2 的有序表, 用 2-路归并排序方法对该序列进行第二趟归并后的结果为 ()。
- A. 15, 25, 35, 50, 80, 20, 85, 40, 70, 36
B. 15, 25, 35, 50, 20, 40, 80, 85, 36, 70
C. 15, 25, 50, 35, 80, 85, 20, 36, 40, 70
D. 15, 25, 35, 50, 80, 20, 36, 40, 70, 85
11. 如果将中国人按照生日 (不考虑年份, 只考虑月、日) 来排序, 那么使用下列排序算法中最快的是 ()。
- A. 归并排序 B. 希尔排序
C. 快速排序 D. 基数排序
12. 对 {05, 46, 13, 55, 94, 17, 42} 进行基数排序, 一趟排序的结果是 ()。
- A. 05, 46, 13, 55, 94, 17, 42
B. 05, 13, 17, 42, 46, 55, 94
C. 42, 13, 94, 05, 55, 46, 17
D. 05, 13, 46, 55, 17, 42, 94

13. 【2013 年计算机联考真题】

对给定的关键字序列 110, 119, 007, 911, 114, 120, 122 进行基数排序, 则第 2 趟分配收集后得到的关键字序列是 ()。

- A. 007, 110, 119, 114, 911, 120, 122
- B. 007, 110, 119, 114, 911, 122, 120
- C. 007, 110, 911, 114, 119, 120, 122
- D. 110, 120, 911, 122, 114, 007, 119

二、综合应用题

1. 已知序列 {503, 87, 512, 61, 908, 170, 897, 275, 653, 462}, 采用二路归并排序法对该序列做升序排序时需要几趟排序? 并给出每一趟的结果。

2. 设待排序的排序码序列为 {12, 2, 16, 30, 28, 10, 16*, 20, 6, 18}, 试写出使用基数排序方法每趟排序后的结果。并说明做了多少次排序码比较。

7.5.4 答案与解析

一、单项选择题

1. C

前面我们知道插入排序不能保证在一趟结束后一定存在元素放在最终位置上。事实上, 归并排序也不能保证, 比如序列 {6, 5, 7, 8, 2, 1, 4, 3} 进行一趟 2-路归并排序 (从小到大) 后序列为 {5, 6, 7, 8, 1, 2, 3, 4}, 显然它们都没有被放在最终位置上。

2. C

基数排序是基于关键字各位的大小进行排序的, 它不是基于关键字的比较进行的。

3. D、C

归并排序算法在平均情况和最坏情况下空间复杂度都会达到 $O(n)$, 快速排序只有在最坏情况下才会达到 $O(n)$, 平均情况下为 $O(\log_2 n)$ 。所以归并排序算法可以看作是本章所有算法中占用辅助空间最多的排序算法。

4. A

前面已讲过选择排序的比较次数与序列初始状态无关, 归并排序也与序列的初始状态无关, 读者还应能从算法的原理方面来考虑为什么和初始状态无关。

5. B

利用上述公式, 这里要求的是 k , 代入可得 $k=3$ 。

6. D

外部排序指待排序文件较大, 内存一次性放不下, 需存放在外部介质中。外部排序通常采用归并排序法。选项 A、B、C 都是内部排序的方法。

7. B

对于 N 个元素进行 k -路归并排序时, 排序的趟数 m 满足 $k^m=N$, 所以 $m=\lceil \log_k N \rceil$, 本题中即为 $\lceil \log_2 n \rceil$ 。

8. A、B

注意到当一个表中最小元素比另一个表中最大元素还大的时候, 比较的次数是最少的, 仅比

较 N 次；而当两个表中元素依次间隔地比较时，即 $a_1 < b_1 < a_2 < b_2 < \dots < a_n < b_n$ 时，比较的次数是最多的，为 $2N-1$ 次。

完成该题后，读者应该对：章节 1.2.3 的第 6 题（链表排序的时间复杂度）、章节 4.5.4 的第 13 题（多个有序表合并成单个有序表在最坏情况下的比较次数）和章节 7.1.2 的第 4 题（基于比较的排序，求最少的比较次数）的思路的异同点有个系统的总结。一定要注意题眼中：1) 问的是“最少”还是“最多”，2) 是比较次数还是时间复杂度！

建议读者对此举一反三：若将本题中的两个有序表的长度分别设为 M 和 N ，那么最多（或最少）的比较次数是多少？时间复杂度又是多少？以上的其他题目类似。

举一反三才可触类旁通，总结提炼方能游刃有余。

9. B

归并排序代码比选择插入排序更复杂，前者空间复杂度是 $O(n)$ ，后者是 $O(1)$ 。但是前者时间复杂度是 $O(n \log n)$ ，后者是 $O(n^2)$ 。所以选项 B 正确。

10. B

由于这里是采用 2-路归并，而且是第二趟排序，所以每 4 个元素放在一起归并，可将序列划分为 $\{25, 50, 15, 35\}$ ， $\{80, 85, 20, 40\}$ 和 $\{36, 70\}$ ，分别对它们进行排序为 $\{15, 25, 35, 50\}$ ， $\{20, 40, 80, 85\}$ 和 $\{36, 70\}$ ，故而在 B。

11. D

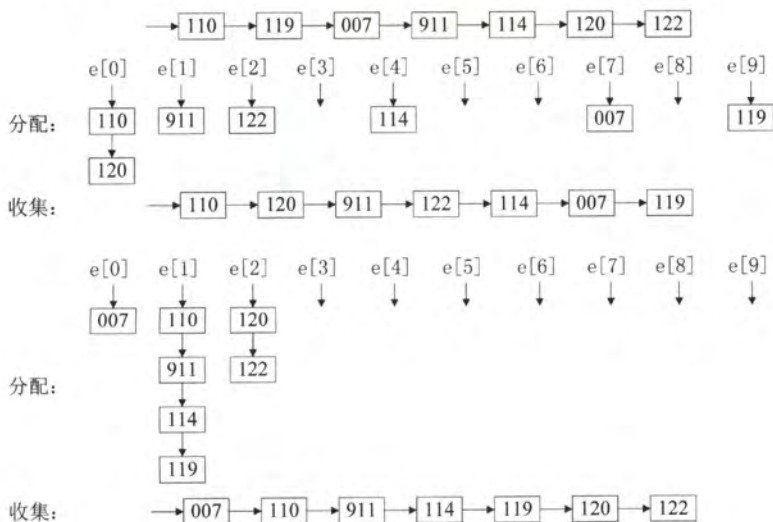
按照所有中国人的生日排序，一方面 N 是非常大的，另一方面关键字所含排序码数为 2，且一个排序码基数为 12，另一个为 31，都是较小的常数值，采用基数排序可以在 $O(N)$ 内完成排序过程。

12. C

比较各数的个位数，按各数的个位数进行排序，可知选 C。

13. C

基数排序的第 1 趟排序是按照个位数字的大小来排序的，第 2 趟排序是按照十位数字的大小进行排序的，排序的过程如下图所示。



二、综合应用题

1. 解答:

$n=10$, 需要排序的趟数= $\lceil \log_2 10 \rceil = 4$, 各趟的排序结果如下:

初始序列: 503, 87, 512, 61, 908, 170, 897, 275, 653, 462

第一趟: 87, 503, 61, 512, 170, 908, 275, 897, 462, 653 (长度为 2)

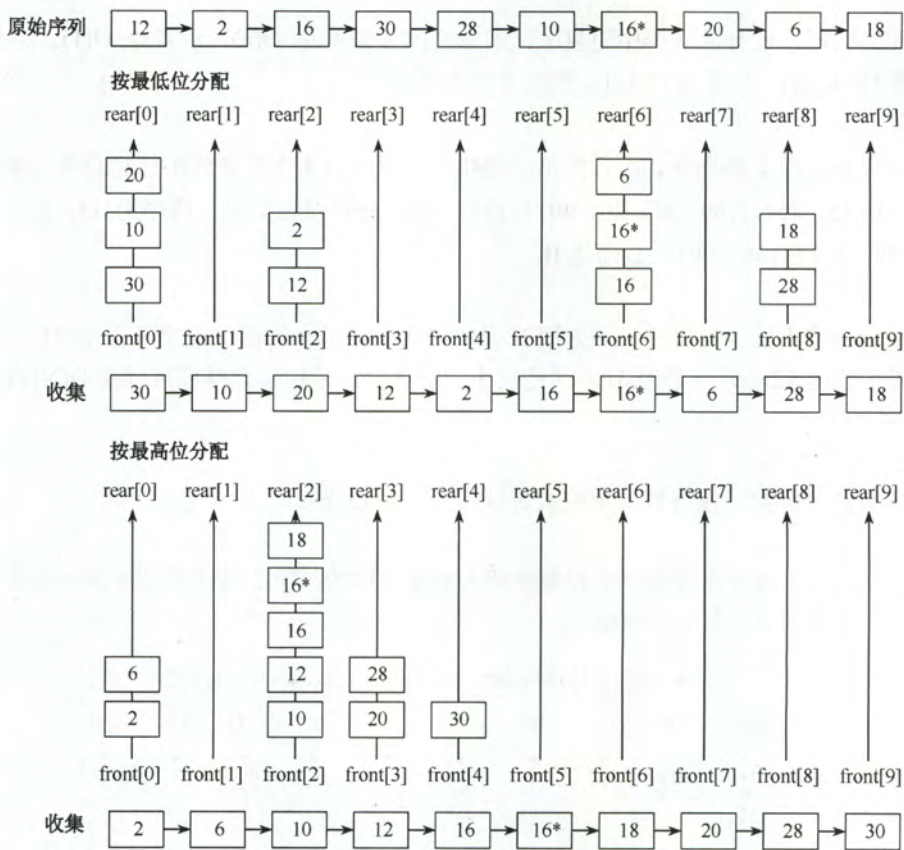
第二趟: 61, 87, 503, 512, 170, 275, 897, 908, 462, 653 (长度为 4)

第三趟: 61, 87, 170, 275, 503, 512, 897, 908, 462, 653 (长度为 8)

第四趟: 61, 87, 170, 275, 462, 503, 512, 653, 897, 908 (长度为 10)

2. 解答:

使用链式队列的基数排序的排序过程如下图所示。



需要通过 2 次“分配”和“收集”完成排序。

7.6 各种内部排序算法的比较及应用

7.6.1 内部排序算法的比较

前面讨论的排序算法有很多, 对各算法的比较是考研中必考的内容。一般基于三个因素进行对比: 时空复杂度、算法的稳定性、算法的过程特征。

从时间复杂度来看, 简单选择排序、直接插入排序和冒泡排序的平均情况下的时间复杂度都为

$O(n^2)$ ，并且实现过程也较简单，但直接插入排序和冒泡排序在最好的情况下时间复杂度可以达到 $O(n)$ ，而简单选择排序则与序列的初始状态无关。希尔排序作为插入排序的拓展，对较大规模的排序都可以达到很高的效率，但是目前未得出其精确的渐近时间。堆排序是利用了一种称为堆的数据结构，可以在线性时间内完成建堆，并且在 $O(n\log_2 n)$ 内完成排序过程。快速排序是基于分治的思想，虽然最坏情况下快速排序时间会达到 $O(n^2)$ ，但快速排序平均性能可以达到 $O(n\log_2 n)$ ，在实际应用中常常优于其他排序算法。归并排序同样是基于分治的思想，但由于其分割子序列与初始序列的排列无关，因此它的最好、最坏和平均时间复杂度均为 $O(n\log_2 n)$ 。

从空间复杂度来看，简单选择排序、插入排序、冒泡排序、希尔排序和堆排序都仅需要借助常数个辅助空间。快速排序在空间上只使用一个小的辅助栈，用于实现递归，平均情况下大小为 $O(\log_2 n)$ ，当然在最坏的情况下可能会增长到 $O(n)$ 。二路归并排序在合并操作中需要借助较多的辅助空间用于元素复制，大小为 $O(n)$ ，虽然有方法可以克服这个缺点，但是其代价是算法会很复杂而且时间复杂度会增加。

从稳定性看，插入排序、冒泡排序、归并排序和基数排序是稳定的排序方法，而简单选择排序、快速排序、希尔排序和堆排序都是不稳定的排序方法。对排序方法的稳定性，读者应从算法本身的原理上去理解，而不应拘泥于死记硬背。

从过程特征看，采用不同的排序算法，在一次循环或几次循环后的排序结果可能是不同的，考研题中经常出现给出一个待排序的初始序列和已经部分排序的序列，问其采用何种排序算法。这就要对各类排序算法的过程特征十分熟悉，如冒泡排序和堆排序在每次循环后都能产生当前的最大值或最小值，而快速排序一次循环就确定一个元素的最终位置等。

下表列出了各种排序算法的时间复杂度和空间复杂度和稳定性情况。其中空间复杂度仅列举了平均情况下的复杂度，由于希尔排序的时间复杂度依赖于增量函数，所以这里无法准确地给出其时间复杂度。

表 7-1 各种排序算法的性质

算法种类	时间复杂度			空间复杂度	是否稳定
	最好情况	平均情况	最坏情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔排序				$O(1)$	否
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	否
2-路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	是
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	是

7.6.2 内部排序算法的应用

通常情况下，对排序算法的比较和应用应考虑以下情况：

(1) 选取排序方法需要考虑的因素：

- ① 待排序的元素数目 n 。
- ② 元素本身信息量的大小。
- ③ 关键字的结构及其分布情况。

④ 稳定性的要求。

⑤ 语言工具的条件, 存储结构及辅助空间的大小等。

(2) 排序算法小结:

① 若 n 较小 ($n \leq 50$), 则可以采用直接插入排序或简单选择排序。由于直接插入排序所需的记录移动操作较简单选择排序多, 因而当记录本身信息量较大时, 用简单选择排序较好。

② 若文件的初始状态已按关键字基本有序, 则选用直接插入或冒泡排序为宜。

③ 若 n 较大, 则应采用时间复杂度为 $O(n\log_2 n)$ 的排序方法: 快速排序、堆排序或归并排序。快速排序被认为是目前基于比较的内部排序法中最好的方法, 当待排序的关键字是随机分布时, 快速排序的平均时间最短。堆排序所需的辅助空间少于快速排序, 并且不会出现快速排序可能出现的最坏情况, 这两种排序都是不稳定的。若要求排序稳定且时间复杂度为 $O(n\log_2 n)$, 则可选用归并排序。但本章介绍的从单个记录起进行两两归并的排序算法并不值得提倡, 通常可以将它和直接插入排序结合在一起使用。先利用直接插入排序求得较长的有序子文件, 然后再两两归并。直接插入排序是稳定的, 因此改进后的归并排序仍是稳定的。

④ 在基于比较的排序方法中, 每次比较两个关键字的大小之后, 仅仅出现两种可能的转移, 因此可以用一棵二叉树来描述比较判定过程, 由此可以证明: 当文件的 n 个关键字随机分布时, 任何借助于“比较”的排序算法, 至少需要 $O(n\log_2 n)$ 的时间。

⑤ 若 n 很大, 记录的关键字位数较少且可以分解时, 采用基数排序较好。

⑥ 当记录本身信息量较大时, 为避免耗费大量时间移动记录, 可用链表作为存储结构。

7.6.3 本节试题精选

一、单项选择题

1. 若要求排序是稳定的, 且关键字为实数, 则在下列排序方法中应选 ()。

A. 直接插入排序 B. 选择排序 C. 基数排序 D. 快速排序

2. 以下排序方法中时间复杂度为 $O(n\log_2 n)$ 且稳定的是 ()。

A. 堆排序 B. 快速排序 C. 归并排序 D. 直接插入排序

3. 设被排序的结点序列共有 N 个结点, 在该序列中的结点已十分接近有序的情况下, 用直接插入排序、归并排序和快速排序对其进行排序, 这些算法的时间复杂度应为 ()。

A. $O(N)$, $O(N)$, $O(N)$ B. $O(N)$, $O(N*\log_2 N)$, $O(N*\log_2 N)$

C. $O(N)$, $O(N*\log_2 N)$, $O(N^2)$ D. $O(N^2)$, $O(N*\log_2 N)$, $O(N^2)$

4. 下列排序算法中属于稳定排序的是 (①), 平均时间复杂度为 $O(n\log_2 n)$ 的是 (②), 在最好的情况下, 时间复杂度可以达到线性时间的有 (③)。

I. 冒泡排序 II. 堆排序 III. 选择排序 IV. 直接插入排序

V. 希尔排序 VI. 归并排序 VII. 快速排序

5. 就排序算法所用的辅助空间而言, 堆排序、快速排序和归并排序的关系是 ()。

A. 堆排序 < 快速排序 < 归并排序 B. 堆排序 < 归并排序 < 快速排序

C. 堆排序 > 归并排序 > 快速排序 D. 堆排序 > 快速排序 > 归并排序

6. 排序趟数与序列的原始状态无关的排序方法是 ()。

I. 直接插入排序 II. 简单选择排序 III. 冒泡排序 IV. 基数排序

A. I、III B. I、II、IV C. I、II、III D. I、IV

7. 【2012年计算机联考真题】

在内部排序过程中,对尚未确定最终位置的所有元素进行一遍处理称为一趟排序。下列排序方法中,每一趟排序结束都至少能够确定一个元素最终位置的方法是()。

- I. 简单选择排序 II. 希尔排序 III. 快速排序
IV. 堆排序 V. 二路归并排序

- A. 仅 I、III、IV B. 仅 I、III、V
C. 仅 II、III、IV D. 仅 III、IV、V

8. 若序列的原始状态为{1, 2, 3, 4, 5, 10, 6, 7, 8, 9}, 要想使得排序过程中元素比较次数最少, 则应该采用()方法。

- A. 插入排序 B. 选择排序 C. 希尔排序 D. 冒泡排序

9. 一般情况下, 以下查找效率最低的数据结构是()。

- A. 有序顺序表 B. 二叉排序树 C. 堆 D. 平衡二叉树

10. 【2015 年计算机联考真题】

下列排序算法中, 元素的移动次数与关键字的初始排列次序无关的是()。

- A. 直接插入排序 B. 起泡排序 C. 基数排序 D. 快速排序

11. 【2017 年计算机联考真题】

下列排序方法中, 若将顺序存储更换为链式存储, 则算法的时间效率会降低的是()。

- I. 插入排序 II. 选择排序 III. 起泡排序 IV. 希尔排序 V. 堆排序
A. 仅 I、II B. 仅 II、III C. 仅 III、IV D. 仅 IV、V

二、综合应用题

1. 设关键字序列为{3, 7, 6, 9, 7, 1, 4, 5, 20}, 对其进行排序的最小交换次数是多少次?

2. 设顺序表用数组 A[] 表示, 表中元素存储在数组下标 1~m+n 的范围内, 前 m 个元素递增有序, 后 n 个元素递增有序, 设计一个算法, 使得整个顺序表有序。

1) 给出算法的基本设计思想。

2) 根据设计思想, 采用 C/C++ 描述算法, 关键之处给出注释。

3) 说明你所设计算法的时间复杂度与空间复杂度。

3. 有一种简单的排序算法, 叫做计数排序(count sorting)。这种排序算法对一个待排序的表(用数组表示)进行排序, 并将排序结果存放到另一个新的表中。必须注意的是, 表中所有待排序的关键码互不相同, 计数排序算法针对表中的每个记录, 扫描待排序的表一趟, 统计表中有多少个记录的关键码比该记录的关键码小, 假设针对某一个记录, 统计出的计数值为 c, 那么, 这个记录在新的有序表中的合适的存放位置即为 c。

1) 设计实现计数排序的算法。

2) 对于有 n 个记录的表, 关键码比较次数是多少?

3) 与简单选择排序相比较, 这种方法是否更好? 为什么?

7.6.4 答案与解析

一、单项选择题

1. A

采用排除法。由于题目要求是稳定排序, 排除 B 和 D, 又由于基数排序不能对 float 和 double 类型的实数进行排序, 故排除 C。

2. C

堆排序和快速排序不是稳定排序方法, 而直接插入排序算法时间复杂度为 $O(n^2)$ 。

3. C

读者应熟练掌握各种排序算法的时空复杂度, 稳定性等, 见表 7-1。

4. ① I、IV、VI ② II、VI、VII ③ I、IV

读者应能从算法的原理上理解算法的稳定性情况。堆排序和归并排序在最坏情况下的时间复杂度与最好情况下时间复杂度是同一个数量级, 都是 $O(n\log_2 n)$ 。

5. A

由于堆排序空间复杂度为 $O(1)$, 快速排序空间复杂度在最坏情况下为 $O(n)$, 平均为 $O(\log_2 n)$, 归并排序空间复杂度为 $O(n)$, 所以不难得出选 A。

6. B

交换类的排序, 其趟数和原始序列状态有关, 故冒泡排序与初始序列有关。直接插入排序: 每一趟排序都是插入一个元素, 所以排序趟数固定为 $n-1$; 简单选择排序: 每趟排序都是选出一个最小(或最大)的元素, 所以排序趟数固定为 $n-1$; 基数排序: 每趟排序都要进行“分配”和“收集”, 排序趟数固定为 d 。

7. A

对于 I, 简单选择排序每次选择未排序列中的最小元素放入其最终位置。对于 II, 希尔排序每次是对划分的子表进行排序, 得到局部有序的结果, 所以不能保证每一趟排序结束都能确定一个元素的最终位置。对于 III, 快速排序每一趟排序结束后都将枢轴元素放到最终位置。对于 IV, 堆排序属于选择排序, 每次都都将大根堆的根结点与表尾结点交换, 确定其最终位置。对于 V, 二路归并排序每趟对子表进行两两归并从而得到若干个局部有序的结果, 但无法确定最终位置。

8. A

选择排序和序列初态无关, 直接排除。初始序列基本有序时, 插入排序比较次数较少。本题中, 插入排序仅需比较 $n+4$ 次, 而希尔排序和冒泡排序均远大于此。

9. C

堆是用于排序的, 在查找时它是无序的, 所以效率没有其他的查找结构效率高。

10. C

基数排序的元素移动次数与关键字的初始排列次序无关, 而其他三种排序都是与关键字的初始排列明显相关的。

11. D

插入排序、选择排序、起泡排序原本时间复杂度是 $O(n^2)$, 更换为链式存储后的时间复杂度还是 $O(n^2)$ 。希尔排序和堆排序都利用了顺序存储的随机访问特性, 而链式存储不支持这种性质, 所以时间复杂度会增加, 因此选 D。

二、综合应用题

1. 解答:

由于关键字序列数较小, 采用直接插入排序或简单选择排序, 直接插入排序的交换次数更多, 选择简单选择排序:

初始序列 3, 7, 6, 9, 7, 1, 4, 5, 20

第一次: 1, 7, 6, 9, 7, 3, 4, 5, 20 交换 1, 3

第二次: 1, 3, 6, 9, 7, 7, 4, 5, 20 交换 3, 7

第三次: 1, 3, 4, 9, 7, 7, 6, 5, 20 交换 4, 6

第四次: 1, 3, 4, 5, 7, 7, 6, 9, 20 交换 5, 9

第五次: 1, 3, 4, 5, 6, 7, 7, 9, 20 交换 6, 7

所以最小交换次数为 5。(注意, 这里是交换次数, 不是移动次数或比较次数)

2. 解答:

1) 算法的基本设计思想:

将数组 $A[1 \cdots m+n]$ 看作一个已经经过了 m 趟插入排序的表, 则从 $m+1$ 趟开始, 将后 n 个元素依次插入前面的有序表中。

2) 算法的实现如下:

```
void Insert_Sort(ElemType A[], int m, int n) {
    int i, j;
    for (i = m + 1; i <= m + n; i++) {           // 依次将 A[m+1...m+n] 插入到有序表中
        A[0] = A[i];                             // 复制为哨兵
        for (j = i - 1; A[j] > A[0]; j--)         // 从后往前插入
            A[j + 1] = A[j];                     // 元素后移
        A[j + 1] = A[0];                         // 插入
    }
}
```

3) 时间复杂度由 m 和 n 共同决定, 从上面的算法不难看出, 在最坏情况下元素的比较次数为 $O(mn)$, 而元素移动的次数为 $O(mn)$, 所以时间复杂度为 $O(mn)$ 。

由于算法只用到了常数个辅助空间, 所以空间复杂度为 $O(1)$ 。

此外, 本题也可以采用归并排序, 将 $A[1 \cdots m]$ 和 $A[m+1 \cdots m+n]$ 看做两个待归并的有序子序列, 算法的时间复杂度为 $O(m+n)$, 空间复杂度为 $O(m+n)$ 。

3. 解答:

1) 算法的思想: 对每一个元素, 统计关键字比它小的元素个数, 然后把它放入到另一个数组中对应的位置上。

算法的实现如下:

```
void CountSort(RecType A[], RecType B[], int n) {
    // 计数排序算法, 将 A 中记录排序放入 B 中
    int cnt;                               // 计数变量
    for (i = 0; i < n; i++) {              // 对每个元素
        for (j = 0, cnt = 0; j < n; j++)
            if (A[j].key < A[i].key)
                cnt++;                     // 统计关键字比它小的元素个数
        B[cnt] = A[i];                    // 放入到对应的位置上
    }
}
```

2) 对于有 n 个记录的表, 每个关键码都要与 n 个记录 (含自身) 进行比较, 因此关键码的比较次数为 n^2 。

3) 简单选择排序算法比本算法好。简单选择排序比较次数是 $n(n-1)/2$, 且只用一个交换记录的空间; 而这种方法比较次数是 n^2 , 且需要另一数组空间。另外, 因题目要求“针对表中的每个记录, 扫描待排序的表一趟”, 所以比较次数是 n^2 。若限制“对任意两个记录之间只进行一次比较”, 则可把以上算法中的比较语句改为:

```
for (i = 0; i < n; i++)
    a[i].count = 0;                       // 各元素再增加一个计数域, 初始化为 0
for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++) {
        if (a[i].key < a[j].key)
            a[j].count++;
    }
```



```

else
    a[i].count++;
}

```

7.7 外部排序

外部排序可能会考查相关概念、方法和排序过程，外部排序的算法比较复杂，不会在算法设计上进行考查。本节的主要内容有：

- ① 外部排序指待排序文件较大，内存一次放不下，需存放在外部介质的文件的排序。
- ② 为减少平衡归并中外存读写次数所采取的方法：增大归并路数和减少归并段个数。
- ③ 利用败者树增大归并路数。
- ④ 利用置换-选择排序增大归并段长度来减少归并段个数。
- ⑤ 由长度不等的归并段，进行多路平衡归并，需要构造最佳归并树。

7.7.1 外部排序的基本概念

前面介绍过的排序方法都是在内存中进行的（称为内部排序）。而在许多实际应用中，经常需要对大文件进行排序，因为文件中的记录很多、信息量庞大，无法将整个文件拷贝进内存中进行排序。因此，需要将待排序的记录存储在外存上，排序时再把数据一部分一部分的调入内存进行排序。在排序过程中需要多次进行内存和外存之间的交换，对外存文件中的记录进行排序后的结果仍然被放到原有文件中。这种排序方法就称为**外部排序**。

7.7.2 外部排序的方法

在实际应用中，由于外存设备不同，通常又可分为磁盘文件排序和磁带文件排序两大类。磁带排序和磁盘排序的基本步骤相类似，主要的不同之处在于初始归并段在外存介质中的分布方式，磁盘是直接存取设备，磁带是顺序存取设备。下面以磁盘为例进行说明。

文件通常是按块存储在磁盘上的，操作系统也是按块对磁盘上的信息进行读写的。因为磁盘读/写的机械动作所需时间远远超过内存运算的时间（相比而言，可以忽略不计）。因此，在外部排序过程中的时间代价主要考虑访问磁盘的次数，即 I/O 次数。

外部排序通常采用归并排序方法。它包括两个相对独立的阶段：首先，根据内存缓冲区的大小，将外存上含 n 个记录的文件分成若干长度为 h 的子文件，依次读入内存并利用有效的内部排序方法对它们进行排序，并将排序后得到的有序子文件重新写回外存，通常称这些有序子文件为归并段或顺串；然后，对这些归并段进行逐趟归并，使归并段（有序的子文件）逐渐由小到大，直至得到整个有序文件为止。

例如，一个含有 2000 个记录的文件，每个磁盘块可容纳 250 个记录，则该文件包含 8 个磁盘块。然后对该文件作二路归并排序，每次往内存读入两个磁盘块，排序后再写回磁盘。若把内存工作区等分为 3 个缓冲区，如图 7-6 所示。其中的两个为输入缓冲区，一个为输出缓冲区，可以在内存中利用 7.5 节中的简单二路归并 merge 函数实现二路归并。

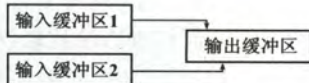


图 7-6 二路归并

首先，从参加归并排序的两个输入归并段 R_1 和 R_2 中分别读入一个块，放在输入缓冲区 1 和输入缓冲区 2 中。然后，在内存中进行二路归并，归并出来的对象顺序存放在输出缓冲区中。若输出缓冲区中对象存满，则将其内的对象顺序写到输出归并段 (R_1') 中，再将该输出缓冲区清

空,继续存放归并后的对象。若某一个输入缓冲区中的对象取空,则从对应的输入归并段中再读取下一块(这种情况在第一趟归并时不会出现),继续参加归并。如此继续,直到两个输入归并段中对象全部读入内存并都归并完成为止。当

R1 和 R2 归并完后,再归并 R3 和 R4、R5 和 R6、最后归并 R7 和 R8,这算作一趟归并。再把上趟的结果 R1'、R2'、R3'和 R4'两两归并,这又是一趟归并。最后把 R1''和 R2''两个归并段归并,结果得到最终的有序文件,一共进行了 3 趟归并,排序过程如图 7-7 所示。

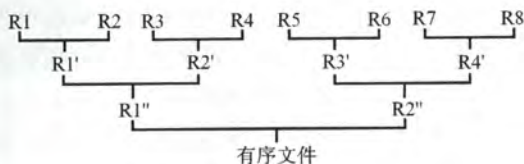


图 7-7 二路平衡归并的排序过程

在外部排序中实现两两归并时,不仅要调用 7.5 节中的 merge 过程,而且要进行外存的读/写,由于不可能将两个有序段及归并结果段同时存放在内存中,需要不停地将数据读出、写入磁盘,这将耗费大量的时间。一般情况下:

外排序的总时间=内部排序所需的时间+外存信息读写的时间+内部归并所需的时间

即, $t_{ES}=r \times t_{IS}+d \times t_{IO}+S \times (n-1) \times t_{mg}$

其中, r 是初始归并段个数, t_{IS} 是对每一个初始归并段进行内部排序的时间, d 是访问外存块的次数, t_{IO} 是每一个块的存取时间, S 是归并趟数, n 是每趟参加二路归并的记录个数, t_{mg} 是每作一次内部归并,取得一个关键字最小记录的时间。显然磁盘存取的时间远远大于内部排序和内部归并的时间,因此要提高外排序的速度,应着力减少 d , 即 I/O 次数。

由于外存上信息的读/写是以“物理块”为单位的,且每个物理块可容纳 250 个记录,可知每一趟归并需进行 8 次“读”和 8 次“写”,3 趟归并加上内部排序时所需进行的读/写使得在外排中总共需进行 $16 \times 4=64$ 次的读写。故上述二路平衡归并排序的总时间为:

$$8 \times t_{IS}+64 \times t_{IO}+3 \times 2000 t_{mg}$$

对于上例,若采用四路归并排序则只需要 2 趟归并,外排时总的读/写次数便减至 $2 \times 16+16=48$ 。因此,增大归并路数,可减少归并趟数,从而减少总的磁盘 I/O 次数,如图 7-8 所示。

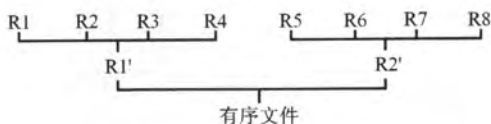


图 7-8 四路平衡归并的排序过程

一般地,对 r 个初始归并段,作 m 路平衡归并,归并树可用严格 m 叉树(即只有度为 m 与度为 0 的结点的 m 叉树)来表示。第一趟可将 r 个初始归并段归并为 $\lceil r/m \rceil$ 个归并段,以后每一趟归并将 ℓ 个归并段归并成 $\lceil \ell/m \rceil$ 个归并段,直到最后形成一个大的归并段为止。树的高度 $= \lceil \log_m r \rceil =$ 归并趟数 S 。可见,只要增大归并路数 m ,或减少初始归并段个数 r ,都能减少归并趟数 S ,以减少读写磁盘次数 d ,达到提高外部排序速度的目的。

7.7.3 多路平衡归并与败者树

在上节讨论过,归并趟数 $S=\lceil \log_m r \rceil$ 。从而增加归并路数 m 可以减少归并趟数 S ,进而减少访问外存的次数(I/O 次数)。然而,当增加归并路数 m 时,内部归并的时间将增加。作内部归并时,在 m 个元素中选择关键字最小的记录需要比较 $m-1$ 次。每趟归并 n 个元素需要作 $(n-1) \times (m-1)$ 次比较, S 趟归并总共需要的比较次数为:

$$S \times (n-1) \times (m-1) = \lceil \log_m r \rceil \times (n-1) \times (m-1) = \lceil \log_2 r \rceil \times (n-1) \times (m-1) / \lceil \log_2 m \rceil$$

其中的 $\lceil \log_2 r \rceil \times (n-1)$ 在初始归并段个数 r 与记录个数 n 一定时是常数。而 $(m-1)\lceil \log_2 m \rceil$ 随 m 增长而增长, 则内部归并时间亦随 m 的增长而增长。这将抵消由于增大 m 而减少外存访问次数所得到的效益。因此, 不能使用普通的内部归并排序算法。

为了使内部归并不受 m 的增大的影响, 引入了败者树。败者树是对树形选择排序的一种变形, 可以看作一棵完全二叉树。每个叶结点存放各归并段在归并过程中当前参加比较的记录, 内部结点用来记忆左右子树中的“失败者”, 而让胜者往上继续进行比较, 一直到根结点。如果比较两个数, 大的为失败者、小的为胜利者, 则根结点指向的数为最小数。

如图 7-9 (a), b_3 与 b_4 比较, b_4 是败者, 因此将段号 4 写入父结点 ls_4 。 b_1 与 b_2 比较, b_2 是败者, 将段号 2 写入 ls_3 。 b_3 与 b_4 的胜者 b_3 与 b_0 比较, b_0 是败者, 将段号 0 写入 ls_2 。最后两个胜者 b_3 与 b_1 比较, b_1 是败者, 段号写入 ls_1 。而将胜者 b_3 的段号写入 ls_0 。此时, 根结点 ls_0 所指的段的关键字最小。 b_3 中的 6 输出后, 将下一关键字填入 b_3 , 继续比较。

因为 m 路归并的败者树深度为 $\lceil \log_2 m \rceil$, 因此 m 个记录中选择最小关键字, 最多需要 $\lceil \log_2 m \rceil$ 次比较。所以总的比较次数为:

$$S \times (n-1) \times \lceil \log_2 m \rceil = \lceil \log_m r \rceil \times (n-1) \times \lceil \log_2 m \rceil = (n-1) \times \lceil \log_2 r \rceil$$

可见, 使用败者树后, 内部归并的比较次数与 m 无关了。因此, 只要内存空间允许, 增大归并路数 m 将有效地减少归并树的高度, 从而减少 I/O 次数 d , 提高外部排序的速度。

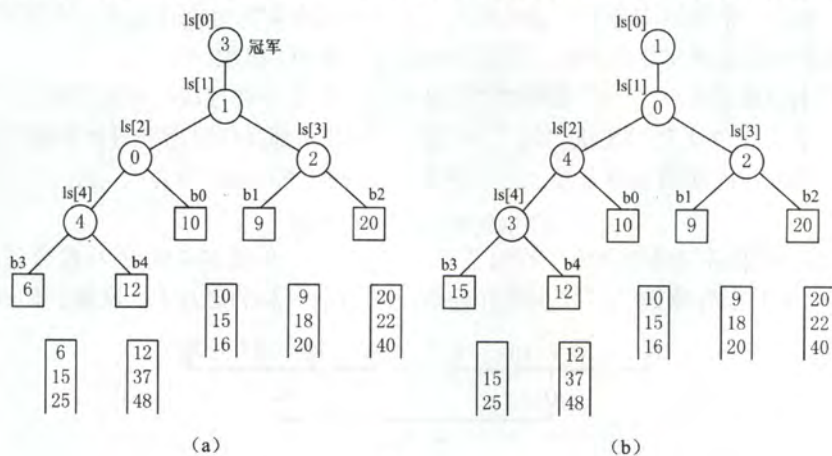


图 7-9 实现 5-路归并的败者树

值得说明的是, 归并路数 m 的选择并不是越大越好。归并路数 m 增大时, 相应地需要增加输入缓冲区个数。如果可供使用的内存空间不变, 势必要减少每个输入缓冲区的容量, 使得内外存交换数据的次数增大。当 m 值过大时, 虽然归并趟数会减少, 但读写外存的次数仍会增加。

7.7.4 置换-选择排序 (生成初始归并段)

上节讨论如何使用 m 路归并来减少磁盘访问次数。从第 2 节的讨论可知, 减少初始归并段个数 r 也可以减少归并趟数 S 。若总的记录个数为 n , 每个归并段的长度为 ℓ , 则归并段的个数 $m = \lceil n/\ell \rceil$ 。如果采用前面介绍过的内部排序方法, 将得到长度都相同的初始归并段。因此, 必须探索新的算法来生成初始归并段, 这就是本节要介绍的置换-选择算法。

设初始待排文件 FI , 初始归并段文件为 FO , 内存工作区为 WA , 内存工作区可容纳 w 个记录。置换-选择算法的步骤如下:

1) 从待排文件 FI 输入 w 个记录到工作区 WA 。

2) 从内存工作区 WA 中选出其中关键字取最小值的记录, 记为 MINIMAX。(以后再选出关键字比它大的记录归入本归并段, 比它小的归入下一归并段)

3) 将 MINIMAX 记录输出到 FO 中去。

4) 若 FI 未读完, 则从 FI 输入下一个记录到 WA 中。

5) 从 WA 中所有关键字比 MINIMAX 记录的关键字大的记录中选出最小的关键字记录, 作为新的 MINIMAX。

6) 重复 3)~5), 直到在 WA 中选不出新的 MINIMAX 记录为止, 由此得到一个初始归并段, 输出一个归并段的结束标志到 FO 中去。

7) 重复 2)~6), 直到 WA 为空。由此得到全部初始归并段。

例如, 设待排文件 FI={17, 21, 05, 44, 10, 12, 56, 32, 29}, 内存工作区容量 w 为 3。排序过程如下 (波浪线标记的为每次的 MINIMAX):

表 7-2 置换-选择排序过程示例

输出文件 FO	工作区 WA	输入文件 FI
-	-	17,21,05,44,10,12,56,32,29
-	17 21 <u>05</u>	44,10,12,56,32,29
05	<u>17</u> 21 44	10,12,56,32,29
05 17	10 <u>21</u> 44	12,56,32,29
05 17 21	10 12 <u>44</u>	56,32,29
05 17 21 44	10 12 <u>56</u>	32,29
05 17 21 44 56	10 12 32	29
05 17 21 44 56 #	<u>10</u> 12 32	29
10	29 <u>12</u> 32	-
10 12	<u>29</u> 32	-
10 12 29	<u>32</u>	-
10 12 29 32	-	-
10 12 29 32 #	-	-

上述算法中, 选择 MINIMAX 记录的过程需利用败者树来实现。

7.7.5 最佳归并树

文件经过置换-选择排序之后, 得到的是长度不等的初始归并段。下面讨论如何组织初始归并段的归并顺序, 使 I/O 访问次数最少。

m-路归并排序可用一棵 m 叉树描述。因为每一次作 m 路归并都需要有 m 个归并段参加, 因此, 归并树是一棵只有度为 0 和度为 m 的结点的严格 m 叉树。

设由置换-选择得到 9 个初始归并段, 其长度 (记录数) 依次为: 9,30,12,18,3,17,2,6,24。现作 3-路平衡归并, 其归并树如图 7-10 所示。

在图 7-10 中, 各叶结点表示参加归并的一个初始归并段, 叶结点上的权值表示该初始归并段中的记录数, 根结点表示最终生成的归并段, 叶结点到根结点的路径长度表示在归并过程中的归并趟数, 各非叶结点代表归并成的新归并段, 则归并树的带权路径长度 WPL 即为归并过程中的总读记录数。因而在归并过程中, 总的 I/O 次数为 $2 \times WPL = 484$ 。

归并方案不同, 所得归并树亦不同, 树的带权路径长度 (外存 I/O 次数) 亦不同。为了优化归并树的 WPL, 可将第 4 章 Huffman 树的思想推广到 m 叉树的情形。在归并树中, 让记录数少的初始归并段最先归并, 记录数多的初始归并段最晚归并, 就可以建立总的 I/O 次数达到最少的最佳归并树。对上述 9 个初始归并段可构成一棵如图 7-11 所示的归并树, 按此树进行归并, 仅需对外存进行 446 次读/写, 这棵归并树便称作最佳归并树。

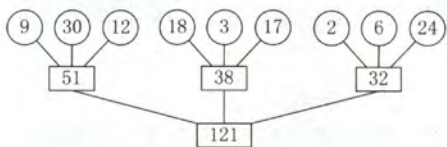


图 7-10 3 路平衡归并的归并树

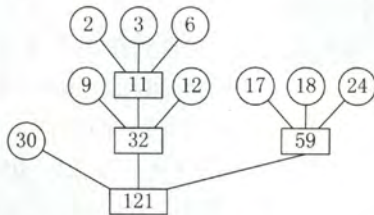


图 7-11 3 路平衡归并的最佳归并树

图 7-10 的 Huffman 树是一棵严格 3 叉树。若只有 8 个初始归并段, 设上例中少了一个长度为 30 的归并段。如果在设计归并方案时, 缺额的归并段留着最后, 即除了最后一次作 2-路归并外, 其他各次归并仍都是 3-路归并, 此归并方案的外存读/写次数为 386。显然不是最佳方案。

正确的做法是, 若初始归并段不足构成一棵严格 m 叉树时, 需添加长度为 0 的“虚段”, 按照 Huffman 树的原则, 权为 0 的叶子应离树根最远。因此, 最佳归并树应如图 7-12 所示。

如何判定添加虚段的数目?

设度为 0 的结点有 $n_0 (=n)$ 个, 度为 m 的结点有 n_m 个, 则对严格 m 叉树有 $n_0 = (m-1)n_m + 1$, 由此可以得出 $n_m = (n_0 - 1) / (m-1)$ 。

- 如果 $(n_0 - 1) \% (m-1) = 0$ ($\%$ 为取余运算), 则说明这 n_0 个叶结点 (初始归并段) 正好可以构造 m 叉归并树。此时, 内结点有 n_m 个。
- 如果 $(n_0 - 1) \% (m-1) = u \neq 0$, 则说明对于这 n_0 个叶结点, 其中有 u 个多余, 不能包含在 m 叉归并树中。为构造包含所有 n_0 个初始归并段的 m 叉归并树, 应在原有 n_m 个内结点的基础上再增加一个内结点。它在归并树中代替了一个叶结点位置, 被代替的叶结点加上刚才多出的 u 个叶结点, 再加上 $m-u-1$ 个空归并段, 就可以建立归并树。

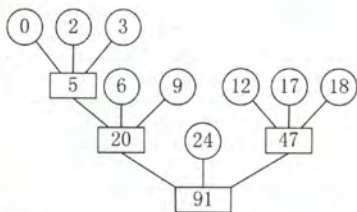


图 7-12 8 个归并段的最佳归并树

以图 7-12 为例, 用 8 个归并段构成 3 叉树, $(n_0 - 1) \% (m-1) = (8-1) \% (3-1) = 1$, 说明 7 个归并段刚好可以构成一个严格 3 叉树 (假设把以 5 为根的树看做一个叶子)。为此, 将叶子 5 变成一个内结点, 再添加 $3-1-1=1$ 个空归并段, 就可以构成一个严格 m 叉树。

7.7.6 本节试题精选

一、单项选择题

1. 设在磁盘上存放有 375 000 个记录, 作 5 路平衡归并排序, 内存工作区能容纳 600 个记录, 为把所有记录排好序, 需要作 () 趟归并排序。

- A. 3 B. 4 C. 5 D. 6

2. 设有 5 个初始归并段, 每个归并段有 20 个记录, 采用 5 路平衡归并排序, 若不采用败者树, 使用传统的顺序选出最小记录 (简单选择排序) 的方法, 总的比较次数是 (①); 若采用败者树最小的方法, 总的比较次数是 (②)。

- A. 20 B. 300 C. 396 D. 500

3. 置换-选择排序的作用是 ()。
- A. 置换-选择排序用于生成外排序的初始归并段
B. 置换-选择排序是完成将一个磁盘文件排序成有序文件的有效的排序算法
C. 置换-选择排序生成的初始归并段的长度是内存工作区的 2 倍
D. 置换-选择排序是对外排序中输入/归并/输出的并行处理
4. 最佳归并树在外排序中的作用是 ()。
- A. 完成 m 路归并排序
B. 设计 m 路归并排序的优化方案
C. 产生初始归并段
D. 与竞赛树的作用类似
5. 在下列关于外排序过程输入/输出缓冲区作用的叙述中不正确的是 ()。
- A. 暂存输入/输出记录
B. 内部归并的工作区
C. 产生初始归并段的工作区
D. 传送用户界面的消息
6. 在作 m 路平衡归并排序的过程中, 为实现输入/内部归并/输出的并行处理, 需要设置 (①) 个输入缓冲区和 (②) 个输出缓冲区。

① A. 2 B. m C. $2m-1$ D. $2m$

② A. 2 B. m C. $2m-1$ D. $2m$

7. 【2013 年计算机联考真题】

已知三叉树 T 中 6 个叶结点的权分别是 2, 3, 4, 5, 6, 7, T 的带权(外部)路径长度最小是 ()。

A. 27 B. 46 C. 54 D. 56

二、综合应用题

1. 多路平衡归并排序是外排序的主要方法, 试问多路平衡归并排序包括哪两个相对独立的阶段? 每个阶段完成何种工作?

2. 如果某个文件经内排序得到 80 个初始归并段, 试问:

1) 若使用多路平衡归并执行 3 趟完成排序, 那么应取得归并路数至少应为多少?

2) 如果操作系统要求一个程序同时可用的输入/输出文件的总数不超过 15 个, 则按多路归并至少需要几趟可以完成排序? 如果限定趟数, 可取的最低路数是多少?

3. 假设文件有 4 500 个记录, 在磁盘上每个块可放 75 个记录。计算机中用于排序的内存区可容纳 450 个记录。试问:

1) 可以建立多少个初始归并段? 每个初始归并段有多少记录? 存放于多少个块中?

2) 应采用几路归并? 请写出归并过程及每趟需要读写磁盘的块数。

4. 设初始归并段为 (10,15,31), (9,20), (22,34,37), (6,15,42), (12,37), (84,95)。试利用败者树进行 m 路归并, 手工执行选择最小的 5 个关键字的过程。

5. 给出 12 个初始归并段, 其长度分别为 30,44,8,6,3,20,60,18,9,62,68,85。现要作 4 路外归并排序, 试画出表示归并过程的最佳归并树, 并计算该归并树的带权路径长度 WPL。

7.7.7 答案与解析

一、单项选择题

1. B

初始归并段个数 $r=375000/600=625$, 因此, 归并趟数 $S=\lceil \log_m r \rceil = \lceil \log_5 625 \rceil = 4$ 。第一趟把 625 个归并段归并成 $625/5=125$; 第二趟把 125 个归并段归并成 $125/5=25$; 第三趟归并成 $25/5=5$ 个归并段; 第四趟归并成 $5/5=1$ 个归并段。

2. C、B

① 不采用败者树时，在 5 个记录中选出最小的需要作 4 次比较，总共有 100 个记录，需要作 99 次选择最小记录的操作，所以需要的比较次数为 $4 \times 99 = 396$ 次，故选 C。

② 采用败者树时，5-路归并意味着败者树的外结点有 5 个，败者树的高度 $h = \lceil \log_2 5 \rceil = 3$ 。每次在参加比较的记录中选择一个关键字最小的记录，比较次数不超过 h ，总共 100 个记录，需要的比较次数不超过 $100 \times 3 = 300$ 次，故选 B。

3. A

置换-选择排序是外排序中生成初始归并段的方法，用此方法得到的初始归并段的长度是不等长的，其长度平均是传统等长初始归并段的 2 倍，从而使得初始归并段数减少到原来的近二分之一。但是，置换-选择排序不是一个完整的生成有序文件的外排序算法。

4. B

最佳归并树在外排序中的作用是设计 m 路归并排序的优化方案，仿照构造 Huffman 树的方法，以初始归并段的长度为权值，构造具有最小带权路径长度的 m 叉 Huffman 树，可以有效地减少归并过程中的读写记录数，加快外排序的速度。

5. D

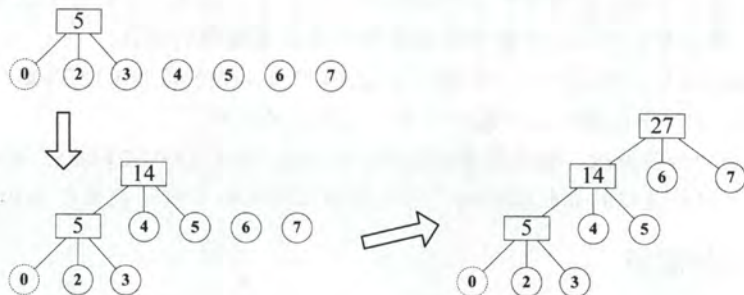
在外排序过程中输入/输出缓冲区就是排序的内存工作区，例如作 m 路平衡归并就需要 m 个输入缓冲区和 1 个输出缓冲区，用以存放参加归并的和归并完成的记录。在产生初始归并段时也可用作内排序的工作区。它没有传送用户界面的消息的任务。

6. D、A

在作 m 路平衡归并排序的过程中，为实现输入/内部归并/输出的并行处理，需要设置 $2m$ 个输入缓冲区和 2 个输出缓冲区，以便在执行内部归并时，能同时进行输入/输出操作。若仅设置 m 个输入缓冲区则仅能进行串行操作，无法并行处理。注重理解二路归并排序并在解题中留心题眼是正确解答的关键。

7. B

将哈夫曼树的思想推广到三叉树的情形。为了构成严格的三叉树，需添加权为 0 的虚叶结点，对于严格的三叉树 $(n_0 - 1) \% (3 - 1) = u \neq 0$ ，需要添加 $m - u - 1 = 3 - 1 - 1 = 1$ 个叶结点，说明 7 个叶结点刚好可以构成一个严格的三叉树。按照哈夫曼树的原则，权为 0 的叶结点应离树根最远，构造最小带权生成树的过程如下：



最小的带权路径长度为 $(2+3) \times 3 + (4+5) \times 2 + (6+7) \times 1 = 46$ 。

二、综合应用题

1. 解答：

多路平衡归并排序由两个相对独立的阶段组成：生成初始归并段和多趟归并排序。生成初始归并段阶段根据内存工作区的大小，将有 n 个记录的磁盘文件分批输入内存，采用有效的内排序

方法分别进行排序,生成若干个有序的子文件,即初始归并段。多趟归并排序阶段采用多路归并方法将这些归并段逐趟归并,最后归并成一个有序文件。

2. 解答:

1) 设归并路数为 m , 初始归并段个数 $r=80$, 根据归并趟数计算公式 $S=\lceil \log_m r \rceil = \lceil \log_m 80 \rceil = 3$, 得: $\log_m 80 \leq 3$, $m^3 \geq 80$ 。由此解得 $m \geq 5$, 即应取的归并路数至少为 5。

2) 设多路归并的归并路数为 m , 需要 m 个输入缓冲区和 1 个输出缓冲区。1 个缓冲区对应一个文件, 有 $m+1=15$, 因此 $m=14$, 可作 14 路归并。由 $S=\lceil \log_m r \rceil = \lceil \log_{14} 80 \rceil = 2$ 。即至少需要 2 趟归并可完成排序。若限定趟数为 2, 由 $S=\lceil \log_m 80 \rceil = 2$, 有 $80 \leq m^2$, 可取的最低路数为 9。即要在 2 趟内完成排序, 进行 9 路归并排序即可。

3. 解答:

1) 文件有 4 500 个记录, 用于排序的内存区可容纳 450 个记录, 可建立的初始归并段有 $4\ 500/450=10$ 个。每个初始归并段中有 450 个记录, 存于 $450/75=6$ 个块中。

2) 内存区可容纳 6 个块, 可建立 6 个缓冲区, 其中 5 个缓冲区用于输入, 1 个缓冲区用于输出, 因此, 可采用 5 路归并。归并过程如图 7-13 所示。

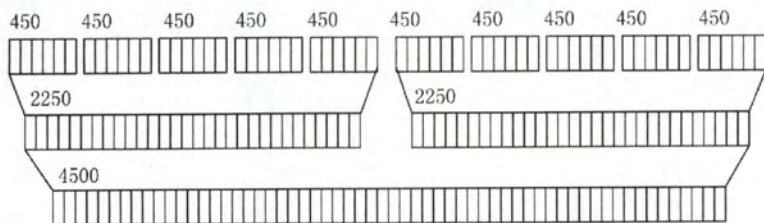


图 7-13 5 路归并的归并过程

共作了 2 趟归并, 每趟需要读 60 块、写 60 块。

4. 解答:

作 6 路归并排序, 选择最小的 5 个关键字的败者树如图 7-14 所示。

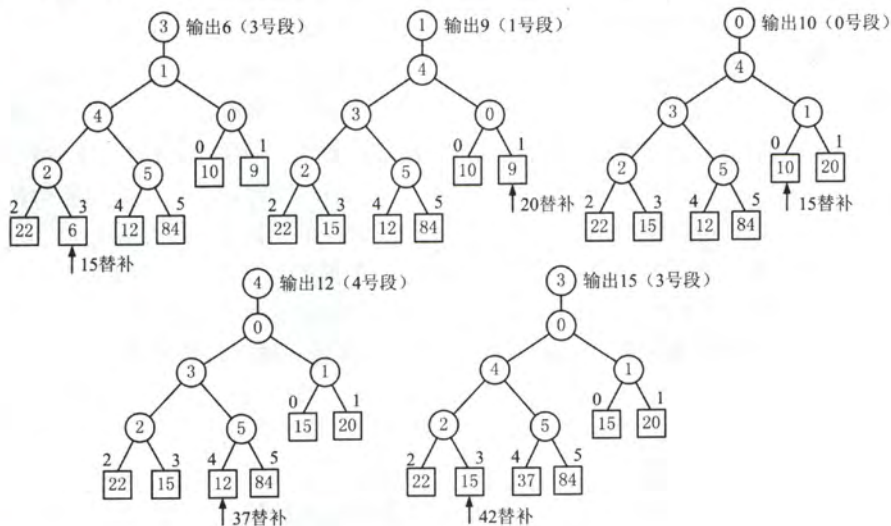


图 7-14 败者树的构造过程

5. 解答:

设初始归并段个数 $n=12$ ，外归并路数 $k=4$ ，计算 $(n-1) \% (k-1) = 11 \% 3 = 2 \neq 0$ ，说明不能做完全的 4 路归并，因为多出了 2 个初始归并段，必须添加 $k-2-1=1$ 个长度为 0 的空归并段，才能构成严格的 4 路归并树，即每次归并都有 k 个归并段参加归并。此时，归并树的内结点应有 $(n-1)/(k-1) = 12/3 = 4$ 个，如图 7-15 所示。

$$WPL = (3+6+8) \times 3 + (9+18+20+30+44+60+62) \times 2 + (68+85) \times 1 = 51 + 486 + 153 = 690。$$

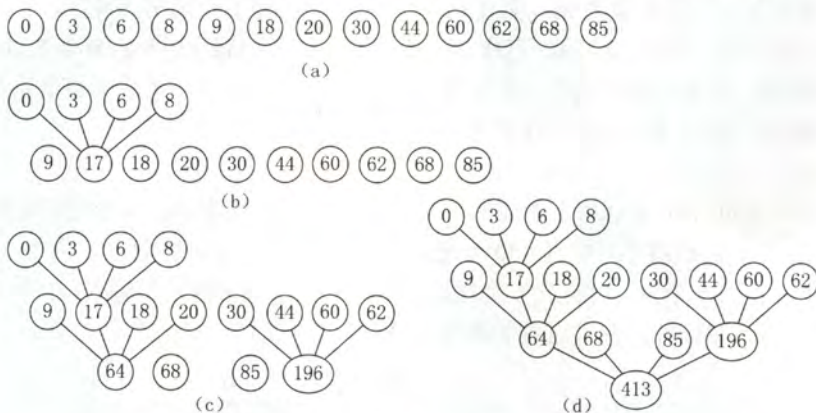


图 7-15 归并树的构造过程

归纳总结

下面我们对本章所介绍的排序算法进行一次系统的比较和复习。

一、直接插入排序、冒泡排序和简单选择排序是基本的排序方法，它们主要用于元素个数 n 不是很大 ($n < 10\,000$) 的情形。

它们的平均时间复杂度均为 $O(n^2)$ ，实现也都非常简单。直接插入排序对于规模很小的元素序列 ($n \leq 25$) 非常有效。它的时间复杂度与待排序元素序列的初始排列有关。在最好情况下，直接插入排序只需要 $n-1$ 次比较操作就可以完成，而且不需要交换操作。在平均情况下和最差情况下，直接插入排序的比较和交换操作都是 $O(n^2)$ 。冒泡排序在最好情况下只需要一趟排序过程就可以完成，此时也只需要 $n-1$ 次比较操作，不需要交换操作。简单选择排序的关键字比较次数与待排序元素序列的初始排列无关，其比较次数总是 $O(n^2)$ ，但元素移动次数则与待排序元素序列的初始排列有关，最好情况下数据不需要移动，最坏情况下元素移动次数不超过 $3(n-1)$ 次。

从空间复杂度来看，这三种基本的排序方法除了一个辅助元素外，都不需要其他额外空间。从稳定性来看，直接插入排序和冒泡排序都是稳定的，但简单选择排序不是。

二、对于中等规模元素序列 ($n \leq 1\,000$)，希尔排序是一种很好的选择。

在希尔排序中，开始时增量较大，分量较多，每个组内的记录数较少，因而记录的比较和移动次数较少，且移动距离较远；到后来步长越小（最后一步为 1），分组越少，每个组内的记录数也越多，但同时记录次序也越来越接近有序，因而记录的比较和移动次数也都比较少。从理论上和实验上都已经证明，在希尔排序中，记录的总的比较次数和总的移动次数比直接插入排序时少得多，特别是当 n 越大时效果越明显。而且，希尔排序代码简单，基本不需要什么额外内存，但希尔排序是一种不稳定的排序算法。

三、对于元素个数 n 很大的情况，可以采用快排、堆排序、归并排序或基数排序，其中快排

和堆排序都是不稳定的，而归并排序和基数排序是稳定的排序算法。

快速排序是最通用的高效的内排序算法，特别是它的划分思想经常在很多算法设计题中出现。平均情况下它的时间复杂度为 $O(n\log_2 n)$ ，一般情况下所需要的额外空间也是 $O(\log_2 n)$ 。但是快速排序在有些情况下也可能会退化（例如元素序列已经有序时），时间复杂度会增加到 $O(n^2)$ ，空间复杂度也会增加到 $O(n)$ 。但我们可以通过“三者取中”法来避免最坏情况的发生。

堆排序也是一种高效的内排序算法，它的时间复杂度是 $O(n\log_2 n)$ ，而且没有什么最坏情况会导致堆排序的运行明显变慢，并且堆排序基本不需要额外的空间。但堆排序不大可能提供比快速排序更好的平均性能。

归并排序也是一个重要的高效排序算法，它的一个重要特性是性能与输入元素序列无关，时间复杂度总是 $O(n\log_2 n)$ 。归并排序的主要缺点是需要 $O(n)$ 的额外存储空间。

基数排序是一种相对特殊的排序算法，这类算法不仅是对元素序列的关键字进行比较，更重要的是它们对关键字的不同位部分进行处理和比较。虽然基数排序具有线性增长的时间复杂度，但是由于在常规编程环境中，基数排序的线性时间开销实际上不比快速排序的时间开销小很多。并且由于基数排序基于的关键字抽取算法受到操作系统和排序元素的影响，其适应性远不如普通的进行比较和交换操作的排序方法。因此，在实际工作中，常规的高效排序算法如快速排序的应用要比基数排序广泛得多。基数排序需要的额外存储空间包括和待排序元素序列规模相同的存储空间以及与基数数目相等的一系列桶（一般用队列实现）。

四、混合使用

我们还可以把不同的排序算法混合使用，这也是得到普遍应用的一种算法改进方法，例如可以将直接插入排序集成到归并排序的算法中去。这种混合算法能够充分发挥不同算法各自的优势，从而在整体上得到更好的性能。

思维拓展

一道看起来很吓人的题目：

如何对 n 个整数进行排序，要求时间复杂度 $O(n)$ ，空间复杂度 $O(1)$ 。

（提示：假设待排序整数的范围为 $0 \sim 65535$ ，则设定一个数组 `int count[65535]` 并初始化为 0，则所需空间与 n 无关，为 $O(1)$ 。扫描一遍待排序列 `X[]`，`count[X[i]]++`，时间复杂度为 $O(n)$ ；再扫描一次 `count[]`，当 `count[i]>0` 时，则输出 `count[i]` 个 `i`，排序完毕所需的时间复杂度也为 $O(n)$ ；故总的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。另外，读者可能会说假如有负整数怎么办，这种情况下我们可以给所有整数都加上一个偏移量，使之都变成正整数，再使用上述方法即可。）

参考文献

- [1] 严蔚敏, 吴伟民. 数据结构(C语言版)[M]. 北京: 清华大学出版社, 2009.
- [2] 严蔚敏, 吴伟民. 数据结构题集(C语言版)[M]. 北京: 清华大学出版社, 2011.
- [3] 李春葆, 尹为民, 等. 数据结构教程[M]. 北京: 清华大学出版社, 2009.
- [4] 陈守孔, 胡潇琨, 李玲. 算法与数据结构考研试题精析[M]. 北京: 机械工业出版社, 2007.
- [5] 夏清国. 数据结构考研教案[M]. 西安: 西北工业大学出版社, 2006.
- [6] 本书编写组. 计算机专业基础综合考试大纲解析[M]. 北京: 高等教育出版社, 2009.
- [7] 李春葆, 等. 数据结构联考辅导教程[M]. 北京: 清华大学出版社, 2010.
- [8] 崔魏, 等. 计算机学科专业基础综合辅导讲义[M]. 北京: 原子能出版社, 2011.
- [9] 翔高教育. 计算机学科专业基础综合复习指南[M]. 上海: 复旦大学出版社, 2009.

2019 年操作系统考研复习指导

2019 年数据结构考研复习指导

2019 年计算机组成原理考研复习指导

2019 年计算机网络考研复习指导

2019 年计算机专业基础综合考试最后 8 套模拟题

计算机考研名校真题精析

王道程序员求职宝典

2019 年 数据结构 考研复习指导



本书是计算机专业研究生入学考试“数据结构”课程的复习用书，内容包括绪论、线性表、栈和队列、树与二叉树、图、查找、排序等。全书严格按照最新计算机统考大纲的数据结构部分，对大纲所涉及的知识进行集中梳理，力求内容精炼、重点突出、深入浅出。本书精选各名校的历年考研真题，并给出详细的解题思路，力求实现讲练结合、灵活掌握、举一反三的功效。通过本书“书本+在线”的学习方式和网上答疑方式，读者在使用本书时遇到的任何疑难点，都可以在王道论坛发帖与道友互动。

本书可作为考生参加计算机专业研究生入学考试的备考复习用书，也可作为计算机专业的学生学习数据结构课程的辅导用书。



策划编辑：谭海平
责任编辑：裴 杰
封面设计：创智时代

ISBN 978-7-121-33708-6



9 787121 337086 >

定价：49.50元