

计算机考研、学习交流
www.cskaoyan.com



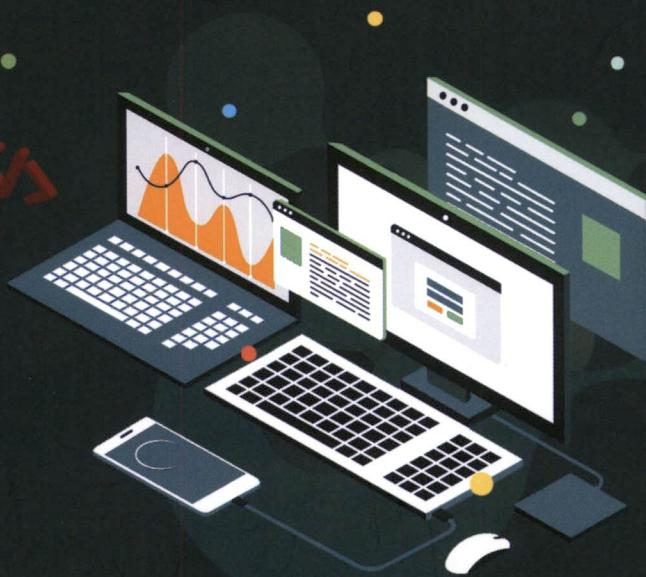
公众号：王道在线

推送计算机考研的报考数据、学习资源和复习经验等信息，获取王道训练营的相关信息。

正版书可以兑换免费配套视频，免费配套视频是2023版付费课程中的考点精讲部分。兑换方法及有效期见内页。



王道 考研系列



2024 年

数据结构 考研复习指导

◆ 王道论坛 组编



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

更多考研考证类日更网盘群&笔记类资料 请关注微信公众号【乘龙考研】

王道考研系列

2024 年

数据结构考研复习指导

王道论坛 组编

关注公众号【乘龙考研】
一手更新 稳定有保障

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、队列和数组，串，树与二叉树，图，查找，排序等。全书严格按照最新计算机考研大纲数据结构部分的要求，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、举一反三的效果。

本书既可作为考生参加计算机专业研究生入学考试的复习用书，又可作为计算机专业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

2024 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，2022.12

ISBN 978-7-121-44471-5

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（2022）第 200986 号

责任编辑：谭海平

印 刷：山东华立印务有限公司

装 订：山东华立印务有限公司

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：787×1092 1/16 印张：23.5 字数：646.7 千字

版 次：2022 年 12 月第 1 版

印 次：2023 年 1 月第 2 次印刷

定 价：75.00 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 zlts@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) 88254552, tan02@phei.com.cn。

本书配套视频使用方法

1

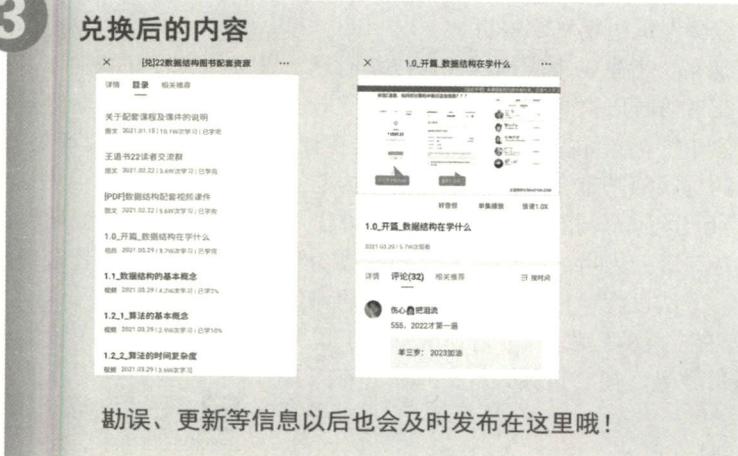


2



3

兑换后的内容



勘误、更新等信息以后也会及时发布在这里哦！

1. 部分解析兑换后扫码可见
2. 盗版书无兑换码请勿购买
3. 配套视频非王道最新网课
4. 配套视频不包含答疑服务



【关于兑换配套视频的说明】

1. 凭兑换码兑换相应科目的免费配套视频及课件，免费配套视频是2023版付费课程中的考点精讲部分，兑换一次即失效，兑换后不支持转赠。
2. 免费配套视频不是2024年付费课程，具体区别请见“王道在线”公众号中的详细说明。
3. 兑换码贴于封面右下角，刮开涂层可见，兑换码区分大小写，且无空格。
4. 兑换期限至2023年12月31日。

前 言

“王道考研系列”辅导书由王道论坛（cskaoyan.com）组织名校状元级选手编写，这套书不仅参考了国内外的优秀教辅，而且结合了高分选手的独特复习经验，包括对考点的讲解及对习题的选择和解析。“王道考研系列”单科辅导书，一共 4 本：

- 《2024 年数据结构考研复习指导》
- 《2024 年计算机组成原理考研复习指导》
- 《2024 年操作系统考研复习指导》
- 《2024 年计算机网络考研复习指导》

关注公众号【乘龙考研】
一手更新 稳定有保障

我们还围绕这套书开发了一系列计算机考研课程，赢得了众多读者的好评。这些课程包含考点精讲、习题详解、暑期直播训练营、冲刺串讲、带学督学和全程答疑服务等，并且只在“中国大学 MOOC”上发售。王道的课程同样是市面上领先的计算机考研课程，对于基础较为薄弱或“跨考”的读者，相信王道的课程和服务定能助你一臂之力。此外，我们也为购买正版图书的读者提供了 23 课程中的考点视频和课件，读者可凭兑换码兑换，23 统考大纲没有变化，该视频和本书完全匹配。考点视频升华了王道单科书中的考点讲解，强烈建议读者结合使用。

在冲刺阶段，王道还将出版 2 本冲刺用书：

- 《2024 年计算机专业基础综合考试冲刺模拟题》
- 《2024 年计算机专业基础综合考试历年真题解析》

深入掌握专业课的内容没有捷径，考生也不应抱有任何侥幸心理。只有扎实打好基础，踏实做题巩固，最后灵活致用，才能在考研时取得高分。我们希望辅导书能够指导读者复习，但学习仍然得靠自己，高分不是建立在任何空中楼阁之上的。对于想继续在计算机领域深造的读者来说，认真学习和扎实掌握计算机专业的这四门基础专业课，是最基本的前提。

“王道考研系列”是计算机考研学子口碑相传的辅导书，自 2011 版首次推出以来，就始终占据同类书销量的榜首位置，这就是口碑的力量。有这么多学长的成功经验，相信只要读者合理地利用辅导书，并且采用科学的复习方法，就一定能收获属于自己的那份回报。

“不包就业、不包推荐，培养有态度的码农。”王道训练营是王道团队打造的线下魔鬼式编程训练营。打下编程功底、增强项目经验，彻底转行入行，不再迷茫，期待有梦想的你！

参与本书编写工作的人员主要有赵霖、罗乐、徐秀瑛、张鸿林、韩京儒、赵淑芬、赵淑芳、罗庆学、赵晓宇、喻云珍、余勇、刘政学等。予人玫瑰，手有余香，王道论坛伴你一路同行！

对本书的任何建议，或有发现错误，欢迎扫码与我们联系，以便于我们及时优化或纠错。



风华漫舞

致 读 者

关注公众号【乘龙考研】

一手更新 稳定有保障

——王道单科辅导书使用方法的道友建议

我是“二战考生”，2012年第一次考研成绩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。从未达到录取线到考出比较满意的成绩，从蒙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择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辅导书展开的，从一遍、两遍、三遍看单科辅导书的积累提升，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。王道书能两次押中算法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。

下面说一说我的具体复习过程。

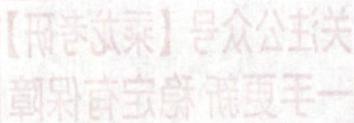
每天划给专业课的时间是3~4小时。第一遍仔细看课本，看完一章做一章单科辅导书上的习题（红笔标注错题），这一遍共持续2个月。第二遍主攻单科辅导书（红笔标注重难点），辅看课本。第二遍看单科辅导书和课本的速度快了很多，但感觉收获更多，常有温故知新的感觉，理解更深刻。（风华注，建议这里再速看第三遍，特别针对错题和重难点。模拟题做完后再跳看第四遍。）

以上是打基础阶段，注意，单科辅导书和课本我仔细精读了两遍，以便尽量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辅导书和课本。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点，最后的模拟就是查漏补缺。模拟题一定要严格按照考试时间去做（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，第三遍跳看单科辅导书，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，雨雪风霜从未间断地跑去自习，考研这人生一站终归没有辜负我的良苦用心。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而拼搏，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，我们都是有梦想的青年，我们正在逆袭，你呢？

感谢风华哥的信任，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



王道训练营

王道是道友们考研路上值得信赖的好伙伴，十多年来陪伴了上百万的计算机考研人，不离不弃。王道尊重的不是考研这个行当，而是考研学生的精神和梦想。考研可能是部分学生实现梦想的阶段，但应试的内容对 CSer 的职业生涯并无太多意义。对计算机专业的学生而言，专业功底和学习能力才是受用终生的资本，它决定了未来在技术道路上能走多远。从王道论坛、考研图书，到辅导课程，再到编程培训，王道只专注于计算机考研及编程领域。

计算机专业是一个靠实力吃饭的专业。我们团队中很多人的经历或许和现在的你们相似，也经历过本科时的迷茫，无非是自知能力太弱，以致底气不足。学历只是敲门砖，同样是名校硕士，有人如鱼得水，最终成为“Offer 帝”，有人却始终难入“编程与算法之门”，再次体会迷茫的痛苦。我们坚信一个写不出合格代码的计算机专业学生，即便考上了研究生，也只是给未来失业判了个“缓期执行”。我们也希望能做点事情帮助大家少走弯路。

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时光，趁着还有时间，应该去弥补本科期间应掌握的能力，缩小与“科班大佬们”的差距。

把参加王道训练营视为一次对自己的投资，投资自身和未来才是最好的投资。

王道训练营的面向人群

1. 面向就业

转行就业，但编程能力偏弱的学生。

考研并不是人生的唯一道路，努力拼搏奋斗的经历总是难忘的，但不论结果如何，都不应有太大的遗憾。不少考研路上的“失败者”在王道都实现了自己在技术发展上的里程碑，我们相信一个肯持续努力、积极上进的学生一定会找到自己正确的人生方向。

再不抓住当下，未来或将持续迷茫，逝去了的青春不复返。在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。

王道训练营致力于给有梦想、肯拼搏、敢奋斗的道友提供最好的平台！

2. 面向硕士

提升能力，刚考上计算机相关专业的准硕士。

考研逐年火爆，能考上名校确实是重要的转折，但硕士文凭早已不再稀缺。考研高分并不等于高薪 Offer，学历也不能保证你拿到好 Offer，名校的光环能让你获得更多面试机会，但真正要拿到好 Offer，比拼的是实力。同为名校硕士，Offer 的成色可能千差万别，有人轻松拿到腾讯、阿里、今日头条、百度等公司的优秀 Offer，有人面试却屡屡碰壁，最后只能“将就”签约。

人生中关键性的转折点不多，但往往能对自己的未来产生深远的影响，甚至决定了你未来的走向，高考、选专业、考研、找工作都是如此，把握住关键转折点需要眼光和努力。

3. 报名要求

- 具有本科学历，愿意通过奋斗去把握自己的人生，实现自身的价值。
- 完成开课前作业，用作业考察态度，才能获得最终的参加资格，宁缺毋滥！对于意志

不够坚定的同学而言，这些作业也算是设置的一道槛，决定了是否有参加的资格。

作业完成情况是最重要的考核标准，我们不会歧视跨度大的同学，坚定转行的同学往往会展更努力。跨度大、学校弱这些是无法改变的标签，唯一可以改变的就是通过持续努力来提升自身的技能，而通过高强度的短期训练是完全有可能逆袭的，太多的往期学员已有过证明。

4. 学习成效

迅速提升编程能力，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。以动手编程为驱动的教学模式，解决你在编程、思维上的不足，也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，训练营中不少往期准硕士学员后来陆续拿到了阿里、腾讯、今日头条、百度、美团、小米等一线互联网大厂的 Offer。这就是竞争力！

王道训练营的优势

这里都是道友，他们信任王道，乐于分享与交流，氛围优秀而纯粹。

一起经历过考研训练的生活、学习，大家很快会成为互帮互助的好战友，相互学习、共同进步，在转行的道路上，这就是最好的圈子。正如某期学员所言：“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。”

考研绝非人生的唯一出路，给自己换一条路走，去职场上好好发展或许会更好。即便考上研究生也不意味着高枕无忧，人生的道路还很漫长。

王道团队皆具有扎实的编程功底，他们用自己的技术和态度去影响训练营的学员，尽可能指导学员走上正确的发展道路是对道友信任的回报，也是一种责任！

王道训练营是一个平台，网罗王道论坛上有梦想、有态度的青年，并为他们的梦想提供土壤和圈子。王道始终相信“物竞天择，适者生存”，这里的生存不是指简简单单地活着，而是指活得有价值、活得有态度！

王道训练营的课程信息

王道训练营只在武汉设有线下校区，开设 4 种班型：

- Linux C 和 C++ 短期班（40~45 天，初试后开课，复试冲刺）
- Java EE 方向（4 个月，武汉校区）
- Linux C/C++ 方向（4 个月，武汉校区）
- Python 大数据方向（3 个半月，直播授课）

短期班的作用是在初试后及春节期间，快速提升学员的编程水平和项目经验，给复试、面试加成。其他三个班型的作用既可以面向就业，又可以提升能力或帮助打算继续考研的学员。

要想了解王道训练营，可以关注王道论坛“王道训练营”版面，或者扫码加老师微信。



扫描二维码，添加我的企业微信

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	8
归纳总结	10
思维拓展	10
第 2 章 线性表	11
2.1 线性表的定义和基本操作	11
2.1.1 线性表的定义	11
2.1.2 线性表的基本操作	12
2.1.3 本节试题精选	12
2.1.4 答案与解析	12
2.2 线性表的顺序表示	13
2.2.1 顺序表的定义	13
2.2.2 顺序表上基本操作的实现	14
2.2.3 本节试题精选	16
2.2.4 答案与解析	18
2.3 线性表的链式表示	27
2.3.1 单链表的定义	27
2.3.2 单链表上基本操作的实现	28
2.3.3 双链表	31
2.3.4 循环链表	33
2.3.5 静态链表	33
2.3.6 顺序表和链表的比较	34
2.3.7 本节试题精选	35
2.3.8 答案与解析	40
归纳总结	59
思维拓展	60
第 3 章 栈、队列和数组	61
3.1 栈	61

3.1.1 栈的基本概念	61
3.1.2 栈的顺序存储结构	62
3.1.3 栈的链式存储结构	64
3.1.4 本节试题精选	64
3.1.5 答案与解析	67
3.2 队列	74
3.2.1 队列的基本概念	74
3.2.2 队列的顺序存储结构	75
3.2.3 队列的链式存储结构	77
3.2.4 双端队列	78
3.2.5 本节试题精选	80
3.2.6 答案与解析	82
3.3 栈和队列的应用	88
3.3.1 栈在括号匹配中的应用	88
3.3.2 栈在表达式求值中的应用	88
3.3.3 栈在递归中的应用	89
3.3.4 队列在层次遍历中的应用	90
3.3.5 队列在计算机系统中的应用	91
3.3.6 本节试题精选	91
3.3.7 答案与解析	93
3.4 数组和特殊矩阵	98
3.4.1 数组的定义	98
3.4.2 数组的存储结构	99
3.4.3 特殊矩阵的压缩存储	99
3.4.4 稀疏矩阵	101
3.4.5 本节试题精选	102
3.4.6 答案与解析	103
归纳总结	104
思维拓展	105
第 4 章 串	106
*4.1 串的定义和实现	106
4.1.1 串的定义	106
4.1.2 串的存储结构	107
4.1.3 串的基本操作	108
4.2 串的模式匹配	108
4.2.1 简单的模式匹配算法	108
4.2.2 串的模式匹配算法——KMP 算法	109
4.2.3 KMP 算法的进一步优化	113
4.2.4 本节试题精选	114
4.2.5 答案与解析	115
归纳总结	119
思维拓展	119
第 5 章 树与二叉树	120
5.1 树的基本概念	120

5.1.1 树的定义	120
5.1.2 基本术语	121
5.1.3 树的性质	122
5.1.4 本节试题精选	122
5.1.5 答案与解析	123
5.2 二叉树的概念	124
5.2.1 二叉树的定义及其主要特性	124
5.2.2 二叉树的存储结构	126
5.2.3 本节试题精选	127
5.2.4 答案与解析	129
5.3 二叉树的遍历和线索二叉树	133
5.3.1 二叉树的遍历	133
5.3.2 线索二叉树	137
5.3.3 本节试题精选	140
5.3.4 答案与解析	145
5.4 树、森林	164
5.4.1 树的存储结构	164
5.4.2 树、森林与二叉树的转换	166
5.4.3 树和森林的遍历	167
5.4.4 本节试题精选	168
5.4.5 答案与解析	170
5.5 树与二叉树的应用	176
5.5.1 哈夫曼树和哈夫曼编码	176
5.5.2 并查集	178
5.5.3 本节试题精选	179
5.5.4 答案与解析	181
归纳总结	185
思维拓展	186
第6章 图	187
6.1 图的基本概念	187
6.1.1 图的定义	187
6.1.2 本节试题精选	190
6.1.3 答案与解析	192
6.2 图的存储及基本操作	194
6.2.1 邻接矩阵法	194
6.2.2 邻接表法	196
6.2.3 十字链表	197
6.2.4 邻接多重表	198
6.2.5 图的基本操作	198
6.2.6 本节试题精选	199
6.2.7 答案与解析	201
6.3 图的遍历	205
6.3.1 广度优先搜索	205
6.3.2 深度优先搜索	207

关注公众号【乘龙考研】
一手更新 稳定有保障

6.3.3 图的遍历与图的连通性	208
6.3.4 本节试题精选	209
6.3.5 答案与解析	211
6.4 图的应用	216
6.4.1 最小生成树	216
6.4.2 最短路径	219
6.4.3 有向无环图描述表达式	222
6.4.4 拓扑排序	222
6.4.5 关键路径	224
6.4.6 本节试题精选	226
6.4.7 答案与解析	234
归纳总结	246
思维拓展	247
第 7 章 查找	248
7.1 查找的基本概念	248
7.2 顺序查找和折半查找	249
7.2.1 顺序查找	249
7.2.2 折半查找	251
7.2.3 分块查找	252
7.2.4 本节试题精选	253
7.2.5 答案与解析	256
7.3 树型查找	261
7.3.1 二叉排序树 (BST)	261
7.3.2 平衡二叉树	265
7.3.3 红黑树	269
7.3.4 本节试题精选	274
7.3.5 答案与解析	277
7.4 B 树和 B+ 树	286
7.4.1 B 树及其基本操作	286
7.4.2 B+ 树的基本概念	289
7.4.3 本节试题精选	290
7.4.4 答案与解析	293
7.5 散列表	298
7.5.1 散列表的基本概念	298
7.5.2 散列函数的构造方法	298
7.5.3 处理冲突的方法	299
7.5.4 散列查找及性能分析	300
7.5.5 本节试题精选	301
7.5.6 答案与解析	304
归纳总结	309
思维拓展	309
第 8 章 排序	310
8.1 排序的基本概念	311

8.1.1 排序的定义	311
8.1.2 本节试题精选	311
8.1.3 答案与解析	312
8.2 插入排序	312
8.2.1 直接插入排序	312
8.2.2 折半插入排序	314
8.2.3 希尔排序	314
8.2.4 本节试题精选	316
8.2.5 答案与解析	317
8.3 交换排序	319
8.3.1 冒泡排序	319
8.3.2 快速排序	321
8.3.3 本节试题精选	323
8.3.4 答案与解析	325
8.4 选择排序	330
8.4.1 简单选择排序	330
8.4.2 堆排序	331
8.4.3 本节试题精选	333
8.4.4 答案与解析	335
8.5 归并排序和基数排序	340
8.5.1 归并排序	340
8.5.2 基数排序	341
8.5.3 本节试题精选	343
8.5.4 答案与解析	344
8.6 各种内部排序算法的比较及应用	346
8.6.1 内部排序算法的比较	346
8.6.2 内部排序算法的应用	347
8.6.3 本节试题精选	348
8.6.4 答案与解析	350
8.7 外部排序	354
8.7.1 外部排序的基本概念	354
8.7.2 外部排序的方法	354
8.7.3 多路平衡归并与败者树	355
8.7.4 置换-选择排序（生成初始归并段）	356
8.7.5 最佳归并树	357
8.7.6 本节试题精选	358
8.7.7 答案与解析	359
归纳总结	362
思维拓展	363
参考文献	364

关注公众号【乘龙考研】
一手更新 稳定有保障

第 1 章 绪 论

关注公众号【乘龙考研】
一手更新 稳定有保障

【考纲内容】

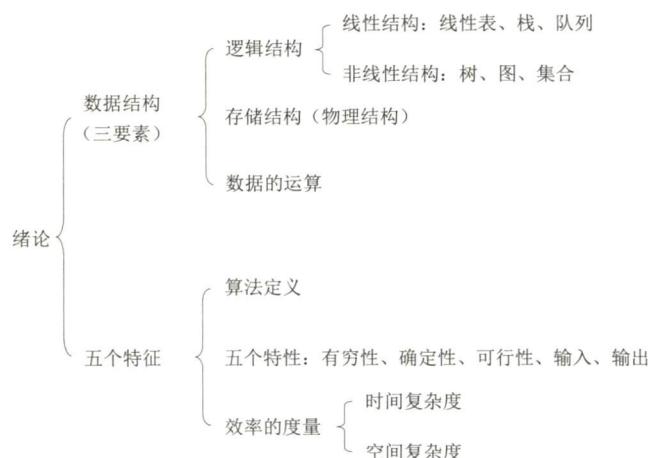
算法时间复杂度和空间复杂度的分析与计算

兑换后



看视频讲解

【知识框架】



【复习提示】

本章内容是数据结构概述，并不在考研大纲中。读者可通过对本章的学习，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，一定要熟练掌握，算法设计题通常都会要求分析时间复杂度、空间复杂度，同时会出现考查时间复杂度的选择题。

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。数据是计算机程序加工的原料。

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。

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的一组操作的总称。

- 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。抽象数据组织及与之相关的操作。

5. 数据结构

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。在任何问题中，数据元素都不是孤立存在的，它们之间存在某种关系，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如图 1.1 所示。

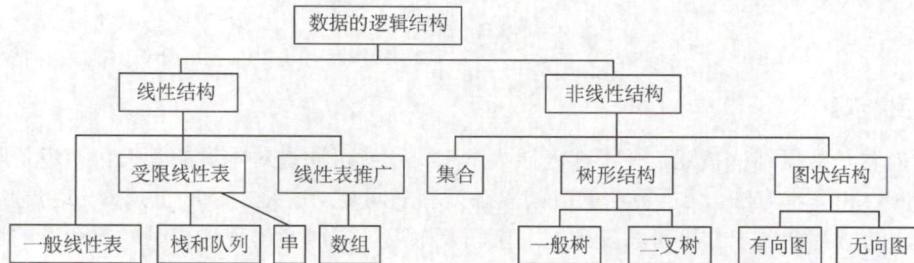


图 1.1 数据的逻辑结构分类图

集合。结构中的数据元素之间除“同属一个集合”外，别无其他关系，如图 1.2(a)所示。

线性结构。结构中的数据元素之间只存在一对一的关系，如图 1.2(b)所示。

树形结构。结构中的数据元素之间存在一对多的关系，如图 1.2(c)所示。

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，如图 1.2(d)所示。

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。数据的存储结构是用计算机语言实现的逻辑结构，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 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。

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中，元素之间的关系由存储单元的邻接关系来体现。其优点是可以实现随机存取，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来表示元素之间的逻辑关系。其优点是不会出现碎片现象，能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，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；缺点是附加的索引表额外占用存储空间。另外，增加和删除数据时也要修改索引表，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，又称哈希（Hash）存储。其优点是检索、增加和删除结点的操作都很快；缺点是若散列函数不好，则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

01. 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
02. 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
03. 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
04. 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
05. 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其存储结构
 D. 数据结构仅由其逻辑结构和存储结构决定
06. 在存储数据时，通常不仅要存储各数据元素的值，而且要存储（ ）。
 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

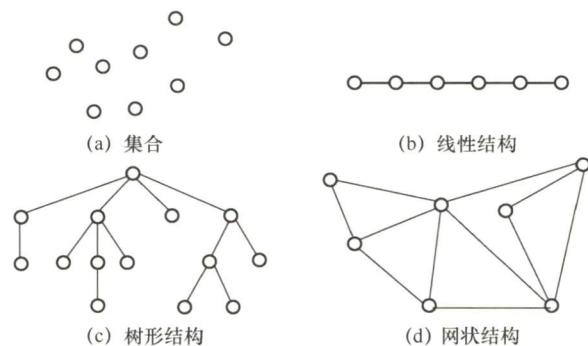


图 1.2 4 类基本结构关系示例图

关注公众号【乘龙考研】
一手更新 稳定有保障

07. 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

01. 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？
02. 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时，其运算效率不同。

1.1.4 答案与解析

一、单项选择题

关注公众号【乘龙考研】
一手更新 稳定有保障

01. 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，从而构成一个完整的数据结构定义。

02. 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。

03. C

顺序表、哈希表和单链表是三种不同的数据结构，既描述逻辑结构，又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，仅描述元素之间的逻辑关系，它既可以链式存储，又可以顺序存储，故属于逻辑结构。

04. 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，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。

05. A

数据的逻辑结构是从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。

06. C

在存储数据时，不仅要存储数据元素的值，而且要存储数据元素之间的关系。

07. A

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

二、综合应用题

01. 【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，前者通常用于表示层次关系，而后者通常用于排序和查找。虽然它们的运算都有建立树、插入结点、删除结点和查找结点等功能，但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 $O(\log_2 n)$ 。

02. 【解答】

线性表既可以用顺序存储方式实现，又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 $O(1)$ 。

关注公众号【乘龙考研】
一手更新 稳定有保障

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，其中的每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。算法中每条指令必须有确切的含义，对于相同的输入只能得出相同的输出。
- 3) 可行性。算法中描述的操作都可以通过已经实现的基本运算执行有限次来实现。
- 4) 输入。一个算法有零个或多个输入，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输出，这些输出是与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。算法应能够正确地解决求解问题。
- 2) 可读性。算法应具有良好的可读性，以帮助人们理解。
- 3) 健壮性。输入非法数据时，算法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) 高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为 $T(n)$ ，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。算法中基本运算（最深层循环内的语句）的频度与 $T(n)$ 同数量级，因此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

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

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 $0 \leq T(n) \leq C f(n)$ 。

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 $A[0..n-1]$ 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;
```

^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， $f(n) = an^3 + bn^2 + cn$ 的时间复杂度为 $O(n^3)$ 。

该算法中语句 3 (基本运算) 的频度不仅与问题规模 n 有关, 而且与输入实例中 A 的各元素的取值及 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

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

b) 乘法规则

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

常见的渐近时间复杂度为

$$O(1) \leq O(\log n) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(n^3) \leq O(2^n) \leq O(n!) \leq O(n^n)$$

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。记为

$$S(n) \equiv O(g(n))$$

一个程序在执行时除需要存储空间来存放本身所用的指令、常数、变量和输入数据外，还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的辅助空间为常量，即 $O(1)$ 。

1.2.3 本节试题精选

一 单项选择题


```
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. $O(\log_2 n)$

04. 有以下算法，其时间复杂度为（ ）。

34. 在以下算法中，其时间复杂度为（ ）

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

有任何网盘群类资料需求都可添加学长微信 Learn52013 祝您考研上岸

```
i++;
}
```

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

05. 程序段如下：

```
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，则最后一行语句的频度在最坏情况下是（ ）。

- A. $O(n)$ B. $O(n \log n)$ C. $O(n^3)$ D. $O(n^2)$

06. 以下算法中加下画线的语句的执行次数为（ ）。

```
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

07. 【2011 统考真题】设 n 是描述问题规模的非负整数，下面的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. $O(n^2)$

08. 【2012 统考真题】求整数 n ($n \geq 0$) 的阶乘的算法如下，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. $O(n^2)$

09. 【2013 统考真题】已知两个长度分别为 m 和 n 的升序链表，若将它们合并为长度为 $m+n$ 的一个降序链表，则最坏情况下的时间复杂度是（ ）。

- A. $O(n)$ B. $O(mn)$ C. $O(\min(m, n))$ D. $O(\max(m, n))$

10. 【2014 统考真题】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. $O(n^2)$

11. 【2017 统考真题】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. $O(n \log n)$

12. 【2019 统考真题】设 n 是描述问题规模的非负整数，下列程序段的时间复杂度是（ ）。

关注公众号【乘龙考研】
一手更新 稳定有保障

```

x=0;
while (n>=(x+1)*(x+1))
x=x+1;

```

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

13. 【2022 统考真题】下列程序段的时间复杂度是（ ）。

```

int sum=0;
for(int i=1;i<n;i*=2)
    for(int j=0;j<i;j++)
        sum++;

```

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

二、综合应用题

01. 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）。

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

式中， n 是问题的规模，为简单起见，设 n 是 2 的整数次幂。

02. 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++;
      }
③   for(i=1;i<=n;i++)
      for(j=1;j<=i;j++)
          for(k=1;k<=j;k++)
              x++;

```

```

②   y=0;
      while((y+1)*(y+1)<=n)
          y=y+1;
④   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

关注公众号【乘龙考研】
一手更新 稳定有保障

01. B

本题是中山大学考研真题，题目本身没有问题，考查的是算法的定义。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实现。不少读者认为 C 也对，它只是算法的必要条件，不能成为算法的定义。

02. C

时间复杂度为 $O(n^2)$ ，说明算法的时间复杂度 $T(n)$ 满足 $T(n) \leq cn^2$ (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 。

03. D

找出基本运算 $i=i*2$ ，设执行次数为 t ，则 $2^t \leq n$ ，即 $t \leq \log_2 n$ ，因此时间复杂度 $T(n) = O(\log_2 n)$ 。

更直观的方法：计算基本运算 $i=i*2$ 的执行次数（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 $T(n) = O(\log_2 n)$ 。（注意，本方法可灵活运用到第 4 题和第 8 题。）

04. C

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

05. D

这是冒泡排序的算法代码，考查最坏情况下的元素交换次数（若觉得理解困难可在学完第 8

章后再回顾)。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 $O(n^2)$ 。

06. A

m++语句的执行次数为

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

07. A

基本运算(执行频率最高的语句)为 $x=2*x$ ，每执行一次 x 乘 2，设执行次数为 t ，则有 $2^{t+1} < n/2$ ，所以 $t < \log_2(n/2) - 1 = \log_2 n - 2$ ，得 $T(n) = O(\log_2 n)$ 。

08. B

本题是求阶乘 $n!$ 的递归代码，即 $n \times (n-1) \times \dots \times 1$ 。每次递归调用时 $\text{fact}()$ 的参数减 1，递归出口为 $\text{fact}(1)$ ，一共执行 n 次递归调用 $\text{fact}()$ ，故 $T(n) = O(n)$ 。

09. 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，因为 $2\max(m, n) \geq m + n$ ，所以时间复杂度为 $O(\max(m, n))$ 。

10. C

内层循环条件 $j \leq n$ 与外层循环的变量无关，各自独立，每执行一次 j 自增 1，每次内层循环都执行 n 次。外层循环条件 $k \leq n$ ，增量定义为 $k*=2$ ，可知循环次数 t 满足 $k = 2^t \leq n$ ，即 $t \leq \log_2 n$ 。即内层循环的时间复杂度为 $O(n)$ ，外层循环的时间复杂度为 $O(\log_2 n)$ 。对于嵌套循环，根据乘法规则可知，该段程序的时间复杂度 $T(n) = T_1(n) \times T_2(n) = O(n) \times O(\log_2 n) = O(n \log_2 n)$ 。

11. B

基本运算 $\text{sum}+=++i$ ，它等价于 $++i; \text{sum}=\text{sum}+i$ ，每执行一次 i 自增 1。 $i=1$ 时 $\text{sum}=0+1$ ； $i=2$ 时 $\text{sum}=0+1+2$ ； $i=3$ 时 $\text{sum}=0+1+2+3$ ，以此类推得出 $\text{sum}=0+1+2+3+\dots+i=(1+i)*i/2$ ，可知循环次数 t 满足 $(1+t)*t/2 < n$ ，因此时间复杂度为 $O(n^{1/2})$ 。

注意：统考真题中经常把 \log_2 书写为 \log ，此时默认底数为 2。

12. B

假设第 k 次循环终止，则第 k 次执行时， $(x+1)^2 > n$ ， x 的初始值为 0，第 k 次判断时， $x=k-1$ ，即 $k^2 > n$ ， $k > \sqrt{n}$ ，因此该程序段的时间复杂度为 $O(\sqrt{n})$ 。因此选 B。

13. B

当外层循环的变量 i 取不同值时，内层循环就执行多少次，因此总循环次数为 i 的所有取值之和。假设外层循环共执行 k 次，当 $i=1, 2, 4, 8, \dots, 2^{k-1}$ ($2^{k-1} < n \leq 2^k$) 时，内层循环执行 i 次，因此总循环次数 $T = 1 + 2 + 4 + 8 + \dots + 2^{k-1} = 2^k - 1$ ，即 $n < T < 2n$ ，时间复杂度为 $O(n)$ 。

二、综合应用题

01. 【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 $T(n) = 2^{\log_2 n} + \log_2 n \times n =$

关注公众号【乘龙考研】
一手更新 稳定有保障

$n(\log_2 n + 1)$, 也就是 $O(n \log_2 n)$ 。

02. 【解答】

- ① 基本语句 $k=k+10*i$ 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 $t^2 \leq n$, 得 $T(n) = O(n^{1/2})$ 。
- ③ 基本语句 $x++$ 的执行次数为 $T(n) = O\left(\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j 1\right) = O\left(\frac{1}{6}n^3\right) = O(n^3)$ 。
- ④ 内循环执行 m 次, 外循环执行 n 次, 根据乘法原理, 共执行了 $m \times n$ 次, 故 $T(m, n) = O(m \times n)$ 。

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, 很多读者在做题时一眼就能看出程序的时间复杂度, 但就是无法规范地表述其推导过程。为此, 编者查阅众多资料, 总结出了此类题型的两种形式, 供大家参考。

1. 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, 将之代入条件中进行计算。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, 记 t 为该程序的执行次数并令 $t=y-5$, 有 $y=t+5$, $(t+5+1) \times (t+5+1) < n$, 得 $t < \sqrt{n} - 6$, 即 $T(n) = O(\sqrt{n})$ 。

2. 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。多层次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 9 的时间复杂度分析如下:

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

即 $T(n) = O(n)$ 。

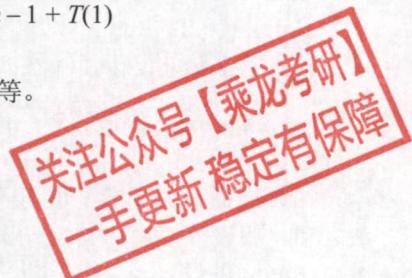
- 非递归程序比较简单, 可以直接累计次数, 例如习题 11 等。

思维拓展

求解斐波那契数列

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

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

关注公众号【乘龙考研】
一手更新稳定有保障

兑换后



看视频讲解

【考纲内容】

- (一) 线性表的基本概念
- (二) 线性表的实现
 - 顺序存储；链式存储
- (三) 线性表的应用

【知识框架】



【复习提示】

线性表是算法题命题的重点。这类算法题实现起来比较容易且代码量较少，但是要求具有最优的性能（时间复杂度、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），在平时的学习中多注重培养动手能力。另外，需要提醒的是，算法最重要的是思想！考场上的时间紧迫，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，而不必过于拘泥每个细节。注意算法题只能用 C/C++语言实现。

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 $n = 0$ 时线性表是一个空表。若用 L 命名线性表，则其一般表示为

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

式中， a_1 是唯一的“第一个”数据元素，又称表头元素； a_n 是唯一的“最后一个”数据元素，又称表尾元素。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（“直接前驱”和“前驱”、“直接后继”和“后继”通常被视为同义词）。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，表中元素有其先后次序。
- 表中元素都是数据元素，每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，而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 中的第 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种存储结构，存储结构不同，算法的实现也不同。②符号“&”表示 C++ 语言中的引用调用，在 C 语言中采用指针也可达到同样的效果。

2.1.3 本节试题精选

单项选择题

01. 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

02. 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

03. 在线性表中，除开始元素外，每个元素（ ）。

- A. 只有唯一的前驱元素 B. 只有唯一的后继元素
C. 有多个前驱元素 D. 有多个后继元素

2.1.4 答案与解析

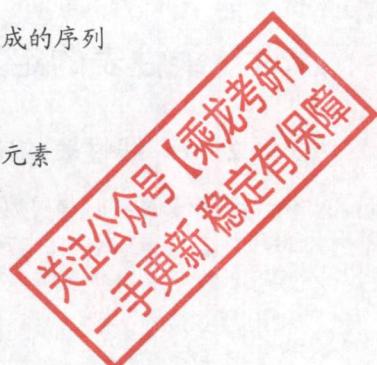
单项选择题

01. C

线性表是由具有相同数据类型的有限数据元素组成的，数据元素是由数据项组成的。

02. B

线性表定义的要求为：相同数据类型、有限序列。选项 C 的元素个数是无穷个，错误；选项



A集合中的元素没有前后驱关系，错误；选项D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。只有选项B符合线性表定义的要求。

03. A

线性表中，除最后一个（或第一个）元素外，每个元素都只有一个后继（或前驱）元素。

2.2 线性表的顺序表示

2.2.1 顺序表的定义

关注公众号【乘龙考研】
一手更新 稳定有保障

线性表的顺序存储又称顺序表。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，称*i*为元素*a_i*在线性表中的位序。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	<i>a₁</i>	LOC(A)
1	<i>a₂</i>	LOC(A)+sizeof(ElemType)
\vdots	\vdots	
<i>i-1</i>	<i>a_i</i>	LOC(A)+(i-1)×sizeof(ElemType)
\vdots	\vdots	
<i>n-1</i>	<i>a_n</i>	LOC(A)+(n-1)×sizeof(ElemType)
\vdots	\vdots	
MaxSize-1	\vdots	LOC(A)+(MaxSize-1)×sizeof(ElemType)

图2.1 线性表的顺序存储结构

每个数据元素的存储位置都和线性表的起始位置相差一个和该数据元素的位序成正比的常数，因此，顺序表中的任意一个数据元素都可以随机存取，所以线性表的顺序存储结构是一种随机存取的存储结构。通常用高级程序设计语言中的数组来描述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，则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，再加入新的数据就会产生溢出，进而导致程序崩溃。

而在动态分配时，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，就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，而不需要为线性表一次性地划分所有空间。

```
#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，它同样属于顺序存储结构，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动态决定。

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在时间 $O(1)$ 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

这里仅讨论顺序表的插入、删除和按值查找的算法，其他基本操作的算法都比较简单。

(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 。若 i 的输入不合法，则返回 $false$ ，表示插入失败；否则，将第 i 个元素及其后的所有元素依次往后移动一个位置，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(SeqList &L, int i, ELEMType e){
    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。为何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，顺序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，用引用变量 e 返回。若 i 的输

入不合法，则返回 `false`；否则，将被删元素赋给引用变量 `e`，并将第 $i+1$ 个元素及其后的所有元素依次往前移动一个位置，返回 `true`。

```
bool ListDelete(SqList &L, int i, Elemtyp e) {
    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 $O(1)$ 。

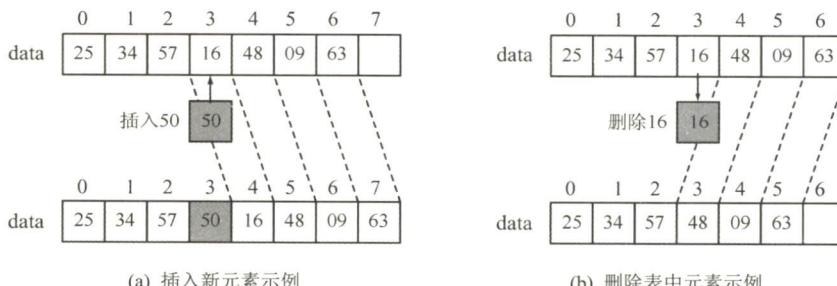
最坏情况：删除表头元素（即 $i = 1$ ），需移动除表头元素外的所有元素，时间复杂度为 $O(n)$ 。

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，顺序表删除算法的平均时间复杂度为 $O(n)$ 。

可见，顺序表中插入和删除操作的时间主要耗费在移动元素上，而移动元素的个数取决于插入和删除元素的位置。图 2.2 所示为一个顺序表在进行插入和删除操作前、后的状态，以及其数据元素在存储空间中的位置变化和表长的变化。在图 2.2(a)中，将第 4 个至第 7 个元素从后往前依次后移一个位置，在图 2.2(b)中，将第 5 个至第 7 个元素从前往后依次前移一个位置。



(a) 插入新元素示例

(b) 删除表中元素示例

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, Elemtyp e) {
    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
}
```

关注公众号【乘龙考研】
一手更新 稳定有保障

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

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

因此，顺序表按值查找算法的平均时间复杂度为 $O(n)$ 。

关注公众号【乘龙考研】
一手更新 稳定有保障

2.2.3 本节试题精选

一、单项选择题

01. 下述（ ）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
02. 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
03. 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
04. 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
05. 一个线性表最常用的操作是存取任意一个指定序号的元素并在最后进行插入、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
06. 在 n 个元素的线性表的数组表示中，时间复杂度为 $O(1)$ 的操作是（ ）。
- I. 访问第 i ($1 \leq i \leq n$) 个结点和求第 i ($2 \leq i \leq n$) 个结点的直接前驱
 - 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 ($1 \leq i \leq n$) 个结点后插入一个结点
- A. I
 - B. II、III
 - C. I、II
 - D. I、II、III
07. 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在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
08. 在一个长度为 n 的顺序表中删除第 i ($1 \leq i \leq n$) 个元素时，需向前移动（ ）个元素。
- A. n
 - B. $i - 1$
 - C. $n - i$
 - D. $n - i + 1$
09. 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，则 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. $0 \leq i \leq n$
11. 顺序表的插入算法中，当 n 个空间已满时，可再申请增加分配 m 个空间，若申请失败，则说明系统没有（ ）可分配的存储空间。
 A. m 个 B. m 个连续 C. $n+m$ 个 D. $n+m$ 个连续

二、综合应用题

01. 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。
02. 设计一个高效算法，将顺序表 L 的所有元素逆置，要求算法的空间复杂度为 $O(1)$ 。
03. 对长度为 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。
04. 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，若 s 或 t 不合理或顺序表为空，则显示出错信息并退出运行。
05. 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，若 s 或 t 不合理或顺序表为空，则显示出错信息并退出运行。
06. 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。
07. 将两个有序顺序表合并为一个新的有序顺序表，并由函数返回结果顺序表。
08. 已知在一维数组 $A[m+n]$ 中依次存放两个线性表 $(a_1, a_2, a_3, \dots, a_m)$ 和 $(b_1, b_2, b_3, \dots, b_n)$ 。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。
09. 线性表 $(a_1, a_2, a_3, \dots, a_n)$ 中的元素递增有序且按顺序存储于计算机内。要求设计一个算法，完成用最少时间在表中查找数值为 x 的元素，若找到，则将其与后继元素位置相交换，若找不到，则将其插入表中并使表中元素仍递增有序。
10. 【2010 统考真题】设将 n ($n > 1$) 个整数存放到一维数组 R 中。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 统考真题】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 统考真题】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 $a_{p1} = a_{p2} = \dots = a_{pm} = x$ 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。

13. 【2018 统考真题】给定一个含 n ($n \geq 1$) 个整数的数组, 请设计一个在时间上尽可能高效的算法, 找出数组中未出现的最小正整数。例如, 数组 $\{-5, 3, 2, 3\}$ 中未出现的最小正整数是 1; 数组 $\{1, 2, 3\}$ 中未出现的最小正整数是 4。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时间复杂度和空间复杂度。

14. 【2020 统考真题】定义三元组 (a, b, c) (a, b, c 均为整数) 的距离 $D = |a-b| + |b-c| + |c-a|$ 。给定 3 个非空整数集合 S_1, S_2 和 S_3 , 按升序分别存储在 3 个数组中。请设计一个尽可能高效的算法, 计算并输出所有可能的三元组 (a, b, c) ($a \in S_1, b \in S_2, c \in S_3$) 中的最小距离。例如 $S_1 = \{-1, 0, 9\}$, $S_2 = \{-25, -10, 10, 11\}$, $S_3 = \{2, 9, 17, 30, 41\}$, 则最小距离为 2, 相应的三元组为 $(9, 10, 9)$ 。要求:

- 1) 给出算法的基本设计思想。
 2) 根据设计思想, 采用 C 语言或 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时间复杂度和空间复杂度。

2.2.4 答案与解析

关注公众号【乘龙考研】
一手更新 稳定有保障

一、单项选择题

01. A

顺序表不像链表那样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。

02. A

本题易误选 B。注意, 存取方式是指读写方式。顺序表是一种支持随机存取的存储结构, 根据起始地址加上元素的序号, 可以很方便地访问任意一个元素, 这就是随机存取的概念。

03. B

顺序表所占的存储空间 = 表长 \times sizeof(元素的类型), 表长和元素的类型显然会影响存储空间的大小。若元素为结构体类型, 则元素中各字段的类型也会影响存储空间的大小。

04. D

题干实际要求能最快存取第 $i-1, i$ 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 $O(n)$; 只有顺序表可以按序号随机存取, 时间复杂度为 $O(1)$ 。

05. A

只有顺序表可以按序号随机存取, 且在最后进行插入和删除操作时不需要移动任何元素。

06. C

I 解析略; II 中, 在最后位置插入新结点不需要移动元素, 时间复杂度为 $O(1)$; III 中, 被删除结点后的结点需依次前移, 时间复杂度为 $O(n)$; IV 中, 需依次顺序访问每个元素, 时间复杂度相同。

07. C

对于 II, 顺序表仅需 3 次交换操作; 链表则需要分别找到两个结点前驱, 第 4 个结点断链后再插入到第 2 个结点后, 效率较低。对于 III, 需依次顺序访问每个元素, 时间复杂度相同。

08. C

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

09. C

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

线性表元素的序号是从 1 开始，而在第 $n+1$ 个位置插入相当于在表尾追加。

11. D

顺序存储需要连续的存储空间，在申请时需申请 $n+m$ 个连续的存储空间，然后将线性表原来的 n 个元素复制到新申请的 $n+m$ 个连续的存储空间的前 n 个单元。

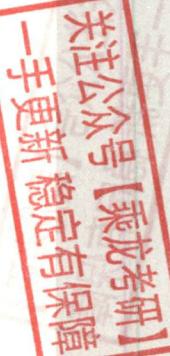
二、综合应用题

01. 【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的位置。

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
    //若删除成功，则返回 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; //假定 0 号元素的值最小
    for(int i=1;i<L.length;i++)
        if(L.data[i]<value){ //循环，寻找具有最小值的元素
            value=L.data[i];
            pos=i;
        }
    L.data[pos]=L.data[L.length-1]; //空出的位置由最后一个元素填补
    L.length--;
    return true; //此时，value 即为最小值
}
```



注意：本题也可用函数返回值返回，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）可以返回多个值。

02. 【解答】

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

本题代码如下：

```
void Reverse(Sqlist &L) {
    ElemType temp; //辅助变量
    for(int 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;
    }
}
```

03. 【解答】

解法 1：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），扫描时将不等于 x 的元素移动到下标 k 的位置，并更新 k 值。扫描结束后修改 L 的长度。

本题代码如下：

```
void del_x_1(Sqlist &L, ElemtType 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, i;                                //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++;
        }
    }
    L.length=k;                                //顺序表 L 的长度等于 k
}
```

解法 2：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。扫描结束后修改 L 的长度。

本题代码如下：

```
void del_x_2(Sqlist &L, ElemtType 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
}
```

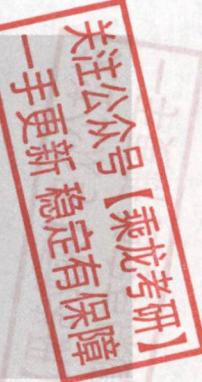
此外，本题还可以考虑设头、尾两个指针 ($i=1, j=n$)，从两端向中间移动，在遇到最左端值 x 的元素时，直接将最右端值非 x 的元素左移至值为 x 的数据元素位置，直到两指针相遇。但这种方法会改变原表中元素的相对位置。

04. 【解答】

在很多教材中（包括本书）指的“有序”，如无特别说明，通常是指“递增有序”。注意本题与上题的区别，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：先寻找值大于或等于 s 的第一个元素（第一个删除的元素），然后寻找值大于 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，只需直接将后面的元素前移。

本题代码如下：

```
bool Del_s_t2(Sqlist &L, ElemtType s, ElemtType 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++); //寻找值大于或等于 s 的第一个元素
    if(i>=L.length)
        return false;                                //所有元素值均小于 s，返回
    for(j=i; j<L.length && L.data[j]<=t; j++); //寻找值大于 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;
        return true;
    }
}

```

05. 【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，因此算法效率高。

本题代码如下：

```

bool Del_s_t(SqList &L, ElemtType s, ElemtType t) {
    //删除顺序表 L 中值在给定值 s 与 t (要求 s<t) 之间的所有元素
    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。

06. 【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视为非重复的有序表。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，若相同，则继续向后判断，若不同，则插入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：

```

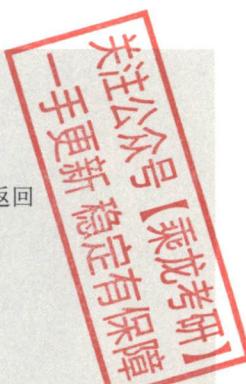
bool Delete_Same(SeqList & L) {
    if(L.length==0)
        return false;
    int i, j;                  //i 存储第一个不相同的元素，j 为工作指针
    for(i=0, j=1; j<L.length; j++)
        if(L.data[i]!=L.data[j]) //查找下一个与上个元素值不同的元素
            L.data[++i]=L.data[j]; //找到后，将元素前移
    L.length=i+1;
    return true;
}

```

对于本题的算法，请读者用序列 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 5 来手动模拟算法的执行过程，在模拟过程中要标注 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：使用散列表。)



07. 【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) {
    //将有序顺序表 A 与 B 合并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;
    return true;
}
```

注意：本算法的方法非常典型，需牢固掌握。

关注公众号【乘龙考研】
一手更新 稳定有保障

08. 【解答】

算法思想：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，即可得到 $(b_1, b_2, b_3, \dots, b_n, a_1, a_2, a_3, \dots, a_m)$ ，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 false;
    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);
}
```

09. 【解答】

算法思想：顺序存储的线性表递增有序，可以顺序查找，也可以折半查找。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
        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
    }
}
```

本题的算法也可写成三个函数：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：

可将问题视为把数组 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;

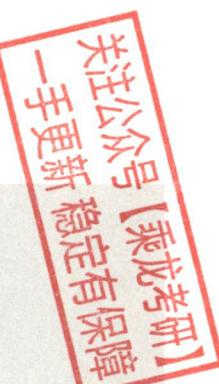
Reverse(0, n-1) 得到 defghabc;

注：在 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;}
}
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])
            return A[m1];           //满足条件①
        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){
                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];
}

```

关注公众号【乘龙考研】
一手更新 稳定有保障

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) 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) 算法实现如下：

```

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{
                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：本题如果采用先排好序再统计的方法 [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，花费大量时间去思考最优解法是得不偿失的。

13. 【解答】

1) 算法的基本设计思想：

要求在时间上尽可能高效，因此采用空间换时间的办法。分配一个用于标记的数组 B[n]，用来记录 A 中是否出现了 1~n 中的正整数，B[0] 对应正整数 1，B[n-1] 对应正整数 n，初始化 B 中全部为 0。由于 A 中含有 n 个整数，因此可能返回的值是 1~n+1，当 A 中 n 个数恰好为 1~n 时返回 n+1。当数组 A 中出现了大于或等于 0 或大于 n 的值时，会导致 1~n 中出现空余位置，返回结果必然在 1~n 中，因此对于 A 中出现了大于或等于 0 或大于 n 的值，可以不采取任何操作。

经过以上分析可以得出算法流程：从 A[0] 开始遍历 A，若 $0 < A[i] \leq n$ ，则令 $B[A[i]-1]=1$ ；否则不做操作。对 A 遍历结束后，开始遍历数组 B，若能查找到第一个满足 $B[i]==0$ 的下标 i，

返回 $i+1$ 即为结果，此时说明 A 中未出现的最小正整数在 $1 \sim n$ 之间。若 $B[i]$ 全部不为 0，返回 $i+1$ （跳出循环时 $i=n$, $i+1$ 等于 $n+1$ ），此时说明 A 中未出现的最小正整数是 $n+1$ 。

2) 算法实现：

```
int findMissMin(int A[], int n)
{
    int i,*B; //标记数组
    B=(int *)malloc(sizeof(int)*n); //分配空间
    memset(B, 0, sizeof(int)*n); //赋初值为 0
    for(i=0;i<n;i++)
        if(A[i]>0&&A[i]<=n) //若 A[i] 的值介于 1~n，则标记数组 B
            B[A[i]-1]=1;
    for(i=0;i<n;i++) //扫描数组 B，找到目标值
        if (B[i]==0) break;
    return i+1; //返回结果
}
```

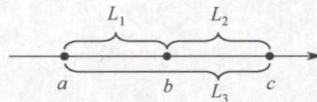
3) 时间复杂度：遍历 A 一次，遍历 B 一次，两次循环内操作步骤为 $O(1)$ 量级，因此时间复杂度为 $O(n)$ 。空间复杂度：额外分配了 $B[n]$ ，空间复杂度为 $O(n)$ 。

14. 【解答】

分析。由 $D = |a - b| + |b - c| + |c - a| \geq 0$ 有如下结论。

① 当 $a = b = c$ 时，距离最小。

② 其余情况。不失一般性，假设 $a \leq b \leq c$ ，观察下面的数轴：



$$L_1 = |a - b|$$

$$L_2 = |b - c|$$

$$L_3 = |c - a|$$

$$D = |a - b| + |b - c| + |c - a| = L_1 + L_2 + L_3 = 2L_3$$

由 D 的表达式可知，事实上决定 D 大小的关键是 a 和 c 之间的距离，于是问题就可以简化为每次固定 c 找一个 a ，使得 $L_3 = |c - a|$ 最小。

1) 算法的基本设计思想

① 使用 D_{\min} 记录所有已处理的三元组的最小距离，初值为一个足够大的整数。

② 集合 S_1 、 S_2 和 S_3 分别保存在数组 A、B、C 中。数组的下标变量 $i=j=k=0$ ，当 $i < |S_1|$ 、 $j < |S_2|$ 且 $k < |S_3|$ 时（ $|S|$ 表示集合 S 中的元素个数），循环执行下面的 a) ~c)。

a) 计算($A[i], B[j], C[k]$)的距离 D ；（计算 D ）

b) 若 $D < D_{\min}$ ，则 $D_{\min} = D$ ；（更新 D ）

c) 将 $A[i]$ 、 $B[j]$ 、 $C[k]$ 中的最小值的下标+1；（对照分析：最小值为 a ，最大值为 c ，这里 c 不变而更新 a ，试图寻找更小的距离 D ）

③ 输出 D_{\min} ，结束。

2) 算法实现：

```
#define INT_MAX 0x7fffffff
int abs_(int a){//计算绝对值
    if(a<0) return -a;
    else return a;
}
```

关注公众号【乘龙考研】
一手更新 稳定有保障

```

bool xls_min(int a,int b,int c){//a 是否是三个数中的最小值
    if(a<=b&&a<=c) return true;
    return false;
}
int findMinofTrip(int A[],int n,int B[],int m,int C[],int p){
    //D_min 用于记录三元组的最小距离, 初值赋为 INT_MAX
    int i=0,j=0,k=0,D_min=INT_MAX,D;
    while(i<n&&j<m&&k<p&&D_min>0){
        D=abs_(A[i]-B[j])+abs_(B[j]-C[k])+abs_(C[k]-A[i]); //计算 D
        if(D<D_min) D_min=D; //更新 D
        if(xls_min(A[i],B[j],C[k])) i++; //更新 a
        else if(xls_min(B[j],C[k],A[i])) j++;
        else k++;
    }
    return D_min;
}

```

3) 设 $n = (|S_1| + |S_2| + |S_3|)$, 时间复杂度为 $O(n)$, 空间复杂度为 $O(1)$ 。



2.3 线性表的链式表示

顺序表的存储位置可以用一个简单直观的公式表示, 它可以随机存取表中任意一个元素, 但插入和删除操作需要移动大量元素。链式存储线性表时, 不需要使用地址连续的存储单元, 即不要求逻辑上相邻的元素在物理位置上也相邻, 它通过“链”建立起元素之间的逻辑关系, 因此插入和删除操作不需要移动元素, 而只需修改指针, 但也会失去顺序表可随机存取的优点。

2.3.1 单链表的定义

线性表的链式存储又称单链表, 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。为了建立数据元素之间的线性关系, 对每个链表结点, 除存放元素自身的信息外, 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。

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;

```



图 2.3 单链表结点结构

利用单链表可以解决顺序表需要大量连续存储单元的缺点, 但单链表附加指针域, 也存在浪费存储空间的缺点。由于单链表的元素离散地分布在存储空间中, 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。

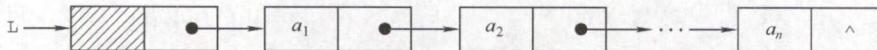


图 2.4 带头结点的单链表

头结点和头指针的区别：不管带不带头结点，头指针都始终指向链表的第一个结点，而头结点是带头结点的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 由于第一个数据结点的位置被存放在头结点的指针域中，因此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，其头指针都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得到了统一。

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。

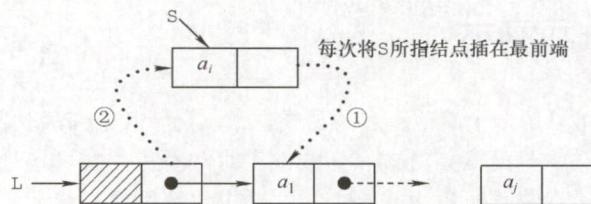
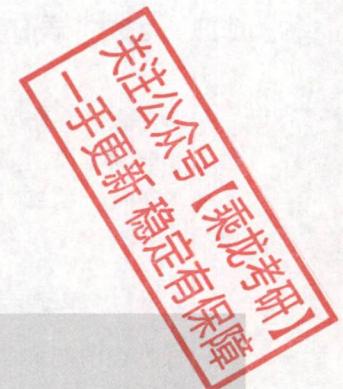


图 2.5 头插法建立单链表

头插法建立单链表的算法如下：

```
LinkList List_HeadInsert(LinkList &L) { //逆向建立单链表
    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①
        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);
    }
    return L;
}
```



采用头插法建立单链表时，读入数据的顺序与生成的链表中的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 ，则总时间复杂度为 $O(n)$ 。

思考一下：若没有设立头结点，则上述代码需要在哪些地方修改？^②

① malloc() 和 free() 是 C 语言的两个标准函数，执行 `s=(LNode*)malloc(sizeof(LNode))` 的作用是由系统生成一个 LNode 型的结点，同时将该结点的起始位置赋给指针变量 s。

② 主要修改处：因为在头部插入新结点，每次插入新结点后，需要将它的地址赋值给头指针 L。

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，则可采用尾插法。该方法将新结点插入到当前链表的表尾，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：

```
LinkList List_TailInsert(LinkList &L){ //正向建立单链表
    int x; //设元素类型为整型
    L=(LinkList)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;
}
```

关注公众号【乘龙考研】
一手更新 稳定有保障

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的相同。

3. 按序号查找结点

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(LinkList L,int i){
    if(i<1)
        return NULL; //若 i 无效，则返回 NULL
    int j=1; //计数，初始为 1
    LNode *p=L->next; //第 1 个结点指针赋给 p
    while(p!=NULL&&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，若 i 大于表长，则返回 NULL
}
```

按序号查找操作的时间复杂度为 $O(n)$ 。

4. 按值查找表结点

从单链表的第一个结点开始，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。

按值查找表结点的算法如下：

```
LNode *LocateElem(LinkList L, ElemtType e) {
    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

插入结点操作将值为 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。

算法首先调用按序号查找算法 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。

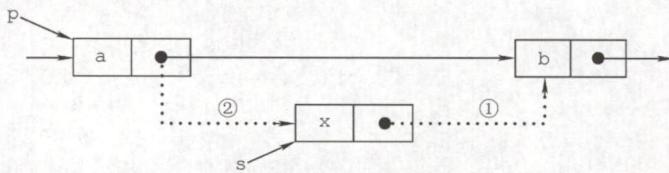


图 2.7 单链表的插入操作

实现插入结点的代码片段如下：

```
①p=GetElem(L, i-1); //查找插入位置的前驱结点
②s->next=p->next; //图 2.7 中操作步骤 1
③p->next=s; //图 2.7 中操作步骤 2
```

算法中，语句②和③的顺序不能颠倒，否则，先执行 $p->next=s$ 后，指向其原后继的指针就不存在，再执行 $s->next=p->next$ 时，相当于执行了 $s->next=s$ ，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，通常都采用后插操作。

以上面的算法为例，首先调用函数 GetElem() 找到第 $i-1$ 个结点，即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转化为后插操作，前提是单链表的头结点开始顺序查找到其前驱结点，时间复杂度为 $O(n)$ 。

此外，可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 $p->data$ 与 $s->data$ 交换，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

删除结点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，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。

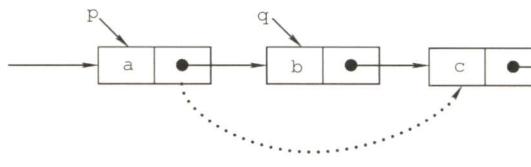


图 2.8 单链表结点的删除

假设结点 $*p$ 为找到的被删结点的前驱结点，为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。

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①
```

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

扩展：删除结点 $*p$ 。

要删除某个给定结点 $*p$ ，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，然后执行删除操作，算法的时间复杂度为 $O(n)$ 。

其实，删除结点 $*p$ 的操作可用删除 $*p$ 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;      //用后继结点的数据域覆盖
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。

注意：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，使得单链表只能从头结点依次顺序地向后遍历。

^① 执行 `free(q)` 的作用是由系统回收一个 `LNode` 型的结点，回收后的空间可供再次生成结点时用。

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。

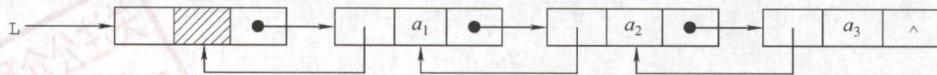


图 2.9 双链表示意图

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;
```

双链表在单链表的结点中增加了一个指向其前驱的 `prior` 指针，因此双链表中的按值查找和按位查找的操作与单链表的相同。但双链表在插入和删除操作的实现上，与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，其关键是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、删除操作的时间复杂度仅为 $O(1)$ 。

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。

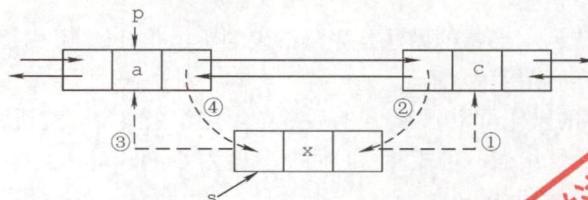


图 2.10 双链表插入结点过程

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;
```

上述代码的语句顺序不是唯一的，但也不是任意的，①和②两步必须在④步之前，否则 `*p` 的后继结点的指针就会丢掉，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。

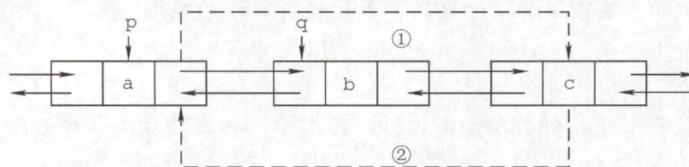


图 2.11 双链表删除结点过程

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。

在建立双链表的操作中，也可采用如同单链表的头插法和尾插法，但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

关注公众号【乘龙考研】
一手更新 稳定有保障

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。

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。

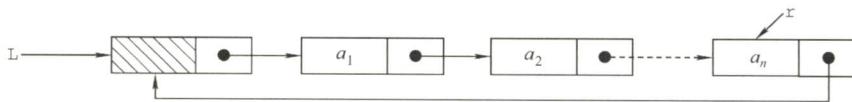


图 2.12 循环单链表

循环单链表的插入、删除算法与单链表的几乎一样，所不同的是若操作是在表尾进行，则执行的操作不同，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一个结点开始遍历整个链表。有时对循环单链表不设头指针而仅设尾指针，以使得操作效率更高。其原因是，若设的是头指针，对在表尾插入元素需要 $O(n)$ 的时间复杂度，而若设的是尾指针 r， $r->next$ 即为头指针，对在表头或表尾插入元素都只需要 $O(1)$ 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。

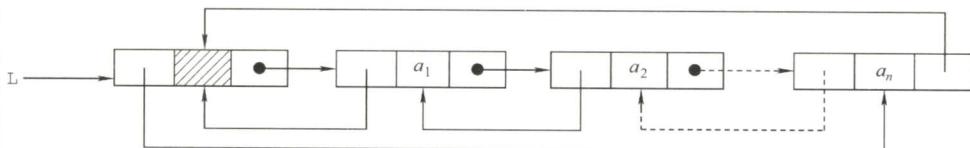


图 2.13 循环双链表

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

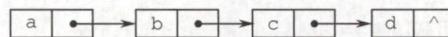
2.3.5 静态链表

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) 静态链表示例



关注公众号【乘龙考研】
一手更新 稳定有保障

(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 `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的相同，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，但在一些不支持指针的高级语言（如 Basic）中，这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（读写）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。例如在第 i 个位置上执行存或取的操作，顺序表仅需一次访问，而链表则需从表头开始依次访问 i 次。

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，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，物理存储位置不一定相邻，对应的逻辑关系是通过指针链接来表示的。

3. 查找、插入和删除操作

对于按值查找，顺序表无序时，两者的时间复杂度均为 $O(n)$ ；顺序表有序时，可采用折半查找，此时的时间复杂度为 $O(\log_2 n)$ 。

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，只需修改相关结点的指针域即可。由于链表的每个结点都带有指针域，故而存储密度不够大。

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，若再加入新元素，则会出现内存溢出，因此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，则会导致分配失败。链式存储的结点空间只在需要时申请分配，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

难以估计线性表的长度或存储规模时，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 $O(n)$ ，因此若经常做的运算是按序号访问数据元素，则显然顺序表优于链表。

在顺序表中进行插入、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；在链表中进行插入、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，而频繁进行插入、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

01. 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. 若频繁使用插入和删除结点操作，则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
02. 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
03. 对于顺序存储的线性表，其算法时间复杂度为 $O(1)$ 的运算应该是（ ）。
- A. 将 n 个元素从小到大排序
 - B. 删除第 i ($1 \leq i \leq n$) 个元素
 - C. 改变第 i ($1 \leq i \leq n$) 个元素的值
 - D. 在第 i ($1 \leq i \leq n$) 个元素后插入一个新元素
04. 下列关于线性表说法中，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 C. IV、V D. III、IV、V



05. 设线性表中有 $2n$ 个元素, () 在单链表上实现要比在顺序表上实现效率更高。
- 删除所有值为 x 的元素
 - 在最后一个元素的后面插入一个新元素
 - 顺序输出前 k 个元素
 - 交换第 i 个元素和第 $2n-i-1$ 个元素的值 ($i=0, \dots, n-1$)
06. 在一个单链表中, 已知 q 所指结点是 p 所指结点的前驱结点, 若在 q 和 p 之间插入结点 s , 则执行 ()。
- $s->next=p->next; p->next=s;$
 - $p->next=s->next; s->next=p;$
 - $q->next=s; s->next=p;$
 - $p->next=s; s->next=q;$
07. 给定有 n 个元素的一维数组, 建立一个有序单链表的最低时间复杂度是 ()。
- $O(1)$
 - $O(n)$
 - $O(n^2)$
 - $O(n \log_2 n)$
08. 将长度为 n 的单链表链接在长度为 m 的单链表后面, 其算法的时间复杂度采用大 O 形式表示应该是 ()。
- $O(1)$
 - $O(n)$
 - $O(m)$
 - $O(n+m)$
09. 单链表中, 增加一个头结点的目的是 ()。
- 使单链表至少有一个结点
 - 标识表结点中首结点的位置
 - 方便运算的实现
 - 说明单链表是线性表的链式存储
10. 在一个长度为 n 的带头结点的单链表 h 上, 设有尾指针 r , 则执行 () 操作与链表的表长有关。
- 删除单链表中的第一个元素
 - 删除单链表中的最后一个元素
 - 在单链表第一个元素前插入一个新元素
 - 在单链表最后一个元素后插入一个新元素
11. 对于一个头指针为 $head$ 的带头结点的单链表, 判定该表为空表的条件是 (); 对于不带头结点的单链表, 判定空表的条件为 ()。
- $head==NULL$
 - $head->next==NULL$
 - $head->next==head$
 - $head!=NULL$
12. 下面关于线性表的一些说法中, 正确的是 ()。
- 对一个设有头指针和尾指针的单链表执行删除最后一个元素的操作与链表长度无关
 - 线性表中每个元素都有一个直接前驱和一个直接后继
 - 为了方便插入和删除数据, 可以使用双链表存放数据
 - 取线性表第 i 个元素的时间与 i 的大小有关
13. 在双链表中向 p 所指的结点之前插入一个结点 q 的操作为 ()。
- $p->prior=q; q->next=p; p->prior->next=q; q->prior=p->prior;$
 - $q->prior=p->prior; p->prior->next=q; q->next=p; p->prior=q->next;$
 - $q->next=p; p->next=q; q->prior->next=q; q->next=p;$
 - $p->prior->next=q; q->next=p; q->prior=p->prior; p->prior=q;$
14. 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 ()。
- $p->prior->next=p->next; p->next->prior=p->prior;$
 - $p->prior=p->prior->prior; p->prior->next=p;$
 - $p->next->prior=p; p->next=p->next->next;$

关注公众号【乘龙考研】
一手更新 稳定有保障

- D. $p \rightarrow next = p \rightarrow prior \rightarrow prior; p \rightarrow prior = p \rightarrow next \rightarrow next;$
15. 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. $O(n \log_2 n)$
16. 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
17. 带头结点的双循环链表 L 为空的条件是（ ）。
- A. $L \rightarrow prior == L \& \& L \rightarrow next == NULL$
B. $L \rightarrow prior == NULL \& \& L \rightarrow next == NULL$
C. $L \rightarrow prior == NULL \& \& L \rightarrow next == L$
D. $L \rightarrow prior == L \& \& L \rightarrow next == L$
18. 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
19. 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. 既有头结点指针又有尾结点指针的循环单链表
20. 一个链表最常用的操作是在最后一个元素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
21. 静态链表中指针表示的是（ ）。
- A. 下一元素的地址 B. 内存存储器地址
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
22. 需要分配较大空间，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
23. 某线性表用带头结点的循环单链表存储，头指针为 $head$ ，当 $head \rightarrow next \rightarrow next == head$ 成立时，线性表长度可能是（ ）。
- A. 0 B. 1 C. 2 D. 可能为 0 或 1
24. 【2016 统考真题】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
------	------	------

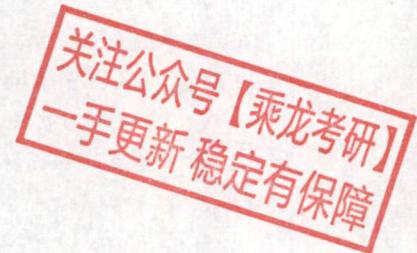
，其中 $prev$ 和 $next$ 分别是指向其直接前驱和直接后继结点的指针。现要删除指针 p 所指向的结点，正确的语句序列是（ ）。
- A. $p \rightarrow next \rightarrow prev = p \rightarrow prev; p \rightarrow prev \rightarrow next = p \rightarrow next; free(p);$
B. $p \rightarrow next \rightarrow prev = p \rightarrow next; p \rightarrow prev \rightarrow next = p \rightarrow prev; free(p);$
C. $p \rightarrow next \rightarrow prev = p \rightarrow next; p \rightarrow prev \rightarrow next = p \rightarrow prev; free(p);$

关注公众号【乘龙考研】
一手更新 稳定有保障

D. `p->next->prev=p->prev; p->prev->next=p->next; free(p);`

25. 【2016 统考真题】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 放于 1014H 处并插入单链表, 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

26. 【2021 统考真题】已知头指针 h 指向一个带头结点的非空单循环链表, 结点结构为 `[data] next`, 其中 next 是指向直接后继结点的指针, p 是尾指针, q 是临时指针。现要删除该链表的第一个元素, 正确的语句序列是 ()。

- A. `h->next=h->next->next; q=h->next; free(q);`
 B. `q=h->next; h->next=h->next->next; free(q);`
 C. `q=h->next; h->next=q->next; if(p!=q) p=h; free(q);`
 D. `q=h->next; h->next=q->next; if(p==q) p=h; free(q);`

二、综合应用题

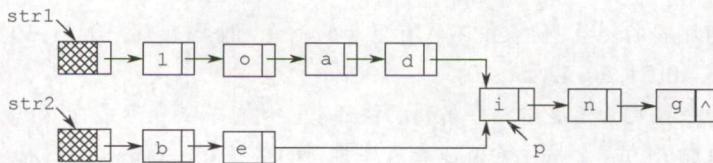
01. 设计一个递归算法, 删除不带头结点的单链表 L 中所有值为 x 的结点。
02. 在带头结点的单链表 L 中, 删除所有值为 x 的结点, 并释放其空间, 假设值为 x 的结点不唯一, 试编写算法以实现上述操作。
03. 设 L 为带头结点的单链表, 编写算法实现从尾到头反向输出每个结点的值。
04. 试编写在带头结点的单链表 L 中删除一个最小值结点的高效算法 (假设最小值结点是唯一的)。
05. 试编写算法将带头结点的单链表就地逆置, 所谓“就地”是指辅助空间复杂度为 O(1)。
06. 有一个带头结点的单链表 L, 设计一个算法使其元素递增有序。
07. 设在一个带表头结点的单链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)。
08. 给定两个单链表, 编写算法找出两个链表的公共结点。
09. 给定一个带表头结点的单链表, 设 head 为头指针, 结点结构为 `(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)。
10. 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。
11. 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 $A = \{a_1, a_2, \dots, a_n\}$, $B = \{b_n, \dots, b_2, b_1\}$ 。
12. 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) 将变为(7, 10, 21, 30, 42, 51, 70)。

13. 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。
14. 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 中的公共元素产生单链表 C ，要求不破坏 A 、 B 的结点。
15. 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 $h1$ 和 $h2$ ，编写一个函数将链表 $h2$ 链接到链表 $h1$ 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 pre （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. 单链表有环，是指单链表的最后一个结点的指针指向了链表中的某个结点（通常单链表的最后一个结点的指针域是空的）。试编写算法判断单链表是否存在环。
 - 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时间复杂度和空间复杂度。
22. 【2009 统考真题】已知一个带有表头结点的单链表，结点结构为

data	link

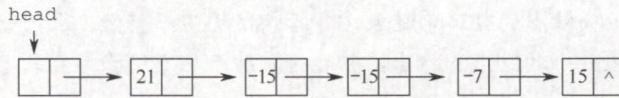
 假设该链表只给出了头指针 $list$ 。在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第 k 个位置上的结点（ k 为正整数）。若查找成功，算法输出该结点的 $data$ 域的值，并返回 1；否则，只返回 0。要求：
 - 1) 描述算法的基本设计思想。
 - 2) 描述算法的详细实现步骤。
 - 3) 根据设计思想和实现步骤，采用程序设计语言描述算法（使用 C、C++ 或 Java 语言实现），关键之处请给出简要注释。
23. 【2012 统考真题】假定采用带头结点的单链表保存单词，当两个单词有相同的后缀时，可共享相同的后缀存储空间，例如，loading 和 being 的存储映像如下图所示。



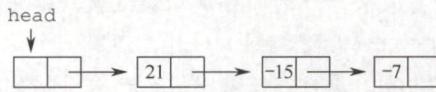
设 str1 和 str2 分别指向两个单词所在单链表的头结点，链表结点结构为 $[\text{data} \mid \text{next}]$ ，请设计一个时间上尽可能高效的算法，找出由 str1 和 str2 所指向两个链表共同后缀的起始位置（如图中字符 i 所在结点的位置 p ）。要求：

- 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。

24. 【2015 统考真题】用单链表保存 m 个整数，结点的结构为 $[\text{data}] [\text{link}]$ ，且 $|\text{data}| \leq n$ (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。

25. 【2019 统考真题】设线性表 $L = (a_1, a_2, a_3, \dots, a_{n-2}, a_{n-1}, a_n)$ 采用带头结点的单链表保存，链表中的结点定义如下：

```
typedef struct node
{
    int data;
    struct node*next;
} NODE;
```

请设计一个空间复杂度为 $O(1)$ 且时间上尽可能高效的算法，重新排列 L 中的各结点，得到线性表 $L' = (a_1, a_n, a_2, a_{n-1}, a_3, a_{n-2}, \dots)$ 。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) 说明你所设计的算法的时间复杂度。

2.3.8 答案与解析

一、单项选择题

01. B

两种存储结构有不同的适用场合，不能简单地说谁好谁坏，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移动大量元素，效率较低，III 的描述刚好相反。顺序存储结构既能随机存取又能顺序存取，而链式结构只能进行顺序存取，IV 正确。

02. B

首先直接排除 A 和 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。链式存储能方便地表示各种逻辑关系，且插入和删除操作的时间复杂度为 $O(1)$ 。

03. C

对 n 个元素进行排序的时间复杂度最小也要 $O(n)$ （初始有序时），通常为 $O(n \log_2 n)$ 或 $O(n^2)$ ，通过第 8 章学习后会更理解。B 和 D 显然错误。顺序表支持按序号的随机存取（读写）方式。

04. D

顺序存储方式同样适合图和树，I 错误。线性表采用顺序存储时 II 错误。III 是静态链表的特点。有序单链表只能依次查找插入位置，时间复杂度为 $O(n)$ ，IV 正确。队列需要在表头删除元素，表尾插入元素，采用带尾指针的循环链表较为方便，插入和删除的时间复杂度都为 $O(1)$ ，V 正确。

05. A

对于 A，在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，因此在单链表上实现效率更高。对于 B 和 D，顺序表的效率更高。C 无区别。

06. C

s 插入后，q 成为 s 的前驱，而 p 成为 s 的后继，选 C。

注意：可能会有读者认为选项 C 中的两条语句交换才正确。实际上，因为本题插入位置的前后结点都有指针指示（这与前面介绍的插入操作是不同的），所以选项 C 中的语句顺序并不会造成断链。在此也提醒读者在学习过程中一定要多动脑思考，而不要生搬硬套。

关注公众号【乘龙考研】
一手更新 稳定有保障

07. D

若先建立链表，然后依次插入建立有序表，则每插入一个元素就需遍历链表寻找插入位置，即直接插入排序，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 $O(n)$ ，数组排序的最好时间复杂度为 $O(n \log_2 n)$ ，总时间复杂度为 $O(n \log_2 n)$ 。故选 D。

08. C

先遍历长度为 m 的单链表，找到该单链表的尾结点，然后将其 next 域指向另一个单链表的首结点，其时间复杂度为 $O(m)$ 。

09. C

单链表设置头结点的目的是方便运算的实现，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就统一了，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置其前驱结点的指针域为 NULL，需要从头开始依次遍历找到该前驱结点，需要 $O(n)$ 的时间，与表长有关。其他操作均与表长无关，读者可自行模拟。

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 域指向第一个元素结点，`head->next==NULL` 表示该单链表为空。在不带头结点的单链表中，head 直接指向第一个元素结点，`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。

13. 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。

14. 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。关键是要保证在结点指针的修改过程中不断链！

注意，请读者仔细对比上述两题，弄清双链表的插入和删除方法。

15. B

设单链表递增有序，首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。查找的时间复杂度为 $O(n)$ ，插入的时间复杂度为 $O(1)$ ，总时间复杂度为 $O(n)$ 。

16. D

在插入和删除操作上，单链表和双链表都不用移动元素，都很方便，但双链表修改指针的操作更为复杂，A 错误。双链表中可以快速访问任何一个结点的前驱和后继结点，D 正确。

17. 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。

18. 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。而寻找尾结点及尾结点的前驱结点时，只有带头结点的双循环链表所需要的时间最少。

19. 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 $O(1)$ 。

20. C

对于 A，在最后一个元素之后插入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），需要先遍历整个链表找到尾结点，再做删除操作，时间复杂度为 $O(n)$ 。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，有尾结点的指针，省去了遍历链表的过程，因此时间复杂度均为 $O(1)$ 。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 $O(1)$ 。

21. 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。

22. 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。

23. D

对一个空循环单链表，有 $head->next==head$ ，推理 $head->next->next==head->$

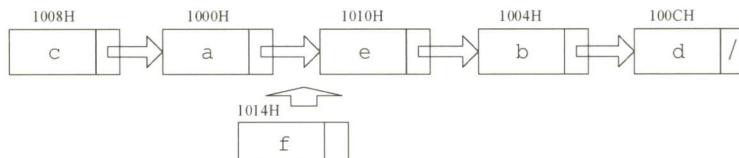
`next==head`。对含有 1 个元素的循环单链表，头结点（头指针 `head` 指示）的 `next` 域指向该唯一元素结点，该元素结点的 `next` 域指向头结点，因此也有 `head->next->next=head`。故选 D。

24. D

选项 A 第二句代码，相当于将 `p` 前驱结点的后继指针指向其自身，错误；选项 B 和 C 的第一句代码，相当于将 `p` 后继结点的前驱指针指向其自身，错误。只有 D 正确。

25. 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`。

26. D

如图 1 所示，要删除带头结点的非空单循环链表中的第一个元素，就要先用临时指针 `q` 指向待删结点，`q=h->next`；然后将 `q` 从链表中断开，`h->next=q->next`（这一步也可写成 `h->next=h->next->next`）；此时要考虑一种特殊情况，若待删结点是链表的尾结点，即循环单链表中只有一个元素（`p` 和 `q` 指向同一个结点），如图 2 所示，则在删除后要将尾指针指向头结点，即 `if(p==q) p=h`；最后释放 `q` 结点即可，答案选 D。

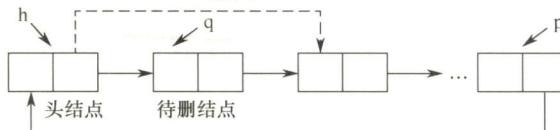


图 1

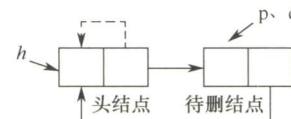


图 2

二、综合应用题

01. 【解答】

设 `f(L, x)` 的功能是删除以 `L` 为头结点指针的单链表中所有值等于 `x` 的结点，显然有 `f(L->next, x)` 的功能是删除以 `L->next` 为头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)=` 不做任何事情； 若 `L` 为空表

递归主体：`f(L, x)=` 删除`*L` 结点；`f(L->next, x)`； 若 `L->data==x`

`f(L, x)=f(L->next, x)`； 其他情况

本题代码如下：

```
void Del_X_3(Linklist &L, ElemtType x) {
    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); // 递归调用
        Del_X_3(L, x);
    }
}
```



```

    }
    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 $O(n)$ 。有读者认为直接去掉 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的，因此不会断链。

02. 【解答】

解法 1：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, ElemenType x) {
    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 指向被删结点
            p=p->next;
            pre->next=p; //将*q 结点从链表中断开
            free(q); //释放*q 结点的空间
        }
        else { //否则，pre 和 p 同步后移
            pre=p;
            p=p->next;
        }
    }
}

```

关注公众号【乘龙考研】
一手更新 稳定有保障

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 2：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, ElemenType x) {
    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
        }
    }
}

```

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 $O(1)$ 。

03. 【解答】

考虑到从头到尾输出比较简单。求解本题时，我们会很自然地想到借助上题中的链表逆置法，改变链表的方向，然后就可从头到尾实现反向输出。

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

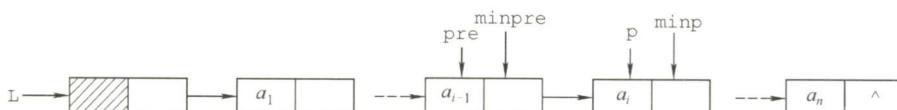
既然能用栈的思想解决，我们也就很自然地联想到了用递归来实现。每当访问一个结点时，先递归输出它后面的结点，再输出该结点自身，这样链表就反向输出了，如右图所示。

本题代码如下：

```
void R_Print(LinkList L) {
    if (L->next!=NULL) {
        R_Print(L->next);           //递归
    }
    if (L!=NULL) print(L->data);   //输出函数
}
void R_Ignore_Head(LinkList L) {
    if (L->next!=NULL) R_Print(L->next);
}
```

04. 【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 $p->data$ 小于 $minp->data$ ，则将 p 、 pre 分别赋值给 $minp$ 、 $minpre$ ，如下图所示。当 p 扫描完毕时， $minp$ 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：

```
LinkList Delete_Min(LinkList &L) {
    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;
}

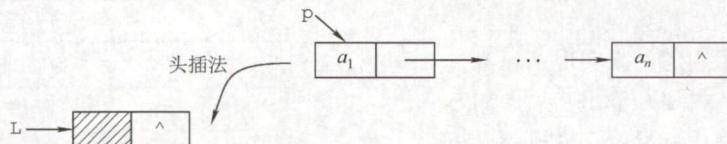
```

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，请读者自行思考。

05. 【解答】

解法 1：将头结点摘下，然后从第一结点开始，依次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，这样就实现了链表的逆置，如下图所示。



本题代码如下：

```

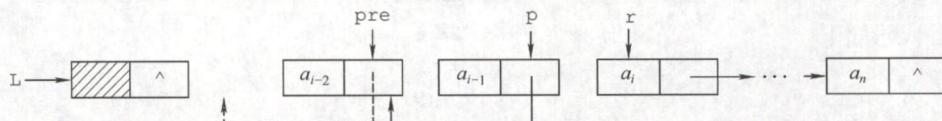
LinkList Reverse_1(LinkList L) {
    LNode *p, *r;
    p=L->next;
    L->next=NULL;
    while(p!=NULL) {
        r=p->next;
        p->next=L->next;
        L->next=p;
        p=r;
    }
    return L;
}

```

关注公众号【乘龙考研】
一手更新 稳定有保障

解法 2：大部分辅导书都只介绍解法 1，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。假设经过若干操作后， $*pre$ 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， $*p$ 的后继结点的链就会断开，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) {
    LNode *pre,*p=L->next,*r=p->next;
    p->next=NULL;
    while(r!=NULL) {
        pre=p;
        p=r;
        r=p->next;
    }
}

```

```

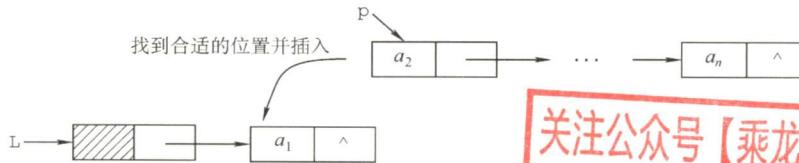
        r=r->next;
        p->next=pre;           //指针反转
    }
    L->next=p;             //处理最后一个结点
    return L;
}

```

上述两个算法的时间复杂度为 $O(n)$, 空间复杂度为 $O(1)$ 。

06. 【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。



关注公众号【乘龙考研】
一手更新 稳定有保障

本题代码如下：

```

void Sort(LinkList &L) {
    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
    }
}

```

细心的读者会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中，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 $O(n\log_2 n)$ ，显然这是以空间换时间的策略。

07. 【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;
    }
}

```

08. 【解答】

两个单链表有公共结点，即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。

本题极容易联想到“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，若找到两个相同的结点，则找到了它们的公共结点。显然，该算法的时间复杂度为 $O(\text{len1} \times \text{len2})$ 。

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：若两个链表有一个公共结点，则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时，只需要分别遍历两个链表到最后一个结点。若两个尾结点是一样的，则说明它们有公共结点，否则两个链表没有公共结点。

然而，在上面的思路中，顺序遍历两个链表到尾结点时，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。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。

根据这一思路中，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){
    int dist;
    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--) //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;
        }
    }
}

```

```

    }
} //while
return NULL;
}

```

09. 【解答】

算法思想：对链表进行遍历，在每次遍历中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 $O(n^2)$ 。

本题代码如下：

```

void Min_Delete(LinkList &head) {
    while(head->next!=NULL) {           //循环到仅剩头结点
        LNode *pre=head;                //pre 为元素最小值结点的前驱结点的指针
        LNode *p=pre->next;             //p 为工作指针
        LNode *u;                      //指向被删除结点
        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);
    }
    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中，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。

本题代码如下：

```

LinkList DisCreate_1(LinkList &A) {
    int i=0;                           //i 记录表 A 中结点的序号
    LinkList B=(LinkList)malloc(sizeof(LNode)); //创建 B 表表头
    B->next=NULL;                    //B 表的初始化
    LNode *ra=A,*rb=B,*p;            //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++;
        if(i%2==0) {
            rb->next=p;
            rb=p;
        }
        else{
            ra->next=p;
            ra=p;
        }
    }
}

```



```

    }
    p=p->next;
}
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 中并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。

本题代码如下：

```

LinkList DisCreat_2(LinkList &A) {
    LinkList B=(LinkList)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;
        if(p!=NULL) {
            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，若不设变量保存其后继结点，则会引起断链，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。

本题代码如下：

```

void Del_Same(LinkList &L) {
    LNode *p=L->next,*q; //p 为扫描工作指针
    if(p==NULL)
        return;
    while(p->next!=NULL) {
        q=p->next;
        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, 依次与已经插入结点的链表的最后一个结点比较, 若不等则直接插入, 否则将当前遍历的结点删除并处理下一个结点, 直到最后一个结点为止。

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。

本题代码如下:

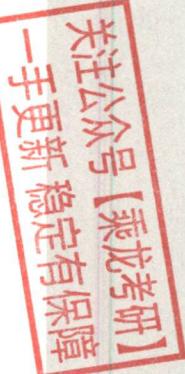
```

void MergeList(LinkList &La, LinkList &Lb) {
    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
                      //链表初始化为空
    while(pa&&pb) //当两链表均不为空时, 循环
        if(pa->data<=pb->data) {
            r=pa->next; //r 暂存 pa 的后继结点指针
            pa->next=La->next;
            La->next=pa;
            La->next=r; //将 pa 结点链于结果表中, 同时逆置
                           //（头插法）
            pa=r; //恢复 pa 为当前待比较结点
        }
        else{
            r=pb->next; //r 暂存 pb 的后继结点指针
            pb->next=La->next;
            La->next=pb;
            pb=r; //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
        }
    if(pa) //通常情况下会剩一个链表非空, 处理
        pb=pa; //剩下的部分
    while(pb){ //处理剩下的一个非空链表
        r=pb->next;
        pb->next=La->next;
        La->next=pb;
        pb=r;
    }
    free(Lb);
}

```

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结点, 使用尾插法插入到新的链表中, 并将两个原表指针后移一位, 直到其中一个链表遍历到表尾。



本题代码如下：

```

void Get_Common(LinkList A,LinkList B){
    LNode *p=A->next,*q=B->next,*r,*s;
    LinkList C=(LinkList)malloc(sizeof(LinkList)); //建立表 C
    r=C; //r 始终指向 C 的尾结点
    while(p!=NULL&&q!=NULL){ //循环跳出条件
        if(p->data<q->data)
            p=p->next; //若 A 的当前元素较小，后移指针
        else if(p->data>q->data)
            q=q->next; //若 B 的当前元素较小，后移指针
        else{ //找到公共元素结点
            s=(LNode*)malloc(sizeof(LNode));
            s->data=p->data; //复制产生结点*s
            r->next=s; //将*s 链接到 C 上（尾插法）
            r=s;
            p=p->next; //表 A 和 B 继续向后扫描
            q=q->next;
        }
    }
    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合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。

本题代码如下：

```

LinkList Union(LinkList &la,LinkList &lb){
    LNode *pa=la->next; //设工作指针分别为 pa 和 pb
    LNode *pb=lb->next;
    LNode *u,*pc=la; //结果表中当前合并结点的前驱指针 pc
    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;
}

```

链表归并类型的试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】

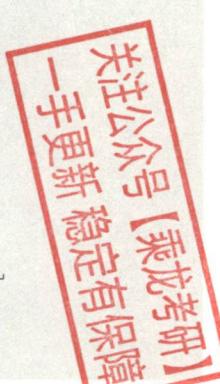
算法思想：因为两个整数序列已存入两个链表中，操作从两个链表的第一个结点开始，若对应数据相等，则后移指针；若对应数据不等，则 A 链表从上次开始比较结点的后继开始，B 链表仍从第一个结点开始比较，直到 B 链表到尾表示匹配成功。A 链表到尾而 B 链表未到尾表示失败。操作中应记住 A 链表每次的开始结点，以便下次匹配时好从其后继开始。

本题代码如下：

```

int Pattern(LinkList A,LinkList B) {
    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
    }
}

```



注意：该题其实是字符串模式匹配的链式表示形式，读者应该结合字符串模式匹配的内容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个数为奇数时) 或相邻 ($p->next=q$ 或 $q->prior=p$, 当循环双链表中结点个数为偶数时) 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。

本题代码如下：

```

int Symmetry(DLinkList L) {
    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
}

```

注意：while 循环第二个判断条件易误写成 $p->next!=q$ ，分析这样会产生什么问题。

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。

本题代码如下：

```

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。

本题代码如下：

```

void Del_All(LinkList &L){
    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){ //找到值更小的结点
                minp=p;
                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
}

```

关注公众号【乘龙考研】
一手更新 稳定有保障

```

        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的频度大的结点，插入位置为该结点之后），并插入到该位置。

本题代码如下：

```

DLinkList Locate(DLinkList &L, ElemtType x) {
    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)
        exit(0);                  //不存在值为 x 的结点
    else{
        p->freq++;              //令元素值为 x 的结点的 freq 域加 1
        if(p->pre==L||p->pre->freq>p->freq)
            return p;             //p 是链表首结点, 或 freq 值小于前驱
        if(p->next!=NULL) p->next->pre=p->pre;
        p->pre->next=p->next;    //将 p 结点从链表上摘下
        q=p->pre;                //以下查找 p 结点的插入位置
        while(q!=L&&q->freq<=p->freq)
            q=q->pre;
        p->next=q->next;
        if(q->next!=NULL) q->next->pre=p; //将 p 结点排在同频率的第一个
        p->pre=q;
        q->next=p;
    }
    return p;                      //返回值为 x 的结点的指针
}

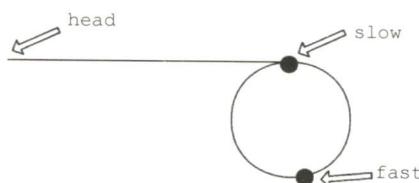
```

21. 【解答】

1) 算法的基本设计思想

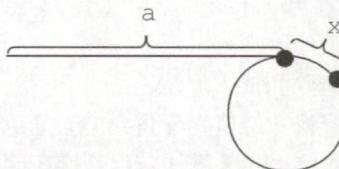
设置快慢两个指针分别为 $fast$ 和 $slow$ ，初始时都指向链表头 $head$ 。 $slow$ 每次走一步，即 $slow=slow->next$ ； $fast$ 每次走两步，即 $fast=fast->next->next$ 。由于 $fast$ 比 $slow$ 走得快，如果有环，那么 $fast$ 一定会先进入环，而 $slow$ 后进入环。当两个指针都进入环后，经过若干操作后两个指针定能在环上相遇。这样就可以判断一个链表是否有环。

如下图所示，当 $slow$ 刚进入环时， $fast$ 早已进入环。因为 $fast$ 每次比 $slow$ 多走一步且 $fast$ 与 $slow$ 的距离小于环的长度，所以 $fast$ 与 $slow$ 相遇时， $slow$ 所走的距离不超过环的长度。



关注公众号
一手更新 稳定有保障
【乘龙考研】

如下图所示，设头结点到环的入口点的距离为 a ，环的入口点沿着环的方向到相遇点的距离为 x ，环长为 r ，相遇时 fast 绕过了 n 圈。



则有 $2(a+x) = a + n \cdot r + x$ ，即 $a = n \cdot r - x$ 。显然从头结点到环的入口点的距离等于 n 倍的环长减去环的入口点到相遇点的距离。因此可设置两个指针，一个指向 head，一个指向相遇点，两个指针同步移动（均为一次走一步），相遇点即为环的入口点。

2) 本题代码如下：

```
LNode* FindLoopStart(LNode *head) {
    LNode *fast=head, *slow=head;      //设置快慢两个指针
    while(fast!=NULL&&fast->next!=NULL) {
        slow=slow->next;           //每次走一步
        fast=fast->next->next;     //每次走两步
        if(slow==fast) break;       //相遇
    }
    if(fast==NULL||fast->next==NULL)   //没有环，返回 NULL
        return NULL;
    LNode *p1=head, *p2=slow;          //分别指向开始点、相遇点
    while(p1!=p2) {
        p1=p1->next;
        p2=p2->next;
    }
    return p1;                         //返回入口点
}
```

3) 当 fast 与 slow 相遇时，slow 肯定没有遍历完链表，故算法的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

22. 【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，通过链表的一次遍历，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 $count=0$ ， p 和 q 指向链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， $count=count+1$ 。
- 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) 算法实现如下:

```

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){ //指针 p、q 指示第一个结点
    LNode *p=list->link,*q=list->link; //指针 p、q 指示第一个结点
    int count=0; //计数，若 count<k 只移动 p
    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; //之后让 p、q 同步移动
        p=p->link;
    } //while
    if(count<k) //查找失败返回 0
        return 0;
    else { //否则打印并返回 1
        printf("%d",q->data);
        return 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, 最高分为 10 分。若采用递归算法得到正确结果, 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。

23. 【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。

1) 算法的基本设计思想如下:

- ① 分别求出 str1 和 str2 所指的两个链表的长度 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
    }
}

```

3) 时间复杂度为 $O(\text{len1} + \text{len2})$ 或 $O(\max(\text{len1}, \text{len2}))$, 其中 len1 、 len2 分别为两个链表的长度。

24. 【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 $|\text{data}| \leq n$ ，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 $q[|\text{data}|]$ 的值，若为 0 则保留该结点，并令 $q[|\text{data}|] = 1$ ；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) 算法实现如下:

```

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 $O(n)$ 。

25. 【解答】

1) 算法的基本设计思想

先观察 $L = (a_1, a_2, a_3, \dots, a_{n-2}, a_{n-1}, a_n)$ 和 $L' = (a_1, a_n, a_2, a_{n-1}, a_3, a_{n-2}, \dots)$, 发现 L' 是由 L 摘取第一个元素, 再摘取倒数第一个元素……依次合并而成的。为了方便链表后半段取元素, 需要先将 L 后半段原地逆置 [题目要求空间复杂度为 $O(1)$, 不能借助栈], 否则每取最后一个结点都需要遍历一次链表。①先找出链表 L 的中间结点, 为此设置两个指针 p 和 q , 指针 p 每次走一步, 指针 q 每次走两步, 当指针 q 到达链尾时, 指针 p 正好在链表的中间结点; ②然后将 L 的后半段结点原地逆置。③从单链表前后两段中依次各取一个结点, 按要求重排。

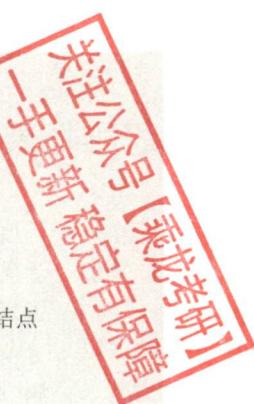
2) 算法实现

```

void change_list(NODE*h) {
    NODE *p, *q, *r, *s;
    p=q=h;
    while(q->next!=NULL) { // 寻找中间结点
        p=p->next; // p 走一步
        q=q->next;
        if(q->next!=NULL) q=q->next; // q 走两步
    }
    q=p->next; // p 所指结点为中间结点, q 为后半段链表的首结点
    p->next=NULL;
    while(q!=NULL) { // 将链表后半段逆置
        r=q->next;
        q->next=p->next;
        p->next=q;
        q=r;
    }
    s=h->next; // s 指向前半段的第一个数据结点, 即插入点
    q=p->next; // q 指向后半段的第一个数据结点
    p->next=NULL;
    while(q!=NULL) { // 将链表后半段的结点插入到指定位置
        r=q->next; // r 指向后半段的下一个结点
        q->next=s->next; // 将 q 所指结点插入到 s 所指结点之后
        s->next=q;
        s=q->next; // s 指向前半段的下一个插入点
        q=r;
    }
}

```

3) 第1步找中间结点的时间复杂度为 $O(n)$, 第2步逆置的时间复杂度为 $O(n)$, 第3步合并链表的时间复杂度为 $O(n)$, 所以该算法的时间复杂度为 $O(n)$ 。



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的代码量一般都比较少, 又具有一

定的算法设计技巧，因此适合笔试考查。考研题中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。

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。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：对于算法设计题，如果能写出数据结构类型的定义、正确的算法思想，那么至少会給一半的分数，如果能用伪代码写出自然更好，比較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...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，则问题便迎刃而解。先用一种时间复杂度为 $O(n \log_2 n)$ 的排序算法将 A[1...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 $i \geq j$ 停止。）

请读者思考本题是否有其他求解算法。

关注公众号【乘龙考研】
一手更新 稳定有保障

第3章 栈、队列和数组



兑换后

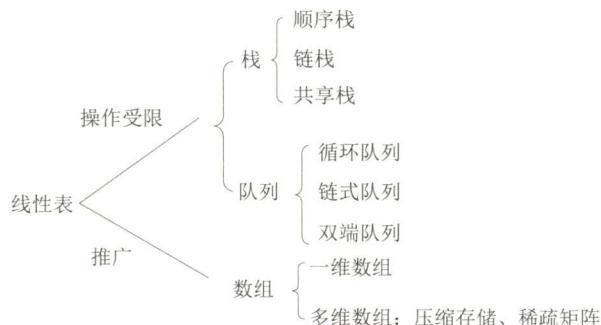


看视频讲解

【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) 多维数组的存储
- (五) 特殊矩阵的压缩存储
- (六) 栈、队列和数组的应用

【知识框架】



【复习提示】

本章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点。由于它们均是线性表的应用和推广，因此也容易出现在算法设计题中。此外，栈和队列的顺序存储、链式存储及其特点、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）是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。

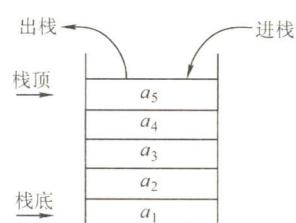


图 3.1 栈的示意图

栈顶 (Top)。线性表允许进行插入删除的那一端。

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，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，栈的操作特性可以明显地概括为后进先出 (Last In First Out, LIFO)。

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。

栈的数学性质： n 个不同元素进栈，出栈元素不同排列的个数为 $\frac{1}{n+1} C_{2n}^n$ 。上述公式称为卡特兰 (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教材。

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 为空则返回 `true`，否则返回 `false`。

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

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

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

`DestroyStack(&S)`：销毁栈，并释放栈 S 占用的存储空间（“`&`”表示引用调用）。

在解答算法题时，若题干未做出限制，则可直接使用这些基本的操作函数。

3.1.2 栈的顺序存储结构

栈是一种操作受限的线性表，类似于线性表，它也有对应的两种存储方式。

1. 顺序栈的实现

采用顺序存储的栈称为顺序栈，它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`) 指示当前栈顶元素的位置。

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 {
    Elemtypete 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。

注意：栈和队列的判空、判满条件，会因实际给的条件不同而变化，上面提到的方法以及下面的代码实现只是在栈顶指针设定的条件下的相应方法，而其他情况则需具体问题具体分析。

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 的相继出栈, 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, 元素 C、D、E 已不在栈中, 读者应通过该示意图深刻理解栈顶指针的作用。

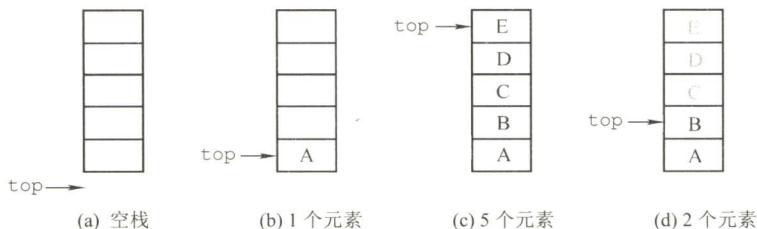


图 3.2 栈顶指针和栈中元素之间的关系

下面是顺序栈上常用的基本运算的实现。

(1) 初始化

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

(2) 判栈空

```
bool StackEmpty(SqStack 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;
}
```

当栈不满时, top 先加 1, 再入栈。若初始时将 top 定义为 0, 函数 3 和 4 应如何改写?

(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;
}
```

关注公众号【乘龙考研】
一手更新 稳定有保障

仅为读取栈顶元素，并没有出栈操作，因此原栈顶元素依然保留在栈中。

注意：这里 `top` 指向的是栈顶元素，所以进栈操作为 `S.data[++S.top]=x`，出栈操作为 `x=S.data[S.top--]`。若栈顶指针初始化为 `S.top=0`，即 `top` 指向栈顶元素的下一位置，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，做题时要灵活应变。

3. 共享栈

利用栈底位置相对不变的特性，可让两个顺序栈共享一个一维数组空间，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。

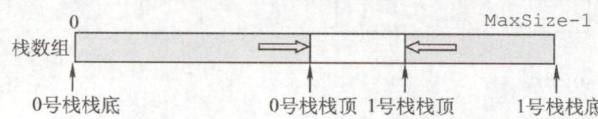


图 3.3 两个顺序栈共享存储空间

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。

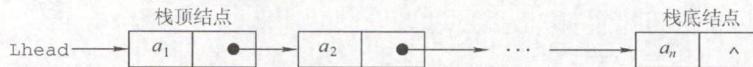


图 3.4 栈的链式存储

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
} *ListStack;               // 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，入栈和出栈的操作都在链表的表头进行。需要注意的是，对于带头结点和不带头结点的链栈，具体的实现会有所不同。

3.1.4 本节试题精选

一、单项选择题

01. 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

02. 栈是（）。

- A. 顺序存储的线性结构 B. 链式存储的非线性结构

关注公众号【乘龙考研】
一手更新 稳定有保障

- C. 限制存取点的线性结构 D. 限制存储点的非线性结构
03. () 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
04. 假定利用数组 $a[n]$ 顺序存储一个栈, 用 top 表示栈顶指针, 用 $\text{top}=-1$ 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. $a[--\text{top}] = x$ B. $a[\text{top}--] = x$ C. $a[+\text{top}] = x$ D. $a[\text{top}+] = x$
05. 设有一个空栈, 栈顶指针为 $1000H$, 每个元素需要一个存储单元, 执行 Push、Push、Pop、Push、Pop、Push、Pop 操作后, 栈顶指针的值为 ()。
 A. $1002H$ B. $1003H$ C. $1004H$ D. $1005H$
06. 和顺序栈相比, 链栈有一个比较明显的优势, 即 ()。
 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
07. 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
08. 向一个栈顶指针为 top 的链栈(不带头结点)中插入一个 x 结点, 则执行 ()。
 A. $\text{top} \rightarrow \text{next} = x$ B. $x \rightarrow \text{next} = \text{top} \rightarrow \text{next}; \text{top} \rightarrow \text{next} = x$
 C. $x \rightarrow \text{next} = \text{top}; \text{top} = x$ D. $x \rightarrow \text{next} = \text{top}; \text{top} = \text{top} \rightarrow \text{next}$
09. 链栈(不带头结点)执行 Pop 操作, 并将出栈的元素存在 x 中, 应该执行 ()。
 A. $x = \text{top}; \text{top} = \text{top} \rightarrow \text{next}$ B. $x = \text{top} \rightarrow \text{data}$
 C. $\text{top} = \text{top} \rightarrow \text{next}; x = \text{top} \rightarrow \text{data}$ D. $x = \text{top} \rightarrow \text{data}; \text{top} = \text{top} \rightarrow \text{next}$
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); GetTop(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, 则下面得不到的出栈序列为 ( )。  
 A. fedcba      B. bcafed      C. dcefba      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. SXSXSSXX      B. SSSXXXSXX      C. SXSSXXSX      D. SXSSXSXX
14. 若一个栈的输入序列是  $1, 2, 3, \dots, n$ , 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.  $n-i+1$
15. 一个栈的输入序列为  $1, 2, 3, \dots, n$ , 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
16. 某栈的输入序列为  $a, b, c, d$ , 下面的 4 个序列中, 不可能为其输出序列的是 ( )。

- A.  $a, b, c, d$       B.  $c, b, d, a$       C.  $d, c, a, b$       D.  $a, c, b, d$
17. 若一个栈的输入序列是  $P_1, P_2, \dots, P_n$ , 输出序列是  $1, 2, 3, \dots, n$ , 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
18. 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.  $3, 4$
19. 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
20. 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
21. 设有一个顺序共享栈 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
22. 【2009 统考真题】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
23. 【2010 统考真题】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, 不可能得到的出栈序列是( )。  
 A.  $dceefa$       B.  $cbdaef$       C.  $bcaefd$       D.  $afedcb$
24. 【2011 统考真题】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
25. 【2013 统考真题】一个栈的入栈序列为  $1, 2, 3, \dots, n$ , 出栈序列为  $P_1, P_2, P_3, \dots, P_n$ 。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
26. 【2017 统考真题】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
27. 【2018 统考真题】若栈 S1 中保存整数, 栈 S2 中保存运算符, 函数 F() 依次执行下述各步操作:  
 1) 从 S1 中依次弹出两个操作数 a 和 b。

关注公众号【乘龙考研】  
一手更新 稳定有保障

2) 从 S2 中弹出一个运算符 op。

3) 执行相应的运算 b op a。

4) 将运算结果压入 S1 中。

假定 S1 中的操作数依次是 5, 8, 3, 2 (2 在栈顶), S2 中的运算符依次是 \*、-、+ (+ 在栈顶)。调用 3 次 F() 后, S1 栈顶保存的值是 ( )。

- A. -15      B. 15      C. -20      D. 20

28. 【2020 统考真题】对空栈 S 进行 Push 和 Pop 操作, 入栈序列为 a, b, c, d, e, 经过 Push、Push、Pop、Push、Pop、Push、Push、Pop 操作后得到的出栈序列是 ( )。

- A. b, a, c      B. b, a, e      C. b, c, a      D. b, c, e

29. 【2022 统考真题】给定有限符号集 S, in 和 out 均为 S 中所有元素的任意排列。对于初始为空的栈 ST, 下列叙述中, 正确的是 ( )。

- A. 若 in 是 ST 的入栈序列, 则不能判断 out 是否为其可能的出栈序列  
 B. 若 out 是 ST 的出栈序列, 则不能判断 in 是否为其可能的入栈序列  
 C. 若 in 是 ST 的入栈序列, out 是对应 in 的出栈序列, 则 in 与 out 一定不同  
 D. 若 in 是 ST 的入栈序列, out 是对应 in 的出栈序列, 则 in 与 out 可能互为倒序

## 二、综合应用题

01. 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 的出栈序列有哪几个?

02. 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?

03. 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. IOIOIOOO      B. IOOIOIOO      C. IIIOIOIO      D. IIIOOIOO

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)。

04. 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。

05. 设有两个栈 s1、s2 都采用顺序栈方式, 并共享一个存储区 [0, ..., 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 s1、s2 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

01. B

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。

02. C

首先栈是一种线性表, 所以 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 A 错。栈和队列都是限制存取点的线性结构。

关注公众号【乘龙考研】  
一手更新 稳定有保障

**03. B**

基本操作是指该结构最核心、最基本的操作，其他较复杂的操作可通过基本操作实现。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。

**04. C**

初始时  $\text{top}$  为 -1，则第一个元素入栈后， $\text{top}$  为 0，即指向栈顶元素，故入栈时应先将指针  $\text{top}$  加 1，再将元素入栈，只有选项 C 符合题意。

**05. A**

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

**06. 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 A。

**07. C**

对于双向循环链表，不管是表头指针还是表尾指针，都可以很方便地找到表头结点，方便在表头做插入或删除操作。而单循环链表通过尾指针可以很方便地找到表头结点，但通过头指针找尾结点则需要遍历一次链表。对于 C，插入和删除结点后，找尾结点需要花费  $O(n)$  的时间。

**08. 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  $x \rightarrow \text{next} = \text{top}$ ），插入完后需将  $\text{top}$  指向该插入的结点  $x$ 。请思考当链栈存在头结点时的情况。

**09. 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  $\text{top}$  指针赋给了  $x$ ，错误；B 中没有修改  $\text{top}$  指针的值；C 为  $\text{top}$  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，栈顶元素  $b$  出栈， $x$  的值为  $b$ ；执行最后一句，读取栈顶元素的值， $x$  的值为  $a$ 。

**11. B**

对于  $n$  个不同元素进栈，出栈序列的个数为

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

关注公众号【乘龙考研】  
一手更新 稳定有保障

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，因此可采用例举方法。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，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。

此题也可采用将各序列逐个代入的方法来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

第 $n$ 个元素第一个出栈，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。

#### 15. 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。

#### 16. 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。

#### 17. 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 $P_1$ 不是栈顶元素，因此 $P_1$ 的值不可能是2。思考：哪些 $P_i$ 可能是2？

#### 18. C

逐个判断每个选项可能的入栈出栈顺序。对于A，可能的顺序是1入，1出，2入，2出，3入，3出，4入，4出。对于B，可能的顺序是1入，2入，3入，3出，2出，4入，4出，1出。对于D，可能的顺序是1入，1出，2入，3入，3出，2出，4入，4出。C没有对应的序列，因为当4在栈中时，意味着前面的所有元素（1、2、3）都已在栈中或曾经入过栈，此时若4第二个出栈，即栈中还有两个元素，且这两个元素是有序的（对应入栈顺序），只能为(1, 2)、(1, 3)、(2, 3)，若是(1, 2)这个序列，则3已在 $p_1$ 位置出栈，不可能再在 $p_4$ 位置出栈，若是(1, 3)和(2, 3)这种情况中的任意一种，则3一定是下一个出栈元素，即 $p_3$ 一定是3，所以 $p_4$ 不可能是3。

**【另解】**对于C， $p_2$ 为最后一个入栈元素4，则只有 $p_1$ 或 $p_3$ 出栈的元素有可能为3（请读者分两种情况自行思考），而 $p_4$ 绝不可能为3。读者在解答此类题时，一定要注意出栈序列中的“最后一个入栈元素”，这样可以节省答题的时间。

#### 19. C

标识符只能以英文字母或下画线开头，而不能是数字开头。故由n、1、\_三个字符组合成的标识符有n1\_，n\_1，\_1n和\_n1四种。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。

#### 20. 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。

#### 21. 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，做题时务必仔细审题和思考。

### 22. 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 | aef | 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】元素的出队顺序和入队顺序相同，因此元素的出栈顺序就是 b, d, c, f, e, a, g，因此元素的入栈出栈次序为 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 操作进行加 1 操作，每做一次 Pop 操作进行减 1 操作，记录容量的最大值为 3，选 C。

### 23. D

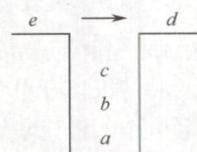
选项 A 可由 a 进, b 进, c 进, d 进, d 出, c 出, e 进, e 出, b 出, f 进, f 出, a 出得到；选项 B 可由 a 进, b 进, c 进, c 出, b 出, d 进, d 出, a 出, e 进, e 出, f 进, f 出得到；选项 C 可由 a 进, b 进, b 出, c 进, c 出, a 出, d 进, e 进, e 出, f 进, f 出, d 出得到；选项 D 可由 a 进, a 出, b 进, c 进, d 进, e 进, f 进, f 出, e 出, d 出, c 出, b 出得到，但要求不允许连续 3 次退栈操作，故选 D。

### 24. B

d 第一个出栈，则 c, b, a 出栈的相对顺序是确定的，出栈顺序必为 d\_c\_b\_a\_，e 的顺序不定，在任意一个 “\_” 上都可能。

【另解】d 首先出栈，则 abc 停留在栈中，此时栈的状态如右图所示。

此时可以有如下 4 种操作：①e 进栈后出栈，则出栈序列为 decba；②c 出栈，e 进栈后出栈，出栈序列为 dceba；③cb 出栈，e 进栈后出栈，出栈序列为 dcbea；④cba 出栈，e 进栈后出栈，出栈序列为 dcbae。思路和上面其实一样。



### 25. 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 是否可取： $P_1$  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。故  $P_3$  可能取除 3 之外的所有数，个数为  $n-1$ 。

### 26. 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 Push, Push, Pop, Pop 操作，出栈次序为 2、1；进行 Push, Pop, Push, Pop 操作，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。

### 27. B

第一次调用：①从 S1 中弹出 2 和 3；②从 S2 中弹出+；③执行  $3+2=5$ ；④将 5 压入 S1 中，第一次调用结束后 S1 中剩余 5、8、5（5 在栈顶），S2 中剩余\*、-（-在栈顶）。第二次调用：①从 S1 中弹出 5 和 8；②从 S2 中弹出-；③执行  $8-5=3$ ；④将 3 压入 S1 中，第二次调用结束后 S1 中剩余 5、3（3 在栈顶），S2 中剩余\*。第三次调用：①从 S1 中弹出 3 和 5；②从 S2 中弹出\*；③执行  $5*3=15$ ；④将 15 压入 S1 中，第三次调用结束后 S1 中仅剩余 15，S2 为空。故选 B。

### 28. D

按题意，出入栈操作的过程如下：

| 操作   | 栈内元素  | 出栈元素 |
|------|-------|------|
| Push | a     |      |
| Push | a b   |      |
| Pop  | a     | b    |
| Push | a c   |      |
| Pop  | a     | c    |
| Push | a d   |      |
| Push | a d e |      |
| Pop  | a d   | e    |

故出栈序列为 b, c, e。

### 29. D

通过模拟出入栈操作，可以判断入栈序列 in 和出栈序列 out 是否合法。因此，已知 in 序列可以判断 out 序列是否为可能的出栈序列；已知 out 序列也可以判断 in 序列是否为可能的入栈序列，A 和 B 错误。如果每个元素入栈后立即出栈，则 in 序列和 out 序列相同，C 错误。如果所有元素都入栈后才依次出栈，则 in 序列和 out 序列互为倒序，D 正确。

## 二、综合应用题

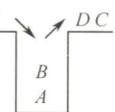
### 01. 【解答】

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，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。



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



### 02. 【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。

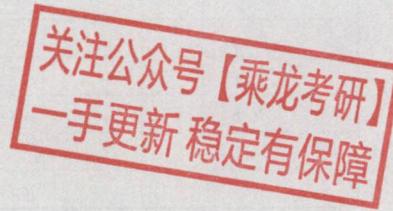
### 03. 【解答】

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：依次逐一扫描入栈出栈序列（即由“I”和“O”组成的字符串），每扫描至任意一个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。

```
bool Judge(char A[]) {
 int i=0;
 int j=k=0; //i 为下标，j 和 k 分别为字母 I 和 O 的个数
 while(A[i]!='\0'){ //未到字符数组尾
 switch(A[i]){
 case 'I': j++; break; //入栈次数增 1
 case 'O': k++; //不论 A[i] 是 “I” 或 “O”，指针 i 均后移
 if(k>j){printf("序列非法\n"); exit(0); }
 }
 i++; //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，因此可将判定一组出入栈序列是否合法转化为一组由 +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。

#### 04. 【解答】

算法思想：使用栈来判断链表中的数据是否中心对称。让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，则将链表中的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){
 int i;
 char s[n/2]; //s 字符栈
 LNode *p=L->next; //工作指针 p，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，然后扫描单链表 L 并比较，直到比较到单链表 L 尾为止，但算法需要两次扫描单链表 L，效率不及上述算法高。

### 05. 【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 栈是通常意义上的栈；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);
 }
}

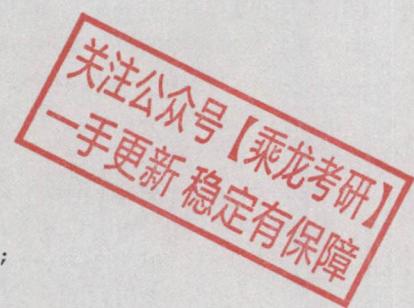
```



```

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

```



## 3.2 队列

### 3.2.1 队列的基本概念

#### 1. 队列的定义

队列 (Queue) 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)，如图 3.5 所示。

队头 (Front)。允许删除的一端，又称队首。

队尾 (Rear)。允许插入的一端。

空队列。不含任何元素的空表。

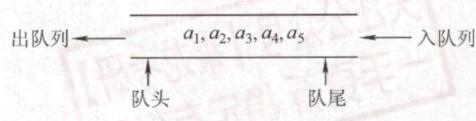


图 3.5 队列示意图

#### 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，栈和队列是操作受限的线性表，因此不是任何对线性表的操作都可以作为栈和队列的操作。比如，不可以随便读取栈或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：队头指针 `front` 指向队头元素，队尾指针 `rear` 指向队尾元素的下一个位置(不同教材对 `front` 和 `rear` 的定义可能不同，例如，可以让 `rear` 指向队尾元素、`front` 指向队头元素。对于不同的定义，出队入队的操作是不同的，本节后面有一些相关的习题，读者可以结合习题思考)。

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{
 ElemtType data[MaxSize]; // 存放队列元素
 int front,rear; // 队头指针和队尾指针
} SqQueue;
```

初始时：`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。

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==MaxSize` 作为队列满的条件呢？显然不能，图 3.6(d)中，队列中仅有一个元素，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”。

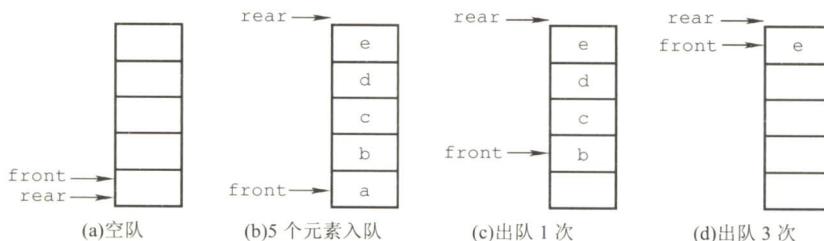


图 3.6 队列的操作

#### 2. 循环队列

前面指出了顺序队列的缺点，这里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视为一个环，称为循环队列。当队首指针 `Q.front=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)%MaxSize`。

队尾指针进 1：`Q.rear=(Q.rear+1)%MaxSize`。

队列长度：`(Q.rear+MaxSize-Q.front)%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`。若入队元素的速度快于出队元素的速度，则队尾指针很快就会赶上队首指针，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。

为了区分是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位置作为队满的标志”，如图 3.7(d2)所示。

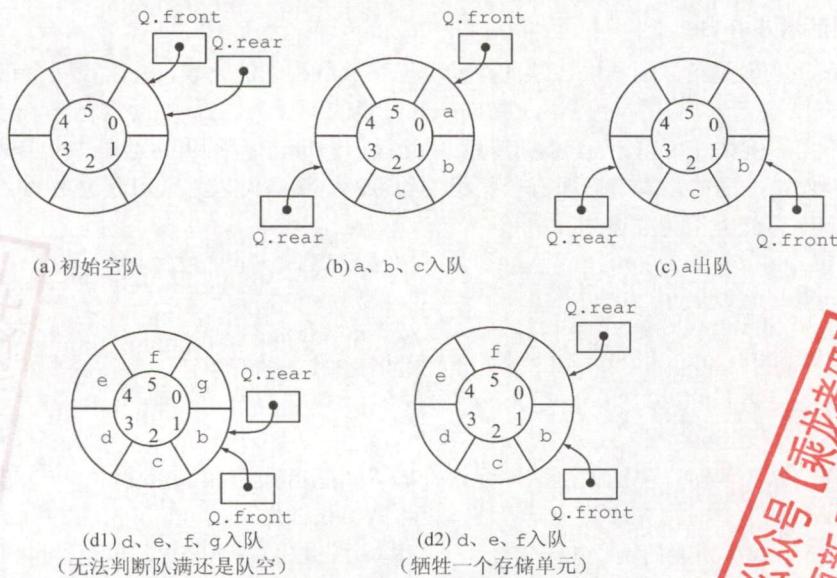


图 3.7 循环队列出入队示意图

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

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

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

- 2) 类型中增设表示元素个数的数据成员。这样，队空的条件为  $Q.\text{size}==0$ ；队满的条件为  $Q.\text{size}==\text{MaxSize}$ 。这两种情况都有  $Q.\text{front}==Q.\text{rear}$ 。
- 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 时，若因删除导致  $Q.\text{front}==Q.\text{rear}$ ，则为队空；tag 等于 1 时，若因插入导致  $Q.\text{front}==Q.\text{rear}$ ，则为队满。

### 3. 循环队列的操作

#### (1) 初始化

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

#### (2) 判队空

```
bool isEmpty(SqQueue 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; // 队满则报错
 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, ElecType &x) {
 if(Q.rear==Q.front) return false; //队空则报错
 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

关注公众号【乘龙考研】  
一手更新 稳定有保障

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。

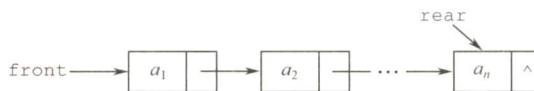


图 3.8 不带头结点的链式队列

队列的链式存储类型可描述为

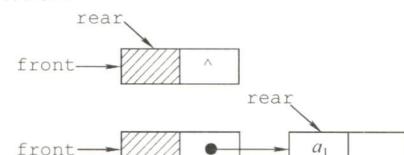
```
typedef struct LinkNode{ //链式队列结点
 ElecType data;
 struct LinkNode *next;
}LinkNode;
typedef struct{ //链式队列
 LinkNode *front, *rear; //队列的队头和队尾指针
}*LinkQueue;
```

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，不带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。



## 2. 链式队列的基本操作

图 3.9 带头结点的链式队列

## (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

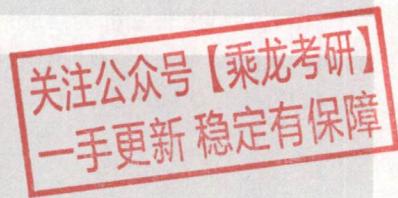
```
bool IsEmpty(LinkQueue Q) {
 if(Q.front==Q.rear) return true;
 else return false;
}
```

## (3) 入队

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

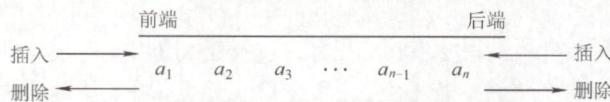


图 3.10 双端队列

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，无论是前端还是后端出队，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。

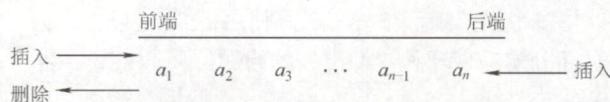


图 3.11 输出受限的双端队列

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。若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。

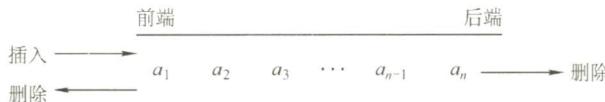


图 3.12 输入受限的双端队列

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，又不能由输出受限的双端队列得到的输出序列。

解：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，则 end2 端的输出相当于队列的输出，即 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 Catalan 公式得出），仅通过 end1 端不能得到的输出序列有  $4! - 14 = 10$  种：

|            |            |            |            |            |
|------------|------------|------------|------------|------------|
| 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， $S_L, X_L$  分别代表 end1 端的进队和出队， $X_R$  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 | $S_L X_R S_L S_L X_L X_R X_R$     | 3, 1, 2, 4 | $S_L S_L S_L X_L S_L X_R X_R X_R$ |
| 2, 4, 1, 3 | $S_L S_L X_L S_L S_L X_L X_R X_R$ | 4, 1, 2, 3 | $S_L S_L S_L S_L X_L X_R X_R X_R$ |
| 3, 4, 1, 2 | $S_L S_L S_L X_L S_L X_L X_R X_R$ | 4, 1, 3, 2 | $S_L S_L S_L S_L X_L X_R X_L X_R$ |
| 3, 1, 4, 2 | $S_L S_L S_L X_L X_R S_L X_L X_R$ | 4, 3, 1, 2 | $S_L S_L S_L S_L X_L X_R X_L X_R$ |

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。若都从 end2 端输入，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，交替从 end1 和 end2 端输入，还可以输出其中 8 种。设  $S_L$  代表 end1 端的输入， $S_R, X_R$  分别代表 end2 端的输入和输出，则可能的输出序列见下表。



图 3.13 输入受限的双端队列



图 3.14 输出受限的双端队列

| 输出序列       | 进队出队顺序                                | 输出序列       | 进队出队顺序                            |
|------------|---------------------------------------|------------|-----------------------------------|
| 1, 4, 2, 3 | $S_L X_R S_L S_R X_R X_R X_R$         | 3, 1, 2, 4 | $S_L S_L S_R X_R X_R S_L X_R X_R$ |
| 2, 4, 1, 3 | $S_L S_L X_R S_L S_R X_R X_R X_R$     | 4, 1, 2, 3 | $S_L S_L S_L S_R X_R X_R X_R X_R$ |
| 3, 4, 1, 2 | $S_L S_L S_L X_R S_R X_R X_R X_R$     | 4, 2, 1, 3 | $S_L S_R S_L S_R X_R X_R X_R X_R$ |
| 3, 1, 4, 2 | $S_L S_L S_L X_R X_R S_R X_R X_R X_R$ | 4, 3, 1, 2 | $S_L S_L S_R S_R X_R X_R X_R X_R$ |

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。



提示：实际双端队列的考题不会这么复杂，通常仅判断序列是否满足题设条件，代入验证即可。

### 3.2.5 本节试题精选

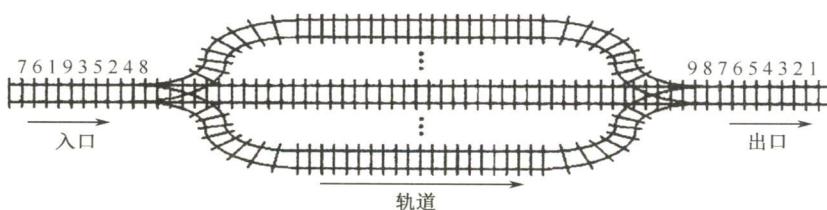
#### 一、单项选择题

01. 栈和队列的主要区别在于（ ）。
- A. 它们的逻辑结构不一样
  - B. 它们的存储结构不一样
  - C. 所包含的元素不一样
  - D. 插入、删除操作的限定不一样
02. 队列的“先进先出”特性是指（ ）。
- I. 最后插入队列中的元素总是最后被删除
  - II. 当同时进行插入、删除操作时，总是插入操作优先
  - III. 每当有删除操作时，总要先做一次插入操作
  - IV. 每次从队列中删除的总是最早插入的元素
- A. I
  - B. I 和 IV
  - C. II 和 III
  - D. IV
03. 允许对队列进行的操作有（ ）。
- A. 对队列中的元素排序
  - B. 取出最近进队的元素
  - C. 在队列元素之间插入元素
  - D. 删除队头元素
04. 一个队列的入队顺序是 1, 2, 3, 4，则出队的输出顺序是（ ）。
- A. 4, 3, 2, 1
  - B. 1, 2, 3, 4
  - C. 1, 4, 3, 2
  - D. 3, 2, 4, 1
05. 循环队列存储在数组 A[0...n] 中，入队时的操作为（ ）。
- A. rear=rear+1
  - B. rear=(rear+1) mod (n-1)
  - C. rear=(rear+1) mod n
  - D. rear=(rear+1) mod (n+1)
06. 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（ ）。
- A. 5
  - B. 6
  - C. 16
  - D. 17
07. 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（ ）。
- A. 3 和 4
  - B. 3 和 0
  - C. 5 和 0
  - D. 5 和 1
08. 假设一个循环队列 Q[MaxSize] 的队头指针为 front，队尾指针为 rear，队列的最大容量为 MaxSize，此外，该队列再没有其他数据成员，则判断该队的列满条件是（ ）。
- A. Q.front==Q.rear
  - B. Q.front+Q.rear>=MaxSize
  - C. Q.front==(Q.rear+1)%MaxSize
  - D. Q.rear==(Q.front+1)%MaxSize
09. 最适合用作链队的链表是（ ）。
- 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
10. 最不适合用作链式队列的链表是（ ）。
- 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

关注公众号【乘龙考研】  
一手更新 稳定有保障

11. 在用单链表实现队列时，队头设在链表的（ ）位置。  
 A. 链头      B. 链尾      C. 链中      D. 以上都可以
12. 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
13. 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. `front=x, front=front->next`  
 B. `x->next=front->next, front=x`  
 C. `rear->next=x, rear=x`  
 D. `rear->next=x, x->next=NULL, rear=x`
14. 假设循环单链表表示的队列长度为  $n$ ，队头固定在链表尾，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.  $O(n \log_2 n)$
15. 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，又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
16. 【2010 统考真题】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.  $b, a, c, d, e$       B.  $d, b, a, c, e$       C.  $d, b, c, a, e$       D.  $e, c, b, a, d$
17. 【2011 统考真题】已知循环队列存储在一维数组  $A[0...n-1]$  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（ ）。  
 A. 0, 0      B. 0,  $n-1$       C.  $n-1$ , 0      D.  $n-1$ ,  $n-1$
18. 【2014 统考真题】循环队列放在一维数组  $A[0...M-1]$  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（ ）。  
 A. 队空：`end1==end2;`      队满：`end1==(end2+1) mod M`  
 B. 队空：`end1==end2;`      队满：`end2==(end1+1) mod (M-1)`  
 C. 队空：`end2==(end1+1) mod M;`      队满：`end1==(end2+1) mod M`  
 D. 队空：`end1==(end2+1) mod M;`      队满：`end2==(end1+1) mod (M-1)`
19. 【2016 统考真题】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（ ）。

关注公众号【乘龙考研】  
一手更新 稳定有保障



- A. 2                    B. 3                    C. 4                    D. 5
20. 【2018 统考真题】现有队列 Q 与栈 S，初始时 Q 中的元素依次是 1, 2, 3, 4, 5, 6 (1 在队头)，S 为空。若仅允许下列 3 种操作：①出队并输出出队元素；②出队并将出队元素入栈；③出栈并输出出栈元素，则不能得到的输出序列是（ ）。
- A. 1, 2, 5, 6, 4, 3      B. 2, 3, 4, 5, 6, 1      C. 3, 4, 5, 6, 1, 2      D. 6, 5, 4, 3, 2, 1
21. 【2021 统考真题】初始为空的队列 Q 的一端仅能进行入队操作，另外一端既能进行入队操作又能进行出队操作。若 Q 的入队序列是 1, 2, 3, 4, 5，则不能得到的出队序列是（ ）。
- A. 5, 4, 3, 1, 2      B. 5, 3, 1, 2, 4      C. 4, 2, 1, 3, 5      D. 4, 1, 3, 2, 5

## 二、综合应用题

01. 若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。
02. 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。
03. 利用两个栈 S1 和 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         |

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         |

04. 【2019 统考真题】请设计一个队列，要求满足：①初始时队列为空；②入队时，允许增加队列占用空间；③出队后，出队元素所占用的空间可重复使用，即整个队列所占用的空间只增不减；④入队操作和出队操作的时间复杂度始终保持为  $O(1)$ 。请回答下列问题：
- 1) 该队列是应选择链式存储结构，还是应选择顺序存储结构？
  - 2) 画出队列的初始状态，并给出判断队空和队满的条件。
  - 3) 画出第一个元素入队后的队列状态。
  - 4) 给出入队操作和出队操作的基本过程。

### 3.2.6 答案与解析

#### 一、单项选择题

01. D

栈和队列的逻辑结构都是线性结构，都可以采用顺序存储或链式存储，C 显然也错误。只有 D 才是栈和队列的本质区别，限定表中插入和删除操作位置的不同。

02. B

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

03. D

删除队头元素即出队，是队列的基本操作之一，故选 D。

04. B

关注公众号【乘龙考研】  
一手更新 稳定有保障

队列的入队顺序和出队顺序是一致的，这是和栈不同的。

**05. D**

数组下标范围  $0 \sim n$ ，因此数组容量为  $n+1$ 。循环队列中元素入队的操作是  $\text{rear} = (\text{rear}+1) \bmod \text{maxsize}$ ，题中  $\text{maxsize}=n+1$ 。因此入队操作应为  $\text{rear} = (\text{rear}+1) \bmod (n+1)$ 。

**06. 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  $\text{front}$  指向当前元素， $\text{rear}$  指向队尾元素的下一个元素的计算是相同的。

**注意：**数组  $A[n]$  的下标范围为  $0 \sim n-1$ 。如果写成  $A[0 \dots n]$ ，则说明下标范围为  $0 \sim n$ 。

**07. 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， $\text{front}=0$ ,  $\text{rear}=3$ 。

**08. 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是队空还是队满，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。

**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。

**09. 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（画蛇添足）。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。

**10. 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。

**11. 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。

**12. D**

队列用链式存储时，删除元素从表头删除，通常仅需修改头指针，但若队列中仅有一个元素，则尾指针也需要被修改，当仅有一个元素时，删除后队列为空，需修改尾指针为  $\text{rear}=\text{front}$ 。

**13. D**

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

**14. A**

依题意，进队操作是在队尾进行，即链表表头。题中已明确说明链表只设头指针，也即没有头结点和尾指针，进队后，循环单链表必须保持循环的性质，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  $O(n)$ 。

**15. 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 4, 3, 2, 1，排除 A；左出、右出、左出、左出可得到 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。

### 16. 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。

【另解】初始时队列为空，第 1 个元素 a 左入（或右入）后，第 2 个元素 b 无论是左入还是右入都必与 a 相邻，而选项 C 中 a 与 b 不相邻，不合题意。

### 17. 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  $(rear+1) \% n$  操作，所以若入队后指针指向 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。

### 18. A

end1 指向队头元素，可知出队操作是先从 A[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[end2]，然后 end2 再加 1。若用 A[0] 存储第一个元素，队列初始时，入队操作是先把数据放到 A[0] 中，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 的初值也为 0，可知队空条件为 end1==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 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 end1=0，end2 指向队尾元素的后一个位置，可知 end2=M-2+1=M-1，所以队满的条件为 end1== (end2+1) mod M。

### 19. C

根据题意：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个队列 ( $n$  条轨道)，且每个队列（轨道）可容纳多个元素（多列列车），为便于区分，队列用字母编号。分析如下：显然先入队的元素必须小于后入队的元素（否则，若 8 和 4 入同一队列，8 在 4 前面，则出队时也只能 8 在 4 前面），这样 8 入队列 A，4 入队列 B，2 入队列 C，5 入队列 B（按照前述原则“大的元素在小的元素后面”也可将 5 入队列 C，但这时剩下的元素 3 就必须放入一个新的队列中，无法确保“至少”），3 入队列 C，9 入队列 A，这时共占了 3 个队列，后面还有元素 1，直接再用一个新的队列 D，1 从队列 D 出队后，剩下的元素 6 和 7 或入队列 B，或入队列 C。综上，共占用了 4 个队列。当然还有其他的入队、出队情况，请读者自己推演，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（即满足题意中“至少”）的队列。

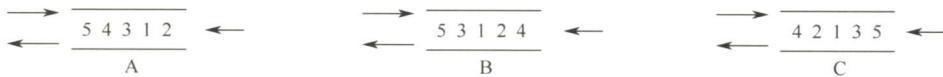
### 20. C

A 的操作顺序为 ①①②②①①③③。B 的操作顺序为 ②①①①①①③。D 的操作顺序为 ②②②②②①③③③③③③。对于 C：首先输出 3，说明 1 和 2 必须先依次入栈，而此后 2 肯定比 1 先输出，因此无法得到 1, 2 的输出顺序。

### 21. D

假设队列左端允许入队和出队，右端只能入队。对于 A，依次从右端入队 1, 2，再从左端入队 3, 4, 5。对于 B，从右端入队 1, 2，然后从左端入队 3，再从右端入队 4，最后从左端入队

5. 对于 C, 从左端入队 1, 2, 然后从右端入队 3, 再从左端入队 4, 最后从右端入队 5。无法验证 D 的序列。



## 二、综合应用题

### 01. 【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 tag=0、front=rear=0。这样队列的 4 要素如下:

队空条件: Q.front==Q.rear 且 Q.tag==0。

队满条件: Q.front==Q.rear 且 Q.tag==1。

进队操作: Q.data[Q.rear]=x; Q.rear=(Q.rear+1)%MaxSize; Q.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, ElemtType 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;
}
```

关注公众号【乘龙考研】  
一手更新 稳定有保障

2) 设“tag”法的循环队列出队算法:

```
int DeQueue1(SqQueue &Q, ElemtType &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;
}
```

### 02. 【解答】

本题主要考查大家对队列和栈的特性与操作的理解。由于对队列的一系列操作不可能将其中的元素逆置, 而栈可以将入栈的元素逆序提取出来, 因此我们可以让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
 }
}

```

### 03. 【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 的出栈操作用做出队，若 S2 为空，则先将 S1 中的所有元素送入 S2。
- 2) 对 S1 的入栈操作用作入队，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, ElemtType 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;
}

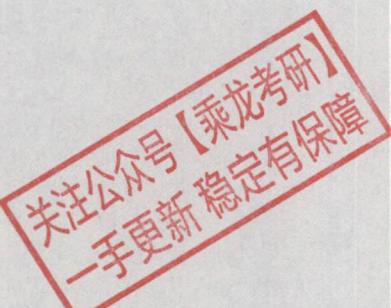
```

出队算法：

```

void DeQueue(Stack &S1, Stack &S2, ElemtType &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：

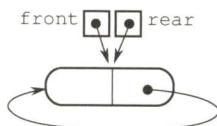
```

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

```

#### 04. 【解答】

- 1) 顺序存储无法满足要求②的队列占用空间随着入队操作而增加。根据要求来分析：要求①容易满足；链式存储方便开辟新空间，要求②容易满足；对于要求③，出队后的结点并不真正释放，用队头指针指向新的队头结点，新元素入队时，有空余结点则无须开辟新空间，赋值到队尾后的第一个空结点即可，然后用队尾指针指向新的队尾结点，这就需要设计成一个首尾相接的循环单链表，类似于循环队列的思想。设置队头、队尾指针后，链式队列的入队操作和出队操作的时间复杂度均为  $O(1)$ ，要求④可以满足。  
因此，采用链式存储结构（两段式单向循环链表），队头指针为 `front`，队尾指针为 `rear`。
- 2) 该循环链式队列的实现可以参考循环队列，不同之处在于循环链式队列可以方便地增加空间，出队的结点可以循环利用，入队时空间不够也可以动态增加。同样，循环链式队列也要区分队满和队空的情况，这里参考循环队列牺牲一个单元来判断。初始时，创建只有一个空闲结点的循环单链表，头指针 `front` 和尾指针 `rear` 均指向空闲结点，如下图所示。

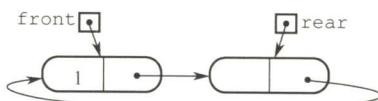


关注公众号【乘龙考研】  
一手更新 稳定有保障

队空的判定条件：`front==rear`。

队满的判定条件：`front==rear->next`。

- 3) 插入第一个元素后的状态如下图所示。



- 4) 操作的基本过程如下：

| 入队操作：                                                                                                                                                       |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 若 ( <code>front==rear-&gt;next</code> ) //队满<br>则在 <code>rear</code> 后面插入一个新的空闲结点；<br>入队元素保存到 <code>rear</code> 所指结点中； <code>rear=rear-&gt;next</code> ；返回。 |
| 出队操作：                                                                                                                                                       |
| 若 ( <code>front==rear</code> ) //队空<br>则出队失败，返回；<br>取 <code>front</code> 所指结点中的元素 <code>e</code> ； <code>front=front-&gt;next</code> ；返回 <code>e</code> 。   |

### 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 “)” 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 “]” 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。
- 4) 以此类推，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，则或使置于栈顶的最急迫期待得以消解，或是不合法的情况（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。

关注公众号【乘龙考研】  
一手更新 稳定有保障

#### 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  $A+B*(C-D)-E/F$  所对应的后缀表达式为  $ABCD-*+EF/-$ 。

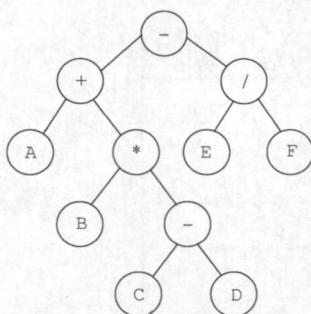


图 3.15  $A+B*(C-D)-E/F$  对应的表达式

中缀表达式转化为后缀表达式的过程，见 3.3.6 节中习题 11 的解析，这里不再赘述。

读者也可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，见图 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：若该项是操作数，则将其压入栈中；若该项是操作符  $<\text{op}>$ ，则连续从栈中退出两个操作数  $Y$  和  $X$ ，形成运算指令  $X<\text{op}>Y$ ，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-\*+EF/-求值的过程需要 12 步，见表 3.1。

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                                                                                        | A B                           |
| 4  | C   | 操作数 | 进栈                                                                                        | A B C                         |
| 5  | D   | 操作数 | 进栈                                                                                        | A B C D                       |
| 6  | -   | 操作符 | D、C 退栈，计算 C-D，结果 R <sub>1</sub> 进栈                                                        | A B R <sub>1</sub>            |
| 7  | *   | 操作符 | R <sub>1</sub> 、B 退栈，计算 B×R <sub>1</sub> ，结果 R <sub>2</sub> 进栈                            | A R <sub>2</sub>              |
| 8  | +   | 操作符 | R <sub>2</sub> 、A 退栈，计算 A+R <sub>2</sub> ，结果 R <sub>3</sub> 进栈                            | R <sub>3</sub>                |
| 9  | E   | 操作数 | 进栈                                                                                        | R <sub>3</sub> E              |
| 10 | F   | 操作数 | 进栈                                                                                        | R <sub>3</sub> E F            |
| 11 | /   | 操作符 | F、E 退栈，计算 E/F，结果 R <sub>4</sub> 进栈                                                        | R <sub>3</sub> R <sub>4</sub> |
| 12 | -   | 操作符 | R <sub>4</sub> 、R <sub>3</sub> 退栈，计算 R <sub>3</sub> -R <sub>4</sub> ，结果 R <sub>5</sub> 进栈 | R <sub>5</sub>                |

### 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，若在一个函数、过程或数据结构的定义中又应用了它自身，则这个函数、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

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

关注公众号【乘龙考研】  
一手更新 稳定有保障

这就是递归的一个典型例子，用程序实现时如下：

```
int Fib(int 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。

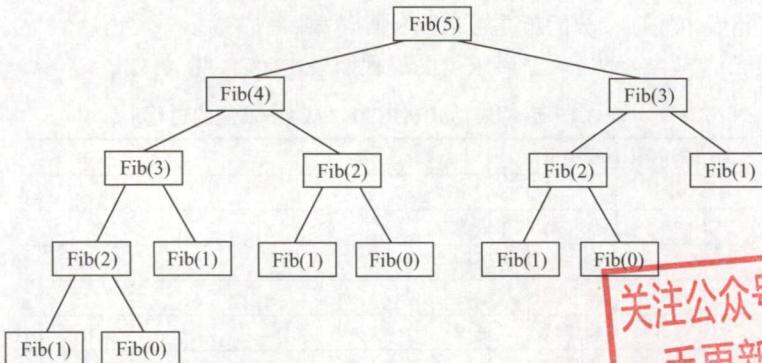


图 3.16 Fib(5) 的递归执行过程

关注公众号【乘龙考研】  
一手更新 稳定有保障

显然，在递归调用的过程中， $\text{Fib}(3)$  被计算了 2 次， $\text{Fib}(2)$  被计算了 3 次。 $\text{Fib}(1)$  被调用了 5 次， $\text{Fib}(0)$  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 5 章的树中利用了递归的思想，代码变得十分简单。通常情况下，初学者很难理解递归的调用过程，若读者想具体了解递归是如何实现的，可以参阅编译原理教材中的相关内容。

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，等到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。

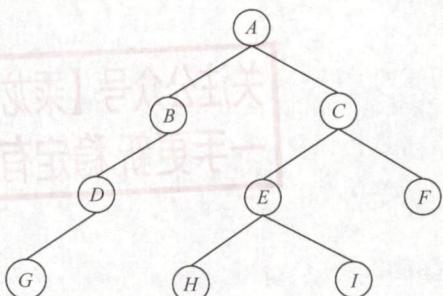


图 3.17 二叉树

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。

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       |        | $ABCDEFGH$ |

### 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做简要说明。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这个缓冲区，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，令其出队，再把CPU分配给新的队首请求的用户使用。这样既能满足每个用户的请求，又使CPU能够正常运行。

### 3.3.6 本节试题精选

#### 一、单项选择题

01. 栈的应用不包括( )。
- A. 递归      B. 进制转换      C. 迷宫求解      D. 缓冲区
02. 表达式  $a*(b+c)-d$  的后缀表达式是( )。
- A. abcd\*+-      B. abc+\*d-      C. abc\*+d-      D. -+\*abcd
03. 下面( )用到了队列。
- A. 括号匹配      B. 表达式求值      C. 页面替换算法      D. 递归
04. 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.  $(A-B)*(C-D)$
05. 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
06. 对于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
07. 执行函数时，其局部变量一般采用()进行存储。
- A. 树形结构 B. 静态链表 C. 栈结构 D. 队列结构

关注公众号【乘龙考研】
一手更新 稳定有保障

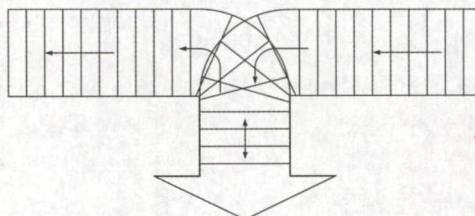
08. 执行()操作时，需要使用队列作为辅助存储空间。
- 查找散列(哈希)表
 - 广度优先搜索图
 - 前序(根)遍历二叉树
 - 深度优先搜索图
09. 下列说法中，正确的是()。
- 消除递归不一定需要使用栈
 - 对同一输入序列进行两组不同的合法入栈和出栈组合操作，所得的输出序列也一定相同
 - 通常使用队列来处理函数或过程调用
 - 队列和栈都是运算受限的线性表，只允许在表的两端进行运算
10. 【2009 统考真题】为解决计算机主机与打印机之间速度不匹配的问题，通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是()。
- 栈
 - 队列
 - 树
 - 图
11. 【2012 统考真题】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 $ab+acd+e/f-* -g +$ 时，用栈来存放暂时还不能确定运算次序的操作符。栈初始时为空时，转换过程中同时保存在栈中的操作符的最大个数是()。
- 5
 - 7
 - 8
 - 11
12. 【2014 统考真题】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()。
- $+(*-$
 - $+(-*$
 - $/+(*-*$
 - $/+-*$
13. 【2015 统考真题】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( )。

- main()→S(1)→S(0)
- S(0)→S(1)→main()
- main()→S(0)→S(1)
- S(1)→S(0)→main()

## 二、综合应用题

01. 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，编写一个算法来判断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。
02. 按下图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(分别用 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。



关注公众号【乘龙考研】  
一手更新 稳定有保障

03. 利用一个栈实现以下递归函数的非递归计算：

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

04. 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；客车先于货车上船，且每上 4 辆客车，才允许放上 1 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

01. D

缓冲区是用队列实现的，A、B、C 都是栈的典型应用。

02. B

后缀表达式中，每个计算符号均直接位于其两个操作数的后面，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，再将操作符提到括号后，最后去掉括号。例如，对于  $(\oplus(\ominus a * (\oplus b + c)) - d)$ ，提出操作符并去掉括号后，可得后缀表达式为  $abc+*d-$ 。

学完第 5 章后，可将表达式画成二叉树的形式，再用后序遍历即可求得后缀表达式。

03. C

页面替换算法中的 FIFO 用到了队列。其余的都只用到了栈。

04. 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，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。

**技巧：**根据算符优先级，统计已依次进栈，但还没有参与计算的算符的个数。以选项 C 为例，'('、'A'、'-'入栈时，'('和'-'还没有参与运算，此时运算符栈大小为 2，'B'和'\*'入栈时运算符大小为 3，'C'入栈时'B\*C'运算，此时运算符栈大小为 2，以此类推。

05. 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，即终止递归的条件。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。

06. 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。

07. 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。

08. B

本题涉及第 5 章和第 6 章的内容，图的广度优先搜索类似于树的层序遍历，都要借助于队列。

关注公众号【乘龙考研】  
一手更新 稳定有保障

**09. 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 正确。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 错误。通常使用栈来处理函数或过程调用，C 错误。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 错误。

**10. B**

在提取数据时必须保持原来数据的顺序，所以缓冲区的特性是先进先出，答案选 B。

**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 $<\text{op}>$ 的优先级来进行栈的变化，我们用  $\text{icp}$  来表示当前扫描到的运算符  $\text{ch}$  的优先级，该运算符进栈后的优先级为  $\text{isp}$ ，则运算符的优先级如下表所示 [ $\text{isp}$  是栈内优先 (in stack priority) 数， $\text{icp}$  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   | #-*()   | /   |
| 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 6 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，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‘(’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。

关注公众号【乘龙考研】  
一手更新 稳定有保障

| 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          |
| $/b+(c*d-e*f)/g$  |        |               | /      | / 入栈               |
| $b+(c*d-e*f)/g$   | /      |               | a      | 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 时，栈中的元素依次是 +(-\*，选 B。

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

步骤 1：按照运算符的优先级对所有的运算单位加括号。

式子变成  $((a/b)+(((c*d)-(e*f))/g))$ 。

步骤 2：转换为前缀或后缀表达式。

前缀：把运算符号移动到对应的括号前面，式子变成  $+(/(ab)/(-(*(cd)*(ef))g))$ 。

把括号去掉：  $+/ab/-*cd*efg$  前缀式子出现。

后缀：把运算符号移动到对应的括号后面，式子变成  $((ab) / (((cd) * (ef) *) - g)) +$ 。

把括号去掉： $ab/cd*ef*-g/+$  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方法快捷得多。

### 13. A

递归调用函数时，在系统栈中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)`。

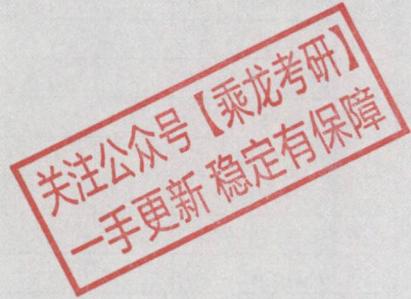
**注意：**在递归调用的过程中，系统为每一层的返回点、局部变量、传入实参等开辟了递归工作栈来进行数据存储。

## 二、综合应用题

### 01. 【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、圆的左括号时进栈，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。最后栈若不为空也为错误。

```
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;
 }
}
```



## 02. 【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; //初始化栈结构
 InitStack(s);
 while(*p) {
 if(*p=='H') //把 H 存入栈中
 Push(s,*p);
 else
 * (q++) =* p; //把 S 调到前部
 p++;
 }
 while(!StackEmpty(s)) {
 Pop(s,c);
 * (q++) =c; //把 H 接在后部
 }
}
```

关注公众号【乘龙考研】  
一手更新 稳定有保障

## 03. 【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;
}
```

## 04. 【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)) //若货车和客车加起来不足 10 辆
 j=11;
 }
}

```

## 3.4 数组和特殊矩阵

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，每个元素在  $n$  个线性关系中的序号称为该元素的下标，下标的取值范围称为数组的维界。

**数组与线性表的关系：**数组是线性表的推广。一维数组可视为一个线性表；二维数组可视为其元素也是定长线性表的线性表，以此类推。数组一旦被定义，其维数和维界就不再改变。因此，除结构的初始化和销毁外，数组只会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  $A[0 \dots n-1]$  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的存储单元。

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  $[0, h_1]$  与  $[0, h_2]$ ，则存储结构关系式为

$$\text{LOC}(a_{i,j}) = \text{LOC}(a_{0,0}) + [i \times (h_2 + 1) + j] \times L$$

例如，对于数组  $A_{[2][3]}$ ，它按行优先方式在内存中的存储形式如图 3.18 所示。

|                                                                                                                           |                                                                                                                                                                                                                                                                                                 |              |              |              |              |              |              |     |  |  |     |  |  |
|---------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|--------------|--------------|--------------|--------------|--------------|-----|--|--|-----|--|--|
| $A_{[2][3]} = \begin{bmatrix} a_{[0][0]} & a_{[0][1]} & a_{[0][2]} \\ a_{[1][0]} & a_{[1][1]} & a_{[1][2]} \end{bmatrix}$ | <table border="1"> <tr><td><math>a_{[0][0]}</math></td><td><math>a_{[0][1]}</math></td><td><math>a_{[0][2]}</math></td><td><math>a_{[1][0]}</math></td><td><math>a_{[1][1]}</math></td><td><math>a_{[1][2]}</math></td></tr> <tr><td colspan="3">第1行</td><td colspan="3">第2行</td></tr> </table> | $a_{[0][0]}$ | $a_{[0][1]}$ | $a_{[0][2]}$ | $a_{[1][0]}$ | $a_{[1][1]}$ | $a_{[1][2]}$ | 第1行 |  |  | 第2行 |  |  |
| $a_{[0][0]}$                                                                                                              | $a_{[0][1]}$                                                                                                                                                                                                                                                                                    | $a_{[0][2]}$ | $a_{[1][0]}$ | $a_{[1][1]}$ | $a_{[1][2]}$ |              |              |     |  |  |     |  |  |
| 第1行                                                                                                                       |                                                                                                                                                                                                                                                                                                 |              | 第2行          |              |              |              |              |     |  |  |     |  |  |

图 3.18 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

$$\text{LOC}(a_{i,j}) = \text{LOC}(a_{0,0}) + [j \times (h_1 + 1) + i] \times L$$

例如，对于数组  $A_{[2][3]}$ ，它按列优先方式在内存中的存储形式如图 3.19 所示。

|                                                                                                                           |                                                                                                                                                                                                                                                                                                                         |              |              |              |              |              |              |     |  |     |  |     |  |
|---------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|--------------|--------------|--------------|--------------|--------------|-----|--|-----|--|-----|--|
| $A_{[2][3]} = \begin{bmatrix} a_{[0][0]} & a_{[0][1]} & a_{[0][2]} \\ a_{[1][0]} & a_{[1][1]} & a_{[1][2]} \end{bmatrix}$ | <table border="1"> <tr><td><math>a_{[0][0]}</math></td><td><math>a_{[1][0]}</math></td><td><math>a_{[0][1]}</math></td><td><math>a_{[1][1]}</math></td><td><math>a_{[0][2]}</math></td><td><math>a_{[1][2]}</math></td></tr> <tr><td colspan="2">第1列</td><td colspan="2">第2列</td><td colspan="2">第3列</td></tr> </table> | $a_{[0][0]}$ | $a_{[1][0]}$ | $a_{[0][1]}$ | $a_{[1][1]}$ | $a_{[0][2]}$ | $a_{[1][2]}$ | 第1列 |  | 第2列 |  | 第3列 |  |
| $a_{[0][0]}$                                                                                                              | $a_{[1][0]}$                                                                                                                                                                                                                                                                                                            | $a_{[0][1]}$ | $a_{[1][1]}$ | $a_{[0][2]}$ | $a_{[1][2]}$ |              |              |     |  |     |  |     |  |
| 第1列                                                                                                                       |                                                                                                                                                                                                                                                                                                                         | 第2列          |              | 第3列          |              |              |              |     |  |     |  |     |  |

图 3.19 二维数组按列优先顺序存放



### 3.4.3 特殊矩阵的压缩存储

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。

#### 1. 对称矩阵

若对一个  $n$  阶矩阵  $A$  中的任意一个元素  $a_{i,j}$  都有  $a_{i,j} = a_{j,i}$  ( $1 \leq i, j \leq n$ )，则称其为对称矩阵。其中的元素可以划分为 3 个部分，即上三角区、主对角线和下三角区，如图 3.20 所示。

对于  $n$  阶对称矩阵，上三角区的所有元素和下三角区的对应元素相同，若仍采用二维数组存放，则会浪费几乎一半的空间，为此将  $n$  阶对称矩阵  $A$  存放在一维数组  $B[n(n+1)/2]$  中，即元素  $a_{i,j}$  存放在  $b_k$  中。比如只存放下三角部分（含主对角）的元素。

在数组  $B$  中，位于元素  $a_{i,j}$  ( $i \geq j$ ) 前面的元素个数为

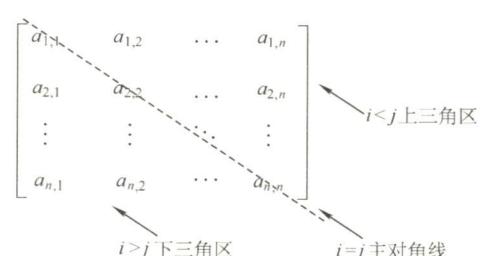


图 3.20  $n$  阶矩阵的划分

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 ( $a_{i,1}, a_{i,2}, \dots, a_{i,j-1}$ )。

因此，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：

$$k = \begin{cases} \frac{i(i-1)}{2} + j - 1, & i \geq j \text{ (下三角区和主对角线元素)} \\ \frac{j(j-1)}{2} + i - 1, & i < j \text{ (上三角区元素 } a_{ij} = a_{ji}) \end{cases}$$

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。

**注意：**二维数组  $A[n][n]$  和  $A[0..n-1][0..n-1]$  的写法是等价的。如果数组写为  $A[1..n][1..n]$ ，则说明指定了从下标 1 开始存储元素。二维数组元素写为  $a[i][j]$ ，注意数组元素下标  $i$  和  $j$  通常是从 0 开始的。矩阵元素通常写为  $a_{i,j}$  或  $a_{(i)(j)}$ ，注意行号  $i$  和列号  $j$  是从 1 开始的。

## 2. 三角矩阵

下三角矩阵 [见图 3.22(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  $n$  阶下三角矩阵  $A$  压缩存储在  $B[n(n+1)/2+1]$  中。

元素下标之间的对应关系为

$$k = \begin{cases} \frac{i(i-1)}{2} + j - 1, & i \geq j \text{ (下三角区和主对角线元素)} \\ \frac{n(n+1)}{2}, & i < j \text{ (上三角区元素)} \end{cases}$$

下三角矩阵在内存中的压缩存储形式如图 3.21 所示。

| 0         | 1         | 2         | 3         | 4         | 5         | ... | ...       | $n(n+1)/2$ |
|-----------|-----------|-----------|-----------|-----------|-----------|-----|-----------|------------|
| $a_{1,1}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ | ... | $a_{n,1}$ | $a_{n,2}$  |
| 第 1 行     | 第 2 行     | 第 3 行     | 第 $n$ 行   | 常数项       |           |     |           |            |

图 3.21 下三角矩阵的压缩存储

$$\begin{bmatrix} a_{1,1} & & & \\ a_{2,1} & a_{2,2} & & \\ \vdots & \vdots & \ddots & \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}$$

(a) 下三角矩阵

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,2} & \cdots & a_{2,n} \\ \vdots & & \ddots & \\ a_{n,n} & & & \end{bmatrix}$$

(b) 上三角矩阵

图 3.22 三角矩阵

上三角矩阵 [见图 3.22(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，可将其压缩存储在  $B[n(n+1)/2+1]$  中。

在数组 B 中，位于元素  $a_{ij}$  ( $i \leq j$ ) 前面的元素个数为  
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

因此，元素  $a_{ij}$  在数组 B 中的下标  $k = n + (n-1) + \dots + (n-i+2) + (j-i+1) - 1 = (i-1)(2n-i+2)/2 + (j-i)$ 。因此，元素下标之间的对应关系如下：

$$k = \begin{cases} \frac{(i-1)(2n-i+2)}{2} + (j-i), & i \leq j \text{ (上三角区和主对角线元素)} \\ \frac{n(n+1)}{2}, & i > j \text{ (下三角区元素)} \end{cases}$$

上三角矩阵在内存中的压缩存储形式如图 3.23 所示。

| 0         | 1         | ... |           |           |           |     |           |     |           | $n(n+1)/2$ |
|-----------|-----------|-----|-----------|-----------|-----------|-----|-----------|-----|-----------|------------|
| $a_{1,1}$ | $a_{1,2}$ | ... | $a_{1,n}$ | $a_{2,2}$ | $a_{2,3}$ | ... | $a_{2,n}$ | ... | $a_{n,n}$ | c          |
|           |           |     |           |           |           |     |           |     |           |            |
|           |           |     |           |           |           |     |           |     |           |            |

第1行                    第2行                    第n行 常数项

图 3.23 上三角矩阵的压缩存储

以上推导均假设数组的下标从 0 开始，若题设有具体要求，则应该灵活应对。



### 3. 三对角矩阵

对角矩阵也称带状矩阵。对于  $n$  阶矩阵  $A$  中的任意一个元素  $a_{ij}$ ，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 3.24 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，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 3.25 所示。

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  $k = 2i + j - 3$ 。

|           |           |               |               |             |          |  |  |  |  |   |
|-----------|-----------|---------------|---------------|-------------|----------|--|--|--|--|---|
| $a_{1,1}$ | $a_{1,2}$ |               |               |             |          |  |  |  |  | 0 |
| $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$     |               |             |          |  |  |  |  |   |
| $a_{3,2}$ | $a_{3,3}$ | $a_{3,4}$     |               |             |          |  |  |  |  |   |
|           |           |               | $\ddots$      | $\ddots$    | $\ddots$ |  |  |  |  |   |
| 0         |           | $a_{n-1,n-2}$ | $a_{n-1,n-1}$ | $a_{n-1,n}$ |          |  |  |  |  |   |
|           |           | $a_{n,n-1}$   | $a_{n,n}$     |             |          |  |  |  |  |   |

图 3.24 三对角矩阵  $A$

|           |           |           |           |           |     |             |             |           |
|-----------|-----------|-----------|-----------|-----------|-----|-------------|-------------|-----------|
| $a_{1,1}$ | $a_{1,2}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ | ... | $a_{n-1,n}$ | $a_{n,n-1}$ | $a_{n,n}$ |
|-----------|-----------|-----------|-----------|-----------|-----|-------------|-------------|-----------|

图 3.25 三对角矩阵的压缩存储

反之，若已知三对角矩阵中某元素  $a_{ij}$  存放于一维数组 B 的第  $k$  个位置，则可得  $i = \lfloor (k+1)/3+1 \rfloor$ ,  $j = k - 2i + 3$ 。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

矩阵中非零元素的个数  $t$ ，相对矩阵元素的个数  $s$  来说非常少，即  $s \gg t$  的矩阵称为稀疏矩阵。例如，一个矩阵的阶为  $100 \times 100$ ，该矩阵中只有少于 100 个非零元素。

若采用常规的方法存储稀疏矩阵，则相当浪费存储空间，因此仅存储非零元素。但通常非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 3.26 所示。然后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。

$$M = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 9 & 0 & 0 \\ 0 & 23 & 0 & 0 \end{bmatrix} \text{ 对应的三元组}$$

| $i$ | $j$ | $v$ |
|-----|-----|-----|
| 0   | 0   | 4   |
| 1   | 2   | 6   |
| 2   | 1   | 9   |
| 3   | 1   | 23  |

图 3.26 稀疏矩阵及其对应的三元组

稀疏矩阵的三元组既可以采用数组存储，也可以采用十字链表法存储。

### 3.4.5 本节试题精选

#### 单项选择题

01. 对特殊矩阵采用压缩存储的主要目的是( )。
- A. 表达变得简单
  - B. 对矩阵元素的存取变得简单
  - C. 去掉矩阵中的多余元素
  - D. 减少不必要的存储空间
02. 对  $n$  阶对称矩阵压缩存储时，需要表长为( )的顺序表。
- A.  $n/2$
  - B.  $n \times n/2$
  - C.  $n(n+1)/2$
  - D.  $n(n-1)/2$
03. 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，则第  $i+1$  行的对角元素  $A[i][i]$  存放于  $B$  中的( )处。
- A.  $(i+3)i/2$
  - B.  $(i+1)i/2$
  - C.  $(2n-i+1)i/2$
  - D.  $(2n-i-1)i/2$
04. 在二维数组  $A$  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( )。
- A.  $SA+141$
  - B.  $SA+144$
  - C.  $SA+222$
  - D.  $SA+255$
05. 将三对角矩阵  $A[1 \dots 100][1 \dots 100]$  按行优先存入一维数组  $B[1 \dots 298]$  中，数组  $A$  中元素  $A[66][65]$  在数组  $B$  中的位置  $k$  为( )。
- A. 198
  - B. 195
  - C. 197
  - D. 196
06. 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( )。
- A.  $i(i+1)/2 + j$
  - B.  $i(i-1)/2 + j - 1$
  - C.  $j(j-1)/2 + i$
  - D.  $j(j-1)/2 + i - 1$
07. 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( )。
- A.  $(j-1)(2n-j+1)/2 + i - j$
  - B.  $(j-1)(2n-j+2)/2 + i - j + 1$
  - C.  $(j-1)(2n-j+2)/2 + i - j$
  - D.  $(j-1)(2n-j+1)/2 + i - j - 1$
08. 【2016 统考真题】有一个 100 阶的三对角矩阵  $M$ ，其元素  $m_{ij}$  ( $1 \leq i, j \leq 100$ ) 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( )。
- A. 86
  - B. 87
  - C. 88
  - D. 89
09. 【2017 统考真题】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
  - B. 三元组表和邻接矩阵
  - C. 十字链表和二叉链表
  - D. 邻接矩阵和十字链表
10. 【2018 统考真题】设有一个  $12 \times 12$  的对称矩阵  $M$ ，将其上三角部分的元素  $m_{ij}$  ( $1 \leq i \leq j \leq 12$ ) 按行优先存入 C 语言的一维数组  $N$  中，元素  $m_{6,6}$  在  $N$  中的下标是( )。
- A. 50
  - B. 51
  - C. 55
  - D. 66
11. 【2020 统考真题】将一个  $10 \times 10$  对称矩阵  $M$  的上三角部分的元素  $m_{ij}$  ( $1 \leq i \leq j \leq 10$ ) 按列优先存入 C 语言的一维数组  $N$  中，元素  $m_{7,2}$  在  $N$  中的下标是( )。
- A. 15
  - B. 16
  - C. 22
  - D. 23
12. 【2021 统考真题】二维数组  $A$  按行优先方式存储，每个元素占用 1 个存储单元。若元素  $A[0][0]$  的存储地址是 100， $A[3][3]$  的存储地址是 220，则元素  $A[5][5]$  的存储地址

关注公众号【乘龙考研】  
一手更新 稳定有保障

是( )。

- A. 295      B. 300      C. 301      D. 306

### 3.4.6 答案与解析

#### 单项选择题

01. D

特殊矩阵中含有很多相同元素或零元素，故可采用压缩存储，以节省存储空间。

02. C

只需要存储其上三角或下三角部分（含对角线），元素个数为  $n + (n - 1) + (n - 2) + \dots + 1 = n(n + 1)/2$ 。

03. 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。

04. D

二维数组计算地址（按行优先顺序）的公式为

$$\text{LOC}(i, j) = \text{LOC}(0, 0) + (i \times m + j) \times L$$

其中， $\text{LOC}(0, 0) = \text{SA}$ ，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  $\text{LOC}(8, 5) = \text{SA} + (8 \times 10 + 5) \times 3 = \text{SA} + 255$ ，故选D。

05. B

对于三对角矩阵，将  $A[1..n][1..n]$  压缩至  $B[1..3n-2]$  时， $a_{i,j}$  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  $2 \times 66 + 65 - 2 = 195$ 。

06. C

按列优先存储，故元素  $a_{i,j}$  前面有  $j-1$  列，共有  $1 + 2 + 3 + \dots + j-1 = j(j-1)/2$  个元素，元素  $a_{i,j}$  在第  $j$  列上是第  $i$  个元素，数组  $B$  的下标是从1开始，因此  $k = j(j-1)/2 + i$ 。

07. B

按列优先存储，故元素  $a_{i,j}$  之前有  $j-1$  列，共有  $n + (n - 1) + \dots + (n - j + 2) = (j - 1)(2n - j + 2)/2$  个元素，元素  $a_{i,j}$  是第  $j$  列上第  $i - j + 1$  个元素，数组  $B$  下标从1开始， $k = (j - 1)(2n - j + 2)/2 + i - j + 1$ 。

08. B

三对角矩阵如下所示。

$$\begin{bmatrix} a_{1,1} & a_{1,2} & & & & \\ a_{2,1} & a_{2,2} & a_{2,3} & & & 0 \\ a_{3,2} & a_{3,3} & a_{3,4} & & & \\ \vdots & \vdots & \ddots & \ddots & & \\ 0 & & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} & \\ & & & a_{n,n-1} & a_{n,n} & \end{bmatrix}$$

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：

关注公众号【乘龙考研】  
一手更新 稳定有保障

|           |           |           |           |           |     |             |             |           |
|-----------|-----------|-----------|-----------|-----------|-----|-------------|-------------|-----------|
| $a_{1,1}$ | $a_{1,2}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ | ... | $a_{n-1,n}$ | $a_{n,n-1}$ | $a_{n,n}$ |
|-----------|-----------|-----------|-----------|-----------|-----|-------------|-------------|-----------|

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  $k = 2i + j - 3$ , 公式很难记忆, 我们通常采用解法 2。

解法 1: 针对该题, 仅需将数字逐一代入公式:  $k = 2 \times 30 + 30 - 3 = 87$ , 结果为 87。

解法 2: 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  $1 \leq i, j \leq 100$ ) 中, 每行都有 3 个元素, 而  $m_{30,30}$  之前仅有一个元素  $m_{30,29}$ , 不难发现元素  $m_{30,30}$  在数组 N 中的下标是  $2 + 28 \times 3 + 2 - 1 = 87$ 。

注意: 矩阵和数组的下标从 0 或 1 开始 (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 (方法无非是多计算 1 或少计算 1 的问题)。

### 09. 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  $O(n^2)$ , 不适合于存储稀疏矩阵。二叉链表又名左孩子右兄弟表示法, 可用于表示树或森林。A 正确。

### 10. A

在 C 语言中, 数组 N 的下标从 0 开始。第一个元素  $m_{1,1}$  对应存入  $n_0$ , 矩阵 M 的第一行有 12 个元素, 第二行有 11 个, 第三行有 10 个, 第四行有 9 个, 第五行有 8 个, 所以  $m_{6,6}$  是第  $12 + 11 + 10 + 9 + 8 + 1 = 51$  个元素, 下标应为 50。

### 11. C

上三角矩阵按列优先存储, 先存储只有 1 个元素的第一列, 再存储有 2 个元素的第二列, 以此类推。 $m_{7,2}$  位于左下角, 对应右上角的元素为  $m_{2,7}$ , 在  $m_{2,7}$  之前存有

第 1 列: 1

第 2 列: 2

.....

第 6 列: 6

第 7 列: 1

前面共存储有  $1 + 2 + 3 + 4 + 5 + 6 + 1 = 22$  个元素 (数组下标范围为 0~21), 注意数组下标从 0 开始, 故  $m_{2,7}$  在数组 N 中的下标为 22, 即  $m_{7,2}$  在数组 N 中的下标为 22。

### 12. B

二维数组 A 按行优先存储, 每个元素占用 1 个存储单元, 由 A[0][0] 和 A[3][3] 的存储地址可知 A[3][3] 是二维数组 A 中的第 121 个元素, 假设二维数组 A 的每行有 n 个元素, 则  $n \times 3 + 4 = 121$ , 求得  $n = 39$ , 故元素 A[5][5] 的存储地址为  $100 + 39 \times 5 + 6 - 1 = 300$ , 故选 B。

关注公众号【乘龙考研】  
一手更新 稳定有保障

## 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中。很多读者看到课本上有好多个函数时很恐惧, 若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, 而不必分函数写出。以顺序栈的操作为例:

(1) 声明一个栈并初始化:

```
Elemtyp 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--]; //单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，链式栈出现的概率要比顺序栈低得多，因此大家应该有所侧重，多训练与顺序栈相关的题目。

## 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，是指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，例如元素 x 进栈，则辅助栈  $stack\_min[top+1]=(x < min) ? x : min;$  即在每次 Push 中，都将当前最小元素放到  $stack\_min$  的栈顶。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。因此  $stack\_min$  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。)

关注公众号【乘龙考研】  
一手更新 稳定有保障

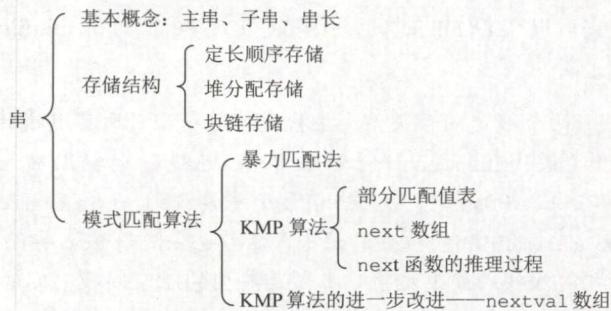
# 第 4 章 串

关注公众号【乘龙考研】  
一手更新 稳定有保障

## 【考纲内容】

字符串模式匹配

## 【知识框架】



兑换后



看视频讲解

## 【复习提示】

本章是统考大纲第 6 章内容，采纳读者建议单独作为一章，大纲只要求掌握字符串模式匹配，重点掌握 KMP 匹配算法的原理及 next 数组的推理过程，手工求 next 数组可以先计算出部分匹配值表然后变形，或根据公式来求解。了解 nextval 数组的求解方法。

## \*4.1 串的定义和实现<sup>①</sup>

字符串简称串，计算机上非数值处理的对象基本都是字符串数据。我们常见的信息检索系统（如搜索引擎）、文本编辑程序（如 Word）、问答系统、自然语言翻译系统等，都是以字符串数据作为处理对象的。本章详细介绍字符串的存储结构及相应的操作。

### 4.1.1 串的定义

串 (string) 是由零个或多个字符组成的有限序列。一般记为

$$S = 'a_1 a_2 \cdots a_n' \quad (n \geq 0)$$

其中， $S$  是串名，单引号括起来的字符序列是串的值； $a_i$  可以是字母、数字或其他字符；串中字符的个数  $n$  称为串的长度。 $n=0$  时的串称为空串（用  $\emptyset$  表示）。

串中任意多个连续的字符组成的子序列称为该串的子串，包含子串的串称为主串。某个字

<sup>①</sup> 本节不在统考大纲范围，仅供学习参考。

符在串中的序号称为该字符在串中的位置。子串在主串中的位置以子串的第一个字符在主串中的位置来表示。当两个串的长度相等且每个对应位置的字符都相等时，称这两个串是相等的。

例如，有串 A='China Beijing', B='Beijing', C='China'，则它们的长度分别为 13, 7 和 5。B 和 C 是 A 的子串，B 在 A 中的位置是 7，C 在 A 中的位置是 1。

需要注意的是，由一个或多个空格（空格是特殊字符）组成的串称为空格串（注意，空格串不是空串），其长度为串中空格字符的个数。

串的逻辑结构和线性表极为相似，区别仅在于串的数据对象限定为字符集。在基本操作上，串和线性表有很大差别。线性表的基本操作主要以单个元素作为操作对象，如查找、插入或删除某个元素等；而串的基本操作通常以子串作为操作对象，如查找、插入或删除一个子串等。

### 4.1.2 串的存储结构

#### 1. 定长顺序存储表示

类似于线性表的顺序存储结构，用一组地址连续的存储单元存放串值的字符序列。在串的定长顺序存储结构中，为每个串变量分配一个固定长度的存储区，即定长数组。

```
#define MAXLEN 255 //预定义最大串长为 255
typedef struct{
 char ch[MAXLEN]; //每个分量存储一个字符
 int length; //串的实际长度
}SString;
```



串的实际长度只能小于或等于 MAXLEN，超过预定义长度的串值会被舍去，称为截断。串长有两种表示方法：一是如上述定义描述的那样，用一个额外的变量 len 来存放串的长度；二是在串值后面加一个不计入串长的结束标记字符“\0”，此时的串长为隐含值。

在一些串的操作（如插入、联接等）中，若串值序列的长度超过上界 MAXLEN，约定用“截断”法处理，要克服这种弊端，只能不限定串长的最大长度，即采用动态分配的方式。

#### 2. 堆分配存储表示

堆分配存储表示仍然以一组地址连续的存储单元存放串值的字符序列，但它们的存储空间是在程序执行过程中动态分配得到的。

```
typedef struct{
 char *ch; //按串长分配存储区，ch 指向串的基址
 int length; //串的长度
}HString;
```

在 C 语言中，存在一个称之为“堆”的自由存储区，并用 malloc() 和 free() 函数来完成动态存储管理。利用 malloc() 为每个新产生的串分配一块实际串长所需的存储空间，若分配成功，则返回一个指向起始地址的指针，作为串的基地址，这个串由 ch 指针来指示；若分配失败，则返回 NULL。已分配的空间可用 free() 释放掉。

上述两种存储表示通常为高级程序设计语言所采用。块链存储表示仅做简单介绍。

#### 3. 块链存储表示

类似于线性表的链式存储结构，也可采用链表方式存储串值。由于串的特殊性（每个元素只有一个字符），在具体实现时，每个结点既可以存放一个字符，也可以存放多个字符。每个结点称为块，整个链表称为块链结构。图 4.1(a)是结点大小为 4（即每个结点存放 4 个字符）的链表，最后一个结点占不满时通常用“#”补上；图 4.1(b)是结点大小为 1 的链表。

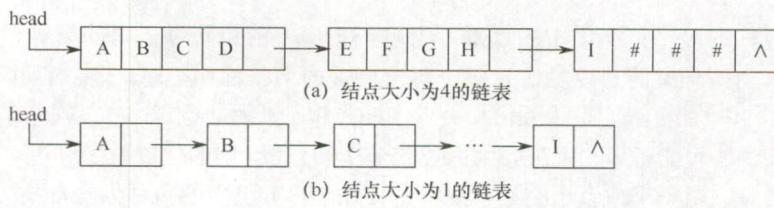


图 4.1 串值的链式存储方式

### 4.1.3 串的基本操作

- `StrAssign (&T, chars)`: 赋值操作。把串 T 赋值为 chars。
- `StrCopy (&T, S)`: 复制操作。由串 S 复制得到串 T。
- `StrEmpty (S)`: 判空操作。若 S 为空串，则返回 TRUE，否则返回 FALSE。
- `StrCompare (S, T)`: 比较操作。若 S>T，则返回值>0；若 S=T，则返回值=0；若 S<T，则返回值<0。
- `StrLength (S)`: 求串长。返回串 S 的元素个数。
- `SubString (&Sub, S, pos, len)`: 求子串。用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。
- `Concat (&T, S1, S2)`: 串联接。用 T 返回由 S1 和 S2 联接而成的新串。
- `Index (S, T)`: 定位操作。若主串 S 中存在与串 T 值相同的子串，则返回它在主串 S 中第一次出现的位置；否则函数值为 0。
- `ClearString (&S)`: 清空操作。将 S 清为空串。
- `DestroyString (&S)`: 销毁串。将串 S 销毁。

不同的高级语言对串的基本操作集可以有不同的定义方法。在上述定义的操作中，串赋值 `StrAssign`、串比较 `StrCompare`、求串长 `StrLength`、串联接 `Concat` 及求子串 `SubString` 五种操作构成串类型的最小操作子集，即这些操作不可能利用其他串操作来实现；反之，其他串操作（除串清除 `ClearString` 和串销毁 `DestroyString` 外）均可在该最小操作子集上实现。

## 4.2 串的模式匹配

关注公众号【乘龙考研】  
一手更新 稳定有保障

### 4.2.1 简单的模式匹配算法

子串的定位操作通常称为串的模式匹配，它求的是子串（常称模式串）在主串中的位置。这里采用定长顺序存储结构，给出一种不依赖于其他串操作的暴力匹配算法。

```
int Index (SString S, SString T) {
 int i=1, j=1;
 while(i<=S.length && j<=T.length) {
 if(S.ch[i]==T.ch[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.length) return i-T.length;
}
```

```
 else return 0;
```

在上述算法中，分别用计数指针  $i$  和  $j$  指示主串  $s$  和模式串  $t$  中当前正待比较的字符位置。算法思想为：从主串  $s$  的第一个字符起，与模式  $t$  的第一个字符比较，若相等，则继续逐个比较后续字符；否则从主串的下一个字符起，重新和模式的字符比较；以此类推，直至模式  $t$  中的每个字符依次和主串  $s$  中的一个连续的字符序列相等，则称匹配成功，函数值为与模式  $t$  中第一个字符相等的字符在主串  $s$  中的序号，否则称匹配不成功，函数值为零。图 4.2 展示了模式  $t='abacac'$  和主串  $s$  的匹配过程，每次匹配失败后，都把模式  $t$  后移一位。

#### 4.2.2 串的模式匹配算法——KMP 算法

图 4.2 的匹配过程，在第三趟匹配中， $i=7$ 、 $j=5$  的字符比较不等，于是又从  $i=4$ 、 $j=1$  重新开始比较。然而，仔细观察会发现， $i=4$  和  $j=1$ ， $i=5$  和  $j=1$  及  $i=6$  和  $j=1$  这三次比较都是不必进行的。从第三趟部分匹配的结果可知，主串中第 4、5 和 6 个字符是 'b'、'c' 和 'a'（即模式中第 2、3 和 4 个字符），因为模式中第一个字符是 'a'，因此它无须再和这 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  $i=7$ 、 $j=2$  时的比较即可。

在暴力匹配中，每趟匹配失败都是模式后移一位再从头开始比较。而某趟已匹配相等的字符序列是模式的某个前缀，这种频繁的重复比较相当于模式串在不断地进行自我比较，这就是其低效率的根源。因此，可以从分析模式本身的结构着手，如果已匹配相等的前缀序列中有某个后缀正好是模式的前缀，那么就可以将模式向后滑动到与这些相等字符对齐的位置，主串  $i$  指针无须回溯，并从该位置开始继续比较。而模式向后滑动位数的计算仅与模式本身的结构有关，与主串无关（这里理解起来会比较困难，没关系，带着这个问题继续往后看）。

### 1. 字符串的前缀、后缀和部分匹配值

要了解子串的结构，首先要弄清楚几个概念：前缀、后缀和部分匹配值。前缀指除最后一个字符以外，字符串的所有头部子串；后缀指除第一个字符外，字符串的所有尾部子串；部分匹配值则为字符串的前缀和后缀的最长相等前后缀长度。下面以'ababa'为例进行说明：

- 'a'的前缀和后缀都为空集，最长相等前后缀长度为0。
  - 'ab'的前缀为{a}，后缀为{b}， $\{a\} \cap \{b\} = \emptyset$ ，最长相等前后缀长度为0。
  - 'aba'的前缀为{a, ab}，后缀为{a, ba}， $\{a, ab\} \cap \{a, ba\} = \{a\}$ ，最长相等前后缀长

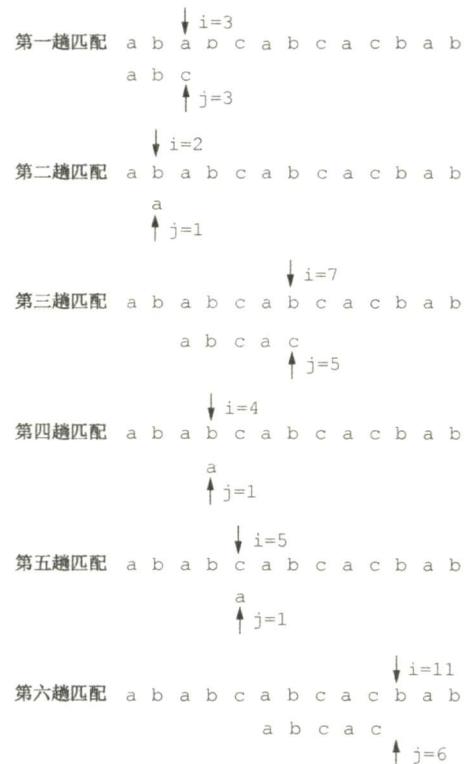


图 4.2 简单模式匹配算法举例

度为 1。

- 'abab' 的前缀 {a, ab, aba} ∩ 后缀 {b, ab, bab} = {ab}, 最长相等前后缀长度为 2。
- 'ababa' 的前缀 {a, ab, aba, abab} ∩ 后缀 {a, ba, aba, baba} = {a, aba}, 公共元素有两个，最长相等前后缀长度为 3。

故字符串 'ababa' 的部分匹配值为 00123。

这个部分匹配值有什么作用呢？

回到最初的问题，主串为 a b a b c a b c a c b a b，子串为 a b c a c。

利用上述方法容易写出子串 'abcac' 的部分匹配值为 00010，将部分匹配值写成数组形式，就得到了部分匹配值（Partial Match, PM）的表。

| 编号 | 1 | 2 | 3 | 4 | 5 |
|----|---|---|---|---|---|
| S  | a | b | c | a | c |
| PM | 0 | 0 | 0 | 1 | 0 |

关注公众号【乘龙考研】  
一手更新 稳定有保障

下面用 PM 表来进行字符串匹配：

|    |   |   |   |   |   |   |   |   |   |   |   |   |   |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 主串 | a | b | a | b | c | a | b | c | a | c | b | a | b |
| 子串 | a | b | c |   |   |   |   |   |   |   |   |   |   |

第一趟匹配过程：

发现 c 与 a 不匹配，前面的 2 个字符 'ab' 是匹配的，查表可知，最后一个匹配字符 b 对应的部分匹配值为 0，因此按照下面的公式算出子串需要向后移动的位数：

$$\text{移动位数} = \text{已匹配的字符数} - \text{对应的部分匹配值}$$

因为  $2 - 0 = 2$ ，所以将子串向后移动 2 位，如下进行第二趟匹配：

|    |   |   |   |   |   |   |   |   |   |   |   |   |   |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 主串 | a | b | a | b | c | a | b | c | a | c | b | a | b |
| 子串 |   |   | a | b | c | a | c |   |   |   |   |   |   |

第二趟匹配过程：

发现 c 与 b 不匹配，前面 4 个字符 'abca' 是匹配的，最后一个匹配字符 a 对应的部分匹配值为 1， $4 - 1 = 3$ ，将子串向后移动 3 位，如下进行第三趟匹配：

|    |   |   |   |   |   |   |   |   |   |   |   |   |   |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 主串 | a | b | a | b | c | a | b | c | a | c | b | a | b |
| 子串 |   |   |   | a | b | c | a | c |   |   |   |   |   |

第三趟匹配过程：

子串全部比较完成，匹配成功。整个匹配过程中，主串始终没有回退，故 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，大大提高了匹配效率。

某趟发生失配时，如果对应的部分匹配值为 0，那么表示已匹配相等序列中没有相等的前后缀，此时移动的位数最大，直接将子串首字符后移到主串当前位置进行下一趟比较；如果已匹配相等序列中存在最大相等前后缀（可理解为首尾重合），那么将子串向右滑动到和该相等前后缀对齐（这部分字符下一趟显然不需要比较），然后从主串当前位置进行下一趟比较。

## 2. KMP 算法的原理是什么？

我们刚刚学会了怎样计算字符串的部分匹配值、怎样利用子串的部分匹配值快速地进行字符串匹配操作，但公式“移动位数 = 已匹配的字符数 - 对应的部分匹配值”的意义是什么呢？

如图 4.3 所示，当 c 与 b 不匹配时，已匹配 'abca' 的前缀 a 和后缀 a 为最长公共元素。已知前缀 a 与 b、c 均不同，与后缀 a 相同，故无须比较，直接将子串移动“已匹配的字符数 - 对应的部分匹配值”，用子串前缀后面的元素与主串匹配失败的元素开始比较即可，如图 4.4 所示。

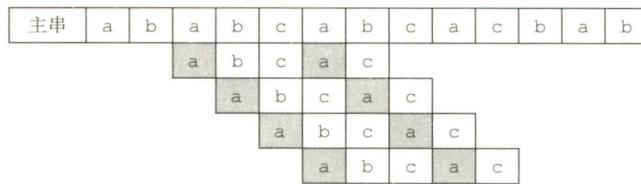


图 4.3 失配后移动情况

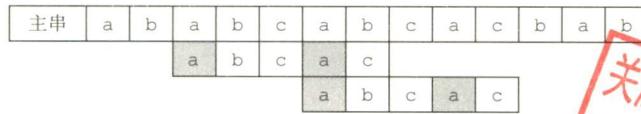


图 4.4 直接移动到合适位置

对算法的改进方法：

已知：右移位数 = 已匹配的字符数 - 对应的部分匹配值。

写成： $\text{Move} = (j-1) - \text{PM}[j-1]$ 。

使用部分匹配值时，每当匹配失败，就去找它前一个元素的部分匹配值，这样使用起来有些不方便，所以将 PM 表右移一位，这样哪个元素匹配失败，直接看它自己的部分匹配值即可。

将上例中字符串 'abcac' 的 PM 表右移一位，就得到了 next 数组：

| 编号   | 1  | 2 | 3 | 4 | 5 |
|------|----|---|---|---|---|
| S    | a  | b | c | a | c |
| next | -1 | 0 | 0 | 0 | 1 |

我们注意到：

- 1) 第一个元素右移以后空缺的用 -1 来填充，因为若是第一个元素匹配失败，则需要将子串向右移动一位，而不需要计算子串移动的位数。
- 2) 最后一个元素在右移的过程中溢出，因为原来的子串中，最后一个元素的部分匹配值是其下一个元素使用的，但显然已没有下一个元素，故可以舍去。

这样，上式就改写为

$$\text{Move} = (j-1) - \text{next}[j]$$

相当于将子串的比较指针  $j$  回退到

$$j=j-\text{Move}=j-((j-1)-\text{next}[j])=\text{next}[j]+1$$

有时为了使公式更加简洁、计算简单，将 next 数组整体 +1。

因此，上述子串的 next 数组也可以写成

| 编号   | 1 | 2 | 3 | 4 | 5 |
|------|---|---|---|---|---|
| S    | a | b | c | a | c |
| next | 0 | 1 | 1 | 1 | 2 |

最终得到子串指针变化公式  $j=\text{next}[j]$ 。在实际匹配过程中，子串在内存里是不会移动的，而是指针在变化，画图举例只是为了让问题描述得更为形象。next[j] 的含义是：在子串的第  $j$  个字符与主串发生失配时，则跳到子串的  $\text{next}[j]$  位置重新与主串当前位置进行比较。

如何推理 next 数组的一般公式？设主串为 ' $s_1s_2\dots s_n$ '，模式串为 ' $p_1p_2\dots p_m$ '，当主串中第  $i$  个字符与模式串中第  $j$  个字符失配时，子串应向右滑动多远，然后与模式中的哪个字符比较？

假设此时应与模式中第  $k$  ( $k < j$ ) 个字符继续比较，则模式中前  $k-1$  个字符的子串必须满足

下列条件，且不可能存在  $k' > k$  满足下列条件：

$$'p_1 p_2 \cdots p_{k-1}' = 'p_{j-k+1} p_{j-k+2} \cdots p_{j-1}'$$

若存在满足如上条件的子串，则发生失配时，仅需将模式向右滑动至模式中第  $k$  个字符和主串第  $i$  个字符对齐，此时模式中前  $k-1$  个字符的子串必定与主串中第  $i$  个字符之前长度为  $k-1$  的子串相等，由此，只需从模式第  $k$  个字符与主串第  $i$  个字符继续比较即可，如图 4.5 所示。

|    |       |     |       |     |           |     |             |     |           |       |     |       |       |     |     |       |
|----|-------|-----|-------|-----|-----------|-----|-------------|-----|-----------|-------|-----|-------|-------|-----|-----|-------|
| 主串 | $s_1$ | ... | ...   | ... | ...       | ... | $s_{i-k+1}$ | ... | $s_{i-1}$ | $s_i$ | ... | ...   | ...   | ... | ... | $s_n$ |
| 子串 |       |     | $p_1$ | ... | $p_{k-1}$ | ... | $p_{j-k+1}$ | ... | $p_{j-1}$ | $p_i$ | ... | $p_m$ |       |     |     |       |
| 右移 |       |     |       |     |           |     | $p_1$       | ... | $p_{k-1}$ | $p_k$ | ... | ...   | $p_m$ |     |     |       |

图 4.5 模式串右移到合适位置（阴影对齐部分表示上下字符相等）

当模式串已匹配相等序列中不存在满足上述条件的子串时（可以看成  $k=1$ ），显然应该将模式串右移  $j-1$  位，让主串第  $i$  个字符和模式第一个字符进行比较，此时右移位数最大。

当模式串第一个字符（ $j=1$ ）与主串第  $i$  个字符发生失配时，规定  $\text{next}[1]=0$ <sup>①</sup>。将模式串右移一位，从主串的下一个位置（ $i+1$ ）和模式串的第一个字符继续比较。

通过上述分析可以得出  $\text{next}$  函数的公式：

$$\text{next}[j] = \begin{cases} 0, & j=1 \\ \max\{k | 1 < k < j \text{ 且 } 'p_1 \cdots p_{k-1}' = 'p_{j-k+1} \cdots p_{j-1}'\}, & \text{当此集合不空时} \\ 1, & \text{其他情况} \end{cases}$$

上述公式不难理解，实际做题求  $\text{next}$  值时，用之前的方法也很好求，但如果想用代码来实现，貌似难度还真不小，我们来尝试推理论解的科学步骤。

首先由公式可知

$$\text{next}[1]=0$$

设  $\text{next}[j]=k$ ，此时  $k$  应满足的条件在上文中已描述。

此时  $\text{next}[j+1]=?$  可能有两种情况：

(1) 若  $p_k=p_j$ ，则表明在模式串中

$$'p_1 \cdots p_{k-1} p_k' = 'p_{j-k+1} \cdots p_{j-1} p_j'$$

并且不可能存在  $k' > k$  满足上述条件，此时  $\text{next}[j+1]=k+1$ ，即

$$\text{next}[j+1]=\text{next}[j]+1$$

(2) 若  $p_k \neq p_j$ ，则表明在模式串中

$$'p_1 \cdots p_{k-1} p_k' \neq 'p_{j-k+1} \cdots p_{j-1} p_j'$$

此时可以把求  $\text{next}$  函数值的问题视为一个模式匹配的问题。用前缀  $p_1 \cdots p_k$  去跟后缀  $p_{j-k+1} \cdots p_j$  匹配，则当  $p_k \neq p_j$  时应将  $p_1 \cdots p_k$  向右滑动至以第  $\text{next}[k]$  个字符与  $p_j$  比较，如果  $\text{next}[k]$  与  $p_j$  还是不匹配，那么需要寻找长度更短的相等前后缀，下一步继续用  $P_{\text{next}[\text{next}[k]]}$  与  $p_j$  比较，以此类推，直到找到某个更小的  $k'=\text{next}[\text{next} \cdots [k]]$  ( $1 < k' < k < j$ )，满足条件

$$'p_1 \cdots p_{k'}' = 'p_{j-k'+1} \cdots p_j'$$

则  $\text{next}[j+1]=k'+1$ 。

也可能不存在任何  $k'$  满足上述条件，即不存在长度更短的相等前后缀，令  $\text{next}[j+1]=1$ 。理解起来有一点费劲？下面举一个简单的例子。

图 4.6 的模式串中已求得 6 个字符的  $\text{next}$  值，现求  $\text{next}[7]$ ，因为  $\text{next}[6]=3$ ，又  $p_6 \neq p_3$ ，

<sup>①</sup> 可理解为将主串第  $i$  个字符和模式串第一个字符的前面空位置对齐，也即模式串右移一位。

则需比较  $p_6$  和  $p_1$ (因  $\text{next}[3]=1$ ), 由于  $p_6 \neq p_1$ , 而  $\text{next}[1]=0$ , 所以  $\text{next}[7]=1$ ; 求  $\text{next}[8]$ , 因  $p_7=p_1$ , 则  $\text{next}[8]=\text{next}[7]+1=2$ ; 求  $\text{next}[9]$ , 因  $p_8=p_2$ , 则  $\text{next}[9]=3$ 。

|         |   |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|
| j       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 模式      | a | b | a | a | b | c | a | b | a |
| next[j] | 0 | 1 | 1 | 2 | 2 | 3 | ? | ? | ? |

图 4.6 求模式串的 next 值

通过上述分析写出求 next 值的程序如下:

```
void get_next(SString T, int next[]){
 int i=1, j=0;
 next[1]=0;
 while(i<T.length){
 if(j==0 || T.ch[i]==T.ch[j]){
 ++i; ++j;
 next[i]=j; //若 $p_i=p_j$, 则 $\text{next}[j+1]=\text{next}[j]+1$
 }
 else
 j=next[j]; //否则令 $j=\text{next}[j]$, 循环继续
 }
}
```

计算机执行起来效率很高, 但对于我们手工计算来说会很难。因此, 当我们需要手工计算时, 还是用最初的方法。

与 next 数组的求解相比, KMP 的匹配算法相对要简单很多, 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, 则应从主串的第  $i+1$  个位置开始匹配。具体代码如下:

```
int Index_KMP(SString S, SString T, int next[]){
 int i=1, j=1;
 while(i<=S.length && j<=T.length){
 if(j==0 || S.ch[i]==T.ch[j]){
 ++i; ++j; //继续比较后继字符
 }
 else
 j=next[j]; //模式串向右移动
 }
 if(j>T.length)
 return i-T.length; //匹配成功
 else
 return 0;
}
```

尽管普通模式匹配的时间复杂度是  $O(mn)$ , KMP 算法的时间复杂度是  $O(m+n)$ , 但在一般情况下, 普通模式匹配的实际执行时间近似为  $O(m+n)$ , 因此至今仍被采用。KMP 算法仅在主串与子串有很多“部分匹配”时才显得比普通算法快得多, 其主要优点是主串不回溯。

### 4.2.3 KMP 算法的进一步优化

前面定义的 next 数组在某些情况下尚有缺陷, 还可以进一步优化。如图 4.7 所示, 模式 'aaaab' 在和主串 'aaabaaaab' 进行匹配时:

|            |   |   |   |   |   |   |   |   |   |
|------------|---|---|---|---|---|---|---|---|---|
| 主串         | a | a | a | b | a | a | a | a | b |
| 模式         | a | a | a | a | b |   |   |   |   |
| j          | 1 | 2 | 3 | 4 | 5 |   |   |   |   |
| next[j]    | 0 | 1 | 2 | 3 | 4 |   |   |   |   |
| nextval[j] | 0 | 0 | 0 | 0 | 4 |   |   |   |   |

图 4.7 KMP 算法进一步优化示例

当  $i=4$ 、 $j=4$  时,  $s_4$  跟  $p_4$  ( $b \neq a$ ) 失配, 如果用之前的  $next$  数组还需要进行  $s_4$  与  $p_3$ 、 $s_4$  与  $p_2$ 、 $s_4$  与  $p_1$  这 3 次比较。事实上, 因为  $p_{next[4]}=p_4=a$ 、 $p_{next[3]}=p_3=a$ 、 $p_{next[2]}=p_2=a$ , 显然后面 3 次用一个和  $p_4$  相同的字符跟  $s_4$  比较毫无意义, 必然失配。那么问题出在哪里呢?

问题在于不应该出现  $p_j=p_{next[j]}$ 。理由是: 当  $p_j \neq s_j$  时, 下次匹配必然是  $p_{next[j]}$  跟  $s_j$  比较, 如果  $p_j=p_{next[j]}$ , 那么相当于拿一个和  $p_j$  相等的字符跟  $s_j$  比较, 这必然导致继续失配, 这样的比较毫无意义。那么如果出现了  $p_j=p_{next[j]}$  应该如何处理呢?

如果出现了, 则需要再次递归, 将  $next[j]$  修正为  $next[next[j]]$ , 直至两者不相等为止, 更新后的数组命名为  $nextval$ 。计算  $next$  数组修正值的算法如下, 此时匹配算法不变。

```
void get_nextval(SString T, int nextval[]) {
 int i=1, j=0;
 nextval[1]=0;
 while(i<T.length){
 if(j==0 || T.ch[i]==T.ch[j]){
 ++i; ++j;
 if(T.ch[i]!=T.ch[j]) nextval[i]=j;
 else nextval[i]=nextval[j];
 }
 else
 j=nextval[j];
 }
}
```

KMP 算法对于初学者来说可能不太容易理解, 读者可以尝试多读几遍本章的内容, 并参考一些其他教材的相关内容来巩固这个知识点。

#### 4.2.4 本节试题精选

##### 一、单项选择题

01. 设有两个串  $S_1$  和  $S_2$ , 求  $S_2$  在  $S_1$  中首次出现的位置的运算称为( )。
  - A. 求子串
  - B. 判断是否相等
  - C. 模式匹配
  - D. 连接
02. 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
03. 设主串的长度为  $n$ , 子串的长度为  $m$ , 则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.  $O(mn)$
  - D.  $O(m+n)$
04. 已知串  $S='aaab'$ , 其  $next$  数组值为( )。
  - A. 0123
  - B. 0112
  - C. 0231
  - D. 1211
05. 串 'ababaaababaaa' 的  $next$  数组值为( )。
  - A. 01234567899
  - B. 012121111212
  - C. 011234223456
  - D. 0123012322345

06. 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
07. 串 'ababaaababaaa' 的 nextval 数组为 ( )。  
 A. 0, 1, 0, 1, 1, 2, 0, 1, 0, 1, 0, 2      B. 0, 1, 0, 1, 1, 4, 1, 1, 0, 1, 0, 2  
 C. 0, 1, 0, 1, 0, 4, 2, 1, 0, 1, 0, 4      D. 0, 1, 1, 1, 0, 2, 1, 1, 0, 1, 0, 4
08. 【2015 统考真题】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.  $i=6, j=2$
09. 【2019 统考真题】设主串 T='abaabaabcbabaabc'，模式串 S = 'abaabc'，采用 KMP 算法进行模式匹配，到匹配成功时为止，在匹配过程中进行的单个字符间的比较次数是( )。  
 A. 9      B. 10      C. 12      D. 15

## 二、综合应用题

01. 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：

$$\text{next}[j] = \begin{cases} 0, & j=1 \\ \max\{k \mid 1 < k < j \text{ 且 } p_1 L p_{k-1} = p_{j-k+1} L p_{j-1}\}, & \text{此集合不空时} \\ 1, & \text{其他情况} \end{cases}$$

- 1) 当  $j=1$  时，为什么要取  $\text{next}[1]=0$ ?  
 2) 为什么要取  $\max\{k\}$ ， $k$  最大是多少?  
 3) 其他情况是什么情况，为什么取  $\text{next}[j]=1$ ?  
 02. 设有字符串 S='aabaaabaabaac'，P='aabaaac'。  
 1) 求出 P 的 next 数组。  
 2) 若 S 作主串，P 作模式串，试给出 KMP 算法的匹配过程。

### 4.2.5 答案与解析

#### 一、单项选择题

01. C

求子串操作是从串  $S$  中截取第  $i$  个字符起长度为  $l$  的子串，A 错。BD 明显错。选 C。

02. B

在 KMP 算法的比较过程中，主串不会回溯，所以主串的指针不会变小。选 B。

03. C、D

尽管实际应用中，一般情况下简单的模式匹配算法的时间复杂度近似为  $O(m + n)$ ，但它的理论时间复杂度还是  $O(mn)$ ，选 C。KMP 算法的时间复杂度为  $O(m + n)$ ，选 D。

04. 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)  $j=3$  时  $k=\text{next}[j-1]=\text{next}[2]=1$ ，观察  $S[j-1]$  ( $S[2]$ ) 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)  $j=4$  时  $k=\text{next}[j-1]=\text{next}[3]=2$ , 观察  $S[j-1]$  ( $S[3]$ ) 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 |
| $\uparrow k=2$ |   |   |   |

最后的结果如下, 选 A。

| 编号   | 1 | 2 | 3 | 4 |
|------|---|---|---|---|
| S    | a | a | a | b |
| next | 0 | 1 | 2 | 3 |

本题采用 next 的推理原理求解, 如果字符串较长, 求解过程会比较烦琐。

### 05. C

这道题采用手工求 next 数组的方法。先求串  $S='ababaaababaa'$  的部分匹配值:

- 'a' 的前后缀都为空, 最长相等前后缀长度为 0。
- 'ab' 的前缀 {a}  $\cap$  后缀 {b} =  $\emptyset$ , 最长相等前后缀长度为 0。
- 'aba' 的前缀 {a, ab}  $\cap$  后缀 {a, ba} = {a}, 最长相等前后缀长度为 1。
- 'bab' 的前缀 {a, ab, aba}  $\cap$  后缀 {b, ab, bab} = {ab}, 最长相等前后缀长度为 2。
- .....

依次求出的部分匹配值如下表第三行所示, 将其整体右移一位, 低位用 -1 填充, 如下表第四行所示。

| 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  |
| PM   | 0  | 0 | 1 | 2 | 3 | 1 | 1 | 2 | 3 | 4  | 5  | 6  |
| next | -1 | 0 | 0 | 1 | 2 | 3 | 1 | 1 | 2 | 3  | 4  | 5  |

选项中  $\text{next}[1]$  等于 0, 故将 next 数组整体加 1, 答案选 C。

### 06. C

解析见上题, 选 C。注意, next 数组是否整体加 1 都正确, 需根据题意具体分析。

注意: 在实际 KMP 算法中, 为了使公式更简洁、计算简单, 如果串的位序是从 1 开始的, 则 next 数组才需要整体加 1; 如果串的位序是从 0 开始的, 则 next 数组不需要整体加 1。

### 07. C

$\text{nextval}$  从 0 开始, 可知串的位序从 1 开始。第一步, 令  $\text{nextval}[1]=\text{next}[1]=0$ 。

| 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        | 1        | 2        | 3        | 4        | 2        | 2        | 3        | 4        | 5        | 6        |
| nextval | 0 | <u>1</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>4</u> | <u>2</u> | <u>1</u> | <u>0</u> | <u>1</u> | <u>0</u> | <u>4</u> |

从  $j=2$  开始, 依次判断  $p_j$  是否等于  $p_{next[j]}$ ? 若是则将  $next[j]$  修正为  $next[next[j]]$ , 直至两者不相等为止。由下述推理可知, 答案选 C。

第2步:  $p_2=b$ 、 $p_{next[2]}=a$ ,  $p_2 \neq p_{next[2]}$ ,  $nextval[2]=next[2]=1$ ;

第3步:  $p_3=a$ 、 $p_{next[3]}=a$ ,  $p_3=p_{next[3]}$ ,  $nextval[3]=nextval[next[3]]=nextval[1]=0$ ;

第4步:  $p_4=b$ 、 $p_{next[4]}=b$ ,  $p_4=p_{next[4]}$ ,  $nextval[4]=nextval[next[4]]=nextval[2]=1$ ;

第5步:  $p_5=a$ 、 $p_{next[5]}=a$ ,  $p_5=p_{next[5]}$ ,  $nextval[5]=nextval[next[5]]=nextval[3]=0$ ;

第6步:  $p_6=a$ 、 $p_{next[6]}=b$ ,  $p_6 \neq p_{next[6]}$ ,  $nextval[6]=next[6]=4$ ;

第7步:  $p_7=a$ 、 $p_{next[7]}=b$ ,  $p_7 \neq p_{next[7]}$ ,  $nextval[7]=next[7]=2$ ;

第8步:  $p_8=b$ 、 $p_{next[8]}=b$ ,  $p_8=p_{next[8]}$ ,  $nextval[8]=nextval[next[8]]=nextval[2]=1$ ;

第9步:  $p_9=a$ 、 $p_{next[9]}=a$ ,  $p_9=p_{next[9]}$ ,  $nextval[9]=nextval[next[9]]=nextval[3]=0$ ;

第10步:  $p_{10}=b$ 、 $p_{next[10]}=b$ ,  $p_{10}=p_{next[10]}$ ,  $nextval[10]=nextval[next[10]]=nextval[4]=1$ ;

第11步:  $p_{11}=a$ 、 $p_{next[11]}=a$ ,  $p_{11}=p_{next[11]}$ ,  $nextval[11]=nextval[next[11]]=nextval[5]=0$ ;

第12步:  $p_{12}=a$ 、 $p_{next[12]}=a$ ,  $p_{12}=p_{next[12]}$ ,  $nextval[12]=nextval[next[12]]=nextval[6]=4$ ;

在第5步的推理中,  $p_5=p_{next[5]}=a$ , 按前面的讲解部分, 应该继续让  $p_3$  和  $p_{next[3]}$  比较 (恰好  $p_3=p_{next[3]}=1$ ), 注意到此时  $nextval[3]$  的值已存在, 故直接将  $nextval[5]$  赋值为  $nextval[3]$ 。

对于一般情况,  $nextval$  数组是从前往后逐步求解的, 发生  $p_j=p_{next[j]}$  时, 因为  $nextval[next[j]]$  早已求得, 所以直接将  $nextval[j]$  赋值为  $nextval[next[j]]$ 。

### 08. 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 |

关注公众号【乘龙考研】  
一手更新 稳定有保障

发生失配时, 主串指针  $i$  不变, 子串指针  $j$  回退到  $next[j]$  位置重新比较, 当  $s[i] \neq t[j]$  时,  $i=j=5$ , 由  $next$  表得知  $next[j]=next[5]=2$  (位序从 0 开始)。因此,  $i=5$ ,  $j=2$ 。

### 09. B

假设位序从 0 开始的, 按照  $next$  数组生成算法, 对于  $s$  有

| 编号   | 0  | 1 | 2 | 3 | 4 | 5 |
|------|----|---|---|---|---|---|
| s    | a  | b | a | a | b | c |
| next | -1 | 0 | 0 | 1 | 1 | 2 |

第一趟连续比较 6 次, 在模式串的 5 号位和主串的 5 号位匹配失败, 模式串的下一个比较位置为  $next[5]$ , 即下一次比较从模式串的 2 号位和主串的 5 号位开始, 然后直到模式串 5 号位和主串 8 号位匹配, 第二趟比较 4 次, 匹配成功。单个字符的比较次数为 10 次, 因此选 B。

## 二、综合应用题

### 01. 【解答】

- 当模式串中的第一个字符与主串的当前字符比较不相等时,  $next[1]=0$ , 表示模式串应右移一位, 主串当前指针后移一位, 再和模式串的第一字符进行比较。
- 当主串的第  $i$  个字符与模式串的第  $j$  个字符失配时, 主串  $i$  不回溯, 则假定模式串的第  $k$  个字符与主串的第  $i$  个字符比较,  $k$  值应满足条件  $1 < k < j$  且 ' $p_1 \cdots p_{k-1}$ ' = ' $p_{j-k+1} \cdots p_{j-1}$ ',