HZ BOOKS
华章IT

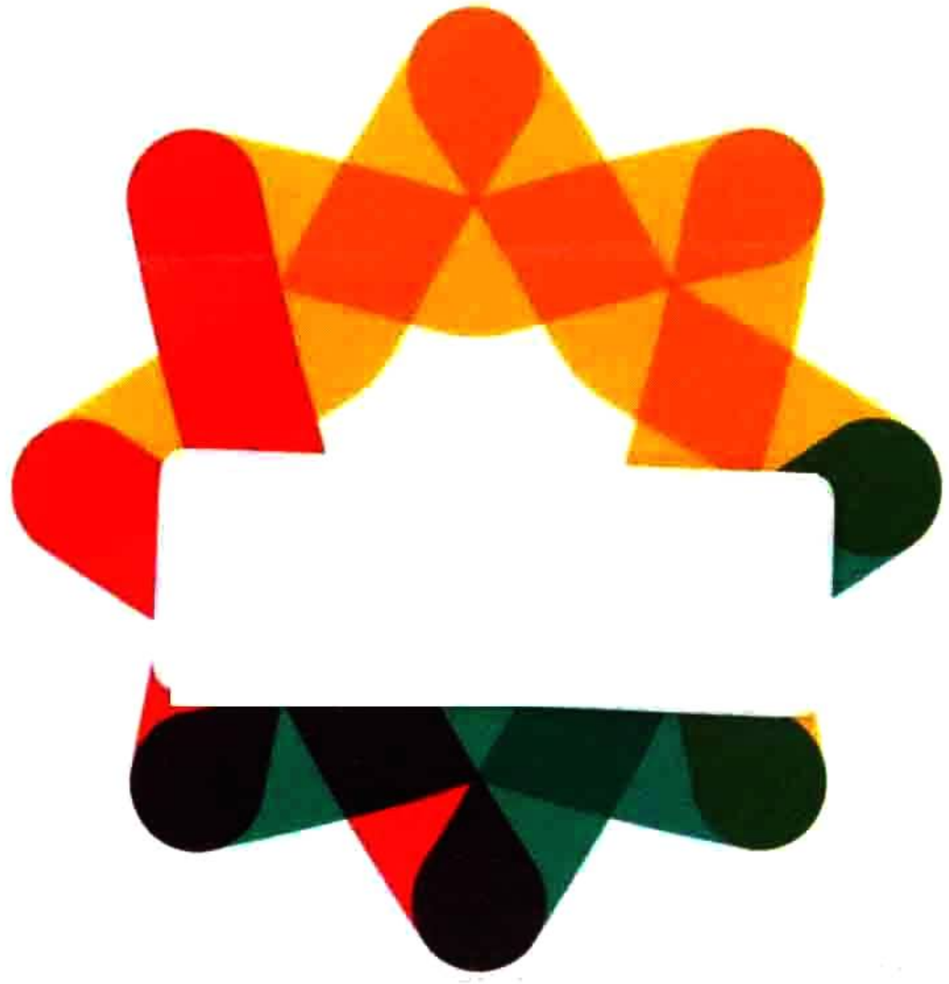
涵盖世界知名IT公司技术面试的程序设计问题及其解题思路
解析IT顶尖企业（微软、谷歌、亚马逊、雅虎、脸谱、苹果、Adobe）的面试过程
针对不同问题，提供多个具有不同复杂度的解决方法。兼顾自学教材和面试辅导的不同需求

# 数据结构与算法经典问题解析

## Java语言描述

（原书第2版）

[印] 纳拉辛哈·卡鲁曼希 (Narasimha Karumanchi) 著　骆嘉伟 李晓鸿 肖正 吴帆 等译



DATA STRUCTURES AND ALGORITHMS MADE EASY IN JAVA

SECOND EDITION

机械工业出版社
China Machine Press



# DATA STRUCTURES AND ALGORITHMS MADE EASY IN JAVA

SECOND EDITION

本书由曾供职于多家知名IT企业的资深软件架构师撰写，以Java为描述语言，介绍计算机编程中使用的数据结构和算法，覆盖相应竞争性考试的主题，目的不是提供关于数据结构和算法的定理及证明，而是强调问题及其分析，讲解必备知识和解题技巧。书中汇集知名IT企业经典的编程面试题目并给出解题思路，为学生应试和软件开发人员面试提供有益指导。

**本书特点：**

- 所有代码用Java实现。
- 数据结构难点启发思考。
- 为每个问题列举可能的解决办法。
- 基于不同复杂度提供多种巧妙的解决方法。
- 覆盖所有竞争性考试的主题。
- 囊括数据结构和算法的面试问题。
- 可作为大学本科生或硕士研究生课程的预习教材。
- 可为IT顶尖公司（微软、谷歌、亚马逊、雅虎、甲骨文、脸谱、苹果等）的求职者提供指导。




投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机\数据结构与算法

ISBN 978-7-111-53845-5

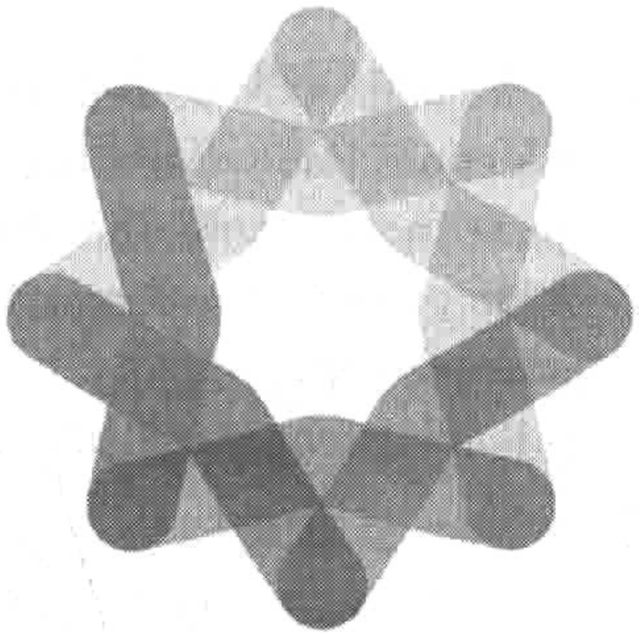
9 787111 538455 >

定价：79.00元

# 数据结构与算法经典问题解析

## Java语言描述

（原书第2版）



DATA STRUCTURES AND
ALGORITHMS MADE EASY IN JAVA

SECOND EDITION

[印] 纳拉辛哈·卡鲁曼希 (Narasimha Karumanchi) 著　骆嘉伟 李晓鸿 肖正 吴帆 等译

机械工业出版社
China Machine Press

**图书在版编目（CIP）数据**

数据结构与算法经典问题解析：Java 语言描述（原书第 2 版）/（印）纳拉辛哈·卡鲁曼希（Narasimha Karumanchi）著；骆嘉伟等译．—北京：机械工业出版社，2016.5
书名原文：Data Structures and Algorithms Made Easy in Java, Second Edition

ISBN 978-7-111-53845-5

I. 数… II. ①纳… ②骆… III. ①数据结构 ②算法分析 ③ JAVA 语言－程序设计
IV. ① TP311.12 ② TP312

中国版本图书馆 CIP 数据核字（2016）第 114674 号

**本书版权登记号：图字：01-2016-0676**

Translation from the English language edition:

Data Structures and Algorithms Made Easy in Java, Second Edition, by Narasimha Karumanchi (ISBN: 9781466304161).

Copyright © 2014 by CareerMonk.com.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanic, including photocopying, recording, or by any information storage retrieval system, without permission of CareerMonk Publications.

Chinese simplified language edition published by China Machine Press.

Copyright © 2016 by China Machine Press.

本书中文简体字版由 CareerMonk Publications 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书是一本数据结构方面的优秀教材，以 Java 为描述语言，介绍了计算机编程中使用的数据结构和算法。本书强调问题及其分析，而非理论阐述，共分为 21 章，讲述了基本概念、递归和回溯、链表、栈、队列、树、优先队列和堆、并查集 DAT、图算法、排序、查找、选择算法（中位数）、符号表、散列、字符串算法、算法设计技术、贪婪算法、分治算法、动态规划算法、复杂度类型等内容。每章首先阐述必要的理论基础，然后给出问题集。全书中大约有 700 个算法问题及相应的解法，对于许多问题，本书提供了多个具有不同复杂度的解决方法。

本书可作为高等院校计算机及其相关专业的数据结构课程的教材或教学参考书，同时也可以作为从事计算机研究与开发的技术人员的参考书，特别是对正在准备面试、参加选拔性考试以及校园面试的读者尤为有用。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：盛思源　　责任校对：殷　虹
印　　刷：北京诚信伟业印刷有限公司　　版　　次：2016 年 6 月第 1 版第 1 次印刷
开　　本：186mm×240mm　1/16　　印　　张：28.5
书　　号：ISBN 978-7-111-53845-5　　定　　价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

**版权所有·侵权必究**
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

# The Translator's Words 译 者 序

数据结构是计算机科学与技术专业非常重要的一门核心基础课，计算机科学各个领域及各种应用软件都要使用相关的数据结构和算法。

作者编著本书的目的是使读者知晓数据结构和算法的设计原理和实现，而并非单纯地讲述定理及证明。为此，本书利用不同的复杂度来改善问题的解。对于许多问题，从穷举解法开始，逐步引入问题的最佳解，并给出算法所需的运行时间和空间。

全书包含 4 个部分，第一部分(第 1～2 章)主要描述抽象数据类型，给出算法的基本概念和复杂度分析与评价方法，并讨论几乎每章都要用到的递归和回溯技术。第二部分(第 3～9 章)介绍基本数据结构，包括链表、栈、队列、树、优先队列、堆、并查集和图，对于每一种数据结构分别采用多个实例进行具体的演示。第三部分(第 10～15 章)介绍数据处理的技术，包括排序、查找、选择、符号表、散列和字符串算法。第四部分(第 16～21 章)重点介绍一些常用的算法设计技术及应用，包括贪婪算法、分治算法、动态规划算法、复杂度类型，并讨论对于面试和考试的一些有用话题。本书强调问题及分析，而不侧重于理论。本书可作为高等院校计算机及其相关专业的数据结构课程的教材或教学参考书，同时也可以作为从事计算机研究与开发的技术人员的参考书，特别是对正在准备面试、参加选拔性考试以及校园面试的读者尤为有用。

本书的前言、第 9～13 章由骆嘉伟翻译，第 2～5 章、第 15 章由李晓鸿翻译，第 16～21 章由肖正翻译，第 1 章、第 6～8 章由吴帆翻译，第 14 章由朱宁波翻译。此外，梁成、夏艳、罗洪、张天伍、杨亦、齐逸也参与了部分翻译工作。

尽管我们从事数据结构和算法的教学和科研工作多年，在翻译过程中本着认真负责、力求精准的精神，但错误难免，希望广大读者批评指正。

译者

2015 年 12 月于长沙

# 前　言 *Preface*

我知道许多读者往往不读前言，但是强烈建议你至少浏览一下本书前言，因为本书前言与众不同。

本书的主要目的不是提供关于数据结构和算法的定理及证明。本书采用的模式是利用不同的复杂度改善问题的解(对于每个问题，你将发现多个具有不同复杂度及降低复杂度的解法)。基本上，这一思路就是列举某个问题的所有可能解。通过这种方式，即使你遇到一个新问题，它也能够向你指明如何思考该问题所有可能的解。本书对于正在准备面试、参加选拔性考试以及校园面试的读者很有帮助。

作为一个求职者，如果你能完整地阅读本书并且很好地领会书中的内容，相信你会从容地面对面试官，这正是本书的目的所在。若作为一个教师来阅读本书，你将能够用简单的方法来提升授课质量，学生也会为选择攻读计算机科学/信息技术学位而感到欣慰。

作为准备参加计算机科学/信息技术专业选拔考试的学生，本书完整而详细地涵盖了所有必需的主题，在撰写本书时，就着眼于帮助正在准备这些考试的学生。

本书对攻读工程学位的学生和研究生都非常有用。在所有的章节中，你会发现本书更强调问题及其分析，而不是理论的阐述。每一章将首先阐述必要的理论基础，然后再给出问题集。书中大约有 700 个算法问题及相应的解。

对于许多问题，本书提供了多个具有不同复杂度的解决方法。我们从蛮力法开始，逐步引入问题的最佳解决方法。对于每一个问题，我们试图知晓算法所需的运行时间和内存空间。

建议读者至少完整地阅读本书一遍以便充分理解所有的主题。在随后的阅读中，你可以直接选择任何一章阅读和参考。即便经过足够的校阅，书中出现小纰漏也在所难免。如果发现了任何此类错误，www. CarrerMonk. com 网站将予以更新，请经常关注本网站以便及时了解任何勘误、新问题和解决方法。此外，请提供宝贵建议至 Info@CarrerMonk. com。

祝愿你一切顺利。我相信你会发现本书很有用。

## 致谢

感谢我的父母，他们为我所做的一切无法衡量，是他们给予的无私的爱、提供的安

定的成长环境和坚持不懈的传统价值观，教会了我赞美和拥抱生活。他们是这世界上最好的父母和榜样，他们使我明白信念、勤奋和决心能够让任何事成为可能！

本书的撰写得到了许多人的帮助，没有他们的帮助本书不可能完成。感谢他们为改进本书终稿所做出的努力。需要说明的是，我已经尽最大努力纠正了审稿人所指出的错误并准确地对各种协议和机制进行了描述。我个人对书中出现的任何其他错误负责。

首先，感谢那些在本书撰写过程中陪我度过难关的人，感谢所有给予我支持的人，感谢所有参与讨论、阅读、编写和提出宝贵意见的人，感谢所有允许我引用他们的评论并协助我编辑、校对和设计本书的人。特别地，我要感谢如下人员：

- Mohan Mullapudi，印度理工学院孟买分校，架构师，dataRPM Pvt. Ltd.
- Navin Kumar Jaiswal，资深咨询师，Juniper Networks Inc.
- Kishore Kumar Jinka，印度理工学院孟买分校
- A. Vamshi Krishna，印度理工学院坎普尔分校，Mentor Graphics Inc.
- Hirak Chatterjee，Yahoo Inc.
- Kondrakunta Murali Krishna，科技学士，技术主管，HCL
- Chaganti Siva Rama Krishna Prasad，创始人，StockMonks Pvt. Ltd.
- Naveen Valsakumar，联合创始人，NotionPress Pvt. Ltd.
- Ramanaiah，讲师，龙树科技学院，MLG

最后，感谢 Guntur Vikas 学院主任 Y. V. Gopala Krishna Murthy 教授、Ayub Khan 教授（ACE 工程学院）、T. R. C. Bose（APTransco 前任主任）、Ch. Venkateswara Rao VNR Vignanajyothi（工程学院，Hyderabad）、Ch. Venkata Narasaiah（IPS）、Yarapathineni Lakshmaiah（Manchikallu，Gurazala），以及所有在本项目期间帮助过我和家人的所有好心人。

——Narasimha Karumanchi
印度理工学院孟买分校理科硕士
CareerMonk. com 创始人

# 目　录 *Contents*


译者序
前言

**第 1 章　绪论** …… 1

1.1　变量 …… 1
1.2　数据类型 …… 1
1.3　数据结构 …… 2
1.4　抽象数据类型 …… 2
1.5　什么是算法 …… 3
1.6　为什么需要算法分析 …… 3
1.7　算法分析的目的 …… 3
1.8　什么是运行时间分析 …… 4
1.9　如何比较算法 …… 4
1.10　什么是增长率 …… 4
1.11　常用的增长率 …… 4
1.12　分析的类型 …… 5
1.13　渐近表示 …… 6
1.14　大 $O$ 表示法 …… 6
1.15　$\Omega$ 表示法 …… 7
1.16　$\Theta$ 表示法 …… 8
1.17　重要说明 …… 9
1.18　为什么称为渐近分析 …… 9
1.19　渐近分析指南 …… 9
1.20　渐近表示法的性质 …… 11
1.21　常用的对数和累加公式 …… 11
1.22　分治法主定理 …… 12
1.23　分治法主定理的相关问题 …… 12
1.24　问题规模减小和递归求解主定理 …… 13
1.25　问题规模减小和递归求解主定理的变型 …… 13
1.26　猜测和确认的方法 …… 14
1.27　平摊分析 …… 15
1.28　算法分析的相关问题 …… 15

**第 2 章　递归和回溯** …… 28

2.1　引言 …… 28
2.2　什么是递归 …… 28
2.3　为什么要用递归 …… 28
2.4　递归函数的格式 …… 28
2.5　递归和内存(可视化) …… 29
2.6　递归与迭代 …… 30
2.7　递归说明 …… 30
2.8　递归算法的经典用例 …… 30
2.9　递归的相关问题 …… 31
2.10　什么是回溯 …… 32
2.11　回溯算法的经典用例 …… 32
2.12　回溯的相关问题 …… 32

**第 3 章　链表** …… 34

3.1　什么是链表 …… 34
3.2　链表抽象数据类型 …… 34
3.3　为什么要用链表 …… 35
3.4　数组概述 …… 35
3.5　链表、数组和动态数组的

比较 …… 36
3.6 单向链表 …… 36
3.7 双向链表 …… 41
3.8 循环链表 …… 46
3.9 一种存储高效的双向链表 …… 51
3.10 松散链表 …… 52
3.11 链表的相关问题 …… 55

**第4章 栈** …… 72

4.1 什么是栈 …… 72
4.2 如何使用栈 …… 72
4.3 栈抽象数据类型 …… 73
4.4 异常 …… 73
4.5 应用 …… 73
4.6 实现 …… 73
4.7 栈的各种实现方法比较 …… 77
4.8 栈的相关问题 …… 78

**第5章 队列** …… 98

5.1 什么是队列 …… 98
5.2 如何使用队列 …… 98
5.3 队列抽象数据类型 …… 99
5.4 异常 …… 99
5.5 应用 …… 99
5.6 实现 …… 99
5.7 队列的相关问题 …… 104

**第6章 树** …… 110

6.1 什么是树 …… 110
6.2 术语 …… 110
6.3 二叉树 …… 111
6.4 二叉树的遍历 …… 114
6.5 通用树(N叉树) …… 135
6.6 线索(无栈或无队列结构)二叉树遍历 …… 141
6.7 表达式树 …… 147
6.8 异或树 …… 149
6.9 二叉搜索树 …… 150
6.10 平衡二叉搜索树 …… 164
6.11 AVL树 …… 165
6.12 树的其他形式 …… 178
6.12.1 红黑树 …… 178
6.12.2 伸展树 …… 179
6.12.3 增强树 …… 179
6.12.4 替罪羊树 …… 179
6.12.5 区间树 …… 180

**第7章 优先队列和堆** …… 181

7.1 什么是优先队列 …… 181
7.2 优先队列 ADT …… 181
7.3 优先队列的应用 …… 182
7.4 优先队列的实现 …… 182
7.5 堆和二叉堆 …… 183
7.6 二叉堆 …… 184
7.7 优先队列(堆)的相关问题 …… 190

**第8章 并查集 ADT** …… 201

8.1 引言 …… 201
8.2 等价关系和等价类 …… 201
8.3 并查集 ADT …… 202
8.4 应用 …… 202
8.5 并查集 ADT 实现中的权衡 …… 202
8.6 快速 UNION 实现(慢 FIND) …… 203
8.7 快速 UNION 实现(快速 FIND) …… 206
8.8 路径压缩 …… 208
8.9 小结 …… 209
8.10 并查集的相关问题 …… 209

**第9章 图算法** …… 211

9.1 引言 …… 211
9.2 术语 …… 211
9.3 图的应用 …… 214
9.4 图的表示 …… 214
9.5 图的遍历 …… 217

9.6 拓扑排序 …… 225
9.7 最短路径算法 …… 226
9.8 最小生成树 …… 231
9.9 图算法的相关问题 …… 235

**第 10 章 排序** …… 256

10.1 什么是排序 …… 256
10.2 为什么需要排序 …… 256
10.3 排序的分类 …… 256
10.4 其他分类方法 …… 257
10.5 冒泡排序 …… 257
10.6 选择排序 …… 258
10.7 插入排序 …… 259
10.8 希尔排序 …… 261
10.9 归并排序 …… 262
10.10 堆排序 …… 264
10.11 快速排序 …… 264
10.12 树排序 …… 266
10.13 排序算法比较 …… 267
10.14 线性排序算法 …… 267
10.15 计数排序 …… 267
10.16 桶排序 …… 268
10.17 基数排序 …… 268
10.18 拓扑排序 …… 269
10.19 外部排序 …… 269
10.20 排序的相关问题 …… 270

**第 11 章 查找** …… 279

11.1 什么是查找 …… 279
11.2 为什么需要查找 …… 279
11.3 查找的类型 …… 279
11.4 符号表和散列 …… 281
11.5 字符串查找算法 …… 281
11.6 查找的相关问题 …… 281

**第 12 章 选择算法(中位数)** …… 304

12.1 什么是选择算法 …… 304
12.2 基于排序的选择算法 …… 304
12.3 基于划分的选择算法 …… 304
12.4 线性选择算法——中位数的中位数算法 …… 305
12.5 按照排序顺序查找 $K$ 个最小元素 …… 305
12.6 选择算法的相关问题 …… 305

**第 13 章 符号表** …… 314

13.1 引言 …… 314
13.2 什么是符号表 …… 314
13.3 符号表的实现 …… 315
13.4 符号表实现方法的比较 …… 315

**第 14 章 散列** …… 317

14.1 什么是散列 …… 317
14.2 为什么用散列 …… 317
14.3 散列表 ADT …… 317
14.4 散列的例子 …… 317
14.5 散列的组成部分 …… 319
14.6 散列表 …… 319
14.7 散列函数 …… 319
14.8 负载因子 …… 320
14.9 冲突 …… 320
14.10 冲突解决技术 …… 320
14.11 分离链接法 …… 320
14.12 开放定址法 …… 321
14.13 冲突解决技术的比较 …… 322
14.14 散列如何达到 $O(1)$ 的时间复杂度 …… 322
14.15 散列技术 …… 323
14.16 不适用散列表的问题 …… 323
14.17 布鲁姆过滤器 …… 323
14.18 散列的相关问题 …… 325

**第 15 章 字符串算法** …… 335

15.1 引言 …… 335
15.2 字符串匹配算法 …… 335
15.3 蛮力法 …… 336

15.4 Robin-Karp 字符串匹配算法 …… 336
15.5 基于有限自动机的字符串匹配算法 …… 337
15.6 KMP 算法 …… 338
15.7 Boyce-Moore 算法 …… 342
15.8 存储字符串的数据结构 …… 342
15.9 字符串的散列表实现 …… 342
15.10 字符串的二叉搜索树实现 … 343
15.11 键树 …… 343
15.12 三叉搜索树 …… 345
15.13 二叉搜索树、键树和三叉搜索树的比较 …… 349
15.14 后缀树 …… 349
15.15 字符串的相关问题 …… 353

**第 16 章 算法设计技术** …… 361

16.1 引言 …… 361
16.2 分类 …… 361
16.3 按实现方法分类 …… 361
16.4 按设计方法分类 …… 362
16.5 其他分类法 …… 363

**第 17 章 贪婪算法** …… 364

17.1 引言 …… 364
17.2 贪婪策略的定义 …… 364
17.3 贪婪算法的要素 …… 364
17.4 贪婪算法的适用范围 …… 365
17.5 贪婪算法的优缺点 …… 365
17.6 贪婪算法的应用 …… 365
17.7 贪婪思想 …… 365
17.8 贪婪算法的相关问题 …… 368

**第 18 章 分治算法** …… 375

18.1 引言 …… 375
18.2 分治策略的定义 …… 375
18.3 分治法的适用范围 …… 375
18.4 分治法的图形化描述 …… 375
18.5 分治思想 …… 376
18.6 主定理 …… 377
18.7 分治法的应用 …… 377
18.8 分治法的相关问题 …… 378

**第 19 章 动态规划算法** …… 390

19.1 引言 …… 390
19.2 动态规划策略的定义 …… 390
19.3 动态规划策略的性质 …… 390
19.4 动态规划的适用范围 …… 390
19.5 动态规划的实现方法 …… 391
19.6 动态规划算法的例子 …… 391
19.7 动态规划思想 …… 391
19.8 动态规划的相关问题 …… 396

**第 20 章 复杂度类型** …… 425

20.1 引言 …… 425
20.2 多项式/指数时间 …… 425
20.3 决策问题的定义 …… 426
20.4 决策过程 …… 426
20.5 复杂度类型的定义 …… 426
20.6 复杂度类型 …… 426
20.7 归约 …… 428
20.8 复杂度类型的相关问题 …… 430

**第 21 章 杂谈** …… 433

21.1 引言 …… 433
21.2 位运算的使用 …… 433
21.2.1 按位与操作 …… 433
21.2.2 按位或操作 …… 434
21.2.3 按位异或操作 …… 434
21.2.4 按位左移操作 …… 434
21.2.5 按位右移操作 …… 434
21.2.6 按位补操作 …… 434
21.2.7 检测第 $K$ 位是否置位 …… 434
21.2.8 第 $K$ 位置位 …… 435
21.2.9 第 $K$ 位清零 …… 435

21.2.10 切换第 $K$ 位 …… 435
21.2.11 切换值为 1 的最右位 …… 435
21.2.12 隔离值为 1 的最右位 …… 435
21.2.13 隔离值为 0 的最右位 …… 435
21.2.14 检查某个数是否是 2 的幂 …… 436
21.2.15 将某个数乘以 2 的幂 …… 436
21.2.16 将某个数除以 2 的幂 …… 436
21.2.17 找到给定操作数的模 …… 436
21.2.18 反转二进制数 …… 436
21.2.19 位值 1 的计数 …… 436
21.2.20 创建末尾位为 0 的掩码 …… 437
21.2.21 交换奇偶位 …… 438
21.2.22 不使用除法来计算平均数 …… 438
21.3 其他编程问题 …… 438

**参考文献** …… 442



# 第1章 Chapter 1
# 绪　论

本章的目的是阐述算法分析的重要性、它们的表示法和关系，并尽可能求解多个问题。首先，让我们重点关注算法的基本要素、分析的重要性，然后再逐步讨论上述提及的其他主题。在完成本章的学习后，能够分析任意给定算法的复杂度(特别是递归函数)。

## 1.1　变量

在开始讨论变量的定义之前，先来看看以前学习过的数学方程式，它与变量是有关联的。许多人从初中阶段就学会了求解许多数学方程式。例如，考虑如下方程式：

$$x^2 + 2y - 2 = 1$$

我们不必关心这个方程式用在什么地方。需要理解的重点是，这个方程式包含一些名称($x$ 和 $y$)，它们能保存值(数据)。即名称($x$ 和 $y$)是用来表示数据的占位符。类似地，在计算机科学中，也需要一些东西来保存数据，变量就是实现这一功能的方式。

## 1.2　数据类型

在上述方程式中，变量 $x$ 和 $y$ 能表示任意值，例如整数(10、20)、实数(0.23、5.5)或仅仅是0和1。为了求解该方程式，需要将它们与其所能表示的数据值的类型关联起来。在计算机科学中，数据类型就用于这一目的。

编程语言中的数据类型是指具有预定义值的一个数据集合。典型的数据类型有：整数、浮点数、字符、字符串等。

计算机内存中全由0和1填充。如果直接用0和1来编码求解问题是非常困难的。为了帮助程序员，编程语言和编译器提供了数据类型。

例如，整数(integer)数据类型占2字节(具体的字节数与编译器有关)，浮点(float)数据类型占4字节等。这就是说，在内存中将2个字节组合起来(16位)可称为整数。类似

地，将 4 个字节组合起来(32 位)可称为浮点数。数据类型可以减少编码的工作量。在顶层，有两种数据类型：

- 系统定义的数据类型(也称为基本数据类型)。
- 用户定义的数据类型。

**1. 系统定义的数据类型(基本数据类型)**

系统定义的数据类型叫作基本数据类型。许多编程语言提供的基本数据类型有：int、float、char、double、bool 等。每一个基本数据类型分配的位数与编程语言、编译器和操作系统有关。对于相同的基本数据类型，不同的编程语言可能使用不同的大小。根据数据类型的大小变化，总的有效数据值(值域)也是变化的。

例如，int 可能占 2 字节或 4 字节。如果占 2 字节(16 位)，则总的取值范围为－32 768～32 767($-2^{15}\sim 2^{15}-1$)；如果占 4 字节(32 位)，那么总的取值范围为－2 147 483 648～2 147 483 648($-2^{31}\sim 2^{31}-1$)。其他数据类型也是这样。

**2. 用户定义的数据类型**

如果系统定义的数据类型不够，那么大多数编程语言允许用户定义自己的数据类型，称为用户定义的数据类型。用户定义的数据类型的经典实例是：C/C++ 中的结构体和 Java 中的类。

例如，下面的代码片段把许多系统定义的数据类型组合在一起，然后用新的名字“newType”将其表示为用户定义的数据类型。这样可以更加灵活和方便地处理计算机内存。

```
public class newType {
    public int data1;
    public int data 2;
    private float data3;
    ...
    private char data;
    // 操作
}
```

## 1.3 数据结构

基于上述讨论，一旦变量中有数据，就需要一种操纵这些数据的机制来求解问题。数据结构(data structure)就是计算机中存储和组织数据的一种特定方式，它将使得数据处理更加有效。一个数据结构就是一种组织和存储数据的特定形式。常用的数据结构包括数组、文件、链表、栈、队列、树和图等。

根据元素的组织方式，数据结构可以分为两种类型：

1) 线性数据结构：可以按线性次序访问元素，但它并不强制将所有元素连续地存储在一起(例如，链表)。例如，链表、栈和队列。

2) 非线性数据结构：这种数据结构的元素是以非线性次序来存储和访问的。例如，树和图。

## 1.4 抽象数据类型

在定义抽象数据类型前，先从不同的视角分析系统定义的数据类型。我们都知道，在默认情况下，所有基本数据类型(int、float 等)都支持基本运算，如加法和减法。系统

实现了这些基本数据类型的基本运算。对于用户自定义的数据类型，也需要定义相应的运算。在实际使用这些运算时需要实现这些运算。即，通常用户定义的数据类型需要与它们的运算一起定义。

为简化求解问题的过程，将数据结构和相关的运算组合起来，将其称为抽象数据类型(ADT)。一个 ADT 包含两个部分：

1) 数据的声明。

2) 运算的声明。

常用 ADT 包括：链表、栈、队列、优先队列、二叉树、字典、并查集(并和查找)、散列表、图和许多其他类型。例如，栈使用后进先出(LIFO)机制将数据存储到数据结构中，最后插入栈的元素第一个被删除。它的常用操作有：创建栈、入栈、出栈、查找栈顶元素、在栈中查找某一个元素等。

当定义 ADT 时，不需要考虑其实现细节。仅当需要使用它们时才考虑具体的实现。不同的 ADT 类型适合不同的应用。有些是用于高度专业化的特定任务。到本书结束时，我们将探讨许多 ADT。读者也能够学会将相关数据结构与需要解决的问题进行关联。

## 1.5 什么是算法

考虑一个煎蛋的问题。为了煎蛋，通常有如下步骤：

1) 准备一个平底锅。

2) 准备油。

a) 有油吗?

i. 如果有，就放油到锅中。

ii. 如果没有，需要去买油吗?

① 如果是，那么就出门买油。

② 如果否，那么就停止煎蛋。

3) 打开炉子……

对于一个给定的问题(煎蛋)，我们所要做的就是通过一步一步的过程来解决它。算法的形式化定义就是：

算法就是用一条接一条的指令来解决给定的问题。

**注意：** 不需要证明算法的每一步。

## 1.6 为什么需要算法分析

从城市 A 到城市 B，可以有许多种方式：可以坐飞机、坐汽车、坐火车，也可以骑自行车。根据可用性和方便性，可以选择最合适的一种方式。类似地，在计算机科学领域中，对于同一个问题，也存在多个有效算法来解决它(例如，排序问题就有许多算法，如插入排序、选择排序、快速排序等其他多种算法)。算法分析能够帮助我们确定在时间和空间开销方面哪一种算法是有效的。

## 1.7 算法分析的目的

算法分析的目标是根据运行时间及其他的一些因素(如内存、开发者的工作量等)来比较算法(或解决方案)的优劣。

## 1.8 什么是运行时间分析

当问题的规模(输入规模)增大时，它研究问题的处理时间是如何增加的。输入规模是指输入元素的个数。根据问题的类型，输入可能有不同的类型。下面是常用的输入类型：

- 数组的大小。
- 多项式的次数。
- 矩阵中元素的个数。
- 用二进制数表示的输入的位数。
- 图中的顶点和边。

## 1.9 如何比较算法

为了比较算法，首先定义几个客观评价指标。

**执行时间** 它不是一个好的评价指标，因为执行时间与特定的计算机有关。

**执行的语句数** 它不是一个好的评价指标，因为执行的语句数与编程语言有关，也与程序员个人的编程风格有关。

**理想的解决方案** 假设用一个函数(如 $f(n)$)来表示一个算法的运行时间，该函数的输入参数就是问题的规模 $n$。然后比较这些不同函数所对应的运行时间。这种比较与机器时间、编程风格等无关。

## 1.10 什么是增长率

所谓增长率就是随着输入规模的增加，算法运行时间增加的速度，它是输入规模的函数。假设你去商店买一辆小车和一辆自行车，这时你遇到了你的朋友，当他问你买什么东西时，通常你会这么回答他，我来买小车。这是因为比起自行车的花费，小车的花费要大得多得多(自行车的花费被小车的花费忽略掉了)。

$$\text{总花费} = \text{小车的花费} + \text{自行车的花费}$$

$$\text{总花费} \approx \text{小车的花费(近似值)}$$

对于上述例子，我们可以把总成本写成小车花费和自行车花费的函数。对于一个给定的函数，我们忽略了相对微不足道的低阶因变量(当输入规模 $n$ 很大时)。例如，一个由 $n^4$、$2n^2$、$100n$ 和 500 相加组成的函数表达式，其函数值近似为 $n^4$。因为 $n^4$ 的增长率最大。

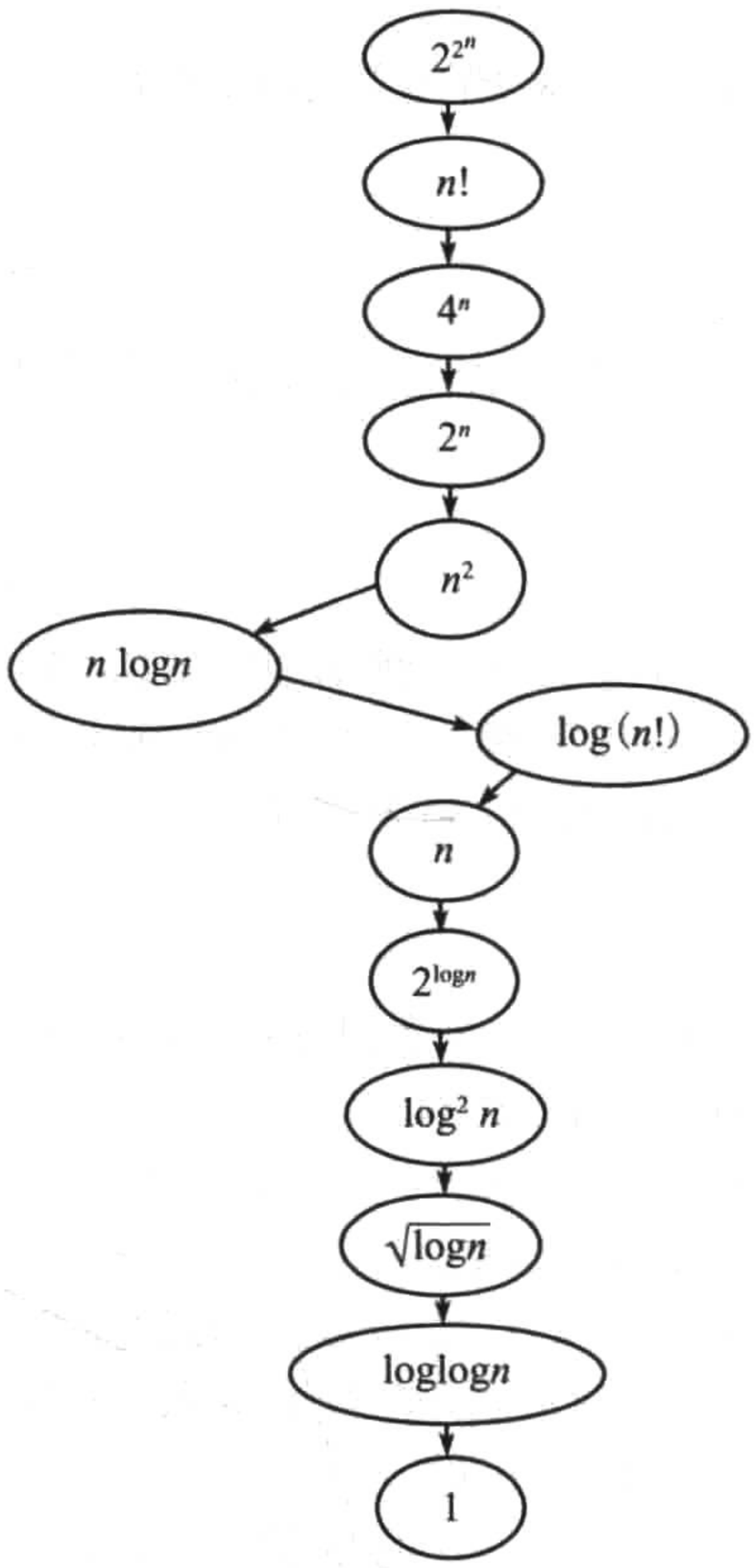
$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

## 1.11 常用的增长率

下面给出的是一些常用的增长率，它们将在之后的章节中进行介绍。

| 时间复杂度 | 名称 | 实例 | 时间复杂度 | 名称 | 实例 |
|---|---|---|---|---|---|
| 1 | 常数 | 在链表的前端增加一个元素 | $n^2$ | 平方 | 求图中两个顶点之间的最短距离 |
| $\log n$ | 对数 | 在有序数组中查找一个元素 | $n^3$ | 立方 | 矩阵乘法 |
| $n$ | 线性 | 在无序数组中查找一个元素 | $2^n$ | 指数 | 汉诺塔问题的求解 |
| $n\log n$ | 线性对数 | 通过分治——归并排序 $n$ 个元素 | | | |

下图显示了不同增长率之间的关系。




## 1.12 分析的类型

为了分析给定的算法，需要知道在什么输入下算法运行时间最短(性能更优)，又是在什么输入下算法运行时间最长。我们知道，算法运行时间可以用表达式来描述。这意味着可以用多个表达式来描述一个算法：其中一个表达式表示该算法所需的最短时间，而另一个表达式则表示该算法所需的最长时间。通常把前一种情况称为算法的最好情况，后者称为算法的最坏情况。为了分析算法，需要某种类型语法，这些语法是渐近分析/表示法的基础。

算法分析有三种类型：

- 最坏情况
  - 定义算法最长运行时间的输入。
  - 这种输入使算法运行最慢。
- 最好情况
  - 定义算法最短运行时间的输入。
  - 这种输入使算法运行最快。
- 平均情况
  - 提供算法运行时间的预测值。

○ 假设输入是随机的，

下界≤平均时间≤上界

对于一个给定的算法，可以用表达式来描述算法的最好、最坏和平均情况。例如，函数 $f(n)$代表给定的算法。

$f(n)=n^2+500$，对应于最坏情况。

$f(n)=n+100n+500$，对应于最好情况。

同样，也可以描述平均情况。这时表达式就表示算法在平均运行时间(或空间)的输入。

## 1.13 渐近表示

有了描述算法的最好、最坏和平均情况的三种表达式后，对每种表达式还需要确定算法的上界和下界。为了表示这些上界和下界，需要某种形式的语法，这也就是接下来要讨论的主题。这里假设给定的算法用函数 $f(n)$来表示。

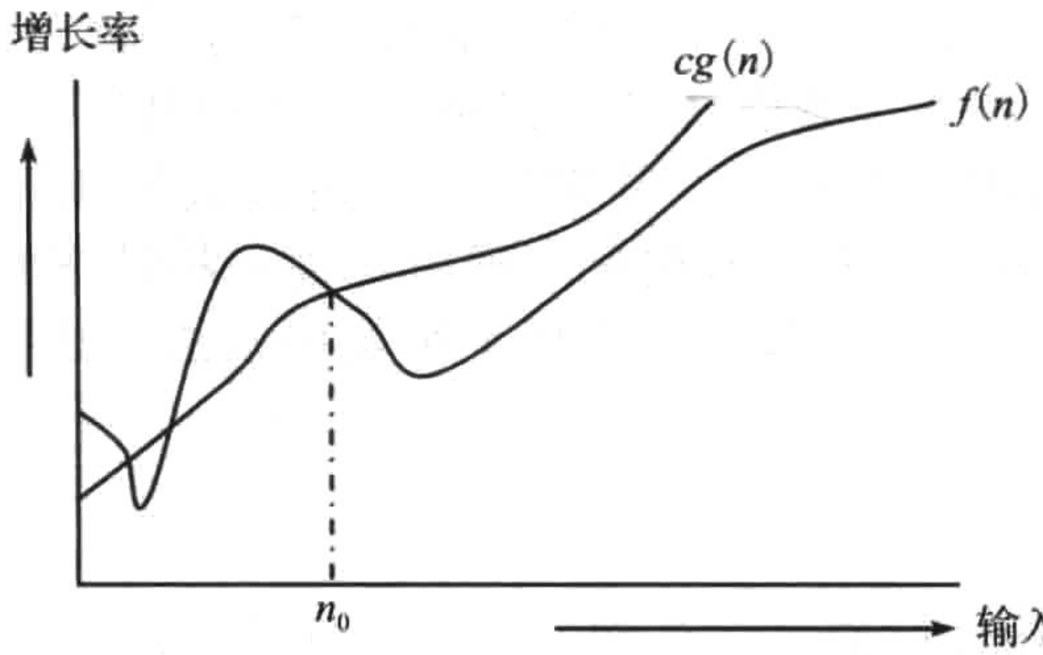
## 1.14 大 O 表示法

大 $O$ 表示法给出了函数 $f(n)$的严格上限。一般来说，它可以表示为 $f(n)=O(g(n))$。这表示当输入规模 $n$ 很大时，$f(n)$的上界是 $g(n)$。

例如，对于给定的算法 $f(n)=n^4+100n^2+10n+50$，那么 $n^4=g(n)$。这意味着随着问题规模 $n$ 的增大，$g(n)$决定了 $f(n)$的最大增长率。

下面更加详细地讨论大 $O$ 表示法的含义。大 $O$ 表示法的定义为，$O(g(n))=\{f(n)$：存在这样的正常数 $c$ 和 $n_0$使得对任意的 $n\geqslant n_0$，$0\leqslant f(n)\leqslant cg(n)$成立$\}$，则 $g(n)$是 $f(n)$的渐近上界。目的是求出大于或等于 $f(n)$的最小的 $g(n)$。

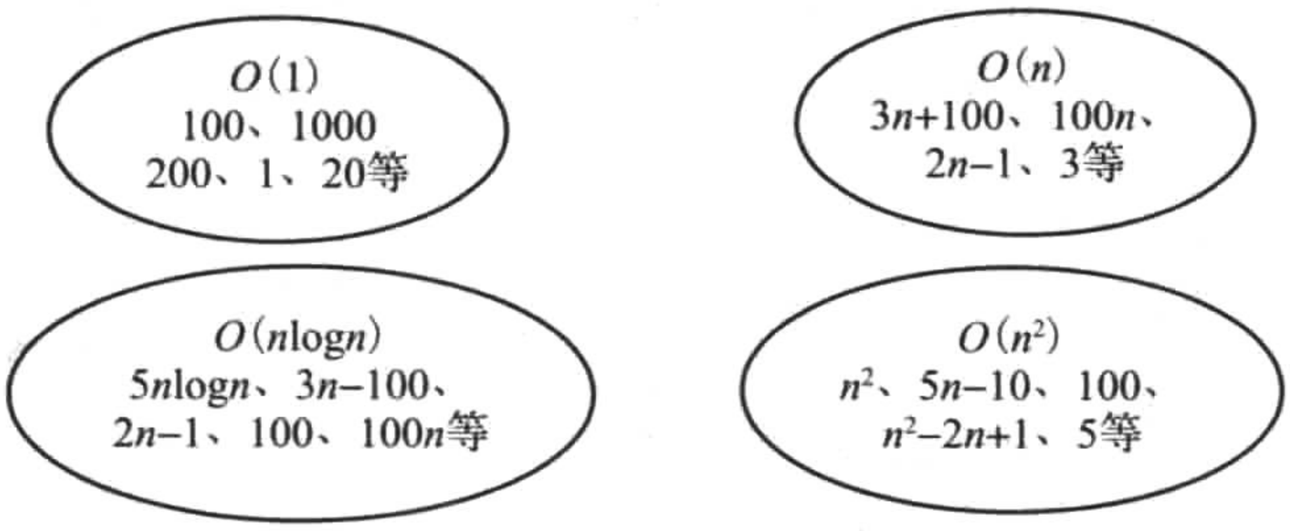
通常我们舍弃 $n$ 的低值部分。也就是说，$n$ 的低值是不重要的。在下图中，$n_0$是一个临界点，在此处需要分析给定算法的增长率的变化规律。而在 $n_0$的下面增长率可能不同。




### 1. 大 O 图示法

$O(g(n))$是指所有与 $g(n)$具有相同增长率或比其增长率小的函数的集合。例如，$O(n^2)$包括 $O(1)$、$O(n)$、$O(n\log n)$等。

**注意：**只在输入规模 $n$ 很大时才分析算法的复杂度，即当 $n<n_0$时不考虑算法运行时间的增长速率。




**2. 大 O 表示法的例子**

**例 1**　求 $f(n)=3n+8$ 的上界。

**解答：** 当 $n\geqslant 8$ 时，$3n+8\leqslant 4n$。

∴当 $c=4$，$n_0=8$ 时，$3n+8=O(n)$。

**例 2**　求 $f(n)=n^2+1$ 的上界。

**解答：** 当 $n\geqslant 1$ 时，$n^2+1\leqslant 2n^2$。

∴当 $c=2$，$n_0=1$ 时，$n^2+1=O(n^2)$。

**例 3**　求 $f(n)=n^4+100n^2+50$ 的上界。

**解答：** 当 $n\geqslant 11$ 时，$n^4+100n^2+50\leqslant 2n^4$。

∴当 $c=2$，$n_0=11$ 时，$n^4+100n^2+50=O(n^4)$。

**例 4**　求 $f(n)=2n^3-2n^2$ 的上界。

**解答：** 当 $n\geqslant 1$ 时，$2n^3-2n^2\leqslant 2n^3$。

∴当 $c=2$，$n_0=1$ 时，$2n^3-2n^2=O(n^3)$。

**例 5**　求 $f(n)=n$ 的上界。

**解答：** 当 $n\geqslant 1$ 时，$n\leqslant n$。

∴当 $c=1$，$n_0=1$ 时，$n=O(n)$。

**例 6**　求 $f(n)=410$ 的上界。

**解答：** 当 $n\geqslant 1$ 时，$410\leqslant 410$。

∴当 $c=1$，$n_0=1$ 时，$410=O(1)$。

**3. 没有唯一性吗**

在证明渐近界限时，$n_0$ 和 $c$ 没有一组确定的解。例如，对于 $100n+5=O(n)$，$n_0$ 和 $c$ 就有多组解。

**解答 1：** 对于任意的 $n\geqslant 5$，$100n+5\leqslant 100n+n=101n\leqslant 101n$，所以 $n_0=5$，$c=101$ 是一组解。

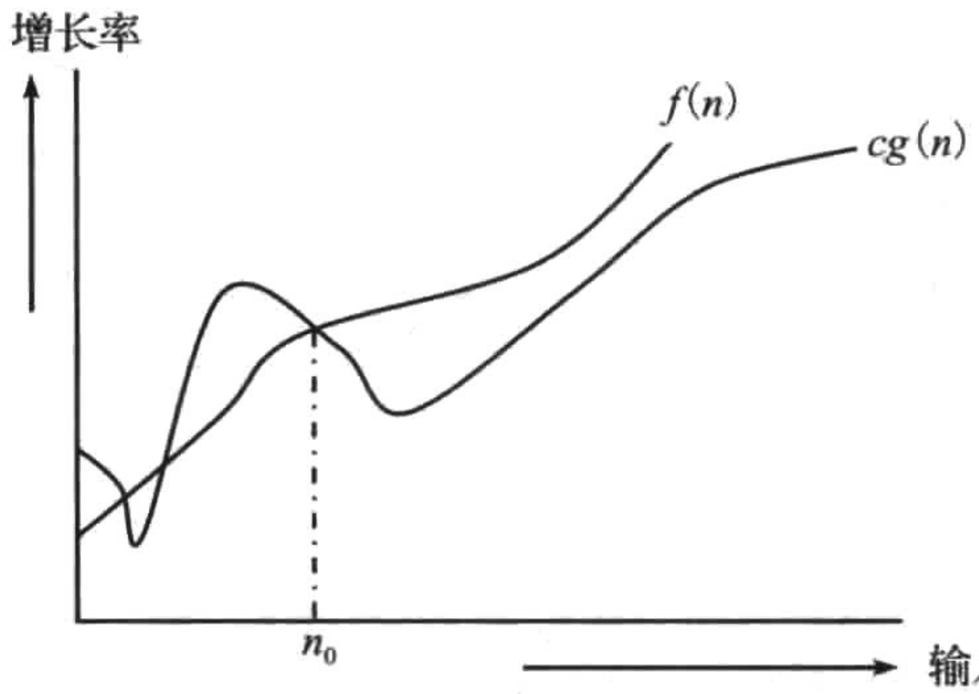
**解答 2：** 对于任意的 $n\geqslant 1$，$100n+5\leqslant 100n+5n=105n\leqslant 105n$，所以 $n_0=1$，$c=105$ 也是一组解。

## 1.15　Ω 表示法

与大 $O$ 表示法类似，$\Omega$ 表示法为给定的算法定义了严格的下界，它表示为 $f(n)=\Omega(g(n))$。也就是说，当输入规模 $n$ 增大时，$f(n)$的严格下界是 $g(n)$。例如，$f(n)=100n^2+10n+50$，$g(n)$是 $\Omega(n^2)$。

$\Omega$ 表示法的定义为：$\Omega(g(n))=\{f(n)$：存在正常数 $n_0$ 和 $c$，使得对于任意的 $n\geqslant n_0$，

$0 \leqslant g(n) \leqslant f(n)$成立}。$g(n)$是 $f(n)$的渐近下界。我们的目的是求出小于或等于增长率 $f(n)$的最大 $g(n)$。




**Ω 的例子**

**例 1** 求 $f(n)=5n^2$ 的下界。

**解答：** 存在常数 $c$，$n_0$ 满足 $0 \leqslant cn^2 \leqslant 5n^2 \Rightarrow cn^2 \leqslant 5n^2 \Rightarrow c=1$ 和 $n_0=1$。

∴当 $c=1$，$n_0=1$ 时，$5n^2=\Omega(n^2)$。

**例 2** 证明 $f(n)=100n+5 \neq \Omega(n^2)$。

**解答：** 存在常数 $c$，$n_0$ 满足 $0 \leqslant cn^2 \leqslant 100n+5$。

对任意 $n \geqslant 1$，有 $100n+5 \leqslant 100n+5n=105n$。

$cn^2 \leqslant 105n \Rightarrow n(cn-105) \leqslant 0$。

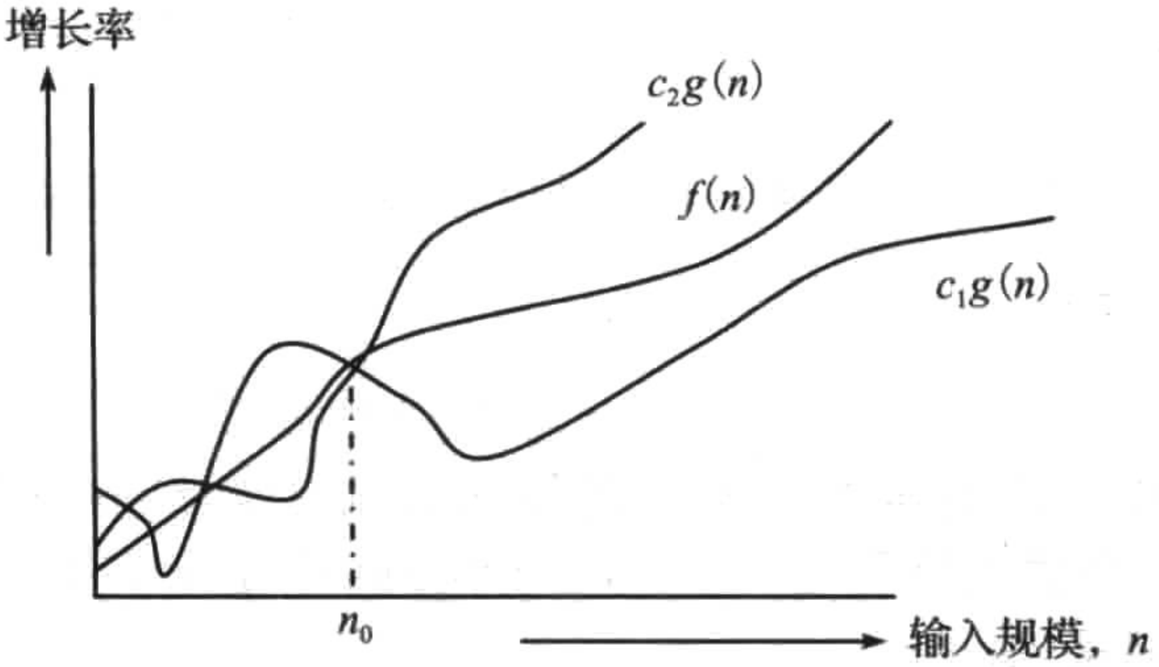
因为 $n>0 \Rightarrow cn-105 \leqslant 0 \Rightarrow n \leqslant 105/c$。

得到矛盾：$n$ 不能小于一个常数。

**例 3** $2n=\Omega(n)$，$n^3=\Omega(n^3)$，$\log n=\Omega(\log n)$。

## 1.16 Θ 表示法

Θ 决定了给定算法的时间增长率的上界和下界是否相同。算法的平均运行时间总是介于上界和下界之间。如果上界($O$)和下界($\Omega$)给出的结果是一样的，那么 Θ 也会得出相同的增长率。例如，假设 $f(n)=10n+n$，那么它的上界 $g(n)$是 $O(n)$，最好情况下算法运行时间的增长率 $g(n)=O(n)$。




在这种情况下，算法运行时间的增长率在最好情况和最坏情况下都是一样的，因此

在平均情况下结果也是相同的。对于一个给定的函数(算法)，如果它的增长率的上界和下界是不同的，那么 $\Theta$ 的增长率可能也会不同。在这种情况下，需要分析所有可能的时间复杂度，然后得出平均情况下的结论(例如，快速排序的平均情况，参见第 10 章)。

$\Theta$ 表示法定义为：$\Theta(g(n))=\{f(n)$：存在正常数 $c_1$、$c_2$ 和 $n_0$，使得对于任意 $n\geqslant n_0$，$0\leqslant c_1 g(n)\leqslant f(n)\leqslant c_2 g(n)$ 成立$\}$。$g(n)$ 是 $f(n)$ 的渐近界。$\Theta(g(n))$ 是所有与 $g(n)$ 增长率相同的函数的集合。

**$\Theta$ 的例子**

**例 1** 求 $f(n)=\frac{n^2}{2}-\frac{n}{2}$ 的 $\Theta$。

**解答：** 当 $n\geqslant 1$ 时，$\frac{n^2}{5}\leqslant\frac{n^2}{2}-\frac{n}{2}\leqslant n^2$。

$\therefore$ 当 $c_1=1/5$，$c_2=1$，$n_0=1$ 时，$\frac{n^2}{2}-\frac{n}{2}=\Theta(n^2)$。

**例 2** 证明 $n\neq\Theta(n^2)$。

**解答：** 若 $c_1 n^2\leqslant n\leqslant c_2 n^2$ 成立$\Rightarrow$仅当 $n\leqslant 1/c_1$——矛盾

$\therefore n\neq\Theta(n^2)$

**例 3** 证明 $6n^3\neq\Theta(n^2)$。

**解答：** 若 $c_1 n^2\leqslant 6n^3\leqslant c_2 n^2$ 成立$\Rightarrow$仅当 $n\leqslant c_2/6$——矛盾

$\therefore 6n^3\neq\Theta(n^2)$

**例 4** 证明 $n\neq\Theta(\log n)$。

**解答：** $c_1\log n\leqslant n\leqslant c_2\log n\Rightarrow$仅当对于任意的 $n\geqslant n_0$，有 $c_2\geqslant\frac{n}{\log n}$——矛盾

$\therefore n\neq\Theta(\log n)$

## 1.17 重要说明

在分析最好、最坏和平均情况时，我们试图给出算法的上界($O$)、下界($\Omega$)和平均运行时间($\Theta$)。从上述例子中可以知道，对于给定算法，得到它的上界($O$)、下界($\Omega$)和平均运行时间($\Theta$)可能并不容易。例如，如果分析一个算法的最好情况，那么应试图给出该算法的上界($O$)、下界($\Omega$)和平均运行时间($\Theta$)。

在本书的后续章节中，通常只关注算法时间复杂度的上界($O$)，因为求下界($\Omega$)没有实际意义。当上界($O$)和下界($\Omega$)相同时，则使用 $\Theta$ 表示法。

## 1.18 为什么称为渐近分析

通过以上对三种表示法(最坏、最好和平均情况)的讨论，可以很容易地理解，对于每一个给定的函数 $f(n)$，试图找到一个函数 $g(n)$，当输入规模 $n$ 增大时，$g(n)$ 近似于 $f(n)$。也就是说，当输入规模 $n$ 增大时，$g(n)$ 的曲线接近于 $f(n)$。在数学上，这样的曲线称为渐近曲线。换言之，$g(n)$ 就是 $f(n)$ 的渐近曲线。因此，算法的复杂度分析又叫作渐近分析。

## 1.19 渐近分析指南

有些通用的规则能帮助我们确定一个算法的运行时间。

1）**循环：**一个循环体的运行时间最多为，循环体内的语句的运行时间(包括循环条件判断)与迭代次数的乘积。

```
// 循环执行n次
for (i=1; i<=n; i++)
    m = m + 2; // 时间常数c
```

总时间$=c\times n=cn=O(n)$。

2）**嵌套循环：**从内到外进行分析。总的运行时间是所有循环规模的乘积。

```
// 外层循环执行n次
for (i=1; i<=n; i++) {
    // 内层循环执行n次
    for (j=1; j<=n; j++)
            k = k+1; // 时间常数
}
```

总时间$=c\times n\times n=cn^2=O(n^2)$。

3）**顺序执行语句：**每条语句的运行时间相加。

```
x = x +1; // 时间常数
// 执行n次
for (i=1; i<=n; i++)
    m = m + 2; // 时间常数
// 外层循环执行n次
for (i=1; i<=n; i++) {
    //内层循环执行n次
    for (j=1; j<=n; j++)
            k = k+1; // 时间常数
}
```

总时间$=c_0+c_1n+c_2n^2=O(n^2)$。

4）if-then-else **条件语句：**最坏情况下的运行时间为，条件判断的时间＋最大值(then部分的语句运行时间或 else 部分的语句运行时间)。

```
//条件：常数
if(length( ) == 0 ) {
    return false; //then部分：常数
}
else { //else部分:(常数+常数) *n
    for (int n = 0; n < length( ); n++) {
            //another if: 常数+常数(无else部分)
            if(!list[n].equals(otherList.list[n]))
            // 常数
            return false;
    }
}
```

总时间$=c_0+c_1+(c_2+c_3)*n=O(n)$。

5）**对数级时间复杂度：**如果算法可以在常数时间把问题的规模按照某个分数(一般是1/2)分解，那么该算法的复杂度为$O(\log n)$。分析下面的例子：

```
for (i=1; i<=n;)
    i = i*2;
```

如果仔细观察可以发现，变量$i$的值每次都是倍增的。初始时$i=1$，下一步$i=1$，在

接下来的各步中 $i=4$，8，…。假设循环执行 $k$ 次，在第 $k$ 次循环时 $2^k=n$，然后循环结束。在等式两边取对数，得到

$\log(2^k)=\log n$
$k\log 2=\log n$
$k=\log n$ //如果假设基数是2

总时间 $=O(\log n)$。

**注意**：类似地，对于下面的例子，最坏情况下的增长率是 $O(\log n)$。同样的讨论对递减序列也成立。

```
for (i=n; i>=1;)
    i = i/2;
```

另一个例子：二分查找(也称为二叉搜索)(在一本 $n$ 页的字典中查找一个单词)。

- 首先查找字典正中间的一页。
- 如果没有找到，确定所要查找的单词在中间页的左边还是右边。
- 对字典的左边或右边重复上述两步，直到找到所要查的单词。

## 1.20 渐近表示法的性质

- 传递性：$f(n)=\Theta(g(n))$且 $g(n)=\Theta(h(n))\Rightarrow f(n)=\Theta(h(n))$。
- 自反性：$f(n)=\Theta(f(n))$，那么 $f(n)=O(f(n))$，$f(n)=\Omega(f(n))$。
- 对称性：$f(n)=\Theta(g(n))$当且仅当 $g(n)=\Theta(f(n))$时。
- 转置对称性：$f(n)=O(g(n))$当且仅当 $g(n)=\Omega(f(n))$时。

## 1.21 常用的对数和累加公式

**对数**

$$\log x^y=y\log x \qquad \log n=\log_{10} n$$
$$\log xy=\log x+\log y \qquad \log^k n=(\log n)^k$$
$$\log\log n=\log(\log n) \qquad \log\frac{x}{y}=\log x-\log y$$
$$a^{\log_b x}=x^{\log_b a} \qquad \log_b x=\frac{\log_a x}{\log_a b}$$

**算术级数**

$$\sum_{k=1}^{n} k=1+2+\cdots+n=\frac{n(n+1)}{2}$$

**几何级数**

$$\sum_{k=0}^{n} x^k=1+x+x^2\cdots+x^n=\frac{x^{n+1}-1}{x-1}(x\neq 1)$$

**调和级数**

$$\sum_{k=1}^{n}\frac{1}{k}=1+\frac{1}{2}+\cdots+\frac{1}{n}\approx\log n$$

**其他重要公式**

$$\sum_{k=1}^{n}\log k\approx n\log n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p \approx \frac{1}{p+1} n^{p+1}$$

## 1.22 分治法主定理

所有的分治算法是(在第 18 章中有详细介绍)把一个问题划分成多个子问题，每个子问题是原问题的一部分，然后执行一些额外的工作来计算最后的答案。例如，归并排序算法(详情请参见第 10 章)计算两个子问题，每个子问题都是原问题规模的一半，然后用 $O(n)$时间的额外工作完成归并。下面给出运行时间计算公式：

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

下面的定理可以用来确定分治算法的运行时间。对于一个给定的程序(算法)，首先尝试找到问题的递归关系。如果问题的递归是下面几种形式之一，那么可以直接得出总运行时间而不必等到把问题完全求解出来。如果问题的递归形式是 $T(n)=aT\left(\frac{n}{b}\right)+\Theta(n^k \log^p n)$，其中 $a \geqslant 1$，$b>1$，$k \geqslant 0$ 且 $p$ 是实数，则：

1) 如果 $a>b^k$，那么 $T(n)=\Theta(n^{\log_b a})$

2) 如果 $a=b^k$

a. 如果 $p>-1$，那么 $T(n)=\Theta(n^{\log_b a} \log^{p+1} n)$

b. 如果 $p=-1$，那么 $T(n)=\Theta(n^{\log_b a} \log\log n)$

c. 如果 $p<-1$，那么 $T(n)=\Theta(n^{\log_b a})$

3) 如果 $a<b^k$

a. 如果 $p \geqslant 0$，那么 $T(n)=\Theta(n^k \log^p n)$

b. 如果 $p<0$，那么 $T(n)=\Theta(n^k)$

## 1.23 分治法主定理的相关问题

对于下面每一个可以用主定理解决的递归形式，都给出了总运行时间 $T(n)$的表达式。否则，就说明该问题不适合用主定理来求解。

**问题 1** $T(n)=3T(n/2)+n^2$

**解答：** $T(n)=3T(n/2)+n^2 \Rightarrow T(n)=\Theta(n^2)$(根据主定理 3. a)

**问题 2** $T(n)=4T(n/2)+n^2$

**解答：** $T(n)=4T(n/2)+n^2 \Rightarrow T(n)=\Theta(n^2 \log n)$(根据主定理 2. a)

**问题 3** $T(n)=T(n/2)+n^2$

**解答：** $T(n)=T(n/2)+n^2 \Rightarrow T(n)=\Theta(n^2)$(根据主定理 3. a)

**问题 4** $T(n)=2^n T(n/2)+n^n$

**解答：** $T(n)=2^n T(n/2)+n^n \Rightarrow$不适用($a$ 不是常数)

**问题 5** $T(n)=16T(n/4)+n$

**解答：** $T(n)=16T(n/4)+n \Rightarrow T(n)=\Theta(n^2)$(根据主定理 1)

**问题 6** $T(n)=2T(n/2)+n\log n$

**解答：** $T(n)=2T(n/2)+n\log n \Rightarrow T(n)=\Theta(n \log^2 n)$(根据主定理 2. a)

**问题 7** $T(n)=2T(n/2)+n/\log n$

**解答：** $T(n)=2T(n/2)+n/\log n \Rightarrow T(n)=\Theta(n\log\log n)$（根据主定理 2. b）

**问题 8** $T(n)=2T(n/4)+n^{0.51}$

**解答：** $T(n)=2T(n/4)+n^{0.51} \Rightarrow T(n)=\Theta(n^{0.51})$（根据主定理 3. b）

**问题 9** $T(n)=0.5T(n/2)+1/n$

**解答：** $T(n)=0.5T(n/2)+1/n \Rightarrow$不适用（$a<1$）

**问题 10** $T(n)=6T(n/3)+n^2\log n$

**解答：** $T(n)=6T(n/3)+n^2\log n \Rightarrow T(n)=\Theta(n^2\log n)$（根据主定理 3. a）

**问题 11** $T(n)=64T(n/8)-n^2\log n$

**解答：** $T(n)=64T(n/8)-n^2\log n \Rightarrow$不适用（函数值非正数）

**问题 12** $T(n)=7T(n/3)+n^2$

**解答：** $T(n)=7T(n/3)+n^2 \Rightarrow T(n)=\Theta(n^2)$（根据主定理 3. a）

**问题 13** $T(n)=4T(n/2)+\log n$

**解答：** $T(n)=4T(n/2)+\log n \Rightarrow T(n)=\Theta(n^2)$（根据主定理 1）

**问题 14** $T(n)=16T(n/4)+n!$

**解答：** $T(n)=16T(n/4)+n! \Rightarrow T(n)=\Theta(n!)$（根据主定理 3. a）

**问题 15** $T(n)=\sqrt{2}T(n/2)+\log n$

**解答：** $T(n)=\sqrt{2}T(n/2)+\log n \Rightarrow T(n)=\Theta(\sqrt{n})$（根据主定理 1）

**问题 16** $T(n)=3T(n/2)+n$

**解答：** $T(n)=3T(n/2)+n \Rightarrow T(n)=\Theta(n^{\log 3})$（根据主定理 1）

**问题 17** $T(n)=3T(n/3)+\sqrt{n}$

**解答：** $T(n)=3T(n/3)+\sqrt{n} \Rightarrow T(n)=\Theta(n)$（根据主定理 1）

**问题 18** $T(n)=4T(n/2)+cn$

**解答：** $T(n)=4T(n/2)+cn \Rightarrow T(n)=\Theta(n^2)$（根据主定理 1）

**问题 19** $T(n)=3T(n/4)+n\log n$

**解答：** $T(n)=3T(n/4)+n\log n \Rightarrow T(n)=\Theta(n\log n)$（根据主定理 3. a）

**问题 20** $T(n)=3T(n/3)+n/2$

**解答：** $T(n)=3T(n/3)+n/2 \Rightarrow T(n)=\Theta(n\log n)$（根据主定理 2. a）

## 1.24 问题规模减小和递归求解主定理

令 $T(n)$为正整数 $n$ 的函数，对于某些常数 $c$，$a>0$，$b>0$，$k\geqslant 0$ 和函数 $f(n)$，$T(n)$ 满足下面的性质：

$$T(n)=\begin{cases}c & n\leqslant 1\\ aT(n-b)+f(n) & n>1\end{cases}$$

如果 $f(n)$的时间复杂度是 $O(n^k)$，则

$$T(n)=\begin{cases}O(n^k) & a<1\\ O(n^{k+1}) & a=1\\ O(n^k a^{\frac{n}{b}}) & a>1\end{cases}$$

## 1.25 问题规模减小和递归求解主定理的变型

方程 $T(n)=T(\alpha n)+T((1-\alpha)n)+\beta n$，其中 $0<\alpha<1$，$\beta>0$，$\alpha$，$\beta$ 均是常数，其复

杂度为 $O(n\log n)$。

## 1.26 猜测和确认的方法

现在让我们来讨论一种可以解决任何递归问题的方法。这种方法的基本思想是

先猜测问题的答案，然后用归纳法证明其正确性。

换言之，它如下求解的问题。如果给定的递归不是主定理中的任何一种形式它应该怎么办？如果先猜测问题的一个解，然后试图用归纳法去证明这个猜测，通常这个猜测要么被证明是对的(这种情况下求解结束)，要么被证明是错的(错误的解可以帮助改善猜测)。

以 $T(n)=\sqrt{n}T(\sqrt{n})+n$ 为例，它不符合分治法主定理提出的几种划分子问题的形式。但是仔细观察递归式可以发现，它与分治法很相似(把问题分成$\sqrt{n}$个子问题，每个子问题的规模是$\sqrt{n}$)。子问题的规模在第一层递归时是 $n$。所以，我们猜测 $T(n)=O(n\log n)$，然后尝试证明该猜测是正确的。

从证明上界 $T(n)\leqslant cn\log n$ 开始：

$$T(n)=\sqrt{n}T(\sqrt{n})+n\leqslant\sqrt{n}c\sqrt{n}\log\sqrt{n}+n=nc\log\sqrt{n}+n=nc\frac{1}{2}\log n+n\leqslant cn\log n$$

最后一个不等式成立的前提条件是 $1\leqslant c\frac{1}{2}\log n$。当 $n$ 足够大时，不管对于取值有多小的任意常数 $c$，这个不等式都是正确的。从上面的证明得出，对上界的猜测是正确的。

现在，证明该递归式的下界。

$$T(n)=\sqrt{n}T(\sqrt{n})+n\geqslant\sqrt{n}k\sqrt{n}\log\sqrt{n}+n=nk\log\sqrt{n}+n=nk\frac{1}{2}\log n+n\geqslant kn\log n$$

最后一个不等式成立的条件是 $1\geqslant k\frac{1}{2}\log n$。当 $n$ 足够大时，对于任意的常数 $k$，这个不等式是不正确的。从以上证明得出，对下界的猜测是错误的。

从以上讨论可以得出，$\Theta(n\log n)$太大了。那么 $\Theta(n)$合适吗？下界很容易直接证明：

$$T(n)=\sqrt{n}T(\sqrt{n})+n\geqslant n$$

现在证明上界也是 $\Theta(n)$。

$$T(n)=\sqrt{n}T(\sqrt{n})+n\leqslant\sqrt{n}c\sqrt{n}+n=nc+n=n(c+1)>cn$$

从以上归纳得出，$\Theta(n)$太小了，而 $\Theta(n\log n)$又太大了。所以，需要一个比 $n$ 大且比 $n\log n$ 小的函数，那么 $n\sqrt{\log n}$合适吗？

现在证明其上界是 $n\sqrt{\log n}$：

$$T(n)=\sqrt{n}T(\sqrt{n})+n\leqslant\sqrt{n}c\sqrt{n}\sqrt{\log\sqrt{n}}+n=nc\frac{1}{\sqrt{2}}\log\sqrt{n}+n\leqslant cn\log\sqrt{n}$$

然后证明其下界是 $n\sqrt{\log n}$：

$$T(n)=\sqrt{n}T(\sqrt{n})+n\geqslant\sqrt{n}k\sqrt{n}\sqrt{\log\sqrt{n}}+n=nk\frac{1}{\sqrt{2}}\log\sqrt{n}+n<kn\log\sqrt{n}$$

最后一步不满足条件，所以 $\Theta(n\sqrt{\log n})$不正确。那么 $n\sim n\log n$ 之间还有什么函数？试试 $n\log\log n$ 是否可行。

证明其上界是 $n\log\log n$：

$$\begin{aligned} T(n) &= \sqrt{n}T(\sqrt{n}) + n \\ &\leqslant \sqrt{n}c\sqrt{n}\log\log\sqrt{n} + n \\ &= nc\log\log n - cn + n \\ &\leqslant cn\log\log n \quad \text{如果 } c \geqslant 1 \end{aligned}$$

证明其下界是 $n\log\log n$：

$$\begin{aligned} T(n) &= \sqrt{n}T(\sqrt{n}) + n \\ &\geqslant \sqrt{n}k\sqrt{n}\log\log\sqrt{n} + n \\ &= nk\log\log n - kn + n \\ &\geqslant kn\log\log n \quad \text{如果 } k \leqslant 1 \end{aligned}$$

从以上证明过程可以得出，当 $c \geqslant 1$ 时，$T(n) \geqslant kn\log\log n$，当 $k \leqslant 1$ 时，$T(n) \leqslant cn\log\log n$。严格地说，上述证明过程仍然缺少对基本情况的证明，但还是可以得出结论 $T(n) = \Theta(n\log\log n)$。

## 1.27 平摊分析

平摊分析是指确定一系列操作的平均运行时间。平摊分析不同于平均情况分析，因为它不对输入数据值的分布做任何假设，而平均情况分析假设输入数据并不是最糟糕的情况(例如，有些排序算法能很好地处理“平均”情况下的输入序列，但对于个别输入序列算法的性能极差)。也就是说，平摊分析是一种对一系列操作而不是对单个操作的最坏情况分析。

在标准最坏情况分析提供了过度悲观的上限的情况下，平摊分析的目的是为了更好地理解某些算法的运行时间。平摊分析通常适用于一系列操作，并且这一系列操作中的绝大部分操作的时间开销小，只有少数操作开销大。如果能够说明开销大的操作特别少，那么可以把它们归咎为开销小的操作，然后仅求开销小操作系列的上限即可。

在平摊分析中，通常的方法是为操作序列中的每个操作赋予一个人为的开销，使得整个操作序列总的人为开销受限于操作序列的实际总开销。这种人为开销称为操作的平摊开销。因此，在分析执行时间时，平摊开销是一种理解总运行时间的正确方法。但注意某些特定的操作仍然会花费很长的运行时间，所以平摊分析不是对操作序列中任意单个操作的运行时间上限的分析。

当操作序列中的一个操作对后一个操作的时间开销有影响时：

- 某个特别操作的时间开销可能很大。
- 但该操作可能将数据结构导向另一个状态，使之后的几个操作时间开销很小。

**例子：**从一个数组中找到第 $k$ 个最小的元素。对于这个问题可以用排序算法来解决。在对给定数组进行排序后，只需要直接返回有序数组中第 $k$ 个位置的元素。执行排序(假设使用基于比较的排序算法)操作的时间开销是 $O(n\log n)$。如果执行 $n$ 次这样的查找，那么每次查找的平均时间开销是 $O(n\log n/n) = O(\log n)$。这清楚地说明，一次排序可以降低后续操作的复杂性。

## 1.28 算法分析的相关问题

**注意：**通过以下问题，试图理解不同时间复杂度($O(n)$、$O(\log n)$、$O(\log\log n)$等)的

例子。

**问题 21** 求解如下递归表达式的时间复杂度。

$$T(n) = \begin{cases} 3T(n-1) & \text{如果 } n > 0 \\ 1 & \text{否则} \end{cases}$$

**解答：** 用代入法求解这个问题。

$$T(n) = 3T(n-1)$$
$$T(n) = 3(3T(n-2)) = 3^2 T(n-2)$$
$$T(n) = 3^2(3T(n-3))$$
$$\vdots$$
$$T(n) = 3^n T(n-n) = 3^n T(0) = 3^n$$

显然，这个函数的时间复杂度是 $O(3^n)$。

**注意：** 对于这个问题也可以用问题规模减小和递归求解主定理来解决。

**问题 22** 求解如下递归表达式的时间复杂度。

$$T(n) = \begin{cases} 2T(n-1)-1 & \text{如果 } n > 0 \\ 1 & \text{否则} \end{cases}$$

**解答：** 用代入法求解这个问题。

$$T(n) = 2T(n-1)-1$$
$$T(n) = 2(2T(n-2)-1)-1 = 2^2 T(n-2)-2-1$$
$$T(n) = 2^2(2T(n-3)-2-1)-1 = 2^3 T(n-4)-2^2-2^1-2^0$$
$$T(n) = 2^n T(n-n)-2^{n-1}-2^{n-2}-2^{n-3}-\cdots-2^2-2^1-2^0$$
$$T(n) = 2^n-2^{n-1}-2^{n-2}-2^{n-3}-\cdots-2^2-2^1-2^0$$
$$T(n) = 2^n-(2^n-1) \quad (\text{注意：} 2^{n-1}+2^{n-2}+\cdots+2^0 = 2^n)$$
$$T(n) = 1$$

所以这个算法的时间复杂度是 $O(1)$。注意，虽然这个递归式的解看起来是指数级的，但得出了一个不同的答案。

**问题 23** 下面这个函数的运行时间是多少？

```
void Function(int n) {
        int i=1, s=1;
        while( s <= n) {
                i++;
                s= s+i;
                System.out.println("*");
        }
}
```

**解答：** 分析下面函数中的注释。

```
void Function (int n) {
        int i=1, s=1;
        // s是以步长i增加而不是以步长1增加
        while( s <= n) {
                i++;
                s= s+i;
                System.out.println("*");
        }
}
```

根据方程式 $s_i = s_{i-1} + i$ 来定义 $s$。每次迭代时，变量 $i$ 的值加 1。在第 $i$ 次迭代时，$s$ 的值是 $i$ 的累加和。如果 $k$ 是函数迭代的总次数，那么当循环终止时应该满足的条件是：

$$1 + 2 + \cdots + k = \frac{k(k+1)}{2} > n \Rightarrow k = O(\sqrt{n})$$

**问题 24** 求下面这个函数的时间复杂度。

```
void Function(int n) {
        int i, count =0;
        for(i=1; i*i<=n; i++)
                count++;
}
```

**解答：**

```
void Function(int n) {
        int i, count =0;
        for(i=1; i*i<=n; i++)
                count++;
}
```

如果 $i^2 \leqslant n \Rightarrow T(n) = O(\sqrt{n})$，则上述函数中的循环将终止。理由与问题 23 相同。

**问题 25** 求下面程序的时间复杂度。

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j + n/2<=n; j= j++)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

**解答：** 分析下面函数中的注释。

```
void function(int n) {
        int i, j, k , count =0;
        // 外层循环执行n/2次
        for(i=n/2; i<=n; i++)
                // 中间循环执行n/2次
                for(j=1; j + n/2<=n; j= j++)
                        // 内层循环执行logn次
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

函数的时间复杂度是 $O(n^2 \log n)$。

**问题 26** 求下面程序的时间复杂度。

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j<=n; j= 2 * j)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

**解答：** 分析下面函数中的注释。

```
void function(int n) {
        int i, j, k , count =0;
        // 外层循环执行n/2次
        for(i=n/2; i<=n; i++)
                // 中层循环执行logn次
                for(j=1; j<=n; j= 2 * j)
                        // 内层循环执行logn次
                        for(k=1; k<=n; k= k*2)
                                count++;
}
```

函数的时间复杂度是 $O(n\log^2 n)$。

**问题 27** 求下面程序的时间复杂度。

```
function( int n ) {
        if(n == 1) return;
        for(int i = 1 ; i <= n ; i + + ) {
                for(int j= 1 ; j <= n ; j + + ) {
                        System.out.println("*");
                        break;
                }
        }
}
```

**解答：** 分析下面函数中的注释。

```
function( int n ) {
        // 常数时间
        if( n == 1 ) return;
        // 外层函数执行n次
        for(int i = 1 ; i <= n ; i + + ) {
                // 内层循环只执行了一次，由于break语句
                for(int j= 1 ; j <= n ; j + + )        {
                        System.out.println("*");
                        break;
                }
        }
}
```

函数的时间复杂度为 $O(n)$。尽管内层循环的迭代上界是 $n$，但由于 break 语句使得内层的循环只执行 1 次。

**问题 28** 下面给出了一个时间复杂度为 $T(n)$的递归函数。用迭代方法证明 $T(n)=\Theta(n^3)$。

```
function( int n ) {
        if( n == 1 ) return;
        for(int i = 1 ; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j + + )
                        System.out.println("*") ;
        function( n -3 );
}
```

**解答：** 分析下面函数中的注释。

```
function (int n) {
        // 常数时间
        if( n == 1 ) return;
        // 外循环执行时间
        for(int i = 1 ; i <= n ; i + + )
                // 内层函数执行n次
                for(int j = 1 ; j <= n ; j + + )
                        // 常数时间
                        System.out.println("*") ;
        function( n-3 );
}
```

显然代码的递归表达式为 $T(n)=T(n-3)+cn^2$，其中常数 $c>0$。输出语句执行 $n^2$ 次，递归调用自身的输入规模为 $n-3$。使用迭代方法，可求得 $T(n)=T(n-3)+cn^2$。根据问题规模减小和递归求解主定理，求得 $T(n)=\Theta(n^3)$。

**问题 29** 确定递归关系 $T(n)=T\left(\frac{n}{2}\right)+n\log n$ 的渐近时间复杂度 $\Theta$。

**解答：** 利用分治法主定理，求得 $O(n\log^2 n)$。

**问题 30** 确定递归关系 $T(n)=T\left(\frac{n}{2}\right)+T\left(\frac{n}{4}\right)+T\left(\frac{n}{8}\right)+n$ 的渐近时间复杂度 $\Theta$。

**解答：** 代入递归公式，求得：

$$T(n)\leqslant c_1 * \frac{n}{2}+c_2 * \frac{n}{4}+c_3 * \frac{n}{8}+cn\leqslant k * n, \quad \text{其中 } k \text{ 为常数}$$

**问题 31** 确定递归关系 $T(n)=T(\lceil n/2 \rceil)+7$ 的渐近时间复杂度 $\Theta$。

**解答：** 利用分治法主定理，求得 $\Theta(\log n)$。

**问题 32** 证明以下代码的运行时间是 $\Omega(\log n)$。

```
Read(int n) {
        int k = 1;
        while( k < n )
                k = 3k;
}
```

**解答：** 当 $k\geqslant n$ 时，while 循环将退出。在每次迭代中，$k$ 的值变为原来的 3 倍。假设迭代次数为 $i$，那么在 $i$ 次迭代后 $k$ 的值为 $3i$。循环终止条件是 $3^i\geqslant n\leftrightarrow i\geqslant\log_3 n$，所以 $i=\Omega(\log n)$。

**问题 33** 确定如下递归关系的渐近时间复杂度。

$$T(n)=\begin{cases}1 & n=1\\ T(n-1)+n(n-1) & n\geqslant 2\end{cases}$$

**解答：** 迭代序列如下：

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

$$\cdots$$

$$T(n)=T(1)+\sum_{i=1}^{n} i(i-1)$$

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

$$T(n)=1+\frac{n(n+1)(2n+1)}{6}-\frac{n(n+1)}{2}$$

$$T(n)=\Theta(n^3)$$

**注意：**这个问题可以用分治法主定理来求解。

**问题 34** 分析下列程序的运行时间。

```
Fib[n]
if(n==0) then return 0
else if(n==1) then return 1
else return Fib[n-1]+Fib[n-2]
```

**解答：**该程序运行时间的递归关系是

$$T(n)=T(n-1)+T(n-2)+c$$

注意，$T(n)$有两个递归调用，可表示为一棵二叉树。每一次递归调用，输入规模分别减少 1 和 2，所以递归树的深度是 $O(n)$。由于这是一棵满二叉树，所以深度为 $n$ 的叶子节点的个数是 $2^n$，每个叶子节点花费 $O(1)$的运行时间。所以最终的运行时间是 $O(2^n)$。

**问题 35** 如下程序的运行时间是多少？

```
function(n) {
        for(int i = 1 ; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j+ = i )
                        System.out.println("*") ;
}
```

**解答：**分析下面函数中的注释。

```
function (n) {
        // 该层循环执行n次
        for(int i = 1 ; i <= n ; i + + )
                // 这层循环执行n/i次，j每次增加i
                for(int j = 1 ; j <= n ; j+ = i )
                        System.out.println("*") ;
}
```

在上面的程序中，对于每一个 $i$，内层循环执行 $n/i$ 次。运行时间是 $n\times\left(\sum_{i=1}^{n}n/i\right)=O(n\log n)$。

**问题 36** 求 $\sum_{i=1}^{n}\log i$ 的时间复杂度。

**解答：**利用对数性质 $\log xy=\log x+\log y$，可以得出原问题等价于

$$\sum_{i=1}^{n}\log i=\log 1+\log 2+\cdots+\log n=\log(1\times 2\times\cdots\times n)=\log(n!)\leqslant\log(n^n)\leqslant n\log n$$

所以时间复杂度为 $O(n\log n)$。

**问题 37** 以下递归程序的运行时间是多少(输入规模为 $n$)？首先写出递归公式，然后求它的时间复杂度。

```
function(int n) {
        if(n <= 1) return ;
        for (int i=1 ; i <= 3; i++ )
                f(⌈n/3⌉);
}
```

**解答：**分析下面函数中的注释。

```
function (int n) {
        // 常数时间
        if(n <= 1) return;
        // 这个循环执行递归循环，值为n/3
        for (int i=1 ; i <= 3; i++ )
                f(⌈n/3⌉);
}
```

可以假设对于任意整数 $k \geqslant 1$，渐近分析 $k = \lceil k \rceil$。这段代码的递归关系式为 $T(n) = 3T\left(\frac{n}{3}\right) + \Theta(1)$。利用主定理，可求得 $T(n) = \Theta(n)$。

**问题 38** 以下递归程序的运行时间是多少(输入规模为 $n$)？首先写出递推公式，然后用归纳法进行求解。

```
function(int n) {
        if(n <= 1) return;
        for (int i=1 ; i <= 3 ; i++ )
                function (n - 1).
}
```

**解答：**分析下面函数中的注释。

```
function (int n) {
        // 常数时间
        if(n <= 1) return;
        // 这个循环执行三次值为n-1的递归调用
        for (int i=1 ; i <= 3 ; i++ )
                function (n - 1).
}
```

if 语句需要常数级($O(1)$)运行时间。使用 for 循环，可以忽略循环的开销，只考虑函数的三次递归调用。这意味着递归的时间复杂度为：

$$T(n) = \begin{cases} c & n \leqslant 1 \\ = c + 3T(n-1) & n > 1 \end{cases}$$

利用问题规模减小和递归求解主定理，可得 $T(n) = \Theta(3^n)$。

**问题 39** 求出以下函数 $f$ 的运行时间的递归公式 $T(n)$。当输入规模为 $n$ 时，程序的运行时间是多少？

```
function (int n) {
        if(n <= 1)  return;
        for(int i = 1; i < n; i + +)
                System.out.println("*");
        function ( 0.8n ) ;
}
```

**解答：**分析下面函数中的注释。

```
function (int n) {
        // 常数时间
        if(n <= 1)  return;
        // 这个循环执行n次常数时间的循环
        for(int i = 1; i < n; i + +)
                System.out.println("*");
```

```
        // 执行递归调用，值为0.8n
        function ( 0.8n ) ;
}
```

代码运行时间的递归公式为 $T(n)=T(0.8n)+O(n)=T\left(\frac{4}{5}n\right)+O(n)=\frac{4}{5}T(n)+O(n)$。利用主定理，可得 $T(n)=O(n)$。

**问题 40** 求递归关系 $T(n)=2T(\sqrt{n})+\log n$ 的时间复杂度。

**解答：** 由于给定的递归公式不符合主定理的形式要求，所以先试图将公式转换为主定理格式。假设 $n=2^m$。等式两边取对数，得 $\log n=m\log 2\Rightarrow m=\log n$。现在，给定的公式可转换为

$$T(n)=T(2^m)=2T(\sqrt{2^m})+m=2T(2^{\frac{m}{2}})+m$$

为简单起见，假设 $S(m)=T(2^m)\Rightarrow S\left(\frac{m}{2}\right)=T(2^{\frac{m}{2}})\Rightarrow S(m)=2S\left(\frac{m}{2}\right)+m$。根据主定理，可得 $S(m)=O(m\log m)$。如果把 $m=\log n$ 代回等式，可求得 $T(n)=S(\log n)=O(\log n\log\log n)$。

**问题 41** 求递归关系 $T(n)=T(\sqrt{n})+1$ 的时间复杂度。

**解答：** 利用问题 40 的求解思路，可求得 $S(m)=S\left(\frac{m}{2}\right)+1$。根据主定理，可得 $S(m)=O(\log m)$。把 $m=\log n$ 代回等式，可求得 $T(n)=S(\log n)=O(\log\log n)$。

**问题 42** 求递归关系 $T(n)=2T(\sqrt{n})+1$ 的时间复杂度。

**解答：** 采用问题 40 的求解思路，可得 $S(m)=2S\left(\frac{m}{2}\right)+1$。根据主定理，可得 $S(m)=O(m^{\log_2{}^2})=O(m)$。把 $m=\log n$ 代回等式，可求得 $T(n)=O(\log n)$。

**问题 43** 求下列函数的时间复杂度。

```
int Function (int n) {
        if(n <= 2) return 1;
        else
                return (Function (floor(sqrt(n))) + 1);
}
```

**解答：** 分析下面函数中的注释。

```
int Function (int n) {
        if(n <= 2) return 1;        // 常数时间
        else
                // 执行√n+1次
                return (Function (floor(sqrt(n))) + 1);
}
```

对于上述函数，可以给出递归公式：$T(n)=T(\sqrt{n})+1$。这与问题 41 相同。

**问题 44** 分析下列递归伪代码的运行时间，$n$ 为函数的输入。

```
void function(int n) {
        if( n < 2 ) return;
        else      counter = 0;
        for i = 1 to 8 do
                function (n/2);
```

```
        for i =1 to n^3 do
                counter = counter + 1;
}
```

**解答：** 分析下面伪代码中的注释，令函数 function($n$)的运行时间为 $T(n)$。

```
void function(int n) {
        if( n < 2 ) return;          // 常数时间
        else     counter = 0;
        // 这个循环执行8次，每次调用函数，参数n减半
        for i = 1 to 8 do
                function (n/2);
        // 这个循环执行n^3次，每次都是常数时间
        for i =1 to n^3 do
                counter = counter + 1;
}
```

因此，$T(n)$可定义为：

$$T(n) = \begin{cases} 1 & \text{当 } n < 2 \\ 8T(n/2) + n^3 + 1 & \text{其他} \end{cases}$$

根据主定理，可求得 $T(n)=\Theta(n^{\log_2 8}\log n)=\Theta(n^3\log n)$。

**问题 45** 分析下列伪代码的时间复杂度。

```
temp  = 1
repeat
        for i = 1 to n
                temp =  temp + 1;
        n  = n/2;
until n <= 1
```

**解答：** 分析下面伪代码中的注释。

```
temp  = 1        // 常数时间
repeat
        // 这个循环执行n次
        for i = 1 to n
                temp =  temp + 1;
        // 递归调用，值为 n/2
        n  = n/2;
until n <= 1
```

该函数的递归关系为 $T(n)=T(n/2)+n$。根据主定理，可求得 $T(n)=O(n)$。

**问题 46** 分析下列程序的运行时间。

```
function(int n) {
            for(int i = 1 ; i <= n ; i + + )
                    for(int j = 1 ; j <= n ; j * = 2 )
                            System.out.println("*");
}
```

**解答：** 分析下面函数中的注释。

```
function(int n) {
        // 这个循环执行n次
        for(int i = 1 ; i <= n ; i + + )
                // 这个循环执行logn次
```

```
                for(int j = 1 ; j <= n ; j * = 2 )
                        System.out.println("*");
}
```

因此时间复杂度为 $O(n\log n)$。

**问题 47** 分析下列程序的运行时间。

```
function(int n) {
        for(int i = 1 ; i <= n/3 ; i + + )
                for(int j = 1 ; j <= n ; j += 4 )
                        System.out.println("*");
}
```

**解答：** 分析下面函数中的注释。

```
function(int n) {
        // 这个循环执行n/3次
        for(int i = 1 ; i <= n/3 ; i + + )
                // 这个循环n/4次
                for(int j = 1 ; j <= n ; j += 4)
                        System.out.println("*");
}
```

因此时间复杂度为 $O(n^2)$。

**问题 48** 分析下面程序的时间复杂度。

```
void function(int n) {
    if(n <=  1) return;
    if(n > 1) {
            System.out.println("*");
            function(n/2);
            function(n/2);
        }
}
```

**解答：** 分析下面函数中的注释。

```
void function(int n) {
        if(n <= 1) return;                          // 常数时间
        if(n > 1) {
                System.out.println("*");             // 常数时间
                // 递归调用，参数值为n/2
                function( n/2 );
                // 递归调用，参数值为n/2
                function( n/2 );
        }
}
```

函数的递归公式为 $T(n)=2T\left(\frac{n}{2}\right)+1$。根据主定理，可求得 $T(n)=O(n)$。

**问题 49** 分析下面程序的时间复杂度。

```
function(int n) {
        int i=1;
        while (i < n) {
                int j=n;
                while(j > 0)
                        j = j/2;
```

```
                i=2*i;
        } // i
}
```

**解答：**

```
function(int n) {
        int i=1;
        while (i < n) {
                int j=n;
                while(j > 0)
                        j = j/2;   //logn代码
                i=2*i; //logn次
        } // i
}
```

时间复杂度为 $O(\log n * \log n)=O(\log^2 n)$。

**问题 50**　求 $\sum_{1\leqslant k\leqslant n} O(n)$ 的复杂度，$O(n)$表示阶数 $n$ 是：

(a) $O(n)$　　(b) $O(n^2)$　　(c) $O(n^3)$　　(d) $O(3n^2)$

(e) $O(1.5n^2)$

**解答：**(b) $\sum_{1\leqslant k\leqslant n} O(n) = O(n) \sum_{1\leqslant k\leqslant n} 1 = O(n^2)$。

**问题 51**　下列 3 个选项中哪些是正确的？

(Ⅰ) $(n+k)^m=\Theta(n^m)$，其中 $k$ 和 $m$ 是常数

(Ⅱ) $2^{n+1}=O(2^n)$

(Ⅲ) $2^{2n+1}=O(2^n)$

(a)(Ⅰ)和(Ⅱ)　　(b)(Ⅰ)和(Ⅲ)

(c)(Ⅱ)和(Ⅲ)　　(d)(Ⅰ)、(Ⅱ)和(Ⅲ)

**解答：**(a)(Ⅰ)$(n+k)^m=n^k+c_1 * n^{k-1}+\cdots+k^m=\Theta(n^k)$和(Ⅱ)$2^{n+1}=2 * 2^n=O(2^n)$。

**问题 52**　考虑以下 3 个函数：

$f(n)=2^n$　　$g(n)=n!$　　$h(n)=n^{\log n}$

关于 3 个函数 $f(n)$、$g(n)$和 $h(n)$的渐近时间复杂度分析，下面哪一个是正确的？

(a) $f(n)=O(g(n))$；$g(n)=O(h(n))$

(b) $f(n)=\Omega(g(n))$；$g(n)=O(h(n))$

(c) $g(n)=O(f(n))$；$h(n)=O(f(n))$

(d) $h(n)=O(f(n))$；$g(n)=\Omega(f(n))$

**解答：**(d)根据增长率：$h(n)<f(n)<g(n)$($g(n)$渐近大于 $f(n)$，$f(n)$渐近大于 $h(n)$)。通过取对数的方法可以很容易看到，上面 3 个函数增长率的顺序：$\log n \log n<n<\log(n!)$。注意，$\log(n!)=O(n\log n)$。

**问题 53**　分析下列 C 代码段：

```
int j=1, n;
while (j <=n)
        j = j*2;
```

对于任意 $n>0$，循环执行中比较的次数是：

(a) ceil($\log_2 n$)+1　　(b) $n$

(c) ceil($\log_2 n$)　　(d) floor($\log_2 n$)+1

**解答：**(a)假设循环执行了 $k$ 次。第 $k$ 次循环执行后，$j=2^k$。等式两边取对数，得到 $k=\log_2 n$。因为要多做一次比较才能退出循环，所以答案是 ceil($\log_2 n$)+1。

**问题 54** 分析下面的 C 代码段。程序的输入为 $n$，$T(n)$表示 for 循环执行的次数，下面哪一个选项是正确的？

```
int IsPrime(int n){
  for(int i=2;i<=sqrt(n);i++)
   if(n%i == 0)
     {printf("Not Prime\n"); return 0;}
  return 1;
}
```

(a) $T(n)=O(\sqrt{n})$和 $T(n)=\Omega(\sqrt{n})$　　(b) $T(n)=O(\sqrt{n})$和 $T(n)=\Omega(1)$

(c) $T(n)=O(n)$和 $T(n)=\Omega(\sqrt{n})$　　(d) 以上都不是

**解答：**(b)大 $O$ 表示法描述了算法的渐近上界，大 $\Omega$ 表示法描述了算法的渐近下界。代码中 for 循环最多执行$\sqrt{n}$次，最少执行 1 次。所以 $T(n)=O(\sqrt{n})$，$T(n)=\Omega(1)$。

**问题 55** 在下列 C 代码段中，令 $n\geqslant m$。该函数递归调用了多少次？

```
int gcd(n,m){
        if (n%m ==0) return m;
        n = n%m;
        return gcd(m,n);
}
```

(a) $\Theta(\log_2 n)$　　(b) $\Omega(n)$

(c) $\Theta(\log_2 \log_2 n)$　　(d) $\Theta(n)$

**解答：**没有选项是正确的。大 $O$ 表示法描述了算法的渐近上界，大 $\Omega$ 表示法描述了算法的渐近下界。当 $m=2$，$n=2^i$ 时，运行时间是 $O(1)$，这与所有选项矛盾。

**问题 56** 假定 $T(n)=2T(n/2)+n$，$T(0)=T(1)=1$，下列哪一项是错误的？

(a) $T(n)=O(n^2)$　　(b) $T(n)=\Theta(n\log n)$

(c) $T(n)=\Omega(n^2)$　　(d) $T(n)=O(n\log n)$

**解答：**(c)大 $O$ 表示法描述了算法的渐近上界，大 $\Omega$ 表示法描述了算法的渐近下界。根据主定理，可求得 $T(n)=\Theta(n\log n)$。也就是说，渐近上界与渐近下界相同，即 $O(n\log n)$和$\Omega(n\log n)$正确。所以选项(c)是错误的。

**问题 57** 分析下面程序的时间复杂度。

```
function(int n) {
   for (int i = 0; i<n; i++)
      for(int j=i; j<i*i; j++)
         if (j %i == 0){
            for (int k = 0; k < j; k++)
               printf("*");
         }
}
```

**解答：**

```
function(int n) {
   for (int i = 0; i<n; i++)                    // 执行n次
      for(int j=i; j<i*i; j++)                  // 执行 n*n次
            if (j %i == 0){
```

```
            for (int k = 0; k < j; k++)    // 执行j=n*n次
                printf("*");
        }
}
```

所以时间复杂度为 $O(n^5)$。

**问题 58**　给出计算 $9^n$ 的算法，并分析算法的时间复杂度。

**解答：**初始值＝1，每次循环执行乘以 9 的运算，执行 $n$ 次。

时间复杂度：一共有$(n-1)$次乘法，每次运算花费的时间为常数级，所以算法的时间复杂度为 $\Theta(n)$。

**问题 59**　对于问题 58，还可以降低其时间复杂度吗？

**解答：**参见第 18 章。

**问题 60**　分析下面程序的时间复杂度。

```
function(int n) {
    int sum = 0;
    for (int i = 0; i<n; i++)
        if (i>j)
            sum = sum +1;
        else {
            for (int k = 0; k < n; k++)
                sum = sum -1;
        }
    }
}
```

**解答：**分析最差情况。

```
function(int n) {
    int sum = 0;
    for (int i = 0; i<n; i++)                    // 执行n次
        if (i>j)
            sum = sum +1;                        // 执行n次
        else {
            for (int k = 0; k < n; k++)          // 执行n次
                sum = sum -1;
        }
    }
}
```

时间复杂度为 $O(n^2)$。



Chapter 2 第2章

# 递归和回溯

## 2.1 引言

本章将探讨一个重要的内容“递归”。本书中几乎每章都要用到递归，同时还介绍一个与之相关的概念“回溯”。

## 2.2 什么是递归

任何调用自身的函数称为递归。用递归方法求解问题，要点在于递归函数调用自身去解决一个规模比原始问题小一些的问题。这个过程称为递归步骤。递归步骤会导致更多的递归调用。因此，保证递归过程能够终止是很重要的。每次函数都会用比原问题规模更小的问题来调用自身。问题随着规模不断变小必须能最终收敛到基本情形。

## 2.3 为什么要用递归

递归是从数学领域借鉴过来的一种有用的技术。递归代码通常比迭代代码更加简洁易懂。一般来说，在编译或解释时，循环会转化为递归函数。当任务能够被相似的子任务定义时，采用递归处理十分有效。例如，排序、搜索和遍历等问题往往有简洁的递归解决方案。

## 2.4 递归函数的格式

递归函数在执行一个任务时，需要调用函数自身来完成一些子任务。在某些时候，函数不需要继续调用函数自身就可以完成当前子任务。函数不再递归的情况称作基本情形(base case，也称为基本情况)。而函数调用自身来执行子任务的情况就称作递归情形(recursive case)。可以用如下的形式来描述所有的递归函数：

```
if（判断是否为基本情形）
        return 该基本情形时函数的值
else if（判断是否为另一种基本情形）
        return 该基本情形时函数的值
// 递归情形
else    return（执行某些工作并递归调用）
```

以阶乘函数为例。$n!$ 等于 $n\sim1$ 之间所有整数的乘积，其递归定义如下：

$$n! = 1 \qquad n = 0$$

$$n! = n*(n-1)! \qquad n > 0$$

该定义很容易将阶乘函数转换为一个递归实现过程。这里问题是确定 $n!$ 的值，子问题是确定$(n-1)!$ 的值。在递归情形下，当 $n$ 大于 1 时，函数调用自身来确定$(n-1)!$ 的值，然后再乘以 $n$。在基本情形下，当 $n$ 等于 0 或 1 时，函数只返回 1。阶乘函数的递归伪代码如下：

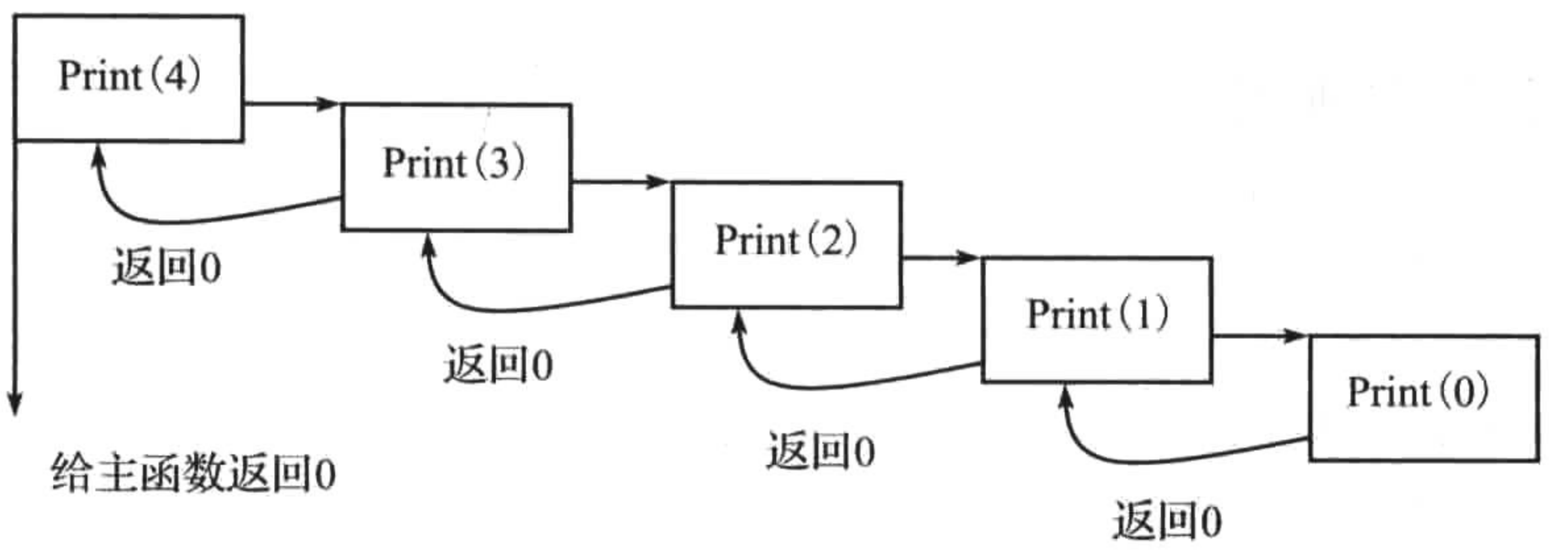
```
// 计算一个正整数的阶乘
int Fact(int n) {
        // 基本情形：当参数为0或1时，返回1
        if(n == 1)
                return 1;
        else if(n == 0)
                return 1;
        // 递归情形：返回 n*(n-1)!
        else
            return n*Fact(n-1);
}
```

## 2.5　递归和内存(可视化)

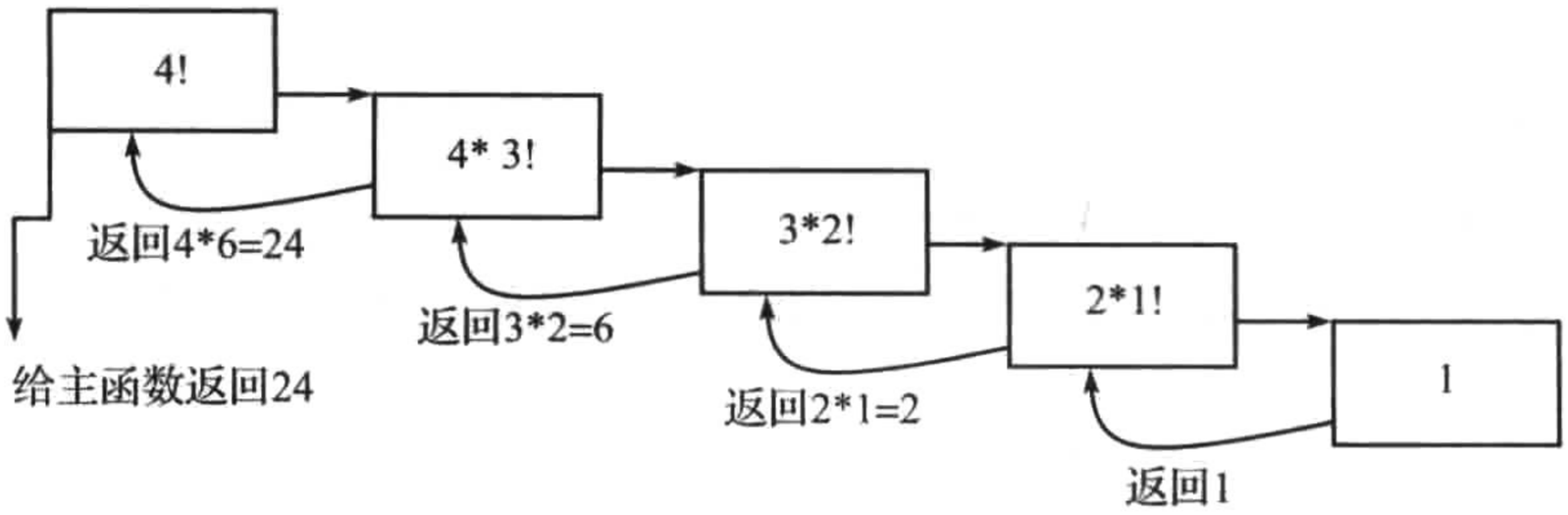
每次递归调用都在内存中生成一个新的函数副本(实际上仅仅是一些相关的变量)。一旦函数结束(即返回某些数据)，该返回函数的副本就从内存中删除。递归方案看起来简单，但是可视化和跟踪递归过程需要花费时间。为了更好地理解归递过程，考虑下面的例子。

```
int Print(int n) {
        if( n = = 0) // 这是用于递归结束的基本情形
                return 0;
        else {   System.out.println(n);
                return Print(n-1); // 再次递归调用自身
        }
}
```

在这个例子中，假设当参数 $n=4$ 时调用 print 函数，内存分配的可视化过程如下图所示。



现在，分析上面的阶乘函数。当参数 $n=4$ 时，阶乘函数的可视化过程如下图所示。




## 2.6 递归与迭代

在讨论递归时，一个基本的问题是递归和迭代，哪种方法更好？这个问题的答案取决于我们想做什么。递归方法通过类似镜像的方式来解决问题。当问题没有明显的答案时，递归方法通过简化问题来解决它。但是，每次递归调用都会增加开销(栈需要空间开销)。

**递归**

- 当到达基本情形时，递归终止。
- 每次递归调用都需要额外的空间用于栈帧(内存)开销。
- 如果出现无穷递归，程序可能会耗尽内存，并出现栈溢出。
- 某些问题采用递归方法更容易解决。

**迭代**

- 当循环条件为假时，迭代终止。
- 每次迭代不需要任何额外的空间开销。
- 由于没有额外的空间开销，所以若出现死循环，则程序会一直循环执行。
- 采用迭代求解问题可能没有递归解决方案那样显而易见。

## 2.7 递归说明

- 递归算法有两类情形：递归情形和基本情形。
- 每个递归函数必须终止于基本情形。
- 通常，迭代解决方案比递归解决方案更加有效(因为后者有函数调用的开销)。
- 一个递归算法可以通过使用栈代替递归函数的方式来实现，但通常是得不偿失的。这意味着任何能用递归求解的问题也能用迭代来求解。
- 对于某些问题，没有明显的迭代求解算法。
- 有些问题非常适合用递归来求解，而有些则不适合。

## 2.8 递归算法的经典用例

- 斐波那契数列、阶乘。
- 归并排序、快速排序。
- 二分查找。
- 树的遍历和许多树问题：中序遍历、前序遍历、后序遍历。
- 图的遍历：深度优先搜索、广度优先搜索。

- 动态规划例子。
- 分治算法。
- 汉诺塔。
- 回溯算法(将在下一节讨论)。

## 2.9　递归的相关问题

本章将介绍几个有关递归的问题，后续章节还将讨论其他递归相关的问题。在整本书的阅读过程中，读者将遇到很多递归问题。

**问题 1**　汉诺塔谜题。

**解答：**汉诺塔是一个数学谜题。有 3 根柱子(或木桩、塔)和一些可以在柱子之间来回移动的不同大小的圆盘。开始时，所有的圆盘按照从小到大的次序自上而下叠放在一根柱子上，形成一个圆锥结构。现在要求把整叠圆盘移动到另一根柱子上，移动时要遵守下面的规则：

- 每次只能移动一个圆盘。
- 每次移动，只能移动柱子最上面的一个圆盘到另一根柱子(这根柱子上有可能已有圆盘)。
- 任何时候不能出现大圆盘在小圆盘上方的情况。

**算法：**

- 将源柱最上面的 $n-1$ 个圆盘移到辅助柱。
- 将第 $n$ 个圆盘从源柱移到目的柱。
- 将辅助柱的 $n-1$ 个圆盘移到目的柱。
- 源柱最上面的 $n-1$ 个圆盘移到辅助柱又可以认为是一个新问题，并且可以用同样的方式解决。一旦能解决了只有 3 个圆盘的汉诺塔问题，那么这个算法可以求解任意数量圆盘的汉诺塔问题。

```
void TowersOfHanoi(int n, char frompeg, char topeg, char auxpeg) {
        /* 如果仅有一个圆盘，直接移动，然后返回 */
        if(n==1) {
                System.out.println("Move disk 1 from peg" + frompeg + " to peg" + topeg);
                return;
        }
        /* 利用C柱作辅助，将A柱最上面的n-1个圆盘移到B柱 */
        TowersOfHanoi(n-1,frompeg,auxpeg,topeg);
        /* 将余下的圆盘从A柱移到C柱*/
         System.out.println("Move disk from peg" + frompeg + " to peg" + topeg);
        /* 利用A柱作为辅助，将B柱上的n-1个圆盘移到C柱*/
        TowersOfHanoi(n-1,auxpeg,topeg,frompeg);
}
```

**问题 2**　给定一个数组，请用递归方法判定数组中的元素是否是有序的。

**解答：**

```
int isArrayInSortedOrder(int[] A, int index){
        if(A.length() == 1) return 1;
        return (A[index -1] <= A[index -2])?0:isArrayInSortedOrder(A, index -1);
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，主要是用于栈空间开销。

## 2.10 什么是回溯

回溯是一种采用分治策略进行穷举搜索的方法。

- 有时求解一个问题的最好算法是尝试所有的可能性。
- 这种方法通常很慢，但有标准工具能够辅助该过程。
- 工具：生成基本对象的算法，例如二进制串($n$ 位二进制串有 $2^n$ 种可能性)、排列($n!$)、组合($n!/r!(n-r)!$)、一般字符串(长度为 $n$ 的 $k$ 进制串有 $k^n$ 种可能性)，等等。
- 通过剪枝回溯可以加速的穷举搜索。

## 2.11 回溯算法的经典用例

- 二进制串：产生所有的二进制串。
- 生成 $k$ 进制串。
- 背包问题。
- 广义字符串。
- 哈密顿回路(参见第 9 章)。
- 图着色问题。

## 2.12 回溯的相关问题

**问题 3** 生成所有 $n$ 位长的字符串。假设 $A[0..n-1]$是一个大小为 $n$ 的数组。

**解答：**

```
void Binary(int n) {
        if(n < 1 )
            System.out.println(A);                    // 假设数组A是全局变量
        else {    A[n-1] = 0;
                  Binary (n - 1);
                  A[n-1] = 1;
                  Binary(n - 1);
        }
}
```

设 $T(n)$为函数 Binary($n$)的运行时间。假设函数 System. out. println 的时间开销为 $O(1)$。

$$T(n)=\begin{cases}c & n<0\\ 2T(n-1)+d & \text{否则}\end{cases}$$

根据问题规模减小和递归求解主定理，可以求得 $T(n)=O(2^n)$。这说明生成二进制串的算法是最优的。

**问题 4** 生成长度为 $n$ 的所有 $k$ 进制串，串中每位的取值为 $0..k-1$。

**解答：** 假设数组 $A[0..n-1]$保存当前的 $k$ 进制串。调用函数 k-string($n$, $k$)：

```
void k-string(int n, int k) {
        // 处理长度为n的所有k进制串
        if(n < 1 )
            System.out.println(A);                    // 假设数组A是全局变量
        else {    for (int j = 0 ; j < k ; j++) {
                      A[n - 1] = j;
```

```
                k-string(n- 1, k);
            }
        }
}
```

设 $T(n)$为函数 k-string($n$, $k$)的运行时间。则有，

$$T(n)=\begin{cases}c & \text{如果 } n<0\\ kT(n-1)+d & \text{否则}\end{cases}$$

根据问题规模减小和递归求解主定理，可以求得 $T(n)=O(k^n)$。

**注意：**更多相关的问题，请参见第 15 章。



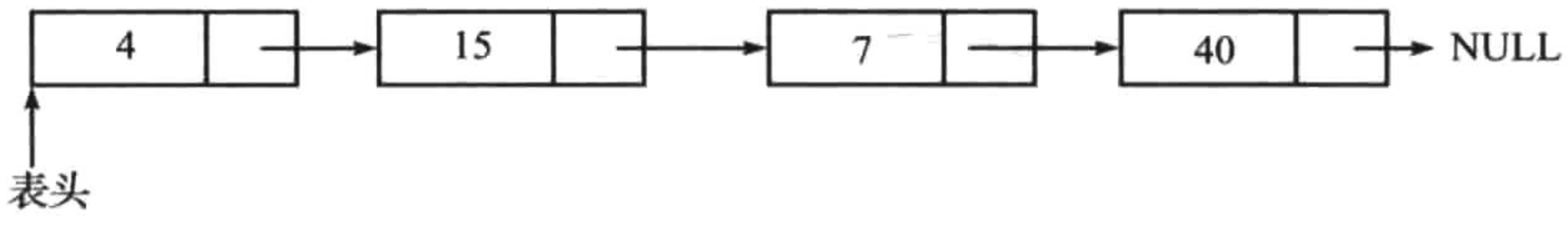
Chapter 3

# 第3章 链　　表

## 3.1　什么是链表

链表是一种用于存储数据集合的数据结构。链表有以下属性：

- 相邻元素之间通过指针连接。
- 最后一个元素的后继指针值为 NULL。
- 在程序执行过程中，链表的长度可以增加或缩小。
- 链表的空间能够按需分配(直到系统内存耗尽)。
- 没有内存空间的浪费(但是链表中的指针需要一些额外的内存开销)




## 3.2　链表抽象数据类型

链表抽象数据类型中的操作如下：

**链表的主要操作**

- 插入：插入一个元素到链表中。
- 删除：移除并返回链表中指定位置的元素。

**链表的辅助操作**

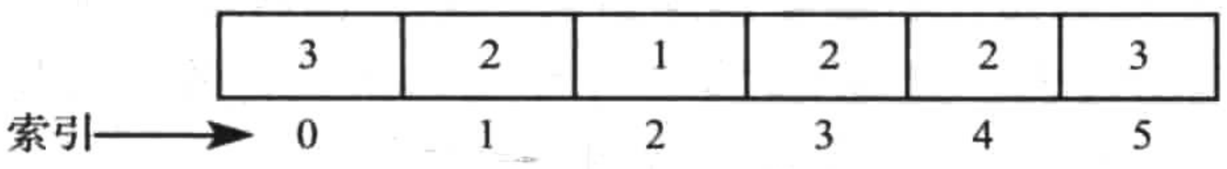
- 删除链表：移除链表中的所有元素(清空链表)。
- 计数：返回链表中元素的个数。
- 查找：寻找从链表表尾开始的第 $n$ 个结点(node)。

## 3.3 为什么要用链表

有许多其他的数据结构可以像链表一样做同样的事情。在讨论链表前，首先要了解链表和数组的区别。链表和数组都可用于存储数据集合。由于两者的用途相同，所以需要对它们的用法进行区分。也就是说，要了解在何种情况下适合用数组，而在何种情况下适合用链表。

## 3.4 数组概述

整个数组所有的元素都存储在操作系统分配的一个内存块中。通过使用特定元素的索引作为数组下标，可以在常数时间内访问数组元素。




### 1. 为什么能在常数时间内访问数组元素

为了访问一个数组元素，该元素的内存地址需要计算其距离数组基地址的偏移量。需用一个乘法计算偏移量，再加上基地址，就可获得某个元素的内存地址。首先计算元素数据类型的存储空间大小，然后将它乘以元素在数组中的索引，最后加上基地址，就可计算出该索引位置元素的地址。整个过程需要一次乘法和一次加法运算。因为这两个运算的执行时间是常数时间，所以可以认为数组访问操作能在常数时间内(执行)完成。

### 2. 数组的优点

- 简单且易用。
- 访问元素快(常数时间)。

### 3. 数组的缺点

- **大小固定：** 数组的大小是静态的(在使用前指定数组的大小)。
- **分配一个连续空间块：** 数组初始分配空间时，有时无法分配能存储整个数组的内存空间(当数组规模太大时)。
- **基于位置的插入操作实现复杂：** 如果要在数组中的给定位置插入元素，可能需要移动存储在数组中的其他元素，这样才能腾出指定的位置来放插入的新元素。如果在数组的开始位置插入元素，那么移动操作的开销将更大。

### 4. 动态数组

动态数组(也称为可增数组、可变长数组、动态表、数组表)是一种可随机存取且可自动调整大小的线性数据结构，能够添加或删除元素。

实现动态数组的一个简单方法是，首先初始化固定大小的数组。一旦数组存储满了，创建一个两倍于原始数组大小的新数组。同样，若数组中存储的元素个数小于数组大小的一半，则把数组大小减少一半。

**注意：** 我们将在第4、5、14章中看到动态数组的实现。

### 5. 链表的优点

链表也有其优缺点。链表的优点是，它们可以在常数时间内扩展。当创建数组时，必须分配能存储一定数量元素的内存。如果向数组中添加更多的元素，那么必须创建一个新的数组，然后把原数组中的元素复制到新数组中，这将花费大量的时间。

当然，可以通过为数组预先分配一个很大的空间来预防上述情况的发生，但是这个方法可能会因为分配超过用户需要的空间而造成内存浪费。而对于链表，初始时仅需要分配一个元素的存储空间，并且添加新的元素也很容易，不需要做任何内存复制和重新分配操作。

**6. 链表的缺点**

链表有许多不足。链表的主要缺点在于访问单个元素的时间开销问题。数组是随机存取的，即存取数组中任一元素的时间开销为 $O(1)$。而链表在最差情况下访问一个元素的开销为 $O(n)$。数组在存取时间方面的另外一个优点是内存的空间局部性。由于数组被定义为连续的内存块，所以任何数组元素与其邻居是物理相邻的。这极大得益于现代 CPU 的缓存模式。

尽管链表的动态分配存储空间有很大的优势，但在存储和检索数据的开销方面却有很大的不足。有时很难对链表操作。如果要删除最后一项，倒数第二项必须更改后继指针值为 NULL。这需要从头遍历链表，找到倒数第二个结点的链接，并设置其后继指针为 NULL。

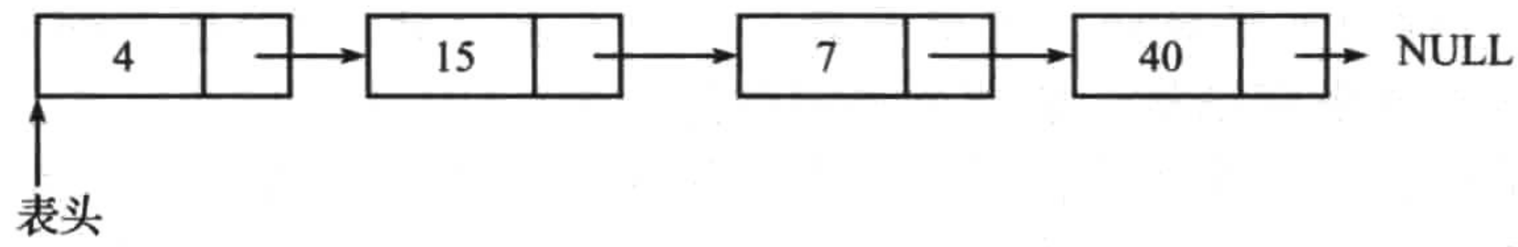
最后，链表中的额外指针引用需要浪费内存。

## 3.5 链表、数组和动态数组的比较

| 参数 | 链表 | 数组 | 动态数组 |
|---|---|---|---|
| 索引 | $O(n)$ | $O(1)$ | $O(1)$ |
| 在最前端插入/删除 | $O(1)$ | $O(n)$，如果数组空间没有填满(需要移动元素) | $O(n)$ |
| 在最末端插入 | $O(n)$ | $O(1)$，如果数组空间没有填满 | $O(1)$，如果数组空间没有填满<br>$O(n)$，如果数组空间已经填满 |
| 在最末端删除 | $O(n)$ | $O(1)$ | $O(n)$ |
| 在中间插入 | $O(n)$ | $O(n)$，如果数组空间没有填满(需要移动元素) | $O(n)$ |
| 在中间删除 | $O(n)$ | $O(n)$，如果数组空间没有填满(需要移动元素) | $O(n)$ |
| 空间浪费 | $O(n)$ | 0 | $O(n)$ |

## 3.6 单向链表

链表通常是指单向链表，它包含多个结点，每个结点有一个指向后继元素的 next(下一个)指针。表中最后一个结点的 next 指针值为 NULL，表示该链表的结束。




下面是一个链表的类型声明：

```
public class ListNode {
        private int data;
        private ListNode next;
        public ListNode(int data){
                this.data = data;
        }
```

```
public void setData(int data){
        this.data = data;
}
public int getData(){
        return data;
}
public void setNext(ListNode next){
        this.next = next;
}
public ListNode getNext(){
        return this.next;
        }
}
```

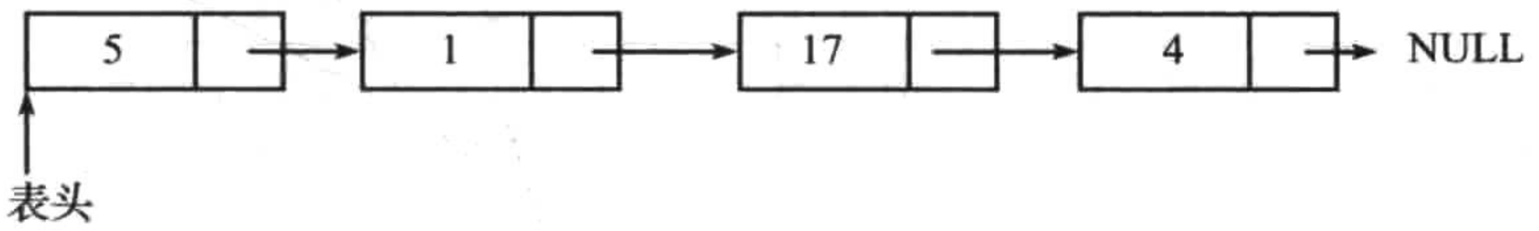
**1. 链表的基本操作**

- 遍历链表。
- 在链表中插入一个元素。
- 从链表中删除一个元素。

**2. 链表的遍历**

假设表头结点的指针指向链表中的第一个结点。遍历链表需完成以下步骤。

- 沿指针遍历。
- 遍历时显示结点的内容(或计数)。
- 当 next 指针的值为 NULL 时结束遍历。




ListLength()函数的输入为链表，其功能是统计链表中结点的个数。下面是函数的代码，还可以添加额外的输出函数来输出链表的数据。

```
int ListLength(ListNode headNode) {
        int length = 0;
        ListNode currentNode = headNode;
        while(currentNode != null){
                length++;
                currentNode = currentNode.getNext();
        }
        return length;
}
```

时间复杂度为 $O(n)$，用于扫描长度为 $n$ 的链表。空间复杂度为 $O(1)$，仅用于创建临时变量。

**3. 单向链表的插入**

单向链表的插入操作可分为以下 3 种情况：

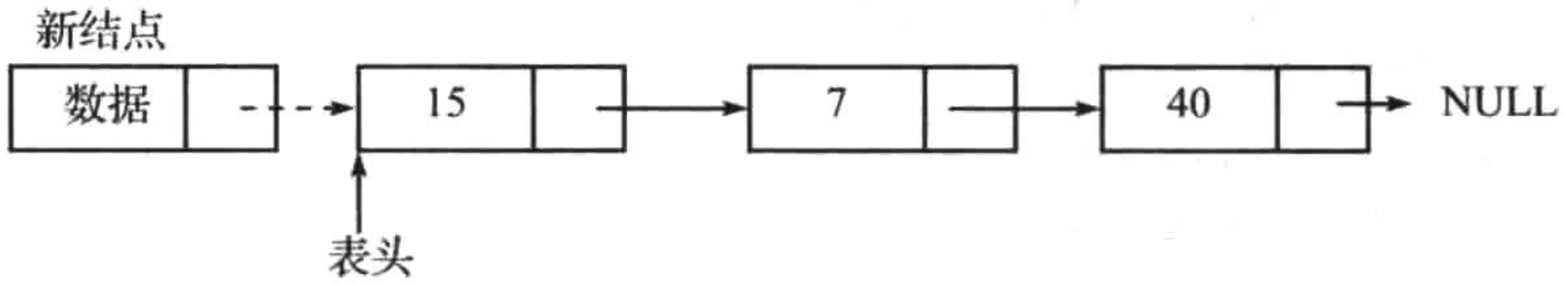
- 在链表的表头前插入一个新结点(链表开始处)。
- 在链表的表尾后插入一个新结点(链表结尾处)。
- 在链表的中间插入一个新结点(随机位置)。

**注意：** 为了在链表的某个位置 $p$ 插入一个元素，假设完成元素插入后，新结点的位置是 $p$。

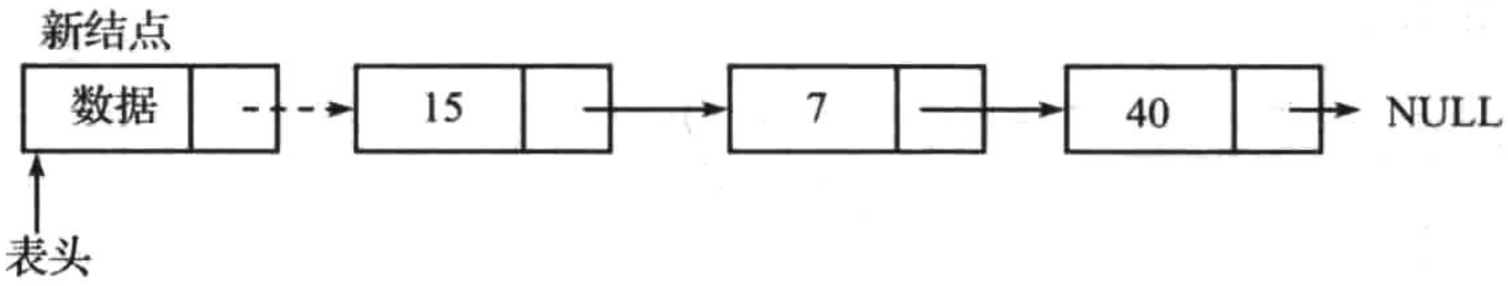
### 4. 在单向链表的开头插入结点

若需要在当前表头结点前插入一个新结点，只需要修改一个 next 指针(新结点的 next 指针)，可通过以下两个步骤完成：

- 更新新节点 next 指针，使其指向当前的表头结点。



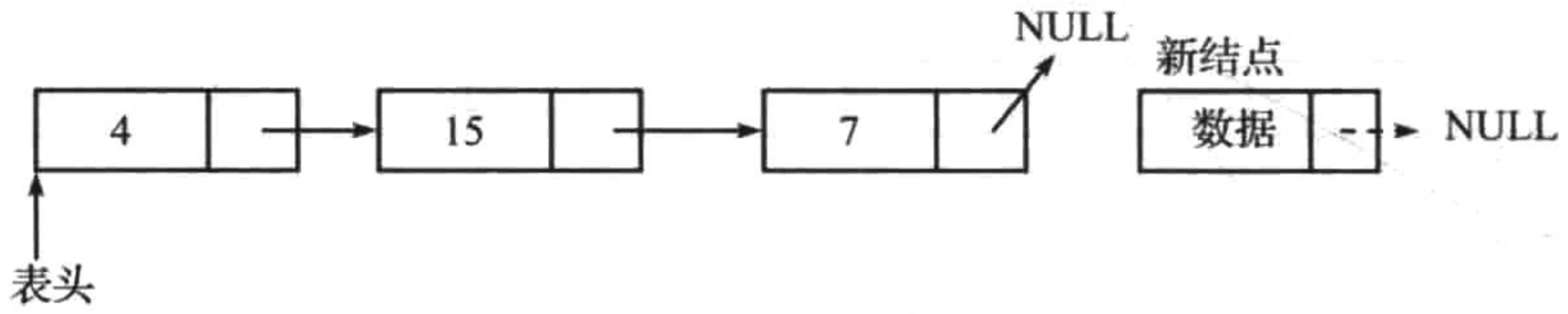

- 更新表头指针的值，使其指向新结点。



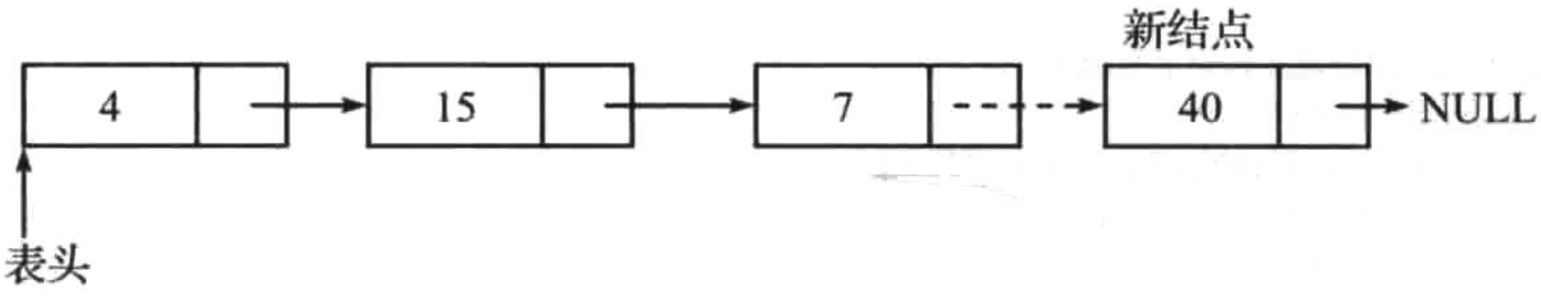

### 5. 在单向链表的结尾插入结点

若需要在表尾后插入新结点，则需修改两个 next 指针(最后一个结点的 next 指针和新结点的 next 指针)。

- 新结点的 next 指针指向 NULL。



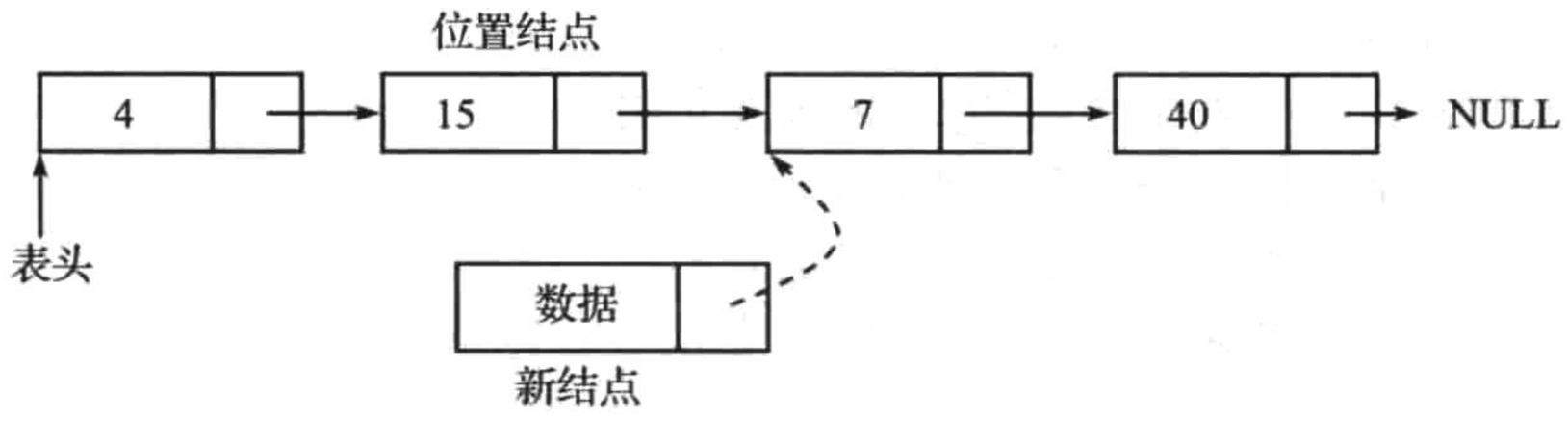

- 最后一个结点的 next 指针指向新结点。



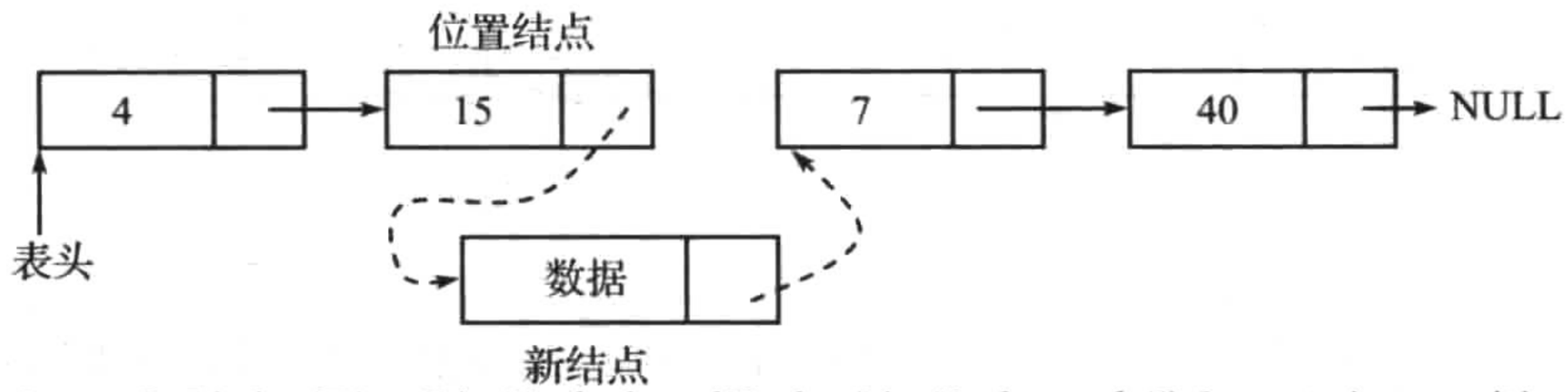

### 6. 在单向链表的中间插入结点

假设给定插入新结点的位置。在这种情况下，需要修改两个 next 指针。

- 如果要在位置 3 增加一个元素，则需要将指针定位于链表的位置 2。即需要从表头开始经过两个结点，然后插入新结点。为了简单起见，假设第二个结点为位置结点，新节点的 next 指针指向位置结点(我们要在此处增加新结点)的下一个结点。



- 位置结点的 next 指针指向新结点。




```
ListNode InsertInLinkedList (ListNode headNode, ListNode nodeToInsert, int position) {
if(headNode == null)                    // 若链表为空，插入
        return nodeToInsert;
int size = ListLength(headNode);
if(position > size+1 || position < 1){
        System.out.println("Position of node to insert is invalid. The valid inputs are 1 to" +
        (size+1));
        return headNode;
}
if(position == 1){                      // 在链表开头插入
        nodeToInsert.setNext(headNode);
        return nodeToInsert;
}else{
        // 在链表中间或末尾插入
        ListNode previousNode = headNode;
        int count = 1;
        while(count < position-1){
                previousNode = previousNode.getNext();
                count++;
        }
        ListNode currentNode = previousNode.getNext();
        nodeToInsert.setNext(currentNode);
        previousNode.setNext(nodeToInsert);
    }
    return headNode;
}
```

**注意：**也可以分别实现插入操作的 3 种情况。

时间复杂度为 $O(n)$。因为在最差情况下，可能需要在链表尾部插入结点。空间复杂度为 $O(1)$，仅用于创建一个临时变量。

### 7. 单向链表的删除

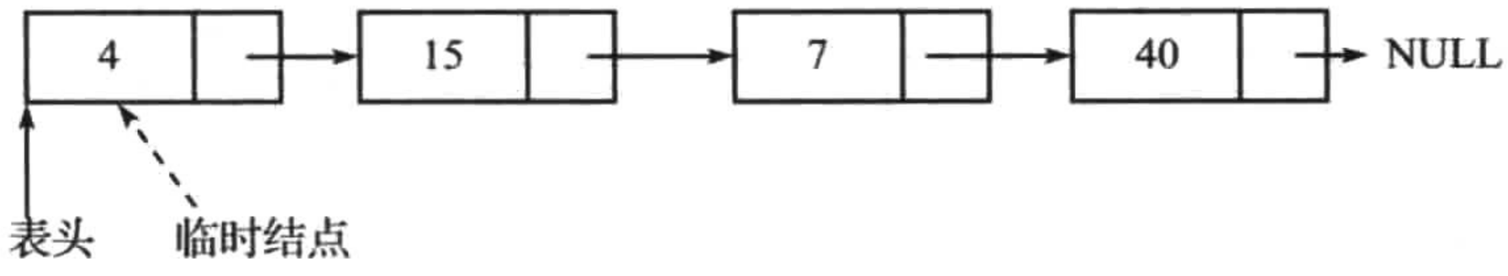
与单向链表的插入相似，删除也分 3 种情况：

- 删除链表的表头(第一个)结点。
- 删除链表的表尾(最后一个)结点。
- 删除链表中间的结点。

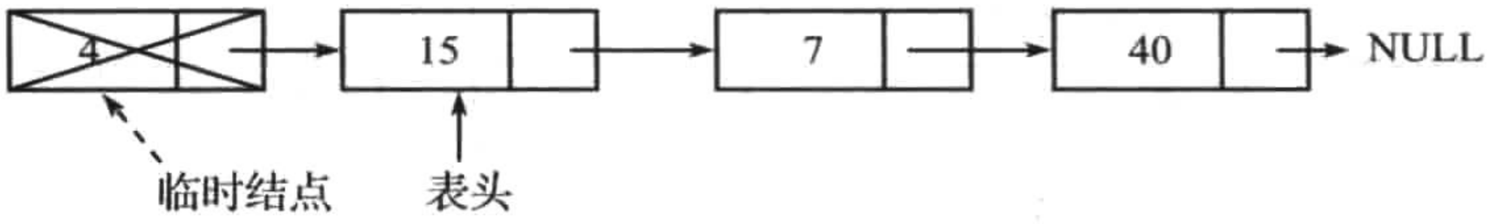
### 8. 删除单向链表的第一个结点

从链表中删除第一个结点，可通过两步来实现：

- 创建一个临时结点，它指向表头指针所指的结点。



- 修改表头指针的值，使其指向下一个结点，并移除临时结点。



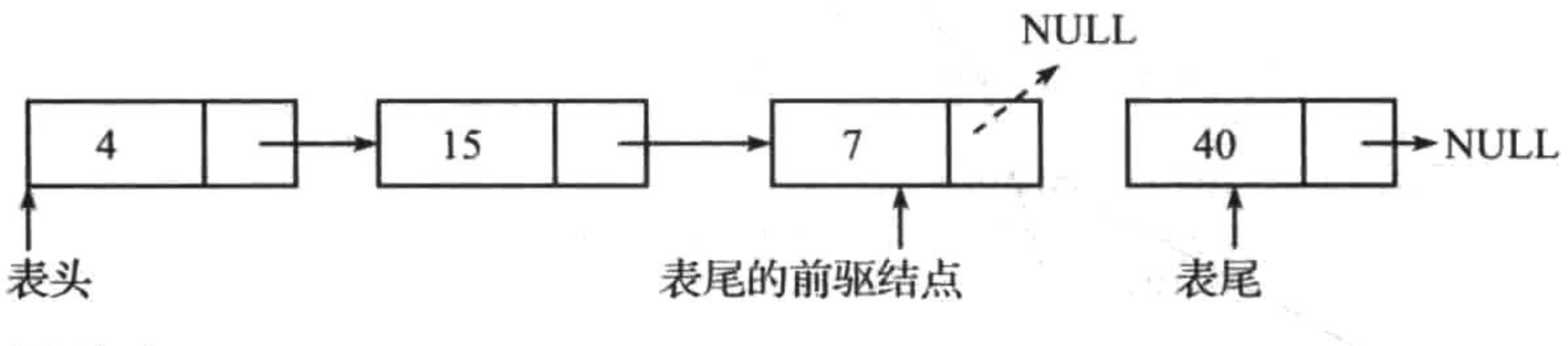

### 9. 删除单向链表的最后一个结点

在这种情况下，链表的最后一个结点将被删除。该操作比删除链表的第一个结点稍微复杂一些，因为算法需要找到表尾结点的前驱结点。这需要 3 步来实现：

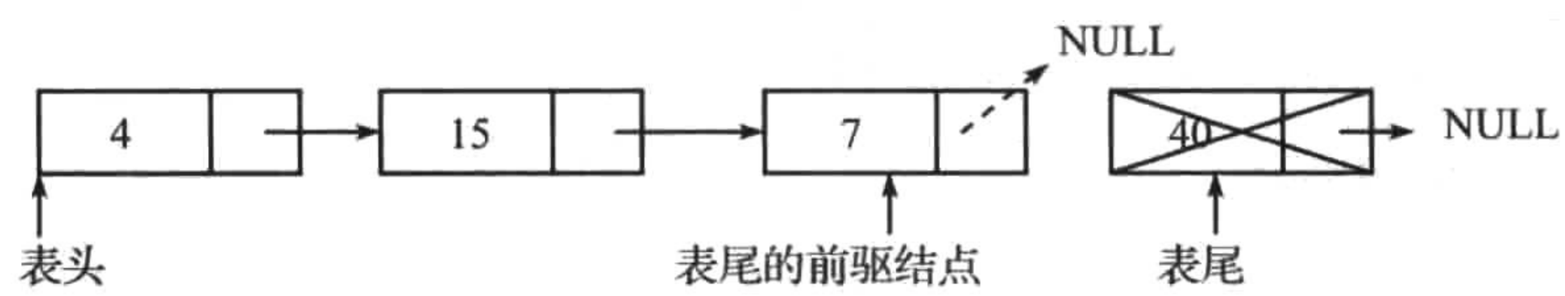
- 遍历链表，在遍历时还要保存前驱(前一次经过的)结点的地址。当遍历到链表的表尾时，将有两个指针，分别是表尾结点的指针 tail(表尾)及指向表尾结点的前驱结点的指针。




- 将表尾的前驱结点的 next 指针更新为 NULL。



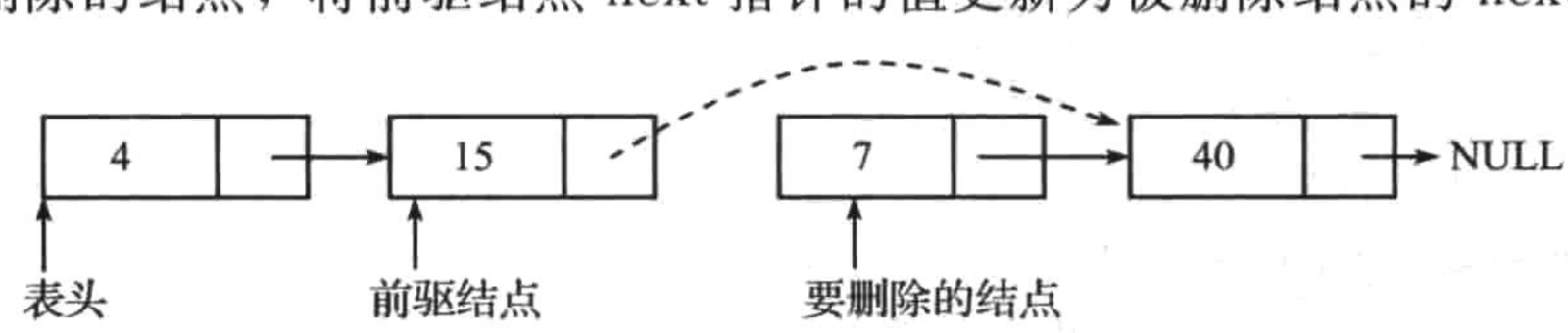

- 移除表尾结点。



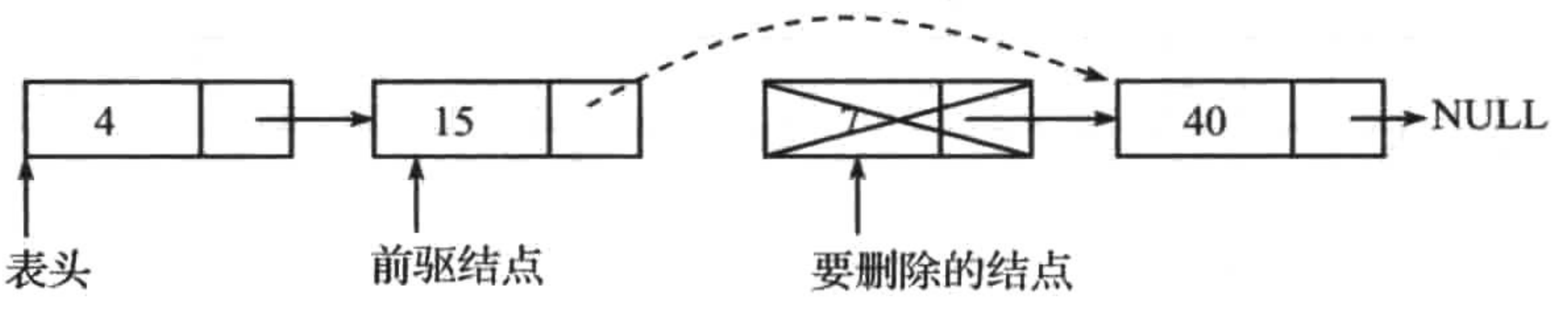

### 10. 删除单向链表中间的一个结点

在这种情况下，被删除的结点总是位于两个结点之间，因此不需要更新表头和表尾的指针。该删除操作可通过两步实现：

- 与上一种删除情况类似，在遍历时保存前驱(前一次经过的)结点的地址。一旦找到要被删除的结点，将前驱结点 next 指针的值更新为被删除结点的 next 指针的值。




- 移除需删除的当前结点。



```
ListNode DeleteNodeFromLinkedList(ListNode headNode, int position){
        int size = getLinkedListLength(headNode);
        if(position > size || position < 1){
                System.out.println("Position of node to delete is invalid. The valid inputs are 1
                to " + size);
                return headNode;
        }
        if(position == 1){          // 删除单向链表的表头结点
                ListNode currentNode = headNode.getNext();
                headNode = null;
                return currentNode;
        }else{                                  // 删除中间或表尾结点
                ListNode previousNode = headNode;
                int count = 1;
                while(count < position){
                        previousNode = previousNode.getNext();
                        count++;
                }
                ListNode currentNode = previousNode.getNext();
                previousNode.setNext(currentNode.getNext());
                currentNode = null;
        }
        return headNode;
}
```

时间复杂度为 $O(n)$。在最差情况下，可能需要删除链表的表尾结点。空间复杂度为 $O(1)$，仅用于创建一个临时变量。

**11. 删除单向链表**

这个操作通过将当前结点存储在临时变量中然后释放当前结点(空间)的方式来完成。当释放完当前结点(空间)后，移动到下一个结点并将其存储在临时变量中，然后不断重复该过程直至释放所有结点。

```
void DeleteLinkedList(ListNode head) {
        ListNode auxilaryNode, iterator = head;
        while (iterator != null) {
                auxilaryNode = iterator.getNext();
                iterator = null;                    // 在Java中，垃圾回收器将自动处理
                iterator = auxilaryNode; // 在实际应用中，不需要实现该内容
        }
}
```

时间复杂度为 $O(n)$，用于扫描大小为 $n$ 的整个链表。

空间复杂度为 $O(1)$，用于创建临时变量。

## 3.7 双向链表

双向链表的优点是：对于链表中一个给定的结点，可以从两个方向进行操作。在单向链表中，只有获得结点的前驱结点的指针，才能删除该结点。然而，在双向链表中，即使没有一个结点的前驱结点的地址，也能删除该结点(因为每个结点有都一个指向前驱结点的指针，可以直接后退到前驱结点)。

双向链表主要的缺点是：

- 每个结点需再添加一个额外的指针，因此需要更多的空间开销。
- 结点的插入或删除更加费时(需要更多的指针操作)。

与单向链表类似，下面开始实现双向链表的各个操作。首先给出双向链表的类型声明：

```
public class DLLNode {
        private int data;
        private DLLNode next;
        private DLLNode previous;
        public DLLNode(int data){
            this.data = data;
    }
    public void setData(int data){
            this.data = data;
    }
    public int getData(){
            return data;
    }
    public void setNext(DLLNode next){
            this.next = next;
    }
    public DLLNode getNext(){
            return this.next;
    }
        public void setPrevious(DLLNode previous){
                this.previous = previous;
        }
        public DLLNode getPrevious(){
                return this.previous;
        }
}
```

### 1. 双向链表的插入操作

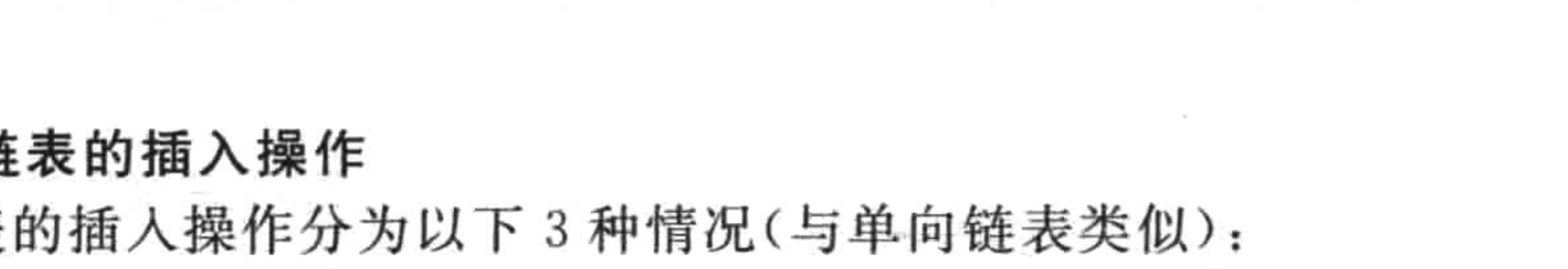
双向链表的插入操作分为以下 3 种情况(与单向链表类似)：

- 在链表的开头前插入一个新结点
- 在链表的末尾后插入一个新结点(链表的最后)
- 在链表的中间插入一个新结点

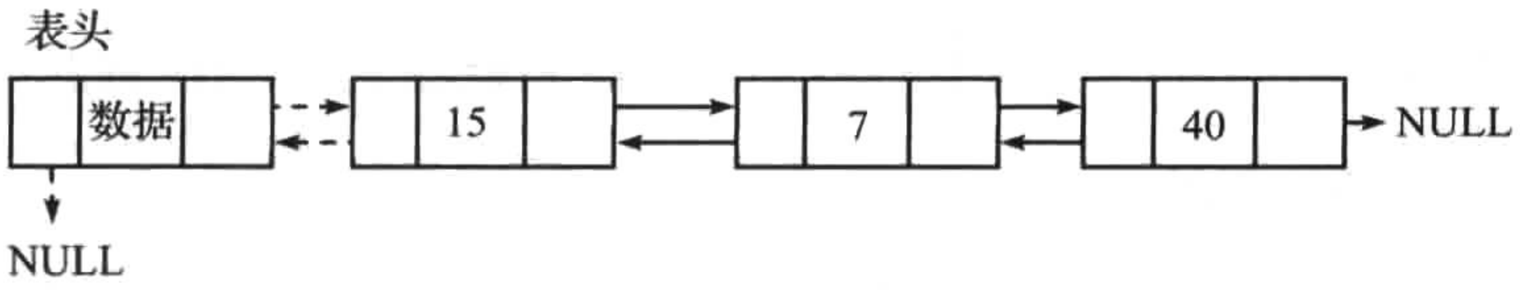
### 2. 在双向链表的开始插入一个结点

在这种情况下，将新结点插入表头结点之前。修改前驱指针和后继指针的值，并通过以下两步实现：

- 将新结点的右(后继)指针更新为指向当前的表头结点(下图中的虚线指针)，将新结点的左(前驱)指针赋值为 NULL。



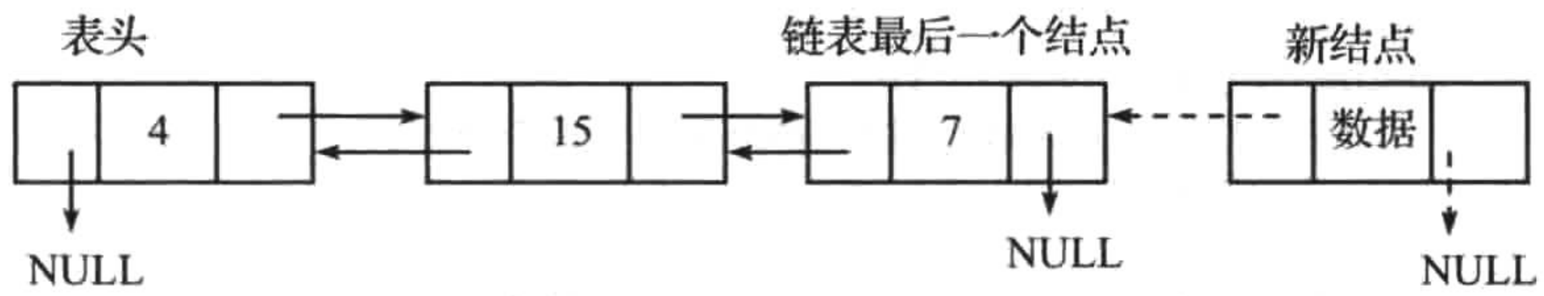

- 将表头结点左(前驱)指针更新为指向新结点，然后将新结点作为表头。



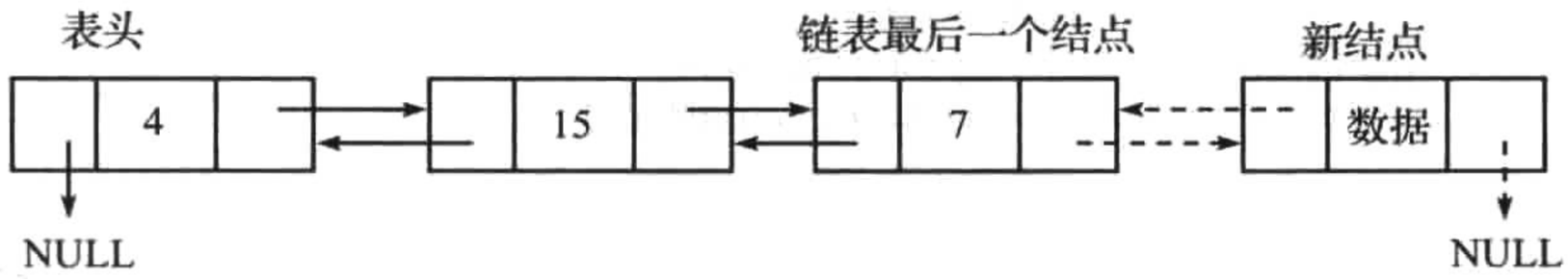

### 3. 在双向链表的末尾插入一个结点

在这种情况下，需要遍历到链表的最后，然后插入新结点。

- 新结点的右(后继)指针指向 NULL，其左(前驱)指针指向表尾结点。



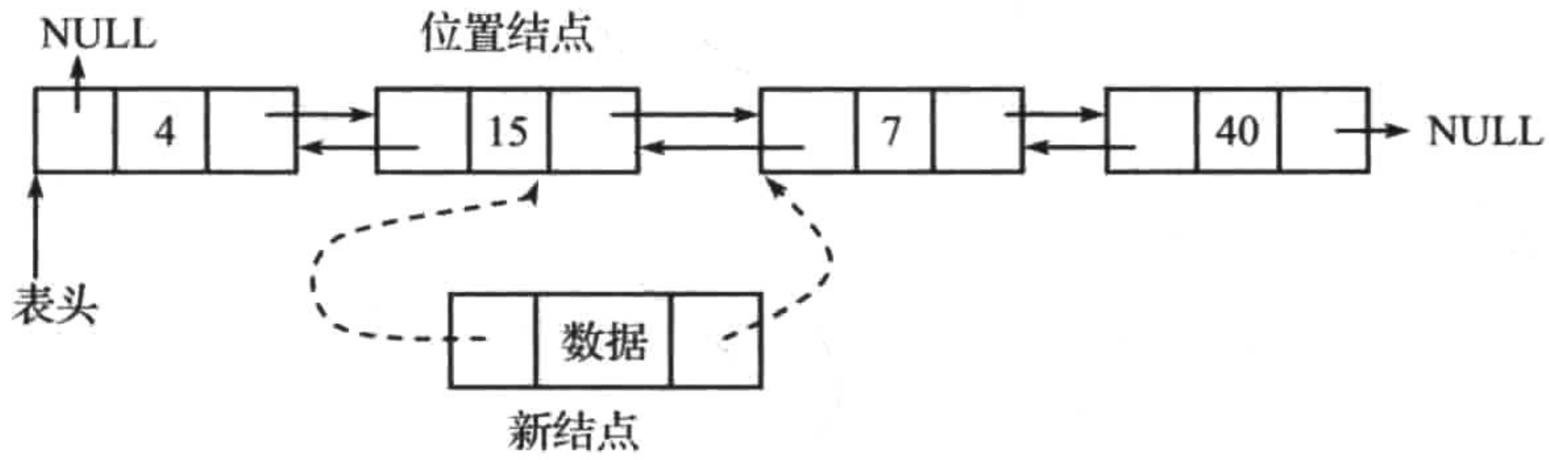

- 更新最后一个结点的右指针，使其指向新结点。



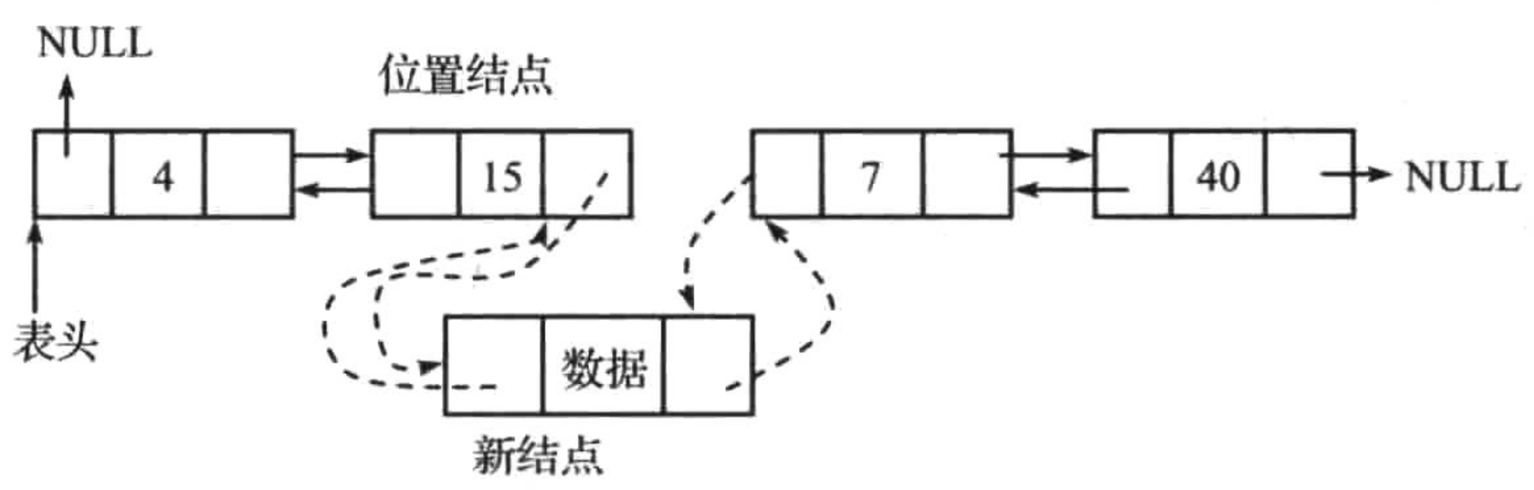

### 4. 在双向链表的中间插入一个结点

如 3.6 节所述，需遍历链表直到位置结点，然后插入新结点。

- 新结点的右(后继)指针指向需要插入新结点的位置结点的后继结点。然后令新结点的左(前驱)指针指向位置结点。




- 位置结点的后继结点的左(前驱)指针指向新结点，位置结点的右(后继)结点指向新结点。




```
DLLNode DLLInsert(DLLNode headNode, DLLNode nodeToInsert, int position){
        if(headNode == null)                    // 最初链表为空时插入
                return nodeToInsert;
        int size = getDLLLength(headNode);
```

```
if(position > size+1 || position < 1){
    System.out.println("Position of nodeToInsert is invalid. " +"The valid inputs are 1 to " +
    (size+1));
    return headNode;
}
if(position == 1){// 在链表开头插入
        nodeToInsert.setNext(headNode);
        headNode.setPrevious(nodeToInsert);
        return nodeToInsert;
}else{    // 在链表中间或末尾插入
        DLLNode previousNode = headNode;
        int count = 1;
        while(count < position-1){
                previousNode = previousNode.getNext();
                count++;
        }
        DLLNode currentNode = previousNode.getNext();
        nodeToInsert.setNext(currentNode);
        if(currentNode != null)
                currentNode.setPrevious(nodeToInsert);
        previousNode.setNext(nodeToInsert);
        nodeToInsert.setPrevious(previousNode);
    }
    return headNode;
}
```

时间复杂度为 $O(n)$。在最差情况下，需要在链表的尾部插入结点。

空间复杂度为 $O(1)$，用于创建临时变量。

### 5. 双向链表的删除

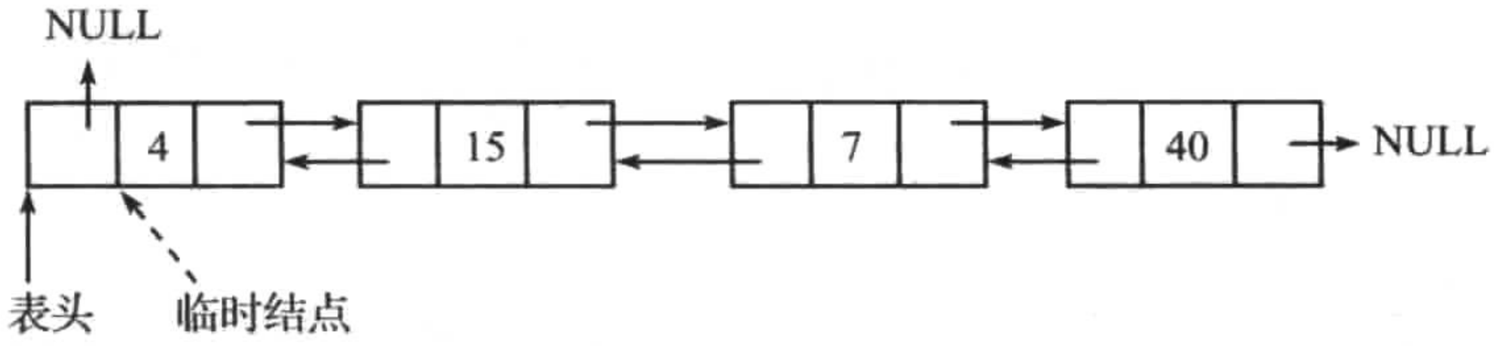
与单向链表的删除相似，也有 3 种情况：

- 删除链表的表头(第一个)结点。
- 删除链表的表尾(最后一个)结点。
- 删除链表中间的一个结点。

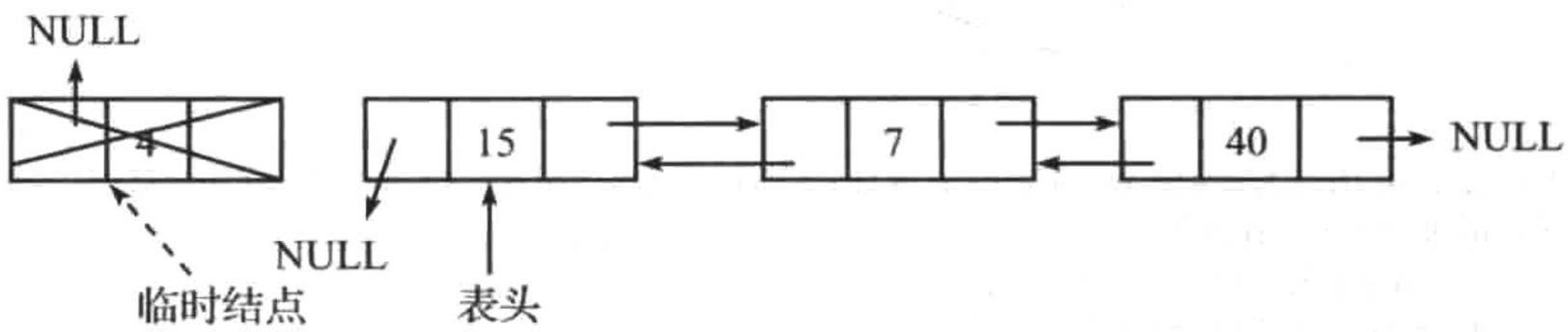
### 6. 删除双向链表的第一个结点

在这种情况下，要从双向链表中删除第一个结点(当前的表头结点)。可通过两步来实现：

- 创建一个临时结点，它与表头指向同一个结点。



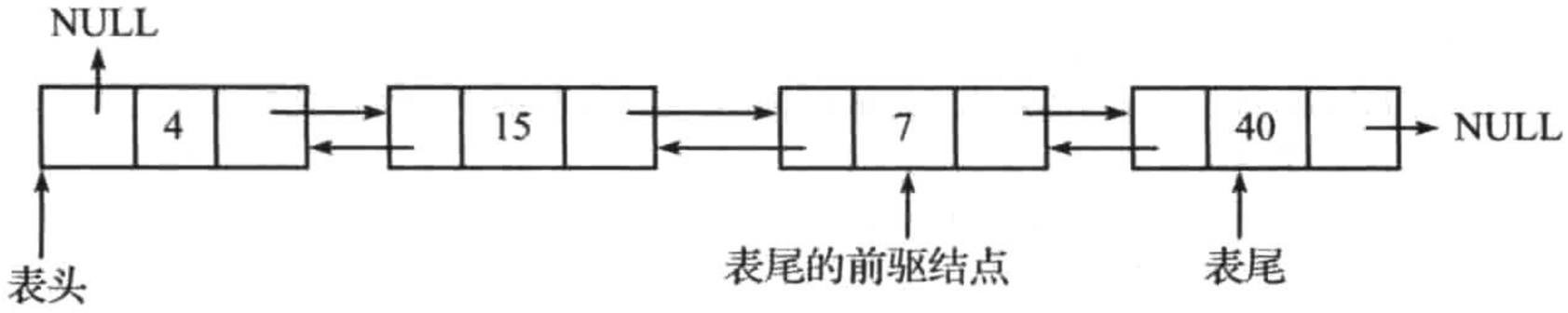

- 修改表头结点指针 head(表头)，使其指向下一个结点，将表头结点的左(前驱)指针更改为 NULL，然后移除临时结点。



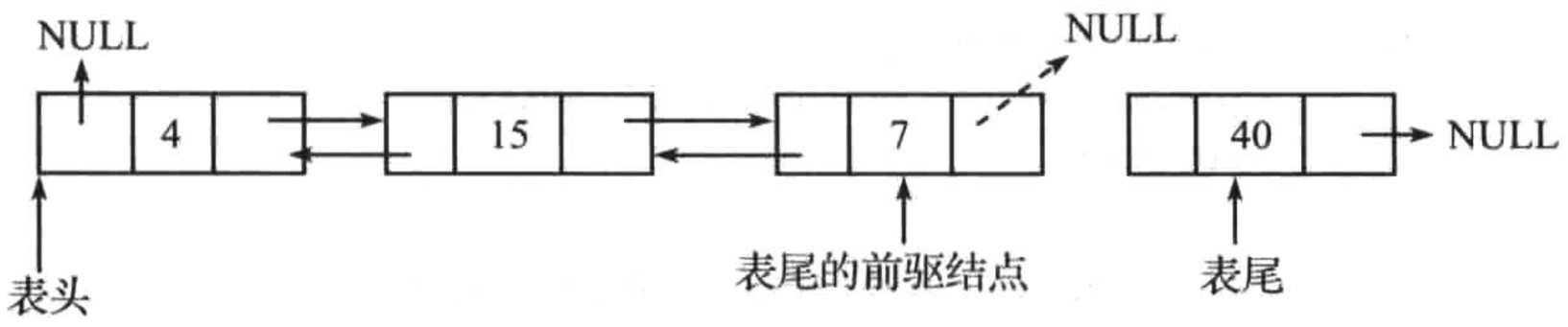
### 7. 删除双向链表的最后一个结点

这种情况的处理比删除双向链表的第一个结点要复杂一些，因为要找到表尾结点的前驱结点。需要3步来实现：

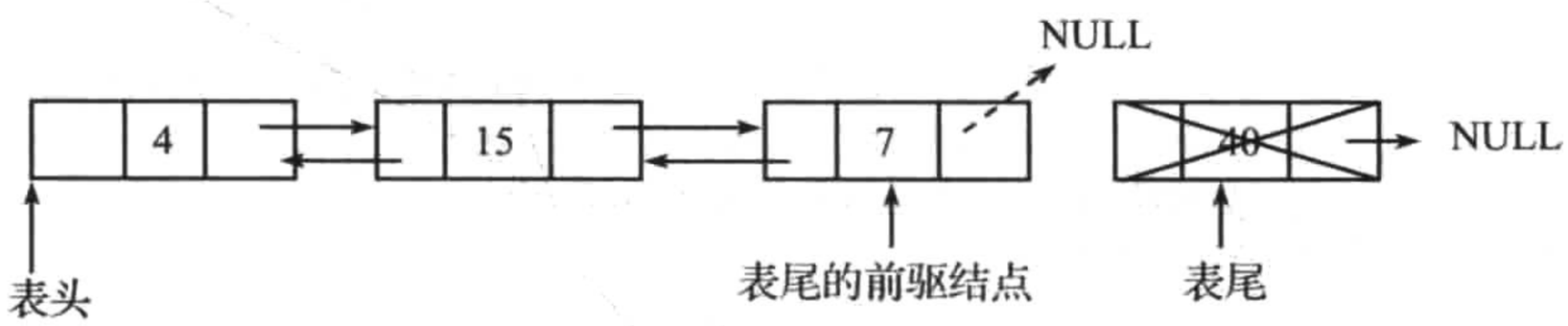
- 遍历链表，在遍历时还要保存前驱(前一次经过的)结点的地址。当遍历到表尾时，有两个指针分别是指向表尾结点的tail(表尾)指针和指向表尾结点的前驱结点的指针。




- 更新表尾的前驱结点的next指针为NULL。



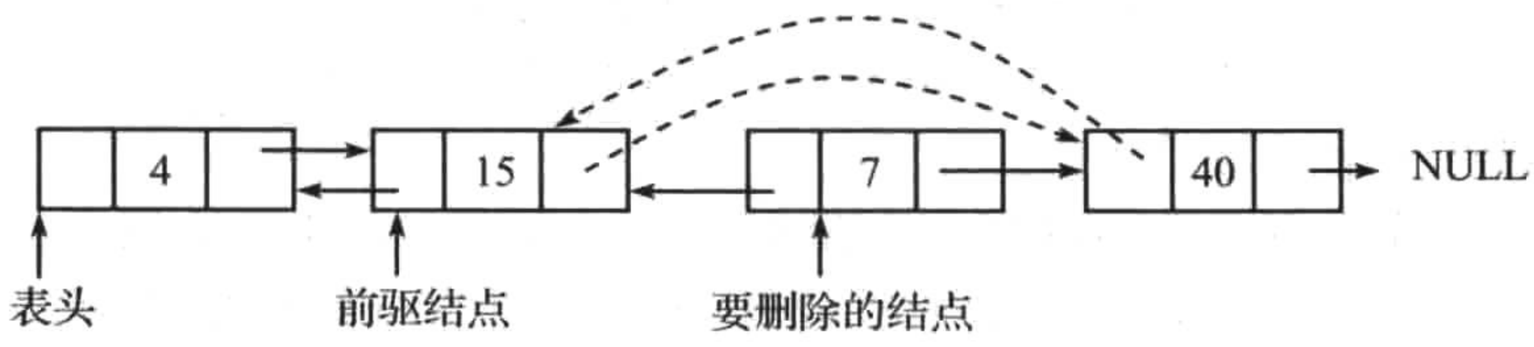

- 移除表尾结点。



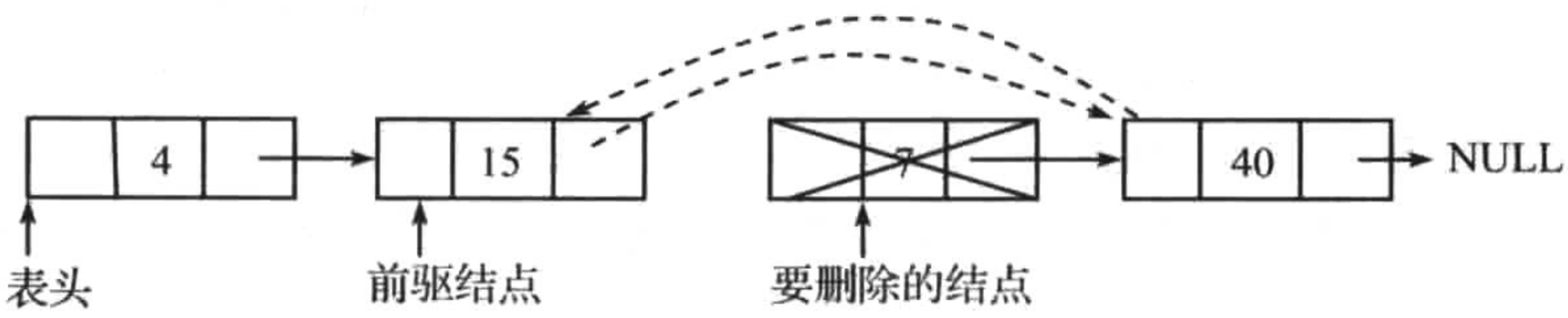

### 8. 删除双向链表中间的一个结点

在这种情况下，被删除的结点总是位于两个结点之间，因此表头和表尾的值不需要更新。该删除操作可通过两步实现：

- 与上一种删除情况类似，在遍历链表时保存前驱(前一次经过的)结点。一旦找到要删除的结点，更改前驱结点的next指针使其指向被删除结点的下一个结点，更改被删除结点的后继结点的previous指针指向被删除结点的前驱结点。




- 移除被删除的当前结点。



```
DLLNode DLLDelete(DLLNode headNode, int position){
        int size  = getDLLLength(headNode);
        // 如果被删除位置不在链表长度范围内，报错并返回
        if(position > size  || position < 1){
                System.out.println("Position of node to delete is invalid.The valid inputs are 1 to" +
                size);
                return headNode;
        }
        if(position == 1){// 删除链表的第一个结点
                DLLNode currentNode = headNode.getNext();
                headNode = null;
                currentNode.setPrevious(null);
                return currentNode;
}else{                          // 删除中间或表尾结点
                DLLNode previousNode = headNode;
                int count = 1;
                while(count < position-1){
                        previousNode = previousNode.getNext();
                        count++;
                }

                DLLNode currentNode = previousNode.getNext();
                DLLNode laterNode = currentNode.getNext();
                previousNode.setNext(laterNode);
                if(laterNode != null)
                        // 如果被删除结点的后继结点不是NULL结点，那么设置其前驱
                        // 指针指向被删除结点的前驱结点
                        laterNode.setPrevious(previousNode);
                currentNode = null;
        }
        return headNode;
}
```

时间复杂度为 $O(n)$，因为最差情况下，可能需要删除链表的表尾结点。

空间复杂度为 $O(1)$，仅用于创建一个临时变量。

## 3.8 循环链表

在单向链表和双向链表中，都采用 NULL 值表示链表的结束。然而，循环链表没有结束标志。当遍历循环链表时需要特别小心，否则将会无限地遍历链表，因为在循环链表中每个结点都有一个后继结点。

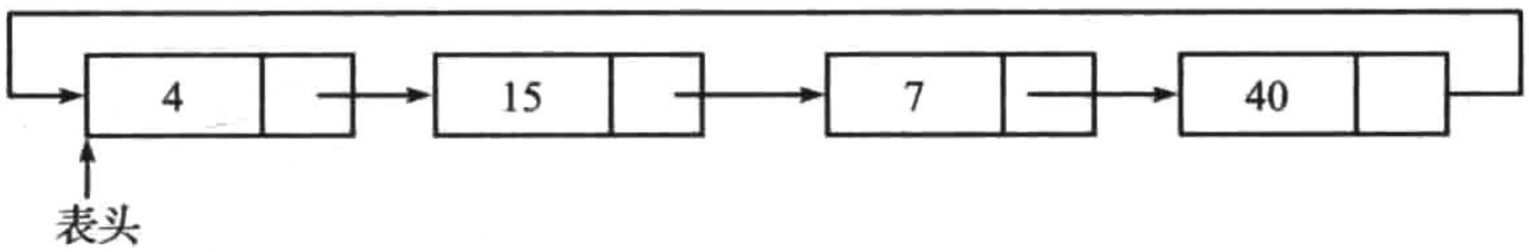
注意，与单向链表不同，循环链表中没有 next 指针为 NULL 的结点。循环链表在某些情况下是非常有用的。例如，当多个进程需要在相同的时间内使用同一个计算机资源(CPU)时，必须确保在所有其他进程使用这些资源完前，没有进程访问该资源(轮询算法)。

在循环链表中，使用表头结点访问元素(与单向链表和双向链表中的表头结点相似)。为了便于阅读，假设循环链表的类名为 CLLNode。

### 1. 统计循环链表的结点个数

循环链表可以通过标记为表头的结点进行访问。为了统计结点个数，只能从标记为表头的结点开始遍历，利用虚拟结点——当前结点(current)，当当前结点再次到达开始

结点表头时结束计数过程。如果链表为空，表头结点为 NULL，在这种情况下设结点个数(count)等于 0。否则，设置当前结点指向第一个结点(表头结点)，然后遍历链表进行计数，直到当前结点达到开始结点。



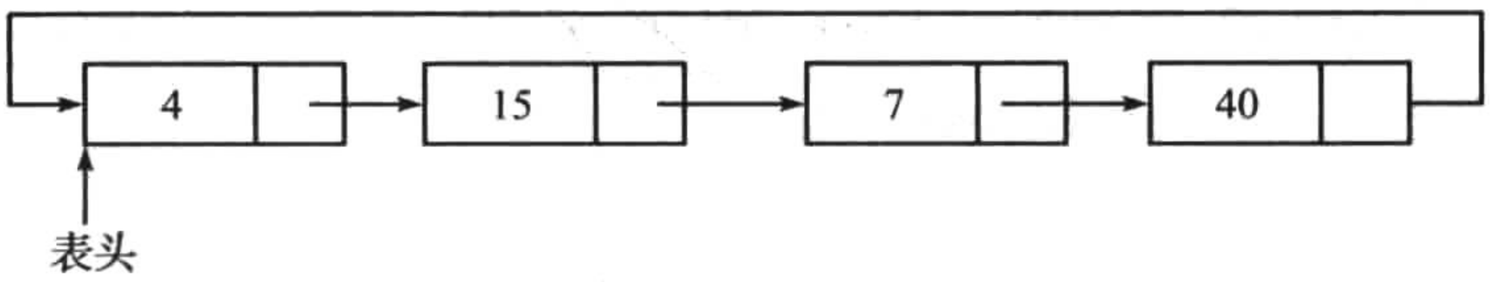

```
int CircularListLength(CLLNode headNode){
        int length = 0;
        CLLNode currentNode = headNode;
        while(currentNode != null){
                length++;
                currentNode = currentNode.getNext();
                if(currentNode == headNode)
                        break;
        }
        return length;
}
```

时间复杂度为 $O(n)$，用于扫描长度为 $n$ 的链表。

空间复杂度为 $O(1)$，用于创建一个临时变量。

**2. 输出循环链表的内容**

假设循环链表可以通过表头结点进行访问。由于所有的结点采用循环方式排列，所以链表的表头结点的前驱结点就是表尾结点。假设要输出从表头结点开始的结点的内容。输出结点内容，移动到下一个结点，继续输出直至再次到达表头结点。




```
void PrintCircularListData(CLLNode headNode){
        CLLNode CLLNode = headNode;
        while(CLLNode != null){
                System.out.print(CLLNode.getData()+"->");
                CLLNode = CLLNode.getNext();
                if(CLLNode == headNode) break;
        }
        System.out.println("(" + CLLNode.getData() + ")headNode");
}
```

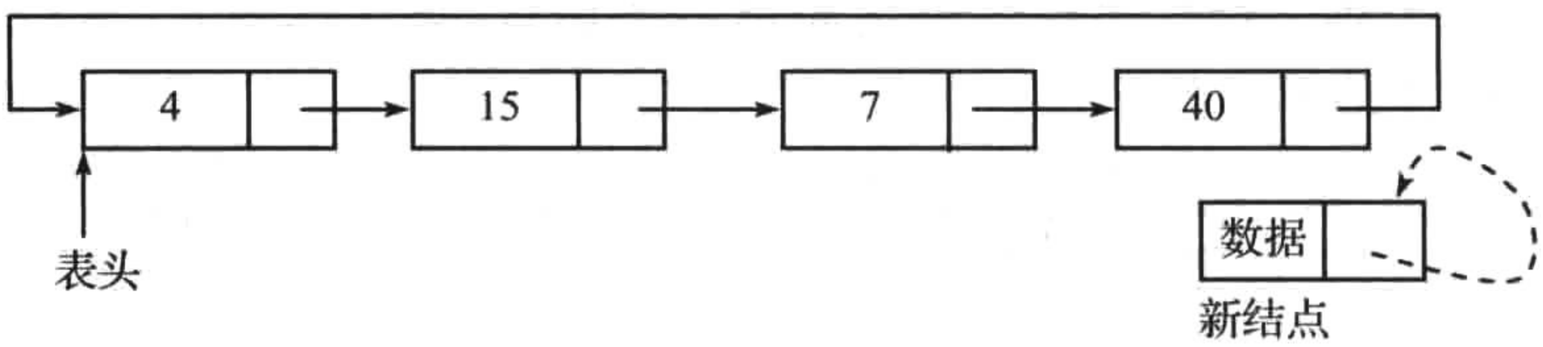
时间复杂度为 $O(n)$，用于扫描大小为 $n$ 的链表。

空间复杂度为 $O(1)$，用于创建一个临时变量。

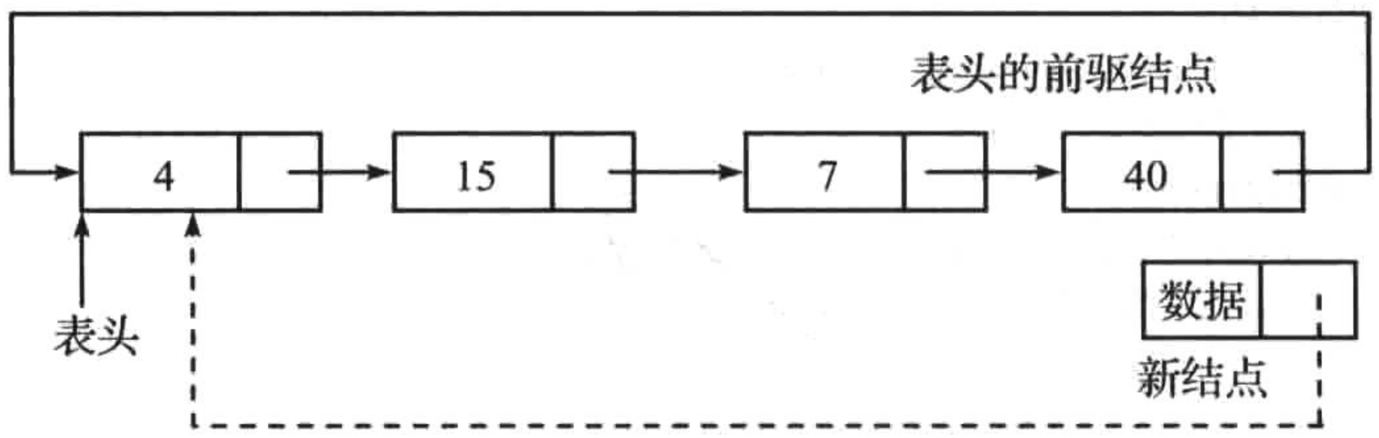
**3. 在循环链表的表尾插入结点**

在由表头开始的循环链表的表尾插入一个包含数据(data)的结点。新结点将放在表尾结点(即循环链表的最后一个结点)的后面，也就是说，在表尾结点和第一个结点之间插入该新结点。

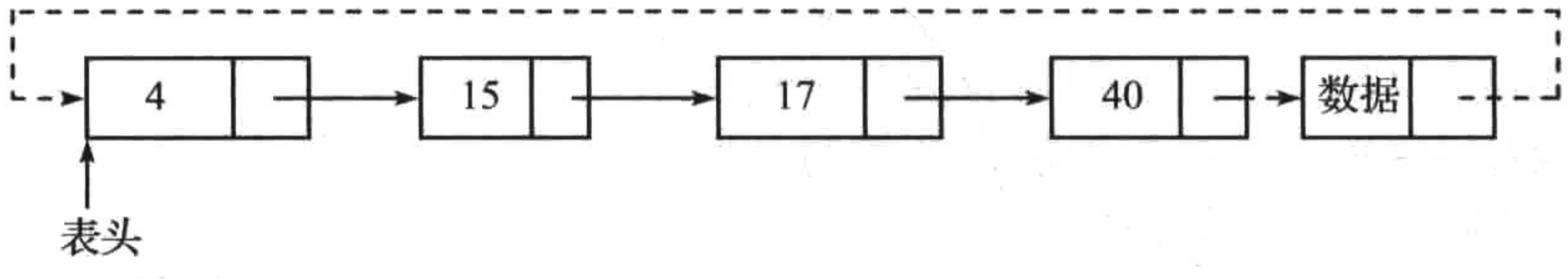
- 创建一个新结点，并且初始化其 next 指针指向该结点自身。




- 更新新结点的 next 指针为表头结点，然后遍历循环链表直至表尾。即插入位置应为循环链表中下一个结点是表头结点的结点位置(该结点是表头的前驱结点)。




- 更新表头的前驱结点的 next 指针指向新结点，得到如下图所示的循环链表。




```
void InsertAtEndInCLL (CLLNode headNode, CLLNode nodeToInsert) {
        CLLNode currentNode = headNode;
        while (currentNode.getNext() != headNode) {
                currentNode.setNext(currentNode.getNext());
        }
        nodeToInsert.setNext(nodeToInsert);
        if(headNode == null)
                headNode = nodeToInsert;
        ese {
                nodeToInsert.setNext(headNode);
                currentNode.setNext(nodeToInsert);
        }
}
```

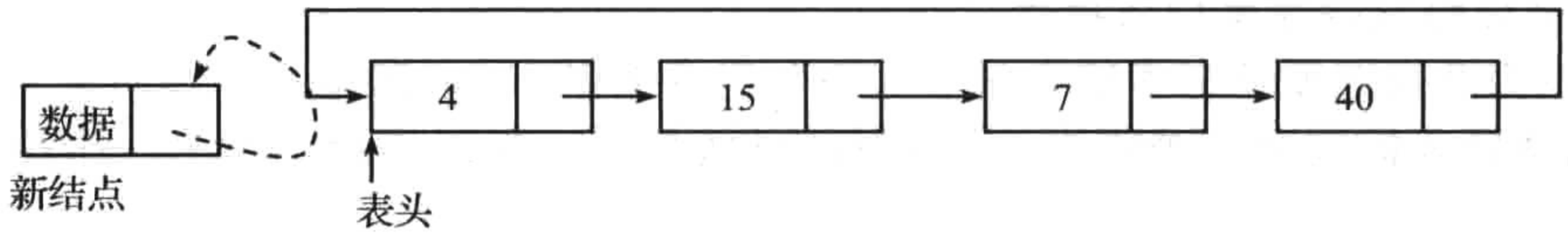
时间复杂度为 $O(n)$，用于扫描长度为 $n$ 的整个链表。

空间复杂度为 $O(1)$，用于创建一个临时变量。

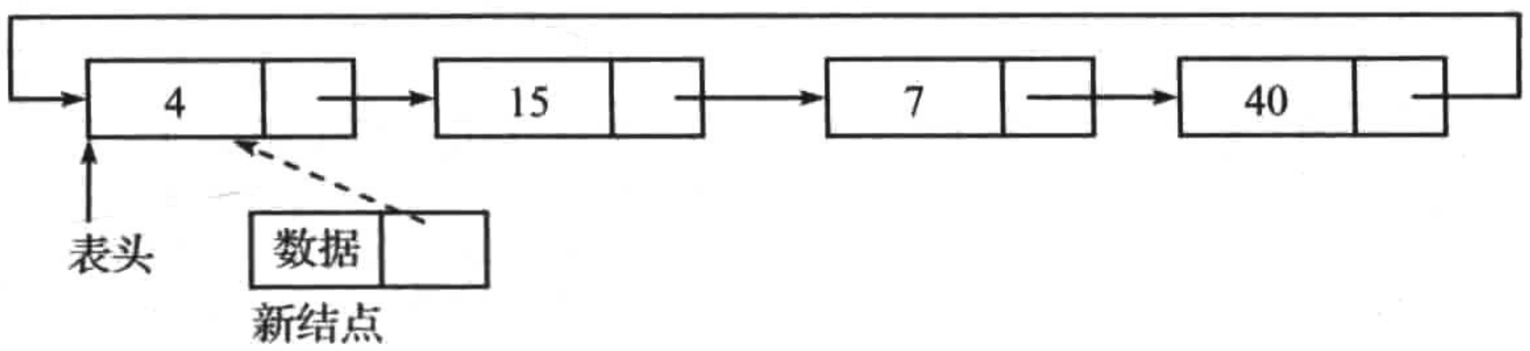
**4. 在循环链表的表头插入结点**

在循环链表的表头前插入结点与在表尾插入结点的唯一区别是，在插入新结点后，还需要更新指针。具体的步骤如下：

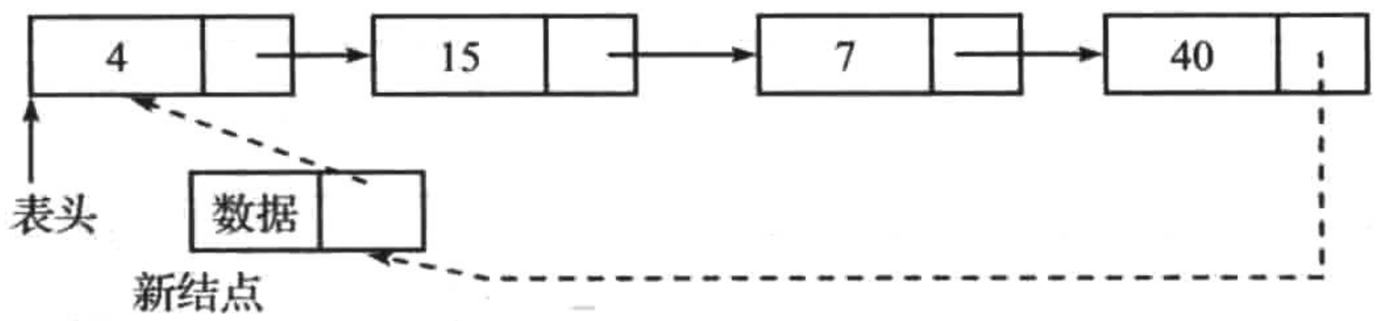
- 创建一个新结点，并且初始化其 next 指针指向结点自身。



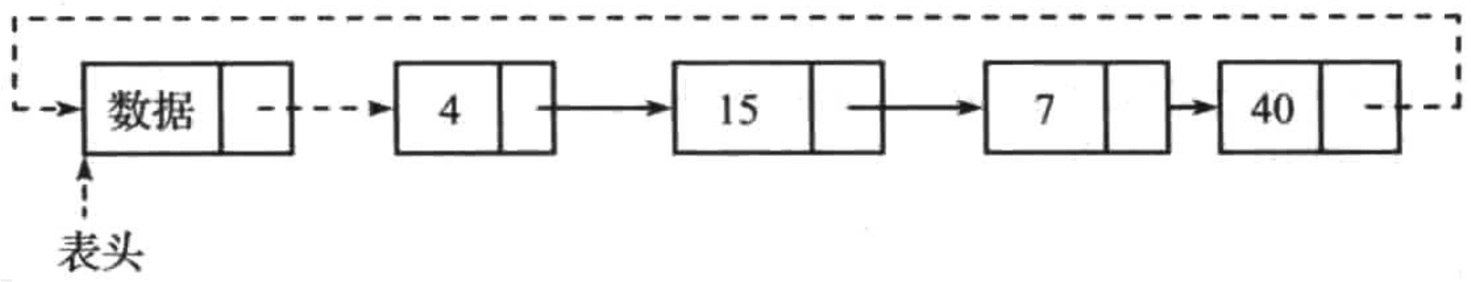
- 更新新结点的 next 指针为表头指针，然后遍历循环链表直至表尾。即插入位置应为循环链表中下一个结点是表头指针结点的结点位置(该结点是表头的前驱结点)。




- 更新链表中表头的前驱结点的 next 指针，使其指向新结点。




- 设置新结点为表头结点。




```
void InsertAtBeginInCLL (CLLNode headNode, CLLNode nodeToInsert) {
        CLLNode currentNode = headNode;
        while (currentNode.getNext() != headNode) {
                currentNode.setNext(currentNode.getNext());
        }
        nodeToInsert.setNext(nodeToInsert);
        if(headNode == null)
                headNode = nodeToInsert;
        else {
                nodeToInsert.setNext(headNode);
                currentNode.setNext(nodeToInsert);
                headNode = nodeToInsert;
        }
}
```

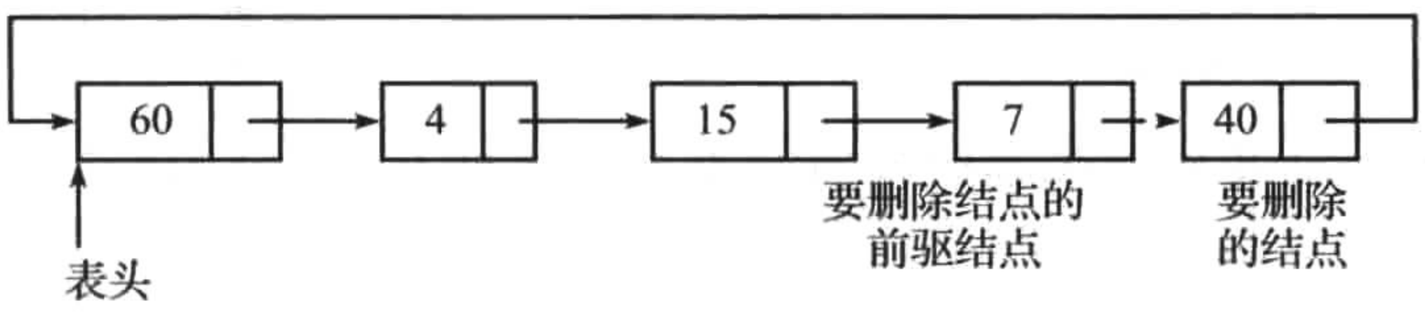
时间复杂度为 $O(n)$，用于扫描长度为 $n$ 的链表。

空间复杂度为 $O(1)$，用于创建一个临时变量。

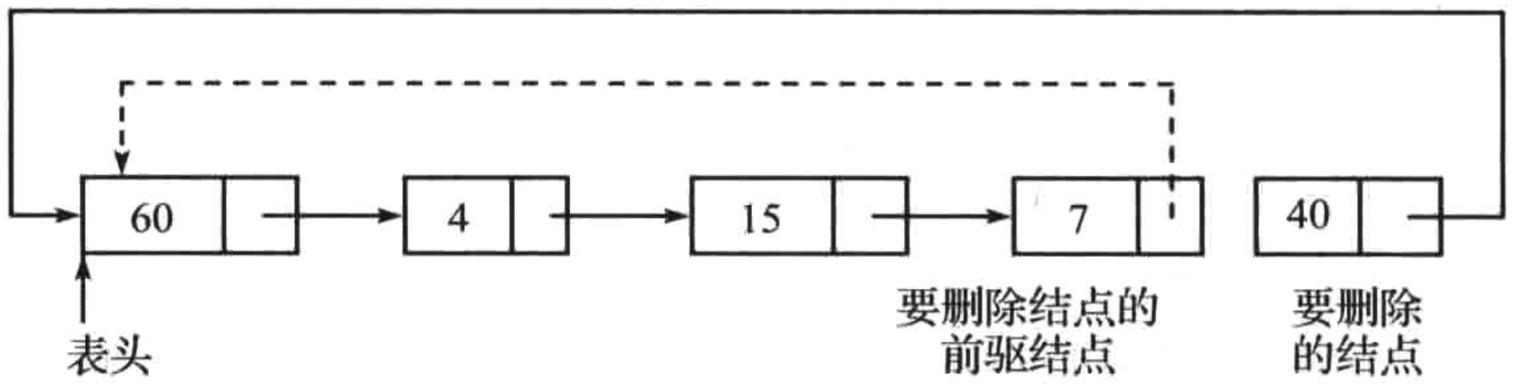
### 5. 删除循环链表中的最后一个结点

为了删除循环链表中的最后一个结点，需要遍历循环链表到倒数第二个结点。该结点将成为新的表尾结点，其 next 指针将指向链表的第一个结点。以下图的链表为例。为了删除最后一个结点 40，首先遍历链表到结点 7，再将结点 7 的 next 指针指向结点 60，并将这个结点重命名为表尾。

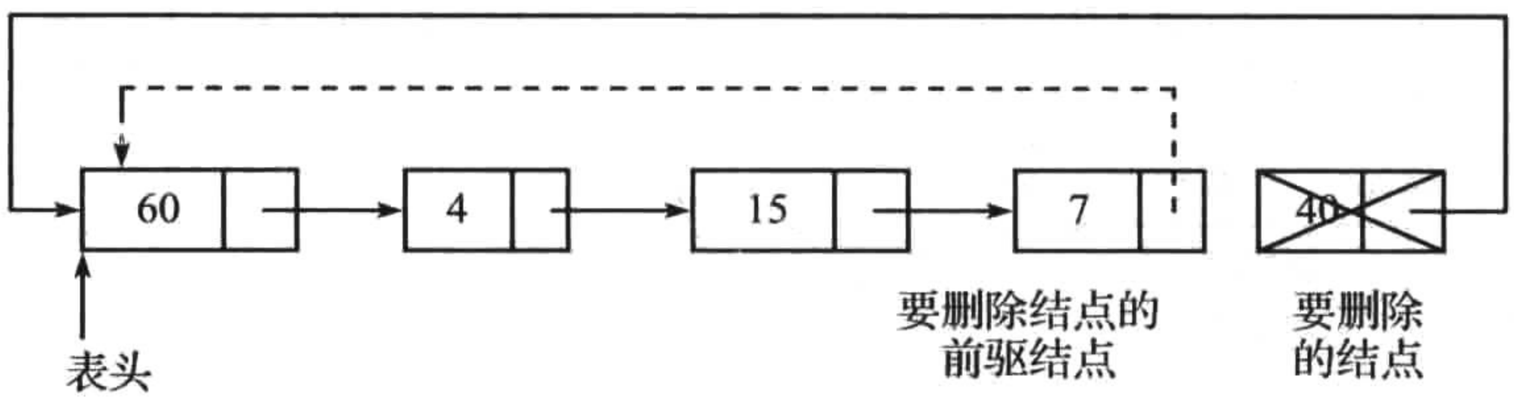
- 遍历循环链表，找到表尾结点及其前驱结点。



- 更新表尾结点的前驱结点的 next 指针，使其指向表头结点。




- 移除表尾结点。




```
void DeleteLastNodeFromCLL (CLLNode headNode) {
        CLLNode temp = headNode;
        CLLNode currentNode = headNode;
        if(head == null) {
                System.out.println("List Empty");
                return;
        }
        while (currentNode.getNext() != headNode) {
                temp = currentNode;
                currentNode = currentNode.getNext();
        }
        temp.setNext (headNode);
        currentNode = null;
        return;
}
```

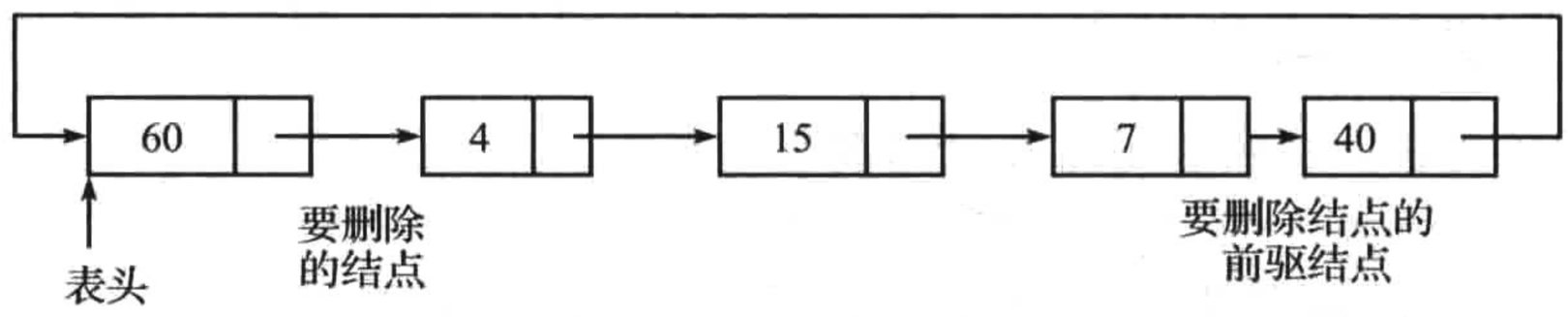
时间复杂度为 $O(n)$，用于扫描长度为 $n$ 的链表。

空间复杂度为 $O(1)$，用于创建一个临时变量。

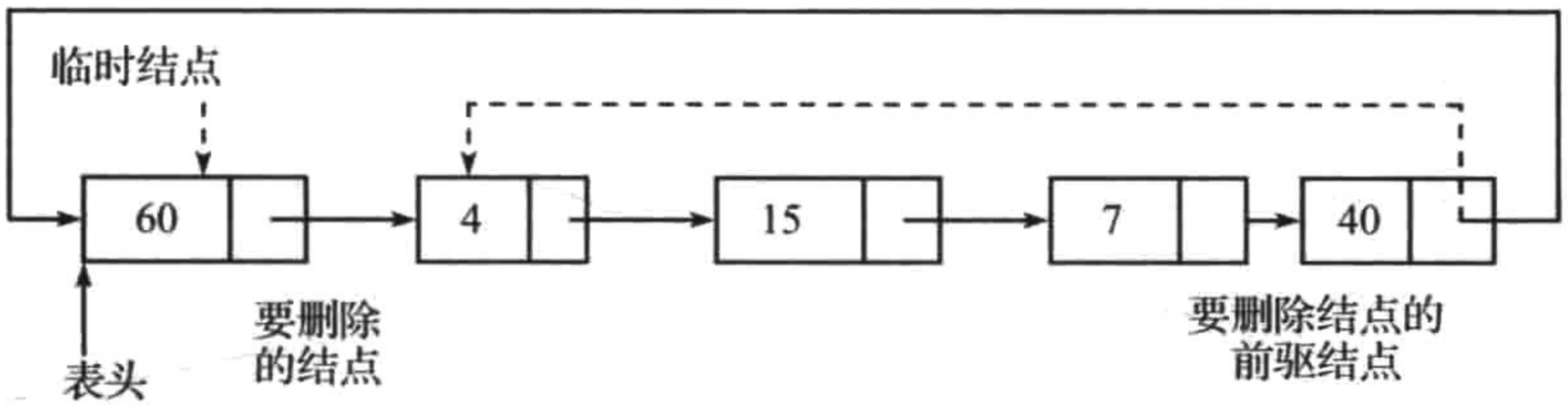
### 6. 删除循环链表中的第一个结点

删除循环链表中的第一个结点的操作很简单，只需将表尾结点的 next 指针指向第一个结点的后继(下一个)结点。

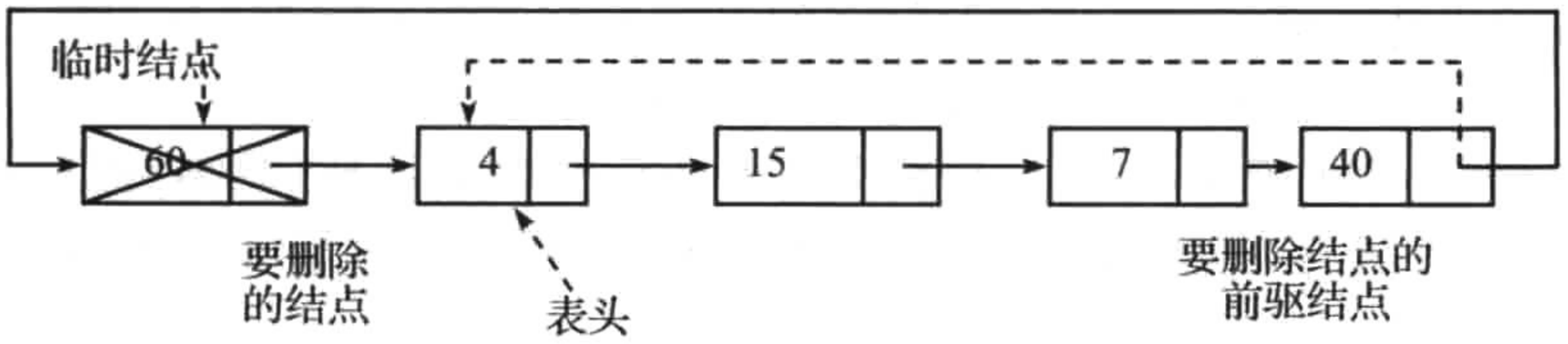
- 遍历循环链表找到表尾结点。表尾结点是将要删除的表头结点的前驱结点。




- 创建一个指向表头结点的临时结点。更新表尾结点的 next 指针，使其指向第一个结点的后继结点(如下图所示)。




- 修改表头指针的值，使其指向后继结点。移除临时结点(如下图所示)。




```
void DeleteFrontNodeFromCLL(CLLNode head) {
        CLLNode temp = head;
        CLLNode current = head;

        if(head == null) {
                System.out.println("List Empty");
                return;
        }
        while (current.getNext() != head)
                current.setNext(current.getNext());
        current.setNext(head.getNext());
        head = head.getNext();
        temp = null;
        return;
}
```

时间复杂度为 $O(n)$，用于扫描长度为 $n$ 的链表。

空间复杂度为 $O(1)$，用于创建临时变量。

**7. 循环链表的应用**

循环链表可用于管理计算机的计算资源，还可用于实现栈和队列。

## 3.9 一种存储高效的双向链表

在双向链表常规的实现中，需要一个指向后继结点的正向指针和一个指向前驱结点的反向指针。这表明双向链表中的结点由数据、一个指向后继结点的指针和一个指向前驱结点的指针构成。

**1. 常规的结点定义**

```
class DLLNode {
        private int data;
        private DLLNode next;
        private DLLNode previous;
        .....
}
```

最近，《Sinha》期刊发表了一个具有插入、遍历和删除操作的双向链表抽象数据类型(ADT)的一种实现方式。该方式基于指针的差运算。每个结点仅使用一个指针域来双向

地遍历链表。

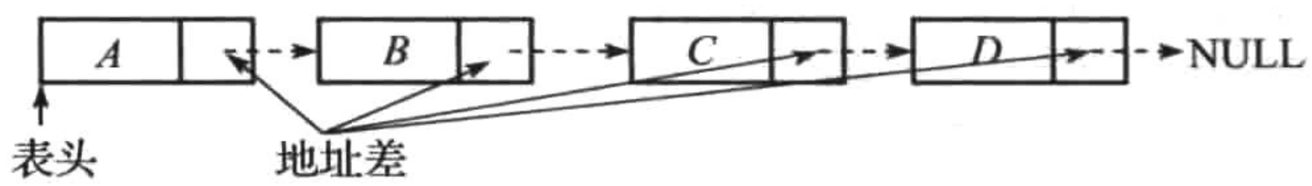
2. **新的结点定义**

```
public class ListNode {
    private int data;
    private ListNode ptrdiff;
    ……
}
```

ptrdiff 指针字段包含后继结点的地址与前驱结点的地址的差。指针的差通过异或运算(用⊕表示异或)来实现。

$$\text{ptrdiff} = \text{后继结点的地址} \oplus \text{前驱结点的地址}$$

表头结点的 ptrdiff 等于 NULL 与表头结点的后继结点的地址的异或。类似地，表尾结点的 ptrdiff 等于表尾结点的前驱结点的地址与 NULL 的异或。例如，考虑下面的链表。




在上面的图例中，

- $A$ 的 next 指针为：NULL $\oplus B$
- $B$ 的 next 指针为：$A \oplus C$
- $C$ 的 next 指针为：$B \oplus D$
- $D$ 的 next 指针为：$C \oplus$NULL

**为什么要这样做？**要回答这个问题，首先分析⊕操作的属性：

$$X \oplus X = 0$$

$$X \oplus 0 = X$$

$$X \oplus Y = Y \oplus X(\text{对称性})$$

$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)(\text{传递性})$$

对于上面的例子，假设想从结点 $C$ 移动到结点 $B$。已知结点 $C$ 的 ptrdiff 值为 $B \oplus D$。如果想移动到结点 $B$，将结点 $C$ 的 ptrdiff 值与结点 $D$ 的地址执行⊕运算，将得到结点 $B$ 的地址。这是因为，

$$(B \oplus D) \oplus D = B \quad (\text{因为 } D \oplus D = 0)$$

类似地，如果想移动到结点 $D$，将结点 $C$ 的 ptrdiff 值与结点 $B$ 的地址执行⊕运算，将得到结点 $D$ 的地址。

$$(B \oplus D) \oplus B = D \quad (\text{因为 } B \oplus B = 0)$$

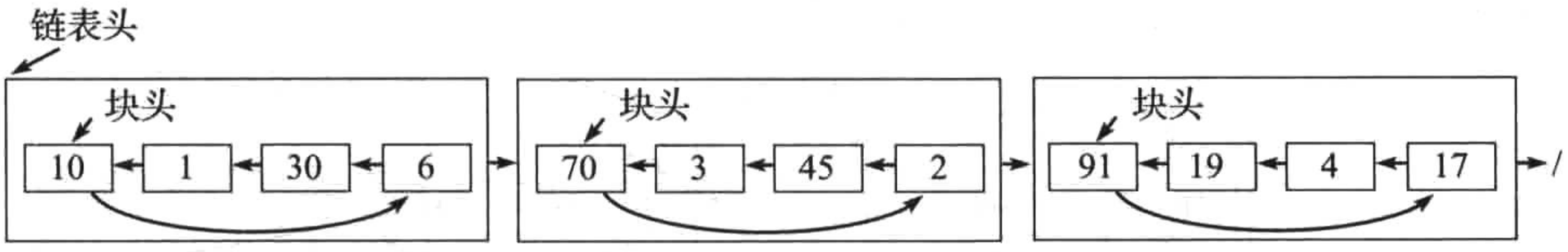
综上所述可以发现，仅使用一个指针就可以实现在双向链表中来回移动。这种存储高效的双向链表实现方式是可行的，且不需要太多的时间开销。

## 3.10 松散链表

与数组相比，链表的最大优势在于，在任何位置插入元素的时间开销仅为 $O(1)$。然而，在链表中查找某个元素的时间开销则是 $O(n)$。下面介绍一种单向链表的简单变种，称为*松散链表*。

松散链表中的每个结点存储多个元素(简称为块)。而每一块中的所有结点由循环链

表链接在一起。



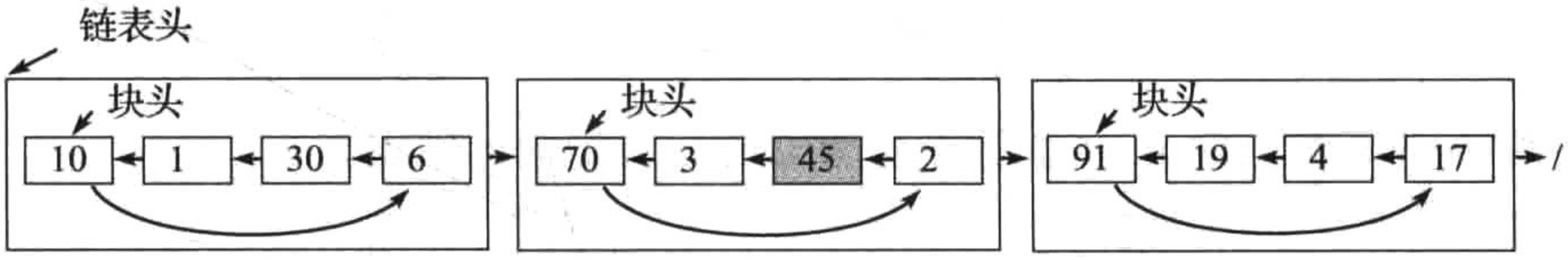

假设在任何时候松散链表中元素的个数不超过 $n$。为了进一步简化问题，假设除了最后一块外，所有块恰好含有$\lceil\sqrt{n}\rceil$个元素。所以，在任何时候，松散链表中块的个数不会超过$\lfloor\sqrt{n}\rfloor$。

**1. 在松散链表中查找一个元素**

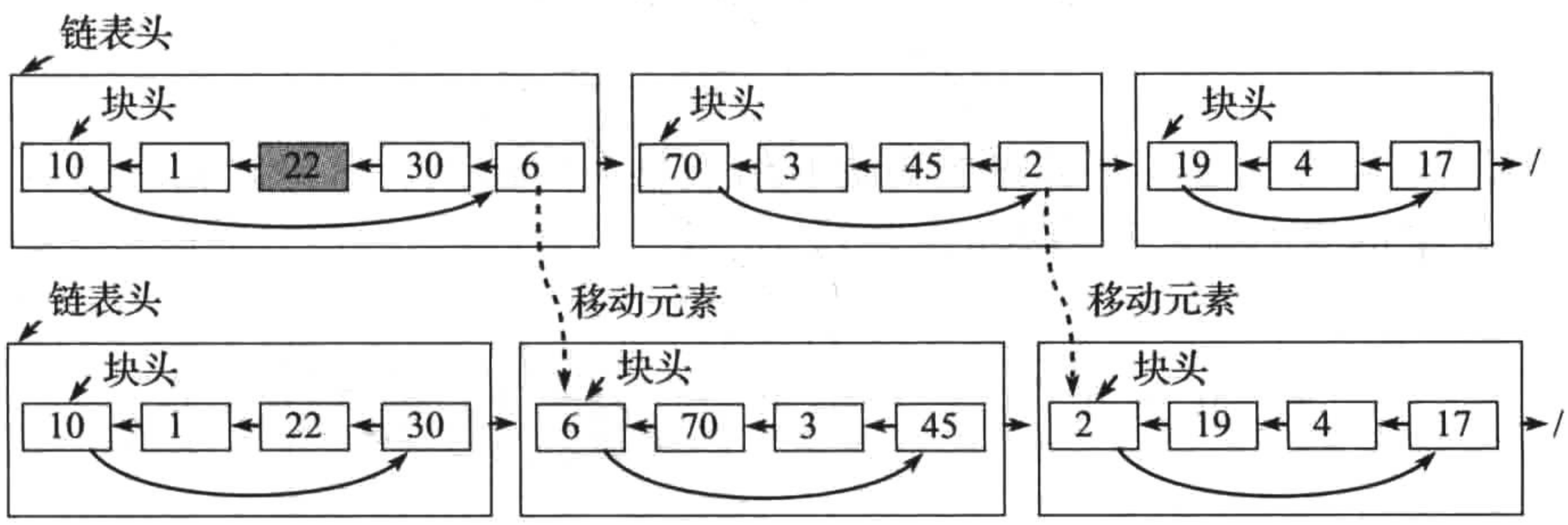
在松散链表中查找第 $k$ 个(位置的)结点的时间开销为 $O(\sqrt{n})$：

1) 遍历块链表找到包含第 $k$ 个结点的块，即第$\left\lceil\frac{k}{\lceil\sqrt{n}\rceil}\right\rceil$个块。因为遍历过程最多经过$\sqrt{n}$个块，所以其时间开销是 $O(\sqrt{n})$。

2) 在这个块的循环链表中找到第($k \bmod \lceil\sqrt{n}\rceil$)个结点。这个过程也需要 $O(\sqrt{n})$的时间开销，因为单个块中的结点数不大于$\lceil\sqrt{n}\rceil$。




**2. 在松散链表中插入一个元素**




当插入结点时，可能需要重新调整松散链表中的结点以保证前面提到的链表属性，即每个块包含$\lceil\sqrt{n}\rceil$个结点。假设要在第 $i$ 个结点的后面插入结点 $x$，且将 $x$ 放在第 $j$ 块中。

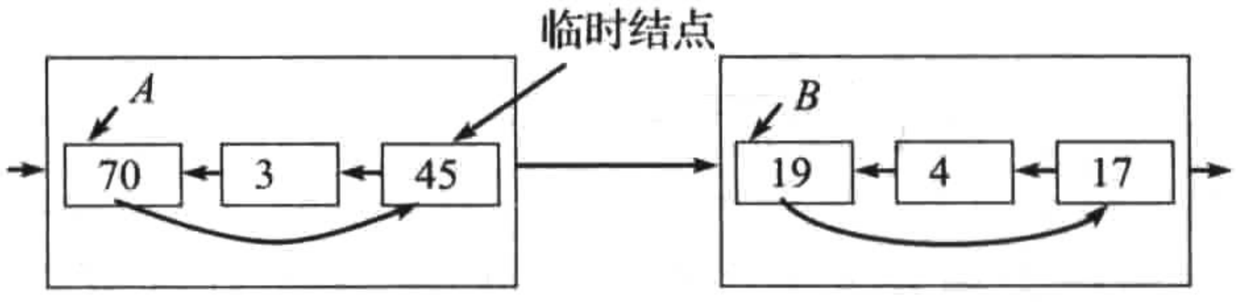
注意第 $j$ 块和第 $j$ 块以后的块中的结点都需要向表尾移动，以保证每块仍然只有$\lceil\sqrt{n}\rceil$个结点。此外，如果链表中最后一个块的空间不足，即已经超过$\lceil\sqrt{n}\rceil$个结点了，则还需在表尾添加一个新块。

**3. 执行移动操作**

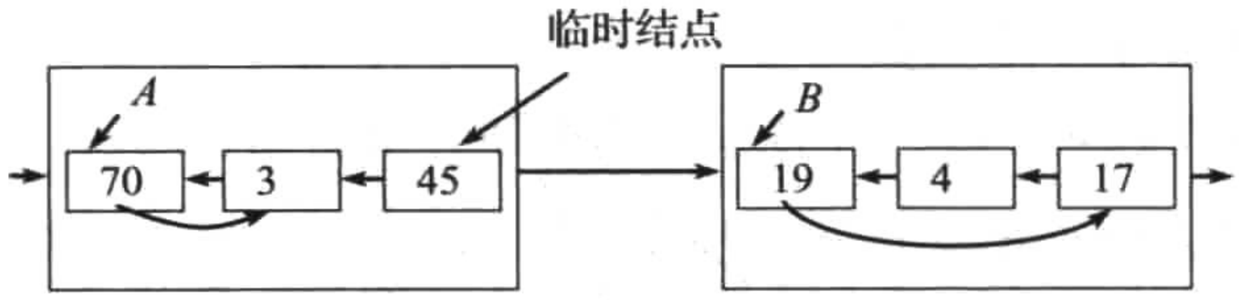
每次移动操作包括从块中的循环链表的表尾移除一个结点并在下一个块中的循环链表的表头插入一个结点，时间开销仅为 $O(1)$。所以，松散链表插入操作的总时间开销是

$O(\sqrt{n})$。因为最多有 $O(\sqrt{n})$个块，所以最多执行 $O(\sqrt{n})$次移动操作。

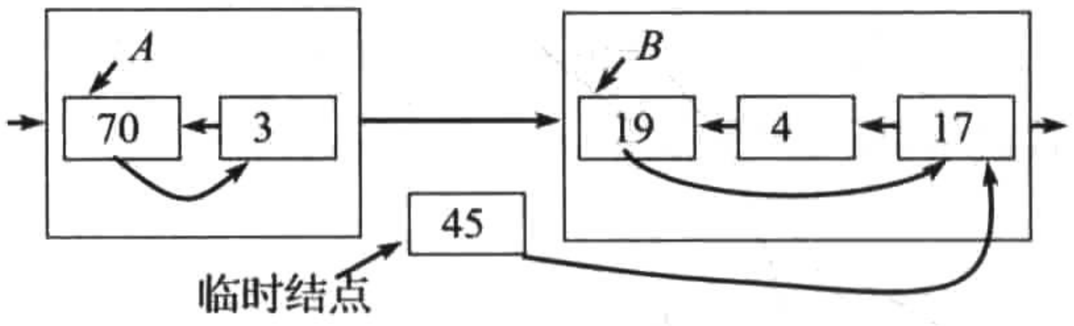
1）需要一个临时指针存储块 $A$ 的表尾结点。



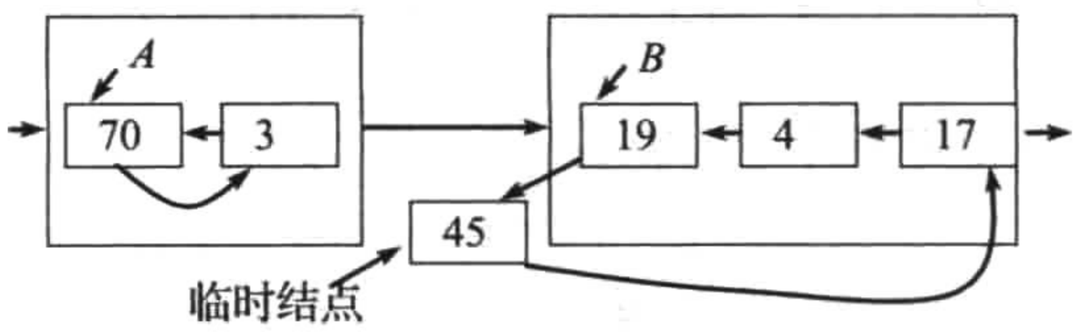

2）在块 $A$ 中，修改表头结点的 next 指针，使其指向块内链表的倒数第二个结点，以便删除块 $A$ 的表尾结点。



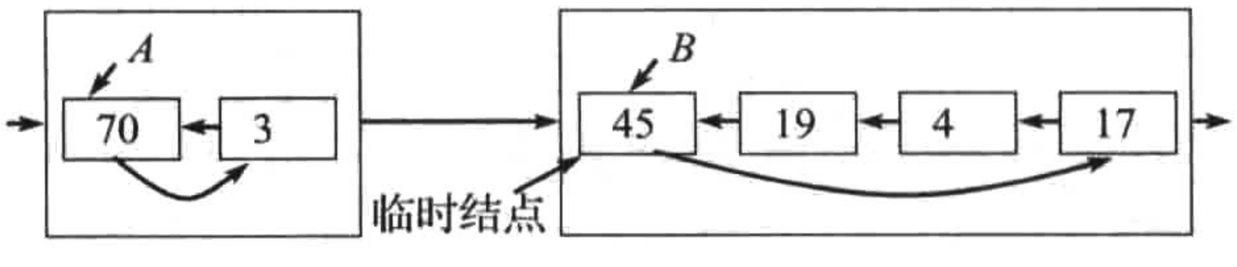

3）将被移动结点(块 $A$ 中原来的表尾结点)的 next 指针指向块 $B$ 的表尾结点。



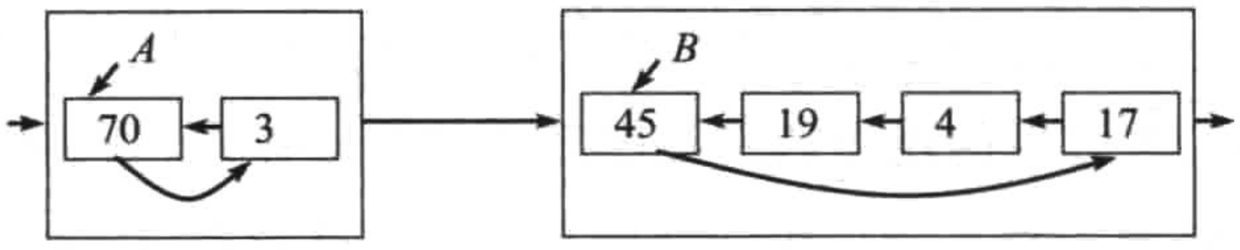

4）将块 $B$ 的头结点的 next 指针设置为临时结点指向的结点。




5）最后，将块 $B$ 的表头指针设置为临时结点指向的结点。这时，临时结点指向的结点就成为块 $B$ 中链表的新表头结点。




6）移除临时结点。整个移动操作完成，块 $A$ 中原来的表尾成为了块 $B$ 中的新表头。




**4. 性能**

松散链表主要有两个优点，一个是时间方面的，另一个是空间方面的。

首先，如果每块中元素个数适中(例如，最多一个缓存行的大小)，那么通过改善内存局部性，能够获得显著更优的缓存性能。

其次，因为在松散链表中链接的大小仅为 $O(n/m)$，其中 $n$ 等于松散链表中元素的个数，$m$ 等于一个块中能够存储的元素个数，所以这也将节省大量的空间。当每个元素的大小较小时，效果会更加明显。

## 3.11 链表的相关问题

**问题 1** 用链表实现栈。

**解答：** 参见第 4 章。

**问题 2** 找到链表的倒数第 $n$ 个结点。

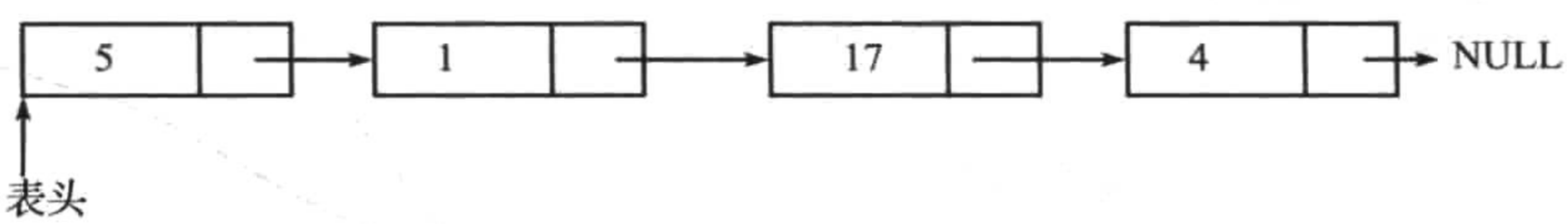
**解答：蛮力法：** 从链表的第一个结点开始，统计当前节点后面的结点个数。如果后面结点的个数小于 $n-1$，那么算法结束并返回消息“链表中的结点个数不足”。如果数量大于 $n-1$，则移动到下一个结点(作为新的当前结点)。重复该过程直至当前结点后面的结点个数等于 $n-1$，算法结束。

时间复杂度为 $O(n^2)$。对于每个结点，都需要从当前结点扫描一次链表中剩余的结点。

空间复杂度为 $O(1)$。

**问题 3** 针对问题 2，能否设计性能更优的方法？

**解答：可以。** 使用散列表。以下面的链表为例。




该方法需要新建一个散列表，表中的条目是<结点的位置，结点地址>。这说明散列表中每条记录的主键是结点在链表中的位置，值是该结点的地址。

| 链表中的位置 | 结点的地址 | 链表中的位置 | 结点的地址 |
|---|---|---|---|
| 1 | 结点 5 的地址 | 3 | 结点 17 的地址 |
| 2 | 结点 1 的地址 | 4 | 结点 4 的地址 |

为了创建散列表，当遍历链表时，可以得到链表的长度。令 $M$ 表示链表的长度，这样就将寻找链表的倒数第 $n$ 个结点的问题，转换为寻找链表正数第 $M-n+1$ 个结点。因为已知链表的长度，所以求解这个问题只需要返回散列表中主键为 $M-n+1$ 的值即可。

时间复杂度为 $O(m)$，主要是创建散列表的时间开销。空间复杂度为 $O(m)$，因为需要建立一个大小为 $m$ 的散列表。

**问题 4** 是否能在不创建散列表的情况下，用问题 3 中的方法解决问题 2？

**解答：可以。** 仔细观察问题 3 的解决方案，实际上就是求链表的长度。也就是说，该方案使用散列表来确定链表的长度。然而，只要从表头结点开始遍历链表，也能得到链表的长度。因此不用创建散列表同样可以求链表的长度。得到长度后，计算 $M-n+1$ 的值，然后从表头开始再遍历一次就能得到第 $M-n+1$ 个结点。这个方法需要两次遍历：一次用于确定链表的长度，另一次用于找到从表头开始的第 $M-n+1$ 个结点。

时间复杂度等于确定链表长度的时间开销加上从表头开始寻找第 $M-n+1$ 个结点的时间开销，所以 $T(n)=O(n)+O(n)\approx O(n)$。因为不需要建立散列表，所以空间复杂度为 $O(1)$。

**问题 5** 能否只用一次(链表)扫描就解决问题 2?

**解答：可以。有效方法为：**使用两个指针 pNthNode 和 pTemp。首先，两个指针都指向链表的表头结点。仅当 pTemp(沿着链表)进行了 $n$ 次移动后，pNthNode 才开始移动。然后两个指针同时移动直至 pTemp 到达表尾。这时 pNthNode 指针所指的结点就是所求的结点，也就是链表的倒数第 $n$ 个结点。

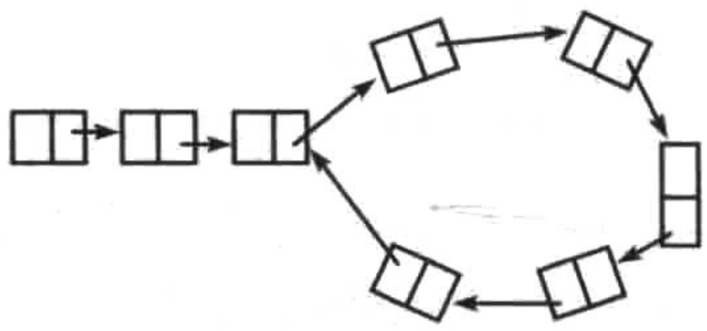
**注意：**任何时候，两个指针每次只向后移动一个结点。

```
ListNode  NthNodeFromEnd(ListNode head , int NthNode) {
      ListNode pTemp = head, pNthNode = null;
      for(int count =1; count< NthNode;count++) {
         if(pTemp != null)
            pTemp = pTemp.getNext();
      }
      while(pTemp!= null){
         if(pNthNode == null)
               pNthNode = head;
         else
               pNthNode = pNthNode.getNext();
         pTemp = pTemp.getNext();
      }
      if(pNthNode !=  null)
            return pNthNode;
      return null;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 6** 判定给定的链表是以 NULL 结尾，还是形成一个环。

**解答：蛮力法。**考虑下面的链表，其中包含一个环。这个链表与常规链表的区别在于，其中有两个结点的后继结点是相同的。在常规链表中是不存在环的，每个结点的后继结点是唯一的。换言之，若链表中出现(多个结点的)后继指针重复，就表明存在环。



一个简单直接的判定方法是，从第一个结点开始，令其为当前结点，然后看看链表中其他结点的后继指针是否指向当前结点。如果存在这样的结点，那么就可以判定链表中存在环。否则，对链表中其余结点重复上述过程。

**这个方法正确吗?** 根据该算法，需要不断地检查后继指针地址，但是如何确定链表的表尾呢? 否则算法将会出现死循环。

**注意：**如果从环中的某一结点开始，根据环的大小上述方法可能有效。

**问题 7** 是否能使用散列表技术求解问题 6?

**解答：可以。**使用散列表是可以求解上题的。

**算法：**

- 从表头结点开始，逐个遍历链表中的每个结点。
- 对于每个结点，检查该结点的地址是否存在于散列表中。
- 如果存在，则表明当前访问的结点已经被访问过了。出现这种情况只能是因为给定

链表中存在环。

- 如果散列表中没有当前结点的地址，那么把该地址插入散列表中。
- 重复上述过程，直至到达表尾或者找到环。

时间复杂度为 $O(n)$，用于扫描链表。注意，只需扫描一次给定的链表。空间复杂度为 $O(n)$，用于散列表的空间开销。

**问题 8** 能否使用排序技术求解问题 6?

**解答：不行**。首先给出如下基于排序的算法。然后，再分析这个算法失败的原因。

**算法：**

- 从表头结点开始，逐个遍历链表的每个结点，并把所有结点的后继指针的值保存在数组中。
- 对该(保存了后继指针值的)数组进行排序。
- 如果给定链表中存在环，那么根据定义，将有两个结点的后继指针指向相同的结点。
- 如果链表中存在环，那么在排序后后继指针值相同的结点将是相邻的。
- 如果出现了这样的结点对，那么可以判定链表中存在环。

时间复杂度为 $O(n\log n)$，用于对存储后继指针值的数组进行排序。空间复杂度为 $O(n)$，用于存储后继指针值数组。

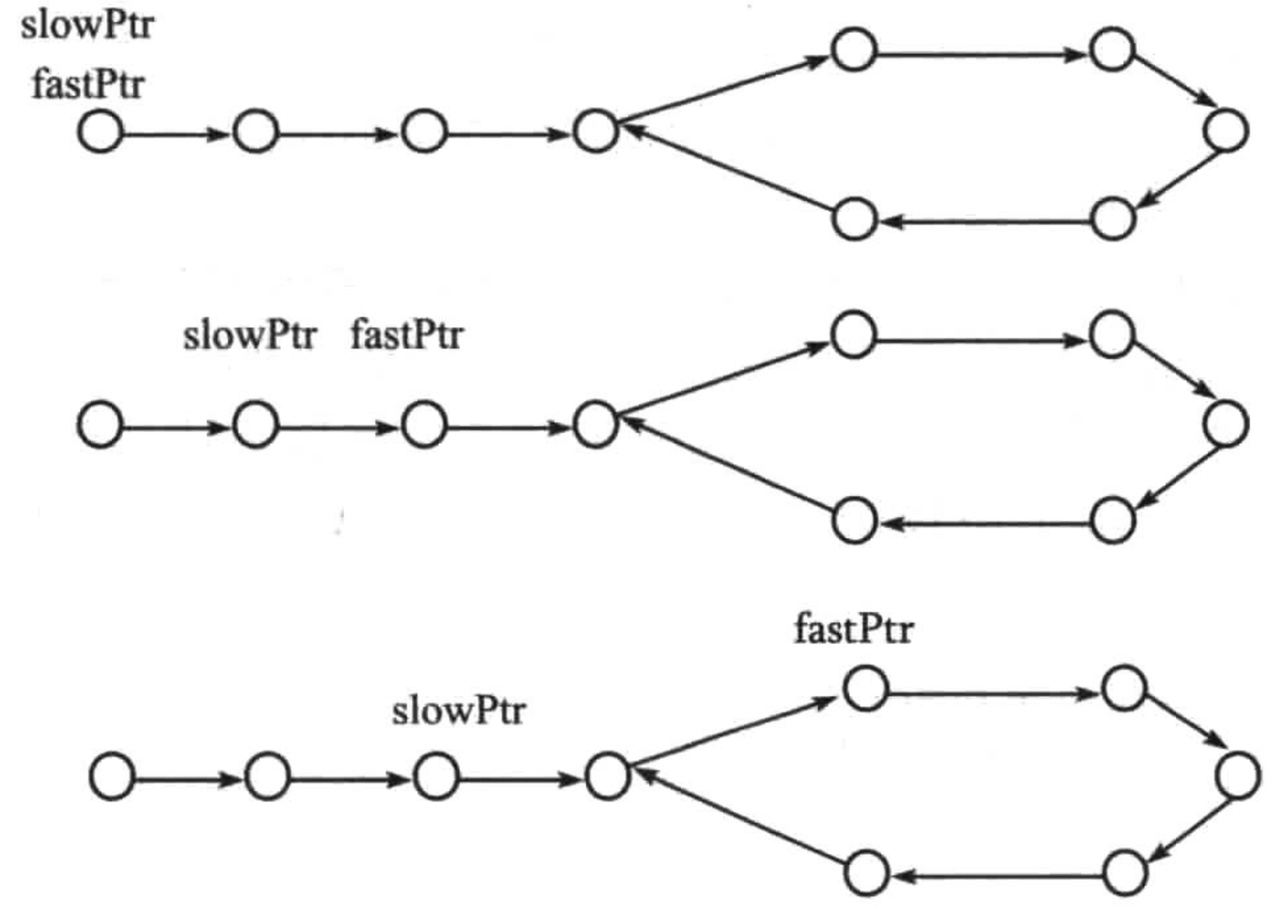
**上述算法有问题吗**? 只有当确定链表的长度时，上述算法才适用。否则，如果链表存在环，那么可能会出现死循环。因此这个算法是无效的。

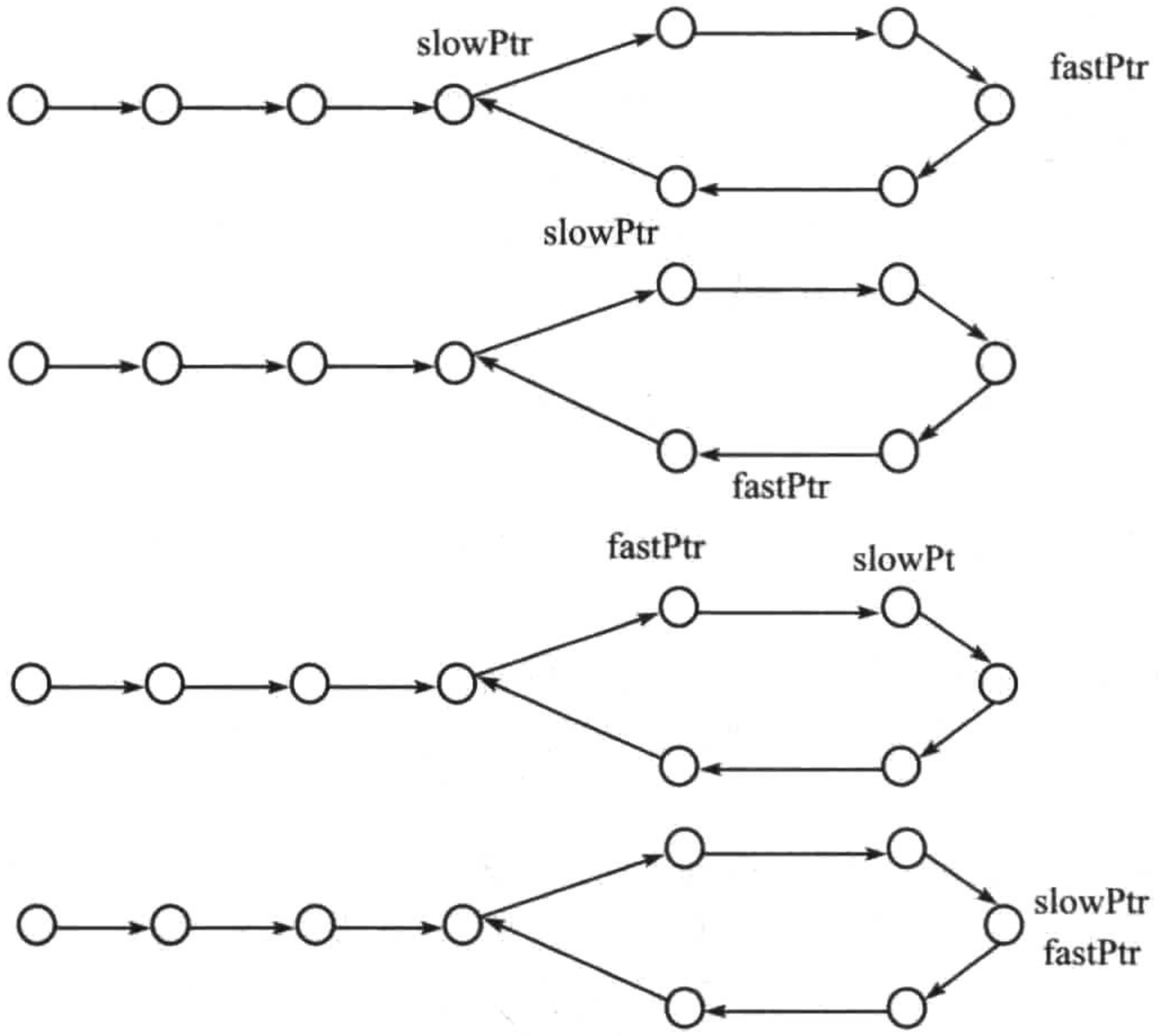
**问题 9** 是否存在时间复杂度为 $O(n)$的算法求解问题 6?

**解答：可以。有效的方法(内存开销更少的方法)**是由 Floyd 提出的，所以该方法称为 Floyd 环判定算法。该方法使用两个在链表中具有不同移动速度的指针。一旦它们进入环就会相遇，即表示存在环。这个判定方法的正确性在于，快速移动指针和慢速移动指针将会指向同一位置的唯一可能情况，就是整个或者部分链表是一个环。

设想一下，乌龟和兔子在一个轨道上赛跑。如果它们在一个环上赛跑，那么跑得快的兔子将赶上乌龟。下面的图例展示了 Floyd 算法的过程。从下图中可以发现，执行最后一步后，它们将在环中可能并非起点的某一点相遇。

**注意：**龟指针 slowPtr 每次后移 1 个结点。兔指针 fastPtr 每次后移 2 个结点。






```
boolean DoesLinkedListContainsLoop(ListNode head) {
   if (head == null )
         return false;
   ListNode slowPtr = head, fastPtr = head;
      while (fastPtr.getNext()!= null && fastPtr.getNext().getNext()!= null ) {
         slowPtr = slowPtr.getNext();
         fastPtr = fastPtr.getNext().getNext();
         if ( slowPtr == fastPtr )
            return true;
      }
      return false;
   }
```

时间复杂度为 $O(n)$。

空间复杂度为 $O(1)$。

**问题 10** 已知链表 $L$ 第一个元素的地址。$L$ 的表尾有两种可能，一种是正常结束(蛇形结构)；另一种是最后的元素指向链表中某个较早出现的元素(蜗牛形结构)。请设计一个算法，测试一个给定的链表 $L$ 是蛇形结构还是蜗牛形结构。

**解答：** 求解方法与问题 6 相同。

**问题 11** 判定给定的链表是否以 NULL 结束。如果链表中存在环，找到环的起始结点。

**解答：** 这个题目的求解方法需要对求解问题 9 的方法进行扩展。在找到链表中的环后，初始化 slowPtr 的值，使其指向链表的表头结点。然后，slowPtr 和 fastPtr 从各自的位置开始沿着链表移动，每次均移动一个结点。它们相遇的位置就是环的开始结点。通常可用这种方法来删除环。

```
int FindBeginofLoop (ListNode head) {
   ListNode slowPtr = head, fastPtr = head;
   boolean loopExists = false;
   if (head == null )
      return false;
```

```
while(fastPtr.getNext()!= null && fastPtr.getNext().getNext() != null) {
    slowPtr = slowPtr.next;
    fastPtr = fastPtr.getNext().getNext();
    if ( slowPtr == fastPtr ) {
        loopExists = true;
        break;
    }
}
if(loopExists) { // 环存在
        slowPtr = head;
        while(slowPtr != fastPtr) {
            fastPtr = fastPtr.getNext();
            slowPtr = slowPtr.getNext();
        }
        return slowPtr; // 返回环的开始结点
    }
    return null; // 环不存在
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 12**　在前面问题的求解方法中，一旦乌龟与兔子相遇就表示链表中存在环。然后，乌龟从表头开始移动，而兔子从相遇位置开始移动，乌龟和兔子每次都移动一个结点，当乌龟与兔子再次相遇时，如何证明它们一定相遇在环的起始结点呢？

**解答：**这个问题是数论的核心。在 Floyd 环判定算法中，可发现当乌龟与兔子移动距离(结点数)等于 $n \times L$(其中 $L$ 是环的长度)时，两者将相遇。此外，由于它们的移动方式，可以确定乌龟一定位于兔子和链表表头位置的中点。因此乌龟与链表表头位置之间的距离也是 $n \times L$。

它们分别从乌龟所在的位置和链表表头位置开始移动，每次移动一步，可知一旦它们都进入环，将马上相遇，因为它们的间距是环长度的倍数——$n \times L$。由于它们中有一个已经在环中，所以只需要单步移动另外一个直到它也进入环，并始终保持两者相距 $n \times L$，那么再次相遇的结点一定是环的起始结点。

**问题 13**　在 Floyd 环判定算法中，如果两个指针每次分别移动 2 个结点和 3 个结点，而不是移动 1 个结点和 2 个结点，算法仍然有效吗？

**解答：可以**。但是通过探索一些例子，可以发现算法的复杂度可能增加。

**问题 14**　判定给定的链表是否以 NULL 结束。如果链表中存在环，返回环的长度。

**解答：**这个题目的求解方法需要对基本的环判定算法进行扩展。在找到链表中的环后，保持 slowPtr 指针不变，fastPtr 指针则继续移动。每次移动 fastPtr 指针时，计数器变量也加 1，直至再一次回到 slowPtr 指针所在的位置，就可求出环的长度。

```
int FindLoopLength(ListNode head) {
    ListNode slowPtr = head, fastPtr = head;
    boolean loopExists = false;
    int counter = 0;
    if (head == null )
        return 0;
    while (fastPtr.getNext()!= null && fastPtr.getNext().getNext() != null ) {
        slowPtr = slowPtr.next;
        fastPtr = fastPtr.getNext().getNext();
        if ( slowPtr == fastPtr ) {
            loopExists = true;
```

```
            break;
        }
    }
    if(loopExists) {
        fastPtr = fastPtr.getNext();
        while(slowPtr  != fastPtr) {
                fastPtr = fastPtr.getNext();
                counter++;
        }
        return counter;
    }
    return 0;                                    // 链表中没有环
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 15**　在有序链表中插入一个结点。

**解答：**遍历链表，找到存放元素的正确位置后，插入结点。

```
ListNode InsertInSortedList(ListNode head, ListNode newNode) {
        ListNode current = head;
        if(head == null) return newNode;
        // 遍历链表，直至找到比新结点中数据值更大的结点
        while (current != null && current.getData() <  newNode.getData()){
                temp = current;
                current = current.getNext();
          }
        // 在该结点前插入新结点
        newNode.setNext(current);
        temp.setNext(newNode);
        return head;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

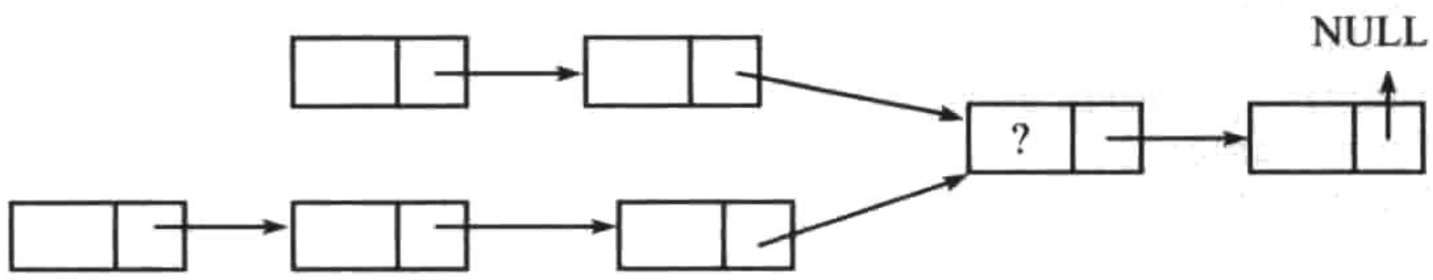
**问题 16**　逆置单向链表。

**解答：**

```
// 迭代版本
ListNode ReverseList(ListNode head ){
        ListNode temp = null, nextNode = null;
        while ( head != null ) {
                nextNode = head.getNext();
                head.setNext(temp);
                temp = head;
                head = nextNode;
        }
        return temp;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 17**　假设两个单向链表在某个(结)点相交后，成为一个单向链表。两个链表的表头结点是已知的，但是相交的结点未知。也就是说，它们相交之前各自的结点数是未知的，并且两个链表的结点数也可能不同。假设链表 List1 和链表 List2 在相交前的结点数分别为 $n$ 和 $m$，那么 $m$ 可能等于或小于 $n$，也可能大于 $n$。请设计算法找到两个链表的合并点。



**解答：蛮力法：**一种简单方法是把第一个链表中的每一个结点指针与第二个链表中的每一个结点指针比较，当出现相等的结点指针时，即为相交结点。但是，这种方法的时间复杂度 $O(mn)$ 较高。

时间复杂度为 $O(mn)$。空间复杂度为 $O(1)$。

**问题 18** 可以使用排序技术求解问题 17 吗？

**解答：不行。**首先给出下面基于排序的算法。然后，再分析这个算法失败的原因。

**算法：**

- 获得第一个链表各个结点指针并保存在数组中，然后排序。
- 获得第二个链表各个结点指针并保存在另一个数组中，然后排序。
- 排序后，定义两个数组的下标，一个用于第一个排序的数组，另一个用于第二个排序的数组。
- 比较两个数组下标的对应值，值小的下标加 1(只有值不相等时才增加数组下标)。
- 一旦发现两个数组下标的对应值相等时，就表明存在两个指向同一个结点的结点，返回该结点。

时间复杂度为排序的时间开销加上(用于比较的)扫描的时间开销，$O(m\log m)+O(n\log n)+O(m+n)$。需要考虑三者中产生最大值的情况。空间复杂度为 $O(m+n)$。

**上述算法有问题吗？**有问题。在该算法中，对两个链表的所有结点指针进行排序。但是，不要忘记，这里有很多重复的元素。因为两个链表在相交后的所有结点的指针是相同的。上述算法只在一种情况下是正确的，即两个链表在表尾相交。

**问题 19** 可以使用散列表技术求解问题 17 吗？

**解答：可以。**

**算法：**

- 选择结点较少的链表(如果链表的长度是未知的，那么随便选择一个链表)，将其所有结点的指针值保存在散列表中。
- 遍历另一个链表，对于该链表中的每一个结点，检查散列表中是否已经保存了其结点指针。
- 如果两个链表存在合并点，那么必定会在散列表中找到记录(结点指针)。

时间复杂度为：第一个链表创建散列表的时间开销加上扫描第二个链表的时间开销，等于 $O(m)+O(n)$(或 $O(n)+O(m)$)，取决于选择哪个链表来建立散列表)。这两种情况具有的算法时间复杂度是相同的。空间复杂度为 $O(n)$ 或 $O(m)$。

**问题 20** 可以使用栈求解问题 17 吗？

**解答：可以。**

**算法：**

- 创建两个栈：一个用于第一个链表，另一个用于第二个链表。
- 遍历第一个链表，把所有结点地址压入第一个栈。
- 遍历第二个链表，把所有结点地址压入第二个栈。
- 这时，两个栈分别包含了对应链表的结点地址。
- 比较两个栈的栈顶元素(结点地址)。
- 如果两者相等，那么弹出两个栈的栈顶元素并保存在临时变量中(因为两个地址是同一个结点，所以只需要一个临时变量)。

- 继续上述过程，直至两个栈的栈顶元素不相等。
- 这就表示找到了两个链表的合并点。
- 返回临时变量的值。

时间复杂度为 $O(m+n)$，用于扫描两个链表。空间复杂度为 $O(m+n)$，用于为两个链表分别建立栈。

**问题 21** 有其他方法求解问题 17 吗？

**解答：有**。使用在数组中“查找第一个重复数”方法(具体算法参见第 11 章)。

**算法：**

- 创建一个数组 $A$，在数组中保存两个链表中所有结点的后继指针。
- 在数组 $A$ 中查找第一个重复元素(具体算法参见第 11 章)。
- 第一个重复元素即为两个链表的合并点。

时间复杂度为 $O(m+n)$。空间复杂度为 $O(m+n)$。

**问题 22** 是否存在求解问题 17 的其他方法？

**解答：是**。结合使用排序和搜索技术，可以设计复杂度更低的方法。

**算法：**

- 创建一个数组 $A$，在数组中保存第一个链表中所有结点的后继指针。
- 对数组 $A$ 进行排序。
- 然后，对于第二个链表中的每一个结点(的后继指针)，在排序数组中搜索(假设使用时间复杂度为 $O(\log n)$的折半查找方法)。
- 由于是逐个扫描第二个链表中的结点，所以出现在数组中的第一个重复元素一定是合并点。

时间复杂度为排序时间开销加上搜索时间开销，等于 $O(\max(m\log m, n\log n))$。空间复杂度为 $O(\max(m, n))$。

**问题 23** 能设计时间复杂度更低的方法求解问题 17 吗？

**解答：可以。**

**算法：**

- 获得两个链表(L1 和 L2)的长度——$O(n)+O(m)=O(\max(m, n))$。
- 计算两个长度的差 $d$——$O(1)$。
- 从较长链表的表头开始，移动 $d$ 步——$O(d)$。
- 在两个链表中开始同时移动，直至出现两个后继指针值相等的情况 $O(\min(m, n))$。
- 时间复杂度为 $O(\max(m, n))$。
- 空间复杂度为 $O(1)$。

```
ListNode FindIntersectingNode(ListNode list1, ListNode list2) {
    int L1=0, L2=0, diff=0;
    ListNode head1 = list1, head2 = list2;
    while(head1 !=  null) {
            L1++;
            head1 = head1.getNext();
    }
    while(head2 !=  null) {
            L2++;
            head2 = head2.getNext();
    }
```

```
if(L1 < L2) {
        head1 = list2;
        head2 = list1;
        diff = L2 - L1;
} else{
        head1 = list1;
        head2 = list2;
        diff = L1 – L2;
}
for(int i = 0; i < diff; i++)
        head1 = head1.getNext();
    while(head1 !=  null && head2 != null) {
        if(head1 == head2)
            return head1.getData();
        head1= head1.getNext();
        head2= head2.getNext();
    }
    return null;
}
```

**问题 24** 如何找到链表的中间结点？

**解答：蛮力法**。在链表中对每个结点统计其后结点的个数，然后判定其是否是为中间结点。

时间复杂度为 $O(n^2)$。空间复杂度为 $O(1)$。

**问题 25** 能否设计时间复杂度更低的方法来求解问题 24？

**解答：可以**。

**算法：**

- 遍历链表，得到链表的长度，记为 $n$。
- 从表头开始再次扫描链表，定位到第 $n/2$ 个结点，即为中间结点。

时间复杂度为求链表长度的时间开销加上定位中间结点的时间开销，等于 $O(n)+O(n)\approx O(n)$。空间复杂度为 $O(1)$。

**问题 26** 可以使用散列表技术求解问题 24 吗？

**解答：可以**。理由和求解问题 3 时相同。

时间复杂度 $T(n)=O(n)$，用于创建散列表。空间复杂度为 $O(n)$，因为需要创建一个大小为 $n$ 的散列表。

**问题 27** 能否只用一次扫描来求解问题 24？

**解答：可以**。使用两个指针。让第一个指针的移动速度是另一个的两倍。当第一个指针到达表尾时，另一个指针则指向中间结点。

**注意：如果链表结点数为奇数，则第$(n/2)$个结点为中间结点。**

```
ListNode FindMiddle(ListNode head) {
        ListNode ptr1x, ptr2x;
        ptr1x = ptr2x = head;
        int i=0;
        // 不断循环，直至达到表尾 (next后继指针为NULL，就表示达到最后一个结点)
        while(ptr1x.getNext() != null) {
            if(i == 0) {
                ptr1x = ptr1x.getNext(); // 只后移第一个指针
                i=1;
            }
```

```
            else if( i == 1) {
                    ptr1x = ptr1x.getNext(); // 两个指针都后移
                    ptr2x = ptr2x.getNext();
                    i = 0;
            }
        }
        return ptr2x;       // 返回ptr2x的值，即为中间结点
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 28** 如何从表尾开始输出链表？

**解答：** 递归遍历至表尾。当返回时，输出元素。

```
// 下面的函数将从表尾开始输出链表
void PrintListFromEnd(ListNode head) {
        if(head == null)
            return;
        PrintListFromEnd(head.getNext());
        System.out.println(head.getData());
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于栈空间。

**问题 29** 检查链表的长度是奇数还是偶数？

**解答：** 使用一个在链表中每次向后移动两个结点的指针。最后，如果指针值为 NULL，那么链表长度为偶数，否则指针指向表尾结点，链表长度为奇数。

```
int IsLinkedListLengthEven(ListNode listHead) {
    while(listHead != null && listHead.getNext() != null)
        listHead = listHead.getNext().getNext();
    if(listHead == null) return 0;
    return 1;
}
```

时间复杂度为 $O(n/2) \approx O(n)$。空间复杂度为 $O(1)$。

**问题 30** 如果当前链表的表头指针指向第 $k$ 个元素，那么如何获取第 $k$ 个元素前的元素？

**解答：** 使用存储高效的链表(异或链表)。

**问题 31** 如何把两个有序链表合并为一个新的有序链表？

**解答：**

```
ListNode MergeList(ListNode a, ListNode b) {
    ListNode result = null;
    if(a==null) return b;
    if(b==null) return a;
    if(a.getData() <= b.getData()) {
        result =a;
        result.setNext(MergeList(a.getNext(), b));
    }
    else {
        result =b;
        result.setNext(MergeList(b.getNext(),a));
    }
    return result;
}
```

时间复杂度为 $O(n)$。

**问题 32** 如何逐对逆置链表？例如，假设初始链表为 1→2→3→4→$X$，那么经过逐对逆置后，新链表应该变为 2→1→4→3→$X$。

**解答：**

```
// 递归版本
ListNode ReversePairRecursive(ListNode head) {
        ListNode temp;
        if(head ==NULL || head.next ==NULL)
                return;  // 递归的基本情形为当前链表为空或只有一个元素
        else {
                //逆置第一对

                temp = head.next;
                head.next = temp.next;
                temp.next = head;
                head = temp;
                // 链表余下的部分继续递归地调用该函数
                head.next.next = ReversePairRecursive(head.next.next);
                return head;
        }
}
/*迭代版本*/
ListNode  ReversePairIterative(ListNode  head) {
    ListNode  temp1 = null;
    ListNode  temp2 = null;
    while (head != null && head.next != null) {
        if (temp1 != null) {
            temp1.next.next = head.next;
        }
        temp1 = head.next;
        head.next = head.next.next;
        temp1.next = head;
        if (temp2 == null)
            temp2 = temp1;
        head = head.next;
    }
    return temp2;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**问题 33** 如何把给定的二叉树转换为双向链表？

**解答：** 参见第 6 章。

**问题 34** 如何对链表进行排序？

**解答：** 参见第 10 章。

**问题 35** 如果要把两个链表连接起来，下面哪种方法的时间复杂度为 $O(1)$？

1）单向链表　　2）双向链表　　3）循环双向链表

**解答：** 循环双向链表。这是因为如果采用单向链表或双向链表，都需要遍历第一个链表直至表尾，然后再添加第二个链表。而对于循环双向链表，则不需要遍历链表。

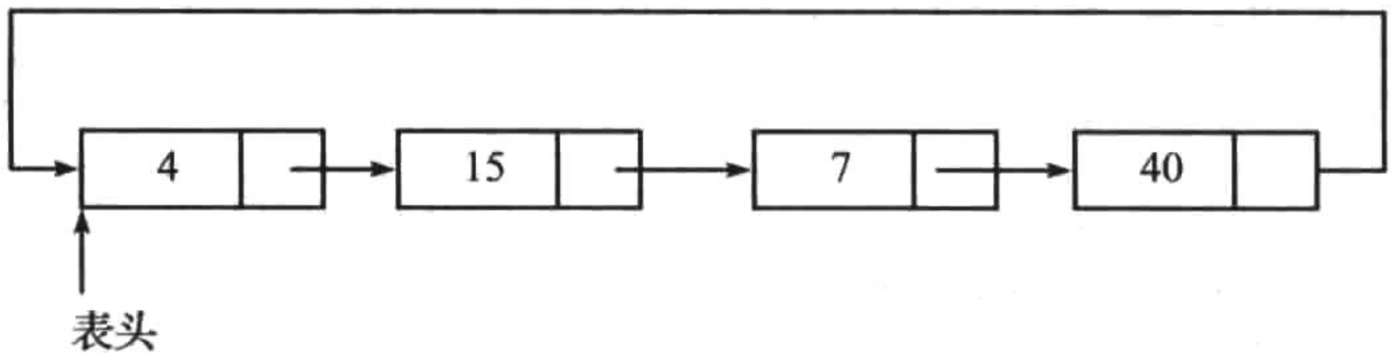
**问题 36** 如何把一个循环链表分割成两个长度相等的部分？如果链表的结点数是奇数，那么让第一个链表的结点数比第二个多 1 个。

**解答：**

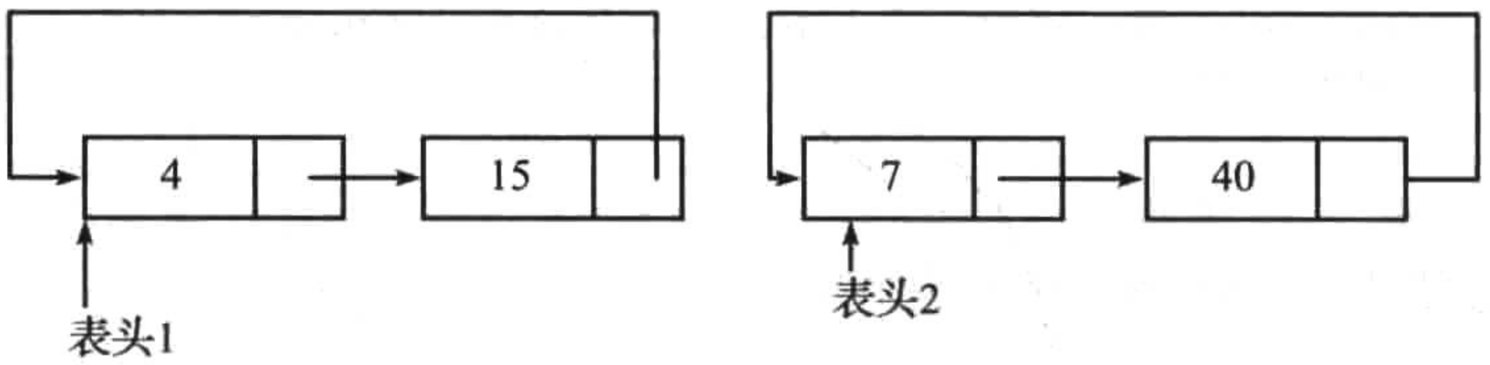
**算法：**

- 使用 Floyd 环判定算法寻找并存储循环链表的中间结点和最后一个结点指针。
- 把后半部分变成环。
- 把前半部分变成环。
- 设置两个链表的表头(head)指针。

例如，考虑如下循环链表。




经过分割后，上面的链表将变为下图的样子。




```
void SplitList(ListNode head, ListNode head1, ListNode head2) {
        ListNode slowPtr = head, fastPtr = head;
        if(head ==NULL) return;
        /* 如果循环链表有奇数个结点，那么fastPtr.getNext()将指向head，
           如果是偶数，那么fastPtr.getNext().getNext()将指向head */
        while(fastPtr.getNext() != head && fastPtr.getNext().getNext() != head)      {
                fastPtr = fastPtr.getNext().getNext();
                slowPtr = slowPtr.getNext();
        }
        /* 如果链表有偶数个元素，那么再向后移动一次fastPtr */
        if(fastPtr.getNext().getNext() == head)
                fastPtr = fastPtr.getNext();
        /* 设置前半部分的head指针 */
        head1 = head;
        /* 设置后半部分的head指针 */
        if(head.getNext() != head)
                head2 = slowPtr.getNext();
        /* 把后半部分变成环*/
        fastPtr.setNext(slowPtr.getNext());
        /* 把前半部分变成环*/
        slowPtr.setNext(head);
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**问题 37** 如何判定链表是否是回文(顺读和倒读都是一样的)?

**解答：**

**算法：**

1）获取链表的中间结点。

2）把链表的后半部分逆置。

3）比较链表的前半部分和后半部分。

4）再次逆置链表的后半部分，并添加到前半部分的后面，构造出原始链表。

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**问题 38** 交换链表中的相邻结点。

**解答：**

```
void ExchangeAdjacentNodes(ListNode head) {
        ListNode curNode, temp, nextNode;
        curNode = head;
        if(curNode ==null || curNode.getNext() == null) return;
        head = curNode.getNext();
        while(curNode != null && curNode.getNext() != null) {
                nextNode = curNode.getNext();
                curNode.setNext(nextNode.getNext());
                temp = curNode.getNext();
                nextNode.setNext(curNode);
                if(temp != null && temp.getNext() != null)
                        curNode.setNext(curNode.getNext().getNext());
                curNode = temp;
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**问题 39** 对于给定的 $K(K>0)$，逆置链表中包含 $K$ 个结点的块。

**例子：** 输入：1 2 3 4 5 6 7 8 9 10，对于不同的 $K$ 值，输出为：

$K=2$:2 1 4 3 6 5 8 7 10 9，$K=3$:3 2 1 6 5 4 9 8 7 10，$K=4$:4 3 2 1 8 7 6 5 9 1

**解答：**

**算法：** 该问题是对链表中交换结点问题的扩展。

1）检查链表当前剩余部分是否还有 $K$ 个结点。

a）如果有，获取(链表当前剩余部分)第 $K+1$ 个结点的指针。

b）否则返回。

2）逆置前面的 $K$ 个结点。

3）$K$ 个结点逆置后，设置其最后一个结点的后继指针指向第 $K+1$ 个结点。

4）(当前位置)移动到第 $K+1$ 个结点。

5）跳转到步骤 1。

6）如果执行了逆置操作，那么第一个块的第 $K-1$ 个结点成为新的表头结点，否则返回原表头结点。

```
ListNode GetKPlusOneThNode(int K, ListNode head) {
        ListNode Kth;
        int i;
        if(head== null) return head;
        for (int i=0, Kth=head; Kth != null && (i < K); i++, Kth=Kth.getNext());
        if(i==K && Kth!=null)
                return Kth;
        return head.getNext();
}
int HasKnodes(ListNode head, int K) {
        int i;
        for(i=0; head != null && (i < K); i++, head=head.getNext());
            if(i == K)
```

```
                return 1;
            return 0;
    }
    ListNode ReverseBlockOfK-nodesInLinkedList(ListNode head, int K) {
            ListNode temp, next, cur = head, newHead;
            if(K==0 || K==1)
                return head;
            if(HasKnodes(cur, K-1))
              newHead = GetKPlusOneThNode(K-1, cur);
            else newHead = head;
            while(cur != null && HasKnodes(cur, K)) {
                // 注意下面的步骤
                temp = GetKPlusOneThNode(K, cur);
                 int i=0;
                while(i<K) {
                    next = cur.getNext();
                    cur.setNext(temp);
                    temp = cur;
                    cur = next;
                    i++;
                }
            }
            return newHead;
    }
```

**问题 40**　是否有可能做到存取链表(元素)的时间复杂度为 $O(1)$?

**解答：可以**。在创建链表的同时建立一个散列表。把链表中的 $n$ 个元素保存在散列表中，需要的预处理时间开销为 $O(n)$。这样读取任意元素仅需常数时间 $O(1)$，读取 $n$ 个元素需要 $n * 1$ 时间单位 $= n$ 个单位时间。因此，使用分摊分析，可以说链表元素的存取能在 $O(1)$常数时间内完成。

**问题 41**　约瑟夫环：$N$ 个人想选出一个领头人，他们排成一个环，沿着环每数到第 $M$ 个人就从环中排除该人，并从下一个人开始重新数。请找出最后留在环中的人。

**解答：**假设输入是一个有 $N$ 个结点的循环链表，每个结点依次有一个编号($1 \sim N$)，表头结点的编号为 1。

```
ListNode  GetJosephusPosition(int N, int M) {
        ListNode p, q;
        // 建立一个包含所有人的循环链表
        p.setData(1);
        q = p;
        for (int i = 2; i <= N; ++i) {
                p = p.getNext();
                p.setData(i);
        }
        p.setNext(q);  // 设置表尾结点的后继指向第一个结点，构建出循环链表
        // 如果链表中选手数大于1，淘汰第M个选手
        for (int count = N; count > 1; --count) {
                for (int i = 0; i < M - 1; ++i)
                        p = p.getNext();
                p.setNext(p.getNext().getNext());  // 从链表中删除被淘汰选手的结点
        }
        System.out.println("Last player left standing (Josephus Position) is" + p.getData());
}
```

**问题 42**　给定一个链表，每个结点包含数据、下一个指针和一个指向链表中某个结点的随机指针。请设计一个复制链表的算法。

**解答：**可以使用散列表把新建的结点与给定链表中结点的实例关联在一起。

**算法：**

- 扫描原(给定)链表，对于每个结点 $X$，创建一个新结点 $Y$，其数据等于 $X$ 的数据，然后用 $X$ 作为主键将二元组($X$，$Y$)存入散列表。注意，在本次扫描中，结点 $Y$ 的下一个指针(Y. next)和随机指针(Y. random)都是 NULL，将在下一次扫描中确定它们的值。
- 现在，对于每个结点 $X$，已经复制了一个 $Y$ 并将 $Y$ 存储在散列表中了。再一次扫描原链表，设置新链表的各个指针。

```
ListNode Clone(ListNode head){
        ListNode X = head, Y;
        Map<ListNode, ListNode> HT = new HashMap<ListNode, ListNode>();
        while (X != null) {
                Y = new ListNode();
                Y.setData(X.getData());
                Y.setNext(null);
                Y.setRandom(null);
                HT.put(X, Y);
                X = X.getNext();
        }
        X = head;
        while (X != null) {
                // 从散列表中获得与Y对应的结点Y
                Y = HT.get(X);
                Y.setNext(HT.get(X.getNext());
                Y.setRandom(HT.get(X.getRandom());
                X = X.getNext();
        }
        // 返回新链表的表头，即结点Y
        return HT.get(head);
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 43** 一个有 $n$ 个结点的链表，在某个指针指向的元素后面插入一个元素的时间开销是

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

**解答：**A。

**问题 44 寻找模结点：**给定一个单向链表，链表的结点编号 $i$ 为[1..$n$]，其中 $n$ 为链表中元素的个数，编写一个函数从表头开始找到最后一个满足 $i\%k==0$ 条件的元素，$k$ 为一个整数常量。例如，如果 $n$ 为 19，$k$ 为 3，那么应该返回第 18 个结点。

**解答：**这个问题中，$n$ 的值预先是未知的。

```
ListNode modularNodes(ListNode head, int k){
    ListNode modularNode = null;
    int i=1;
    if(k<=0)
        return null;
    for (;head!= null; head = head.getNext()){
        if(i%k == 0){
            modularNode = head;
        }
        i++;
    }
```

```
    return modularNode;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**问题 45　从表尾开始找模结点：**给定一个单向链表，编写一个函数从表尾开始找到第一个满足 $n\%k==0$ 条件的元素，其中 $n$ 为链表中元素的个数，$k$ 为一个整数常量。例如，如果 $n$ 为 19，$k$ 为 3，那么应该返回第 17 个结点。

**解答：**这个问题中，$n$ 的值预先是未知的。求解方法与查找从链表表尾开始的第 $k$ 个元素类似。

```
ListNode modularNodes(ListNode *head, int k){
    ListNode modularNode=head;
    int i=0;
    if(k<=0)
        return null;
    for (i=0; i < k; i++){
        if(head!=null)
            head = head.getNext();
        else
            return null;
    }
    while(head!= null)
        modularNode = modularNode.getNext();
        head = head.getNext();
    }
    return modularNode;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**问题 46　找分数结点：**给定一个单向链表，编写一个函数找第 $\frac{n}{k}$ 个元素，其中 $n$ 为链表中元素的个数。

**解答：**这个问题中，$n$ 的值预先是未知的。

```
ListNode fractionalNodes(ListNode head, int k){
    ListNode fractionalNode;
    int i=0;
    if(k<=0)
        return null;
    for (;head!= null; head = head.getNext()){
        if(i%k == 0){
            if(fractionalNode!=null)
                fractionalNode = head;
            else fractionalNode = fractionalNode.getNext();
        }
        i++;
    }
    return fractionalNode;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**问题 47　求整数的无穷级数的中位数。**

**解答：**在一个有序整数数列中，当元素数量为奇数时，处于中间位置的值即为中位

数；当元素数量为偶数时，中位数则为处于中间位置的2个值的平均数。

可以使用链表求解这个问题(有序和无序链表都可以)。

首先，尝试使用无序链表来实现。在无序链表中，可以在表头或表尾插入元素。这个方法的缺点是找中位数的时间开销为$O(n)$。另外，插入操作的时间开销为$O(1)$。

现在，尝试用有序链表来实现。如果记录了中间元素，那么寻找中位数的时间开销为$O(1)$。在链表中的特定位置执行插入操作的时间开销也为$O(1)$。但是，找到正确的插入位置的时间开销为$O(n)$，而不像在有序数组内能在$O(\log n)$内完成。因为即使链表是有序的，也不能(像有序数组那样)执行二分查找。

所以，使用有序链表是不合适的，与有序数组一样，插入操作的时间开销是$O(n)$，寻找中位数的时间开销是$O(1)$。在有序数组中执行插入操作需要移位，所以时间开销是线性增长的。由于在链表中不能执行二分查找，所以插入操作的时间开销也是线性增长的。

**注意：** 有效的求解方法参见第7章。



Chapter 4

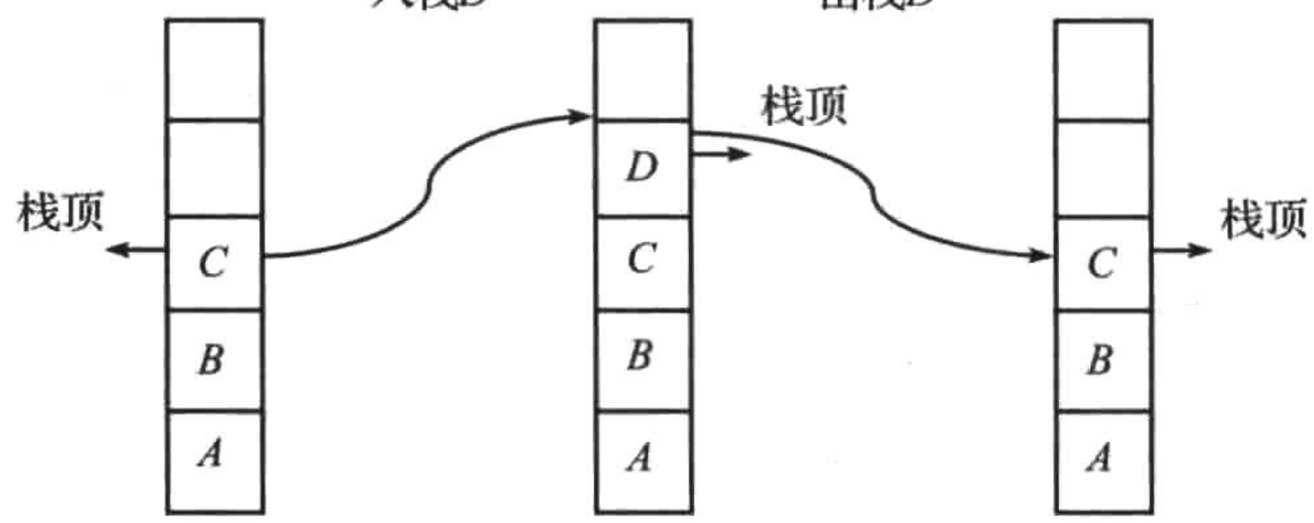
# 第4章 栈

## 4.1 什么是栈

栈是一种用于存储数据的简单数据结构(与链表类似)。数据入栈的次序是栈的关键。可以把自助餐厅中的一堆盘子看作一个栈的例子。当盘子洗干净后，它们会添加到栈的顶端。当需要盘子时，也是从栈的顶端拿取。所以第一个放入栈中的盘子最后才能被拿取。

**定义：** 栈(stack)是一个有序线性表，只能在表的一端(称为栈顶，top)执行插入和删除操作。最后插入的元素将第一个被删除。所以，栈也称为后进先出(Last In First Out，LIFO)或先进后出(First In Last Out，FILO)线性表。

两个改变栈操作都有专用名称。一个称为入栈(push)，表示在栈中插入一个元素；另一个称为出栈(pop)，表示从栈中删除一个元素。试图对一个空栈执行出栈操作称为下溢(underflow)；试图对一个满栈执行入栈操作称为溢出(overflow)。通常，溢出和下溢均认为是异常。下图是一个栈的例子。




## 4.2 如何使用栈

设想在某个办公室的一个普通工作日，有一位正忙于一个长期项目的开发人员。经

理突然给开发人员分配了一个更加紧急的新任务。开发人员只好将长期项目放在一边，开始处理新的任务。这时，电话响了，这是具有最高优先级的情况，因为必须立即应答。所以，开发人员只好暂停当前的工作，开始接听电话。结束通话后，开发人员回到刚才暂时搁置的任务，继续工作。如果还有其他的电话，开发人员仍按刚才的方式处理，直至最终完成分配的新任务。这时，开发人员才会回到长期项目中继续工作。

## 4.3 栈抽象数据类型

下面给出栈抽象数据类型中的操作。为了简单起见，假设数据类型为整型。

### 1. 栈的主要操作

- void push(int data)：将 data(数据)插入栈。
- int pop()：删除并返回最后一个插入栈的元素。

### 2. 栈的辅助操作

- int top()：返回最后一个插入栈的元素，但不删除。
- int size()：返回存储在栈中元素的个数。
- int isEmpty()：判断栈中是否有元素。
- int isStackFull()：判断栈中是否存满元素。

## 4.4 异常

在执行操作时发生的错误称为异常。当操作不能执行时，会“抛出”异常。在栈抽象数据类型中，pop 操作和 top 操作在栈空时是不能执行的。试图对一个空栈执行 pop(或 top)操作会抛出异常。试图对一个满栈执行 push 操作也会抛出异常。

## 4.5 应用

栈在下列应用中具有重要的作用。

### 1. 直接应用

- 符号匹配。
- 中缀表达式转换为后缀表达式。
- 计算后缀表达式。
- 实现函数调用(包括递归)。
- 求范围误差(极差)(在股票市场中求极差，参看 4.8 节)。
- 网页浏览器中已访问页面的历史记录(后退(back)按钮)。
- 文本编辑器中的撤销(undo)序列。
- HTML 和 XML 文件中的标签(tag)匹配。

### 2. 间接应用

- 作为一个算法的辅助数据结构(例如，树遍历算法)。
- 其他数据结构的组件(例如，模拟队列，参见第 5 章)。

## 4.6 实现

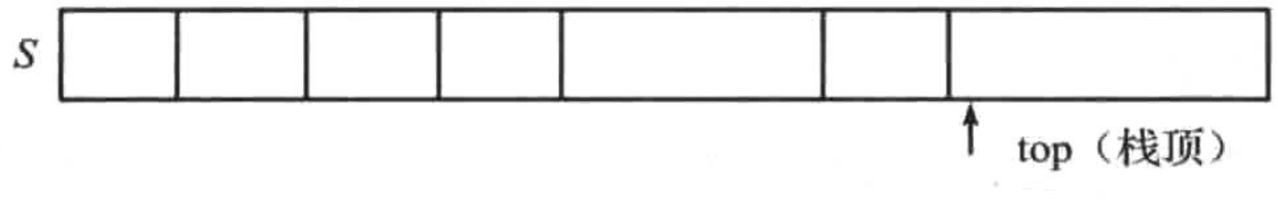
栈抽象数据类型有多种实现方式。下面是常用的方法。

- 基于简单数组的实现方法。

- 基于动态数组的实现方法。
- 基于链表的实现方法。

**1. 简单数组实现**

首先介绍基于简单数组实现栈抽象数据类型的方法。如下图所示，从左至右向数组中添加所有的元素，并定义一个变量用来记录数组当前栈顶(top)元素的下标。




当数组存满了栈元素时，执行入栈(插入元素)操作将抛出栈满异常。当对一个没有存储栈元素的数组执行出栈(删除元素)操作时，将抛出栈空异常。

```
public class  ArrayStack{
        private int top;
        private int capacity;
        private int[] array;
        public ArrayStack(){
                capacity = 1;
                array= new int[capacity];
                top = -1;
        }
        public boolean isEmpty (){
                // if the condition is true then 1 is returned else 0 is returned
                return (top == -1);
        }
        public int isStackFull(){
        //if the condition is true then 1 is returned else 0 is returned
        return (top == capacity - 1);  //or return (top == array.length);
}
public void push(int data){
        if(isStackFull(S)) System.out.println( “Stack Overflow”);
        else /*Increasing the ‘top’ by 1 and storing the value at ‘top’ position*/
                array[++top]= data;
}
public int pop(){
        if(isEmpty(S)){                         /* top == - 1 indicates empty stack*/
                System.out.println("Stack is Empty");
                return 0;
        }
        else      return ( array[top--]);
}
public void deleteStack(){
        top = -1;
        }
}
```

**性能和局限性**

**性能：**假设 $n$ 为栈中元素的个数。在基于简单数组的栈实现中，各种栈操作的算法复杂度如下表所示。

| | | | |
|---|---|---|---|
| 空间复杂度(用于 $n$ 次 push 操作) | $O(n)$ | isEmpty()的时间复杂度 | $O(1)$ |
| push()的时间复杂度 | $O(1)$ | isStackFull()的时间复杂度 | $O(1)$ |
| pop()的时间复杂度 | $O(1)$ | deleteStack()的时间复杂度 | $O(1)$ |
| size()的时间复杂度 | $O(1)$ | | |

**局限性：**栈的最大空间必须预先声明且不能改变。试图对一个满栈执行入栈操作将产生一个针对简单数组这种特定实现栈方式的异常。

**2. 动态数组实现**

在上述基于简单数组的栈实现方法中，采用一个下标变量 top，它始终指向栈中最新插入元素的位置。当插入(或 push)元素时，先增加下标变量 top 的值，然后在数组中该下标位置存储新元素。类似地，当删除(或 pop)元素时，先获取下标变量 top 位置的元素，然后减小变量 top 的值。当下标变量 top 的值等于－1 时，表示栈为空。然而仍然需要解决的一个问题是，在固定大小的数组中，如何处理所有空间都已经保存了栈元素这种情况。

a) **解决方法 1：**当栈满时，每次将数组的大小增加 1 将怎么样？

- Push()：$S[]$的大小增加 1。
- Pop()：$S[]$的大小减小 1。

b) **这种方法存在的问题**

这种增加数组大小的方法开销太大。具体分析如下。例如，当 $n=1$ 时，执行 push 操作的过程为，新建一个大小为 2 的数组，复制原数组中所有元素到新数组中，在新数组末端添加新元素。当 $n=2$ 时，执行 push 操作的过程为，新建一个大小为 3 的数组，复制原数组中所有元素到新数组中，在新数组末端添加新元素。

以此类推，当 $n=n-1$ 时，执行 push 操作的过程为，新建一个大小为 $n$ 的数组，复制原数组中所有元素到新数组中，在新数组末端添加新元素。在执行 $n$ 次 push 操作后，总时间开销 $T(n)$(复制操作的数量)为 $1+2+\cdots+n\approx O(n^2)$。

c) **解决方法 2：重复倍增。**

可以使用数组倍增技术来提高性能。如果数组空间已满，新建一个比原数组空间大一倍的新数组，然后复制元素。采用这种处理方法，执行 $n$ 次 push 操作的时间开销为 $n$(不是 $n^2$)。

为了简单起见，假设初始时从 $n=1$ 开始，一直增大到 $n=32$，即按照 1、2、4、8、16 的次序倍增。另一种相同的操作方式是，当 $n=1$ 时，执行 push 操作添加一个元素，新建一个比原数组空间大一倍的数组，复制原数组中所有元素到新数组中。

当 $n=1$ 时，需执行 1 次复制操作；当 $n=2$ 时，需执行 2 次复制操作；而当 $n=4$ 时，需执行 4 次复制操作；以此类推，当 $n=32$ 的时候，复制操作的总数为 $1+2+4+8+16=31$，其值约等于 $2n(32)$。仔细分析可以发现，倍增操作需执行 $\log n$ 次。

对上述的讨论归纳如下，执行 $n$ 次 push 操作需执行 $\log n$ 次数组空间的倍增操作。也就是说，下面表达式中共有 $\log n$ 项。那么 $n$ 次 push 操作的总时间开销 $T(n)$为

$$1+2+4+8+\cdots+\frac{n}{4}+\frac{n}{2}+n=n+\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\cdots+4+2+1$$

$$=n\left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{4}{n}+\frac{2}{n}+\frac{1}{n}\right)$$

$$=n(2)\approx 2n=O(n)$$

$T(n)$为 $O(n)$，一次 push 操作的平摊时间为 $O(1)$。

```
public class  DynArrayStack{
        private int top;
        private int capacity;
        private int[] array;
        public DynArrayStack(){
                capacity = 1;
```

```
            array= new int[capacity];
            top = -1;
    }
    public boolean isEmpty(){
            // if the condition is true then 1 is returned else 0 is returned
            return (top == -1);
    }
    public int isStackFull(){
            //if the condition is true then 1 is returned else 0 is returned
            return (top == capacity - 1);  //or return (top == array.length);
    }
    public void push(int data){
            if(isStackFull(S))
                    doubleStack(S);
             array[++top]= data;
    }
    private void doubleStack(){
            int newArray[] = new int[capacity*2];
            System.arraycopy(array, 0, newArray, 0, capacity);
            capacity = capacity*2;
            array = newArray;
    }
    public int pop() {
            if(isEmpty(S))   System.out.println( "Stack Overflow");
            else      return ( array[top--]);
    }
    public void deleteStack(){
            top = -1;
            }
    }
```

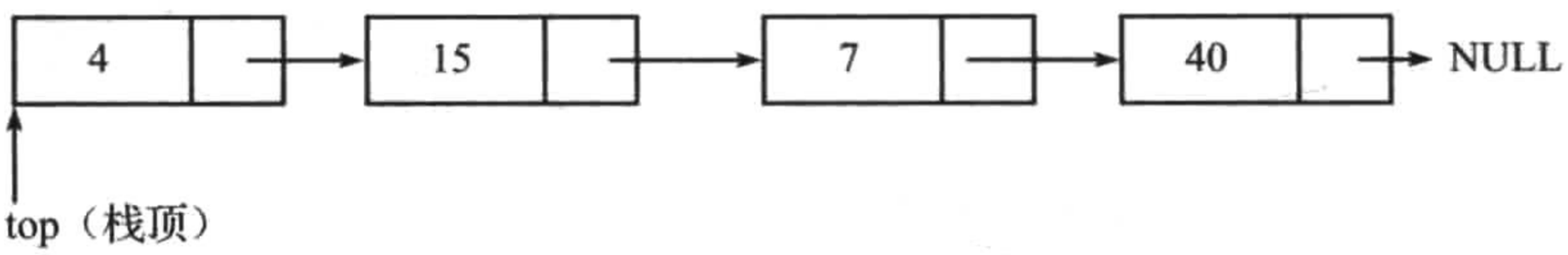
**性能**：假设 $n$ 为栈中元素的个数。在基于动态数组的栈实现中，各种栈操作的算法复杂度如下表所示。

| 空间复杂度(用于 $n$ 次 push 操作) | $O(n)$ | Top()的时间复杂度 | $O(1)$ |
|---|---|---|---|
| DynArrayStack()的时间复杂度(创建栈) | $O(1)$ | isEmpty()的时间复杂度 | $O(1)$ |
| push()的时间复杂度 | $O(1)$(平均) | isStackFull()的时间复杂度 | $O(1)$ |
| pop()的时间复杂度 | $O(1)$ | deleteStack()的时间复杂度 | $O(1)$ |

**注意**：倍增太多可能导致内存溢出。

### 3. 链表实现

使用链表也可以实现栈。通过在链表的表头插入元素的方式实现 push 操作，删除链表的表头结点(栈顶结点)实现 pop 操作。




```
public class LLStack{
        private LLNode headNode;
        public LLStack(){
                this.headNode = new LLNode(null);
        }
```

```
public void Push(int data){
        if(headNode == null){
                headNode = new LLNode(data);
        }else if(headNode.getData() == null){
                headNode.setData(data);
        }else{
                LLNode llNode = new LLNode(data);
                llNode.setNext(headNode);
                headNode = llNode;
        }
}
public int top(){
        if(headNode == null)
                return null;
        else   return headNode.getData();
}
public int pop(){
        if(headNode == null){
                throw new EmptyStackException("Stack empty");
        }else{
                int data = headNode.getData();
                headNode = headNode.getNext();
                return data;
        }
}
public boolean isEmpty(){
        if(headNode == null)      return true;
        else      return false;
        }
        public void deleteStack(){
                headNode null;
        }
}
```

**性能：**假设 $n$ 为栈中元素的个数。基于链表的栈实现中，各种栈操作的算法复杂度如下表所示。

| 空间复杂度(用于 $n$ 次 push 操作) | $O(n)$ | top()的时间复杂度 | $O(1)$ |
|---|---|---|---|
| LLStack()的时间复杂度(创建栈) | $O(1)$ | isEmpty()的时间复杂度 | $O(1)$ |
| push()的时间复杂度 | $O(1)$(平均) | deleteStack()的时间复杂度 | $O(n)$ |
| pop()的时间复杂度 | $O(1)$ | | |

## 4.7 栈的各种实现方法比较

### 1. 递增策略和倍增策略的比较

通过分析完成 $n$ 个 push 操作的总时间开销 $T(n)$来比较递增策略和倍增策略的区别。从长度为 1 的数组表示的空栈开始，一次 push 操作的平摊时间等于一组 push 操作的总时间开销的平均值，记为 $T(n)/n$。

**递增策略：**实现 push 操作的平摊时间开销为 $O(n)[O(n^2)/n]$。

**倍增策略：**实现 push 操作的平摊时间开销为 $O(1)[O(n)/n]$。

**注意：**具体分析请参见 4.6 节。

### 2. 基于数组实现和基于链表实现的比较

a) 基于数组实现的栈

- 各个操作都是常数时间开销。
- 每隔一段时间倍增操作的开销较大。
- (从空栈开始)$n$ 个操作的任意序列的平摊时间开销为 $O(n)$。

b) 基于链表实现的栈

- 栈规模的增加和减小都很简洁。
- 各个操作都是常数时间开销。
- 每个操作都要使用额外的空间和时间开销来处理指针。

## 4.8 栈的相关问题

**问题 1** 如何使用栈来判定括号是否匹配?

**解答:** 对于给定的表达式，可以使用栈来实现括号匹配判定算法。这个算法在编译器中非常重要。解析器每次读入一个字符，如果字符是一个开分隔符(如(、{、或[)，那么将其入栈。若读入的是一个闭分隔符(如)、}、或])，那么将栈顶的开分隔符出栈，并与闭分隔符比较。如果两者匹配，则继续解析字符串。如果不匹配，解析器显示匹配错误。下面给出一个时间复杂度为 $O(n)$的基于栈的符号匹配判定算法。

**算法:**

a) 创建一个栈。

b) 当(当前字符不等于输入的结束字符)

1) 如果当前字符不是匹配的字符，则忽略它。

2) 如果字符是一个开分隔符(如(、{、或[)，那么将其入栈。

3) 如果字符是一个闭分隔符(如)、}、或])，且栈不为空，栈顶元素出栈，否则提示匹配错误。

4) 如果出栈的字符不是相匹配的开分隔符，提示匹配错误。

c) 字符串处理结束后，如果栈不为空，则提示匹配错误。

| 例子 | 合法 | 说明 |
|---|---|---|
| $(A+B)+(C-D)$ | 是 | 表达式括号匹配 |
| $((A+B)+(C-D)$ | 否 | 缺少一个闭分隔符 |
| $((A+B)+[C-D])$ | 是 | 开分隔符与其相近的闭分隔符都匹配 |
| $((A+B)+[C-D]\}$ | 否 | 最后一个闭分隔符与第一个开分隔符不匹配 |

为了跟踪算法，假设输入为：()((())[()])。

| 输出符号，$A[i]$ | 操作 | 栈 | 输出 |
|---|---|---|---|
| ( | Push 字符 ( | ( | |
| ) | Pop 字符 (<br>检验字符(与 $A[i]$是否匹配？匹配 | | |
| ( | Push 字符 ( | ( | |
| ( | Push 字符 ( | (( | |
| ) | Pop 字符 (<br>检验字符( 与 $A[i]$是否匹配？匹配 | ( | |
| [ | Push 字符 [ | ([ | |

（续）

| 输出符号，$A[i]$ | 操作 | 栈 | 输出 |
|---|---|---|---|
| ( | Push 字符 ( | ([( | |
| ) | Pop 字符 (<br>检验字符( 与 $A[i]$是否匹配？匹配 | ([ | |
| ] | Pop 字符 [<br>检验字符 [ 与 $A[i]$是否匹配？匹配 | ( | |
| ) | Pop 字符 (<br>检验字符( 与 $A[i]$是否匹配？匹配 | | |
| | 检验栈是否为空？为空 | | TRUE(匹配) |

时间复杂度为 $O(n)$，只需扫描输入一次。空间复杂度为 $O(n)$，用于栈空间。

**问题 2** 如何使用栈来实现将中缀表达式转换为后缀表达式的算法？

**解答：** 在讨论算法前，首先介绍中缀、前缀和后缀表达式的定义。

**中缀：** 中缀表达式由一个单一字符或运算符，连接前后两个中缀字符串共同组成。

$$A$$

$$A+B$$

$$(A+B)+(C-D)$$

**前缀：** 前缀表达式由一个单一字符或运算符，随后是两个前缀字符串共同组成。每个前缀字符串长度大于 1，包含一个运算符、第一个操作数和第二个操作数。

$$A$$

$$+AB$$

$$++AB-CD$$

**后缀：** 后缀表达式(逆波兰表达式)由两个后缀字符串，随后是一个单一字符或运算符共同组成。每个后缀字符串长度大于 1，包含第一个操作数和第二个操作数，随后是一个运算符。

$$A$$

$$AB+$$

$$AB+CD-+$$

前缀和后缀表达式是无需括号描述数学表达式的方法。计算后缀和前缀表达式的时间开销是 $O(n)$，$n$ 是数组中元素的个数。

| 中缀 | 前缀 | 后缀 |
|---|---|---|
| $A+B$ | $+AB$ | $AB+$ |
| $A+B-C$ | $-+ABC$ | $AB+C-$ |
| $(A+B)*C-D$ | $-*+ABCD$ | $AB+C*D-$ |

下面设计表达式转换算法。在中缀表达式中，除非使用括号，否则运算符优先级是隐式的。因此，在设计将中缀表达式转换为后缀表达式的算法时，必须定义运算符的优先级。下表给出了运算符之间的优先级和相关性(计算的先后次序)。

| 符号 | 运算 | 优先级 | 相关性 |
| --- | --- | --- | --- |
| ( )<br>[ ]<br>→ . | 函数调用<br>数组元素<br>结构体或联合体的成员 | 17 | 从左向右 |
| ++ -- | 自增，自减 | 16 | 从左向右 |
| -- ++<br>!<br>~<br>- +<br>& *<br>sizeof | 自减，自增<br>逻辑非<br>按位反<br>一元减或加<br>取地址或取值<br>求变量的空间大小(字节数) | 15 | 从右向左 |
| (type) | 类型转换 | 14 | 从右向左 |
| * / % | 乘法 | 13 | 从左向右 |
| + - | 二进制加或减 | 12 | 从左向右 |
| << >> | 移位 | 11 | 从左向右 |
| > >=<br>< <= | 关系 | 10 | 从左向右 |
| == ! = | 等式 | 9 | 从左向右 |
| & | 按位与 | 8 | 从左向右 |
| ^ | 按位异或 | 7 | 从左向右 |
| \| | 按位或 | 6 | 从左向右 |
| && | 逻辑与 | 5 | 从左向右 |
| \|\| | 逻辑或 | 4 | 从左向右 |
| ?: | 条件 | 3 | 从右向左 |
| = += -= /= *= %=<br><<= >>=<br>&= ^= \|= | 赋值 | 2 | 从右向左 |
| , | 逗号 | 1 | 从左向右 |

**重要性质**

- 仔细比较中缀表达式 2+3*4 和后缀表达式 2 3 4 * +，可以发现两种表达式中数字(操作数)的次序是相同的，都是 2 3 4。但是运算符 * 和+的次序在两个表达式中是不同的。
- 只需一个栈就可以把中缀表达式转换为后缀表达式。利用栈把表达式中运算符的次序从中序改变为后序。栈中仅存储运算符和左括号‘(’。由于后缀表达式中不包含括号，所以输出后缀表达式时将不输出括号。

**算法：**

a) 创建一个栈

b) for (输入字符串中的每个字符 t) {

```
if ( t 是一个操作数) 输出 t
else if (t 是一个右括号)
```

```
                出栈并输出该符号,直至一个左括号出栈(但左括号不输出)
        else // t 是一个运算符或左括号{
                出栈并输出该符号,直至出现一个比 t 的优先级小的符号,或者出现
                一个左括号,或者栈空
                t 入栈
        }
    }
```

c) 出栈并输出该符号,直至栈空

为了更好地理解上述转换算法，给出表达式 $A*B-(C+D)+E$ 的具体执行过程。

| 输入字符 | 对栈执行的操作 | 栈 | 后缀表达式 |
|---|---|---|---|
| $A$ | | 空 | $A$ |
| * | 入栈 | * | $A$ |
| $B$ | | * | $AB$ |
| − | 检查和入栈 | − | $AB*$ |
| ( | 入栈 | −( | $AB*$ |
| $C$ | | −( | $AB*C$ |
| + | 检查和入栈 | −(+ | $AB*C$ |
| $D$ | | | $AB*CD$ |
| ) | 出栈并在'('前添加后缀 | − | $AB*CD+$ |
| + | 检查和入栈 | + | $AB*CD+-$ |
| $E$ | | + | $AB*CD+-E$ |
| 输入结束 | 出栈直到栈空 | | $AB*CD+-E+$ |

**问题 3** 给定一个有 $n$ 个符号的数组，有多少种可能的栈排列？

**解答：** $n$ 个符号的栈排列数需用卡塔兰数(catalan number)表示，在第 19 章中将讨论这个问题。

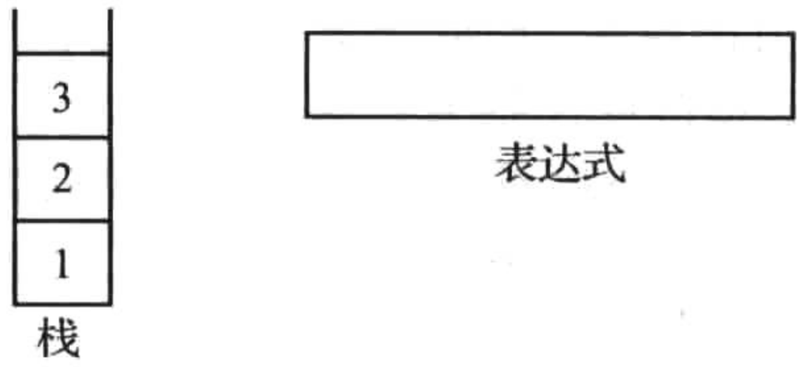
**问题 4** 使用栈计算后缀表达式的值？

**解答：**

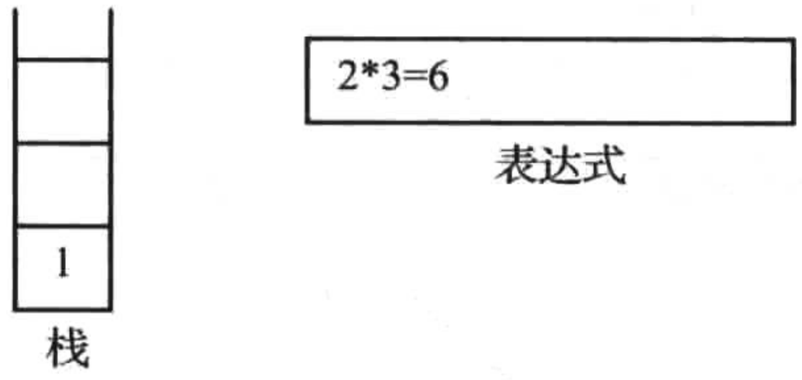
**算法：**

1) 从左向右扫描后缀表达式字符串。
2) 初始化一个空栈。
3) 重复第 4 步和第 5 步，直至扫描完所有字符。
4) 如果被扫描的字符是一个操作数，将其入栈。
5) 如果被扫描的字符是一个运算符，并且是一个一元运算符，那么只出栈一个元素。如果是一个二元运算符，那么出栈两个元素。元素出栈后，应用运算符对其计算并将计算结果入栈。
6) 在所有字符扫描结束后，栈中应该只有一个元素。
7) 栈顶的值作为结果返回。

**例子：** 下面用一个例子来解释上面的算法是如何工作的。假设后缀表达式字符串为 123 * +5−。首先初始化一个空栈。扫描的前 3 个字符分别是 1、2 和 3，均为操作数。按照扫描的先后次序依次入栈。



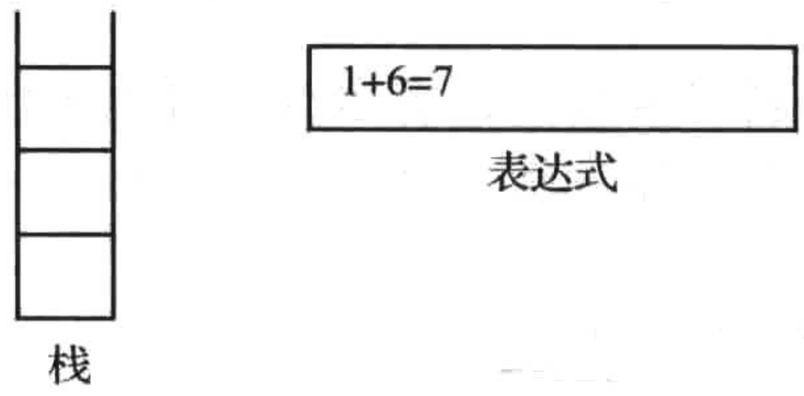

下一个被扫描的字符“*”是一个运算符。所以，将栈顶的两个元素出栈，并对两个元素执行“*”操作。注意第一个出栈的元素是第二个操作数。



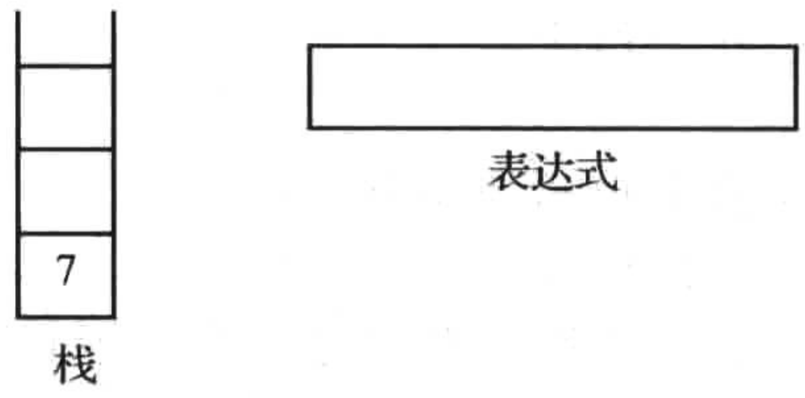

将表达式(2 * 3)的计算结果 6 入栈。



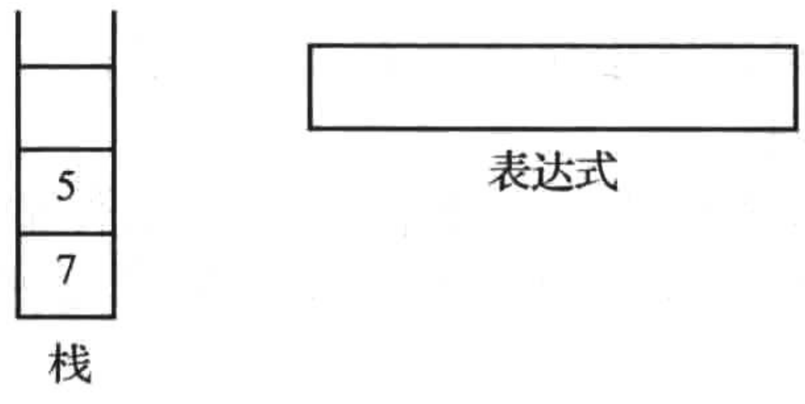

下一个扫描的字符“+”是一个运算符。所以，将栈顶的两个元素出栈，并对两个元素执行“+”操作。注意第一个出栈的元素是第二个操作数。



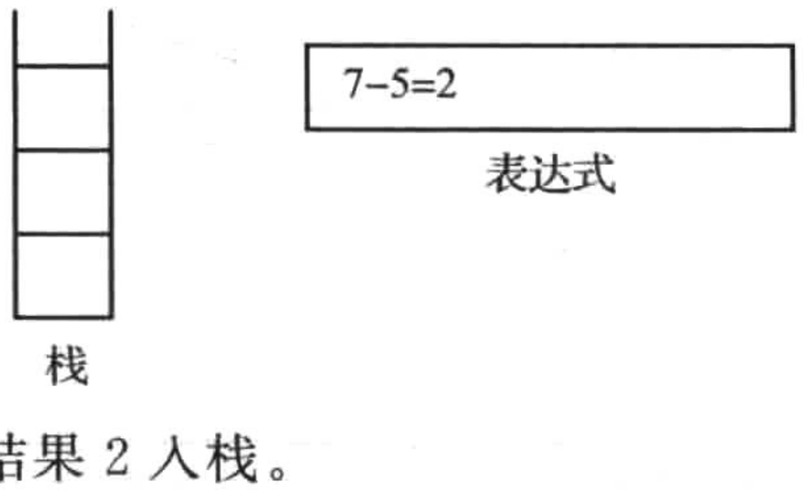

将表达式(1+6)的计算结果 7 入栈。



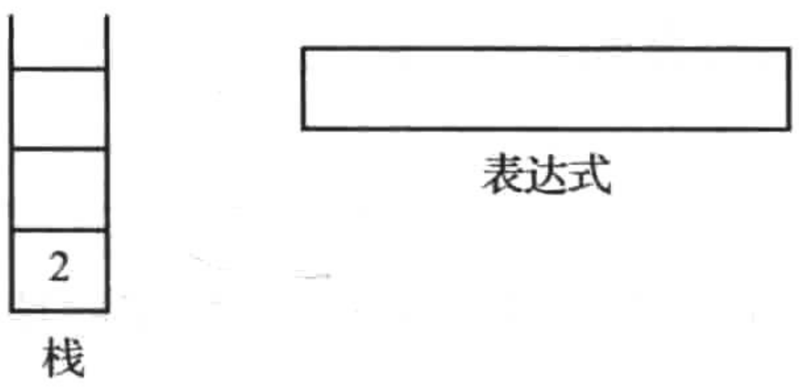

下一个扫描的字符“5”是一个操作数，将其入栈。



下一个扫描的字符“－”是一个运算符。所以，将栈顶的两个元素出栈，并对两个元素执行“－”操作。注意第一个出栈的元素是第二个操作数。




将表达式(7－5)的计算结果2入栈。




因为所有字符都被扫描，所以栈中剩下的元素(栈中只能有一个元素)将作为最终计算结果返回。

- 后缀表达式字符串：123 * ＋5－
- 计算结果：2

**问题5** 如何用栈只扫描一遍就能计算中缀表达式的值？

**解答：** 使用2个栈，不需要转换为后缀表达式，只扫描一遍就能计算中缀表达式的值。

**算法：**

1）创建一个空操作符栈
2）构造一个空操作数栈
3）对于输入(中缀表达式)串中的每一个符号
  a. 获取中缀表达式字符串的下一个符号
  b. 如果下一个符号是操作数，则把它压入操作数栈
  c. 如果下一个符号是运算符
    i. 评价运算符优先级
    ii. 若操作符栈不为空，则操作符和操作数出栈(左和右)，对两个操作数用操作符执行计算，并将得到的结果压入操作数栈。
4）从操作数栈出栈，得到计算结果。

**问题6** 如何设计一个栈，使得getMinimum( )操作的时间复杂度为$O(1)$？

**解答：** 设计一个辅助栈保存原栈中所有元素的最小值。并且，假设(辅助)栈中每个元素的值都比它下面的元素小。为了简单起见，原栈称为main stack，辅助栈称为min stack。

当main stack中的元素出栈时，min stack中的元素也出栈。当main stack中有新元素入栈时，将其与当前最小值进行比较，将两者中更小的元素压入min stack中。在任何时候，如果需要获取最小值，那么只需从min stack中返回栈顶元素。以下面的问题为例，初始时假设2、6、4、1和5依次入栈。按照上述算法，执行步骤如下：

| main stack | min stack |
|---|---|
| 5→栈顶 | 1→栈顶 |
| 1 | 1 |
| 4 | 2 |
| 6 | 2 |
| 2 | 2 |

经过两次出栈后：

| main stack | min stack |
|---|---|
| 4→栈顶 | 2→栈顶 |
| 6 | 2 |
| 2 | 2 |

基于上述的讨论，push、pop 和 getMinimum()操作的伪代码如下：

```
public class AdvancedStack implements Stack{
        private Stack elementStack = new LLStack();
        private Stack minStack = new LLStack();
        public void push(int data){
                elementStack.push(data);
                if(minStack.isEmpty() || (Integer)minStack.top() >= (Integer)data){
                        minStack.push(data);
                }else{
                        minStack.push(minStack.top());
                }
        }
        public int pop(){
                if(elementStack.isEmpty()) return null;
                minStack.pop();
                return elementStack.pop();
        }
        public int getMinimum(){
                return minStack.top();
        }
        public int top(){
                return elementStack.top();
        }
        public boolean isEmpty(){
                return elementStack.isEmpty();
        }
}
```

时间复杂度为 $O(1)$。

空间复杂度为 $O(n)$，用于 min stack 的空间开销。

**问题 7** 能否设计空间复杂度更优的算法求解问题 6？

**解答：可以。**上题求解方法的主要问题在于，每次 push 操作时，辅助栈也执行了一次 push 操作(新元素或当前的最小元素)。也就是说，重复执行了最小值的入栈操作。

现在，修改算法来降低空间复杂度。仍然设置一个 min stack，但是，只有当从 main stack 中出栈的元素等于 min stack 栈顶的元素时，才对 min stack 执行出栈操作。只有当 main stack 入栈的元素小于或等于当前最小值时，才对 min stack 执行入栈操作。如果需

要获得最小值，只需要返回 min stack 的栈顶元素。按照上述新算法，上题中实例的执行过程如下：

| main stack | min stack |
|---|---|
| 1→栈顶 | |
| 5 | |
| 1 | |
| 4 | 1→栈顶 |
| 6 | 1 |
| 2 | 2 |

从栈顶开始执行出栈操作，因为 1==1，所以两个栈都执行出栈操作：

| main stack | min stack |
|---|---|
| 5→栈顶 | |
| 1 | |
| 4 | |
| 6 | 1→栈顶 |
| 2 | 2 |

因为 5>1，所以只有 main stack 执行出栈操作：

| main stack | min stack |
|---|---|
| 1→栈顶 | |
| 4 | |
| 6 | 1→栈顶 |
| 2 | 2 |

因为 1==1，所以两个栈都执行出栈操作：

| main stack | min stack |
|---|---|
| 4→栈顶 | |
| 6 | |
| 2 | 2→栈顶 |

**注意：只在入栈和出栈操作时有区别。**

```
public class AdvancedStack  implements Stack{
        private Stack elementStack = new LLStack();
        private Stack minStack = new LLStack();
        public void Push(int data){
                elementStack.push(data);
                if(minStack.isEmpty() || (Integer)minStack.top() >= (Integer)data){
                        minStack.push(data);
                }
        }
        public int Pop(){
                if(elementStack.isEmpty())
                        return null;
```

```
                Integer minTop = (Integer) minStack.top();
                Integer elementTop = (Integer) elementStack.top();
                if(minTop.intValue() == elementTop.intValue())
                        minStack.pop();
                return elementStack.pop();
        }
        public int GetMinimum(){
                return minStack.top();
        }
        public int Top(){
                return elementStack.top();
        }
        public boolean isEmpty(){
                return elementStack.isEmpty();
        }
}
```

时间复杂度为 $O(1)$。

空间复杂度为 $O(n)$，用于 min stack 的空间开销。当算法运行过程中很少遇到“新最小值或相等”情况时，该算法的空间利用效率将大大改善。

**问题 8** 给定一个由多个 a 字符和 b 字符组成的字符数组。字符串中有一个特殊的字符 X 位于串的正中间(例如，ababa…ababXbabab…baaa)。如何判定该字符串是否为回文?

**解答：** 下面介绍一个最简单的算法。具体方法如下：首先定义两个下标分别指向字符串的头和尾。每次比较两个下标位置的值是否相等。如果不相等，那么输入的字符串不是回文。如果相等，左边的下标加 1，右边的下标减 1。重复上述步骤直至两个下标都指向串的正中间(X 所在位置)或者确定字符串不是回文。

```
int isPalindrome(String inputStr) {
        int i=0, j = inputStr.length;
        while(i < j && A[i] == A[j]) {
                i++;
                j--;
        }
        if(i < j ) {
                System.out.println("Not a Palindrome");
                return 0;
        }
        else {
                System.out.println("Palindrome");
                return 1;
        }
}
```

时间复杂度为 $O(n)$。

空间复杂度为 $O(1)$。

**问题 9** 对于问题 8，如果输入的字符串保存在一个单向链表中，如何判定该字符串是否为回文(即无法回退)。

**解答：** 参见第 3 章。

**问题 10** 能否使用栈技术求解问题 8?

**解答：可以。**

**算法：**

- 遍历链表直至遇到字符 X。

- 在遍历过程中将经过的每个字符(X 以前的字符)入栈。
- 对于链表的后一半，把每个元素与栈顶元素比较。如果相等，执行一次出栈操作，并且移动到下一个元素继续比较。
- 如果比较时出现不相等，那么输入的字符串不是回文。
- 继续这个过程，直至栈空或者字符串不是回文。

```
boolean isPalindrome(String inputStr){
        char inputChar[] = inputStr.toCharArray();
        Stack s = new LLStack();
        int i=0;
        while(inputChar[i] != 'X'){
                s.push(inputChar[i]);
                i++;
        }
        i++;
        while(i<inputChar.length){
                if(s.isEmpty()) return false;
                 if(inputChar[i] != ((Character)s.pop()).charValue()) return false;
                 i++;
        }
        return true;
}
```

时间复杂度为 $O(n)$。

空间复杂度为 $O(n/2)\approx O(n)$。

**问题 11**　给定一个栈，如何只使用栈操作(push 和 pop)逆置栈中的内容?

**解答:**

**算法:**

- 首先，将栈中所有的元素递归出栈，直至栈空。
- 每次递归向上步骤时，将上一步中出栈的元素插入栈底

```
public class StackReversal {
        public static void reverseStack(Stack stack){
                if(stack.isEmpty()) return;
                int temp = stack.pop();
                reverseStack(stack);
                insertAtBottom(stack, temp);
        }
        private static void insertAtBottom(Stack stack , int data){
                if(stack.isEmpty()){
                        stack.push(data);
                        return;
                }
                int temp = stack.pop();
                insertAtBottom(stack, data);
                stack.push(temp);
        }
}
```

时间复杂度为 $O(n^2)$。

空间复杂度为 $O(n)$，用于栈开销。

**问题 12**　如何有效地使用两个栈实现一个队列？分析队列相关操作的运行时间。

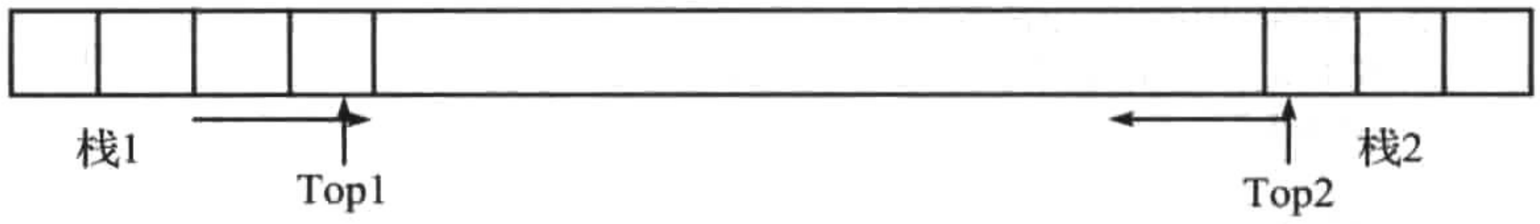
**解答:** 参见第 5 章。

**问题 13** 如何有效地使用两个队列实现一个栈？分析栈相关操作的运行时间。

**解答：** 参见第 5 章。

**问题 14** 如何只用一个数组实现两个栈？并且保证只要数组中还有剩余空间，栈操作就不能提示异常。

**解答：**




**算法：**

- 初始化两个下标变量分别指向数组的左右两端。
- 左边的下标指示第一个栈，右边的下标指示第二个栈。
- 如果需要对第一个栈执行元素入栈操作，那么将元素赋值到左边下标变量指示的位置。
- 如果需要对第二个栈执行元素入栈操作，那么将元素赋值到右边下标变量指示的位置。
- 第一个栈向右增长，第二个栈向左增长。

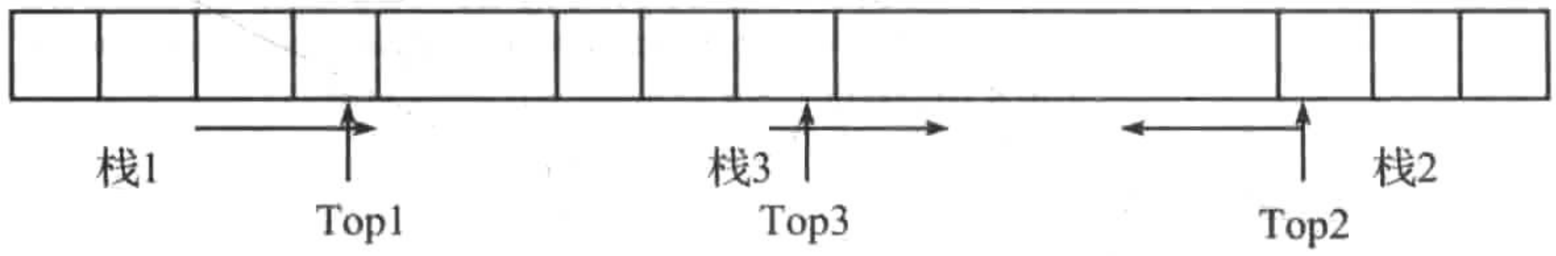
两个栈的 push（入栈）和 pop（出栈）操作的时间复杂度都为 $O(1)$。空间复杂度为$O(1)$。

```
public class ArrayWithTwoStacks{
        private int[] dataArray;
        private int size;
        private int topOne;
        private int topTwo;
        public ArrayWithTwoStacks(int size){
                if(size<2) throw new IllegalStateException("size < 2 is no persmissible");
                dataArray = new int[size];
                this.size = size;
                topOne = -1;
                topTwo = size;
        }
        public void push(int stackId, int data){
                if(topTwo == topOne+1) throw new StackOverflowException("Array is full");
                if(stackId == 1){
                        dataArray[++topOne] = data;
                }else if(stackId == 2){
                        dataArray[--topTwo] = data;
                }else return;
        }
        public int pop(int stackId){
                if(stackId == 1){
                        if(topOne == -1) throw new EmptyStackException("First Stack is Empty");
                        int toPop = dataArray[topOne];
                        dataArray[topOne--] = null;
                        return toPop;
                }else if(stackId == 2){
                        if(topTwo == this.size) throw new EmptyStackException ("Second Stack is
                        Empty");
                        int toPop = dataArray[topTwo];
```

```
                dataArray[topTwo++] = null;
                return toPop;
        }else return null;
}
public int top(int stackId){
        if(stackId == 1){
                if(topOne == -1) throw new EmptyStackException("First Stack is Empty");
                        return dataArray[topOne];
                }else if(stackId == 2){
                        if(topTwo == this.size)
                                throw new EmptyStackException("Second Stack is Empty");
                                return dataArray[topTwo];
                        }else return null;
        }
        public boolean isEmpty(int stackId){
                if(stackId == 1){
                                return topOne == -1;
                        }else if(stackId == 2){
                                return topTwo == this.size;
                        }else return true;
        }
}
```

**问题 15** 如何在一个数组中实现 3 个栈？

**解答：** 对于这个问题，可能有其他的解决方法。下面将给出一种可能的方法，只要数组有剩余空间，该方法就有效。




为了实现 3 个栈，需要保存以下信息：

- 第一个栈的下标(Top1)：标明第一个栈的大小。
- 第二个栈的下标(Top2)：标明第二个栈的大小。
- 第三个栈的开始下标(第三个栈的基地址)。
- 第三个栈栈顶的下标(Top3)。

下面，基于这种栈实现方式定义入栈和出栈操作。

**入栈：**

- 为了在第一个栈中执行入栈操作，首先需要判断如果添加一个新元素，它是否会碰到第三个栈。如果会碰到，那么要向右移动第三个栈，然后在(start1＋Top1)位置插入新元素。
- 为了在第二个栈中执行入栈操作，首先需要判断如果添加一个新元素后，它是否会碰到第三个栈。如果会碰到，那么要向左移动第三个栈，然后在(start2－Top2)位置插入新元素。
- 当在第三个栈执行入栈操作，需要判断是否会碰到第二个栈。如果会碰到，那么要向左移动第三个栈，然后在(start3＋Top3)位置插入新元素。

时间复杂度为 $O(n)$，因为可能需要移动第三个栈。空间复杂度为 $O(1)$。

**出栈：** 执行出栈操作，不需要移动，仅需要减小相应栈的大小。

时间复杂度为 $O(1)$，空间复杂度为 $O(1)$。

**该方法的伪代码**

```
public class ArrayWithThreeStacks {
        private int[] dataArray;
        private int size, topOne, topTwo, baseThree, topThree;
        public ArrayWithThreeStacks(int size){
                if(size<3) throw new IllegalStateException("Size < 3 is no persmissible");
                dataArray = new int[size];
                this.size = size;
                topOne = -1;
                topTwo = size;
                baseThree = size/2;
                topThree = baseThree;
}
public void push(int stackId, int data){
        if(stackId == 1){
                if(topOne+1 == baseThree){
                        if(stack3IsRightShiftable()){
                                shiftStack3ToRight();
                                dataArray[++topOne] = data;
                        }else throw new StackOverflowException("Stack1 has reached
                        max limit");
                }else dataArray[++topOne] = data;
        }else if(stackId == 2){
                if(topTwo-1 == topThree){
                        if(stack3IsLeftShiftable()){
                                shiftStack3ToLeft();
                                dataArray[--topTwo] = data;
                        }else throw new StackOverflowException("Stack2 has reached
                        max limit");
                }else dataArray[--topTwo] = data;
        }else if(stackId == 3){
                if(topTwo-1 == topThree){
                        if(stack3IsLeftShiftable()){
                                shiftStack3ToLeft();
                                dataArray[++topThree] = data;
                        }else throw new StackOverflowException("Stack3 has reached
                        max limit");
                }else dataArray[++topThree] = data;
        }else return;
}
public int pop(int stackId){
        if(stackId == 1){
                if(topOne == -1) throw new EmptyStackException("First Stack is Empty");
                int toPop = dataArray[topOne];
                dataArray[topOne--] = null;
                return toPop;
}else if(stackId == 2){
                if(topTwo == this.size) throw new EmptyStackException("Second Stack
                is Empty");
                int toPop = dataArray[topTwo];
                dataArray[topTwo++] = null;
                return toPop;
}else if(stackId == 3){
                if(topThree == this.size && dataArray[topThree] == null)
                        throw new EmptyStackException("Third Stack is Empty");
                int toPop = dataArray[topThree];
                if(topThree > baseThree) dataArray[topThree--] = null;
                if(topThree == baseThree) dataArray[topThree] = null;
```

```
                return toPop;
        }else return null;
}
public int top(int stackId){
        if(stackId == 1){
                if(topOne == -1) throw new EmptyStackException("First Stack is Empty");
                        return dataArray[topOne];
                }else if(stackId == 2){
                        if(topTwo == this.size)
                                throw new EmptyStackException("Second Stack is Empty");
                        return dataArray[topTwo];
                }else if(stackId == 3){
                        if(topThree == baseThree && dataArray[baseThree] == null)
                                throw new EmptyStackException("Third Stack is Empty");
                        return dataArray[topThree];
                }else return null;
}
public boolean isEmpty(int stackId){
        if(stackId == 1){
                        return topOne == -1;
        }else if(stackId == 2){
                return topTwo == this.size;
        }else if(stackId == 3){
                return (topThree == baseThree) && (dataArray[baseThree] == null);
        }else return true;
}
private void shiftStack3ToLeft() {
        for(int i=baseThree-1; i<=topThree-1;i++){
                dataArray[i] = dataArray[i+1];
        }
        dataArray[topThree--] = null;
        baseThree--;
}
private boolean stack3IsLeftShiftable() {
        if(topOne+1 < baseThree){
                return true;
        }
        return false;
}
private void shiftStack3ToRight() {
        for(int i=topThree+1; i>=baseThree+1;i--){
                dataArray[i] = dataArray[i-1];
        }
        dataArray[baseThree++] = null;
        topThree++;
}
private boolean stack3IsRightShiftable() {
        if(topThree+1 < topTwo){
                return true;
        }
        return false;
}
}
```

**问题 16** 针对问题 15，能否利用其他方法实现中间(第三个)栈?

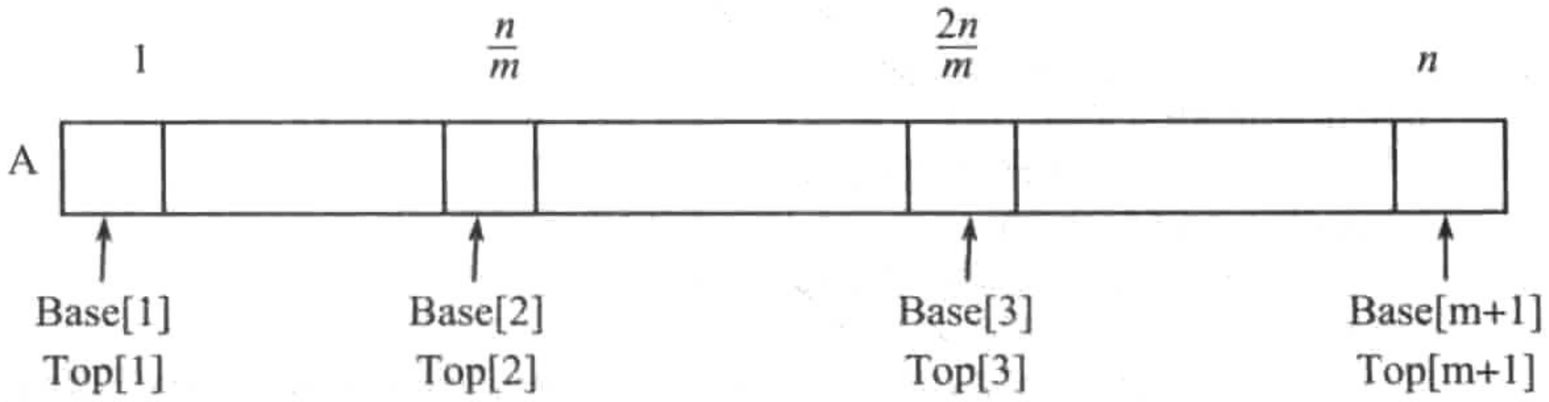
**解答：可以**。当左栈(向右增长)或右栈(向左增长)碰到中间栈时，需要移动整个中间栈来腾出空间。如果在中间栈执行入栈操作时碰到右边的栈，则也同样需要执行移动

操作。解决这个(大量移动)问题的方法是，在中间栈的两边交替地执行入栈操作。(例如，偶数次压入的元素在左栈执行入栈操作，奇数次则在右栈执行入栈操作)。这将使中间栈在数组中心保持平衡。但是，无论是中间栈自己增长，还是邻居栈增长，如果中间栈碰到左边栈或右边栈，则仍然需要移动中间栈。

如果三个栈的增减速率和平均大小不同，则可以通过设置 3 个栈的初始位置进行优化。例如，假设某个栈改动不大。如果将其放置在左边，则中间栈向反方向(向右)执行入栈操作，并将在中间栈和右边栈之间留下空间。如果它们相遇，那么数组的空间很可能已经用完。这种方法没有改变时间复杂度，但是减小了平均移动次数。

**问题 17** 如何在一个数组中实现 $m$ 个栈？

**解答：** 假设数组下标从 1～$n$。与问题 15 的求解方法类似，为了在一个数组中实现 $m$ 个栈，需要把数组分为 $m$ 个部分(如下图所示)。每个部分的大小为 $n/m$。




如上图所示，第一个栈的起始位置是下标 1(起始下标保存在 Base[1]中)，第二个栈的起始位置是下标 $n/m$(起始下标保存在 Base[2]中)，第三个栈的起始位置是下标 $2n/m$(起始下标保存在 Base[3]中)，以此类推。

与保存每个栈的起始下标的 Base 数组类似，定义一个保存每个栈顶元素下标的 Top 数组。为了便于讨论，定义下列术语。

- Top[$i$]($1 \leqslant i \leqslant m$)，保存第 $i$ 个栈的栈顶元素的下标。
- 如果 Base[$i$]==Top[$i$]，那么第 $i$ 个栈是空栈。
- 如果 Top[$i$]==Base[$i+1$]，那么第 $i$ 个栈已满。
  初始化时，Base[$i$]=Top[$i$]=$(i-1)n/m$($1 \leqslant i \leqslant m$)。
- 第 $i$ 个栈从 Base[$i$]+1 向 Base[$i+1$]的方向入栈元素。

**对第 $i$ 个栈执行入栈操作：**

1) 对第 $i$ 个栈执行入栈操作，首先需要检查第 $i$ 个栈的栈顶位置下标是否为 Base[$i+1$](这种情况表明第 $i$ 个栈已满)。也就是说，当向第 $i$ 个栈添加新元素时，判断是否会碰到第$i+1$个栈。如果出现这种情况，那么从第 $i+1$ 个栈一直到第 $m$ 个栈都要右移。最后才能在 Base[$i$]+Top[$i$]下标位置插入新元素。

2) 如果无法右移，那么可以把第 1～$i-1$ 个栈左移。

3) 如果两个方向都无法移动，那么返回所有的栈都已满。

```
void Push(int StackID, int data){
        if(Top[i] == Base[i+1])
                Print  i^th Stack is full and does the necessary action (shifting);
        Top[i] = Top[i]+1;
        A[Top[i]] = data;
}
```

时间复杂度为 $O(n)$，因为可能需要移动栈。

空间复杂度为 $O(1)$。

**对第 $i$ 个栈执行出栈操作：**

元素出栈，无需移动，仅需要减小相应栈的大小。唯一需要检查的是栈是否为空。

```
int Pop(int StackID) {
        if(Top[i] == Base[i])
                Print i^th Stack is empty;
        return  A[Top[i]--];
}
```

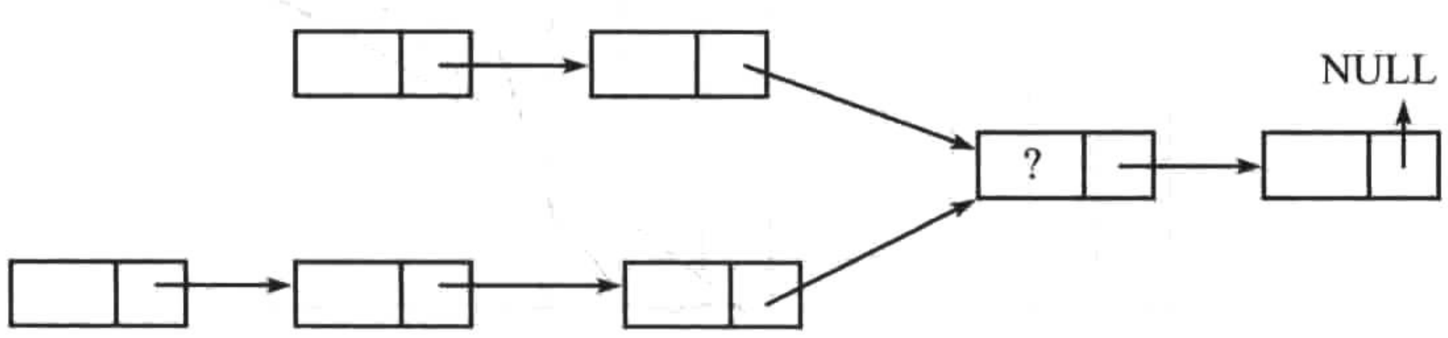
时间复杂度为 $O(1)$。

空间复杂度为 $O(1)$。

**问题 18** 假定有一个能存储整数的空栈。整数序列 123456 依次从左至右入栈。假设 $S$ 表示入栈，$X$ 表示出栈。通过一系列的入栈和出栈操作，能否输出序列 325641 和序列 154623 呢？

**解答：** SSSXXSSXSXXX 输出 325641。不能输出序列 154623，因为 2 比 3 先入栈，那么只能先输出 3 后，才能输出 2。

**问题 19** 假设两个单向链表在某个(结)点相交后，成为一个单向链表。两个链表的表头结点是已知的，但是相交的结点未知。也就是说，它们相交之前各自的结点数是未知的，并且两个链表的结点数也可能不同。令链表 List1 和链表 List2 在相交前的结点数分别为 $n$ 和 $m$，那么 $m$ 可能等于 $n$ 或小于 $n$，也可能大于 $n$。可以使用栈实现算法来找到两个链表的相交点吗？




**解答：可以。**具体算法参见第 3 章节。

**问题 20** 在本章讨论用动态数组实现栈时使用了“重复倍增”方法。针对同样的问题，如果采用创建一个大小为 $n+K$ 的数组方法替代倍增，算法的复杂度是多少？

**解答：** 假设栈的初始大小为 0。为了简单起见，假设 $K=10$。当插入新元素时，需要创建一个大小为 $0+10=10$ 的新数组。当大小为 10 的数组空间已满时，需要创建一个大小为 $10+10=20$ 的数组。这个过程持续进行下去，新建数组的大小为 30、40、50……也就是说，如果有 $n$ 个元素要入栈，那么将在第 $\frac{n}{10}$、$\frac{n}{20}$、$\frac{n}{30}$、$\frac{n}{40}$……个元素入栈时，创建新的数组。

复制操作的总数等于

$$\frac{n}{10}+\frac{n}{20}+\frac{n}{30}+\cdots+\frac{n}{10}n=\frac{n}{10}\left(\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\right)=\frac{n}{10}\log n\approx O(n\log n)$$

如果共执行了 $n$ 次入栈(push)操作，那么每次操作的开销为 $O(\log n)$。

**问题 21** 给定一个包含 $n$ 个 $S$ 和 $n$ 个 $X$ 的字符串，其中 $S$ 表示入栈操作，$X$ 表示出栈操作，栈初始时为空。请制定一个规则来检查给定的操作字符串是否容许 $S$ 操作？

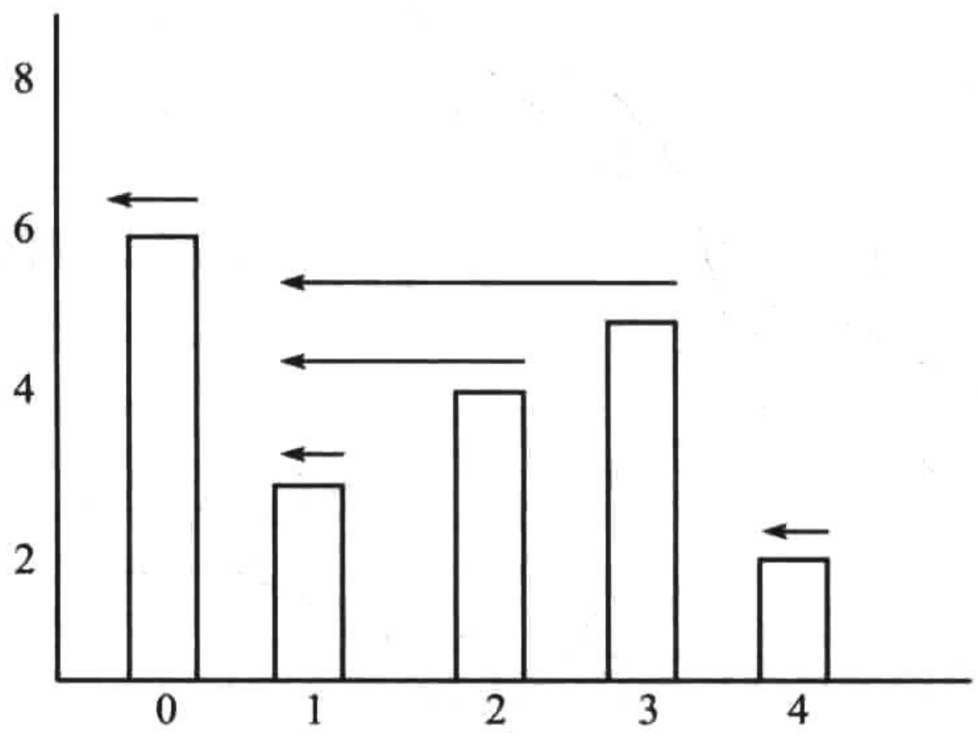
**解答：** 给定一个长度为 $2n$ 的字符串，检查给定操作的字符串在栈上的操作是否可

行。唯一受限制的操作是出栈操作，它的前提条件是栈非空。那么当从左至右遍历字符串时，在出栈 pop 操作之前，栈不能为空。这意味着，$S$ 的次数始终要大于等于 $X$ 的次数。因此，这个判定规则是在处理字符串的任何阶段，入栈(push)操作的次数要大于出栈(pop)操作的次数。

**问题 22** 计算跨度：给定数组 $A$，$A[i]$的跨度 $S[i]$定义为：满足 $A[j]\leqslant A[j+1]$且在 $A[i]$之前的连续元素 $A[j]$的最大个数。

**解答：** 在股票市场上这是一个非常常见的寻找峰值问题。在金融分析中要计算跨度(例如，股票在 52 周高位)。某一天一只股票的价格跨度 $i$ 等于股票价格小于或等于这天价格的最大连续天数(一直到当天)。例如，考虑下面的表和相应的跨度图。图中箭头表示跨度的长度。

| 日期的编号 $i$ | 输入的数组 $A[i]$ | $S[i]$：跨度 of $A[i]$ | 日期的编号 $i$ | 输入的数组 $A[i]$ | $S[i]$：跨度 of $A[i]$ |
|---|---|---|---|---|---|
| 0 | 6 | 1 | 3 | 5 | 3 |
| 1 | 3 | 1 | 4 | 2 | 1 |
| 2 | 4 | 2 | | | |




下面给出计算跨度的算法。一个简单的方法是，每天检查有多少连续日的股票价格低于当前的价格。

```
int[] FindingSpans(int[] inputArray){
        int[] spans = new int[inputArray.length];
        for(int i=0;i<inputArray.length;i++){
                int span = 1;
                int j=i-1;
                while(j>=0 && inputArray[j]<=inputArray[j+1]){
                        span++;
                        j--;
                }
                spans[i] = span;
        }
        return spans;
}
```

时间复杂度为 $O(n^2)$。空间复杂度为 $O(1)$。

**问题 23** 针对问题 22，能否设计性能更优的方法？

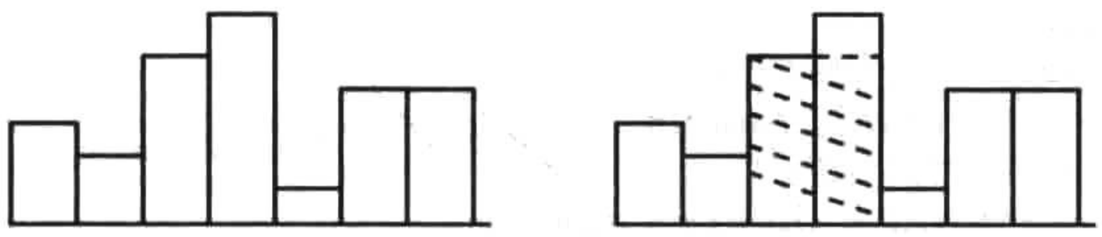
**解答：** 观察上题的例子可发现，如果能找到最近股票价格比第 $i$ 天高的是哪天，那么就能很容易地计算第 $i$ 天的跨度 $S[i]$。假定这一天记为 $P$。如果能找到这天，那么跨度

$S[i]=i-P$。

```
int[] FindingSpans(int[] inputArray){
        int[] spans = new int[inputArray.length];
        Stack stack = new LLStack();
        int p = 0;
        for(int i=0;i<inputArray.length;i++){
                while (!stack.isEmpty() && inputArray[i] > inputArray[(Integer) stack.top()])
                        stack.pop();
                if( stack.isEmpty())
                        p = -1;
                else    p = (Integer) stack.top();
                spans[i] = i - p;
                stack.push(i);
        }
        return spans;
}
```

时间复杂度：数组中的每个下标入栈一次，并且最多出栈一次。循环中的语句最多执行 $n$ 次。尽管这个算法有嵌套循环，但算法在整个执行过程中内层循环只执行 $n$ 次，因此算法的时间复杂度为 $O(n)$(可跟踪一个例子来统计内层循环成功执行了多少次)。空间复杂度为 $O(n)$，用于栈空间开销。

**问题 24　直方图中的最大矩形：**直方图是由排列在同一基线上的一系列矩形组成的多边形。为了简单起见，假设这些矩形的宽度相等但高度可能不同。例如，下面左图给出了一个直方图，其中各个矩形的高度为 3、2、5、6、1、4、4，宽度为标准单位 1。当给定一个保存了所有矩形高度的数组(假设宽度为 1)时，如何找到其中最大的矩形。对于给定的例子，最大矩形是如下面右图所示的斜线部分。



**解答：不完整子问题栈的线性搜索：**有很多方法解决这个问题。Judge 给出了一个基于栈求解该问题的算法。首先按照从左至右的次序来处理元素，并将已经开始但尚未完成的子直方图信息保存在一个栈中。如果栈为空，通过将元素压入栈来开启一个新的子问题。否则，将元素与栈顶元素比较。如果新元素大，那么将其入栈。如果两者相等，跳过。继续处理下一个新元素。

如果新元素小，那么用栈顶元素更新最大区域，并结束最顶层的子问题，然后删除栈顶元素，保持当前的新元素并重复上述这个过程(继续比较)。这样，所有的子问题都会结束直至栈为空，或者栈顶元素小于或等于新元素，导致上述行为。如果所有元素都被处理过而栈仍然不为空，那么用栈顶元素更新最大区域来结束剩余的子问题。

```
public class StackItem {
        public int height;
        public int index;
}
int MaxRectangleArea(int A[], int n){
        long maxArea = 0;
        if (A == null || A.length == 0)
                return maxArea;
        Stack<StackItem> S = new Stack<StackItem>();
```

```
        S.push(new StackItem(Integer.MIN_VALUE, -1));
        for (int i = 0; i <= n; i++) {
            StackItem cur = new StackItem((i < n ? A[i] : Integer.MIN_VALUE), i);
            if (cur.height > S.top().height) {
                S.push(cur);
                continue;
            }
            while (S.size() > 1) {
                StackItem prev = S.top();
                long area = (i - prev.index) * prev.height;
                if (area > maxArea) {
                    maxArea = area;
                }
                prev.height = cur.height;
                if (prev.height > S.get(S.size() - 2).height) {
                    break;
                }
                S.pop();
            }
        }
        return maxArea;
    }
```

每个元素入栈和出栈最多一次。函数中每步最多一个元素入栈或出栈。由于判定和更新操作的时间是常数，所以依据平摊分析，算法的时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于栈空间开销。

**问题 25** 设计一个可以把栈中元素按照升序排列的排序算法，并且算法不能对栈的具体实现方式有限定。

**解答：**

```
public class StackSorter {
    public static Stack sort(Stack s) {
        LLStack r = new LLStack();
        while(!s.isEmpty()) {
            int tmp = (Integer) s.pop();
            while(!r.isEmpty() && (Integer) r.top() > tmp) {
                s.push(r.pop());
            }
            r.push(tmp);
        }
        return r;
    }
}
```

时间复杂度为 $O(n^2)$。空间复杂度为 $O(n)$，用于栈空间开销。

**问题 26** 给定一个整数栈，如何检查栈中每对相邻数字是否连续。每对数字的值可以是递增或递减的，如果栈中元素的个数是奇数，那么组对时忽略栈顶元素。例如，假设栈中元素为[4，5，－2，－3，11，10，5，6，20]，那么算法应该输出真，因为每对二元组(4，5)、(－2，－3)、(11，10)和(5，6)都是连续的数字。

**解答：** 参见第 5 章。

**问题 27** 删除所有相邻的重复元素：给定一个数字数组，删除相邻的重复数字，结果数组中不能有任何相邻的重复数字。

| | |
|---|---|
| 输入：1，5，6，8，8，8，0，1，1，0，6，5<br>输出：1 | 输入：1，9，6，8，8，8，0，1，1，0，6，5<br>输出：1，9，5 |

**解答：**该题的解决方法要用到一个“就地栈”(in-place stack)的概念。当栈中元素与当前数字不相等时，将当前数字入栈。如果当前数字与栈顶元素相等，跳过该数字，直到找到与栈顶元素不相等的数字，并从栈中删除该元素。

```
public class RemoveAdjacentDuplicates {
    public int removeAdjacentDuplicates(int []A){
        int stkptr=-1;
        int i=0;
        while (i<A.length){
            if (stkptr == -1 || A[stkptr]!=A[i]){
                stkptr++;
                A[stkptr]=A[i];
                i++;
            }else {
                while(i < A.length&& A[stkptr]==A[i])
                    i++;
                stkptr--;
            }
        }
        return stkptr;
    }
}
public class TestRemoveAdjacentDuplicates {
    public static void main(String[] args) {
        RemoveAdjacentDuplicates obj = new RemoveAdjacentDuplicates();
        int[] A = {1,5,6, 8,8,8,0,1,1,0,6,5};
        int index = obj.removeAdjacentDuplicates(A);
        for (int i = 0; i <= index; i++) {
            System.out.print(" " + A[i]);
        }
    }
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于栈空间开销。



Chapter 5

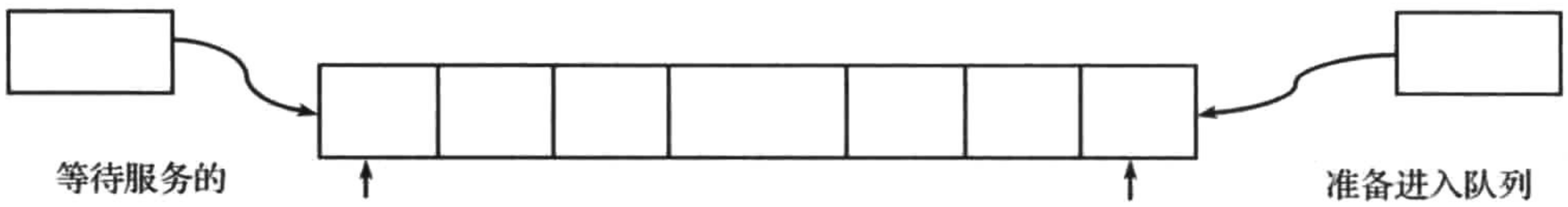
# 第5章 队　列

## 5.1　什么是队列

队列是一种用于存储数据的数据结构(与链表和栈类似)。数据到达的次序是队列的关键。在日常生活中队列是指从序列的开始按照顺序排列等待服务的一队人或物。

**定义：**队列是一种只能在一端插入(队尾)，在另一端删除(队首)的有序线性表。队列中第一个插入的元素也是第一个被删除的元素。所以，队列是一种先进先出(FIFO，First In First Out)或后进后出(LILO，Last In Last Out)线性表。

与栈类似，两个改变队列的操作各有专用名称。在队列中插入一个元素，称为入队(EnQueue)，从队列中删除一个元素，称为出队(DeQueue)。试图对一个空队列执行出队操作称为下溢(underflow)，试图对一个满队列执行入队操作称为溢出(overflow)。通常认为溢出和下溢是异常。下图是一个队列的例子。




## 5.2　如何使用队列

可以通过在售票柜台前排队购票的例子来理解队列的概念。新来购票的人只能从队尾开始排队等待，而队列最前面的人将是下一个被服务的对象，他将离开队列去柜台前买票。然后，他的下一位排队者将成为队列中的第一个人，并且将成为下一个离开队列买票的人。随着队列中最前面的人不断离开队列，其他人都随之向队列前面移动。最后，队列中的每个人都会依次到达队首，然后离开队列去接受服务。当需要维持一个先后次

序(或到达次序)时，采用队列是非常有用的。

## 5.3 队列抽象数据类型

下面给出队列抽象数据类型中的基本操作。队列中的插入和删除必须遵守先进先出原则。为了简单起见，假设元素都是整数。

### 1. 主要的队列操作

- enQueue(int data)：在队列的队尾插入一个元素。
- int deQueue()：删除并返回队首的元素。

### 2. 辅助的队列操作

- int Front()：返回队首的元素，但不删除。
- int QueueSize()：返回队列中存储的元素个数。
- int isEmpty()：指明队列是否存储了元素。

## 5.4 异常

与其他抽象数据类型类似，对一个空队列执行出队操作会抛出“队列空异常”，对一个满队列执行入队操作会抛出“队列满异常”。

## 5.5 应用

下面列举了一些队列的应用。

### 1. 直接相关的应用

- 操作系统根据(具有相同优先级的)任务到达的顺序调度任务(例如，打印队列)。
- 模拟现实世界中的队列，如售票柜台前的队伍，或者任何需要先来先服务的场景。
- 多道程序设计。
- 异步数据传输(文件输入输出、管道、套接字)。
- 客户在呼叫中心的等待时间。
- 确定超市收银员的数量。

### 2. 间接相关的应用

- 作为算法的辅助数据结构。
- 其他数据结构的组件。

## 5.6 实现

队列抽象数据类型(与栈类似)有多种实现方式。下面是常用的方法。

- 基于简单循环数组的实现方法。
- 基于动态循环数组的实现方法。
- 基于链表的实现方法。

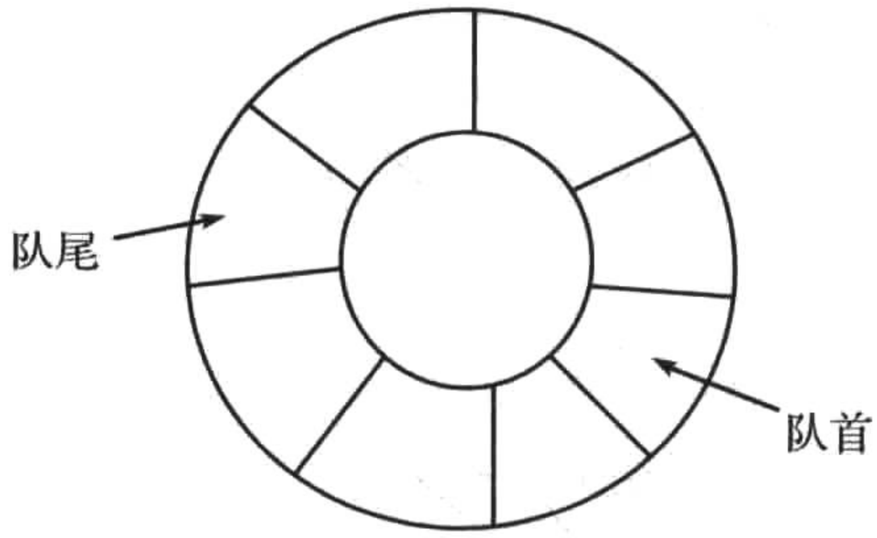
### 1. 为什么需要循环数组

首先，分析是否可以借鉴基于简单数组实现栈的方法来实现队列。由队列的定义可知，只能在队列一端执行插入操作，而在另一端执行删除操作。当执行多次插入和删除操作后，就可以很容易地发现使用简单数组实现队列的问题。

如下图例所示，可以清楚地看到数组中靠前的空间被浪费了，所以基于简单数组实

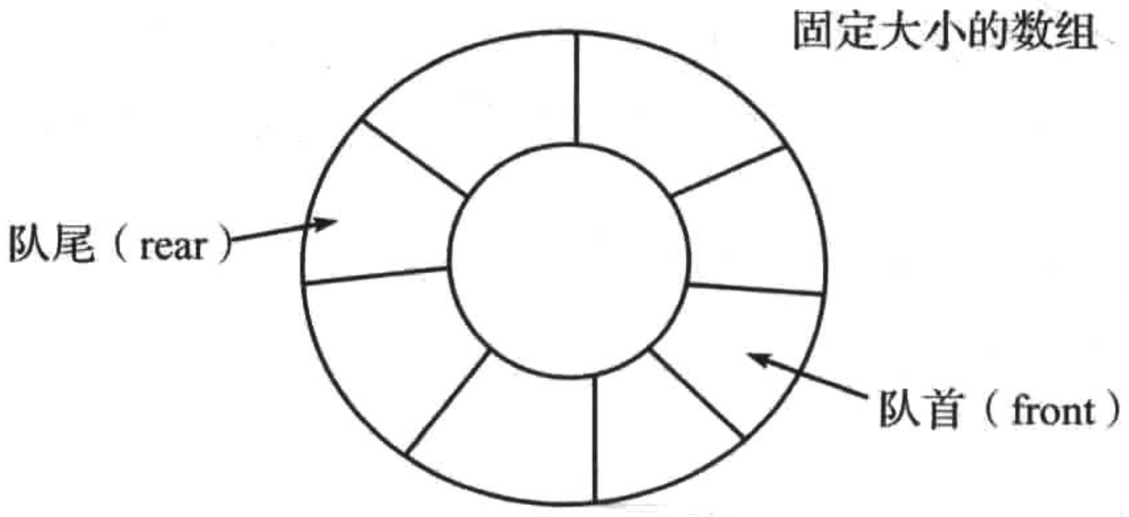
现队列不是一个有效的方法。为了解决这个问题，假设数组是循环存储的方式，即将数组最后一个元素和第一个元素看作连续的。依据这个假设，如果数组前端有空闲的空间，指向队尾的指针就能够很容易地移动到下一个空闲的位置。




**注意：**基于简单循环数组和动态循环数组来实现队列与基于数组实现栈的方法非常相似。可参见第 4 章来分析这些实现方法。

### 2. 基于简单循环数组实现队列




队列抽象数据类型的这种简单实现使用数组。在数组中，采用循环增加元素的方式，并使用两个变量分别记录队首元素和队尾元素。通常，使用 front 变量和 rear 变量分别表示队列中的队首元素和队尾元素。

基于数组来存储队列中的元素，可能会出现数组被填满的情况。这时，如果执行入队(enQueue)操作将抛出“队列满异常”。类似地，如果对空队列执行元素删除操作将抛出“队列空异常”。

**注意：**初始化时，front 和 rear 变量的值都置为－1，表示队列为空。

```
public class ArrayQueue {
        private int front;
        private int rear;
        private int capacity;
        private int [] array;
        private ArrayQueue(int size){
                capacity = size;
                front = -1;
```

```
            rear = -1;
            array = new int [size];
}
public static ArrayQueue createQueue(int size){
            return new ArrayQueue(size);
}
public boolean isEmpty(){
            return (front == -1);
}
public boolean isFull(){
            return ((rear+1)%capacity == front);
}
public int getQueueSize(){
            return((capacity-front+rear+1)%capacity);
}
public void enQueue(int  data){
            if(isFull()){
                        throw new QueueOverflowException("Queue Overflow");
            }else{
                        rear = (rear+1)%capacity;
                        array[rear] = data;
                        if(front == -1){
                                    front = rear;
                        }
            }
}
public int deQueue(){
            int  data = null;
            if(isEmpty()){
                        throw new EmptyQueueException("Queue Empty");
            }else{
                        data = array[front];
                        if(front == rear){
                                    front = rear-1;
                                    }else{
                                                front = (front+1)%capacity;
                                    }
                        }
                        return data;
            }
}
```

**性能和局限性**

**性能：** 假设 $n$ 为队列中元素的个数。

| | |
|---|---|
| 空间复杂度(用于 $n$ 次 enQueue 操作) | $O(n)$ |
| enQueue ()的时间复杂度 | $O(1)$ |
| deQueue ()的时间复杂度 | $O(1)$ |
| isEmpty()的时间复杂度 | $O(1)$ |
| isFull()的时间复杂度 | $O(1)$ |
| getQueueSize ()的时间复杂度 | $O(1)$ |

**局限性：** 用于实现队列的数组的最大空间必须预先声明且不能改变。试图对一个满队列执行入队操作会产生一个针对简单数组这种特定实现队列方式的异常。

### 3. 基于动态循环数组实现队列

```
public class DynArrayQueue implements Queue{
        private int front;
        private int rear;
        private int capacity;
        private int[] array;
        private DynArrayQueue(){
                capacity = 1;
                front = -1;
                rear = -1;
                array = new int[1];
        }
        public static DynArrayQueue createDynArrayQueue(){
                return new DynArrayQueue();
        }
        public boolean isEmpty(){
                return (front == -1);
        }
        private boolean isFull(){
                return ((rear+1)%capacity == front);
        }
        public int getQueueSize(){
                if(front == -1) return 0;
                int size = (capacity-front+rear+1)%capacity;
                if(size == 0) {
                        return capacity;
                }else     return size;
        }
        private void resizeQueue(){
                int initCapacity = capacity;
                capacity*=2;
                int[] oldArray = array;
                array = new int[this.capacity];
                for(int i=0;i<oldArray.length;i++){
                        array[i] = oldArray[i];
                }
                if(rear<front){
                        for(int i=0;i<front;i++){
                                array[i+initCapacity] = this.array[i];
                                array[i] = null;
                        }
                        rear = rear + initCapacity;
                }
        }
        public void enQueue(int data){
                if(isFull())
                        resizeQueue();
                rear = (rear+1)%capacity;
                array[rear] = data;
                if(front == -1)
                        front = rear;
        }
        public int deQueue(){
                int data = null;
                if(isEmpty())
                        throw new EmptyQueueException("Queue Empty");
                else{
                        data = array[front];
```

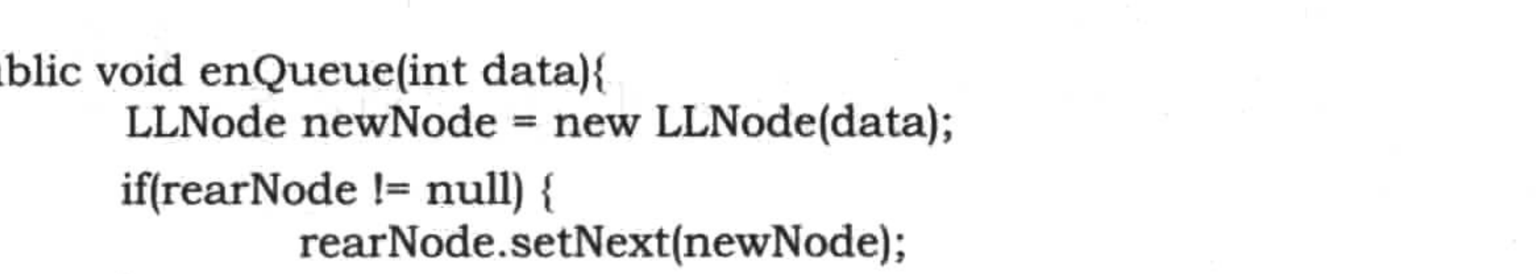
```
                    if(front == rear) front = rear = -1;
                    else front = (front+1)%capacity;
                }
                return data;
        }
}
```

**性能：** 假设 $n$ 为队列中元素的个数。

| | |
|---|---|
| 空间复杂度(用于 $n$ 次 enQueue 操作) | $O(n)$ |
| enQueue ()的时间复杂度 | $O(1)$(平均) |
| deQueue ()的时间复杂度 | $O(1)$ |
| getQueueSize ()的时间复杂度 | $O(1)$ |
| isEmpty()的时间复杂度 | $O(1)$ |
| isFull()的时间复杂度 | $O(1)$ |

### 4. 基于链表实现队列

实现队列的另一种方法是使用链表。通过在链表末端插入元素的方法实现入队(EnQueue)操作。通过删除链表表头元素的方法实现出队操作(DeQueue)。




```
public class LLQueue{
        private LLNode frontNode; //represents headNode
        private LLNode rearNode; //represents lastNode
        private LLQueue(){
                this.frontNode = null;
                this.rearNode = null;
        }
        public static LLQueue createQueue(){
                return new LLQueue();
        }
        public boolean isEmpty(){
                return (frontNode == null);
        }
        public void enQueue(int data){
                LLNode newNode = new LLNode(data);
                if(rearNode != null) {
                        rearNode.setNext(newNode);
                }
                rearNode = newNode;
                if(frontNode == null) {
                        frontNode = rearNode;
                }
}
public int deQueue(){
        int data = null;
        if(isEmpty()){
                throw new EmptyQueueException("Queue Empty");
        }else{
                data = frontNode.getData();
```

```
                frontNode = frontNode.getNext();
            }
            return data;
        }
}
```

**性能：**假设 $n$ 为队列中元素的个数。

| | |
|---|---|
| 空间复杂度(用于 $n$ 次 enQueue 操作) | $O(n)$ |
| enQueue ()的时间复杂度 | $O(1)$(平均) |
| deQueue ()的时间复杂度 | $O(1)$ |
| isEmpty()的时间复杂度 | $O(1)$ |
| deleteQueue ()的时间复杂度 | $O(1)$ |

### 5. 各种队列实现方法的比较

**注意：**各种队列实现方法的比较与栈类似。参见第 4 章。

## 5.7 队列的相关问题

**问题 1** 设计一个逆置队列元素的算法。要求算法在访问队列元素时，只能使用队列抽象数据类型中的操作。

**解答：**

```
public class QueueReversal {
    public static Queue reverseQueue(Queue queue){
        Stack stack = new LLStack();
        while(!queue.isEmpty()){
            stack.push(queue.deQueue());
        }
        while(!stack.isEmpty()){
            queue.enQueue(stack.pop());
        }
        return queue;
    }
}
```

时间复杂度为 $O(n)$。

**问题 2** 如何使用两个栈来实现队列？

**解答：**假设用来实现队列的两个栈分别为 S1 和 S2。下面定义入队和出队操作。

**入队(enQueue)算法：**

- 只需将元素压入栈 S1。

时间复杂度为 $O(1)$。

**出队(deQueue)算法：**

- 如果栈 S2 不为空，那么对 S2 执行出栈操作并返回出栈的元素。
- 如果栈 S2 为空，那么把 S1 中的所有元素移到 S2 中，然后弹出 S2 栈顶的元素并返回该元素(可以做一个小优化，即只把 S1 中的 $n-1$ 个元素移到 S2 中，然后弹出 S1 的第 $n$ 个元素并返回该元素)。
- 如果栈 S1 也为空，那么抛出错误。

时间复杂度：分析算法可知，如果栈 S2 为非空，那么时间复杂度为 $O(1)$。如果栈

S2 为空，那么需要将元素从栈 S1 移到栈 S2。但是，如果仔细观察可以看出，移到栈 S2 的元素个数与 S2 出栈元素的个数是相等的。因此，在这种情况下，出栈操作的平均复杂度为 $O(1)$。出队操作的平摊时间复杂度为 $O(1)$。

```
public class QueueWithTwoStacks {
        Stack stack1;
        Stack stack2;
        public QueueWithTwoStacks(){
                stack1 = new LLStack();
                stack2 = new LLStack();
        }
//default implementation
public boolean isEmpty(){
        if(stack2.isEmpty()){
                while(!stack1.isEmpty()){
                        stack2.push(stack1.pop());
                }
        }
        return stack2.isEmpty();
}
public void enQueue(Object data){
        stack1.push(data);
}
public Object deQueue(){
        if(!stack2.isEmpty()){
                return stack2.pop();
        }else{  while(!stack1.isEmpty()){
                        stack2.push(stack1.pop());
                }
                return stack2.pop();
        }
    }
}
```

**问题 3** 如何使用两个队列来高效地实现一个栈，并分析该栈基本操作的时间复杂度。

**解答：** 假设用来实现栈的两个队列分别为 Q1 和 Q2。只需为栈定义入栈和出栈操作即可。

在下面给出的算法中，必须确保有一个队列总是空的。

**入栈(push)算法：** 在任何一个非空队列中插入元素。

- 检查队列 Q1 是否为空。如果 Q1 为空，那么对 Q2 执行入队操作。
- 否则对 Q1 执行入队操作。

时间复杂度为 $O(1)$。

**出栈(pop)算法：** 将 $n-1$ 个元素移到另一个队列，删除当前队列中的最后一个元素来完成出栈操作。

- 如果队列 Q1 为非空，那么从 Q1 移 $n-1$ 个元素到 Q2 中，然后对 Q1 中的最后一个元素执行出队操作并返回该元素。
- 如果队列 Q2 为非空，那么从 Q2 移 $n-1$ 个元素到 Q1 中，然后对 Q2 中的最后一个元素执行出队操作并返回该元素。

时间复杂度：出栈(pop)操作的时间复杂度为 $O(n)$。因为每次调用出栈操作时，都

需要从一个队列向另一个队列转移元素。

```
public class StackWithTwoQueues{
        LLQueue queue1;
        LLQueue queue2;
        public StackWithTwoQueues(){
                queue1 = new LLQueue();
                queue1 = new LLQueue();
        }
        public void push(int data){
                if(queue1.isEmpty())
                        queue2.enQueue(data);
        else    queue1.enQueue(data);
}
public int Pop() {
        int i, size;
        if(queue2.isEmpty()){
                size = queue1.getQueueSize();
                i = 0;
                while(i < size-1) {
                        queue2.enQueue(queue1.deQueue());
                        i++;
                }
                return queue1.deQueue();
        }
        else {  size = queue2.getQueueSize();
                while(i < size-1) {
                        queue1.enQueue(queue2.deQueue());
                        i++;
                }
                return queue2.deQueue();
        }
    }

}
```

**问题 4　滑动窗口最大值问题：**给定一个滑动窗口大小为 $w$ 的数组 $A[]$，该滑动窗口从数组的最左边向最右边移动。假设只能看到在窗口中的 $w$ 个数字，且每次窗口向右移动一个位置。例如：假设数组为[1 3 −1 −3 5 3 6 7]，窗口大小 $w$ 等于 3。

| 窗口位置 | 最大值 |
|---|---|
| [1 3 −1]−3 5 3 6 7 | 3 |
| 1[3 −1 −3]5 3 6 7 | 3 |
| 1 3[−1 −3 5]3 6 7 | 5 |
| 1 3 −1[−3 5 3]6 7 | 5 |
| 1 3 −1 −3[5 3 6]7 | 6 |
| 1 3 −1 −3 5[3 6 7] | 7 |

**输入：**一个长数组 $A[]$和窗口大小 $w$。**输出：**数组 $B[]$，其中 $B[i]$的值为 $A[i]$到 $A[i+w-1]$之间的最大值。**要求：**找到求 $B[i]$的最佳方法。

**解答：**可以使用双端队列(一种在两端都支持插入和删除的队列)来求解这个问题。具体算法参见第 7 章。

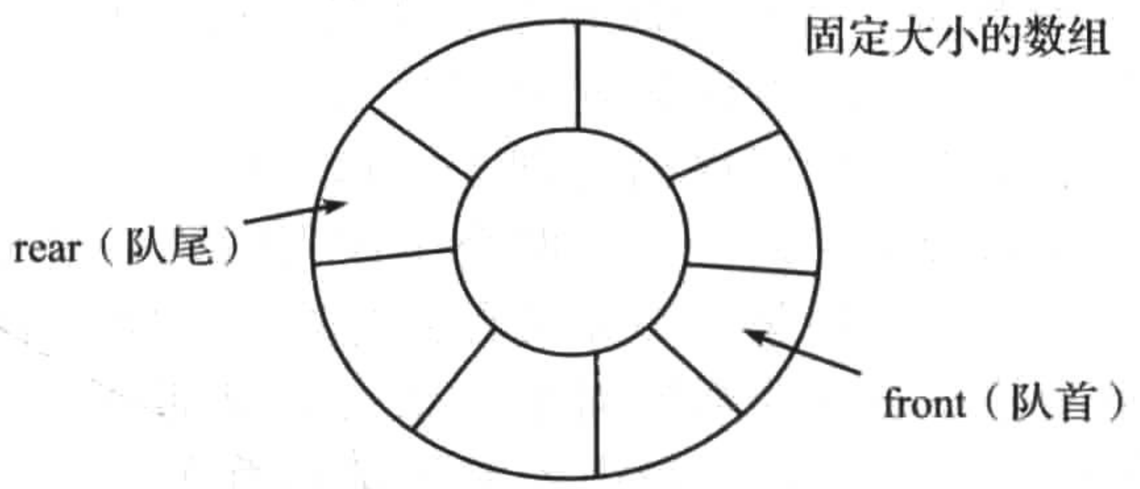
**问题 5**　给定一个含有 $n$ 个元素的队列 Q，现要求把 Q 中的元素移到栈 S(初始为空)

中，使得Q的队首元素保存在栈顶且其他元素按照在Q中的顺序保存在S中。请利用队列的出队和入队操作以及栈的入栈和出栈操作，设计一个时间复杂度为$O(n)$的有效算法，并且算法的额外空间开销为常数。

**解答：** 假设队列Q中的元素为$a_1$，$a_2$，…，$a_n$。首先将队列中的所有元素依次出队并压入栈中，那么栈顶元素为$a_n$，栈底元素为$a_1$。因为每次出队和入栈操作是常数时间开销，所以上述移动过程的时间开销为$O(n)$。这时队列已空。然后，将栈中的元素依次出栈并入队，这个过程的时间开销为$O(n)$。最后将队列中的元素依次出队并入栈，那么栈顶的元素即为$a_1$，这个过程的时间开销也为$O(n)$。因为算法复杂度的计算可以忽略常数因子，所以整个算法的时间复杂度为$O(n)$。算法所需的额外存储开销仅为临时保存一个元素的空间即可。

**问题6** 用循环数组$A[0..n-1]$来构建队列，且队首(front)和队尾(rear)的定义与一般队列一致。假设数组中有$n-1$个有效位置用于保存元素(剩下的一个位置用于检测队列的空或满状态)。请给出利用rear、front和$n$，计算队列中当前元素个数的公式。

**解答：** 下图清晰地描述了如何利用循环数组构建队列。




- 队尾(rear)在队首(front)顺时针方向上的某个位置。
- 当有元素入队时，将rear沿着顺时针方向移动一个位置，并将新元素保存在该位置。
- 当元素出队时，将front沿着顺时针方向移动一个位置。
- 随着元素入队和出队，队列沿着顺时针方向不断移动。
- 仔细检查队列是空还是满。
- 分析各种可能的情况后(画一些图来观察当队列为空、部分填充和完全填满时，front和rear的位置)，可以得到队列中当前元素个数的计算公式为：

$$\text{元素的个数} = \begin{cases} \text{rear} - \text{front} + 1 & \text{rear} = \text{front} \\ \text{rear-front} + n & \text{否则} \end{cases}$$

**问题7** 如果需要反向输出队列中的元素，哪种数据结构最合适？

**解答：** 栈。

**问题8** 给定一个整数栈，如何检查栈中每对相邻数字是否是连续的。每对数字的值可以是递增或递减的。如果栈中元素的个数是奇数，那么组对时忽略栈顶元素。例如，假设栈中元素为[4，5，－2，－3，11，10，5，6，20]，那么算法应该输出真，因为每对二元组(4，5)、(－2，－3)、(11，10)和(5，6)都是连续的数字。

**解答：**

```
public static boolean checkStackPairwiseOrder(Stack<Integer> s) {
    Queue<Integer> q = new LinkedList<Integer>();
    boolean pairwiseOrdered = true;
    while (!s.isEmpty())
```

```
        q.add(s.pop());
    while (!q.isEmpty())
        s.push(q.remove());
    while (!s.isEmpty()) {
        int n = s.pop();
        q.add(n);
        if (!s.isEmpty()) {
            int m = s.pop();
            q.add(m);
            if (Math.abs(n - m) != 1) {
                pairwiseOrdered = false;
            }
        }
    }
        while (!q.isEmpty())
            s.push(q.remove());
        return pairwiseOrdered;
    }
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 9** 给定一个整数队列，如何把队列中前半部分和后半部分的元素相互交叉，完成队列中元素的重新排列。例如，假设队列为[11，12，13，14，15，16，17，18，19，20]，那么其前半部分和后半部分的子序列分别为[11，12，13，14，15]和[16，17，18，19，20]。以交替方式合并形成交叉对序列：两个子序列的第一个元素合并为第一个交叉对(11 和 16)，然后是两个子序列的第二个元素合并为第二个交叉对(12 和 17)，第三个交叉对为(13 和 18)，以此类推。在每个数字对中，来自前半部分子序列的元素在前，来自后半部分子序列的元素在后。最后，队列中元素重新排列如下：[11，16，12，17，13，18，14，19，15，20]。

**解答：**

```
public void interLeavingQueue(Queue <Integer> q) {
    if (q.size() % 2 != 0)
        throw new IllegalArgumentException();
    Stack<Integer> s = new ArrayStack<Integer>();
    int halfSize = q.size() / 2;
    for (int i = 0; i< halfSize; i++)
        s.push(q.dequeue());
    while (!s.isEmpty())
        q.enqueue(s.pop());
    for (int i = 0; i< halfSize; i++)
        q.enqueue(q.dequeue());
    for (int i = 0; i < halfSize; i++)
        s.push(q.dequeue());
    while (!s.isEmpty()) {
        q.enqueue(s.pop());
        q.enqueue(q.dequeue());
    }
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 10** 给定一个整数 $k$ 和一个整数队列，如何把队列中前 $k$ 个元素逆置，其余的元素次序保持不变？例如，如果 $k$ 等于 4，队列中元素序列为[10，20，30，40，50，60，70，80，90]，那么应该输出[40，30，20，10，50，60，70，80，90]。

**解答：**

```
public static void reverseQueueFirstKElements(int k, Queue<Integer> q) {
    if (q == null || k > q.size()) {
        throw new IllegalArgumentException();
    }
    else if (k > 0) {
         Stack<Integer> s = new Stack<Integer>();
         for (int i = 0; i < k; i++) {
             s.push(q.remove());
        }
        while (!s.isEmpty()) {
             q.add(s.pop());
        }
        for (int i = 0; i < q.size() - k; i++) { //逆置剩余元素
             q.add(q.remove());
        }
    }
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

Chapter 6

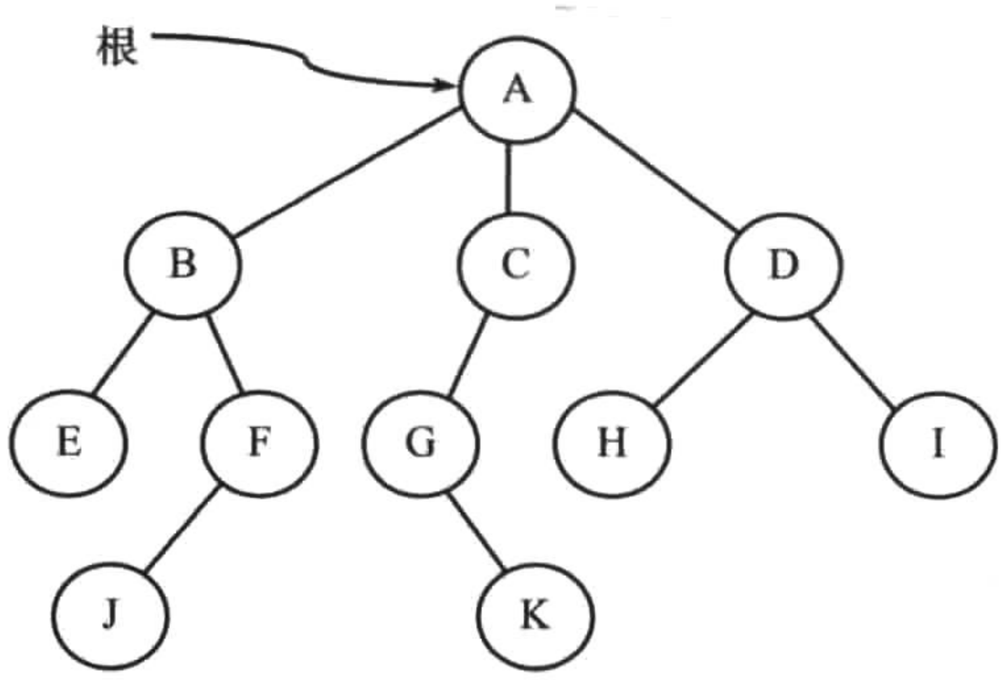
# 第6章 树

## 6.1 什么是树

树是一种类似于链表的数据结构，不过链表的结点是以线性方式简单地指向其后继结点，而树的一个结点可以指向许多个结点。树是一种典型的非线性结构。树结构是表达具有层次特性的图结构的一种方法。

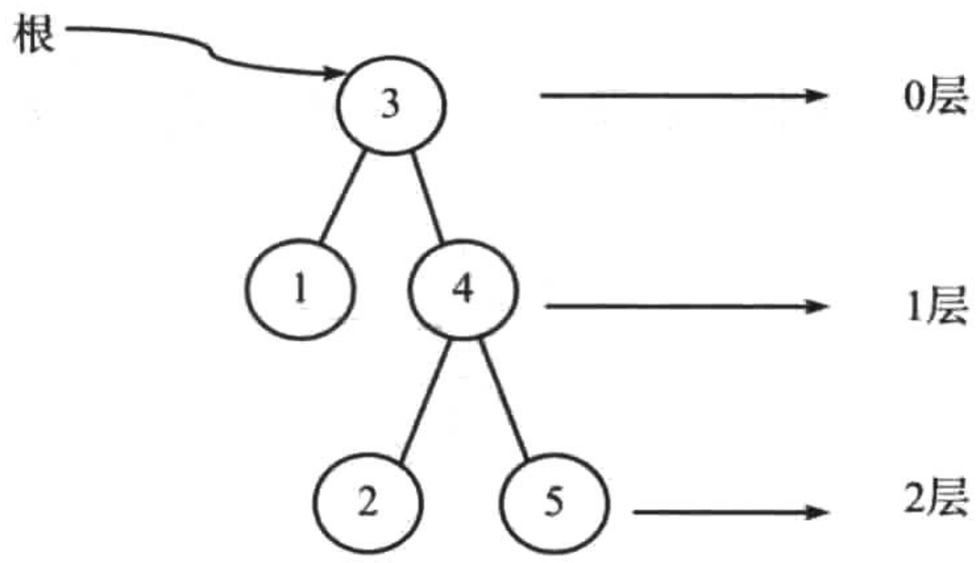
对于树 ADT(抽象数据类型)，元素的顺序不是考虑的重点。如果需要用到元素的顺序信息，那么可以使用链表、栈、队列等线性数据结构。

## 6.2 术语

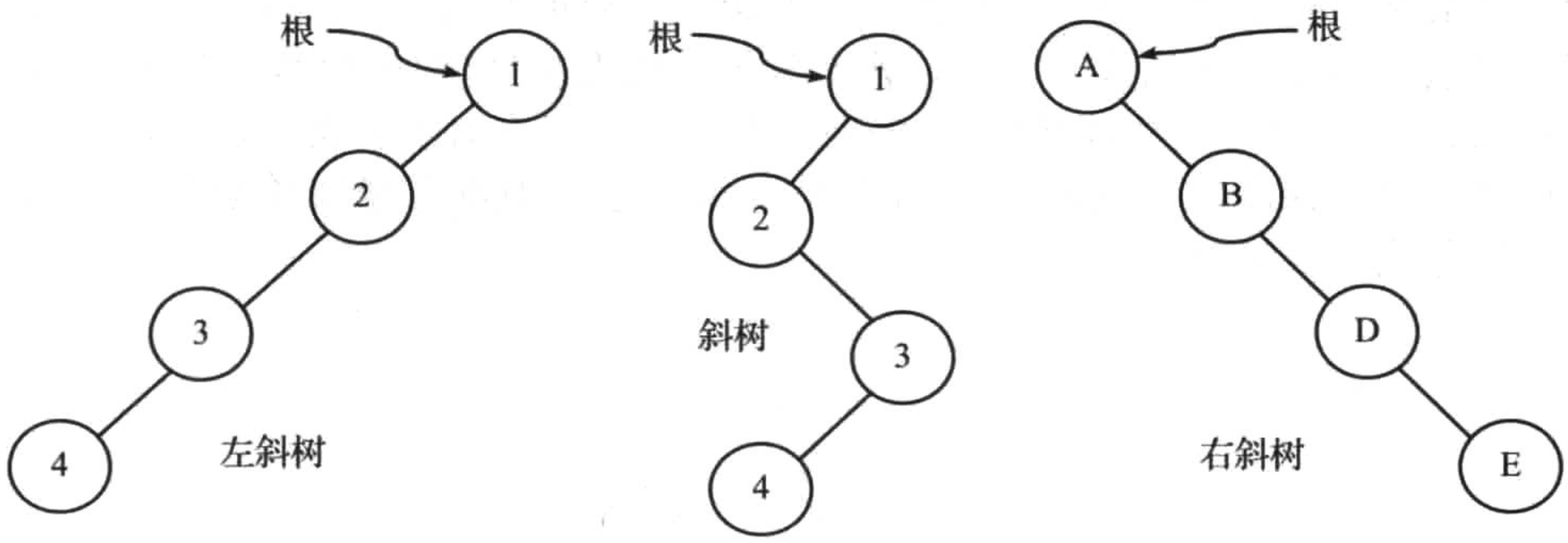



- 根结点：根结点是一个没有双亲结点的结点。一棵树中最多有一个根结点(如上图的结点 *A* 就是根结点)。
- 边：边表示从双亲结点到孩子结点的链接(如上图中所有的链接)。
- 叶子结点：没有孩子结点的结点叫作叶子结点(如 *E*、*J*、*K*、*H* 和 *I*)。

- 兄弟结点：拥有相同双亲结点的所有孩子结点叫作兄弟结点($B$、$C$、$D$ 是 $A$ 的兄弟结点，$E$、$F$ 是 $B$ 的兄弟结点)
- 祖先结点：如果存在一条从根结点到结点 $q$ 的路径，且结点 $p$ 出现在这条路径上，那么就可以把结点 $p$ 叫作结点 $q$ 的祖先结点，结点 $q$ 也叫作 $p$ 的子孙结点。例如，$A$、$C$ 和 $G$ 是 $K$ 的祖先结点。
- 结点的大小：结点的大小是指子孙的个数，包括其自身。(子树 $C$ 的大小为 3)。
- 树的层：位于相同深度的所有结点的集合叫作树的层($B$、$C$ 和 $D$ 具有相同的层)。根结点位于 0 层。



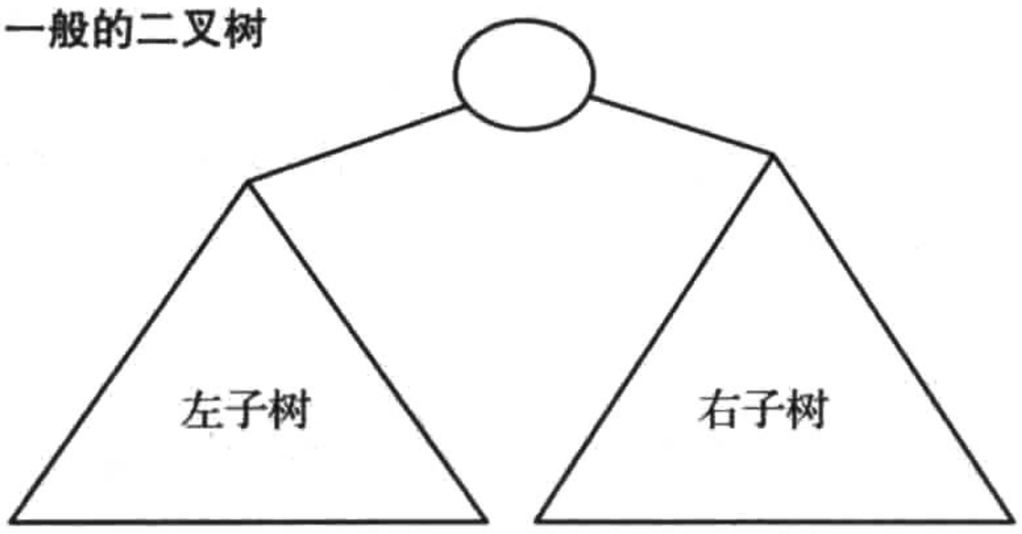

- 结点的深度：是指从根结点到该结点的路径长度($G$ 点的深度为 2，$A$—$C$—$G$)。
- 结点的高度：是指从该结点到最深结点的路径长度。树的高度是指从根结点到树中最深结点的路径长度。只含有根结点的树的高度为 0。在前面的例子中，$B$ 的高度为 2($B$—$F$—$J$)。
- 树的高度：是树中所有结点高度的最大值，树的深度是树中所有结点深度的最大值。对于同一棵树，其深度和高度是相同的。但是对于各个结点，其深度和高度不一定相同。
- 斜树：如果树中除了叶子结点外，其余每一结点只有一个孩子结点，则这种树称作斜树。对于每个结点仅有一个左孩子结点的树叫作左斜树。类似地，对于每个结点仅有右孩子结点的树叫作右斜树。



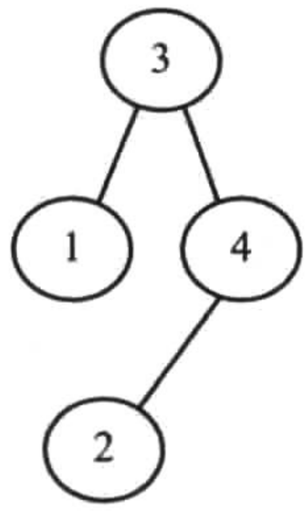

## 6.3 二叉树

如果一棵树中的每个结点有 0、1 或者 2 个孩子结点，那么这棵树就称为二叉树。空树也是一棵有效的二叉树。一棵二叉树可以看作由根结点和两棵不相交的子树(分别称为左子树和右子树)组成。如下图所示。

一般的二叉树

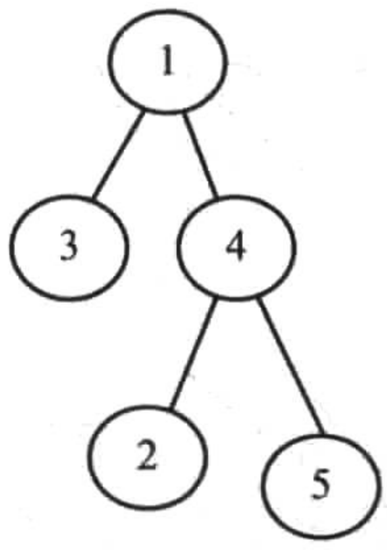



例子

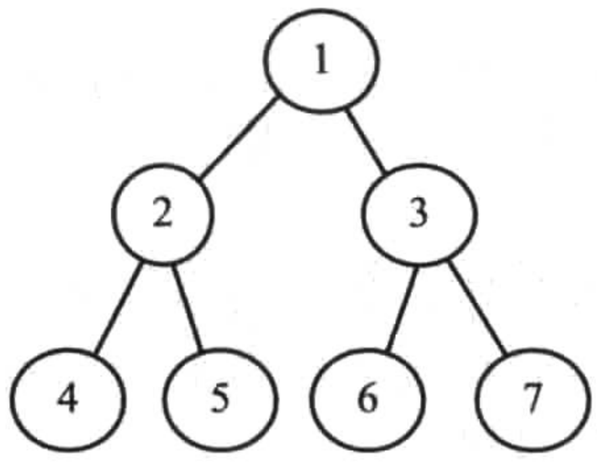



1. 二叉树的类型

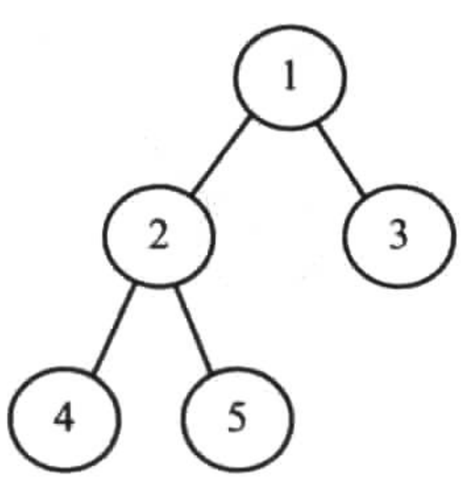
**严格二叉树**：二叉树中的每个结点要么有两个孩子结点，要么没有孩子结点。




**满二叉树**：二叉树中的每个结点恰好有两个孩子结点且所有叶子结点都在同一层。

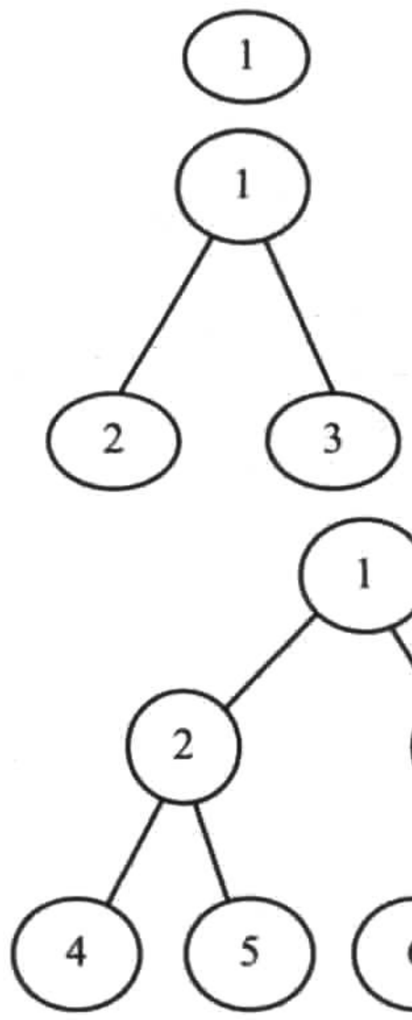



**完全二叉树**：在定义完全二叉树之前，假定二叉树的高度为 $h$。对于完全二叉树，如果将所有结点从根结点开始从左至右，从上至下，依次编号(假定根结点的编号为 1)，那么将得到从1～$n$($n$ 为结点总数)的完整序列。在遍历过程中对于空指针也应赋予编号。如果所有叶子结点的深度为 $h$ 或 $h-1$，且在结点编号序列中没有漏掉任何数字，那么这样的二叉树叫作完全二叉树。




2. 二叉树的性质

为了讨论二叉树的下述性质，假定树的高度为 $h$，且根结点的深度为 0。



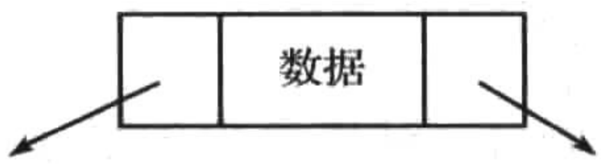

| 高度 | 层 $h$ 上结点的个数 |
| --- | --- |
| $h$=0 | $2^0$=1 |
| $h$=1 | $2^1$=2 |
| $h$=2 | $2^2$=4 |

从上图可以得到如下性质：

- 满二叉树的结点个数 $n$ 为 $2^{h+1}-1$。因为该树共有 $h$ 层，所以每一层的结点个数都是满的，即有$[2^0+2^1+2^2+\cdots+2^h=2^{h+1}-1]$。
- 完全二叉树的结点个数为 $2^h \sim 2^{h+1}-1$。更多信息，请参见第 7 章。
- 满二叉树的叶子结点个数是 $2^h$。
- 对于 $n$ 个结点的完全二叉树，空指针的个数为 $n+1$。

### 3. 二叉树的结构

接下来定义二叉树的结构。为了简单起见，假定结点的数据为整数。表示一个结点(包含数据)的方法之一是定义两个指针，一个指向左孩子结点，另一个指向右孩子结点，中间为数据字段(如下图所示)。




或者




```
public class BinaryTreeNode {
        private int data;
        private BinaryTreeNode left;
        private BinaryTreeNode right;
        public int getData() {
                return data;
        }
        public void setData(int data) {
                this.data = data;
        }
        public BinaryTreeNode getLeft() {
                return left;
        }
        public void setLeft(BinaryTreeNode left) {
                this.left = left;
        }
        public BinaryTreeNode getRight() {
                return right;
        }
```

```
    public void setRight(BinaryTreeNode right) {
        this.right = right;
    }
}
```

**注意**：在树中，默认的流向是从双亲结点指向孩子结点，但并不强制表示为有向边。在后续讨论中，假定如下两种表示方式是相同的。







**4. 二叉树的操作**

a)基本操作

- 向树中插入一个元素。
- 从树中删除一个元素。
- 查找一个元素。
- 遍历树。

b)辅助操作

- 获取树的大小。
- 获取树的高度。
- 获取其和最大的层。
- 对于给定的两个或多个结点，找出它们的最近公共祖先(Least Common Ancestor，LCA)。

**5. 二叉树的应用**

下面是二叉树的一些重要应用：

- 编译器中的表达式树。
- 用于数据压缩算法中的赫夫曼编码树。
- 支持在集合中查找、插入和删除，其平均时间复杂度为 $O(\log n)$的二叉搜索(或称为查找)树(BST)。
- 优先队列(PQ)，它支持以对数时间(最坏情况下)对集合中的最小(或最大)数据元素进行搜索和删除。

## 6.4 二叉树的遍历

为了对树结构进行处理，需要一种机制来遍历树中的结点，这是本节要讨论的主要内容。访问树中所有结点的过程叫作树遍历。在遍历过程中，每个结点只能被处理一次，尽管其有可能被访问多次。我们已经知道，在线性数据结构(例如，链表、栈、队列)中，数据元素是以顺序方式被访问的。但是，在树结构中，有多种不同的方式。树遍历类似于在树中进行搜索操作，遍历的目标是按某种特定的顺序访问树的所有结点，且遍历过程中所有的结点均需要处理，而搜索操作在找到指定结点时就会停止。

**1. 遍历的方式**

从二叉树的根结点开始遍历，需要执行 3 个主要步骤。这 3 个步骤执行的不同顺序定义了树的不同遍历类型。这 3 个步骤是：在当前结点上执行一个动作(对应于访问当前结点，用符号“D”表示)；遍历该结点的左子树(用符号“L”表示)；遍历该结点的右子树

(用符号“R”表示)。这个过程很容易利用递归来实现。基于以上定义，有6种可能方法来实现树的遍历：

1) LDR：先遍历左子树，访问当前结点，再遍历右子树。
2) LRD：先遍历左子树，再遍历右子树，访问当前结点。
3) DLR：访问当前结点，遍历左子树，再遍历右子树。
4) DRL：访问当前结点，遍历右子树，遍历左子树。
5) RDL：遍历右子树，访问当前结点，遍历左子树。
6) RLD：遍历右子树，遍历左子树，访问当前结点。

### 2. 遍历的分类

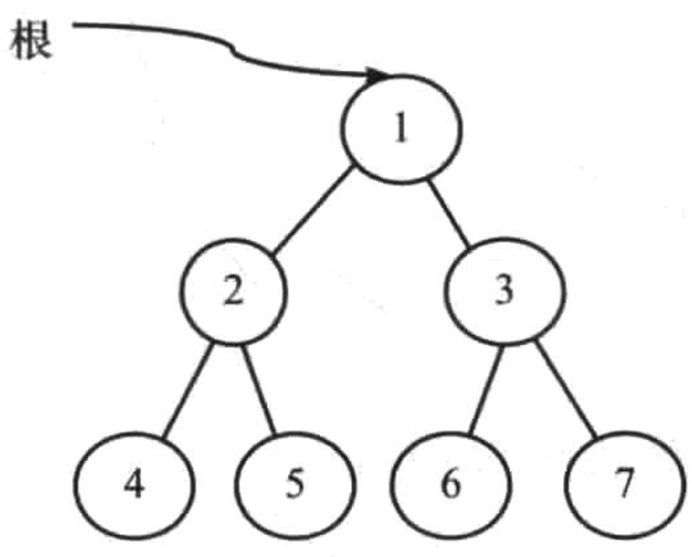
根据实体(结点)处理顺序的不同，可以定义不同的遍历方法。遍历分类可以根据当前结点被处理的顺序来划分：如果基于当前结点(D)来分类，假设D出现在中间，那么不管L在D的左边还是R在D的左边，D都是在中间处理。同样，不管L在D的右边还是R在D的右边。基于此，6种可能情况减少到了3种[⊖]，分别是：

- 前序遍历(DLR)。
- 中序遍历(LDR)。
- 后序遍历(LRD)。

还有一种遍历方法，它不需要依赖上述顺序，即，

- 层次遍历：这种方法从图的广度优先遍历方法启发得来。

后续讨论以下图所示的树为例。




### 3. 前序遍历

在前序遍历中，每个结点都是在它的子树遍历之前进行处理。这是最容易理解的遍历方法。然而，尽管每个结点在其子树之前进行了处理，但在向下移动的过程中仍然需要保留一些信息。以上图为例，首先访问结点1，随后遍历其左子树，最后遍历其右子树。因此，当左子树遍历完后，必须要返回到其右子树来继续遍历。为了能够在左子树遍历完成后移动到右子树，必须保留根结点的信息。能够实现该信息存储的抽象数据类型显而易见是栈。由于它是LIFO结构，所以它可以以逆序来获取该信息并返回到右子树。

前序遍历可以如下定义：

- 访问根结点。
- 按前序遍历方式遍历左子树。

---

⊖ 一般规定先遍历左子树，然后才遍历右子树，即L总在R的前面。——译者注

- 按前序遍历方式遍历右子树。

利用前序遍历方法遍历上图所示的树的输出序列为：1 2 4 5 3 6 7。

```
void PreOrder(BinaryTreeNode root){
        if(root != null) {
                System.out.println(root.getData());
                PreOrder(root.getLeft());
                PreOrder(root.getRight());
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**4. 非递归前序遍历**

在递归方法中，需要采用一个栈来记录当前结点以便在完成左子树后能返回到右子树进行遍历。为了模拟相同的操作，首先处理当前结点，在遍历左子树之前，把当前结点保留到栈中，当遍历完左子树后，将该元素出栈，然后找到其右子树进行遍历。持续该过程直到栈为空。

```
void PreOrderNonRecursive(BinaryTreeNode  root){
        if(root  == null) return null;
        LLStack S = new LLStack();
        while(true){
                while(root  != null){
                        System.out.println(root.getData());
                        S.push(root);
                        root = root.getLeft();
                }
                if(S.isEmpty())
                        break;
                root = (BinaryTreeNode ) S.pop();
                root = root.getRight();
        }
        return;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**5. 中序遍历**

在中序遍历中，根结点的访问在两棵子树的遍历中间完成。中序遍历如下定义：

- 按中序遍历方式遍历左子树。
- 访问根结点。
- 按中序遍历方式遍历右子树。

基于中序遍历，上图所示树的中序遍历输出顺序为：4 2 5 1 6 3 7。

```
void InOrder(BinaryTreeNode root){
        if(root != null) {
                InOrder(root.getLeft());
                System.out.println(root.getData());
                InOrder(root.getRight);
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**6. 非递归中序遍历**

非递归中序遍历类似于前序遍历。唯一的区别是，首先要移动到结点的左子树，完

成左子树的遍历后，再将结点出栈进行处理(即发生在左子树遍历完后)。

```
void InOrderNonRecursive(BinaryTreeNode  root ){
        if(root == null) return null;
        LLStack S = new LLStack();
        while(true){
                while(root  != null){
                        S.push(root );
                        root = root.getLeft();
                }
                if(stack.isEmpty())
                        break;
                root = (BinaryTreeNode) S.pop();
                System.out.println(root.getData());
                root = root.getRight();
        }
        return;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**7. 后序遍历**

在后序遍历中，根结点的访问是在其两棵子树都遍历完后进行的。后序遍历如下定义：

- 按后序遍历左子树。
- 按后序遍历右子树。
- 访问根结点。

对上图所示的二叉树，后序遍历产生的输出序列为：4　5　2　6　7　3　1。

```
void PostOrder(BinaryTreeNode root){
        if(root != null)    {
                PostOrder(root.getLeft());
                PostOrder(root.getRight());
                System.out.println(root.getData());
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**8. 非递归后序遍历**

在前序和中序遍历中，当元素出栈后就不需要再次访问该结点了。但是在后序遍历中，每个结点需要访问两次。这就意味着，在遍历完左子树后，需要访问当前结点，之后遍历完右子树时还需要访问该当前结点。但只有在第二次访问时才处理当前结点。问题是如何区分这两次访问：到底是遍历完左子树后的返回还是遍历完右子树后的返回？

解决这个问题的方法是，当从栈中出栈一个元素时，检查这个元素与栈顶元素的右子结点是否相同。如果相同，则说明已经完成了左、右子树的遍历。此时，只需要再将栈顶元素出栈一次并输出该结点数据即可。

```
void PostOrderNonRecursive(BinaryTreeNode root){
        LLStack S = new LLStack();
        while (1) {
                if  (root != null) {
                        S.push(root);
                        root = root.getLeft();
                }
```

```
        else {
                if(S.isEmpty()) {
                        System.out.println("Stack is Empty");
                        return;
                }
                else
                        if(S.top().getRight() == null) {
                                root= S.pop();
                                System.out.println(root.getData());
                                if(root == S.top().getRight()) {
                                        System.out.println(S.top(S.getData());
                                        S.pop();
                                }
                        }
                        if(!S.isEmpty())
                                root=S.top().getRight();
                        else    root= null;
                }
        }
        S.deleteStack();
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**9. 层次遍历**

层次遍历的定义如下：

- 访问根结点。
- 在访问第 $l$ 层时，将 $l+1$ 层的结点按顺序保存在队列中。
- 进入下一层并访问该层的所有结点。
- 重复上述操作直至所有层都访问完。

对于上图所示的二叉树，层次遍历产生的输出序列为：1 2 3 4 5 6 7。

```
void LevelOrder(BinaryTreeNode root){
        BinaryTreeNode temp;
        LLQueue Q = new LLQueue();
        if(root == null)
                return;
        Q.enQueue(root);
        while(!Q.isEmpty()) {
                temp = Q.deQueue();
                //处理当前结点
                System.out.println(temp.getData());
                if(temp.getLeft())
                        Q.enQueue(temp.getLeft());
                if(temp.getRight())
                        Q.enQueue(temp.getRight());
        }
        Q.deleteQueue();
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$，因为在最坏情况下，最后一层的所有结点可能同时在队列中。

**10. 二叉树的相关问题**

**问题 1** 给出查找二叉树中最大元素的算法。

**解答：** 解决此问题的一个简单方法是：利用递归思想，分别找到左子树中的最大元

素和右子树中的最大元素，然后将它们与根结点的值进行比较，这3个值中最大的就是问题的解。这个方法利用递归很容易实现。

```
int FindMax(BinaryTreeNode  root) {
        int root_val, left, right, max = INT_MIN;
        if(root != null) {
                root_val = root.getData();
                left = FindMax(root.getLeft());
                right = FindMax(root.getRight());
                //在3个值中找出最大值
                if(left > right)
                        max = left;
                else
                        max = right;
                if(root_val > max)
                        max = root_val;
        }
        return max;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题2** 用非递归的方法实现查找二叉树中的最大元素。

**解答：** 利用层次遍历方法，在删除结点时观察其数据值是否是最大的。

```
int FindMaxUsingLevelOrder(BinaryTreeNode root){
        BinaryTreeNode temp;
        int max = INT_MIN;
        LLQueue Q = new LLQueue();
        Q.enQueue(root);
        while(!Q.isEmpty()) {
                temp = Q.deQueue();
                //所有值中的最大值
                if(max < temp.getData())
                        max =  temp.getData();
                if(temp.getLeft())
                        Q.enQueue(temp.getLeft());
                if(temp.getRight())
                        Q.enQueue(temp.getRight());
        }
        Q.deleteQueue();
        return max;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题3** 给出在二叉树中搜索某个元素的算法。

**解答：** 对于给定的二叉树，如果发现树中某结点的数据值与之相同，则返回true。递归地从树的根结点向下，比较左子树与右子树中各个结点的值来实现算法。

```
Boolean FindInBinaryTreeUsingRecursion(BinaryTreeNode root, int data) {
        Boolean temp;
        //基本情况==空树，在此情况下，数据未找到因此返回false
        if(root == null)
                return false;
        else {    //判断是否等于当前根结点的值
                if(data == root.getData())
                        return true;
```

```
                else {    //否则从子树继续递归向下搜索
                          temp = FindInBinaryTreeUsingRecursion(root.getLeft(), data)
                          if(temp != true)
                                    return temp;
                          else      return(FindInBinaryTreeUsingRecursion(root.getRight(), data));
                }
        }
        return 0;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 4** 利用非递归算法来搜索二叉树中的某个元素。

**解答：** 可以利用层次遍历方法来解决这个问题。唯一的不同是在本算法中不是输出数据而是判断根结点的数据是否等于需要搜索元素的值。

```
Boolean SearchUsingLevelOrder(BinaryTreeNode root, int data){
        BinaryTreeNode temp;
        LLQueue Q = new LLQueue();
        if(root == null) return -1;
        Q.enQueue(root);
        while(!Q.isEmpty()) {
                temp = Q.deQueue();
                //判断是否等于当前根结点的值
                if(data == root.getData())
                        return true;
                if(temp.getLeft())
                        Q.enQueue(temp.getLeft());
                if(temp.getRight())
                        Q.enQueue(temp.getRight());
        }
        Q.deleteQueue();
        return false;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 5** 实现将一个元素插入二叉树中的算法。

**解答：** 因为给定的树是二叉树，所以能在任意地方插入元素。为了插入一个元素，可以使用层次遍历方法找到一个左孩子或右孩子结点为空的结点，然后插入该元素。

```
void InsertInBinaryTree(BinaryTreeNode root, int data){
        LLQueue Q = new LLQueue();
        BinaryTreeNode temp;
        BinaryTreeNode newNode = new BinaryTreeNode();
        newNode.setLeft(null);
        newNode.setRight(null);
        if(newNode == null) {
                System.out.println("Memory Error");
                return;
        }
        if(root == null) {
                root = newNode;
                return;
        }
        Q.enQueue(root);
        while(!Q.isEmpty()) {
                temp = Q.deQueue();
                if(temp.getLeft())
```

```
                Q.enQueue(temp.getLeft());
        else {
                temp.setLeft(newNode);
                Q.deleteQueue();
                return;
        }
        if(temp.getRight())
                Q.enQueue(temp.getRight());
        else {
                temp.setRight(newNode);
                Q.deleteQueue();
                return;
                }
        }
        Q.deleteQueue();
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 6**　给出获取二叉树结点个数的算法

**解答：**递归地计算左子树和右子树的大小，再加 1(当前结点)，然后返回给其双亲结点。

```
//计算树中结点的个数
int SizeOfBinaryTree(BinaryTreeNode root) {
        if(root == null) return 0;
        else    return(SizeOfBinaryTree(root.getLeft()) + 1 +  SizeOfBinaryTree(root.getRight()));
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 7**　能利用非递归方法解决问题 6 吗?

**解答：可以**，利用层次遍历方法。

```
int SizeofBTUsingLevelOrder(BinaryTreeNode root) {
        BinaryTreeNode temp;
        LLQueue Q = new LLQueue();
        int count = 0;
        if(root == null) return 0;
        Q.enQueue(root);
        while(!Q.isEmpty()) {
                temp = Q.deQueue();
                count++;
                if(temp.getLeft())
                        Q.enQueue(temp.getLeft());
                if(temp.getRight())
                        Q.enQueue(temp.getRight());
        }
        Q.deleteQueue();
        return count;
}
```

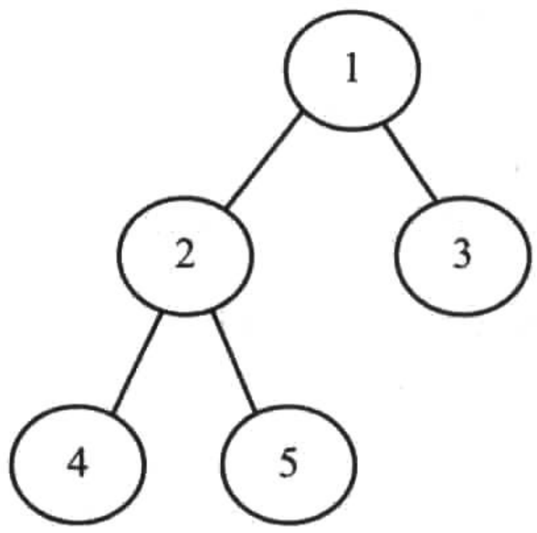
时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 8**　实现删除树的算法。

**解答：**为了删除树，需要遍历树的所有结点，然后一个一个地删除它们。但中序遍历、前序遍历、后序遍历以及层次遍历，应该选择哪种遍历方法呢?

在删除双亲结点之前，应该先删除其孩子结点。可以使用后序遍历，在删除过程中

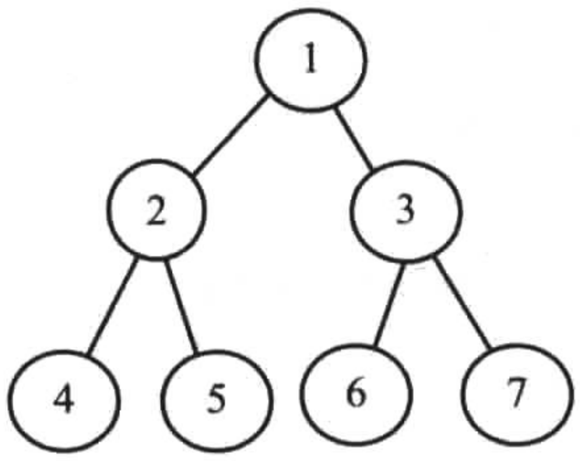
不需要存储任何信息。也可以用其他遍历方法删除树，不过需要额外的空间复杂度。下面这棵树的删除顺序为：4，5，2，3，1。




```
void DeleteBinaryTree(BinaryTreeNode root) {
        if(root == null)
                return;
        /*首先删除两棵子树*/
        DeleteBinaryTree(root.getLeft());
        DeleteBinaryTree(root.getRight());
        //仅当子树删除后再删除当前结点
        root = null;                    // 在Java中，将由垃圾回收器对其进行清理
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 9** 给出算法，逆向逐层输出树中的元素。例如，下图所示二叉树的输出顺序应为：4 5 6 7 2 3 1。




**解答：**

```
void LevelOrderTraversalInReverse(BinaryTreeNode root) {
        LLQueue Q = new LLQueue();
        LLStack S = new LLStack();
        BinaryTreeNode temp;
        if(root == null)    return;
        Q.enQueue( root);
        while(!Q.isEmpty()) {
                temp = Q.deQueue();
                if(temp.getRight())
                        Q.enQueue(temp.getRight());
                if(temp.getLeft())
                        Q.enQueue(temp.getLeft());
                S.push(temp);
        }
        while(!S.isEmpty())
                System.out.println(S.pop().getData());
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 10**　求已知二叉树高度(深度)的算法。

**解答：**递归地计算左子树和右子树的高度，然后找出这两棵子树高度中的最大值，再加 1，就是树的高度。这类似于前序遍历(或者图的深度优先算法)。

```
int HeightOfBinaryTree(BinaryTreeNode   root) {
        int leftheight, rightheight;
        if(root == null) return 0;
        else {    /*计算每棵子树的深度*/
                leftheight = HeightOfBinaryTree(root.getLeft());
                rightheight = HeightOfBinaryTree(root.getRight());
                if(leftheight > rightheight)
                        return (leftheight + 1);
                else      return (rightheight + 1);
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 11**　能否利用非递归方法解决问题 10?

**解答：可以**，使用层次遍历算法，这类似于图的广度优先算法。空指针为层次遍历结束的标志。

```
int FindHeightofBinaryTree(BinaryTreeNode root) {
        int level=1;
        LLQueue Q = new LLQueue();
        if(root == null)  return 0;
Q.enQueue(root);
//第一层结束
Q.enQueue(null);
while(!Q.isEmpty()) {
        root=Q.deQueue();
        //当前层遍历结束
        if(root==null) {
                //为下一层增加一个标记
                if(!Q.isEmpty())
                        Q.enQueue(null);
                level++;
        }
        else {   if(root.getLeft())
                        Q.enQueue( root.getLeft());
                if(root.getRight())
                        Q.enQueue( root.getRight());
        }
        return level;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 12**　实现查找二叉树中最深结点的算法。

**解答：**

```
BinaryTreeNode DeepestNodeinBinaryTree(BinaryTreeNode root) {
        BinaryTreeNode temp;
        LLQueue Q = new LLQueue();
        if(root == null) return null;
        Q.enQueue(root);
```

```
while(!Q.isEmpty()) {
        temp = Q.deQueue();
        if(temp.getLeft())
                Q.enQueue(temp.getLeft());
        if(temp.getRight())
                Q.enQueue(temp.getRight());
        }
        Q.deleteQueue();
        return temp;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 13** 实现删除二叉树中某元素的算法(假设数据是给定的)。

**解答：** 二叉树中删除结点可以按照如下步骤来实现：

- 从根结点开始，找到要删除的结点。
- 找到树中最深的结点。
- 用最深的结点替换要删除的结点。
- 然后删除最深的结点。

**问题 14** 用非递归算法获取二叉树中叶子结点的个数。

**解答：** 左孩子结点和右子结点都为空的结点就是叶子结点。

```
int NumberOfLeavesInBTusingLevelOrder(BinaryTreeNode root) {
        BinaryTreeNode temp;
        LLQueue Q = new LLQueue();
        int count = 0;
        if(root == null)
                return 0;
        Q.enQueue(root);
        while(!Q.isEmpty()) {
                temp = Q.deQueue();
                if(!temp.getLeft() && !temp.getRight())
                        count++;
                else {  if(temp.getLeft())
                                Q.enQueue(temp.getLeft());
                        if(temp.getRight())
                                Q.enQueue(temp.getRight());
                }
        }
        Q.deleteQueue();
        return count;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 15** 用非递归算法实现查找二叉树中满结点的个数。

**解答：** 既有左孩子结点，又有右孩子结点的结点叫作满结点，这些结点的个数就是问题的解。

```
int NumberOfFullNodesInBTusingLevelOrder(BinaryTreeNode root) {
        BinaryTreeNode temp;
        LLQueue Q = new LLQueue();
        int count = 0;
        if(root == null)
                return 0;
        Q.enQueue(root);
```

```
        while(!Q.isEmpty()) {
                temp = Q.deQueue();
                if(temp.getLeft() && temp.getRight())
                        count++;
                if(temp.getLeft())
                        Q.enQueue(temp.getLeft());
                if(temp.getRight())
                        Q.enQueue(temp.getRight());
        }
        Q.deleteQueue();
        return count;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 16**　用非递归算法实现查找二叉树中半结点(仅有一个孩子结点的结点)的个数。

**解答：**半结点就是只有左孩子结点或者只有右孩子结点的结点，半结点只能有一个孩子结点。

```
int NumberOfHalfNodesInBTusingLevelOrder(BinaryTreeNode root) {
        BinaryTreeNode temp;
        LLQueue Q = new LLQueue();
        int count = 0;
        if(root == null) return 0;
        Q.enQueue(root);
        while(!Q.isEmpty()) {
                temp = Q.deQueue();
                //可以使用如下判断条件来代替两个temp.getLeft() ^ temp.getRight()
                if(!temp.getLeft() && temp.getRight() || temp.getLeft() && !temp.getRight())
                        count++;
                if(temp.getLeft())
                        Q.enQueue(temp.getLeft());
                if(temp.getRight())
                        Q.enQueue(temp.getRight());
        }
        Q.deleteQueue();
        return count;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 17**　对于给定的两棵树，判断它们的结构是否相同，如果相同就返回 true。

**解答：**

**算法：**

- 如果两棵树都为空树，则返回 true。
- 如果两棵树都不为空，则比较数据并递归地判断左子树与右子树是否相同。

```
//如果都相同则返回true
Boolean AreStructurullySameTrees(BinaryTreeNode root1, BinaryTreeNode root2) {
        //若两棵树都为空→1
        if(root1== null && root2== null)
                return true;
        if(root1== null || root2== null)
                return false;
        //若都不为空→进行比较
        return(root1.getData() == root2.getData() &&
                AreStructurullySameTrees(root1.getLeft(), root2.getLeft()) &&
                AreStructurullySameTrees(root1.getRight(), root2.getRight()));
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 18** 求二叉树直径的算法。树的直径(有时也叫作树的宽度)就是树中两个叶子结点之间的最长路径中的结点个数。

**解答：** 为了获取树的直径，首先需要递归地计算左子树的直径和右子树的直径。找出两者中最大值，再加 1 返回。

```
//假设diamerter是一个静态变量
int DiameterOfTree(BinaryTreeNode root, int diameter) {
        int left, right;
        if(root == null) return 0;
        left = DiameterOfTree(root.getLeft(), diameter);
        right = DiameterOfTree(root.getRight(), diameter);
        if(left + right > diameter)
                diameter = left + right;
        return Math.max(left, right)+1;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 19** 找出二叉树中同一层结点数据之和最大的层。

**解答：** 逻辑上类似于查找二叉树的层数。唯一不同的是，还需要跟踪每一层结点的数据和。

```
int FindLevelwithMaxSum(BinaryTreeNode root) {
        BinaryTreeNode temp;
        int level=0, maxLevel=0;
        LLQueue Q = new LLQueue();
int currentSum = 0, maxSum = 0;
if(root == null) return 0;
Q.enQueue(root);
Q.enQueue(null);                                    //第一层结束
while(!Q.isEmpty()) {
        temp =Q.deQueue();
        //若当前层遍历完成则比较和
        if(temp == null) {
                if(currentSum> maxSum) {
                        maxSum = currentSum;
                        maxLevel = level;
                }
                currentSum = 0;
                //将标记下一层结束的指示器置入队尾
                if(!Q.isEmpty())
                        Q.enQueue(null);
                level++;
                }
                else {    currentSum     += temp.getData();
                        if(temp.getLeft())
                                temp.enQueue(temp.getLeft());
                        if(root.getRight())
                                temp.enQueue(temp.getRight());
                }
        }
        return maxLevel;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 20** 对于一棵给定的二叉树，输出所有从根结点到叶子结点的路径。

**解答：** 参考函数中的注释。

```
public void printPaths() {
        int[] path = new int[256];
        printPaths(node, path, 0);
}
private void printPaths(BinaryTreeNode  node, int[] path, int pathLen) {
        if (node == null) return;
        //将该结点添加到路径数组中
        path[pathLen] = node.getData();
        pathLen++;
        //当前为叶子结点，因此输出到这里的路经
        if (node.getLeft() == null && node.getRight() == null) {
                printArray(path, pathLen);
        }
        else {    //否则，继续遍历两棵子树
                printPaths(node.getLeft(), path, pathLen);
                printPaths(node.getRight(), path, pathLen);
        }
}
private void printArray(int[] ints, int len) {
        for (int i=0; i<len; i++) {
                System.out.print(ints[i] + " ");
        }
        System.out.println();
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$上的所有结点，用于栈递归。

**问题 21** 给出一个算法，判断是否存在路径的数据和等于给定值。也就是说，判断是否存在一条从根结点到任意结点的路径，其所经过结点的数据和为给定值。

**解答：** 对这个问题的策略是：递归地实现如下步骤，在调用其孩子结点之前，先把 sum 值减去该结点的值。然后在运行过程中检查 sum 值是否为 0。

```
public boolean hasPathSum(int sum) {
        return(hasPathSum(root, sum));
}
boolean hasPathSum(BinaryTreeNode node, int sum) {
        //如果所有结点已被访问且sum==，则返回true
        if (node == null)
                return (sum == 0);
        else {//否则检查两棵子树
                int subSum = sum – node.getData();
                return(hasPathSum(node.getLeft(),subSum)||hasPathSum(node.getRight(),subSum));
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 22** 实现算法，求出二叉树所有结点数据之和。

**解答：** 递归地求出左子树与右子树的和，然后将它们的值加到当前结点的值上。

```
int Add(BinaryTreeNode  root) {
        if(root == null) return 0;
        else return(root.getData() + Add(root.getLeft()) +  Add(root.getRight()));
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

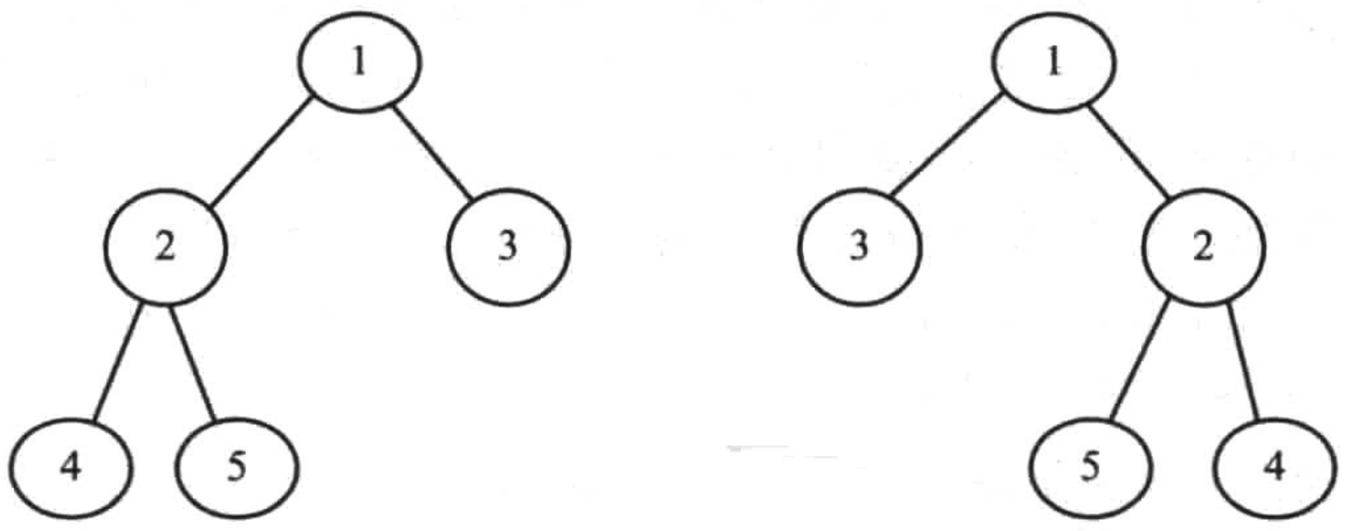
**问题 23** 能否用非递归方法求问题 22？

**解答：** 只需要对层次遍历算法进行简单修改。即每次从队列中删除元素时，将该结点的值加到 sum 变量上。

```
int SumofBTusingLevelOrder(BinaryTreeNode root){
        BinaryTreeNode temp;
        LLQueue Q = new LLQueue();
        int sum = 0;
        if(root == null)
            return 0;
        Q.enQueue(root);
        while(!Q.isEmpty()) {
            temp = Q.deQueue();
            sum += temp.getData();
            if(temp.getLeft())
                Q.enQueue(temp.getLeft());
            if(temp.getRight())
                Q.enQueue(temp.getRight());
        }
        Q.deleteQueue();
        return sum;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 24** 实现将一棵树转换为其镜像的算法。树的镜像是指相互交换非叶子结点的左子树、右子树，得到的另一棵树。下面的两棵树就互为镜像。




**解答：**

```
BinaryTreeNode MirrorOfBinaryTree(BinaryTreeNode root) {
        BinaryTreeNode  temp;
        if(root) {
            MirrorOfBinaryTree(root.getLeft());
            MirrorOfBinaryTree(root.getRight());
            /*交换结点内的两个指针*/
            temp  = root.getLeft();
            root.setLeft(root.getRight());
            root.setRight(temp);
        }
        return root;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 25** 给定两棵树，设计算法判断它们是否互为镜像？

**解答：**

```
int AreMirrors(BinaryTreeNode  root1, BinaryTreeNode  root2) {
        if(root1 == null && root2 == null)
                    return 1;
        if(root1 == null || root2 == null)
                    return 0;
        if(root1.getData() != root2.getData())
                   return 0;
        else return (AreMirrors(root1.getLeft(), root2.getRight()) && AreMirrors(root1.getRight(),
        root2.getLeft()));
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

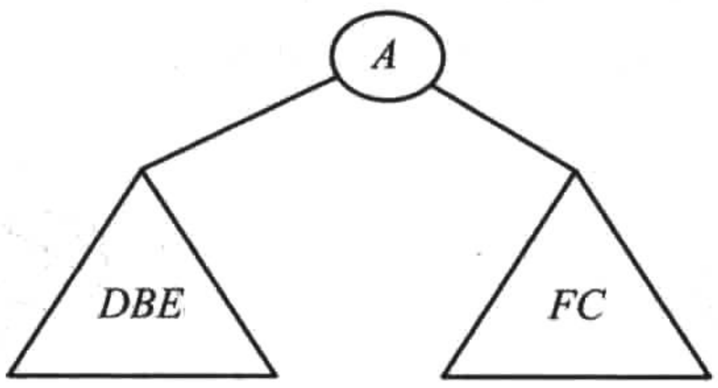
**问题 26**　给出一个算法，根据给定的中序遍历和前序遍历序列来构建二叉树。

**解答：**考虑以下遍历序列：

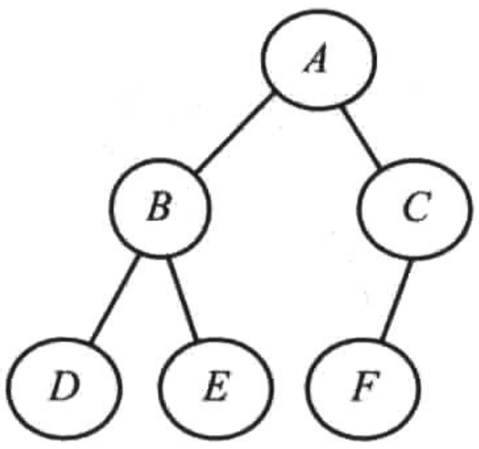
中序序列：*D*　*B*　*E*　*A*　*F*　*C*

前序序列：*A*　*B*　*D*　*E*　*C*　*F*

在前序序列中，最左边的元素是树的根结点。所以 *A* 是给定序列的根。通过在中序序列中找到‘*A*’，能够找到‘*A*’左边的所有元素，它们来自左子树，也能找到‘*A*’右边的所有元素，它们来自右子树。因此可以得出如下的二叉树。




基于以上步骤，可以递归地得到如下二叉树。




**算法：**BuildTree()。

1）从前序序列中取一个元素，将前序索引变量(下面代码中的 preIndex)加 1，用于选取下一次递归调用时的元素。
2）根据选择元素的数据值，创建一个新的树结点(newNode)。
3）查找所选结点在中序序列中的索引，用变量 inIndex 标记。
4）调用 BuildBinaryTree 为 inIndex 之前的所有结点构建一棵子树，将其作为 newNode 结点的左子树。
5）调用 BuildBinaryTree 为 inIndex 之后的所有结点构建一棵子树，将其作为 newNode 结点的右子树。
6）返回 newNode。

```
BinaryTreeNode  BuildBinaryTree(int inOrder[], int preOrder[], int inStrt, int inEnd) {
        static int preIndex = 0;
        BinaryTreeNode newNode = new BinaryTreeNode();
        if(inStrt > inEnd) return null;
        if(newNode == null) {
                System.out.println("Memory Error");
                return null;
        }
        //利用preIndex在前序序列中选择当前结点
        newNode.setData(preOrder[preIndex]);
        preIndex++;
        if(inStrt == inEnd)                          /*若该结点没有孩子结点则返回*/
                return newNode;
        /*否则在中序序列中找到该结点的索引*/
        int inIndex = Search(inOrder, inStrt, inEnd, newNode.getData());
        /*利用中序序列中结点的索引分别建立左子树和右子树*/
        newNode.setLeft(BuildBinaryTree(inOrder, preOrder, inStrt, inIndex-1));
        newNode.setRight(BuildBinaryTree(inOrder, preOrder, inIndex+1, inEnd));
        return newNode;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 27**　给定两个遍历序列，能否构建一棵唯一的二叉树？

**解答：**这依赖于给定的是什么遍历序列。如果其中有一个序列是中序序列，那么就可以唯一构建，否则不行。




因此，如果是以下序列组合，就能确定一棵唯一的二叉树：

- 中序和前序
- 中序和后序
- 中序和层次遍历

以下组合不能确定一棵唯一的树。

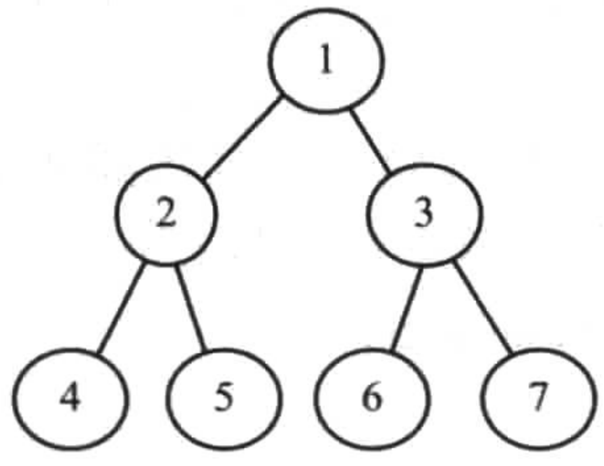
- 后序和前序
- 前序和层次遍历
- 后序和层次遍历

例如，上图所示的两棵二叉树的前序、层次和后序序列都是相同的：

$$前序遍历 = AB$$
$$后序遍历 = BA$$
$$层次遍历 = AB$$

因此，即使 3 个序列(前序、层次遍历和后序)都给出，也不能唯一地构建一棵树。

**问题 28**　设计算法，打印二叉树中某结点的所有祖先结点。如下面的树，7 的祖先是 1 3 7。




**解答：**除了基于树的深度优先搜索方法外，可以利用以下递归方法来输出祖先结点。

```
int PrintAllAncestors(BinaryTreeNode root, BinaryTreeNode node){
        if(root == null)
                return 0;
        if(root.getLeft() == node || root.getRight() == node ||
          PrintAllAncestors(root.getLeft(), node) || PrintAllAncestors(root.getRight(), node)) {
            System.out.println(root.getData());
            return 1;
        }
        return 0;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

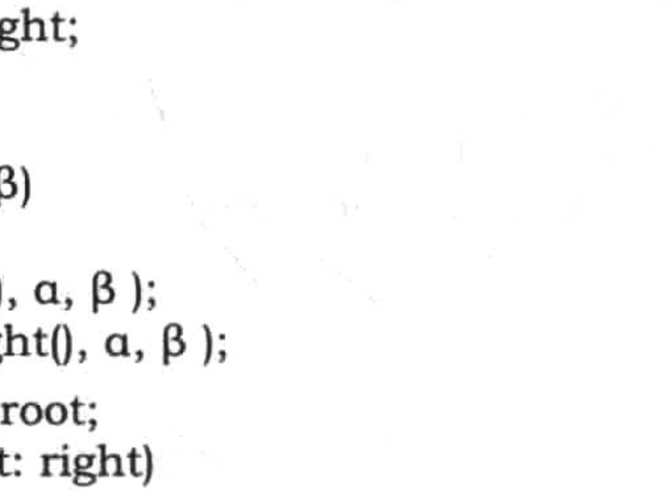
**问题 29** 设计算法查找二叉树中两个结点的最近公共祖先(LCA)。

**解答：**

```
BinaryTreeNode LCA(BinaryTreeNode root, BinaryTreeNode α, BinaryTreeNode β) {
        BinaryTreeNode left, right;
        if(root == null)
                return root;
        if(root == α || root == β)
                return root;
        left = LCA(root.getLeft(), α, β );
        right = LCA(root.getRight(), α, β );
        if(left && right) return root;
        else      return(left? left: right)
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于递归。

**问题 30 Zigzag 树遍历：**设计算法以 Zigzag 顺序遍历二叉树。例如，下面树的输出顺序应为：1 3 2 4 5 6 7。




**解答：**这个问题用两个栈就很容易解决。假设两个栈为：currentLevel 和 nextLevel。还用一个变量来跟踪当前层的顺序(是从左到右，还是从右到左)。将结点从 currentLevel 中出栈并输出其值。当当前层顺序是从左到右时，按照先左孩子结点后右孩子结点压入

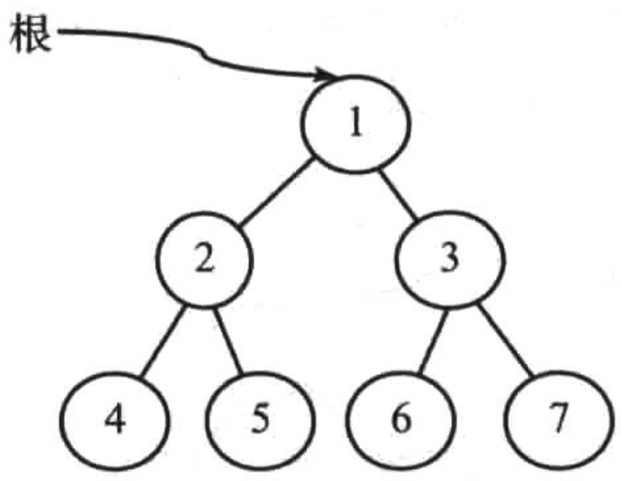
栈 nextLevel 中。因为栈是后进先出(LIFO)结构，所以下次当结点从 nextLevel 出栈时，它将按相反的顺序输出。

另一方面，如果当前层顺序是从右到左，则按照先右孩子结点后左孩子结点的顺序入栈。最后，需要在每一层结束时交换这两个栈(即当 currentLevel 为空时)。

```
void ZigZagTraversal(BinaryTreeNode root) {
        BinaryTreeNode temp;
        int leftToRight = 1;
        if(root == null)
            return;
Stack currentLevel currentLevel = new CreateStack(), nextLevel = new CreateStack();
Push(currentLevel, root);
while(!isEmpty(currentLevel)) {
        temp = Pop(currentLevel);
        if(temp) {
                System.out.println(temp.getData());
                if(leftToRight) {
                        if(temp.getLeft())
                                Push(nextLevel, temp.getLeft());
                        if(temp.getRight())
                                Push(nextLevel, temp.getRight());
                }
                else {    if(temp.getRight())
                                Push(nextLevel, temp.getRight());
                        if(temp.getLeft())
                                Push(nextLevel, temp.getLeft());
                }
                if(isEmpty(currentLevel)) {
                        leftToRight = 1-leftToRight;
                        swap(currentLevel, nextLevel);
                }
        }
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)+O(n)=O(n)$，用于两个栈。

**问题 31** 设计算法找到二叉树的垂直和。例如，




这棵树有 5 条垂直线：

垂直线 1：结点 4⇒垂直和是 4。

垂直线 2：结点 2⇒垂直和是 2。

垂直线 3：结点 1，5，6⇒垂直和是＝1＋5＋6＝12。

垂直线 4：结点 3⇒垂直和是 3。

垂直线 5：结点 7⇒垂直和是 7。

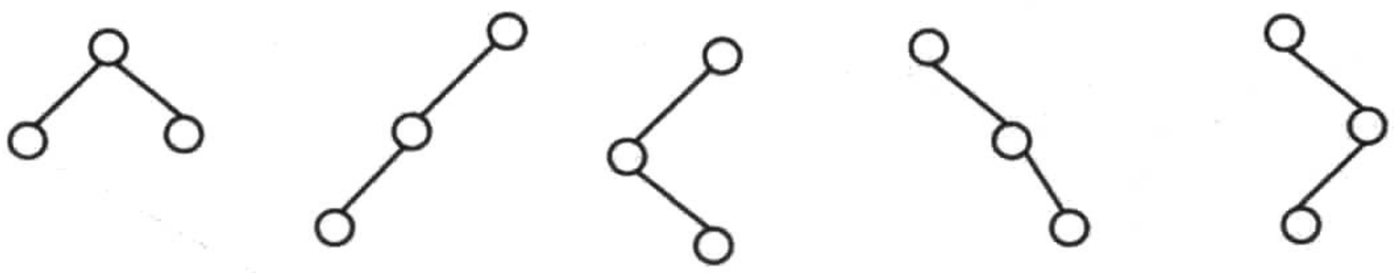
因此输出是：4 2 12 3 7。

**解答：**可以执行中序遍历，并根据列建立散列表。调用 VerticalSumInBinaryTree(root，0)，该函数表示根结点在第 0 列。在遍历时，对列进行散列，并利用 root. getData() 增加它的值。

```
void VerticalSumInBinaryTree(BinaryTreeNode root, int column) {
        if(root==null)
            return;
        VerticalSumInBinaryTree(root.getLeft(), column-1);
        //散列表的实现请参见第14章
        Hash[column] += root.getData();
        VerticalSumInBinaryTree(root.getRight(), column+1);
}
VerticalSumInBinaryTree(root, 0);
Print Hash;
```

**问题 32** $n$ 个结点可以组合成多少棵不同的二叉树？

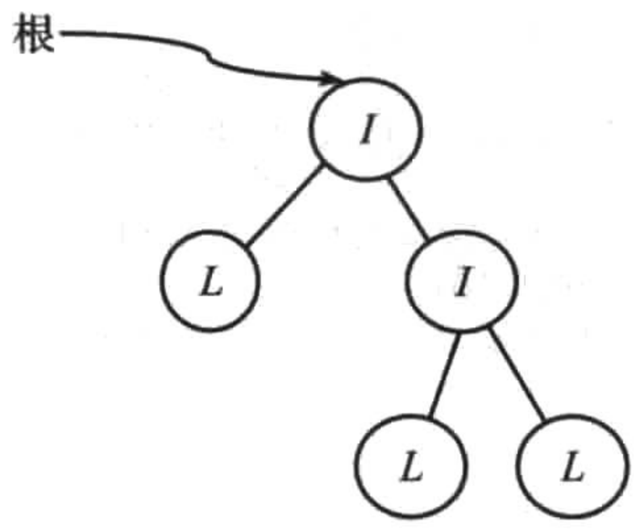
**解答：**例如，考虑 3 个结点的树($n=3$)。它最多有 $5(2^3-3)$种不同的组合形式，即有 5 棵不同的树。



一般来讲，如果有 $n$ 个结点，则存在 $2^n-n$ 棵不同的树。

**问题 33** 假设一棵树，叶子结点用‘$L$’表示，内部结点用‘$I$’表示，同时假定每个结点只能有 0 个或者 2 个孩子结点。根据这棵树的前序序列，构建这棵树。

**例子：**给出前序序列为⇒$ILILL$。




**解答：**首先来看前序序列是如何生成的。前序遍历意味着处理的第一个结点是根结点，然后前序遍历左子树，最后前序遍历右子树。在一般情况下，如果仅仅用前序遍历不可能检测出哪个结点是左子树的最后一个结点和右子树的第一个结点。由于现在每个结点有 2 个孩子结点或没有孩子结点，所以能够确定如果一个孩子结点存在则其兄弟结点也必定存在。因此当计算子树时，也需要计算其兄弟子树。

其次，不论何时在输入序列中遇到‘$L$’，都表明这是一个叶子结点，且在该结点可以结束某个特定的子树。在‘$L$’结点(双亲结点的左孩子结点)后，接下来的就是其兄弟结点。如果‘$L$’是其双亲结点的右孩子结点，那么需要返回到上一层计算下一棵子树。根据上述描述，可以容易确定一棵子树何时已经结束，另一棵子树开始。这就意味着，对于

任意一个起始结点，容易生成对应的子树，但需要选取正确的起始结点。

```
BinaryTreeNode BuildTreeFromPreOrder(char[] A, int i) {
        if(A == null)                                    //边界条件
                return null;
        BinaryTreeNode newNode = new BinaryTreeNode();
        newNode.setData(A[i]);
        newNode.setLeft(null);
        newNode.setRight(null);
        if(A[i] == 'L')                                  //到达一个叶子结点，返回
                return newNode;
        i = i + 1;                                       //构建左子树
        newNode.setLeft(BuildTreeFromPreOrder(A, i));
        i = i + 1;                                       //构建右子树
        newNode.setRight(BuildTreeFromPreOrder(A, i));
        return newNode;
}
```

时间复杂度为 $O(n)$。

**问题 34** 给定一棵带有 3 个指针(左指针、右指针和下一个兄弟指针)的二叉树。假设下一个兄弟指针初始化为空，设计算法来填充下一个兄弟指针。

**解答：** 使用简单队列可以实现。

```
void FillNextSiblings(BinaryTreeNode root) {
        LLQueue Q = new LLQueue();
        BinaryTreeNode temp;
        if(root == null)  return 0;
        Q.enQueue(root);
        Q.enQueue(null);
        while(!Q.isEmpty()) {
                root=Q.deQueue();
                //当前层结束
                if(root == null) { //为下一层放置另一个标记
                        if(!Q.isEmpty())
                                Q.enQueue(null);
                }
                else {   temp.setNextSibling(Q.getFront());
                        if(root.getLeft())
                                Q.enQueue( root.getLeft());
                        if(root.getRight())
                                Q.enQueue( root.getRight());
                }
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 35** 还有其他方法来解决问题 34 吗?

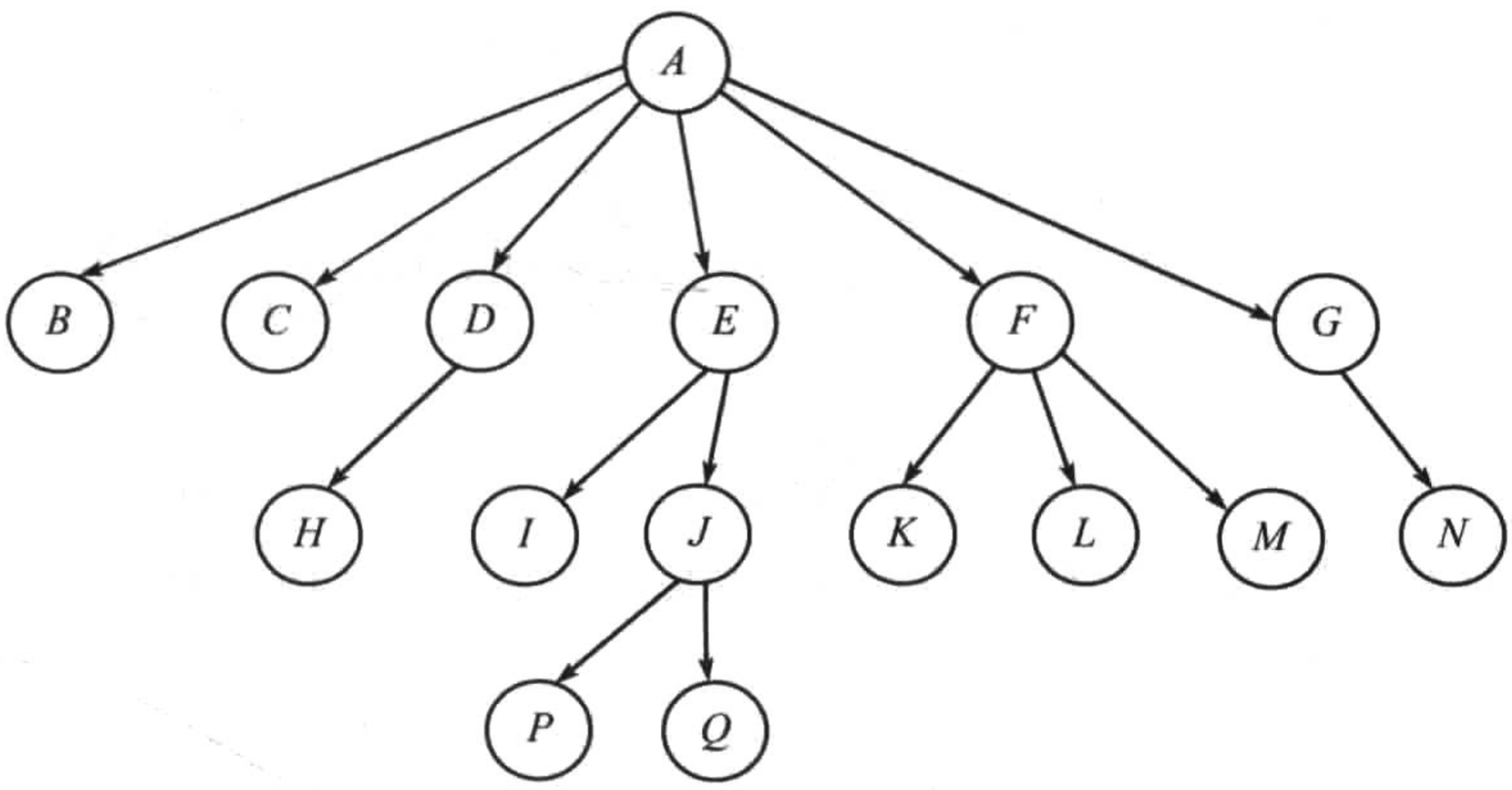
**解答：** 关键是利用已填充的 nextSibling 指针。如前所述，只需增加一个步骤。在传递左孩子结点指针 left 和右孩子结点指针 right 给递归函数之前，将右孩子的 nextSibling 填充为当前结点的下一个兄弟的左孩子结点。该算法正常执行的前提是，当前结点的 nextSibling 指针已填充。在本例中，该条件是成立的。

```
void FillNextSiblings(BinaryTreeNode root) {
        if (root == null) return;
        if (root.getLeft())
                root.getLeft().setNextSibling(root.getRight());
```

```
        if (root.getRight())
                if(root.getNextSibling())
                        root.getRight().setNextSibling(root.getNextSibling().getLeft());
                else    root.getRight().setNextSibling(null);
        FillNextSiblings(root.getLeft());
        FillNextSiblings(root.getRight());
}
```

时间复杂度为 $O(n)$。

## 6.5　通用树(N 叉树)




在上一节中，我们讨论了每个结点最多有 2 个孩子结点的二叉树，这种树很容易用两个指针来表示。但是若一棵树中每个结点可以有任意多个子结点，而且并不知道一个结点到底有多少个子结点，该如何表示它们？例如，考虑上图所示的树。

### 1. 怎么表示这种树

在上图所示的树中，结点分别有 6 个孩子结点、3 个孩子结点、2 个孩子结点和 0 个孩子结点。为了表示这棵树，需要考虑最坏的情况(例如，6 个孩子结点)，并且给每个结点分配在最坏情况下的孩子指针的个数。基于此，每个结点可以表示如下：

```
public class TreeNode {
        public int data;
        public TreeNode firstChild;
        public TreeNode secondChild;
        public TreeNode thirdChild;
        public TreeNode fourthChild;
        public TreeNode fifthChild;
        public TreeNode sixthChild;
        .....
}
```

由于并不是在所有情况下都需要使用所有的指针，所以将导致大量的内存浪费。此外，另一个问题是，事先并不知道每个结点的孩子结点数。

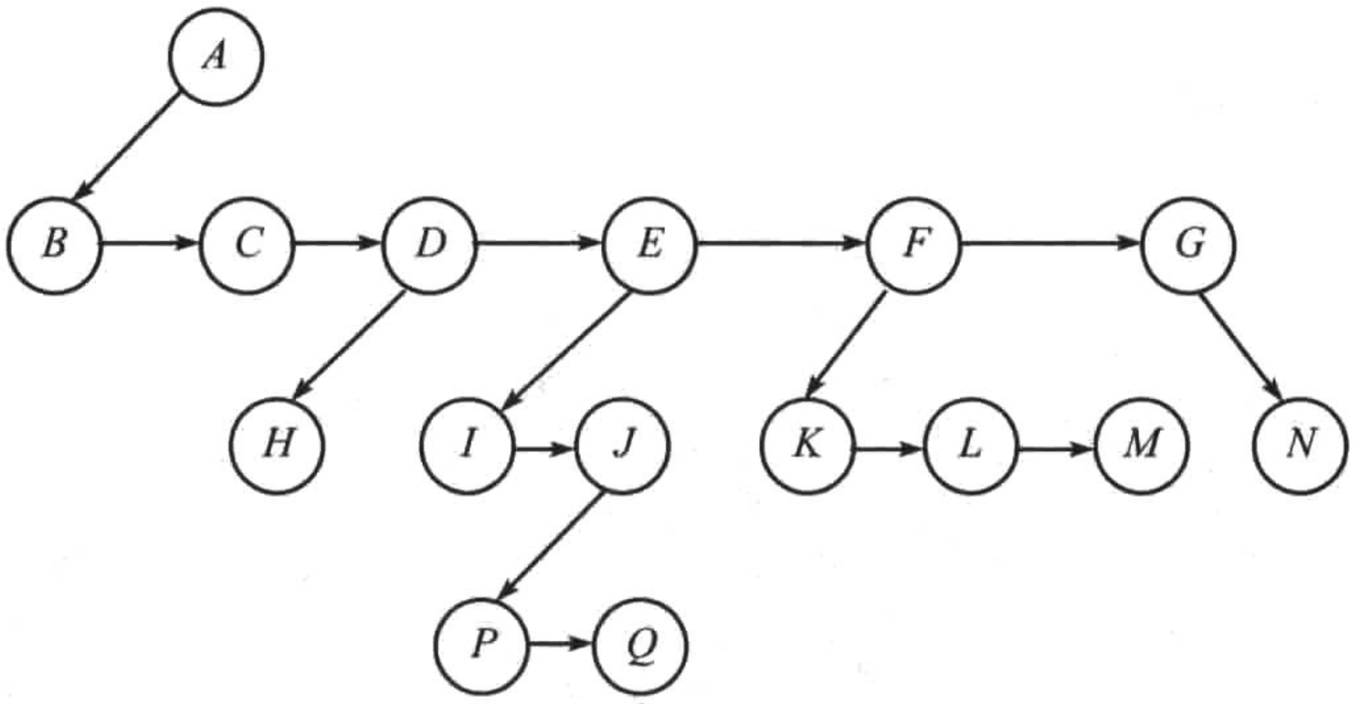
为了解决这个问题，需要一种表示方法，它既能减少内存的浪费，也能接受具有任意多个孩子结点的结点。

### 2. 通用树的表示

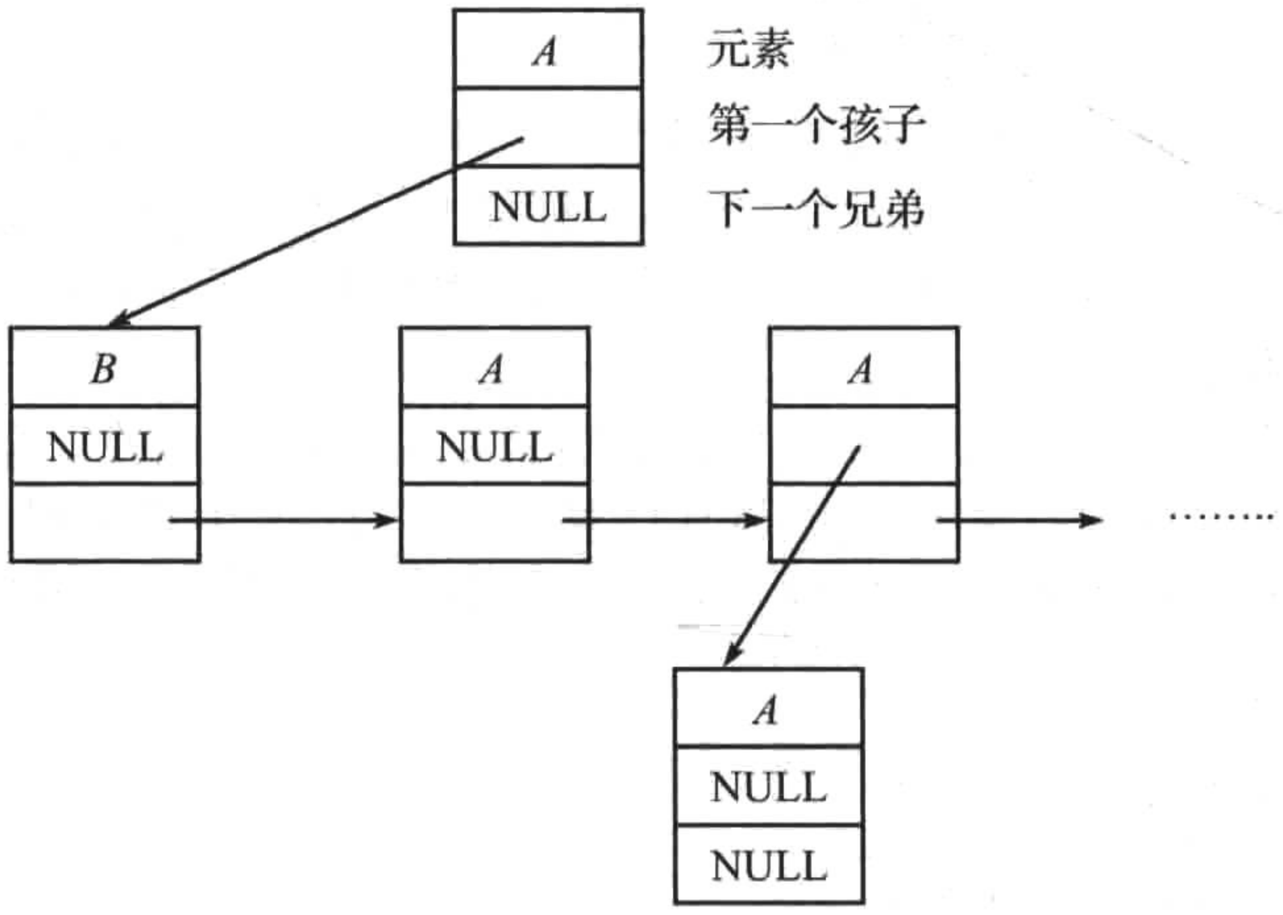
因为需要遍历树中的所有结点，所以一种可能的解决方法是：

- 同一个双亲结点(兄弟)的孩子结点从左至右排列。
- 双亲结点只指向第一个孩子结点，删除从双亲结点到其他孩子结点的链接。

上述的具体含义是，如果孩子结点之间有一条链路相连，那么双亲结点就不需要额外的指针指向所有的孩子结点。这是因为从双亲结点的第一个孩子结点开始就能够遍历所有的元素。因此，只要双亲结点用一个指针指向其第一个孩子结点，且同一个双亲结点的所有孩子结点之间都有链路，那么就能解决上面的问题。




这种表示有时叫作第一个孩子/下一个兄弟表示法。通用树的孩子/兄弟表示法如上图所示。这棵树的实际表示如下：




基于上面的讨论，通用树的结点定义如下：

```
public class TreeNode {
        public int data;
        public TreeNode firstChild;
        public TreeNode nextSibling;
        public int getData() {
                return data;
        }
        public void setData(int data) {
                this.data = data;
        }
        public BinaryTreeNode getFirstChild() {
```

```
        return firstChild;
}
public void setFirstChild(BinaryTreeNode firstChild) {
        this.firstChild = firstChild;
}
public BinaryTreeNode getNextSibling () {
        return nextSibling;
}
public void setNextSibling(BinaryTreeNode nextSibling) {
        this.nextSibling = nextSibling;
}
```

**注意：**由于能够用二叉树表示任意的通用树，所以在实际中仅使用二叉树。

### 3. 通用树的相关问题

**问题 36**　给定一棵树，设计算法来获取树的所有结点值之和。

**解答：**该问题的解决方法与简单二叉树类似。即遍历整个链表，并不断累加其值。可以使用层次遍历或者简单递归方式来实现。

```
int FindSum(TreeNode root) {
        if(root == null)
                return 0;
        return root.getData() + FindSum(root.getFirstChild()) + FindSum(root.getNextSibling());
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$(如果不考虑栈空间)，否则为 $O(n)$。

**注意：**所有针对二叉树讨论的问题也适用于通用树。只是结点的两个指针不再叫作左指针和右指针而是用 firstChild 和 nextSibling 表示。

**问题 37**　对于 4 叉树(每个结点最多有 4 个孩子结点)，100 个结点的 4 叉树的最大可能高度是多少？假定只有一个结点的树的高度为 0。

**解答：**在 4 叉树中，每个结点可以有 0～4 个孩子结点。当每个双亲结点只有一个孩子结点时，才能得到树的最大高度。有 100 个结点的树的最大可能高度为 99。如果加上一个限制，令至少有一个结点有 4 个孩子结点，那么就可使得树中有一个结点有 4 个孩子结点而其他结点都只有一个孩子结点。在这种情况下，树的最大可能高度为 96。类似地，$n$ 个结点的树的最大可能高度为 $n-4$。

**问题 38**　对于 4 叉树(每个结点最多 4 个孩子结点)，有 $n$ 个结点的树的最小高度是多少？

**解答：**类似于上面的推理，要想得到树的最小高度，就需要使每个结点都有最多的孩子结点(即 4 个)。根据下表可得出在给定高度时的最大结点数。

| 高度 | 高度最大的结点 $h=4^h$ | 所有结点的高度 $h=(4^{h+1}-1)/3$ |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 4 | 1+4 |
| 2 | 4 * 4 | 1+4 * 4 |
| 3 | 4 * 4 * 4 | 1+4 * 4+4 * 4 * 4 |

对于给定高度为 $h$ 的树，最多可能具有的结点数为：$\frac{4^{h+1}-1}{3}$。为了使高度最小，两

边分别取对数即可。

$$n=\frac{4^{h+1}-1}{3}\Rightarrow 4^{h+1}=3n+1\Rightarrow(h+1)\log 4=\log(3n+1)$$

$$\Rightarrow h+1=\log_4(3n+1)\Rightarrow h=\log_4(3n+1)-1$$

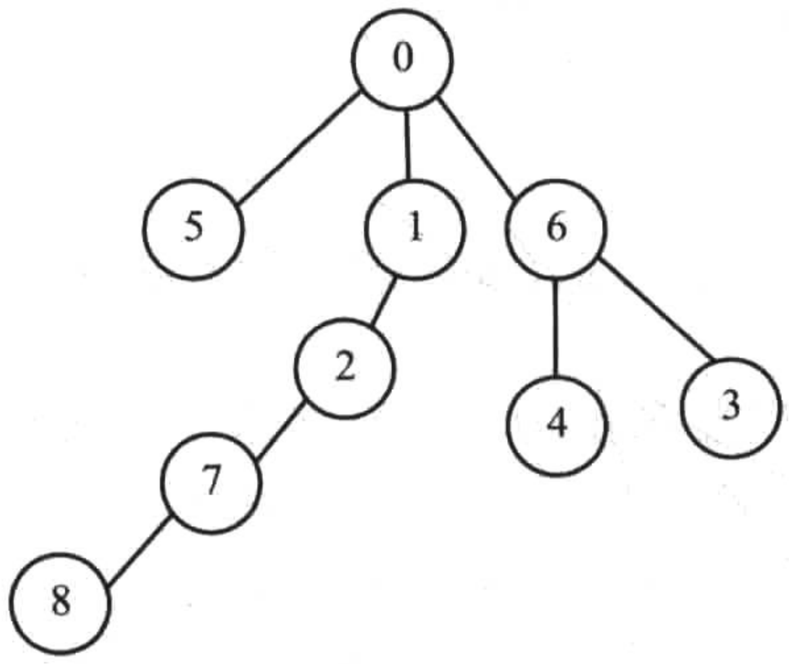
**问题 39** 给定一个双亲结点数组 $P$，其中 $P[i]$是树的第 $i$ 个结点的双亲结点(假设根结点的双亲结点用－1 表示)。设计算法获取树的高度或深度。

**解答：** 根据问题的定义，给定数组表示双亲结点数组。这意味着，需要考察数组对应的树，从而求出树的高度。

例如，如果数组 $P$ 为：

| −1 | 0 | 1 | 6 | 6 | 0 | 0 | 2 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

其对应的树就是：




这棵给定树的深度为 4。如果仔细观察可得，只需要从每一个结点开始，跟踪找其双亲结点直到找到－1 为止，同时跟踪所有结点深度的最大值，就可以得到问题的解。

```
int FindDepthInGenericTree(int P[], int n) {
        int maxDepth =-1, currentDepth =-1, j;
        for(int i = 0; i < n; i++)   {
                currentDepth = 0; j = i;
                while(P[j] != -1) {
                        currentDepth++; j = P[j];
                }
                if(currentDepth > maxDepth)
                        maxDepth = currentDepth;
        }
        return maxDepth;
}
```

时间复杂度为 $O(n^2)$，对于斜树，将重复计算出与 $i$ 相同的值。空间复杂度为 $O(1)$。

**注意：** 代码优化的方法是：将已计算得到的结点深度存储到散列表或其他数组中。这能减少时间复杂度，不过需要使用额外的空间。

**问题 40** 给定通用树中的一个结点，设计算法来求其兄弟结点的个数。

**解答：** 对于树中一个给定结点，只需要遍历其所有的兄弟结点即可。

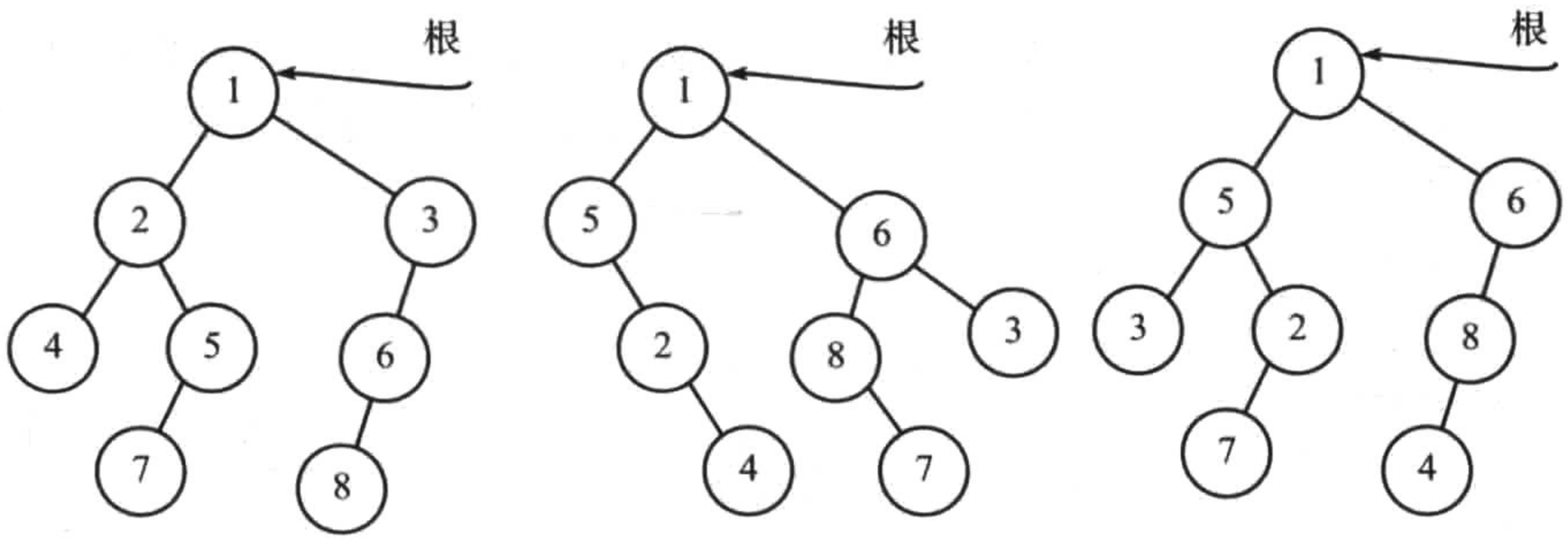
```
int SiblingsCount(TreeNode current) {
        int count = 0;
        while(current)   {
```

```
                count++;
                current = current.getNextSibling();
        }
        reutrn count;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**问题 41** 给定两棵树，如何判断这两棵树是否互为同构树？

**解答：** 如果两棵二叉树 root1 和 root2 具有相同的结构，则它们是同构的。树中结点的值不影响两棵树是否是同构的。在下图中，中间的树与其他树不是同构的，但右边的树与左边的树是同构的。

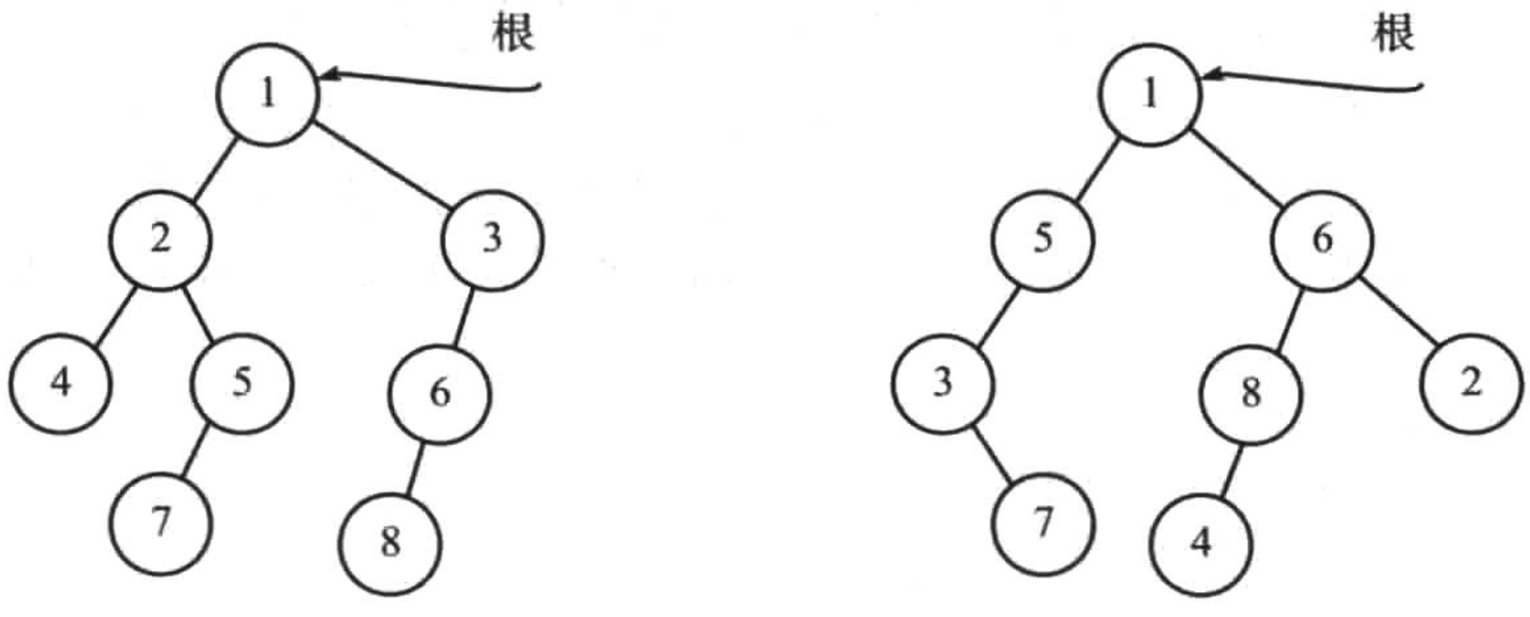



```
int IsIsomorphic(TreeNode  root1, TreeNode root2) {
        if(root1 == null && root2 == null)
                return 1;
        if((root1 == null && root2 != null) || (root1 != null && root2 == null))
                return 0;
        return (IsIsomorphic(root1.getLeft(), root2.getLeft())  && IsIsomorphic(root1.getRight(),
        root2.getRight()));
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 42** 如何判断给定的两棵树是否互为准同构？

**解答：** 给定两棵树 root1 和 root2，如果通过交换 root1 中某些结点的左孩子结点与右孩子结点能使 root1 转换为 root2，那么就可说树 root1 和 root2 是准同构的。在决定两棵树是否是准同构的时，结点中的数据并不重要，只有其形状才是重要的。下面这两棵树是准同构的，因为如果左边结点的孩子结点交换后，就可以得到右边的树。



```
int QuasiIsomorphic(TreeNode  root1, TreeNode  root2) {
        if(root1 == null && root2 == null)
                return true;
        if((root1 == null && root2 != null) || (root1 != null && root2 == null))
                return false;
        return (QuasiIsomorphic(root1.getLeft(), root2.getLeft()) &&
                QuasiIsomorphic(root1.getRight(), root2.getRight()) ||
                QuasiIsomorphic(root1.getRight(), root2.getLeft()) &&
                QuasiIsomorphic(root1.getLeft(), root2.getRight()));
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

**问题 43** 给定通用树中的一个结点，设计算法来计算该结点的孩子结点数。

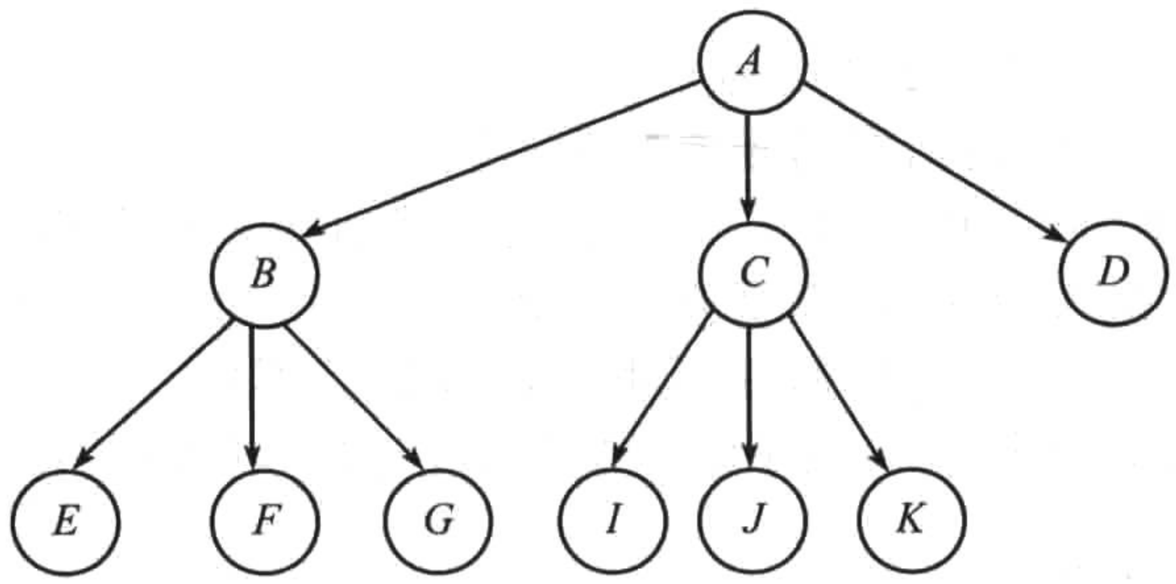
**解答：** 给定树中一个结点，找到其第一个孩子结点，然后遍历该孩子结点的所有兄弟结点，就可求出解。

```
int ChildCount(TreeNode current) {
        int count = 0;
        current = current.getFirstChild();
        while(current != null)     {
                count++;
                current = current.getNextSibling();
        }
        reutrn count;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**问题 44** 满 $k$ 叉树是指每个结点要么有 $k$ 个孩子结点，要么没有孩子结点。给定一棵满 $k$ 叉树的前序序列。给出构建该满 $k$ 叉树的算法。

**解答：** 在 $k$ 叉树中，第 $i$ 个位置结点的孩子结点在 $k*i+1 \sim k*i+k$ 个位置上。例如，下例是一棵满 3 叉树。




如上图所示，在前序遍历中，首先处理第一棵左子树，然后访问根结点，最后处理右子树。因此，为了构建满 $k$ 叉树，只需要不断创建新的结点，而不需要考虑之前已创建结点的影响。可以使用一个全局索引来递归地构建树。$k$ 叉树的定义和声明如下：

```
public class K-aryTreeNode {
        public int data;
        public K-aryTreeNode[] child;
        public K-aryTreeNode(int k){
                child = new K-aryTreeNode[k];
        }
```

```
public void setData(int dataInput){
        data = dataInput;
}
public int getChild(){
        return data;
}
public void setChild(int i, K-aryTreeNode childNode){
        child[i]= childNode;
}
        public K-aryTreeNode getChild(int i){
                return child[i];
        }
        ...
}
int Ind = 0;
K-aryTreeNode BuildK-aryTree(int A[], int n, int k) {
        if(n <= 0)
                return null;
        K-aryTreeNode newNode = new K-aryTreeNode(k);
        if(newNode == null) {
                System.out.println("Memory Error");
                return;
        }
        newNode.setData(A[Ind]);
        for(int i = 0; i<k; i++) {
                if(k * Ind + i <n) {
                        Ind++;
                        newNode.setChild(BuildK-aryTree(A, n, k,Ind ));
                }
                else  newNode.setChild(null);
        }
        return newNode;
}
```

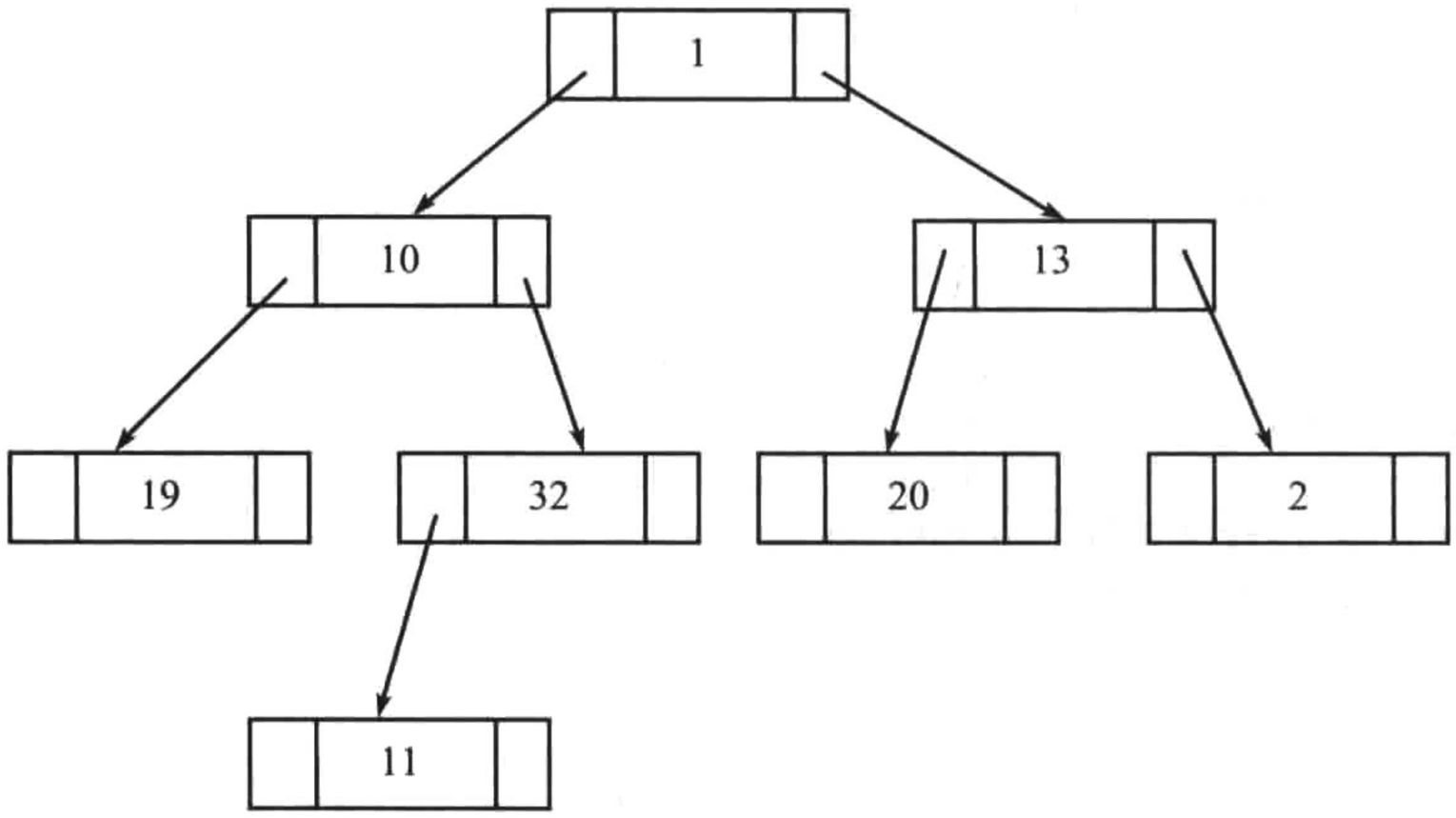
时间复杂度为 $O(n)$，其中 $n$ 为前序数组的大小。这是因为算法顺序地访问每个结点一次，且不需要访问已经构建的结点。

## 6.6　线索(无栈或无队列结构)二叉树遍历

在本章先前的章节中，我们已经学习了二叉树的前序、中序和后序遍历，这些遍历方式使用栈作为辅助数据结构，另外层次遍历使用队列作为辅助数据结构。在本节中，我们将讨论一种新的遍历算法，它即不需要栈也不需要队列。这种遍历算法叫作线索二叉树遍历或无栈/无队列遍历。

### 1. 常规二叉树遍历的问题

- 栈和队列需要的存储空间很大。
- 任意一棵二叉树往往存在大量的空指针。例如，拥有 $n$ 个结点的二叉树有 $n+1$ 个空指针，它们占用的空间被浪费了。
- 很难直接找到一个给定结点的后继结点(前序、中序和后序后继)。




### 2. 线索二叉树的动机

为解决上述问题，一种方法就是在空指针中存储一些有用的信息。通过仔细观察以前的遍历之所以需要栈/队列，是因为必须记录当前位置以便在处理完左子树后移动到右子树。如果在空指针中存储有用的信息，那么就不需要将这些信息存储在栈/队列中。在空指针中存储此类信息的二叉树叫作*线索二叉树*。根据上面的讨论，假定要在空指针中存储有用的信息，那么第二个问题是，存储什么呢？

通常的做法是存储前驱或后继结点信息。这意味着，如果处理前序遍历，那么对于一个给定结点，空左指针将包含前序序列前驱信息，而空右指针将包含前序序列后继信息。这些特殊的指针就叫作*线索*。

### 3. 线索二叉树分类

该分类基于是否在两个空指针中都存储还是只在其中一个空指针中存储信息来确定的。

- 如果只在空左指针中存储前驱信息，则把这样的二叉树叫作*左线索二叉树*。
- 如果只在空右指针中存储后继信息，则把这样的二叉树叫作*右线索二叉树*。
- 如果在空左指针中存储前驱信息并在空右指针中存储后继信息，则把这样的二叉树叫作*满线索二叉树*或简单地称为*线索二叉树*。

**注意：**后面只讨论(满)线索二叉树。

### 4. 线索二叉树的类型

基于上述讨论，线索二叉树有 3 种类型。

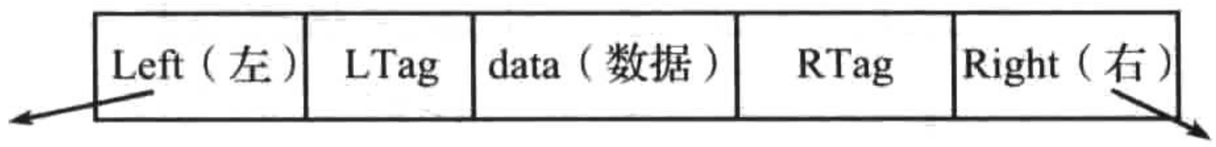
- *前序线索二叉树*：空左指针存储前序序列前驱信息，空右指针将存储前序序列后继信息。
- *中序线索二叉树*：空左指针存储中序序列前驱信息，空右指针将存储中序序列后继信息。
- *后序线索二叉树*：空左指针存储后序序列前驱信息，空右指针将存储后序序列后继信息。

**注意：**由于表示方式相似，所以接下来将使用中序线索二叉树。

### 5. 线索二叉树结构

任何程序在访问树必须能够区分*左/右*指针还是线索。为此，需要为每个结点使用两

个附加字段。对于线索树，结点形式如下：



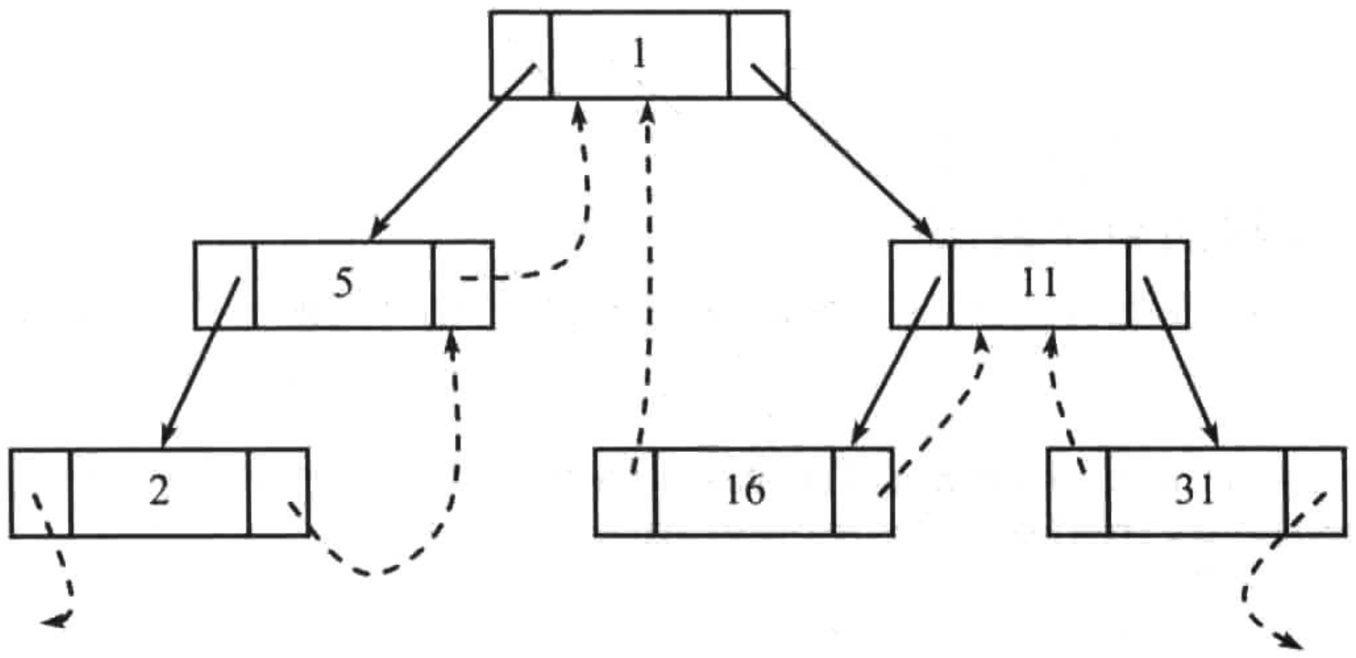

```
public class ThreadedBinaryTreeNode {
        public ThreadedBinaryTreeNode left;
        public int LTag;
        public int data;
        public int RTag;
        public ThreadedBinaryTreeNode right;
        .....
}
```

### 6. 二叉树和线索二叉树的区别

| | 常规二叉树 | 线索二叉树 |
|---|---|---|
| 如果 LTag==0 | 空 | 指向中序序列的左结点 |
| 如果 LTag==1 | 指向左孩子的左结点 | 指向左孩子的左结点 |
| 如果 RTag==0 | 空 | 指向中序序列的右结点 |
| 如果 RTag==1 | 指向右孩子的右结点 | 指向右孩子的右结点 |

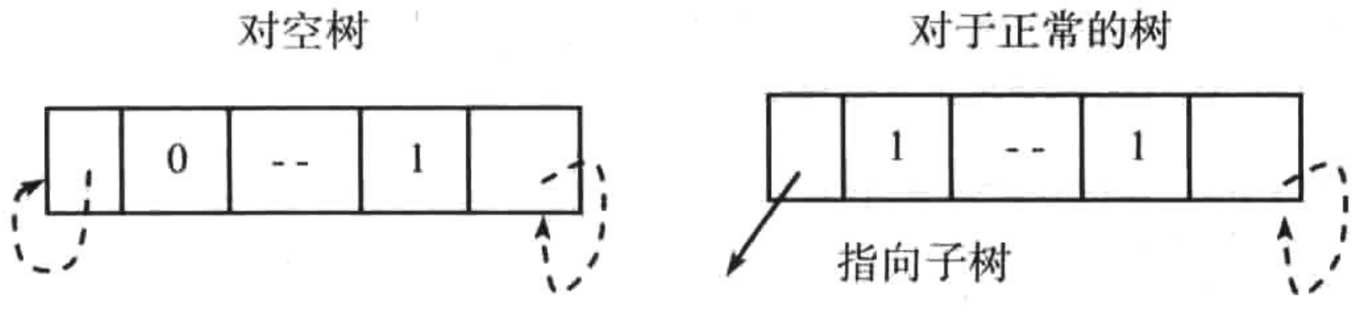
**注意：** 类似地，也可定义前序/后序二叉树与线索二叉树的不同点。

例如，让我们试着表示一棵中序线索二叉树。下图给出的树展示了中序线索二叉树是什么样子。虚线箭头表示线索。通过观察可知，最左边结点(2)的左指针和最右边结点的右指针(31)是悬空的。

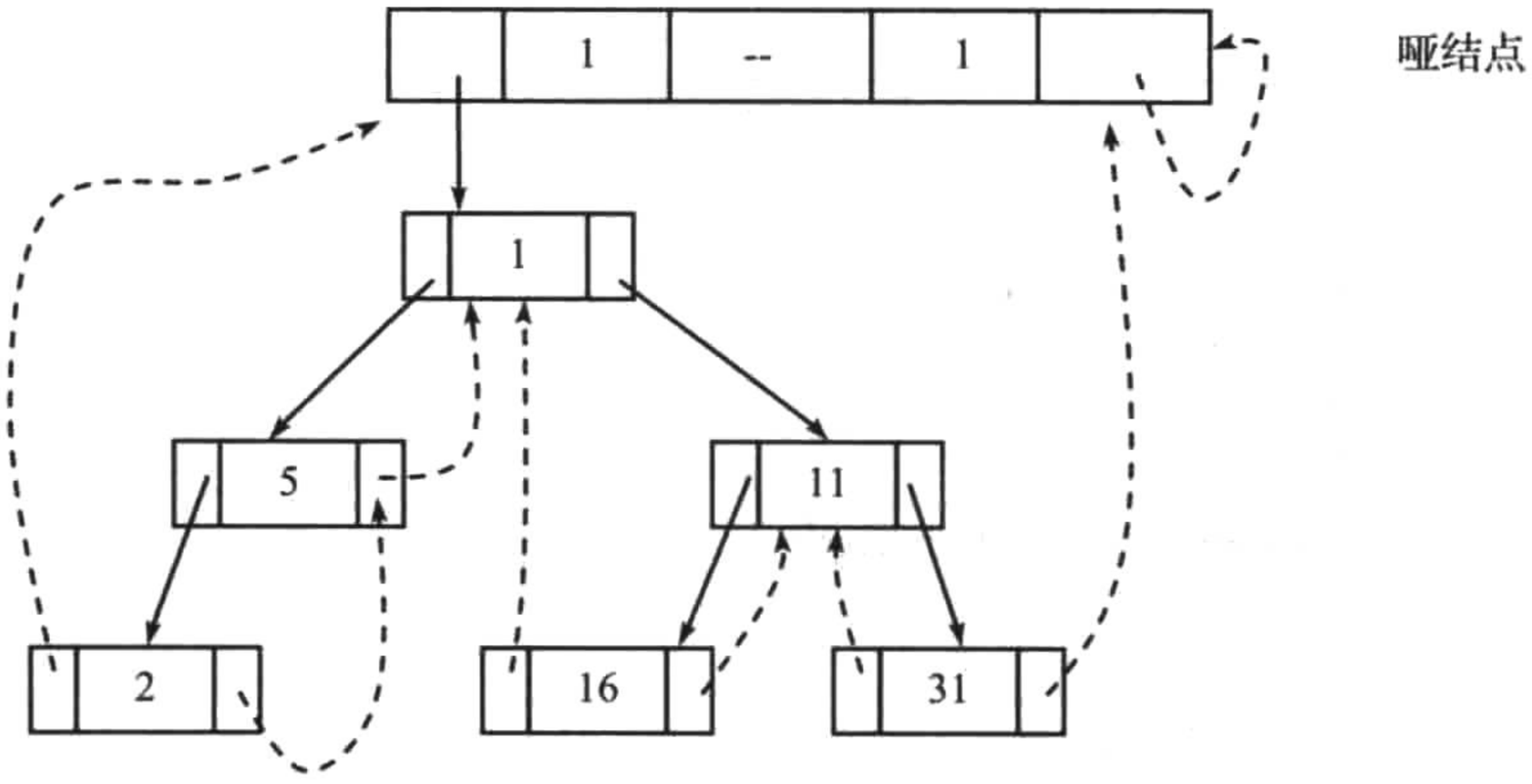



**最左和最右指针应该指向什么呢**

在线索二叉树的表示中，一个通常的约定是使用一个特定的哑结点 Dummy，对于空树也是如此。哑结点的右标签(RTag)是 1，且其右孩子指针指向其自身。




根据这种约定，上面的树可以表示为：




### 7. 在中序线索二叉树中查找中序后继

为了在不使用栈的情况下，找到给定结点中序序列后继结点，假设给定结点为 $P$。

**策略：**如果 $P$ 没有右子树，那么返回 $P$ 的右孩子结点。如果 $P$ 有右孩子结点，那么返回左子树包含 $P$ 的最近结点的左孩子结点。

```
ThreadedBinaryTreeNode InorderSuccessor(ThreadedBinaryTreeNode P) {
        ThreadedBinaryTreeNode Position;
        if(P→RTag == 0)
                return P.getRight();
        else {
                Position = P.getRight();
                while(Position.getLTag() == 1)
                        Position = Position.getLeft();
                return Position;
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

### 8. 中序索引二叉树的中序遍历

从哑结点开始，递归调用 InorderSuccessor()来访问每一个结点，直至再次到达哑结点。

```
void InorderTraversal(ThreadedBinaryTreeNode root) {
        ThreadedBinaryTreeNode P = InorderSuccessor(root);
        while(P != root) {
                P = InorderSuccessor(P);
                System.out.println(P.getData());
        }
}
```

**另外一种实现方式：**

```
void InorderTraversal(ThreadedBinaryTreeNode root) {
        ThreadedBinaryTreeNode P = root;
        while(1) {
                P = InorderSuccessor(P);
                if(P == root) return;
                System.out.println(P.getData());
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**9. 中序线索二叉树中查找前序后继**

**策略：** 如果 $P$ 有左子树，那么返回 $P$ 的左孩子结点。如果 $P$ 没有左子树，那么返回右子树包含 $P$ 的最近结点的右孩子结点。

```
ThreadedBinaryTreeNode* PreorderSuccessor(ThreadedBinaryTreeNode P) {
        ThreadedBinaryTreeNode Position;
        if(P.getLTag() == 1)  return P.getLeft();
        else {
                Position = P;
                while(Position.getRTag() == 0)
                        Position = Position.getRight();
                return Position.getRight();
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**10. 中序二叉树的前序遍历**

与中序遍历相同，前序遍历从哑结点开始，递归调用 PreorderSuccessor()函数访问每一个结点，直至再次回到哑结点。

```
void PreorderTraversal(ThreadedBinaryTreeNode root) {
        ThreadedBinaryTreeNode P;
        P = PreorderSuccessor(root);
        while(P != root) {
                P = PreorderSuccessor(P);
                System.out.println(P.getData());
        }
}
```

**另一种实现方法：**

```
void PreorderTraversal(ThreadedBinaryTreeNode root) {
        ThreadedBinaryTreeNode P = root;
        while(1) {
                P = PreorderSuccessor(P);
                if(P == root) return;
                System.out.println(P.getData());
        }
}
```

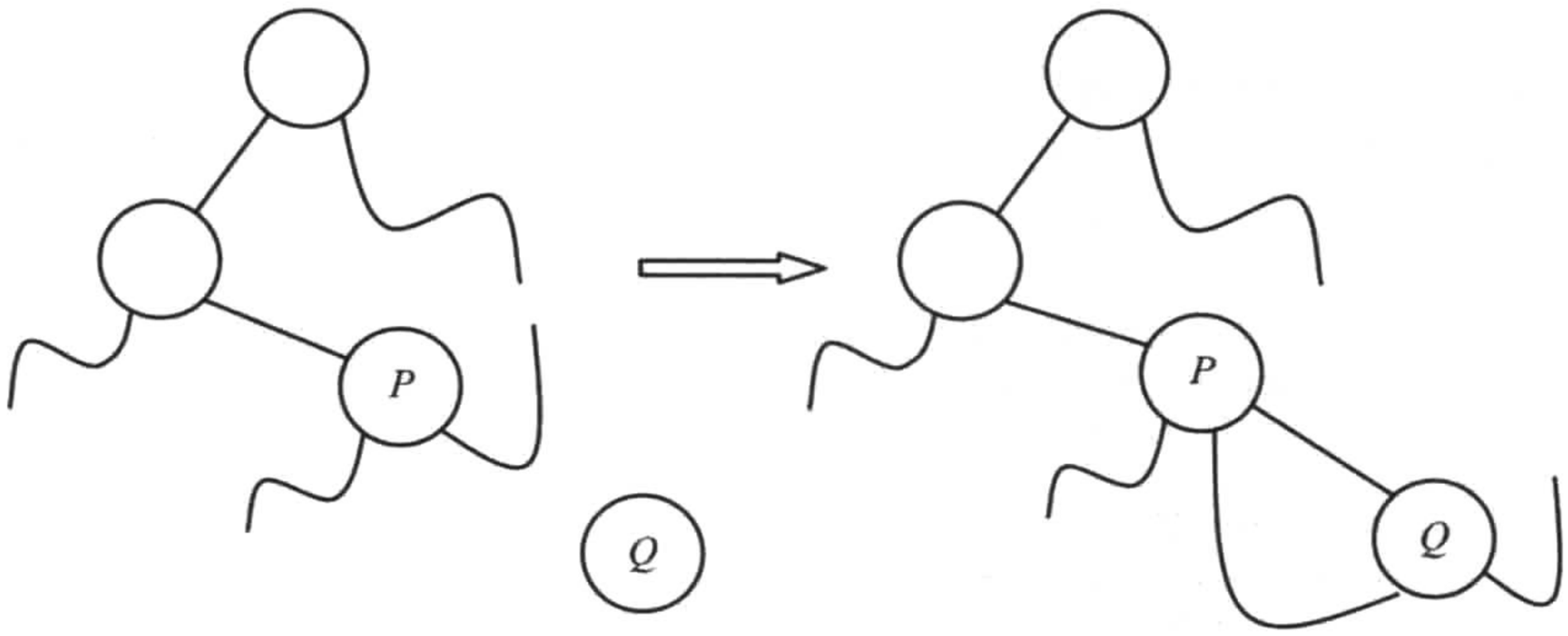
时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**注意：** 从上述讨论中可知，利用线索二叉树来查找中序后继和先序后继很容易。但若不使用栈，查找后序后继非常困难。

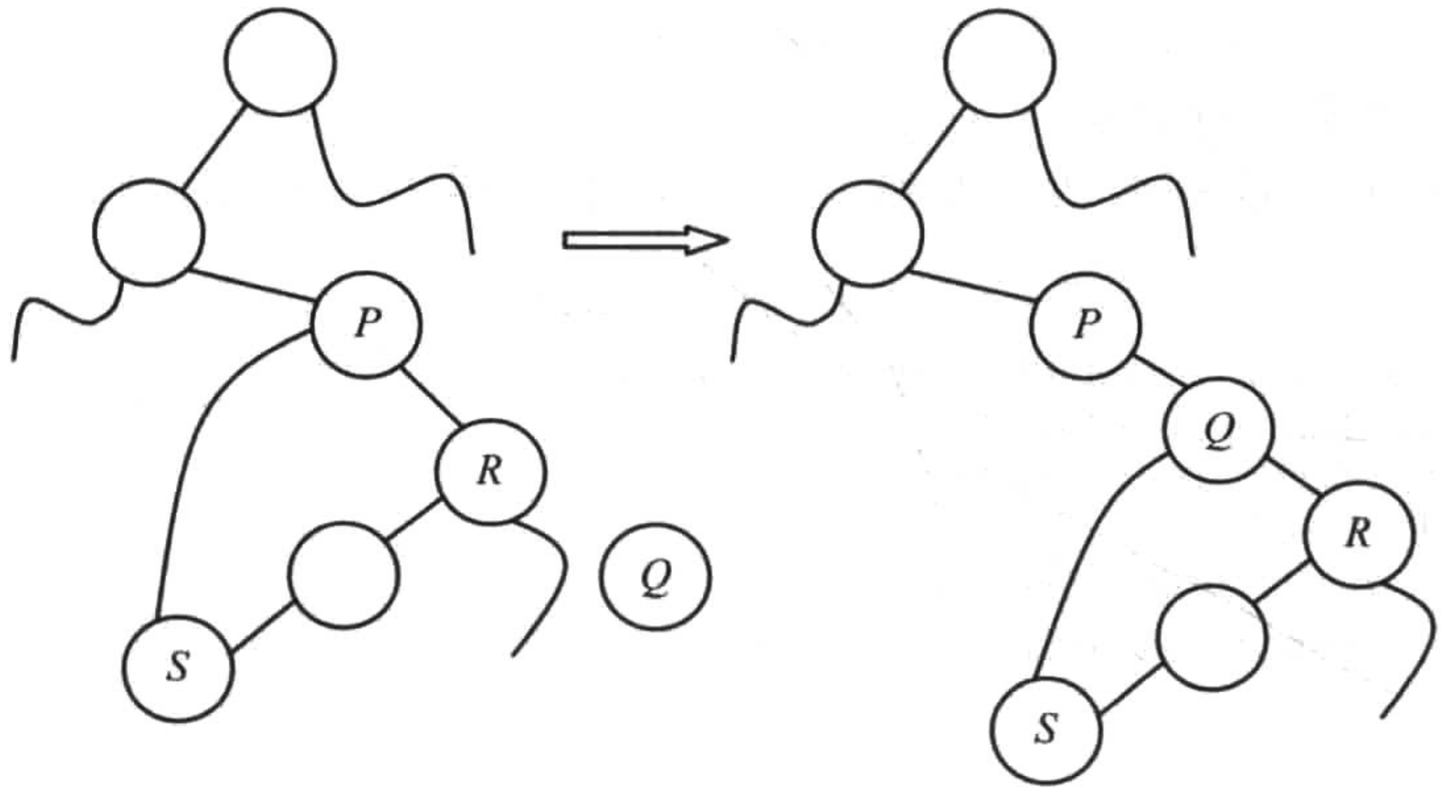
**11. 在中序线索二叉树中插入结点**

为了简单起见，假定有两个结点 $P$ 和 $Q$，现在要将 $Q$ 连接到 $P$ 的右边。为此有两种情况需要考虑：

- 结点 $P$ 没有右孩子结点。在这种情况下，只需要将 $Q$ 连接到 $P$ 上，同时改变其左指针和右指针即可。




- 结点 $P$ 有右孩子结点(假设为 $R$)。在此情况下，需要遍历 $R$ 的左子树，并找到最左边的结点，然后更新这个结点的左指针和右指针(如下图所示)。




```
void InsertRightInInorderTBT(ThreadedBinaryTreeNode P,  ThreadedBinaryTreeNode Q) {
        ThreadedBinaryTreeNode Temp;
        Q.setRight(P.getRight());
        Q.setRTag(P.getRTag());
        Q.setLeft(P);
        Q.setLTag(0);
        P.setRight(Q);
        P.setRTag( 1);
        if(Q.getRTag() == 1) {                                    //第2种情况
                Temp = Q.getRight();
                while(Temp.getLTag())
                        Temp = Temp.getLeft();
                Temp.setLeft(Q);
        }
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**12. 线索二叉树相关的问题**

**问题 45** 给定一棵二叉树(非线索二叉树)，如何找到前序序列后继？

**解答：** 为了解决这个问题，需要使用辅助栈 $S$。在第一次调用时，参数结点是指向树头的指针，其值为空。由于只需要简单获取上次函数调用时得到结点的后继，所以有必要将栈 S 的内容和指向最后访问结点的指针 $P$(从函数一次调用到下一个的)保存。它们定义为静态变量。

```
//查找非线索二叉树前序序列后继
BinaryTreeNode PreorderSuccssor(BinaryTreeNode node) {
        static BinaryTreeNode P;
        LLStack S = new LLStack();
        if(node != null)
                P = node;
        if(P.getLeft() != null) {
                S.push(P);
                P = P.getLeft();
        }
        else {
                while(P.getRight() == null)
                        P = S.pop();
                P = P.getRight();
        }
        return P;
}
```

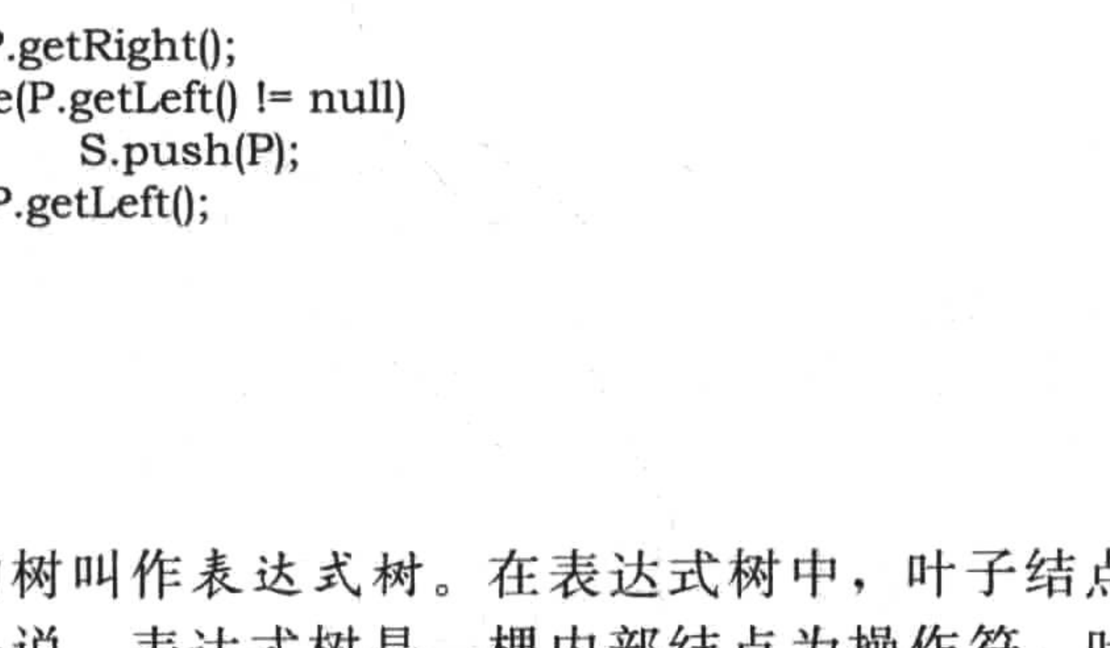
**问题 46** 给定一棵二叉树(非线索二叉树)，如何找到其中序遍历后继？

**解答：** 与上述讨论类似，利用如下算法能够找到某个结点的中序遍历后继：

```
//查找非线索二叉树的中序序列后继
BinaryTreeNode InorderSuccssor(BinaryTreeNode node) {
        static BinaryTreeNode P;
        LLStack S = new LLStack();
        if(node != null)
                P = node;
        if(P.getRight() == null)
                P = S.pop();
        else {
                P = P.getRight();
                while(P.getLeft() != null)
                        S.push(P);
                P = P.getLeft();
        }
        return P;
}
```

## 6.7 表达式树

用来表示表达式的树叫作表达式树。在表达式树中，叶子结点是操作数，而非叶子结点是操作符。也就是说，表达式树是一棵内部结点为操作符，叶子结点为操作数的二叉树。表达式树由二元表达式组成。但是，对于一元操作符，一个子树将为空。下图为表达式：(A+B * C)/D 所对应的一个简单表达式树。

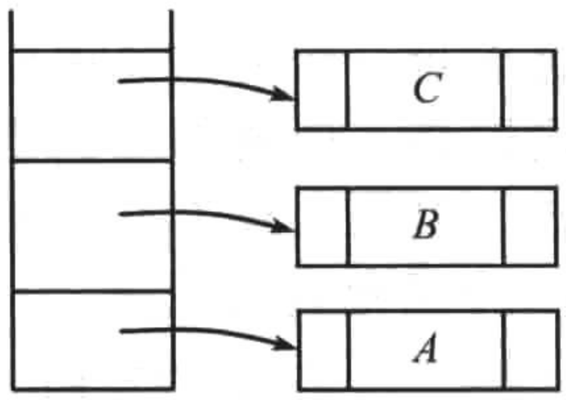


### 1. 基于后缀表达式来构建表达式树的算法

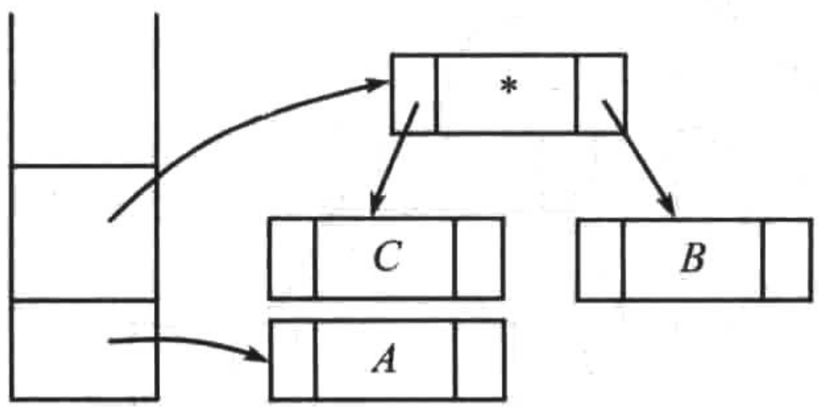
```
BinaryTreeNode BuildExprTree(char postfixExpr[], int size) {
        LLStack S = new LLStack();
        for(int i = 0; i< size; i++) {
                if(postfixExpr[i] is an operand) {
                        BinaryTreeNode newNode = new BinaryTreeNode();
                        if(newNode == null) {
                                System.out.println("Memory Error");
                                return null;
                        }
                        newNode.setData(postfixExpr[i]);
                        newNode.setLeft(null);
                        newNode.setRight(null);
                        S.push(newNode);
                }
                else {  BinaryTreeNode T2 = S.pop(), T1 = S.pop();
                        BinaryTreeNode newNode = new BinaryTreeNode();
                        if(newNode == null) {
                                System.out.println("Memory Error");
                                return null;
                        }
                        newNode.setData(postfixExpr[i]);
                        //使T2和T1分别为新结点的右孩子和左孩子
                        newNode.setLeft(T1); newNode.setRight(T2);
                        S.push(newNode);
                }
        }
        return S;
}
```

### 2. 例子

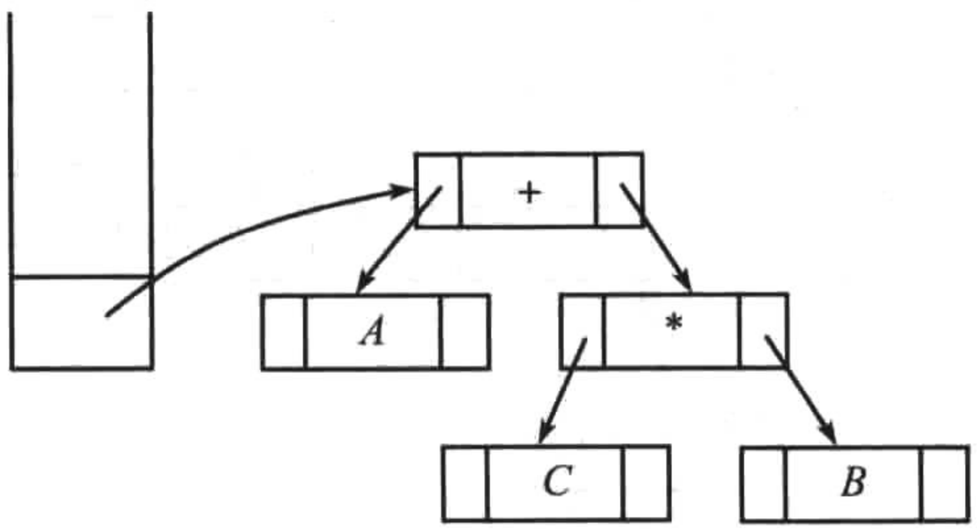
假定每次读入一个符号。如果该符号是操作数，就创建一个结点，并把指向该结点的指针入栈。如果该符号是操作符，则从栈中弹出两个指向树 $T_1$ 和 $T_2$ 的指针(其中 $T_1$ 先出栈)，并产生一棵新树，该树以读到的操作符作为根结点，两个指针分别作为根结点的左孩子结点和右孩子结点，然后再将指向新树的指针入栈。例如，假定输入的后缀表达式为：$A\ B\ C\ *\ +D/$。那么前 3 个符号是操作数，所以产生 3 个结点，并将它们入栈，如下图所示。



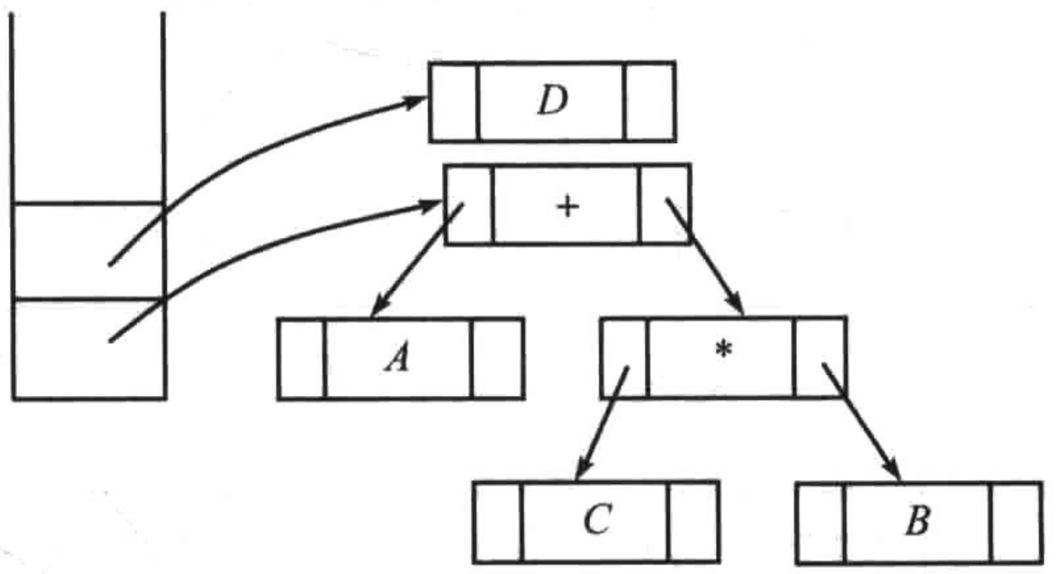

接下来，读入操作符‘ * ’，因此栈中指向两棵树的指针出栈，并形成一棵新树，最后将指向新树的指针入栈。如下图所示。



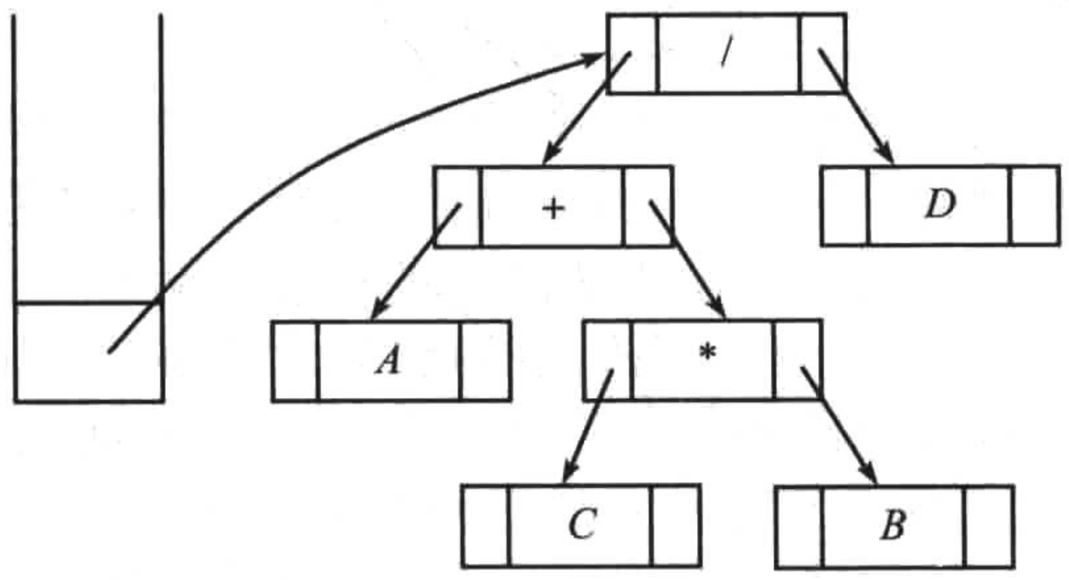
接下来，读入操作符‘+’。因此将指向树的两个指针出栈，形成一棵新树，并将指向新树的指针入栈。




接下来，读入操作数‘*D*’，产生包含一个结点的树，将指向该树的指针入栈。



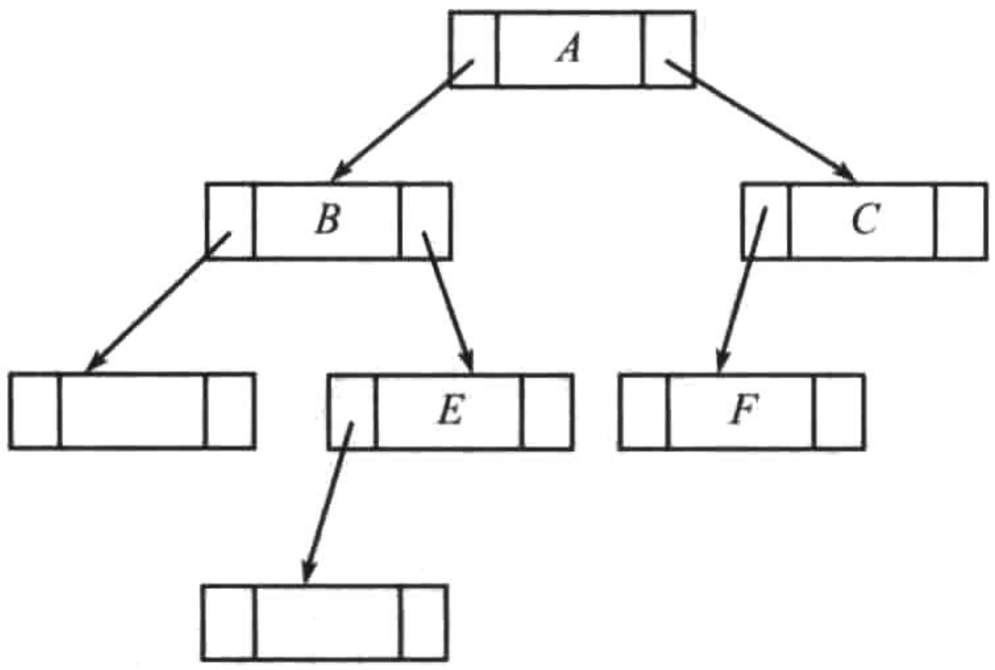

最后，读入操作符‘/’，将栈中指针所对应的两棵树合并，并将指向最后树的指针入栈，如下图所示。



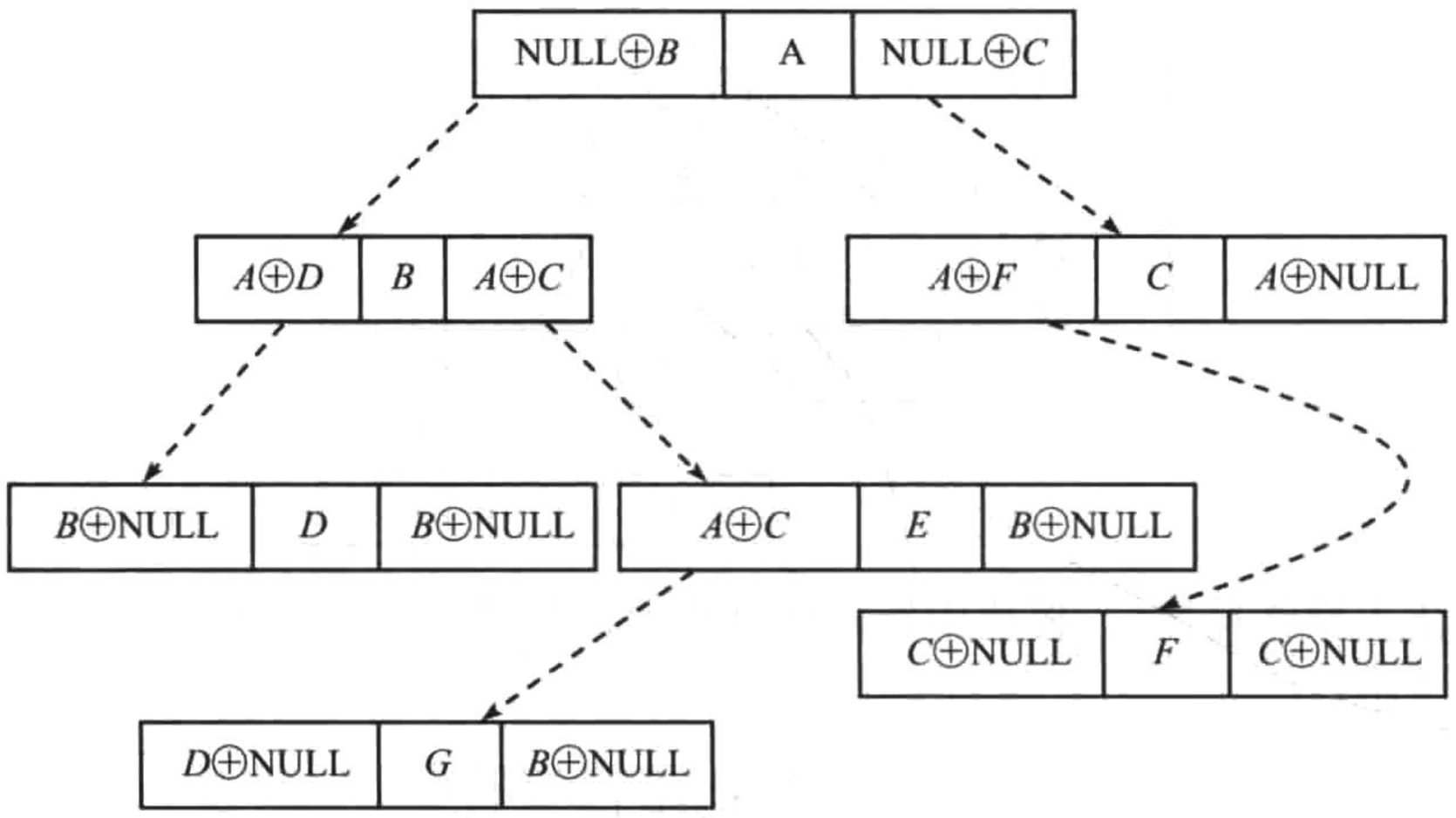

## 6.8 异或树

这个概念与第3章中的存储高效的双向链表类似。并且，与线索二叉树类似，这种表示方式不需要栈或队列来遍历树。这种表示利用⊕操作来进行后向遍历(至双亲结点)和前向遍历(至孩子结点)。为了表示的一致性，每个结点都遵从以下规则：

- 每个结点的 left 字段为其双亲结点与其左孩子结点的异或。
- 每个结点的 right 字段为其双亲结点与其右孩子结点的异或。
- 根结点的字段结点为空(NULL)，并且叶子结点的孩子结点也是空(NULL)结点。




基于以上规则和讨论，树可以表示如下：




该表示方法的主要目的是能够方便地找到结点的双亲结点和孩子结点。下面讨论如何利用该表示方法遍历整棵树。例如，假设当前结点为 $B$，需要移动到其双亲结点 $A$，那么仅需将其左孩子的地址与上述结点左边的内容进行异或运算(也可以基于右孩子结点寻找双亲结点)。类似地，如果需要访问其孩子结点(如左孩子 $D$)，那么仅需将其双亲结点的地址与结点左边的内容进行异或运算。上述过程的一个关键是：如何知道 $B$ 的孩子结点 $D$ 的地址信息？由于遍历始于根结点，所以将根结点左边的值与 NULL 进行异或运算就能得到其左孩子结点 $B$，然后将 $A$ 的地址与 $B$ 左边的值进行异或运算就可以知道 $D$ 的地址。

## 6.9 二叉搜索树

### 1. 为什么要使用二叉搜索树

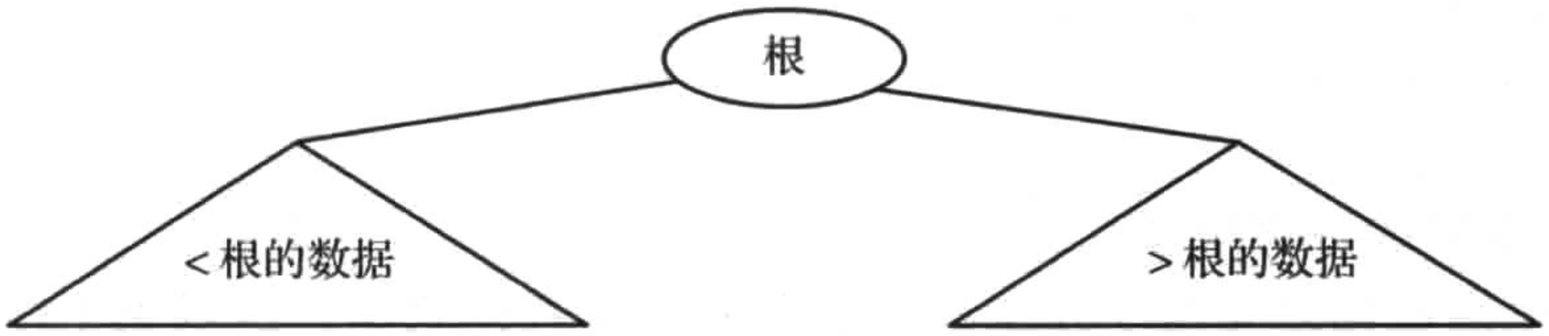
在前面的章节中，已经讨论了树的不同表示方法。在这些表示方法中，对于结点的数据没有任何限制。因此，为了搜索一个元素，需要分别检查左子树和右子树。因而，在最坏情况下，搜索的时间复杂度为 $O(n)$。

在本节中，我们将讨论另二叉树的另一种变型：二叉搜索树(BST)。顾名思义，这种表达方式主要用来实现搜索操作。在这种表示中，对结点所包含的数据进行了一定的约束。因此它使得最坏情况下平均搜索的时间复杂度降低至 $O(\log n)$。

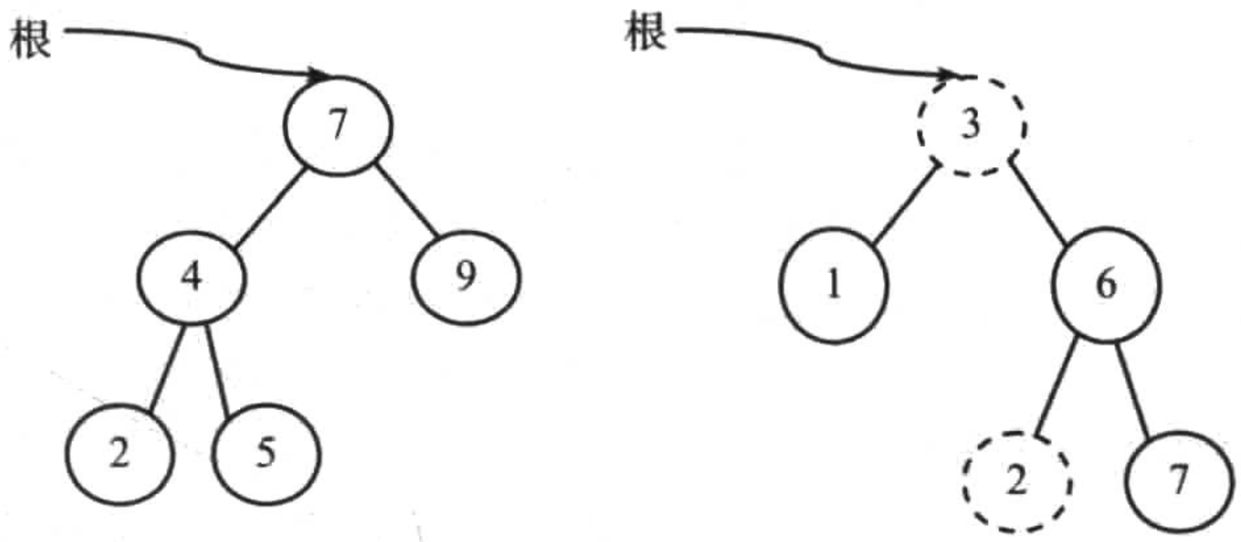
### 2. 二叉搜索树的性质

在二叉搜索树中，所有左子树结点的元素小于根结点数据，所有右子树结点的元素大于根结点数据。这就是二叉搜索树的性质。注意，树中每个结点都应满足这个性质。

- 一个结点的左子树只能包含键值小于该结点键值的结点。
- 一个结点的右子树只能包含键值大于该结点键值的结点。
- 左子树和右子树也都必须是二叉搜索树。




**例子：**左子树是一棵二叉搜索树，而右树不是二叉搜索树(结点 6 不满足二叉搜索树的性质)。




### 3. 二叉搜索树的声明

二叉搜索树的声明与一般二叉树的声明没有区别。唯一的区别在于数据而不是结构。但为了方便起见，重新定义结构名如下：

```
public class BinarySearchTreeNode {
        private int data;
        private BinarySearchTreeNode left;
        private BinarySearchTreeNode right;
        public int getData() {
                return data;
        }
        public void setData(int data) {
                this.data = data;
        }
        public BinarySearchTreeNode getLeft() {
                return left;
        }
        public void setLeft(BinarySearchTreeNode left) {
                this.left = left;
        }
        public BinarySearchTreeNode getRight() {
                return right;
        }
        public void setRight(BinarySearchTreeNode right) {
                this.right = right;
        }
}
```

### 4. 二叉搜索树的操作

**主要操作：**

- 在二叉搜索树中查找元素/查找最小元素/查找最大元素。
- 在二叉搜索树中插入元素。
- 在二叉搜索树中删除元素。

**辅助操作：**

- 判定一棵树是否为二叉搜索树。
- 寻找树中第 $k$ 小的元素。
- 给二叉搜索树中元素排序等其他操作。

### 5. 二叉搜索树的注意事项

- 由于根结点数据总是处于左子树数据和右子树数据之间，所以当中序遍历二叉搜索树时，将得到一个有序表。
- 当处理二叉搜索树中的问题时，首先处理左子树，然后是根结点，最后是右子树。这就意味着，处理不同问题仅取决于中间步骤(处理根结点数据)的不同，不需要涉及第一步和第三步。
- 搜索一个元素时，如果左子树根结点数据小于要搜索的元素值，则跳过它。同样，右子树也是这样……正因为如此，所以二叉搜索树在搜索一个元素时所需要的时间就小于一般的二叉树。换句话说，二叉搜索树在搜索元素时，要么在左子树中搜索，要么在右子树中搜索，不需要同时在两棵子树中搜索。

### 6. 在二叉搜索树中寻找元素

二叉搜索树的寻找操作简单有效。从根结点开始，基于二叉搜索树的性质移动到左子树或右子树继续搜索。如果待搜索数据与根结点数据一致，则返回当前结点。如果待搜索数据小于根结点数据，则搜索当前结点的左子树；否则，搜索当前结点的右子树。如果数据不存在，则返回一个空指针。

```
BinarySearchTreeNode Find(BinarySearchTreeNode root, int data) {
        if( root == null) return null;
        if( data < root.getData() )
                return Find(root.getLeft(), data);
        else if( data > root.getData() )
                return( Find(root.getRight(), data );
        return root;
}
```

时间复杂度：在最坏情况下(当给定二叉搜索树为一棵斜树时)为 $O(n)$。空间复杂度为 $O(n)$，用于递归栈。

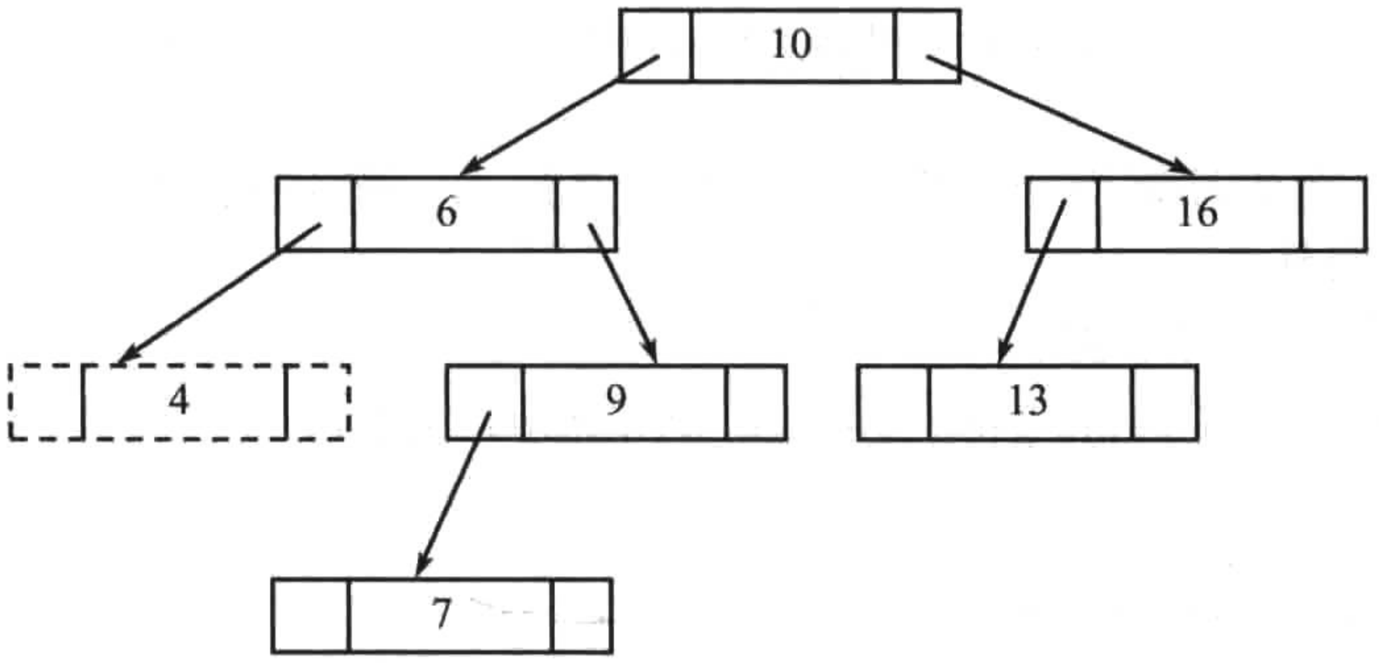
上面算法的非递归版本为：

```
BinarySearchTreeNode Find(BinarySearchTreeNode root, int data ) {
        if( root == null)  return null;
        while(root != null) {
                if(data == root.getData())
                        return root;
                else if(data > root.getData())
                        root = root.getRight();
                else      root = root.getLeft();
        }
        return null;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**7. 在二叉搜索树中寻找最小元素**

在二叉搜索树中，最左边的结点为最小元素，它没有左子结点。在下图所示的二叉搜索树中，最小元素为 **4**。




```
BinarySearchTreeNode FindMin(BinarySearchTreeNode root) {
        if(root == null)
            return null;
        else
            if( root.getLeft() == null ) return root;
            else return FindMin(root.getLeft() );
}
```

时间复杂度：在最坏情况下(当给定二叉搜索树为一棵斜树时)为 $O(n)$。空间复杂度为 $O(n)$，用于递归栈。

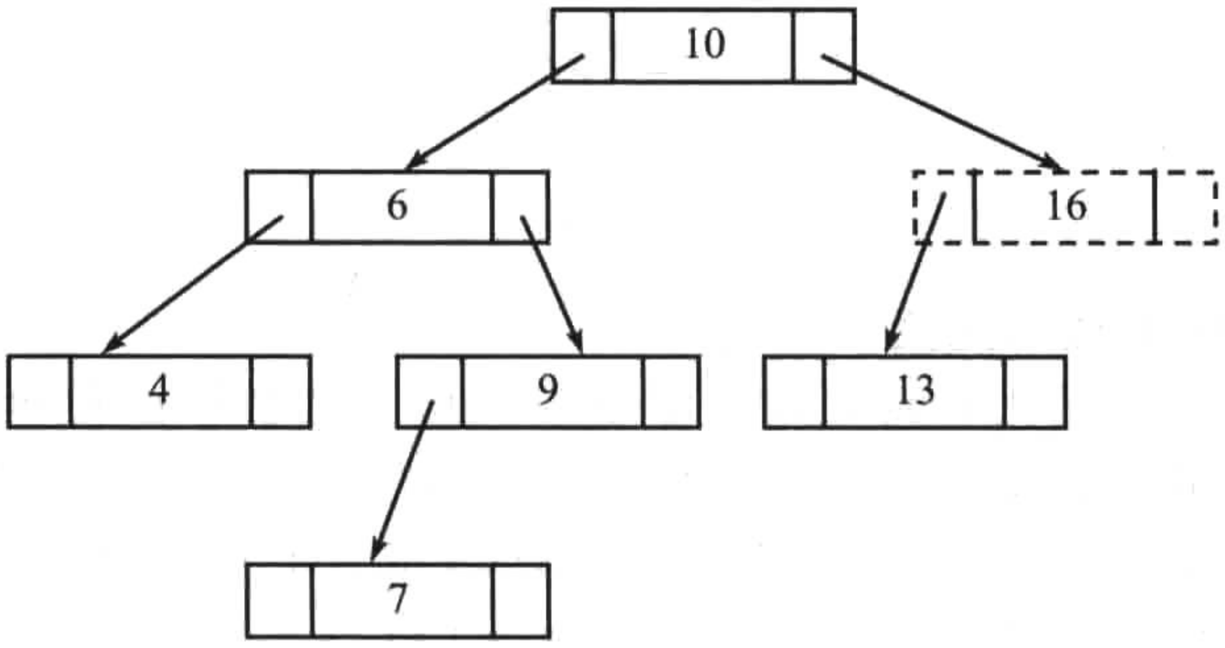
上面算法的非递归实现为：

```
BinarySearchTreeNode FindMin(BinarySearchTreeNode  root ) {
        if( root == null)
                return null;
        while( root.getLeft() != null) root = root.getLeft();
        return root;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**8. 在二叉搜索树中寻找最大元素**

在二叉搜索树中，最大元素在树的最右端，它没有右子结点。在下图所示的二叉搜索树中最大元素为 **16**。



```
BinarySearchTreeNode FindMax(BinarySearchTreeNode root) {
        if(root == null)
                return null;
        else
                if( root.getRight() == null)
                        return root;
                else    return FindMax( root.getRight());
}
```

时间复杂度：在最坏情况下(当给定二叉搜索树为一棵斜树时)为 $O(n)$。空间复杂度为 $O(n)$，用于递归栈。

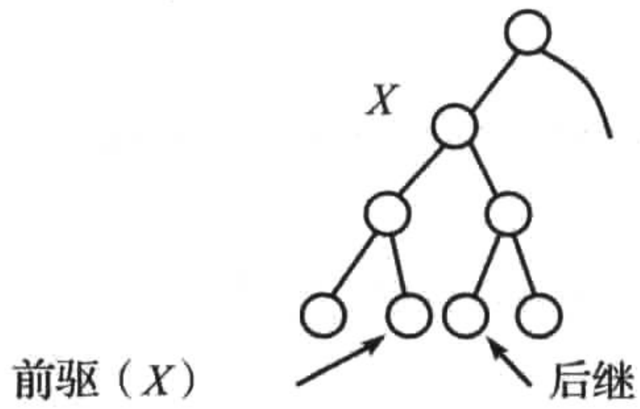
上面算法的非递归实现为：

```
BinarySearchTreeNode FindMax(BinarySearchTreeNode  root ) {
        if( root == null)
                return null;
        while( root.getRight() != null)
            root = root.getRight();
        return root;
}
```

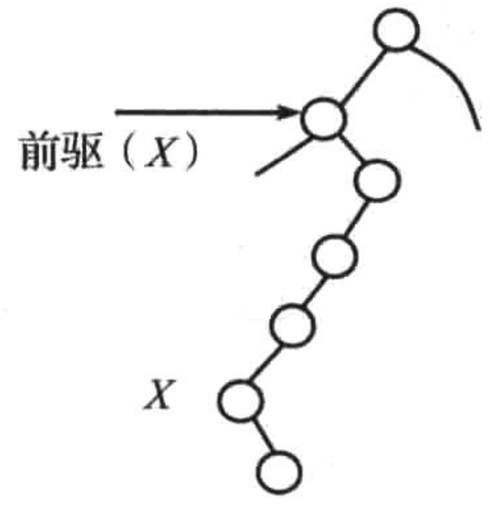
时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**9. 寻找中序序列前驱和后继**

假定二叉搜索树中所有关键字值是唯一的，那么树中节点 $X$ 的中序序列前驱和后继在哪里？如果 $X$ 有两个孩子结点，那么中序序列前驱为其左子树中值最大的元素，而其后继为其右子树中的最小元素。

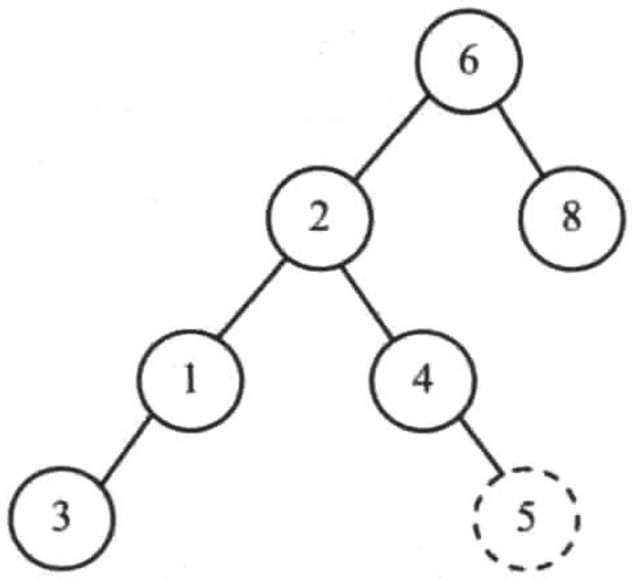



如果它没有左孩子结点，则该结点中序序列前驱是其第一个左祖先结点。




**10. 在二叉搜索树中插入元素**

为了在二叉搜索树中插入数据，首先需要找到插入该数据的位置。遵循与寻找(find)操作相同的机制，就能够找到插入的位置。当找到插入位置时；如果该数据已经在处于该位置，那么只需要忽略并返回。否则，将该数据插入走过的路径的最后位置上。以下图所示的这棵树为例，虚线结点表示要插入的元素(5)。




为了插入 5，使用 find 函数遍历这棵树。在键值为 4 的结点处，需要访问其右子树，但是由于结点 4 没有右子树，所以 5 没有在树中，因此这就是要插入的位置。

```
BinarySearchTreeNode Insert(BinarySearchTreeNode root, int data) {
        if( root == null) {
                root = new BinarySearchTreeNode();
                if( root == null) {
                        System.out.println("Memory Error");
                        return;
                }
                else {
                        root.setData(data);
                        root.setLeft(null); root.setRight(null);
                }
        }
        else {
                if( data < root.getData() )
                        root.setLeft(Insert(root.getLeft(), data));
                else if( data > root.getData() )
                        root.setRight(Insert(root.getRight(), data));
        }
        return root;
}
```

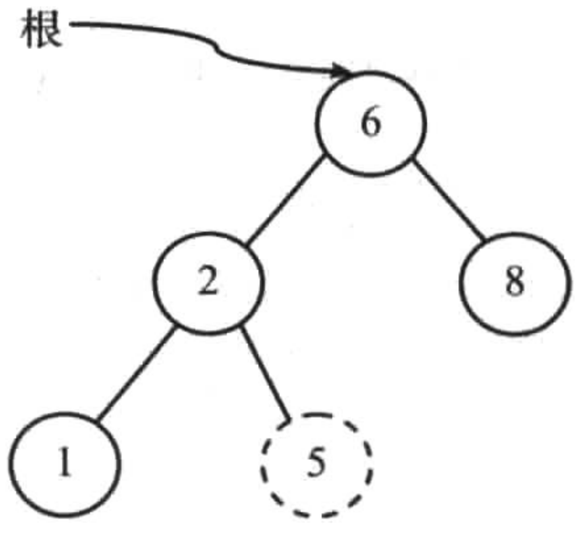
**注意**：在上述代码中，在元素插入子树后，该树将返回到其双亲结点。因此，整棵树都将更新。

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于递归栈。对于迭代算法，空间复杂度为 $O(1)$。

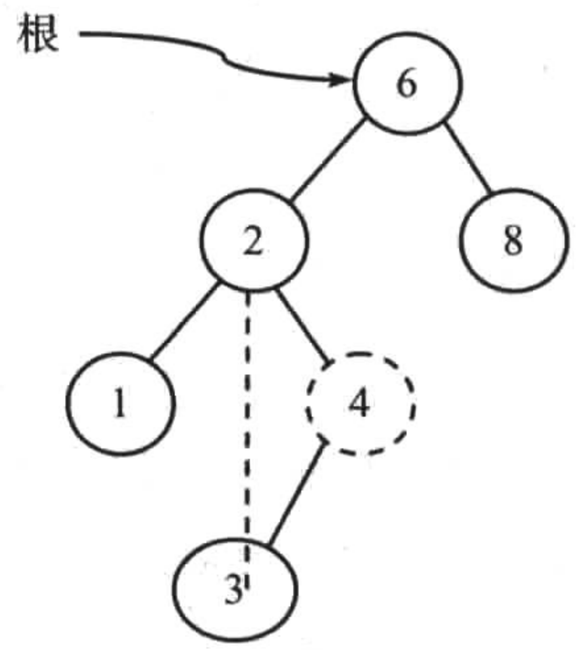
### 11. 在二叉搜索树中删除元素

删除操作比其他操作都复杂。这是因为待删除的元素可能不是叶子结点。在这个操作中，首先需要找到待删除元素的位置。一旦找到待删除的元素，需要考虑以下情况：

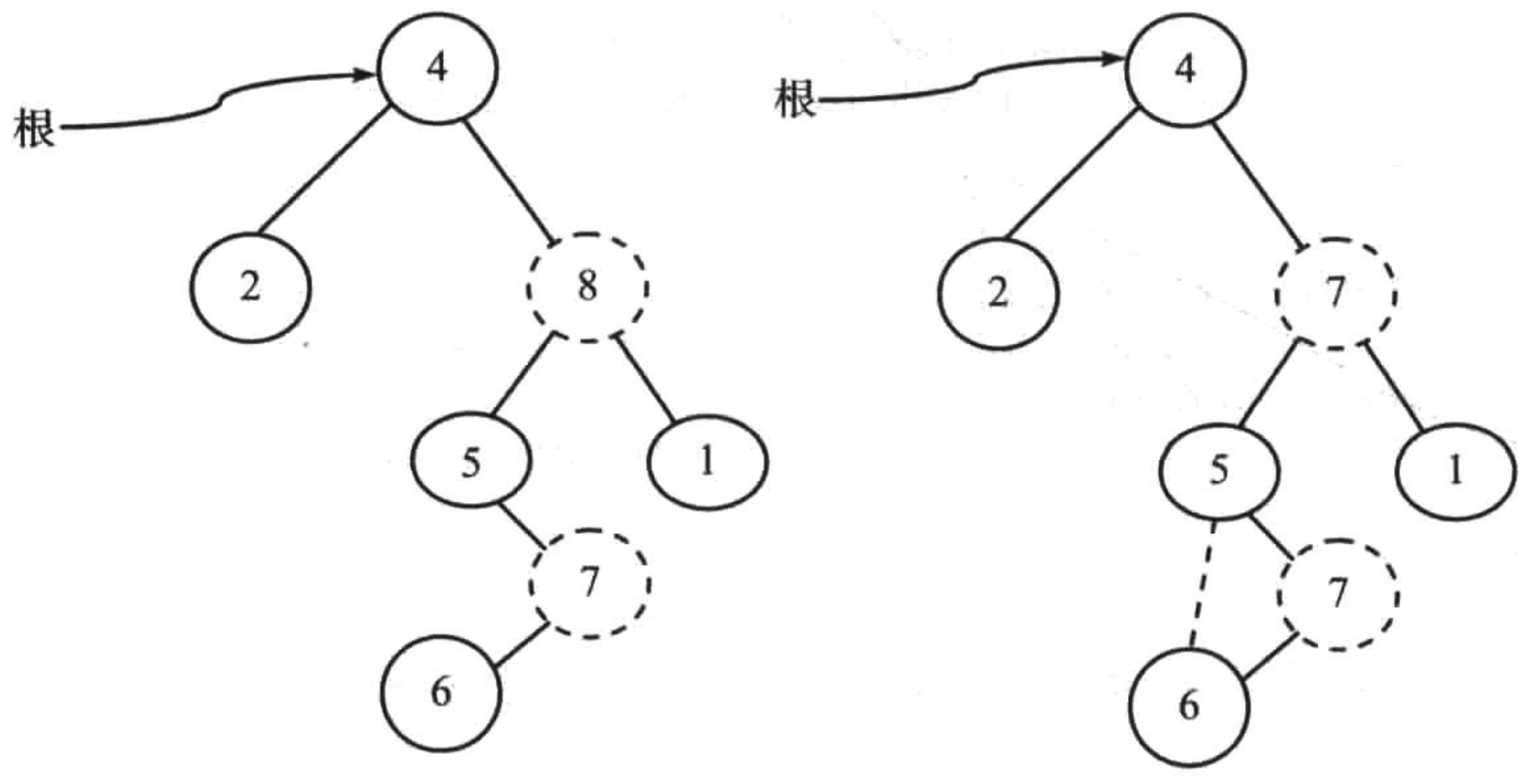
- 如果待删除元素为叶子结点，则返回 NULL 给其双亲结点，即将其相应的孩子结点指针设置为 NULL。在下面的树中删除 5，将其双亲结点 2 的孩子指针设置为 NULL。



- 如果待删除结点有一个孩子结点，在这种情况下，只需要将待删除结点的孩子结点返回给双亲结点。如下图所示，要删除 4，4 的左子树设置为其双亲结点 2 的一棵子树。




- 如果待删除元素有两个孩子结点，通常的做法是从其左子树中找到最大元素来代替这个结点的主键，然后再删除那个结点(现在为空)。左子树中最大元素结点没有右孩子，第二次删除操作是很容易的。




例如，要删除元素为 8，它是根结点的右孩子结点。主键是 8。用它的左子树(7)中最大主键来代替它，然后再删除 7 这个结点(第二种情况)。

**注意：** 也可以用右子树中的最小元素来代替。

```
BinarySearchTreeNode Delete(BinarySearchTreeNode root, int data) {
        BinarySearchTreeNode temp;
        if( root == null)
        System.out.println("Element not there in tree");
else if(data < root.data)
        root.left = Delete(root.getLeft(),data);
else if(data > root→ data)
        root.right = Delete(root.getRight(), data);
else {    // 找到该元素
        if( root.getLeft() && root.getRight() ) {
                /*用左子树中的最大值代替*/
                temp = FindMax( root.getLeft() );
                root.getData() = temp.data;
                root.left = Delete(root.getLeft(), root.getData());
        }
        else {              /*一个孩子结点*/
```

```
                        temp = root;
                        if( root.getLeft() == null )
                                root = root.getRight();
                        if( root.getRight() == null)
                                root = root.getLeft();
                        free( temp );
                }
        }
        return root;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于递归栈。对于迭代算法，空间复杂度为：$O(1)$。

### 12. 二叉搜索树的相关问题

**问题 47**　给出在 BST 树中寻找两个结点之间最短路径的算法。

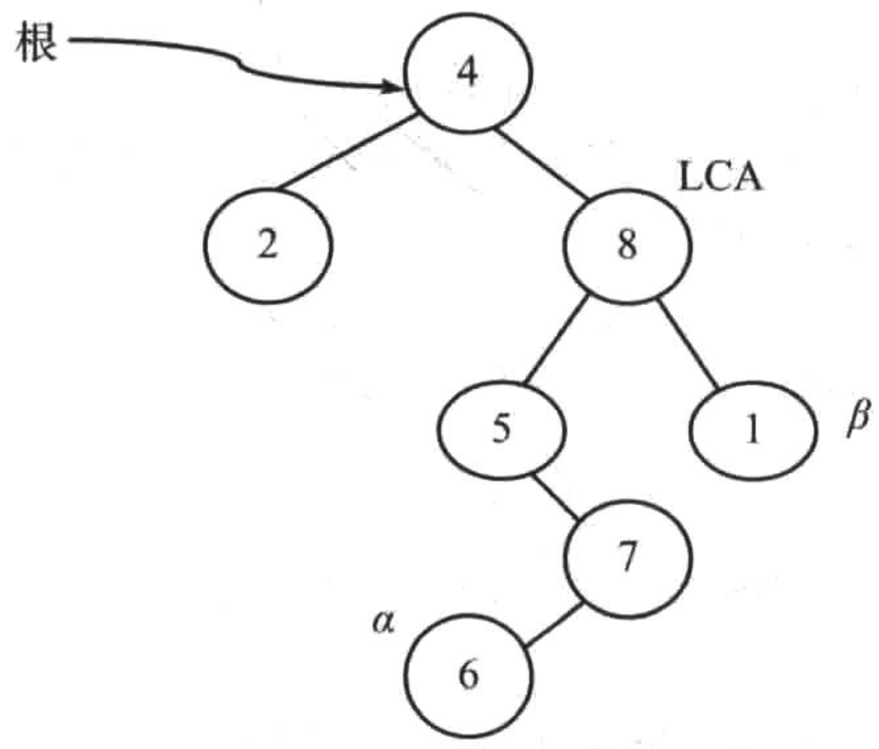
**解答：**只需要在 BST 树中寻找两个结点的 LCA 即可。

**问题 48**　给出计算具有 $n$ 个结点的 BST 树的个数的算法。

**解答：**这是一个 DP 问题，请参见第 19 章。

**问题 49**　已知二叉搜索树中指向两个结点的指针，寻找它们之间的最近公共祖先(LCA)。假定这两个值已经在这棵树中。

**解答：**求解这个问题的主要思路是：当从根结点到底部遍历这棵 BST 树时，遇到 $\alpha \sim \beta$ 之间的值(即，$\alpha <$ node→data $< \beta$)的第一个结点，就是 $\alpha$ 和 $\beta$ 的最近公共祖先结点。所以在前序遍历 BST 树时，如果找到某结点的值为 $\alpha \sim \beta$，那么这个结点就是 LCA。如果结点的值大于 $\alpha$ 和 $\beta$，则 LCA 应处于该结点的左侧；如果结点的值小于 $\alpha$ 和 $\beta$，则 LCA 处于其右侧。




```
BinarySearchTreeNode FindLCA(BinarySearchTreeNode root, BinarySearchTreeNode α,
BinarySearchTreeNode  β) {
        while(1) {
                if((α.getData() < root.getData() && β.getData() > root.getData()) ||
                        (α.getData() > root.getData() && β.getData() < root.getData()))
                        return root;
                if(α.getData() < root.getData())
                        root = root.getLeft();
                else    root = root.getRight();
        }
}
```

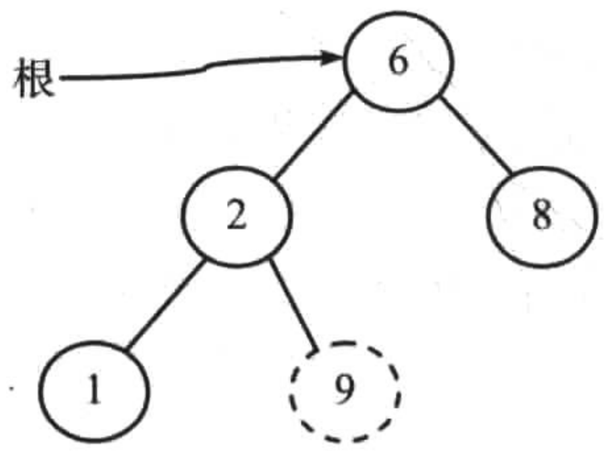
时间复杂度为 $O(n)$。对于斜树，空间复杂度为 $O(n)$。

**问题 50** 给出算法，判断一棵给定的二叉树是否为 BST 树。

**解答：** 可以参考下面的简单程序。对于每个结点，检查其左孩子结点是否比它小，而右孩子结点是否比它大。

```
int IsBST(BinaryTreeNode root) {
        if(root == null) return 1;
        /*若左孩子结点大于根结点，返回false*/
        if(root.getLeft() != null && root.getLeft().getData() > root.getData())
                return 0;
        /*若右孩子结点大于根结点，返回false*/
        if(root.getRight() != null && root.getRight().getData() < root.getData())
                return 0;
        /*递归判断左子树和右子树，若其中有任何一棵子树不是BST树，则返回false*/
        if(!IsBST(root.getLeft()) || !IsBST(root.getRight()))
                return 0;
        /*若通过所有的判断，则是一棵BSF树*/
        return 1;
}
```

但这种方法是不对的，例如下图所示的二叉树，算法将返回 true。因此仅仅检查当前结点是不够的。




**问题 51** 如何得到求解问题 50 的正确算法？

**解答：** 对于每个结点，需要检查其左子树中的最大值是否小于当前结点的值，且右子树中的最小值是否大于该结点的值。

```
/*若二叉树是一棵二叉搜索树，则返回true*/
int IsBST(BinaryTreeNode root) {
        if(root == null)
                return 1;
        /*或左子树的最大值大于根结点，返回false*/
        if(root.getLeft() != null && FindMax(root.getLeft()) > root.getData())
                return 0;
        /*或右子树的最小值小于根结点，返回false*/
        if(root.getRight() != null && FindMin(root.getRight()) < root.getData())
                return 0;
        /*递归判断左子树和右子树，若其中有任何一棵子树不是BSF树，则返回false*/
        if(!IsBST(root.getLeft()) || !IsBST(root.getRight()))
                return 0;
        /*若通过所有的判断，则是一棵BSF树*/
        return 1;
}
```

这里假定函数 FindMin()和 FindMax()能得到一棵非空树的最小值或最大值。

时间复杂度为 $O(n^2)$。空间复杂度为 $O(n)$。

**问题 52** 能否降低问题 51 算法的复杂度？

**解答：可以，**一个更好的算法是对每个结点只需要计算一次。关键是实现一个实用

的函数 ISBSTUtil(BinaryTreeNode * root, int min, int max)，该函数在遍历时，跟踪 min 和 max 值的变化，这样每个结点只需要计算一次。Min 和 max 值初始化值为 INT _ MIN 和 INT _ MAX。

```
Initial call: IsBST(root, INT_MIN, INT_MAX);
int IsBST(BinaryTreeNode root, int min, int max) {
        if(root == null)
            return 1;
        return (root.getData() >min && root.getData() < max &&
            IsBSTUtil(root.getLeft(), min, root.getData()) &&
            IsBSTUtil(root.getRight(),  root.getData(), max));
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 53** 是否能进一步降低问题 51 的复杂度？

**解答：可以**，使用中序遍历。这个思路是，中序遍历将产生有序表，因此在中序遍历时，对每个结点，判断其值是否都大于其前面所访问结点的值。利用变量 prev 来保存结点前驱的值，并初始化 prev 为可能的最小值(假设为 INT _ MIN)。

```
int prev = INT_MIN;
int IsBST(BinaryTreeNode root, int prev) {
        if(root == null) return 1;
        if(!IsBST(root.getLeft(), prev))
            return 0;
        if(root.getData() < prev)
            return 0;
        prev = root.getData();
        return IsBST(root.getRight(), prev);
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于栈空间开销。

**问题 54** 给出空间复杂度为 $O(1)$的算法将 BST 树转化为循环双向链表。

**解答：**将左子树和右子树转化为双向链表，并维护它们的端点。然后，调整指针。

```
BinarySearchTreeNode BST2DLL(BinarySearchTreeNode root, BinarySearchTreeNode Ltail) {
        BinarySearchTreeNode left, ltail, right, rtail;
        if(root == null) {
            ltail = null;
            return null;
        }
        left = BST2DLL(root.getLeft(), ltail);
        right = BST2DLL(root.getRight(), rtail);
        root.setLeft(ltail);
        root.setRight(right);
        if(right == null)
            ltail = root;
        else {
            right.setLeft(root);
            ltail = rtail;
        }
        if(left == null)
            return root;
        else {    ltail.setRight(root);
            return left;
        }
}
```

时间复杂度为 $O(n)$。

**问题 55** 给定一个有序双向链表，设计算法将其转换为平衡二叉搜索树。

**解答：** 寻找中间结点，然后调整指针。

```
DLLNode DLLtoBalanced BST(DLLNode head) {
        DLLNode temp, p, q;
        if( head == null || head.getNext() == null)
                return head;
        temp = FindMiddleNode(head);
        p = head;
        while(p.getNext()! = temp)
                p = p.getNext();
        p.setNext(null);
        q = temp.getNext();
        temp.setNext(null);
        temp.setPrev(DLLtoBalancedBST(head));
        temp.setNext(DLLtoBalancedBST(q));
        return temp;
}
```

时间复杂度为 $2T(n/2)+O(n)$(寻找中间结点)$=O(n\log n)$。

**注意：** 对于 FindMiddleNode 函数参见第 3 章。

**问题 56** 给定一个有序数组，设计算法将其转换为 BST。

**解答：** 如果需要从数组中选择一个元素作为平衡 BST 树的根结点，应该选择哪个结点呢？平衡 BST 树的根结点应该是有序数组的中间元素。在每次迭代中，选择有序数组的中间元素作为子树的根结点。然后以这个元素构建树的一个结点。当该选择元素后，下一步如何做？能否将此问题分解成子问题？分成两个数组——一个位于该元素的左边，另一个位于它的右边。这两个数组就是原始问题的子问题，因为它们都是有序的。而且，它们是当前结点的左子树和右子树。下面的代码实现了在 $O(n)$时间内基于有序数组创建平衡 BST 树($n$ 是数组中元素的个数)。比较而言，该算法与二叉搜索算法很类似，都利用了分治法。

```
BinaryTreeNode BuildBST(int A[], int left, int right) {
        BinaryTreeNode newNode;
        if(left > right)
            return null;
        newNode = new BinaryTreeNode();
        if(newNode == null) {
                System.out.println("Memory Error"); return;
        }
        if(left == right) {
                newNode.setData(A[left]);
                newNode.setLeft(null);
                newNode.setRight(null);
        }
        else {
                int mid = left + (right-left)/ 2;
                newNode.setData(A[mid]);
                newNode.setLeft(BuildBST(A, left, mid - 1));
                newNode.setRight(BuildBST(A, mid + 1, right));
        }
        return newNode;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于递归栈。

**问题 57** 给定一个元素按升序排列的单向链表，将其转换为一棵高度平衡 BST 树。

**解答：** 一个最简单的方法就是直接应用问题 55 的求解方法。在每次递归调用时，都需要遍历链表长度的一半来寻找中间结点。显然运行时间复杂度为 $O(n\log n)$，其中 $n$ 为链表中元素的个数。这是因为每一层的递归调用都需要对链表遍历 $n/2$ 次，并且总共有 $\log n$ 层(即平衡树的高度)。

**问题 58** 对于问题 57，能否降低其复杂度？

**解答：**

**提示：** 可以根据链表顺序来插入结点。如果能成功做到这点，将不再需要寻找中间元素，因为这样就可以在遍历链表的同时将元素作为结点插入树中。

**最好的解决方案：** 通常，最好的方法需要从另一个角度来思考问题。换句话说，即不再使用自顶向下的方法来创建树的结点。而是用自底向上的方法来创建结点，并指定它们的双亲结点。自底向上的方法能够以链表的顺序来创建树的结点(42)。

自底向上的方法是否有效呢？当自顶向下的方法难以解决问题时，都可以尝试用自底向上的方法。尽管自底向上的方法不是最自然的思考方式，但在某些场合中是很有帮助的。然而，通常首选自顶向下而不是自底向上的方法，因为后者更加难验证。

下面的代码实现了将单向链表转化为平衡 BST 树。注意，该算法需要将链表长度作为函数的输入参数。链表长度通过遍历整个链表一次可以在 $O(n)$时间内获取。递归调用遍历链表，并按照链表顺序产生树的结点，需要花费 $O(n)$的时间。因此，整个时间复杂度还是 $O(n)$。

```
BinaryTreeNode SortedListToBST(ListNode list, int start, int end) {
        if(start > end)
                return null;
        // 等同于(start+end)/2，避免溢出
        int mid = start + (end - start) / 2;
        BinaryTreeNode leftChild = SortedListToBST(list, start, mid-1);
        BinaryTreeNode  parent = new BinaryTreeNode();
        if(parent == null)
                System.out.println("Memory Error"); return;
        parent.setData(list.getData());
        parent.setLeft(leftChild);
        list = list.getNext();
        parent.setRight(SortedListToBST(list, mid+1, end));
        return parent;
}
BinaryTreeNode  SortedListToBST(ListNode head, int n) {
        return SortedListToBST(head, 0, n-1);
}
```

**问题 59** 给出查找 BST 树中第 $k$ 小元素的算法。

**解答：** 求解的思路是，利用中序遍历 BST 树来产生有序表。在中序遍历 BST 树时，记录已访问过的元素的个数。

```
BinarySearchTreeNode kthSmallestInBST(BinarySearchTreeNode root, int k, int count) {
        if(root == null)
                return null;
        BinarySearchTreeNode left = kthSmallestInBST(root.getLeft(), k, count);
        if( left != null)
                return left;
```

```
        if(++count == k)
            return root;
        return kthSmallestInBST(root.getRight(), k, count);
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**问题 60　上下界：**如果给定的键值小于 BST 树根结点的键值，那么该键值的下界(BST 树中小于或等于键值的最大键值)必定在左子树中。如果该键值大于根结点的键值，那么该键值的下界可能在右子树中，但仅当在右子树中有一个键值小于或等于该键值；否则(如果该键值等于根结点的键值)，那么根结点的键值是该键值的下界。查找上界类似于左右交换。例如，如果有序数组是{1，2，8，10，10，12，19}，那么

对于 $x=0$，数组中不存在 floor，而 ceil=1。对于 $x=1$，floor=1 且 ceil=1。

对于 $x=5$，floor=2 且 ceil=8。对于 $x=20$，floor=19，但数组中不存在 ceil。

**解答：**求解思路是，中序遍历 BST 将产生有序表。当以中序遍历 BST 树时，记录访问的值。如果根结点大于给定的值，那么返回遍历过程中维护的前一个结点的值。如果根结点数据等于给定数据，则返回根结点数据。

```
BinaryTreeNode FloorInBST(BinaryTreeNode root, int data) {
        BinaryTreeNode prev=null;
        return FloorInBSTUtil(root, prev, data);
}
BinaryTreeNode FloorInBSTUtil(BinaryTreeNode root, BinaryTreeNode prev, int data) {
            if(root == null) return null;
            if(!FloorInBSTUtil(root.getLeft(), prev, data))
                return 0;
            if(root.getData() == data) return root;
            if(root.getData() > data) return prev;
            prev = root;
            return FloorInBSTUtil(root.getRight(), prev, data);
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于栈空间。

对于 ceiling，只需要先调用右子树，然后再调用左子树。

```
BinaryTreeNode CeilingInBST(BinaryTreeNode root, int data) {
        BinaryTreeNode prev=null;
        return CeilingInBSTUtil(root, prev, data);
}
BinaryTreeNode CeilingInBSTUtil(BinaryTreeNode root,        BinaryTreeNode prev, int data) {
            if(root == null) return null;
            if(!CeilingInBSTUtil(root.getRight(), prev, data))        return 0;
            if(root.getData() == data) return root;
            if(root.getData() < data) return prev;
            prev = root;
            return CeilingInBSTUtil(root.getLeft(), prev, data);
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于栈空间。

**问题 61　**给出算法，寻找两棵 BST 树的并集和交集。假定双亲指针可用(例如，线索二叉树)。并且，假定两棵 BST 树的长度分别是 $m$ 和 $n$。

**解答：**如果双亲指针能直接获得，那么问题就变成两个有序链表的合并问题。这是因为每次调用中序序列后继时，能得到下一个最大元素。只是需要决定调用哪一个 Inor-

derSuccessor。

时间复杂度为 $O(m+n)$。空间复杂度为 $O(1)$。

**问题 62** 对于问题 61，如果双亲指针不能直接获得那么应该怎么办？

**解答：** 如果双亲指针不能直接获得，那么可以将 BST 树转换为链表，然后对其合并。

1) 在 $O(m+n)$时间内将两棵 BST 转换为有序的双向链表，得到两个有序表。

2) 将两个双向表合并为一个，同时维护总的元素个数，耗时 $O(m+n)$。

3) 在 $O(m+n)$时间内将有序双向链表转变为高度平衡树。

时间复杂度为 $O(m+n)$。空间复杂度为 $O(\max(m, n))$。

**问题 63** 对于问题 61，是否还有其他的解决方法？

**解答：是的**，使用中序遍历。

- 在其中一棵 BST 树上执行中序遍历。
- 在执行遍历时，将它们存储到表中(散列表)。
- 当完成第一棵 BST 树的遍历后，遍历第二棵 BST 树，然后将它们与散列表中的元素进行比较。

时间复杂度为 $O(m+n)$。空间复杂度为 $O(\max(m, n))$。

**问题 64** 给定一棵 BST 树和两个数 $K1$ 和 $K2$，给出算法，输出 BST 树中所有元素值为 $K1 \sim K2$ 的数。

**解答：**

```
void RangePrinter(BinarySearchTreeNode root, int  K1, int K2) {
        if(root == null)
              return;
        if(root.getData() >= K1)
              RangePrinter(root.getLeft(), K1, K2);
        if(root.getData() >= K1 && root.getData() <= K2)
              System.out.println( root.getData());
        if(root.getData() <= K2)
              RangePrinter(root.getRight(), K1, K2);
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于栈空间。

**问题 65** 对于问题 64，是否还有其他的解决方法？

**解答：** 可以使用层次遍历，在将元素加入队列时检查是否处于这个区间。

```
void  RangeSeachLevelOrder(BinarySearchTreeNode root, int  K1, int K2) {
        BinarySearchTreeNode temp;
        LLQueue Q = new LLQueue();
        if(root == null)   return null;
        Q.enQueue( root);
        while(!Q.isEmpty()) {
                temp=Q.deQueue();
                if(temp.getData() >= K1 && temp.getData() <= K2)
                        System.out.println(temp.getData());
                if(temp.getLeft() && temp.getData() >= K1)
                        Q.enQueue( temp.getLeft());
                if(temp.getRight() && temp.getData() <= K2)
                        Q.enQueue( temp.getRight());
        }
        Q.deleteQueue();
        return null;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，用于队列。

**问题 66** 对于问题 64，还能有其他的方法来解决吗？

**解答：** 首先使用普通二叉搜索方法定位 $K1$，然后使用中序遍历寻找后继，直到遇到 $K2$ 为止。具体算法请参见 6.6 节。

**问题 67** 给定二叉搜索树的根结点，修剪这棵树，使得新树中所有元素处于输入数 $A \sim B$之间。

**解答：** 这只是问题 64 的另一种提问方式。

**问题 68** 给定两棵 BST 树，检查其元素是否相同。例如，两棵 BST 树，其数据分别为：10 5 20 15 30 和 10 20 15 30 5，应该返回 true。而如果两棵 BST 树，其数据分别是：10 5 20 15 30 和 10 15 30 20 5，那应该返回 false。

**注意：** BST 树的数据可以是任意顺序。

**解答：** 简单的方法就是，第一步对第一棵树执行中序遍历并将其数据存储到散列表中。第二步对第二棵树执行中序遍历，然后检查其数据是否已经在散列表中(如果存在，则标记为－1 或者其他唯一数值)。在第二棵树的遍历过程中，如果存在不匹配，则返回 false。当第二棵树遍历完后，检查散列表中所有结点是否都标记为－1(这将保证第一棵树不存在多余的数据)。

时间复杂度为 $O(\max(m, n))$，其中 $m$ 和 $n$ 是第一棵和第二棵 BST 数的元素个数。空间复杂度为 $O(\max(m, n))$，这取决于第一棵树的大小。

**问题 69** 对于问题 68，能否降低其时间复杂度？

**解答：** 不需要对两棵树依次执行中序遍历，而可以同时并行地中序遍历两棵树。因为中序遍历给出有序表，所以只需要检查两棵树是否产生了相同的序列。

时间复杂度为 $O(\max(m, n))$。空间复杂度为 $O(1)$，这取决于第一棵树的大小。

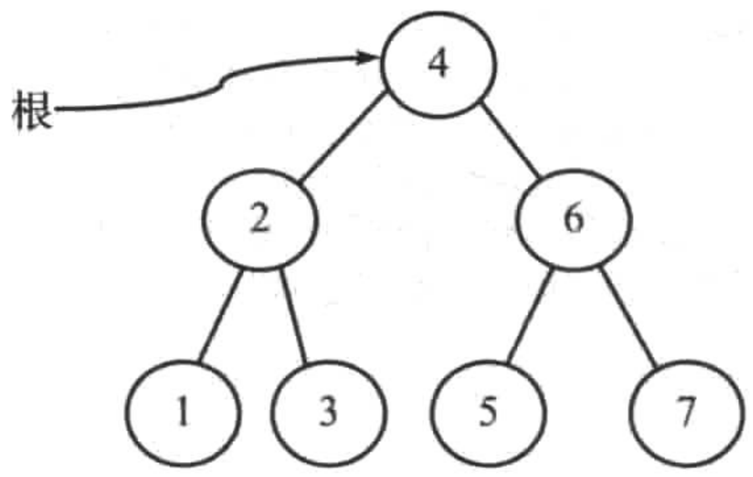
## 6.10 平衡二叉搜索树

从前面章节中已经知道不同类型的树在最坏情况下搜索操作的复杂度为 $O(n)$，其中 $n$ 为树中的结点个数。当树为斜树时就产生这情况。本节将介绍通过限制树的高度可以将最坏情况下的时间复杂度减少至 $O(\log n)$。

通常，高度平衡树用符号 HB($k$)表示，其中 $k$ 为左子树和右子树的高度差。有时也把 $k$ 叫作平衡因子。

**完全平衡二叉搜索树**

在 HB($k$)中，如果 $k=0$(如果平衡因子为 0)，那么就把这种二叉搜索树叫作完全平衡二叉树。即，在 HB(0)的二叉搜索树中，左子树和右子树的高度差最多为 0。这就能够保证树为完全二叉树。例如，



**注意：** 构建 HB(0)树，请参见 6.11 节中 AVL 树的相关问题。

## 6.11 AVL 树

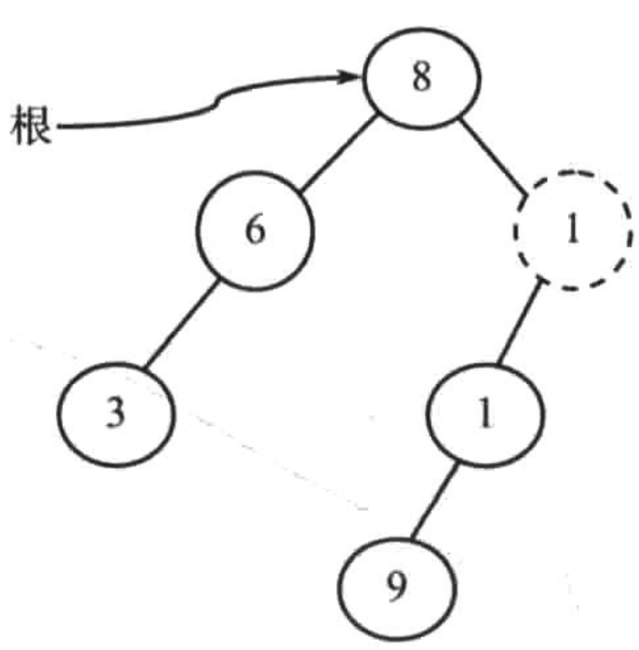
在 HB($k$)中，如果 $k=1$(如果平衡因子为 1)，那么这样的二叉搜索树叫作 AVL(Adelson-Velskii and Landis)树。即一棵 AVL 树是带有平衡条件的二叉搜索树：左子树和右子树的高度差最多不能超过 1。

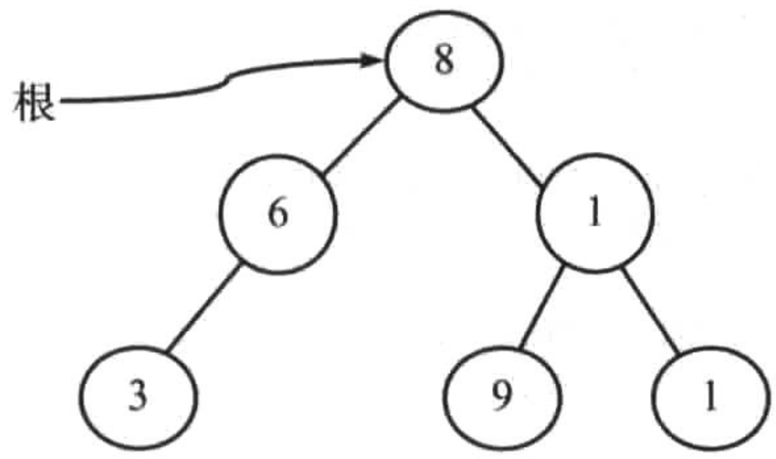
### 1. AVL 树的性质

一棵二叉树为 AVL 树，当且仅当满足如下条件：

- 它是一棵二叉搜索树。
- 对任意结点 $X$，其左子树的高度与其右子树的高度的差最多不超过 1。

例如，在下面的二叉搜索树中，左边的树不是 AVL 树，而右边的二叉查找树为 AVL 树。




### 2. AVL 树的最小/最大结点数

为了简单起见，假定 AVL 树的高度是 $h$，$N(h)$表示高度为 $h$ 的 AVL 树的结点数。

为了得到高度为 $h$ 的 AVL 树的最小结点数，应该尽可能用最少的结点数来填充这棵树。即假定填充左子树的高度为 $h-1$，那么右子树的高度只填充到 $h-2$。这样，高度为 $h$ 的 AVL 树的最小结点数为：$N(h)=N(h-1)+N(h-2)+1$。

在上面的公式中：

- $N(h-1)$表示高度为 $h-1$ 的左子树的最小结点数。
- $N(h-2)$表示高度为 $h-2$ 的右子树的最小结点数。
- 在上面的表达式中，“1”表示当前结点。

可以将 $N(h-1)$赋予左子树或右子树。求解上述递归式可得：

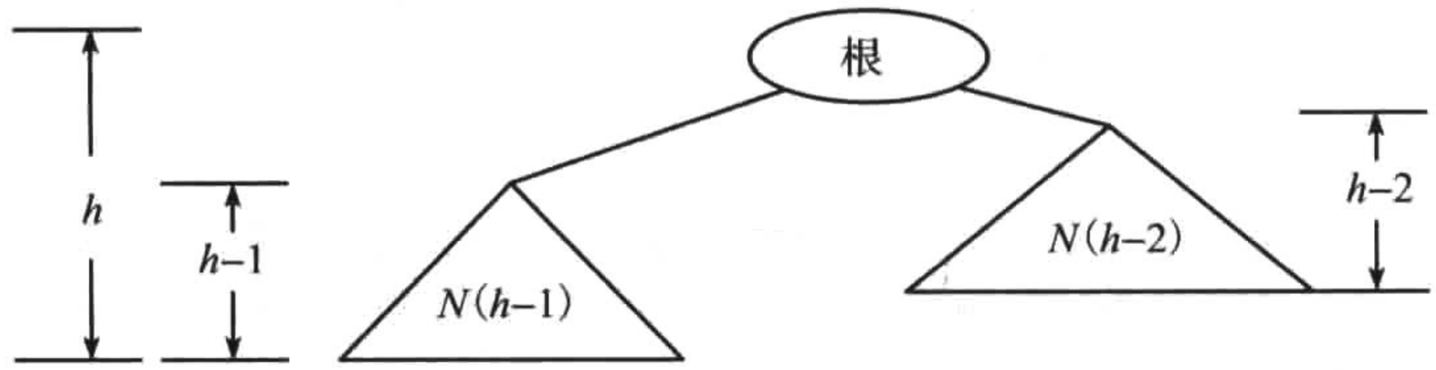
$$N(h) = O(1.618^h) \Rightarrow h = 1.44\log n \approx O(\log n)$$

其中 $n$ 是 AVL 树的结点数。上述推导也表明 AVL 树的最大高度为 $O(\log n)$。

类似地，为获取最大结点数，需要将左子树和右子树都填充到高度为 $h-1$。这样就有：$N(h)=N(h-1)+N(h-1)+1=2N(h-1)+1$。上面的表达式定义了完全二叉树的情况。求解该递归式可以得到：

$$N(h) = O(2^h) \Rightarrow h = \log n \approx O(\log n)$$

所以在这两种情况下，AVL 树的性质可以确保带有 $n$ 个结点的 AVL 树的高度为 $O(\log n)$。




### 3. AVL 树的定义

因为 AVL 树是一棵 BST 树，所以 AVL 树的定义类似于 BST 树的定义。但为了简化操作，将高度也作为定义中的一部分。

```
public class AVLTreeNode {
        private int data;
        private int height;
        private AVLTreeNode left;
        private AVLTreeNode right;
        public int getData() {
                return data;
        }
        public void setData(int data) {
                this.data = data;
        }
        public int getHeight() {
                return height;
        }
        public void setHeight(int height) {
                this.height = height;
        }
        public AVLTreeNode getLeft() {
                return left;
        }
        public void setLeft(AVLTreeNode left) {
                this.left = left;
        }
        public AVLTreeNode getRight() {
                return right;
        }
        public void setRight(AVLTreeNode right) {
                this.right = right;
        }
}
```

### 4. 求 AVL 树的高度

```
int Height(AVLTreeNode root ) {
        if( root == null)
            return -1;
        else
            return root.getHeight();
}
```

时间复杂度为 $O(1)$。

### 5. 旋转

当树结构发生变化时(例如，插入或删除结点)，就需要改变树的结构来保证 AVL 树的特性。这个操作可以用单旋转或双旋转来实现。因为插入/删除包括增加/删除一个单

结点，这将导致子树的高度增加 1 或减少 1。因此，如果 AVL 树的性质在结点 X 遭到破坏，则表明 $X$ 的左子树高度与右子树高度的差为 2。这是因为，每次平衡 AVL 树时，在每一个结点，其左右子树的高度差不能大于 1。旋转是用来保持 AVL 树性质的技术。即，需要在结点 $X$ 应用旋转操作。

**观察：**其中一个重要现象是，在插入结点后，只有从插入结点到根的路径上的结点的平衡因子才可能改变，因为只有这些结点的子树可能改变了。为恢复 AVL 树的性质，从插入点开始访问直到树的根结点。在移动到根结点的过程中，需要寻找第一个不满足 AVL 树性质的结点。从这个结点到根结点的路径上的每个结点都存在这个问题。

因此，如果修复第一个结点的问题，那么在通往根结点路径上的其他所有结点都将自动满足 AVL 树性质。这就意味着，总是关注从插入点到根结点路径上的第一个不满足 AVL 性质的结点，并修复它。

### 6. 违背 AVL 树性质的类型

假定结点 $X$ 是必须要重新平衡的结点。因为任意结点最多有两个孩子结点，并且高度不平衡需要 $X$ 结点的两棵子树的高度差为 2，所以容易观测出以下 4 种情况可能产生违反 AVL 树性质：

1）在结点 $X$ 的左孩子结点的左子树中插入元素。

2）在结点 $X$ 的左孩子结点的右子树中插入元素。

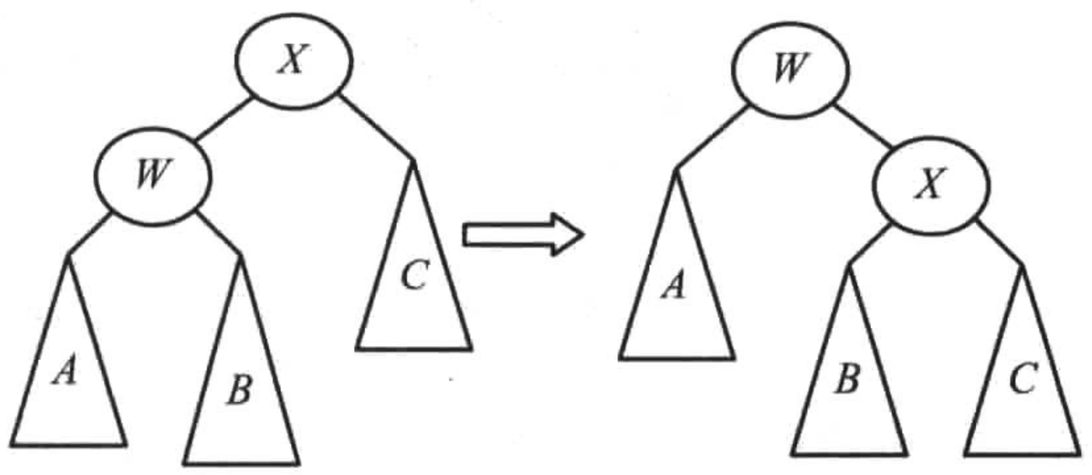
3）在结点 $X$ 的右孩子结点的左子树中插入元素。

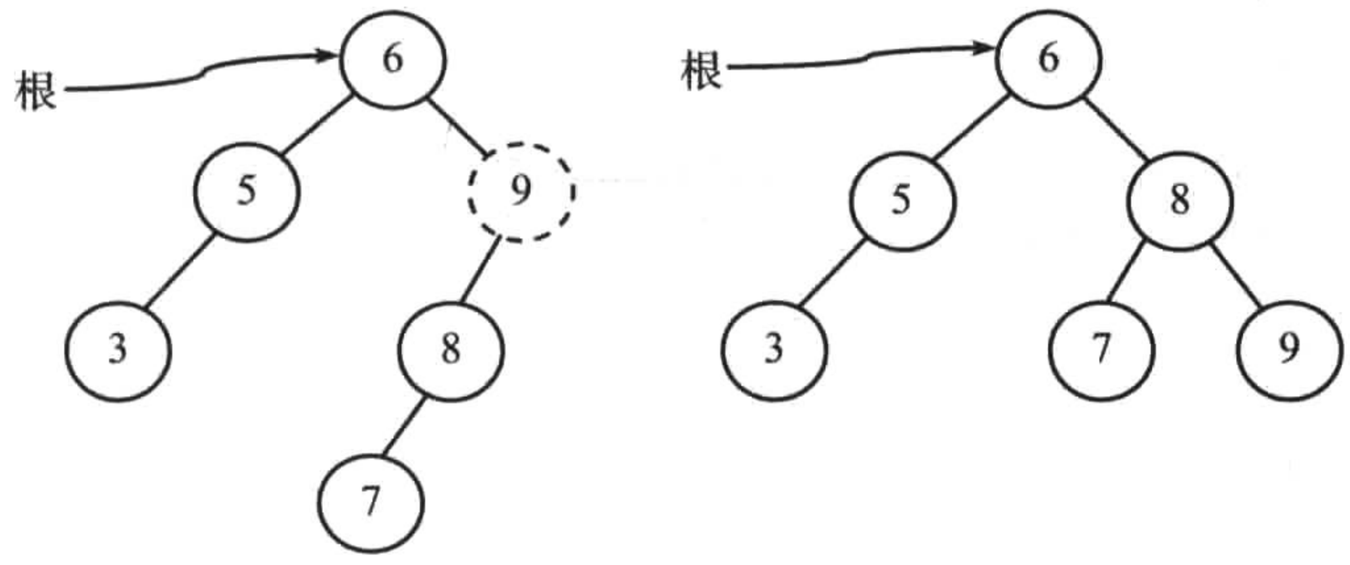
4）在结点 $X$ 的右孩子结点的右子树中插入元素。

第 1 种情况和第 4 种情况是对称的，很容易由单旋转来解决。类似地，第 2 种情况和第 3 种情况也是对称的，需要双旋转才能解决(需要两个单旋转)。

### 7. 单旋转

**左左旋转(LL 旋转)(情况 1)：**在下面情况中，在结点 $X$，AVL 树性质不满足。由上述讨论可知，不必在树的根结点旋转。通常，从插入结点处开始，向上遍历树，并更新在这个路径上的每个结点的平衡信息。例如，在下图中，在左边原始 AVL 树插入 7 后，结点 9 变为非平衡的。因此，在 9 处执行左左单旋转，结果得到右边这棵树。

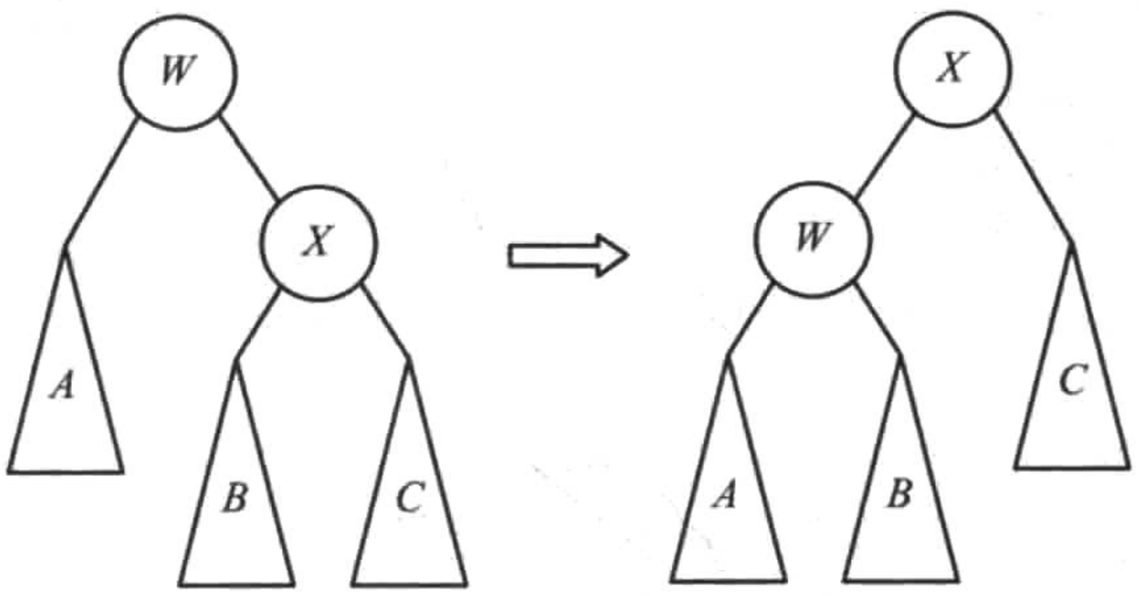


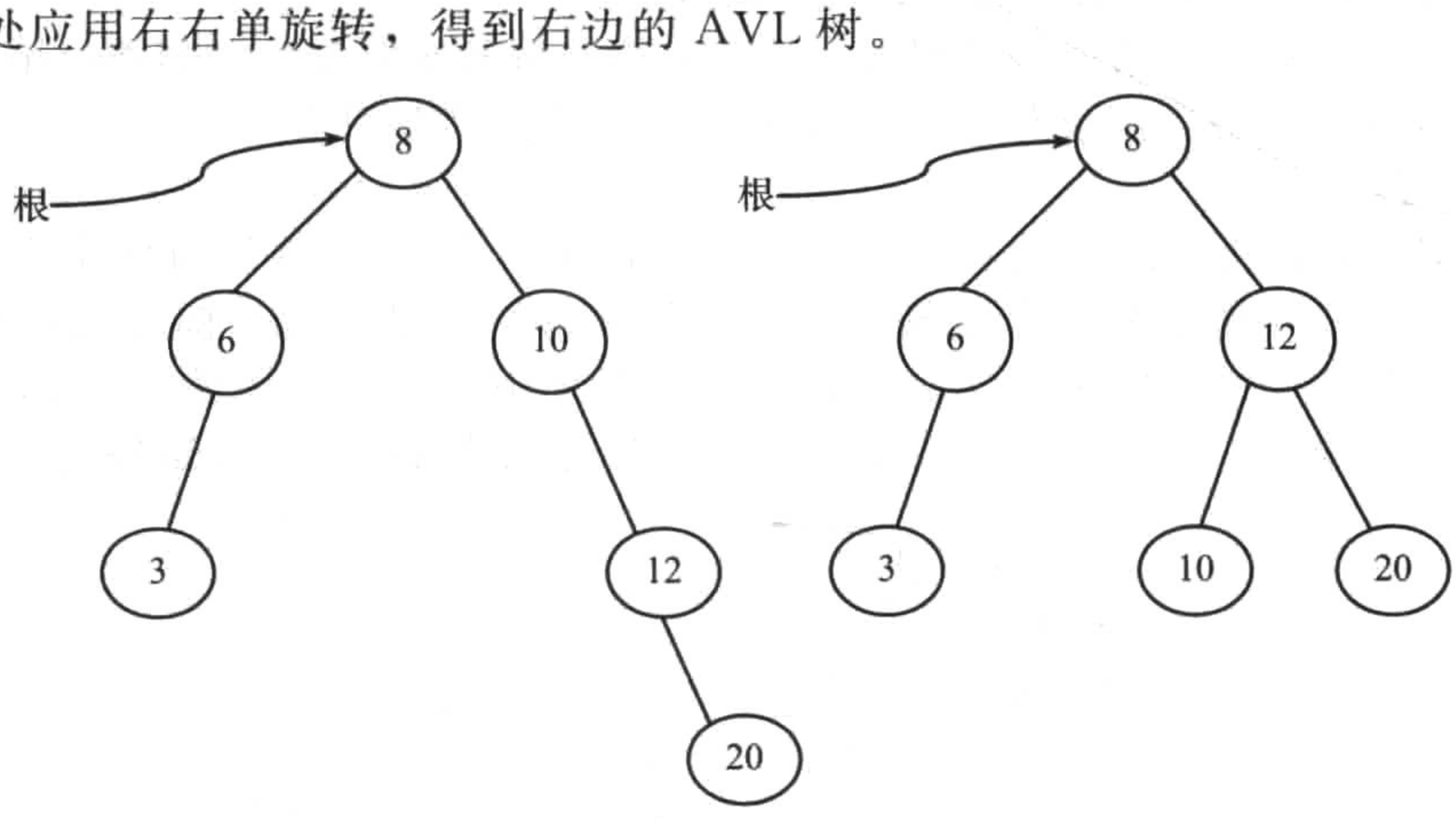
```
AVLTreeNode SingleRotateLeft(AVLTreeNode X ) {
        AVLTreeNode W = X.getLeft();

        X.setLeft(W.getRight());
        W.setRight(X);

        X.setHeight(Math.max( Height(X→left), Height(X.getRight()) ) + 1);
        W.setHeight(Math.max( Height(W→left), X→height ) + 1);

        return W;        /*新的根结点*/
}
```

时间复杂度为 $O(1)$。空间复杂度为 $O(1)$。

**右右旋转(RR 旋转)(情况 4)**：在这种情况下，结点 $X$ 不满足 AVL 树性质。




例如，下图中在左边原始 AVL 树中插入结点 20 后，结点 10 就不平衡了。因此需要在结点 10 处应用右右单旋转，得到右边的 AVL 树。

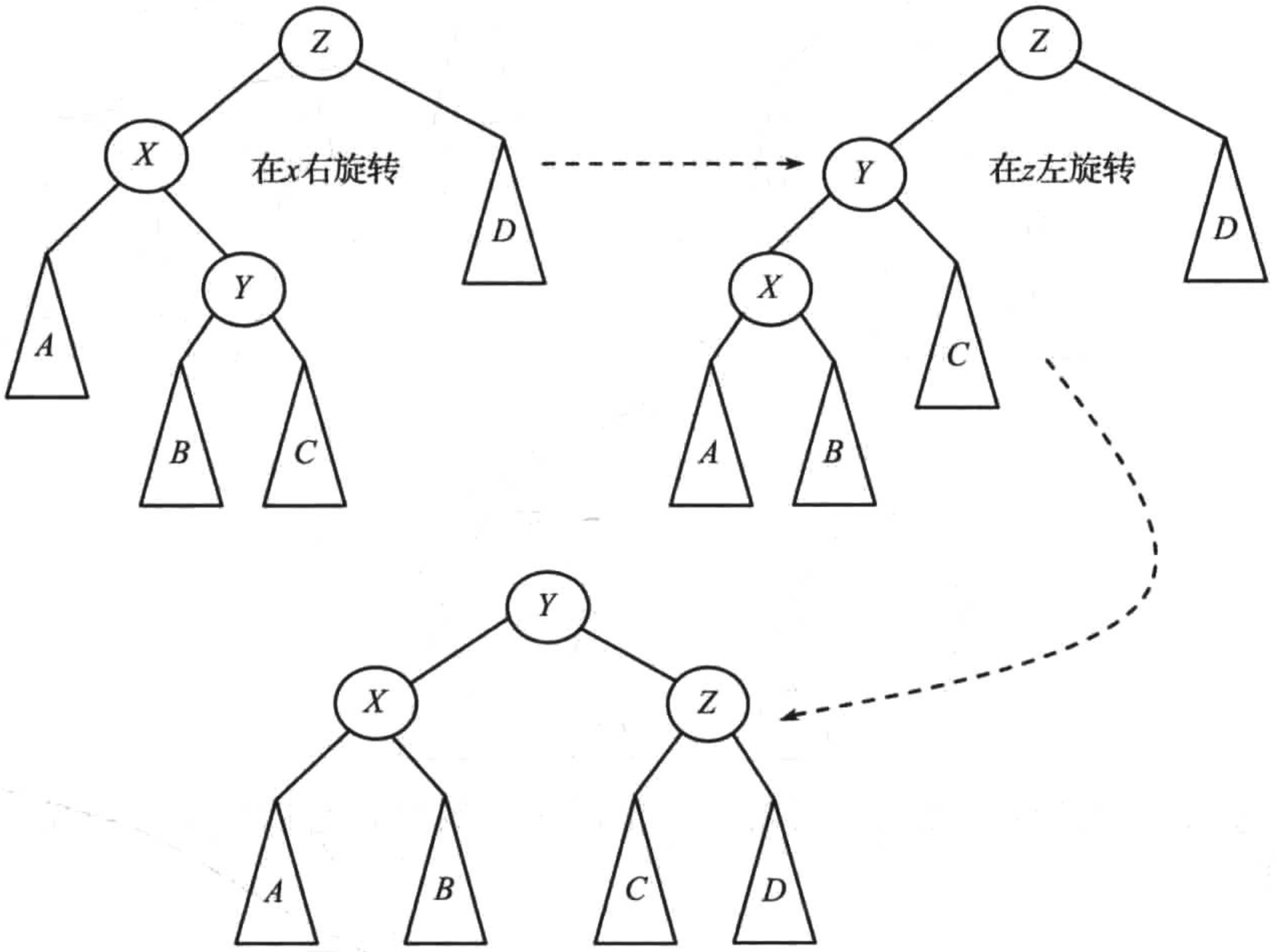


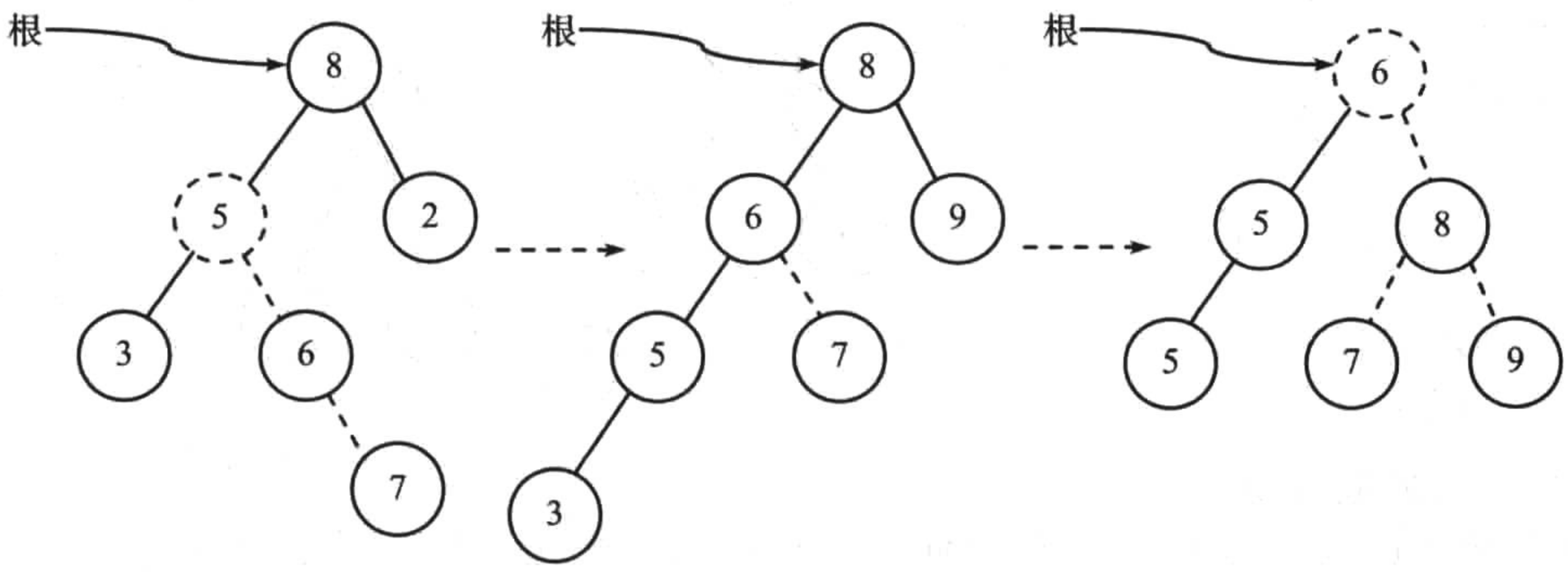

```
AVLTreeNode SingleRotateRight(AVLTreeNode W ) {
        AVLTreeNode X  = W.getRight();

        W.setRight(X.getLeft());
          X.setLeft(W);
        W.setHeight(Math.max( Height(W.getRight()), Height(W.getLeft()) ) + 1);
        X.setHeight(Math.max( Height(X.getRight()), W→height) + 1);

        return X;
}
```

时间复杂度为 $O(1)$，空间复杂度为 $O(1)$。

### 8. 双旋转

**左右旋转(LR 旋转)(情况 2)**：对于情况 2 和情况 3，单旋转不能解决问题。需要执行两次旋转才能解决。

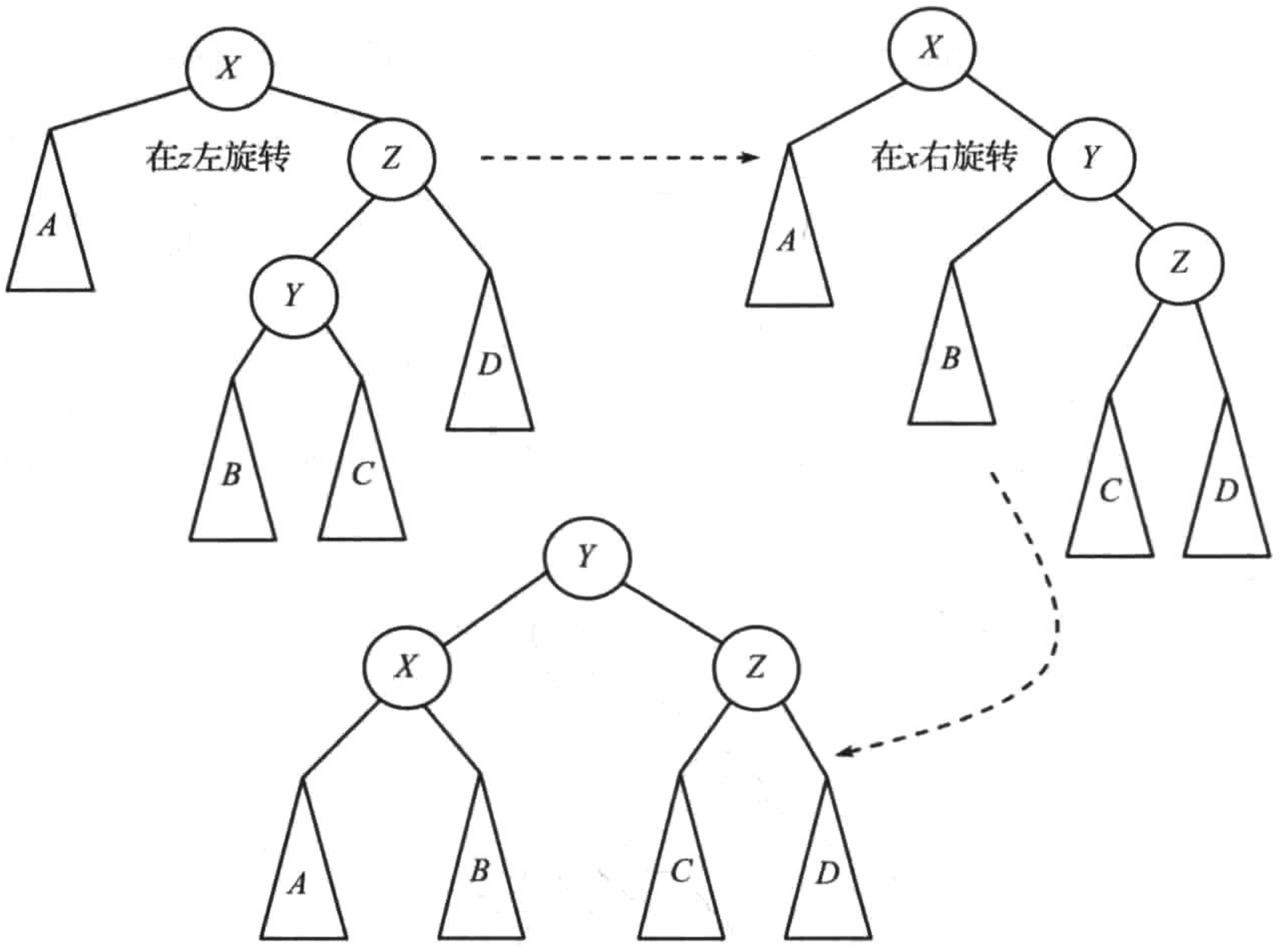


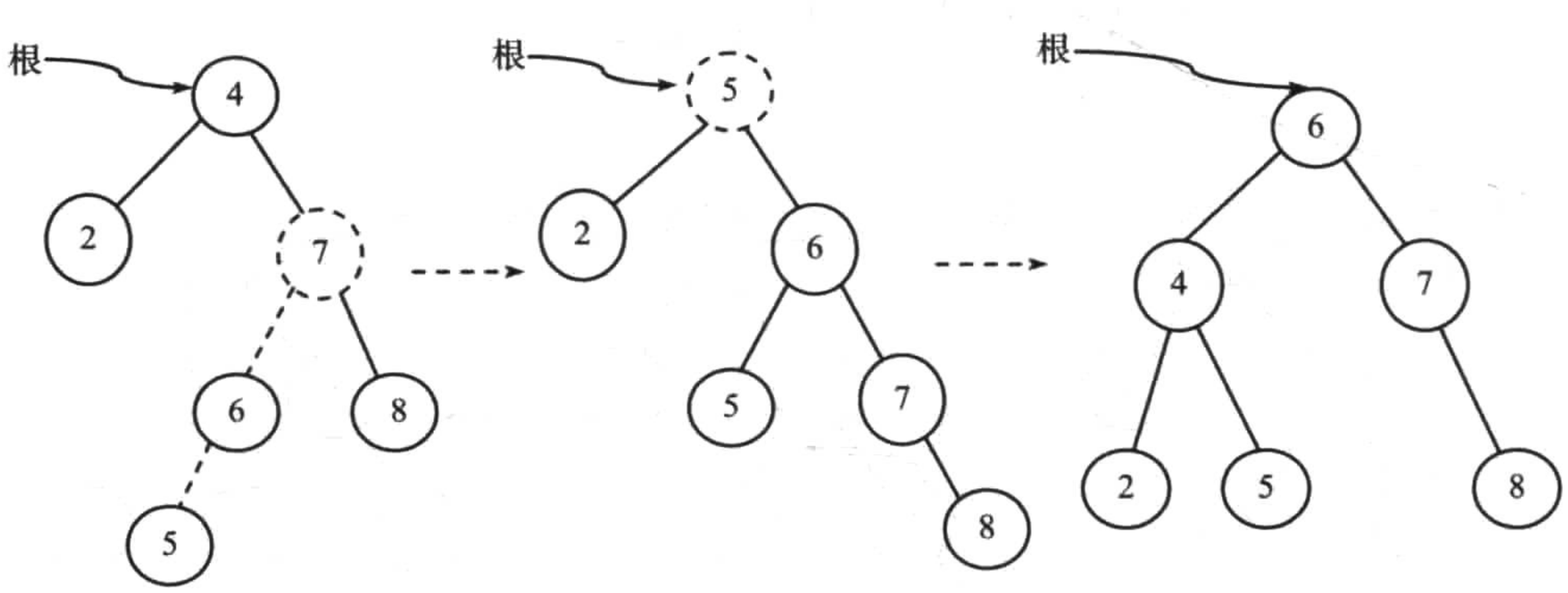

例如，考虑上面的树，在插入 7 后，就出现情况 2 的场景。右边的树就是在双旋转后形成的 AVL 树。

```
AVLTreeNode DoubleRotatewithLeft( AVLTreeNode Z ) {
        Z.setLeft( SingleRotateRight(Z.getLeft()) );
        return SingleRotateLeft(Z);
}
```

**右左旋转(RL 旋转)(情况 3)**：类似于情况 2，需要执行两次旋转才能解决这个问题。




例如，考虑下面的树，插入 6 后就产生情况 3 的场景，右边的树就是通过双旋转后得到的 AVL 树。




### 9. AVL 树的插入操作

AVL 树的插入类似于 BST 树的插入。在插入一个元素后，需要检查高度是否平衡。如果不平衡，需要调用相应的旋转函数。

```
AVLTreeNode Insert( AVLTreeNode root, AVLTreeNode parent, int data) {
        if( root == null) {
                root = new AVLTreeNode();
                root.setData(data);
                root.setHeight(0);
                root.setLeft(null);
                root.setRight(null);
        }
        else  if( data < root.getData() ) {
                root.setLeft(Insert( root.getLeft(), root, data ));
                if( ( Height( root.getLeft() ) - Height( root.getRight() ) ) == 2 ) {
```

```
                if( data < root.getLeft().getData() )
                    root = SingleRotateLeft( root );
                else
                    root = DoubleRotateLeft( root );
            }
}
        else  if( data > root.getData() ) {
                root.setRight(Insert( root.getRight(), root, data ) );
                if( ( Height( root.getRight() ) - Height( root.getLeft() ) ) == 2 ) {
                        if( data < root.getRight().getData() )
                            root = SingleRotateRight( root );
                        else
                            root = DoubleRotateRight( root );
                    }
            }
            /*否则，数据已经在树中，程序什么也不做。*/
            root.setHeight(Math.max( Height(root.getLeft()), Height(root.getRight()) ) + 1);
            return root;
    }
```

时间复杂度为 $O(n)$，空间复杂度为 $O(\log n)$。

### 10. AVL 树的相关问题

**问题 70**　给定树的高度 $h$，设计算法产生 HB(0)。

**解答：**由上述讨论可知，HB(0)就是一棵完全二叉树。高度为 $h$ 的完全二叉树所具有的结点数为：$2^{h+1}-1$(假设只有一个结点的树的高度为 0)。由此，所有结点可以编号为：$1\sim 2^{h+1}-1$。

```
BinarySearchTreeNode BuildHB0(int h) {
        BinarySearchTreeNode temp;
        if(h == 0)
         return null;
        temp = newBinarySearchTreeNode();
        temp.setLeft(BuildHB0 (h-1));
        temp.getData(count++); // 假设count是一个全局变量
        temp.setRight(BuildHB0 (h-1));
        return temp;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(\log n)$，其中 $\log n$ 表示最大的栈大小，且等于该树的高度。

**问题 71**　是否还有其他的方法来求解问题 70?

**解答：有**。可以使用归并排序的思想。即利用范围而不是高度来解决问题。这样就不需要维护一个全局计数器。

```
Struct BinarySearchTreeNode BuildHB0(int l, int r) {
        BinarySearchTreeNode temp;
        int mid =l + (r-l)/2;
        if( l > r)
            return null;
        temp = (BinarySearchTreeNode ) malloc (sizeof(BinarySearchTreeNode));
        temp.setData(mid);
        temp.setLeft(BuildHB0(l, mid-1));
        temp.setRight(BuildHB0(mid+1, r));
        return temp;
}
```

BuildHB0 函数的初始调用是：BuildHB0(1，1<<$h$)。1<<$h$ 执行左移操作来计算 $2^{h+1}-1$。

时间复杂度为 $O(n)$。空间复杂度为 $O(\log n)$。其中 $\log n$ 表示栈的最大空间大小，其值等于树的高度。

**问题 72** 构建高度为 0、1、2、3、4 和 5 的最小 AVL 树。高度为 6 的最小 AVL 树中的结点数是多少？

**解答：** 设 $N(h)$为高度为 $h$ 的最小 AVL 树的结点数。

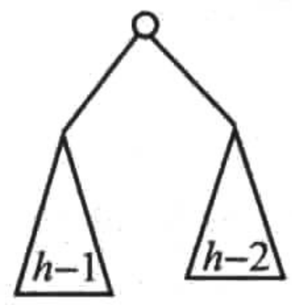
$N(0)=1$



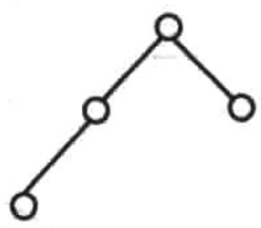
$N(1)=2$



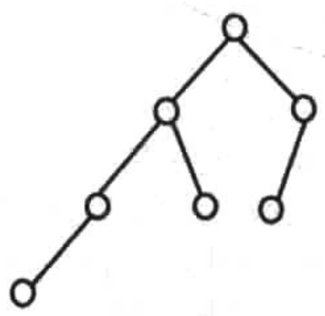
$N(h)=1+N(h-1)+N(h-2)$



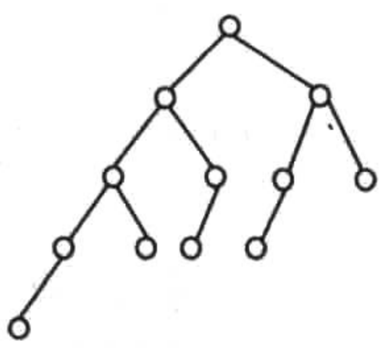

$N(2)=1+N(1)+N(0)$
$=1+2+1=4$



$N(3)=1+N(2)+N(1)$
$=1+4+2=7$



$N(4)=1+N(3)+N(2)$
$=1+7+4=12$

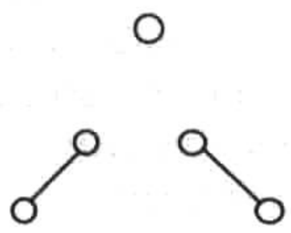


$N(5)=1+N(4)+N(3)$
$=1+12+7=20$

**问题 73** 对于问题 70，高度 $h$ 的最小 AVL 树有多少种不同的形态？

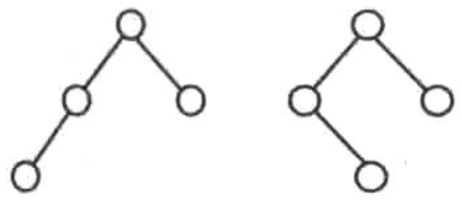
**解答：** 设 NS($h$)为高度 $h$ 的最小 AVL 树的不同形态数。

NS(0)=1

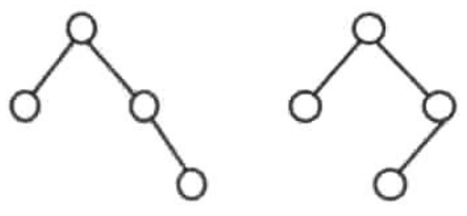


NS(1)=2

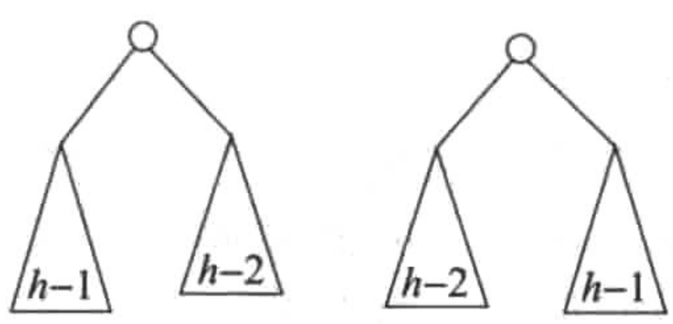
$$NS(2)=2*NS(1)*NS(0)$$
$$=2*2*1=4$$



$$NS(3)=2*NS(2)*NS(1)$$
$$=2*4*1=8$$



$$NS(h)=2*NS(h-1)*NS(h-2)$$




**问题 74** 给定一棵二叉搜索树，判断其是否为 AVL 树？

**解答：** 设函数 IsAVL 可以判断给定二叉搜索树是否为 AVL 树。如果不是 AVL 树，则返回－1。在判断过程中，每个结点将其高度传递给其双亲结点。

```
int IsAVL(BinarySearchTreeNode root) {
        int left, right;
        if(root == null)
            return 0;
        left = IsAVL(root.getLeft());
        if(left == -1)
                return left;
        right = IsAVL(root.getRight());
        if(right == -1)
                return right;
        if(Math.abs(left-right)>1)
                return -1;
        return Math.max(left, right)+1;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

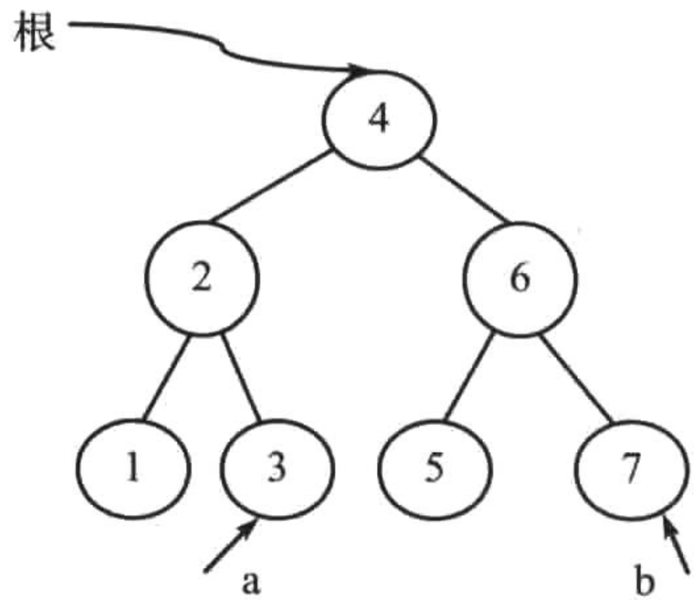
**问题 75** 给定树的高度 $h$，设计算法构建结点数最少的 AVL 树。

**解答：** 为得到最少结点数，一棵子树填充高度为 $h-1$，另一棵树填充高度为 $h-2$。

```
AVLTreeNode GenerateAVLTree(int h) {
        AVLTreeNode temp;
        if(h == 0)
                return null;
        temp = new AVLTreeNode();
        temp.setLeft(GenerateAVLTree(h-1));
        temp.getData(count++); //假设count为全局变量
        temp.setRight(GenerateAVLTree(h-2));
        temp.setHeight(temp.getLeft().getHeight()+1); //或者temp→height = h;
        return temp;
}
```

**问题 76** 给定包含 $n$ 个整数结点的 AVL 树及两个整数 $a$ 和 $b$，其中 $a$ 和 $b$ 可以为任意整数，且 $a \leqslant b$。给出算法，计算$[a, b]$中的结点个数。

**解答：**




算法思路是利用二叉搜索树的递归性质。有 3 种情况需要考虑：当前结点是否处于[$a$，$b$]中；在[$a$，$b$]的左侧；在[$a$，$b$]的右侧。只有可能包含此类结点的子树才会被这 3 种情况中的任意一种处理。

```
int RangeCount(AVLNode root, int a, int b) {
        if(root == null)
                return 0;
        else if(root.getData() > b)
                return RangeCount(curr.getLeft(), a, b);
        else if(root.getData() < a)
                return RangeCount(root.getRight(), a, b);
        else if(root.getData() >= a && root.getData() <= b)
                return RangeCount(root.getLeft(), a, b) + RangeCount(root.getRight(), a, b) + 1;
}
```

复杂度类似于中序遍历树，但跳过不包含解的左子树或右子树。因此，在最坏情况下，如果范围覆盖树的所有结点，则需要遍历所有 $n$ 个结点才能得到解。最坏情况下，时间复杂度为 $O(n)$。

如果范围很窄，只包含树底部某个小子树的少数几个元素，则时间复杂度将为 $O(h)=O(\log n)$，其中 $h$ 为树的高度。这是因为只需遍历一条简单路径就能到达底部的小子树，高层的许多子树都被剪掉了。

**注意：** 请参考 BST 中的类似问题。

**问题 77** 有限长整数序列求中位数问题。

**解答：** 中位数是有序数字表中的中间数(如果有序表中有奇数个元素)。如果有序表中有偶数个元素，则中位数就是求有序表中中间两个数的平均值。

为了解决这个问题，可以使用一棵特殊的二叉搜索树，该树中每个结点包含左子树结点数和右子树结点数的附加信息。同时也保存树的总结点数。通过使用附加信息，基于当前结点的左子树和右子树的孩子个数来选择树的合适分支，能够在 $O(\log n)$时间内找到中位数。但是，插入操作的复杂度为 $O(n)$，因为如果恰好数据是有序的，则一棵标准二叉搜索树就退化为链表。

因此，可用平衡二叉搜索树来避免标准二叉搜索树的最坏情况。对于此问题，平衡因子就是左子树结点的个数减去右子树结点的个数。并且只有平衡因子为 1 或 0 的结点才认为是平衡的。因此，左子树的结点数要么等于右子树的结点数，要么比其多 1，但不会少。

如果确保树中每个结点都具有这种平衡因子，那么当树的元素个数是奇数时，树的

根结点就是中位数。对于元素个数是偶数的情况，中位数是根结点的值和中序序列中其后继结点值的平均值，该后继结点是根结点的右子树中最左边的孙子结点。

因此，如果结点数为偶数，且假定在每次插入时均要计算根结点在中序序列的后继结点，那么保持平衡条件的插入操作的复杂度是 $O(\log n)$，获取中位数操作的复杂度为 $O(1)$。插入和平衡操作非常类似于 AVL 树，不过不是更新高度，而是更新结点的个数信息。平衡二叉搜索树似乎是最优的解决方案，插入的时间复杂度为 $O(\log n)$，找到中位数的时间复杂度是 $O(1)$。

**注意：** 对于高效算法，请参见第 7 章。

**问题 78** 给定一棵二叉树，如何删除所有的半结点(即只有一个孩子的结点)？注意：不能涉及叶子结点。

**解答：** 使用后序遍历能有效地解决这个问题。首先处理左孩子，然后是右孩子，最后是结点本身。因此算法自底向上，从叶子到根结点来形成一棵新树。当处理到当前结点时，其左子树和右子树都已经处理完了。

```
BinaryTreeNode removeHalfNodes(BinaryTreeNode root){
   if (root == null)
      return null;

   root.left=removeHalfNodes(root.getLeft());
   root.right=removeHalfNodes(root.getRight());

   if (root.getLeft() == null && root.getRight() == null)
      return root;

   if (root.getLeft() == null)
      return root.getRight();

   if (root.getRight() == null)
      return root.getLeft();

   return root;
}
```

时间复杂度为 $O(n)$。

**问题 79** 给定一棵二叉树，如何删除所有的叶子结点？

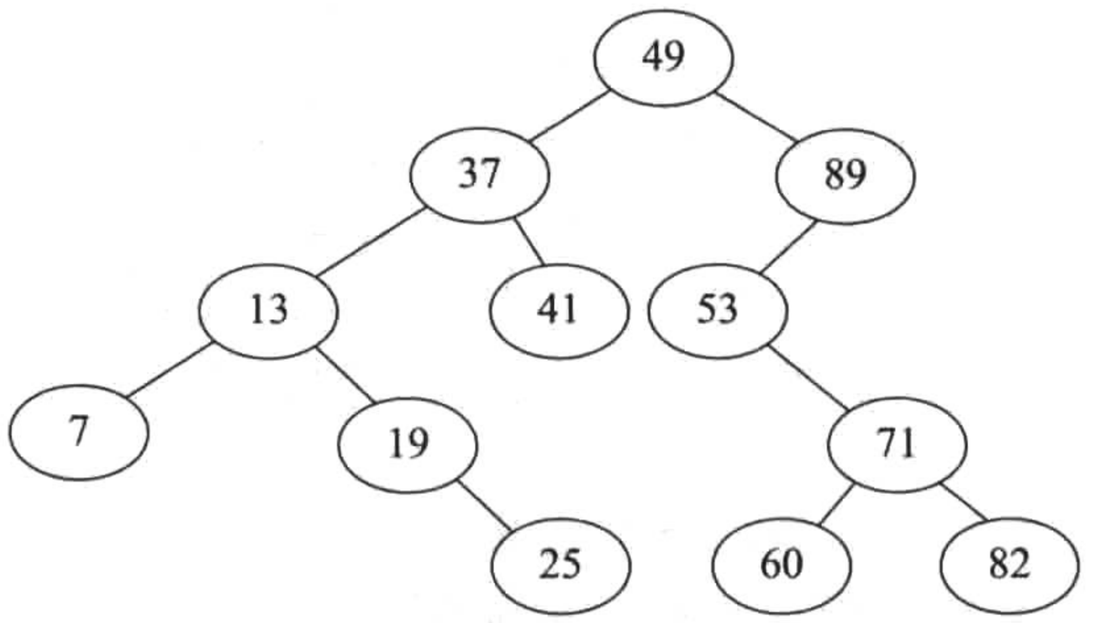
**解答：** 使用后序遍历能解决此问题(其他遍历也可以)。

```
BinaryTreeNode removeLeaves(BinaryTreeNode root) {
    if (root != null) {
        if (root.getLeft() == null && root.getRight() == null) {
            root = null;
        } else {
            root.left = removeLeaves(root.getLeft());
            root.right = removeLeaves(root.getRight());
        }
    }
    return root;
}
```

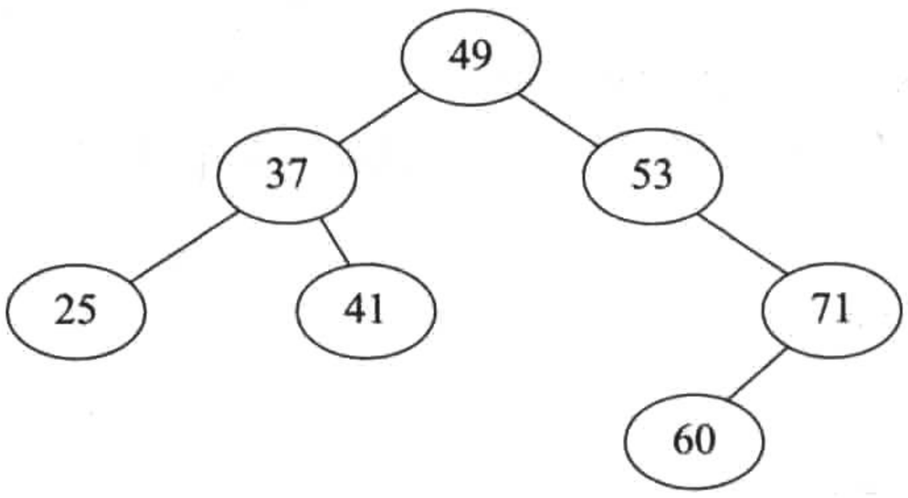
时间复杂度为 $O(n)$。

**问题 80** 给定一棵 BST 树和两个整数(最小整数和最大整数)作为参数，如何从树中删除所有不在该区间的元素(剪枝)。

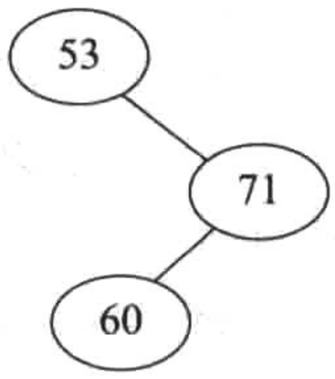
例子



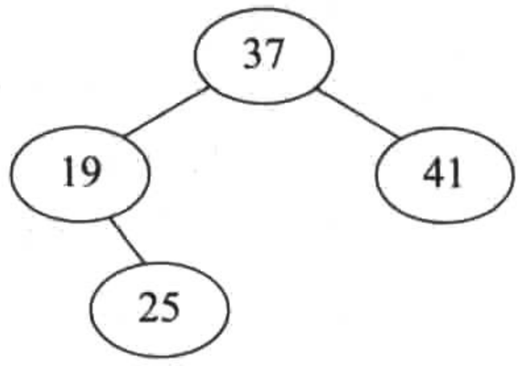

PruneBST(24,71);




PruneBST(53,79);




PruneBST(17,41);




**解答：观察：**由于需要检查树中每一个元素，并且子树的改变应该反映在双亲结点上，所以可以考虑使用后序遍历。因此，结点的处理顺序是从叶子结点到根结点。因此，当处理到结点本身时，其左子树和右子树为有效剪枝的 BST 树。

在每个结点，将基于其值返回一个指针，这将作为该结点双亲结点指向左孩子或右孩子的指针，取决于当前结点是其双亲结点的左孩子还是右孩子。如果当前结点的值为 $A \sim B(A \leqslant$结点数据$\leqslant B)$，那么不需要执行其他动作，只需要返回指向其自身的指针。

如果当前结点的值小于 $A$，那么返回指向其右子树的指针，并丢弃其左子树。因为结点的值小于 $A$，所以根据二叉搜索树性质，其左孩子小于 $A$。但是其右孩子可能小于 $A$，也可能不小于 $A$，这无法确定，因此返回指向它的指针。由于算法执行自底向上的后序遍历，所以其右子树已经是被修剪好的有效二叉搜索树(可能为空)，并且其左子树肯定为空，因为其所有结点都肯定小于 $A$，并且在后序遍历中被删除了。

当结点的值大于 $B$ 时，情况是类似的，只需要返回指向其左子树的指针，因为如果结点的值大于 $B$，那么其右孩子肯定大于 $B$。又因为其左孩子可能大于 $B$，也可能不大于 $B$，所以丢弃右子树，返回指向其有效左子树的指针。

```
BinarySearchTreeNode PruneBST(BinarySearchTreeNode root, int  A, int B){
    if(root == null)
        return null;
    root.left= PruneBST(root->getLeft(),A,B);
    root.right= PruneBST(root->getRight(),A,B);
    if(A<=root.getData() && root.getData()<=B)
        return root;
    if(root.getData()<A)
        return root.getRight();
    if(root.getData()>B)
        return root.getLeft();
}
```

时间复杂度：最坏情况下是$O(n)$，平均情况下是$O(\log n)$。

**注意：**如果给定的BST树是AVL树，那么平均时间复杂度是$O(n)$。

**问题81** 给定一棵二叉树，如何连接同一层的所有邻近点？假定给定的二叉树除了左指针和右指针外，还有一个下一个(next)指针。

**解答：**一个简单的方法是使用层次遍历，并不断更新下一个(next)指针。当遍历时，需要把结点链接到下一层。如果一个结点有左孩子结点和右孩子结点，将左孩子结点链连到右孩子结点。如果结点有下一个结点，那么将当前结点的最右孩子结点链接到下一个结点的最左孩子结点。

```
public void linkLevelNodes(BinaryTreeNode root){
    Queue Q = CreateQueue();
    BinaryTreeNode prev;     // 指向当前层的前一个结点的指针
    BinaryTreeNode temp;
   int currentLevelNodeCount;
   int nextLevelNodeCount;
    if(root == null)
       return;

    EnQueue(Q, root);
    currentLevelNodeCount = 1;
    nextLevelNodeCount = 0;
    prev = NULL;

    while (!IsEmptyQueue(Q)) {
       temp = DeQueue(Q);

       if (temp.left != null){
       EnQueue(Q, temp.left);
       nextLevelNodeCount++;
   }
   if (temp.right != null){
       EnQueue(Q, temp.right);
       nextLevelNodeCount++;
   }

   //将当前层的前一个结点链接到该结点
   if (prev)
       prev.next = temp;

   //Set the previous node to the current
   prev = temp;

   currentLevelNodeCount--;
   if (currentLevelNodeCount == 0) {   //如果该结点是当前层的最后一个结点
       currentLevelNodeCount = nextLevelNodeCount;
       nextLevelNodeCount = 0;
       prev = NULL;
       }
    }
}
```

时间复杂度为$O(n)$，空间复杂度为$O(n)$。

**问题82** 能否降低问题81的空间复杂度？

**解答：**可以在不使用队列的情况下逐层处理树。其思路是当处理下一层的结点时，确保当前层已经链接完成。

```
public void linkLevelNodes(BinaryTreeNode root) {
   if(root==null)
```

```
        return;
    BinaryTreeNode rightMostNode = null;
    BinaryTreeNode nextHead = null;
    BinaryTreeNode temp = root;
    // 将当前根结点的下一层链接到该层
    while(temp!=null){
        if(temp.left!=null)
            if(rightMostNode==null){
                rightMostNode=temp.left;
                nextHead=temp.left;
            }
            else{
                rightMostNode.next = temp.left;
                rightMostNode = rightMostNode.next;
            }
        if(temp.right!=null)
            if(rightMostNode==null){
                rightMostNode=temp.right;
                nextHead=temp.right;
            }
            else{
                rightMostNode.next = temp.right;
                rightMostNode = rightMostNode.next;
            }
        temp=temp.next;
    }
    connect(nextHead);
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(h)$，$h$ 为树的高度，用于栈开销。

## 6.12 树的其他形式

本节将举例说明树的其他形式。从前面章节中可以知道，AVL 树就是具有平衡性质的 BST 树。现在讨论更多类型的平衡二叉搜索树：红黑树和伸展树。

### 6.12.1 红黑树

在红黑树中，每个结点关联一个额外的属性：红色或黑色中的一种颜色。为了得到对数级的复杂度，红黑树具有以下限制条件。

**定义**：红黑树是一棵满足以下性质的二叉搜索树：

- 根性质：根结点是黑色的。
- 外部性质：每个叶子结点是黑色的。
- 内部性质：红色结点的孩子结点是黑色的。
- 深度性质：所有叶子结点都同是黑色。

类似于 AVL 树，如果红黑树变为非平衡树，那么需要执行旋转来使其重新达到平衡。利用红黑树，在最坏情况下能够在 $O(\log n)$时间内完成如下操作，其中 $n$ 为树的结点数。

- 插入
- 删除
- 查找前驱
- 查找后继
- 查找最小值

- 查找最大值

### 6.12.2 伸展树

伸展树是具有自我调整性质的 BST 树。它的另一个有趣性质是：从空树开始，具有最大 $n$ 个结点的任意 $K$ 个操作序列在最坏情况下所需的时间复杂度为 $O(K\log n)$。

伸展树易于编程实现，并且能够快速地访问最近访问的元素。类似于 AVL 和红黑树，任何时刻伸展树出现不平衡，都需要旋转来保证平衡性质。

伸展树不能保证最坏情况下 $O(\log n)$的复杂度。但提供平均 $O(\log n)$的复杂度。即使单个操作相当耗时，但整个操作序列具有对数级的复杂度。一个操作可能花费更多时间(比如，$O(n)$的时间)，但是后续操作可能不会达到最坏情况的复杂度，因而，每个操作的平均复杂度为 $O(\log n)$。

### 6.12.3 增强树

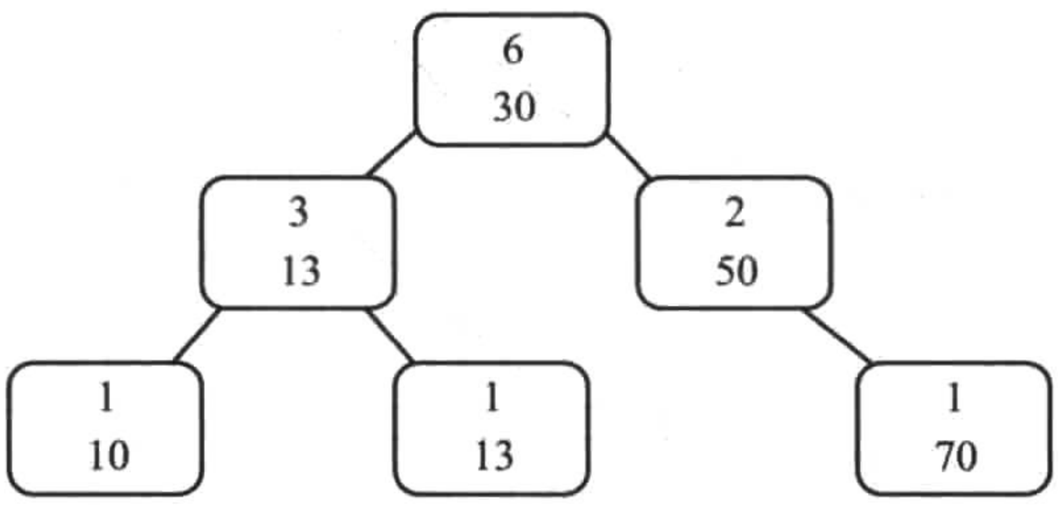
在前面的章节中，已经讨论了查找树中第 $K$ 小元素和其他许多类似问题。对于这些问题，最坏情况下的复杂度为 $O(n)$，其中 $n$ 是树的结点数。如果需要在 $O(\log n)$时间内完成这些操作，则增强树是十分有用的。在增强树中，每个结点都包含附加的信息，具体存储的信息则取决于待求解的问题。

例如，用增强树求解二叉搜索树第 $K$ 小元素问题，假设使用红黑树作为平衡 BST(或者其他平衡 BST)，并且每个结点包含数据的大小(size)信息。

对于红黑树中的结点 $X$，字段 size($X$)等于子树的结点数，按下式计算：

$$\text{size}(X) = \text{size}(X \rightarrow \text{left}) + \text{size}(X \rightarrow \text{right}) + 1$$

**例子：**根据附加的大小(size)信息，增强树如下所示：




查找第 $K$ 小元素的操作定义为：

```
BinarySearcTreeNode KthSmallest (BinarySearcTreeNode X, int K) {
        int r = size(X.getLeft()) + 1;
        if(K == r)
                return X;
        if(K < r)
                return KthSmallest (X.getLeft(), K);
        if(K > r)
                return KthSmallest (X.getRight(), K-r);
}
```

时间复杂度为 $O(\log n)$，空间复杂度为 $O(\log n)$。

### 6.12.4 替罪羊树

替罪羊树是一棵自平衡二叉搜索树，由 Arne Andersson 提出。它提供了最坏情况下

$O(\log n)$的查找时间，平均 $O(\log n)$的插入和删除操作时间。

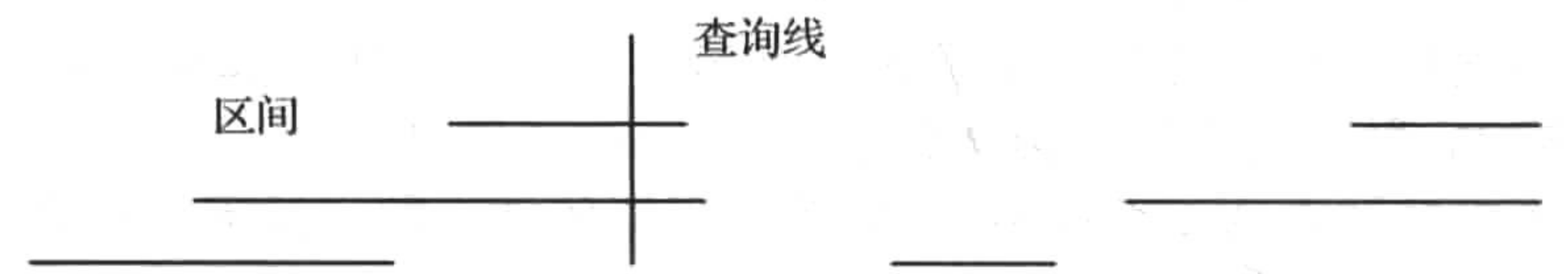
当两个兄弟子树的高度差大于 1 时，AVL 树需要进行再平衡。当一个孩子的大小(size)超过了双亲结点的一定比例时替罪羊树进行再平衡，该比例用 $\alpha$ 表示。插入一个结点后，反向遍历树，如果孩子的大小(size)超过了双亲结点的大小(size)乘以 $\alpha$，则认为树不平衡，这时从该双亲结点(替罪羊)重新构建子树。

有可能存在多个替罪羊，但仅需挑选其中一个。最优的替罪羊实际上是由高度平衡来决定的。当删除时，如果树的总大小(size)小于上次重建后的最大大小(size)的 $\alpha$ 倍，则重新构建整棵树。替罪羊树的 $\alpha$ 可以取 0.5～1 之间的任何值。0.5 将导致最佳平衡，而 1 将导致从不发生再平衡，等效于一个 BST。

### 6.12.5 区间树

区间树也是二叉搜索树，它在结点中存储区间信息。即维护 $n$ 个区间$[i_1, i_2]$的集合，使得可以有效查询其中一个集合包含某个查询点 $Q$。区间树常用于高效的范围查询操作。

**例子：**给定一组区间：$S=\{[2\text{-}5], [6\text{-}7], [6\text{-}10], [8\text{-}9], [12\text{-}15], [15\text{-}23], [25\text{-}30]\}$。查询 $Q=9$ 就返回[6，10]或[8，9](假设这些是所有区间中包含 9 的区间)。查询 $Q=23$ 就返回[15，23]。



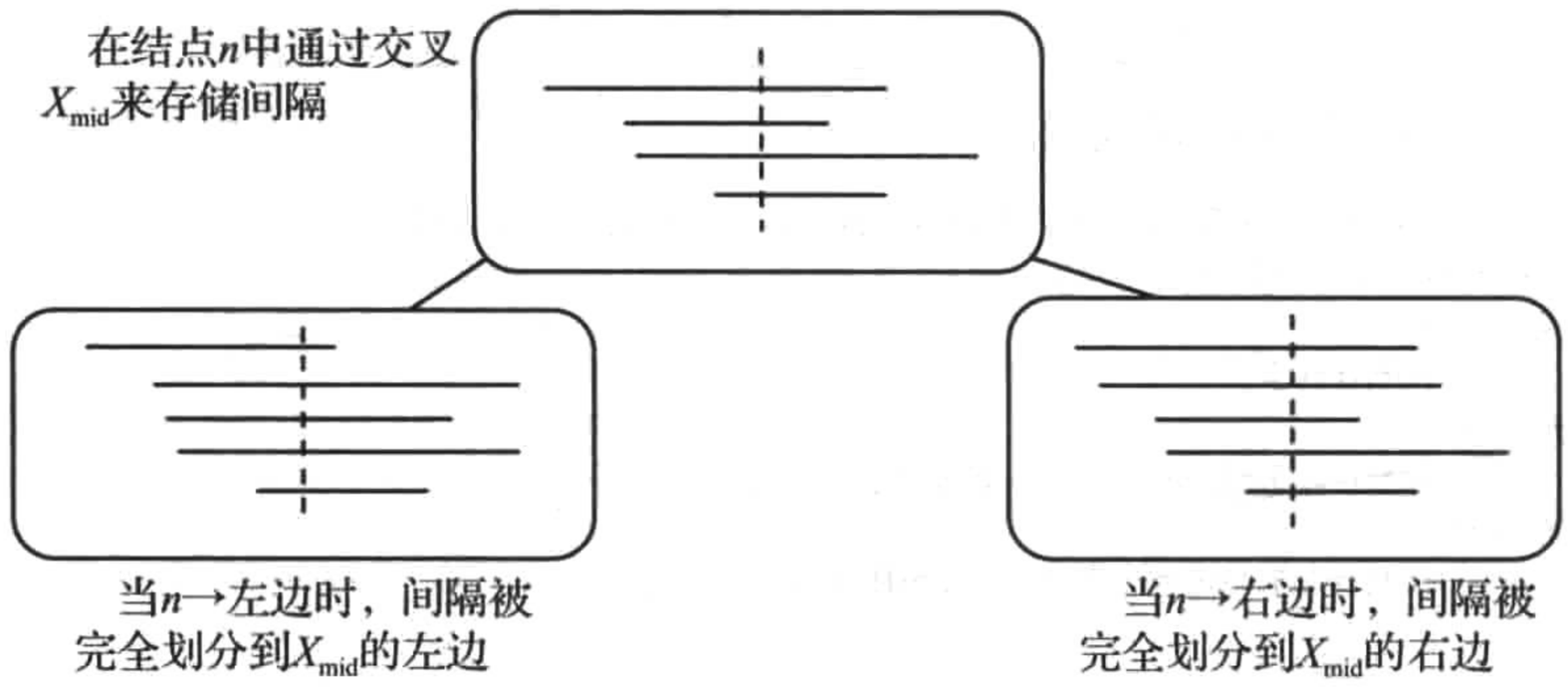

**构建区间树**

假设有一个包含 $n$ 个区间(也叫作段)的集合 $S$。这 $n$ 个区间将有 $2n$ 个端点。现在考虑如何构建区间树。

**算法：**

在区间集合 $S$ 上递归地构建树，步骤如下：

- 将 $2n$ 个端点排序。
- 设 $X_{min}$为中位数点。




构建区间树的时间复杂度为：$O(n\log n)$。由于选择了中位数，所以区间树是近似平衡的。这保证每次能够平分端点集合。树的深度为 $O(\log n)$。为了简化查找过程，通常 $X_{min}$存储于每个结点中。

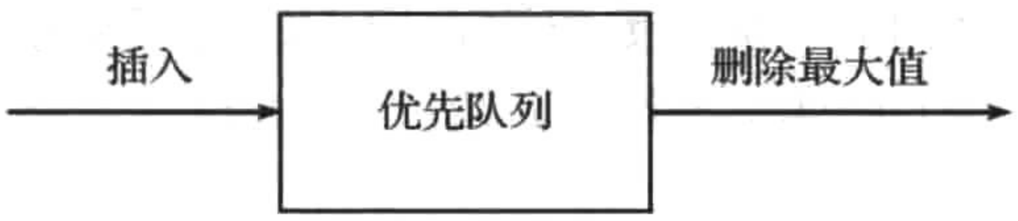
第7章 Chapter 7

# 优先队列和堆

## 7.1 什么是优先队列

在有些情况下，可能需要找到元素集合中的最小或最大元素。可以利用优先队列ADT来完成该操作。优先队列ADT是一种数据结构，它支持插入(Insert)和删除最小值(DeleteMin)操作(返回并删除最小元素)或删除最大值(DeleteMax)操作(返回并删除最大元素)。

这些操作等价于队列的EnQueue和DnQueue操作。区别在于，对于优先队列，元素进入队列的顺序可能与其被操作的顺序不同。作业调度是优先队列的一个应用实例，它根据优先级高低而不是先到先服务的方式来进行调度。




如果最小键值元素拥有最高的优先级，那么这种优先队列叫作升序优先队列(即总是先删除最小的元素)。类似地，如果最大键值元素拥有最高的优先级，那么这种优先队列叫作降序优先队列(即总是先删除最大的元素)。由于这两种类型是对称的，所以只需要关注其中一种，如升序优先队列。

## 7.2 优先队列ADT

下面操作组成了优先队列的一个ADT。

### 1. 优先队列的主要操作

优先队列是元素的容器，每个元素有一个相关键值。

- Insert(key，data)：插入键值为 key 的数据到优先队列中，元素以其 key 进行排序。
- DeleteMin/DeleteMax：删除并返回最小/最大键值的元素。
- GetMinimum/GetMaximum：返回最小/最大键值的元素，但不删除它。

**2. 优先队列的辅助操作**

- 第 $k$ 最小/第 $k$ 最大：返回优先队列中键值为第 $k$ 个最小/最大的元素。
- 大小(Size)：返回优先队列中的元素个数。
- 堆排序(Heap Sort)：基于键值的优先级将优先队列中的元素进行排序。

## 7.3 优先队列的应用

优先队列有许多应用，如下列举了部分应用：

- 数据压缩：赫夫曼编码算法。
- 最短路径算法：Dijkstra 算法。
- 最小生成树算法：Prim 算法。
- 事件驱动仿真：顾客排队算法。
- 选择问题：查找第 $k$ 个最小元素。

## 7.4 优先队列的实现

在讨论实际的实现方式前，首先列出其可能的实现方式。

**1. 无序数组实现**

将元素插入数组中，不考虑元素的顺序，则插入的时间复杂度为 $O(1)$。删除(DeleteMin)操作需要首先找到相应的键值，然后进行删除，DeleteMin 的时间复杂度为 $O(n)$。

**2. 无序链表实现**

非常类似于数组实现，只是用链表代替了数组。插入的时间复杂度为 $O(1)$。DeleteMin 的时间复杂度为 $O(n)$。

**3. 有序数组实现**

元素按照基于键值排序的方式插入数组中，插入的时间复杂度为 $O(n)$。删除元素操作只在数组的一端执行，DeleteMin 的时间复杂度为 $O(1)$。

**4. 有序链表实现**

元素按照基于键值排序的方式插入链表中。删除元素操作只在数组的一端执行，因此能够保证优先队列的状态。所有与链表 ADT 关联的其他功能都可以在无需修改的情况下执行。插入的时间复杂度为 $O(n)$。DeleteMin 的时间复杂度为为 $O(1)$。

**5. 二叉搜索树实现**

若随机插入元素，则插入和删除操作的平均时间复杂度都为 $O(\log n)$(请参见第 6 章)。

**6. 平衡二叉搜索树实现**

在最坏情况下，插入和删除操作的时间复杂度都为 $O(\log n)$(请参见第 6 章)。

**7. 二叉堆实现**

在后面的章节中，我们将讨论堆实现的细节。二叉堆实现搜索、插入和删除的时间复杂度均为 $O(\log n)$，而找到最大或最小元素的时间复杂度均为 $O(1)$。

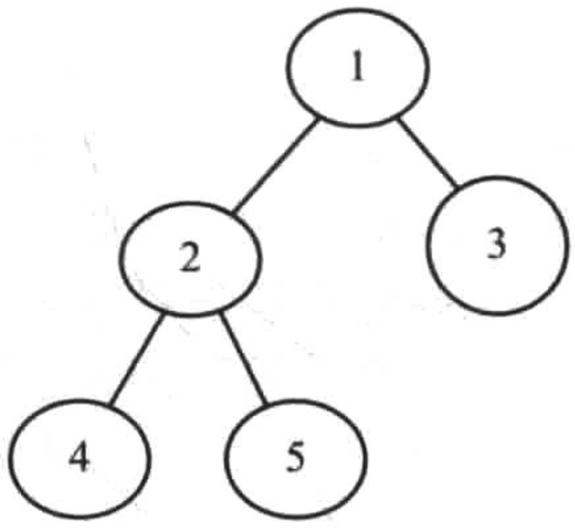
### 8. 实现比较

| 实现 | 插入 | 删除(DeleteMax) | 寻找最小值 |
|---|---|---|---|
| 无序数组 | 1 | $n$ | $n$ |
| 无序链表 | 1 | $n$ | $n$ |
| 有序数组 | $n$ | 1 | 1 |
| 有序链表 | $n$ | 1 | 1 |
| 二叉搜索树 | $\log n$(平均) | $\log n$(平均) | $\log n$(平均) |
| 平衡二叉搜索树 | $\log n$ | $\log n$ | $\log n$ |
| 二叉堆 | $\log n$ | $\log n$ | 1 |

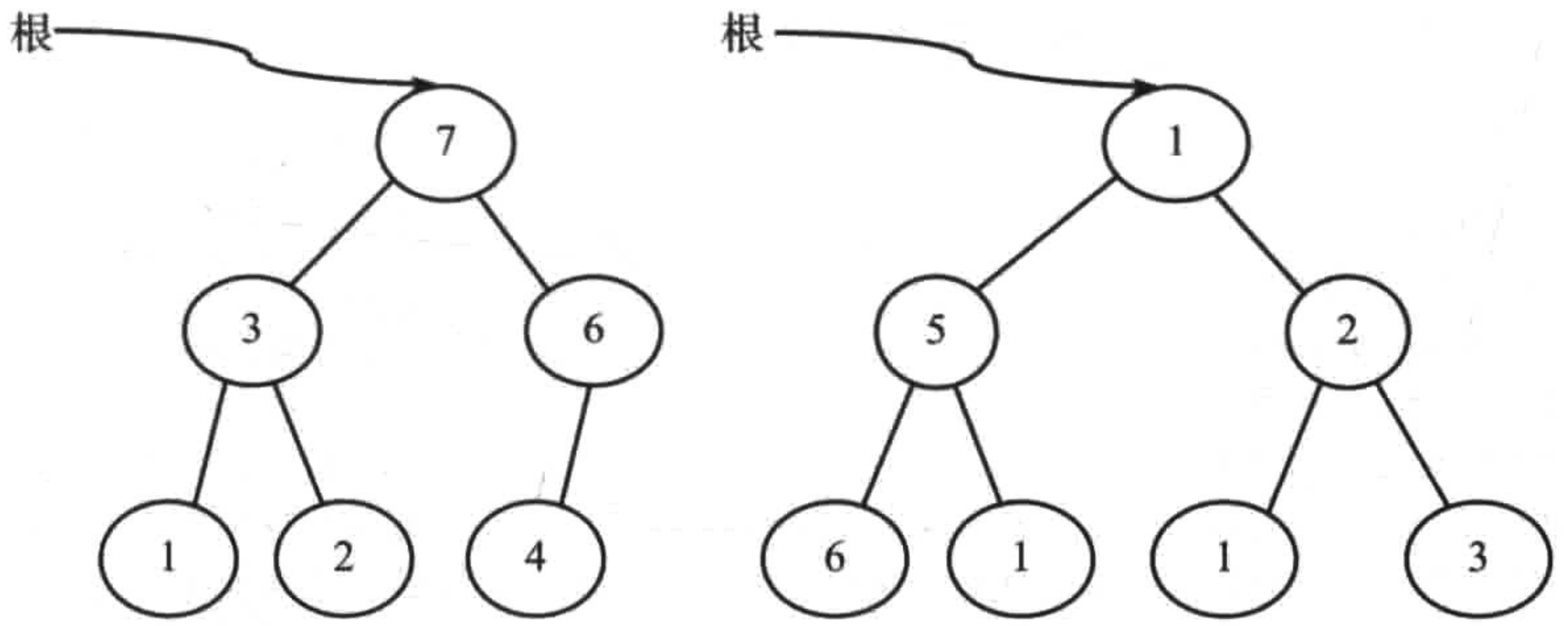
## 7.5　堆和二叉堆

### 1. 什么是堆

堆是一棵具有特定性质的二叉树。堆的基本要求是堆中所有结点的值必须大于或等于(或小于或等于)其孩子结点的值。这也称为堆的性质。堆还有另一个性质，就是当 $h>0$ 时，所有叶子结点都处于第 $h$ 或 $h-1$ 层(其中 $h$ 为树的高度，完全二叉树)。也就是说，堆应该是一棵完全二叉树，如下图所示。



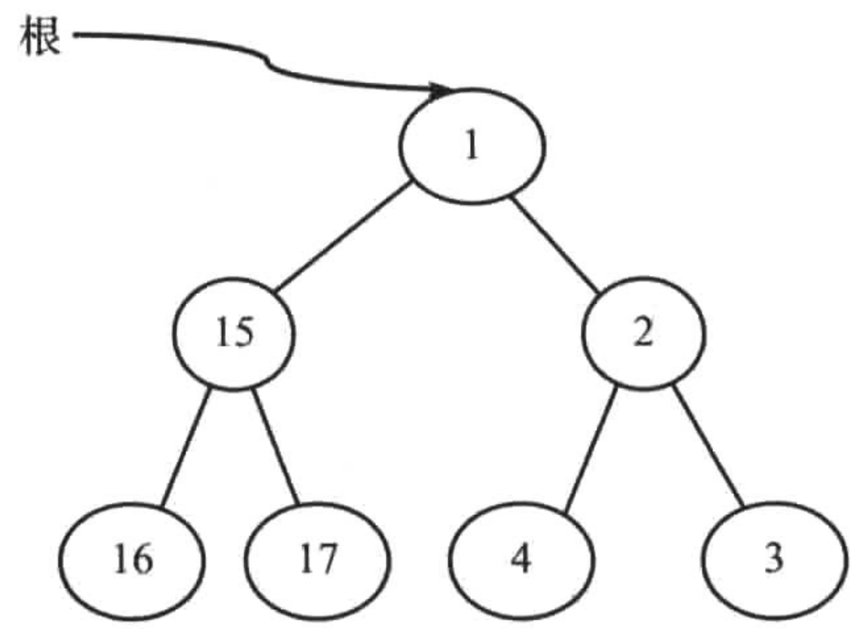

在下面的例子中，左边的树是堆(每个元素都大于其孩子结点的值)，而右边的树不是堆(因为 5 大于右孩子 1)。



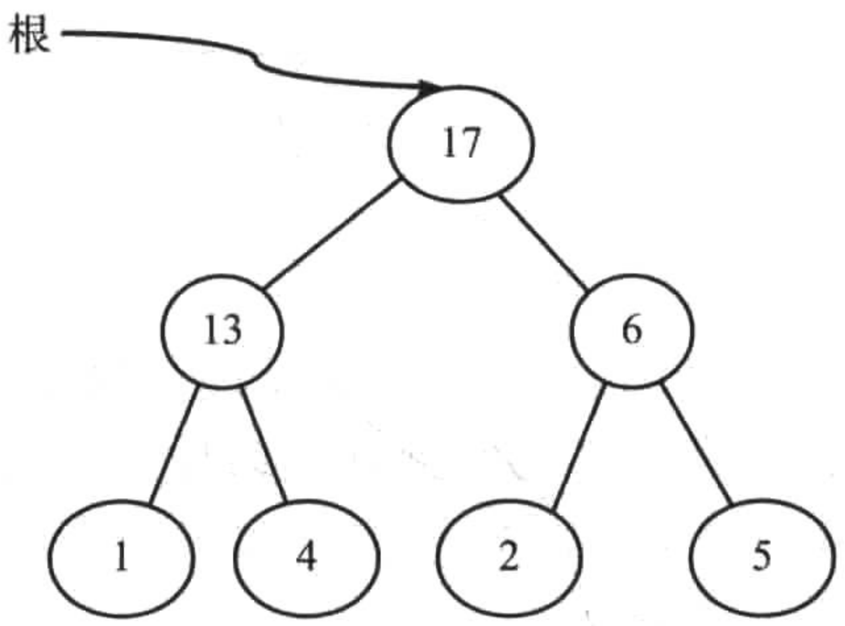

### 2. 堆的类型

根据堆的性质可以将堆分为两类：

- **最小堆：** 结点的值必须小于或等于其孩子结点的值。



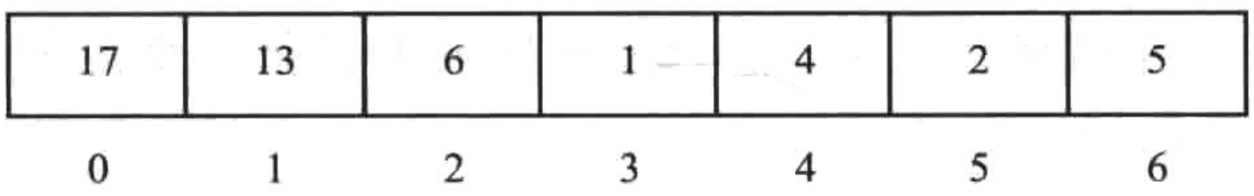

- **最大堆：**结点的值必须大于或等于其孩子结点的值。




## 7.6 二叉堆

在二叉堆中，每个结点最多有两个孩子结点。在实际应用中，二叉堆已经足够满足需求，因此接下来主要讨论二叉最小堆和二叉最大堆。

**堆的表示：**在描述堆的操作前，首先来看堆是怎样表示的。一种可能的方法是使用数组。因为堆在形式上是一棵完全二叉树，用数组来存储它不会浪费任何空间。以下讨论假定所有元素都存储在数组中，从下标 0 开始。上述最大堆可以表示如下：




**注意：**下面的讨论假定操作的对象是最大堆。

### 1. 堆的声明

```
public class  Heap {
        public int[] array;
        public int count;                                   // 堆中元素的个数
        public int capacity;                                // 堆的大小
        public int heap_type;                               // 最小堆或最大堆
        public Heap(int capacity, int heap_type)            {//具体实现如下}
        public Parent(int capacity, int heap_type)          { //具体实现如下 }
        public int LeftChild(int i)                         {//具体实现如下 }
        public int RightChild(int i)                        {//具体实现如下 }
        public int GetMaximum(int i)                        {//具体实现如下 }
        .....
}
```

**注意：**假定下面所有函数都是该类的一部分。

### 2. 创建堆

```
public Heap(int capacity, int heap_type) {
        this.heap_type = heap_type;
        this.count = 0;
        this.capacity = capacity;
        this.array = new int[capacity];
}
```

时间复杂度为 $O(1)$。

### 3. 结点的双亲

对于第 $i$ 个位置上的结点，其双亲结点处在$(i-1)/2$ 位置上。在前面的例子中，元素 6 在 2 号位置，而其双亲在 0 号位置。

```
public int Parent (int i) {
        if(i <= 0 || i >= this.count)
                return -1;
        return i-1/2;
}
```

时间复杂度为 $O(1)$。

### 4. 结点的孩子

类似于上面的讨论，第 $i$ 个位置结点的孩子处在 $2*i+1$ 和 $2*i+2$ 位置上。例如，在上面的树中，元素 6 处在 2 号位置，它的孩子 2 和 5 分别处在 $5(2*i+1=2*2+1)$和 6 号$(2*i+2=2*2+2)$位置。

```
public int LeftChild(int i) {
        int left = 2 * i + 1;
        if(left >= this.count)
                return -1;
        return left;
}
```

时间复杂度为 $O(1)$。

```
public int RightChild(int i) {
        int right = 2 * i + 2;
        if(right >= this.count)
                return -1;
        return right;
}
```

时间复杂度为 $O(1)$。

### 5. 获取最大元素

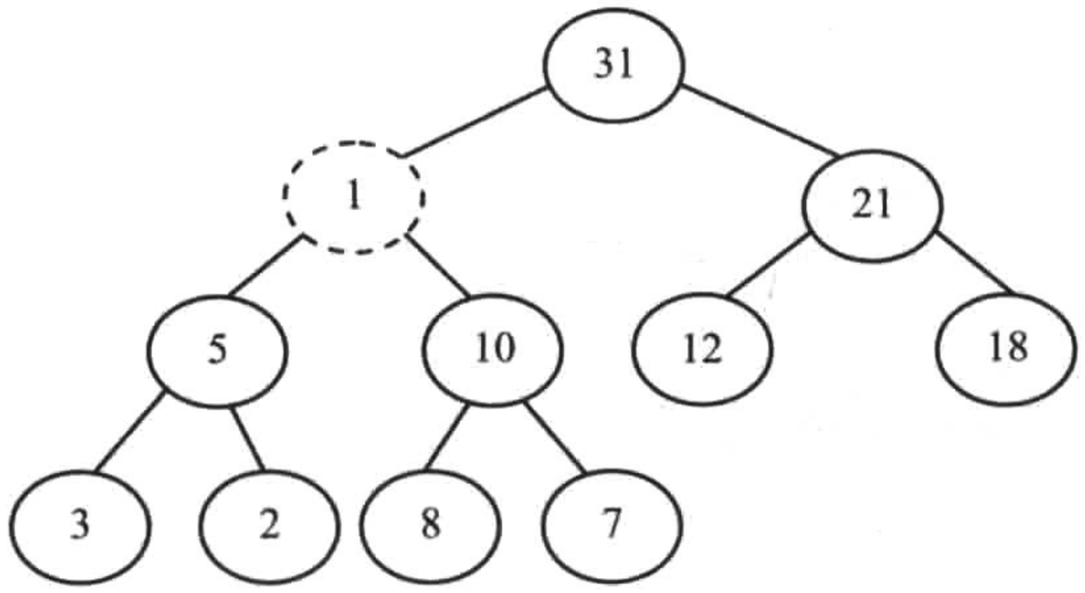
因为最大堆中的最大元素总是根结点，所以它存储在数组的 0 号位置。

```
public int GetMaximum() {
        if(this.count == 0)
                return -1;
        return this.array[0];
}
```

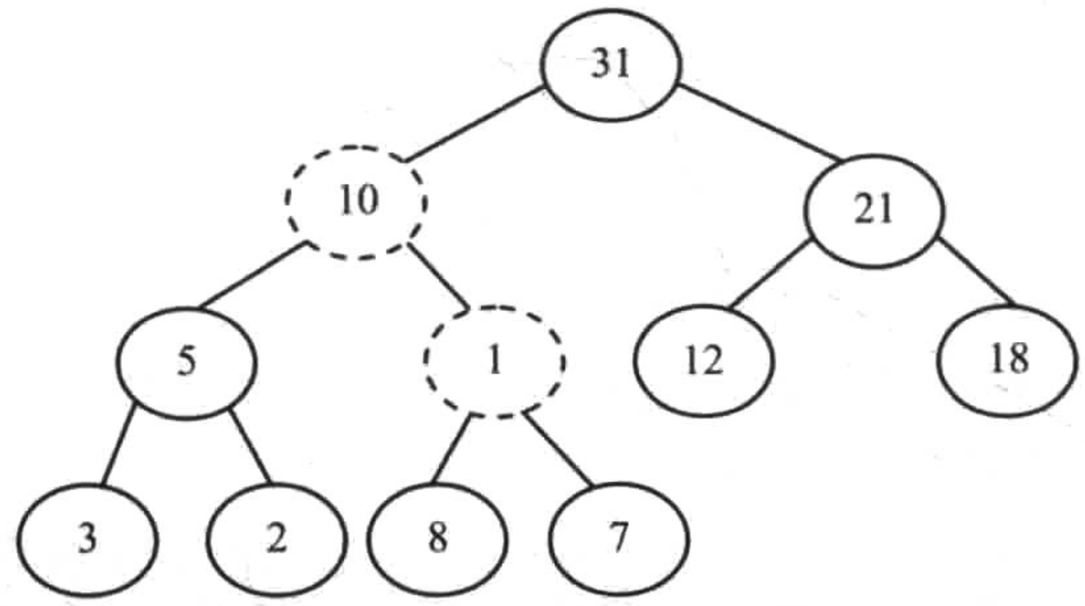
时间复杂度为 $O(1)$。

### 6. 堆化元素

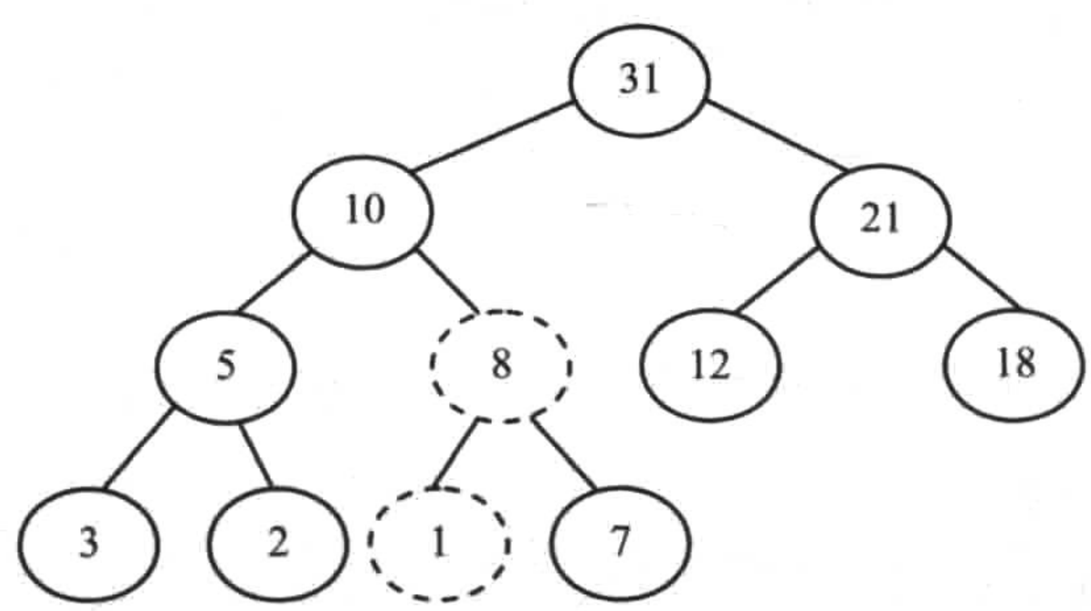
当插入一个元素到堆中时，它可能不满足堆的性质。在这种情况下，需要调整堆中元素的位置使之重新变为堆。这个过程叫作堆化(heapifying)。在最大堆中，要堆化一个元素，需要找到它的孩子结点中的最大值，然后将它与当前元素交换，重复该过程直至每个结点都满足堆的性质为止。




**观察：**堆的一个重要性质是，如果一个元素不满足堆的性质，那么从这个元素开始到根结点的所有元素都存在这个问题。在下图所示的例子中，元素 1 不满足堆的性质，它的父结点 31 也存在同样的问题。类似地，如果堆化该元素，那么从这个元素到根结点的所有元素都自动满足堆的性质。让我们通过一个例子来具体说明。在上面的堆中，元素 1 不满足堆的性质。需要堆化这个元素。为了堆化 1，先找到它孩子结点中的最大值，然后交换它们的位置。




需要持续这个过程，直到元素满足堆的性质。现在，交换 1 和 8。




此时树已经满足堆的性质。在上面堆化的过程中，由于是自顶向下移动，所以这个过程有时又叫作向下渗透。

```
//堆化当前元素
public void PercolateDown(int i) {
        int l, r, max, temp;
        l = LeftChild(i);
        r = RightChild(i);
        if(l != -1 && this.array[l] > this.array[i])
                max = l;
        else
                max = i;
```

```
        if(r != -1&& this.array[r] > this.array[max])
                max = r;
        if(max != i) {  //交换this.array[i]和this.array[max];
                temp = this.array[i]; this.array[i] = this.array[max]; this.array[max] = temp;
        }
        PercolateDown(max);
}
```

时间复杂度为 $O(\log n)$，这是因为堆是一个完全二叉树，在最坏情况下，需要从根结点出发，一直向下到叶子结点。这等于完全二叉树的高度。

空间复杂度为 $O(1)$。

**7. 删除元素**

为了从堆中删除元素，只需要从根结点删除元素。这是标准堆支持的唯一操作(最大元素)。当删除根结点后，将堆(树)的最后一个元素复制到这个位置，然后删除最后的元素。当用最后的元素替换树的根结点后，可能导致树不满足堆的性质。为使其再次成为堆，需要调用函数 PercolateDown。

- 将第一个元素复制到其他变量。
- 将最后一个元素复制到第一个元素位置。
- PercolateDown(第一个元素)。

```
int DeleteMax() {
        if(this.count == 0)
                return -1;
        int data = this.array[0];
        this.array[0] = this.array[this.count-1];
        this.count--; //减小堆的大小
        PercolateDown(0);
        return data;
}
```

**注意：** 使用函数 PercolateDown 删除元素。

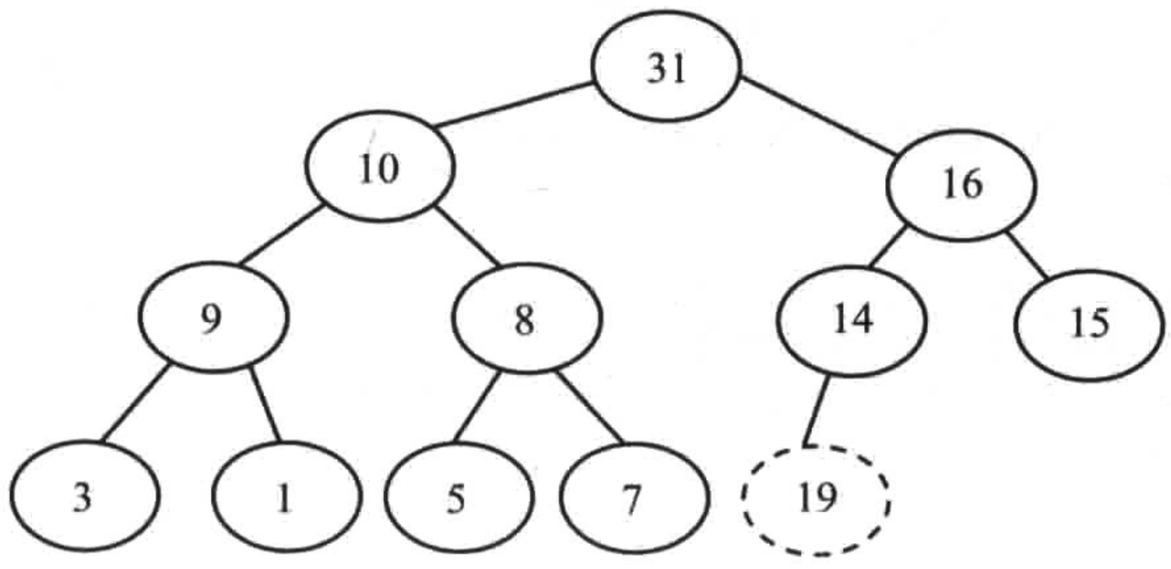
与堆化函数一样，时间复杂度是 $O(\log n)$。

**8. 插入元素**

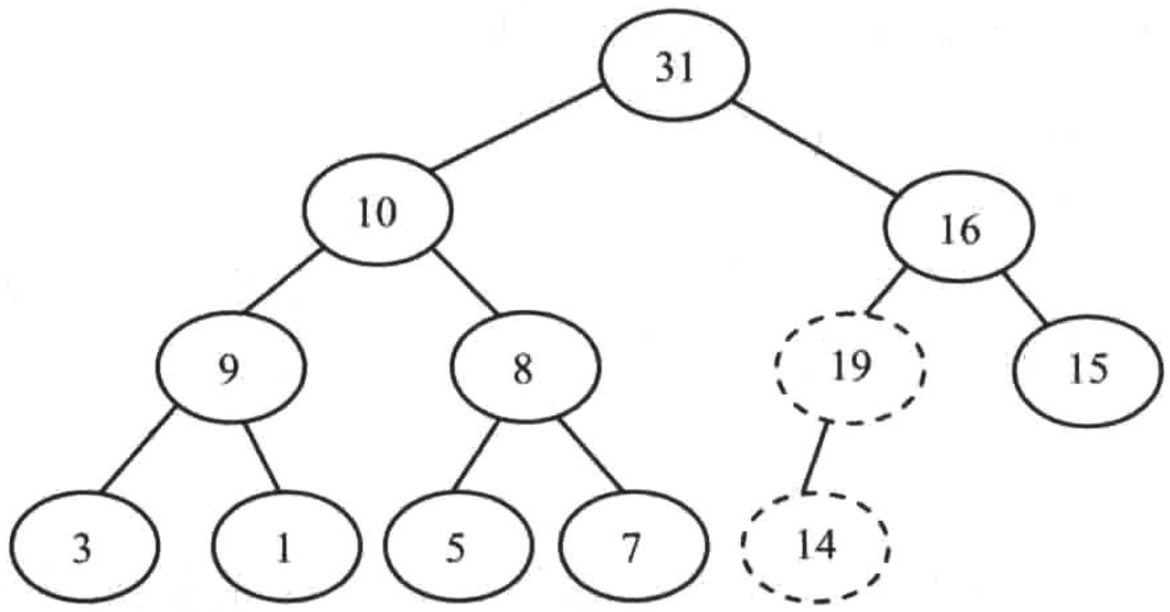
插入元素类似于堆化和删除过程。

- 堆的大小加 1。
- 将新元素放在堆(树)的尾部。
- 从下至上(根)堆化这个元素。

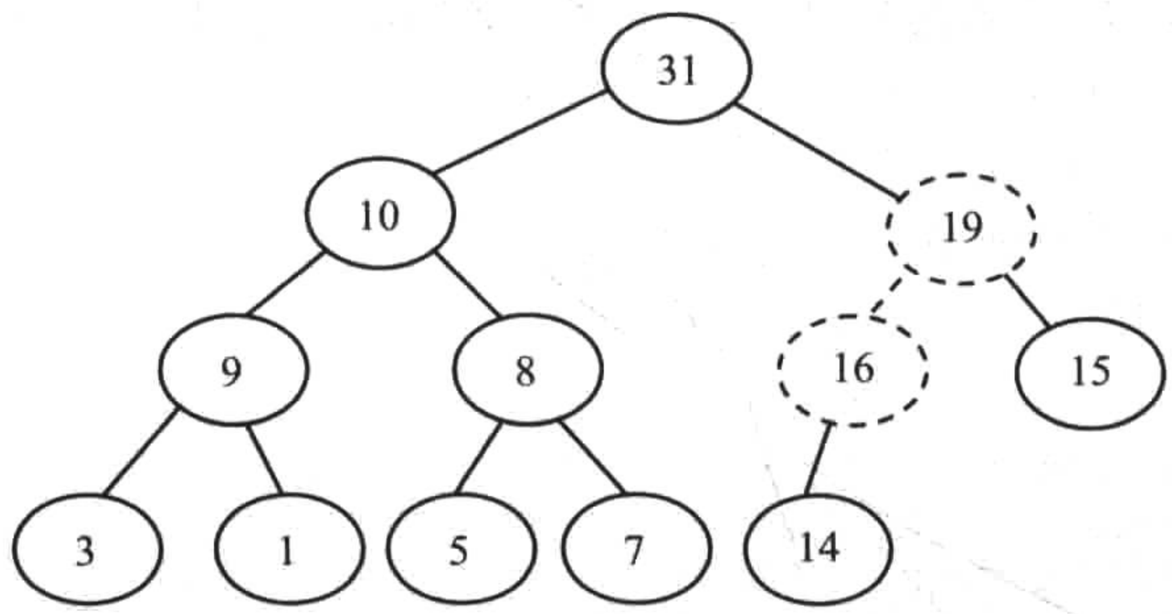
在给出实现代码之前，先来看一个例子。在堆的尾部插入元素 19，这就使得树不再满足堆的性质。



为了堆化元素 19，需要将它与它的双亲大小进行比例，并调整它们。交换 19 和 14，结果如下图所示。




交换 19 和 16，结果如下图所示。




现在该树满足堆的性质。因为这种方法从下至上来比较和调整元素，所以有时也将其称为向上渗透。

```
int Insert(int data) {
        int i;
        if(this.count == this.capacity)
                ResizeHeap();
        this.count++;                    //增加堆的大小来放置新元素
        i = this.count-1;
        while(i >=0 && data > this.array[(i-1)/2]) {
                this.array[i] = this.array[(i-1)/2];
                i = i-1/2;
        }
        this.array[i] = data;
}
void ResizeHeap() {
        int[] array_old = new int[this.capacity];
        System.arraycopy(this.array, 0, array_old, this.count-1);
        this.array = new int[this.capacity * 2];
        if(this.array == null) {
                System.out.println("Memory Error");
                return;
        }
        for (int i = 0; i < this.capacity; i ++)
                this.array[i] = array_old[i];
        this.capacity *= 2;
        array_old = null;
}
```

时间复杂度为 $O(\log n)$。解释与堆化函数一致。

### 9. 清空堆

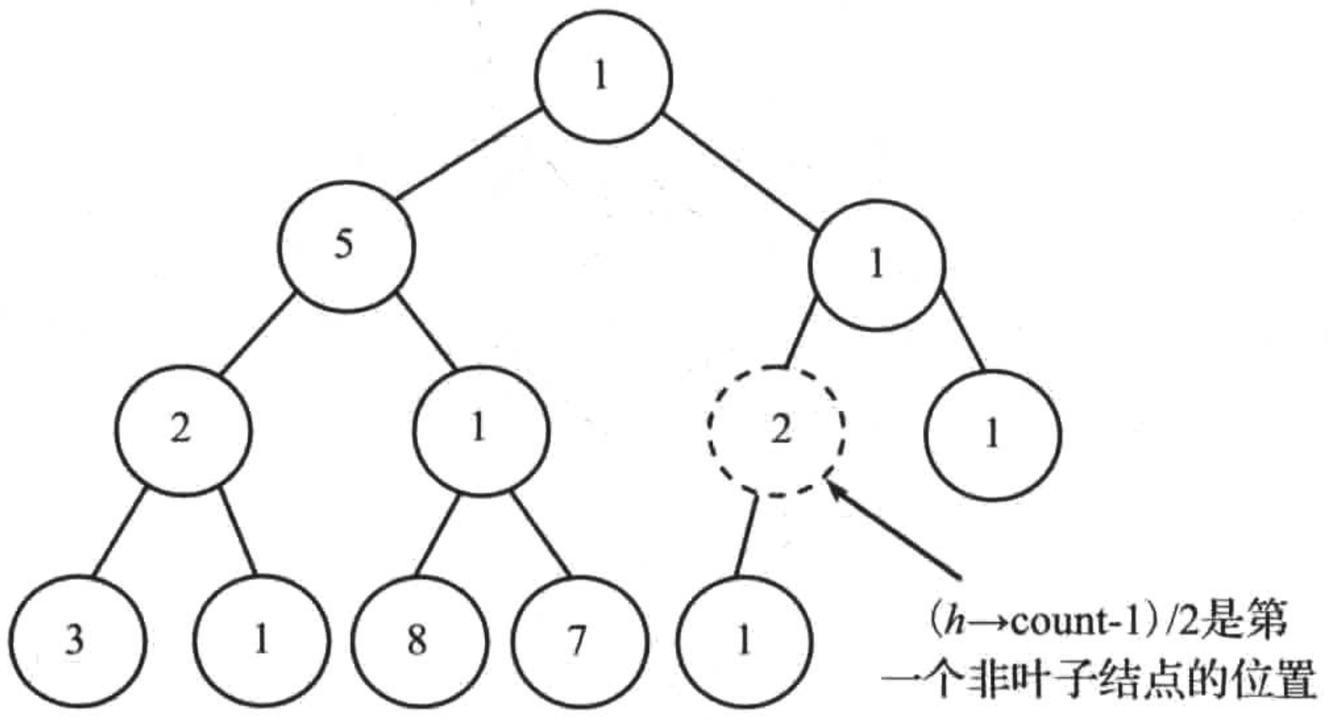
```
void DestroyHeap() {
        this.count = 0;
        this.array = null;
}
```

### 10. 数组建堆

创建堆的一个简单方法是，取 $n$ 个输入数据，然后将它们插入空堆中。这需要 $n$ 次连续的插入操作，在最坏情况下时间代价为 $O(n\log n)$，这是因为每次插入操作的代价为 $O(\log n)$。

**观察**：叶子结点总是满足堆的性质，不需要关注它们。叶子结点元素总是在最后面，因此要对给定的数组建堆，只要堆化非叶子结点即可。现在，先来关注如何找到第一个非叶子结点。堆的最后一个元素所处的位置为 $h\rightarrow$count$-1$，通过最后元素的双亲结点就能找到第一个非叶子结点。

```
void BuildHeap(Heap h, int A[], int n) {
        if(h == null) return;
        while (n > this.capacity)
                h.ResizeHeap();
        for (int i = 0; i < n; i ++)
                h.array[i] = A[i];
        this.count = n;
        for (int i = (n-1)/2; i >=0; i --)
                h.PercolateDown(i);
}
```




时间复杂度：创建堆是线性时间，通过计算所有结点高度的和就可以得出。对于一棵高度为 $h$ 的完全二叉树，它有 $n=2^{h+1}-1$ 个结点，结点高度的和是 $n-h-1=n-\log n-1$（证明请参见 7.7 节）。这就意味着，通过逆序层次遍历对每个非叶子结点函数 PercolateDown，创建堆操作就可以在线性时间 $O(n)$ 内完成。

### 11. 堆排序

堆 ADT 的一个主要应用是排序（堆排序）。堆排序算法首先将所有元素（从未排序的数组中）插入堆中，然后从堆的根结点依次删除它们，直到堆空为止。需要注意的是，堆排序可以利用现有数组就地完成排序。具体做法是，当删除一个元素时，只将数组中第一个元素与最后一个元素交换位置而不是真正地从数组中移除该元素，同时减少堆的大小（数组的大小），然后再对第一个元素进行堆化。持续这个过程直到剩余元素的个数是 1。

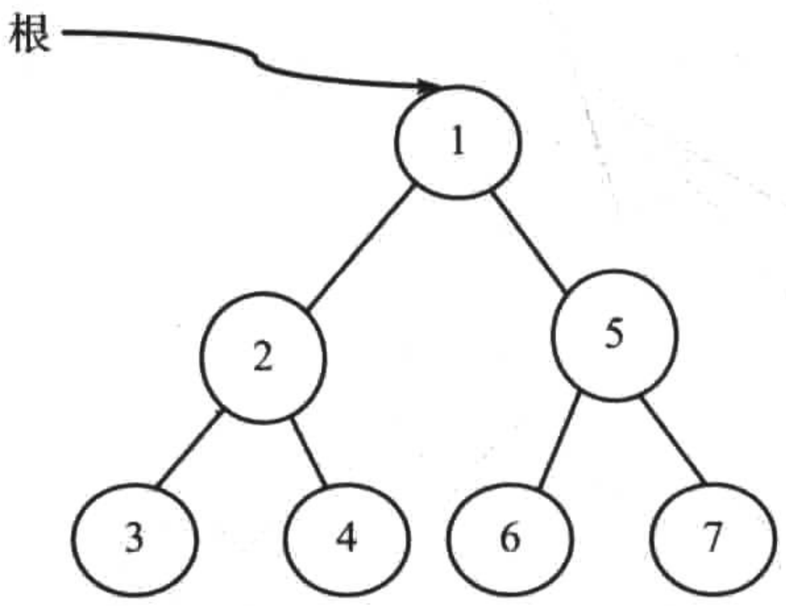
```
void Heapsort(int A[], in n) {
        Heap h = new Heap(n, 0);
        int old_size, i, temp;
        BuildHeap(h, A, n);
        old_size = h.count;
        for(i = n-1; i > 0; i--) { //h.array[0] 存储最大元素
                temp = h.array[0]; h.array[0] = h.array[h.count-1];
                h.count--;
                h.PercolateDown(i);
        }
        h.count = old_size;
}
```

时间复杂度：当从堆中删除元素时，数据就变为有序的(因为最大元素总是在堆的根结点)。由于插入算法和删除算法的时间复杂度为 $O(\log n)$(其中 $n$ 是堆中数据的个数)，所以堆排序的时间复杂度为 $O(n\log n)$。

## 7.7 优先队列(堆)的相关问题

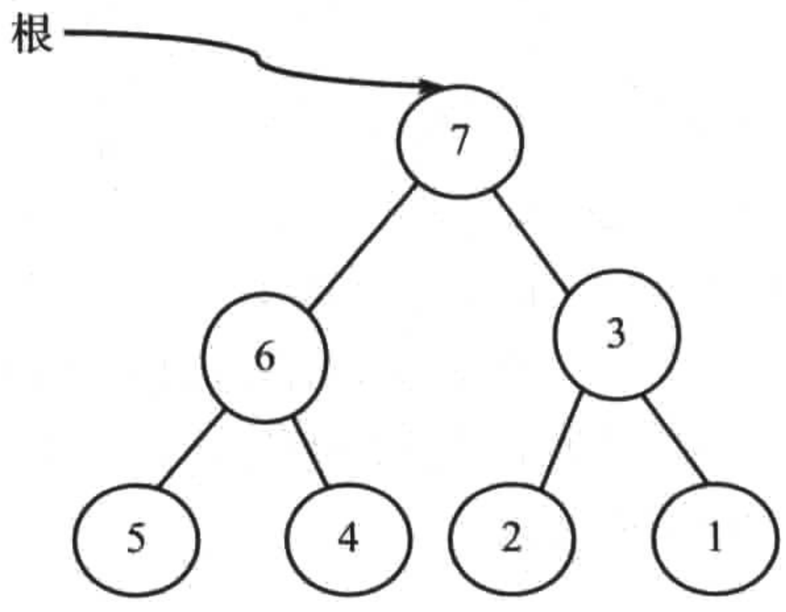
**问题 1** 是否存在一个包含 7 个不同元素的最小堆，其前序遍历序列是有序的？

**解答：是。**对于下面这棵树，前序遍历序列为升序。




**问题 2** 是否存在一个包含 7 个不同元素的最大堆，其前序遍历序列是否是有序的？

**解答：是。**对于下面这棵树，前序遍历序列为降序。

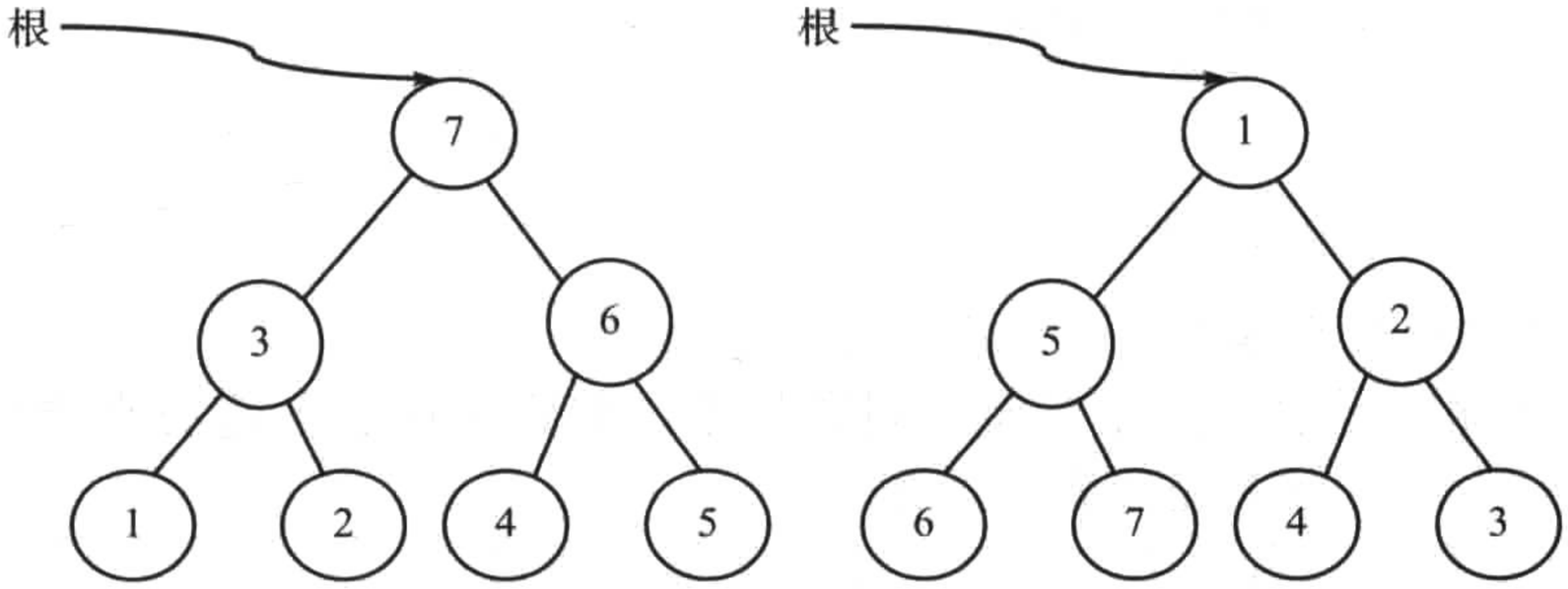



**问题 3** 是否存在一个含有 7 个不同数据的最小堆/最大堆，其中序遍历序列是有序的？

**解答：不存在，**因为堆必须是最小堆或最大堆，相应地根结点的值是最小值或最大值。中序遍历在第二步才访问树的根结点，当树的根结点包含最小值或最大值时，无法在输出序列中找到合适的位置放置该元素。

**问题 4** 是否存在一个含有 7 个不同数据的最小堆/最大堆，其后序遍历序列是有序的？

**解答：是。**如果树为最大堆，可以得到降序序列(下图左边的树)，如果树为最小堆，可以得到升序序列(下图右边的树)。




**问题 5**　高度为 $h$ 的堆，其最小元素个数和最大元素个数是多少？

**解答：**因为堆是一棵完全二叉树(除了最底层外，其他所有层都是满的)，它最多有 $2^{h+1}-1$ 个元素(如果最底层也是满的)。这是因为，要得到最大结点个数，需要将每层都填满，所以最大元素个数是 $h$ 层中所有元素个数之和。

为了得到最小结点个数，需要填满 $h-1$ 层，最后一层只需要填充一个元素即可。由此，最小结点个数为 $h-1$ 层满的元素个数之和再加 1，即 $2^h-1+1=2^h$(当最底层只有 1 个元素，而其余各层都满时)。

**问题 6**　请说明有 $n$ 个元素的堆的高度为 $\log n$？

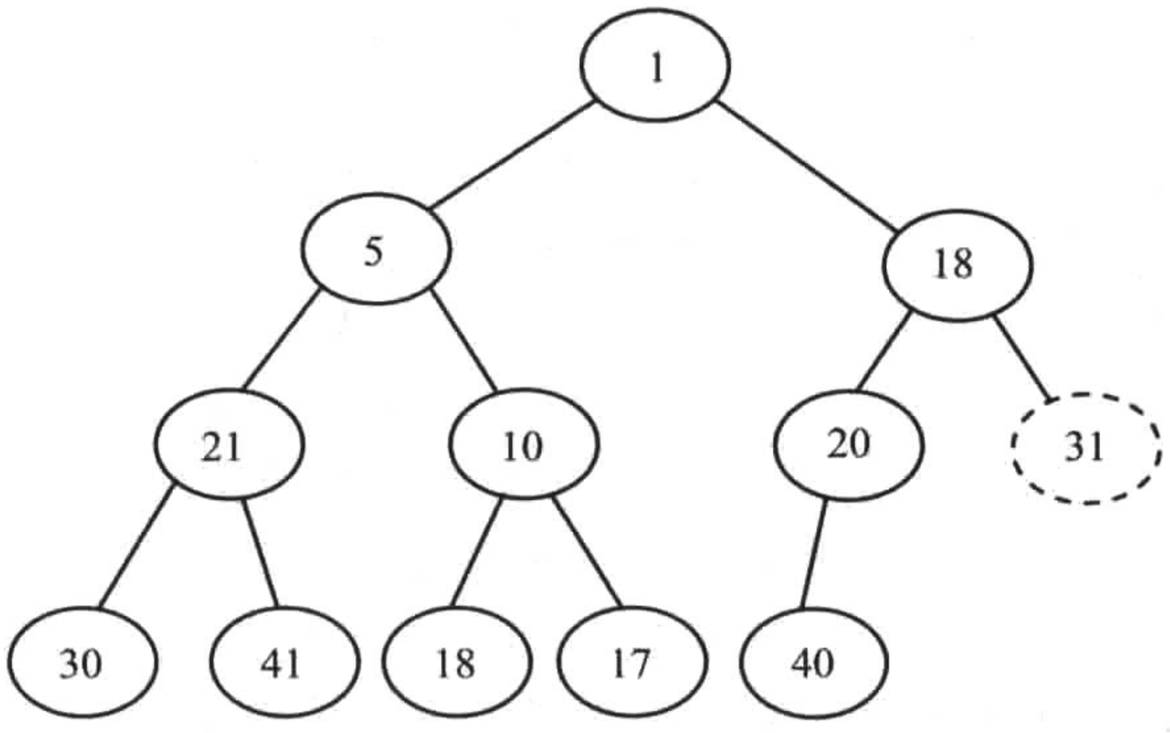
**解答：**堆是完全二叉树。除了最底层外，其他所有层都是满的。因此堆至少有 $2^h$ 个元素，最多有 $2^{h+1}-1$ 元素，即 $2^h \leqslant n \leqslant 2^{h+1}-1$。这表明 $h \leqslant \log n \leqslant h+1$。由于 $h$ 为整数，所以 $h=\log n$。

**问题 7**　给定一个最小堆，给出查找最大元素的算法。

**解答：**对于最小堆，最大元素只可能为叶子结点。接下来的问题是如何查找树的叶子结点。如果仔细观察可知，最后一个结点的双亲结点的下一个结点是第一个叶子结点。由于最后一个元素总是处在 $h \rightarrow \text{count}-1$ 位置，所以双亲结点的下一个结点可以由下式得出：

$$\frac{h \rightarrow \text{count}-1}{2}+1 \approx \frac{h \rightarrow \text{count}+1}{2}$$

接下来，剩下的步骤就是扫描叶子结点，然后找到它们中的最大元素。



```
int FindMaxInMinHeap(Heap h) {
        int Max = -1;
        for(int i = (h.count+1)/2; i < h.count; i++)
                if(h→array[i] > Max)
                        Max = h.array[i];
}
```

时间复杂度为 $O\left(\frac{n}{2}\right)\approx O(n)$。

**问题 8** 给出从最小堆中删除任意元素的算法。

**解答：** 为了删除元素，首先需要找到该元素。假定使用层次遍历来查找元素。找到元素后再用 DeleteMin 函数完成删除操作。

时间复杂度 = 查找元素的时间 + 删除元素的时间

$= O(n) + O(\log n) \approx O(n)$ // 主要考虑查找元素的时间开销

**问题 9** 给出删除最小堆中第 $i$ 个元素的算法。

**解答：**

```
Int Delete(Heap h, int i) {
        int key;
        if(n < i) {
                System.out.println("Wrong position"); return;
        }

        key = h.array[i];
        h.array[i]= h.array[h.count-1];
        h.count--;
        h.PercolateDown(i);
        return key;
}
```

时间复杂度为 $O(\log n)$。

**问题 10** 证明对于高度为 $h$ 的完全二叉树，所有结点的高度之和为 $O(n-h)$。

**解答：** 完全二叉树的第 $i$ 层有 $2^i$ 个结点。此外，一个在第 $i$ 层的结点，其深度为 $i$，高度为 $h-i$。假定 $S$ 表示所有结点的高度之和，则 $S$ 可用下式计算得出：

$$S = \sum_{i=0}^{h} 2^i (h-i)$$

$$S = h + 2(h-1) + 4(h-2) + \cdots + 2^{h-1}(1)$$

等式两边乘以 2，得出：

$$2S = 2h + 4(h-1) + 8(h-2) + \cdots + 2^h(1)$$

现在将 $2S$ 减去 $S$ 得出：

$$2S - S = -h + 2 + 4 + \cdots + 2^h \Rightarrow S = (2^{h+1} - 1) - (h-1)$$

然而，已知高度为 $h$ 的完全二叉树的结点总个数为 $n = 2^{h+1} - 1$。由此得出：$h = \log(n+1)$。最后，用 $n$ 代替 $2^{h+1} - 1$ 可得：

$$S = n - (h-1) = O(n - \log n) = O(n-h)$$

**问题 11** 给出在二叉堆中找出所有小于值 $k$ 的元素的算法。

**解答：** 从堆的根结点开始，如果根结点的值小于 $k$，那么就输出它的值，然后对其左孩子结点和右孩子结点分别递归调用该函数。如果结点的值大于或等于 $k$，那么函数就停止且不输出它的值。这个算法的时间复杂度是 $O(n)$，其中 $n$ 为堆中结点的个数。这个上

界发生在最坏情况下，即堆中每个结点的值都小于 $k$，因此堆中的每个结点都需要调用该函数。

**问题 12**　给出合并两个二叉最大堆(两个堆分别包含 $m$ 和 $n$ 个元素)的算法。假定第一个堆的大小为 $m+n$，第二个堆的大小为 $n$。

**解答：**解决此问题的一个简单方法是：

- 假定第一个数组(大小为 $m+n$)的元素在前面，即数组前面的 $m$ 个单元已填充，后面 $n$ 个单元为空。
- 不改变第一个堆，只是将第二个堆附加到后面，然后再对数组进行堆化处理。
- 由于新数组中元素的个数为 $m+n$，所以每次堆化操作的代价为 $O(\log(m+n))$。

算法的时间复杂度为 $O((m+n)\log(m+n))$。

**问题 13**　能否降低问题 12 的时间复杂度？

**解答：**不必将数组中 $m+n$ 个元素都进行堆化处理，通过“利用数组中元素建堆”技术可以降低其时间复杂度。可以从非叶子结点开始，然后堆化处理它们。算法如下：

- 假定第一个数组(大小为 $m+n$)元素放在前面，即数组前面 $m$ 个单元已填充，后面 $n$ 个单元为空。
- 不改变第一个堆，仅仅将第二个堆附加在第一个堆的后面。
- 接下来，先找到第一个非叶子结点，由此元素开始堆化处理非叶子结点。

在本章的理论部分中可知，构建 $n$ 个元素的堆需要 $O(n)$时间。因此用此技术来合并堆的时间复杂度为 $O(m+n)$。

**问题 14**　是否存在有效的算法来合并 2 个最大堆(存在数组中)？假定都有 $n$ 个元素。

**解答：**解决此问题所选择的方法取决于堆的类型。如果是标准堆，即每个结点最多有两个孩子结点，且叶子结点最多处在不同的两行上，则合并这样的堆算法的最佳性能只能是 $O(n)$。对于大小分别为 $m$ 和 $n$ 的二叉堆的合并，存在时间复杂度为 $O(\log m \times \log n)$的算法。对于 $m=n$，这个算法需要 $O(\log^2 n)$。由于算法较为复杂且超出了本书的范围，所以这里跳过。为得到更好的合并性能，可以使用二叉堆的另一种变型，即 Fibonacci 堆，在平均情况下这种堆的合并操作只需要 $O(1)$。

**问题 15**　给出在最小堆中找到第 $k$ 个最小元素的算法。

**解答：**解决此问题的一个简单方法是：在最小堆上执行 $k$ 次删除操作。

```
int FindKthLargestEle(Heap h, int k) {
        //删除前k-1个元素并返回第k个元素即可
        for(int i=0;i<k-1;i++)
                h.DeleteMin();
        return h.DeleteMin();
}
```

时间复杂度为 $O(k\log n)$。因为需要执行 $k$ 次删除操作，所以每次删除操作的代价为 $O(\log n)$。

**问题 16**　对于问题 15，能否降低其时间复杂度？

**解答：**假定原始最小堆为 HOrig，辅助最小堆为 HAux。初始时，将 HOrig 顶部的最小元素插入 HAux 中。这里不需要在 HOrig 中做删除操作。

```
Heap HOrig, HAux;
int FindKthLargestEle( int k ) {
        int heapElement;//假设堆中的数据为整数
```

```
int count=1;
HAux.Insert(HOrig.DeleteMin());
while( true ) {
        //返回最小元素并将其从HA堆中删除
        heapElement = HAux.DeleteMin();
        if(++count == k ) {
                return heapElement;
        }
        else {    //将HO中最小元素的左孩子和右孩子插入到HA中
                HAux.Insert(heapElement.LeftChild());
                HAux.Insert(heapElement.RightChild());
        }
    }
}
```

每一次 while－loop 迭代中给出了当前最小元素，需要 $k$ 次循环来得到第 $k$ 个最小元素。因为辅助堆的大小总是小于 $k$，所以每执行一次 while－loop 迭代，辅助堆的大小增加 1，在查找期间原始堆不需要做任何操作，所以运行时间为 $O(k\log k)$。

**问题 17** 从最大堆中寻找 $k$ 个最大元素。

**解答：** 求解此问题的一个简单方法是：建立最大堆，然后执行 $k$ 次删除操作。

$$T(n) = \text{执行 } k \text{ 次删除操作} = \Theta(k\log n)$$

**问题 18** 对于问题 17，是否还存在其他的解决方法？

**解答：** 可以使用问题 16 的解决方法。最后辅助堆中包含 $k$ 个最大元素。不需要删除元素，只需要在 HAux 中一直添加元素即可。

**问题 19** 如何用堆来实现栈？

**解答：** 为使用优先队列 PQ(使用最小堆)来实现栈，假定使用一个额外的整数变量 $c$。同时假定 $c$ 的初始值为任意已知值(如 0)。栈 ADT 的实现如下所示，这里 $c$ 用来表示元素在优先队列插入或删除时的优先级。

```
void Push(int element) {
        PQ.Insert(c, element);
        c--;
}

int Pop() {
        return PQ.DeleteMin();
}
int Top() {
        return PQ.Min();
}
int Size() {
        return PQ.Size();
}
int isEmpty() {
        return PQ.isEmpty();
}
```

**注意：** 当元素出栈时，可以使 $c$ 加 1。

**观察：** 可以使用当前系统时间的负数值来代替 $c$(避免溢出)。基于此的实现如下：

```
void Push(int element) {
        PQ.insert(-gettime(),element);
}
```

**问题 20** 如何使用堆来实现队列？

**解答：**为了使用优先队列 PQ(最小堆)来实现队列，类似于栈的模拟方式，假定使用一个额外的整数变量 $c$。同样，假定 $c$ 可以初始化为任意已知值(如 0)。队列 ADT 的实现如下所示。这里 $c$ 用来表示元素从 PQ 中插入/删除时的优先级。

```
void Push(int element) {
        PQ.Insert(c, element);
        c++;
}
int Pop() {
        return PQ.DeleteMin();
}
int Top() {
        return PQ.Min();
}
int Size() {
        return PQ.Size();
}
int isEmpty() {
        return PQ.isEmpty();
}
```

**注意：**当元素出队时，可以使 $c$ 减 1。

**观察：**可以使用当前系统时间的值来代替 $c$(避免溢出)。基于此的实现如下：

```
void Push(int element) {
        PQ.insert(gettime(),element);
}
```

**注意：**唯一的区别是需要取 $c$ 为正数而不是负数。

**问题 21**　给定一个包含上百万个数字的大文件，如何在这个文件中找出最大的 10 个值?

**解答：**切记，当需要寻找最大 $n$ 个元素时，最好的数据结构是优先队列。

此问题的一种解决方法是将数据分割成 1000 个元素的集合，然后将其创建为堆。然后依次从堆中取出 10 个元素。最后采用堆对所有的 10 个元素的集合进行排序，取出前 10 个元素。这种方法的问题是如何存储从每个堆中取出的 10 个元素。因为有几百万个数字，所以这可能需要相当大的内存。

重复使用前面堆中的 10 个元素能解决这个问题。即第一个块用 1000 个元素，后面的每个块都只有 990 个元素。初始时，对第一个 1000 个元素的集合进行堆排序，取出其中最大的 10 个元素并将其与第二个集合中的 990 个元素混合。然后再对这 1000 个元素进行堆排序(10 个来自第一个集合，990 个来自第二个集合)，取出最大的 10 个值，然后与第三个集合的 990 个元素混合。重复此操作直至最后一个集合的 990 个(或小于 990 个)元素，并在最后的堆中取出最大的 10 个元素就可以得到问题的解。

时间复杂度为 $O(n)=n/1000\times$(1000 个元素堆排序的复杂度)。因为对 1000 个元素进行堆排序的复杂度为一个常数，所以复杂度 $O(n)=n$ 为线性复杂度。

**问题 22　归并 $k$ 个有序表，其总的元素个数为 $n$：**已知元素总个数为 $n$ 的 $k$ 个有序表，给出将它们归并到一个简单有序表的算法。

**解答：**由于有 $k$ 个大小相等的线性表，其总的元素的个数为 $n$，所以每个线性表的大小是 $\frac{n}{k}$。解决此问题的一个简单方法是：

- 将第一个线性表与第二个线表归并。因为每个线性表的大小为$\frac{n}{k}$，所以这一步产生大小为$\frac{2n}{k}$的有序表。逻辑上，这类似于归并排序。这步的时间复杂度为$\frac{2n}{k}$，因为需要扫描两个表中的所有元素。
- 然后，将产生的新有序表与第三个线性表归并。这步将产生大小为$\frac{3n}{k}$的有序表。这步的时间复杂度为$\frac{3n}{k}$。因为需要扫描两个表中的所有元素$\left(\text{一个是}\frac{2n}{k}\text{，另一个是}\frac{n}{k}\right)$。
- 重复这个操作直到所有的线性表归并到一个线性表中。

总的时间复杂度为$\frac{2n}{k}+\frac{3n}{k}+\frac{4n}{k}+\cdots+\frac{kn}{k}=\sum_{i=2}^{n}\frac{in}{k}=\frac{n}{k}\sum_{i=2}^{n}i\approx\frac{n(k^2)}{k}\approx O(nk)$。空间复杂度为 $O(1)$。

**问题 23** 对于问题 22，能否降低其时间复杂度？

**解答：**

1）将链表分成对并归并它们。即，每次取两个线性表进行归并，使得对于所有列表，访问元素的总个数为 $O(n)$。这个操作将得到 $k/2$ 个有序表。

2）重复第 1 步直到表的个数为 1 为止。

时间复杂度：第 1 步执行 $\log k$ 次，每个操作访问所有表中的所有 $n$ 个元素，并产生 $k/2$ 个有序表。例如，有 8 个表，经过第 1 步就变成了 4 个，并访问了所有的 $n$ 个元素。第 2 步将变成 2 个表，又访问了所有的 $n$ 个元素。第 3 次变成 1 个表，也访问了 $n$ 个元素。因此总的时间复杂度为 $O(n\log n)$。空间复杂度为 $O(n)$。

**问题 24** 对于问题 23，能否降低其空间复杂度？

**解答：** 可以使用堆来减少空间复杂度。

1）取每个表的第 1 个元素来构建最大堆，大小为 $O(k)$。

2）每一步抽取堆的最大元素，并将其添加到输出的尾部。

3）从被抽取元素所在的表中取出下一个元素放到堆中。即，需要从上一步被抽取元素所在的表中选取其下一个元素。

4）重复第 2 步和第 3 步，直到所有表中的所有元素被处理完。

时间复杂度为 $O(n\log k)$。在某一时刻最大堆中有 $k$ 个元素，对于所有 $n$ 个元素，访问堆只需要 $\log k$ 时间，空间复杂度$=O(\log k)$（最大的堆）。

**问题 25** 已知两个数组 $A$ 和 $B$，各包含 $n$ 个元素。给出一个算法找出最大的 $n$ 个对 $(A[i], B[j])$。

**解答：**

**算法：**

- 对 $A$ 和 $B$ 分别创建堆。这步需要 $O(2n)\approx O(n)$。
- 然后依次分别删除两个堆中的元素，每一步花费 $O(2\log n)\approx O(\log n)$。

总的时间复杂度为 $O(n\log n)$。

**问题 26 最小-最大堆：** 给出一个算法，实现在 $O(1)$时间内求出最小值和最大值，在 $O(\log n)$时间内实现插入、删除最小值和删除最大值的操作。即设计数据结构支持如下操作：

| 操作 | 复杂度 | 操作 | 复杂度 |
|---|---|---|---|
| 输入 | $O(n)$ | 寻找最大值 | $O(1)$ |
| 插入 | $O(\log n)$ | 删除最小值 | $O(\log n)$ |
| 寻找最小值 | $O(1)$ | 删除最大值 | $O(\log n)$ |

**解答：** 这个问题可用两个堆来解决。将这两个堆分别表示为最小堆 $H_{min}$ 和最大堆 $H_{max}$。同时假定在这两个数组中的元素都有互指针(mutual pointer)。即在 $H_{min}$ 中的元素有指向 $H_{max}$ 中相同元素的指针，而在 $H_{max}$ 的元素也有指向 $H_{min}$ 中相同元素的指针。

| | |
|---|---|
| Init | 以 $O(n)$时间创建 $H_{min}$，以 $O(n)$时间创建 $H_{max}$ |
| Insert(x) | 以 $O(\log n)$时间在 $H_{min}$ 中插入 $x$，以 $O(\log n)$时间在 $H_{max}$ 中插入 $x$，以 $O(1)$时间更新指针 |
| FindMin() | 以 $O(1)$时间返回 root($H_{min}$) |
| FindMax | 以 $O(1)$时间返回 root($H_{max}$) |
| DeleteMin | 以 $O(\log n)$时间从 $H_{min}$ 删除最小值，以 $O(\log n)$时间采用指针删除 $H_{max}$ 中相同的元素 |
| DeleteMax | 以 $O(\log n)$时间从 $H_{max}$ 删除最大值，以 $O(\log n)$时间采用指针删除 $H_{min}$ 中相同的元素 |

**问题 27**　动态中位数的查找。设计一个堆数据结构来支持查找中位数。

**解答：** 在 $n$ 个元素的集合中，中位数是指中间元素，即在集合中小于中位数的元素个数等于大于中位数的元素个数。如果 $n$ 是奇数，则将集合排序后即可找到中位数。如果 $n$ 是偶数，中位数常定义为中间两个数的平均数。即使表中有相等的数据这个算法也是可行的。例如，集合{1，1，2，3，5}的中位数是 2，而集合{1，1，2，3，5，8}的中位数是 2.5。

“中位数堆”是堆的变型，可以用来访问中位数。中位数堆可以用两个堆来实现，每个堆各包含一半的数据元素。一个是最大堆，包含最小的元素；另一个是最小堆，包含最大元素。如果元素的总个数是偶数，则最大堆的大小可能等于最小堆的大小。在这种情况下，中位数是最大堆中最大值和最小堆中最小值的平均值。如果元素的个数是奇数，最大堆将比最小堆多一个元素。在这种情况下，中位数就是最大堆中的最大值。

**问题 28**　滑动窗口中的最大和。已知数组 $A[]$和一个大小为 $w$ 的滑动窗口，该窗口从数组的最左边移动到最右边。假定只能看到窗口中的 $w$ 个数，并且每次滑动窗口往右移动一个位置。例如，数组＝[1 3 −1 −3 5 3 6 7]，$w$=3。

| 窗口位置 | 最大值 | 窗口位置 | 最大值 |
|---|---|---|---|
| [1 3 −1] −3 5 3 6 7 | 3 | 1 3 −1 [−3 5 3] 6 7 | 5 |
| 1 [3 −1 −3] 5 3 6 7 | 3 | 1 3 −1 −3 [5 3 6] 7 | 6 |
| 1 3 [−1 −3 5] 3 6 7 | 5 | 1 3 −1 −3 5[3 6 7] | 7 |

**输入：** 长数组 $A[]$和宽度为 $w$ 的窗口。**输出：** 数组 $B[]$，$B[i]$是从 $A[i]$到 $A[i+w-1]$的最大值。**要求：** 寻找一个最优方法来得到 $B[i]$。

**解答：** 可以采用蛮力法来解决，每次窗口移动时，在窗口中寻找 $w$ 个元素。

时间复杂度为 $O(nw)$。

**问题 29**　对于问题 28，能否降低其复杂度？

**解答：可以。**使用堆数据结构。它能将算法的时间复杂度降低到 $O(n\log w)$。插入操作需要 $O(\log w)$时间，其中 $w$ 为堆的大小。然而，获取最大值是容易的，它只需要常数时间，因为最大值总是处在堆的根部(头部)。当窗口向右滑动时，堆中的有些元素将不再有效(处在当前窗口外的数据)。那么应该如何删除它们呢？这里需要非常小心。由于只能移除窗口范围外的元素，所以还需要跟踪元素的位置索引。

**问题 30** 对于问题 28，能否进一步降低其复杂度？

**解答：可以。**双端队列是求解此问题的最佳数据结构。它支持从队列的后端插入/前端删除。关键是要找到一种方法，使窗口中的最大元素总是出现在队列的前面。当从队列中入队和出队元素时，如何保证这个要求。

而且，注意队列中有一些冗余元素，这些元素根本不需要考虑。例如，假设当前队列有元素：[10 5 3]，窗口中的新元素是 11，只需要清空队列而不需要考虑元素 10、5、3，然后将元素 11 插入队列中即可。

通常，人们想要尽量将队列大小与窗口大小保持一致。试着摆脱这个想法，换一个角度来思考这个问题。删除冗余的元素，并且只将需要考虑的元素存储到队列中是实现如下效率为 $O(n)$的解决方案的关键。这是因为表中的每个元素最多被插入和删除一次。因此总的插入和删除操作之和为 $2n$。

```
void MaxSlidingWindow(int A[], int n, int w, int B[]) {
        DoubleEndQueue Q = new DoubleEndQueue();
        for (int i = 0; i < w; i++) {
                while (!Q.isEmpty() && A[i] >= A[Q.QBack()])
                        Q.PopBack();
                Q.PushBack(i);
        }
        for (int i = w; i < n; i++) {
                B[i-w] = A[Q.QFront()];
                while (!Q.isEmpty() && A[i] >= A[Q.QBack()])
                        Q.PopBack();
                while (!Q.isEmpty() && Q.QFront() <= i-w)
                        Q.PopFront();
                Q.PushBack(i);
        }
        B[n-w] = A[Q.QFront()];
}
```

**问题 31** 优先队列是一个表，表中的每个数据项关联一个优先级。数据项从优先队列中退出时是按照它们的优先级顺序进行的，第一个退出的数据项拥有的优先级最高。如果需要最大优先级的数据项，可以构建一个堆，这个堆中每个结点的优先级要大于它的孩子结点的优先级。

设计这样一个堆，首先是中间优先级的数据项被收回。如果堆中有 $n$ 个数据项，当 $n$ 为奇数时，比中间优先级小的数据项的个数是 $\frac{n}{2}$，否则是 $\frac{n}{2}\mp 1$。

解释收回和插入是如何操作的，计算它们的复杂度，并解释数据结构是如何构建的。

**解答：**可以使用一个最小堆和一个最大堆，并使最小堆的根结点大于最大堆的根结点。最小堆的大小应当等于或比最大堆的大小少 1。因此，中间元素总是最大堆的根结点。

对于插入操作，如果新的数据项小于最大堆的根结点，那么就将其插入最大堆中，否则就插入最小堆中。在回收操作或插入操作后，如果堆的大小不满足上述给定的要求，

那么就将最大堆的根元素传送给最小堆或者相反。

利用上面的实现方法，插入和回收操作将在 $O(\log n)$时间内完成。

**问题 32**　已知两个堆，如何合并它们?

**解答：**二叉堆能够快速地支持各种操作：寻找最小值、插入、减少键值。如果有两个最小堆，H1 和 H2，没有有效的方法将它们合并成一个最小堆。

为了有效地解决此问题，可以使用可归并堆。可归并堆支持高效联合操作。它是一种支持如下操作的数据结构：

- Create-Heap()：创建空堆。
- Insert($H$，$X$，$K$)：将键值为 $K$ 的元素 $x$ 插入堆 $H$ 中。
- Find-Min($H$)：返回拥有最小键值的元素。
- Delete-Min($H$)：返回和删除。
- Union(H1，H2)：归并堆 H1 和 H2。

可归并堆的实例如下：

- 二项堆。
- 斐波那契堆。

这两种堆也支持如下操作：

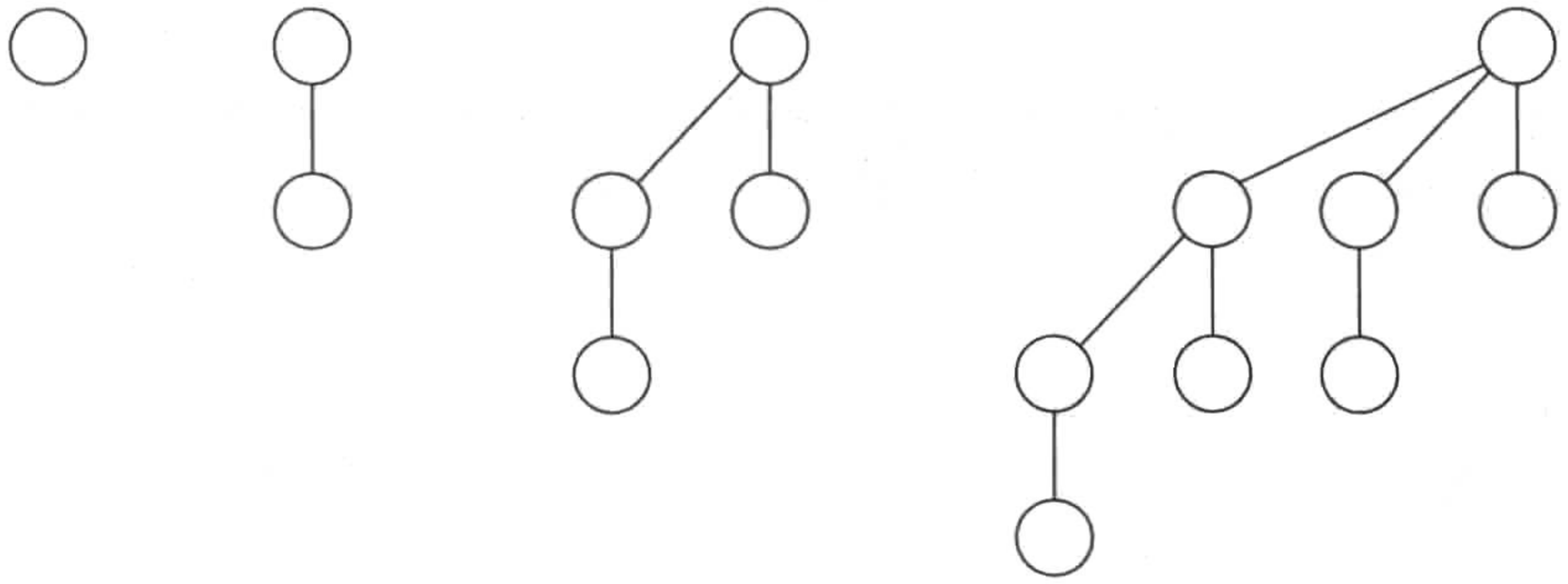
- Decrease-Key($H$，$X$，$K$)：给元素 $Y$ 设置一个更小的键值 $K$。
- Delete($H$，$X$)：删除元素 $X$。

**二项堆：**与只由一棵树组成的二叉堆不同，一个二项堆是由一组组件树构成的集合，当执行联合操作时，不需要重建所有的组件树。每个组件树具有特定的格式，称为二项树。

一个 $k$ 阶的二项树，用 $B_k$ 表示，其递归定义如下：

- $B_0$ 是一棵只有一个结点的树。
- 对于 $k \geqslant 1$，$B_k$ 由两个 $B_{k-1}$ 树连接而成，其中一棵树的根结点是另一棵树根结点的最左孩子结点。

**例子：**



$B_0$　　$B_1$　　$B_2$　　$B_3$

**斐波那契堆：**斐波那契堆是可归并堆的另一个例子。对于任何操作它没有最好和最坏情况(除了插入/建堆)。在执行每个操作时斐波那契堆能很好地分摊计算开销。与二项堆一样，斐波那契堆由一组最小堆的有序组件树组成。然而，与二项树不同的是，它还具有如下性质：

- 树的数目不受限制(最多可到 $O(n)$)。
- 树的高度不受限制(最多可到 $O(n)$)。

而且，在最坏情况下，寻找最小值、删除最小值、联合、减少键值、删除等操作的运行时间为 O($n$)。然而，开销分摊后，每个操作执行得很快。

| 操作 | 二叉堆 | 二项堆 | 斐波那契堆 |
|---|---|---|---|
| 创建堆 | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| 寻找最小值 | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(1)$ |
| 删除最小值 | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ |
| 插入 | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(1)$ |
| 删除 | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ |
| 减少键值 | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(1)$ |
| 联合 | $\Theta(n)$ | $\Theta(\log n)$ | $\Theta(1)$ |

**问题 33** 求整数的无穷级数的中位数。

**解答：** 中位数是有序表中的中间数(如果有序表有奇数个元素)。如果有偶数个数据元素，则中位数是有序表中中间两个数的平均值。

用两个堆可以有效地解决此问题：一个最大堆和一个最小堆。

1) 最大堆包含整数中最小的那一半。

2) 最小堆中包含整数中最大的那一半。

最大堆中的整数总是小于或等于最小堆中的整数。同时，最大堆中的元素个数与最小堆中的元素个数相等或者比最小堆中的元素个数多 1。

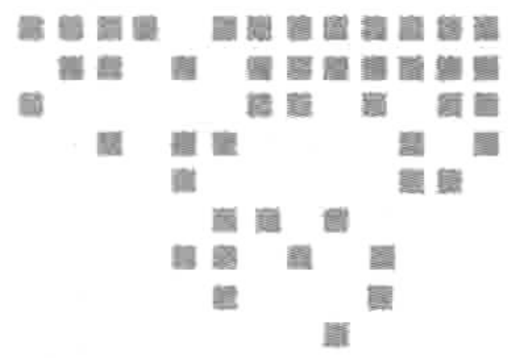
如果从数据流中得到 $2n$ 个元素(在任意时间点)，则最大堆和最小堆将含有相等的数据元素个数(在这种情况下，每个堆中有 $n$ 个元素)。否则，如果接收到 $2n+1$ 个元素，则最大堆将包含 $n+1$ 个元素，而最小堆中有 $n$ 个元素。

接下来，查找中位数：如果有 $2n+1$ 个元素(奇数)，则接收到的元素中的中位数将是最大堆中的最大值(就是最大堆的根结点)。否则，接收到的元素中的中位数是最大堆中的最大值(就是最大堆的根结点)与最小堆中的最小值(就是最小堆的根结点)的平均值。这可以在 $O(1)$时间内计算完成。

将一个元素插入堆中能够在 $O(\log n)$时间内完成。注意，任何包含 $n+1$ 个元素的堆可能需要一次删除操作(并且在另一个堆中执行一次插入操作)。

**例子：**

- 插入 1：插入最大堆。
  MaxHeap：{1}，MinHeap：{}
- 插入 9：插入最小堆，因为 9 大于 1 且最小堆中保存最大元素。
  MaxHeap：{1}，MinHeap：{9}
- 插入 2：插入最大堆，因为 2 小于最小堆中的所有元素。
  MaxHeap：{1，2}，MinHeap：{9}
- 插入 0：因为最大堆包含的数据已经多于一半，所以必须要删除最大堆中的最大值并将其插入最小堆中。因此删除 2，并把它插入最小堆中。操作完成后变为：
  MaxHeap：{1}，MinHeap：{2，9}
- 这时，将 0 插入最大堆中。
  总的时间复杂度为 $O(\log n)$。



# 第8章 Chapter 8

# 并查集 ADT

## 8.1 引言

本章将介绍一种非常重要的数学概念：集合。它说明如何表示一组无需考虑顺序的元素。并查集 ADT 可以表示一组无序元素，可用来解决等价问题。并查集易于实现，使用一个简单数组就能实现它，且每个函数也只需几行代码。在许多算法中，并查集 ADT 是作为一个辅助数据结构而存在的(例如，图论中的 Kruscal 算法)。在讨论并查集 ADT 前，首先了解集合的几个基本性质。

## 8.2 等价关系和等价类

为了方便后续的讨论，假定 $S$ 是集合，它包含元素和定义在集合上的关系 $R$。对于集合中的每一对元素 $a$，$b \in S$，$aRb$ 要么为真，要么为假。如果 $aRb$ 为真，则 $a$ 与 $b$ 相关，否则 $a$ 与 $b$ 不相关。如果一个关系 $R$ 满足如下性质，则该关系是等价关系：

- 自反性：对于任意元素 $a \in S$，$aRa$ 为真。
- 对称性：对于任意两个元素 $a$，$b \in S$，如果 $aRb$ 为真，那么 $bRa$ 也为真。
- 传递性：对于任意三个元素 $a$，$b$，$c \in S$，如果 $aRb$ 和 $bRc$ 都为真，那么 $aRc$ 也为真。

例如，一个整数集合上的关系≤(小于或等于)和≥(大于或等于)就不是等价关系。它们有自反性($a \leqslant a$)和传递性($a \leqslant b$，$b \leqslant c$ 则 $a \leqslant c$)，但没有对称性($a \leqslant b$ 成立，但 $b \leqslant a$ 不成立)。

类似地，铁路连接是一种等价关系。这种关系是自反的，因为任何位置都连接它自身。如果城市 $a$ 和城市 $b$ 之间有一条连接线，那么城市 $b$ 也连接城市 $a$，所以关系也是对称的。如果城市 $a$ 连接城市 $b$ 且城市 $b$ 连接城市 $c$，那么城市 $a$ 也连接城市 $c$。

元素 $a \in S$ 的等价类是 $S$ 的一个子集，该子集包含所有与 $a$ 相关的元素。等价类对集

合 $S$ 产生一个分割。$S$ 中的每个成员都只属于一个等价类。判断 $aRb$ 是否为真，需要判断 $a$ 和 $b$ 是否属于同一个等价类。

在上述例子中，如果两个城市之间有铁路相连，那么它们属于同一个等价类。否则，它们属于不同的等价类。

由于任意两个等价类的交集为空(∅)，所以等价类有时也称为并查集。在后续的章节中，我们将了解在等价类上执行的一些操作。这些操作包括：

- 创建一个等价类(构造一个集合)。
- 查找等价类(FIND)。
- 合并等价类(UNION)。

## 8.3 并查集 ADT

为了处理集合中的元素，需要在集合上定义一些基本操作。本章主要关注以下集合操作：

- MAKESET($X$)：创建仅包含元素 $X$ 的新集合。
- UNION($X$，$Y$)：通过合并元素 $X$ 和 $Y$ 来产生新集合，同时删除包含 $X$ 和 $Y$ 的原集合。
- FIND($X$)：返回包含元素 $X$ 的集合名。

## 8.4 应用

并查集 ADT 有许多应用，列举几个如下：

- 表示网络连通性。
- 图像处理。
- 查找最近公共祖先。
- 定义有限状态自动机的等价性。
- Kruscal 最小生成树算法(图论)。
- 博弈算法。

## 8.5 并查集 ADT 实现中的权衡

首先了解实现并查集操作的一些方法。初始时，假设输入元素为 $n$ 个集合，每个集合仅有一个元素。这说明初始表示假定所有关系都是假的(除了自反性外)。每个集合都有不同元素，因此 $S_i \cap S_j = \phi$。这使得各个集合不相交。

为添加关系 $aRb$(UNION)，需要首先检查 $a$ 和 $b$ 是否已经相关。可以通过在 $a$ 和 $b$ 上执行 FIND 操作来进行验证，并判断它们是否在同一个等价类(集合)中。如果它们不在同一个等价类，那么就执行 UNION 操作。该操作是将包含 $a$ 和 $b$ 的两个等价类合并到一个新的等价类，即创建集合 $S_k = S_i \cup S_j$，同时删除集合 $S_i$ 和 $S_j$。主要有两种方式实现上述 FIND/UNION 操作：

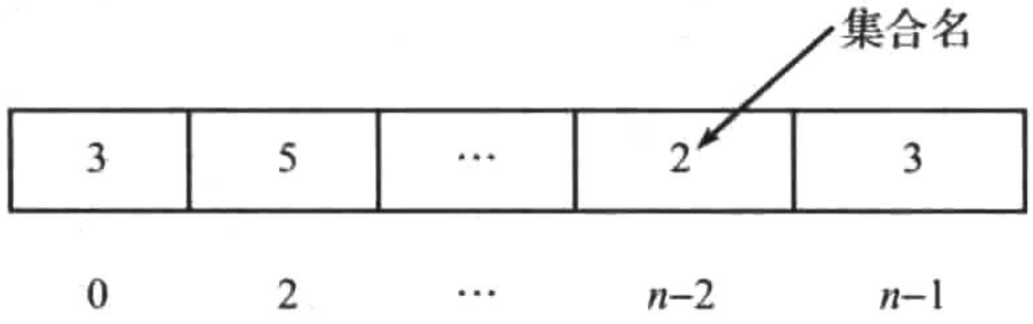
- 快速 FIND 实现(也叫作 Quick FIND)。
- 快速 UNION 操作实现(也叫作 Quick UNION)。

### 1. 快速 FIND 实现(Quick FIND)

可以使用数组来实现这种方法。以下面的数组为例。在下面的表示中数组包含每个

元素的集合名。为了简单起见，假定所有元素都是按顺序从 0～$n-1$ 编号。

在下面的例子中，元素 0 的集合名为 3，元素 2 的集合名为 5，以此类推。利用这种表示方法，FIND 操作只需 $O(1)$时间。这是因为通过访问数组位置可以在常数时间内找到任意元素的集合名。




在这种表示中，为了执行 UNION($a$，$b$)(假定 $a$ 在集合 $i$ 中，$b$ 在集合 $j$ 中)，需要扫描整个数组，并将所有 $i$ 中元素移动到集合 $j$ 中。这需要花费 $O(n)$时间。在最坏情况下，$n-1$ 个并集操作序列需要的时间为 $O(n^2)$。如果有 $O(n^2)$个 FIND 操作，则该性能是良好的，因为对于每个 UNION 或 FIND 操作，其平均时间复杂度为 $O(1)$。如果 FIND 操作很少，那么该时间复杂度就是不可接受的。

**2. 快速 UNION 实现(Quick UNION)**

在本节和下一节中，我们将讨论更快的 UNION 实现和它的变型。有许多不同的方法来实现 UNION 操作，如下列表给出了其中一些方法：

- 快速 UNION 实现(慢 FIND)。
- 快速 UNION 实现(快 FIND)。
- 利用路径压缩来实现快速 UNION。

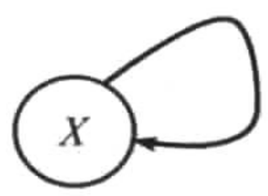
## 8.6　快速 UNION 实现(慢 FIND)

由上述讨论可知，当且仅当元素在同一个集合中时，FIND 操作才返回相同的值(集合名)。在表示并查集时，主要目标是为每一组赋予不同的集合名。通常，集合名并不重要。实现这种集合的一种可能方法是树，因为每一个元素都有唯一的根结点，可以使用它作为集合名。

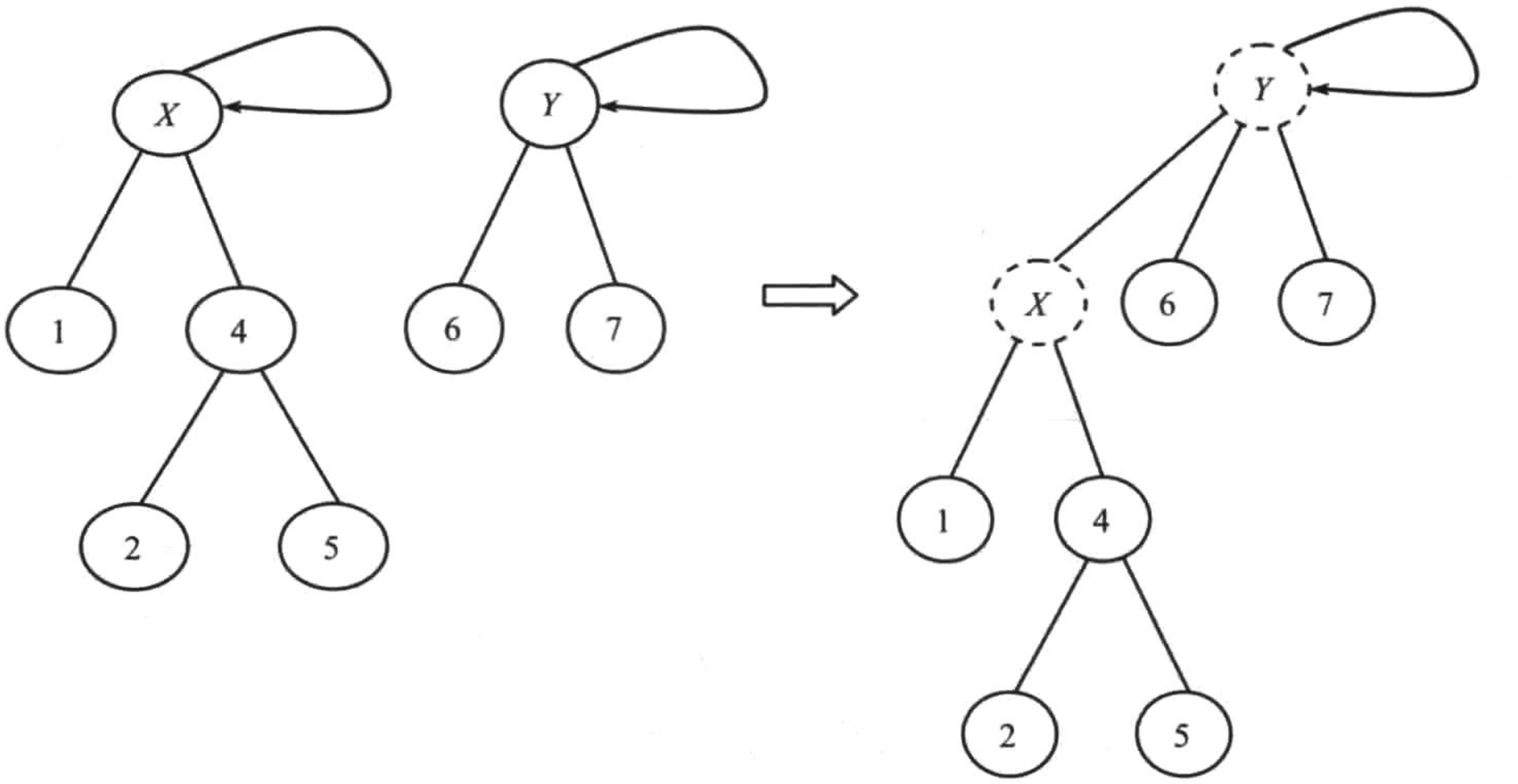
**如何表示**

一种方法是使用数组实现：对于每个元素，将根结点作为其集合的名称。但是用这种表示方法会碰到在 FIND 数组实现中的相同问题。为了解决此问题，不存储根结点，而是保存元素的双亲结点。因此，使用数组存储每个元素的双亲结点就可以解决问题。为了区分根结点，假定数组中根结点的双亲结点与其相同。基于这种表示方法，MAKESET、FIND、UNION 操作可以定义如下：

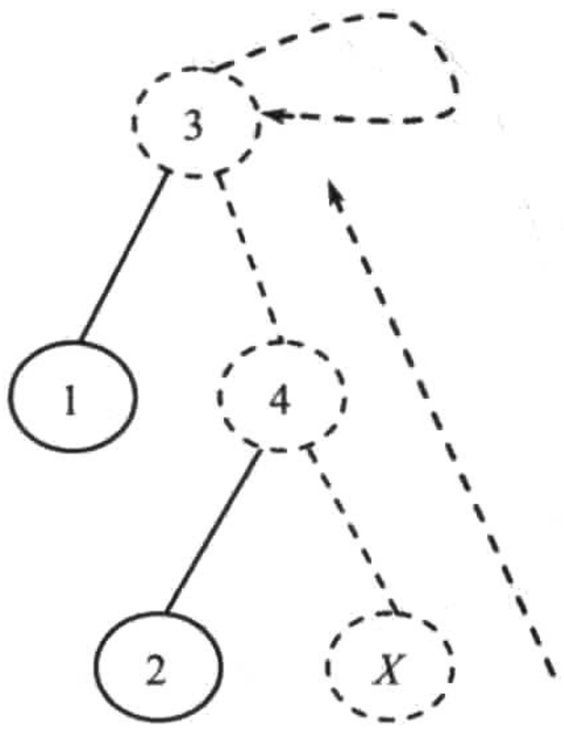
- MAKESET($X$)：创建一个新集合，它只包含一个元素 $X$，并且在数组中更新 $X$ 的双亲结点为 $X$。这就意味着 $X$ 的根结点是 $X$(集合名)。



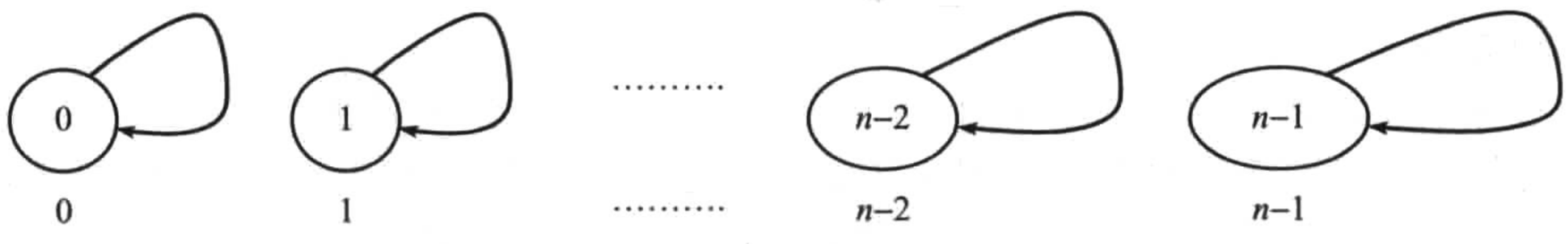

- UNION($X$，$Y$)：合并包含 $X$ 和 $Y$ 的两个集合，用合并后的集合替换这两个集合，并在此数组中将 $X$ 的双亲结点更新为 $Y$。




- FIND($X$)：返回元素 $X$ 所在的集合名。持续查找 $X$ 的集合名直至到达树的根结点。



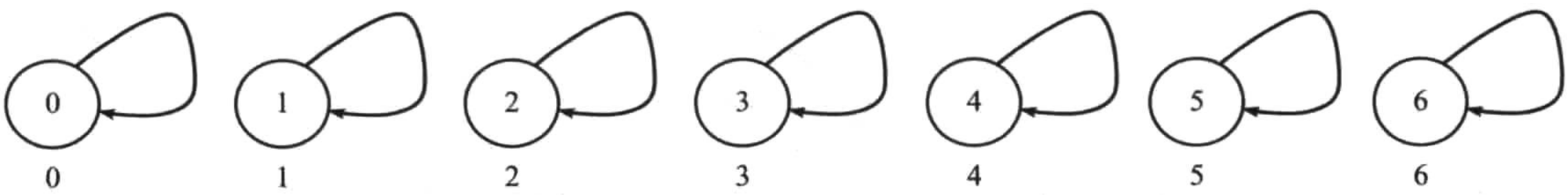

对于元素 0～$n-1$，初始表示为：




双亲数组

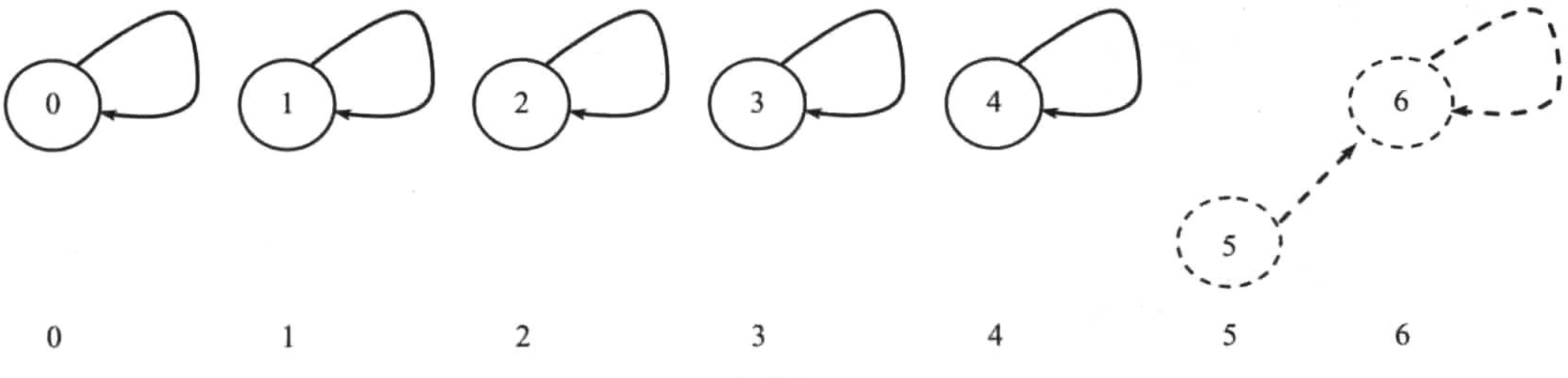
为了对两个集合执行 UNION 操作，将一棵树的根结点指向另一棵树的根结点来合并这两棵树。

对于元素 0～6 的初始设置为：



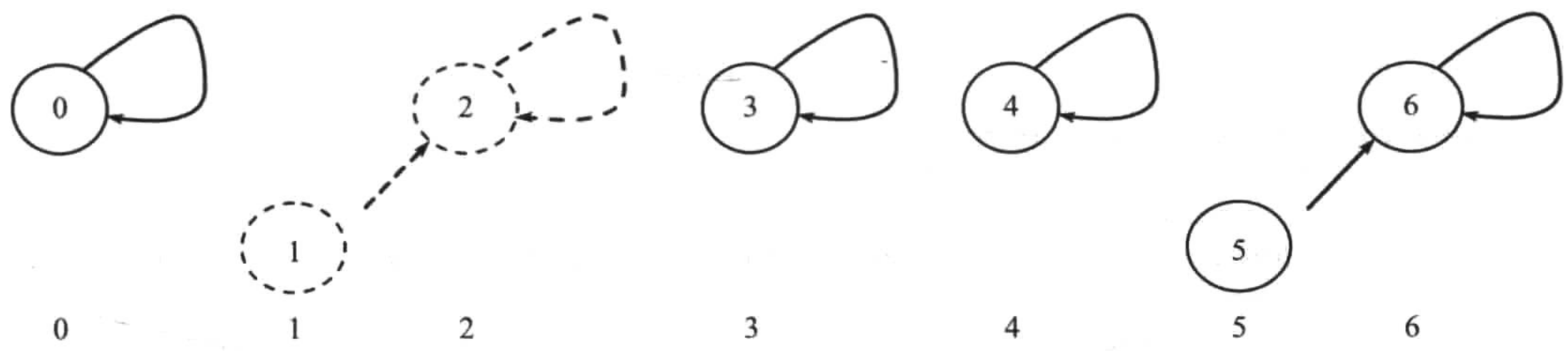

双亲数组

在执行完 UNION(5，6)后，



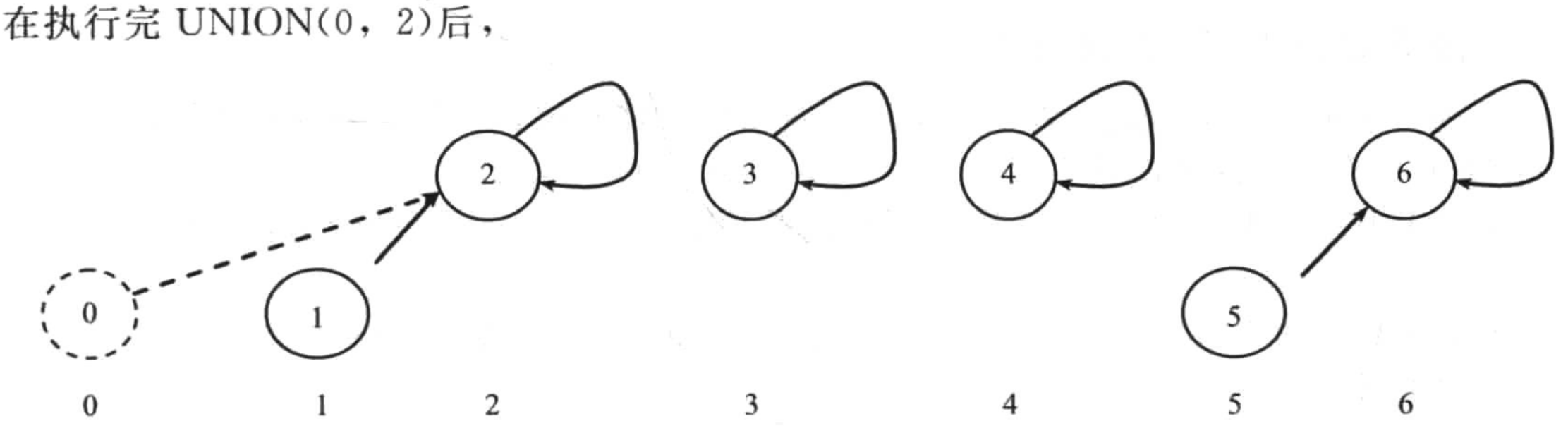

双亲数组

在执行完 UNION(1，2)后，




双亲数组

在执行完 UNION(0，2)后，




双亲数组

这里的一个要点是，UNION 操作只改变根结点的双亲结点而不改变集合中其他元素的双亲节点。由此，UNION 操作的时间复杂度为 $O(1)$。FIND($X$)操作将返回包含 $X$ 的树的根结点，执行此操作的时间复杂度与 $X$ 在该树中的深度成正比。使用这个方法，可能产生深度为 $n-1$ 的树(斜树)，最坏情况下，FIND 操作的运行时间是 $O(n)$，$m$ 个连续的 FIND 操作需要 $O(mn)$。

```
public class  DisjointSet {
        public int[] S;
        public int size;                                    // 集合中元素的个数
        public MAKESET(int size)                            { //具体如下所示 }
        public int FIND(int X)                              { //具体如下所示 }
        public int UNION(int root1, int root2)              {//具体如下所示}
        ……
}
```

MAKESET

```
public void MAKESET(int size){
        S = new int[size];
        for(int i = size-1; i >=0; i-- )
            S[i] = i;
}
```

FIND

```
public int FIND(int X){
        if(! (X >= 0 && X < size)) )
            return;
        if( S[X] == X )
            return X;
        else
            return FIND(S, S[X]);
}
```

UNION

```
public void UNION(int root1, int root2){
        if(FIND(root1) == FIND(root2))
            return;
        if(! ((root1 >= 0 && root1 < size) && (root1 >= 0 && root1 < size) ) )
            return;
        S[root1] = root2;
}
```

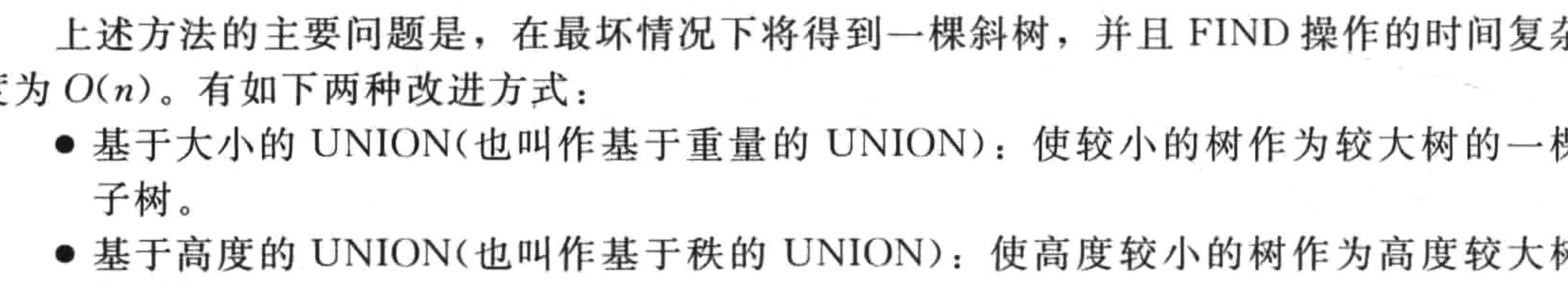
## 8.7 快速 UNION 实现(快速 FIND)

上述方法的主要问题是，在最坏情况下将得到一棵斜树，并且 FIND 操作的时间复杂度为 $O(n)$。有如下两种改进方式：

- 基于大小的 UNION(也叫作基于重量的 UNION)：使较小的树作为较大树的一棵子树。
- 基于高度的 UNION(也叫作基于秩的 UNION)：使高度较小的树作为高度较大树的子树。

### 1. 基于大小的 UNION

在前面的表示中，对于每个元素 $i$，若该元素是根元素，则存储 $i$(在双亲结点数组中)，若 $i$ 是其他元素，则存储 $i$ 的双亲结点。但是在本方法中，存储树的大小的负值(即，如果树的大小为 3，则根结点元素需要在双亲结点数组中存储－3)。对于前面的例子(在 UNION(0，2)后)，新的表示如下：




双亲数组

假定包含一个元素的集合大小为 1，且存储为－1。其他的没有改变。

MAKESET

```
public void MAKESET(int size) {
        for(int i = size-1; i >= 0; i-- )
            S[i] = -1;
}
```

FIND

```
public int FIND(int X) {
        if(! (X >= 0 && X < size)) )
                return;
        if( S[X] == -1 )
                return X;
        else        return FIND(S, S[X]);
}
```

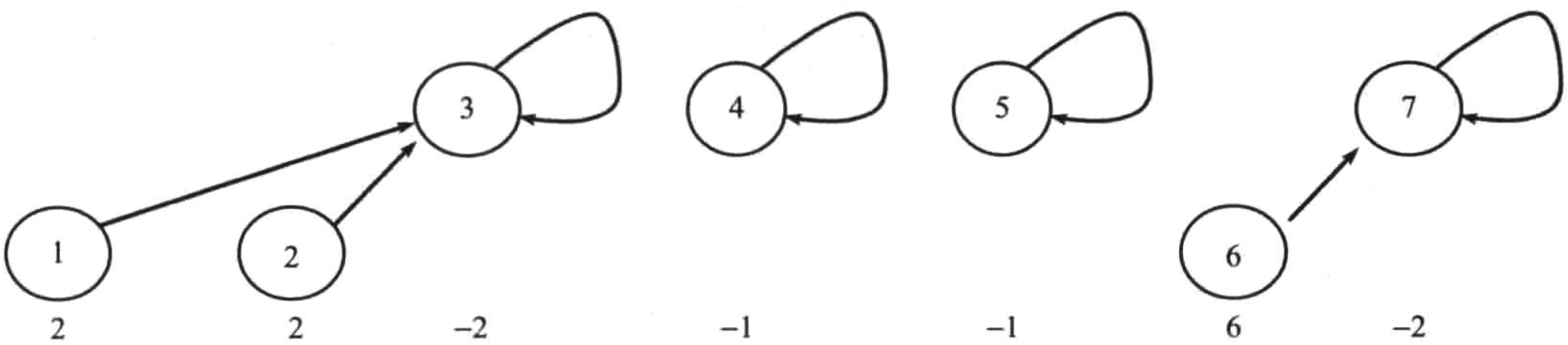
基于大小的 UNION

```
public void UNIONBySize(int root1, int root2) {
        if(FIND(root1) == FIND(root2))
                return;
        if( S[root2] < S[root1] )    {
                S[root1] = root2;
                S[root2] += S[root1];
        }
        else {
                S[root2] = root1;
                S[root1] += S[root2];
        }
}
```

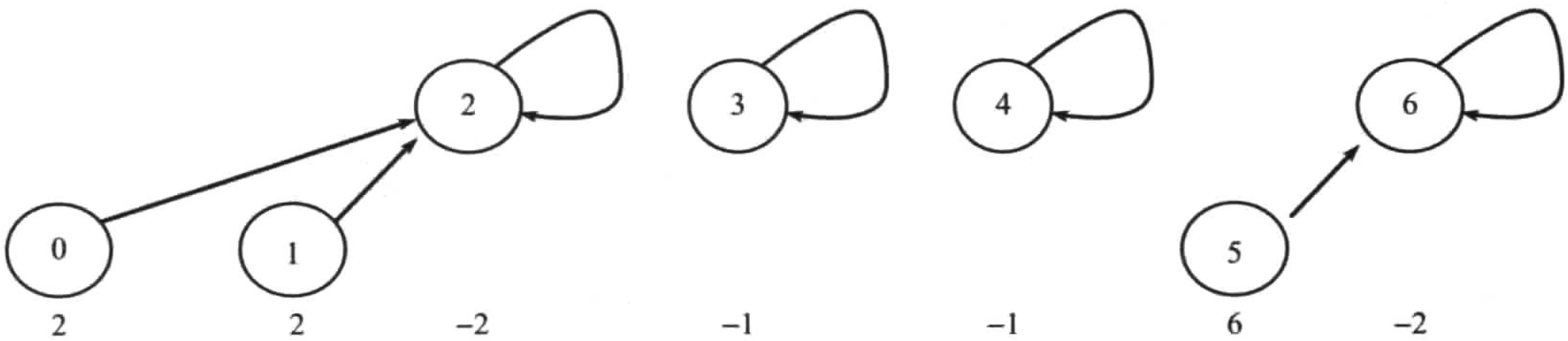
**注意**：FIND 操作实现没有改变。

**2. 基于高度的 UNION（基于秩的 UNION）**

与基于大小的 UNION 类似，本方法存储树的高度的负值(即，如果树的高度为 3，则根结点元素存储－3 到双亲结点数组中)。假定只有一个元素的树的高度为 1。对于上面的例子(在 UNION(0，2)后)，新的表示如下：




双亲数组




双亲数组

UNION by Height

```
public void UNIONByHeight(int root1, int root2) {
        if(FIND(root1) == FIND(root2))
                return;
        if( S[root2] < S[root1] )
                S[root1] = root2;
        else {
                if( S[root2] == S[root1] )
                        S[root1]--;
                S[root2] = root1;
        }
}
```

**注意：**对于 FIND 操作，在实现中没有变化。

**3. 比较基于大小的 UNION 和基于高度的 UNION**

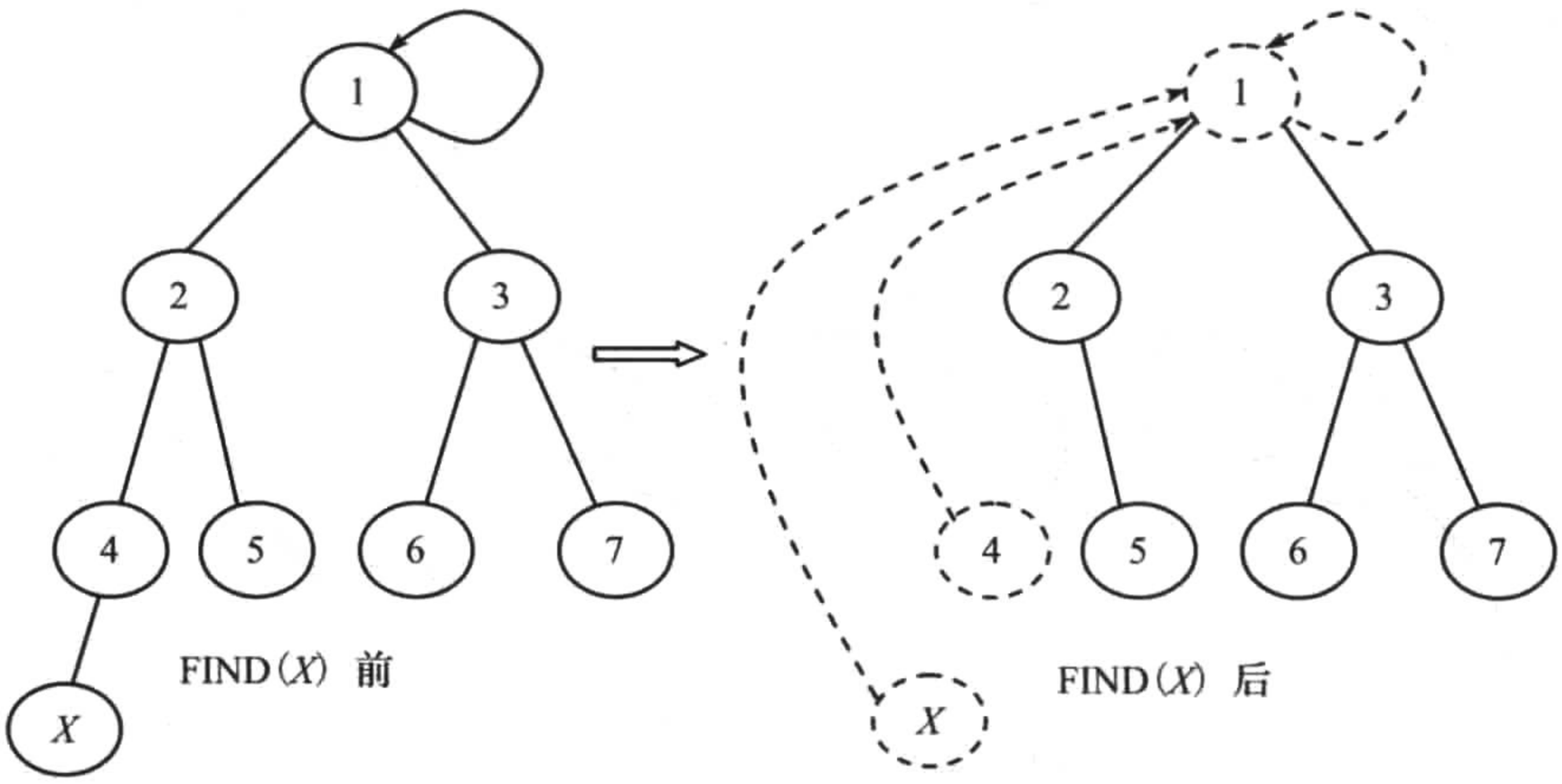
使用基于大小的 UNION，任意结点的高度永远不会大于 log$n$。这是因为一个结点在初始化时高度为 0。当由于 UNION 操作使其高度增加时，它被放置在至少是原来 2 倍大小的树中。即它的高度最多是以 log$n$ 倍增加的。这意味着 FIND 操作的运行时间为 $O(\log n)$，$m$ 次连续执行该操作需要 $O(m\log n)$时间。

基于高度的 UNION 也与上面的操作类似，如果对两棵相同高度的树进行 UNION 操作，UNION 后树的高度比之前树的高度增加 1，否则就等于两棵树高度最大的那一个。这就使得 $n$ 个结点的树的高度增长的倍数大于 $O(\log n)$，$m$ 次连续执行 UNION 操作操作和 FIND 操作仍然需要 $O(m\log n)$的时间。

## 8.8 路径压缩

FIND 操作遍历从当前结点到根结点路径上的一系列结点。通过将这些结点的每个父指针直接指向根结点，可以使后面的 FIND 操作更高效。这个过程叫作路径压缩。例如，在执行 FIND($X$)操作时，遍历从 $X$ 到树的根结点的路径上的各个结点。有效的路径压缩是将该路径上的每个结点的双亲结点直接变为树的根结点。

对 FIND 函数，使用路径压缩的唯一改变是 $S[X]$的值等于 FIND 函数的返回值。通过递归地寻找该结点集合的根结点，然后令 $X$ 直接指向根结点。该过程递归地应用于到根结点路径上的每个结点。



使用路径压缩的 FIND：

```
public int FIND(int X) {
        if(! (X >= 0 && X < size)) )
                return;
        if( S[X] <= 0 )
                return X;
        else    return( S[X] = FIND(S[X] ) );
}
```

**注意**：路径压缩与基于大小的 UNION 兼容，但与基于高度的 UNION 不兼容，因为没有有效的方法来改变树的高度。

## 8.9　小结

在包含 $n$ 个对象的集合中执行 $m$ 次的 UNION-FIND 操作，最坏情况下的时间复杂度如下表所示。

| 算法 | 最坏情况时间 |
|---|---|
| 快速 FIND | $mn$ |
| 快速 UNION | $mn$ |
| 基于大小/高度的 UNION | $n+m\log n$ |
| 路径压缩 | $n+m\log n$ |
| 基于大小/高度的快速 UNION＋路径压缩 | $(n+m)\log n$ |

## 8.10　并查集的相关问题

**问题 1**　考虑一系列城市 $c_1$，$c_2$，…，$c_n$。假设有关系 $R$，对于任意 $i$，$j$，如果城市 $c_i$ 和 $c_j$ 在同一个州，则 $R(c_i, c_j)$为 1，否则为 0。如果 $R$ 用表存储，那么它需要的空间是多少？

**解答**：任意两个城市的关系 $R$ 都需要在表中有空间与之对应。这需要 $\Theta(n^2)$空间复杂度。

**问题 2**　对于问题 1，使用并查集 ADT，给出算法将每个城市放在一个集合中，使得当且仅当 $c_i$ 和 $c_j$ 处于同一州时，它们才属于相同的集合。

**解答**：

```
for (int i = 1; i<= n; i++) {
        MAKESET(ci);
        for (int j = 1; j <= i-1; j++) {
                if(R(cj, ci)) {
                        UNION(cj, ci);
                        break;
                }
        }
}
```

**问题 3**　对问题 1，当城市已存储在并查集 ADT 中时，如果给定两个城市 $c_i$ 和 $c_j$，如何判断它们是否处于同一州？

**解答**：当且仅当 FIND($c_i$)＝FIND($c_j$)，城市 $c_i$ 和 $c_j$ 才处于同一州。

**问题 4**　对问题 1，如果使用基于大小的 UNION 的链表来实现 UNION-FINDADT，需要多少空间来存储城市？

**解答：**每一个城市都有一个结点，所以空间复杂度为 $\Theta(n)$。

**问题 5** 对于问题 1，如果使用基于秩的 UNION 的树结构，问题 2 算法在最坏情况下的运行时间是多少？

**解答：**当采用问题 2 的算法执行 UNION 操作时，第二个参数是大小为 1 的树。因此，所有树的高度为 1。所以每次 UNION 的时间为 $O(1)$。所以最坏情况下的运行时间是 $\Theta(n^2)$。

**问题 6** 如果使用树，但不使用基于秩的 UNION。问题 2 的算法在最坏情况下的运行时间是多少。是否有比问题 5 更差的情况？

**解答：**由于 UNION 的特定情况，基于秩的 UNION 并没有对算法产生影响，所以所有情况都与问题 5 一样。

# 第9章 Chapter 9

# 图 算 法

## 9.1 引言

在现实世界中，许多问题是由对象以及它们之间的联系所描述的。例如，在航空地图中，我们可能对这样的问题感兴趣："从海德拉巴去纽约，哪种方式最快?"或者"哪种方式价格最便宜?"为了回答这些问题，需要关于对象(城镇)之间的联系(飞行路线)信息。图就是用来解决这类问题的数据结构。

## 9.2 术语

**图**：一个图可表示为$(V, E)$，其中$V$是结点的集合，称为顶点；$E$是顶点对的集合，称为边。

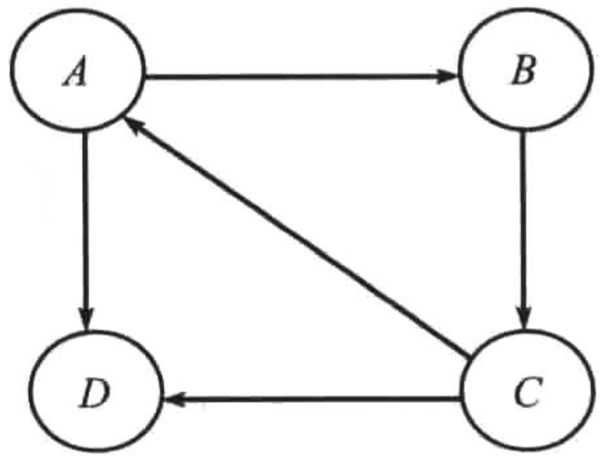
- 顶点和边代表位置和存储元素。
- 需要用到如下的定义：
  - 有向边
    - 有序顶点对$(u, v)$。
    - 第一个顶点$u$是源点。
    - 第二个顶点$v$是终点。
    - 例子：单行道。



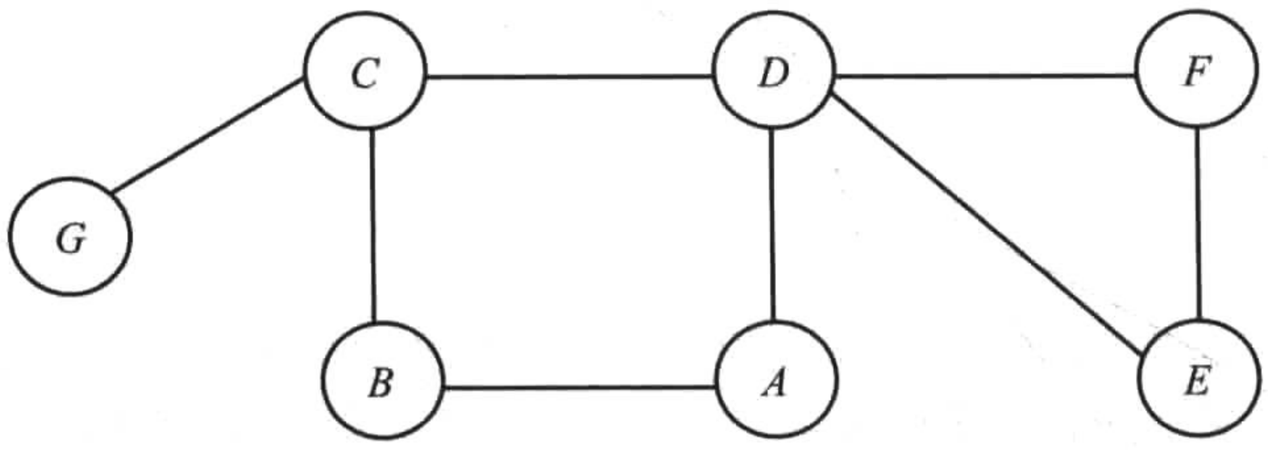

  - 无向边
    - 无序顶点对$(u, v)$。
    - 例子：铁路线。



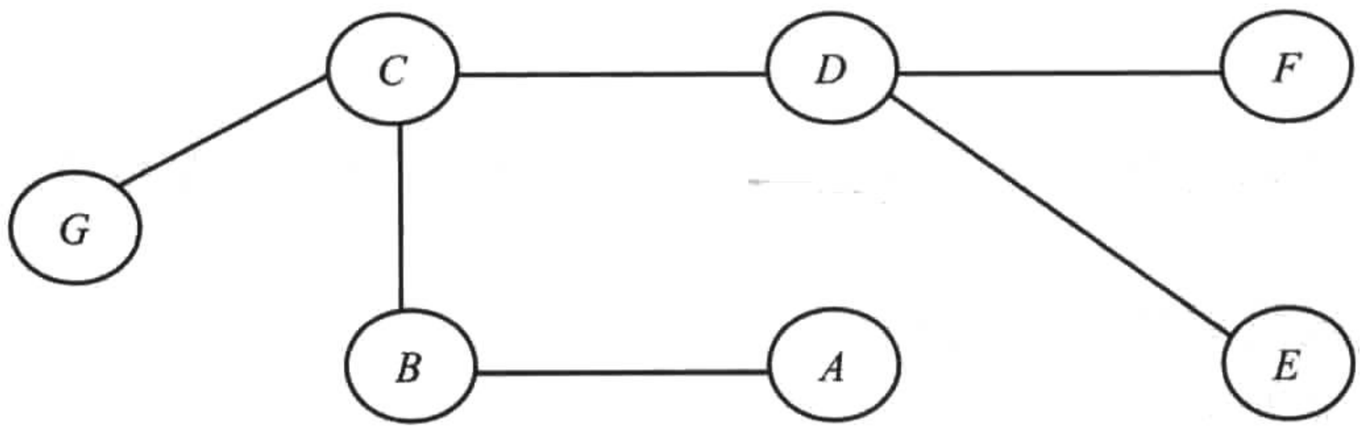

- 有向图
  - 所有的边都是有向边。
  - 例子：路由网络。



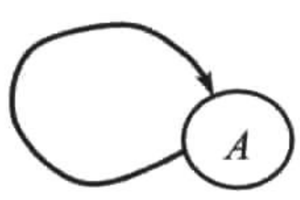

- 无向图
  - 所有的边都是无向边。
  - 例子：飞行网络。




- 当一条边连接两个顶点时，称这两个顶点为相互邻近，且该边关联这两个顶点。
- 无环图称为树，树是不包含环的连通图。




- 自环指的是一条连接顶点及其自身的边。

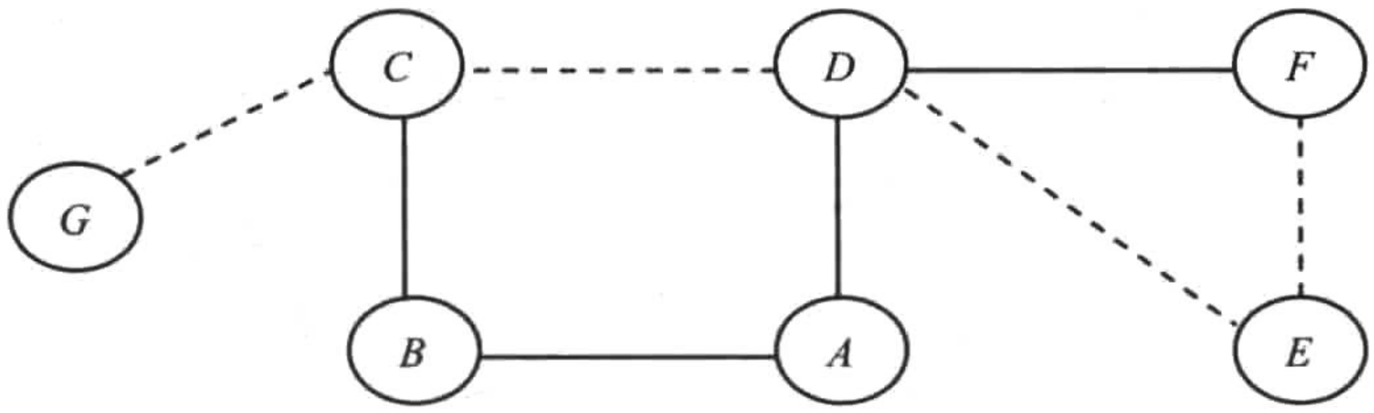



- 平行边指的是两条连接相同顶点对的边。

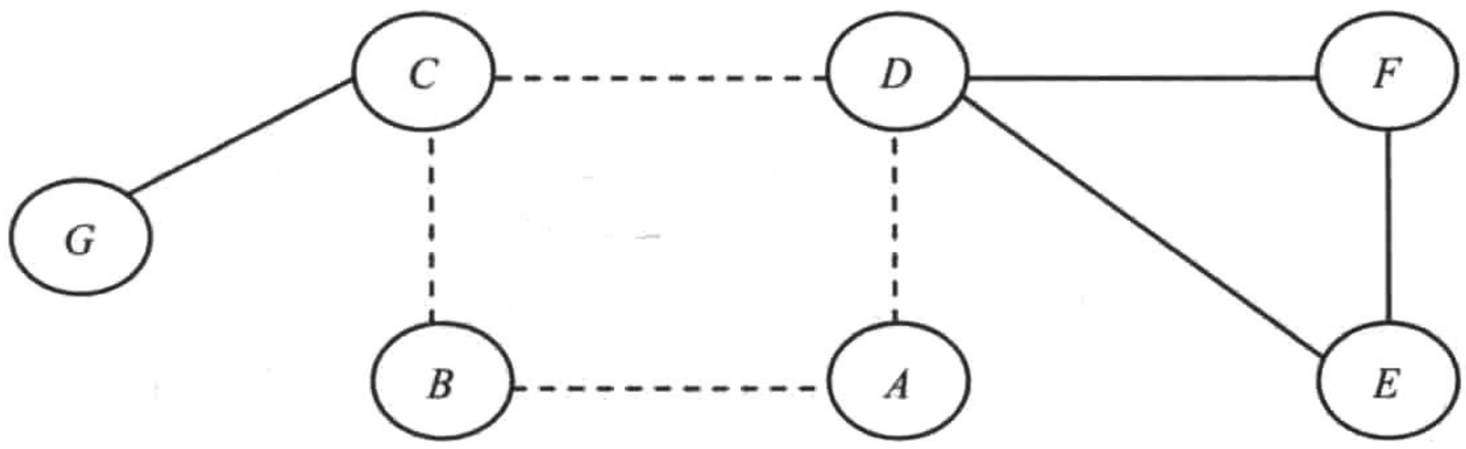



- 顶点的度是指关联该顶点的边的数目。
- 子图是图的边(及边所关联的顶点)的子集所形成的图。
- 图中的路径指的是一系列相邻顶点。简单路径是一条不包含重复顶点的路径。在下

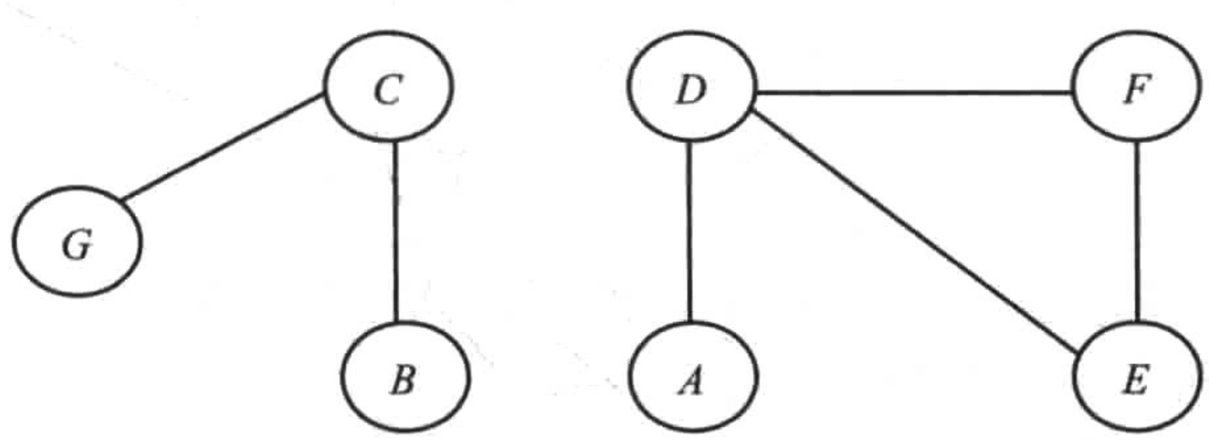
图中，虚线表示一条从 $G$ 到 $F$ 的路径。



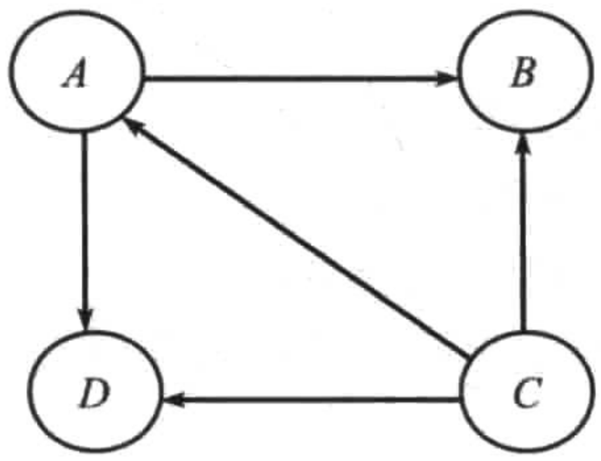

- 环路是起点与终点相同的路径。简单环路是不包含重复顶点和边的环(除了起点与终点外)。



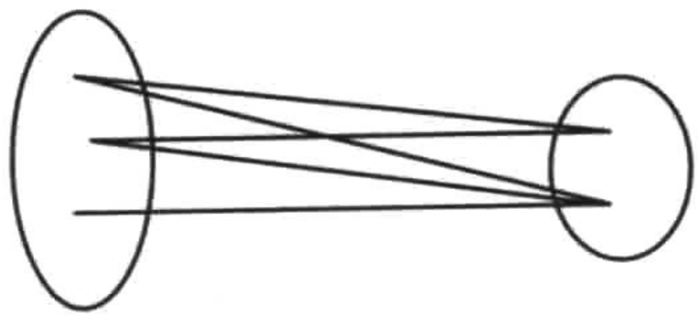

- 如果两个顶点之间存在一条路径，则称这两个顶点是连通的。
- 如果图中每对顶点之间都有路径相连，则称该图是连通图。
- 如果一个图是非连通的，那么它由一组连通分量组成。



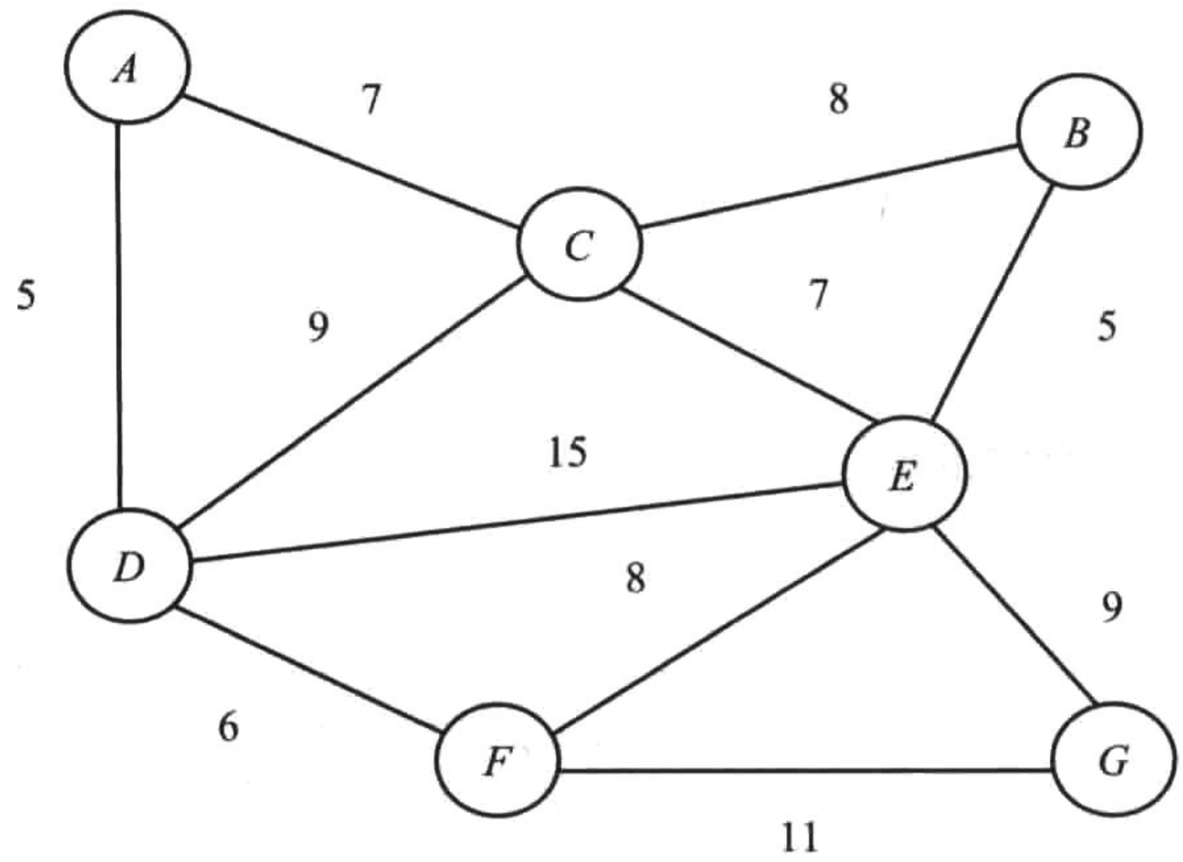

- 有向无环图是一种不包含环的有向图。



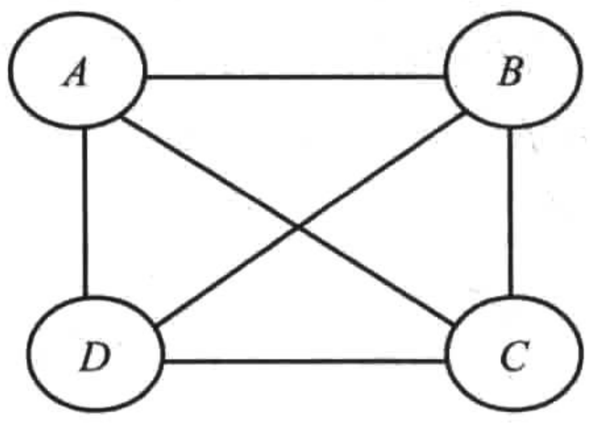

- 森林是一组不相交的树的集合。
- 连通图的生成树是一个包含所有图顶点的子图，并且是一棵单独的树。图的生成森林是连通分量的生成树的集合。
- 二分图是一个顶点能被分成两个集合的图，其中所有的边只连接来自不同集合的顶点。



- 在一个有权图中，给每条边赋值一个整数(权重)来代表(距离或花费)。




- $|V|$表示图中顶点的数目，$|E|$表示图中边的数目。在无向图中，$E$ 的取值范围是从 0～$|V|(|V|-1)/2$。这是因为在无向图中每个结点都能连接其他的结点。
- 包含所有边的图称为完全图。




- 仅包含少量边(当边数$<|V|\log|V|$)的图，称作稀疏图。
- 仅有少量边缺失的图，称作稠密图。
- 有向有权图有时称作网络。

## 9.3 图的应用

- 表示电子线路中组件之间的关系。
- 交通网络：高速网络、飞行网络。
- 计算机网络：局域网、因特网、万维网。
- 数据库：在数据库中用来表示 ER(实体-关系)图，为了表示数据库中数据表之间的依赖关系。

## 9.4 图的表示

与其他抽象数据类型相似，为了对图进行操作，需要以某种有用的形式来表示图。基本上，有如下两种表示形式：

- 邻接矩阵。
- 邻接表。

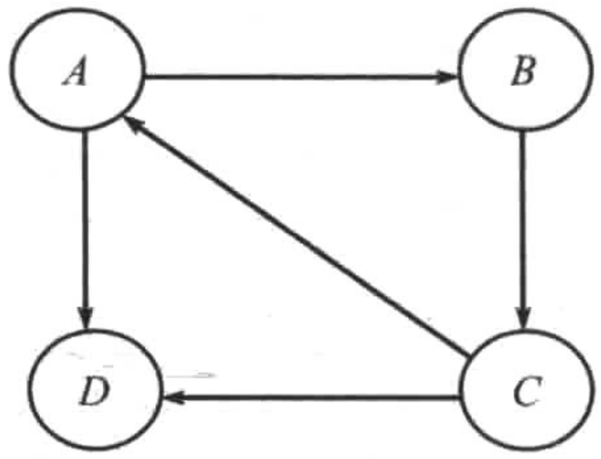
### 1. 邻接矩阵

**图的邻接矩阵表示方式**

首先需要了解图数据结构的组成部分。图的表示需要顶点数、边数以及它们之间的

连接关系。本方法采用一个大小为 $V \times V$ 的矩阵 Adj，其中矩阵的值为布尔值。如果存在一条从顶点 $u$ 到顶点 $v$ 的边，则设置 Adj[$u$，$v$]为 1，否则为 0。

在矩阵中，无向图的每条边可以用两个二进制数表示。即一条从 $u$ 到 $v$ 的边用 Adj[$u$，$v$]=1 和 Adj[$u$，$v$]=1 表示。为了节省时间，只需要处理该对称矩阵的上三角或下三角元素。此外，可以假设每个顶点有一条连接其自身的边。所以，对于所有顶点，Adj[$u$，$u$]都设置为 1。如果图是有向图，那么仅需要在邻接矩阵中标记一条边。考虑如下所示的有向图。




这个图的邻接矩阵可以表示为：

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 0 | 1 |
| B | 0 | 0 | 1 | 0 |
| C | 1 | 0 | 0 | 1 |
| D | 0 | 0 | 0 | 0 |

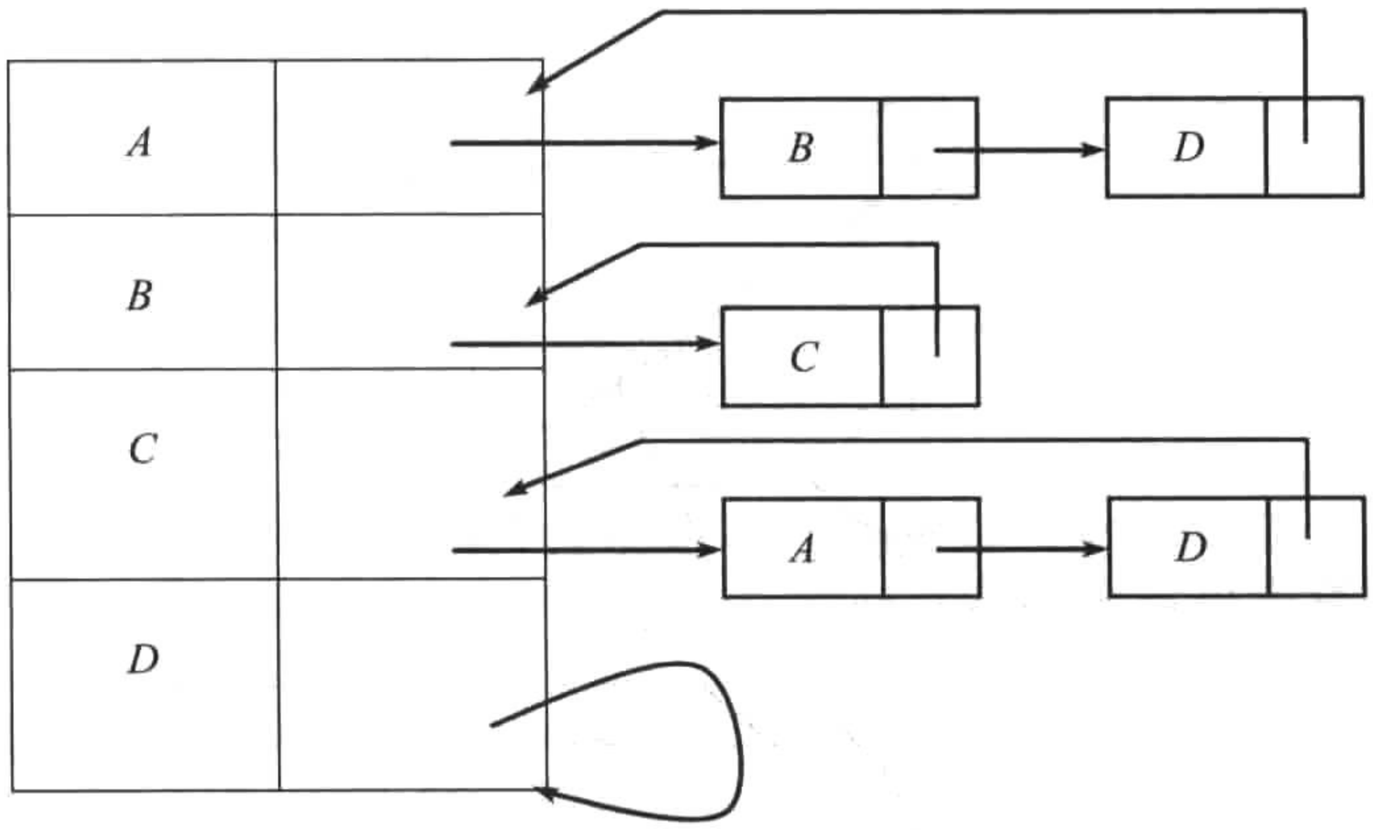
现在关注如何实现矩阵表示。为了读一个图，一种方式是先读顶点，然后再读顶点对(边)。读无向图的代码如下所示。

```
public class Graph {
        private boolean adjMatrix[][];
        private int vertexCount;
        public Graph(int vertexCount) {
                this.vertexCount = vertexCount;
                adjMatrix = new boolean[vertexCount][vertexCount];
        }
        public void addEdge(int i, int j) {
                if (i >= 0 && i < vertexCount && j > 0 && j < vertexCount) {
                        adjMatrix[i][j] = true;
                        adjMatrix[j][i] = true;
                }
        }
        public void removeEdge(int i, int j) {
                if (i >= 0 && i < vertexCount && j > 0 && j < vertexCount) {
                        adjMatrix[i][j] = false;
                        adjMatrix[j][i] = false;
                }
        }
        public boolean isEdge(int i, int j) {
                if (i >= 0 && i < vertexCount && j > 0 && j < vertexCount)
                        return adjMatrix[i][j];
                else    return false;
        }
}
```

当图是稠密图时，邻接矩阵是一种很好的表示方式。邻接矩阵需要 $O(V^2)$ 个存储位和 $O(V^2)$ 时间来初始化。如果边数与 $V^2$ 成正比，那么毫无疑问需要 $V^2$ 步来读这些边。但如果图是稀疏的，那么初始化矩阵仍然需要 $O(V^2)$ 时间，并且初始化过程占据了整个算法的大部分运行时间。

**2. 邻接表**

a）图的邻接表表示方式




在这种表示方式中，所有与某个顶点 $v$ 相连的顶点都在 $v$ 的邻接表中列出，采用链表很容易实现。也就是说，邻接表中的每一个顶点 $v$ 都有一个与其对应的链表，链表结点表示 $v$ 的邻接点与 $v$ 之间的连接。链表的总数等于图中的顶点数。以前面给出的邻接矩阵表示的图为例，邻接表的表示方式如下所示。由于顶点 $A$ 与 $B$ 和 $D$ 有边相连，所以将 $B$ 和 $D$ 添加到 $A$ 的邻接表中。其他顶点的邻接表也类似。

```
public class Graph {
        private ArrayList<Integer> vertices;
        private ListNode[] edges;
        private int vertexCount = 0;
        public Graph(int vertexCount) {
                this.vertexCount = vertexCount;
                vertices = new ArrayList<Integer>();
                edges = new ListNode[vertexCount];
                for (int i = 0; i < vertexCount; i++) {
                        vertices.add(i);
                        edges[i] = new ListNode ();
                }
        }
        public void addEdge(int source, int destination) {
                int i = vertices.indexOf(source);
                int j = vertices.indexOf(destination);
                if (i != -1 || j != -1) {
                        edges[i].insertAtBeginning(destination);
                        edges[j].insertAtBeginning(source);
                }
        }
}
```

对于邻接表的表示方式边的读入顺序是很重要的。这是因为边的顺序决定了顶点在

邻接表中的顺序。相同的图在邻接表中可以有许多不同的表示方式。边在邻接表中出现的顺序也会影响算法处理边的顺序。

b）邻接表的缺点

使用邻接表表示方式无法有效地完成某些操作。以删除某个结点为例。在邻接表中，如果直接从邻接表中删除该结点，是可以做到的。然而，在邻接表中当该结点和其他结点有边相连时，则必须搜索其他结点对应的链表来删除该结点。尽管可以通过在两个表结点之间建立一条特殊的边使得邻接表变为双向链表来解决此问题，但是处理这些额外的链接是有风险的。

### 3. 邻接集

类似于邻接表，但是采用并查集(Union-Find)来代替链表。详细内容请参见第 8 章。

### 4. 各种图表示的比较

有向图和无向图均采用相同的数据结构来表示。对于有向图，除了每条边仅被表示一次外(即从 $x$ 到 $y$ 的边在邻接矩阵中表示为 Adj[$x$][$y$]=1 或者将 $y$ 添加到 $x$ 对应的邻接表)，其他与无向图是相同的；对于有权图，除了采用权重代替布尔值来填充邻接矩阵外，其他也都是相同的。

| 表示方式 | 空间复杂度 | 判断 v 和 w 之间是否有边 | 遍历依附于 v 的边 |
|---|---|---|---|
| 边的列表 | $E$ | $E$ | $E$ |
| 邻接矩阵 | $V^2$ | 1 | $V$ |
| 邻接表 | $E+V$ | Degree($v$) | Degree($v$) |
| 邻接集 | $E+V$ | log(Degree($v$)) | Degree($v$) |

## 9.5 图的遍历

为了解决有关图的问题，需要一种机制来遍历图。图的遍历算法也叫作图搜索算法。与树遍历算法(中序、前序、后序以及层序遍历)一样，图搜索算法也可以看作从图的某个源点开始，通过遍历边和标记顶点来搜索图。下面讨论两种遍历图的算法：

- 深度优先搜索(DFS)。
- 广度优先搜索(BFS)。

### 1. 深度优先搜索

深度优先搜索(DFS)算法的原理类似于树的前序遍历，本质上也使用栈来实现。

以迷宫为例。假设一个人被困在一个迷宫里。为了走出迷宫，这个人需要访问每条路径和每一个十字路口(在最坏情况下)。假设此人使用两种颜色的涂料来标记已经经过的十字路口。当发现一个新的十字路口时，他将其标成灰色，并且继续往更深处走。当到达一个“末端”时，则表明从标记为灰色的十字路口出发的所有路径都已经被访问过，并将该十字路口标成黑色。因此，这个“末端”或者是一个已经被标成灰色或黑色的十字路口，或者就是一条不再有十字路口的路径。

迷宫的十字路口是图的顶点，而十字路口之间的路径就是图的边。从“末端”返回的过程叫作回溯。我们尝试从起始点开始尽可能深地访问图中的结点，直到必须回溯到先前的灰色顶点。在 DFS 算法中，包括如下类型的边。

| | |
|---|---|
| 树边：遇到一个新顶点的边 | 前向边：从祖先到子孙的边 |
| 回退边：从子孙到祖先的边 | 交叉边：在一棵树或子树之间的边 |

对于大多数算法，用布尔值足以区分未访问过/访问过的结点(基于 3 种颜色的实现方法请参见 9.9 节)。这说明，对于某些问题需要 3 种颜色进行区分，但是就当前讨论的问题而言，两种颜色就够了。

false ——→ 顶点没被访问过

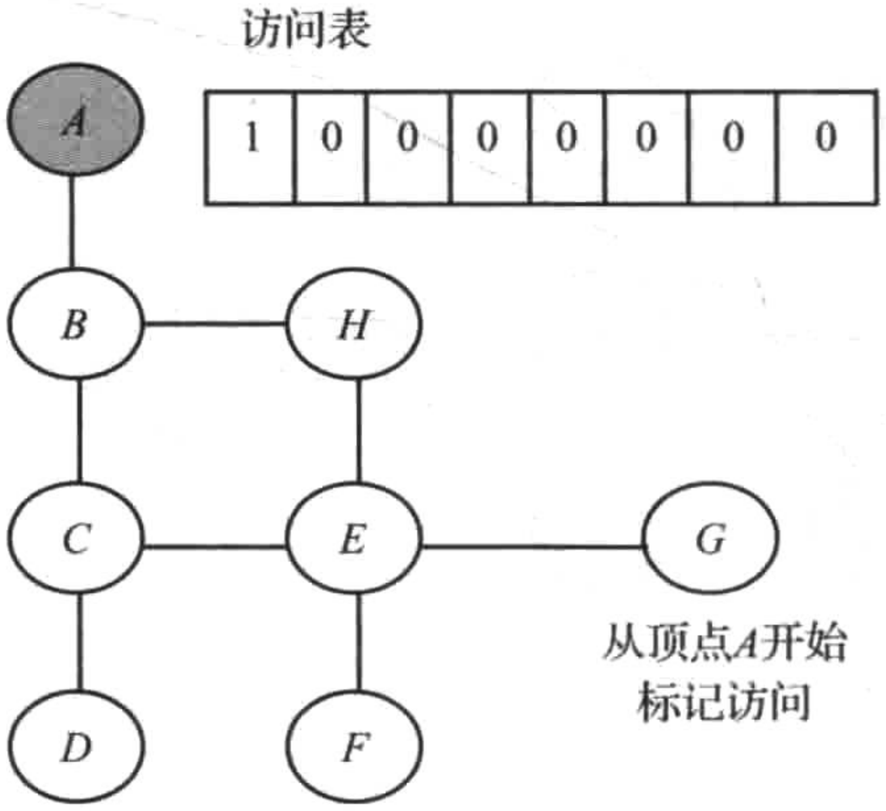
true ——→ 顶点被访问过

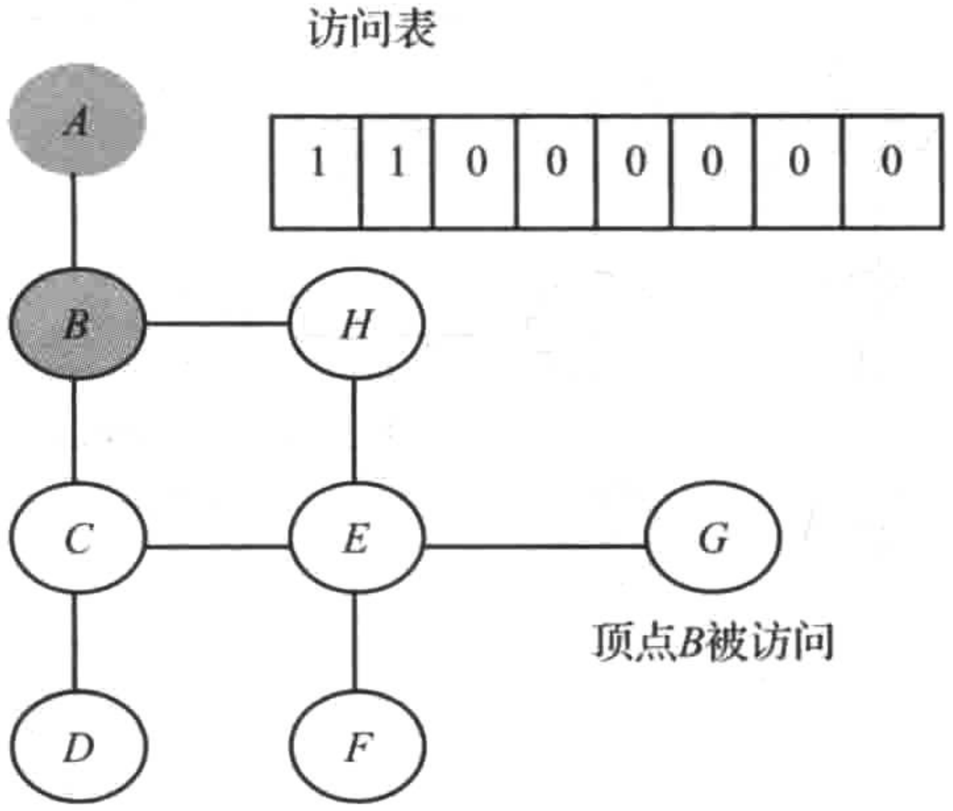
初始时所有顶点都被标记为未被访问过(false)。DFS 算法从图中一个顶点 $u$ 开始，首先考虑从 $u$ 到其他顶点的边。如果该边通往一个已经被访问过的顶点，那么回溯到当前顶点 $u$。如果该边通往一个未被访问过的顶点，则到达该顶点，并从该顶点开始进行访问，即将新的顶点变为当前顶点。重复这个过程直到算法到达“末端”。然后从在这个“末端”点开始回溯。当回溯到起始顶点时整个过程结束。下面给出基于这个策略的算法：假定 Visited[]是一个全局数组。

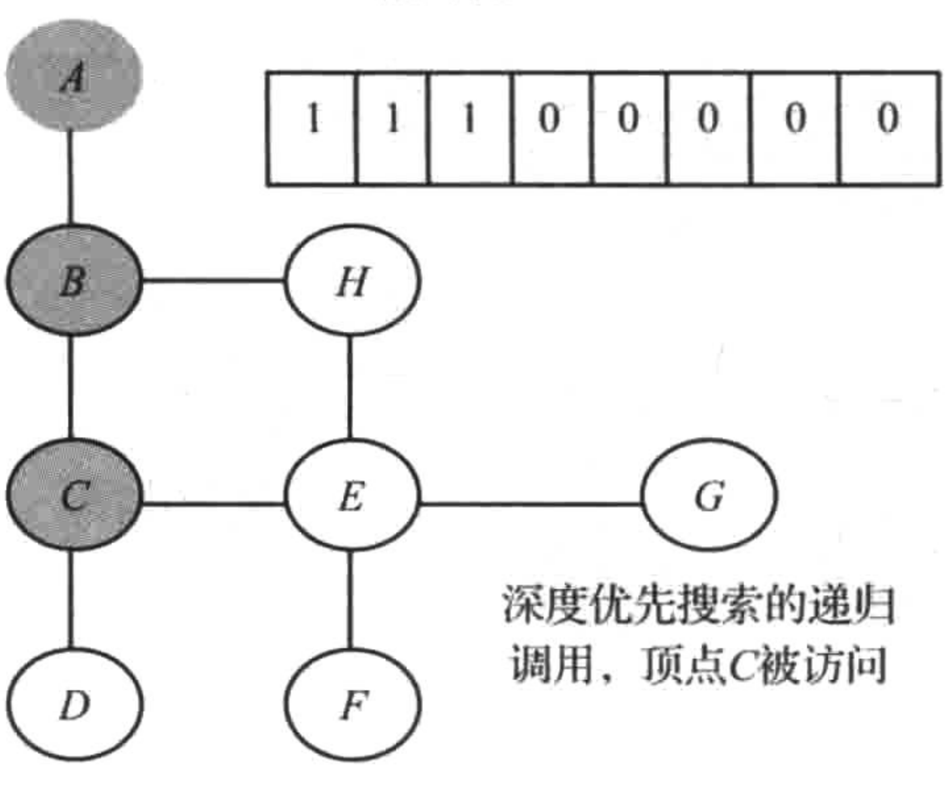
```
class Vertex {
        public char label;
        public boolean visited;
        public Vertex(char lab)   {
                label = lab;
                visited = false;
        }
}
class Graph {
        private final int maxVertices = 20;
        private Vertex vertexList[];
        private int adjMatrix[][];
        private int vertexCount;
        private Stack theStack;
        public Graph() {
                vertexList = new Vertex[maxVertices];
                adjMatrix = new int[maxVertices][maxVertices];
                vertexCount = 0;
                for(int y=0; y<maxVertices; y++)
                        for(int x=0; x<maxVertices; x++)
                                adjMatrix[x][y] = 0;
                theStack = new Stack();
        }
        public void addVertex(char lab) {
                vertexList[vertexCount++] = new Vertex(lab);
        }
        public void addEdge(int start, int end) {
                adjMatrix[start][end] = 1;
                adjMatrix[end][start] = 1;
        }
        public void displayVertex(int v) {
                System.out.print(vertexList[v].label);
        }
        public void dfs() {
                vertexList[0].visited = true;
                displayVertex(0);
                theStack.push(0);
                while( !theStack.isEmpty() )  {
                        // get an unvisited vertex adjacent to stack top
                        int v = getAdjUnvisitedVertex( theStack.peek() );
                        if(v == -1)
```

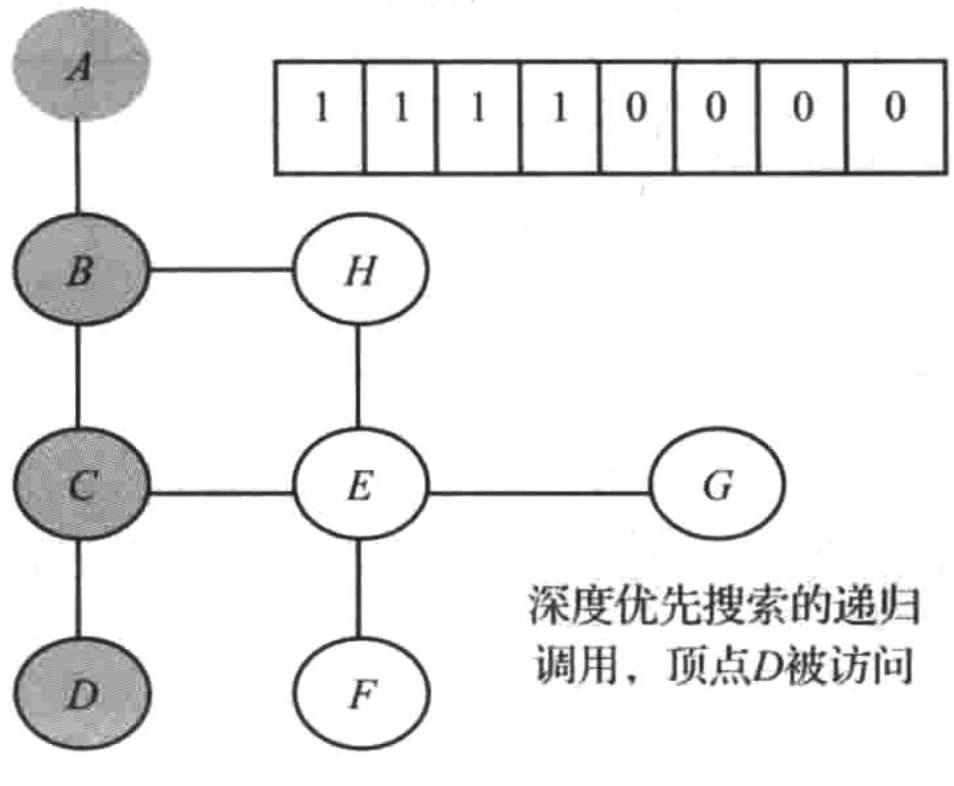
```
                    theStack.pop();
                else {
                    vertexList[v].visited = true;
                    displayVertex(v);
                    theStack.push(v);
                }
            }
            for(int j=0; j<vertexCount; j++)          // 初始化标记
                vertexList[j].visited = false;
    }
    public int getAdjUnvisitedVertex(int v) {
            for(int j=0; j<vertexCount; j++)
                if(adjMatrix[v][j]==1 && vertexList[j].visited==false)
                    return j;
            return -1;
        }
}
```

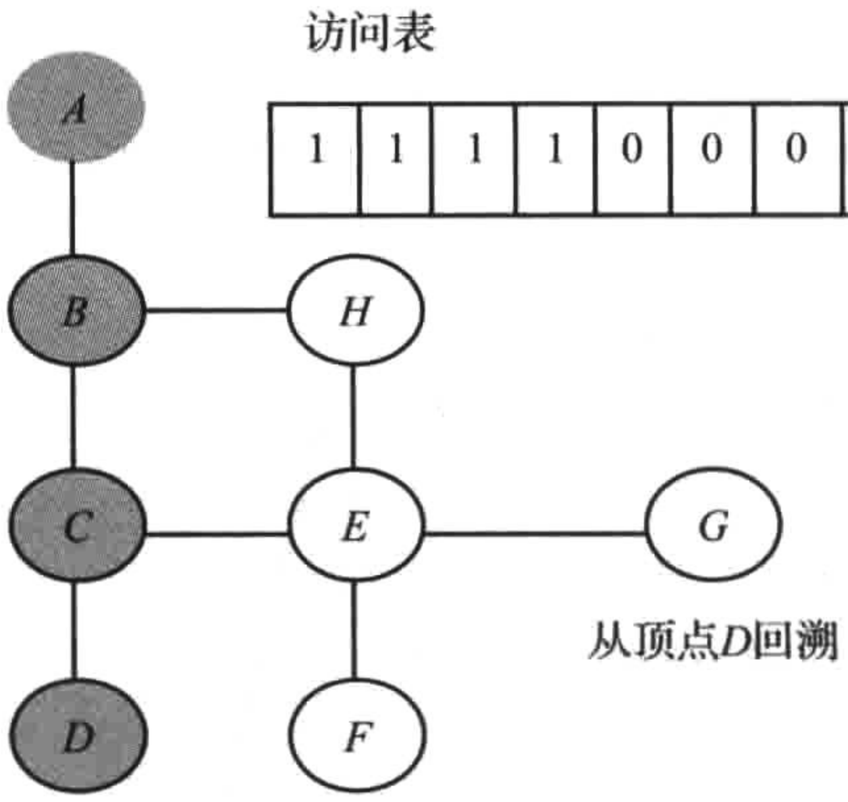
以下面的图为例。有时边会通往一个已经被访问过的顶点，这些边称为回退边，其余的边称为树边，因为从图中删除回退边会产生一棵树。最终产生的树称为 DFS 树，顶点的访问顺序称为顶点的 DFS 编号。在下图中，灰色表示该顶点被访问过。我们需要关注访问表何时被更新。

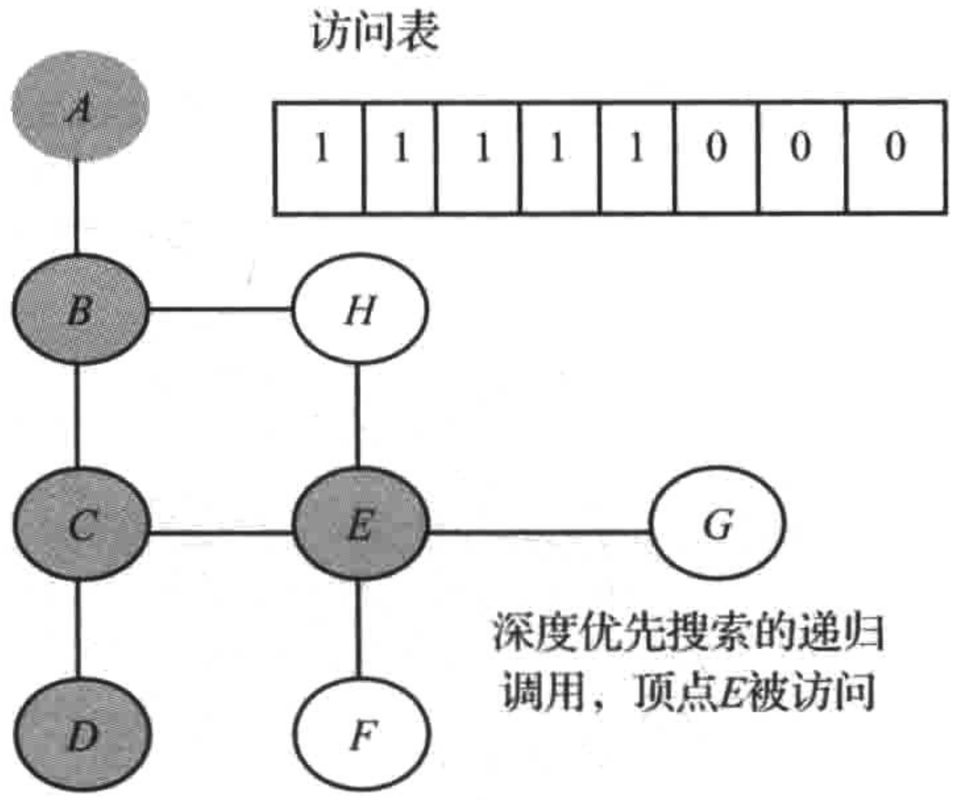


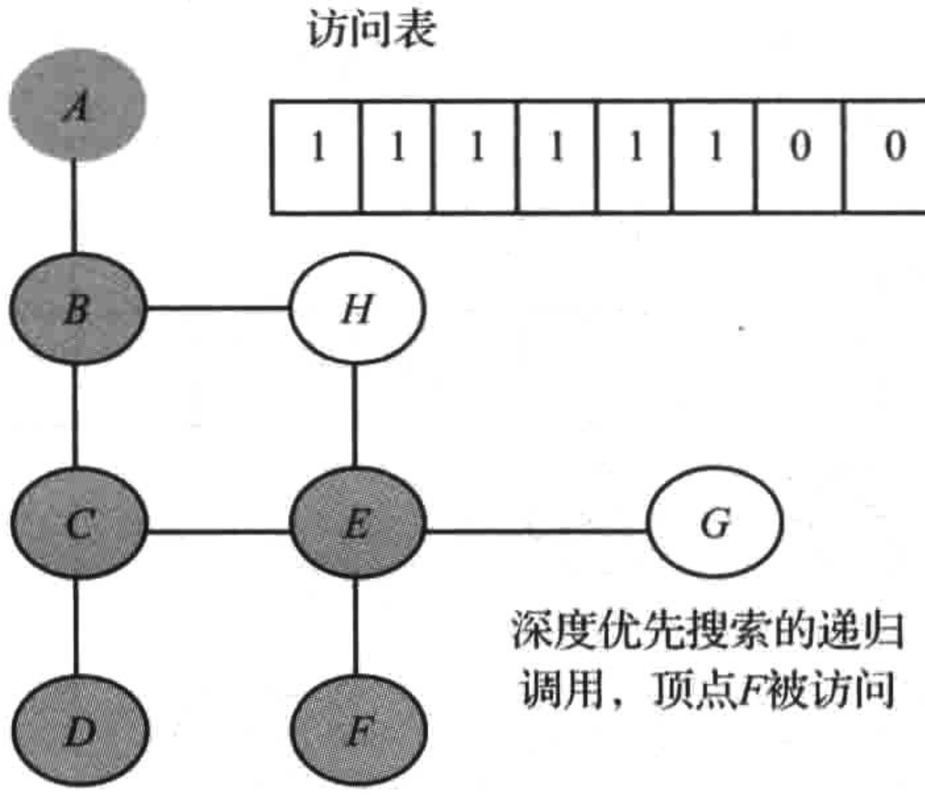



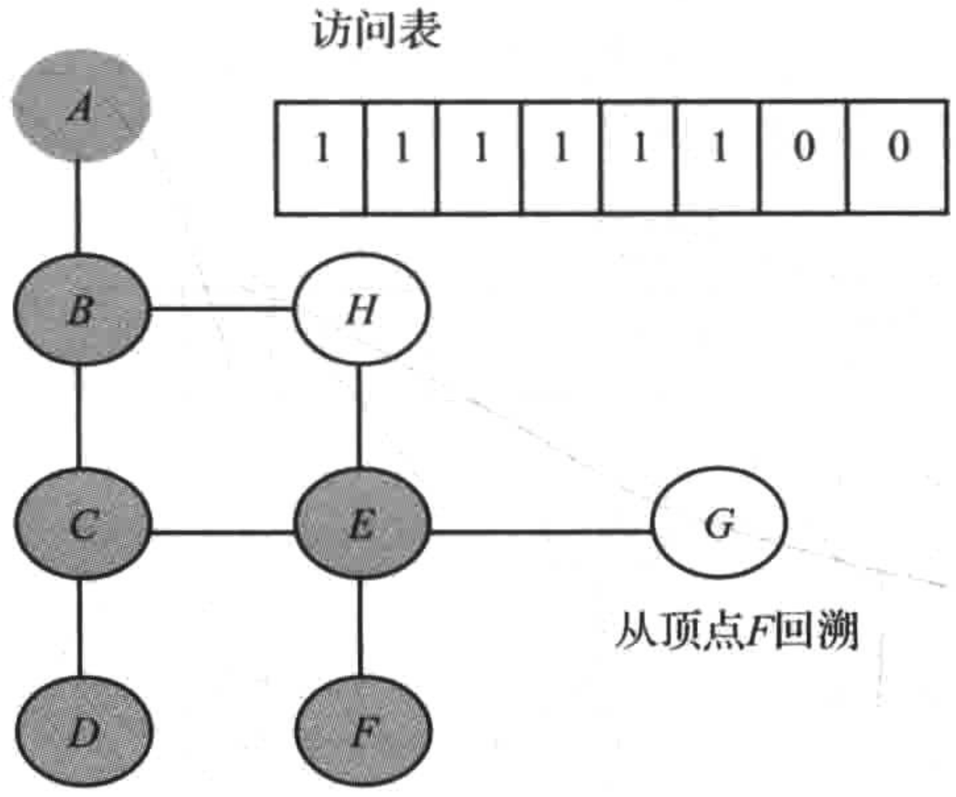



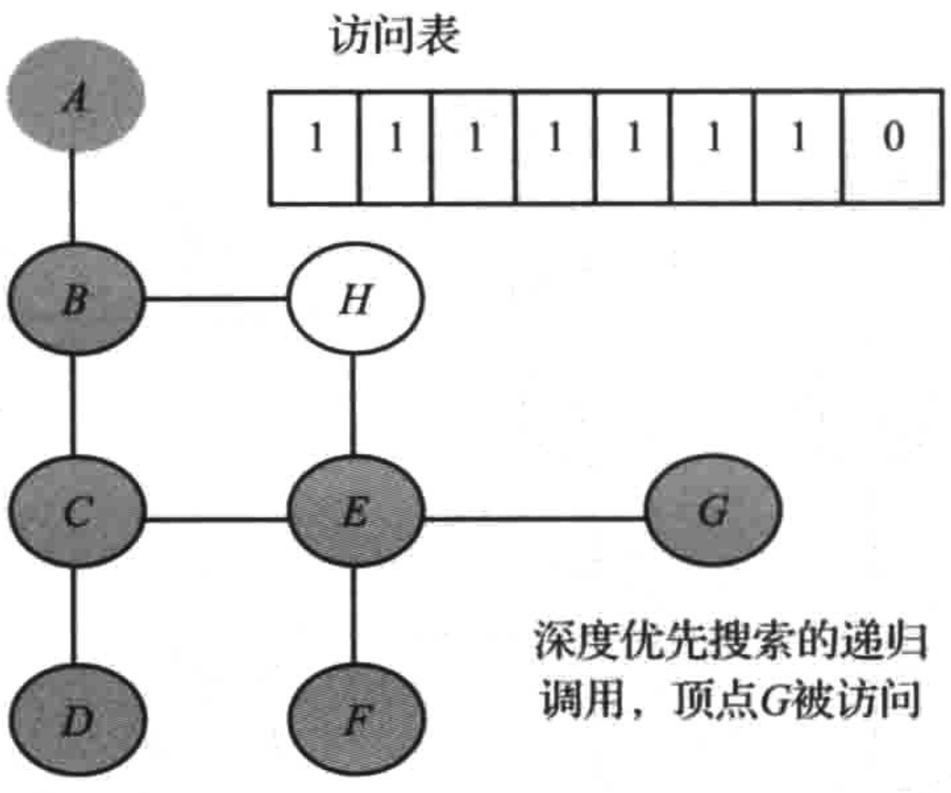



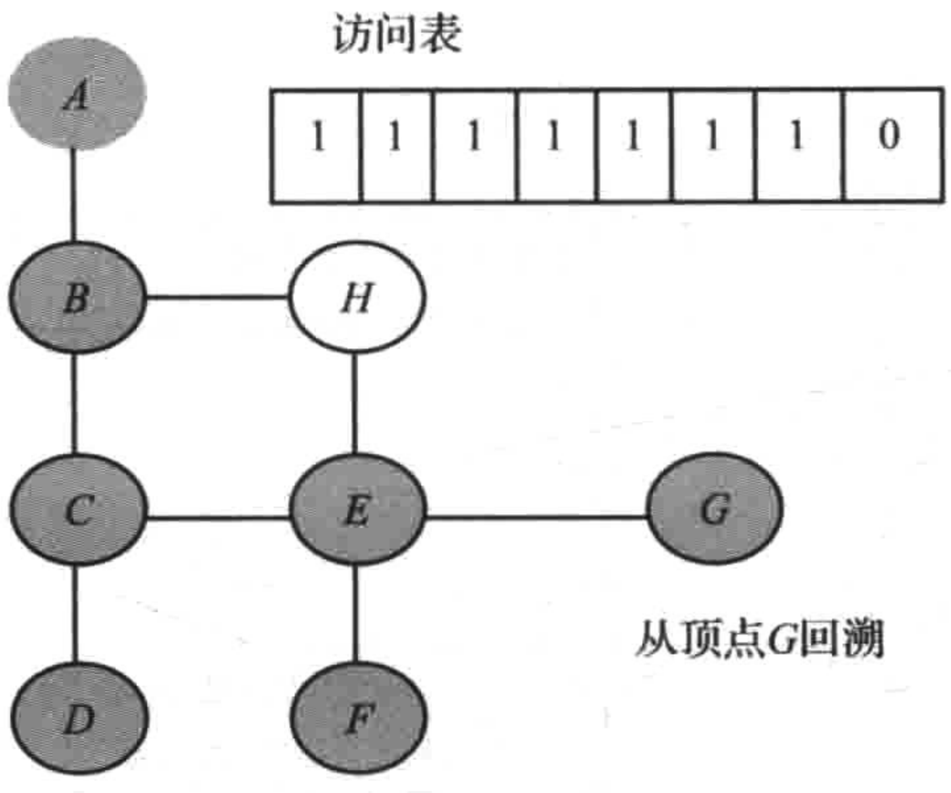

访问表
1 1 1 1 0 0 0 0
A
B
H
C
E
G
D
F
从顶点D回溯

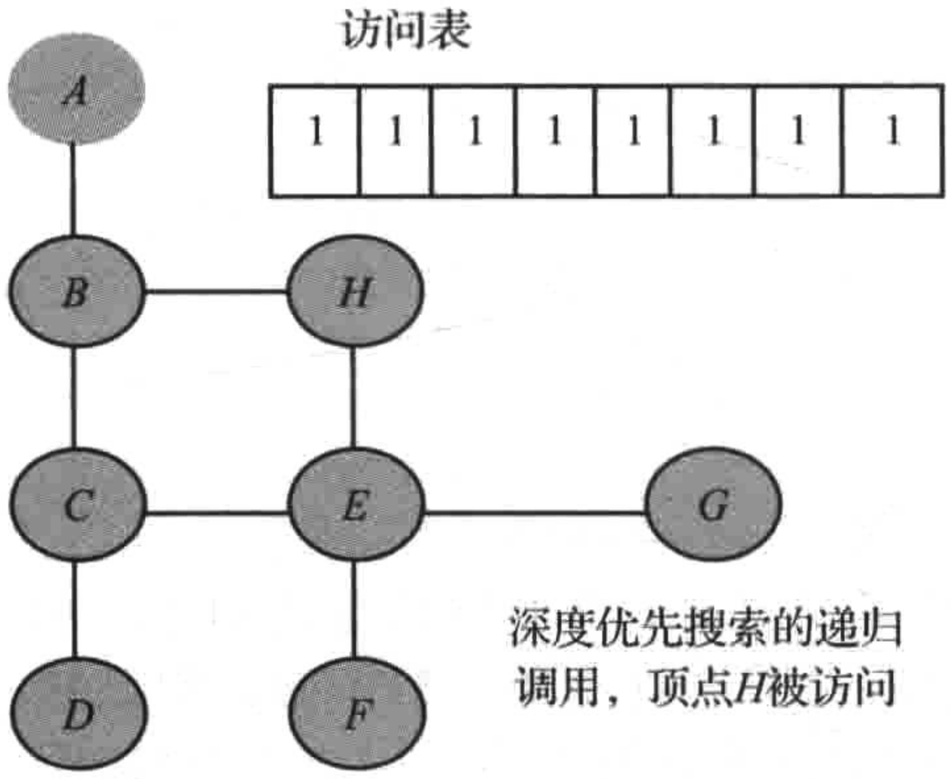

访问表
1 1 1 1 1 0 0 0
A
B
H
C
E
G
D
F
深度优先搜索的递归
调用，顶点E被访问


访问表
1 1 1 1 1 1 0 0
A
B
H
C
E
G
D
F
深度优先搜索的递归
调用，顶点F被访问

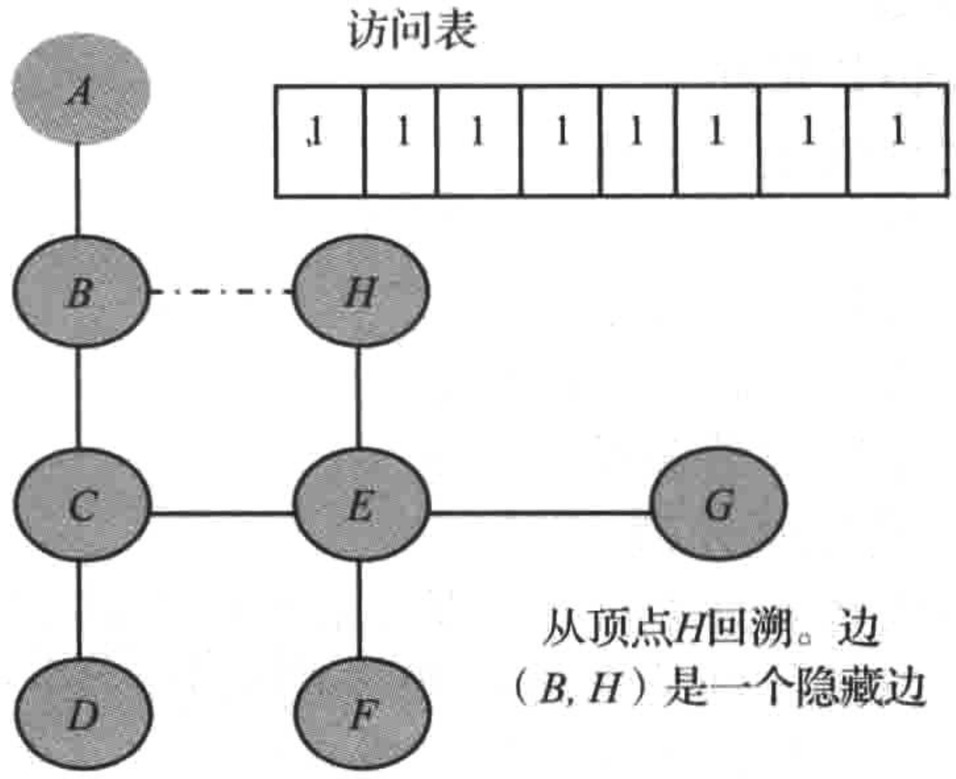

访问表
1 1 1 1 1 1 0 0
A
B
H
C
E
G
D
F
从顶点F回溯


访问表
1 1 1 1 1 1 1 0
A
B
H
C
E
G
D
F
深度优先搜索的递归
调用，顶点G被访问

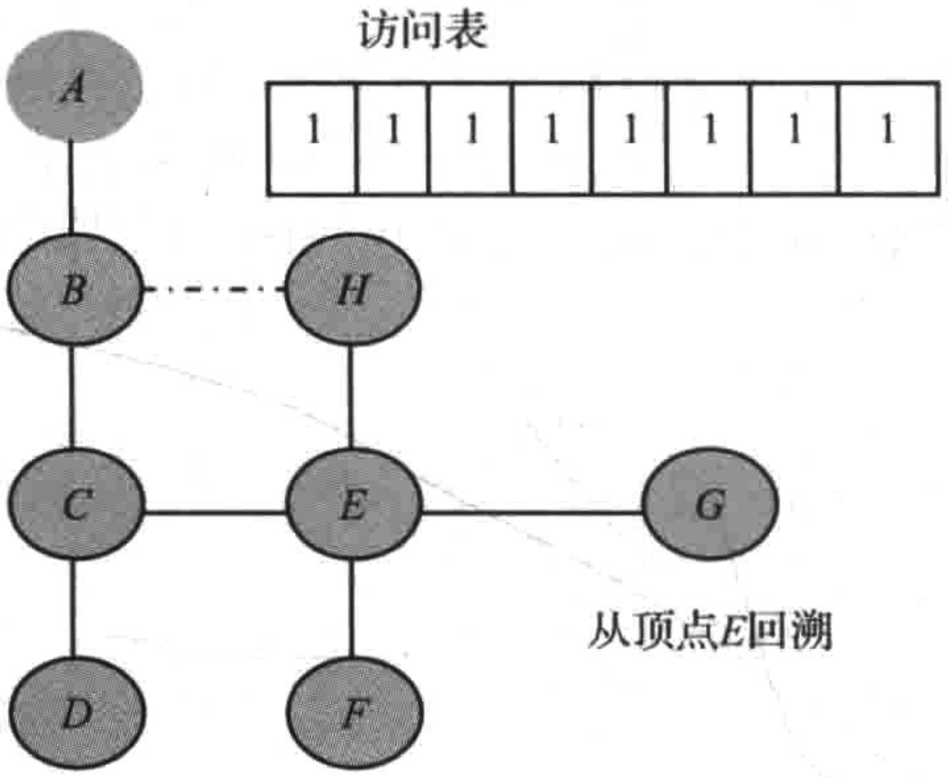

访问表
1 1 1 1 1 1 1 0
A
B
H
C
E
G
D
F
从顶点G回溯



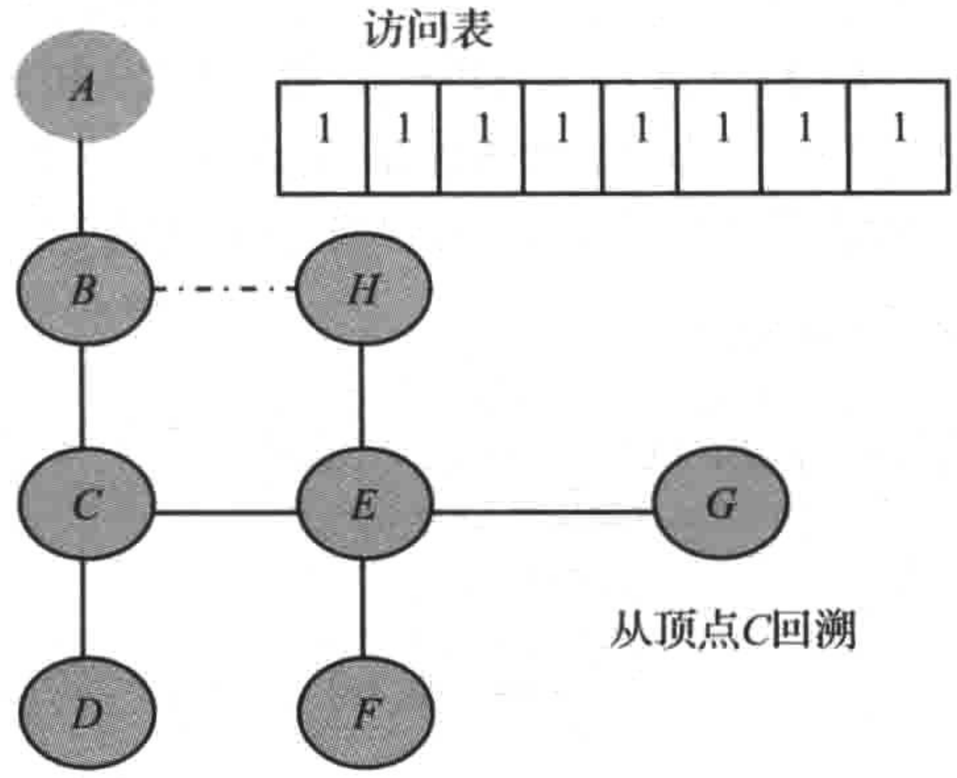

深度优先搜索的递归调用，顶点*H*被访问



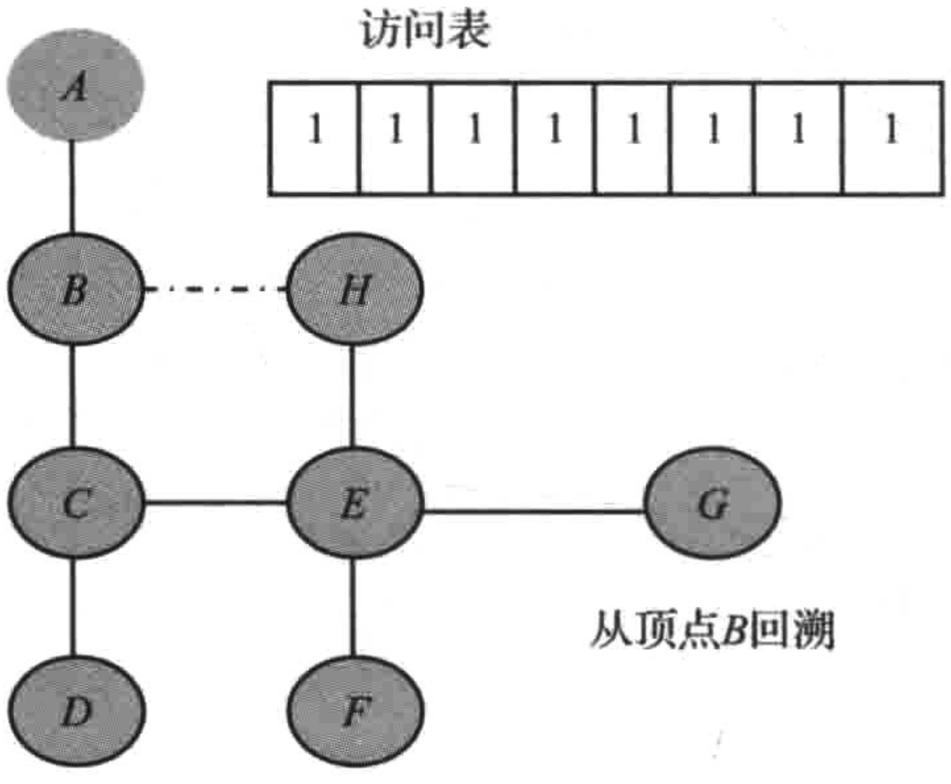

从顶点*H*回溯。边（*B*, *H*）是一个隐藏边



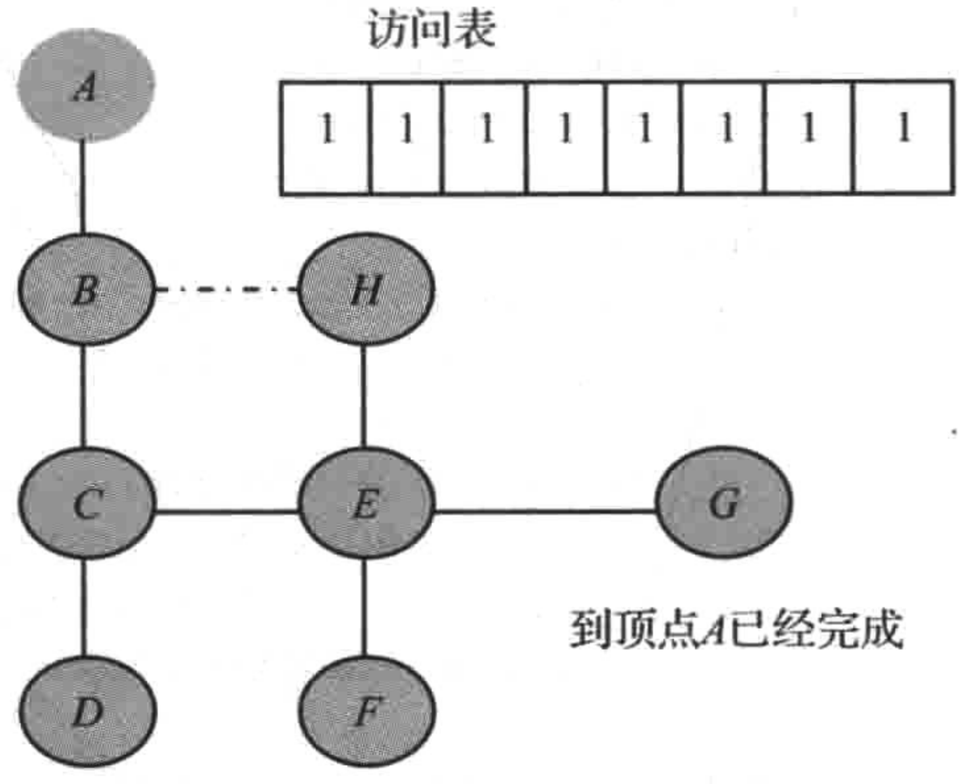

从顶点*E*回溯




从顶点*C*回溯




从顶点*B*回溯




到顶点*A*已经完成

从上面的图中可以看出，DFS 遍历将产生一棵树(没有回退边)，该树称为 DFS 树。即使给定的图中包含连通分量，上述算法也能运行。

如果使用邻接表来表示图，则 DFS 算法的时间复杂度是 $O(V+E)$。这是因为从某个顶点开始，算法只访问该顶点尚未被访问的邻接点。类似地，如果用邻接矩阵来表示图，那么算法难以快速找到所有与某个顶点相邻的边，因此其时间复杂度为 $O(V^2)$。

DFS 的应用

- 拓扑排序
- 查找连通分量
- 查找图的关节点(割点)
- 查找强连通分量
- 解决类似迷宫的问题

关于算法请参见 9.9 节。

### 2. 宽度优先搜索

宽度优先搜索(BFS)算法的原理类似于树的层次遍历，并且算法也使用了队列。事实上，层次遍历也是受到了 BFS 的启发。BFS 逐层对图进行遍历。初始时，BFS 从一个给定的顶点出发，该顶点位于第 0 层。第一步，它访问所有处于第一层的顶点(即图中到起始顶点距离为 1 的顶点)。第二步，访问第二层的所有顶点，即与第一层顶点相邻的顶点。

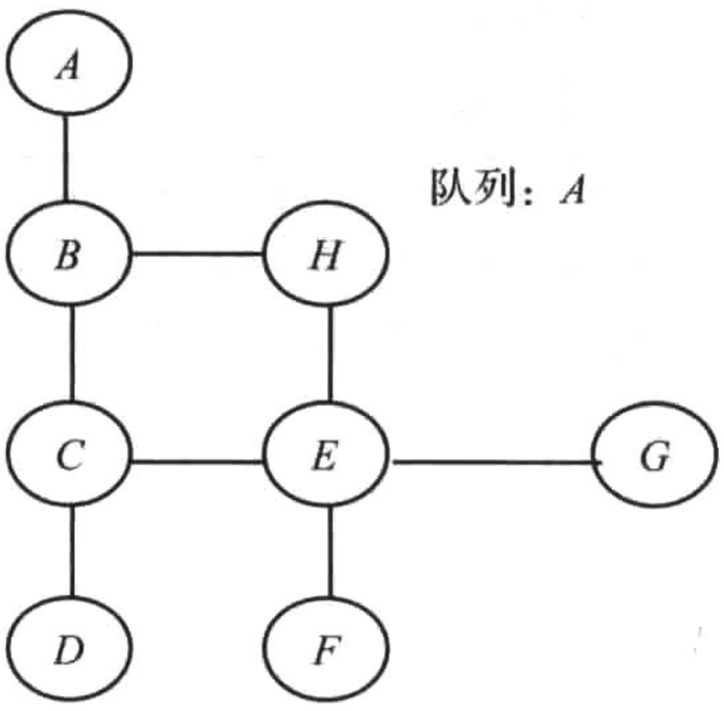
BFS 算法重复这个过程，直至图的所有层都访问一遍。通常，BFS 算法使用队列来存储每一层的顶点。与 DFS 类似，假设初始时所有顶点都标记为未被访问过(false)。已经处理过并已从队列中移除的顶点标记为已访问过(true)。利用另一个队列来表示所有已被访问的结点的集合，该队列将记录结点第一次被访问的顺序。上述思想的实现代码如下。

```
class Vertex {
        public char label;
        public boolean visited;
        public Vertex(char lab)  {
                label = lab;
                visited = false;
        }
}
class Graph {
        private final int maxVertices = 20;
        private Vertex vertexList[];
        private int adjMatrix[][];
        private int vertexCount;
        private Queue theQueue;
        public Graph() {
                vertexList = new Vertex[maxVertices];
                adjMatrix = new int[maxVertices][maxVertices];
                vertexCount = 0;
                for(int y=0; y<maxVertices; y++)
                        for(int x=0; x<maxVertices; x++)
                                adjMatrix[x][y] = 0;
                theQueue = new Queue();
}
public void addVertex(char lab) {
        vertexList[vertexCount++] = new Vertex(lab);
}
public void addEdge(int start, int end) {
        adjMatrix[start][end] = 1;
        adjMatrix[end][start] = 1;
}
```

```
public void displayVertex(int v) {
        System.out.print(vertexList[v].label);
}
public void bfs() {
        vertexList[0].wasVisited = true;
        displayVertex(0);
        theQueue.insert(0);
        int v2;
        while( !theQueue.isEmpty() ) {
            int v1 = theQueue.remove();
            while( (v2=getAdjUnvisitedVertex(v1)) != -1 ) {
                vertexList[v2].wasVisited = true;
                displayVertex(v2);
                theQueue.insert(v2);
            }
        }
        for(int j=0; j<nVerts; j++)
            vertexList[j].wasVisited = false;
}
public int getAdjUnvisitedVertex(int v) {
        for(int j=0; j<vertexCount; j++)
            if(adjMatrix[v][j]==1 && vertexList[j].visited==false)
                return j;
            return -1;
        }
}
```

以 DFS 算法中给出的图为例，BFS 遍历图如下。

从顶点*A*开始被标记为未访问。假设这是在0级上

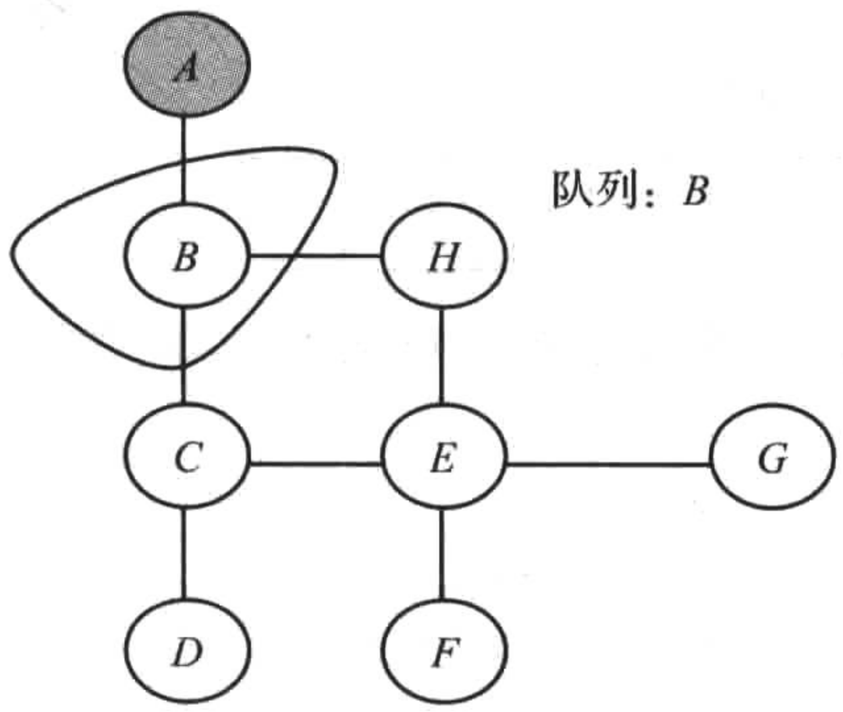



队列：*A*

访问表

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

顶点*A*完成了。被圈的部分是1级而且被添加到队列中



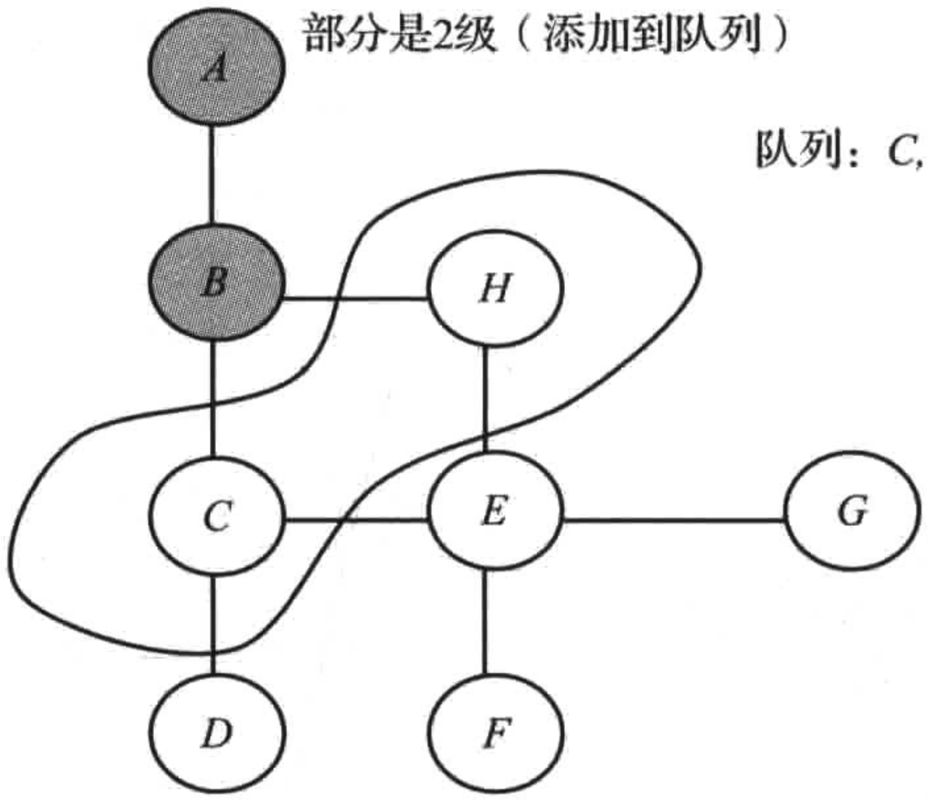

队列：*B*

访问表

| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

顶点*B*完成了。被选择的部分是2级（添加到队列）

队列：*C*, *H*

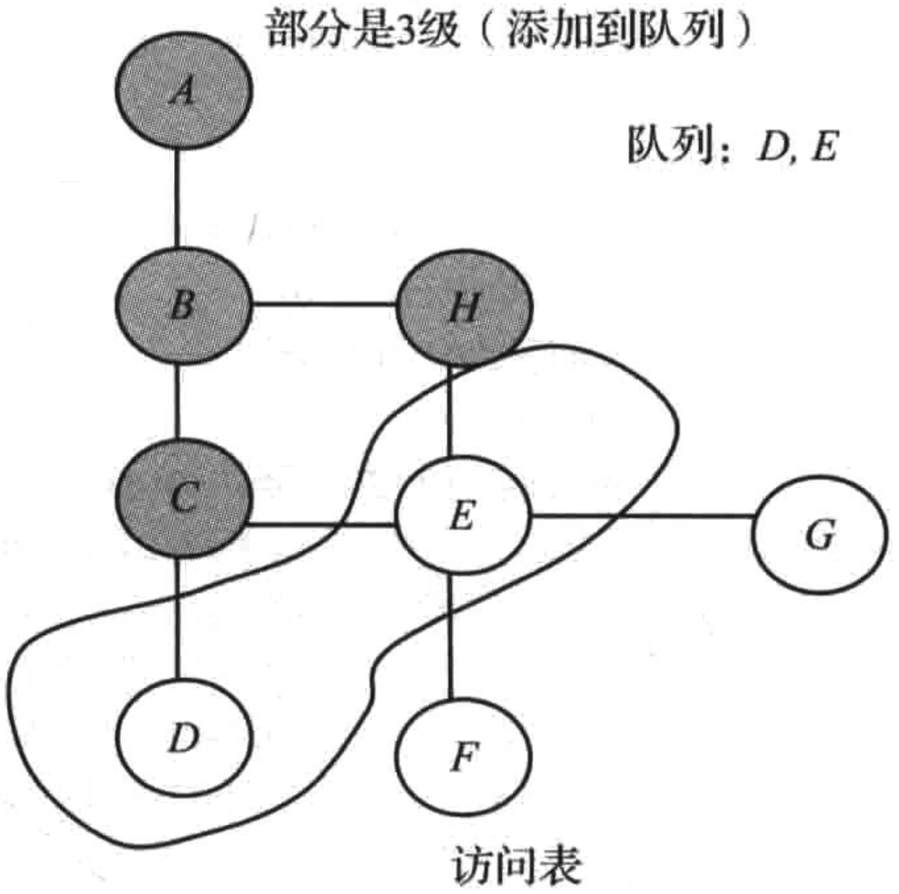



访问表

| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

顶点*C*和*H*完成了。被圈的部分是3级（添加到队列）

队列：*D*, *E*

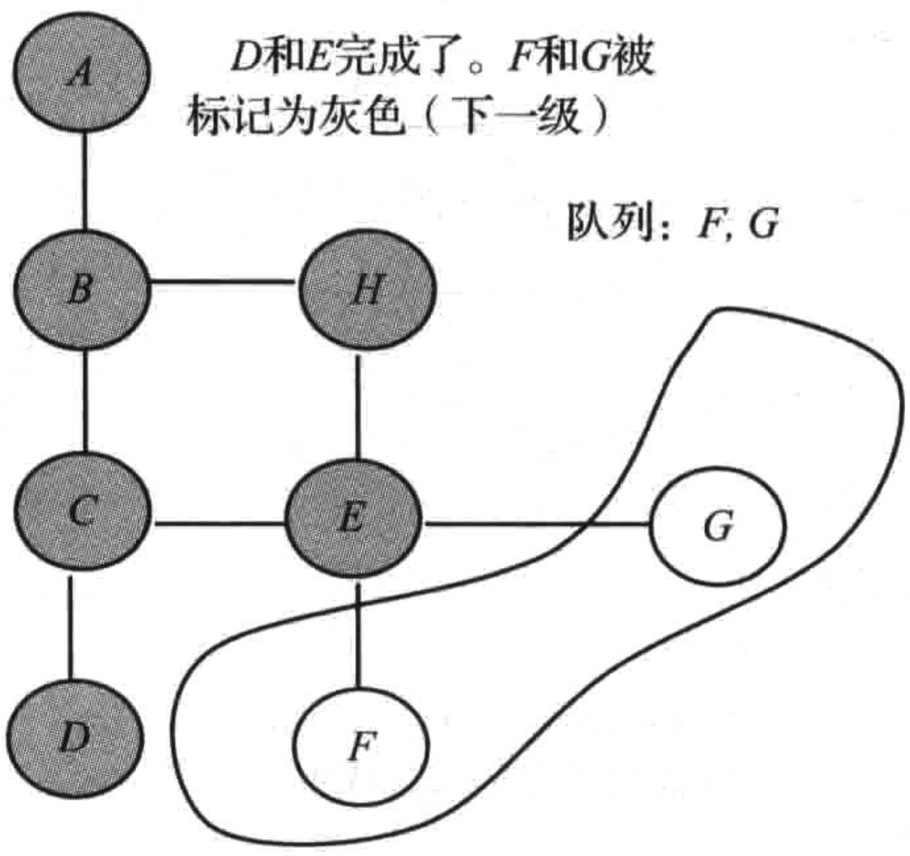



访问表

| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

*D*和*E*完成了。*F*和*G*被标记为灰色（下一级）

队列：*F*, *G*

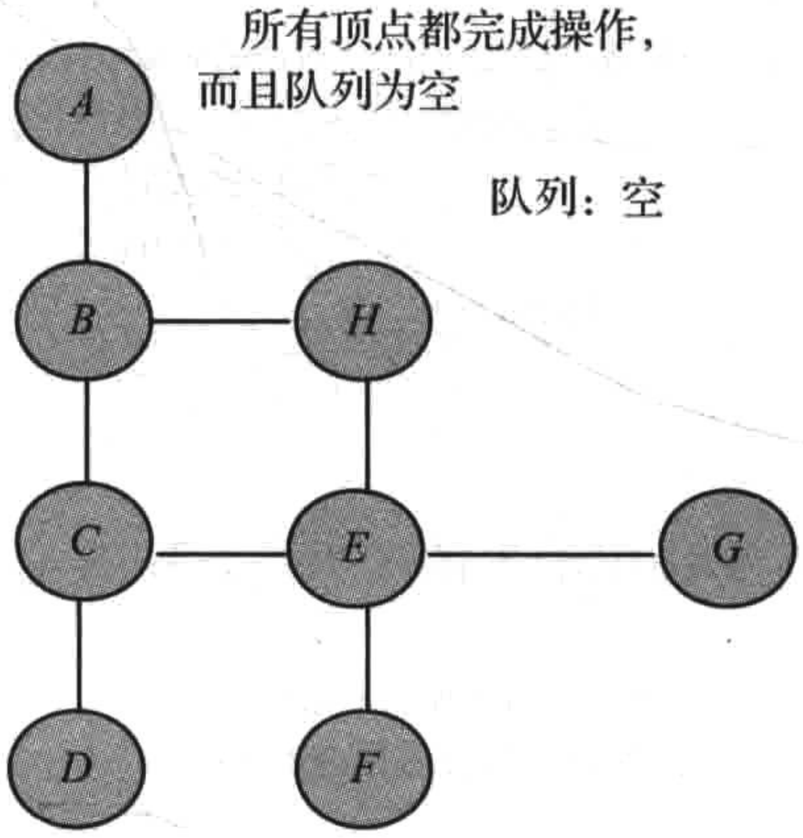



访问表

| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

所有顶点都完成操作，而且队列为空

队列：空




访问表

| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

如果使用邻接表来表示图，那么 BFS 算法的时间复杂度为 $O(V+E)$；使用邻接矩阵时，为 $O(V^2)$。

**BFS 的应用**

- 查找图中所有的连通分量。
- 查找某个连通图中的所有结点。
- 查找两个结点之间的最短路径。
- 测试图的二分性。

**3. DFS 和 BFS 的比较**

对比 BFS 和 DFS 可知，DFS 的最大优势在于它的内存开销要远远小于 BFS，因为它不需要存储每一层结点的所有孩子结点指针。根据数据和查找内容的不同，DFS 和 BFS 各有

优势。例如，在一个家族树中，如果需要查找某个人是否仍然健在且假设这个人处于树的末端，那么 DFS 是一个更好的选择。而 BFS 可能需要花费非常长的时间达到最后一层。

DFS 算法能更快地找到目标。现在，如果要寻找一个已经过世很长时间的人，那么这个人可能更接近于树的顶端。在这种情况下，BFS 查找比 DFS 快。因此，每种算法的优势取决于数据和要查找的内容。

DFS 与树的前序遍历有关。与前序遍历一样，DFS 先访问某个结点，再访问该结点的孩子结点。BFS 算法的原理类似于树的层次遍历。

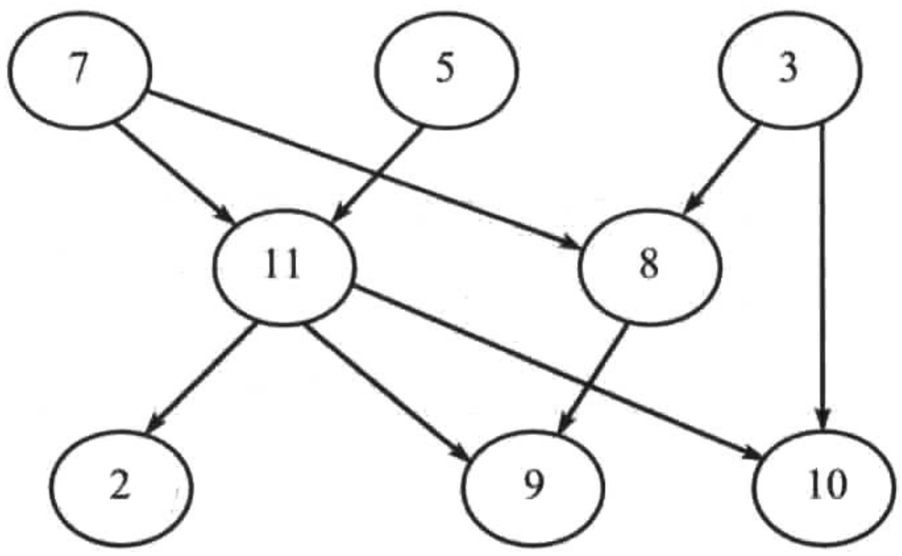
DFS 和 BFS 哪个更好？答案取决于需要解决的问题类型。BFS 每次访问一层，若预先知道需要搜索的结果处在一个较低的深度(更接近树的顶端)，那么 BFS 是合适的。若结果处于最大深度，则 DFS 是更好的选择。下表给出了 DFS 和 BFS 在应用方面的不同之处。

| 应用 | DFS | BFS |
|---|---|---|
| 生成森林、连通分量、路径、环路 | 是 | 是 |
| 最短路径 | | 是 |
| 内存开销最小 | 是 | |

## 9.6 拓扑排序

拓扑排序是在一个有向无环图(DAG)中对顶点的排序。在这个有向无环图中，每个顶点都排在所有以它为起点的相邻结点之前。以大学课程的选修优先条件为例。一条有向边($v$，$w$)表示课程 $v$ 必须在课程 $w$ 之前完成。在该例中，拓扑排序是课程选修的一个序列，该序列符合给定的课程领先关系。每一个 DAG 图可以有一个或多个拓扑排序。若图中存在环，则无法进行拓扑排序，因为环上的两个点 $v$ 和 $w$ 将相互依赖，即 $v$ 先于 $w$ 且 $w$ 也先于 $v$。

拓扑排序有一个有趣的特性，即排好序的所有连续顶点对之间都是有边相连的。这些边在 DAG 图中会形成一个有向哈密顿路径(请参见 9.9 节)。若有一条哈密顿路径存在，则拓扑排序的顺序是唯一的。如果拓扑排序没有形成哈密顿路径，则 DAG 可能有两个或更多的拓扑顺序。在下图所示的例子中，7，5，3，11，8，2，9，10 和 3，5，7，8，11，2，9，10 都是拓扑序列。




初始时，计算所有顶点的入度，并从入度为 0 的顶点出发，因为这些顶点没有任何先决条件。可以使用队列来跟踪这些入度为 0 的顶点。

将所有入度为 0 的顶点放入队列中。当队列不为空时，从队列中移除顶点 $v$，并将 $v$ 的所有相邻顶点的入度减 1。一旦某个顶点的入度变为 0，就将其放入队列中。因此，拓扑顺序就是队列顶点出队的顺序。当采用邻接表时，该算法的时间复杂度为 $O(|E|+|V|)$。

```
void TopologicalSort( Graph G ) {
        LLQueue Q = new LLQueue();
        int counter;
        int v, w;
        counter =  0;
        for (v = 0; v< G.vertexCount; v++)
                if( indegree[v] ==   0 )
                        Q.enQueue(v );
        while( !Q.isEmpty() ) {
                v = Q.deQueue();
                topologicalOrder[v] = ++counter;
                for each w adjacent to v
                        if( --indegree[w] ==  0 )
                                Q.enQueue(w);
        }
        if( counter != G.vertexCount)
                System.out.println("Graph has cycle");
        Q.deleteQueue();
}
```

拓扑排序的总运行时间是 $O(V+E)$。

**注意：** DFS 能够用于解决拓扑排序问题。具体算法实现请参见 9.9 节。

**拓扑排序的应用**

- 课程选修的先决条件
- 检测死锁
- 计算作业的流水线
- 检查符号链接环
- 解析电子表格中的公式

## 9.7 最短路径算法

现在讨论图中另一个重要的问题。给定一个图 $G=(V, E)$和一个特殊的顶点 $s$，需要查找从 $s$ 到图中其他每个顶点的最短路径。根据输入图形的类型不同，最短路径算法相应地有一些变化，主要包括以下三种。

**最短路径算法的种类**

| 无权图中的最短路径 |
| --- |
| 有权图中的最短路径 |
| 带有负边的有权图中的最短路径 |

**最短路径算法的应用**

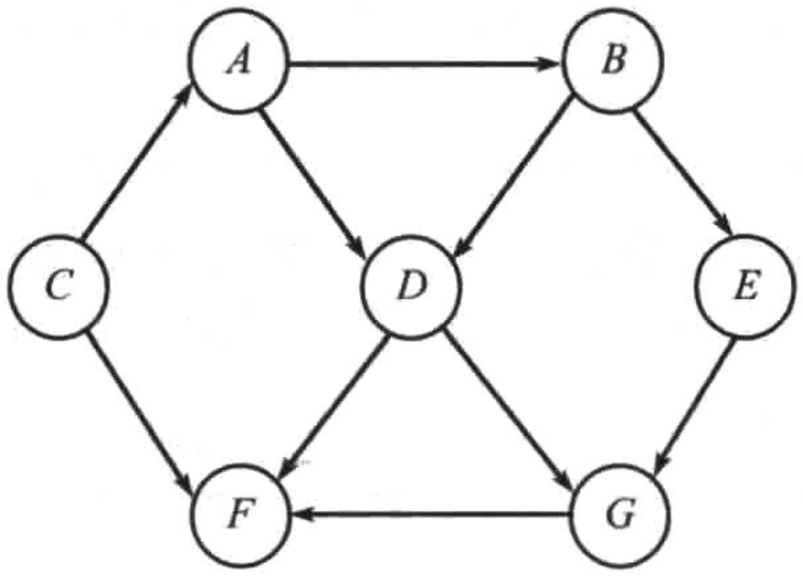
- 查找从一个地方到另一个地方的最快方式。
- 查找从一个城市到另一个城市的飞行/发送数据的最便宜的方式。

### 1. 无权图中的最短路径

假设要寻找某个输入顶点 $s$ 到所有其他顶点的最短路径。无权图是有权最短路径问题的特例，即边的权重均为 1。算法类似于 BFS，需要使用下列数据结构：

- 一个包含 3 列的距离表(每行对应一个顶点)。
  - 到源点的距离。
  - 路径——包含最短路径上经过的顶点。
- 一个用于实现宽度优先搜索的队列。它包含到源点距离已知的结点以及尚未访问的相邻顶点。

以下图为例，考虑它的邻接表表示。




上图的邻接表为：

**A**：$B \rightarrow D$

**B**：$D \rightarrow E$

**C**：$A \rightarrow F$

**D**：$F \rightarrow G$

**E**：$G$

**F**：—

**G**：$F$

设 $s=C$。从 $C$ 到 $C$ 的距离是 0。初始时，$C$ 到所有其他结点的距离尚未确定，将距离表上除了 $C$ 以外的其他点的第二列设为 $-1$，如下表所示。

| 顶点 | Distance[$v$] | 获得 Distance[$v$]的前一个顶点 | 顶点 | Distance[$v$] | 获得 Distance[$v$]的前一个顶点 |
|---|---|---|---|---|---|
| $A$ | $-1$ | — | $E$ | $-1$ | — |
| $B$ | $-1$ | — | $F$ | $-1$ | — |
| $C$ | 0 | — | $G$ | $-1$ | — |
| $D$ | $-1$ | — | | | |

**算法**

```
void UnweightedShortestPath(Graph G, int s) {
        LLQueue Q = new LLQueue();
        int v, w;
        Q.enQueue( s);
        for (int i = 0; i< G.vertexCount;i++)
            Distance[i]=-1;
        Distance[s]= 0;
        while (!Q.isEmpty()) {
            v = Q.deQueue();

            for each w adjacent to v
                if(Distance[w] == -1)      {     //每个顶点最多检查1次
                    Distance[w] = Distance[v] + 1;
                    Path[w] = v;
                    Q.enQueue(w); //每个顶点最多进入队列1次
                }
        }
        Q.deleteQueue();
}
```

如果使用邻接表，则运行时间为 $O(|E|+|V|)$。在 for 循环中，算法检查每个顶点的出边；在 while 循环中所有访问过的边的和等于边的数目，即为 $O(|E|)$。

如果使用矩阵表示，则时间复杂度是 $O(|V|^2)$，因为必须在长度为 $|V|$ 的矩阵中读入一整行，以便查找给定顶点的相邻顶点。

**2. 有权图中的最短路径(Dijkstra 算法)**

Dijkstra 给出了一个著名的最短路径问题解决方法。Dijkstra 算法是对 BFS 算法的推广。普通 BFS 算法不能解决最短路径问题，因为它不能保证处于队列最前面的顶点是最接近源 $s$ 的顶点。

在给出代码实现前，先来理解算法的实现过程。与在无权最短路径算法中一样，这里也使用距离表。算法在距离表中保存从源点到顶点 $v$ 的最短路径。Distance[$v$]记录从 $s$ 到 $v$ 的距离。源点到它自身的最短距离为 0。距离表中将一个顶点到其他顶点的距离设为 −1，表示那些尚未访问的顶点。

| 顶点 | Distance[$v$] | 获得 Distance[$v$]的前一个顶点 | 顶点 | Distance[$v$] | 获得 Distance[$v$]的前一个顶点 |
|---|---|---|---|---|---|
| $A$ | −1 | — | $E$ | −1 | — |
| $B$ | −1 | — | $F$ | −1 | — |
| $C$ | 0 | — | $G$ | −1 | — |
| $D$ | −1 | — | | | |

在算法完成后，距离表记录从源点 $s$ 到每个顶点 $v$ 的最短距离。为了更容易地理解 Dijkstra 算法，将给定的顶点分成两个集合。初始时，第一个集合仅包含源点，而第二个集合包含所有剩余顶点。在第 $k$ 次迭代后，第一个集合包含 $k$ 个距离源点最近的顶点。这 $k$ 个顶点是从源点计算最短距离的顶点。

a) 对 Dijkstra 算法的注解

- 采用贪婪法：总是选取最接近源点的顶点。
- 使用优先队列并按照到 $s$ 的距离来存储未被访问过的顶点。
- 不能用于权值为负的情况。

b) 无权最短路径算法与 Dijkstra 算法之间的区别

1) 为了在邻接表中表示权重，每个顶点要包含边的权重信息(除了它们的标识符外)。

2) 使用优先队列来代替常规队列(距离作为优先级)，选择并访问距离最短的顶点。

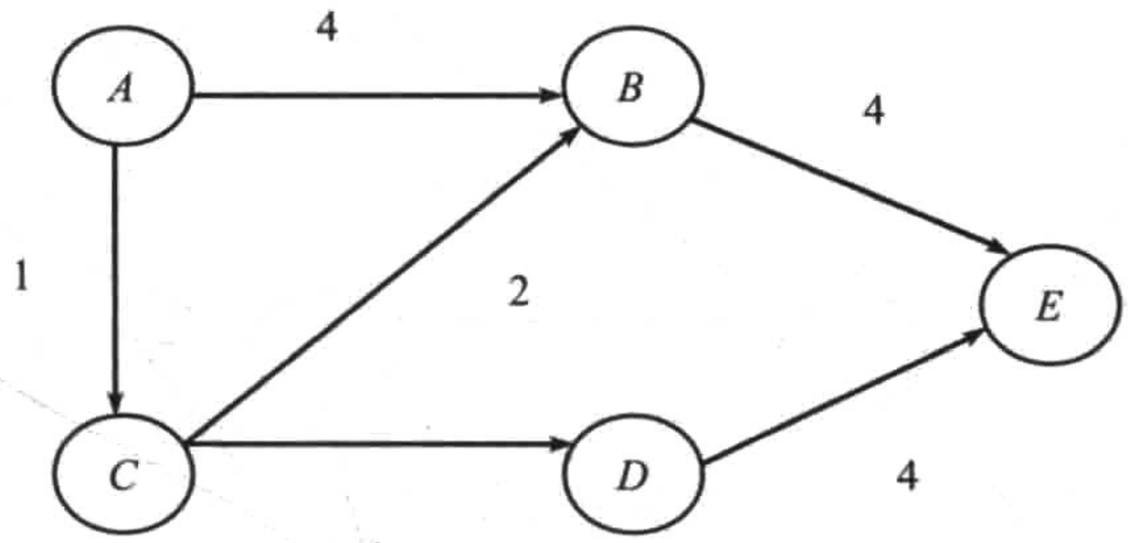
3) 源点到某个顶点的距离为从源点到该顶点的路径上的所有边权值之和。

4) 当新计算得到的距离小于原有的距离值时，更新距离。

```
void Dijkstra(Graph G, int s) {
        Heap PQ = new Heap();
        int v, w;
        PQ.enQueue(s);
        for (int i = 0; i< G.vertexCount;i++)
                Distance[i]=-1;
        Distance[s] = 0;
        while ((!PQ .isEmpty()) {
                v = PQ.deleteMin();
                for all adjacent vertices w of v {
                        Compute new distance d= Distance[v] + weight[v][w];
                        if(Distance[w] == -1) {
```

```
                    Distance[w] = new distance d;
                    Insert w in the priority queue with priority d
                    Path[w] = v;
                }
                if(Distance[w]  > new distance d) {
                    Distance[w] = new distance d;
                    Update priority of vertex w to be d;
                    Path[w] = v;
                }
            }
        }
```

下面通过一个例子来加深对算法的理解。该例子能够解释算法操作的每一步以及距离是如何计算的。在下图所示的有权图中有 $A\sim E$ 5 个顶点。两个顶点之间的值即为边上的权重。例如，$A$ 和 $C$ 之间的边权值是 1。Dijkstra 算法能够用来查找从源点 $A$ 到图中其余顶点的最短路径。

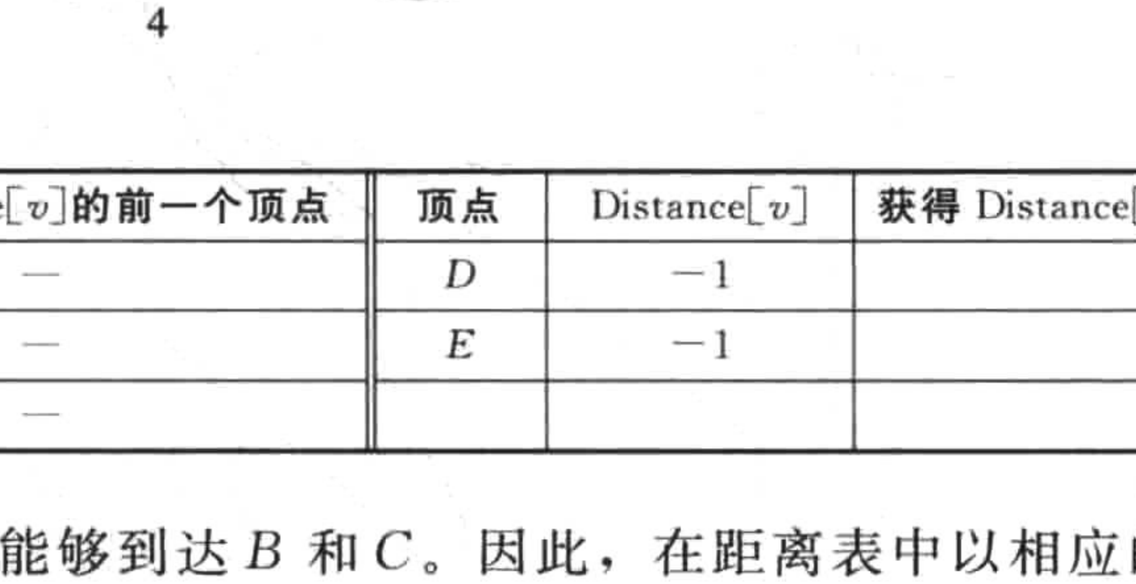



初始化距离表为：

| 顶点 | Distance[$v$] | 获得 Distance[$v$]的前一个顶点 | 顶点 | Distance[$v$] | 获得 Distance[$v$]的前一个顶点 |
|---|---|---|---|---|---|
| $A$ | 0 | — | $D$ | −1 | — |
| $B$ | −1 | — | $E$ | −1 | — |
| $C$ | −1 | — | | | |

完成第一步后，从顶点 $A$ 能够到达 $B$ 和 $C$。因此，在距离表中以相应的边权值来更新顶点 $B$ 和 $C$ 的可达性，同样如下图所示。

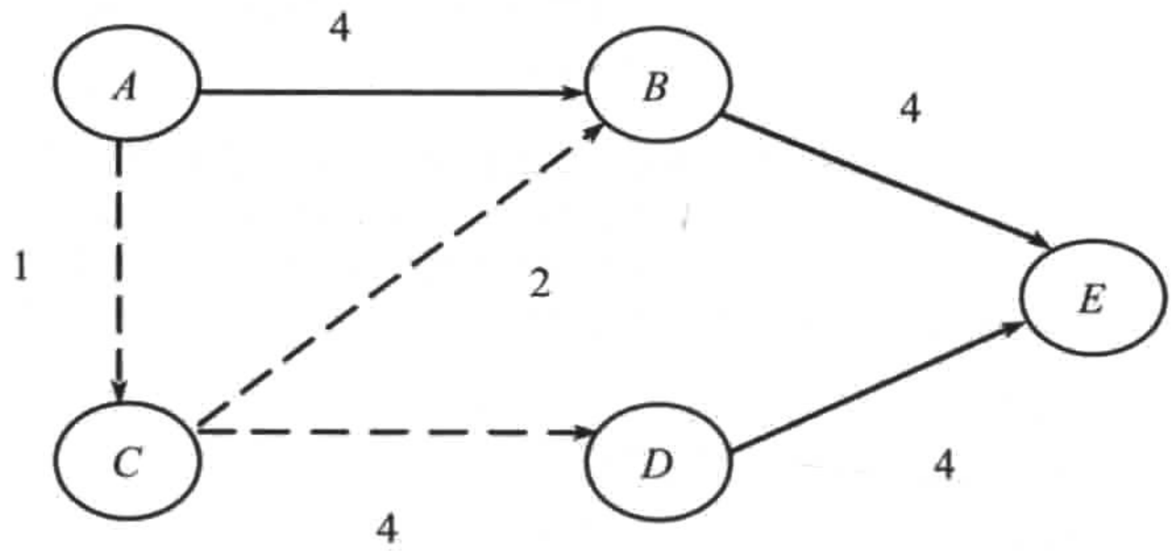
| | | |
|---|---|---|
| *A* | 0 | – |
| *B* | 4 | A |
| *C* | 1 | A |
| *D* | –1 | – |
| *E* | –1 | – |




现在，从距离表中选择一个最小距离。最小距离顶点是 $C$。这表明必须通过这两个顶点($A$ 和 $C$)才能到达其他顶点。例如，从 $A$ 和 $C$ 都能到达 $B$。在这种情况下，必须选择代价更小的那条边。因为通过 $C$ 到 $B$ 的代价(1＋2)更小，所以在距离表中用 3 来更新顶

点 $B$，并且保存产生此代价的顶点 $C$。

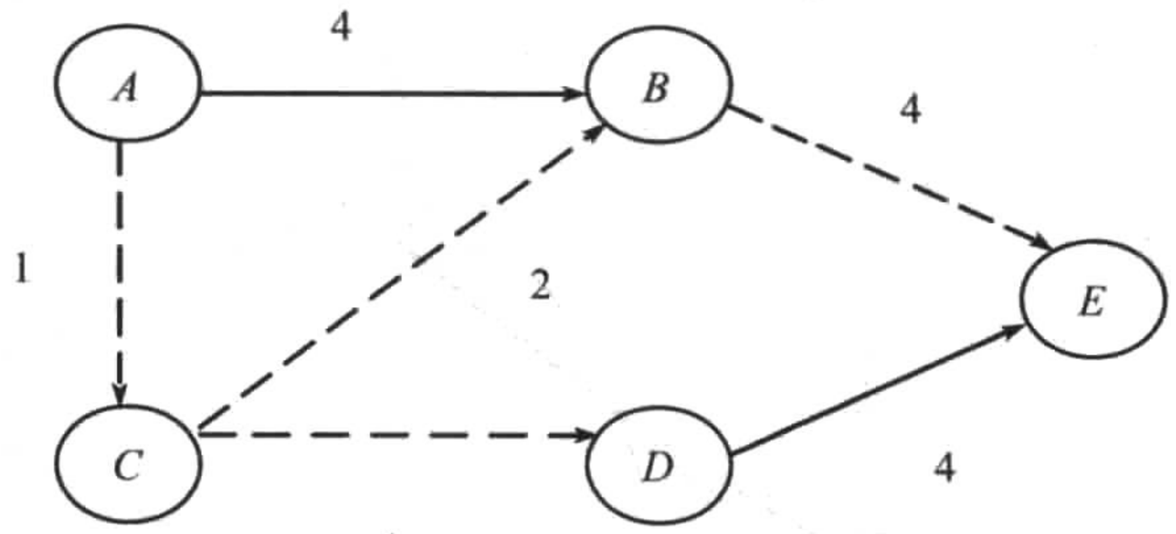
| | | |
|---|---|---|
| *A* | 0 | – |
| *B* | 3 | C |
| *C* | 1 | A |
| *D* | 5 | C |
| *E* | –1 | – |



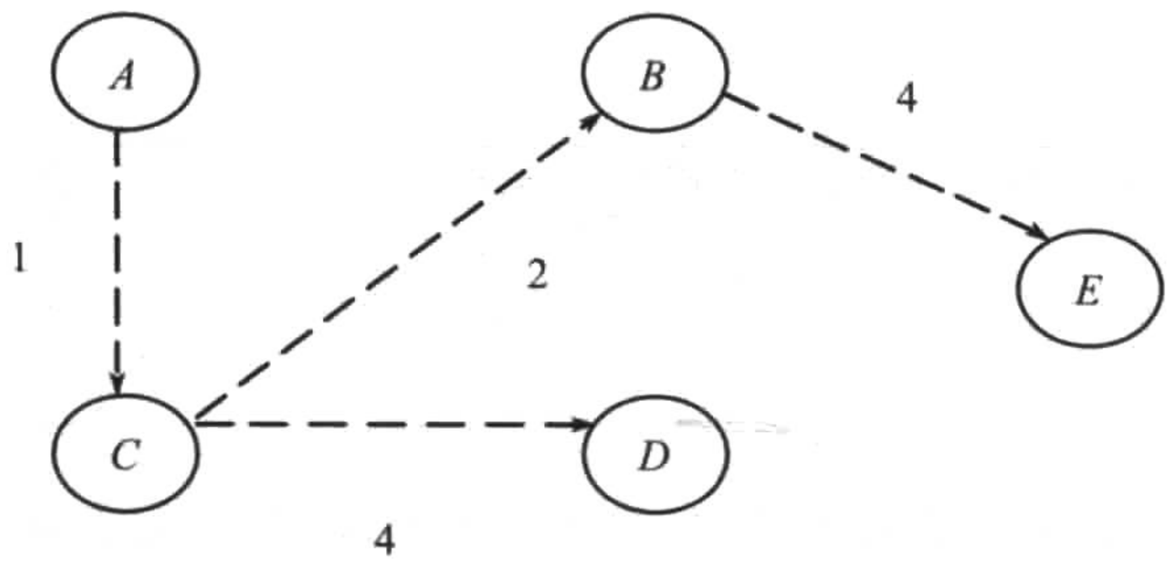

使用$C$作为中间结点到达$B$、$D$的最短路径

当前唯一未被访问的结点为 $E$。为了到达 $E$，需要找出所有可以到达 $E$ 的路径并选择其中代价最小的路径。可以发现，当使用经过 $C$ 到达的 $B$ 顶点作为中间顶点时具有最小代价。

| | | |
|---|---|---|
| *A* | 0 | – |
| *B* | 3 | C |
| *C* | 1 | A |
| *D* | 5 | C |
| *E* | 7 | B |




Dijkstra 算法最终产生的最小代价树如下图所示。




c）性能

在 Dijkstra 算法中，性能取决于 DeleteMins 和更新优先队列的次数。如果采用标准二进制堆，则算法的时间复杂度是 $O(E\log V)$。其中 $E\log V$ 来自于对标准堆的 $E$ 次更新（每次更新花费 $\log V$）。若使用数组来表示集合，则时间复杂度是 $O(E+V^2)$。

d）Dijkstra 算法的缺点

- 如上所述，该算法主要缺点是，盲目搜索导致的时间浪费和资源浪费。
- 另一个缺点是，它不能处理边权值为负的情况。这将导致无环图，并且很多时候它不能得到正确的最短路径。

e）与 Dijkstra 的相关算法

- Bellman-Ford 算法计算有权图中的单源最短路径。它采用与 Dijkstra 算法相同的原理，但能处理边权值为负的情况。它的时间复杂度比 Dijkstra 算法高。
- Prim 算法在一个连通有权图中查找最小生成树。这表示边的子集形成一棵树，这

棵树中所有边的权值之和是最小的。

### 3. Bellman-Ford 算法

Dijkstra 算法不能用于边权值为负的情况。这是由于当某个顶点 $u$ 被标记为已访问时，仍然存在这样一种可能，即存在一条从某个未被访问的顶点 $v$ 到 $u$ 的负路径。在这种情况下，从 $s$ 出发经过 $v$ 再到 $u$ 的路径的长度小于从 $s$ 出发到 $u$ 但不经过 $v$ 的路径的长度。

Dijkstra 算法与无权图算法相结合可以解决这个问题。用 $s$ 初始化队列。然后，在每一步将顶点 $v$ 出队。找到 $v$ 的所有相邻顶点 $w$ 使得：

到 $v$ 的距离 + 边$(v,w)$ 的权值 < 到 $w$ 的原有距离

对 $w$ 的原有距离和路径进行更新，并且若 $w$ 不在队列中，则将 $w$ 入队。可以为每个顶点设置一个标记位来表示它是否已经在队列中。重复该过程直至队列为空。

```
void BellmanFordAlgorithm(Graph G, int s) {
        LLQueue Q = new LLQueue();
        int v, w;
        Q.enQueue( s);
        //假定用INT_MAX填充距离表
        Distance[s] = 0;
        while ((!Q.isEmpty()) {
                v = Q.deQueue();
                for all adjacent vertices w of v {
                        Compute new distance d= Distance[v] + weight[v][w];
                        if(old distance to w > new distance d ) {
                                Distance[v] = (distance to v) + weight[v][w];
                                Path[w] = v;
                                if(w is there in queue)
                                        Q.enQueue( w);
                        }
                }
        }
}
```

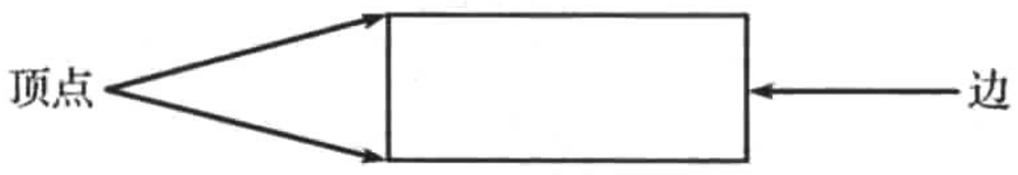
当图中不存在由负边形成的回路时，算法能够正常运行。每个顶点最多出队 $|V|$ 次，所以当使用邻接表时运行时间为 $O(|E||V|)$。

### 4. 最短路径算法总结

| | |
|---|---|
| 无权图的最短路径(改进的 BFS) | $O(\|E\|+\|V\|)$ |
| 有权图的最短路径(Dijkstra) | $O(\|E\|\log\|V\|)$ |
| 含有负边的有权图的最短路径(Bellman-Ford) | $O(\|E\|\|V\|)$ |
| 有权无环图的最短路径 | $O(\|E\|+\|V\|)$ |

## 9.8 最小生成树

图的生成树是一个包含所有顶点的子图并且是一棵树。一个图可能有很多生成树。以下图所示的包含 4 个顶点的图为例。假设图中的拐角代表顶点。



对于这个简单图，有多棵生成树，如下图所示。



接下来讨论无向图中的最小生成树算法。假设给定的图是有权图，当图是无权图时，仍然可以采用适用于有权图的最小生成树算法来解决无权图的问题，只需要将有权图中边的权值视为都相等即可。无向图的最小生成树是由图的边所组成的树，这些边连接图的所有顶点且边的权值之和最小。仅当图是连通的才存在该图的最小生成树。

有两个著名的算法用于解决最小生成树问题：

- Prim 算法
- Kruscal 算法

### 1. Prim 算法

Prim 算法和 Dijkstra 算法几乎相同。与 Dijkstra 算法类似，Prim 算法也利用距离表来保存距离和路径。唯一的区别是，由于距离的定义不同，所以更新操作也略有不同。相比之下更新操作更简单。

```
void Prims(Graph G, int s) {
        Heap PQ = new Heap();
        int v, w;
        PQ.enQueue(s);

        //假设距离表用-1填充
        Distance[s] = 0;
        while ((!PQ.isEmpty()) {
                v = PQ.DeleteMin();

                for all adjacent vertices w of v {
                        Compute new distance d= Distance[v] + weight[v][w];

                        if(Distance[w] == -1) {
                                Distance[w] = weight[v][w];
                                Insert w in the priority queue with priority d
                                Path[w] = v;
                        }
                                if(Distance[w]  > new distance d) {
                                        Distance[w] = weight[v][w];
                                Update priority of vertex w to be d;
                                Path[w] = v;
                        }
                }
        }
}
```

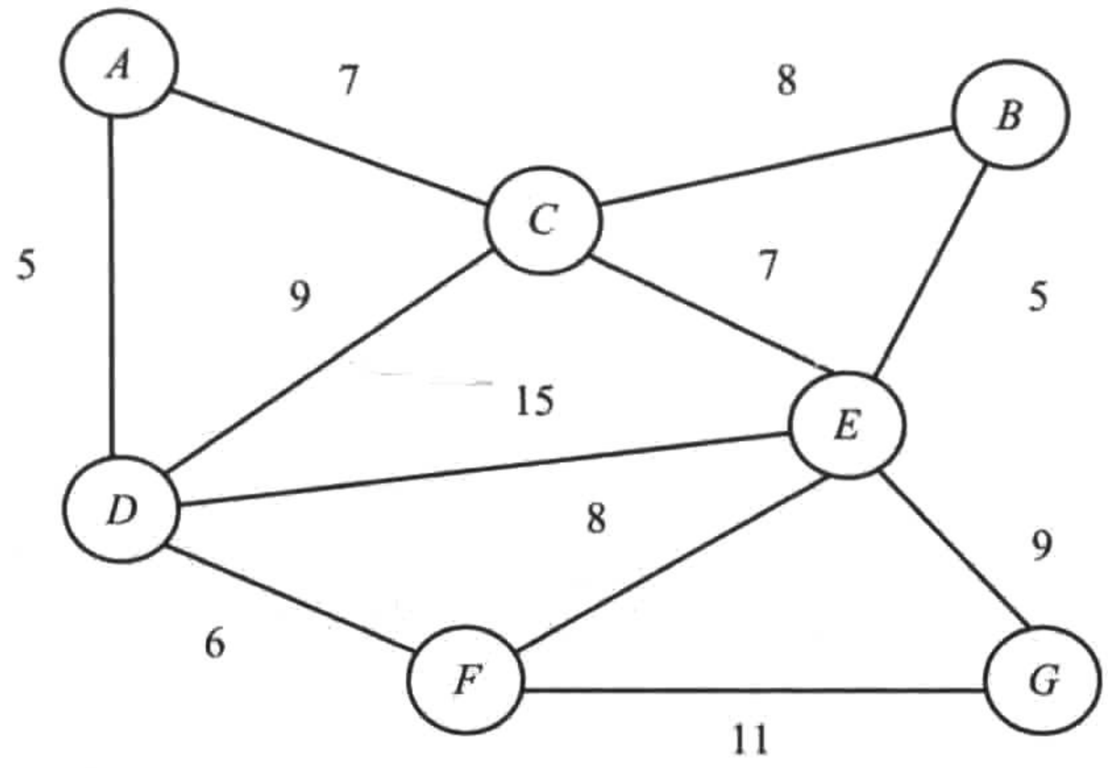
这个算法的实现过程与 Dijkstra 算法是相同的。在不使用堆的情况下，算法的运行时间是 $O(|V|^2)$（适用于稠密图），使用二叉堆时为 $O(E\log V)$（适用于稀疏图）。

### 2. Kruskal 算法

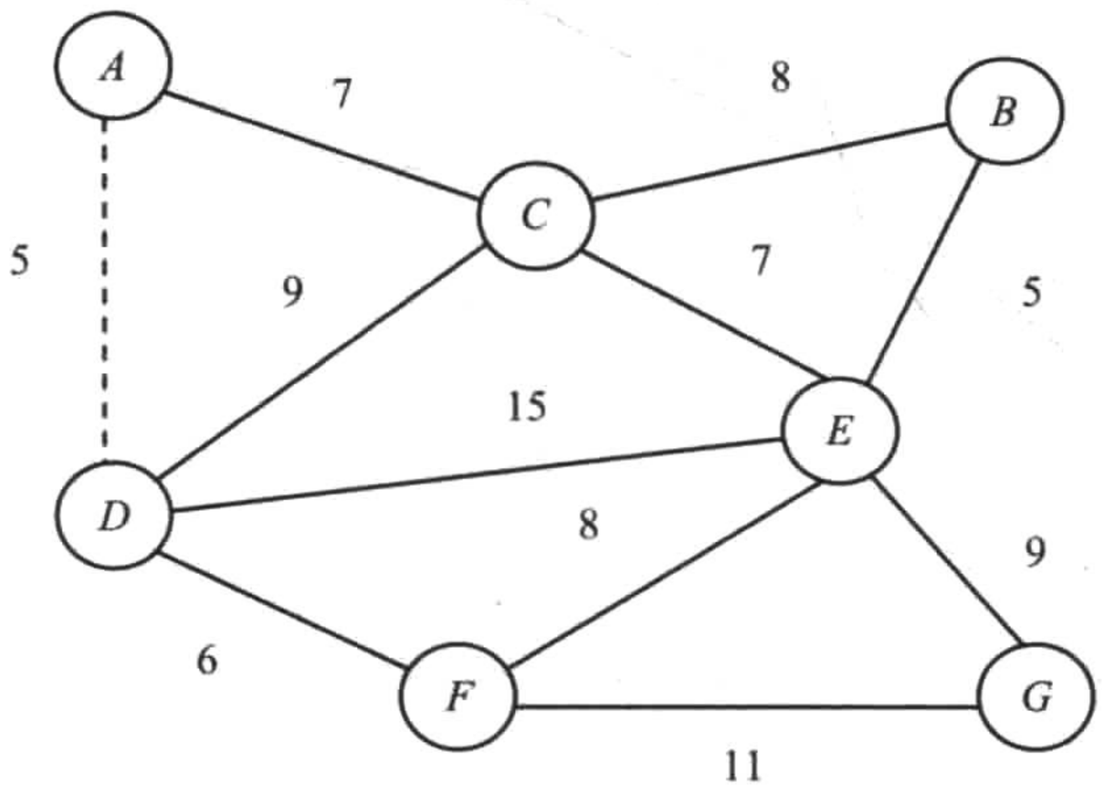
算法从 $V$ 个不同的树开始，其中 $V$ 为图中的顶点。当构造最小生成树时，算法每次选择一条权值最小且不会形成回路的边并将其加入生成树中。因此，初始化时共有 $|V|$ 棵单顶点树在森林中。当加入一条边时，两棵树就合成为一棵树。当算法完成时，将只剩一棵树，该树即为最小生成树。有如下两种方式可以实现 Kruskal 算法：

- 使用并查集：使用 UNION 和 FIND 操作。
- 使用优先队列：维持优先队列中的权重。

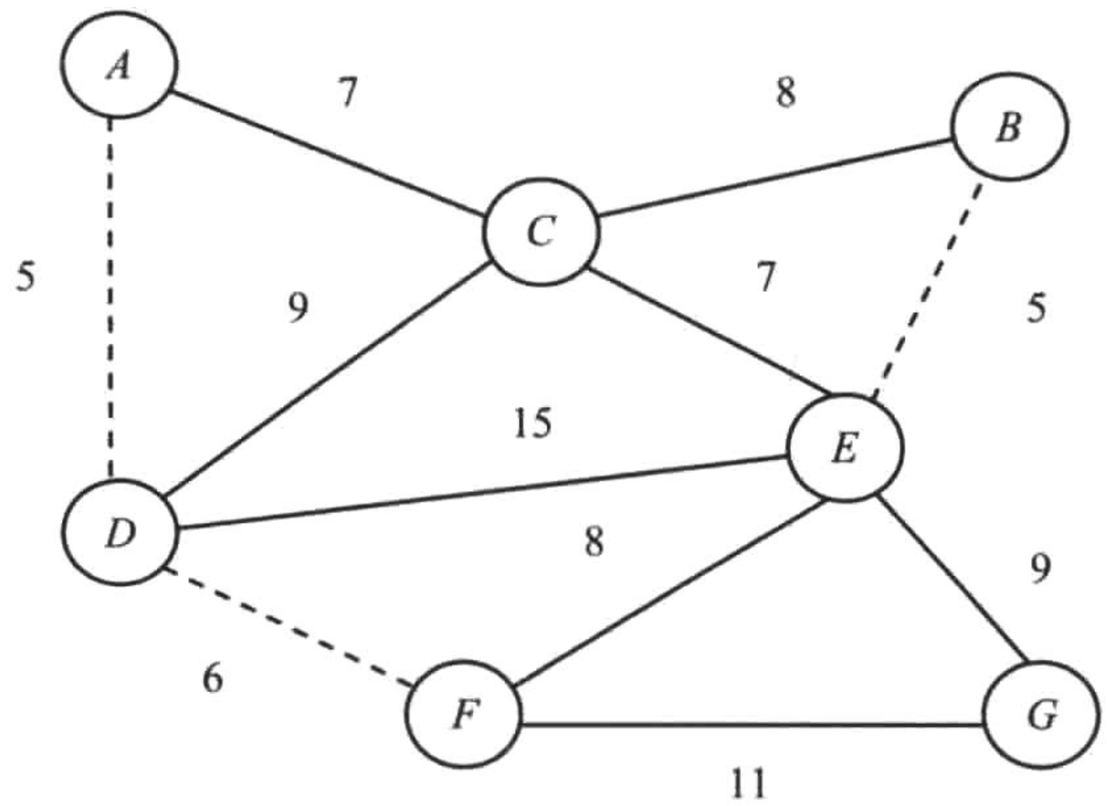
合适的数据结构是并查集，使用 UNION/FIND 算法(森林的应用)。两个顶点属于同一个集合当且仅当它们在当前的生成森林中是连通的。初始时每个顶点分别属于各自的集合。若顶点 $u$ 和 $v$ 在同一个集合中，则拒绝添加 $u$ 和 $v$ 之间的边，因为会形成回路。否则，接受两点之间的边，并将包含 $u$ 和 $v$ 的两个集合合并。考虑下图(图中各边的数值表示相应边的权重)。



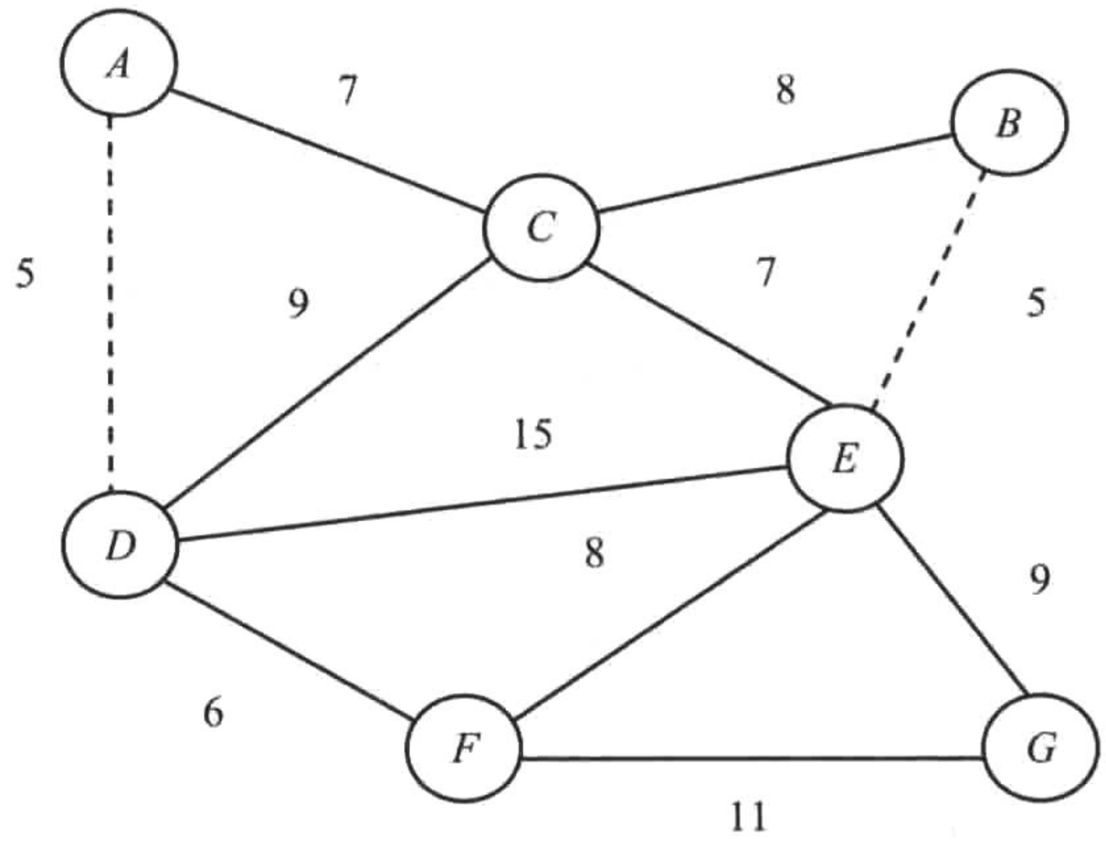

现在以该图为例执行 Kruscal 算法。总是选择权重(开销)最小的边。



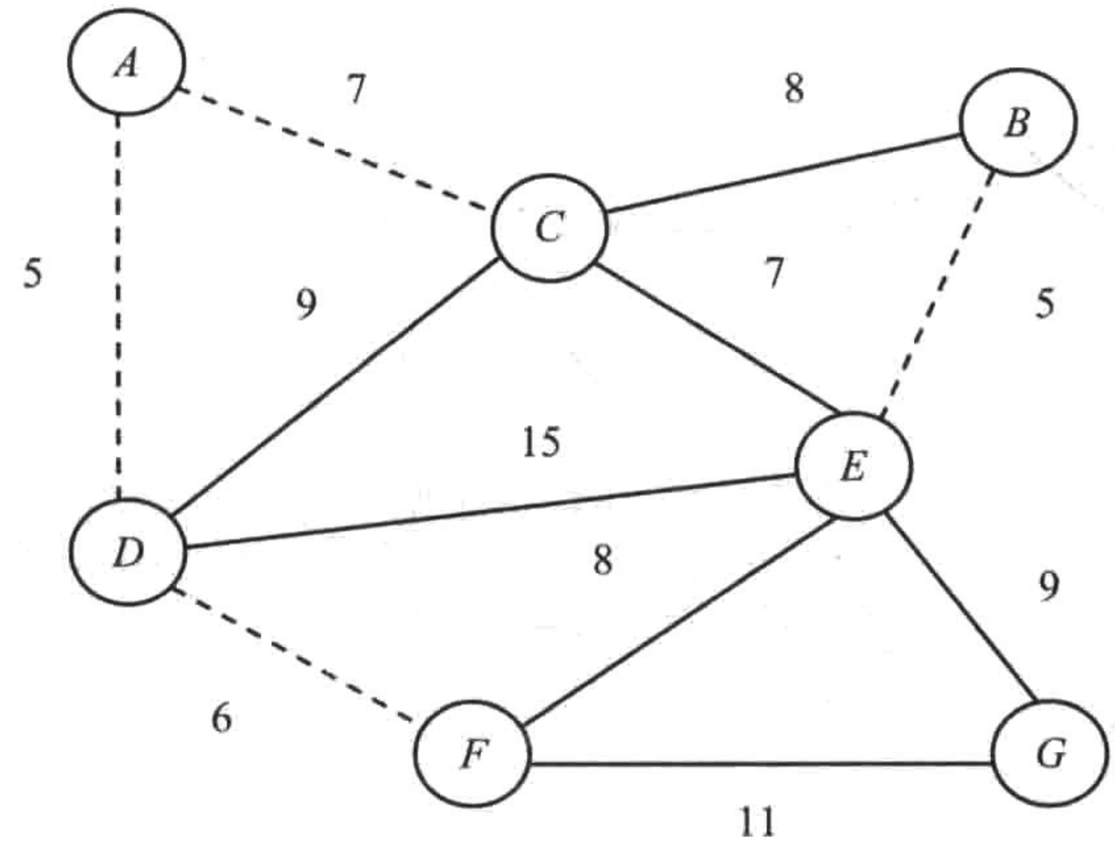

从上图可以看出，权重最小的边（开销）是*AD*和*BE*。两者之间可以任选一个。假设选择的是*AD*（图中的虚线边）



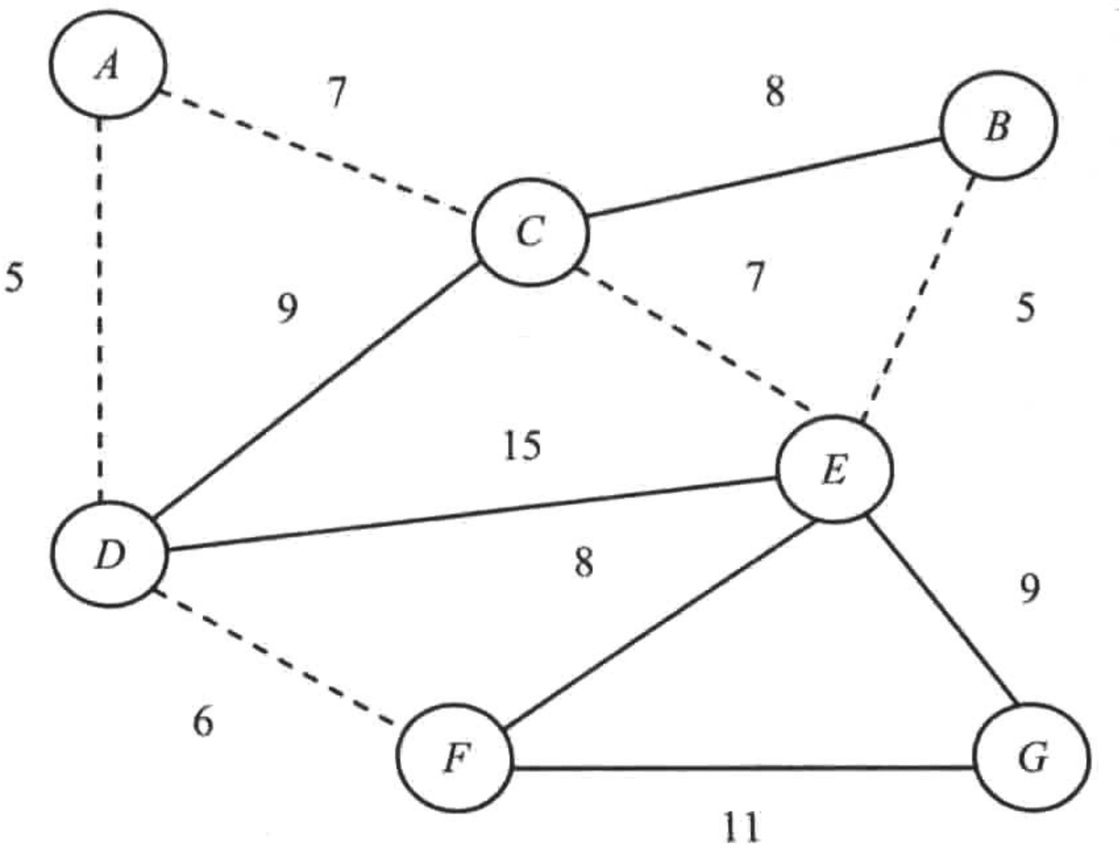

*DF*是下一条权重（6）最小的边



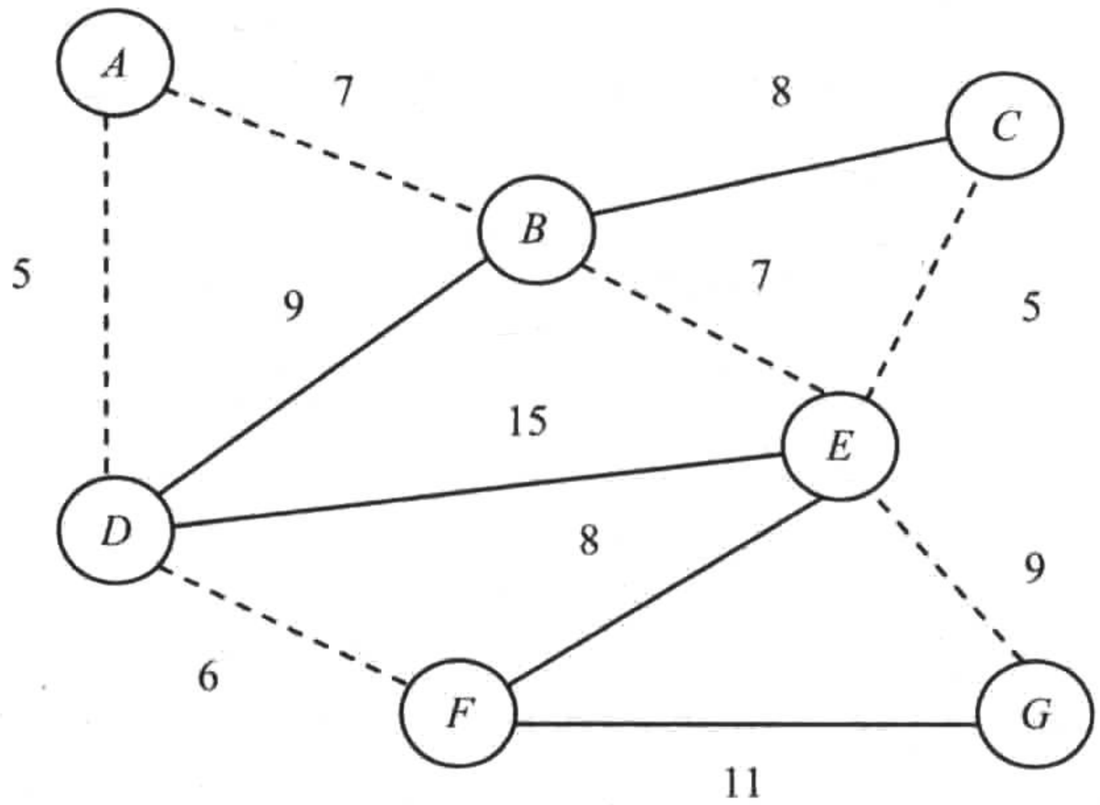

现在*BE*在所有边中有最小的权重，因此选择它（虚线表示选选中的边）




接下来，*AC*和*CE*具有的权重（7）最小，假设选择的是*AC*




选择*CE*，因为它不仅具有的权重（7）最小并且也没有形成环




接下来，权重最小的边是*CB*和*EF*，但是如果选择*CB*，将形成环，因此放弃这条边。*EF*同样如此。因此不能选择这两条边。接下来，较小的权重是9（*BD*和*EG*）。选择*BD*会形成环，所以放弃这边条。加入*EG*后不会形成环，并且在加入这条边后，生成树就包含了图中的所有顶点

```
void Kruskal(Graph G) {
        //参考第8章
        S = ф;  //最终S将包含所有最小生成树的边
        for (int v = 0; v< G→V; v++)
            MakeSet (v);
        Sort edges of E by increasing weights w;
        for each edge (u, v) in E { //来自有序表
                if(FIND (u) != FIND (v)) {
                        S = S ∪ {(u, v)};
                        UNION (u, v);
                }
        }
        return S;
}
```

**注意：** 关于 UNION 和 FIND 操作的具体实现请参考第 8 章。

在最坏情况下这个算法的时间复杂度是 $O(E\log E)$，主要是由堆操作所决定的。换言之，构造包含 $E$ 条边的堆需要的时间为 $O(E\log E)$。

## 9.9 图算法的相关问题

**问题 1** $n$ 个顶点的无向简单图具有的最大边数是多少？假设不允许自环。

**解答：** 由于每个结点能连接其他所有结点，所以第一个结点能连 $n-1$ 个结点。第二个结点能连 $n-2$ 个结点(因为一条边已经与第一个结点连接)。边的总数是：$1+2+3+\cdots+n-1=n(n-1)/2$ 条边。

**问题 2** 含有 $n$ 个顶点和 $E$ 条边的图有多少个不同的邻接矩阵？

**解答：** 图中所有顶点的排列数，即 $n!$。

**问题 3** $n$ 个顶点和 $E$ 条边的图有多少个不同的邻接表？

**解答：** 图中所有边的排列数，即 $E!$。

**问题 4** 对于无向图来说，采用哪种方式判断某个顶点是否是孤立点(即不与图中任何其他顶点有边连接)是最合适的？

**解答：邻接表。** 如果使用邻接矩阵，那么需要检查整行才能确定某个顶点是否有边。使用邻接表进行判断则非常容易，只需要检查顶点指向下一个元素的指针是否为 NULL 即可(NULL 表示顶点没有连接到其他任何顶点)。

**问题 5** 判断从源点 $s$ 到目标顶点 $t$ 是否存在一条路径，并查集和 DFS 哪个更合适？

**解答：** 并查集与 DFS 之间的区别如下表所示。表中条目代表任意一对结点(即 $s$ 和 $t$)的情况。

| 方法 | 处理时间 | 查询时间 | 空间 |
|---|---|---|---|
| 并查集 | $V+E\log V$ | $\log V$ | $V$ |
| DFS | $E+V$ | 1 | $E+V$ |

**问题 6** 具有 $n$ 个顶点且不含有向环的有向图的最大边数是多少？

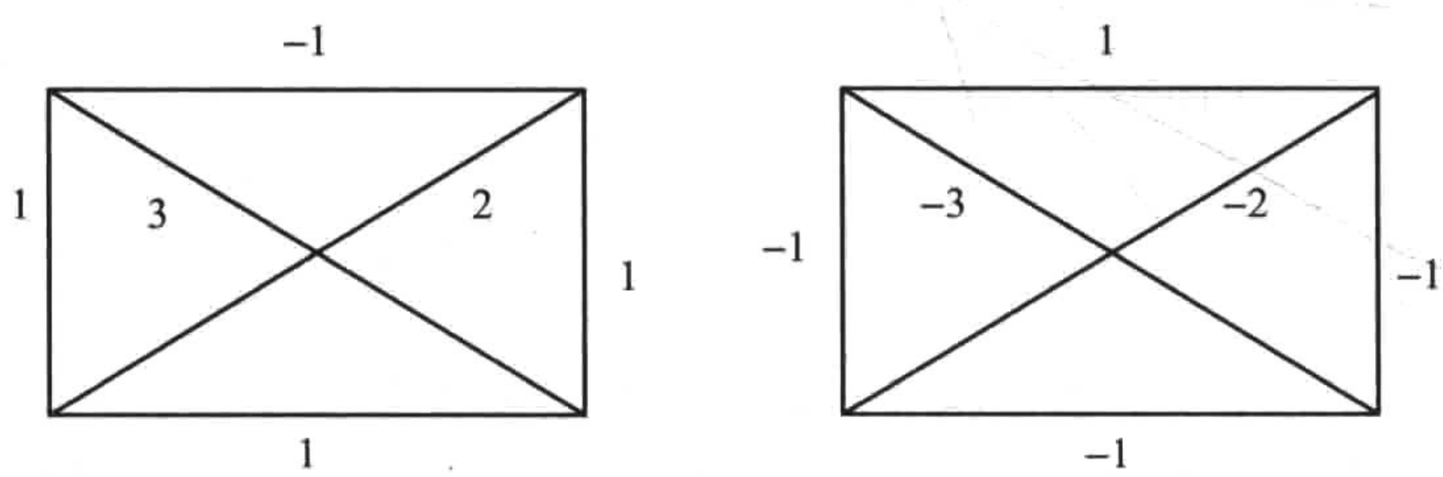
**解答：** $V(V-1)/2$。任何有向图最多有 $n^2$ 条边。然而因为该图没有自环，所以任意顶点对($x$，$y$)最多有($x$，$y$)和($y$，$x$)两条边。因此，边的总数最多为$(V^2-V)/2$。有向图具有 $V(V-1)/2$ 边条是可能的。首先将 $n$ 个结点标记为：1，2，…，$n$。当且仅当 $x<y$ 时，加入一条边($x$，$y$)。以这种方式生成的图就具有要求数目的边，并且不包含环(任何一条路径都是由一组顺序递增的结点组成)。

**问题 7** 对于 $V$，有多少个可能的不带平行边和自环的简单有向图？

**解答：** $(V)\times(V-1)$。因为不计算自环，所以每个顶点能连接 $V-1$ 个顶点。

**问题 8** 本章已经讨论过最小生成树算法。现在，能给出查找最大生成树的算法吗？

**解答：**




给定的图　　转换后带有负权重的图

利用给定图中的顶点和边构造一个新图。用边权重的负值来代替原来的值，即新图中边的权重＝给定图中对应边权重的负值。随后，在新图上应用已有的最小生成树算法。最后得到生成树就是原图的最大生成树。

**问题 9** DFS 与 BFS 的区别是什么？

**解答：**

| DFS | BFS |
|---|---|
| 可以从末端回溯 | 不能回溯 |
| 遍历过程中访问的顶点是以 LIFO 顺序处理的 | 待访问的顶点是在 FIFO 队列中存储的 |
| 按某种特定的方向进行遍历 | 在遍历过程中同一层的顶点是并行的 |

**问题 10** 给出一个算法，判定在给定图 $G$ 中，是否存在一条从源点 $s$ 到终点 $d$ 的简单路径。设图 $G$ 采用邻接矩阵来表示。

**解答：** 对每个顶点调用 DFS，判断当前顶点是否是终点。若两个顶点相同，返回 1。否则，调用 DFS 遍历当前顶点未访问过的邻居结点。需要注意的是，只对还未访问过的顶点调用 DFS 算法。

```
void HasSimplePath(Graph G, int s, int d) {
        Viisited[s] = 1;
        if(s == d) return 1;
        for(int t = 0; t < G.vertexCount; t++) {
                if(G.adjMatrix[s][t] && !Viisited[t])
                        if(DFS(G, t, d))  return 1;
        }
        return 0;
}
```

时间复杂度为 $O(E)$。在上述算法中，对于每个结点，不需要对它所有的邻居结点调用 DFS(通过 if 条件语句跳过)。空间复杂度为 $O(V)$。

**问题 11** 对于给定的图 $G$，统计从源点 $s$ 到终点 $d$ 的简单路径数。设图 $G$ 采用邻接矩阵表示。

**解答：** 与问题 10 类似，从某个结点开始调用 DFS。调用的结果是，算法能够遍历给定图中所有可达顶点。即算法遍历了起始结点所在连通分量内的所有结点。如果仍然还有未访问的结点，则重新从未访问的结点中选择一个结点并调用 DFS。在每个连通分量中第一次调用 DFS 前，对连通分量的计数加 1。重复该过程直至图中所有结点都访问过。最终就能得到所有连通分量的数目。基于该策略的算法实现如下：

```
void CountSimplePaths(Graph G, int s, int d) {
        Viisited[s] = 1;
        if(s == d) {
                count++;
                Visited[s] = 0;
                return;
        }
        for(int t = 0; t < G.vertexCount; t++) {
                if(G.adjMatrix[s][t] && !Viisited[t]){
                        DFS(G, t, d);
                        Visited[t] = 0;
                }
        }
}
```

**问题 12** **所有顶点对的最短路径问题：** 查找给定图中每对顶点间的最短距离。假设给定图中不含负边。

**解答：** 使用 $n$ 次 Dijkstra 算法可以解决该问题。也就是说，对图中的每个顶点调用一次 Dijkstra 算法。如果图有负边，则该算法不适用。

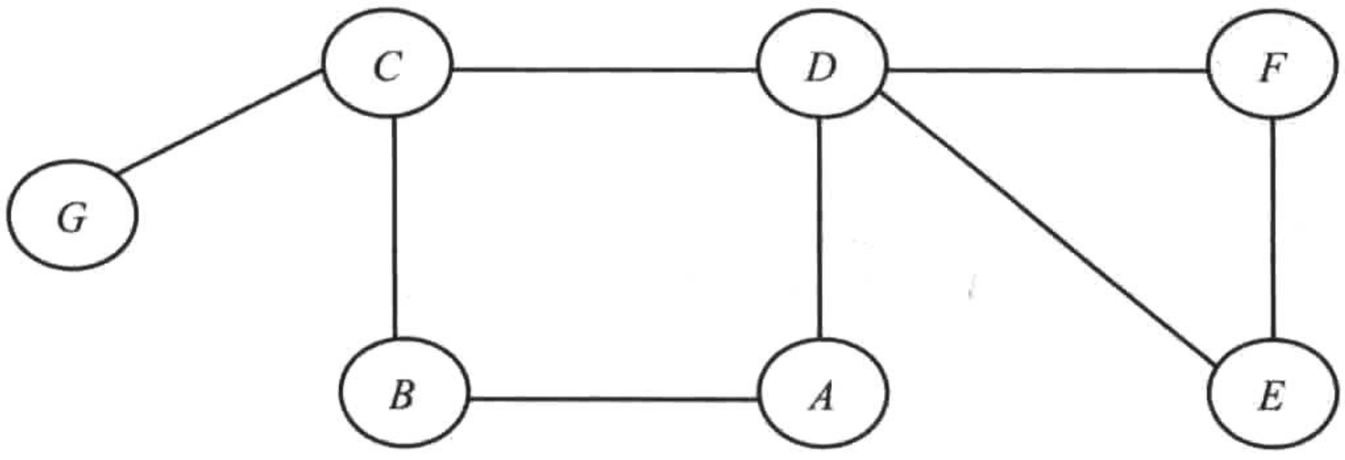
**问题 13** 在问题 12 中，如果图有负边，怎样计算所有顶点对间的最短路径呢?

**解答：** 可以使用 Floyd-Warshall 算法来解决此问题。该算法是动态规划的一个实例，适用于有权图有负边的情况。请参见第 19 章。

**问题 14** **DFS 应用：** 割点或者关节点。

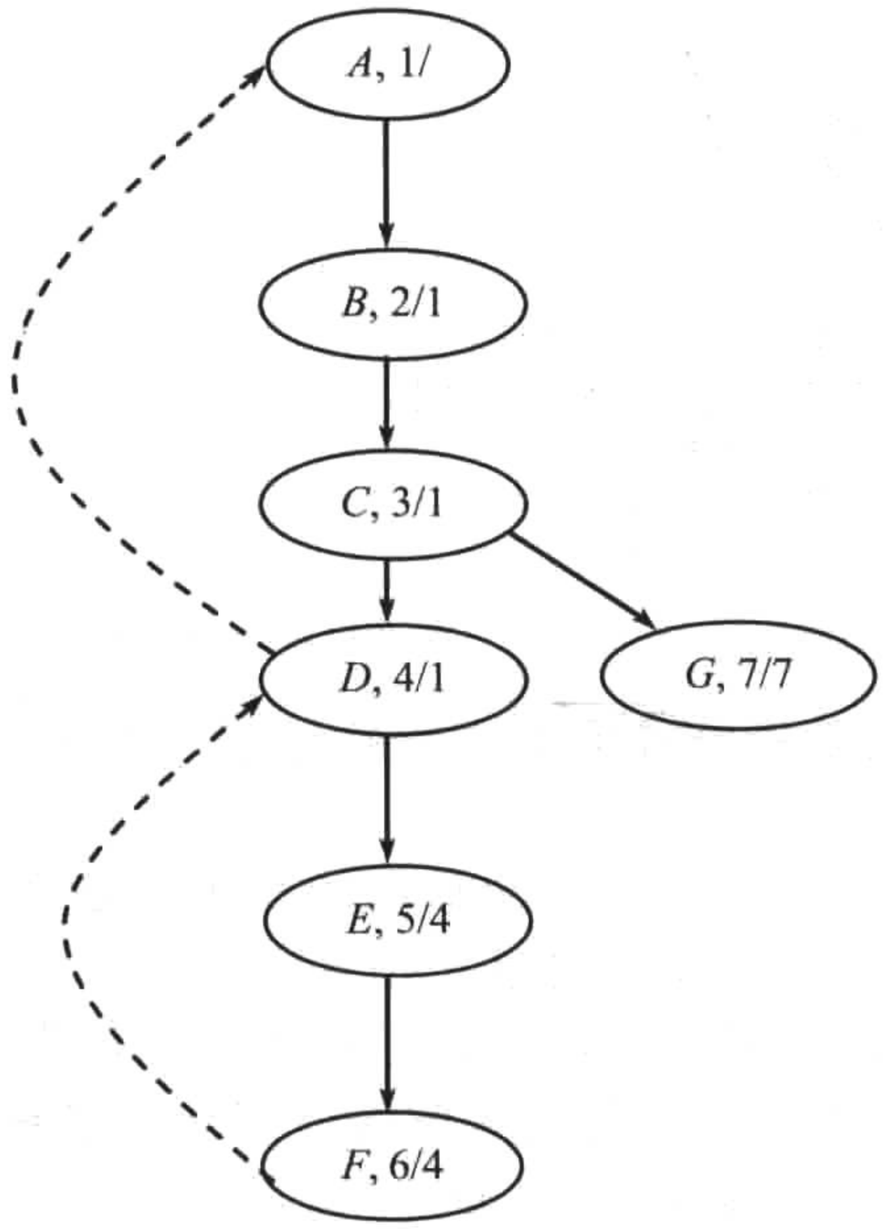
**解答：** 在无向图中，割点(或关节点)是这样一类顶点，即如果移除它，那么图将分割成两个不相连的子图。以下图为例，移除顶点 $D$ 将导致图被分成两个连通分量$\{E, F\}$和$\{A, B, C, G\}$。类似地，移除顶点 $C$ 将使图分成$\{G\}$和$\{A, B, D, E, F\}$。对于该图，$D$ 和 $C$ 就是割点。

**注意：** 如果一个无向连通图在移除任何一个顶点后仍然是连通的，则该图称作双连通图。




DFS 提供了一个在连通图中查找所有割点的线性时间算法 $O(n)$。从任意顶点开始，调用 DFS 并对访问过的结点编号。对于每个顶点 $v$，调用 DFS 获得的对应编号为 dfsnum($v$)。DFS 遍历所产生的树称作 DFS 生成树。那么，对于 DFS 生成树中的每个顶点 $v$，计算编号最小的顶点，称作 low($v$)，即从 $v$ 点经过 0 条或多条树边及某条可能的回退边到达该点。

基于上面的讨论，需要如下信息来运行算法。DFS 树中每个顶点的 dfsnum(当该 DFS 树被访问时)。对于每个顶点 $v$，它在 DFS 树中所有子孙的邻居结点的最小深度，称作 low。在调用 DFS 的过程中可以计算 dfsnum。在 $v$ 的所有子孙被访问后(刚好在 $v$ 弹出 DFS 栈前)，可以计算 low($v$)，并作为 $v$ 的所有邻居结点(而不是在 DFS 树中 $v$ 的父结点)的 dfsnum 的最小值，以及 DFS 树中 $v$ 的所有孩子结点的 low 值。

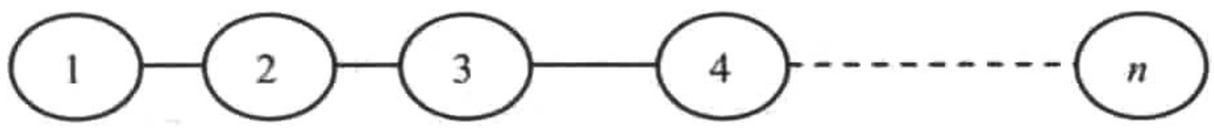



根结点是一个割点当且仅当它至少有两个孩子结点。一个非根顶点 $u$ 是割点当且仅当有一个 $u$ 的孩子结点 $v$ 满足 low($v$)≥dfsnum($u$)。当 DFS 从 $u$ 的每个孩子结点返回时(也就是恰好 $u$ 弹出 DFS 栈时)，能够对该属性进行测试。如果为 true，则 $u$ 把图分割成不同的双连通分量。可以通过首先寻找每个这样 $v$ 结点所产生的一个双连通分量(一个含 $v$ 的连通分量将包含 $v$ 的子树和 $u$)，接着从该树中删除 $v$ 的子树。对于给定的图，包含 dfsnum/low 的 DFS 树如上图所示。算法的实现过程如下：

```
int adjMatrix [256] [256] ;
int dfsnum [256], num = 0, low [256];
void CutVertices( int u ) {
        low[u] = dfsnum[u] = num++;
        for (int v = 0 ; v < 256; ++v ) {
                if(adjMatrix[u][v] && dfsnum[v] == -1) {
                        CutVertices( v ) ;
                        if(low[v] > dfsnum[u])
                                System.out.println("Cut Vetex:" + u);
                        low[u] = min ( low[u] , low[v] ) ;
                }
                else    // (u,v)是一条回退边
                        low[u ] = min(low[u] , dfsnum[v]) ;
        }
}
```

**问题 15** 假设图 $G$ 是一个含有 $n$ 个顶点的连通图。图 $G$ 所包含的最大割点数是多少?

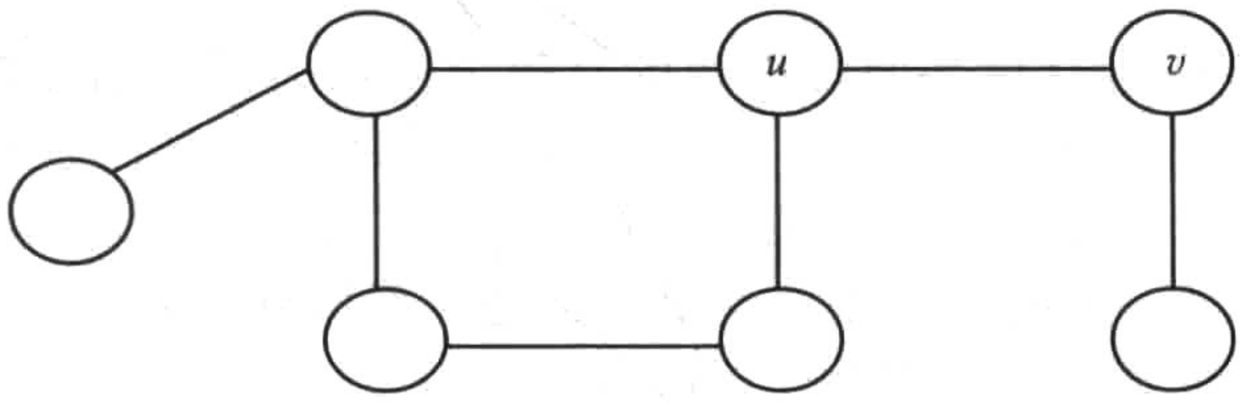
**解答:** $n-2$。以下图为例。在下图中，除了顶点 1 和 $n$ 外，所有其他顶点都是割点。这是因为移除顶点 1 和 $n$ 不能把图分割成两部分。这是能得到最大割点数的情况。




**问题 16** DFS 应用：割桥或割边。

**解答:**

**定义:** 假设 $G$ 为连通图。$uv$ 是图 $G$ 中的边，如果移除 $uv$ 后 $G$ 不连通，则边 $uv$ 称作 $G$ 的一个桥。考虑如下所示的图。




在上图中，如果移除边 $uv$，那么图分割成两个连通分量。对于这个图，$uv$ 是一个桥。前面对割点的讨论同样适用于桥。唯一的区别是用边来代替顶点。需要注意的是，如果边($u$，$v$)是某个回路的一部分，那么它就不是桥。若边($u$，$v$)不是回路的一部分，则它是桥。

利用回退边可以检测 DFS 中的环。边($u$，$v$)是桥当且仅当 $v$ 和 $v$ 的孩子中不存在到 $u$ 或 $u$ 的祖先的回退边。为了判断 $v$ 的孩子是否有到 $u$ 的双亲的回退边，可以采用与前面类似的方法来计算以 $v$ 为根结点的子树可到达的最小的 dfsnum。

```
int dfsnum[256], num = 0, low [256];
void Bridges( Graph G, int u ) {
        low[u] = dfsnum[u] = num++;
        for (int v = 0 ; G.vertexCount; ++v ) {
                if(G.adjMatrix[u][v] && dfsnum[v] == -1) {
                        cutVertices( v ) ;
                        if(low[v] > dfsnum[u])
```

```
                        print (u,v)  as a bridge
                    low[u] = min ( low[u] , low[v] ) ;
                }
                else    // (u,v) 是一条回退边
                    low[u ] = min(low[u] , dfsnum[v]) ;
        }
}
```

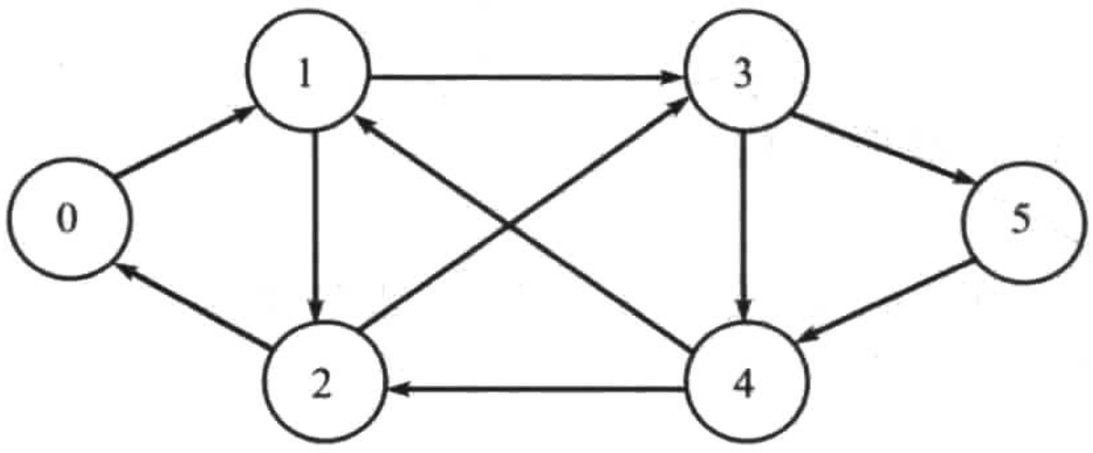
**问题 17　DFS 应用：讨论欧拉回路。**

**解答：**在讨论该问题前，先给出相关术语：

- 欧拉路径：一条包含所有边且无重复边的路径。
- 欧拉回路：一条包含所有边且无重复边的路径，并且起点和终点都是同一个点。
- 欧拉图：包含欧拉回路的图。
- 偶点：具有偶数条依附边的点。
- 奇点：具有奇数条依附边的点。

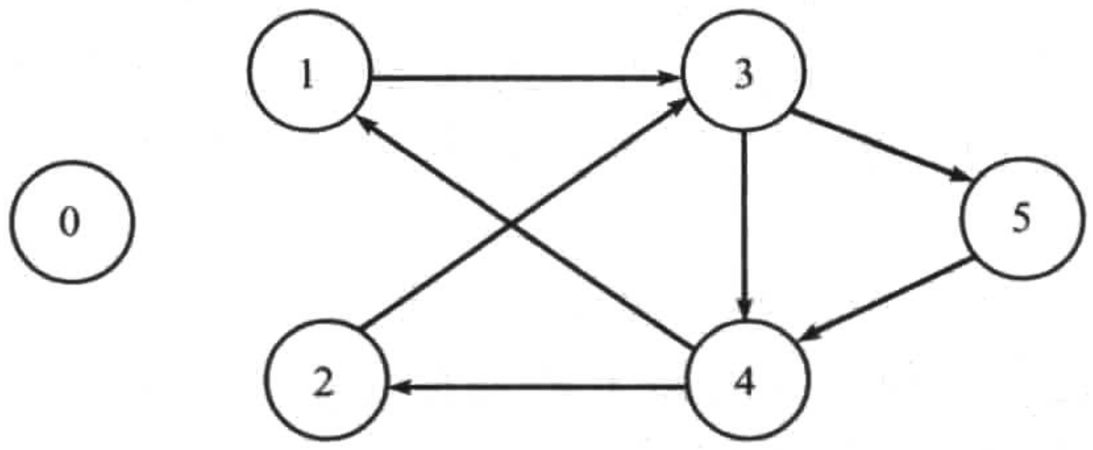
欧拉回路：给定一个图，使用一支笔来重画它，使得每条线恰好只画一次。画的时候笔始终不离开纸。也就是说，需要在图中找到这样一条路径，该路径恰好访问每条边一次，这个问题称为欧拉路径或欧拉通路问题。

这个难题有较简单的基于 DFS 的解决方案。

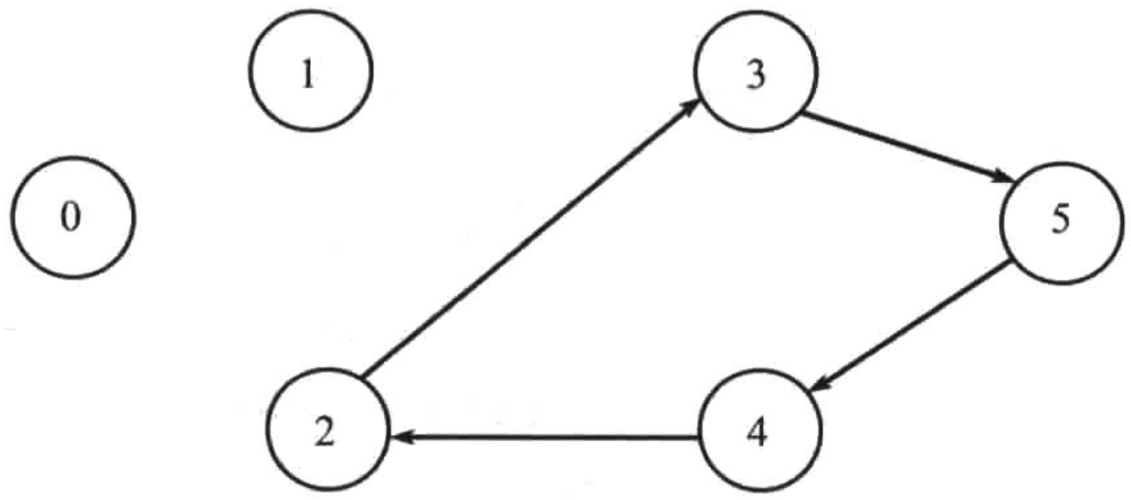



一条欧拉回路存在当且仅当图是连通的且每个顶点的邻居结点数为偶数。从任意顶点开始，选择一条尚未访问过的出边，并沿着该边遍历。重复该过程直至不存在未选择的出边。以下图为例：一条合理的欧拉回路是 0 1 3 4 2 3 5 4 2 0。

如果从顶点 0 开始，可选择一条到顶点 1 的边，然后选择到顶点 2 的边，最后选择到顶点 0 的边。因此，从顶点 0 开始不存在尚未选择的边：




现在有一个回路 0，1，2，0，但没有遍历每条边。所以，继续选择回路上的某些顶点，比如 1。然后对剩余的边调用 DFS。假如选择到顶点 3 的边，然后是 4，最后是 1。由于从结点 1 开始不存在尚未选择的边，所以遍历过程再次终止。现在将这条路径 1，3，4，1 拼接到原有的路径 0，1，2，0 中得到：0，1，3，4，1，2，0。那么，尚未选择的边如下图所示。




继续选择另一个顶点调用 DFS。如选择顶点 2，并将路径 2，3，5，4，2 与原有路径进行拼接得到最终的回路 0，1，3，4，1，2，3，5，4，2，0。一个相似的问题是在无向图中寻找一遍历图中每个顶点的简单环，即哈密顿回路问题。尽管该问题看上去与欧拉回路问题几乎相同，但到目前为止还没有有效解决该问题的算法。

**注意：**

- 一个连通无向图是欧拉图当且仅当每个顶点有偶数条边，或者恰好有两个顶点有奇数条边。
- 一个有向图是欧拉图，如果它是强连通的且每个顶点的入度和出度相同。

**应用：** 邮递员为了送信和包裹必须访问一组街道。他必须找到这样一条路径：路径的起点和终点都是邮局，并且经过每条街道(边)恰好经过一次。通过这条路径，邮递员能够将送信和包裹所有必须经过的道路遍历一次且在路上花费的开销/时间最小。

**问题 18　DFS 应用：** 查找强连通分量。

**解答：** 这是 DFS 的另一个应用。在有向图中，两个顶点 $u$ 和 $v$ 是强连通的当且仅当同时存在从 $u$ 到 $v$ 和从 $v$ 到 $u$ 的路径。强连通性是一个等价关系。

- 一个顶点与自身是强连通的。
- 如果顶点 $u$ 与顶点 $v$ 是强连通的，那么 $v$ 与 $u$ 也是强连通的。
- 如果顶点 $u$ 与顶点 $v$ 是强连通的，且 $v$ 与顶点 $x$ 是强连通的，那么 $u$ 与 $x$ 也是强连通的。

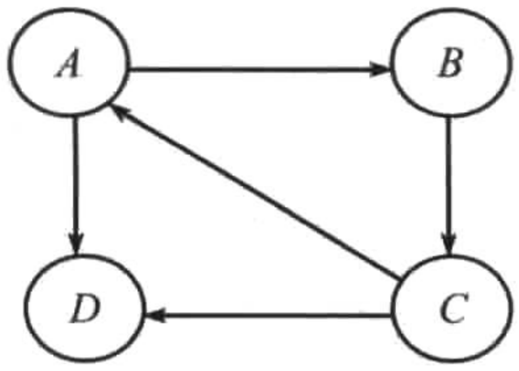
对于给定的有向图，能够将其分成多个强连通分量。这个问题能通过两个深度优先搜索来解决。使用两个 DFS 搜索，能够判定一个给定的有向图是否是强连通的。算法也能够得到强连通顶点的集合。

**算法：**

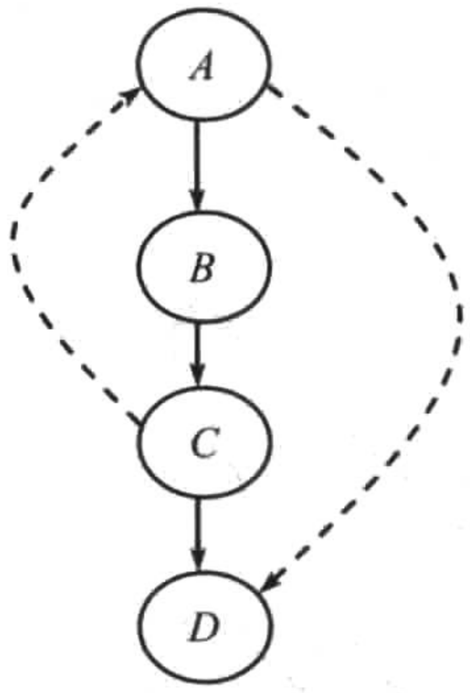
- 在给定图 $G$ 上执行 DFS。
- 根据深度优先生成森林的后序遍历，对给定图 $G$ 中的顶点编号。
- 反转 $G$ 中的所有边，构造图 $G_r$。
- 在 $G_r$ 上调用 DFS：总是从编号最高的顶点开始调用新的 DFS。
- 深度优先生成森林中的每棵树对应一个强连通分量。

**为什么该算法起作用**

考虑两个顶点 $v$ 和 $w$。如果两个顶点在同一个强连通分量中，则在原始图 $G$ 中有从顶点 $v$ 到 $w$ 和 $w$ 到 $v$ 的路径，并且在 $G_r$ 中也有这样的两条路径。如果两个顶点 $v$ 和 $w$ 不在同一个 $G_r$ 的深度优先生成树中，则说明这两个顶点不在同一个强连通分量中。以下图 $G$ 为例。



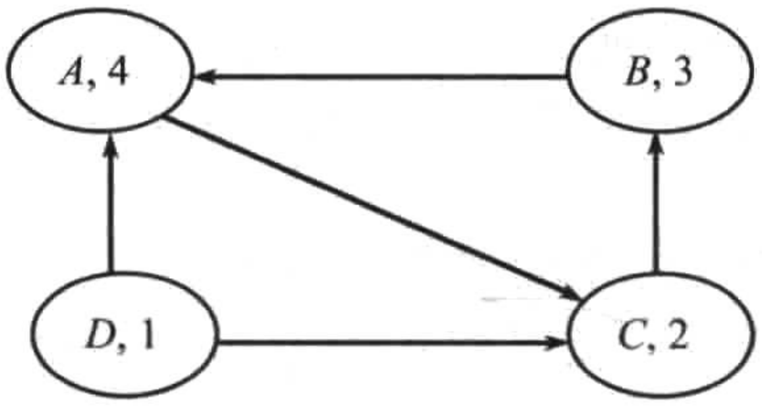

根据算法，首先在图 $G$ 上调用 DFS。虚线表示从 $C$ 到 $A$ 的一条回退边。



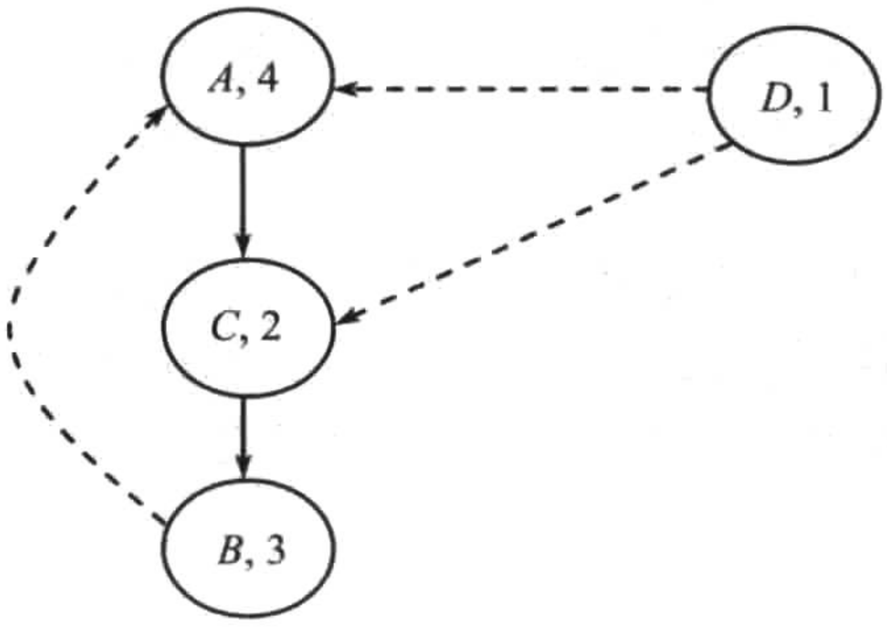

在该树上执行后序遍历可得：$D$，$C$，$B$，$A$。

| 顶点 | 后序编号 | 顶点 | 后序编号 |
|---|---|---|---|
| $A$ | 4 | $C$ | 2 |
| $B$ | 3 | $D$ | 1 |

现在反转给定图 $G$ 得到 $G_r$。同时，将得到的后序遍历的编号赋值给图中的顶点。反转图 $G_r$ 将是这样：




最后，在反转图 $G_r$ 上调用 DFS。当执行 DFS 时，需要考虑具有最大 DFS 编号的顶点。所以，首先从 $A$ 开始，根据 DFS 先遍历到 $C$，然后是 $B$。到达 $B$ 后无法继续遍历。这就是说，$\{A, B, C\}$ 是强连通图。现在仅剩余顶点 $D$，那么第二次 DFS 调用即终止于 $D$ 本身。所以强连通分量是 $\{A, B, C\}$ 和 $\{D\}$。



基于上面的讨论，算法实现如下：

```
//图以邻接矩阵表示
int adjMatrix [256][256], table[256];
vector <int> st ;
int counter = 0 ;
//这个表包含DFS搜索编号
int dfsnum [256], num = 0, low[256] ;
void StronglyConnectedComponents( int u ) {
        low[u] = dfsnum[ u ] = num++;
        Push(st, u) ;
        for( int v = 0 ; v < 256; ++v ) {
                if(graph[u][v] && table[v] == -1) {
                        if( dfsnum[v] == -1)
                                StronglyConnectedComponents(v) ;
                        low[u] = min(low[u] , low[v]) ;
                }
        }
        if(low[u] == dfsnum[u]) {
                while( table[u] != counter) {
                        table[st.back()] = counter;
                        Push(st) ;
                }
                ++ counter;
        }
}
```

**问题 19** 用邻接矩阵表示图 $G$，统计图 $G$ 中的连通分量的个数。

**解答：** 可在 DFS 中增加一个额外的计数器来解决此问题。

```
//Visited[]是一个全局数组
int Visited[G→V];
void DFS(Graph G, int u) {
        Visited[u] = 1;
        for( int v = 0; v < G.vertexCount; v++ ) {
                /*例如,如果邻接矩阵用来表示这个图，那么用来查找u的未被访问的邻接点的
                  条件是: if( !Visited[v] && G→Adj[u][v] ) */
                for each unvisited adjacent node v of u {
                        DFS(v);
                }
        }
}
void DFSTraversal(Graph G) {
        int count = 0;
        for (int i = 0; i< G.vertexCount;i++)
                Visited[i]=0;
        //当图具有不止一个连通分量时需要此循环
        for (int i = 0; i< G.vertexCount;i++)
                if(!Visited[i]) {
                        DFS(G, i);
                        count++;
                }

        return count;
}
```

时间复杂度：与 DFS 相同，依赖于图的实现方式。采用邻接表时复杂度是 $O(|E|+|V|)$，采用邻接矩阵时复杂度是 $O(|V|^2)$。

**问题 20** 能否用 BFS 解决问题 19？

**解答：可以。**可以通过在 BFS 中增加一个额外的计数器来解决该问题。

```
void BFS(Graph G, int u) {
        int v,
        LLQueue Q = new LLQueue();
        Q.enQueue( u);
        while(!Q.isEmpty()) {
                u = Q.deQueue();
                Process u; //例如，print
                Visited[s]=1;
                /*例如，如果采用邻接矩阵表示图，那么用来查找u的未被访问的邻接点
                的条件是: if( !Visited[v] && G→Adj[u][v] ) */
                for each unvisited adjacent node v of u {
                        Q.enQueue( v);
                }
        }
}
void BFSTraversal(Graph G) {
        for (int i = 0; i< G.vertexCount;i++)
                Visited[i]=0;
        //当图具有不止一个连通分量时需要此循环
        for (int i = 0; i< G.vertexCount; i++)
                if(!Visited[i])
                        BFS(G, i);
}
```

时间复杂度：与 BFS 相同，依赖于图的实现方式。采用邻接表时复杂度是 $O(|E|+|V|)$，采用邻接矩阵时复杂度是 $O(|V|^2)$。

**问题 21** 假设 $G(V, E)$是一个无向图。给出一个算法来查找生成树(不一定是最小生成树)，要求算法的时间复杂度为 $O(|E|)$。

**解答：**通过给集合 $S$ 的顶点加标记，可以在常数时间内完成对回路的检测。如果一条边的两个顶点都被标记，则该边将产生一个环。

**算法：**

```
S = {}; //假设S是集合
for each edge e ∈ E {
        if(adding e to S doesn't form a cycle) {
                add e to S;
                mark e;
        }
}
```

**问题 22** 有无其他方法解决问题 20？

**解答：有。**可以调用 BFS 算法来查找图的 BFS 树(图的层次树)。然后从根结点开始遍历下一层，同时在遍历的过程中只访问下一层中的结点一次。也就是说，如果某个结点有多条输入边，那么仅考虑其中一条边，否则会形成环。

**问题 23** 检测无向图中的环。

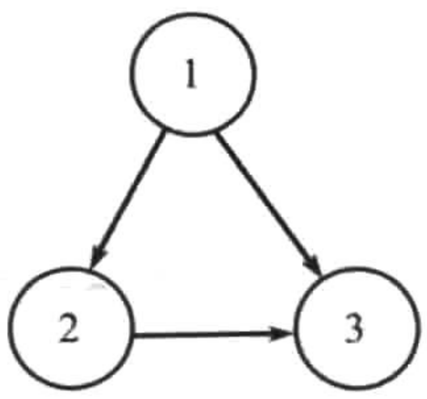
**解答：**一个无向图是无环的当且仅当 DFS 不产生回退边，而且在边$(u, v)$中 $v$ 是 $u$ 的祖先且已经被访问过。

- 在图上执行 DFS。
- 如果有回退边，那么图存在环。

如果图不包含环，那么$|E|<|V|$且DFS的时间开销为$O(|V|)$。如果图包含环，那么最多在$2|V|$步后就能够发现回退边。

**问题24** 检测DAG中的环。

**解答：** 在图中检测环与在树中不同，这是由于在图中一个结点可以有多个父亲。在树中，可以采用DFS算法来检测环，并对遍历过的结点进行标记。如果在遍历过程中遇到已经标记过的结点，则表明树中存在环。然而，该算法不适用于图。考虑如下所示的图。如果使用树的环检测算法，那么它将返回错误的结果，即认为该图存在环。但实际上这个图中并没有环。这是因为从结点1开始遍历将访问结点3两次。




可以对树的环检测算法进行简单的改进后使其适用于图。在一个无环图中调用DFS，某个结点的所有被访问过的子孙都有可能被再次访问，但这并不代表图中存在环。但是，如果当某个结点在其所有子孙被访问前就被第二次访问，那么图中必定存在环。

这是为什么呢？假设某个环包含结点$A$。这意味着从$A$的某个子孙可以到达$A$。当DFS访问该子孙且在完成对$A$的所有子孙访问前，它再次遍历到$A$，则表明图中存在环。因此，可以修改深度优先搜索算法来检测图中的环。

```
int DetectCycle(Graph G) {
        for (int i = 0; i< G.vertexCount; i++) {
                Visited[s]=0;
                Predecessor[i] = 0;
        }
        for (int i = 0; i < G.vertexCount;i++) {
                if(!Visited[i] && HasCycle(G, i))
                        return 1;
        }
        return false;;
}
int HasCycle(Graph G, int u) {
        Visited[u]=1;
        for (int i = 0; i< G.vertexCount; i++) {
                if(G→Adj[s][i]) {
                        if(Predecessor[i] != u && Visited[i])
                                return 1;
                        else {
                                Predecessor[i] = u;
                                return  HasCycle(G, i);
                        }
                }
        return 0;
}
```

时间复杂度为$O(V+E)$。

**问题25** 给定一个有向无环图，请给出算法来查找其深度。

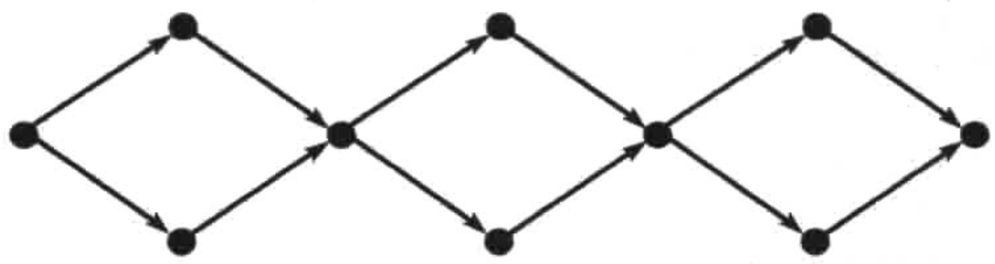
**解答：** 可以采用与查找树的深度相似的方法来解决此问题。在树中，可以通过层次

遍历来确定树的深度(需要一个额外的特定符号来表示层的结束)。

```
//假定给定的图是DAG图
int DepthInDAG( Graph G ) {
        LLQueue Q = new LLQueue();
        int counter;
        int v, w;
        counter =  0;
        for (v = 0; v< G.vertexCount; v++)
                if( indegree[v] ==   0 )
                        Q.enQueue(v);
        Q.enQueue( '$' );
        while( !Q.isEmpty()) {
                v = Q.deQueue();
                if(v == '$') {
                        counter++;
                        if(!Q.isEmpty())
                                Q.enQueue( '$' );
                }
                for each w adjacent to v
                        if( --indegree[w] ==  0 )
                                Q.enQueue(w );
        }
        Q.deleteQueue();
        return counter;
}
```

总的时间复杂度是 $O(V+E)$。

**问题 26**　下面的有向无环图有多少个拓扑排序？



**解答：**观察上图可以发现，该图共分为 3 个阶段，其中每个阶段包含两个顶点。由本章前面的讨论可知，在任何时间拓扑排序都是选取入度为 0 的顶点。在每一个阶段，可以选择顶部顶点或底部顶点中的任意一个。因此，在这三个阶段的每一个阶段都有两种可能性。所以总的可能性为每个阶段所含有的可能性的乘积：$2\times2\times2=8$。

**问题 27　唯一拓扑次序：**设计算法来判定有向图是否具有唯一拓扑次序。

**解答：**有向图有唯一拓扑次序当且仅当以拓扑次序排列的连续顶点对间存在一条有向边。也可以按如下方式定义：有向图有唯一拓扑次序当且仅当该图存在一条哈密顿路径。如果有向图有多个拓扑次序，那么可以通过交换连续顶点对来获得第二个拓扑次序。

**问题 28**　以印度理工学院孟买分校课程的选修条件为例。假设所有选修条件都是强制的，每个学期会开设全部课程，并且不限定每个学期可修的课程门数。如何确定完成该专业所需的最少的学期数。描述该问题所需要的数据结构，并给出一个线性时间算法来解决此问题。

**解答：**可以使用有向无环图 DAG 来表示该问题。其中顶点表示课程，边表示印度理工学院孟买分校课程间的选修关系。因为选修关系不存在环，所以采用有向无环图。

完成专业所需的学期数是 DAG 图中最长路径的长度加 1。可以通过在线性时间内递归地计算 DFS 树来实现。如果 $x$ 的出度为 0，则以顶点 $x$ 为起始点的最长路径长度为 0，

否则是 1+max{以 $y$ 为起始点的最长路径的长度|($x$，$y$)为 $G$ 的边}。

**问题 29** 在某所大学(比如，印度理工学院孟买分校)，有一个课程列表及其选修的条件，即有如下两个列表：

$A$——课程列表

$B$——选修条件：$B$ 包含由来自 $A$ 中的课程(即 $x$，$y\in A$)组成的($x$，$y$)对，且课程 $x$ 必须在课程 $y$ 之前选修。

假设某个学生在一个学期内只想选修一门课程。为该学生设计课程安排表。

**例子**：$A$={C 语言，数据结构，操作系统，计算机组成，算法，设计模式，程序设计}。$B$={(C 语言，计算机组成)，(操作系统，计算机组成)，(数据结构，算法)，(设计模式，程序设计)}。一个可能的课程安排表如下所示：

第一学期：数据结构。

第二学期：算法。

第三学期：C 语言。

第四学期：操作系统。

第五学期：计算机组成。

第六学期：设计模式。

第七学期：程序设计。

**解答**：该问题的解和拓扑排序的解是完全相同的。假设课程名字由{1..$n$}的整数表示，$n$ 为已知的，但不是常量。课程间的关系可用有向图 $G=(V，E)$来表示，其中 $V$ 是课程集合。如果课程 $i$ 是课程 $j$ 的先修课程，则 $E$ 将包含边($i$，$j$)。假如用邻接表来表示图。

下面讨论在 DAG 中时间复杂度是 $O(|V|+|E|)$的拓扑排序算法。

- 查找所有顶点的入度——$O(|V|)+|E|)$。
- 重复：查找入度为 0 的顶点 $v$——$O(|V|)$。
  输出 $v$，并将该结点从 $G$ 中移除，沿着它的边——$O(|V|)$。
  如果($v$，$u$)是 $G$ 的一条边，顶点 $u$ 的入度减 1，并用列表来保存所有入度为 0 的顶点——$O(\text{degree}(v))$。

重复该过程直到所有顶点被移除。

该算法的时间复杂度与拓扑排序相同，为 $O(|V|+|E|)$。

**问题 30** 在问题 29 中，某个学生想用最少的学期完成 $A$ 中的所有课程，即学生能够在一个学期内选修任意门课程。在此情况下为学生设计课程安排表。一个可能的安排表是：

第一学期：C 语言，操作系统，设计模式。

第二学期：数据结构，计算机组成，程序设计。

第三学期：算法。

**解答**：稍微改动上述拓扑排序算法可解决此问题：在每个学期中，不仅选修一门课程中，而是选修所有入度为 0 的课程。即在所有度为 0 的结点上运行该算法(在每个阶段操作和输出所有的源点，而不仅仅是一个源点)。

时间复杂度为 $O(|V|+|E|)$。

**问题 31 DAG 的最近公共祖先(LCA)**：给出一个 DAG 图和两个顶点 $v$ 及 $w$，找出 $v$

和 $w$ 的最近公共祖先。$v$ 和 $w$ 的 LCA 是 $v$ 和 $w$ 的祖先，且没有任何子孙也是 $v$ 和 $w$ 的祖先。

**提示：** 定义 DAG 中顶点 $v$ 的高度为从根到 $v$ 的最大路径长度。在 $v$ 和 $w$ 的祖先顶点中，具有最大高度的顶点即为 $v$ 和 $w$ 的 LCA。

**问题 32　最短祖先路径：** 给定一个 DAG 和两个顶点 $v$ 及 $w$，找出 $v$ 和 $w$ 之间的最短祖先路径。一条 $v$ 和 $w$ 间的祖先路径是指一个 $v$ 与 $w$ 的公共祖先 $x$ 以及从 $v$ 到 $x$ 的最短路径和从 $w$ 到 $x$ 的最短路径。最短祖先路径即为长度最短的祖先路径。

**提示：** 调用 BFS 两次。第一次调用从 $v$ 开始，第二次调用从 $w$ 开始。查找一个 DAG 图，其中最短祖先路径包含公共祖先 $x$，且 $x$ 不是 $v$ 和 $w$ 的 LCA。

**问题 33**　假设有两个图 $G_1$ 和 $G_2$。怎样判断它们是否是同构的？

**解答：** 有许多方式可以表示同一个图。以下面的简单图为例。可以看到它们都有相同的顶点数和边数。



**定义：** 图 $G_1=\{V_1, E_1\}$ 和 $G_2=\{V_2, E_2\}$ 是同构的，如果满足：

1）有从 $V_1$ 到 $V_2$ 的一一对应关系。

2）存在 $E_1$ 到 $E_2$ 的一一对应关系，该关系将 $G_1$ 中的每条边映射到 $G_2$ 中。

对于给定的图，如何判断它们是否是同构的呢？

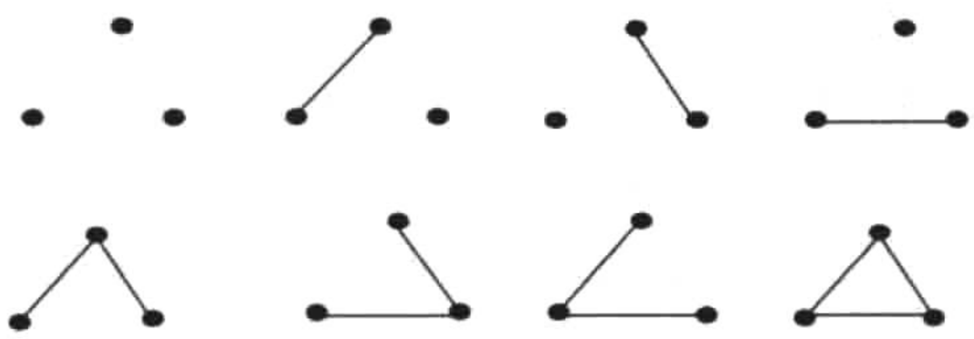
通常，很难证明两个图是同构的。因此可以考虑同构图的某些性质，即当图是同构时必须满足的性质。如果给定图不满足这些性质，则认为它们不是同构的。

性质：两个图是同构的当且仅当对于顶点的某些次序它们的邻接矩阵是相同的。

基于上面的性质，可以判断给定图是否是同构的。为了检查该性质，需要对矩阵进行转换操作。

**问题 34**　$n$ 个顶点可以有多少个简单无向非同构图？

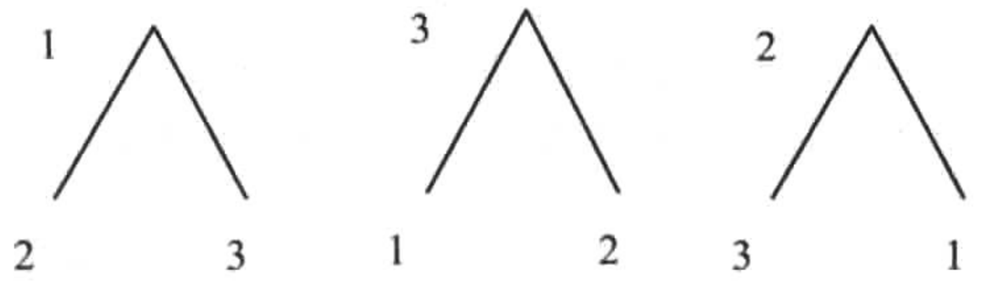
**解答：** 可以分两步来回答该问题。首先，统计所有带标记的图。假设下面所有表示方式都由{1，2，3}来标记顶点。$n=3$ 时所有图的集合是：



对每条边仅有两个选择，要么存在，要么不存在。因此，由于边的最大数是 $\binom{n}{2}$，（因为 $n$ 个顶点的无向图中的最大边数是 $n(n-1)/2=n_{c_2}=\binom{n}{2}$，）所以总的无向标记图数是：$2^{\binom{n}{2}}$。

**问题 35**　给定含 $n$ 个顶点的图 $G$，可以构建多少棵生成树？

**解答：** 对于该问题，有一个简单的以 Arthur Cayley 命名的计算公式。给定含 $n$ 个已标记顶点的图，可以找到的树的数目是 $n^{n-2}$。不同的 $n$ 值所对应的树的数目如下表所示。

| $n$ | $n^{n-2}$ | 树的树目 |
|---|---|---|
| 2 | 1 |  |
| 3 | 3 |  |

**问题 36** 给定含 $n$ 个顶点的图 $G$，可以构建多少棵正则树？

**解答：** 答案与问题 35 相同。它只是该问题的另一种提问方式。因为正则树与生成树的边数相同。

**问题 37 DAG 中的哈密顿路径：** 给定一个 DAG，设计一个线性时间算法来判定是否存在一条恰好访问每个顶点一次的路径。

**解答：** 哈密顿路径问题是一个 NP 完全问题(更多细节可参见第 20 章)。为了解决这个问题，可尝试给出一个近似算法(即能够产生问题的解，但不能保证是最优解)。

下面考虑用拓扑排序算法来解决这个问题。拓扑排序有一个有趣的性质：所有有序的连续顶点对之间有边连接，这些边在 DAG 图中形成了一条有向哈密顿路径。如果哈密顿路径存在，则拓扑排序是唯一的。此外，如果一个拓扑排序不形成哈密顿路径，则 DAG 将有至少两个拓扑次序。

近似算法：计算拓扑排序，并检查拓扑顺序的每个连续顶点对之间是否存在边。

在无权图中，查找一条从 $s$ 到 $t$ 且恰好访问每个顶点 1 次的路径。基于回溯的基本解决方案是，从 $s$ 开始，递归地访问它的所有邻居，并确保每个顶点仅被访问一次。基于上述算法的具体实现如下：

```
bool seenTable[32];
void HamiltonianPath( Graph G, int u ) {
        if( u == t )
                /* 检查是否已访问过所有顶点 */
        else {
                for( int v = 0; v < n; v++ )
                        if( !seenTable[v] && G.adjMatrix [u][v] ) {
                                seenTable[v] = true;
                                HamiltonianPath( v );
                                seenTable[v] = false;
                        }
        }
}
```

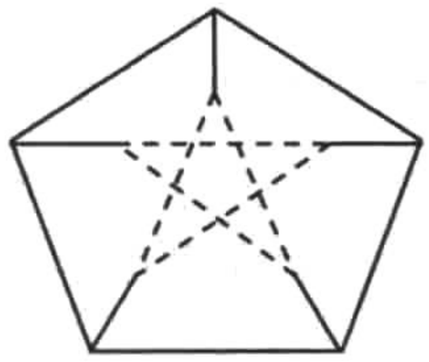
需要注意的是，如果有从 $s$ 到 $u$ 的部分路径包含顶点 $s=v_1$，$v_2$，…，$v_k=u$，那么为了确定下一个要访问的结点，不必关心路径中结点的访问次序。只需要知道所有已经被访问过的结点集(seenTable[]数组)和当前访问的顶点 $u$。共有 $2^n$ 种可能的结点集合和 $n$ 种 $u$ 的选择方式。换言之，HamiltonianPath()函数有 $2^n$ 个可能的 seenTable[]数组和 $n$ 个不同的参数取值。在递归调用的过程中，HamiltonianPath()的作用完全由 seenTable[]数组和参数 $u$ 决定。

**问题 38** 哈密顿环(或哈密顿回路)问题：能否访问图中的每个顶点恰好一次，且起

点和终点是同一个顶点？

**解答：**由于哈密顿路径问题是一个 NP 完全问题，所以哈密顿环也是一个 NP 完全问题。哈密顿环是一个访问图中每个顶点恰好一次的环。尚未有解决此问题的充分必要条件。目前有如下一些充分条件：

- 对于有哈密顿环的图，每个顶点的度必须大于或等于 2。
- 如下所示的佩特森图没有哈密顿环。



- 一般而言，图具有的边越多，越有可能包含哈密顿环。
- 假设 $G$ 是一个含有 $n \geqslant 3$ 个顶点的简单图。如果每个顶点的度至少为 $n/2$，那么 $G$ 有哈密顿环。
- 目前已知的查找哈密顿环的最佳算法在最坏情况下的时间复杂度是指数级的。

如上所述，哈密顿的近似算法请参见第 19 章。

**问题 39** Dijkstra 与 Prim 算法的区别是什么？

**解答：**Dijkstra 算法几乎等同于 Prim 算法。算法都从一个特定的顶点开始，在图中向外扩展直至所有顶点都被访问。唯一的区别是，Prim 算法存储代价最小的边，而 Dijkstra 算法存储从源点到当前顶点的总代价。更简单地说，Dijkstra 算法存储代价最小的边的总和，而 Prim 算法只存储最多一个代价最小的边。

**问题 40 反转图：**给出算法，返回有向图的反转图(从 $v$ 到 $w$ 的边由 $w$ 到 $v$ 来替换)。

**解答：**在图论中，有向图 $G$ 的反转(也叫作转置)是在同一顶点集上的另一个有向图，它的所有边是反转的。即如果 $G$ 含有边($u$，$v$)，那么 $G$ 的反转图则含有边($v$，$u$)，反之亦然。

**算法：**

```
Graph ReverseTheDirectedGraph(Graph G) {
        假设生成的反转新图名为ReversedGraph
        //反转图包含相同数量的顶点和边
        for each vertex of given graph G {
                for each vertex w adjacent to v {
                        在ReversedGraph图中添加w到v的边；
                        //即只需要反转邻接矩阵中的二进制位
                }
        }
        return ReversedGraph;

}
```

**问题 41 旅行商问题：**找到图中的一条最短路径，使得图中的每个顶点至少被访问一次且路径的起始和终点都是同一个顶点。

**解答：**旅行商问题(TSP)与查找哈密顿环相关。给定一个有权图 $G$，查找包含所有顶点的最短环(该问题可能不是简单问题)。

**近似算法：**该算法无法给出最优解，但给出了一个时间复杂度在 2 倍最优解时间内的

可行解(在最坏情况下)。

1) 查找最小生成树(MST)。

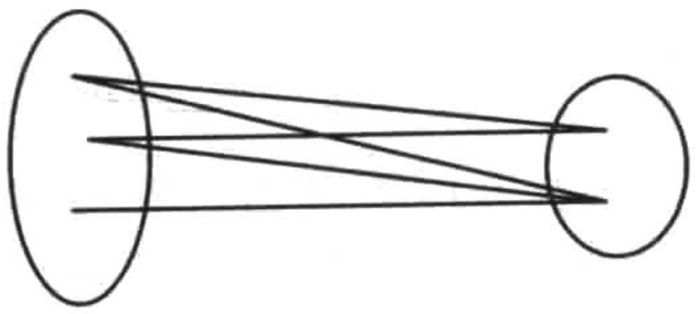
2) 对 MST 调用 DFS 搜索。

更多细节请参见第 20 章。

**问题 42** 讨论二分图匹配问题。

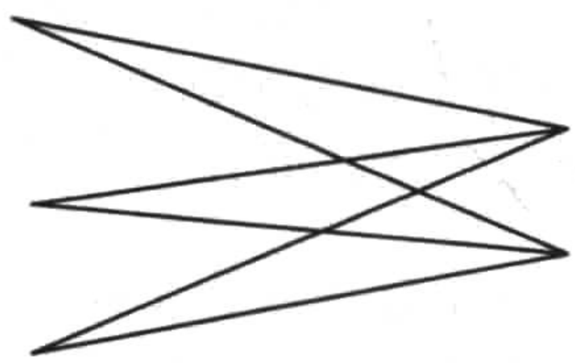
**解答:** 在二分图中，图中的顶点可以分成两个不相交的集合，图中的每条边都由一个集合的顶点连接到另一个集合的顶点(如下图所示)。

**定义:** 简单图 $G=(V, E)$被称作二分图，如果它的顶点能被分成两个不相交的集合 $V=V_1 \cup V_2$，并且使得每条边形如 $e=(a, b)$，其中 $a \in V_1$，$b \in V_2$。重要条件是 $V_1$或 $V_2$ 中的顶点之间是没有边相连的。

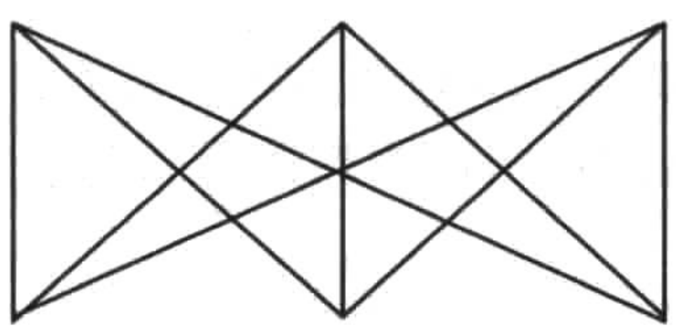


**二分图的性质:**

- 一个图被称作二分图当且仅当给定图没有奇数长度的环。
- 完全二分图 $K_{m,n}$是这样一个二分图，一个集合的每个顶点都与另一个集合的每个顶点有边相连。

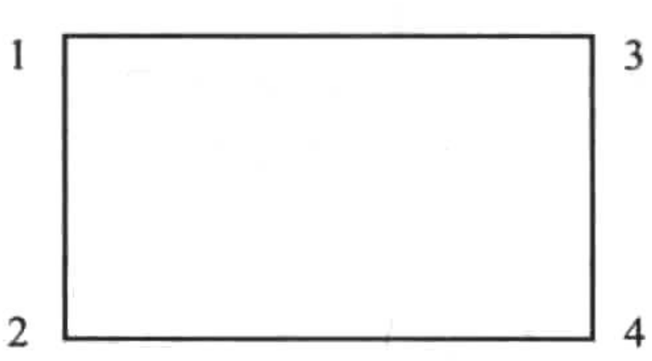


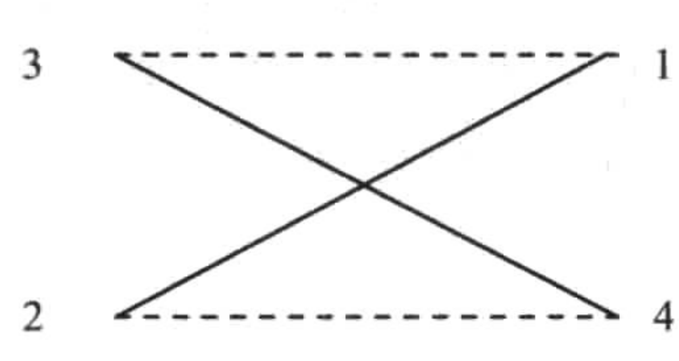
$K_{2,3}$



$K_{3,3}$

- 边的子集 $M \subset E$ 是匹配的，如果任意两条边之间没有公共顶点。如下图所示，边的匹配集由虚线表示。匹配集 $M$ 被称作最大匹配集，如果它含有最大的可能边数。在图中，虚线边表示给定图的交替匹配。




- 一个匹配 $M$ 是完美的，如果它匹配所有顶点。为了进行完美匹配，必须有 $V_1=V_2$。
- 一条交替路径是这样一条路径，路径中的边由匹配边和非匹配边交替组成。若能够找到一边交替路径，则可以改进匹配。这是因为交替路径由匹配边和非匹配边组成，而且非匹配边的数目等于匹配边的数目加 1。因此，一条交替路径总是使匹配加 1。

**下一个问题是怎样查找完美匹配。**

基于上述理论和定义，使用下面的近似算法可以找到完美匹配。

**匹配算法(匈牙利算法)**

1）从一个非匹配顶点开始。

2）查找交替路径。

3）若存在，把匹配边变为非匹配边，反之亦然。若不存在，选择另一个非匹配顶点。

4）如果边的数目等于 $V/2$，则程序停止。否则继续执行第一步，重复该过程直至所有的顶点均被检查过且没有找到任何交替路径。

**匹配算法的时间复杂度**

迭代次数是 $O(V)$。

使用 BFS 发现交替路径的时间复杂度为 $O(E)$。

因此，总的时间复杂度为 $O(V \times E)$。

**问题 43** 婚姻与人事问题。

**婚姻问题：**有 $X$ 个男人和 $Y$ 个女人想要结婚。参与者指出在异性中谁是自己可能的配偶。每个女人最多和一个男人结婚，每个男人也最多是和一个女人结婚。怎样让每个人和他(她)喜欢的人结婚？

**人事问题：**你是一家公司的老板。公司有 $M$ 个工人和 $N$ 份工作。每个工人能胜任其中的某些工作，而无法胜任另一些工作。如何把工作分配给每个工人？

**解答：**这是二分图的另一种提问方式，解决方法与问题 42 一样。

**问题 44** 在完全二分图 $K_{m,n}$ 中有多少条边？

**解答：** $m \times n$。因为第一个集合中的每个顶点都与第二个集合中的所有顶点相连。

**问题 45** 如果一个图有多条边但没有环，并且图中的每个顶点有相同的邻居数，即每个顶点的度均相同，则该图称作正则图。如果是一个 $K_{m,n}$ 正则图，那么 $m$ 和 $n$ 之间的关系是什么？

**解答：**因为每个顶点有相同的度，所以 $m$ 和 $n$ 的关系应该是 $m=n$。

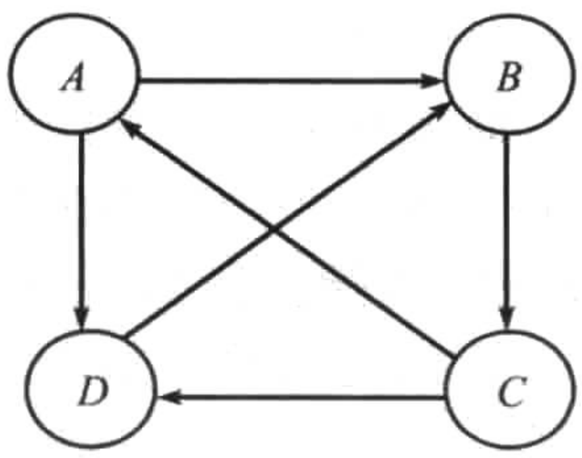
**问题 46** 对于 $n$ 个顶点的二分图，最大匹配包含的最大边数是多少？

**解答：**根据匹配的定义，边之间不存在公共顶点。所以在二分图中，每个顶点仅能连接一个顶点。由于所有的顶点被分成两个集合，所以如果将顶点对半分开，就能得到最大的边数。最终答案是 $n/2$。

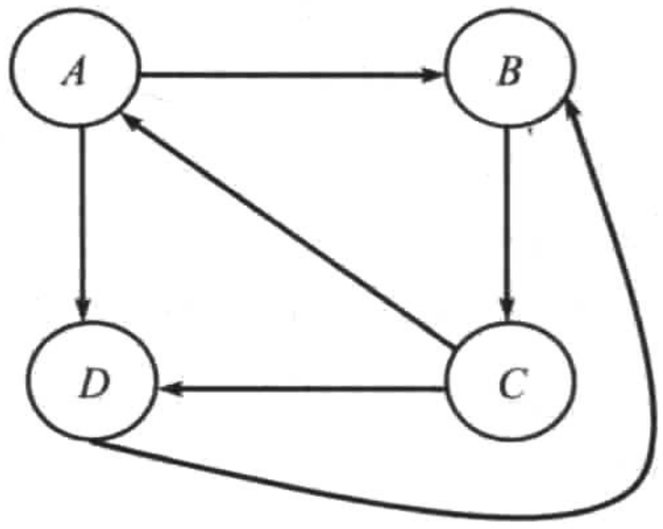
**问题 47** 讨论平面图及相关内容。

平面图：能否在边不相交的情况下画出图中的所有边吗？

**解答：**如果能在平面上以这样一种方式画出一个图：除了在边的公共端点外，任意两条边都不相交，则该图称作平面图。任何一种这样的画法称作图的平面画法。以如下所示的图为例。



可以很容易地将该图转化为如下的平面图(没有任何交叉边)。




如何确定一个图是否为平面图？该问题没有简单解。然而，通过某些有趣的性质能够判定给定图是否是平面的。

**平面图的性质**

- 如果图 $G$ 是有 $V$ 个顶点的简单平面连通图，当 $V=3$ 时，边 $E=3V-6$。
- $K_5$ 是非平面图($K_5$ 代表 5 个顶点的完全图)。
- 如果图 $G$ 是有 $V$ 个顶点和 $E$ 条边的简单平面连通图，且图中不包含三角形，那么 $E=2V-4$。
- $K_{3,3}$ 是非平面图($K_{3,3}$ 代表两个顶点集合各包含 3 个顶点的二分图。$K_{3,3}$ 共包含 6 个顶点)。
- 如果图 $G$ 是简单平面连通图，那么 $G$ 中至少有一个顶点的度小于或等于 5。
- 图是平面的当且仅当它不含有 $K_5$ 和 $K_{3,3}$ 这类子图。
- 如果图 $G$ 含有非平面子图，那么 $G$ 是非平面的。
- 如果图 $G$ 是平面的，那么 $G$ 的每个子图也都是平面的。
- 对于任意平面连通图 $G=(V, E)$，$V+F-E=2$ 成立，其中 $F$ 表示图的面数。
- 对于任意含有 $K$ 个分量的平面图 $G=(V, E)$，$V+F-E=1+K$ 成立。

为了判定给定图的平面性，需要利用上述这些性质来确定它是否是平面的。需要注意的是，上面所有的性质仅仅是必要条件而不是充分条件。

**问题 48** $K_{2,3}$ 有多少面？

**解答：** 根据上述讨论可知，$V+F-E=2$。从上述问题可知，$E=m\times n=2\times 3=6$ 和 $v=m+n=5$。所以，从 $5+F-6=2$ 推出 $F=3$。

**问题 49** 讨论图着色及相关内容。

**解答：** 对图 $G$ 进行 $k$ 着色是指，使用最多 $k$ 种颜色对 $G$ 中的每个顶点着色，并且任意两个相邻顶点具有不同的颜色。一个图称为 $k$ 可着色的当且仅当它包含一个 $k$ 种颜色。

**图着色应用：** 图着色问题有许多应用。例如，行程安排、编译器中的寄存器分配、移动通信的频率分配等。

**团：** 图 $G$ 中的团指的是最大完全子图，用 $\omega(G)$ 表示。

**色数：** 图的色数是使 $G$ 为 $k$ 可着色时最小的 $k$ 值，用 $X(G)$ 表示。

$X(G)$ 的下界是 $\omega(G)$，即 $\omega(G)\leqslant X(G)$。

**色数的性质：** 假设 $G$ 是有 $n$ 个顶点的图，$G'$ 是它的补集。则有，

- $X(G)\leqslant \Delta(G)+1$，其中 $\Delta(G)$ 是 $G$ 的最大度数
- $X(G)\omega(G')\geqslant n$
- $X(G)+\omega(G')\leqslant n+1$

- $X(G)+(G')\leqslant n+1$

**$k$ 可着色问题：** 给定一个图 $G=(V, E)$和正整数 $k\leqslant V$。判断 $G$ 是否为 $k$ 可着色？

这是一个 NP 完全问题，将在第 20 章详细讨论。

**图着色算法：** 由上述讨论可知，该问题是一个 NP 完全问题。所以没有多项式时间算法来确定 $X(G)$。下面给出近似算法。

- 考虑图 $G$ 中两个不相邻的顶点 $a$ 和 $b$。通过在 $a$ 和 $b$ 之间加入一条边可以得到连接 $G_1$。通过把$\{a, b\}$合并成一个点 $c(a, b)$，并将点 $c$ 连接到顶点 $a$ 和 $b$ 的每个邻居，可以获得构造 $G_2$(移除多余的边)。
- 当 $a$ 和 $b$ 使用相同的颜色对图 $G$ 着色时，将产生 $G_1$ 的一个着色。当 $a$ 和 $b$ 使用不同的颜色对图 $G$ 着色时，将产生 $G_2$ 的一个着色。
- 在产生的每个图中重复连接和构造操作，直到所有的图都是团。如果最小团的大小为 $k$，那么$(G)=k$。

**关于图着色的重要提示**

- 任何一个简单平面图 $G$ 能用 6 种颜色着色。
- 每个简单平面图能够用最多 5 种颜色着色。

**问题 50** 什么是四着色问题？

**解答：** 任意一个地图能够构成一个图。地图中的区域可由图的顶点来表示。如果顶点对应的区域是邻接的，则在两个顶点之间加入一条边。得到的图是平面的，即该图能够在边不相交的情况下在平面上画出。

四着色问题：平面图中的顶点是否能用最多 4 种颜色来着色，使得任意两个相邻顶点的颜色不同。

**历史：** 四着色问题最早是由 Francis Guthrie 提出的。他是英国伦敦大学学院的学生，并且在 Augusts De Morgan 指导下学习。从伦敦毕业后，他学习过法律，但几年后他的弟弟 Frederick Guthrie 也成了 De Morgan 的学生。某天，Francis 请他弟弟去和 De Morgan 讨论这个问题。

**问题 51** 当采用邻接矩阵表示图时，大多数图算法的时间复杂度为 $O(V^2)$。能否在 $O(v)$时间内确定用邻接矩阵表示的有向图是否包含汇点。汇点是入度为$|V|-1$ 且出度为 0 的顶点(在图中只存在一个汇点)。

**解答：** 一个顶点 $i$ 是汇点当且仅当对于所有的 $j$，$M[i, j]=0$ 并且当 $j\neq i$ 时，$M[j, i]=1$。对于任意一对顶点 $i$ 和 $j$：

$M[i, j]=1\rightarrow$顶点 $i$ 不可能是汇点

$M[i, j]=0\rightarrow$顶点 $j$ 不可能是汇点

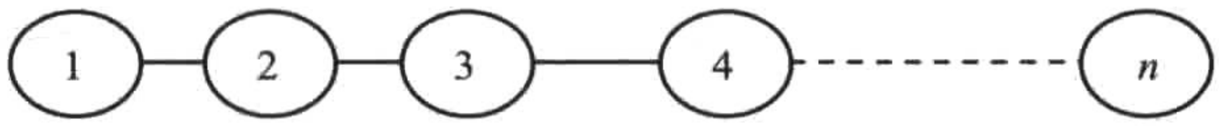
**算法：**

- 初始时，$i=1$，$j=1$。
- 如果 $M[i, j]=0\rightarrow i$ 胜出，$j++$。
- 如果 $M[i, j]=1\rightarrow j$ 胜出，$i++$。
- 重复该过程直到 $j=n$ 或者 $i=n+1$。
- 如果 $i==n+1$，该图不含汇点。
- 否则，检查第 $i$ 行——应全为 0，以及第 $i$ 列(除了 $M[i, i]$外，其余应全为 1)。如果满足这两个条件，则 $i$ 就是一个汇点。

时间复杂度为 $O(V)$，因为最多需要检查矩阵中 $2|V|$ 个单元。

**问题 52** DFS 内存使用的最坏情况是什么？

**解答：** 当图是一个线性表时，此时的空间开销为 $O|V|$。因此，当图的直径较小时，算法的存储效率较高。




**问题 53** DFS 能否找到从起始点到某个点 $w$ 的最短路径？

**解答：不能。**DFS 在遍历的过程中不需要选择权值最小的边。



Chapter 10 第 10 章

# 排　序

## 10.1　什么是排序

排序是按照某种顺序(升序或降序)排列序列元素的一种算法。排序的输出是输入的排列或重新排序。

## 10.2　为什么需要排序

排序是计算机科学中的重要算法，排序有时可以显著降低问题的复杂度，可以使用排序作为减少查找复杂度的一种技术。鉴于排序的重要性，已有针对排序算法的大量研究并将其用于多种计算机算法(例如，查找元素)、数据库算法和其他算法中。

## 10.3　排序的分类

排序算法通常基于以下参数分类。

### 1. 比较的次数

在这种方法中，采用基于比较的次数对排序算法进行分类。对于基于比较的排序算法，在最好情况下时间复杂度为 $O(n\log n)$，而在最坏情况下则为 $O(n^2)$。基于比较的排序算法通过关键字比较操作来排列序列元素，并且对于大多数输入至少需要 $O(n\log n)$次比较。

本章后面将讨论计数排序、桶排序和基数排序等几个非比较的排序算法。该类排序算法利用某些输入限制来降低复杂度。

### 2. 交换的次数

在此方法中，排序算法以交换的次数来对算法分类。

### 3. 内存使用

有些排序算法是“原地的”(即不占用额外的内存空间)，仅需要 $O(1)$或 $O(\log n)$的内存开销用于创建临时排序数据的辅助存储位置。

#### 4. 递归

排序算法可以是递归(如快速排序)或非递归(如选择排序和插入排序)排序，也有同时采用递归和非递归的排序算法(如归并排序)。

#### 5. 稳定性

假设对于所有的索引 $i$ 和 $j$ 使得键值 $A[i]$等于 $A[j]$，并且在原始文件中 $R[i]$领先于 $R[j]$。若在排序后序列中记录 $R[i]$仍然位于 $R[j]$前面，则称这种排序算法是稳定的。少数排序算法能够维持具有相等键值元素的相对次序(即使排序后相等元素仍然保持它们的相对位置)。

#### 6. 适应性

少数排序算法的复杂度依赖于序列的初始排列情况(如快速排序)，输入的初始排列将会影响算法的运行时间，有这种情况出现的算法称作适应算法。

## 10.4　其他分类方法

另一种排序的分类方法是：

- 内部排序。
- 外部排序。

#### 1. 内部排序

在排序时仅使用主存储器的排序算法称为内部排序(internal sort)。该算法对所有的存储器都能够高速随机存取。

#### 2. 外部排序

在排序时需要使用磁带或磁盘等外部存储器的排序算法都属于外部排序(external sort)。

## 10.5　冒泡排序

冒泡排序(bubble sort)是一种最简单的排序算法。其基本思想是迭代地对输入序列中的第一个元素到最后一个元素进行两两比较，当需要时交换这两个元素(位置)。该过程持续迭代直到在一趟排序过程中不需要交换操作为止。冒泡排序得名于键值较小的元素如同“气泡”一样逐渐漂浮到序列的顶端。通常，插入排序比冒泡排序有更好的性能。由于冒泡排序的简单性和复杂度，有些学者建议不讲授该排序算法。

相比于其他排序算法冒泡排序的唯一显著优势是，它可以检测输入序列是否已经是排序的。

```
void BubbleSort(int A[], int n) {
        for (int pass = n - 1; pass >= 0; pass--)    {
                for (int i = 0; i < pass - 1 ; i++)    {
                        if(A[i] > A[i+1]) {
                                //交换元素
                                int temp = A[i];
                                A[i] = A[i+1];
                                A[i+1] = temp;
                        }
                }
        }
}
```

上述算法的时间复杂度为 $O(n^2)$(即使是在最好的情况下)。可以通过增加一个附加标

记来改进该算法。在排序过程中，没有交换操作则意味着排序完成。如果序列已经是有序的，则可以通过判断该标记来结束算法。

```
void BubbleSortImproved(int A[], int n) {
        int pass, i, temp, swapped = 1;
        for (pass = n - 1; pass >= 0 && swapped; pass--) {
                swapped = 0;
                for (i = 0; i < pass - 1 ; i++) {
                        if(A[i] > A[i+1]) {
                                // 交换元素
                                temp = A[i];
                                A[i] = A[i+1];
                                A[i+1] =  temp;
                                swapped = 1;
                        }
                }
        }
}
```

在最好情况下上述改进的冒泡算法的时间复杂度为 $O(n)$。

**性能**

| |
|---|
| 最坏情况下时间复杂度：$O(n^2)$ |
| 最好情况下时间复杂度(改进版)：$O(n)$ |
| 平均情况下时间复杂度(基本版)：$O(n^2)$ |
| 最坏情况下空间复杂度：$O(1)$　辅助 |

## 10.6　选择排序

选择排序(selection sort)是一种原地(in-place)排序算法，适用于小文件。由于选择操作是基于键值的且交换操作只在需要时才执行，所以选择排序常用于数值较大和键值较小的文件。

**1. 优点**

- 容易实现。
- 原地排序(不需要额外的存储空间)。

**2. 缺点**

- 扩展性较差：$O(n^2)$。

**3. 算法**

1）寻找序列中的最小值。

2）用当前位置的值交换最小值。

3）对所有元素重复上述过程，直到整个序列排序完成。

该算法称为选择排序，因为它重复选择最小的元素。

```
void Selection(int A[], int n) {
        int i, j, min, temp;
        for (i = 0; i < n - 1; i++) {
                min = i;
                for (j = i+1; j < n; j++) {
                        if(A [j] < A [min])
                                min = j;
                }
```

```
                //交换元素
                temp = A[min];
                A[min] = A[i];
                A[i] = temp;
        }
}
```

4. **性能**

| 最坏情况下时间复杂度：$O(n^2)$ |
| --- |
| 最好情况下时间复杂度：$O(n)$ |
| 平均情况下时间复杂度：$O(n^2)$ |
| 最坏情况下空间复杂度：$O(1)$　辅助 |

## 10.7　插入排序

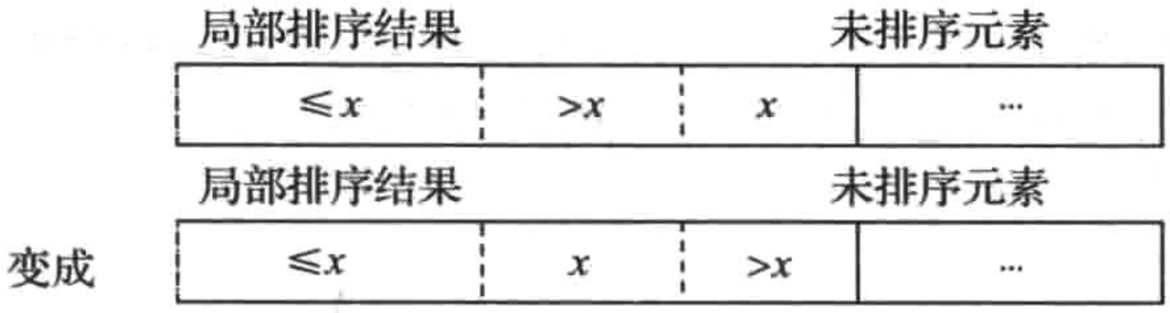
插入排序(insertion sort)是一种简单且有效的比较排序算法。在每次迭代过程中算法随机地从输入序列中移除一个元素，并将该元素插入待排序序列的正确位置。重复该过程，直到所有输入元素都被选择一次。

1. **优点**

- 实现简单。
- 数据量较少时效率高。
- 适应性(adaptive)：如果输入序列已预排序(可能是不完全的预排序)，则时间复杂度为 $O(n+d)$，$d$ 是反转的次数。
- 算法的实际运行效率优于选择排序和冒泡排序，即使在最坏情况下三个算法的时间复杂度均为 $O(n^2)$。
- 稳定性(stable)：键值相同时它能够保持输入数据的原有次序。
- 原地(in-place)：仅需要常量 $O(1)$的辅助内存空间。
- 即时(online)：插入排序能够在接收序列的同时对其进行排序。

2. **算法**

插入排序重复如下过程：每次从输入数据中移除一个元素并将其插入已排序序列的正确位置，直到所有输入元素都插入有序序列中。插入排序是典型的原地排序。经过 $k$ 次迭代后数组具有性质：前 $k+1$ 个元素已经排序。




每个与 $x$ 比较且大于 $x$ 的元素复制到 $x$ 的右边。

```
void InsertionSort(int A[], int n) {
        int i, j, v;
        for (i = 2; i <= n - 1; i++) {
                v = A[i];
                j = i;
```

```
                while (A[j-1] > v && j >= 1) {
                        A[j] = A[j-1];
                        j--;
                }
                A[j] = v;
        }
}
```

**例子：** 给定一个序列：6 8 1 4 5 3 7 2，按升序排序。

**6** 8 1 4 5 3 7 2(考虑索引位置 0)

**6 8** 1 4 5 3 7 2(考虑索引位置 0～1)

**1 6 8** 4 5 3 7 2(考虑索引位置 0～2：在 6 和 8 前插入 1)

**1 4 6 8** 5 3 7 2(重复以上过程直到序列被排序)

**1 4 5 6 8** 3 7 2

**1 3 4 5 6 7 8** 2

**1 2 3 4 5 6 7 8**(已排序的序列!)

### 3. 分析

**最坏情况分析：** 对每个 $i$，其内层循环需要移动所有元素 $A[1]$，…，$A[i-1]$的时间开销为 $\Theta(i-1)$(当 $A[i]$的键值小于所有其他元素的键值时)。

$$T(n)=\Theta(1)+\Theta(2)+\Theta(3)+\cdots+\Theta(n-1)$$

$$=\Theta(1+2+3+\cdots+n-1)=\Theta\left(\frac{n(n-1)}{2}\right)\approx\Theta(n^2)$$

**平均情况分析：** 平均情况下，在 $A[1]$，…，$A[i-1]$中插入 $A[i]$的内层循环所花费的时间为$\Theta(i/2)$。

$$T(n)=\sum_{i=1}^{n}\Theta(i/2)\approx\Theta(n^2)$$

### 4. 性能

| |
|---|
| 最坏情况下时间复杂度：$O(n^2)$ |
| 最好情况下时间复杂度：$O(n^2)$ |
| 平均情况下时间复杂度：$O(n^2)$ |
| 最坏情况下空间复杂度：$O(n^2)$　总计，$O(1)$　辅助 |

### 5. 与其他算法的比较

插入排序是一种最坏情况下时间复杂度为 $O(n^2)$的基本排序算法。在数据几乎都已经排序或者输入数据规模较小时可以使用插入排序。由于上述原因以及插入排序的稳定性，插入排序可用于归并和快速排序等高开销的分治排序算法的递归基础情形(当问题规模小时)。

**注意：**

- 在平均和最坏情况下冒泡排序比较和交换的次数都为$\frac{n^2}{2}$。
- 选择排序比较的次数为$\frac{n^2}{2}$，交换的次数为 $n$。
- 在平均情况下插入排序的比较次数为$\frac{n^2}{4}$，交换次数为$\frac{n^2}{8}$，而在最坏情况下则加倍。

- 插入排序对部分有序的输入来说几乎是线性排序。
- 选择排序最适用于值较大且键值较小的元素排序。

## 10.8　希尔排序

希尔排序(shell sort)又称为缩小增量排序(diminishing increment sort)，算法由 Donnald Shell 提出而得名。该算法是一个泛化的插入排序。插入排序在输入序列几乎已经有序的情况下非常有效。希尔排序也称为 $n$ 间距($n$-gap)插入排序。希尔排序分多路并使用不同的间距来比较相邻元素，而不是仅比较相邻对。通过逐步减少间距，最终以 1 为间距或者进行一次常规的插入排序即可。

插入排序中的比较是在两个相邻元素之间进行的，每次比较最多进行 1 次反转交换。希尔排序利用可变增量使算法直到最后一步才比较相邻元素，所以希尔排序的最后一步是一个有效的插入排序算法。希尔排序通过允许比较和交换具有一定距离的元素对插入排序进行改进。希尔排序是比较排序算法中第一个低于二次复杂度的算法。

本质上希尔排序是对插入排序的一种简单扩展。两者的主要差异在于希尔排序具有交换相距较远元素的能力，这使得它能较快地把元素交换到所需位置。例如，如果初始数组中最小的元素恰好位于数组的末端，那么插入排序需要经过对整个数组的比较和交换才能把最小元素放置到数组的开头，而希尔排序使元素能够一次移动多步而非一步，从而能够以较少的交换次数把元素放到合适的位置。

希尔排序的主要思想是比较和交换数组中每个距离为 $h$ 的元素。对其进一步解释可使算法的思路更加清晰：$h$ 决定相距多远的元素能够进行交换，例如，如果 $h$ 值为 13，则第 1 个元素(索引 0)将与第 14 个元素(索引 13)进行交换(如果需要)。第 2 个元素与第 15 个元素进行交换，以此类推。如果 $h$ 为 1，则希尔排序就与常规插入排序完全一样。

希尔排序首先选择足够大的间距 $h$(但不能超过数组的大小)开始排序，这样能允许相具较远的合适元素进行交换。一旦以某个特定的 $h$ 完成排序，则称数组是以 $h$ 间距排序的数组。接下来的步骤是以某一确定的序列依次减少间距 $h$ 的值，并重新开始新一轮以 $h$ 为间距的排序。一旦 $h$ 变为 1 且是 $h$ 间距排序的，则数组排序完成。注意，由于 $h$ 的最后一个序列值为 1，所以最后一次排序总是一个插入排序，但此时数组已经变得基本有序且更容易排序。

希尔排序使用的序列 $h1$，$h2$，…，$ht$ 称为增量序列。任何增量序列都是可以的只要 $h1=1$，且某些增量序列的效果要优于其他的增量序列。希尔排序把输入序列分成大小相同的多路子序列，然后对每路利用插入排序算法进行排序。希尔排序通过快速移动元素值到目的位置来提高插入排序的效率。

```
void ShellSort(int A[], int array_size) {
        int i, j, h, v;
        for (h = 1; h = array_size/9; h = 3*h+1);
        for ( ; h > 0; h = h/3) {
                for (i = h+1; i = array_size; i += 1) {
                        v = A[i];
                        j = i;
                        while (j > h && A[j-h] > v) {
                                A[j] = A[j-h];
                                j -= h;
                        }
```

```
                    A[j] = v;
                }
            }
    }
```

注意，当 $h==1$ 时，算法将遍历整个序列，并比较相邻元素，但对元素的交换操作非常少。当 $h==1$ 时，希尔排序的排序方式与插入排序相同，除了在前面的步骤中(即 $h>1$时)算法已经极大地减少了需要交换元素的次数。

### 1. 分析

希尔排序对中等大小的序列非常有效，对于较大的序列它不是最好的选择，但希尔排序是所有时间复杂度为 $O(n^2)$的排序算法中最快的算法。

希尔排序的缺点是它的算法思路复杂且远不及归并、堆和快速排序有效。希尔排序明显比归并、堆和快速排序慢，但它却是一种相对简单的算法。若不考虑速度的重要性，希尔排序对于数据量小于 5000 的序列是不错的选择。对较小的列表进行重复排序时，希尔排序也是一个非常好的选择。

使用希尔排序的最佳情况是序列已经完全排序，此时比较次数较少。希尔排序的运行时间取决于所选择的增量序列。

### 2. 性能

| |
|---|
| 最坏情况下时间复杂度取决于间隔序列。最好情况下：$O(n\log^2 n)$ |
| 最好情况下时间复杂度：$O(n)$ |
| 平均情况下时间复杂度取决于间隔序列 |
| 最坏情况下空间复杂度：$O(n)$ |

## 10.9 归并排序

归并排序(merge sort)是分治的一个实例。

### 1. 重要提示

- 归并是把两个已排序文件合并成一个更大的已排序文件的过程。
- 选择是把一个文件分成包含 $k$ 个最小元素和 $n-k$ 个最大元素两个部分的过程。
- 选择和归并互为逆操作：
  - 选择把一个序列分成两部分。
  - 归并把两个文件合并成一个文件。
- 归并排序是快速排序的补充。
- 归并排序以连续的方式访问数据。
- 归并排序适用于链表排序。
- 归并排序对输入的初始次序不敏感。
- 快速排序中的大部分任务在递归调用前完成。快速排序从最大子文件开始并以最小子文件结束，因此需要栈结构。此外，快速排序算法也不稳定。归并排序把序列分为两个部分，并对每个部分分别处理。归并排序从最小子文件开始并以最大子文件结束，因此不需要栈，并且归并排序算法是稳定的算法。

```
void Mergesort(int A[], int temp[], int left, int right) {
        int mid;
        if(right > left) {
```

```
                mid = (right + left) / 2;
                Mergesort(A, temp, left, mid);
                Mergesort(A, temp, mid+1, right);
                Merge(A, temp, left, mid+1, right);
        }
}
void Merge(int A[], int temp[], int left, int mid, int right) {
        int i, left_end, size, temp_pos;
        left_end = mid - 1;
        temp_pos = left;
        size = right - left + 1;
        while ((left <= left_end) && (mid <= right)) {
                if(A[left] <= A[mid]) {
                        temp[temp_pos] = A[left];
                        temp_pos = temp_pos + 1;
                        left = left +1;
                }
                else {
                        temp[temp_pos] = A[mid];
                        temp_pos = temp_pos + 1;
                        mid = mid + 1;
                }
        }
        while (left <= left_end) {
                temp[temp_pos] = A[left];
                left = left + 1;
                temp_pos = temp_pos + 1;
        }
        while (mid <= right) {
                temp[temp_pos] = A[mid];
                mid = mid + 1;
                temp_pos = temp_pos + 1;
        }
        for (i = 0; i <= size; i++) {
                A[right] = temp[right];
                right = right - 1;
        }
}
```

2. 分析

归并排序将输入序列分成两部分并递归地处理每一部分。当子问题解决后，算法又将子问题的解合并。假设具有 $n$ 个元素的归并排序的复杂度表示为 $T(n)$，则归并排序的递归式可定义为：

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

根据分治法主定理，有

$$T(n) = \Theta(n\log n)$$

**注意：**更多细节请参见第 18 章。

3. 性能

| |
|---|
| 最坏情况下时间复杂度：$\Theta(n\log n)$ |
| 最好情况下时间复杂度：$\Theta(n\log n)$ |
| 平均情况下时间复杂度：$\Theta(n\log n)$ |
| 最坏情况下空间复杂度：$\Theta(n)$　辅助 |

## 10.10 堆排序

堆排序(heapsort)是一种基于比较的排序算法，该算法同时属于选择排序。尽管在大多数计算机上堆排序的运行效率低于快速排序，但它的优势是在最坏情况下运行时间也仅为$\Theta(n\log n)$。堆排序是一种不稳定的原地排序算法。

**性能**

| |
|---|
| 最坏情况下时间复杂度：$\Theta(n\log n)$ |
| 最好情况下时间复杂度：$\Theta(n\log n)$ |
| 平均情况下时间复杂度：$\Theta(n\log n)$ |
| 最坏情况下空间复杂度：$\Theta(n)$　总计，$\Theta(1)$　辅助 |

关于堆排序的详细信息参见第 7 章。

## 10.11 快速排序

快速排序(quicksort)是分治算法技术的一个实例，也称为分区交换排序。快速排序采用递归调用对元素进行排序，是基于比较的排序算法中的一个著名算法。

划分：数组 $A[\text{low}..\text{high}]$被分成两个非空子数组 $A[\text{low}..q]$和 $A[q+1..\text{high}]$，使得 $A[\text{low}..q]$中的每一个元素都小于或等于 $A[q+1..\text{high}]$中的元素。在划分过程中需要计算索引 $q$ 的位置。

分而治之：对两个子数组 $A[\text{low}..q]$和 $A[q+1..\text{high}]$递归调用快速排序。

### 1. 算法

递归算法由以下 4 步组成：

1）如果数组中仅有一个元素或者没有元素需要排序，则返回。

2）选择数组中的一个元素作为枢轴(pivot)点(通常选择数组最左边的元素)。

3）把数组分成两部分——一部分元素大于枢轴，而另一部分元素小于枢轴。

4）对两部分数组递归调用该算法。

```
Quicksort( int A[], int low, int high ) {
        int pivot;
        /* 终止条件! */
        if( high > low ) {
                pivot = Partition( A, low, high );
                Quicksort( A, low, pivot-1 );
                Quicksort( A, pivot+1, high );
        }
}
int Partition( int A, int low, int high ) {
        int left, right, pivot_item = A[low];
        left = low;
        right = high;
        while ( left < right ) {
                /* 当item < pivot移动左指针*/
                while( A[left] <= pivot_item )
                        left++;
                /*当item > pivot移动右指针*/
                while( A[right] > pivot_item )
                        right--;
                if( left < right )
```

```
                swap(A,left,right);
        }
        /*right即是枢轴的最终位置*/
        A[low] = A[right];
        A[right] = pivot_item;
        return right;
}
```

2. 分析

假设快速排序的复杂度为 $T(n)$且所有元素都不相同。$T(n)$取决于两个子问题的规模，而规模又取决于枢轴点。如果枢轴是第 $i$ 个最小元素，则恰好有$(i-1)$个元素将被分到左半部分，$(n-i)$个元素被分到右半部分，我们称其为 $i$ 划分。由于每个元素选作枢轴的概率相等，所以选择第 $i$ 个元素作为枢轴的概率是$\frac{1}{n}$。

**最好情况：**每个划分把数组分成相等的两部分。

$$T(n)=2T\left(\frac{n}{2}\right)+\Theta(n)=\Theta(n\log n)\quad\text{（使用分治法主定理）}$$

**最坏情况：**每个划分把数组分成不相等的两部分。

$$T(n)=T(n-1)+\Theta(n)=\Theta(n^2)\quad\text{（问题规模减小和递归求解主定理）}$$

最坏情况发生在序列已经排序且选择最后一个元素作为枢轴时。

**平均情况：**在快速排序的平均情况下，不确定在什么地方划分序列。因此，应该考虑划分位置的所有可能值，对各种情况下的时间开销求和后再除以 $n$ 得到平均时间复杂度。

$$T(n)=\sum_{i=1}^{n}\frac{1}{n}(\text{随着 } i \text{ 的分离运行})+n+1$$

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

// 最好情况下，可以假设 $T(n-i)$ 和 $T(i-1)$ 相等

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

$$=\frac{2}{n}\sum_{i=0}^{n-1}T(i)+n+1$$

等式两边同乘以 $n$ 得到：

$$nT(n)=2\sum_{i=0}^{n-1}T(i)+n^2+n$$

对 $n-1$ 有同样的公式：

$$(n-1)T(n-1)=2\sum_{i=0}^{n-2}T(i)+(n-1)^2+(n-1)$$

从 $n$ 中减去 $n-1$ 的公式得到：

$$nT(n)-(n-1)T(n-1)=2\sum_{i=0}^{n-1}T(i)+n^2+n-\left(2\sum_{i=0}^{n-2}T(i)+(n-1)^2+(n-1)\right)$$

$$nT(n)-(n-1)T(n-1)=2T(n-1)+2n$$

$$nT(n)=(n+1)T(n-1)+2n$$

公式两边同时除以 $n(n+1)$得到：

$$\frac{T(n)}{n+1}=\frac{T(n-1)}{n}+\frac{2}{n+1}=\frac{T(n-2)}{n-1}+\frac{2}{n}+\frac{2}{n+1}$$

$$\cdots$$

$$=O(1)+2\sum_{i=3}^{n}\frac{1}{i}=O(1)+O(2\log n)$$

$$\frac{T(n)}{n+1}=O(\log n)$$

$$T(n)=O((n+1)\log n)=O(n\log n)$$

时间复杂度为 $T(n)=O(n\log n)$。

**3. 性能**

| |
|---|
| 最坏情况下时间复杂度：$O(n^2)$ |
| 最好情况下时间复杂度：$O(n\log n)$ |
| 平均情况下时间复杂度：$O(n\log n)$ |
| 最坏情况下空间复杂度：$O(1)$ |

**4. 随机化快速排序**

在快速排序的平均情况下，输入序列中数字的所有排列都是等概率的。然而，并不能保证始终是这种情况。可以在快速排序中加入随机化来降低出现最坏情况的可能性。

有两种在快速排序中增加随机化的方法：在数组中随机放置输入数据或者在输入数据中随机选择一个元素作为枢轴。后一种方式更加容易分析和实现，它仅需要改变划分算法即可。

在常规快速排序中，始终选择序列最左边的一个元素作为枢轴元素进行排序。在随机化快速排序中，从子数组 A[low..high]中随机选择一个元素来代替 A[low]元素作为枢轴，即通过交换 A[low]与从 A[low..high]中随机选择的元素来实现。这样能够保证子数组 high－low＋1 个元素中的任意一个元素选作枢轴元素的概率相等。由于枢轴元素是随机选择的，所以可以确保输入数组的划分在平均情况下是均衡的，这样就能阻止由非均衡划分导致的快速排序最坏情况的发生。

不过，尽管随机化快速排序降低了最坏情况下的复杂度，但其最坏情况下的复杂度仍然是 $O(n^2)$。改进随机化快速排序的一种方法是选择特定的元素作为枢轴点，而不只是从数组随机选择一个元素。一个常用的方法是从数组中随机选择 3 个元素，取其中位数作为枢轴。

## 10.12 树排序

树排序(tree sort)采用二叉搜索树。它包括扫描每个输入元素并将其放置到二叉搜索树适当的位置这两个阶段：

- 第一阶段是使用给定的数组元素创建一个二叉搜索树。
- 第二阶段是中序遍历给定的二叉搜索树并生成有序数组。

**性能**

该方法的平均比较次数为 $O(n\log n)$。但在最坏情况下，即当排序树是一棵偏斜树时，

比较次数会升至 $O(n^2)$。

## 10.13　排序算法比较

| 算法名 | 平均情况 | 最坏情况 | 辅助存储器 | 是否稳定 | 其他注释 |
|---|---|---|---|---|---|
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | 1 | 是 | 代码量较少 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | 1 | 否 | 稳定性取决于实现 |
| 插入排序 | $O(n^2)$ | $O(n^2)$ | 1 | 是 | 平均情况下也可以是 $O(n+d)$，其中 $d$ 为反转数 |
| 希尔排序 | — | $O(n\log^2 n)$ | 1 | 否 | |
| 归并排序 | $O(n\log n)$ | $O(n\log n)$ | 不确定 | 是 | |
| 堆排序 | $O(n\log n)$ | $O(n\log n)$ | 1 | 否 | |
| 快速排序 | $O(n\log n)$ | $O(n^2)$ | $O(\log n)$ | 不确定 | 是否稳定取决于如何处理枢轴 |
| 树排序 | $O(n\log n)$ | $O(n^2)$ | $O(n)$ | 不确定 | 能够实现稳定排序 |

注：$n$ 表示输入元素的个数。

## 10.14　线性排序算法

前面几节已经列举了多个基于比较的排序算法。最佳的基于比较排序的算法时间复杂度为 $O(n\log n)$。本节将讨论其他类型的算法：线性排序算法。为了能够改进这些算法的时间复杂度，可以对输入数据做一些假设。线性排序算法举例如下：

- 计数排序。
- 桶排序。
- 基数排序。

## 10.15　计数排序

计数排序(counting sort)不属于比较排序算法，其排序复杂度为 $O(n)$。为获得 $O(n)$ 的复杂度，计数排序假定每个输入元素都是 $1 \sim K$ 之间的整数($k$ 为整数)。当 $K=O(n)$ 时，计数排序的运行时间为 $O(n)$。计数排序的基本思想是，对于每一个输入元素 $X$，确定小于 $X$ 的元素的个数，根据此信息就可以将 $X$ 直接放到正确的位置。例如，如果有 10 个元素小于 $X$，则 $X$ 的输出位置为 11。

在如下所示的代码中，$A[0..n-1]$是长度为 $n$ 的输入数组。在计数排序中还需要两个数组：假定数组 $B[0..n-1]$用于存放排序结果，数组 $C[0..K-1]$提供临时存储空间。

```
void CountingSort (int A[], int n, int B[], int K) {
    int C[K], i, j;
    // 复杂度: O(K)
    for (i =0 ; i<K; i++)
        C[i] = 0;
    // 复杂度: O(n)
    for (j =0 ; j<n; j++)
        C[A[j]] = C[A[j]] + 1;
     // C[i]包含所有等于i的元素
        // 复杂度: O(K)
        for (i =1 ; i<K; i++)
            C[i] = C[i] + C[i-1];
```

```
            //C[i]包含所有小于等于i的元素
            // 复杂度:O(n)
            for (j = n-1; j>=0; j--) {
                B[C[A[j]]] = A[j];
                C[A[j]] = C[A[j]] - 1;
            }
    }
```

总复杂度：若 $K=O(n)$，则 $O(K)+O(n)+O(K)+O(n)=O(n)$。若 $K=O(n)$，则空间复杂度为 $O(n)$。

**注意：** 若 $K=O(n)$，则计数排序运行较快，否则复杂度将更高。

## 10.16 桶排序

与计数排序类似，桶排序(bucket sort)也对输入加以限制来提高算法性能。换言之，如果输入序列来自固定的集合，则桶排序效率较高。桶排序是计数排序的泛化。例如，假设所有输入元素来自{0，1，…，$K-1$}，即在区间[0，$K-1$]上的整数集合，这就表示 $K$ 是输入序列中最远距离元素的数目。桶排序采用 $K$ 个计数器，第 $i$ 个计数器记录第 $i$ 个元素的出现次数。含有两个桶的桶排序是快速排序的一个有效版本。

```
#define BUCKETS 10
void BucketSort(int A[], int array_size) {
        int i, j, k;
        int buckets[BUCKETS];
        for(j =0; j < BUCKETS; j++)
                buckets[j] = 0;
        for(i =0; i < array_size; i++)
                ++ buckets[A[i]];
        for(i =0, j=0; j < BUCKETS; j++)
                for(k = buckets[j];k > 0; --k)
                        A[i++] = j;
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(n)$。

## 10.17 基数排序

类似于计数排序和桶排序，基数排序(radix sort)也对输入元素做某种假设。假定待排序的输入值以 $d$ 为基数，即所有数值均为 $d$ 数位数。

基数排序首先基于最后一位数字(最低有效位)排序，随后该结果又以第二位(最低有效位的相邻位)排序，重复该过程直至算法到达最高位。然后，利用某个稳定排序算法以最后一位数字进行排序，接着以第二个最低有效位进行排序，然后是第三个最低有效位，以此类推。若采用计数排序作为稳定排序算法，则总的时间复杂度为 $O(nd)\approx O(n)$。

**算法：**

1）取每个元素的最低有效位。

2）基于最低有效位对序列中的元素进行排序，并保持具有相同位的元素的原有次序(即稳定排序)。

3）对下一个最低有效位重复该过程。

基数排序的速度取决于内部基本操作，包括子序列的插入和删除功能以及分离所需位的过程。如果这些操作效率较低，则基数排序的运行速度就低于快速排序和归并排序

等其他算法。如果输入数值具有的位数长度不等，则还需要对附加位进行排序，这是基数排序最慢的部分之一，也是最难进行效率优化的部分之一。

由于基数排序取决于数字位或字母，所以算法的灵活性不如其他类型的排序。对于每一种不同类型的数据，基数排序都需要重写，而且如果排序顺序改变，算法也需要重写。总之，基数排序需要花费更多的时间编写代码，并且难以编写一个可以处理所有数据类型的通用基数排序算法。

对于许多需要以较快速度速进行排序的程序而言，基数排序是一个好的选择。然而存在其他更快的排序算法，这也是为什么基数排序不如其他一些排序算法使用广泛的原因。

时间复杂度为 $O(nd)\approx O(n)$，若 $d$ 取值较小。

## 10.18　拓扑排序

拓扑排序(topological sort)请参见第 9 章。

## 10.19　外部排序

外部排序(external sorting)是对一类可以处理大量数据的排序算法的统称。外部排序算法适用于处理主存不能容纳的大文件。与内部排序算法一样，外部排序也有多种算法，如外部归并排序。实际上内部排序是外部排序算法的补充。

**简单外部归并排序**

将每个磁带中的多条记录读入内存，并使用内部排序算法对这些记录排序后输出到磁带。为了简单起见，假设仅利用 100MB 的内存对 900MB 的数据进行排序。

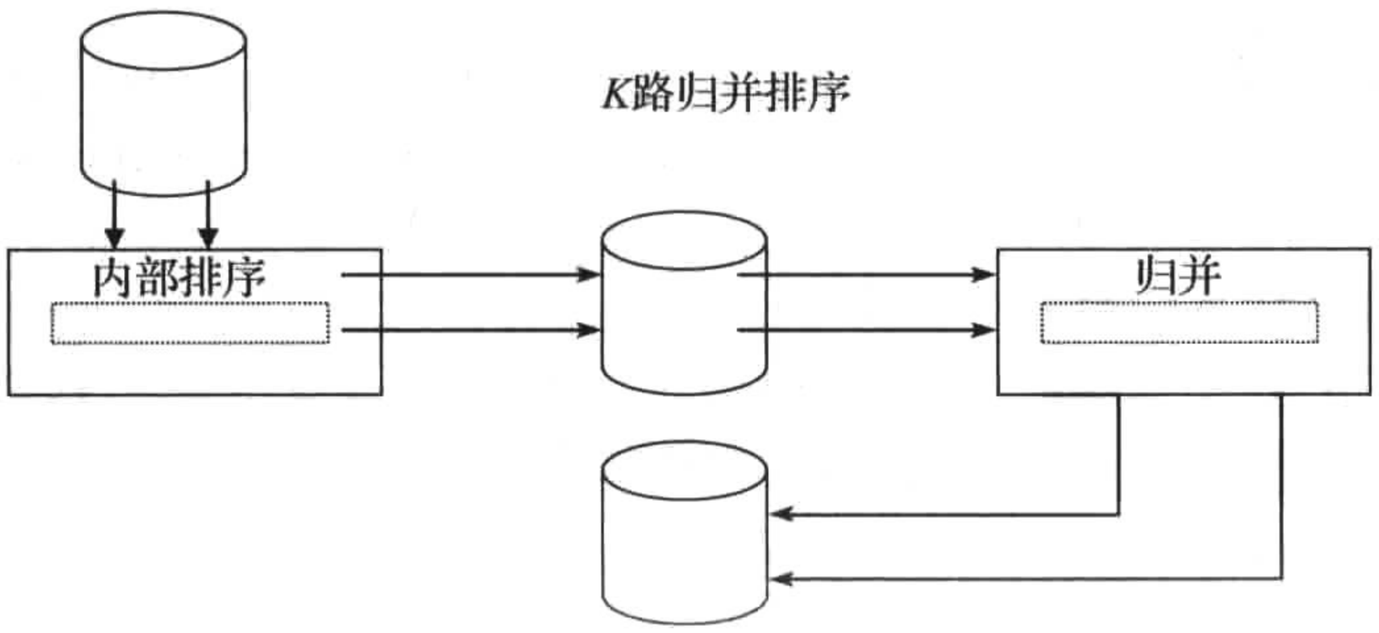
1) 将 100MB 数据读入内存，并采用某些常规方法对其排序(如快速排序)。

2) 把排序后的数据写入磁盘。

3) 重复第 1 步和第 2 步，直到所有的数据都被有序存储在大小为 100MB 的文件块中。接下来，需要将它们合并成一个单一的有序文件。

4) 将每个有序块内的前 10MB 内容(称为输入缓冲区)读入主内存(总计 90MB)，并将剩余的 10MB 作为输出缓冲区。

5) 执行 9 路归并排序(9-way Mergesort)并将结果存储在输出缓冲区。如果输出缓冲区已满，则将结果写入最后的排序文件。若 9 路输入缓冲区中的任何一个为空，则用与之对应的 100MB 文件块中的下一个 10MB 内容填补。若在该有序文件块中已经没有需要读入的数据，则将其标记为已完成，并不再将它用于归并操作。



上述算法能够进一步泛化。假设要排序的数据量超过可用内存的 $K$ 倍，则共有 $K$ 个数据块需要排序，且需要执行 $K$ 路归并排序。

若 $X$ 是可用主存的数量，则有 $K$ 个输入缓冲区和 1 个大小为 $X/(K+1)$的输出缓冲区。若输出缓冲区足够大(例如，输出缓冲区是输入缓冲区的两倍)，那么根据各种因素(硬盘读/写速度等)可以进一步提高算法性能。

2 路外部归并排序的复杂度：在每一步归并操作中需要读和写文件的每一页，假设一个文件有 $n$ 页，则需要$\lceil \log n \rceil+1$ 次归并操作，总代价为 $2n(\lceil \log n \rceil+1)$。

## 10.20 排序的相关问题

**问题 1** 给定含有重复元素的 $n$ 个数的数组 $A[0..n-1]$。给出算法，检查是否存在重复元素。假设不允许使用额外的空间(即可以使用临时变量，$O(1)$存储空间)。

**解答：** 由于不允许使用任何额外的空间，所以一个简单的方法是逐个扫描元素并检查该元素是否出现在剩余的元素中，若找到一个这样的匹配则返回真。

```
int CheckDuplicatesInArray(in A[], int n) {
        for (int i  =  0; i < n; i++)
                for (int j   = i + 1; j < n; j++)
                        if(A[i]==A[j])
                                reutrn true;
        return false;
}
```

内部循环(即以 $j$ 为索引的循环)的每次迭代使用 $O(1)$的存储空间，对固定的 $i$ 值，$j$ 循环执行 $n-i$ 次。外层循环执行 $n-1$ 次，所以整个函数的运行时间：

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

时间复杂度为 $O(n^2)$，空间复杂度为 $O(1)$。

**问题 2** 能否降低问题 1 的时间复杂度？

**解答：可以**，使用排序技术。

```
int CheckDuplicatesInArray(in A[], int n) {
        //关于堆排序的具体内容请见第7章
        Heapsort( A, n );
        for (int i  =  0; i < n-1; i++)
                if(A[i]==A[i+1])
                        reutrn true;
        return false;
}
```

Heapsort 函数的时间开销为 $O(n\log n)$，空间开销为 $O(1)$。扫描共执行 $n-1$ 次迭代，每次迭代的时间开销为 $O(1)$，总的时间开销为 $O(n\log n+n)=O(n\log n)$。因此时间复杂度为 $O(n\log n)$，空间复杂度为 $O(1)$。

**注意：** 该问题的衍生问题请参见第 11 章。

**问题 3** 给定数组 $A[0..n-1]$，其中每个元素代表选举中的一张选票，假设每张选票以一个整数来表示候选人的 ID，给出一个算法来判定谁赢得选举。

**解答：** 该问题就是在数组中寻找最大重复次数的元素。解答类似于问题 1：记录计数。

```
int CheckWhoWinsTheElection(in A[], int n) {
        int i, j, counter = A[0], maxCounter = 0, candidate;
        for (i  =  0; i < n; i++) {
                candidate = A[i];
                counter = 0;
                for (j   = i + 1; j < n; j++) {
                        if(A[i]==A[j])
                                counter++;
                }
                if(counter > maxCounter) {
                        maxCounter = counter;
                        candidate = A[i];
                }
        }
        return candidate;
}
```

时间复杂度为 $O(n^2)$，空间复杂度为 $O(1)$。

**注意：** 该问题的衍生问题请参见第 11 章。

**问题 4**　若不使用任何额外空间，能够降低问题 3 的时间复杂度？

**解答：可以**。方法是基于候选人 ID 对选票排序，然后扫描排序数组并统计哪个候选人的选票最多。由于只需要记录获胜者，所以只需要一个简单的数据结构。可以利用堆排序来解决该问题。

```
int CheckWhoWinsTheElection(in A[], int n) {
        int i, j, currentCounter = 1, maxCounter = 1;
        int currentCandidate, maxCandidate;
        currentCandidate = maxCandidate= A[0];
        //for heap sort algorithm refer Priority Queues Chapter
        Heapsort( A, n );
        for (int i  =  0; i < n; i++) {
                if( A[i] == currentCandidate)
                        currentCounter ++;
                else {    currentCandidate = A[i];
                        currentCounter = 1;
                }
                if(currentCounter > maxCounter)
                        maxCounter = currentCounter;
                else {    maxCandidate = currentCandidate;
                        maxCounter = currentCounter;
                }
        }
        return candidate;
}
```

由于堆排序的时间复杂度为 $O(n\log n)$并且是原地排序，所以除了输入数组外它只使用一个额外的 $O(1)$存储空间。排序数组共需要 $n-1$ 次扫描，每次扫描均为常数时间，因此时间开销为 $O(n)$，总时间约束是 $O(n\log n)$。

**问题 5**　是否能进一步降低问题 3 的时间复杂度？

**解答：** 在给定的问题中，候选人数较少，但是选票数量巨大，因此可以采用计数排序来求解。

时间复杂度：$O(n)$，$n$ 是数组中选票(数组元素)的数目。空间复杂度：$O(k)$，$k$ 是参与选举的候选者人数。

**问题 6**　给定包含 $n$ 个元素的数组 $A$，每个元素都是[1，$n^2$]内的整数，如何在 $O(n)$

时间内对数组排序？

**解答：** 如果对每个元素减去 1，则所有元素的取值范围变为[0，$n^2-1$]。假设所有元素都是以 $n$ 为基数的两位数，每一位的值范围为 0～$n^2-1$。采用基数排序，仅需要调用计数排序两次。排序完成后所有数再加 1。由于存在两次调用，所以复杂度为 $O(2n)\approx O(n)$。

**问题 7** 对问题 6，若取值范围为[1..$n^3$]时，该如何解决？

**解答：** 如果对每个元素减去 1，则得到元素的取值范围为[0，$n^3-1$]。假设所有元素都是以 $n$ 为基数的 3 位数，每一位的取值范围为 0～$n^3-1$。采用基数排序，仅需要调用计数排序 3 次。排序完成后所有数再加 1。由于存在 3 次调用，所以复杂度为 $O(3n)\approx O(n)$。

**问题 8** 给定一个含有 $n$ 个整数的数组，其中每个元素的值小于 $n^{100}$，能否在线性时间内对其排序？

**解答：可以**。解决思路与问题 6 和问题 7 一样。

**问题 9** 给定两个各包含 $n$ 个元素的数组 $A$ 和 $B$，对于某个数 $K$，给出一个时间复杂度为 $O(n\log n)$的算法，判断是否存在 $a\in A$ 且 $b\in B$ 使得 $a+b=K$。

**解答：** 由于要求时间复杂度为 $O(n\log n)$，所以需要使用排序来解决该问题。

```
int Find( int A[], int B[], int n, K ) {
        int i, c;
        Heapsort( A, n );                          //O(nlogn)
        for (i =0; i < n; i++) {                   //O(n)
                c = k-B[i];                        //O(1)
                if(BinarySearch(A, c))             //O(logn)
                        return 1;
        }
        return 0;
        }
```

**注意：** 该问题的衍生问题，请参见第 11 章。

**问题 10** 给定 3 个分别包含 $n$ 个元素的数组 $A$、$B$ 和 $C$。对于某个数 $K$，给出一个时间复杂度为 $O(n\log n)$的算法，判断是否存在 $a\in A$、$b\in B$ 且 $c\in C$，使得 $a+b+c=K$。

**解答：** 请参见第 11 章。

**问题 11** 给定一个包含 $n$ 个元素的数组，是否能在 $O(n+K\log K)$时间内输出已排序数组的中位数后的 $K$ 个元素？

**解答：可以**。利用中位数的划分方法找到中位数及划分，这样能得到所有大于中位数的元素集合。再找到该集合中第 $K$ 个最大元素及相应的划分，得到所有小于第 $K$ 个最大元素的元素集合。输出最终元素集合的排序序列。显然，该操作的时间复杂度为 $O(n+K\log K)$。

**问题 12** 考虑以下排序算法：冒泡排序、插入排序、选择排序、归并排序、堆排序和快速排序。哪些是稳定的排序算法？

**解答：** 假设数组 $A$ 是待排序的数组，元素 $R$ 和 $S$ 具有有相同的键值，且在数组中 $R$ 出现在 $S$ 之前，即 $R$ 在 $A[i]$位置，$S$ 在 $A[j]$位置，且 $i<j$。对于任何稳定的算法，在排序结果中 $R$ 必须先于 $S$。

**冒泡排序：** 是。仅当较小记录出现在较大记录后才改变顺序。由于 $S$ 不小于 $R$，所以 $S$ 不会在 $R$ 的前面。

**选择排序：** 不是。选择排序把数组分成已排序和未排序两部分，并在未排序部分迭代地查找最小值。在找到一个最小值 $x$ 后，若算法通过交换操作把 $x$ 移动到数组的已排

序部分，则被交换的元素可能是 $R$，即 $R$ 被移动到 $S$ 之后。这将反转 $R$ 和 $S$ 的位置，所以在一般情况下选择排序是不稳定的排序。若能避免采用交换操作，则选择排序是稳定的排序，但是时间开销将显著增加。

**插入排序：**是。当 $S$ 被插入有序子数组 $A[1..j-1]$时，只有大于 $S$ 的元素位置会发生改变。因此，在插入 $S$ 时 $R$ 的位置不会改变，即 $R$ 一直在 $S$ 之前。

**归并排序：**是。当出现具有相同键值的记录时，在左边子数组的记录具有优先权，而这些记录首先取自未排序的数组部分。因此，对具有相同键值的记录，左边子数组的记录先于后面记录输出。

**堆排序：**不是。假设 $i=1$ 且 $R$ 和 $S$ 碰巧是输入中具有最大键值的两个记录。数组堆化后 $R$ 将保持在位置 1，且在堆排序的第一次迭代中 $R$ 将被置于位置 $n$，因此 $S$ 将先于 $R$ 输出。

**快速排序：**不是。划分过程将多次交换记录位置，因此具有相同键值的两个记录在最终输出时可能交换位置。

**问题 13**　对于问题 12 中提到的算法，哪些是原地(in-place)排序算法？

**解答：**

**冒泡排序：**是，因为只需两个整数。

**插入排序：**是，因为需要保存两个整数和一条记录。

**选择排序：**是。算法可能需要保存两个整数和一条记录的空间。

**归并排序：**不是。数组需要执行归并(如果数据以链表形式表示，则算法可以实现原地排序，但这需要大量的修改)。

**堆排序：**是，因为堆和部分已排序的数组分别占据输入数组的两端。

**快速排序：**不是。由于算法是递归的且栈中保存 $O(\log n)$的活动记录。将算法修改为非递归算法虽然可行但难以实现。

**问题 14**　在快速排序、插入排序、选择排序和堆排序算法中，哪个算法所需要的交换次数最少？

**解答：**选择排序，仅需 $n$ 次交换(参见 10.6 节)。

**问题 15**　在含有 $n$ 个整数的有序数组中，判断某个整数出现的次数是否超过 $n/2$ 所需要的最少的比较次数是多少？

**解答：**请参见第 11 章。

**问题 16　对由 0、1 和 2 构成的数组排序：**给定一个只包含 0、1 和 2 的数组 $A[]$，给出一个算法对数组 $A[]$排序，要求算法将所有的 0 放在最前面，随后是 1，最后是 2。

**例子：**输入＝{0，1，1，0，1，2，1，2，0，0，0，1}，输出＝{0，0，0，0，0，1，1，1，1，1，2，2}

**解答：**使用计数排序。由于只有 3 个元素且最大的值是 2，所以只需要一个包含 3 个元素的临时数组。

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**注意：**该问题的衍生问题请参见第 11 章。

**问题 17**　是否存在解决问题 16 的其他方法？

**解答：**使用快速排序。既然已知数组序列只由 3 个元素(0、1 和 2)构成，那么可以选择 1 作为快速排序的枢轴。快速排序只需要一次扫描就可以将所有的 0 和 2 分别移到 1 的左边和右边，并确定 1 的正确位置。

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**注意：**更有效的算法请参见第 11 章。

**问题 18** 如何找到数组中出现次数最多的数？

**解答：**一个简单的方法是对给定的数组进行排序并扫描已排序的数组，在扫描数组的过程中记录出现次数最多的元素。

**算法：**

```
QuickSort(A, n);
int i, j, count=1, Number=A[0], j=1;
for(i=1;i < n;i++) {
        if(arr[j]==A) {
                count++;
                Number=A[j];
        }
        j=i;
}
System.out.println("Number:" + Number + " , count: " + count);
```

时间复杂度＝排序所需时间＋扫描所需时间＝$O(n\log n)+O(n)=O(n\log n)$，空间复杂度为 $O(1)$。

**注意：**该问题的衍生问题，请参见第 11 章。

**问题 19** 是否存在解决问题 18 的其他方法？

**解答：**使用二叉树。创建一棵带附加计数字段的二叉树，该计数字段用来统计某个输入元素出现的次数。在创建一棵二叉搜索树(BST)后，中序遍历 BST 生成排序序列。在中序遍历的同时记录出现次数最多的元素。

时间复杂度为 $O(n)+O(n)\approx O(n)$。第一个参数对应于构建 BST，第二个参数对应于中序遍历。

空间复杂度为 $O(2n)\approx O(n)$，因为 BST 中的每个结点需要两个额外的指针。

**问题 20** 是否还存在解决问题 18 的其他方法？

**解答：使用散列表**。对给定数组的每个元素使用一个计数器，元素每出现一次，对应的计数器加 1。最终只需返回具有最大计数的元素。

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。构建散列表需要 $O(n)$。

**注意：**更多有效算法请参见第 11 章。

**问题 21** 给定一个大小为 2GB 的文件，其中文件的每一行是一个字符串。可以使用哪种排序算法对文件进行排序，为什么？

**解答：**当有 2GB 的大小限制时，这说明无法将所有的数据读入主存。

**算法：**有多少可用内存？假设有 $X$ MB 的可用内存，则把文件分成 $K$ 块，满足 $X\times K\geqslant 2$GB。

- 将每个文件块读入内存，并对文件块中的行排序(可使用任何 $O(n\log n)$的算法)。
- 将排序后的行写回文件。
- 将下一个文件块读入内存并排序。
- 完成上述操作后，对有序文件块逐个归并。当一个文件块操作完成时，从相关文件块读入更多数据。

上述算法就是外部排序。步骤 3 和步骤 4 称为 $k$ 路归并。数据大小决定是否需要采用外部排序，当数据较大时，无法将数据一次性读入主存，因此需要基于磁盘实现的排序

算法。

**问题 22　近似有序**(Nearly sorted)：给定一个含有 $n$ 个元素的数组，其中每个元素距离自己的正确位置最多有 $K$ 个位置，设计一个时间复杂度为 $O(n\log K)$的排序算法。

**解答：**把数组中的元素分成大小为 $K$ 的 $n/K$ 组，并对每一组数据采用时间复杂度为 $O(K\log K)$的归并排序算法进行排序。这能够保证如下性质：没有任何元素相距自己的正确位置超过 $K$ 个元素。随后将每组(包含 $K$ 个元素)和其左边的组进行合并。

**问题 23**　是否存在解决问题 22 的其他方法？

**解答：**选择前 $K$ 个元素并将它们插入一个二叉堆中，再将数组中下一个元素插入堆中并从堆中删除最小元素，重复此过程。

**问题 24　归并 $K$ 组有序序列：**给定 $K$ 组共有 $n$ 个元素的有序序列，给出算法生成一个包含所有 $n$ 个元素的有序序列，要求该算法的时间复杂度为 $O(n\log K)$。

**解答：**归并 $K$ 组有序序列的简单算法：假设每组各有 $n/K$ 个元素。取第一组序列，使用归并排序等线性时间算法与第二组序列进行归并，然后把包含 $2n/K$ 个元素的归并结果序列再与第三个序列进行归并，然后把新归并产生的含 $3n/K$ 个元素的序列再与第四组序列进行归并，重复该过程直到最终生成一个含 $n$ 个元素的有序列表。

时间复杂度为每次迭代归并 $n/K$ 个元素。

$$T(n)=\frac{2n}{K}+\frac{3n}{K}+\frac{4n}{K}+\cdots+\frac{Kn}{K}(n)=\frac{n}{K}\sum_{i=2}^{K}i$$

$$T(n)=\frac{n}{K}\left[\frac{K(K+1)}{2}\right]\approx O(nK)$$

**问题 25**　是否可以降低问题 24 的时间复杂度？

**解答：**一个可行的方法是重复地对有序序列进行配对，并将其依次归并。这种方法也可以视为执行归并排序的一个尾部分量，这里分析清晰，也称为锦标赛排序方法(Tournament Method)。锦标赛排序方法的最大深度是 $\log K$ 且每次迭代都要扫描所有的 $n$ 个元素。

时间复杂度为 $O(n\log K)$。

**问题 26**　是否还存在解决问题 24 的其他方法？

**解答：**另一种方法是对 $K$ 组的每一组中的最小元素使用一个优先队列。在每一步中，输出优先队列中存储的最小元素，并确定该元素来自 $K$ 组序列中的哪一个序列，接着将该序列中的下一个最小元素插入优先队列中。由于使用优先队列，所以其最大深度是 $\log K$。

时间复杂度为 $O(n\log K)$。

**问题 27**　哪一种排序方法更适合对链表排序？

**解答：**归并排序是更好的选择。由于需要中间结点把给定的链表分成两个长度相等的子链表，所以初看起来归并排序可能不是一个好的选择。通过交替地将链表中的元素分配给两个子链表，可以很容易地解决该问题(参见第 3 章)，然后递归地对这两个链表排序并将结果归并为一个单一的链表，这样就可以得到排好的链表[27]。

```
ListNode LinkedListMergeSort(ListNode first) {
        ListNode list1HEAD = null;
        ListNode list1TAIL = null;
        ListNode list2HEAD = null;
```

```
ListNode list2TAIL = null;
if(first==null || first.next==null)
        return first;
while (first != null) {
        Append(first, list1HEAD, list1TAIL);
        if(first != null)
                Append(first, list2HEAD, list2TAIL);
}
list1HEAD = LinkedListMergeSort(list1HEAD);
list2HEAD = LinkedListMergeSort(list2HEAD);
return Merge(list1HEAD, list2HEAD);
}
```

**注意**：Append()将第一个参数链接到一单向链表的尾部，该单向链表的头结点和尾结点分别由第二个和第三个参数定义。

由于每个文件可以看作一个只能被顺序访问的链表，所以所有的外部排序算法可用于对链表排序。对于双链表排序，可以将其指向下一个结点的指针字段看作单向链表，并在一次额外扫描对其排序后重构其指向前一个结点的指针字段。

**问题 28** 是否可以用快速排序对链表排序？

**解答**：由于无法在单向链表中回退，所以原始快速排序不能用于单链表排序。可以对原始快速排序进行修改使其能够用于单链表排序。

考虑如下修改快速排序实现的方式。用输入链表的第一个结点作为枢轴并移到相等端。随后将每个结点与枢轴比较，如果结点值小于(等于或大于)枢轴则将结点移动到小端(相等端或大端)，然后对小端和大端进行递归排序。最后，将小端、相等端和大端合并成一个单向链表，从而生成一个有序的单向链表。

Append()将第一个参数链接到一个单向链表的尾部，该单向链表的头结点和尾结点分别由第二个和第三个参数定义。返回后，第一个参数将被修改并使它的相等端指向链表的下一个结点。Join()将由第三个和第四个参数定义的头结点和尾结点的链表添加到由第一个和第二个参数定义的头结点和尾结点的链表的后面。为了简单起见，把第一个和第四个参数作为结果链表的头结点和尾结点[27]。

```
void Qsort(ListNode first, ListNode last){
        ListNode lesHEAD=null, lesTAIL=null;
        ListNode equHEAD=null, equTAIL=null;
        ListNode larHEAD=null, larTAIL=null;
        ListNode current = first;
        int pivot, info;
if(current == null)
        return;
pivot = current.getData();
Append(current, equHEAD, equTAIL);
while (current != null) {
        info = current.getData();
        if(info < pivot)
                Append(current, lesHEAD, lesTAIL)
        else if(info > pivot)
                Append(current, larHEAD, larTAIL)
        else    Append(current, equHEAD, equTAIL);
}
Quicksort(lesHEAD, lesTAIL);
```

```
        Quicksort(larHEAD, larTAIL);
        Join(lesHEAD, lesTAIL,equHEAD, equTAIL);
        Join(lesHEAD, equTAIL,larHEAD, larTAIL);
        first = lesHEAD;
        last = larTAIL;
}
```

**问题 29**　给定一个含 100 000 个像素颜色值的数组，数组中的每个值是[0，255]中的整数。哪一种排序算法更适合对该数组排序？

**解答：**计数排序。只有 256 个键值，所以辅助数组的大小为 256，并且只需要对数据遍历两遍，时间开销和空间开销都会较小。

**问题 30**　类似问题 29，如果有一个含 100 000 条记录的电话簿，哪一种排序算法最适合它？

**解答：**桶排序。在桶排序中桶由电话号码的最后 7 位数字定义，这需要一个大小为 1 000万的辅助数组，并且仅需要遍历一遍在磁盘上存储的数据。每个桶包含所有区号不同但最后 7 位数相同的电话号码。随后可以使用选择或插入排序对有限数目的区号进行桶排序。

**问题 31**　给出一个归并 $K$ 有序列表的算法。

**解答：**请参见第 7 章。

**问题 32**　给定一个包含数十亿数字的大文件，找出文件中最大的 10 个数字。

**解答：**请参见第 7 章。

**问题 33**　已知两个有序数组 $A$ 和 $B$，$A$ 的大小是 $m+n$ 但只包含 $m$ 个元素，$B$ 的大小是 $n$ 且包含 $n$ 个元素。将两个数组中的元素合并到第一个大小为 $m+n$ 的数组中并对其排序。

**解答：**该问题的技巧在于用最大的元素从后向前填充目标数组，最终将得到一个合并的有序数组。

```
void Merge(int[] A[], int m, int B[], int n) {
        int count = m;
        int i = n – 1, j = count – 1, k = m - 1;
        for(;k>=0;k--) {
                if(B[i] > A[j] || j < 0) {
                        A[k] =B[i];
                        i--;
                        if(i<0)
                                break;
                }
                else {    A[k] = A[j];
                        j--;
                }
        }
}
```

时间复杂度为 $O(m+n)$。空间复杂度为 $O(1)$。

**问题 34　螺母和螺栓问题：**给定一组大小不一的 $n$ 个螺母和 $n$ 个螺栓，这些螺母和螺栓之间存在一一对应关系，为每个螺母寻找与其相对应的螺栓。假设只能拿螺母与螺栓比较，不能将螺母与螺母、螺栓和螺栓进行比较。

**另一种描述：**有一个盒子里装着螺栓和螺母。假设有 $n$ 个螺母和 $n$ 个螺栓，每个螺母仅能匹配一个螺栓(反之亦然)。通过尝试匹配螺栓和螺母，可以观察哪一个更大，但不

能直接比较两个螺栓或两个螺母。设计一个高效的螺母和螺栓匹配算法。

**解答：蛮力法：**从第一个螺栓开始，将每个螺母与其比较直到找到匹配的螺母。在最坏情况下，需要 $n$ 次比较。对后续所有剩余螺栓重复该过程，时间复杂度为 $O(n^2)$。

**问题 35** 能否降低问题 34 的复杂度？

**解答：**在问题 34 中最坏情况下（如果螺栓升序排列而螺母降序排列）的复杂度为 $O(n^2)$，其分析结果与快速排序一致，对其改进仍然具有相同的复杂度量级。

为了降低最坏情况下的复杂度，可以随机选择一个螺栓与螺母匹配，而不是每次都选择第一个螺栓。这种随机选择方法降低了出现最坏情况的可能性，但在最坏情况下复杂度仍然为 $O(n^2)$。

**问题 36** 对于问题 34，还能进一步降低复杂度吗？

**解答：**可利用分治技术解决该问题，解决方法与随机快速排序非常类似。为了简单起见，假定螺栓和螺母分别用两个数组 $B$ 和 $N$ 表示。

该算法首先执行一个如下所示的划分操作：随机选择一个螺栓 $B[i]$，使用该螺栓，将螺母数组重新排列分成三组元素：

- 比 $B[i]$小的螺母。
- 与 $B[i]$匹配的螺母。
- 比 $B[i]$大的螺母。

接下来，使用与 $B[i]$匹配的螺母将螺栓数组进行类似划分。这对划分操作可以很容易地在 $O(n)$时间内实现，而且可以对螺栓和螺母进行较好的划分，即作为枢轴的螺栓和螺母彼此相互匹配，而其他所有的螺栓和螺母都正确地分布在这些枢轴点的两边——较小螺母和螺栓在枢轴点的前面，较大的螺母和螺栓在枢轴的后面。接着将该算法递归地应用于枢轴左边和右边位置的子数组以便匹配剩余的螺栓和螺母。可以假设经过 $n$ 次递归调用将正确地匹配剩余的螺栓。

可以使用类似随机快速排序的分析方法来分析该算法的运行时间。因此，根据快速排序的分析方法可知，该算法的时间复杂度为 $O(n\log n)$。

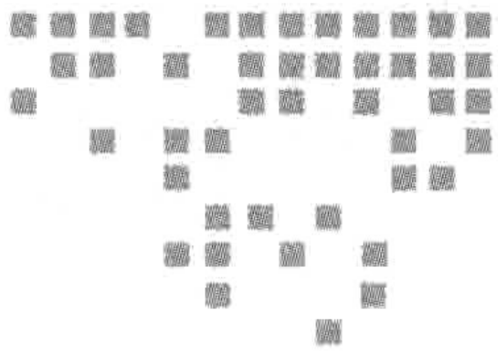
**另一种分析方法：**可以对快速排序进行小的改进来解决该问题。假设选择最后一个元素作为枢轴且该元素是一个螺母。当向下遍历数组时，该螺母只比较螺栓。这将对螺栓数组进行划分，每一个小于枢轴螺母的螺栓将在左边，而大于枢轴螺母的螺栓将在右边。

当向下遍历数组时，将会发现与枢轴螺母匹配的螺栓。随后再次使用匹配的螺栓对螺母数组进行划分，因此所有小于匹配螺栓的螺母将分在左边，所有大于匹配螺栓的螺母将分在右边，并在左边和右边数组中递归地调用该算法。

时间复杂度为 $O(2n\log n)\approx O(n\log n)$。

**问题 37** 给定一棵二叉树，利用中序遍历，能否以 $O(n)$时间有序输出元素吗？

**解答：可以，**当是一棵二叉搜索树(BST)时。具体信息请参见第 11 章。



# 第 11 章 Chapter 11

# 查　找

## 11.1　什么是查找

在计算机科学中，查找(或称为搜索)就是从一个项目的集合中寻找某个具有特定属性的项目的过程。项目可以是存储在数据库中的记录、数组中的简单数据元素、文件中的文本、树中的结点、图中的顶点和边，或者其他搜索空间的元素。

## 11.2　为什么需要查找

查找是计算机科学中的一个核心算法。众所周知，今天的计算机存储了大量的信息。要想有效地获取这些信息，需要非常高效的查找算法。

有很多组织数据的方式可以提高查找效率。这意味着，如果以合适的顺序保存数据，则很容易查找所需要的元素。排序是一种用来将元素有序化的技术之一。本章将探讨不同的查找算法。

## 11.3　查找的类型

下面是本章要讨论的查找类型。

- 无序线性查找。
- 排序/有序线性查找。
- 二分查找。
- 符号表和散列。
- 字符串查找算法：键树、三叉搜索树和后缀树。

### 1. 无序线性查找

假设给定一个数组，其元素的排列顺序是未知的，即数组中的元素是无序的。在这种情况下，如果要查找某个元素，必须扫描整个数组才能判断该元素是否在给定的数组中。

```
int UnsorteddLinearSearch (int A[], int n, int data) {
        for (int i  = 0; i < n; i++) {
                if(A[i] == data)
                        return i;
        }
        return -1;
}
```

时间复杂度为 $O(n)$，在最坏的情况下，需要扫描整个数组。空间复杂度为 $O(1)$。

### 2. 排序/有序线性查找

如果数组的元素已经排序，则在许多情况下都不需要扫描整个数组来判断该元素是否在给定的数组中。从如下所示的算法中可以看出，在任何时候，如果 $A[i]$的值大于要查找的 data 的值，则直接返回－1，而不用再查找数组中剩余的元素。

```
int SortedLinearSearch(int A[], int n, int data) {
        for (int i  = 0; i < n; i++) {
                if(A[i] == data)    return i;
                else if(A[i] > data)
                        return -1;
        }
        return -1;
}
```

这个算法的时间复杂度为 $O(n)$。这是因为在最坏的情况下，算法需要扫描整个数组。但在平均情况下，即使增长率相同，算法还是降低了复杂性。空间复杂度为 $O(1)$。

**注意：** 通过以更快的速度递增索引(如步长为 2)可以进一步改进上述算法。这将减少在排序列表中查找元素所需要的比较次数。

### 3. 二分查找

考虑在英文字典中查找某个单词的问题。通常，我们直接从某些相近页码(如字典的中间页)处开始查找。如果那个单词恰巧是所要寻找的，那么查找完成。如果需要查找的页位于选定页之前，那么就对字典的前半部分采用同样的方法进行查找。否则，对字典的后半部分应用相同的方法进行查找。二分查找就是采用这种方式。采用这种策略的算法就是二分查找算法。

```
//迭代的二分查找算法
int BinarySearchIterative[int A[], int n, int data) {
        int low =  0, high = n-1;
        while (low <= high) {
        mid  = low + (high-low)/2; //避免溢出
                if(A[mid] == data)
                        return mid;
                else if(A[mid] < data)
                        low =  mid + 1;
                else      high =  mid - 1;
        }
        return -1;
}
//递归的二分查找算法
int BinarySearchRecursive[int A[], int low, int high, int data) {
        int mid  = low + (high-low)/2; //避免溢出
```

```
        if(A[mid] == data)          return mid;
        else if(A[mid] < data)
                return BinarySearchRecursive (A, mid + 1, high, data);
        else    return BinarySearchRecursive (A, low, mid - 1 , data);
        return -1;
}
```

二分查找的循环次数为 $T(n)=T(\frac{n}{2})+\Theta(1)$。这是因为算法始终只考虑输入列表的一半，而忽略另一半。采用分治法主定理可得，$T(n)=O(\log n)$。

时间复杂度为 $O(\log n)$。空间复杂度为 $O(1)$（对于迭代算法）。

### 4. 基本查找算法比较

| 实现 | 最坏情况 | 平均情况 | 实现 | 最坏情况 | 平均情况 |
|---|---|---|---|---|---|
| 无序数组 | $n$ | $\frac{n}{2}$ | 有序列表 | $n$ | $\frac{n}{2}$ |
| 有序数组（二分查找） | $\log n$ | $\log n$ | 二叉搜索树（偏斜树） | $n$ | $\log n$ |
| 无序列表 | $n$ | $\frac{n}{2}$ | | | |

注：关于二叉搜索树的讨论请参见第 6 章。

## 11.4　符号表和散列

请参见第 13 章和第 14 章。

## 11.5　字符串查找算法

请参见第 15 章。

## 11.6　查找的相关问题

**问题 1**　给定一个长度为 $n$ 的数组，给出一个算法，判定在该数组中是否存在重复的元素。

**解答：**这是最简单的问题之一。一个显而易见的答案是，穷尽搜索整个数组，检查是否存在重复元素。即对于每个输入元素，都要检查数组中是否存在与其具有相同值的元素。只需要两个简单的循环就可以解决该问题。实现代码为：

```
void CheckDuplicatesBruteForce(int A[], int n) {
        for(int i = 0; i < n; i++) {
                for(int j = i+1; j < n; j++) {
                        if(A[i] == A[j])     {
                                System.out.println("Duplicates exist:" + A[i]);
                                return;
                        }
                }
        }
        System.out.println("No duplicates in given array.");
}
```

时间复杂度为 $O(n^2)$，由于有两个嵌套的 for 循环。

空间复杂度为 $O(1)$。

**问题 2** 能否降低问题 1 的解决方法的复杂度？

**解答：可以。**对给定的数组进行排序，排序后，值相等的所有元素相邻。则只需扫描该有序数组，检查是否存在值相等的相邻元素。

```
void CheckDuplicatesSorting(int A[], int n) {
        Sort(A, n);                                    //对数组排序
        for(int i = 0; i < n-1; i++) {
                if(A[i] == A[i+1]) {
                        System.out.println("Duplicates exist: " + A[i]);
                        return;
                }
        }
        System.out.println("No duplicates in given array.");
}
```

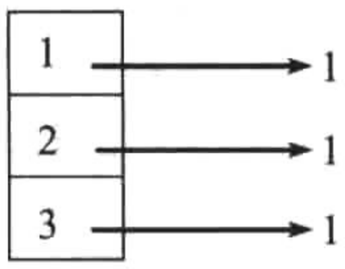
时间复杂度为 $O(n\log n)$，主要在排序。

空间复杂度为 $O(1)$。

**问题 3** 是否有其他方法可以解决问题 1？

**解答：可以，**采用散列表。散列表是用于实现字典的一种简单而有效的方法，其查找一个元素的平均时间是 $O(1)$，最坏情况下的时间是 $O(n)$。有关散列算法的具体内容请参见第 14 章。以数组 $A=\{3, 2, 1, 2, 2, 3\}$为例。

扫描输入数组，并把元素插入散列表中。对于每一个插入的元素，将 counter 设置为 1(假设初始时元素的值都设为 0)。这表明对应的元素已经存在于散列表中。对于给定的数组，散列表如下图所示(插入前三个元素 3，2，1 后)：



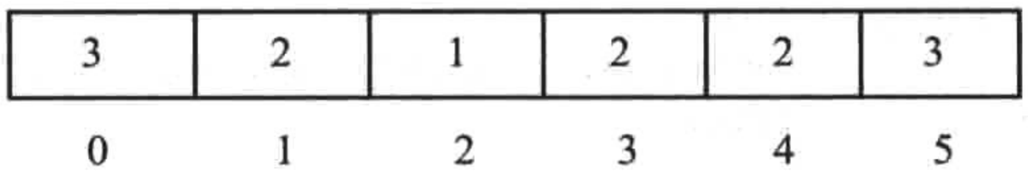

现在，如果要插入 2，由于 2 的计数值已经是 1，则说明元素已经出现了两次。

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

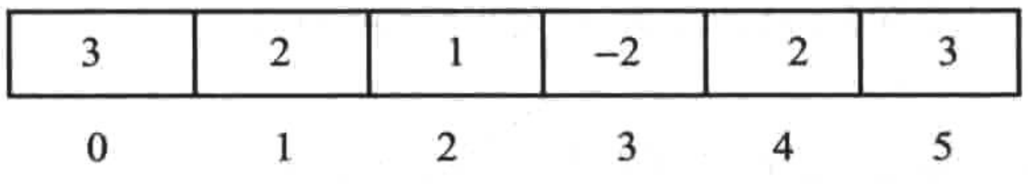
**问题 4** 能否进一步降低问题 1 的解决方法的复杂度？

**解答：**假设数组中的元素都是取值范围在 $0\sim n-1$ 的正数。对于每个元素 $A[i]$，寻找索引为 $A[i]$的数组元素。即选择 $A[A[i]]$且将其标记为 $-A[A[i]]$(求 $A[A[i]]$的负数)。重复该过程，直到遇到的元素值已经为负数。如果存在这样一个元素，则说明在给定的数组中存在重复元素。以数组 $A=\{3, 2, 1, 2, 2, 3\}$为例。

初始时，




在步骤 1 中，对 $A[\text{abs}(A[0])]$求负，




在步骤 2 中，对 $A[\text{abs}(A[1])]$求负，

| 3 | 2 | −1 | −2 | 2 | 3 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

在步骤 3 中，对 $A[\text{abs}(A[2])]$求负，

| 3 | −2 | −1 | −2 | 2 | 3 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

在步骤 4 中，对 $A[\text{abs}(A[3])]$求负，

| 3 | −2 | −1 | −2 | 2 | 3 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

在步骤 4 中，可以发现 $A[\text{abs}(A[3])]$已经为负数。这表明该元素已经被访问两次。

```
void CheckDuplicates(int A[], int n) {
        for(int i = 0; i < n; i++) {
                if(A[Math.abs(A[i])] < 0) {
                        System.out.println("Duplicates exist: " + A[i]);
                        return;
                }
                else    A[A[i]] = - A[A[i]];
        }
        System.out.println("No duplicates in given array.");
}
```

时间复杂度为 $O(n)$，因为只需要扫描一次；空间复杂度为 $O(1)$。

**注意：**

- 该方法不适用于只读数组。
- 该方法仅适用于所有元素为正数的数组。
- 如果数组中元素的取值范围不在 $0\sim n-1$，则可能出现异常。

**问题 5**　已知一个包含 $n$ 个数字的数组。给出算法寻找数组中出现次数最多的元素。

**蛮力法：**一个简单的解决方法是，对于每个输入元素，检查是否有与该元素值相同的元素，并且每出现一次这样的数，计数器加 1。每次将当前计数器的值与最大计数器的值进行比较，如果该值大于最大计数器的值，则更新最大计数器的值。这样，只需要两个简单的 for 循环就可以解决该问题。

```
int CheckDuplicatesBruteForce(int A[], int n) {
        int counter =0, max=0;
        for(int i = 0; i < n; i++) {
                counter=0;
                for(int j = 0; j < n; j++) {
                        if(A[i] == A[j])
                                counter++;
                }
                if(counter > max) max = counter;
        }
        return max;
}
```

时间复杂度为 $O(n^2)$，由于使用两个嵌套的 for 循环；空间复杂度为 $O(1)$。

**问题 6**　能否降低问题 5 解决方法的复杂度？

**解答：可以。**对给定数组进行排序。当所有元素排序后，值相等的元素相邻。只需要对已排序的数组再进行一次扫描并检查哪个元素出现的次数最多。

时间复杂度为 $O(n\log n)$，由于采用了排序；空间复杂度为 $O(1)$。

**问题 7** 是否存在其他方法可以解决问题 5？

**解答：可以，**采用散列表。对于输入的每个元素，记录该元素在输入数组中出现的次数。计数器的值表示该元素出现的次数。

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 8** 对于问题 5，能否降低时间复杂度？假设所有元素的取值范围均为 $0\sim n-1$。

**解答：可以。**通过两次扫描就可以解决该问题。因为元素会重复出现，所以无法使用类似问题 3 求负数的方法来解决该问题。在第一次扫描时，对每个元素增加 $n$ 而不求负数，即每扫描一个元素，都将数组长度加到该元素上(使该元素的值增加数组的长度)。在第二次扫描时，首先将元素值除以 $n$，并返回值最大的元素。对应的代码实现如下所示。

```
void MaxRepititions(int A[], int n){
        int i =  0, max = 0, maxIndex;
        for(i = 0; i < n; i++)
                A[A[i]%n] +=n;
        for(i = 0; i < n; i++) {
                if(A[i]/n > max) {
                        max = A[i]/n;
                        maxIndex =i;
                }
        }
        return maxIndex;
}
```

**注意：**

- 该方法不适用于只读数组。
- 该方法仅适用于所有元素为正数的数组。
- 如果数组中元素的取值范围不在 $0\sim n-1$，则可能出现异常。

时间复杂度为 $O(n)$，由于没有嵌套的 for 循环。空间复杂度为 $O(1)$。

**问题 9** 已知一个由 $n$ 个数字组成的数组，给出算法寻找该数组中重复出现的第一个元素。例如，在数组 $A=\{3, 2, 1, 2, 2, 3\}$ 中，第一个重复出现的数字是 3(而不是 2)。即需要在所有重复的元素中返回第一个元素。

**解答：**可以采用解决问题 1 的蛮力法来解决该问题。对于每个元素，检查是否存在重复的元素，任何最先重复的元素将被返回。

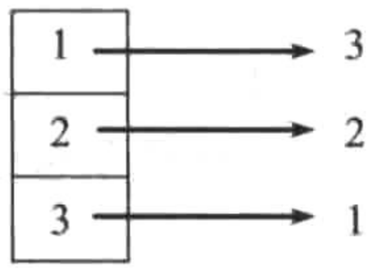
**问题 10** 对于问题 9，能否采用排序技术？

**解答：不行。**为了给出一个反例，考虑如下数组。例如，数组 $A=\{3, 2, 1, 2, 2, 3\}$。对它进行排序后，得到的新数组为 $A=\{1, 2, 2, 2, 3, 3\}$，在该有序数组中，第一次重复出现的元素是 2 而不是正确答案 3。

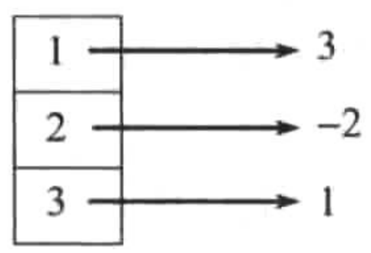
**问题 11** 对于问题 9，能采用散列法吗？

**解答：可以。**但是采用解决问题 3 所使用的简单散列法是不行的。例如，假设输入数组 $A=\{3, 2, 1, 2, 3\}$，则第一个重复出现的元素是 3，但如果采用简单散列法将得到 2。这是因为 2 在 3 之前先出现两次。现在，对散列表的操作方法进行改变从而得到第一个重复出现的元素。假设初始时不存储 1，而是存储元素在数组中的位置。因此，散列表

如下所示(插入 3，2，1 后)：




现在，如果再遇到元素 2，则对散列表中 2 对应的值求负。也就是说，使其计数器的值为－2。散列表中的负值表明该元素已经被访问两次。类似地，对于 3 来说(输入的下一个元素)，也将其在散列表中对应的值求负数，最后得到的散列表如下所示：




当处理完整个数组后，扫描散列表并返回最高的负索引值(在该例中是－1)。最高的负值表示该元素被最先访问(在所有重复的元素中)并重复出现。

**当有元素重复出现的次数超过两次时应该如何解决？** 在本例中，只需要在扫描过程中跳过计数值已经为负的元素即可。

**问题 12**　对于问题 9，能否采用解决问题 3 的方法(求负操作)来解决呢？

**解答：不行。**可以给出一个反例。对于数组 $A=\{3, 2, 1, 2, 2, 3\}$，第一次重复出现的元素是 3。但若采用求负方法，结果是 2。

**问题 13　发现缺失的数：**已知一个由 $n-1$ 个取值范围在 $1\sim n$ 的整数组成的列表。假设列表中没有重复出现的整数，则列表中缺失某个整数。给出算法寻找该缺失的整数。

**例子：**I/P：[1，2，4，6，3，7，8]　O/P：5

**蛮力法：**一个解决该问题的简单方法是，对于每一个在 $1\sim n$ 的数，检查该数是否在给定的数组中。

```
int FindMissingNumber(int A[], int n){
        int i, j, found=0;
        for (i = 1; i < =n; i ++) {
                found = 0;
                for (j = 0; j < n; j ++) {
                        if(A[j]==i)
                                found = 1;
                }
                if(!found) return i;
        }
        return -1;
}
```

时间复杂度为 $O(n^2)$。空间复杂度为 $O(1)$。

**问题 14**　对于问题 13，能否使用排序方法来解决？

**解答：可以。**对列表进行排序，将元素按照递增顺序排列，此时再通过一次扫描就能找到缺失的数字。

时间复杂度为 $O(n\log n)$。空间复杂度为 $O(1)$。

**问题 15**　对于问题 13，能否采用散列法解决？

**解答：可以。**扫描输入数组并在散列表中插入元素。对于插入的元素，使计数器的

值为 1(假设初始时所有的值都是 0)。这表明相应的元素已经出现过。只需扫描散列表并返回计数器值为 0 的元素即可。

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 16** 对于问题 13，能否降低其时间复杂度？

**解答：** 使用求和公式。

1）对 $n$ 个数求和，sum=$n\times(n+1)/2$。

2）用 sum 减去输入序列中的所有元素，可以得到缺失的数。

时间复杂度为 $O(n)$，用于扫描整个数组。

**问题 17** 在问题 13 中，如果数字的总和超出所允许的最大整数范围，则可能会导致整数溢出而不能得到正确的答案。能否解决该问题？

**解答：**

1）对数组中所有的元素执行异或(XOR)操作，设异或后的结果为 $X$。

2）对从 1～$n$ 的所有元素执行异或(XOR)操作，设异或后的结果为 $Y$。

3）对 $X$ 与 $Y$ 进行异或可得到缺失的数。

```
int FindMissingNumber(int A[], int n){
        int i, X, Y;
        for (i = 0; i < n; i ++)
                X ^= A[i];
        for (i = 1; i <= n; i ++)
                Y ^= i;
        //事实上仅需一个变量
        return X ^ Y;
}
```

时间复杂度为 $O(n)$，用于扫描整个数组。空间复杂度为 $O(1)$。

**问题 18 发现出现次数是奇数的数：** 给定一个正数数组，除了一个元素的出现为奇数次外，其他元素的出现均为偶数次。给出算法，在 $O(n)$时间且常数空间内求得该数。

**例子：** I/P=[1，2，3，2，3，1，3] O/P=3

**解答：** 对所有元素执行按位异或操作，可以得到出现次数是奇数的元素，因为 $A$ XOR $A=0$。

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 19 在给定的数组中发现两个重复出现的元素：** 给定一个含有 $n+2$ 个元素的数组，每个元素的取值范围在 1～$n$，并且除了两个元素出现两次外，其他元素均出现一次。求出这两个重复的元素。例如，假设数组为 4，2，4，5，2，3，1，这里 $n=5$。数组中共有 $n+2=7$ 个元素，除了 2 和 4 出现两次外，其他只出现一次。所以，输出结果应当是 4 和 2。

**解答：** 一个简单的方法是，针对输入数组中的每一个元素，都对整个数组扫描一遍。这意味着要使用两个循环。外层循环用来依次选择每个元素，内层循环用来统计所选择元素出现的次数。算法的实现代码如下，假设使用 $n+2$ 作为参数调用 PrintRepeatedElements 函数，其中 $n+2$ 表示数组的大小。

```
void PrintRepeatedElements(int A[], int n){
        for(int i = 0; i < n; i++)
                for(int j = i+1; j <n; j++)
                        if(A[i] == A[j])
                                System.out.println( A[i]);
}
```

时间复杂度为 $O(n^2)$。空间复杂度为 $O(1)$。

**问题 20**　对于问题 19，能否降低其时间复杂度？

**解答：**使用任何比较排序算法对数组进行排序，判断是否出现连续相同值的元素。

时间复杂度为 $O(n\log n)$。空间复杂度为 $O(1)$。

**问题 21**　对于问题 19，能否降低其时间复杂度？

**解答：**使用计数数组。该解决方案类似于使用散列表。为了简单起见，用数组来存储计数。对数组遍历一次，并使用一个大小为 $n$ 的临时数组 count[]记录数组中所有元素的计数。当某个元素的计数已经设置时，将其作为重复元素输出。假设使用 $n+2$ 作为参数调用 PrintRepeatedElements 函数，其中 $n+2$ 表示数组的大小。

```
void PrintRepeatedElements(int A[], int n){
        int *count = (int *)calloc(sizeof(int), (n - 2));
        for(int i = 0; i < size; i++) {
                count[A[i]]++;
                if(count[A[i]] == 2)
                        System.out.println( A[i]);
        }
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 22**　考虑问题 19。假设数字的取值范围在 $1\sim n$。是否存在其他解决该问题的方法？

**解答：**可以采用异或操作。假设重复的数字是 $X$ 和 $Y$，如果对数组中所有元素及从 $1\sim n$的所有元素进行异或操作，则结果将是 $X$ 异或 $Y$ 的值。在 $X$ XOR $Y$ 的二进制表示中，1 对应于 $X$ 和 $Y$ 之间的不同位。如果 $X$ XOR $Y$ 的第 $k$ 位是 1，则可以对数组中所有元素及从 $1\sim n$ 的所有整数执行异或操作，并且其第 $k$ 位的值必定是 1。该结果将是 $X$ 或 $Y$。

```
void PrintRepeatedElements (int A[], int size){
        int XOR = A[0];
        int i, right_most_set_bit_no, X= 0, Y = 0;
        for(i = 1; i < size; i++)                /*对A[]中所有元素执行异或操作*/
                XOR ^= A[i];
        for(i = 1; i <= n; i++) /*对所有{1,2...,n}执行异或操作  */
                XOR ^= i;
        right_most_set_bit_no = XOR & ~( XOR -1);     // 获取right_most_set_bit_no最右位

        /*根据最右位将所有元素分为两组*/
        for(i = 0; i < size; i++) {
                if(A[i] & right_most_set_bit_no)
                        X = X^ A[i];  /*对A[]中第一个数据集异或*/
                else      Y = Y ^ A[i];  /*对A[]中对二个数据集异或*/
        }
        for(i = 1; i <= n; i++) {
                if(i & right_most_set_bit_no)
                        X = X ^ i; /*对A[]中第一个数据集及{1,2,...,n}异或*/
                else      Y = Y ^ i; /*对A[]中第二个数据集及{1,2,...,n}异或*/
        }
        System.out.println("Values X: "+ X " and Y:" + Y);
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 23** 考虑问题 19。假设数字的取值范围在 1～$n$。是否存在其他解决该问题的方法？

**解答：** 可以通过创建两个简单的数学公式解决这个问题。假设要寻找的两个数分别是 $X$ 和 $Y$。我们知道 $n$ 个数字的和及乘积分别是 $n(n+1)/2$ 和 $n!$。使用求和及求积公式得到两个方程，使用这两个方程获得两个未知数的值。假设数组中所有数字的总和是 $S$，乘积是 $P$，重复出现的数字是 $X$ 和 $Y$。

$$X+Y=\frac{n(n+1)}{2}-S$$

$$XY=n!/P$$

根据上述两个方程可确定 $X$ 和 $Y$。采用这种方法可能会出现加法和乘法溢出问题。

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 24** 与问题 19 相似，假设数字的取值范围在 1～$n$，并且有 $n-1$ 个元素重复出现 3 次，其余元素重复出现 2 次。查找重复出现 2 次的元素。

**解答：** 如果对数组中所有的元素及所有从 1～$n$ 的整数进行异或，那么所有出现 3 次的元素将变为 0。这是因为元素重复 3 次，再与取值范围内的整数异或将使该元素出现 4 次。因此，a XOR a XOR a XOR a=0。该方法适用于所有重复出现 3 次的元素。

根据同样的逻辑，对于重复 2 次的元素，如果对输入元素和取值范围内的整数进行异或将使元素总共出现 3 次。因此，a XOR a XOR a=a。最后可以得到所有重复 2 次的元素。

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 25** 给定一个含 $n$ 个元素的数组。在数组中查找两个元素，这两个元素的和等于给定的元素 $K$。

**蛮力法：** 该问题的一个简单解法是，对于每个输入的元素，判断是否存在和为 $K$ 的元素。可以通过使用两个简单的 for 循环解决该问题。算法的代码实现如下：

```
void BruteForceSearch[int A[], int n, int K){
        for (int i = 0; i < n; i++) {
                for(int j = i; j < n; j++) {
                        if(A[i]+A[j] == K) {
                                System.out.println("Items Found, i: " + i + " j:" + j);
                                return;
                        }
                }
        }
        System.out.println("Items not found: No such elements");
}
```

时间复杂度为 $O(n^2)$，因为有两个嵌套的 for 循环。空间复杂度为 $O(1)$。

**问题 26** 对于问题 25，能否降低其时间复杂度？

**解答：可以。** 假定已经对给定的数组排序。该操作需要 $O(n\log n)$时间。在有序数组中，保持索引 loIndex=0，hiIndex=$n-1$，并计算 $A$[loIndex]+$A$[hiIndex]。如果和等于 $K$，那么就找到了问题的解。如果和小于 $K$，将 hiIndex 减 1。如果和大于 $K$，使 loIndex 加 1。

```
void Search[int A[], int n, int K){
        int i, j, temp;
        Sort(A, n);
        for(i =  0, j = n-1; i < j) {
```

```
                temp  = A[i] + A[j];
                if(temp  == K) {
                        System.out.println("Items Found, i: " + i + " j:" + j);
                        return;
                }
                else if(temp  < K)
                        i =  i + 1;
                else    j =  j - 1;
        }
        return;
}
```

时间复杂度为 $O(n\log n)$，如果给定数组是有序的，则时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 27**　如果数组是无序的，则问题 25 的解决方案是否适用？

**解答：可以。**由于检查了所有的可能性，所以算法确保如果存在这样的一对数则一定能找到。

**问题 28**　是否存在其他解决问题 25 的方法？

**解答：存在，**使用散列表。因为我们的目标是在数组中查找两个索引号，且这两个索引号所对应的元素之和等于给定的元素 $K$。假设这两个索引号是 $X$ 和 $Y$，则有 $A[X]+A[Y]=K$。算法的目标是，对于输入数组 $A[X]$的每个元素，检查输入数组中是否存在值为 $K-A[X]$的元素。可以用散列表来简化查找过程。

- 将输入数组的每个元素都插入散列表。假设当前的元素是 $A[X]$。
- 在访问下一个元素前，判断在散列表中是否存在值等于 $K-A[X]$的元素。
- 如果存在这样的数，则说明算法可以发现这样的索引号。
- 否则，继续访问下一个元素。

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 29**　给定一个由 $n$ 个元素组成的数组 $A$。在数组 $A$ 中能否找到 3 个元素 $i$、$j$ 和 $k$，满足 $A[i]^2+A[j]^2=A[k]^2$？

**解答：**

**算法：**

- 原地排列给定数组。
- 对数组中的每一个索引号 $i$，计算 $A[i]^2$，并将其存储在数组中。
- 与问题 25 类似，从 $0\sim i-1$ 的数组中查找两个数且它们的和等于 $A[i]$。可以在 $O(n)$时间内得到结果。如果找到这样的和则返回真，否则算法继续。

```
Sort(A); // 对输入数组排序
for (int i=0; i < n; i++)
      A[i] = A[i]*A[i];
for (i=n; i > 0; i--) {
      res = false;
      if(res) {
                //问题11和问题12的解
      }
}
```

时间复杂度为排序的时间$+n\times$(查找和的时间)$=O(n\log n)+n\times O(n)=n^2$。空间复杂度为 $O(1)$。

**问题 30　查找和最接近于 0 的两个元素：**给定一个包含正数和负数的数组，查找两个元素，使得它们的和最接近于 0。例如，对于一个数组 1，60，－10，70，－80，85，算法找到的元素将是－80 和 85。

**蛮力法：**对于数组中的每个元素，计算它与其他每个元素之和并比较和的大小。最后，返回和最小的两个元素。

```
void TwoElementsWithMinSum(int A[], int n){
        int i, j, min_sum, sum, min_i, min_j, inv_count = 0;
        if(n < 2) {
        System.out.println("Invalid Input");
        return;
}
/* 初始化数值*/
min_i = 0;
min_j = 1;
min_sum = A[0] + A[1];
for(i= 0; i < n - 1; i ++)    {
        for(j = i + 1; j < n; j++)    {
                sum = A[i] + A[j];
                if(Math.abs(min_sum) > Math.abs(sum)) {
                        min_sum = sum;
                        min_i = i;
                        min_j = j;
                }
        }
}
        System.out.println(" The two elements are “ + arr[min_i] + “ and " + arr[min_j]);
}
```

时间复杂度为 $O(n^2)$。空间复杂度为 $O(1)$。

**问题 31　**是否能降低问题 30 的时间复杂度？

**解答：**可以使用排序。

**算法：**

1）对给定输入数组的所有元素排序。

2）使用两个索引，一个在开始位置($i=0$)，另一个在结束位置($j=n-1$)。另外，使用两个变量来分别记录最接近 0 的最小正数和以及最小负数和。

3）当 $i<j$ 时，

a. 如果当前元素对的和大于 0 且小于 positiveClosest，则更新 positiveClosest 的值。$j$ 的值减 1。

b. 如果当前元素对的和小于 0 且大于 negativeClosest，则更新 negativeClosest 的值。$i$ 的值增 1。

c. 否则，输出当前元素对。

```
void TwoElementsWithMinSum(int A[], int n) {
        int i = 0, j = n-1, temp, postiveClosest = INT_MAX, negativeClosest = INT_MIN;
        Sort(A, n);
        while(i < j) {
                temp  = A[i] + A[j];
                if(temp  > 0) {
                        if (temp < postiveClosest)
                                postiveClosest = temp;
```

```
                j--;
            }
            else if (temp  < 0) {
                if (temp > negativeClosest)
                    negativeClosest = temp;
                i++;
            }
            else printf("Closest Sum: %d ", A[i] + A[j]);
        }
        return (Math.abs(negativeClosest)> postiveClosest: postiveClosest: negativeClosest);
}
```

时间复杂度为 $O(n\log n)$，由于需要排序。空间复杂度为 $O(1)$。

**问题 32**　给定由 $n$ 个元素组成的数组。在数组中查找 3 个元素，使得它们的和等于 $K$。

**蛮力法：**默认的解决方案是，对于每一对输入的元素，判断是否有另一个元素使得三者之和是 $K$。可以通过使用 3 个简单的 for 循环来解决。算法的实现代码如下：

```
void BruteForceSearch[int A[], int n, int data){
        for (int i = 0; i < n; i++) {
            for(int j = i+1; j < n; j++) {
                for(int k = j+1; k < n; k++)        {
                    if(A[i] + A[j] + A[k]== data) {
                        System.out.println("Items Found, i:"+ i+" j:"+j+" k:"+k);
                        return;
                    }
                }
            }
        System.out.println("Items not found: No such elements");
}
```

时间复杂度为 $O(n^3)$，由于有 3 个嵌套的 for 循环。空间复杂度为 $O(1)$。

**问题 33**　假如数组是无序的，问题 32 的解决方案是否还适用？

**解答：可以。**由于检查了所有的可能性，所以该算法保证如果存在 3 个数的和是 $K$，则一定能够找到。

**问题 34**　可以采用排序的方法来解决问题 32 吗？

**解答：可以。**

```
void Search[int A[], int n, int data){
        Sort(A, n);
        for(int k = 0; k < n; k++) {
            for(int i =  k + 1, j = n-1; i < j;  ) {
                if(A[k] + A[i] + A[j]  == data) {
                    System.out.println("Items Found, i:" + i + " j:" + j + " k:" + k);
                    return;
                }
                else if(A[k] + A[i] + A[j]  < data)
                    i =  i + 1;
                else      j =  j - 1;
            }
        }
        return;
}
```

时间复杂度为排序时间＋在排序列表中的查找时间＝$O(n\log n)+O(n^2)\approx O(n^2)$，因为有两个嵌套的 for 循环。空间复杂度为 $O(1)$。

**问题 35** 能否采用散列方法来解决问题 32？

**解答：可以。**因为我们的目标是发现数组的 3 个索引，使得对应元素之和等于 $K$。如果这 3 个索引号是 $X$、$Y$ 和 $Z$，则有 $A[X]+A[Y]+A[Z]=K$。

假设已经把所有可能的每两个元素的和存储到散列表中，即散列表的键值是 $K-A[X]$，$K-A[X]$所对应的值是和为 $K-A[X]$的所有可能的元素对。

**算法：**

- 在开始搜索前，将所有可能的和值及对应的元素对插入散列表。
- 对于输入数组的每个元素，将其插入散列表。假设当前的元素是 $A[X]$。
- 检查在散列表中是否存在一项，它的键值是 $K-A[X]$。
- 如果存在这样的元素，则扫描 $K-A[X]$对应的元素对，然后返回包括 $A[X]$在内的所有可能的元素对。
- 如果不存在这样的元素(以 $K-A[X]$为键值)，则跳转到下一个元素。

时间复杂度为将所有可能的元素对存储在散列表的时间＋查找时间＝$O(n^2)+O(n^2)\approx O(n^2)$。空间复杂度为 $O(n)$。

**问题 36** 给定由 $n$ 个整数组成的数组，3 和问题就是查找 3 个整数，且它们的和最接近于 0。

**解答：**与问题 32 类似，只不过 $K$ 值是 0。

**问题 37** 设 $A$ 是一个由 $n$ 个不同的整数组成的数组。假设 $A$ 具有以下属性：存在一个索引 $1\leqslant k\leqslant n$，使得 $A[1]$，…，$A[k]$是一个递增序列，而 $A[k+1]$，…，$A[n]$是递减序列。设计和分析一种有效的算法来查找 $k$。

**类似的问题：**假设给定的数组是有序的，但从负数开始，正数结束(这样的函数叫单调递增函数)。在这个数组中找到正数的起始索引。假设输入数组的长度已知，设计一个时间复杂度是 $O(n\log n)$的算法。

**解答：**可使用二分查找的一种衍生算法。

```
int Search (int A[], int n, int first, int last){
        int mid, first = 0, last = n-1;
        while(first <= last) {
                //若当前数组长度为1
                if(first == last)
                        return A[first];
                //若当前数组长度为2
                else if(first == last-1)
                        return max(A[first], A[last]);
                //若当前数组长度大于等于3
                else {
                        mid = first + (last-first)/2;
                        if(A[mid-1] < A[mid] && A[mid] > A[mid+1])
                                return A[mid];
                        else if(A[mid-1] < A[mid] && A[mid] < A[mid+1])
                                first = mid+1;
                        else if(A[mid-1] > A[mid] && A[mid] > A[mid+1])
                                last = mid-1;
                        else    return INT_MIN ;
                } //else结束
        } // while结束
}
```

递归方程是 $T(n)=2T(n/2)+c$。使用主定理可得，时间复杂度为 $O(\log n)$。

**问题 38**　如果事先不确定 $n$ 的值，应该如何解决问题 37？

**解答：**重复计算 $A[1]$、$A[2]$、$A[4]$、$A[8]$、$A[16]$等，直到找到一个值 $n$，使得 $A[n]>0$。

时间复杂度为 $O(n\log n)$，因为算法是以 2 倍的速度移动。详情请参见第 1 章。

**问题 39**　给定一个大小未知的输入数组，该数组中所有的 1 在首部，0 在尾部。在数组中查找 0 首次出现的索引。假设在数组中有成千上万个 1 和 0。例如，数组为

1111111…1100000…0000000

**解答：**这个问题几乎与问题 38 类似。以 $2^K$ 速率检查位，这里 $K=0$，1，2，…，由于算法的移动速率是 2，所以时间复杂度是 $O(\log n)$。

**问题 40**　给定一个由 $n$ 个元素组成的有序整数数组，该数组中的元素已经经过多次转换(次数未知)，给出一个时间复杂度为 $O(\log n)$的算法，查找数组中指定的某个元素。

**例子：**在数组(15 16 19 20 25 1 3 4 5 7 10 14)中寻找 5。

**输出：**8(5 在数组中的索引)。

**解答：**假设给定的数组是 $A[]$，使用问题 37 的扩展方法解决该问题。下面的函数 FindPivot 的返回值是 $k$(假设函数的返回值是对应的索引)。找到枢轴点，把数组分成两个子数组并调用二分查找。发现枢轴点的主要思想是：对于一个有序(按递增顺序)的数组，枢轴元素是唯一一个其下一个元素比它小的元素。采用上述标准和二分查找方法可以在$O(\log n)$内找到枢轴元素。

**算法：**

1）查找枢轴元素，将数组分成两个子数组。

2）对两个子数组中的任意一个调用二分查找。

a. 如果元素比第一个元素大，则在左边的子数组中查找。

b. 否则在右边的子数组中查找。

3）如果元素能够在所选择的子数组中查找到，则返回其对应的索引，否则返回－1。

```
int FindPivot(int A[], int start, int finish) {
        if(finish - start == 0)
                return start;
        else if(start == finish - 1) {
                if(A[start] >= A[finish])
                        return start;
                else    return finish;
        }
        else {  mid = start + (finish-start)/2;
                if(A[start] >= A[mid])
                        return FindPivot(A, start, mid);
                else    return FindPivot(A, mid, finish);
        }
}
int Search(int A[], int n, int x) {
        int pivot = FindPivot(A, 0, n-1);
        if(A[pivot] == x) return pivot;
        if(A[pivot] <= x)
                return BinarySearch(A, 0, pivot-1, x);
        else    return BinarySearch(A, pivot+1, n-1, x);
}
```

```
int BinarySearch(int A[], int low, int high, int x) {
        if(high >= low)   {
                int mid = low + (high - low)/2;
                if(x == A[mid])
                        return mid;
                if(x > A[mid])
                        return BinarySearch(A, (mid + 1), high, x);
                else    return BinarySearch(A, low, (mid -1), x);
        }
        /*当没有找到对应元素时返回-1*/
        return-1;
}
```

时间复杂度为 $O(\log n)$。

**问题 41** 对于问题 40，能否只通过一次扫描解决问题？

**解答：可以。**

```
int BinarySearchRotated(int A[], int start, int finish, int data) {
        if(start > finish) return -1;
        int mid = start + (finish - start) / 2;
        if(data == A[mid]) return mid;
        else if(A[start] <= A[mid]) {          // start部分是有序的
                if(data >= A[start] && data < A[mid])
                        return BinarySearchRotated(A, start, mid - 1, data);
                else    return BinarySearchRotated(A, mid + 1, finish, data);
        }
        else {  // A[mid] <= A[finish], finish部分是有序的
                if(data > A[mid] && data <= A[finish])
                        return BinarySearchRotated(A, mid + 1, finish, data);
                else    return BinarySearchRotated(A, start, mid - 1, data);
        }
}
```

时间复杂度为 $O(\log n)$。

**问题 42 双调查找：**如果一个数组是由递增顺序的整数序列组成的，且紧跟一个递减顺序的整数序列，则该数组称为双调数组。给定一个由 $n$ 个不同整数组成的双调数组 $A$，描述如何在 $O(\log n)$步确定给定的整数是否在数组中。

**解答：**解决方法与问题 37 的解类似。

**问题 43** 问题 37 的其他描述方式：设 $A[]$是一个数组，该数组刚开始递增，达到最大值后递减。设计一个时间复杂度为 $O(\log n)$的算法来寻找最大值的索引。

**问题 44** 给出一个时间复杂度为 $O(n\log n)$的算法，计算一个长度为 $n$ 的整数序列的中位数。

**解答：**排序，然后返回在$\frac{n}{2}$处的元素。

**问题 45** 给定两个大小分别为 $m$ 和 $n$ 的有序序列，在 $O(\log(m+n))$时间内查找所有元素的中位数。

**解答：**请参见第 18 章。

**问题 46** 给定一个由 $n$ 个元素组成的有序数组 $A$，数组中的元素可能存在重复，在 $O(\log n)$时间内查找某个元素首次出现的索引。

**解答：**要想找到某个元素首次出现的索引，需要按照下面的条件检查。如果下面任意一种情况为真，则返回相应的位置。

```
                mid == low && A[mid] == data  ||  A[mid] == data && A[mid-1] < data
int BinarySearchFirstOccurrence(int A[], int low, int high, int data) {
        if(high >= low) {
                int mid = low + (high-low) / 2;
                if((mid == low && A[mid] == data) || (A[mid] == data && A[mid - 1] < data))
                        return mid;
                //优先考虑数据的左半部分
                else if(A[mid] >= data)
                        return BinarySearchFirstOccurrence (A, low, mid - 1, data);
                else    return BinarySearchFirstOccurrence (A, mid + 1, high, data);
        }
        return -1;
}
```

时间复杂度为 $O(\log n)$。

**问题 47**　给定由 $n$ 个元素组成的有序数组 $A$，在 $O(\log n)$时间内查找某个元素最后出现的索引。

**解答：**要想找到某个元素最后出现的索引，需要按照下面的条件检查。如果下面任意一种情况为真，则返回相应的位置。

```
        mid == high && A[mid] == data  ||  A[mid] == data && A[mid+1] > data
int BinarySearchLastOccurrence(int A[], int low, int high, int data) {
        if(high >= low) {
                int mid = low + (high-low) / 2;
                if((mid == high && A[mid] == data) || (A[mid] == data && A[mid + 1] > data))
                        return mid;
                //优先考虑数组的左半部分
                else if(A[mid] <= data)
                        return BinarySearchLastOccurrence (A, mid + 1, high, data);
                else    return BinarySearchLastOccurrence (A, low, mod - 1, data);
        }
        return -1;
}
```

时间复杂度为 $O(\log n)$。

**问题 48**　给定一个由 $n$ 个元素组成的有序数组，数组中的元素可能存在重复，查找某个元素出现的次数。

**蛮力法：**对数组执行线性查找，当在数组中发现该元素时，计数值增加 1。

```
int LinearSearchCount(int A[], int n, int data) {
        int count = 0;
        for (int i = 0; i < n; i++)  {
                if(A[i] == k)
                        count++;
        }
        return count;
}
```

时间复杂度为 $O(n)$。

**问题 49**　能否降低问题 48 的时间复杂度？

**解答：可以。**首先利用二分查找及另一个小的扫描来解决这个问题。

**算法：**

- 在数组中，对于要查找的数据执行二分查找。假设它的位置为 $K$。

- 从 $K$ 开始向左边查找，统计要查找数据出现的次数，用 leftCount 来计数。
- 同样，从 $K$ 开始向右边查找，统计要查找数据出现的次数，用 rightCount 来计数。
- 出现的总次数＝leftCount＋1＋rightCount。

时间复杂度为 $O(\log n+S)$，这里 $S$ 是数据出现的总次数。

**问题 50** 是否有其他方法可以解决问题 48?

**解答：**

**算法：**

- 寻找数据首次出现的位置，设索引为 firstOccurrence(参考问题 46 的算法)。
- 寻找数据末次出现的位置，设索引为 lastOccurrence(参考问题 47 的算法)。
- 返回 lastOccurence－firstOccurence＋1。

时间复杂度为 $O(\log n+\log n)=O(\log n)$。

**问题 51** 序列 1，11，21 中的下一个数字是什么？为什么?

**解答：**请用英文大声阅读给定的数字。该问题只是为了有趣。

一 一

二 一

一二，一一→1211

所以，答案就是下一个数字是前一个数字的阅读表示。

**问题 52** 高效地查找第二个最小的数字。

**解答：**对于给定的元素，可以构建一个堆，最多使用 $n-1$ 次比较(算法请参见第 7 章)。对于 GetMax()操作，使用 log$n$ 次比较，可以找出第二个最小的数字。总的来说，需要 $n+\log n+$constant 次比较得到要查找的数字。

**问题 53** 还有其他解决方案能解决问题 52 吗?

**解答：**另一种可行的方式是，把 $n$ 个数字分成两组，连续执行 $n/2$ 次比较，使用锦标赛方法就可以找到最大值。第一轮将在 $n-1$ 次比较后找到最大值。第二轮将对第一轮的获胜者和出栈的最大值进行比较。这将产生 log$n-1$ 次比较，比较的总次数为 $n+\log n-2$。上面的解决方案叫作锦标赛问题。

**问题 54** 如果一个元素的出现次数超过 $n/2$ 以上，则它是主元素。给出一个算法，它包含 $n$ 个元素的数组作为参数，识别主元素(如果它存在)。

**解答：**基本解决方案是有两个循环并记录所有不同元素的最大计数。如果最大计数大于 $n/2$，则跳出循环，并返回具有最大计数值的元素。如果最大计数不超过 $n/2$，则主元素不存在。

时间复杂度为 $O(n^2)$。空间复杂度为 $O(1)$。

**问题 55** 是否可以将问题 54 的时间复杂度降至 $O(n\log n)$?

**解答：**采用二分查找可以实现。二叉搜索树的结点(用于本方法)描述如下。

```
public class TreeNode {
        public int element;
        public int count;
        public TreeNode left;
        public TreeNode right;
        ……
}
```

在 BST 中依次插入元素，如果一个元素已经存在，则该结点的计数值加 1。在任何

阶段，如果一个结点的计数值超过 $n/2$，则返回。该方法适用于多数元素的出现次数为 $n/2+1$ 且均位于数组的起始位置的情况，例如{1，1，1，1，1，2，3，4}。

时间复杂度：如果使用二叉搜索树，那么最坏情况下的时间复杂度为 $O(n^2)$，如果使用平衡二叉搜索树，则为 $O(n\log n)$。空间复杂度为 $O(n)$。

**问题 56** 是否还有其他时间复杂度为 $O(n\log n)$且同样能解决问题 54 的方法？

**解答：** 对输入数组排序并扫描已排序数组来查找多数元素。

时间复杂度为 $O(n\log n)$。空间复杂度为 $O(1)$。

**问题 57** 能否降低问题 54 的时间复杂度？

**解答：** 如果某个元素在 $A$ 中的出现次数超过 $n/2$，那么它一定是 $A$ 的中位数。但相反时是不正确的，所以一旦发现中位数，必须查看它在 $A$ 中出现了多少次。可以使用线性选择方法，该方法需要执行 $O(n)$次(相关算法可以参见第 12 章)。

```
int CheckMajority(int A[], in n) {
        1)  Use linear selection to find the median m of A.
        2)  Do one more pass through A and count the number of occurrences of m.
              a.  If m occurs more than n/2 times then return true;
              b.  Otherwise return false.
}
```

**问题 58** 是否还有其他方法可以解决问题 54？

**解答：** 因为只有一个元素重复，所以可以对输入数组进行简单扫描，并记录元素的出现次数。如果计数是 0，则可以判定元素是第一次出现，否则将是结果元素。

```
int MajorityNum(int[] A, int n) {
    int majNum, count = 0, element = -1;
    for(int i = 0; i < n; i++) {
            // 如果当前计数为0则设置当前候选为majNum并设置计数为1
            if(count == 0) {
                element = A[i];
                count = 1;
            }
            else if(element == A[i]) {
                //若计数不是0且element与当前候选相等,则计数加1
                count++;
            }
            else {    //若计数不是0且element不同于当前候选,则计数减1
                count--;
            }
    }
    return element;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 59** 给定一个由 $2n$ 个元素组成的数组，其中 $n$ 个元素是相同的，剩余的 $n$ 个元素互不相同。找出主元素。

**解答：** 重复出现的元素占了数组的一半。不管数组的排列如何，只有下面的一种情况是真的。

- 所有重复元素的相对距离为 2。

  **例子：** $n$，1，$n$，100，$n$，54，$n$，…
- 至少有两个重复的元素是相邻的。

**例子：** $n$，$n$，1，100，$n$，54，$n$，…

$n$，1，$n$，$n$，$n$，54，100，…

1，100，54，$n$，$n$，$n$，$n$，…

在最坏的情况下，需要对数组进行两次遍历。

- 第一次：比较 $A[i]$和 $A[i+1]$
- 第二次：比较 $A[i]$和 $A[i+2]$

匹配的元素将是要查找的元素。时间开销和空间开销分别为 $O(n)$和 $O(1)$。

**问题 60** 给定一个由 $2n+1$ 个整数元素组成的数组，$n$ 个元素在数组中的任意地方出现两次，一个整数在数组中的某处仅出现一次。使用 $O(n)$次操作和 $O(1)$个额外的内存寻找这个整数。

**解答：** 除了一个元素外，所有其他元素都是重复的。我们知道 $A$ XOR $A=0$。基于此，如果异或所有的输入元素，则可以得到剩下的元素。

```
int Solution(int A[], int n) {
        int i, res;
        for (i = res = 0; i < 2n+1; i++)
                res = res ^ A[i];
         return res;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 61 从 $n$ 层建筑物扔鸡蛋问题：** 假设有一个 $n$ 层建筑物和一些鸡蛋，当一个鸡蛋从第 $F$ 层或更高层抛出时，鸡蛋会被打破，否则不会被打破。当打破 $O(\log n)$个鸡蛋时，设计一个策略来确定楼层 $F$ 的值。

**解答：** 参见第 18 章。

**问题 62 数组的局部最小值：** 给定一个由 $n$ 个不同整数组成的整数数组，设计一个时间复杂度为 $O(\log n)$的算法来查找局部最小值：对某个索引 $i$，使得 $A[i-1]<A[i]<A[i+1]$。

**解答：** 检查中位数 $A[n/2]$，以及两个邻居的值 $A[n/2-1]$和 $A[n/2+1]$。如果 $A[n/2]$是局部最小值，算法停止；否则在值较小邻居所处的一半进行查找。

**问题 63** 给定一个 $n\times n$ 数组的元素，每行是升序排序，每列也是升序排序，设计一个时间复杂度为 $O(n)$的算法来确定数组中是否存在某个给定的元素 $x$。可以假设数组中所有的元素是不同的。

**解答：** 假设给定的矩阵为 $A[n][n]$。从最后一行第一列开始(或第一行最后一列)，如果要查找的元素比位于 $A[1][n]$的元素值大，则第一列可以忽略。如果要查找的元素小于位于 $A[1][n]$的元素，则最后一行可以忽略。一旦第一列或最后一行被忽略，则从剩余数组的左下部重新开始这个过程。在该算法中，将最多有 $n$ 个元素与所查找的元素进行比较。

时间复杂度为 $O(n)$，这是因为最多遍历 $2n$ 个点。空间复杂度为 $O(1)$。

**问题 64** 给定一个由 $n^2$ 个元素组成的 $n\times n$ 数组，给出一个时间复杂度为 $O(n)$的算法来查找一对索引 $i$ 和 $j$，使得 $A[i][j]<A[i+1][j]$，$A[i][j]<A[i][j+1]$，$A[i][j]<A[i-1][j]$和 $A[i][j]<A[i][j-1]$。

**解答：** 这个问题与问题 63 相同。

**问题 65**　给定一个 $n \times n$ 矩阵，在每行中所有的 1 后面都是 0。查找具有最多 0 的行。

**解答：**从第一行的最后一列开始。如果元素是 0，那么移动到同一行的前一列，同时计数加 1，表示 0 出现的最大次数。如果元素是 1，则移动到同一列的下一行。重复这个过程，直到算法到达最后一行的第一列为止。

时间复杂度为 $O(2n) \approx O(n)$（与问题 63 类似）。

**问题 66**　给定一个大小未知的输入数组，其中所有的数字在前面，特殊符号在后面。查找这些特殊符号在数组中首次出现的索引。

**解答：**参见第 18 章。

**问题 67　将偶数与奇数分开：**给定一个数组 A[]，写出一个函数，使其能够将偶数与奇数分开。该函数把所有偶数放在前面，奇数放在后面。

**例子：**输入＝{12，34，45，9，8，90，3}，输出＝{12，34，90，8，9，45，3}。

**注意：**在输出时，数字的顺序可以改变，也就是说 34 可以出现在 12 前面，3 也可以出现在 9 前面。

**解答：**这个问题与把数组中的 0 和 1 分开是非常相似的（问题 68），这两个问题都是著名的荷兰国旗问题的变体。

**算法：**逻辑思路与快速排序算法类似：

1）初始化两个索引变量 left 和 right：left＝0，right＝$n-1$。

2）持续递增左边的索引变量，直到发现一个奇数。

3）持续递减右边的索引变量，直到发现一个偶数。

4）如果 left＜right，则互换 A[left]和 A[right]。

```
void DutchNationalFlag(int A[], int n) {
        int left = 0, right = n-1;                          /*初始化左、右索引*/
        while(left < right) {
                /*当在左边遇到0时左索引加1*/
                while(A[left]%2 == 0 && left < right)
                        left++;
                /*当在右边遇到1时右索引减1*/
                while(A[right]%2 == 1 && left < right)
                        right--;
                if(left < right) {
                        /*交换A[left]和A[right]*/
                        swap(&A[left], &A[right]);
                        left++;
                        right--;
                }
        }
}
```

时间复杂度为 $O(n)$。

**问题 68**　下面是问题 67 的另一种描述形式，但几乎没有差别。

**把数组中的 0 和 1 分开：**已知一个由 0 和 1 按照随机顺序排列的数组。把 0 排在数组的左侧，把 1 排在数组的右侧。数组仅遍历一次。

输入数组＝[0，1，0，1，0，0，1，1，1，0]

输出数组＝[0，0，0，0，0，1，1，1，1，1]

**解答：**对 0 或 1 计数。

1）统计 0 出现的次数。让计数值为 $C$。

2）一旦统计结束，把 $C$ 个 0 放在数组的开始部分，把 1 放在剩余的 $n-C$ 个位置。

时间复杂度为 $O(n)$，该方案扫描数组两遍。

**问题 69** 能否通过一次扫描来解决问题 68？

**解答：可以。**在遍历时使用两个索引：维护这两个索引。初始化第一个索引 left 为 0，第二个索引 right 为 $n-1$。当 left＜right 时，执行下列操作：

1）当有 0 存在时，使索引 left 递增。

2）当有 1 存在时，使索引 right 递减。

3）如果 left＜right，交换 $A$[left]和 $A$[right]。

```
/*将0和1分别放在左边和右边的函数*/
void Separate0and1(int A[], int n) {
        /*初始化左、右索引*/
        int left = 0, right = n-1;
        while(left < right) {
                /*在左边遇到0时计数加1*/
                while(A[left] == 0 && left < right)
                        left++;
                /*在右边遇到1时计数减1*/
                while(A[right] == 1 && left < right)
                        right--;
                /* 若left索引对应的元素小于right索引对应的元素，则说明左边有一个1,
                   右边有一个0,交换A[left]和A[right] */
                if(left < right) {
                        A[left] = 0;
                        left++;
                        right--;
                }
        }
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 70 对由 0、1 和 2(或 R、G、B)组成的数组排序：**给定一个由 0、1 和 2 组成的数组 $A$[]，给出一种算法对 $A$[]排序。算法应首先把所有 0 放在前面，然后是 1，最后把所有的 2 放在后面。

**例子：**输入＝{0，1，1，0，1，2，1，2，0，0，0，1}，输出＝{0，0，0，0，0，1，1，1，1，1，2，2}

**解答：**

```
void Sorting012sDutchFlagProblem(int A[],int n){
        int low=0,mid=0,high=n-1;
        while(mid <=high){
                switch(A[mid]){
                        case 0:
                                swap(A[low],A[mid]);
                                low++;mid++;
                                break;
                        case 1:
                                mid++;
                                break;
                        case 2:
                                swap(A[mid],A[high]);
                                high--;
```

```
                    break;
                }
            }
        }
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 71　两个元素的最大差：**给定一个整数数组 $A[]$，找出任何两个元素之间的最大差，要求数组 $A[]$中较大的元素出现在较小数的后面。

**例子：**如果数组为[2，3，10，6，4，8，1]，那么返回值应该是 8(10 与 2 之间的差)。如果数组为[7，9，5，6，3，2]，那么返回值应该是 2(7 与 9 之间的差)。

**解答：**参见第 18 章。

**问题 72**　给定一个由 101 个元素组成的数组。其中 25 个元素重复 2 次，12 个元素重复 4 次，一个元素重复 3 次。要求在 $O(1)$时间内找到重复出现 3 次的元素。

**解答：**在解决这个问题前，考虑 XOR 操作的属性：a XOR a=0，即如果应用相同的元素进行 XOR，则结果为 0。

**算法：**

- 对给定数组的所有元素进行 XOR，假设结果为 $A$。
- 执行完该操作后，出现 3 次的数中 2 次出现变成 0，1 次出现保持不变。
- 出现 4 次的 12 个元素变成 0。
- 出现 2 次的 25 个元素变成 0。
- 因此，对所有元素执行 XOR 操作后，可得到结果。

时间复杂度为 $O(n)$，因为仅执行一遍扫描。空间复杂度为 $O(1)$。

**问题 73**　给定一个数字 $n$，给出一个算法来寻找 $n!$ 中尾数 0 的个数。

**解答：**

```
int NumberOfTrailingZerosInNumber(int n) {
        int i, count = 0;
        if(n < 0)
                return -1;
        for (i = 5; n / i > 0; i *= 5)
                count += n / i;
        return count;
}
```

时间复杂度为 $O(\log n)$。

**问题 74**　给定一个由 $2n$ 个整数组成的数组，格式为 $a1\ a2\ a3 \ldots an\ b1\ b2\ b3 \ldots bn$。在没有额外内存的情况下，把该数组排列成 $a1\ b1\ a2\ b2\ a3\ b3 \ldots an\ bn$。

**解答：**蛮力法涉及两个嵌套循环，它将数组后半部分的元素轮换到左边。第一个循环运行 $n$ 次，涉及数组后半部分的所有元素。第二个循环轮换左边的元素。注意，第二个循环的开始索引取决于要轮换的元素，最后的索引取决于需要移到左边多少个位置。

```
void ShuffleArray() {
        int n = 4, A[] = {1,3,5,7,2,4,6,8};
        for (int i = 0, q =1, k = n; i < n; i++, k++, q++) {
                for (int j = k; j > i + q; j--) {
                        int tmp = A[j-1];
```

```
                    A[j-1] = A[j];
                    A[j] = tmp;
                }
            }
            for (int i = 0; i  < 2*n; i++)
                System.out.println(A[i]);
}
```

时间复杂度为 $O(n^2)$。

**问题 75** 能否改善问题 74 的解决方案？

**解答：** 参见第 18 章。采用分治技术可以实现一个时间复杂度为 $O(n\log n)$的更好的解决方案。考虑如下所示的例子；

1）开始数组为：$a1\ a2\ a3\ a4\ b1\ b2\ b3\ b4$。

2）把数组分成两部分：$a1\ a2\ a3\ a4$：$b1\ b2\ b3\ b4$。

3）围绕中心交换元素：用 $b1\ b2$ 交换 $a3\ a4$，得到 $a1\ a2\ b1\ b2\ a3\ a4\ b3\ b4$。

4）将 $a1\ a2\ b1\ b2$ 分成 $a1\ a2$：$b1\ b2$，然后把 $a3\ a4\ b3\ b4$ 分成 $a3\ a4$：$b3\ b4$。

5）对每个子数组围绕中心交换元素，得到 $a1\ b1\ a2\ b2\ a3\ b3\ a4\ b4$。

注意，这个解决方案仅适用于 $n=2^i(i=0, 1, 2, 3$ 等)的情况。在该例中，$n=2^2=4$，因此很容易递归地把数组分成两半。在调用递归函数前，围绕中心交换元素的基本思想是产生较小规模的问题。如果都是一些具有特定性质的元素，则采用线性时间复杂度的解决方案就可以实现。例如，可以使用元素本身的值来计算元素的新位置。这只不过是一个散列技术。

**问题 76** 给定一个数组 $A[]$，查找最大的 $j-i$ 使得 $A[j]>A[i]$。

**例子：** 输入$=\{34, 8, 10, 3, 2, 80, 30, 33, 1\}$，输出$=6(j=7, i=1)$。

**解答：蛮力法。**运行两个循环。在外层循环中，依次从左边选择元素。在内层循环中，将选择的元素与右边开始的元素进行比较。当发现某个元素大于所选择元素时，停止内层循环，并持续更新最大的 $j-i$。

```
int maxIndexDiff(int A[], int n){
    int maxDiff = -1;
    int i, j;
    for (i = 0; i < n; ++i){
        for (j = n-1; j > i; --j){
            if(A[j] > A[i] && maxDiff < (j - i))
                maxDiff = j - i;
        }
    }
    return maxDiff;
}
```

时间复杂度为 $O(n^2)$。空间复杂度为 $O(1)$。

**问题 77** 能否降低问题 74 的复杂度？

**解答：** 为了解决这个问题，需要获取 $A[]$的两个最佳索引：左索引 $i$ 和右索引 $j$。对于某个元素 $A[i]$，如果有一个元素位于 $A[i]$的左边，且其值比 $A[i]$小，则不需要考虑 $A[i]$作为左索引。类似地，如果有一个更大的元素位于 $A[j]$的右边，那么不需要考虑这个 $j$ 作为右索引。

所以构建两个辅助数组 LeftMins[]和 RightMaxs[]，这样 LeftMins[]拥有 $A[i]$左边

最小的元素且包括 $A[i]$，RightMaxs[]拥有 $A[j]$右边最大的元素且包括 $A[j]$。构建这两个辅助数组后，从左到右遍历这两个数组。

当遍历 LeftMins[]和 RightMaxs[]时，如果发现 LeftMins[$i$]大于 RightMaxs[$j$]，则必须在 LeftMins[]中向前移动(或执行 $i++$)，因为 LeftMins[$i$]左边的所有元素都大于或等于 LeftMins[$i$]。否则必须在 RightMaxs[$j$]中向前移动来寻找一个更大的 $j-i$ 值。

```
int maxIndexDiff(int A[], int n){
    int maxDiff;
    int i, j;
    int *LeftMins = (int *)malloc(sizeof(int)*n);
    int *RightMaxs = (int *)malloc(sizeof(int)*n);
    LeftMins[0] = A[0];
    for (i = 1; i < n; ++i)
        LeftMins[i] = min(A[i], LeftMins[i-1]);
    RightMaxs[n-1] = A[n-1];
    for (j = n-2; j >= 0; --j)
        RightMaxs[j] = max(A[j], RightMaxs[j+1]);
    i = 0, j = 0, maxDiff = -1;
    while (j < n && i < n){
        if (LeftMins[i] < RightMaxs[j]){
            maxDiff = max(maxDiff, j-i);
            j = j + 1;
        }
        else
            i = i+1;
    }
    return maxDiff;
}
```

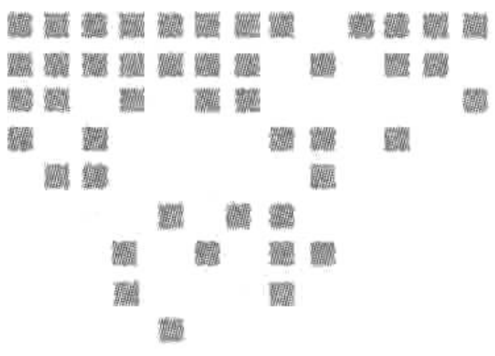
时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 78**　给定一个数组，如何检查该数组是否是成对有序的？如果数组中每一个连续的数字对是有序的(非递减)，则认为该数组是成对有序的。

**解答：**

```
public boolean isPairwiseSorted(int A[], int n) {
  if (n == 0 || n == 1)
    return true;
  for (int i = 0; i < n - 1; i += 2){
    if (A[i] > A[i+1])
      return false;
  }
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。



Chapter 12 第 12 章

# 选择算法（中位数）

## 12.1 什么是选择算法

选择算法是在某个列表中寻找第 $k$ 个最小/最大数字(也称为第 $k$ 个顺序统计量)的算法。这包括查找最小值、最大值和中位数。对于查找第 $k$ 个顺序统计量，有多种不同复杂度的解决方案，本章将列举所有这些可能的解决方案。

## 12.2 基于排序的选择算法

选择问题可以转换为排序问题。此类方法首先对输入元素排序，然后得到所需要的元素。如果要选择多个元素，这种方法是高效的。例如，假设要获取最小元素。对输入元素排序后，仅返回第一个元素即可(假设数组以升序排序)。现在，假设要获取第二小的元素，可以简单地从排序列表中返回第二个元素即可。这意味着为了寻找第二小的元素，不需要再次进行排序。对于后续查找也是一样。即使想要查找第 $k$ 小的元素，只需要对有序列表扫描一次就能找到该元素(若该元素存储在数组中，则只需要返回第 $k$ 个索引对应的元素值)。

从上述讨论可知，在进行初始排序后，只需要通过一次扫描就可以得到查询结果，其时间复杂度为 $O(n)$。通常情况下，此方法需要 $O(n\log n)$时间(用来排序)，其中 $n$ 是输入列表的长度。若执行 $n$ 次查询，则每次操作的平均开销是 $\frac{n\log n}{n}\approx O(\log n)$。这种分析称为平摊分析。

## 12.3 基于划分的选择算法

请参考本章问题 6 所用的算法。该算法类似于快速排序。

## 12.4　线性选择算法——中位数的中位数算法

| | |
|---|---|
| 最坏情况下性能 | $O(n)$ |
| 最好情况下性能 | $O(n)$ |
| 最坏情况下空间复杂度 | $O(1)$　辅助 |

请参考本章问题 11。

## 12.5　按照排序顺序查找 $K$ 个最小元素

请参考本章问题 16 所用的算法。

## 12.6　选择算法的相关问题

**问题 1**　在长度为 $n$ 的数组 $A$ 中查找最大的元素。

**解答：**扫描整个数组并返回最大的元素。

```
void FindLargestInArray(int n, int[] A) {
        int large = A[0];
        for (int i = 1; i <= n-1; i++)
            if(A[i] > large)
                large = A[i];
        System.out.println("Largest: " + large);
}
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

**注意：**任何通过键值比较来查找 $n$ 个键值中最大值的确定性算法都至少需要 $n-1$ 次比较。

**问题 2**　在长度为 $n$ 的数组 $A$ 中查找最小元素和最大元素。

**解答：**

```
void FindSmallestAndLargestInArray (int[] A, int n) {
        int small = A[0];
        int large =A[0];
        for(int i = 1; i <= n-1; i++)
            if(A[i] < small)
                small = A[i];
            else if(A[i] > large)
                large = A[i];
        System.out.println("Smallest: " + small + " Largest: " + large);
}
```

时间复杂度为 $O(n)$，最坏情况下的比较次数为 $2(n-1)$；空间复杂度为 $O(1)$。

**问题 3**　是否能对上述算法进行改进？

**解答：**可以通过两两比较来改进算法。

```
void FindWithPairComparison (int A[], int n) {//n为偶数。两两比较
        int large = small = -1;
        for (int i = 0; i <= n - 1; i = i + 2) {          //步长2
            if(A[i] < A[i + 1]) {
                if(A[i] < small)
                    small = A[i];
```

```
                    if(A[i + 1] > large)
                        large = A[i + 1];
                }
                else {
                    if(A[i + 1] < small)
                        small = A[i + 1];
                    if(A[i] > large)
                        large = A[i];
                }
        }
        System.out.println("Smallest: " + small + " Largest: " + large);
    }
```

时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

$$\text{比较次数：}\begin{cases}\dfrac{3n}{2}-2 & n\text{ 是偶数}\\ \dfrac{3n}{2}-\dfrac{3}{2} & n\text{ 是奇数}\end{cases}$$

**总结：**

| |
|---|
| 简单比较：$2(n-1)$次比较 |
| 只有当对最大元素的比较失败时才对最小元素进行比较 |
| 最好情况：数组为升序序列时需要 $n-1$ 次比较 |
| 最坏情况：数组为降序序列时需要 $2(n-1)$次比较 |
| 平均情况：$3n/2-1$ 次比较 |

**注意：**对于分治技术请参见第 18 章。

**问题 4** 给出算法，在给定的输入列表中查找第二大的元素。

**解答：**蛮力法

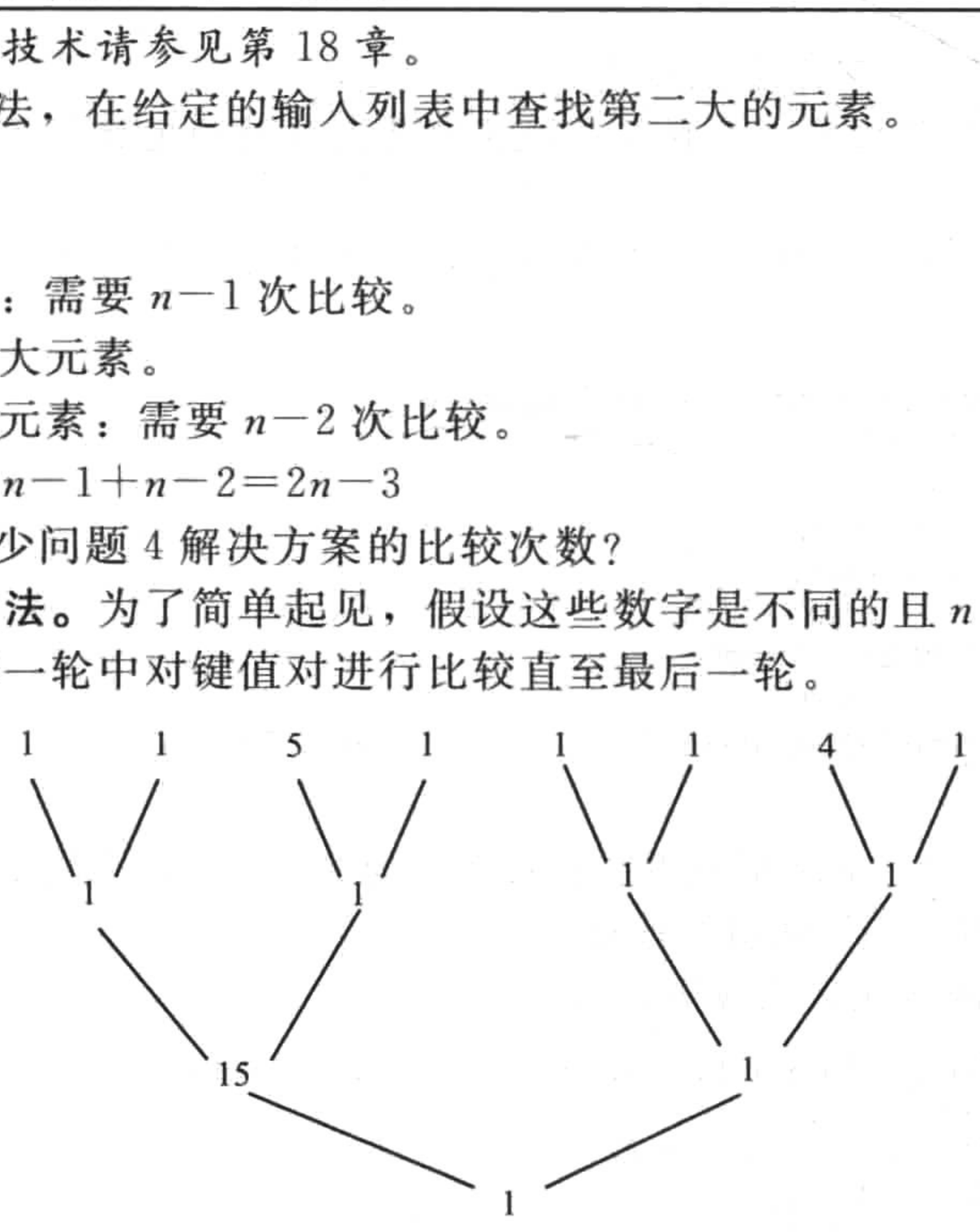
**算法：**

- 查找最大元素：需要 $n-1$ 次比较。
- 删除(丢弃)最大元素。
- 再次查找最大元素：需要 $n-2$ 次比较。

比较的总次数：$n-1+n-2=2n-3$

**问题 5** 能否减少问题 4 解决方案的比较次数？

**解答：锦标赛方法。**为了简单起见，假设这些数字是不同的且 $n$ 是 2 的整数次方。将键值组成对，并在每一轮中对键值对进行比较直至最后一轮。




如果输入包含 8 个键值，则在第一轮有 4 次比较，在第二轮有 2 次比较，在最后一轮

有 1 次比较。最后一轮的胜者就是最大的键值。上图展示了该方法。锦标赛方法仅当 $n$ 是 2 的整数次方时才适用。当 $n$ 不是 2 的整数次方时，只需要在数组后面添加足够的元素使数组的长度成为 2 的整数次方即可。如果该树是完全树，则树的最大高度是 $\log n$。若构建一棵完全二叉树，则需要 $n-1$ 次比较就可以找到最大值。

第二大的键值必定位于那些与最大键值比较过程中输掉的键值中。这意味着，第二大的元素一定是最大元素的对手之一。与最大键值比较并输掉的键值的数目等于树的高度，即 $\log n$(如果这棵树是完全二叉树)。利用选择算法在这些输掉的键值中查找最大键值，需要 $\log n-1$ 次比较。因此查找最大和第二大的键值的总的比较次数为 $n+\log n-2$。

**问题 6**　在 $n$ 个元素组成的数组 $S$ 中使用划分方法寻找前 $k$ 个最小元素。

输入：正整数 $n$ 和 $k$，其中 $k=n$，数组 $S$ 的索引为从 $1\sim n$。

输出：数组 $S$ 的前 $k$ 个最小元素，即选择函数的返回值。

**解答：蛮力法。**对数组扫描 $k$ 次以便得到所需要的元素。该方法用于冒泡排序(和选择排序)，每次通过比较整个序列的所有元素来寻找最小元素。这种方法必须遍历序列 $k$ 次。因此复杂度为 $O(n\times k)$。

**问题 7**　能否使用排序技术解决问题 6?

**解答：可以。**排序后取出前 $k$ 个元素。

1) 对这些数字排序。

2) 选择前 $k$ 个元素。

复杂度非常低。对 $n$ 个数字排序是 $O(n\log n)$，选择第 $k$ 个元素是 $O(k)$。总复杂度是 $O(n\log n+k)=O(n\log n)$。

**问题 8**　能否使用树排序技术解决问题 6?

**解答：可以。**

1) 将所有元素插入一棵二叉搜索树。

2) 执行中序遍历并输出前 $k$ 个元素，就可以得到 $k$ 个最小的元素。

创建一棵具有 $n$ 个元素的二叉搜索树的代价是 $O(n\log n)$，遍历 $k$ 个元素的复杂度是 $O(k)$。因此总的时间复杂度是 $O(n\log n+k)=O(n\log n)$。

**缺点：**若元素是降序排序的，则将得到一个向左偏的树。在这种情况下，构建这棵树的代价是 $0+1+2+\cdots+(n-1)=\dfrac{n(n-1)}{2}$，即复杂度为 $O(n^2)$。为了避免这种情况，可以使这棵树保持平衡，这样构建这棵树的代价仅为 $n\log n$。

**问题 9**　能否改进树排序技术来解决问题 6?

**解答：可以。**采用一棵较小的树可以得出相同的结果。

1) 取序列中前 $k$ 个元素来创建一棵 $k$ 个结点的平衡树(代价为 $k\log k$)。

2) 对其余的元素依次执行如下操作：

a. 如果该元素比树中最大的元素大，则返回。

b. 如果该元素比树中最大的元素小，则删除树的最大元素，添加该元素。这是为了确保一个较小的元素替换树中一个较大的元素。由于这棵树是由 $k$ 个元素组成的平衡树，所以该操作的代价是 $\log k$。

一旦步骤 2 完成，这棵由 $k$ 个元素组成的平衡树将会包含最小的 $k$ 个元素。随后只需要输出这棵树中的最大元素即可。

时间复杂度：

1）对于前 $k$ 个元素，创建一棵平衡树，因此代价是 $k\log k$。

2）对于其余的 $n-k$ 个元素，复杂度是 $O(\log k)$。

步骤 2 的复杂度为 $(n-k)\log k$，总的开销是 $k\log k+(n-k)\log k=n\log k$，即 $O(n\log k)$。事实上，该复杂度的上限要优于之前提供的方法。

**问题 10** 能否使用划分技术解决问题 6？

**解答：可以。**

**算法：**

1）从数组中选择一个枢轴。

2）对数组进行划分，使得：

$$A[\text{low}..\text{pivotpoint}-1]\leqslant \text{pivotpoint}\leqslant A[\text{pivotpoint}+1..\text{high}]$$

3）如果 $k<$pivotpoint，则它必定在枢轴左侧，因此对左边进行同样的递归操作。

4）如果 $k=$pivotpoint，则它必定是枢轴，因此输出从 low 到 pivotpoint 所有的元素。

5）如果 $k>$pivotpoint，则它必定在枢轴右侧，因此对右边进行同样的递归操作。

顶层函数调用应为 kthSmallest=Selection(1，n，k)。

```
int Selection (int low, int high, int k) {
        int pivotpoint;
        if(low == high)
                return S[low];
        else {
                pivotpoint = Partition (low, high);
                if(k == pivotpoint)
                        return S[pivotpoint];
                else if(k < pivotpoint)
                        return Selection (low, pivotpoint - 1, k);
                else    return Selection (pivotpoint + 1, high, k);
        }
}
void Partition (int low, int high) {
        int i, j = low, pivotitem= S[low];
        for (i = low + 1; i <= high; i++)
                if(S[i] < pivotitem) {
                        j++;
                        Swap S[i] and S[j];
                }
        pivotpoint = j;
        Swap S[low] and S[pivotpoint];
        return pivotpoint;
}
```

时间复杂度：最坏情况下与快速排序相同，为 $O(n^2)$。尽管如此，但该方法的平均性能要好得多($O(n\log k)$——平均情况)。

**问题 11** 给定一个含有 $n$ 个元素的数组 $S$，采用最好的方式查找 $S$ 中的第 $k$ 小元素。

**解答：**该问题类似于问题 6，问题 6 的所有解对于该问题都是有效的。唯一的区别是，不需要输出所有 $k$ 个元素，只需要输出第 $k$ 个元素即可。可以通过中位数的中位数算法来改进该解法。中位数是选择算法的一个特例。算法 Selection($A$，$k$)用来从具有 $n$ 个

元素的集合 $A$ 中查找第 $k$ 个最小的元素，具体如下。

**算法：** Selection($A$，$k$)

1）将 $A$ 划分为 ceil$\left(\frac{\text{length}(A)}{5}\right)$个组，每组有 5 个元素(最后一组可能有较少的元素)。

2）分别对每组排序(比如，插入排序)。

3）找到$\frac{n}{5}$组中每个组的中位数并将其存储在某个数组中(记为 $A'$)。

4）采用 Selection 函数递归地查找 $A'$的中位数(中位数的中位数)，记为 $m$。

$$m = \text{Selection}\left(A', \frac{\frac{\text{length}(A)}{5}}{2}\right)$$

5）令 $q$=$A$ 中小于 $m$ 的元素。

6）If ($k$==$q$+1 )

returnm ;

/ * 利用枢轴进行划分 * /

7）否则，将 $A$ 分为 $X$ 和 $Y$ 两部分。

$X$=｛小于 $m$ 的元素｝

$Y$=｛大于 $m$ 的元素｝

/ * 随后，形成一个子问题 * /

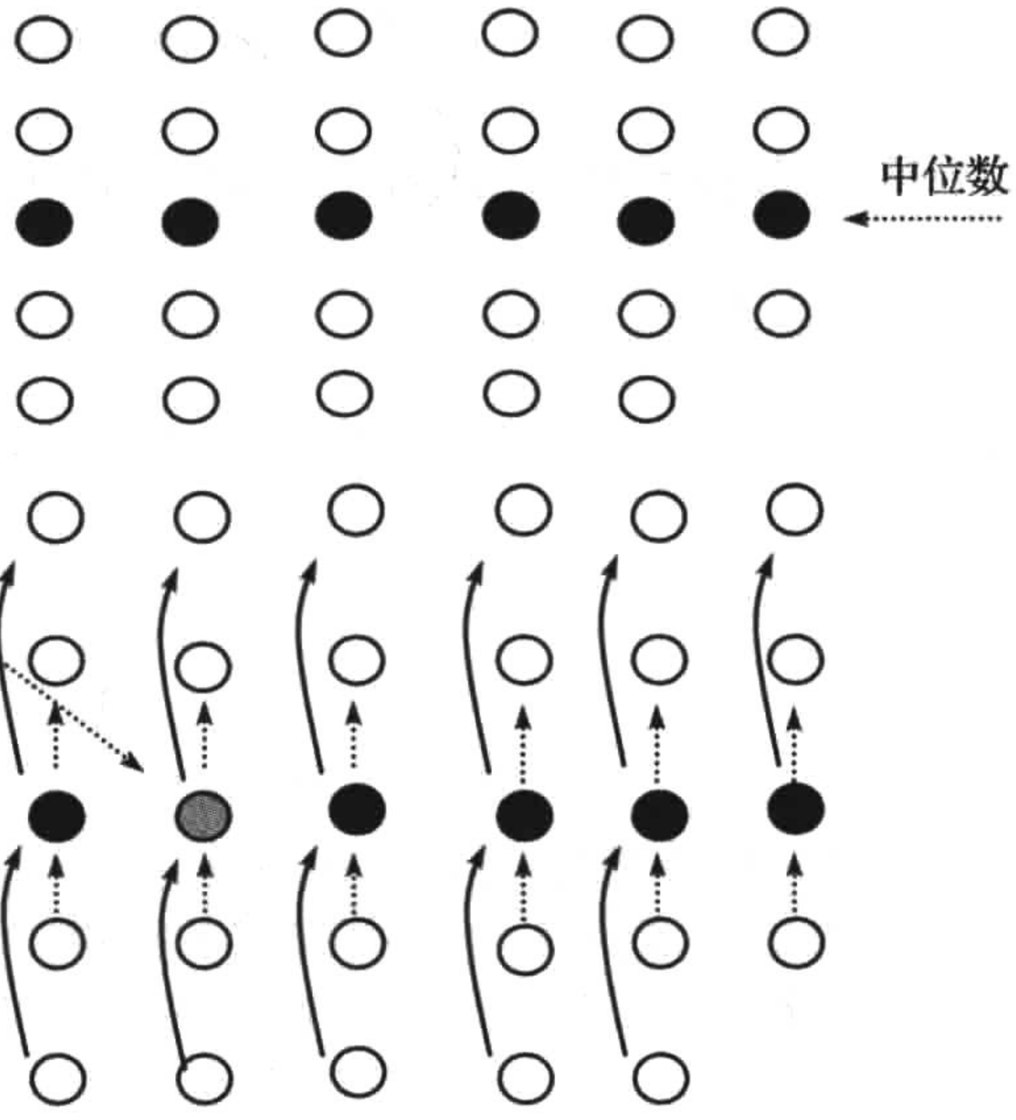
8）If($k$<$q$+1 )

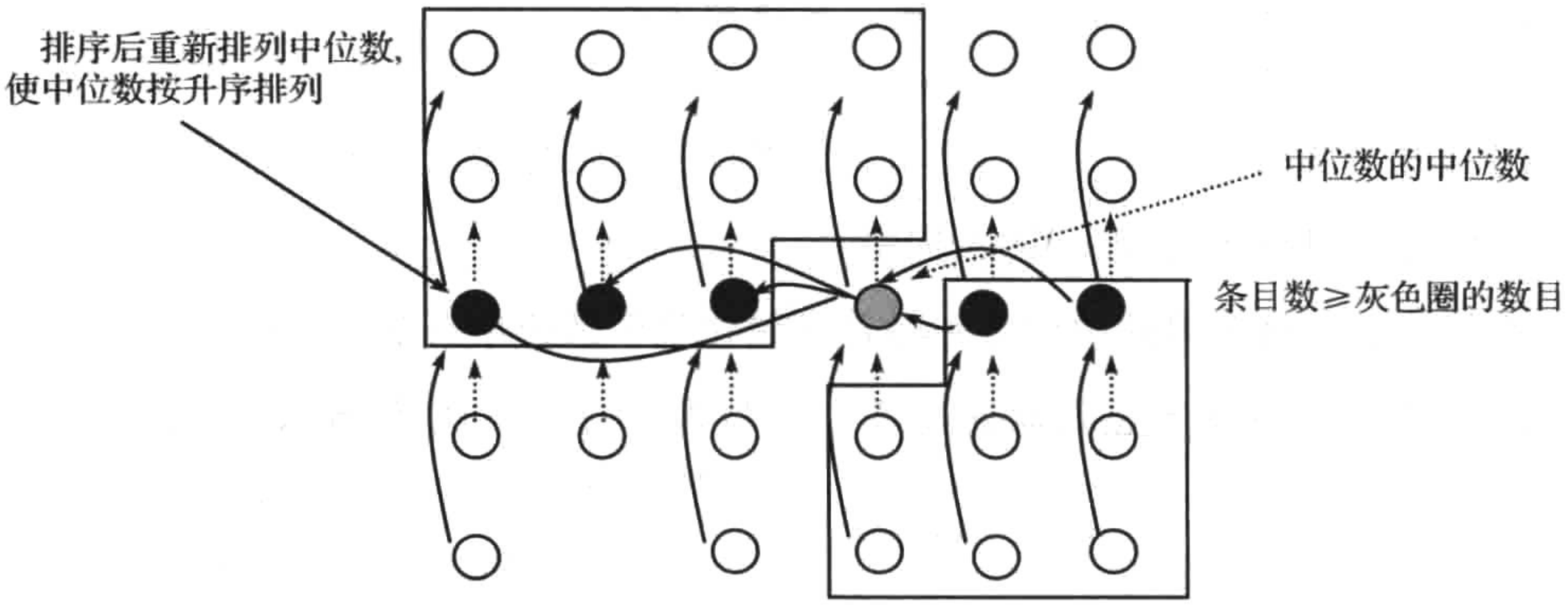
return Selection($X$，$k$)；

9）Else

return Selection($Y$，$k$−($q$+1))；

在递归调用前，考虑如下表示作为输入。图中每个圆圈代表一个元素，每一列由 5 个元素组成。黑色的圆圈表示每组 5 个元素的中位数。如前所述，使用复杂度为常数时间的插入排序对每一列进行排序。






在上图中，灰色圆点是中位数的中位数(记为 $m$)。可以看出，在由 5 个元素构成的数组中，至少有 1/2 个组的中位数$\leqslant m$。同样，这 1/2 个组贡献了 3 个($\geqslant m$ 的)元素，除了 2 个组(可能包含少于 5 个元素的最后一组和包含 $m$ 的组)外。同样，如上图所示，至少有 1/2 个组贡献了 3 个($\geqslant m$ 的)元素。除了 2 个组外，至少有 1/2 个组贡献 3 个元素有：$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil-2\right)\approx\frac{3n}{10}-6$。剩余的是 $n-\frac{3n}{10}-6\approx\frac{7n}{10}+6$。因此$\frac{7n}{10}+6$ 大于$\frac{3n}{10}-6$，所以需要考虑$\frac{7n}{10}+6$ 作为最坏情况进行分析。

**递归的构成：**

- 在选择算法中，选择中位数的中位数 $m$ 作为枢轴，把 $A$ 分成两个集合 $X$ 和 $Y$。需要选择具有最大规模的集合。
- 从过程 partition 调用 Selection 函数时所需的时间。当从该过程调用 Selection 函数时，输入的键值数是$\frac{n}{5}$。
- 划分数组所需的比较次数。该值为 length($S$)，记为 $n$。

因此，可以建立如下递归：$T(n)=T\left(\frac{n}{5}\right)+\Theta(n)+\mathrm{Max}\{T(X),\ T(Y)\}$。

从上述讨论可以看到，若选择中位数的中位数 $m$ 作为枢轴，划分的大小是：$\frac{3n}{10}-6$ 和$\frac{7n}{10}+6$。若选择这些中的最大的，则有，

$$T(n)=T\left(\frac{n}{5}\right)+\Theta(n)+T\left(\frac{7n}{10}+6\right)\approx T\left(\frac{n}{5}\right)+\Theta(n)+T\left(\frac{7n}{10}\right)+O(1)$$

$$\leqslant c\,\frac{7n}{10}+c\,\frac{n}{5}+\Theta(n)+O(1)$$

最后得到

$$T(n)=\Theta(n)$$

**问题 12** 在问题 11 中，输入数组被分为由 5 个元素组成的组。常数 5 在分析中发挥了重要作用。是否可以将数组分成由 3 个元素构成的组并在线性时间内求解问题 11?

**解答：**在这种情况下，修改会导致程序运行超过线性时间。最坏情况下，在分组步

骤中寻找$\left\lceil \frac{n}{3} \right\rceil$个中位数，其中至少有一半大于的中位数的中位数 $m$，但是只有两个组提供大于 $m$ 的元素少于两个。因此，作为上界，大于枢轴的元素数至少是：

$$2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil - 2\right) \geqslant \frac{n}{3} - 4$$

同样，这也是下界。因此，最多有 $n-\left(\frac{n}{3}-4\right)=\frac{2n}{3}+4$ 个元素在递归调用 Select 函数时作为输入。递归步骤用来在规模为$\left\lceil \frac{n}{3} \right\rceil$的问题上查找中位数的中位数，因此递归的时间是：

$$T(n) = T(\lceil n/3 \rceil) + T(2n/3+4) + \Theta(n)$$

假设 $T(n)$是单调递增的，则可以得出结论 $T\left(\frac{2n}{3}+4\right)\geqslant T\left(\frac{2n}{3}\right)\geqslant 2T\left(\frac{n}{3}\right)$，因此可以认为上界是 $T(n)\geqslant 3T\left(\frac{n}{3}\right)+\Theta(n)$，时间复杂度为 $O(n\log n)$。因此，不能选择 3 作为分组的大小。

**问题 13**　与问题 12 一样，能否用大小为 7 的组？

**解答：**根据类似的理由可以再次修改程序，现在用 7 个元素的分组代替 5 个元素的分组。最坏情况下，在分组步骤中寻找$\left\lceil \frac{n}{7} \right\rceil$个中位数，其中至少有一半大于中位数的中位数 $m$，但是只有两个组提供大于 $m$ 的元素少于 4 个。因此作为一个上界，大于枢轴的元素数至少是：

$$4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{7} \right\rceil \right\rceil - 2\right) \geqslant \frac{2n}{7} - 8$$

同样，这也是一个下界。因此最多有 $n-\left(\frac{2n}{7}-8\right)=\frac{5n}{7}+8$ 个元素在递归调用 Select 函数时作为输入。查找中位数的中位数的递归步骤运行在一个规模为$\left\lceil \frac{n}{7} \right\rceil$的问题上，因此递归的时间是

$$T(n) = T\left(\left\lceil \frac{n}{7} \right\rceil\right) + T\left(\frac{5n}{7}+8\right) + O(n)$$

$$T(n) \leqslant c\left\lceil \frac{n}{7} \right\rceil + c\left(\frac{5n}{7}+8\right) + O(n)$$

$$\leqslant c\frac{n}{7} + c\frac{5n}{7} + 8c + an \quad (a \text{ 是一个常数})$$

$$= cn - c\frac{n}{7} + an + 9c$$

$$= (a+c)n - \left(c\frac{n}{7} - 9c\right)$$

当 $c\frac{n}{7}-9c\geqslant 0$ 时，该公式由$(a+c)n$ 界定。因此可以选择 7 作为分组的大小。

**问题 14**　给定两个数组，每个数组都包含 $n$ 个有序元素，给出一个时间复杂度为 $O(\log n)$的算法来查找这 $2n$ 个元素的中位数。

**解答：**该问题的一个简单解法是合并这两个列表来计算中间两个元素的平均值(注意，合并总是包含偶数个值)。但是，合并所需要的时间复杂度是 $O(n)$，不能满足这个问题的要求。为了得到 log$n$ 的复杂度，通常的做法是来用折半查找。令 medianA 和 medianB 分别表示两个列表的中位数(因为两个列表是有序的，所以很容易获得)。如果 medianA==medianB，那么该值就是所求并集的总的中位数。否则，并集的中位数一定在 medianA 和 medianB 之间。假设 medianA<medianB(相反的情况完全类似)，则需要寻找如下两个集合的并集的中位数：

$$\{x \text{ in } A \mid x \geqslant \text{medianA}\}\{x \text{ in } B \mid x \leqslant \text{medianB}\}$$

因此，可以通过递归地重新设置两个数组的边界来执行这个操作。算法用两个索引变量来跟踪两个排序的数组。这些索引用来存取和比较这两个数组的中位数，以便寻找总的中位数的位置。

```
FindMedian(int A[], int alo , int ahi, int B[], int blo int bhi) {
        amid = alo + (ahi-alo)/2;
        amed = a[amid];
        bmid = blo + (bhi-blo)/2;
        bmed = b[bmid];
        if( ahi - alo + bhi - blo < 4) {
                处理边界情况并在O(1)时间内解决这个小问题
                return;
        }
        else if(amed < bmed)
                FindMedian(A, amid, ahi, B, blo, bmid+1);
        else    FindMedian(A, alo, amid+1,B, bmid+1, bhi);
}
```

时间复杂度为 $O(\log n)$，因为算法每次将问题的规模减半。

**问题 15** $A$ 和 $B$ 是两个各包含 $n$ 个元素的有序数组。在 $A$ 中寻找第 $k$ 个最小元素只需要简单地输出 $A[k]$，时间复杂度为 $O(1)$。同样，可以很容易地在 $B$ 中找出第 $k$ 个最小元素。给定一个时间复杂度为 $O(\log k)$ 的算法来寻找合并数组中的第 $k$ 个最小元素(也就是 $A$ 和 $B$ 的并集中的第 $k$ 个最小值)。

**解答：**这只是问题 14 的另一种描述方法。

**问题 16 在有序列表中寻找 $k$ 个最小元素：**给定由 $n$ 个有序元素构成的集合，寻找 $k$ 个最小元素，并按顺序列出。分析算法在最坏情况下的运行时间。

**解答：**对这些元素排序，然后列出 $k$ 个最小值。

$$\begin{aligned} T(n) &= \text{排序的时间复杂度} + \text{列出 } k \text{ 个最小元素的时间复杂度} \\ &= \Theta(n\log n) + \Theta(n) = \Theta(n\log n) \end{aligned}$$

**问题 17** 对于问题 16，若按照如下所示的方法，时间复杂度是多少？

**解答：**利用堆排序中的优先队列结构，构造一个关于此集合的最小堆，并执行提取最小值操作 $k$ 次。更多细节请参见第 7 章。

**问题 18** 对于问题 16，若按照如下所示的方法，时间复杂度是多少？

寻找集合中第 $k$ 个最小元素，以该元素作为枢轴进行划分，并对这 $k$ 个最小元素排序。

**解答：**

$$\begin{aligned} T(n) &= \text{求第 } k \text{ 个最小元素的时间复杂度} + \text{查找枢轴} + k \text{ 个最小元素排序} \\ &= \Theta(n) + \Theta(n) + \Theta(k\log k) \\ &= \Theta(n + k\log k) \end{aligned}$$

因为 $k \leqslant n$，所以该方法优于问题 16 和问题 17 的解法。

**问题 19**　在 $O(n)$时间内，寻找与 $n$ 个不同元素的中位数最近的 $k$ 个数。

**解答：**假设数组的元素是有序的。现在寻找 $n$ 个数字的中位数，将其索引记为 $X\left(\text{因为数组是有序的，所以中位数应在}\frac{n}{2}\text{的位置}\right)$。算法只需要在 $X-1 \sim 0$ 之间及 $X+1 \sim n-1$ 之间选择 $k$ 个与中位数的差的绝对值最小的元素。

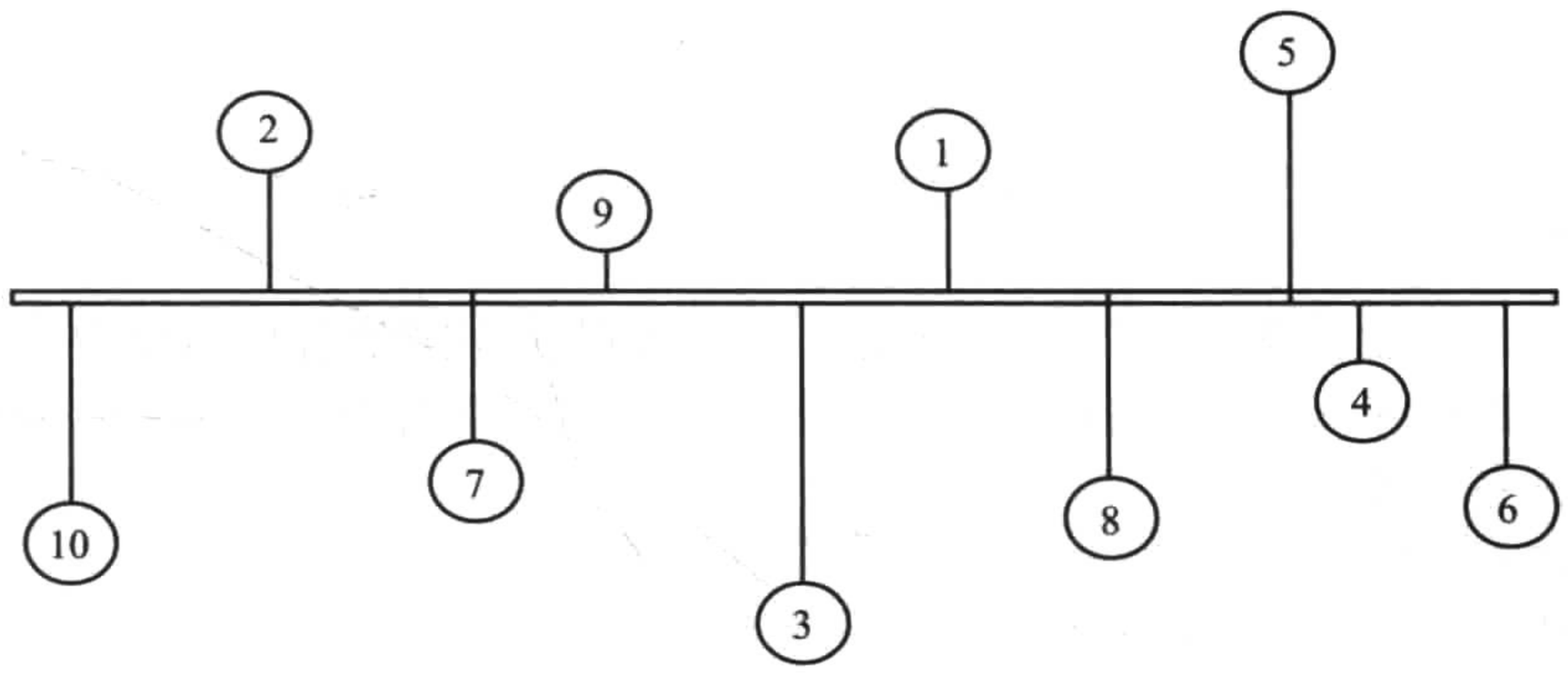
时间复杂度：每一步需要 $O(n)$时间，故算法的总时间复杂度是 $O(n)$。

**问题 20**　有其他方法可以解决问题 19 吗?

**解答：**为了简单起见，假设 $n$ 是奇数而 $k$ 是偶数。如果集合 $A$ 是有序的，则中位数就是在 $n/2$ 位置，并且在 $A$ 中最接近于中位数的 $k$ 个元素的位置为$(n-k)/2 \sim (n+k)/2$。

首先用线性时间的选择算法寻找第$(n-k)/2$ 个、第 $n/2$ 个和第$(n+k)/2$ 个元素，遍历集合 $A$ 查找小于$(n+k)/2$、大于$((n-k)/2$ 和不等于 $n/2$ 的元素。因为需要使用线性时间的选择算法 3 次，并对集合 $A$ 中的 $n$ 个元素遍历一次，所以算法的时间复杂度为$O(n)$。

**问题 21**　给出 $n$ 个房屋的坐标$(x, y)$，如何选取地点来建造一条平行于 $x$ 轴的公路，使得车道的建设成本最小。



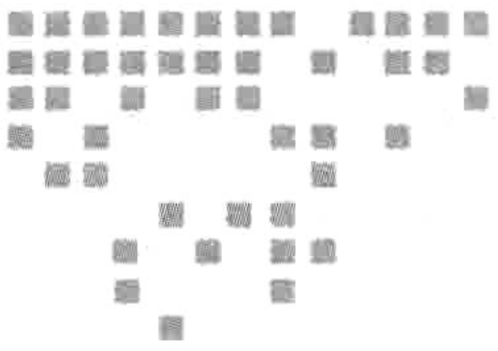

**解答：**公路的构建没有花费，而车道是需要花费的。车道的花费与其到公路的距离成正比。显然，它们与路是垂直的。解决办法是把街道建在所有 $y$ 坐标的中位数处。

**问题 22**　给定一个包含数十亿数字的大文件。从该文件中寻找 10 个最大的数。

**解答：**请参见第 7 章。

**问题 23**　假设有一家牛奶公司。该公司每天从它的代理商收集牛奶。这些代理商位于不同的地方。为了收集牛奶且使总的距离最小，从哪里开始最好呢?

**解答：**从中间开始可以减少总的行驶距离，因为那是到所有其他地方的中心位置。



Chapter 13

# 第 13 章

# 符 号 表

## 13.1 引言

从童年开始，我们都曾经使用过字典，很多人现在使用带有拼写检查程序的文字处理软件(如微软的 Word)。拼写检查程序也是一种使用范围有限的字典。字典的例子很多但以下类型的字典较少：

- 拼写检查程序。
- 建立在数据库管理应用程序基础上的数据字典。
- 由加载器(loader)、汇编器(assembler)和编译器(compiler)产生的符号表。
- 网络组件(DNS 查找)中的路由表。

在计算机科学中，当涉及抽象数据类型时，通常使用术语符号表而不使用字典。

## 13.2 什么是符号表

符号表是关联值和键值的一种数据结构，它支持以下操作：

- 查找某个特殊名称是否在表中。
- 获取名称的属性。
- 修改名称的属性。
- 插入名称及其属性。
- 删除名称及其属性。

符号表中仅有 3 种基本操作：查找、插入和删除。

**例子**：DNS 查找。假设键值是 URL，值是 IP 地址。

- 插入具有指定 IP 地址的 URL。
- 给定 URL，查找对应的 IP 地址。

| 键值(网站) | 值(IP地址) | 键值(网站) | 值(IP地址) |
|---|---|---|---|
| www. abc. com | 128. 112. 136. 11 | www. klm. com | 128. 103. 060. 55 |
| www. def. com | 128. 112. 128. 15 | www. CareerMonk. com | 209. 052. 165. 60 |
| www. ghi. com | 130. 132. 143. 21 | | |

## 13.3 符号表的实现

在实现符号表前，先来列举可能的实现方法。符号表有多种实现方法，下面是其中的几种实现方法。

**1. 无序数组实现**

该方法只需要维护一个数组，在最坏情况下查找、插入和删除操作的时间复杂度为 $O(n)$。

**2. 有序数组实现**

该方法需要维护一个包含键值和值的有序数组。

- 按键值进行排序存储。
- keys[$i$]=第 $i$ 个最大键值。
- values[$i$]=第 $i$ 个最大键值对应的值。

由于元素在数组中有序存储，所以可采用二分查找来寻找元素。最坏情况下，查找操作的时间复杂度为 $O(\log n)$，插入和删除操作的时间复杂度为 $O(n)$。

**3. 无序链表实现**

该方法只需要维护一个具有两个数据值的链表。在最坏情况下，查找、插入和删除操作的时间复杂度为 $O(n)$。

**4. 有序链表实现**

该方法插入键值时需要维护链表中键值的顺序。即使对链表进行排序，最坏情况下查找、插入和删除操作的时间复杂度也为 $O(n)$。

**5. 二叉搜索树实现**

参见第6章。该方法的优点是代码量较少和快速搜索(平均时间复杂度为 $O(\log n)$)。

**6. 平衡二叉搜索树实现**

参见第6章。该方法是二叉搜索树实现的扩展。最坏情况下，查找、插入和删除操作的时间复杂度为 $O(\log n)$。

**7. 三叉搜索实现**

参见第15章。这是一种实现字典的重要方法。

**8. 散列实现**

该方法很重要，完整讨论请参见第14章。

## 13.4 符号表实现方法的比较

下面是所有符号表实现方法的比较。

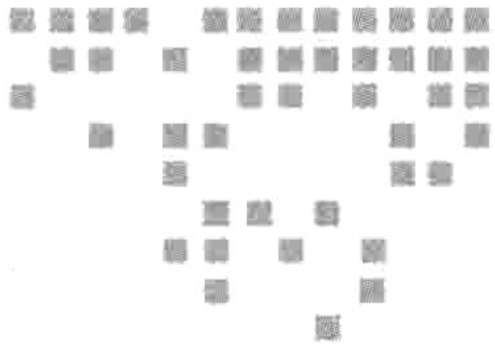
| 实现方法 | 查找 | 插入 | 删除 |
|---|---|---|---|
| 无序数组 | $n$ | $n$ | $n$ |
| 有序数组(能用数组二叉搜索实现) | $\log n$ | $n$ | $n$ |

（续）

| 实现方法 | 查找 | 插入 | 删除 |
| --- | --- | --- | --- |
| 无序链表 | $n$ | $n$ | $n$ |
| 有序链表 | $n$ | $n$ | $n$ |
| 二叉搜索树（平均时间复杂度 $O(\log n)$） | $\log n$ | $\log n$ | $\log n$ |
| 平衡二叉搜索树（最坏时间复杂度 $O(\log n)$） | $\log n$ | $\log n$ | $\log n$ |
| 三叉搜索（仅对数底数改变） | $\log n$ | $\log n$ | $\log n$ |
| 散列（平均时间复杂度 $O(1)$） | 1 | 1 | 1 |

注：上表中 $n$ 是输入大小。

上表中仅罗列了本书中讨论的方法，还可能存在其他方法。



第 14 章 Chapter 14

# 散　列

## 14.1　什么是散列

散列是一种用以实现信息存储和快速检索的技术。它常用于执行优化搜索和符号表的实现。

## 14.2　为什么用散列

在第 6 章中已经介绍了平衡二叉搜索树的插入、删除和检索等操作的时间复杂度为 $O(\log n)$。在实际应用中，如果需要这些操作的时间复杂度为 $O(1)$，散列可以提供平均时间复杂度为 $O(1)$的实现方法，尽管在最坏情况下散列的时间复杂度仍然是 $O(n)$。

## 14.3　散列表 ADT

对散列表的常见操作是：

- CreatHashTable：创建一个新的散列表。
- HashSearch：在散列表中搜索关键字。
- HashInsert：在散列表中插入一个新的关键字。
- HashDelete：在散列表中删除一个关键字。
- DeleteHashTable：删除散列表。

## 14.4　散列的例子

简单地说，可以把数组看作一个散列表。为了理解散列表的使用，考虑如下的例子。如果数据中存在重复的元素，请给出输出第一个重复字符的算法。

设想可能的解决方案。一个简单直接的方法是：在给定的字符串中，检查每个字符是否重复。这种方法的时间复杂度为 $O(n^2)$，空间复杂度为 $O(1)$。

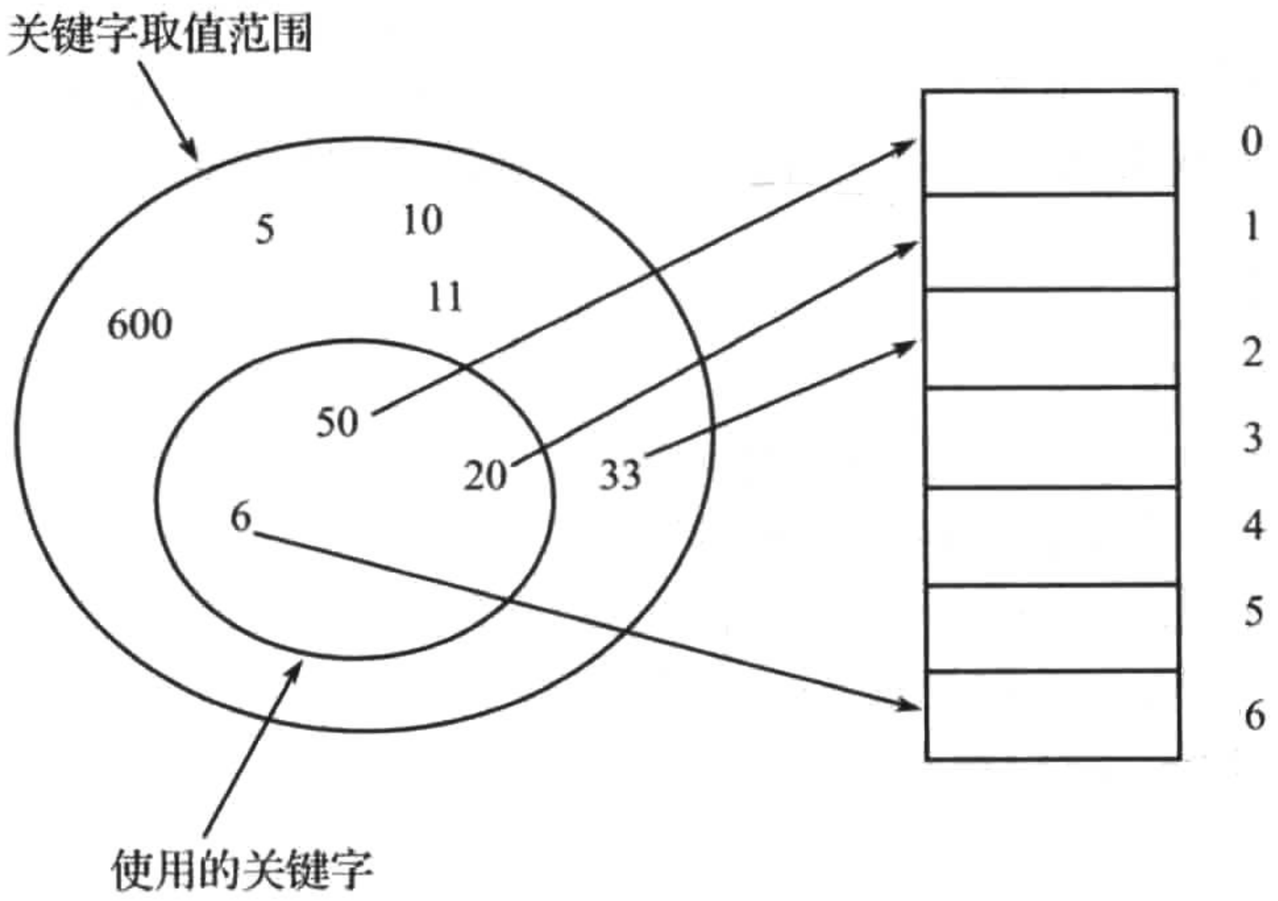
现在，给出一个更好的解决方案。既然目标是发现第一个重复的字符，那么如果记住某些数组前面的字符将怎样呢？

我们知道可能的字符集合的大小是 256(为了简单起见，假设只有 ASCII 字符)。由此创建一个长度为 256 的数组，并将其所有元素初始化为 0。将每个输入字符放到该数组对应的位置，并增加其计数。由于使用的是数组，所以它仅需要常数时间就可以到达任意指定的位置。当扫描输入字符时，若得到了一个计数器值已经是 1 的字符，则可以认为该字符就是第一个重复的字符。

```
char FirstRepeatedChar ( char [] str ) {
        int count[256]; //additional array
        for(int i=0; i<256; ++i)
                count[i] = 0;
        for(int i=0; i< str.length; ++i) {
                if(count[str[i]]==1) {
                        System.out.println(str[i]);
                        break;
                }
                else
                        count[str[i]]++;
        }
        if(i==len)
                System.out.println("No Repeated Characters");
        return 0;
}
```

**为什么不使用数组**

在前面的问题中，使用了长度为 256 的数组，这是因为我们事先知道不同的可能字符个数是 256。现在，对该问题稍稍进行改动。假设给定的数组是数字而不是字符，那么应该如何解决这个问题呢？




在这种情况下，关键字取值范围为无穷大(至少是非常大)。因此创建一个规模巨大的数组并存储计数器的值是不可能的，这意味着在内存中有一组通用的关键字和有限的位置。如果想要解决这个问题，则需要以某种方式将所有这些可能的关键字映射到可能的内存位置。

从上面的讨论和上图中可以看出，我们需要一个映射来将可能的关键字放到某个可用的存储位置。因此，使用简单的数组来解决那些关键字取值范围巨大的问题并不是一个正确的选择。将关键字映射到存储位置的过程称为散列。

**注意：** 现在，不用担心关键字如何映射到存储位置，这取决于所使用的转换函数。这类函数的一个简单实例就是关键字%表长。

## 14.5　散列的组成部分

散列有 4 个主要组成部分：

1）散列表。

2）散列函数。

3）冲突。

4）冲突解决技术。

## 14.6　散列表

散列表是数组的一个推广。在数组中，关键字为 $k$ 的元素将存储到数组的位置 $k$。这意味着，给定一个关键字 $k$，仅通过查看数组的第 $k$ 个位置就可以找到该元素，这称为直接寻址。

当能够为每个可能的关键字分配数组中的一个位置时，那么直接寻址是适用的。假设没有足够的空间来为每一个可能的关键字分配一个位置，那么就需要一个机制来处理这种情况。换句话说，当面临较少的存储位置和较多可能的关键字时，仅利用简单数组是不足以解决这种问题的。

在这些情况中，一种解决方案就是使用散列表。散列表或散列映射是一种数据结构，用以存储关键字及其关联的值。散列表利用散列函数将关键字映射到其关联的值。通用惯例是，当关键字的实际存储数目相对于关键字的取值范围较小时，可以使用散列表。

## 14.7　散列函数

散列函数用于将关键字转换成索引。理想情况下，散列函数应该将每个可能的关键字映射到唯一的槽索引，但在实践中难以实现。

### 1. 如何选择散列函数

与创建散列表相关的基本问题是：

- 应该设计一种有效的散列函数，使得插入对象的索引值在散列表中均匀分布。
- 应该设计一种有效的冲突解决算法，使得它能够为与之前插入散列表中的索引值发生冲突的当前关键字计算一个替代的索引值。
- 必须选择一个可以被快速计算的散列函数，且这个散列函数的取值在散列表位置的范围内，并能够最大限度地减少冲突。

### 2. 好的散列函数的特点

一个好的散列函数应具有以下特点：

- 最大限度地减少冲突。
- 简单并快速计算。
- 将键值(key value)在散列表中均匀分布。

- 能使用关键字提供的所有信息。
- 对一组给定的关键字具有一个高负载因子。

## 14.8 负载因子

一个非空散列表的负载因子是存储在表中的元素个数除以表的长度。这是面临再次散列或扩大现有散列表记录时的决策参数，有助于确定散列函数的效率。也就是说，该参数指出了散列函数是否将关键字均匀分布。

$$\text{负载因子} = \frac{\text{散列表中元素的个数}}{\text{散列表的长度}}$$

## 14.9 冲突

散列函数用于将每个关键字映射到不同的地址空间，但当无法创建一个将每个关键字映射到不同地址空间的散列函数时称为发生了冲突。冲突是指两个记录存储在相同位置的情况。

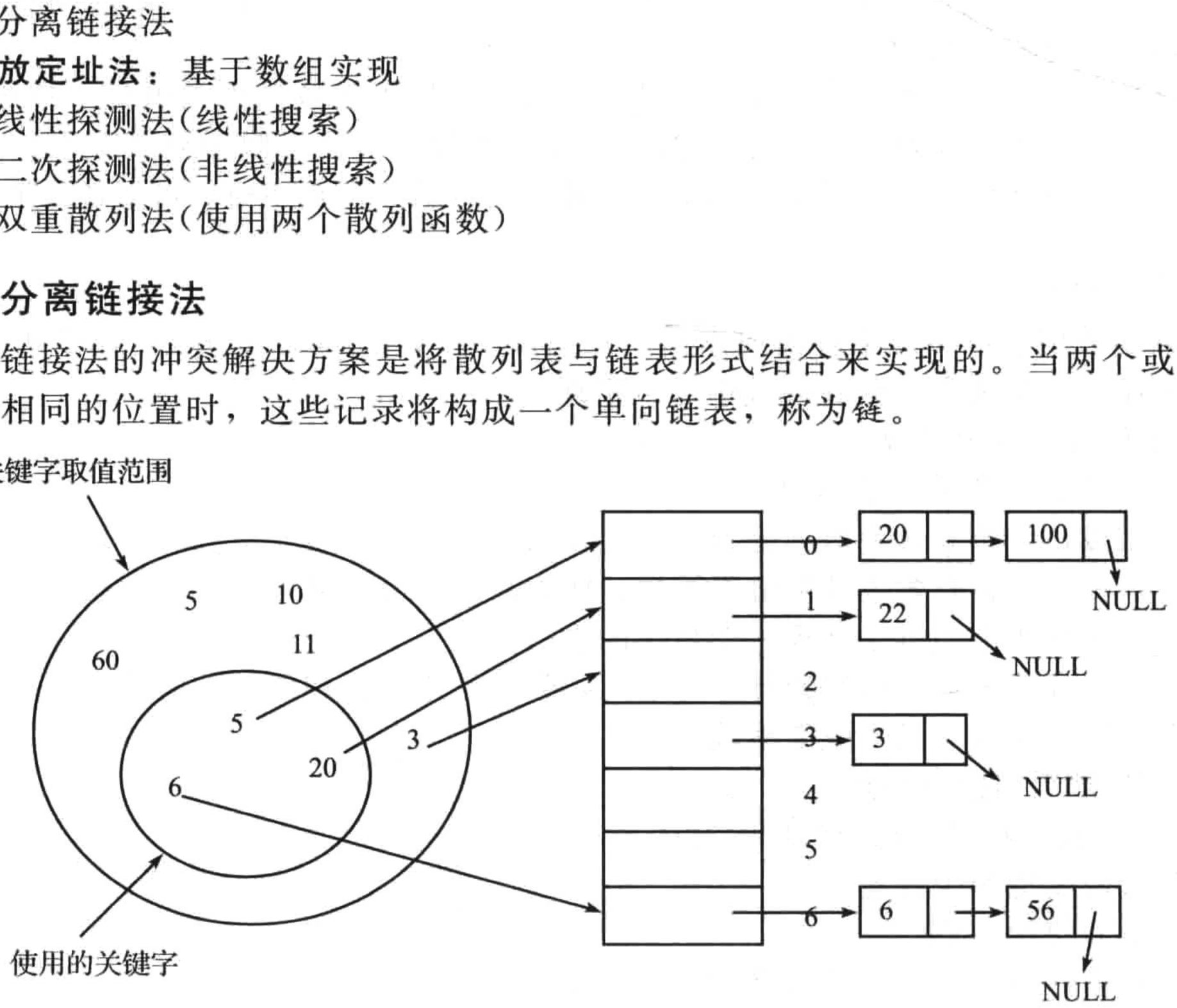
## 14.10 冲突解决技术

寻找替代位置的过程称为冲突解决。尽管散列表存在冲突问题，但在许多情况下，散列表比所有其他数据结构(如搜索树)更有效。目前已经有很多冲突解决技术，其中最常用的是直接链接法和开放定址法。

- **直接链接法**：链表数组的应用
  - 分离链接法
- **开放定址法**：基于数组实现
  - 线性探测法(线性搜索)
  - 二次探测法(非线性搜索)
  - 双重散列法(使用两个散列函数)

## 14.11 分离链接法

基于链接法的冲突解决方案是将散列表与链表形式结合来实现的。当两个或多个记录散列到相同的位置时，这些记录将构成一个单向链表，称为链。



## 14.12　开放定址法

在开放定址法中，所有关键字存储在散列表自身中，这种方法也称为闭散列法。该过程是基于探测的，即通过探测来解决冲突。

**1. 线性探测法**

探测间隔为固定值 1。在线性探测中，从发生冲突的原始位置开始按顺序搜索散列表。如果表中的某个位置被占据，则查找下一个位置。必要时，还可以从表的最后一个位置循环到表的第一个位置进行搜索。用于再次散列的函数如下：

$$\text{rehash(key)} = (n+1)\%\text{tablesize}$$

线性探测的一个问题是，表项往往在散列表中聚集，即散列表包含一组连续的被占据的位置，这一现象称为聚集。

聚集可以彼此接近，甚至合并成一个更大的聚集。因此，散列表中的某部分可能相当密集，即使另一部分元素相对较少。因此聚集会导致较长的探测搜索，从而降低整体效率。

探测的下一个位置由步长确定，探测步长的值(大于 1)可以是任意值。步长应该与散列表的长度互质，即它们的最大公约数应该等于 1。如果表的长度是素数，那么任何步长与表的长度都是互质的。但较大的步长值并不能避开表中的聚集。

**2. 二次探测法**

探测间隔的增加与散列值成正比(因此间隔线性地增加，索引值由一个二次函数描述)。聚集问题可以使用二次探测方法消除。在二次探测中，从发生冲突的初始位置 $i$ 开始，如果某个位置被占据，则探测 $i+1^2$、$i+2^2$、$i+3^2$、$i+4^2$ 等位置。如果有必要，将从表的最后一个位置循环到表的第一个位置进行探测。再次散列的函数如下：

$$\text{rehash(key)} = (n+k^2)\%\text{tablesize}$$

**例子：** 假设表的长度是 11(0..10)。

散列函数：$h(\text{key}) = \text{key mod } 11$

插入关键字：

31 mod 11 = 9

19 mod 11 = 8

2 mod 11 = 2

13 mod 11 = 2 → $(2+1^2)$mod 11= 3

25 mod 11 = 3 → $(3+1^2)$mod 11= 4

24 mod 11 = 2 → $(2+1^2)$mod 11，$(2+2^2)$mod 11 = 6

21 mod 11 = 10

9 mod 11 = 9 →$(9+1^2)$mod 11，$(9+2^2)$mod 11，$(9+3^2)$mod 11 = 7

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 2 |
| 3 | 13 |
| 4 | 25 |
| 5 | 5 |
| 6 | 24 |
| 7 | 9 |
| 8 | 19 |
| 9 | 31 |
| 10 | 21 |

尽管二次探测避免了聚集，但还是存在出现聚集的可能。聚集是由多个关键字映射到同一个散列值引起的，所以与这些关键字相关的探测序列将随着重复冲突的出现而被延长。线性探测和二次探测方法均使用了与搜索关键字相独立的探测序列。

**3. 双重散列法**

探测间隔由另一个散列函数计算生成，双重散列法用一种更好的方式减少了聚集。

由于探测序列的增量使用第二个散列函数计算，所以第二个散列函数 $h_2$ 应遵循：

$$h_2(\text{key}) \neq 0 \quad \text{且} \quad h_2 \neq h_1$$

算法首先探测位置 $h_1(\text{key})$。若该位置已经被占据，那么继续探测位置 $h_1(\text{key})+h_2(\text{key})$，$h_1(\text{key})+2\times h_2(\text{key})$，…。

**例子**：表长是 11(0..10)。

散列函数：假设$h_1(\text{key}) = \text{key mod } 11$

$$h_2(\text{key}) = 7-(\text{key mod } 7)$$

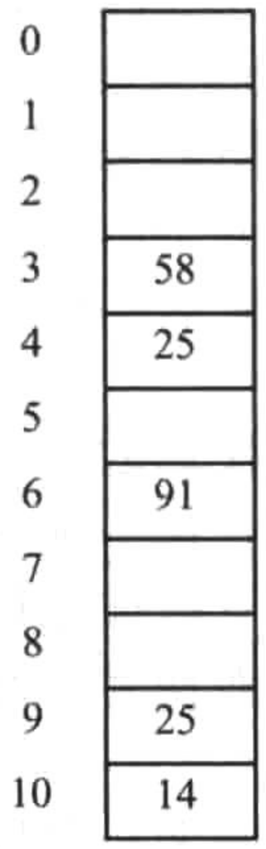
插入关键字：

58 mod 11 = 3

14 mod 11 = 3 → 3 + 7 = 10

91 mod 11 = 3 → 3 + 7，3+2×7 mod 11 = 6

25 mod 11 = 3 → 3 + 3，3 + 2×3 = 9




## 14.13 冲突解决技术的比较

### 1. 线性探测法与双重散列法的比较

线性探测法和双重散列法之间的选择取决于计算散列函数的代价和散列表的负载因子(每个槽的元素数)，虽然都使用探测但双重散列法需要更多时间，因为在长关键字的每次散列时该方法都要比较两个散列函数的值。

### 2. 开放定址法与分离链接法的比较

由于必须考虑内存的使用情况，所以两者都有些复杂。分离链接法用额外的内存来保存链接，开放定址法需要在散列表中包含额外内存来终止探测序列。当数据不具有唯一的关键字时，无法使用开放定址的散列表。一种替代方案就是使用分离链接散列表。

### 3. 开放定址方法的比较

| 线性探测法 | 二次探测法 | 双重散列法 |
| --- | --- | --- |
| 最快 | 最容易实现和应用 | 利用内存更有效 |
| 使用少量探测 | 使用额外的内存来保存链接，并不探测表中的所有位置 | 使用少量探测序列但需要花更多的时间 |
| 存在基本聚集的问题 | 存在二级聚集的问题 | 实现更加复杂 |
| 探测间隔通常固定为 1 | 探测间隔的增加与散列值成正比 | 探测间隔由另一个散列函数计算得到 |

## 14.14 散列如何达到 O(1)的时间复杂度

通过前面的讨论，有些人可能会质疑，假如将多个元素映射到同一位置，散列如何达到 O(1)的时间复杂度？这个问题的答案很简单。通过使用负载因子，可以确保每一块(例如，分离链接法中的链表)平均存储元素的最大数量小于负载因子。实际上，负载因子是一个常数(一般来说，10 或 20)。因此，在 20 个元素或 10 个元素中的搜索就变为常数。

如果在一块中的平均元素数大于负载因子，那么需用更大长度的散列表再次散列元素。有一点需要记住，当决定再次散列时，需要考虑平均占用率(散列表中的元素总数除

以表的长度)。

散列表的访问时间取决于负载因子，而负载因子反过来又依赖于散列函数。这是因为散列函数将元素分散到散列表中。为此，散列表具有 $O(1)$的平均时间复杂度。同样，在搜索操作多于插入和删除操作的情况下，通常使用散列表。

## 14.15　散列技术

目前有两种类型的散列技术：静态散列和动态散列。

### 1. 静态散列

如果数据是不固定的，则静态散列性能不佳，而动态散列是一个替代方案。在动态散列中，关键字集合可以动态改变。

### 2. 动态散列

如果数据是不固定的，则静态散列性能不佳，而动态散列是这种数据类型的一个选择。在动态散列中，关键字集合可以动态改变。

## 14.16　不适用散列表的问题

- 需要数据排序的问题。
- 具有多维数据的问题。
- 需要前缀搜索，特别是关键字很长且其长度变化的问题。
- 包含动态数据的问题。
- 数据不具有唯一关键字的问题。

## 14.17　布鲁姆过滤器

布鲁姆过滤器(Bloom filter)是一种概率数据结构，它用于判别一个元素是否存在于一个集合中，它具有较好的空间和时间效率。该方法可以判别元素要么一定不在集合中，要么可能在集合中。布鲁姆过滤器的基本数据结构是一个位向量。

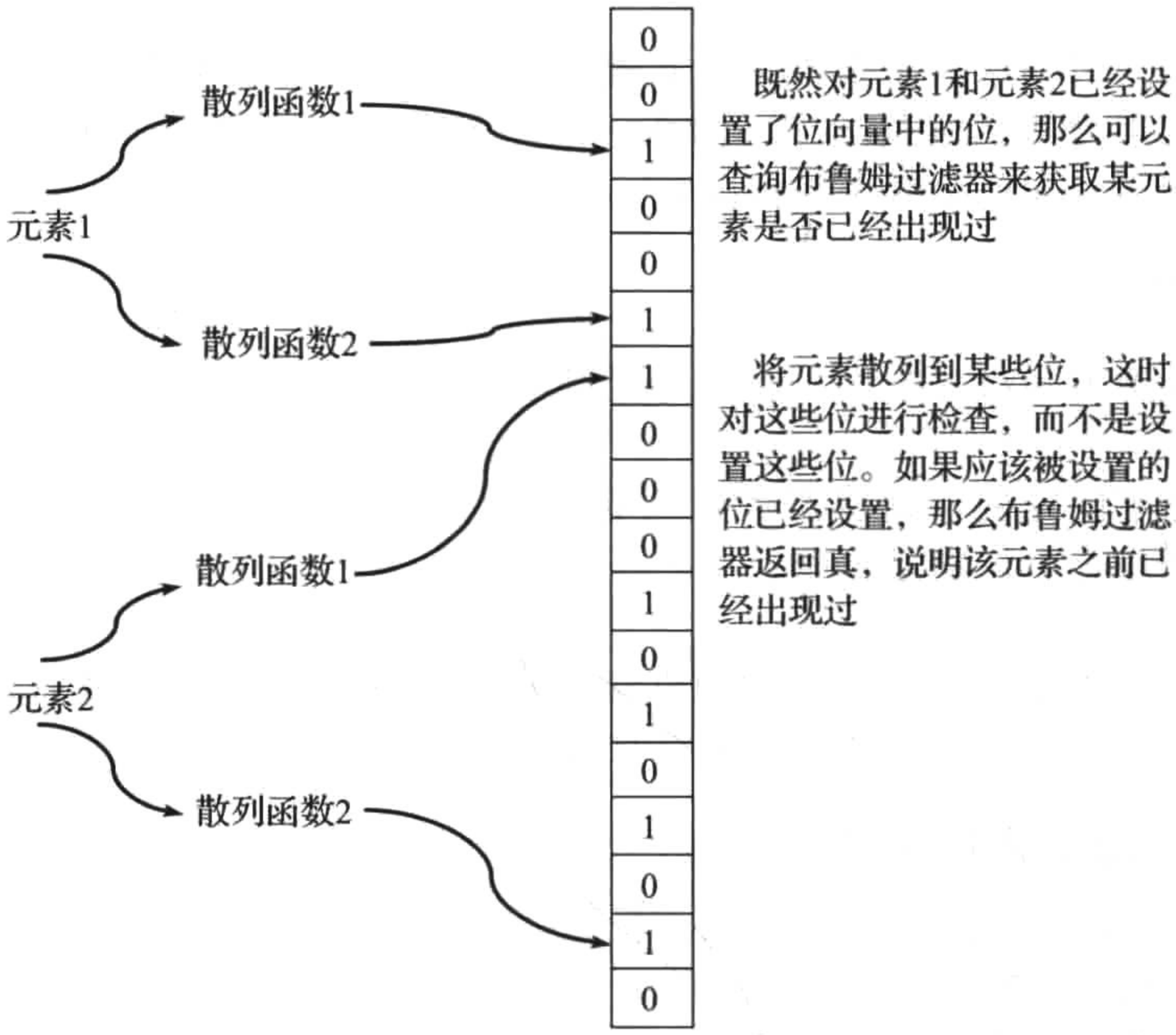
该算法由伯顿·布鲁姆(Burton Bloom)于 1970 年提出，它依赖于一系列不同散列函数的使用。

### 1. 它是如何工作的

布鲁姆过滤器从一个初始化为 0 的位数组开始。为了存储一数据值，简单地应用 $k$ 个不同的散列函数并将结果 $k$ 个值作为索引放入数组，设置 $k$ 个数组元素的值均为 1 并对遇到的每一个元素重复这个操作。

假设现在有一个元素出现并且我们想知道之前是否见过它。要做的就是应用 $k$ 个散列函数并查找指定的数组元素。如果其中任何一个散列函数的值为 0，则 100%可以确定以前从未遇到过该元素。如果之前遇到过该元素，则相应的位已经设置为 1。然而，即使所有的元素都为 1，也不能得出以前遇到过该元素这一结论，因为所有的位都会通过 $k$ 个散列函数应用于多个其他元素而被设置，因此可以得出的结论是：之前可能遇到过该元素。

注意，从布鲁姆过滤器中移除一个元素是不可能的，原因很简单，不能取消属于一个元素位的设置，因为这位也可能用于另一个元素的设置。




如果位数组大部分为空，即设置为 0，并且 $k$ 个散列函数是相互独立的，那么假阳性的概率(即认为已经看到某个数据项而实际上并没有看到)很低。例如，如果只设置了 $k$ 个位，则可以得出假阳性的概率非常接近 0。因为发生错误的唯一可能性是遇到了一个产生 $k$ 个相同散列值的数据项。但只要散列函数是独立的，产生相同散列值就是不可能的。

随着位数组的填充，假阳性的概率将逐渐地增加。当位数组被填满后，每个元素都视为已经出现过。显然可以用空间来交换精度和时间。

从布鲁姆过滤器中一次移除一个元素，可以通过另一个包含已被移除元素的布鲁姆过滤器来模拟实现。然而，第二个过滤器中的假阳性变成组合过滤器中的假阴性，这并不是期望出现的。在这种方法中，增加一个之前移除的数据项是不可能的，因为必须从包含移除元素的过滤器中移除它。

**2. 选择散列函数**

对于一个数值较大的 $k$，设计 $k$ 个不同的独立散列函数是不允许的。而对于一个具有宽输出的好的散列函数，任何不同位字段之间的相关性应该很小，所以这种类型的散列可以通过将它的输出划分为多个位字段来生成多个不同的散列函数。或者可以传送 $k$ 个不同的初始值(如 0，1，…，$k-1$)给一个需要初始值的散列函数，或添加(或附加)这些值给一个关键字。对于较大的 $m$ 和 $k$，散列函数之间的独立性可以适当放松，假阳性率的增加可以忽略不计。

**3. 选择位向量的长度**

布鲁姆过滤器具有 1%的误差和最优 $k$ 值，相反，每个元素只需要大约 9.6 位——无论元素的大小是多少。这种优势部分来自继承于数组的紧凑性，部分来自其概率性质。仅需要对每个元素增加大概 4.8 位，就可以将 1%的假阳性率降低一个数量级。

**4. 空间优势**

虽然面临假阳性的危险，但布鲁姆过滤器相对于表示集合的其他数据结构(如平衡二

叉搜索树、键树、散列表、简单数组或链表等)有较强的空间优势。这些结构中的大部分至少要求存储数据项本身，从几位(如小整数)到任意位(如字符串)(键树是一个例外，因为具有相等前缀的元素之间可以共享存储)。链接结构使用指针也将产生额外的线性空间开销。

然而，如果可能值的数量很小，而且其中的许多可以包含在集合中，那么布鲁姆过滤器的空间优势将很容易被确定位数的数组所取代，因为确定位数的数组对每个可能元素只要求 1 位。

**5. 时间优势**

布鲁姆过滤器还具有独特的属性，就是增加项或检查某项是否在集合中所需要的时间是一个固定常数 $O(k)$，完全独立于集合中已存在的项目数。其他固定空间大小的数据结构没有这个属性，但在实践中，稀疏散列表的平均访问时间比某些布鲁姆过滤器更短。然而，在硬件实现中，布鲁姆过滤器更快，因为它的 $k$ 查找是独立的且可以并行化。

## 14.18　散列的相关问题

**问题 1**　实现分离链接冲突解决技术。此外，讨论每个函数的时间复杂度。

**解答：** 创建一个给定长度为 $n$ 的散列表，并分配一组 $n/L$(其值通常是 5～20)指针给列表，初始化为空。执行搜索/插入/删除操作，首先使用散列函数对给定关键字计算表的索引值，然后在该位置所维护的线性列表中做相应的操作。为了使散列表的关键字均匀分布，表长为素数。

```
public class ListNode {
        private int key;
private int data;
private ListNode next;
public int getKey() {
        return key;
}
public void setKey(int key) {
        this.key = key;
}
public int getData() {
        return data;
}
public void setData(int data) {
        this.data = data;
}
public ListNode getNext() {
        return next;
}
        public void setNext(ListNode next) {
                this.next = next;
        }
}
public class HashTableNode {
        private int blockCount;
        private ListNode startNode;
        public int getBlockCount() {
                return blockCount;
        }
        public void setBlockCount(int blockCount) {
                this.blockCount = blockCount;
        }
```

```
        public ListNode getStartNode() {
                return startNode;
        }
        public void setStartNode(ListNode startNode) {
                this.startNode = startNode;
        }
}
public class HashTable {
        private int tSize;
        private int count;
        private HashTableNode[] table;
public int getTSize() {
        return tSize;
}
public void setTSize(int size) {
        tSize = size;
        table = new HashTableNode[size];
}
public int getCount() {
        return count;
}
public void setCount(int count) {
        this.count = count;
}
public HashTableNode[] getTable() {
        return table;
}
public void setTable(HashTableNode[] table) {
        this.table = table;
        }
}
public class HashTableOperations {
        public final static int LOADFACTOR = 20;
        public static HashTable createHashTable(int size){
                HashTable h = new HashTable();
                //count默认设置为0;
                h.setTSize(size/LOADFACTOR);
                for(int i=0;i<h.getTSize();i++){
                        h.getTable()[i].setStartNode(null);
                }
                return h;
        }
        public static int hashSearch(HashTable h, int data){
                ListNode temp;
                temp = h.getTable()[Hash(data, h.getTSize())].getStartNode();
                while(temp) {
                        if(temp.getData() == data)
                                return 1;
                        temp = temp.getNext();
                }
                return 0;

}
public static void hashInsert(HashTable h, int data){
        int index;
        ListNode temp, newNode;
        if(hashSearch(h, data))
                return 0;
```

```
    index = Hash(data, h.getTSize()); //假设Hash是内置函数
    temp = h.getTable()[index].getNext();
    newNode = new ListNode();
    if(newNode == null) {
            System.out.println("Memory Error");
            return;
}
newNode.setKey(index);
newNode.setData(data);
newNode.setNext(h.getTable()[index].getNext());
h.getTable()[index].setNext(newNode);
            h.getTable()[index].setBlockCount(h.getTable()[index].getBlockCount() + 1);
            h.setCount(h.getCount() + 1);
            if(h.getCount() / h.getTSize()  > LOAD_FACTOR)
                    Rehash(h);
            return;
}
public static boolean hashDelete(HashTable h, int data){
            ListNode temp, prev;
            int index = Hash(data, h.getTSize());
            for(temp = h.getTable()[index].getNext(), prev = null; temp;
                                    prev = temp, temp = temp.getNext()) {
                    if(temp.getData() == data) {
                            if(prev != null)
                                    prev.setNext(temp.getNext());
                            temp = null;
                            h.getTable()[index].setBlockCount(h.getTable()
                            [index].getBlockCount() - 1);
                            h.setCount(h.getCount() - 1);
                            return 1;
                    }
            }
            return 0;
}
public static void rehash(HashTable h){
            int oldsize, i, index;
            ListNode p, temp, temp2;
            HashTableNode oldTable;
            oldsize = h.getTSize();
            oldTable = h.getTable();
            h.setTSize(h.getTSize() * 2);
            h = new HashTable();
            if(!h.getTable()) {
                    System.out.println( "Memory Error");
                    return;
            }
            for(i = 0; i < oldsize; i++) {
                    for(temp = oldTable[i].getNext(); temp; temp = temp.getNext()) {
                            index = Hash(temp.getData(), h.getTSize());
                            temp2 = temp;
                            temp = temp.getNext();
                            temp2.setNext(h.getTable()[i].getNext());
                            h.getTable()[index].setNext(temp2);
                    }
            }

    }
}
```

CreatHashTable 的时间复杂度为 $O(n)$，HashSearch 的平均时间复杂度为 $O(1)$。

HashInsert 的平均时间复杂度为 $O(1)$，HashDelete 的平均时间复杂度为 $O(1)$。

**问题 2** 给出一个在给定字符数组中删除重复字符的算法。

**解答：** 从第一个字符开始，使用简单线性搜索检查它是否出现在字符串的其余部分。如果重复就把最后一个字符放到该位置并将字符串的长度减 1。对给定字符串的每个不同的字符，逐一执行这个过程。

```
void RemoveDuplicates(char[] s, int n) {
        for(int i = 0; i < n; i++) {
                for(int j = i+1; j < n; ) {
                        if(s[i] == s[j])
                                s[j] = s[--n];
                        else    j++;
                }
        }
        s[i] = '\0';
}
```

时间复杂度为 $O(n^2)$，空间复杂度为 $O(1)$。

**问题 3** 能否找到比时间复杂度 $O(n^2)$ 更好的其他方法来解决问题 2？不用考虑解决方案中的字符顺序。

**解答：** 使用排序将重复字符聚集在一起。通过扫描数组，删除连续重复位置的字符。

```
//字符数组作为输入
public static void removeDuplicates(char[] str, int len) {
        if (str == null)    return;
        if (len < 2)    return;
        int tail = 1;
        for (int i = 1; i < len; ++i) {
                for (int j = 0; j < tail; ++j) {
                        if (str[i] == str[j]) break;
                }
                if (j == tail) {
                        str[tail] = str[i];
                        ++tail;
                }
        }
        str[tail] = 0;
}
//字符串作为输入
public static String removeDuplicates(String s) {
        StringBuilder noDupes = new StringBuilder();
        for (int i = 0; i < s.length(); i++) {
                String si = s.substring(i, i + 1);
                if (noDupes.indexOf(si) == -1) {
                        noDupes.append(si);
                }
        }
        return noDupes.toString();
}
//排序方法
public static String removeDuplicates(String s) {
        char[] chars = s.toCharArray();
        Arrays.sort(chars);
        String sorted = new String(chars);
```

```
        System.out.println(sorted);
}
```

时间复杂度为 $\Theta(n\log n)$，空间复杂度为 $O(1)$。

**问题 4**　能否只扫描给定数组一次来解决问题 2？

**解答：**可以使用散列表来检查字符是否在给定字符串中重复出现。如果当前字符在散列表中不存在，则将当前字符插入散列表并在给定字符串中也保留该字符。如果当前字符存在于散列表中，则跳过该字符。

```
void RemoveDuplicates(char[] s) {
        int src, dst;
        HastTable h = new HashTable();
        current = last = 0;
        for(; s[current]; current++) {
                if( !HashSearch(h, s[current]))    {
                        s[last++] = s[current];

                        HashInsert(h, s[current]);
                }
        }
        s[last] = '\0';
}
```

平均时间复杂度为 $\Theta(n)$，空间复杂度为 $O(n)$。

**问题 5**　给定两个无序的数字数组，检查两个数组是否有相同的一组数字？

**解答：**假设两个给定的数组是 $A$ 和 $B$。该问题的一个简单解决方案是，对于 $A$ 的每个元素，检查该元素是否在 $B$ 中。但如果数组中有重复元素，则该方法将出现问题。例如，考虑下面的输入：

$$A = \{2,5,6,8,10,2,2\}$$
$$B = \{2,5,5,8,10,5,6\}$$

上面的算法将给出错误的结果，因为 $A$ 的每个元素在 $B$ 中也有。但如果观察出现的次数，它们是不相同的。该问题可以通过把已经比较过的元素移到列表末尾来解决。这意味着，如果在 $B$ 中找到一个元素，那么把此元素移到 $B$ 的末尾，在下次搜索中将不会找到那些元素。但这样做的缺点是，它需要额外的交换操作。这种方法的时间复杂度是 $O(n^2)$。因为对于 $A$ 中的每个元素，算法都将在数组 $B$ 中查找。

**问题 6**　能否降低问题 5 的时间复杂度？

**解答：可以**。为了降低时间复杂度，假设这两个列表已经排序。由于两个数组的长度都是 $n$，所以需要 $O(n\log n)$时间对它们排序。排序后，只需要用两个指针扫描这两个数组并查看它们每次是否指向相同的元素，并且持续移动指针直至数组的末尾。

这个方法的时间复杂度为 $O(n\log n)$，因为需要 $O(n\log n)$时间来为数组排序，排序后，需要 $O(n)$时间来扫描，但与 $O(n\log n)$相比它更小。

**问题 7**　能否进一步降低问题 5 的时间复杂度？

**解答：可以**。通过使用散列表。为此，考虑如下算法。

**算法：**

- 以数组 $A$ 的元素为关键字构建散列表。
- 在插入元素的同时跟踪每个数字的出现频率。即如果有重复元素，则增加相应关键字的计数值。

- 为数组 $A$ 的元素构建散列表后，扫描数组 $B$。
- 对于 $B$ 中元素的每一次出现，减少相应计数器的值。
- 最后，检查所有计数器是否为 0。
- 如果所有计数器都是 0，那么这两个数组相同，否则数组是不同的。

时间复杂度为 $O(n)$，主要用于扫描数组。空间复杂度为 $O(n)$，主要用于散列表的空间。

**问题 8** 给定一个数字对表，如果$(i, j)$、$(j, i)$对存在，则输出所有这样的对。例如，列表{{1，3}，{2，6}，{3，5}，{7，4}，{5，3}，{8，7}}中，{3，5}和{5，3}都存在，当遇到{5，3}时，则输出这个对。这种对称为均衡对。给出一个高效算法来找出所有这样的均衡对。

**解答：** 通过使用散列技术，仅用一次扫描就能解决这个问题，考虑如下算法。

**算法：**

- 逐一读取元素对并将它们插入散列表中。对于每对，将第一个元素看成关键字并将第二个元素看成值。
- 插入元素时，检查当前对的第二个元素的散列值是否与当前对的第一个数字相同。
- 如果它们相同，则表明存在均衡对，输出该对。
- 否则，插入该元素，即用当前对的第一个数字作为关键字，第二个数字作为值，并插入散列表中。
- 当完成对所有对扫描的同时，也输出了所有的均衡对。

时间复杂度为 $O(n)$，主要用于扫描数组，注意算法做的只是扫描输入。空间复杂度为 $O(n)$，主要用于散列表的空间。

**问题 9** 给定一个单向链表，检查链表中是否存在环。

**解答：** 使用散列表。

**算法：**

- 逐一遍历链表结点。
- 检查结点地址是否在散列表中。
- 如果它已经在散列表中，则表明正在访问的结点是已经访问过的结点。但这种情况只有在给定链表中存在环时才会出现。
- 如果该结点的地址在散列表中不存在，则将该结点地址插入散列表中。
- 继续这个过程，直至到达链表的末端或发现环。

时间复杂度为 $O(n)$，主要用于扫描链表，注意算法做的仅是扫描输入。空间复杂度为 $O(n)$，主要用于散列表的空间。

**注意：** 高效的解决方案可参见第 3 章。

**问题 10** 给定一个具有 101 个元素的数组。数组中有 50 个元素是不同的、24 个元素重复 2 次、1 个元素重复 3 次。在 $O(n)$时间内找出重复 3 次的元素。

**解答：** 使用散列表。

**算法：**

- 逐一扫描输入数组。
- 检测元素在散列表中是否存在。
- 如果它已经在散列表中，则增加它的计数值(该值表明元素出现的次数)。

- 如果该元素不在散列表中，则将这个结点插入散列表中，其计数值为 1。
- 继续这个过程，直至数组的末端。

时间复杂度为 $O(n)$，主要需要两次扫描。空间复杂度为 $O(n)$，主要用于散列表的空间。

**注意：** 高效的解决方案可参见第 11 章。

**问题 11**　给定 $m$ 个含有 $n$ 个元素的整数集合。给出一个算法来寻找这些集合中出现次数最多的元素。

**解答：** 使用散列表。

**算法：**

- 逐一扫描输入集合。
- 对每个元素跟踪计数，计数器表示元素在所有集合中的出现频率。
- 在完成所有集合的扫描后，选择具有最大计数值的一个元素。

时间复杂度为 $O(mn)$，主要需要扫描所有的集合。空间复杂度为 $O(mn)$，主要用于散列表的空间。因为在最坏情况下，所有的元素可能都不同。

**问题 12**　给定两个集合 $A$ 和 $B$，以及一个数字 $K$。给出一个算法来查找是否存在这样一对元素，其中一个来自 $A$，另一个来自 $B$，且两元素之和为 $K$。

**解答：** 为了简单起见，假设集合 $A$ 的长度为 $m$，集合 $B$ 的长度为 $n$。

**算法：**

- 选择具有较少元素的集合。
- 为所选择的集合创建一个散列表。所使用的关键字和值可以相同。
- 现在扫描第二个数组并检查（$K$ 减去选择的元素）是否在散列表中。
- 如果存在，则返回这对元素。
- 否则继续，直至集合的末尾。

时间复杂度为 $O(\text{Max}(m, n))$，主要需要两次扫描。空间复杂度为 $O(\text{Min}(m, n))$，主要用于散列表的空间，可以选择元素少的集合创建散列表。

**问题 13**　给出一个算法，从一个给定字符串中删除指定的字符，这些字符由另一个字符串给出。

**解答：** 为了简单起见，假设不同字符的最大数量是 256。首先，创建一个辅助数组并初始化为 0。扫描要删除的字符，并且将这些字符值设置为 1，该值表明需要删除的字符。初始化后，扫描输入字符串并检查每个字符是否需要删除。如果当前字符已被标记，则直接跳到下一个字符，否则保持该字符在输入字符串中，继续这个过程直至输入字符串的末尾。所有操作如下所示。

```
void RemoveChars(char[] str, char[] removeTheseChars) {
        int srcInd, destInd;
        int auxi[256]; //辅助数组
        for(srcInd =0; srcInd<256; srcInd++)
                auxi[srcInd]=0;
        //对所有要移除的字符串置1
        srcInd =0;
        while(remove[srcInd]) {
                auxi[removeTheseChars[srcInd]]=1;
                srcInd++;
        }
```

```
            //复制不需要移除的字符
            srcInd=destInd=0;
            while(str[srcInd++]) {
                    if(!auxi[str[srcInd]])
                            str[destInd++]=str[srcInd];
            }
    }
```

时间复杂度为扫描要删除字符的时间＋扫描输入数组的时间＝$O(n)+O(m)\approx O(n)$。这里 $m$ 为要删除字符的长度，$n$ 为输入字符串的长度。

空间复杂度为 $O(m)$，要删除字符的长度。由于假设不同字符的最大个数是 256，所以可以将它视为常量，但需要记住，当处理多字节字符时，不同字符的总数是远远大于 256 的。

**问题 14** 给出算法，在字符串中寻找第一个没有重复出现的字符。例如，在字符串“abzddab”中，第一个没有重复的字符是“z”。

**解答：** 这个问题的解决方案很简单。对于给定字符串中的每一个字符，可以扫描剩余的字符串，看该字符是否出现在其中。如果没有出现在其中，则算法结束并返回该字符。如果该字符出现在剩余的字符串中，则访问下一个字符。

```
char FirstNonRepeatedChar( char[] str , int len) {
        int i, j, repeated = 0;
        for(i = 0; i < len; i++ )     {
                repeated = 0;
                for( j = 0; j < len; j++ )    {
                        if( i != j && str[i] == str[j] ) {
                                repeated = 1;
                                break;
                        }
                }
                if( repeated == 0 ) {
                // 找到第一个未重复出现的字符
                        return str[i];
                }
        }
        return '';
}
```

时间复杂度为 $O(n^2)$，由于有两个 for 循环。空间复杂度为 $O(1)$。

**问题 15** 是否能降低问题 14 的时间复杂度？

**解答：可以。**通过使用散列表可以降低时间复杂度。通过读取输入字符串中的所有字符来创建一个散列表并保持对每个字符出现次数的计数。在创建散列表后，可以读取整个散列表并查看哪个元素的计数值等于 1。这个方法使用了 $O(n)$空间，但也将时间复杂度降到 $O(n)$。

```
char FirstNonRepeatedCharUsinghash( char[] str, int len ) {
        int i, count[256]; //additional array
        for(i=0;i<len;++i)
                count[i] = 0;
        for(i=0;i<len;++i)
                count[str[i]]++;
        for(i=0; i<len; ++i) {
                if(count[str[i]]==1) {
                        System.out.println(str[i]);
```

```
                    break;
                }
            }
            if(i==len) System.out.println("No Non-repeated Characters");
            return 0;
    }
```

时间复杂度：用 $O(n)$ 创建散列表并用另一个 $O(n)$ 读取整个散列表，所以总时间是 $O(n)+O(n)=O(2n)\approx O(n)$。空间复杂度为 $O(n)$，用于保持计数值。

**问题 16**　给出一个算法，用于寻找给定字符串中第一个重复的字母。

**解答：**这个问题的解决方案几乎与问题 14 和问题 15 相似，但唯一不同的是，不用扫描散列表两次，只需要对散列表扫描一次就可以得出答案。这是因为在元素插入散列表中时可以判断该元素是否已经存在。如果它已经存在，那么只需要返回该字符。

```
char FirstRepeatedCharUsinghash(char[] str, int len) {
        int i, count[256]; //辅助数组
        for(i=0;i<len;++i)
                count[i] = 0;
        for(i=0; i<len; ++i) {
                if(count[str[i]]==1) {
                        System.out.println("%s",str[i]);
                        break;
                }
                else    count[str[i]]++;
        }
        if(i==len) System.out.println("No Repeated Characters");
        return 0;
}
```

时间复杂度：用 $O(n)$ 扫描和创建散列表，注意，对于这个问题，只需要一次扫描，所以总的时间复杂度是 $O(n)$。空间复杂度为 $O(n)$，用于保持计数值。

**问题 17**　给定具有 $n$ 个数的数组。给出一个算法，输出所有和为 $S$ 的数对。

**解答：**该问题与问题 12 相似。但不需要使用两个数组，仅使用一个数组即可。

**算法：**

- 逐一扫描输入数组的元素并创建一个散列表。使用的关键字和值可以相同。
- 在创建散列表后，开始扫描输入数组并检查（$S$ 减去选择的元素）是否在散列表中。
- 如果在散列表中，则返回这对元素。
- 否则继续，直至读取完数组的所有元素。

时间复杂度：用 $O(n)$ 创建散列表并用另一个 $O(n)$ 读取整个散列表，所以总时间是 $O(n)+O(n)=O(2n)\approx O(n)$。空间复杂度为 $O(n)$，用于保持计数值。

**问题 18**　是否有其他方法解决问题 17？

**解答：可以。**这个问题的替代解决方案涉及排序。首先将输入数组排序。排序后，使用两个指针，一个指向数组头，另一个指向数组尾。每次将两个索引对应的值相加并查看它们的和是否等于 $S$。如果和为 $S$，则输出这对。否则，如果和小于 $S$，则增加左指针，如果和大于 $S$，则减小右指针。

时间复杂度：排序时间＋扫描时间 $=O(n\log n)+O(n)\approx O(n\log n)$。空间复杂度为 $O(1)$。

**问题 19**　有一个包含数百万行数据的文件，其中只有两行是相同的，其余行都是唯

一的。文件的每一行都很长，以至于一行数据无法完全保存在内存中。在该文件中寻找相同行的最有效的解决方案是什么？

**解答：**完整的一行可能不能完全存放在主存中，但可以读取行的一部分并对这部分计算其散列值，然后再读取行的下一部分并计算散列值，在计算新散列值时同样使用先前计算的散列值。持续该过程直至为整行找到散列值。

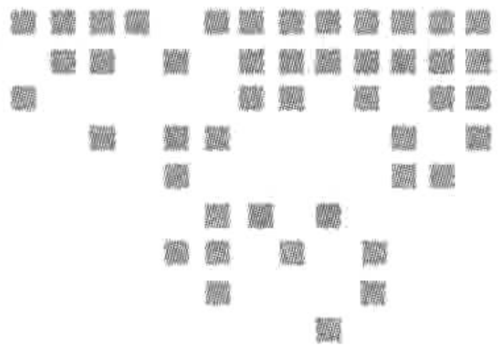
对每行执行这个操作并将所有的散列值存储在某个文件中(或维护这些散列的散列表)。在任何位置，如果得到相同的散列值，则一部分一部分地读取相应的行并进行比较。

**注意：**请参见第 11 章的相关问题。

**问题 20** 如果 $h$ 是散列函数，并且用来将 $n$ 个关键字散列到长度为 $s$ 的表中，其中 $n \leqslant s$，则涉及某个特定关键字 $X$ 的期望冲突数为：

(A) 小于 1　　(B) 小于 $n$　　(C) 小于 $s$　　(D) 小于 $\frac{n}{2}$

**解答：**A。



# 第 15 章 Chapter 15
# 字符串算法

## 15.1 引言

为了理解字符串算法的重要性，考虑在任意一个浏览器(如 Internet Explorer、Firefox 或 Google Chrome)中输入一个 URL(Uniform Resource Locator，统一资源定位符)时发生的情况。在键入 URL 的一些前缀字符后，可以看见浏览器会显示一个所有可能的 URL 列表。这意味着，浏览器执行了一些内部处理后，给用户一个可能匹配的 URL 列表。这种技术常称为自动完成。

类似地，考虑在(Windows 和 UNIX 的)命令行界面中输入一个目录名时发生的情况。在键入目录名的一些前缀字符后，如果按下 tab 键，那么会列出一个所有可能匹配的目录名列表。这是自动完成技术的又一个例子。

为了支持这类操作，需要一个数据结构来有效地存储字符串数据。本章将阐述可以有效实现字符串算法的数据结构。

本章先从字符串的基本问题开始讨论。给定一个字符串，如何查找一个子串(模式)？该问题称为字符串匹配问题。在讨论完各种字符串匹配算法后，将介绍可用于存储字符串的各种不同的数据结构。

## 15.2 字符串匹配算法

本节主要关注的问题是检查模式 $P$ 是否是另一个字符串 $T$($T$ 代表文本)的子串。因为要检查整个定长的字符串 $P$，所以有时这些算法称为精确字符串匹配算法。为了便于讨论，假设给定文本 $T$ 的长度为 $n$，要匹配的模式 $P$ 的长度为 $m$。即，$T$ 有从 $0\sim n-1$ 个字符(记为 $T[0..n-1]$)，$P$ 有从 $0\sim m-1$ 个字符(记为 $P[0..m-1]$)。

本章将从蛮力法开始，逐步介绍更好的算法。

- 蛮力法

- Robin-Karp 字符串匹配算法
- 基于有限自动机的字符串匹配算法
- KMP 算法
- Boyce-Moore 算法
- 后缀树

## 15.3 蛮力法

这个方法的思路是，对于文本 $T$ 中每个可能的位置，检查模式 $P$ 是否匹配。由于 $T$ 的长度为 $n$，所以有 $n-m+1$ 个可选的位置来比较。因为模式 $P$ 的长度为 $m$，所以 $T$ 的最后 $m-1$ 个位置无需检查。下面的算法用来搜索模式字符串 $P$ 在文本字符串 $T$ 中第一次出现的位置。

**算法：**

```
int BruteForceStringMatch (int T[], int n,  int P[], int m) {
        for (int i  = 0; i <=n - m; i++) {
                int j  = 0;
                while (j < m && P[j] == T[i + j])
                        j  = j + 1;
                if(j == m ) return i;
        }
        return -1;
}
```

时间复杂度为 $O((n-m+1)\times m) \approx O(n\times m)$。

空间复杂度为 $O(1)$。

## 15.4 Robin-Karp 字符串匹配算法

这个方法的思路为，使用散列技术代替对文本 $T$ 中每个可能位置进行检查的方法，即仅在模式 $P$ 的散列值与 $T$ 中 $m$ 个字符的散列值相等时才检查。初始化时，应用散列函数处理 $T$ 中的最前 $m$ 个字符，然后检查是否与 $P$ 的散列函数值相等。如果不相等，则从 $T$ 的下一个字符(第二个字符)开始的 $m$ 个字符再用散列函数处理。如果相等，则把 $T$ 中当前的 $m$ 个字符与 $P$ 比较。

**选择散列函数**

在 Robin-Karp 字符串匹配算法中，每一步都要对文本 $T$ 的 $m$ 个字符进行散列计算，因此需要一个有效的散列函数。如果每一步中散列函数的时间复杂度为 $O(m)$，那么总时间复杂度将为 $O(n\times m)$。因为算法首先进行散列函数运算，然后再做比较，所以这会使得整个算法的性能比蛮力法还差。

如何选择一个算法时间复杂度为 $O(1)$的散列函数来生成文本 $T$ 中 $m$ 个字符的散列值，成为 Robin-Karp 字符串匹配算法优化的目标，这将降低算法总的时间复杂度。如果散列函数选择不好(最差情况)，则 Robin-Karp 字符串匹配算法的时间复杂度为 $O((n-m+1)\times m) \approx O(n\times m)$。如果选择一个好的散列函数，则 Robin-Karp 字符串匹配算法的时间复杂度为 $O(m+n)$。下面阐述如何选择一个算法时间复杂度为 $O(1)$的字符散列函数。

为了简单起见，假设字符串中的字符都是整数，即 $T$ 中的所有字符都属于{0，1，

2，…，9}。由于所有字符都是整数，所以可以把 $m$ 个连续字符串看作十进制数。例如，串“61815”对应的十进制数是 61815。

按照上面的假设，模式 $P$ 也可看作一个十进制数，可以假设 $P$ 对应的十进制数是 $p$。对于给定文本 $T[0..n-1]$，令 $t(i)(i=0, 1, \cdots, n-m-1)$ 表示长度为 $m$ 的子串 $T[i..i+m-1]$ 对应的十进制数。那么，当且仅当 $T[i..i+m-1]$ 等于 $P[0..m-1]$ 时，$t(i)==p$。使用 Horner 规则，计算 $p$ 的时间复杂度为 $O(m)$，方法如下：

$$p = P[m-1] + 10(P[m-2] + 10(P[m-3] + \cdots + 10(P[1] + 10P[0])\cdots))$$

```
value = 0;
for (int i = 0; i <= m-1; i++) {
        value = value * 10;
        value = value + P[i];
}
```

计算所有 $t(i)(i=0, 1, \cdots, n-m-1)$ 的总时间复杂度为 $O(n)$。从 $T[0..m-1]$ 计算 $t(0)$ 值的方法与计算 $p$ 类似，所以复杂度也为 $O(m)$。而计算剩下的 $t(1)$，$t(2)$，…，$t(n-m-1)$ 的值，可以在常数时间内利用 $t(i)$ 来计算 $t(i+1)$。

$$t(i+1) = 10 \times (t(i) - 10^{m-1} \times T[i+1]) + T[i+m+1]$$

例如，如果 $T=$“123456”，$m=3$，$t(0)=123$，$t(1)=10\times(123-100\times1)+4=234$

逐步解释如下：

第一步：移除第一个数字，$123-100\times1=23$。

第二步：乘以 10 来移动第一步的结果，$23\times10=230$。

第三步：加上最后一个数字，$230+4=234$。

然后算法对 $t$[i]与 $p$ 进行比较。当 $t(i)=p$ 时，表示在文本 $T$ 中第 $i$ 个位置找到子串 $P$。

## 15.5　基于有限自动机的字符串匹配算法

这个方法的思路是：使用计算理论中的有限自动机概念来设计字符串匹配算法。在介绍算法前，首先给出有限自动机的定义。

### 1. 有限自动机

有限自动机(Finite Automata，FA)$F$ 是一个 5 元组：$F=(Q, q_0, A, \Sigma, \delta)$，其中，

- $Q$ 是一个非空的有限状态集合。
- $q_0$ 是一个起始状态，$q_0 \in Q$。
- $A$ 是一个接受状态的集合，$A \subseteq Q$。
- $\Sigma$ 是一个有限输入字母表。
- $\delta$ 是状态转移函数，根据给定的当前状态和输入，决定下一步状态。

### 2. 有限自动机的工作原理

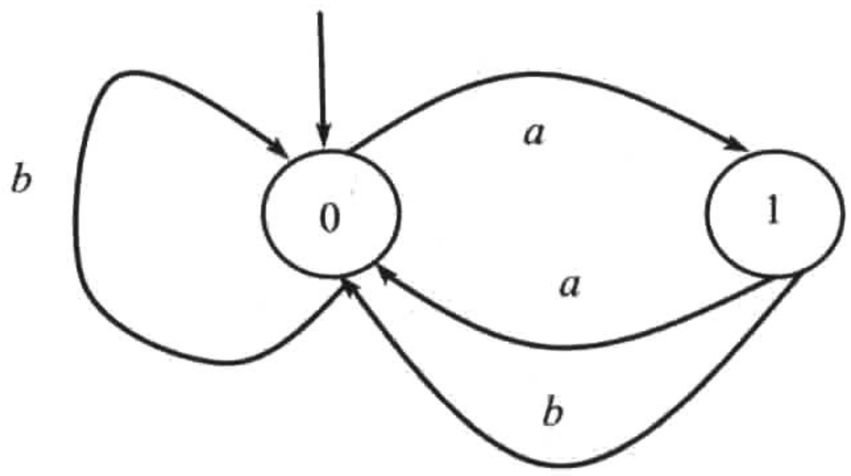
- 有限自动机 $F$ 从状态 $q_0$ 开始。
- 从字母表 $\Sigma$ 中每次读入一个字符。
- 如果有限自动机 $F$ 在状态 $q$ 时读入字符 $a$，则依据状态转移函数移动到状态 $\delta(q, a)$。
- 如果到达 $A$ 中状态的末尾，那么，我们说，$F$ 接受该字符串。
- 反之，则拒绝该字符串。

**例子：**假设 $Q=\{0, 1\}$，$q_0=0$，$A=\{1\}$，$\Sigma=\{a, b\}$。$\delta(q, a)$ 由下面的状态转移表/图定义。可接受的字符串一定是由奇数个 $a$ 字符结束。例如：abbaaa 是接受的字符

串，aa 是拒绝的字符串。

| 状态 | 输入 a | 输入 b |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 0 |

转移函数/表




**3. 构建有限自动机的重要说明**

在构建有限自动机时，首先从初始状态开始。如果已经匹配了模式串中的 $k$ 个字符，那么有限自动机将停在状态 $k$。如果文本串中的下一个字符等于模式串中的字符 $c$，那么前 $k+1$ 个字符都匹配，有限自动机移到状态 $k+1$。如果不相等，那么根据在字符 $c$ 之前匹配了多少个模式字符，有限自动机移动到状态 0，1，2，…，或者 $k$。

**4. 匹配算法**

下面给出基于有限自动机的字符串匹配算法。

- 对于给定的模式串 $P[0..m-1]$，首先需要构建有限自动机 $F$。
  - 状态集合是 $Q=\{0, 1, 2, \cdots, m\}$。
  - 开始状态是 0。
  - 唯一的接受状态是 $m$。
  - 如果字母表 $\Sigma$ 很大，则构建 $F$ 的时间开销也很大。
- 扫描文本串 $T[0..n-1]$，查找包含模式串 $P[0..m-1]$的所有位置。
- 这个字符串匹配算法是高效的：$\Theta(n)$。
  - 每个字符只检查一次。
  - 对每个字符的处理开销是常数时间。
  - 但是因为 $\delta$ 有 $O(m|\Sigma|)$项，所以计算 $\delta$（状态转移函数）的时间复杂度为 $O(m|\Sigma|)$。如果假设$|\Sigma|$是一常数，那么时间复杂度变为 $O(m)$。

**算法：**

```
//输入: 模式串 P[0..m-1]、δ和F。目标: 输出所有有效的移动
FiniteAutomataStringMatcher(int P[], int m, F, δ) {
        q = 0;
        for (i = 0; i < m; i++)
            q = δ(q,T[i]);
            if(q == m)
                System.out.println("Pattern occurs with shift:" + (i-m));
}
```

时间复杂度为 $O(m)$。

## 15.6 KMP 算法

与前面章节一样，假设被搜索的字符串为 $T$，需要匹配的模式串为 $P$。KMP 串匹配算法由 Knuth、Morris 和 Pratt 共同提出。匹配模式串的时间复杂度为 $O(n)$。该算法避免了 $T$ 中部分元素的再次比较，如果这些元素已经与模式串 $P$ 的部分元素比较过，这使

得算法的时间复杂度接近 $O(n)$。

算法使用一个表 $F$，$F$ 通常称为前缀函数、前缀表或失配函数。下面首先介绍如何构造表 $F$，然后阐述如何使用该表进行模式串匹配。

前缀函数 $F$ 中存储了模式串自身如何移动的信息。此信息用来避免模式串 $P$ 不必要的移动。也就是说，此表用以避免在字符串 $T$ 中的回溯。

**前缀表**

```
int F[]; //假设F是一个全局数组
void Prefix-Table(int P[], int m) {
        int i=1, j=0, F[0]=0;
        while(i<m) {
                if(P[i]==P[j]) {
                        F[i]=j+1;
                        i++;
                        j++;
                }
                else if(j>0)
                        j=F[j-1];
                else {  F[i]=0;
                        i++;
                }
        }
}
```

例如，假设模式串 $P$ 为 a b a b a c a。依据如下步骤构造前缀表 $F$。初始时：$m=$ length$[P]=7$，$F[0]=0$ 和 $F[1]=0$。

**步骤1**：$i=1$，$j=0$，$F[1]=0$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $P$ | a | b | a | b | a | c | a |
| $F$ | 0 | 0 | | | | | |

**步骤2**：$i=2$，$j=0$，$F[2]=1$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $P$ | a | b | a | b | a | c | a |
| $F$ | 0 | 0 | 1 | | | | |

**步骤3**：$i=3$，$j=1$，$F[3]=2$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $P$ | a | b | a | b | a | c | a |
| $F$ | 0 | 0 | 1 | 2 | | | |

**步骤4**：$i=4$，$j=2$，$F[4]=3$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $P$ | a | b | a | b | a | c | a |
| $F$ | 0 | 0 | 1 | 2 | 3 | | |

**步骤5**：$i=5$，$j=3$，$F[5]=1$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $P$ | a | b | a | b | a | c | a |
| $F$ | 0 | 0 | 1 | 2 | 3 | 0 | |

**步骤6**：$i=6$，$j=1$，$F[6]=1$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $P$ | a | b | a | b | a | c | a |
| $F$ | 0 | 0 | 1 | 2 | 3 | 0 | 1 |

至此，前缀表构造完成。

**匹配算法**

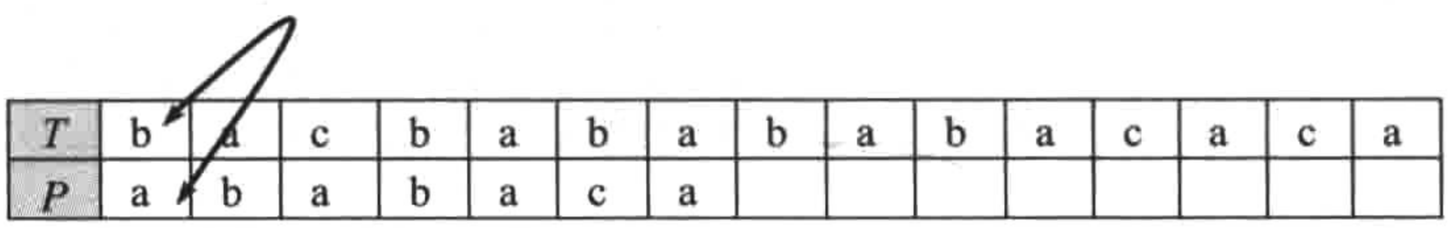
KMP 算法把模式串 $P$、文本串 $T$ 和前缀函数 $F$ 作为输入，查找文本 $T$ 中匹配 $P$ 的位置。

```
int KMP(char T[], int n, int P[], int m) {
        int i=0,j=0;
        Prefix-Table(P,m);
        while(i<n) {
            if(T[i]==P[j]) {
                if(j==m-1)
                    return i-j;
                else {   i++;
                    j++;
                }
            }
            else if(j>0)
                j=F[j-1];
            else     i++;
        }
        return -1;
}
```

时间复杂度为 $O(m+n)$，其中 $m$ 是模式串 $P$ 的长度，$n$ 是文本串 $T$ 的长度。空间复杂度为 $O(m)$。

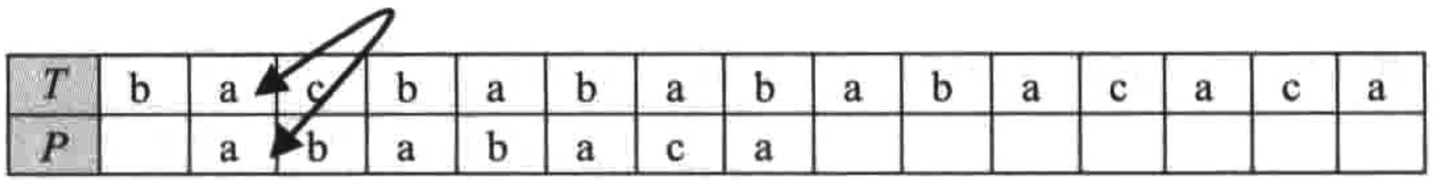
下面，举例说明 KMP 算法的流程。假设 $T=$ b a c b a b a b a c a c a，$P=$ a b a b a c a。由于前缀表已经构造完成，所以直接执行匹配算法。初始时，$T$ 的大小 $n$ 等于 15，$P$ 的大小 $m$ 等于 7。

**步骤 1**：$i=0$，$j=0$，比较 $P[0]$和 $T[0]$。$P[0]$不等于 $T[0]$。$P$ 向右移动一个位置。



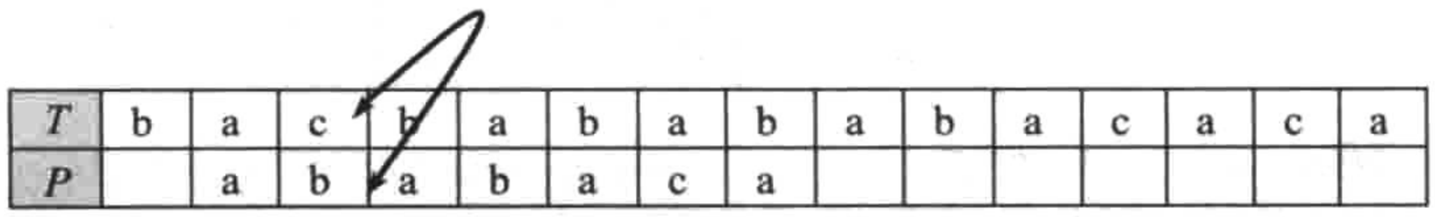
| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | a | b | a | b | a | c | a | | | | | | | | |

**步骤 2**：$i=1$，$j=0$，比较 $P[0]$和 $T[1]$。$P[0]$等于 $T[1]$。因为相等，所以 $P$ 不用移动。



| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | a | b | a | b | a | c | a | | | | | | | |

**步骤 3**：$i=2$，$j=1$，比较 $P[1]$和 $T[2]$。$P[1]$不等于 $T[2]$。$P$ 回溯，比较 $P[0]$和 $T[2]$。



| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | a | b | a | b | a | c | a | | | | | | | |

**步骤 4**：$i=3$，$j=0$，比较 $P[0]$和 $T[3]$。$P[0]$不等于 $T[3]$。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | a | b | a | b | a | c | a | | | | | |

**步骤 5**：$i = 4$，$j = 0$，比较 $P[0]$和 $T[4]$。$P[0]$等于 $T[4]$。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | | a | b | a | b | a | c | a | | | | |

**步骤 6**：$i = 5$，$j = 1$，比较 $P[1]$和 $T[5]$。$P[1]$等于 $T[5]$。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | | a | b | a | b | a | c | a | | | | |

**步骤 7**：$i = 6$，$j = 2$，比较 $P[2]$和 $T[6]$。$P[2]$等于 $T[6]$。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | | a | b | a | b | a | c | a | | | | |

**步骤 8**：$i = 7$，$j = 3$，比较 $P[3]$和 $T[7]$。$P[3]$等于 $T[7]$。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | | a | b | a | b | a | c | a | | | | |

**步骤 9**：$i = 8$，$j = 4$，比较 $P[4]$和 $T[8]$。$P[4]$等于 $T[8]$。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | | a | b | a | b | a | c | a | | | | |

**步骤 10**：$i = 9$，$j = 5$，比较 $P[5]$和 $T[9]$。$P[5]$不等于 $T[9]$。$P$ 回溯，比较$P[4]$和 $T[9]$，因为失配后，$j$ 等于$F[4]$等于 3。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | | a | b | a | b | a | c | a | | | | |

比较 $P[3]$和 $T[9]$。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | | | | a | b | a | b | a | c | a | | |

**步骤 11**：$i = 10$，$j = 4$，比较 $P[4]$和 $T[10]$。$P[4]$等于 $T[10]$。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | | | | a | b | a | b | a | c | a | | |

**步骤 12**：$i = 11$，$j = 5$，比较 $P[5]$和 $T[11]$。$P[5]$等于 $T[11]$。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | | | | a | b | a | b | a | c | a | | |

**步骤 13**：$i = 12$，$j = 6$，比较 $P[6]$和 $T[12]$。$P[6]$等于 $T[12]$。

| T | b | a | c | b | a | b | a | b | a | b | a | c | a | c | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | | | | | | | a | b | a | b | a | c | a | | |

模式串 $P$ 在文本串 $T$ 中完全匹配。在完全匹配前移动的总次数等于 $i-m = 13-7=6$。

**注意：**

- KMP 算法从左向右比较。
- KMP 算法需要一个时间和空间开销为 $O(m)$的预处理(前缀函数)过程。
- 匹配查找的时间复杂度为 $O(n+m)$(不依赖于字母表大小)。

## 15.7 Boyce-Moore 算法

与 KMP 算法类似，Boyce-Moore(简称 BM)算法也要进行一些预处理工作，它称为后缀函数(last function)。该算法从模式串最右边的字符开始，从右至左扫描模式串中的字符。在文本 $T$ 中查找模式串 $P$ 的过程中，处理失配的操作如下：假设不匹配，进行匹配的文本串的字符 $c$ 是 $T[i]$，模式串的字符是 $P[j]$。若 $c$ 在模式串 $P$ 中根本不存在，那么模式串 $P$ 可以整体移过 $T[i]$。否则，移动 $P$ 使 $P$ 中的字符 $c$ 与 $T[i]$对齐。这种技术通过相对于文本串来移动模式串，避免了不必要的比较。

处理后缀函数需要 $O(m+|\Sigma|)$的时间开销，实际的查找过程需要 $O(nm)$的时间开销。所以最差情况下，Boyer-Moore 算法的时间复杂度为 $O(nm+|\Sigma|)$。这表明 BM 算法在最差情况下的时间复杂度是平方级的，当 $n==m$ 时，与蛮力法的时间复杂度相同。

- 当字母表很大时，Boyer-Moore 算法非常快(相对于模式串的长度)。
- 对于小的字母表，Boyer-Moore 算法并不好。
- 对于二进制字符串，推荐使用 KMP 算法。
- 对于非常短的模式串，蛮力法更好。

## 15.8 存储字符串的数据结构

给定一个字符串集合(例如，词典中的所有单词)，可能需要在集合中查找某个单词。为了更快速地完成查找操作，需要有效的存储字符串的方法。可以使用下列数据结构来存储字符串集合。

- 散列表
- 二叉搜索(查找)树
- 键树
- 三叉搜索树

## 15.9 字符串的散列表实现

第 14 章已经介绍了散列表可以存储整数或字符串。这种情况下，关键字只能是字符

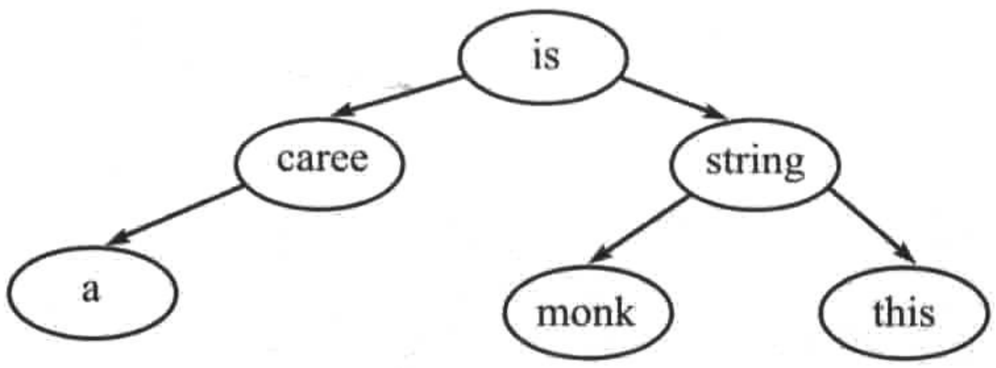
串。散列表实现的问题在于次序信息的丢失。这是因为当使用散列函数后，不知道它将关键字映射到哪里。因此有些查询会花费更多的时间。例如，若要查找所有以字母“K”开头的单词，如果使用散列表存储方式，那么需要扫描整个散列表。这是因为散列函数使用整个关键字来执行散列过程，并且不知道每个单词的存储位置。

## 15.10　字符串的二叉搜索树实现

在这种实现方式中，每个结点按照字母顺序存储字符串。因为字符串天生就是有序的：A 在 B 的前面，C 在 B 的后面等。利用单词之间有序性，可以使用二叉搜索树(Binary Search Tree，BST)来实现单词的存储和检索。例如，假设想使用 BST 存储下面的字符串：

this is a career monk string

对于给定的字符串，使用 BST 可以有多种表示方式。下图给出了一种可行的 BST。




### 二叉搜索树实现字符串的问题

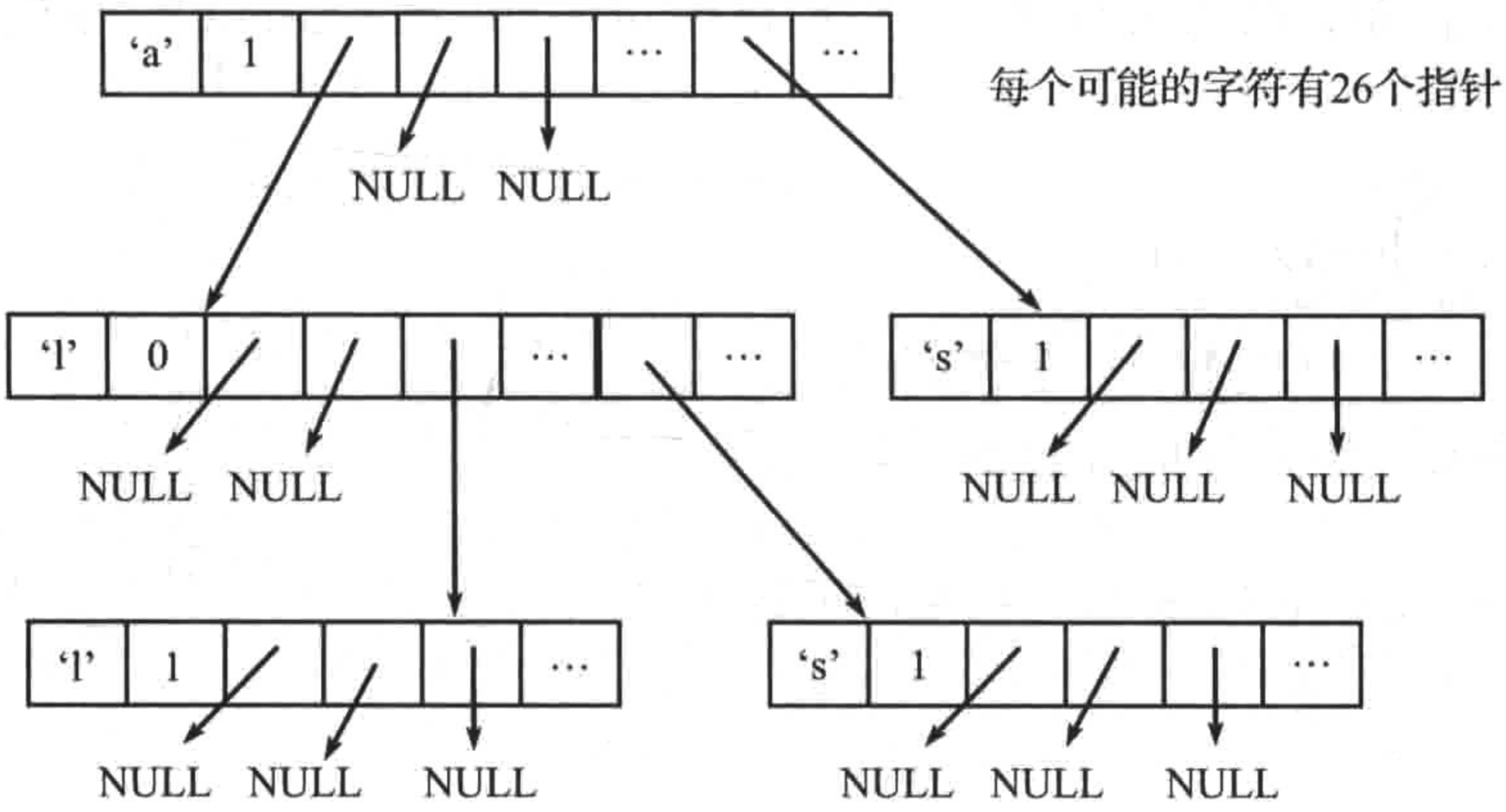
这种实现方式有好的存储效率。但是，其实现方式的缺点在于，对于每一个结点，搜索操作需要把结点数据与给定关键字进行完全匹配，这将导致搜索操作的时间复杂度增加。所以，字符串的 BST 实现方式在存储方面是高效的，但在时间效率方面是较差的。

## 15.11　键树

下面介绍一种实现字符串的替代方法，它可以有效减小查找的时间复杂度。这种方法叫作键树(又称为字典树)。

### 1. 什么是键树

键树是一棵树，树中每个结点所包含的指针数等于字母表的字符数。例如，假设所有字符串都是由“a”～“z”的英文字母组成，那么键树中的每个结点包含 26 个指针。若要存储字符串“a”、“all”、“als”和“as”，则这几个字符串的键树如下图所示。



### 2. 为什么要用键树

基于键树实现字符串的插入和查找的时间复杂度可以达到 $O(L)$（$L$ 是单个单词的长度），其性能比字符串的散列表和二叉搜索树实现都快。

### 3. 键树的结点声明

键树的基本元素 TrieNode 的数据结构如下所示。

```
public class TrieNode {
    char data;
    boolean is_End_Of_String;
    Collection<TrieNode> child;
}
```

键树抽象数据类型定义的结点数据结构 TrieNode 包括 data（字符型）、is_End_Of_String（布尔型）和孩子结点集合（TrieNode 集合）。抽象数据类型还包括一个函数 subNode(char)，这个函数需要一个字符作为参数，并返回字符类型的孩子结点，该孩子结点存储了字符作为参数。

```
public class TrieNode {
    char data;
    boolean is_End_Of_String;
    Collection<TrieNode> child;
    public TrieNode(char c){
        child = new LinkedList<TrieNode>();
        is_End_Of_String = false;
        data = c;
    }
    public TrieNode subNode(char c){
        if(child!=null){
            for(TrieNode eachChild:child){
                if(eachChild.data == c)
                    return eachChild;
            }
        }
        return null;
    }
}
```

### 4. 键树的声明

前面已给出键树结点的定义，接下来将给出键树类的伪代码。幸运的是，键树的数据结构实现非常简单，因为只有两个主要的操作 insert() 和 search()。下面将阐述这两个操作的实现思想。

```
public class Trie{
    private TrieNode  root;
    public Trie(){
        root = new TrieNode(' ');
    }
    public void InsertInTrie(String s){
        //详见下一节
    }
    public boolean SearchInTrie(String s){
        //详见下一节
    }
}
```

### 5. 在键树中插入一个字符串

要实现字符串的插入，需要从根结点开始，沿着相应的路径(从根结点开始的路径表示给定字符串的前缀字符)，当到达 NULL 指针时，只需要创建尾结点来保存给定字符串剩余的字符即可。插入算法的具体步骤如下。

1) 若输入字符串的长度等于 0，将根结点的 marker 标志设置为 true。

2) 若输入字符串的长度大于 0，对每个字符重复执行第 3 步和第 4 步。

3) 若字符出现在当前结点的孩子结点中，设置当前结点指针指向孩子结点。

4) 若字符没有出现在孩子结点中，那么插入一个新结点，设置当前结点指针指向新插入的结点。

5) 当处理完最后一个字符时，设置 marker 标志为 true。

时间复杂度为 $O(L)$，其中 $L$ 是插入字符串的长度。

**注意：** 在实际的词典实现中，可能需要更多的检查，如检查给定串是否已经在词典中。

### 6. 在键树中查找一个字符串

与插入操作类似，查找也需从根结点开始，沿着指针不断查找。查找操作的时间复杂度等于需查找字符串的长度。查找算法的具体步骤如下。

1) 对于字符串中的每个字符，检查是否有孩子结点的内容是该字符。

2) 如果该字符在所有孩子结点中不存在，返回 false。

3) 如果有孩子结点中保存了该字符，则重复第 1 步。

4) 重复上述三步，直至字符串中的每个字符都处理完。

5) 当处理完整个字符串时，如果当前结点的 marker 标志为 true，则返回 true，否则返回 false。

时间复杂度为 $O(L)$，其中 $L$ 是要查找字符串的长度。

### 7. 采用键树实现字符串的相关问题

键树实现字符串的主要缺点是保存字符串需要大量的空间开销。每个结点都有许多结点指针。许多情况下，每个结点的空间使用率很低。基于键树数据结构实现字符串的最终结论是，键树的相关操作很快，但需要大量内存空间来存储字符串。

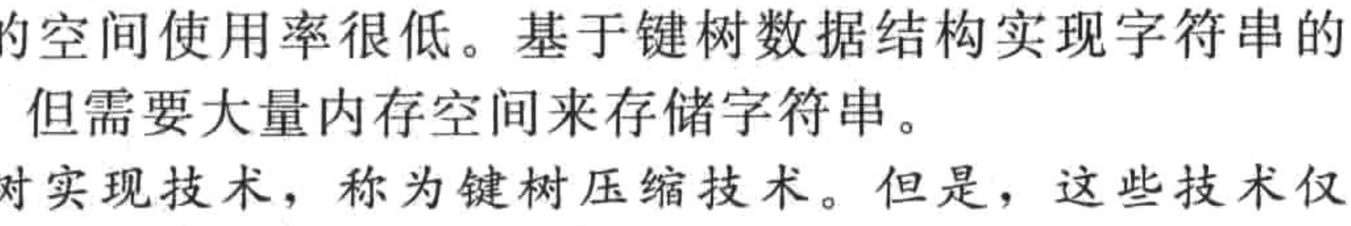
**注意：** 已经有了一些优化的键树实现技术，称为键树压缩技术。但是，这些技术仅能减小叶子结点的空间开销，但不能减小中间结点的空间开销。

## 15.12　三叉搜索树

这种实现方式最初是由 Jon Bentley 和 Sedgewick 提出的。三叉搜索树(Ternary Search Tree，TST)具有二叉搜索树和键树的优点，既有二叉搜索树节省空间的优点，又有键树查询速度快的优点。

### 1. 三叉搜索树的声明

```
public class TSTNode {
        char data;
        boolean is_End_Of_String;
        TSTNode left;
        TSTNode eq;
        TSTNode right;
}
```

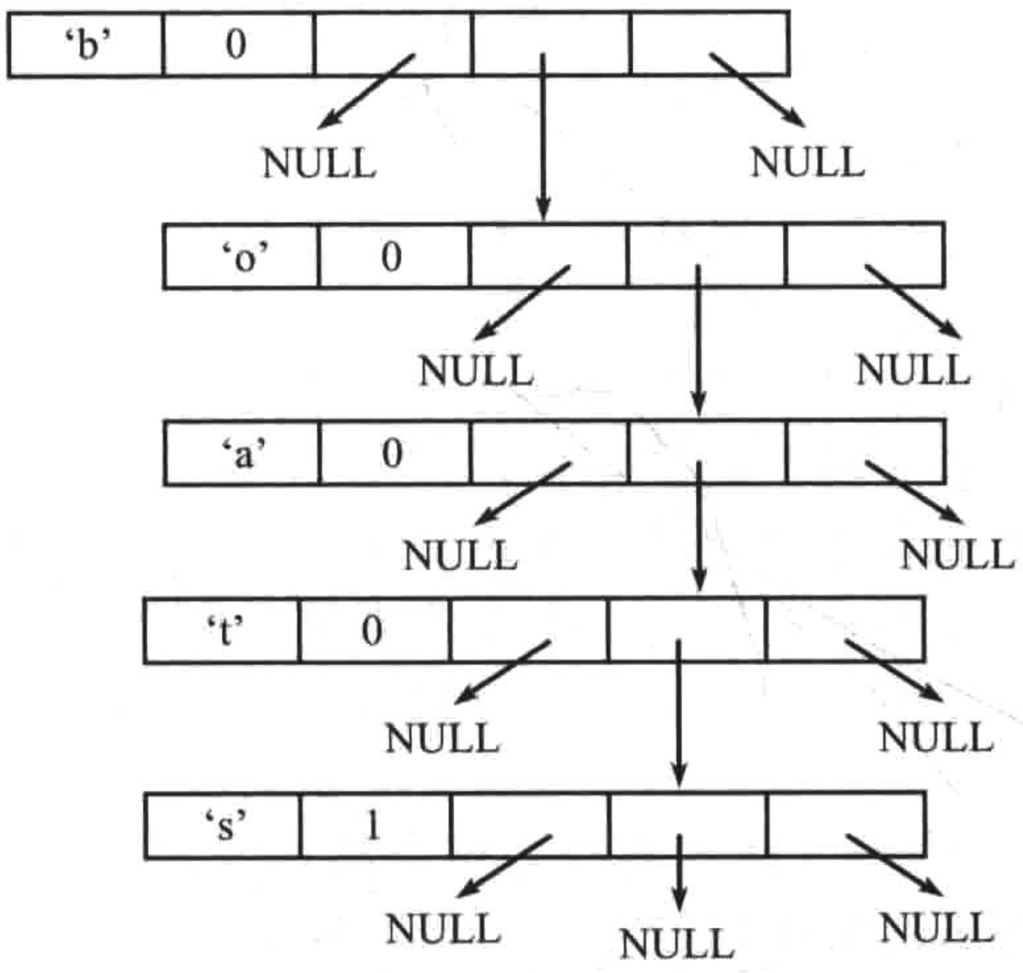


三叉搜索树中每个结点中有 3 个指针：

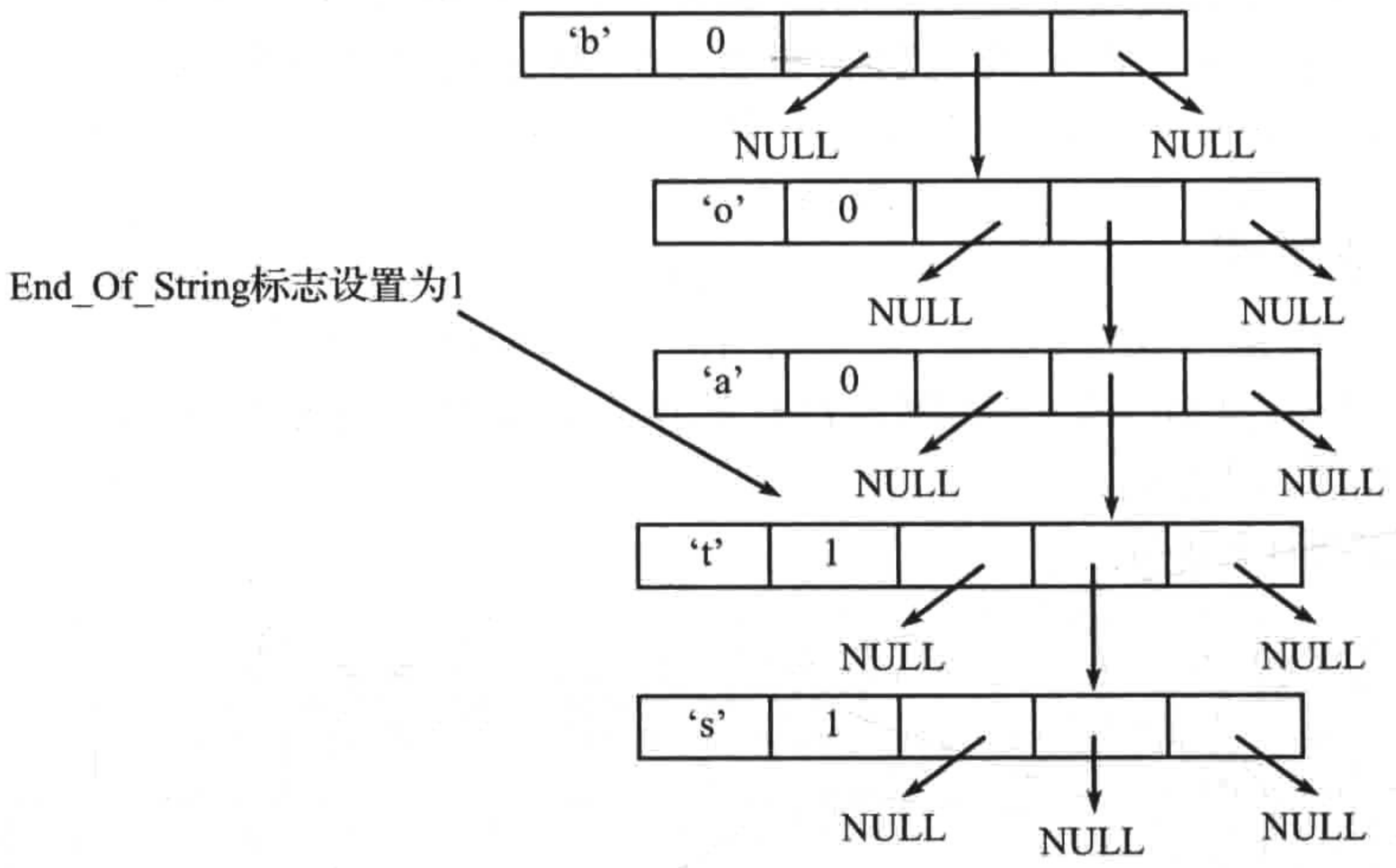
- 结点中的 left 指针所指向的子树包含了所有按字母顺序小于 data 的字符串。
- 结点中的 right 指针所指向的子树包含了所有按字母顺序大于 data 的字符串。
- 结点中的 eq 指针所指向的子树包含了所有按字母顺序等于 data 的字符串。这意味着，如果想要搜索一个字符串，且输入字符串的当前字符与三叉搜索树当前结点的 data 相等，那么需要继续在由 eq 指针所指向的子树中搜索输入串中的下一个字符。

**2. 在三叉搜索树中插入字符串**

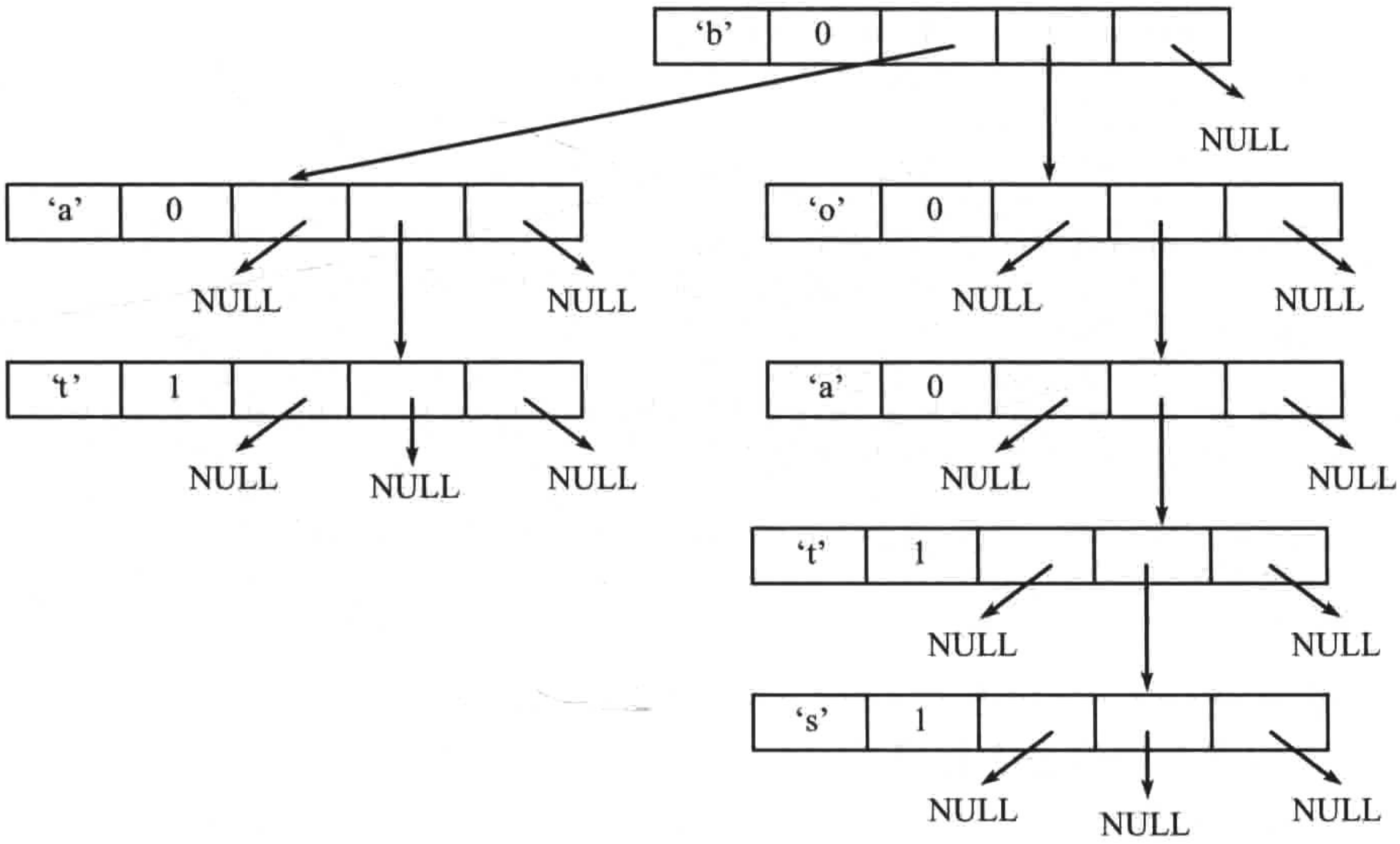
为了简单起见，假定在三叉搜索树中依次存入下列单词：boats、boat、bat 和 bats。首先，从字符串 boats 开始。



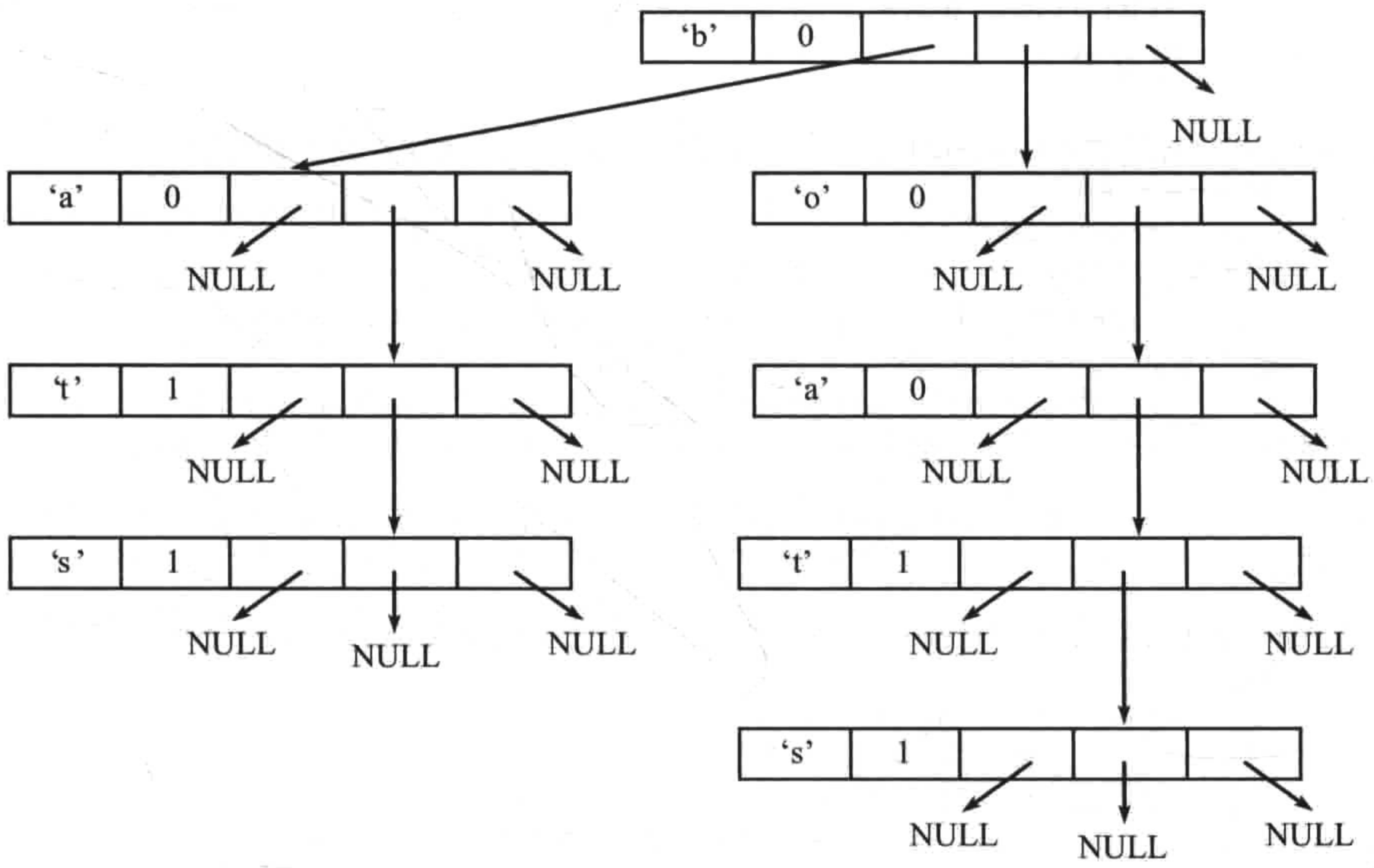

然后，插入字符串 boat，三叉搜索树的变化如下：与上图相比，唯一的改变是“t”结点的 is_End_Of_String 标记设置为 1。




然后，插入下一个字符串 bat。




然后，插入最后一个字符串 bats。




根据这些例子，结合二叉搜索树和键树的插入操作，可以写出如下的三叉搜索树插入算法。

```
//position的初始值为0
public TSTNode  InsertInTST(TSTNode  root, String word, int position) {
        if(root == null) {
                if(word.length() <= position) return root;
                root = new TSTNode();
                root.setData(word.charAt(position));
                root.setLeft(null);
                root.setEq(null);
                root.setRight(null);
```

```
                if(position == word.length()-1){
                        root.setMarker(true);
                    return root;
                }else return root.setEq(root, word, position+1);
        }
        if(word.charAt(position) < root.getData())
                root.setLeft( InsertInTST(root.getLeft(), word, position));
        else if(word.charAt(position) == root.getData()) {
                if(word.length <= position)
                        root.setEq(InsertInTST (root.getEq(), word, position + 1));
                else    root.setMarker(true);
        }
        else    root.setRight( InsertInTST(root.getRight(), word, position));
        return root;
}
```

时间复杂度为 $O(L)$，其中 $L$ 等于需插入的字符串的长度。

### 3. 在三叉搜索树中查找字符串

如果要查找已经插入三叉搜索树中的单词，那么必须遵循与二叉搜索树相同的规则。唯一的区别是，当出现匹配时，与二叉搜索树立即返回不同，三叉搜索树需要在 eq 所指向的子树中检查剩下的字符。同样，与二叉搜索树一样，可以设计三叉搜索树查找操作的递归和非递归实现算法。

```
boolean SearchInTSTRecursive(TSTNode  root, String word, int position) {
        if(root == null)
                return false;
        if(word.charAt(position) < root.getData())
                return SearchInTSTRecursive(root.getLeft(), word, position);
        else if(word.charAt(position) > root.getData())
                return SearchInTSTRecursive(root.getRight(), word, position);
        else {  if(root.getMarker() && word.length == position)
                        return true;
                return SearchInTSTRecursive(root.getEq(), word, position + 1);
        }
}
boolean SearchInTSTNon-Recursive(TSTNode  root, String word, int position) {
        while (root) {
                if(word.charAt(position) < root.getData())
                        root = root.getLeft();
                else if(word.charAt(position)  == root.getData()) {
                        if(root.getMarker() && word.length == position)
                                return true;
                        position ++;
                        root = root.getEq();
                }
                else    root = root.getRight();
        }
        return false;
}
```

时间复杂度为 $O(L)$，其中 $L$ 等于需查找的字符串的长度。

### 4. 输出三叉搜索树中的所有单词

若要输出三叉搜索树中的所有字符串，那么可以使用下面的算法。如果想有序地输

出所有字符串，可以对三叉搜索树进行中序遍历。

```
String word;
int i = 0;
void DisplayAllWords(TSTNode  root) {
        if(root == null)   return;
        DisplayAllWords(root.getLeft());
        word.setCharAt(i, root.getData());
        if(root.getMarker()) {
                System.out.println(word);
        }
        i++;
        DisplayAllWords(root.getEq());
        i--;
        DisplayAllWords(root.getRight());
}
```

### 5. 获取三叉搜索树中最长单词的长度

算法思想与求二叉搜索树高度的算法类似，伪代码如下：

```
int MaxLengthOfLargestWordInTST(TSTNode  root) {
        if(root == null)   return 0;
        return Math.max(MaxLengthWordInTST(root.getLeft()),MaxLengthWordInTST(root.getEq())+1,
                MaxLengthWordInTST(root.getRight()));
}
```

## 15.13　二叉搜索树、键树和三叉搜索树的比较

- 散列表和二叉搜索树实现在每个结点存储一个完整的字符串。所以，在搜索时需花费较多的时间。但是空间利用率高。
- 三叉搜索树的内存空间可以动态地扩大和缩小，而散列表仅基于负载因子调整大小。
- 三叉搜索树支持部分搜索，而二叉搜索树和散列表不支持。
- 三叉搜索树可以按序输出字符串，但是在散列表中不能实现有序字符串输出。
- 键树执行查找操作的速度非常快，但是需要花费巨大的内存来存储字符串。
- 三叉搜索树具有二叉搜索树和键树的优点：既有二叉搜索树节省空间的优点，又有键树查询速度快的优点。

## 15.14　后缀树

后缀树是一种实现字符串的重要数据结构，它能非常快速地实现查询功能。但是，后缀树需要一些预处理和构建过程。虽然后缀树的构建比较复杂，但是它能在线性时间内求解与字符串相关的许多问题。

**注意：**一棵后缀树只表示一个字符串，而散列表、二叉搜索树和三叉搜索树则可以存储一个字符串集合。这意味着，后缀树只能完成对一个字符串的查询问题。

下面给出用于这种实现方式的相关术语。

### 1. 前缀和后缀

对于给定的字符串 $T=T_1T_2\cdots T_n$，$T$ 的前缀是字符串 $T_1\cdots T_i$，$i$ 的取值范围为 $1\sim n$。例如，若 $T=$banana，那么 $T$ 的前缀字符串有：b、ba、ban、bana、banan、banana。类似地，对于给定的字符串 $T=T_1T_2\cdots T_n$，$T$ 的后缀是字符串 $T_i\cdots T_n$，$i$ 的取值范围为 $n\sim 1$。

例如，若 $T$=banana，那么 $T$ 的后缀字符串有：a、na、ana、nana、anana、banana。

**2. 评论**

从上面的例子中可以很容易地看出，给定文本 $T$ 和模式 $P$，精确字符串匹配问题也可以定义为：

- 查找 $T$ 的后缀，$P$ 是这个后缀的前缀，或
- 查找 $T$ 的前缀，$P$ 是这个前缀的后缀。

**例子：**若文本 $T$=accbkkbac，模式 $P$=kkb。那么 $P$ 是 $T$ 的后缀 kkbac 的前缀，也是 $T$ 的前缀 accbkkb 的后缀。

**3. 什么是后缀树**

简单来说，文本 $T$ 的后缀树是一种包含了 $T$ 的所有后缀的类似键树的数据结构。后缀树的定义如下：$n$ 个字符的串 $T[1..n]$的后缀树是一个具有如下属性的有根树：

- 后缀树中包含 $n$ 个叶子结点，所有叶子结点从 1～$n$ 编号。
- 每一个中间结点，除了根结点以外，至少有 2 个孩子结点。
- 树中的每条边都用 $T$ 的一个非空子串来标识。
- 出自同一结点的任意两条(指向孩子结点)边的标识不会以相同的字符开始。
- 从根结点到所有叶子结点的路径标识了 $T$ 的所有后缀。

**4. 构建后缀树**

**算法：**

1) 令集合 $S$ 表示 $T$ 的所有后缀。在每个后缀的后面添加字符 $。

2) 按照首字母把 $S$ 中的后缀排序。

3) 对于每个组 $S_c(c\in\Sigma)$，

(i) 如果组 $S_c$ 中只有一个元素，那么创建一个叶子结点。

(ii) 否则，找到组 $S_c$ 中所有后缀的最长公共前缀，创建一个中间结点，然后递归地执行第 2 步，同时将 $S$ 更新为 $S_c$ 中去掉最长公共前缀后剩下的后缀。

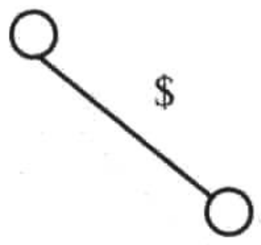
为了更好地理解后缀树，下面给出一个样例。假设给定文本 $T$=tatat。对这个字符串的每个后缀进行编号。

| 编号 | 后缀 | 编号 | 后缀 |
|---|---|---|---|
| 1 | $ | 4 | tat $ |
| 2 | t $ | 5 | atat $ |
| 3 | at $ | 6 | tatat $ |

首先，基于后缀串中的首字符对后缀串进行排序。

| 编号 | 后缀 | |
|---|---|---|
| 1 | $ | 基于 a 的组 $S_1$ |
| 3 | at $ | 基于 a 的组 $S_2$ |
| 5 | atat $ | |
| 2 | t $ | 基于 t 的组 $S_3$ |
| 4 | tat $ | |
| 6 | tatat $ | |

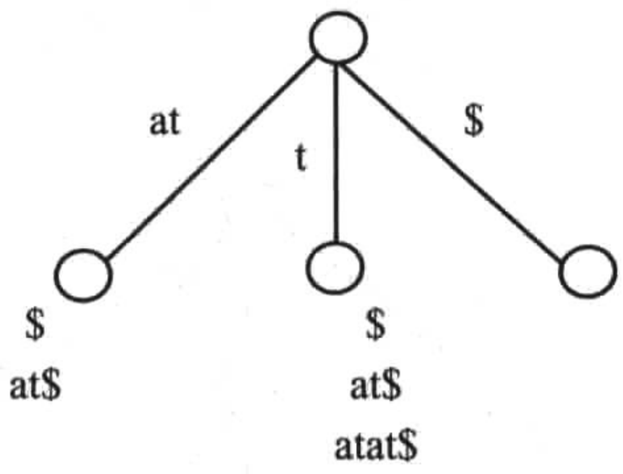
在 3 个组中，第一个组只有一个元素。因此，按照算法创建一个叶子结点，如下图所示。




对于 $S_2$和 $S_3$(因为它们中不止有一个元素)，需要找到每组中的最长前缀，结果如下表所示。

| 组 | 组的编号 | 每组中的最长前缀 |
|---|---|---|
| $S_2$ | 3、5 | at |
| $S_3$ | 2、4、6 | t |

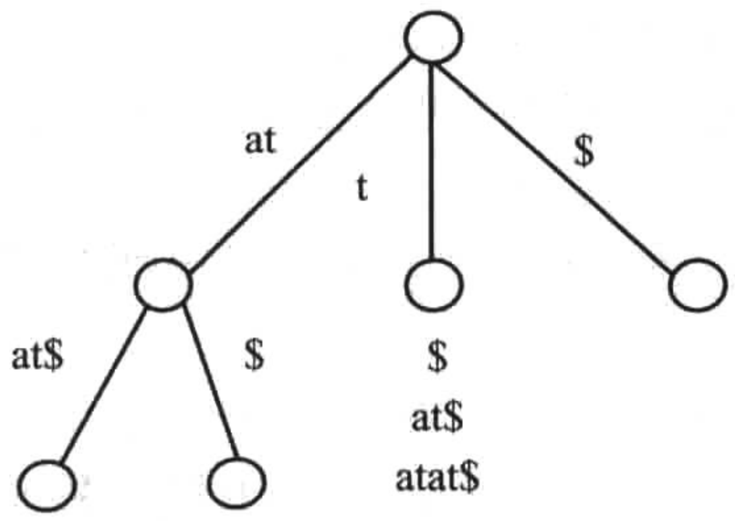
对于 $S_2$和 $S_3$，创建中间结点，边的值为该组的最长公共前缀。




然后，删除 $S_2$和 $S_3$中每个元素的最长公共前缀。

| 组 | 组的编号 | 每组中的最长前缀 | 删除每个元素的最长公共前缀 |
|---|---|---|---|
| $S_2$ | 3、5 | at | $、at$ |
| $S_3$ | 2、4、6 | t | $、at$、$、atat$ |

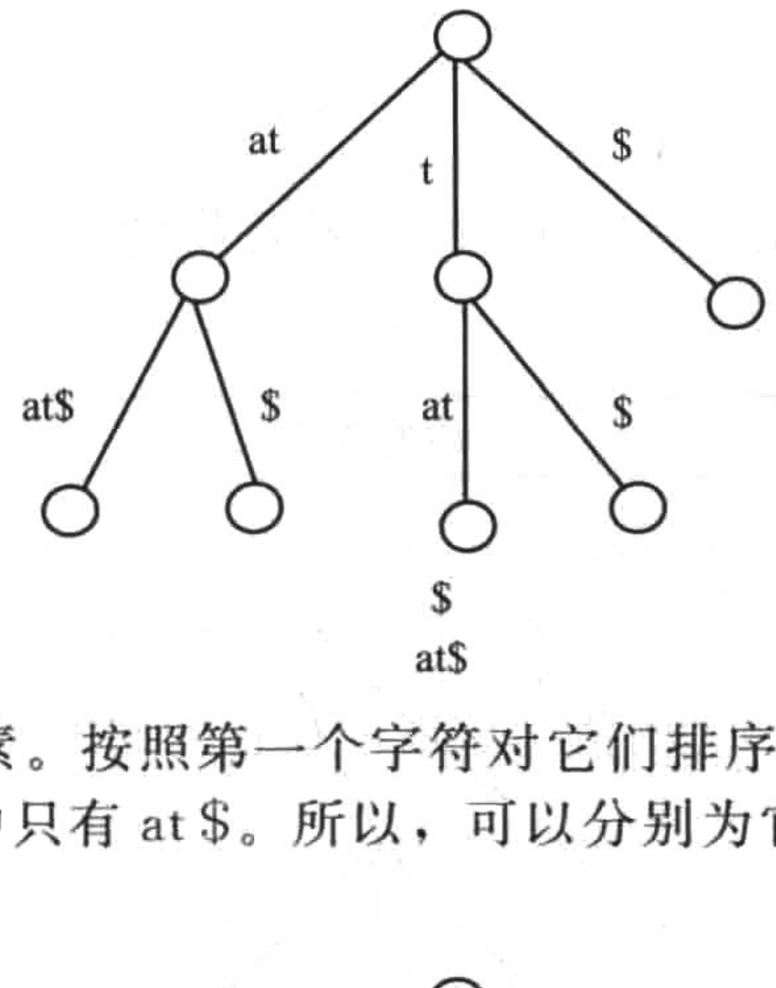
然后，用上述方法递归处理 $S_2$和 $S_3$。首先处理 $S_2$，按照第一个字符对 $S_2$中的元素排序，显而易见，第一个组中只包含一个元素 $，第二个组中也只包含一个元素 at$。因为这两个组都各自只有一个元素，所以可直接分别为它们创建叶子结点。



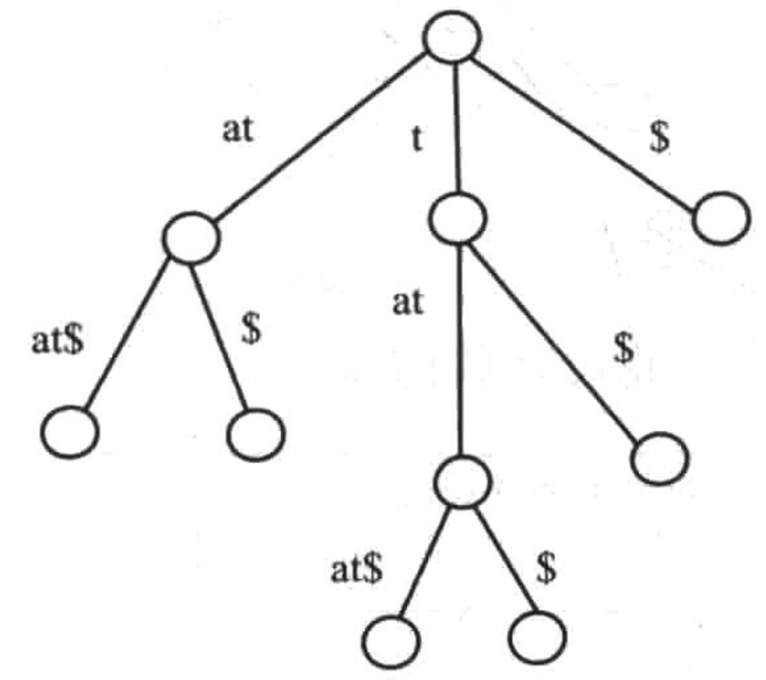

接着，处理组 $S_3$，按照同样的步骤，首先根据第一个字符对 $S_3$中的元素排序，显而易见，$S_3$中的第一个组只包含一个元素，即 $。而 $S_3$中的第二个组则需要删除最长前缀。

| 组 | 组的编号 | 每组中的最长前缀 | 删除每个元素的最长公共前缀 |
|---|---|---|---|
| $S_3$ | 4、6 | at | $、at$ |

$S_3$的第二个组中的两个元素分别变为 $和 at $。对于 $S_3$的第一个组中的唯一一个 $，直接添加一个叶子结点。新树如下图所示。




现在，$S_3$只包含两个元素。按照第一个字符对它们排序，显而易见它们被分为两组，第一组只有元素 $，第二组中只有 at $。所以，可以分别为它们创建叶子结点。新树如下图所示。




因为所有元素都已处理，所以字符串 $T$=tatat 的后缀树构建过程结束。

使用上述算法构建后缀树的时间复杂度为 $O(n^2)$，$n$ 是输入字符串的长度。因为有 $n$ 个不同的后缀，所以最长后缀的长度为 $n$，第二长的长度为 $n-1$，以此类推。

**注意：**

- 存在时间复杂度为 $O(n)$的后缀树构建算法。
- 为了降低复杂度，树的分支(的值)可以使用编号代替字符串。

### 5. 后缀树的应用

下面所有与字符串相关的问题(不限于这些)可以用后缀树非常有效地解决(算法请参见 15.15 节)。

- **精确字符串匹配算法**：给定文本 $T$ 和模式 $P$，如何判定 $P$ 是否在 $T$ 中？
- **最长重复子串**：给定文本 $T$，如何找到 $T$ 中最长的重复子串？
- **最长回文**：给定文本 $T$，如何找到 $T$ 中最长的回文？
- **最长公共子串**：给定两个字符串，如何找到最长的公共子串？
- **最长公共前缀**：给定两个字符串 $X[i..n]$和 $Y[j..m]$，如何找到它们的最长公共前缀？
- 如何在文本 $T$ 中寻找一个正则表达式？
- 给定文本 $T$ 和模式 $P$，如何找到模式 $P$ 在文本 $T$ 中第一次出现的位置？

## 15.15　字符串的相关问题

**问题 1**　给定一段文字，设计一个算法来寻找出现次数最多的词。若段落在不断向下滚动(一些词将消失，一些词仍然还有，一些新词将出现)，寻找当前出现次数最多的词。因此这个算法应该是动态的。

**解答：**对于这个问题，可以结合优先队列和键树来求解。首先创建一棵键树，将出现的每个单词作为叶子结点插入键树中。同时每个结点还包含一个指针，指针指向该单词在堆(优先队列)中创建的结点。堆中的结点包括一个记录单词出现次数的 counter 变量和一个指向键树中叶子结点的指针。因为键树中的结点已经保存了单词，所以不需要保存该单词两次。每当出现一个新单词时，就在键树中查找，若查找成功，那么堆中对应结点的出现次数的变量值加 1，并进行堆化操作，保证堆结构的正确性。堆化操作可以保证在任何时候，堆的根结点是出现次数最多的词。当段落滚动时，一个单词消失，堆中对应单词的 counter 变量值减 1。若此时 counter 值仍然大于 0，则需要进行堆化操作以便适应值的修改。若 counter 的值等于 0，则将结点从堆和键树中删除。

**问题 2**　如何寻找两个字符串的最长公共子串？

**解答：**假设两个字符串分别用 $T_1$ 和 $T_2$ 表示。可以通过为 $T_1$ 和 $T_2$ 构建一棵广义后缀树来寻找 $T_1$ 和 $T_2$ 的最长公共子串。这需要为两个字符串构建一棵后缀树。需要对每个结点进行标记，表明它是 $T_1$ 的后缀还是 $T_2$ 串的后缀，还是 $T_1$ 和 $T_2$ 的后缀。这需要为两个字符串使用不同的标记符号(例如，第一个串用 $，第二个串用#)。公共后缀树构建完后，$T_1$ 和 $T_2$ 的标记最深的结点表示最长公共子串。

**其他方法：**为字符串 $T_1\$T_2$# 构建一棵后缀树。这等价于为两个字符串构建一棵公共后缀树。

时间复杂度为 $O(m+n)$，其中 $m$ 和 $n$ 分别是串 $T_1$ 和 $T_2$ 的长度。

**问题 3　最长回文：**给定一个文本 $T$，如何寻找 $T$ 中最长回文所对应的子串？

**解答：**寻找最长回文 $T[1..n]$ 算法的时间复杂度为 $O(n)$。算法思路为，首先为串 $T\$\text{reverse}(T)$# 构建一棵后缀树，或者为 $T$ 和 reverse($T$) 构建一棵广义后缀树。然后，在构建的后缀树中，寻找被 $和# 标记最深的结点。这与寻找最长公共子串类似。

**问题 4**　给定一个字符串(单词)，设计一个算法寻找其在词典中的下一个单词。

**解答：**假定使用键树来存储词典中的单词。为了寻找在键树中的下一个单词，可以使用下面这个简单方法。从最右边的字符开始，一个一个地递增字符。一旦到达字符 Z，则移动到左边的下一个字符。每次递增时，需要检查增加字符的单词是否在词典中。若在，则返回这个单词，否则继续递增。如果使用三叉搜索树，那么还可以找到当前单词的中序后继。

**问题 5**　给出一个反转字符串的算法。

**解答：**

```
//若字符串str可编辑
class ReverseString {
        public String ReversingString(String str) {
                char temp, start = 0, end = str.length();
                for (start = 0; start < end; start++, end--) {
                        temp = str.charAt(start);
```

```
                    str.setCharAt(start, str.charAt(end));
                    str.setCharAt(end, temp);
                }
                return str;
        }
}
```

时间复杂度为 $O(n)$，其中 $n$ 是给定字符串的长度。空间复杂度为 $O(n)$。

**问题 6** 如果字符串不可编辑，可以反转字符串吗？

**解答：可以。**如果字符串不可编辑，那么需要创建一个数组，并返回指向该数组的指针。

```
//若字符串str是常量字符串（不可编辑）
class ReverseString {
        public static String reverseIt(String str) {
                int i, len = str.length();
                StringBuffer dest = new StringBuffer(len);
                for (i = (len - 1); i >= 0; i--)
                        dest.append(str.charAt(i));
                return dest.toString();
        }
}
```

时间复杂度为 $O\left(\frac{n}{2}\right)\approx O(n)$，其中 $n$ 是给定字符串的长度。空间复杂度为 $O(n)$。

**问题 7** 能否不使用任何临时变量实现字符串的反转？

**解答：可以。**可以使用 XOR 逻辑运算来实现变量值的交换。

```
String ReversingString(String str) {
        int end= str.length()-1;
        int start = 0;
        while( start<end ) {
                str.setCharAt(start,  str.charAt(start)^ str.charAt(end));
                str.setCharAt(end,   str.charAt(end) ^  str.charAt(start));
                str.setCharAt(start,  str.charAt(start) ^ str.charAt(end));
                ++start;
                --end;
        }
        return str;
}
```

时间复杂度为 $O\left(\frac{n}{2}\right)\approx O(n)$，其中 $n$ 是给定字符串的长度。空间复杂度为 $O(n)$。

**问题 8** 给出一个反转给定句子中单词的算法。

**例子：**输入：“This is a Career Monk String”，输出：“String Monk Career a is This”。

**解答：**从句子开头开始，不断反转各个单词。在下面的实现方法中，假设“ ”（空格）是句中单词的分隔符。

```
public class ReverseSentence {
        public static void ReversingSentence(String Line) {
                //指定" "空格为分隔符
                StringTokenizer st = new StringTokenizer(strLine, " ");
                String strReversedLine = "";
                while(st.hasMoreTokens()) {
                        strReversedLine = st.nextToken() + " " + strReversedLine;
                }
```

```
                System.out.println("Reversed string by word is : " + strReversedLine);
        }
}
```

时间复杂度为 $O(2n)\approx O(n)$，其中 $n$ 是给定字符串的长度。空间复杂度为 $O(1)$。

**问题 9　字符串的排列(回文构词法)：**给出一个算法，输出一个字符串中字符的所有可能的排列。与组合不同，两个包含相同字符但不同字符次序的排列被认为是不同的。为了简单起见，假设每个出现的重复字符都是不同的字符。即，如果输入是“aaa”，输出应是重复“aaa”6 次的字符串。可以用任何次序输出排列。

**解答：**解决方案是为长度为 $n$ 的字符串生成 $n!$ 个字符串，其中 $n$ 是输入字符串的长度。

```
public class Permutations {
        public  static void permutationsInOrder(String s) {
                permutationsInOrder("", s);
        }
        private static void permutationsInOrder(String prefix, String s) {
                int len = s.length();
                if (len == 0)
                        System.out.println(prefix);
                else {    for (int i = 0; i < len; i++)
                                permutationsInOrder(prefix + s.charAt(i), s.substring(0, i) +
                                s.substring(i+1, len));
                }
        }
        public static void permutationsNotInOrder(String s) {
                int len = s.length();
                char[] a = new char[len];
                for (int i = 0; i < len; i++)
                        a[i] = s.charAt(i);
                permutationsNotInOrder(a, len);
}
private static void permutationsNotInOrder(char[] a, int n) {
        if (n == 1) {
                System.out.println(a);
                return;
        }
        for (int i = 0; i < n; i++) {
                swap(a, i, n-1);
                permutationsNotInOrder(a, n-1);
                swap(a, i, n-1);
        }
}
        // 交换索引i和j对应的字符
        private static void swap(char[] a, int i, int j) {
                char c;
                c = a[i]; a[i] = a[j]; a[j] = c;
        }
}
```

**问题 10　字符串的组合：**与排列不同，一个字符串的两个组合如果包含相同的字符则认为是相同的，其中忽略字符的次序。给出一个可以输出一个字符串中的字符所有可能组合的算法。例如，对于输入串“abc”、“ac”和“ab”是不同的组合，而“ab”和“ba”是相同的组合。

**解答：**解决方案是从 1～$n$ 的每个长度 $r$ 均生成 $n!/r!(n-r)!$ 个字符串，其中 $n$ 是输入字符串的长度。

**算法：**

对于输入字符中的每个字符，

a. 将当前字符放入输出字符串中，并输出。

b. 如果存在剩余的字符，用剩余的字符继续组合串。

```
public class Combinations {
        // 输出串str所有的字符子集
        public static void CombinationsOne(String str) {
                CombinationsOne("", str);
        }
        // 给定前缀，输出所有剩余字符的子集
        private static void CombinationsOne(String prefix, String str) {
                if (str.length() > 0) {
                        System.out.println(prefix + str.charAt(0));
                        CombinationsOne(prefix + str.charAt(0), str.substring(1));
                        CombinationsOne(prefix,  str.substring(1));
                }
        }
        // 替代方案
        public static void CombinationsTwo(String str) {
                CombinationsTwo ("", s);
        }
        private static void CombinationsTwo(String prefix, String str) {
                System.out.println(prefix);
                for (int i = 0; i < str.length(); i++)
                        CombinationsTwo(prefix + str.charAt(i), str.substring(i + 1));
        }
}
```

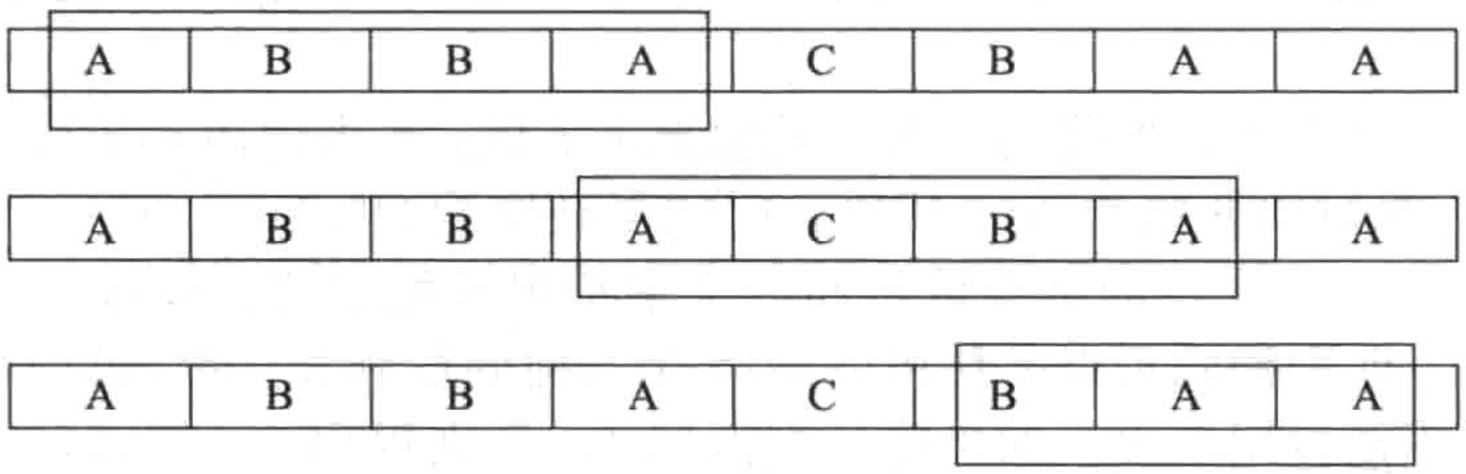
**问题 11**　给定串“ABCCBCBA”。设计一个算法，可以不断移除相同的相邻字符。例如，ABCCBCBA→ABBCBA→ACBA

**解答：**首先，需要检查给定字符串是否存在相同的字符对。如果存在，则删除它。然后，检查下一个和前一个字符。继续删除直至到达串首或串尾或找不到相同字符对时就停止。

```
void RremoveAdjacentPairs(char[] str, int len) {
        int j = 0;
        for (int i=1; i <= len; i++) {
                while ((str[i] == str[j]) && (j >= 0)){          //删除对
                        i++;
                        j--;
                }
                str[++j] = str[i];
        }
        return;
}
```

**问题 12**　给定一个字符集 CHARS 和一个输入字符串 INPUT，设计一个时间复杂度为 $O(n)$的算法，在 INPUT 中寻找包含 CHARS 所有字符的最小窗口。例如，INPUT＝ABBACBAA，CHARS＝AAB 的最小窗口是 BAA。

**解答：**这个算法的思路是基于滑动窗口方法。首先从串首开始，向右扩大窗口。一旦窗口中包含了所有需要的字符，就尝试尽可能向右滑动整个窗口，并保证窗口中包含所有需要覆盖的字符。如果当前窗口的长度小于到目前为止找到窗口的最小长度，那么更新最小长度。例如，如果输入串是 ABBACBAA，最小窗口应该覆盖字符集 AAB，那么滑动窗口的移动过程应该如下所示。




**算法：**

Input 是给定的字符串，chars 是需要寻找的字符集合。

1）定义一个长度为 256 的整数数组 shouldfind[]。该数组中的第 $i$ 个元素存储为寻找 ASCII 码值为 $i$ 的元素所需要的次数 count。

2）定义另一个数组，它记录寻找 256 个元素的次数，它记录到目前为止需要覆盖的元素被找到的次数 count。

3）count $\leqslant 0$。

4）循环处理 input[$i$]。

a）若 input[$i$]中的元素没有找到，继续处理下一个。

b）若 input[$i$]中的元素是需要被覆盖的，count 值加 1。

c）若 count 等于 chars 数组的长度，向右尽可能地滑动窗口。

d）若当前窗口的长度小于到目前为止找到的最小窗口长度，则更新最小长度。

```
#define MAX 256
void MinLengthWindow(char[] input, int iplen, char[] chars, int charlen) {
        int shouldfind[MAX] = {0,}, hasfound[MAX] = {0,};
        int j=0, cnt = 0, start, finish, minwindow = INT_MAX;
        for (int i=0; i< charlen; i++)
                shouldfind[chars[i]] += 1;

        start = 0;
        finish = iplen;
        for (int i=0; i< iplen; i++) {
if(!shouldfind[input[i]])
        continue;
hasfound[input[i]] += 1;

if(shouldfind[input[i]] >= hasfound[input[i]])
        cnt++;
if(cnt == charlen) {
        while (shouldfind[input[j]] == 0 || hasfound[input[j]] >
        shouldfind[input[j]])          {
                if(hasfound[input[j]] > shouldfind[input[j]])
                        hasfound[input[j]]--;
                j++;
        }
```

```
                        if(minwindow > (i - j +1)) {
                            minwindow = i - j +1;
                            finish = i;
                            start = j;
                        }
                    }
                }
            System.out.println("Start:" + start + " and Finish: " + finish);
    }
```

复杂度分析：根据上面的代码，在最差情况下 $i$ 和 $j$ 最多遍历 $n$ 步（$n$ 是 INPUT 的大小），两者相加，总时间开销为 $2n$。所以，时间复杂度为 $O(n)$。

**问题 13** 给定一个二维的字符数组和一个字符串匹配模式。设计一个在二维数组中查找模式的算法。匹配模式可以是任何方向（8 个方向的所有邻居都可以），但是在匹配过程中相同的字符不能使用两次。如果找到则返回 1，否则返回 0。例如，在下面的矩阵中寻找“MICROSOFT”。

| | | | | |
|---|---|---|---|---|
| A | C | P | R | C |
| X | S | O | P | C |
| V | O | V | N | I |
| W | G | F | M | N |
| Q | A | T | I | T |

**解答：** 由于人工解决这个问题的方案是相对直观的，所以只需要描述人工求解该问题的算法。但具有讽刺意味的是，描述该算法并不容易。

**如何进行人工操作？** 首先匹配第一个元素，如果匹配成功，则在第一次匹配成功的元素的 8 个邻居中匹配第二个元素，递归执行这个过程，当输入模式的最后一个字符也匹配时，返回匹配成功。在上述过程中，注意二维数组中的任何单元都不能使用两次。为此，需要用某些符号标记每个已被访问的单元。如果模式在某处匹配失败，那么要在剩余的单元中从模式的开始重新匹配。当返回时，取消已访问单元的标记。

下面把上述直观方法转换为算法。因为在模式匹配中每次都执行相似的检测，所以需要一个递归的解决方案。在递归方案中，需要检测已经过的子串是否在给定矩阵中被匹配成功，前提是不使用已经使用过的单元。为了寻找已经使用过的单元，在函数中需要利用另一个二维数组（或者可以使用输入数组中未被使用的空间）。此外，还需要输入矩阵的当前位置来表示从哪里开始。因为与实际得到的信息相比，需要传递更多的信息，所以需要一个包装器函数来初始化需要被传递的额外信息。

**算法：**

```
如果匹配了模式的最后一个字符
    返回 true
如果遇到已使用的单元
    返回 false
如果越过了二维矩阵的边界
    返回 false
如果查找的第一个元素与单元不匹配
    按照行优先顺序（或列优先顺序）的下一个单元进行 FindMatch
否则 如果字符匹配
```

标记这个单元为已使用
res = 在 8 个邻居中与模式的下一个位置字符进行 FindMatch
标记这个单元为未使用
返回 res
否则
返回 false

```
#define MAX 100
bool FindMatch_wrapper(char[][] mat, char[] pat, int patLen, int nrow, int ncol) {
        if(strlen(pat) > nrow*ncol)
                return false;
        int used[MAX][MAX] = {{0,},};
        return FindMatch(mat, pat, patLen, used, 0, 0, nrow, ncol, 0);
}
//level: 进行索引直至模式匹配. x, y: 二维数组的当前位置
boolean FindMatch(char[][] mat, char[] pat, int patLen, int used[MAX][MAX], int x, int y,
int nrow, int ncol, int level) {
        if(level == patLen) //模式串匹配
           return true;
        if(nrow == x || ncol == y)
           return false;
        if(used[x][y])
           return false;
        if(mat[x][y] != pat[level] && level == 0) {
           if(x < (nrow - 1))
             return FindMatch(mat, pat, patLen,used, x+1, y, nrow, ncol, level);
           //同一行的下一个元素
        else if(y < (ncol - 1))
           return FindMatch(mat, pat, patLen,used, 0, y+1, nrow, ncol, level);
           //同一列的第一个元素
        else return false;
}
else if(mat[x][y] == pat[level]) {
        boolean res;
        used[x][y] = 1;       //标记该单元为已使用
        //在8个方向的邻居中寻找子串模式
        res = (x > 0  ? FindMatch(mat, pat, used, x-1, y, nrow, ncol, level+1) :   false) ||
              (res = x < (nrow - 1)  ? FindMatch(mat, pat, used, x+1, y, nrow, ncol, level+1) :
               false) ||
              (res = y > 0 ? FindMatch(mat, pat, used, x, y-1, nrow, ncol, level+1) :   false) ||
              (res = y < (ncol - 1)  ? FindMatch(mat, pat, used, x, y+1, nrow, ncol, level+1) :
               false) ||
              (res = x < (nrow - 1) && (y < ncol -1) ? FindMatch(mat, pat, used,
               x+1, y+1, nrow,  ncol, level+1) :  false) ||
              (res = x < (nrow - 1) && y > 0  ? FindMatch(mat, pat, used, x+1, y-1, nrow, ncol,
              level+1) : false)
||
              (res = x > 0 && y < (ncol - 1) ? FindMatch(mat, pat, used, x-1, y+1, nrow, ncol,
               level+1) : false)
||
              (res = x > 0 && y > 0 ? FindMatch(mat, pat, used, x-1, y-1, nrow, ncol, level+1) :
               false);
      used[x][y] = 0;              //标记该单元为未使用
```

```
                return res;
            }
            else return false;
    }
```

**问题 14** 给定一个大小为 $n \times n$ 的矩阵来存储随机整数。设计一个检查行列是否匹配的算法。例如，若第 $i$ 行与第 $j$ 列匹配，第 $i$ 行包含的元素为[2，6，5，8，9]，那么第 $j$ 列的元素也应该是[2，6，5，8，9]。

**解答：** 需要为列中的数据构建一棵键树(同理也可以用行中的数据进行构建)，比较行与键树。只要行的开始不匹配任何列就退出(回溯)。每次都要把一行与所有列进行检查。

如果不想因为空指针浪费空间，可通过构建后缀树来进一步改进解决方法。



第 16 章 Chapter 16

# 算法设计技术

## 16.1 引言

前面的章节针对不同的问题介绍了各种算法。在求解一个新问题时，通常的思路是寻找当前问题与已解决问题之间的相似之处，从而轻松找到新问题的求解方法。

本章将对各种算法按照不同的方法进行分类，然后在随后的 3 章中分别介绍 3 个算法设计思想(即贪婪、分治和动态规划)。

## 16.2 分类

关于算法的分类有很多种方法，下面列出了其中的一些分类方法：

- 实现方法
- 设计方法
- 其他分类方法

## 16.3 按实现方法分类

### 1. 递归或迭代

递归算法指算法反复地调用自身，直到满足某个基准条件。它在 C、C++ 等函数式编程语言中被普遍采用。迭代算法中经常使用循环等结构，并有时采用栈和队列等数据结构来求解问题。

有些问题适合用递归实现，另一些问题则适合用迭代实现。例如，汉诺塔问题的递归实现非常容易理解。每一个递归实现均可以改成迭代实现，反之亦然。

### 2. 过程式或声明式(非过程)

对于声明式编程语言，只需要指明所需达到的目标，而无需给出实施的细节。过程式编程语言需要指明得到期望结果的具体步骤。例如，SQL 更倾向于是一种声明式语言，

而非过程式，因为查询操作不需要给出查询的具体步骤。而 C、PHP、PERL 等则是过程式语言的例子。

**3. 串行或并行或分布式**

当讨论算法时，一般认为计算机一次执行一条指令，这就是所谓的串行程序(算法)。并行算法则利用计算机体系结构的特点一次处理多条指令，并将原问题分割成多个子问题，然后将它们分配到多个处理器或者线程来分别求解。迭代算法一般是可并行化的。如果将并行算法分布到不同的机器上，则将这样的算法称为分布式算法。

**4. 确定性或不确定性**

确定性算法基于预定义的过程来求解问题，而不确定性算法在每一步通过某种启发式规则来推测最优解。

**5. 精确或近似**

许多问题不一定能够找出最优解。因此，那些能够给出最优解的算法称为精确算法。在计算机科学中，对于那些不能够提供精确解的问题，常常给出近似算法。近似算法往往用于求解 NP 难问题(详细描述见第 20 章)。

## 16.4 按设计方法分类

除了实现方法外，还可以根据设计方法来对算法进行分类。

**1. 贪婪法**

贪婪算法将问题分为多个阶段。在每一个阶段，选取当前状态的最优决策，而不考虑对后续决策的影响。这意味着算法在执行过程中会选取某些局部最优解。贪婪法假设通过局部最优解可以获得全局最优解。

**2. 分治法**

分治法按以下 3 个步骤求解问题：

1) 分：将原问题分成多个子问题，这些子问题是与原问题类型相同的规模更小的实例。

2) 递归：递归求解子问题。

3) 治：合理地组合子问题的解。

**例子**：归并排序和二分查找算法。

**3. 动态规划方法**

动态规划(Dynamic Programming，DP)与备忘录方法相结合。动态规划与分治法的不同是：分治法的子问题之间不存在依赖关系，而 DP 的子问题存在重叠。通过备忘录技术(用一个表保存已解决子问题的答案)，动态规划法可将许多问题的复杂度从指数级降低为多项式级($O(n^2)$、$O(n^3)$等)。

动态规划与递归的不同之处在于递归调用的备忘录技术。当各个子问题之间相互独立时，没有重复调用，则这种备忘录技术对降低复杂度没有任何帮助，因此动态规划不是一种适用于所有问题的方法。通过备忘录技术(用一个表保存已解决子问题的答案)，动态规划可以将算法复杂度从指数级降低为多项式级。

**4. 线性规划方法**

在线性规划中，在满足不等式约束的条件下，最大化(或最小化)输入变量的线性函数。许多问题(如，有向图的最大流)可以采用线性规划方法。

### 5. 归约(转化和求解)

这种方法的思想是，将一复杂的问题转化和求解为一个已有渐近最优解的已知问题。该方法的目标是寻找一个归约算法，使得归约后的算法复杂度不会更高。例如，寻找线性表中中位数的算法是：首先将线性表排序，然后寻找有序表的中间值。这两个步骤分别称为转化和求解。

## 16.5　其他分类法

### 1. 按研究领域分类

在计算机科学中，每一个领域都有各自的问题并需要有效的算法。例如，查找算法、排序算法、归并算法、数值算法、图算法、字符串算法、几何算法、组合算法、机器学习、加密、并行算法、数据解压缩算法等。

### 2. 按复杂度分类

算法根据与输入规模相关的求解时间进行分类。有些算法具有线性时间复杂度($O(n)$)，另一些算法耗费指数级时间，甚至某些算法从不停止。注意某些问题可能有不同复杂度的多种求解算法。

### 3. 随机算法

有些算法需要进行随机选择。对于某些问题，其最快的求解算法必须涉及随机操作，例子：快速排序。

### 4. 分支定界、枚举和回溯

这些方法在人工智能领域都有使用，这里不展开叙述。对于回溯方法，可参考第 2 章。

**注意**：在后面的三章中，将分别讨论贪婪、分治和动态规划三种算法设计技术。关注这三种技术，因为用这三种技术求解的问题比其他技术多。



Chapter 17

# 第 17 章

# 贪婪算法

## 17.1 引言

首先通过对一个简单理论的讨论，初步理解贪婪思想。以下棋为例，每一步的决策都需要考虑对后续棋局的影响。而在网球(或排球)比赛中，选手的行为仅取决于当前的状况，选择当下最为正确的动作，而不关心后续的影响。这说明在某些情况下选择当下最佳行为的决策，可以得到一个最优解(贪婪)，但并非所有情况都如此，贪婪策略适用于上述第二类问题。

## 17.2 贪婪策略的定义

贪婪算法将问题分为多个阶段。在每一个阶段，选取当前状态下的最优决策，而不考虑对后续决策的影响。这意味着算法在执行过程中会选取某些局部最优解。贪婪法假设通过局部最优解可以获得全局最优解。

## 17.3 贪婪算法的要素

最优贪婪算法需要满足两个基本性质：

1) 贪婪选择性质。

2) 最优子结构。

**贪婪选择性质：**全局最优解可以通过寻找局部最优解获得(贪婪)，局部最优解的选择可能依赖于之前的决策，而不是后续的决策。通过迭代方式算法进行一个个贪婪选择，将原问题简化为规模更小的问题。

**最优子结构：**如果原问题的最优解包含子问题的最优解，则认为该问题具有最优子结构。这意味着可以对子问题求解并构建规模更大问题的解。

## 17.4 贪婪算法的适用范围

选择局部最优解不是对于所有问题都适用，所以贪婪算法并不总是能得到最优解。在17.8节和第19章将给出这样的例子。

## 17.5 贪婪算法的优缺点

贪婪算法的优点是直观，易于理解，并易于编码实现。当前的决策不会对已经计算出的结果有任何影响，因此不需要再对已有的局部解进行检查。

缺点是，对于许多问题，无法用贪婪算法求解。即在许多情况下，无法保证局部最优解能够产生全局最优解。

## 17.6 贪婪算法的应用

- 排序问题：选择排序、拓扑排序。
- 优先队列：堆排序。
- 赫夫曼编码压缩算法。
- Prim 和 Kruskal 算法。
- 加权图的最短路径问题(Dijkstra 算法)。
- 硬币找零问题。
- 分数背包问题。
- 并查集的按大小或高度合并问题(或排名)。
- 任务调度算法。
- 贪婪算法可用于求解复杂问题的近似算法。

## 17.7 贪婪思想

本节通过一个例子来更好地理解贪婪思想，更多细节参见17.6节的相关主题。

**赫夫曼编码算法**

**定义：** 给定来自字母表 $A$ 的 $n$ 个字符的集合(字符 $c \in A$)，并已知每个字符出现的频率 freq($c$)，为每一个字符 $c \in A$ 找到一个二进制编码，使得 $\sum_{c \in A} \text{freq}(c)\,|\text{binarycode}(c)|$ 的值最小，其中 $|\text{binarycode}(c)|$ 表示字符 $c$ 的二进制编码的长度。以上公式表明所有字符编码长度之和(每个字符出现频率与编码的位数乘积之和)最小。

赫夫曼编码算法的基本思想是，对于出现频率较大的字符用更少的位来编码。利用可变长度编码，赫夫曼算法可以压缩数据存储所需的空间。计算机系统采用8位来表示每一个字符，但并非所有的位都被使用。此外，某些字符的使用更为频繁。当读取一个文件时，系统通常每次读取8位来确定一个字符。但是这种8位编码机制是低效的，因为相比而言，有些字符使用更为频繁。例如，字符'e'往往比字符'q'的使用频率高10倍。

因此，如果对于字符'e'用7位编码，而'q'用9位编码，这将减少整个消息的长度。平均而言，对于标准文件，使用赫夫曼编码在长度上能够减少10%～30%，具体的值取决于字符的频率。这种编码思想是，对于较少使用的字符或字符组采用较长的二进制编码。此外，赫夫曼编码满足任意两个字符的编码互不为前缀。

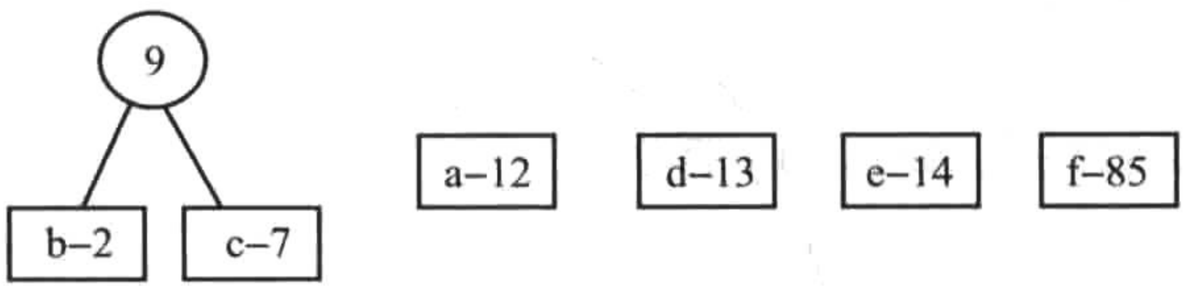
**例子：**假设扫描一个文件，得出以下字符频率：

| 字符 | 频率 |
|---|---|
| a | 12 |
| b | 2 |
| c | 7 |
| d | 13 |
| e | 14 |
| f | 85 |

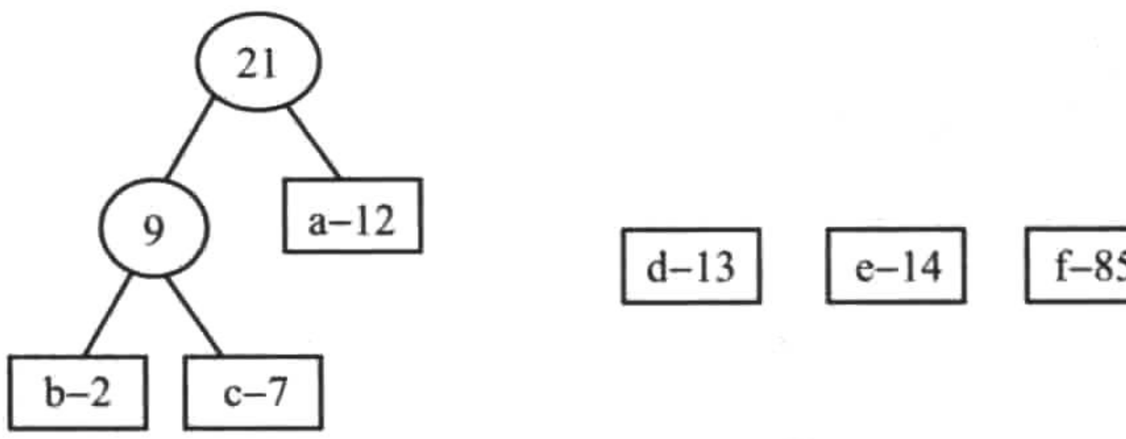
首先，为每一个字符创建一棵二叉树，并将其出现频率存储在结点中(见下图)。

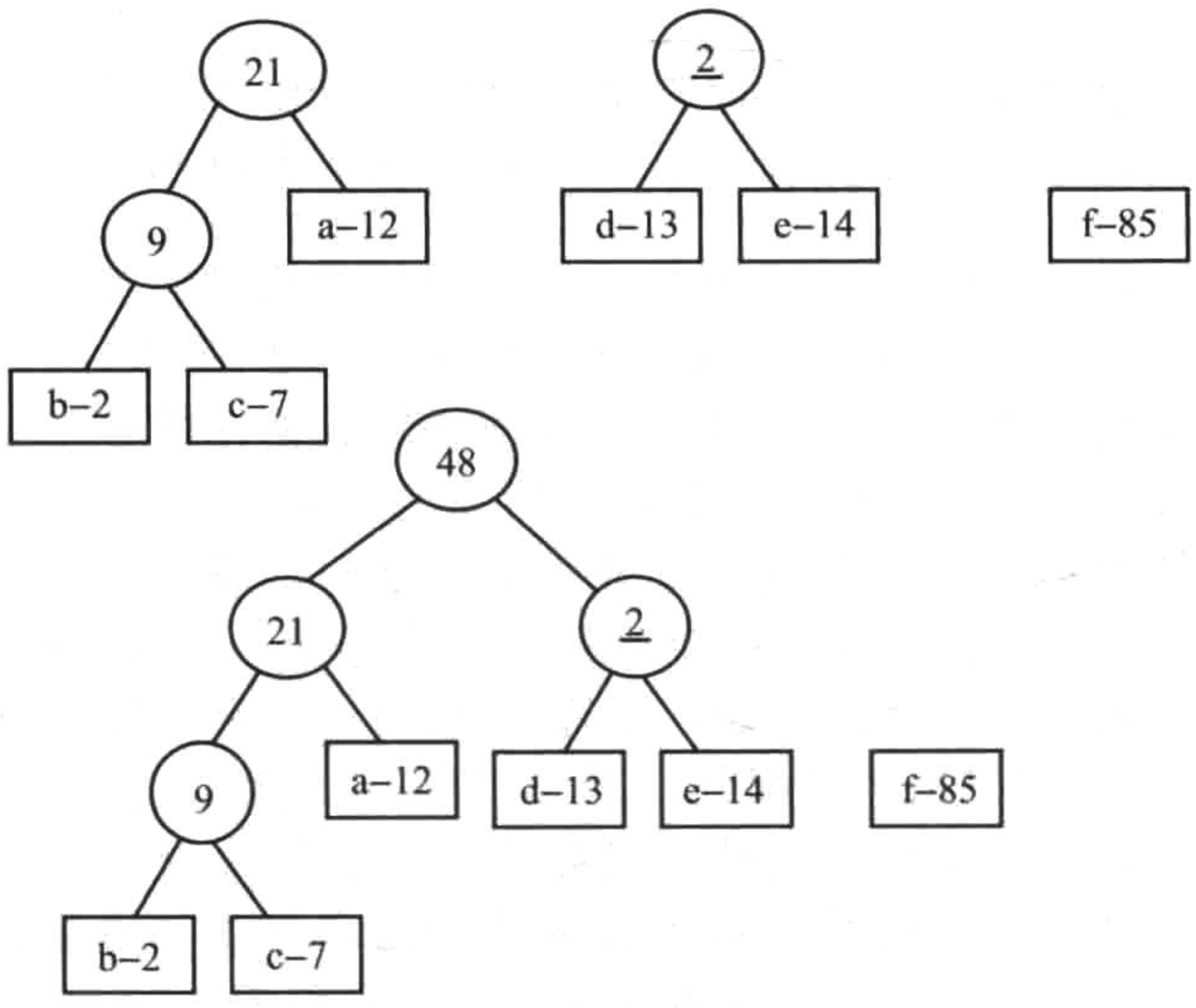


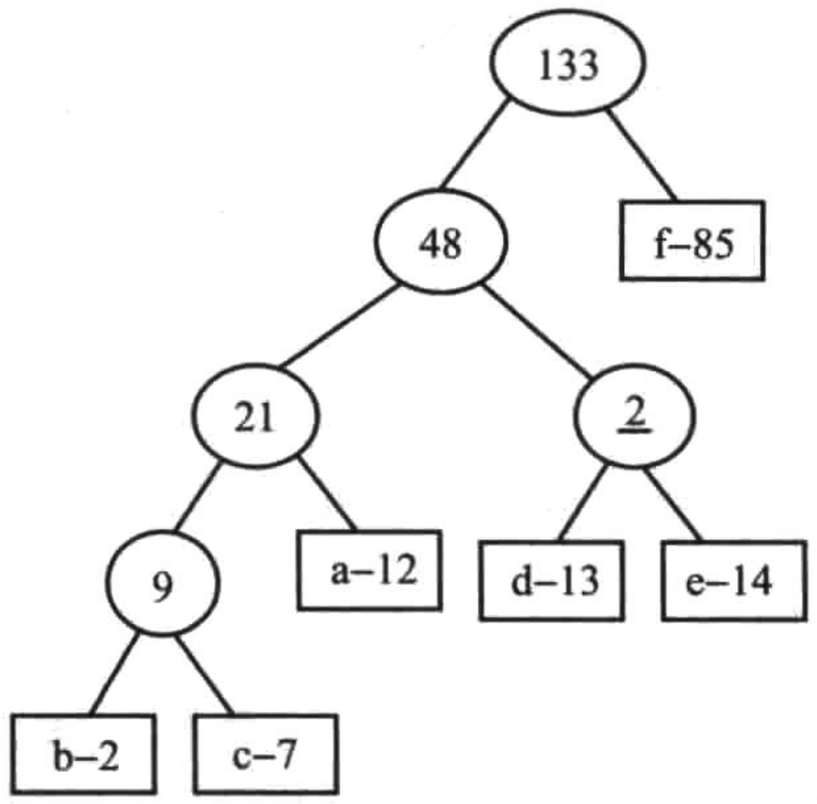

赫夫曼算法的流程如下：在列表中寻找根结点中存有最小频率值的两棵二叉树，创建一个不存储任何字符的结点，将这两棵二叉树作为新建结点的左右子树，并将其孩子结点的频率值之和存储到新建结点中。由此可以得出下图：




重复上述过程直至仅剩下一棵树。






当树构造完后，每一个叶子结点对应于一个字母及其编码。从根结点遍历到叶子结点，就可以得到每一个结点的编码。对于左分支，在编码中添加 0；对于右分支，添加 1。对于上述产生的树，可以得到以下编码：

| 字母 | 编码 | 字母 | 编码 |
|---|---|---|---|
| a | 001 | d | 010 |
| b | 0000 | e | 011 |
| c | 0001 | f | 1 |

**计算减少的位数：**接下来对使用赫夫曼编码减少的位数进行统计，只需要用原始的数据存储位数减去采用赫夫曼编码后的存储位数。

在本例中，由于仅有 6 个字母，所以假设每个字母用 3 位进行编码。因为共有 133 个字符(总频率乘以 3)，所以需要的总位数为 3×133＝399。使用赫夫曼编码，所需要的位数是：

| 字母 | 编码 | 频率 | 总位数 |
|---|---|---|---|
| a | 001 | 12 | 36 |
| b | 0000 | 2 | 8 |
| c | 0001 | 7 | 28 |
| d | 010 | 13 | 39 |
| e | 011 | 14 | 42 |
| f | 1 | 85 | 85 |
| 共计 | | | 238 |

因此，减少了 399－238＝161 位，将近减少 40%的存储空间。

```
HuffmanCodingAlgorithm(int A[], int n) {
        // 在A中初始化一个包含n个元素的优先队列PQ;
        Heap PQ = new Heap();
        BinaryTreeNode temp;
        for (i =  1; i<n; i++) {
                temp   = new BinaryTreeNode();
                temp.setLeft(PQ .deleteMin());
```

```
                temp.setRight(PQ.DeleteMin());
                temp.setData(temp.getLeft().getData() + temp.getRight().getData());
                PQ.insert(temp);
            }
            return PQ;
    }
```

时间复杂度为 $O(n\log n)$，其中包含在不超过 $n$ 个元素的优先队列上的一个建堆操作，$2n-2$ 次堆删除操作和 $n-2$ 次堆插入操作，详见第 7 章。

## 17.8 贪婪算法的相关问题

**问题 1** 给定一个大小为 $n$ 的数组 $F$，数组元素 $F[i]$表示第 $i$ 个文件的长度。现在需要将所有文件合并成一个文件，以下算法是否提供了该问题的最优解？

**算法：** 依次连续合并文件，即按照默认顺序选择最前两个文件进行合并，然后将合并后的文件与第三个文件合并，以此类推。

**注意：** 给定大小分别为 $m$ 和 $n$ 的两个文件 $A$ 和 $B$，则合并的复杂度为 $O(m+n)$。

**解答：** 上述算法不是最优的求解算法。下面给出反例，假设文件大小的数组 $F$ 如下。

$$F = \{10,5,100,50,20,15\}$$

根据上述算法，首先合并前两个文件(大小分别为 10 和 5)，合并后的文件列表如下，其中 15 表示合并大小为 10 和 5 两个文件的代价。

$$\{15,100,50,20,15\}$$

同理，合并大小为 15 的文件和下一个大小为 100 的文件，得到{115，50，20，15}。以此类推，产生的文件列表如下：

$$\{165,20,15\}$$

$$\{185,15\}$$

最后为

$$\{200\}$$

合并的总代价＝所有合并代价之和＝15＋115＋165＋185＋200＝680。为了确定上述结果是否是最优解，考虑文件的另一种排列：{5，10，15，20，50，100}。对于该例，同样按照上述过程，合并总代价＝15＋30＋50＋100＋200＝395。因此，该算法不能给出最优的解决方法。

**问题 2** 与问题 1 类似，以下算法是否给出问题的最优解？

**算法：** 两两合并文件，即将文件依次两两分组合并。第一轮后，产生 $n/2$ 个中间文件。接下来，将上一轮产生的中间文件两两分组合并，以此类推。

**注意：** 有些文献将上述算法称为两路归并。若不是两两分组，而是一次合并 $K$ 个文件，则称为 $K$ 路归并。

**解答：** 上述算法仍然不是最好的求解算法。仍然以上题中给出的例子作为反例进行说明。根据算法思想，第一轮中对第一组(文件大小为 10 和 5)、第二组(文件大小 100 和 50)、第三组(20 和 15)分别合并，得到以下文件列表。

$$\{15,150,35\}$$

同理，将以上文件列表两两分组合并，结果为(最后一个分组中仅包含一个元素，结果保持不变)：

{165,35}

最后为

{185}

合并的总代价＝所有合并代价之和＝15＋150＋35＋165＋185＝550。该代价高于 395（见问题 1），因此上述算法不能给出最好的解决方法。

**问题 3**　对于问题 1，将所有文件合并为一个文件最好的方法是什么？

**解答：**使用贪婪算法降低文件合并的总时间。

**算法：**

1）按照文件的大小构建一个优先队列，元素的关键值为文件长度。

2）重复以下步骤直至队列中只有一个文件，

a. 选择两个最小的元素 $X$ 和 $Y$。

b. 将 $X$ 和 $Y$ 合并，并将合并后的新文件插入优先队列中。

**同样算法的变形：**

1）将所有文件按照文件大小进行升序排序。

2）重复以下步骤直至只有一个文件，

a. 选择有序表中最前的两个元素（最小）$X$ 和 $Y$。

b. 合并 $X$ 和 $Y$，并将合并后的文件插入有序表中。

通过对前面例子的分析来分析上述算法的性能。已知数组

$$F = \{10,5,100,50,20,15\}$$

根据上述算法，对数组进行排序得到{5，10，15，20，50，100}。合并两个最小的文件（大小分别为 5 和 10），结果如下所示，其中 15 表示合并大小为 10 和 5 两个文件的代价。

{15,15,20,50,100}

同理，继续合并两个最小的文件（大小为 15 和 15），得到{20，30，50，100}。接下来，依次得到以下文件列表：

{50,50,100} // 合并 20 和 30

{100,100} // 合并 20 和 30

最后为

{200}

合并的总代价＝所有合并代价之和＝15＋30＋50＋100＋200＝395。因此，该算法得到该合并问题最好的解决方法。

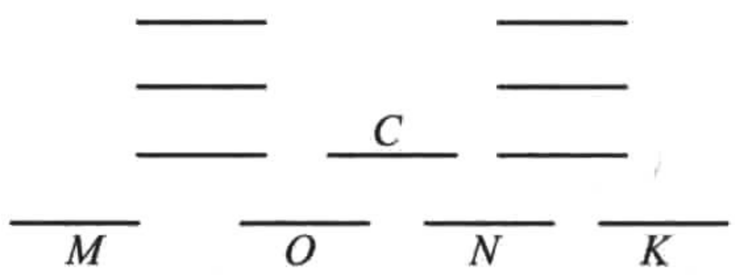
时间复杂度：使用堆查找最佳合并模式的时间开销 $O(n\log n)$ 加上文件合并的最低代价。

**问题 4　区间调度算法：**给定一个含 $n$ 个区间的集合 $S=\{(\text{start}_i, \text{end}_i) \mid 1\leqslant i\leqslant n\}$，寻找 $S$ 的一个最大子集 $S'$，使得 $S'$ 中任意一对区间都不重合。分析以下算法是否可行？

**算法：**

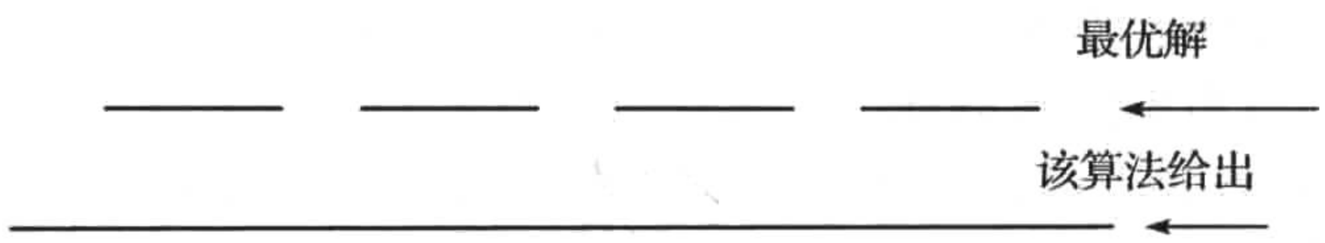
```
while(S 非空){
    选择与其他区间重合数最少的区间 I;
    将 I 添加到最终解集合 S'中;
    将所有与 I 有重合的区间从 S 中删除;
}
```

**解答：**该算法不能找到非重合区间的最大子集。考虑以下区间，其最优解是{$M$，$O$，$N$，$K$}。然而，与其他区间重合数最小的是 $C$，因此该算法首先选取 $C$。



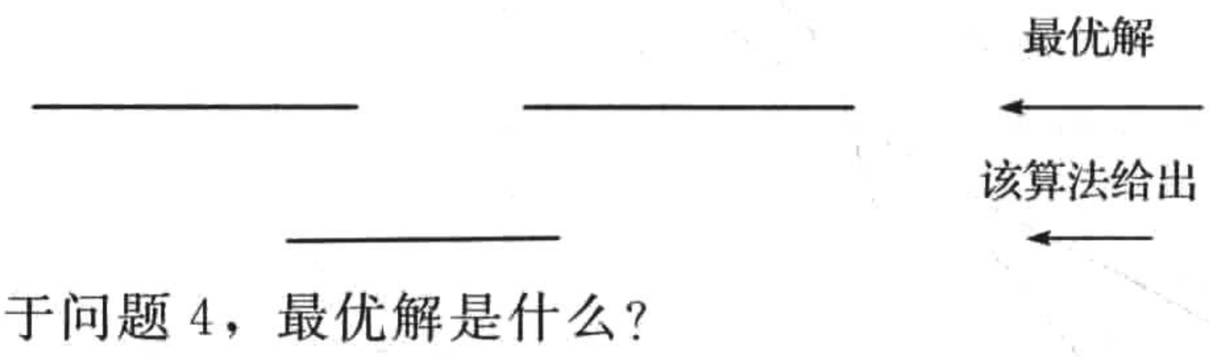

**问题 5** 对于问题 4，如果选择最先出现的区间(并且与已经选择的区间不重合)，则该算法是否能给出最优解？

**解答：不能。**该算法也不能给出最优解。反例如下，最优解为 4，而该算法得到的解为 1。




**问题 6** 对于问题 4，如果选择最短的区间(并且与已经选择的区间不重合)，此算法是否能给出最优解？

**解答：**该算法同样不能给出最优解。反例如下，最优解为 2，而该算法得到的解为 1。




**问题 7** 那么对于问题 4，最优解是什么？

**解答：**考虑使用贪婪算法寻找最优解。

**算法：**

将所有区间按照最右端(结束时间)进行排序

对每一个连续的区间{

　　如果其最左端在上一个选择区间的最右端之后，则选取该区间

　　否则，舍弃该区间，进入下一次迭代

}

时间复杂度＝排序的时间＋扫描的时间＝$O(n\log n+n)=O(n\log n)$。

**问题 8** 考虑如下问题。

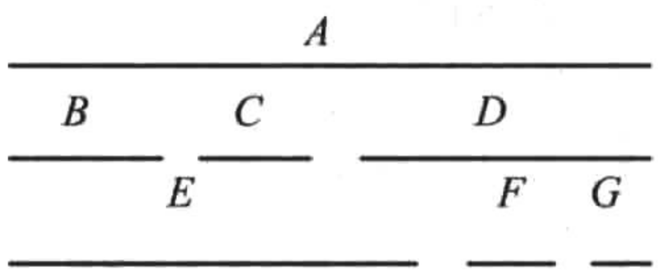
输入：区间集合 $S=\{(\text{start}_i,\ \text{end}_i)\mid 1\leqslant i\leqslant n\}$。区间$(\text{start}_i,\ \text{end}_i)$表示在时间段 $\text{start}_i$ 到 $\text{end}_i$，某个课程在某个教室上的一个使用申请。

输出：寻找所有课程的教室安排方案，使得占用教室最少。

**算法：**安排尽可能多的课程到第一间教室，然后安排尽可能多的课程到第二间教室，然后安排尽可能多的课程到第三间教室，以此类推。这个算法能否给出最优解？

**注意：**实际上，该问题与区间调度问题类似，只是应用背景不一样。

**解答：**上述算法未解决区间着色问题，考虑如下例子：



最大化第一间教室安排的课程数，则{B，C，F，G}在同一间教室，课程 A、D 和 E 分别各占一间教室，共需 4 间教室。最优方案是将 A 安排在一间教室，{B，C，D}安排在一间教室，{E，F，G}安排在一间教室，共 3 间。

**问题 9**　对于问题 8，考虑如下算法。根据课程开始时间按递增顺序依次处理，假设当前正在为课程 $C$ 安排教室，如果教室 $R$ 已经安排了 $C$ 之前的课程，且将 $C$ 安排到 $R$ 不会与之前安排的课程重叠，那么将课程 $C$ 安排到教室 $R$，否则将 $C$ 安排到一个新教室。这个算法能否解决该问题呢？

**解答：**上述算法能够解决区间着色问题。因为算法按照课程开始的时间顺序进行教室的安排，所以如果该贪婪算法为当前课程 $c_i$ 安排了一间新教室，这意味着 $c_i$ 的开始时间与所有已使用教室的最后一门课程的时间冲突。因此贪婪算法安排最后一间教室 $n$，因为当前课程的开始时间与其他 $n-1$ 间教室的课程相冲突。假设任何时刻，任何课程最多只与 $s$ 门课程冲突，则必有 $n \leqslant s$，即 $s$ 是所需教室总数的下界。综上可知，贪婪算法是可行的，可以给出最优解。

**注意：**求最优解的算法参见问题 7，代码参见问题 10。

**问题 10**　假设两个数组 Start[1..$n$]和 Finish[1..$n$]分别给出每一个课程的开始时间和结束时间，现在需要寻找最大的子集 $X \in \{1, 2, \cdots, n\}$，使得其中任意两个元素 $i$，$j \in X$ 满足 Start[$i$]>Finish[$j$]或 Start[$j$]>Finish[$i$]。

**解答：**目标是尽可能早地完成第一门课，这样为其他课程腾出更多的时间。按照结束时间的顺序轮询每一门课程，当时间不与上一门已选课程相冲突时，选择该门课程。

```
int LargestTasks(int Start[], int n,  int Finish []) {
        sort Finish[];
        rearrange Start[] to match;
        count = 1;
        X[count]  = 1;
        for (int i  = 2;  i<n;  i++) {
                if(Start[i] > Finish[X[count]])       {
                        count  = count + 1;
                        X[count] =  I;
                }
        }
        return X[1 .. count];
}
```

容易看出，该算法由于排序需要 $O(n\log n)$的时间开销。

**问题 11**　思考印度的货币找零问题。问题的输入为整数 $M$，输出为兑换 $M$ 卢比所需的最少硬币数。在印度，假设可用的硬币面额有 1、5、10、20、25、50 卢比，并且每一种面值的硬币数量不限。

**对于该问题，以下算法能否给出最优解？**

尽可能取用面额最高的硬币，例如，兑换 234 卢比的零钱，贪婪算法将使用 4 个面额 50 卢比的硬币，1 个 25 卢布的硬币，1 个 5 卢比的硬币和 4 个 1 卢比的硬币。

**解答：**对于面额为 1、5、10、20、25、50 卢比的硬币找零问题，上述贪婪算法不能给出使硬币数最少的最优解。为了兑换 40 卢比的硬币，贪婪算法的结果为 3 个硬币，面额分别为 25、10 和 5 卢比，而最优解则为使用两个 20 先令的硬币。

**注意：**对于该问题的最优解，参见第 19 章。

**问题 12** 假设在城市 $A$ 和 $B$ 进行长途自驾之旅。在准备旅行时，下载了一份包含自驾路线上所有加油站之间距离(以英里为单位)的地图。假设汽车的油箱能装载使汽车行驶 $n$ 英里的汽油($n$ 为已知)。如果在每一个加油站都停车加油，是否是最优解？

**解答：**上述算法不能给出最优解。原因很明显，在每一个加油站均加油不能产生最优解。

**问题 13** 对于问题 12，当且仅当没有足够到下一个加油站的汽油时才停车，并且一旦停车，则将油箱加满油。证明该算法能够或不能解决上述问题。

**解答：**贪婪算法策略如下：将油箱装满油，从 $A$ 开始旅行，根据地图决定在旅行路线上 $n$ 英里以内最远的加油站，在该加油站停车并加满油，然后再次根据地图决定从该停车点出发 $n$ 英里以内最远的加油站，重复该过程直至到达 $B$。

**注意：**算法代码参见第 19 章。

**问题 14 分数背包问题：**有 $t_1$，$t_2$，…，$t_n$ 件物品(希望放入背包的物品)，其重量分别为 $s_1$，$s_2$，…，$s_n$，同时每件物品的回报值为 $v_1$，$v_2$，…，$v_n$，在物品净重不超过 $C$ 的前提下如何最大化回报值？

**解答：**

**算法：**

1) 为每件物品计算单位重量的价值密度 $d_i = v_i / s_i$。

2) 根据以上价值密度对物品进行排序。

3) 尽可能多地将价值密度大的物品放入背包中。

时间复杂度：排序为 $O(n\log n)$，贪婪选择为 $O(n)$。

**注意：**将物品放入一个优先队列，然后一个一个拿出放入背包中直至背包装满或者所有的物品被取出。实际上，这将获得一个更好的运行时间复杂度 $O(n+c\log n)$，其中 $c$ 为被选取物品的实际数量。如果 $c=O(n)$，则运行时间会继续减少，否则复杂度不变。

**问题 15 火车站台数：**一个火车站有一个所有火车到达和离开的时间表，需要找出最小的站台数，使得按照此时间表调度时，可以容纳所有的火车。例如，如下的时间表，最小站台数为 3，否则车站将无法容纳所有的火车。

| 火车 | 到达时间 | 离开时间 | 火车 | 到达时间 | 离开时间 |
|---|---|---|---|---|---|
| 车次 A | 0900 | 0930 | 车次 C | 1030 | 1100 |
| 车次 B | 0915 | 1300 | 车次 D | 1045 | 1145 |

**解答：**从上面给出的例子可以看出，计算站台的数目等同于确定车站在任一时刻容纳的最大火车数。

首先，在一个数组中对到达时间($A$)和离开时间($D$)进行排序，将相应的到达或离开状态也保存在数组中。排序后的数组如下表所示。

| 0900 | 0915 | 0930 | 1030 | 1045 | 1100 | 1145 | 1300 |
|---|---|---|---|---|---|---|---|
| *A* | *A* | *D* | *A* | *A* | *D* | *D* | *D* |

其次，将数组中的 $A$ 替换为 1，$D$ 换成 $-1$，替换后的数组为：

| 1 | 1 | −1 | 1 | 1 | −1 | −1 | −1 |
|---|---|---|---|---|---|---|---|

最后，根据上述数组计算一个累积值数组：

| 1 | 2 | 1 | 2 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|

数组中的最大值即为问题的解，这里是3。

**注意：**如果一列火车的到达时间与另一列火车的离开时间相同，则在排序数组中离开时间优先。

**问题16** 假设某个国家有很长的马路，居民的房子分散在马路两旁。居民使用移动电话，需要沿着马路架设移动电话基站，每一个基站覆盖范围为7千米。设计一个有效算法，使得需要的基站数最少。

**解答：**该算法用于定位最少数量的基站。

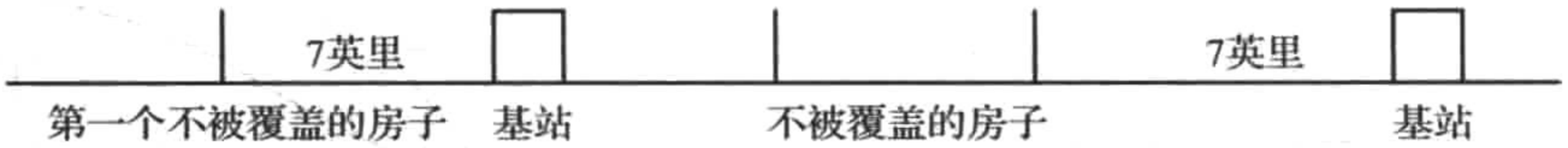
1）从马路起点处开始。

2）找到马路上第一个未被覆盖的房子。

3）如果找不到这样的房子，则算法终止，否则转到下一步。

4）沿着马路在距离步骤2中的房子7英里处架设基站。

5）转到步骤2。




**问题17 准备音乐磁带：**假设有$n$首歌曲，需要将其存储到一盘磁带中。以后用户需要从磁带中读取这些歌曲，但从磁带中读一首歌曲与从磁盘中读取的方式不同。首先需要快进越过其他歌曲，这需要花费大量的时间。假设数组$A[1..n]$列出了每一首歌曲的长度，例如歌曲$i$的长度为$A[i]$。如果歌曲按照$1\sim n$的顺序存储，那么访问第$k$首歌曲的开销为：

$$C(k)=\sum_{i=1}^{k}A[i]$$

这个开销表明在访问第$k$首歌曲之前需要越过磁带前面的所有歌曲。如果改变磁带上歌曲的顺序，则访问歌曲的开销也随之改变。有些歌曲的访问变得非常耗时，而另一些则非常容易。不同的歌曲顺序可能导致不同的期望开销。如果假设每一首歌曲访问的可能性是相等的，那么采用何种歌曲顺序能够使得期望开销尽可能小？

**解答：**答案非常简单，按照从最短歌曲到最长歌曲的顺序存储，将较短的歌曲存储在靠前的位置减少了访问剩余歌曲的快进时间。

**问题18** 考虑海德拉巴会议中心(Hyderabad Convention Center，HITEX)的一组活动安排，假设有$n$项活动，每项活动需要一个单位时长。如果活动$i$在时间$T[i]$或之前开始，则可以创收$P[i]$卢比($P[i]>0$)，其中$T[i]$为一个任意数字。如果直至$T[i]$活动还没开始，则无任何收入。所有活动最早开始时间为时刻0。设计一个有效算法，找到一个收入最大的活动调度方案。

**解答：算法**

- 根据floor($T[i]$)对所有活动进行排序(从大到小)。floor()表示下取整函数。

- 令 $t$ 为当前调度时刻(初始时 $t=\text{floor}(T[i])$)。
- 所有 $\text{floor}(T[i])=t$ 的活动插入一个优先队列，其收入 $g_i$ 作为关键字。
- 执行 DeleteMax 操作选取一个在时刻 $t$ 开始的活动。
- 然后 $t$ 自减，重复上述过程。

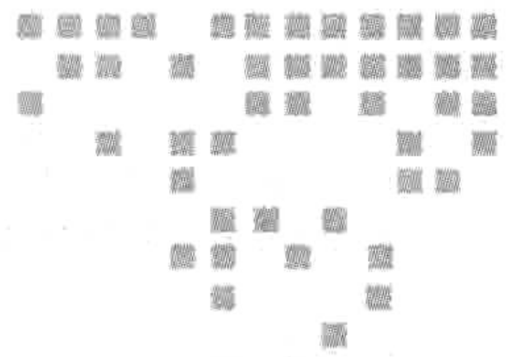
显然时间复杂度为 $O(n\log n)$，其中排序花费 $O(n\log n)$，算法中包含最多 $n$ 次优先队列的插入和 DeleteMax 操作，每一个操作花费 $O(\log n)$。

**问题 19** 考虑一个客户服务员(如，移动客户服务)，有 $n$ 个客户排队等待服务。为了简单起见，假设每一个顾客的服务时间是已知的，顾客 $i$ 的服务时间为 $w_i$ 分钟。因此如果按照 $i$ 递增的顺序对顾客进行服务，那么第 $i$ 位顾客需要等待 $\sum_{j=1}^{n-1} w_j$ 分钟，所有顾客的总等待时间 $= \sum_{i=1}^{n}\sum_{j=1}^{i-1} w_j$ 。

如何确定服务客户的最好方法，使得总等待时间减少？

**解答：** 这个问题可以使用贪婪方法轻易地解决。因为目标是减少总等待时间，所以要达到此目的只需选择服务时间较少的用户。即，如果按照服务时间递增的顺序服务顾客，就能够减少总等待时间。

时间复杂度为 $O(n\log n)$。



第 18 章 Chapter 18

# 分治算法

## 18.1 引言

对于第 17 章列举的许多问题，贪婪策略不能提供最优解。而其中的某些问题可通过分治(Divid and Conquer，D&C)法来轻松求解。分治法是一种重要的基于递归的算法设计技术，分治算法递归地将问题分解为两个或多个同类型的子问题，直至这些子问题简单到能够直接求解，然后再将这些子问题的解合成为原始问题的解。

## 18.2 分治策略的定义

应用分治策略求解问题包括以下步骤：

1) 分(divide)：将初始问题分割成多个子问题，这些子问题是与初始问题同类型的规模更小的实例。

2) 递归(recursion)：递归求解子问题。

3) 治(conquer)：合理地组合子问题的解。

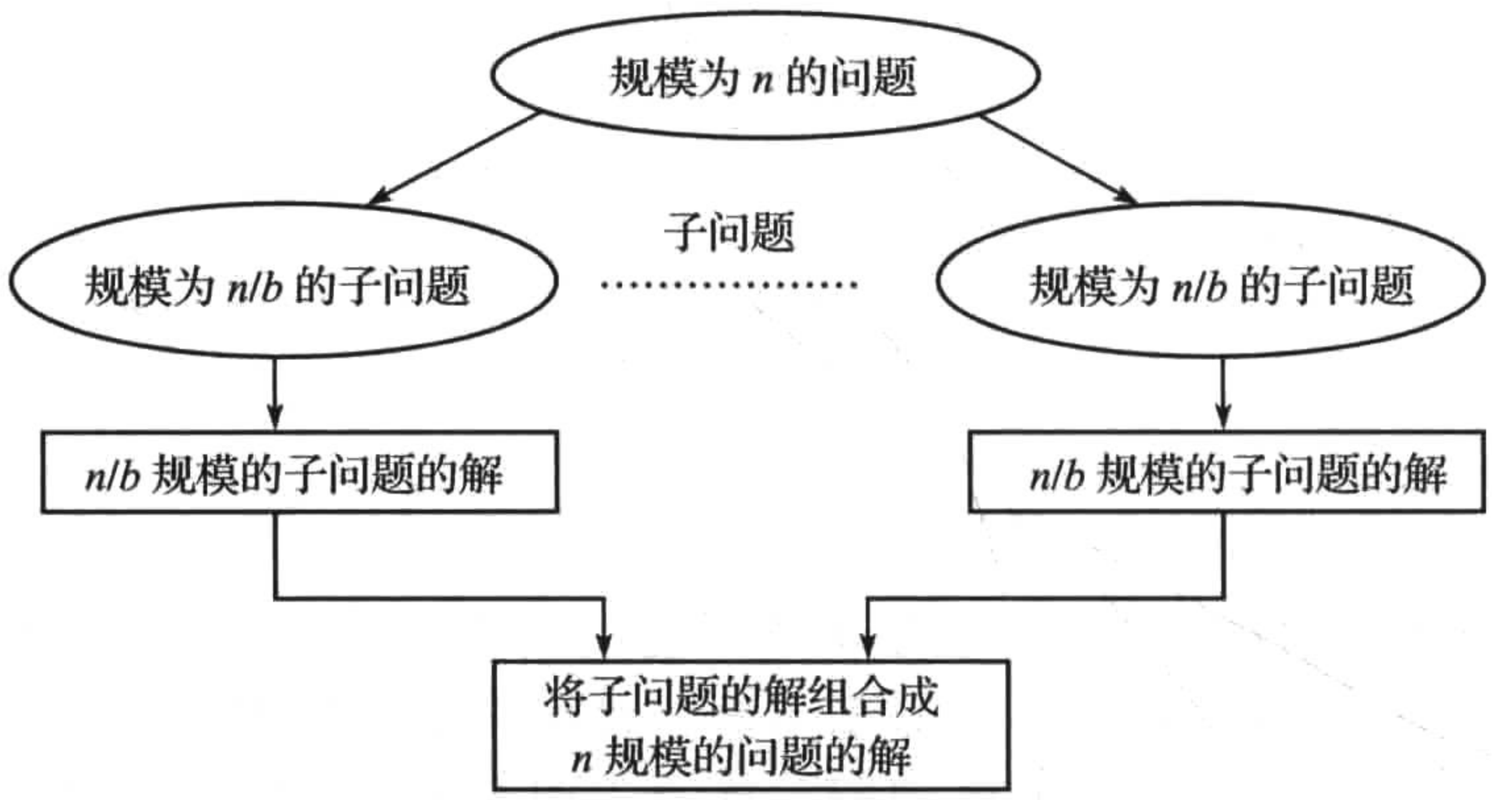
## 18.3 分治法的适用范围

分治法并不能解决所有问题。根据分治法的定义，算法递归地求解同类型的子问题。但不是任何问题都能找到具有同样类型和规模的子问题，因此分治法不能适用于所有问题。

## 18.4 分治法的图形化描述

为了更好地理解分治法，下面给出分治法的一个图形化描述。假设初始问题的的规模为 $n$，根据上述分治法的定义，将该问题分解为规模为 $n/b$ 的子问题($b$ 为常数)。然后递归地求解这些子问题，最后将各个子问题的解合成为初始问题的解。

```
DivideAndConquer ( P ) {
        if( small ( P ) )
        // 为了让解决方法更加显著，让P取值很小
                return solution ( n );
        divide the problem P into k sub problems P1, P2, ..., Pk;
        return (
                Combine (
                        DivideAndConquer ( P1 ),
                        DivideAndConquer ( P2 ),
                        ...
                        DivideAndConquer ( Pk )
                        )
                );
}
```




## 18.5 分治思想

为了清晰地理解分治法，考虑如下故事。曾经有一个富有的老农，他有 7 个儿子。他担心死后他的土地和财产将在他的 7 个儿子之间分割，由此可能会引发彼此之间的争吵。于是，一天他将 7 个儿子召集在一块，拿出捆绑在一起的 7 根棍子，并说："谁能将其折断，谁将继承所有财产"。7 个儿子都试了一遍，但没有人能够折断这捆棍子，最后老人解开了它，将每一根棍子一一折断。于是兄弟 7 人决定齐心协力，共同努力，共同成功。而对于问题求解来说，其寓意则不一样，如果不能直接求解问题，可以将其分解为多个部分，依次求解每一部分。

前面的章节已经使用分治法策略求解了许多问题，如二叉搜索、归并排序、快速排序等。这些方法都折射出分治法的思想。下面给出一些其他实际问题，它们也能够利用分治法策略轻松求解。在这些问题中都能够找到与初始问题相似的子问题。

1）在电话簿中查找某个名字：电话簿中的名字按照字母顺序排列。给出一个名字，如何确定该名字是否在电话簿中？

2）将石头打碎为粉尘：将一块石头变成粉尘(非常小的石头)。

3）寻找酒店的出口：在一个很长的酒店大厅内，遍布了很多的门，找出哪扇门是出口。

4）在停车场内寻找自己的汽车。

### 1. 分治法的优点

**求解复杂的问题**：分治法是求解复杂问题的有效方法。例如，汉诺塔问题。这需要将原始问题分解为子问题，然后求解这些简单的子问题，最后通过合成子问题的解来实现初始问题的求解。如何将问题分解为子问题并使子问题的解又能够组合成原始问题的解是设计一个新算法的主要难点。对于许多这类复杂的问题，分治法提供了简单的解决方法。

**并行性**：因为分治法独立地求解各个子问题，不同的子问题可以由不同的处理器来执行，所以分治法算法适合在多处理器的机器上运行，特别是共享内存的系统(处理器之间的数据通信开销可忽略不计)。

**内存访问**：分治法算法能够更好地利用内存缓存机制。这是因为一旦子问题规模足够小，其派生的所有子问题都能够在缓存中求解，而不需要访问存取速度较慢的主存。

### 2. 分治法的缺点

分治法方法的一个缺点是递归的执行速度较慢，这是由于反复的子问题调用所引入的开销。另外，分治法方法需要栈来存储这些调用(每次递归调用的当前状态)。事实上，这些开销的大小取决于具体的实现方式。如果递归有大量的基本情况，则许多问题的递归开销将变得微不足道。

分治法的另一个缺点是：对于某些问题，分治法方法比迭代方法更复杂。例如，对于 $n$ 个数求和，一个简单的循环将所有数依次相加比分治法要容易得多，分治法需要将所有的数分成两半，然后各自递归求和，最后再将两个和依次相加得到最终的解。

## 18.6　主定理

如前所述，分治法递归地求解子问题。所有问题一般按照递归进行定义，用主定理(Master theorem)容易求得这些递归问题的时间复杂度。主定理的详细介绍见第 1 章。为了描述的连续性，下面对定理再次进行说明。

如果递归时间复杂度具有如下形式 $T(n)=aT\left(\frac{n}{b}\right)+\Theta(n^k\log^p n)$，其中 $a\geqslant 1$，$b>1$，$k\geqslant 0$，$p$ 为实数，那么复杂度的表达式可分为以下形式：

1）如果 $a>b^k$，那么 $T(n)=\Theta(n^{\log_b a})$。

2）如果 $a=b^k$，

a. 如果 $p>-1$，那么 $T(n)=\Theta(n^{\log_b a}\log^{p+1} n)$。

b. 如果 $p=-1$，那么 $T(n)=\Theta(n^{\log_b a}\log\log n)$。

c. 如果 $p<-1$，那么 $T(n)=\Theta(n^{\log_b a})$。

3）如果 $a<b^k$，

a. 如果 $p\geqslant 0$，那么 $T(n)=\Theta(n^k\log^p n)$。

b. 如果 $p<0$，那么 $T(n)=\Theta(n^k)$。

## 18.7　分治法的应用

- 二分查找
- 归并排序
- 快速排序
- 中间值查找

- 最大和最小值查找
- 矩阵乘法
- 最近点对问题

## 18.8 分治法的相关问题

**问题 1** 假设算法 A 是求解某问题的一个算法，该算法将初始问题分解为规模减半的 5 个子问题，然后递归地求解初始问题，并在线性时间内对子问题的解进行组合。问该算法的复杂度是多少？

**解答：** 假设输入规模为 $n$，$T(n)$是求解给定问题所需要的时间。根据以上描述，将初始问题分解为规模为$\frac{n}{2}$的 5 个子问题，则求解这 5 个子问题需要 $5T\left(\frac{n}{2}\right)$时间，然后组合 5 个子问题的解(相当于依次扫描一个数组)需要线性的时间。因此，该问题的递归算法总时间为：

$$T(n) = 5T\left(\frac{n}{2}\right)+O(n)$$

利用分治法主定理，时间复杂度为 $O(n^{\log_2 5})\approx O(n^{2+})\approx O(n^3)$。

**问题 2** 类似于问题 1，假设算法 B 求解规模为 $n$ 的问题的过程为：递归求解规模为 $n-1$的两个子问题，然后在常数时间内组合子问题的解。问该算法的时间复杂度是多少？

**解答：** 假设输入规模为 $n$，$T(n)$为求解该问题所需要的时间。由于该算法将初始问题分解为规模为 $n-1$ 的两个子问题，所以求解子问题的时间为 $2T(n-1)$，然后在常数时间内组合子问题的解，该递归算法的总时间复杂度为：

$$T(n) = 2T(n-1)+O(1)$$

利用分治法主定理，时间复杂度为 $O(n^0 2^{\frac{n}{1}})=O(2^n)$(更多细节请见第 1 章)。

**问题 3** 同样类似于问题 1，算法 C 通过将问题分解为 9 个大小为$\frac{n}{3}$的子问题，递归求解每个子问题，然后以 $O(n^2)$的时间组合子问题的解。问该算法的时间复杂度是多少？

**解答：** 令输入大小为 $n$，$T(n)$为求解该问题所需要的时间。由于该算法将初始问题分解为大小为$\frac{n}{3}$的 9 个子问题，所以需要用 $9T\left(\frac{n}{3}\right)$时间求解这些子问题。然后合成这些解需要二次方的时间。

该递归算法的总时间复杂度为：

$$T(n) = 9T\left(\frac{n}{3}\right)+O(n^2)$$

利用分治法主定理，时间复杂度为 $O(n^2\log n)$。

**问题 4** 求解以下递归代码的时间复杂度是多少？

```
void function(int n) {
        if(n > 1) {
                System.out.println(("*");
                function(n/2);
                function(n/2);
        }
}
```

**解答：**假设输入规模为 $n$，$T(n)$为求解该问题所需要的时间。根据输出字符后的代码可知，将初始问题分解为大小为$\frac{n}{2}$的两个子问题来分别求解，因此子问题的求解时间为 $2T\left(\frac{n}{2}\right)$。

在子问题求解后，该程序不需要组合子问题的解，所以该算法的时间复杂度为：

$$T(n) = 2T\left(\frac{n}{2}\right)+O(1)$$

利用分治法主定理，时间复杂度为 $O(n^{\log_2 2})\approx O(n^1)=O(n)$。

**问题 5**　给定一个数组，设计一个寻找最大值和最小值的算法。

**解答：**请参见第 12 章。

**问题 6**　讨论二分查找及其复杂度。

**解答：**关于二分查找的讨论见第 11 章。

**分析：**假设输入规模为 $n$，$T(n)$为求解该问题所需要的时间。所有元素有序排列，二分查找算法将选取有序列表中的中间元素并与待查找的值比较，如果相等则返回该元素。

如果待查找值大于中间元素，则在左边的子数组中继续查找，并忽略右边的子数组。类似地，如果待查找值小于中间元素值，则在右边的子数组中继续查找，而忽略左边的子数组。

这表明在两种情况下算法都将忽略一半的子数组，仅考虑另一半的子数组。每一次迭代都将数组分成两等份。

以上描述可归纳为：每次将初始问题分解为规模为$\frac{n}{2}$的两个子问题，然后求解其中之一，花费时间为 $T\left(\frac{n}{2}\right)$。递归算法的总时间复杂度为：

$$T(n) = T\left(\frac{n}{2}\right)+O(1)$$

利用分治法主定理，时间复杂度为 $O(\log n)$。

**问题 7**　将以上二分查找做如下修改：将数组分为三等份(三分查找，又称为三叉搜索)，而不是两等份，设计三分查找的递归算法，并计算其时间复杂度。

**解答：**根据问题 6 的讨论可知，二分查找具有以下递归关系：$T(n)=T\left(\frac{n}{2}\right)+O(1)$。类似于问题 6 的讨论，用“3”代替递归中的“2”，即表示将数组分为 3 个同等大小的子数组，然后在其中的一个子数组中继续查找，三分查找的递归算法复杂度为：

$$T(n) = T\left(\frac{n}{3}\right)+O(1)$$

利用分治法主定理，可得时间复杂度为 $O(\log_3 n)\approx O(\log n)$(不需要考虑对数的底，因为通过换底公式可变为一个常数因子)。

**问题 8**　对于问题 6，如果将数组分成约为初始问题规模 1/3 和 2/3 的两个集合又如何呢？

**解答：**在三分查找的基础上稍微修改：每一次迭代只进行一次比较，将数组分成两

部分，一部分大约包含$\frac{n}{3}$的元素，另一部分包含$\frac{2n}{3}$个元素。最坏情况下该算法为每次都对包含$\frac{2n}{3}$个元素的较大子数组进行递归调用，其相应的复杂度递归表达式为：

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

利用分治法主定理，算法复杂度为$O(\log n)$。注意一个有趣的现象：当$n$趋于无穷时，任何$k$分查找算法都具有相同的复杂度(只要$k$是一个不依赖于$n$的固定常数)。

**问题 9** 讨论归并排序及其复杂度。

**解答：** 有关归并排序的讨论见第 10 章。对于归并排序，如果待排序元素的数量大于 1，则将其划分到两个规模相等的子集中，然后在子集上递归调用该算法，最后合并返回的有序子集，得到初始集合的有序排列。归并排序算法复杂度的递归式为：

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n) & n > 1 \\ 0 & n = 1 \end{cases}$$

利用分治法主定理，求解上述递归式得到$O(n\log n)$。

**问题 10** 讨论快速排序及其复杂度。

**解答：** 有关快速排序的讨论见第 10 章。对于快速排序，其复杂度在最好情况和最坏情况是不相同的。

**最好情况：** 在快速排序时，如果待排序元素的数量大于 1，则将它们划分为两个相等的子集，然后在子集上递归调用快速排序算法。当得到各个子问题的解后，不需要进行合并，因为快速排序保证各子问题的解已经是有序排序的。但是，需要扫描所有元素对其进行划分。快速排序最好情况下的复杂度递归式为：

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n) & n > 1 \\ 0 & n = 1 \end{cases}$$

利用分治法主定理，求解该递归式的复杂度为$O(n\log n)$。

**最坏情况：** 假设输入规模为$n$，$T(n)$为求解规模为$n$的问题所花费的时间。最坏情况下，快速排序算法将元素划分为两个集合，其中一个集合仅包含一个元素，另一个集合则包含$n-1$个元素，求解两个子问题所需的时间分别为$T(1)$和$T(n-1)$。但快速排序算法需要扫描所有的输入元素来将元素分为两个集合(花费$O(n)$的时间)。

在求解了子问题后，快速排序算法仅需要常数时间将子问题的解进行合并，因此，求解该问题的总时间开销可用以下递归式计算：$T(n)=T(n-1)+O(1)+O(n)$。

上式展开后是一个明显的等差数列求和式，所以$T(n)=\frac{n(n+1)}{2}=O(n^2)$。

**注意：** 平均情况下的复杂度分析，见第 10 章。

**问题 11** 已知一个无穷大数组，前$n$个单元存储有序排列的整数，其余的单元包含一些特殊的符号(如，$)。假设不知道$n$到底多大，对于输入的整数$K$，设计一个算法在$O(\log n)$的时间内找到$K$在数组中插入的位置。

**解答：** 由于限定了算法的时间复杂度为$O(\log n)$，那么遍历已知数组中所有元素是不可行的(因为该方法的复杂度为$O(n)$)。为了得到$O(\log n)$的复杂度，一种可能的方法是

利用二分查找，但在该问题中，因为不清楚数组的结束位置，所以二分查找不适用。那么首要的问题就是要找到数组的结束位置，可以从第一个元素开始，以倍数间隔递增的方式查找，即从下标 1 开始，然后 2，4，8…

```
int FindInInfiniteSeries(int A[]) {
        int mid, l = r = 1;
        while( A[r] != '$') {
                l = r;
                r = r × 2;
        }
        while( (r – l > 1 ) {
                mid = (r – l)/2 + l;
                if( A[mid] == '$')
                        r = mid;
                else     l = mid;
        }
}
```

显然，一旦确定了 $K$ 可能插入的区间 $A[i, \cdots, 2i]$，由于该区间的长度最大为 $n$(因为数组 $A$ 中仅包含 $n$ 个整数)，所以利用二分查找来搜索 $K$ 花费 $O(\log n)$时间。

**问题 12**　已知一个不包含相同元素的有序数组 $A[1..n]$，查找是否存在下标 $i$ 满足 $A[i]=i$，给出求解该问题的复杂度为 $O(\log n)$的分治算法。

**解答：**可以采用二分查找，其复杂度为 $O(\log n)$。

```
int IndexSearcher(int A[] , int l, int r) {
        int mid = (r – l)/2 + l;
        if(r-l <= 1) {
                if(A[l] == l || A[r] == r)
                        return 1;
                else     return 0;
        }
        if(A[mid] < mid )
                return IndexSearcher( A, mid + 1,r);
        if(A[mid] > mid)
                return IndexSearcher( A, 1, mid - 1);
        return mid;
}
```

上述函数复杂度的递归表达式为 $T(n)=T(n/2)+O(1)$。

根据分治法主定理，$T(n)=O(\log n)$。

**问题 13**　已知大小为 $n$ 的两个有序表，设计一个查找两个表并集的中位数的算法。

**解答：**利用归并排序的归并操作(见第 10 章)，当比较两个数组的元素时，对当前已排序元素进行计数。如果计数器为 $n$(因为共有 $2n$ 个元素)，则表明到达两个列表并集的中间点，计算归并后数组的第 $n-1$ 和 $n$ 个元素的平均值。

时间复杂度为 $O(n)$。

**问题 14**　继续上述问题，如果两个数组的大小不一样，如何计算呢？

**解答：**该问题的解法与前一个问题的解法相似。假设两个列表的长度分别为 $m$ 和 $n$，则当计数器达到$(m+n)/2$ 时停止。

时间复杂度为 $O((m+n)/2)$。

**问题 15**　能否将问题 13 的时间复杂度降低为 $O(\log n)$？

**解答：可以**，使用分治法。假设两个数组分别为 $L1$ 和 $L2$。

**算法：**

1）找出两个给定的有序数组 $L1[]$和 $L2[]$的中位数，假设分别为 $m1$ 和 $m2$。

2）如果 $m1$ 与 $m2$ 相等，则返回 $m1$(或者 $m2$)。

3）如果 $m1$ 大于 $m2$，那么最终的中位数可能是以下两个子数组之一。

a）从 $L1$ 的第一个元素开始至 $m1$。

b）从 $m2$ 开始到 $L2$ 的最后一个元素。

4）如果 $m2$ 大于 $m1$，那么最终的中位数可能是以下两个子数组之一。

a）从 $m1$ 开始到 $L1$ 的最后一个元素。

b）从 $L2$ 的第一个元素开始至 $m2$。

5）重复上述过程直至两个子数组的大小为 2。

6）如果两个子数组的大小为 2，则使用以下公式计算中位数。

$$\text{中位数}=\left(\frac{\max(L1[0],\ L2[0]+\min(L1[1],\ L2[1]))}{2}\right)$$

时间复杂度为 $O(\log n)$，因为每次迭代仅考虑一半的输入，而忽略另一半。

**问题 16** 已知数组 $A$，其元素值允许重复，在数组中查找给定元素的下标，若元素有重复，则给出最高的下标。

**解答：** 参见第 11 章。

**问题 17** 讨论基于分治法的 Strassen 矩阵乘法算法。已知两个大小为 $n\times n$ 的矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，计算 $n\times n$ 的矩阵 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$，其元素按照下式计算：

$$C_{i,j}=\sum_{k=0}^{n-1}A_{i,k}B_{k,j}$$

**解答：** 在讨论 Strassen 算法前，先回顾基本的分治算法。求解该问题的一般方法是，为了得到 $C[i,\ j]$的值，需要 $\boldsymbol{A}$ 的第 $i$ 行与 $\boldsymbol{B}$ 的第 $j$ 列相乘。

```
// 初始化C
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
        C[i, j] += A[i, k] * B[k, j];
```

矩阵乘法问题可以利用分治法来求解。实现分治算法需要将原问题分解为与其相似的多个子问题。在本例中，将 $n\times n$ 矩阵看作一个 $2\times 2$ 的矩阵，其元素为 $\frac{n}{2}\times\frac{n}{2}$ 的子矩阵，因此，原矩阵的乘法 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$ 可以转换为：

$$\begin{bmatrix}C_{1,1} & C_{1,2}\\ C_{2,1} & C_{2,2}\end{bmatrix}=\begin{bmatrix}A_{1,1} & A_{1,2}\\ A_{2,1} & A_{2,2}\end{bmatrix}\times\begin{bmatrix}B_{1,1} & B_{1,2}\\ B_{2,1} & B_{2,2}\end{bmatrix}$$

其中 $A_{i,j}$、$B_{i,j}$、$C_{i,j}$ 为 $\frac{n}{2}\times\frac{n}{2}$ 矩阵。

从 $C_{i,j}$ 的定义可知，各个子矩阵可按下式计算：

$$\begin{aligned}C_{1,1}&=A_{1,1}\times B_{1,1}+A_{1,2}\times B_{2,1}\\ C_{1,2}&=A_{1,1}\times B_{1,2}+A_{1,2}\times B_{2,2}\\ C_{2,1}&=A_{2,1}\times B_{1,1}+A_{2,2}\times B_{2,1}\\ C_{2,2}&=A_{2,1}\times B_{1,2}+A_{2,2}\times B_{2,2}\end{aligned}$$

其中符号＋和×分别表示$\frac{n}{2}\times\frac{n}{2}$矩阵的加法和乘法运算。

为了计算原 $n\times n$ 矩阵乘法，需要计算 8 个$\frac{n}{2}\times\frac{n}{2}$矩阵积(分)以及 4 个$\frac{n}{2}\times\frac{n}{2}$矩阵和(治)。因为矩阵加法是一个复杂度为 $O(n^2)$的操作，所以矩阵乘法运算的总时间可用如下递归式表达：

$$T(n)=\begin{cases}O(1) & n=1\\ 8T\left(\frac{n}{2}\right)+O(n^2) & n>1\end{cases}$$

基于分治法主定理，$T(n)=O(n^3)$。

幸运的是，结果表明在 8 个矩阵乘法中有一个是冗余的(由 Strassen 发现)。考虑下面 7 个$\frac{n}{2}\times\frac{n}{2}$矩阵序列：

$$\begin{aligned}\boldsymbol{M}_0&=(A_{1,1}+A_{2,2})\times(B_{1,1}+B_{2,2})\\ \boldsymbol{M}_1&=(A_{1,2}-A_{2,2})\times(B_{2,1}+B_{2,2})\\ \boldsymbol{M}_2&=(A_{1,1}-A_{2,1})\times(B_{1,1}+B_{1,2})\\ \boldsymbol{M}_3&=(A_{1,1}+A_{1,2})\times B_{2,2}\\ \boldsymbol{M}_4&=A_{1,1}\times(B_{1,2}-B_{2,2})\\ \boldsymbol{M}_5&=A_{2,2}\times(B_{2,1}-B_{1,1})\\ \boldsymbol{M}_6&=(A_{2,1}+A_{2,2})\times B_{1,1}\end{aligned}$$

每个等式仅包含一个乘法运算。计算 $\boldsymbol{M}_0$ 到 $\boldsymbol{M}_6$ 需要 10 个加法和 7 个乘法。当求出 $\boldsymbol{M}_0$ 到 $\boldsymbol{M}_6$ 后，可以如下计算乘积矩阵 $\boldsymbol{C}$ 的个元素：

$$\begin{aligned}C_{1,1}&=\boldsymbol{M}_0+\boldsymbol{M}_1-\boldsymbol{M}_3+\boldsymbol{M}_5\\ C_{1,2}&=\boldsymbol{M}_3+\boldsymbol{M}_4\\ C_{2,1}&=\boldsymbol{M}_5+\boldsymbol{M}_6\\ C_{2,2}&=\boldsymbol{M}_0-\boldsymbol{M}_2+\boldsymbol{M}_4-\boldsymbol{M}_6\end{aligned}$$

该方法需要 7 个$\frac{n}{2}\times\frac{n}{2}$矩阵乘法和 18 个$\frac{n}{2}\times\frac{n}{2}$矩阵加法。因此，最坏情况下运行时间可以由下面的递归式给出：

$$T(n)=\begin{cases}O(1) & n=1\\ 7T\left(\frac{n}{2}\right)+O(n^2) & n=1\end{cases}$$

基于分治法主定理，$T(n)=O(n^{\log_2 7})=O(n^{2.81})$。

**问题 18　股票价格问题：**考虑 CareerMonk.com 网站连续 $n$ 天的股票价格，即输入是由公司股票价格组成的数组。股票价格每天都会变。根据股票的价格，在股票价格高涨的日期，可以卖出所持股份；而在价格下跌的日期可以买进。现在找出股票买进和卖出的日期，使得收益最大。

**解答：**根据问题描述，假设输入是股票价格(整数)的数组，即 $A[1]$，…，$A[n]$。在该数组中找出两个日期(一个买进，一个卖出)，使得收益最大。另一个隐含条件是买进的日期应该在卖出之前。一个简单的方法是比较所有可能的买进和卖出日期。

```
void StockStrategy(int A[], int n, int *buyDateIndex, int *sellDateIndex) {
        int profit=0;
        *buyDateIndex =0; *sellDateIndex =0;;
        for (int i   = 1; i < n; i++)          //显示购买日期
                //显示销售日期
                for(int j  =  i; j < n;  j++)
                        if(A[j] - A[i] > profit)       {
                                profit = A[j] - A[i];
                                *buyDateIndex = i;
                                *sellDateIndex = j;
                        }
}
```

这两个嵌套的循环包含 $n(n+1)/2$ 次运算，因此时间复杂度为 $O(n^2)$。

**问题 19**　对于问题 18，是否能降低其时间复杂度？

**解答：可以**，通过分治法可得到复杂度为 $O(n\log n)$的算法。将输入数组分成两部分，递归地寻找各个部分的解。这里，原问题的解有 3 种情况：

- buyDateIndex 和 sellDateIndex 均在前半时间段内。
- buyDateIndex 和 sellDateIndex 均在后半时间短内。
- buyDateIndex 在前半时间段内，而 sellDateIndex 在后半时间段内。

前两种情况可通过递归直接求得。但因为第三种情况的 buyDateIndex 和 sellDateIndex 处在不同的部分，所以需要额外处理，即在两个子部分中寻找最小价格和最大价格，该操作可在线性时间内完成。

```
void StockStrategy(int A[], int left, int right) {
        //声明必要的变量;
        if(left + 1 = right)
                return (0, left, left);
        mid   = left + (right - left) / 2;
        (profitLeft, buyDateIndexLeft, sellDateIndexLeft) =  StockStrategy(A, left, mid);
        (profitRight, buyDateIndexRight, sellDateIndexRight)  = StockStrategy(A, mid, right);
        minLeft  = Min(A, left, mid);
        maxRight  = Max(A, mid, right);
        profit  = A[maxRight] - A[minLeft];
        if(profitLeft > max{profitRight, profit})
                return (profitLeft, buyDateIndexLeft, sellDateIndexLeft);
        else if(profitRight > max{profitLeft, profit})
                return (profitRight, buyDateIndexRight, sellDateIndexRight);
        else
                return (profit, minLeft, maxRight);
}
```

在输入大小为原始问题一半的两个子问题上递归调用算法 StockStrategy，然后花费 $\Theta(n)$时间查找最大价格和最小价格，因此时间复杂度可用递归式 $T(n)=2T(n/2)+\Theta(n)$计算，根据分治法主定理，可得 $O(n\log n)$。

**问题 20**　便携式计算机需要进行抗摔测试，证明其抗摔能力。详细过程如下：在一幢 $n$ 层的楼房上逐层测试，寻找便携式计算机落下而无损坏的最低楼层(称为“上限”)。假设只有两台需要确定其抗摔能力上限的便携式计算机，设计一个算法使尝试的次数 $f(n)$最小(希望 $f(n)$是次线性的，因为一个简单的算法就能达到线性的 $f(n)$)。

**解答：**对于该问题，无法使用二分查找，因为无法对原问题进行分解并递归求解。

考虑如下例子，假设答案是 14，即需要通过 14 次尝试才能确定抗摔上限。首先将便携式计算机从 14 层摔下，如果摔坏，那么从 1～13 层依次尝试；如果没有摔坏，那么剩下 13 次尝试。因此接下来尝试第 14＋13＝27 层。这是因为如果在第 27 层摔坏，则可以通过 12 次(剩下的总尝试次数)测试验证 15～26 层；如果没有摔坏，那么剩下 12 次尝试确认抗摔上限楼层。

从上述例子可知，第一次尝试的间隔为 14，然后 13，再接着 12，…。因此，如果解为 $k$，那么尝试的间隔为 $k$，$k-1$，$k-2$，…，1。如果总楼层数为 $n$，那么需要建立 $k$ 与 $n$ 之间的关系。因为最大可测试的楼层为 $n$，那么间隔之和应该小于 $n$，即

$$k+(k-1)+(k-2)+\cdots+1\leqslant n$$

$$\frac{k(k+1)}{2}\leqslant n$$

$$k\leqslant\sqrt{n}$$

该算法的复杂度是 $O(\sqrt{n})$。

**问题 21**　已知 $n$ 个数，检查是否存在两个数相等。

**解答：**参见第 11 章。

**问题 22**　给出一个算法，判断一个整数是否是某个整数的平方。例如，16 是，而 15 不是。

**解答：**假设初始时 $i=2$，计算 $i\times i$ 的值，判断该值是否等于给定的整数，如果相等，算法结束，否则将 $i$ 增加 1。继续此过程直至 $i\times i$ 大于或者等于给定的整数。

时间复杂度为 $O(\sqrt{n})$。空间复杂度为 $O(1)$。

**问题 23**　**螺母和螺栓问题：**有 $n$ 个大小不一的螺母和 $n$ 个与螺母一一匹配的螺栓，需要找出与每一个螺母匹配的螺栓。假设只能在螺母和螺栓之间进行比较(不能进行螺母之间或者螺栓之间的比较)。

**解答：**参见第 10 章。

**问题 24**　**最近点对问题：**已知包含 $n$ 个点的集合 $S=\{p_1, p_2, \cdots, p_n\}$，其中 $p_i=(x_i, y_i)$。寻找距离最小的一对点。为了简单起见，假设所有点分布在一维空间中。

**解答：**假设所有的点是有序的。因为点分布在一维空间，所以排序后所有点在一条直线上(在 $X$ 轴或 $Y$ 轴上)。排序算法的复杂度为 $O(n\log n)$。排序后只需要依次比较相邻两点的距离，并寻找距离最小的两个点。由上可知，求解该问题的复杂度为 $O(n\log n)$，主要取决于排序时间。

时间复杂度为 $O(n\log n)$。

**问题 25**　对于问题 24，如果点分布在二维空间中，应该如何求解呢？

**解答：**在描述算法前，先给出以下数学公式：

$$\text{distance}(p_1, p_2)=\sqrt{(x_1-x_2)^2-(y_1-y_2)^2}$$

上式用于计算两点 $p_1=(x_1, y_1)$和 $p_2=(x_2, y_2)$之间的距离。

**蛮力法：**

- 计算所有两点之间的距离。在 $n$ 个点中任意选择其中两个点有 $C_n^2$ 种组合方式($C_n^2=O(n^2)$)。
- 在得到所有 $n^2$ 种可能的距离后，寻找距离最小的点对的时间为 $O(n^2)$。

算法总的时间复杂度为 $O(n^2)$。

**问题 26** 对于问题 24，给出复杂度为 $O(n\log n)$的算法。

**解答：** 可以使用分治技术来得到 $O(n\log n)$的算法。在介绍该分治算法前，假设点已根据 $x$ 坐标升序排列。然后基于点的 $x$ 坐标的中位数将所有点分为两个相等的部分。此时原问题变成在两个相同大小的集合中寻找最近点对。为了简单起见，使用以下算法来说明该过程。

**算法：**

1）根据 $x$ 坐标值对 $S$(给定的点集)中的点进行排序，然后以穿过 $S$ 的中位数的线 $l$ 为轴将 $S$ 分成两个相等的子集 $S_1$ 和 $S_2$，这一步骤对应分治法中的分操作。

2）分别找出 $S_1$ 和 $S_2$ 中的最近点对，在递归实现中用 $L$ 和 $R$ 表示。

3）步骤 4～8 对应分治法中的治操作。

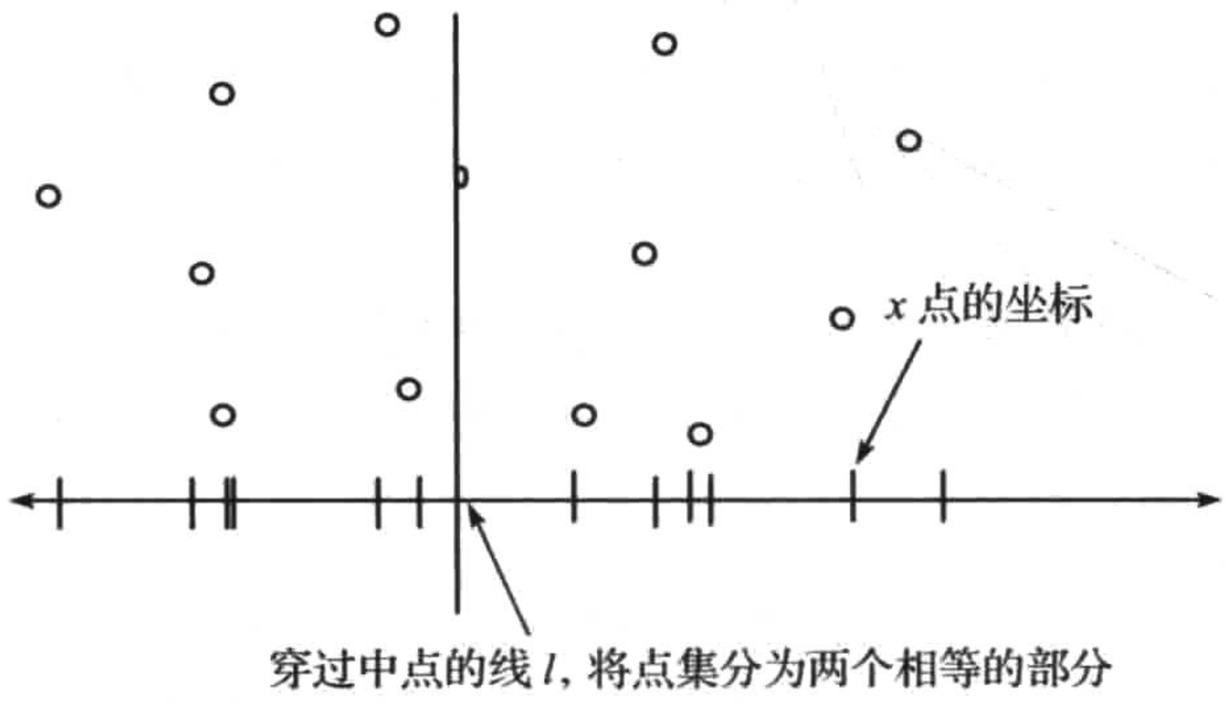
4）假设 $\delta=\min(L, R)$。

5）将与 $l$ 的距离大于 $\delta$ 的点去掉。

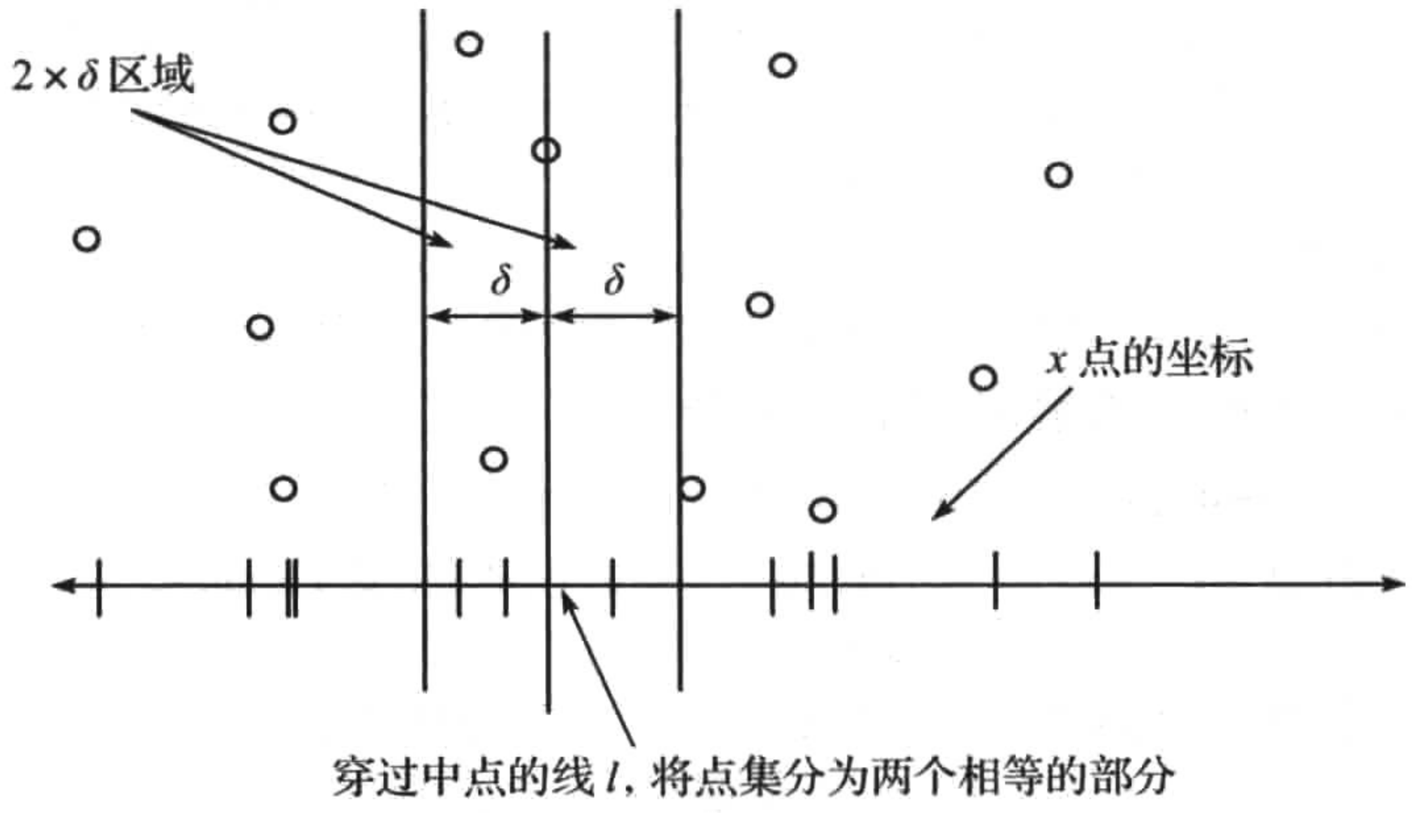
6）对于剩下的点，根据 $y$ 坐标值排序。

7）扫描排序后的点，计算每个点到其所有邻居(以该点的 $y$ 值为轴且距离不超过 $2\times\delta$)的距离(这是按照 $y$ 值大小排序的原因)。

8）如果距离小于 $\delta$，则更新 $\delta$。



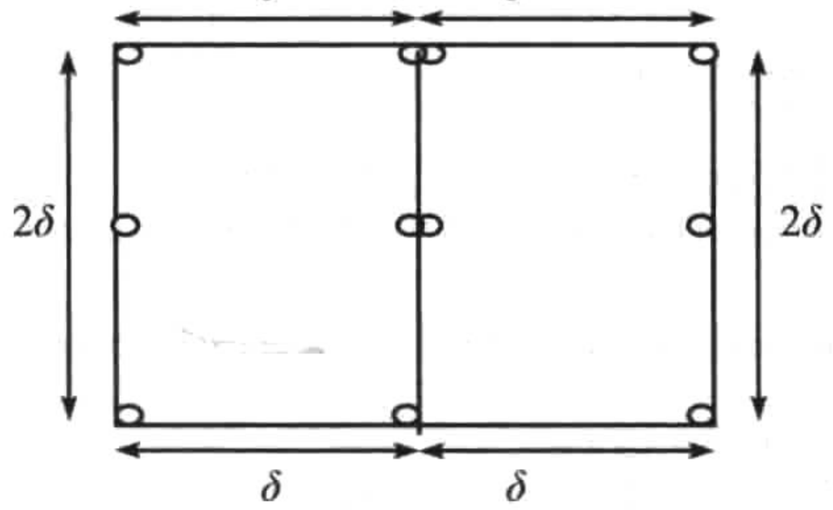

**线性时间内对解进行合并：**




设 $\delta=\min(L, R)$，其中 $L$ 是第一个子问题的解，$R$ 对应第二个子问题的解。最近点对还可能跨过分割线，位于距离分割线小于 $\delta$ 的范围内，因此仅需要考虑分布在分割

线附近 $2\times\delta$ 区域内的点，如上图所示。现在对该区域内的所有点按下图所示进行进一步处理。

在上图中，在一个方形区域内距离不小于 $\delta$ 的点最多可容纳 12 个，这意味着只需要检查有序表中 11 个位置的距离值。该过程与上面的过程类似，唯一的区别是上述子问题的合并操作在垂直方向不存在界限。因此可以将上述 12 点方盒策略反复应用于以分割线作为中间轴的 $2\times\delta$ 区域内的所有可能的方盒。因为该区域内最多包含 $n$ 个这样的方盒，所以在该带状区域内寻找最近点对的总时间是 $O(n)$。




**分析：**

1）步骤 1 和步骤 2 花费 $O(n\log n)$时间进行排序和递归寻找最小值。

2）步骤 4 花费 $O(1)$的时间。

3）步骤 5 花费 $O(n)$的时间扫描和删除。

4）步骤 6 花费 $O(n\log n)$的时间用于排序。

5）步骤 7 花费 $O(n)$的时间用于扫描。

总复杂度为 $T(n)=O(n\log n)+O(1)+O(n)+O(n\log n)+O(n)\approx O(n\log n)$。

**问题 27　最大连续子序列和：**已知由 $n$ 个数 $A(1)\cdots A(n)$组成的序列。找出一个连续的子序列，且该子序列中元素之和为所有连续子序列中最大。

**例子：**{−2，**11**，**−4**，**13**，−5，2}→20，{1，−3，**4**，**−2**，**−1**，**6**}→7。

**解答：**将输入序列分为两个部分，最大连续子序列和可能包含以下 3 种情况：

- 情况 1：完全位于前个半序列中。
- 情况 2：完全位于后半个序列中。
- 情况 3：开始于前半个序列而终止于后半个序列。

首先对情况 3 进行分析。为了避免由于单独考虑 $n/2$ 个开始点和 $n/2$ 个结束点而导致的嵌套循环，使用两个连续的循环代替两层嵌套循环。连续的两个大小为 $n/2$ 的循环实现合并只需要线性的工作量。任何开始于前半个序列而终止于后半个序列的连续子序列必须包含前半个序列的最后一个元素和后半个序列的第一个元素。

对于前两种情况，可以使用相同的策略将序列进行更多次平分。总之，可归结为以下步骤：

1）递归计算完全位于前半个序列的最大连续子序列和。

2）递归计算完全位于后半个序列的最大连续子序列和。

3）通过两个连续的循环，计算开始于前半个序列而终止于后半个序列的最大连续子序列和。

4）选择 3 个和中的最大值。

```
int MaxSumRec(int A[], int left, int right) {
        int MaxLeftBorderSum=0,MaxRightBorderSum=0,LeftBorderSum=0,RightBorderSum=0;
        int mid = left + (right - left) / 2;
        if(left == right) // 基本情况
                return A[left] > 0 ? A[left] : 0;
        int MaxLeftSum = MaxSumRec(A, left, mid);
        int MaxRightSum = MaxSumRec(A, mid + 1, right);
        for int i = mid; i >= left; i--) {
                LeftBorderSum += A[i];
                if(LeftBorderSum > MaxLeftBorderSum)
                        MaxLeftBorderSum = LeftBorderSum;
        }
        for (int j = mid + 1; j <= right; j++) {
                RightBorderSum += A[j];
                if(RightBorderSum > MaxRightBorderSum)
                        MaxRightBorderSum = RightBorderSum;
        }
        return Max(MaxLeftSum, MaxRightSum,MaxLeftBorderSum + MaxRightBorderSum);
}
int MaxSubsequenceSum(int A, int n) {
        return n > 0 ? MaxSumRec(A, 0, n – 1) : 0;
}
```

递归中基本情况的开销为 $O(1)$。程序包含两个递归调用以及计算情况 3 的最大和的线性运算时间，因此递归式为：

$$T(1)=1$$

$$T(n)=2T(n/2)+n$$

根据分治法主定理，时间复杂度为 $T(n)=O(n\log n)$。

**问题 28** 已知包含 $2n$ 个整数的数组 $a1\ a2\ a3\cdots an\ b1\ b2\ b3\cdots bn$。不使用额外的内存空间将数组打乱成如下形式 $a1\ b1\ a2\ b2\ a3\ b3\cdots an\ bn$。

**解答：** 首先考虑如下例子(关于蛮力法详参见第 11 章)

1) 假设数组为 $a1\ a2\ a3\ a4\ b1\ b2\ b3\ b4$。

2) 将数组分成两部分，$a1\ a2\ a3\ a4$ ： $b1\ b2\ b3\ b4$。

3) 从中心进行元素交换，$a3\ a4$ 与 $b1\ b2$ 互换，得到：$a1\ a2\ b1\ b2\ a3\ a4\ b3\ b4$。

4) 将 $a1\ a2\ b1\ b2$ 分成 $a1\ a2$ ： $b1\ b2$；$a3\ a4\ b3\ b4$ 分成 $a3\ a4$ ： $b3\ b4$。

5) 对于每一个子数组围绕中心进行元素交换，得到：$a1\ b1\ a2\ b2$ 和 $a3\ b3\ a4\ b4$。

注意，该算法只能够处理 $n=2^i(i=0, 1, 2, 3$ 等)的情况。在上述例子中，$n=2^2=4$，能够容易地通过递归将数组分成两部分。在递归调用前，围绕中心进行元素交换的基本思想是产生更小规模的问题。如果元素具有某种特殊的性质，如可以根据元素本身的值计算其新位置，那么可以设计一个线性时间复杂度的算法，这属于散列(hashing)技术。

```
void ShuffleArray(int A[], int l, int r) {
                //数组中心
                int c = (l + r)/2, q = 1 + (l + c)/2;
                if(l == r) //当数组只有一个元素时的基本情况
                        return;
                for (int k = 1, i = q; i <= c; i++, k++) {
                        //围绕中心交换元素
                        int tmp = A[i];    A[i] = A[c + k];    A[c + k] = tmp;
                }
```

```
        ShuffleArray(A, l, c);              //对左半部分进行递归调用
        ShuffleArray(A, c + 1, r); );       //对右半部分进行递归调用
}
```

时间复杂度为 $O(n\log n)$。

**问题 29**　设计一个计算 $k^n$ 的算法，并讨论其复杂度。

**解答：** 计算 $k^n$ 的一个简单算法是，从 1 开始，每次乘 $k$ 直到 $k^n$。该方法包含 $n-1$ 次乘法。假设乘法仅需要常数时间，则复杂度为 $O(n)$。

但是，存在一个更快的算法来计算 $k^n$，例如，

$$9^{24} = (9^{12})^2 = ((9^6)^2)^2 = (((9^3)^2)^2)^2 = (((9^2.9)^2)^2)^2$$

计算一个数的平方需要一次乘法，按此计算，计算 $9^{24}$ 仅需要 5 次乘法，而不是 23 次。

```
int Exponential(int k, int n) {
    if (k == 0)
        return 1;
    else{
        if (n%2 == 1){
            a = Exponential(k, n-1);
            return a*k;
        }
        else{
            a= Exponential(k, n/2);
            return a*a;
        }
    }
}
```

假设 $T(n)$是计算 $k^n$ 所需要的乘法次数，为了简单起见，令 $k=2^i$，$i\geqslant 1$。

$$T(n) = T\left(\frac{n}{2}\right)+1$$

根据分治法主定理，$T(n)=O(\log n)$。



Chapter 19

# 第 19 章

# 动态规划算法

## 19.1 引言

本章将试图求解那些采用其他技术(如分治法和贪婪法)无法获得最优解的问题。动态规划(Dynamic Programming，DP)是一项虽简单但较难掌握的技术。一个容易识别和求解 DP 问题的方法是通过求解尽可能多的问题。“Programming”一词并不是指编程，而是表示填充表格(类似于线性规划)。

## 19.2 动态规划策略的定义

动态规划和备忘录共同作用。动态规划与分治法的主要区别是：对于后者，子问题是相互独立的，而在动态规划中子问题可能是重叠的，通过使用备忘录(用一个表来保存已解决子问题的答案)，对于大部分问题，动态规划能够将待求解问题的复杂度由指数级降低为多项式级(如 $O(n^2)$、$O(n^3)$等)。动态规划主要包含以下两个部分：

- 递归：递归求解子问题。
- 备忘录：将已计算的值存储在表中。

动态规划 = 递归 + 备忘录

## 19.3 动态规划策略的性质

以下两个动态规划的性质可用于判断动态规划方法是否适用于给定的问题：

- 最优子结构：问题的最优解包含其子问题的最优解。
- 子问题重叠：递归求解过程中包含少量不同子问题的多次重复计算。

## 19.4 动态规划的适用范围

与贪婪法和分治法一样，动态规划方法也不能求解所有问题。有些问题无法用任何

算法技术(贪婪法、分治法和动态规划法)解决。

动态规划与直接递归的区别在于对递归调用的备忘。如果子问题是相互独立的，不存在重叠，那么备忘录功能没有任何作用，因此动态规划不是一种适用于所有问题的方法。

## 19.5　动态规划的实现方法

对于动态规划问题，有两种基本的实现方法：

- 自底向上动态规划法
- 自顶向下动态规划法

### 1. 自底向上动态规划法

该方法从最小的可能输入参数值开始对函数进行调用，然后逐步增大参数值来计算其返回值。在计算返回值时，将所有已计算的值存储在表中(备忘录)。当对较大的参数进行计算时，就可以利用已经计算的小参数的值。

### 2. 自顶向下动态规划法

该方法将问题分解为一系列子问题，对每一个子问题求解，记住问题的解，以备后用。此外，递归函数的最终操作是保存每个已计算的值，而最先的操作是判断是否存在已计算的值。

### 3. 自底向上与自顶向下的比较

对于自底向上方法，程序员必须选择对哪个值进行计算，并决定计算的顺序。所有可能涉及的子问题都需要事先求解，然后在此基础上得到更大问题的解。而对于自顶向下方法，保存代码的递归结构，但避免不必要的重复计算。将问题分解为子问题，子问题求解后将子问题的解保存，以备后用。

**注意：一些问题可以同时用这两种方法进行求解，后续小节中将给出这样的实例。**

## 19.6　动态规划算法的例子

- 许多字符串算法，如最长公共子序列、最长递增子序列、最长公共子串、编辑距离等。
- 关于图的有效求解算法：寻找图中最短距离的 Bellman-Ford 算法、Floyd 的所有顶点间最短路径算法等。
- 链矩阵乘法。
- 子集和。
- 0/1 背包问题。
- 旅行商问题等。

## 19.7　动态规划思想

在讨论问题前，首先通过例子来理解动态规划是如何工作的。

### 1. 斐波那契(Fibonacci)数列

在斐波那契数列中，当前数是前两个数之和，其定义如下：

$$\begin{aligned} \mathrm{Fib}(n) &= 0 && n = 0 \\ &= 1 && n = 1 \\ &= \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) && n > 1 \end{aligned}$$

递归实现如下：

```
int RecursiveFibonacci(int n) {
        if(n == 0) return 0;
        if(n == 1) return 1;
        return RecursiveFibonacci(n -1) + RecursiveFibonacci(n -2);
}
```

上述实现的复杂度递归表达式为：

$$T(n) = T(n-1) + T(n-2) + 1 \approx \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 2^n = O(2^n)$$

**注意：**上式的证明见第 1 章。

**备忘录如何作用？**调用 fib(5)产生调用树，该树对同一个参数值的函数调用多次：

fib(5)

fib(4) + fib(3)

(fib(3) + fib(2)) + (fib(2) + fib(1))

((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

在上例中，fib(2)被计算了 3 次(子问题的重叠)。如果 $n$ 比较大，那么将有更多 fib 函数(子问题)的值被重复计算，从而导致算法的复杂度为指数级。可以通过保存先前计算的值来避免重复求解同一个子问题，并降低算法的复杂度。

备忘录的作用如下：当开始执行一个递归函数时，增加一个函数参数与函数返回值的映射表。因此，如果同一参数的函数被第二次调用，那么可以简单地在表中查询相应的结果。

**改进：**现在讨论动态规划如何将问题求解算法的复杂度从指数级降为多项式级。正如之前描述的，有两种方法可以求解。第一种是自底向上：从输入的最小参数值开始，逐步构建更大参数值的解。

```
int fib[n];
int fib(int n) {
        if(n == 0 || n == 1) return 1;
        fib[0] = 1;
        fib[1] = 1;
        for (int i = 2; i < n; i++)
                fib[i] = fib[i - 1] + fib[i - 2];
        return fib[n - 1];
}
```

另一种方法是自顶向下。在该方法中，保留递归调用，如果某个子问题的解已计算则使用该值。该方法的实现如下：

```
int fib[n];
int fibonacci( int n ) {
        if(n == 1) return 1;
        if(n == 2) return 1;
        if( fib[n] != 0)
                return fib[n] ;
        return fib[n] = fibonacci(n-1) + fibonacci(n -2) ;
}
```

**注意：**对于所有问题，可能无法用自顶向下和自底向上的算法同时进行求解。

斐波那契数列的两个实现版本都明显将复杂度降为 $O(n)$。这是因为如果子问题的返回值已经计算过，则不再求解该子问题，而是直接从表中取出对应的值。

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，主要来自表结构的开销。

**进一步改进：**进一步观察斐波那契数列可以发现：当前值仅仅是前两次计算结果之和。这意味着不需要存储所有先前的返回值，而只需要存储最后两次计算的值就能计算出当前值。具体的实现如下：

```
int fibonacci(int n) {
        int a = 0, b = 1, sum, i;
        for (i=0;i < n;i++) {
                sum = a + b;
                a = b;
                b = sum;
        }
        return sum;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**注意：**这个方法并不适用于所有问题。

**观察：**当用动态规划法求解时，试图弄清以下问题：

- 如何基于子问题对原问题进行递归定义？
- 是否能够使用表(备忘录)避免重复的计算？

### 2. 数的阶乘

下面介绍另一个例子——阶乘问题：$n!$ 是 $n$ 和 1 之间所有整数的乘积。阶乘的递归定义如下：

$$n! = n * (n-1)!$$
$$1! = 1$$
$$0! = 1$$

根据该定义可以容易实现阶乘。原问题是计算 $n!$ 的值，子问题是计算 $(n-1)!$ 的值。在递归实现中，当 $n$ 大于 1 时，函数调用自身计算 $(n-1)!$ 的值，然后乘以 $n$。基本情况为当 $n$ 等于 0 或者 1 时，函数直接返回数值 1。

```
int fact(int n) {
        if(n == 1)                          // 基本情况：0或1的阶乘是1
            return 1;
        else if(n == 0)
              return 1;
        // 递归的情况：n乘以(n-1)的阶乘
        else return n *fact(n -1);
}
```

上述实现的复杂度递归式为 $T(n) = n \times T(n-1) \approx O(n)$

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$，递归调用需要大小为 $n$ 的栈。

在上述递归关系及实现中，对于任意 $n$ 值，没有重复计算(不存在子问题重叠)，因此阶乘函数通过动态规划得不到任何改善。现在，假设需要计算一系列的阶乘 $m!$，其中 $m$ 为任意整数值。利用上述算法，每次调用能够在 $O(m)$时间内得到 $m$ 的阶乘。例如，计算 $n!$ 和 $m!$，总时间复杂度为 $O(m+n)$。

时间复杂度为 $O(m+n)$。空间复杂度为 $O(\max(m, n))$，递归调用需要的栈大小为

m 和 n 之间的较大值。

**改进：**下面探讨动态规划方法如何减少求解算法的复杂度。从上述递归定义可知，fact($n$)仅取决于 fact($n-1$)和 $n$。每一次调用 fact($n$)时，可以将之前计算的阶乘值存储在一个表中，在计算一个新的阶乘时，直接使用表中的值，具体的实现如下：

```
int facto[n];
int fact(int n) {
        // 基本情况：0或1的阶乘是1
        if(n == 1)
            return 1;
        else if(n == 0)
            return 1;
        //已计算过的情况
        else if(facto[n]!=0) return facto[n];
        // 递归的情况：n乘以(n-1)的阶乘
        else return facto[n]= n *fact(n -1);
}
```

为了简单起见，假设已经计算得到 $n!$ 的值，现在需要计算 $m!$，这时可以查询表，如果表中已存在某些与输入对应的值，则可以直接使用。如果 $m<n$，那么不需要重新计算 $m!$；如果 $m>n$，可以使用 $n!$ 值并只需要计算 $n$ 之后的阶乘。

这种实现的复杂度明显降为 $O(\max(m, n))$，因为 fact($n$)已经存在，而不需要再重复对其计算。如果将新计算的值填充到表中，那么后续的调用将会进一步降低复杂度。

时间复杂度为 $O(\max(m, n))$。空间复杂度为 $O(\max(m, n))$，用于表结构的开销。

### 3. 最长公共子序列

已知两个字符串，长度为 $m$ 的字符串 $X[X(1..m)]$和长度为 $n$ 的字符串$[Y(1..n)]$，找出最长公共子序列：在两个字符串中从左至右均出现的最长字符序列(不一定是连续块)。例如，$X$ = “ABCBDAB”，$Y$ = “BDCABA”，最长公共子序列 LCS($X$, $Y$) = {“BCBA”，“BDAB”，“BCAB”}。由此例可知，可能存在多个最优解。

**蛮力法：**一个简单的思路是检查 $X[1..m]$($m$ 是字符串 $X$ 的长度)中的每个子序列是否也是 $Y[1..n]$($n$ 是字符串 $Y$ 的长度)的一个子序列。一次检查花费 $O(n)$的时间，而 $X$ 有 $2^m$ 个子序列，那么运行时间为指数级 $O(n * 2^m)$。对于大序列，该方法是低效的。

**递归求解：**在介绍动态规划方法前，先给出递归算法，然后再加入备忘录来降低复杂度。首先从 LCS 问题的一些简单现象开始，假设有两个字符串，比如“ABCBDAB”和“BDCABA”，若将第一个字符串中的字符与第二个字符串中相应的字符用直线连接，那么任意两条直线都不相交：




从上述观察可知，$X$ 和 $Y$ 的当前字符可能匹配，也可能不匹配。假设两个首字符不同，那么这两个字符不可能都属于公共子序列，公共子序列可能包含其中一个，或者两个都不包含。一旦明确了如何处理字符串的首字符，那么后续的子问题仍然是两个更短的字符串上的 LCS 问题，因此可以通过递归方式进行求解。

LCS 问题求解过程涉及 $X$ 和 $Y$ 中的两个子序列，假设 $X$ 子序列的开始下标为 $i$，$Y$

子序列的开始下标为 $j$。$X[i..m]$是一个 $X$ 的从 $i$ 开始到末尾的子串，$Y[j..n]$是一个 $Y$ 的从 $j$ 开始到末尾的子串。

根据上述讨论，可能存在以下情况：

1）如果 $X[i]==Y[j]$，则 $1+\text{LCS}(i+1, j+1)$。

2）如果 $X[i]\neq Y[j]$，则 $\text{LCS}(i, j+1)$。　// 跳过 $Y$ 的第 $j$ 个字符

3）如果 $X[i]\neq Y[j]$，则 $\text{LCS}(i+1, j)$。　// 跳过 $X$ 的第 $i$ 个字符

对于第一种情况，如果 $X[i]$等于 $Y[j]$，则找到一个匹配对，并将其计入 LCS 的总长度中。否则需要跳过 $X$ 的第 $i$ 个字符或者 $Y$ 的第 $j$ 个字符，继续搜寻找最长公共子序列。$\text{LCS}(i, j)$的定义如下：

$$\text{LCS}(i,j)=\begin{cases}0 & i=m \text{ 或 } j=n\\ \text{Max}\{\text{LCS}(i,j+1),\text{LCS}(i+1,j)\} & X[i]\neq Y[i]\\ 1+\text{LCS}[i+1,j+1] & X[i]==Y[j]\end{cases}$$

最长公共子序列有许多应用。在网页搜索中，如果需要找出将一个单词修改为另一个单词所需要的最小变化数，这里变化可以插入、删除或者替换一个字符。

```
// 第一次调用: LCSLength(X, 0, m-1, Y, 0, n-1);
int LCSLength( int X[], int i, int m, int T[], int j, int n ) {
        if (i == m || j == n)
                return 0;
        else if (X[i] == Y[j]) return 1 + LCSLength(X, i+1, m, Y, j+1, n);
        else return max( LCSLength(X, i+1, m, Y, j, n),  LCSLength(X, i, m, Y, j+1, n));
}
```

该算法可以正确求解最长公共子序列问题，但非常耗时。例如，如果两个字符串没有匹配的字符，那么算法总是执行最后一行代码，当 $m$ 等于 $n$ 时，复杂度接近于 $O(2^n)$。

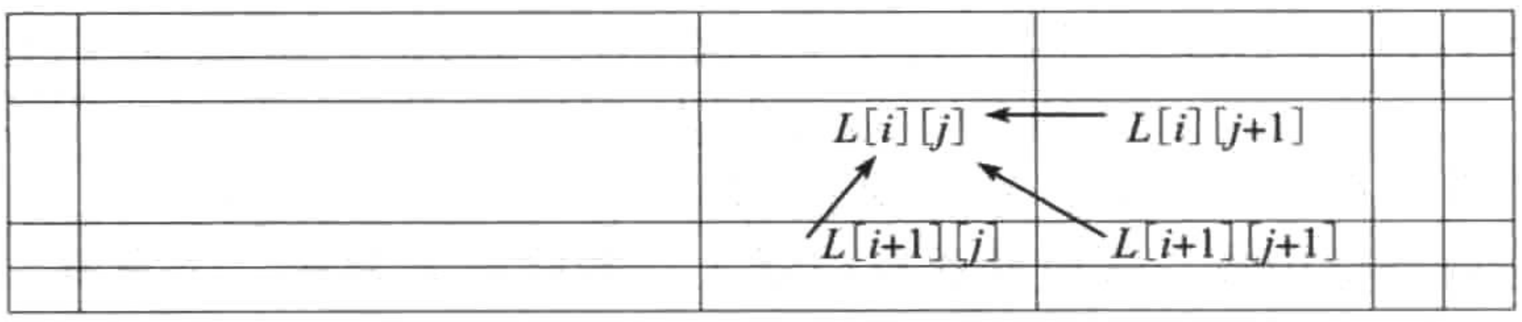
**动态规划法求解：增加备忘录。**递归求解的问题是子问题被调用了多次，其中子问题对应于 LCSLength 函数的一次调用，参数为 $X$ 和 $Y$ 的两个后缀，因此共有$(i+1)(j+1)$个可能的子问题(一个相对较小的数)。但如果有 $2^n$ 个递归调用，那么这些子问题中的某一些必然被反复求解。

在需要求解某个子问题时，动态规划法检测该子问题以前是否已经被解决，因此有些计算是查询而不是再次求解。本问题动态规划算法最直接的实现方式只需要把递归代码稍做修改，具体如下所示。

```
int LCS[1024][1024];
int LCSLength( int X[], int m, int Y[], int n ) {
        // 基本情况
for( int i = 0; i <= m; i++ )
        LCS[i][n] = 0;
for( int j = 0; j <= n; j++ )
        LCS[m][j] = 0;
for( int i = m - 1; i >= 0; i) {
        for( int j = n - 1; j >= 0; j) {
                LCS[i][j] = LCS[i + 1][j + 1]; // 把 X[i]与 Y[j] 相匹配
                if( X[i] == Y[j] )
                        LCS[i][j]++; // 匹配了一对

                // 其他两种情况——插入一个间隙
```

```
                    if(LCS[i][j + 1] > LCS[i][j] )
                        LCS[i][j] = LCS[i][j + 1];
                    if(LCS[i + 1][j] > LCS[i][j] )
                        LCS[i][j] = LCS[i + 1][j];
                }
            }
            return LCS[0][0];
    }
```

首先，创建一个行和列都比两个字符串长度大 1 的 LCS 表，并填充初值(递归中的基本情况)，然后用循环的方式迭代地填充表中的各个单元。该方法类似向后递归，或者自底向上法。




LCS[$i$][$j$]的值取决于 3 个值(LCS[$i$+1][$j$+1]、LCS[$i$][$j$+1]和 LCS[$i$+1][$j$])，其中均选择 $i$ 或 $j$ 中较大的值。按照 $i$ 和 $j$ 递减的顺序填充该表，这样能够保证当需要填充 LCS[$i$][$j$]时，它所依赖的 3 个单元的值是已知的。

时间复杂度为 $O(mn)$，因为 $i$ 从 1～$m$ 依次取值，$j$ 从 1～$n$ 依次取值。空间复杂度为 $O(mn)$。

**注意：**上述讨论所基于的假设是，LCS($i$，$j$)是 $X[i..m]$和 $Y[j..n]$的最长公共子序列的长度。只需要将 LCS($i$，$j$)的定义改为 $X[1..i]$和 $Y[1..j]$的最长公共子序列的长度就可以解决问题。

**输出子序列：**上述算法给出的是最长子序列的长度，而不是实际的最长子序列。为了得到该序列，可以对表进行追踪。从单元(0，0)开始，已知 LCS[0][0]的值是其相邻 3 个单元中的最大值，所以只要简单地重算一次 LCS[0][0]的值，就知道哪个单元具有最大值。然后移动到该单元((1，1)、(0，1)和(1，0)其中之一)，重复上述过程直至到达表的边界。每次经过 $X[i]==Y[j]$的单元($i$，$j$)，则意味着两个字符相匹配并输出 $X[i]$。最后，在 $O(mn)$时间内可以输出最长公共子序列。

输出最长公共子序列的另一种方法是为每一个单元维护一个单独的表，指明是由哪个方向的单元值计算得到该单元的值。最后，再次从单元(0，0)出发，沿着表中记录的方向直至表的对角即可。

从上述这些例子中，我们应该对动态规划方法的思想有所了解。接下来将介绍更多用动态规划法来轻松求解的问题实例。

**注意：**递归是动态规划算法的重要组成部分。如果已知递归的形式，则将其编码实现是一件容易的事。因此，对于下面将要介绍的问题，本书着重介绍如何得到其递归式。

## 19.8 动态规划的相关问题

**问题 1** 给出下面递归式的代码实现。

$$T(0) = T(1) = 2$$

$$T(n)=\sum_{i=1}^{n-1}2\times T(i)\times T(i-1)\quad n>1$$

**解答：**上述给定递归式的代码实现如下：

```
int f(int n) {
        int sum = 0;
        if(n==0 || n==1)
                return 2;                       //基本情况
        for(int i=1; i < n;i++)                 //递归情况
                sum += 2 * f(i) * f(i-1);
        return sum;
}
```

**问题 2**　能否利用动态规划的备忘录来改进问题 1 的算法？

**解答：可以。**在给出相应算法前，先来了解函数值是如何计算的。

$$T(0)=T(1)=2$$

$$T(2)=2*T(1)*T(0)$$

$$T(3)=2*T(1)*T(0)+2*T(2)*T(1)$$

$$T(4)=2*T(1)*T(0)+2*T(2)*T(1)+2*T(3)*T(2)$$

上述计算过程清晰地表明，对于相同的输入参数，存在许多重复的计算，可以使用表结构来避免这些重复的计算，具体实现如下：

```
int f(int n) {
        T[0] = T[1] = 2;
        for(int i=2; i <= n; i++) {
                T[i] = 0;
                for (int j=1; j < i; j++)
                        T[i] +=2 * T[j] * T[j-1];
        }
        return T[n];
}
```

时间复杂度为 $O(n^2)$，由于存在两个 for 循环。

空间复杂度为 $O(n)$，用于表结构。

**问题 3**　能否进一步降低问题 2 的复杂度？

**解答：可以。**由于所有的子问题仅依赖于前面的计算结果，所以算法可以修改如下：

```
int f(int n) {
        T[0] = T[1] = 2;
        T[2] = 2 * T[0] * T[1];
        for(int i=3; i <= n; i++)
                T[i]=T[i-1] + 2 * T[i-1] * T[i-2];
        return T[n];
}
```

时间复杂度为 $O(n)$，因为仅有一个 for 循环。

空间复杂度为 $O(n)$。

**问题 4　最大连续子序列和：**已知包含 $n$ 个元素的数组，设计算法找出一个连续子序列 $A(i)\cdots A(j)$，使其元素之和最大。

**例子：**{−2，**11**，**−4**，**13**，−5，2}→20；{1，−3，**4**，**−2**，**−1**，**6**}→7

**解答：**

输入：包含 $n$ 个元素的数组 $A(1)\cdots A(n)$。

目标：如果不包含任何负数，则答案为给定数组的所有元素之和；如果存在负数，则最大化元素之和(连续元素的和中可能包含负数)。

一种简单的蛮力法是列出所有可能的序列和，然后选择值最大的一个。

```
int MaxContigousSum(int A[], in n) {
        int maxSum = 0;
        for(int i = 0; i < n; i++)                    // 每一个可能开始的点
                for(int j = i; j < n; j++)    {       // 每一个可能结束的点
                        int currentSum = 0;
                        for(int k = i; k <= j; k++)
                                currentSum += A[k];
                        if(currentSum > maxSum)
                                maxSum = currentSum;
                }
        }
        return maxSum;
}
```

时间复杂度为 $O(n^3)$。

空间复杂度为 $O(1)$。

**问题 5** 是否能够降低问题 4 的复杂度？

**解答：可以。**仔细观察可以发现，如果已经求得子序列 $i$，…，$j-1$ 的和，那么仅需要再进行一次加法就能得到子序列 $i$，…，$j$ 的和。但是问题 4 给出的算法忽略了该特点。如果利用该特点，就能够得到一个改进的复杂度为 $O(n^2)$的算法。

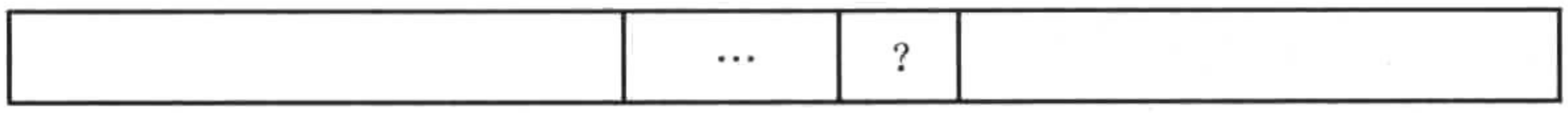
```
int MaxContigousSum(int A[], int n) {
        int maxSum = 0;
        for( int i = 0; i < n;   i++) {
                int currentSum = 0;
                for( int j = i; j < n; j++)    {
                        currentSum += a[j];
                        if(currentSum > maxSum)
                                maxSum = currentSum;
                }
        }
        return maxSum;
}
```

时间复杂度为 $O(n^2)$。

空间复杂度为 $O(1)$。

**问题 6** 能否利用动态规划来求解问题 4？

**解答：可以。**为了简单起见，令 $M(i)$表示以 $i$ 为末端的所有窗口的最大和。给定数组 $A$，基于第 $i$ 个元素的选取方式，定义递归式。



为了找出最大和，可以进行下面两种操作，选择其中可以获得最大和的一种操作来执行。

- 通过增加 $A[i]$来扩充原来的和。
- 开始一个始于元素 $A[i]$的新窗口。

$$M(i) = \text{Max}\begin{cases} M(i-1) + A[i] \\ 0 \end{cases}$$

其中，$M(i-1)+A[i]$表示通过增加 $A[i]$来扩充原来的和，0 表示始于 $A[i]$的新窗口。

```
int MaxContigousSum(int A[], int n) {
        int M[n] = 0, maxSum = 0;
        if(A[0] > 0)
            M[0] = A[0];
        else M[0] = 0;
        for( int i = 1; i < n;   i++) {
                if( M[i-1] + A[i] > 0)
                        M[i] = M[i-1] + A[i];
                else
                        M[i] = 0;
        }
        maxSum = 0;
        for( int i = 0; i < n;   i++) {
                if(M[i] > maxSum) maxSum = M[i];
        }
        return maxSum;
}
```

时间复杂度为 $O(n)$。

空间复杂度为 $O(n)$，用于表结构。

**问题 7**　是否有其他方法求解问题 4?

**解答：可以。**不需要使用动态规划法(无需备忘录)也可以求解该问题。该算法有一些小技巧。一个简单的方法是寻找数组中所有连续正数段(sumEndingHere)，并在所有正数段中跟踪最大和连续段(sumSoFar)。当找到一个正数段时，将它与 sumSoFar 比较，如果大于 sumSoFar，则更新 sumSoFar。上述描述对应的代码如下：

```
int MaxContigousSum(int A[], int n) {
        int sumSoFar = 0, sumEndingHere = 0;
        for(int i = 0; i < n; i++)    {
                sumEndingHere = sumEndingHere + A[i];
                if(sumEndingHere < 0) {
                        sumEndingHere = 0;
                        continue;
                }
                if(sumSoFar < sumEndingHere)
                        sumSoFar = sumEndingHere;
        }
        return sumSoFar;
}
```

**注意：**如果输入数组全部由负数组成，则算法不能正确运行。如果所有数都是负数，那么返回值为 0。为了避免出现这种情况，需要在算法实际运行前增加一个额外的检查，

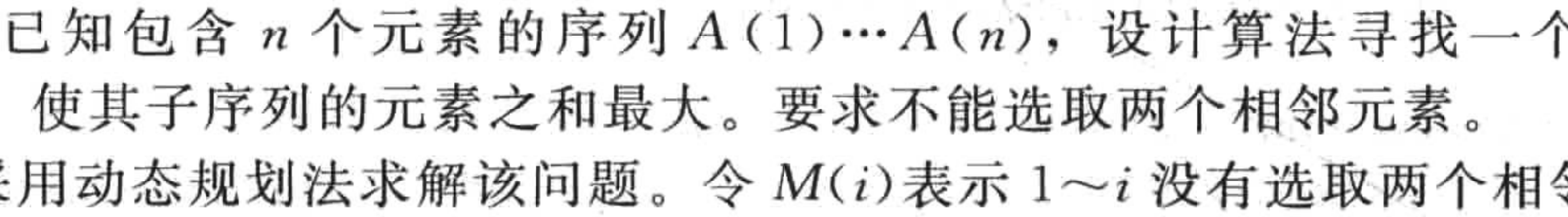
即如果所有元素都为负数，则返回这些元素中的最大值(或者绝对值最小的值)。

时间复杂度为 $O(n)$，因为该算法仅需要扫描一次。空间复杂度为 $O(1)$。

**问题 8** 在问题 7 的求解中，$M(i)$表示所有以 $i$ 为末端的窗口的最大和，能否令$M(i)$为所有以 $i$ 为起始端，$n$ 为末端的窗口的最大和？

**解答：可以。**为了简单起见，令 $M(i)$为所有始于 $i$ 的窗口的最大和。

给定数组 $A$，基于第 $i$ 个元素的选取方式，定义递归式。




为了找出最大和，可以进行下面两种操作，选择其中可以获得最大和的一种操作来执行。

- 增加 $A[i]$来扩充原来的和。
- 开始一个始于元素 $A[i]$的新窗口。

$$M(i) = \text{Max}\begin{cases} M(i+1)+A[i] & M(i+1)+A[i] > 0 \\ 0 & M(i+1)+A[i] \leqslant 0 \end{cases}$$

其中，$M(i+1)+A[i]$表示增加 $A[i]$来扩充原来的和，0 表示始于 $A[i]$的新窗口。

时间复杂度为 $O(n)$。

空间复杂度为 $O(n)$，需要一个表结构。

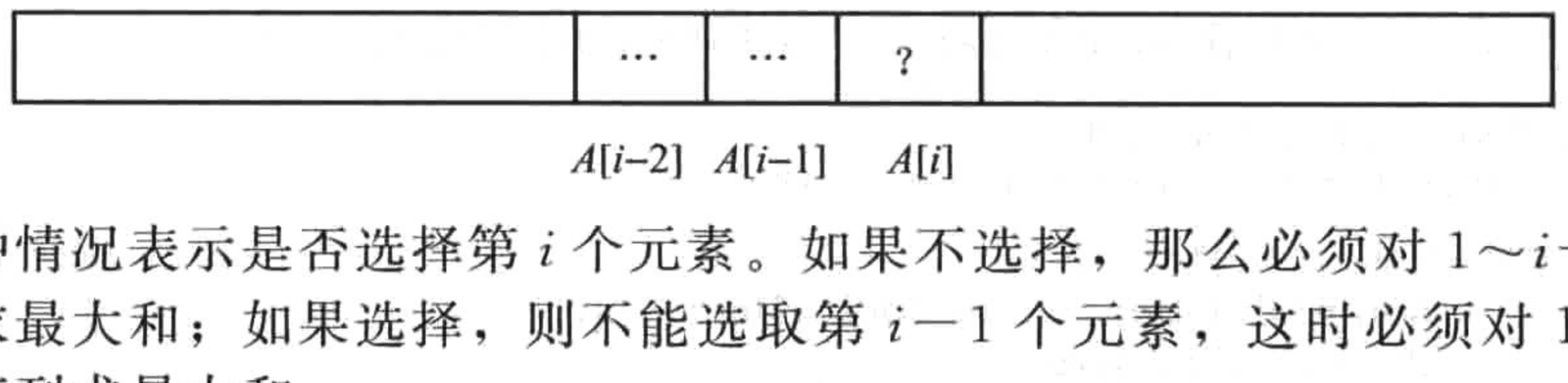
**注意：**对于复杂度为 $O(n\log n)$的算法，请参见第 18 章。

**问题 9** 已知包含 $n$ 个元素的序列 $A(1)\cdots A(n)$，设计算法寻找一个连续子序列 $A(i)\cdots A(j)$，使其子序列的元素之和最大。要求不能选取两个相邻元素。

**解答：**采用动态规划法求解该问题。令 $M(i)$表示 $1\sim i$ 没有选取两个相邻元素的子序列的最大和。当计算 $M(i)$时，需要确定是否选择第 $i$ 个元素。有两种可能的选择，基于此可得到以下递归式：

$$M(i) = \begin{cases} \text{Max}\{A[i]+M(i-2), M(i-1)\} & i > 2 \\ A[1] & i = 1 \\ \text{Max}\{A[1], A[2]\} & i = 2 \end{cases}$$

给定数组 $A$，基于第 $i$ 个元素的选取方式，定义递归式。




- 第一种情况表示是否选择第 $i$ 个元素。如果不选择，那么必须对 $1\sim i-1$ 之间的子序列求最大和；如果选择，则不能选取第 $i-1$ 个元素，这时必须对 $1\sim i-2$ 之间的子序列求最大和。
- 递归表示中的后两种情况是递归的基本情况。

```
int maxSumWithNoTwoContinuousNumbers(int A[], int n) {
        int M[n+1];
        M[0]=A[0];
        M[1]=(A[0]>A[1]?A[0]:A[1]);
        for(i=2, i<n; i++)
                M[i]= (M[i-1]>M[i-2]+A[i]? M[i-1]: M[i-2]+A[i]);
        return M[n-1];
}
```

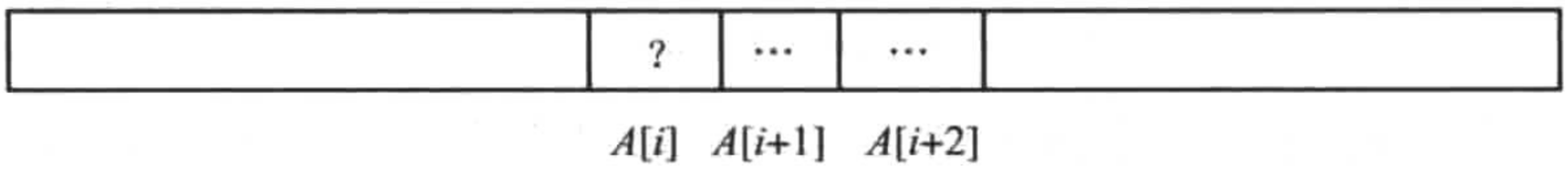
时间复杂度为 $O(n)$。

空间复杂度为 $O(n)$。

**问题 10**　在问题 9 中，$M(i)$表示从 $1\sim i$ 之间没有选择两个相邻元素的子序列的最大和，如果将 $M(i)$定义为 $i\sim n$ 之间没有选择两个相邻元素的子序列的最大和，那么是否依然可以求解该问题？

**解答：可以。**令 $M(i)$表示 $i\sim n$ 之间未选择两个相邻元素的子序列的最大和。

给定数组 $A$，基于第 $i$ 个元素的选取方式，定义递归式。




与问题 9 算法类似，递归式为：

$$M(i)=\begin{cases}\text{Max}\{A[i]+M(i+2),M(i+1)\} & i>2\\ A[1] & i=1\\ \text{Max}\{A[1],A[2]\} & i=2\end{cases}$$

- 第一种情况表示是否选择第 $i$ 个元素。如果不选择，则最大化 $i+1\sim n$ 之间的子序列的和；如果选择，则不能选择第 $i+1$ 个元素，这时需要最大化 $i+2\sim n$ 之间的子序列的和。
- 递归表示中的后两种情况是递归的基本情况。

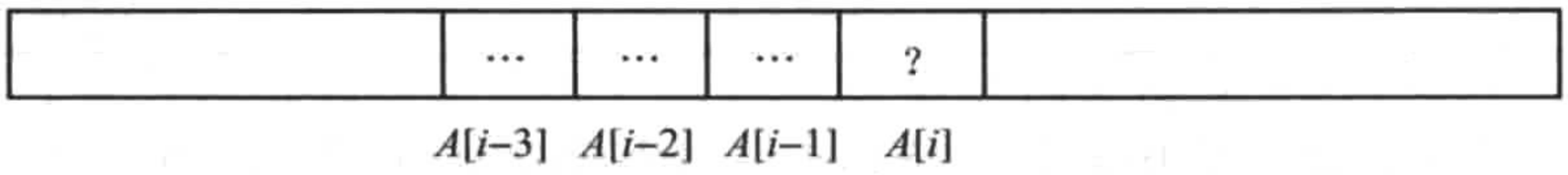
时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 11**　已知包含 $n$ 个元素的序列 $A(1)\cdots A(n)$，设计算法找到一个连续子序列 $A(i)\cdots A(j)$，使其子序列元素之和最大。要求不能选取 3 个相邻的元素。

**解答：**令 $M(i)$表示 $1\sim i$ 之间不包含 3 个相邻元素的子序列的最大和。计算 $M(i)$时，需要确定是否选择第 $i$ 个元素，有 3 种可能的选择。基于此，可得到以下递归式：

$$M(i)=\text{Max}\begin{cases}A[i]+A[i-1]+M(i-3)\\ A[i]+M(i-2)\\ M(i-1)\end{cases}$$

给定数组 $A$，基于第 $i$ 个元素的选取方式，定义递归式。




- 该问题要求不能选取 3 个相邻的元素，那么可以选择 2 个相邻的元素，而跳过第 3 个。这对应于上式中的第一种情况，即跳过 $A[i-2]$。
- 另一个可能性是选取第 $i$ 个元素，跳过第 $i-1$ 个元素，对应于第二种情况(跳过 $A[i-1]$)。
- 第三种情况表示不选取第 $i$ 个元素，这意味着需要求解参数为 $i-1$ 的子问题。

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 12**　在问题 11 中，$M(i)$表示 $1\sim i$ 之间没有选取 3 个相邻元素的子序列的最大和。如果将 $M(i)$定义为 $i\sim n$ 之间没有选取 3 个相邻元素的子序列的最大和，那么是否依然可以求解该问题？

**解答：可以。**令 $M(i)$表示 $i\sim n$ 之间不选择 3 个相邻元素的子序列的最大和。当计算 $M(i)$时，需要确定是否选择第 $i$ 个元素。有以下几种可能性：

$$M(i) = \text{Max}\begin{cases} A[i]+A[i+1]+M(i+3) \\ A[i]+M(i+2) \\ M(i+1) \end{cases}$$

给定数组 $A$，基于第 $i$ 个元素的选取方式，定义递归式。

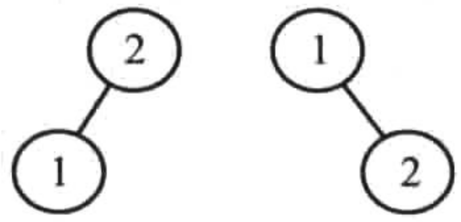
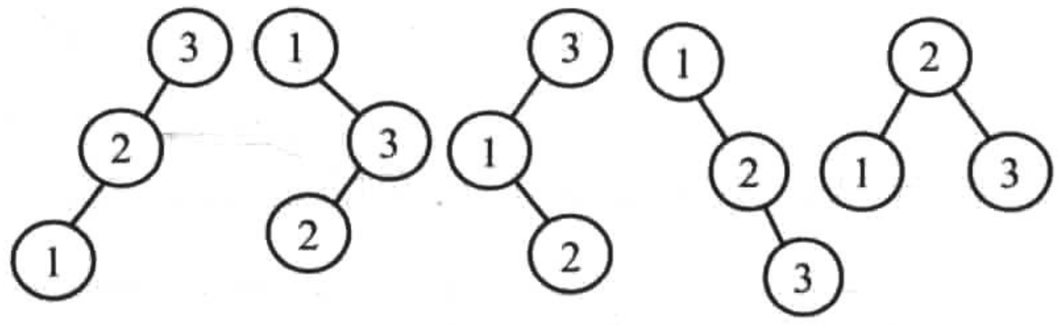



- 该问题要求不能选择 3 个相邻的元素，但是可以选择 2 个相邻的元素并跳过第 3 个，这对应于上式中的第一种情况，跳过 $A[i+2]$。
- 另一个可能性是选择第 $i$ 个元素，跳过第 $i+1$ 个元素，对应于第二种情况(跳过 $A[i+1]$)。
- 第三种情况表示不选取第 $i$ 个元素，这意味着需要求解参数为 $i+1$ 的子问题。

时间复杂度为 $O(n)$。空间复杂度为 $O(n)$。

**问题 13　卡塔兰数(Catalan Number)：** $n$ 个结点的二叉搜索树总共有多少棵？

**解答：**

| 结点数 | 树的数目 |
|---|---|
| 1 |  |
| 2 |  |
| 3 |  |

二叉搜索树(BST)是一棵树，其左子树的元素都小于根元素，而右子树的元素都大于根元素。树中每一个结点都满足这个性质。$n$ 个结点的 BST 的数目称为卡塔兰数(Catalan Number)，记为 $C_n$。例如，2 个结点的 BST 有 2 棵，3 个结点的 BST 有 5 棵。

假设将树的结点用 $1\sim n$ 进行编号。从这些结点中选择一个作为根结点，然后将小于根结点的结点划分到左子树，而将大于根结点的结点划分到右子树。由于已对结点按升序编号，所以假设第 $i$ 个结点被选为根结点，那么共有 $i-1$ 个结点位于左子树，$n-i$ 个结点位于右子树。如果 $C_n$ 表示 $n$ 个元素的卡塔兰数，那么 $C_{i-1}$ 表示左子树($i-1$ 个元素)的卡塔兰数，$C_{n-i}$ 表示右子树的卡塔兰数。由于这两个子树是相互独立的，所以只需要简单地将两个数相乘。即对于某一确定的值 $i$，其卡塔兰数等于 $C_{i-1}\times C_{n-i}$。因为共有 $n$ 个结点，所以 $i$ 有 $n$ 种选择。$n$ 个结点的卡塔兰数为：

$$C_n = \sum_{i=1}^{n} C_{i-1} \times C_{n-i}$$

对于上述简单形式的递归定义，可以设计如下计算卡塔兰数的函数：

```
int CatalanNumber( int n ) {
        if( n == 0 ) return 1;
        int count = 0;
        for( int i = 1; i <= n; i++ )
                count += CatalanNumber (i -1) * CatalanNumber (n -i);
        return count;
}
```

时间复杂度为 $O(4^n)$，证明见第 1 章。

**问题 14**　能否使用动态规划法降低问题 13 的时间复杂度？

**解答：**递归调用 $C_n$ 仅依赖于 $C_0 \sim C_{n-1}$。对于任意的 $i$，存在许多重复计算，可以维护一个表，表中记录所有已经计算的 $C_i$ 值。如果函数 CatalanNumber()的参数为 $i$，且该函数已经计算过，那么通过查表就可以避免重复计算子问题。

```
int Table[1024];
int CatalanNumber( int n ) {
        if( Table[n] ) != 1 )
                return Table[n];
        Table[n] = 0;
        for( int i = 1; i <= n; i++ )
                Table[n] += CatalanNumber( i -1) * CatalanNumber(n -i);
        return Table[n];
}
```

该实现的时间复杂度为 $O(n^2)$，因为计算 CatalanNumber($n$)，需要计算所有从 0～$n-1$之间的 CatalanNumber($i$)。每一个函数仅计算一次，花费线性的时间。

在数学上，卡塔兰数可以直接用代数式$\frac{(2n)!}{n!\ (n+1)!}$计算。

**问题 15　矩阵乘积括号化：**已知矩阵 $\boldsymbol{A}_1$、$\boldsymbol{A}_2$、…、$\boldsymbol{A}_n$，以及它们的维度，将所有矩阵相乘。如何添加括号使乘法运算的次数最少？假设采用的是标准的矩阵乘法算法而不是 Strassen 算法。

**解答：**由于矩阵乘法满足结合律，所以矩阵乘法问题存在多种可能。无论如何用括号改变矩阵乘法的运算顺序，结果都保持不变。例如，有 4 个矩阵 $\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$ 和 $\boldsymbol{D}$，可能的乘法运算过程有：

$$(\boldsymbol{ABC})\boldsymbol{D} = (\boldsymbol{AB})(\boldsymbol{CD}) = \boldsymbol{A}(\boldsymbol{BCD}) = \boldsymbol{A}(\boldsymbol{BC})\boldsymbol{D} = \cdots$$

维度大小为($p \times q$)的矩阵与($q \times r$)的矩阵相乘，需要 $pqr$ 次乘法运算。上述不同的乘积运算方式对应不同数量的乘法运算。为了选择最好的方式，可以对每一个可能的括号进行检查(蛮力法)，但这需要 $O(2^n)$的时间，效率非常低。

下面，通过动态规划法来降低复杂度，令 $M[i, j]$表示 $A_i \cdots A_j$ 相乘所需的最少乘法运算次数。

$$M[i,j] = \begin{cases} 0 & i = j \\ \text{Min}\{M[i,k] + M[k+1,j] + P_{i-1}P_kP_j\} & i < j \end{cases}$$

上述递归式表明，为了计算 $M[i, j]$，需要找出得到最少乘法运算次数的分割点 $k$。对所有可能的 $k$ 值一一验证，然后选取乘法次数最少的 $k$ 值。可以使用一个额外的表($S[i, j]$)来重构最优的放置括号的方式，并以自底向上的方式计算 $M[i, j]$和 $S[i, j]$。

```
/* P是矩阵的大小，矩阵i的维度大小为P[i-1]×P[i]。
M[i, j]为矩阵i到j乘法运算次数的最小值。
S[i, j]保存分割点，用于回溯。*/
void MatrixChainOrder(int P[], int length) {
        int n = length - 1;
        int M[n][n], S[n][n];
for (int i = 1; i <= n; i++)
        M[i][i] = 0;                    // 填充矩阵的对角线
for (int l=2; l<= n; l++)   {           // l表示的是链的长度
        for (int i=1; i<= n -l+1; i++) {
                int j = i+l-1;
                M[i][j] = MAX_VALUE;
                // 尝试所有可能的分割点 i..k 和 k..j
                for (int k=i; k<=j-1; k++) {
                        int thisCost = M[i][k] + M[k+1][j] + P[i-1]*P[k]*P[j];
                        if(thisCost < M[i][j]) {
                                M[i][j] = thisCost;
                                S[i][j] = k;
                        }
                }
        }
}
```

**有多少子问题？** 在上面的公式中，$i$ 和 $j$ 的取值范围均为 1～$n$，因此一共有 $n^2$ 个子问题和 $n-1$ 个矩阵乘法运算(因为 $\boldsymbol{A}_1 \times \boldsymbol{A}_2 \times \cdots \times \boldsymbol{A}_n$ 包含 $n-1$ 个矩阵乘法运算)。综上所述，时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^2)$。

**问题 16** 对于问题 15，能否采用贪婪法？

**解答：** 贪婪法求解该问题不能得到最优解，下面将给出一些反例。在每一步决策时贪婪法总是选取局部最好的，而不考虑对后续解的影响。因此在本例中，贪婪法总是首先结合乘法运算次数最少的矩阵，但该方法得到的括号放置方式并不是最优的。

**例子：** 考虑维度大小分别为 3×100、100×2 和 2×2 的 3 个矩阵的乘法 $\boldsymbol{A}_1 \times \boldsymbol{A}_2 \times \boldsymbol{A}_3$。基于贪婪法，加括弧的结果为：$\boldsymbol{A}_1 \times (\boldsymbol{A}_2 \times \boldsymbol{A}_3)$，乘法运算次数为 100×2×2+3×100×2=1000。但最优解为$(\boldsymbol{A}_1 \times \boldsymbol{A}_2) \times \boldsymbol{A}_3$，相应的乘法运算次数为 3×100×2+3×2×2=612。所以不能采用贪婪法求解该问题。

**问题 17 整数背包问题(允许重复物品)：** 有 $n$ 种物品，其中第 $i$ 种物品的大小为整数值 $s_i$，价值为 $v_i$。若用总容量为 $C$ 的背包装载这些物品，如何装载使得背包包含物品的价值最大。同一种物品可以放多个到背包里。

**注意：** 关于分数背包问题请参见第 17 章。

**解答：**

输入：$n$ 种物品，其中第 $i$ 种物品的大小为 $s_i$，价值为 $v_i$，并假设同一种物品的数量是无限的。

目标：用 $n$ 种物品填充容量为 $C$ 的背包，使其价值最大。

一个需要特别注意的地方是：可以不必完全填满整个背包。即，假设完全填满背包(此时大小为 $C$)得到的价值为 $V$，而不完全填满整个背包(比如此时大小为 $C-1$)的价值为 $U$，若 $V<U$，则选取第二个方案，装填背包至大小为 $C-1$。如果对于 $C-1$ 出现上述相同情况，则尝试用大小为 $C-2$ 的物品装填背包以便得到最大价值。

令 $M(j)$表示背包大小为 $j$ 时可获得的最大价值，用子问题的解递归地将其表示为：

$$M(j)=\begin{cases}\max\{M(j-1),\max\limits_{i=1\sim n}(M(j-s_i))+v_i\} & j\geqslant 1\\ 0 & j\leqslant 0\end{cases}$$

对于该问题，决策依赖于是否将第 $i$ 种的一个物品放入大小为 $j$ 的背包中。

- 如果选择第 $i$ 种物品放入背包，则将其价值 $v_i$ 计入最优解，并将背包的大小减小为 $j-s_i$。
- 如果不选择，那么检测背包大小为 $j-1$ 时是否能得到更好的解。

$M(C)$的值将包含最优解的值，通过维护和追踪“回溯点”就能够得到最优解所包含的物品列表。

时间复杂度：计算每一个 $M(j)$值需要 $O(n)$时间，共需要计算 $C$ 次。因此，总运行时间为 $\Theta(nC)$。空间复杂度为 $\Theta(C)$。

**问题 18　0-1 背包问题：**对于问题 17，如果物品不允许重复(每种物品的数量是有限的，且每种物品仅能使用 0 或 1 次)，又该如何求解？

**一个现实中的例子：**乘飞机时，对行李的重量有限制，并且乘客可以携带多种类型的物品(如笔记本等)。此时，需要选择具有最大价值的物品，即需要告诉海关选择重量较大而价值(收益)较小的物品。

**解答：**输入：大小为 $s_i$，价值为 $v_i$ 的 $n$ 个物品的集合，容量为 $C$ 的用于装载全部或部分物品的背包。

利用动态规划法求解该问题的递归式。令 $M(i,\ j)$表示将物品(1～$i$)装入容量为 $j$ 的背包可获得的最大价值，递归式如下：

$$M(i,j)=\underbrace{\mathrm{Max}\{M(i-1,j)}_{\text{不装第 }i\text{ 个物品}},\underbrace{M(i-1,j-s_i)+v_i\}}_{\text{装第 }i\text{ 个物品}}$$

因为 $i$ 取值范围为 1～$n$，$j$ 取值范围为 1～$C$，所以共有 $nC$ 个子问题。上述递归式的含义为：

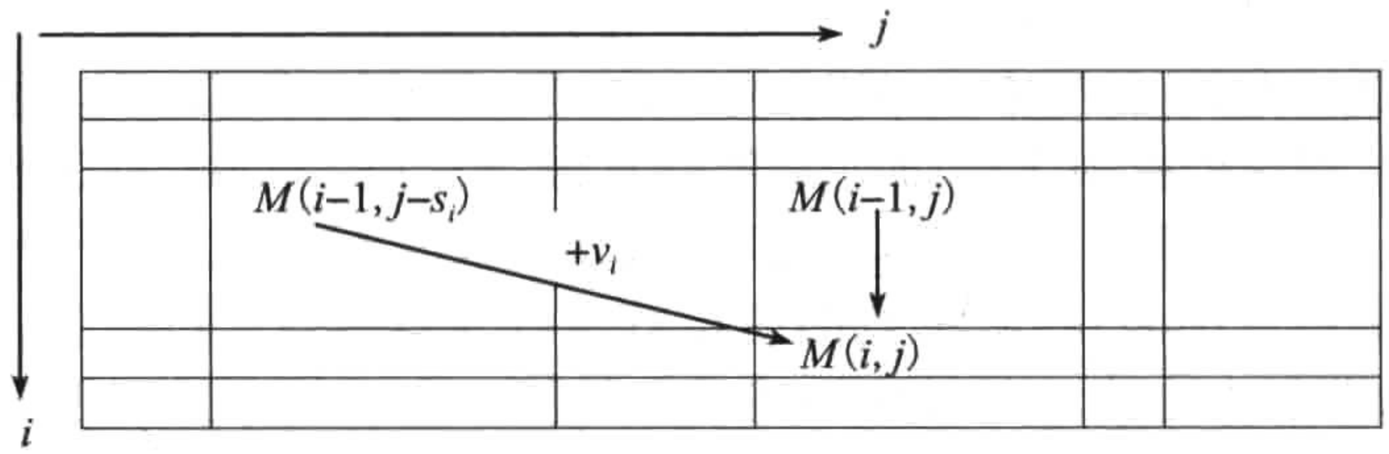
- $M(i-1,\ j)$：表示不选择第 $i$ 个物品。在该种情况下，因为没有将任何物品装入背包，所以子问题中背包的大小保持不变，但移除第 $i$ 个物品。
- $M(i-1,\ j-s_i)+v_i$：表示选择第 $i$ 件物品。因为将第 $i$ 件物品放入背包，所以子问题中背包的大小减少为 $j-s_i$，同时将其价值 $v_i$ 加入最优解。

在得到所有的 $M(i,\ j)$值后，最优的目标值可由$\mathrm{Max}\limits_{j}\{M(n,\ j)\}$计算得到。这是因为不能确定背包容量为多少时可以获得最优解。

时间复杂度：$O(nC)$，因为需求解 $nC$ 个子问题，每一个子问题耗时 $O(1)$。

空间复杂度：$O(nC)$，整数背包问题仅为 $O(C)$。

下图显示了最优解的构造过程，它有助于更好地理解该算法。矩阵的大小为 $M$。

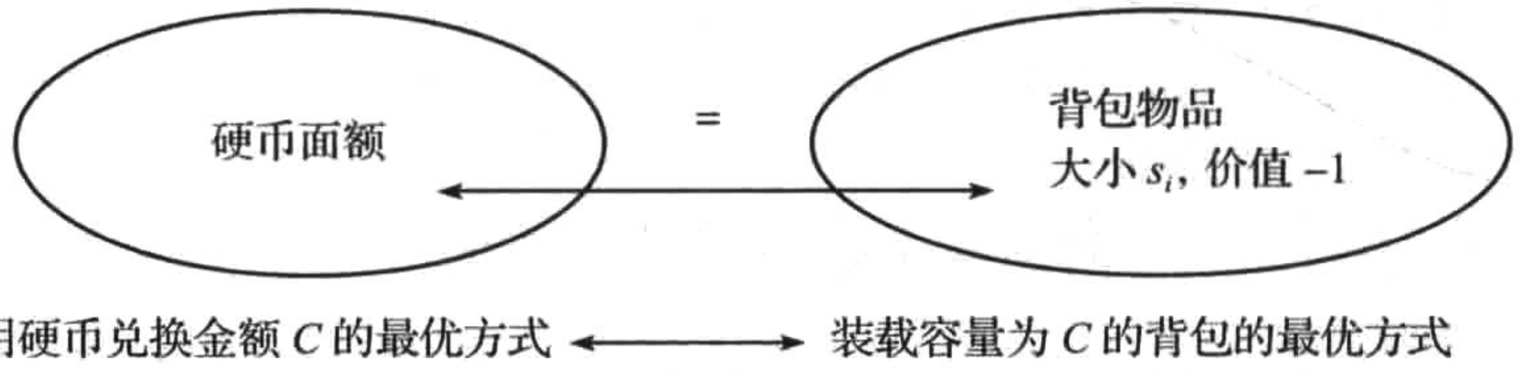



$M(i, j)$为 $M(i-1, j)$与 $M(i-1, j-s_i)+v_i$ 的最大值，这两个值($M(i-1, j)$和 $M(i-1, j-s_i)+v_i$)出现在前面的行和列中，因此可通过查询表中前一行的两个值得到$M(i, j)$。

**问题 19　找零问题：**有 $n$ 种面额的硬币，其面额存在下列关系 $v_1<v_2<\cdots<v_n$(整数)。假设 $v_1=1$，从而保证对于任意金额 $C$ 都能够进行兑换。设计一个用尽可能少的硬币兑换金额 $C$ 的算法。

**解答：**该问题与整数背包问题相同，只是在本例中换成面额为 $v_i$ 的硬币。可以构造一个背包问题的实例：每件物品的大小 $s_i$ 等同于硬币的面额 $v_1$，每件物品的价值都为 $-1$。

容易理解用最少的硬币兑换金额 $C$ 的最优解与填充容量为 $C$ 的背包相同，这是因为每件物品价值为 $-1$，所以这时背包算法使用尽可能少的物品(对应于尽可能少的硬币)，使价值最大。




现在定义递归式，令 $M[j]$表示兑换金额 $j$ 所需要的最少硬币数。

$$M(j)=\min_i\{M(j-v_i)\}+1$$

上式表明，如果第 $i$ 种面额的硬币是最后加入解的硬币，那么包含该枚硬币的最优解为兑换金额 $j-v_i$ 的最优找零数，然后加上该枚面额为 $v_i$ 的硬币。

```
int Table[128];                    //初始化
int MakingChange(int n) {
        if(n < 0)
            return -1;
        if(n == 0)
            return 0;
        if(Table[n] != -1)
                return Table[n];
        int ans = -1;
        for ( int i = 0 ; i < num_denomination ; ++i )
                ans = Min( ans , MakingChange(n - denominations [ i ] ) ) ;
        return Table[ n ] = ans + 1 ;
}
```

时间复杂度为 $O(nC)$。因为需要求解 $C$ 个子问题，所以每一个子问题需要在 $n$ 项中取最小值。

空间复杂度为 $O(C)$。

**问题 20　最长递增子序列：**已知包含 $n$ 个数的序列 $A_1 \cdots A_n$，找出元素严格递增的最长子序列(不一定连续)。

**解答：**

输入：由 $n$ 个数组成的序列 $A_1 \cdots A_n$。

目标：找出由其子集构成的一个子序列，不一定连续，但子序列中的元素必须严格递增，同时包含尽可能多的元素。

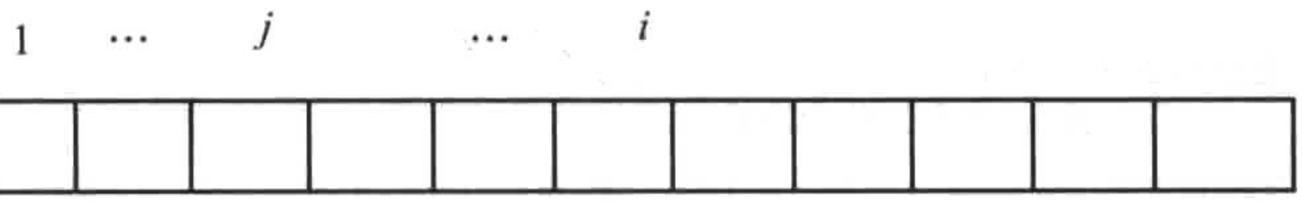
例如，若序列为(5，6，2，3，4，1，9，9，8，9，5)，那么(5，6)、(3，5)、(1，8，9)都是递增子序列，其中最长的子序列为(2，3，4，8，9)，设计一个算法找出这样的最长子序列。

首先，重点放在找出最长子序列的算法。然后，通过追踪表结构来输出该子序列。因此，第一步要做的是找出递归式。首先确定递归的基本情况。如果输入序列中仅有一个元素，那么该问题无需求解，仅需要返回该元素。对于其他序列，从第一个元素 $A[1]$ 开始。由于已知最长递增子序列 LIS 的第一个元素，所以接下来需要找出第二个元素，从序列中取 $A[2]$。如果 $A[2]$大于 $A[1]$，那么将 $A[2]$放入 LIS 作为第二个元素；否则，放弃$A[2]$，这时 LIS 为包含一个元素的子序列($A[1]$)。

现在将上述讨论推广到第 $i$ 个元素。令 $L(i)$表示由 $A[1]$开始且以 $A[i]$结束的最优子序列。得到以 $A[i]$结束的严格递增子序列的最好方法是扩展 $i$ 之前某个位置 $j$ 的子序列，由此可以得到以下递归式：

$$L(i) = \max_{j<i \text{且} A[j]<A[i]} \{L(j)\} + 1$$

上述递归式表明，为了得出 $L(i)$，必须选择 $i$ 之前的具有最长子序列的位置 $j$，递归式中的 1 表示加入第 $i$ 个元素。




在找出所有位置的最长子序列后，选择其中的最大值作为最优解：

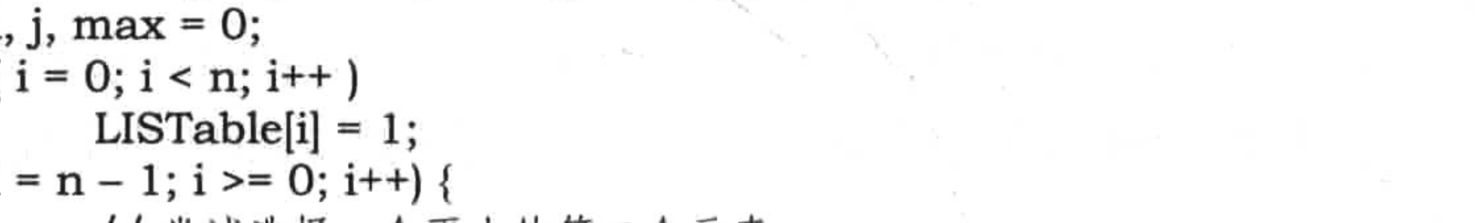
$$\max_{i}\{L(i)\}$$

```
int LISTable [1024];
int LongestIncreasingSequence( int A[], int n ) {
        int i, j, max = 0;
        for ( i = 0; i < n; i++ )
                LISTable[i] = 1;
        for ( i = 0; i< n; i++ ) {
                for ( j = 0; j < i; j++ ) {
                        if( A[i] > A[j] && LISTable[i] < LISTable[j] + 1 )
                                LISTable[i] = LISTable[j] + 1;
                }
        }
        for ( i = 0; i < n; i++)
                if( max < LISTable[i] )
                        max = LISTable[i];
        return max;
}
```

时间复杂度为 $O(n^2)$，因为有两层 for 循环。空间复杂度为 $O(n)$，用于表结构。

**问题 21　最长递增子序列：**在问题 20 中，$L(i)$表示从 $A[1]$开始到 $A[i]$结束的最优子序列。下面将此定义改为：$L(i)$表示从 $A[i]$开始到 $A[n]$结束的最优子序列。该定义下是否存在求解方法？

**解答：可以。**逻辑和推理过程与问题 20 基本一致。




令 $L(i)$表示从 $A[i]$开始到 $A[n]$结束的最优子序列。计算从位置 $i$ 开始的严格递增子序列的最佳方式是扩展在 $i$ 之后某个位置 $j$ 的子序列，此时其递归式如下：

$$L(i) = \max_{i<j且A[i]<A[j]} \{L(j)\} + 1$$

该递归式的计算必须选择 $i$ 之后的具有最长子序列的位置 $j$。递归式中的 1 表示加入第 $i$ 个元素。在计算出所有位置的最长子序列后，选择其中的最大值作为最优解：

$$\max_{i}\{L(i)\}$$

```
int LISTable [1024];
int LongestIncreasingSequence( int A[], int n ) {
        int i, j, max = 0;
        for ( i = 0; i < n; i++ )
                LISTable[i] = 1;
        for(i = n – 1; i >= 0; i++) {
                // 尝试选择一个更大的第二个元素
                for( j = i + 1; j < n; j++ ) {
                        if( A[i] < A[j] && LISTable [i] < LISTable [j] + 1)
                                LISTable[i] = LISTable[j] + 1;
                }
        }
        for ( i = 0; i < n; i++ ) {
                if( max < LISTable[i] )
                        max = LISTable[i];
        }
        return max;
}
```

时间复杂度为 $O(n^2)$，因为有两层 for 循环。空间复杂度为 $O(n)$，用于表结构。

**问题 22　**是否存在求解问题 21 的其他方法？

**解答：存在。**其中一个方法是将给定序列进行排序，并将排序后的序列存储到另一个数组中，然后找出两个数组的“最长公共子序列”（LCS）。该方法的复杂度为 $O(n^2)$。对于 LCS 问题参见本章前面各节。

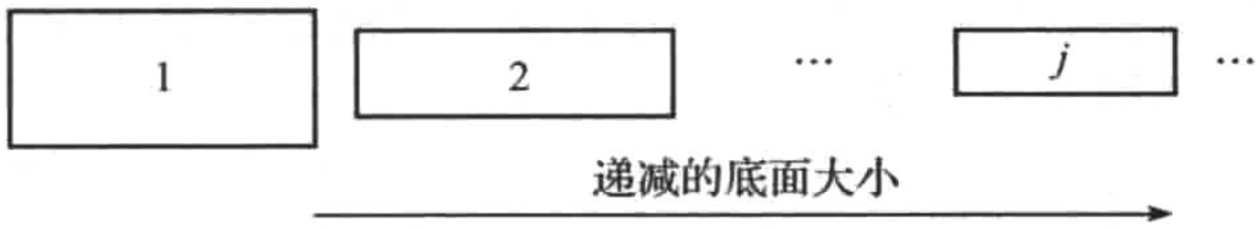
**问题 23　堆箱子问题：**假设有 $n$ 个立方体形状的箱子，第 $i$ 个箱子的维度是高 $h_i$、宽 $w_i$、深 $d_i$。现在将箱子堆得尽可能高，但是箱子只能按其二维底面的大小以严格递减的方式从下向上堆放，可以旋转箱子使任意一个面为底，并且可以使用多个维度相同的箱子。

**解答：**堆箱子问题能够转化为最长递增子序列问题(问题 21)。

输入：$n$ 个箱子，其中第 $i$ 个箱子的尺寸为高 $h_i$、宽 $w_i$、深 $d_i$。由于可以旋转，所以需要考虑 $n$ 个箱子的所有面。即如果有一个箱子大小为 1×2×3，那么可视作 3 个尺寸不同的箱子：

$$1\times2\times3\Rightarrow\begin{cases}1\times(2\times3) & \text{高 1、底长 2 和宽 3}\\ 2\times(1\times3) & \text{高 2、底长 1 和宽 3}\\ 3\times(1\times2) & \text{高 3、底长 1 和宽 2}\end{cases}$$

上述简化使得不需要考虑箱子的旋转问题，而只需要考虑高为 $h_i$，底面为($w_i\times d_i$)的 $3n$ 种箱子的堆放方法。假设 $w_i\leqslant d_i$。目标是将箱子尽可能堆高，具有最大的高度。只有当箱子 $i$ 在两个维度上都小于箱子 $j$(即 $w_i<w_j$ 且 $d_i<d_j$)时，才允许箱子 $i$ 放在箱子 $j$ 上。现在，使用动态规划法求解该问题。首先按箱子底面大小递减的顺序选择。




令 $H(j)$表示箱子 $j$ 在最上方时堆放的最大高度，这与最长递增子序列问题类似，因为箱子按底面降序排列，以箱子 $j$ 为最上方箱子的堆放方式相当于前 $j$ 个箱子的一个子序列，箱子的堆放顺序与序列的顺序相同。

下面给出 $H(j)$的递归式。为了得到箱子 $j$ 在最上方的堆放方式，需要扩充在 $j$ 之前以箱子 $i$ 在最上方的堆放方式，即将箱子 $j$ 放在箱子 $i$ 上(箱子 $i$ 是当前堆放在最上方的箱子)，该操作需要满足条件 $w_i>w_j$ 且 $d_i>d_j$(保证下层的箱子比上层的箱子底面更大)。基于该逻辑，可以得到如下递归式：

$$H(j)=\max_{i<j\text{且}w_i>w_j\text{且}d_i>d_j}\{H(i)\}+h_i$$

与最长递增子序列问题类似，因为不能确定哪个箱子放在最上方是最优的，所以最后需要在所有可能的堆放方式中找出最佳的 $j$。

$$\max_j\{H(j)\}$$

时间复杂度为 $O(n^2)$。

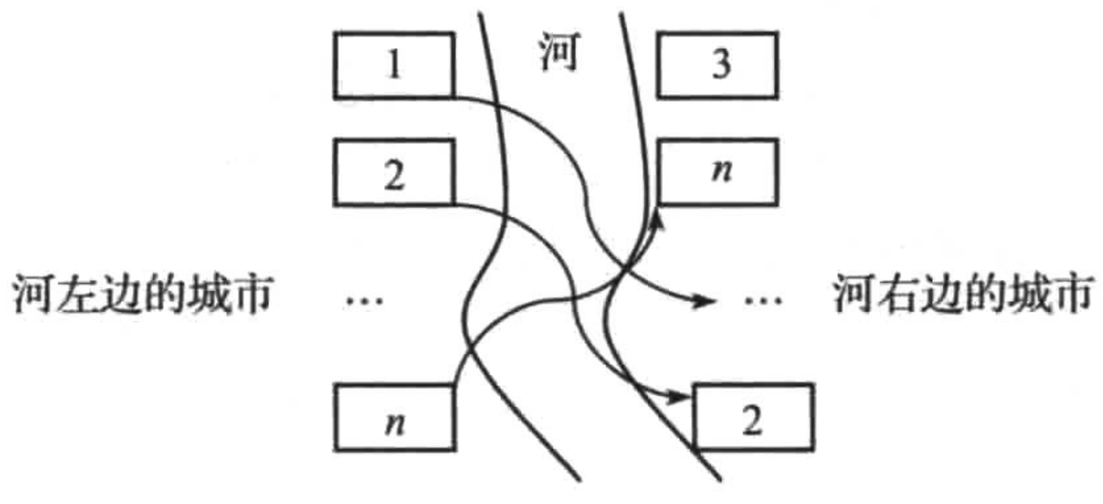
**问题 24　印度建桥问题：**有一条自北向南的长河，河的两边各有 $n$ 座城市，左边有 $n$ 座，右边有 $n$ 座。这些城市用 $1\sim n$ 进行编号，但是顺序未知。现在需要用桥将左右两边的城市尽可能多地连接起来，并且两座桥之间不允许交叉。当连接两座城市时，只允许将左边城市 $i$ 与右边城市 $i$ 进行连接。

**解答：**

输入：两个配对的集合，分别用 $1\sim n$ 的数字编号。

目标：尽可能多地构建连接左边城市和右边城市的桥，且任意两座桥之间不能交叉。

为了更好地理解该问题，请参考下面的图。从图中可知，河的左边有 $n$ 座城市，右边也有 $n$ 座城市，具有同样编号的城市才允许用桥连接(问题的要求)。目标是用不交叉的桥连接最多的城市。为了简单起见，将其中一边的城市有序排列。



仔细观察可以发现，因为左边的城市已经有序排列，所以该问题转化为寻找右边城市的最大递增子序列。这意味着可用 LIS 算法找到右边城市的最长递增序列。

时间复杂度为 $O(n^2)$（与 LIS 一样）。

**问题 25　子集和：**有 $n$ 个正数的序列 $A_1 \cdots A_n$。设计算法检查是否存在 $A$ 的一个子集，其元素和等于 $T$。

**解答：**这是背包问题的一个变型。例如，有以下数组：

$$A = [3,2,4,19,3,7,13,10,6,11]$$

假设需要检查是否存在元素和等于 17 的子集。该问题的答案是存在，因为 4＋13＝17，所以{4，13}是这样的一个子集。

下面用动态规划法求解该问题。定义一个 $n \times T$ 矩阵，其中 $n$ 是输入数组中元素的个数，$T$ 是需要检查的和。若能从 1～$i$ 之间找到一个元素和等于 $j$ 的子集，则令 $M[i, j]=1$，否则 $M[i, j]=0$。

$$M[i,j] = \mathrm{Max}(M[i-1,j], M[i-1,j-A_i])$$

根据上述与背包问题类似的递归式，可知得到元素和等于 $j$ 的子集中不包含第 $i$ 个元素，也可以包含第 $i$ 个元素，但前提是存在不包含第 $i$ 个元素且元素和为 $j-A_i$ 的子集。这与背包问题一样，除了这里保存 0 或 1 而不是价值外。在下面的实现中，可以使用 *OR* 运算得到 $M[i-1, j]$和 $M[i-1, j-A_i]$中的最大值。

```
int SubsetSum( int A[], int n, int  T ) {
        int i, j, M[n+1][T +1];
        M[0][0]=0;
        for (i=1; i<= T; i++)
                M[0][i]= 0;
        for (i=1; i<=n; i++) {
                for (j = 0; j<= T; j++) {
                        M[i][j] = M[i-1][j] || M[i-1][j] - A[i];
                }
        }
        return M[n][T];
}
```

**有多少个子问题？**在上式中，$i$ 的取值范围为 1～$n$，$j$ 的取值范围为 1～$T$，因此共有 $nT$ 个子问题，每一个子问题的求解时间为 $O(1)$，所以时间复杂度为 $O(nT)$，这不是多项式时间，因为运行时间取决于两个变量（$n$ 和 $T$），并且两个变量呈现指数函数关系。空间复杂度为 $O(nT)$。

**问题 26　**已知包含 $n$ 个整数的集合，其中所有元素之和最多为 $K$，求这 $n$ 个元素的一个子集，其和正好是 $n$ 个元素和的一半。

**解答：**

输入：$n$ 个数 $A_1 \cdots A_n$，其和至多为 $K$。

目标：$n$ 个数的一个子集，其和为 $n$ 个元素总和的一半。

用动态规划法求解该问题。创建一个大小为 $K+1$ 的布尔数组 $T$。如果存在一个和为 $x$ 的子集，则 $T[x]=1$。这说明在算法结束后，$T[K]$等于 1 当且仅当存在和为 $K$ 的子集。一旦得到 $T$，则返回 $T[K/2]$。如果其值为 1，则说明存在和为总和一半的子集。

初始时设置 $T$ 的所有值为 0。由于空集的和为 0，所以令 $T[0]=1$。如果 $A$ 不包括任何元素，那么算法结束；否则选择第一个元素 $A[0]$，可以将其放入子集或者忽略该元素，

即将数组 $T$ 中的 $T[0]$和 $T[A[0]]$设置为 1。这是递归的基本情况。对 $A$ 中的第二个元素继续该过程。

假设已经分析了 $A$ 数组中的前 $i-1$ 个元素。现在针对 $A[i]$，看看如何更新数组 $T$。在处理了 $i-1$ 个元素后，所有从已处理的数字可以得到的和，数组 $T$ 相应位置均设置为 1。如果加上这个新的数字 $A[i]$，$T$ 会发生什么变化？首先，可以忽略 $A[i]$，这时 $T$ 保持不变。现在假设某个位置(比如 $T[j]$为 1)，这表明除了 $A[i]$以外在其他元素集中可以找到一个子集，其和为 $j$。如果将 $A[i]$加入到该子集中，则得到一个新的子集，和为 $j+A[i]$，这时将 $T[j+A[i]]$也设置为 1。综上所述，可以得到以下算法：

```
int T[10240];
int SubsetHalfSum( int  A[], int n ) {
        int K = 0;
        for( int i = 0; i < n; i++ )
                K += A[i];
        T[0] = 1;
        for( int i = 1; i <= K; i++ )          // 初始化表
                T[i] = 0;
        for( int i = 0; i < n; i++ ) {       // 一个一个地处理数字
                for( int j = K - A[i]; j >= 0; j--)     {
                        if( T[j] )
                                T[j + A[i]] = 1;
                }
        }
        return T[K / 2];
}
```

在上述代码中，$j$ 从右向左循环，避免了重复计数问题。如果从左至右循环，那么可能存在重复计算。

时间复杂度为 $O(nk)$，因为两个 for 循环。空间复杂度为 $O(k)$，由布尔表 $T$ 产生。

**问题 27**　能否改善问题 26 的性能？

**解答：可以。**在问题 26 的实现代码中，内部的 $j$ 循环从 $K$ 开始向左移动，这表明不需要每次都对整个表进行扫描。

实际上，只需要找出 $T$ 中所有为 1 的项。开始时，仅有第 0 项为 1。如果用一个变量记录最右端的 1 在数组 $T$ 中的位置，那么可以从该位置而不是表的最右端向左移动。

为了充分利用这一特性，首先将 $A$ 数组排序，这样最右端为 1 的项将尽可能慢地向右移动。事实上，本问题并不需要知道表的右半部分($T[K/2]$之后)取何值，因为如果$T[x]$为 1，那么 $T[Kx]$最终也必定为 1，即对应于和为 $x$ 的子集的补集。基于以上讨论，实现代码为：

```
int T[10240];
int SubsetHalfSumEfficient( int  A[], int n ) {
        int K = 0;
        for( int i = 0; i < n; i++ )
                K += A[i];
        Sort(A,n));
        T[0] = 1;          // 初始化表
        for( int i = 1; i <= sum; i++ )
                T[i] = 0;
```

```
        int R = 0;        // 最右边的一项
        for( int i = 0; i < n; i++)  {          // 一个一个地处理数字
                for( int j = R; j >= 0; j--) {
                        if( T[j] )
                                T[j + A[i]] = 1;
                        R = min(K / 2, R + C[i] );
                }
        }
        return T[K / 2];
}
```

改进代码的时间复杂度仍然为 $O(nK)$，但移除了一些多余的步骤。

**问题 28　布尔表达式括号计数问题：**已知由符号‘true’、‘false’、‘and’、‘or’和‘xor’组成的布尔表达式，求有多少种添加括号的方式使得该表达式的最终值为真。例如，对于表达式‘false and false xor true’，只有一种添加括号方式使得该表达式为真。

**解答：**

输入：由符号 $T$ 和 $F$ 组成的布尔表达式。假设符号的个数为 $n$，符号之间是 and、or、xor 等布尔运算符。例如，若 $n=4$，则有 $T$ or $F$ and $T$ xor $F$。

目标：对表达式添加括号，计算共有多少种方式使得表达式的值为真。例如，在上例中，如果这样添加括号 $T$ or(($F$ and $T$)xor $F$)，其表达式值为真。

$$T \text{ or}((F \text{ and } T)\text{xor } F) = \text{True}$$

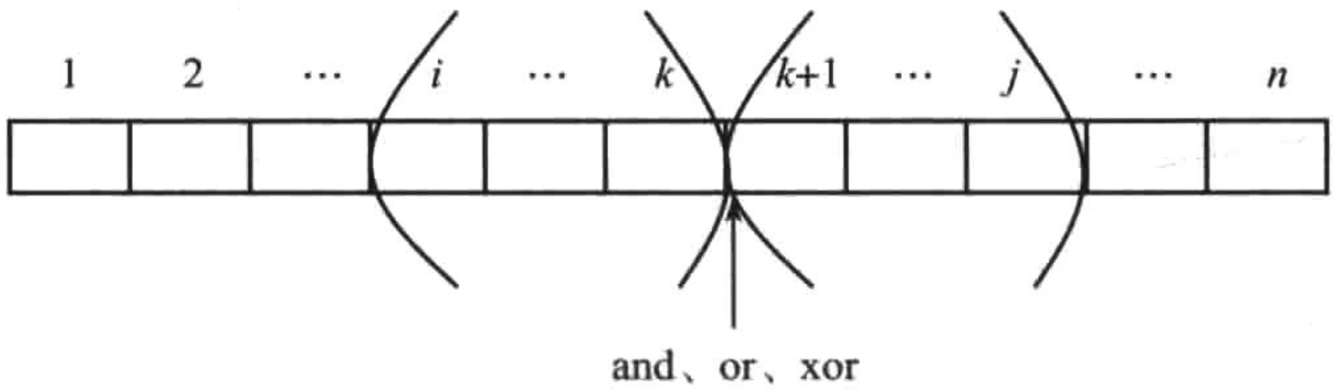
下面使用动态规划法求解该问题。令 $T(i, j)$表示对于由符号 $i$～符号 $j$ 之间(符号仅包含 $T$ 和 $F$，不含运算符)的所有符号和布尔运算符组成的子布尔表达式，使其值为真的添加括号方式的总数。$i$ 和 $j$ 的取值范围均为 1～$n$。例如，在上例中，因为不存在任何添加括号的方式使得表达式 F and T xor F 值为真，所以 $T(2, 4)=0$。

为了简单起见，令 $F(i, j)$表示对于由符号 $i$～符号 $j$ 之间的所有符号和布尔运算符组成的子布尔表达式，使其值为假的添加括号方式的总数。递归的基本情况为 $T(i, i)$和 $F(i, i)$。

对于所有的 $i$ 值，计算 $T(i, i+1)$和 $F(i, i+1)$，以此类推，计算 $T(i, i+2)$和 $F(i, i+2)$。上述过程可以推广为如下形式：

$$T(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k)T(k+1,j) & \text{对于“and”} \\ \text{Total}(i,k)\text{Total}(k+1,j) - F(i,k)F(k+1,j) & \text{对于“or”} \\ T(i,k)F(k+1,j) + F(i,k)T(k+1,j) & \text{对于“xor”} \end{cases}$$

其中，$\text{Total}(i, k)=T(i, k)+F(i, k)$。




在上述递归式中，$T(i, j)$表示使表达式为真的添加括号的方式数。假设按中间符号 $k$ 分解可以得到多个子问题，这时符号 $i$～$j$ 添加括号的方式数等于符号 $i$～$k$ 的方式数与符号 $k+1$～$j$ 的方式数之和。给符号 $k$ 和 $k+1$ 之间添加括号可根据运算符分为 3 种情况：

- 如果符号 $k$ 和 $k+1$ 之间的运算符是“and”，仅当两个子表达式的值都为真时，表达式的值才为真。将这些为真的方式数组合起来就得到最后的方式数。
- 如果符号 $k$ 和 $k+1$ 之间的运算符是“or”，这时只要其中之一的子表达式为真，那么结果就为真。上式中不直接统计为真的组合数，而是将为假的组合数从总方式数中减去就得到表达式为真的方式数。
- 对于“xor”运算符，讨论同上。

在找到所有的值后，需要选择产生为真最多的方式数的 $k$ 值，而 $k$ 有 $i \sim j-1$ 种可能。

**有多少子问题？** 在上述递归式中，$i$ 的取值范围为 $1 \sim n$，$j$ 的取值范围也是从 $1 \sim n$，因此共有 $n^2$ 个子问题，然后需要对所有子问题的值进行求和运算，所以时间复杂度为$O(n^3)$。

**问题 29　任意两点之间的最短路径问题(弗洛伊德(Floyd)算法)：** 已知加权有向图 $G=(V, E)$，其中 $V=\{1, 2, \cdots, n\}$。求图中任意两点之间的最短路径。矩阵 $C[V][V]$ 表示权值，$C[i][j]$表示结点 $i$ 和 $j$ 之间的权值(或开销)。如果结点 $i$ 和 $j$ 之间不存在任何路径，则 $C[i][j]=\infty$或$-1$。

**解答：** 下面给出该问题的动态规划算法(弗洛伊德算法)。弗洛伊德算法用矩阵 $A[1..n][1..n]$表示任意两点之间最短路径的长度，初始时，

$$A[i,j] = C[i,j] \quad i \neq j$$
$$= 0 \qquad\quad i = j$$

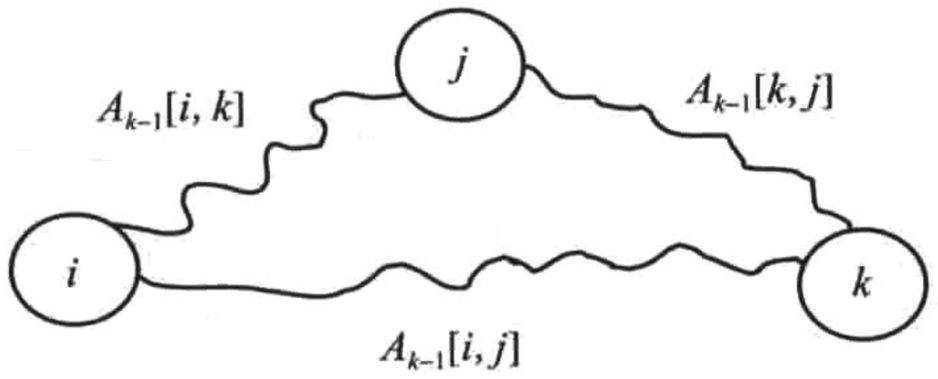
从定义可知，若 $i$ 和 $j$ 之间没有路径，则 $C[i][j]=\infty$。矩阵 $\boldsymbol{A}$ 有 $n$ 次迭代，用 $\boldsymbol{A}_0$，$\boldsymbol{A}_1$，$\cdots$，$\boldsymbol{A}_n$ 表示 $\boldsymbol{A}$ 的 $n$ 次迭代的结果，$\boldsymbol{A}_0$ 为初始值。

在 $k-1$ 次迭代后，$A_{k-1}[i, j]$＝所有从顶点 $i$ 到 $j$ 且不经过顶点$\{k+1, k+2, \cdots, n\}$的路径的最小长度，即这条路径可能通过顶点$\{1, 2, 3, \cdots, k-1\}$。

每一次迭代，$A[i][j]$用 $A_{k-1}[i, j]$和 $A_{k-1}[i, k]+A_{k-1}[k, j]$中的最小值对其进行更新。

$$A[i,j] = \min\begin{cases} A_{k-1}[i,j] \\ A_{k-1}[i,k] + A_{k-1}[k,j] \end{cases}$$

实质上第 $k$ 次迭代是探索对于所有 $i$ 和 $j$ 的值，顶点 $k$ 是否位于从 $i$ 到 $j$ 的最优路径中，如下图所示。



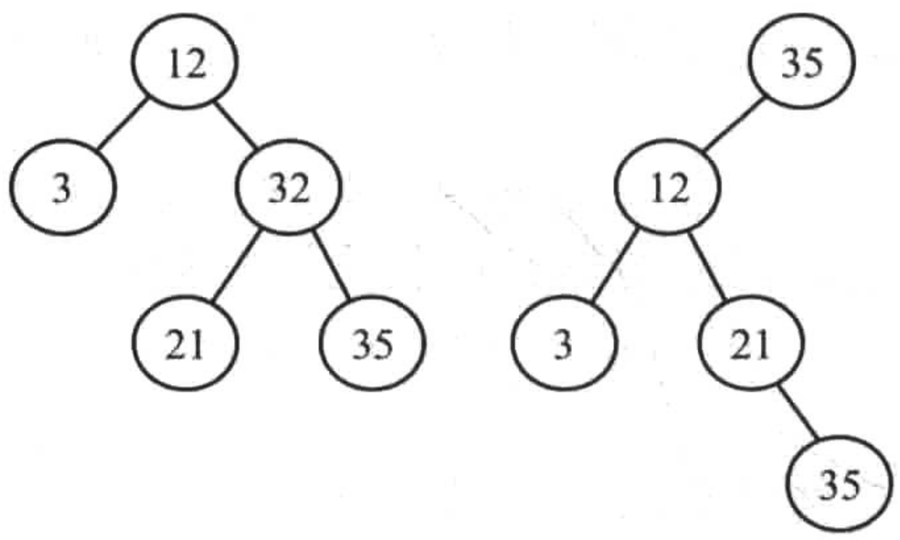

```
void Floyd(int C[][], int A[][], int n) {
        int i, j, k;
        for(i = 0, i <= n - 1;i + +)
                for(j = 0; j <= n - 1, j + +)
                        A[i][j] = C[i][j];
        for(i = 0;i <= n - 1;i + +)
                A[i][i] = 0;
```

```
            for(k = 0;k <= n - 1;k + +) {
                    for(i = 0;i <= n - 1;i + +) {
                            for(j = 0;j <= n - 1, j + +)
                                    if(A[i][k] + A[k][j] < A[i][j])
                                            A[i][j] = A[i][k] + A[k][j];
                    }
            }
    }
```

时间复杂度为 $O(n^3)$。

**问题 30　最优二叉搜索树：** $A[1..n]$为 $n$ 个键值(有序)的集合，基于 $A$ 中的元素构建一棵最优二叉搜索树。假设已知每一个元素的查询频率，即某元素在二叉搜索树中被检索的次数。构建最优二叉搜索树使得总查询时间尽可能少。

**解答：** 在求解该问题前，先通过一个例子来了解问题的含义。假设已知数组 $A=[3, 12, 21, 32, 35]$，存在多种方式来构建二叉搜索树，下面给出其中的两种。




**上面两棵搜索树，哪一棵更好？** 元素的查询时间取决于结点的深度，左边搜索树中元素的平均比较次数为：$\frac{1+2+2+3+3}{5}=\frac{11}{5}$，而对于右边的搜索树，其平均比较次数为：$\frac{1+2+3+3+4}{5}=\frac{13}{5}$。因此，对于这两棵搜索树，第一棵更好。

如果查询频率是未知的，并且需要查询所有的元素，则上述简单的计算就可用来决定最优二叉搜索树。如果频率是已知的，则最优二叉搜索树的性能取决于元素的查询频率和元素的深度。为了简单起见，元素集合用 $A$ 表示，相应的查询频率用数组 $F$ 表示。$F[i]$表示第 $i$ 个元素 $A[i]$的查询频率。基于这些定义，根为 root 的树的总查询时间 $S(\text{root})$可用下式计算：

$$S(\text{root}) = \sum_{i=1}^{n}(\text{depth}(\text{root},i)+1)\times F[i])$$

在上述表达式中，$\text{depth}(\text{root}, i)+1$ 表示查询第 $i$ 个元素所需要的比较次数。根据二叉搜索树的定义，左子树的元素值小于根元素，右子树的元素值大于根元素。如果分成左子树查询时间和右子树查询时间，则上述表达式可表示为：

$$S(\text{root}) = \sum_{i=1}^{r-1}(\text{depth}(\text{root}\rightarrow\text{left},i)+1)\times F[i] + \sum_{i=1}^{n}F[i]$$

$$+\sum_{i=r+1}^{n}(\text{depth}(\text{root}\rightarrow\text{right},i)+1)\times F[i]$$

其中 $r$ 表示数组中根元素的位置。

左子树和右子树的查询时间可用递归调用代替，因此得到下式：

$$S(\text{root}) = S(\text{root} \rightarrow \text{left}) + S(\text{root} \rightarrow \text{right}) + \sum_{i=1}^{n} F[i]$$

**二叉搜索树结点声明**

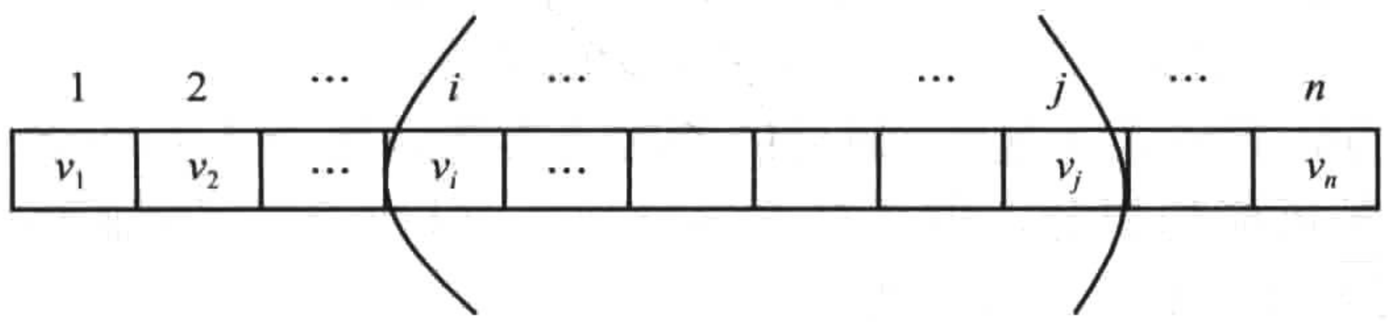
参见第 6 章。

```
BinarySearchTreeNode OptimalBST(int A[], int F[], int low, int high) {
        int r, minTime = 0;
        BinarySearchTreeNode newNode = new BinarySearchTreeNode();
        if(newNode == null) {
                System.out.println("Memory Error");
                return;
        }
        for (r =0, r <= n-1; r++)  {
                root.setLeft(OptimalBST(A, F, low, r-1));
                root.setRight(OptimalBST(A, F, r+1, high));
                root.setData(A[r]);
                if(minTime > S(root))
                        minTime = S(root);
        }
        return minTime;
}
```

**问题 31　最优游戏策略：**有 $n$ 个硬币排成一行，面值分别为 $v_1$，$v_2$，…，$v_n$，其中 $n$ 为偶数(因为这是二人博弈)。在与对手的博弈中，选手轮流从这排硬币中选取第一或者最后一枚，取出该枚硬币，并累加硬币对应的面额。如果率先选取硬币，求能够赢得的最大金额。

**解答：**目标是，最大化博弈过程中获得的金额之和。谁先开始，谁应该赢得该次游戏，即获得最大的总金额。需要注意的是，在选取硬币的过程中不要被对手的选取方式所干扰。

采用动态规划法求解该问题。在每一轮中，要么是我方要么是对方从这排硬币的两端选取其中一枚硬币。可以将子问题定义如下：




$V(i, j)$：表示如果剩下硬币为 $v_i \cdots v_j$，那么当轮到我方挑选硬币时能够赢得的最大金额。

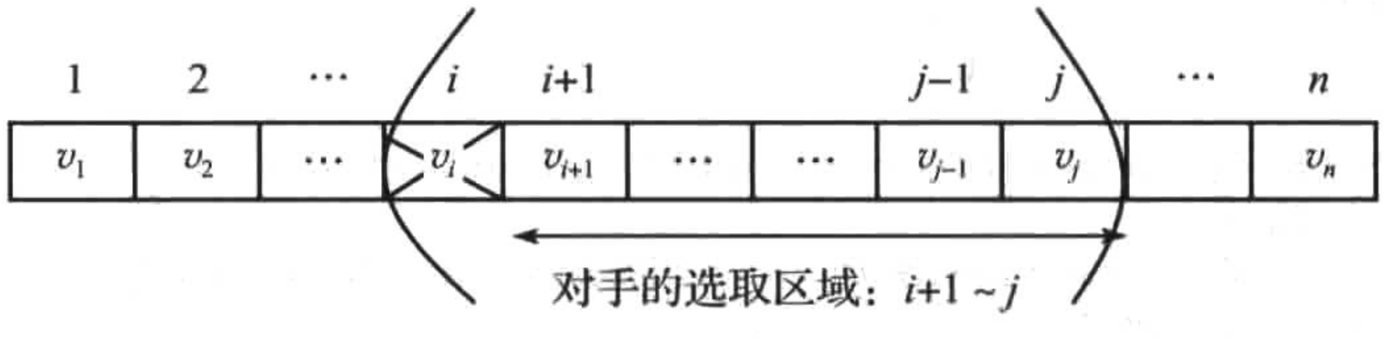
基本情况：对于任意 $i$，$V(i, i)$和 $V(i, i+1)$。

从这些值中可以计算 $V(i, i+2)$、$V(i, i+3)$等。每一个子问题的 $V(i, j)$可以定义为：

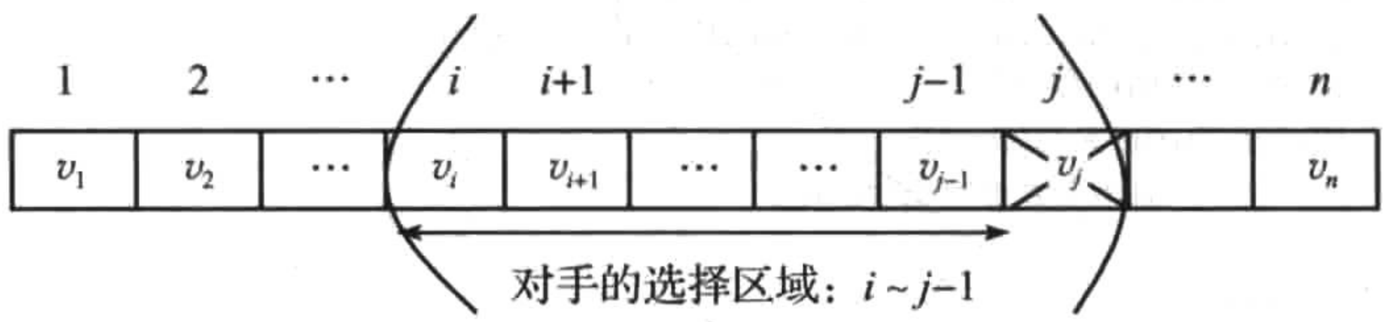
$$V(i,j) = \text{Max}\left\{\text{Min}\begin{Bmatrix} V(i+1,j-1) \\ V(i+2,j) \end{Bmatrix} + v_i,\quad \text{Min}\begin{Bmatrix} V(i,j-2) \\ V(i+1,j-1) \end{Bmatrix} + v_j\right\}$$

上述递归式关注的对象是第 $i$ 个硬币到第 $j$ 个硬币($v_i \cdots v_j$)。因为轮到我方挑选硬币，所以有两个可能：选取 $v_i$ 或者 $v_j$。上式中的第一项对应于选取第 $i$ 个硬币的情形($v_i$)，第二项对应于选取第 $j$ 个硬币的情形($v_j$)。外部的 Max 函数表示选取能够产生最大金额的硬币，现在关注细节：

- 选取第 $i$ 枚硬币：如果选取第 $i$ 枚硬币，那么剩余的硬币为 $i+1 \sim j$。因为选取了第 $i$ 枚硬币，所以累加其面额。对于剩余的 $i+1 \sim j$ 枚硬币，对手可以选择第 $i+1$ 或者第 $j$ 枚硬币，但对手的选择将是最小化我方收益(Min 函数)，如下图所示。



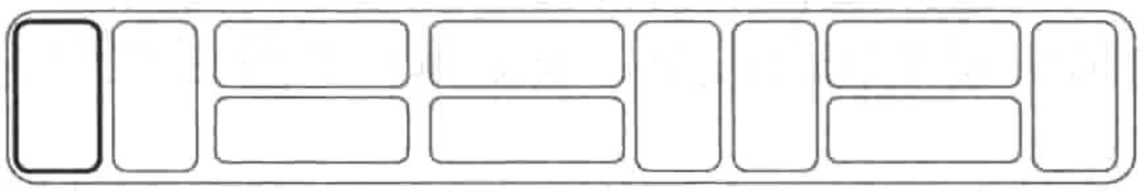

- 选取第 $j$ 枚硬币：剩余的硬币为 $i \sim j-1$。因为选取了第 $j$ 枚硬币，所以累加其面额 $v_j$。对于剩余的 $i \sim j-1$ 枚硬币，对手可以选择第 $i$ 或者第 $j-1$ 枚硬币，但对手的选择将是最小化我方收益(Min 函数)。




**有多少个子问题？** 在上述公式中，$i$ 的取值范围为从 $1 \sim n$，$j$ 也是从 $1 \sim n$，那么共有 $n^2$ 个子问题，求解每一个子问题耗时 $O(1)$，因此总的时间复杂度为 $O(n^2)$。

**问题 32　盖瓦：** 假设使用大小为 $2 \times 1$ 的多米诺骨牌覆盖高度为 2 的无限长条纹。那么，使用 $2 \times 1$ 大小的多米诺骨牌覆盖 $2 \times n$ 的矩形条纹有多少种方式？

**解答：**



注意问题中的瓦片可以竖直或者水平放置。若竖直放置，则至少需要 $1 \times 2$ 的间隙；若水平放置，则需要 $2 \times 1$ 的间隙。按此原理，上述问题转换为求数字 1 和 2 完全划分 $n$ 的方式(考虑次序)[1]。例如，$11=1+2+2+1+2+2+1$。

对于 12 的划分，可以在上述 11 划分的基础上在末尾增加 1 或者基于 10 的划分加 2。同理，令 $n$ 有 $F_n$ 种可能的划分，那么对于$(n+1)$，可以在其末尾加 1，或者在$(n-1)$划分的基础上在末尾加 2。基于上述分析，可知，

$$F_{n+1} = F_n + F_{n-1}$$

将上述理论应用到本问题中：

- 有多少种方式覆盖 $2 \times 1$ 的条纹：1 种→仅需 1 个竖直放置的瓦片。
- 有多少种方式覆盖 $2 \times 2$ 的条纹：2 种→2 个竖直或者水平放置的瓦片。
- 有多少种方式覆盖 $2 \times 3$ 的条纹：3 种→在解决 $2 \times 2$ 条纹的两个解的末尾竖直放置一个瓦片，或在解决 $2 \times 1$ 条纹的唯一解的末尾水平放置两个瓦片$(2+1=3)$。
- 同理，有多少种方式覆盖 $2 \times n$ 的条纹：在 $2 \times (n-1)$的条纹的所有解后竖直放置一个瓦片，或者在 $2 \times (n-2)$的条纹的所有解后水平放置两个瓦片$(F_{n-1}+F_{n-2})$。
- 由此可得上述递归式：$F_n=F_{n-1}+F_{n-2}$，其中 $F_1=1$，$F_2=2$。

**问题 33　编辑距离：** 已知长度为 $m$ 的字符串 $A$ 和长度为 $n$ 的字符串 $B$，给定以下类

型的操作将字符串 $A$ 变换成字符串 $B$，求所需的最小操作数。具体操作有：从 $A$ 中删除一个字符、在 $A$ 中插入一个字符，或者将 $A$ 中的某个字符替换成一个新的字符。$A$ 变换成 $B$ 所需的最少操作数称为 $A$ 与 $B$ 之间的编辑距离。

**解答：**在介绍求解算法前，先分析将 $A$ 变换成 $B$ 的可能操作。

- 如果 $m>n$，需要从 $A$ 中删除一些字符。
- 如果 $m==n$，则可能需要替换 $A$ 中的某些字符。
- 如果 $m<n$，则需要在 $A$ 中插入一些字符。

因此，所需的操作为插入一个字符、替换一个字符和删除一个字符，每个操作的代价定义如下：

**操作的代价：**

| | |
|---|---|
| 插入字符 | $c_i$ |
| 替换字符 | $c_r$ |
| 删除字符 | $c_d$ |

下面重点介绍该问题的递归表达式，令 $T(i, j)$表示将 $A$ 的前 $i$ 个字符变换成 $B$ 的前 $j$ 个字符所需的最小代价，即从 $A[1\cdots i]$到 $B[1\cdots j]$。

$$T(i,j)=\min\begin{cases}c_d+T(i-1,j)\\ T(i,j-1)+c_i\\ \begin{cases}T(i-1,j-1) & A[i]==B[j]\\ T(i-1,j-1)+c_r & A[i]\neq B[j]\end{cases}\end{cases}$$

基于上述讨论，有如下各种情况。

- 如果删除 $A$ 中的第 $i$ 个字符，那么需要将 $A$ 剩下的 $i-1$ 个字符变换成 $B$ 的前 $j$ 个字符。
- 如果插入第 $i$ 个字符到 $A$ 中，那么需要将 $A$ 的前 $i$ 个字符变换成 $B$ 的前 $j-1$ 个字符。
- 如果 $A[i]==B[j]$，那么只需将 $A$ 的前 $i-1$ 个字符变换成 $B$ 的前 $j-1$ 个字符。
- 如果 $A[i]\neq B[j]$，那么需要用 $B$ 的第 $j$ 个字符替换 $A$ 的第 $i$ 个字符，然后将 $A$ 剩下的 $i-1$ 个字符变换成 $B$ 的 $j-1$ 个字符。

在计算所有可能性后，选择其中代价最小的一个操作。

**有多少个子问题？**在上式中，$i$ 的取值范围为 $1\sim m$，$j$ 的取值范围为 $1\sim n$，那么共有 $mn$ 个子问题，求解每一个子问题耗时 $O(1)$，因此时间复杂度为 $O(mn)$。空间复杂度：$O(mn)$，其中 $m$ 和 $n$ 分别是备忘录矩阵的行数和列数。

**问题 34　最长回文子序列：**回文指无论从左至右或从右至左读，结果都一样的序列。例如，A，C，G，G，G，G，C，A。已知一个长度为 $n$ 的序列，设计算法输出最长回文子序列的长度。例如，字符串 A，G，C，T，C，B，M，A，A，C，T，G，G，A，M 有很多回文子序列，其中 A，G，T，C，M，C，T，G，A 的长度为 9。

**解答：**采用动态规划法求解该问题，考虑字符串 $A$ 的子串 $A[i, \cdots, j]$，如果 $A[i]==A[j]$，那么可以找到一个长度至少为 2 的回文子序列，如果不相等，则需要在子序列 $A[i+1, \cdots, j]$和 $A[i, \cdots, j-1]$中寻找最长回文子序列。

此外，每一个字符是一个长度为 1 的回文，因此基本情况为 $A[i, i]=1$。定义子串 $A[i, \cdots, j]$的最长回文子序列长度为 $L(i, j)$。

$$L(i,j)=\begin{cases}L(i+1,j-1)+2 & A[i]==A[j]\\ \text{Max}\{L(i+1,j),L(i,j-1)\} & \text{否则}\end{cases}$$

$$L(i,i)=1 \quad i=1\sim n$$

```
int LongestPalindromeSubsequence(int A[], int n) {
        int max  = 1;
        int i,k, L[n][ n];
        for (i  = 1; i<= n -1; i++) {
                L[i][i] =1;
                if(A[i]==A[i+1]) {
                        L[i][i + 1]  = 1;
                        max = 2;
                }
                else      L[i][i + 1]  = 0;
}
for (k=3;k<= n;k++) {
                for (i  = 1;i <= n-k +1; i++) {
                        j =  i + k - 1;
                        if(A[i] == A[j])      {
                                L[i, j] =  2 + L[i + 1][j - 1];
                                max = k;
                        }
                        else L[i, j]  = max(L[i + 1][j - 1], L[i][j - 1]);
                }
}
        return max;
}
```

时间复杂度：第一个 for 循环花费 $O(n)$的时间，而第二个 for 循环花费 $O(n-k)$的时间，等价于 $O(n)$，因此总运行时间为 $O(n^2)$。

**问题 35　最长回文子串：**已知字符串 $A$，求 $A$ 的正序和逆序完全相同的最长子串。

**解答：**最长回文子串和最长回文子序列的基本区别在于：对于最长回文子串，其输出的最长回文由连续字符组成；而对于最长回文子序列，组成最长回文的字符不一定是连续的，但在给定字符串中的位置必须是升序排列的。

蛮力法穷举长度为 $n$ 的字符串的所有可能的 $n(n+1)/2$ 个子串，判断其是否为回文，记录当前最长的回文。在最坏情况下该方法的复杂度为 $O(n^3)$，但是由于回文要么是以某个字符(奇数长度回文)或字符间的空格(偶数长度回文)对称，所以可以利用这一特点得到更好的解决方法：检查所有 $n+1$ 个中心，找出以其为中心的最长回文，并跟踪所有中心的最长回文。最坏情况下该方法的复杂度为 $O(n^2)$。

下面使用动态规划法求解该问题。值得注意的是，对于长度为 $n$ 的字符串，子串数不超过 $O(n^2)$(而子序列数有 $2^n$ 个)。因此可以扫描每一个子串，判断其是否为回文，然后更新目前为止最长回文子串的长度。因为判断回文需要的时间是子串长度的线性函数，所以该方法复杂度为 $O(n^3)$。可以利用动态规划法改善该性能，对于 $1\leqslant i\leqslant j\leqslant n$，

$$L(i,j)=\begin{cases}1 & \text{如果 } A[i]\cdots A[j] \text{ 是一个回文子串}\\ 0 & \text{否则}\end{cases}$$

$$L[i,i]=1$$

$$L[i,j]=L[i,i+1],if\ A[i]==A[i+1] \quad 1\leqslant i\leqslant j\leqslant n-1$$

对于长度不小于 3 的字符串，有 $L[i, j]=(L[i+1, j-1]$且 $A[i]=A[j])$。

需要注意的是，为了得到一个定义良好的递归式，需要显式地初始化布尔矩阵 $L[i, j]$两条对角线上的值，这是因为项$[i, j]$的递归式需要使用相隔两个对角线的$[i-1, j-1]$的值(即对于长度为 $k$ 的子串，需要知道长度为 $k-2$ 的子串的状态值)。

```
int LongestPalindromeSubstring(int A[], int n) {
        int max  = 1;
        int i,k, L[n][n];
        for (i  = 1; i<=n-1; i++) {
                L[i][i] =1;
                if(A[i]==A[i+1]) {
                        L[i][i + 1]  = 1;
                        max = 2;
                }
                else L[i][i + 1]  = 0;
}
for (k=3;k<=n;k++) {
        for (i  = 1;i <= n-k +1; i++) {
                j =  i + k - 1;
                if(A[i] == A[j] && L[i + 1][j - 1]) {
                        L[i][j] =  1;
                        max = k;
                }
                else L[i][j]  = 0;
        }
}
return max;
}
```

时间复杂度：第一个 for 循环花费 $O(n)$的时间，而第二个 for 循环花费 $O(n-k)$的时间，等价于 $O(n)$，因此总运行时间为 $O(n^2)$。

**问题 36**　已知两个字符串 $S$ 和 $T$，设计算法求字符串 $S$ 在 $T$ 中出现的次数。不要求 $S$ 中的所有字符在 $T$ 中连续出现。例如，若 $S$=ab，$T$=abadcb，那么解为 3，因为 ab 在 abadcb 中出现 3 次。

**解答：** 令 $L(i, j)$表示 $S$ 中的 $i$ 个字符在 $T$ 的 $j$ 个字符中出现的次数。

$$L(i,j)=\mathrm{Max}\begin{cases}0 & j=0\\ 1 & i=0\\ L(i-1,j-1)+L(i,j-1) & S[i]==T[j]\\ L(i-1,j) & S[i]\neq T[j]\end{cases}$$

上述递归式中各个分量的含义如下：

- 如果 $j=0$，则 $T$ 为空串，所以计数值也为 0。
- 如果 $i=0$，认为空串 $S$ 也可出现在 $T$ 中，计数值为 1。
- 如果 $S[i]==T[j]$，则意味着 $S$ 的第 $i$ 个字符与 $T$ 的第 $j$ 个字符一样，该情况需要看 $S$ 的 $i-1$ 个字符与 $T$ 的 $j-1$ 个字符组成的子问题，以及 $S$ 的 $i$ 个字符与 $T$ 的 $j-1$个字符对应问题的解，因为 $S$ 的 $i$ 个字符可能出现在 $T$ 的 $j-1$ 个字符内。
- 如果 $S[i]\neq T[j]$，原问题的解取决于 $S$ 的 $i$ 个字符与 $T$ 的 $j-1$ 个字符组成的子问题的解。

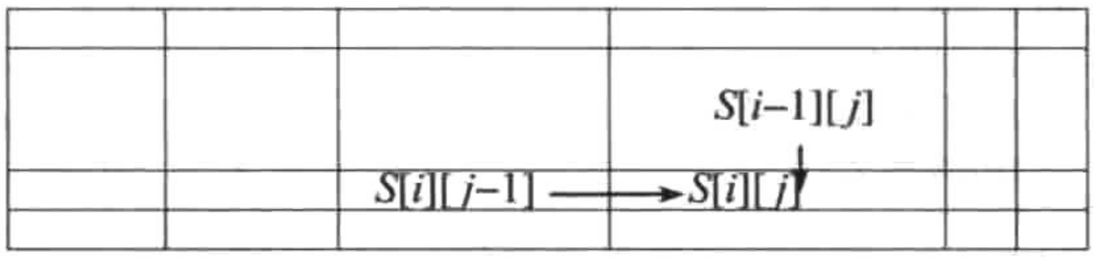
在计算所有值后，选择其中数值最大的一个。

**有多少子问题？** 在上述递归式中，$i$ 的取值范围为 $1\sim m$，$j$ 的取值范围为 $1\sim n$，共

有 mn 个子问题，求解每一个子问题耗时 $O(1)$，因此时间复杂度为 $O(mn)$。空间复杂度为$O(mn)$，其中 $m$ 和 $n$ 分别为备忘录矩阵的行数和列数。

**问题 37** 已知 $n$ 行 $m$ 列的矩阵($n\times m$)，其每一个元素值代表放在该格子内苹果的数量，从矩阵左上角开始，向下或者向右移动一个格子，最终到达右下角，求该过程中能够收集苹果的最大数量。每经过一个格子时，将收集其中所有的苹果。

**解答：** 用 $\boldsymbol{A}[n][m]$表示该矩阵，首先可知最多有两种方式进入某个单元格——从左边(如果该单元格不是在第一列)或从上方(如果不是在最上面一行)。




为了找出从某个格子开始的最优解，需要先得到到达该单元格所有可能经过的单元格的最优解。从上述描述可以容易得出以下递归关系：

$$S(i,j)=\left\{A[i][j]+\mathrm{Max}\begin{Bmatrix}S(i,j-1) & j>0\\ S(i-1,j) & i>0\end{Bmatrix}\right\}$$

按照从上向下遍历每一行，在每一行内从左至右的顺序计算 $S(i, j)$，或者从左至右遍历每一列，在每一列内按照从上至下的顺序计算 $S(i, j)$。

```
int FindApplesCount(int A[][], int n, int m) {
        int S[n][m];
        for(int i = 0;i<n;i++) {
                for(int j = 0;i<m;j++) {
                        S[i][j] = A[i][j];
                        if(j>0 && S[i][j] < S[i][j] + S[i][j-1])
                                S[i][j] += S[i][j-1];
                        if(i>0 && S[i][j] < S[i][j] + S[i-1][j])
                                S[i][j] +=S[i-1][j];
                }
        }
        return S[n][m];
}
```

**有多少个子问题？** 在上述公式中，$i$ 的取值范围从 $1\sim n$，$j$ 取 $1\sim m$ 的所有整数，那么共有 $nm$ 个子问题。求解每一个子问题耗时 $O(1)$。算法的时间复杂度为 $O(nm)$。空间复杂度为 $O(nm)$，其中 $n$ 是矩阵的行数，$m$ 是列数。

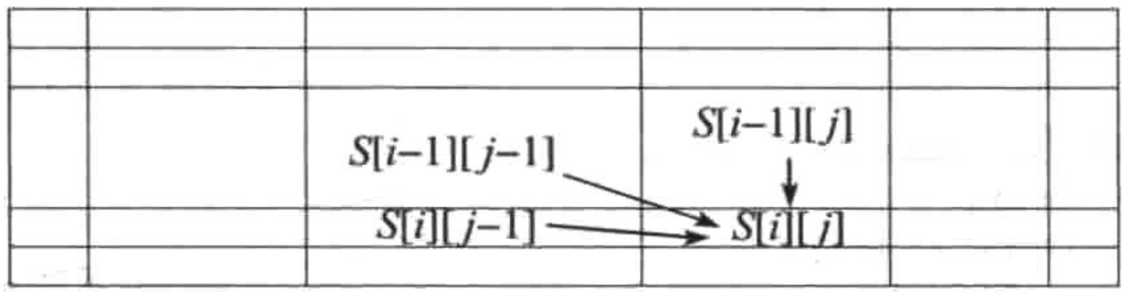
**问题 38** 类似于问题 37，但假设不仅可以向下和向右移动，而且还可以沿着对角线移动。给出一个动态规划算法，计算到达右下角时收集的最多苹果数。

**解答：** 该问题的讨论与问题 37 非常相似。令 $\boldsymbol{A}[n][m]$表示该矩阵，首先可知有 3 种方式进入某个单元格——从左边、从上方(如果单元格不是处于最上方的行)或从左上角。为了得到某个单元格的最优解，需要已知到达该单元格所有可能经过的单元格的最优解。根据上述描述，可得到以下递归关系式：

$$S(i,j)=\left\{A[i][j]+\mathrm{Max}\begin{Bmatrix}S(i,j-1) & j>0\\ S(i-1,j) & i>0\\ S(i-1,j-1) & i>0\ \text{且}\ j>0\end{Bmatrix}\right\}$$

按照从上向下遍历每一行，在每一行内从左至右的顺序计算 $S(i, j)$，或者从左至右

遍历每一列，在每一列内按照从上至下的顺序计算 S($i$，$j$)。




**有多少个子问题？**在上述公式中，$i$ 的取值范围为 1～$n$，$j$ 的取值范围为 1～$m$ 的所有整数，那么共有 $nm$ 个子问题，求解每一个子问题耗时 $O(1)$。算法的时间复杂度为 $O(nm)$。空间复杂度为 $O(nm)$，其中 $n$ 是矩阵的行数，$m$ 是列数。

**问题 39　所有元素为 1 的最大子方阵：**已知一个值为 0 或 1 的矩阵，设计一个算法求所有元素值为 1 的最大子方阵。例如，有以下二值矩阵：

```
0 1 1 0 1
1 1 0 1 0
0 1 1 1 0
1 1 1 1 0
1 1 1 1 1
0 0 0 0 0
```

值为 1 的最大子方阵为

```
1 1 1
1 1 1
1 1 1
```

**解答：**利用动态规划方法求解该问题。用 ***B***[$m$][$n$]表示该二元矩阵。算法的思想是构造一个临时矩阵 ***L***[][]，其中元素 $L[i][j]$表示包括 $B[i][j]$在内的值全为 1 的子方阵的大小，$B[i][j]$是子方阵最右下角的元素。

**算法：**

1）基于已知矩阵 ***B***[$m$][$n$]构造一个矩阵 ***L***[$m$][$n$]。

a. 将 ***B***[][]的第一行和第一列的元素值复制到 ***L***[][]的第一行和第一列。

b. 对于其他元素，用以下表达式计算。

```
if(B[i][j])
        L[i][j] = min(L[i][j − 1], L[i − 1][j], L[i − 1][j − 1]) + 1;
else    L[i][j] = 0;
```

2）找出 ***L***[$m$][$n$]中的最大值。

3）根据 $L[i]$中得到的最大值和其坐标，输出 ***B***[][]对应的子方阵。

```
void MatrixSubSquareWithAllOnes(int B[][], int m, int n) {
        int i, j, L[m][n], max_of_s, max_i, max_j;
        for(i = 0; i < m; i++)                    // 设置L[][]的第一列
                L[i][0] = B[i][0];
        // 设置L[][]的第一行
        for(j = 0; j < n; j++)
                L[0][j] = B[0][j];
        // 构造L[][]的其他项
        for(i = 1; i < m; i++) {
                for(j = 1; j < n; j++) {
                        if(B[i][j] == 1)
```

```
                                    L[i][j] = min(L[i][j-1], L[i-1][j], L[i-1][j-1]) + 1;
                            else    L[i][j] = 0;
                    }
            }
            max_of_s = L[0][0]; max_i = 0; max_j = 0;
            for(i = 0; i < m i++) {
                    for(j = 0; j < n; j++) {
                            if(L[i][j] > max_of_s) {
                                    max_of_s = L[i][j];
                                    max_i = i;
                                    max_j = j;
                            }
                    }
            }
            System.out.println("Maximum sub-matrix");
            for(i = max_i; i > max_i - max_of_s; i--) {
                    for(j = max_j; j > max_j - max_of_s; j--)
                            System.out.println(B[i][j]);
            }
    }
```

**有多少个子问题？** 在上述公式中，$i$ 的取值范围为 $1\sim m$，$j$ 的取值范围为 $1\sim n$ 的所有整数，那么共有 $nm$ 个子问题。求解每一个子问题耗时 $O(1)$。因此算法的时间复杂度为$O(nm)$。空间复杂度为 $O(nm)$，其中 $m$ 是矩阵的行数，$n$ 是列数。

**问题 40　最大和子矩阵：** 给定一个 $n\times n$ 的矩阵 $\boldsymbol{M}$，其元素值为正或负整数。设计一个算法求具有最大和的子矩阵。

**解答：** 令 Aux[$r$，$c$]表示由 $\boldsymbol{M}$ 的角元素[1，1]和对角元素[$r$，$c$]形成的子矩阵的和。因为有 $n^2$ 个这样的子矩阵，所以需要 $O(n^2)$计算时间。在得到这些值后，可在常数时间内得到 $\boldsymbol{M}$ 的任意一个子矩阵的和。该方法给出了一个 $O(n^4)$的算法，基于 Aux 表，容易计算具有任意左上角和右下角的子矩阵的和。

**问题 41**　能够降低问题 40 的复杂度？

**解答：** 只需对问题 4 中的算法做细微的调整就可以改善问题 40 中的复杂度。问题 4 中给出了在一维数组中求最大和子数组的算法：一次扫描一个元素，跟踪扫描元素的和，如果在某一点的和值为负，则将其设置为 0。该算法有时称为 Kadane 算法。下面将利用该算法作为辅助函数求解二维问题[1]，方法如下：

```
void FindMaximumSubMatrix(int A[][], int n){
   //计算列的垂直前缀之和
   int M[n][n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (j == 0)
            M[j][i] = A[j][i];
         else
            M[j][i] = A[j][i] + M[j - 1][i];
    }
}
int maxSoFar = 0;
int min , subMatrix;
//应用 Kadane's Alg 遍历所有的可能组合
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        min = 0;
```

```
subMatrix = 0;
for (int k = 0; k < n; k++) {
    if (i == 0) {
        subMatrix += M[j][k];
    } else {
        subMatrix += M[j][k] - M[i - 1 ][k];
    }
    if(subMatrix < min){
        min = subMatrix;
    }
    if((subMatrix - min) > maxSoFar){
                maxSoFar = subMatrix - min;
            }
        }
    }
}
```

时间复杂度为 $O(n^3)$。

**问题 42**　已知数 $n$，求最少的平方数，使其和为 $n$。例如，$\min[1]=1=1^2$，$\min[2]=2=1^2+1^2$，$\min[4]=1=2^2$，$\min[13]=2=3^2+2^2$。

**解答：**该问题可以转换为硬币找零问题。硬币面额为 $1\sim\sqrt{n}$，求兑换金额为 $n$ 的最少硬币数。

**问题 43　到达最后元素的最小跳数问题：**已知一个数组，从第一个元素经过若干跳到达最后一个元素，一跳的最大长度是数组当前位置的值，最优的结果是以最少的跳数到达目标。

**例子：**给定数组 $A=\{2, 3, 1, 1, 4\}$，到达最后元素的可能方式(以下标进行表示)有：

- 0，2，3，4(首跳长度为 2 到达下标 2，然后越过长度 1 到达下标 3，最后一跳的距离为 1 到达下标 4)。
- 0，1，4(首跳长度为 1 到达下标 1，然后再一跳经过长度 3 到达下标 4)。

因为第二个方案仅包括 2 跳，所以它是最优解。

**解答：**该问题是一个经典的动态规划的例子。尽管该问题可以通过蛮力法求解，但其复杂度高。可以利用求解最长递增子序列(LIS)的方法来求解本问题。遍历整个数组，得到到达任意位置(下标)的最小跳数并更新结果数组。一旦到达最后一个元素，则结果数组的最后一个元素就是最优解。

**如何得到到达任意位置(下标)的最小跳数？**对于第一个元素，最优的跳数为 0。如果第一个元素的值为 0，则不能跳到任何一个元素，因此返回无穷大。对于第 $n+1$ 个元素，初始化结果数组 result$[n+1]$为无穷大，然后从 $0\sim n$ 循环，检测某下标 $i$ 是否可以从 $i$ 跳到 $n+1$。如果可以，接下来看其跳数(result$[i]+1$)是否小于 result$[n+1]$，如果是，则更新 result$[n+1]$，否则继续下一个下标。

```
//定义MAX至少为1，使得加1后不会为0
#define MAX 0xFFFFFFFE;
unsigned int jump(int *array, int n) {
        unsigned answer, int *result = new unsigned int[n];
        int i, j;
        if(n==0 || array[0] == 0)
                return MAX;
```

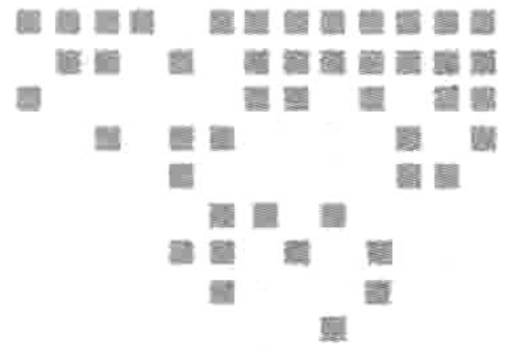
```
        result[0] = 0;  //不需要跳转到第一个元素
        for (i = 1; i < n; i++) {
                result[i] = MAX; //初始化result[i]
                for (j = 0; j < i; j++) {
                        //检验是否跳转可能来自j
                        if(array[j] >= (i-j)) {
                                //检验是否有更好的解决方案
                                if((result[j] + 1) < result[i])
                                        result[i] = result[j] + 1;  //更新result[i]
                        }
                }
        }
        answer = result[n-1];                    //返回result[n-1]
        delete[] result;
        return answer;
}
```

上述代码将返回最优跳数。为了同时得到每一跳的下标，对代码进行简单修改就能轻松解决该问题。

时间复杂度：因为有两层嵌套循环，每一个循环的迭代范围为 $0\sim i$，所以总时间为 $1+2+3+4+\cdots+n-1$，时间复杂度为 $O(n\times(n-1)/2)=O(n^2)$。空间复杂度为 $O(n)$，用于结果数组的存储。

**问题 44**　如果采用动态规划算法求解不具有重叠子问题的问题，将产生什么结果？

**解答：**因为子问题的解无法重复使用，所以将导致内存的浪费，并且算法的运行时间与分治算法的时间是一样的。



# 第 20 章 Chapter 20
# 复杂度类型

## 20.1 引言

在前面的各章中，描述了不同问题求解的复杂度。某些算法随着问题规模的增加其复杂度的增长速率较低，而另一些则有比较高的增长速率。对于具有较低增长率的问题，称为简单问题(或易求解问题)；对于具有较高复杂度的问题，称为难问题(或难求解问题)。该分类是基于求解某个问题时算法的运行时间(或者占用内存)决定的。

| 时间复杂度 | 名称 | 例子 | 问题类型 |
|---|---|---|---|
| $O(1)$ | 常量 | 在链表的头部增加一个元素 | 易求解问题 |
| $O(\log n)$ | 对数 | 在一二叉搜索树中查找某个元素 | |
| $O(n)$ | 线性 | 在无序的数组中查找某个元素 | |
| $O(n\log n)$ | 线性对数 | 归并排序 | |
| $O(n^2)$ | 二次函数 | 图中两个结点的最短路径 | |
| $O(n^3)$ | 三次函数 | 矩阵乘法 | |
| $O(2^n)$ | 指数 | 汉诺塔问题 | 难求解问题 |
| $O(n!)$ | 阶乘 | 字符串的所有排列 | |

现实中，有很多不知道其解决算法的问题。目前本书介绍的所有问题都是计算机可以在确定时间内求解的问题。在开始讨论前，首先介绍本章所使用的基本术语。

## 20.2 多项式/指数时间

指数时间，实质上，意味着尝试每一种可能性(例如，回溯算法)，此类算法的求解速度是非常慢的。多项式时间意味着问题有某种更聪明的求解算法，而不是尝试每一种

可能性。数学上，用以下符号表示：

- 多项式时间：存在某个 $k$，时间复杂度为 $O(n^k)$。
- 指数时间：存在某个 $k$，时间复杂度为 $O(k^n)$。

## 20.3 决策问题的定义

决策问题是一个答案为是或否的问题，并且答案取决于输入的值。例如，问题“包含 $n$ 个元素的数组是否存在任何重复的元素?”是一个决策问题，该问题的答案可能为是，也可能为否，取决于输入数组的值。




## 20.4 决策过程

对于决策问题，假设已经给出某个求解算法。以算法的形式描述求解该决策问题的过程称为该问题的决策过程。

## 20.5 复杂度类型的定义

在计算机科学领域中，为了理解尚未有求解算法的问题，将问题分为不同的类型，称为复杂度类型。在复杂度理论中，一个复杂度类型是指具有相似复杂度的一类问题。这是计算理论的一个分支，它研究求解问题进行相应计算时所耗费的资源。最常见的资源是时间(算法求解问题需要花费多长时间)和空间(算法所消耗的内存大小)。

## 20.6 复杂度类型

### 1. P 类型

复杂度类型 P 是指一个确定性机器在多项式时间内能够求解的决策问题的集合(P 代表多项式时间)。P 问题的求解算法容易得到。

### 2. NP 类型

复杂度类型 NP(NP 表示非确定性的多项式时间)是指一个非确定性机器在多项式时间内可以求解的决策问题的集合。NP 问题难以得到相应的求解算法，但对某个解易于验证。

为了更好地理解，考虑以下例子。假设一个学院登记在册的学生有 500 名，并有 100 间教室可用。选择其中 100 名学生结对放入教室，但学生的班主任有一个出于某种原因不能结对安排的学生列表。所有可能结对的方案非常庞大，但提供给班主任的方案(结对的学生名单)容易验证是否存在错误。

如果某个禁止的结对出现在学生名单中，则发生错误。在该问题中，尝试所有的可能性非常困难，但验证某个解比较容易。这意味着，如果提供一个问题的解，可以在多项式时间内判断其正确与否。基于上述讨论，对于答案为是的 NP 问题，其结果可以在多项式时间内完成验证。

### 3. Co-NP 类型

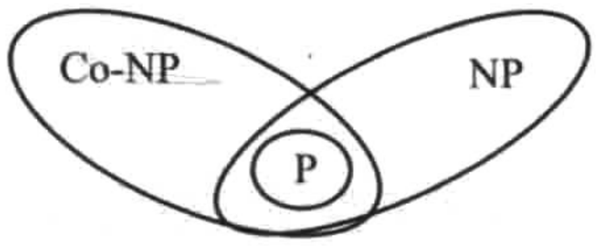
Co-NP 是 NP 的反(NP 的补集)。如果 Co-NP 问题的解为否，那么该结果可以在多项

式时间内验证。

| | |
|---|---|
| P | 多项式时间内可解 |
| NP | 为是的解可以在多项式时间内验证 |
| Co-NP | 为否的解可以在多项式时间内验证 |

### 4. P、NP 和 Co-NP 问题的关系

P 中的任意一个决策问题也是一个 NP 问题。因为对于一个 P 问题，其为是的解可以在多项式时间内验证。同理，任意一个 P 问题也是一个 Co-NP 问题。在理论计算机科学领域中一个重要的开放问题是 P 是否等于 NP(P＝NP)，该问题尚未有答案。从直观上看，明显有结论 P≠NP，但无法对该结论进行证明。



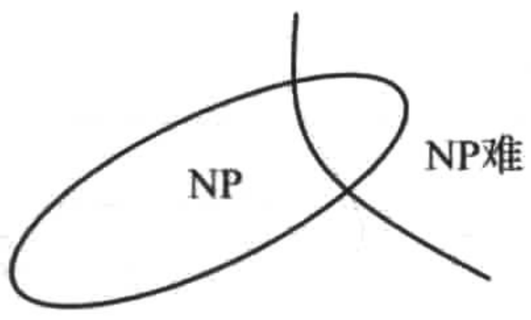

另一个开放问题是 NP 与 Co-NP 是否不同。即使能够快速验证为是的解，也并不意味着能快速地验证为否的解。一般地，认为 NP≠Co-NP，但同样无法证明该结论。

### 5. NP 难问题

NP 难问题是指这样一类问题，所有 NP 问题都能归约到该类问题。NP 难问题不一定属于 NP 问题，因此即使验证其解也需要花费较长的时间，这意味着即使提供 NP 难问题的一个解，也需要很长的时间去验证该解正确与否。

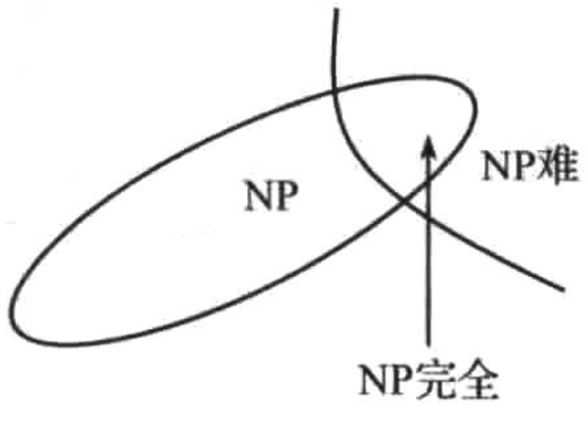
如果问题 $K$ 是 NP 难问题，则意味着如果 $K$ 存在一个多项式时间算法，那么对于 NP 中每一个问题都存在一个多项式时间算法。因此，

$K$ 是 NP 难问题表明如果 $K$ 能够在多项式时间内求解，则 $P = NP$。




### 6. NP 完全问题

最后，如果一个问题介于 NP 难问题与 NP 问题之间，则该问题是 NP 完全问题。NP 完全问题是 NP 问题中最难的。如果对于一个 NP 完全问题能够找到一个多项式时间算法，那么对于每一个 NP 完全问题都能够找到相应的多项式时间算法。这意味着能够快速地验证问题的解，并且所有 NP 问题都能归约到该问题。

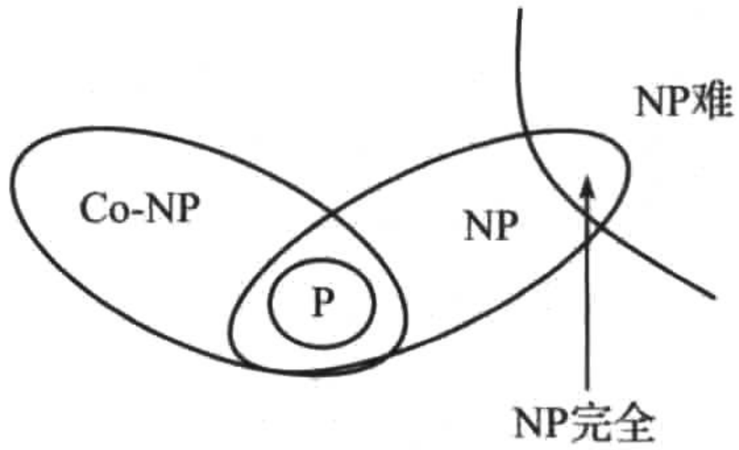


### 7. P、NP、Co-NP、NP 难和 NP 完全问题的关系

NP 难问题是 NP 完全问题的严格超集。有些问题(如停机问题)是 NP 难问题，但不是 NP 问题。NP 难问题一般很难找到求解方法。

NP 难和 NP 完全问题在难的程度上有所区别，因为 NP 问题包含所有非最难问题——如果一个问题不是 NP 问题，则该问题比所有 NP 问题都要难。

基于上述讨论，可以知道不同类型之间关系(记住，这仅是一个假想)。




### 8. P=NP 吗

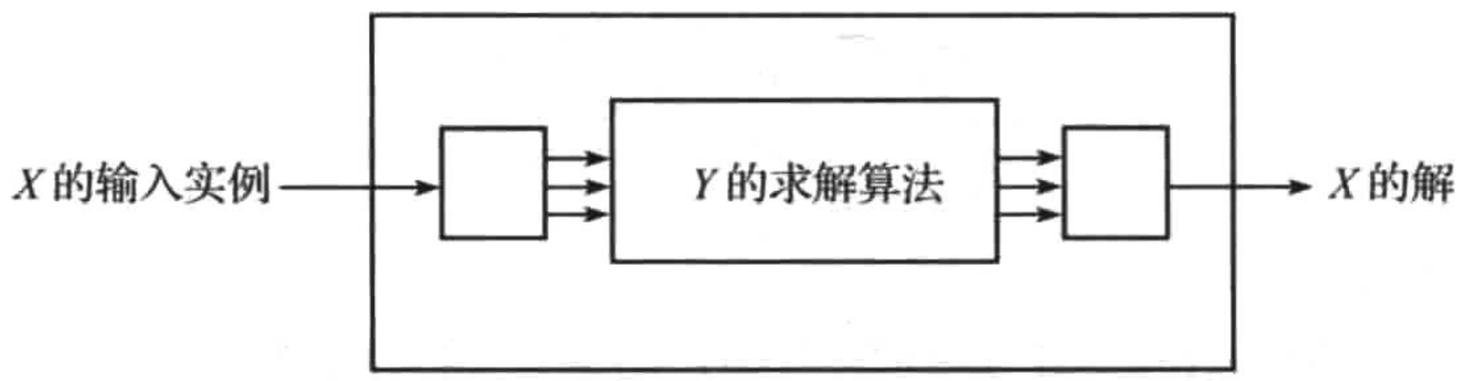
如果 P=NP，则意味着所有能够快速验证其解的问题也一定能够快速求解(注意判断一个问题的解是否正确与实际求解一个问题的区别)。

这是一个重要的问题(没人知道其答案)，因为目前存在大量的 NP 完全问题不能够快速求解。如果 P=NP，则意味着存在一个快速求解方法。注意“快速”表示采用非试错法。快速算法可能需要上亿年，但是只要不采用试错法，就可以快速求解。未来，一台计算机将能够将上亿年的计算时间变成几分钟。

## 20.7 归约

在讨论归约前，首先考虑以下情景。如果需要求解问题 $X$，但此问题非常复杂，这时该怎么办?

首先能想到的是，如果有与 $X$ 相似的问题(如 $Y$)，那么可以将 $X$ 映射到 $Y$，然后使用 $Y$ 的算法来求解 $X$。这个过程就称为归约。




为了将问题 $X$ 映射到问题 $Y$，需要某些算法，该算法可能需要花费线性时间或更多。基于该讨论，求解问题 $X$ 的代价可以表示为：

$$\text{求解 } X \text{ 的代价} = \text{求解 } Y \text{ 的代价} + \text{归约时间}$$

下面考虑另一种情况。求解问题 $X$ 有时需要多次使用 $Y$ 的算法，在这种情况下，

$$\text{求解 } X \text{ 的代价} = \text{次数} \times \text{求解 } Y \text{ 的代价} + \text{归约时间}$$

NP 完全问题的一个重要特点就是可归约性，即将给定的 NP 完全问题归约(或转化)为其他已知的 NP 完全问题。由于 NP 完全问题难于求解，所以为了证明给定问题难于求解，可以选择一个已知的难问题(已证明的难问题)，然后将给定问题映射到该难问题，由此证明给定问题也是难求解的。

**注意**：为了证明给定问题的求解困难性，将给定问题归约为已知的难问题并不是必需的，有时可将已知的难问题归约为给定问题。

## 重要的 NP 完全问题(用于归约)

**可满足性问题**：如果一个布尔表达式是多个子句的合取(AND)，而每一个子句又是多个文字的析取(OR)，且每一个文字要么是一个变量要么是其否定形式，则称该表达式为合取公式(CNF)。例如：$(a \vee b \vee c \vee d \vee e) \wedge (b \vee \sim c \vee \sim d) \wedge (\sim a \vee c \vee d) \wedge (a \vee \sim b)$。

3-CNF 公式是每个子句包含 3 个文字的 CNF 公式。上例不是 3-CNF 公式，因为它的第一个子句包含 5 个文字，而最后一个子句只有 2 个文字。

3-SAT 问题：3-SAT 只是受限于 3-CNF 公式的 SAT 问题。已知一个 3-CNF 公式，是否存在一个变量的赋值，使得公式的值为真?

2-SAT 问题：2-SAT 只是受限于 2-CNF 公式的 SAT 问题。已知一个 2-CNF 公式，是否存在一个变量的赋值，使得公式的值为真?

**电路可满足性问题**：已知一个由与门、或门和非门组成的布尔组合电路，其是否是可满足的? 也就是说，给定一个通过电线适当连接的由与门、或门和非门组成的布尔电路，电路可满足性(Circuit-SAT)问题就是确定是否存在一个输入赋值，使其输出为真。

**哈密顿路径问题(Ham-Path)**：给定一个无向图，是否存在一条仅访问每个顶点一次的路径?

**哈密顿回路问题(Ham-Cycle)**：给定一个无向图，是否存在一条仅访问每个顶点一次的回路(起始点和终点相同)?

**有向哈密顿回路问题(Dir-Ham-Cycle)**：给定一个有向图，是否存在一条仅访问每个顶点一次的回路(起始点和终点相同)?

**旅行商问题(TSP)**：已知城市的列表以及任意两个城市之间的距离，如何找出一条访问每个城市仅一次的最短旅行线路。

**最短路径问题(Shortest Path)**：已知一个有向图以及两个顶点 $s$ 和 $t$，判断是否存在一条从 $s$ 到 $t$ 的最短简单路径。

**图着色问题**：图的 $k$ 着色是将 $k$ 种颜色分别映射到每个顶点，满足一条边相连的两个顶点的颜色不同。图着色问题是找出某图合理着色时所需要的最少颜色数。

**3 色问题**：给定一个图，是否能够用三种颜色对其着色，使每条边的两个顶点的颜色不同?

**团问题(也称完全图问题)**：已知一个图，团问题试图计算其最大完全子图具有的结点数，即需要找出为完全图的最大子图。

**独立子集问题(Ind_Set)**：令 $G$ 为任意图，$G$ 的一个独立集是 $G$ 顶点的一个子集，其中任意顶点之间不存在相连的边。最大独立集合问题是找出给定图的最大独立子集的大小。

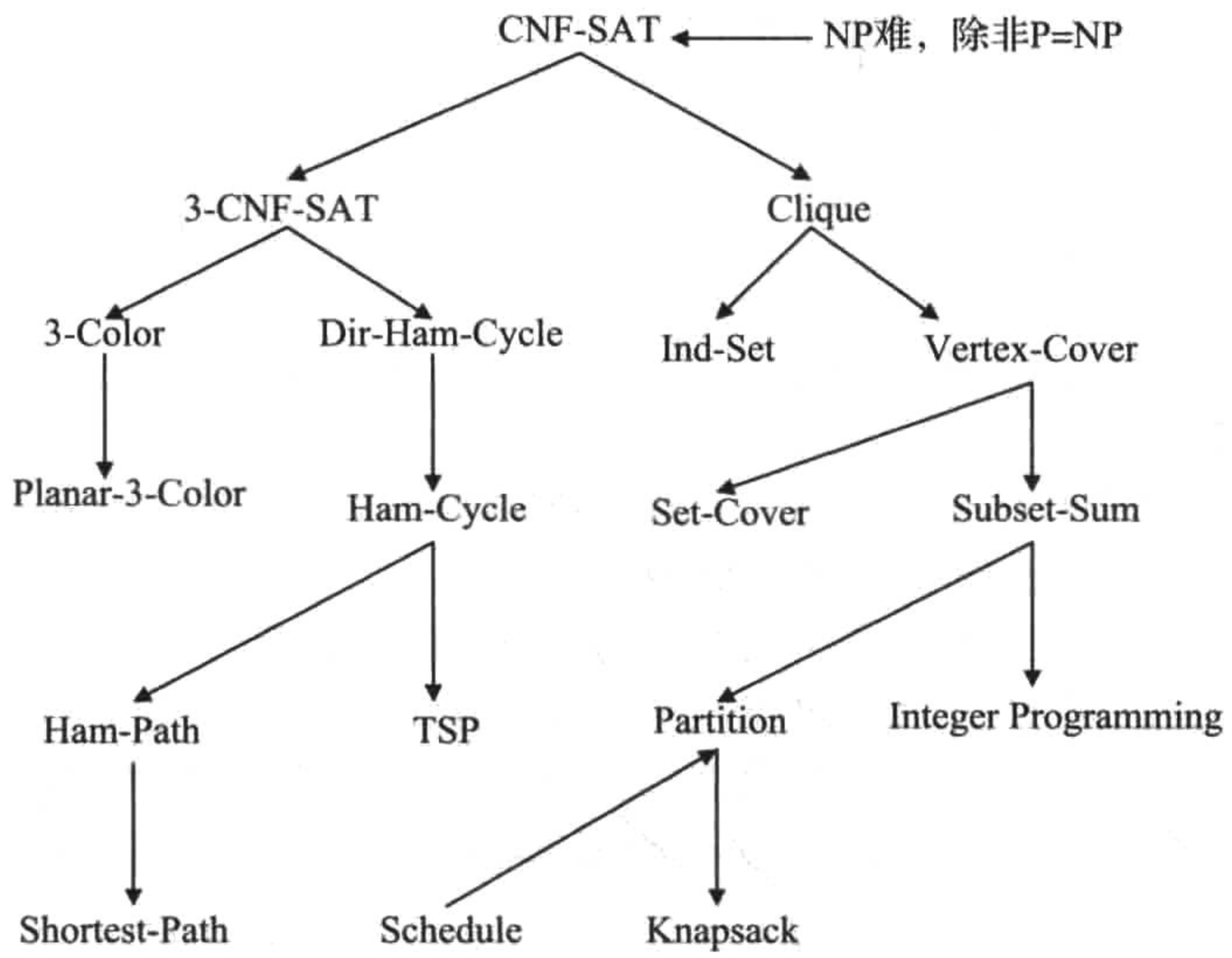
**顶点覆盖问题(Vertex-Cover)**：图的顶点覆盖是指一个顶点子集，使图中的每一条边都至少依附集合中的一个顶点。顶点覆盖问题就是找出图的最小顶点覆盖集合。

**子集和问题(Subset-Sum)**：已知一个整数集合 $S$ 和一个整数 $T$，确定是否存在 $S$ 的一个子集，其子集元素和等于 $T$。

**整数规划问题**：已知整数 $b_i$ 和 $a_{ij}$，确定二元变量 $x_i$ 的值满足线性等式。

$$\sum_{j=1}^{N} a_{ij}x_j = b_i \qquad 1 \leqslant i \leqslant M$$

$$x_j \in \{0,1\} \quad 1 \leqslant j \leqslant N$$




在上图中，箭头表示归约。例如，Ham-Cycle(哈密顿回路问题)可以归约为 CNF-SAT 问题，对于其他问题，也存在同样的推理。本章不讨论问题之间具体的归约过程。Cook 定理证明了电路可满足性问题是 NP 难问题。这说明，电路可满足性问题是一个已知的 NP 难问题。

**注意：**因为下面介绍的问题是 NP 完全问题，所以它也是 NP 和 NP 难问题。为了简单起见，不考虑它们的归约证明。

## 20.8 复杂度类型的相关问题

**问题 1** 什么是快速算法？

**解答：**快速算法不是试错型求解算法。算法可能需要上亿年的求解时间，但只要不使用试错法，则算法是高效的。未来的计算机能够将上亿年的计算时间缩短为几分钟。

**问题 2** 什么是高效算法？

**解答：**一个高效算法满足以下性质：

- 随着输入规模的变化而变化。
- 不关心常量。
- 渐近运行时间：多项式时间。

**问题 3** 能否在多项式时间内求解所有问题？

**解答：不能。**答案是明显的，因为已经介绍了许多比多项式时间更长的问题。

**问题 4** 是否存在 NP 难问题？

**解答：**根据定义，NP 难问题是非常难的。难于证明和验证它是难的。Cook 定理证明了电路可满足性问题是 NP 难问题。

**问题 5** 对于 2-SAT 问题，以下结论哪些是正确的？

(a) P　　(b) NP　　(c) CoNP　　(d) NP 难
(e) CoNP 难　　(f) NP 完全　　(g) CoNP 完全

**解答：** 2-SAT 可在多项式时间内求解，所以是 P、NP 和 CoNP。

**问题 6**　对于 3-SAT 问题，以下结论哪些是正确的？

(a) P　　(b) NP　　(c) CoNP　　(d) NP 难
(e) CoNP 难　　(f) NP 完全　　(g) CoNP 完全

**解答：** 3-SAT 是 NP 完全问题，所以是 NP、NP 难和 NP 完全。

**问题 7**　对于 2 团问题，以下结论哪些是正确的？

(a) P　　(b) NP　　(c) CoNP　　(d) NP 难
(e) CoNP 难　　(f) NP 完全　　(g) CoNP 完全

**解答：** 2 团问题可以在多项式时间内求解(可以在 $O(n^2)$时间内判断所有顶点对之间是否存在相连的边)，所以是 P、NP 和 CoNP。

**问题 8**　对于 3 团问题，以下结论哪些是正确的？

(a) P　　(b) NP　　(c) CoNP　　(d) NP 难
(e) CoNP 难　　(f) NP 完全　　(g) CoNP 完全

**解答：** 3 团问题可以在多项式时间内求解(可以在 $O(n^3)$时间内判断所有顶点组成的三元组之间是否构成三角形)，所以是 P、NP 和 CoNP。

**问题 9**　考虑一个决策问题。对于布尔表达式，检查变量的每一个赋值是否都满足该表达式。以下结论哪些是正确的？

(a) P　　(b) NP　　(c) CoNP　　(d) NP 难
(e) CoNP 难　　(f) NP 完全　　(g) CoNP 完全

**解答：** 永真式(Tautology)是可满足性问题的互补问题，可满足性问题是 NP 完全问题，因此，永真式是 CoNP 完全问题。所以它是 CoNP、CoNP 难和 CoNP 完全。

**问题 10**　若 $S$ 是一个 NP 完全问题，$Q$ 和 $R$ 两个问题不确定是否是 NP 问题。$Q$ 在多项式时间内可归约为 $S$，$S$ 在多项式时间内可归约为 $R$，以下描述哪些是正确的？

(a) $R$ 是 NP 完全　(b) $R$ 是 NP 难　　(c) $Q$ 是 NP 完全　　(d) $Q$ 是 NP 难

**解答：** $R$ 是 NP 难问题(b)。

**问题 11**　若 $A$ 是找出图 $G=(V, E)$中哈密顿回路的问题，其中 $|V|$能被 3 整除。$B$ 是判断在图 $G$ 中是否存在哈密顿回路的问题。下列说法哪一个是正确的？

(a) $A$ 和 $B$ 都是 NP 难　　(b) $A$ 是 NP 难，但 $B$ 不是
(c) $A$ 是 NP 难，但 $B$ 不是　　(d) $A$ 和 $B$ 都不是 NP 难

**解答：** $A$ 和 $B$ 都是 NP 难(a)。

**问题 12**　若 $A$ 是 NP 问题，以下说法哪一个是正确的？

(a) 不存在 $A$ 的多项式时间算法。
(b) 如果 $A$ 能够被确定性机器在多项式时间内求解，那么 P=NP。
(c) 如果 $A$ 是 NP 难问题，那么 $A$ 是 NP 完全问题。
(d) 以上结论都不成立。

**解答：** 如果 $A$ 是 NP 难问题，那么 $A$ 是 NP 完全问题(c)。

**问题 13**　假设顶点覆盖问题是 NP 完全问题。根据归约的定义，独立子集问题是否是 NP 完全问题？

**解答：是。**满足 NP 完全问题成立的两个条件：

- 独立子集问题是 NP 问题，前文已经说明。
- 已知的 NP 完全问题可以归约为该问题。

**问题 14** 假设已知独立子集问题是 NP 完全问题。根据归约的定义，顶点覆盖问题是否是 NP 完全问题？

**解答：不是。**即使将顶点覆盖问题归约为独立子集问题，也不清楚顶点覆盖问题的求解难度，这是因为独立子集问题可能比顶点覆盖问题难很多。目前尚无法对其证明。

**问题 15** NP 类型是一类不能在多项式时间内被一台确定性图灵机计算模型所接受的语言，该结论是否正确？请解释。

**解答：**

- NP 类型是一类可在多项式时间内验证的语言。
- P 类型是一类可在多项式时间内完成决策判断的语言。
- P 类型是一类可在多项式时间内被一台确定性图灵机计算模型所接受的语言。

由于 P⊆NP 和“P 类型的语言在多项式时间内可接受”，所以“NP 类型的语言在多项式时间内不可接受”的说话是错误的。

术语 NP 来自非确定性多项式时间，含义为非确定性图灵机可在多项式时间内求解，与“多项式时间内不可接受”无关。

**问题 16** 对同一个算法采用不同的编码方式可能导致不同的时间复杂度，该说法是正确的吗？

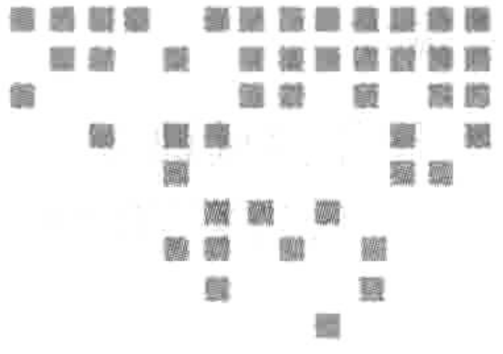
**解答：正确。**同一个算法的一元编码和二元编码的时间复杂度不同，但是如果两个编码方式是多项式相关的(例如，基 2 和基 3 编码)，则编码方式的改变不会引起时间复杂度的改变。

**问题 17** 如果 P＝NP，那么 NPC(NP 完全)⊆P。正确吗？

**解答：正确。**如果 P＝NP，那么对于任意语言 L∈NPC，L 是 NP 难的。根据 L∈NPC，有 L∈NPC⊆NP＝P⇒NPC⊆P。

**问题 18** 如果 NPC⊆P，那么 P＝NP。正确吗？

**解答：正确。**所有的 NP 问题能够在多项式时间内归约为任意一个 NPC 问题，并且因为 NPC⊆P，所以 NPC 问题可以在多项式时间求解，可推出 NP 问题在多项式时间内可求解，即 NP⊆P。又易知 P⊆NP，所以 NP＝P。



# 第21章 Chapter 21

# 杂　谈

## 21.1　引言

本章将介绍一些对于面试和考试有用的话题。

## 21.2　位运算的使用

在C和C++中，可以有效地使用位操作。首先，介绍位操作的定义，然后介绍求解实际问题的各种技巧。C和C++有6种基本的位操作。

| 符号 | 操作 |
| --- | --- |
| & | 按位与 |
| \| | 按位或 |
| ^ | 按位异或 |
| << | 按位左移 |
| >> | 按位右移 |
| ~ | 按位补 |

### 21.2.1　按位与操作

按位与(AND)操作测试两个二进制数，如果两个数对应的位值均为1，则返回值中该位值为1；如果不同时为1，则返回0。

```
   01001011
&  00010101
   ---------
   00000001
```

### 21.2.2 按位或操作

按位或(OR)操作测试两个二进制数，如果两个数对应的位值至少有一个为 1，则返回值中该位值为 1；仅当它们同时为 0 时，返回 0。

```
  01001011
| 00010101
----------
  01011111
```

### 21.2.3 按位异或操作

按位异或操作测试两个二进制数，如果两个数对应的位值不相同，则返回值中该位值为 1；如果相同，则返回 0。

```
  01001011
^ 00010101
----------
  01011110
```

### 21.2.4 按位左移操作

按位左移操作将操作数中的所有位向左移动，并用 0 填充空出的位。

```
  01001011
<< 2
--------
  00101100
```

### 21.2.5 按位右移操作

按位右移操作将操作数的所有位向右移动。

```
  01001011
>> 2
--------
  ??010010
```

注意在填充位时符号“?”的使用。左移操作中用 0 填充空出的位，右移操作仅当操作数是一个无符号数时才用 0 填充。如果操作数是有符号数，那么右移操作将用符号位或者 0(根据实现时的定义)填充空出的位。因此，最安全的做法是不要右移有符号的操作数。

### 21.2.6 按位补操作

按位补操作将一个二进制操作数的位取反。

```
  01001011
~
---------
  10110100
```

### 21.2.7 检测第 K 位是否置位

已知数 $n$，检测其第 $K$ 位可以用以下表达式：$n\&(1\ll K-1)$。如果表达式为真，则

第 $K$ 位置位(即为 1)。

**例子：**

$n=01001011$，$K=4$

| | |
|---|---|
| $1\ll K-1$ | 00001000 |
| $n\ \&\ (1\ll K-1)$ | 00001000 |

### 21.2.8　第 K 位置位

对于一个给定的操作数 $n$，设置其第 $K$ 位可以用表达式 $n\,|\,1\ll(K-1)$。

**例子：**

$n=01001011$，$K=3$

| | |
|---|---|
| $1\ll K-1$ | 00000100 |
| $n\,\|\,1\ll(K-1)$ | 01001111 |

### 21.2.9　第 K 位清零

将给定操作数 $n$ 的第 $K$ 位清零，可以用表达式 $n\ \&\ \sim(1\ll K-1)$。

**例子：**

$n=01001011$，$K=4$

| | |
|---|---|
| $1\ll K-1$ | 00001000 |
| $\sim(1\ll K-1)$ | 11110111 |
| $n\ \&\ \sim(1\ll K-1)$ | 01000011 |

### 21.2.10　切换第 K 位

切换给定操作数 $n$ 的第 $K$ 位，可以用表达式 $n\ \wedge(1\ll K-1)$。

**例子：**

$n=01001011$，$K=3$

| | |
|---|---|
| $1\ll K-1$ | 00000100 |
| $n\ \wedge(1\ll K-1)$ | 01001111 |

### 21.2.11　切换值为 1 的最右位

切换给定操作数 $n$ 的值为 1 的最右位，可以使用表达式 $n\ \&\ n-1$。

**例子：**

$n=01001011$

| | |
|---|---|
| $n-1$ | 01001010 |
| $n\ \&\ n-1$ | 01001010 |

### 21.2.12　隔离值为 1 的最右位

隔离给定操作数 $n$ 的值为 1 的最右位，可以使用表达式 $n\&-n$。

**例子：**

$n=01001011$

| | |
|---|---|
| $-n$ | 10110101 |
| $n\ \&-n$ | 00000001 |

**注意：**计算 $-n$ 时，采用补码表示，即将 $n$ 的所有位取反，再加 1。

### 21.2.13　隔离值为 0 的最右位

隔离给定操作数 $n$ 的值为 0 的最右位，可以使用表达式 $\sim n\ \&\ n+1$。

**例子：**

| | |
|---|---|
| $n$ | 01001011 |
| $\sim n$ | 10110100 |
| $n+1$ | 01001100 |
| $\sim n \ \& \ n+1$ | 00000100 |

### 21.2.14 检查某个数是否是 2 的幂

给定一个数 $n$，检查其是否满足 $2^n$ 的形式，可以使用表达式 if($n \ \& \ n-1==0$)。

**例子：**

| | |
|---|---|
| $n$ | 01001011 |
| $n-1$ | 01001010 |
| $n \ \& \ n-1$ | 01001010 |
| if($n \ \& \ n-1==0$) | 0 |

### 21.2.15 将某个数乘以 2 的幂

对于一个给定的数 $n$，将其乘以 $2^K$，可以使用表达式 $n<<K$。

**例子：** $n=00001011$，$K=2$

| | |
|---|---|
| $n<<K$ | 00101100 |

### 21.2.16 将某个数除以 2 的幂

给定操作数 $n$，将其除以 $2^K$，可以使用表达式 $n>>K$。

**例子：** $n=00001011$，$K=2$

| | |
|---|---|
| $n>>K$ | 00010010 |

### 21.2.17 找到给定操作数的模

给定操作数 $n$，计算其 8 的模(%8)，可以使用表达式 $n \ \& \ 0x7$。同理，32 的模，表达式为 $n \ \& \ 0x1F$。

**注意：** 类似地，可以计算任何数的模。

### 21.2.18 反转二进制数

给定操作数 $n$，反转其所有位(二进制数的镜像)，其代码如下：

```
unsigned int n, nReverse = n;
int s = sizeof(n);
for (; n; n >>= 1) {
        nReverse <<= 1;
        nReverse |= n & 1;
        s--;
}
nReverse <<= s;
```

时间复杂度：每位对应一次迭代，迭代次数取决于操作数的大小。

### 21.2.19 位值 1 的计数

给定一个操作数 $n$，对其二进制表示的位值 1 进行计数，可以用以下方法实现。

**方法 1：** 按位处理

```
unsigned int n, count=0;
while(n) {
        count += n & 1;
        n >>= 1;
}
```

时间复杂度：该方法每位需要一次迭代，迭代的次数取决于系统。

**方法 2**：使用取模操作

```
unsigned int n, count=0;
while(n) {
        if(n%2 ==1)
                count++;
        n = n/2;
}
```

时间复杂度：该方法每位需要一次迭代，迭代的次数取决于系统。

**方法 3**：使用切换方法：$n \,\&\, n-1$

```
unsigned int n, count=0;
while(n) {
        count++;
        n &= n - 1;
}
```

时间复杂度：迭代的次数取决于操作数中 1 的个数。

**方法 4**：采用预处理思想。在该方法中，以分组方式进行位处理。例如，如果以 4 位为一组，每次处理一个分组，可以创建一个表来列出一个分组的所有可能情况。

| | | | |
|---|---|---|---|
| 0000→0 | 0100→1 | 1000→1 | 1100→2 |
| 0001→1 | 0101→2 | 1001→2 | 1101→3 |
| 0010→1 | 0110→2 | 1010→2 | 1110→3 |
| 0011→2 | 0111→3 | 1011→3 | 1111→4 |

下面的代码给出了基于该方法的位 1 的计数算法。

```
int Table = {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};
int count = 0;
for(; n; n >>= 4)
        count = count + Table[n & 0xF];
return count;
```

时间复杂度：该方法每 4 位进行一次迭代，迭代的次数取决于系统。

### 21.2.20　创建末尾位为 0 的掩码

给定操作数 $n$，创建尾端位为 0 的掩码，可以使用表达式$(n \,\&\, -n)-1$。

**例子：**

| | |
|---|---|
| $n$ | 01001011 |
| $-n$ | 10110101 |
| $n \,\&\, -n$ | 00000001 |
| $(n \,\&\, -n)-1$ | 00000000 |

**注意**：在上例中，因为操作数的末尾没有 0，所以掩码全为 0。

### 21.2.21 交换奇偶位

**例子：**

$n$=01001011

找出操作数的奇数位(evenN)=$n$ & 0xAA　　00001010

找出操作数的偶数位(oddN) =$n$ & 0x55　　01000001

evenN>>=1　　00000101

oddN<<=1　　10000010

最终表达式：evenN|oddN　　10000111

### 21.2.22 不使用除法来计算平均数

是否存在更快的算法来取代除法操作 mid=(low+high)/2(在二分查找和归并排序中使用)？可以使用 mid=(low+high)>>1。注意在使用(low+high)/2 计算中间点时，若存在整数溢出则不能正确工作。可以采用以下方法克服溢出问题：low+((high−low)/2)，使用移位操作则为 low+((high−low)>>1)。

## 21.3 其他编程问题

**问题 1** 设计一个算法，按照螺旋顺序依次输出矩阵中的元素。

**解答：**非递归算法涉及左、右、上、下四个方向，以及相应的矩阵下标操作。一旦第一行输出完毕，方向从向右变为向下，通过增加上限将该行丢弃；一旦最后一列输出完毕，方向从向下变为向左，通过减小右限将该列丢弃。

```
void Spiral(int[][]values, int m, int n) {
                    int rowStart=0, columnStart=0;
                    int rowEnd=m-1, columnEnd=n-1;
                     while(rowStart<=rowEnd && columnStart<=columnEnd) {
                          int i=rowStart, j=columnStart;
                          for(j=columnStart; j<=columnEnd; j++) printf("%d ",values[i][j]);
                          for(i=rowStart+1, j--; i<=rowEnd; i++) printf("%d ",values[i][j]);
                          for(j=columnEnd-1, i--; j>=columnStart; j--) printf("%d ",values[i][j]);
                          for(i=rowEnd-1, j++; i>=rowStart+1; i--) printf("%d ",values[i][j]);
                          rowStart++; columnStart++; rowEnd--; columnEnd--;
                     }
}
```

时间复杂度为 $O(n^2)$。空间复杂度为 $O(1)$。

**问题 2** 设计一个洗牌算法。

**解答：**将代表 52 张牌的数组(从 0～51，无重复)打乱，如洗牌一样。首先按顺序将代表牌的数值放入数组中，然后扫描数组中每一个元素，随机选择一个元素与其交换。一个元素可能与自身交换，这不会影响算法的有效性。

```
void Shuffle(int[] cards, int n){ //假设n=52
    for (int i=0; i<n; i++)
       cards[i] = i;                                    //用卡号填充该数组
    for (int i=0; i < n; i++) {
      //这个random()方法返回0.0到1.0之间的一个随机数
       int r = i + ( Math.random() * (n - i) );  //剩余的随机位置
       int temp = cards[i]; cards[i] = cards[r]; cards[r] = temp;
    }
    System.out.println("Shuffled Cards:");
```

```
        for (int i=0; i<n; i++)
            System.out.println(cards[i]);
    }
```

时间复杂度为$O(n)$。空间复杂度为$O(1)$。

**问题3** 基于反转算法的数组旋转问题。设计一个函数rotate($A[]$，$d$，$n$)，该函数将大小为$n$的数组旋转$d$个元素。例如，数组1，2，3，4，5，6，7在经过2个元素的旋转后变为3，4，5，6，7，1，2。

**解答：** 参见以下算法。

**算法：**

```
rotate(Array[], d, n)
reverse(Array[], 1, d) ;
reverse(Array[], d + 1, n);
reverse(Array[], l, n);
```

令$AB$为输入数组的两个部分，其中$A$=Array[0..$d-1$]，$B$=Array[$d$..$n-1$]。算法的思想为：

反转$A$得到$A^rB$ /*$A^r$是$A$的反转*/

反转$B$得到$A^rB^r$ /*$B^r$是$B$的反转*/

整个反转得到$(A^rB^r)^r=BA$

例如，如果Array[]=[1，2，3，4，5，6，7]，$d$=2，$n$=7，那么$A$=[1，2]，$B$=[3，4，5，6，7]。

反转$A$，得到$A^rB$=[2，1，3，4，5，6，7]，反转$B$得$A^rB^r$=[2，1，7，6，5，4，3]

整个反转，得到$(A^rB^r)^r$=[3，4，5，6，7，1，2]。

**实现：**

```
/*在大小为n的数组Array[]中，左旋d个元素*/
void leftRotate(int[] Array, int d, int n) {
            rvereseArray(Array, 0, d-1);
            rvereseArray(Array, d, n-1);
            rvereseArray(Array, 0, n-1);
}
/*功能函数：输出一个数组*/
void printArray(int[] Array, int size){
            for(int i = 0; i < size; i++)
                        printf("%d ", Array[i]);
            printf("%\n ");
}
/*反转数组Array[]下标start到end之间元素的函数*/
void rvereseArray(int[] Array, int start, int end) {
            int i;
            int temp;
            while(start < end){
              temp = Array[start];
              Array[start] = Array[end];
              Array[end] = temp;
              start++;
              end--;
            }
}
```

**问题4** 已知数组$s[1..n]$和反转函数($s$，$i$，$j$)，反转函数颠倒$i$到$j$之间的数组元

素(包括第 $i$ 和 $j$ 个元素)的顺序。以下序列的功能是什么？($1<k\leqslant n$)

```
reverse (s, 1, k);
reverse (s, k + 1, n);
reverse (s, 1, n);
```

(a) 左旋 $s$ 的 $k$ 个元素　(b) 保持 $s$ 不变　(c) 反转 $s$ 的所有元素　(d) 以上三项都不是

**解答：**(a)。上述 3 个反转的效果相当于将大小为 $n$ 的数组左旋 $k$ 个元素(参见问题 3)。

**问题 5**　字符串由词和空格组成，设计一个程序将字符串中所有空格移到字符串的最前面，要求仅遍历数组一次，并且在原字符串中进行调整。

输入 = “move these spaces to beginning”，　输出 = “movethesespacestobeginning”

**解答：**维护两个下标变量 $i$ 和 $j$，从后向前遍历数组，如果当前下标所对应的位置是字符，则将下标 $i$ 处的字符与下标 $j$ 处的字符交换。该方法将所有空格移到数组的最前端。

```
void mySwap(char[] A,int i,int j){
    char temp=A[i];
    A[i]=A[j];
    A[j]=temp;
}
void moveSpacesToBegin(char[] A){
    int i= A.length-1;
    int j=i;
    for(; j>=0; j--){
        if(!isspace(A[j]))
            mySwap(A,i--,j);
    }
}
```

时间复杂度为 $O(n)$，其中 $n$ 是输入数组的字符个数。空间复杂度为 $O(1)$。

**问题 6**　能否降低问题 5 算法的复杂度？

**解答：**可以使用一个简单计数器代替交换操作。但该方法并不能降低整个复杂度。

```
void moveSpacesToBegin(char[] A){
    int n=A.length-1,count=n;
    int i=n;
    for(;i>=0;i--){
        if(A[i]!=' ')
            A[count--]=A[i];
    }
    while(count>=0)
        A[count--]=' ';
}
```

时间复杂度为 $O(n)$，其中 $n$ 是输入数组的字符个数。空间复杂度为 $O(1)$。

**问题 7**　对于一个包含词和空格的字符串，设计一个程序将所有空格移到字符串的末尾，要求仅遍历数组一次，并且在原字符串中进行调整。

输入 = “move these spaces to end”，　输出 = “movethesespacestoend”

**解答：**从左至右遍历数组，维护一个非空格元素的计数器。对于每一个非空格字符 $A[i]$，将其放到 $A$[count]中，然后递增 count。遍历结束后，所有非空格字符都移到前端，count 是第一个空格对应的下标。接下来，要做的就是从 count 开始用空格循环填充所有元素直至数组末尾。

```
void moveSpacesToEnd(char[] A){
    int count = 0;  // 计算非空格元素的个数
    int n = A.length-1;
    int i =0;
    for (; i <= n; i++)
        if (!isspace(A[i]))
            A[count++] = A[i];

    while (count <= n)
        A[++count] = ' ';
}
```

时间复杂度为 $O(n)$，其中 $n$ 是输入数组的字符个数。空间复杂度为 $O(1)$。

**问题 8** 移动 0 到末尾。给定由 $n$ 个整数组成的数组，将数组中的 0 移动到末尾。例如，如果有数组{1，9，8，4，0，0，2，7，0，6，0}，应该变为{1，9，8，4，2，7，6，0，0，0}。所有其他元素的顺序不变。

**解答：** 维护两个变量 $i$ 和 $j$，初始为 0。对于数组中的每一个元素 $A[i]$，如果它不是 0，那么用 $A[i]$替换元素 $A[j]$。变量 $i$ 每次迭代都递增直至 $n-1$，但仅当 $i$ 所指的元素不是 0 时 $j$ 才递增。

```
void moveZerosToEnd(int[] A){
    int i=0,j=0;
    while (i <= A.length - 1){
        if (A[i] != 0){
            A[j++] = A[i];
        }
        i++;
    }
    while (j <= size - 1)
        A[j++] = 0;
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 9** 能否降低问题 8 算法的复杂度？

**解答：** 使用简单的交换操作就能避免上述代码中的第二个 while 循环。

```
void mySwap(int[] A,int i,int j){
    int temp=A[i];
    A[i]=A[j];
    A[j]=temp;
}
void moveZerosToEnd(int[] A, int len){
    int i, j;
    for(i=0,j=0; i<len; i++)  {
        if (A[i] !=0)
            mySwap(A,j++,i);
    }
}
```

时间复杂度为 $O(n)$。空间复杂度为 $O(1)$。

**问题 10** 问题 9 的变型。已知一个包含正数和负数的数组，设计一个算法将正数和负数分开，要求正数和负数的相对顺序保持不变。输入：－5，3，2，－1，4，－8，输出：－5，－1，－8，3，4，2。

**解答：** 将函数 moveZerosToEnd 中的条件 $A[i]!=0$ 替换成 $A[i]<0$。

# 参考文献


[1] Akash. Programming Interviews. http://tech-queries. blogspot. com.

[2] Alfred V. Aho, J. E. (1983). Data Structures and Algorithms. Addison-Wesley.

[3] Algorithms. Retrieved from http://www. cs. princeton. edu/algs4/home.

[4] Anderson., S. E. Bit Twiddling Hacks. Retrieved 2010, from Bit Twiddling Hacks: http://www-graphics. stanford. edu /~seander/ bithacks. html.

[5] Bentley, J. AT&T Bell Laboratories. Retrieved from AT&T Bell Laboratories.

[6] Chen. Algorithms http://www2. hawaii. edu/~chenx.

[7] Database, P. Problem Database. Retrieved 2010, from Problem Database: datastructures. net.

[8] Drozdek, A. (1996). Data Structures and Algorithms in C++.

[9] Ellis Horowitz, S. S. Fundamentals of Data Structures.

[10] Gilles Brassard, P. B. (1996). Fundamentals of Algorithmics.

[11] Hunter., J. Introduction to Data Structures and Algorithms. Retrieved 2010, from Introduction to Data Structures and Algorithms.

[12] James F. Korsh, L. J. Data Structures, Algorithms and Program Style Using C.

[13] John Mongan, N. S. (2002). Programming Interviews Exposed.. Wiley-India.

[14] Judges. Comments on Problems and solutions. http://www. informatik. uni-ulm. de.

[15] Kalid. P, NP, and NP-Complete. Retrieved from P, NP, and NP-Complete.: http://www. cs. princeton. edu.

[16] Knuth., D. E. (1973). Fundamental Algorithms, volume 1 of The Art of Computer Programming. Addison-Wesley.

[17] Leon., J. S. Computer Algorithms. Retrieved 2010, from Computer Algorithms.: math. uic. edu..

[18] Leon., J. S. Computer Algorithms. http://www. math. uic. edu/~leon/cs-mcs401-s08.

[19] OCF. Algorithms. Retrieved 2010, from Algorithms: http://www. ocf. berkeley. edu..

[20] Parlante., N. Binary Trees. Retrieved 2010, from cslibrary. stanford. edu: cslibrary. stanford. edu..

[21] Patil., V. Fundamentals of data structures. Nirali Prakashan.

[22] Poundstone., W. HOW WOULD YOU MOVE MOUNT FUJI? New York Boston.: Little, Brown and Company..

[23] Pryor, M. Tech Interview. Retrieved 2010, from Tech Interview: http://techinterview. org.

[24] Questions, A. C. A Collection of Technical Interview Questions. Retrieved 2010, from A Collection of Technical Interview Questions: www. spellscroll. com.

[25] Sedgewick., R. (1988). Algorithms. Addison-Wesley.

[26] Sells, C. (2010). Interviewing at Microsoft. Retrieved 2010, from Interviewing at Microsoft: http://www. sellsbrothers. com/fun/msiview.

[27] Shene, C.-K. Linked Lists Merge Sort implementation. Retrieved 2010, from Linked Lists Merge Sort implementation: http://www. cs. mtu. edu/~shene.

[28] Sinha, P. Linux Journal. Retrieved 2010, from http://www. linuxjournal. com/article/6828.

[29] Structures., d. D. www. math-cs. gordon. edu. Retrieved 2010, from www. math-cs. gordon. edu.

[30] T. H. Cormen, C. E. (1997). Introduction to Algorithms. Cambridge: The MIT press.

[31] Tsiombikas, J. Pointers Explained. http://nuclear. sdf-eu. org.

[32] Warren., H. S. (2003). Hackers Delight. Addison-Wesley.
[33] Weiss., M. A. (1992). Data Structures and Algorithm Analysis in C.
[34] Wikipedia, T. F. The Free wikipedia. Retrieved from The Free wikipedia: en. wikipedia. org.
[35] Zhang., C. programheaven. Retrieved 2010, programheaven. blogspot. com.
[36] Mohammed Abualrob, Interview Code Snippets, 2010, interviewcodesnippets. com.
[37] Technical Questions. www. ihas1337code. com.