TURING 图灵原版计算机科学系列

PEARSON

计算机程序设计艺术

卷3：排序与查找

（英文版·第2版）

[美] Donald E. Knuth 著

The Art of Computer Programming

Vol 3: Sorting and Searching

Second Edition

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵原版计算机科学系列

计算机程序设计艺术
卷3：排序与查找
（英文版·第2版）

[美] Donald E. Knuth 著

The Art of Computer Programming
Vol 3: Sorti
Second Editio

人民邮电出版社
北京

图书在版编目（CIP）数据

计算机程序设计艺术 : 第2版. 第3卷, 排序与查找: 英文 / (美) 高德纳 (Knuth, D. E.) 著. -- 北京 : 人民邮电出版社, 2010.10
(图灵原版计算机科学系列)
书名原文: The Art of Computer Programming, Vol 3: Sorting and Searching
ISBN 978-7-115-23499-5

Ⅰ. ①计… Ⅱ. ①高… Ⅲ. ①程序设计－英文 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2010)第143156号

内 容 提 要

《计算机程序设计艺术》系列被公认为计算机科学领域的权威之作，深入阐述了程序设计理论，对计算机领域的发展有着极为深远的影响。本书是该系列的第3卷，扩展了第1卷中信息结构的内容，主要讲排序和查找。书中对排序和查找算法进行了详细的介绍，并对各种算法的效率做了大量的分析。

本书适合从事计算机科学、计算数学等各方面工作的人员阅读，也适合高等院校相关专业的师生作为教学参考书，对于想深入理解计算机算法的读者，是一份必不可少的珍品。

图灵原版计算机科学系列

计算机程序设计艺术 卷3：排序与查找（英文版·第2版）

◆ 著 [美] Donald E. Knuth
责任编辑 杨海玲
执行编辑 谢灵芝

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京铭成印刷有限公司印刷

◆ 开本：700×1000 1/16
印张：49.75
字数：956千字 2010年10月第1版
印数：1－2 500册 2010年10月北京第1次印刷

著作权合同登记号 图字：01-2010-4019号

ISBN 978-7-115-23499-5

定价：119.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

前　言

烹饪成了一门艺术、一门高雅的学科，
厨师是有身份的人。
——提图斯·李维的《自建城以来》，XXXIX.vi
（罗伯特·伯顿的《忧郁的解剖》，1.2.2.2 节）

本书是第 1 卷的第 2 章中有关信息结构内容的自然延续，因为它为其他基本结构化思想增加了线性有序数据的概念。

书名中的“排序与查找”可能会使人误以为本书面向的只是从事一般性排序工作或信息检索工作的系统程序员。事实上，排序与查找为讨论众多重要的一般性问题提供了一个理想的框架：

- 好算法是怎么发现的？
- 如何改进给定的算法和程序？
- 如何从数学的角度分析算法的效率？
- 对于给定的任务，如何在不同的算法之间做出合理的选择？
- 在什么意义下，可以证明算法是“最可行的”？
- 计算理论同实际考虑如何相互影响？
- 如何将磁带、磁鼓或磁盘这样的外存有效应用于大型数据库？

事实上，我认为程序设计的几乎所有重要的方面都与排序或查找有关！

本卷包含整套书中的第 5 章和第 6 章。第 5 章讨论排序，这个大问题主要划分成两个部分，即内部排序和外部排序。此外，这一章还有几个辅助性小节，讨论了有关排列（5.1 节）和最优排序方法（5.3 节）的辅助理论。第 6 章讨论在表或文件中查找特定项的问题，该问题可以分为顺序查找、通过比较键进行查找、按数位性质进行查找以及散列法查找等，然后讨论了更难的辅键查找问题。这两章内容有着惊人的相互影响和很强的相似性。除了第 2 章介绍的信息结构外，本书还讨论了两种重要的信息结构，即优先队列（5.2.3 节）和表示成平衡树的线性表（6.2.3 节）。

与第 1 卷和第 2 卷一样，本书包含了其他出版物所没有的许多内容。许多人曾经以书面或口头的形式向我表达了他们的思想，我希望在用自己的语言表述时没有过度地歪曲他们的原意。

我一直没有时间系统地检索专利文献。事实上，我对当前为算法申请专利的做法深感不满（见 5.4.5 节）。如果有人把本书中没有引用的相关专利发给我一份，我会在未来的版本中加以忠实的引用。不过，我还是希望人们继续发扬几百年以来形成的优良的数学传统，让新发现的算法进入公众领域。防范他人利用我们在计算机科学领域所做的贡献，并不是一种最好的谋生手段。

在从教学岗位上退下来之前，我曾用本书给大学三年级学生和研究生讲授第二门数据结构课，教学过程中省略了大部分数学内容。我还曾把本书的数学部分用作研究生算法分析课程的基础，并特别强调了 5.1 节、5.2.2 节、6.3 节和 6.4 节。也可以以 5.3 节、5.4.4 节以及第 2 卷的 4.3.3 节、4.6.3 节和 4.6.4 节的内容为基础，开设有关具体的计算复杂性的研究生课程。

除少数内容与第 1 卷介绍的 MIX 计算机有关之外，本书的大部分内容都是自成一体的。附录 B 列出了本书用到的数学符号，其中有些与传统数学书中的符号略有不同。

第 2 版前言

这一新版本与第 1 卷和第 2 卷的第 3 版一样，都使用 TEX 和 METAFONT 进行排版，以此纪念这两个排版系统的完成。

将内容转换成电子格式使我能够借此机会逐个审阅文字和标点符号。我力图在保持原有的蓬勃朝气的同时加入一些可能更成熟的论断。新版增加了几十道新的习题，并为原有的几十道习题给出了改进过的新答案。改动可谓无处不在，但最重要的改动集中在 5.1.4 节（关于排列和图表）、5.3 节（关于最优排序）、5.4.9 节（关于磁盘排序）、6.2.2 节（关于熵）、6.4 节（关于通用散列法）和 6.5 节（关于多维树和 Tries）。

尽管如此，《计算机程序设计艺术》这套书尚未完稿，而有关排序与查找的研究也还在快速发展。因此书中有些部分还带有“建设中”的图标，以向读者致歉——这部分内容还不是最新的。例如，如果我再给本科生讲授数据结构，我一定会介绍 Treap 这样的随机化结构，但现在我只能引用一些重要的论文并预告未来 6.2.5 节的内容安排。我电脑中的文件里堆满了重要的材料，打算写进第 3 卷最终的壮丽无比的第 3 版中，或许从现在算起还需要 17 年的时间。但我必须先完成第 4 卷和第 5 卷，非到万不得已，我不想拖延这两卷的出版时间。

非常感谢在过去 35 年中帮助我搜集素材和改进内容的数百人。准备这一新版本的大部分艰苦工作是由 Phyllis Winkler、Silvio Levy 和 Jeffrey Oldham 完成的，Phyllis 把第 1 版的文本转化为 TEX 格式，Silvio 编辑了这一版的文字内容，并绘制了好几十幅插图，而 Jeffrey 把原有的 250 多幅插图都转换为了 METAPOST 格式。Addison-Wesley 排版印制部门的员工也一如既往地提供了很多帮助。

我修正了细心的读者在第 1 版中发现的所有错误（以及读者未能察觉的一些错

误），并尽量避免在新增内容中引入新的错误。但我想可能仍然有一些缺陷，我希望能尽快加以改正。因此，我很乐意向首先发现技术性错误、印刷错误或历史知识错误的人按每处错误 2.56 美元支付报酬。下列网页上列出了所有已反馈给我的错误的最新更正：http://www-cs-faculty.stanford.edu/~knuth/taocp.html。

高德纳（Donald E. Knuth）

1998 年 2 月于加利福尼亚州斯坦福市

作者总是有一些特权，
一个好处是，读者通常没有理由来怀疑你。
尤其是别人不理解你的地方，可以这样下结论：
此中有真意啊！
——斯威夫特《桶的故事》前言（1704）

NOTES ON THE EXERCISES

THE EXERCISES in this set of books have been designed for self-study as well as classroom study. It is difficult, if not impossible, for anyone to learn a subject purely by reading about it, without applying the information to specific problems and thereby being encouraged to think about what has been read. Furthermore, we all learn best the things that we have discovered for ourselves. Therefore the exercises form a major part of this work; a definite attempt has been made to keep them as informative as possible and to select problems that are enjoyable as well as instructive.

In many books, easy exercises are found mixed randomly among extremely difficult ones. This is sometimes unfortunate because readers like to know in advance how long a problem ought to take—otherwise they may just skip over all the problems. A classic example of such a situation is the book *Dynamic Programming* by Richard Bellman; this is an important, pioneering work in which a group of problems is collected together at the end of some chapters under the heading "Exercises and Research Problems," with extremely trivial questions appearing in the midst of deep, unsolved problems. It is rumored that someone once asked Dr. Bellman how to tell the exercises apart from the research problems, and he replied, "If you can solve it, it is an exercise; otherwise it's a research problem."

Good arguments can be made for including both research problems and very easy exercises in a book of this kind; therefore, to save the reader from the possible dilemma of determining which are which, *rating numbers* have been provided to indicate the level of difficulty. These numbers have the following general significance:

Rating *Interpretation*

00 An extremely easy exercise that can be answered immediately if the material of the text has been understood; such an exercise can almost always be worked "in your head."

10 A simple problem that makes you think over the material just read, but is by no means difficult. You should be able to do this in one minute at most; pencil and paper may be useful in obtaining the solution.

20 An average problem that tests basic understanding of the text material, but you may need about fifteen or twenty minutes to answer it completely.

30 A problem of moderate difficulty and/or complexity; this one may involve more than two hours' work to solve satisfactorily, or even more if the TV is on.

40 Quite a difficult or lengthy problem that would be suitable for a term project in classroom situations. A student should be able to solve the problem in a reasonable amount of time, but the solution is not trivial.

50 A research problem that has not yet been solved satisfactorily, as far as the author knew at the time of writing, although many people have tried. If you have found an answer to such a problem, you ought to write it up for publication; furthermore, the author of this book would appreciate hearing about the solution as soon as possible (provided that it is correct).

By interpolation in this "logarithmic" scale, the significance of other rating numbers becomes clear. For example, a rating of *17* would indicate an exercise that is a bit simpler than average. Problems with a rating of *50* that are subsequently solved by some reader may appear with a *45* rating in later editions of the book, and in the errata posted on the Internet.

The remainder of the rating number divided by 5 indicates the amount of detailed work required. Thus, an exercise rated *24* may take longer to solve than an exercise that is rated *25*, but the latter will require more creativity.

The author has tried earnestly to assign accurate rating numbers, but it is difficult for the person who makes up a problem to know just how formidable it will be for someone else to find a solution; and everyone has more aptitude for certain types of problems than for others. It is hoped that the rating numbers represent a good guess at the level of difficulty, but they should be taken as general guidelines, not as absolute indicators.

This book has been written for readers with varying degrees of mathematical training and sophistication; as a result, some of the exercises are intended only for the use of more mathematically inclined readers. The rating is preceded by an *M* if the exercise involves mathematical concepts or motivation to a greater extent than necessary for someone who is primarily interested only in programming the algorithms themselves. An exercise is marked with the letters "*HM*" if its solution necessarily involves a knowledge of calculus or other higher mathematics not developed in this book. An "*HM*" designation does *not* necessarily imply difficulty.

Some exercises are preceded by an arrowhead, "▶"; this designates problems that are especially instructive and especially recommended. Of course, no reader/student is expected to work *all* of the exercises, so those that seem to be the most valuable have been singled out. (This is not meant to detract from the other exercises!) Each reader should at least make an attempt to solve all of the problems whose rating is *10* or less; and the arrows may help to indicate which of the problems with a higher rating should be given priority.

Solutions to most of the exercises appear in the answer section. Please use them wisely; do not turn to the answer until you have made a genuine effort to

solve the problem by yourself, or unless you absolutely do not have time to work this particular problem. *After* getting your own solution or giving the problem a decent try, you may find the answer instructive and helpful. The solution given will often be quite short, and it will sketch the details under the assumption that you have earnestly tried to solve it by your own means first. Sometimes the solution gives less information than was asked; often it gives more. It is quite possible that you may have a better answer than the one published here, or you may have found an error in the published solution; in such a case, the author will be pleased to know the details. Later editions of this book will give the improved solutions together with the solver's name where appropriate.

When working an exercise you may generally use the answers to previous exercises, unless specifically forbidden from doing so. The rating numbers have been assigned with this in mind; thus it is possible for exercise $n + 1$ to have a lower rating than exercise n, even though it includes the result of exercise n as a special case.

Summary of codes:			
		00	Immediate
		10	Simple (one minute)
		20	Medium (quarter hour)
▸	Recommended	*30*	Moderately hard
M	Mathematically oriented	*40*	Term project
HM	Requiring "higher math"	*50*	Research problem

EXERCISES

▸ **1.** [*00*] What does the rating "*M20*" mean?

2. [*10*] Of what value can the exercises in a textbook be to the reader?

3. [*HM45*] Prove that when n is an integer, $n > 2$, the equation $x^n + y^n = z^n$ has no solution in positive integers x, y, z.

Two hours' daily exercise ... will be enough to keep a hack fit for his work.

— M. H. MAHON, *The Handy Horse Book* (1865)

CONTENTS

CHAPTER FIVE

SORTING

There is nothing more difficult to take in hand,
more perilous to conduct, or more uncertain in its success,
than to take the lead in the introduction of
a new order of things.

— NICCOLÒ MACHIAVELLI, *The Prince* (1513)

"But you can't look up all those license
numbers in time," Drake objected.
"We don't have to, Paul. We merely arrange a list
and look for duplications."

— PERRY MASON, in *The Case of the Angry Mourner* (1951)

"Treesort" Computer — With this new 'computer-approach'
to nature study you can quickly identify over 260
different trees of U.S., Alaska, and Canada,
even palms, desert trees, and other exotics.
To sort, you simply insert the needle.

— EDMUND SCIENTIFIC COMPANY, *Catalog* (1964)

IN THIS CHAPTER we shall study a topic that arises frequently in programming: the rearrangement of items into ascending or descending order. Imagine how hard it would be to use a dictionary if its words were not alphabetized! We will see that, in a similar way, the order in which items are stored in computer memory often has a profound influence on the speed and simplicity of algorithms that manipulate those items.

Although dictionaries of the English language define "sorting" as the process of separating or arranging things according to class or kind, computer programmers traditionally use the word in the much more special sense of marshaling things into ascending or descending order. The process should perhaps be called *ordering*, not sorting; but anyone who tries to call it "ordering" is soon led into confusion because of the many different meanings attached to that word. Consider the following sentence, for example: "Since only two of our tape drives were in working order, I was ordered to order more tape units in short order, in order to order the data several orders of magnitude faster." Mathematical terminology abounds with still more senses of order (the order of a group, the order of a permutation, the order of a branch point, relations of order, etc., etc.). Thus we find that the word "order" can lead to chaos.

Some people have suggested that "sequencing" would be the best name for the process of sorting into order; but this word often seems to lack the right

connotation, especially when equal elements are present, and it occasionally conflicts with other terminology. It is quite true that "sorting" is itself an overused word ("I was sort of out of sorts after sorting that sort of data"), but it has become firmly established in computing parlance. Therefore we shall use the word "sorting" chiefly in the strict sense of sorting into order, without further apologies.

Some of the most important applications of sorting are:

a) *Solving the "togetherness" problem*, in which all items with the same identification are brought together. Suppose that we have 10000 items in arbitrary order, many of which have equal values; and suppose that we want to rearrange the data so that all items with equal values appear in consecutive positions. This is essentially the problem of sorting in the older sense of the word; and it can be solved easily by sorting the file in the new sense of the word, so that the values are in ascending order, $v_1 \le v_2 \le \cdots \le v_{10000}$. The efficiency achievable in this procedure explains why the original meaning of "sorting" has changed.

b) *Matching items in two or more files.* If several files have been sorted into the same order, it is possible to find all of the matching entries in one sequential pass through them, without backing up. This is the principle that Perry Mason used to help solve a murder case (see the quotation at the beginning of this chapter). We can usually process a list of information most quickly by traversing it in sequence from beginning to end, instead of skipping around at random in the list, unless the entire list is small enough to fit in a high-speed random-access memory. Sorting makes it possible to use sequential accessing on large files, as a feasible substitute for direct addressing.

c) *Searching for information by key values.* Sorting is also an aid to searching, as we shall see in Chapter 6, hence it helps us make computer output more suitable for human consumption. In fact, a listing that has been sorted into alphabetic order often looks quite authoritative even when the associated numerical information has been incorrectly computed.

Although sorting has traditionally been used mostly for business data processing, it is actually a basic tool that every programmer should keep in mind for use in a wide variety of situations. We have discussed its use for simplifying algebraic formulas, in exercise 2.3.2–17. The exercises below illustrate the diversity of typical applications.

One of the first large-scale software systems to demonstrate the versatility of sorting was the LARC Scientific Compiler developed by J. Erdwinn, D. E. Ferguson, and their associates at Computer Sciences Corporation in 1960. This optimizing compiler for an extended FORTRAN language made heavy use of sorting so that the various compilation algorithms were presented with relevant parts of the source program in a convenient sequence. The first pass was a lexical scan that divided the FORTRAN source code into individual tokens, each representing an identifier or a constant or an operator, etc. Each token was assigned several sequence numbers; when sorted on the name and an appropriate sequence number, all the uses of a given identifier were brought together. The

"defining entries" by which a user would specify whether an identifier stood for a function name, a parameter, or a dimensioned variable were given low sequence numbers, so that they would appear first among the tokens having a given identifier; this made it easy to check for conflicting usage and to allocate storage with respect to `EQUIVALENCE` declarations. The information thus gathered about each identifier was now attached to each token; in this way no "symbol table" of identifiers needed to be maintained in the high-speed memory. The updated tokens were then sorted on another sequence number, which essentially brought the source program back into its original order except that the numbering scheme was cleverly designed to put arithmetic expressions into a more convenient "Polish prefix" form. Sorting was also used in later phases of compilation, to facilitate loop optimization, to merge error messages into the listing, etc. In short, the compiler was designed so that virtually all the processing could be done sequentially from files that were stored in an auxiliary drum memory, since appropriate sequence numbers were attached to the data in such a way that it could be sorted into various convenient arrangements.

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms have been in common use. The real truth probably involves all three of these possibilities, but in any event we can see that sorting is worthy of serious study, as a practical matter.

Even if sorting were almost useless, there would be plenty of rewarding reasons for studying it anyway! The ingenious algorithms that have been discovered show that sorting is an extremely interesting topic to explore in its own right. Many fascinating unsolved problems remain in this area, as well as quite a few solved ones.

From a broader perspective we will find also that sorting algorithms make a valuable *case study* of how to attack computer programming problems in general. Many important principles of data structure manipulation will be illustrated in this chapter. We will be examining the evolution of various sorting techniques in an attempt to indicate how the ideas were discovered in the first place. By extrapolating this case study we can learn a good deal about strategies that help us design good algorithms for other computer problems.

Sorting techniques also provide excellent illustrations of the general ideas involved in the *analysis of algorithms*—the ideas used to determine performance characteristics of algorithms so that an intelligent choice can be made between competing methods. Readers who are mathematically inclined will find quite a few instructive techniques in this chapter for estimating the speed of computer algorithms and for solving complicated recurrence relations. On the other hand, the material has been arranged so that readers without a mathematical bent can safely skip over these calculations.

Before going on, we ought to define our problem a little more clearly, and introduce some terminology. We are given N items

$$R_1, R_2, \ldots, R_N$$

to be sorted; we shall call them *records*, and the entire collection of N records will be called a *file*. Each record R_j has a *key*, K_j, which governs the sorting process. Additional data, besides the key, is usually also present; this extra "satellite information" has no effect on sorting except that it must be carried along as part of each record.

An ordering relation "$<$" is specified on the keys so that the following conditions are satisfied for any key values a, b, c:

i) Exactly one of the possibilities $a < b$, $a = b$, $b < a$ is true. (This is called the law of trichotomy.)

ii) If $a < b$ and $b < c$, then $a < c$. (This is the familiar law of transitivity.)

Properties (i) and (ii) characterize the mathematical concept of *linear ordering*, also called *total ordering*. Any relationship "$<$" satisfying these two properties can be sorted by most of the methods to be mentioned in this chapter, although some sorting techniques are designed to work only with numerical or alphabetic keys that have the usual ordering.

The goal of sorting is to determine a permutation $p(1)\,p(2)\ldots p(N)$ of the indices $\{1, 2, \ldots, N\}$ that will put the keys into nondecreasing order:

$$K_{p(1)} \le K_{p(2)} \le \cdots \le K_{p(N)}. \tag{1}$$

The sorting is called *stable* if we make the further requirement that records with equal keys should retain their original relative order. In other words, stable sorting has the additional property that

$$p(i) < p(j) \qquad \text{whenever} \quad K_{p(i)} = K_{p(j)} \quad \text{and} \quad i < j. \tag{2}$$

In some cases we will want the records to be physically rearranged in storage so that their keys are in order. But in other cases it will be sufficient merely to have an auxiliary table that specifies the permutation in some way, so that the records can be accessed in order of their keys.

A few of the sorting methods in this chapter assume the existence of either or both of the values "∞" and "$-\infty$", which are defined to be greater than or less than all keys, respectively:

$$-\infty < K_j < \infty, \qquad \text{for } 1 \le j \le N. \tag{3}$$

Such extreme values are occasionally used as artificial keys or as sentinel indicators. The case of equality is excluded in (3); if equality can occur, the algorithms can be modified so that they will still work, but usually at the expense of some elegance and efficiency.

Sorting can be classified generally into *internal sorting*, in which the records are kept entirely in the computer's high-speed random-access memory, and *external sorting*, when more records are present than can be held comfortably in

memory at once. Internal sorting allows more flexibility in the structuring and accessing of the data, while external sorting shows us how to live with rather stringent accessing constraints.

The time required to sort N records, using a decent general-purpose sorting algorithm, is roughly proportional to $N \log N$; we make about $\log N$ "passes" over the data. This is the minimum possible time, as we shall see in Section 5.3.1, if the records are in random order and if sorting is done by pairwise comparisons of keys. Thus if we double the number of records, it will take a little more than twice as long to sort them, all other things being equal. (Actually, as N approaches infinity, a better indication of the time needed to sort is $N(\log N)^2$, if the keys are distinct, since the size of the keys must grow at least as fast as $\log N$; but for practical purposes, N never really approaches infinity.)

On the other hand, if the keys are known to be randomly distributed with respect to some continuous numerical distribution, we will see that sorting can be accomplished in $O(N)$ steps on the average.

EXERCISES — First Set

1. [*M20*] Prove, from the laws of trichotomy and transitivity, that the permutation $p(1)\,p(2)\ldots p(N)$ is *uniquely* determined when the sorting is assumed to be stable.

2. [*21*] Assume that each record R_j in a certain file contains *two* keys, a "major key" K_j and a "minor key" k_j, with a linear ordering $<$ defined on each of the sets of keys. Then we can define *lexicographic order* between pairs of keys (K, k) in the usual way:

$$(K_i, k_i) < (K_j, k_j) \quad \text{if} \quad K_i < K_j \quad \text{or if} \quad K_i = K_j \quad \text{and} \quad k_i < k_j.$$

Alice took this file and sorted it first on the major keys, obtaining n groups of records with equal major keys in each group,

$$K_{p(1)} = \cdots = K_{p(i_1)} < K_{p(i_1+1)} = \cdots = K_{p(i_2)} < \cdots < K_{p(i_{n-1}+1)} = \cdots = K_{p(i_n)},$$

where $i_n = N$. Then she sorted each of the n groups $R_{p(i_{j-1}+1)}, \ldots, R_{p(i_j)}$ on their minor keys.

Bill took the same original file and sorted it first on the minor keys; then he took the resulting file, and sorted it on the major keys.

Chris took the same original file and did a single sorting operation on it, using lexicographic order on the major and minor keys (K_j, k_j).

Did everyone obtain the same result?

3. [*M25*] Let $<$ be a relation on $K_1, \ldots, K_N$ that satisfies the law of trichotomy but *not* the transitive law. Prove that even without the transitive law it is possible to sort the records in a stable manner, meeting conditions (1) and (2); in fact, there are at least three arrangements that satisfy the conditions!

▸ **4.** [*21*] Lexicographers don't actually use strict lexicographic order in dictionaries, because uppercase and lowercase letters must be interfiled. Thus they want an ordering such as this:

$$\mathrm{a} < \mathrm{A} < \mathrm{aa} < \mathrm{AA} < \mathrm{AAA} < \mathrm{Aachen} < \mathrm{aah} < \cdots < \mathrm{zzz} < \mathrm{ZZZ}.$$

Explain how to implement dictionary order.