图灵原版计算机科学系列

PEARSON
Addison Wesley

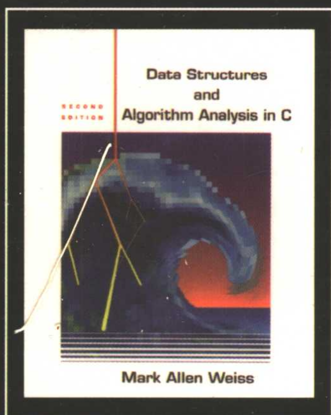
Data Structures and Algorithm Analysis in C (Second Edition)

数据结构与算法分析

——C语言描述

（英文版·第2版）

[美] Mark Allen Weiss 著
陈 越 改编



人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目（CIP）数据

数据结构与算法分析：C 语言描述：第 2 版/（美）维斯（Weiss, M.A.）著.
—北京：人民邮电出版社，2005.8
（图灵原版. 计算机科学系列）

ISBN 7-115-13984-9

Ⅰ. 数… Ⅱ. 维… Ⅲ. ①数据结构－英文 ②算法分析－英文 ③C 语言－程序设计－英文 Ⅳ. TP311.12

中国版本图书馆 CIP 数据核字（2005）第 092227 号

图灵原版计算机科学系列

数据结构与算法分析——C 语言描述（英文版・第 2 版）

◆ 著 [美] Mark Allen Weiss
改 编 陈 越
责任编辑 陈贤舜

◆ 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
北京市大中印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：800×1000 1/16
印张：32.5
字数：727 千字 2005 年 8 月第 1 版
印数：1－3 000 册 2005 年 8 月北京第 1 次印刷
著作权合同登记号 图字：01-2005-3578 号
ISBN 7-115-13984-9/TP・4957

定价：49.00 元

读者服务热线：(010)88593802 印装质量热线：(010)67129223

Adapter's Foreword

Purpose

The original of this book is an excellent work of Mark Allen Weiss. All the fundamental topics are covered. The ADT concepts and the analysis of the algorithms (especially the average case analysis) are emphasized. The extensive examples are also quite helpful to the students.

Till now the original book has been introduced to Chinese students for two years and has received positive feedbacks from many instructors and students. This re-composition is made to trim the contents of the book so that it better fits a second-year undergraduate course in data structures and algorithm analysis for the Chinese students.

What's New

The recomposition includes two major structure changes. First, the review section of mathematics has been canceled since sophomore students in China have taken sufficient courses in mathematics in their first-year study, including calculus, linear algebra, and discrete mathematics. Secondly, the original Chapter 5 is moved to follow Chapter 7, in order to show hashing as a method to break the lower bound of searching by comparisons only.

Other minor changes include adding some interesting data structures and methods, and rearranging part of the contents. Introduction of the sparse matrix representation is added as an example of application of multilists in Section 3.2. At the mean time, bucket sort and radix sort are discussed in more details in Chapter 6 (which was Chapter 7 in the original book) instead of being given as an example in Section 3.2. In Chapter 4, the two sections about tree traversals, namely Sections 4.1.2 and 4.6, are merged into one and are inserted into Section 4.2.3. Threaded binary tree is then formally introduced instead of being mentioned in exercises only. At the beginning of Chapter 7 (which was Chapter 5 in the original book), *Hashing*, a method called ***interpolation search*** is briefly discussed to make the point that it is possible to break the lower bound if we search by methods other than comparisons. Finally in Section 6.8, *Sorting Large Structures*, we introduce ***table sort*** as a method to handle the case in which physically sorting large structures is required.

Acknowledgments

We feel grateful to Mark Allen Weiss, the author of the original book, and Pearson Education, the original publisher, for their great support on this recomposition. It is their understanding and generosity that make it possible for more Chinese students to enjoy this distinguished book.

Thanks to all the colleagues and students who have communicated with me regarding to their impressions of the original book. Special thanks go to Professor Qinming He, for his helpful feedbacks and suggestions.

Finally I would like to thank everyone at Turing Book Company, the publisher of this adapted edition, who have put in great effort to make this kind of cooperation possible.

It is my very first attempt on making a recomposition of a textbook. If you have any suggestions for improvements, I would very much appreciate your comments.

Yue Chen
chenyue@cs.zju.edu.cn
Zhejiang University

PREFACE

Purpose/Goals

This book describes *data structures*, methods of organizing large amounts of data, and *algorithm analysis*, the estimation of the running time of algorithms. As computers become faster and faster, the need for programs that can handle large amounts of input becomes more acute. Paradoxically, this requires more careful attention to efficiency, since inefficiencies in programs become most obvious when input sizes are large. By analyzing an algorithm before it is actually coded, students can decide if a particular solution will be feasible. For example, in this text students look at specific problems and see how careful implementations can reduce the time constraint for large amounts of data from 16 years to less than a second. Therefore, no algorithm or data structure is presented without an explanation of its running time. In some cases, minute details that affect the running time of the implementation are explored.

Once a solution method is determined, a program must still be written. As computers have become more powerful, the problems they must solve have become larger and more complex, requiring development of more intricate programs. The goal of this text is to teach students good programming and algorithm analysis skills simultaneously so that they can develop such programs with the maximum amount of efficiency.

This book is suitable for either an advanced data structures (CS7) course or a first-year graduate course in algorithm analysis. Students should have some knowledge of intermediate programming, including such topics as pointers and recursion, and some background in discrete math.

Approach

I believe it is important for students to learn how to program for themselves, not how to copy programs from a book. On the other hand, it is virtually impossible to discuss realistic programming issues without including sample code. For this reason, the book usually provides about one-half to three-quarters of an implementation, and the student is encouraged to supply the rest. Chapter 12, which is new to this edition, discusses additional data structures with an emphasis on implementation details.

The algorithms in this book are presented in ANSI C, which, despite some flaws, is arguably the most popular systems programming language. The use of C instead of Pascal allows the use of dynamically allocated arrays (see, for instance, rehashing in Chapter 7). It also produces simplified code in several places, usually because the *and* (&&) operation is short-circuited.

Most criticisms of C center on the fact that it is easy to write code that is barely readable. Some of the more standard tricks, such as the simultaneous assignment and testing against 0 via

```
if (x=y)
```

are generally not used in the text, since the loss of clarity is compensated by only a few keystrokes and no increased speed. I believe that this book demonstrates that unreadable code can be avoided by exercising reasonable care.

Overview

Chapter 1 contains review material on recursion. I believe the only way to be comfortable with recursion is to see good uses over and over. Therefore, recursion is prevalent in this text, with examples in every chapter except Chapter 7.

Chapter 2 deals with algorithm analysis. This chapter explains asymptotic analysis and its major weaknesses. Many examples are provided, including an in-depth explanation of logarithmic running time. Simple recursive programs are analyzed by intuitively converting them into iterative programs. More complicated divide-and-conquer programs are introduced, but some of the analysis (solving recurrence relations) is implicitly delayed until Chapter 6, where it is performed in detail.

Chapter 3 covers lists, stacks, and queues. The emphasis here is on coding these data structures using ADTs, fast implementation of these data structures, and an exposition of some of their uses. There are almost no programs (just routines), but the exercises contain plenty of ideas for programming assignments.

Chapter 4 covers trees, with an emphasis on search trees, including external search trees (B-trees). The UNIX file system and expression trees are used as examples. AVL trees and splay trees are introduced but not analyzed. Seventy-five percent of the code is written, leaving similar cases to be completed by the student. More careful treatment of search tree implementation details is found in Chapter 12. Additional coverage of trees, such as file compression and game trees, is deferred until Chapter 10. Data structures for an external medium are considered as the final topic in several chapters.

Chapter 5 is about priority queues. Binary heaps are covered, and there is additional material on some of the theoretically interesting implementations of priority queues. The Fibonacci heap is discussed in Chapter 11, and the pairing heap is discussed in Chapter 12.

Chapter 6 covers sorting. It is very specific with respect to coding details and analysis. All the important general-purpose sorting algorithms are covered and compared. Four algorithms are analyzed in detail: insertion sort, Shellsort, heapsort, and quicksort. The analysis of the average-case running time of heapsort is new to this edition. External sorting is covered at the end of the chapter.

Chapter 7 is a relatively short chapter concerning hash tables. Some analysis is performed, and extendible hashing is covered at the end of the chapter.

Chapter 8 discusses the disjoint set algorithm with proof of the running time. This is a short and specific chapter that can be skipped if Kruskal's algorithm is not discussed.

Chapter 9 covers graph algorithms. Algorithms on graphs are interesting, not only because they frequently occur in practice but also because their running time is so heavily dependent on the proper use of data structures. Virtually all of the standard algorithms are presented along with appropriate data structures, pseudocode, and analysis of running time. To place these problems in a proper context, a short discussion on complexity theory (including *NP*-completeness and undecidability) is provided.

Chapter 10 covers algorithm design by examining common problem-solving techniques. This chapter is heavily fortified with examples. Pseudocode is used in these later chapters so that the student's appreciation of an example algorithm is not obscured by implementation details.

Chapter 11 deals with amortized analysis. Three data structures from Chapters 4 and 5 and the Fibonacci heap, introduced in this chapter, are analyzed.

Chapter 12 is new to this edition. It covers search tree algorithms, the *k*-d tree, and the pairing heap. This chapter departs from the rest of the text by providing complete and careful implementations for the search trees and pairing heap. The material is structured so that the instructor can integrate sections into discussions from other chapters. For example, the top-down red black tree in Chapter 12 can be discussed under AVL trees (in Chapter 4).

Chapters 1–9 provide enough material for most one-semester data structures courses. If time permits, then Chapter 10 can be covered. A graduate course on algorithm analysis could cover Chapters 6–11. The advanced data structures analyzed in Chapter 11 can easily be referred to in the earlier chapters. The discussion of *NP*-completeness in Chapter 9 is far too brief to be used in such a course. Garey and Johnson's book on *NP*-completeness can be used to augment this text.

Exercises

Exercises, provided at the end of each chapter, match the order in which material is presented. The last exercises may address the chapter as a whole rather than a specific section. Difficult exercises are marked with an asterisk, and more challenging exercises have two asterisks.

References

References are placed at the end of each chapter. Generally the references either are historical, representing the original source of the material, or they represent extensions and improvements to the results given in the text. Some references represent solutions to exercises.

Code Availability

The example program code in this book is available via anonymous ftp at ftp://ftp.cs.fiu.edu/pub/weiss/WEISS_2E.tar.Z

Acknowledgments

Many, many people have helped me in the preparation of books in this series. Some are listed in other versions of the book; thanks to all.

For this edition, I would like to thank my editors at Addison-Wesley, Carter Shanklin and Susan Hartman. Teri Hyde did another wonderful job with the production, and Matthew Harris and his staff at Publication Services did their usual fine work putting the final pieces together.

M.A.W.

Miami, Florida
July, 1996

CONTENTS

CHAPTER 1

Introduction

In this chapter, we discuss the aims and goals of this text and briefly review programming concepts. We will

- See that how a program performs for reasonably large input is just as important as its performance on moderate amounts of input.
- Briefly review recursion.

1.1. What's the Book About?

Suppose you have a group of N numbers and would like to determine the kth largest. This is known as the *selection problem*. Most students who have had a programming course or two would have no difficulty writing a program to solve this problem. There are quite a few "obvious" solutions.

One way to solve this problem would be to read the N numbers into an array, sort the array in decreasing order by some simple algorithm such as bubblesort, and then return the element in position k.

A somewhat better algorithm might be to read the first k elements into an array and sort them (in decreasing order). Next, each remaining element is read one by one. As a new element arrives, it is ignored if it is smaller than the kth element in the array. Otherwise, it is placed in its correct spot in the array, bumping one element out of the array. When the algorithm ends, the element in the kth position is returned as the answer.

Both algorithms are simple to code, and you are encouraged to do so. The natural questions, then, are which algorithm is better and, more important, is either algorithm good enough? A simulation using a random file of 1 million elements and $k = 500{,}000$ will show that neither algorithm finishes in a reasonable amount of time; each requires several days of computer processing to terminate (albeit eventually with a correct answer). An alternative method, discussed in Chapter 6, gives a solution in about a second. Thus, although our proposed algorithms work, they cannot be considered good algorithms, because they are entirely impractical for input sizes that a third algorithm can handle in a reasonable amount of time.

A second problem is to solve a popular word puzzle. The input consists of a two-dimensional array of letters and a list of words. The object is to find the words in the puzzle. These words may be horizontal, vertical, or diagonal in any direction. As an example, the puzzle shown in Figure 1.1 contains the words *this, two, fat,* and *that*. The word *this* begins at row 1, column 1, or (1,1), and extends to (1,4); *two* goes from (1,1) to (3,1); *fat* goes from (4,1) to (2,3); and *that* goes from (4,4) to (1,1).

Again, there are at least two straightforward algorithms that solve the problem. For each word in the word list, we check each ordered triple (*row, column, orientation*) for the presence of the word. This amounts to lots of nested *for* loops but is basically straightforward.

Alternatively, for each ordered quadruple (*row, column, orientation, number of characters*) that doesn't run off an end of the puzzle, we can test whether the word indicated is in the word list. Again, this amounts to lots of nested *for* loops. It is possible to save some time if the maximum number of characters in any word is known.

It is relatively easy to code up either method of solution and solve many of the real-life puzzles commonly published in magazines. These typically have 16 rows, 16 columns, and 40 or so words. Suppose, however, we consider the variation where only the puzzle board is given and the word list is essentially an English dictionary. Both of the solutions proposed require considerable time to solve this problem and therefore are not acceptable. However, it is possible, even with a large word list, to solve the problem in a matter of seconds.

An important concept is that, in many problems, writing a working program is not good enough. If the program is to be run on a large data set, then the running time becomes an issue. Throughout this book we will see how to estimate the running time of a program for large inputs and, more important, how to compare the running times of two programs without actually coding them. We will see techniques for drastically improving the speed of a program and for determining program bottlenecks. These techniques will enable us to find the section of the code on which to concentrate our optimization efforts.

Figure 1.1 Sample word puzzle

	1	2	3	4
1	t	h	i	s
2	w	a	t	s
3	o	a	h	g
4	f	g	d	t