Handbook of ALGORITHMS and DATA STRUCTURES

G. H. Gonnet

73·87221073
G639

Handbook of ALGORITHMS and DATA STRUCTURES

G. H. Gonnet

(c3

ADDISON-WESLEY PUBLISHING COMPANY

London · Reading, Massachusetts · Menlo Park, California · Amsterdam
Don Mills, Ontario · Manila · Singapore · Sydney · Tokyo

DS7/10

Set by the author on an APS-5 photosetter using output generated by troff, the UNIX text-processing system (UNIX is a trademark of Bell Laboratories)

Cover illustration produced by John Vince using PICASO, the computer graphics system at Middlesex Polytechnic

Printed in Finland by OTAVA. Member of Finnprint

OTABIND
PAT PEND

British Library Cataloguing in Publication Data
Gonnet, G. H.
Handbook of algorithms and data structures.
1. Electronic digital computers—Programming
2. Algorithms
I. Title
001.6'42 QA76.6

ISBN 0-201-14218-X

Library of Congress Cataloging in Publication Data
Gonnet, G. H. (Gaston H.)
Handbook of algorithms and data structures.
(International computer science series)
Bibliography: p.
Includes index.
1. Electronic digital computers—Programming.
2. Algorithms. 3. Data structures (Computer science)
I. Title. II. Series
QA76.6.G636 1984 001.64'2 83-27301

ISBN 0-201-14218-X

ABCDEF8987654

Preface

Computer Science has been, throughout its evolution, more an art than a science. My favourite example which illustrates this point is to compare a major software project (like the writing of a compiler) with any other major project (like the construction of the CN tower in Toronto). It would be absolutely unthinkable to let the tower fall down a few times while its design was being debugged: even worse would be to open it to the public before discovering some other fatal flaw. Yet this mode of operation is being used everyday by almost everybody in software production.

Presently it is very difficult to "stand on your predecessor's shoulders", most of the time we stand on our predecessor's toes, at best. This handbook was written with the intention of making available to the computer scientist, instructor or programmer the wealth of information which the field has generated in the last 20 years.

Most of the results are extracted from the given references. In some cases the author has completed or generalized some of these results. Accuracy is certainly one of our goals, and consequently the author will cheerfully pay $2.00 for each **first** report of any type of error appearing in this handbook.

Many people helped me directly or indirectly to complete this project. Firstly I owe my family hundreds of hours of attention. All my students and colleagues had some impact. In particular I would like to thank M. C. Momard, N. Ziviani, J. I. Munro, P. A. Lanson, D. Rotem and D. Wood. Very special thanks go to F. W. Tompa who is also the coauthor of chapter 2. The source material for this chapter appears in a joint paper in the November 1983 *Communications of the ACM*.

Montevideo
December 1983

G. H. Gonnet

Contents

Chapter 1: INTRODUCTION

This handbook is intended to contain most of the information available on algorithms and their data structures; thus it is designed to serve a wide spectrum of users, from the programmer who wants to code efficiently to the student or researcher who needs quick information.

The main emphasis is placed on algorithms. For these we present their description, code in one or more languages, theoretical results and extensive lists of references.

1.1 Structure of the chapters

The handbook is organized by topics. The second chapter offers a formalization of the description of algorithms and data structures, chapters 3 to 6 discuss searching, sorting, selection and arithmetic algorithms respectively. Appendix I describes some probability distributions encountered in data processing; appendix II contains a collection of asymptotic formulas related to the analysis of algorithms; appendix III contains the main list of references and appendix IV contains alternate code for some algorithms.

The chapters describing algorithms are divided in sections and sub-sections as needed. Each algorithm is described in its own sub-section, and all have roughly the same format, though we may make slight deviations or omissions when information is unavailable or trivial. The general format includes:

(1) Definition and explanation of the algorithm and its classification according to the basic operations described in chapter 2.

(2) Theoretical results on the algorithm's complexity. We are mainly interested in measurements which indicate an algorithm's running time and its space requirements. Useful quantities to measure for this information include the number of comparisons, data accesses, assignments, or exchanges an algorithm might make. When looking at space requirements, we might consider the number of words, records, or pointers involved in an implementation. Time complexity covers a much broader range of measurements. For example, in our examination of searching algorithms, we might be able to attach

8750103

meaningful interpretations to most of the combinations of the

$$\left\{\begin{array}{c}\text{average}\\ \text{variance}\\ \text{minimum}\\ \text{worst-case}\\ \text{average w.c.}\end{array}\right\}\text{ number of }\left\{\begin{array}{c}\text{comparisons}\\ \text{accesses}\\ \text{assignments}\\ \text{exchanges}\\ \text{function calls}\end{array}\right\}\text{ when we }\left\{\begin{array}{c}\text{query}\\ \text{add a record into}\\ \text{delete a record from}\\ \text{modify a record of}\\ \text{reorganize}\\ \text{build}\\ \text{read sequentially}\end{array}\right\}$$

the structure. Other theoretical results may also be presented, such as enumerations, generating functions, or behaviour of the algorithm when the data elements are distributed according to special distributions.

(3) The Algorithm. We have selected Pascal and C to describe the algorithms. Algorithms that may be used in practice are described in one or both of these languages. For algorithms which are only of theoretical interest, we do not provide their code. Algorithms which are coded both in Pascal and in C will have one code with the algorithm and the other in appendix IV.

(4) Recommendations. Following the algorithm description we give several hints and tips on how to use the algorithm. We point out pitfalls to avoid in coding, suggest when to use the algorithm and when not to, tell when to expect best and worst performances, and provide a variety of other comments.

(5) Tables. Whenever possible, we present tables which show exact values of complexity measures in selected cases. These are intended to give a feeling for how the algorithm behaves. When precise theoretical results are not available we give simulation results, generally in the form $xxx \pm yy$ where the value yy is chosen so that the resulting interval has a confidence level of 95%. In other words, the actual value of the complexity measure falls out of the given interval only once every twenty simulations.

(6) Differences between internal and external storage. Some algorithms may perform better for internal storage than external, or vice versa. When this is true, we will give recommendations for applications in the two different cases. Since most of our analysis up to this point implicitly assumes that internal memory is used, in this section we will look more closely at the external case (if appropriate). We analyze the algorithm's behaviour when working with external storage, and discuss any significant practical considerations in using the algorithm externally.

(7) With the description of each algorithm we include a list of relevant references. General references, surveys, or tutorials are collected at the end of chapters or sections. The second appendix contains an alphabetical list of all references with cross-references to the relevant algorithms.

1.2 Naming of variables

The naming of variables throughout this handbook is a compromise between uniformity of notation and accepted terminology in the specific areas.

Except for very few exceptions, explicitly noted, we use:

n for the number of objects or elements or components in a structure;

m for the size of a structure;

b for bucket sizes, or maximum number of elements in a physical block;

d for the digital cardinality or size of the alphabet.

The complexity measures are also named uniformly throughout the handbook. Complexity measures are named X_n^Z and should be read as "the number of X's performed while doing Z onto a structure of size n".

Typical values for X are:

A: accesses, probes or node inspections;

C: comparisons or node inspections;

E: external accesses;

h: height of a recursive structure (typically a tree);

I: iterations (or number of function calls);

L: length (of path or longest probe sequence);

M: moves or assignments (usually related to record or key movements).

Typical values for Z are:

null (no superscript): successful search (or default operation, when there is only one possibility);

C: construction (building) of structure;

D: deletion of an element;

E: extraction of an element (mostly for priority queues);

I: insertion of a new element;

M: merging of structures;

Opt: optimal construction or optimal structure (the operation is usually implicit);

MM: minimax, or minimum number of X's in the worst case: this is usually used to give upper and lower bounds on the complexity of a problem.

Note that X_n^I means number of operations done to insert an element into a structure of size n or to insert the $n+1^{st}$ element.

Although these measures are random variables (as these depend on the particular structure on which they are measured), we will make exceptions for C_n and C_n' which most of the literature considers to be expected values.

1.3 Probabilities

The probability of a given event is denoted by $Pr\{event\}$. Random variables follow the convention described in the preceding section. The expected value of a random variable X is written $E[X]$ and its variance is $\sigma^2(X)$. In particular, for discrete variables X

$$E[X] = \sum_i i \, Pr\{X=i\}$$

$$\sigma^2(X) = \sum_i i^2 Pr\{X=i\} - E[X]^2 = E[X^2] - E[X]^2$$

We will always make explicit the probability universe on which expected values are computed. This is ambiguous in some cases, and is a ubiquitous problem with expected values.

To illustrate the problem without trying to confuse the reader, suppose that we fill a hashing table with keys and then we want to know about the average number of accesses to retrieve one of the keys. We have two potential probability universes: the key selected for retrieval (the one inserted first, the one inserted second, etc.) and the actual values of the keys, or their probing sequence. We can compute expected values with respect to the first, the second, or both universes. In simpler terms, we can find the expected value of any key for a given file, or the expected value of a given key for any file, or the expected value of any key for any file.

Unless otherwise stated, (i) the distribution of our elements is always random independent uniform $U(0,1)$; (ii) the selection of a given element is uniform discrete between all possible elements; (iii) expected values which relate to multiple universes are computed with respect to all universes. In terms of the above example, we will compute expected values with respect to randomly selected variables drawn from a uniform $U(0,1)$ distribution.

1.4 Asymptotic notation

Most of the complexity measures in this handbook are asymptotic in the size of the problem. The asymptotic notation we will use is fairly standard and is given below:

$$f(n) = O(g(n))$$

implies that there exists k and n_0 such that $|f(n)| < kg(n)$ for $n > n_0$

$$f(n) = o(g(n)) \longrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \Theta(g(n))$$

implies that there exists k_1, k_2, $(k_1 \times k_2 > 0)$ and n_0 such that $k_1 g(n) < f(n) < k_2 g(n)$ for $n > n_0$, or equivalently that $f(n) = O(f(n))$ and $g(n) = O(f(n))$.

$$f(n) = \Omega(g(n)) \longrightarrow g(n) = O(f(n))$$

$$f(n) = \omega(g(n)) \longrightarrow g(n) = o(f(n))$$

$$f(n) \approx g(n) \longrightarrow f(n) - g(n) = o(g(n))$$

We will freely use arithmetic operations with the order notation, e.g.

$$f(n) = h(n) + O(g(n))$$

means

$$f(n) - h(n) = O(g(n))$$

Whenever we write $f(n) = O(g(n))$ it is with the understanding that we know of no better asymptotic bound, i.e. we know of no $h(n) = o(g(n))$ such that $f(n) = O(h(n))$.

1.5 About the programming languages

We use two languages to code our algorithms: Pascal and C. After writing many algorithms we still find situations for which these languages do not present a very "clean" or understandable code. Therefore, whenever possible, we use the language which presents the shortest and most readable code. We intentionally allow our Pascal and C style of coding to resemble each other.

A minimal number of Pascal programmes contain **goto** statements. These statements were used in place of the equivalent C statements **return** and **break**, and are correspondingly so commented. Indeed we view their absence from Pascal as a shortcoming of the language. Another irritant in coding some algorithms in Pascal is the lack of order in the evaluation of logical expressions. This is unfortunate since such a feature makes algorithms easier to understand. The typical stumbling block is

```
while ( p ≠ nil ) and ( key ≠ p↑.key ) do ....
```

Such a statement works in C if we use the sequential and operator (&&), but for Pascal we have to use instead:

```
while p ≠ nil do begin
        if key = p↑.k then goto 999 {*** break ***};
         ....
999:
```

Other minor objections are: the inability to compute addresses of non-heap objects in Pascal (which makes treatment of lists more difficult); the lack of a **with** statement in C and the lack of **var** parameters in C. (Although this is technically possible to overcome, it obscures the algorithms.)

Our Pascal code conforms, as fully as possible, to the language described in *Pascal User Manual and Report* by K. Jensen and N. Wirth. The C code conforms to the language described in *The C Programming Language* by B.W. Kernighan and D.M. Ritchie.

1.6 On the code for the algorithms

Except for very few algorithms which are obviously written in pseudo-code, the algorithms in this handbook were run and tested under two different compilers. Actually the same text which is printed is used for compiling, for testing, for running simulations and for obtaining timings. This was done in an attempt to eliminate (or at least drastically reduce!) errors.

Each family of algorithms has a "tester set" which not only checks for correct behaviour of the algorithm, but also checks proper handling of limiting conditions (will a sorting routine sort a null file? one with one element? one with all equal keys? etc.).

In most cases the algorithms are described as a function or a procedure or a small set of functions or procedures. In a few cases, for very simple algorithms, the code is described as in-line code, which could be encapsulated in a procedure or could be inserted into some other piece of code.

Some algorithms, most notably the searching algorithms, are building blocks or components of other algorithms or programmes. Some standard actions should not be specified for the algorithm itself, but rather will be specified once that the algorithm is "composed" with other parts (chapter 2 defines composition in more detail). A typical example of a standard action is an error condition. The algorithms coded for this handbook use always the same names for these standard actions.

Error detection of an unexpected condition during execution. Whenever *Error* is encountered it can be substituted by any block of statements. For example our testers print an appropriate message.

found(record) function call that is executed upon completion of a successful search. Its argument is a record or a pointer to a record which contains the searched key.

notfound(key) function called upon an unsuccessful search. Its argument is the key which was not found.

A special effort has been made to avoid duplication of these standard actions for identical conditions. This makes it easier to substitute blocks of code for these.

1.7 Complexity measures and real timings

For some families of algorithms we include a comparison of real timings. These timings are to be interpreted with caution as they reflect only one sample point in the many dimensions of hardwares, compilers, operating systems, etc. Yet we have equally powerful reasons to present at least one set of real complexities.

The main reasons for including real timing comparisons are:

(i) these take into account the actual cost of operations,

(ii) these take into account hidden costs, like storage allocation, indexing, etc.

The main objections, or the factors which may invalidate these real timing tables, are:

(i) the results are compiler dependent: although the same compiler is used for each language, a compiler may favour some construct over others;

(ii) the results are hardware dependent,

(iii) in some cases, when large amounts of memory are used, the timings may be load dependent.

The timings were done on a VAX 11/780 running the Berkeley Unix 4.1 operating system. Both C and Pascal compilers were run with the optimizer, or object code improver, to obtain the best implementation for the algorithms.

There were no attempts made to compare timings across languages. All the timing results are computed relative to the fastest algorithm. To avoid the incidence of start up costs, loading, etc. the tests were run on problems of significant size. Under these circumstances, some $O(n^2)$ algorithms appear to perform very poorly.

Chapter 2: BASIC CONCEPTS

2.1 Data structure description

The formal description of data structure implementations is similar to the formal description of programming languages. In defining a programming language, one typically begins by presenting a syntax for valid programmes in the form of a grammar and then sets further validity restrictions (e.g., usage rules for symbolic names) which give constraints that are not captured by the grammar. Similarly, a valid data structure implementation will be one that satisfies a syntactic grammar and also obeys certain constraints. For example, for a particular data structure to be a valid weight balanced binary tree, it must satisfy the grammatical rules for binary trees and it must also satisfy a specific balancing constraint.

2.1.1 Grammar for data objects

A sequence of real numbers can be defined by the BNF production

<S> ::= [real , <S>] | nil

Thus a sequence of reals can have the form nil, [real,nil], [real,[real,nil]], and so on. Similarly, sequences of integers, characters, strings, boolean constants, etc. could be defined. However, this would result in a bulky collection of production rules which are all very much alike. One might first try to eliminate this repetitiveness by defining

<S> ::= [<D> , <S>] | nil

where <D> is given as the list of data types

<D> ::= real | int | bool | string | char

However, this pair of productions generates unwanted sequences such as

[real,[int,nil]]

as well as the homogeneous sequences desired.

To overcome this problem, the syntax of a data object class can be defined using a W-grammar (also called a two-level or van Wijngaarden grammar). Actually the full capabilities of W-grammars will not be utilized; rather the syntax will be defined using the equivalent of standard BNF productions together with the uniform replacement rule as described below.

A W-grammar generates a language in two steps (levels). In the first step, a collection of generalized rules is used to create more specific production rules. In the second step, the production rules generated in the first step are used to define the actual data structures.

First, the problem of listing repetitive production rules is solved by starting out with generalized rule-forms known as *hyperrules*, rather than the rules themselves. The generalized form of a sequence S is given by the hyperrule

$$\mathbf{s-D : [D, s-D] ; nil}$$

The set of possible substitutions for **D** are now defined in a *metaproduction*, as distinguished from a conventional BNF-type production. For example, if **D** is given as

$$\mathbf{D :: real; int; bool; string; char;} \cdots$$

a sequence of real numbers is defined in two steps as follows. The first step consists of choosing a value to substitute for **D** from the list of possibilities given by the appropriate metaproduction; in this instance, **D** $\rightarrow$ **real.** Next invoke the uniform replacement rule to substitute the string **real** for **D** everywhere it appears in the hyperrule that defines **s−D**. This substitution gives

$$\mathbf{s-real : [real , s-real] ; nil}$$

Thus the joint use of the metaproduction and the hyperrule generates an ordinary BNF-like production defining real sequences. The same two statements can generate a production rule for sequences of any other valid data type (integer, character, etc.).

Figures 2.1 and 2.2 contain a W-grammar which will generate many conventional data objects. As further examples of the use of this grammar, consider the generation of a binary tree of real numbers. With **D→real** and **LEAF→nil**, HR[3] generates the production rule

$$\mathbf{bt-real-nil : [\ real , bt-real-nil , bt-real-nil\ ] ; nil}$$

Since **bt−real−nil** is one of the legitimate values for **D** according to M[1] let **D→bt−real−nil** from which HR[1] indicates that such a binary tree is a legitimate data structure.

Secondly consider the specification for a hash table to be used with direct chaining. The production

$$\mathbf{s-(string,int) : [\ (string,int) , s-(string,int)\ ] ; nil}$$

and M[1] yield

$$\mathbf{D} \rightarrow \{\mathbf{s-(string,int)}\}_0^{96}$$

Thus HR[1] will yield a production for an array of sequences of string/integer pairs usable, for example, to record NAME/AGE entries using hashing.

Finally consider a production rule for structures to contain B-trees of strings using HR[4] and the appropriate metaproductions to yield

$$\mathbf{mt-10-string-nil : [int,\{string\}_1^{10},\{mt-10-string-nil\}_0^{10}] ; nil}$$

Metaproductions:

M[1]	D ::	**real;int;bool;string;char;...;**	# atomic data types
		{ D }$^{N}_{N}$;	# array
		REC ; (REC) ;	# record
		[D] ;	# reference
		s−D;	# sequence
		gt−D−LEAF;	# general tree
		DICT;	# dictionary structures
		... .	# other structure classes E.g. graphs, sets, priority queues.
M[2]	DICT ::	**{KEY}$^{N}_{N}$; s−KEY;**	# sequential search
		bt−KEY−LEAF;	# binary tree
		mt−N−KEY−LEAF;	# multi-way tree
		tr−N−KEY.	# trie
M[3]	REC ::	**D; D, REC.**	# record definition
M[4]	LEAF ::	**nil; D.**	
M[5]	N ::	**DIGIT; DIGIT N.**	
M[6]	DIGIT ::	**0;1;2;3;4;5;6;7;8;9.**	
M[7]	KEY ::	**real;int;string;char;(KEY,REC).**	# search key

Figure 2.1: Metaproductions for data objects

Hyperrules:

HR[1]	**data structure :**	**D.**
HR[2]	**s−D :**	**[D,s−D] ; nil.**
HR[3]	**bt−D−LEAF :**	**[D,bt−D−LEAF,bt−D−LEAF] ; LEAF.**
HR[4]	**mt−N−D−LEAF :**	**[int,{D}$^{N}_{1}$,{mt−N−D−LEAF}$^{N}_{0}$] ; LEAF.**
HR[5]	**gt−D−LEAF :**	**[D,s−gt−D−LEAF] ; LEAF.**
HR[6]	**tr−N−D:**	**[{ tr−N−D }$^{N}_{1}$] ; [D] ; nil.**

Figure 2.2: Hyperrules for data objects

In this multitree, each node contains ten keys and has eleven descendants. Certain restrictions on B-trees, however, are not included in this description (that the number of actual keys is to be stored in the **int** field in each node, that this number must be between 5 and 10, that the actual keys will be stored contiguously in the keys-array starting at position 1, etc.); these will