

**ALGORITHMS +
DATA STRUCTURES =
PROGRAMS**

NIKLAUS WIRTH

*Eidgenossische Technische Hochschule
Zurich, Switzerland*

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, N.J.

Library of Congress Cataloging in Publication Data

WIRTH, NIKLAUS.

Algorithms + data structures = programs.

Bibliography: p.

Includes index.

1. Electronic digital computers—Programming.
2. Data structures (Computer science)
3. Algorithms.

I. Title.

QA76.6.W56 001.6'42 75-11599

ISBN 0-13-022418-9

To Nani

© 1976
by PRENTICE-HALL, INC.
Englewood Cliffs, New Jersey

All rights reserved. No part of this
book may be reproduced in any form
or by any means without permission
in writing from the publisher.

10 9 8 7 6

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA, PTY., LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA (PTE.) LTD., *Singapore*

Structure	Declaration	Selector	Access to Components by	Component Types	Cardinality
Array	$a: \text{array}[I] \text{ of } T_0$	$a[i] \quad (i \in I)$	Selector with computable index i	All identical (T_0)	$\text{card}(T_0)^{\text{card}(I)}$
Record	$r: \text{record } s_1: T_1; \\ s_2: T_2; \\ \dots \\ s_n: T_n \\ \text{end}$	$r.s \quad (s \in \{s_1, \dots, s_n\})$	Selector with declared component name s	May individually differ	$\prod_{i=1}^n \text{card}(T_i)$
Set	$s: \text{set of } T_0$	None	Membership test with relational operator in	All identical (and of scalar type T_0)	$2^{\text{card}(T_0)}$

Table 1.3 Fundamental data structures.

```

var s,t: course;
    trialset: selection;
begin s := 1;
    while  $\neg(s \text{ in remaining})$  do s := s + 1;
    session := [s]; trialset := remaining;
    for t := 1 to N do
        if t in trialset then
            begin if conflict[t] * session = [t] then
                session := session + [t];
            end
        end
    end

```

Evidently, this solution for selecting “suitable” sessions is not necessarily optimal in any special case the number of sessions may be as large as that which would be possible if simultaneous scheduling were feasible.

1.10. REPRESENTATION OF ARRAY, RECORD, AND SET STRUCTURES

The essence of the use of abstractions in programming may be conceived, understood, and verified on the basis of the abstractions and that it is not necessary to have a detailed edge about the ways in which the abstractions are represented in a particular computer. Nevertheless, it is important for the programmer to have an understanding of widely used programming concepts representing the basic concepts of programming abstract data structures. It is helpful in the sense that it enables the programmer to make sensible decisions about programming in the light not only of the abstract properties of structures but also of their realizations on actual computers, taking into account their capabilities and limitations.

The problem of data representation is that of representing a structure into a computer store. Computer stores are organized as arrays of individual storage cells called words. The words are called addresses.

```
var store: array[address] of word;
```

The cardinalities of the types *address* and *word* are related to the size of the word. A particular problem is the great variability in the size of the word. Its logarithm is called the *wordsize*, the number of bits that a storage cell consists of.


```

    if  $p1 \uparrow.lh$  then
    begin {RL}  $p2 := p1 \uparrow.left$ ;  $p1 \uparrow.lh := false$ ;
       $p1 \uparrow.left := p2 \uparrow.right$ ;  $p2 \uparrow.right := p1$ ;
       $p \uparrow.right := p2 \uparrow.left$ ;  $p2 \uparrow.left := p$ ;  $p := p2$ 
    end
  end else
  begin  $h := h - 1$ ; if  $h \neq 0$  then  $p \uparrow.rh := true$ 
  end
end else
begin  $p \uparrow.count := p \uparrow.count + 1$ ;  $h := 0$ 
end
end {search}

```

Note that the actions to be taken for node re-arrangement very strongly resemble those developed in the balanced tree search algorithm (4.63). From (4.87) it is evident that all four cases can be implemented by simple pointer rotations: single rotations in the *LL* and *RR* cases, double rotations in the *LR* and *RL* cases. In fact, procedure (4.87) appears slightly simpler than (4.63). Clearly, the hedge-tree scheme emerges as an alternative to the AVL-balance criterion. A performance comparison is therefore both possible and desirable.

We refrain from involved mathematical analysis and concentrate on some basic differences. It can be proven that the *AVL-balanced trees are a subset of the hedge-trees*. Hence, the class of the latter is larger. It follows that their path length is on the average larger than in the AVL case. Note in this connection the “worst-case” tree (4) in Fig. 4.53. On the other hand, node re-arrangement will be called for less frequently. The balanced tree will therefore be preferred in those applications in which key retrievals are much more frequent than insertions (or deletions); if this quotient is moderate, the hedge-tree scheme may be preferred.

It is very difficult to say where the borderline lies. It strongly depends not only on the quotient between the frequencies of retrieval and structural change, but also on the characteristics of an implementation. This is particularly the case if the node records have a densely packed representation and consequently access to fields involves part word selection. Boolean fields (*lh*, *rh* in the case of hedge-trees) may be handled more efficiently on many implementations than three-valued fields (*bal* in the case of balanced trees).

4.6. KEY TRANSFORMATIONS (HASHING)

The general problem addressed in the last section and used to develop solutions demonstrating dynamic data allocation techniques is the following:

Given a set S of items characterized by a key value upon which an ordering relation is defined, how is S to be organized so that

retrieval of an item with a given key k involve a minimum number of comparisons.

Clearly, in a computer store each item is ultimately associated with a storage address a . Hence, the stated problem is essentially that of finding an appropriate mapping H of keys (K) into addresses (A):

$$H: K \rightarrow A$$

In Sect. 4.5 this mapping was implemented in the form of an array and tree search algorithms based on different underlying principles. Here we present yet another approach that is both simpler and more efficient in many cases. The fact that it also has some disadvantages is discussed subsequently.

The data organization used in this technique is called *key transformation* and is therefore a mapping transforming keys into array indices. The reason for the term *key transformation* that is generally used is that it should be noted that we shall not need to rely on the traditional search procedures because the array is one of the fundamental data structures. This paragraph is thus somewhat misplaced under the heading of dynamic information structures, but since it is often used where tree structures are comparable competitors, the appropriate place for its presentation.

The fundamental difficulty in using a key transformation is that the set of possible key values is very much larger than the set of possible addresses (array indices). A typical example is the use of words with, say, up to 10 letters as keys for the identification of a set of, say, up to a thousand persons. Hence, there are 10^{10} possible keys which are to be mapped onto 10^3 possible indices. This is obviously a many-to-one function. Given a key k , the (search) operation is to compute its associated index h in the array—evidently necessary—step is to verify whether or not the key k is indeed identified by h in the array (table) T , i.e., $T[H(k)].key = k$. We are immediately confronted with the following questions:

1. What kind of function H should be used?
2. How do we cope with the situation that H does not always identify the desired item?

The answer to question 2 is that some method must be used to find the native location, say index h' , and, if this is still not the desired one, yet a third index h'' , and so on. The case in which the desired one is at the identified location is called a *collision*. Handling collisions is termed *collision handling*. We shall discuss the choice of a transformation function and collision handling in the next section.

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts



Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research



With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips



Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.