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

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*

To Nani

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

Table 1.3 Fundamental data structures.


```
var s,t: course;
    trialset: selection;
begin s := 1;
    while ¬(s in remaining) do s := s-
    session := [s]; trialset := remainin
    for t := 1 to N do
        if t in trialset then
        begin if conflict[t] * session = [
            session := session + [t]
        end
end
```

Evidently, this solution for selecting "suitable" ses
timetable which is necessarily optimal in any spec
cases the number of sessions may be as large as that
taneous scheduling were feasible.

1.10. REPRESENTATION OF ARRAY, RECORD, AND SET STRUCTURES

The essence of the use of abstractions in progra
may be conceived, understood, and verified on the b
the abstractions and that it is not necessary to have
edge about the ways in which the abstractions ar
sented in a particular computer. Nevertheless, it
programmer to have an understanding of widely u
senting the basic concepts of programming abstra
mental data structures. It is helpful in the sense
programmer to make sensible decisions about pr
the light not only of the abstract properties of st
realizations on actual computers, taking into accou
capabilities and limitations.

The problem of data representation is that
structure into a computer store. Computer stores
tion—arrays of individual storage cells called w
words are called addresses.

var *store*: **array**[*address*] **of**

The cardinalities of the types *address* and *word*
to another. A particular problem is the great va
of the word. Its logarithm is called the *wordsize*, b
bits that a storage cell consists of.

1.10.1. Representation of Arrays

A representation of an array structure is a mapping of the (abstract) array with components of type T onto the store which is an array with components of type *word*.

The array should be mapped in such a way that the computation of addresses of array components is as simple (and therefore efficient) as possible. The address or store index i of the jth array component is computed by the linear mapping function

$$i = i_0 + j * s \qquad (1.32)$$

where i_0 is the address of the first component, and s is the number of words that a component "occupies." Since the word is by definition the smallest individually accessible unit of store, it is evidently highly desirable that s be a whole number, the simplest case being $s = 1$. If s is not a whole number (and this is the normal case), then s is usually rounded up to the next larger integer $\lceil s \rceil$. Each array component then occupies $\lceil s \rceil$ words, whereby $\lceil s \rceil - s$ words are left unused (see Figs. 1.5 and 1.6). Rounding up of the number of

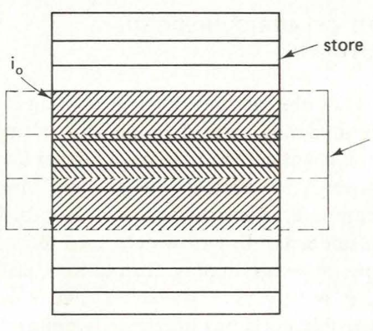

Fig. 1.5 Mapping an array onto a store.

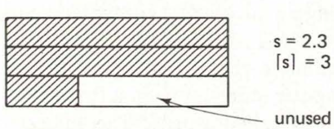

Fig. 1.6 Padded representation of a record.

words needed to the next whole number is called *padding*. The storage utilization factor u is the quotient of the minimal amounts of storage needed to represent a structure and of the amounts actually used:

$$u = \frac{s}{s'} = \frac{s}{\lceil s \rceil} \qquad (1.33)$$

Since an implementor will have to aim for a storage utilization as close to

1 as possible, and since accessing parts of wor
relatively inefficient process, he will have to comp
considerations to be made:

1. Padding will decrease storage utilization.
2. Omission of padding may necessitate inefficie
3. Partial word access may cause the code (com and therefore to counteract the gain obtained

In fact, considerations 2 and 3 are usually so do
always use padding automatically. We notice tha
always be $u > 0.5$, if $s > 0.5$. However, if $s \leq$
may be significantly increased by putting more t
into each word. This technique is called *packing*. I
into a word, the utilization factor is (see Fig. 1.7)

$$u = \frac{n \cdot s}{\lceil n \cdot s \rceil}$$

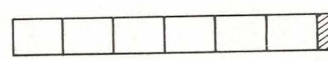

Fig. 1.7 Packing six components into o

Access to the ith component of a packed arra
of the word address j in which the desired compo
the computation of the respective component pos

$$j = i \textbf{ div } n$$
$$k = i \textbf{ mod } n = i - j*n$$

In most programming languages the progra
over the representation of the abstract data stru
be possible to indicate the desirability of packin
which more than one component would fit into
gain of storage economy by a factor of 2 and m
introduce the convention to indicate the desirabi
the symbol **array** (or **record**) in the declaration by

EXAMPLE

type *alfa* = **packed array** [1 . .

This feature is particularly valuable on compu
relatively convenient accessibility of partial fiel
property of this prefix is that it does in no way c
rectness) of a program. This means that the cho
sentation can be easily indicated with the implied
of the program remains unaffected.

```
            if p1↑.lh then
            begin {RL} p2 := p1↑.left; p1↑.lh := false;
                p1↑.left := p2↑.right; p2↑.right := p1;
                p↑.right := p2↑.left; p2↑.left := p; p := p2
            end
        end else
        begin h := h−1; if h ≠ 0 then p↑.rh := true
        end
    end else
    begin p↑.count := p↑.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
> possible.

Clearly, in a computer store each item is ultimately
a storage address *a*. Hence, the stated problem is esse
appropriate mapping *H* of keys (*K*) into addresses (

$$H\colon K \longrightarrow A$$

In Sect. 4.5 this mapping was implemented in
and tree search algorithms based on different underl
Here we present yet another approach that is ba
efficient in many cases. The fact that it also has son
discussed subsequently.

The data organization used in this technique is
is therefore a mapping transforming keys into arr
reason for the term *key transformation* that is generall
It should be noted that we shall not need to rely on
procedures because the array is one of the fundam
This paragraph is thus somewhat misplaced under
dynamic information structures, but since it is ofter
where tree structures are comparable competitors, th
priate place for its presentation.

The fundamental difficulty in using a key transf
of possible key values is very much larger than th
addresses (array indices). A typical example is the u
with, say, up to 10 letters as keys for the identifica
set of, say, up to a thousand persons. Hence, there
which are to be mapped onto 10^3 possible indices. Th
obviously a many-to-one function. Given a key *k*, th
(search) operation is to compute its associated index *h*
—evidently necessary—step is to verify whether or n
k is indeed identified by *h* in the array (table) *T*,
$T[H(k)]$.key $= k$. We are immediately confronted w

1. What kind of function *H* should be used?
2. How do we cope with the situation that *H* does
   the desired item?

The answer to question 2 is that some method must b
native location, say index *h*′, and, if this is still not th
item, yet a third index *h*″, and so on. The case in wh
desired one is at the identified location is called a *co*
erating alternative indices is termed *collision handli*
shall discuss the choice of a transformation function

DOCKET ALARM

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.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.