

Volume 1 / **Fundamental Algorithms**

H *Stanford University*

**THE ART OF
COMPUTER PROGRAMMING
SECOND EDITION**

ISHING COMPANY

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

This book is in the
**ADDISON-WESLEY SERIES IN
COMPUTER SCIENCE AND INFORMATION PROCESSING**

RICHARD S. VARGA and MICHAEL A. HARRISON, Editors

COPYRIGHT © 1973, 1968 BY ADDISON-WESLEY PUBLISHING COMPANY, INC. ALL RIGHTS RESERVED. NO PART OF THIS PUBLICATION MAY BE REPRODUCED, STORED IN A RETRIEVAL SYSTEM, OR TRANSMITTED, IN ANY FORM OR BY ANY MEANS, ELECTRONIC, MECHANICAL, PHOTOCOPYING, RECORDING, OR OTHERWISE, WITHOUT THE PRIOR WRITTEN PERMISSION OF THE PUBLISHER. PRINTED IN THE UNITED STATES OF AMERICA. PUBLISHED SIMULTANEOUSLY IN CANADA. LIBRARY OF CONGRESS CATALOG CARD NO. 73-1830.

ISBN 0-201-03809-9
CDEFGHIJ-MA-79876

He
publis
receipt
Now we ca
if you

The process of preparing not only because it can because it can be an aes This book is the first ve signed to train the reade

The following chapti puter programming; th perience. The prerequi time and practice before puter. The reader shou

- a) Some idea of how a the electronics, rath machine's memory a language will be hel
- b) An ability to put th computer can "unde they have not yet le no more and no les first tries to use a c
- c) Some knowledge of looping (performing and the use of index
- d) A little knowledge o "bits," "floating po are given brief defin

* or she. Masculine pronoun
Occasional chauvinistic com

2.2. LINEAR LISTS

2.2.1. Stacks, Queues, and Deques

Usually there is much more structural information present in the data than we actually want to represent directly in a computer. In each "playing card" node of the preceding section, for example, we have a NEXT field to specify what card is beneath it in the pile, but there is no direct way to find what card, if any, is *above* a given card, or to find which pile of playing cards which has been totally suppressed from the computer representation: the details of the design on the back of the cards, the relation of the cards to other objects in the room where the game is being played, the molecules which compose the cards, etc. It is conceivable that such structural information would be relevant in certain computer applications, but obviously we never want to store *all* of the structure present in every situation. Indeed, for most card-playing situations we would not need all of the facts retained in our earlier example; thus the TAG field, which tells whether a card is face up or face down, will often be unnecessary.

It is therefore clear that we must decide in each case how much structure to represent in our tables, and how accessible to make each piece of information. To make this decision, we need to know what operations are to be performed on the data. For each problem considered in this chapter, *we therefore consider not only the data structure but also the class of operations to be done on the data*; the design of computer representations depends on the desired function of the data as well as on its intrinsic properties. Such an emphasis on "function" as well as "form" is basic to design problems in general.

In order to illustrate this point further, let us consider a simple example which arises in computer hardware design. A computer memory is often classified as a "random access memory," i.e., MIX's main memory; or as a "read only memory," i.e., one which is to contain essentially constant information; or a "secondary bulk memory," like MIX's disk units, which cannot be accessed at high speed although large quantities of information can be stored; or an "associative memory," more properly called a "content-addressed memory," i.e., one for which information is addressed by values stored with it rather than by its location; and so on. Note that the intended function of each kind of memory is so important that it enters into the name of the particular memory type; all of these devices are "memory" units, but the purposes to which they are put profoundly influence their design and their cost.

A *linear list* is a set of $n \geq 0$ nodes $x[1], x[2], \dots, x[n]$ whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes: the facts that, if $n > 0$, $x[1]$ is the first node; when $1 < k < n$, the k th node $x[k]$ is preceded by $x[k-1]$ and followed by $x[k+1]$; and $x[n]$ is the last node.

The operations we might want to perform on linear lists include, for example, the following.

- i) Gain access to the contents of its
- ii) Insert a new node
- iii) Delete the k th node
- iv) Combine two or more
- v) Split a linear list into
- vi) Make a copy of a
- vii) Determine the number
- viii) Sort the nodes of the nodes.
- ix) Search the list for some field.

In operations (i), (ii), and (iii) the importance since the first is more than a general element is. This chapter, since these topics

A computer application of their full generality, so we are depending on the class of operations is difficult to design a structure all of these operations are performed the k th node of a long list at the same time we are inserting a new node before we distinguish between operations to be performed on a list are distinguished by their

Linear lists in which information is always at the first or the last end are given them special names:

A *stack* is a linear list in which all accesses are made at the same end.

A *queue* is a linear list in which all deletions (and insertions) are made at the same end.

A *deque* ("double-ended queue") is a linear list in which deletions (and insertions) are made at both ends.

A deque is therefore more common with a deck of cards. In some disciplines it is distinguished *output-restricted* and *input-restricted* insertions, respectively, are used to describe any kind of list structure cases identified above are

ion present in the data than we
er. In each "playing card" node
NEXT field to specify what card is
way to find what card, if any,
n card is in. Of course, there is
f playing cards which has been
ation: the details of the design
ds to other objects in the room
hich compose the cards, etc. It
t would be relevant in certain
ant to store *all* of the structure
rd-playing situations we would
mple; thus the TAG field, which
ll often be unnecessary.

ach case how much structure to
ake each piece of information.
erations are to be performed on
apter, *we therefore consider not
ions to be done on the data*; the
he desired function of the data
phasis on "function" as well as

us consider a simple example
outer memory is often classified
memory; or as a "read only
ly constant information; or a
which cannot be accessed at
on can be stored; or an "asso-
-addressed memory," i.e., one
red with it rather than by its
ion of each kind of memory is
particular memory type; all of
poses to which they are put

], . . . , X[n] whose structural
dimensional) relative positions
first node; when $1 < k < n$,
llowed by $X[k + 1]$; and $X[n]$

near lists include, for example,

- i) Gain access to the k th node of the list to examine and/or to change the contents of its fields.
- ii) Insert a new node just before the k th node.
- iii) Delete the k th node.
- iv) Combine two or more linear lists into a single list.
- v) Split a linear list into two or more lists.
- vi) Make a copy of a linear list.
- vii) Determine the number of nodes in a list.
- viii) Sort the nodes of the list into ascending order based on certain fields of the nodes.
- ix) Search the list for the occurrence of a node with a particular value in some field.

In operations (i), (ii), and (iii) the special cases $k = 1$ and $k = n$ are of principal importance since the first and last items of a linear list may be easier to get at than a general element is. We will not discuss operations (viii) and (ix) in this chapter, since these topics are the subjects of Chapters 5 and 6, respectively.

A computer application rarely calls for all nine of the above operations in their full generality, so we find there are many ways to represent linear lists depending on the class of operations which are to be done most frequently. It is difficult to design a single representation method for linear lists in which all of these operations are efficient; for example, the ability to gain access to the k th node of a long list for random k is comparatively hard to do if at the same time we are inserting and deleting items in the middle of the list. Therefore we distinguish between types of linear lists depending on the principal operations to be performed, just as we have noted that computer memories are distinguished by their intended applications.

Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are very frequently encountered, and we give them special names:

A *stack* is a linear list for which all insertions and deletions (and usually all accesses) are made at one end of the list.

A *queue* is a linear list for which all insertions are made at one end of the list; all deletions (and usually all accesses) are made at the other end.

A *deque* ("double-ended queue") is a linear list for which all insertions and deletions (and usually all accesses) are made at the ends of the list.

A deque is therefore more general than a stack or a queue; it has some properties in common with a deck of cards, and it is pronounced the same way. We also distinguish *output-restricted* or *input-restricted* dequeues, in which deletions or insertions, respectively, are allowed to take place at only one end.

In some disciplines the word "queue" has been used in a much broader sense to describe any kind of list that is subject to insertions and deletions; the special cases identified above are then called various "queuing disciplines." Only the

2.2.2. Sequential Allocation

The simplest and most natural way to keep a linear list inside a computer is to put the list items in sequential locations, one node after the other. We thus will have

$$\text{LOC}(X[j+1]) = \text{LOC}(X[j]) + c,$$

where c is the number of words per node. (Usually $c = 1$. When $c > 1$, it is sometimes more convenient to split a single list into c "parallel" lists, so that the k th word of node $X[j]$ is stored a fixed distance from the location of the first word of $X[j]$. We will continually assume, however, that adjacent groups of c words form a single node.) In general,

$$\text{LOC}(X[j]) = L_0 + cj, \quad (1)$$

where L_0 is a constant called the *base address*, the location of an artificially assumed node $X[0]$.

This technique for representing a linear list is so obvious and well-known that there seems to be no need to dwell on it at any length. But we will be seeing many other "more sophisticated" methods of representation later on in this chapter, and it is a good idea to examine the simple case first to see just how far we can go with it. It is important to understand the limitations as well as the power of the use of sequential allocation.

Sequential allocation is quite convenient for dealing with a *stack*. We simply have a variable T called the *stack pointer*. When the stack is empty, we let $T = 0$. To place a new element Y on top of the stack, we set

$$T \leftarrow T + 1; \quad X[T] \leftarrow Y. \quad (2)$$

And when the stack is not empty, we can set Y equal to the top node and delete that node by reversing the actions of (2):

$$Y \leftarrow X[T]; \quad T \leftarrow T - 1. \quad (3)$$

(Inside a computer it is usually most efficient to maintain the value cT instead of T , because of (1). Such modifications are easily made, so we will continue our discussion as though $c = 1$.)

The representation of a *queue* or a more general *deque* is a little trickier. An obvious solution is to keep two pointers, say F and R (for the front and rear of the queue), with $F = R = 0$ when the queue is empty. Then inserting an element at the rear of the queue would be

$$R \leftarrow R + 1; \quad X[R] \leftarrow Y. \quad (4)$$

Removing the front node (F points just below the front) would be

$$F \leftarrow F + 1; \quad Y \leftarrow X[F]; \quad \text{if } F = R, \text{ then set } F \leftarrow R \leftarrow 0. \quad (5)$$

But note what can happen: If R always stays ahead of F (so there is always at

least one node in the queue) and infinitum, and this is true (4), (5) is therefore to be used to R quite regularly (for example the queue).

To circumvent the problem aside M nodes $X[1], \dots, X[M]$. Then the above procedure

if $R = M$ then
if $F = M$ then

This circular queuing action is a discussion of input-output

The above discussion assumed nothing could go wrong with a queue, we assumed that there is a node onto a stack or queue. Clearly the method (6), (7) methods (2), (3), (4), (5) are within any given computer. The above actions must be revised that these restrictions are

$X \leftarrow Y$ (insert into stack)

$Y \leftarrow X$ (delete from stack)

$X \leftarrow Y$ (insert into queue)

$Y \leftarrow X$ (delete from queue)

Here we assume that $X[1]$ is the first node in the list; OVERFLOW and UNDERFLOW are the initial setting $F = R = 0$; use (6a) and (7a); we should

The reader is urged to consider the details of this simple queuing method.

The next question is, In the case of UNDERFLOW usually a meaningful correction to govern the flow of a pro-