Data Structures and Network Algorithms

44

ROBERT ENDRE TARJAN
Bell Laboratories
Murray Hill, New Jersey

CBMS-NSF
REGIONAL CONFERENCE SERIES
IN APPLIED MATHEMATICS

SPONSORED BY
CONFERENCE BOARD OF
THE MATHEMATICAL SCIENCES

SUPPORTED BY
NATIONAL SCIENCE
FOUNDATION

Data Structures and Network Algorithms

ROBERT ENDRE TARJAN
Bell Laboratories
Murray Hill, New Jersey

SOCIETY for INDUSTRIAL and APPLIED MATHEMATICS • 1983

PHILADELPHIA, PENNSYLVANIA 19103

Library of Congress Catalog Card Number: 83-61374.
ISBN: 0-89871-187-8

Preface

In the last fifteen years there has been an explosive growth in the field of combinatorial algorithms. Although much of the recent work is theoretical in nature, many newly discovered algorithms are quite practical. These algorithms depend not only on new results in combinatorics and especially in graph theory, but also on the development of new data structures and new techniques for analyzing algorithms. My purpose in this book is to reveal the interplay of these areas by explaining the most efficient known algorithms for a selection of combinatorial problems. The book covers four classical problems in network optimization, including a development of the data structures they use and an analysis of their running times. This material will be included in a more comprehensive two-volume work I am planning on data structures and graph algorithms.

My goal has been depth, precision and simplicity. I have tried to present the most advanced techniques now known in a way that makes them understandable and available for possible practical use. I hope to convey to the reader some appreciation of the depth and beauty of the field of graph algorithms, some knowledge of the best algorithms to solve the particular problems covered, and an understanding of how to implement these algorithms.

The book is based on lectures delivered at a CBMS Regional Conference at the Worcester Polytechnic Institute (WPI) in June, 1981. It also includes very recent unpublished work done jointly with Dan Sleator of Bell Laboratories. I would like to thank Paul Davis and the rest of the staff at WPI for their hard work in organizing and running the conference, all the participants for their interest and stimulation, and the National Science Foundation for financial support. My thanks also to Cindy Romeo and Marie Wenslau for the diligent and excellent job they did in preparing the manuscript, to Michael Garey for his penetrating criticism, and especially to Dan Sleator, with whom it has been a rare pleasure to work.

Contents

CHAPTER 1

Foundations

1.1. Introduction. In this book we shall examine efficient computer algorithms for four classical problems in network optimization. These algorithms combine results from two areas: data structures and algorithm analysis, and network optimization, which itself draws from operations research, computer science and graph theory. For the problems we consider, our aim is to provide an understanding of the most efficient known algorithms.

We shall assume some introductory knowledge of the areas we cover. There are several good books on data structures and algorithm analysis [1], [35], [36], [44], [49], [58] and several on graph algorithms and network optimization [8], [11], [21], [38], [39], [41], [50]; most of these touch on both topics. What we shall stress here is how the best algorithms arise from the interaction between these areas. Many presentations of network algorithms omit all but a superficial discussion of data structures, leaving a potential user of such algorithms with a nontrivial programming task. One of our goals is to present good algorithms in a way that makes them both easy to understand and easy to implement. But there is a deeper reason for our approach. A detailed consideration of computational complexity serves as a kind of "Occam's razor": the most efficient algorithms are generally those that compute exactly the information relevant to the problem situation. Thus the development of an especially efficient algorithm often gives us added insight into the problem we are considering, and the resultant algorithm is not only efficient but simple and elegant. Such algorithms are the kind we are after.

Of course, too much detail will obscure the most beautiful algorithm. We shall not develop FORTRAN programs here. Instead, we shall work at the level of simple operations on primitive mathematical objects, such as lists, trees and graphs. In §§1.3 through 1.5 we develop the necessary concepts and introduce our algorithmic notation. In Chapters 2 through 5 we use these ideas to develop four kinds of composite data structures that are useful in network optimization.

In Chapters 6 through 9, we combine these data structures with ideas from graph theory to obtain efficient algorithms for four network optimization tasks: finding minimum spanning trees, shortest paths, maximum flows, and maximum matchings. Not coincidentally, these are four of the five problems discussed by Klee in his excellent survey of network optimization [34]. Klee's fifth problem, the minimum tour problem, is one of the best known of the so-called "NP-complete" problems; as far as is known, it has no efficient algorithm. In §1.2, we shall review some of the concepts of computational complexity, to make precise the idea of an efficient algorithm and to provide a perspective on our results (see also [53], [55]).

We have chosen to formulate and solve network optimization problems in the setting of graph theory. Thus we shall omit almost all mention of two areas that

provide alternative approaches: matroid theory and linear programming. The books of Lawler [38] and Papadimitriou and Steiglitz [41] contain information on these topics and their connection to network optimization.

1.2. Computational complexity. In order to study the efficiency of algorithms, we need a model of computation. One possibility is to develop a denotational definition of complexity, as has been done for program semantics [19], but since this is a current research topic we shall proceed in the usual way and define complexity operationally. Historically the first machine model proposed was the *Turing machine* [56]. In its simplest form a Turing machine consists of a finite state control, a two-way infinite memory tape divided into squares, each of which can hold one of a finite number of symbols, and a read/write head. In one step the machine can read the contents of one tape square, write a new symbol in the square, move the head one square left or right, and change the state of the control.

The simplicity of Turing machines makes them very useful in high-level theoretical studies of computational complexity, but they are not realistic enough to allow accurate analysis of practical algorithms. For this purpose a better model is the *random-access machine* [1], [14]. A random-access machine consists of a finite program, a finite collection of registers, each of which can store a single integer or real number, and a memory consisting of an array of n words, each of which has a unique address between 1 and n (inclusive) and can hold a single integer or real number. In one step, a random-access machine can perform a single arithmetic or logical operation on the contents of specified registers, fetch into a specified register the contents of a word whose address is in a register, or store the contents of a register in a word whose address is in a register.

A similar but somewhat less powerful model is the *pointer machine* [35], [46], [54]. A pointer machine differs from a random-access machine in that its memory consists of an extendable collection of *nodes,* each divided into a fixed number of named *fields.* A field can hold a number or a pointer to a node. In order to fetch from or store into one of the fields in a node, the machine must have in a register a pointer to the node. Operations on register contents, fetching from or storing into node fields, and creating or destroying a node take constant time. In contrast to the case with random-access machines, address arithmetic is impossible on pointer machines, and algorithms that require such arithmetic, such as hashing [36], cannot be implemented on such machines. However, pointer machines make lower bound studies easier, and they provide a more realistic model for the kind of list-processing algorithms we shall study. A pointer machine can be simulated by a random-access machine in real time. (One operation on a pointer machine corresponds to a constant number of operations on a random-access machine.)

All three of these machine models share two properties: they are *sequential,* i.e., they carry out one step at a time, and *deterministic,* i.e., the future behavior of the machine is uniquely determined by its present configuration. Outside this section we shall not discuss parallel computation or nondeterminism, even though parallel algorithms are becoming more important because of the novel machine architectures made possible by very large scale integration (VLSI), and nondeterminism of

various kinds has its uses in both theory and practice [1], [19], [23]. One important research topic is to determine to what extent the ideas used in sequential, deterministic computation carry over to more general computational models.

An important caveat concerning random-access and pointer machines is that if the machine can manipulate numbers of arbitrary size in constant time, it can perform hidden parallel computation by encoding several numbers into one. There are two ways to prevent this. Instead of counting each operation as one step (the *uniform cost measure*), we can charge for an operation a time proportional to the number of bits needed to represent the operands (the *logarithmic cost measure*). Alternatively we can limit the size of the integers we allow to those representable in a constant times log n bits, where n is a measure of the input size, and restrict the operations we allow on real numbers. We shall generally use the latter approach; all our algorithms are implementable on a random-access or pointer machine with integers of size at most n^c for some small constant c with only comparison, addition, and sometimes multiplication of input values allowed as operations on real numbers, with no clever encoding.

Having picked a machine model, we must select a complexity measure. One possibility is to measure the complexity of an algorithm by the length of its program. This measure is *static*, i.e., independent of the input values. Program length is the relevant measure if an algorithm is only to be run once or a few times, and this measure has interesting theoretical uses [10], [37], [42], but for our purposes a better complexity measure is a *dynamic* one, such as running time or storage space as a function of input size. We shall use running time as our complexity measure; most of the algorithms we consider have a space bound that is a linear function of the input size.

In analyzing running times we shall ignore constant factors. This not only simplifies the analysis but allows us to ignore details of the machine model, thus giving us a complexity measure that is machine independent. As Fig. 1.1 illustrates, for large enough problem sizes the relative efficiencies of two algorithms depend on their running times as an asymptotic function of input size, independent of constant factors. Of course, what "large enough" means depends upon the situation; for some problems, such as matrix multiplication [15], the asymptotically most efficient known algorithms beat simpler methods only for astronomical problem sizes. The algorithms we shall consider are intended to be practical for moderate problem sizes. We shall use the following notation for asymptotic running times: If f and g are functions of nonnegative variables $n, m, \cdots$ we write "f is $O(g)$" if there are positive constants c_1 and c_2 such that $f(n, m, \cdots) \leq c_1 g(n, m, \cdots) + c_2$ for all values of $n, m, \cdots$. We write "f is $\Omega(g)$" if g is $O(f)$, and "f is $\Theta(g)$" if f is $O(g)$ and $\Omega(g)$.

We shall generally measure the running time of an algorithm as a function of the worst-case input data. Such an analysis provides a performance guarantee, but it may give an overly pessimistic estimate of the actual performance if the worst case occurs rarely. An alternative is an average-case analysis. The usual kind of averaging is over the possible inputs. However, such an analysis is generally much harder than worst-case analysis, and we must take care that our probability

COMPLEXITY \ SIZE	20	50	100	200	500	1000
$1000n$	.02 sec	.05 sec	.1 sec	.2 sec	.5 sec	1 sec
$1000n \lg n$	.09 sec	.3 sec	.6 sec	1.5 sec	4.5 sec	10 sec
$100n^2$	.04 sec	.25 sec	1 sec	4 sec	25 sec	2 min
$10n^3$	.02 sec	1 sec	10 sec	1 min	21 min	2.7 hr
$n \lg n$	.4 sec	1.1 hr	220 DAYS	125 CENT	5×10^8 CENT	
$2^{n/3}$	.0001 sec	.1 sec	2.7 hr	3×10^4 CENT		
2^n	1 sec	35 YR	3×10^4 CENT			
3^n	58 min	2×10^9 CENT				

FIG. 1.1. *Running time estimates. One step takes one microsecond,* lgn *denotes* $\log_2 n$.

distribution accurately reflects reality. A more robust approach is to allow the algorithm to make probabilistic choices. Thus for worst-case input data we average over possible algorithms. For certain problem domains, such as table look-up [9], [57], string matching [31], and prime testing [3], [43], [48], such randomized algorithms are either simpler or faster than the best known deterministic algorithms. For the problems we shall consider, however, this is not the case.

A third kind of averaging is *amortization.* Amortization is appropriate in situations where particular algorithms are repeatedly applied, as occurs with operations on data structures. By averaging the time per operation over a worst-case sequence of operations, we sometimes can obtain an overall time bound much smaller than the worst-case time per operation multiplied by the number of operations. We shall use this idea repeatedly.

By an *efficient algorithm* we mean one whose worst-case running time is bounded by a polynomial function of the input size. We call a problem *tractable* if it has an efficient algorithm and *intractable* otherwise, denoting by P the set of tractable problems. Cobham [12] and Edmonds [20] independently introduced this idea. There are two reasons for its importance. As the problem size increases, polynomial-time algorithms become unusable gradually, whereas nonpolynomial-time algorithms have a problem size in the vicinity of which the algorithm rapidly becomes completely useless, and increasing by a constant factor the amount of time allowed or the machine speed doesn't help much. (See Fig. 1.2.) Furthermore, efficient algorithms usually correspond to some significant structure in the problem, whereas inefficient algorithms often amount to brute-force search, which is defeated by combinatorial explosion.

TIME / COMPLEXITY	1 sec	10^2 sec (1.7 min)	10^4 sec (2.7 hr)	10^6 sec (12 DAYS)	10^8 sec (3 YEARS)	10^{10} sec (3 CENT.)
$1000n$	10^3	10^5	10^7	10^9	10^{11}	10^{13}
$1000n \lg n$	1.4×10^2	7.7×10^3	5.2×10^5	3.9×10^7	3.1×10^9	2.6×10^{11}
$100n^2$	10^2	10^3	10^4	10^5	10^6	10^7
$10n^3$	46	2.1×10^2	10^3	4.6×10^3	2.1×10^4	10^5
$n \lg n$	22	36	54	79	112	156
$2^{n/3}$	59	79	99	119	139	159
2^n	19	26	33	39	46	53
3^n	12	16	20	25	29	33

FIG. 1.2. *Maximum size of a solvable problem. A factor of ten increase in machine speed corresponds to a factor of ten increase in time.*

Figure 1.3 illustrates what we call the "spectrum of computational complexity," a plot of problems versus the complexities of their fastest known algorithms. There are two regions, containing the tractable and intractable problems. At the top of the plot are the *undecidable* problems, those with *no* algorithms at all. Lower are the problems that do have algorithms but only inefficient ones, running in exponential or superexponential time. These intractable problems form the subject matter of *high-level complexity*. The emphasis in high-level complexity is on proving non-

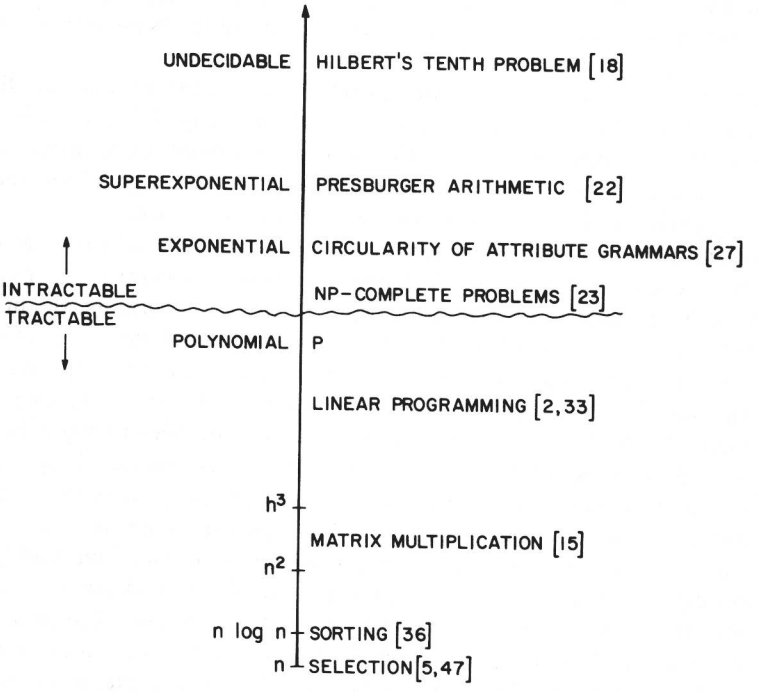

FIG. 1.3. *The spectrum of computational complexity.*

polynomial lower bounds on the time or space requirements of various problems. The machine model used is usually the Turing machine; the techniques used, simulation and diagonalization, derive from Godel's incompleteness proof [24], [40] and have their roots in classical self-reference paradoxes.

Most network optimization problems are much easier than any of the problems for which exponential lower bounds have been proved; they are in the class NP of problems solvable in polynomial time on a nondeterministic Turing machine. A more intuitive definition is that a problem is in NP if it can be phrased as a yes-no question such that if the answer is "yes" there is a polynomial-length proof of this. An example of a problem in NP is the *minimum tour problem:* given n cities and pairwise distances between them, find a tour that passes through each city once, returns to the starting point, and has minimum total length. We can phrase this as a yes-no question by asking if there is a tour of length at most x; a "yes" answer can be verified by exhibiting an appropriate tour.

Among the problems in NP are those that are hardest in the sense that if one has a polynomial-time algorithm then so does every problem in NP. These are the NP-*complete* problems. Cook [13] formulated this notion and illustrated it with several NP-complete problems; Karp [29], [30] established its importance by compiling a list of important problems, including the minimum tour problem, that are NP-complete. This list has now grown into the hundreds; see Garey and Johnson's book on NP-completeness [23] and Johnson's column in the Journal of Algorithms [28]. The NP-complete problems lie on the boundary between intractable and tractable. Perhaps the foremost open problem in computational complexity is to determine whether P = NP; that is, whether or not the NP-complete problems have polynomial-time algorithms.

The problems we shall consider all have efficient algorithms and thus lie within the domain of *low-level complexity,* the bottom half of Fig. 1.3. For such problems lower bounds are almost nonexistent; the emphasis is on obtaining faster and faster algorithms and in the process developing data structures and algorithmic techniques of wide applicability. This is the domain in which we shall work.

Although the theory of computational complexity can give us important information about the practical behavior of algorithms, it is important to be aware of its limitations. An example that illustrates this is *linear programming,* the problem of maximizing a linear function of several variables constrained by a set of linear inequalities. Linear programming is the granddaddy of network optimization problems; indeed, all four of the problems we consider can be phrased as linear programming problems. Since 1947, an effective, but not efficient algorithm for this problem has been known, the *simplex method* [16]. On problems arising in practice, the simplex method runs in low-order polynomial time, but on carefully constructed worst-case examples the algorithm takes an exponential number of arithmetic operations. On the other hand, the newly discovered *ellipsoid method* [2], [33], which amounts to a very clever n-dimensional generalization of binary search, runs in polynomial time with respect to the logarithmic cost measure but performs very poorly in practice [17]. This paradoxical situation is not well understood but is perhaps partially explained by three observations: (i) hard problems for the simplex method seem to be relatively rare; (ii) the average-case running time of the ellipsoid

method seems not much better than that for its worst case; and (iii) the ellipsoid method needs to use very high precision arithmetic, the cost of which the logarithmic cost measure underestimates.

1.3. Primitive data structures. In addition to integers, real numbers and bits (a bit is either **true** or **false**), we shall regard certain more complicated objects as primitive. These are *intervals, lists, sets,* and *maps.* An *interval* $[j\,..\,k]$ is a sequence of integers $j, j+1, \cdots, k$. We extend the notation to represent *arithmetic progressions:* $[j, k\,..\,l]$ denotes the sequence $j, j+\Delta, j+2\Delta, \cdots, j+i\Delta$, where $\Delta = k-j$ and $i = \lfloor (l-j)/\Delta \rfloor$. (If x is a real number, $\lfloor x \rfloor$ denotes the largest integer not greater than x and $\lceil x \rceil$ denotes the smallest integer not less than x.) If $i < 0$, the progression is empty; if $j = k$, the progression is undefined. We use $\in$ to denote membership and $\notin$ to denote nonmembership in intervals, lists and sets; thus for instance $i \in [j\,..\,k]$ means i is an integer such that $j \leq i \leq k$.

A *list* $q = [x_1, x_2, \cdots, x_n]$ is a sequence of arbitrary elements, some of which may be repeated. Element x_1 is the *head* of the list and x_n is the *tail;* x_1 and x_n are the *ends* of the list. We denote the *size* n of the list by $|q|$. An *ordered pair* $[x_1, x_2]$ is a list of two elements; [ ] denotes the *empty list* of no elements. There are three fundamental operations on lists:

Access. Given a list $q = [x_1, x_2, \cdots, x_n]$ and an integer i, return the ith element $q(i) = x_i$ on the list. If $i \notin [1\,..\,n]$, $q(i)$ has the special value **null.**

Sublist. Given a list $q = [x_1, x_2, \cdots, x_n]$ and a pair of integers i and j, return the list $q[i\,..\,j] = [x_i, x_{i+1}, \cdots, x_j]$. If j is missing or greater than n it has an implied value of n; similarly if i is missing or less than one it has an implied value of 1. Thus for instance $q[3\,..\,] = [x_3, x_4, \cdots, x_n]$. We can extend this notation to denote sublists corresponding to arithmetic progressions.

Concatenation. Given two lists $q = [x_1, x_2, \cdots, x_n]$ and $r = [y_1, y_2, \cdots, y_m]$, return their *concatenation* $q \,\&\, r = [x_1, x_2, \cdots, x_n, y_1, y_2, \cdots, y_m]$.

We can represent arbitrary insertion and deletion in lists by appropriate combinations of sublist and concatenation. Especially important are the special cases of access, sublist and concatenation that manipulate the ends of a list:

Access head. Given a list q, return $q(1)$.
Push. Given a list q and an element x, replace q by $[x] \,\&\, q$.
Pop. Given a list q, replace q by $q[2\,..\,]$.
Access tail. Given a list q, return $q(|q|)$.
Inject. Given a list q and an element x, replace q by $q \,\&\, [x]$.
Eject. Given a list q, replace q by $q[\,..\,|q|-1]$.

A list on which the operations access head, push and pop are possible is a *stack.* With respect to insertion and deletion a stack functions in a last-in, first-out manner. A list on which access head, inject and pop are possible is a *queue.* A queue functions in a first-in, first-out manner. A list on which all six operations are possible is a *deque* (double-ended queue). If all operations but eject are possible the list is an *output-restricted deque.* (See Fig. 1.4.)

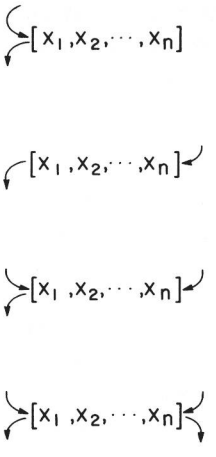

FIG. 1.4. *Types of lists.* (a) *Stack.* (b) *Queue.* (c) *Output-restricted deque.* (d) *Deque.*

A *set* $s = \{x_1, x_2, \cdots, x_n\}$ is a collection of distinct elements. Unlike a list, a set has no implied ordering of its elements. We extend the size notation to sets; thus $|s| = n$. We denote the empty set by $\{\ \}$. The important operations on sets are union $\cup$, intersection $\cap$, and difference $-$: if s and t are sets, $s - t$ is the set of all elements in s but not in t. We extend difference to lists as follows: if q is a list and s a set, $q - s$ is the list formed from q by deleting every copy of every element in s.

A *map* $f = \{[x_1, y_1], [x_2, y_2], \cdots, [x_n, y_n]\}$ is a set of ordered pairs no two having the same first coordinate (head). The *domain* of f is the set of first coordinates, **domain** $(f) = \{x_1, x_2, \cdots, x_n\}$. The *range* of f is the set of second coordinates (tails), **range** $(f) = \{y_1, y_2, \cdots, y_n\}$. We regard f as a function from the domain to the range; the *value* $f(x_i)$ of f at an element x_i of the domain is the corresponding second coordinate y_i. If $x \notin$ **domain** (f), $f(x) =$ **null.** The size $|f|$ of f is the size of its domain. The important operations on functions are accessing and redefining function values. The assignment $f(x) := y$ deletes the pair $[x, f(x)]$ (if any) from f and adds the pair $[x, y]$. The assignment $f(x) :=$ **null** merely deletes the pair $[x, f(x)]$ (if any) from f. We can regard a list q as a map with domain $[1\ .\ .\ |q|]$.

There are several good ways to represent maps, sets, and lists using arrays and linked structures (collections of nodes interconnected by pointers). We can represent a map as an array of function values (if the domain is an interval or can be easily transformed into an interval or part of one) or as a node field (if the domain is a set of nodes). These representations correspond to the memory structures of random-access and pointer machines respectively; they allow accessing or redefining $f(x)$ given x in $O(1)$ time. We shall use functional notation rather than dot notation to represent the values of node fields; depending upon the circumstances $f(x)$ may represent the value of map f at x, the value stored in position x of array f, the value of field f in node x, or the value returned by the function f when applied to x. These are all just alternative ways of representing functions. We shall use a small circle to denote function composition: $f \circ g$ denotes the function defined by $(f \circ g)(x) = f(g(x))$.

We can represent a set by using its characteristic function over some universe or by using one of the list representations discussed below and ignoring the induced order of the elements. If s is a subset of a universe U, its characteristic function χ_s over U is $\chi_s(x) =$ **true** if $x \in S$, **false** if $x \in U - S$. We call the value of $\chi_s(x)$ the *membership bit* of x (with respect to s). A characteristic function allows testing for membership in $O(1)$ time and can be updated in $O(1)$ time under addition or deletion of a single element. We can define characteristic functions for lists in the same way. Often a characteristic function is useful in combination with another set or list representation. If we need to know the size of a set frequently, we can maintain the size as an integer; updating the size after a one-element addition or deletion takes $O(1)$ time.

We can represent a list either by an array or by a linked structure. The easiest kind of list to represent is a stack. We can store a stack q in an array aq, maintaining the last filled position as an integer k. The correspondence between stack and array is $q(i) = aq(k + 1 - i)$; if $k = 0$ the stack is empty. With this representation each of the stack operations takes $O(1)$ time. In addition, we can access and even redefine arbitrary positions in the stack in $O(1)$ time. We can extend the representation to deques by keeping two integers j and k indicating the two ends of the deque and allowing the deque to "wrap around" from the back to the front of the array. (See Fig. 1.5.) The correspondence between deque and array is $q(i) = aq(((j + i - 1) \textbf{ mod } n) + 1)$, where n is the size of the array and x **mod** y denotes the remainder of x when divided by y. Each of the deque operations takes $O(1)$ time. If the elements of the list are nodes, it is sometimes useful to have a field in each node called a *list index* indicating the position of the node in the array. An array representation of a list is a good choice if we have a reasonably tight upper bound on the maximum size of the list and we do not need to perform many sublist and concatenate operations; such operations may require extensive copying.

There are many ways to represent a list as a linked structure. We shall consider eight, classified in three ways: as endogenous or exogenous, single or double and linear or circular. (See Fig. 1.6.) We call a linked data structure defining an arrangement of nodes *endogenous* if the pointers forming the "skeleton" of the structure are contained in the nodes themselves and *exogenous* if the skeleton is outside the nodes. In a single list, each node has a pointer to the next node on the list

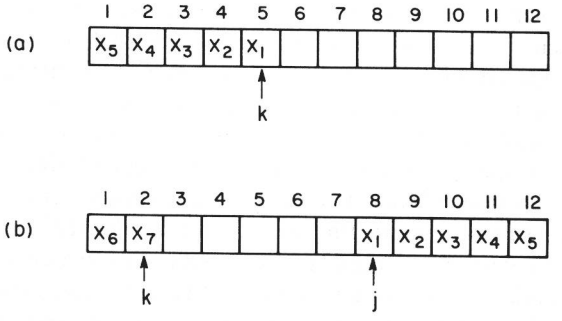

FIG. 1.5. *Array representation of lists.* (a) *Stack.* (b) *Deque that has wrapped around the array.*

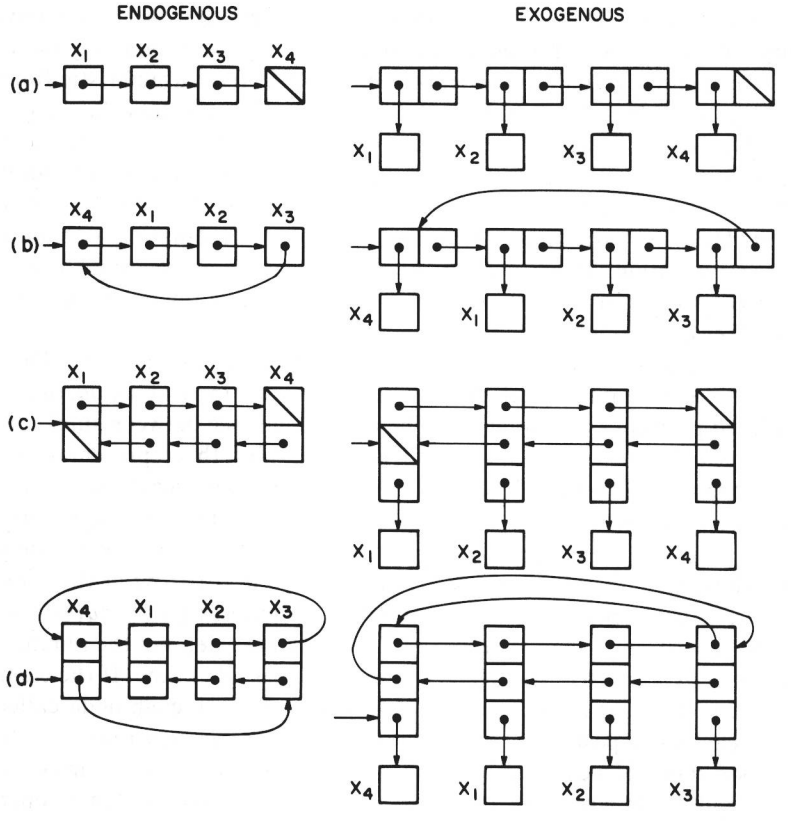

FIG. 1.6. *Linked representations of lists. Missing pointers are* **null.** (a) *Single linear.* (b) *Single circular.* (c) *Double linear.* (d) *Double circular.*

(its *successor*); in a double list, each node also has a pointer to the previous node (its *predecessor*). In a linear list, the successor of the last node is **null,** as is the predecessor of the first node; in a circular list, the successor of the last node is the first node and the predecessor of the first node is the last node. We access a linear list by means of a pointer to its head, a circular list by means of a pointer to its tail.

Figure 1.7 indicates the power of these representations. A single linear list suffices to represent a stack so that each access head, push, or pop operation takes $O(1)$ time. A single circular list suffices for an output-restricted deque and also allows concatenation in $O(1)$ time if we allow the concatenation to destroy its inputs. (All our uses of concatenation will allow this.) Single linking allows insertion of a new element after a specified one or deletion of the element after a specified one in $O(1)$ time; to have this capability if the list is exogenous we must store in each list element an inverse pointer indicating its position in the list. Single linking also allows scanning the elements of a list in order in $O(1)$ time per element scanned.

Double linking allows inserting a new element before a given one or deleting any element. It also allows scanning in reverse order. A double circular list suffices to represent a deque.

	SINGLE		DOUBLE	
	LINEAR	CIRCULAR	LINEAR	CIRCULAR
ACCESS HEAD	YES	YES	YES	YES
PUSH	YES	YES	YES	YES
POP	YES	YES	YES	YES
ACCESS TAIL	NO	YES	NO	YES
INJECT	NO	YES	NO	YES
EJECT	NO	NO	NO	YES
INSERT AFTER	YES(a)	YES(a)	YES(a)	YES(a)
INSERT BEFORE	NO	NO	YES(a)	YES(a)
DELETE AFTER	YES(a)	YES(a)	YES(a)	YES(a)
DELETE	NO	NO	YES(a)	YES(a)
CONCATENATE	NO	YES	NO	YES
REVERSE	NO	NO	NO	YES(b)
FORWARD SCAN	YES	YES	YES	YES
BACKWARD SCAN	NO	NO	YES	YES

FIG. 1.7. *The power of list representations. "Yes" denotes an* $O(1)$*–time operation (*$O(1)$ *time per element for forward and backward scanning).* (a) *If the representation is exogenous, insertion and deletion other than at the ends of the list require the position of the element. Inverse pointers furnish this information.* (b) *Reversal requires a modified representation. (See Fig.* 1.8.)

Endogenous structures are more space-efficient than exogenous ones, but they require that a given element be in only one or a fixed number of structures at a time. The array representation of a list can be regarded as an exogenous structure.

Some variations on these representations are possible. Instead of using circular linking, we can use linear linking but maintain a pointer to the tail as well as to the head of a list. Sometimes it is useful to make the head of a list a special dummy node called a *header;* this eliminates the need to treat the empty list as a special case.

Sometimes we need to be able to *reverse* a list, i.e. replace $q = [x_1, x_2, \cdots, x_n]$ by **reverse** $(q) = [x_n, x_{n-1}, \cdots, x_1]$. To allow fast reversal we represent a list by a double circular list accessed by a pointer to the tail, with the following modification: each node except the tail contains two pointers, to its predecessor and successor, but in no specified order; only for the tail is the order known. (See Fig. 1.8.) Since any access to the list is through the tail, we can establish the identity of predecessors and successors as nodes are accessed in $O(1)$ time per node. This representation allows all the deque operations, concatenation and reversal to be performed in $O(1)$ time per operation.

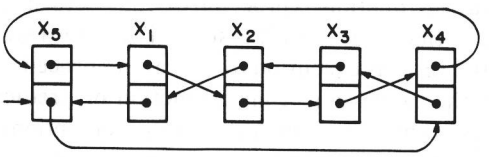

FIG. 1.8. *Endogenous representation of a reversible list.*