Data Structures*

Roberto Tamassia

Department of Computer Science
Brown University
115 Waterman Street
Providence, RI 02912–1910
rt@cs.brown.edu

January 2, 1996

1 Introduction

The study of data structures, i.e., methods for organizing data that are suitable for computer processing, is one of the classic topics of computer science. At the hardware level, a computer views storage devices such as internal memory and disk as holders of elementary data units (bytes), each accessible through its address (an integer). When writing programs, instead of manipulating the data at the byte level, it is convenient to organize them into higher level entities, called *data structures*.

Most data structures can be viewed as *containers* that store a collection of objects of a given type, called the *elements* of the container. Often a total order is defined among the elements (e.g., alphabetically ordered names, points in the plane ordered by x-coordinate).

A data structure has an associated repertory of operations, classified into *queries*, which retrieve information on the data structure (e.g., return the number of elements, or test the presence of a given element), and *updates*, which modify the data structure (e.g., insertion and deletion of elements). The performance of a data structure is characterized by the space requirement and the time complexity of the operations in its repertory. The *amortized* time complexity of an operation is the average time over a suitably defined sequence of operations.

Efficiency is not the only quality measure of a data structure. Simplicity and ease of implementation should be taken into account when choosing a data structure for solving a practical problem.

*Short contribution to the ACM 50th Anniversary issue of *Computing Surveys*.

Data structures are concrete implementations of *abstract data types* (ADT's). A *data type* is a collection of objects. A data type can be mathematically specified (e.g., real number, directed graph) or concretely specified within a programming language (e.g., `int` in C, `set` in Pascal). An ADT is a mathematically specified data type equipped with operations that can be performed on the objects. Object-oriented programming languages, such as C++, provide support for expressing ADTs by means of *classes*. ADTs specify the data stored and the operations to be performed on them.

The following issues are of foremost importance in the study of data structures.

Static vs. Dynamic A *static* data structure supports only queries, while a dynamic data structure supports also updates. A *dynamic* data structure is often more complicated than its static counterpart supporting the same repertory of queries. A *persistent* data structure is a dynamic data structure that supports operations on past versions. There are many problems for which no efficient dynamic data structures are known. It has been observed that there are strong similarities among the classes of problems that are difficult to parallelize and those that are difficult to dynamize. Further investigations are needed to study the relationship between parallel and incremental complexity.

Implicit vs. Explicit Two fundamental data organization mechanisms are used in data structures. In an *explicit* data structure, pointers (i.e., memory addresses) are used to link the elements and access them (e.g., a singly linked list, where each element has a pointer to the next one). In *implicit* data structure, mathematical relationships support the retrieval of elements (e.g., array representation of a heap). Explicit data structures must use additional space to store pointers. However, they are more flexible for complex problems. Most programming languages support pointers and basic implicit data structures, such as arrays.

Internal vs. External Memory In a typical computer, there are two levels of memory: internal memory (RAM) and external memory (disk). The internal memory is much faster than external memory but has much smaller capacity. Data structures designed to work for data that fit into internal memory may not perform well for large amounts of data that need to be stored in external memory. For large-scale problems, data structures need to be designed that take into account the two levels of memory. For example, two-level indices such as B-trees have been designed to efficiently search in large databases.

Space vs. Time Data structures often exhibit a tradeoff between space and time complexity. For example, suppose we want to represent a set of integers in the range $[0, N]$ (e.g., for a set of social security numbers $N =$

$10^{10}-1$) such that we can efficiently query whether a given element is in the set, insert an element, or delete an element. Two possible data structures for this problem are an N-element bit-array (where the bit in position i indicates the presence of integer i in the set), and a balanced search tree (such as a 2-3 tree or a red-black tree). The bit-array has optimal time complexity, since it supports queries, insertions and deletions in constant time. However, it uses space proportional to the size N of the range, irrespectively of the number of elements actually stored. The balanced search tree supports queries, insertions and deletions in logarithmic time but uses optimal space proportional to the current number of elements stored.

Theory vs. Practice A large and ever growing body of theoretical research on data structures is available, where the performance is measured in asymptotic terms ("big-Oh" notation). While asymptotic complexity analysis is an important mathematical subject, it does not completely capture the notion of efficiency of data structures in practical scenarios, where constant factors cannot be disregarded and the difficulty of implementation substantially affects design and maintenance costs. Experimental studies comparing the practical efficiency of data structures for specific classes of problems should be encouraged to bridge the gap between the theory and practice of data structures.

2 Fundamental Data Structures

The following four data structures are ubiquitously used in the description of discrete algorithms, and serve as basic building blocks for realizing more complex data structures.

Sequence A sequence is a container that stores elements in a certain linear order, which is imposed by the operations performed. The basic operations supported are retrieving, inserting, and removing an element given its position. Special types of sequences include stacks and queues, where insertions and deletions can be done only at the head or tail of the sequence. The basic realization of sequences are by means of arrays and linked lists. Concatenable queues support additional operations such as splitting and splicing, and determining the sequence containing a given element. In external memory, a sequence is typically associated with a file.

Priority Queue A priority queue is a container of elements from a totally ordered universe that supports the basic operations of inserting an element and retrieving/removing the largest element. A key application of priority queues is to sorting algorithms. A heap is an efficient realization of a priority queue that embeds the elements into the ancestor/descendant partial order of a binary tree. A heap also admit an implicit realization where the nodes of the tree are mapped into the elements of an array. Sophisticated variations of priority queues include pagodas, binomial heaps, and Fibonacci heaps. The buffer tree is efficient external-memory realization of a priority queue.

Dictionary A dictionary is a container of elements from a totally ordered universe that supports the basic operations of inserting/deleting elements and searching for a given element. Hash tables provide an efficient implicit realization of a dictionary. Efficient explicit implementations include balanced search trees (e.g., AVL-trees, red-black trees, 2-3 trees, weight-balanced trees) and skip lists. The technique of fractional cascading speeds up searching for the same element in a collection of dictionaries. In external memory, dictionaries are typically implemented as B-trees and their variations.

Union-Find A union-find data structure represents a collection disjoint sets and supports the two fundamental operations of merging two sets and finding the set containing a given element. There is a simple and optimal union-find data structure (rooted tree with path compression) whose time complexity analysis is very difficult to analyze.

Examples of fundamental data structures used in three major application domains are mentioned below.

Graphs and Networks adjacency matrix, adjacency lists, link-cut tree, dynamic expression tree, topology tree, block-cutpoint tree, SPQR-tree, sparsification tree.

Text Processing string, suffix tree, Patricia tree.

Geometry and Graphics binary space partition tree, chain tree, trapezoid tree, range tree, segment-tree, interval-tree, priority-search tree, hull-tree, quad-tree, R-tree, grid file, metablock tree.

3 Further Information

Many textbooks and monographs have been written on data structures, e.g., [1, 3, 5, 6, 7, 8, 9, 10, 13, 14, 16, 17, 15, 19].

Recent papers surveying the state-of-the art in data structures include [2, 4, 12, 18].

The LEDA project [11] aims at developing a C++ library of efficient and reliable implementations of sophisticated data structures.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[2] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434, Sept. 1992.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.

[4] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, 1991.

[5] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison Wesley, 1991.

[6] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Science Press, Potomac, Maryland, 1983.

[7] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1968.

[8] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[9] H. R. Lewis and L. Denenberg. *Data Structures and Their Algorithms*. Harper Collins, 1991.

[10] K. Mehlhorn. *Data Structures and Algorithms*. Springer-Verlag, 1984. Volumes 1–3.

[11] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *CACM*, 38:96–102, 1995. `http://www.mpi-sb.mpg.de/guide/staff/uhrig/leda.html`.

[12] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*. Elsevier, Amsterdam, 1990.

[13] J. Nievergelt and K. H. Hinrichs. *Algorithms and Data Structures: With Applications to Graphics and Geometry*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[14] M. H. Overmars. *The design of dynamic data structures*, volume 156 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[15] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[16] R. Sedgewick. *Algorithms in C++*. Addison Wesley, Reading, MA, 1992.

[17] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial Applied Mathematics, 1983.

[18] J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, pages 431–524. Elsevier, Amsterdam, 1990.

[19] D. Wood. *Data Structures, Algorithms, and Performance*. Addison Wesley, 1993.