Part V: Data structures

The tools of bookkeeping

When thinking of algorithms we emphasize a dynamic sequence of actions: "Take this and do that, then that, then ... ." In human experience, "take" is usually a straightforward operation, whereas "do" means work. In programming, on the other hand, there are lots of interesting examples where "do" is nothing more complex than incrementing a counter or setting a bit; but "take" triggers a long, sophisticated search. Why do we need fancy data structures at all? Why can't we just spread out the data on a desk top? Everyday experience does not prepare us to appreciate the importance of data structure—it takes programming experience to see that algorithms are nothing without data structures. The algorithms presented so far were carefully chosen to require only the simplest of data structures: static arrays. The geometric algorithms of Part VI, on the other hand, and lots of other useful algorithms, depend on sophisticated data structures for their efficiency.

The key insight in understanding data structures is the recognition that an algorithm in execution is, at all times, in some state, chosen from a potentially huge state space. The state records such vital information as what steps have already been taken with what results, and what remains to be done. Data structures are the bookkeepers that record all this state information in a tidy manner so that any part can be accessed and updated efficiently. The remarkable fact is that there are a relatively small number of standard data structures that turn out to be useful in the most varied types of algorithms and problems, and constitute essential knowledge for any programmer.

The literature on data structures. Whereas one can present some algorithms without emphasizing data structures, as we did in Part III, it appears pointless to discuss data structures without some of the typical algorithms that use them; at the very least, access and update algorithms form a necessary part of any data structure. Accordingly, a new data structure is typically published in the context of a particular new algorithm. Only later, as one notices its general applicability, it may find its way into textbooks. The data structures that have become standard today can be found in many books, such as [AHU 83], [CLR 90], [GB 91], [HS 82], [Knu 73a], [Knu 73b], [Meh 84a], [Meh 84c], [RND 77], [Sam 90a], [Sam 90b], [Tar 83], and [Wir 86].

18. What is a data structure?

Learning objectives:

- data structures for manual use (e.g. edge-notched cards)
- general-purpose data structures
- abstract data types specify functional properties only
- data structures include access and maintenance algorithms and their implementation
- performance criteria and measures
- asymptotics

Data structures old and new

The discipline of data structures, as a systematic body of knowledge, is truly a creation of computer science. The question of how best to organize data was a lot simpler to answer in the days before the existence of computers: the organization had to be simple, because there was no automatic device that could have processed an elaborate data structure, and there is no human being with enough patience to do it. Consider two examples.

1. Manual files and catalogs, as used in business offices and libraries, exhibit several distinct organizing principles, such as sequential and hierarchical order and cross-references. From today's point of view, however, manual files are not well-defined data structures. For good reasons, people did not rigorously define those aspects that we consider essential when characterizing a data structure: what constraints are imposed on the data, both on the structure and its content; what operations the data structure must support; what constraints these operations must satisfy. As a consequence, searching and updating a manual file is not typically a process that can be automated: It requires common sense, and perhaps even expert training, as is the case for a library catalog.
2. In manual computing (with pencil and paper or a nonprogrammable calculator) the *algorithm* is the focus of attention, not the data structure. Most frequently, the person computing writes data (input, intermediate results, output) in any convenient place within his field of vision, hoping to find them again when he needs them. Occasionally, to facilitate highly repetitive computations (such as income tax declarations), someone designs a form to prompt the user, one operation at a time, to write each data item into a specific field. Such a form specifies both an algorithm and a data structure with considerable formality. Compared to the general-purpose data structures we study in this chapter, however, such forms are highly special purpose.

Edge-notched cards are perhaps the most sophisticated data structures ever designed for manual use. Let us illustrate them with the example of a database of English words organized so as to help in solving crossword puzzles. We write one word per card and index it according to which vowels it contains and which ones it does not contain. Across the top row of the card we punch 10 holes labeled A, E, I, O, U, ~A, ~E, ~I, ~O, ~U. When a word, say ABACA, exhibits a given vowel, such as A, we cut a notch above the hole for A; when it does not, such as E, we cut a notch above the hole for ~E (pronounced "not E"). Exhibit 18.1 shows the encoding of the words BEAUTIFUL, EXETER, OMAHA, OMEGA. For example, we search for words that contain at least one E, but no U, by sticking

two needles through the pack of cards at the holes E and ~U. EXETER and OMEGA will drop out. In principle it is easy to make this sample database more powerful by including additional attributes, such as "A occurs exactly once", "A occurs exactly twice", "A occurs as the first letter in the word", and so on. In practice, a few dozen attributes and thousands of cards will stretch this mechanical implementation of a multikey data structure to its limits of feasibility.

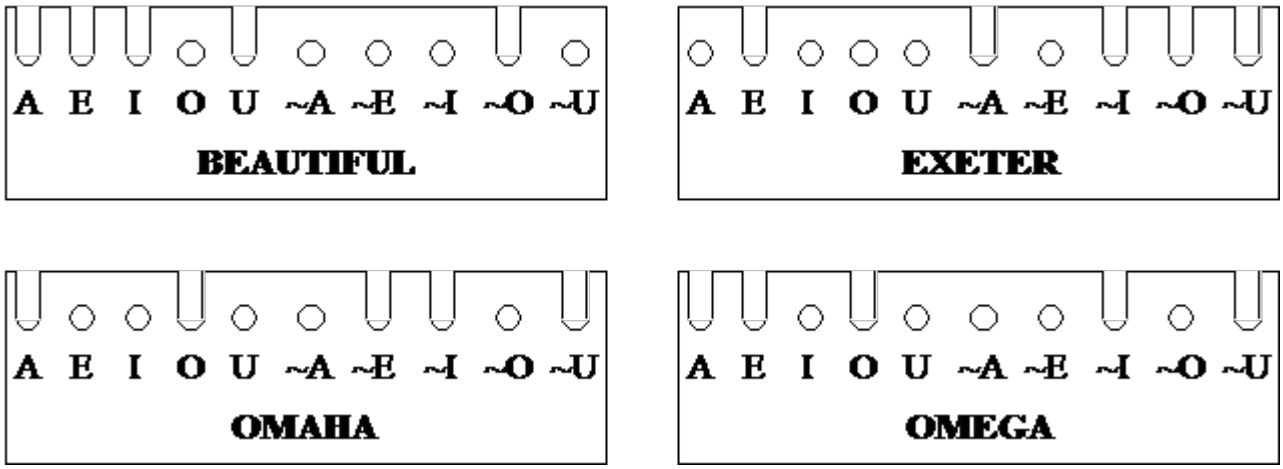

Exhibit 18.1: Encoding of different words in edge-notched cards.

In contrast to data structures suitable for manual processing, those developed for automatic data processing can be complex. Complexity is not a goal in itself, of course, but it may be an unavoidable consequence of the search for efficiency. Efficiency, as measured by processing time and memory space required, is the primary concern of the discipline of data structures. Other criteria, such as simplicity of the code, play a role, but the first question to be asked when evaluating a data structure that supports a specified set of operations is typically: How much time and space does it require?

In contrast to the typical situation of manual computing (consideration of the algorithm comes first, data gets organized only as needed), programmed computing typically proceeds in the opposite direction: First we define the organization of the data rigorously, and from this the structure of the algorithm follows. Thus algorithm design is often driven by data structure design.

The range of data structures studied

We present generally useful data structures along with the corresponding query, update, and maintenance algorithms; and we develop concepts and techniques designed to organize a vast body of knowledge into a coherent whole. Let us elaborate on both of these goals.

"Generally useful" refers to data structures that occur naturally in many applications. They are relatively simple from the point of view of the operations they support—tables and queues of various types are typical examples. These basic data structures are the building blocks from which an applications programmer may construct more elaborate structures tailored to her particular application. Although our collection of specific data structures is rather small, it covers the great majority of techniques an applications programmer is likely to need.

We develop a unified scheme for understanding many data structures as special cases of general concepts. This includes:

- The separation of abstract data types, which specify only functional properties, from data structures, which also involve aspects of implementation
- The classification of all data structures into three major types: implicit data structures, lists, and address computation
- A rough assessment of the performance of data structures based on the asymptotic analysis of time and memory requirements

The simplest and most common assumption about the elements to be stored in a data structure is that they belong to a domain on which a total order ≤ is defined. *Examples:* integers ordered by magnitude, a character set with its alphabetic order, character strings of bounded length ordered lexicographically. We assume that each element in a domain requires as much storage as any other element in that domain; in other words, that a data structure manages memory fragments of fixed size. Data objects of greatly variable size or length, such as fragments of text, are typically not considered to be "elements"; instead, they are broken into constituent pieces of fixed size, each of which becomes an element of the data structure.

The elements stored in a data structure are often processed according to the order ≤ defined on their domain. The topic of *sorting*, which we surveyed in “Sorting and its complexity”, is closely related to the study of data structures: Indeed, several sorting algorithms appear "for free" in “List structures”, because every structure that implements the abstract data type *dictionary* leads to a sorting algorithm by successive insertion of elements, followed by a traversal.

Performance criteria and measures

The design of data structures is dominated by considerations of efficiency, specifically with respect to time and memory. But efficiency is a multifaceted quality not easily defined and measured. As a scientific discipline, the study of data structures is not directly concerned with the number of microseconds, machine cycles, or bytes required by a specific program processing a given set of data on a particular system. It is concerned with general statements from which an expert practitioner can predict concrete outcomes for a specific processing task. Thus, measuring run times and memory usage is not the typical way to evaluate data structures. We need concepts and notations for expressing the performance of an algorithm independently of machine speed, memory size, programming language, and operating system, and a host of other details that vary from run to run.

The solution to this problem emerged over the past two decades as the discipline of computational complexity was developed. In this theory, algorithms are "executed" on some "mathematical machine", carefully designed to be as simple as possible to reflect the bare essentials of a problem. The machine makes available certain *primitive operations*, and we measure "time" by counting how many of those are executed. For a given algorithm and all the data sets it accepts as input, we analyze the number of primitive operations executed as a function of the size of the data. We are often interested in the *worst case*, that is, a data set of given size that causes the algorithm to run as long as possible, and the *average case*, the run time averaged over all data sets of a given size.

Among the many different mathematical machines that have been defined in the theory of computation, data structures are evaluated almost exclusively with respect to a theoretical *random access machine* (RAM). A RAM is essentially a memory with as many locations as needed, each of which can hold a data element, such as an integer, or a real number; and a processing unit that can read from any one or two locations, operate on their content, and write the result back into a third location, all in one time unit. This model is rather close to actual sequential

computers, except that it incorporates no bounds on the memory size—either in terms of the number of locations or the size of the content of this location. It implies, for example, that a multiplication of two very large numbers requires no more time than $2 \cdot 3$ does. This assumption is unrealistic for certain problems, but is an excellent one for most program runs that fit in central memory and do not require variable-precision arithmetic or variable-length data elements. The point is that the programmer has to understand the model and its assumptions, and bears responsibility for applying it judiciously.

In this model, time and memory requirements are expressed as functions of input data size, and thus comparing the performance of two data structures is reduced to comparing functions. *Asymptotics* has proven to be just the right tool for this comparison: sharp enough to distinguish different growth rates, blunt enough to ignore constant factors that differ from machine to machine.

As an example of the concise descriptions made possible by asymptotic operation counts, the following table evaluates several implementations for the abstract data type 'dictionary'. The four operations 'find', 'insert', 'delete', and 'next' (with respect to the order $\leq$) exhibit different asymptotic time requirements for the different implementations. The student should be able to explain and derive this table after studying this part of the book.

	Ordered array	Linear list	Balanced tree	Hash table
find	O(log n)	O(n)	O(log n)	O(1)[a]
next	O(1)	O(1)	O(log n)	O(n)
insert	O(n)	O(n)	O(log n)	O(1)[a]
delete	O(n)	O(n)	O(log n)	O(1)[b]

[a] On the average, but not necessarily in the worst case

[b] Deletions are possible but may degrade performance

Exercise

1. Describe the manual data structures that have been developed to organize libraries (e.g. catalogs that allow users to get access to the literature in their field of interest, or circulation records, which keep track of who has borrowed what book). Give examples of queries that can be answered by these data structures.