CHAPTER 1

Introduction

1.1 Why Compact Data Structures?

Google's stated mission, "to organize the world's information and make it universally accessible and useful," could not better capture the immense ambition of modern society for gathering all kinds of data and putting them to use to improve our lives. We are collecting not only huge amounts of data from the physical world (astronomical, climatological, geographical, biological), but also human-generated data (voice, pictures, music, video, books, news, Web contents, emails, blogs, tweets) and society-based behavioral data (markets, shopping, traffic, clicks, Web navigation, likes, friendship networks).

Our hunger for more and more information is flooding our lives with data. Technology is improving and our ability to store data is growing fast, but the data we are collecting also grow fast – in many cases faster than our storage capacities. While our ability to store the data in secondary or perhaps tertiary storage does not yet seem to be compromised, performing the desired processing of these data in the main memory of computers is becoming more and more difficult. Since accessing a datum in main memory is about 10^5 times faster than on disk, operating in main memory is crucial for carrying out many data-processing applications.

In many cases, the problem is not so much the size of the actual data, but that of the *data structures* that must be built on the data in order to efficiently carry out the desired processing or queries. In some cases the data structures are one or two orders of magnitude larger than the data! For example, the DNA of a human genome, of about 3.3 billion bases, requires slightly less than 800 megabytes if we use only 2 bits per base (`A`, `C`, `G`, `T`), which fits in the main memory of any desktop PC. However, the suffix tree, a powerful data structure used to efficiently perform sequence analysis on the genome, requires at least 10 bytes per base, that is, more than 30 gigabytes.

The main techniques to cope with the growing size of data over recent years can be classified into three families:

Efficient secondary-memory algorithms. While accessing a random datum from disk is comparatively very slow, subsequent data are read much faster, only 100 times slower than from main memory. Therefore, algorithms that minimize the random accesses to the data can perform reasonably well on disk. Not every problem, however, admits a good disk-based solution.

Streaming algorithms. In these algorithms one goes to the extreme of allowing only one or a small number of sequential passes over the data, storing intermediate values on a comparatively small main memory. When only one pass over the data is allowed, the algorithm can handle situations in which the data cannot even be stored on disk, because they either are too large or flow too fast. In many cases streaming algorithms aim at computing approximate information from the data.

Distributed algorithms. These are parallel algorithms that work on a number of computers connected through a local-area network. Network transfer speeds are around 10 times slower than those of disks. However, some algorithms are amenable to parallelization in a way that the data can be partitioned over the processors and little transfer of data is needed.

Each of these approaches pays a price in terms of performance or accuracy, and neither one is always applicable. There are also cases where memory is limited and a large secondary memory is not at hand: routers, smartphones, smartwatches, sensors, and a large number of low-end embedded devices that are more and more frequently seen everywhere (indeed, they are the stars of the promised Internet of Things).

A topic that is strongly related to the problem of managing large volumes of data is *compression*, which seeks a way of representing data using less space. Compression builds on Information Theory, which studies the minimum space necessary to represent the data.

Most compression algorithms require decompressing all of the data from the beginning before we can access a random datum. Therefore, compression generally serves as a space-saving *archival* method: the data can be *stored* using less space but must be fully decompressed before being used again. Compression is not useful for managing more data in main memory, except if we need only to process the data sequentially.

Compact data structures aim precisely at this challenge. A compact data structure maintains the data, and the desired extra data structures over it, in a form that not only uses less space, but is able to access and query the data *in compact form*, that is, without decompressing them. Thus, a compact data structure allows us to fit and efficiently query, navigate, and manipulate much larger datasets in main memory than what would be possible if we used the data represented in plain form and classical data structures on top.

Compact data structures lie at the intersection of Data Structures and Information Theory. One looks at data representations that not only need space close to the minimum possible (as in compression) but also require that those representations allow one to efficiently carry out some operations on the data. In terms of information, data structures are fully *redundant*: they can be reconstructed from the data itself. However, they are built for efficiency reasons: once they are built from the data, data structures speed up operations significantly. When designing compact data structures, one struggles with this tradeoff: supporting the desired operations as efficiently as possible while

increasing the space as little as possible. In some lucky cases, a compact data structure reaches almost the minimum possible space to represent the data and provides a rich functionality that encompasses what is provided by a number of independent data structures. General trees and text collections are probably the two most striking success stories of compact data structures (and they have been combined to store the human genome *and* its suffix tree in less than 4 gigabytes!).

Compact data structures usually require more steps than classical data structures to complete the same operations. However, if these operations are carried out on a faster memory, the net result is a faster (and smaller) representation. This can occur at any level of the memory hierarchy; for example, a compact data structure may be faster because it fits in cache when the classical one does not. The most dramatic improvement, however, is seen when the compact data structure fits in main memory while the classical one needs to be handled on disk (even if it is a solid-state device). In some cases, such as limited-memory devices, compact data structures may be the only approach to operate on larger datasets.

The other techniques we have described can also benefit from the use of compact data structures. For example, distributed algorithms may use fewer computers to carry out the same task, as their aggregated memory is virtually enlarged. This reduces hardware, communication, and energy costs. Secondary-memory algorithms may also benefit from a virtually larger main memory by reducing the amount of disk transfers. Streaming algorithms may store more accurate estimations within the same main memory budget.

1.2 Why This Book?

The starting point of the formal study of compact data structures can be traced back to the 1988 Ph.D. thesis of Jacobson, although earlier works, in retrospect, can also be said to belong to this area. Since then, the study of these structures has fluorished, and research articles appear routinely in most conferences and journals on algorithms, compression, and databases. Various software repositories offer mature libraries implementing generic or problem-specific compact data structures. There are also indications of the increasing use of compact data structures inside the products of Google, Facebook, and others.

We believe that compact data structures have reached a level of maturity that deserves a book to introduce them. There are already established compact data structures to represent bitvectors, sequences, permutations, trees, grids, binary relations, graphs, tries, text collections, and others. Surprisingly, there are no other books on this topic as far as we know, and for many relevant structures there are no survey articles.

This book aims to introduce the reader to the fascinating algorithmic world of the compact data structures, with a strong emphasis on practicality. Most of the structures we present have been implemented and found to be reasonably easy to code and efficient in space and time. A few of the structures we present have not yet been implemented, but based on our experience we believe they will be practical as well. We have obtained the material from the large universe of published results and from our own experience, carefully choosing the results that should be most relevant to a

practitioner. Each chapter finishes with a list of selected references to guide the reader who wants to go further.

On the other hand, we do not leave aside the theory, which is essential for a solid understanding of why and how the data structures work, and thus for applying and extending them to face new challenges. We gently introduce the reader to the beauty of the algorithmics and the mathematics that are behind the study of compact data structures. Only a basic background is expected from the reader. From algorithmics, knowledge of sorting, binary search, dynamic programming, graph traversals, hashing, lists, stacks, queues, priority queues, trees, and $\mathcal{O}$-notation suffices (we will briefly review this notation later in this chapter). This material corresponds to a first course on algorithms and data structures. From mathematics, understanding of induction, basic combinatorics, probability, summations, and limits, that is, a first-year university course on algebra or discrete mathematics, is sufficient.

We expect this book to be useful for advanced undergraduate students, graduate students, researchers, and professionals interested in algorithmic topics. Hopefully you will enjoy the reading as much as I have enjoyed writing it.

1.3 Organization

The book is divided into 13 chapters. Each chapter builds on previous ones to introduce a new concept and includes a section on applications and a bibliographic discussion at the end. Applications are smaller or more specific problems where the described data structures provide useful solutions. Most can be safely skipped if the reader has no time, but we expect them to be inspiring. The bibliography contains annotated references pointing to the best sources of the material described in the chapter (which not always are the first publications), the most relevant historic landmarks in the development of the results, and open problems. This section is generally denser and can be safely skipped by readers not interested in going deeper, especially into the theoretical aspects.

Pseudocode is included for most of the procedures we describe. The pseudocode is presented in an algorithmic language, not in any specific programming language. For example, well-known variables are taken as global without notice, widely known procedures such as a binary search are not detailed, and tedious but obvious details are omitted (with notice). This lets us focus on the important aspects that we want the pseudocode to clear up; our intention is not that the pseudocode is a cut-and-paste text to get the structures running without understanding them. We refrain from making various programming-level optimizations to the pseudocode to favor clarity; any good programmer should be able to considerably speed up a verbatim implementation of the pseudocodes without altering their logic.

After this introductory chapter, Chapter 2 introduces the concepts of Information Theory and compression needed to follow the book. In particular, we introduce the concepts of worst-case, Shannon, and empirical entropy and their relations. This is the most mathematical part of the book. We also introduce Huffman codes and codes suitable for small integers.

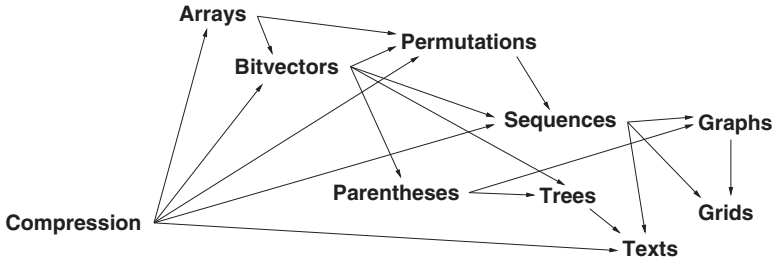

Figure 1.1. The most important dependencies among Chapters 2–11.

The subsequent chapters describe compact data structures for different problems. Each compact data structure stores some kind of data and supports a well-defined set of operations. Chapter 3 considers arrays, which support the operations of reading and writing values at arbitrary positions. Chapter 4 describes bitvectors, arrays of bits that in addition support a couple of bit-counting operations. Chapter 5 covers representations of permutations that support both the application of the permutation and its inverse as well as powers of the permutation. Chapter 6 considers sequences of symbols, which, apart from accessing the sequence, support a couple of symbol-counting operations. Chapter 7 addresses hierarchical structures described with balanced sequences of parentheses and operations to navigate them. Chapter 8 deals with the representation of general trees, which support a large number of query and navigation operations. Chapter 9 considers graph representations, both general ones and for some specific families such as planar graphs, allowing navigation toward neighbors. Chapter 10 considers discrete two-dimensional grids of points, with operations for counting and reporting points in a query rectangle. Chapter 11 shows how text collections can be represented so that pattern search queries are supported.

As said, each chapter builds upon the structures described previously, although most of them can be read independently with only a conceptual understanding of what the operations on previous structures mean. Figure 1.1 shows the most important dependencies for understanding why previous structures reach the claimed space and time performance.

These chapters are dedicated to *static* data structures, that is, those that are built once and then serve many queries. These are the most developed and generally the most efficient ones. We pay attention to construction time and, especially, construction space, ensuring that structures that take little space can also be built within little extra memory, or that the construction is disk-friendly. Structures that support updates are called *dynamic* and are considered in Chapter 12.

The book concludes in Chapter 13, which surveys some current research topics on compact data structures: encoding data structures, indexes for repetitive text collections, and data structures for secondary storage. Those areas are not general or mature enough to be included in previous chapters, yet they are very promising and will probably be the focus of much research in the upcoming years. The chapter then also serves as a guide to current research topics in this area.

Although we have done our best to make the book error-free, and have manually verified the algorithms several times, it is likely that some errors remain. A Web page with comments, updates, and corrections on the book will be maintained at http://www.dcc.uchile.cl/gnavarro/CDSbook.

1.4 Software Resources

Although this book focuses on understanding the compact data structures so that the readers can implement them by themselves, it is worth noting that there are several open-source software repositories with mature implementations, both for general and for problem-specific compact data structures. These are valuable both for practitioners that need a structure implemented efficiently, well tested, and ready to be used, and for students and researchers that wish to build further structures on top of them. In both cases, understanding why and how each structure works is essential to making the right decisions on which structure to use for which problem, how to parameterize it, and what can be expected from it.

Probably the most general, professional, exhaustive, and well tested of all these libraries is Simon Gog's *Succinct Data Structure Library (SDSL)*, available at https://github.com/simongog/sdsl-lite. It contains `C++` implementations of compact data structures for bitvectors, arrays, sequences, text indexes, trees, range minimum queries, and suffix trees, among others. The library includes tools to verify correctness and measure efficiency along with tutorials and examples.

Another generic library is Francisco Claude's *Library of Compact Data Structures (LIBCDS)*, available at https://github.com/fclaude/libcds. It contains optimized and well-tested `C++` implementations of bitvectors, sequences, permutations, and others. A tutorial on how to use the library and how it works is included.

Sebastiano Vigna's *Sux* library, available at http://sux.di.unimi.it, contains high-quality `C++` and/or `Java` implementations of various compact data structures, including bitvectors, arrays with cells of varying lengths, and (general and monotone) minimal perfect hashing. Other projects accessible from there include sophisticated tools to manage inverted indexes and Web graphs in compressed form.

Giuseppe Ottaviano's *Succinct* library provides efficient `C++` implementations of bitvectors, arrays of fixed and variable-length cells, range minimum queries, and others. It is available at https://github.com/ot/succinct.

Finally, Nicola Prezza's *Dynamic* library provides C++ implementations of various data structures supporting insertions of new elements: partial sums, bitvectors, sparse arrays, strings, and text indexes. It is available at https://github.com/nicolaprezza/DYNAMIC.

The authors of many of these libraries have explored much deeper practical aspects of the implementation, including cache efficiency, address translation, word alignments, machine instructions for long computer words, instruction pipelining, and other issues beyond the scope of this book.

Many other authors of articles on practical compact data structures for specific problems have left their implementations publicly available or are willing to share them upon request. There are too many to list here, but browsing the personal pages

of the authors, or requesting the code, is probably a fast way to obtain a good implementation.

1.5 Mathematics and Notation

This final technical section is a reminder of the mathematics behind the $\mathcal{O}$-notation, which we use to describe the time performance of algorithms and the space usage of data structures. We also introduce other notation used throughout the book.

***$\mathcal{O}$*-notation.** This notation is used to describe the asymptotic growth of functions (for example, the cost of an algorithm as a function of the size of the input) in a way that considers only sufficiently large values of the argument (hence the name "asymptotic") and ignores constant factors.

Formally, $\mathcal{O}(f(n))$ is the set of all functions $g(n)$ for which there exist constants $c > 0$ and $n_0 > 0$ such that, for all $n > n_0$, it holds $|g(n)| \leq c \cdot |f(n)|$. We say that $g(n)$ is $\mathcal{O}(f(n))$, meaning that $g(n) \in \mathcal{O}(f(n))$. Thus, for example, $3n^2 + 6n - 3$ is $\mathcal{O}(n^2)$ and also $\mathcal{O}(n^3)$, but it is not $\mathcal{O}(n \log n)$. In particular, $\mathcal{O}(1)$ is used to denote a function that is always below some constant. For example, the cost of an algorithm that, independently of the input size, performs 3 accesses to tables and terminates is $\mathcal{O}(1)$. An algorithm taking $\mathcal{O}(1)$ time is said to be constant-time.

It is also common to abuse the notation and write $g(n) = \mathcal{O}(f(n))$ to mean $g(n) \in \mathcal{O}(f(n))$, and even to write, say, $g(n) < 2n + \mathcal{O}(\log n)$, meaning that $g(n)$ is smaller than $2n$ plus a function that is $\mathcal{O}(\log n)$. Sometimes we will write, for example, $g(n) = 2n - \mathcal{O}(\log n)$, to stress that $g(n) \leq 2n$ and the function that separates $g(n)$ from $2n$ is $\mathcal{O}(\log n)$.

Several other notations are related to $\mathcal{O}$. Mostly for lower bounds, we write $g(n) \in \Omega(f(n))$, meaning that there exist constants $c > 0$ and $n_0 > 0$ such that, for all $n > n_0$, it holds $|g(n)| \geq c \cdot |f(n)|$. Alternatively, we can define $g(n) \in \Omega(f(n))$ iff $f(n) \in \mathcal{O}(g(n))$. We say that $g(n)$ is $\Theta(f(n))$ to mean that $g(n)$ is $\mathcal{O}(f(n))$ and also $\Omega(f(n))$. This means that both functions grow, asymptotically, at the same speed, except for a constant factor.

To denote functions that are asymptotically negligible compared to $f(n)$, we use $g(n) = o(f(n))$, which means that $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$. For example, saying that a data structure uses $2n + o(n)$ bits means that it uses $2n$ plus a number of bits that grows sublinearly with n, such as $2n + \mathcal{O}(n/\log n)$. The notation $o(1)$ denotes a function that tends to zero as n tends to infinity, for example, $\log\log n/\log n = o(1)$. Finally, the opposite of the $o(\cdot)$ notation is $\omega(\cdot)$, where $g(n) = \omega(f(n))$ iff $f(n) = o(g(n))$. In particular, $\omega(1)$ denotes a function that tends to infinity (no matter how slowly) when n tends to infinity. For example, $\log\log n = \omega(1)$.

When several variables are used, as in $o(n \log \sigma)$, it must be clear to which the $o(\cdot)$ notation refers. For example, $n \log\log \sigma$ is $o(n \log \sigma)$ if the variable is σ, or if the variable is n but σ grows with n (i.e., $\sigma = \omega(1)$ as a function of n). Otherwise, if we refer to n but σ is a constant, then $n \log\log \sigma$ is not $o(n \log \sigma)$.

These notations are also used on decreasing functions of n, to describe error margins. For example, we may approximate the harmonic number $H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln$

$n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \ldots$, where $\gamma \approx 0.577$ is a constant, with any of the following formulas, having a decreasing level of detail:[1]

$$\begin{aligned} H_n &= \ln n + \gamma + \frac{1}{2n} + \mathcal{O}\left(\frac{1}{n^2}\right) \\ &= \ln n + \gamma + \mathcal{O}\left(\frac{1}{n}\right) \\ &= \ln n + \mathcal{O}(1) \\ &= \mathcal{O}(\log n), \end{aligned}$$

depending on the degree of accuracy we want. We can also use $o(\cdot)$ to give less details about the error level, for example,

$$\begin{aligned} H_n &= \ln n + \gamma + \frac{1}{2n} + o\left(\frac{1}{n}\right) \\ &= \ln n + \gamma + o(1) \\ &= \ln n + o(\log n). \end{aligned}$$

We can also write the error in relative form, for example,

$$\begin{aligned} H_n &= \ln n + \gamma + \frac{1}{2n} \cdot \left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right) \\ &= \ln n \cdot \left(1 + \mathcal{O}\left(\frac{1}{\log n}\right)\right) \\ &= \ln n \cdot (1 + o(1)). \end{aligned}$$

When using the notation to denote errors, the inequality $\frac{1}{1+x} = 1 - x + x^2 - \ldots = 1 - \mathcal{O}(x)$, for any $0 < x < 1$, allows us to write $\frac{1}{1+o(1)} = 1 + o(1)$, which is useful for moving error terms from the denominator to the numerator.

Logarithm. This is a very important function in Information Theory, as it is the key to describing the entropy, or amount of information, in an object. When the entropy (or information) is described in bits, the logarithm must be to the base 2. We use log to denote the logarithm to the base 2. When we use a logarithm to some other base b, we write $\log_b$. As shown, the natural logarithm is written as ln. Of course, the base of the logarithm makes no difference inside $\mathcal{O}$-formulas (unless it is in the exponent!).

The inequality $\frac{x}{1+x} \leq \ln(1+x) \leq x$ is useful in many cases, in particular in combination with the $\mathcal{O}$-notation. For example,

$$\ln(n(1 + o(1))) = \ln n + \ln(1 + o(1)) \leq \ln n + o(1).$$

It also holds

$$\ln(n(1 + o(1))) \geq \ln n + \frac{o(1)}{1 + o(1)} = \ln n + o(1).$$

[1] In the first line, we use the fact that the tail of the series converges to $\frac{c}{n^2}$, for some constant c.

Therefore, $\ln(n(1+o(1))) = \ln n + o(1)$. More generally, if $f(n) = o(1)$, and b is any constant, we can write $\log_b(n(1+f(n))) = \log_b n + \mathcal{O}(f(n))$. For example, $\log(n + \log n) = \log n + \mathcal{O}(\log n/n)$.

Model of computation. We consider realistic computers, with a computer word of w bits, where we can carry out in constant time all the basic arithmetic ($+$, $-$, $\cdot$, $/$, mod, ceilings and floors, etc.) and logic operations (bitwise *and*, *or*, *not*, *xor*, bit shifts, etc.). In modern computers w is almost always 32 or 64, but several architectures allow for larger words to be handled natively, reaching, for example, 128, 256, or 512 bits.

When connecting with theory, this essentially corresponds to the RAM model of computation, where we do not pay attention to restrictions in some branches of the RAM model that are unrealistic on modern computers (for example, some variants disallow multiplication and division). In the RAM model, it is usually assumed that the computer word has $w = \Theta(\log n)$ bits, where n is the size of the data in memory. This logarithmic model of growth of the computer word is appropriate in practice, as w has been growing approximately as the logarithm of the size of main memories. It is also reasonable to expect that we can store any memory address in a constant number of words (and in constant time).

For simplicity and practicality, we will use the assumption $w \geq \log n$, which means that with one computer word we can address any data element. While the assumption $w = \mathcal{O}(\log n)$ may also be justified (we may argue that the data should be large enough for the compact storage problem to be of interest), this is not always the case. For example, the dynamic structures (Chapter 12) may grow and shrink over time. Therefore, we will not rely on this assumption. Thus, for example, we will say that the cost of an algorithm that inspects n bits by chunks of w bits, processing each chunk in constant time, is $\mathcal{O}(n/w) = \mathcal{O}(n/\log n) = o(n)$. Instead, we will not take an $\mathcal{O}(w)$-time algorithm to be $\mathcal{O}(\log n)$.

Strings, sequences, and intervals. In most cases, our arrays start at position 1. With $[a, b]$ we denote the set $\{a, a+1, a+2, \ldots, b\}$, unless we explicitly imply it is a real interval. For example, $A[1, n]$ denotes an array of n elements $A[1], A[2], \ldots, A[n]$. A *string* is an array of elements drawn from a finite universe, called the *alphabet*. Alphabets are usually denoted $\Sigma = [1, \sigma]$, where σ is some integer, meaning that $\Sigma = \{1, 2, \ldots, \sigma\}$. The alphabet elements are called *symbols*, *characters*, or *letters*. The *length* of the string $S[1, n]$ is $|S| = n$. The set of all the strings of length n over alphabet Σ is denoted Σ^n, and the set of all the strings of any length over Σ is denoted $\Sigma^* = \cup_{n \geq 0} \Sigma^n$. Strings and sequences are basically synonyms in this book; however, substring and subsequence are different concepts. Given a string $S[1, n]$, a *substring* $S[i, j]$ is, precisely, the array $S[i], S[i+1], \ldots, S[j]$. Particular cases of substrings are *prefixes*, of the form $S[1, j]$, and *suffixes*, of the form $S[i, n]$. When $i > j$, $S[i, j]$ denotes the empty string ε, that is, the only string of length zero. A *subsequence* is more general than a substring: it can be any $S[i_1] \,.\, S[i_2] \ldots S[i_r]$ for $i_1 < i_2 < \ldots < i_r$, where we use the dot to denote concatenation of symbols (we might also simply write one symbol after the other, or mix strings and symbols in a concatenation). Sometimes we will also use $\langle a, b \rangle$ to denote the same as $[a, b]$ or write sequences as $\langle a_1, a_2, \ldots, a_n \rangle$. Finally, given a string $S[1, n]$, S^{rev} denotes the reversed string, $S[n] \,.\, S[n-1] \ldots S[2] \,.\, S[1]$.

1.6 Bibliographic Notes

Growth of information and computing power. Google's mission is stated in http://www.google.com/about/company.

There are many sources that describe the amount of information the world is gathering. For example, a 2011 study from *International Data Corporation (IDC)* found that we are generating a few zettabytes per year (a zettabyte is 2^{70}, or roughly 10^{21}, bytes), and that data are more than doubling per year, outperforming Moore's law (which governs the growth of hardware capacities).[2] A related discussion from 2013, arguing that we are much better at storing than at using all these data, can be read in *Datamation*.[3] For a shocking and graphical message, the 2012 poster of *Domo* is also telling.[4]

There are also many sources about the differences in performance between CPU, caches, main memory, and secondary storage, as well as how these have evolved over the years. In particular, we used the book of Hennessy and Patterson (2012, Chap. 1) for the rough numbers shown here.

Examples of books about the mentioned algorithmic approaches to solve the problem of data growth are, among many others, Vitter (2008) for secondary-memory algorithms, Muthukrishnan (2005) for streaming algorithms, and Roosta (1999) for distributed algorithms.

Suffix trees. The book by Gusfield (1997) provides a good introduction to suffix trees in the context of bioinformatics. Modern books pay more attention to space issues and make use of some of the compact data structures we describe here (Ohlebusch, 2013; Mäkinen *et al.*, 2015). Our size estimates for compressed suffix trees are taken from the Ph.D. thesis of Gog (2011).

Compact data structures. Despite some previous isolated results, the Ph.D. thesis of Jacobson (1988) is generally taken as the starting point of the systematic study of compact data structures. Jacobson coined the term *succinct data structure* to denote a data structure that uses $\log N + o(\log N)$ bits, where N is the total number of different objects that can be encoded. For example, succinct data structures for arrays of n bits must use $n + o(n)$ bits, since $N = 2^n$. To exclude mere data compressors, succinct data structures are sometimes required to support queries in constant time (Munro, 1996).

In this book we use the term *compact data structure*, which refers to the broader class of data structures that aim at using little space and query time. Other related terms are used in the literature (not always consistently) to refer to particular subclasses of data structures (Ferragina and Manzini, 2005; Gál and Miltersen, 2007; Fischer and Heun, 2011; Raman, 2015): *compressed* or *opportunistic* data structures are those using $\mathcal{H} + o(\log N)$ bits, where $\mathcal{H}$ is the entropy of the data under some compression model (such as the bit array representations we describe in Section 4.1.1); data structures using $\mathcal{H} + o(\mathcal{H})$ bits are sometimes called *fully compressed* (for example, the Huffman-shaped wavelet trees of Section 6.2.4 are almost fully compressed). A data structure that adds

[2] http://www.emc.com/about/news/press/2011/20110628-01.htm.
[3] http://www.datamation.com/applications/big-data-analytics-overview.html.
[4] http://www.domo.com/blog/2012/06/how-much-data-is-created-every-minute.