Chapter 1

Introduction

Every computer science curriculum in the world includes a course on data structures and algorithms. Data structures are *that* important; they improve our quality of life and even save lives on a regular basis. Many multi-million and several multi-billion dollar companies have been built around data structures.

How can this be? If we stop to think about it, we realize that we interact with data structures constantly.

- Open a file: File system data structures are used to locate the parts of that file on disk so they can be retrieved. This isn't easy; disks contain hundreds of millions of blocks. The contents of your file could be stored on any one of them.
- Look up a contact on your phone: A data structure is used to look up a phone number in your contact list based on partial information even before you finish dialing/typing. This isn't easy; your phone may contain information about a lot of people—everyone you have ever contacted via phone or email—and your phone doesn't have a very fast processor or a lot of memory.
- Log in to your favourite social network: The network servers use your login information to look up your account information. This isn't easy; the most popular social networks have hundreds of millions of active users.
- Do a web search: The search engine uses data structures to find the web pages containing your search terms. This isn't easy; there are

over 8.5 billion web pages on the Internet and each page contains a lot of potential search terms.

- Phone emergency services (9-1-1): The emergency services network looks up your phone number in a data structure that maps phone numbers to addresses so that police cars, ambulances, or fire trucks can be sent there without delay. This is important; the person making the call may not be able to provide the exact address they are calling from and a delay can mean the difference between life or death.

1.1 The Need for Efficiency

In the next section, we look at the operations supported by the most commonly used data structures. Anyone with a bit of programming experience will see that these operations are not hard to implement correctly. We can store the data in an array or a linked list and each operation can be implemented by iterating over all the elements of the array or list and possibly adding or removing an element.

This kind of implementation is easy, but not very efficient. Does this really matter? Computers are becoming faster and faster. Maybe the obvious implementation is good enough. Let's do some rough calculations to find out.

Number of operations: Imagine an application with a moderately-sized data set, say of one million (10^6), items. It is reasonable, in most applications, to assume that the application will want to look up each item at least once. This means we can expect to do at least one million (10^6) searches in this data. If each of these 10^6 searches inspects each of the 10^6 items, this gives a total of $10^6 \times 10^6 = 10^{12}$ (one thousand billion) inspections.

Processor speeds: At the time of writing, even a very fast desktop computer can not do more than one billion (10^9) operations per second.[1] This

[1] Computer speeds are at most a few gigahertz (billions of cycles per second), and each operation typically takes a few cycles.

means that this application will take at least $10^{12}/10^9 = 1000$ seconds, or roughly 16 minutes and 40 seconds. Sixteen minutes is an eon in computer time, but a person might be willing to put up with it (if he or she were headed out for a coffee break).

Bigger data sets: Now consider a company like Google, that indexes over 8.5 billion web pages. By our calculations, doing any kind of query over this data would take at least 8.5 seconds. We already know that this isn't the case; web searches complete in much less than 8.5 seconds, and they do much more complicated queries than just asking if a particular page is in their list of indexed pages. At the time of writing, Google receives approximately $4,500$ queries per second, meaning that they would require at least $4,500 \times 8.5 = 38,250$ very fast servers just to keep up.

The solution: These examples tell us that the obvious implementations of data structures do not scale well when the number of items, n, in the data structure and the number of operations, m, performed on the data structure are both large. In these cases, the time (measured in, say, machine instructions) is roughly $n \times m$.

The solution, of course, is to carefully organize data within the data structure so that not every operation requires every data item to be inspected. Although it sounds impossible at first, we will see data structures where a search requires looking at only two items on average, independent of the number of items stored in the data structure. In our billion instruction per second computer it takes only 0.000000002 seconds to search in a data structure containing a billion items (or a trillion, or a quadrillion, or even a quintillion items).

We will also see implementations of data structures that keep the items in sorted order, where the number of items inspected during an operation grows very slowly as a function of the number of items in the data structure. For example, we can maintain a sorted set of one billion items while inspecting at most 60 items during any operation. In our billion instruction per second computer, these operations take 0.00000006 seconds each.

The remainder of this chapter briefly reviews some of the main concepts used throughout the rest of the book. Section 1.2 describes the in-

terfaces implemented by all of the data structures described in this book and should be considered required reading. The remaining sections discuss:

- some mathematical review including exponentials, logarithms, factorials, asymptotic (big-Oh) notation, probability, and randomization;
- the model of computation;
- correctness, running time, and space;
- an overview of the rest of the chapters; and
- the sample code and typesetting conventions.

A reader with or without a background in these areas can easily skip them now and come back to them later if necessary.

1.2 Interfaces

When discussing data structures, it is important to understand the difference between a data structure's interface and its implementation. An interface describes what a data structure does, while an implementation describes how the data structure does it.

An *interface*, sometimes also called an *abstract data type*, defines the set of operations supported by a data structure and the semantics, or meaning, of those operations. An interface tells us nothing about how the data structure implements these operations; it only provides a list of supported operations along with specifications about what types of arguments each operation accepts and the value returned by each operation.

A data structure *implementation*, on the other hand, includes the internal representation of the data structure as well as the definitions of the algorithms that implement the operations supported by the data structure. Thus, there can be many implementations of a single interface. For example, in Chapter 2, we will see implementations of the `List` interface using arrays and in Chapter 3 we will see implementations of the `List` interface using pointer-based data structures. Each implements the same interface, `List`, but in different ways.

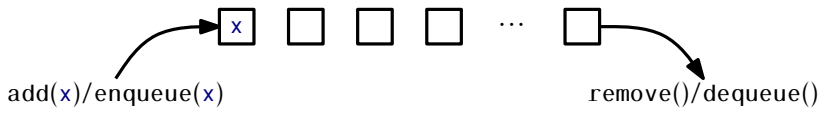

Figure 1.1: A FIFO Queue.

1.2.1 The Queue, Stack, and Deque Interfaces

The Queue interface represents a collection of elements to which we can add elements and remove the next element. More precisely, the operations supported by the Queue interface are

- add(x): add the value x to the Queue
- remove(): remove the next (previously added) value, y, from the Queue and return y

Notice that the remove() operation takes no argument. The Queue's *queueing discipline* decides which element should be removed. There are many possible queueing disciplines, the most common of which include FIFO, priority, and LIFO.

A *FIFO (first-in-first-out) Queue*, which is illustrated in Figure 1.1, removes items in the same order they were added, much in the same way a queue (or line-up) works when checking out at a cash register in a grocery store. This is the most common kind of Queue so the qualifier FIFO is often omitted. In other texts, the add(x) and remove() operations on a FIFO Queue are often called enqueue(x) and dequeue(), respectively.

A *priority Queue*, illustrated in Figure 1.2, always removes the smallest element from the Queue, breaking ties arbitrarily. This is similar to the way in which patients are triaged in a hospital emergency room. As patients arrive they are evaluated and then placed in a waiting room. When a doctor becomes available he or she first treats the patient with the most life-threatening condition. The remove(x) operation on a priority Queue is usually called deleteMin() in other texts.

A very common queueing discipline is the LIFO (last-in-first-out) discipline, illustrated in Figure 1.3. In a *LIFO Queue*, the most recently added element is the next one removed. This is best visualized in terms of a stack of plates; plates are placed on the top of the stack and also

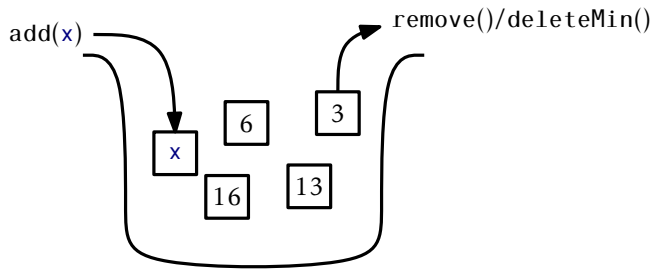

Figure 1.2: A priority Queue.

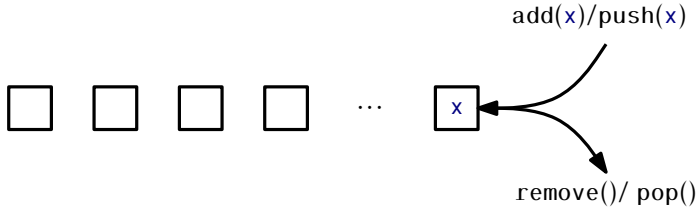

Figure 1.3: A stack.

removed from the top of the stack. This structure is so common that it gets its own name: Stack. Often, when discussing a Stack, the names of add(x) and remove() are changed to push(x) and pop(); this is to avoid confusing the LIFO and FIFO queueing disciplines.

A Deque is a generalization of both the FIFO Queue and LIFO Queue (Stack). A Deque represents a sequence of elements, with a front and a back. Elements can be added at the front of the sequence or the back of the sequence. The names of the Deque operations are self-explanatory: addFirst(x), removeFirst(), addLast(x), and removeLast(). It is worth noting that a Stack can be implemented using only addFirst(x) and removeFirst() while a FIFO Queue can be implemented using addLast(x) and removeFirst().

1.2.2 The List Interface: Linear Sequences

This book will talk very little about the FIFO Queue, Stack, or Deque interfaces. This is because these interfaces are subsumed by the List interface. A List, illustrated in Figure 1.4, represents a sequence, $x_0, \ldots, x_{n-1}$,

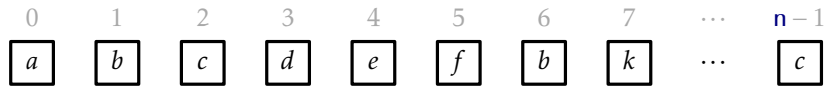

Figure 1.4: A `List` represents a sequence indexed by $0, 1, 2, \ldots, n$. In this `List` a call to `get(2)` would return the value c.

of values. The `List` interface includes the following operations:

1. `size()`: return n, the length of the list
2. `get(i)`: return the value x_i
3. `set(i,x)`: set the value of x_i equal to x
4. `add(i,x)`: add x at position i, displacing $x_i, \ldots, x_{n-1}$;
   Set $x_{j+1} = x_j$, for all $j \in \{n-1, \ldots, i\}$, increment n, and set $x_i = x$
5. `remove(i)` remove the value x_i, displacing $x_{i+1}, \ldots, x_{n-1}$;
   Set $x_j = x_{j+1}$, for all $j \in \{i, \ldots, n-2\}$ and decrement n

Notice that these operations are easily sufficient to implement the `Deque` interface:

$$\begin{aligned} \texttt{addFirst(x)} &\Rightarrow \texttt{add(0,x)} \\ \texttt{removeFirst()} &\Rightarrow \texttt{remove(0)} \\ \texttt{addLast(x)} &\Rightarrow \texttt{add(size(),x)} \\ \texttt{removeLast()} &\Rightarrow \texttt{remove(size()-1)} \end{aligned}$$

Although we will normally not discuss the `Stack`, `Deque` and FIFO `Queue` interfaces in subsequent chapters, the terms `Stack` and `Deque` are sometimes used in the names of data structures that implement the `List` interface. When this happens, it highlights the fact that these data structures can be used to implement the `Stack` or `Deque` interface very efficiently. For example, the `ArrayDeque` class is an implementation of the `List` interface that implements all the `Deque` operations in constant time per operation.

1.2.3 The USet Interface: Unordered Sets

The USet interface represents an unordered set of unique elements, which mimics a mathematical *set*. A USet contains n *distinct* elements; no element appears more than once; the elements are in no specific order. A USet supports the following operations:

1. size(): return the number, n, of elements in the set

2. add(x): add the element x to the set if not already present;
   Add x to the set provided that there is no element y in the set such that x equals y. Return true if x was added to the set and false otherwise.

3. remove(x): remove x from the set;
   Find an element y in the set such that x equals y and remove y. Return y, or null if no such element exists.

4. find(x): find x in the set if it exists;
   Find an element y in the set such that y equals x. Return y, or null if no such element exists.

These definitions are a bit fussy about distinguishing x, the element we are removing or finding, from y, the element we may remove or find. This is because x and y might actually be distinct objects that are nevertheless treated as equal.[2] Such a distinction is useful because it allows for the creation of *dictionaries* or *maps* that map keys onto values.

To create a dictionary/map, one forms compound objects called Pairs, each of which contains a *key* and a *value*. Two Pairs are treated as equal if their keys are equal. If we store some pair (k, v) in a USet and then later call the find(x) method using the pair x = (k, null) the result will be y = (k, v). In other words, it is possible to recover the value, v, given only the key, k.

[2] In Java, this is done by overriding the class's equals(y) and hashCode() methods.

1.2.4 The SSet Interface: Sorted Sets

The SSet interface represents a sorted set of elements. An SSet stores elements from some total order, so that any two elements x and y can be compared. In code examples, this will be done with a method called compare(x,y) in which

$$\texttt{compare}(x,y)\begin{cases} <0 & \text{if } x<y \\ >0 & \text{if } x>y \\ =0 & \text{if } x=y \end{cases}$$

An SSet supports the size(), add(x), and remove(x) methods with exactly the same semantics as in the USet interface. The difference between a USet and an SSet is in the find(x) method:

4. find(x): locate x in the sorted set;
   Find the smallest element y in the set such that $y \geq x$. Return y or null if no such element exists.

This version of the find(x) operation is sometimes referred to as a *successor search*. It differs in a fundamental way from USet.find(x) since it returns a meaningful result even when there is no element equal to x in the set.

The distinction between the USet and SSet find(x) operations is very important and often missed. The extra functionality provided by an SSet usually comes with a price that includes both a larger running time and a higher implementation complexity. For example, most of the SSet implementations discussed in this book all have find(x) operations with running times that are logarithmic in the size of the set. On the other hand, the implementation of a USet as a ChainedHashTable in Chapter 5 has a find(x) operation that runs in constant expected time. When choosing which of these structures to use, one should always use a USet unless the extra functionality offered by an SSet is truly needed.

1.3 Mathematical Background

In this section, we review some mathematical notations and tools used throughout this book, including logarithms, big-Oh notation, and proba-

bility theory. This review will be brief and is not intended as an introduction. Readers who feel they are missing this background are encouraged to read, and do exercises from, the appropriate sections of the very good (and free) textbook on mathematics for computer science [50].

1.3.1 Exponentials and Logarithms

The expression b^x denotes the number b raised to the power of x. If x is a positive integer, then this is just the value of b multiplied by itself $x-1$ times:

$$b^x = \underbrace{b \times b \times \cdots \times b}_{x} \ .$$

When x is a negative integer, $b^x = 1/b^{-x}$. When $x = 0$, $b^x = 1$. When b is not an integer, we can still define exponentiation in terms of the exponential function e^x (see below), which is itself defined in terms of the exponential series, but this is best left to a calculus text.

In this book, the expression $\log_b k$ denotes the *base-b logarithm* of k. That is, the unique value x that satisfies

$$b^x = k \ .$$

Most of the logarithms in this book are base 2 (*binary logarithms*). For these, we omit the base, so that $\log k$ is shorthand for $\log_2 k$.

An informal, but useful, way to think about logarithms is to think of $\log_b k$ as the number of times we have to divide k by b before the result is less than or equal to 1. For example, when one does binary search, each comparison reduces the number of possible answers by a factor of 2. This is repeated until there is at most one possible answer. Therefore, the number of comparison done by binary search when there are initially at most $n+1$ possible answers is at most $\lceil \log_2(n+1) \rceil$.

Another logarithm that comes up several times in this book is the *natural logarithm*. Here we use the notation $\ln k$ to denote $\log_e k$, where e — *Euler's constant* — is given by

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \approx 2.71828 \ .$$

The natural logarithm comes up frequently because it is the value of a particularly common integral:

$$\int_1^k 1/x \, \mathrm{d}x = \ln k \ .$$

Two of the most common manipulations we do with logarithms are removing them from an exponent:

$$b^{\log_b k} = k$$

and changing the base of a logarithm:

$$\log_b k = \frac{\log_a k}{\log_a b} \ .$$

For example, we can use these two manipulations to compare the natural and binary logarithms

$$\ln k = \frac{\log k}{\log e} = \frac{\log k}{(\ln e)/(\ln 2)} = (\ln 2)(\log k) \approx 0.693147 \log k \ .$$

1.3.2 Factorials

In one or two places in this book, the *factorial* function is used. For a non-negative integer n, the notation $n!$ (pronounced "n factorial") is defined to mean

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \ .$$

Factorials appear because $n!$ counts the number of distinct permutations, i.e., orderings, of n distinct elements. For the special case $n = 0$, $0!$ is defined as 1.

The quantity $n!$ can be approximated using *Stirling's Approximation*:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha(n)} \ ,$$

where

$$\frac{1}{12n+1} < \alpha(n) < \frac{1}{12n} \ .$$

Stirling's Approximation also approximates $\ln(n!)$:

$$\ln(n!) = n \ln n - n + \frac{1}{2}\ln(2\pi n) + \alpha(n)$$

(In fact, Stirling's Approximation is most easily proven by approximating $\ln(n!) = \ln 1 + \ln 2 + \cdots + \ln n$ by the integral $\int_1^n \ln n \, \mathrm{d}n = n \ln n - n + 1$.)

Related to the factorial function are the *binomial coefficients*. For a non-negative integer n and an integer $k \in \{0, \ldots, n\}$, the notation $\binom{n}{k}$ denotes:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \ .$$

The binomial coefficient $\binom{n}{k}$ (pronounced "n choose k") counts the number of subsets of an n element set that have size k, i.e., the number of ways of choosing k distinct integers from the set $\{1, \ldots, n\}$.

1.3.3 Asymptotic Notation

When analyzing data structures in this book, we want to talk about the running times of various operations. The exact running times will, of course, vary from computer to computer and even from run to run on an individual computer. When we talk about the running time of an operation we are referring to the number of computer instructions performed during the operation. Even for simple code, this quantity can be difficult to compute exactly. Therefore, instead of analyzing running times exactly, we will use the so-called *big-Oh notation*: For a function $f(n)$, $O(f(n))$ denotes a set of functions,

$$O(f(n)) = \left\{ \begin{array}{c} g(n) : \text{there exists } c > 0, \text{ and } n_0 \text{ such that} \\ g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0 \end{array} \right\} \ .$$

Thinking graphically, this set consists of the functions $g(n)$ where $c \cdot f(n)$ starts to dominate $g(n)$ when n is sufficiently large.

We generally use asymptotic notation to simplify functions. For example, in place of $5n \log n + 8n - 200$ we can write $O(n \log n)$. This is proven as follows:

$$\begin{aligned} 5n \log n + 8n - 200 &\leq 5n \log n + 8n \\ &\leq 5n \log n + 8n \log n \qquad \text{for } n \geq 2 \text{ (so that } \log n \geq 1) \\ &\leq 13n \log n \ . \end{aligned}$$

This demonstrates that the function $f(n) = 5n \log n + 8n - 200$ is in the set $O(n \log n)$ using the constants $c = 13$ and $n_0 = 2$.

A number of useful shortcuts can be applied when using asymptotic notation. First:

$$O(n^{c_1}) \subset O(n^{c_2}) \ ,$$

for any $c_1 < c_2$. Second: For any constants $a, b, c > 0$,

$$O(a) \subset O(\log n) \subset O(n^b) \subset O(c^n) \ .$$

These inclusion relations can be multiplied by any positive value, and they still hold. For example, multiplying by n yields:

$$O(n) \subset O(n \log n) \subset O(n^{1+b}) \subset O(nc^n) \ .$$

Continuing in a long and distinguished tradition, we will abuse this notation by writing things like $f_1(n) = O(f(n))$ when what we really mean is $f_1(n) \in O(f(n))$. We will also make statements like "the running time of this operation is $O(f(n))$" when this statement should be "the running time of this operation is *a member of* $O(f(n))$." These shortcuts are mainly to avoid awkward language and to make it easier to use asymptotic notation within strings of equations.

A particularly strange example of this occurs when we write statements like

$$T(n) = 2 \log n + O(1) \ .$$

Again, this would be more correctly written as

$$T(n) \leq 2 \log n + [\text{some member of } O(1)] \ .$$

The expression $O(1)$ also brings up another issue. Since there is no variable in this expression, it may not be clear which variable is getting arbitrarily large. Without context, there is no way to tell. In the example above, since the only variable in the rest of the equation is n, we can assume that this should be read as $T(n) = 2 \log n + O(f(n))$, where $f(n) = 1$.

Big-Oh notation is not new or unique to computer science. It was used by the number theorist Paul Bachmann as early as 1894, and is immensely useful for describing the running times of computer algorithms. Consider the following piece of code:

Simple

```
void snippet() {
  for (int i = 0; i < n; i++)
    a[i] = i;
}
```

One execution of this method involves

- 1 assignment (`int i = 0`),
- `n` + 1 comparisons (`i < n`),
- `n` increments (`i++`),
- `n` array offset calculations (`a[i]`), and
- `n` indirect assignments (`a[i] = i`).

So we could write this running time as

$$T(n) = a + b(n+1) + cn + dn + en \ ,$$

where a, b, c, d, and e are constants that depend on the machine running the code and represent the time to perform assignments, comparisons, increment operations, array offset calculations, and indirect assignments, respectively. However, if this expression represents the running time of two lines of code, then clearly this kind of analysis will not be tractable to complicated code or algorithms. Using big-Oh notation, the running time can be simplified to

$$T(n) = O(n) \ .$$

Not only is this more compact, but it also gives nearly as much information. The fact that the running time depends on the constants a, b, c, d, and e in the above example means that, in general, it will not be possible to compare two running times to know which is faster without knowing the values of these constants. Even if we make the effort to determine these constants (say, through timing tests), then our conclusion will only be valid for the machine we run our tests on.

Big-Oh notation allows us to reason at a much higher level, making it possible to analyze more complicated functions. If two algorithms have

the same big-Oh running time, then we won't know which is faster, and there may not be a clear winner. One may be faster on one machine, and the other may be faster on a different machine. However, if the two algorithms have demonstrably different big-Oh running times, then we can be certain that the one with the smaller running time will be faster *for large enough values of* n.

An example of how big-Oh notation allows us to compare two different functions is shown in Figure 1.5, which compares the rate of grown of $f_1(\mathsf{n}) = 15\mathsf{n}$ versus $f_2(n) = 2\mathsf{n}\log\mathsf{n}$. It might be that $f_1(n)$ is the running time of a complicated linear time algorithm while $f_2(n)$ is the running time of a considerably simpler algorithm based on the divide-and-conquer paradigm. This illustrates that, although $f_1(\mathsf{n})$ is greater than $f_2(n)$ for small values of n, the opposite is true for large values of n. Eventually $f_1(\mathsf{n})$ wins out, by an increasingly wide margin. Analysis using big-Oh notation told us that this would happen, since $O(\mathsf{n}) \subset O(\mathsf{n}\log\mathsf{n})$.

In a few cases, we will use asymptotic notation on functions with more than one variable. There seems to be no standard for this, but for our purposes, the following definition is sufficient:

$$O(f(n_1,\ldots,n_k)) = \left\{ \begin{array}{l} g(n_1,\ldots,n_k) : \text{there exists } c > 0, \text{ and } z \text{ such that} \\ \quad g(n_1,\ldots,n_k) \leq c \cdot f(n_1,\ldots,n_k) \\ \quad\quad \text{for all } n_1,\ldots,n_k \text{ such that } g(n_1,\ldots,n_k) \geq z \end{array} \right\}.$$

This definition captures the situation we really care about: when the arguments $n_1,\ldots,n_k$ make g take on large values. This definition also agrees with the univariate definition of $O(f(n))$ when $f(n)$ is an increasing function of n. The reader should be warned that, although this works for our purposes, other texts may treat multivariate functions and asymptotic notation differently.

1.3.4 Randomization and Probability

Some of the data structures presented in this book are *randomized*; they make random choices that are independent of the data being stored in them or the operations being performed on them. For this reason, performing the same set of operations more than once using these structures could result in different running times. When analyzing these data struc-

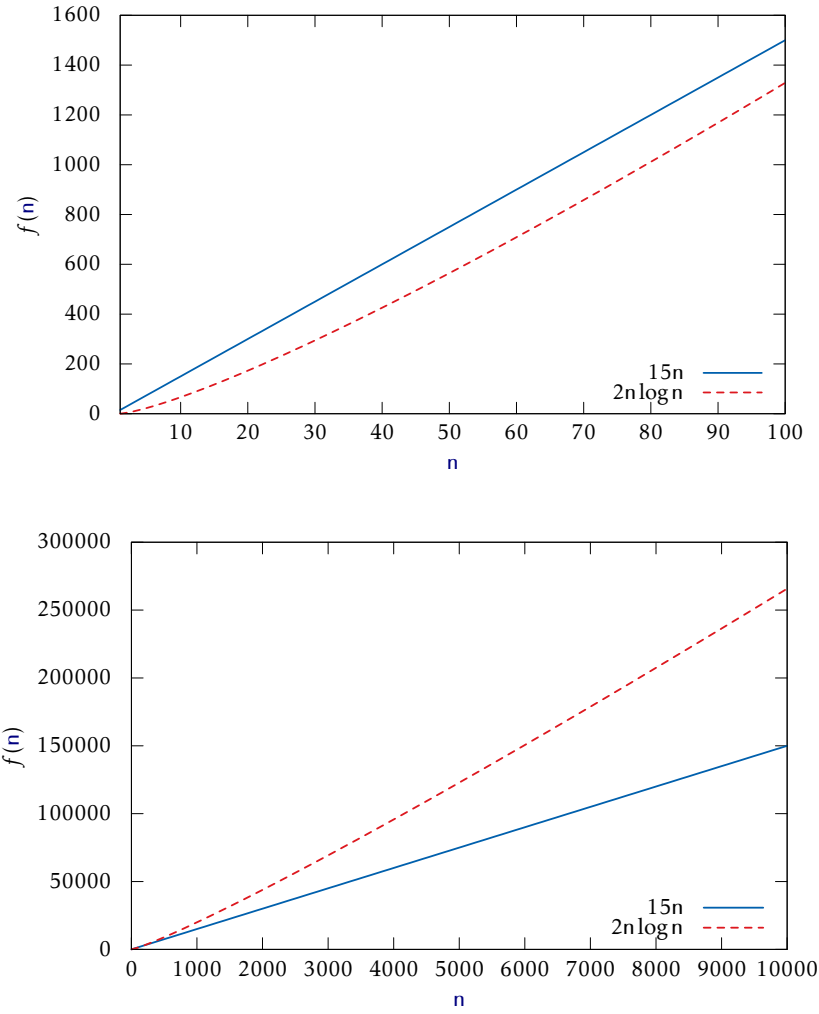

Figure 1.5: Plots of $15n$ versus $2n \log n$.

tures we are interested in their average or *expected* running times.

Formally, the running time of an operation on a randomized data structure is a random variable, and we want to study its *expected value*. For a discrete random variable X taking on values in some countable universe U, the expected value of X, denoted by $\mathrm{E}[X]$, is given by the formula

$$\mathrm{E}[X] = \sum_{x \in U} x \cdot \Pr\{X = x\} \ .$$

Here $\Pr\{\mathcal{E}\}$ denotes the probability that the event $\mathcal{E}$ occurs. In all of the examples in this book, these probabilities are only with respect to the random choices made by the randomized data structure; there is no assumption that the data stored in the structure, nor the sequence of operations performed on the data structure, is random.

One of the most important properties of expected values is *linearity of expectation*. For any two random variables X and Y,

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y] \ .$$

More generally, for any random variables $X_1, \ldots, X_k$,

$$\mathrm{E}\left[\sum_{i=1}^{k} X_k\right] = \sum_{i=1}^{k} \mathrm{E}[X_i] \ .$$

Linearity of expectation allows us to break down complicated random variables (like the left hand sides of the above equations) into sums of simpler random variables (the right hand sides).

A useful trick, that we will use repeatedly, is defining *indicator random variables*. These binary variables are useful when we want to count something and are best illustrated by an example. Suppose we toss a fair coin k times and we want to know the expected number of times the coin turns up as heads. Intuitively, we know the answer is $k/2$, but if we try to prove it using the definition of expected value, we get

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=0}^{k} i \cdot \Pr\{X = i\} \\ &= \sum_{i=0}^{k} i \cdot \binom{k}{i} / 2^k \end{aligned}$$

$$= k \cdot \sum_{i=0}^{k-1} \binom{k-1}{i} / 2^k$$
$$= k/2 \ .$$

This requires that we know enough to calculate that $\Pr\{X = i\} = \binom{k}{i}/2^k$, and that we know the binomial identities $i\binom{k}{i} = k\binom{k-1}{i}$ and $\sum_{i=0}^{k}\binom{k}{i} = 2^k$.

Using indicator variables and linearity of expectation makes things much easier. For each $i \in \{1, \ldots, k\}$, define the indicator random variable

$$I_i = \begin{cases} 1 & \text{if the } i\text{th coin toss is heads} \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$\mathrm{E}[I_i] = (1/2)1 + (1/2)0 = 1/2 \ .$$

Now, $X = \sum_{i=1}^{k} I_i$, so

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k} I_i\right] \\ &= \sum_{i=1}^{k} \mathrm{E}[I_i] \\ &= \sum_{i=1}^{k} 1/2 \\ &= k/2 \ . \end{aligned}$$

This is a bit more long-winded, but doesn't require that we know any magical identities or compute any non-trivial probabilities. Even better, it agrees with the intuition that we expect half the coins to turn up as heads precisely because each individual coin turns up as heads with a probability of 1/2.

1.4 The Model of Computation

In this book, we will analyze the theoretical running times of operations on the data structures we study. To do this precisely, we need a mathematical model of computation. For this, we use the w-*bit word-RAM* model.

RAM stands for Random Access Machine. In this model, we have access to a random access memory consisting of *cells*, each of which stores a w-bit *word*. This implies that a memory cell can represent, for example, any integer in the set $\{0,\ldots,2^w-1\}$.

In the word-RAM model, basic operations on words take constant time. This includes arithmetic operations ($+$, $-$, $*$, $/$, $\%$), comparisons ($<$, $>$, $=$, $\leq$, $\geq$), and bitwise boolean operations (bitwise-AND, OR, and exclusive-OR).

Any cell can be read or written in constant time. A computer's memory is managed by a memory management system from which we can allocate or deallocate a block of memory of any size we would like. Allocating a block of memory of size k takes $O(k)$ time and returns a reference (a pointer) to the newly-allocated memory block. This reference is small enough to be represented by a single word.

The word-size w is a very important parameter of this model. The only assumption we will make about w is the lower-bound $w \geq \log n$, where n is the number of elements stored in any of our data structures. This is a fairly modest assumption, since otherwise a word is not even big enough to count the number of elements stored in the data structure.

Space is measured in words, so that when we talk about the amount of space used by a data structure, we are referring to the number of words of memory used by the structure. All of our data structures store values of a generic type T, and we assume an element of type T occupies one word of memory. (In reality, we are storing references to objects of type T, and these references occupy only one word of memory.)

The w-bit word-RAM model is a fairly close match for the (32-bit) Java Virtual Machine (JVM) when $w = 32$. The data structures presented in this book don't use any special tricks that are not implementable on the JVM and most other architectures.

1.5 Correctness, Time Complexity, and Space Complexity

When studying the performance of a data structure, there are three things that matter most:

Correctness: The data structure should correctly implement its interface.

Time complexity: The running times of operations on the data structure should be as small as possible.

Space complexity: The data structure should use as little memory as possible.

In this introductory text, we will take correctness as a given; we won't consider data structures that give incorrect answers to queries or don't perform updates properly. We will, however, see data structures that make an extra effort to keep space usage to a minimum. This won't usually affect the (asymptotic) running times of operations, but can make the data structures a little slower in practice.

When studying running times in the context of data structures we tend to come across three different kinds of running time guarantees:

Worst-case running times: These are the strongest kind of running time guarantees. If a data structure operation has a worst-case running time of $f(n)$, then one of these operations *never* takes longer than $f(n)$ time.

Amortized running times: If we say that the amortized running time of an operation in a data structure is $f(n)$, then this means that the cost of a typical operation is at most $f(n)$. More precisely, if a data structure has an amortized running time of $f(n)$, then a sequence of m operations takes at most $mf(n)$ time. Some individual operations may take more than $f(n)$ time but the average, over the entire sequence of operations, is at most $f(n)$.

Expected running times: If we say that the expected running time of an operation on a data structure is $f(n)$, this means that the actual running time is a random variable (see Section 1.3.4) and the expected value of this random variable is at most $f(n)$. The randomization here is with respect to random choices made by the data structure.

To understand the difference between worst-case, amortized, and expected running times, it helps to consider a financial example. Consider the cost of buying a house:

Worst-case versus amortized cost: Suppose that a home costs $120 000. In order to buy this home, we might get a 120 month (10 year) mortgage with monthly payments of $1 200 per month. In this case, the worst-case monthly cost of paying this mortgage is $1 200 per month.

If we have enough cash on hand, we might choose to buy the house outright, with one payment of $120 000. In this case, over a period of 10 years, the amortized monthly cost of buying this house is

$$\$120\,000/120 \text{ months} = \$1\,000 \text{ per month} \ .$$

This is much less than the $1 200 per month we would have to pay if we took out a mortgage.

Worst-case versus expected cost: Next, consider the issue of fire insurance on our $120 000 home. By studying hundreds of thousands of cases, insurance companies have determined that the expected amount of fire damage caused to a home like ours is $10 per month. This is a very small number, since most homes never have fires, a few homes may have some small fires that cause a bit of smoke damage, and a tiny number of homes burn right to their foundations. Based on this information, the insurance company charges $15 per month for fire insurance.

Now it's decision time. Should we pay the $15 worst-case monthly cost for fire insurance, or should we gamble and self-insure at an expected cost of $10 per month? Clearly, the $10 per month costs less *in expectation*, but we have to be able to accept the possibility that the *actual cost* may be much higher. In the unlikely event that the entire house burns down, the actual cost will be $120 000.

These financial examples also offer insight into why we sometimes settle for an amortized or expected running time over a worst-case running time. It is often possible to get a lower expected or amortized running time than a worst-case running time. At the very least, it is very often possible to get a much simpler data structure if one is willing to settle for amortized or expected running times.

1.6 Code Samples

The code samples in this book are written in the Java programming language. However, to make the book accessible to readers not familiar with all of Java's constructs and keywords, the code samples have been simplified. For example, a reader won't find any of the keywords `public`, `protected`, `private`, or `static`. A reader also won't find much discussion about class hierarchies. Which interfaces a particular class implements or which class it extends, if relevant to the discussion, should be clear from the accompanying text.

These conventions should make the code samples understandable by anyone with a background in any of the languages from the ALGOL tradition, including B, C, C++, C#, Objective-C, D, Java, JavaScript, and so on. Readers who want the full details of all implementations are encouraged to look at the Java source code that accompanies this book.

This book mixes mathematical analyses of running times with Java source code for the algorithms being analyzed. This means that some equations contain variables also found in the source code. These variables are typeset consistently, both within the source code and within equations. The most common such variable is the variable `n` that, without exception, always refers to the number of items currently stored in the data structure.

1.7 List of Data Structures

Tables 1.1 and 1.2 summarize the performance of data structures in this book that implement each of the interfaces, `List`, `USet`, and `SSet`, described in Section 1.2. Figure 1.6 shows the dependencies between various chapters in this book. A dashed arrow indicates only a weak dependency, in which only a small part of the chapter depends on a previous chapter or only the main results of the previous chapter.

List implementations			
	get(i)/set(i,x)	add(i,x)/remove(i)	
ArrayStack	$O(1)$	$O(1+n-i)^{A}$	§ 2.1
ArrayDeque	$O(1)$	$O(1+\min\{i,n-i\})^{A}$	§ 2.4
DualArrayDeque	$O(1)$	$O(1+\min\{i,n-i\})^{A}$	§ 2.5
RootishArrayStack	$O(1)$	$O(1+n-i)^{A}$	§ 2.6
DLList	$O(1+\min\{i,n-i\})$	$O(1+\min\{i,n-i\})$	§ 3.2
SEList	$O(1+\min\{i,n-i\}/b)$	$O(b+\min\{i,n-i\}/b)^{A}$	§ 3.3
SkiplistList	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 4.3

USet implementations			
	find(x)	add(x)/remove(x)	
ChainedHashTable	$O(1)^{E}$	$O(1)^{A,E}$	§ 5.1
LinearHashTable	$O(1)^{E}$	$O(1)^{A,E}$	§ 5.2

^{A} Denotes an *amortized* running time.
^{E} Denotes an *expected* running time.

Table 1.1: Summary of List and USet implementations.

SSet implementations			
	find(x)	add(x)/remove(x)	
SkiplistSSet	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 4.2
Treap	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 7.2
ScapegoatTree	$O(\log n)$	$O(\log n)^{A}$	§ 8.1
RedBlackTree	$O(\log n)$	$O(\log n)$	§ 9.2
BinaryTrie^{I}	$O(w)$	$O(w)$	§ 13.1
XFastTrie^{I}	$O(\log w)^{A,E}$	$O(w)^{A,E}$	§ 13.2
YFastTrie^{I}	$O(\log w)^{A,E}$	$O(\log w)^{A,E}$	§ 13.3
BTree	$O(\log n)$	$O(B+\log n)^{A}$	§ 14.2
BTree^{X}	$O(\log_B n)$	$O(\log_B n)$	§ 14.2

(Priority) Queue implementations			
	findMin()	add(x)/remove()	
BinaryHeap	$O(1)$	$O(\log n)^{A}$	§ 10.1
MeldableHeap	$O(1)$	$O(\log n)^{E}$	§ 10.2

[I] This structure can only store w-bit integer data.
[X] This denotes the running time in the external-memory model; see Chapter 14.

Table 1.2: Summary of SSet and priority Queue implementations.

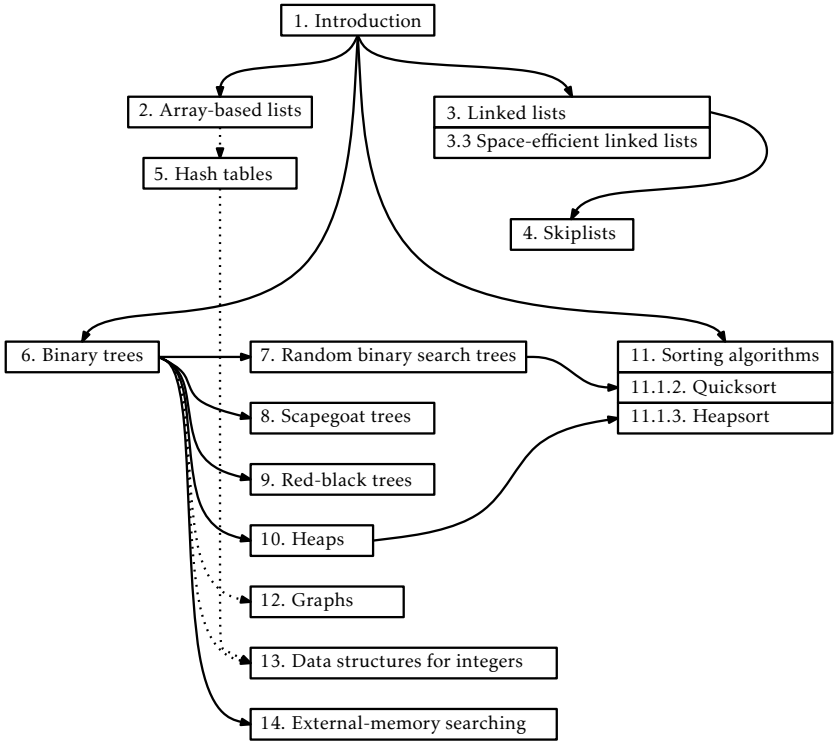

Figure 1.6: The dependencies between chapters in this book.

1.8 Discussion and Exercises

The `List`, `USet`, and `SSet` interfaces described in Section 1.2 are influenced by the Java Collections Framework [54]. These are essentially simplified versions of the `List`, `Set`, `Map`, `SortedSet`, and `SortedMap` interfaces found in the Java Collections Framework. The accompanying source code includes wrapper classes for making `USet` and `SSet` implementations into `Set`, `Map`, `SortedSet`, and `SortedMap` implementations.

For a superb (and free) treatment of the mathematics discussed in this chapter, including asymptotic notation, logarithms, factorials, Stirling's approximation, basic probability, and lots more, see the textbook by Leyman, Leighton, and Meyer [50]. For a gentle calculus text that includes formal definitions of exponentials and logarithms, see the (freely available) classic text by Thompson [73].

For more information on basic probability, especially as it relates to computer science, see the textbook by Ross [65]. Another good reference, which covers both asymptotic notation and probability, is the textbook by Graham, Knuth, and Patashnik [37].

Readers wanting to brush up on their Java programming can find many Java tutorials online [56].

Exercise 1.1. This exercise is designed to help familiarize the reader with choosing the right data structure for the right problem. If implemented, the parts of this exercise should be done by making use of an implementation of the relevant interface (`Stack`, `Queue`, `Deque`, `USet`, or `SSet`) provided by the Java Collections Framework.

Solve the following problems by reading a text file one line at a time and performing operations on each line in the appropriate data structure(s). Your implementations should be fast enough that even files containing a million lines can be processed in a few seconds.

1. Read the input one line at a time and then write the lines out in reverse order, so that the last input line is printed first, then the second last input line, and so on.

2. Read the first 50 lines of input and then write them out in reverse order. Read the next 50 lines and then write them out in reverse

order. Do this until there are no more lines left to read, at which point any remaining lines should be output in reverse order.

In other words, your output will start with the 50th line, then the 49th, then the 48th, and so on down to the first line. This will be followed by the 100th line, followed by the 99th, and so on down to the 51st line. And so on.

Your code should never have to store more than 50 lines at any given time.

3. Read the input one line at a time. At any point after reading the first 42 lines, if some line is blank (i.e., a string of length 0), then output the line that occured 42 lines prior to that one. For example, if Line 242 is blank, then your program should output line 200. This program should be implemented so that it never stores more than 43 lines of the input at any given time.

4. Read the input one line at a time and write each line to the output if it is not a duplicate of some previous input line. Take special care so that a file with a lot of duplicate lines does not use more memory than what is required for the number of unique lines.

5. Read the input one line at a time and write each line to the output only if you have already read this line before. (The end result is that you remove the first occurrence of each line.) Take special care so that a file with a lot of duplicate lines does not use more memory than what is required for the number of unique lines.

6. Read the entire input one line at a time. Then output all lines sorted by length, with the shortest lines first. In the case where two lines have the same length, resolve their order using the usual "sorted order." Duplicate lines should be printed only once.

7. Do the same as the previous question except that duplicate lines should be printed the same number of times that they appear in the input.

8. Read the entire input one line at a time and then output the even numbered lines (starting with the first line, line 0) followed by the odd-numbered lines.

9. Read the entire input one line at a time and randomly permute the lines before outputting them. To be clear: You should not modify the contents of any line. Instead, the same collection of lines should be printed, but in a random order.

Exercise 1.2. A *Dyck word* is a sequence of +1's and -1's with the property that the sum of any prefix of the sequence is never negative. For example, $+1,-1,+1,-1$ is a Dyck word, but $+1,-1,-1,+1$ is not a Dyck word since the prefix $+1-1-1<0$. Describe any relationship between Dyck words and `Stack` `push(x)` and `pop()` operations.

Exercise 1.3. A *matched string* is a sequence of {, }, (,), [, and] characters that are properly matched. For example, "{{()[]}}" is a matched string, but this "{{()]}" is not, since the second { is matched with a]. Show how to use a stack so that, given a string of length `n`, you can determine if it is a matched string in $O(n)$ time.

Exercise 1.4. Suppose you have a `Stack`, `s`, that supports only the `push(x)` and `pop()` operations. Show how, using only a FIFO `Queue`, `q`, you can reverse the order of all elements in `s`.

Exercise 1.5. Using a `USet`, implement a `Bag`. A `Bag` is like a `USet`—it supports the `add(x)`, `remove(x)` and `find(x)` methods—but it allows duplicate elements to be stored. The `find(x)` operation in a `Bag` returns some element (if any) that is equal to `x`. In addition, a `Bag` supports the `findAll(x)` operation that returns a list of all elements in the `Bag` that are equal to `x`.

Exercise 1.6. From scratch, write and test implementations of the `List`, `USet` and `SSet` interfaces. These do not have to be efficient. They can be used later to test the correctness and performance of more efficient implementations. (The easiest way to do this is to store the elements in an array.)

Exercise 1.7. Work to improve the performance of your implementations from the previous question using any tricks you can think of. Experiment and think about how you could improve the performance of `add(i,x)` and `remove(i)` in your `List` implementation. Think about how you could improve the performance of the `find(x)` operation in your `USet` and `SSet` implementations. This exercise is designed to give you a feel for how difficult it can be to obtain efficient implementations of these interfaces.