1

Data Structures and Algorithms

How many cities with more than 250,000 people lie within 500 miles of Dallas, Texas? How many people in my company make over $100,000 per year? Can we connect all of our telephone customers with less than 1,000 miles of cable? To answer questions like these, it is not enough to have the necessary information. We must organize that information in a way that allows us to find the answers in time to satisfy our needs.

Representing information is fundamental to computer science. The primary purpose of most computer programs is not to perform calculations, but to store and retrieve information — usually as fast as possible. For this reason, the study of data structures and the algorithms that manipulate them is at the heart of computer science. And that is what this book is about — helping you to understand how to structure information to support efficient processing.

This book has three primary goals. The first is to present the commonly used data structures. These form a programmer's basic data structure "toolkit." For many problems, some data structure in the toolkit provides a good solution.

The second goal is to introduce the idea of tradeoffs and reinforce the concept that there are costs and benefits associated with every data structure. This is done by describing, for each data structure, the amount of space and time required for typical operations.

The third goal is to teach how to measure the effectiveness of a data structure or algorithm. Only through such measurement can you determine which data structure in your toolkit is most appropriate for a new problem. The techniques presented also allow you to judge the merits of new data structures that you or others might invent.

There are often many approaches to solving a problem. How do we choose between them? At the heart of computer program design are two (sometimes conflicting) goals:

1. To design an algorithm that is easy to understand, code, and debug.
2. To design an algorithm that makes efficient use of the computer's resources.

Ideally, the resulting program is true to both of these goals. We might say that such a program is "elegant." While the algorithms and program code examples presented here attempt to be elegant in this sense, it is not the purpose of this book to explicitly treat issues related to goal (1). These are primarily concerns of the discipline of Software Engineering. Rather, this book is mostly about issues relating to goal (2).

How do we measure efficiency? Chapter 3 describes a method for evaluating the efficiency of an algorithm or computer program, called **asymptotic analysis**. Asymptotic analysis also allows you to measure the inherent difficulty of a problem. The remaining chapters use asymptotic analysis techniques to estimate the time cost for every algorithm presented. This allows you to see how each algorithm compares to other algorithms for solving the same problem in terms of its efficiency.

This first chapter sets the stage for what is to follow, by presenting some higher-order issues related to the selection and use of data structures. We first examine the process by which a designer selects a data structure appropriate to the task at hand. We then consider the role of abstraction in program design. We briefly consider the concept of a design pattern and see some examples. The chapter ends with an exploration of the relationship between problems, algorithms, and programs.

1.1 A Philosophy of Data Structures

1.1.1 The Need for Data Structures

You might think that with ever more powerful computers, program efficiency is becoming less important. After all, processor speed and memory size still continue to improve. Won't any efficiency problem we might have today be solved by tomorrow's hardware?

As we develop more powerful computers, our history so far has always been to use that additional computing power to tackle more complex problems, be it in the form of more sophisticated user interfaces, bigger problem sizes, or new problems previously deemed computationally infeasible. More complex problems demand more computation, making the need for efficient programs even greater. Worse yet, as tasks become more complex, they become less like our everyday experience. Today's computer scientists must be trained to have a thorough understanding of the principles behind efficient program design, because their ordinary life experiences often do not apply when designing computer programs.

In the most general sense, a data structure is any data representation and its associated operations. Even an integer or floating point number stored on the computer can be viewed as a simple data structure. More commonly, people use the term "data structure" to mean an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring.

Given sufficient space to store a collection of data items, it is always possible to search for specified items within the collection, print or otherwise process the data items in any desired order, or modify the value of any particular data item. Thus, it is possible to perform all necessary operations on any data structure. However, using the proper data structure can make the difference between a program running in a few seconds and one requiring many days.

A solution is said to be **efficient** if it solves the problem within the required **resource constraints**. Examples of resource constraints include the total space available to store the data — possibly divided into separate main memory and disk space constraints — and the time allowed to perform each subtask. A solution is sometimes said to be efficient if it requires fewer resources than known alternatives, regardless of whether it meets any particular requirements. The **cost** of a solution is the amount of resources that the solution consumes. Most often, cost is measured in terms of one key resource such as time, with the implied assumption that the solution meets the other resource constraints.

It should go without saying that people write programs to solve problems. However, it is crucial to keep this truism in mind when selecting a data structure to solve a particular problem. Only by first analyzing the problem to determine the performance goals that must be achieved can there be any hope of selecting the right data structure for the job. Poor program designers ignore this analysis step and apply a data structure that they are familiar with but which is inappropriate to the problem. The result is typically a slow program. Conversely, there is no sense in adopting a complex representation to "improve" a program that can meet its performance goals when implemented using a simpler design.

When selecting a data structure to solve a problem, you should follow these steps.

1. Analyze your problem to determine the basic operations that must be supported. Examples of basic operations include inserting a data item into the data structure, deleting a data item from the data structure, and finding a specified data item.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to selecting a data structure operationalizes a data-centered view of the design process. The first concern is for the data and the operations to be performed on them, the next concern is the representation for those data, and the final concern is the implementation of that representation.

Resource constraints on certain key operations, such as search, inserting data records, and deleting data records, normally drive the data structure selection process. Many issues relating to the relative importance of these operations are addressed by the following three questions, which you should ask yourself whenever you must choose a data structure:

- Are all data items inserted into the data structure at the beginning, or are insertions interspersed with other operations? Static applications (where the data are loaded at the beginning and never change) typically require only simpler data structures to get an efficient implementation than do dynamic applications.
- Can data items be deleted? If so, this will probably make the implementation more complicated.
- Are all data items processed in some well-defined order, or is search for specific data items allowed? "Random access" search generally requires more complex data structures.

1.1.2 Costs and Benefits

Each data structure has associated costs and benefits. In practice, it is hardly ever true that one data structure is better than another for use in all situations. If one data structure or algorithm is superior to another in all respects, the inferior one will usually have long been forgotten. For nearly every data structure and algorithm presented in this book, you will see examples of where it is the best choice. Some of the examples might surprise you.

A data structure requires a certain amount of space for each data item it stores, a certain amount of time to perform a single basic operation, and a certain amount of programming effort. Each problem has constraints on available space and time. Each solution to a problem makes use of the basic operations in some relative proportion, and the data structure selection process must account for this. Only after a careful analysis of your problem's characteristics can you determine the best data structure for the task.

Example 1.1 A bank must support many types of transactions with its customers, but we will examine a simple model where customers wish to open accounts, close accounts, and add money or withdraw money from accounts. We can consider this problem at two distinct levels: (1) the requirements for the physical infrastructure and workflow process that the bank uses in its interactions with its customers, and (2) the requirements for the database system that manages the accounts.

The typical customer opens and closes accounts far less often than he or she accesses the account. Customers are willing to wait many minutes while accounts are created or deleted but are typically not willing to wait more than a brief time for individual account transactions such as a deposit or withdrawal. These observations can be considered as informal specifications for the time constraints on the problem.

It is common practice for banks to provide two tiers of service. Human tellers or automated teller machines (ATMs) support customer access

to account balances and updates such as deposits and withdrawals. Special service representatives are typically provided (during restricted hours) to handle opening and closing accounts. Teller and ATM transactions are expected to take little time. Opening or closing an account can take much longer (perhaps up to an hour from the customer's perspective).

From a database perspective, we see that ATM transactions do not modify the database significantly. For simplicity, assume that if money is added or removed, this transaction simply changes the value stored in an account record. Adding a new account to the database is allowed to take several minutes. Deleting an account need have no time constraint, because from the customer's point of view all that matters is that all the money be returned (equivalent to a withdrawal). From the bank's point of view, the account record might be removed from the database system after business hours, or at the end of the monthly account cycle.

When considering the choice of data structure to use in the database system that manages customer accounts, we see that a data structure that has little concern for the cost of deletion, but is highly efficient for search and moderately efficient for insertion, should meet the resource constraints imposed by this problem. Records are accessible by unique account number (sometimes called an **exact-match query**). One data structure that meets these requirements is the hash table described in Chapter 9.4. Hash tables allow for extremely fast exact-match search. A record can be modified quickly when the modification does not affect its space requirements. Hash tables also support efficient insertion of new records. While deletions can also be supported efficiently, too many deletions lead to some degradation in performance for the remaining operations. However, the hash table can be reorganized periodically to restore the system to peak efficiency. Such reorganization can occur offline so as not to affect ATM transactions.

Example 1.2 A company is developing a database system containing information about cities and towns in the United States. There are many thousands of cities and towns, and the database program should allow users to find information about a particular place by name (another example of an exact-match query). Users should also be able to find all places that match a particular value or range of values for attributes such as location or population size. This is known as a **range query**.

A reasonable database system must answer queries quickly enough to satisfy the patience of a typical user. For an exact-match query, a few seconds is satisfactory. If the database is meant to support range queries that can return many cities that match the query specification, the entire opera-

tion may be allowed to take longer, perhaps on the order of a minute. To meet this requirement, it will be necessary to support operations that process range queries efficiently by processing all cities in the range as a batch, rather than as a series of operations on individual cities.

The hash table suggested in the previous example is inappropriate for implementing our city database, because it cannot perform efficient range queries. The B^+-tree of Section 10.5.1 supports large databases, insertion and deletion of data records, and range queries. However, a simple linear index as described in Section 10.1 would be more appropriate if the database is created once, and then never changed, such as an atlas distributed on a CD or accessed from a website.

1.2 Abstract Data Types and Data Structures

The previous section used the terms "data item" and "data structure" without properly defining them. This section presents terminology and motivates the design process embodied in the three-step approach to selecting a data structure. This motivation stems from the need to manage the tremendous complexity of computer programs.

A **type** is a collection of values. For example, the Boolean type consists of the values `true` and `false`. The integers also form a type. An integer is a **simple type** because its values contain no subparts. A bank account record will typically contain several pieces of information such as name, address, account number, and account balance. Such a record is an example of an **aggregate type** or **composite type**. A **data item** is a piece of information or a record whose value is drawn from a type. A data item is said to be a **member** of a type.

A **data type** is a type together with a collection of operations to manipulate the type. For example, an integer variable is a member of the integer data type. Addition is an example of an operation on the integer data type.

A distinction should be made between the logical concept of a data type and its physical implementation in a computer program. For example, there are two traditional implementations for the list data type: the linked list and the array-based list. The list data type can therefore be implemented using a linked list or an array. Even the term "array" is ambiguous in that it can refer either to a data type or an implementation. "Array" is commonly used in computer programming to mean a contiguous block of memory locations, where each memory location stores one fixed-length data item. By this meaning, an array is a physical data structure. However, array can also mean a logical data type composed of a (typically homogeneous) collection of data items, with each data item identified by an index number. It is possible to implement arrays in many different ways. For exam-

ple, Section 12.2 describes the data structure used to implement a sparse matrix, a large two-dimensional array that stores only a relatively few non-zero values. This implementation is quite different from the physical representation of an array as contiguous memory locations.

An **abstract data type** (ADT) is the realization of a data type as a software component. The interface of the ADT is defined in terms of a type and a set of operations on that type. The behavior of each operation is determined by its inputs and outputs. An ADT does not specify *how* the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as **encapsulation**.

A **data structure** is the implementation for an ADT. In an object-oriented language such as **C**++, an ADT and its implementation together make up a **class**. Each operation associated with the ADT is implemented by a **member function** or **method**. The variables that define the space required by a data item are referred to as **data members**. An **object** is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

The term "data structure" often refers to data stored in a computer's main memory. The related term **file structure** often refers to the organization of data on peripheral storage, such as a disk drive or CD.

Example 1.3 The mathematical concept of an integer, along with operations that manipulate integers, form a data type. The **C**++ `int` variable type is a physical representation of the abstract integer. The `int` variable type, along with the operations that act on an `int` variable, form an ADT. Unfortunately, the `int` implementation is not completely true to the abstract integer, as there are limitations on the range of values an `int` variable can store. If these limitations prove unacceptable, then some other representation for the ADT "integer" must be devised, and a new implementation must be used for the associated operations.

Example 1.4 An ADT for a list of integers might specify the following operations:

- Insert a new integer at a particular position in the list.
- Return `true` if the list is empty.
- Reinitialize the list.
- Return the number of integers currently in the list.
- Delete the integer at a particular position in the list.

From this description, the input and output of each operation should be clear, but the implementation for lists has not been specified.

One application that makes use of some ADT might use particular member functions of that ADT more than a second application, or the two applications might have different time requirements for the various operations. These differences in the requirements of applications are the reason why a given ADT might be supported by more than one implementation.

Example 1.5 Two popular implementations for large disk-based database applications are hashing (Section 9.4) and the B^+-tree (Section 10.5). Both support efficient insertion and deletion of records, and both support exact-match queries. However, hashing is more efficient than the B^+-tree for exact-match queries. On the other hand, the B^+-tree can perform range queries efficiently, while hashing is hopelessly inefficient for range queries. Thus, if the database application limits searches to exact-match queries, hashing is preferred. On the other hand, if the application requires support for range queries, the B^+-tree is preferred. Despite these performance issues, both implementations solve versions of the same problem: updating and searching a large collection of records.

The concept of an ADT can help us to focus on key issues even in non-computing applications.

Example 1.6 When operating a car, the primary activities are steering, accelerating, and braking. On nearly all passenger cars, you steer by turning the steering wheel, accelerate by pushing the gas pedal, and brake by pushing the brake pedal. This design for cars can be viewed as an ADT with operations "steer," "accelerate," and "brake." Two cars might implement these operations in radically different ways, say with different types of engine, or front- versus rear-wheel drive. Yet, most drivers can operate many different cars because the ADT presents a uniform method of operation that does not require the driver to understand the specifics of any particular engine or drive design. These differences are deliberately hidden.

The concept of an ADT is one instance of an important principle that must be understood by any successful computer scientist: managing complexity through abstraction. A central theme of computer science is complexity and techniques for handling it. Humans deal with complexity by assigning a label to an assembly of objects or concepts and then manipulating the label in place of the assembly. Cognitive psychologists call such a label a **metaphor**. A particular label might be related to other pieces of information or other labels. This collection can in turn be given a label, forming a hierarchy of concepts and labels. This hierarchy of labels allows us to focus on important issues while ignoring unnecessary details.

Example 1.7 We apply the label "hard drive" to a collection of hardware that manipulates data on a particular type of storage device, and we apply the label "CPU" to the hardware that controls execution of computer instructions. These and other labels are gathered together under the label "computer." Because even the smallest home computers today have millions of components, some form of abstraction is necessary to comprehend how a computer operates.

Consider how you might go about the process of designing a complex computer program that implements and manipulates an ADT. The ADT is implemented in one part of the program by a particular data structure. While designing those parts of the program that use the ADT, you can think in terms of operations on the data type without concern for the data structure's implementation. Without this ability to simplify your thinking about a complex program, you would have no hope of understanding or implementing it.

Example 1.8 Consider the design for a relatively simple database system stored on disk. Typically, records on disk in such a program are accessed through a buffer pool (see Section 8.3) rather than directly. Variable length records might use a memory manager (see Section 12.3) to find an appropriate location within the disk file to place the record. Multiple index structures (see Chapter 10) will typically be used to access records in various ways. Thus, we have a chain of classes, each with its own responsibilities and access privileges. A database query from a user is implemented by searching an index structure. This index requests access to the record by means of a request to the buffer pool. If a record is being inserted or deleted, such a request goes through the memory manager, which in turn interacts with the buffer pool to gain access to the disk file. A program such as this is far too complex for nearly any human programmer to keep all of the details in his or her head at once. The only way to design and implement such a program is through proper use of abstraction and metaphors. In object-oriented programming, such abstraction is handled using classes.

Data types have both a **logical** and a **physical** form. The definition of the data type in terms of an ADT is its logical form. The implementation of the data type as a data structure is its physical form. Figure 1.1 illustrates this relationship between logical and physical forms for data types. When you implement an ADT, you are dealing with the physical form of the associated data type. When you use an ADT elsewhere in your program, you are concerned with the associated data type's logical form. Some sections of this book focus on physical implementations for a

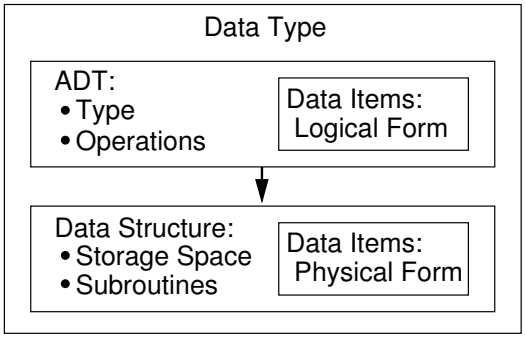

Figure 1.1 The relationship between data items, abstract data types, and data structures. The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

given data structure. Other sections use the logical ADT for the data structure in the context of a higher-level task.

Example 1.9 A particular C++ environment might provide a library that includes a list class. The logical form of the list is defined by the public functions, their inputs, and their outputs that define the class. This might be all that you know about the list class implementation, and this should be all you need to know. Within the class, a variety of physical implementations for lists is possible. Several are described in Section 4.1.

1.3 Design Patterns

At a higher level of abstraction than ADTs are abstractions for describing the design of programs — that is, the interactions of objects and classes. Experienced software designers learn and reuse patterns for combining software components. These have come to be referred to as **design patterns**.

A design pattern embodies and generalizes important design concepts for a recurring problem. A primary goal of design patterns is to quickly transfer the knowledge gained by expert designers to newer programmers. Another goal is to allow for efficient communication between programmers. It is much easier to discuss a design issue when you share a technical vocabulary relevant to the topic.

Specific design patterns emerge from the realization that a particular design problem appears repeatedly in many contexts. They are meant to solve real problems. Design patterns are a bit like templates. They describe the structure for a design solution, with the details filled in for any given problem. Design patterns are a bit like data structures: Each one provides costs and benefits, which implies