1 Introduction

1.1 What Are Data Structures and Algorithms?

If this book is about data structures and algorithms, then perhaps we should start by defining these terms. We begin with a definition for "algorithm."

> **Algorithm**: A finite sequence of steps for accomplishing some computational task. An algorithm must

- Have steps that are simple and definite enough to be done by a computer, and
- Terminate after finitely many steps.

This definition of an algorithm is similar to others you may have seen in prior computer science courses. Notice that an algorithm is a sequence of steps, not a program. You might use the same algorithm in different programs, or express the same algorithm in different languages, because an algorithm is an entity that is abstracted from implementation details. Part of the point of this course is to introduce you to algorithms that you can use no matter what language you program in. We will write programs in a particular language, but we are really studying the algorithms, not their implementations.

The definition of a data structure is a bit more involved. We begin with the notion of an abstract data type.

> **Abstract data type (ADT)**: A set of values (the **carrier set**), and operations on those values.

Here are some examples of ADTs:

> *Boolean*—The carrier set of the Boolean ADT is the set {true, false}. The operations on these values are negation, conjunction, disjunction, conditional, is equal to, and perhaps some others.

> *Integer*—The carrier set of the Integer ADT is the set {…, -2, -1, 0, 1, 2, …}, and the operations on these values are addition, subtraction, multiplication, division, remainder, is equal to, is less than, is greater than, and so on. Note that although some of these operations yield other Integer values, some yield values from other ADTs (like true and false), but all have at least one Integer argument.

> *String*—The carrier set of the String ADT is the set of all finite sequences of characters from some alphabet, including the empty sequence (the empty string). Operations on string values include concatenation, length of, substring, index of, and so forth.

Bit String—The carrier set of the Bit String ADT is the set of all finite sequences of bits, including the empty strings of bits, which we denote λ. This set is {λ, 0, 1, 00, 01, 10, 11, 000, …}. Operations on bit strings include complement (which reverses all the bits), shifts (which rotates a bit string left or right), conjunction and disjunction (which combine bits at corresponding locations in the strings), and concatenation and truncation.

The thing that makes an abstract data type *abstract* is that its carrier set and its operations are mathematical entities, like numbers or geometric objects; all details of implementation on a computer are ignored. This makes it easier to reason about them and to understand what they are. For example, we can decide how *div* and *mod* should work for negative numbers in the Integer ADT without having to worry about how to make this work on real computers. Then we can deal with implementation of our decisions as a separate problem.

Once an abstract data type is implemented on a computer, we call it a data type.

Data type: An implementation of an abstract data type on a computer.

Thus, for example, the Boolean ADT is implemented as the `boolean` type in Java, and the `bool` type in C++; the Integer ADT is realized as the `int` and `long` types in Java, and the `Integer` class in Ruby; the String ADT is implemented as the `String` class in Java and Ruby.

Abstract data types are very useful for helping us understand the mathematical objects that we use in our computations but, of course, we cannot use them directly in our programs. To use ADTs in programming, we must figure out how to implement them on a computer. Implementing an ADT requires two things:

- Representing the values in the carrier set of the ADT by data stored in computer memory, and
- Realizing computational mechanisms for the operations of the ADT.

Finding ways to represent carrier set values in a computer's memory requires that we determine how to arrange data (ultimately bits) in memory locations so that each value of the carrier set has a unique representation. Such things are data structures.

Data structure: An arrangement of data in memory locations to represent values of the carrier set of an abstract data type.

Realizing computational mechanisms for performing operations of the type really means finding algorithms that use the data structures for the carrier set to implement the operations of the ADT. And now it should be clear why we study data structures and algorithms together: to implement an ADT, we must find data structures to represent the values of its carrier set and algorithms to work with these data structures to implement its operations.

A course in data structures and algorithms is thus a course in implementing abstract data types. It may seem that we are paying a lot of attention to a minor topic, but abstract data types are really the foundation of everything we do in computing. Our computations work on data. This data must represent things and be manipulated according to rules. These things and the rules for their manipulation amount to abstract data types.

Usually there are many ways to implement an ADT. A large part of the study of data structures and algorithms is learning about alternative ways to implement an ADT and evaluating the alternatives to determine their advantages and disadvantages. Typically some alternatives will be better for certain applications and other alternatives will be better for other applications. Knowing how to do such evaluations to make good design decisions is an essential part of becoming an expert programmer.

1.2 Structure of the Book

In this book we will begin by studying fundamental data types that are usually implemented for us in programming languages. Then we will consider how to use these fundamental types and other programming language features (such as references) to implement more complicated ADTs. Along the way we will construct a classification of complex ADTs that will serve as the basis for a class library of implementations. We will also learn how to measure an algorithm's efficiency and use this skill to study algorithms for searching and sorting, which are very important in making our programs efficient when they must process large data sets.

1.3 The Ruby Programming Language

Although the data structures and algorithms we study are not tied to any program or programming language, we need to write particular programs in particular languages to practice implementing and using the data structures and algorithms that we learn. In this book, we will use the Ruby programming language.

Ruby is an interpreted, purely object-oriented language with many powerful features, such as garbage collection, dynamic arrays, hash tables, and rich string processing facilities. We use Ruby because it is a fairly popular, full-featured, object-oriented language, but it can be learned well enough to write substantial programs fairly quickly. Thus we will be able to use a powerful language and still have time to concentrate on data structures and algorithms, which is what we are really interested in. Also, it is free.

Ruby is dynamically typed, does not support design-by-contract, and has a somewhat frugal collection of features for object-oriented programming. Although this makes the language easier to learn and use, it also opens up many opportunities for errors. Careful attention to types and mechanisms to help detect type errors early, fully understanding preconditions for executing methods, and thoughtful use of class hierarchies are important for novice programmers, so we will pay close attention to these matters in our discussion of data structures and algorithms and we will, when possible, incorporate this material into Ruby code. This sometimes results in code that does not conform to the style prevalent in the Ruby community. However, programmers must understand and appreciate these matters so that they can handle data structures in more strongly typed languages such as Java, C++, or C#.

1.4 Review Questions

1. What are the carrier set and some operations of the Character ADT?
2. How might the Bit String ADT carrier set be represented on a computer in some high level language?
3. How might the concatenation operation of the Bit String ADT be realized using the carrier set representation you devised for question two above?
4. What do your answers to questions two and three above have to do with data structures and algorithms?

1.5 Exercises

1. Describe the carrier sets and some operations for the following ADTs:
   a) The Real numbers
   b) The Rational numbers
   c) The Complex numbers
   d) Ordered pairs of Integers
   e) Sets of Characters
   f) Grades (the letters A, B, C, D, and F)

2. For each of the ADTs in exercise one, either indicate how the ADT is realized in some programming language, or describe how the values in the carrier set might be realized using the facilities of some programming language, and sketch how the operations of the ADT might be implemented.

1.6 Review Question Answers

1. We must first choose a character set; suppose we use the ASCII characters. Then the carrier set of the Character ADT is the set of ASCII characters. Some operations of this ADT might be those to change character case from lower to upper and the reverse, classification operations to determine whether a character is a letter, a digit, whitespace, punctuation, a printable character, and so forth, and operations to convert between integers and characters.
2. Bit String ADT values could be represented in many ways. For example, bit strings might be represented in character strings of "0"s and "1"s. They might be represented by arrays or lists of characters, Booleans, or integers.
3. If bit strings are represented as characters strings, then the bit string concatenation operation is realized by the character string concatenation operation. If bit strings are represented by arrays or lists, then the concatenation of two bit strings is a new array or list whose size is the sum of the sizes of the argument data structures consisting of the bits from the first bit string copied into the initial portion of the result array or list, followed by the bits from the second bit string copied into the remaining portion.
4. The carrier set representations described in the answer to question two are data structures, and the implementations of the concatenation operation described in the answer to question three are (sketches of) algorithms.