Data Structures

en.wikibooks.org

July 5, 2015

Contents

Figure 1

Figure 2 CC

5

Computers can store and process vast amounts of data. Formal data structures enable a programmer to mentally structure large amounts of data into conceptually manageable relationships.

Sometimes we use data structures to allow us to do more: for example, to accomplish fast searching or sorting of data. Other times, we use data structures so that we can do *less* : for example, the concept of the stack is a limited form of a more general data structure. These limitations provide us with guarantees that allow us to reason about our programs more easily. Data structures also provide guarantees about algorithmic complexity — choosing an appropriate data structure for a job is crucial for writing good software.

Because data structures are higher-level abstractions, they present to us operations on groups of data, such as adding an item to a list, or looking up the highest-priority item in a queue. When a data structure provides operations, we can call the data structure an **abstract data type** (sometimes abbreviated as ADT). Abstract data types can minimize dependencies in your code, which is important when your code needs to be changed. Because you are abstracted away from lower-level details, some of the higher-level commonalities one

1 http://en.wikipedia.org/wiki/Creative%20Commons
2 http://creativecommons.org/licenses/by-sa/3.0/
3 http://creativecommons.org/compatiblelicenses
4 http://en.wikipedia.org/wiki/
5 http://en.wikibooks.org/wiki/Category%3AData%20Structures

data structure shares with a different data structure can be used to replace one with the other.

Our programming languages come equipped with a set of built-in types, such as integers and floating-point numbers, that allow us to work with data objects for which the machine's processor has native support. These built-in types are abstractions of what the processor actually provides because built-in types hide details both about their execution and limitations.

For example, when we use a floating-point number we are primarily concerned with its value and the operations that can be applied to it. Consider computing the length of a hypotenuse:

```
let c := sqrt(a * a + b * b)
```

The machine code generated from the above would use common patterns for computing these values and accumulating the result. In fact, these patterns are so repetitious that high-level languages were created to avoid this redundancy and to allow programmers to think about *what* value was computed instead of *how* it was computed.

Two useful and related concepts are at play here:

- **Encapsulation** is when common patterns are grouped together under a single name and then parameterized, in order to achieve a higher-level understanding of that pattern. For example, the multiplication operation requires two source values and writes the product of those two values to a given destination. The operation is parameterized by both the two sources and the single destination.
- **Abstraction** is a mechanism to hide the implementation details of an abstraction away from the users of the abstraction. When we multiply numbers, for example, we don't need to know the technique actually used by the processor, we just need to know its properties.

A programming language is both an abstraction of a machine and a tool to encapsulate-away the machine's inner details. For example, a program written in a programming language can be compiled to several different machine architectures when that programming language sufficiently encapsulates the user away from any one machine.

In this book, we take the abstraction and encapsulation that our programming languages provide a step further: When applications get to be more complex, the abstractions of programming languages become too low-level to effectively manage. Thus, we build our own abstractions on top of these lower-level constructs. We can even build further abstractions on top of those abstractions. Each time we build upwards, we lose access to the lower-level implementation details. While losing such access might sound like a bad trade off, it is actually quite a bargain: We are primarily concerned with solving the problem at hand rather than with any trivial decisions that could have just as arbitrarily been replaced with a different decision. When we can think on higher levels, we relieve ourselves of these burdens.

Each data structure that we cover in this book can be thought of as a single unit that has a set of values and a set of operations that can be performed to either access or change these values. The data structure itself can be understood as a set of the data structure's