Data Structures/Introduction

Data Structures
Introduction - Asymptotic Notation - Arrays - List Structures & Iterators
Stacks & Queues - Trees - Min & Max Heaps - Graphs
Hash Tables - Sets - Tradeoffs

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 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 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 operations together with each operation's properties (i.e., what the operation does and how long we could expect it to take).

Big-oh notation is a common way of expressing a computer code's performance. The notation creates a relationship between the number of items in memory and the average performance for a function. For a set of items, indicates that a particular function will operate on the set times on average. indicates that the function always performs a constant number of operations regardless of the number of items. The notation only represents algorithmic complexity so a function may perform more operations but constant multiples of are dropped by convention.

The Node

The first data structure we look at is the node structure. A node is simply a container for a value, plus a pointer to a "next" node (which may be `null`).

The above is an abstraction of a *structure*:

In some languages, structures are called *records* or *classes*. Some other languages provide no direct support for structures, but instead allow them to be built from other constructs (such as *tuples* or *lists*).

Here, we are only concerned that nodes contain values of some form, so we simply say its type is "element" because the type is not important. In some programming languages no type ever needs to be specified (as in dynamically typed languages, like Scheme, Smalltalk or Python). In other languages the type might need to be restricted to integer or string (as in statically typed languages like C). In still other languages, the decision of the type of the contained element can be delayed until the type is actually used (as in languages that support generic types, like C++ and Java). In any of these cases, translating the pseudocode into your own language should be relatively simple.

Each of the node operations specified can be implemented quite easily:

```
// Create a new node, with v as its contained value and next as
// the value of the next pointer
function make-node(v, node next): node
  let result := new node {v, next}
  return result
end

// Returns the value contained in node n
function get-value(node n): element
  return n.value
end

// Returns the value of node n's next pointer
function get-next(node n): node
  return n.next
end

// Sets the contained value of n to be v
function set-value(node n, v)
  n.value := v
end

// Sets the value of node n's next pointer to be new-next
function set-next(node n, new-next)
  n.next := new-next
  return new-next
end
```

Principally, we are more concerned with the operations and the implementation strategy than we are with the structure itself and the low-level implementation. For example, we are more concerned about the time requirement specified, which states that all operations take time that is . The above implementation meets this criteria, because the length of time each operation takes is constant. Another way to think of constant time operations is to think of them as operations whose analysis is not dependent on any variable. (The notation is mathematically defined in the next chapter. For now, it is safe to assume it just means constant time.)

Because a node is just a container both for a value and container to a pointer to another node, it shouldn't be surprising how trivial the node data structure itself (and its implementation) is.

Building a Chain from Nodes

Although the node structure is simple, it actually allows us to compute things that we couldn't have computed with just fixed-size integers alone.

But first, we'll look at a program that doesn't need to use nodes. The following program will read in (from an input stream; which can either be from the user or a file) a series of numbers until the end-of-file is reached and then output what the largest number is and the average of all numbers:

```
program(input-stream in, output-stream out)
  let total := 0
  let count := 0
  let largest :=

  while has-next-integer(in):
    let i := read-integer(in)
    total := total + i
    count := count + 1
    largest := max(largest, i)
  repeat

  println out "Maximum: " largest

  if count != 0:
    println out "Average: " (total / count)
  fi
end
```

But now consider solving a similar task: read in a series of numbers until the end-of-file is reached, and output the largest number and the average of all numbers that evenly divide the largest number. This problem is different because it's possible the largest number will be the last one entered: if we are to compute the average of all numbers that divide that number, we'll need to somehow remember all of them. We could use variables to remember the previous numbers, but variables would only help us solve the problem when there aren't too many numbers entered.

For example, suppose we were to give ourselves 200 variables to hold the state input by the user. And further suppose that each of the 200 variables had 64-bits. Even if we were very clever with our program, it could only compute results for different types of input. While this is a very large number of combinations, a list of 300 64-bit numbers would require even more combinations to be properly encoded. (In general, the problem is said to require *linear space*. All programs that need only a finite number of variables can be solved in *constant space*.)

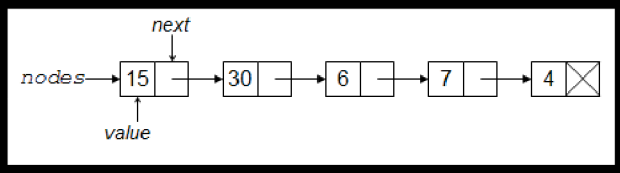
Instead of building-in limitations that complicate coding (such as having only a constant number of variables), we can use the properties of the node abstraction to allow us to remember as many numbers as our computer can hold:

```
program(input-stream in, output-stream out)
  let largest :=
  let nodes := null

  while has-next-integer(in):
    let i := read-integer(in)
    nodes := make-node(i, nodes) // contain the value i,
                                 // and remember the previous numbers too
    largest := max(largest, i)
  repeat
  println out "Maximum: " largest

  // now compute the averages of all factors of largest
  let total := 0
  let count := 0
  while nodes != null:
    let j := get-value(nodes)
    if j divides largest:
      total := total + j
      count := count + 1
```

```
    fi
    nodes := get-next(nodes)
  repeat
  if count != 0:
    println out "Average: " (total / count)
  fi
end
```

Above, if n integers are successfully read there will be n calls made to **make-node**. This will require n nodes to be made (which require enough space to hold the *value* and *next* fields of each node, plus internal memory management overhead), so the memory requirements will be on the order of . Similarly, we construct this chain of nodes and then iterate over the chain again, which will require steps to make the chain, and then another steps to iterate over it.

Note that when we iterate the numbers in the chain, we are actually looking at them in reverse order. For example, assume the numbers input to our program are 4, 7, 6, 30, and 15. After EOF is reached, the *nodes* chain will look like this:



Such chains are more commonly referred to as *linked-lists*. However, we generally prefer to think in terms of *lists* or *sequences*, which aren't as low-level: the linking concept is just an implementation detail. While a list can be made with a chain, in this book we cover several other ways to make a list. For the moment, we care more about the abstraction capabilities of the node than we do about one of the ways it is used.

The above algorithm only uses the make-node, get-value, and get-next functions. If we use set-next we can change the algorithm to generate the chain so that it keeps the original ordering (instead of reversing it).

```
program (input-stream in, output-stream out)
  let largest :=
  let nodes := null
  let tail_node := null

  while has-next-integer (in):
    let i := read-integer (in)
    if (nodes == null)
      nodes := make-node(i, null) // construct first node in the list
      tail_node := nodes //there is one node in the list=> first and last are the same
    else
      tail_node := set-next (tail_node, make-node (i, null)) // append new node to the end of the list
    Largest := max(Largest, i)
  repeat
  println out "Maximum: " Largest

  // now compute the averages of all factors of Largest
  let total := 0
  let count := 0
  while nodes != null:
    let j := get-value(nodes)
    if j divides largest:
      total := total + j
      count := count + 1
    fi
    nodes := get-next(nodes)
  repeat
  if count != 0:
    println out "Average: " (total / count)
  fi
end
```

The Principle of Induction

The chains we can build from nodes are a demonstration of the principle of mathematical induction:

Mathematical Induction

1. Suppose you have some property of numbers
2. If you can prove that when holds that must also hold, then
3. All you need to do is prove that holds to show that holds for all natural

For example, let the property be the statement that "you can make a chain that holds numbers". This is a property of natural numbers, because the sentence makes sense for specific values of :

- you can make a chain that holds 5 numbers
- you can make a chain that holds 100 numbers
- you can make a chain that holds 1,000,000 numbers

Instead of proving that we can make chains of length 5, 100, and one million, we'd rather prove the general statement instead. Step 2 above is called the **Inductive Hypothesis**; let's show that we can prove it:

- Assume that holds. That is, that we can make a chain of elements. Now we must show that holds.
- Assume `chain` is the first node of the -element chain. Assume `i` is some number that we'd like to add to the chain to make an length chain.
- The following code can accomplish this for us:

```
let bigger-chain := make-node(i, chain)
```

- Here, we have the new number *i* that is now the contained value of the first link of the *bigger-chain*. If *chain* had elements, then *bigger-chain* must have elements.

Step 3 above is called the **Base Case**, let's show that we can prove it:

- We must show that holds. That is, that we can make a chain of one element.
- The following code can accomplish this for us:

```
let chain := make-node(i, null)
```

The principle of induction says, then, that we have proven that we can make a chain of elements for all value of . How is this so? Probably the best way to think of induction is that it's actually a way of creating a formula to describe an infinite number of proofs. After we prove that the statement is true for , the base case, we can apply the inductive hypothesis to that fact to show that holds. Since we now know that holds, we can apply the inductive hypothesis again to show that must hold. The principle says that there is nothing to stop us from doing this repeatedly, so we should assume it holds for all cases.

Induction may sound like a strange way to prove things, but it's a very useful technique. What makes the technique so useful is that it can take a hard sounding statement like "prove holds for all " and break it into two smaller, easier to prove statements. Typically base cases are easy to prove because they are not general statements at all. Most of the proof work is usually in the inductive hypothesis, which can often require clever ways of reformulating the statement to "attach on" a proof of the case.

You can think of the contained value of a node as a base case, while the next pointer of the node as the inductive hypothesis. Just as in mathematical induction, we can break the hard problem of storing an arbitrary number of elements into an easier problem of just storing one element and then having a mechanism to attach on further elements.

Induction on a Summation

The next example of induction we consider is more algebraic in nature:

Let's say we are given the formula and we want to prove that this formula gives us the sum of the first numbers. As a first attempt, we might try to just show that this is true for 1

,

for 2

,

for 3

and so on, however we'd quickly realize that our so called proof would take infinitely long to write out! Even if you carried out this proof and showed it to be true for the first billion numbers, that doesn't nescessarily mean that it would be true for one billion and one or even a hundred billion. This is a strong hint that maybe induction would be useful here.

Let's say we want to prove that the given formula really does give the sum of the first n numbers using induction. The first step is to prove the base case; i.e. we have to show that it is true when n = 1. This is relatively easy; we just substitute 1 for the variable n and we get ( ), which shows that the formula is correct when n = 1.

Now for the inductive step. We have to show that if the formula is true for j, it is also true for j + 1. To phrase it another way, assuming we've already proven that the sum from 1 to (j) is , we want to prove that the sum from 1 to (j+1) is . Note that those two formulas came about just by replacing n with (j) and (j+1) respectively.

To prove this inductive step, first note that to calculate the sum from 1 to j+1, you can just calculate the sum from 1 to j, then add j+1 to it. We already have a formula for the sum from 1 to j, and when we add j+1 to that formula, we get this new formula: . So to actually complete the proof, all we'd need to do is show that .

We can show the above equation is true via a few simplification steps:

Data Structures
Introduction - Asymptotic Notation - Arrays - List Structures & Iterators
Stacks & Queues - Trees - Min & Max Heaps - Graphs
Hash Tables - Sets - Tradeoffs