



## Data Structures: Crash Course Computer Science #14

Crash Course: Computer Science

<https://youtube.com/watch?v=DuDz6B4cqVc>

<https://nerdfighteria.info/v/DuDz6B4cqVc>

Hi, I'm Carrie Anne, and welcome to Crash Course Computer Science!

Last episode, we discussed a few example classic algorithms, like sorting a list of numbers and finding the shortest path in a graph. What we didn't talk much about, is how the data the algorithms ran on was stored in computer memory.

You don't want your data to be like John Green's college dorm room, with food, clothing and papers strewn everywhere. Instead, we want our data to be structured, so that it's organized, allowing things to be easily retrieved and read. For this, computer scientists use Data Structures!

### INTRO

We already introduced one basic data structure last episode, arrays, also called lists or vectors in some languages. These are a series of values stored in memory. So instead of just a single value being saved into a variable, like 'j equals 5', we can define a whole series of numbers, and save that into an array variable.

To be able to find a particular value in this array, we have to specify an index. Almost all programming languages start arrays at index 0, and use a square bracket syntax to denote array access. So, for example, if we want to add the values in the first and third spots of our array 'j', and save that into a variable 'a', we would write a line of code like this.

How an array is stored in memory is pretty straightforward. For simplicity, let's say that the compiler chose to store ours at memory location 1,000. The array contains 7 numbers, and these are stored one after another in memory, as seen here.

So when we write "j index of 0", the computer goes to memory location 1,000, with an offset of 0, and we get the value 5. If we wanted to retrieve "j index of 5", our program goes to memory location 1000, plus an offset of 5, which in this case, holds a value of 4. It's easy to confuse the fifth number in the array with the number at index 5.

They are not the same. Remember, the number at index 5 is the 6th number in the array because the first number is at index 0. Arrays are extremely versatile data structures, used all the time, and so there are many functions that can handle them to do useful things.

For example, pretty much every programming language comes with a built-in sort function, where you just pass in your array, and it comes back sorted. So there's no need to write that algorithm from scratch. Very closely related are strings, which are just arrays of characters, like letters, numbers, punctuation and other written symbols.

We talked about how computers store characters way back in Episode 4. Most often, to save a string into memory, you just put it in quotes, like so. Although it doesn't look like an array, it is.

Behind the scenes, the memory looks like this. Note that the string ends with a zero in memory. It's not the character zero, but the binary value 0.

This is called the null character, and denotes the end of the string in memory. This is important because if I call a function like "print quote", which writes the string to the screen, it prints out each character in turn starting at the first memory location, but it needs to know when to stop! Otherwise, it would print out every single thing in memory as text.

The zero tells the string functions when to stop. Because computers

work with text so often, there are many functions that specifically handle strings. For example, many programming languages have a string concatenation function, or "string cat," which takes in two strings, and copies the second one to the end of the first.

We can use arrays for making one-dimensional lists, but sometimes you want to manipulate data that is two dimensional, like a grid of numbers in a spreadsheet, or the pixels on your computer screen. For this, we need a matrix. You can think of a matrix as an array of arrays!

So a 3 by 3 matrix is really an array of size 3, with each index storing an array of size 3. We can initialize a matrix like so. In memory, this is packed together in order like this.

To access a value, you need to specify two indexes, like "J index of 2, then index of 1" - this tells the computer you're looking for the item in subarray 2 at position 1. And this would give us the value 12. The cool thing about matrices is we're not limited to 3 by 3 -- we can make them any size we want -- and we can also make them any number of dimensions we want.

For example, we can create a five dimensional matrix and access it like this. That's right, you now know how to access a five dimensional matrix -- tell your friends! So far, we've been storing individual numbers or letters into our arrays or matrices.

But often it's useful to store a block of related variables together. Like, you might want to store a bank account number along with its balance. Groups of variables like these can be bundled together into a struct.

Now we can create variables that aren't just single numbers, but are compound data structures, able to store several pieces of data at once. We can even make arrays of structs that we define, which are automatically bundled together in memory. If we access, for example, J index of 0, we get back the whole struct stored there, and we can pull the specific account number and balance data we want.

This array of structs, like any other array, gets created at a fixed size that can't be enlarged to add more items. Also, arrays must be stored in order in memory, making it hard to add a new item to the middle. But, the struct data structure can be used for building more complicated data structures that avoid these restrictions.

Let's take a look at this struct that's called a "node". It stores a variable, like a number, and also a pointer. A pointer is a special variable that points, hence the name, to a location in memory.

Using this struct, we can create a linked list, which is a flexible data structure that can store many nodes. It does this by having each node point to the next node in the list. Let's imagine we have three node structs saved in memory, at locations 1000, 1002 and 1008.

They might be spaced apart, because they were created at different times, and other data can sit between them. So, you see that the first node contains the value 7, and the location 1008 in its "next" pointer. This means that the next node in the linked list is located at memory location 1008.

Looking down the linked list, to the next node, we see it stores the value 112 and points to another node at location 1002. If we follow that, we find a node that contains the value 14 and points back to the first node at location 1000. So this linked list happened to be circular, but it could also have been terminated by using a next pointer value of 0 -- the null value -- which would indicate we've reached the end of the list.



## Data Structures: Crash Course Computer Science #14

Crash Course: Computer Science

<https://youtube.com/watch?v=DuDz6B4cqVc>

<https://nerdfighteria.info/v/DuDz6B4cqVc>

When programmers use linked lists, they rarely look at the memory values stored in the next pointers. Instead, they can use an abstraction of a linked list, that looks like this, which is much easier to conceptualize. Unlike an array, whose size has to be pre-defined, linked lists can be dynamically extended or shortened.

For example, we can allocate a new node in memory, and insert it into this list, just by changing the next pointers. Linked Lists can also easily be re-ordered, trimmed, split, reversed, and so on. Which is pretty nifty!

And pretty useful for algorithms like sorting, which we talked about last week. Owing to this flexibility, many more-complex data structures are built on top of linked lists. The most famous and universal are queues and stacks. A queue – like the line at your post office – goes in order of arrival.

The person who has been waiting the longest, gets served first. No matter how frustrating it is that all you want to do is buy stamps and the person in front of you seems to be mailing 23 packages. But, regardless, this behavior is called First-In First-Out, or FIFO.

That's the first part. Not the 23 packages thing. Imagine we have a pointer, named "post office queue," that points to the first node in our linked list.

Once we're done serving Hank, we can read Hank's next pointer, and update our "post office queue" pointer to the next person in the line. We've successfully dequeued Hank -- he's gone, done, finished. If we want to enqueue someone, that is, add them to the line, we have to traverse down the linked list until we hit the end, and then change that next pointer to point to the new person.

With just a small change, we can use linked lists as stacks, which are LIFO... Last-In First-Out. You can think of this like a stack of pancakes... as you make them, you add them to the top of stack. And when you want to eat one, you take them from the top of the stack.

Delicious! Instead of enqueueing and dequeuing, data is pushed onto the stack and popped from the stacks. Yep, those are the official terms!

If we update our node struct to contain not just one, but two pointers, we can build trees, another data structure that's used in many algorithms. Again, programmers rarely look at the values of these pointers, and instead conceptualize trees like this: The top most node is called the root. And any nodes that hang from other nodes are called children nodes.

As you might expect, nodes above children are called parent nodes. Does this example imply that Thomas Jefferson is the parent of Aaron Burr? I'll leave that to your fanfiction to decide.

And finally, any nodes that have no children -- where the tree ends -- are called Leaf Nodes. In our example, nodes can have up to two children, and for that reason, this particular data structure is called a binary tree. But you could just as easily have trees with three, four or any number of children by modifying the data structure accordingly.

You can even have tree nodes that use linked lists to store all the nodes they point to. An important property of trees – both in real life and in data structures – is that there's a one-way path from roots to leaves. It'd be weird if roots connected to leaves that connected to roots.

For data that links arbitrarily, that include things like loops, we can use a graph data structure instead. Remember our graph from last

episode of cities connected by roads? This can be stored as nodes with many pointers, very much like a tree, but there is no notion of roots and leaves, and children and parents... Anything can point to anything!

So that's a whirlwind overview of pretty much all of the fundamental data structures used in computer science. On top of these basic building blocks, programmers have built all sorts of clever variants, with slightly different properties -- data structures like red-black trees and heaps, which we don't have time to cover. These different data structures have properties that are useful for particular computations.

The right choice of data structure can make your job a lot easier, so it pays off to think about how you want to structure your data before you jump in. Fortunately, most programming languages come with libraries packed full of ready-made data structures. For example, C++ has its Standard Template Library, and Java has the Java Class Library.

These mean programmers don't have to waste time implementing things from scratch, and can instead wield the power of data structures to do more interesting things, once again allowing us to operate at a new level of abstraction! I'll see you next week.