What's a Linked List, Anyway? [Part 1]

Vaidehi Joshi · Follow
Published in basecs · 9 min read · Jan 16, 2017

8.4K 25

Information is all around us.

In the world of software, the ways that we choose to organize our information is half the battle. Here's the thing though: there are *so many ways* to solve a problem. And when it comes to organizing our data, there are lots of tools that could work for the job. The trick is knowing which tool is the *right* one to use.

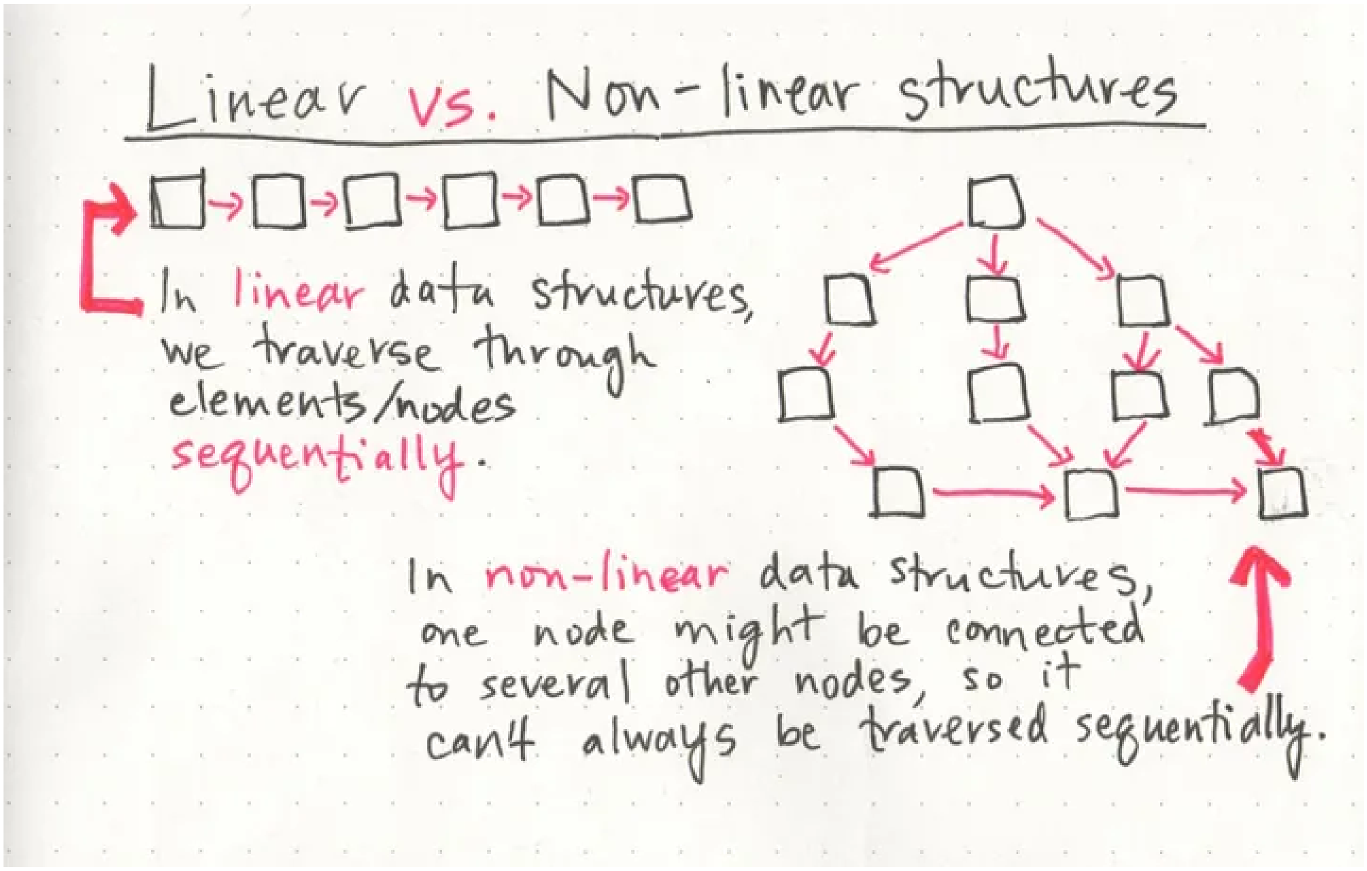
Regardless of which language we start coding in, one of the first things that we encounter are **data structures**, which are the different ways that we can organize our data; *variables, arrays, hashes,* and *objects* are all types of data structures. But these are still just the tip of the iceberg when it comes to data structures; there are a lot more, some of which start to sound super complicated the more that you hear about them.

One of those complicated things for me has always been **linked lists**. I've known about linked lists for a few years now, but I can never quite keep them straight in my head. I only really think about them when I'm preparing for (or sometimes, in the middle of) a technical interview, and someone asks me about them. I'll do a little research and think that I understand what they're about, but after a few weeks, I forget them again. The whole thing is pretty inefficient, and it all stems from the fact that I know they exist, but I don't fundamentally *understand* them! So, it's time to change that and answer the question: what on earth is a linked list, anyway?

Linear data structures

If we really want to understand the basics of linked lists, it's important that we talk about what *type* of data structure they are.

One characteristic of linked lists is that they are **linear data structures**, which means that there is a sequence and an order to how they are constructed and traversed. We can think of a linear data structure like a game of hopscotch: in order to get to the end of the list, we have to go through all of the items in the list in order, or *sequentially*. Linear structures, however, are the opposite of non-linear structures. In **non-linear data structures**, items don't have to be arranged in order, which means that we could traverse the data structure *non-sequentially*.


Linear vs. Non-linear structures
In linear data structures, we traverse through elements/nodes sequentially.
In non-linear data structures, one node might be connected to several other nodes, so it can't always be traversed sequentially.

Linear versus non-linear data structures

We might not always realize it, but we all work with linear and non-linear data structures everyday! When we organize our data into *hashes* (sometimes called *dictionaries*), we're implementing a non-linear data structure. *Trees* and *graphs* are also non-linear data structures that we traverse in different ways, but we'll talk more about them in more depth later in the year.

Similarly, when we use *arrays* in our code, we're implementing a linear data structure! It can be helpful to think of arrays and linked lists as being similar in the way that we sequence data. In both of these structures, ***order matters***. But what makes arrays and linked lists different?

Memory management

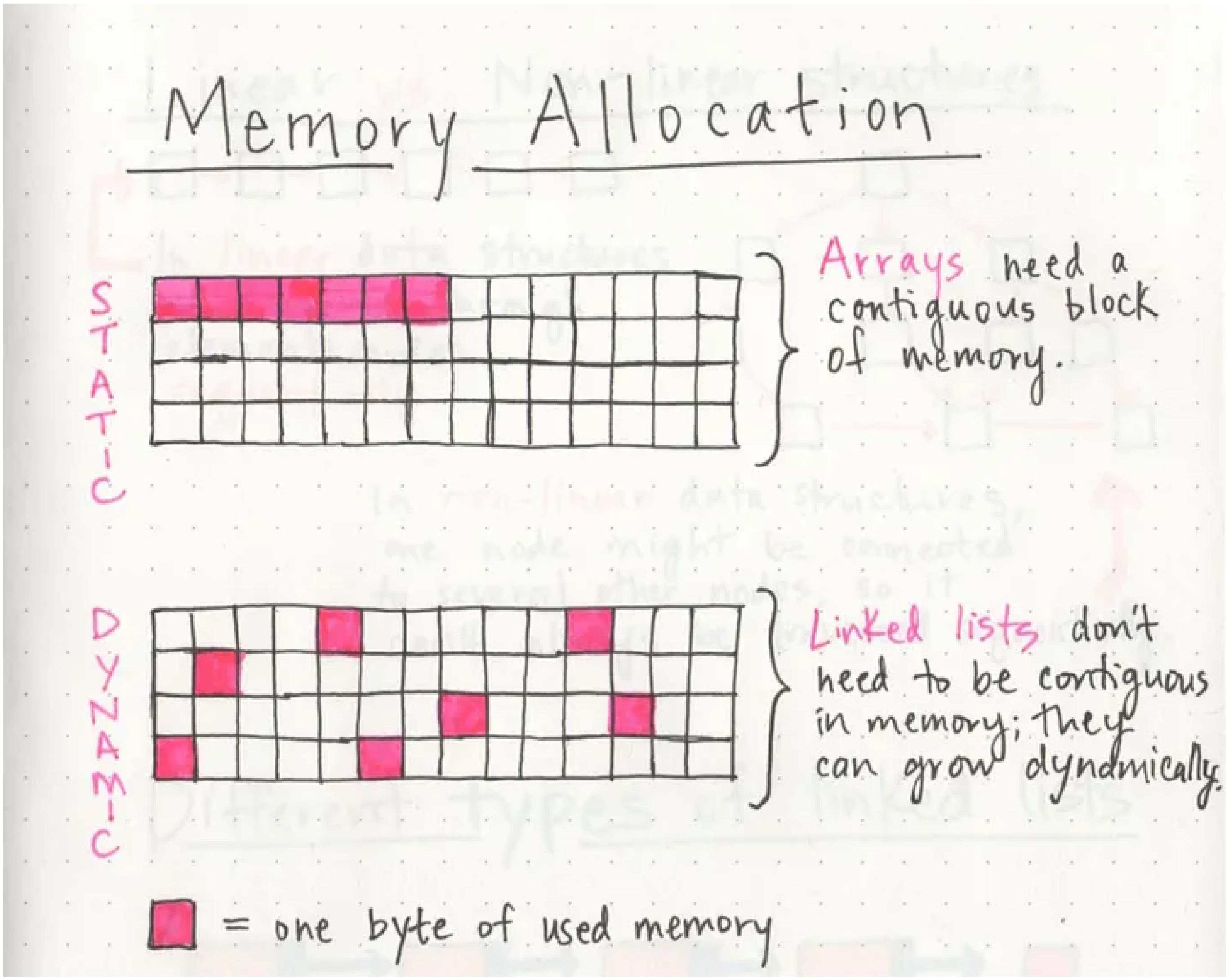
The biggest differentiator between arrays and linked lists is the way that they use memory in our machines. Those of us who work with dynamically typed languages like Ruby, JavaScript, or Python don't have to think about how much memory an array uses when we write our code on a day to day basis because there are several layers of abstraction that end up with us not having to worry about memory allocation at all.

But that doesn't mean that memory allocation isn't happening! Abstraction isn't magic, it's just the simplicity of hiding away things that you don't need to see or deal with all of the time. Even if we don't have to think about memory allocation when we write code, if we want to truly understand what's going on in a linked list and what makes it powerful, we have to get down to the rudimentary level.

We've already learned about binary and how data can be broken up into bits and bytes. Just as characters, numbers, words, sentences require bytes of memory to represent them, so do data structures.

When an **array** is created, it needs a certain amount of memory. If we had 7 letters that we needed to store in an array, we would need 7 bytes of memory to represent that array. But, we'd need all of that *memory in one contiguous block*. That is to say, our computer would need to locate 7 bytes of memory that was free, one byte next to the another, all together, in one place.

On the other hand, when a **linked list** is born, it doesn't need 7 bytes of memory all in one place. One byte could live somewhere, while the next byte could be stored in another place in memory altogether! Linked lists don't need to take up a single block of memory; instead, the memory that they use *can be scattered throughout*.


Memory Allocation
STATIC
Arrays need a contiguous block of memory.
DYNAMIC
Linked lists don't need to be contiguous in memory; they can grow dyndmically.
= one byte of used memory

Memory allocation in static versus dynamic data structures

The fundamental difference between arrays and linked lists is that arrays are **static data structures**, while linked lists are **dynamic data structures**. A static data structure needs all of its resources to be allocated when the structure is created; this means that even if the structure was to grow or shrink in size and elements were to be added or removed, it still *always needs a given size and amount of memory*. If more elements needed to be added to a static data structure and it didn't have enough memory, you'd need to copy the data of that array, for example, and recreate it with more memory, so that you could add elements to it.

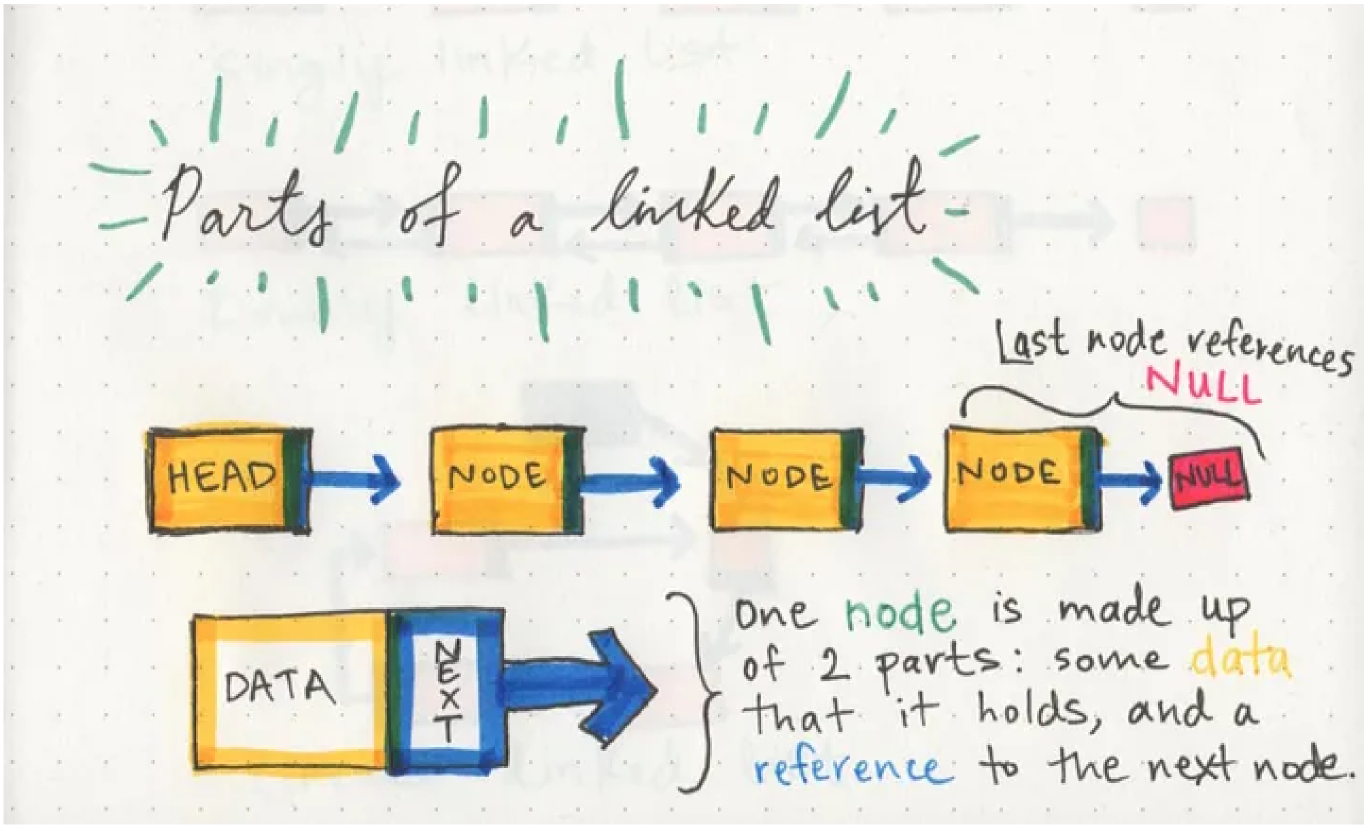
On the other hand, a dynamic data structure can shrink and grow in memory. It doesn’t need a set amount of memory to be allocated in order to exist, and its size and shape can change, and the amount of memory it needs can change as well.

By now, we can already begin to see some major differences between arrays and linked lists. But this begs the question: *what allows a linked list to have its memory scattered everywhere?* To answer this question, we need to look at the way that a linked list is structured.

Parts of a linked list

A linked list can be small or huge, but no matter the size, the parts that make it up are actually fairly simple. A linked list is made up of a series of **nodes**, which are the elements of the list.

The starting point of the list is a reference to the first node, which is referred to as the **head**. Nearly all linked lists must have a head, because this is effectively the only entry point to the list and all of its elements, and without it, you wouldn’t know where to start! The end of the list isn’t a node, but rather a node that points to **null**, or an empty value.


Parts of a linked list
Last node references NULL
HEAD
NODE
NODE
NODE
NULL
DATA
NEXT
One node is made up of 2 parts: some data that it holds, and a reference to the next node.

Parts of a linked list: it's all just a bunch of nodes, really.

A single node is also pretty simple. It has just two parts: **data**, or the information that the node contains, and a reference to the **next node**.

If we can wrap our heads around this, then we're halfway there. The way that nodes work is super important, and super powerful, and could be summarized as this:

> A node only knows about what data it contains, and who its neighbor is.

A single node doesn't know how long the linked list is, and it may not necessarily even know where it starts, or where it ends. All a node is concerned with is the data it contains, and which node its pointer references to — the next node in the list.

And this is the very reason *why* a linked list doesn't need a contiguous block of memory. Because a single node has the "address" or a reference to the next node, they don't need to live right next to one another, the way that the elements have to in an array. Instead, we can just rely on the fact that we can traverse our list by leaning on the pointer references to the next node, which means that our machines don't need to block off a single chunk of memory in order to represent our list.

This is also the explanation for why linked lists can grow and shrink dynamically during a program's execution. Adding or removing a node with a linked list becomes as simple as rearranging some pointers, rather than copying over the elements of an array! However, there are also some drawbacks to linked lists that I'm not mentioning to you just yet — but more on those next week.

For now, we'll just bask in the glory of how cool linked lists are!

Lists for all shapes and sizes

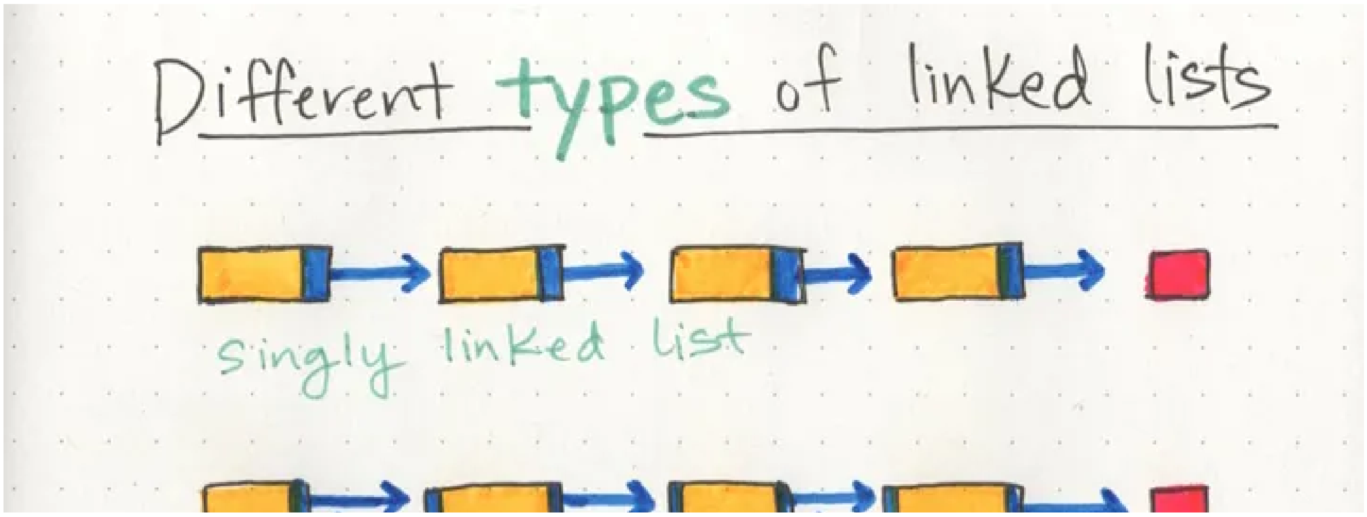
Even though the parts of a linked list don't change, the way that we structure our linked lists *can* be quite different. Like most things in software, depending on the problem that we're trying to solve, one type of linked lists might be a better tool for the job than another.

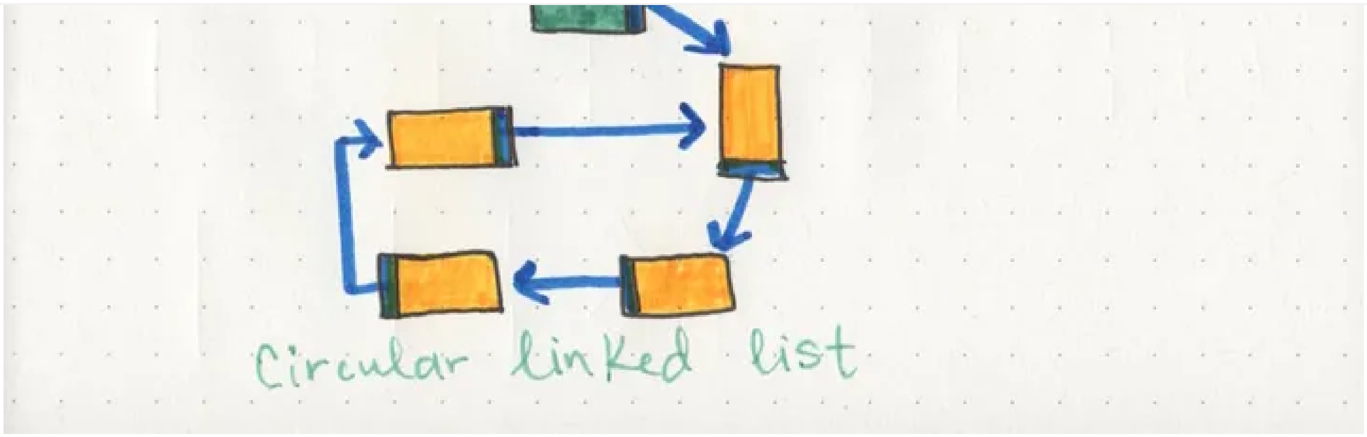
Singly linked lists are the simplest type of linked list, based solely on the fact that they only go in one direction. There is a **single track** that we can traverse the list in; we start at the **head** node, and traverse from the root until the **last** node, which will end at an empty **null** value.

But just as a node can reference its subsequent neighbor node, it can also have a reference pointer to its preceding node, too! This is what we call a

doubly linked list, because there are **two references** contained within each node: a reference to the **next** node, as well as the **previous** node. This can be helpful if we wanted to be able to traverse our data structure not just in a single track or direction, but also backwards, too.

For example, if we wanted to be able to hop between one node and the node previous without having to go back to the *very beginning of the list*, a doubly linked list would be a better data structure than a singly linked list. However, everything requires space and memory, so if our node had to store two reference pointers instead of just one, that would be another thing to consider.





Different types of linked lists

A ***circular linked list*** is a little odd in that it doesn't end with a node pointing to a null value. Instead, it has a node that acts as the *tail* of the list (rather than the conventional head node), and the node after the tail node is the beginning of the list. This organization structure makes it really easy to add something to the end of the list, because you can begin traversing it at the **tail** node, as the first element and last element point to one another. Circular linked lists can start to get really crazy because we can turn both a singly linked list and a doubly linked list *into* a circular linked list!

But no matter how complicated a linked list is, if we can remember the fundamentals of a node and how it works and how the different pointer references in our list are structured, there's no linked list we can't tackle!

Next week, in part 2 of this series, we'll sink our teeth into the space time complexity of linked lists, and how they compare to their cousin, the array. I promise that it's actually a lot more fun than it sounds!

Resources

If you think linked lists are super cool, check out these helpful resources.

1. Differences between Arrays and Linked Lists, Damien Wintour
2. Data Structures: Arrays vs Linked Lists, mycodeschool
3. Linked Lists: The Basics, Dr. Edward Gehringer
4. Introduction to Linked Lists, Dr. Victor Adamchik
5. Data Structures & Implementations, Dr. Jennifer Welch